Python标准库

虽然Python语言参考描述了Python语言的确切语法和语义，但该库参考手册描述了随Python分发的标准库。它还介绍了Python发行版中通常包含的一些可选组件。

Python的标准库非常广泛，提供了一系列广泛的工具，如下面列出的长表所示。该库包含内置模块（用C语言编写），可以访问系统功能，例如Python程序员无法访问的文件I / O，以及用Python编写的模块，这些模块为出现的许多问题提供标准化的解决方案日常编程。其中一些模块明确地设计为通过将特定于平台的特性抽象为与平台无关的API来鼓励和增强Python程序的可移植性。

Windows平台的Python安装程序通常包含整个标准库，并且通常还包含许多其他组件。对于类Unix操作系统，Python通常以包的集合形式提供，因此可能需要使用随操作系统提供的打包工具来获取部分或全部可选组件。

除了标准库之外，还有越来越多的数千个组件（从单独的程序和模块到包和整个应用程序开发框架）可以从Python包索引中获得。

1.介绍

“Python库”包含几种不同类型的组件。

它包含通常被认为是语言“核心”的一部分的数据类型，例如数字和列表。对于这些类型，Python语言核心定义了文字的形式并对其语义进行了一些约束，但是没有完全定义语义。（另一方面，语言核心确实定义了句法属性，如运算符的拼写和优先级。）

该库还包含内置的函数和异常 - 可以被所有Python代码使用的对象，而不需要`import`语句。其中一些是由核心语言定义的，但是其中很多不是核心语义所必需的，这里只介绍它们。

然而，图书馆的大部分是由一系列模块组成的。解析这个集合有很多方法。有些模块是用C语言编写的，并内置在Python解释器中; 其他人用Python编写并以源代码形式导入。一些模块提供了非常特定于Python的接口，如打印堆栈跟踪; 一些提供特定于特定操作系统的接口，例如访问特定硬件; 其他人则提供特定于特定应用程序域的接口，如万维网。一些模块可用于Python的所有版本和端口; 其他只有在底层系统支持或需要它们时才可用; 还有一些只有在编译和安装Python时选择了特定的配置选项才可用。

本手册“从内到外”组织起来：它首先描述了内置函数，数据类型和异常，最后描述了相关模块章节中分类的模块。

这意味着，如果您从一开始就阅读本手册，并在感到无聊时跳到下一章，您将会对Python库支持的可用模块和应用领域有一个合理的概述。当然，你不要*有像*小说一样读它-你也可以浏览目录（在手册的前面），或者寻找在指数特定的功能，模块或长期（在后面）。最后，如果你喜欢学习随机主题，你可以选择一个随机页码（参见模块 `random`）并阅读一两节内容。无论您阅读本手册各章的顺序如何，它都有助于从章节 内置函数开始，因为本手册的其余部分假定您熟悉本资料。

让节目开始！

2.内置函数

Python解释器内置了许多功能和类型，它们始终可用。它们按字母顺序排列在这里。

内置函数

abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

abs (*x*)

返回一个数字的绝对值。参数可以是整数或浮点数。如果参数是一个复数，则返回其大小。

all (*可迭代*)

返回True如果的所有元素*迭代*是真实的（或者如果可迭代为空）。相当于：

```
def all(iterable):
    for element in iterable:
        if not element:
            return False
    return True
```

any (*可迭代*)

True如果*迭代的*任何元素为真，则返回。如果迭代器为空，则返回False。相当于：

```
def any(iterable):
    for element in iterable:
        if element:
            return True
    return False
```

ascii (*object*)

如repr()，由返回的字符串中返回一个包含对象的可打印表示一个字符串，但逃避非ASCII字符 repr()使用\x，\u或\U逃逸。这会生成一个类似于repr()Python 2 中返回的字符

串。

bin (*x*)

将整数转换为以“0b”为前缀的二进制字符串。结果是一个有效的Python表达式。如果*x*不是Python `int`对象，则必须定义一个`__index__()`返回整数的方法。一些例子：

```
>>> bin(3)
'0b11'
>>> bin(-10)
'-0b1010'
```

如果需要前缀“0b”，则可以使用以下任一方式。

```
>>> format(14, '#b'), format(14, 'b')
('0b1110', '1110')
>>> f'{14:#b}', f'{14:b}'
('0b1110', '1110')
```

另见`format()`更多信息。

class bool ([*x*])

返回一个布尔值，即一个`True`或`False`。 *x*使用标准真相测试程序进行转换。如果*x*为假或省略，则返回`False`; 否则返回`True`。的 `bool`类是的子类`int`（参见数值类型-整数，浮点，复合）。它不能进一步分类。它唯一的例子是`False`和 `True`（参见布尔值）。

class bytearray ([*source* [, *encoding* [, *errors*]]])

返回一个新的字节数组。的`bytearray`类是整数的范围为0 <= X一个可变的序列<256它具有最的可变序列，在所描述的常规方法的可变序列类型，以及大多数方法，该`bytes`类型已见字节和ByteArray操作。

可选的*source*参数可以用几种不同的方式初始化数组：

- 如果它是一个*字符串*，则还必须给出*编码*（以及可选的 *错误*）参数; `bytearray()`然后使用字符串转换为字节`str.encode()`。
- 如果它是一个*整数*，则数组将具有该大小，并将用空字节进行初始化。
- 如果它是符合*缓冲区*接口的对象，则将使用该对象的只读缓冲区来初始化字节数组。
- 如果它是*可迭代的*，它必须是范围内的整数的迭代 ，它们被用作数组的初始内容。`0 <= x < 256`

没有参数，就会创建一个大小为0的数组。

另请参见二进制序列类型 - 字节，bytearray，memoryview和Bytearray对象。

class bytes ([*source* [, *encoding* [, *errors*]]])

返回一个新的“字节”对象，它是范围内的一个不可变的整数序列。 是一个不可变的版本 - 它具有相同的非变异方法和相同的索引和切片行为。`0 <= x < 256`bytesbytearray

因此，构造函数参数被解释为`bytearray()`。

字节对象也可以使用文字创建，请参阅字符串和字节文字。

另请参见二进制序列类型 - 字节，字节阵列，内存视图，字节对象以及字节和字节阵列操作。

callable (*object*)

True如果*对象*参数显示为可调用，False则返回， 如果不是。如果这返回true，那么调用失败仍然是可能的，但如果它是false，调用*对象*将永远不会成功。请注意，类是可调用的（调用一个类返回一个新的实例）; 如果它们的类有一个__call__()方法，则实例可以被调用。

*3.2版本中的新*功能：此功能在Python 3.0中首先被删除，然后在Python 3.2中被带回。

chr (*i*)

返回表示Unicode代码点为整数*i*的字符的字符串。例如，chr(97)返回字符串'a'，同时chr(8364)返回字符串'€'。这是与之相反的ord()。

参数的有效范围是从0到1,114,111（基于16的0x10FFFF）。 ValueError如果*我*在这个范围之外，会被提高。

@classmethod

将方法转换为类方法。

类方法将类作为隐式第一个参数接收，就像实例方法接收实例一样。要声明一个类方法，使用这个习惯用法：

```
class C:
    @classmethod
    def f(cls, arg1, arg2, ...): ...
```

该@classmethod表单是一个函数装饰器 - 有关详细信息，请参阅函数定义中的函数定义说明。

它可以在类（如C.f()）或实例（如C().f()）上调用。该实例被忽略，除了它的类。如果为派生类调用类方法，则派生类对象将作为隐含的第一个参数传递。

类方法与C ++或Java静态方法不同。如果你想要这些，请参阅staticmethod()本节。

有关类方法的更多信息，请参阅标准类型层次结构中关于标准类型层次结构的文档。

compile (*source*，*filename*，*mode*，*flags = 0*，*dont_inherit = False*，*optimize = -1*)

将*源*编译为代码或AST对象。代码对象可以由exec()或执行eval()。 *源*可以是普通字符串，字节字符串或AST对象。ast有关如何使用AST对象的信息，请参阅模块文档。

该*文件名*参数应该给从代码读取的文件; 如果没有从文件中读取（'<string>'通常使用），则传递一些可识别的值。

该*模式*参数指定什么样的代码必须进行编译; 它可以是'exec'，如果*源*包含的语句序列的，'eval'如果它由一个单一的表达的，或者'single'如果它由单个交互式声明的（在后一种情况下，计算结果为比其他东西表达式语句None将被打印）。

可选参数*flags*和*dont_inherit*控制哪些将来的语句（请参阅**PEP 236**）影响*源*的编译。如果两者都不存在（或两者均为零），则代码将与正在调用的代码中生效的那些未来语句一起编译`compile()`。如果给出了 *flags*参数，并且 *dont_inherit*不是（或者是零），那么除了那些将被使用的参数之外，将使用 *flags*参数指定的将来语句。如果 *dont_inherit*是一个非零整数，那么 *flags*参数就是它 - 将忽略围绕调用编译的未来声明。

将来的语句由可以按位或运算来指定多个语句的位指定。指定给定特征所需的位域可以作为模块中实例的`compiler_flag`属性找到。`_Feature__future__`

参数*optimize*指定编译器的优化级别; 默认值`-1`选择由`-O`选项给出的解释器的优化级别。显式级别是`0`（没有优化; `__debug__`是），`1`（断言被删除，`__debug__`是假的）或`2`（文档字符串也被删除）。

`SyntaxError`如果编译的源无效，并且`ValueError`源包含空字节，则此函数会引发。

如果您想将Python代码解析为其AST表示形式，请参阅 `ast.parse()`。

> **注意：** 使用多行代码输入`'single'`或`'eval'`模式编译字符串时，输入必须至少由一个换行符终止。这是为了便于检测`code`模块中不完整和完整的语句。

> **警告：** 由于Python AST编译器中的堆栈深度限制，编译为AST对象时，可能会使Python解释器崩溃为足够大/复杂的字符串。

*在版本3.2中更改：*允许使用Windows和Mac换行符。在`'exec'`模式下输入也不必以换行符结束。添加了*优化*参数。

*在版本3.5中更改：*以前，`TypeError`在*源*中遇到空字节时引发。

class complex（[*real*[，*imag*]]）

返回值为*real* + *imag* * 1j的复数或者将字符串或数字转换为复数。如果第一个参数是一个字符串，它将被解释为一个复数，并且该函数必须在没有第二个参数的情况下被调用。第二个参数不能是一个字符串。每个参数可以是任何数字类型（包括复数）。如果*IMAG*被省略，默认为零，并且构造用作数字转换等 `int`和`float`。如果两个参数都被省略，则返回`0j`。

> **注意：** 从字符串转换时，该字符串不得在中央`+`或`-`运算符周围包含空格。例如，`complex('1+2j')`很好，但引发。`complex('1 + 2j')` `ValueError`

复数类型在数字类型 - int，float，complex中描述。

*在版本3.6中更改：*允许使用下划线对代码进行分组，如代码文字。

delattr（*object*，*name*）

这是一个相对的`setattr()`。参数是一个对象和一个字符串。该字符串必须是对象属性之一的名称。该函数删除指定的属性，只要该对象允许。例如，相当于。`delattr(x, 'foobar')` `del x.foobar`

*班*dict（***kwarg*）
*类*dict（*映射*，***kwarg*）

*类*dict（*可迭代的*，** *kwarg* ）

创建一个新的字典。该dict对象是字典类。请参阅dict和映射类型 - dict有关此类的文档。

对于其他容器看到内置list ， set以及 tuple类，还有collections模块。

dir（[*object*]）

如果没有参数，则返回当前本地作用域中的名称列表。使用参数尝试返回该对象的有效属性列表。

如果该对象有一个名为的方法__dir__() ，则该方法将被调用并且必须返回属性列表。这允许实现自定义__getattr__()或__getattribute__()功能的对象自 定义dir()报告其属性的方式 。

如果对象没有提供__dir__() ，函数会尽最大努力从对象的__dict__属性（如果已定义）和其类型对象中收集信息。结果列表不一定完整，并且在对象具有自定义时可能不准确__getattr__()。

默认dir()机制对不同类型的对象的行为不同，因为它试图产生最相关的信息，而不是完整的信息：

- 如果对象是模块对象，则列表包含模块属性的名称。
- 如果对象是一个类型或类对象，则该列表包含其属性的名称，并递归地显示其基础的属性。
- 否则，该列表包含对象的属性名称，其类属性的名称以及其类的基类的属性的递归。

结果列表按字母顺序排序。例如：

```
>>> import struct
>>> dir()   # show the names in the module namespace
['__builtins__', '__name__', 'struct']
>>> dir(struct)   # show the names in the struct module
['Struct', '__all__', '__builtins__', '__cached__', '__doc__', '__file__',
 '__initializing__', '__loader__', '__name__', '__package__',
 '_clearcache', 'calcsize', 'error', 'pack', 'pack_into',
 'unpack', 'unpack_from']
>>> class Shape:
...     def __dir__(self):
...         return ['area', 'perimeter', 'location']
>>> s = Shape()
>>> dir(s)
['area', 'location', 'perimeter']
```

注意： 由于dir()主要是为了便于在交互式提示下使用而提供，因此它会尝试提供一组有趣的名称，而不是试图提供严格或一致定义的名称集，并且其详细行为可能会在各版本之间发生变化。例如，当参数是一个类时，元类属性不在结果列表中。

divmod（*a* , *b* ）

以两个（非复数）数字作为参数，并在使用整数除法时返回由它们的商和余数组成的一对数字。使用混合操作数类型时，适用二元算术运算符的规则。对于整数，结果与之相同。对于浮点数的结果是，其中*q*通常是但可能比1小1。在任何情况下都非常接近*a*，如果非

零，则它与*b*具有相同的符号，并且。(a // b, a % b)(q, a % b)math.floor(a / b)q * b + a % ba % b0 <= abs(a % b) < abs(b)

enumerate (*iterable* , *start = 0*)

返回一个枚举对象。*iterable*必须是一个序列，一个 迭代器或其他支持迭代的对象。__next__()通过enumerate()返回的迭代器的方法 返回一个包含count的元组（从*start开始*，默认值为0）以及从*iterable迭代*获得的值。

```
>>> seasons = ['Spring', 'Summer', 'Fall', 'Winter']
>>> list(enumerate(seasons))
[(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]
>>> list(enumerate(seasons, start=1))
[(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]
```

相当于：

```
def enumerate(sequence, start=0):
    n = start
    for elem in sequence:
        yield n, elem
        n += 1
```

eval (*表达式* , *globals = None* , *locals = None*)

参数是一个字符串和可选的全局变量和局部变量。如果提供， *全局变量*必须是字典。如果提供，*本地人*可以是任何映射对象。

该*表达*参数分析，并作为一个Python表达式来使用（从技术上来说，一个条件列表）*全局*和*当地人*的字典作为全局和局部名字空间。如果*globals*字典存在并且缺少'__builtins__'，则 在解析*表达式*之前将当前全局变量复制到*全局变量中*。这意味着*表达式*通常可以完全访问标准模块，并且传播受限制的环境。如果省略了*本地*字典，则默认为*全局变量*builtins字典。如果两个字典都被省略，表达式会在eval()调用的环境中执行。返回值是评估表达式的结果。语法错误被报告为例外。例：

```
>>> x = 1
>>> eval('x+1')
2
```

这个函数也可以用来执行任意代码对象（比如那些创建的对象compile() ）。在这种情况下，传递一个代码对象而不是一个字符串。如果代码对象已经被编译'exec'为 *模式*参数，那么eval()返回值将是None。

提示：该exec() 函数支持动态执行语句。的globals()和locals()功能返回当前全球和本地词典，分别，其可以是通过以绕过使用有用eval()或exec()。

请参阅有关ast.literal_eval()可以安全地使用仅包含文字的表达式评估字符串的函数。

exec (*object* [, *globals* [, *locals*]])

这个函数支持Python代码的动态执行。*对象*必须是字符串或代码对象。如果它是一个字符串，则将该字符串解析为一组Python语句，然后执行该语句（除非发生语法错误）。[1]如果它是一个代码对象，它只是被执行。在任何情况下，执行的代码都应该作为文件输入有

效（请参见参考手册中的“文件输入”部分）。请注意， 即使在传递给函数的代码的上下文中，也不能在函数定义之外使用return和yield语句 exec()。返回值是None。

在所有情况下，如果省略可选部件，则代码将在当前范围内执行。如果只提供*全局变量*，则它必须是一个字典，它将用于全局变量和局部变量。如果*全局*和 *当地人*给出，它们分别用于全局和局部变量。如果提供，*本地人*可以是任何映射对象。请记住，在模块级别，全局变量和本地变量是相同的字典。如果exec获取两个单独的对象作为*全局变量*和*本地变量*，那么代码将被嵌入类定义中执行。

如果*globals*字典不包含该键的值，则 __builtins__将该内置模块的字典引用 builtins插入该键的下方。通过这种方式，您可以通过在将自己的__builtins__字典传递给*全局变量*之前，将其自己的字典插入到*可执行*代码中来控制可用的内置 *变量*exec()。

注意: 内置的函数globals()并分别locals()返回当前的全局和本地字典，这可能对传递用作第二个和第三个参数非常有用exec()。

注意: 默认*本地人的*行为如locals()下面的功能所述：不应尝试对默认*本地人*字典的修改。传递一个明确的*当地人*解释，如果你需要看到的代码的作用*当地人*后的功能exec()恢复。

filter(*函数*, *可迭代*)

从构造的那些元件的迭代器*可迭代*为哪些*函数*返回真。 *可迭代*可以是序列，支持迭代的容器或迭代器。如果*函数*是None，则假定标识函数，即删除所有*可迭代的*元素。

注意，如果函数不是且函数是 ，则等价于生成器表达式。filter(function, iterable)(item for item in iterable if function(item))None(item for item in iterable if item)None

请参阅itertools.filterfalse()用于返回哪个*函数*返回false 的*可迭代*元素的补充函数。

class float([*x*])

返回一个由数字或字符串*x*构造的浮点数。

如果参数是一个字符串，它应该包含一个十进制数字，可选地以一个符号开头，并且可以嵌入空格。可选标志可以是'+'或'-'；一个'+'标志对所产生的价值没有影响。该参数也可以是表示NaN（非数字）或正或负无穷大的字符串。更确切地说，在删除前后空白字符后，输入必须符合以下语法：

```
sign           ::= “+” | “-”
 infinity       ::= “Infinity” | “inf”
 nan            ::= “nan”
 numeric_value ::=   floatnumber | infinity | numeric_string ::= [ ]nan
signnumeric_value
```

这floatnumber是浮点文字中描述的Python浮点文字的形式。情况并不重要，因此，例如，“inf”，“Inf”，“INFINITY”和“iNfINity”对于正无穷大都是可接受的拼写。

否则，如果参数是整数或浮点数，则返回具有相同值（在Python的浮点精度内）的浮点数。如果参数超出了Python浮点的范围，OverflowError则会引发。

对于一般的Python对象x，float(x)委托给 x.__float__()。

如果没有提供参数，0.0则返回。

例子：

```
>>> float('+1.23')
1.23
>>> float('   -12345\n')
-12345.0
>>> float('1e-003')
0.001
>>> float('+1E6')
1000000.0
>>> float('-Infinity')
-inf
```

浮点类型用数字类型 - int，float，complex来描述。

*在版本3.6中更改：*允许使用下划线对代码进行分组，如代码文字。

format (*value* [, *format_spec*])

将*值*转换为“格式化”表示，由*format_spec*控制 。*format_spec*的解释将取决于*value*参数的类型，但是存在大多数内置类型使用的标准格式化语法：Format Specification Mini-Language。

*format_spec*的默认值是一个空字符串，通常与调用效果相同str(value)。

调用将被转换为 在搜索该值的方法时绕过实例字典。一 ，如果该方法搜索到达引发异常和*format_spec*非空，或者这两个 *format_spec*或返回值是不是字符串。 format(value, format_spec) type(value).__format__(value, format_spec) __format__() TypeError object

在版本3.4中更改： 如果*format_spec*不是空字符串，则object().__format__(format_spec) 引发。 TypeError

*类*frozenset ([*iterable*])

返回一个新的frozenset对象，可选的元素来自 *iterable*。 frozenset是一个内置的类。请参阅frozenset和 设置类型 - 设置，冻结关于此类的文档。

对于其他容器看到内置的set ， list ， tuple ，和dict类，还有collections 模块。

getattr (*object* , *name* [, *default*])

返回*对象*的指定属性的*值*。 *名称*必须是字符串。如果字符串是对象属性之一的名称，则结果是该属性的值。例如，相当于 。如果指定的属性不存在，则返回*默认值*（如果提供），否则返回。 getattr(x, 'foobar') x.foobar AttributeError

globals ( )

返回表示当前全局符号表的字典。这总是当前模块的字典（在函数或方法内部，这是定义它的模块，而不是调用它的模块）。

hasattr (*object* , *name*)

参数是一个对象和一个字符串。结果是True如果字符串是对象属性之一的名称，False如果不是。（这是通过调用并观察它是否引发一个实现的。）getattr(object, name)AttributeError

hash（*object*）

返回对象的散列值（如果有）。哈希值是整数。它们用于在字典查找期间快速比较字典键。比较相等的数值具有相同的散列值（即使它们具有不同的类型，就像1和1.0一样）。

注意： 对于具有自定义__hash__()方法的对象，请注意hash()根据主机的位宽截断返回值。详情请参阅__hash__()。

help（[*object*]）

调用内置的帮助系统。（此功能用于交互式使用。）如果未提供参数，则交互式帮助系统将在解释器控制台上启动。如果参数是一个字符串，那么该字符串将被查找为模块，函数，类，方法，关键字或文档主题的名称，并在控制台上打印帮助页面。如果参数是任何其他类型的对象，则会生成对象上的帮助页面。

该功能被模块添加到内置命名空间中site。

*在版本3.4中进行了*更改：更改pydoc并inspect意味着可报告的已报告签名现在更加全面和一致。

hex（*x*）

将整数转换为以“0x”为前缀的小写十六进制字符串。如果*x*不是Python int对象，则必须定义一个__index__()返回整数的 方法。一些例子：

```
>>> hex(255)
'0xff'
>>> hex(-42)
'-0x2a'
```

如果要将整数转换为带有前缀或不带前缀的大写或小写十六进制字符串，可以使用以下任一方式：

```
>>> '%#x' % 255, '%x' % 255, '%X' % 255
('0xff', 'ff', 'FF')
>>> format(255, '#x'), format(255, 'x'), format(255, 'X')
('0xff', 'ff', 'FF')
>>> f'{255:#x}', f'{255:x}', f'{255:X}'
('0xff', 'ff', 'FF')
```

另见format()更多信息。

另请参阅int()使用16的基数将十六进制字符串转换为整数。

注意： 要获取浮点数的十六进制字符串表示形式，请使用该 float.hex()方法。

id（*object*）

返回一个对象的“身份”。这是一个整数，它在其生命周期中保证对这个对象唯一且恒定。两个非重叠生命期的对象可能具有相同的id()值。

CPython实现细节：这是内存中对象的地址。

input([*提示*])

如果*提示*参数存在，则将其写入标准输出而没有尾随换行符。然后该函数从输入中读取一行，将其转换为一个字符串（剥离尾随的换行符），然后返回该行。当EOF被读取时，EOFError被提出。例：

```
>>> s = input('--> ')
--> Monty Python's Flying Circus
>>> s
"Monty Python's Flying Circus"
```

如果readline模块已加载，那么input()将使用它来提供精细的线条编辑和历史记录功能。

class int(*x=0*)

class int(*x*, *base=10*)

返回由数字或字符串*x*构造的整数对象，0如果没有给定参数，则返回 。如果*x*定义__int__()，则int(x)返回x.__int__()。如果*x*定义__trunc__()，它返回x.__trunc__()。对于浮点数，这将截断为零。

如果*X*不是数字或如果*基*给出，则*X*必须是一个字符串， bytes或bytearray代表一个实例字面整数在基数*基*。可选地，文字可以在+或之前-（没有间隔）并且被空白包围。base-n文字由数字0到n-1组成，其中a to z（或Ato Z）的值为10到35.默认*基数*为10.允许的值为0和2-36。base-2，-8和-16文字可以有选择地以0b/ 0B， 0o/ 0O或0x/0X与代码中的整数文字一样。基0意味着精确地解释为文字代码，使得实际碱是2，8，10，或16，并且使得不合法，而 是，以及。int('010', 0) int('010') int('010', 8)

整数类型在数值类型 - int，float，complex中描述。

*在版本3.4中进行了更改：*如果*base*不是实例，int并且*基础*对象具有 base.__index__方法，则会调用该方法以获取基础的整数。以前的版本用来 base.__int__代替base.__index__。

*在版本3.6中更改：*允许使用下划线对代码进行分组，如代码文字。

isinstance(*object*, *classinfo*)

如果*对象*参数是*classinfo* 参数的实例或其（直接，间接或虚拟）子类的实例，则返回true。如果*object*不是给定类型的对象，则该函数总是返回false。如果*classinfo*是类型对象的元组（或者是递归的，其他这样的元组），如果*object*是任何类型的实例，则返回true 。如果*classinfo*不是类型和元组的类型或元组，TypeError则会引发异常。

issubclass(*class*, *classinfo*)

如果返回true *类*是一个子类（直接，间接或虚拟）的*CLASSINFO*。一个类被认为是它自己的一个子类。*classinfo*可以是类对象的元组，在这种情况下，将检查*classinfo中的*每个条目。在其他情况下，TypeError会引发异常。

iter (*object* [, *sentinel*])

返回一个迭代器对象。根据第二个参数的存在，第一个参数被解释得非常不同。如果没有第二个参数，*对象*必须是支持迭代协议（ `__iter__()` 方法）的集合对象，或者它必须支持序列协议（ `__getitem__()` 整数参数始于的方法0 ）。如果它不支持这些协议中的任何一个， `TypeError`则会引发。如果给出了第二个参数*sentinel*，那么*object*必须是可调用的对象。在这种情况下创建的迭代器将为每个对其方法的调用调用没有参数的 *对象* `__next__()` ; 如果返回的值等于 *哨兵*，`StopIteration` 将被提出，否则该值将被退回。

另请参阅迭代器类型。

第二种形式的一个有用的应用`iter()`是读取文件的行，直到达到某一行。以下示例读取文件，直到该`readline()`方法返回空字符串：

```
with open('mydata.txt') as fp:
    for line in iter(fp.readline, ''):
        process_line(line)
```

len (*s*)

返回对象的长度（项目数量）。参数可以是一个序列（如字符串，字节，元组，列表或范围）或集合（如字典，集合或冻结集合）。

*类*list ([*iterable*])

而不是一个函数，`list`实际上是一个可变的序列类型，如列表和序列类型 - 列表，元组，范围所记录。

locals ( )

更新并返回表示当前本地符号表的字典。`locals()`在功能块中调用自由变量时会返回自由变量，但不会在类块中调用它们。

> **注意：** 这本词典的内容不应该修改; 更改可能不会影响解释器使用的本地变量和自由变量的值。

map (*function* , *iterable* , ...)

返回一个将*函数*应用于每个*迭代*项的*迭代器*，从而产生结果。如果传递额外的*迭代*参数，*函数*必须采用多个参数并应用于并行所有迭代中的项目。使用多个迭代器时，当最短迭代器耗尽时，迭代器停止。对于函数输入已经安排到参数元组中的情况，请参见`itertools.starmap()`。

max (*可迭代* , *** [, *key* , *default*])

max (*arg1* , *arg2* , ** args* [, *key*])

返回iterable中的最大项或两个或更多个参数中最大的项。

如果提供了一个位置参数，它应该是一个可迭代的。迭代中最大的项目被返回。如果提供了两个或多个位置参数，则返回最大的位置参数。

有两个可选的仅关键字参数。该*关键*参数指定一个参数的排序功能，类似于用于`list.sort()`。该 *默认*参数指定的对象返回如果提供的迭代是空的。如果迭代器为空并且未提供*缺省值*， `ValueError`则会引发a。

如果多个项目最大，则该函数返回遇到的第一个项目。这与其他类型稳定保存工具（如 and ）一致 。 sorted(iterable, key=keyfunc, reverse=True)[0]heapq.nlargest(1, iterable, key=keyfunc)

*在新版本3.4：在默认情况下*只有关键字的说法。

memoryview (*obj*)

返回从给定参数创建的“内存视图”对象。请参阅 内存视图了解更多信息。

min (*可迭代* , ***[, *key* , *default*])

min (*arg1* , *arg2* , ** args* [, *key*])

返回可迭代中的最小项或两个或更多个参数中的最小项。

如果提供了一个位置参数，它应该是一个可迭代的。迭代中的最小项返回。如果提供两个或多个位置参数，则返回最小的位置参数。

有两个可选的仅关键字参数。该*关键*参数指定一个参数的排序功能，类似于用于 list.sort()。该 *默认*参数指定的对象返回如果提供的迭代是空的。如果迭代器为空并且未提供*缺省值*， ValueError则会引发a。

如果多个项目最小，则该函数返回遇到的第一个项目。这与其他类型稳定保存工具（如 and ）一致 。 sorted(iterable, key=keyfunc)[0]heapq.nsmallest(1, iterable, key=keyfunc)

*在新版本3.4：在默认情况下*只有关键字的说法。

next (*iterator* [, *default*])

通过调用它的 方法从*迭代器中*检索下一个项目__next__()。如果给出了*默认值*，则在迭代器耗尽时返回，否则返回StopIteration。

*类*object

返回一个新的无特征的对象。 object是所有课程的基础。它具有所有Python类实例通用的方法。这个函数不接受任何参数。

> **注意：** object没有*不*有__dict__，所以你不能指定任意属性的实例object类。

oct (*x*)

将整数转换为以“0o”为前缀的八进制字符串。结果是一个有效的Python表达式。如果*x*不是Python int对象，则必须定义一个__index__()返回整数的方法。例如：

```
>>> oct(8)
'0o10'
>>> oct(-56)
'-0o70'
```

如果要将整数转换为八进制字符串，前缀为“0o”，则可以使用以下任一方式。

```
>>> '%#o' % 10, '%o' % 10
('0o12', '12')
```

```
>>> format(10, '#o'), format(10, 'o')
('0o12', '12')
>>> f'{10:#o}', f'{10:o}'
('0o12', '12')
```

另见format()更多信息。

open(*file*，*mode='r'*，*buffering=-1*，*encoding=None*，*errors=None*，*newline=None*，*closefd=True*，*opener=None*)

打开*文件*并返回相应的文件对象。如果文件无法打开，OSError则会提出。

*文件*是一个类似路径的对象，它提供要打开的文件的路径名（绝对或相对于当前工作目录）或要打包的文件的整数文件描述符。（如果给出文件描述符，则在返回的I / O对象关闭时关闭，除非*closefd* 设置为False。）

*mode*是一个可选字符串，用于指定打开文件的模式。它默认'r'打开文本模式下的阅读方式。其他常见的值是'w'写入（如果文件已经存在，则截断文件），'x'独占创建和'a'附加（在*某些*Unix系统上，这意味着*所有*写入追加到文件末尾，而不管当前搜索位置如何）。在文本模式下，如果*编码*未指定使用的编码是与平台相关的：locale.getpreferredencoding(False)被称为获取当前的本地编码。（用于读取和写入原始字节使用二进制模式并且 未指定*编码*。）可用的模式有：

字符	含义
'r'	打开阅读（默认）
'w'	打开写入，首先截断文件
'x'	打开独占创建，如果文件已经存在则失败
'a'	打开写入，追加到文件末尾（如果存在）
'b'	二进制模式
't'	文本模式（默认）
'+'	打开磁盘文件进行更新（读取和写入）
'U'	通用换行符模式（已弃用）

默认模式是'r'（打开阅读文本，同义词'rt'）。对于二进制读写访问，该模式'w+b'打开并将文件截断为0字节。'r+b'不截断打开文件。

如概述中所述，Python区分二进制和文本I / O。以二进制模式打开的文件（包括'b'在*模式*参数中）以bytes没有任何解码的对象的形式返回内容。在文本模式下（默认情况下，或者't'包含在*mode*参数中时），文件的内容将返回为str：字节首先使用平台相关编码进行解码，或者使用指定的 *编码*（如果给出）。

注意： Python不依赖于底层操作系统的文本文件概念; 所有的处理都由Python自己完成，因此是平台无关的。

*缓冲*是用于设置缓冲策略的可选整数。通过0切换缓冲关闭（仅在二进制模式下允许），1选择行缓冲（仅在文本模式下可用）以及> 1的整数，以指示固定大小的块缓冲区的大小（以字节为单位）。如果未给出*缓冲*参数，则默认缓冲策略的工作方式如下所示：

- 二进制文件以固定大小的块进行缓冲; 缓冲区的大小是通过试图确定底层设备的“块大小”并重新开始的启发式来选择的io.DEFAULT_BUFFER_SIZE。在很多系统上，缓冲区的长度通常为4096或8192字节。
- “交互式”文本文件（ isatty() 返回的文件True ）使用行缓冲。其他文本文件使用上述的二进制文件策略。

*编码*是用于解码或编码文件的编码的名称。这只能用于文本模式。默认编码是平台相关的（无论locale.getpreferredencoding()返回），但可以使用Python支持的任何 文本编码。请参阅codecs模块以获取支持的编码列表。

*错误*是一个可选字符串，指定如何处理编码和解码错误 - 这不能用于二进制模式。有多种标准错误处理程序可用（列在“ 错误处理程序”下 ），但已注册的任何错误处理名称codecs.register_error()也是有效的。标准名称包括：

- 'strict' ValueError如果存在编码错误，则引发异常。默认值None具有相同的效果。
- 'ignore' 忽略错误。请注意，忽略编码错误可能会导致数据丢失。
- 'replace' 导致替换标记（例如'?'）被插入有错误数据的地方。
- 'surrogateescape' 会将任何不正确的字节表示为Unicode专用区域中的代码点，范围从U + DC80到U + DCFF。 surrogateescape当写入数据时使用错误处理程序时，这些私有代码点将被转回相同的字节。这对处理未知编码中的文件很有用。
- 'xmlcharrefreplace' 仅在写入文件时才受支持。编码不支持的字符将替换为适当的XML字符引用&#nnn;。
- 'backslashreplace' 用Python的反斜杠转义序列替换畸形数据。
- 'namereplace'（也仅在写入时才支持）用换\N{...}码序列替换不支持的字符。

*换行符*控制通用换行符模式的工作方式（仅适用于文本模式）。它可以是None，''，'\n'，'\r'，和'\r\n'。它的工作原理如下：

- 当从流中读取输入时，如果*换行符*是None，则启用通用换行符模式。输入中的行可以以'\n'，，'\r'或结束'\r\n'，并且'\n'在返回给调用者之前将这些行翻译成。如果是''，则启用通用换行符模式，但行结束符将返回给调用方未翻译。如果它具有任何其他合法值，则输入行仅由给定字符串终止，并且行尾以未翻译形式返回给调用者。
- 将输出写入流时，如果*换行符*是None，则'\n'写入的任何字符都将转换为系统默认行分隔符 os.linesep。如果*换行符*是''或'\n'，则不会进行翻译。如果*换行符*是任何其他合法值，则'\n'写入的任何字符都将转换为给定的字符串。

如果*closefd*是False文件描述符而不是文件名，那么文件关闭时底层文件描述符将保持打开状态。如果给定文件名，*closefd*必须是True（默认），否则会引发错误。

可以通过传递可调用的*开罐器*来使用自定义*开罐器*。然后通过调用*opener*（*file*，*flags*）来获得文件对象的底层文件描述符。*开*os.open叫*器*必须返回一个打开的文件描述符（ 作为*开*叫*者的*结果传递类似于传递的功能 None ）。

新创建的文件是不可继承的。

以下示例使用该函数的dir_fd参数 os.open() 来打开相对于给定目录的文件：

```
>>> import os
>>> dir_fd = os.open('somedir', os.O_RDONLY)
>>> def opener(path, flags):
...     return os.open(path, flags, dir_fd=dir_fd)
...
>>> with open('spamspam.txt', 'w', opener=opener) as f:
```

```
...     print('This will be written to somedir/spamspam.txt', file=f)
...
>>> os.close(dir_fd)  # don't leak a file descriptor
```

函数返回的文件对象的类型open()取决于模式。当open()使用在文本模式（打开一个文件'w'，'r'，'wt'，'rt'，等等），它返回的一个子类 io.TextIOBase（具体而言io.TextIOWrapper）。当用缓冲区以二进制模式打开文件时，返回的类是其子类io.BufferedIOBase。确切的类有所不同：在读取二进制模式下，它返回一个io.BufferedReader;在写二进制和附加二进制模式下，它返回一个io.BufferedWriter，并在读/写模式下，它返回一个io.BufferedRandom。当缓冲被禁用，原始流，的一个子类io.RawIOBase， io.FileIO，则返回。

参见文件处理模块，如，fileinput，io （其中open()被声明），，os，os.path，tempfile和shutil。

在版本3.3中更改：

- 在*揭幕战*中添加参数。
- 该'x'模式已添加。
- IOError曾经被提出，它现在是一个别名OSError。
- FileExistsError如果在独占创建模式（'x'）中打开的文件已经存在，现在会产生。

在版本3.4中更改：

- 该文件现在是不可继承的。

*自版本3.4起弃用，将在版本4.0中删除：*该'U'模式。

在版本3.5中更改：

- 如果系统调用中断并且信号处理程序不引发异常，则此函数现在重试系统调用而不是引发 InterruptedError异常（请参阅**PEP 475**为理由）。
- 在'namereplace'加入错误处理程序。

在版本3.6中更改：

- 添加支持以接受实现的对象os.PathLike。
- 在Windows上，打开控制台缓冲区可能会返回io.RawIOBase除以外的子类io.FileIO。

ord(*c*)

给定一个表示一个Unicode字符的字符串，返回一个表示该字符的Unicode代码点的整数。例如，ord('a')返回整数97和ord('€')（欧元符号）返回8364。这是与之相反的chr()。

pow(*x*, *y*[, *z*])

将*x*返回给*y*；如果*z*存在，则将*x*返回给*y*，模*z*（比其更有效地计算）。双参数形式相当于使用权力运算符：。pow(x, y) % zpow(x, y)x**y

参数必须有数字类型。对于混合操作数类型，适用于二元算术运算符的强制规则。对于int操作数，除非第二个参数为负数，否则结果与操作数的类型相同（强制后）。在这种情况下，所有参数都将转换为浮点数并传递浮点结果。例如，10**2 返回100，但10**-2返回

0.01。如果第二个参数是负数，则必须省略第三个参数。如果*z*存在，则*x*和*y* 必须是整数类型，并且*y*必须是非负数。

print (**objects*，*sep =''*，*end ='\n'*，*file = sys.stdout*，*flush = False*)

将*对象*打印到文本流*文件中*，以*sep*分隔，*然后以end结尾*。 *必须将sep* ， *end* ， *file*和*flush*（如果存在）作为关键字参数给出。

所有非关键字参数都会转换为字符串`str()`，并写入流中，然后由*sep*分隔并*结尾*。无论*九月*和*年底*必须是字符串; 他们也可以`None`，这意味着使用默认值。如果没有任何*对象*，`print()`只会写 *结束*。

的*文件*参数必须是与对象`write(string)`方法; 如果它不存在或`None`，`sys.stdout`将被使用。由于打印参数转换为文本字符串，`print()`因此不能与二进制模式文件对象一起使用。对于这些，请`file.write(...)`改用。

输出是否缓冲通常由*文件*决定，但如果 *flush*关键字参数为true，则强制刷新流。

*版本3.3中更改：*添加了*flush*关键字参数。

class property (*fget = None*，*fset = None*，*fdel = None*，*doc = None*)

返回一个属性属性。

*fget*是获取属性值的函数。 *fset*是用于设置属性值的函数。*fdel*是删除属性值的功能。然后*doc*为该属性创建一个文档字符串。

一个典型的用途是定义一个托管属性`x`：

```
class C:
    def __init__(self):
        self._x = None

    def getx(self):
        return self._x

    def setx(self, value):
        self._x = value

    def delx(self):
        del self._x

    x = property(getx, setx, delx, "I'm the 'x' property.")
```

如果*c*是*C的*一个实例，`c.x`将调用getter， 将调用setter和deleter。`c.x = value`del `c.x`

如果给定，*doc*将是属性属性的文档字符串。否则，该属性将复制*fget*的文档字符串（如果存在）。这使得它能够很容易地创建只读属性使用`property()`作为装饰：

```
class Parrot:
    def __init__(self):
        self._voltage = 100000

    @property
    def voltage(self):
```

```
        """Get the current voltage."""
        return self._voltage
```

该@property装饰变成的voltage()方法变成“吸”为只读具有相同名称的属性，并将其设置的文档字符串的 *电压*为“获取当前的电压。”

属性对象具有getter，setter和deleter可用作装饰器的方法，这些装饰器用相应的存取器函数设置为装饰函数来创建属性的副本。这可以用一个例子来解释：

```
class C:
    def __init__(self):
        self._x = None

    @property
    def x(self):
        """I'm the 'x' property."""
        return self._x

    @x.setter
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x
```

此代码与第一个示例完全等效。一定要赋予与原始属性同名的附加功能（ x在这种情况下）。

返回的属性对象也有属性fget，fset以及 fdel相对应的构造函数的参数。

*版本3.5中已更改：*属性对象的文档字符串现在可写入。

range（*停止*）
range（*开始*，*停止*[，*步骤*]）
而不是一个函数，range实际上是一个不可变的序列类型，如范围和序列类型 - 列表，元组，范围所记录。

repr（*object*）
返回一个包含对象可打印表示的字符串。对于许多类型，此函数尝试返回一个字符串，该字符串在传递时会产生具有相同值的对象eval()，否则该表示是一个用尖括号括起来的字符串，其中包含对象类型的名称以及附加信息通常包括对象的名称和地址。一个类可以通过定义一个__repr__()方法来控制这个函数为其实例返回的内容。

reversed（*seq*）
返回一个反向迭代器。 *seq*必须是一个具有__reversed__()方法或支持序列协议的对象（__len__()方法和__getitem__()整数参数始于的方法0）。

round（*number*[，*ndigits*]）
返回*数字*在小数点后舍入到精度*ndigits*。如果*ndigits*被忽略或是None，它返回最接近的整数到它的输入。

对于内置类型支持round()，值将四舍五入为功率减去*ndigits*的10 *倍数*；如果两个倍数同样接近，舍入朝向甚至选择完成（因此，例如，这两个round(0.5)和round(-0.5)是0，和round(1.5)是2）。任何整数值对*ndigits*有效（正数，零或负数）。如果*ndigits*被省略或者返回值是一个整数None。否则，返回值与*数字的*类型相同。

对于一般的Python对象number，round委托给 number.__round__。

> **注意：** round()对于浮动的行为可能会令人惊讶：例如， 给出而不是预期的。这不是一个错误：这是由于大多数小数不能完全表示为浮点数的结果。有关更多信息，请参阅浮点算术：问题和限制。 round(2.675, 2)2.672.68

*类*set（[*iterable*]）

返回一个新的set对象，可选的元素来自 *iterable*。 set是一个内置的类。请参阅set和 设置类型 - 设置，冻结关于此类的文档。

对于其他容器看到内置的frozenset，list， tuple，和dict类，还有collections 模块。

setattr（*对象*，*名称*，*值*）

这是与之相对的getattr()。参数是一个对象，一个字符串和一个任意值。该字符串可以命名现有的属性或新的属性。如果该对象允许，该函数将该值分配给该属性。例如，相当于。 setattr(x, 'foobar', 123)x.foobar = 123

class slice（*stop* ）

*类*slice（*开始*，*停止*[，*步骤*] ）

返回表示由指定的索引集的切片对象 。在*启动*和*步*参数默认为 。切片对象具有只读数据属性， 并且仅返回参数值（或其默认值）。他们没有其他明确的功能；然而，它们被Numerical Python和其他第三方扩展使用。当使用扩展索引语法时，也会生成切片对象。例如：或。查看 返回迭代器的备用版本。 range(start, stop, step)Nonestartstopstepa[start:stop:step]a[start:stop, i]itertools.islice()

sorted（*iterable*，*，*key* = *None*，*reverse* = *False* ）

从*迭代中*的项目中返回一个新的排序列表。

有两个可选参数，必须将其指定为关键字参数。

*key*指定一个用于从每个列表元素中提取比较键的参数的函数：key=str.lower。默认值是None （直接比较元素）。

*reverse*是一个布尔值。如果设置为True，则列表元素按照每个比较被颠倒的顺序进行排序。

用functools.cmp_to_key()一个老式的转换*CMP*功能的 *关键*功能。

内置sorted()功能保证稳定。如果确保不会更改比较相等的元素的相对顺序，则排序是稳定的 - 这对于多次排序（例如，按部门排序，然后按薪级）进行排序很有帮助。

有关排序示例和简要的排序教程，请参阅对如何排序。

@staticmethod

将方法转换为静态方法。

静态方法不会收到隐式的第一个参数。要声明一个静态方法，使用这个习惯用法：

```
class C:
    @staticmethod
    def f(arg1, arg2, ...): ...
```

该@staticmethod表单是一个函数装饰器 - 有关详细信息，请参阅函数定义中的函数定义说明。

它可以在类（如C.f()）或实例（如C().f()）上调用。该实例被忽略，除了它的类。

Python中的静态方法类似于Java或C ++中的方法。另请参阅 classmethod()有关创建备用类构造函数的变体。

像所有的装饰器一样，它也可以staticmethod作为一个常规函数调用，并对其结果进行处理。在需要从类体中引用函数的某些情况下，需要这样做，并且您希望避免自动转换为实例方法。对于这些情况，请使用以下习惯用法：

```
class C:
    builtin_open = staticmethod(open)
```

有关静态方法的更多信息，请参阅标准类型层次结构中标准类型层次结构的文档。

class str（*object =''*）
class str（*object = b''*，*encoding ='utf-8'*，*errors ='strict'*）

返回一个*对象*的str版本。详情请参阅。 str()

str是内置的字符串类。有关字符串的一般信息，请参阅文本序列类型 - str。

sum（*iterable*[，*start*]）

资金*开始*和的项目*迭代*由左到右，并返回总。 *开始*默认为0。该*迭代*的项目通常是数字，起始值不允许是一个字符串。

对于一些使用情况，有很好的选择sum()。连接字符串序列的首选方法是通过调用''.join(sequence)。要以扩展精度添加浮点值，请参阅math.fsum()。要连接一系列迭代，请考虑使用 itertools.chain()。

super（[*type*[，*object-or-type*]]）

返回一个代理对象，委托方法调用到父母或兄弟姐妹类的*类型*。这对于访问在类中被覆盖的继承方法很有用。 getattr()除了*类型*本身被跳过之外，搜索顺序与使用的顺序相同 。

该*类型*的__mro__属性列出了两者和使用的方法解析搜索顺序。该属性是动态的，只要继承层次更新就可以更改。 getattr() super()

如果省略第二个参数，则返回的超级对象是未绑定的。如果第二个参数是一个对象，则必须为真。如果第二个参数是一个类型，则必须为true（这对于类方法很有用）。 isinstance(obj, type) issubclass(type2, type)

*super*有两种典型的用例。在具有单一继承的类层次结构中，*super*可以用于引用父类而不显式命名它们，从而使代码更易于维护。这种用法与其他编程语言中*超级*用法非常类似。

第二个用例是在动态执行环境中支持协作式多重继承。这个用例是Python独有的，在静态编译的语言或仅支持单一继承的语言中找不到。这使得在多个基类实现相同方法的情况下实现“菱形图”成为可能。良好的设计规定，这种方法在每种情况下都具有相同的调用签名（因为调用顺序是在运行时确定的，因为该顺序适用于类层次结构中的更改，并且因为该顺序可以包含运行时未知的同级类）。

对于这两种用例，典型的超类调用如下所示：

```
class C(B):
    def method(self, arg):
        super().method(arg)    # This does the same thing as:
                               # super(C, self).method(arg)
```

请注意，它`super()`是作为显式点状属性查找的绑定过程的一部分实现的，例如`super().__getitem__(name)`。它通过实现自己的`__getattribute__()`方法以可预测的顺序搜索类，支持协作多重继承。因此，`super()`对于使用语句或运算符（如，）的隐式查找，未定义`super()[name]`。

还要注意，除了零参数形式外，`super()`不限于使用内部方法。这两个参数形式完全指定了参数，并提供了适当的参考。零参数表单只能在类定义中使用，因为编译器会填充必要的细节以正确检索正在定义的类，以及访问普通方法的当前实例。

有关如何使用设计协作类的实用建议`super()`，请参阅使用super（）的指南。

tuple（[*可迭代*]）

而不是一个函数，`tuple`实际上是一个不可变的序列类型，如元组和序列类型 - 列表，元组，范围中记载的那样。

class type（*object*）

*类*type（*名称*，*基地*，*字典*）

使用一个参数，返回一个*对象*的类型。返回值是一个类型对象，通常与返回的对象相同`object.__class__`。

`isinstance()`建议使用内置函数来测试对象的类型，因为它考虑了子类。

有三个参数，返回一个新的类型对象。这实质上是`class`声明的一种动态形式。该*名字*符串类名，并成为`__name__`属性; 所述*碱*元组逐条列出的基类和成为`__bases__`属性; 和*字典*的字典是包含用于类定义体的命名空间和被复制到标准词典成为`__dict__`属性。例如，以下两条语句创建相同的`type`对象：

```
>>> class X:
...     a = 1
...
>>> X = type('X', (object,), dict(a=1))
```

另请参见键入对象。

在版本3.6中更改： type其中不覆盖的子类type.__new__可能不再使用单参数形式来获取对象的类型。

vars（[*object*]）

使用__dict__属性返回模块，类，实例或任何其他对象的__dict__属性。

诸如模块和实例之类的对象具有可更新__dict__ 属性; 但是，其他对象可能会对其__dict__属性进行写入限制（例如，类使用a types.MappingProxyType来防止直接更新字典）。

没有争论，vars()就像locals()。请注意，本地字典仅用于读取，因为本地字典的更新被忽略。

zip（ **iterables* ）

制作一个迭代器，用于聚合来自每个迭代器的元素。

返回元组的迭代器，其中第*i*个元组包含来自每个参数序列或迭代的第*i*个元素。当最短的输入迭代耗尽时，迭代器停止。使用单个迭代参数，它将返回1元组的迭代器。没有参数，它返回一个空的迭代器。相当于：

```
def zip(*iterables):
    # zip('ABCD', 'xy') --> Ax By
    sentinel = object()
    iterators = [iter(it) for it in iterables]
    while iterators:
        result = []
        for it in iterators:
            elem = next(it, sentinel)
            if elem is sentinel:
                return
            result.append(elem)
        yield tuple(result)
```

可保证迭代的从左到右的评估顺序。这使得使用一个将数据序列聚类为n长度组的成语成为可能zip(*[iter(s)]*n)。这将重复*相同的*迭代器n时间，以便每个输出元组都有n调用迭代器的结果。这具有将输入划分为n长度块的效果。

zip()只有当您不关心较长迭代中的尾随，不匹配值时才应使用不等长输入。如果这些值很重要，请itertools.zip_longest()改用它。

zip()与*运营商一起可以用来解压一个列表：

```
>>> x = [1, 2, 3]
>>> y = [4, 5, 6]
>>> zipped = zip(x, y)
>>> list(zipped)
[(1, 4), (2, 5), (3, 6)]
>>> x2, y2 = zip(*zip(x, y))
>>> x == list(x2) and y == list(y2)
True
```

__import__（ *名称*，*globals* = *None*，*locals* = *None*，*fromlist* = *（ ）*，*level* = *0* ）

注意： 与日常Python编程不同，这是一种高级功能importlib.import_module()。

该函数由import语句调用。它可以被替换（通过导入builtins模块并分配给builtins.__import__）以改变import语句的语义，但是**强烈**建议不要这样做，因为使用导入钩子通常更简单（请参阅**PEP 302**）来实现相同的目标，并且不会导致假定默认导入实施正在使用的代码出现问题。直接使用__import__()也是不鼓励的importlib.import_module()。

该函数导入模块*名称*，可能使用给定的*全局变量*和*局部变量*来确定如何解释包上下文中的名称。在*fromlist*里给出了应该从给出的模块导入的对象或子的名字*命名*。标准实现完全不使用它的*本地*参数，并且仅使用它的*全局变量*来确定import语句的包上下文。

*级别*指定是使用绝对导入还是相对导入。0（默认）意味着只执行绝对导入。*级别的*正值表示要相对于模块调用目录搜索的父目录的数量__import__()（请参阅**PEP 328**的细节）。

当*name*变量具有这种形式时package.module，通常会返回顶层包（名称直到第一个点），*而不是*按*名称*命名的模块。但是，如果给出非空的*fromlist*参数，则会返回按*名称*命名的模块。

例如，该语句导致类似于以下代码的字节码：import spam

```
spam = __import__('spam', globals(), locals(), [], 0)
```

声明结果在这个调用中：import spam.ham

```
spam = __import__('spam.ham', globals(), locals(), [], 0)
```

请注意，__import__()这里返回顶层模块是因为这是通过import语句绑定到名称的对象。

另一方面，声明结果from spam.ham import eggs, sausage as saus

```
_temp = __import__('spam.ham', globals(), locals(), ['eggs', 'sausage'], 0)
eggs = _temp.eggs
saus = _temp.sausage
```

在这里，spam.ham模块从中返回__import__()。从这个对象中，要导入的名称被检索并分配给它们各自的名称。

如果您只是想按名称导入模块（可能位于软件包内），请使用importlib.import_module()。

*在版本3.3中更改：*不再支持*级别的*负值（它也将默认值更改为0）。

脚注

[1] 使用换行解释器以外的Windows以外的行结束约定时，如果您正在阅读文件中的代码，请确保

3.内置常量

少量的常量存在于内置的命名空间中。他们是：

False
: bool类型的错误值。分配False 是非法的，并提出一个SyntaxError。

True
: bool类型的真正价值。分配True 是非法的，并提出一个SyntaxError。

None
: 类型的唯一值NoneType。 None经常用来表示没有值，因为默认参数没有传递给函数。分配None是非法的，并提出一个SyntaxError。

NotImplemented
: 这应该由二进制特殊的方法被返回（如特殊的值__eq__()，__lt__()，__add__()，__rsub__()，等等），以指示该操作不相对于另一种类型的实施; 可通过就地二进制特殊的方法（例如被返回__imul__()，__iand__()为了相同的目的，等等）。它的真实价值是真实的。

  注意: 当二进制（或in-place）方法返回NotImplemented解释器时，将尝试对另一个类型（或其他某个后备，取决于操作符）的反射操作。如果所有尝试都返回NotImplemented，则解释器会提出适当的例外。错误地返回NotImplemented将导致错误的错误消息或NotImplemented返回到Python代码的值。
  有关示例，请参阅实现算术运算。

  注意: NotImplementedError和NotImplemented不能互换，即使他们有相似的名称和用途。请参阅NotImplementedError有关何时使用它的详细信息。

Ellipsis
: 和...。一样。特别值与用户定义的容器数据类型的扩展切片语法结合使用。

__debug__
: 如果Python没有使用-O选项启动，这个常量是正确的。另见assert声明。

注意: 名称None，False，True并且__debug__ 不能被重新分配（分配给他们，甚至作为一个属性名称，提高 SyntaxError），因此它们可以被认为是“真正的”常数。

3.1。site模块添加的常量

该site模块（在启动过程中自动导入，除非-S给出了命令行选项）将几个常量添加到内置命名空间。它们对交互式解释器外壳非常有用，不应在程序中使用。

quit(*code = None*)

exit（*code* = *None* ）

当打印时，打印一条消息，如“使用退出（ ）或Ctrl-D（即EOF）退出”，并在调用时SystemExit使用指定的退出代码进行提升。

copyright
credits

在打印或调用时分别打印版权或信用文本的对象。

license

当打印时，打印消息“输入license（ ）以查看完整的许可证文本”，当被调用时，以类似于寻呼机的方式（一次一个屏幕）显示完整的许可证文本。

4.内置类型

以下部分描述了内置到解释器中的标准类型。

主要的内置类型是数字，序列，映射，类，实例和异常。

一些集合类是可变的。那些添加，减少或重新排列其成员的方法，并且不返回特定项目，但永远不会返回集合实例本身None。

一些操作由几种对象类型支持; 特别是，实际上所有对象都可以进行比较，测试真值，并将其转换为字符串（使用repr()函数或稍微不同的 str()函数）。当函数写入对象时，隐式使用后者print()函数。

4.1。真值测试

任何对象都可以测试真值，以用于下面的布尔运算的a if或 while条件或操作数。

默认情况下，一个对象被认为是真的，除非它的类定义了一个 __bool__()返回False的__len__()方法或者一个方法返回零，当被对象调用时。[1] 以下是大多数被认为是错误的内置对象：

- 定义为false的常量：None和False。
- 任何数值类型的零：0，0.0，0j，Decimal(0)， Fraction(0, 1)
- 空序列和集合：''，()，[]，{}，set()， range(0)

除非另有说明，否则具有布尔结果的操作和内置函数总是返回0 或False为false 1或True为true。（重要的例外：布尔操作or并and始终返回其操作数之一。）

4.2。布尔运算- ，，and or not

这些是布尔操作，按优先级升序排列：

手术	结果	笔记
x or y	如果x是假的，那么y，否则 x	（1）
x and y	如果x是假的，那么x，否则 y	（2）
not x	如果x是假的，那么True，否则False	（3）

笔记：

1. 这是一个短路运算符，因此如果第一个参数为假，它只会计算第二个参数。
2. 这是一个短路操作符，所以如果第一个参数为真，它只会计算第二个参数。
3. not具有比非布尔运算符更低的优先级，因此被解释为，并且是语法错误。not a == bnot (a == b) a == not b

4.3。比较

Python中有八个比较操作。它们都具有相同的优先级（这比布尔操作的优先级高）。比较可以任意链接; 例如等同于，除了y只被评估一次（但是在两种情况下，当z被发现为假时根本不被评估）。x < y <= z x < y and y <= z x < y

本表总结了比较操作：

手术	含义

`<`	严格小于
`<=`	小于或等于
`>`	严格大于
`>=`	大于或等于
`==`	等于
`!=`	不等于
`is`	对象身份
`is not`	否定对象身份

除了不同的数字类型，不同类型的对象永远不会相等。此外，某些类型（例如函数对象）仅支持简单的比较概念，即任何两种类型的对象都不相等。的`<`，`<=`，`>`和`>=`运营商将提出一个`TypeError`比较复杂的数字时异常另一个内置数字型，当对象是不同的类型，而且不能比拟的，或在其他情况下，没有定义排序的。

除非类定义`__eq__()`方法，否则通常将类的不同实例比较为不相等。

一个类的实例不能相对于同一类的其他实例，或其他类型的对象进行排序，除非类定义了足够的方法`__lt__()`，`__le__()`，`__gt__()`，和`__ge__()`（在一般情况下，`__lt__()`并`__eq__()`有足够的，如果你想要的传统含义比较运算符）。

`is`和运营商的行为不能定制; 它们也可以应用于任何两个对象，并且不会引发异常。`is not`

具有相同优先级的语法两个操作，`in`并且，只能通过序列类型（下）的支持。`not in`

4.4。数字类型- ，，`int` `float` `complex`

有三种不同的数字类型：*整数*，*浮点数*和*复数*。另外，布尔是一个整数的子类型。整数具有无限精度。浮点数通常`double`在C中实现; 有关运行程序的机器的浮点数的精度和内部表示的信息可在`sys.float_info`。复数有一个实部和虚部，每个部分都是一个浮点数。要从复数*z*中提取这些零件，请使用`z.real`和`z.imag`。（标准库包含额外的数字类型，`fractions`用于保存有理数据和`decimal`这些浮点数用用户可定义的精度保存。）

数字由数字文字或内置函数和运算符的结果创建。未经修饰的整数文字（包括十六进制，八进制和二进制数字）将产生整数。包含小数点或指数符号的数字文字会生成浮点数字。追加'`j`'或'`J`'数字文字会产生一个虚数（一个具有零实数部分的复数），您可以将其添加到整数或浮点数以获得具有实部和虚部的复数。

Python完全支持混合算术：当二进制算术运算符的操作数具有不同的数值类型时，“窄”类型的操作数会扩展到另一个操作数，其中整数比浮点窄，比浮点窄。混合类型的数字之间的比较使用相同的规则。[2]的构造`int()`，`float()`以及`complex()`可以用来产生特定类型的号码。

所有数字类型（除复数）都支持以下操作，按优先级升序排列（所有数字操作的优先级均高于比较操作）：

手术	结果	笔记	完整的文档
`x + y`	*x*和*y*之和		
`x - y`	*x*和*y*的差异		
`x * y`	*x*和*y*的乘积		
`x / y`	*x*和*y*的商数		

手术	结果	笔记	先进的类似
x // y	x和 y的商数	（1）	
x % y	剩下的 x / y	（2）	
-x	x否定		
+x	x不变		
abs(x)	绝对值或x的大小		abs()
int(x)	x转换为整数	（3）	int()
float(x)	x转换为浮点	（4）	float()
complex(re, im)	复数与实部 重，虚部IM。 我默认为零	（6）	complex()
c.conjugate()	复数c的共轭		
divmod(x, y)	这对 (x // y, x % y)	（2）	divmod()
pow(x, y)	x到权力y	（5）	pow()
x ** y	x到权力y	（5）	

笔记：

1. 也被称为整数除法。结果值是一个整数，虽然结果的类型不一定是int。结果总是朝着负无穷的方向变化：1//2是0，(-1)//2是 -1，1//(-2)是-1，(-1)//(-2)是0。
2. 不适用于复杂的数字。相反，转换为浮动使用abs()。
3. 从C中浮点到整数的转换可能会舍入或截断; 查看功能math.floor()和math.ceil()定义明确的转换。
4. 对于非数字（NaN）和正或负无穷，float还接受带有可选前缀“+”或“ - ”的字符串“nan”和“inf”。
5. Python定义和将会像编程语言一样常见。pow(0, 0)0 ** 01
6. 接受的数字文字包括数字0到9或任何Unicode当量（与代码点Nd属性）。

   请参阅http://www.unicode.org/Public/9.0.0/ucd/extracted/DerivedNumericType.txt 以获取具有该Nd属性的代码点的完整列表。

所有numbers.Real类型（int和float）还包括以下操作：

手术	结果
math.trunc(x)	x截断为Integral
round(x[, n])	x四舍五入为n位数，四舍五入为偶数。如果省略n，则默认为0。
math.floor(x)	最大的Integral <= x
math.ceil(x)	至少Integral> = x

有关其他数字操作，请参阅math和cmath 模块。

4.4.1。整数类型的按位运算

按位运算只对整数有意义。负数被视为它们的2的补码值（假定有足够的位数以便在操作期间不发生溢出）。

二进制按位运算的优先级均低于数值运算并高于比较; 一元运算~与其他一元运算（+和-）具有相同的优先级。

此表列出按升序优先级排序的按位运算：

手术	结果	笔记
x | y	按位*或*的*x*和 *y*	
x ^ y	按位*异或*的 *x*和*y*	
x & y	按位*和*的*x*和 *y*	
x << n	*x*左移*n*位	(1)(2)
x >> n	*x*右移*n*位	(1)(3)
~x	*x*的位反转	

笔记：

1. 负转移次数是非法的，并导致a `ValueError`被提出。
2. 左移*n*位相当于 没有溢出检查的乘法。`pow(2, n)`
3. 右移*n*位相当于没有溢出检查的除法。`pow(2, n)`

4.4.2。整数类型的其他方法

int类型实现抽象基类。另外，它还提供了更多的方法：`numbers.Integral`

`int.`**`bit_length`**`( )`

返回表示二进制整数所需的位数，不包括符号和前导零：

```
>>> n = -37
>>> bin(n)
'-0b100101'
>>> n.bit_length()
6
```

更确切地说，如果`x`不为零，则`x.bit_length()`是唯一的正整数`k`这样。等价地，当小到足以得到正确的舍入对数时，则。如果是零，则返回。`2**(k-1) <= abs(x) < 2**k``abs(x)``k = 1 + int(log(abs(x), 2))``x``x.bit_length()``0`

相当于：

```
def bit_length(self):
    s = bin(self)       # binary representation:  bin(-37) --> '-0b100101'
    s = s.lstrip('-0b') # remove leading zeros and minus sign
    return len(s)       # len('100101') --> 6
```

版本3.1中的新功能。

`int.`**`to_bytes`**`(` *长度*`,` *字节顺序*`, *, signed = False )`

返回表示整数的字节数组。

```
>>> (1024).to_bytes(2, byteorder='big')
b'\x04\x00'
>>> (1024).to_bytes(10, byteorder='big')
b'\x00\x00\x00\x00\x00\x00\x00\x00\x04\x00'
>>> (-1024).to_bytes(10, byteorder='big', signed=True)
b'\xff\xff\xff\xff\xff\xff\xff\xff\xfc\x00'
>>> x = 1000
>>> x.to_bytes((x.bit_length() + 7) // 8, byteorder='little')
b'\xe8\x03'
```

整数用*长度*字节表示。OverflowError 如果整数不能用给定的字节数来表示，则会引发An 。

的*字节顺序*参数确定用于表示整数的字节顺序。如果字节*序*是"big"，则最重要的字节在字节数组的开始处。如果字节*序*是"little"，则最重要的字节在字节数组的末尾。要请求主机系统的本地字节顺序，请使用sys.byteorder字节顺序值。

带*符号的*参数确定是否使用二进制补码表示整数。如果*signed*是False且给出了一个负整数，则会产生一个OverflowError。*signed*的默认值 是False。

3.2版本中的新功能

classmethod int.**from_bytes**(*bytes*, *byteorder*, *, *signed = False*)

返回给定字节数组所表示的整数。

```
>>> int.from_bytes(b'\x00\x10', byteorder='big')
16
>>> int.from_bytes(b'\x00\x10', byteorder='little')
4096
>>> int.from_bytes(b'\xfc\x00', byteorder='big', signed=True)
-1024
>>> int.from_bytes(b'\xfc\x00', byteorder='big', signed=False)
64512
>>> int.from_bytes([255, 0, 0], byteorder='big')
16711680
```

参数*字节*必须是类似字节的对象或可迭代的生成字节。

的*字节顺序*参数确定用于表示整数的字节顺序。如果字节*序*是"big"，则最重要的字节在字节数组的开始处。如果字节*序*是"little"，则最重要的字节在字节数组的末尾。要请求主机系统的本地字节顺序，请使用sys.byteorder字节顺序值。

带*符号的*参数指示是否使用二进制补码表示整数。

3.2版本中的新功能

4.4.3。浮点附加方法

浮点类型实现抽象基类。float还有以下附加方法。numbers.Real

float.**as_integer_ratio**( )

返回一对整数，其比例与原始浮点数完全相等，并带有一个正的分母。提高 OverflowError无穷大和ValueErrorNaNs。

float.**is_integer**( )

返回True如果浮子实例是有限的与积分值，并False以其他方式：

```
>>> (-2.0).is_integer()
True
>>> (3.2).is_integer()
False
```

两种方法支持与十六进制字符串的转换。由于Python的浮点数在内部存储为二进制数，因此将浮点数转换为*十进制*字符串或从*十进制*字符串转换浮点数 通常会导致一个小的舍入误差 相反，十六进制字符串允许精确表示和指定浮点数。这在调试和数值工作时非常有用。

float.hex()

将浮点数的表示形式返回为十六进制字符串。对于有限的浮点数，这种表示将总是包括前导0x和尾随p以及指数。

classmethod float.fromhex(*s*)

Class方法返回由十六进制字符串*s*表示的float。字符串s可能有前导和尾随空白。

请注意，这float.hex()是一种实例方法，而它float.fromhex()是一种类方法。

十六进制字符串采用以下形式：

```
[sign] ['0x'] integer ['.' fraction] ['p' exponent]
```

其中可选sign可以是+或者-，integer并且fraction是十六进制数字的字符串，并且exponent是具有可选前导符号的十进制整数。大小写不重要，整数或小数中至少有一个十六进制数字。此语法与C99标准的6.4.4.2节中指定的语法类似，也与Java 1.5之后的语法类似。特别是，输出float.hex()可用作C或Java代码中的十六进制浮点字面值，并且由C的%a格式字符或Java产生的十六进制字符串Double.toHexString被接受float.fromhex()。

请注意，指数是用十进制而不是十六进制编写的，它给出了乘以系数的2的幂。例如，十六进制字符串0x3.a7p10表示浮点数，或者：(3 + 10./16 + 7./16**2) * 2.0**10 3740.0

```
>>> float.fromhex('0x3.a7p10')
3740.0
```

将反向转换应用于3740.0给出代表相同数字的不同十六进制字符串：

```
>>> float.hex(3740.0)
'0x1.d380000000000p+11'
```

4.4.4。数字类型的散列

对于数字x和y可能是不同类型的数据，每当需要时（请参阅 方法文档以获取更多详细信息）。为了便于实施和效率在各种的数字类型的（包括，，和）Python的数字类型的散列是基于对于任何合理的数目来定义，因此，一个单一的数学函数将应用于的所有实例 和所有的有限的情况下，和中 和。本质上，这个函数是通过对一个固定素数进行减模求得的。Python 的值可用作属性。hash(x) == hash(y) x == y__hash__() int float decimal.Decimal fractions.Fraction int fractions.Fraction float decimal.Decimal PPPmodulus sys.hash_info

CPython实现细节：目前，主要使用的是32位C长的机器和64位C长的机器。P = 2**31 - 1P = 2**61 - 1

详细规则如下：

- 如果是一个非负的有理数，并且不能被整除，定义为，其中给出了模的倒数。x = m / nnPhash(x) m * invmod(n, P) % Pinvmod(n, P) nP
- 如果是一个非负的有理数，并且可以被整除（但不是），那么没有逆模，并且上面的规则不适用;在这种情况下定义 为常数值。x = m / nnPmnPhash(x) sys.hash_info.inf
- 如果是负数有理数定义 为。如果生成的散列是，替换它。x = m / nhash(x) -hash(-x) -1 -2
- 特定的值sys.hash_info.inf，-sys.hash_info.inf 并且sys.hash_info.nan被用作正无穷大，负无穷大或nans（分别）的散列值。（所有可散列的nans具有相同的散列值。）

- 对于一个complex数字z，实部和虚部的散列值通过计算结合起来，减少模数 以使其位于中 。再次，如果结果是，它被替换。 hash(z.real) + sys.hash_info.imag * hash(z.imag) 2**sys.hash_info.widthrange(-2**(sys.hash_info.width - 1), 2**(sys.hash_info.width - 1))-1-2

为了澄清上述规则，下面是一些Python代码示例，等同于内置哈希，用于计算有理数的散列float，或complex：

```
import sys, math

def hash_fraction(m, n):
    """Compute the hash of a rational number m / n.

    Assumes m and n are integers, with n positive.
    Equivalent to hash(fractions.Fraction(m, n)).

    """
    P = sys.hash_info.modulus
    # Remove common factors of P.  (Unnecessary if m and n already coprime.)
    while m % P == n % P == 0:
        m, n = m // P, n // P

    if n % P == 0:
        hash_value = sys.hash_info.inf
    else:
        # Fermat's Little Theorem: pow(n, P-1, P) is 1, so
        # pow(n, P-2, P) gives the inverse of n modulo P.
        hash_value = (abs(m) % P) * pow(n, P - 2, P) % P
    if m < 0:
        hash_value = -hash_value
    if hash_value == -1:
        hash_value = -2
    return hash_value

def hash_float(x):
    """Compute the hash of a float x."""

    if math.isnan(x):
        return sys.hash_info.nan
    elif math.isinf(x):
        return sys.hash_info.inf if x > 0 else -sys.hash_info.inf
    else:
        return hash_fraction(*x.as_integer_ratio())

def hash_complex(z):
    """Compute the hash of a complex number z."""

    hash_value = hash_float(z.real) + sys.hash_info.imag * hash_float(z.imag)
    # do a signed reduction modulo 2**sys.hash_info.width
    M = 2**(sys.hash_info.width - 1)
    hash_value = (hash_value & (M - 1)) - (hash_value & M)
    if hash_value == -1:
        hash_value = -2
    return hash_value
```

4.5。迭代器类型

Python支持对容器进行迭代的概念。这是使用两种不同的方法实现的; 这些用于允许用户定义的类支持迭代。下面更详细地描述的序列总是支持迭代方法。

需要为容器对象定义一个方法来提供迭代支持：

container.__iter__()
返回一个迭代器对象。该对象需要支持下面描述的迭代器协议。如果一个容器支持不同类型的迭代，可以提供额外的方法来专门为这些迭代类型请求迭代器。（支持多种形式迭代的对象的一个例子是支持广度优先和深度优先遍历的树结构。）此方法对应tp_iter于Python / C API中Python对象的类型结构的 槽。

迭代器对象本身需要支持以下两种方法，它们一起构成*迭代器协议*：

iterator.__iter__()
返回迭代器对象本身。这是允许容器和迭代器与for和in语句一起使用所必需的。此方法对应tp_iter于Python / C API中Python对象的类型结构的插槽。

iterator.__next__()
从容器中返回下一个项目。如果没有其他项目，请举例说明StopIteration。此方法对应tp_iternext于Python / C API中Python对象的类型结构的 插槽。

Python定义了几个迭代器对象来支持对通用和特定序列类型，字典以及其他更专用的形式的迭代。除了迭代器协议的实现之外，特定类型并不重要。

一旦迭代器的__next__()方法提出 StopIteration，它必须继续在后续调用中这样做。不遵守这一财产的实施被视为破产。

4.5.1。发电机类型

Python的生成器提供了一种实现迭代器协议的便捷方式。如果容器对象的__iter__()方法作为生成器实现，它将自动返回提供__iter__()和__next__()方法的迭代器对象（技术上讲，生成器对象）。有关生成器的更多信息可以在yield表达式的文档中找到。

4.6。序列类型- ，，list tuplerange

有三种基本的序列类型：列表，元组和范围对象。在专用章节中描述了为处理二进制数据和文本字符串而量身定制的其他序列类型 。

4.6.1。常见序列操作

大多数序列类型都支持下表中的操作，它们都是可变的和不可变的。在collections.abc.Sequence美国广播公司提供，使其更容易正确地执行自定义序列类型这些操作。

此表列出按升序优先级排序的序列操作。在表中，*s*和*t*是相同类型的序列，*n*，*i*，*j*和*k*是整数，*x*是满足由*s*施加的任何类型和值限制的任意对象。

在in和操作具有比较操作相同的优先级。的（串联）和（重复）操作具有相同的优先级对应的数字操作。[3]not in+*

手术	结果	笔记
x in s	True如果一个*s*的项等于*x*，否则False	（1）
x not in s	False如果一个*s*的项等于*x*，否则True	（1）
s + t	*s*和 *t*的连接	（6）（7）

s * n 要么 n * s	相当于增加*小号*到本身*ñ*倍	（2）（7）
s[i]	*我*的个项*小号*，原点0	（3）
s[i:j]	的切片*小号*从*我*到*ĵ*	（3）（4）
s[i:j:k]	的切片*小号*从*我*到*ĵ*与步骤*ķ*	（3）（5）
len(s)	*s*的长度	
min(s)	*s*的最小项	
max(s)	最大的项目*小号*	
s.index(x[, i[, j]])	的第一次出现的指数*x*在*小号*（在或索引后*我*和指数前*ĵ*）	（8）
s.count(x)	出现的总次数 *x*在*小号*	

相同类型的序列也支持比较。具体来说，元组和列表通过比较相应的元素按字典顺序进行比较。这意味着为了比较相等，每个元素必须相等并且两个序列必须是相同类型并具有相同长度。（有关完整的详细信息，请参阅语言参考中的比较。）

笔记：

1. 虽然in和操作仅用于简单容纳测试在一般情况下，一些专门的序列（如，和）也将它们用于序列检测：not instrbytesbytearray

```
>>> "gg" in "eggs"
True
```

2. 小于*n*的值0被视为0（其产生与*s*相同类型的空序列）。请注意，序列中*的*项目不会被复制; 它们被多次引用。这经常困扰着新的Python程序员; 考虑：

```
>>> lists = [[]] * 3
>>> lists
[[], [], []]
>>> lists[0].append(3)
>>> lists
[[3], [3], [3]]
```

发生了什么[[]]是一个包含一个空列表的单元素列表，因此所有三个元素都是对这个单个空列表的引用。修改任何修改此单个列表的元素。您可以通过以下方式创建不同列表的列表：[[]] * 3lists

```
>>> lists = [[] for i in range(3)]
>>> lists[0].append(3)
>>> lists[1].append(5)
>>> lists[2].append(7)
>>> lists
[[3], [5], [7]]
```

进一步的解释可以在FAQ条目中找到 如何创建多维列表？。

3. 如果*i*或*j*是负数，则该索引是相对于序列*s*的末尾： 或被替换。但请注意，依然如此。len(s) + ilen(s) + j-00

4. 的切片*小号*从*我*到*ĵ*被定义为物品的具有索引序列 *ķ*使得。如果*我*或*j*大于，请使用 。如果*我*被省略或者使用。如果省略了*j*，或者 使用。如果*我*大于或等于*j*，则切片为空。 i <= k <

jlen(s) len(s) None 0 None len(s)

5. 步骤*k*中从*i*到*j*的*s*片段被定义为具有这样的索引的项目序列。换句话说，在指数，，，等等，停车时 *j*达到（但绝不包括*j*）。当*k*是正数时，如果它们更大，则 *i*和*j*会减少。当*k*是负数时，如果*i*和*j*更大，则*i*和*j*减小。如果*我*或*j*被省略或 `x = i + n*k 0 <= n < (j-i)/k i i+k i+2*k i+3*k len(s) len(s) - 1 None`，它们变成了“终点”值（终点取决于*k*的符号）。请注意，*k*不能为零。如果*k*是`None`，它被视为像`1`。

6. 串联不可变序列总是会产生一个新对象。这意味着通过重复串联构建序列在总序列长度中将具有二次运行成本。要获得线性运行时成本，您必须切换到以下选项之一：

   - 如果连接`str`对象，则可以创建列表并`str.join()`在最后使用 ，或者写入`io.StringIO` 实例并在完成时检索其值
   - 如果连接`bytes`对象，则可以同样使用 `bytes.join()`或`io.BytesIO`，或者可以在`bytearray` 对象中进行就地连接。 `bytearray` 对象是可变的并且具有有效的分配机制
   - 如果级联`tuple`对象，扩展`list`代替
   - 对于其他类型，请查看相关的类文档

7. 某些序列类型（例如`range`）仅支持遵循特定模式的项目序列，因此不支持序列连接或重复。

8. `index` `ValueError`当在*s*中找不到*x*时引发。并非所有实现都支持传递附加参数*i*和*j*。这些参数允许有效地搜索序列的子部分。传递额外的参数大致相当于使用，只是不复制任何数据，并且返回的索引是相对于序列的开始而不是片段的开始。`s[i:j].index(x)`

4.6.2。不可变序列类型

不可变序列类型通常实现的唯一操作也不能通过可变序列类型实现，这是对`hash()` 内置的支持。

这种支持允许不可改变的序列，如`tuple`实例中，以用作`dict`密钥，并存储在`set`和`frozenset` 实例。

尝试散列包含不可取值的不可变序列将导致`TypeError`。

4.6.3。可变序列类型

下表中的操作定义在可变序列类型上。在`collections.abc.MutableSequence`美国广播公司提供，使其更容易正确地执行自定义序列类型这些操作。

在表中，*s*是可变序列类型的一个实例，*t*是任何可迭代对象，*x*是满足由*s*施加的任何类型和值限制的任意对象（例如，`bytearray`只接受满足值限制的整数）。`0 <= x <= 255`

手术	结果	笔记
`s[i] = x`	项*我*的*小号*被替换为 *X*	
`s[i:j] = t`	的切片*小号*从*我*到*ĵ* 由迭代的内容替换*吨*	
`del s[i:j]`	与...一样 `s[i:j] = []`	
`s[i:j:k] = t`	元素`s[i:j:k]` 被替换为*t*的元素	（1）
`del s[i:j:k]`	`s[i:j:k]`从列表中删除元素	
`s.append(x)`	将*x*附加到序列的末尾（与之相同 ） `s[len(s):len(s)] = [x]`	
`s.clear()`	删除所有项目`s` （与之相同）`del s[:]`	（5）
`s.copy()`	创建一个浅的副本`s` （与之相同 `s[:]` ）	（5）

s.extend(t) 要么 s += t	用*t*的内容扩展*s*（大部分相同） s[len(s):len(s)] = t	
s *= n	更新*小号*其内容重复*ñ*次	（6）
s.insert(i, x)	插入*x*成*小号*由下式给出的指数在*我*（同） s[i:i] = [x]	
s.pop([i])	在*i*处检索项目并将其从*s*中移除	（2）
s.remove(x)	从取出的第一个项目*小号*哪里s[i] == x	（3）
s.reverse()	反转*s*的项目到位	（4）

笔记：

1. *t*必须与它正在替换的切片具有相同的长度。
2. 可选参数*我*默认为-1，以便默认情况下最后一项被删除并返回。
3. remove ValueError当在*s*中找不到*x*时引发。
4. 该reverse()方法在倒转大序列时修改用于空间经济的序列。要提醒用户它是以副作用方式运行的，它不会返回相反的顺序。
5. clear()并且copy()包含它们是为了与不支持切片操作的可变容器（例如dict和set）的接口一致，

   *3.3版新增功能：*clear()和copy()方法。
6. 值*n*是一个整数或实现的对象 __index__()。*n*的零和负值清除序列。序列中的项目不会被复制; 它们被多次引用，正如在Common Sequence Operations下所解释的那样。 s * n

4.6.4。列表

列表是可变序列，通常用于存储同类项目的集合（其中精确的相似度因应用程序而异）。

class list（[*iterable*]）

列表可以用几种方式构建：

- 用一对方括号表示空列表： []
- 使用方括号，用逗号分隔项目：[a]，[a, b, c]
- 使用列表理解： [x for x in iterable]
- 使用类型构造函数：list()或list(iterable)

构造函数构建一个列表，其项目与*iterable*的项目相同并且顺序相同。 *可迭代*可以是序列，支持迭代的容器或迭代器对象。如果*可迭代* 已经是一个列表，则复制将被创建并返回，类似于iterable[:]。例如，list('abc')退货和 退货。如果没有给出参数，构造函数会创建一个新的空列表。['a', 'b', 'c']list((1, 2, 3))[1, 2, 3][]

许多其他操作也会生成列表，包括sorted() 内置的。

列出所有常见和 可变顺序操作。列表还提供了以下附加方法：

sort（ *, *key* = *None*，*reverse* = *False* ）

此方法就地对列表进行排序，仅使用<项目之间的比较。异常不会被抑制 - 如果任何比较操作失败，整个排序操作将失败（并且列表可能会保留在部分修改状态）。

sort()接受只能通过关键字传递的两个参数（仅关键字参数）：

*key*指定一个参数的函数，该参数用于从每个列表元素中提取比较键（例如，key=str.lower）。与列表中的每个项目相对应的键将被计算一次，然后用于整个排序过程。默认值None 意味着列表项直接排序而不需要计算单独的键值。

该functools.cmp_to_key()实用程序可用于将2.x样式的*cmp*函数转换为*关键*函数。

*reverse*是一个布尔值。如果设置为True，则列表元素按照每个比较被颠倒的顺序进行排序。

这种方法在排序大序列时修改空间经济的序列。为了提醒用户它是以副作用方式运行的，它不会返回已排序的序列（用于sorted()显式请求一个新的已排序列表实例）。

该sort()方法保证稳定。如果确保不会更改比较相等的元素的相对顺序，则排序是稳定的 - 这对于多次排序（例如，按部门排序，然后按薪级）进行排序很有帮助。

CPython实现细节：列表正在排序时，尝试对列表进行更改或甚至检查的影响是未定义的。Python的C实现使得列表在持续时间内显示为空，并且ValueError如果它能够检测列表在排序过程中发生了变异，就会引发该列表。

4.6.5。元组

元组是不可变的序列，通常用于存储异构数据的集合（例如由enumerate() 内置生成的2元组）。元组也用于需要不变序列的同类数据的情况（例如允许存储在一个set或 一个dict实例中）。

class tuple（[*iterable*]）

元组可以通过多种方式构建：

- 使用一对括号来表示空元组： ()
- 使用单个元组的尾部逗号：a, 或(a,)
- 用逗号分隔项目：或a, b, c(a, b, c)
- 使用tuple()内置的：tuple()或tuple(iterable)

构造函数构建一个元组，其元素与*iterable*的元素相同且顺序相同。 *可迭代*可以是序列，支持迭代的容器或迭代器对象。如果*可迭代* 已经是一个元组，它将不会被返回。例如， tuple('abc')退货和 退货。如果没有给出参数，构造函数将创建一个新的空元组，。('a', 'b', 'c')tuple([1, 2, 3])(1, 2, 3)()

请注意，它实际上是制作元组的逗号，而不是括号。括号是可选的，除了在空元组情况下，或者当需要避免句法歧义时。例如， 是具有三个参数的函数调用，而具有三元 组作为唯一参数的函数调用。f(a, b, c)f((a, b, c))

元组实现了所有常见的序列操作。

对于按名称访问比按索引访问更清晰的异构数据集合，collections.namedtuple()可能是比简单元组对象更合适的选择。

4.6.6。范围

该range类型表示一个不可变的数字序列，通常用于在循环中循环特定的次数for 。

class range（ *stop* ）
*类*range（ *开始*，*停止*[，*步骤*] ）

范围构造函数的参数必须是整数（内置 int或实现__index__特殊方法的任何对象）。如果省略*step*参数，则默认为1。如果省略*start*参数，则默认为0。如果*步*长为零，ValueError则升起。

对于积极的*步骤*，范围的内容r由公式where 和 确定。r[i] = start + step*ii >= 0r[i] < stop

对于负*步骤*，该范围的内容仍然由公式确定，但约束 和。r[i] = start + step*ii >= 0r[i] > stop

如果r[0]不符合值约束，范围对象将为空。范围确实支持负指数，但这些被解释为从正指数确定的序列末尾开始的索引。

包含的绝对值大于sys.maxsize允许的范围，但某些功能（如len()）可能会提高 OverflowError。

范围示例：

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(0, 30, 5))
[0, 5, 10, 15, 20, 25]
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
>>> list(range(0, -10, -1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> list(range(0))
[]
>>> list(range(1, 0))
[]
```

范围实现除串联和重复以外的所有常见序列操作（由于范围对象只能表示遵循严格模式的序列，重复和串联通常会违反该模式）。

start
*起始*参数的值（或者0如果参数未提供）

stop
*停止*参数的值

step
*step*参数的值（或者1如果参数未提供）

所述的优点range类型通过常规list或 tuple是一个range对象将始终以相同的内存（小）数量，无论它代表的范围内的大小（因为它仅存储start，stop和step值，计算各个项目和子范围需要）。

Range对象实现collections.abc.SequenceABC，并提供诸如包含测试，元素索引查找，切片和负指数支持等功能（请参见序列类型 - 列表，元组，范围）：

```
>>> r = range(0, 20, 2)
>>> r
range(0, 20, 2)
>>> 11 in r
False
>>> 10 in r
```

```
True
>>> r.index(10)
5
>>> r[5]
10
>>> r[:5]
range(0, 10, 2)
>>> r[-1]
18
```

测试对象范围与平等==和!=比较它们作为序列。也就是说，如果两个范围对象表示相同的值序列，则它们被认为是相等的。（请注意，比较相等的两个范围的对象可能有不同的start， stop和step属性，例如 或）。 range(0) == range(2, 1, 3) range(0, 3, 2) == range(0, 4, 2)

*在版本3.2中更改：*实现序列ABC。支持切片和负指数。 int以恒定时间测试对象的对象，而不是迭代所有项目。

*在版本3.3中更改：*定义'=='和'！='以根据它们定义的值序列比较范围对象（而不是根据对象标识进行比较）。

*新的3.3版：*的start ， stop和step 属性。

也可以看看:

- 该linspace配方 展示了如何实现一个懒惰的版本范围的适用于浮点应用程序。

4.7。文本序列类型 - str

Python中的文本数据由str对象或*字符串*处理。字符串是Unicode代码点的不可变 序列。字符串文字用各种方式书写：

- 单引号：'allows embedded "double" quotes'
- 双引号：。"allows embedded 'single' quotes"
- 三重引用：，'''Three single quotes'''"""Three double quotes"""

三重引用的字符串可能会跨越多行 - 所有关联的空白都将包含在字符串文字中。

作为单个表达式的一部分并且只有空白字符串的字符串将被隐式转换为单个字符串文字。那就是，。("spam " "eggs") == "spam eggs"

有关字符串文字的各种形式（包括支持的转义序列）和r禁用大多数转义序列处理的（“raw”）前缀，请参阅字符串和字节文字。

字符串也可以使用str 构造函数从其他对象创建。

由于没有单独的“字符”类型，索引的字符串产生一个长度1。也就是说，对于一个非空字符串的字符串*小号*，。 s[0] == s[0:1]

也没有可变的字符串类型，但str.join()还是 io.StringIO可以用来有效地构造从多个片段字符串。

*版本3.3中更改：*为了向后兼容Python 2系列，u字符串文字再次允许前缀。它对字符串文字的含义没有影响，不能与r前缀结合使用。

class str（*object* =''）
class str（*object* = b''，*encoding* ='utf-8'，*errors* ='strict'）

返回一个字符串版本的*对象*。如果没有提供*对象*，则返回空字符串。否则，行为str() 取决于是否给出*编码*或*错误*，如下所示。

如果既没有*编码*也不*错误*给出，str(object) 返回 object.__str__() ，这是“非正规”或很好的可打印字符串表示*对象*。对于字符串对象，这是字符串本身。如果*对象*没有__str__() 方法，则str() 返回返回 repr(object)。

如果至少给出了*编码*或*错误*之一，则*对象*应该是类似 字节的对象（例如bytes或bytearray ）。在这种情况下，如果*对象*是bytes （或bytearray ）对象，则等同于 。否则，在调用之前获取缓冲区对象下的字节对象 。有关缓冲区对象的信息，请参阅二进制序列类型 - 字节，字节阵列，内存视图和 缓冲区协议。 str(bytes, encoding, errors) bytes.decode(encoding, errors) bytes.decode()

传递一个没有*编码* 或*错误*参数的bytes对象属于返回非正则字符串表示形式的第一种情况（另请参阅Python 的命令行选项）。例如： str()-b

```
>>> str(b'Zoot!')
"b'Zoot!'"
```

有关str该类及其方法的更多信息，请参见下面的 文本序列类型 - str和字符串方法部分。要输出格式化字符串，请参阅格式化字符串文字和格式字符串语法 部分。另外，请参阅文本处理服务部分。

4.7.1 字符串方法

字符串实现所有常见的序列操作，以及下面描述的其他方法。

字符串还支持两种类型的字符串格式化的，一个提供了很大程度的灵活性和定制（见str.format() ，格式化字符串的语法和自定义字符串格式化）和其他基于C printf风格的格式，处理范围较窄的类型，是稍硬使用正确，但它可以处理的情况通常更快（printf样式的字符串格式）。

标准库的文本处理服务部分涵盖了许多其他提供各种文本相关实用程序（包括re模块中的正则表达式支持）的模块。

str.capitalize()

返回字符串的一个副本，其首字母大写，其余的小写。

str.casefold()

返回字符串的casefolded副本。可折叠的字符串可用于无外壳匹配。

Casefolding与lowercasing类似，但更具侵略性，因为它旨在删除字符串中的所有大小写区别。例如，德文小写字母'ß' 相当于"ss"。既然已经小写了， lower()什么都不会做'ß'；casefold() 将其转换为"ss"。

Unicode标准的第3.13节描述了casefolding算法。

3.3版本的新功能

str.center(*width*[, *fillchar*])

返回以一个长度*宽度*的字符串为中心。填充是使用指定的*fillchar*完成的（默认是ASCII空间）。如果*宽度*小于或等于，则返回原始字符串len(s)。

str.count(*sub*[, *start*[, *end*]])

返回[*start* , *end*] 范围内子串*sub*的非重叠次数。可选参数*开始*和*结束*被解释为切片符号。

str.encode(*encoding="utf-8", errors="strict"*)

以字节对象的形式返回字符串的编码版本。默认编码是'utf-8'。可能会给出*错误*来设置不同的错误处理方案。*错误*的默认值是'strict'，这意味着编码错误会引发一个错误UnicodeError。其他可能的值'ignore'，'replace'，'xmlcharrefreplace'，'backslashreplace'和其他任何名义通过挂号 codecs.register_error()，见错误处理程序。有关可能的编码列表，请参见标准编码部分。

*在版本3.1中进行了更改：*添加了对关键字参数的支持。

str.endswith(*后缀*[, *start*[, *end*]])

返回True字符串是否与指定的结束*后缀*，否则返回 False。 *后缀*也可以是后缀的元组来查找。随着可选 *启动*，测试开始在那个位置。选择*结束时*，停止在该位置进行比较。

str.expandtabs(*tabsize = 8*)

返回字符串的副本，其中所有制表符由一个或多个空格替换，具体取决于当前列和给定制表符大小。选项卡位置出现在每个*制表符大小*字符中（默认值为8，在列*0,8,16*等处提供制表位置）。要扩展字符串，当前列设置为零，字符串逐个检查。如果该字符是一个制表符（\t），则结果中会插入一个或多个空格字符，直到当前列等于下一个制表符位置。（制表符本身不被复制。）如果该字符是换行符（\n）或返回（\r），它被复制并且当前列被重置为零。任何其他字符都将被不变地复制，而当前列增加1，无论打印时字符如何表示。

```
>>> '01\t012\t0123\t01234'.expandtabs()
'01      012     0123    01234'
>>> '01\t012\t0123\t01234'.expandtabs(4)
'01  012 0123    01234'
```

str.find(*sub*[, *start*[, *end*]])

返回切片中找到substring *子*字符串的最低索引s[start:end]。可选参数*开始*和*结束*被解释为切片符号。-1如果未找到*子项*，则返回。

注意： find()只有在您需要知道*sub*的位置时才应该使用该方法。要检查*sub*是否是子字符串，请使用 in运算符：

```
>>> 'Py' in 'Python'
True
```

str.format(**args*, ***kwargs*)

执行字符串格式化操作。调用此方法的字符串可以包含由大括号分隔的文本文本或替换字段 {}。每个替换字段包含位置参数的数字索引或关键字参数的名称。返回字符串的副本，其中每个替换字段将替换为相应参数的字符串值。

```
>>> "The sum of 1 + 2 is {0}".format(1+2)
'The sum of 1 + 2 is 3'
```

请参阅格式字符串语法以获取可在格式字符串中指定的各种格式选项的说明。

注意： 当格式化数（int，float，float 与和亚类）n类型（例如：'{:n}'.format(1234)），功能暂时设定LC_CTYPE的区域设置到LC_NUMERIC 区域解码decimal_point和thousands_sep领域localeconv()如果它们是非ASCII或长于1个字节，而 LC_NUMERIC区域设置为比不同在LC_CTYPE语言环境。此临时更改影响其他线程。

*在版本3.6.5中进行了更改：*在使用n类型格式化数字时，函数会在某些情况下LC_CTYPE将LC_NUMERIC语言环境临时设置为语言环境。

str.format_map(*映射*)

类似于str.format(**mapping)，除了mapping直接使用和不复制到一个dict。例如，如果mapping是字典子类，这很有用：

```
>>> class Default(dict):
...     def __missing__(self, key):
...         return key
...
>>> '{name} was born in {country}'.format_map(Default(name='Guido'))
'Guido was born in country'
```

3.2版本中的新功能

str.index(*sub*[, *start*[, *end*]])

喜欢find()，但ValueError在没有找到子字符串时引发。

str.isalnum()

如果字符串中的所有字符都是字母数字并且至少有一个字符，则返回true，否则返回false。字符c是字母数字，如果下面的返回之一True：c.isalpha()，c.isdecimal()，c.isdigit()，或c.isnumeric()。

str.isalpha()

如果字符串中的所有字符都是字母并且至少有一个字符，则返回true，否则返回false。字母字符是在Unicode字符数据库中定义为“Letter”的那些字符，即具有通用类别属性为“Lm”，“Lt”，“Lu”，“Ll”或“Lo”之一的那些字符。请注意，这与Unicode标准中定义的“字母”属性不同。

str.isdecimal()

如果字符串中的所有字符都是十进制字符并且至少有一个字符，则返回true，否则返回false。十进制字符是可以用来形成10位数字的那些字符，例如U + 0660，ARABIC-INDIC DIGIT ZERO。形式上，十进制字符是Unicode常规类别“Nd”中的字符。

str.isdigit()

如果字符串中的所有字符都是数字并且至少有一个字符，则返回true，否则返回false。数字包含需要特殊处理的十进制字符和数字，例如兼容性上标数字。这包括不能用于形成基数为10的数字，如Kharosthi数字。形式上，数字是具有属性值Numeric_Type = Digit或Numeric_Type = Decimal的字符。

str.isidentifier()

根据语言定义标识符和关键字部分，如果字符串是有效标识符，则返回true。

使用keyword.iskeyword()测试保留的标识符，例如def和class。

str.islower()

如果字符串中的所有套用字符[4]都是小写，并且至少有一个套用字符，则返回true，否则返回false。

str.isnumeric()

如果字符串中的所有字符都是数字字符，并且至少有一个字符，则返回true，否则返回false。数字字符包括数字字符和具有Unicode数值属性的所有字符，例如U + 2155，VULGAR FRACTION ONE FIFTH。形式上，数字字符是具有属性值Numeric_Type = Digit，Numeric_Type = Decimal或Numeric_Type = Numeric的字符。

str.isprintable()

如果字符串中的所有字符都可打印或字符串为空，则返回true，否则返回false。非可打印字符是在Unicode字符数据库中定义为“Other”或“Separator”的字符，但ASCII空格（0x20）被认为是可打印的。（请注意，在这个上下文中的可打印字符repr()是在字符串上调用时不应该转义 的字符，它与写入sys.stdout或的字符串的处理无关sys.stderr。

str.isspace()

如果字符串中只有空格字符，并且至少有一个字符，则返回true，否则返回false。空格字符是在Unicode字符数据库中定义为“其他”或“分隔符”的那些字符，以及具有“WS”，“B”或“S”之一的双向属性的那些字符。

str.istitle()

如果字符串是一个标题字符串并且至少有一个字符，则返回true，例如，大写字符只能跟在未写入的字符之后，而小写字母只能在已封装的字符之后。否则返回false。

str.isupper()

如果字符串中的所有套用字符[4]都是大写且至少有一个套用字符，则返回true，否则返回false。

str.join(*可迭代*)

返回一个字符串，它是*可迭代*的字符串的串联。TypeError如果*迭代*中包含任何非字符串值（包括bytes对象），则会引发 A. 元素之间的分隔符是提供此方法的字符串。

str.ljust(*width*[, *fillchar*])

以长度*宽度*的字符串返回左对齐的字符串。填充是使用指定的*fillchar*完成的（默认是ASCII空间）。如果*宽度*小于或等于，则返回原始字符串len(s)。

str.lower()

返回字符串的一个副本，并将所有装入字符的字符[4]转换为小写字母。

Unicode标准的第3.13节描述了使用的低级算法。

str.lstrip([*chars*])

返回删除前导字符的字符串的副本。的*字符* 参数是要除去的字符串指定的字符集。如果省略或者None，*chars*参数默认删除空格。该*字符* 参数不是前缀; 相反，其值的所有组合都被剥离：

```
>>> '   spacious   '.lstrip()
'spacious   '
>>> 'www.example.com'.lstrip('cmowz.')
'example.com'
```

static str.maketrans(*x*[, *y*[, *z*]])

这个静态方法返回一个可用于的翻译表str.translate()。

如果只有一个参数，它必须是一个将Unicode序号（整数）或字符（长度为1的字符串）映射到Unicode序号，字符串（任意长度）或者字典的字典None。字符键将被转换为序号。

如果有两个参数，它们必须是等长的字符串，并且在结果字典中，x中的每个字符将映射到y中相同位置的字符。如果有第三个参数，它必须是一个字符串，其字符将被映射到None结果中。

str.partition(*sep*)

在*sep*第一次出现时拆分字符串，并返回包含分隔符之前的部分，分隔符本身和分隔符之后的部分的三元组。如果未找到分隔符，则返回包含该字符串本身的三元组，然后返回两个空字符串。

str.replace(*旧的*, *新的*[, *count*])

返回所有出现的*旧*字符串替换为 *新*字符串的副本。如果给出可选的参数*计数*，则仅 替换第一个*计数*事件。

str.rfind(*sub* [, *start* [, *end*]])

返回找到substring *子*字符串的最高索引，这样*sub*就包含在其中s[start:end]。可选参数*开始* 和*结束*被解释为切片符号。返回-1失败。

str.rindex(*sub* [, *start* [, *end*]])

像rfind()但ValueError子字符串 *子*未找到时引发。

str.rjust(*width* [, *fillchar*])

以字符串长度*宽度*返回右对齐的字符串。填充是使用指定的*fillchar*完成的（默认是ASCII空间）。如果*宽度*小于或等于，则返回原始字符串len(s)。

str.rpartition(*sep*)

在最后出现的*sep*处拆分字符串，并返回包含分隔符之前的部分，分隔符本身和分隔符之后的部分的三元组。如果未找到分隔符，则返回包含两个空字符串的三元组，然后返回字符串本身。

str.rsplit(*sep = None*, *maxsplit = -1*)

使用*sep*作为分隔符字符串，返回字符串中单词的列表。如果给出*maxsplit*，则最多*maxsplit*分裂，*最右边* 的分裂。如果没有指定*sep*或者None任何空格字符串是分隔符。除了从右边分开外，rsplit()其行为 split()如下所述。

str.rstrip([*chars*])

返回删除了尾随字符的字符串副本。的*字符* 参数是要除去的字符串指定的字符集。如果省略或者None，*chars*参数默认删除空格。该*字符* 参数不是后缀; 相反，其值的所有组合都被剥离：

```
>>> '   spacious   '.rstrip()
'   spacious'
>>> 'mississippi'.rstrip('ipz')
'mississ'
```

str.split(*sep = None*, *maxsplit = -1*)

使用*sep*作为分隔符字符串，返回字符串中单词的列表。如果给出*maxsplit*，最多可以完成*maxsplit*分割（因此，列表最多只有maxsplit+1元素）。如果未指定*maxsplit*或者-1，则分割数量没有限制（所有可能的分割）。

如果给出了*sep*，则连续分隔符不会分组在一起，并被视为分隔空字符串（例如，'1,,2'.split(',')返回 ）。该*月*的参数可以由多个字符（例如，返回）。用指定的分隔符分割空字符串返回。['1', '', '2']'1<>2<>3'.split('<>')['1', '2', '3']['']

例如：

```
>>> '1,2,3'.split(',')
['1', '2', '3']
>>> '1,2,3'.split(',', maxsplit=1)
['1', '2,3']
>>> '1,2,,3,'.split(',')
['1', '2', '', '3', '']
```

如果未指定*sep*或is `None`，则应用不同的分割算法：将连续空白的运行视为单个分隔符，并且如果字符串具有前导或尾随空白，则结果的开始或结束将不包含空字符串。因此，用`None`分隔符分隔一个空字符串或一个只包含空格的字符串会返回`[]`。

例如：

```
>>> '1 2 3'.split()
['1', '2', '3']
>>> '1 2 3'.split(maxsplit=1)
['1', '2 3']
>>> '   1   2   3   '.split()
['1', '2', '3']
```

str.**splitlines**([*keepends*])

返回字符串中行的列表，在行边界处突破。除非换行符不包括在结果列表中*keepends*，并给出正确的。

此方法分割在以下行边界上。特别是，边界是通用换行符的超集。

表示	描述
\n	换行
\r	回车
\r\n	回车+换行
\v 要么 \x0b	线条制表
\f 要么 \x0c	换页
\x1c	文件分隔符
\x1d	组分隔符
\x1e	记录分隔符
\x85	下一行（C1控制代码）
\u2028	线分隔符
\u2029	段落分隔符

在版本3.2中更改：`\v`并`\f`添加到行边界列表中。

例如：

```
>>> 'ab c\n\nde fg\rkl\r\n'.splitlines()
['ab c', '', 'de fg', 'kl']
>>> 'ab c\n\nde fg\rkl\r\n'.splitlines(keepends=True)
['ab c\n', '\n', 'de fg\r', 'kl\r\n']
```

与给定`split()`分隔符字符串*sep*时不同，此方法返回空字符串的空列表，并且终端行分隔符不会导致多余的行：

```
>>> "".splitlines()
[]
>>> "One line\n".splitlines()
['One line']
```

为了比较，split('\n')给出：

```
>>> ''.split('\n')
['']
>>> 'Two lines\n'.split('\n')
['Two lines', '']
```

str.**startswith**(*prefix*[, *start*[, *end*]])

返回True字符串是否与开始*前缀*，否则返回False。 *前缀*也可以是要查找的前缀元组。使用可选*启动*，测试字符串从该位置开始。使用可选*结束时*，停止在该位置比较字符串。

str.**strip**([*chars*])

返回删除前导字符和尾随字符的字符串副本。的*字符*参数是要除去的字符串指定的字符集。如果省略或者None，*chars*参数默认删除空格。该*字符*参数不是前缀或后缀; 相反，其值的所有组合都被剥离：

```
>>> '   spacious   '.strip()
'spacious'
>>> 'www.example.com'.strip('cmowz.')
'example'
```

从字符串中剥离最外部的前导*字符*和结尾*字符*参数值。字符从前端删除，直到达到未包含在集字符的字符串文字*字符*。尾端发生类似的行为。例如：

```
>>> comment_string = '#....... Section 3.2.1 Issue #32 .......'
>>> comment_string.strip('.#! ')
'Section 3.2.1 Issue #32'
```

str.**swapcase**( )

返回大写字符转换为小写字符串的副本，反之亦然。请注意，这不一定是真的。 s.swapcase().swapcase() == s

str.**title**( )

返回字符串的字幕版本，其中字以大写字符开头，其余字符为小写字母。

例如：

```
>>> 'Hello world'.title()
'Hello World'
```

该算法使用简单的与语言无关的单词作为连续字母组的定义。该定义在许多情况下都有效，但这意味着收缩和占有者中的撇号会形成单词边界，这可能不是理想的结果：

```
>>> "they're bill's friends from the UK".title()
"They're Bill's Friends From The Uk"
```

可以使用正则表达式构造撇号的解决方法：

```
>>> import re
>>> def titlecase(s):
...     return re.sub(r"[A-Za-z]+('[A-Za-z]+)?",
...                   lambda mo: mo.group(0)[0].upper() +
...                              mo.group(0)[1:].lower(),
...                   s)
...
>>> titlecase("they're bill's friends.")
"They're Bill's Friends."
```

str.translate(*表*)

返回通过给定转换表映射每个字符的字符串的副本。该表必须是实现索引的对象__getitem__()，通常是映射或 序列。当通过Unicode序数（整数）索引时，表对象可以执行以下任何操作：返回Unicode序号或字符串，将字符映射为一个或多个其他字符; 返回 None，从返回字符串中删除字符; 或者引发 LookupError异常，将角色映射到自身。

您可以使用str.maketrans()不同格式的字符到字符映射来创建翻译地图。

另请参阅codecs模块，以获得更灵活的自定义字符映射方法。

str.upper()

返回字符串的一个副本，并将所有装入字符的字符[4]转换为大写字符。请注意，str.upper().isupper()可能是False如果s包含无套管的字符或如果所得到的字符（县）的Unicode类别不是“吕氏春秋”（字母，大写），但如“LT”（字母，首字母大写）。

Unicode标准的第3.13节描述了使用的大写算法。

str.zfill(*宽度*)

返回左侧填充有ASCII '0' 数字的字符串的副本，以生成一个长度*宽度*的字符串。前导符号前缀（'+'/'-'）是通过在符号字符*之后*插入填充而不是之前处理的。如果*宽度*小于或等于，则返回原始字符串len(s)。

例如：

```
>>> "42".zfill(5)
'00042'
>>> "-42".zfill(5)
'-0042'
```

4.7.2。printf风格的字符串格式

注意: 这里描述的格式化操作展现了各种各样的怪癖，导致了许多常见错误（例如，未能正确显示元组和字典）。使用新的格式化字符串文字或str.format()界面有助于避免这些错误。这些替代方案还为格式化文本提供了更强大，灵活和可扩展的方法。

字符串对象有一个独特的内置操作：%操作符（模）。这也被称为字符串*格式化*或*插值*运算符。鉴于（其中*格式*是字符串），*格式*中的转换规范将替换为零个或多个*值*元素。其效果与在C语言中使用类似。format % values%sprintf()

如果*格式*需要单个参数，则*值*可能是单个非元组对象。[5] 否则，*值*必须是具有由格式字符串或单个映射对象（例如字典）指定的项目数量的元组。

转换说明符包含两个或多个字符，并具有以下组件，它们必须按以下顺序出现：

1. 该'%'字符表示说明符的开始。
2. 映射键（可选），由括号括起来的字符序列组成（例如(somename)）。
3. 转换标志（可选），影响某些转换类型的结果。
4. 最小字段宽度（可选）。如果指定为'*'（星号），则将从元组的下一个元素*值中*读取实际宽度，并且要转换的对象位于最小字段宽度和可选精度之后。
5. 精度（可选），以'.'（点）形式给出，然后是精度。如果指定为'*'（星号），则实际精度将从元组的下一个元素的*值中*读取，并且要转换的值位于精度之后。
6. 长度修饰符（可选）。
7. 转换类型。

当正确的参数是一个字典（或其他映射类型）时，字符串中的格式*必须*在字符后面插入一个带括号的映射关键字'%'。映射键从映射中选择要格式化的值。例如：

```
>>> print('%(language)s has %(number)03d quote types.' %
...       {'language': "Python", "number": 2})
Python has 002 quote types.
```

在这种情况下*，格式中不会出现说明符（因为它们需要一个连续的参数列表）。

转换标志字符是：

旗	含义
'#'	值转换将使用"替代形式"（下面定义）。
'0'	转换将为数字值填充零。
'-'	转换后的值是左侧调整的（'0'如果两者都给出，则覆盖转换）。
' '	换一个空格）在由正确的数字（或空字符串）产生的一个空白应该留下一个有符号的转
'+'	符号字符（'+'或'-'）将在转换之前（覆盖"空格"标志）。

长度修饰符（h，l或L）可能存在，但会被忽略，因为它对于Python不是必需的 - 例如与之%ld相同%d。

转换类型是：

转变	含义	笔记
'd'	带符号的整数小数。	
'i'	带符号的整数小数。	
'o'	签署八进制值。	（1）
'u'	已过时的类型 - 与之相同'd'。	（6）
'x'	签名的十六进制（小写）。	（2）
'X'	签名的十六进制（大写）。	（2）
'e'	浮点指数格式（小写）。	（3）
'E'	浮点指数格式（大写）。	（3）
'f'	浮点十进制格式。	（3）
'F'	浮点十进制格式。	（3）
'g'	浮点格式。如果指数小于-4或小于精度，则使用小写指数格式，否则使用	（4）
'G'	浮点格式。如果指数小于-4或者不小于精度，则使用大写指数格式，否则使用十进制格式。	（4）
'c'	单个字符（接受整数或单个字符串）。	
'r'	字符串（使用任何Python对象转换 repr() ）。	（5）
's'	字符串（使用任何Python对象转换 str() ）。	（5）
'a'	字符串（使用任何Python对象转换 ascii() ）。	（5）
'%'	没有参数被转换，导致结果中的'%'字符。	

笔记：

1. 替代形式导致'0o'在第一个数字之前插入前导八进制说明符（ ）。
2. 替代形式会导致在第一个数字之前插入'0x'或'0X'取决于是否使用了'x'或'X'格式。
3. 替代形式会导致结果始终包含小数点，即使没有数字跟着它。

   精度决定小数点后的位数，默认为6。
4. 替代形式导致结果始终包含小数点，并且尾部零不会像原本那样被删除。

   精度决定小数点前后的有效位数，默认值为6。
5. 如果精度是N，则输出被截断为N字符。
6. 看到 **PEP 237**。

由于Python字符串具有明确的长度，因此%s转换不会假定'\0'字符串的末尾。

在版本3.1中进行了更改：%f绝对值超过1e50的数字的%g转换不再被转换取代。

4.8。二进制序列类型- ，，bytes bytearraymemoryview

操作二进制数据的核心内置类型是bytesand bytearray。它们受到支持，memoryview它使用缓冲区协议来访问其他二进制对象的内存而无需进行复制。

该array模块支持基本数据类型（如32位整数和IEEE754双精度浮点值）的高效存储。

4.8.1。字节对象

字节对象是单个字节的不可变序列。由于许多主要的二进制协议都基于ASCII文本编码，因此字节对象提供了几种方法，这些方法仅在使用ASCII兼容数据时有效，并且以各种其他方式与字符串对象密切相关。

class bytes（[*source* [，*encoding* [，*errors*]]] ）

首先，字节文字的语法与字符串文字的语法基本相同，只是b添加了前缀：

- 单引号： b'still allows embedded "double" quotes'
- 双引号：。b"still allows embedded 'single' quotes"
- 三重引用：，b'''3 single quotes'''b"""3 double quotes"""

只允许以字节为单位的ASCII字符（不管声明的源代码编码如何）。必须使用适当的转义序列将127以上的任何二进制值输入字节文字。

与字符串文字一样，字节文字也可以使用r前缀来禁用转义序列的处理。请参阅字符串和字节文字以获取有关各种形式字节文字的更多信息，包括支持的转义序列。

尽管字节文字和表示是基于ASCII文本的，但字节对象实际上表现得像不可变的整数序列一样，序列中的每个值都受到限制，以致（违反此限制的尝试将触发。这是故意强调的，尽管许多二进制格式包括基于ASCII的元素，并且可以用一些面向文本的算法进行有效的操作，但对于任意二进制数据通常不会这样（将文本处理算法盲目地应用于不兼容ASCII的二进制数据格式通常会导致数据损坏）。0 <= x < 256ValueError

除了文字形式之外，字节对象还可以通过其他方式创建：

- 指定长度的零填充字节对象： `bytes(10)`
- 从可重复的整数： `bytes(range(20))`
- 通过缓冲区协议复制现有的二进制数据： `bytes(obj)`

另请参阅内置的字节。

由于2个十六进制数字精确对应于单个字节，因此十六进制数字是描述二进制数据的常用格式。因此，字节类型具有额外的类方法来读取该格式的数据：

classmethod fromhex (*string*)

这个bytes类方法返回一个字节对象，解码给定的字符串对象。该字符串必须包含每个字节两个十六进制数字，并且忽略ASCII空格。

```
>>> bytes.fromhex('2Ef0 F1f2  ')
b'.\xf0\xf1\xf2'
```

存在反向转换函数将字节对象转换为其十六进制表示形式。

hex ( )

返回包含实例中每个字节的两个十六进制数字的字符串对象。

```
>>> b'\xf0\xf1\xf2'.hex()
'f0f1f2'
```

3.5版本中的新功能。

由于字节对象是整数序列（类似于一个元组），对于一个字节对象*b*，`b[0]`它将是一个整数，而`b[0:1]`它将是一个长度为1的字节对象。（这与文本字符串形成对比，其中索引和切片都会产生一个长度为1的字符串）

字节对象的表示使用字面格式（`b'...'`），因为它通常比例如更有用。您始终可以使用一个字节对象转换为一个整数列表。`bytes([46, 46, 46])list(b)`

注意： 对于Python 2.x用户：在Python 2.x系列中，允许在8位字符串（最接近的2.x提供给内置二进制数据类型）和Unicode字符串之间进行各种隐式转换。这是一个向后兼容的解决方法，以说明Python最初只支持8位文本，而Unicode文本是后来的补充。在Python 3.x中，这些隐式转换消失了 - 8位二进制数据和Unicode文本之间的转换必须是明确的，而字节和字符串对象总是会比较不等。

4.8.2。ByteArray的对象

bytearray对象是对象的可变bytes 对象。

class bytearray ([*source* [, *encoding* [, *errors*]]])

bytearray对象没有专门的文字语法，而是始终通过调用构造函数来创建它们：

- 创建一个空实例： `bytearray()`
- 创建一个给定长度的零填充实例： `bytearray(10)`
- 从可重复的整数： `bytearray(range(20))`
- 通过缓冲区协议复制现有的二进制数据： `bytearray(b'Hi!')`

由于bytearray对象是可变的，除了Bytes和Bytearray操作中描述的常见字节和字节阵列操作外，它们还支持 可变序列操作。

另请参阅内置的bytearray。

由于2个十六进制数字精确对应于单个字节，因此十六进制数字是描述二进制数据的常用格式。因此，bytearray类型有一个额外的类方法来读取该格式的数据：

classmethod fromhex (*string*)

这个bytearray类方法返回bytearray对象，解码给定的字符串对象。该字符串必须包含每个字节两个十六进制数字，并且忽略ASCII空格。

```
>>> bytearray.fromhex('2Ef0 F1f2  ')
bytearray(b'.\xf0\xf1\xf2')
```

存在一个逆转换函数来将一个bytearray对象转换为其十六进制表示。

hex ( )

返回包含实例中每个字节的两个十六进制数字的字符串对象。

```
>>> bytearray(b'\xf0\xf1\xf2').hex()
'f0f1f2'
```

3.5版本中的新功能。

由于bytearray对象是整数序列（类似于列表），对于bytearray对象*b*，`b[0]`它将是一个整数，而`b[0:1]`它将是长度为1的bytearray对象。（与文本字符串形成对比，其中索引和切片都会生成长度为1的字符串）

bytearray对象的表示使用bytes literal格式（`bytearray(b'...')`），因为它通常比eg更有用。您始终可以使用一个bytearray对象转换为一个整数列表。`bytearray([46, 46, 46])list(b)`

4.8.3。字节和Bytearray操作

字节和bytearray对象都支持常见的 序列操作。它们不仅与相同类型的操作数交互操作，而且还与任何类似字节的对象交互操作。由于这种灵活性，它们可以在操作中自由混合而不会造成错误。但是，结果的返回类型可能取决于操作数的顺序。

注意： 字节和bytearray对象上的方法不接受字符串作为它们的参数，就像字符串上的方法不接受字节作为它们的参数一样。例如，你必须写：

```
a = "abc"
b = a.replace("a", "f")
```

和：

```
a = b"abc"
b = a.replace(b"a", b"f")
```

某些字节和字节阵列操作假定使用ASCII兼容的二进制格式，因此在处理任意二进制数据时应该避免使用。这些限制如下。

注意： 使用这些基于ASCII的操作来处理未以ASCII格式存储的二进制数据可能会导致数据损坏。

字节和字节数组对象的以下方法可以与任意二进制数据一起使用。

bytes.count(*sub* [, *start* [, *end*]])
bytearray.count(*sub* [, *start* [, *end*]])

返回范围[*start* , *end*] 中的*子*序列*子*的不重叠次数。可选参数*开始*和*结束*被解释为切片符号。

要搜索的子序列可以是任何类似字节的对象或0到255范围内的整数。

*版本3.3中更改：*还接受范围为0到255的整数作为子序列。

bytes.decode(*encoding="utf-8"* , *errors="strict"*)
bytearray.decode(*encoding="utf-8"* , *errors="strict"*)

返回从给定字节解码的字符串。默认编码是 'utf-8'。可能会给出*错误*来设置不同的错误处理方案。*错误*的默认值是'strict'，这意味着编码错误会引发一个错误UnicodeError。其他可能的值'ignore'，'replace'并通过注册的任何其他名称 codecs.register_error() ，请参见错误处理程序。有关可能的编码列表，请参见标准编码部分。

> **注意：** 将*编码*参数传递给str允许直接解码任何 类似字节的对象，而不需要创建临时字节或字节数组对象。

*在版本3.1中进行了更改：*添加了对关键字参数的支持。

bytes.endswith(*后缀*[, *start* [, *end*]])
bytearray.endswith(*后缀*[, *start* [, *end*]])

返回True如果二进制数据与指定的结束*后缀*，否则返回False。 *后缀*也可以是后缀的元组来查找。随着可选*启动*，测试开始在那个位置。选择*结束时*，停止在该位置进行比较。

搜索的后缀可以是任何类似字节的对象。

bytes.find(*sub* [, *start* [, *end*]])
bytearray.find(*sub* [, *start* [, *end*]])

返回找到*子*序列*子*数据的最低索引，这样*sub*就包含在切片中s[start:end]。可选参数*开始*和*结束*被解释为切片符号。-1如果未找到*子项*，则返回 。

要搜索的子序列可以是任何类似字节的对象或0到255范围内的整数。

> **注意：** find()只有在您需要知道*sub*的位置时才应该使用该方法。要检查*sub*是否是子字符串，请使用 in运算符：
>
> ```
> >>> b'Py' in b'Python'
> True
> ```

*版本3.3中更改：*还接受范围为0到255的整数作为子序列。

bytes.index(*sub* [, *start* [, *end*]])
bytearray.index(*sub* [, *start* [, *end*]])

喜欢find() ，但ValueError在没有找到子序列时引发。

要搜索的子序列可以是任何类似字节的对象或0到255范围内的整数。

*版本3.3中更改：*还接受范围为0到255的整数作为子序列。

bytes.join(*可迭代*)

bytearray.join(*可迭代*)

返回一个字节或bytearray对象，它是*可迭代*的二进制数据序列的串联。TypeError如果*迭代*中有任何不是类字节对象（包括str对象）的值，则会引发A。元素之间的分隔符是提供此方法的字节或bytearray对象的内容。

*静态*bytes.maketrans(*从*，*到*)

*静态*bytearray.maketrans(*从*，*到*)

此静态方法返回一个转换表可用于 bytes.translate()将每个字符映射在*从*成在相同位置处的字符*到*; *从*和*到*必须都是 类似字节的对象，并具有相同的长度。

版本3.1中的新功能。

bytes.partition(*sep*)

bytearray.partition(*sep*)

在*sep*第一次出现时拆分序列，并在分隔符，分隔符本身或其字节阵列副本以及分隔符之后返回包含该部分的3元组。如果找不到分隔符，则返回包含原始序列副本的3元组，然后返回两个空字节或bytearray对象。

要搜索的分隔符可以是任何类似字节的对象。

bytes.replace(*旧的*，*新的*[，*count*])

bytearray.replace(*旧的*，*新的*[，*count*])

返回序列的一个副本，其中所有出现的*旧子*序列都 被*new*替换。如果给出可选的参数*计数*，则仅替换第一个*计数*事件。

搜索和替换的子序列可以是任何 类似字节的对象。

> **注意：** 该方法的bytearray版本不*适用* - 即使没有更改，它总是会生成一个新对象。

bytes.rfind(*sub*[，*start*[，*end*]])

bytearray.rfind(*sub*[，*start*[，*end*]])

返回找到*子*序列*子*序列的最高索引，使*子*包含在其中s[start:end]。可选参数*开始*和*结束*被解释为切片符号。返回 -1失败。

要搜索的子序列可以是任何类似字节的对象或0到255范围内的整数。

*版本3.3中更改：*还接受范围为0到255的整数作为子序列。

bytes.rindex(*sub*[，*start*[，*end*]])

bytearray.rindex(*sub*[，*start*[，*end*]])

像rfind()但ValueError在*子*序列*子*未找到时引发。

要搜索的子序列可以是任何类似字节的对象或0到255范围内的整数。

*版本3.3中更改：*还接受范围为0到255的整数作为子序列。

bytes.rpartition(*sep*)

bytearray.rpartition(*sep*)

在最后出现的*sep*处拆分序列，并在分隔符，分隔符本身或其字节阵列副本以及分隔符之后返回包含该部分的3元组。如果找不到分隔符，则返回包含原始序列副本的3元组，然后返回两个空字节或bytearray对象。

要搜索的分隔符可以是任何类似字节的对象。

bytes.**startswith**(*prefix* [, *start* [, *end*]])
bytearray.**startswith**(*prefix* [, *start* [, *end*]])

返回True如果二进制数据与指定的开始*前缀*，否则返回False。 *前缀*也可以是要查找的前缀元组。随着可选*启动*，测试开始在那个位置。选择*结束时*，停止在该位置进行比较。

要搜索的前缀可以是任何类似字节的对象。

bytes.**translate**(*table* , *delete* = *b''*)
bytearray.**translate**(*table* , *delete* = *b''*)

返回字节或bytearray对象的副本，其中*删除*了可选参数*delete*中出现的所有字节，其余字节已通过给定转换表映射，该转换表必须是长度为256的字节对象。

您可以使用该bytes.maketrans()方法创建转换表。

将*表格*参数设置None为仅用于删除字符的翻译：

```
>>> b'read this short text'.translate(None, b'aeiou')
b'rd ths shrt txt'
```

*在版本3.6中更改：*现在支持将*delete*作为关键字参数。

字节和字节数组对象的以下方法具有假定使用ASCII兼容二进制格式的默认行为，但仍可通过传递适当的参数与任意二进制数据一起使用。请注意，本节中的所有bytearray方法都不*适用*，而是产生新的对象。

bytes.**center**(*width* [, *fillbyte*])
bytearray.**center**(*width* [, *fillbyte*])

返回以长度*宽度*序列为中心的对象的副本。填充是使用指定的填充*字节*完成的（默认是ASCII空间）。对于bytes对象，如果*宽度*小于或等于，则返回原始序列 len(s)。

> **注意：** 该方法的bytearray版本不*适用* - 即使没有更改，它总是会生成一个新对象。

bytes.**ljust**(*width* [, *fillbyte*])
bytearray.**ljust**(*width* [, *fillbyte*])

以长度*宽度*的顺序返回左对齐的对象的副本。填充是使用指定的填充*字节*完成的（默认是ASCII空间）。对于bytes对象，如果*宽度*小于或等于，则返回原始序列 len(s)。

> **注意：** 该方法的bytearray版本不*适用* - 即使没有更改，它总是会生成一个新对象。

bytes.**lstrip**([*chars*])
bytearray.**lstrip**([*chars*])

返回删除指定前导字节的序列副本。的 *字符*参数是指定一组字节值的二进制序列被除去-名称指的是这种方法通常是用ASCII字符使用的事实。如果省略或者None ，*chars*参数默认为删除ASCII空格。该*字符*参数不是前缀; 相反，其值的所有组合都被剥离：

```
>>> b'   spacious   '.lstrip()
b'spacious   '
>>> b'www.example.com'.lstrip(b'cmowz.')
b'example.com'
```

要移除的字节值的二进制序列可以是任何 类似字节的对象。

> **注意:** 该方法的bytearray版本不*适用* - 即使没有更改，它总是会生成一个新对象。

bytes.**rjust**(*width* [, *fillbyte*])
bytearray.**rjust**(*width* [, *fillbyte*])

以长度*宽度*的顺序返回右对齐的对象的副本。填充是使用指定的填充*字节*完成的（默认是ASCII空间）。对于bytes对象，如果*宽度*小于或等于，则返回原始序列 len(s)。

> **注意:** 该方法的bytearray版本不*适用* - 即使没有更改，它总是会生成一个新对象。

bytes.**rsplit**(*sep = None*, *maxsplit = -1*)
bytearray.**rsplit**(*sep = None*, *maxsplit = -1*)

将二进制序列拆分成相同类型的子序列，使用*sep* 作为分隔符字符串。如果给出*maxsplit*，则最多*maxsplit*分裂，*最右边*的分裂。如果没有指定*sep*或者None任何仅由ASCII空白组成的子序列是分隔符。除了从右边分开外，rsplit()其行为 split()如下所述。

bytes.**rstrip**([*chars*])
bytearray.**rstrip**([*chars*])

返回删除指定尾随字节的序列副本。的 *字符*参数是指定一组字节值的二进制序列被除去-名称指的是这种方法通常是用ASCII字符使用的事实。如果省略或者None ，*chars*参数默认为删除ASCII空格。该*字符*参数不是后缀; 相反，其值的所有组合都被剥离：

```
>>> b'   spacious   '.rstrip()
b'   spacious'
>>> b'mississippi'.rstrip(b'ipz')
b'mississ'
```

要移除的字节值的二进制序列可以是任何 类似字节的对象。

> **注意:** 该方法的bytearray版本不*适用* - 即使没有更改，它总是会生成一个新对象。

bytes.**split**(*sep = None*, *maxsplit = -1*)
bytearray.**split**(*sep = None*, *maxsplit = -1*)

将二进制序列拆分成相同类型的子序列，使用*sep* 作为分隔符字符串。如果给出*maxsplit*并且是非负数，则最多 可以完成*maxsplit*分割（因此列表最多只有maxsplit+1 元素）。如果*maxsplit*没有被指定或者是-1，那么分割数量没有限制（所有可能的分割）。

如果给出了*sep*，则连续的分隔符不会分组在一起，并被视为分隔空的子序列（例如，b'1,,2'.split(b',') 返回）。该*月*的参数可以由多字节序列（例如，返回 ）。用指定的分隔符分割空序列会返回或取决于要分割的对象的类型。该*月*的参数可以是任何 类字节对象。[b'1', b'', b'2']b'1<>2<>3'.split(b'<>')[b'1', b'2', b'3'][b''][bytearray(b'')]

例如：

```
>>> b'1,2,3'.split(b',')
[b'1', b'2', b'3']
>>> b'1,2,3'.split(b',', maxsplit=1)
[b'1', b'2,3']
>>> b'1,2,,3,'.split(b',')
[b'1', b'2', b'', b'3', b'']
```

如果未指定*sep*或is None，则应用不同的分割算法：将连续的ASCII空格运行视为单个分隔符，并且如果序列具有前导空格或尾随空格，则结果将在开始或结束处不包含空字符串。因此，分割空白序列或仅由ASCII空白组成的序列而不指定分隔符将返回[]。

例如：

```
>>> b'1 2 3'.split()
[b'1', b'2', b'3']
>>> b'1 2 3'.split(maxsplit=1)
[b'1', b'2 3']
>>> b'   1   2   3   '.split()
[b'1', b'2', b'3']
```

bytes.**strip**([*chars*])
bytearray.**strip**([*chars*])

返回删除指定前导字节和尾随字节的副本。的*字符*参数是指定一组字节值的二进制序列被除去-名称指的是这种方法通常是用ASCII字符使用的事实。如果省略或者None，*chars* 参数默认为删除ASCII空格。该*字符*参数不是前缀或后缀; 相反，其值的所有组合都被剥离：

```
>>> b'   spacious   '.strip()
b'spacious'
>>> b'www.example.com'.strip(b'cmowz.')
b'example'
```

要移除的字节值的二进制序列可以是任何 类似字节的对象。

> **注意:** 该方法的bytearray版本不*适用* - 即使没有更改，它总是会生成一个新对象。

字节和字节数组对象的以下方法假定使用ASCII兼容的二进制格式，不应将其应用于任意二进制数据。请注意，本节中的所有bytearray方法都不*适用*，而是产生新的对象。

bytes.**capitalize**( )
bytearray.**capitalize**( )

返回序列的一个副本，每个字节解释为一个ASCII字符，第一个字节大写，其余的小写。非ASCII字节值通过不变。

> **注意:** 该方法的bytearray版本不*适用* - 即使没有更改，它总是会生成一个新对象。

bytes.**expandtabs**(*tabsize = 8*)
bytearray.**expandtabs**(*tabsize = 8*)

根据当前列和给定的选项卡大小，将所有ASCII选项卡字符替换为一个或多个ASCII空格的序列副本返回。选项卡位置出现在每个*制表符大小*字节中（默认值为8，在列*0,8,16*等处提供制表位置）。要扩展序列，将当前列设置为零，并逐个字节地检查序列。如果字节是ASCII制表符（b'\t'），则在结果中插入一个或多个空格字符，直到当前列等于下一个制表符位置。（制表符本身不被复制。）如果当前字节是ASCII换行符（b'\n'）或回车符（b'\r'），它被复制并且当前列被重置为零。不管打印时字节值的表示方式如何，任何其他字节值都将保持不变并且当前列增加1：

```
>>> b'01\t012\t0123\t01234'.expandtabs()
b'01      012     0123    01234'
>>> b'01\t012\t0123\t01234'.expandtabs(4)
b'01  012 0123    01234'
```

注意: 该方法的bytearray版本不*适用* - 即使没有更改，它总是会生成一个新对象。

bytes.**isalnum ( )**
bytearray.**isalnum ( )**

如果序列中的所有字节均为字母ASCII字符或ASCII十进制数字，且序列不为空，则返回true，否则返回false。按字母顺序排列的ASCII字符是序列中的那些字节值b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'。ASCII十进制数字是序列中的那些字节值b'0123456789'。

例如：

```
>>> b'ABCabc1'.isalnum()
True
>>> b'ABC abc1'.isalnum()
False
```

bytes.**isalpha ( )**
bytearray.**isalpha ( )**

如果序列中的所有字节均为字母ASCII字符且序列不为空，则返回true，否则返回false。按字母顺序排列的ASCII字符是序列中的那些字节值b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'。

例如：

```
>>> b'ABCabc'.isalpha()
True
>>> b'ABCabc1'.isalpha()
False
```

bytes.**isdigit ( )**
bytearray.**isdigit ( )**

如果序列中的所有字节均为ASCII十进制数字，且序列不为空，则返回true，否则返回false。ASCII十进制数字是序列中的那些字节值b'0123456789'。

例如：

```
>>> b'1234'.isdigit()
True
>>> b'1.23'.isdigit()
False
```

bytes.**islower ( )**
bytearray.**islower ( )**

如果序列中至少有一个小写ASCII字符，并且没有大写ASCII字符，则返回true，否则返回false。

例如：

```
>>> b'hello world'.islower()
True
>>> b'Hello world'.islower()
False
```

小写ASCII字符是序列中的那些字节值 b'abcdefghijklmnopqrstuvwxyz'。大写ASCII字符是序列中的那些字节值b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'。

bytes.**isspace**( )
bytearray.**isspace**( )

如果序列中的所有字节均为ASCII空格且序列不为空，则返回true，否则返回false。ASCII空格字符是序列中的那些字节值（空格，制表符，换行符，回车符，垂直制表符，换页符）。 b'\t\n\r\x0b\f'

bytes.**istitle**( )
bytearray.**istitle**( )

如果序列是ASCII标题并且序列不为空，则返回true，否则返回false。请参阅bytes.title() 关于“titlecase”定义的更多细节。

例如：

```
>>> b'Hello World'.istitle()
True
>>> b'Hello world'.istitle()
False
```

bytes.**isupper**( )
bytearray.**isupper**( )

如果序列中至少有一个大写字母ASCII字符，并且没有小写ASCII字符，则返回true，否则返回false。

例如：

```
>>> b'HELLO WORLD'.isupper()
True
>>> b'Hello world'.isupper()
False
```

小写ASCII字符是序列中的那些字节值 b'abcdefghijklmnopqrstuvwxyz'。大写ASCII字符是序列中的那些字节值b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'。

bytes.**lower**( )
bytearray.**lower**( )

将所有大写ASCII字符转换为其对应的小写字母的序列的副本。

例如：

```
>>> b'Hello World'.lower()
b'hello world'
```

小写ASCII字符是序列中的那些字节值 b'abcdefghijklmnopqrstuvwxyz'。大写ASCII字符是序列中的那些字节值b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'。

> **注意：** 该方法的bytearray版本不*适用* - 即使没有更改，它总是会生成一个新对象。

bytes.**splitlines**(*keepends* = *False*)
bytearray.**splitlines**(*keepends* = *False*)

返回二进制序列中的行列表，以ASCII行边界为中心。此方法使用通用换行符方法来分割线条。除非换行符不包括在结果列表中*keepends*，并给出正确的。

例如：

```
>>> b'ab c\n\nde fg\rkl\r\n'.splitlines()
[b'ab c', b'', b'de fg', b'kl']
>>> b'ab c\n\nde fg\rkl\r\n'.splitlines(keepends=True)
[b'ab c\n', b'\n', b'de fg\r', b'kl\r\n']
```

与给定`split()`分隔符字符串*sep*时不同，此方法返回空字符串的空列表，并且终端行分隔符不会导致多余的行：

```
>>> b"".split(b'\n'), b"Two lines\n".split(b'\n')
([b''], [b'Two lines', b''])
>>> b"".splitlines(), b"One line\n".splitlines()
([], [b'One line'])
```

bytes.**swapcase**( )

bytearray.**swapcase**( )

将所有小写ASCII字符转换为相应的大写字母，反之亦然。

例如：

```
>>> b'Hello World'.swapcase()
b'hELLO wORLD'
```

小写ASCII字符是序列中的那些字节值 `b'abcdefghijklmnopqrstuvwxyz'`。大写ASCII字符是序列中的那些字节值`b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`。

不同的`str.swapcase()`是，对于二进制版本，情况总是如此 。大小写转换在ASCII中是对称的，即使对于任意的Unicode代码点通常不是这样。 `bin.swapcase().swapcase() == bin`

> **注意:** 该方法的bytearray版本不*适用* - 即使没有更改，它总是会生成一个新对象。

bytes.**title**( )

bytearray.**title**( )

返回二进制序列的标题版本，其中单词以大写ASCII字符开头，其余字符为小写。Uncased字节值保持不变。

例如：

```
>>> b'Hello world'.title()
b'Hello World'
```

小写ASCII字符是序列中的那些字节值 `b'abcdefghijklmnopqrstuvwxyz'`。大写ASCII字符是序列中的那些字节值`b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`。所有其他字节值都是uncased。

该算法使用简单的与语言无关的单词作为连续字母组的定义。该定义在许多情况下都有效，但这意味着收缩和占有者中的撇号会形成单词边界，这可能不是理想的结果：

```
>>> b"they're bill's friends from the UK".title()
b"They'Re Bill'S Friends From The Uk"
```

可以使用正则表达式构造撇号的解决方法：

```
>>> import re
>>> def titlecase(s):
...     return re.sub(rb"[A-Za-z]+('[A-Za-z]+)?",
...                   lambda mo: mo.group(0)[0:1].upper() +
...                              mo.group(0)[1:].lower(),
...                   s)
...
>>> titlecase(b"they're bill's friends.")
b"They're Bill's Friends."
```

注意： 该方法的bytearray版本不*适用* - 即使没有更改，它总是会生成一个新对象。

bytes.**upper**( )
bytearray.**upper**( )

将所有小写ASCII字符转换为其对应的大写字母的序列的副本。

例如：

```
>>> b'Hello World'.upper()
b'HELLO WORLD'
```

小写ASCII字符是序列中的那些字节值 b'abcdefghijklmnopqrstuvwxyz'。大写ASCII字符是序列中的那些字节值b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'。

注意： 该方法的bytearray版本不*适用* - 即使没有更改，它总是会生成一个新对象。

bytes.**zfill**(*宽度*)
bytearray.**zfill**(*宽度*)

返回左边填充ASCII b'0'数字的序列的副本，以制作一个长度*宽度*的序列。前导符号前缀（b'+'/b'-'是通过在符号字符*之后*插入填充符来处理的，对于bytes对象，如果*宽度*小于或等于，则返回原始序列len(seq)。

例如：

```
>>> b"42".zfill(5)
b'00042'
>>> b"-42".zfill(5)
b'-0042'
```

注意： 该方法的bytearray版本不*适用* - 即使没有更改，它总是会生成一个新对象。

4.8.4。printf风格字节格式

注意： 这里描述的格式化操作展现了各种各样的怪癖，导致了许多常见错误（例如，未能正确显示元组和字典）。如果要打印的值可能是元组或字典，请将其包装在一个元组中。

字节对象（bytes/bytearray）有一个独特的内置操作：%操作符（模）。这也被称为字节*格式化*或*插值*运算符。鉴于（其中*format*是一个字节对象），*格式*中的转换规范将被替换为零个或多个*值*元素。其效果与在C语言中使用类似。format % values%sprintf()

如果*格式*需要单个参数，则*值*可能是单个非元组对象。[5] 否则，*值*必须是具有由格式字节对象或单个映射对象（例如字典）指定的项目数量的元组。

转换说明符包含两个或多个字符，并具有以下组件，它们必须按以下顺序出现：

1. 该'%'字符表示说明符的开始。
2. 映射键（可选），由括号括起来的字符序列组成（例如(somename)）。
3. 转换标志（可选），影响某些转换类型的结果。
4. 最小字段宽度（可选）。如果指定为'*'（星号），则将从元组的下一个元素*值中*读取实际宽度，并且要转换的对象位于最小字段宽度和可选精度之后。
5. 精度（可选），以'.'（点）形式给出，然后是精度。如果指定为'*'（星号），则实际精度将从元组的下一个元素的*值中*读取，并且要转换的值位于精度之后。
6. 长度修饰符（可选）。
7. 转换类型。

当正确的参数是一个字典（或其他映射类型）时，字节对象中的格式*必须*在该字典后面插入一个带括号的映射关键字'%'。映射键从映射中选择要格式化的值。例如：

```
>>> print(b'%(language)s has %(number)03d quote types.' %
...       {b'language': b"Python", b"number": 2})
b'Python has 002 quote types.'
```

在这种情况下*，格式中不会出现说明符（因为它们需要一个连续的参数列表）。

转换标志字符是：

旗	含义
'#'	值转换将使用“替代形式”（下面定义）。
'0'	转换将为数字值填充零。
'-'	转换后的值是左侧调整的（'0'如果两者都给出，则覆盖转换）。
' '	换-一个空格）在由正确的数字（或空字符串）产生的一个空白应该留下一个有符号的转
'+'	符号字符（'+'或'-'）将在转换之前（覆盖“空格”标志）。

长度修饰符（h，l或L）可能存在，但会被忽略，因为它对于Python不是必需的 - 例如与之%ld相同%d。

转换类型是：

转变	含义	笔记
'd'	带符号的整数小数。	
'i'	带符号的整数小数。	
'o'	签署八进制值。	（1）
'u'	已过时的类型 - 与之相同'd'。	（8）
'x'	签名的十六进制（小写）。	（2）
'X'	签名的十六进制（大写）。	（2）
'e'	浮点指数格式（小写）。	（3）
'E'	浮点指数格式（大写）。	（3）
'f'	浮点十进制格式。	（3）
'F'	浮点十进制格式。	（3）
'g'	浮点格式。如果指数小于-4或小于精度，则使用小写指数格式，否则使用	（4）
'G'	使用指数格式。如果指数小于-4或者不小于精度，则使用大写指数格式，否则	（4）
'c'	单字节（接受整数或单字节对象）。	

'b'	字节（任何遵循 缓冲区协议或具有的对象 __bytes__() ）。	（5）
's'	's' 是'b' Python2 / 3代码库的别名，应该仅用于Python2 / 3代码库。	（6）
'a'	repr(obj).encode('ascii','backslashreplace) ）。	（5）
'r'	'r' 是'a' Python2 / 3代码库的别名，应该仅用于Python2 / 3代码库。	（7）
'%'	没有参数被转换，导致结果中的'%' 字符。	

笔记：

1. 替代形式导致'0o' 在第一个数字之前插入前导八进制说明符（ ）。
2. 替代形式会导致在第一个数字之前插入'0x' 或'0X' 取决于是否使用了'x' 或'X' 格式。
3. 替代形式会导致结果始终包含小数点，即使没有数字跟着它。

   精度决定小数点后的位数，默认为6。
4. 替代形式导致结果始终包含小数点，并且尾部零不会像原本那样被删除。

   精度决定小数点前后的有效位数，默认值为6。
5. 如果精度是N，则输出被截断为N字符。
6. b'%s' 已弃用，但在3.x系列期间不会被删除。
7. b'%r' 已弃用，但在3.x系列期间不会被删除。
8. 看到 **PEP 237**。

注意: 该方法的bytearray版本不*适用* - 即使没有更改，它总是会生成一个新对象。

也可以看看: **PEP 461**。

3.5版本中的新功能。

4.8.5。内存视图

memoryview对象允许Python代码访问支持缓冲协议的对象的内部数据，而无需复制。

class memoryview（ *obj* ）

创建一个memoryview引用*obj*。 *obj*必须支持缓冲区协议。支持缓冲协议的内置对象包括 bytes和bytearray。

A memoryview具有*元素*的概念，*元素*是由原始对象*obj*处理的原子内存单元。对于许多简单的类型，例如bytes和bytearray，元素是单个字节，但其他类型array.array可能会有更大的元素。

len(view)等于长度tolist。如果，长度为1.如果长度等于视图中元素的数量。对于更高维度，长度等于视图的嵌套列表表示的长度。该属性将为您提供单个元素中的字节数。 view.ndim = 0view.ndim = 1itemsize

A memoryview支持切片和索引以显示其数据。一维切片将产生一个子视图：

```
>>> v = memoryview(b'abcefg')
>>> v[1]
98
>>> v[-1]
103
>>> v[1:4]
<memory at 0x7f3ddc9f4350>
>>> bytes(v[1:4])
b'bce'
```

如果format是来自struct模块的本地格式说明符之一，则还支持使用整数或整数元组进行索引，并返回具有正确类型的单个*元素*。一维记忆体视图可以用一个整数或一个整数元组索引。多维内存视图可以用精确*ndim*整数的元组索引，其中*ndim*是维数。零维内存视图可以使用空元组索引。

这是一个非字节格式的例子：

```
>>> import array
>>> a = array.array('l', [-11111111, 22222222, -33333333, 44444444])
>>> m = memoryview(a)
>>> m[0]
-11111111
>>> m[-1]
44444444
>>> m[::2].tolist()
[-11111111, -33333333]
```

如果底层对象是可写的，则内存视图支持一维切片分配。不允许调整大小：

```
>>> data = bytearray(b'abcefg')
>>> v = memoryview(data)
>>> v.readonly
False
>>> v[0] = ord(b'z')
>>> data
bytearray(b'zbcefg')
>>> v[1:4] = b'123'
>>> data
bytearray(b'z123fg')
>>> v[2:3] = b'spam'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: memoryview assignment: lvalue and rvalue have different structures
>>> v[2:6] = b'spam'
>>> data
bytearray(b'z1spam')
```

具有格式“B”，“b”或“c”的可哈希（只读）类型的一维记忆视图也是可散列的。哈希被定义为 ：`hash(m) == hash(m.tobytes())`

```
>>> v = memoryview(b'abcefg')
>>> hash(v) == hash(b'abcefg')
True
>>> hash(v[2:4]) == hash(b'ce')
True
>>> hash(v[::-2]) == hash(b'abcefg'[::-2])
True
```

*在版本3.3中更改：*现在可以切割一维记忆体视图。格式为'B'，'b'或'c'的一维记忆体现在是可散列的。

在版本3.4中更改： memoryview现在使用自动注册 `collections.abc.Sequence`

*在版本3.5中进行了更改：*现在可以使用整数元组对索引进行索引。

`memoryview` 有几种方法：

__eq__（*出口商*）

一个内存视图和一个 如果它们的形状是相同的，并且如果所有相应的值在操作数各自的格式代码使用`struct`语法进行解释时是相等的，则 **PEP 3118**导出器是相等的。

对于子集`struct`的格式字符串目前支持的 `tolist()` ，`v`并且`w`是相等的，如果：`v.tolist() == w.tolist()`

```
>>> import array
>>> a = array.array('I', [1, 2, 3, 4, 5])
>>> b = array.array('d', [1.0, 2.0, 3.0, 4.0, 5.0])
>>> c = array.array('b', [5, 3, 1])
>>> x = memoryview(a)
>>> y = memoryview(b)
>>> x == a == y == b
True
>>> x.tolist() == a.tolist() == y.tolist() == b.tolist()
True
>>> z = y[::-2]
>>> z == c
True
>>> z.tolist() == c.tolist()
True
```

如果`struct`模块不支持任何一种格式的字符串，那么这些对象总会被比较为不相等（即使格式字符串和缓冲区内容相同）：

```
>>> from ctypes import BigEndianStructure, c_long
>>> class BEPoint(BigEndianStructure):
...     _fields_ = [("x", c_long), ("y", c_long)]
...
>>> point = BEPoint(100, 200)
>>> a = memoryview(point)
>>> b = memoryview(point)
>>> a == point
False
>>> a == b
False
```

需要注意的是，如浮点数，并*没有*暗示 对memoryview对象。`v is wv == w`

*版本3.3中的更改：*以前的版本比较了不考虑项目格式和逻辑阵列结构的原始内存。

tobytes（）

以字节串形式返回缓冲区中的数据。这相当于`bytes`在memoryview上调用构造函数。

```
>>> m = memoryview(b"abc")
>>> m.tobytes()
b'abc'
```

```
>>> bytes(m)
b'abc'
```

对于非连续数组，结果等于所有元素转换为字节的展平列表表示。tobytes() 支持所有格式字符串，包括那些不在struct模块语法中的字符串 。

hex ( )

返回包含缓冲区中每个字节的两个十六进制数字的字符串对象。

```
>>> m = memoryview(b"abc")
>>> m.hex()
'616263'
```

3.5版本中的新功能。

tolist ( )

将缓冲区中的数据作为元素列表返回。

```
>>> memoryview(b'abc').tolist()
[97, 98, 99]
>>> import array
>>> a = array.array('d', [1.1, 2.2, 3.3])
>>> m = memoryview(a)
>>> m.tolist()
[1.1, 2.2, 3.3]
```

版本3.3中的更改： tolist()现在支持struct模块语法中的所有单字符本机格式 以及多维表示。

release ( )

释放由memoryview对象公开的底层缓冲区。许多对象在对其进行查看时会采取特殊的操作（例如，a bytearray会暂时禁止调整大小）；因此，调用release ( ) 可以方便地尽快删除这些限制（并释放任何悬挂的资源）。

在这个方法被调用之后，对视图的任何进一步操作都会引发一个ValueError（除了release()它本身可以被多次调用的）：

```
>>> m = memoryview(b'abc')
>>> m.release()
>>> m[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operation forbidden on released memoryview object
```

上下文管理协议可以用于类似的效果，使用with语句：

```
>>> with memoryview(b'abc') as m:
...     m[0]
...
97
>>> m[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operation forbidden on released memoryview object
```

3.2版本中的新功能

cast（*格式*[，*形状*]）

将内存视图转换为新的格式或形状。*形状*默认为 [byte_length//new_itemsize]，这意味着结果视图将是一维的。返回值是一个新的内存视图，但缓冲区本身未被复制。支持的演员阵容是1D - > C- 连续 和C连续 - > 1D。

目标格式在 struct 语法上仅限于单个元素原生格式 。其中一种格式必须是字节格式（'B'，'b'或'c'）。结果的字节长度必须与原始长度相同。

投1D /长到1D /无符号字节：

```
>>> import array
>>> a = array.array('l', [1,2,3])
>>> x = memoryview(a)
>>> x.format
'l'
>>> x.itemsize
8
>>> len(x)
3
>>> x.nbytes
24
>>> y = x.cast('B')
>>> y.format
'B'
>>> y.itemsize
1
>>> len(y)
24
>>> y.nbytes
24
```

将1D /无符号字节转换为1D / char：

```
>>> b = bytearray(b'zyz')
>>> x = memoryview(b)
>>> x[0] = b'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: memoryview: invalid value for format "B"
>>> y = x.cast('c')
>>> y[0] = b'a'
>>> b
bytearray(b'ayz')
```

将1D /字节转换为3D /整数至1D / signed char：

```
>>> import struct
>>> buf = struct.pack("i"*12, *list(range(12)))
>>> x = memoryview(buf)
>>> y = x.cast('i', shape=[2,2,3])
>>> y.tolist()
[[[0, 1, 2], [3, 4, 5]], [[6, 7, 8], [9, 10, 11]]]
>>> y.format
'i'
>>> y.itemsize
4
>>> len(y)
2
>>> y.nbytes
```

```
48
>>> z = y.cast('b')
>>> z.format
'b'
>>> z.itemsize
1
>>> len(z)
48
>>> z.nbytes
48
```

将1D /无符号字符转换为2D /无符号长整型：

```
>>> buf = struct.pack("L"*6, *list(range(6)))
>>> x = memoryview(buf)
>>> y = x.cast('L', shape=[2,3])
>>> len(y)
2
>>> y.nbytes
48
>>> y.tolist()
[[0, 1, 2], [3, 4, 5]]
```

3.3版本的新功能

*在版本3.5中更改：*转换为字节视图时，源格式不再受到限制。

还有几个只读属性可用：

obj

内存视图的基础对象：

```
>>> b  = bytearray(b'xyz')
>>> m = memoryview(b)
>>> m.obj is b
True
```

3.3版本的新功能

nbytes

`nbytes == product(shape) * itemsize == len(m.tobytes())`。这是数组在连续表示中使用的空间数量。它不一定等于len（m）：

```
>>> import array
>>> a = array.array('i', [1,2,3,4,5])
>>> m = memoryview(a)
>>> len(m)
5
>>> m.nbytes
20
>>> y = m[::2]
>>> len(y)
3
>>> y.nbytes
12
>>> len(y.tobytes())
12
```

多维数组：

```
>>> import struct
>>> buf = struct.pack("d"*12, *[1.5*x for x in range(12)])
>>> x = memoryview(buf)
>>> y = x.cast('d', shape=[3,4])
>>> y.tolist()
[[0.0, 1.5, 3.0, 4.5], [6.0, 7.5, 9.0, 10.5], [12.0, 13.5, 15.0, 16.5]]
>>> len(y)
3
>>> y.nbytes
96
```

3.3版本的新功能

readonly

指示内存是否只读的布尔值。

format

包含struct视图中每个元素的格式（模块样式）的字符串。内存视图可以使用任意格式的字符串从导出器创建，但某些方法（例如tolist()）仅限于本地单个元素格式。

*版本3.3中更改：*格式'B'现在根据结构模块语法进行处理。这意味着。memoryview(b'abc')[0] == b'abc'[0] == 97

itemsize

memoryview每个元素的大小（以字节为单位）：

```
>>> import array, struct
>>> m = memoryview(array.array('H', [32000, 32001, 32002]))
>>> m.itemsize
2
>>> m[0]
32000
>>> struct.calcsize('H') == m.itemsize
True
```

ndim

一个整数，指示内存所代表的多维数组的维数。

shape

一个整数元组，ndim表示将存储器的形状作为一个N维数组的长度。

*版本3.3中更改：*一个空元组，而不是None当ndim = 0时。

strides

整数元组的长度，ndim以字节为单位给出数组中每个元素的每个元素的长度。

*版本3.3中更改：*一个空元组，而不是None当ndim = 0时。

suboffsets

在内部用于PIL风格的阵列。该值仅供参考。

c_contiguous

一个bool指示内存是否是C- 连续的。

3.3版本的新功能

`f_contiguous`
一个bool指示内存是否是Fortran 连续的。

3.3版本的新功能

`contiguous`
指示内存是否连续的 bool 。

3.3版本的新功能

4.9。设置类型 - `set` ， `frozenset`

甲*组*对象是不同的无序集合可哈希对象。常见用途包括成员资格测试，删除序列中的重复项，以及计算数学运算，如交集，联合，差异和对称差异。（对于其它容器看到内置的`dict` ， `list`和`tuple`类和`collections`模块）。

像其他收藏品，集支持，和。作为无序集合，集合不会记录元素位置或插入顺序。因此，集合不支持索引，切片或其他类序列行为。`x in set` `len(set)` `for x in set`

目前有两个内置集类型，`set`和`frozenset`。该`set`类型是可变的 - 可以使用类似`add()`和的方法更改内容`remove()`。由于它是可变的，因此它没有散列值，不能用作字典键或另一个集的元素。该`frozenset`类型是不可变的且可拆分的 - 其内容在创建后不能更改; 因此它可以用作字典键或另一组元素。

非空集（非frozensets）可以通过在花括号中放置逗号分隔的元素列表来创建，例如：，除了 构造函数。`{'jack', 'sjoerd'}` `set`

这两个类的构造函数都是一样的：

class `set([iterable])`
class `frozenset([iterable])`
返回其元素从*迭代*中获取的新集或冻结集对象 。一组元素必须是可散列的。为了表示集合集合，内部集合必须是`frozenset` 对象。如果未指定*iterable*，则返回一个新的空集。

实例`set`并`frozenset`提供以下操作：

`len(s)`
返回集合的元素数*小号*（基数*小号*）。

`x in s`
测试*x*在*s中的*成员身份。

`x not in s`
测试*x*对于*s中的*非成员资格。

`isdisjoint(其他)`
`True`如果集合没有与*其他*元素相同的元素，则返回。当且仅当它们的交集是空集时，集合才是不相交的。

`issubset(其他)`
`set <= other`
测试集合中的每个元素是否在*其他*元素中。

`set < other`
测试该集合是否是*其他*集合的正确子集，也就是说。`set <= other and set != other`

`issuperset(其他)`
`set >= other`
测试*其他*元素中的每个元素是否在集合中。

`set > other`
测试集合是否是*其他*集合的正确集合，也就是说。`set >= other and set != other`

`union(*其他)`
`set | other | ...`
用集合中的元素和所有其他元素返回一个新集合。

`intersection(*其他)`
`set & other & ...`
返回一个新集合，其中包含该集合和其他所有元素共有的元素。

`difference(*其他)`
`set - other - ...`
返回集合中不包含其他元素的新集合。

`symmetric_difference(其他)`
`set ^ other`
用集合中的元素或*其他*元素返回一个新集合，但不能同时返回两个元素。

`copy()`
用s的浅拷贝返回一个新的集合。

注意，对非运营商的版本`union()`，`intersection()`，`difference()`，和`symmetric_difference()`，`issubset()`和`issuperset()`方法将接受任何可迭代作为参数。相反，他们的基于操作员的对应方要求他们的参数是集合。这排除了易于出错的结构，例如 有利于更具可读性的结构。`set('abc') & 'cbs' set('abc').intersection('cbs')`

双方`set`并`frozenset`支持设置来设置比较。当且仅当每个集合中的每个元素都包含在另一个中（每个元素是另一个的子集）时，两个集合是相等的。当且仅当第一组是第二组的合适子集（是子集，但不相等）时，集合小于另一集合。当且仅当第一个集合是第二个集合的适当超集（是超集，但不相等）时，集合比另一集合大。

的实例`set`进行比较的情况下，`frozenset` 根据自己的成员。例如， 返回等等。`set('abc') == frozenset('abc') True set('abc') in set([frozenset('abc')])`

子集和等式比较不推广到总排序函数。例如，任何两个非空不相交的集合不相等，并且不彼此的子集，所以*所有*的以下返回`False`：`a<b`， `a==b`，或`a>b`。

由于集合只定义了部分排序（子集关系），所以该`list.sort()`方法的输出对于集合列表是未定义的。

设置元素，如字典键，必须是可散列的。

混合`set`实例的二进制操作`frozenset` 返回第一个操作数的类型。例如：返回一个实例。`frozenset('ab') | set('bc') frozenset`

下表列出了`set`不适用于以下不可变实例的可用操作`frozenset`：

update (*其他*)
set |= other | ...
更新集合，添加所有其他元素。

intersection_update (*其他*)
set &= other & ...
更新集合，只保留其中的元素和其他所有元素。

difference_update (*其他*)
set -= other | ...
更新设置，删除其他设备中的元素。

symmetric_difference_update (*其他*)
set ^= other
更新集合，只保留任一集合中的元素，但不能同时存在于两者中。

add (*elem*)
将元素*elem*添加到集合中。

remove (*elem*)
从集合中删除元素*elem*。KeyError如果*元素*不包含在集合中则引发。

discard (*elem*)
如果它存在，则从集合中移除元素*elem*。

pop ( )
删除并返回该集合中的任意元素。KeyError如果该集合为空，则引发 。

clear ( )
删除集合中的所有元素。

需要注意的的非运营商的版本update() ， intersection_update() ，difference_update() ，和symmetric_difference_update()方法会接受任何迭代器作为参数。

请注意，该*ELEM*参数的__contains__() ，remove()和 discard()方法可能是一组。为了支持搜索等效的冷冻集，从*elem*创建一个临时集。

4.10。映射类型 - dict

甲映射对象映射可哈希值到任意对象。映射是可变对象。目前只有一种标准映射类型，即*字典*。（对于其它容器看到内置的 list ，set和tuple类和 collections模块）。

字典的键*几乎是*任意值。不可 散列的值，即包含列表，字典或其他可变类型（通过值而不是对象标识进行比较）的值不能用作关键字。用于键的数字类型服从数字比较的正常规则：如果两个数字比较相等（如1和1.0），则它们可以互换使用以索引相同的字典条目。（但请注意，由于计算机将浮点数字存储为近似值，因此将它们用作字典键通常是不明智的。）

字典可以通过 在花括号中放置逗号分隔的列表来创建，例如：或者，或者通过构造函数。key: value {'jack': 4098, 'sjoerd': 4127} {4098: 'jack', 4127: 'sjoerd'} dict

class dict (** *kwarg*)
*类*dict (*映射*, ** *kwarg*)

*类*dict（*可迭代的*，***kwarg*）

返回从可选的位置参数和可能为空的关键字参数集合初始化的新字典。

如果没有给出位置参数，则创建空字典。如果给出了位置参数并且它是一个映射对象，则将使用与映射对象相同的键值对创建一个字典。否则，位置参数必须是可迭代的对象。迭代器中的每个项目本身必须是一个具有两个对象的迭代器。每个项目的第一个对象成为新字典中的一个键，第二个对象成为相应的值。如果某个键出现多次，则该键的最后一个值将成为新词典中的对应值。

如果给出关键字参数，则将关键字参数及其值添加到从位置参数创建的字典中。如果添加的键已经存在，那么来自关键字参数的值将替换位置参数中的值。

为了说明，下面的例子都返回一个字典，等于：{"one": 1, "two": 2, "three": 3}

```
>>> a = dict(one=1, two=2, three=3)
>>> b = {'one': 1, 'two': 2, 'three': 3}
>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
>>> d = dict([('two', 2), ('one', 1), ('three', 3)])
>>> e = dict({'three': 3, 'one': 1, 'two': 2})
>>> a == b == c == d == e
True
```

如第一个示例中提供的关键字参数仅适用于有效Python标识符的键。否则，可以使用任何有效的密钥。

这些是词典支持的操作（因此，自定义映射类型也应该支持）：

len(d)

返回字典中的项目数量*d*。

d[key]

用钥匙*鍵*返回*d*的项目。引发一个if *鍵*不在地图中。KeyError

如果dict的一个子类定义了一个方法__missing__()并且*鍵*不存在，那么该d[key]操作将使用该键的*鍵*作为参数来调用该方法。d[key]然后该操作返回或提出__missing__(key)通话所返回或提出的任何内容。没有其他操作或方法调用__missing__()。如果 __missing__()没有定义，KeyError则提出。 __missing__()必须是一种方法; 它不能是一个实例变量：

```
>>> class Counter(dict):
...     def __missing__(self, key):
...         return 0
>>> c = Counter()
>>> c['red']
0
>>> c['red'] += 1
>>> c['red']
1
```

上面的例子显示了部分实现 collections.Counter。不同的__missing__方法被使用collections.defaultdict。

d[key] = value

设置d[key]为*值*。

del d[key]

d[key]从*d*中删除。引发一个KeyErrorif *鍵*不在地图中。

`key in d`
返回，`True`如果*d*有一个关键的*键*，否则`False`。

`key not in d`
相当于。`not key in d`

`iter(d)`
返回字典键的迭代器。这是一个捷径`iter(d.keys())`。

`clear ( )`
从字典中删除所有项目。

`copy ( )`
返回字典的浅表副本。

classmethod `fromkeys ( seq [ , value ] )`
使用来自*seq的*键和值设置为*值*创建一个新字典。

`fromkeys()`是一个返回新字典的类方法。*值* 默认为`None`。

`get ( key [ , default ] )`
如果*键*在字典中，则返回*键*的值，否则返回*默认值*。如果*默认*没有给出，则默认为，所以，这种方法从未提出了一个。`None``KeyError`

`items ( )`
返回字典项目（对）的新视图。查看视图对象的文档。`(key, value)`

`keys ( )`
返回字典密钥的新视图。查看视图对象的文档。

`pop ( key [ , default ] )`
如果*键*在字典中，请将其删除并返回其值，否则返回 *默认值*。如果未给出*缺省值*并且*键*不在字典中，`KeyError`则会引发a。

`popitem ( )`
从字典中删除并返回任意一对。`(key, value)`

`popitem()`对于在字典中进行破坏性迭代很有用，正如集合算法中经常使用的那样。如果字典是空的，调用 `popitem()`引发a `KeyError`。

`setdefault ( key [ , default ] )`
如果*密钥*在字典中，则返回其值。如果没有，则插入 具有*默认*值的*键*并返回*默认值*。 *默认*默认为 。`None`

`update ( [ other ] )`
更新与来自键/值对字典*等*，覆盖现有的密钥。返回`None`。

`update()`接受另一个字典对象或者键/值对的迭代（作为元组或其他长度为2的迭代）。如果指定了关键字参数，则字典随后会使用这些键/值对进行更新：。`d.update(red=1, blue=2)`

`values ( )`
返回字典值的新视图。查看视图对象的 文档。

当且仅当它们具有相同的对时，字典比较相等。订单比较（'<'，'<='，'> ='，'>'）提高 。(key, value)TypeError

也可以看看： types.MappingProxyType可以用来创建a的只读视图dict。

4.10.1。字典视图对象

返回的对象的dict.keys()，dict.values()并且 dict.items()是*视图对象*。它们提供了有关字典条目的动态视图，这意味着当字典更改时，视图反映了这些更改。

字典视图可以迭代以产生它们各自的数据，并支持成员资格测试：

len(dictview)
返回字典中的条目数量。

iter(dictview)
返回字典中的键，值或项目（表示为元组）的迭代器 。(key, value)

键和值以非随机的任意顺序迭代，在Python实现中有所不同，并取决于字典的插入和删除历史。如果按键，值和项目视图被重复执行而不对词典进行中间修改，则项目顺序将直接对应。这允许使用以下内容创建对：。另一种创建相同列表的方法是。(value, key)zip()pairs = zip(d.values(), d.keys())pairs = [(v, k) for (k, v) in d.items()]

在添加或删除字典中的条目时迭代视图可能会引起RuntimeError或无法迭代所有条目。

x in dictview
返回True如果*x*是在底层的字典的键，值或项（在后一种情况下，*x*应是一个元组）。(key, value)

按键视图像集合一样，因为它们的条目是独特且可散列的。如果所有的值都是可散列的，那么对就是唯一且可散列的，那么项目视图也是类似的。（因为条目通常不是唯一值的观点不被视为设置状）。对于组样的观点，都为抽象基类中定义的操作的是可用的（例如，，，或）。(key, value)collections.abc.Set==<^

字典视图用法的示例：

```
>>> dishes = {'eggs': 2, 'sausage': 1, 'bacon': 1, 'spam': 500}
>>> keys = dishes.keys()
>>> values = dishes.values()

>>> # iteration
>>> n = 0
>>> for val in values:
...     n += val
>>> print(n)
504

>>> # keys and values are iterated over in the same order
>>> list(keys)
['eggs', 'bacon', 'sausage', 'spam']
>>> list(values)
[2, 1, 1, 500]

>>> # view objects are dynamic and reflect dict changes
>>> del dishes['eggs']
```

```
>>> del dishes['sausage']
>>> list(keys)
['spam', 'bacon']

>>> # set operations
>>> keys & {'eggs', 'bacon', 'salad'}
{'bacon'}
>>> keys ^ {'sausage', 'juice'}
{'juice', 'sausage', 'bacon', 'spam'}
```

4.11。上下文管理器类型

Python的with语句支持由上下文管理器定义的运行时上下文的概念。这是通过一对方法来实现的，这些方法允许用户定义的类定义一个运行时上下文，该上下文在语句正文执行之前输入，并在语句结束时退出：

contextmanager.__enter__()

输入运行时上下文并返回此对象或与运行时上下文相关的另一个对象。此方法返回的值绑定到使用此上下文管理器as的with语句子句中的标识符。

一个返回自身的上下文管理器的例子是一个文件对象。文件对象从__enter__()返回，以允许open()在with语句中用作上下文表达式。

返回相关对象的上下文管理器的示例是返回的对象decimal.localcontext()。这些管理器将激活的小数上下文设置为原始小数上下文的副本，然后返回副本。这允许对with语句正文中的当前小数上下文进行更改，而不影响with语句外的代码 。

contextmanager.__exit__(*exc_type*, *exc_val*, *exc_tb*)

退出运行时环境并返回一个布尔标志，指示是否应该抑制发生的任何异常。如果执行with语句正文时发生异常，则参数将包含异常类型，值和追溯信息。否则，所有三个参数都是None。

从此方法返回一个真值将导致with语句禁止该异常，并继续执行语句之后的with语句。否则在该方法执行完成后异常继续传播。在执行此方法期间发生的异常将替换发生在with语句正文中的任何异常。

传入的异常不应该显式重新显示 - 相反，此方法应该返回一个false值，以指示该方法已成功完成，并且不希望抑制引发的异常。这允许上下文管理代码容易地检测__exit__()方法是否实际失败。

Python定义了几个上下文管理器来支持简单的线程同步，及时关闭文件或其他对象，并简化对活动的十进制算术上下文的操作。特定的类型没有特别对待它们的上下文管理协议的实现。有关contextlib示例，请参阅 模块。

Python的生成器和contextlib.contextmanager装饰器提供了实现这些协议的便捷方式。如果一个生成器函数用contextlib.contextmanager装饰器修饰，它将返回一个实现必要__enter__()和 __exit__()方法的上下文管理器，而不是由未修饰的生成器函数生成的迭代器。

请注意，Python / C API中的Python对象的类型结构中没有任何这些方法的特定插槽。想要定义这些方法的扩展类型必须提供它们作为普通的Python可访问方法。与设置运行时环境的开销相比，单个类字典查找的开销可以忽略不计。

4.12。其他内置类型

解释器支持其他几种对象。其中大多数只支持一两个操作。

4.12.1。模块

模块上唯一的特殊操作是属性访问：`m.name`，其中 *m*是模块，*名称*访问*m*的符号表中定义的名称。模块属性可以分配给。（请注意，该`import` 语句严格来说不是对模块对象的操作; 不需要名为*foo*的模块对象存在，而是需要某个名为*foo*的模块的（外部）*定义*。）`import foo`

每个模块的特殊属性是`__dict__`。这是包含模块符号表的字典。修改这个字典实际上会改变模块的符号表，但直接赋值给 `__dict__`属性是不可能的（你可以写 ，定义为，但不能写 ）。不建议直接修改。`m.__dict__['a'] = 1m.a1m.__dict__ = {}__dict__`

内置在解释器中的模块是这样写的：。如果从一个文件加载，它们被写为。`<module 'sys' (built-in)><module 'os' from '/usr/local/lib/pythonX.Y/os.pyc'>`

4.12.2。类和类实例

请参阅对象，值和类型以及这些类的定义。

4.12.3。函数

函数对象由函数定义创建。函数对象的唯一操作是调用它：`func(argument-list)`。

实际上有两种功能对象：内置函数和用户定义函数。两者都支持相同的操作（来调用函数），但实现是不同的，因此不同的对象类型。

有关更多信息，请参阅函数定义

4.12.4。方法

方法是使用属性表示法调用的函数。有两种风格：内置方法（如`append()`列表）和类实例方法。内置方法用支持它们的类型来描述。

如果通过实例访问方法（在类名称空间中定义的函数），则会得到一个特殊对象：*绑定方法*（也称为*实例方法*）对象。当被调用时，它会将`self`参数添加到参数列表中。绑定方法有两个特殊的只读属性： `m.__self__`方法运行的对象，`m.__func__`是实现该方法的函数。通话 与通话完全等同。`m(arg-1, arg-2, ..., arg-n)m.__func__(m.__self__, arg-1, arg-2, ..., arg-n)`

像函数对象一样，绑定方法对象支持获取任意属性。但是，由于方法属性实际上存储在底层函数对象（`meth.__func__`）中，因此不允许在绑定方法上设置方法属性。尝试在方法上设置属性会导致`AttributeError`引发。为了设置一个方法属性，你需要在底层函数对象上明确地设置它：

```
>>> class C:
...     def method(self):
...         pass
...
>>> c = C()
>>> c.method.whoami = 'my name is method'  # can't set on the method
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'method' object has no attribute 'whoami'
>>> c.method.__func__.whoami = 'my name is method'
>>> c.method.whoami
'my name is method'
```

有关更多信息，请参阅标准类型层次结构

4.12.5。代码对象

代码对象被实现用来表示“伪编译”的可执行Python代码，如函数体。它们不同于函数对象，因为它们不包含对其全局执行环境的引用。代码对象由内置compile()函数返回，并可以通过函数对象的__code__属性提取。另请参阅code模块。

A码对象可以被执行或通过将其（而不是源字符串）所评估的exec()或eval() 内置函数。

有关更多信息，请参阅标准类型层次结构

4.12.6。类型对象

类型对象表示各种对象类型。对象的类型由内置函数访问type()。类型没有特别的操作。标准模块types定义了所有标准内置类型的名称。

类型是这样写的：。<class 'int'>

4.12.7。空对象

该对象由不显式返回值的函数返回。它不支持特殊操作。只有一个空对象，名为 None（内置名称）。type(None)()产生相同的单身人士。

它被写为None。

4.12.8。省略对象

此对象通常由切片使用（请参见切片）。它不支持特殊操作。有一个省略号对象，名为 Ellipsis（内置名称）。 type(Ellipsis)()产生 Ellipsis单身人士。

它被写为Ellipsis或...。

4.12.9。NotImplemented对象

当它们被要求对不支持的类型进行操作时，该对象从比较和二进制操作中返回。有关更多信息，请参阅比较。只有一个NotImplemented对象。 type(NotImplemented)()生成单例实例。

它被写为NotImplemented。

4.12.10。布尔值

布尔值是两个常量对象False和True。它们被用来表示真值（尽管其他值也可以被认为是错误或真实的）。在数值上下文中（例如，当用作算术运算符的参数时），它们的行为分别与整数0和1相同。bool()如果该值可以被解释为真值（参见上面的真值测试部分），那么可以使用内置函数将任何值转换为布尔值 。

它们分别写成False和True。

11年4月12日。内部对象

请参阅此信息的标准类型层次结构。它描述了堆栈框架对象，追溯对象和切片对象。

4.13。特殊属性

该实现为几个相关的对象类型添加了一些特殊的只读属性。其中一些不是由 dir() 内置函数报告的。

object.__dict__
用于存储对象（可写）属性的字典或其他映射对象。

instance.__class__
类实例所属的类。

class.__bases__
类对象的基类的元组。

definition.__name__
类，函数，方法，描述符或生成器实例的名称。

definition.__qualname__
类，函数，方法，描述符或生成器实例的限定名称。

3.3版本的新功能

class.__mro__
此属性是在方法解析期间查找基类时所考虑的类的元组。

class.mro()
这个方法可以被一个元类覆盖，为它的实例定制方法解析顺序。它在类实例化中被调用，并且其结果被存储在__mro__。

class.__subclasses__()
每个类都保留一个对其直接子类的弱引用列表。该方法返回所有这些仍然存在的引用的列表。例：

```
>>> int.__subclasses__()
[<class 'bool'>]
```

脚注

[1] 有关这些特殊方法的更多信息，请参阅Python参考手册（基本定制）。

[2] 结果，这个列表被认为是相等的，对于元组也是如此。[1, 2] [1.0, 2.0]

[3] 自解析器无法分辨操作数的类型以来，它们必须具备。

[4] *(1，2，3，4)* 下套管字符是那些具有一般类别属性是“路”（字母，大写），“LL”（字母，小写），或“LT”（字母，首字母大写）中的一个。

[5] *(1，2)* 要格式化只有一个元组因此，你应该提供一个单元组的唯一元件是要格式化的元组。

5.内置的例外

在Python中，所有异常都必须是派生自类的实例 BaseException。在一个 提到特定类try的except子句的声明中，该子句还处理从该类派生的任何异常类（但不包括派生*它的*异常类）。两个不通过子类关联的异常类永远不会相同，即使它们具有相同的名称。

下面列出的内置例外可以由解释器或内置函数生成。除了提到的地方，它们有一个“关联值”，表示错误的详细原因。这可能是几条信息（例如错误代码和解释代码的字符串）的字符串或元组。关联的值通常作为参数传递给异常类的构造函数。

用户代码可以引发内置的异常。这可用于测试异常处理程序或报告错误情况，“就像”解释程序引发相同异常的情况一样; 但要注意，没有什么可以防止用户代码引发不适当的错误。

内置的异常类可以被分类以定义新的异常; 鼓励程序员从Exception 类或它的一个子类派生新的异常，而不是从中派生BaseException。有关定义异常的更多信息，请参见Python教程中 用户定义的例外。

当在exceptor finally子句中 引发（或重新提升）异常时，__context__会自动设置为最后捕获的异常; 如果未处理新的异常，最终显示的追踪将包括原始异常和最终异常。

当提升一个新的异常（而不是使用裸raise重新提高目前正在处理的异常），隐含的例外情况下可以通过使用补充有明确的病因from有 raise ：

```
raise new_exc from original_exc
```

以下表达式from必须是例外或None。它将被设置为__cause__引发的异常。设置 __cause__也隐式地将__suppress_context__ 属性设置为True，以便用于 有效地替换旧的异常和新的异常以用于显示目的（例如转换为，而在调试时将旧异常保留用于内省。 raise new_exc from NoneKeyErrorAttributeError__context__

除了异常本身的回溯之外，默认回溯显示代码还显示链式异常。__cause__存在时总是显示明确链接的异常。__context__仅在__cause__ is None和__suppress_context__is为false时才会显示隐式链接的异常。

在任何一种情况下，任何链式异常之后总会显示异常，这样追踪的最后一行总是显示最后一次引发的异常。

5.1。基类

以下例外主要用作其他例外的基类。

*异常*BaseException

所有内置异常的基类。它并不意味着被用户定义的类直接继承（为此，使用Exception）。如果 str()在此类的实例上调用，则返回实例的参数表示，或者在没有参数时返回空字符串。

args

赋给异常构造函数的参数元组。一些内置的异常（例如OSError）会期望一定数量的参数并为这个元组赋予特殊的含义，而另一些内置异常通常只用一个字符串来调用，从而给出错误消息。

with_traceback(*tb*)

此方法将*tb*设置为异常的新追溯并返回异常对象。它通常用于这样的异常处理代码中：

```
try:
    ...
except SomeException:
    tb = sys.exc_info()[2]
    raise OtherException(...).with_traceback(tb)
```

*异常*Exception

所有内置的，非系统退出的异常都源自这个类。所有用户定义的异常也应该从这个类派生。

*异常*ArithmeticError

对于那些内置异常的基类时引发的各种算术错误：OverflowError，ZeroDivisionError，FloatingPointError。

*异常*BufferError

当无法执行缓冲区相关操作时引发。

*异常*LookupError

在映射或序列上使用的键或索引无效时引发异常的基类：IndexError，KeyError。这可以直接通过codecs.lookup()。

5.2。具体例外

以下例外是通常引发的例外情况。

*异常*AssertionError

当一个assert陈述失败时引发。

*异常*AttributeError

当属性引用（请参阅属性引用）或分配失败时引发。（当一个对象根本不支持属性引用或属性赋值时，TypeError会引发。）

*异常*EOFError

当input()函数碰到文件结束条件（EOF）而不读取任何数据时引发。（注意：当方法碰到EOF时io.IOBase.read()，io.IOBase.readline()方法会返回一个空字符串。）

*异常*FloatingPointError

当浮点操作失败时引发。这个异常总是被定义的，但是只能在Python配置了--with-fpectl选项或者文件中WANT_SIGFPE_HANDLER定义了符号时才会引发 pyconfig.h。

*异常*GeneratorExit

当发电机或协同程序关闭时引发；见generator.close()和coroutine.close()。它直接继承BaseException而不是Exception因为它在技术上不是一个错误。

*异常*ImportError

当import语句遇到尝试加载模块的麻烦时引发。当“从列表”中 有一个名称不能找到时也引发。 from ... import

该name和path属性可以只使用关键字参数的构造函数中设置。设置时，它们分别表示尝试导入的模块的名称和触发异常的任何文件的路径。

*版本3.3中更改：*添加name和path属性。

*异常*ModuleNotFoundError

其中的一个子类ImportError是import 在找不到模块时引发的。当它被None 发现时它也被提出sys.modules。

3.6版本中的新功能。

*异常*IndexError

当序列下标超出范围时引发。（切片索引被无声地截断以落入允许的范围;如果索引不是整数，TypeError则会被引发。）

*异常*KeyError

在现有密钥集中未找到映射（字典）键时引发。

*异常*KeyboardInterrupt

当用户点击中断键时（通常Control-C或 Delete），引发。在执行期间，会定期检查中断。该异常继承，以防BaseException意外捕获代码，Exception从而阻止解释器退出。

*异常*MemoryError

当一个操作用完内存时引发，但情况仍然可能被救出（通过删除一些对象）。关联的值是一个字符串，指示什么样的（内部）操作耗尽内存。请注意，由于底层内存管理架构（C的malloc()功能），解释器可能无法总是从这种情况中恢复; 它仍然会引发一个异常，以便可以打印堆栈回溯，以防止失控程序的发生。

*异常*NameError

当没有找到本地或全球名称时引发。这仅适用于非限定名称。关联的值是包含无法找到的名称的错误消息。

*异常*NotImplementedError

这个异常来源于RuntimeError。在用户定义的基类中，抽象方法在它们需要派生类重写方法时或在开发类时表示仍需要添加实际实现时才会引发此异常。

> **注意：** 它不应该用来表示一个操作符或者方法根本不是被支持的 - 在这种情况下，要么保持操作符/方法未定义，要么将其设置为子类None。

> **注意：** NotImplementedError和NotImplemented不能互换，即使他们有相似的名称和用途。请参阅 NotImplemented有关何时使用它的详细信息。

*异常*OSError（[*arg*]）
*异常*OSError（ *errno* ，*strerror* [，*filename* [，*winerror* [，*filename2*]]] ）

当系统函数返回与系统相关的错误（包括“文件未找到”或“磁盘已满”等I / O故障（不适用于非法参数类型或其他偶然错误））时会引发此异常。

构造函数的第二种形式设置了相应的属性，如下所述。None如果未指定，属性默认为。为了向后兼容，如果传递了三个参数，则该args属性只包含前两个构造函数参数的二元组。

构造函数通常实际返回一个子类OSError，如下面的OS例外所述。特定的子类取决于最终errno值。此行为仅在OSError直接构建或通过别名构造时发生，并且在子类化时不会继承。

errno

来自C变量的数字错误代码errno。

winerror

在Windows下，这会给你本机的Windows错误代码。errno然后，该属性是POSIX术语中的该本地错误代码的近似翻译。

在Windows下，如果*winerror*构造函数参数是一个整数，则该errno属性由Windows错误代码确定，并且忽略*errno*参数。在其他平台上，*winerror*参数被忽略，并且该winerror属性不存在。

strerror

相应的错误消息，由操作系统提供。它由perror()POSIX FormatMessage()下的C函数和Windows 下的格式化。

filename
filename2

对于涉及文件系统路径（例如open()或os.unlink()）的异常，filename是传递给该函数的文件名。对于涉及两个文件系统路径（例如os.rename()）filename2的函数，对应于传递给该函数的第二个文件名。

*改变在3.3版：*EnvironmentError，IOError，WindowsError，socket.error，select.error和mmap.error已合并到OSError，并构造可能会返回一个子类。

*在版本3.4中更改：*此filename属性现在是传递给函数的原始文件名，而不是从文件系统编码编码或解码的名称。此外，还添加了*filename2*构造函数参数和属性。

*异常*OverflowError

当算术运算的结果太大而无法表示时引发。整数不会发生这种情况（宁可MemoryError放弃也不 放弃）。但是，由于历史原因，有时会针对超出所需范围的整数引发OverflowError。由于C中缺少浮点异常处理的标准化，因此大多数浮点操作都不会被检查。

*异常*RecursionError

这个异常来源于RuntimeError。当解释器检测到最大递归深度（见sys.getrecursionlimit()）被超过时，它会被提高。

*3.5版本中的新功能：*以前，平原RuntimeError被提出。

*异常*ReferenceError

当weakref.proxy()函数创建的弱引用代理被用于访问垃圾收集后指示对象的属性时，会引发此异常 。有关弱引用的更多信息，请参阅weakref模块。

*异常*RuntimeError

当检测到不属于任何其他类别的错误时引发。关联的值是一个字符串，指示出错的地方。

*异常*StopIteration

通过内置函数next()和迭代器的 __next__()方法来提示，迭代器没有产生更多项目。

异常对象具有单个属性value，在构造异常时将其作为参数给出，并且默认为None。

当生成器或协程函数返回时，StopIteration会引发一个新实例，并将函数返回的值用作value异常构造函数的 参数。

如果在存在指令的情况下定义的生成器函数提升，它将被转换为（保留 为新异常的原因）。from __future__ import generator_stopStopIterationRuntimeErrorStopIteration

*在版本3.3中进行了更改：*添加了value属性以及生成器函数使用它返回值的功能。

*在版本3.5中更改：*引入了RuntimeError转换。

*异常*StopAsyncIteration

必须通过异步迭代器对象的__anext__()方法 来提高以停止迭代。

3.5版本中的新功能。

*异常*SyntaxError

解析器遇到语法错误时引发。这可能发生在一个 import声明，调用内置的功能exec() 或eval()阅读初始脚本或标准输入（也交互）时，或。

这个类的实例有属性filename，lineno， offset并且text为的细节更容易获得。 str() 的异常实例只返回消息。

*异常*IndentationError

与不正确缩进相关的语法错误的基类。这是一个子类SyntaxError。

*异常*TabError

当缩进含有不一致的制表符和空格时引发。这是一个子类IndentationError。

*异常*SystemError

当翻译发现内部错误时引发，但情况看起来并不严重，导致它放弃所有希望。关联的值是一个字符串，表示出现了什么问题（以低级方式）。

您应该将此报告给您的Python解释器的作者或维护者。一定要报告Python解释器的版本（sys.version它也会在交互式Python会话开始时打印），确切的错误消息（异常的关联值）以及触发错误的程序源（如果可能）。

*异常*SystemExit

这个例外是由sys.exit()函数引发的。它从继承而来， BaseException而Exception不是偶然被代码捕获Exception。这允许异常正确传播并导致解释器退出。当它不被处理时，Python解释器退出; 没有堆栈回溯被打印。构造函数接受传递给的可选参数sys.exit()。如果该值是一个整数，它指定系统退出状态（传递给C的exit()函数）; 如果是None，退出状态为零; 如果它有其他类型（如字符串），则打印该对象的值并且退出状态为一。

调用将sys.exit()被转换为异常，以便可以执行清理处理程序（ finally语句子句try ），以便调试程序可以执行脚本而不会失去控制权。 os._exit()如果确实需要立即退出（例如，在调用后的子进程中os.fork() ），则可以使用该函数。

code
: 传递给构造函数的退出状态或错误消息。（默认为None。）

*异常*TypeError
: 当操作或功能应用于不适当类型的对象时引发。关联的值是一个字符串，提供有关类型不匹配的详细信息。

用户代码可能会引发此异常，以指示对某个对象的尝试操作不受支持，并且不应该这样做。如果一个对象是为了支持一个给定的操作，但还没有提供实现，那么这NotImplementedError是合适的例外。

错误类型的传递参数（例如传递list一个时 int ，预计）应导致TypeError ，但错误的值传递参数（例如一个号码外预期边界）应导致ValueError。

*异常*UnboundLocalError
: 在函数或方法中引用局部变量时引发，但没有值绑定到该变量。这是一个子类NameError。

*异常*UnicodeError
: 当发生与Unicode相关的编码或解码错误时引发。它是一个子类ValueError。

UnicodeError具有描述编码或解码错误的属性。例如， err.object[err.start:err.end]给出编解码器失败的特定无效输入。

encoding
: 引发错误的编码的名称。

reason
: 描述特定编解码器错误的字符串。

object
: 编解码器试图编码或解码的对象。

start
: 中的无效数据的第一个索引object。

end
: 最后一个无效数据之后的索引object。

*异常*UnicodeEncodeError
在编码期间发生与Unicode相关的错误时引发。它是一个子类 UnicodeError。

*异常*UnicodeDecodeError
在解码期间发生与Unicode相关的错误时引发。它是一个子类 UnicodeError。

*异常*UnicodeTranslateError
在翻译过程中发生与Unicode相关的错误时引发。它是一个子类UnicodeError。

*异常*ValueError
当内建的操作或函数接收到具有正确类型但值不恰当的参数时引发，并且这种情况不会被更精确的异常（例如，）所描述IndexError。

*异常*ZeroDivisionError
当分部或模运算的第二个参数为零时引发。关联的值是一个指示操作数和操作类型的字符串。

为了与以前的版本兼容，保留以下例外; 从Python 3.3开始，它们是别名OSError。

*异常*EnvironmentError

*异常*IOError

*异常*WindowsError
仅在Windows上可用。

5.2.1。操作系统例外

以下例外是它们的子类OSError，它们根据系统错误代码而被引发。

*异常*BlockingIOError
当操作阻塞设置为非阻塞操作的对象（例如套接字）时引发。对应于errno EAGAIN，EALREADY，EWOULDBLOCK和EINPROGRESS。

除了那些之外OSError，BlockingIOError还可以有一个属性：

characters_written
包含在阻塞之前写入流的字符数的整数。此属性在使用io模块中的缓冲I / O类时可用。

*异常*ChildProcessError
在对子进程执行操作失败时引发。对应于errno ECHILD。

*异常*ConnectionError
连接相关问题的基类。

子类是 BrokenPipeError，ConnectionAbortedError，ConnectionRefusedError 和 ConnectionResetError。

*异常*BrokenPipeError

ConnectionError在试图在管道上写入而另一端已关闭的情况下尝试写入已关闭写入的套接字时引发的子类。对应于errno EPIPE和ESHUTDOWN。

*异常*ConnectionAbortedError

ConnectionError连接尝试被对等方中止时引发的子类。对应于errno ECONNABORTED。

*异常*ConnectionRefusedError

ConnectionError连接尝试被对等方拒绝时引发的子类。对应于errno ECONNREFUSED。

*异常*ConnectionResetError

ConnectionError连接被同级重置时引发的子类。对应于errno ECONNRESET。

*异常*FileExistsError

尝试创建已存在的文件或目录时引发。对应于errno EEXIST。

*异常*FileNotFoundError

当请求文件或目录但不存在时引发。对应于errno ENOENT。

*异常*InterruptedError

当系统调用被传入信号中断时引发。对应于errno EINTR。

*在版本3.5中进行了更改：*当系统调用被信号中断时，Python现在会重试系统调用，除非信号处理程序引发异常（请参阅**PEP 475** 为理由），而不是提高InterruptedError。

*异常*IsADirectoryError

os.remove()在目录上请求文件操作（例如）时引发。对应于errno EISDIR。

*异常*NotADirectoryError

当目录操作（例如os.listdir()）被请求的目录操作不是目录时引发。对应于errno ENOTDIR。

*异常*PermissionError

尝试运行没有足够访问权限的操作时引发 - 例如文件系统权限。对应于errno EACCES和EPERM。

*异常*ProcessLookupError

当一个给定的过程不存在时引发。对应于errno ESRCH。

*异常*TimeoutError

当系统功能在系统级别超时时引发。对应于errno ETIMEDOUT。

*3.3版新增功能：*OSError添加了上述所有子类。

也可以看看： **PEP 3151** - 修改OS和IO异常层次结构

5.3。警告

以下例外用作警告类别; 请参阅warnings 模块了解更多信息。

*异常*Warning
: 警告类别的基类。

*异常*UserWarning
: 用户代码生成的警告的基类。

*异常*DeprecationWarning
: 有关不推荐使用的功能的警告的基类。

*异常*PendingDeprecationWarning
: 基类，用于警告将来不推荐使用的功能。

*异常*SyntaxWarning
: 有关可疑语法的警告的基类。

*异常*RuntimeWarning
: 用于警告有关可疑运行时行为的基类。

*异常*FutureWarning
: 基类用于警告将来将在语义上改变的构造。

*异常*ImportWarning
: 基类，用于警告模块导入中可能出现的错误。

*异常*UnicodeWarning
: 与Unicode相关的警告的基类。

*异常*BytesWarning
: 与bytes和的相关警告的基类bytearray。

*异常*ResourceWarning
: 与资源使用有关的警告的基类。

3.2版本中的新功能

5.4。异常层次结构

内置异常的类层次结构是：

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
```

```
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      |    +-- ModuleNotFoundError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
      +-- ReferenceError
      +-- RuntimeError
      |    +-- NotImplementedError
      |    +-- RecursionError
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
      +-- SystemError
      +-- TypeError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
           +-- ResourceWarning
```

6.文字处理服务

本章介绍的模块提供了各种字符串操作操作和其他文本处理服务。

codecs在二进制数据服务下描述的模块也与文本处理高度相关。另外，请参阅文本序列类型 - str中 Python的内置字符串类型的文档。

6.1。string- 通用字符串操作

源代码： Lib / string.py

也可以看看： 文本序列类型 - str

字符串方法

6.1.1。字符串常量

这个模块中定义的常量是：

string.**ascii_letters**
下面描述的ascii_lowercase和ascii_uppercase常量的连接。该值不是区域设置相关的。

string.**ascii_lowercase**
小写字母'abcdefghijklmnopqrstuvwxyz'。此值不是区域设置相关的，不会更改。

string.**ascii_uppercase**
大写字母'ABCDEFGHIJKLMNOPQRSTUVWXYZ'。此值不是区域设置相关的，不会更改。

string.**digits**
字符串'0123456789'。

string.**hexdigits**
字符串'0123456789abcdefABCDEF'。

string.**octdigits**
字符串'01234567'。

string.**punctuation**
在C区域设置中被认为是标点符号的ASCII字符字符串。

string.**printable**
被认为可打印的ASCII字符字符串。这是一个组合digits，ascii_letters，punctuation，和whitespace。

string.**whitespace**
包含所有被认为是空白的ASCII字符的字符串。这包括字符空间，制表符，换行符，返回，换页和垂直制表符。

6.1.2。自定义字符串格式

内置的字符串类提供了通过上述format()方法 进行复杂变量替换和值格式化的功能**PEP 3101**。模块中的Formatter类string允许您使用与内置format()方法相同的实现来创建和自定义您自己的字符串格式化行为。

*类*string.Formatter

该Formatter班有下列公共方法：

format（*format_string*，** args*，*** kwargs*）

主要的API方法。它需要一个格式字符串和一组任意的位置和关键字参数。这只是一个包装，呼吁vformat()。

*自3.5版弃用：*将格式字符串作为关键字参数*format_string*传递已被弃用。

vformat（*format_string*，*args*，*kwargs*）

该功能完成格式化的实际工作。它是作为一个单独的函数公开的，您希望传入参数的预定义字典，而不是使用*args和**kwargs 语法将字典作为单独参数解包并重新打包。vformat()将格式字符串分解为字符数据和替换字段的工作。它调用下面描述的各种方法。

另外，Formatter定义了一些打算由子类替换的方法：

parse（*format_string*）

循环format_string并返回一组元组（*literal_text*，*field_name*，*format_spec*，*转换*）。这用于vformat()将字符串分解为文本文本或替换字段。

元组中的值在概念上表示一个字面文本的跨度，后面跟着一个替换字段。如果没有文本文本（如果两个替换字段连续出现，则可能发生这种情况），那么 *literal_text*将是一个零长度的字符串。如果没有替换字段，那么*field_name*，*format_spec*和*转换*的值 将是None。

get_field（*field_name*，*args*，*kwargs*）

鉴于*FIELD_NAME*如通过返回parse()（见上文），将其转换为一个对象进行格式化。返回一个元组（obj，used_key）。默认版本采用在中定义的格式的字符串**PEP 3101**，如“0 [name]”或“label.title”。 *args*和 *kwargs*被传入 vformat()。返回值*used_key*的含义与 *key*的参数相同 get_value()。

get_value（*key*，*args*，*kwargs*）

检索给定的字段值。的*关键*参数将是一个整数或字符串。如果它是一个整数，它表示*args*中位置参数的索引; 如果它是一个字符串，那么它表示*kwargs*中的命名参数。

该*ARGS*参数设置的位置参数列表 vformat()，以及*kwargs*参数设置为关键字参数的字典。

对于复合字段名称，仅为字段名称的第一个组件调用这些函数; 后续组件通过正常的属性和索引操作进行处理。

因此，例如，字段表达式'0.name'将导致 get_value()用*关键*参数0 调用该name 属性。get_value()通过调用内置getattr()函数在返回后查找该属性。

如果索引或关键字引用不存在的项目，则 应该提出IndexError或KeyError应该提出。

check_unused_args（*used_args*，*args*，*kwargs*）

如果需要，检查未使用的参数。该函数的参数是格式字符串中实际引用的所有参数键集（整数用于位置参数，用于命名参数的字符串）以及对传递给vformat 的*args*和*kwargs*的引用。未使用的参数集可以从这些参数中计算出来。 check_unused_args()如果检查失败，则认为会引发异常。

format_field（*value*，*format_spec*）

format_field()只需调用全局format()内置。提供的方法使得子类可以覆盖它。

convert_field（*价值*，*转换*）

转换get_field()给定转换类型的值（返回的值）（如parse()方法返回的元组中）。默认版本理解''（str），'r'（repr）和'a'（ascii）转换类型。

6.1.3。格式字符串语法

该str.format()方法和Formatter类共享格式字符串的相同语法（尽管在Formatter子类中可以定义它们自己的格式字符串语法）。该语法与格式化的字符串文字相关，但存在差异。

格式字符串包含由花括号包围的“替换字段”{}。任何不包含在大括号中的内容都将被视为文字文本，并将其原样复制到输出中。如果您需要在文字中包含大括号字符，则可以通过加倍：{{和来避开}}。

替换字段的语法如下所示：

```
replacement_field ::= "{" [field_name] ["!" conversion] [":" format
 field_name        ::= arg_name ("." attribute_name | "[" element_index
 arg_name          ::= [identifier | digit+]
 attribute_name    ::=   element_index     ::=   + | index_string      :
 "s" | "a"
 format_spec       ::= <在下一节中描述>
identifier
digit index_string
```

在不太正式的术语中，替换字段可以以*field_name*开头，该字段指定要格式化其值的对象，并将其插入到输出中而不是替换字段中。所述*FIELD_NAME*任选地跟随一个*转换*字段，它是由前面带有感叹号'!'，和一个*format_spec*，这是一个冒号之前':'。这些指定替代值的非默认格式。

另请参阅格式规范迷你语言部分。

所述*FIELD_NAME*本身开始于*arg_name*是一个数字或一个关键字。如果它是一个数字，它指的是一个位置参数，如果它是一个关键字，则它指的是一个已命名的关键字参数。如果格式字符串中的数字arg_names依次为0,1,2，...，则它们可以全部省略（不仅仅是一些），并且数字0,1,2，...将按照该顺序自动插入。由于*arg_name*不是以引号分隔的，因此不可能在格式字符串中指定任意字典键（例如，字符串'10'或':-]'）。的*arg_name*之后可以进行任何数量的索引或属性表达式。表单的表达式'.name'使用选择指定的属性getattr()，而表单的表达式'[index]'使用索引查找__getitem__()。

*版本3.1中更改：*位置参数说明符可以省略，所以相当于。'{} {}' '{0} {1}'

一些简单的格式字符串示例

```
"First, thou shalt count to {0}"  # References first positional argument
"Bring me a {}"                   # Implicitly references the first positional argume
"From {} to {}"                   # Same as "From {0} to {1}"
"My quest is {name}"              # References keyword argument 'name'
"Weight in tons {0.weight}"       # 'weight' attribute of first positional arg
"Units destroyed: {players[0]}"   # First element of keyword argument 'players'.
```

格式化之前，*转换*字段会导致类型强制*转换*。通常，格式化值的工作是通过__format__()值本身的方法完成的。但是，在某些情况下，希望强制将某个类型格式化为字符串，并覆盖自己的格式定义。通过在调用之前将该值转换为字符串__format__()，正常的格式化逻辑被绕过。

目前支持三种转换标志：'!s'它要求str()的价值，'!r'这就要求repr()和'!a'它调用ascii()。

一些例子：

```
"Harold's a clever {0!s}"         # Calls str() on the argument first
"Bring out the holy {name!r}"     # Calls repr() on the argument first
"More {!a}"                       # Calls ascii() on the argument first
```

所述*format_spec*字段包含的值应该如何被呈现，包括这样的细节如场宽度，对齐，填充，小数精度等的规范。每个值类型都可以定义自己的“格式化迷你语言”或*format_spec的*解释。

大多数内置类型支持通用的格式化迷你语言，下一节将对此进行介绍。

甲*format_spec*字段还可以包括在其内嵌套替换字段。这些嵌套替换字段可能包含字段名称，转换标志和格式指定，但不允许更深的嵌套。format_spec中的替换字段在解释*format_spec*字符串之前被替换。这允许动态指定值的格式。

有关示例，请参阅格式示例部分。

6.1.3.1。格式化规范的迷你语言

“格式规范”用于格式字符串中包含的替换字段中，以定义如何呈现单个值（请参阅 格式字符串语法和格式化字符串文字）。它们也可以直接传递给内置 format()函数。每个格式表类型可以定义格式规范如何解释。

大多数内置类型为格式规范实现了以下选项，尽管某些格式化选项仅受数字类型支持。

一般惯例是空字符串（""）产生的结果与您调用str()该值时相同。非空格式字符串通常会修改结果。

*标准格式说明符*的一般形式是：

```
format_spec     ::= [[ fill] align] [ sign] [#] [0] [ width] [ grouping_option] [。
 fill           ::= <任何字符>
```

```
align            ::= "<" | ">" | "=" | "^"
符号             ::= "+" | "-" | " "
width            ::=  digit+
grouping_option  ::= "_" | ","
precision        ::=  digit+
type             ::= "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" | '
```

如果指定了有效的*对齐*值，则可以在前面加上一个*填充*字符，该字符可以是任何字符，如果省略，则默认为空格。在格式化字符串文字中或使用该 方法时，不可能使用文字大括号（“{”或“}”）作为*填充*字符。但是，可以插入带有嵌套替换字段的大括号。该限制不影响该功能。`str.format()` `format()`

各种对齐选项的含义如下：

选项	含义
'<'	强制字段在可用空间内左对齐（这是大多数对象的默认对象）。
'>'	强制字段在可用空间内右对齐（这是数字的默认值）。
'='	强制将填充放置在符号（如果有）之后但位于数字之前。这用于以'+000000120'格式打印字段。此对齐选项仅适用于数字类型。当'0'紧接在字段宽度之前时，它成为默认值。
'^'	强制该字段在可用空间内居中。

请注意，除非定义了最小字段宽度，否则字段宽度将始终与要填充它的数据的大小相同，以便在此情况下对齐选项无效。

该*标志*的选择是仅适用于数字类型，并且可以是下列之一：

选项	含义
'+'	表示标志应该用于正数和负数。
'-'	表示一个标志只能用于负数（这是默认行为）。
空间	表示应在正数上使用前导空格，在负数上使用负号。

该'#'选项会导致“替代表单”用于转换。替代形式对于不同类型的定义是不同的。该选项仅适用于整数，浮点型，复数型和小数类型。为整数，当二进制，八进制或十六进制输出时，此选项添加前缀相应'0b'，'0o'或'0x'该输出值。对于浮点数，复数和十进制的替代形式会导致转换结果始终包含小数点字符，即使没有数字跟着它。通常，只有在数字跟随它时，这些转换的结果中才会出现一个小数点字符。另外，对于'g'和'G'转换，结果中不会删除结尾的零。

该','选项表示使用千位分隔符的逗号。对于感知区域的分隔符，请'n'改为使用整数表示类型。

*在版本3.1中更改：*添加了该','选项（另请参阅**PEP 378**）。

该'_'选项表示为浮点表示类型和整数表示类型使用千位分隔符的下划线'd'。对于整数呈现类型'b'，'o'，'x'，和'X'，下划线将每4位插入。对于其他表示类型，指定此选项是一个错误。

*在版本3.6中更改：*添加了'_'选项（另请参阅**PEP 515**）。

*宽度*是定义最小字段宽度的十进制整数。如果未指定，则字段宽度将由内容决定。

如果未指定明确的对齐方式，则在*宽度*字段前加上一个零（'0'）字符将为数字类型启用符号感知零填充。这相当于一个*对齐*类型为的*填充*字符。'0''='

的*精度*是指示多少位数应的小数点格式化与浮点值之后显示的十进制数'f'和'F'，或之前和小数点用于与格式化的浮点值之后'g'或'G'。对于非数字类型，该字段指示最大字段大小 - 换言之，字段内容将使用多少个字符。整数值不允许使用*精度*。

最后，*类型*决定了数据应该如何呈现。

可用的字符串演示文稿类型是：

类型	含义
's'	字符串格式。这是字符串的默认类型，可以省略。
没有	和's'。一样。

可用的整数表示类型是：

类型	含义
'b'	二进制格式。输出基数为2的数字。
'c'	字符。打印前将整数转换为相应的unicode字符。
'd'	十进制整数。以10为基数输出数字。
'o'	八进制格式。输出基数为8的数字。
'x'	十六进制格式 输出基数为16的数字，对于9以上的数字使用小写字母。
'X'	十六进制格式 以16为基数输出数字，使用9以上数字的大写字母。
'n'	数。这与'd'使用当前语言环境设置插入适当的数字分隔符相同。
没有	和'd'。一样。

除了上述表示类型之外，整数还可以使用下面列出的浮点表示类型进行格式化（除了'n'和None）。这样做时，float()用于在格式化之前将整数转换为浮点数。

浮点和小数值的可用表示类型是：

类型	含义
'e'	指数表示法。使用字母'e'以科学记数法打印数字以指示指数。默认的精度是6。
'E'	指数表示法。相同'e'，除了它使用一个大写"E"作为分隔符。
'f'	固定点。将该号码显示为一个定点号码。默认的精度是6。
'F'	固定点。同'f'，但转换nan到NAN和inf到INF。

类型	含义
'g'	一般格式。对于一个给定的精度，这个数字会将数字四舍五入为有效数字，然后根据其大小将结果格式化为定点格式或科学记数法。p >= 1p 精确的规则如下：假设格式化为呈现类型'e'和精度的结果p-1将具有指数exp。然后，如果数字格式与演示文稿类型和精度 。否则，该数字将使用表示类型和精度进行格式化。在这两种情况下，无效尾随零将从有效位数中移除，如果其后没有剩余数字，则小数点也会被移除。-4 <= exp < p'f'p-1-exp'e'p-1 正和负无穷大，正的和负的零，和NaN，被格式化为inf，-inf，0，-0和nan的分别，而不管精度。 精确度0被视为等同于精确度1。默认的精度是6。
'G'	一般格式。与'g'除了切换到'E'数字变得太大相同。无限大和NaN的表示也是大写的。
'n'	数。这与'g'使用当前语言环境设置插入适当的数字分隔符相同。
'%'	百分比。将数字乘以100'f'，然后以固定（）格式显示，然后显示百分号。
没有	类似的'g'，除了定点符号在使用时，至少有一位数字超过小数点。默认精度与表示特定值所需的一样高。整体效果是匹配str()由其他格式修改器改变的输出。

6.1.3.2。格式示例

本节包含str.format()语法和与旧%格式比较的示例。

在大多数情况下，语法与旧%格式相似，但使用{}和的替换:使用而不是%。例如，'%03.2f'可以翻译成'{:03.2f}'。

新的格式语法也支持新的和不同的选项，如下面的例子所示。

按位置访问参数：

```
>>> '{0}, {1}, {2}'.format('a', 'b', 'c')
'a, b, c'
>>> '{}, {}, {}'.format('a', 'b', 'c')  # 3.1+ only
'a, b, c'
>>> '{2}, {1}, {0}'.format('a', 'b', 'c')
'c, b, a'
>>> '{2}, {1}, {0}'.format(*'abc')      # unpacking argument sequence
'c, b, a'
>>> '{0}{1}{0}'.format('abra', 'cad')   # arguments' indices can be repeated
'abracadabra'
```

按名称访问参数：

```
>>> 'Coordinates: {latitude}, {longitude}'.format(latitude='37.24N', longitude='-115.
'Coordinates: 37.24N, -115.81W'
>>> coord = {'latitude': '37.24N', 'longitude': '-115.81W'}
>>> 'Coordinates: {latitude}, {longitude}'.format(**coord)
'Coordinates: 37.24N, -115.81W'
```

访问参数的属性：

```
>>> c = 3-5j
>>> ('The complex number {0} is formed from the real part {0.real} '
...  'and the imaginary part {0.imag}.').format(c)
'The complex number (3-5j) is formed from the real part 3.0 and the imaginary part -5.
>>> class Point:
...     def __init__(self, x, y):
...         self.x, self.y = x, y
...     def __str__(self):
...         return 'Point({self.x}, {self.y})'.format(self=self)
...
>>> str(Point(4, 2))
'Point(4, 2)'
```

访问参数的项目：

```
>>> coord = (3, 5)
>>> 'X: {0[0]};  Y: {0[1]}'.format(coord)
'X: 3;  Y: 5'
```

替换%s和%r：

```
>>> "repr() shows quotes: {!r}; str() doesn't: {!s}".format('test1', 'test2')
"repr() shows quotes: 'test1'; str() doesn't: test2"
```

对齐文本并指定宽度：

```
>>> '{:<30}'.format('left aligned')
'left aligned                  '
>>> '{:>30}'.format('right aligned')
'                 right aligned'
>>> '{:^30}'.format('centered')
'           centered           '
>>> '{:*^30}'.format('centered')  # use '*' as a fill char
'***********centered***********'
```

更换%+f，%-f以及与指定的标志：% f

```
>>> '{:+f}; {:+f}'.format(3.14, -3.14)  # show it always
'+3.140000; -3.140000'
>>> '{: f}; {: f}'.format(3.14, -3.14)  # show a space for positive numbers
' 3.140000; -3.140000'
>>> '{:-f}; {:-f}'.format(3.14, -3.14)  # show only the minus -- same as '{:f}; {:f}'
'3.140000; -3.140000'
```

替换%x并将%o值转换为不同的基础：

```
>>> # format also supports binary numbers
>>> "int: {0:d};  hex: {0:x};  oct: {0:o};  bin: {0:b}".format(42)
'int: 42;  hex: 2a;  oct: 52;  bin: 101010'
>>> # with 0x, 0o, or 0b as prefix:
>>> "int: {0:d};  hex: {0:#x};  oct: {0:#o};  bin: {0:#b}".format(42)
'int: 42;  hex: 0x2a;  oct: 0o52;  bin: 0b101010'
```

使用逗号作为千位分隔符：

```
>>> '{:,}'.format(1234567890)
'1,234,567,890'
```

表示一个百分比：

```
>>> points = 19
>>> total = 22
>>> 'Correct answers: {:.2%}'.format(points/total)
'Correct answers: 86.36%'
```

使用特定于类型的格式：

```
>>> import datetime
>>> d = datetime.datetime(2010, 7, 4, 12, 15, 58)
>>> '{:%Y-%m-%d %H:%M:%S}'.format(d)
'2010-07-04 12:15:58'
```

嵌套论据和更复杂的例子：

```
>>> for align, text in zip('<^>', ['left', 'center', 'right']):
...     '{0:{fill}{align}16}'.format(text, fill=align, align=align)
...
'left<<<<<<<<<<<<'
'^^^^^center^^^^^'
'>>>>>>>>>>>right'
>>>
>>> octets = [192, 168, 0, 1]
>>> '{:02X}{:02X}{:02X}{:02X}'.format(*octets)
'C0A80001'
>>> int(_, 16)
3232235521
>>>
>>> width = 5
>>> for num in range(5,12):
...     for base in 'dXob':
...         print('{0:{width}{base}}'.format(num, base=base, width=width), end=' ')
...     print()
...
    5     5     5   101
    6     6     6   110
    7     7     7   111
    8     8    10  1000
    9     9    11  1001
```

```
  10      A     12   1010
  11      B     13   1011
```

6.1.4。模板字符串

模板提供了更简单的字符串替换，如下所述 **PEP 292**。而不是%基于正常的替换，模板支持$替换，使用以下规则：

- $$是一种逃避; 它被替换为一个单一的$。
- $identifier 命名一个匹配映射关键字的替换占位符 "identifier"。默认情况下，"identifier"限制为以下划线或ASCII字母开头的任何不区分大小写的ASCII字母数字字符串（包括下划线）。$字符终止此占位符规范后的第一个非标识符字符。
- ${identifier}相当于$identifier。当有效的标识符字符跟随占位符时，它是必需的，但不是占位符的一部分，例如"${noun}ification"。

$字符串中的任何其他外观都会导致ValueError 引发。

该string模块提供了一个Template实现这些规则的类。方法Template是：

*类*string.Template（*模板*）

构造函数接受一个参数，它是模板字符串。

substitute（*映射*，** *kwds* ）

执行模板替换，返回一个新的字符串。 *映射*是任何类似字典的对象，其键与模板中的占位符相匹配。或者，您可以提供关键字参数，其中关键字是占位符。当*映射*和*kwds*都给出并且有重复时，来自*kwds*的占位符优先。

safe_substitute（*映射*，** *kwds* ）

就像substitute()，除了如果占位符从*映射*和*kwds*丢失，而不是引发KeyError异常，原始占位符将完整地出现在结果字符串中。此外，与此不同的是substitute()，$意志的任何其他外观只会返回$而不是升起ValueError。

虽然其他异常仍可能发生，但此方法称为“安全”，因为替换总是尝试返回可用的字符串，而不是引发异常。在另一种意义上，safe_substitute()可能是安全以外的任何东西，因为它会默默地忽略包含悬挂分隔符，不匹配大括号或无效Python标识符占位符的格式错误的模板。

Template 实例还提供一个公共数据属性：

template

这是传递给构造函数的*模板*参数的对象。一般来说，你不应该改变它，但是只读访问不被强制执行。

以下是如何使用模板的示例：

```
>>> from string import Template
>>> s = Template('$who likes $what')
>>> s.substitute(who='tim', what='kung pao')
'tim likes kung pao'
>>> d = dict(who='tim')
```

```
>>> Template('Give $who $100').substitute(d)
Traceback (most recent call last):
...
ValueError: Invalid placeholder in string: line 1, col 11
>>> Template('$who likes $what').substitute(d)
Traceback (most recent call last):
...
KeyError: 'what'
>>> Template('$who likes $what').safe_substitute(d)
'tim likes $what'
```

高级用法：可以派生子类来自`Template`定义占位符语法，分隔符或用于分析模板字符串的整个正则表达式。为此，您可以覆盖这些类属性：

- *分隔符* - 这是描述引入分隔符的占位符的文字字符串。默认值是`$`。请注意，这应该*不会*是一个正则表达式，为实现将调用`re.escape()`根据需要在此字符串。
- *idpattern* - 这是描述非支撑占位符模式的正则表达式（大括号将根据需要自动添加）。默认值是正则表达式`(?-i:[_a-zA-Z][_a-zA-Z0-9]*)`。

  > **注意：** 由于默认*标志*是`re.IGNORECASE`，模式`[a-z]`可以匹配一些非ASCII字符。这就是我们`-i`在这里使用当地国旗的原因。
  >
  > 当为了向后兼容而保留*标志的*`re.IGNORECASE`，可以将其覆盖到子类`0`或当子类化时。`re.IGNORECASE | re.ASCII`

- *flags* - 编译用于识别替换的正则表达式时将应用的正则表达式标志。默认值是`re.IGNORECASE`。请注意，`re.VERBOSE`它将始终添加到标志中，因此自定义的*idpattern*必须遵循冗长正则表达式的约定。

  3.2版本中的新功能

或者，您可以通过覆盖类属性*模式*来提供整个正则表达式*模式*。如果您这样做，则该值必须是具有四个命名捕获组的正则表达式对象。捕获组对应于上面给出的规则以及无效的占位符规则：

- *转义* - 该组匹配转义序列，例如`$$`，以默认模式。
- *named* - 此组匹配未占用的占位符名称; 它不应该在捕获组中包含分隔符。
- *braced* - 此组匹配大括号括起来的占位符名称; 它不应该在捕获组中包含分隔符或大括号。
- *无效* - 该组匹配任何其他分隔符模式（通常是单个分隔符），它应该出现在正则表达式的最后。

6.1.5。帮助函数

`string.`**capwords**(*s*, *sep = None*)

将参数拆分为单词，使用`str.split()`每个单词大写`str.capitalize()`，并使用大写字母加入`str.join()`。如果可选的第二个参数*sep*不存在，或者`None`空白字符的运行被替换为单个空格，并且前导空格和尾部空白被删除，否则使用*sep*分割和连接单词。

6.2。 re- 正则表达式操作

源代码： Lib / re.py

该模块提供了与Perl中的类似的正则表达式匹配操作。

要搜索的模式和字符串都可以是Unicode字符串（ str ）以及8位字符串（ bytes ）。但是，Unicode字符串和8位字符串不能混合使用：也就是说，无法将Unicode字符串与字节模式匹配，反之亦然; 同样，当要求替换时，替换字符串必须与模式和搜索字符串的类型相同。

正则表达式使用反斜杠字符（'\'）来表示特殊形式或允许使用特殊字符而不用调用其特殊含义。这与Python在字符串文字中用于相同目的的相同字符的使用相冲突; 例如，要匹配文字反斜杠，可能必须将其写'\\\\'为模式字符串，因为正则表达式必须是\\，并且每个反斜杠必须表示为\\在常规Python字符串文字中。

解决方案是将Python的原始字符串表示法用于正则表达式模式; 反斜杠不以任何特殊方式在字符串文字前面加上'r'。所以，r"\n"是包含两个字符的字符串 '\' 和'n'，虽然"\n"是包含一个换行符一个一个字符的字符串。通常，模式将使用此原始字符串表示法以Python代码表示。

需要注意的是，大多数正则表达式操作在已编译的正则表达式中都可用作模块级函数和方法 。这些函数是快捷方式，不需要先编译正则表达式对象，但会错过一些微调参数。

也可以看看： 第三方正则表达式模块具有与标准库re模块兼容的API ，但提供了额外的功能和更全面的Unicode支持。

6.2.1。正则表达式语法

正则表达式（或RE）指定一组与之匹配的字符串; 这个模块中的函数可以让你检查一个特定的字符串是否与给定的正则表达式匹配（或者一个给定的正则表达式是否匹配一个特定的字符串，这个字符串可以归结为同一个东西）

正则表达式可以连接起来形成新的正则表达式; 如果*A* 和*B*都是正则表达式，那么*AB*也是一个正则表达式。通常，如果一个字符串*p*匹配*A*而另一个字符串*q*匹配*B*，则字符串*pq*将匹配AB。除非*A*或*B*包含低优先级操作; *A*和*B*之间*的*边界条件; 或者有编号的组参考。因此，复杂的表达式可以很容易地从简单的基本表达式构建，就像这里描述的那样。有关正则表达式的理论和实现的详细信息，请参阅Friedl的书[Frie09]或几乎所有关于编译器构建的教科书。

正则表达式格式的简要说明如下。有关更多信息和更温和的演示，请参阅正则表达式HOWTO。

正则表达式可以包含特殊字符和普通字符。大多数普通字符（如'A'，，'a'或'0'）是最简单的正则表达式; 他们只是匹配自己。您可以连接普通字符，以便last匹配字符串'last'。（在本节的其余部分中，我们将写入RE ，通常不带引号，并且要匹配字符串。）this special style'in single quotes'

一些字符，如'|'或者'('，是特殊的。特殊字符代表普通字符的类，或者影响它们周围正则表达式的解释方式。

重复限定符（*，+，?，{m,n}，等等）不能直接嵌套。这避免了非贪心修饰符后缀?和其他修饰符在其他实现中的歧义。要将第二次重复应用于内部重复，可以使用括号。例如，该表达式(?:a{6})*匹配六个'a'字符的任意倍数。

特殊字符是：

.
（点）。在默认模式下，它匹配除换行符以外的任何字符。如果该DOTALL标志已被指定，则它匹配包括换行符的任何字符。

^
（插入符）匹配字符串的开头，并且在MULTILINE每个换行符之后，模式也立即匹配。

$
匹配字符串的结尾或紧跟在字符串末尾的换行符之前，并且在MULTILINE模式中也匹配换行符之前。 foo 匹配'foo'和'foobar'，而正则表达式foo$只匹配'foo'。更有趣的是， 通常foo.$在'foo1\nfoo2\n'匹配'foo2'中搜索，但在MULTILINE模式中搜索'foo1' 搜索单个$in 'foo\n'将会找到两个（空的）匹配：一个在换行符之前，另一个在字符串末尾。

*
使得到的RE匹配前面的RE的0个或更多个重复，尽可能多的重复。 ab*将匹配'a'，'ab'或'a'，后跟任意数量的'b'。

+
使得到的RE匹配前面的RE的一个或多个重复。 ab+将匹配'a'，然后匹配'b'的非零数; 它不会仅仅匹配'a'。

?
使得到的RE匹配前面的RE的0或1个重复。 ab?将匹配'a'或'ab'。

*?，+?，??
的'*'，'+'和'?'预选赛都是*贪婪的*; 它们匹配尽可能多的文本。有时候这种行为是不希望的; 如果RE <.*>匹配，它将匹配整个字符串，而不仅仅是。在限定符之后添加使得它以*非贪婪*或*最小的*方式执行匹配; 尽可能*少的* 字符将被匹配。使用RE 只会匹配。'<a> b <c>' '<a>' ?<.*?>'<a>'

{m}
指定应该匹配前一个RE的正好*m个*副本; 更少的匹配导致整个RE不匹配。例如，a{6}将恰好匹配六个'a'字符，但不是五个。

{m,n}
使得到的RE匹配前面RE的*m*到*n*次重复，试图尽可能多地匹配重复。例如， a{3,5}将匹配3到5个'a'字符。省略*m*指定零的下限，省略*n*指定无限上限。作为一个例子，a{4,}b将匹配'aaaab'或一千个'a'字符后跟一个'b'，但不是'aab'。逗号不能省略，否则修饰符会与前面描述的形式混淆。

{m,n}?
使得到的RE从*m*到*n*重复前面的RE，尝试匹配尽可能*少的*重复。这是以前限定符的非贪婪版本。例如，在6个字符的字符串中'aaaaaa'，a{3,5}将匹配5个'a'字符，而a{3,5}?只匹配3个字符。

\

要么逃避特殊字符（允许您匹配像'*'，'?'等等字符 ），要么发送特殊序列; 下面讨论特殊的序列。

如果您没有使用原始字符串来表示模式，请记住Python也使用反斜杠作为字符串文本中的转义序列; 如果转义序列不被Python的解析器识别，则反斜杠和后续字符将包含在结果字符串中。但是，如果Python能够识别结果序列，则反斜杠应重复两次。这很复杂，也很难理解，所以强烈建议您使用原始字符串，除了最简单的表达式。

`[]`

用于指示一组字符。在一组中：

- 人物可以单独上市，如`[amk]`将匹配'a'，'m'或'k'。
- 字符的范围可以通过给两个字符，并通过把它们分开来表示'-'，例如`[a-z]`将匹配任何小写ASCII字母，`[0-5][0-9]`将所有的后两位数字从匹配00到59，并`[0-9A-Fa-f]`会匹配任何十六进制数字。如果-被转义（例如`[a\-z]`）或者被放置为第一个或最后一个字符（例如`[-a]`或`[a-]`），则它将与文字匹配'-'。
- 特殊字符在集合内部失去其特殊含义。例如，`[(+*)]`将匹配任何文字字符的'('，'+'，'*'，或')'。
- 诸如`\w`或`\S`（定义如下）的字符类也被接受在一个集合内，尽管它们匹配的字符取决于是否`ASCII`或`LOCALE`模式有效。
- 不在一个范围内的字符可以通过 对该集合进行*补充*来匹配。如果该集合的第一个字符是'^'，所有*不在*集合中的字符将被匹配。例如，`[^5]`将匹配除任何字符'5'，`[^^]`并将匹配除任何字符'^'。 ^如果它不是集合中的第一个字符，那么它没有特别的意义。
- 要匹配']'集合内的文字，在其前面加上反斜线，或将其放在集合的开头。例如，无论是`[()[\]{}]`和`[]()[{}]`都将匹配一个括号。

`|`

`A|B`，其中*A*和*B*可以是任意RE，创建一个匹配*A*或*B*的正则表达式。'|'以这种方式可以分离任意数量的RE 。这可以在组内使用（见下文）。当目标字符串被扫描时，'|'由左向右尝试分隔的RE 。当一个模式完全匹配时，该分支被接受。这意味着一旦*A*匹配，*B*将不会被进一步测试，即使它会产生更长的整体匹配。换句话说，'|'操作员从不贪婪。为了匹配文字'|'，使用`\|`或将其包含在字符类中，如in `[|]`。

`(...)`

匹配括号内的任何正则表达式，并指示组的开始和结束; 一个组的内容可以在匹配完成后被检索到，并且可以在后面的字符串中与`\number` 特殊序列匹配，如下所述。要匹配的文字'('或')'使用`\(`或`\)`，或将它们括字符类中：`[(]`，`[)]`。

`(?...)`

这是一个扩展符号（'?'后面的a '('不是有意义的）。'?'确定构造的含义和进一步语法之后的第一个字符是。扩展通常不会创建新的组; `(?P<name>...)`是这条规则的唯一例外。以下是当前支持的扩展。

`(?aiLmsux)`

（从所述一组一个或多个字母'a'，'i'，'L'，'m'，'s'，'u'，'x'。 ）该组匹配空字符串; 这些字母设置了相应的标志：（`re.A`仅匹配ASCII码），`re.I`（忽略大小写），`re.L`（区域依赖），`re.M`（多行），`re.S`（点匹配全部），`re.U`（Unicode匹配）和`re.X`（详细）表达。（这些标志在模块内容中有描述。）如果您希望将标志作为正则表达式的一部分包含在内，而不是将*标志*参数传递给该`re.compile()`函数，这非常有用。在表达式字符串中应该首先使用标志。

`(?:...)`

常规圆括号的非捕获版本。匹配括号内的任何正则表达式，但匹配的子字符串 在执行匹配或稍后引用模式后*无法*检索。

(?imsx-imsx:...)

（零或从所述一组多个字母'i'，'m'，'s'，'x'，任选接着进行'-'随后的一个或多个字母，从同一组。）中的字母设置或取消相应的标志： re.I（忽略大小写）， re.M（多线）， re.S （点匹配全部）和re.X（详细），用于表达的部分。（这些标志在模块内容中有描述。）

3.6版本中的新功能。

(?P<name>...)

类似于普通括号，但该组匹配的子字符串是通过符号组名称访问的*名称*。组名称必须是有效的Python标识符，并且每个组名称只能在正则表达式中定义一次。一个符号组也是一个编号组，就好像该组没有被命名一样。

命名组可以在三种情况下被引用。如果模式是 (?P<quote>['"]).*?(?P=quote)（即匹配用单引号或双引号引用的字符串）：

参考组"引用"的上下文	方法来引用它
在相同的模式本身	• (?P=quote)（如图所示） • \1
处理匹配对象*m时*	• m.group('quote') • m.end('quote')（等等。）
在传递给*repl* 参数的字符串中re.sub()	• \g<quote> • \g<1> • \1

(?P=name)

对指定组的反向引用; 它匹配任何文本是由早些时候被任命为组匹配的*名称*。

(?#...)

一条评论; 圆括号的内容被简单地忽略。

(?=...)

如果...匹配，则匹配下一个，但不会消耗任何字符串。这被称为*前瞻断言*。例如，只有跟在后面才会匹配。 Isaac (?=Asimov)'Isaac ''Asimov'

(?!...)

匹配如果...不匹配。这是一个*负面的前瞻断言*。例如，只有在*没有* 跟随时才会匹配。 Isaac (?!Asimov)'Isaac ''Asimov'

(?<=...)

如果字符串中的当前位置在当前位置... 处结束匹配，则匹配。这被称为*积极向后看断言*。 (?<=abc)def会找到一个匹配'abcdef'，因为lookbehind会备份3个字符并检查包含的模式是否匹配。所包含的模式必须只匹配一些固定长度的串，这意味着 abc或者a|b是允许的，但a*并a{3,4}不是。请注意，以正向lookbehind断言开始的模式在搜索字符串的开头不匹配; 你很可能会想使用这个 search()函数而不是match()函数：

```
>>> import re
>>> m = re.search('(?<=abc)def', 'abcdef')
```

```
>>> m.group(0)
'def'
```

本示例在连字符后面查找单词：

```
>>> m = re.search(r'(?<=-)\w+', 'spam-egg')
>>> m.group(0)
'egg'
```

*在版本3.5中更改：*增加了对固定长度的组引用的支持。

(?<!...)
如果字符串中的当前位置没有匹配，则匹配 ...。这被称为*否定向后断言*。与正向lookbehind断言类似，所包含的模式只能匹配某些固定长度的字符串。以反向lookbehind断言开头的模式可能会匹配搜索字符串的开头。

(?(id/name)yes-pattern|no-pattern)
yes-pattern如果存在具有给定*ID*或*名称*的组，则尝试匹配; 如果no-pattern不存在，则尝试匹配。no-pattern是可选的，可以省略。例如，(<)?(\w+@\w+(?:\.\w+)+)(?(1)>|$)是一个贫穷的电子邮件匹配模式，这将匹配'<user@host.com>'以及'user@host.com'，但不与'<user@host.com'也'user@host.com>'。

特殊序列由'\'以下列表中的一个字符组成。如果普通字符不是ASCII数字或ASCII字母，则生成的RE将匹配第二个字符。例如，\$匹配字符'$'。

\number
匹配相同编号的组的内容。组从1开始编号。例如，匹配或，但不是（注意组之后的空格）。该特殊序列只能用于匹配前99个组中的一个。如果第一个数字*数*是0，或*数*为3个八进制数字长，也不会被解释为一组匹配，但与八进制值的字符*数*。在 字符类和字符类中，所有数字转义都被视为字符。(.+) \1'the the''55 55''thethe''['']'

\A
只匹配字符串的开头。

\b
匹配空字符串，但仅限于单词的开头或结尾。一个词被定义为一个单词字符序列。请注意，形式上，\b被定义为a \w和\W字符之间的界限（反之亦然），或者在\w字符串的开始/结束之间。这意味着，r'\bfoo\b'比赛'foo'，'foo.'，'(foo)'，但不还是。'bar foo baz''foobar''foo3'

默认情况下，Unicode字母数字是Unicode模式中使用的字母数字，但可以通过使用该ASCII标志来更改。如果使用该LOCALE标志，字边界由当前的区域设置决定。在字符范围内，\b表示退格字符，以便与Python的字符串文字兼容。

\B
匹配空字符串，但仅限于它*不在*单词的开头或结尾。这意味着，r'py\B'比赛'python'，'py3'，'py2'，而不是'py'，'py.'或'py!'。\B正好相反\b，所以Unicode模式中的单词字符是Unicode字母数字或下划线，尽管这可以通过使用ASCII标志来更改。如果使用该LOCALE标志，字边界由当前的区域设置决定。

\d
对于Unicode（str）模式：
匹配任何Unicode十进制数字（即，Unicode字符类别[Nd]中的任何字符）。这包括[0-9]，还有许多其他数字字符。如果ASCII标志只用于[0-9]匹配（但标志会影响整个正

则表达式，所以在这种情况下使用明确[0-9]可能是更好的选择）。

对于8位（字节）模式：

匹配任何十进制数字; 这相当于[0-9]。

\D

匹配任何不是十进制数字的字符。这是相反的\d。如果这个ASCII标志被使用，这变成了等价物[^0-9]（但是这个标志影响了整个正则表达式，所以在这种情况下使用显式[^0-9]可能是更好的选择）。

\s

对于Unicode（str）模式：

匹配Unicode空白字符（其中包括 许多其他字符，例如许多语言中印刷术规则规定的非空白空格）。如果使用标志，只 匹配（但标志影响整个正则表达式，所以在这种情况下使用明确 可能是一个更好的选择）。[\t\n\r\f\v]ASCII[\t\n\r\f\v][\t\n\r\f\v]

对于8位（字节）模式：

在ASCII字符集中匹配被认为是空白的字符; 这相当于。[\t\n\r\f\v]

\S

匹配任何不是空白字符的字符。这是相反的\s。如果这个ASCII标志被使用，这变成了等价物（但是这个标志影响了整个正则表达式，所以在这种情况下使用显式可能是更好的选择）。[^ \t\n\r\f\v][^ \t\n\r\f\v]

\w

对于Unicode（str）模式：

匹配Unicode字符; 这包括大多数可以是任何语言的单词的一部分的字符，以及数字和下划线。如果使用ASCII标志，只[a-zA-Z0-9_]匹配（但标志影响整个正则表达式，所以在这种情况下使用明确[a-zA-Z0-9_]可能是一个更好的选择）。

对于8位（字节）模式：

匹配ASCII字符集中被认为是字母数字的字符; 这相当于[a-zA-Z0-9_]。如果使用该LOCALE标志，则匹配在当前语言环境和下划线中被认为是字母数字的字符。

\W

匹配任何不是单词字符的字符。这是相反的\w。如果这个ASCII标志被使用，这变成了等价物[^a-zA-Z0-9_]（但是这个标志影响了整个正则表达式，所以在这种情况下使用显式[^a-zA-Z0-9_]可能是更好的选择）。如果使用该LOCALE标志，则匹配在当前语言环境和下划线中被认为是字母数字的字符。

\Z

只匹配字符串的末尾。

大多数由Python字符串文字支持的标准转义符也被正则表达式解析器接受：

```
\a      \b      \f      \n
\r      \t      \u      \U
\v      \x      \\
```

（注意\b用于表示单词边界，仅在字符类内部表示“退格”。）

’\u’ 并且’\U’ 转义序列只能在Unicode模式中识别。以字节模式它们是错误的。

八进制逃生包括在一个有限的形式。如果第一个数字是0，或者如果有三个八进制数字，则它被认为是八进制转义。否则，它是一个组参考。至于字符串文字，八进制转义字符的长度总是最多三位数字。

*改变在3.3版本：*在'\u'和'\U'转义序列已被添加。

在版本3.6中更改：'\'现在包含和一个ASCII字母的未知转义是错误。

6.2.2。模块内容

该模块定义了几个函数，常量和一个异常。一些函数是编译正则表达式的全功能方法的简化版本。大多数不平凡的应用程序总是使用编译后的表格。

*在版本3.6中更改：*标志常量现在是实例RegexFlag，它是的子类 enum.IntFlag。

re.compile(*pattern* , *flags = 0*)

编译一个正则表达式模式为正则表达式对象，其可用于使用它的匹配 match() ， search() 以及其他方法，说明如下。

表达式的行为可以通过指定一个*标志*值来修改。值可以是以下任何变量，使用按位或（ | 操作符）组合。

序列

```
prog = re.compile(pattern)
result = prog.match(string)
```

相当于

```
result = re.match(pattern, string)
```

但是re.compile()如果在单个程序中多次使用该表达式，则使用并保存生成的正则表达式对象以便重用将会更有效。

> **注意：** 传递给最新模式的编译版本 re.compile()和模块级匹配函数都被缓存，因此一次只使用少量正则表达式的程序不需要担心编译正则表达式。

re.A

re.ASCII

让\w，\W，\b，\B，\d，\D，\s和\S 执行ASCII-只匹配完整的Unicode匹配代替。这只对Unicode模式有意义，并且在字节模式中被忽略。对应于内联标志(?a)。

请注意，为了向后兼容，该re.U标志仍然存在（以及它的同义词re.UNICODE及其嵌入的对应物(?u)），但这些在Python 3中是多余的，因为匹配对于字符串是默认的Unicode（并且字节不允许Unicode匹配）。

re.DEBUG

显示关于编译表达式的调试信息 没有相应的内联标志。

re.I
re.IGNORECASE

执行不区分大小写的匹配; 表达式也[A-Z]将匹配小写字母。除非该标志用于禁用非ASCII匹配，否则全面的Unicode匹配（例如Ü匹配 ü ）也可以工作re.ASCII。除非re.LOCALE使用了该标志，否则当前语言环境不会更改此标志的效果。对应于内联标志(?i)。

请注意，当Unicode模式[a-z]或[A-Z]与IGNORECASE标志结合使用时，它们将匹配52个ASCII字母和4个额外的非ASCII字母：'İ'（U + 0130，拉丁大写字母I，带上面的点），'ı'（U + 0131，拉丁小写字母无点i），'ſ'（U + 017F，拉丁小写字母长）和'K'（U + 212A，开尔文符号）。如果使用该ASCII标志，只有字母'a'到'z'和'A'到'Z'匹配（但标志会影响整个正则表达式，所以在这种情况下使用明确(?-i:[a-zA-Z])可能是更好的选择）。

re.L
re.LOCALE

让\w，\W，\b，\B和区分大小写的匹配取决于当前的语言环境。该标志只能用于字节模式。由于区域设置机制非常不可靠，因此不鼓励使用此标志，它一次只处理一种“文化”，并且仅适用于8位语言环境。对于Unicode（str）模式，默认情况下，Unicode匹配已在Python 3中启用，并且它能够处理不同的语言环境/语言。对应于内联标志(?L)。

*在版本3.6中更改：*re.LOCALE只能与字节模式一起使用，并且不兼容re.ASCII。

re.M
re.MULTILINE

指定时，模式字符'^'匹配字符串的开头和每行的开头（紧跟在每个换行符后面）; 并且模式字符'$'匹配字符串的末尾和每行末尾（紧接在每个换行符之前）。默认情况下，'^'只匹配字符串的开始位置，并且'$'只匹配字符串的末尾和匹配字符串末尾的换行符（如果有）。对应于内联标志(?m)。

re.S
re.DOTALL

使'.'特殊字符匹配任何字符，包括换行符; 没有这个标志，'.'将会匹配*除*换行符*之外*的任何内容。对应于内联标志(?s)。

re.X
re.VERBOSE

该标志允许您编写正则表达式，通过允许您在视觉上分离模式的逻辑部分并添加注释，该正则表达式看起来更好，并且更易读。在该模式内的空白被忽略，当在字符类，或当由反斜杠之后除，或令牌内的类似*?，(?:或(?P<...>。当一行包含一个#不在字符类中并且没有非转义反斜杠#的行时，最左边的所有字符 都将被忽略。

这意味着匹配一个十进制数的下面两个正则表达式对象在功能上是相等的：

```
a = re.compile(r"""\d +  # the integral part
                   \.    # the decimal point
                   \d *  # some fractional digits""", re.X)
b = re.compile(r"\d+\.\d*")
```

对应于内联标志(?x)。

re. search (*pattern* , *string* , *flags = 0*)

扫描*字符串*查找正则表达式*模式*产生匹配的第一个位置 ，并返回相应的匹配对象。None如果字符串中没有位置与模式匹配，则返回; 请注意，这与在字符串中的某处找到零长度匹配不同。

re. match (*pattern* , *string* , *flags = 0*)

如果*字符串*开头的零个或多个字符与正则表达式*模式*匹配，则返回相应的匹配对象。None如果字符串与模式不匹配，则返回; 请注意，这与零长度匹配不同。

请注意，即使在MULTILINE模式下，re.match()只会匹配字符串的开头，而不是每行的开头。

如果您想在*字符串*中的任何位置找到匹配项，search()请改用（另请参阅search（ ）与match（ ））。

re. fullmatch (*pattern* , *string* , *flags = 0*)

如果整个*字符串*匹配正则表达式*模式*，则返回相应的匹配对象。None如果字符串与模式不匹配，则返回; 请注意，这与零长度匹配不同。

3.4版新增功能

re. split (*pattern* , *string* , *maxsplit = 0* , *flags = 0*)

根据*模式*的出现拆分*字符串*。如果在*模式*中使用捕获括号，则*模式*中所有组的文本也会作为结果列表的一部分返回。如果*maxsplit*不为零，则最多 发生*maxsplit*分割，并且字符串的其余部分作为列表的最后一个元素返回。

```
>>> re.split(r'\W+', 'Words, words, words.')
['Words', 'words', 'words', '']
>>> re.split(r'(\W+)', 'Words, words, words.')
['Words', ', ', 'words', ', ', 'words', '.', '']
>>> re.split(r'\W+', 'Words, words, words.', 1)
['Words', 'words, words.']
>>> re.split('[a-f]+', '0a3B9', flags=re.IGNORECASE)
['0', '3', '9']
```

如果分隔符中有捕获组，并且它在字符串的起始处匹配，则结果将以空字符串开头。字符串的结尾也是一样：

```
>>> re.split(r'(\W+)', '...words, words...')
['', '...', 'words', ', ', 'words', '...', '']
```

这样，分隔符组件总是在结果列表中的相同索引处找到。

注意: split()当前没有在空模式匹配上分割字符串。例如：

```
>>> re.split('x*', 'axbc')
['a', 'bc']
```

尽管'x*'在'a'之前，'b'和'c'之间以及'c'之后也匹配0'x'，但是当前这些匹配被忽略。在未来的Python版本中，将会执行正确的行为（即分裂空匹配并返回），但由于这是一种向

后不兼容的变化，因此会在此期间引发一次。['', 'a', 'b', 'c', '']FutureWarning
目前只能匹配空字符串的模式永远不会拆分字符串。由于这与预期行为不符，ValueError因此将从Python 3.5开始提出：

```
>>> re.split("^$", "foo\n\nbar\n", flags=re.M)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  ...
ValueError: split() requires a non-empty pattern match.
```

*在版本3.1中更改：*添加了可选的标志参数。

*在版本3.5中更改：*分割可能匹配空字符串的模式现在引发警告。现在只能匹配空字符串的模式被拒绝。

re.**findall**(*pattern*, *string*, *flags = 0*)

返回的所有非重叠的匹配*模式*的*字符串*，如字符串列表。该*字符串*进行扫描左到右，并匹配以发现的顺序返回。如果模式中存在一个或多个组，请返回组列表; 如果模式有多个组，这将是一个元组列表。结果中包含空匹配项。

注意： 由于当前实施的限制，在下一场比赛中不包括空比赛之后的人物，因此 返回（注意错过“t”）。这在Python 3.7中有所改变。 findall(r'^|\w+', 'two words')['', 'wo', 'words']

re.**finditer**(*pattern*, *string*, *flags = 0*)

返回一个迭代得到匹配的对象上的所有RE非重叠的匹配*图案*中*的字符串*。该*字符串* 进行扫描左到右，并匹配以发现的顺序返回。结果中包含空匹配项。另见关于的说明findall()。

re.**sub**(*pattern*, *repl*, *string*, *count = 0*, *flags = 0*)

通过用替换*repl*替换*字符串中*最左边不重叠出现的*模式而*获得的*字符串*。如果未找到该模式，则 *字符串*将保持不变。 *repl*可以是一个字符串或一个函数; 如果它是一个字符串，则处理其中的任何反斜杠转义。也就是说，转换为单个换行符，转换为回车符，等等。未知的转义如独立。反向引用，例如，被模式中的组6匹配的子串替换。例如：\n\r\&\6

```
>>> re.sub(r'def\s+([a-zA-Z_][a-zA-Z_0-9]*)\s*\(\s*\):',
...         r'static PyObject*\npy_\1(void)\n{',
...         'def myfunc():')
'static PyObject*\npy_myfunc(void)\n{'
```

如果*repl*是一个函数，那么它被称为每个非重叠*模式的*发生 。该函数采用单个匹配对象 参数，并返回替换字符串。例如：

```
>>> def dashrepl(matchobj):
...     if matchobj.group(0) == '-': return ' '
...     else: return '-'
>>> re.sub('-{1,2}', dashrepl, 'pro----gram-files')
'pro--gram files'
>>> re.sub(r'\sAND\s', ' & ', 'Baked Beans And Spam', flags=re.IGNORECASE)
'Baked Beans & Spam'
```

该模式可能是一个字符串或模式对象。

可选参数*count*是要替换的模式发生的最大数量; *count*必须是一个非负整数。如果省略或为零，则所有事件将被替换。只有在与先前的匹配不相邻时才替换该模式的空匹配，因此返回 。 sub('x*', '-', 'abc')'-a-b-c-'

在string-type *repl*参数中，除了上述的字符转义和反向引用外， \g<name>还将使用name由(?P<name>...)语法定义的名为group的匹配子字符串。 \g<number>使用相应的组号码; \g<2>因此相当于\2 ，但在替代方法中不会含糊不清\g<2>0。 \20将被解释为对组20的引用，而不是对组2的引用，后面是字面字符'0'。反向引用\g<0>替换RE中匹配的整个子字符串。

*在版本3.1中更改：*添加了可选的标志参数。

*在版本3.5中更改：*不匹配的组被替换为空字符串。

*在版本3.6中更改：*现在包含和ASCII字母的*模式中的*未知转义'\'错误。

*从版本3.5开始弃用，将在版本3.7中删除：*由*repl*组成的'\'和ASCII字母中的未知转义现在引发弃用警告，并且在Python 3.7中将被禁止。

re.subn(*pattern*, *repl*, *string*, *count = 0*, *flags = 0*)

执行与之相同的操作sub()，但返回一个元组。(new_string, number_of_subs_made)

*在版本3.1中更改：*添加了可选的标志参数。

*在版本3.5中更改：*不匹配的组被替换为空字符串。

re.escape(*模式*)

逃生中的所有字符*图案*，除了ASCII字母，数字和'_'。如果您想匹配任何可能具有正则表达式元字符的文字字符串，这非常有用。例如：

```
>>> print(re.escape('python.exe'))
python\.exe

>>> legal_chars = string.ascii_lowercase + string.digits + "!#$%&'*+-.^_`|~:"
>>> print('[%s]+' % re.escape(legal_chars))
[abcdefghijklmnopqrstuvwxyz0123456789\!\#\$\%\&\'\*\+\-\.\^_\`\|\~\:]+

>>> operators = ['+', '-', '*', '/', '**']
>>> print('|'.join(map(re.escape, sorted(operators, reverse=True))))
\/|\-|\+|\*\*|\*
```

这个函数不能用于替换字符串， sub() 并且subn()只有反斜杠才能被转义。例如：

```
>>> digits_re = r'\d+'
>>> sample = '/usr/sbin/sendmail - 0 errors, 12 warnings'
>>> print(re.sub(digits_re, digits_re.replace('\\', r'\\'), sample))
/usr/sbin/sendmail - \d+ errors, \d+ warnings
```

*版本3.3中更改：*该'_'字符不再逃脱。

re.purge( )

清除正则表达式缓存。

*异常*re.error(*msg*, *pattern = None*, *pos = None*)

当传递给其中一个函数的字符串不是有效的正则表达式（例如，它可能包含不匹配的圆括号）或编译或匹配过程中发生其他错误时引发的异常。如果一个字符串不包含匹配的模式，那永远不会出错。错误实例具有以下附加属性：

msg

未格式化的错误消息。

pattern

正则表达式模式。

pos

编译失败（可能）的*模式*中的索引None。

lineno

与*pos*对应的行（可能None）。

colno

与*pos*对应的列（可能None）。

*版本3.5中已更改：*添加了其他属性。

6.2.3。正则表达式对象

编译的正则表达式对象支持以下方法和属性：

regex.search(*string* [, *pos* [, *endpos*]])

扫描*字符串*查找此正则表达式生成匹配的第一个位置，并返回相应的匹配对象。None如果字符串中没有位置与模式匹配，则返回; 请注意，这与在字符串中的某处找到零长度匹配不同。

可选的第二个参数*pos*在搜索要开始的字符串中给出一个索引; 它默认为0。这不完全等同于切分字符串; 该'^'模式字符在字符串的真正开始，并在仅仅一个换行符后的位置相匹配，但不一定，其中搜索是启动索引。

可选参数*endpos*限制字符串搜索的距离; 它就好像字符串是*endpos*字符一样长，所以只有从*pos*到的字符才会被搜索到匹配。如果*endpos*小于*pos*，则不会找到匹配; 否则，如果*rx*是编译的正则表达式对象，则等价于。endpos - 1rx.search(string, 0, 50)rx.search(string[:50], 0)

```
>>> pattern = re.compile("d")
>>> pattern.search("dog")     # Match at index 0
<_sre.SRE_Match object; span=(0, 1), match='d'>
>>> pattern.search("dog", 1)  # No match; search doesn't include the "d"
```

regex.match(*string* [, *pos* [, *endpos*]])

如果*字符串开头*的零个或多个字符与此正则表达式匹配，则返回相应的匹配对象。如果字符串与模式不匹配，则返回; 请注意，这与零长度匹配不同。None

可选的*pos*和*endpos*参数具有与该search()方法相同的含义 。

```
>>> pattern = re.compile("o")
>>> pattern.match("dog")      # No match as "o" is not at the start of "dog".
>>> pattern.match("dog", 1)   # Match as "o" is the 2nd character of "dog".
<_sre.SRE_Match object; span=(1, 2), match='o'>
```

如果您想在*字符串*中的任何位置找到匹配项， search()请改用（另请参阅search（ ）与match（ ） ）。

regex.fullmatch(*string* [, *pos* [, *endpos*]])

如果整个*字符串*匹配此正则表达式，则返回相应的 匹配对象。None如果字符串与模式不匹配，则返回; 请注意，这与零长度匹配不同。

可选的*pos*和*endpos*参数具有与该search()方法相同的含义 。

```
>>> pattern = re.compile("o[gh]")
>>> pattern.fullmatch("dog")      # No match as "o" is not at the start of "dog".
>>> pattern.fullmatch("ogre")     # No match as not the full string matches.
>>> pattern.fullmatch("doggie", 1, 3)   # Matches within given limits.
<_sre.SRE_Match object; span=(1, 3), match='og'>
```

3.4版新增功能

regex.split(*string* , *maxsplit = 0*)

与split()使用编译模式的函数相同。

regex.findall(*string* [, *pos* [, *endpos*]])

与findall()函数类似，使用编译模式，但也接受限制搜索区域的可选*pos*和*endpos*参数search()。

regex.finditer(*string* [, *pos* [, *endpos*]])

与finditer()函数类似，使用编译模式，但也接受限制搜索区域的可选*pos*和*endpos*参数search()。

regex.sub(*repl* , *string* , *count = 0*)

与sub()使用编译模式的函数相同。

regex.subn(*repl* , *string* , *count = 0*)

与subn()使用编译模式的函数相同。

regex.flags

正则表达式匹配标志。这是给出的 标志，模式中的compile()任何(?...)内联标志和隐式标志（如UNICODE模式是Unicode字符串）的组合。

regex.groups
模式中的捕获组数量。

regex.groupindex
一个字典，用于映射由(?P<id>)组号定义的任何符号组名。如果模式中没有使用符号组，则字典为空。

regex.pattern
编译RE对象的模式字符串。

6.2.4。匹配对象

匹配对象始终有一个布尔值True。由于match()并且 在没有匹配时search()返回None ，您可以测试是否与简单if语句匹配 ：

```
match = re.search(pattern, string)
if match:
    process(match)
```

匹配对象支持以下方法和属性：

match.expand（*模板*）
如方法所示，通过在模板字符串*模板*上执行反斜杠替换来返回字符串sub()。诸如\n转换为适当字符的转义，以及相应组的内容替换数字反向引用（ \1 ，\2 ）和命名反向引用（ \g<1> ，\g<name> ）。

*在版本3.5中更改：*不匹配的组被替换为空字符串。

match.group（[*group1*，...]）
返回匹配的一个或多个子组。如果有一个参数，结果是一个单一的字符串; 如果有多个参数，则结果是每个参数有一个项目的元组。没有参数，*group1*默认为零（整个匹配被返回）。如果*groupN*参数为零，则相应的返回值是整个匹配的字符串; 如果它在包含范围[1..99]中，则它是匹配相应括号组的字符串。如果组编号为负数或大于模式中定义的组数，IndexError则会引发异常。如果一个组包含在不匹配的模式的一部分中，则相应的结果是None。如果一个组包含在多次匹配的模式的一部分中，则返回最后的匹配。

```
>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
>>> m.group(0)       # The entire match
'Isaac Newton'
>>> m.group(1)       # The first parenthesized subgroup.
'Isaac'
>>> m.group(2)       # The second parenthesized subgroup.
'Newton'
>>> m.group(1, 2)    # Multiple arguments give us a tuple.
('Isaac', 'Newton')
```

如果正则表达式使用(?P<name>...)语法，则*groupN* 参数也可以是通过其组名称标识组的字符串。如果字符串参数未在模式中用作组名称，IndexError 则会引发异常。

一个中等复杂的例子：

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
>>> m.group('first_name')
'Malcolm'
>>> m.group('last_name')
'Reynolds'
```

命名组也可以通过它们的索引来引用：

```
>>> m.group(1)
'Malcolm'
>>> m.group(2)
'Reynolds'
```

如果一个组匹配多次，只能访问最后一场比赛：

```
>>> m = re.match(r"(..)+", "a1b2c3")  # Matches 3 times.
>>> m.group(1)                        # Returns only the last match.
'c3'
```

match.__getitem__(*g*)

这与之相同m.group(g)。这允许从比赛中更容易地访问个人组：

```
>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
>>> m[0]       # The entire match
'Isaac Newton'
>>> m[1]       # The first parenthesized subgroup.
'Isaac'
>>> m[2]       # The second parenthesized subgroup.
'Newton'
```

3.6版本中的新功能。

match.groups(*默认=无*)

返回一个包含匹配所有子组的元组，从1开始，直到模式中有多个组。该*默认*参数用于那些没有参加比赛组; 它默认为None。

例如：

```
>>> m = re.match(r"(\d+)\.(\d+)", "24.1632")
>>> m.groups()
('24', '1632')
```

如果我们将小数点后的所有数字都设为可选，则并非所有组都可以参与比赛。None除非给出*默认*参数，否则这些组将默认为：

```
>>> m = re.match(r"(\d+)\.?(\d+)?", "24")
>>> m.groups()      # Second group defaults to None.
('24', None)
>>> m.groups('0')   # Now, the second group defaults to '0'.
('24', '0')
```

match.groupdict(*默认=无*)

返回包含匹配的所有*命名*子组的字典的子集名称。该*默认*参数用于那些没有参加比赛组; 它默认为None。例如：

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
>>> m.groupdict()
{'first_name': 'Malcolm', 'last_name': 'Reynolds'}
```

match.start([*组*])

match.end([*组*])

返回按*组*匹配的子串的开始和结束索引; *组*默认为零（意味着整个匹配的子字符串）。-1如果 *组*存在但返回不匹配。对于一个匹配对象*m*和一个对匹配有贡献的组*g*，由组*g*匹配的子串（相当于m.group(g)）是

```
m.string[m.start(g):m.end(g)]
```

请注意，如果*组*匹配一个空字符串，m.start(group)它将相等。例如，之后，是1，是2，并且都是2，并引发异常。m.end(group)m = re.search('b(c?)', 'cba')m.start(0)m.end(0)m.start(1)m.end(1)m.start(2)IndexError

一个将从电子邮件地址中删除*remove_this*的示例：

```
>>> email = "tony@tiremove_thisger.net"
>>> m = re.search("remove_this", email)
>>> email[:m.start()] + email[m.end():]
'tony@tiger.net'
```

match.span([*组*])

对于匹配*m*，返回2元组。请注意，如果*小组*没有对比赛做出贡献，则是。*组*默认为零，整个比赛。(m.start(group), m.end(group))(-1, -1)

match.pos

所述的值*POS*将其传递给search()或 match()一个的方法regex对象。这是RE引擎开始寻找匹配的字符串的索引。

match.endpos

传递给正则表达式对象或 方法的*endpos*的值。这是RE引擎不会去的字符串的索引。search()match()

match.lastindex

最后匹配的捕获组的整数索引，或者None根本没有匹配的组。例如，表述(a)b，((a)(b))以及((ab))将具有如果施加到串，而表达将有，如果施加到相同的字符串。lastindex == 1'ab'(a)(b)lastindex == 2

match.lastgroup

最后匹配的捕获组None的名称，或者该组没有名称，或者根本没有匹配组。

match.re

该正则表达式对象，其match()或 search()方法生产的这个匹配实例。

match.string

传递给match()或的字符串search()。

6.2.5。正则表达式示例

6.2.5.1。检查一对

在这个例子中，我们将使用以下辅助函数来更加优雅地显示匹配对象：

```
def displaymatch(match):
    if match is None:
        return None
    return '<Match: %r, groups=%r>' % (match.group(), match.groups())
```

假设您正在编写一个扑克程序，其中玩家的手表示为5个字符的字符串，每个字符代表一张牌，“a”代表王牌，“k”代表国王，“q”代表女王，“j”代表插孔， “t”为10，“2”至“9”代表具有该值的卡。

要查看给定的字符串是否是有效的手，可以执行以下操作：

```
>>> valid = re.compile(r"^[a2-9tjqk]{5}$")
>>> displaymatch(valid.match("akt5q"))  # Valid.
"<Match: 'akt5q', groups=()>"
>>> displaymatch(valid.match("akt5e"))  # Invalid.
>>> displaymatch(valid.match("akt"))    # Invalid.
>>> displaymatch(valid.match("727ak"))  # Valid.
"<Match: '727ak', groups=()>"
```

最后一只手，"727ak"包含一对，或两个相同的价值卡。为了与正则表达式匹配，可以使用反向引用：

```
>>> pair = re.compile(r".*(.).*\1")
>>> displaymatch(pair.match("717ak"))     # Pair of 7s.
"<Match: '717', groups=('7',)>"
>>> displaymatch(pair.match("718ak"))     # No pairs.
>>> displaymatch(pair.match("354aa"))     # Pair of aces.
"<Match: '354aa', groups=('a',)>"
```

为了找出这对卡片组成的卡片，可以group()按照以下方式使用 匹配对象的方法：

```
>>> pair.match("717ak").group(1)
'7'

# Error because re.match() returns None, which doesn't have a group() method:
>>> pair.match("718ak").group(1)
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    re.match(r".*(.).*\1", "718ak").group(1)
AttributeError: 'NoneType' object has no attribute 'group'
```

```
>>> pair.match("354aa").group(1)
'a'
```

6.2.5.2。模拟scanf（ ）

Python目前没有与之相当的功能scanf()。正则表达式通常比scanf()格式化字符串更强大，但也更加冗长 。下面的表格提供了scanf()格式标记和正则表达式之间的一些或多或少的等价映射。

scanf() 代币	正则表达式
%c	.
%5c	.{5}
%d	[-+]?\d+
%e , %E , %f , %g	[-+]?(\d+(\.\d*)?|\.\d+)([eE][-+]?\d+)?
%i	[-+]?(0[xX][\dA-Fa-f]+|0[0-7]*|\d+)
%o	[-+]?[0-7]+
%s	\S+
%u	\d+
%x , %X	[-+]?(0[xX])?[\dA-Fa-f]+

从字符串中提取文件名和数字

```
/usr/sbin/sendmail - 0 errors, 4 warnings
```

你会使用scanf()类似的格式

```
%s - %d errors, %d warnings
```

等效的正则表达式是

```
(\S+) - (\d+) errors, (\d+) warnings
```

6.2.5.3。search（ ）与match（ ）

Python提供了基于正则表达式的两种不同的基本操作： re.match()只在字符串的开始处re.search()检查匹配，并检查字符串 中任何位置的匹配（这是Perl默认执行的操作）。

例如：

```
>>> re.match("c", "abcdef")    # No match
>>> re.search("c", "abcdef")   # Match
<_sre.SRE_Match object; span=(2, 3), match='c'>
```

以正值开始的正则表达式'^'可用于search()限制字符串开始处的匹配：

```
>>> re.match("c", "abcdef")    # No match
>>> re.search("^c", "abcdef")  # No match
>>> re.search("^a", "abcdef")  # Match
<_sre.SRE_Match object; span=(0, 1), match='a'>
```

但是请注意，在MULTILINE模式中，match()只匹配字符串的开始处，而使用search()以开头的正则表达式'^'匹配每行的开头。

```
>>> re.match('X', 'A\nB\nX', re.MULTILINE)  # No match
>>> re.search('^X', 'A\nB\nX', re.MULTILINE)  # Match
<_sre.SRE_Match object; span=(4, 5), match='X'>
```

6.2.5.4。制作电话簿

split()将字符串分割成由传递模式分隔的列表。该方法对于将文本数据转换为可由Python轻松读取和修改的数据结构非常有用，如以下创建电话簿的示例所示。

首先，这是输入。通常它可能来自一个文件，这里我们使用三引号字符串语法：

```
>>> text = """Ross McFluff: 834.345.1254 155 Elm Street
...
... Ronald Heathmore: 892.345.3428 436 Finley Avenue
... Frank Burger: 925.541.7625 662 South Dogwood Way
...
...
... Heather Albrecht: 548.326.4584 919 Park Place"""
```

条目由一个或多个换行符分隔。现在我们将字符串转换为一个列表，每个非空行都有自己的条目：

```
>>> entries = re.split("\n+", text)
>>> entries
['Ross McFluff: 834.345.1254 155 Elm Street',
'Ronald Heathmore: 892.345.3428 436 Finley Avenue',
'Frank Burger: 925.541.7625 662 South Dogwood Way',
'Heather Albrecht: 548.326.4584 919 Park Place']
```

最后，将每个条目分成一个名字，姓氏，电话号码和地址。我们使用的maxsplit参数是split()因为地址有空格，我们的分裂模式，在它里面：

```
>>> [re.split(":? ", entry, 3) for entry in entries]
[['Ross', 'McFluff', '834.345.1254', '155 Elm Street'],
['Ronald', 'Heathmore', '892.345.3428', '436 Finley Avenue'],
['Frank', 'Burger', '925.541.7625', '662 South Dogwood Way'],
['Heather', 'Albrecht', '548.326.4584', '919 Park Place']]
```

该:?模式与姓氏后的冒号匹配，以便它不会出现在结果列表中。有了maxsplit之后4，我们可以将房门号码与街道名称分开：

```
>>> [re.split(":? ", entry, 4) for entry in entries]
[['Ross', 'McFluff', '834.345.1254', '155', 'Elm Street'],
```

```
['Ronald', 'Heathmore', '892.345.3428', '436', 'Finley Avenue'],
['Frank', 'Burger', '925.541.7625', '662', 'South Dogwood Way'],
['Heather', 'Albrecht', '548.326.4584', '919', 'Park Place']]
```

6.2.5.5。文本打印

sub()用字符串或函数的结果替换每个模式的出现。这个例子演示了如何使用sub()一个函数来“敲击”文本，或者随机化除第一个和最后一个字符之外的每个单词中所有字符的顺序：

```
>>> def repl(m):
...     inner_word = list(m.group(2))
...     random.shuffle(inner_word)
...     return m.group(1) + "".join(inner_word) + m.group(3)
>>> text = "Professor Abdolmalek, please report your absences promptly."
>>> re.sub(r"(\w)(\w+)(\w)", repl, text)
'Poefsrosr Aealmlobdk, pslaee reorpt your abnseces plmrptoy.'
>>> re.sub(r"(\w)(\w+)(\w)", repl, text)
'Pofsroser Aodlambelk, plasee reoprt yuor asnebces potlmrpy.'
```

6.2.5.6。寻找所有副词

findall()匹配*所有*出现的模式，而不仅仅是第一个模式search()。例如，如果一个人是作家，想要在某些文本中找到所有的副词，他或她可以findall()按照以下方式使用：

```
>>> text = "He was carefully disguised but captured quickly by police."
>>> re.findall(r"\w+ly", text)
['carefully', 'quickly']
```

6.2.5.7。寻找所有副词及其位置

如果想要了解与匹配文本相比所有匹配模式匹配的更多信息，finditer()这很有用，因为它提供了匹配对象而不是字符串。继续前面的例子，如果一个作家想要在一些文本中找到所有的副词*和他们的位置*，他或她会finditer()按照以下方式使用：

```
>>> text = "He was carefully disguised but captured quickly by police."
>>> for m in re.finditer(r"\w+ly", text):
...     print('%02d-%02d: %s' % (m.start(), m.end(), m.group(0)))
07-16: carefully
40-47: quickly
```

6.2.5.8。原始字符串符号

原始字符串符号（r"text"）使正则表达式保持正常。没有它，'\'正则表达式中的每个反斜杠（）都必须以另一个反斜杠作为前缀来转义它。例如，以下两行代码在功能上是相同的：

```
>>> re.match(r"\W(.)\1\W", " ff ")
<_sre.SRE_Match object; span=(0, 4), match=' ff '>
>>> re.match("\\W(.)\\1\\W", " ff ")
<_sre.SRE_Match object; span=(0, 4), match=' ff '>
```

当想要匹配文字反斜杠时，它必须在正则表达式中转义。用原始字符串表示法，这意味着r"\\"。没有原始字符串表示法，必须使用"\\\\"，使以下代码行功能相同：

```
>>> re.match(r"\\", r"\\")
<_sre.SRE_Match object; span=(0, 1), match='\\'>
>>> re.match("\\\\", r"\\")
<_sre.SRE_Match object; span=(0, 1), match='\\'>
```

6.2.5.9。编写一个标记

分词器或扫描器 分析字符串以对字符组进行分类。这是编写编译器或解释器的第一步。

文本类别使用正则表达式指定。该技术将这些组合成一个主正则表达式，并循环连续匹配：

```
import collections
import re

Token = collections.namedtuple('Token', ['typ', 'value', 'line', 'column'])

def tokenize(code):
    keywords = {'IF', 'THEN', 'ENDIF', 'FOR', 'NEXT', 'GOSUB', 'RETURN'}
    token_specification = [
        ('NUMBER',  r'\d+(\.\d*)?'),  # Integer or decimal number
        ('ASSIGN',  r':='),           # Assignment operator
        ('END',     r';'),            # Statement terminator
        ('ID',      r'[A-Za-z]+'),    # Identifiers
        ('OP',      r'[+\-*/]'),      # Arithmetic operators
        ('NEWLINE', r'\n'),           # Line endings
        ('SKIP',    r'[ \t]+'),       # Skip over spaces and tabs
        ('MISMATCH',r'.'),            # Any other character
    ]
    tok_regex = '|'.join('(?P<%s>%s)' % pair for pair in token_specification)
    line_num = 1
    line_start = 0
    for mo in re.finditer(tok_regex, code):
        kind = mo.lastgroup
        value = mo.group(kind)
        if kind == 'NEWLINE':
            line_start = mo.end()
            line_num += 1
        elif kind == 'SKIP':
            pass
        elif kind == 'MISMATCH':
            raise RuntimeError(f'{value!r} unexpected on line {line_num}')
        else:
            if kind == 'ID' and value in keywords:
                kind = value
            column = mo.start() - line_start
            yield Token(kind, value, line_num, column)

statements = '''
    IF quantity THEN
        total := total + price * quantity;
        tax := price * 0.05;
```

```
    ENDIF;
'''

for token in tokenize(statements):
    print(token)
```

令牌生成器产生以下输出：

```
Token(typ='IF', value='IF', line=2, column=4)
Token(typ='ID', value='quantity', line=2, column=7)
Token(typ='THEN', value='THEN', line=2, column=16)
Token(typ='ID', value='total', line=3, column=8)
Token(typ='ASSIGN', value=':=', line=3, column=14)
Token(typ='ID', value='total', line=3, column=17)
Token(typ='OP', value='+', line=3, column=23)
Token(typ='ID', value='price', line=3, column=25)
Token(typ='OP', value='*', line=3, column=31)
Token(typ='ID', value='quantity', line=3, column=33)
Token(typ='END', value=';', line=3, column=41)
Token(typ='ID', value='tax', line=4, column=8)
Token(typ='ASSIGN', value=':=', line=4, column=12)
Token(typ='ID', value='price', line=4, column=15)
Token(typ='OP', value='*', line=4, column=21)
Token(typ='NUMBER', value='0.05', line=4, column=23)
Token(typ='END', value=';', line=4, column=27)
Token(typ='ENDIF', value='ENDIF', line=5, column=4)
Token(typ='END', value=';', line=5, column=9)
```


[Frie09]	[illegible]

6.3。difflib- 助手计算

源代码： Lib / difflib.py

该模块提供了用于比较序列的类和函数。它可以用于比较文件，并可以产生各种格式的差异信息，包括HTML和上下文以及统一差异。为了比较目录和文件，请参阅filecmp模块。

*类*difflib.SequenceMatcher

这是一个灵活的类，用于比较任何类型的序列对，只要序列元素是可哈希的。基本算法早于20世纪80年代后期由Ratcliff和Obershelp在双曲线名称“形态模式匹配”下发布的算法，并且比它更有趣。该想法是找到不包含“垃圾”元素的最长连续匹配子序列; 这些“垃圾”元素是某些意义上不感兴趣的元素，例如空白行或空白。（处理垃圾是对Ratcliff和Obershelp算法的扩展。）然后，将相同的想法递归应用于匹配子序列左侧和右侧的序列片段。这不会产生最小的编辑序列，但往往会产生对人们“看起来正确”的匹配。

时机：基本Ratcliff-Obershelp算法在最坏情况下为立方时间，在预期情况下为二次时间。SequenceMatcher是最坏情况下的平方时间，并且预期情况行为以复杂的方式依赖于序列有多少共同元素; 最佳案例时间是线性的。

自动垃圾启发式： SequenceMatcher支持将某些序列项自动视为**垃圾的启发式**方法。启发式计算每个单独项目出现在序列中的次数。如果一个项目的重复（在第一个之后）占序列的1%以上并且序列长度至少为200个项目，则该项目被标记为“流行”并且为了序列匹配而被视为垃圾。通过将autojunk参数设置为False创建时，可以关闭此启发式SequenceMatcher。

新版本3.2：在autojunk参数。

*类*difflib.Differ

这是一个用于比较文本行序列并生成人类可读的差异或增量的类。Differ使用SequenceMatcher两者来比较行的序列，并比较类似（近似匹配）行内的字符序列。

Differ增量的每一行都以两个字母的代码开头：

码	含义
'- '	线1是唯一的
'+ '	线2是唯一的
' '	这两条序列通用
'? '	线不存在于任何输入序列中

以'?'开始的行试图引导眼睛进入内部的区别，并且不存在于任何输入序列中。如果序列包含制表符，这些行可能会引起混淆。

*类*difflib.HtmlDiff

这个类可以用来创建一个HTML表格（或一个包含该表格的完整的HTML文件），并排显示文本与行间和行内变化高亮的逐行比较。该表格可以以完整或上下文差异模式生成。

这个类的构造函数是：

__init__（*tabsize = 8*，*wrapcolumn = None*，*linejunk = None*，*charjunk = IS_CHARACTER_JUNK*）

初始化的实例HtmlDiff。

*tabsize*是一个可选的关键字参数，用于指定制表位间距和默认值8。

*wrapcolumn*是一个可选的关键字，用于指定行被破坏和包装的列号，默认为None不包装行的位置。

*linejunk*和*charjunk*是传入的可选关键字参数ndiff()（用于HtmlDiff生成并排的HTML差异）。请参阅 ndiff()参数默认值和说明的文档。

以下方法是公开的：

make_file（*fromlines*，*tolines*，*fromdesc =''*，*todesc =''*，*context = False*，*numlines = 5*，***，*charset ='utf-8'*）

比较*fromlines*和*tolines*（字符串列表）并返回一个字符串，该字符串是一个完整的HTML文件，其中包含一个表格，显示逐行显示的行间差异和行内变化。

*fromdesc*和*todesc*是可选的关键字参数，用于指定文件列标题字符串（均默认为空字符串）。

*上下文*和*numlines*都是可选的关键字参数。设置*上下文*以显示 True上下文差异，否则默认 False显示完整文件。*numlines*默认为5。当*上下文*为True *numlines*时，控制围绕差异亮点的上下文行数。当*上下文*为False *numlines*时，控制在使用“下一个”超链接时显示在差别突出显示之前的行数（设置为零将导致“下一个”超链接将下一个突出显示放在浏览器顶部，而没有任何前导上下文）。

*在版本3.5中更改：*仅添加了*charset*关键字参数。HTML文档的默认字符集从更改’ISO-8859-1’为’utf-8’。

make_table（*fromlines*，*tolines*，*fromdesc =''*，*todesc =''*，*context = False*，*numlines = 5*）

比较*fromlines*和*tolines*（字符串列表）并返回一个字符串，它是一个完整的HTML表格，显示逐行显示的行间差异和行内变化。

该方法的参数与该方法的参数相同make_file()。

Tools/scripts/diff.py 是这个类的命令行前端，包含了它的一个很好的例子。

difflib.context_diff（*a*，*b*，*fromfile =''*，*tofile =''*，*fromfiledate =''*，*tofiledate =''*，*n = 3*，*lineterm ='\n'*）

比较*a*和*b*（字符串列表）；以上下文差异格式返回一个delta（一个产生delta行的发生器）。

上下文差异是一种紧凑的方式，只显示已经改变的行以及几行上下文。所做的更改以前/后样式显示。上下文行的数量由*n*设置，缺省值为3。

默认情况下，diff控制行（带有***或的---）是使用尾随换行符创建的。这是有用的 io.IOBase.readlines()，io.IOBase.writelines()因为由于输入和输出都有尾随的换行符，因此适用于差异结果 的输入产生。

对于没有尾随换行符的输入，请将*lineterm*参数设置为 `""` 使输出将统一换行。

上下文差异格式通常具有文件名和修改时间的标题。任何或所有这些可以使用*fromfile*，*tofile*，*fromfiledate*和*tofiledate*字符串指定。修改时间通常以ISO 8601格式表示。如果未指定，则字符串默认为空白。

```
>>> s1 = ['bacon\n', 'eggs\n', 'ham\n', 'guido\n']
>>> s2 = ['python\n', 'eggy\n', 'hamster\n', 'guido\n']
>>> sys.stdout.writelines(context_diff(s1, s2, fromfile='before.py', tofile='after
*** before.py
--- after.py
***************
*** 1,4 ****
! bacon
! eggs
! ham
  guido
--- 1,4 ----
! python
! eggy
! hamster
  guido
```

请参阅命令行界面difflib以获取更详细的示例。

difflib.**get_close_matches**(*字*, *可能性*, *n* = *3*, *截止值*= *0.6*)

返回最佳“足够好”匹配的列表。 *单词*是需要紧密匹配的序列（通常是一个字符串），*可能性*是匹配*单词*的序列列表（通常是字符串列表）。

可选参数*n*（默认值3）是要返回的最近匹配的最大数量; *n*必须大于0。

可选参数*截止*（默认值0.6）是范围[0，1]中的浮点数。没有得到至少与*单词*相似的可能性将被忽略。

列表中返回可能性中最好的（不超过*n*个）匹配，按相似性得分排序，最先类似。

```
>>> get_close_matches('appel', ['ape', 'apple', 'peach', 'puppy'])
['apple', 'ape']
>>> import keyword
>>> get_close_matches('wheel', keyword.kwlist)
['while']
>>> get_close_matches('pineapple', keyword.kwlist)
[]
>>> get_close_matches('accept', keyword.kwlist)
['except']
```

difflib.**ndiff**(*a*, *b*, *linejunk* = *None*, *charjunk* = *IS_CHARACTER_JUNK*)

比较*a*和*b*（字符串列表）; 返回一个Differ三角形（生成三角形线的发生器）。

可选关键字参数*linejunk*和*charjunk*是过滤函数（或None）：

linejunk：接受单个字符串参数的函数，如果字符串是垃圾，则返回true;否则返回false。默认是None。还有一个模块级的函数IS_LINE_JUNK()，除了最多一个字符（'#' - ）外，它过滤掉没有可见字符的行，但是底层的SequenceMatcher类会动态地分析哪些行是如此频繁以至于构成噪声，并且这通常起作用比使用此功能更好。

charjunk：接受一个字符（一个长度为1的字符串）的函数，如果该字符是垃圾，则返回;否则返回false。缺省值是模块级函数IS_CHARACTER_JUNK()，它可以过滤掉空格字符（空格或制表符;将新行包含在这里是个不错的主意！）。

Tools/scripts/ndiff.py 是这个函数的命令行前端。

```
>>> diff = ndiff('one\ntwo\nthree\n'.splitlines(keepends=True),
...              'ore\ntree\nemu\n'.splitlines(keepends=True))
>>> print(''.join(diff), end="")
- one
?  ^
+ ore
?  ^
- two
- three
?  -
+ tree
+ emu
```

difflib.**restore**(*sequence*, *which*)

返回生成增量的两个序列之一。

给定由or生成的*序列*，从文件1或2（参数*which*）中提取行，剥离行前缀。

Differ.compare() ndiff()

例：

```
>>> diff = ndiff('one\ntwo\nthree\n'.splitlines(keepends=True),
...              'ore\ntree\nemu\n'.splitlines(keepends=True))
>>> diff = list(diff) # materialize the generated delta into a list
>>> print(''.join(restore(diff, 1)), end="")
one
two
three
>>> print(''.join(restore(diff, 2)), end="")
ore
tree
emu
```

difflib.**unified_diff**(*a*, *b*, *fromfile*='', *tofile*='', *fromfiledate*='', *tofiledate*='', *n*=3, *lineterm*='\n')

比较*a*和*b*（字符串列表）；以统一差异格式返回一个delta（一个产生delta行的发生器）。

统一差异是一种紧凑的方式，只显示已经改变的行和几行上下文。这些更改以内联样式显示（而不是在块之前/之后单独显示）。上下文行的数量由*n*设置，缺省值为3。

默认情况下，差异控制线（那些`---`，`+++`或`@@`）与尾部换行符创建。这是有用的`io.IOBase.readlines()`，`io.IOBase.writelines()`因为由于输入和输出都有尾随的换行符，因此适用于差异结果 的输入产生。

对于没有尾随换行符的输入，请将*lineterm*参数设置为`""`使输出将统一换行。

上下文差异格式通常具有文件名和修改时间的标题。任何或所有这些可以使用*fromfile*，*tofile*，*fromfiledate*和*tofiledate*字符串指定。修改时间通常以ISO 8601格式表示。如果未指定，则字符串默认为空白。

```
>>> s1 = ['bacon\n', 'eggs\n', 'ham\n', 'guido\n']
>>> s2 = ['python\n', 'eggy\n', 'hamster\n', 'guido\n']
>>> sys.stdout.writelines(unified_diff(s1, s2, fromfile='before.py', tofile='after
--- before.py
+++ after.py
@@ -1,4 +1,4 @@
-bacon
-eggs
-ham
+python
+eggy
+hamster
 guido
```

请参阅命令行界面difflib以获取更详细的示例。

`difflib.`**diff_bytes**(*dfunc*, *a*, *b*, *fromfile* = *b''*, *tofile* = *b''*, *fromfiledate* = *b''*, *tofiledate* = *b''*, *n* = *3*, *lineterm* = *b'\n'*)

使用*dfunc*比较*a*和*b*（字节对象列表）；以*dfunc*返回的格式生成一系列增量行（也是字节）。 *dfunc*必须是可调用的，通常是或者 。`unified_diff()` `context_diff()`

允许您将数据与未知或不一致的编码进行比较。除*n*以外的所有输入必须是字节对象，而不是str。通过无损地将所有输入（*n*除外）转换为str并调用。然后将*dfunc*的输出 转换回字节，因此您收到的增量行与*a*和*b*具有相同的未知/不一致的编码。`dfunc(a, b, fromfile, tofile, fromfiledate, tofiledate, n, lineterm)`

3.5版本中的新功能。

`difflib.`**IS_LINE_JUNK**(*线*)

对于可忽略的线条返回true。线*线*是可忽略的，如果*线*为空或包含一个单一的`'#'`，否则就不是忽略。作为参数的默认*linejunk*在`ndiff()`旧版本中。

`difflib.`**IS_CHARACTER_JUNK**(*ch*)

对于可忽略的字符返回true。如果*ch* 是一个空格或制表符，那么字符*ch*是可以忽略的，否则它是不可忽略的。用作参数*charjunk*中的默认值。`ndiff()`

也可以看看:

模式匹配：完形法

讨论John W. Ratcliff和DE Metzener的类似算法。这是1988年7月在Dobb博士的杂志上发表的。

6.3.1。SequenceMatcher对象

这个SequenceMatcher类有这个构造函数：

class difflib. SequenceMatcher (*isjunk = None* , *a =''* , *b =''* , *autojunk = True*)

可选参数*isjunk*必须是None（缺省值）或一个采用序列元素的单参数函数，并且当且仅当元素是“垃圾”且应该被忽略时返回true。通过None对*isjunk*相当于通过; 换句话说，没有元素被忽略。例如，通过：lambda x: 0

```
lambda x: x in " \t"
```

如果您将线条作为字符序列进行比较，并且不想在空白或硬标签上同步。

可选参数*a*和*b*是要比较的序列; 都默认为空字符串。两个序列的元素必须是可散列的。

可选参数*autojunk*可用于禁用自动垃圾启发式。

*新版本3.2：*在*autojunk*参数。

SequenceMatcher对象获得三个数据属性：*bjunk*是集的元素*b*为哪些*isjunk*是True; *bpopular*是由启发式流行的非垃圾元素集合（如果未被禁用）; *b2j*是一个将*b*的剩余元素映射到它们出现的位置列表的dict 。只要*b*用 set_seqs() 或重置，所有三个都会重置 set_seq2()。

*新版本3.2：*在*bjunk*和*bpopular*属性。

SequenceMatcher 对象有以下方法：

set_seqs (*a* , *b*)

设置要比较的两个序列。

SequenceMatcher计算并缓存关于第二个序列的详细信息，因此如果您想要将一个序列与多个序列进行比较，请使用set_seq2()一次设置常用序列并set_seq1()重复调用一次，每个其他序列一次。

set_seq1 (*a*)

设置要比较的第一个序列。第二个要比较的序列没有改变。

set_seq2 (*b*)

设置要比较的第二个序列。第一个要比较的序列没有改变。

find_longest_match (*alo* , *ahi* , *blo* , *bhi*)

在a[alo:ahi]和中找到最长的匹配块b[blo:bhi]。

如果*isjunk*省略或None，find_longest_match()返回 ，使得等于，其中和。对于所有符合这些条件的，额外的条件，和如果，也达到了。换言之，所有最大匹配块，返回一

个启动在最早一个，以及所有开始最早的那些最大匹配块的一个，返回在开始最早的一个*b*。(i, j, k)a[i:i+k]b[j:j+k]alo <= i <= i+k <= ahiblo <= j <= j+k <= bhi(i', j', k')k >= k'i <= i'i == i'j <= j'

```
>>> s = SequenceMatcher(None, " abcd", "abcd abcd")
>>> s.find_longest_match(0, 5, 0, 9)
Match(a=0, b=4, size=5)
```

如果提供了*isjunk*，则首先确定最长的匹配块，如上所述，但附加的限制是块中不会出现垃圾元素。然后通过匹配（仅）两侧的垃圾元素尽可能扩展该块。因此，除非相同的垃圾恰好毗邻有趣的匹配，否则得到的块从不匹配垃圾。

这里和以前一样，但考虑到空白是垃圾。这可以防止直接匹配第二个序列的尾部。相反，只能匹配，并匹配第二个序列中最左边的部分：' abcd' abcd'abcd'abcd'

```
>>> s = SequenceMatcher(lambda x: x==" ", " abcd", "abcd abcd")
>>> s.find_longest_match(0, 5, 0, 9)
Match(a=1, b=0, size=4)
```

如果没有块匹配，则返回。(alo, blo, 0)

此方法返回一个指定的元组。Match(a, b, size)

get_matching_blocks()

返回描述匹配子序列的三元组列表。每个三元组都是这种形式，并且意味着。三元组在*i*和*j*中单调递增。(i, j, n)a[i:i+n] == b[j:j+n]

最后一个三元组是虚拟的，并具有价值。这是唯一的三重奏。如果和 在列表中是相邻的三元组，并且第二个不是列表中的最后一个三元组，则或; 换句话说，相邻三元组总是描述不相邻的相等块。(len(a), len(b), 0)n == 0(i, j, n)(i', j', n')i+n != i'j+n != j'

```
>>> s = SequenceMatcher(None, "abxcd", "abcd")
>>> s.get_matching_blocks()
[Match(a=0, b=0, size=2), Match(a=3, b=2, size=2), Match(a=5, b=4, size=0)]
```

get_opcodes()

返回的描述如何把5元组列表一个进入*b*。每个元组都是这种形式。第一个元组具有，其余元组的*i1*等于来自前一个元组的*i2*，并且同样，*j1*等于先前的*j2*。(tag, i1, i2, j1, j2)i1 == j1 == 0

该*标签*值是字符串，这些含义：

值	含义
'replace'	a[i1:i2]应该被替换 b[j1:j2]。
'delete'	a[i1:i2]应该删除。请注意， 在这种情况下。j1 == j2
'insert'	b[j1:j2]应插入 a[i1:i1]。请注意，在这种情况下。i1 == i2
'equal'	a[i1:i2] == b[j1:j2] （子序列相等）。

例如：

```
>>> a = "qabxcd"
>>> b = "abycdf"
>>> s = SequenceMatcher(None, a, b)
>>> for tag, i1, i2, j1, j2 in s.get_opcodes():
...     print('{:7}   a[{}:{}] --> b[{}:{}] {!r:>8} --> {!r}'.format(
...         tag, i1, i2, j1, j2, a[i1:i2], b[j1:j2]))
delete    a[0:1] --> b[0:0]      'q' --> ''
equal     a[1:3] --> b[0:2]     'ab' --> 'ab'
replace   a[3:4] --> b[2:3]      'x' --> 'y'
equal     a[4:6] --> b[3:5]     'cd' --> 'cd'
insert    a[6:6] --> b[5:6]       '' --> 'f'
```

get_grouped_opcodes (*n = 3*)

返回最多包含*n*行上下文的组的生成器。

从返回的组开始get_opcodes() ，此方法分割出更小的变更集群，并消除没有变化的干预范围。

这些组以与格式相同的格式返回get_opcodes()。

ratio ( )

将范围[0，1]中的浮点数返回到序列相似度的度量。

其中T是两个序列中元素的总数，M是匹配的数量，这是2.0 * M / T.请注意，这是1.0序列是否相同，以及0.0它们没有共同之处。

这是计算，如果昂贵的get_matching_blocks()或 get_opcodes()尚未被调用，在这种情况下，你可能想尝试quick_ratio()或real_quick_ratio()抢先拿到的上限。

quick_ratio ( )

ratio()相对较快地返回上限。

real_quick_ratio ( )

ratio()很快返回上限。

由于不同的逼近级别，返回匹配与总字符比率的三种方法可以给出不同的结果，尽管quick_ratio()并且real_quick_ratio()总是至少与以下一样大 ratio() ：

```
>>> s = SequenceMatcher(None, "abcd", "bcde")
>>> s.ratio()
0.75
>>> s.quick_ratio()
0.75
>>> s.real_quick_ratio()
1.0
```

6.3.2。SequenceMatcher示例

这个例子比较了两个字符串，考虑到空格是“垃圾”：

```
>>> s = SequenceMatcher(lambda x: x == " ",
...                     "private Thread currentThread;",
...                     "private volatile Thread currentThread;")
```

ratio()在[0，1]中返回一个浮点数，测量序列的相似度。作为一个经验法则，ratio()超过0.6的值意味着序列是近似匹配的：

```
>>> print(round(s.ratio(), 3))
0.866
```

如果你只对序列匹配的地方感兴趣，那么 get_matching_blocks()很方便：

```
>>> for block in s.get_matching_blocks():
...     print("a[%d] and b[%d] match for %d elements" % block)
a[0] and b[0] match for 8 elements
a[8] and b[17] match for 21 elements
a[29] and b[38] match for 0 elements
```

请注意，返回的最后一个元组get_matching_blocks()始终是一个哑元，并且这是唯一的情况下最后一个元组元素（匹配的元素数）是。(len(a), len(b), 0) 0

如果您想知道如何将第一个序列更改为第二个序列，请使用 get_opcodes()：

```
>>> for opcode in s.get_opcodes():
...     print("%6s a[%d:%d] b[%d:%d]" % opcode)
 equal a[0:8] b[0:8]
insert a[8:8] b[8:17]
 equal a[8:29] b[17:38]
```

也可以看看:

- 该get_close_matches()模块中的功能显示了如何使用简单的代码构建SequenceMatcher来完成有用的工作。
- 用于构建小型应用程序的简单版本控制配方SequenceMatcher。

6.3.3。不同对象

请注意，Differ生成的增量不会声称是**最小的** 差异。相反，最小差异常常是违反直觉的，因为它们可以在任何可能的地方同步，有时意外地相隔100页。限制连续匹配的同步点保留了一些局部性的概念，偶尔会产生更长的差异。

这个Differ类有这个构造函数：

class difflib.**Differ**(*linejunk = None*，*charjunk = None*)

可选的关键字参数*linejunk*和*charjunk*用于过滤器函数（或None）：

linejunk：接受单个字符串参数的函数，如果字符串是垃圾，则返回true。默认值是None，这意味着没有行被认为是垃圾。

charjunk：接受单个字符参数（长度为1的字符串）的函数，如果字符是垃圾，则返回true。默认值是None，这意味着没有人物被认为是垃圾。

这些垃圾过滤功能可加速匹配以找出差异，并且不会导致任何不同的行或字符被忽略。请阅读 find_longest_match() 方法*isjunk* 参数的说明以获取解释。

Differ 通过一种方法使用对象（生成增量）：

compare（*a*，*b* ）

比较两个序列的行，并生成delta（一系列行）。

每个序列必须包含以换行符结尾的单个单行字符串。这种序列可以从readlines()文件类对象的方法中获得 。生成的增量也由换行符终止的字符串组成，可以通过writelines()文件类对象的方法按原样打印。

6.3.4。不同示例

这个例子比较两个文本。首先我们设置文本，以换行符结尾的单个单行字符串序列（这些序列也可以从readlines()文件类对象的方法中获得）：

```
>>> text1 = '''  1. Beautiful is better than ugly.
...   2. Explicit is better than implicit.
...   3. Simple is better than complex.
...   4. Complex is better than complicated.
... '''.splitlines(keepends=True)
>>> len(text1)
4
>>> text1[0][-1]
'\n'
>>> text2 = '''  1. Beautiful is better than ugly.
...   3.   Simple is better than complex.
...   4. Complicated is better than complex.
...   5. Flat is better than nested.
... '''.splitlines(keepends=True)
```

接下来我们实例化一个Differ对象：

```
>>> d = Differ()
```

请注意，在实例化Differ对象时，我们可能会传递函数以过滤掉行和字符“垃圾”。有关Differ()详细信息，请参阅构造函数。

最后，我们比较两者：

```
>>> result = list(d.compare(text1, text2))
```

result 是一个字符串列表，所以让我们打印它：

```
>>> from pprint import pprint
>>> pprint(result)
['    1. Beautiful is better than ugly.\n',
```

```
'-   2. Explicit is better than implicit.\n',
'-   3. Simple is better than complex.\n',
'+   3.   Simple is better than complex.\n',
'?     ++\n',
'-   4. Complex is better than complicated.\n',
'?            ^                     ---- ^\n',
'+   4. Complicated is better than complex.\n',
'?          ++++ ^                         ^\n',
'+   5. Flat is better than nested.\n']
```

作为单个多行字符串，它看起来像这样：

```
>>> import sys
>>> sys.stdout.writelines(result)
    1. Beautiful is better than ugly.
-   2. Explicit is better than implicit.
-   3. Simple is better than complex.
+   3.   Simple is better than complex.
?     ++
-   4. Complex is better than complicated.
?            ^                     ---- ^
+   4. Complicated is better than complex.
?          ++++ ^                         ^
+   5. Flat is better than nested.
```

6.3.5。difflib的命令行界面

这个例子展示了如何使用difflib来创建一个类似diff的工具。它也包含在Python源代码分发中，如 Tools/scripts/diff.py。

```
#!/usr/bin/env python3
""" Command line interface to difflib.py providing diffs in four formats:

* ndiff:    lists every line and highlights interline changes.
* context:  highlights clusters of changes in a before/after format.
* unified:  highlights clusters of changes in an inline format.
* html:     generates side by side comparison with change highlights.

"""

import sys, os, difflib, argparse
from datetime import datetime, timezone

def file_mtime(path):
    t = datetime.fromtimestamp(os.stat(path).st_mtime,
                               timezone.utc)
    return t.astimezone().isoformat()

def main():

    parser = argparse.ArgumentParser()
    parser.add_argument('-c', action='store_true', default=False,
                        help='Produce a context format diff (default)')
    parser.add_argument('-u', action='store_true', default=False,
```

```
                        help='Produce a unified format diff')
    parser.add_argument('-m', action='store_true', default=False,
                        help='Produce HTML side by side diff '
                             '(can use -c and -l in conjunction)')
    parser.add_argument('-n', action='store_true', default=False,
                        help='Produce a ndiff format diff')
    parser.add_argument('-l', '--lines', type=int, default=3,
                        help='Set number of context lines (default 3)')
    parser.add_argument('fromfile')
    parser.add_argument('tofile')
    options = parser.parse_args()

    n = options.lines
    fromfile = options.fromfile
    tofile = options.tofile

    fromdate = file_mtime(fromfile)
    todate = file_mtime(tofile)
    with open(fromfile) as ff:
        fromlines = ff.readlines()
    with open(tofile) as tf:
        tolines = tf.readlines()

    if options.u:
        diff = difflib.unified_diff(fromlines, tolines, fromfile, tofile, fromdate, to
    elif options.n:
        diff = difflib.ndiff(fromlines, tolines)
    elif options.m:
        diff = difflib.HtmlDiff().make_file(fromlines,tolines,fromfile,tofile,context=
    else:
        diff = difflib.context_diff(fromlines, tolines, fromfile, tofile, fromdate, to

    sys.stdout.writelines(diff)

if __name__ == '__main__':
    main()
```

6.4。 textwrap- 文本包装和填充

源代码： Lib / textwrap.py

该textwrap模块提供了一些便利功能，以及TextWrapper完成所有工作的类。如果你只是包装或填充一个或两个文本字符串，便利功能应该足够好; 否则，你应该使用TextWrapper效率的实例。

textwrap.**wrap**(*文本* , *宽度*= *70* , ** *kwargs*)

包装*文本中*的单个段落（一个字符串），因此每行至多是 *宽度*字符长。返回输出行的列表，没有最终的换行符。

可选的关键字参数对应于TextWrapper下面记录的实例属性。 *宽度*默认为70。

有关其行为TextWrapper.wrap()如何的其他详细信息，请参阅方法 wrap()。

textwrap.**fill**(*文本* , *宽度*= *70* , ** *kwargs*)

包装*文本中*的单个段落，并返回包含包装段落的单个字符串。 fill()是速记

```
"\n".join(wrap(text, ...))
```

特别是，fill()接受与wrap()。完全相同的关键字参数。

textwrap.**shorten**(*文本* , *宽度* , ** *kwargs*)

折叠并截断给定的*文本*以适应给定的*宽度*。

首先，*文本中*的空白符被折叠（所有空白符被单个空格符替代）。如果结果符合*宽度*，则返回。否则，从结尾删除足够的单词，以便剩下的单词加上placeholder适合的内容width：

```
>>> textwrap.shorten("Hello  world!", width=12)
'Hello world!'
>>> textwrap.shorten("Hello  world!", width=11)
'Hello [...]'
>>> textwrap.shorten("Hello world", width=10, placeholder="...")
'Hello...'
```

可选的关键字参数对应于TextWrapper下面记录的实例属性。需要注意的是文本传递给之前的空白被折叠 功能，所以改变的价值，， ，和将没有任何效果。 TextWrapper fill() tabsize expand_tabs drop_whitespace replace_whitespace

3.4版新增功能

textwrap.**dedent**(*文本*)

从*文本中的每*一行删除任何常见的空白字符。

这可以用来使三引号字符串与显示的左边缘对齐，同时仍然以缩进的形式将它们呈现在源代码中。

需要注意的是制表符和空格都被视为空白，但他们不是平等的：线条和被认为没有共同的前导空格。`"  hello""\thello"`

例如：

```
def test():
    # end first line with \ to avoid the empty line!
    s = '''\
    hello
      world
    '''
    print(repr(s))          # prints '    hello\n      world\n    '
    print(repr(dedent(s)))  # prints 'hello\n  world\n'
```

textwrap.**indent**(*文本*, *前缀*, *谓词=无*)

在*文本中*的选定行的开头添加*前缀*。

线路通过呼叫分开`text.splitlines(True)`。

默认情况下，*前缀*被添加到所有不仅由空白（包括任何行尾）组成的行中。

例如：

```
>>> s = 'hello\n\n \nworld'
>>> indent(s, '  ')
'  hello\n\n \n  world'
```

可选*谓词*参数可用于控制缩进的行。例如，很容易为甚至空白和只有空白的行添加*前缀*：

```
>>> print(indent(s, '+ ', lambda line: True))
+ hello
+
+
+ world
```

3.3版本的新功能

`wrap()`，`fill()`并`shorten()`通过创建一个`TextWrapper`实例并调用一个方法来工作。该实例不会被重用，因此对于使用`wrap()`和/或处理许多文本字符串的应用程序，`fill()`创建自己的`TextWrapper`对象可能更有效。

文本最好包装在空格中，紧接连字符后的连字符后面；只有这样，如果有必要，长词将被打破，除非`TextWrapper.break_long_words`被设置为假。

class textwrap.**TextWrapper**(***kwargs*)

该`TextWrapper`构造函数接受多种可选关键字参数。每个关键字参数都对应一个实例属性，例如

```
wrapper = TextWrapper(initial_indent="* ")
```

是相同的

```
wrapper = TextWrapper()
wrapper.initial_indent = "* "
```

您可以TextWrapper多次重复使用同一个对象，并且可以通过直接分配使用之间的实例属性来更改其任何选项。

的TextWrapper是实例属性（和关键字参数构造函数），如下所示：

width
: （默认70：）包装行的最大长度。只要输入文本中没有单独的单词长于width，就TextWrapper保证没有输出行比 width字符长。

expand_tabs
: （默认值：）True如果为true，则*文本*中的所有制表符将被扩展为使用*文本*expandtabs()方法的空格。

tabsize
: （默认值：）8如果expand_tabs为true，则*文本*中的所有制表符将被展开为零个或多个空格，具体取决于当前列和给定的制表符大小。

3.3版本的新功能

replace_whitespace
: （默认值：）True如果为true，则在标签扩展之后但在包装之前，该wrap()方法将用一个空格替换每个空白字符。替换的空白字符如下所示：选项卡，换行符，垂直制表符，换页符和回车符（'\t\n\v\f\r'）。

  注意: 如果expand_tabs为假，并且replace_whitespace是真实的，每个选项卡字符将通过一个单一的空间，这是被替换*不*一样制表符扩展。

  注意: 如果replace_whitespace为false，换行符可能会出现在一行的中间并导致奇怪的输出。出于这个原因，文本应该被拆分成str.splitlines()分开包装的段落（使用或类似）。

drop_whitespace
: （默认值True：）如果为true，则会删除每行的开头和结尾处的空白（包装之后但缩进之前）。但是，如果非空白符号跟随，则在段落开始处的空格不会被删除。如果被删除的空白占用了整行，整行将被删除。

initial_indent
: （默认值''：）将被预置到包装输出的第一行的字符串。计算第一行的长度。空字符串不缩进。

subsequent_indent
: （默认值''：）除了第一个以外，将被预置于所有包装输出行的字符串。计算除第一行以外的每行的长度。

fix_sentence_endings

（默认值：）False如果为true，则TextWrapper尝试检测句子结尾并确保句子总是由两个空格分隔。这通常用于等宽字体的文本。然而，这句话检测算法是不完美的：它假定一个句子的结尾由一个小写字母后跟之一'.'，'!'或者'?'，可能还跟着一个'"'或"'"，后面加一个空格。与此有关的一个问题是算法是无法检测到"Dr."之间的差异

```
[...] Dr. Frankenstein's monster [...]
```

和"Spot"

```
[...] See Spot. See Spot run [...]
```

fix_sentence_endings 默认为false。

由于句子检测算法依赖于string.lowercase"小写字母"的定义，并且在一段时间后使用两个空格来在同一行上分隔句子，所以它是特定于英语文本的。

break_long_words

（默认：）True如果为true，那么长度超过的单词width将被破坏，以确保没有多行比单词长width。如果它是错误的，长词不会被打破，有些行可能会比width。（长字将自行放在一行中，以尽量减少width超过的数量。）

break_on_hyphens

（默认值True：）如果为true，则最好在复合词中的空白和连字符后面进行包装，因为这在英语中是习惯的。如果为false，则只有空格才会被视为换行符的潜在好位置，但break_long_words如果您想要真正不易驾驭的单词，则需要将其设置为false。以前版本的默认行为是始终允许打破带连字符的单词。

max_lines

（默认值：）None如果不是None，那么输出将至多包含 *max_lines*行，并且*占位符*出现在输出的末尾。

3.4版新增功能

placeholder

（默认值：）如果输出文本被截断，它将出现在输出文本的末尾。' [...]'

3.4版新增功能

TextWrapper 还提供了一些公共方法，类似于模块级别的便利功能：

wrap（*文本*）

将*文本中*的单个段落（一个字符串）包装起来，这样每一行最多只有一个 width字符。所有包装选项都取自实例的实例属性TextWrapper。返回输出行的列表，没有最终的换行符。如果包装的输出没有内容，则返回的列表为空。

fill（*文本*）

包装*文本中*的单个段落，并返回包含包装段落的单个字符串。

6.5。unicodedata- Unicode数据库

此模块提供对Unicode字符数据库（UCD）的访问，该字符数据库为所有Unicode字符定义字符属性。此数据库中包含的数据是从UCD 9.0.0版编译而来的。

该模块使用Unicode标准附件＃44 “Unicode字符数据库”中定义的相同名称和符号。它定义了以下功能：

unicodedata.lookup（*名字*）
按名称查找字符。如果找到具有给定名称的字符，则返回相应的字符。如果没有找到，KeyError则提出。

*在版本3.3中进行了更改：*已添加对名称别名[1]和命名序列[2]的支持。

unicodedata.name（*chr*[，*default*]）
以字符串形式返回分配给字符*chr*的名称。如果没有定义名称，则返回*默认值*，否则ValueError引发。

unicodedata.decimal（*chr*[，*default*]）
以整数形式返回分配给字符*chr*的十进制值。如果没有定义这样的值，则返回*缺省*值，否则ValueError引发*缺省*值。

unicodedata.digit（*chr*[，*default*]）
以整数形式返回分配给字符*chr*的数字值。如果没有定义这样的值，则返回*缺省*值，否则ValueError引发*缺省*值。

unicodedata.numeric（*chr*[，*default*]）
以float形式返回分配给字符*chr*的数字值。如果没有定义这样的值，则返回*缺省*值，否则ValueError引发*缺省*值。

unicodedata.category（*chr*）
以字符串形式返回分配给字符*chr*的常规类别。

unicodedata.bidirectional（*chr*）
以字符串形式返回分配给字符*chr*的双向类。如果没有定义这样的值，则返回空字符串。

unicodedata.combining（*chr*）
以整数形式返回分配给字符*chr*的规范组合类。0如果未定义组合类，则返回。

unicodedata.east_asian_width（*chr*）
以字符串形式返回分配给字符*chr*的东亚宽度。

unicodedata.mirrored（*chr*）
以整数形式返回分配给字符*chr*的镜像属性。1如果该字符在双向文本中被识别为“镜像”字符，0则返回，否则返回。

unicodedata.**decomposition**(*chr*)

以字符串形式返回分配给字符*chr*的字符分解映射。如果没有定义这样的映射，则返回空字符串。

unicodedata.**normalize**(*form*, *unistr*)

返回Unicode字符串*unistr*的常规表单*形式*。*表单*的有效值为'NFC'，'NFKC'，'NFD'和'NFKD'。

Unicode标准根据规范等价和兼容性等价的定义来定义Unicode字符串的各种规范化形式。在Unicode中，可以用各种方式表示几个字符。例如，字符U + 00C7（带有CEDILLA的拉丁大写字母C）也可以表示为序列U + 0043（拉丁文大写字母C）U + 0327（CEDILLA组合）。

对于每个字符，有两种正常形式：标准形式C和标准形式D.标准形式D（NFD）也称为规范分解，并将每个字符转换为其分解形式。标准形式C（NFC）首先应用规范分解，然后再组合预先组合的字符。

除了这两种形式外，还有两种基于兼容性等效的额外正常形式。在Unicode中，支持通常与其他字符统一的某些字符。例如，U + 2160（ROMAN NUMERAL ONE）与U + 0049（拉丁大写字母I）确实是相同的东西。但是，Unicode支持与现有字符集兼容（例如gb2312）。

标准格式KD（NFKD）将应用兼容性分解，即将所有兼容性字符替换为它们的等价物。标准形式KC（NFKC）首先应用兼容性分解，然后是规范组合。

即使两个unicode字符串被标准化，并且与人类阅读器看起来相同，如果一个字符组合了字符而另一个字符串没有，它们可能不会相等。

另外，该模块公开以下常量：

unicodedata.**unidata_version**

此模块中使用的Unicode数据库的版本。

unicodedata.**ucd_3_2_0**

这是一个与整个模块具有相同方法的对象，但对于需要此特定版本的Unicode数据库（如IDNA）的应用程序，则使用Unicode数据库版本3.2。

例子：

```
>>> import unicodedata
>>> unicodedata.lookup('LEFT CURLY BRACKET')
'{'
>>> unicodedata.name('/')
'SOLIDUS'
>>> unicodedata.decimal('9')
9
>>> unicodedata.decimal('a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: not a decimal
>>> unicodedata.category('A')  # 'L'etter, 'u'ppercase
```

```
'Lu'
>>> unicodedata.bidirectional('\u0660') # 'A'rabic, 'N'umber
'AN'
```

脚注

[1] http://www.unicode.org/Public/9.0.0/ucd/NameAliases.txt

[2] http://www.unicode.org/Public/9.0.0/ucd/NamedSequences.txt

6.6。 stringprep- Internet字符串准备

源代码： Lib / stringprep.py

在识别互联网中的东西（例如主机名称）时，通常需要比较这些“平等”的标识。具体如何执行此比较可能取决于应用程序域，例如它是否应该不区分大小写。可能还需要限制可能的标识，以仅允许由“可打印”字符组成的标识。

RFC 3454定义了一个用于在Internet协议中“准备”Unicode字符串的过程。在将线传递到线上之前，将它们用准备程序处理，之后它们具有一定的规范化形式。RFC定义了一组表，这些表可以组合成配置文件。每个配置文件必须定义它使用哪些表，以及stringprep 过程的其他可选部分是配置文件的一部分。 stringprep配置文件的一个示例是 nameprep ，用于国际化域名。

该模块stringprep只公开RFC 3454中的表格。由于这些表格将非常大以将它们表示为字典或列表，因此该模块在内部使用Unicode字符数据库。模块源代码本身是使用该mkstringprep.py实用程序生成的。

结果，这些表作为函数而不是数据结构。RFC中有两种表格：集合和映射。对于一个集合，stringprep提供“特征函数”，即一个函数，如果参数是集合的一部分，则返回true。对于映射，它提供了映射函数：给定键，它返回相关的值。以下是模块中所有可用功能的列表。

stringprep.**in_table_a1**（*代码*）
确定*代码*是否在表A.1中（Unicode 3.2中的未分配代码点）。

stringprep.**in_table_b1**（*代码*）
确定*代码*是否在表B.1中（通常映射为空）。

stringprep.**map_table_b2**（*代码*）
根据表B.2（用于与NFKC一起使用的案例折叠映射）返回*代码*的映射值。

stringprep.**map_table_b3**（*代码*）
根据表B.3 返回映射的*代码*值（使用没有规范化的情况下折叠映射）。

stringprep.**in_table_c11**（*代码*）
确定*代码*是否在tableC.1.1（ASCII空格字符）中。

stringprep.**in_table_c12**（*代码*）
确定*代码*是否在表C.1.2（非ASCII空格字符）中。

stringprep.**in_table_c11_c12**（*代码*）
确定*代码*是否在表C.1中（空格字符，C.1.1和C.1.2的联合）。

stringprep.**in_table_c21**（*代码*）
确定*代码*是否在表C.2.1（ASCII控制字符）中。

stringprep.**in_table_c22**（*代码*）

确定*代码*是否在表C.2.2（非ASCII控制字符）中。

stringprep.in_table_c21_c22（*代码*）
确定*代码*是否在表C.2中（控制字符，C.2.1和C.2.2的联合）。

stringprep.in_table_c3（*代码*）
确定*代码*是否在表C3中（私人使用）。

stringprep.in_table_c4（*代码*）
确定*代码*是否在表C.4中（非字符代码点）。

stringprep.in_table_c5（*代码*）
确定*代码*是否在表C.5（代理代码）中。

stringprep.in_table_c6（*代码*）
确定*代码*是否在表C.6中（不适用于纯文本）。

stringprep.in_table_c7（*代码*）
确定*代码*是否在表C.7中（不适用于规范表示）。

stringprep.in_table_c8（*代码*）
确定*代码*是否在表C.8中（更改显示属性或已弃用）。

stringprep.in_table_c9（*代码*）
确定*代码*是否在表C.9中（标记字符）。

stringprep.in_table_d1（*代码*）
确定*代码*是否在表D.1中（具有双向属性“R”或“AL”的字符）。

stringprep.in_table_d2（*代码*）
确定*代码*是否在表D.2中（具有双向属性“L”的字符）。

6.7。readline- GNU readline接口

该readline模块定义了许多函数以方便Python解释器完成和读取/写入历史文件。该模块可以直接使用，也可以通过rlcompleter支持在交互式提示符下完成Python标识符的模块使用。使用此模块进行的设置会影响解释程序的交互提示和内置input() 函数提供的提示的行为。

注意: 底层Readline库API可以由libedit库而不是GNU readline 来实现。在MacOS X上，readline模块会在运行时检测正在使用哪个库。

配置文件libedit与GNU readline 的配置文件不同。如果以编程方式加载配置字符串，则可以检查文本“libedit” readline.__doc__ 以区分GNU readline和libedit。

Readline键绑定可以通过初始化文件进行配置，通常 .inputrc在您的主目录中。请参阅 GNU Readline手册中的Readline Init File，了解该文件的格式和允许的结构以及一般Readline库的功能。

6.7.1。初始文件

以下函数与init文件和用户配置有关：

readline.**parse_and_bind**(*字符串*)
执行*字符串*参数中提供的init行。这调用 rl_parse_and_bind()了底层库。

readline.**read_init_file**([*文件名*])
执行一个readline初始化文件。默认文件名是最后使用的文件名。这调用 rl_read_init_file()了底层库。

6.7.2。行缓冲区

以下功能在线路缓冲区上运行：

readline.**get_line_buffer**( )
返回行缓冲区的当前内容（rl_line_buffer 在底层库中）。

readline.**insert_text**(*字符串*)
将文本插入光标位置的行缓冲区中。这会调用 rl_insert_text()底层库，但会忽略返回值。

readline.**redisplay**( )
更改屏幕上显示的内容以反映行缓冲区的当前内容。这调用rl_redisplay()了底层库。

6.7.3。历史文件

以下功能在历史文件上运行：

readline.**read_history_file**([*文件名*])
加载一个readline历史文件，并将其附加到历史列表中。默认文件名是~/.history。这调用read_history()了底层库。

readline.**write_history_file**([*文件名*])
将历史列表保存到readline历史文件，覆盖任何现有文件。默认文件名是~/.history。这调用 write_history()了底层库。

readline.**append_history_file**(*nelements* [, *filename*])
将历史最后的*元素*项添加到文件中。默认文件名是 ~/.history。该文件必须已经存在。这调用 append_history()了底层库。这个函数只有在Python为支持它的库版本编译时才存在。

3.5版本中的新功能。

readline.**get_history_length**( )
readline.**set_history_length**(*长度*)
设置或返回所需的行数以保存在历史文件中。该write_history_file()函数使用此值通过调用history_truncate_file()底层库来截断历史文件。负值意味着无限的历史文件大小。

6.7.4。历史列表

以下功能在全局历史列表上运行：

readline.**clear_history**( )
清除当前的历史记录。这调用clear_history()了底层库。如果Python是为支持它的库的一个版本编译的，Python函数才存在。

readline.**get_current_history_length**( )
返回历史中当前项目的数量。（这get_history_length()与之不同 ，它返回将写入历史文件的最大行数。）

readline.**get_history_item**(*索引*)
在*索引*处返回历史项目的当前内容。项目索引是基于一个的。这调用history_get()了底层库。

readline.**remove_history_item**(*pos*)
从历史记录中删除由其位置指定的历史记录项目。该职位是从零开始的。这调用remove_history()了底层库。

readline.**replace_history_item**(*pos* , *line*)
通过替换与位置指定的历史项目*线*。该职位是从零开始的。这调用replace_history_entry()了底层库。

readline.**add_history**(*线*)

将*行*追加到历史记录缓冲区，就好像它是最后一行输入一样。这调用add_history()了底层库。

readline.set_auto_history(*启用*)

add_history()通过readline读取输入时，启用或禁用自动调用。该*启用的*参数应该是一个布尔值，如果为true，使汽车的历史，而且当假的，禁用自动历史。

3.6版本中的新功能。

CPython实现细节：默认启用自动历史记录，对其进行的更改不会在多个会话中持续存在。

6.7.5。启动钩

readline.set_startup_hook([*function*])

设置或移除由rl_startup_hook 基础库的回调调用的函数。如果指定了*函数*，它将被用作新的钩子函数; 如果省略或者None已经安装的任何功能被删除。在readline打印第一个提示之前，钩子被调用时没有参数。

readline.set_pre_input_hook([*function*])

设置或移除由rl_pre_input_hook 基础库的回调调用的函数。如果指定了*函数*，它将被用作新的钩子函数; 如果省略或者None已经安装的任何功能被删除。在第一个提示已被打印之后，在readline开始读取输入字符之前，钩子被调用时没有参数。这个函数只有在Python为支持它的库版本编译时才存在。

6.7.6。完成

以下功能与实现自定义字完成功能有关。这通常由Tab键操作，并且可以建议并自动完成输入的单词。默认情况下，Readline被设置为用于rlcompleter为交互式解释器完成Python标识符。如果该readline模块要与自定义完成器一起使用，则应设置一组不同的单词分隔符。

readline.set_completer([*function*])

设置或删除完成者功能。如果指定了*函数*，它将用作新的完成函数; 如果省略，或者None已经安装的完成程序功能被删除。完成者函数被调用为 ，对*国家*的，，，...，直到它返回一个非字符串值。它应该返回从*文本*开始的下一个可能的完成。function(text, state)012

已安装的完成程序功能由传入到基础库中的*entry_func*回调调用rl_completion_matches()。该*文本*字符串来自于第一个参数rl_attempted_completion_function的基础库的回调。

readline.get_completer()

获取完成功能，或者None如果没有设置完成功能。

readline.get_completion_type()

获取正在尝试的完成类型。这会rl_completion_type以整数形式返回 底层库中的变量。

readline.**get_begidx**( )
readline.**get_endidx**( )
获取完成范围的开始或结束索引。这些索引是传递给底层库回调的*开始*和*结束*参数 rl_attempted_completion_function。

readline.**set_completer_delims**(*字符串*)
readline.**get_completer_delims**( )
设置或获取单词分隔符以完成。这些决定了要考虑完成的单词的开始（完成范围）。这些函数访问rl_completer_word_break_characters 底层库中的变量。

readline.**set_completion_display_matches_hook**([*function*])
设置或删除完成显示功能。如果指定了*功能*，它将用作新的完成显示功能; 如果省略，或者 None 已经安装的任何完成显示功能被删除。这会设置或清除 rl_completion_display_matches_hook底层库中的 回调。完成显示功能被称为 每次匹配需要显示一次。function(substitution, [matches], longest_match_length)

6.7.7。示例

以下示例演示如何使用readline模块的历史读取和写入功能自动加载并保存.python_history从用户主目录命名的历史文件。下面的代码通常会在用户的交互式会话期间自动执行 PYTHONSTARTUP 文件。

```
import atexit
import os
import readline

histfile = os.path.join(os.path.expanduser("~"), ".python_history")
try:
    readline.read_history_file(histfile)
    # default history len is -1 (infinite), which may grow unruly
    readline.set_history_length(1000)
except FileNotFoundError:
    pass

atexit.register(readline.write_history_file, histfile)
```

当Python以交互模式运行时，这段代码实际上会自动运行（请参阅Readline配置）。

以下示例通过追加新历史来实现相同的目标，但支持并发交互式会话。

```
import atexit
import os
import readline
histfile = os.path.join(os.path.expanduser("~"), ".python_history")

try:
    readline.read_history_file(histfile)
    h_len = readline.get_current_history_length()
except FileNotFoundError:
    open(histfile, 'wb').close()
```

```
    h_len = 0

def save(prev_h_len, histfile):
    new_h_len = readline.get_current_history_length()
    readline.set_history_length(1000)
    readline.append_history_file(new_h_len - prev_h_len, histfile)
atexit.register(save, h_len, histfile)
```

以下示例将该code.InteractiveConsole类扩展为支持历史保存/恢复。

```
import atexit
import code
import os
import readline

class HistoryConsole(code.InteractiveConsole):
    def __init__(self, locals=None, filename="<console>",
                 histfile=os.path.expanduser("~/.console-history")):
        code.InteractiveConsole.__init__(self, locals, filename)
        self.init_history(histfile)

    def init_history(self, histfile):
        readline.parse_and_bind("tab: complete")
        if hasattr(readline, "read_history_file"):
            try:
                readline.read_history_file(histfile)
            except FileNotFoundError:
                pass
            atexit.register(self.save_history, histfile)

    def save_history(self, histfile):
        readline.set_history_length(1000)
        readline.write_history_file(histfile)
```

6.8。rlcompleter- GNU readline的完成函数

源代码： Lib / rlcompleter.py

该rlcompleter模块readline通过完成有效的Python标识符和关键字来定义适合该模块的完成功能。

当该模块在具有readline可用模块的Unix平台上导入时，Completer会自动创建该类的一个实例，并将其complete()方法设置为readline完成者。

例：

```
>>> import rlcompleter
>>> import readline
>>> readline.parse_and_bind("tab: complete")
>>> readline. <TAB PRESSED>
readline.__doc__          readline.get_line_buffer(  readline.read_init_file(
readline.__file__         readline.insert_text(      readline.set_completer(
readline.__name__         readline.parse_and_bind(
>>> readline.
```

该rlcompleter模块设计用于Python的 交互模式。除非Python使用该-S选项运行，否则 该模块将自动导入并配置（请参阅Readline配置）。

在没有的平台上readline，Completer此模块定义的类仍可用于自定义目的。

6.8.1。Completer对象

Completer对象具有以下方法：

Completer.complete(*文本*, *状态*)

返回*文本的*完成*状态*。

如果要求*的文本*不包括一段字符（'.'），它将完成从当前定义的名称__main__，builtins（由定义和关键字keyword模块）。

如果调用一个虚线名称，它将尝试评估任何没有明显副作用（函数不会被评估，但可以生成调用 __getattr__() ）的东西，直到最后一部分，并通过该dir()函数查找其余部分的匹配。表达式评估过程中发生的任何异常都会被捕获，静音并None返回。

7.二进制数据服务

本章介绍的模块提供了一些处理二进制数据的基本服务操作。其他有关二进制数据的操作，特别是有关文件格式和网络协议的操作，将在相关章节中介绍。

在文本处理服务下描述的一些库也可以与ASCII兼容的二进制格式（例如`re`）或所有二进制数据（例如`difflib`）一起工作。

另外，请参阅二进制序列类型中的 Python内置二进制数据类型文档 - bytes，bytearray，memoryview。

7.1。 struct- 将字节解释为打包的二进制数据

源代码： Lib / struct.py

这个模块执行Python值和表示为Python bytes对象的C结构之间的转换。这可用于处理存储在文件或网络连接中的二进制数据以及其他来源。它使用 格式字符串作为C结构布局的紧凑描述，以及从Python值转换的预期转换。

> **注意：** 默认情况下，打包给定C结构的结果包括填充字节，以便为所涉及的C类型保持正确的对齐; 同样，开箱时要考虑到对齐。选择此行为以使打包结构的字节完全对应于相应C结构的内存中的布局。要处理与平台无关的数据格式或省略隐式填充字节，请使用standard大小和对齐而不是 native大小和对齐：有关详细信息，请参阅字节顺序，大小和对齐。

几个struct函数（和方法Struct）需要一个*缓冲区*参数。这是指实现缓冲区协议并提供可读或可读写缓冲区的对象。用于该目的的最常见的类型是bytes和bytearray，但许多其它类型的可以被看作是一个字节数组实现缓冲协议，从以使它们可以被读取/填充无需额外的复制bytes对象。

7.1.1。函数和例外

该模块定义了以下异常和功能：

*异常*struct. error
在各种场合提出例外; 参数是描述错误的字符串。

struct. pack（*fmt*，*v1*，*v2*，...）
返回包含根据格式字符串*fmt*打包的值*v1*，*v2*，... 的字节对象。参数必须完全匹配格式所需的值。

struct. pack_into（*fmt*，*buffer*，*offset*，*v1*，*v2*，...）
根据格式字符串*fmt*打包值*v1*，*v2*，... 并将打包字节写入可写缓冲区*缓冲区*，从位置*偏移*开始。请注意，*偏移量*是必需的参数。

struct. unpack（*fmt*，*buffer*）
根据格式字符串*fmt*从缓冲区*缓冲区*解压缩（可能打包）。结果是一个元组，即使它只包含一个项目。缓冲区的大小（以字节为单位）必须与格式所需的大小相匹配，反映如下。
pack(fmt, ...) calcsize()

struct. unpack_from（*fmt*，*buffer*，*offset = 0*）
根据格式字符串*fmt*，从位置*偏移量*开始的*缓冲区中*解压缩。结果是一个元组，即使它只包含一个项目。缓冲区的大小（以字节为单位）减去*偏移量*必须至少为该格式所需的大小，如所反映的。 calcsize()

struct. iter_unpack（*fmt*，*buffer*）

根据格式字符串*fmt*从缓冲区*缓冲区*解压缩。该函数返回一个迭代器，它将从缓冲区中读取大小相同的块，直到其所有内容都被消耗完。缓冲区的大小（以字节为单位）必须是格式所需大小的倍数，如所反映的。`calcsize()`

每次迭代产生一个由格式字符串指定的元组。

3.4版新增功能

`struct.calcsize(fmt)`

返回与格式字符串*fmt*相对应的结构体的大小（以及由此生成的字节对象的大小 ）。`pack(fmt, ...)`

7.1.2。格式字符串

格式字符串是在打包和解包数据时用于指定预期布局的机制。它们由格式字符构成，它指定了打包/解压缩数据的类型。另外，还有用于控制字节顺序，大小和对齐的特殊字符。

7.1.2.1。字节顺序，大小和对齐

默认情况下，C类型以机器的本机格式和字节顺序表示，并在必要时通过跳过填充字节（根据C编译器使用的规则）进行适当对齐。

或者，根据下表，格式字符串的第一个字符可用于指示打包数据的字节顺序，大小和对齐方式：

字符	字节顺序	尺寸	对准
@	本地人	本地人	本地人
=	本地人	标准	没有
<	小尾数	标准	没有
>	大端	标准	没有
!	网络（=）	标准	没有

网络（=）big-

假设第一个字符不是其中之一'@'。

本地字节顺序是大端或小端，取决于主机系统。例如，Intel x86和AMD64（x86-64）是小端的; 摩托罗拉68000和PowerPC G5是高端的; ARM和Intel Itanium具有可切换的字节序（双字节序）。使用`sys.byteorder`来检查你的系统的字节序。

原生大小和对齐是使用C编译器的`sizeof`表达式确定的 。这总是与本地字节顺序相结合。

标准尺寸仅取决于格式字符; 请参阅格式字符部分中的表格。

需要注意的区别'@'和'='：都使用本地字节顺序，但后者的大小和排列是标准化的。

表单'!'适用于那些声称自己不记得网络字节顺序是大端还是小端的穷人。

没有办法指示非本地字节顺序（强制字节交换）; 使用'<'或的适当选择'>'。

笔记：

1. 填充只会自动添加在连续的结构成员之间。在编码结构的开头或结尾不添加填充。
2. 使用非原生大小和对齐时，例如'<'，'>'，'='和'！'不会添加填充。
3. 要将结构的末尾与特定类型的对齐要求对齐，请使用重复计数为零的该类型的代码结束格式。参见示例。

7.1.2.2。格式字符

格式字符具有以下含义; 考虑到它们的类型，C和Python值之间的转换应该是显而易见的。“标准大小”栏是指使用标准大小时打包值的大小（以字节为单位）; 也就是说，格式字符串以“’<’，’>’”’!’或“”之一开头’=’。使用原生尺寸时，打包值的大小取决于平台。

格式	C型	Python类型	标准尺寸	笔记
x	填充字节	没有价值		
c	char	长度为1的字节	1	
b	signed char	整数	1	（3），（4）
B	unsigned char	整数	1	（3）
?	_Bool	布尔	1	（1）
h	short	整数	2	（3）
H	unsigned short	整数	2	（3）
i	int	整数	4	（3）
I	unsigned int	整数	4	（3）
l	long	整数	4	（3）
L	unsigned long	整数	4	（3）
q	long long	整数	8	（2），（3）
Q	unsigned long long	整数	8	（2），（3）
n	ssize_t	整数		（4）
N	size_t	整数		（4）
e	（7）	浮动	2	（5）
f	float	浮动	4	（5）
d	double	浮动	8	（5）
s	char[]	字节		
p	char[]	字节		
P	void *	整数		（6）

*在版本3.3中进行了更改：*添加了对’n’和’N’格式的支持。

*在版本3.6中进行了更改：*添加了对’e’格式的支持。

笔记：

1. 的’?’转换码对应于_Bool由C99定义的类型。如果此类型不可用，则使用a进行模拟char。在标准模式下，它总是由一个字节表示。

2. 将'q'和'Q'只有在平台C编译器支持C转换代码在本地模式中可用，或者在Windows上，。它们始终可用于标准模式。`long long`__`int64`

3. 当尝试使用任何整数转换代码打包非整数时，如果非整数具有`__index__()`方法，则在打包之前调用该方法将参数转换为整数。

   在版本3.2中进行了更改：`__index__()`对于非整数的使用方法在3.2中是新的。

4. 的'n'和'N'转换码只适用于本机的大小（选择为默认或与'@'字节顺序字符）。对于标准大小，您可以使用适合您的应用程序的其他任何整数格式。

5. 对于'f'，'d'和'e'转换码，填充表示使用IEEE 754 binary32，binary64或binary16格式（'f'，'d'或'e'分别地），而不管由所述平台中使用的浮点格式的。

6. 该'P'格式字符仅适用于本地字节顺序（选择为默认或与'@'字节顺序字符）。字节顺序字符'='选择使用基于主机系统的小端或大端排序。结构模块不会将其解释为本地排序，因此该'P'格式不可用。

7. IEEE 754 binary16"半精度"型在2008年IEEE 754标准修订版中引入。它有一个符号位，一个5位指数和11位精度（显式存储10位），并且可以表示大约`6.1e-05`和`6.5e+04`全精度之间的数字。C编译器没有广泛支持此类型：在典型的机器上，可以使用无符号短整型存储，但不能用于数学运算。有关更多信息，请参阅维基百科页面上的半精度浮点格式。

格式字符之前可以有一个整数重复计数。例如，格式字符串的'4h'含义与"完全相同"'hhhh'。

格式之间的空格字符被忽略; 计数及其格式不能包含空格。

对于's'格式字符，计数被解释为字节的长度，而不是像其他格式字符一样的重复计数; 例如，'10s'意味着一个10字节的字符串，而'10c'意味着10个字符。如果没有给出计数，则默认为1.对于打包，字符串将被截断或填充为空字节，以使其合适。对于解包，结果字节对象总是具有指定的字节数。作为一种特殊情况，'0s'意味着一个单一的空字符串（同时'0c'表示0个字符）。

当包装一个值`x`使用的整数格式（一个'b'，'B'，'h'，'H'，'i'，'I'，'l'，'L'，'q'，'Q'），如果`x`是在有效范围之外为该格式然后`struct.error`上升。

*在版本3.1中更改：*在3.0中，某些整数格式包装超出范围的值并引发`DeprecationWarning`而不是`struct.error`。

的'p'格式字符编码"帕斯卡串"，意思是存储在一个很短的可变长度的字符串*的固定数目的字节*，由计数给出。存储的第一个字节是字符串的长度，或255，以较小者为准。字符串的字节在后面。如果传入的字符串`pack()`太长（比计数减1更长），则仅`count-1`存储字符串的前导字节。如果字符串小于`count-1`，则填充空字节，以便使用所有字节中的精确计数字节。请注意，对于`unpack()`中，'p'格式字符占用`count`的字节，但返回的字符串不能包含超过255个字节。

对于'?'格式字符，返回值是`True`或`False`。打包时，使用参数对象的真值。本地或标准bool表示中的0或1将被打包，并且`True`在解包时会有任何非零值。

7.1.2.3。示例

注意： 所有的例子都假设了一个本地字节顺序，大小，并与一个big-endian机器对齐。

打包/解包三个整数的基本示例：

```
>>> from struct import *
>>> pack('hhl', 1, 2, 3)
b'\x00\x01\x00\x02\x00\x00\x00\x03'
>>> unpack('hhl', b'\x00\x01\x00\x02\x00\x00\x00\x03')
(1, 2, 3)
>>> calcsize('hhl')
8
```

解包字段可以通过将它们分配给变量或通过将结果包装到指定的元组中来命名：

```
>>> record = b'raymond   \x32\x12\x08\x01\x08'
>>> name, serialnum, school, gradelevel = unpack('<10sHHb', record)

>>> from collections import namedtuple
>>> Student = namedtuple('Student', 'name serialnum school gradelevel')
>>> Student._make(unpack('<10sHHb', record))
Student(name=b'raymond   ', serialnum=4658, school=264, gradelevel=8)
```

格式字符的排序可能会影响大小，因为满足对齐要求所需的填充是不同的：

```
>>> pack('ci', b'*', 0x12131415)
b'*\x00\x00\x00\x12\x13\x14\x15'
>>> pack('ic', 0x12131415, b'*')
b'\x12\x13\x14\x15*'
>>> calcsize('ci')
8
>>> calcsize('ic')
5
```

以下格式'llh0l'在末尾指定两个填充字节，假定长度在4个字节的边界上对齐：

```
>>> pack('llh0l', 1, 2, 3)
b'\x00\x00\x00\x01\x00\x00\x00\x02\x00\x03\x00\x00'
```

这只适用于本地大小和对齐有效的情况; 标准尺寸和对齐不强制任何对齐。

也可以看看:

模 array
　　打包的二进制存储的同质数据。

模 xdrlib
　　打包和解压XDR数据。

7.1.3。类

该struct模块还定义了以下类型：

*类*struct.Struct（*格式*）

返回一个新的Struct对象，它根据格式字符串*格式*写入和读取二进制数据。创建一个Struct对象并调用它的方法比调用struct具有相同格式的函数更有效率，因为格式字符串只需编译一次。

编译结构对象支持以下方法和属性：

pack（*v1*，*v2*，...）

与该pack()函数相同，使用编译后的格式。（len(result)将等于size。）

pack_into（*buffer*，*offset*，*v1*，*v2*，...）

与该pack_into()函数相同，使用编译后的格式。

unpack（*缓冲区*）

与该unpack()函数相同，使用编译后的格式。缓冲区的大小以字节为单位必须相等size。

unpack_from（*buffer*，*offset = 0*）

与该unpack_from()函数相同，使用编译后的格式。缓冲区的大小（以字节为单位，减去*偏移量*）必须至少为 size。

iter_unpack（*缓冲区*）

与该iter_unpack()函数相同，使用编译后的格式。缓冲区的大小以字节为单位size。

3.4版新增功能

format

用于构造此Struct对象的格式字符串。

size

计算出的结构体的大小（以及由此pack()方法生成的字节对象的大小）对应于format。

7.2。 codecs-编解码器的注册表和基类

源代码： Lib / codecs.py

此模块为标准Python编解码器（编码器和解码器）定义基类，并提供对内部Python编解码器注册表的访问，该注册表管理编解码器和错误处理查找过程。大多数标准的编解码器都是文本编码，它们将文本编码为字节，但也提供了将文本编码为文本以及将字节编码为字节的编解码器。自定义编解码器可以在任意类型之间进行编码和解码，但某些模块功能仅限于与文本编码或与编码的编解码器 专用 bytes。

该模块为任何编解码器定义了以下用于编码和解码的功能：

codecs. encode（*obj*，*encoding ='utf-8'*，*errors ='strict'*）

编码*OBJ*使用注册的编解码器*编码*。

可能会给出*错误*来设置所需的错误处理方案。默认错误处理程序'strict'意味着编码错误会增加 ValueError（或更多编解码器特定的子类，如 UnicodeEncodeError）。有关编解码器错误处理的更多信息，请参阅编解码器基类。

codecs. decode（*obj*，*encoding ='utf-8'*，*errors ='strict'*）

使用为编码注册的*编码*解码器解码*obj*。

可能会给出*错误*来设置所需的错误处理方案。默认错误处理程序'strict'意味着解码错误会增加 ValueError（或更多的编解码器特定的子类，如 UnicodeDecodeError）。有关编解码器错误处理的更多信息，请参阅编解码器基类。

每个编解码器的全部细节也可以直接查询：

codecs. lookup（*编码*）

在Python编解码器注册表中查找编解码器信息并返回CodecInfo如下定义的 对象。

编码首先在注册表的缓存中查找。如果未找到，则会扫描已注册搜索功能的列表。如果没有CodecInfo找到对象，LookupError则会引发a。否则，该CodecInfo对象存储在缓存中并返回给调用者。

class codecs. CodecInfo（*encode*，*decode*，*streamreader = None*，*streamwriter = None*，*incrementalencoder = None*，*incrementaldecoder = None*，*name = None*）

查找编解码器注册表时的编解码器详细信息。构造函数参数存储在相同名称的属性中：

name

编码的名称。

encode
decode

无状态的编码和解码功能。这些必须是与Codec实例的方法encode()和decode()方法具有相同接口的函数或方法（请参阅编解码器接口）。预期功能或方法在无状态模式下

工作。

incrementalencoder
incrementaldecoder
: 增量编码器和解码器类或工厂功能。这些必须分别提供基类IncrementalEncoder和接口定义的接口 IncrementalDecoder。增量编解码器可以保持状态。

streamwriter
streamreader
: 流式写入器和读取器类或工厂功能。这些必须分别提供基类StreamWriter和接口定义的接口 StreamReader。流编解码器可以维护状态。

为了简化对各种编解码器组件的访问，模块提供了lookup()用于编解码器查找的这些附加功能：

codecs.getencoder(*编码*)
: 查找给定编码的编解码器并返回其编码器功能。

LookupError在无法找到编码的情况下引发一次。

codecs.getdecoder(*编码*)
: 查找给定编码的编解码器并返回其解码器功能。

LookupError在无法找到编码的情况下引发一次。

codecs.getincrementalencoder(*编码*)
: 查找给定编码的编解码器并返回其增量编码器类或工厂功能。

提出了一个LookupError在情况下，编码不能找到或编解码器不支持的增量编码器。

codecs.getincrementaldecoder(*编码*)
: 查找给定编码的编解码器并返回其增量解码器类或工厂功能。

LookupError在编码不能被找到或者编解码器不支持增量解码器的情况下引发a 。

codecs.getreader(*编码*)
: 查找给定编码的编解码器并返回其StreamReader 类或工厂函数。

LookupError在无法找到编码的情况下引发一次。

codecs.getwriter(*编码*)
: 查找给定编码的编解码器并返回其StreamWriter 类或工厂函数。

LookupError在无法找到编码的情况下引发一次。

通过注册合适的编解码器搜索功能可以使用自定义编解码器：

codecs.register(*search_function*)

注册编解码器搜索功能。搜索功能需要一个参数，即所有小写字母的编码名称，并返回一个 CodecInfo对象。如果搜索功能找不到给定的编码，则应该返回None。

> **注意：** 搜索功能注册目前不可逆，这在某些情况下可能会导致问题，如单元测试或模块重新加载。

尽管内置模块open()和相关io模块是推荐的使用编码文本文件的方法，但该模块提供了额外的实用程序函数和类，以便在使用二进制文件时可以使用更广泛的编解码器：

codecs.open（*filename*，*mode='r'*，*encoding = None*，*errors='strict'*，*buffering = 1* ）

使用给定*模式*打开一个编码文件并返回一个实例 StreamReaderWriter，提供透明的编码/解码。默认文件模式是'r'，意思是以读模式打开文件。

> **注意：** 底层编码文件始终以二进制模式打开。'\n'阅读和写作没有自动转换。所述*模式*参数可以是任何二进制模式可接受的内置 open()功能; 这'b'是自动添加的。

*编码*指定要用于该文件的编码。任何对字节进行编码和解码的编码都是允许的，并且文件方法支持的数据类型取决于所使用的编解码器。

可能会给出*错误*来定义错误处理。它默认为在编码错误发生的情况下'strict'导致a ValueError被引发的原因。

*缓冲*与内置open()函数具有相同的含义。它默认为行缓冲。

codecs.EncodedFile（*文件*，*data_encoding*，*file_encoding =无*，*误差='严格'* ）

返回一个StreamRecoder实例，这是一个 提供透明转码的*文件*包装版本。当封装版本关闭时，原始文件关闭。

写入包裹文件数据被根据给定的解码 *data_encoding*，然后写入到原来的文件作为字节使用 *file_encoding*。字节从原始文件读出是根据解码*file_encoding*，其结果是使用编码*data_encoding*。

如果*file_encoding*没有给出，则默认为*data_encoding*。

可能会给出*错误*来定义错误处理。它默认为'strict'，ValueError在发生编码错误的情况下会导致提升。

codecs.iterencode（*iterator*，*encoding*，*errors='strict'*，*** kwargs* ）

使用增量编码器来迭代编码由所提供的输入*的迭代器*。这个功能是一个发生器。的*误差*参数（以及任何其他关键字参数）通过以增量编码器通过。

该功能要求编解码器接受str要编码的文本对象。因此它不支持字节到字节编码器，如base64_codec。

codecs.iterdecode（*iterator*，*encoding*，*errors='strict'*，*** kwargs* ）

使用增量解码器迭代解码所提供的输入*的迭代器*。这个功能是一个发生器。的*误差*参数（以及任何其他关键字参数）通过以增量解码器通过。

此功能要求编解码器接受bytes要解码的对象。因此它不支持文本到文本编码器等rot_13，尽管rot_13可以等同使用 iterencode()。

该模块还提供以下常数，这些常数对读取和写入平台相关文件非常有用：

codecs.BOM
codecs.BOM_BE
codecs.BOM_LE
codecs.BOM_UTF8
codecs.BOM_UTF16
codecs.BOM_UTF16_BE
codecs.BOM_UTF16_LE
codecs.BOM_UTF32
codecs.BOM_UTF32_BE
codecs.BOM_UTF32_LE

这些常量定义了各种字节序列，即多个编码的Unicode字节顺序标记（BOM）。它们用于UTF-16和UTF-32数据流以指示所使用的字节顺序，并以UTF-8作为Unicode签名。BOM_UTF16是 BOM_UTF16_BE或者BOM_UTF16_LE取决于平台的本地字节顺序，BOM是for BOM_UTF16， BOM_LEfor BOM_UTF16_LE和BOM_BEfor 的别名BOM_UTF16_BE。其他代表UTF-8和UTF-32编码的BOM。

7.2.1。编解码基础类

该codecs模块定义了一组基类，它们定义了用于使用编解码器对象的接口，并且还可以用作自定义编解码器实现的基础。

每个编解码器必须定义四个接口，以使其可用作Python中的编解码器：无状态编码器，无状态解码器，流读取器和流编写器。流读取器和写入器通常重新使用无状态编码器/解码器来实现文件协议。编解码器作者还需要定义编解码器将如何处理编码和解码错误。

7.2.1.1。错误处理程序

为了简化和标准化错误处理，编解码器可以通过接受*错误*字符串参数来实现不同的错误处理方案。以下字符串值由所有标准Python编解码器定义和实现：

值	含义
'strict'	提升UnicodeError（或子类）；这是默认值。在...中实施strict_errors()。
'ignore'	忽略格式错误的数据，并继续不另行通知。在...中实施ignore_errors()。

以下错误处理程序仅适用于 文本编码：

值	含义
'replace'	用适当的替换标记；Python将使用其标准内置的编解码器，'?'编码。在...中实施 replace_errors()。
'xmlcharrefreplace'	替换为适当的XML字符引用（仅用于编码）。在...中实施xmlcharrefreplace_errors()。
'backslashreplace'	用反斜杠转义序列替换。在...中实施backslashreplace_errors()。

值	含义
'namereplace'	用换\N{...}码序列替换（仅用于编码）。在...中实施namereplace_errors()。
'surrogateescape'	在解码时，将字节替换为范围从U+DC80至的 单个代理代码U+DCFF。'surrogateescape'在编码数据时使用错误处理程序时，此代码将被重新转换为相同的字节 。（看到更多PEP 383。

另外，下面的错误处理程序是特定于给定的编解码器的：

值	编解码器	含义
'surrogatepass'	utf-8, utf-16, utf-32, utf-16-be, utf-16-le, utf-32-be, utf-32-le	允许代理的编码和解码。这些编解码器通

*新的3.1版：*在'surrogateescape'和'surrogatepass'错误处理程序。

*改变在3.4版本：*该'surrogatepass'错误处理程序现在可以使用UTF-16 *和UTF-32 *编解码器。

*新的3.5版：*该'namereplace'错误处理程序。

*改变在3.5版本：*该'backslashreplace'错误处理程序现在可以与解码和翻译。

允许值的集合可以通过注册一个新的命名错误处理程序来扩展：

codecs.register_error（*name*，*error_handler*）

在名称*名称*下注册错误处理函数*error_handler*。当*名称*被指定为错误参数时，*error_handler*参数将在编码和解码期间被调用以防错误。

对于编码，*error_handler*将被一个UnicodeEncodeError 实例调用，该实例包含关于错误位置的信息。错误处理程序必须提出这个或另一个异常，或返回一个元组，替换输入的不可解码部分和编码应该继续的位置。替换可以是str或者 bytes。如果替换为字节，编码器将简单地将它们复制到输出缓冲区中。如果替换为字符串，则编码器将对替换进行编码。编码继续在指定位置的原始输入。负位置值将被视为相对于输入字符串的末尾。如果由此产生的位置超出限制，IndexError则会提高。

解码和翻译的工作方式相似，除了UnicodeDecodeError或 UnicodeTranslateError将传递给处理程序，并且将错误处理程序的替换直接放入输出中。

以前注册的错误处理程序（包括标准错误处理程序）可以按名称查找：

codecs.lookup_error（*名字*）

返回之前在名称*名称*下注册的错误处理程序。

引发LookupError无法找到处理程序的情况。

以下标准错误处理程序也可用作模块级功能：

codecs.strict_errors（*例外*）

实现'strict'错误处理：每个编码或解码错误引发一个错误UnicodeError。

codecs.replace_errors（*例外*）

实现'replace'错误处理（仅适用于文本编码）：替代'?'编码错误（由编解码器编码）和'\ufffd'（Unicode替换字符）解码错误。

codecs.ignore_errors(*例外*)
实现'ignore'错误处理：忽略格式错误的数据，继续编码或解码，恕不另行通知。

codecs.xmlcharrefreplace_errors(*例外*)
实现'xmlcharrefreplace'错误处理（仅适用于使用 文本编码进行编码）：不可编码的字符被适当的XML字符引用所替代。

codecs.backslashreplace_errors(*例外*)
实现'backslashreplace'错误处理（仅适用于 文本编码）：格式错误的数据被替换为反斜杠转义序列。

codecs.namereplace_errors(*例外*)
实现'namereplace'错误处理（仅用于使用 文本编码进行编码）：不可编码的字符被\N{...}转义序列替换。

3.5版本中的新功能。

7.2.1.2。无状态编码和解码

基Codec类定义了这些方法，它们还定义了无状态编码器和解码器的功能接口：

Codec.encode(*输入*[,*错误*])
编码对象*输入*并返回一个元组（输出对象，消耗的长度）。例如，文本编码使用特定的字符集编码（例如，cp1252或iso-8859-1）将字符串对象转换为字节对象。

该*错误*参数定义错误处理申请。它默认'strict'处理。

该方法可能不会在Codec实例中存储状态。使用 StreamWriter了其必须保持状态，以便使编码效率的编解码器。

在这种情况下，编码器必须能够处理零长度输入并返回输出对象类型的空对象。

Codec.decode(*输入*[,*错误*])
解码对象*输入*并返回一个元组（输出对象，消耗的长度）。例如，对于文本编码，解码将使用特定字符集编码编码的字节对象转换为字符串对象。

对于文本编码和字节到字节的编解码器，*输入*必须是一个字节对象或一个提供只读缓冲区接口的对象，例如缓冲区对象和内存映射文件。

该*错误*参数定义错误处理申请。它默认'strict'处理。

该方法可能不会在Codec实例中存储状态。使用 StreamReader了其必须保持状态，以便使解码效率的编解码器。

在这种情况下，解码器必须能够处理零长度输入并返回输出对象类型的空对象。

7.2.1.3。增量编码和解码

的IncrementalEncoder和IncrementalDecoder类提供了增量编码和解码的基本接口。编码/解码输入不与一个呼叫到无状态编码器/解码器功能进行，但用多次调用 encode()/ decode()增量编码器/解码器的方法。增量编码器/解码器在方法调用期间跟踪编码/解码过程。

对encode()/ decode()方法调用的连接输出与将 所有单个输入合并成一个输入相同，并且使用无状态编码器/解码器对此输入进行编码/解码。

7.2.1.3.1。IncrementalEncoder对象

的IncrementalEncoder类是用于在多个步骤中编码的输入。它定义了每个增量编码器必须定义的以下方法，以便与Python编解码器注册表兼容。

class codecs. IncrementalEncoder (*errors ='strict'*)

一个IncrementalEncoder实例的构造函数。

所有增量编码器都必须提供此构造器接口。他们可以自由添加额外的关键字参数，但只有在这里定义的关键字参数被Python编解码器注册表使用。

该IncrementalEncoder可以通过提供实现不同的错误处理方案*的错误*关键字参数。请参阅错误处理程序了解可能的值。

该*错误*参数将被分配到相同名称的属性。赋予该属性可以在IncrementalEncoder 对象的生命周期内在不同的错误处理策略之间切换。

encode (*object* [, *final*])

编码*对象*（考虑编码器的当前状态）并返回结果编码对象。如果这是最后一次调用*final*（必须为true）（默认值为false）。 encode()

reset ( )

将编码器重置为初始状态。输出被丢弃：调用 ，必要时传递一个空字节或文本字符串，以重置编码器并获取输出。.encode(object, final=True)

getstate ( )

返回编码器的当前状态，该状态必须是整数。实施应确保这0是最常见的状态。（比整数更复杂的状态可以通过编组/整理状态并将结果字符串的字节编码为整数来转换为整数）。

setstate (*状态*)

将编码器的状态设置为*状态*。*状态*必须是由返回的编码器状态getstate()。

7.2.1.3.2。IncrementalDecoder对象

的IncrementalDecoder类是用于在多个步骤中进行解码的输入。它定义了每个增量解码器必须定义的以下方法，以便与Python编解码器注册表兼容。

class codecs. IncrementalDecoder (*errors ='strict'*)

一个IncrementalDecoder实例的构造函数。

所有增量解码器都必须提供此构造器接口。他们可以自由添加额外的关键字参数，但只有在这里定义的关键字参数被Python编解码器注册表使用。

该IncrementalDecoder可以通过提供实现不同的错误处理方案*的错误*关键字参数。请参阅错误处理程序了解可能的值。

该*错误*参数将被分配到相同名称的属性。赋予该属性可以在IncrementalDecoder 对象的生命周期内在不同的错误处理策略之间切换。

decode（*object*[，*final*]）

解码*对象*（考虑解码器的当前状态）并返回结果解码对象。如果这是最后一次调用*final*（必须为true）（默认值为false）。如果*final*为true，则解码器必须完全解码输入，并且必须清空所有缓冲区。如果这是不可能的（例如由于输入末尾的字节序列不完整），它必须启动错误处理，就像在无状态的情况下一样（这可能会引发异常）。decode()

reset（）

将解码器重置为初始状态。

getstate（）

返回解码器的当前状态。这必须是一个包含两项的元组，第一个必须是包含尚未解码输入的缓冲区。第二个必须是一个整数，可以是附加的状态信息。（该实现应该确保这0是最常见的附加状态信息。）如果这个附加状态信息是0必须的，则必须可以将解码器设置为没有输入缓冲0的状态并且 作为附加状态信息，以便馈送先前缓冲输入到解码器将其返回到先前的状态而不产生任何输出。（通过编组/整理信息并将结果字符串的字节编码为整数，可以将比整数更复杂的其他状态信息转换为整数。）

setstate（*状态*）

将编码器的状态设置为*状态*。*状态*必须是由返回的解码器状态getstate()。

7.2.1.4。流编码和解码

在StreamWriter和StreamReader类提供可用于非常容易地实现新的编码子模块的通用的工作界面。请参阅有关encodings.utf_8如何完成的示例。

7.2.1.4.1。StreamWriter对象

的StreamWriter类是的一个子类Codec，并且限定其中每个流作家必须按顺序定义以下方法为与编解码器的Python注册表兼容。

class codecs.StreamWriter（*stream*，*errors ='strict'*）

构造函数的一个StreamWriter实例。

所有流写入器都必须提供此构造器接口。他们可以自由添加额外的关键字参数，但只有在这里定义的关键字参数被Python编解码器注册表使用。

该*流*参数必须是一个类似文件的对象打开用于写入文本或二进制数据，以适合特定的编解码器。

该StreamWriter可以通过提供实现不同的错误处理方案*的错误*关键字参数。请参阅错误处理程序以了解基础流编解码器可能支持的标准错误处理程序。

该*错误*参数将被分配到相同名称的属性。赋予该属性可以在StreamWriter对象的生命周期内在不同的错误处理策略之间切换。

write (*object*)
将编码对象的内容写入流中。

writelines (*列表*)
将连接的字符串列表写入流中（可能通过重用该write()方法）。标准的字节到字节编解码器不支持这种方法。

reset ( )
刷新并重置用于保持状态的编解码器缓冲区。

调用此方法应确保将输出中的数据置于干净状态，以允许附加新的新数据，而无需重新扫描整个流以恢复状态。

除了上述方法之外，StreamWriter还必须从基础流继承所有其他方法和属性。

7.2.1.4.2。StreamReader对象

的StreamReader类是的一个子类Codec，并且限定其中每个流读取器必须按顺序定义以下方法为与编解码器的Python注册表兼容。

class codecs. StreamReader (*stream* , *errors ='strict'*)
构造函数的一个StreamReader实例。

所有流读取器都必须提供此构造器接口。他们可以自由添加额外的关键字参数，但只有在这里定义的关键字参数被Python编解码器注册表使用。

的*流*参数必须是一个类文件对象开放阅读文本或二进制数据，以适合特定的编解码器。

该StreamReader可以通过提供实现不同的错误处理方案*的错误*关键字参数。请参阅错误处理程序以了解基础流编解码器可能支持的标准错误处理程序。

该*错误*参数将被分配到相同名称的属性。赋予该属性可以在StreamReader对象的生命周期内在不同的错误处理策略之间切换。

*错误*参数的允许值的集合可以扩展 register_error()。

read ([*size* [, *chars* [, *firstline*]]])
解码流中的数据并返回结果对象。

该*字符*参数指示解码码点或字节返回的数量。该read()方法永远不会返回比请求更多的数据，但如果没有足够的可用空间，它可能会返回更少的数据。

所述*尺寸*参数指示来读取用于解码编码的字节或码点的近似最大数量。解码器可以根据需要修改此设置。默认值-1表示尽可能多地读取和解码。该参数旨在防止在一个步骤中解码大文件。

该*FIRSTLINE*标志表示，这将是足以只返回第一线，如果有对后世行解码错误。

该方法应该使用贪婪读取策略，这意味着它应该读取尽可能多的数据，比如在编码的定义和给定大小内所允许的数据，例如，如果可选的编码结束或状态标记在流上可用，则这些也应该被读取。

`readline ( [ size [ , keepends ] ] )`

从输入流中读取一行并返回解码后的数据。

大小，如果给定，作为大小参数传递给流的 `read()` 方法。

如果*keepends*是错误的行结束将从返回的行中被删除。

`readlines ( [ sizehint [ , keepends ] ] )`

读取输入流中可用的所有行，并将它们作为行列表返回。

行尾是使用编解码器的解码器方法实现的，并且如果*keepends*为真，则包含在列表条目中。

sizehint（如果给定）作为*size*参数传递给流的 `read()` 方法。

`reset ( )`

重置用于保持状态的编解码器缓冲区。

请注意，不应该发生流重定位。这种方法主要是为了能够从解码错误中恢复。

除了上述方法之外，`StreamReader`还必须从基础流继承所有其他方法和属性。

7.2.1.4.3。StreamReaderWriter对象

这`StreamReaderWriter`是一个便利的类，它允许包装可在读取和写入模式下工作的流。

设计是这样的，可以使用函数返回的工厂 `lookup()` 函数来构造实例。

class codecs. `StreamReaderWriter ( stream , Reader , Writer , errors ='strict' )`

创建一个`StreamReaderWriter`实例。*流*必须是文件类对象。*读者*和*作家*必须是工厂的功能或提供的类 `StreamReader`和`StreamWriter`接口RESP。错误处理以与流读取器和写入器定义相同的方式完成。

`StreamReaderWriter`实例定义`StreamReader`和`StreamWriter`类的组合接口 。它们从基础流继承所有其他方法和属性。

7.2.1.4.4。StreamRecoder对象

的`StreamRecoder`从一个编码到另一个，不同的编码处理的环境时，其有时是有用的转换数据。

设计是这样的，可以使用函数返回的工厂 lookup() 函数来构造实例。

class codecs. StreamRecoder（ *流*，*编码*，*解码*，*Reader*，*Writer*，*错误='strict'* ）

创建StreamRecoder一个实现双向转换的实例： *编码*和*解码*工作，对前端-可见的代码调用数据read()和write()，而*读者*和*作家*在后台工作-数据*流*。

您可以使用这些对象从例如Latin-1到UTF-8进行透明转码并返回。

该*流*参数必须是一个类似文件的对象。

该*编码*和*解码*参数必须坚持Codec接口。 *Reader*和 *Writer*必须是工厂函数或分别提供StreamReader和StreamWriter接口对象的类 。

错误处理以与流读取器和写入器定义相同的方式完成。

StreamRecoder实例定义StreamReader和StreamWriter类的组合接口 。它们从基础流继承所有其他方法和属性。

7.2.2。编码和

字符串在内部存储为范围内的代码点序列0x0- 0x10FFFF。（看到**PEP 393**关于实现的更多细节。）一旦在CPU和内存之外使用字符串对象，字节顺序以及如何将这些数组作为字节存储成为问题。与其他编解码器一样，将字符串串行化为字节序列称为*编码*，并且从字节序列重新创建字符串称为*解码*。有各种不同的文本序列化编解码器，它们被称为文本编码。

最简单的文本编码（称为'latin-1'或'iso-8859-1'）将代码点0-255映射到字节0x0- 0xff，这意味着包含以上代码点的字符串对象U+00FF不能用此编解码器编码。这样做会引起一个UnicodeEncodeError 看起来像以下（尽管错误消息的细节可能有所不同）：。

```
UnicodeEncodeError: 'latin-1' codec can't encode character '\u1234' in position 3:
ordinal not in range(256)
```

还有另一组选择所有Unicode码点以及如何将这些代码点的不同子集映射到字节编码（所谓的字符表编码）的0x0- 0xff。要查看这是如何完成的，只需打开eg encodings/cp1252.py（这是一种主要用于Windows的编码）。有一个256个字符的字符串常量，显示哪个字符映射到哪个字节值。

所有这些编码只能编码Unicode中定义的1114112个码点中的256个。可以存储每个Unicode代码点的简单而直接的方法是将每个代码点存储为四个连续的字节。有两种可能性：以big endian或little endian顺序存储字节。这两种编码被称为UTF-32-BE和UTF-32-LE分别。他们的缺点是，如果你使用UTF-32-BE一个小端机，你将不得不在编码和解码时交换字节。UTF-32避免了这个问题：字节总是以自然字节顺序排列。当这些字节被具有不同字节顺序的CPU读取时，字节必须交换。为了能够检测a UTF-16或者的字节顺序UTF-32字节序列，就是所谓的BOM（“字节顺序标记”）。这是Unicode字符 U+FEFF。这个字符可以附加在每一个UTF-16或UTF-32 字节序列。该字符（0xFFFE）的字节交换版本是非法字符，可能不会出现在Unicode文本中。所以当一个UTF-16或一个UTF-32字节序列中的第一个字符看来是一个U+FFFE字节必须在解码时交换。不幸的是，角色U+FEFF有第二个目的：一个没有宽度并且不允许单词被分割的字符。它可以例如用于给连字算法提供提示。使用Unicode 4.0 作为一个已被弃用（与（ZERO WIDTH NO-BREAK SPACE U+FEFF ZERO WIDTH NO-BREAK SPACE U+2060 WORD JOINER）承担这个角色）。尽管如此，Unicode软件仍然必须能够处理U+FEFF这两种角色：作为BOM，它是确定编码字节的存储布局

的设备，并且一旦字节序列被解码为字符串就消失; 因为它是一个正常的字符，将被解码为任何其他字符。 ZERO WIDTH NO-BREAK SPACE

还有另一种编码能够编码全部的Unicode字符：UTF-8。UTF-8是一个8位编码，这意味着在UTF-8中没有字节顺序的问题。UTF-8字节序列中的每个字节由两部分组成：标记位（最高有效位）和有效负载位。标记位是一个0到4 1位的序列，后跟一个0位。Unicode字符是这样编码的（x是有效载荷位，当连接时给出Unicode字符）：

范围	编码
U-00000000 ... U-0000007F	0xxxxxxx
U-00000080 ... U-000007FF	110xxxxx 10xxxxxx
U-00000800 ... U-0000FFFF	1110xxxx 10xxxxxx 10xxxxxx
U-00010000 ... U-0010FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

Unicode字符的最低有效位是最右边的x位。

由于UTF-8是8位编码，因此不需要BOM U+FEFF，并且解码字符串中的任何字符（即使它是第一个字符）都被视为a 。 ZERO WIDTH NO-BREAK SPACE

没有外部信息，就不可能可靠地确定使用哪种编码来编码字符串。每个charmap编码可以解码任何随机字节序列。但UTF-8无法实现，因为UTF-8字节序列具有不允许任意字节序列的结构。为了提高可检测到UTF-8编码的可靠性，Microsoft ″utf-8-sig″为其记事本程序发明了一种UTF-8变体（Python 2.5调用 ）：在将任何Unicode字符写入文件之前，UTF-8编码BOM（它看起来像这样作为字节序列：0xef，0xbb，0xbf）被写入。因为任何charmap编码文件都以这些字节值开头（例如映射到）是不可能的

拉丁文小写字母I与DIAERESIS
正确的双角度引号
反向问题标记

在iso-8859-1中），这增加了utf-8-sig从字节序列正确猜测编码的可能性。所以这里的BOM并不能用于确定用于生成字节序列的字节顺序，而是用作帮助猜测编码的签名。于编码的UTF-8-SIG的编解码器将写0xef，0xbb，0xbf如前三个字节到该文件。解码utf-8-sig时，如果它们显示为文件中的前三个字节，则会跳过这三个字节。在UTF-8中，不鼓励使用BOM，通常应避免使用。

7.2.3。标准编码

Python带有一些内置的编解码器，可以作为C函数或字典作为映射表来实现。下表按名称列出了编解码器，以及一些常用别名，以及可能使用编码的语言。别名列表和语言列表都不是详尽无遗的。请注意，仅在大小写不同的情况下使用拼写替代方法或使用连字符而不是下划线也是有效的别名; 因此，例如’utf-8’，对于’utf_8’编解码器而言是有效的别名。

CPython实现细节：一些常见的编码可以绕过编解码器查找机制来提高性能。这些优化机会仅由CPython针对有限的一组（不区分大小写）别名来识别：utf-8，utf8，latin-1，latin1，iso-8859-1，iso8859-1，mbcs（仅限Windows），ascii，us -ascii，utf-16，utf16，utf-32，utf32，并且使用下划线而不是破折号。对这些编码使用替代别名可能会导致执行速度变慢。

*在版本3.6中更改：*为us-ascii识别的优化机会。

许多字符集都支持相同的语言。它们在个别字符（例如，欧元符号是否被支持）以及在将字符分配给代码位置上有所不同。对于欧洲语言尤其如此，通常存在以下变体：

- ISO 8859代码集
- 一个Microsoft Windows代码页，通常从8859代码集派生而来，但用其他图形字符替换控制字符
- 一个IBM EBCDIC代码页
- IBM PC代码页，与ASCII兼容

编解码器	别名	语言
ASCII	646，us-ascii	英语
big5	big5-tw，csbig5	繁体中文
big5hkscs	big5-hkscs，hkscs	繁体中文
cp037	IBM037，IBM039	英语
cp273	273，IBM273，csIBM273	德语 *3.4版新增功能*
cp424	EBCDIC-CP-HE，IBM424	希伯来语
CP437	437，IBM437	英语
CP500	EBCDIC-CP-BE，EBCDIC-CP-CH，IBM500	西欧
CP720		阿拉伯
cp737		希腊语
CP775	IBM775	波罗的语言
CP850	850，IBM850	西欧
CP852	852，IBM852	中欧和东欧
cp855	855，IBM855	保加利亚语，白俄罗斯语，马其顿语，俄语，塞尔维亚语
cp856		希伯来语
cp857	857，IBM857	土耳其
cp858	858，IBM858	西欧
CP860	860，IBM860	葡萄牙语
cp861	861，CP-IS，IBM861	冰岛的
cp862	862，IBM862	希伯来语
cp863	863，IBM863	加拿大
cp864	IBM864	阿拉伯
cp865	865，IBM865	丹麦语，挪威语
CP866	866，IBM866	俄语
cp869	869，CP-GR，IBM869	希腊语
cp874		泰国
cp875		希腊语
CP932	932，ms932，mskanji，ms-kanji	日本

编解码器	别名	语言
CP949	949，ms949，uhc	朝鲜的
CP950	950，ms950	繁体中文
cp1006		乌尔都语
cp1026	ibm1026	土耳其
cp1125	1125，ibm1125，cp866u，ruscii	乌克兰 *3.4版新增功能*
cp1140	ibm1140	西欧
CP1250	窗户-1250	中欧和东欧
CP1251	窗户-1251	保加利亚语，白俄罗斯语，马其顿语，俄语，塞尔维亚语
CP1252	窗口1252	西欧
cp1253	窗户-1253	希腊语
cp1254	窗户-1254	土耳其
cp1255	窗户-1255	希伯来语
cp1256	窗户-1256	阿拉伯
cp1257	窗户-1257	波罗的语言
cp1258	窗户-1258	越南
cp65001		仅Windows：Windows UTF-8（CP_UTF8） *3.3版本的新功能*
EUC_JP	eucjp，ujis，u-jis	日本
euc_jis_2004	jisx0213，eucjis2004	日本
euc_jisx0213	eucjisx0213	日本
EUC_KR	euckr，korean，rksc5601，ks_c-5601-1987，ks_c-5601，ks_c_5601-1987	朝鲜的
GB2312	chinese，csiso58gb231280，gb2312-80，gb231280，gb_2312-80，iso-ir-58	简体中文
GBK	936，cp936，ms936	统一的中文
GB18030	GB18030-2000	统一的中文
赫兹	hzgb，hz-gb，hz-gb-2312	简体中文
iso2022_jp	csiso2022jp，iso2022jp，iso-2022-jp	日本
iso2022_jp_1	iso2022jp-1，iso-2022-jp-1	日本
iso2022_jp_2	iso2022jp-2，iso-2022-jp-2	日本语，韩语，简体中文，西欧，希腊语
iso2022_jp_2004	iso2022jp-2004，iso-2022-jp-2004	日本
iso2022_jp_3	iso2022jp-3，iso-2022-jp-3	日本
iso2022_jp_ext	iso2022jp-ext，iso-2022-jp-ext	日本
iso2022_kr	csiso2022kr，iso2022kr，iso-2022-kr	朝鲜的
LATIN_1	iso8859-1，iso-8859-1，iso8859，iso-8859，latin，latin1，L1	西欧
iso8859_2	iso-8859-2，latin2，L2	中欧和东欧

编解码器	别名	语言
iso8859_3	iso-8859-3，latin3，L3	世界语，马耳他
iso8859_4	iso-8859-4，latin4，L4	波罗的语言
iso8859_5	iso-8859-5，西里尔文	保加利亚语塞尔维亚语，马其顿
iso8859_6	iso-8859-6，阿拉伯语	阿拉伯
iso8859_7	iso-8859-7，希腊语，希腊语8	希腊语
iso8859_8	iso-8859-8，希伯来语	希伯来语
iso8859_9	iso-8859-9，latin5，L5	土耳其
iso8859_10	iso-8859-10，latin6，L6	北欧语言
iso8859_11	iso-8859-11，泰语	泰语
iso8859_13	iso-8859-13，latin7，L7	波罗的语言
iso8859_14	iso-8859-14，latin8，L8	凯尔特语言
iso8859_15	iso-8859-15，latin9，L9	西欧
iso8859_16	iso-8859-16，latin10，L10	东南欧
裘哈	cp1361，ms1361	朝鲜的
koi8_r		俄语
koi8_t		塔吉克 *3.5版本中的新功能。*
koi8_u		乌克兰
kz1048	kz_1048，strk1048_2002，rk1048	哈萨克人 *3.5版本中的新功能。*
mac_cyrillic	maccyrillic	保加利亚语塞尔维亚语，马其顿
mac_greek	macgreek	希腊语
mac_iceland	maciceland	冰岛的
mac_latin2	maclatin2，maccentraleurope	中欧和东欧
mac_roman	宏观人，麦金托什	西欧
mac_turkish	macturkish	土耳其
ptcp154	csptcp154，pt154，cp154，西里尔文的亚洲人	哈萨克人
shift_jis访问	csshiftjis，shiftjis，sjis，s_jis	日本
shift_jis_2004	shiftjis2004，sjis_2004，sjis2004	日本
shift_jisx0213	shiftjisx0213，sjisx0213，	日本
utf_32	U32，utf32	所有语言
utf_32_be	UTF-32BE	所有语言
utf_32_le	UTF-32LE	所有语言
utf_16	U16，utf16	所有语言
utf_16_be	UTF-16BE	所有语言
utf_16_le	UTF-16LE	所有语言
utf_7	U7，unicode-1-1-utf-7	所有语言

编解码器	别名	语言
UTF_8	U8，UTF，utf8	所有语言
utf_8_sig		所有语言

版本3.4中更改： utf-16 *和utf-32 *编码器不再允许代理码点（U+D800- U+DFFF）进行编码。utf-32 *解码器不再解码对应于替代码点的字节序列。

7.2.4。Python特定的编码

一些预定义的编解码器是Python特有的，因此它们的编解码器名称在Python之外没有任何意义。根据预期的输入和输出类型，这些在下表中列出（请注意，虽然文本编码是编解码器最常见的用例，但底层编解码器基础结构支持任意数据转换，而不仅仅支持文本编码）。对于不对称编解码器，所述目的描述编码方向。

7.2.4.1。文本编码

以下编解码器提供str到bytes编码和 类字节对象到str解码，类似于Unicode文本编码。

编解码器	别名	目的
IDNA		器物 **RFC 3490**，另请参阅 encodings.idna。仅errors='strict'支持。
MBCS	ansi，dbcs	仅Windows：根据ANSI代码页（CP_ACP）对操作数进行编码
OEM		仅限Windows：根据OEM代码页（CP_OEMCP）对操作数进行编码 *3.6版本中的新功能。*
的PalmOS		PalmOS 3.5的编码
Punycode码		器物 **RFC 3492**。不支持状态编解码器。
raw_unicode_escape		Latin-1编码与 \uXXXX和 \UXXXXXXXX其他代码点它现有的反斜杠不会以这种方式逃避应用程序的参与。
未定义		错误处理程序的参与，甚至是空字符串。
unicode_escape		编解码器，请注意，不会转义
unicode_internal		返回操作数的内部表示。不支持状态编解码器。 *自从版本3.3开始弃用：*此表示已被废弃 **PEP 393**。

7.2.4.2。二进制变换

以下编解码器提供二进制转换：类似字节的对象 到bytes映射。它们不受bytes.decode()（仅产生str输出）的支持。

编解码器	别名	目的	编码器/解码器

编解码器	别名	目的	编码器/解码器
base64_codec [1]	base64, base_64	将操作数转换为多行MIME base64（结果始终包含尾部'\n'） *在版本3.4中进行了更改：*接受任何 类似字节的对象 作为编码和解码的输入	base64.encodebytes() / base64.decodebytes()
bz2_codec	BZ2	使用bz2压缩操作数	bz2.compress() / bz2.decompress()
hex_codec	十六进制	将操作数转换为带有两个十六进制数字的每个字节	binascii.b2a_hex() / binascii.a2b_hex()
quopri_codec	quotedprintable, quoted_printable	使用可打印的ASCII字符将操作数转换为可引用	quopri.encode() 带 quotetabs=True/ quopri.decode()
uu_codec	UU	使用uuencode转换操作数	uu.encode() / uu.decode()
zlib_codec	zip，zlib	使用gzip压缩操作数	zlib.compress() / zlib.decompress()

[1] 除类字节对象外，'base64_codec'还接受str用于解码的纯ASCII实例

*3.2版本中的新功能：*恢复二进制转换。

*在版本3.4中更改：*恢复二进制转换的别名。

7.2.4.3。文本转换

下面的编解码器提供了一个文本转换：一str来str 映射。它不被str.encode()（仅产生 bytes 输出）支持。

编解码器	别名	目的
rot_13	ROT13	返回操作数的凯撒密码加

*3.2版新增功能：*恢复rot_13文本转换。

*在版本3.4中更改：*恢复rot13别名。

7.2.5。encodings.idna- 应用程序中的国际化域名

这个模块实现 **RFC 3490**（应用程序中的国际化域名）和**RFC 3492**（Nameprep：国际化域名（IDN）的Stringprep配置文件）。它建立在punycode编码和stringprep。

这些RFC一起定义了一个协议来支持域名中的非ASCII字符。包含非ASCII字符（例如 www.Alliancefrançaise.nu ）的域名将 被转换为与ASCII兼容的编码（如，ACE www.xn--alliancefranaise-npb.nu ）。然后，域名的ACE形式将用于协议不允许任意字符的所有地方，例如DNS查询，HTTP *主机*字段等。该转换在应用程序中执行; 如果可能的话对用户不可见：应

用程序应该在线上将Unicode域标签透明地转换为IDNA，并在将ACE标签呈现给用户之前将它们转换回Unicode。

Python以多种方式支持这种转换：idna编解码器在Unicode和ACE之间执行转换，根据3.1（1）节中定义的分隔符将输入字符串分隔为标签**RFC 3490，**并根据需要将每个标签转换为ACE，反之将输入字节字符串.分隔为基于分隔符的标签，并将所有ACE标签转换为unicode。此外，socket模块将Unicode主机名透明地转换为ACE，以便应用程序在将主机名传递给套接字模块时不必担心自己将其转换为主机名。最重要的是，具有主机名作为函数参数的模块（例如http.client和ftplib）接受Unicode主机名（http.client然后在 *Host*字段中透明地发送IDNA主机名（ 如果它根本发送该字段的话）。

从线路接收主机名时（例如在反向名称查找中），不会自动转换为Unicode：希望向用户显示此类主机名的应用程序应将它们解码为Unicode。

该模块encodings.idna还实现了nameprep过程，该过程对主机名执行一定的规范化，实现国际域名不区分大小写，统一类似字符。如果需要，可以直接使用nameprep函数。

encodings.idna.**nameprep**（*标签*）
　返回*标签*的namepreppedd版本。该实现目前假定查询字符串，这AllowUnassigned是事实。

encodings.idna.**ToASCII**（*标签*）
　按照指定将标签转换为ASCII **RFC 3490**。UseSTD3ASCIIRules被假定为假。

encodings.idna.**ToUnicode**（*标签*）
　按照指定将标签转换为Unicode **RFC 3490**。

7.2.6。encodings.mbcs- Windows ANSI代码页

根据ANSI代码页（CP_ACP）编码操作数。

可用性：仅限Windows。

*版本3.3中更改：*支持任何错误处理程序。

*在版本3.2中更改：*在3.2之前，*错误*参数被忽略;'replace'总是用于编码和'ignore'解码。

7.2.7。encodings.utf_8_sig- 带BOM签名的UTF-8编解码器

该模块实现了UTF-8编解码器的一种变体：在编码时，UTF-8编码的BOM将被添加到UTF-8编码的字节中。对于有状态编码器，这只会执行一次（在第一次写入字节流时）。在数据开始时解码可选的UTF-8编码BOM将被跳过。

8.数据类型

本章介绍的模块提供了各种专用数据类型，如日期和时间，固定类型数组，堆队列，同步队列和集合。

蟒蛇也提供了一些内置的数据类型，特别是 `dict` ， `list` ， `set`和`frozenset`和 `tuple`。在`str`类用于保存Unicode字符串，以及`bytes`类是用于保存二进制数据。

本章介绍以下模块：

8.1。 datetime- 基本日期和时间类型

源代码： Lib / datetime.py

该datetime模块提供简单和复杂的操作日期和时间的类。在支持日期和时间算法的同时，实现的重点是对输出格式和操作进行有效的属性提取。有关相关功能，另请参阅 time和calendar模块。

有两种日期和时间对象：“天真”和“意识”。

知道的对象具有足够的适用算法和政治时间调整的知识，例如时区和夏令时信息，以相对于其他感知对象进行定位。意识对象用于表示一个不能解释的特定时刻[1]。

天真的对象不包含足够的信息来明确地定位其相对于其他日期/时间对象的本身。天真的对象是否代表协调世界时（ UTC ），本地时间或其他时区中的时间，完全取决于程序，就像程序是否代表米，英里或质量一样。天真的对象很容易理解和使用，代价是无视现实的某些方面。

对于需要了解对象的应用程序，datetime以及time 对象有一个可选的时区信息属性tzinfo，可以被设置为抽象的子类的实例tzinfo类。这些tzinfo对象捕获有关UTC时间偏移的信息，时区名称以及夏令时是否生效。请注意，模块只提供一个具体tzinfo类，即timezone类datetime。该timezone类可以表示具有UTC固定偏移的简单时区，例如UTC本身或北美EST和EDT时区。更深层次的细节支持时区则取决于应用程序。全球时间调整的规则比理性更具政治性，经常变化，除了UTC以外，没有适用于所有应用的标准。

该datetime模块导出以下常量：

datetime.MINYEAR
: a date或datetimeobject中允许的最小年份数。 MINYEAR是1。

datetime.MAXYEAR
: a date或datetime对象中允许的最大年份数。 MAXYEAR是9999。

也可以看看:

模 calendar
: 一般日历相关功能。

模 time
: 时间访问和转换。

8.1.1。可用类型

类 datetime.date
: 一个理想化的天真约会，假定当前的公历始终是，而且一直是有效的。属性： year ， month ，和 day。

类 datetime.time
: 一个理想化的时间，独立于任何特定的日子，假设每天有24 * 60 * 60秒（这里没有“闰秒”的概念）。属性： hour ， minute ， second ， microsecond ，和tzinfo。

类 datetime.**datetime**

日期和时间的组合。属性：year ，month ， day ，hour ，minute ，second ，microsecond ，和tzinfo。

类 datetime.**timedelta**

表达两者之间的差异的持续时间date ， time或datetime实例微秒的分辨率。

类 datetime.**tzinfo**

时区信息对象的抽象基类。这些由 课程datetime和time课程用于提供时间调整的可定制概念（例如，考虑时区和/或夏令时）。

类 datetime.**timezone**

一个将tzinfo抽象基类实现为UTC固定偏移量的类。

3.2版本中的新功能

这些类型的对象是不可变的。

date类型的对象总是天真的。

类型的对象time或datetime可能天真或意识。一个datetime对象*d*意识到，如果d.tzinfo不None和d.tzinfo.utcoffset(d) 不返回None。如果d.tzinfo是 None ，或者如果d.tzinfo不是， None但是d.tzinfo.utcoffset(d) 回报None ，*d*是天真的。一个time对象*牛逼*意识到，如果t.tzinfo不None和t.tzinfo.utcoffset(None)不返回 None。否则，*t*是天真的。

天真与感知之间的区别不适用于timedelta 对象。

子类关系：

```
object
    timedelta
    tzinfo
        timezone
    time
    date
        datetime
```

8.1.2。 timedelta对象

甲timedelta对象表示的持续时间，两个日期或时间之间的差。

*类*datetime.**timedelta**(*天数*= *0* , *秒*= *0* , *微秒*= *0* , *毫秒*= *0* , *分钟*= *0* , *小时*= *0* , *星期*= *0*)

所有参数都是可选的，默认为0。参数可能是整数或浮点数，可能是正数或负数。

只有*几天*，*几秒*和*几微秒*存储在内部。参数被转换为这些单位：

- 毫秒转换为1000微秒。
- 一分钟转换为60秒。
- 一小时转换为3600秒。
- 一周转换为7天。

然后对天，秒和微秒进行归一化，以便该表示是唯一的

- 0 <= microseconds < 1000000
- 0 <= seconds < 3600*24 （一天中的秒数）
- -999999999 <= days <= 999999999

如果任何参数是浮点数并且有分数微秒，则将所有参数剩下的小数微秒组合起来，并使用半到平局的破解器将它们的总和四舍五入到最接近的微秒。如果没有参数是浮点数，则转换和规范化过程是精确的（不会丢失任何信息）。

如果标准化天数在指定范围之外， OverflowError则会提高。

请注意，负值的标准化起初可能令人惊讶。例如，

```
>>> from datetime import timedelta
>>> d = timedelta(microseconds=-1)
>>> (d.days, d.seconds, d.microseconds)
(-1, 86399, 999999)
```

类属性是：

timedelta.**min**

最负面的timedelta对象，timedelta(-999999999)。

timedelta.**max**

最积极的timedelta对象，。 timedelta(days=999999999, hours=23, minutes=59, seconds=59, microseconds=999999)

timedelta.**resolution**

不等于timedelta对象 之间可能的最小差异timedelta(microseconds=1)。

请注意，由于规范化， timedelta.max> -timedelta.min 。 -timedelta.max 不能表现为一个timedelta对象。

实例属性（只读）：

属性	值
days	介于-999999999和999999999之间
seconds	介于0和86399之间
microseconds	介于0和999999之间（含）

支持的操作：

手术	结果
t1 = t2 + t3	总和*t2*和*t3*。之后*t1* - *t2* == *t3*和*t1* - *t3* == *t2*为真。（1）
t1 = t2 - t3	*t2*和*t3*的区别。之后*t1* == *t2* - *t3*和*t2* == *t1* + *t3*为真。（1）
t1 = t2 * i or t1 = i * t2	Delta乘以整数。之后*t1* // i == *t2*为真，提供。 i != 0
	一般来说，*t1* * i == *t1* *（i-1）+ *t1* 是真的。（1）
t1 = t2 * f or t1 = f * t2	Delta乘以浮点数，结果四舍五入到最接近的偶数。
f = t2 / t3	*t2*的分部（3）由*t3*。返回一个 float对象。
t1 = t2 / f or t1 = t2 / i	Delta除以浮点数或整数。结果四舍五入到最接近的偶数。
t1 = t2 // i 要么 t1 = t2 // t3	计算底数的余数（如果有的话）被扔掉。在第二种情况下，返回的余数是作为一个timedelta对象计算的。（3）
t1 = t2 % t3	

q, r = divmod(t1, t2)	等算q = t1 // t2r = t1 % t2timedelta, r是一个 对
+t1	返回timedelta具有相同值的对象。（2）
-t1	相当于timedelta（ - *t1.days* ， - *t1.seconds* ， - *t1.microseconds* ）和*t1* * -1。（1）（4）
abs(t)	相当于+ *t的*时候，以及 - *t的*时候。（2）t.days >= 0t.days < 0
str(t)	返回表单中的字符串 ，其中D为负数。（5）[D day[s],][H]H:MM:SS[.UUUUUU]t
repr(t)	datetime.timedelta(D[, S[, U]])t。（5）

笔记：

1. 这是确切的，但可能会溢出。
2. 这是确切的，不能溢出。
3. 由0加分ZeroDivisionError。
4. - *timedelta.max*不能表示为一个timedelta对象。
5. timedelta对象的字符串表示与其内部表示相似。这导致负面timedeltas有点不寻常的结果。例如：

```
>>> timedelta(hours=-5)
datetime.timedelta(-1, 68400)
>>> print(_)
-1 day, 19:00:00
```

除了上面列出的操作之外，timedelta对象还支持使用date和datetime 对象进行某些增加和减少操作（请参见下文）。

*在版本3.2中进行了更改：*现在支持timedelta对另一个timedelta对象进行地板划分和对象 划分，以及余数操作和divmod()函数。现在支持timedelta对float对象进行真分割和乘法运算 。

表示更小时间间隔的timedelta对象支持比较对象 timedelta。为了阻止混合类型的比较回落到按对象地址进行默认比较，当一个timedelta对象与不同类型的对象进行TypeError比较时，除非比较结果为==或，否则将引发该混合类型比较!=。后面的情况分别返回False或 True。

timedelta对象是可散列的（可用作字典键），支持高效的酸洗，并且在布尔上下文中，timedelta当且仅当它不等于时，才认为对象为真timedelta(0)。

实例方法：

timedelta.total_seconds（）

返回持续时间中包含的总秒数。相当于 。td / timedelta(seconds=1)

请注意，对于非常大的时间间隔（在大多数平台上超过270年），此方法将失去微秒精度。

3.2版本中的新功能

用法示例：

```
>>> from datetime import timedelta
>>> year = timedelta(days=365)
>>> another_year = timedelta(weeks=40, days=84, hours=23,
...                          minutes=50, seconds=600)  # adds up to 365 days
>>> year.total_seconds()
31536000.0
>>> year == another_year
True
>>> ten_years = 10 * year
>>> ten_years, ten_years.days // 365
(datetime.timedelta(3650), 10)
>>> nine_years = ten_years - year
>>> nine_years, nine_years.days // 365
(datetime.timedelta(3285), 9)
>>> three_years = nine_years // 3;
>>> three_years, three_years.days // 365
(datetime.timedelta(1095), 3)
>>> abs(three_years - ten_years) == 2 * three_years + year
True
```

8.1.3。date对象

一个date对象代表一个理想化的日历日期（年，月，日），当前的公历无限期延长两个方向。第1年的1月1日被称为第1日，第1年的1月2日被称为第2日，依此类推。这与Dershowitz和Reingold的书Calendrical Calculations中的“预测格里历”日历的定义相匹配，它是所有计算的基本日历。请参阅本书，了解在格雷戈里公会和许多其他日历系统之间进行转换的算法。

*班级*datetime.date（*年*，*月*，*日*）

所有参数都是必需的。参数可以是整数，范围如下：

- MINYEAR <= year <= MAXYEAR
- 1 <= month <= 12
- 1 <= day <= number of days in the given month and year

如果超出这些范围的论点ValueError被提出。

其他构造函数，所有类方法：

classmethod date.today（）

返回当前的本地日期。这相当于 date.fromtimestamp(time.time())。

classmethod date.fromtimestamp（*timestamp*）

返回与POSIX时间戳相对应的本地日期，例如返回的日期time.time()。这可能会提高OverflowError，如果时间戳是出在平台的C支持的值范围的localtime()功能，并OSError在localtime()故障。从1970年到2038年这种情况通常会受到限制。请注意，在非POSIX系统中，在其时间戳概念中包含闰秒时，闰秒会被忽略fromtimestamp()。

*在3.3版本中更改：*提高OverflowError而不是ValueError如果时间戳是出在平台的C支持的值范围的 localtime()功能。提高OSError，而不是 ValueError对localtime()失败。

classmethod date.fromordinal（*ordinal*）

返回对应于格雷高尔顺序的日期，第一年的1月1日有顺序1. ValueError除非是。对于任何日期*d*，。 1 <= ordinal <= date.max.toordinal() date.fromordinal(d.toordinal()) == d

类属性：

date.min
最早的可表示日期。date(MINYEAR, 1, 1)

date.max
最新的可表示日期。date(MAXYEAR, 12, 31)

date.resolution
不相等的日期对象之间可能的最小差异 timedelta(days=1)。

实例属性（只读）：

date.year
之间MINYEAR和MAXYEAR包容性。

date.month
1至12之间。

date.day
在1和给定年的给定月份中的天数之间。

支持的操作：

手术	结果
date2 = date1 + timedelta	*date2*是timedelta.days从*date1*中删除的天数。（1）
date2 = date1 - timedelta	计算*DATE2*这样。（2）date2 + timedelta == date1
timedelta = date1 - date2	（3）
date1 < date2	当*date1*在*date2*之前及时时，*date1*被认为小于*date2*。（4）

笔记：

1. *date2*在时间上向前移动if 或向后移动if 。之后。 并被忽略。 如果小于 或大于，则提高。timedelta.days > 0timedelta.days < 0date2 - date1 == timedelta.daystimedelta.secondstimedelta.microsecondsOverflowErrordate2.yearMINYEARMAX
2. 这与date1 + (-timedelta)不完全等价，因为在date1 - timedelta没有的情况下，-timedelta在隔离时可能会溢出。 timedelta.seconds并被timedelta.microseconds忽略。
3. 这是确切的，不能溢出。timedelta.seconds和timedelta.microseconds是0，而date2 + timedelta == date1之后。
4. 换句话说，当且仅当。为了阻止比较从回落到比较对象地址的默认方案，如果另一个比较不是对象，日期比较通常会引发 。但是，如果其他比较具有属性，则返回 。这个钩子为其他类型的日期对象提供了一个实现混合类型比较的机会。如果没有，则将 对象与不同类型的对象进行比较时，除非比较为或，否则将引发该对象。后面的情况分别返回 或。 date1 < date2date1.toordinal() < date2.toordinal()TypeErrordateNotImplementedtimetuple()dateTypeError==!=FalseTrue

日期可以用作字典键。在布尔上下文中，所有date 对象都被认为是真的。

实例方法：

date.replace(*year = self.year*，*month = self.month*，*day = self.day*)

使用相同的值返回一个日期，除了那些由指定的关键字参数给定新值的参数。例如，如果，那么。d == date(2002, 12, 31)d.replace(day=26) == date(2002, 12, 26)

date.timetuple ( )

返回一个time.struct_time如返回的time.localtime()。小时，分钟和秒为0，DST标志为-1。d.timetuple() 相当于今年1月1 日开始的当天数字在 哪里。 time.struct_time((d.year, d.month, d.day, 0, 0, 0, d.weekday(), yday, -1))yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 11

date.toordinal ( )

返回日期，其中1月1日1年的有序1.对于任何的proleptic阳历序date对象*d*， 。date.fromordinal(d.toordinal()) == d

date.weekday ( )

将星期几作为整数返回，其中星期一为0，星期日为6.例如，星期三。另见 。date(2002, 12, 4).weekday() == 2isoweekday()

date.isoweekday ( )

将星期几作为整数返回，其中星期一为1，星期日为7.例如，星期三。另见 ，。date(2002, 12, 4).isoweekday() == 3weekday()isocalendar()

date.isocalendar ( )

返回3元组（ISO年，ISO周编号，ISO工作日）。

ISO日历是格里高利日历的一个广泛使用的变体。请参阅https://www.staff.science.uu.nl/~gent0113/calendar/isocalendar.htm以获取更好的解释。

ISO年度包括52周或53周，以及周一开始于周一至周日结束。ISO年的第一周是包含一个星期四的第一个（格里高利）历年。这称为第1周，该星期四的ISO年与公历年相同。

例如，2004开始在星期四，因此2004年ISO的第一个星期在周一开始2003年12月29日，周日结束，4 2004年一月，以便 和。date(2003, 12, 29).isocalendar() == (2004, 1, 1)date(2004, 1, 4).isocalendar() == (2004, 1, 7)

date.isoformat ( )

以ISO 8601格式返回表示日期的字符串'YYYY-MM-DD'。例如，。date(2002, 12, 4).isoformat() == ’2002-12-04’

date.__str__ ( )

对于日期*d*而言，str(d)相当于d.isoformat()。

date.ctime ( )

例如，返回表示日期的字符串。相当于 在本地C 函数（调用 但未调用）符合C标准的平台上。date(2002, 12, 4).ctime() == ’Wed Dec 4 00:00:00 2002’d.ctime()time.ctime(time.mktime(d.timetuple()))ctime()time.ctime()date.ctime()

date.strftime (*格式*)

返回一个表示日期的字符串，由一个明确的格式字符串控制。指代小时，分钟或秒的格式代码将会看到0个值。有关格式化指令的完整列表，请参阅strftime ( ) 和strptime ( ) 行为。

date.__format__（*格式*）

和...一样date.strftime()。这使得可以为格式化字符串文本中的date对象和使用时的格式字符串指定格式字符串。有关格式化指令的完整列表，请参阅 strftime（ ）和strptime（ ）行为。str.format()

计算事件日期的示例：

```
>>> import time
>>> from datetime import date
>>> today = date.today()
>>> today
datetime.date(2007, 12, 5)
>>> today == date.fromtimestamp(time.time())
True
>>> my_birthday = date(today.year, 6, 24)
>>> if my_birthday < today:
...     my_birthday = my_birthday.replace(year=today.year + 1)
>>> my_birthday
datetime.date(2008, 6, 24)
>>> time_to_birthday = abs(my_birthday - today)
>>> time_to_birthday.days
202
```

使用示例date：

```
>>> from datetime import date
>>> d = date.fromordinal(730920) # 730920th day after 1. 1. 0001
>>> d
datetime.date(2002, 3, 11)
>>> t = d.timetuple()
>>> for i in t:
...     print(i)
2002                # year
3                   # month
11                  # day
0
0
0
0                   # weekday (0 = Monday)
70                  # 70th day in the year
-1
>>> ic = d.isocalendar()
>>> for i in ic:
...     print(i)
2002                # ISO year
11                  # ISO week number
1                   # ISO day number ( 1 = Monday )
>>> d.isoformat()
'2002-03-11'
>>> d.strftime("%d/%m/%y")
'11/03/02'
>>> d.strftime("%A %d. %B %Y")
'Monday 11. March 2002'
>>> 'The {1} is {0:%d}, the {2} is {0:%B}.'.format(d, "day", "month")
'The day is 11, the month is March.'
```

8.1.4。datetime对象

甲datetime对象是包含来自所有信息的单个对象date的对象和time对象。就像一个date 物体一样，datetime假定当前的格里历日历在两个方向上延伸; 像时间对象一样，datetime假设每天都有3600 * 24秒。

构造函数：

*类*datetime.datetime（*年*，*月*，*日*，*小时= 0*，*分钟= 0*，*秒= 0*，*微秒= 0*，*tzinfo =无*，***，*折叠= 0* ）

年，月和日的参数是必需的。 *tzinfo*可能是None一个tzinfo子类的实例。其余参数可以是整数，范围如下：

- MINYEAR <= year <= MAXYEAR，
- 1 <= month <= 12，
- 1 <= day <= number of days in the given month and year，
- 0 <= hour < 24，
- 0 <= minute < 60，
- 0 <= second < 60，
- 0 <= microsecond < 1000000，
- fold in [0, 1]。

如果超出这些范围的论点ValueError被提出。

*3.6版*新增功能：增加了fold参数。

其他构造函数，所有类方法：

classmethod datetime.today（）

返回当前本地日期时间。这相当于。另见，。tzinfo None datetime.fromtimestamp(time.time()) now() fromtimestamp()

classmethod datetime.now（*tz = None* ）

返回当前的本地日期和时间。如果可选参数*tz*是None 或未指定，则类似于此today()，但如果可能，提供的精度比通过time.time()时间戳记所获得的精度要高（例如，在提供C gettimeofday()函数的平台上可能会这样做）。

如果*tz*不是None，它必须是一个tzinfo子类的实例，并且当前的日期和时间被转换为*tz*的时区。在这种情况下，结果等同于tz.fromutc(datetime.utcnow().replace(tzinfo=tz))。另见today()，utcnow()。

classmethod datetime.utcnow（）

返回当前的UTC日期和时间。这就像是，但返回当前的UTC日期和时间，作为一个天真的对象。知道当前的UTC日期时间可以通过调用获得。另见。tzinfo None now() datetime datetime.now(timezone.utc) now()

classmethod datetime.fromtimestamp（*timestamp*，*tz = None* ）

返回与POSIX时间戳相对应的本地日期和时间，例如返回的时间戳time.time()。如果可选参数*tz*是None或未指定，则时间戳将转换为平台的本地日期和时间，并且返回的datetime对象是幼稚的。

如果*tz*不是None，它必须是一个tzinfo子类的实例，并且时间戳被转换为*tz*的时区。在这种情况下，结果等同于 tz.fromutc(datetime.utcfromtimestamp(timestamp).replace(tzinfo=tz))。

fromtimestamp()可能会提高OverflowError，如果时间戳是从由平台的C支持的值的范围内的localtime()或 gmtime()功能，和OSError上localtime()或 gmtime()失败。在1970年到2038年这种情况通常会受到限制。请注意，在非POSIX系统中，在其时间戳概念中包含闰秒时，闰秒会被忽略fromtimestamp()，然后可能会有两个时间戳相差一秒产生相同的datetime物体。另见utcfromtimestamp()。

*在3.3版本改变：*提高OverflowError，而不是ValueError如果时间戳是从由平台的C支持的值的范围内的 localtime()或gmtime()功能。 提高OSError ，而不是ValueError对localtime()或gmtime() 失败。

*在版本3.6中更改：*fromtimestamp()可能会返回fold设置为1的实例。

classmethod datetime.**utcfromtimestamp**(*timestamp*)

返回与datetimePOSIX时间戳相对应 的UTC 。这可能会提高，如果时间戳是出在平台的C支持的值范围的功能，并在故障。 1970年到2038年这种情况通常会受到限制。 tzinfo NoneOverflowErrorgmtime()OSErrorgmtime()

要获得知晓的datetime对象，请致电fromtimestamp()：

```
datetime.fromtimestamp(timestamp, timezone.utc)
```

在符合POSIX的平台上，它等同于以下表达式：

```
datetime(1970, 1, 1, tzinfo=timezone.utc) + timedelta(seconds=timestamp)
```

除了后者的公式总是支持全年范围：介于MINYEAR和MAXYEAR包含之间 。

*在3.3版本中更改：*提高OverflowError而不是ValueError如果时间戳是出在平台的C支持的值范围的 gmtime()功能。提高OSError，而不是 ValueError对gmtime()失败。

classmethod datetime.**fromordinal**(*ordinal*)

返回datetime相应的格雷戈里顺序，第一年的1月1日有顺序1. ValueError除非。结果的小时，分钟，秒和微秒全部为0，并且是。1 <= ordinal <= datetime.max.toordinal()tzinfoNone

classmethod datetime.**combine**(*日期*, *时间*, *tzinfo = self.tzinfo*)

返回一个新的datetime对象，其日期分量等于给定的date对象，并且其时间分量等于给定的time对象。如果提供了*tzinfo* 参数，则其值用于设置tzinfo结果的tzinfo属性，否则将使用*time*参数的属性。

对于任何datetime对象*d*， 。如果date是一个 对象，它的时间组件和属性将被忽略。 d == datetime.combine(d.date(), d.time(), d.tzinfo)datetimetzinfo

*在版本3.6中更改：*添加了*tzinfo*参数。

classmethod datetime.**strptime**(*date_string*, *format*)

返回一个datetime对应的*date_string*，根据*格式*进行解析 。这相当于。如果date_string和format不能被解析，或者它返回的值不是时间元组，则会引发该错误。有关格式化指令的完整

列表，请参阅 strftime（ ）和strptime（ ）行为。 datetime(*(time.strptime(date_string, format)[0:6]))ValueErrortime.strptime()

类属性：

datetime.**min**

最早的代表datetime，。 datetime(MINYEAR, 1, 1, tzinfo=None)

datetime.**max**

最新的代表datetime，。 datetime(MAXYEAR, 12, 31, 23, 59, 59, 999999, tzinfo=None)

datetime.**resolution**

不等于datetime对象 之间可能的最小差异timedelta(microseconds=1)。

实例属性（只读）：

datetime.**year**

之间MINYEAR和MAXYEAR包容性。

datetime.**month**

1至12之间。

datetime.**day**

在1和给定年的给定月份中的天数之间。

datetime.**hour**

在range(24)。

datetime.**minute**

在range(60)。

datetime.**second**

在range(60)。

datetime.**microsecond**

在range(1000000)。

datetime.**tzinfo**

该对象作为*tzinfo*参数传递给datetime构造函数，或者None如果没有传递。

datetime.**fold**

在。用于在重复的时间间隔内消除墙壁时间的歧义。（在夏令时结束时回滚时钟或由于政治原因导致当前区域的UTC偏移量减少时发生重复的时间间隔。）值0（1）表示两个时刻中较早（较晚）的时间相同的墙壁时间表示。[0, 1]

3.6版本中的新功能。

支持的操作：

手术	结果
datetime2 = datetime1 + timedelta	(1)
datetime2 = datetime1 - timedelta	(2)

timedelta = datetime1 - datetime2	（3）
datetime1 < datetime2	比较datetime到 datetime。（4）

1. datetime2是从datetime1中删除的timedelta的持续时间，如果timedelta.days> 0 则向前移动，如果timedelta.days<0则向后移动。结果tzinfo与输入日期时间具有相同的属性，而datetime2 - datetime1 == timedelta之后。OverflowError如果datetime2.year将小于MINYEAR或大于， 则会提高MAXYEAR。请注意，即使输入是感知对象，也不会进行时区调整。

2. 计算datetime2，使datetime2 + timedelta == datetime1。至于添加，结果具有与tzinfo输入日期时间相同的属性，并且即使输入已知，也不会执行时区调整。这与datetime1 +（-timedelta）并不完全等价，因为在datetime1 - timedelta没有的情况下，单独的-timedelta可能会溢出。

3. 仅当两个操作数都是天真的，或者如果两者都知道的话，才datetime从a中减去a datetime。如果一个人意识到而另一个天真，TypeError则会被提出。

   如果两者都幼稚，或两者都知道并具有相同的tzinfo属性，则tzinfo属性将被忽略，其结果是一个timedelta 对象*吨*使得。在这种情况下，不会进行时区调整。 datetime2 + t == datetime1

   如果两者都知道并且具有不同的tzinfo属性，a-b就好像*a*和*b*首先被转换为朴素的UTC日期时间。结果是除了实现不会溢出。 (a.replace(tzinfo=None) - a.utcoffset()) - (b.replace(tzinfo=None) - b.utcoffset())

4. 当*datetime1*在*datetime2*之前 及时时，*datetime1*被认为小于*datetime2*。

   如果一个比较数是天真的，另一个是知道的，TypeError 如果尝试进行定单比较则会被提出。对于相等比较，天真实例永远不会等于知道的实例。

   如果两个比较数据都知道并且具有相同的tzinfo属性，tzinfo则忽略通用属性并比较基准日期时间。如果两个比较值都知道并具有不同的tzinfo 属性，则首先通过减去它们的UTC偏移量（从中获得self.utcoffset()）来调整比较值。

   *版本3.3中更改：*天真和感知datetime 实例之间的平等比较不会升高TypeError。

   > **注意：** 为了停止比较从回落到比较对象地址的默认方案，TypeError如果另一个比较值不是对象，则日期时间比较通常会引发datetime。但是，NotImplemented如果其他比较具有timetuple()属性， 则返回 。这个钩子为其他类型的日期对象提供了一个实现混合类型比较的机会。如果没有，则将datetime 对象与不同类型的对象进行TypeError比较时，除非比较为==或，否则将引发该对象!=。后面的情况分别返回 False或True。

datetime对象可以用作字典键。在布尔上下文中，所有datetime对象都被认为是真的。

实例方法：

datetime.**date**( )
返回date年份，月份和日期相同的对象。

datetime.**time**( )
time以相同的小时，分钟，秒，微秒和折叠返回对象。 tzinfo是None。另见方法timetz()。

*在版本3.6中更改：*折叠值被复制到返回的time对象。

datetime.**timetz**()

time使用相同的小时，分钟，秒，微秒，折叠和tzinfo属性返回对象。另见方法time()。

*在版本3.6中更改：*折叠值被复制到返回的time对象。

datetime.**replace**(*year = self.year*, *month = self.month*, *day = self.day*, *hour = self.hour*, *minute = self.minute*, *second = self.second*, *microsecond = self.microsecond*, *tzinfo = self.tzinfo*, ** = 0*)

使用相同的属性返回一个日期时间，除了那些通过指定任何关键字参数给出新值的属性。请注意， tzinfo=None可以指定从可识别的日期时间创建天真的日期时间，而不会转换日期和时间数据。

*3.6版新增功能：*增加了fold参数。

datetime.**astimezone**(*tz =无*)

返回datetime带有新tzinfo属性*tz*的对象，调整日期和时间数据，以便UTC与*自身的*时间相同，但在*tz*的当地时间。

如果提供，*TZ*必须是一个实例tzinfo子类，它 utcoffset()和dst()方法不能返回None。如果*自我*是天真的（），它被认为代表系统时区中的时间。 self.tzinfo is None

如果不带参数调用（或使用tz=None），系统本地时区被假定为目标时区。.tzinfo转换后的日期时间实例的属性将设置为具有timezone 从OS获得的区域名称和偏移量的实例。

如果self.tzinfo是*tz*，self.astimezone(tz)就等于*self*：不调整日期或时间数据。否则的结果是当地时间的时区*TZ*，表示相同UTC时间的*自我*：之后 ，将具有相同的日期和时间数据。 astz = dt.astimezone(tz) astz - astz.utcoffset() dt - dt.utcoffset()

如果仅仅想将时区对象*tz*附加到日期时间*dt*而不调整日期和时间数据，请使用dt.replace(tzinfo=tz)。如果您只想在不转换日期和时间数据的情况下从感知日期时间*dt中*删除时区对象，请使用dt.replace(tzinfo=None)。

请注意，默认tzinfo.fromutc()方法可以在tzinfo子类中重写 以影响返回的结果astimezone()。忽略错误情况，astimezone()行为如下：

```
def astimezone(self, tz):
    if self.tzinfo is tz:
        return self
    # Convert self to UTC, and attach the new time zone object.
    utc = (self - self.utcoffset()).replace(tzinfo=tz)
    # Convert from UTC to tz's local time.
    return tz.fromutc(utc)
```

*在版本3.3中更改：*tz现在可以省略。

*在版本3.6中更改：*astimezone()现在可以在假定表示系统本地时间的天真实例上调用该方法。

datetime.**utcoffset**()

如果tzinfo是None，则返回None，否则返回 self.tzinfo.utcoffset(self)，如果后者不返回None，则引发异常，或者timedelta表示整数分钟的对象，其大小小于一天。

datetime.**dst**( )

如果tzinfo是None，则返回None，否则返回 self.tzinfo.dst(self)，如果后者不返回None，则引发异常 ，或者timedelta表示整数分钟的对象，其大小小于一天。

datetime.**tzname**( )

如果tzinfo是None，则返回None，否则返回 self.tzinfo.tzname(self)，如果后者不返回None或发生字符串对象则引发异常 ，

datetime.**timetuple**( )

返回一个time.struct_time如返回的time.localtime()。 d.timetuple()相当于今年1月1 日开始的当天数字在哪里。结果标志根据方法设置：is 或returns ，设置为; 否则，如果返回一个非零值，则设置为; 其他设置为。 time.struct_time((d.year, d.month, d.day, d.hour, d.minute, d.second, d.weekday(), yday, dst)) yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1 1 tm_isdst dst() tzinfo None dst() None tm_isdst - 1 dst() tm_isdst 1 tm_isdst 0

datetime.**utctimetuple**( )

如果datetime实例*d*天真，这与d.timetuple()除了tm_isdst被强制设置为0 以外的情况相同，无论d.dst()返回的是什么 。DST在UTC时间从未有效。

如果*d*知道，则*d*被标准化为UTC时间，通过减去 d.utcoffset()，并且time.struct_time返回归一化时间的a 。 tm_isdst 被强制为0.请注意， OverflowError 如果*d*。 年 MINYEAR 和或 MAXYEARUTC调整溢出一年边界，则可能会提高。

datetime.**toordinal**( )

返回日期的格雷戈里序数。和self.date().toordinal()。一样 。

datetime.**timestamp**( )

返回datetime 实例对应的POSIX时间戳。返回值float与返回值类似time.time()。

初始datetime实例被假定为表示本地时间，并且此方法依赖于平台C mktime() 函数来执行转换。由于datetime 支持的值范围比mktime()在许多平台上的范围更广，因此这种方法可能会OverflowError在过去或未来的很长时间内提高。

对于意识到的datetime实例，返回值计算如下：

```
(dt - datetime(1970, 1, 1, tzinfo=timezone.utc)).total_seconds()
```

3.3版本的新功能

*在版本3.6中更改：*该timestamp()方法使用该fold属性来消除重复时间间隔内的时间。

注意: 没有办法直接从datetime代表UTC时间的朴素实例获取POSIX时间戳。如果您的应用程序使用此惯例并且您的系统时区未设置为UTC，则可以通过提供tzinfo=timezone.utc以下内容来获取POSIX时间戳 ：

```
timestamp = dt.replace(tzinfo=timezone.utc).timestamp()
```

或者直接计算时间戳：

```
timestamp = (dt - datetime(1970, 1, 1)) / timedelta(seconds=1)
```

datetime.**weekday**( )

将星期几作为整数返回，其中星期一为0，星期日为6.与self.date().weekday()。相同。另见isoweekday()。

datetime.**isoweekday**( )

将星期几作为整数返回，其中星期一为1，星期日为7.与self.date().isoweekday()。相同。另见weekday()， isocalendar()。

datetime.**isocalendar**( )

返回3元组（ISO年，ISO周编号，ISO工作日）。和self.date().isocalendar()。一样。

datetime.**isoformat**(*sep='T'*, *timespec='auto'*)

以ISO 8601格式YYYY-MM-DDTHH：MM：SS.mmmmmm返回表示日期和时间的字符串，如果microsecond为0，则返回YYYY-MM-DDTHH：MM：SS

如果utcoffset()不返回None，则会附加一个6个字符的字符串，给出（带符号）小时和分钟的UTC偏移：YYYY-MM-DDTHH：MM：SS.mmmmmm + HH：MM或者如果microsecond为0 YYYY-MM-DDTHH：MM：SS + HH：MM

可选参数*sep*（默认'T'）是一个单字符分隔符，位于结果的日期和时间部分之间。例如，

```
>>> from datetime import tzinfo, timedelta, datetime
>>> class TZ(tzinfo):
...     def utcoffset(self, dt): return timedelta(minutes=-399)
...
>>> datetime(2002, 12, 25, tzinfo=TZ()).isoformat(' ')
'2002-12-25 00:00:00-06:39'
```

可选参数*timespec*指定要包含的附加组件的数量（默认值'auto'）。它可以是以下之一：

- 'auto'：同'seconds'，如果microsecond是0，同'microseconds'其他。
- 'hours'：包含hour两位数的HH格式。
- 'minutes'：包含hour并minute以HH：MM格式。
- 'seconds'：包括hour，minute和second HH：MM：SS格式。
- 'milliseconds'：包含全部时间，但截断小数秒部分到毫秒。HH：MM：SS.sss格式。
- 'microseconds'：将全职时间包括在HH：MM：SS.mmmmmm格式中。

> **注意：** 排除时间组件被截断，而不是四舍五入。

ValueError将在无效的*timespec*参数上提出。

```
>>> from datetime import datetime
>>> datetime.now().isoformat(timespec='minutes')
'2002-12-25T00:00'
>>> dt = datetime(2015, 1, 1, 12, 30, 59, 0)
>>> dt.isoformat(timespec='microseconds')
'2015-01-01T12:30:59.000000'
```

*3.6版新增功能：*添加了*timespec*参数。

datetime.**__str__**( )

对于datetime实例*d而言*，str(d)相当于。d.isoformat(' ')

datetime.**ctime ( )**

例如，返回表示日期和时间的字符串。相当于在本地C 函数（调用 但未调用）符合C标准的平台 上 。 datetime(2002, 12, 4, 20, 30, 40).ctime() == 'Wed Dec 4 20:30:40 2002' d.ctime() time.ctime(time.mktime(d.timetuple())) ctime() time.ctime() datetime.ctime()

datetime.**strftime (*格式*)**

返回一个表示日期和时间的字符串，由一个明确的格式字符串控制。有关格式化指令的完整列表，请参阅 strftime ( ) 和strptime ( ) 行为。

datetime.**__format__ (*格式*)**

和...一样datetime.strftime()。这使得可以为格式化字符串文本中的datetime对象和使用时的格式字符串指定格式字符串。有关格式化指令的完整列表，请参阅 strftime ( ) 和strptime ( ) 行为。 str.format()

使用日期时间对象的示例：

```
>>> from datetime import datetime, date, time
>>> # Using datetime.combine()
>>> d = date(2005, 7, 14)
>>> t = time(12, 30)
>>> datetime.combine(d, t)
datetime.datetime(2005, 7, 14, 12, 30)
>>> # Using datetime.now() or datetime.utcnow()
>>> datetime.now()
datetime.datetime(2007, 12, 6, 16, 29, 43, 79043)   # GMT +1
>>> datetime.utcnow()
datetime.datetime(2007, 12, 6, 15, 29, 43, 79060)
>>> # Using datetime.strptime()
>>> dt = datetime.strptime("21/11/06 16:30", "%d/%m/%y %H:%M")
>>> dt
datetime.datetime(2006, 11, 21, 16, 30)
>>> # Using datetime.timetuple() to get tuple of all attributes
>>> tt = dt.timetuple()
>>> for it in tt:
...     print(it)
...
2006    # year
11      # month
21      # day
16      # hour
30      # minute
0       # second
1       # weekday (0 = Monday)
325     # number of days since 1st January
-1      # dst - method tzinfo.dst() returned None
>>> # Date in ISO format
>>> ic = dt.isocalendar()
>>> for it in ic:
...     print(it)
...
2006    # ISO year
47      # ISO week
2       # ISO weekday
>>> # Formatting datetime
>>> dt.strftime("%A, %d. %B %Y %I:%M%p")
'Tuesday, 21. November 2006 04:30PM'
```

```
>>> 'The {1} is {0:%d}, the {2} is {0:%B}, the {3} is {0:%I:%M%p}.'.format(dt, "day", "mo
'The day is 21, the month is November, the time is 04:30PM.'
```

在tzinfo中使用datetime：

```
>>> from datetime import timedelta, datetime, tzinfo
>>> class GMT1(tzinfo):
...     def utcoffset(self, dt):
...         return timedelta(hours=1) + self.dst(dt)
...     def dst(self, dt):
...         # DST starts last Sunday in March
...         d = datetime(dt.year, 4, 1)   # ends last Sunday in October
...         self.dston = d - timedelta(days=d.weekday() + 1)
...         d = datetime(dt.year, 11, 1)
...         self.dstoff = d - timedelta(days=d.weekday() + 1)
...         if self.dston <=  dt.replace(tzinfo=None) < self.dstoff:
...             return timedelta(hours=1)
...         else:
...             return timedelta(0)
...     def tzname(self,dt):
...          return "GMT +1"
...
>>> class GMT2(tzinfo):
...     def utcoffset(self, dt):
...         return timedelta(hours=2) + self.dst(dt)
...     def dst(self, dt):
...         d = datetime(dt.year, 4, 1)
...         self.dston = d - timedelta(days=d.weekday() + 1)
...         d = datetime(dt.year, 11, 1)
...         self.dstoff = d - timedelta(days=d.weekday() + 1)
...         if self.dston <=  dt.replace(tzinfo=None) < self.dstoff:
...             return timedelta(hours=1)
...         else:
...             return timedelta(0)
...     def tzname(self,dt):
...         return "GMT +2"
...
>>> gmt1 = GMT1()
>>> # Daylight Saving Time
>>> dt1 = datetime(2006, 11, 21, 16, 30, tzinfo=gmt1)
>>> dt1.dst()
datetime.timedelta(0)
>>> dt1.utcoffset()
datetime.timedelta(0, 3600)
>>> dt2 = datetime(2006, 6, 14, 13, 0, tzinfo=gmt1)
>>> dt2.dst()
datetime.timedelta(0, 3600)
>>> dt2.utcoffset()
datetime.timedelta(0, 7200)
>>> # Convert datetime to another time zone
>>> dt3 = dt2.astimezone(GMT2())
>>> dt3
datetime.datetime(2006, 6, 14, 14, 0, tzinfo=<GMT2 object at 0x...>)
>>> dt2
datetime.datetime(2006, 6, 14, 13, 0, tzinfo=<GMT1 object at 0x...>)
>>> dt2.utctimetuple() == dt3.utctimetuple()
True
```

8.1.5。time对象

时间对象表示一天中的（当地）时间，与任何特定日子无关，并且可以通过tzinfo对象进行调整。

class datetime.**time**（*hour = 0*，*minute = 0*，*second = 0*，*microsecond = 0*，*tzinfo = None*，***，*fold = 0* ）

所有参数都是可选的。 *tzinfo*可能是None一个tzinfo子类的实例 。其余参数可以是整数，范围如下：

- 0 <= hour < 24，
- 0 <= minute < 60，
- 0 <= second < 60，
- 0 <= microsecond < 1000000，
- fold in [0, 1]。

如果超出这些范围的论点ValueError被提出。0除*tzinfo*之外，所有的默认值都是默认的None。

类属性：

time.**min**

最早的代表time ，。 time(0, 0, 0, 0)

time.**max**

最新的代表time ，。 time(23, 59, 59, 999999)

time.**resolution**

非平等之间的最小可能差time的对象， timedelta(microseconds=1)虽然注意到算术 time不支持的对象。

实例属性（只读）：

time.**hour**

在range(24)。

time.**minute**

在range(60)。

time.**second**

在range(60)。

time.**microsecond**

在range(1000000)。

time.**tzinfo**

该对象作为tzinfo参数传递给time构造函数，或者 None如果没有传递。

time.**fold**

在。用于在重复的时间间隔内消除墙壁时间的歧义。（在夏令时结束时回滚时钟或由于政治原因导致当前区域的UTC偏移量减少时发生重复的时间间隔。）值0（1）表示两个时刻中较早（较晚）的时间相同的墙壁时间表示。[0, 1]

3.6版本中的新功能。

支持的操作：

- 的比较`time`来`time`，其中*一个*被认为是小于*b*当*一先b*中的时间。如果一个比较数是天真的，另一个是知道的，`TypeError`如果尝试进行定单比较则会被提出。对于相等比较，天真实例永远不会等于知道的实例。

  如果两个比较都知道，并且具有相同的`tzinfo`属性，`tzinfo`则忽略通用属性并比较基准时间。如果两个比较值都知道并具有不同的`tzinfo`属性，则首先通过减去它们的UTC偏移量（从中获得`self.utcoffset()`）来调整比较值。为了阻止混合类型的比较回落到按对象地址进行默认比较，当一个`time`对象与不同类型的对象进行`TypeError`比较时，除非比较结果为`==`或，否则将引发该混合类型比较`!=`。后面的情况分别返回`False`或`True`。

  *版本3.3中更改：*天真和感知`time`实例之间的平等比较不会升高`TypeError`。

- 散列，用作字典键
- 高效的酸洗

在布尔上下文中，一个`time`对象总是被认为是真的。

*在3.5版本中更改：*在Python 3.5之前，如果一个`time`对象表示UTC的午夜，则该对象被认为是错误的。这种行为被认为是模糊的，容易出错，并且在Python 3.5中已被删除。完整的细节见bpo-13936。

实例方法：

`time.` **`replace`** (*hour = self.hour*，*minute = self.minute*，*second = self.second*，*microsecond = self.microsecond*，*tzinfo = self.tzinfo*，** fold = 0*)

`time`使用相同的值返回a，除了通过指定任何关键字参数给出新值的那些属性。请注意，`tzinfo=None`可以指定`time`从知道创建一个天真的`time`，而不需要转换时间数据。

*3.6版*新增功能：增加了`fold`参数。

`time.` **`isoformat`** (*timespec ='auto'*)

以ISO 8601格式返回表示时间的字符串，HH：MM：SS.mmmmmm，或者如果`microsecond`为0，`utcoffset()`则返回HH：MM：SS如果不返回`None`，则会追加6个字符的字符串，小时和分钟：HH：MM：SS.mmmmmm + HH：MM或，如果self.microsecond为0，则HH：MM：SS + HH：MM

可选参数*timespec*指定要包含的附加组件的数量（默认值`'auto'`）。它可以是以下之一：

- `'auto'`：同`'seconds'`，如果`microsecond`是0，同`'microseconds'`其他。
- `'hours'`：包含`hour`两位数的HH格式。
- `'minutes'`：包含`hour`并`minute`以HH：MM格式。
- `'seconds'`：包括`hour`，`minute`和`second` HH：MM：SS格式。
- `'milliseconds'`：包含全部时间，但截断小数秒部分到毫秒。HH：MM：SS.sss格式。
- `'microseconds'`：将全职时间包括在HH：MM：SS.mmmmmm格式中。

> **注意：** 排除时间组件被截断，而不是四舍五入。

`ValueError`将在无效的*timespec*参数上提出。

```
>>> from datetime import time
>>> time(hour=12, minute=34, second=56, microsecond=123456).isoformat(timespec='minute
'12:34'
>>> dt = time(hour=12, minute=34, second=56, microsecond=0)
>>> dt.isoformat(timespec='microseconds')
'12:34:56.000000'
>>> dt.isoformat(timespec='auto')
'12:34:56'
```

3.6版新增功能：添加了timespec参数。

time.__str__()

在时间*t*，str(t)相当于t.isoformat()。

time.strftime(*格式*)

返回一个表示时间的字符串，由一个明确的格式字符串控制。有关格式化指令的完整列表，请参阅 strftime() 和strptime() 行为。

time.__format__(*格式*)

和...一样time.strftime()。这使得可以为格式化字符串文本中的time对象和使用时的格式字符串指定格式字符串。有关格式化指令的完整列表，请参阅 strftime() 和strptime() 行为。str.format()

time.utcoffset()

如果tzinfo是None，返回None，否则返回self.tzinfo.utcoffset(None)，并且如果后者不返回，则引发异常，None或者timedelta表示整数分钟的对象，其大小小于一天。

time.dst()

如果tzinfo是None，则返回None，否则返回 self.tzinfo.dst(None)，如果后者不返回None，则引发异常 ，或者timedelta表示整数分钟的对象，其大小小于一天。

time.tzname()

如果tzinfo是None，返回None，否则返回 self.tzinfo.tzname(None)，或者如果后者没有返回None或字符串对象则引发异常。

例：

```
>>> from datetime import time, tzinfo, timedelta
>>> class GMT1(tzinfo):
...     def utcoffset(self, dt):
...         return timedelta(hours=1)
...     def dst(self, dt):
...         return timedelta(0)
...     def tzname(self,dt):
...         return "Europe/Prague"
...
>>> t = time(12, 10, 30, tzinfo=GMT1())
>>> t
datetime.time(12, 10, 30, tzinfo=<GMT1 object at 0x...>)
>>> gmt = GMT1()
>>> t.isoformat()
'12:10:30+01:00'
>>> t.dst()
```

```
datetime.timedelta(0)
>>> t.tzname()
'Europe/Prague'
>>> t.strftime("%H:%M:%S %Z")
'12:10:30 Europe/Prague'
>>> 'The {} is {:%H:%M}.'.format("time", t)
'The time is 12:10.'
```

8.1.6。tzinfo对象

*类*datetime.tzinfo

这是一个抽象基类，意味着这个类不应该直接实例化。您需要派生一个具体的子类，并且（至少）提供您使用的tzinfo方法所需的标准方法的 实现datetime。该datetime模块提供的一个简单的具体子类tzinfo，timezone，其可以代表与从UTC固定偏移，如UTC本身或北美EST和EDT时区。

（的具体子类）的实例tzinfo可以传递给构造函数datetime和time对象。后者对象将它们的属性视为在本地时间，并且该tzinfo对象支持揭示UTC的本地时间偏移量，时区名称和DST偏移量的方法，所有这些方法都相对于传递给它们的日期或时间对象。

酸洗的特殊要求：一个tzinfo子类必须有一个__init__()可以不带参数调用的 方法，否则它可能会被腌渍，但可能不会再次取消。这是未来可能会放松的技术要求。

具体的子类tzinfo可能需要实现以下方法。究竟需要哪些方法取决于由感知datetime对象组成的用途 。如果有疑问，只需实施所有这些。

tzinfo.utcoffset(*dt*)

在UTC以东的分钟内，从UTC返回当地时间的偏移量。如果当地时间在UTC以西，则应该是负值。请注意，这应该是UTC的总偏移量; 例如，如果某个tzinfo对象同时代表时区和DST调整，utcoffset()则应返回它们的总和。如果UTC偏移量未知，则返回None。否则，返回的值必须是一个 timedelta对象，指定在-1439至1439（含1440 = 24 * 60;偏移的幅度必须小于一天）范围内的整数分钟。大多数的实现utcoffset()可能看起来像这两个之一：

```
return CONSTANT                 # fixed-offset class
return CONSTANT + self.dst(dt)  # daylight-aware class
```

如果utcoffset()不返回None，dst()也不应该返回 None。

utcoffset()引发 的默认实现NotImplementedError。

tzinfo.dst(*dt*)

返回UTC以东分钟的夏令时（DST）调整，或者 None如果DST信息未知。 timedelta(0)如果DST没有生效，则返回。如果DST生效，请将偏移量作为timedelta对象返回（ utcoffset()详情请参阅）。请注意，如果适用，DST偏移量已被添加到返回的UTC偏移量utcoffset()，因此dst()除非您有兴趣单独获取DST信息，否则无需咨询。例如，datetime.timetuple()调用其tzinfo 属性的dst()方法来确定如何设置tm_isdst标志，并tzinfo.fromutc()调用dst()跨越时区时考虑DST更改。

在这个意义上，模拟标准时间和日光时间的子类的实例*tz*tzinfo必须一致：

```
tz.utcoffset(dt) - tz.dst(dt)
```

必须为每个*dt*返回与 For sane 子类相同的结果，这个表达式产生时区的“标准偏移量”，它不应该取决于日期或时间，而只取决于地理位置。 依赖于此的实施，却无法检测到违规行为; 程序员有责任确保它。如果一个子类不能保证这一点，它可能能够重写默认实现， 以便与不管。 datetime dt.tzinfo == tz tzinfo datetime.astimezone() tzinfo tzinfo.fromutc() astimezone()

大多数的实现dst()可能看起来像这两个之一：

```
def dst(self, dt):
    # a fixed-offset class:  doesn't account for DST
    return timedelta(0)
```

要么

```
def dst(self, dt):
    # Code to set dston and dstoff to the time zone's DST
    # transition times based on the input dt.year, and expressed
    # in standard local time.  Then

    if dston <= dt.replace(tzinfo=None) < dstoff:
        return timedelta(hours=1)
    else:
        return timedelta(0)
```

dst()引发的默认实现NotImplementedError。

tzinfo.**tzname**(*dt*)

以字符串形式返回与datetime对象*dt*相对应的时区名称。关于字符串名称的任何内容都不是由datetime模块定义的，并且没有要求它具体说明任何内容。例如，“GMT”，“UTC”，“-500”，“-5：00”，“EDT”，“US / Eastern”，“America / New York”都是有效的答复。None如果字符串名称未知，则返回。请注意，这是一种方法，而不是固定字符串，主要是因为某些tzinfo子类会根据传递的*dt*的具体值返回不同的名称，特别是如果tzinfo类占了白天时间。

tzname()引发的默认实现NotImplementedError。

这些方法由一个datetime或一个time对象调用，以响应它们的相同名称的方法。一个datetime对象将自身作为参数传递，并且一个time对象None作为参数传递。一个tzinfo因此子类的方法应该准备接受*DT*的说法None，或一类datetime。

什么时候None通过，是由班级设计师来决定最佳答案。例如，None如果班级希望说时间对象不参与tzinfo协议，则返回是合适的。utcoffset(None)返回标准UTC偏移量可能更有用，因为没有其他约定用于发现标准偏移量。

当datetime响应某个datetime方法传递一个对象时，dt.tzinfo它与*自己*是同一个对象。 tzinfo方法可以依赖于此，除非用户代码tzinfo直接调用方法。意图是这些tzinfo方法将*dt*解释为在当地时间，而不需要担心其他时区中的对象。

还有tzinfo一个子类可能希望覆盖的方法：

tzinfo.**fromutc**(*dt*)

这是从默认datetime.astimezone()实现中调用的。当从那里调用时，dt.tzinfo是*自己的*，*dt*的日期和时间数据被视为表示UTC时间。的目的fromutc()是调整日期和时间数据，在返回等效

的datetime *自*本地时间。

大多数tzinfo子类应该能够继承默认 fromutc()实现而不会出现问题。它足够强大，可以处理固定偏移时区，并且时区可以兼顾标准时间和日光时间，而后者即使在不同年份的DST过渡时间不同。 fromutc() 在所有情况下，默认实现可能无法正确处理的时区示例是标准偏移量（来自UTC）取决于特定日期和时间的情况，这可能出于政治原因而发生。如果结果是跨越标准偏移量更改时的其中一个小时，则默认实现astimezone()和 fromutc()可能不会生成所需的结果。

跳过错误情况的代码，默认fromutc()实现的行为如下所示：

```
def fromutc(self, dt):
    # raise ValueError error if dt.tzinfo is not self
    dtoff = dt.utcoffset()
    dtdst = dt.dst()
    # raise ValueError if dtoff is None or dtdst is None
    delta = dtoff - dtdst  # this is self's standard offset
    if delta:
        dt += delta   # convert to standard local time
        dtdst = dt.dst()
        # raise ValueError if dtdst is None
    if dtdst:
        return dt + dtdst
    else:
        return dt
```

示例tzinfo类：

```
from datetime import tzinfo, timedelta, datetime, timezone

ZERO = timedelta(0)
HOUR = timedelta(hours=1)
SECOND = timedelta(seconds=1)

# A class capturing the platform's idea of local time.
# (May result in wrong values on historical times in
#  timezones where UTC offset and/or the DST rules had
#  changed in the past.)
import time as _time

STDOFFSET = timedelta(seconds = -_time.timezone)
if _time.daylight:
    DSTOFFSET = timedelta(seconds = -_time.altzone)
else:
    DSTOFFSET = STDOFFSET

DSTDIFF = DSTOFFSET - STDOFFSET

class LocalTimezone(tzinfo):

    def fromutc(self, dt):
        assert dt.tzinfo is self
        stamp = (dt - datetime(1970, 1, 1, tzinfo=self)) // SECOND
        args = _time.localtime(stamp)[:6]
        dst_diff = DSTDIFF // SECOND
        # Detect fold
        fold = (args == _time.localtime(stamp - dst_diff))
        return datetime(*args, microsecond=dt.microsecond,
                        tzinfo=self, fold=fold)
```

```
    def utcoffset(self, dt):
        if self._isdst(dt):
            return DSTOFFSET
        else:
            return STDOFFSET

    def dst(self, dt):
        if self._isdst(dt):
            return DSTDIFF
        else:
            return ZERO

    def tzname(self, dt):
        return _time.tzname[self._isdst(dt)]

    def _isdst(self, dt):
        tt = (dt.year, dt.month, dt.day,
              dt.hour, dt.minute, dt.second,
              dt.weekday(), 0, 0)
        stamp = _time.mktime(tt)
        tt = _time.localtime(stamp)
        return tt.tm_isdst > 0

Local = LocalTimezone()


# A complete implementation of current DST rules for major US time zones.

def first_sunday_on_or_after(dt):
    days_to_go = 6 - dt.weekday()
    if days_to_go:
        dt += timedelta(days_to_go)
    return dt


# US DST Rules
#
# This is a simplified (i.e., wrong for a few cases) set of rules for US
# DST start and end times. For a complete and up-to-date set of DST rules
# and timezone definitions, visit the Olson Database (or try pytz):
# http://www.twinsun.com/tz/tz-link.htm
# http://sourceforge.net/projects/pytz/ (might not be up-to-date)
#
# In the US, since 2007, DST starts at 2am (standard time) on the second
# Sunday in March, which is the first Sunday on or after Mar 8.
DSTSTART_2007 = datetime(1, 3, 8, 2)
# and ends at 2am (DST time) on the first Sunday of Nov.
DSTEND_2007 = datetime(1, 11, 1, 2)
# From 1987 to 2006, DST used to start at 2am (standard time) on the first
# Sunday in April and to end at 2am (DST time) on the last
# Sunday of October, which is the first Sunday on or after Oct 25.
DSTSTART_1987_2006 = datetime(1, 4, 1, 2)
DSTEND_1987_2006 = datetime(1, 10, 25, 2)
# From 1967 to 1986, DST used to start at 2am (standard time) on the last
# Sunday in April (the one on or after April 24) and to end at 2am (DST time)
# on the last Sunday of October, which is the first Sunday
# on or after Oct 25.
DSTSTART_1967_1986 = datetime(1, 4, 24, 2)
DSTEND_1967_1986 = DSTEND_1987_2006
```

```
def us_dst_range(year):
    # Find start and end times for US DST. For years before 1967, return
    # start = end for no DST.
    if 2006 < year:
        dststart, dstend = DSTSTART_2007, DSTEND_2007
    elif 1986 < year < 2007:
        dststart, dstend = DSTSTART_1987_2006, DSTEND_1987_2006
    elif 1966 < year < 1987:
        dststart, dstend = DSTSTART_1967_1986, DSTEND_1967_1986
    else:
        return (datetime(year, 1, 1), ) * 2

    start = first_sunday_on_or_after(dststart.replace(year=year))
    end = first_sunday_on_or_after(dstend.replace(year=year))
    return start, end


class USTimeZone(tzinfo):

    def __init__(self, hours, reprname, stdname, dstname):
        self.stdoffset = timedelta(hours=hours)
        self.reprname = reprname
        self.stdname = stdname
        self.dstname = dstname

    def __repr__(self):
        return self.reprname

    def tzname(self, dt):
        if self.dst(dt):
            return self.dstname
        else:
            return self.stdname

    def utcoffset(self, dt):
        return self.stdoffset + self.dst(dt)

    def dst(self, dt):
        if dt is None or dt.tzinfo is None:
            # An exception may be sensible here, in one or both cases.
            # It depends on how you want to treat them.  The default
            # fromutc() implementation (called by the default astimezone()
            # implementation) passes a datetime with dt.tzinfo is self.
            return ZERO
        assert dt.tzinfo is self
        start, end = us_dst_range(dt.year)
        # Can't compare naive to aware objects, so strip the timezone from
        # dt first.
        dt = dt.replace(tzinfo=None)
        if start + HOUR <= dt < end - HOUR:
            # DST is in effect.
            return HOUR
        if end - HOUR <= dt < end:
            # Fold (an ambiguous hour): use dt.fold to disambiguate.
            return ZERO if dt.fold else HOUR
        if start <= dt < start + HOUR:
            # Gap (a non-existent hour): reverse the fold rule.
            return HOUR if dt.fold else ZERO
        # DST is off.
```

```
        return ZERO

    def fromutc(self, dt):
        assert dt.tzinfo is self
        start, end = us_dst_range(dt.year)
        start = start.replace(tzinfo=self)
        end = end.replace(tzinfo=self)
        std_time = dt + self.stdoffset
        dst_time = std_time + HOUR
        if end <= dst_time < end + HOUR:
            # Repeated hour
            return std_time.replace(fold=1)
        if std_time < start or dst_time >= end:
            # Standard time
            return std_time
        if start <= std_time < end - HOUR:
            # Daylight saving time
            return dst_time


Eastern  = USTimeZone(-5, "Eastern",  "EST", "EDT")
Central  = USTimeZone(-6, "Central",  "CST", "CDT")
Mountain = USTimeZone(-7, "Mountain", "MST", "MDT")
Pacific  = USTimeZone(-8, "Pacific",  "PST", "PDT")
```

请注意tzinfo ，在DST转换点，标准时间和日光时间的子类中每年有两次不可避免的微妙事件。具体方面，请考虑美国东部时间（UTC -0500），其中美国东部时间在美国东部时间3月第二个星期日1:59（美国东部时间）之后的分钟开始，并在11月第一个星期日1:59（美国东部时间）之后的分钟结束：

```
  UTC   3:MM  4:MM  5:MM  6:MM  7:MM  8:MM
  EST  22:MM 23:MM  0:MM  1:MM  2:MM  3:MM
  EDT  23:MM  0:MM  1:MM  2:MM  3:MM  4:MM

start  22:MM 23:MM  0:MM  1:MM  3:MM  4:MM

  end  23:MM  0:MM  1:MM  1:MM  2:MM  3:MM
```

当DST开始时（“开始”行），本地挂钟从1:59跳到3:00。表格2：MM的墙上时间在当天并没有意义，所以 astimezone(Eastern)不会在DST开始的那天提供结果。例如，在2016年春季前瞻性转型中，我们得到了hour == 2

```
>>> u0 = datetime(2016, 3, 13, 5, tzinfo=timezone.utc)
>>> for i in range(4):
...     u = u0 + i*HOUR
...     t = u.astimezone(Eastern)
...     print(u.time(), 'UTC =', t.time(), t.tzname())
...
05:00:00 UTC = 00:00:00 EST
06:00:00 UTC = 01:00:00 EST
07:00:00 UTC = 03:00:00 EDT
08:00:00 UTC = 04:00:00 EDT
```

当DST结束时（“结束”），有一个潜在的更糟糕的问题：有一个小时不能在本地时间内明确拼写：夏时制的最后一个小时。在东部，这是当天白天时间形式为5：MM UTC的时间结束。当地的挂钟从1:59（白天时间）再次跳回到1:00（标准时间）。表格1：MM的当地时间不明确。 astimezone()通过将两个相邻的UTC小时映射到当时相同的小时来模拟本地时钟的行为。在东部示例中，格式

5：MM和6：MM的UTC时间在转换为东部时映射到1：MM，但较早的时间将fold属性设置为0，而较晚的时间将其设置为1.例如，在2016年的倒退过渡中，我们得到了

```
>>> u0 = datetime(2016, 11, 6, 4, tzinfo=timezone.utc)
>>> for i in range(4):
...     u = u0 + i*HOUR
...     t = u.astimezone(Eastern)
...     print(u.time(), 'UTC =', t.time(), t.tzname(), t.fold)
...
04:00:00 UTC = 00:00:00 EDT 0
05:00:00 UTC = 01:00:00 EDT 0
06:00:00 UTC = 01:00:00 EST 1
07:00:00 UTC = 02:00:00 EST 0
```

请注意，datetime只有fold属性值不同的实例 在比较中被认为是相等的。

不能承受时间歧义的应用程序应明确检查fold属性的值或避免使用混合 tzinfo子类; 使用timezone时或任何其他固定偏移tzinfo量子类（例如仅代表EST（固定偏移-5小时）的类，或仅EDT（固定偏移-4小时））时不存在歧义。

也可以看看:

dateutil.tz

标准库具有timezone用于处理来自UTC的任意固定偏移和timezone.utcUTC时区实例的类。

*dateutil.tz*库将*IANA时区数据库*（也称为Olson数据库）引入Python，并建议使用它。

IANA时区数据库

时区数据库（通常称为tz，tzdata或zoneinfo）包含代表全球许多代表性地点的本地时间历史记录的代码和数据。它会定期更新以反映政治团体对时区边界，UTC偏移和夏时制规则所做的更改。

8.1.7。 timezone对象

的timezone类是的一个子类tzinfo，每一个实例，其中表示通过从UTC的固定偏移定义的时区。请注意，此类别的对象不能用于表示一年中不同日期或历史更改时间不同的位置的时区信息。

class datetime. timezone（*offset*，*name* = *None*）

所述*偏移*参数必须被指定为timedelta 代表的本地时间和UTC之间的差对象。它必须严格地在-timedelta(hours=24)和之间， timedelta(hours=24)并且代表整数分钟，否则会ValueError被提出。

该*名称*参数是可选。如果指定，它必须是一个字符串，它将用作datetime.tzname()方法返回的值。

3.2版本中的新功能

timezone. utcoffset（*dt*）

返回timezone构造实例时指定的固定值。该*DT*参数将被忽略。返回值是一个 timedelta实例，等于本地时间和UTC之间的差异。

timezone. tzname（*dt*）

返回timezone构造实例时指定的固定值。如果*名称*未在构造函数中提供，则返回的名称tzname(dt)将offset根据以下值生成 。如果*偏移*是timedelta(0) ，名字是“UTC”，否则它是一个字符串“UTC±HH ： MM”，其中±是符号offset ，HH和MM是两个数字offset.hours 和offset.minutes分别。

*在版本3.6中更改：*从offset=timedelta(0)现在生成的名称现在是纯“UTC”，而不是“UTC + 00 ：00”。

timezone.dst（*dt* ）
始终返回None。

timezone.fromutc（*dt* ）
返回。该*DT*参数必须是一个知道的 情况下，与设置为。dt + offsetdatetimetzinfoself

类属性：

timezone.utc
UTC时区timezone(timedelta(0))。

8.1.8。strftime()和strptime()行为

date，datetime和time所有对象都支持一个 strftime(format)方法，在显式格式字符串的控制下创建一个表示时间的字符串。一般来说，d.strftime(fmt) 就像time模块一样， 尽管不是所有的对象都支持一种方法。time.strftime(fmt, d.timetuple()) timetuple()

相反，datetime.strptime()类方法通过datetime表示日期和时间的字符串以及相应的格式字符串创建一个对象。相当于。datetime.strptime(date_string, format)datetime(*(time.strptime(date_string, format)[0:6]))

对于time对象，不应使用年，月和日的格式代码，因为时间对象不具有此类值。如果无论如何都使用它们，1900 则替换年份以及1月份和日期。

对于date对象，不应使用小时，分钟，秒和微秒的格式代码，因为date对象不具有此类值。如果他们无论如何0都被使用，将被替代。

支持的全套格式代码在不同的平台上有所不同，因为Python将平台C库的strftime()功能称为平台，并且平台变体是常见的。要查看平台上支持的全套格式代码，请参阅*strftime（3）*文档。

以下是C标准（1989版本）要求的所有格式代码的列表，并且这些代码可以在所有平台上使用标准的C实现。请注意，1999版本的C标准添加了额外的格式代码。

指示	含义	例	笔记
%a	星期几作为区域设置的缩写名称。	Sun，Mon，...，Sat（en_US）；所以，Mo，...，Sa（de_DE）	（1）

指示	含义	例	笔记
%A	星期几作为区域的全名。	星期日，星期一，...，星期六（en_US）; Sonntag，Montag，...，Samstag（de_DE）	（1）
%w	星期几作为十进制数字，其中0表示星期	0，1，...，6	
%d	一个月中的一天作为零填充的十进制数字。	01，02，...，31	
%b	月为区域设置的缩写名称。	1月，2月，...，12月（en_US）; Jan，Feb，...，Dez（de_DE）	（1）
%B	月作为区域的全名。	一月，二月，...，十二月（en_US）; Januar，Februar，...，Dezember（de_DE）	（1）
%m	月份作为零填充十进制数字。	01，02，...，12	
%y	没有世纪的一年是一个零填充的十进制数字。	00，01，...，99	
%Y	世纪作为十进制数字。	0001，0002，...，9998，9999	（2）
%H	小时（24小时制）作为零填充十进制数字。	00，01，...，23	
%I	小时（12小时制）作为零填充的十进制数字。	01，02，...，12	
%p	区域设置相当于AM或PM。	AM，PM（en_US）; am，pm（de_DE）	（1），（3）
%M	分钟作为零填充的十进制数字。	00，01，...，59	
%S	其次作为零填充的十进制数字。	00，01，...，59	（4）
%f	微秒为十进制数字，左侧为零填充。	000000，000001，...，	（5）
%z	UTC偏移量格式为+HHMM或-HHMM（如果对象天真，则为空字符	（空），+0000，-0400，+1030	（6）
%Z	时区名称（如果对象天真，则为空字符	（空），UTC，EST，CST	
%j	一年中的一天为零填充的十进制数字。	001，002，...，366	
%U	[illegible]	00，01，...，53	（7）
%W	[illegible]	00，01，...，53	（7）
%c	区域设置的适当日期和时间表示。	星期二8月16日21:30:00 1988（en_US）; Di 8月16日21:30:00 1988（de_DE）	（1）
%x	区域设置的适当日期表示。	08/16/88（None）; 08/16/1988（en_US）; 16.08.1988（de_DE）	（1）
%X	区域设置适当的时间表示。	21:30:00（en_US）; 21:30:00（de_DE）	（1）

指示	含义	例	笔记
%%	字面'%'字符。	%	

为方便起见，还包含了C89标准所不需要的其他一些指令。这些参数全部对应于ISO 8601日期值。与该strftime() 方法一起使用时，这些可能不适用于所有平台。ISO 8601年和ISO 8601周指令不能与上面的年份和周编号指令互换。 strptime()使用不完整或不明确的ISO 8601指令进行调用将引发一个问题ValueError。

指示	含义	例	笔记
%G	世纪ISO 8601年代表包含ISO周的大部分年份（%V）。	0001，0002，...，2013，2014，...，9998，9999年，	（8）
%u	表示星期一星期一作为十进制数字，其中1表示ISO 8601	1，2，...，7	
%V	ISO 8601周作为十进制数字，周一为第一周的开始。第1周是包含1月4日的周	01，02，...，53	（8）

新的3.6版：%G，%u和%V添加。

笔记：

1. 由于格式取决于当前的语言环境，因此在对输出值进行假设时应该小心。字段排序会有所不同（例如，“月/日/年”相对于“日/月/年”）使用区域设置的默认编码进行编码，并输出可能包含Unicode字符（例如，如果当前区域是ja_JP，默认编码可以是中的任一项eucJP，SJIS或utf-8;使用locale.getlocale() 以确定当前区域的编码）。

2. 该strptime()方法可以在完整的[1,9999]范围内解析年份，但年份<1000必须填零至4位宽度。

   *在版本3.2中更改：*在以前的版本中，strftime()方法被限制为年>> = 1900。

   *版本3.3中更改：*在版本3.2中，strftime()方法被限制为年>> 1000。

3. 与strptime()方法一起使用时，%p如果%I指令用于解析小时，则指令仅影响输出小时字段。

4. 与time模块不同，该datetime模块不支持闰秒。

5. 当与该strptime()方法一起使用时，该%f指令在右侧接受一到六位数字和零填充。 %f是C标准中格式字符集的扩展（但是在datetime对象中单独实现，因此始终可用）。

6. 对于一个天真的对象，%z和%Z格式代码被替换为空字符串。

   对于知道的对象：

   %z

   utcoffset()被转换成格式为+ HHMM或-HHMM的5个字符的字符串，其中HH是一个2位字符串，表示UTC偏移小时数，MM是一个2位数字符串，表示UTC偏移分钟数。例如，如果utcoffset()返回 ，则替换为字符串 。 timedelta(hours=-3, minutes=-30)%z'-0330'

   %Z

   如果tzname()返回None，%Z则由空字符串替换。否则%Z被返回的值替换，该值必须是一个字符串。

   *在版本3.2改变：*当%z指令被提供给strptime()法，感知datetime对象会产生。该tzinfo结果将被设置为一个timezone实例。

7. 当与使用strptime()方法，%U以及%W一周的一天，历年（当计算仅使用%Y指定）。

8. 类似%U并且%W，%V在计算时，星期几和ISO年份（只用%G）在指定 strptime() 格式字符串。还要注意%G并且%Y不能互换。

脚注

[1] 如果，那就是我们忽略相对论的影响

8.2。calendar- 与日历相关的常规功能

源代码： Lib / calendar.py

此模块允许您输出日历，如Unix **cal**程序，并提供与日历相关的其他有用功能。默认情况下，这些日历的星期一为一周的第一天，而星期天为最后一天（欧洲大会）。用于setfirstweekday()将星期的第一天设置为星期日（6）或任何其他工作日。指定日期的参数以整数形式给出。有关相关功能，另请参阅datetime和time模块。

这些功能和类中的大多数都依赖于datetime使用理想化日历的模块，当前的格里历日历在两个方向上扩展。这与Dershowitz和Reingold的书“Calendrical Calculations”中的“pregraptic Gregorian”日历的定义相匹配，它是所有计算的基本日历。

class calendar.Calendar（*firstweekday = 0*）

创建一个Calendar对象。*firstweekday*是一个整数，指定一周中的第一天。0是星期一（默认），6是星期天。

甲Calendar对象提供了可用于制备日历数据进行格式化的几种方法。这个类本身不做任何格式化。这是子类的工作。

Calendar 实例具有以下方法：

iterweekdays（）

返回将在一周内使用的星期数字的迭代器。迭代器的第一个值将与该firstweekday属性的值相同。

itermonthdates（*年*，*月*）

返回一个迭代月*月*在今年（1-12）*年*。此迭代器将返回所有datetime.date月份（作为对象）的月份和本月开始前的所有日期或月份结束后所需的一整周。

itermonthdays2（*年*，*月*）

返回一个迭代月*月*在今年*年*类似 itermonthdates()。返回的日期将是由日数和星期数组成的元组。

itermonthdays（*年*，*月*）

返回一个迭代月*月*在今年*年*类似 itermonthdates()。返回的日子将只是日数。

monthdatescalendar（*年*，*月*）

返回在该月的周列表一个*月*中的*年*作为全周。周是七个datetime.date对象的列表。

monthdays2calendar（*年*，*月*）

返回在该月的周列表一个*月*中的*年*作为全周。星期是七天的数字和星期几数字的列表。

monthdayscalendar（*年*，*月*）

返回在该月的周列表一个*月*中的*年*作为全周。周是七天数字的列表。

yeardatescalendar (*year* , *width = 3*)

返回指定年份的数据以准备格式化。返回值是月份行的列表。每个月的行包含最多*宽度*月份（默认为3）。每个月包含4至6周，每周包含1-7天。天是datetime.date物体。

yeardays2calendar (*year* , *width = 3*)

返回准备格式化的指定年份的数据（类似于 yeardatescalendar() ）。星期列表中的条目是日数和星期几数的元组。本月以外的日数为零。

yeardayscalendar (*year* , *width = 3*)

返回准备格式化的指定年份的数据（类似于 yeardatescalendar() ）。星期列表中的条目是天数。本月以外的日数为零。

class calendar. TextCalendar (*firstweekday = 0*)

这个类可以用来生成纯文本日历。

TextCalendar 实例具有以下方法：

formatmonth (*的某些地方* , *themonth* , *W = 0* , *L = 0*)

用多行字符串返回一个月份的日历。如果提供了*w*，它指定了居中的日期列的宽度。如果给出*l*，则它指定每周将使用的行数。取决于构造函数中指定的第一个工作日或由setfirstweekday()方法设置。

prmonth (*的某些地方* , *themonth* , *W = 0* , *L = 0*)

打印返回的月份日历formatmonth()。

formatyear (*的某些地方* , *W = 2* , *L = 1* , *C = 6* , *M = 3*)

将一整年的*m*列日历作为多行字符串返回。可选参数*w*，*l*和*c*分别是日期列宽，每周行数和月份列之间的空格数。取决于构造函数中指定的第一个工作日或由setfirstweekday()方法设置。可以生成日历的最早的一年是平台相关的。

pryear (*的某些地方* , *W = 2* , *L = 1* , *C = 6* , *M = 3*)

打印返回的整年的日历formatyear()。

class calendar. HTMLCalendar (*firstweekday = 0*)

这个类可以用来生成HTML日历。

HTMLCalendar 实例具有以下方法：

formatmonth (*的某些地方* , *themonth* , *withyear =真*)

将一个月份的日历作为HTML表格返回。如果*年份*为真，年份将包含在标题中，否则仅使用月份名称。

formatyear (*theyear* , *width = 3*)

将一年的日历作为HTML表格返回。*宽度*（默认为3）指定每行的月数。

formatyearpage (*theyear* , *width = 3* , *css ='calendar.css'* , *encoding = None*)

将一年的日历作为完整的HTML页面返回。*宽度*（默认为3）指定每行的月数。*css*是要使用的级联样式表的名称。None如果不使用样式表可以通过。*编码*指定要用于输出的

编码（默认为系统默认编码）。

class calendar.LocaleTextCalendar（*firstweekday = 0*，*locale = None* ）

此子类TextCalendar可以在构造函数中传递一个语言环境名称，并将返回指定语言环境中的月份和星期几名称。如果此区域设置包含编码，则所有包含月份和星期几名称的字符串将作为unicode返回。

class calendar.LocaleHTMLCalendar（*firstweekday = 0*，*locale = None* ）

此子类HTMLCalendar可以在构造函数中传递一个语言环境名称，并将返回指定语言环境中的月份和星期几名称。如果此区域设置包含编码，则所有包含月份和星期几名称的字符串将作为unicode返回。

注意： 在formatweekday()和formatmonthname()这两个类的方法当前区域临时改变为给定*的语言环境*。由于当前语言环境是全过程设置，因此它们不是线程安全的。

对于简单的文本日历，该模块提供以下功能。

calendar.setfirstweekday（*周日*）

设置星期几（0星期一，6星期日）每周开始。值MONDAY，TUESDAY，WEDNESDAY，THURSDAY，FRIDAY，SATURDAY，和SUNDAY被提供了方便。例如，要将第一个工作日设置为星期日：

```
import calendar
calendar.setfirstweekday(calendar.SUNDAY)
```

calendar.firstweekday（ ）

返回星期几开始每周的当前设置。

calendar.isleap（*年*）

True如果*年份*是闰年，则返回，否则返回False。

calendar.leapdays（*y1*，*y2* ）

返回从*y1*到*y2*（独占）范围内的闰年数，其中*y1*和*y2*是年。

此功能适用于跨越百年变化的范围。

calendar.weekday（*年*，*月*，*日*）

返回*一年中*（-...），*一个月*（-），*一天*（-）的星期几（0星期一）。19701121 31

calendar.weekheader（*n* ）

返回包含缩写周日名称的标题。*n*指定一个工作日的字符宽度。

calendar.monthrange（*年*，*月*）

返回指定*年份*和*月份*的月份第一天的工作日和月份中的天数。

calendar.monthcalendar（*年*，*月*）

返回表示月份日历的矩阵。每行代表一周; 本月以外的日子由零表示。除非设置，否则每周从星期一开始setfirstweekday()。

calendar.prmonth(*的某些地方*, *themonth*, *W = 0*, *L = 0*)
打印返回的月份日历month()。

calendar.month(*的某些地方*, *themonth*, *W = 0*, *L = 0*)
返回使用多线串一个月的日历formatmonth() 中的TextCalendar类。

calendar.prcal(*年*, *w = 0*, *l = 0*, *c = 6*, *m = 3*)
打印返回的整年的日历 calendar()。

calendar.calendar(*年*, *w = 2*, *l = 1*, *c = 6*, *m = 3*)
返回一整年的3列日历中使用一个多行字符串formatyear()中的TextCalendar类。

calendar.timegm(*元组*)
一个无关但方便的函数，需要一个时间元组（如模块中的gmtime()函数time返回的值），并返回相应的Unix时间戳值（假设为1970年）和POSIX编码。其实，time.gmtime()并且timegm()是彼此的倒数。

该calendar模块导出以下数据属性：

calendar.day_name
表示当前语言环境中每周的几天的数组。

calendar.day_abbr
一个数组，表示当前语言环境中的星期几缩写。

calendar.month_name
一个数组，表示当前语言环境中的一年中的月份。这遵循1月份的正常约定，即1月份，所以它的长度为13，并且 month_name[0]是空字符串。

calendar.month_abbr
一个数组，代表当前语言环境中一年的缩写月份。这遵循1月份的正常约定，即1月份，所以它的长度为13，并且 month_abbr[0]是空字符串。

也可以看看:

模 datetime
面向对象的接口用于与time模块具有类似功能的日期和时间 。

模 time
与时间相关的低级功能。

8.3。collections- 容器数据类型

源代码： Lib / collections / __ init__.py

这个模块实现专门的容器数据类型提供替代Python的通用内置容器中，dict，list， set，和tuple。

namedtuple()	工厂函数用于创建具有命名字段的元组子类
deque	带有快速追加和两端弹出的类列表容器
ChainMap	类似于字典的类来创建多个映射的单个视图
Counter	dict用于计算可哈希对象的子类
OrderedDict	字典子类，记录添加的订单条目
defaultdict	dict调用工厂函数以提供缺失值的子类
UserDict	包装字典对象以便更容易的字典子类化
UserList	将列表对象封装在一起以便更容易地列出子类
UserString	包装字符串对象以便更容易的字符串子类化

*版本3.3中更改：*将集合的抽象基类移至collections.abc模块。为了向后兼容，它们在该模块中也可以继续显示。

8.3.1。ChainMap对象

3.3版本的新功能

ChainMap提供一个类用于快速链接多个映射，以便将它们视为一个单元。它通常比创建新字典和运行多个update()调用要快得多。

该类可以用来模拟嵌套的作用域，并且在模板中很有用。

*类*collections.ChainMap（ **地图* ）

将ChainMap多个字典或其他映射组合在一起以创建单个可更新视图。如果未指定*地图*，则会提供单个空字典，以便新链至少具有一个映射。

基础映射存储在一个列表中。该列表是公开的，可以使用*maps*属性进行访问或更新。没有其他国家。

查找先后搜索底层映射，直到找到密钥。相反，写入，更新和删除仅对第一个映射起作用。

A ChainMap通过引用合并基础映射。因此，如果其中一个基础映射得到更新，这些更改将反映在ChainMap。

所有常用的字典方法都受支持。另外，还有一个 *map*属性，一个用于创建新的子上下文的方法，以及一个用于访问除第一个映射之外的所有属性的属性：

maps

用户可更新的映射列表。该列表从首次搜索到最后搜索排序。它是唯一存储的状态，可以修改以更改搜索的映射。该列表应始终包含至少一个映射。

new_child (*m* =*无*)

返回ChainMap包含新地图的新的地图，后面是当前实例中的所有地图。如果m指定，则它将成为映射列表前面的新映射；如果未指定，则使用空字典，以便调用 d.new_child() 相当于：。此方法用于创建可在不更改任何父映射中的值的情况下更新的子上下文。 ChainMap({}, *d.maps)

在版本3.4中更改： m添加了可选参数。

parents

属性返回一个ChainMap包含当前实例中除第一个之外的所有映射的新对象。这对于跳过搜索中的第一个地图很有用。用例与嵌套作用域中nonlocal使用的关键字类似 。用例还与内置函数的用例相平行 。引用相当于： 。 super() d.parents ChainMap(*d.maps[1:])

也可以看看:

- Enthought CodeTools包中的MultiContext类具有支持写入链中任何映射的选项。
- Django的 用于模板化的Context类是映射的只读链。它还具有类似于new_child()方法和 parents()属性的上下文推送和弹出 。
- 所述嵌套上下文配方具有选项来控制写入和其它突变是否只适用于第一映射或链中的任何映射。
- 一个大大简化的Chainmap只读版本。

8.3.1.1。ChainMap示例和食谱

本节介绍了使用链接地图的各种方法。

模拟Python内部查找链的示例：

```
import builtins
pylookup = ChainMap(locals(), globals(), vars(builtins))
```

让用户指定的命令行参数优先于优先于默认值的环境变量的示例：

```
import os, argparse

defaults = {'color': 'red', 'user': 'guest'}

parser = argparse.ArgumentParser()
parser.add_argument('-u', '--user')
parser.add_argument('-c', '--color')
namespace = parser.parse_args()
command_line_args = {k:v for k, v in vars(namespace).items() if v}

combined = ChainMap(command_line_args, os.environ, defaults)
```

```
print(combined['color'])
print(combined['user'])
```

使用ChainMap该类模拟嵌套上下文的示例模式：

```
c = ChainMap()          # Create root context
d = c.new_child()       # Create nested child context
e = c.new_child()       # Child of c, independent from d
e.maps[0]               # Current context dictionary -- like Python's locals()
e.maps[-1]              # Root context -- like Python's globals()
e.parents               # Enclosing context chain -- like Python's nonlocals

d['x']                  # Get first key in the chain of contexts
d['x'] = 1              # Set value in current context
del d['x']              # Delete from current context
list(d)                 # All nested values
k in d                  # Check all nested values
len(d)                  # Number of nested values
d.items()               # All nested items
dict(d)                 # Flatten into a regular dictionary
```

该ChainMap班不仅使更新（写入和删除），以在链中的第一个映射，同时查找将搜索上满链。但是，如果需要进行深入的写入和删除操作，则可以很容易地创建一个更新链中更深的键的子类：

```
class DeepChainMap(ChainMap):
    'Variant of ChainMap that allows direct updates to inner scopes'

    def __setitem__(self, key, value):
        for mapping in self.maps:
            if key in mapping:
                mapping[key] = value
                return
        self.maps[0][key] = value

    def __delitem__(self, key):
        for mapping in self.maps:
            if key in mapping:
                del mapping[key]
                return
        raise KeyError(key)

>>> d = DeepChainMap({'zebra': 'black'}, {'elephant': 'blue'}, {'lion': 'yellow'})
>>> d['lion'] = 'orange'         # update an existing key two levels down
>>> d['snake'] = 'red'           # new keys get added to the topmost dict
>>> del d['elephant']            # remove an existing key one level down
DeepChainMap({'zebra': 'black', 'snake': 'red'}, {}, {'lion': 'orange'})
```

8.3.2。Counter对象

提供计数器工具以支持方便快捷的计数。例如：

```
>>> # Tally occurrences of words in a list
>>> cnt = Counter()
```

```
>>> for word in ['red', 'blue', 'red', 'green', 'blue', 'blue']:
...     cnt[word] += 1
>>> cnt
Counter({'blue': 3, 'red': 2, 'green': 1})

>>> # Find the ten most common words in Hamlet
>>> import re
>>> words = re.findall(r'\w+', open('hamlet.txt').read().lower())
>>> Counter(words).most_common(10)
[('the', 1143), ('and', 966), ('to', 762), ('of', 669), ('i', 631),
 ('you', 554),  ('a', 546), ('my', 514), ('hamlet', 471), ('in', 451)]
```

class collections.Counter([*iterable-or-mapping*])

A Counter是dict用于计算可哈希对象的子类。它是一个无序的集合，其元素以字典键的形式存储，并将其计数存储为字典值。计数允许为包括零或负计数的任何整数值。该Counter课程类似于其他语言的书包或多媒体课程。

元素从一个*迭代*中计数或从另一个*映射*（或计数器）初始化：

```
>>> c = Counter()                           # a new, empty counter
>>> c = Counter('gallahad')                 # a new counter from an iterable
>>> c = Counter({'red': 4, 'blue': 2})      # a new counter from a mapping
>>> c = Counter(cats=4, dogs=8)             # a new counter from keyword args
```

计数器对象有一个字典界面，只不过它们会为丢失的项目返回零计数，而不是引发KeyError：

```
>>> c = Counter(['eggs', 'ham'])
>>> c['bacon']                              # count of a missing element is zero
0
```

将计数设置为零不会从计数器中删除元素。使用del完全删除它：

```
>>> c['sausage'] = 0                        # counter entry with a zero count
>>> del c['sausage']                        # del actually removes the entry
```

版本3.1中的新功能。

Counter对象支持超出所有字典可用的三种方法：

elements( )

将元素返回一个迭代器，每次重复的次数与它的次数相同。元素以任意顺序返回。如果一个元素的数量少于一个，elements()将忽略它。

```
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> sorted(c.elements())
['a', 'a', 'a', 'a', 'b', 'b']
```

most_common([*n*])

从最常见到最不常见的列表中，列出*n*个最常见的元素及其数量。如果省略*n*None，则most_common()返回计数器中的*所有*元素。具有相同计数的元素可以任意排序：

```
>>> Counter('abracadabra').most_common(3)
[('a', 5), ('r', 2), ('b', 2)]
```

subtract ([*iterable-or-mapping*])

元素从*可迭代*或从另一个*映射*（或计数器）中减去。像dict.update()但减去计数而不是替换它们。输入和输出都可以是零或负数。

```
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> d = Counter(a=1, b=2, c=3, d=4)
>>> c.subtract(d)
>>> c
Counter({'a': 3, 'b': 0, 'c': -3, 'd': -6})
```

3.2版本中的新功能

通常的字典方法可用于Counter对象，但对于计数器的工作方式不同。

fromkeys (*可迭代*)

这个类方法没有为Counter对象实现。

update ([*iterable-or-mapping*])

元素从一个*迭代*中计数或从另一个 *映射*（或计数器）加入。像dict.update()但增加了计数而不是替换它们。此外，*可迭代*预计是一系列元素，而不是一系列对。(key, value)

处理Counter对象的常见模式：

```
sum(c.values())                 # total of all counts
c.clear()                       # reset all counts
list(c)                         # list unique elements
set(c)                          # convert to a set
dict(c)                         # convert to a regular dictionary
c.items()                       # convert to a list of (elem, cnt) pairs
Counter(dict(list_of_pairs))    # convert from a list of (elem, cnt) pairs
c.most_common()[:-n-1:-1]       # n least common elements
+c                              # remove zero and negative counts
```

提供了几个数学运算来组合Counter 对象以产生多集（具有大于零的计数的计数器）。加法和减法通过加或减相应元素的计数来组合计数器。相交和联合会返回相应计数的最小值和最大值。每个操作都可以接受带符号计数的输入，但输出将排除计数为零或更小的结果。

```
>>> c = Counter(a=3, b=1)
>>> d = Counter(a=1, b=2)
>>> c + d                       # add two counters together:  c[x] + d[x]
Counter({'a': 4, 'b': 3})
>>> c - d                       # subtract (keeping only positive counts)
Counter({'a': 2})
>>> c & d                       # intersection:  min(c[x], d[x])
Counter({'a': 1, 'b': 1})
>>> c | d                       # union:  max(c[x], d[x])
Counter({'a': 3, 'b': 2})
```

一元加法和减法是添加空白计数器或从空白计数器中减去的快捷方式。

```
>>> c = Counter(a=2, b=-4)
>>> +c
Counter({'a': 2})
>>> -c
Counter({'b': 4})
```

*3.3版*新增功能：增加了对一元加号，一元减号和就地多重集操作的支持。

注意： 计数器主要设计为用积极整数来表示运行计数; 但是，注意不要不必要地排除需要其他类型或负值的用例。为了帮助解决这些用例，本节介绍了最小范围和类型限制。

- 在Counter类本身是一本字典的子类，在其键和值没有任何限制。这些值旨在表示计数的数字，但您*可以*在值字段中存储任何内容。
- 该most_common()方法只要求值是可订购的。
- 对于就地操作，例如，值类型只需要支持加法和减法。所以分数，浮点数和小数将会起作用并且负值被支持。同样也是如此 ，并允许负和用于输入和输出的零个值。 c[key] += 1update() subtract()
- multiset方法仅用于具有正值的用例。输入可能为负值或零，但只能创建正值的输出。没有类型限制，但值类型需要支持加法，减法和比较。
- 该elements()方法需要整数计数。它忽略零和负数。

也可以看看:

- Smalltalk中的Bag类。
- 维基百科进入Multisets。
- 带有示例的C ++ multisets教程。
- 有关multisets及其用例的数学运算，请参阅 *Knuth，Donald。计算机编程艺术II卷4.6.3节练习19*。
- 要枚举给定元素集上给定大小的所有不同多重集，请参阅itertools.combinations_with_replacement()：

  map（Counter，combination_with_replacement（'ABC'，2））- > AA AB AC BB BC CC

8.3.3。deque对象

class collections.deque（[*iterable* [，*maxlen*]]）

使用append()来自*iterable的*数据从左到右地初始化（使用）返回一个新的deque对象。如果未指定*迭代*，则新的双端队列为空。

Deques是堆栈和队列的泛化（名称发音为“deck”，是“双端队列”的缩写）。Deques支持线程安全，高效的内存追加，并从双侧出现，并且在任一方向都具有大致相同的O（1）性能。

尽管list对象支持类似的操作，但它们针对快速固定长度操作进行了优化，并且会产生O（n）内存移动成本pop(0)和操作，这些成本 会改变底层数据表示的大小和位置。insert(0, v)

如果未指定*maxlen*或Nonedeque，则deques可能增长到任意长度。否则，deque被限制到指定的最大长度。一旦有限长度的deque已满，当添加新项目时，相应数量的项目将从相反的一端被丢弃。有界长度deques提供的功能类似于tailUnix中的过滤器。它们也可用于跟踪只有最近的活动感兴趣的交易和其他数据池。

Deque对象支持以下方法：

append(*x*)
将*x*添加到双端队列的右侧。

appendleft(*x*)
将*x*添加到双端队列的左侧。

clear()
删除deque中的所有元素，使其长度为0。

copy()
创建一个deque的浅表副本。

3.5版本中的新功能。

count(*x*)
计算deque元素的数量等于*x*。

3.2版本中的新功能

extend(*可迭代*)
通过追加iterable参数中的元素来扩展双端队列的右侧。

extendleft(*可迭代*)
通过附加*iterable中的*元素来扩展双端队列的左侧。请注意，一系列左边附加结果颠倒了迭代参数中元素的顺序。

index(*x*[, *start*[, *stop*]])
返回deque中的*x*的位置（在索引*开始*处或 索引*停止*之前）。返回第一个匹配项，ValueError如果找不到则返回 。

3.5版本中的新功能。

insert(*i*, *x*)
将*x*插入位置*i*处的deque 。

如果插入会导致一个有界的deque增长超过*maxlen*， IndexError则会引发一个。

3.5版本中的新功能。

pop ( )

从deque的右侧移除并返回一个元素。如果没有元素存在，则引发一个IndexError。

popleft ( )

从deque的左侧移除并返回一个元素。如果没有元素存在，则引发一个IndexError。

remove (*价值*)

删除第一次出现的*值*。如果找不到，就提出一个 ValueError。

reverse ( )

在原地颠倒deque的元素，然后返回None。

3.2版本中的新功能

rotate (*n* = *1*)

向右旋转deque *n*个步骤。如果*n*为负数，则向左旋转。

当deque不空时，向右d.appendleft(d.pop())旋转一步相当于向左旋转一步相当于d.append(d.popleft())。

Deque对象还提供一个只读属性：

maxlen

一个deque的最大大小或者None是否是无限的。

版本3.1中的新功能。

除上述外，双端支持迭代，酸洗，len(d) ， reversed(d) ，copy.copy(d) ，copy.deepcopy(d) ，会员与测试in运营商，和下标引用，如d[-1]。索引访问在两端都是O（1），但在中间减慢到O（n）。对于快速随机访问，请使用列表。

在3.5版本开始，双端的支持__add__() ，__mul__()和__imul__()。

例：

```
>>> from collections import deque
>>> d = deque('ghi')                 # make a new deque with three items
>>> for elem in d:                   # iterate over the deque's elements
...     print(elem.upper())
G
H
I

>>> d.append('j')                    # add a new entry to the right side
>>> d.appendleft('f')                # add a new entry to the left side
>>> d                                # show the representation of the deque
deque(['f', 'g', 'h', 'i', 'j'])

>>> d.pop()                          # return and remove the rightmost item
'j'
>>> d.popleft()                      # return and remove the leftmost item
'f'
>>> list(d)                          # list the contents of the deque
```

```
['g', 'h', 'i']
>>> d[0]                            # peek at leftmost item
'g'
>>> d[-1]                           # peek at rightmost item
'i'

>>> list(reversed(d))               # list the contents of a deque in reverse
['i', 'h', 'g']
>>> 'h' in d                        # search the deque
True
>>> d.extend('jkl')                 # add multiple elements at once
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])
>>> d.rotate(1)                     # right rotation
>>> d
deque(['l', 'g', 'h', 'i', 'j', 'k'])
>>> d.rotate(-1)                    # left rotation
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])

>>> deque(reversed(d))              # make a new deque in reverse order
deque(['l', 'k', 'j', 'i', 'h', 'g'])
>>> d.clear()                       # empty the deque
>>> d.pop()                         # cannot pop from an empty deque
Traceback (most recent call last):
    File "<pyshell#6>", line 1, in -toplevel-
        d.pop()
IndexError: pop from an empty deque

>>> d.extendleft('abc')             # extendleft() reverses the input order
>>> d
deque(['c', 'b', 'a'])
```

8.3.3.1。deque食谱

本节介绍与deques合作的各种方法。

有界长度的deques提供了类似于`tail`Unix中的过滤器的功能：

```
def tail(filename, n=10):
    'Return the last n lines of a file'
    with open(filename) as f:
        return deque(f, n)
```

另一种使用deques的方法是通过附加到右边并弹出到左边来维护一系列最近添加的元素：

```
def moving_average(iterable, n=3):
    # moving_average([40, 30, 50, 46, 39, 44]) --> 40.0 42.0 45.0 43.0
    # http://en.wikipedia.org/wiki/Moving_average
    it = iter(iterable)
    d = deque(itertools.islice(it, n-1))
    d.appendleft(0)
    s = sum(d)
    for elem in it:
        s += elem - d.popleft()
```

```
        d.append(elem)
        yield s / n
```

该rotate()方法提供了一种实现deque切片和删除的方法。例如，一个纯粹的Python实现依赖于该方法来定位要弹出的元素：del d[n]rotate()

```
def delete_nth(d, n):
    d.rotate(-n)
    d.popleft()
    d.rotate(n)
```

要实施deque切片，请使用类似的方法 rotate()来将目标元素放置在双侧的左侧。删除旧条目popleft()，添加新条目extend()，然后反转。随着对这种做法的微小变化，很容易实现第四风格栈操作，如dup，drop，swap，over，pick， rot，和roll。

8.3.4。defaultdict对象

class collections.**defaultdict**([*default_factory* [, ...]])

返回一个新的类似字典的对象。 defaultdict是内置类的一个子dict类。它覆盖一个方法并添加一个可写实例变量。其余的功能与dict该类相同， 不在此处记载。

第一个参数提供了default_factory 属性的初始值; 它默认为None。所有其余参数的处理方式与将它们传递给dict构造函数（包括关键字参数）相同。

defaultdict除了标准dict操作外，对象还支持以下方法：

__missing__(*重点*)

如果该default_factory属性是None，则会引发一个KeyError以*键*为参数的 异常。

如果default_factory不是None，则调用不带参数来为给定*键*提供默认值，将此值插入到*键*的字典中并返回。

如果调用default_factory引发异常，则此异常将传播不变。

当未找到请求的密钥时，该__getitem__()方法由dict类的方法 调用; 无论它返回还是提出，然后返回或提出__getitem__()。

请注意，__missing__()是*不*要求任何操作之外 __getitem__()。这意味着get()，与普通字典一样，它将None作为默认返回而不是使用 default_factory。

defaultdict 对象支持以下实例变量：

default_factory

该属性被该__missing__()方法使用; 它从第一个参数初始化到构造函数，如果存在的话，或者 None如果没有的话。

8.3.4.1。defaultdict示例

使用list这种方式default_factory，很容易将一系列键值对组合成一个列表字典：

```
>>> s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]
>>> d = defaultdict(list)
>>> for k, v in s:
...     d[k].append(v)
...
>>> sorted(d.items())
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

当第一次遇到每个密钥时，它不在映射中; 所以使用default_factory 返回空的函数自动创建一个条目list。 list.append() 然后该操作将该值附加到新列表中。当再次遇到密钥时，查找正常进行（返回该密钥的列表），并且该 list.append() 操作向列表添加另一个值。这种技术比使用dict.setdefault()以下技术的等效技术更简单快捷：

```
>>> d = {}
>>> for k, v in s:
...     d.setdefault(k, []).append(v)
...
>>> sorted(d.items())
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

设置default_factory以int使 defaultdict有用的计数（如其他语言包或多重集）：

```
>>> s = 'mississippi'
>>> d = defaultdict(int)
>>> for k in s:
...     d[k] += 1
...
>>> sorted(d.items())
[('i', 4), ('m', 1), ('p', 2), ('s', 4)]
```

当第一次遇到字母时，它从映射中丢失，所以 default_factory函数调用int()默认计数为零。增量操作然后为每个字母建立计数。

int()总是返回零的函数只是常数函数的特例。创建常量函数的更快更灵活的方法是使用可以提供任何常量值（不仅仅是零）的lambda函数：

```
>>> def constant_factory(value):
...     return lambda: value
>>> d = defaultdict(constant_factory('<missing>'))
>>> d.update(name='John', action='ran')
>>> '%(name)s %(action)s to %(object)s' % d
'John ran to <missing>'
```

设置default_factory为set使 defaultdict建立集合的字典有用：

```
>>> s = [('red', 1), ('blue', 2), ('red', 3), ('blue', 4), ('red', 1), ('blue', 4)]
>>> d = defaultdict(set)
>>> for k, v in s:
...     d[k].add(v)
...
>>> sorted(d.items())
[('blue', {2, 4}), ('red', {1, 3})]
```

8.3.5。namedtuple()具有命名字段的元组的工厂函数

命名元组赋予元组中每个位置的含义并允许更具可读性的自编写代码。它们可以在任何使用常规元组的地方使用，并且可以通过名称而不是位置索引来添加字段。

collections.namedtuple（*typename*，*field_names*，*，*verbose = False*，*rename = False*，*module = None* ）

返回一个名为*typename*的新的元组子类。新的子类用于创建具有可通过属性查找访问的字段的类似元组的对象，以及可索引和可迭代的对象。子类的实例也有一个有用的docstring（带有typename和field_names）和一个有用的__repr__()方法，它以一种name=value格式列出元组内容。

的*FIELD_NAMES*是字符串如序列。或者，*field_names*可以是单个字符串，每个字段名用空格和/或逗号分隔，例如或。['x', 'y']'x y''x, y'

任何有效的Python标识符都可以用于字段名称，除了以下划线开头的名称。有效标识符由字母，数字和下划线，但不以数字或下划线开始，不能是keyword诸如*类*，*对*，*回报*，*全球*，*通过*，或*提高*。

如果*重命名*为true，则无效的字段名会自动替换为位置名称。例如，转换为，消除关键字和重复的字段名称。['abc', 'def', 'ghi', 'abc']['abc', '_1', 'ghi', '_3']defabc

如果*verbose*为true，则在构建它之后打印类定义。此选项已过时; 相反，打印_source属性更简单。

如果定义了*模块*__module__，则指定元组的属性将设置为该值。

命名元组实例没有每个实例字典，因此它们是轻量级的，并且不需要比常规元组更多的内存。

*在版本3.1中更改：*添加了对*重命名的*支持。

*改变在3.6版本：*在*冗长*和*重命名*参数成为 唯一的关键字参数。

*在版本3.6中更改：*添加了*模块*参数。

```
>>> # Basic example
>>> Point = namedtuple('Point', ['x', 'y'])
>>> p = Point(11, y=22)     # instantiate with positional or keyword arguments
>>> p[0] + p[1]             # indexable like the plain tuple (11, 22)
33
>>> x, y = p                # unpack like a regular tuple
>>> x, y
(11, 22)
>>> p.x + p.y               # fields also accessible by name
33
>>> p                       # readable __repr__ with a name=value style
Point(x=11, y=22)
```

命名元组对于将字段名称分配给由csv或sqlite3模块返回的结果元组特别有用：

```
EmployeeRecord = namedtuple('EmployeeRecord', 'name, age, title, department, paygrade'

import csv
for emp in map(EmployeeRecord._make, csv.reader(open("employees.csv", "rb"))):
    print(emp.name, emp.title)

import sqlite3
conn = sqlite3.connect('/companydata')
cursor = conn.cursor()
cursor.execute('SELECT name, age, title, department, paygrade FROM employees')
for emp in map(EmployeeRecord._make, cursor.fetchall()):
    print(emp.name, emp.title)
```

除了从元组继承的方法外，命名元组还支持三个附加方法和两个属性。为防止与字段名冲突，方法和属性名称以下划线开头。

classmethod somenamedtuple._make(*iterable*)

从现有序列创建新实例或迭代的类方法。

```
>>> t = [11, 22]
>>> Point._make(t)
Point(x=11, y=22)
```

somenamedtuple._asdict( )

返回一个新的OrderedDict映射字段名称到它们相应的值：

```
>>> p = Point(x=11, y=22)
>>> p._asdict()
OrderedDict([('x', 11), ('y', 22)])
```

*在版本3.1中更改：*返回OrderedDict而不是常规dict。

somenamedtuple._replace(** *kwargs*)

返回指定元组的新实例，用新值替换指定字段：

```
>>> p = Point(x=11, y=22)
>>> p._replace(x=33)
Point(x=33, y=22)

>>> for partnum, record in inventory.items():
...     inventory[partnum] = record._replace(price=newprices[partnum], timestamp=
```

somenamedtuple._source

一个包含纯Python源代码的字符串，用于创建指定的元组类。源使命名的元组自我记录。它可以打印，使用exec()或执行，或保存到文件并导入。

3.3版本的新功能

somenamedtuple._fields

列出字段名称的字符串元组。用于内省和从现有命名元组中创建新的命名元组类型。

```
>>> p._fields            # view the field names
('x', 'y')

>>> Color = namedtuple('Color', 'red green blue')
>>> Pixel = namedtuple('Pixel', Point._fields + Color._fields)
>>> Pixel(11, 22, 128, 255, 0)
Pixel(x=11, y=22, red=128, green=255, blue=0)
```

要检索名称存储在字符串中的字段，请使用以下getattr() 函数：

```
>>> getattr(p, 'x')
11
```

要将字典转换为命名元组，请使用双星运算符（如解包参数列表中所述）：

```
>>> d = {'x': 11, 'y': 22}
>>> Point(**d)
Point(x=11, y=22)
```

由于命名元组是一个常规的Python类，因此使用子类很容易添加或更改功能。以下是如何添加计算字段和固定宽度的打印格式：

```
>>> class Point(namedtuple('Point', ['x', 'y'])):
...     __slots__ = ()
...     @property
...     def hypot(self):
...         return (self.x ** 2 + self.y ** 2) ** 0.5
...     def __str__(self):
...         return 'Point: x=%6.3f  y=%6.3f  hypot=%6.3f' % (self.x, self.y, self.hyp

>>> for p in Point(3, 4), Point(14, 5/7):
...     print(p)
Point: x= 3.000  y= 4.000  hypot= 5.000
Point: x=14.000  y= 0.714  hypot=14.018
```

上面显示的子类设置__slots__为空元组。这有助于通过防止创建实例字典来降低内存需求。

子类化对于添加新的存储字段没有用处。相反，只需从_fields属性创建一个新的命名元组类型：

```
>>> Point3D = namedtuple('Point3D', Point._fields + ('z',))
```

可以通过直接分配__doc__ 字段来自定义文档字符串：

```
>>> Book = namedtuple('Book', ['id', 'title', 'authors'])
>>> Book.__doc__ += ': Hardcover book in active collection'
>>> Book.id.__doc__ = '13-digit ISBN'
>>> Book.title.__doc__ = 'Title of first printing'
>>> Book.authors.__doc__ = 'List of authors sorted by last name'
```

*版本3.5中更改：*属性文档字符串变为可写。

默认值可以通过使用_replace()定制原型实例来实现：

```
>>> Account = namedtuple('Account', 'owner balance transaction_count')
>>> default_account = Account('<owner name>', 0.0, 0)
>>> johns_account = default_account._replace(owner='John')
>>> janes_account = default_account._replace(owner='Jane')
```

也可以看看:

- 指定元组的抽象基类和 Jan Kaliszewski 的元类混合。除了为命名元组提供抽象基类之外，它还支持一个替代的基于元类的构造函数，它便于在命名元组被子类化的用例中使用。
- 查看types.SimpleNamespace()基于底层字典而不是元组的可变名称空间。
- 查看有关typing.NamedTuple()为命名元组添加类型提示的方法。

8.3.6。OrderedDict对象

有序字典就像普通字典一样，但他们记得插入项目的顺序。在迭代有序字典时，将按照首次添加密钥的顺序返回项目。

class collections.OrderedDict([*items*])

返回一个字典子类的实例，支持通常的dict 方法。一个*OrderedDict*是记住的键首先被插入的顺序的字典。如果新条目覆盖现有条目，则原始插入位置保持不变。删除一个条目并重新插入它将会把它移到最后。

版本3.1中的新功能。

popitem(*last = True*)

popitem()有序字典的方法返回并删除（键，值）对。如果*最后一个*为真，则按LIFO顺序返回对，否则按FIFO顺序返回 。

move_to_end(*key*, *last = True*)

将现有*密钥*移至有序字典的末尾。如果*最后一个*为真（默认），则项目移动到右侧，如果*最后一个*为假，则移动到开始。KeyError如果*密钥*不存在则引发：

```
>>> d = OrderedDict.fromkeys('abcde')
>>> d.move_to_end('b')
>>> ''.join(d.keys())
'acdeb'
>>> d.move_to_end('b', last=False)
>>> ''.join(d.keys())
'bacde'
```

3.2版本中的新功能

除了通常的映射方法外，有序字典还支持使用反向迭代reversed()。

OrderedDict 对象之间的平等测试是顺序敏感的，并按照实现list(od1.items())==list(od2.items())。OrderedDict对象和其他 Mapping对象之间的平等测

试与常规字典一样对命令不敏感。这允许OrderedDict在使用常规字典的任何地方替换对象。

*改变在3.5版本：*该物品，钥匙，和值的观点 的OrderedDict ，现在支持使用反向迭代reversed()。

*在版本3.6中更改：*随着接受**PEP 468**，保留传递给OrderedDict构造函数及其update() 方法的关键字参数的顺序。

8.3.6.1。OrderedDict示例和食谱

由于有序字典会记住它的插入顺序，因此它可以与排序结合使用以创建一个排序后的字典：

```
>>> # regular unsorted dictionary
>>> d = {'banana': 3, 'apple': 4, 'pear': 1, 'orange': 2}

>>> # dictionary sorted by key
>>> OrderedDict(sorted(d.items(), key=lambda t: t[0]))
OrderedDict([('apple', 4), ('banana', 3), ('orange', 2), ('pear', 1)])

>>> # dictionary sorted by value
>>> OrderedDict(sorted(d.items(), key=lambda t: t[1]))
OrderedDict([('pear', 1), ('orange', 2), ('banana', 3), ('apple', 4)])

>>> # dictionary sorted by length of the key string
>>> OrderedDict(sorted(d.items(), key=lambda t: len(t[0])))
OrderedDict([('pear', 1), ('apple', 4), ('orange', 2), ('banana', 3)])
```

删除条目时，新排序的字典会保持排序顺序。但是，当添加新的密钥时，密钥被追加到最后，并且不保持排序。

创建一个有序的字典变体也是直接的，可以记住键*最后*插入的顺序。如果新条目覆盖现有条目，则会更改原始插入位置并移至最后：

```
class LastUpdatedOrderedDict(OrderedDict):
    'Store items in the order the keys were last added'

    def __setitem__(self, key, value):
        if key in self:
            del self[key]
        OrderedDict.__setitem__(self, key, value)
```

一个有序的字典可以和这个Counter类结合起来，这样计数器就能记住第一次遇到的顺序元素：

```
class OrderedCounter(Counter, OrderedDict):
    'Counter that remembers the order elements are first encountered'

    def __repr__(self):
        return '%s(%r)' % (self.__class__.__name__, OrderedDict(self))

    def __reduce__(self):
        return self.__class__, (OrderedDict(self),)
```

8.3.7。UserDict对象

该类UserDict充当字典对象的包装。这个类的需要已经被直接子类的能力部分地取代了dict; 然而，这个类可以更容易处理，因为底层字典可以作为属性访问。

class collections.UserDict([*initialdata*])

模拟字典的类。实例的内容保存在一个常规的字典中，通过实例的data属性 可以访问该字典UserDict。如果提供*initialdata*， data则用其内容进行初始化; 请注意，对*initialdata*的引用将不会保留，允许将其用于其他目的。

除了支持映射的方法和操作外， UserDict实例还提供以下属性：

data

用于存储UserDict 课程内容的真正词典。

8.3.8。UserList对象

这个类充当列表对象的包装。对于自己的类列表类来说，它是一个有用的基类，可以从它们继承，并覆盖现有的方法或添加新的方法。通过这种方式，可以将新行为添加到列表中。

这个类的需要已经被直接子类的能力部分地取代了list; 然而，这个类可以更容易处理，因为底层列表可以作为属性访问。

class collections.UserList([*list*])

模拟列表的类。实例的内容保存在一个常规列表中，该列表可通过 实例的data属性访问UserList。实例的内容最初设置为*列表*副本，默认为空列表[]。 *列表*可以是任何可迭代的，例如真正的Python列表或UserList对象。

除了支持可变序列的方法和操作外， UserList实例还提供以下属性：

data

list用于存储UserList课程内容的真实对象 。

子类化要求：子类需要 UserList提供一个构造函数，可以使用无参数或一个参数来调用构造函数。返回新序列的列表操作尝试创建实际实现类的实例。为此，它假定构造函数可以用一个参数来调用，该参数是一个用作数据源的序列对象。

如果一个派生类不希望遵守这个要求，那么这个类支持的所有特殊方法都需要被覆盖; 请咨询消息来源，了解有关在这种情况下需要提供的方法的信息。

8.3.9。UserString对象

该类UserString充当字符串对象的包装。这个类的需要已经被直接子类的能力部分地取代了str; 然而，这个类可以更容易处理，因为底层字符串可以作为属性访问。

class collections.UserString([*sequence*])

模拟字符串或Unicode字符串对象的类。实例的内容保存在一个普通的字符串对象中，通过实例的data属性可以访问该对象 UserString。实例的内容最初设置为*序列*的副本。该*序列*可以是，，（或子类）的一个实例bytes，或者可以使用内置函数转换为字符串的任意序列。 strUserStringstr()

*改变在3.5版本：*新的方法 __getnewargs__ ， __rmod__ ， casefold ， format_map ， isprintable，和maketrans。

8.4。collections.abc-抽象基类的容器

*3.3版本的新功能：*以前，这个模块是模块的一部分collections。

源代码： Lib / _collections_abc.py

该模块提供了可用于测试某个类是否提供特定接口的抽象基类；例如，它是否可散列或者是否是映射。

8.4.1。集合抽象基类

收藏夹模块提供了以下ABC：

ABC	继承	抽象方法	混合方法
Container		__contains__	
Hashable		__hash__	
Iterable		__iter__	
Iterator	Iterable	__next__	__iter__
Reversible	Iterable	__reversed__	
Generator	Iterator	send , throw	close , __iter__ , __next__
Sized		__len__	
Callable		__call__	
Collection	Sized , Iterable , Container	__contains__ , __iter__ , __len__	
Sequence	Reversible , Collection	__getitem__ , __len__	__contains__ , __iter__ , __reversed__ , index , 和count
MutableSequence	Sequence	__getitem__ , __setitem__ , __delitem__ , __len__ , insert	继承Sequence方法和 append , reverse , extend , pop , remove , 和__iadd__
ByteString	Sequence	__getitem__ , __len__	继承的Sequence方法
Set	Collection	__contains__ , __iter__ , __len__	__le__ , __lt__ , __eq__ , __ne__ , __gt__ , __ge__ , __and__ , __or__ , __sub__ , __xor__ , 和isdisjoint
MutableSet	Set	__contains__ , __iter__ , __len__ , add , discard	继承Set方法和 clear , pop , remove , __ior__ , __iand__ , __ixor__ , 和 __isub__
Mapping	Collection	__getitem__ , __iter__ , __len__	__contains__ , keys , items , values , get , __eq__ , 和__ne__

ABC	继承	抽象方法	混合方法
MutableMapping	Mapping	__getitem__ , __setitem__ , __delitem__ , __iter__ , __len__	继承Mapping方法和 pop , popitem , clear , update , 和setdefault
MappingView	Sized		__len__
ItemsView	MappingView , Set		__contains__ , __iter__
KeysView	MappingView , Set		__contains__ , __iter__
ValuesView	MappingView		__contains__ , __iter__
Awaitable		__await__	
Coroutine	Awaitable	send , throw	close
AsyncIterable		__aiter__	
AsyncIterator	AsyncIterable	__anext__	__aiter__
AsyncGenerator	AsyncIterator	asend , athrow	aclose , __aiter__ , __anext__

*类*collections.abc.Container
*类*collections.abc.Hashable
*类*collections.abc.Sized
*类*collections.abc.Callable
对于类的ABC分别提供的方法__contains__() , __hash__() , __len__() ,和__call__()。

*类*collections.abc.Iterable
ABC提供了该__iter__()方法的类。

检查检测到已注册为或有方法的类，但未检测到与该方法重复的类。确定对象是否可迭代的唯一可靠方法是调用。isinstance(obj, Iterable)Iterable__iter__()__getitem__()iter(obj)

*类*collections.abc.Collection
用于大小可迭代的容器类的ABC。

3.6版本中的新功能。

*类*collections.abc.Iterator
ABC提供的类__iter__()和 __next__()方法。另请参阅迭代器的定义 。

*类*collections.abc.Reversible
用于迭代类的ABC也提供了该__reversed__() 方法。

3.6版本中的新功能。

*类*collections.abc.Generator
用于实现在中定义的协议的生成器类的ABC **PEP 342**延伸与所述迭代器send() , throw()和close()方法。另见发电机的定义。

3.5版本中的新功能。

*类*collections.abc.Sequence
*类*collections.abc.MutableSequence
*类*collections.abc.ByteString
ABCs为只读和可变序列。

实现注释：一些mixin方法，例如 `__iter__()` ，`__reversed__()` 和 `index()` 重复调用基础 `__getitem__()` 方法。因此，如果 `__getitem__()` 以不变的访问速度实现，mixin方法将具有线性性能; 但是，如果底层方法是线性的（就像链接列表一样），mixins将具有二次性能，并且可能需要被覆盖。

版本3.5中已更改： index（）方法添加了对*停止*和*启动*参数的支持。

*类*collections.abc.Set
*类*collections.abc.MutableSet
用于只读和可变集的ABCs。

*类*collections.abc.Mapping
*类*collections.abc.MutableMapping
ABCs为只读和可变映射。

*类*collections.abc.MappingView
*类*collections.abc.ItemsView
*类*collections.abc.KeysView
*类*collections.abc.ValuesView
用于映射的ABCs，项目，键和值视图。

*类*collections.abc.Awaitable
ABC用于等待对象，可用于 `await` 表达式中。自定义实现必须提供该 `__await__()` 方法。

协程对象和 `Coroutine` ABC的实例都是这个ABC的实例。

> **注意：** 在CPython中，即使没有方法，基于生成器的协程（装有 `types.coroutine()` 或者的生成器 `asyncio.coroutine()` ）也是可以 *等待* 的 `__await__()` 。为他们使用将返回。使用探测到它们。 `isinstance(gencoro, Awaitable)` `False` `inspect.isawaitable()`

3.5版本中的新功能。

*类*collections.abc.Coroutine
用于协同兼容类的ABC。这些实现以下方法，在定义协程对象： `send()` ，`throw()` ，和 `close()` 。自定义实现也必须实现 `__await__()` 。所有 `Coroutine` 实例也是实例 `Awaitable` 。另请参阅协程的定义。

> **注意：** 在CPython中，即使没有方法，基于生成器的协程（装有 `types.coroutine()` 或者的生成器 `asyncio.coroutine()` ）也是可以 *等待* 的 `__await__()` 。为他们使用将返回。使用探测到它们。 `isinstance(gencoro, Coroutine)` `False` `inspect.isawaitable()`

3.5版本中的新功能。

*类*collections.abc.AsyncIterable

ABC提供__aiter__方法的类。另请参见异步可迭代的定义。

3.5版本中的新功能。

*类*collections.abc.AsyncIterator

ABC提供的类__aiter__和__anext__ 方法。另请参阅异步迭代器的定义。

3.5版本中的新功能。

*类*collections.abc.AsyncGenerator

用于实现定义在协议中的异步生成器类的ABC **PEP 525**和**PEP 492**。

3.6版本中的新功能。

这些ABCs允许我们询问类或实例是否提供特定的功能，例如：

```
size = None
if isinstance(myvar, collections.abc.Sized):
    size = len(myvar)
```

一些ABCs也可以用作mixin，从而更容易开发支持容器API的类。例如，为了写一个类支持所有Set API，它仅需要提供三个底层抽象方法：__contains__()，__iter__()，和__len__()。美国广播公司提供其余的方法，如__and__()和 isdisjoint()：

```
class ListBasedSet(collections.abc.Set):
    ''' Alternate set implementation favoring space over speed
        and not requiring the set elements to be hashable. '''
    def __init__(self, iterable):
        self.elements = lst = []
        for value in iterable:
            if value not in lst:
                lst.append(value)

    def __iter__(self):
        return iter(self.elements)

    def __contains__(self, value):
        return value in self.elements

    def __len__(self):
        return len(self.elements)

s1 = ListBasedSet('abcdef')
s2 = ListBasedSet('defghi')
overlap = s1 & s2            # The __and__() method is supported automatically
```

关于使用Set和MutableSet作为mixin的注意事项：

1. 由于某些集合操作会创建新集合，因此默认的mixin方法需要通过迭代来创建新实例。假设类构造函数在表单中有一个签名ClassName(iterable)。这个假设因素等，以所谓的内部类方法_from_iterable()这就要求cls(iterable)制作了一套新的。如果Setmixin在具有不同构造函数签名的类中使用，则需要_from_iterable() 用可以从可迭代参数构造新实例的类方法重写。
2. 要覆盖比较（据推测为速度，因为语义是固定的），重新定义__le__()和__ge__()，则其他操作会自动效仿。

3. 的Set混入提供了一种_hash()计算用于该组的散列值的方法; 然而，__hash__()没有定义，因为并不是所有的集合都是可散列的或不可变的。要使用混入添加集hashability，来自继承Set()和Hashable()，然后再定义。__hash__ = Set._hash

也可以看看:

- OrderedSet配方为基础构建的示例MutableSet。
- 有关ABC的更多信息，请参阅abc模块和**PEP 3119**。

8.5。 heapq- 堆队列算法

源代码： Lib / heapq.py

该模块提供了堆队列算法的实现，也称为优先级队列算法。

堆是二叉树，每个父节点的值小于或等于其任何子节点。该实现使用数组 ，对于所有的*k*，从零开始计数元素。为了比较，不存在的元素被认为是无限的。堆的有趣属性是它的最小元素始终是根，。 heap[k] <= heap[2*k+1]heap[k] <= heap[2*k+2]heap[0]

下面的API与教科书堆算法在两个方面不同：（a）我们使用从零开始的索引。这使得节点的索引与其子节点的索引之间的关系略微不明显，但是由于Python使用了从零开始的索引，因此更合适。（b）我们的弹出方法返回最小的项目，而不是最大的项目（在教科书中称为“最小堆”;由于其适用于原地排序，因此“最大堆”在文本中更常见）。

这两个可以将堆看作一个常规的Python列表而不出意外：heap[0]是最小的项目，并且heap.sort()保持堆不变！

要创建堆，请使用初始化为的列表[]，或者您可以通过函数将填充列表转换为堆heapify()。

提供以下功能：

heapq.heappush（*堆*，*项目*）
将值*项*推入*堆中*，保持堆不变。

heapq.heappop（*堆*）
弹出并返回*堆中*的最小项，保持堆不变。如果堆是空的，IndexError则升起。要访问最小的物品而不弹出它，请使用heap[0]。

heapq.heappushpop（*堆*，*项目*）
按下堆上的*项目*，然后弹出并从*堆中*返回最小的项目 。联合行动的运行效率比heappush()单独呼叫后的效率要高heappop()。

heapq.heapify（*x*）
在线性时间内将列表*x*转换为堆。

heapq.heapreplace（*堆*，*项目*）
弹出并返回*堆中*最小的项目，并推送新*项目*。堆大小不会改变。如果堆是空的，IndexError则升起。

这一步操作比heappop()后面的 步骤更有效heappush()，在使用固定大小的堆时更为合适。pop / push组合总是从堆中返回一个元素，并用*item*替换它。

返回的值可能会大于添加的*项目*。如果不需要，请考虑使用heappushpop()。它的push / pop组合返回两个值中较小的一个，在堆上留下较大的值。

该模块还提供了三种基于堆的通用功能。

heapq.**merge**(**iterables*, *key = None*, *reverse = False*)

将多个排序后的输入合并到一个排序后的输出中（例如，合并来自多个日志文件的时间戳条目）。返回 排序后的值的迭代器。

与sorted(itertools.chain(*iterables))返回iterable 类似，不会一次将数据拉入内存，并假定每个输入流已经排序（从最小到最大）。

有两个可选参数，必须将其指定为关键字参数。

*key*指定用于从每个输入元素中提取比较键的一个参数的关键函数。默认值是 None（直接比较元素）。

*reverse*是一个布尔值。如果设置为True，则将输入元素合并，就好像每个比较都是相反的。

*在版本3.5中更改：*添加了可选的*密钥*和*反向*参数。

heapq.**nlargest**(*n*, *iterable*, *key = None*)

返回由*iterable*定义的数据集中*n*个最大元素 的列表。 *键*（如果提供）指定一个参数的函数，该参数用于从迭代器中的每个元素提取比较键： 等同于：key=str.lowersorted(iterable, key=key, reverse=True)[:n]

heapq.**nsmallest**(*n*, *iterable*, *key = None*)

返回由*iterable*定义的数据集中*n*个最小元素 的列表。 *键*（如果提供）指定一个参数的函数，该参数用于从迭代器中的每个元素提取比较键： 等同于：key=str.lowersorted(iterable, key=key)[:n]

后两个函数对于较小的*n*值表现最好。对于较大的值，使用该sorted()函数会更有效。另外，何时 n==1使用内置 函数min()和max()函数更有效。如果需要重复使用这些函数，请考虑将迭代器转换为实际的堆。

8.5.1。基本示例

甲堆排序可以通过按所有值到堆，然后突然离开在时刻的最小值的一个来实现：

```
>>> def heapsort(iterable):
...     h = []
...     for value in iterable:
...         heappush(h, value)
...     return [heappop(h) for i in range(len(h))]
...
>>> heapsort([1, 3, 5, 7, 9, 2, 4, 6, 8, 0])
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

这与此类似sorted(iterable)，但与sorted()此不同，此实现不稳定。

堆元素可以是元组。这对于在跟踪的主要记录旁边分配比较值（例如任务优先级）很有用：

```
>>> h = []
>>> heappush(h, (5, 'write code'))
```

```
>>> heappush(h, (7, 'release product'))
>>> heappush(h, (1, 'write spec'))
>>> heappush(h, (3, 'create tests'))
>>> heappop(h)
(1, 'write spec')
```

8.5.2。优先级队列实现说明

一个优先级队列是堆共同使用，并呈现一些执行方面的挑战：

- 排序稳定性：如何获得具有相同优先级的两个任务，按照它们最初添加的顺序返回？
- 如果优先级相同且任务没有默认比较顺序，则（优先级，任务）对的元组比较会中断。
- 如果任务的优先级发生变化，您如何将其移至堆中的新位置？
- 或者如果需要删除待处理任务，您如何找到并将其从队列中删除？

前两个挑战的解决方案是将条目存储为3元素列表，包括优先级，条目计数和任务。输入计数用作决胜因此具有相同优先级的两个任务按照它们添加的顺序返回。由于没有两个入口计数是相同的，所以元组比较决不会直接比较两个任务。

剩下的挑战围绕着寻找未决任务和改变优先级或完全移除它。查找任务可以通过指向队列中的条目的字典完成。

删除条目或改变其优先级更困难，因为它会打破堆结构不变量。因此，一个可能的解决方案是将条目标记为已删除，并添加一个新条目，并修改优先级：

```
pq = []                         # list of entries arranged in a heap
entry_finder = {}               # mapping of tasks to entries
REMOVED = '<removed-task>'      # placeholder for a removed task
counter = itertools.count()     # unique sequence count

def add_task(task, priority=0):
    'Add a new task or update the priority of an existing task'
    if task in entry_finder:
        remove_task(task)
    count = next(counter)
    entry = [priority, count, task]
    entry_finder[task] = entry
    heappush(pq, entry)

def remove_task(task):
    'Mark an existing task as REMOVED.  Raise KeyError if not found.'
    entry = entry_finder.pop(task)
    entry[-1] = REMOVED

def pop_task():
    'Remove and return the lowest priority task. Raise KeyError if empty.'
    while pq:
        priority, count, task = heappop(pq)
        if task is not REMOVED:
            del entry_finder[task]
            return task
    raise KeyError('pop from an empty priority queue')
```

8.5.3。理论

堆是数组，对于所有 *k*，计数从0开始的元素。为了比较，不存在的元素被认为是无限的。堆的有趣属性是它始终是其最小的元素。`a[k] <= a[2*k+1]a[k] <= a[2*k+2]a[0]`

上面这个奇怪的不变性意味着成为比赛的高效内存表示。下面的数字是*k*，而不是`a[k]`：

```
                                0

                1                               2

        3               4               5               6

    7       8       9       10      11      12      13      14

15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30
```

在上面的树中，每个单元格*k*都在顶部`2*k+1`和顶部`2*k+2`。在我们通常看到的体育双星比赛中，每个单元格都是两个单元格上的赢家，我们可以在树上追踪赢家，看看他/她拥有的所有对手。但是，在这样的锦标赛的许多计算机应用程序中，我们不需要追踪获胜者的历史。为了提高记忆效率，当获胜者晋升时，我们试图用低级别的东西替代它，规则变成了一个单元格和它的两个单元格包含三个不同的项目，但顶级单元格“胜利”在两个顶部的单元格上。

如果这个堆不变量始终受到保护，那么索引0显然是总体赢家。最简单的算法去除它并找到“下一个”获胜者是将一些输家（比如上图中的单元格30）移动到0位置，然后将这个新的0渗透到树的下方，交换数值，直到不变量重新建立。这对树中的项目总数显然是对数的。通过迭代所有项目，您将得到一个O（n log n）排序。

这种排序的一个很好的特点是，如果插入的项目不比你提取的最后一个元素“更好”，你可以在排序进行时有效地插入新项目。这在模拟上下文中特别有用，其中树保存所有传入事件，“胜利”条件意味着最短的预定时间。当一个事件安排其他事件执行时，它们将被安排在将来，以便它们可以轻松地进入堆中。所以，堆是实现调度程序的好结构（这是我用于MIDI定序器的:-)。

实施调度程序的各种结构已经得到广泛研究，堆对此有好处，因为它们速度相当快，速度几乎不变，最坏的情况与平均情况相差不大。但是，还有其他表述总体上更有效率，但最糟糕的情况可能会很糟糕。

堆在大磁盘排序中也非常有用。你很可能都知道，大的排序意味着产生“运行”（这是预先排序的序列，其大小通常与CPU内存的数量有关），然后是这些运行的合并过程，合并通常非常巧妙有组织[1]。初始排序产生尽可能最长的运行是非常重要的。锦标赛是实现这一目标的好方法。如果使用所有可用的内存来举办锦标赛，您将替换并渗透恰好与当前运行相匹配的项目，您将生成运行内存大小为随机输入大小的两倍，并且对于模糊排序的输入更好。

此外，如果您在磁盘上输出第0个项目，并获得可能不适合当前锦标赛的输入（因为值“胜过”最后一个输出值），它无法放入堆中，所以堆减少。释放的内存可以被巧妙地重复使用，以逐步构建第二堆，第一堆的堆积速度与第一堆堆积的速度完全相同。当第一个堆完全消失时，您切换堆并开始新的运行。聪明，相当有效！

总而言之，堆是知道有用的内存结构。我在一些应用程序中使用它们，我认为保持'堆'模块是很好的。:-)

脚注

[1]

8.6。bisect- 数组二等分算法

源代码： Lib / bisect.py

该模块支持按排序顺序维护列表，而无需在每次插入后对列表进行排序。对于昂贵的比较操作的长项目列表，这可能是比较常见的方法的改进。该模块被称为bisect是因为它使用基本的二分算法来完成其工作。源代码可能是最有用的算法的实例（边界条件已经正确！）。

提供以下功能：

bisect.**bisect_left**(*a* , *x* , *lo = 0* , *hi = len（a）*)

找到了插入点*X*在一个维持有序。参数*lo*和*hi*可以用来指定应该考虑的列表子集; 默认情况下使用整个列表。如果*x*已存在于*a*中，则插入点将位于任何现有条目之前（在其左侧）。返回值适合用作list.insert()假定*a*已经排序的第一个参数。

返回的插入点*我*分隔阵列一个分为两半，使得用于左侧和 用于右侧。all(val < x for val in a[lo:i])all(val >= x for val in a[i:hi])

bisect.**bisect_right**(*a* , *x* , *lo = 0* , *hi = len（a）*)

bisect.**bisect**(*a* , *x* , *lo = 0* , *hi = len（a）*)

类似bisect_left()，但返回其自带后（到右侧）的任何现有条目的插入点*X*在一个。

返回的插入点*我*分隔阵列一个分为两半，使得用于左侧和 用于右侧。all(val <= x for val in a[lo:i])all(val > x for val in a[i:hi])

bisect.**insort_left**(*a* , *x* , *lo = 0* , *hi = len（a）*)

插入*X*在一个排序顺序。这相当于 假设*a*已经排序。请记住，O（log n）搜索主要由缓慢的O（n）插入步骤决定。a.insert(bisect.bisect_left(a, x, lo, hi), x)

bisect.**insort_right**(*a* , *x* , *lo = 0* , *hi = len（a）*)

bisect.**insort**(*a* , *x* , *lo = 0* , *hi = len（a）*)

类似于insort_left()，但在*x* 的任何现有条目之后的*x*中插入*x*。

也可以看看： 使用bisect来构建全功能集合类的SortedCollection配方，具有直接的搜索方法并支持按键功能。预先计算密钥以在搜索过程中保存对按键功能的不必要调用。

8.6.1。搜索排序列表

上述bisect()功能对于查找插入点非常有用，但对于常见的搜索任务可能会非常棘手或难以使用。以下五个函数显示如何将它们转换为排序列表的标准查找：

```
def index(a, x):
    'Locate the leftmost value exactly equal to x'
    i = bisect_left(a, x)
    if i != len(a) and a[i] == x:
        return i
```

```
    raise ValueError

def find_lt(a, x):
    'Find rightmost value less than x'
    i = bisect_left(a, x)
    if i:
        return a[i-1]
    raise ValueError

def find_le(a, x):
    'Find rightmost value less than or equal to x'
    i = bisect_right(a, x)
    if i:
        return a[i-1]
    raise ValueError

def find_gt(a, x):
    'Find leftmost value greater than x'
    i = bisect_right(a, x)
    if i != len(a):
        return a[i]
    raise ValueError

def find_ge(a, x):
    'Find leftmost item greater than or equal to x'
    i = bisect_left(a, x)
    if i != len(a):
        return a[i]
    raise ValueError
```

8.6.2。其他示例

该bisect()函数可用于数字表查找。这个例子使用bisect()根据一组有序的数字断点来查找考试成绩（比如说）的字母等级：90和up是'A'，80到89是'B'，依此类推：

```
>>> def grade(score, breakpoints=[60, 70, 80, 90], grades='FDCBA'):
...     i = bisect(breakpoints, score)
...     return grades[i]
...
>>> [grade(score) for score in [33, 99, 77, 70, 89, 90, 100]]
['F', 'A', 'C', 'C', 'B', 'A', 'A']
```

与sorted()函数不同，函数bisect()具有*关键*或*颠倒的*参数是没有意义的，因为这会导致设计效率低下（对等函数的连续调用不会“记住”所有先前的关键字查找）。

相反，最好搜索预先计算的键列表以查找有问题的记录的索引：

```
>>> data = [('red', 5), ('blue', 1), ('yellow', 8), ('black', 0)]
>>> data.sort(key=lambda r: r[1])
>>> keys = [r[1] for r in data]         # precomputed list of keys
>>> data[bisect_left(keys, 0)]
('black', 0)
>>> data[bisect_left(keys, 1)]
('blue', 1)
```

```
>>> data[bisect_left(keys, 5)]
('red', 5)
>>> data[bisect_left(keys, 8)]
('yellow', 8)
```

8.7。 array- 有效的数值数组

该模块定义了一个对象类型，它可以紧凑地表示一组基本值：字符，整数，浮点数。数组是序列类型，其行为与列表非常相似，除了存储在其中的对象类型受到约束。该类型是在对象创建时使用*类型代码*指定的，该 *类型代码*是单个字符。以下类型代码被定义：

输入代码	C型	Python类型	最小字节数	笔记
'b'	签名字符	INT	1	
'B'	无符号字符	INT	1	
'u'	Py_UNICODE	Unicode字符	2	（1）
'h'	签署简称	INT	2	
'H'	无符号短	INT	2	
'i'	签名int	INT	2	
'I'	无符号整数	INT	2	
'l'	长签字	INT	4	
'L'	无符号长整数	INT	4	
'q'	签了很久	INT	8	（2）
'Q'	无符号long long	INT	8	（2）
'f'	浮动	浮动	4	
'd'	双	浮动	8	

笔记：

1. 的'u'类型代码对应于Python的过时的Unicode字符（ Py_UNICODE这是wchar_t ）。取决于平台，它可以是16位或32位。

   'u'将会与Py_UNICODE API 的其余部分一起被删除。

   从版本3.3开始弃用，将在版本4.0中删除。

2. 该'q'和'Q'类型代码只可如果用来构建Python平台的C编译器支持C ，或者，在Windows上，。 long long__int64

   3.3版本的新功能

值的实际表示由机器体系结构决定（严格来说，由C实现）。实际大小可以通过itemsize属性访问。

该模块定义了以下类型：

class array. array（ *typecode* [，*initializer*] ）

一个新的数组，其项目受*typecode*限制，并从可选的*初始*值设定项值*初始化*，该值必须是一个列表，类似 字节的对象，或可在相应类型的元素上迭代。

如果给一个列表或字符串，初始化传递给新阵列的 `fromlist()` ， `frombytes()` 或 `fromunicode()` 方法（见下文）初始项添加到阵列。否则，可迭代的初始化器会传递给 `extend()` 方法。

array.**typecodes**

包含所有可用类型代码的字符串。

数组对象支持索引，切片，连接和乘法的普通序列操作。使用切片分配时，分配的值必须是具有相同类型代码的数组对象; 在所有其他情况下，都会 `TypeError` 被提出。数组对象也实现了缓冲接口，并且可以在支持类字节对象的任何地方使用。

以下数据项和方法也受支持：

array.**typecode**

用于创建数组的typecode字符。

array.**itemsize**

内部表示中一个数组项的长度（以字节为单位）。

array.**append**(*x*)

将值为*x*的新项添加到数组的末尾。

array.**buffer_info**()

返回一个元组，给出当前内存地址以及用于保存数组内容的缓冲区元素的长度。内存缓冲区的大小以字节为单位可以计算为。在使用需要内存地址的低级别（和固有不安全）I / O接口时，这有时会很有用，如某些 操作。只要数组存在并且不对其应用长度更改操作，返回的数字就是有效的。 `(address, length)array.buffer_info()[1] * array.itemsizeioctl()`

> **注意：** 当使用C或C ++编写的代码使用数组对象（唯一有效利用这些信息的方法）时，使用数组对象支持的缓冲区接口更有意义。这种方法保持向后兼容性，应该在新代码中避免。缓冲区接口记录在缓冲区协议中。

array.**byteswap**()

“Byteswap”阵列的所有项目。这仅适用于大小为1,2,4或8字节的值; 对于其他类型的价值观而言，这`RuntimeError`是提出 从不同字节顺序的机器上写入文件读取数据时非常有用。

array.**count**(*x*)

返回数组中*x*的出现次数。

array.**extend**(*可迭代*)

将*iterable*中的项追加到数组的末尾。如果*可迭代*是另一个数组，它必须具有*完全相同*的类型代码; 如果没有，`TypeError`将会被提出。如果*iterable*不是数组，则它必须是可迭代的，并且其元素必须是要附加到数组的正确类型。

array.**frombytes**(*s*)

追加字符串中的项目，将字符串解释为机器值的数组（如同使用fromfile()方法从文件中读取一样）。

*3.2版本中的新功能：*为了清晰起见fromstring()而重新命名frombytes()。

array.**fromfile**(*f*, *n*)

从文件对象 *f*中读取*n*项（作为机器值）并将它们附加到数组的末尾。如果少于*n*个项目可用，则引发，但可用的项目仍插入到数组中。*f*必须是一个真正的内置文件对象; 其他方法不会做。 EOFErrorread()

array.**fromlist**(*列表*)

附加列表中的项目。这相当于除了如果存在类型错误，数组不变。 `for x in list: a.append(x)`

array.**fromstring**()

已弃用的别名frombytes()。

array.**fromunicode**(*s*)

用给定的unicode字符串的数据扩展这个数组。该数组必须是一个类型'u'数组; 否则ValueError会提出。使用 `array.frombytes(unicodestring.encode(enc))` 为Unicode数据追加到一些其它类型的阵列。

array.**index**(*x*)

返回最小的*i*，这样*我*就是数组中第一个*x*的索引。

array.**insert**(*i*, *x*)

在位置*i*之前在数组中插入一个值为*x*的新项目。负值被视为相对于数组的末尾。

array.**pop**([*i*])

从数组中删除具有索引*i*的项目并将其返回。可选参数默认为-1，所以默认情况下最后一项被移除并返回。

array.**remove**(*x*)

从数组中删除第一个*x*。

array.**reverse**()

颠倒数组中项目的顺序。

array.**tobytes**()

将数组转换为一组机器值并返回字节表示形式（与tofile()方法将写入文件的字节序列相同）。

*3.2版本中的新功能：*为了清晰起见tostring()而重新命名tobytes()。

array.**tofile**(*f*)

将所有项目（作为机器值）写入文件对象 *f*。

array.**tolist**()

将数组转换为具有相同项目的普通列表。

array.tostring()

已弃用的别名tobytes()。

array.tounicode()

将数组转换为unicode字符串。该数组必须是一个类型'u'数组; 否则ValueError会提出。用于array.tobytes().decode(enc)从其他类型的数组中获取一个unicode字符串。

当一个数组对象被打印或转换为一个字符串时，它被表示为。的*初始值设定*，如果数组为空被省略，否则它是一个字符串，如果*类型代码*是，否则是号码的列表。该字符串保证能够使用相同的类型和值转换回数组，只要类已经使用导入。例子：array(typecode, initializer)'u'eval()arrayfrom array import array

```
array('l')
array('u', 'hello \u2641')
array('l', [1, 2, 3, 4, 5])
array('d', [1.0, 2.0, 3.14])
```

也可以看看:

模 struct

打包和解压缩异构二进制数据。

模 xdrlib

对某些远程过程呼叫系统中使用的外部数据表示（XDR）数据进行打包和解包。

数值Python文档

数字Python扩展（NumPy）定义了另一种数组类型; 有关Numerical Python的更多信息，请参阅 http://www.numpy.org/。

8.8。 weakref- 弱引用

源代码： Lib / weakref.py

该weakref模块允许Python程序员创建对象的*弱引用*。

在下文中，术语*指代*对象是指由弱参考引用的对象。

对对象的弱引用不足以使对象保持活动状态：当对引用的唯一剩余引用是弱引用时， 垃圾收集可以自由地销毁指示对象并将其内存用于其他事情。然而，直到对象被实际销毁，弱引用可能会返回对象，即使没有强引用。

弱引用的主要用途是实现缓存或映射，以保存大对象，因为希望大对象不会因为它出现在缓存或映射中而保持活动状态。

例如，如果您有大量的二进制图像对象，您可能希望将每个对象的名称关联起来。如果您使用Python字典将名称映射到图像或将图像映射到名称，则图像对象将保持活动状态，只是因为它们在字典中显示为值或键。模块提供的 WeakKeyDictionary和WeakValueDictionary类weakref是一种替代方法，它使用弱引用来构造映射，这些映射不会仅仅因为它们出现在映射对象中而使对象保持活动状态。例如，如果一个图像对象是a中的一个值 WeakValueDictionary ，那么当对该图像对象的最后剩余引用是弱映射所持有的弱引用时，垃圾收集可以回收该对象，并且简单地删除其在弱映射中的对应条目。

WeakKeyDictionary并WeakValueDictionary在其实现中使用弱引用，在弱引用上设置回调函数，在通过垃圾回收回收键或值时通知弱字典。 WeakSet实现set接口，但保持对其元素的弱引用，就像一样 WeakKeyDictionary。

finalize提供了一种直接的方式来注册一个清理函数，当一个对象被垃圾收集时被调用。这比在原始弱引用上设置回调函数更简单，因为模块会自动确保终结器保持活动状态，直到收集对象。

大多数程序应该发现，使用这些弱容器类型之一或者finalize他们只需要它们 - 通常不需要直接创建自己的弱引用。底层机器由weakref模块暴露以利于高级用途。

并非所有的对象都可以被弱引用。那些可以包含类实例，用Python编写的函数（但不包含在C中），实例方法，集合，frozensets，一些文件对象，生成器，类型对象，套接字，数组，deques，正则表达式模式对象和代码对象的对象。

*在版本3.2中进行了更改：*添加了对thread.lock，threading.Lock和代码对象的支持。

几种内置类型，例如list和dict不直接支持弱引用，但可以通过子类添加支持：

```
class Dict(dict):
    pass

obj = Dict(red=1, green=2, blue=3)   # this object is weak referenceable
```

其他内置类型，例如tuple和int不支持弱引用，即使在子类化时（这是一个实现细节，可能在不同的Python实现中有所不同）。

扩展类型可以很容易地支持弱引用; 请参阅 弱参考支持。

class weakref. ref（*object*[，*callback*]）

返回*对象*的弱引用。如果指示对象仍然存在，则可以通过调用参考对象来检索原始对象; 如果所指对象不再存在，则调用引用对象将导致None返回。如果提供了*回调*而没有提供*回调* None，并且返回的weakref对象仍然存在，则当对象即将完成时将调用回调; 弱引用对象将作为回调的唯一参数传递; 所指对象将不再可用。

允许为同一个对象构造许多弱引用。为每个弱引用注册的回调将从最近注册的回调调用到最早的注册回调。

回调引发的异常将在标准错误输出中注明，但不能传播; 它们的处理方式与从对象__del__()方法中引发的异常完全相同。

弱引用可哈希如果*对象*是可哈希。即使删除*对象*后，它们仍会保留其哈希值。如果 hash()仅在*对象*被删除后第一次被调用，则调用将会引发TypeError。

弱引用支持测试的平等，但不排序。如果参照物仍然存在，则两个参照物与它们的参照物具有相同的平等关系（不管*回调*）。如果任何对象已被删除，则只有当引用对象是同一对象时，引用才相等。

这是一个可分类的类型而不是工厂函数。

__callback__

该只读属性返回当前与weakref关联的回调。如果没有回调，或者如果weakref的引用不再有效，那么这个属性将有价值None。

*在版本3.4中更改：*添加了该__callback__属性。

weakref. proxy（*object*[，*callback*]）

将代理返回给使用弱引用的*对象*。这支持在大多数情况下使用代理，而不需要使用弱引用对象的显式解引用。返回的对象将具有式的任一 ProxyType或CallableProxyType，这取决于是否*对象*是可调用。代理对象不可指派，不管指示对象如何; 这避免了一些与它们的根本可变性有关的问题，并防止它们用作字典键。 *回调*与该函数的同名参数相同ref()。

weakref. getweakrefcount（*object*）

返回引用*对象*的弱引用和代理的数量。

weakref. getweakrefs（*object*）

回到这指的是所有弱引用和代理对象的列表*对象*。

class weakref. WeakKeyDictionary（[*dict*]）

弱映射键的映射类。当不再有对键的强烈引用时，字典中的条目将被丢弃。这可用于将附加数据与应用程序其他部分拥有的对象关联，而无需向这些对象添加属性。这对覆盖属性访问的对象可能特别有用。

> **注意：** 警告：因为a WeakKeyDictionary是建立在Python字典的基础之上的，所以它在迭代时不能改变大小。这可能难以确保，WeakKeyDictionary因为程序在迭代过程中执行的操作可能会导致字典中的项目“通过魔法”消失（作为垃圾收集的副作用）。

WeakKeyDictionary对象具有直接公开内部引用的附加方法。这些引用在使用时并不保证是“实时”的，因此调用引用的结果需要在使用之前进行检查。这可以用来避免创建引用，这会导致垃圾回收器将密钥保持在比需要更长的时间。

WeakKeyDictionary.keyrefs()

将弱引用的迭代返回给键。

class weakref.WeakValueDictionary([*dict*])

弱映射值的映射类。当没有对该值的强引用时，字典中的条目将被丢弃。

> **注意：** 警告：因为a WeakValueDictionary是建立在Python字典的基础之上的，所以它在迭代时不能改变大小。这可能难以确保，WeakValueDictionary因为程序在迭代过程中执行的操作可能会导致字典中的项目“通过魔法”消失（作为垃圾收集的副作用）。

WeakValueDictionary对象必须具有相同的问题，作为一个额外的方法keyrefs()的方法WeakKeyDictionary的对象。

WeakValueDictionary.valuerefs()

将弱引用的迭代返回给值。

class weakref.WeakSet([*elements*])

设置对其元素保持弱引用的类。当没有强引用时，元素将被丢弃。

*类*weakref.WeakMethod(*方法*)

一种自定义ref子类，它模拟对绑定方法的弱引用（即，在类上定义的方法并在实例上查找）。由于绑定方法是短暂的，所以标准的弱引用不能保持它。 WeakMethod有特殊的代码重新创建绑定的方法，直到对象或原始函数死亡：

```
>>> class C:
...     def method(self):
...         print("method called!")
...
>>> c = C()
>>> r = weakref.ref(c.method)
>>> r()
>>> r = weakref.WeakMethod(c.method)
>>> r()
<bound method C.method of <__main__.C object at 0x7fc859830220>>
>>> r()()
method called!
>>> del c
>>> gc.collect()
0
>>> r()
>>>
```

3.4版新增功能

class weakref.finalize(*obj*, *func*, **args*, ***kwargs*)

返回一个可调用的终结器对象，当*obj*被垃圾收集时将被调用。与普通的弱引用不同，终结器将一直存在，直到收集到引用对象，这极大地简化了生命周期管理。

一个终结器被认为是*活着的*，直到它被调用（明确地或者在垃圾回收），并且在那之后它已经*死了*。调用一个实时终结器返回评估结果，而调用一个终止器返回。func(*arg, **kwargs) None

垃圾收集期间终结器回调引发的异常将显示在标准错误输出上，但无法传播。它们的处理方式与从对象__del__()方法引发的异常或弱引用的回调相同。

程序退出时，除非其atexit属性设置为false，否则将调用每个剩余的实时终结器。它们按照创建的相反顺序进行调用。

在解释器关闭的后期，当模块全局变为可能被替换时，终结器将永远不会调用它的回调None。

__call__()

如果*自己*活着，那么将其标记为死亡并返回调用结果。如果*自己死了*，然后返回。func(*args, **kwargs) None

detach()

如果*自己*活着，那么将其标记为死并返回元组。如果*自己死了*，然后返回。(obj, func, args, kwargs) None

peek()

如果*self*是活着的，则返回该元组。如果*自己死了*，然后返回。(obj, func, args, kwargs) None

alive

如果终结者存活，则属性为真，否则为假。

atexit

一个可写的布尔属性，默认为true。当程序退出时，它会调用所有剩余的实时终结器atexit。它们按照创建的相反顺序进行调用。

注意： 确保*func*，*args*和*kwargs*不直接或间接地拥有任何对*obj*的引用是很重要的，否则*obj*将永远不会被垃圾收集。特别是，*func*不应该是*obj*的绑定方法。

3.4版新增功能

weakref.ReferenceType

弱引用对象的类型对象。

weakref.ProxyType

不可调用对象的代理类型对象。

weakref.CallableProxyType
可调用对象的代理的类型对象。

weakref.ProxyTypes
包含代理的所有类型对象的序列。这可以更简单地测试一个对象是否是代理而不依赖于命名这两种代理类型。

*异常*weakref.ReferenceError
使用代理对象但已收集基础对象时引发异常。这与标准ReferenceError例外相同。

也可以看看:

PEP 205 - 弱引用
该功能的建议和基本原理，包括与早期实现的链接以及其他语言中类似功能的信息。

8.8.1。弱参考对象

弱引用对象除此之外没有方法也没有属性 ref.__callback__。弱引用对象允许通过调用它来获得指示对象，如果它仍然存在的话：

```
>>> import weakref
>>> class Object:
...     pass
...
>>> o = Object()
>>> r = weakref.ref(o)
>>> o2 = r()
>>> o is o2
True
```

如果所指对象不再存在，则调用引用对象返回 None：

```
>>> del o, o2
>>> print(r())
None
```

测试弱参考对象是否仍然存在应该使用表达式来完成。通常，需要使用引用对象的应用程序代码应该遵循以下模式：ref() is not None

```
# r is a weak reference object
o = r()
if o is None:
    # referent has been garbage collected
    print("Object has been deallocated; can't frobnicate.")
else:
    print("Object is still live!")
    o.do_something_useful()
```

对“活力”使用单独的测试会在线程应用程序中创建竞争条件; 另一个线程可能会导致弱引用在调用弱引用之前变为无效状态; 上面显示的习语在线程应用程序以及单线程应用程序中是安全的。

专用版本的ref对象可以通过子类创建。这用于实现WeakValueDictionary该映射中的每个条目以减少内存开销。这对于将附加信息与引用关联起来可能最为有用，但也可以用于在调用中插入附加处理以检索指示对象。

这个例子展示了如何使用一个子类ref来存储关于一个对象的附加信息并影响访问对象时返回的值：

```
import weakref

class ExtendedRef(weakref.ref):
    def __init__(self, ob, callback=None, **annotations):
        super(ExtendedRef, self).__init__(ob, callback)
        self.__counter = 0
        for k, v in annotations.items():
            setattr(self, k, v)

    def __call__(self):
        """Return a pair containing the referent and the number of
        times the reference has been called.
        """
        ob = super(ExtendedRef, self).__call__()
        if ob is not None:
            self.__counter += 1
            ob = (ob, self.__counter)
        return ob
```

8.8.2。示例

这个简单的例子展示了应用程序如何使用对象ID来检索它以前见过的对象。然后，可以在其他数据结构中使用对象的ID，而不强制对象保持活动状态，但如果对象仍然可以通过ID检索对象，则仍然可以使用ID检索对象。

```
import weakref

_id2obj_dict = weakref.WeakValueDictionary()

def remember(obj):
    oid = id(obj)
    _id2obj_dict[oid] = obj
    return oid

def id2obj(oid):
    return _id2obj_dict[oid]
```

8.8.3。终结者对象

使用的主要好处finalize是它可以简单地注册回调而不需要保留返回的终结器对象。例如

```
>>> import weakref
>>> class Object:
...     pass
```

```
...
>>> kenny = Object()
>>> weakref.finalize(kenny, print, "You killed Kenny!")
<finalize object at ...; for 'Object' at ...>
>>> del kenny
You killed Kenny!
```

终结器也可以直接调用。但是终结器最多只会调用一次回调。

```
>>> def callback(x, y, z):
...     print("CALLBACK")
...     return x + y + z
...
>>> obj = Object()
>>> f = weakref.finalize(obj, callback, 1, 2, z=3)
>>> assert f.alive
>>> assert f() == 6
CALLBACK
>>> assert not f.alive
>>> f()                     # callback not called because finalizer dead
>>> del obj                 # callback not called because finalizer dead
```

您可以使用其detach() 方法取消注册终结器。这会杀死终结器，并在创建时返回传递给构造器的参数。

```
>>> obj = Object()
>>> f = weakref.finalize(obj, callback, 1, 2, z=3)
>>> f.detach()
(<__main__.Object object ...>, <function callback ...>, (1, 2), {'z': 3})
>>> newobj, func, args, kwargs = _
>>> assert not f.alive
>>> assert newobj is obj
>>> assert func(*args, **kwargs) == 6
CALLBACK
```

除非您将该atexit属性设置为 False ，否则当程序退出时，如果它仍然存在，则会调用终结器。例如

```
>>> obj = Object()
>>> weakref.finalize(obj, print, "obj dead or exiting")
<finalize object at ...; for 'Object' at ...>
>>> exit()
obj dead or exiting
```

8.8.4。比较终结器和__del__()方法

假设我们想创建一个实例代表临时目录的类。当发生以下第一个事件时，应该删除其目录中的目录：

- 该对象被垃圾收集，
- 该对象的remove()方法被调用，或者
- 该程序退出。

我们可以尝试使用__del__()如下方法实现该类：

```
class TempDir:
    def __init__(self):
        self.name = tempfile.mkdtemp()

    def remove(self):
        if self.name is not None:
            shutil.rmtree(self.name)
            self.name = None

    @property
    def removed(self):
        return self.name is None

    def __del__(self):
        self.remove()
```

从Python 3.4开始，__del__()方法不再阻止引用循环被垃圾收集，并且None在解释器关闭期间不再强制模块全局变量。所以这段代码应该在CPython上没有任何问题。

但是，处理__del__()方法的方式是特定于实现的，因为它取决于解释器垃圾收集器实现的内部细节。

更强大的替代方案可以是定义仅引用它所需的特定函数和对象的终结器，而不是访问对象的完整状态：

```
class TempDir:
    def __init__(self):
        self.name = tempfile.mkdtemp()
        self._finalizer = weakref.finalize(self, shutil.rmtree, self.name)

    def remove(self):
        self._finalizer()

    @property
    def removed(self):
        return not self._finalizer.alive
```

像这样定义，我们的终结器只接收到它需要清理目录的详细信息的引用。如果对象永远不会被垃圾回收，那么终结器仍然会在退出时被调用。

基于weakref的终结器的另一个优点是，它们可以用于为定义由第三方控制的类（例如卸载模块时运行代码）注册终结器：

```
import weakref, sys
def unloading_module():
    # implicit reference to the module globals from the function body
weakref.finalize(sys.modules[__name__], unloading_module)
```

注意: 如果在程序退出时在守护进程线程中创建终结器对象，那么终止器在退出时不会被调用。然而，在恶魔的线程 atexit.register()，并且 不保证清理无论发生。try: ... finally: ... with: ...

8.9。 types- 为内置类型创建动态类型和名称

源代码： Lib / types.py

该模块定义了效用函数以协助动态创建新类型。

它还定义了用于由标准Python解释使用，但不暴露为建宏像某些对象类型名称`int`或 `str`是。

最后，它提供了一些额外的与类型相关的实用程序类和函数，这些类和函数不足以构建内核。

8.9.1。动态类型创建

types. **new_class**(*name*，*bases = ( )*，*kwds = None*，*exec_body = None*)

使用适当的元类动态创建一个类对象。

前三个参数是组成类定义头的组件：类名，基类（按顺序），关键字参数（如`metaclass`）。

该*exec_body*参数是用来填充新创建的类名称空间的回调。它应该接受类名称空间作为其唯一参数，并直接使用类内容更新名称空间。如果没有提供回调，它与传入效果相同。`lambda ns: ns`

3.3版本的新功能

types. **prepare_class**(*name*，*bases = ( )*，*kwds = None*)

计算适当的元类并创建类名称空间。

参数是组成类定义头的组件：类名，基类（按顺序）和关键字参数（如`metaclass`）。

返回值是一个三元组： `metaclass, namespace, kwds`

*元类*是适当的元类，*名称空间*是准备好的类名称空间，而*kwds*是已删除任何条目的传入*kwds*参数的更新副本'`metaclass`'。如果没有*传入kwds* 参数，这将是一个空字典。

3.3版本的新功能

版本3.6 中更改：`namespace`返回的元组的元素的默认值已更改。现在，当元类没有`__prepare__`方法时使用插入顺序保留映射，

也可以看看:

元类
这些功能支持的类创建过程的全部细节

PEP 3115 - Python 3000中的元类
引入了`__prepare__`命名空间钩子

8.9.2。标准解释器类型

该模块提供了许多实现Python解释器所需的类型的名称。它故意避免包括在处理过程中偶然出现的某些类型，例如 `listiterator`类型。

这些名称的典型用途是用于`isinstance()`或 `issubclass()`检查。

标准名称是为以下类型定义的：

types.FunctionType
types.LambdaType
由`lambda` 表达式创建的用户定义函数和函数的类型 。

types.GeneratorType
生成器 - 生成器对象的类型，由生成器函数创建。

types.CoroutineType
协程对象的类型，由函数创建 。 `async def`

3.5版本中的新功能。

types.AsyncGeneratorType
异步生成器对象的类型，由异步生成器函数创建。

3.6版本中的新功能。

types.CodeType
代码对象的类型，如返回的`compile()`。

types.MethodType
用户定义的类实例的方法的类型。

types.BuiltinFunctionType
types.BuiltinMethodType
所述类型的等内置函数`len()`或`sys.exit()` ，和内置类的方法。（这里，“内置”一词的意思是“用C写成”）。

class types.ModuleType（ *name* ，*doc = None* ）
模块的类型。构造函数获取要创建的模块的名称以及可选的文档字符串。

> **注意：** `importlib.util.module_from_spec()`如果您希望设置各种导入控制的属性，请使用创建一个新模块。

__doc__
模块的文档字符串。默认为None。

__loader__

所述装载器，其装载的模块。默认为None。

*版本3.4中更改：*默认为None。之前该属性是可选的。

__name__

模块的名称。

__package__

模块属于哪个包。如果模块是顶层（即不是任何特定软件包的一部分），则应该将该属性设置为''，否则应该将其设置为软件包的名称（ __name__如果模块本身就是软件包）。默认为None。

*版本3.4中更改：*默认为None。之前该属性是可选的。

types.TracebackType

回溯对象的类型，如发现于sys.exc_info()[2]。

types.FrameType

框架对象的类型（如在tb.tb_frameif中找到）tb是一个追溯对象。

types.GetSetDescriptorType

扩展模块中定义的对象类型PyGetSetDef，例如FrameType.f_locals或array.array.typecode。该类型用作对象属性的描述符; 它具有与property类型相同的用途，但是用于扩展模块中定义的类。

types.MemberDescriptorType

扩展模块中定义的对象类型PyMemberDef，例如datetime.timedelta.days。该类型用作使用标准转换函数的简单C数据成员的描述符; 它具有与property类型相同的用途，但是用于扩展模块中定义的类。

CPython实现细节：在Python的其他实现中，此类型可能与之相同GetSetDescriptorType。

*类*types.MappingProxyType(*映射*)

映射的只读代理。它提供了映射条目的动态视图，这意味着映射更改时，视图反映了这些更改。

3.3版本的新功能

key in proxy

True如果底层映射具有*密钥*，则返回，否则返回 False。

proxy[key]

返回与关键基础测绘项目*的关键*。引发一个 KeyErrorif *键*不在底层映射中。

iter(proxy)

通过底层映射的键返回一个迭代器。这是一个捷径iter(proxy.keys())。

len(proxy)

返回底层映射中的项目数量。

copy ( )
返回底层映射的浅表副本。

get (*key* [, *default*])
如果*key*位于底层映射中，则返回*key*的值，否则返回 *default*。如果*默认*没有给出，则默认为，所以，这种方法从未提出了一个。 None KeyError

items ( )
返回底层映射项目（ 对）的新视图。 (key, value)

keys ( )
返回底层映射关键字的新视图。

values ( )
返回底层映射值的新视图。

8.9.3。其他实用工具类和函数

类 types. SimpleNamespace
一个简单的object子类，提供对其名称空间的属性访问权限，以及一个有意义的repr。

不像object ，SimpleNamespace你可以添加和删除属性。如果一个SimpleNamespace对象使用关键字参数进行初始化，那么这些对象会直接添加到底层的命名空间中。

该类型大致等同于以下代码：

```
class SimpleNamespace:
    def __init__(self, **kwargs):
        self.__dict__.update(kwargs)

    def __repr__(self):
        keys = sorted(self.__dict__)
        items = ("{}={!r}".format(k, self.__dict__[k]) for k in keys)
        return "{}({})".format(type(self).__name__, ", ".join(items))

    def __eq__(self, other):
        return self.__dict__ == other.__dict__
```

SimpleNamespace可能作为替代品有用。但是，对于结构化的记录类型，请 改为使用。 class NS: passnamedtuple()

3.3版本的新功能

types. DynamicClassAttribute (*fget = None* , *fset = None* , *fdel = None* , *doc = None*)
将类的路径属性访问路由到__getattr__。

这是一个描述符，用于定义在通过实例和类访问时的行为不同的属性。实例访问保持正常，但通过类访问属性将被路由到类的__getattr__方法; 这是通过引发AttributeError来完成

的。

这样可以让一个实例具有活动属性，并且具有相同名称的类的虚拟属性（请参阅Enum for example）。

3.4版新增功能

8.9.4。协程实用函数

types.coroutine（*gen_func* ）

该函数将生成器函数转换为 协程函数，该函数返回基于生成器的协程。基于生成器的协程仍然是一个生成器迭代器，但也被认为是协程对象并且是可 等待的。但是，它可能不一定实施该__await__()方法。

如果*gen_func*是一个生成器函数，它将被原地修改。

如果*gen_func*不是生成器函数，它将被包装。如果它返回一个实例collections.abc.Generator，则该实例将被包装在一个可*等待的*代理对象中。所有其他类型的对象将按原样返回。

3.5版本中的新功能。

8.10。 copy- 浅而深的复制操作

源代码： Lib / copy.py

Python中的赋值语句不会复制对象，它们会在目标和对象之间创建绑定。对于可变项目或包含可变项目的集合，有时需要副本，以便可以更改一个副本而不更改其他副本。该模块提供通用的浅层和深层复制操作（如下所述）。

接口总结：

copy. **copy** (*x*)
返回*x*的浅表副本。

copy. **deepcopy** (*x*)
返回*x*的深层副本。

*异常*copy. **error**
引发模块特定的错误。

浅层和深层复制之间的区别仅与复合对象（包含其他对象的对象，如列表或类实例）相关：

- 甲*浅拷贝*构造新化合物对象，然后（在可能的范围）插入*引用*到它在原始找到的对象。
- 甲*深层副本*构造新化合物的对象，然后，递归地，插入 *拷贝*到它在原始找到的对象的。

对于浅拷贝操作不存在的深拷贝操作常常存在两个问题：

- 递归对象（直接或间接包含对自身的引用的复合对象）可能会导致递归循环。
- 因为深拷贝复制了可能拷贝太多的所有内容，例如打算在拷贝之间共享的数据。

该deepcopy()功能可以通过以下方式避

- 保留在当前复制过程中已经复制的对象的“备忘录”字典; 和
- 让用户定义的类覆盖复制操作或复制的组件集合。

这个模块不会复制像模块，方法，堆栈跟踪，堆栈框架，文件，套接字，窗口，数组或类似类型的类型。它通过不变地返回原始对象来“复制”函数和类（浅而深）这与这些pickle模块处理的方式是一致的。

例如，可以dict.copy()通过分配整个列表的一部分 来制作浅表副本。 `copied_list = original_list[:]`

类可以使用相同的接口来控制他们用来控制酸洗的复制。有关pickle这些方法的信息，请参阅模块的说明。实际上，copy模块使用模块中已注册的pickle功能copyreg。

为了让类定义自己的拷贝实现，它可以定义特殊的方法__copy__()和__deepcopy__()。前者被称为实施浅拷贝操作; 没有其他参数传递。后者被称为执行深层复制操作; 它传递了一个参数，备忘录字典。如果__deepcopy__()实现需要创建组件的深层副本，则应该deepcopy()使用组件作为第一个参数并将备注字典作为第二个参数来调用该函数。

也可以看看:

模 pickle
讨论用于支持对象状态检索和恢复的特殊方法。

8.11。pprint- 漂亮的数据打印机

源代码： Lib / pprint.py

该pprint模块提供了以可以用作解释器输入的形式“漂亮地”打印任意Python数据结构的能力。如果格式化结构包含不是基本Python类型的对象，则该表示可能无法加载。如果包含诸如文件，套接字或类的对象，以及许多其他不能用Python文字表示的对象，则可能会出现这种情况。

如果可以的话，格式化的表示将对象保留在一行上，如果它们不在允许的宽度内，则将它们分成多行。PrettyPrinter如果需要调整宽度约束，则显式构造对象。

字典在计算显示之前按键排序。

该pprint模块定义了一个类：

class pprint.PrettyPrinter（*indent = 1*，*width = 80*，*depth = None*，*stream = None*，***，*compact = False* ）

构建一个PrettyPrinter实例。这个构造函数理解几个关键字参数。输出流可以使用*stream*关键字来设置; 在流对象上使用的唯一方法是文件协议的 write()方法。如果没有指定，则PrettyPrinter采用 sys.stdout。为每个递归级别添加的缩进量由*缩进*指定; 默认值是1。其他值可能会导致输出看起来有点奇怪，但可以使嵌套更容易找到。可以打印的层数由*深度*控制; 如果正在打印的数据结构太深，则下一个包含的级别将被替换为...。默认情况下，格式化对象的深度没有限制。所需的输出宽度使用*width*参数进行约束; 默认是80个字符。如果一个结构不能在约束宽度内格式化，将尽最大努力。如果*compact*为false（默认），则长序列中的每个项目将被格式化为单独的行。如果*结构紧凑*，则在每个输出行上将格式化*宽度*范围内的项目。

*在版本3.4中更改：*添加了*紧凑*参数。

```
>>> import pprint
>>> stuff = ['spam', 'eggs', 'lumberjack', 'knights', 'ni']
>>> stuff.insert(0, stuff[:])
>>> pp = pprint.PrettyPrinter(indent=4)
>>> pp.pprint(stuff)
[   ['spam', 'eggs', 'lumberjack', 'knights', 'ni'],
    'spam',
    'eggs',
    'lumberjack',
    'knights',
    'ni']
>>> pp = pprint.PrettyPrinter(width=41, compact=True)
>>> pp.pprint(stuff)
[['spam', 'eggs', 'lumberjack',
  'knights', 'ni'],
 'spam', 'eggs', 'lumberjack', 'knights',
 'ni']
>>> tup = ('spam', ('eggs', ('lumberjack', ('knights', ('ni', ('dead',
...  ('parrot', ('fresh fruit',)))))))
>>> pp = pprint.PrettyPrinter(depth=6)
```

```
>>> pp.pprint(tup)
('spam', ('eggs', ('lumberjack', ('knights', ('ni', ('dead', (...)))))))
```

该pprint模块还提供了几个快捷功能：

pprint.**pformat**(*object*, *indent = 1*, *width = 80*, *depth = None*, ***, *compact = False*)

以字符串形式返回*对象*的格式化表示。 *缩进*， *宽度*，*深度*和*紧凑*将PrettyPrinter 作为格式参数传递给构造函数。

*在版本3.4中更改：*添加了*紧凑*参数。

pprint.**pprint**(*object*, *stream = None*, *indent = 1*, *width = 80*, *depth = None*, ***, *compact = False*)

打印正在*流中*的*对象*的格式化表示，然后是换行符。如果*流*是，使用。这可以在交互式解释器中使用，而不是用于检查值的函数（甚至可以重新分配以在范围内使用）。 *缩进*，*宽度*，*深度*和*紧凑*将作为格式参数传递给构造函数。 Nonesys.stdoutprint()print = pprint.pprintPrettyPrinter

*在版本3.4中更改：*添加了*紧凑*参数。

```
>>> import pprint
>>> stuff = ['spam', 'eggs', 'lumberjack', 'knights', 'ni']
>>> stuff.insert(0, stuff)
>>> pprint.pprint(stuff)
[<Recursion on list with id=...>,
 'spam',
 'eggs',
 'lumberjack',
 'knights',
 'ni']
```

pprint.**isreadable**(*object*)

确定*对象*的格式化表示是“可读的”，还是可以用来重构使用的值eval()。这总是返回False递归对象。

```
>>> pprint.isreadable(stuff)
False
```

pprint.**isrecursive**(*object*)

确定*对象是否*需要递归表示。

还定义了一个支持功能：

pprint.**saferepr**(*object*)

返回*对象*的字符串表示形式，保护*对象*不受递归数据结构的影响。如果*对象*的表示公开递归条目，则递归引用将被表示为。该表示不是格式化的。<Recursion on typename with id=number>

```
>>> pprint.saferepr(stuff)
"[<Recursion on list with id=...>, 'spam', 'eggs', 'lumberjack', 'knights', 'ni']"
```

8.11.1。PrettyPrinter对象

PrettyPrinter 实例具有以下方法：

PrettyPrinter.**pformat**(*object*)
返回*对象*的格式化表示。这考虑到传递给PrettyPrinter构造函数的选项。

PrettyPrinter.**pprint**(*object*)
在配置的流上打印*对象*的格式化表示，然后是换行符。

以下方法提供了相同名称的相应功能的实现。由于PrettyPrinter不需要创建新对象，因此在实例上使用这些方法稍微有效一些。

PrettyPrinter.**isreadable**(*object*)
确定对象的格式化表示是“可读的”，还是可用于重构使用的值eval()。请注意，这将返回False递归对象。如果设置了*深度*参数 PrettyPrinter并且对象比允许的深度更深，则返回False。

PrettyPrinter.**isrecursive**(*object*)
确定对象是否需要递归表示。

该方法作为钩子提供，允许子类修改对象转换为字符串的方式。默认实现使用实现的内部saferepr()。

PrettyPrinter.**format**(*object*, *context*, *maxlevels*, *level*)
返回三个值：作为字符串的*对象*的格式化版本，指示结果是否可读的标志以及指示是否检测到递归的标志。第一个参数是要呈现的对象。第二个是一个字典，它包含作为id()当前表示上下文的一部分的对象（ 影响该表示的*对象的*直接和间接容器）作为关键字; 如果需要呈现的对象已经在*上下文中*表示，则第三个返回值应该是True。递归调用format()方法应该为容器添加额外的条目到这个字典。第三个参数， *maxlevels*，给出了递归的请求限制; 这将是0如果没有要求的限制。这个参数应该不加修改地传递给递归调用。第四个参数，*水平*，给出当前的水平; 递归调用应该传递一个小于当前调用的值。

8.11.2。示例

为了演示pprint()函数及其参数的几种用法，让我们从PyPI获取有关项目的信息：

```
>>> import json
>>> import pprint
>>> from urllib.request import urlopen
>>> with urlopen('http://pypi.org/project/Twisted/json') as url:
...     http_info = url.info()
...     raw_data = url.read().decode(http_info.get_content_charset())
>>> project_info = json.loads(raw_data)
```

在其基本形式中，pprint()显示了整个对象：

```
>>> pprint.pprint(project_info)
{'info': {'_pypi_hidden': False,
          '_pypi_ordering': 125,
          'author': 'Glyph Lefkowitz',
          'author_email': 'glyph@twistedmatrix.com',
          'bugtrack_url': '',
          'cheesecake_code_kwalitee_id': None,
          'cheesecake_documentation_id': None,
          'cheesecake_installability_id': None,
          'classifiers': ['Programming Language :: Python :: 2.6',
                          'Programming Language :: Python :: 2.7',
                          'Programming Language :: Python :: 2 :: Only'],
          'description': 'An extensible framework for Python programming, with '
                         'special focus\r\n'
                         'on event-based network programming and multiprotocol '
                         'integration.',
          'docs_url': '',
          'download_url': 'UNKNOWN',
          'home_page': 'http://twistedmatrix.com/',
          'keywords': '',
          'license': 'MIT',
          'maintainer': '',
          'maintainer_email': '',
          'name': 'Twisted',
          'package_url': 'http://pypi.org/project/Twisted',
          'platform': 'UNKNOWN',
          'release_url': 'http://pypi.org/project/Twisted/12.3.0',
          'requires_python': None,
          'stable_version': None,
          'summary': 'An asynchronous networking framework written in Python',
          'version': '12.3.0'},
 'urls': [{'comment_text': '',
           'downloads': 71844,
           'filename': 'Twisted-12.3.0.tar.bz2',
           'has_sig': False,
           'md5_digest': '6e289825f3bf5591cfd670874cc0862d',
           'packagetype': 'sdist',
           'python_version': 'source',
           'size': 2615733,
           'upload_time': '2012-12-26T12:47:03',
           'url': 'https://pypi.org/packages/source/T/Twisted/Twisted-12.3.0.tar.bz2'}
          {'comment_text': '',
           'downloads': 5224,
           'filename': 'Twisted-12.3.0.win32-py2.7.msi',
           'has_sig': False,
           'md5_digest': '6b778f5201b622a5519a2aca1a2fe512',
           'packagetype': 'bdist_msi',
           'python_version': '2.7',
           'size': 2916352,
           'upload_time': '2012-12-26T12:48:15',
           'url': 'https://pypi.org/packages/2.7/T/Twisted/Twisted-12.3.0.win32-py2.7.
```

结果可以限制在一定的*深度*（省略号用于更深的内容）：

```
>>> pprint.pprint(project_info, depth=2)
{'info': {'_pypi_hidden': False,
          '_pypi_ordering': 125,
          'author': 'Glyph Lefkowitz',
          'author_email': 'glyph@twistedmatrix.com',
          'bugtrack_url': '',
          'cheesecake_code_kwalitee_id': None,
          'cheesecake_documentation_id': None,
          'cheesecake_installability_id': None,
          'classifiers': [...],
          'description': 'An extensible framework for Python programming, with '
                         'special focus\r\n'
                         'on event-based network programming and multiprotocol '
                         'integration.',
          'docs_url': '',
          'download_url': 'UNKNOWN',
          'home_page': 'http://twistedmatrix.com/',
          'keywords': '',
          'license': 'MIT',
          'maintainer': '',
          'maintainer_email': '',
          'name': 'Twisted',
          'package_url': 'http://pypi.org/project/Twisted',
          'platform': 'UNKNOWN',
          'release_url': 'http://pypi.org/project/Twisted/12.3.0',
          'requires_python': None,
          'stable_version': None,
          'summary': 'An asynchronous networking framework written in Python',
          'version': '12.3.0'},
 'urls': [{...}, {...}]}
```

此外，可以建议最大字符*宽度*。如果不能拆分长对象，则会超出指定的宽度：

```
>>> pprint.pprint(project_info, depth=2, width=50)
{'info': {'_pypi_hidden': False,
          '_pypi_ordering': 125,
          'author': 'Glyph Lefkowitz',
          'author_email': 'glyph@twistedmatrix.com',
          'bugtrack_url': '',
          'cheesecake_code_kwalitee_id': None,
          'cheesecake_documentation_id': None,
          'cheesecake_installability_id': None,
          'classifiers': [...],
          'description': 'An extensible '
                         'framework for Python '
                         'programming, with '
                         'special focus\r\n'
                         'on event-based network '
                         'programming and '
                         'multiprotocol '
                         'integration.',
          'docs_url': '',
          'download_url': 'UNKNOWN',
          'home_page': 'http://twistedmatrix.com/',
          'keywords': '',
          'license': 'MIT',
          'maintainer': '',
```

```
          'maintainer_email': '',
          'name': 'Twisted',
          'package_url': 'http://pypi.org/project/Twisted',
          'platform': 'UNKNOWN',
          'release_url': 'http://pypi.org/project/Twisted/12.3.0',
          'requires_python': None,
          'stable_version': None,
          'summary': 'An asynchronous networking '
                     'framework written in '
                     'Python',
          'version': '12.3.0'},
 'urls': [{...}, {...}]}
```

8.12。reprlib- 替代repr()实现

源代码： Lib / reprlib.py

该reprlib模块提供了一种生成对象表示的方法，对结果字符串的大小有限制。这在Python调试器中使用，也可能在其他上下文中有用。

该模块提供了一个类，一个实例和一个函数：

*类*reprlib.Repr
提供格式化服务的类，用于实现与内置类似的功能repr()；不同对象类型的大小限制被添加以避免产生过长的表示。

reprlib.aRepr
这是Repr用来提供repr()下面描述的 功能的实例。更改此对象的属性将影响repr()Python调试器使用的大小限制。

reprlib.repr（*obj*）
这是的repr()方法aRepr。它返回一个类似于同名内置函数返回的字符串，但对大多数尺寸有限制。

除了大小限制工具之外，该模块还提供了一个装饰器，用于检测递归调用__repr__()并代替占位符字符串。

@reprlib.recursive_repr（*fillvalue =“...”*）
Decorator用于__repr__()检测同一线程内的递归调用的方法。如果进行递归调用，则返回*fillvalue*，否则进行通常的__repr__()调用。例如：

```
>>> class MyList(list):
...     @recursive_repr()
...     def __repr__(self):
...         return '<' + '|'.join(map(repr, self)) + '>'
...
>>> m = MyList('abc')
>>> m.append(m)
>>> m.append('x')
>>> print(m)
<'a'|'b'|'c'|...|'x'>
```

3.2版本中的新功能

8.12.1。Repr对象

Repr 实例提供了几个可用于为不同对象类型的表示提供大小限制的属性，以及格式化特定对象类型的方法。

Repr.maxlevel

对创建递归表示的深度限制。默认是6。

Repr.maxdict
Repr.maxlist
Repr.maxtuple
Repr.maxset
Repr.maxfrozenset
Repr.maxdeque
Repr.maxarray
限制为指定对象类型表示的条目数量。默认值是4for maxdict ，5for maxarray和 6for others。

Repr.maxlong
整数表示中的最大字符数。数字从中间被丢弃。默认是40。

Repr.maxstring
限制字符串表示中的字符数。请注意，字符串的“正常”表示形式用作字符源：如果在表示形式中需要转义序列，则缩短表示形式时这些转义序列可能会被损坏。默认是30。

Repr.maxother
此限制用于控制对象上没有特定格式化方法的对象类型的大小Repr。它以与...相似的方式应用maxstring。默认是20。

Repr.repr(*obj*)
等同于repr()使用实例施加的格式的内置内容。

Repr.repr1(*obj* , *level*)
递归实现使用repr()。这使用*obj*类型来确定调用哪种格式化方法，并将其传递给*obj*和*level*。特定repr1()于类型的方法应调用执行递归格式化，并在递归调用中使用*level*的值。level - 1

Repr.repr_TYPE(*obj* , *等级*)
特定类型的格式化方法实现为具有基于类型名称的名称的方法。在方法名称中，**TYPE**被替换为'_'.join(type(obj).__name__.split())。派遣到这些方法是由处理repr1()。需要递归地格式化值的类型特定方法应该调用。self.repr1(subobj, level - 1)

8.12.2。继承Repr对象

通过使用动态调度Repr.repr1()允许子类 Repr添加对其他内置对象类型的支持或修改已支持类型的处理。这个例子展示了如何添加对文件对象的特殊支持：

```
import reprlib
import sys

class MyRepr(reprlib.Repr):

    def repr_TextIOWrapper(self, obj, level):
        if obj.name in {'<stdin>', '<stdout>', '<stderr>'}:
```

```
            return obj.name
        return repr(obj)

aRepr = MyRepr()
print(aRepr.repr(sys.stdin))          # prints '<stdin>'
```

8.13。 enum-为枚举支持

3.4版新增功能

源代码： Lib / enum.py

枚举是一组绑定到唯一常量值的符号名称（成员）。在枚举中，可以通过标识比较成员，枚举本身可以迭代。

8.13.1。模块内容

该模块定义了可以用于定义名称和值的独特的套4枚举类： Enum ， IntEnum ， Flag ，和 IntFlag。它还定义了一个装饰器unique() ，和一个帮助器auto。

*类*enum. Enum
创建枚举常量的基类。有关替代构造语法，请参见功能API部分 。

*类*enum. IntEnum
用于创建也是子类的枚举常量的基类int。

*类*enum. IntFlag
用于创建枚举常量的基类，可以使用按位运算符进行组合，而不会丢失其IntFlag成员资格。 IntFlag成员也是子类int。

*类*enum. Flag
创建枚举常量的基类，可以使用按位运算进行组合，而不会丢失其Flag成员资格。

enum. unique ( )
枚举类装饰器，确保只有一个名称绑定到任何一个值。

*类*enum. auto
实例被替换为Enum成员的适当值。

新的3.6版： Flag ， IntFlag ， auto

8.13.2。创建一个

枚举是使用class语法创建的，这使得它们易于读取和写入。Functional API中介绍了另一种创建方法 。要定义一个枚举，子类Enum如下：

```
>>> from enum import Enum
>>> class Color(Enum):
...     RED = 1
...     GREEN = 2
...     BLUE = 3
...
```

注意: 枚举成员值

成员值可以是任何内容：`int` ，`str`等等。如果确切值不重要，您可以使用`auto`实例并为您选择适当的值。如果您`auto`与其他值混合，必须小心 。

注意: 命名法

- 该类`Color`是一个*枚举*（或*枚举*）
- 的属性`Color.RED` ，`Color.GREEN`等等，都是 *枚举成员*（或*枚举成员*）和在功能上是常量。
- 枚举成员具有*名称*和*值*（名称 `Color.RED`是`RED` ，值`Color.BLUE`是 `3`等）

注意: 尽管我们使用`class`语法来创建枚举，但枚举并不是普通的Python类。看看Enums有什么不同？更多细节。

枚举成员具有可读的字符串表示形式：

```
>>> print(Color.RED)
Color.RED
```

...虽然他们`repr`有更多的信息：

```
>>> print(repr(Color.RED))
<Color.RED: 1>
```

枚举成员的*类型*是它所属的枚举：

```
>>> type(Color.RED)
<enum 'Color'>
>>> isinstance(Color.GREEN, Color)
True
>>>
```

枚举成员也有一个只包含他们项目名称的属性：

```
>>> print(Color.RED.name)
RED
```

按定义顺序，枚举支持迭代：

```
>>> class Shake(Enum):
...     VANILLA = 7
...     CHOCOLATE = 4
...     COOKIES = 9
...     MINT = 3
...
>>> for shake in Shake:
...     print(shake)
...
Shake.VANILLA
```

```
Shake.CHOCOLATE
Shake.COOKIES
Shake.MINT
```

枚举成员是可散列的，因此它们可以用在字典和集合中：

```
>>> apples = {}
>>> apples[Color.RED] = 'red delicious'
>>> apples[Color.GREEN] = 'granny smith'
>>> apples == {Color.RED: 'red delicious', Color.GREEN: 'granny smith'}
True
```

8.13.3。编程访问枚举成员及其属性

有时，以编程方式访问枚举中的成员是有用的（即在编程时Color.RED不知道确切颜色的情况下不会执行的情况）。 Enum允许这样的访问：

```
>>> Color(1)
<Color.RED: 1>
>>> Color(3)
<Color.BLUE: 3>
```

如果您想通过*名称*访问枚举成员，请使用项目访问权限：

```
>>> Color['RED']
<Color.RED: 1>
>>> Color['GREEN']
<Color.GREEN: 2>
```

如果你有一个枚举成员并需要它name或者value：

```
>>> member = Color.RED
>>> member.name
'RED'
>>> member.value
1
```

8.13.4。复制枚举成员和值

有两个同名的枚举成员是无效的：

```
>>> class Shape(Enum):
...     SQUARE = 2
...     SQUARE = 3
...
Traceback (most recent call last):
...
TypeError: Attempted to reuse key: 'SQUARE'
```

但是，允许两个枚举成员具有相同的值。给定两个成员A和B具有相同的值（并且首先定义A），则B是A的别名。通过A和B的值的按值查找将返回A.通过B的名称查找还将返回A：

```
>>> class Shape(Enum):
...     SQUARE = 2
...     DIAMOND = 1
...     CIRCLE = 3
...     ALIAS_FOR_SQUARE = 2
...
>>> Shape.SQUARE
<Shape.SQUARE: 2>
>>> Shape.ALIAS_FOR_SQUARE
<Shape.SQUARE: 2>
>>> Shape(2)
<Shape.SQUARE: 2>
```

注意： 尝试创建与已定义属性（另一个成员，方法等）相同名称的成员或试图创建与成员具有相同名称的属性是不允许的。

8.13.5。确保独特的枚举值

默认情况下，枚举允许多个名称作为相同值的别名。当不需要这种行为时，可以使用以下装饰器来确保每个值在枚举中只使用一次：

@enum.unique

一个class专门用于枚举的装饰器。它搜索一个枚举的__members__聚集它找到的任何别名; 如果发现任何被发现ValueError的细节：

```
>>> from enum import Enum, unique
>>> @unique
... class Mistake(Enum):
...     ONE = 1
...     TWO = 2
...     THREE = 3
...     FOUR = 3
...
Traceback (most recent call last):
...
ValueError: duplicate values found in <enum 'Mistake'>: FOUR -> THREE
```

8.13.6。使用自动值

如果确切的值不重要，您可以使用auto：

```
>>> from enum import Enum, auto
>>> class Color(Enum):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
```

```
>>> list(Color)
[<Color.RED: 1>, <Color.BLUE: 2>, <Color.GREEN: 3>]
```

值由以下选择_generate_next_value_()，可以被覆盖：

```
>>> class AutoName(Enum):
...     def _generate_next_value_(name, start, count, last_values):
...         return name
...
>>> class Ordinal(AutoName):
...     NORTH = auto()
...     SOUTH = auto()
...     EAST = auto()
...     WEST = auto()
...
>>> list(Ordinal)
[<Ordinal.NORTH: 'NORTH'>, <Ordinal.SOUTH: 'SOUTH'>, <Ordinal.EAST: 'EAST'>, <Ordinal.
```

注意： 默认_generate_next_value_()方法的目标是int按顺序提供下一个int提供的最后一个，但其执行方式是实现细节并可能会更改。

8.13.7。迭代

迭代枚举的成员不会提供别名：

```
>>> list(Shape)
[<Shape.SQUARE: 2>, <Shape.DIAMOND: 1>, <Shape.CIRCLE: 3>]
```

特殊属性__members__是将名称映射到成员的有序字典。它包括在枚举中定义的所有名称，包括别名：

```
>>> for name, member in Shape.__members__.items():
...     name, member
...
('SQUARE', <Shape.SQUARE: 2>)
('DIAMOND', <Shape.DIAMOND: 1>)
('CIRCLE', <Shape.CIRCLE: 3>)
('ALIAS_FOR_SQUARE', <Shape.SQUARE: 2>)
```

该__members__属性可用于详细的编程访问枚举成员。例如，查找所有别名：

```
>>> [name for name, member in Shape.__members__.items() if member.name != name]
['ALIAS_FOR_SQUARE']
```

8.13.8。比较

枚举成员通过身份进行比较：

```
>>> Color.RED is Color.RED
True
>>> Color.RED is Color.BLUE
False
>>> Color.RED is not Color.BLUE
True
```

在枚举值之间有序的比较*不*支持。枚举成员不是整数（但请参见下面的IntEnum）：

```
>>> Color.RED < Color.BLUE
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'Color' and 'Color'
```

平等比较的定义如下：

```
>>> Color.BLUE == Color.RED
False
>>> Color.BLUE != Color.RED
True
>>> Color.BLUE == Color.BLUE
True
```

与非枚举值比较总是比较不相等（同样，IntEnum明确设计的行为不同，请参见下文）：

```
>>> Color.BLUE == 2
False
```

8.13.9。枚举允许的成员和属性

上面的例子使用整数作为枚举值。使用整数很简单且方便（并且由Functional API默认提供），但不是严格执行。在绝大多数用例中，人们并不关心枚举的实际值是什么。但是，如果值*很*重要，枚举可以具有任意值。

枚举是Python类，并且可以像往常一样具有方法和特殊方法。如果我们有这个枚举：

```
>>> class Mood(Enum):
...     FUNKY = 1
...     HAPPY = 3
...
...     def describe(self):
...         # self is the member here
...         return self.name, self.value
...
...     def __str__(self):
...         return 'my custom str! {0}'.format(self.value)
...
...     @classmethod
...     def favorite_mood(cls):
...         # cls here is the enumeration
...         return cls.HAPPY
...
```

然后：

```
>>> Mood.favorite_mood()
<Mood.HAPPY: 3>
>>> Mood.HAPPY.describe()
('HAPPY', 3)
>>> str(Mood.FUNKY)
'my custom str! 1'
```

允许的规则如下：以一个下划线开头和结尾的名称由枚举保留，不能使用; 枚举中定义将成为该列举的成员的所有其他属性，具有特殊的方法（除__str__()，__add__()等）和描述符（方法也描述符）。

注意：如果你的枚举定义了__new__()和/或__init__()任何赋予枚举成员的值将被传入这些方法。以Planet为例。

10年8月13日。枚举的受限子类

仅当枚举未定义任何成员时才允许继承枚举。所以这是禁止的：

```
>>> class MoreColor(Color):
...     PINK = 17
...
Traceback (most recent call last):
...
TypeError: Cannot extend enumerations
```

但是这是允许的：

```
>>> class Foo(Enum):
...     def some_behavior(self):
...         pass
...
>>> class Bar(Foo):
...     HAPPY = 1
...     SAD = 2
...
```

允许定义成员的枚举的子类化会导致违反类型和实例的一些重要不变量。另一方面，允许在一组枚举之间共享一些共同的行为是有意义的。（有关示例，请参阅OrderedEnum。）

11年8月13日。酸洗

枚举可以被腌渍和取消：

```
>>> from test.test_enum import Fruit
>>> from pickle import dumps, loads
>>> Fruit.TOMATO is loads(dumps(Fruit.TOMATO))
True
```

酸洗的通常限制条件如下：可剔除的枚举必须在模块的顶层定义，因为unpickling要求它们可以从该模块导入。

> **注意：** 通过使用pickle协议版本4，可以轻松地将嵌套在其他类中的枚举枚举出来。

通过`__reduce_ex__()`在枚举类中定义，可以修改枚举成员如何腌渍/取消挑选 。

12年8月13日。功能性

的`Enum`类是可调用，提供以下功能API：

```
>>> Animal = Enum('Animal', 'ANT BEE CAT DOG')
>>> Animal
<enum 'Animal'>
>>> Animal.ANT
<Animal.ANT: 1>
>>> Animal.ANT.value
1
>>> list(Animal)
[<Animal.ANT: 1>, <Animal.BEE: 2>, <Animal.CAT: 3>, <Animal.DOG: 4>]
```

这个API的语义相似`namedtuple`。调用的第一个参数`Enum`是枚举的名称。

第二个参数是枚举成员名称的*来源*。它可以是空格分隔的名称字符串，名称序列，具有键/值对的2元组序列或名称到值的映射（例如字典）。最后两个选项可以将任意值分配给枚举; 其他人自动分配从1开始的递增整数（使用`start`参数指定不同的起始值）。`Enum`返回派生自的新类。换句话说，上述分配`Animal`等同于：

```
>>> class Animal(Enum):
...     ANT = 1
...     BEE = 2
...     CAT = 3
...     DOG = 4
...
```

原因默认为`1`为起始号，而不是`0`是`0`是`False`在布尔意义，但枚举成员都执行为`True`。

使用功能性API创建的酸洗枚举可能会非常棘手，因为使用框架堆栈实现细节来试图找出哪个模块正在被创建（例如，如果您在单独的模块中使用实用程序功能，它也会失败，并且可能无法工作在IronPython或Jython上）。解决方案是明确指定模块名称，如下所示：

```
>>> Animal = Enum('Animal', 'ANT BEE CAT DOG', module=__name__)
```

> **警告：** 如果`module`没有提供，并且Enum不能确定它是什么，那么新的Enum成员将不会是不可取的; 为了使错误更接近源头，酸洗将被禁用。

在一些情况下，新的泡菜协议4还依赖于 `__qualname__`被设置为泡菜能够找到该类的位置。例如，如果该类在全局范围的类SomeData中可用：

```
>>> Animal = Enum('Animal', 'ANT BEE CAT DOG', qualname='SomeData.Animal')
```

完整的签名是：

```
Enum(value='NewEnumName', names=<...>, *, module='...', qualname='...', type=<mixed-in
```

值：	新的Enum类会记录它的名字。

名称：	Enum成员。这可以是一个空格或逗号分隔的字符串（除非另有说明，值将从1开始）：
	`'RED GREEN BLUE' | 'RED,GREEN,BLUE' | 'RED, GREEN, BLUE'`
	或名称的迭代器：
	`['RED', 'GREEN', 'BLUE']`
	或（名称，值）对的迭代器：
	`[('CYAN', 4), ('MAGENTA', 5), ('YELLOW', 6)]`
	或映射：
	`{'CHARTREUSE': 7, 'SEA_GREEN': 11, 'ROSEMARY': 42}`

模块：	可以找到新的Enum类的模块的名称。
qualname：	在模块中可以找到新的Enum类。
类型：	键入混合到新的Enum类。
开始：	如果只传入名字，则开始计数。

在3.5版本中更改：将启动加入参数。

13年8月13日。派生枚举

8.13.13.1。IntEnum

提供的第一个变体`Enum`也是一个子类 `int`。一个成员`IntEnum`可以与整数相比较;通过扩展，不同类型的整数枚举也可以相互比较：

```
>>> from enum import IntEnum
>>> class Shape(IntEnum):
...     CIRCLE = 1
...     SQUARE = 2
...
>>> class Request(IntEnum):
...     POST = 1
...     GET = 2
```

```
...
>>> Shape == 1
False
>>> Shape.CIRCLE == 1
True
>>> Shape.CIRCLE == Request.POST
True
```

但是，它们仍然不能与标准Enum枚举进行比较：

```
>>> class Shape(IntEnum):
...     CIRCLE = 1
...     SQUARE = 2
...
>>> class Color(Enum):
...     RED = 1
...     GREEN = 2
...
>>> Shape.CIRCLE == Color.RED
False
```

IntEnum 值的行为像您期望的其他方式的整数一样：

```
>>> int(Shape.CIRCLE)
1
>>> ['a', 'b', 'c'][Shape.CIRCLE]
'b'
>>> [i for i in range(Shape.SQUARE)]
[0, 1]
```

8.13.13.2。INTFLAG

下一变化Enum提供IntFlag ，也是基于int。 IntFlag成员之间的区别可以使用按位运算符（ &，|，^，~ ）进行组合，结果仍然是 IntFlag成员。然而，顾名思义，IntFlag 成员也是子类，int并且可以在任何地方int使用。 IntFlag除了按位操作之外，对成员的任何操作都将失去IntFlag成员资格。

3.6版本中的新功能。

示例IntFlag类：

```
>>> from enum import IntFlag
>>> class Perm(IntFlag):
...     R = 4
...     W = 2
...     X = 1
...
>>> Perm.R | Perm.W
<Perm.R|W: 6>
>>> Perm.R + Perm.W
6
>>> RW = Perm.R | Perm.W
```

```
>>> Perm.R in RW
True
```

也可以命名这些组合：

```
>>> class Perm(IntFlag):
...     R = 4
...     W = 2
...     X = 1
...     RWX = 7
>>> Perm.RWX
<Perm.RWX: 7>
>>> ~Perm.RWX
<Perm.-8: -8>
```

之间的另一个重要区别IntFlag和Enum的是，如果没有标志被设置（其值为0）时，它的布尔评价是False：

```
>>> Perm.R & Perm.X
<Perm.0: 0>
>>> bool(Perm.R & Perm.X)
False
```

因为IntFlag成员也int可以与它们的子类相结合：

```
>>> Perm.X | 8
<Perm.8|X: 9>
```

8.13.13.3。标记

最后的变化是Flag。同样IntFlag，Flag 可以使用按位运算符（&，|，^，~）组合成员。不同的是 IntFlag，它们不能与任何其他Flag枚举相结合，也不能相互比较int。虽然可以直接指定值，但建议将其auto用作值并Flag选择适当的值。

3.6版本中的新功能。

就像IntFlag，如果Flag成员组合没有设置标志，布尔评估是False：

```
>>> from enum import Flag, auto
>>> class Color(Flag):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> Color.RED & Color.GREEN
<Color.0: 0>
>>> bool(Color.RED & Color.GREEN)
False
```

单个标志的值应该是2的幂（1,2,4,8，...），而标志的组合不会：

```
>>> class Color(Flag):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...     WHITE = RED | BLUE | GREEN
...
>>> Color.WHITE
<Color.WHITE: 7>
```

给“no flags set”条件命名不会改变它的布尔值：

```
>>> class Color(Flag):
...     BLACK = 0
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> Color.BLACK
<Color.BLACK: 0>
>>> bool(Color.BLACK)
False
```

注意： 对于大多数新的代码，Enum并Flag强烈建议，因为IntEnum并IntFlag打破枚举的一些语义的承诺（以之中，从而通过传递给其他不相关枚举媲美整数）。 IntEnum 而IntFlag只应在情况下使用Enum，并 Flag不会做; 例如，当整数常量被枚举代替时，或与其他系统互操作时。

8.13.13.4。其他

虽然IntEnum是enum模块的一部分，但独立实现将非常简单：

```
class IntEnum(int, Enum):
    pass
```

这演示了如何定义相似派生枚举; 例如一个StrEnum混合str而不是int。

一些规则：

1. 子类化时Enum，混合类型必须出现在Enum基类序列之前 ，如上IntEnum 例所示。
2. 虽然Enum可以有任何类型的成员，但是一旦混合使用其他类型，所有成员都必须具有该类型的值，例如 int上面的值。此限制不适用于仅添加方法且未指定其他数据类型（如intor）的混合 str。
3. 当另一数据类型被混合，该value属性是*不一样*的枚举构件本身，虽然它是等效，将比较相等。
4. %风格的格式： *%s*和 *%r*分别调用Enum类 __str__()和方法__repr__()；其他代码（例如IntEnum的 *%i*或 *%h*）将该枚举成员视为其混入类型。
5. 格式化字符串文字，str.format()以及format()将采用混合式的__format__()。如果Enum类的str()或者 repr()是需要的，请使用 *！小号*或 *！* - *[R*格式代码。

14年8月13日。有趣的例子

虽然Enum，IntEnum，IntFlag，并Flag预计将覆盖大部分的使用情况，他们不能涵盖所有。以下是可以直接使用的一些不同枚举类型的配方，或者作为创建自己的枚举的示例。

8.13.14.1。忽略值

在许多使用情况下，人们并不关心枚举的实际价值。有几种方法可以定义这种简单的枚举类型：

- 使用auto值的实例
- 使用实例object作为值
- 使用描述性字符串作为值
- 使用元组的值和一个自定义__new__()与替换元组int值

使用这些方法中的任何一种都可以向用户表示这些值不重要，并且使得用户能够添加，移除或重新排序成员，而不必重新编号剩余的成员。

无论您选择哪种方法，您都应该提供一个repr()也隐藏（不重要）值的方法：

```
>>> class NoValue(Enum):
...     def __repr__(self):
...         return '<%s.%s>' % (self.__class__.__name__, self.name)
...
```

8.13.14.1.1。使用 auto

使用auto看起来像：

```
>>> class Color(NoValue):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> Color.GREEN
<Color.GREEN>
```

8.13.14.1.2。使用 object

使用object看起来像：

```
>>> class Color(NoValue):
...     RED = object()
...     GREEN = object()
...     BLUE = object()
...
>>> Color.GREEN
<Color.GREEN>
```

8.13.14.1.3。使用描述性字符串

使用字符串作为值将如下所示：

```
>>> class Color(NoValue):
...     RED = 'stop'
...     GREEN = 'go'
...     BLUE = 'too fast!'
...
>>> Color.GREEN
<Color.GREEN>
>>> Color.GREEN.value
'go'
```

8.13.14.1.4。使用自定义 __new__()

使用自动编号__new__()将如下所示：

```
>>> class AutoNumber(NoValue):
...     def __new__(cls):
...         value = len(cls.__members__) + 1
...         obj = object.__new__(cls)
...         obj._value_ = value
...         return obj
...
>>> class Color(AutoNumber):
...     RED = ()
...     GREEN = ()
...     BLUE = ()
...
>>> Color.GREEN
<Color.GREEN>
>>> Color.GREEN.value
2
```

注意： 该__new__()方法（如果已定义）在创建Enum成员期间使用; 然后它被替换__new__()为在创建类之后用于查找现有成员的Enum 。

8.13.14.2。OrderedEnum

有序枚举不基于IntEnum并保持常规Enum不变量（例如不能与其他枚举进行比较）：

```
>>> class OrderedEnum(Enum):
...     def __ge__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value >= other.value
...         return NotImplemented
...     def __gt__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value > other.value
...         return NotImplemented
...     def __le__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value <= other.value
...         return NotImplemented
...     def __lt__(self, other):
...         if self.__class__ is other.__class__:
```

```
...             return self.value < other.value
...         return NotImplemented
...
>>> class Grade(OrderedEnum):
...     A = 5
...     B = 4
...     C = 3
...     D = 2
...     F = 1
...
>>> Grade.C < Grade.A
True
```

8.13.14.3。DuplicateFreeEnum

如果找到重复的成员名称而不是创建别名，则会引发错误：

```
>>> class DuplicateFreeEnum(Enum):
...     def __init__(self, *args):
...         cls = self.__class__
...         if any(self.value == e.value for e in cls):
...             a = self.name
...             e = cls(self.value).name
...             raise ValueError(
...                 "aliases not allowed in DuplicateFreeEnum:  %r --> %r"
...                 % (a, e))
...
>>> class Color(DuplicateFreeEnum):
...     RED = 1
...     GREEN = 2
...     BLUE = 3
...     GRENE = 2
...
Traceback (most recent call last):
...
ValueError: aliases not allowed in DuplicateFreeEnum:  'GRENE' --> 'GREEN'
```

注意： 这是一个很有用的例子，它让Enum子类添加或更改其他行为以及禁止别名。如果唯一需要的更改不允许别名，则`unique()`可以使用装饰器代替。

8.13.14.4。星球

如果`__new__()`或被`__init__()`定义，枚举成员的值将被传递给这些方法：

```
>>> class Planet(Enum):
...     MERCURY = (3.303e+23, 2.4397e6)
...     VENUS   = (4.869e+24, 6.0518e6)
...     EARTH   = (5.976e+24, 6.37814e6)
...     MARS    = (6.421e+23, 3.3972e6)
...     JUPITER = (1.9e+27,   7.1492e7)
...     SATURN  = (5.688e+26, 6.0268e7)
...     URANUS  = (8.686e+25, 2.5559e7)
...     NEPTUNE = (1.024e+26, 2.4746e7)
```

```
...     def __init__(self, mass, radius):
...         self.mass = mass       # in kilograms
...         self.radius = radius   # in meters
...     @property
...     def surface_gravity(self):
...         # universal gravitational constant  (m3 kg-1 s-2)
...         G = 6.67300E-11
...         return G * self.mass / (self.radius * self.radius)
...
>>> Planet.EARTH.value
(5.976e+24, 6378140.0)
>>> Planet.EARTH.surface_gravity
9.802652743337129
```

15年8月13日。Enums有什么不同？

枚举有一个自定义的元类，它影响派生的Enum类和它们的实例（成员）的许多方面。

8.13.15.1。枚举类

该EnumMeta元类是负责提供 `__contains__()`，`__dir__()`，`__iter__()`和其他方法，使一个做事有Enum类于一个典型的类失败，如*清单（彩色）*或*some_var的颜色*。 EnumMeta是负责确保对最终各种其它方法Enum 类是正确的（例如 `__new__()`，`__getnewargs__()`， `__str__()`和 `__repr__()`）。

8.13.15.2。枚举成员（又名实例）

关于Enum成员最有趣的是他们是单身人士。 EnumMeta在它创建Enum 类本身的同时创建它们，然后放入一个定制`__new__()`以确保不会通过仅返回现有的成员实例来实例化新实例。

8.13.15.3。精细点

8.13.15.3.1。支持的__dunder__名称

__members__是OrderedDict的member_name：member 项目。它只在课堂上提供。

`__new__()`如果指定，则必须创建并返回枚举成员; _value_适当地设置成员也是一个非常好的主意。一旦所有成员被创建，它就不再使用。

8.13.15.3.2。支持的_sunder_名称

- _name_ - 成员的名字
- _value_ - 会员的价值; 可以在中设置/修改__new__
- _missing_ - 未找到值时使用的查找函数; 可能会被覆盖
- _order_ - 在Python 2/3代码中用于确保成员顺序一致（在创建类时删除类属性）
- _generate_next_value_ - 由Functional API使用，并 auto为enum成员获取适当的值; 可能会被覆盖

新的3.6版： _missing_ ， _order_ ， _generate_next_value_

为了帮助保持Python 2 / Python 3代码同步， _order_可以提供一个属性。它将根据枚举的实际顺序进行检查，并在两者不匹配时提出错误：

```
>>> class Color(Enum):
...     _order_ = 'RED GREEN BLUE'
...     RED = 1
...     BLUE = 3
...     GREEN = 2
...
Traceback (most recent call last):
...
TypeError: member order does not match _order_
```

注意： 在Python 2代码中， _order_属性是必需的，因为定义顺序在可以被记录之前丢失。

8.13.15.3.3。 Enum成员类型

Enum成员是他们Enum班的实例，并且通常作为访问EnumClass.member。在某些情况下，它们也可以被访问EnumClass.member.member ，但是你不应该这样做，因为查找可能会失败，或者更糟糕的是，返回除了Enum你正在寻找的成员之外的东西 （ 这是为成员使用全大写名称的另一个好理由 ）：

```
>>> class FieldTypes(Enum):
...     name = 0
...     value = 1
...     size = 2
...
>>> FieldTypes.value.size
<FieldTypes.size: 2>
>>> FieldTypes.size.value
2
```

在版本3.5中更改。

8.13.15.3.4。 Enum类和成员的布尔值

Enum了与非混合成员Enum类型（ 如 int ， str等 ）是根据评估的混合型的规则; 否则，所有成员评估为True。为了让您自己的Enum的布尔评估取决于成员的价值，请将以下内容添加到您的课程中：

```
def __bool__(self):
    return bool(self.value)
```

Enum类总是评估为True。

8.13.15.3.5。 Enum带方法的类

如果你给你的Enum子类额外的方法，比如 上面的Planet类，那么这些方法将显示在dir()成员中，但不是该类的成员：

```
>>> dir(Planet)
['EARTH', 'JUPITER', 'MARS', 'MERCURY', 'NEPTUNE', 'SATURN', 'URANUS', 'VENUS', '__cla
>>> dir(Planet.EARTH)
['__class__', '__doc__', '__module__', 'name', 'surface_gravity', 'value']
```

8.13.15.3.6。结合　　成员Flag

如果Flag成员的组合未被命名，那么repr()将包含所有已命名的标志和所有在该值中的命名的标志组合：

```
>>> class Color(Flag):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...     MAGENTA = RED | BLUE
...     YELLOW = RED | GREEN
...     CYAN = GREEN | BLUE
...
>>> Color(3)  # named combination
<Color.YELLOW: 3>
>>> Color(7)      # not named combination
<Color.CYAN|MAGENTA|BLUE|YELLOW|GREEN|RED: 7>
```

9.数字和数学模块

本章介绍的模块提供数字和数学相关的函数和数据类型。该numbers模块定义了数字类型的抽象层次结构。在math和cmath模块包含了对浮点和复数各种数学函数。该decimal 模块支持使用任意精度算术的十进制数的精确表示。

本章介绍以下模块：

9.1。numbers-数字抽象基类

源代码： Lib / numbers.py

该numbers模块（**PEP 3141**）定义了逐步定义更多操作的数字抽象基类的层次结构。本模块中定义的任何类型都不能实例化。

*类*numbers.Number
数字层次结构的根。如果你只是想检查一个参数 *x*是否是一个数字，而不关心什么样的话，那就使用。isinstance(x, Number)

9.1.1。数字塔

*类*numbers.Complex
这种类型的子类描述了复数，并包含了对内建complex类型起作用的操作。它们是：转换到complex和bool，real，imag，+，-，*，/，abs()，conjugate()，==，和!=。所有除了-和!=是抽象的。

real
抽象。检索此数字的实际组成部分。

imag
抽象。检索这个数字的虚部。

abstractmethod conjugate()
抽象。返回复共轭。例如，。(1+3j).conjugate() == (1-3j)

*类*numbers.Real
To Complex，Real增加了对实数起作用的操作。

总之，这些是：一次转化float，math.trunc()，round()，math.floor()，math.ceil()，divmod()，//，%，<，<=，>，和>=。

真正还提供了默认值complex()，real，imag，和conjugate()。

*类*numbers.Rational
子类型Real和增加 numerator和denominator属性，应该是最低的。有了这些，它提供了一个默认值 float()。

numerator
抽象。

denominator
抽象。

*类*numbers.Integral

子类型Rational并向其添加转换int。提供了默认值float()，numerator和 denominator。增加了对抽象方法**和比特字符串操作：<<，>>，&，^，|，~。

9.1.2。类型实现者的注释

实现者应该小心地使相等的数字相等并将它们散列为相同的值。如果有两个不同的实数扩展名，这可能很微妙。例如，fractions.Fraction 实现hash()如下：

```
def __hash__(self):
    if self.denominator == 1:
        # Get integers right.
        return hash(self.numerator)
    # Expensive check, but definitely correct.
    if self == float(self):
        return hash(float(self))
    else:
        # Use tuple's hash to avoid a high collision rate on
        # simple fractions.
        return hash((self.numerator, self.denominator))
```

9.1.2.1。添加更多的数字基本知识

当然，对于数字来说，有更多可能的ABC，如果它排除了添加这些数字的可能性，这将是一个糟糕的等级。你可以添加和 MyFoo之间：ComplexReal

```
class MyFoo(Complex): ...
MyFoo.register(Real)
```

9.1.2.2。实现算术运算

我们希望实现算术运算，以便混合模式操作或者调用作者知道两个参数类型的实现，或者将两者都转换为最接近的内置类型并在那里执行操作。对于其中的子类型Integral，这意味着__add__()并__radd__()应该被定义为：

```
class MyIntegral(Integral):

    def __add__(self, other):
        if isinstance(other, MyIntegral):
            return do_my_adding_stuff(self, other)
        elif isinstance(other, OtherTypeIKnowAbout):
            return do_my_other_adding_stuff(self, other)
        else:
            return NotImplemented

    def __radd__(self, other):
        if isinstance(other, MyIntegral):
            return do_my_adding_stuff(other, self)
        elif isinstance(other, OtherTypeIKnowAbout):
            return do_my_other_adding_stuff(other, self)
        elif isinstance(other, Integral):
```

```
            return int(other) + int(self)
        elif isinstance(other, Real):
            return float(other) + float(self)
        elif isinstance(other, Complex):
            return complex(other) + complex(self)
        else:
            return NotImplemented
```

对于子类的混合类型操作有5种不同的情况Complex。我将参考上述所有没有提及的代码，MyIntegral并将其OtherTypeIKnowAbout作为“样板”。a将是一个实例A，它是Complex（），和的子类型。我会考虑：a : A <: Complexb : B <: Complexa + b

1. 如果A定义一个__add__()接受b，一切都很好。
2. 如果A退回到样板代码，并从中返回一个值__add__()，我们会错过B定义更智能的可能性__radd__()，所以样板应该NotImplemented从中 返回__add__()。（或者A可能根本没有实施__add__()。）
3. 随后B的__radd__()得到一个机会。如果它接受 a，一切都很好。
4. 如果它回落到样板，则没有更多可能的方法来尝试，所以这是默认实现应该存在的地方。
5. 如果，Python尝试之前 。这是可以的，因为它是在知道的情况下实现的，所以它可以在委托之前处理这些实例。B <: AB.__radd__A.__add__AComplex

如果和没有分享任何其他知识，那么适当的共享操作就是内置的操作，并且两者都在那里，所以。A <: ComplexB <: Realcomplex__radd__()a+b == b+a

因为任何给定类型的大多数操作都非常相似，所以定义一个辅助函数会很有用，它可以生成任何给定操作符的正向和反向实例。例如，fractions.Fraction使用：

```
def _operator_fallbacks(monomorphic_operator, fallback_operator):
    def forward(a, b):
        if isinstance(b, (int, Fraction)):
            return monomorphic_operator(a, b)
        elif isinstance(b, float):
            return fallback_operator(float(a), b)
        elif isinstance(b, complex):
            return fallback_operator(complex(a), b)
        else:
            return NotImplemented
    forward.__name__ = '__' + fallback_operator.__name__ + '__'
    forward.__doc__ = monomorphic_operator.__doc__

    def reverse(b, a):
        if isinstance(a, Rational):
            # Includes ints.
            return monomorphic_operator(a, b)
        elif isinstance(a, numbers.Real):
            return fallback_operator(float(a), float(b))
        elif isinstance(a, numbers.Complex):
            return fallback_operator(complex(a), complex(b))
        else:
            return NotImplemented
    reverse.__name__ = '__r' + fallback_operator.__name__ + '__'
    reverse.__doc__ = monomorphic_operator.__doc__

    return forward, reverse
```

```
def _add(a, b):
    """a + b"""
    return Fraction(a.numerator * b.denominator +
                    b.numerator * a.denominator,
                    a.denominator * b.denominator)

__add__, __radd__ = _operator_fallbacks(_add, operator.add)

# ...
```

9.2。 math- 数学函数

该模块始终可用。它提供对由C标准定义的数学函数的访问。

这些功能不能用于复数; cmath如果您需要支持复杂数字，请使用模块中相同名称的功能。支持复数的功能和不支持的功能之间的区别是由于大多数用户不想学习理解复数所需的太多数学。接收一个异常而不是一个复杂的结果，可以更早地检测出用作参数的意外复数，这样程序员就可以确定首先产生的方式和原因。

以下功能由该模块提供。除非另外明确指出，否则所有返回值都是浮点数。

9.2.1。数论和表示函数

math.ceil (*x*)
返回*x*的最大值，*x*是大于或等于*x*的最小整数。如果*x*不是浮点数，则委托给x.__ceil__()，它应该返回一个 Integral值。

math.copysign (*x* , *y*)
返回一个浮点，其大小（绝对值）为*x*，但*y*的符号 。在支持带符号的零的平台上， 返回*-1.0*。copysign(1.0, -0.0)

math.fabs (*x*)
返回*x*的绝对值。

math.factorial (*x*)
返回*x*阶乘。ValueError如果*x*不是整数或者是负数，则引发。

math.floor (*x*)
返回*x*的最小整数，小于或等于*x*。如果*x*不是浮点数，则委托给x.__floor__()，它应该返回一个 Integral值。

math.fmod (*x* , *y*)
返回，由平台C库定义。请注意，Python表达式可能不会返回相同的结果。C标准的意图是精确地（数学上的;以无限的精度）等于某个整数*n*，使得结果的符号与*x*和幅度小于。Python的 返回结果会带有*y*的符号，并且对于float参数可能不是完全可计算的。例如，是的，但Python的结果是，它不能完全表示为一个浮点数，并且令人惊讶。出于这个原因，功能fmod(x, y)x % yfmod(x, y)x - n*yabs(y)x % yfmod(-1e-100, 1e100)-1e-100-1e-100 % 1e1001e100-1e-1001e100fmod()在使用浮点数时一般首选，而在使用整数时首选Python 。x % y

math.frexp (*x*)
返回*x*的尾数和指数作为一对。 *m*是一个浮点数，*e*是一个整数。如果*x*为零，则返回，否则返回。这用于以便携方式“分离”浮点的内部表示。(m, e)x == m * 2**e(0.0, 0)0.5 <= abs(m) < 1

math.**fsum**(*可迭代*)

在迭代器中返回一个精确的浮点和值。通过跟踪多个中间部分和来避免精度损失：

```
>>> sum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])
0.9999999999999999
>>> fsum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])
1.0
```

该算法的精度取决于IEEE-754算术保证和舍入模式为半均匀的典型情况。在一些非Windows版本中，底层C库使用扩展精度加法，偶尔会使中间和加倍，导致它在最低有效位中关闭。

有关进一步的讨论和两种替代方法，请参阅ASPN食谱配方以获得准确的浮点总和。

math.**gcd**(*a*, *b*)

返回整数*a*和*b*的最大公约数。如果 *a*或*b*不为零，那么值是分割*a*和*b*的最大正整数。 返回 。 `gcd(a, b)` `gcd(0, 0)` `0`

3.5版本中的新功能。

math.**isclose**(*a*, *b*, ***, *rel_tol = 1e-09*, *abs_tol = 0.0*)

返回`True`如果值一和*b*接近对方， `False`否则。

根据给定的绝对和相对公差确定两个值是否被认为是接近的。

*rel_tol*是相对容差 - 它是*a*和*b*之间允许的最大差值，相对于*a*或*b*的绝对值较大。例如，要设置5%的容差，请通过`rel_tol=0.05`。默认容差是`1e-09`，它确保这两个值在大约9位十进制数字内相同。 *rel_tol*必须大于零。

*abs_tol*是最小绝对容差 - 对于接近零的比较有用。*abs_tol*必须至少为零。

如果没有错误发生，结果将是： 。 `abs(a-b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)`

的IEEE 754特殊值`NaN`，`inf`以及`-inf`将根据IEEE规则处理。具体而言，`NaN`不被视为接近任何其他值，包括`NaN`。 `inf`并且`-inf`只被认为接近他们自己。

3.5版本中的新功能。

> **也可以看看：** **PEP 485** - 用于测试近似相等的函数

math.**isfinite**(*x*)

返回`True`如果*X*既不是无穷大也不是NaN和 `False`其他。（注意这`0.0` *被*认为是有限的。）

3.2版本中的新功能

math.**isinf**(*x*)

`True`如果*x*是正或负无穷大，则返回， `False`否则返回。

math.**isnan**(*x*)

返回True如果*X*为NaN（非数字），以及False其他。

math.**ldexp**(*x*, *i*)

返回。这基本上是函数的反函数。 `x * (2**i)` `frexp()`

math.**modf**(*x*)

返回*x*的小数部分和整数部分。两个结果都带有*x*的符号并且是浮点数。

math.**trunc**(*x*)

将截断 的Real值*x*返回Integral（通常是一个整数）。代表参加 `x.__trunc__()`。

注意，`frexp()`和`modf()`具有比它们的C当量的不同调用/返回的模式：它们采取单参数和返回的一对值，而不是通过“输出参数”返回其第二返回值（有在Python没有这样的事情）。

对于`ceil()`，`floor()`和`modf()`功能，请注意，*所有的* 足够大的幅度的浮点数字是准确的整数。Python浮点运算的精度通常不超过53位（与平台C的双精度类型相同），在这种情况下，任何有 必要的浮点数*x*都没有小数位。 `abs(x) >= 2**52`

9.2.2。功率和对数函数

math.**exp**(*x*)

返回`e**x`。

math.**expm1**(*x*)

返回。对于小浮点数*x*，减法 可导致精度的显着损失；该 函数提供了一种以完全精确的方式计算此数量的方法：`e**x - 1` `exp(x) - 1` `expm1()`

```
>>> from math import exp, expm1
>>> exp(1e-5) - 1  # gives result accurate to 11 places
1.0000050000069649e-05
>>> expm1(1e-5)    # result accurate to full precision
1.0000050000166668e-05
```

3.2版本中的新功能

math.**log**(*x*[, *base*])

用一个参数，返回*x*的自然对数（以*e*为底）。

用两个参数，将*x*的对数返回给定的*基数*，计算为`log(x)/log(base)`。

math.**log1p**(*x*)

返回*1 + x*的自然对数（基数*e*）。计算结果的方式对于接近零的*x*是准确的。

math.**log2**(*x*)

返回*x*的基数2的对数。这通常比.....更准确 。 `log(x, 2)`

3.3版本的新功能

也可以看看： int.bit_length() 返回表示二进制整数所需的位数，不包括符号和前导零。

math.log10(*x*)
返回*x*的基数为10的对数。这通常比……更准确。log(x, 10)

math.pow(*x*, *y*)
返回x提升到权力y。例外情况应尽可能遵循C99标准的附件'F'。特别是， 和总是返回，即使是零或为NaN。如果两个和是有限的， 是负的，而不是整数然后 是未定义的，并提高。pow(1.0, x) pow(x, 0.0) 1.0 x x y x y pow(x, y) ValueError

与内置**运算符不同，math.pow() 将其两个参数都转换为类型float。使用**或内置 pow() 函数来计算精确的整数幂。

math.sqrt(*x*)
返回*x*的平方根。

9.2.3。三角函数

math.acos(*x*)
返回*x*的反余弦，单位为弧度。

math.asin(*x*)
以弧度返回*x*的反正弦。

math.atan(*x*)
以弧度返回*x*的反正切。

math.atan2(*y*, *x*)
返回，以弧度表示。结果在和之间。从原点到点的平面中的矢量与正X轴形成此角度。重点是两个输入的符号都是已知的，所以它可以计算角度的正确象限。例如，并且都是，但就是。atan(y / x) -pi pi (x, y) atan2() atan(1) atan2(1, 1) pi/4 atan2(-1, -1) -3*pi/4

math.cos(*x*)
返回x弧度的余弦。

math.hypot(*x*, *y*)
返回欧几里得规范，。这是从原点到点的矢量长度。sqrt(x*x + y*y) (x, y)

math.sin(*x*)
返回x弧度的正弦值。

math.tan(*x*)
返回x弧度的切线。

9.2.4。角度转换

math.degrees(*x*)
将角度*x*从弧度转换为度数。

math.radians(*x*)
将角度*x*从度数转换为弧度。

9.2.5。双曲函数

双曲函数 是基于双曲线而不是圆的三角函数的类比。

math.acosh(*x*)
返回*x*的反双曲余弦。

math.asinh(*x*)
返回*x*的反双曲正弦。

math.atanh(*x*)
返回*x*的反双曲正切。

math.cosh(*x*)
返回*x*的双曲余弦。

math.sinh(*x*)
返回*x*的双曲正弦。

math.tanh(*x*)
返回*x*的双曲正切。

9.2.6。特殊功能

math.erf(*x*)
返回*x*处 的错误函数。

该erf()函数可用于计算传统统计函数，如累积标准正态分布：

```
def phi(x):
    'Cumulative distribution function for the standard normal distribution'
    return (1.0 + erf(x / sqrt(2.0))) / 2.0
```

3.2版本中的新功能

math.erfc(*x*)

返回*x*处的补充错误函数。的互补误差函数被定义为 。它用于较大的*x*值，其中从一个中减去会导致重要性的损失。`1.0 - erf(x)`

3.2版本中的新功能

math.**gamma**(*x*)
返回*x*处 的Gamma函数。

3.2版本中的新功能

math.**lgamma**(*x*)
返回*x*处Gamma函数绝对值的自然对数。

3.2版本中的新功能

9.2.7。常量

math.**pi**
数学常数π= 3.141592 ...，可用精度。

math.**e**
数学常数e = 2.718281 ...，可用精度。

math.**tau**
数学常数τ= 6.283185 ...，可用精度。Tau是一个等于2π的圆常量，即圆的周长与其半径之比。要了解更多关于Tau的信息，请查看Vi Hart的视频Pi（仍然）是错误的，并开始通过吃两倍的馅饼来庆祝 Tau日！

3.6版本中的新功能。

math.**inf**
浮点正无限。（对于负无穷大，使用 `-math.inf`。）等于输出`float('inf')`。

3.5版本中的新功能。

math.**nan**
浮点“不是数字”（NaN）值。相当于输出 `float('nan')`。

3.5版本中的新功能。

CPython实现细节：该`math`模块主要由平台C数学库函数周围的精简包装器组成。在特殊情况下的行为适用于C99标准附录F. 当前的实现将提高 `ValueError`对于像无效操作`sqrt(-1.0)`或`log(0.0)`（其中C99附件F建议信令无效操作或分频零），`OverflowError`对于结果溢出（例如，`exp(1000.0)`）。NaN不会从上述任何函数返回，除非一个或多个输入参数是NaN; 在这种情况下，大多数函数都会返回一个NaN，但是（在C99附录F之后），这个规则也有例外或者例外。`pow(float('nan'), 0.0)` `hypot(float('nan'), float('inf'))`
请注意，Python不会区分NaN和安静的NaN，并且信号NaNs的行为仍未明确。典型的行为是将所有NaN视为安静。

也可以看看:

模 cmath
: 许多这些功能的复数版本。

9.3。 cmath- 复数的数学函数

该模块始终可用。它提供对复数的数学函数的访问。该模块中的函数接受整数，浮点数或复数作为参数。它们还将接受任何具有a `__complex__()`或a `__float__()` 方法的Python对象：这些方法分别用于将对象转换为复数或浮点数，然后将该函数应用于转换结果。

注意： 在硬件和系统级支持平台符号的零，涉及分支机构削减功能是连续*两个*分支切割面：零的符号与其他分支切口的一侧区别。在不支持带符号零的平台上，连续性如下所述。

9.3.1。来自极坐标的转换

Python复数`z`使用*矩形*或*笛卡尔*坐标在内部存储。它完全由*其实部* `z.real`和*虚部决定* `z.imag`。换一种说法：

```
z == z.real + z.imag*1j
```

*极坐标*给出了表示复数的另一种方式。在极坐标中，复数*z*由模量*r*和相位角*φ定义*。模数*r*是从*z*到原点的距离，而相位*φ*是从正x轴到连接原点到*z的*线段以弧度测量的逆时针角度。

可以使用以下函数将本地直角坐标转换为极坐标并将其转换回。

`cmath.`**phase**`(x)`

返回的相位*X*（也被称为*自变量*的*X*），作为浮动。 `phase(x)`相当于。结果在[-π，π]范围内，并且此操作的分支切割位于负实轴上，从上面开始连续。在支持带符号零的系统（包括当前使用的大多数系统）上，这意味着结果的符号与符号相同，即使 为零：`math.atan2(x.imag, x.real)` `x.imag` `x.imag`

```
>>> phase(complex(-1.0, 0.0))
3.141592653589793
>>> phase(complex(-1.0, -0.0))
-3.141592653589793
```

注意： 复数*x*的模数（绝对值）可以使用内置`abs()`函数来计算。`cmath`此操作没有单独的模块功能。

`cmath.`**polar**`(x)`

返回极坐标中*x*的表示形式。返回一对，其中*r*是*x*的模数，phi是*x*的相位。 相当于。`(r, phi)` `polar(x)` `(abs(x), phase(x))`

`cmath.`**rect**`(r, phi)`

用极坐标*r*和*phi*返回复数*x*。相当于。`r * (math.cos(phi) + math.sin(phi)*1j)`

9.3.2。功率和对数函数

cmath.exp(*x*)
返回指数值e**x。

cmath.log(*x*[, *base*])
将*x*的对数返回给定的*基数*。如果未指定*基数*，则返回*x*的自然对数。有一个分支从0开始沿负实轴旋转到-∞，从上向下连续。

cmath.log10(*x*)
返回*x*的基数为10的对数。这个分支与之相同 log()。

cmath.sqrt(*x*)
返回*x*的平方根。这个分支与之相同log()。

9.3.3。三角函数

cmath.acos(*x*)
返回*x*的反余弦。有两个分支切割：一个从实轴向右延伸至∞，从下向下延伸。另一个从实轴-1从左向上延伸到-∞，从上面连续。

cmath.asin(*x*)
返回*x*的反正弦。这与之相同的分支切割acos()。

cmath.atan(*x*)
返回*x*的反正切。有两个分支切割：一个从1j虚轴延伸 到∞j右连续。另一个从-1j虚轴延伸到-∞j从左边连续。

cmath.cos(*x*)
返回*x*的余弦。

cmath.sin(*x*)
返回*x*的正弦值。

cmath.tan(*x*)
返回*x*的正切值。

9.3.4。双曲函数

cmath.acosh(*x*)
返回*x*的反双曲余弦。有一个分支切割，从实轴向左延伸到-∞，从上面连续。

cmath.asinh(*x*)
返回*x*的反双曲正弦。有两个分支切割：一个从1j虚轴延伸到∞j右连续。另一个从-1j虚轴延伸到-∞j从左边连续。

cmath.atanh(*x*)

返回*x*的反双曲正切。有两个分支切割：一个从1实轴延伸到下一个∞连续。另一个从-1实轴延伸到-∞从上面继续。

cmath.**cosh**(*x*)
返回*x*的双曲余弦。

cmath.**sinh**(*x*)
返回*x*的双曲正弦。

cmath.**tanh**(*x*)
返回*x*的双曲正切。

9.3.5。分类功能

cmath.**isfinite**(*x*)
返回True如果两个实部和虚部*X*是有限的， False否则。

3.2版本中的新功能

cmath.**isinf**(*x*)
返回True如果任一真实的或虚部*X*是一个无穷大，False否则。

cmath.**isnan**(*x*)
返回True如果任一真实的或虚部*X*是NaN，和False其它。

cmath.**isclose**(*a*，*b*，*，*rel_tol = 1e-09*，*abs_tol = 0.0*)
返回True如果值一和*b*接近对方， False否则。

根据给定的绝对和相对公差确定两个值是否被认为是接近的。

*rel_tol*是相对容差 - 它是*a*和*b*之间允许的最大差值，相对于*a*或*b*的绝对值较大。例如，要设置5%的容差，请通过rel_tol=0.05。默认容差是1e-09，它确保这两个值在大约9位十进制数字内相同。 *rel_tol*必须大于零。

*abs_tol*是最小绝对容差 - 对于接近零的比较有用。*abs_tol*必须至少为零。

如果没有错误发生，结果将是： 。abs(a-b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)

的IEEE 754特殊值NaN，inf以及-inf将根据IEEE规则处理。具体而言，NaN不被视为接近任何其他值，包括NaN。 inf并且-inf只被认为接近他们自己。

3.5版本中的新功能。

也可以看看： **PEP 485** - 用于测试近似相等的函数

9.3.6。常量

cmath.**pi**

数学常数π，作为一个浮点数。

cmath.**e**

数学常数e，作为一个浮点数。

cmath.**tau**

数学常数τ，作为一个浮点数。

3.6版本中的新功能。

cmath.**inf**

浮点正无限。相当于`float('inf')`。

3.6版本中的新功能。

cmath.**infj**

具有零实部和正无限虚部的复数。相当于。`complex(0.0, float('inf'))`

3.6版本中的新功能。

cmath.**nan**

浮点“不是数字”（NaN）值。相当于`float('nan')`。

3.6版本中的新功能。

cmath.**nanj**

具有零实部和NaN虚部的复数。相当于。`complex(0.0, float('nan'))`

3.6版本中的新功能。

请注意，功能的选择与模块中的功能类似，但不完全相同`math`。有两个模块的原因是一些用户对复数不感兴趣，甚至不知道它们是什么。他们宁愿`math.sqrt(-1)`举起一个例外，而不是返回一个复数。还要注意`cmath`，即使答案可以表示为实数（在这种情况下复数的虚数部分为零），定义的函数也会返回一个复数。

关于分支切割的注意事项：它们是给定函数无法连续的曲线。它们是许多复杂功能的必备功能。假设如果您需要使用复杂的函数进行计算，您将了解有关分支切割的信息。请教关于复杂变量的几乎任何（不太简单）的书籍。有关为数字目的正确选择分支切割的信息，请参考以下内容：

也可以看看： Kahan，W：分支削减复杂的基本功能;或者，关于什么都不是标志位。Iserles，A.和Powell，M.（eds。），数值分析的现状。Clarendon Press（1987）pp165-211。

9.4。 decimal- 十进制定点和浮点运算

源代码： Lib / decimal.py

该decimal模块支持快速正确舍入十进制浮点运算。它比float数据类型提供了几个优点 ：

- 十进制“是基于一个浮点模型，该模型是以人为本设计的，并且必须有一个最重要的指导原则 - 计算机必须提供一种与人们在学校学习的算术相同的算法。” - 摘自十进制算术规范。
- 十进制数字可以精确表示。相比之下，数字在二进制浮点中像 1.1和2.2没有精确的表示。最终用户通常不会像使用二进制浮点一样显示。 1.1 + 2.23.3000000000000003
- 精确性被转化为算术。在十进制浮点数中，正好等于零。在二进制浮点中，结果是。尽管接近于零，但差异阻止了可靠的平等测试，并且可能会积累差异。由于这个原因，在具有严格的等式不变量的会计应用中，小数是优选的。 0.1 + 0.1 + 0.1 - 0.35.551151231257827e-017
- 十进制模块包含了重要位置的概念，因此是。尾随零保持显示重要性。这是货币应用的惯例。对于乘法，“教科书”方法使用被乘数中的所有数字。举例来说，给人的同时给人。 1.30 + 1.202.501.3 * 1.21.561.30 * 1.201.5600
- 与基于硬件的二进制浮点不同，十进制模块具有用户可更改的精度（默认为28个位置），对于给定的问题，该精度可以与需要的一样大：

```
>>> from decimal import *
>>> getcontext().prec = 6
>>> Decimal(1) / Decimal(7)
Decimal('0.142857')
>>> getcontext().prec = 28
>>> Decimal(1) / Decimal(7)
Decimal('0.1428571428571428571428571429')
```

- 二进制和十进制浮点都是按照已发布的标准实现的。尽管内置的float类型只展示了它的功能中的一小部分，但十进制模块公开了标准的所有必需部分。在需要时，程序员可以完全控制舍入和信号处理。这包括通过使用例外来阻止任何不精确操作来强制执行精确算术的选项。
- 十进制模块被设计为支持“没有偏见，精确的未被占用的十进制算术（有时称为定点算术）和四舍五入的浮点算术。” - 摘自十进制算术规范。

模块设计以三个概念为中心：十进制数，算术环境和信号。

十进制数是不可变的。它有一个符号，系数数字和一个指数。为了保留重要性，系数数字不会截断尾随零。小数也包括特殊值，如 Infinity ，-Infinity和NaN。该标准还区分-0的+0。

算术环境是指定精度的环境，舍入规则，指数限制，指示操作结果的标志，以及确定信号是否被视为例外的陷阱启动器。舍入选项包括ROUND_CEILING ，ROUND_DOWN ， ROUND_FLOOR ，ROUND_HALF_DOWN ，ROUND_HALF_EVEN ， ROUND_HALF_UP ，ROUND_UP ，和ROUND_05UP。

信号是在计算过程中出现的一组特殊情况。根据应用的需要，信号可能被忽略，被视为信息性的，或被视为例外。十进制模块中的信号是：Clamped，InvalidOperation，DivisionByZero，Inexact，Rounded，Subnormal，Overflow，Underflow和FloatOperation。

每个信号都有一个标志和一个陷阱启动器。遇到信号时，其标志被设置为1，然后，如果陷阱启用码设置为1，则会引发异常。标志是粘性的，所以用户在监控计算之前需要重置它们。

也可以看看:

- IBM的通用十进制算术规范，通用十进制算术规范。

9.4.1。快速入门教程

通常开始使用小数将导入模块，查看当前上下文，getcontext()并在必要时为精度，舍入或启用的陷阱设置新值：

```
>>> from decimal import *
>>> getcontext()
Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
        capitals=1, clamp=0, flags=[], traps=[Overflow, DivisionByZero,
        InvalidOperation])

>>> getcontext().prec = 7       # Set a new precision
```

十进制实例可以用整数，字符串，浮点数或元组构造。从整数或浮点构造执行该整数或浮点的值的精确转换。十进制数字包括特殊值，例如 NaN“非数字”，正数和负数 Infinity，以及-0：

```
>>> getcontext().prec = 28
>>> Decimal(10)
Decimal('10')
>>> Decimal('3.14')
Decimal('3.14')
>>> Decimal(3.14)
Decimal('3.140000000000000124344978758017532527446746826171875')
>>> Decimal((0, (3, 1, 4), -2))
Decimal('3.14')
>>> Decimal(str(2.0 ** 0.5))
Decimal('1.4142135623730951')
>>> Decimal(2) ** Decimal('0.5')
Decimal('1.4142135623730950488016887242')
>>> Decimal('NaN')
Decimal('NaN')
>>> Decimal('-Infinity')
Decimal('-Infinity')
```

如果FloatOperation信号被捕获，构造函数中小数点和浮点数的意外混合或排序比较引发了一个例外：

```
>>> c = getcontext()
>>> c.traps[FloatOperation] = True
>>> Decimal(3.14)
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
decimal.FloatOperation: [<class 'decimal.FloatOperation'>]
>>> Decimal('3.5') < 3.7
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.FloatOperation: [<class 'decimal.FloatOperation'>]
>>> Decimal('3.5') == 3.5
True
```

3.3版本的新功能

新Decimal的意义仅由输入的位数决定。上下文精度和舍入仅在算术运算中发挥作用。

```
>>> getcontext().prec = 6
>>> Decimal('3.0')
Decimal('3.0')
>>> Decimal('3.1415926535')
Decimal('3.1415926535')
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal('5.85987')
>>> getcontext().rounding = ROUND_UP
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal('5.85988')
```

如果C版本的内部限制被超过，构造一个小数就会提高InvalidOperation：

```
>>> Decimal("1e9999999999999999999")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.InvalidOperation: [<class 'decimal.InvalidOperation'>]
```

在版本3.3中更改。

小数与Python的其余部分很好地交互。这里有一个小小数浮点飞行马戏团：

```
>>> data = list(map(Decimal, '1.34 1.87 3.45 2.35 1.00 0.03 9.25'.split()))
>>> max(data)
Decimal('9.25')
>>> min(data)
Decimal('0.03')
>>> sorted(data)
[Decimal('0.03'), Decimal('1.00'), Decimal('1.34'), Decimal('1.87'),
 Decimal('2.35'), Decimal('3.45'), Decimal('9.25')]
>>> sum(data)
Decimal('19.29')
>>> a,b,c = data[:3]
>>> str(a)
'1.34'
>>> float(a)
1.34
>>> round(a, 1)
Decimal('1.3')
>>> int(a)
1
>>> a * 5
```

```
Decimal('6.70')
>>> a * b
Decimal('2.5058')
>>> c % a
Decimal('0.77')
```

还有一些数学函数也可用于Decimal：

```
>>> getcontext().prec = 28
>>> Decimal(2).sqrt()
Decimal('1.4142135623730950488016887242')
>>> Decimal(1).exp()
Decimal('2.7182818284590452353602874713')
>>> Decimal('10').ln()
Decimal('2.3025850929940456840179914547')
>>> Decimal('10').log10()
Decimal('1')
```

该quantize()方法将数字四舍五入为固定的指数。这种方法对于货币应用程序非常有用，这些应用程序通常会将结果转换为固定数量的地方

```
>>> Decimal('7.325').quantize(Decimal('.01'), rounding=ROUND_DOWN)
Decimal('7.32')
>>> Decimal('7.325').quantize(Decimal('1.'), rounding=ROUND_UP)
Decimal('8')
```

如上所示，该getcontext()功能访问当前上下文并允许更改设置。这种方法可以满足大多数应用的需求。

对于更高级的工作，使用Context（）构造函数创建备用上下文可能很有用。要使备用激活，请使用该setcontext()功能。

根据该标准，该decimal模块提供了两个准备使用的标准上下文，BasicContext并且ExtendedContext。前者对于调试特别有用，因为许多陷阱都已启用：

```
>>> myothercontext = Context(prec=60, rounding=ROUND_HALF_DOWN)
>>> setcontext(myothercontext)
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857142857142857142857')

>>> ExtendedContext
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
        capitals=1, clamp=0, flags=[], traps=[])
>>> setcontext(ExtendedContext)
>>> Decimal(1) / Decimal(7)
Decimal('0.142857143')
>>> Decimal(42) / Decimal(0)
Decimal('Infinity')

>>> setcontext(BasicContext)
>>> Decimal(42) / Decimal(0)
Traceback (most recent call last):
  File "<pyshell#143>", line 1, in -toplevel-
    Decimal(42) / Decimal(0)
DivisionByZero: x / 0
```

上下文还有信号标志用于监控计算过程中遇到的异常情况。这些标志保持设置直到明确清除，所以最好在每一组监控计算之前使用该clear_flags()方法清除标志。

```
>>> setcontext(ExtendedContext)
>>> getcontext().clear_flags()
>>> Decimal(355) / Decimal(113)
Decimal('3.14159292')
>>> getcontext()
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
        capitals=1, clamp=0, flags=[Inexact, Rounded], traps=[])
```

该*标志*条目显示合理的近似Pi四舍五入（数字超出范围内精度被扔掉了），而且结果不准确（有些丢弃数字为非零）。

单个陷阱是使用traps上下文字段中的字典设置的：

```
>>> setcontext(ExtendedContext)
>>> Decimal(1) / Decimal(0)
Decimal('Infinity')
>>> getcontext().traps[DivisionByZero] = 1
>>> Decimal(1) / Decimal(0)
Traceback (most recent call last):
  File "<pyshell#112>", line 1, in -toplevel-
    Decimal(1) / Decimal(0)
DivisionByZero: x / 0
```

大多数程序只在程序开始时调整当前上下文一次。而且，在许多应用程序中，数据会Decimal在循环内转换为单个数据。通过创建上下文集和小数点，程序的大部分操作数据的方式与其他Python数字类型无异。

9.4.2。十进制对象

class decimal.Decimal（*value =“0”*，*context = None* ）

Decimal基于*价值*构建一个新的对象。

*值*可以是整数，字符串，元组float或其他Decimal 对象。如果没有给出*价值*，则返回Decimal('0')。如果*value*是一个字符串，它应该符合十进制数字字符串语法，前后的空格字符以及全部下划线将被删除：

```
sign           ::=  '+' | '-'
digit          ::=  '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
indicator      ::=  'e' | 'E'
digits         ::=  digit [digit]...
decimal-part   ::=  digits '.' [digits] | ['.'] digits
exponent-part  ::=  indicator [sign] digits
infinity       ::=  'Infinity' | 'Inf'
nan            ::=  'NaN' [digits] | 'sNaN' [digits]
numeric-value  ::=  decimal-part [exponent-part] | infinity
numeric-string ::=  [sign] numeric-value | [sign] nan
```

其他Unicode十进制数字也可以在digit 上面显示。这些包括来自各种其他字母表十进制数字与全角数字的沿（例如，阿拉伯-印度文和梵文位数）'\uff10'通过'\uff19'。

如果*值*是a tuple，它应该有三个组件，一个符号（0用于正数或1负数），一个tuple数字和一个整数指数。例如，退货。Decimal((0, (1, 4, 1, 4), -3))Decimal('1.414')

如果*值为*a float，则二进制浮点值将无损地转换为精确的十进制等值。这种转换通常需要53位或更多的精度。例如，Decimal(float('1.1'))转换为Decimal('1.100000000000000088817841970012523233890533447265625')。

该*背景*下的精度并不影响多少位被存储。这是由在数字位数专门确定*值*。例如，Decimal('3.00000')即使上下文精度只有三个，也要记录所有五个零。

*上下文*参数的目的是确定如果*value*是格式错误的字符串该怎么办。如果上下文陷阱InvalidOperation，则会引发异常; 否则，构造函数将返回一个新的Decimal，其值为NaN。

一旦构建，Decimal对象是不可变的。

*在版本3.2中更改：*构造函数的参数现在被允许为float 实例。

*在版本3.3中更改：*float如果FloatOperation 设置了陷阱，则会引发异常。默认情况下，陷阱关闭。

*在版本3.6中更改：可以使用*下划线进行分组，如使用代码中的积分和浮点文字。

十进制浮点对象与其他内置数字类型（如float和）共享许多属性int。所有常用的数学运算和特殊方法都适用。同样，小数对象可以被复制，腌制，打印，用作字典键，用作设置元素，比较，排序和强制到另一种类型（如float或 int）。

在小数对象上的算术和整数和浮点数上的算术之间有一些小的差异。当余数运算符%应用于十进制目的，结果的符号是的符号 *被除数*而不是除数的符号：

```
>>> (-7) % 4
1
>>> Decimal(-7) % Decimal(4)
Decimal('-3')
```

整数除法运算符的//行为类似，返回真商的整数部分（截断为零）而不是其底，以保持通常的身份：x == (x // y) * y + x % y

```
>>> -7 // 4
-2
>>> Decimal(-7) // Decimal(4)
Decimal('-1')
```

的%和//运营商实现remainder和 divide-integer操作（分别）如说明书中所述。

十进制对象通常不能与浮点数或fractions.Fraction算术运算的实例组合：例如，尝试将一个添加Decimal到a float，会引发一个TypeError。但是，可以使用Python的比较运算符将Decimal 实例x与另一个数字进行比较y。这样可以避免在不同类型的数字之间进行相等比较时出现令人困惑的结果。

*在版本3.2中更改：*Decimal现在完全支持实例和其他数字类型之间的混合类型比较。

除了标准的数字属性外，十进制浮点对象还有许多特殊的方法：

adjusted ( )
移出系数最右边的数字后，返回调整后的指数，直到只剩下前导数字：Decimal('321e+5').adjusted()返回7。用于确定最重要数字相对于小数点的位置。

as_integer_ratio ( )
将一组代表给定实例的整数 作为分数返回，以最小值和正分母表示：(n, d) Decimal

```
>>> Decimal('-3.14').as_integer_ratio()
(-157, 50)
```

转换是确切的。在NaN上引发溢出错误和ValueError溢出错误。

3.6版本中的新功能。

as_tuple ( )
返回数字的命名元组表示形式： 。DecimalTuple(sign, digits, exponent)

canonical ( )
返回参数的规范编码。目前，一个Decimal实例的编码总是规范的，所以这个操作返回它的参数不变。

compare (*其他*, *上下文=无*)
比较两个Decimal实例的值。 compare()返回一个Decimal实例，如果其中一个操作数是NaN，那么结果是NaN：

```
a or b is a NaN  ==> Decimal('NaN')
a < b            ==> Decimal('-1')
a == b           ==> Decimal('0')
a > b            ==> Decimal('1')
```

compare_signal (*其他*, *上下文=无*)
compare()除了所有的NaNs信号之外，该操作与该方法相同。也就是说，如果两个操作数都不是一个NaN信号，那么任何安静的NaN操作数都被视为一个信号NaN。

compare_total (*其他*, *上下文=无*)
使用它们的抽象表示比较两个操作数，而不是它们的数值。与该compare()方法类似，但结果给出了Decimal实例的总排序。Decimal具有相同数值但不同表示的两个 实例在此顺序中比较不等：

```
>>> Decimal('12.0').compare_total(Decimal('12'))
Decimal('-1')
```

安全和信令NaNs也包括在总的顺序中。这个函数的结果是，Decimal('0')如果两个操作数具有相同的表示形式，Decimal('-1')如果第一个操作数在总顺序中低于第二个，

并且Decimal('1')第一个操作数在总顺序中高于第二个操作数。有关总订单的详细信息，请参阅规格。

该操作不受上下文影响，并且安静：没有标志被改变，也没有执行舍入。作为例外，如果第二个操作数无法完全转换，则C版本可能会引发InvalidOperation。

compare_total_mag (*其他*, *上下文=无*)

比较两个操作数使用它们的抽象表示而不是它们的值compare_total() ，但忽略每个操作数的符号。 x.compare_total_mag(y) 相当于 x.copy_abs().compare_total(y.copy_abs())。

该操作不受上下文影响，并且安静：没有标志被改变，也没有执行舍入。作为例外，如果第二个操作数无法完全转换，则C版本可能会引发InvalidOperation。

conjugate ( )

只是返回自我，这种方法只符合十进制规范。

copy_abs ( )

返回参数的绝对值。此操作不受上下文影响，并且很安静：没有标志被更改，也没有执行舍入。

copy_negate ( )

返回参数的否定。此操作不受上下文影响，并且很安静：没有标志被更改，也没有执行舍入。

copy_sign (*其他*, *上下文=无*)

返回第一个操作数的副本，并将符号设置为与第二个操作数的符号相同。例如：

```
>>> Decimal('2.3').copy_sign(Decimal('-1.5'))
Decimal('-2.3')
```

该操作不受上下文影响，并且安静：没有标志被改变，也没有执行舍入。作为例外，如果第二个操作数无法完全转换，则C版本可能会引发InvalidOperation。

exp (*context = None*)

返回e**x给定数字处的（自然）指数函数的值。结果使用ROUND_HALF_EVEN舍入模式正确舍 入。

```
>>> Decimal(1).exp()
Decimal('2.718281828459045235360287471')
>>> Decimal(321).exp()
Decimal('2.561702493119680037517373933E+139')
```

from_float (*f*)

精确地将float转换为十进制数的Classmethod。

注意*Decimal.from_float（0.1）与Decimal（'0.1'）不同*。由于0.1在二进制浮点中不能完全表示，因此该值存储为最接近的可表示值，即 *0x1.999999999999ap-4*。该十进制等效值是 *0.1000000000000000055511151231257827021181583404541015625*。

注意： 从Python 3.2开始，一个Decimal实例也可以直接从a构建float。

```
>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')
>>> Decimal.from_float(float('nan'))
Decimal('NaN')
>>> Decimal.from_float(float('inf'))
Decimal('Infinity')
>>> Decimal.from_float(float('-inf'))
Decimal('-Infinity')
```

版本3.1中的新功能。

fma (*其他*，*第三*，*上下文=无*)

融合乘加。返回自己*其他+第三没有舍去中间产品自己*其他。

```
>>> Decimal(2).fma(3, 5)
Decimal('11')
```

is_canonical ( )

True如果参数是规范的，False 否则返回。目前，一个Decimal实例总是规范的，所以这个操作总是返回True。

is_finite ( )

返回True如果参数是一个有限数量，以及 False如果参数为无穷大或为NaN。

is_infinite ( )

返回True如果参数为正或负无穷大，False否则。

is_nan ( )

返回True如果参数是（安静或信令）的NaN和 False其它。

is_normal (*context = None*)

True如果参数是一个*正常的*有限数字，则返回。返回 False如果参数为0，低于正常，无穷大或NaN的。

is_qnan ( )

True如果参数是安静的NaN，则返回， False否则返回。

is_signed ( )

True如果参数有负号False则返回， 否则返回。请注意，零和NaN都可以携带符号。

is_snan ( )

返回True如果参数信令楠False 否则。

is_subnormal (*context = None*)

True如果参数是低于正常的，则返回，False 否则返回。

is_zero ( )

返回True如果参数为一（正或负）和零 False否则。

ln（*context = None* ）

返回操作数的自然对数（基数e）。结果使用ROUND_HALF_EVEN舍入模式正确舍入。

log10（*context = None* ）

返回操作数的十进制对数。结果使用ROUND_HALF_EVEN舍入模式正确舍入。

logb（*context = None* ）

对于非零数字，返回其操作数的调整指数作为 Decimal实例。如果操作数为零，则Decimal('-Infinity')返回该DivisionByZero标志并引发该标志。如果操作数是无穷大，则Decimal('Infinity')返回。

logical_and（*其他*，*上下文=无*）

logical_and()是一个逻辑操作，它需要两个*逻辑操作数*（请参阅逻辑操作数）。结果是and两个操作数的数字方式。

logical_invert（*context = None* ）

logical_invert()是一种逻辑运算。结果是操作数的数字反转。

logical_or（*其他*，*上下文=无*）

logical_or()是一个逻辑操作，它需要两个*逻辑操作数*（请参阅逻辑操作数）。结果是or两个操作数的数字方式。

logical_xor（*其他*，*上下文=无*）

logical_xor()是一个逻辑操作，它需要两个*逻辑操作数*（请参阅逻辑操作数）。结果是数字独占或两个操作数。

max（*其他*，*上下文=无*）

像不同的是在返回之前和施加的上下文舍入规则值用信号通知或忽略（取决于上下文以及它们是否信令或安静）。 max(self, other)NaN

max_mag（*其他*，*上下文=无*）

与该max()方法类似，但使用操作数的绝对值进行比较。

min（*其他*，*上下文=无*）

像不同的是在返回之前和施加的上下文舍入规则值用信号通知或忽略（取决于上下文以及它们是否信令或安静）。 min(self, other)NaN

min_mag（*其他*，*上下文=无*）

与该min()方法类似，但使用操作数的绝对值进行比较。

next_minus（*context = None* ）

返回给定上下文中的最大数字（或者在当前线程的上下文中，如果没有给出上下文），该数字小于给定的操作数。

next_plus（*context = None* ）

返回大于给定操作数的给定上下文（或者在当前线程的上下文中，如果没有给出上下文）中可表示的最小数字。

next_toward（*其他*，*上下文=无*）

如果两个操作数不相等，则在第二个操作数的方向上返回最接近第一个操作数的数字。如果两个操作数在数值上相等，则返回第一个操作数的副本，并将符号设置为与第二个操作数的符号相同。

normalize（*context = None*）

通过汽提最右边的尾随零和转换等于任何结果归一化数 Decimal('0') 到 Decimal('0e0')。用于为等价类的属性生成规范值。例如，Decimal('32.100') 与 Decimal('0.321000e+2') 两个正常化为等效值 Decimal('32.1')。

number_class（*context = None*）

返回描述操作数*类*的字符串。返回的值是以下十个字符串之一。

- "-Infinity"，表明操作数是负无穷大。
- "-Normal"，表明该操作数是一个负的正常数。
- "-Subnormal"表明操作数是负数和低于正常的。
- "-Zero"，表明操作数是负零。
- "+Zero"，表明操作数是一个正的零。
- "+Subnormal"，表明操作数是正的和低于正常的。
- "+Normal"，表示操作数是正数。
- "+Infinity"，表明操作数是正无穷。
- "NaN"，表明操作数是一个安静的NaN（不是数字）。
- "sNaN"，表明操作数是一个信号NaN。

quantize（*exp*，*rounding = None*，*context = None*）

在舍入后返回一个等于第一个操作数并具有第二个操作数的指数的值。

```
>>> Decimal('1.41421356').quantize(Decimal('1.000'))
Decimal('1.414')
```

与其他操作不同，如果量化操作之后的系数长度大于精度，则会InvalidOperation发出 an 信号。这保证了，除非有错误条件，否则量化指数总是等于右侧操作数的指数。

与其他操作不同，即使结果是次正常且不精确，量化也不会发出下溢信号。

如果第二个操作数的指数大于第一个操作数的指数，则可能需要舍入。在这种情况下，舍入模式由rounding参数确定，如果给出，则由给定 context参数确定; 如果没有给出参数，则使用当前线程上下文的舍入模式。

每当结果指数大于Emax或小于时，都会返回错误 Etiny。

radix（）

返回Decimal(10)，Decimal 类中所有算术的基数（基数）。包括与规范的兼容性。

remainder_near（*其他*，*上下文=无*）

从分返回，其余*自*通过*其他*。这不同于 其余部分的符号被选择为使其绝对值最小化。更确切地说，返回值是 其中最接近的精确值的整数，如果两个整数都同样接近，则即

使一个选择。self % otherself - n * othernself / other

如果结果为零，那么它的符号将是*自我*的标志。

```
>>> Decimal(18).remainder_near(Decimal(10))
Decimal('-2')
>>> Decimal(25).remainder_near(Decimal(10))
Decimal('5')
>>> Decimal(35).remainder_near(Decimal(10))
Decimal('-5')
```

rotate (*其他*，*上下文=无*)

将第一个操作数的数字旋转一个由第二个操作数指定的量的结果。第二个操作数必须是精度范围-precision中的整数。第二个操作数的绝对值给出了要旋转的地方的数量。如果第二个操作数是正的，那么旋转是在左边; 否则旋转是正确的。如有必要，第一个操作数的系数在左侧填充为零以达到长度精度。第一个操作数的符号和指数保持不变。

same_quantum (*其他*，*上下文=无*)

测试自我和其他人是否有相同的指数或两者是否都是 NaN。

该操作不受上下文影响，并且安静：没有标志被改变，也没有执行舍入。作为例外，如果第二个操作数无法完全转换，则C版本可能会引发InvalidOperation。

scaleb (*其他*，*上下文=无*)

返回第一个操作数，指数由第二个调整。等价地，返回第一个操作数乘以10**other。第二个操作数必须是整数。

shift (*其他*，*上下文=无*)

返回将第一个操作数的数位移位第二个操作数指定的数值的结果。第二个操作数必须是精度范围-precision中的整数。第二个操作数的绝对值给出了要移位的位数。如果第二个操作数是正的，那么这个移位是在左边; 否则这个转变是在右边。移入系数的数字是零。第一个操作数的符号和指数保持不变。

sqrt (*context = None*)

将参数的平方根返回到完全精度。

to_eng_string (*context = None*)

如果需要指数，则转换为字符串，使用工程符号。

工程符号的指数是3的倍数。这可以在小数点左边留下最多3位数字，并且可能需要添加一个或两个尾随零。

例如，这转换Decimal('123E+1')为Decimal('1.23E+3')。

to_integral (*四舍五入=无*，*上下文=无*)

与该to_integral_value()方法相同。该to_integral 名称一直保持与旧版本的兼容性。

to_integral_exact (*四舍五入=无*，*上下文=无*)

四舍五入到最接近的整数，发信号Inexact或 Rounded酌情发生舍入。舍入模式由rounding给定的参数确定，否则由给定的参数 确定context。如果没有给出参数，则使用当前上下文的舍入模式。

to_integral_value (*四舍五入=无*, *上下文=无*)
四舍五入到最接近的整数，无需信号Inexact或 Rounded。如果给出，则适用*四舍五入*；否则，在提供的*上下文*或当前上下文中使用舍入方法。

9.4.2.1。逻辑操作数

在logical_and() ，logical_invert() ，logical_or() ，和logical_xor()方法，希望他们的论点是*合乎逻辑的操作数*。一个*合乎逻辑的操作数*是一个Decimal实例，其指数和符号均为零，而其数字都是要么 0或1。

9.4.3。上下文对象

上下文是算术运算的环境。它们控制精度，设置舍入规则，确定哪些信号被视为例外，并限制指数的范围。

每个线程都有自己的当前上下文，可以使用getcontext()和setcontext()函数访问或更改它们：

decimal.getcontext ( )
返回活动线程的当前上下文。

decimal.setcontext (*c*)
将活动线程的当前上下文设置为*c*。

您也可以使用with语句和localcontext() 函数临时更改活动上下文。

decimal.localcontext (*ctx = None*)
返回一个上下文管理器，它将活动线程的当前上下文设置为进入with-statement时的*ctx*副本，并在退出with-statement时恢复上一个上下文。如果没有指定上下文，则使用当前上下文的副本。

例如，以下代码将当前小数精度设置为42位，执行计算，然后自动恢复以前的上下文：

```
from decimal import localcontext

with localcontext() as ctx:
    ctx.prec = 42   # Perform a high precision calculation
    s = calculate_something()
s = +s  # Round the final result back to the default precision
```

新的上下文也可以使用Context下面描述的构造函数创建。另外，该模块提供了三个预先制定的上下文：

*类*decimal.BasicContext

这是通用十进制算术规范定义的标准上下文。精度设置为9。舍入被设置为ROUND_HALF_UP。所有标志都被清除。所有的陷阱启用（例外处理）除外Inexact，Rounded以及 Subnormal。

由于许多陷阱都已启用，因此此上下文对调试很有用。

*类*decimal.ExtendedContext

这是通用十进制算术规范定义的标准上下文。精度设置为9。舍入被设置为ROUND_HALF_EVEN。所有标志都被清除。没有陷阱被启用（以便在计算期间不会引发异常）。

由于陷阱被禁用，因此此上下文对于偏好具有结果值NaN或Infinity不是引发异常的应用程序非常有用。这允许应用程序在存在条件的情况下完成运行，否则该条件会暂停程序。

*类*decimal.DefaultContext

这个上下文被Context构造函数用作新上下文的原型。更改一个字段（如精度）会改变Context构造函数创建的新上下文的默认值。

这个上下文在多线程环境中非常有用。在启动线程之前更改其中一个字段具有设置系统范围内默认值的效果。不建议在线程启动后更改字段，因为它需要线程同步来防止竞争条件。

在单线程环境中，最好不要使用这个上下文。相反，只需按照下面的描述直接创建上下文。

默认值是prec= 28 ， rounding= ROUND_HALF_EVEN ，并启用陷阱Overflow ，InvalidOperation和 DivisionByZero。

除了三个提供的上下文之外，还可以使用Context构造函数创建新的上下文 。

class decimal.Context（*prec = None*，*四舍五入=无*，*Emin =无*，*Emax =无*，*大写=无*，*clamp =无*，*flags =无*，*陷阱=无*）

创建一个新的上下文。如果一个字段没有被指定或者是None ，那么缺省值将被复制DefaultContext。如果*标志*字段没有被指定或者是None，则所有标志都被清除。

*prec*是范围[1 ，MAX_PREC] 中的一个整数，用于设置上下文中算术运算的精度。

*舍入*选项是“ *舍入模式* ”一节中列出的常量之一 。

该*陷阱*和*标志*字段列出任何信号进行设置。通常情况下，新的上下文应该只设置陷阱并且清除标志。

的*额敏*和*的Emax*字段指定允许的指数外部界限整数。*Emin*必须在[MIN_EMIN ，0] 范围内的*Emax*范围内，[0 ，MAX_EMAX]。

的*首都*字段是0或1（缺省值）。如果设置为 1 ，指数用大写打印E;否则，使用小写字母e ：Decimal('6.02e+23')。

所述*夹具*字段是0（缺省值）或1。如果设置为1 ，则 在此上下文中可表示e的Decimal实例的指数严格限于该范围。如果*钳制*是 一个较弱的条件成立：实例的调整指数最多。当*钳*是

，大量正常数量将在可能情况下，具有其指数降低，并且添加到其系数的零的相应数量的，为了适应指数限制; 这保留了数字的值，但丢失了关于重要尾随零的信息。例如：Emin - prec + 1 <= e <= Emax - prec + 1 0 Decimal Emax 1

```
>>> Context(prec=6, Emax=999, clamp=1).create_decimal('1.23e999')
Decimal('1.23000E+999')
```

甲*钳位*的值1允许与IEEE 754中指定的固定宽度的小数交换格式的兼容性。

在Context类定义了几个通用的方法以及大量直接在特定情况下做算术的方法。另外，对于每个Decimal上述（具有的异常的方法adjusted()和as_tuple()方法）有相应的Context方法。例如，对于一个Context 实例C和Decimal实例x，C.exp(x)相当于x.exp(context=C)。每个Context方法在接受int Decimal实例的任何地方接受一个Python整数（的一个实例）。

clear_flags ( )
: 将所有标志重置为0。

clear_traps ( )
: 将所有陷阱重置为0。

3.3版本的新功能

copy ( )
: 返回上下文的副本。

copy_decimal (*num*)
: 返回Decimal实例num的副本。

create_decimal (*num*)
: 从*num*创建一个新的Decimal实例，但使用*self*作为上下文。与Decimal构造函数不同，上下文精度，舍入方法，标志和陷阱应用于转换。

这很有用，因为常量的精度通常比应用程序所需要的要高。另一个好处是，四舍五入可以消除超出当前精度的数字的意外影响。在以下示例中，使用未接地输入意味着将零加到和可以改变结果：

```
>>> getcontext().prec = 3
>>> Decimal('3.4445') + Decimal('1.0023')
Decimal('4.45')
>>> Decimal('3.4445') + Decimal(0) + Decimal('1.0023')
Decimal('4.44')
```

此方法实现了IBM规范的定数操作。如果参数是一个字符串，则不允许使用前导或尾随空格或下划线。

create_decimal_from_float (*f*)
: 从float *f*创建新的Decimal实例，但使用*self*作为上下文进行舍入。与Decimal.from_float()类方法不同，上下文精度，舍入方法，标志和陷阱应用于转换。

```
>>> context = Context(prec=5, rounding=ROUND_DOWN)
>>> context.create_decimal_from_float(math.pi)
Decimal('3.1415')
>>> context = Context(prec=5, traps=[Inexact])
>>> context.create_decimal_from_float(math.pi)
Traceback (most recent call last):
    ...
decimal.Inexact: None
```

版本3.1中的新功能。

Etiny ( )

返回一个值，该值等于次正常结果的最小指数值。发生下溢时，指数设置为。Emin - prec + 1Etiny

Etop ( )

返回等于的值。Emax - prec + 1

使用小数的常用方法是创建Decimal 实例，然后应用在活动线程的当前上下文中进行的算术运算。另一种方法是在上下文中使用上下文方法进行计算。这些方法与Decimal班级的方法类似，只是在这里简要叙述。

abs (*x*)

返回*x*的绝对值。

add (*x* , *y*)

返回*x*和*y*的和。

canonical (*x*)

返回相同的Decimal对象*x*。

compare (*x* , *y*)

数字比较*x*和*y*。

compare_signal (*x* , *y*)

以数字方式比较两个操作数的值。

compare_total (*x* , *y*)

使用抽象表示法比较两个操作数。

compare_total_mag (*x* , *y*)

使用它们的抽象表示比较两个操作数，忽略符号。

copy_abs (*x*)

返回符号设置为0 的*x*的副本。

copy_negate (*x*)

返回符号倒置的*x*的副本。

copy_sign (*x* , *y*)

将标志从*y*复制到*x*。

divide (*x* , *y*)
返回*x*除以*y*。

divide_int (*x* , *y*)
返回*x*除以*y*，截断为整数。

divmod (*x* , *y*)
将两个数字相除并返回结果的整数部分。

exp (*x*)
返回*e* ** *x*。

fma (*x* , *y* , *z*)
返回*x*乘以*y*，再加上*z*。

is_canonical (*x*)
True如果*x*是规范的，则返回; 否则返回False。

is_finite (*x*)
True如果*x*是有限的，则返回; 否则返回False。

is_infinite (*x*)
True如果*x*是无限的，则返回; 否则返回False。

is_nan (*x*)
True如果*x*是qNaN或sNaN，则返回; 否则返回False。

is_normal (*x*)
True如果*x*是正常数字，则返回; 否则返回False。

is_qnan (*x*)
True如果*x*是安静的NaN，则返回; 否则返回False。

is_signed (*x*)
True如果*x*是负数，则返回; 否则返回False。

is_snan (*x*)
True如果*x*是信号NaN，则返回; 否则返回False。

is_subnormal (*x*)
返回True如果*X*是低于正常; 否则返回False。

is_zero (*x*)
True如果*x*是零，则返回; 否则返回False。

ln (*x*)
返回*x*的自然对数（基数e）。

log10 (*x*)
返回*x*的基数10的对数。

logb (*x*)
返回操作数MSD大小的指数。

logical_and (*x* , *y*)
应用逻辑操作*以及*每个操作数的数字。

logical_invert (*x*)
反转*x中的*所有数字。

logical_or (*x* , *y*)
应用逻辑运算*或*每个操作数的数字。

logical_xor (*x* , *y*)
在每个操作数的数字之间应用逻辑运算*xor*。

max (*x* , *y*)
数字比较两个值并返回最大值。

max_mag (*x* , *y*)
将数值与它们的符号忽略进行比较。

min (*x* , *y*)
数字比较两个值并返回最小值。

min_mag (*x* , *y*)
将数值与它们的符号忽略进行比较。

minus (*x*)
减号对应于Python中的一元前缀减运算符。

multiply (*x* , *y*)
返回*x*和*y*的乘积。

next_minus (*x*)
返回小于*x*的最大可表示数。

next_plus (*x*)
返回大于*x*的最小可表示数。

next_toward (*x* , *y*)
返回数字最接近*X*向，方向*ÿ*。

normalize (*x*)
将*x*简化为最简单的形式。

number_class (*x*)

返回*x*的类的指示。

plus (*x*)

Plus对应于Python中的一元前缀加运算符。此操作应用上下文精度和舍入，因此它*不是*身份验证操作。

power (*x* , *y* , *modulo = None*)

如果给定，则返回x到y模的减少模modulo。

有两个参数，计算x**y。如果x是负数，那么y 必须是不可分割的 除非y是积分并且结果是有限的，并且可以精确地用“精确”数字表示，结果将是不精确的。使用上下文的舍入模式。结果总是在Python版本中正确舍入。

版本3.3中更改： C模块根据power()正确舍入 exp()和ln()函数进行计算。结果是明确的，但只有“几乎总是正确的圆整”。

有三个参数，计算。对于三个参数表单，参数的以下限制保留：(x**y) % modulo

- 所有三个参数都必须是不可分割
- y 必须是非负的
- 至少有一个x或y必须是非零
- modulo 必须非零，并且至多有'精度'数字

由此产生的值等于通过以无限精度进行计算所获得的值，但计算效率更高。结果的指数为零，不管指数是多少，和。结果总是确切的。 Context.power(x, y, modulo) (x**y) % modulo x y modulo

quantize (*x* , *y*)

返回一个等于*x*（四舍五入）的值，指数为*y*。

radix ( )

只需返回10，因为这是Decimal，:)

remainder (*x* , *y*)

返回整数除法的余数。

结果的符号（如果非零）与原始股息的符号相同。

remainder_near (*x* , *y*)

返回值，其中*n*是最接近精确值的整数（如果结果为0，则其符号将为*x*的符号）。 x - y * n x / y

rotate (*x* , *y*)

返回*x*，*y*次的旋转副本。

same_quantum (*x* , *y*)

True如果两个操作数具有相同的指数，则返回。

scaleb (*x* , *y*)

在添加exp的第二个值后返回第一个操作数。

shift (*x* , *y*)
返回*x*，*y*次的移位副本。

sqrt (*x*)
上下文精度的非负数的平方根。

subtract (*x* , *y*)
返回*x*和*y*之间的差异。

to_eng_string (*x*)
如果需要指数，则转换为字符串，使用工程符号。

工程符号的指数是3的倍数。这可以在小数点左边留下最多3位数字，并且可能需要添加一个或两个尾随零。

to_integral_exact (*x*)
舍入为整数。

to_sci_string (*x*)
使用科学记数法将数字转换为字符串。

9.4.4。常量

本节中的常量仅与C模块相关。它们也包含在纯Python版本中以兼容。

	32位	**64位**
decimal.MAX_PREC	425000000	999999999999999999
decimal.MAX_EMAX	425000000	999999999999999999
decimal.MIN_EMIN	-425000000	-999999999999999999
decimal.MIN_ETINY	-849999999	-1999999999999999997

decimal.HAVE_THREADS
默认值是True。如果Python编译时没有线程，C版本会自动禁用昂贵的线程本地上下文机制。在这种情况下，值是False。

9.4.5。舍入模式

decimal.ROUND_CEILING
往前走Infinity。

decimal.ROUND_DOWN
向零回合。

decimal.ROUND_FLOOR
往前走-Infinity。

decimal.ROUND_HALF_DOWN
将关系向最接近零的方向调整。

decimal.ROUND_HALF_EVEN
以最接近的偶数整数将关系舍入到最近。

decimal.ROUND_HALF_UP
离领带最近的距离为零。

decimal.ROUND_UP
从零开始回合。

decimal.ROUND_05UP
如果舍入到零后的最后一位数字为0或5，则从零开始舍去; 否则向零回合。

9.4.6。信号

信号代表计算过程中出现的条件。每个对应于一个上下文标志和一个上下文陷阱启用器。

上下文标志在遇到条件时设置。在计算之后，可以检查标志以用于信息目的（例如，以确定计算是否准确）。检查标志后，确保在开始下一次计算之前清除所有标志。

如果为信号设置了上下文的陷阱启用码，则该条件会引发Python异常。例如，如果DivisionByZero设置了陷阱，则DivisionByZero在遇到该情况时会引发异常。

*类*decimal.Clamped
改变了指数以适应表示限制。

通常，当指数超出上下文Emin和Emax限制范围时，会发生钳位。如果可能的话，通过给系数加零来使指数减小到合适的值。

*类*decimal.DecimalException
其他信号的基类和一个子类ArithmeticError。

*类*decimal.DivisionByZero
用零表示非无限数的划分。

可能发生在划分，模块划分或将某个数字提升至负值时。如果该信号未被捕获，则返回Infinity或者 -Infinity通过计算输入确定的符号。

*类*decimal.Inexact
表示发生舍入且结果不准确。

舍入时丢弃非零数字的信号。四舍五入的结果被返回。信号标志或陷阱用于检测结果是否不准确。

*类*decimal.**InvalidOperation**

执行了无效的操作。

表示请求的操作没有意义。如果没有被困，返回NaN。可能的原因包括：

```
Infinity - Infinity
0 * Infinity
Infinity / Infinity
x % 0
Infinity % x
sqrt(-x) and x > 0
0 ** 0
x ** (non-integer)
x ** Infinity
```

*类*decimal.**Overflow**

数值溢出。

指示指数大于Emax发生舍入后的指数。如果没有陷入，结果取决于舍入模式，要么向内拉到最大可表示的有限数字，要么向外舍入Infinity。在这两种情况下，Inexact并Rounded也发出信号。

*类*decimal.**Rounded**

尽管可能没有信息丢失，但发生了舍入。

每当四舍五入舍弃数字时发信号通知; 即使这些位是零（如舍入5.00到5.0）。如果未被捕获，则返回结果不变。该信号用于检测有效数字的丢失。

*类*decimal.**Subnormal**

指数比Emin四舍五入之前更低。

当运算结果不正常时（指数太小）发生。如果未被捕获，则返回结果不变。

*类*decimal.**Underflow**

数值下溢，结果舍入为零。

通过舍入将低于正常值的结果推到零时发生。Inexact 并Subnormal发出信号。

*类*decimal.**FloatOperation**

为混合浮点数和小数点启用更严格的语义。

如果信号未被捕获（默认），则在Decimal构造函数 create_decimal()和所有比较运算符中允许混合浮点数和小数点。转换和比较都是确切的。通过FloatOperation在上下文标志中进行设置，可以静默记录混合操作的任何发生。使用显式转换from_float() 或create_decimal_from_float()不设置标志。

否则（信号被捕获），只有平等比较和显式转换是无声的。所有其他混业经营提高FloatOperation。

下表总结了信号的层次结构：

```
exceptions.ArithmeticError(exceptions.Exception)
    DecimalException
        Clamped
        DivisionByZero(DecimalException, exceptions.ZeroDivisionError)
        Inexact
            Overflow(Inexact, Rounded)
            Underflow(Inexact, Rounded, Subnormal)
        InvalidOperation
        Rounded
        Subnormal
        FloatOperation(DecimalException, exceptions.TypeError)
```

9.4.7。浮点注释

9.4.7.1。以更高的精度减少舍入误差

使用十进制浮点消除了十进制表示错误（可以0.1精确表示）；但是，当非零数字超过固定精度时，某些操作仍然会产生舍入误差。

舍入误差的影响可以通过增加或减少近似偏移量来放大，从而导致显着性的损失。Knuth提供了两个有启发意义的例子，其中精度不足的四舍五入浮点运算导致加法的关联和分布性质的崩溃：

```
# Examples from Seminumerical Algorithms, Section 4.2.2.
>>> from decimal import Decimal, getcontext
>>> getcontext().prec = 8

>>> u, v, w = Decimal(11111113), Decimal(-11111111), Decimal('7.51111111')
>>> (u + v) + w
Decimal('9.5111111')
>>> u + (v + w)
Decimal('10')

>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
Decimal('0.01')
>>> u * (v+w)
Decimal('0.0060000')
```

该decimal模块可以通过充分扩展精度来恢复身份，以避免失去重要性：

```
>>> getcontext().prec = 20
>>> u, v, w = Decimal(11111113), Decimal(-11111111), Decimal('7.51111111')
>>> (u + v) + w
Decimal('9.51111111')
>>> u + (v + w)
Decimal('9.51111111')
>>>
>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
Decimal('0.0060000')
```

```
>>> u * (v+w)
Decimal('0.0060000')
```

9.4.7.2。特殊值

对于数字系统decimal模块提供了特殊的值，包括NaN，sNaN，-Infinity，Infinity，和两个零，+0和-0。

无穷可以直接用： Decimal('Infinity')。另外，当DivisionByZero信号未被捕获时，它们可以通过除以零而产生。同样，当Overflow信号未被捕获时，无限大可能因四舍五入而超出最大可表示数的极限。

无穷大被签名（仿射），可用于算术运算中，它们被视为非常大，不确定的数字。例如，将一个常数加到无穷大给出了另一个无限的结果。

有些操作是不确定的并返回NaN，或者如果 InvalidOperation信号被困住，引发异常。例如，0/0返回NaN意味着“不是数字”。这种多样性 NaN是安静的，一旦创建，将会流过其他计算总是导致另一个计算NaN。这种行为对于偶尔缺少输入的一系列计算很有用 - 它允许在将特定结果标记为无效时继续进行计算。

一个变体是sNaN每个操作后信号而不是保持安静。当无效结果需要中断特殊处理的计算时，这是一个有用的返回值。

NaN涉及到a的时候，Python的比较运算符的行为可能有点令人惊讶 。一个操作数是安静的或信号NaN总是返回的False（即使是这样 Decimal('NaN')==Decimal('NaN')）的平等测试，而不平等测试总是返回 True。尝试使用任何比较两个小数< ， <= ，>或>=运营商将提高InvalidOperation信号如果操作数是一个NaN，并返回False如果这个信号没有被限制。请注意，通用十进制算术规范并未指定直接比较的行为; 这些涉及a的比较规则NaN取自IEEE 854标准（参见5.7节中的表3）。为确保遵守严格的标准，请使用compare() 和compare-signal() 方法。

有符号的零可能源自下溢的计算结果。如果计算过程更加精确，它们会保留原来的符号。由于它们的大小为零，所以正和负零都被视为相等并且它们的符号是信息性的。

除了两个不同而又相等的有符号零点之外，零点的各种表示还有不同的精度，但其价值相同。这需要一些习惯。对于习惯于归一化浮点表示的眼睛来说，下面的计算返回等于零的值并不明显：

```
>>> 1 / Decimal('Infinity')
Decimal('0E-1000026')
```

9.4.8。使用线程

该getcontext()函数Context为每个线程访问不同的对象。具有独立的线程上下文意味着线程可以进行更改（例如getcontext().prec=10）而不会干扰其他线程。

同样，该setcontext()函数会自动将其目标分配给当前线程。

如果setcontext()之前未被调用getcontext()， getcontext()则会自动创建一个新的上下文以供当前线程使用。

新的上下文是从名为*DefaultContext*的原型上下文中复制的。要控制默认值，以便每个线程在整个应用程序中使用相同的值，请直接修改*DefaultContext*对象。这应该*在*任何线程启动*之前*完成 ，以便在线程调用之间不会出现争用条件getcontext()。例如：

```
# Set applicationwide defaults for all threads about to be launched
DefaultContext.prec = 12
DefaultContext.rounding = ROUND_DOWN
DefaultContext.traps = ExtendedContext.traps.copy()
DefaultContext.traps[InvalidOperation] = 1
setcontext(DefaultContext)

# Afterwards, the threads can be started
t1.start()
t2.start()
t3.start()
 . . .
```

9.4.9。食谱

以下是一些用作实用功能的食谱，并演示了如何与Decimal课程一起工作：

```
def moneyfmt(value, places=2, curr='', sep=',', dp='.',
             pos='', neg='-', trailneg=''):
    """Convert Decimal to a money formatted string.

    places:  required number of places after the decimal point
    curr:    optional currency symbol before the sign (may be blank)
    sep:     optional grouping separator (comma, period, space, or blank)
    dp:      decimal point indicator (comma or period)
             only specify as blank when places is zero
    pos:     optional sign for positive numbers: '+', space or blank
    neg:     optional sign for negative numbers: '-', '(', space or blank
    trailneg:optional trailing minus indicator:  '-', ')', space or blank

    >>> d = Decimal('-1234567.8901')
    >>> moneyfmt(d, curr='$')
    '-$1,234,567.89'
    >>> moneyfmt(d, places=0, sep='.', dp='', neg='', trailneg='-')
    '1.234.568-'
    >>> moneyfmt(d, curr='$', neg='(', trailneg=')')
    '($1,234,567.89)'
    >>> moneyfmt(Decimal(123456789), sep=' ')
    '123 456 789.00'
    >>> moneyfmt(Decimal('-0.02'), neg='<', trailneg='>')
    '<0.02>'

    """
    q = Decimal(10) ** -places      # 2 places --> '0.01'
    sign, digits, exp = value.quantize(q).as_tuple()
    result = []
    digits = list(map(str, digits))
```

```
    build, next = result.append, digits.pop
    if sign:
        build(trailneg)
    for i in range(places):
        build(next() if digits else '0')
    if places:
        build(dp)
    if not digits:
        build('0')
    i = 0
    while digits:
        build(next())
        i += 1
        if i == 3 and digits:
            i = 0
            build(sep)
    build(curr)
    build(neg if sign else pos)
    return ''.join(reversed(result))

def pi():
    """Compute Pi to the current precision.

    >>> print(pi())
    3.141592653589793238462643383

    """
    getcontext().prec += 2  # extra digits for intermediate steps
    three = Decimal(3)      # substitute "three=3.0" for regular floats
    lasts, t, s, n, na, d, da = 0, three, 3, 1, 0, 0, 24
    while s != lasts:
        lasts = s
        n, na = n+na, na+8
        d, da = d+da, da+32
        t = (t * n) / d
        s += t
    getcontext().prec -= 2
    return +s               # unary plus applies the new precision

def exp(x):
    """Return e raised to the power of x.  Result type matches input type.

    >>> print(exp(Decimal(1)))
    2.718281828459045235360287471
    >>> print(exp(Decimal(2)))
    7.389056098930650227230427461
    >>> print(exp(2.0))
    7.38905609893
    >>> print(exp(2+0j))
    (7.38905609893+0j)

    """
    getcontext().prec += 2
    i, lasts, s, fact, num = 0, 0, 1, 1, 1
    while s != lasts:
        lasts = s
        i += 1
        fact *= i
```

```
        num *= x
        s += num / fact
    getcontext().prec -= 2
    return +s

def cos(x):
    """Return the cosine of x as measured in radians.

    The Taylor series approximation works best for a small value of x.
    For larger values, first compute x = x % (2 * pi).

    >>> print(cos(Decimal('0.5')))
    0.87758256189037271611628158260
    >>> print(cos(0.5))
    0.87758256189
    >>> print(cos(0.5+0j))
    (0.87758256189+0j)

    """
    getcontext().prec += 2
    i, lasts, s, fact, num, sign = 0, 0, 1, 1, 1, 1
    while s != lasts:
        lasts = s
        i += 2
        fact *= i * (i-1)
        num *= x * x
        sign *= -1
        s += num / fact * sign
    getcontext().prec -= 2
    return +s

def sin(x):
    """Return the sine of x as measured in radians.

    The Taylor series approximation works best for a small value of x.
    For larger values, first compute x = x % (2 * pi).

    >>> print(sin(Decimal('0.5')))
    0.47942553860420300027328793520
    >>> print(sin(0.5))
    0.479425538604
    >>> print(sin(0.5+0j))
    (0.479425538604+0j)

    """
    getcontext().prec += 2
    i, lasts, s, fact, num, sign = 1, 0, x, 1, x, 1
    while s != lasts:
        lasts = s
        i += 2
        fact *= i * (i-1)
        num *= x * x
        sign *= -1
        s += num / fact * sign
    getcontext().prec -= 2
    return +s
```

9.4.10。十进制常见问题

问：输入很麻烦decimal.Decimal('1234.5')。使用交互式解释器时，是否有办法减少键入？

答：有些用户将构造函数缩写为一个字母：

```
>>> D = decimal.Decimal
>>> D('1.23') + D('3.45')
Decimal('4.68')
```

问：在具有两位小数的定点应用程序中，某些输入有许多地方，需要四舍五入。其他人不应该有过多的数字，需要验证。应该使用什么方法？

A.该quantize()方法舍入到固定的小数位数。如果Inexact设置了陷阱，它对验证也很有用：

```
>>> TWOPLACES = Decimal(10) ** -2       # same as Decimal('0.01')
```

```
>>> # Round to two places
>>> Decimal('3.214').quantize(TWOPLACES)
Decimal('3.21')
```

```
>>> # Validate that a number does not exceed two places
>>> Decimal('3.21').quantize(TWOPLACES, context=Context(traps=[Inexact]))
Decimal('3.21')
```

```
>>> Decimal('3.214').quantize(TWOPLACES, context=Context(traps=[Inexact]))
Traceback (most recent call last):
   ...
Inexact: None
```

问：一旦我有有效的两个位置输入，我如何在整个应用程序中保持不变？

答：一些操作，如加，减，乘整数将自动保留固定点。其他的操作，如分割和非整数乘法，将改变小数位数，需要后续quantize()步骤：

```
>>> a = Decimal('102.72')           # Initial fixed-point values
>>> b = Decimal('3.17')
>>> a + b                           # Addition preserves fixed-point
Decimal('105.89')
>>> a - b
Decimal('99.55')
>>> a * 42                          # So does integer multiplication
Decimal('4314.24')
>>> (a * b).quantize(TWOPLACES)     # Must quantize non-integer multiplication
Decimal('325.62')
>>> (b / a).quantize(TWOPLACES)     # And quantize division
Decimal('0.03')
```

在开发定点应用程序时，定义处理该quantize()步骤的函数很方便：

```
>>> def mul(x, y, fp=TWOPLACES):
...     return (x * y).quantize(fp)
>>> def div(x, y, fp=TWOPLACES):
...     return (x / y).quantize(fp)
```

```
>>> mul(a, b)                           # Automatically preserve fixed-point
Decimal('325.62')
>>> div(b, a)
Decimal('0.03')
```

问：表达相同价值的方法很多。这些数字200， 200.000，2E2，并且02E+4都在不同的精度相同的值。有没有办法将它们转换为单一可识别的规范值？

A.该normalize()方法将所有等价值映射到单个代表：

```
>>> values = map(Decimal, '200 200.000 2E2 .02E+4'.split())
>>> [v.normalize() for v in values]
[Decimal('2E+2'), Decimal('2E+2'), Decimal('2E+2'), Decimal('2E+2')]
```

问：一些小数值总是以指数表示法打印。有没有办法获得非指数表示？

答：对于某些值，指数符号是表示系数中重要位置数的唯一方式。例如，表达 5.0E+3为5000保持值不变，但不能显示原始的两位意义。

如果应用程序不关心跟踪重要性，则很容易去除指数和尾随零，失去重要性，但保持值不变：

```
>>> def remove_exponent(d):
...     return d.quantize(Decimal(1)) if d == d.to_integral() else d.normalize()
```

```
>>> remove_exponent(Decimal('5E+3'))
Decimal('5000')
```

问：是否有方法将常规浮点数转换为Decimal？

答：是的，任何二进制浮点数都可以精确地表示为十进制，虽然精确转换可能比直觉所建议的要精确得多：

```
>>> Decimal(math.pi)
Decimal('3.141592653589793115997963468544185161590576171875')
```

问：在复杂的计算过程中，我如何确保由于精度不足或舍入异常导致我没有得到虚假结果。

答：十进制模块可以很容易地测试结果。最佳做法是使用更高精度和各种舍入模式重新运行计算。广泛不同的结果表明精度不足，舍入模式问题，病态输入或数字不稳定算法。

问：我注意到上下文的精确度适用于操作的结果，但不适用于输入。混合不同精度值时有什么需要注意的吗？

A.是的。原则是所有的值都被认为是精确的，这些值的算术也是如此。只有结果四舍五入。投入的优点是“你输入的是你所得到的”。缺点是，如果忘记输入未被舍入，结果可能会显得很奇怪：

```
>>> getcontext().prec = 3
>>> Decimal('3.104') + Decimal('2.104')
Decimal('5.21')
>>> Decimal('3.104') + Decimal('0.000') + Decimal('2.104')
Decimal('5.20')
```

解决方法是提高精度或强制使用一元加运算对输入进行四舍五入：

```
>>> getcontext().prec = 3
>>> +Decimal('1.23456789')      # unary plus triggers rounding
Decimal('1.23')
```

或者，输入可以在创建时使用以下Context.create_decimal()方法进行四舍五入 ：

```
>>> Context(prec=5, rounding=ROUND_DOWN).create_decimal('1.2345678')
Decimal('1.2345')
```

9.5。fractions- 理性数字

源代码： Lib / fractions.py

该fractions模块提供对有理数算术的支持。

分数实例可以由一对整数，另一个有理数或字符串构造而成。

class fractions. Fraction (*分子= 0* , *分母= 1*)
class fractions. Fraction (*other_fraction*)
*类*fractions. Fraction (*float*)
*类*fractions. Fraction (*十进制*)
*类*fractions. Fraction (*字符串*)

第一个版本要求*分子*和*分母*是实例，numbers.Rational并返回一个Fraction带有值的新实例numerator/denominator。如果*分母*是0，它提出了一个ZeroDivisionError。第二个版本要求 *other_fraction*是一个实例，numbers.Rational并返回一个Fraction具有相同值的 实例。接下来的两个版本接受一个float或一个decimal.Decimal实例，并返回一个Fraction具有完全相同值的 实例。需要注意的是，由于与二进制浮点常见问题（见浮点运算：问题和限制），该参数Fraction(1.1)是不完全等于11/10，所以 Fraction(1.1)不*不像*人们所期望的那样回报。（但请参阅下面方法的文档。）构造函数的最后一个版本需要一个字符串或unicode实例。这种情况的通常形式是：Fraction(11, 10) limit_denominator()

```
[sign] numerator ['/' denominator]
```

其中可选项sign可以是'+'或' - '， numerator并且denominator（如果存在）是十进制数字的字符串。此外，float构造函数也接受任何代表有限值并被构造函数接受的字符串Fraction。在任何一种形式中，输入字符串也可能具有前导和/或尾随空白。这里有些例子：

```
>>> from fractions import Fraction
>>> Fraction(16, -10)
Fraction(-8, 5)
>>> Fraction(123)
Fraction(123, 1)
>>> Fraction()
Fraction(0, 1)
>>> Fraction('3/7')
Fraction(3, 7)
>>> Fraction(' -3/7 ')
Fraction(-3, 7)
>>> Fraction('1.414213 \t\n')
Fraction(1414213, 1000000)
>>> Fraction('-.125')
Fraction(-1, 8)
>>> Fraction('7e-6')
Fraction(7, 1000000)
>>> Fraction(2.25)
Fraction(9, 4)
>>> Fraction(1.1)
```

```
Fraction(2476979795053773, 2251799813685248)
>>> from decimal import Decimal
>>> Fraction(Decimal('1.1'))
Fraction(11, 10)
```

本Fraction类从抽象基类继承 numbers.Rational ，并实现所有从该类的方法和操作。Fraction实例是可散列的，应该被视为不可变的。另外， Fraction还有以下属性和方法：

*改变在3.2版本：*在Fraction构造函数现在接受float和 decimal.Decimal实例。

numerator
: 最低分数的分子分子。

denominator
: 最低期的分数分母。

from_float (*flt*)
: 这个类方法构造了一个Fraction代表*flt*的确切值，它必须是a float。当心这Fraction.from_float(0.3)与价值不同。Fraction(3, 10)

> **注意:** 从Python 3.2开始，你也可以Fraction直接从a 构造一个 实例float。

from_decimal (*dec*)
: 这个类方法构造了一个Fraction代表*dec*的确切值的值，它必须是一个decimal.Decimal实例。

> **注意:** 从Python 3.2开始，您也可以Fraction直接从decimal.Decimal 实例构造 一个实例。

limit_denominator (*max_denominator = 1000000*)
: 查找并返回最接近Fraction于self具有至多max_denominator的分母。此方法对于找到给定浮点数的有理逼近很有用：

```
>>> from fractions import Fraction
>>> Fraction('3.1415926535897932').limit_denominator(1000)
Fraction(355, 113)
```

或者恢复一个表示为浮点的有理数：

```
>>> from math import pi, cos
>>> Fraction(cos(pi/3))
Fraction(4503599627370497, 9007199254740992)
>>> Fraction(cos(pi/3)).limit_denominator()
Fraction(1, 2)
>>> Fraction(1.1).limit_denominator()
Fraction(11, 10)
```

__floor__ ( )
: 返回最大值。这个方法也可以通过函数来访问：int <= selfmath.floor()

```
>>> from math import floor
>>> floor(Fraction(355, 113))
3
```

__ceil__ ( )

最少返回。这个方法也可以通过函数访问。 int >= selfmath.ceil()

__round__ ( )

__round__ (*ndigits*)

第一个版本返回最接近int的self，舍入一半到偶数。第二个版本轮到self最接近的倍数（逻辑上，如果 是负值），再次舍入一半。这个方法也可以通过函数访问。 Fraction(1, 10**ndigits) ndigits round()

fractions. gcd (*a* , *b*)

返回整数*a*和*b*的最大公约数。如果 *a*或*b*不为零，那么绝对值是将*a*和*b*分开的最大整数。如果*b*不为零，则与*b*具有相同的符号; 否则它需要的符号一个。 返回。 gcd(a, b) gcd(a, b) gcd(0, 0) 0

自3.5版以来已弃用： math. gcd() 改为使用。

也可以看看:

模 numbers

组成数字塔的抽象基类。

9.6。 random- 生成伪随机数

源代码： Lib / random.py

该模块为各种分配实施伪随机数发生器。

对于整数，从一个范围有统一的选择。对于序列，有一个随机元素的统一选择，一个就地生成一个列表的随机置换的函数，以及一个用于随机抽样而不需要替换的函数。

在实际情况中，有函数可以计算均匀，正态（高斯），对数正态，负指数，伽玛和贝塔分布。为了产生角度分布，冯·米塞斯分布是可用的。

几乎所有的模块函数都依赖于基本函数random()，它在半开放范围[0.0,1.0）内均匀地生成随机浮点数。Python使用Mersenne Twister作为核心生成器。它生成53位精度浮点数，周期为2 ** 19937-1。C中的底层实现既快速又线程安全。Mersenne Twister是现存最广泛测试的随机数生成器之一。然而，它是完全确定性的，并不适用于所有目的，并且完全不适用于加密目的。

这个模块提供的函数实际上是random.Random类的隐藏实例的绑定方法。您可以实例化您自己的实例Random以获取不共享状态的生成器。

类Random也可以，如果你想用你自己设计的不同的基本发电机子类：在这种情况下，覆盖random()， seed()，getstate()，和setstate()方法。可选地，新的发生器可以提供一种getrandbits()方法 - 这允许randrange()在任意大的范围内产生选择。

该random模块还提供了SystemRandom使用系统函数os.urandom()从操作系统提供的源生成随机数的类。

警告： 该模块的伪随机生成器不应用于安全目的。为了安全或加密使用，请参阅 secrets模块。

也可以看看： M.Matsumoto和T.Nishimura，“Mersenne Twister：A 623-dimensionalally equidistributed uniform pseudorandom number generator”，ACM Transactions on Modeling and Computer Simulation Vol。8，第1号，1月，pp.3-30 1998年。

补充 - 与进位乘法配方，用于兼容的替代随机数发生器，具有长周期和相对简单的更新操作。

9.6.1。簿记功能

random.seed（*a* = *None*，*版本*= *2* ）

初始化随机数发生器。

如果*一个*被省略或None，当前系统时间被使用。如果随机源由操作系统提供，则使用它们而不是系统时间（os.urandom()有关可用性的详细信息，请参阅该函数）。

如果*a*是一个int，则直接使用它。

对于版本2（默认），a `str` ，`bytes`或`bytearray` 对象被转换为一个，`int`并使用其所有位。

与版本1（提供用于从旧版本的Python再现随机序列）时，算法`str`和`bytes`产生更窄的范围种子。

*版本3.2中已更改：已*移至使用字符串种子中所有位的版本2方案。

random.**getstate**( )

返回捕获发生器当前内部状态的对象。该对象可以传递`setstate()`到恢复状态。

random.**setstate**(*状态*)

*状态*应该是从以前的调用中获得的`getstate()` ，并将 `setstate()`发生器的内部状态恢复到当时`getstate()`所调用的状态。

random.**getrandbits**(*k*)

返回一个具有*k个*随机位的Python整数。该方法由MersenneTwister生成器提供，其他一些生成器也可以将其作为API的可选部分提供。可用时，`getrandbits()`可以 `randrange()`处理任意大的范围。

9.6.2。整数函数

random.**randrange**(*停止*)

random.**randrange**(*开始* , *停止*[, *步骤*])

从中随机选择一个元素。这相当于，但实际上并不构建范围对象。`range(start, stop, step) choice(range(start, stop, step))`

位置参数模式匹配`range()`。不应该使用关键字参数，因为函数可能以意想不到的方式使用它们。

在版本3.2中进行了更改：`randrange()`在生成同等分布的值时更加复杂。以前它使用了`int(random()*n)`可能产生稍微不均匀分布的风格。

random.**randint**(*a* , *b*)

返回一个随机整数*N*使得。别名 。`a <= N <= b randrange(a, b+1)`

9.6.3。序列函数

random.**choice**(*seq*)

从非空序列*seq中*返回一个随机元素。如果*seq*为空，则引发`IndexError`。

random.**choices**(*人口* , *权重=无* , * , *cum_weights =无* , *k = 1*)

从替换*人群中*选择一个*k*大小的元素列表。如果*人口*是空的，提高。 `IndexError`

如果指定了*权重*序列，则根据相对权重进行选择。或者，如果给出*cum_weights*序列，则根据累积权重（可能使用计算`itertools.accumulate()` ）进行选择。例如，相对权重 等于

累计权重 。在进行选择之前，相对权重在内部转换为累计权重，因此提供累计权重可以节省工作量。`[10, 5, 30, 5]`[10, 15, 45, 50]

如果没有指定*权重*和*cum_weights*，则选择的概率相等。如果提供了权重序列，它必须与*群体*序列的长度相同。这是`TypeError` 指定*重量*和*cum_weights*。

的*权重*或*cum_weights*可以使用任何数值类型与该互操作`float`由返回的值`random()`（其包括整数，浮点数，和分数，但不包括小数）。

3.6版本中的新功能。

random.**shuffle**(*x*[, *random*])

将序列*x*随机混合。

可选参数*random*是一个0参数函数，返回[0.0，1.0）中的随机浮点数; 默认情况下，这是该功能`random()`。

要洗牌不可变的序列并返回一个新的洗牌列表，请改为使用 。`sample(x, k=len(x))`

请注意，即使对于较小的情况`len(x)`，*x*的排列总数 也可能快于大多数随机数发生器的周期。这意味着永远不会产生长序列的大部分排列。例如，长度为2080的序列是可以在Mersenne Twister随机数生成器的周期内拟合的最大序列。

random.**sample**(*人口*, *k*)

返回从总体序列或集合中选择的唯一元素的*k*长度列表。用于无需更换的随机抽样。

返回包含来自群体的元素的新列表，同时保持原始人口不变。结果列表按选择顺序排列，以便所有子片也将是有效的随机样本。这允许抽奖获奖者（样本）被划分为大奖和第二名获奖者（子公司）。

人口成员不必是可排除的或独特的。如果总体包含重复，则每个事件都是样本中可能的选择。

要从一系列整数中选择一个样本，请使用一个`range()`对象作为参数。这对于从大量人群中抽样来说尤其快速且节省空间： 。`sample(range(10000000), k=60)`

如果样本量大于总体规模，`ValueError` 则会提高a。

9.6.4。实值分布

以下函数生成特定的实值分布。函数参数是根据分布方程中的相应变量命名的，正如常用的数学实践中所用的那样; 大多数这些方程可以在任何统计文本中找到。

random.**random**()

返回范围[0.0，1.0）中的下一个随机浮点数。

random.**uniform**(*a*, *b*)

返回一个随机浮点数*N*，以便for 和for 。`a <= N <= ba <= bb <= N <= ab < a`

b取决于等式中的浮点舍入，终点值可能包含或不包含在范围内。`a + (b-a) * random()`

random.**triangular**(*低*, *高*, *模式*)

返回一个随机的浮点数*N*，并使用这些边界之间的指定*模式*。该*低*和*高*界默认的0和1。所述*模式*参数默认为边界之间的中点，给人一种对称分布。`low <= N <= high`

random.**betavariate**(*alpha*, *beta*)

Beta分布。参数条件是和 。返回值介于0和1之间。`alpha > 0beta > 0`

random.**expovariate**(*lambd*)

指数分布。 *lambd*是1.0除以所需的平均值。它应该是非零的。（该参数将被称为“拉姆达”，但是这是在Python保留字。）返回值的范围从0到正无穷大如果*lambd*为正，且从负无穷大到0，如果*lambd*为负。

random.**gammavariate**(*alpha*, *beta*)

伽马分布。（*不是*伽玛函数！）参数条件是和。`alpha > 0beta > 0`

概率分布函数是：

```
            x ** (alpha - 1) * math.exp(-x / beta)
pdf(x) =  --------------------------------------
            math.gamma(alpha) * beta ** alpha
```

random.**gauss**(*mu*, *sigma*)

高斯分布。 *mu*是平均值，而*σ*是标准偏差。这比`normalvariate()`下面定义的函数稍快。

random.**lognormvariate**(*mu*, *sigma*)

记录正态分布。如果你采用这种分布的自然对数，你将得到一个平均值为*mu*和标准偏差*σ的*正态分布。 *宙*可以有任何价值，*西格玛*必须大于零。

random.**normalvariate**(*mu*, *sigma*)

正态分布。 *mu*是平均值，而*σ*是标准偏差。

random.**vonmisesvariate**(*mu*, *kappa*)

*mu*是平均角度，以弧度表示，介于0和2 * *pi之间*，*kappa* 是浓度参数，必须大于或等于零。如果 *kappa*等于零，则该分布在0到2 * *pi*的范围内降低到均匀的随机角度。

random.**paretovariate**(*alpha*)

帕累托分布。 *alpha*是形状参数。

random.**weibullvariate**(*alpha*, *beta*)

威布尔分布。 *alpha*是比例参数，*beta*是形状参数。

9.6.5。替代生成器

class random.**SystemRandom**([*seed*])

使用该os.urandom()函数从操作系统提供的源生成随机数的类。不适用于所有系统。不依赖于软件状态，并且序列不可重现。因此，该seed()方法没有效果并被忽略。该getstate()和setstate()方法提高NotImplementedError，如果调用。

9.6.6。再现性注意事项

有时能够重现由伪随机数生成器给出的序列是有用的。通过重新使用种子值，只要多个线程没有运行，相同的序列应该从运行到运行再现。

大多数随机模块的算法和种子功能在Python版本中可能会发生变化，但有两个方面保证不会改变：

- 如果添加新的播种方法，则会提供向后兼容的播种机。
- random()当兼容播种机被赋予相同种子时，发生器的方法将继续产生相同的序列。

9.6.7。示例和食谱

基本示例：

```
>>> random()                            # Random float:  0.0 <= x < 1.0
0.37444887175646646

>>> uniform(2.5, 10.0)                  # Random float:  2.5 <= x < 10.0
3.1800146073117523

>>> expovariate(1 / 5)                  # Interval between arrivals averaging 5 seco
5.148957571865031

>>> randrange(10)                       # Integer from 0 to 9 inclusive
7

>>> randrange(0, 101, 2)                # Even integer from 0 to 100 inclusive
26

>>> choice(['win', 'lose', 'draw'])     # Single random element from a sequence
'draw'

>>> deck = 'ace two three four'.split()
>>> shuffle(deck)                       # Shuffle a list
>>> deck
['four', 'two', 'ace', 'three']

>>> sample([10, 20, 30, 40, 50], k=4)   # Four samples without replacement
[40, 10, 50, 30]
```

模拟：

```
>>> # Six roulette wheel spins (weighted sampling with replacement)
>>> choices(['red', 'black', 'green'], [18, 18, 2], k=6)
['red', 'green', 'black', 'black', 'red', 'black']
```

```
>>> # Deal 20 cards without replacement from a deck of 52 playing cards
>>> # and determine the proportion of cards with a ten-value
>>> # (a ten, jack, queen, or king).
>>> deck = collections.Counter(tens=16, low_cards=36)
>>> seen = sample(list(deck.elements()), k=20)
>>> seen.count('tens') / 20
0.15

>>> # Estimate the probability of getting 5 or more heads from 7 spins
>>> # of a biased coin that settles on heads 60% of the time.
>>> trial = lambda: choices('HT', cum_weights=(0.60, 1.00), k=7).count('H') >= 5
>>> sum(trial() for i in range(10000)) / 10000
0.4169

>>> # Probability of the median of 5 samples being in middle two quartiles
>>> trial = lambda : 2500 <= sorted(choices(range(10000), k=5))[2]  < 7500
>>> sum(trial() for i in range(10000)) / 10000
0.7958
```

的实施例的统计自举使用具有置换重采样来估计置信区间大小的5的样品的平均值：

```
# http://statistics.about.com/od/Applications/a/Example-Of-Bootstrapping.htm
from statistics import mean
from random import choices

data = 1, 2, 4, 4, 10
means = sorted(mean(choices(data, k=5)) for i in range(20))
print(f'The sample mean of {mean(data):.1f} has a 90% confidence '
      f'interval from {means[1]:.1f} to {means[-2]:.1f}')
```

用于确定药物与安慰剂之间观察到的差异的统计学显着性或p值的重采样置换测试的实例：

```
# Example from "Statistics is Easy" by Dennis Shasha and Manda Wilson
from statistics import mean
from random import shuffle

drug = [54, 73, 53, 70, 73, 68, 52, 65, 65]
placebo = [54, 51, 58, 44, 55, 52, 42, 47, 58, 46]
observed_diff = mean(drug) - mean(placebo)

n = 10000
count = 0
combined = drug + placebo
for i in range(n):
    shuffle(combined)
    new_diff = mean(combined[:len(drug)]) - mean(combined[len(drug):])
    count += (new_diff >= observed_diff)

print(f'{n} label reshufflings produced only {count} instances with a difference')
print(f'at least as extreme as the observed difference of {observed_diff:.1f}.')
print(f'The one-sided p-value of {count / n:.4f} leads us to reject the null')
print(f'hypothesis that there is no difference between the drug and the placebo.')
```

在单个服务器队列中模拟到达时间和服务交付：

```
from random import expovariate, gauss
from statistics import mean, median, stdev

average_arrival_interval = 5.6
average_service_time = 5.0
stdev_service_time = 0.5

num_waiting = 0
arrivals = []
starts = []
arrival = service_end = 0.0
for i in range(20000):
    if arrival <= service_end:
        num_waiting += 1
        arrival += expovariate(1.0 / average_arrival_interval)
        arrivals.append(arrival)
    else:
        num_waiting -= 1
        service_start = service_end if num_waiting else arrival
        service_time = gauss(average_service_time, stdev_service_time)
        service_end = service_start + service_time
        starts.append(service_start)

waits = [start - arrival for arrival, start in zip(arrivals, starts)]
print(f'Mean wait: {mean(waits):.1f}.  Stdev wait: {stdev(waits):.1f}.')
print(f'Median wait: {median(waits):.1f}.  Max wait: {max(waits):.1f}.')
```

也可以看看: 统计黑客Jake Vanderplas 的视频教程 只使用几个基本概念进行统计分析，包括模拟，采样，混排和交叉验证。

经济学模拟Peter Norvig对 市场的模拟 ，显示了该模块提供的许多工具和分布（高斯，统一，样本，变差，选择，三角和兰格）的有效使用。

概率的具体介绍（使用Python）Peter Norvig编写的 一篇教程，涵盖了概率论的基础知识，如何编写模拟以及如何使用Python执行数据分析。

9.7。 statistics- 数学统计函数

3.4版新增功能

源代码： Lib / statistics.py

该模块提供计算数字（ Real估值）数据的数学统计的功能。

> **注意:** 除非另有明确说明，这些功能的支持int ， float ，decimal.Decimal和fractions.Fraction。其他类型的行为（无论是否在数字塔中）当前不受支持。混合类型也是未定义的和依赖于实现的。如果您的输入数据由混合类型组成，那么您可以使用它map()来确保一致的结果，例如 。 map(float, input_data)

9.7.1。平均值和中心位置的措施

这些函数计算总体或样本的平均值或典型值。

mean()	数据的算术平均值（“平均值”）。
harmonic_mean()	数据的谐波平均值。
median()	数据的中间值（中间值）。
median_low()	数据的中位数较低。
median_high()	数据的中位数高。
median_grouped()	分组数据的中位数或第50百分位。
mode()	离散数据的模式（最常见的值）。

9.7.2。传播的措施

这些函数用于计算人口或样本偏离典型值或平均值的程度。

pstdev()	数据的人口标准差。
pvariance()	数据的人口差异。
stdev()	数据标准差的样本。
variance()	数据的样本方差。

9.7.3。功能细节

注意：这些功能不需要给予他们的数据进行排序。但是，为了阅读方便，大多数示例显示了排序顺序。

statistics.mean（*数据*）

返回可以是序列或迭代器的*数据的*算术平均值。

算术平均值是数据的总和除以数据点的数量。它通常被称为“平均数”，尽管它只是许多不同数学平均数中的一个。这是衡量数据中心位置的一个指标。

如果*数据*为空，StatisticsError则会提出。

一些使用示例：

```
>>> mean([1, 2, 3, 4, 4])
2.8
>>> mean([-1.0, 2.5, 3.25, 5.75])
2.625

>>> from fractions import Fraction as F
>>> mean([F(3, 7), F(1, 21), F(5, 3), F(1, 3)])
Fraction(13, 21)

>>> from decimal import Decimal as D
>>> mean([D("0.5"), D("0.75"), D("0.625"), D("0.375")])
Decimal('0.5625')
```

注意： 均值受到异常值的强烈影响，对于中央位置不是一个稳健的估计量：均值不一定是数据点的典型示例。为了更强大，尽管效率较低，中央位置的措施，请参阅median()和mode()。（在这种情况下，“高效”是指统计效率，而不是计算效率。）
样本均值给出了真实总体均值的无偏估计，这意味着对所有可能的样本取平均值，mean(sample)收敛于整个人口的真实平均值。如果*数据*代表整个人口而不是样本，那么mean(data)相当于计算真实总体平均值μ。

statistics.**harmonic_mean**(*数据*)

返回*数据*的调和平均值，实数值序列或迭代器。

调和平均值（有时称为分包平均值）是mean()数据倒数算术的倒数。例如，三个值*a*，*b*和*c*的调和平均值 将相当于。3/(1/a + 1/b + 1/c)

调和平均值是一种平均值，是衡量数据中心位置的一种度量。对速率或比率（例如速度）进行平均时，通常是合适的。例如：

假设投资者购买三家公司中每家公司的股票价值相等，市盈率（价格/收益）比率分别为2.5,3和10.什么是投资者投资组合的平均市盈率？

```
>>> harmonic_mean([2.5, 3, 10])  # For an equal investment portfolio.
3.6
```

使用算术平均值会得到大约5.167的平均值，这太高了。

StatisticsError如果*数据*为空，或者任何元素小于零，则会引发。

3.6版本中的新功能。

statistics.**median**(*数据*)

使用常见的“中间二次均值”方法返回数值数据的中位数（中间值）。如果*数据*是空的，StatisticsError则引发。 *数据*可以是一个序列或迭代器。

中位数是衡量中央位置的有力措施，受数据中存在异常值的影响较小。当数据点数为奇数时，返回中间数据点：

```
>>> median([1, 3, 5])
3
```

当数据点数为偶数时，通过取两个中间值的平均值来插入中值：

```
>>> median([1, 3, 5, 7])
4.0
```

这适用于您的数据是离散数据的时候，并且您不介意中位数可能不是实际的数据点。

也可以看看： median_low() ，median_high() ，median_grouped()

statistics.**median_low**(*数据*)

返回数字数据的中位数。如果*数据*是空的， StatisticsError则引发。 *数据*可以是一个序列或迭代器。

低中位数始终是数据集的成员。当数据点数为奇数时，返回中间值。当它是偶数时，返回两个中间值中较小的一个。

```
>>> median_low([1, 3, 5])
3
>>> median_low([1, 3, 5, 7])
3
```

当你的数据是离散数据时，使用低位中位数，你更喜欢中位数是实际数据点而不是插值。

statistics.**median_high**(*数据*)

返回高位数据。如果*数据*是空的，StatisticsError 则引发。 *数据*可以是一个序列或迭代器。

高位数始终是数据集的成员。当数据点数为奇数时，返回中间值。当它是偶数时，返回两个中间值中的较大值。

```
>>> median_high([1, 3, 5])
3
>>> median_high([1, 3, 5, 7])
5
```

当您的数据是离散数据时，使用高位中位数，并且您希望中位数是实际数据点而不是插值。

statistics.**median_grouped**(*data*，*interval = 1*)

使用内插返回按第50百分位数计算的分组连续数据的中位数。如果*数据*是空的，StatisticsError 则引发。 *数据*可以是一个序列或迭代器。

```
>>> median_grouped([52, 52, 53, 54])
52.5
```

在下面的例子中，数据是四舍五入的，所以每个值代表数据类的中点，例如1是类的中点0.5-1.5，2是1.5-2.5的中点，3是2.5-3.5的中点等等。根据给出的数据，中间值落在类别3.5-4.5中的某个位置，并且使用内插来估计它：

```
>>> median_grouped([1, 2, 2, 3, 4, 4, 4, 4, 4, 5])
3.7
```

可选的参数*间隔*代表类间隔，默认为1.更改类间隔自然会改变插值：

```
>>> median_grouped([1, 3, 3, 5, 7], interval=1)
3.25
>>> median_grouped([1, 3, 3, 5, 7], interval=2)
3.5
```

该功能不检查数据点是否至少 *间隔*分开。

CPython实现细节：在某些情况下，`median_grouped()`可能会将数据点强制为浮点数。这种行为在未来可能会改变。

也可以看看:

- “行为科学统计”，Frederick J Gravetter和Larry B Wallnau（第8版）。
- 计算中位数。
- Gnome Gnumeric电子表格中的SSMEDIAN函数，包括此讨论。

statistics.**mode**(*数据*)

从离散*数据*或标称*数据中*返回最常见的数据点。模式（当它存在时）是最典型的值，并且是中心位置的可靠度量。

如果*数据*是空的，或者如果没有一个最常见的值，`StatisticsError`则会引发。

`mode`假定离散数据，并返回单个值。这是学校通常教授的标准治疗模式：

```
>>> mode([1, 1, 2, 3, 3, 3, 3, 4])
3
```

该模式是独一无二的，因为它是唯一也适用于名义（非数字）数据的统计量：

```
>>> mode(["red", "blue", "blue", "red", "green", "red", "red"])
'red'
```

statistics.**pstdev**(*数据*, *mu=无*)

返回总体标准偏差（总体方差的平方根）。参见`pvariance()`参数和其他细节。

```
>>> pstdev([1.5, 2.5, 2.5, 2.75, 3.25, 4.75])
0.986893273527251
```

statistics.**pvariance**(*数据*, *mu=无*)

返回*数据*的总体方差，一个非空实数值的迭代。方差或平均值的第二时刻是衡量数据可变性（扩散或扩散）的指标。差异很大表示数据分散; 一个小的变化表明它紧密聚集在均值附

近。

如果给出了可选的第二个参数*mu*，它应该是*数据*的平均值。如果缺失或None（默认），平均值将自动计算。

使用此函数来计算整个人口的方差。为了估计样本的方差，variance()函数通常是更好的选择。

StatisticsError如果*数据*为空则引发。

例子：

```
>>> data = [0.0, 0.25, 0.25, 1.25, 1.5, 1.75, 2.75, 3.25]
>>> pvariance(data)
1.25
```

如果您已经计算了数据的平均值，则可以将其作为可选的第二个参数*mu*传递以避免重新计算：

```
>>> mu = mean(data)
>>> pvariance(data, mu)
1.25
```

此功能不会尝试验证您是否已将实际平均值作为*由来通过*。使用*mu的*任意值可能会导致无效或不可能的结果。

小数和分数支持：

```
>>> from decimal import Decimal as D
>>> pvariance([D("27.5"), D("30.25"), D("30.25"), D("34.5"), D("41.75")])
Decimal('24.815')

>>> from fractions import Fraction as F
>>> pvariance([F(1, 4), F(5, 4), F(1, 2)])
Fraction(13, 72)
```

注意： 当与全体人口打电话时，这给人口差异σ²。当调用一个样本时，这是有偏差的样本方差s²，也称为具有N个自由度的方差。
如果你知道真实的总体平均值μ，你可以使用这个函数来计算样本的方差，给出已知的总体平均值作为第二个参数。如果数据点具有代表性（例如独立且分布相同），则结果将是对人口变化的无偏估计。

statistics.**stdev**(*data*, *xbar=None*)

返回样本标准偏差（样本方差的平方根）。参见variance()参数和其他细节。

```
>>> stdev([1.5, 2.5, 2.5, 2.75, 3.25, 4.75])
1.0810874155219827
```

statistics.**variance**(*data*, *xbar=None*)

返回*数据*的样本方差，这是一个至少有两个实数值的迭代。方差或平均值的第二时刻是衡量数据可变性（扩散或扩散）的指标。差异很大表示数据分散; 一个小的变化表明它紧密聚集在均值附近。

如果给出可选的第二个参数*xbar*，它应该是*数据*的平均值 。如果缺失或None（默认），平均值将自动计算。

当您的数据是来自群体的样本时使用此功能。要计算整个人口的方差，请参阅pvariance()。

StatisticsError如果*数据*少于两个值则引发。

例子：

```
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> variance(data)
1.3720238095238095
```

如果您已经计算了数据的平均值，则可以将其作为可选的第二个参数*xbar*传递以避免重新计算：

```
>>> m = mean(data)
>>> variance(data, m)
1.3720238095238095
```

此函数不会试图验证您是否已将实际平均值作为*xbar*传递。对*xbar*使用任意值可能会导致无效或不可能的结果。

支持十进制和分数值：

```
>>> from decimal import Decimal as D
>>> variance([D("27.5"), D("30.25"), D("30.25"), D("34.5"), D("41.75")])
Decimal('31.01875')

>>> from fractions import Fraction as F
>>> variance([F(1, 6), F(1, 2), F(5, 3)])
Fraction(67, 108)
```

注意： 这是采用贝塞尔校正的样本方差s²，也称为具有N-1自由度的方差。如果数据点具有代表性（例如独立且分布相同），则结果应该是真实总体变化的无偏估计。
如果你知道实际的总体平均值μ，你应该将它pvariance()作为*μ*参数传递给 函数，以获得样本的方差。

9.7.4。例外

定义一个例外：

*异常*statistics.StatisticsError
与ValueError统计相关的异常的子类。

10.函数编程模块

本章介绍的模块提供了支持函数式编程风格的函数和类，以及可调用函数的一般操作。

本章介绍以下模块：

10.1。itertools- 为高效循环创建迭代器的函数

该模块实现了许多受APL，Haskell和SML构造启发的迭代器构建块。每一个都以适合Python的形式进行了重新编排。

该模块标准化了一套核心快速，高效的内存工具，这些工具本身或其组合都很有用。它们一起构成了一个“迭代器代数”，可以在纯Python中简洁高效地构建专用工具。

例如，SML提供了一个制表工具：`tabulate(f)`它产生一个序列。通过组合和形成，Python可以达到同样的效果。`f(0), f(1), ...` `map()` `count()` `map(f, count())`

这些工具及其内置对应模块也可以很好地处理`operator`模块中的高速功能。例如，可以将乘法运算符映射到两个向量上以形成高效的点积： 。`sum(map(operator.mul, vector1, vector2))`

无限迭代器：

迭代器	参数	结果	例
`count()`	骤]始，[步	骤始.., 开始+步骤，开始+ 2 *步	`count(10) --> 10 11 12 13 14 ...`
`cycle()`	p	p0，p1，... plast，p0，p1，...	`cycle('ABCD') --> A B C D A B C D ...`
`repeat()`	elem [，n]	最多n次元素，元素，.....无休止或	`repeat(10, 3) --> 10 10 10`

迭代器终止于最短的输入序列：

迭代器	参数	结果	例
`accumulate()`	p [，func]	p0 + p1，p0 + p1 + p2	`accumulate([1,2,3,4,5]) --> 1 3 6 10 15`
`chain()`	p，q，...	p0，p1，... plast，	`chain('ABC', 'DEF') --> A B C D E F`
`chain.from_iterable()`	迭代	p0，p1，... plast，	`chain.from_iterable(['ABC', 'DEF']) --> A B C D E F`
`compress()`	数据，选	如果selectors [1]	`compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F`
`dropwhile()`	pred，seq	seq [n]，从pred第一次失败时	`dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1`
`filterfalse()`	pred，seq	pred(elem)为false的seq元素，	`filterfalse(lambda x: x%2, range(10)) --> 0 2 4 6 8`
`groupby()`	迭代[，键]	按键值函数v）分组的	
`islice()`	seq [，start] [，stop] [，step]	来自seq的元素，从start开始（如果	`islice('ABCDEFG', 2, None) --> C D E F G`
`starmap()`	func，seq	func(*seq [0])，func(*seq [1])，	`starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000`
`takewhile()`	pred，seq	seq [0]，seq [1]，直到pred失败，	`takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4`
`tee()`	它，它	一个迭代器分成n个，将一	

迭代器	参数	结果	例
zip_longest()	p，q，...	(p[0]，q[0])，...	zip_longest('ABCD'，'xy'，fillvalue='-') --> Ax By C- D-

组合迭代器：

迭代器	参数	结果
product()	p[，q，...][，repeat=1]	笛卡尔积，相当于一个嵌套的for循
permutations()	p[，r]	长度r的元组，所有可能的排序，没有重复的元素
combinations()	p，r	长度r的元组，按排序顺序，没有重复的元素
combinations_with_replacement()	p，r	长度r的元组，按排序顺序，允许有重复元素的排序次序的长度元组
product('ABCD'，repeat=2)		AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD
permutations('ABCD'，2)		AB AC AD BA BC BD CA CB CD DA DB DC
combinations('ABCD'，2)		AB AC AD BC BD CD
combinations_with_replacement('ABCD'，2)		AA AB AC AD BB BC BD CC CD DD

10.1.1。Itertool函数

以下模块函数都构造并返回迭代器。一些提供无限长度的流，所以它们只能由截断流的函数或循环访问。

itertools.accumulate(*iterable*[，*func*])

创建一个迭代器，它返回累计和或其他二进制函数的累计结果（通过可选的*func*参数指定）。如果提供了*func*，它应该是两个参数的函数。输入*迭代的*元素可以是任何可以被接受为*func*参数的类型。（例如，使用默认添加操作，元素可以是任何可添加类型，包括Decimal或）Fraction。如果输入迭代器为空，则输出迭代器也将为空。

大致相当于：

```
def accumulate(iterable, func=operator.add):
    'Return running totals'
    # accumulate([1,2,3,4,5]) --> 1 3 6 10 15
    # accumulate([1,2,3,4,5], operator.mul) --> 1 2 6 24 120
    it = iter(iterable)
    try:
        total = next(it)
    except StopIteration:
        return
    yield total
    for element in it:
        total = func(total, element)
        yield total
```

*func*参数有很多用途。它可以设置 min() 为运行最小值，max() 运行最大值或operator.mul()正在运行的产品。摊销表可以通过累积利息和申请付款来建立。一阶 递归

关系 可以通过在迭代中提供初始值并仅使用*func*参数中的累计总数来建模：

```
>>> data = [3, 4, 6, 2, 1, 9, 0, 7, 5, 8]
>>> list(accumulate(data, operator.mul))     # running product
[3, 12, 72, 144, 144, 1296, 0, 0, 0, 0]
>>> list(accumulate(data, max))              # running maximum
[3, 4, 6, 6, 6, 9, 9, 9, 9, 9]

# Amortize a 5% loan of 1000 with 4 annual payments of 90
>>> cashflows = [1000, -90, -90, -90, -90]
>>> list(accumulate(cashflows, lambda bal, pmt: bal*1.05 + pmt))
[1000, 960.0, 918.0, 873.9000000000001, 827.5950000000001]

# Chaotic recurrence relation https://en.wikipedia.org/wiki/Logistic_map
>>> logistic_map = lambda x, _:  r * x * (1 - x)
>>> r = 3.8
>>> x0 = 0.4
>>> inputs = repeat(x0, 36)     # only the initial value is used
>>> [format(x, '.2f') for x in accumulate(inputs, logistic_map)]
['0.40', '0.91', '0.30', '0.81', '0.60', '0.92', '0.29', '0.79', '0.63',
 '0.88', '0.39', '0.90', '0.33', '0.84', '0.52', '0.95', '0.18', '0.57',
 '0.93', '0.25', '0.71', '0.79', '0.63', '0.88', '0.39', '0.91', '0.32',
 '0.83', '0.54', '0.95', '0.20', '0.60', '0.91', '0.30', '0.80', '0.60']
```

请参阅functools.reduce()类似的函数，只返回最终的累计值。

3.2版本中的新功能

*版本3.3中更改：*添加了可选的*func*参数。

itertools.**chain**(******iterables*)

创建一个迭代器，它从第一个迭代器返回元素，直到它耗尽，然后继续下一个迭代器，直到所有迭代器都耗尽。用于将连续序列作为单个序列进行处理。大致相当于：

```
def chain(*iterables):
    # chain('ABC', 'DEF') --> A B C D E F
    for it in iterables:
        for element in it:
            yield element
```

classmethod chain.**from_iterable**(*iterable*)

替代构造函数chain()。从单个迭代参数中获取链接的输入，这个参数是懒惰评估的。大致相当于：

```
def from_iterable(iterables):
    # chain.from_iterable(['ABC', 'DEF']) --> A B C D E F
    for it in iterables:
        for element in it:
            yield element
```

itertools.**combinations**(*iterable*, *r*)

从输入*迭代中*返回元素的*r*长度子序列。

组合按字典顺序排列。所以，如果输入*迭代*被排序，组合元组将按排序顺序生成。

元素根据他们的位置被视为唯一的，而不是他们的价值。因此，如果输入元素是唯一的，则每个组合中都不会有重复值。

大致相当于：

```
def combinations(iterable, r):
    # combinations('ABCD', 2) --> AB AC AD BC BD CD
    # combinations(range(4), 3) --> 012 013 023 123
    pool = tuple(iterable)
    n = len(pool)
    if r > n:
        return
    indices = list(range(r))
    yield tuple(pool[i] for i in indices)
    while True:
        for i in reversed(range(r)):
            if indices[i] != i + n - r:
                break
        else:
            return
        indices[i] += 1
        for j in range(i+1, r):
            indices[j] = indices[j-1] + 1
        yield tuple(pool[i] for i in indices)
```

代码for `combinations()` 也可以表示为`permutations()` 过滤条件后的子序列，其中元素不是按排序顺序（根据它们在输入池中的位置）：

```
def combinations(iterable, r):
    pool = tuple(iterable)
    n = len(pool)
    for indices in permutations(range(n), r):
        if sorted(indices) == list(indices):
            yield tuple(pool[i] for i in indices)
```

返回的项目数是何时 或何时为零。`n! / r! / (n-r)!0 <= r <= nr > n`

itertools.**combinations_with_replacement**(*iterable*, *r*)

从输入*迭代中*返回元素的*r*长度子序列， 允许单个元素重复多次。

组合按字典顺序排列。所以，如果输入*迭代*被排序，组合元组将按排序顺序生成。

元素根据他们的位置被视为唯一的，而不是他们的价值。因此，如果输入元素是唯一的，则生成的组合也将是唯一的。

大致相当于：

```
def combinations_with_replacement(iterable, r):
    # combinations_with_replacement('ABC', 2) --> AA AB AC BB BC CC
    pool = tuple(iterable)
    n = len(pool)
    if not n and r:
```

```
        return
    indices = [0] * r
    yield tuple(pool[i] for i in indices)
    while True:
        for i in reversed(range(r)):
            if indices[i] != n - 1:
                break
        else:
            return
        indices[i:] = [indices[i] + 1] * (r - i)
        yield tuple(pool[i] for i in indices)
```

代码for `combinations_with_replacement()` 也可以表示为`product()` 过滤条件后的子序列，其中元素不是按排序顺序（根据它们在输入池中的位置）：

```
def combinations_with_replacement(iterable, r):
    pool = tuple(iterable)
    n = len(pool)
    for indices in product(range(n), repeat=r):
        if sorted(indices) == list(indices):
            yield tuple(pool[i] for i in indices)
```

返回的项目数是何时。`(n+r-1)! / r! / (n-1)!n > 0`

版本3.1中的新功能。

itertools.**compress**（*数据*，*选择器*）

创建一个迭代器，用于过滤来自*数据的*元素，仅返回那些在*选择器*中具有相应元素的计算结果`True`。*数据*或*选择器*迭代*器*耗尽时停止。大致相当于：

```
def compress(data, selectors):
    # compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F
    return (d for d, s in zip(data, selectors) if s)
```

版本3.1中的新功能。

itertools.**count**（*start = 0*，*step = 1*）

创建一个迭代器，返回从数字*开始开始的*均匀间隔值。通常用作参数`map()` 来生成连续的数据点。此外，用于`zip()` 添加序列号。大致相当于：

```
def count(start=0, step=1):
    # count(10) --> 10 11 12 13 14 ...
    # count(2.5, 0.5) -> 2.5 3.0 3.5 ...
    n = start
    while True:
        yield n
        n += step
```

使用浮点数进行计数时，有时可以通过替换乘法代码来实现更高的精度，例如：。`(start + step * i for i in count())`

*在版本3.1中进行了更改：*添加了*步骤*参数并允许使用非整数参数。

itertools.**cycle**(*可迭代*)

使迭代器从迭代器中返回元素并保存每个元素的副本。当迭代器耗尽时，从保存的副本中返回元素。重复无限期地。大致相当于：

```
def cycle(iterable):
    # cycle('ABCD') --> A B C D A B C D A B C D ...
    saved = []
    for element in iterable:
        yield element
        saved.append(element)
    while saved:
        for element in saved:
              yield element
```

请注意，该工具包的这个成员可能需要大量的辅助存储（取决于可迭代的长度）。

itertools.**dropwhile**(*谓词*, *可迭代*)

只要谓词为真，就创建一个从迭代器中删除元素的迭代器; 之后，返回每个元素。请注意，只有谓词首先变为false时，迭代器才会产生 *任何*输出，因此可能会有较长的启动时间。大致相当于：

```
def dropwhile(predicate, iterable):
    # dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1
    iterable = iter(iterable)
    for x in iterable:
        if not predicate(x):
            yield x
            break
    for x in iterable:
        yield x
```

itertools.**filterfalse**(*谓词*, *可迭代*)

创建一个迭代器，用于过滤来自iterable的元素，仅返回谓词所针对的元素False。如果*谓词*是None，则返回false的项目。大致相当于：

```
def filterfalse(predicate, iterable):
    # filterfalse(lambda x: x%2, range(10)) --> 0 2 4 6 8
    if predicate is None:
        predicate = bool
    for x in iterable:
        if not predicate(x):
            yield x
```

itertools.**groupby**(*iterable*, *key = None*)

创建一个从迭代器中返回连续键和组的*迭代器*。的*关键*是计算每个元素的密钥值的函数。如果未指定或是None，*键*默认为恒等函数，并返回元件不变。一般来说，迭代器需要在同一个关键函数上进行排序。

其操作groupby()与uniqUnix中的过滤器类似。每当关键函数的值发生变化时它就会生成一个中断或新的组（这就是为什么通常需要使用相同的关键函数对数据进行排序的原因）。该行为与SQL的GROUP BY不同，后者聚合公共元素而不管其输入顺序如何。

返回的组本身就是一个迭代器，它共享基础迭代器groupby()。由于源是共享的，因此当groupby()对象进阶时，先前的组不再可见。因此，如果稍后需要该数据，则应将其存储为列表：

```
groups = []
uniquekeys = []
data = sorted(data, key=keyfunc)
for k, g in groupby(data, keyfunc):
    groups.append(list(g))      # Store group iterator as a list
    uniquekeys.append(k)
```

groupby()大致相当于：

```
class groupby:
    # [k for k, g in groupby('AAAABBBCCDAABBB')] --> A B C D A B
    # [list(g) for k, g in groupby('AAAABBBCCD')] --> AAAA BBB CC D
    def __init__(self, iterable, key=None):
        if key is None:
            key = lambda x: x
        self.keyfunc = key
        self.it = iter(iterable)
        self.tgtkey = self.currkey = self.currvalue = object()
    def __iter__(self):
        return self
    def __next__(self):
        while self.currkey == self.tgtkey:
            self.currvalue = next(self.it)    # Exit on StopIteration
            self.currkey = self.keyfunc(self.currvalue)
        self.tgtkey = self.currkey
        return (self.currkey, self._grouper(self.tgtkey))
    def _grouper(self, tgtkey):
        while self.currkey == tgtkey:
            yield self.currvalue
            try:
                self.currvalue = next(self.it)
            except StopIteration:
                return
            self.currkey = self.keyfunc(self.currvalue)
```

itertools.islice(*可迭代*, *停止*)

itertools.islice(*可迭代*, *开始*, *停止*[, *步骤*])

创建一个从迭代器中返回选定元素的迭代器。如果*start*不为零，则跳过迭代器中的元素直到达到起始点。之后，元素将被连续返回，除非将*步骤*设置为高于导致项目被跳过的元素。如果*停止*的None，那么迭代继续进行，直到迭代器被耗尽，如果在所有;否则，它停在指定位置。与常规切片不同，islice()不支持*开始*，*停止*或*步骤*的负值。可用于从内部结构扁平化的数据中提取相关字段（例如，多行报告可在每三行中列出名称字段）。大致相当于：

```
def islice(iterable, *args):
    # islice('ABCDEFG', 2) --> A B
    # islice('ABCDEFG', 2, 4) --> C D
    # islice('ABCDEFG', 2, None) --> C D E F G
    # islice('ABCDEFG', 0, None, 2) --> A C E G
```

```
    s = slice(*args)
    start, stop, step = s.start or 0, s.stop or sys.maxsize, s.step or 1
    it = iter(range(start, stop, step))
    try:
        nexti = next(it)
    except StopIteration:
        # Consume *iterable* up to the *start* position.
        for i, element in zip(range(start), iterable):
            pass
        return
    try:
        for i, element in enumerate(iterable):
            if i == nexti:
                yield element
                nexti = next(it)
    except StopIteration:
        # Consume to *stop*.
        for i, element in zip(range(i + 1, stop), iterable):
            pass
```

如果*开始*是None，然后重复从零开始。如果*步骤*是None，则步骤默认为1。

itertools.permutations(*可迭代*, *r*=*无*)

返回*迭代*中元素的连续*r*长度排列。

如果未指定*r*或者*r*None，则*r*默认为*可迭代*的长度，并且会生成所有可能的全长置换。

排列以词典排序顺序排列。所以，如果输入*迭代*被排序，排列元组将按排序顺序生成。

元素根据他们的位置被视为唯一的，而不是他们的价值。因此，如果输入元素是唯一的，则每个排列中都不会有重复值。

大致相当于：

```
def permutations(iterable, r=None):
    # permutations('ABCD', 2) --> AB AC AD BA BC BD CA CB CD DA DB DC
    # permutations(range(3)) --> 012 021 102 120 201 210
    pool = tuple(iterable)
    n = len(pool)
    r = n if r is None else r
    if r > n:
        return
    indices = list(range(n))
    cycles = list(range(n, n-r, -1))
    yield tuple(pool[i] for i in indices[:r])
    while n:
        for i in reversed(range(r)):
            cycles[i] -= 1
            if cycles[i] == 0:
                indices[i:] = indices[i+1:] + indices[i:i+1]
                cycles[i] = n - i
            else:
                j = cycles[i]
                indices[i], indices[-j] = indices[-j], indices[i]
                yield tuple(pool[i] for i in indices[:r])
                break
```

```
        else:
            return
```

该代码permutations()也可以表示为子序列 product() ，通过过滤排除具有重复元素的条目（来自输入池中相同位置的条目）：

```
def permutations(iterable, r=None):
    pool = tuple(iterable)
    n = len(pool)
    r = n if r is None else r
    for indices in product(range(n), repeat=r):
        if len(set(indices)) == r:
            yield tuple(pool[i] for i in indices)
```

返回的项目数是何时 或何时为零。n! / (n-r)!0 <= r <= nr > n

itertools.**product**(** iterables* , *repeat = 1*)

输入迭代的笛卡尔乘积。

大致相当于生成器表达式中的嵌套for循环。例如， 返回与。product(A, B)((x,y) for x in A for y in B)

嵌套循环像一个里程表一样循环，最右边的元素在每次迭代中前进。这种模式创建了一个词典排序，因此如果输入的可迭代序列被排序，则产品元组将按排序顺序排列。

要计算与自身的迭代产品，请使用可选的*repeat*关键字参数指定重复次数。例如， 意味着相同。product(A, repeat=4)product(A, A, A, A)

这个函数大致等价于下面的代码，只是实际的实现不会在内存中建立中间结果：

```
def product(*args, repeat=1):
    # product('ABCD', 'xy') --> Ax Ay Bx By Cx Cy Dx Dy
    # product(range(2), repeat=3) --> 000 001 010 011 100 101 110 111
    pools = [tuple(pool) for pool in args] * repeat
    result = [[]]
    for pool in pools:
        result = [x+[y] for x in result for y in pool]
    for prod in result:
        yield tuple(prod)
```

itertools.**repeat**(*object* [, *times*])

创建一个一遍又一遍地返回*对象*的迭代器。除非指定了*times*参数，否则无限期地运行。用作map()不变参数到被调用函数的参数。还用于zip()创建元组记录的不变部分。

大致相当于：

```
def repeat(object, times=None):
    # repeat(10, 3) --> 10 10 10
    if times is None:
        while True:
            yield object
    else:
```

```
        for i in range(times):
            yield object
```

*重复*使用的一个常见用途是为*映射*或*压缩*提供一个常量值流：

```
>>> list(map(pow, range(10), repeat(2)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

itertools.starmap（*函数*，*可迭代*）

创建一个迭代器，该迭代器使用从迭代器获得的参数来计算函数。用于替代map()参数参数已经从单个迭代中分组的元组（数据已被“预先压缩”）。之间的差异map()和starmap()相似之处之间的区别function(a,b)和function(*c)。大致相当于：

```
def starmap(function, iterable):
    # starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000
    for args in iterable:
        yield function(*args)
```

itertools.takewhile（*谓词*，*可迭代*）

只要谓词为真，就创建一个从迭代器返回元素的迭代器。大致相当于：

```
def takewhile(predicate, iterable):
    # takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4
    for x in iterable:
        if predicate(x):
            yield x
        else:
            break
```

itertools.tee（*可迭代*，*n = 2*）

从一个迭代中返回*n个*独立的迭代器。

下面的Python代码有助于解释什么是*TEE*（尽管实际实现更复杂，只使用一个底层的 FIFO 队列）。

大致相当于：

```
def tee(iterable, n=2):
    it = iter(iterable)
    deques = [collections.deque() for i in range(n)]
    def gen(mydeque):
        while True:
            if not mydeque:             # when the local deque is empty
                try:
                    newval = next(it)   # fetch a new value and
                except StopIteration:
                    return
                for d in deques:        # load it to all the deques
                    d.append(newval)
            yield mydeque.popleft()
    return tuple(gen(d) for d in deques)
```

一旦tee()进行了拆分，原始*迭代器*不应该用于其他任何地方; 否则，可以在没有通知T恤对象的情况下使*迭代*得到提前。

这个itertool可能需要大量的辅助存储（取决于需要存储多少临时数据）。一般来说，如果一个迭代器使用了大部分或全部数据的另一迭代开始前，它是更快地使用 list() 替代 tee()。

itertools.**zip_longest**(**iterables*, *fillvalue = None*)

制作一个迭代器，用于聚合来自每个迭代器的元素。如果迭代的长度不均匀，缺少的值将用*fillvalue*填充。迭代继续下去，直到最长的迭代耗尽。大致相当于：

```
class ZipExhausted(Exception):
    pass

def zip_longest(*args, **kwds):
    # zip_longest('ABCD', 'xy', fillvalue='-') --> Ax By C- D-
    fillvalue = kwds.get('fillvalue')
    counter = len(args) - 1
    def sentinel():
        nonlocal counter
        if not counter:
            raise ZipExhausted
        counter -= 1
        yield fillvalue
    fillers = repeat(fillvalue)
    iterators = [chain(it, sentinel(), fillers) for it in args]
    try:
        while iterators:
            yield tuple(map(next, iterators))
    except ZipExhausted:
        pass
```

如果其中一个迭代可能是无限的，那么该zip_longest() 函数应该包含限制调用次数的内容（例如islice()或takewhile()）。如果未指定，则 *fillvalue*默认为None。

10.1.2。Itertools食谱

本部分显示了使用现有itertools作为构建块来创建扩展工具集的配方。

扩展工具提供了与底层工具集相同的高性能。优越的内存性能是通过一次处理一个元素来保持的，而不是一次将整个迭代器放入内存中。通过将这些工具链接在一起，以一种功能性风格帮助消除临时变量，从而使代码量保持较小。通过使用for循环和发生器来优化 “矢量化”积木，从而保留高速度，这会导致解释器开销。

```
def take(n, iterable):
    "Return first n items of the iterable as a list"
    return list(islice(iterable, n))

def prepend(value, iterator):
    "Prepend a single value in front of an iterator"
    # prepend(1, [2, 3, 4]) -> 1 2 3 4
    return chain([value], iterator)
```

```
def tabulate(function, start=0):
    "Return function(0), function(1), ..."
    return map(function, count(start))

def tail(n, iterable):
    "Return an iterator over the last n items"
    # tail(3, 'ABCDEFG') --> E F G
    return iter(collections.deque(iterable, maxlen=n))

def consume(iterator, n=None):
    "Advance the iterator n-steps ahead. If n is None, consume entirely."
    # Use functions that consume iterators at C speed.
    if n is None:
        # feed the entire iterator into a zero-length deque
        collections.deque(iterator, maxlen=0)
    else:
        # advance to the empty slice starting at position n
        next(islice(iterator, n, n), None)

def nth(iterable, n, default=None):
    "Returns the nth item or a default value"
    return next(islice(iterable, n, None), default)

def all_equal(iterable):
    "Returns True if all the elements are equal to each other"
    g = groupby(iterable)
    return next(g, True) and not next(g, False)

def quantify(iterable, pred=bool):
    "Count how many times the predicate is true"
    return sum(map(pred, iterable))

def padnone(iterable):
    """Returns the sequence elements and then returns None indefinitely.

    Useful for emulating the behavior of the built-in map() function.
    """
    return chain(iterable, repeat(None))

def ncycles(iterable, n):
    "Returns the sequence elements n times"
    return chain.from_iterable(repeat(tuple(iterable), n))

def dotproduct(vec1, vec2):
    return sum(map(operator.mul, vec1, vec2))

def flatten(listOfLists):
    "Flatten one level of nesting"
    return chain.from_iterable(listOfLists)

def repeatfunc(func, times=None, *args):
    """Repeat calls to func with specified arguments.

    Example:  repeatfunc(random.random)
    """
    if times is None:
        return starmap(func, repeat(args))
```

```
    return starmap(func, repeat(args, times))

def pairwise(iterable):
    "s -> (s0,s1), (s1,s2), (s2, s3), ..."
    a, b = tee(iterable)
    next(b, None)
    return zip(a, b)

def grouper(iterable, n, fillvalue=None):
    "Collect data into fixed-length chunks or blocks"
    # grouper('ABCDEFG', 3, 'x') --> ABC DEF Gxx"
    args = [iter(iterable)] * n
    return zip_longest(*args, fillvalue=fillvalue)

def roundrobin(*iterables):
    "roundrobin('ABC', 'D', 'EF') --> A D E B F C"
    # Recipe credited to George Sakkis
    num_active = len(iterables)
    nexts = cycle(iter(it).__next__ for it in iterables)
    while num_active:
        try:
            for next in nexts:
                yield next()
        except StopIteration:
            # Remove the iterator we just exhausted from the cycle.
            num_active -= 1
            nexts = cycle(islice(nexts, num_active))

def partition(pred, iterable):
    'Use a predicate to partition entries into false entries and true entries'
    # partition(is_odd, range(10)) --> 0 2 4 6 8   and  1 3 5 7 9
    t1, t2 = tee(iterable)
    return filterfalse(pred, t1), filter(pred, t2)

def powerset(iterable):
    "powerset([1,2,3]) --> () (1,) (2,) (3,) (1,2) (1,3) (2,3) (1,2,3)"
    s = list(iterable)
    return chain.from_iterable(combinations(s, r) for r in range(len(s)+1))

def unique_everseen(iterable, key=None):
    "List unique elements, preserving order. Remember all elements ever seen."
    # unique_everseen('AAAABBBCCDAABBB') --> A B C D
    # unique_everseen('ABBCcAD', str.lower) --> A B C D
    seen = set()
    seen_add = seen.add
    if key is None:
        for element in filterfalse(seen.__contains__, iterable):
            seen_add(element)
            yield element
    else:
        for element in iterable:
            k = key(element)
            if k not in seen:
                seen_add(k)
                yield element

def unique_justseen(iterable, key=None):
    "List unique elements, preserving order. Remember only the element just seen."
```

```
    # unique_justseen('AAAABBBCCDAABBB') --> A B C D A B
    # unique_justseen('ABBCcAD', str.lower) --> A B C A D
    return map(next, map(itemgetter(1), groupby(iterable, key)))

def iter_except(func, exception, first=None):
    """ Call a function repeatedly until an exception is raised.

    Converts a call-until-exception interface to an iterator interface.
    Like builtins.iter(func, sentinel) but uses an exception instead
    of a sentinel to end the loop.

    Examples:
        iter_except(functools.partial(heappop, h), IndexError)   # priority queue ite
        iter_except(d.popitem, KeyError)                         # non-blocking dict
        iter_except(d.popleft, IndexError)                       # non-blocking deque
        iter_except(q.get_nowait, Queue.Empty)                   # loop over a produc
        iter_except(s.pop, KeyError)                             # non-blocking set i

    """
    try:
        if first is not None:
            yield first()            # For database APIs needing an initial cast to d
        while True:
            yield func()
    except exception:
        pass

def first_true(iterable, default=False, pred=None):
    """Returns the first true value in the iterable.

    If no true value is found, returns *default*

    If *pred* is not None, returns the first item
    for which pred(item) is true.

    """
    # first_true([a,b,c], x) --> a or b or c or x
    # first_true([a,b], x, f) --> a if f(a) else b if f(b) else x
    return next(filter(pred, iterable), default)

def random_product(*args, repeat=1):
    "Random selection from itertools.product(*args, **kwds)"
    pools = [tuple(pool) for pool in args] * repeat
    return tuple(random.choice(pool) for pool in pools)

def random_permutation(iterable, r=None):
    "Random selection from itertools.permutations(iterable, r)"
    pool = tuple(iterable)
    r = len(pool) if r is None else r
    return tuple(random.sample(pool, r))

def random_combination(iterable, r):
    "Random selection from itertools.combinations(iterable, r)"
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.sample(range(n), r))
    return tuple(pool[i] for i in indices)
```

```
def random_combination_with_replacement(iterable, r):
    "Random selection from itertools.combinations_with_replacement(iterable, r)"
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.randrange(n) for i in range(r))
    return tuple(pool[i] for i in indices)

def nth_combination(iterable, r, index):
    'Equivalent to list(combinations(iterable, r))[index]'
    pool = tuple(iterable)
    n = len(pool)
    if r < 0 or r > n:
        raise ValueError
    c = 1
    k = min(r, n-r)
    for i in range(1, k+1):
        c = c * (n - k + i) // i
    if index < 0:
        index += c
    if index < 0 or index >= c:
        raise IndexError
    result = []
    while r:
        c, n, r = c*r//n, n-1, r-1
        while index >= c:
            index -= c
            c, n = c*(n-r)//n, n-1
        result.append(pool[-1-n])
    return tuple(result)
```

请注意，上述许多配方可以通过将全局查找替换为定义为默认值的局部变量进行优化。例如，*dotproduct*配方可以写为：

```
def dotproduct(vec1, vec2, sum=sum, map=map, mul=operator.mul):
    return sum(map(mul, vec1, vec2))
```

10.2。functools- 可调用对象的高阶函数和操作

源代码： Lib / functools.py

该functools模块用于高阶函数：作用于或返回其他函数的函数。一般而言，任何可调用对象都可以作为本模块用途的函数来处理。

该functools模块定义了以下功能：

functools.**cmp_to_key**(*func*)

将旧式比较功能转换为按键功能。使用接受钥匙功能的工具（如sorted()，min()，max()，heapq.nlargest()，heapq.nsmallest()， itertools.groupby()）。该函数主要用作从支持使用比较函数的Python 2转换而来的程序的转换工具。

比较函数是可接受两个参数的任何可调用函数，比较它们，并返回一个负数，小于等于零，或者返回大于等于零的正数。关键函数是可调用的，它接受一个参数并返回另一个值作为排序关键字。

例：

```
sorted(iterable, key=cmp_to_key(locale.strcoll))  # locale-aware sort order
```

有关排序示例和简要的排序教程，请参阅对如何排序。

3.2版本中的新功能

@functools.**lru_cache**(*maxsize = 128*，*typed = False*)

装饰包裹可赎回一个memoizing，节省高达功能 *MAXSIZE*最近通话。当使用相同的参数周期性地调用昂贵的或I / O绑定函数时，它可以节省时间。

由于字典用于缓存结果，函数的位置和关键字参数必须是可散列的。

如果*maxsize*设置为None，LRU功能被禁用，并且缓存可以无限制增长。当*maxsize*是“2的幂数”时，LRU功能表现最佳。

如果*typed*设置为true，则不同类型的函数参数将被分别缓存。例如，f(3)并且f(3.0)将被视为产生截然不同的结果不同的呼叫。

为了帮助衡量缓存的有效性并调整*maxsize* 参数，被包装的函数被装备了一个cache_info() 函数，该函数返回一个命名的元组，显示*命中*，*未命中*， *最大大小*和*currsize*。在多线程环境中，命中和未命中是近似的。

装饰器还提供cache_clear()清除或使缓存失效的功能。

原始的底层功能可以通过__wrapped__属性访问 。这对于自省，绕过缓存或用不同缓存重新包装函数很有用。

一个LRU（最近最少使用）缓存效果最好的最近通话是即将到来的呼叫的最佳预测（例如，新闻服务器上的最流行的文章往往每天更换）。缓存的大小限制可确保缓存在长时间运行的流程（如Web服务器）上不会增长。

静态网页内容的LRU缓存示例：

```
@lru_cache(maxsize=32)
def get_pep(num):
    'Retrieve text of a Python Enhancement Proposal'
    resource = 'http://www.python.org/dev/peps/pep-%04d/' % num
    try:
        with urllib.request.urlopen(resource) as s:
            return s.read()
    except urllib.error.HTTPError:
        return 'Not Found'

>>> for n in 8, 290, 308, 320, 8, 218, 320, 279, 289, 320, 9991:
...     pep = get_pep(n)
...     print(n, len(pep))

>>> get_pep.cache_info()
CacheInfo(hits=3, misses=8, maxsize=32, currsize=8)
```

使用高速缓存实现 动态编程 技术高效计算斐波那契数的示例 ：

```
@lru_cache(maxsize=None)
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)

>>> [fib(n) for n in range(16)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]

>>> fib.cache_info()
CacheInfo(hits=28, misses=16, maxsize=None, currsize=16)
```

3.2版本中的新功能

*版本3.3中更改：*添加了*键入的*选项。

@functools.total_ordering

给定一个定义一个或多个比较排序方法的类，这个类装饰器提供剩余的。这简化了指定所有可能的丰富比较操作的工作：

这个类必须定义之一__lt__()，__le__()，__gt__()，或__ge__()。另外，班级应该提供一种__eq__()方法。

例如：

```
@total_ordering
class Student:
    def _is_valid_operand(self, other):
        return (hasattr(other, "lastname") and
```

```
                hasattr(other, "firstname"))
    def __eq__(self, other):
        if not self._is_valid_operand(other):
            return NotImplemented
        return ((self.lastname.lower(), self.firstname.lower()) ==
                (other.lastname.lower(), other.firstname.lower()))
    def __lt__(self, other):
        if not self._is_valid_operand(other):
            return NotImplemented
        return ((self.lastname.lower(), self.firstname.lower()) <
                (other.lastname.lower(), other.firstname.lower()))
```

注意： 虽然这个装饰可以很容易地创建很乖全序类型，它*确实*来得比较慢的执行和更复杂的堆栈跟踪派生比较方法的成本。如果性能基准测试表明这是给定应用程序的瓶颈，那么实施全部六种丰富的比较方法可能会提供轻松的速度提升。

3.2版本中的新功能

*版本3.4中已更改：*现在支持从无法识别的类型的基础比较函数中返回NotImplemented。

functools.**partial**(*func*, **args*, ***关键字*)

返回一个新的partial对象，当被调用时，它的行为就像 调用参数*args*和关键字参数*关键字*的*func*。如果更多的参数被提供给调用，它们被附加到*参数*。如果提供了其他关键字参数，它们将扩展并覆盖*关键字*。大致相当于：

```
def partial(func, *args, **keywords):
    def newfunc(*fargs, **fkeywords):
        newkeywords = keywords.copy()
        newkeywords.update(fkeywords)
        return func(*args, *fargs, **newkeywords)
    newfunc.func = func
    newfunc.args = args
    newfunc.keywords = keywords
    return newfunc
```

所述partial()用于局部功能应用其中“冻结”的函数的参数和/或产生具有简化签名的新对象的关键字一些部分。例如，partial()可用于创建一个可调用的行为，int()其中的*基本*参数默认为两个：

```
>>> from functools import partial
>>> basetwo = partial(int, base=2)
>>> basetwo.__doc__ = 'Convert base 2 string to an int.'
>>> basetwo('10010')
18
```

*类*functools.**partialmethod**(*func*, **args*, ***关键字*)

返回一个新的partialmethod描述符，其行为类似于partial它被设计为用作方法定义而不是直接可调用的。

*func*必须是描述符或可调用对象（它们与普通函数一样都是作为描述符来处理）。

当*FUNC*是一个描述符（例如正常Python函数，`classmethod()`，`staticmethod()`，`abstractmethod()`或的另一个实例`partialmethod`），则调用`__get__`委派到底层描述符，和一个适当的`partial`对象作为结果返回。

当*func*是一个非描述符可调用时，动态创建一个适当的绑定方法。当用作方法时，它的行为与普通的Python函数相似：即使在提供给构造函数的*参数*和*关键字*之前，*self*参数将作为第一个位置参数插入。`partialmethod`

例：

```
>>> class Cell(object):
...     def __init__(self):
...         self._alive = False
...     @property
...     def alive(self):
...         return self._alive
...     def set_state(self, state):
...         self._alive = bool(state)
...     set_alive = partialmethod(set_state, True)
...     set_dead = partialmethod(set_state, False)
...
>>> c = Cell()
>>> c.alive
False
>>> c.set_alive()
>>> c.alive
True
```

3.4版新增功能

functools.**reduce**(*function*, *iterable*[, *initializer*])

应用*功能*的两个参数来累计的项目*序列*，从左至右，这样的顺序，以减少单个值。例如，计算。左边的参数*x*是累加值，右边的参数*y*是*序列中*的更新值。如果存在可选的*初始*值设定项，则将其置于计算中序列的项目之前，并在序列为空时用作默认值。如果*初始化*程序没有给出，并且*序列*只包含一个项目，则返回第一个项目。`reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])`((((1+2)+3)+4)+5)

大致相当于：

```
def reduce(function, iterable, initializer=None):
    it = iter(iterable)
    if initializer is None:
        value = next(it)
    else:
        value = initializer
    for element in it:
        value = function(value, element)
    return value
```

@functools.**singledispatch**

将函数转换为单一调度 通用函数。

要定义一个通用函数，请用@singledispatch 装饰器对其进行修饰。请注意，调度发生在第一个参数的类型上，相应地创建你的函数：

```
>>> from functools import singledispatch
>>> @singledispatch
... def fun(arg, verbose=False):
...     if verbose:
...         print("Let me just say,", end=" ")
...     print(arg)
```

要将重载实现添加到该函数，请使用register() 通用函数的属性。它是一个装饰器，接受一个类型参数并装饰一个实现该类型操作的函数：

```
>>> @fun.register(int)
... def _(arg, verbose=False):
...     if verbose:
...         print("Strength in numbers, eh?", end=" ")
...     print(arg)
...
>>> @fun.register(list)
... def _(arg, verbose=False):
...     if verbose:
...         print("Enumerate this:")
...     for i, elem in enumerate(arg):
...         print(i, elem)
```

要启用注册lambda表达式和预先存在的函数，register()可以以功能形式使用该 属性：

```
>>> def nothing(arg, verbose=False):
...     print("Nothing.")
...
>>> fun.register(type(None), nothing)
```

这个register()属性返回的是未修饰的函数，它使装饰器堆叠，酸洗，以及独立地为每个变体创建单元测试：

```
>>> @fun.register(float)
... @fun.register(Decimal)
... def fun_num(arg, verbose=False):
...     if verbose:
...         print("Half of your number:", end=" ")
...     print(arg / 2)
...
>>> fun_num is fun
False
```

在调用时，泛型函数调度第一个参数的类型：

```
>>> fun("Hello, world.")
Hello, world.
>>> fun("test.", verbose=True)
Let me just say, test.
>>> fun(42, verbose=True)
Strength in numbers, eh? 42
```

```
>>> fun(['spam', 'spam', 'eggs', 'spam'], verbose=True)
Enumerate this:
0 spam
1 spam
2 eggs
3 spam
>>> fun(None)
Nothing.
>>> fun(1.23)
0.615
```

在没有针对特定类型的注册实现的情况下，其方法解析顺序用于找到更通用的实现。装饰的原始函数`@singledispatch`注册为基`object`类型，这意味着如果没有找到更好的实现，则使用它。

要检查泛型函数为给定类型选择哪个实现，请使用`dispatch()`属性：

```
>>> fun.dispatch(float)
<function fun_num at 0x1035a2840>
>>> fun.dispatch(dict)    # note: default implementation
<function fun at 0x103fe0000>
```

要访问所有注册的实现，请使用只读`registry` 属性：

```
>>> fun.registry.keys()
dict_keys([<class 'NoneType'>, <class 'int'>, <class 'object'>,
           <class 'decimal.Decimal'>, <class 'list'>,
           <class 'float'>])
>>> fun.registry[float]
<function fun_num at 0x1035a2840>
>>> fun.registry[object]
<function fun at 0x103fe0000>
```

3.4版新增功能

functools.**update_wrapper**(*wrapper*, *wrapped*, *assigned* = *WRAPPER_ASSIGNMENTS*, *updated* = *WRAPPER_UPDATES*)

将*包装*函数更新为看起来像*包装*函数。可选参数是元组，用于指定将原始函数的哪些属性直接分配给包装函数上的匹配属性，以及使用原始函数的相应属性更新包装函数的哪些属性。这些参数的默认值是模块级的常量`WRAPPER_ASSIGNMENTS`（其中分配给包装函数的`__module__`，`__name__`，`__qualname__`，`__annotations__`和`__doc__`，文档字符串）和`WRAPPER_UPDATES`（其更新的包装函数的`__dict__`，即实例字典）。

为了允许访问原始函数进行自省和其他目的（例如绕过缓存装饰器等`lru_cache()`），该函数会自动`__wrapped__`向封装器添加一个属性，该属性指向被封装的函数。

该函数的主要用途是装饰器函数，它包装装饰后的函数并返回包装器。如果包装函数未更新，则返回的函数的元数据将反映包装器定义而不是原始函数定义，这通常不太有用。

`update_wrapper()`可以与功能以外的可调用一起使用。在被包装的对象中丢失的*分配*或*更新*中*指定的*任何属性都将被忽略（即，此函数不会尝试将它们设置在包装函数上）。`AttributeError`如果包装函数本身缺少*更新中*指定的任何属性，则仍然引发。

*3.2版新增功能：*自动添加__wrapped__属性。

*版本3.2中的新功能：*__annotations__默认情况下复制属性。

*在版本3.2中更改：*缺少属性不再触发AttributeError。

*在版本3.4中更改：*__wrapped__即使该函数定义了一个__wrapped__属性，该属性现在始终引用包装的函数。（见bpo-17482）

@functools.wraps(*包装*，*分配= WRAPPER_ASSIGNMENTS*，*更新= WRAPPER_UPDATES*)

这是update_wrapper()在定义包装函数时作为函数装饰器调用的便利函数。这相当于。例如：partial(update_wrapper, wrapped=wrapped, assigned=assigned, updated=updated)

```
>>> from functools import wraps
>>> def my_decorator(f):
...     @wraps(f)
...     def wrapper(*args, **kwds):
...         print('Calling decorated function')
...         return f(*args, **kwds)
...     return wrapper
...
>>> @my_decorator
... def example():
...     """Docstring"""
...     print('Called example function')
...
>>> example()
Calling decorated function
Called example function
>>> example.__name__
'example'
>>> example.__doc__
'Docstring'
```

如果没有使用这个装饰器工厂，示例函数的名称应该是这样'wrapper'，并且原始文档字符串example()已经丢失。

10.2.1。partial对象

partial对象是由可创建的可调用对象partial()。他们有三个只读属性：

partial.func

可调用的对象或函数。调用partial对象将被转发给func新的参数和关键字。

partial.args

将被提供给partial对象调用的位置参数前置的最左边的位置参数。

partial.keywords

partial调用对象时将提供的关键字参数。

partial对象就像function对象一样可以被调用，弱引用，并且可以有属性。有一些重要的区别。例如，__name__和__doc__属性不会自动创建。此外，partial类中定义的对象的行为与静态方法相似，并且在实例属性查找期间不会转换为绑定方法。

10.3。operator- 标准操作符作为函数

源代码： Lib / operator.py

该operator模块导出一组对应于Python内部运算符的高效函数。例如，相当于表达式。许多函数名称是用于特殊方法的函数名称，没有双下划线。为了向后兼容，其中很多都有保留双下划线的变体。为了清晰起见，没有双下划线的变体是优选的。 operator.add(x, y) x+y

这些函数分为执行对象比较，逻辑运算，数学运算和序列运算的类别。

对象比较函数对于所有对象都很有用，并且以它们支持的丰富比较运算符命名：

operator.lt (*a* , *b*)
operator.le (*a* , *b*)
operator.eq (*a* , *b*)
operator.ne (*a* , *b*)
operator.ge (*a* , *b*)
operator.gt (*a* , *b*)
operator.__lt__ (*a* , *b*)
operator.__le__ (*a* , *b*)
operator.__eq__ (*a* , *b*)
operator.__ne__ (*a* , *b*)
operator.__ge__ (*a* , *b*)
operator.__gt__ (*a* , *b*)

在*a*和*b*之间执行“丰富的比较”。具体而言，就相当于，等同于，相当于，等同于， 相当于和相当于。请注意，这些函数可以返回任何值，这些值可以或不可以作为布尔值来解释。有关富比较的更多信息，请参阅 比较。 lt(a, b) a < b le(a, b) a <= b eq(a, b) a == b ne(a, b) a != b gt(a, b) a > b ge(a, b) a >= b

逻辑操作通常也适用于所有对象，并支持真值测试，身份测试和布尔操作：

operator.not_ (*obj*)
operator.__not__ (*obj*)

返回*obj*的结果。（注意对象实例没有 方法;只有解释器内核定义了这个操作，结果受到和方法的影响。）not __not__() __bool__() __len__()

operator.truth (*obj*)

True如果*obj*为真，False则返回，否则返回。这相当于使用bool构造函数。

operator.is_ (*a* , *b*)

返回。测试对象标识。 a is b

operator.is_not (*a* , *b*)

返回。测试对象标识。 a is not b

数学和位运算是最多的：

operator.abs(*obj*)
operator.__abs__(*obj*)
返回*obj*的绝对值。

operator.add(*a*, *b*)
operator.__add__(*a*, *b*)
返回，用于一个和*b*的数字。a + b

operator.and_(*a*, *b*)
operator.__and__(*a*, *b*)
返回按位和*a*和*b*。

operator.floordiv(*a*, *b*)
operator.__floordiv__(*a*, *b*)
返回。a // b

operator.index(*a*)
operator.__index__(*a*)
返回一个转换为整数。相当于a.__index__()。

operator.inv(*obj*)
operator.invert(*obj*)
operator.__inv__(*obj*)
operator.__invert__(*obj*)
返回数字*obj*的按位倒数。这相当于~obj。

operator.lshift(*a*, *b*)
operator.__lshift__(*a*, *b*)
返回一个左移*b*。

operator.mod(*a*, *b*)
operator.__mod__(*a*, *b*)
返回。a % b

operator.mul(*a*, *b*)
operator.__mul__(*a*, *b*)
返回，用于一个和*b*的数字。a * b

operator.matmul(*a*, *b*)
operator.__matmul__(*a*, *b*)
返回。a @ b

3.5版本中的新功能。

operator.neg(*obj*)
operator.__neg__(*obj*)
返回*obj*否定(-obj)。

operator.or_(*a*, *b*)
operator.__or__(*a*, *b*)
按位或返回*a*和*b*。

operator.pos(*obj*)
operator.__pos__(*obj*)
返回*obj* positive(+obj)。

operator.pow(*a*, *b*)
operator.__pow__(*a*, *b*)
返回，用于一个*a*和*b*的数字。a ** b

operator.rshift(*a*, *b*)
operator.__rshift__(*a*, *b*)
返回一个由右移*b*。

operator.sub(*a*, *b*)
operator.__sub__(*a*, *b*)
返回。a - b

operator.truediv(*a*, *b*)
operator.__truediv__(*a*, *b*)
返回2/3是.66而不是0.这也被称为“真实”分割。a / b

operator.xor(*a*, *b*)
operator.__xor__(*a*, *b*)
返回按位独占或*a*和*b*。

与序列一起工作的操作(其中一些与映射也一样)包括：

operator.concat(*a*, *b*)
operator.__concat__(*a*, *b*)
返回为一个*a*和*b*序列。a + b

operator.contains(*a*, *b*)
operator.__contains__(*a*, *b*)
返回测试的结果。请注意颠倒的操作数。b in a

operator.countOf(*a*, *b*)
返回的出现次数*b*中的一个。

operator.delitem(*a*, *b*)

operator.__delitem__(*a*, *b*)
除去的值一个索引*b*。

operator.getitem(*a*, *b*)
operator.__getitem__(*a*, *b*)
返回的值一个索引*b*。

operator.indexOf(*a*, *b*)
返回第一个的发生的索引*b*中一个。

operator.setitem(*a*, *b*, *c*)
operator.__setitem__(*a*, *b*, *c*)
设定的值一个索引*b*到Ç。

operator.length_hint(*obj*, *default = 0*)
返回对象*o*的估计长度。首先尝试返回其实际长度，然后使用估计值 object.__length_hint__()，最后返回默认值。

3.4版新增功能

该operator模块还定义了用于通用属性和项目查找的工具。这些是使现场快速提取作为参数都非常有用 map()，sorted()，itertools.groupby()，或期望的功能参数等功能。

operator.attrgetter(*attr*)
operator.attrgetter(**attrs*)
返回一个可从其操作数中获取*attr*的可调用对象。如果请求多个属性，则返回一个属性元组。属性名称也可以包含点。例如：

- 之后，通话返回。f = attrgetter('name')f(b)b.name
- 之后，通话返回。f = attrgetter('name', 'date')f(b)(b.name, b.date)
- 之后，通话 返回。f = attrgetter('name.first', 'name.last')f(b)(b.name.first, b.name.last)

相当于：

```
def attrgetter(*items):
    if any(not isinstance(item, str) for item in items):
        raise TypeError('attribute name must be a string')
    if len(items) == 1:
        attr = items[0]
        def g(obj):
            return resolve_attr(obj, attr)
    else:
        def g(obj):
            return tuple(resolve_attr(obj, attr) for attr in items)
    return g

def resolve_attr(obj, attr):
    for name in attr.split("."):
        obj = getattr(obj, name)
    return obj
```

operator.**itemgetter**(*item*)
operator.**itemgetter**(**项目*)

返回一个可调用的对象，使用操作数的方法从其操作数中获取*项目*__getitem__()。如果指定了多个项目，则返回查找值的元组。例如：

- 之后，通话返回。f = itemgetter(2)f(r)r[2]
- 之后，通话返回。g = itemgetter(2, 5, 3)g(r)(r[2], r[5], r[3])

相当于：

```
def itemgetter(*items):
    if len(items) == 1:
        item = items[0]
        def g(obj):
            return obj[item]
    else:
        def g(obj):
            return tuple(obj[item] for item in items)
    return g
```

这些项目可以是操作数__getitem__() 方法接受的任何类型。字典接受任何可排序的值。列表，元组和字符串接受索引或片：

```
>>> itemgetter(1)('ABCDEFG')
'B'
>>> itemgetter(1,3,5)('ABCDEFG')
('B', 'D', 'F')
>>> itemgetter(slice(2,None))('ABCDEFG')
'CDEFG'
```

使用itemgetter()从元组记录中检索特定字段的示例：

```
>>> inventory = [('apple', 3), ('banana', 2), ('pear', 5), ('orange', 1)]
>>> getcount = itemgetter(1)
>>> list(map(getcount, inventory))
[3, 2, 5, 1]
>>> sorted(inventory, key=getcount)
[('orange', 1), ('banana', 2), ('apple', 3), ('pear', 5)]
```

operator.**methodcaller**(*name*[, *args* ...])

返回一个可调用对象，它在其操作数上调用方法*名称*。如果给出了额外的参数和/或关键字参数，它们也会被赋予该方法。例如：

- 之后，通话返回。f = methodcaller('name')f(b)b.name()
- 之后，通话返回。f = methodcaller('name', 'foo', bar=1)f(b)b.name('foo', bar=1)

相当于：

```
def methodcaller(name, *args, **kwargs):
    def caller(obj):
        return getattr(obj, name)(*args, **kwargs)
    return caller
```

10.3.1。将运算符映射到函数

此表显示了抽象操作在Python语法和operator模块中的函数中如何对应运算符。

手术	句法	功能
加成	a + b	add(a, b)
级联	seq1 + seq2	concat(seq1, seq2)
遏制测试	obj in seq	contains(seq, obj)
师	a / b	truediv(a, b)
师	a // b	floordiv(a, b)
按位和	a & b	and_(a, b)
按位独占或	a ^ b	xor(a, b)
按位反转	~ a	invert(a)
按位或	a | b	or_(a, b)
幂	a ** b	pow(a, b)
身分	a is b	is_(a, b)
身分	a is not b	is_not(a, b)
索引分配	obj[k] = v	setitem(obj, k, v)
索引删除	del obj[k]	delitem(obj, k)
索引	obj[k]	getitem(obj, k)
左移	a << b	lshift(a, b)
模	a % b	mod(a, b)
乘法	a * b	mul(a, b)
矩阵乘法	a @ b	matmul(a, b)
否定（算术）	- a	neg(a)
否定（逻辑）	not a	not_(a)
正	+ a	pos(a)
右移	a >> b	rshift(a, b)
切片分配	seq[i:j] = values	setitem(seq, slice(i, j), values)
切片删除	del seq[i:j]	delitem(seq, slice(i, j))
切片	seq[i:j]	getitem(seq, slice(i, j))
字符串格式	s % obj	mod(s, obj)
减法	a - b	sub(a, b)
真相测验	obj	truth(obj)
订购	a < b	lt(a, b)
订购	a <= b	le(a, b)
平等	a == b	eq(a, b)
区别	a != b	ne(a, b)
订购	a >= b	ge(a, b)
订购	a > b	gt(a, b)

10.3.2。就地运营商

许多操作都有一个“就地”版本。下面列出的是比通常的语法提供对原地操作符的更原始访问的函数；例如，该陈述 相当于 。另一种说法就是说这 相当于复合语句 。 x += yx = operator.iadd(x, y)z = operator.iadd(x, y)z = x; z += y

在这些示例中，请注意，调用就地方法时，计算和分配分两步执行。下面列出的就地功能只做第一步，调用就地方法。第二步，分配，不处理。

对于不可变目标，如字符串，数字和元组，计算更新后的值，但不会将其分配回输入变量：

```
>>> a = 'hello'
>>> iadd(a, ' world')
'hello world'
>>> a
'hello'
```

对于可变目标（如列表和词典），inplace方法将执行更新，因此不需要进行后续分配：

```
>>> s = ['h', 'e', 'l', 'l', 'o']
>>> iadd(s, [' ', 'w', 'o', 'r', 'l', 'd'])
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
>>> s
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
```

operator.**iadd**(*a*, *b*)
operator.**__iadd__**(*a*, *b*)
a = iadd(a, b)相当于。a += b

operator.**iand**(*a*, *b*)
operator.**__iand__**(*a*, *b*)
a = iand(a, b)相当于。a &= b

operator.**iconcat**(*a*, *b*)
operator.**__iconcat__**(*a*, *b*)
a = iconcat(a, b)相当于用于一个和b序列。a += b

operator.**ifloordiv**(*a*, *b*)
operator.**__ifloordiv__**(*a*, *b*)
a = ifloordiv(a, b)相当于。a //= b

operator.**ilshift**(*a*, *b*)
operator.**__ilshift__**(*a*, *b*)
a = ilshift(a, b)相当于。a <<= b

operator.**imod**(*a*, *b*)
operator.**__imod__**(*a*, *b*)

a = imod(a, b)相当于。a %= b

operator.imul(*a*, *b*)
operator.__imul__(*a*, *b*)
a = imul(a, b)相当于。a *= b

operator.imatmul(*a*, *b*)
operator.__imatmul__(*a*, *b*)
a = imatmul(a, b)相当于。a @= b

3.5版本中的新功能。

operator.ior(*a*, *b*)
operator.__ior__(*a*, *b*)
a = ior(a, b)相当于。a |= b

operator.ipow(*a*, *b*)
operator.__ipow__(*a*, *b*)
a = ipow(a, b)相当于。a **= b

operator.irshift(*a*, *b*)
operator.__irshift__(*a*, *b*)
a = irshift(a, b)相当于。a >>= b

operator.isub(*a*, *b*)
operator.__isub__(*a*, *b*)
a = isub(a, b)相当于。a -= b

operator.itruediv(*a*, *b*)
operator.__itruediv__(*a*, *b*)
a = itruediv(a, b)相当于。a /= b

operator.ixor(*a*, *b*)
operator.__ixor__(*a*, *b*)
a = ixor(a, b)相当于。a ^= b

11.文件和目录访问

本章介绍的模块处理磁盘文件和目录。例如，有用于读取文件属性，以便携方式操作路径以及创建临时文件的模块。本章中完整的模块列表是：

也可以看看:

模 os
操作系统接口，包括用于处理比Python 文件对象更低级别的文件的函数。

模 io
Python内置的I / O库，包括抽象类和一些具体的类，比如文件I / O。

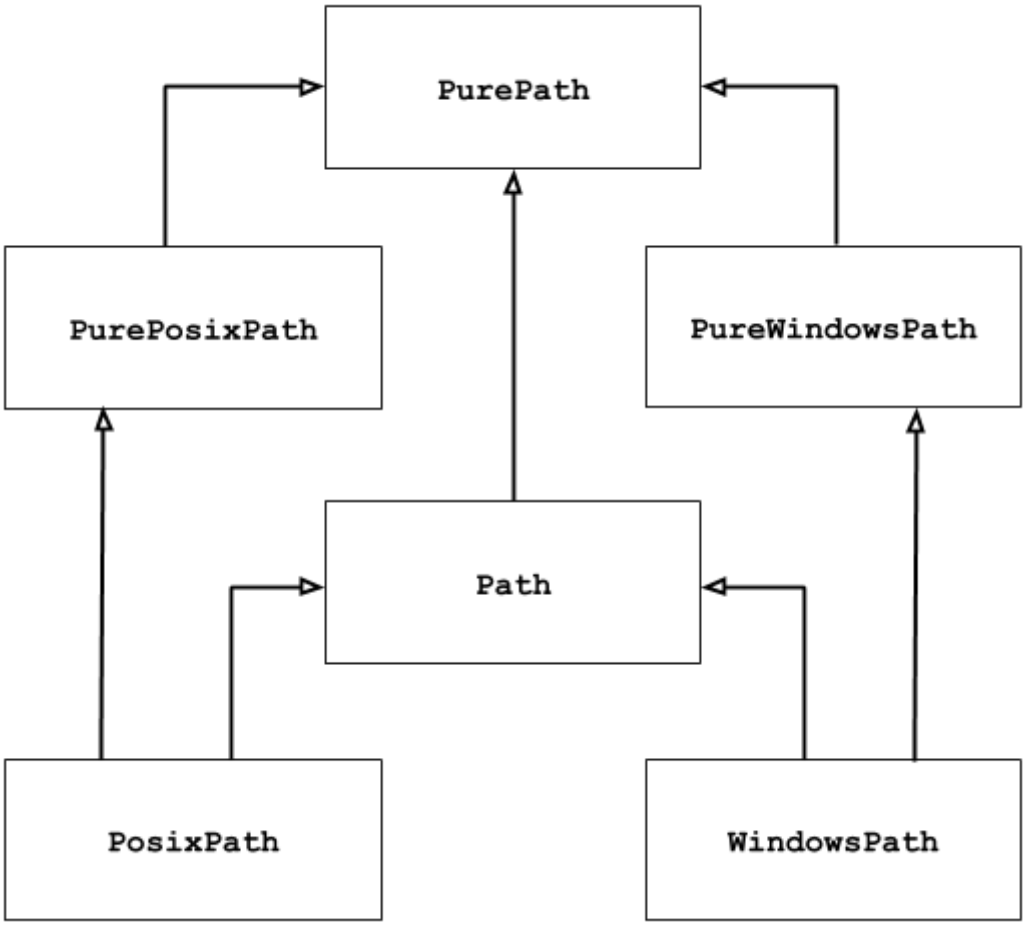
内置功能 open()
使用Python打开文件进行读写的标准方式。

11.1。pathlib-面向对象的文件系统路径

3.4版新增功能

源代码： Lib / pathlib.py

这个模块提供了代表文件系统路径的类，其语义适用于不同的操作系统。路径类在纯路径之间划分，纯路径提供没有I / O的纯粹计算操作，以及具体路径，它们从纯路径继承，但也提供I / O操作。



如果你以前从未使用过这个模块，或者只是不确定哪个类适合你的任务，那么Path很可能是你需要的。它为代码运行的平台实例化一个具体路径。

纯路径在一些特殊情况下很有用; 例如：

1. 如果你想在Unix机器上操作Windows路径（反之亦然）。WindowsPath在Unix上运行时不能实例化，但可以实例化PureWindowsPath。
2. 您希望确保您的代码仅在不实际访问操作系统的情况下操作路径。在这种情况下，实例化其中一个纯类可能很有用，因为这些类没有任何操作系统访问操作。

也可以看看： **PEP 428**：pathlib模块 - 面向对象的文件系统路径。

也可以看看： 对于字符串的低级路径操作，您也可以使用该 os.path模块。

11.1.1。基本使用

导入主类：

```
>>> from pathlib import Path
```

列出子目录：

```
>>> p = Path('.')
>>> [x for x in p.iterdir() if x.is_dir()]
[PosixPath('.hg'), PosixPath('docs'), PosixPath('dist'),
 PosixPath('__pycache__'), PosixPath('build')]
```

在此目录树中列出Python源文件：

```
>>> list(p.glob('**/*.py'))
[PosixPath('test_pathlib.py'), PosixPath('setup.py'),
 PosixPath('pathlib.py'), PosixPath('docs/conf.py'),
 PosixPath('build/lib/pathlib.py')]
```

在目录树内导航：

```
>>> p = Path('/etc')
>>> q = p / 'init.d' / 'reboot'
>>> q
PosixPath('/etc/init.d/reboot')
>>> q.resolve()
PosixPath('/etc/rc.d/init.d/halt')
```

查询路径属性：

```
>>> q.exists()
True
>>> q.is_dir()
False
```

打开文件：

```
>>> with q.open() as f: f.readline()
...
'#!/bin/bash\n'
```

11.1.2。纯路径

纯路径对象提供不实际访问文件系统的路径处理操作。有三种方法可以访问这些类，我们也称之为*flavor*：

class pathlib. PurePath (* *pathsegments*)

表示系统路径风格的泛型类（实例化它将创建一个PurePosixPath或一个PureWindowsPath）：

```
>>> PurePath('setup.py')      # Running on a Unix machine
PurePosixPath('setup.py')
```

*pathsegments*的每个元素可以是表示路径段os.PathLike的字符串，实现返回字符串的接口的对象或其他路径对象：

```
>>> PurePath('foo', 'some/path', 'bar')
PurePosixPath('foo/some/path/bar')
>>> PurePath(Path('foo'), Path('bar'))
PurePosixPath('foo/bar')
```

当*pathsegments*为空时，假定当前目录：

```
>>> PurePath()
PurePosixPath('.')
```

当给出几条绝对路径时，最后一条被当作锚点（模仿os.path.join()行为）：

```
>>> PurePath('/etc', '/usr', 'lib64')
PurePosixPath('/usr/lib64')
>>> PureWindowsPath('c:/Windows', 'd:bar')
PureWindowsPath('d:bar')
```

但是，在Windows路径中，更改本地根目录不会丢弃先前的驱动器设置：

```
>>> PureWindowsPath('c:/Windows', '/Program Files')
PureWindowsPath('c:/Program Files')
```

假斜线和单点被折叠，但双点（'..'）不是，因为这会改变符号链接中路径的含义：

```
>>> PurePath('foo//bar')
PurePosixPath('foo/bar')
>>> PurePath('foo/./bar')
PurePosixPath('foo/bar')
>>> PurePath('foo/../bar')
PurePosixPath('foo/../bar')
```

（一种天真的方法将会PurePosixPath('foo/../bar')等同于PurePosixPath('bar')，如果foo是到另一个目录的符号链接，这是错误的）

纯路径对象实现os.PathLike接口，允许它们在接口被接受的任何地方使用。

*在版本3.6中更改：*增加了对os.PathLike接口的支持。

class pathlib.PurePosixPath(**pathsegments*)

PurePath该路径的一个子类代表非Windows文件系统路径：

```
>>> PurePosixPath('/etc')
PurePosixPath('/etc')
```

*pathsegments*的指定方式与PurePath。

class pathlib.PureWindowsPath (**pathsegments*)

PurePath此路径风格的子类表示Windows文件系统路径：

```
>>> PureWindowsPath('c:/Program Files/')
PureWindowsPath('c:/Program Files')
```

*pathsegments*的指定方式与PurePath。

无论你在哪个系统上运行，你都可以实例化所有这些类，因为它们不提供任何系统调用的操作。

11.1.2.1。一般属性

路径是不可变的，可散列的。相同风味的路径是可比的和可订购的。这些属性尊重风格的case-folding语义：

```
>>> PurePosixPath('foo') == PurePosixPath('FOO')
False
>>> PureWindowsPath('foo') == PureWindowsPath('FOO')
True
>>> PureWindowsPath('FOO') in { PureWindowsPath('foo') }
True
>>> PureWindowsPath('C:') < PureWindowsPath('d:')
True
```

不同风味的路径比较不平等，不能订购：

```
>>> PureWindowsPath('foo') == PurePosixPath('foo')
False
>>> PureWindowsPath('foo') < PurePosixPath('foo')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'PureWindowsPath' and 'PurePosixPatl
```

11.1.2.2。运营商

斜杠运算符有助于创建子路径，类似于os.path.join()：

```
>>> p = PurePath('/etc')
>>> p
PurePosixPath('/etc')
>>> p / 'init.d' / 'apache2'
PurePosixPath('/etc/init.d/apache2')
>>> q = PurePath('bin')
>>> '/usr' / q
PurePosixPath('/usr/bin')
```

路径对象可用于os.PathLike 接受对象实现的任何位置：

```
>>> import os
>>> p = PurePath('/etc')
>>> os.fspath(p)
'/etc'
```

路径的字符串表示形式是原始文件系统路径本身（以本机形式，例如Windows下的反斜杠），您可以将它传递给以文件路径作为字符串的任何函数：

```
>>> p = PurePath('/etc')
>>> str(p)
'/etc'
>>> p = PureWindowsPath('c:/Program Files')
>>> str(p)
'c:\\Program Files'
```

类似地，调用bytes一个路径将原始文件系统路径作为字节对象进行编码，编码方式如下os.fsencode()：

```
>>> bytes(p)
b'/etc'
```

注意: 调用bytes只在Unix下推荐。在Windows下，unicode表单是文件系统路径的标准表示。

11.1.2.3。访问各个部分

要访问路径的单个“零件”（组件），请使用以下属性：

PurePath.**parts**

访问路径各个组件的元组：

```
>>> p = PurePath('/usr/bin/python3')
>>> p.parts
('/', 'usr', 'bin', 'python3')

>>> p = PureWindowsPath('c:/Program Files/PSF')
>>> p.parts
('c:\\', 'Program Files', 'PSF')
```

（注意驱动器和本地根在一个部分中是如何重新组合的）

11.1.2.4。方法和属性

纯路径提供以下方法和属性：

PurePath.**drive**

表示驱动器号或名称的字符串（如果有）：

```
>>> PureWindowsPath('c:/Program Files/').drive
'c:'
>>> PureWindowsPath('/Program Files/').drive
''
>>> PurePosixPath('/etc').drive
''
```

UNC股票也被认为是驱动力：

```
>>> PureWindowsPath('//host/share/foo.txt').drive
'\\\\host\\share'
```

PurePath.**root**

表示（本地或全局）根目录的字符串，如果有的话：

```
>>> PureWindowsPath('c:/Program Files/').root
'\\'
>>> PureWindowsPath('c:Program Files/').root
''
>>> PurePosixPath('/etc').root
'/'
```

UNC股份始终有一个根源：

```
>>> PureWindowsPath('//host/share').root
'\\'
```

PurePath.**anchor**

驱动器和根的连接：

```
>>> PureWindowsPath('c:/Program Files/').anchor
'c:\\'
>>> PureWindowsPath('c:Program Files/').anchor
'c:'
>>> PurePosixPath('/etc').anchor
'/'
>>> PureWindowsPath('//host/share').anchor
'\\\\host\\share\\'
```

PurePath.**parents**

提供对路径的逻辑祖先的访问的不可变序列：

```
>>> p = PureWindowsPath('c:/foo/bar/setup.py')
>>> p.parents[0]
PureWindowsPath('c:/foo/bar')
>>> p.parents[1]
PureWindowsPath('c:/foo')
>>> p.parents[2]
PureWindowsPath('c:/')
```

PurePath.**parent**

路径的逻辑父项：

```
>>> p = PurePosixPath('/a/b/c/d')
>>> p.parent
PurePosixPath('/a/b/c')
```

你不能通过锚点或空路径：

```
>>> p = PurePosixPath('/')
>>> p.parent
PurePosixPath('/')
>>> p = PurePosixPath('.')
>>> p.parent
PurePosixPath('.')
```

注意: 这是纯粹的词汇操作，因此具有以下行为：

```
>>> p = PurePosixPath('foo/..')
>>> p.parent
PurePosixPath('foo')
```

如果你想向上走任意的文件系统路径，建议首先调用`Path.resolve()`以解决符号链接并消除“..”组件。

PurePath.**name**

表示最终路径组件的字符串，不包括驱动器和根（如果有）：

```
>>> PurePosixPath('my/library/setup.py').name
'setup.py'
```

不考虑UNC驱动器名称：

```
>>> PureWindowsPath('//some/share/setup.py').name
'setup.py'
>>> PureWindowsPath('//some/share').name
''
```

PurePath.**suffix**

最终组件的文件扩展名（如果有）：

```
>>> PurePosixPath('my/library/setup.py').suffix
'.py'
>>> PurePosixPath('my/library.tar.gz').suffix
'.gz'
>>> PurePosixPath('my/library').suffix
''
```

PurePath.**suffixes**

路径的文件扩展名列表：

```
>>> PurePosixPath('my/library.tar.gar').suffixes
['.tar', '.gar']
>>> PurePosixPath('my/library.tar.gz').suffixes
```

```
['.tar', '.gz']
>>> PurePosixPath('my/library').suffixes
[]
```

PurePath.**stem**

最终路径组件，没有后缀：

```
>>> PurePosixPath('my/library.tar.gz').stem
'library.tar'
>>> PurePosixPath('my/library.tar').stem
'library'
>>> PurePosixPath('my/library').stem
'library'
```

PurePath.**as_posix ( )**

用正斜杠（/）返回路径的字符串表示形式：

```
>>> p = PureWindowsPath('c:\\windows')
>>> str(p)
'c:\\windows'
>>> p.as_posix()
'c:/windows'
```

PurePath.**as_uri ( )**

将路径表示为file URI。 ValueError如果路径不是绝对的，则引发。

```
>>> p = PurePosixPath('/etc/passwd')
>>> p.as_uri()
'file:///etc/passwd'
>>> p = PureWindowsPath('c:/Windows')
>>> p.as_uri()
'file:///c:/Windows'
```

PurePath.**is_absolute ( )**

返回路径是否是绝对路径。如果一个路径同时拥有一个根目录（如果允许的话）一个驱动器，则该路径被认为是绝对路径：

```
>>> PurePosixPath('/a/b').is_absolute()
True
>>> PurePosixPath('a/b').is_absolute()
False

>>> PureWindowsPath('c:/a/b').is_absolute()
True
>>> PureWindowsPath('/a/b').is_absolute()
False
>>> PureWindowsPath('c:').is_absolute()
False
>>> PureWindowsPath('//some/share').is_absolute()
True
```

PurePath.**is_reserved ( )**

随着PureWindowsPath，返回True如果路径被认为是Windows下的保留，False否则。与PurePosixPath， False总是返回。

```
>>> PureWindowsPath('nul').is_reserved()
True
>>> PurePosixPath('nul').is_reserved()
False
```

保留路径上的文件系统调用可能会以神秘方式失败或产生意想不到的效果。

PurePath.**joinpath**(**other*)

调用此方法相当于将路径依次与*其他*每个参数相结合：

```
>>> PurePosixPath('/etc').joinpath('passwd')
PurePosixPath('/etc/passwd')
>>> PurePosixPath('/etc').joinpath(PurePosixPath('passwd'))
PurePosixPath('/etc/passwd')
>>> PurePosixPath('/etc').joinpath('init.d', 'apache2')
PurePosixPath('/etc/init.d/apache2')
>>> PureWindowsPath('c:').joinpath('/Program Files')
PureWindowsPath('c:/Program Files')
```

PurePath.**match**(*模式*)

将此路径与提供的全局样式匹配。True 如果匹配成功False则返回，否则返回。

如果*模式*是相对的，则路径可以是相对或绝对的，并且从右侧进行匹配：

```
>>> PurePath('a/b.py').match('*.py')
True
>>> PurePath('/a/b/c.py').match('b/*.py')
True
>>> PurePath('/a/b/c.py').match('a/*.py')
False
```

如果*pattern*是绝对路径，则路径必须是绝对路径，并且整个路径必须匹配：

```
>>> PurePath('/a.py').match('/*.py')
True
>>> PurePath('a/b.py').match('/*.py')
False
```

与其他方法一样，可以观察到大小写敏感性：

```
>>> PureWindowsPath('b.py').match('*.PY')
True
```

PurePath.**relative_to**(**other*)

计算相对于*其他*路径所代表的路径的版本 。如果不可能，则会产生ValueError：

```
>>> p = PurePosixPath('/etc/passwd')
>>> p.relative_to('/')
PurePosixPath('etc/passwd')
```

```
>>> p.relative_to('/etc')
PurePosixPath('passwd')
>>> p.relative_to('/usr')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pathlib.py", line 694, in relative_to
    .format(str(self), str(formatted)))
ValueError: '/etc/passwd' does not start with '/usr'
```

PurePath.**with_name**(*名字*)

name改变后返回一个新路径。如果原始路径没有名称，则会引发ValueError：

```
>>> p = PureWindowsPath('c:/Downloads/pathlib.tar.gz')
>>> p.with_name('setup.py')
PureWindowsPath('c:/Downloads/setup.py')
>>> p = PureWindowsPath('c:/')
>>> p.with_name('setup.py')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/antoine/cpython/default/Lib/pathlib.py", line 751, in with_name
    raise ValueError("%r has an empty name" % (self,))
ValueError: PureWindowsPath('c:/') has an empty name
```

PurePath.**with_suffix**(*后缀*)

suffix改变后返回一个新路径。如果原始路径没有后缀，则会附加新的*后缀*：

```
>>> p = PureWindowsPath('c:/Downloads/pathlib.tar.gz')
>>> p.with_suffix('.bz2')
PureWindowsPath('c:/Downloads/pathlib.tar.bz2')
>>> p = PureWindowsPath('README')
>>> p.with_suffix('.txt')
PureWindowsPath('README.txt')
```

11.1.3。具体路径

具体路径是纯路径类的子类。除了由后者提供的操作外，他们还提供了对路径对象执行系统调用的方法。有三种方式来实例化具体路径：

class pathlib.**Path**(**pathsegments*)

PurePath该类的一个子类表示系统路径风格的具体路径（实例化它创建一个 PosixPath或一个WindowsPath）：

```
>>> Path('setup.py')
PosixPath('setup.py')
```

*pathsegments*的指定方式与PurePath。

class pathlib.**PosixPath**(**pathsegments*)

的子类Path和PurePosixPath，这个类表示具体的非Windows文件系统的路径：

```
>>> PosixPath('/etc')
PosixPath('/etc')
```

*pathsegments*的指定方式与PurePath。

class pathlib.**WindowsPath**(**pathsegments*)

的子类Path和PureWindowsPath，这个类代表具体的Windows文件系统的路径：

```
>>> WindowsPath('c:/Program Files/')
WindowsPath('c:/Program Files')
```

*pathsegments*的指定方式与PurePath。

您只能实例化与您的系统相对应的类风格（允许在不兼容的路径风格上进行系统调用可能导致应用程序中的错误或故障）：

```
>>> import os
>>> os.name
'posix'
>>> Path('setup.py')
PosixPath('setup.py')
>>> PosixPath('setup.py')
PosixPath('setup.py')
>>> WindowsPath('setup.py')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pathlib.py", line 798, in __new__
    % (cls.__name__,))
NotImplementedError: cannot instantiate 'WindowsPath' on your system
```

11.1.3.1。方法

除了纯路径方法外，具体路径还提供以下方法。OSError如果系统调用失败（例如因为路径不存在），这些方法中的很多方法都会引发：

classmethod Path.**cwd**( )

返回表示当前目录的新路径对象（由返回的os.getcwd()）：

```
>>> Path.cwd()
PosixPath('/home/antoine/pathlib')
```

classmethod Path.**home**( )

返回代表用户的主目录（如返回的新路径对象os.path.expanduser()与~结构）：

```
>>> Path.home()
PosixPath('/home/antoine')
```

3.5版本中的新功能。

Path.**stat**( )

返回有关此路径的信息（与之类似os.stat()）。每次调用此方法时都会查看结果。

```
>>> p = Path('setup.py')
>>> p.stat().st_size
956
>>> p.stat().st_mtime
1327883547.852554
```

Path.**chmod**(*模式*)

更改文件模式和权限，如os.chmod()：

```
>>> p = Path('setup.py')
>>> p.stat().st_mode
33277
>>> p.chmod(0o444)
>>> p.stat().st_mode
33060
```

Path.**exists**()

路径是指向现有文件还是目录：

```
>>> Path('.').exists()
True
>>> Path('setup.py').exists()
True
>>> Path('/etc').exists()
True
>>> Path('nonexistentfile').exists()
False
```

注意： 如果路径指向符号链接，则exists()返回符号链接是*指向*现有文件还是目录。

Path.**expanduser**()

返回具有展开式~和~user构造式的新路径，返回值为os.path.expanduser()：

```
>>> p = PosixPath('~/films/Monty Python')
>>> p.expanduser()
PosixPath('/home/eric/films/Monty Python')
```

3.5版本中的新功能。

Path.**glob**(*模式*)

在此路径表示的目录中遍历给定的*模式*，产生所有匹配的文件（任何类型）：

```
>>> sorted(Path('.').glob('*.py'))
[PosixPath('pathlib.py'), PosixPath('setup.py'), PosixPath('test_pathlib.py')]
>>> sorted(Path('.').glob('*/*.py'))
[PosixPath('docs/conf.py')]
```

“**”模式意思是“这个目录和所有子目录，递归地”。换句话说，它可以实现递归通配：

```
>>> sorted(Path('.').glob('**/*.py'))
[PosixPath('build/lib/pathlib.py'),
 PosixPath('docs/conf.py'),
 PosixPath('pathlib.py'),
 PosixPath('setup.py'),
 PosixPath('test_pathlib.py')]
```

注意: **在大型目录树中使用“ ”模式可能会消耗大量的时间。

Path.group ( )

返回拥有该文件的组的名称。 KeyError如果在系统数据库中找不到该文件的gid，则会引发该问题。

Path.is_dir ( )

返回True如果路径指向目录（或指向目录的符号链接），False如果它指向另一个类型的文件。

False如果路径不存在或者是符号链接断开，也会返回; 其他错误（例如权限错误）会被传播。

Path.is_file ( )

返回True如果路径指向一个普通文件（或指向一个普通文件的符号链接），False如果它指向另一个类型的文件。

False如果路径不存在或者是符号链接断开，也会返回; 其他错误（例如权限错误）会被传播。

Path.is_symlink ( )

True如果路径指向符号链接，False则返回，否则返回。

False如果路径不存在也会返回; 其他错误（例如权限错误）会被传播。

Path.is_socket ( )

返回True如果路径指向Unix套接字（或指向Unix套接字的符号链接），False如果它指向另一个类型的文件。

False如果路径不存在或者是符号链接断开，也会返回; 其他错误（例如权限错误）会被传播。

Path.is_fifo ( )

返回True如果路径指向FIFO（或指向FIFO的符号链接），False如果它指向另一个类型的文件。

False如果路径不存在或者是符号链接断开，也会返回; 其他错误（例如权限错误）会被传播。

Path.is_block_device ( )

返回True如果路径指向一个块设备（或指向块设备的符号链接），False如果它指向另一类型的文件。

False如果路径不存在或者是符号链接断开，也会返回; 其他错误（例如权限错误）会被传播。

Path.**is_char_device**()

返回True如果路径指向一个字符设备（或指向一个字符设备的符号链接），False如果它指向另一个类型的文件。

False如果路径不存在或者是符号链接断开，也会返回; 其他错误（例如权限错误）会被传播。

Path.**iterdir**()

当路径指向一个目录时，产生目录内容的路径对象：

```
>>> p = Path('docs')
>>> for child in p.iterdir(): child
...
PosixPath('docs/conf.py')
PosixPath('docs/_templates')
PosixPath('docs/make.bat')
PosixPath('docs/index.rst')
PosixPath('docs/_build')
PosixPath('docs/_static')
PosixPath('docs/Makefile')
```

Path.**lchmod**(*模式*)

就像Path.chmod()但是，如果路径指向一个符号链接，符号链接的模式会改变，而不是其目标。

Path.**lstat**()

像Path.stat()但是，如果路径指向符号链接，则返回符号链接的信息而不是其目标。

Path.**mkdir**(*mode = 0o777*, *parents = False*, *exist_ok = False*)

在给定路径中创建一个新目录。如果给出了*模式*，它将与进程的umask值结合以确定文件模式和访问标志。如果路径已经存在，FileExistsError 则引发。

如果*父母*是真实的，那么这条道路上失踪的父母会根据需要创建; 它们是使用默认权限创建的，无需考虑 *模式*（模仿POSIX 命令）。mkdir -p

如果*父母*是虚假的（默认），失踪的父母提出 FileNotFoundError。

如果*exist_ok*为假（缺省值），FileExistsError则在目标目录已存在时引发。

如果*exist_ok*为true，那么FileExistsError异常将被忽略（与POSIX 命令行为相同），但*前提*是最后一个路径组件不是现有的非目录文件。mkdir -p

*在3.5版本中更改：*将*exist_ok*加入参数。

Path.**open**(*mode ='r'*, *buffering = -1*, *encoding = None*, *errors = None*, *newline = None*)

打开路径指向的文件，就像内置open() 函数一样：

```
>>> p = Path('setup.py')
>>> with p.open() as f:
...     f.readline()
...
'#!/usr/bin/env python3\n'
```

Path.**owner**( )

返回拥有该文件的用户的名称。 KeyError如果在系统数据库中找不到该文件的uid，则会引发该问题。

Path.**read_bytes**( )

以字节对象的形式返回指向文件的二进制内容：

```
>>> p = Path('my_binary_file')
>>> p.write_bytes(b'Binary file contents')
20
>>> p.read_bytes()
b'Binary file contents'
```

3.5版本中的新功能。

Path.**read_text**(*encoding = None*，*errors = None*)

以字符串形式返回指向文件的解码内容：

```
>>> p = Path('my_text_file')
>>> p.write_text('Text file contents')
18
>>> p.read_text()
'Text file contents'
```

可选参数与in中的含义相同open()。

3.5版本中的新功能。

Path.**rename**(*目标*)

将该文件或目录重命名为给定的*目标*。在Unix上，如果 *目标*存在并且是一个文件，则在用户有权限时将被替换为无提示。 *目标*可以是字符串或其他路径对象：

```
>>> p = Path('foo')
>>> p.open('w').write('some text')
9
>>> target = Path('bar')
>>> p.rename(target)
>>> target.open().read()
'some text'
```

Path.**replace**(*目标*)

将该文件或目录重命名为给定的*目标*。如果*目标*指向一个现有的文件或目录，它将被无条件地替换。

Path.**resolve**(*strict = False*)

使路径绝对，解决任何符号链接。返回一个新的路径对象：

```
>>> p = Path()
>>> p
PosixPath('.')
>>> p.resolve()
PosixPath('/home/antoine/pathlib')
```

“..”组件也被淘汰（这是唯一的方法）：

```
>>> p = Path('docs/../setup.py')
>>> p.resolve()
PosixPath('/home/antoine/pathlib/setup.py')
```

如果路径不存在，并*严格*为True，FileNotFoundError 提高。如果*严格的*是False，路径尽可能解决任何其余没有检查它是否存在追加。如果沿分辨率路径遇到无限循环，RuntimeError 则会引发。

*新的3.6版：在严格的*论证。

Path.**rglob**(*模式*)

这就像是在给定*模式*前添加Path.glob()“**”：

```
>>> sorted(Path().rglob("*.py"))
[PosixPath('build/lib/pathlib.py'),
 PosixPath('docs/conf.py'),
 PosixPath('pathlib.py'),
 PosixPath('setup.py'),
 PosixPath('test_pathlib.py')]
```

Path.**rmdir**()

删除这个目录。该目录必须为空。

Path.**samefile**(*other_path*)

返回此路径是否指向与*other_path*相同的文件，其可以是Path对象或字符串。语义与os.path.samefile()和类似os.path.samestat()。

一OSError，如果两个文件不能因为某些原因被访问可以提高。

```
>>> p = Path('spam')
>>> q = Path('eggs')
>>> p.samefile(q)
False
>>> p.samefile('spam')
True
```

3.5版本中的新功能。

Path.**symlink_to**(*target*, *target_is_directory = False*)

将此路径作为*目标*的符号链接。在Windows下，如果链接的目标是目录，则*target_is_directory*必须为true（默认False）。在POSIX下，*target_is_directory*的值被忽

略。

```
>>> p = Path('mylink')
>>> p.symlink_to('setup.py')
>>> p.resolve()
PosixPath('/home/antoine/pathlib/setup.py')
>>> p.stat().st_size
956
>>> p.lstat().st_size
8
```

注意： 参数的顺序（链接，目标）与os.symlink()s 的相反。

Path.**touch**(*mode = 0o666*, *exist_ok = True*)

在给定的路径上创建一个文件。如果给出了*模式*，它将与进程的umask值结合以确定文件模式和访问标志。如果文件已经存在，则函数成功，如果*exist_ok* 为true（并且其修改时间更新为当前时间），否则FileExistsError将引发该函数。

Path.**unlink**( )

删除此文件或符号链接。如果路径指向一个目录，则Path.rmdir()改为使用。

Path.**write_bytes**(*数据*)

打开以字节模式指向的文件，向其写入*数据*并关闭文件：

```
>>> p = Path('my_binary_file')
>>> p.write_bytes(b'Binary file contents')
20
>>> p.read_bytes()
b'Binary file contents'
```

现有的同名文件被覆盖。

3.5版本中的新功能。

Path.**write_text**(*data*, *encoding = None*, *errors = None*)

打开文本模式中指向的文件，向其写入*数据*并关闭文件：

```
>>> p = Path('my_text_file')
>>> p.write_text('Text file contents')
18
>>> p.read_text()
'Text file contents'
```

3.5版本中的新功能。

11.2。 os.path- 通用路径名操作

源代码： Lib / posixpath.py（用于POSIX），Lib / ntpath.py（用于Windows NT）和Lib / macpath.py（用于Macintosh）

该模块在路径名上实现了一些有用的功能。要读取或写入文件open()，请参阅os模块，以及访问文件系统 。路径参数可以作为字符串或字节传递。鼓励应用程序将文件名称表示为（Unicode）字符串。不幸的是，某些文件名在Unix上可能不能表示为字符串，因此需要在Unix上支持任意文件名的应用程序应该使用字节对象来表示路径名。反之亦然，使用字节对象不能代表Windows上的所有文件名（使用标准mbcs编码），因此Windows应用程序应该使用字符串对象来访问所有文件。

与unix shell不同，Python不会执行任何*自动*路径扩展。当应用程序需要类似shell的路径扩展时expanduser()，expandvars()可以显式调用诸如和的函数。（另见glob模块。）

也可以看看： 该pathlib模块提供高级路径对象。

注意： 所有这些函数都只接受字节或只接受字符串对象作为它们的参数。如果返回路径或文件名，结果是相同类型的对象。

注意： 由于不同的操作系统具有不同的路径名称约定，因此标准库中有该模块的几个版本。该 os.path模块始终是适用于Python运行的操作系统的路径模块，因此可用于本地路径。但是，如果要操纵*始终*采用不同格式之一的路径，还可以导入和使用各个模块。它们都具有相同的界面：

- posixpath 对于UNIX风格的路径
- ntpath 为Windows路径
- macpath 适用于旧式MacOS路径

os.path.**abspath**(*路径*)

返回路径名*路径*的归一化绝对版本。在大多数平台上，这相当于normpath()如下调用函数：。normpath(join(os.getcwd(), path))

*在版本3.6中更改：*接受类似路径的对象。

os.path.**basename**(*路径*)

返回路径名*路径*的基本名称。这是通过传递函数*路径*返回的对中的第二个元素split()。请注意，该函数的结果与Unix **基本**程序不同，其中**基名**为'/foo/bar/'回报'bar'，该basename()函数返回一个空字符串（''）。

*在版本3.6中更改：*接受类似路径的对象。

os.path.**commonpath**(*路径*)

返回序列*路径*中每个路径名的最长公共子 *路径*。如果*路径*包含绝对路径名和相对路径名，或*路径*为空，则引发ValueError 。不同的是commonprefix()，这返回一个有效的路径。

可用性：Unix，Windows

3.5版本中的新功能。

*在版本3.6中更改：*接受一系列路径类对象。

os.path.**commonprefix**(*列表*)

返回*列表*中所有路径前缀的最长路径前缀（逐个字符）。如果*列表*为空，则返回空字符串（''）。

> **注意：** 此函数可能会返回无效路径，因为它一次处理一个字符。要获得有效的路径，请参阅 commonpath()。
>
> ```
> >>> os.path.commonprefix(['/usr/lib', '/usr/local/lib'])
> '/usr/l'
>
> >>> os.path.commonpath(['/usr/lib', '/usr/local/lib'])
> '/usr'
> ```

*在版本3.6中更改：*接受类似路径的对象。

os.path.**dirname**(*路径*)

返回路径名*路径*的目录名称。这是通过传递函数*路径*返回的第一个元素split()。

*在版本3.6中更改：*接受类似路径的对象。

os.path.**exists**(*路径*)

返回True如果*路径*是指现有的路径或一个打开的文件描述符。返回False已损坏的符号链接。在某些平台上，False如果未授予os.stat()对所请求的文件执行权限（即使*路径*实际存在），此函数可能会返回。

*在版本3.3中更改：路径*现在可以是一个整数：True如果它是一个打开的文件描述符，则返回; False否则返回。

*在版本3.6中更改：*接受类似路径的对象。

os.path.**lexists**(*路径*)

True如果*路径*指向现有路径，则返回。返回True已损坏的符号链接。相当于exists()缺乏平台 os.lstat()。

*在版本3.6中更改：*接受类似路径的对象。

os.path.**expanduser**(*路径*)

在Unix和Windows上，返回该参数的初始组件~或 ~user由该*用户*的主目录替换。

在Unix上，初始~值由环境变量替换HOME 如果它被设置; 否则通过内置模块在密码目录中查找当前用户的主目录pwd。初始~user 值直接在密码目录中查找。

在Windows上， HOME 和 USERPROFILE 将被使用，如果设置，否则组合 HOMEPATH 和 HOMEDRIVE将会被使用。 ~user通过剥离上面创建的用户路径中的最后一个目录组件来处理

初始化。

如果展开失败或路径不以波形开始，则路径将保持不变。

*在版本3.6中更改：*接受类似路径的对象。

os.path.**expandvars**(*路径*)

返回扩展了环境变量的参数。子表格的子字符串 $name或被${name}环境变量*名称*的值替换。格式错误的变量名称和对不存在的变量的引用保持不变。

在Windows上，%name%除了$name和 ，还支持扩展${name}。

*在版本3.6中更改：*接受类似路径的对象。

os.path.**getatime**(*路径*)

返回上次访问*路径*的时间。返回值是一个给出历元以来秒数的数字（参见 time模块）。提高 OSError如果文件不存在或无法访问。

如果os.stat_float_times()返回True，则结果是一个浮点数。

os.path.**getmtime**(*路径*)

返回最后修改*路径*的时间。返回值是一个给出历元以来秒数的数字（参见 time模块）。提高OSError如果文件不存在或无法访问。

如果os.stat_float_times()返回True，则结果是一个浮点数。

*在版本3.6中更改：*接受类似路径的对象。

os.path.**getctime**(*路径*)

返回系统的ctime，在某些系统上（如Unix）是最后一次元数据更改的时间，而在另一些系统（如Windows）上则是创建*路径*的时间。返回值是一个给出历元以来秒数的数字（参见time模块）。提高OSError如果文件不存在或无法访问。

*在版本3.6中更改：*接受类似路径的对象。

os.path.**getsize**(*路径*)

返回*路径*的大小（以字节为单位）。提高OSError如果文件不存在或无法访问。

*在版本3.6中更改：*接受类似路径的对象。

os.path.**isabs**(*路径*)

True如果*路径*是绝对路径名，则返回。在Unix上，这意味着它以斜线开始，在Windows上，它在切断潜在驱动器盘符后以（后退）斜杠开始。

*在版本3.6中更改：*接受类似路径的对象。

os.path.**isfile**(*路径*)

True如果*路径*是existing常规文件，则返回。这是继符号链接，这样既islink()并且isfile()可以为相同的路径是正确的。

*在版本3.6中更改：*接受类似路径的对象。

os.path.**isdir**(*路径*)

True如果*路径*是existing目录，则返回。这是继符号链接，这样既islink()并且isdir()可以为相同的路径是正确的。

*在版本3.6中更改：*接受类似路径的对象。

os.path.**islink**(*路径*)

True如果*路径*指向existing一个符号链接的目录条目，则返回。总是False如果Python运行时不支持符号链接。

*在版本3.6中更改：*接受类似路径的对象。

os.path.**ismount**(*路径*)

True如果路径名*路径*是*装载点*，则返回：*装入*不同文件系统的文件系统中的一个点。在POSIX上，函数检查*路径*的父节点是否path/..位于与*路径*不同的设备上，path/..以及*路径*是否指向同一设备上的同一个节点 - 这应该检测所有Unix和POSIX变体的安装点。在Windows上，驱动器盘符和共享UNC始终是挂载点，并且GetVolumePathName调用任何其他路径以查看它是否与输入路径不同。

*3.4版新增功能：*支持检测Windows上的非根挂载点。

*在版本3.6中更改：*接受类似路径的对象。

os.path.**join**(*路径*, **路径*)

智能地加入一个或多个路径组件。返回值是*路径*和**路径的*任何成员的连接，os.sep每个除了最后一个非空部分之后，都有一个目录分隔符（ ），这意味着如果最后一部分为空，结果将仅在分隔符中结束。如果某个组件是绝对路径，则所有先前的组件都将被丢弃，并从绝对路径组件继续加入。

在Windows上，当遇到绝对路径组件（例如，r'\foo'）时，驱动器号不会重置。如果组件包含驱动器号，则以前的所有组件都将被丢弃，并重置驱动器号。请注意，由于每个驱动器都有当前目录，因此 表示相对于drive （ ）上当前目录的路径，而不是。os.path.join("c:", "foo")C:c:fooc:\foo

*改变在3.6版：*接受一个路径状物体为*路径*和*路径*。

os.path.**normcase**(*路径*)

规范化路径名的情况。在Unix和Mac OS X上，这将返回路径不变; 在不区分大小写的文件系统上，它将路径转换为小写。在Windows上，它也将正斜杠转换为反斜杠。如果*路径*类型不是str或bytes（直接或间接通过os.PathLike接口）引发TypeError 。

*在版本3.6中更改：*接受类似路径的对象。

os.path.**normpath**(*路径*)

通过折叠冗余分离器和上一级引用正常化的路径名，这样A//B，A/B/，A/./B和A/foo/../B一切变得A/B。此字符串操作可能会更改包含符号链接的路径的含义。在Windows上，它将正斜杠转换为反斜杠。要标准化案例，请使用normcase()。

*在版本3.6中更改：*接受类似路径的对象。

os.path.**realpath**(*路径*)

返回指定文件名的规范路径，消除路径中遇到的任何符号链接（如果它们受操作系统支持）。

*在版本3.6中更改：*接受类似路径的对象。

os.path.**relpath**(*path*, *start = os.curdir*)

从当前目录或从可选的*开始*目录中将相关文件路径返回到*路径*。这是一个路径计算：不访问文件系统来确认*路径*或*开始*的存在或性质。

*开始*默认为os.curdir。

可用性：Unix，Windows。

*在版本3.6中更改：*接受类似路径的对象。

os.path.**samefile**(*path1*, *path2*)

返回True如果两个路径参数指的是同一个文件或目录。这由设备号和i-node号确定，如果os.stat()任一路径名上的呼叫失败，则引发异常。

可用性：Unix，Windows。

*在版本3.2中更改：*添加了Windows支持。

在3.4版中更改： Windows现在使用与所有其他平台相同的实现。

*在版本3.6中更改：*接受类似路径的对象。

os.path.**sameopenfile**(*fp1*, *fp2*)

返回True如果文件描述符*FP1*和*FP2*指的是同一个文件。

可用性：Unix，Windows。

*在版本3.2中更改：*添加了Windows支持。

*在版本3.6中更改：*接受类似路径的对象。

os.path.**samestat**(*stat1*, *stat2*)

返回True如果统计的元组*STAT1*和*STAT2*指的是同一个文件。可能已经返回这些结构的os.fstat()，os.lstat()或os.stat()。这个函数实现samefile()和使用的底层比较sameopenfile()。

可用性：Unix，Windows。

*在版本3.4中进行了更改：*添加了Windows支持。

*在版本3.6中更改：*接受类似路径的对象。

os.path.**split**(*路径*)

将路径名*路径*拆分为一对，其中*tail*是最后一个路径名组件，*head*是导致该路径名的所有内容。在 *尾部*部分永远不会包含一个斜杠; 如果*路径*以斜线结尾，则*尾部* 将为空。如果*路径中*没有斜线，则*头部*将是空的。如果 *路径*为空，则*头部*和*尾部*都是空的。除非是根部（仅限一个或多个斜线），否则尾部的斜线会从*头部*剥离。在所有情况下，返回到路径相同位置的*路径*(head, tail) join(head, tail) （但字符串可能不同）。另请参阅函数dirname()和 basename()。

*在版本3.6中更改：*接受类似路径的对象。

os.path. splitdrive（*路径*）

将路径名*路径*拆分为一对，其中*驱动器*是装入点或空字符串。在不使用驱动器规格的系统上，*驱动器*将始终为空字符串。在所有情况下，都将与*路径*相同。(drive, tail) drive + tail

在Windows上，将路径名分割为驱动器/ UNC共享点和相对路径。

如果路径包含驱动器号，则驱动器将包含直至并包括冒号的所有内容。例如splitdrive("c:/dir")退货("c:", "/dir")

如果路径包含UNC路径，则驱动器将包含主机名和共享，直至但不包括第四个分隔符。例如splitdrive("//host/computer/dir")退货("//host/computer", "/dir")

*在版本3.6中更改：*接受类似路径的对象。

os.path. splitext（*路径*）

分裂路径名*路径*成一对 ，使得，和*分机*是空的或用一个周期开始，并且包含至多一个周期。基准名称上的主要时段将被忽略；返回 。(root, ext) root + ext == path splitext('.cshrc') ('.cshrc', '')

*在版本3.6中更改：*接受类似路径的对象。

os.path. splitunc（*路径*）

*自3.1版弃用：*改为使用*splitdrive*。

将路径名*路径*拆分为一对，以便*unc*是UNC挂载点（如）（如果存在），并*保留*路径的其余部分（如 ）。对于包含驱动器号的路径， *unc*将始终为空字符串。(unc, rest) r'\\host\mount' r'\path\file.ext'

可用性：Windows。

os.path. supports_unicode_filenames

True 如果任意Unicode字符串可以用作文件名（在由文件系统施加的限制内）。

11.3。 fileinput- 迭代来自多个输入流的行

源代码： Lib / fileinput.py

该模块实现了一个辅助类和函数，可以在标准输入或文件列表上快速编写循环。如果您只想读取或写入一个文件，请参阅open()。

典型的用途是：

```
import fileinput
for line in fileinput.input():
    process(line)
```

这遍历了列出的所有文件的行sys.argv[1:]，默认sys.stdin列表为空。如果文件名是'-'，它也被替换sys.stdin。要指定替代的文件名列表，请将其作为第一个参数传递给input()。单个文件名也是允许的。

所有的文件都在文本模式下，默认打开的，但你可以通过指定覆盖此*模式*在调用参数input()或FileInput。如果在打开或读取文件期间发生I / O错误， OSError则会引发。

在版本3.3中改变： IOError曾经被提出; 它现在是一个别名OSError。

如果sys.stdin多次使用，第二次和以后的使用将不会返回任何行，除了可能用于交互式使用，或者它已被明确重置（例如使用sys.stdin.seek(0)）之外。

空文件打开并立即关闭; 他们出现在文件名列表中的唯一时刻是显而易见的，那就是最后打开的文件是空的。

行与任何换行符一并返回，这意味着文件的最后一行可能没有。

您可以控制文件是由通过设置开口钩打开 *openhook*参数fileinput.input()或FileInput()。钩子必须是一个函数，它接受两个参数，即*文件名*和*模式*，并返回相应打开的类文件对象。这个模块已经提供了两个有用的钩子。

以下功能是该模块的主要界面：

fileinput.input（*files = None*，*inplace = False*，*backup =''*，*bufsize = 0*，*mode ='r'*，*openhook = None* ）

创建FileInput类的一个实例。该实例将用作此模块功能的全局状态，并在迭代期间返回使用。这个函数的参数将被传递给FileInput类的构造函数。

该FileInput实例可以用作with语句中的上下文管理器 。在此示例中，即使发生异常，语句退出后*输入*也会关闭 with ：

```
with fileinput.input(files=('spam.txt', 'eggs.txt')) as f:
    for line in f:
        process(line)
```

*在版本3.2中更改：*可以用作上下文管理器。

：自3.6版本弃用，将在3.8版本中移除的BUFSIZE参数。

以下功能使用创建的全局状态fileinput.input()；如果没有活动状态，RuntimeError则提出。

fileinput.**filename**( )

返回当前正在读取的文件的名称。在阅读第一行之前，返回None。

fileinput.**fileno**( )

返回当前文件的整数“文件描述符”。当没有文件打开时（在第一行之前和文件之间），返回-1。

fileinput.**lineno**( )

返回刚刚读取的行的累计行号。在阅读第一行之前，返回0。读完最后一个文件的最后一行后，返回该行的行号。

fileinput.**filelineno**( )

返回当前文件中的行号。在阅读第一行之前，返回0。读完最后一个文件的最后一行后，返回文件中该行的行号。

fileinput.**isfirstline**( )

如果刚刚读取的行是其文件的第一行，则返回true，否则返回false。

fileinput.**isstdin**( )

如果读取最后一行sys.stdin，则返回true，否则返回false。

fileinput.**nextfile**( )

关闭当前文件，以便下一次迭代将读取下一个文件的第一行（如果有的话）；不从文件中读取的行数不会计入累计行数。只有在读取下一个文件的第一行之后，文件名才会更改。在阅读第一行之前，此功能无效；它不能用于跳过第一个文件。读完最后一个文件的最后一行后，此功能不起作用。

fileinput.**close**( )

关闭序列。

实现模块提供的序列行为的类也可用于子类化：

class fileinput.**FileInput**(*files = None*，*inplace = False*，*backup =''*，*bufsize = 0*，*mode ='r'*，*openhook = None*)

类FileInput是实现；它的方法filename()，fileno()，lineno()，filelineno()，isfirstline()，isstdin()，nextfile()和close()对应于所述模块中的相同名称的功能。另外它有一个readline()返回下一个输入行的__getitem__()方法和一个实现序列行为的方法。序列必须严格按顺序访问；随机访问并readline()不能混合使用。

使用*模式*，您可以指定将传递给哪个文件模式open()。它必须是一个'r'，'rU'，'U'和'rb'。

的*openhook*，当给出时，必须是一个函数，它有两个参数， *文件名*和*模式*，并返回相应的打开的文件对象。您不能使用*就地*和*openhook*在一起。

一个FileInput实例可以用作with语句中的上下文管理器 。在此示例中，即使发生异常，语句退出后*输入*也会关闭 with ：

```
with FileInput(files=('spam.txt', 'eggs.txt')) as input:
    process(input)
```

*在版本3.2中更改：*可以用作上下文管理器。

*自从3.4版本不推荐使用：*在'rU'和'U'模式。

*：自3.6版本弃用，将在3.8版本中移除*的*BUFSIZE*参数。

可选就地过滤：如果将关键字参数inplace=True传递给构造函数fileinput.input()或将其传递给FileInput构造函数，则文件将移至备份文件，并将标准输出定向至输入文件（如果与备份文件具有相同名称的文件已存在，它会被静静地取代）。这使得可以编写一个过滤器来重写其输入文件。如果给出*备份*参数（通常为 ），则它指定备份文件的扩展名，并且备份文件保留在附近; 默认情况下，扩展名是当输出文件关闭时被删除。当读取标准输入时，就地过滤被禁用。backup='.<some extension>''.bak'

以下两个打开挂钩由该模块提供：

fileinput.hook_compressed（*文件名*，*模式*）

透明地打开使用gzip和bzip2压缩的文件（由扩展程序识别）'.gz'和'.bz2'使用gzip和bz2模块。如果文件扩展名不是'.gz'或'.bz2'，文件是正常打开的（即，open()没有任何解压缩）。

用法示例： fi = fileinput.FileInput(openhook=fileinput.hook_compressed)

fileinput.hook_encoded（*encoding*，*errors = None* ）

返回一个打开每个文件的钩子open()，使用给定的 *编码*和*错误*来读取文件。

用法示例： fi = fileinput.FileInput(openhook=fileinput.hook_encoded("utf-8", "surrogateescape"))

*在版本3.6中更改：*添加了可选*错误*参数。

11.4。 stat- 解释stat()结果

源代码： Lib / stat.py

该stat模块定义用于解释结果的常数和函数os.stat()，os.fstat()以及os.lstat()（如果它们存在）。有关完整的详细信息stat()，fstat()并 lstat()呼叫，请咨询您的系统的文档。

在版本3.4中更改： stat模块由C实现支持。

该stat模块定义了以下功能来测试特定的文件类型：

stat.S_ISDIR（*模式*）
    如果模式来自目录，则返回非零值。

stat.S_ISCHR（*模式*）
    如果模式来自字符特殊设备文件，则返回非零值。

stat.S_ISBLK（*模式*）
    如果模式来自块特殊设备文件，则返回非零值。

stat.S_ISREG（*模式*）
    如果模式来自常规文件，则返回非零值。

stat.S_ISFIFO（*模式*）
    如果模式来自FIFO（命名管道），则返回非零值。

stat.S_ISLNK（*模式*）
    如果模式来自符号链接，则返回非零值。

stat.S_ISSOCK（*模式*）
    如果模式来自套接字，则返回非零值。

stat.S_ISDOOR（*模式*）
    如果模式来自门，则返回非零值。

    3.4版新增功能

stat.S_ISPORT（*模式*）
    如果模式来自事件端口，则返回非零值。

    3.4版新增功能

stat.S_ISWHT（*模式*）
    如果模式来自白板，则返回非零值。

    3.4版新增功能

为更一般的文件模式操作定义了两个额外的功能：

stat.S_IMODE(*模式*)
返回可由os.chmod()文件的权限位，粘滞位，set-group-id和set-user-id位（在支持它们的系统上）设置的文件模式部分。

stat.S_IFMT(*模式*)
返回描述文件类型的文件模式部分（由上述S_IS*()函数使用）。

通常，您可以使用这些os.path.is*()函数来测试文件的类型; 当您对同一文件进行多个测试并希望避免stat()每次测试的系统调用开销时，此处的函数都很有用。当检查有关未处理的文件的信息时，这些也很有用os.path，例如对块和字符设备的测试。

例：

```
import os, sys
from stat import *

def walktree(top, callback):
    '''recursively descend the directory tree rooted at top,
       calling the callback function for each regular file'''

    for f in os.listdir(top):
        pathname = os.path.join(top, f)
        mode = os.stat(pathname).st_mode
        if S_ISDIR(mode):
            # It's a directory, recurse into it
            walktree(pathname, callback)
        elif S_ISREG(mode):
            # It's a file, call the callback function
            callback(pathname)
        else:
            # Unknown file type, print a message
            print('Skipping %s' % pathname)

def visitfile(file):
    print('visiting', file)

if __name__ == '__main__':
    walktree(sys.argv[1], visitfile)
```

提供了一个额外的实用程序功能，用于以可读的字符串转换文件的模式：

stat.filemode(*模式*)
将文件的模式转换为'-rwxrwxrwx'格式的字符串。

3.3版本的新功能

*在版本3.4中更改：*该功能支持S_IFDOOR，S_IFPORT和S_IFWHT。

下面所有的变量都只是象征性的指标到返回的10元组os.stat()，os.fstat()或os.lstat()。

stat.ST_MODE

Inode保护模式。

stat.ST_INO
Inode号码。

stat.ST_DEV
设备inode驻留在上。

stat.ST_NLINK
到inode的链接数。

stat.ST_UID
所有者的用户标识。

stat.ST_GID
所有者的组ID。

stat.ST_SIZE
纯文件的字节大小; 等待某些特殊文件的数据量。

stat.ST_ATIME
上次访问的时间。

stat.ST_MTIME
上次修改时间。

stat.ST_CTIME
操作系统报告的“ctime”。在某些系统上（如Unix）是最后一次元数据更改的时间，而在另一些系统上（如Windows）则是创建时间（有关详细信息，请参阅平台文档）。

“文件大小”的解释根据文件类型而改变。对于普通文件，这是文件的大小（以字节为单位）。对于在Unix（包括Linux尤其是）的最风味FIFO和插座，“大小”是在等待调用时要读取的字节数os.stat()， os.fstat()或os.lstat()；这有时会很有用，特别是在非阻塞打开后轮询其中一个特殊文件。根据底层系统调用的实现，其他字符和块设备的大小字段的含义各不相同。

下面的变量定义了该ST_MODE字段中使用的标志。

使用上述函数比使用第一组标志更具可移植性：

stat.S_IFSOCK
插座。

stat.S_IFLNK
符号链接。

stat.S_IFREG
普通文件。

stat.S_IFBLK

阻止设备。

stat.S_IFDIR
目录。

stat.S_IFCHR
字符设备。

stat.S_IFIFO
FIFO。

stat.S_IFDOOR
门。

3.4版新增功能

stat.S_IFPORT
事件端口。

3.4版新增功能

stat.S_IFWHT
白化。

3.4版新增功能

注意： S_IFDOOR，S_IFPORT或者S_IFWHT当平台不支持文件类型时将其定义为0。

以下标志也可用于*模式*参数os.chmod()：

stat.S_ISUID
设置UID位。

stat.S_ISGID
设置组ID位。这个位有几个特殊用途。对于一个目录，它表示BSD语义将被用于该目录：其中创建的文件从目录中继承它们的组ID，而不是从创建过程的有效组ID中继承它们，并且在那里创建的目录也将获得该S_ISGID位设置。对于没有设置组执行位（S_IXGRP）的文件，set-group-ID位指示强制文件/记录锁定（另请参见S_ENFMT）。

stat.S_ISVTX
粘滞的位。在目录上设置此位时，表示该目录中的文件只能由文件所有者，目录所有者或特权进程重命名或删除。

stat.S_IRWXU
文件所有者权限的掩码。

stat.S_IRUSR
所有者已阅读权限。

stat.S_IWUSR
所有者拥有写入权限。

stat.S_IXUSR
所有者拥有执行权限。

stat.S_IRWXG
组权限掩码。

stat.S_IRGRP
组具有读取权限。

stat.S_IWGRP
组有写入权限。

stat.S_IXGRP
组拥有执行权限。

stat.S_IRWXO
面向其他人的权限（不在组中）。

stat.S_IROTH
其他人已阅读权限。

stat.S_IWOTH
其他人有写权限。

stat.S_IXOTH
其他人有执行许可。

stat.S_ENFMT
系统V文件锁定实施。该标志与S_ISGID以下内容共享：对没有设置组执行位（S_IXGRP）的文件执行文件/记录锁定。

stat.S_IREAD
Unix V7的代名词S_IRUSR。

stat.S_IWRITE
Unix V7的代名词S_IWUSR。

stat.S_IEXEC
Unix V7的代名词S_IXUSR。

以下标志可用于*flags*参数中os.chflags()：

stat.UF_NODUMP
不要转储文件。

stat.UF_IMMUTABLE

该文件可能不会更改。

stat.UF_APPEND
该文件只能附加到。

stat.UF_OPAQUE
通过联合堆栈查看时，该目录是不透明的。

stat.UF_NOUNLINK
该文件可能不会被重命名或删除。

stat.UF_COMPRESSED
该文件被压缩存储（Mac OS X 10.6+）。

stat.UF_HIDDEN
该文件不应该显示在GUI（Mac OS X 10.5+）中。

stat.SF_ARCHIVED
该文件可能已存档。

stat.SF_IMMUTABLE
该文件可能不会更改。

stat.SF_APPEND
该文件只能附加到。

stat.SF_NOUNLINK
该文件可能不会被重命名或删除。

stat.SF_SNAPSHOT
该文件是一个快照文件。

有关更多信息，请参阅* BSD或Mac OS系统手册页*chflags（2）*。

在Windows上，当测试`st_file_attributes`返回的成员中的位时，可使用以下文件属性常量`os.stat()`。有关 这些常量的含义的更多详细信息，请参阅Windows API文档。

stat.FILE_ATTRIBUTE_ARCHIVE
stat.FILE_ATTRIBUTE_COMPRESSED
stat.FILE_ATTRIBUTE_DEVICE
stat.FILE_ATTRIBUTE_DIRECTORY
stat.FILE_ATTRIBUTE_ENCRYPTED
stat.FILE_ATTRIBUTE_HIDDEN
stat.FILE_ATTRIBUTE_INTEGRITY_STREAM
stat.FILE_ATTRIBUTE_NORMAL
stat.FILE_ATTRIBUTE_NOT_CONTENT_INDEXED
stat.FILE_ATTRIBUTE_NO_SCRUB_DATA
stat.FILE_ATTRIBUTE_OFFLINE

stat.FILE_ATTRIBUTE_READONLY
stat.FILE_ATTRIBUTE_REPARSE_POINT
stat.FILE_ATTRIBUTE_SPARSE_FILE
stat.FILE_ATTRIBUTE_SYSTEM
stat.FILE_ATTRIBUTE_TEMPORARY
stat.FILE_ATTRIBUTE_VIRTUAL

3.5版本中的新功能。

11.5。filecmp- 文件和目录比较

源代码： Lib / filecmp.py

该filecmp模块定义了比较文件和目录的功能，以及各种可选的时间/正确性折衷。为了比较文件，请参阅difflib模块。

该filecmp模块定义了以下功能：

filecmp.**cmp**（*f1*，*f2*，*shallow = True* ）

比较名为*f1*和*f2*的文件，True如果看起来相同False则返回， 否则返回。

如果*浅*，则具有相同os.stat()签名的文件被认为是相等的。否则，将比较文件的内容。

请注意，此功能不会调用外部程序，因此具有可移植性和效率。

该函数使用缓存来进行过去的比较和结果，如果os.stat()文件的信息发生变化，缓存条目将失效。整个缓存可能会被清除clear_cache()。

filecmp.**cmpfiles**（*dir1*，*dir2*，*common*，*shallow = True* ）

比较两个目录中的文件*dir1*和*dir2*，它们的名称由*common指定*。

返回三个文件名列表：*匹配*，*不匹配*， *错误*。 *匹配*包含*匹配*的文件列表，*不匹配*包含那些*不匹配*的文件的名称，*错误*列出无法比较的文件的名称。如果文件不存在于其中一个目录中，则文件被列为*错误*，用户缺乏读取权限或者由于某些其他原因无法完成比较。

该*浅*参数具有相同的含义和默认值作为 filecmp.cmp()。

例如，将比较有 和用。 并且将分别位于三个返回的列表之一中。 cmpfiles('a', 'b', ['c', 'd/e'])a/cb/ca/d/eb/d/e'c''d/e'

filecmp.**clear_cache**（ ）

清除filecmp缓存。如果文件在被修改后如此快速地进行比较以使其处于基础文件系统的mtime分辨率内，这可能是有用的。

3.4版新增功能

11.5.1。本dircmp类

class filecmp.**dircmp**（*a*，*b*，*ignore = None*，*hide = None* ）

构造一个新的目录比较对象，比较目录*a* 和*b*。 *忽略*是要忽略的名称列表，并且默认为filecmp.DEFAULT_IGNORES。 *hide*是要隐藏的名称列表，默认为。[os.curdir, os.pardir]

如上所述，dircmp该类通过进行*浅层*比较来比较文件filecmp.cmp()。

本dircmp类提供了以下方法：

report ( )
打印（对sys.stdout）*a*和*b*之间*的*比较。

report_partial_closure ( )
打印之间的比较一和*b*，共同立即子目录。

report_full_closure ( )
打印之间的比较一*个*和*b*和公共子目录（递归的）。

本dircmp类提供了许多可用于对被比较的目录树得到的各种信息片段有趣的属性。

请注意，通过__getattr__()钩子，所有属性都会被延迟计算，因此如果仅使用那些轻量级计算的属性，则不会有速度损失。

left
目录*a*。

right
目录*b*。

left_list
文件和子目录一，通过过滤*隐藏*和*忽略*。

right_list
*b中的*文件和子目录，通过*隐藏*和*忽略进行*过滤。

common
*a*和*b中的*文件和子目录。

left_only
文件和子目录只在*a中*。

right_only
文件和子目录仅在*b*。

common_dirs
*a*和*b中的*子目录。

common_files
*a*和*b中的*文件。

common_funny
*a*和*b中的*名称，使得目录中的类型不同，或者os.stat()报告错误的名称不同。

same_files
使用类的文件比较运算符，在*a*和*b中*都是相同的文件。

diff_files
在*a*和*b中*的文件，其内容根据类的文件比较操作符而不同。

funny_files

文件在*a*和*b*中，但无法比较。

subdirs

将名称映射common_dirs到dircmp 对象的字典。

filecmp.DEFAULT_IGNORES

3.4版新增功能

dircmp默认情况下忽略的目录列表。

以下是使用subdirs属性通过两个目录递归搜索以显示公共不同文件的简单示例：

```
>>> from filecmp import dircmp
>>> def print_diff_files(dcmp):
...     for name in dcmp.diff_files:
...         print("diff_file %s found in %s and %s" % (name, dcmp.left,
...               dcmp.right))
...     for sub_dcmp in dcmp.subdirs.values():
...         print_diff_files(sub_dcmp)
...
>>> dcmp = dircmp('dir1', 'dir2')
>>> print_diff_files(dcmp)
```

11.6。 tempfile- 生成临时文件和目录

源代码： Lib / tempfile.py

该模块创建临时文件和目录。它适用于所有支持的平台。 TemporaryFile ， NamedTemporaryFile 和 TemporaryDirectory ，并且SpooledTemporaryFile是提供自动清理并且可以用作上下文管理器的高级接口。 mkstemp()并且 mkdtemp()是需要手动清理的较低级别的功能。

所有用户可调用的函数和构造函数都带有额外的参数，这些参数允许直接控制临时文件和目录的位置和名称。该模块使用的文件名包括一串随机字符，这些字符允许在共享临时目录中安全地创建这些文件。为了保持向后兼容性，论证顺序有点奇怪; 为了清楚起见，建议使用关键字参数。

该模块定义了以下用户可调用的项目：

tempfile. TemporaryFile (*mode ='w + b'* , *buffering = None* , *encoding = None* , *newline = None* , *suffix = None* , *prefix = None* , *dir = None*)

返回可用作临时存储区域的文件类对象。该文件安全地创建，使用相同的规则mkstemp()。一旦它关闭，它就会被销毁（包括当对象被垃圾收集时隐式关闭）。在Unix下，文件的目录条目根本不会创建，或者在创建文件后立即删除。其他平台不支持这个; 您的代码不应该依赖使用此函数创建的临时文件，而该文件在文件系统中具有或不具有可见名称。

结果对象可以用作上下文管理器（请参阅 示例 ）。在完成文件对象的上下文或销毁时，临时文件将从文件系统中移除。

所述*模式*参数默认为'w+b'使得所创建的文件可以被读取并且没有被关闭写入。使用二进制模式，以便在所有平台上保持一致，而不考虑存储的数据。 *缓冲* ， *编码*和*换行*解释为open()。

该*目录* ， *前缀*和*后缀*的参数具有相同的含义和缺省值与mkstemp()。

返回的对象是POSIX平台上的真实文件对象。在其他平台上，它是一个文件类对象，其file属性是底层的真实文件对象。

如果os.O_TMPFILE标志可用且可用（使用Linux，需要Linux内核3.11或更高版本），则使用该标志。

在版本3.5中更改： os.O_TMPFILE现在使用该标志（如果可用）。

tempfile. NamedTemporaryFile (*mode ='w + b'* , *buffering = None* , *encoding = None* , *newline = None* , *suffix = None* , *prefix = None* , *dir = None* , *delete = True*)

这个函数的运行方式与此相同TemporaryFile()，除了该文件保证在文件系统中具有可见名称（在Unix上，目录项不是未链接的）。该名称可以从name返回的类文件对象的属性中检索。当命名的临时文件仍处于打开状态时，名称是否可用于第二次打开该文件，这在不同平台上有所不同（它可以在Unix上使用，不能在Windows NT或更高版本上使用）。如果*删除*为真（默认），则该文件在关闭后立即删除。返回的对象始终是一个类文件对象，其

file 属性是底层的真实文件对象。这个类文件对象可以在一个with语句中使用，就像普通文件一样。

tempfile.SpooledTemporaryFile（*max_size = 0*，*mode ='w + b'*，*buffering = None*，*encoding = None*，*newline = None*，*suffix = None*，*prefix = None*，*dir = None* ）

TemporaryFile()除了数据在内存中被缓存直到文件大小超过*max_size*或fileno()调用文件的方法之前，此功能的运行方式与此相同，此时内容将写入磁盘并且操作将继续进行TemporaryFile()。

生成的文件有一个额外的方法，rollover()它会导致文件无论大小如何都会转到磁盘上的文件。

返回的对象是一个类文件对象，其_file属性是一个io.BytesIO或一个io.StringIO对象（取决于是指定二进制还是文本*模式*）还是一个真正的文件对象，具体取决于是否rollover()被调用。这个类文件对象可以在一个with语句中使用，就像普通文件一样。

在版本3.3中改变了： truncate方法现在接受一个size参数。

tempfile.TemporaryDirectory（*suffix = None*，*prefix = None*，*dir = None* ）

该功能可以使用与以下相同的规则安全地创建临时目录mkdtemp()。结果对象可以用作上下文管理器（请参阅 示例）。在完成临时目录对象的上下文或销毁时，新创建的临时目录及其所有内容将从文件系统中删除。

目录名称可以从name返回对象的属性中检索。当返回的对象用作上下文管理器时 ，如果有的话，name它将被分配给语句中as子句的目标with。

通过调用该cleanup()方法可以明确地清除该目录 。

3.2版本中的新功能

tempfile.mkstemp（*suffix = None*，*prefix = None*，*dir = None*，*text = False* ）

尽可能以最安全的方式创建一个临时文件。假设平台正确实现了该os.O_EXCL标志，文件创建时就没有竞争条件os.open()。该文件只能通过创建用户标识来读写。如果平台使用权限位来指示文件是否可执行，则该文件可由任何人执行。文件描述符不被子进程继承。

与此不同的是TemporaryFile()，用户mkstemp()负责在完成临时文件时删除临时文件。

如果*后缀*不是None，文件名将以该后缀结尾，否则不会有后缀。 mkstemp()不在文件名和后缀之间加点; 如果你需要，把它放在*后缀*的开头。

如果*前缀*不是None，文件名将以该前缀开头; 否则，使用默认的前缀。默认值是gettempprefix()或的返回值 gettempprefixb()，如适用。

如果*dir*不是None，则该文件将在该目录中创建; 否则，使用默认目录。默认目录是从依赖于平台的列表中选择的，但应用程序的用户可以通过设置*TMPDIR*，*TEMP*或*TMP* 环境变量来控制目录位置。因此不能保证生成的文件名将具有任何好的属性，例如当通过外部命令传递时不需要引用os.popen()。

如果任何*后缀*，*前缀*和*目录*不是 None，它们必须是相同的类型。如果它们是字节，则返回的名称将是字节而不是str。如果你想用另外的默认行为强制返回一个字节值，传递

suffix=b''。

如果指定了*文本*，则表示是否以二进制模式（默认）或文本模式打开文件。在某些平台上，这没有什么区别。

mkstemp() os.open()以该顺序返回一个包含操作系统级句柄的元组到一个打开的文件（将返回）以及该文件的绝对路径名。

版本3.5中已更改：后缀，*前缀*和*目录*现在可以按字节提供以获取字节返回值。在此之前，只有str是被允许的。 *后缀*和*前缀*现在接受并默认为None使用适当的默认值。

tempfile.**mkdtemp**（*suffix = None*，*prefix = None*，*dir = None* ）

尽可能以最安全的方式创建临时目录。目录创建时没有竞争条件。该目录只能通过创建用户标识才可读，可写和可搜索。

完成后，用户mkdtemp()负责删除临时目录及其内容。

该*前缀*，*后缀*和*DIR*参数是一样的 mkstemp()。

mkdtemp() 返回新目录的绝对路径名。

版本3.5中已更改：后缀，*前缀*和*目录*现在可以按字节提供以获取字节返回值。在此之前，只有str是被允许的。 *后缀*和*前缀*现在接受并默认为None使用适当的默认值。

tempfile.**gettempdir**（ ）

返回用于临时文件的目录的名称。这为该模块中的所有函数定义了*dir*参数的默认值。

Python搜索一个标准的目录列表，找到一个主叫用户可以创建文件的列表。列表如下：

1. 由。命名的目录 TMPDIR 环境变量。
2. 由。命名的目录 TEMP 环境变量。
3. 由。命名的目录 TMP 环境变量。
4. 平台特定的位置：
    - 在Windows中，目录C:\TEMP，C:\TMP， \TEMP，并\TMP按此顺序。
    - 在所有其他平台，目录/tmp，/var/tmp以及 /usr/tmp在这个顺序。
5. 作为最后的手段，当前的工作目录。

此搜索的结果被缓存，请参阅tempdir下面的说明 。

tempfile.**gettempdirb**（ ）

相同，gettempdir()但返回值以字节为单位。

3.5版本中的新功能。

tempfile.**gettempprefix**（ ）

返回用于创建临时文件的文件名前缀。这不包含目录组件。

tempfile.**gettempprefixb**（ ）

相同，gettempprefix()但返回值以字节为单位。

3.5版本中的新功能。

该模块使用全局变量来存储用于返回临时文件的目录名称gettempdir()。可以直接设置它来覆盖选择过程，但这是不鼓励的。该模块中的所有函数都带有一个*dir*参数，可用于指定目录，这是推荐的方法。

tempfile.**tempdir**

当设置为除此之外的值时None，此变量为本模块中定义的函数定义*dir*参数的默认值。

如果tempdir未设置，或者None对上述任何函数的调用，除了gettempprefix()按照上述算法进行初始化gettempdir()。

11.6.1。示例

以下是tempfile模块典型用法的一些示例：

```
>>> import tempfile

# create a temporary file and write some data to it
>>> fp = tempfile.TemporaryFile()
>>> fp.write(b'Hello world!')
# read data from file
>>> fp.seek(0)
>>> fp.read()
b'Hello world!'
# close the file, it will be removed
>>> fp.close()

# create a temporary file using a context manager
>>> with tempfile.TemporaryFile() as fp:
...     fp.write(b'Hello world!')
...     fp.seek(0)
...     fp.read()
b'Hello world!'
>>>
# file is now closed and removed

# create a temporary directory using the context manager
>>> with tempfile.TemporaryDirectory() as tmpdirname:
...     print('created temporary directory', tmpdirname)
>>>
# directory and contents have been removed
```

11.6.2。弃用的函数和变量

创建临时文件的历史方法是首先使用该mktemp()函数生成一个文件名，然后使用该名称创建一个文件。不幸的是，这是不安全的，因为不同的进程可能会在调用mktemp()和随后尝试通过第一个进程创建文件之间的时间内创建一个具有此名称的文件。解决方案是组合这两个步骤并立即创建文件。这种方法被mkstemp()上面描述的其他功能使用。

tempfile.**mktemp**(*suffix=''*, *prefix='tmp'*, *dir=None*)

*自2.3版弃用：*mkstemp()改为使用。

返回进行调用时不存在的文件的绝对路径名。该*前缀*，*后缀*和*DIR*参数类似于那些 mkstemp()，除了字节的文件名，suffix=None 并且prefix=None不被支持。

警告： 使用此功能可能会在程序中引入安全漏洞。当你回过头来对文件名做任何事情时，其他人可能会殴打你。 mktemp()用法可以轻松替换NamedTemporaryFile()，并传递delete=False参数：

```
>>> f = NamedTemporaryFile(delete=False)
>>> f.name
'/tmp/tmptjujjt'
>>> f.write(b"Hello World!\n")
13
>>> f.close()
>>> os.unlink(f.name)
>>> os.path.exists(f.name)
False
```

11.7。glob- Unix样式路径名称模式扩展

源代码： Lib / glob.py

该glob模块发现所有匹配的根据由Unix外壳使用的规则指定模式的路径名，尽管结果在任意的顺序返回。没有代字符扩展已完成，但是*，?和表达的字符范围[]将正确匹配。这是通过使用函数os.scandir()和fnmatch.fnmatch()函数完成的，而不是通过实际调用子shell来完成的。请注意，不同的是fnmatch.fnmatch()，作为特殊情况glob处理以点（.）开头的文件名。（对于代字号和外壳变量扩展，请使用os.path.expanduser()和 os.path.expandvars()。）

对于文字匹配，将元字符括在括号中。例如，'[?]'匹配字符'?'。

也可以看看： 该pathlib模块提供高级路径对象。

glob.**glob**（*pathname*，***，*recursive = False*）

返回匹配*路径名*的可能为空的路径名列表，该列表必须是包含路径规范的字符串。*路径名*可以是绝对的（如/usr/src/Python-1.5/Makefile）或相对的（如../../Tools/*/*.gif），并且可以包含shell风格的通配符。结果中包含损坏的符号链接（如在shell中）。

如果*递归*为真，模式“**”将匹配任何文件和零个或多个目录和子目录。如果该模式后跟一个os.sep，则只有目录和子目录匹配。

注意： **在大型目录树中使用“**”模式可能会消耗大量的时间。

*在版本3.5中更改：*使用“**”支持递归球。

glob.**iglob**（*pathname*，***，*recursive = False*）

返回一个产生相同值的迭代器，glob()而不是实际同时存储它们。

glob.**escape**（*路径名*）

转义所有特殊字符（'?'，'*'和'['）。如果你想匹配一个可能有特殊字符的任意字符串，这很有用。驱动器/ UNC共享点中的特殊字符不会被转义，例如在Windows 返回时。escape('//?/c:/Quo vadis?.txt')'//?/c:/Quo vadis[?].txt'

3.4版新增功能

例如，考虑包含以下文件的目录：1.gif，2.txt，card.gif和子目录sub 仅包含文件3.txt。glob()会产生以下结果。注意如何保留路径的任何主要组件。

```
>>> import glob
>>> glob.glob('./[0-9].*')
['./1.gif', './2.txt']
>>> glob.glob('*.gif')
['1.gif', 'card.gif']
>>> glob.glob('?.gif')
['1.gif']
```

```
>>> glob.glob('**/*.txt', recursive=True)
['2.txt', 'sub/3.txt']
>>> glob.glob('./**/', recursive=True)
['./', './sub/']
```

如果目录中包含的文件.以默认值不匹配开头。例如，考虑一个包含card.gif和 的目录.card.gif：

```
>>> import glob
>>> glob.glob('*.gif')
['card.gif']
>>> glob.glob('.c*')
['.card.gif']
```

也可以看看:

模 fnmatch
Shell风格的文件名（不是路径）扩展

11.8。 fnmatch- Unix文件名模式匹配

源代码： Lib / fnmatch.py

该模块提供用于Unix shell风格的通配符，这是支持*不*一样的正则表达式（这是中记录re模块）。shell式通配符中使用的特殊字符是：

模式	含义
*	匹配一切
?	匹配任何单个字符
[seq]	匹配*seq*中的任何字符
[!seq]	匹配不在*seq*中的任何字符

对于文字匹配，将元字符括在括号中。例如，'[?]'匹配字符'?'。

请注意，文件名分隔符（'/'在Unix上）对于这个模块*并不*特别。请参阅模块glob以获取路径名称扩展（glob用于 fnmatch()匹配路径名段）。同样，以句点开头的文件名对于这个模块并不是特别的，并且*和? 模式匹配。

fnmatch.**fnmatch**(*文件名*, *模式*)

测试*文件名*字符串是否匹配*模式*字符串，返回 True或False。这两个参数都是使用情况归一化的os.path.normcase()。 fnmatchcase()可用于执行区分大小写的比较，而不管操作系统是否为标准。

这个例子将打印当前目录中所有扩展名为的文件名.txt：

```
import fnmatch
import os

for file in os.listdir('.'):
    if fnmatch.fnmatch(file, '*.txt'):
        print(file)
```

fnmatch.**fnmatchcase**(*文件名*, *模式*)

测试*文件名*是否匹配*模式*，返回True或 False; 比较区分大小写，不适用os.path.normcase()。

fnmatch.**filter**(*名称*, *模式*)

返回匹配*模式*的*名称*列表的子集。这是相同的 ，但更高效地实施。[n for n in names if fnmatch(n, pattern)]

fnmatch.**translate**(*模式*)

将shell样式*模式*转换为正则表达式以供使用re.match()。

例：

```
>>> import fnmatch, re
>>>
>>> regex = fnmatch.translate('*.txt')
>>> regex
'(?s:.*\\.txt)\\Z'
>>> reobj = re.compile(regex)
>>> reobj.match('foobar.txt')
<_sre.SRE_Match object; span=(0, 10), match='foobar.txt'>
```

也可以看看:

模 glob
Unix shell风格的路径扩展。

11.9。linecache- 随机访问文本行

源代码： Lib / linecache.py

该linecache模块允许从Python源文件中获取任何行，同时尝试使用缓存进行内部优化，这是一种从单个文件中读取多行的常见情况。这被traceback模块用来检索包含在格式化回溯中的源代码行。

该tokenize.open()功能用于打开文件。该函数用于tokenize.detect_encoding()获取文件的编码; 在没有编码标记的情况下，文件编码默认为UTF-8。

该linecache模块定义了以下功能：

linecache.getline（*文件名*，*lineno*，*module_globals* =*无*）
从文件命名*文件名中*获取*lineno*行*号*。这个函数永远不会引发异常 - 它会在错误时返回（终止的换行符将包含在找到的行中）。''

如果找不到名为*filename的文件*，则该函数将在模块搜索路径中查找它sys.path，首先检查一个在模块从zipfile或其他非文件系统导入源导入的情况下__loader__，*module_globals中的*PEP 302。

linecache.clearcache（）
清除缓存。如果您不再需要先前读取的文件中的行，请使用此功能getline()。

linecache.checkcache（*filename = None*）
检查缓存的有效性。如果缓存中的文件可能已在磁盘上发生更改，并且您需要更新的版本，请使用此功能。如果省略了*filename*，它将检查缓存中的所有条目。

linecache.lazycache（*文件名*，*module_globals*）
捕获足够的细节，关于非基于文件的模块，允许获取其行通过后getline()，即使*module_globals*是None在以后调用。这避免了I / O直到实际需要一行，而不必无限地携带模块全局变量。

3.5版本中的新功能。

例：

```
>>> import linecache
>>> linecache.getline(linecache.__file__, 8)
'import sys\n'
```

11.10。 shutil- 高级文件操作

源代码： Lib / shutil.py

该shutil模块提供了许多关于文件和文件集合的高级操作。特别提供了支持文件复制和删除的功能。有关单个文件的操作，另请参阅 os模块。

警告: 即使更高级别的文件复制功能（ shutil.copy() ， shutil.copy2() ）也不能复制所有文件元数据。

在POSIX平台上，这意味着文件所有者和组以及ACL都会丢失。在Mac OS上，不使用资源分叉和其他元数据。这意味着资源将会丢失，文件类型和创建者代码将不正确。在Windows上，不会复制文件所有者，ACL和备用数据流。

11.10.1。目录和文件操作

shutil. copyfileobj (*FSRC* , *fdst* [, *长度*])

在类文件对象的内容复制*金管会*的文件对象*fdst*。整数*长度*（如果给出）是缓冲区大小。具体而言，负 *长度*值意味着复制数据，而不会以块的形式循环源数据; 默认情况下，数据以块读取，以避免不受控制的内存消耗。请注意，如果*fsrc*对象的当前文件位置不为0，则只会复制当前文件位置到文件末尾的内容。

shutil. copyfile (*src* , *dst* , * , *follow_symlinks = True*)

将名为*src*的文件的内容（无元数据）复制到名为*dst*的文件 并返回*dst*。 *src*和*dst*是以字符串形式给出的路径名。 *dst*必须是完整的目标文件名; 查看shutil.copy() 接受目标目录路径的副本。如果*src*和*dst* 指定相同的文件，SameFileError则会引发。

目标位置必须可写; 否则OSError 会引发异常。如果*dst*已经存在，它将被替换。特殊文件如字符或块设备和管道不能使用此功能进行复制。

如果*follow_symlinks*为false且*src*是符号链接，则会创建一个新的符号链接，而不是将文件*src*指向的文件复制到该链接。

在版本3.3中更改： IOError用于提高而不是OSError。增加了*follow_symlinks*参数。现在返回*dst*。

在版本3.4中更改： 升起SameFileError而不是Error。由于前者是后者的一个子类，这种变化是向后兼容的。

*异常*shutil. SameFileError

如果源和目标copyfile() 是相同的文件，则会引发此异常。

3.4版新增功能

shutil. copymode (*src* , *dst* , * , *follow_symlinks = True*)

将权限位从*src*复制到*dst*。文件内容，所有者和组不受影响。 *src*和*dst*是以字符串形式给出的路径名。如果*follow_symlinks*为false，并且*src*和*dst*都是符号链接， `copymode()` 则会尝试修改*dst*本身的模式（而不是它指向的文件）。此功能在每个平台上都不可用; 请参阅 `copystat()` 更多信息。如果 `copymode()` 不能修改本地平台上的符号链接，并且要求这样做，它将不会做任何事情并返回。

*版本3.3中更改：*添加了*follow_symlinks*参数。

shutil.copystat（*src*，*dst*，*，*follow_symlinks = True*）

将权限位，上次访问时间，上次修改时间以及*src*中的标志复制到*dst*。在Linux上，`copystat()` 也尽可能复制“扩展属性”。文件内容，所有者和组不受影响。 *src*和*dst*是以字符串形式给出的路径名。

如果*follow_symlinks*为false，并且*src*和*dst*都引用符号链接，则它们`copystat()`将在符号链接本身上运行，而不是符号链接引用的文件 - 从*src*符号链接读取信息 ，并将信息写入 *dst*符号链接。

> **注意：** 并非所有平台都能够检查和修改符号链接。Python本身可以告诉你哪些功能是本地可用的。
>
> - 如果是 ，可以修改符号链接的权限位。 `os.chmod` in `os.supports_follow_symlinks` True `copystat()`
> - 如果是 ，可以修改符号链接的上次访问和修改时间。 `os.utime` in `os.supports_follow_symlinks` True `copystat()`
> - 如果是 ，可以修改符号链接的标志。（在所有平台上都不可用）。 `os.chflags` in `os.supports_follow_symlinks` True `copystat()` `os.chflags`
>
> 在某些或全部功能不可用的平台上，当被要求修改符号链接时， `copystat()` 将复制它所能做的所有事情。 `copystat()` 永远不会失败。
>
> 请参阅`os.supports_follow_symlinks` 更多信息。

*在版本3.3中进行了更改：*添加了*follow_symlinks*参数并支持Linux扩展属性。

shutil.copy（*src*，*dst*，*，*follow_symlinks = True*）

将文件*src*复制到文件或目录*dst*。 *src*和*dst* 应该是字符串。如果*dst*指定了一个目录，则该文件将使用*src*中的基本文件名复制到*dst*中。返回新创建文件的路径。

如果*follow_symlinks*为false，并且*src*是符号链接，则 *dst*将被创建为符号链接。如果*follow_symlinks* 为true并且*src*是符号链接，则*dst*将是*src*引用的文件的副本。

`copy()` 复制文件数据和文件的权限模式（请参阅`os.chmod()` ）。其他元数据，如文件的创建和修改时间，不会保留。要保留原始文件中的所有文件元数据，请`copy2()`改为使用 。

*版本3.3中更改：*添加了*follow_symlinks*参数。现在将路径返回到新创建的文件。

shutil.copy2（*src*，*dst*，*，*follow_symlinks = True*）

与`copy()`除了`copy2()` 还会尝试保留所有文件元数据相同。

当*follow_symlinks*为false且*src*为符号链接时，会`copy2()`尝试将所有元数据从*src*符号链接复制 到新创建的*dst*符号链接。但是，此功能在所有平台上都不可用。在部分或全部功能不

可用的平台上，copy2()将保留所有可能的元数据; copy2()永远不会失败。

copy2()用于copystat()复制文件元数据。请参阅copystat()有关修改符号链接元数据的平台支持的更多信息。

*在版本3.3中进行了更改：*增加了*follow_symlinks*参数，尝试复制扩展文件系统属性（目前仅限于Linux）。现在将路径返回到新创建的文件。

shutil.ignore_patterns(**模式*)

这个工厂函数创建一个可以作为一个可调用一个函数 copytree()的*忽略*说法，忽略匹配通配符式样的一个文件和目录的*方式*提供。看下面的例子。

shutil.copytree(*src*，*dst*，*symlinks = False*，*ignore = None*，*copy_function = copy2*，*ignore_dangling_symlinks = False*)

递归复制以*src*为根的整个目录树，返回目标目录。由*dst*命名的目标目录不能存在; 它将被创建以及丢失的父目录。权限和目录的时间被复制copystat() ，个别文件被复制使用shutil.copy2()。

如果*符号链接*为真，则源树中的符号链接将表示为新树中的符号链接，原始链接的元数据将被复制到平台允许的范围内; 如果为false或省略，链接文件的内容和元数据将被复制到新树中。

当*符号链接*为false时，如果*符号链接指向*的文件不存在，则将Error在复制过程结束时异常中引发的错误列表中添加异常。如果您想要消除此异常，可以将可选*ignore_dangling_symlinks*标志设置为true。请注意，该选项对不支持的平台没有影响os.symlink()。

如果给出了*忽略*，它必须是一个可调用的函数，它将接收作为其参数的被访问目录copytree() ，以及其返回的内容列表os.listdir()。由于copytree()被递归调用，所以被复制的每个目录都会调用一次*忽略*调用。可调用函数必须返回一系列相对于当前目录的目录和文件名称（即第二个参数中的项目子集）; 这些名称将在复制过程中被忽略。ignore_patterns()可用于创建可忽略基于全局样式模式的名称的可调用对象。

如果发生异常，Error则列出一个理由列表。

如果给出了*copy_function*，它必须是可用于复制每个文件的可调用*函数*。它将以源路径和目标路径作为参数进行调用。默认shutil.copy2()使用，但可以使用任何支持相同签名（如shutil.copy() ）的函数。

*在版本3.3中更改：*当*符号链接*为false 时复制元数据。现在返回*dst*。

*在版本3.2中更改：*添加了*copy_function*参数以能够提供自定义复制功能。当*符号链接*为false 时，将*silent_dangling_symlinks*参数添加到无提示符号链接错误中。

shutil.rmtree(*path*，*ignore_errors = False*，*onerror = None*)

删除整个目录树; *路径*必须指向一个目录（但不是指向目录的符号链接）。如果*ignore_errors*为true，那么由失败的清除导致的错误将被忽略; 如果为false或省略，则通过调用由*onerror*指定的处理程序来处理此类错误，或者如果省略，则会引发异常。

注意： 在支持必要的基于fd的功能的平台上，rmtree()缺省情况下使用符号链接攻击抵抗版本。在其他平台上，rmtree()实现容易受到符号链接攻击：在适当的时间和环境下，攻击者可以操纵文件系统上的符号链接来删除他们无法访问的文件。应用程序可以使用rmtree.avoids_symlink_attacks 函数属性来确定哪种情况适用。

如果提供了*onerror*，它必须是可接受的，它接受三个参数：*函数*，*路径*和*excinfo*。

第一个参数*函数*是引发异常的函数; 它取决于平台和实施。第二个参数 *path*将是传递给*函数*的路径名。第三个参数 *excinfo*将是返回的异常信息 sys.exc_info()。通过抛出的异常*的onerror*不会被抓住。

*在版本3.3中更改：*添加了符号链接攻击抵抗版本，如果平台支持基于fd的功能，则会自动使用该版本。

rmtree.avoids_symlink_attacks

指示当前平台和实施是否提供符号链接攻击抵抗版本rmtree()。目前，这仅适用于支持基于fd的目录访问功能的平台。

3.3版本的新功能

shutil.move（*src*，*dst*，*copy_function* = *copy2* ）

将文件或目录（*src*）递归移动到另一个位置（*dst*）并返回目标。

如果目的地是一个现有的目录，那么*src*会在该目录内移动。如果目标已经存在但不是目录，则可能会根据os.rename()语义覆盖目标。

如果目标位于当前文件系统上，则会os.rename()被使用。否则，使用*copy_function*将*src*复制到*dst*，然后删除。在符号链接的情况下，指向*src*目标的新符号链接 将在*dst中*创建，或者*dst*和*src*将被删除。

如果给出了*copy_function*，那么它必须是一个可调用的函数，它接受两个参数 *src*和*dst*，如果 不能使用，将用于将*src*复制到*dest*os.rename()。如果源是一个目录， copytree()则调用它，传递它copy_function()。默认的*copy_function*是copy2()。使用copy()作为*copy_function*允许移动成功时，它不可能也复制元数据，在没有任何复制元数据的费用。

*在版本3.3中进行了更改：*为外部文件系统添加了显式符号链接处理，从而使其适应GNU **mv**的行为。现在返回*dst*。

*在版本3.5中进行了更改：*添加了*copy_function*关键字参数。

shutil.disk_usage（*路径*）

返回有关给定的路径作为盘使用统计命名元组 与所述属性*总量*，*使用*和*自由*，这是总的量，所用和可用空间，以字节为单位。在Windows上，*路径*必须是目录; 在Unix上，它可以是一个文件或目录。

3.3版本的新功能

可用性：Unix，Windows。

shutil.chown（*路径*，*用户=无*，*组=无*）

更改所有者*用户*和/或给定*路径的组*。

*用户*可以是系统用户名或uid; 这同样适用于*组*。至少需要一个参数。

另请参阅os.chown()底层函数。

可用性：Unix。

3.3版本的新功能

shutil.**which**(*cmd*，*mode = os.F_OK | os.X_OK*，*path = None*)

如果给定*cmd*被调用，则返回可执行文件的路径。如果不会调用*cmd*，则返回None。

*模式*是传递给的权限掩码os.access()，默认情况下确定文件是否存在并且可执行。

如果未指定*路径*，os.environ()则使用结果，返回“PATH”值或后退os.defpath。

在Windows上，当前目录始终位于*路径的*前面，而不管您是否使用默认*路径*或提供您自己的*路径*，这是命令shell在查找可执行文件时使用的行为。另外，在*路径中*找到 *cmd*时，会检查环境变量。例如，如果您调用，将搜索 以知道它应该在*路径* 目录中查找。例如，在Windows上：PATHEXT shutil.which("python") which() PATHEXT python.exe

```
>>> shutil.which("python")
'C:\\Python33\\python.EXE'
```

3.3版本的新功能

*异常*shutil.**Error**

此异常收集多文件操作期间引发的异常。因为copytree()，异常参数是一个3元组列表（*srcname*，*dstname*，*exception*）。

11.10.1.1。copytree示例

这个例子是copytree()上面描述的函数的实现，省略了docstring。它演示了该模块提供的许多其他功能。

```
def copytree(src, dst, symlinks=False):
    names = os.listdir(src)
    os.makedirs(dst)
    errors = []
    for name in names:
        srcname = os.path.join(src, name)
        dstname = os.path.join(dst, name)
        try:
            if symlinks and os.path.islink(srcname):
                linkto = os.readlink(srcname)
                os.symlink(linkto, dstname)
            elif os.path.isdir(srcname):
                copytree(srcname, dstname, symlinks)
            else:
                copy2(srcname, dstname)
            # XXX What about devices, sockets etc.?
```

```
            except OSError as why:
                errors.append((srcname, dstname, str(why)))
            # catch the Error from the recursive copytree so that we can
            # continue with other files
            except Error as err:
                errors.extend(err.args[0])
        try:
            copystat(src, dst)
        except OSError as why:
            # can't copy file access times on Windows
            if why.winerror is None:
                errors.extend((src, dst, str(why)))
        if errors:
            raise Error(errors)
```

另一个使用ignore_patterns()助手的例子是：

```
from shutil import copytree, ignore_patterns

copytree(source, destination, ignore=ignore_patterns('*.pyc', 'tmp*'))
```

这将复制除.pyc文件和文件或名称以其开头的目录之外的所有内容tmp。

另一个使用*ignore*参数添加日志记录调用的示例：

```
from shutil import copytree
import logging

def _logpath(path, names):
    logging.info('Working in %s', path)
    return []   # nothing will be ignored

copytree(source, destination, ignore=_logpath)
```

11.10.1.2。rmtree例子

此示例显示如何在Windows上删除某些文件的只读位已设置的目录树。它使用onerror回调来清除只读位，并重新尝试删除。任何后续的失败都会传播。

```
import os, stat
import shutil

def remove_readonly(func, path, _):
    "Clear the readonly bit and reattempt the removal"
    os.chmod(path, stat.S_IWRITE)
    func(path)

shutil.rmtree(directory, onerror=remove_readonly)
```

11.10.2。归档操作

3.2版本中的新功能

*在版本3.5中进行了更改：*增加了对*xztar*格式的支持。

还提供了创建和读取压缩和存档文件的高级实用程序。他们依靠zipfile和tarfile模块。

shutil.make_archive（*base_name*，*format*[，*root_dir*[，*base_dir*[，*verbose*[，*dry_run*[，*owner*[，*group*[，*logger*]]]]]]]）

创建一个存档文件（如zip或tar）并返回其名称。

*base_name*是要创建的文件的名称，包括路径，减去任何格式特定的扩展名。*格式*是归档格式：“zip”（如果zlib模块可用），“tar”，“gztar”（如果 zlib模块可用），“bztar”（如果bz2模块可用）或“xztar”（如果lzma模块可用）。

*root_dir*是一个将成为存档根目录的目录; 例如，我们通常在创建归档之前先chdir到*root_dir*。

*base_dir*是我们开始归档的目录; 即*base_dir*将是档案中所有文件和目录的通用前缀。

*root_dir*和*base_dir*都默认为当前目录。

如果*dry_run*为true，则不创建存档，但将执行的操作记录到*记录器*。

*所有者*和*组*在创建tar归档时使用。默认情况下，使用当前所有者和组。

*记录器*必须是兼容的对象**PEP 282**，通常是一个实例 logging.Logger。

的*详细*参数是未使用和废弃。

shutil.get_archive_formats（）

返回支持格式的列表进行存档。返回序列的每个元素都是一个元组。(name, description)

默认情况下shutil提供这些格式：

- *zip*：ZIP文件（如果zlib模块可用）。
- *tar*：解压缩的tar文件。
- *gztar*：gzip'ed tar文件（如果zlib模块可用）。
- *bztar*：bzip2'ed tar文件（如果bz2模块可用）。
- *xztar*：xz'ed tar文件（如果lzma模块可用）。

您可以通过使用注册新格式或为任何现有格式提供您自己的归档程序register_archive_format()。

shutil.register_archive_format（*名称*，*函数*[，*extra_args*[，*description*]]）

注册归档程序以获取格式*名称*。

*函数*是可用于解压缩存档的可调用*函数*。可调用对象将接收要创建的文件的*base_name*，然后接收 *base_dir*（默认为os.curdir）以开始从中进行归档。更多参数作为关键字参数传递：*owner*，*group*， *dry_run*和*logger*（如传入make_archive()）。

如果给定，则*extra_args*是一组将用作额外关键字参数的对的序列，当使用归档器可调用时。(name, value)

*描述*用于get_archive_formats()返回归档器列表。缺省为空字符串。

shutil.unregister_archive_format(*名字*)

从支持的格式列表中删除存档格式*名称*。

shutil.unpack_archive(*filename*[, *extract_dir*[, *format*]])

解压档案。*filename*是存档的完整路径。

*extract_dir*是存档解压目标目录的名称。如果未提供，则使用当前的工作目录。

*格式*是归档格式：“zip”，“tar”，“gztar”，“bztar”或“xztar”之一。或以其他任何格式注册register_unpack_format()。如果未提供，unpack_archive()则将使用存档文件扩展名并查看是否为该扩展程序注册了解包器。如果没有发现，ValueError则提出a。

shutil.register_unpack_format(*名称*, *扩展名*, *函数*[, *extra_args*[, *description*]])

注册一个解包格式。*名称*是格式的名称，*扩展名*是与格式相对应的扩展名列表，例如.zipZip文件。

*函数*是可用于解压缩存档的可调用*函数*。可调用的将接收归档的路径，后面是归档必须提取到的目录。

提供时，*extra_args*是一系列将作为关键字参数传递给可调用对象的元组。(name, value)

*描述*可被提供来描述的格式，并且将被返还get_unpack_formats()的功能。

shutil.unregister_unpack_format(*名字*)

取消注册解压缩格式。*名称*是格式的名称。

shutil.get_unpack_formats()

返回所有注册格式的解包列表。返回序列的每个元素都是一个元组。(name, extensions, description)

默认情况下shutil提供这些格式：

- *zip*：ZIP文件（解压缩压缩文件只在相应的模块可用时才起作用）。
- *tar*：解压缩的tar文件。
- *gztar*：gzip'ed tar文件（如果zlib模块可用）。
- *bztar*：bzip2'ed tar文件（如果bz2模块可用）。
- *xztar*：xz'ed tar文件（如果lzma模块可用）。

您可以通过使用注册新格式或为任何现有格式提供您自己的解包器register_unpack_format()。

11.10.2.1。归档示例

在这个例子中，我们创建一个包含.ssh用户目录中所有文件的gzip'ed tar文件存档：

```
>>> from shutil import make_archive
>>> import os
>>> archive_name = os.path.expanduser(os.path.join('~', 'myarchive'))
>>> root_dir = os.path.expanduser(os.path.join('~', '.ssh'))
```

```
>>> make_archive(archive_name, 'gztar', root_dir)
'/Users/tarek/myarchive.tar.gz'
```

结果存档包含：

```
$ tar -tzvf /Users/tarek/myarchive.tar.gz
drwx------ tarek/staff       0 2010-02-01 16:23:40 ./
-rw-r--r-- tarek/staff     609 2008-06-09 13:26:54 ./authorized_keys
-rwxr-xr-x tarek/staff      65 2008-06-09 13:26:54 ./config
-rwx------ tarek/staff     668 2008-06-09 13:26:54 ./id_dsa
-rwxr-xr-x tarek/staff     609 2008-06-09 13:26:54 ./id_dsa.pub
-rw------- tarek/staff    1675 2008-06-09 13:26:54 ./id_rsa
-rw-r--r-- tarek/staff     397 2008-06-09 13:26:54 ./id_rsa.pub
-rw-r--r-- tarek/staff   37192 2010-02-06 18:23:10 ./known_hosts
```

11.10.3。查询输出终端的大小

shutil.**get_terminal_size**(*fallback = (columns , lines)*)

获取终端窗口的大小。

对于这两个维度中的每一个，分别检查环境变量COLUMNS 和环境变量LINES。如果变量已定义且值为正整数，则使用该变量。

当COLUMNS或LINES未定义时，通常情况下，连接到的终端sys.__stdout__通过调用进行查询os.get_terminal_size()。

如果无法成功查询终端大小，或者因为系统不支持查询，或者因为我们没有连接到终端，fallback则会使用参数中给出的值。fallback默认为许多终端仿真器使用的默认大小。(80, 24)

返回的值是一个指定的类型元组os.terminal_size。

另请参见：单一UNIX规范版本2， 其他环境变量。

3.3版本的新功能

11.11。 macpath- Mac OS 9路径操作功能

源代码： Lib / macpath.py

该模块是模块的Mac OS 9（及更早版本）实现`os.path` 。它可用于在Mac OS X（或任何其他平台）上操作旧式Macintosh路径名。

以下功能在此模块中可用：`normcase()` ， `normpath()` ，`isabs()` ，`join()` ，`split()` ，`isdir()` ，`isfile()` ，`walk()` ，`exists()`。对于其他可用的`os.path`虚拟对口功能。

12.数据持久性

本章描述的模块支持将Python数据以永久形式存储在磁盘上。的pickle和marshal模块可以把许多Python数据类型转换成字节流，然后重新创建从字节的对象。各种与DBM相关的模块支持一系列基于散列的文件格式，用于存储字符串到其他字符串的映射。

本章描述的模块列表是：

12.1。pickle- Python对象序列化

源代码： Lib / pickle.py

该pickle模块实现了用于序列化和反序列化Python对象结构的二进制协议。 *“Pickling”*是将Python对象层次结构转换为字节流的过程， *“unpickling”*是相反的操作，即字节流（来自二进制文件或类似字节的对象）被转换回对象层次结构。酸洗（与在unpickle）可替代地被称为“序列”，“编组” [1]或“压扁”; 然而，为了避免混淆，这里使用的术语是“酸洗”和“不沾”。

警告: 该pickle模块对于错误或恶意构建的数据不安全。切勿取消从不可信或未经认证的来源收到的数据。

12.1.1。与其他Python模块的关系

12.1.1.1。与 比较marshal

Python有一个更原始的序列化模块marshal，但通常pickle应该是序列化Python对象的首选方式。 marshal主要是为了支持Python的.pyc 文件。

该pickle模块与以下marshal几个重要方面有所不同：

- 该pickle模块跟踪它已经序列化的对象，以便以后对同一对象的引用不会再次序列化。marshal不这样做。

  这对递归对象和对象共享都有影响。递归对象是包含对自己的引用的对象。这些不是由编组处理的，实际上，尝试编组递归对象会导致Python解释器崩溃。如果在被序列化的对象层次结构中的不同位置存在对同一对象的多个引用，则会发生对象共享。 pickle只存储一次这样的对象，并确保所有其他引用指向主副本。共享对象保持共享，这对于可变对象非常重要。

- marshal不能用于序列化用户定义的类及其实例。 pickle可以透明地保存和恢复类实例，但类定义必须是可导入的，并且与存储对象时位于同一模块中。

- 该marshal序列化格式是不能保证整个Python版本移植。因为它的主要工作是支持.pyc文件，所以Python实现者保留在需要时以非向后兼容方式更改序列化格式的权利。该pickle序列化格式是保证不同的Python版本向后兼容。

12.1.1.2。与 比较json

pickle协议和JSON（JavaScript Object Notation）之间有着根本的区别：

- JSON是一种文本序列化格式（它输出unicode文本，虽然大部分时间都是编码的utf-8），而pickle是一种二进制序列化格式。
- JSON是人类可读的，而腌菜不是;
- JSON可以在Python生态系统之外互操作并广泛使用，而pickle则是Python特有的;

- 默认情况下，JSON只能表示Python内置类型的一个子集，并且不包含自定义类; pickle可以代表大量的Python类型（其中许多是自动的，通过巧妙使用Python的内省设施;复杂的情况可以通过实现特定的对象API来解决）。

也可以看看： 该json模块：允许JSON序列化和反序列化的标准库模块。

12.1.2。数据流格式

所使用的数据格式pickle是Python特有的。这具有如下优点：不存在由诸如JSON或XDR的外部标准（其不能表示指针共享）施加的限制; 然而这意味着非Python程序可能无法重构pickled Python对象。

默认情况下，pickle数据格式使用相对紧凑的二进制表示。如果您需要最佳大小特征，则可以高效地 压缩 pickled数据。

该模块pickletools包含分析生成的数据流的工具pickle。 pickletools源代码对pickle协议使用的操作码有广泛的评论。

目前有5种不同的协议可用于酸洗。使用的协议越高，读取pickle所需的Python版本越新。

- 协议版本0是原始的“人类可读”协议，并且与早期版本的Python向后兼容。
- 协议版本1是一种旧的二进制格式，它也与早期版本的Python兼容。
- 协议版本2是在Python 2.3中引入的。它提供了更有效的酸洗新式课程。参考**PEP 307**了解方案2带来的改进信息。
- 协议版本3是在Python 3.0中添加的。它对bytes对象有明确的支持， 并且不能被Python 2.x取消选择。这是默认协议，并且需要与其他Python 3版本兼容时推荐的协议。
- 协议版本4在Python 3.4中添加。它增加了对非常大的对象的支持，酸洗更多类型的对象以及一些数据格式优化。参考**PEP 3154**了解协议4带来的改进信息。

注意： 序列化是比持久性更原始的概念; 虽然 pickle读取和写入文件对象，但它不处理命名持久对象的问题，也不处理并发访问持久对象的（更复杂的）问题。该pickle模块可以将复杂对象转换为字节流，并且可以将字节流转换为具有相同内部结构的对象。也许对这些字节流最明显的做法是将它们写入文件，但也可以将它们发送到网络或将它们存储在数据库中。该shelve 模块提供了一个简单的界面，可以在DBM样式的数据库文件上腌制和取消对象。

12.1.3。模块接口

要序列化对象层次结构，只需调用该dumps()函数即可。同样，为了反序列化数据流，您可以调用该loads()函数。但是，如果您想要更多地控制序列化和反序列化，则可以分别创建一个Pickler或一个Unpickler对象。

该pickle模块提供以下常量：

pickle.HIGHEST_PROTOCOL
一个整数， 可用的最高协议版本。这个值可以作为一个被传递*协议*的价值函数 dump()和dumps()以及该Pickler 构造函数。

pickle.DEFAULT_PROTOCOL

整数，用于酸洗的默认协议版本。可能会少于HIGHEST_PROTOCOL。目前默认的协议是3，一种为Python 3设计的新协议。

该pickle模块提供以下功能，使酸洗过程更加方便：

pickle.**dump**(*obj*, *file*, *protocol* = *None*, ***, *fix_imports* = *True*)

将*obj*的pickle表示写入打开的文件对象*文件*。这相当于。Pickler(file, protocol).dump(obj)

可选的*协议*参数，一个整数，告诉pickler使用给定的协议；支持的协议是0到HIGHEST_PROTOCOL。如果未指定，则默认为DEFAULT_PROTOCOL。如果指定了负数，HIGHEST_PROTOCOL则选中。

的*文件*参数必须具有接受单个字节的参数写（）方法。因此它可以是为二进制写入，io.BytesIO实例或任何符合此接口的其他自定义对象打开的磁盘上文件。

如果*fix_imports*为true且*protocol*小于3，pickle会尝试将新的Python 3名称映射到Python 2中使用的旧模块名称，以便pickle数据流可以用Python 2读取。

pickle.**dumps**(*obj*, *protocol* = *None*, ***, *fix_imports* = *True*)

将对象的pickled表达式作为bytes对象返回，而不是将其写入文件。

参数*protocol*和*fix_imports*的含义与in相同 dump()。

pickle.**load**(*file*, ***, *fix_imports* = *True*, *encoding* =*“ASCII”*, *errors* =*“strict”*)

从打开的文件对象*文件*读取一个pickle对象表示并返回其中指定的重组对象层次结构。这相当于Unpickler(file).load()。

自动检测泡菜的协议版本，因此不需要任何协议参数。经过腌渍对象的表示的字节被忽略。

参数*文件*必须有两个方法，一个使用整数参数的read（）方法和一个不需要参数的readline（）方法。两种方法都应该返回字节。因此，*文件*可以是为二进制读取打开的磁盘上文件，io.BytesIO对象或符合此界面的任何其他自定义对象。

可选的关键字参数是*fix_imports*，*编码*和*错误*，用于控制由Python 2生成的pickle stream的兼容性支持。如果*fix_imports*为true，pickle将尝试将旧的Python 2名称映射到Python 3中使用的新名称。*编码*和*错误*告诉pickle如何解码由Python 2腌制的8位字符串实例；这些默认值分别为'ASCII'和'strict'。该*编码*可以是“字节”作为字节对象读取这些8位串的实例。

pickle.**loads**(*bytes_object*, ***, *fix_imports* = *True*, *encoding* =*“ASCII”*, *errors* =*“strict”*)

从bytes对象中读取一个pickled object层次结构并返回其中指定的重组对象层次结构。

自动检测泡菜的协议版本，因此不需要任何协议参数。经过腌渍对象的表示的字节被忽略。

可选的关键字参数是*fix_imports*，*编码*和*错误*，用于控制由Python 2生成的pickle stream的兼容性支持。如果*fix_imports*为true，pickle将尝试将旧的Python 2名称映射到Python 3中

使用的新名称。*编码*和 *错误*告诉pickle如何解码由Python 2腌制的8位字符串实例; 这些默认值分别为'ASCII'和'strict'。该*编码*可以是“字节”作为字节对象读取这些8位串的实例。

该pickle模块定义了三个例外：

*异常*pickle.PickleError
其他酸洗例外的通用基类。它继承 Exception。

*异常*pickle.PicklingError
遇到不可取的对象时引发错误Pickler。它继承PickleError。

请参阅什么可以腌制和取消腌制？了解可以腌制什么样的对象。

*异常*pickle.UnpicklingError
在取消对象时出现问题（例如数据损坏或安全违规）时引发的错误。它继承PickleError。

请注意，其他异常也可能在取消过程中引发，包括（但不一定限于）AttributeError，EOFError，ImportError和IndexError。

该pickle模块导出两个类，Pickler并且 Unpickler：

class pickle.Pickler（*file*，*protocol* = *None*，*，*fix_imports* = *True* ）
这需要一个二进制文件来写入一个pickle数据流。

可选的*协议*参数，一个整数，告诉pickler使用给定的协议; 支持的协议是0到HIGHEST_PROTOCOL。如果未指定，则默认为DEFAULT_PROTOCOL。如果指定了负数，HIGHEST_PROTOCOL则选中。

的*文件*参数必须具有接受单个字节的参数写（ ）方法。因此它可以是为二进制写入，io.BytesIO实例或任何符合此接口的其他自定义对象打开的磁盘上文件。

如果*fix_imports*为true且*protocol*小于3，pickle会尝试将新的Python 3名称映射到Python 2中使用的旧模块名称，以便pickle数据流可以用Python 2读取。

dump（*obj* ）
向构造函数中给出的打开的文件对象写一个腌制的*obj*表示形式。

persistent_id（*obj* ）
默认不做任何事情。这存在，所以一个子类可以覆盖它。

如果persistent_id()返回None，*obj*像往常一样腌制。任何其他值都会导致Pickler将返回的值作为*obj*的持久ID发出。这个持久ID的含义应该由Unpickler.persistent_load()。定义。请注意，由返回的值persistent_id()本身不具有持久性ID。

有关使用的详细信息和示例，请参阅外部对象的持久性。

dispatch_table
一个皮克勒对象的调度表的注册表*还原功能*，可使用声明的那种 copyreg.pickle()。它是一个映射，其键是类，其值是简化函数。约简函数接受关联类的单个参数，并且

应该与__reduce__() 方法相同。

默认情况下，pickler对象不具有 dispatch_table属性，而是使用copyreg模块管理的全局调度表。但是，要为特定的pickler对象自定义酸洗，可以将该dispatch_table属性设置为类似字典的对象。或者，如果一个Pickler具有 dispatch_table属性的子类，那么它将被用作该类实例的默认调度表。

有关使用示例，请参阅调度表。

3.3版本的新功能

fast

已过时。如果设置为真值，则启用快速模式。快速模式禁用备忘录的使用，因此通过不会生成多余的PUT操作码来加快酸洗过程。它不应该与自引用对象一起使用，否则会导致Pickler无限递归。

pickletools.optimize()如果你需要更紧凑的腌菜，请使用。

class pickle.Unpickler (*file* , * , *fix_imports = True* , *encoding =“ASCII”* , *errors =“strict”*)

这需要一个二进制文件来读取一个pickle数据流。

自动检测泡菜的协议版本，因此不需要任何协议参数。

参数*文件*必须有两个方法，一个使用整数参数的read（ ）方法和一个不需要参数的readline（ ）方法。两种方法都应该返回字节。因此，*文件*可以是为二进制读取而打开的磁盘上的文件对象，io.BytesIO对象或符合此界面的任何其他自定义对象。

可选的关键字参数是*fix_imports*，*编码*和*错误*，用于控制由Python 2生成的pickle stream的兼容性支持。如果*fix_imports*为true，pickle将尝试将旧的Python 2名称映射到Python 3中使用的新名称。*编码*和 *错误*告诉pickle如何解码由Python 2腌制的8位字符串实例; 这些默认值分别为'ASCII'和'strict'。该*编码*可以是“字节”作为字节对象读取这些8位串的实例。

load ( )

从构造函数中给出的打开文件对象中读取一个pickle对象表示形式，并返回其中指定的重构对象层次结构。经过腌渍对象的表示的字节被忽略。

persistent_load (*pid*)

提高UnpicklingError默认值。

如果定义了，persistent_load()应返回由持久ID *pid*指定的对象。如果遇到无效的持久性ID， UnpicklingError应该引发一个。

有关使用的详细信息和示例，请参阅外部对象的持久性。

find_class (*模块* , *名称*)

如有必要，导入*模块*并返回名为*name*的对象，其中*模块*和*名称*参数是str对象。请注意，与名称不同，find_class()它也用于查找功能。

子类可以覆盖这个来获得对什么类型的对象以及它们如何被加载的控制，从而潜在地降低安全风险。有关详细信息，请参阅 限制全局。

12.1.4。什么可以腌制和不沾？

以下类型可以被腌制：

- None，True和False
- 整数，浮点数，复数
- 字符串，字节，字节数组
- 元组，列表，集合和仅包含可选对象的字典
- 在模块顶层定义的函数（def不 使用lambda）
- 在模块顶层定义的内置函数
- 在模块顶层定义的类
- 这些类的实例__dict__或调用的结果__getstate__()是可挑选的（请参见Pickling类实例一节了解详细信息）。

尝试pickle unpicklable对象会引发PicklingError 异常; 发生这种情况时，可能已将未指定数量的字节写入底层文件。试图腌制一个高度递归的数据结构可能会超过最大递归深度，RecursionError 在这种情况下会引发一次。你可以谨慎地提高这个限制sys.setrecursionlimit()。

请注意，函数（内置的和用户定义的）由“完全限定”名称引用进行挑选，而不是按值进行。[2] 这意味着只有函数名称被pickle，以及定义该函数的模块的名称。函数的代码及其任何函数属性都不被pickle。因此，定义模块必须可以在取消环境中导入，并且模块必须包含指定的对象，否则将引发异常。[3]

同样，类按名称引用进行挑选，因此在取消环境中适用相同的限制。请注意，没有任何类的代码或数据被腌制，因此在下面的示例中，attr不会在unpickling环境中恢复class属性：

```
class Foo:
    attr = 'A class attribute'

picklestring = pickle.dumps(Foo)
```

这些限制是为什么必须在模块的顶层定义可调用的函数和类。

同样，当类实例被腌制时，他们的类的代码和数据不会随着它们一起被腌制。只有实例数据被腌制。这是有意完成的，因此您可以修复类中的错误或向类中添加方法，并仍然加载使用该类的早期版本创建的对象。如果您计划使用能够看到许多版本的类的长效对象，则可能需要在对象中添加版本号，以便可以通过类的__setstate__()方法进行适当的转换。

12.1.5。酸洗类实例

在本节中，我们将描述可用于定义，定制和控制如何对实例进行pickle和unpickled的一般机制。

在大多数情况下，不需要额外的代码来使实例可供选择。默认情况下，pickle将通过内省检索实例的类和属性。当一个类实例是unpickled时，它的__init__()方法通常*不会*被调用。默认行为

首先创建一个未初始化的实例，然后恢复保存的属性。以下代码显示了此行为的实现：

```
def save(obj):
    return (obj.__class__, obj.__dict__)

def load(cls, attributes):
    obj = cls.__new__(cls)
    obj.__dict__.update(attributes)
    return obj
```

类可以通过提供一种或多种特殊方法来改变默认行为：

object.__getnewargs_ex__()

在协议2和更新版本中，实现该__getnewargs_ex__()方法的类 可以指定__new__()在取消时传递给方法的值 。该方法必须返回一个对 ，其中*args*是位置参数的元组，并且*kwargs*是用于构造该对象的命名参数的字典。这些将在拆除时传递给方法。(args, kwargs) __new__()

如果__new__()您的类的方法需要关键字参数，则应该实现此方法。否则，建议实现兼容性__getnewargs__()。

*在版本3.6中更改：*__getnewargs_ex__()现在在协议2和3中使用。

object.__getnewargs__()

这个方法与其相似__getnewargs_ex__()，但只支持位置参数。它必须返回一个参数元组，这个参数元组 在拆除时args将传递给该__new__()方法。

__getnewargs__()如果__getnewargs_ex__()被定义，将不会被调用。

*在版本3.6中更改：*在Python 3.6之前，__getnewargs__()被调用而不是__getnewargs_ex__()在协议2和3中。

object.__getstate__()

课程可以进一步影响他们的实例如何腌制; 如果类定义了该方法__getstate__()，则会调用该方法，并将返回的对象作为实例的内容进行挑选，而不是实例字典的内容。如果__getstate__()方法不存在，那么实例__dict__就像往常一样腌制。

object.__setstate__(*状态*)

在取消订单时，如果类定义__setstate__()，它将被调用与unpickled状态。在这种情况下，状态对象不需要是字典。否则，pickled状态必须是一个字典，并且其项目被分配给新实例的字典。

> **注意：** 如果__getstate__()返回一个错误值，该__setstate__()方法在取消时不会被调用。

参考部分处理状态的对象有关如何使用方法的详细信息__getstate__()和__setstate__()。

> **注意：** 在在unpickle时，一些方法，如__getattr__()，__getattribute__()或__setattr__()可在该实例调用。如果这些方法依赖于某种内部不变是真实的，那么该类型应

该实现__getnewargs__()或__getnewargs_ex__()建立这样的不变量; 否则，既__new__()不会也__init__()不会被调用。

正如我们将看到的，腌菜不直接使用上述方法。实际上，这些方法是实现__reduce__()特殊方法的复制协议的一部分 。复制协议为检索酸洗和复制对象所需的数据提供了一个统一的界面。[4]

虽然功能强大，但__reduce__()直接在类中实现却容易出错。出于这个原因，类设计者应尽可能使用高级接口（即__getnewargs_ex__()，__getstate__()和 __setstate__()）。但是，我们将展示使用__reduce__()是唯一的选择或导致更高效的酸洗或两者的情况。

object.__reduce__()

界面目前定义如下。该__reduce__()方法不带任何参数，并返回一个字符串或最好是一个元组（返回的对象通常被称为“减少值”）。

如果返回一个字符串，则应将该字符串解释为全局变量的名称。它应该是相对于其模块的对象的本地名称; pickle模块搜索模块名称空间以确定对象的模块。这种行为通常对单身人士有用。

当一个元组返回时，它必须在两到五个项目之间。可选项可以省略，None也可以作为其值提供。每个项目的语义是按顺序的：

- 可调用的对象，将被调用来创建该对象的初始版本。
- 可调用对象的参数元组。如果可调用对象不接受任何参数，则必须给出一个空元组。
- 可选地，该对象的状态将被传递给对象的 __setstate__()方法，如前所述。如果对象没有这种方法，那么该值必须是一个字典，并且它将被添加到该对象的__dict__属性中。
- 可选地，迭代器（而不是序列）产生连续的项目。这些项目将被追加到对象，使用obj.append(item)或批量使用obj.extend(list_of_items)。这主要用于列表子类，但可以由其他类使用，只要它们具有相应的签名append()并且extend()具有适当的签名方法。（无论append()或extend()使用取决于哪泡菜协议版本被用作以及项目追加的次数，所以两者都必须被支持。）
- 可选地，迭代器（不是序列）产生连续的键值对。这些项目将被存储到使用的对象。这主要用于字典子类，但只要它们实现，可以由其他类使用。obj[key] = value__setitem__()

object.__reduce_ex__(*协议*)

或者，__reduce_ex__()可以定义一种方法。唯一的区别是这个方法应该采用一个整数参数，协议版本。定义后，pickle会优先于__reduce__()方法。另外，__reduce__()自动成为扩展版本的同义词。此方法的主要用途是为旧版Python发行版提供向后兼容的减少值。

12.1.5.1。外部对象的持久性

为了获得对象持久性，pickle模块支持对pickle数据流之外的对象的引用的概念。这样的目的是通过一个永久ID，这应该是一个字母数字字符（协议0）字符串中引用[5]或只是任意对象（为任何较新协议）。

这种持久性ID的解析不是由pickle模块定义的; 它将委托此分辨率对皮克勒和Unpickler会，用户定义的方法persistent_id()和 persistent_load()分别。

要pickle具有外部持久性id的对象，picker必须有一个自定义persistent_id()方法，它将一个对象作为参数，并返回None该对象的持久性id或该持久性id。当None返回时，只需皮克勒泡菜对象为正常。当返回一个持久化的ID字符串时，pickler将pickle这个对象和一个标记，这样unpickler就会将它识别为一个持久化的ID。

要取消对外部对象的取消操作，unpickler必须具有一个自定义 persistent_load()方法，该方法采用持久ID对象并返回引用的对象。

下面是一个综合示例，介绍如何使用持久标识来引用外部对象。

```
# Simple example presenting how persistent ID can be used to pickle
# external objects by reference.

import pickle
import sqlite3
from collections import namedtuple

# Simple class representing a record in our database.
MemoRecord = namedtuple("MemoRecord", "key, task")

class DBPickler(pickle.Pickler):

    def persistent_id(self, obj):
        # Instead of pickling MemoRecord as a regular class instance, we emit a
        # persistent ID.
        if isinstance(obj, MemoRecord):
            # Here, our persistent ID is simply a tuple, containing a tag and a
            # key, which refers to a specific record in the database.
            return ("MemoRecord", obj.key)
        else:
            # If obj does not have a persistent ID, return None. This means obj
            # needs to be pickled as usual.
            return None


class DBUnpickler(pickle.Unpickler):

    def __init__(self, file, connection):
        super().__init__(file)
        self.connection = connection

    def persistent_load(self, pid):
        # This method is invoked whenever a persistent ID is encountered.
        # Here, pid is the tuple returned by DBPickler.
        cursor = self.connection.cursor()
        type_tag, key_id = pid
        if type_tag == "MemoRecord":
            # Fetch the referenced record from the database and return it.
            cursor.execute("SELECT * FROM memos WHERE key=?", (str(key_id),))
            key, task = cursor.fetchone()
            return MemoRecord(key, task)
        else:
            # Always raises an error if you cannot return the correct object.
            # Otherwise, the unpickler will think None is the object referenced
            # by the persistent ID.
            raise pickle.UnpicklingError("unsupported persistent object")
```

```
def main():
    import io
    import pprint

    # Initialize and populate our database.
    conn = sqlite3.connect(":memory:")
    cursor = conn.cursor()
    cursor.execute("CREATE TABLE memos(key INTEGER PRIMARY KEY, task TEXT)")
    tasks = (
        'give food to fish',
        'prepare group meeting',
        'fight with a zebra',
        )
    for task in tasks:
        cursor.execute("INSERT INTO memos VALUES(NULL, ?)", (task,))

    # Fetch the records to be pickled.
    cursor.execute("SELECT * FROM memos")
    memos = [MemoRecord(key, task) for key, task in cursor]
    # Save the records using our custom DBPickler.
    file = io.BytesIO()
    DBPickler(file).dump(memos)

    print("Pickled records:")
    pprint.pprint(memos)

    # Update a record, just for good measure.
    cursor.execute("UPDATE memos SET task='learn italian' WHERE key=1")

    # Load the records from the pickle data stream.
    file.seek(0)
    memos = DBUnpickler(file, conn).load()

    print("Unpickled records:")
    pprint.pprint(memos)


if __name__ == '__main__':
    main()
```

12.1.5.2。调度表

如果想要自定义一些类的酸洗而不干扰依赖酸洗的其他代码，那么可以创建一个带有私有调度表的pickler。

由该copyreg模块管理的全局调度表可用copyreg.dispatch_table。因此，可以选择使用修改副本copyreg.dispatch_table作为私人调度表。

例如

```
f = io.BytesIO()
p = pickle.Pickler(f)
```

```
p.dispatch_table = copyreg.dispatch_table.copy()
p.dispatch_table[SomeClass] = reduce_SomeClass
```

创建一个专门pickle.Pickler处理SomeClass该类的私有调度表的实例。或者，代码

```
class MyPickler(pickle.Pickler):
    dispatch_table = copyreg.dispatch_table.copy()
    dispatch_table[SomeClass] = reduce_SomeClass
f = io.BytesIO()
p = MyPickler(f)
```

也是一样的，但MyPickler默认情况下所有的实例共享相同的调度表。使用该copyreg模块的等效代码 是

```
copyreg.pickle(SomeClass, reduce_SomeClass)
f = io.BytesIO()
p = pickle.Pickler(f)
```

12.1.5.3。处理有状态对象

下面是一个示例，显示如何修改一个类的酸洗行为。本TextReader类打开一个文本文件，并返回每一次它的行号和行内容，readline()方法被调用。如果一个 TextReader实例被腌制，*除*文件对象成员*之外*的所有属性*都*将被保存。当实例取消选中时，将重新打开该文件，并从最后一个位置继续读取。该__setstate__()和 __getstate__()方法来实现此行为。

```
class TextReader:
    """Print and number lines in a text file."""

    def __init__(self, filename):
        self.filename = filename
        self.file = open(filename)
        self.lineno = 0

    def readline(self):
        self.lineno += 1
        line = self.file.readline()
        if not line:
            return None
        if line.endswith('\n'):
            line = line[:-1]
        return "%i: %s" % (self.lineno, line)

    def __getstate__(self):
        # Copy the object's state from self.__dict__ which contains
        # all our instance attributes. Always use the dict.copy()
        # method to avoid modifying the original state.
        state = self.__dict__.copy()
        # Remove the unpicklable entries.
        del state['file']
        return state

    def __setstate__(self, state):
        # Restore instance attributes (i.e., filename and lineno).
```

```
        self.__dict__.update(state)
        # Restore the previously opened file's state. To do so, we need to
        # reopen it and read from it until the line count is restored.
        file = open(self.filename)
        for _ in range(self.lineno):
            file.readline()
        # Finally, save the file.
        self.file = file
```

示例用法可能如下所示：

```
>>> reader = TextReader("hello.txt")
>>> reader.readline()
'1: Hello world!'
>>> reader.readline()
'2: I am line number two.'
>>> new_reader = pickle.loads(pickle.dumps(reader))
>>> new_reader.readline()
'3: Goodbye!'
```

12.1.6。限制全局

默认情况下，unpickling会导入它在pickle数据中找到的任何类或函数。对于许多应用程序来说，这种行为是不可接受的，因为它允许unpickler导入和调用任意代码。只要考虑一下这个手工制作的pickle数据流在加载时会做什么：

```
>>> import pickle
>>> pickle.loads(b"cos\nsystem\n(S'echo hello world'\ntR.")
hello world
0
```

在这个例子中，unpickler导入`os.system()`函数，然后应用字符串参数“echo hello world”。虽然这个例子是无害的，但不难想象一个可能会损坏你的系统的例子。

出于这个原因，您可能想要控制通过自定义取消选中的内容 `Unpickler.find_class()`。与其名称暗示的不同， `Unpickler.find_class()`只要请求全局（即类或函数）就会被调用。因此可以完全禁止全局变量或将它们限制为安全的子集。

下面是一个unpickler的例子，它只允许`builtins`加载模块中的几个安全类 ：

```
import builtins
import io
import pickle

safe_builtins = {
    'range',
    'complex',
    'set',
    'frozenset',
    'slice',
}

class RestrictedUnpickler(pickle.Unpickler):
```

```
    def find_class(self, module, name):
        # Only allow safe classes from builtins.
        if module == "builtins" and name in safe_builtins:
            return getattr(builtins, name)
        # Forbid everything else.
        raise pickle.UnpicklingError("global '%s.%s' is forbidden" %
                                     (module, name))

def restricted_loads(s):
    """Helper function analogous to pickle.loads()."""
    return RestrictedUnpickler(io.BytesIO(s)).load()
```

我们的unpickler工作的一个示例使用意图是：

```
>>> restricted_loads(pickle.dumps([1, 2, range(15)]))
[1, 2, range(0, 15)]
>>> restricted_loads(b"cos\nsystem\n(S'echo hello world'\ntR.")
Traceback (most recent call last):
  ...
pickle.UnpicklingError: global 'os.system' is forbidden
>>> restricted_loads(b'cbuiltins\neval\n'
...                  b'(S\'getattr(__import__("os"), "system")'
...                  b'("echo hello world")\'\ntR.')
Traceback (most recent call last):
  ...
pickle.UnpicklingError: global 'builtins.eval' is forbidden
```

正如我们的示例所示，您必须小心谨慎，以免被取消。因此，如果担心安全问题，您可能需要考虑替代方案，例如xmlrpc.client第三方解决方案中的编组API 。

12.1.7。性能

最近版本的pickle协议（来自协议2及以上版本）以几种常见功能和内置类型的高效二进制编码为特色。此外，该pickle模块具有用C编写的透明优化器。

12.1.8。示例

对于最简单的代码，使用dump()和load()函数。

```
import pickle

# An arbitrary collection of objects supported by pickle.
data = {
    'a': [1, 2.0, 3, 4+6j],
    'b': ("character string", b"byte string"),
    'c': {None, True, False}
}

with open('data.pickle', 'wb') as f:
    # Pickle the 'data' dictionary using the highest protocol available.
    pickle.dump(data, f, pickle.HIGHEST_PROTOCOL)
```

以下示例读取所产生的腌制数据。

```
import pickle

with open('data.pickle', 'rb') as f:
    # The protocol version used is detected automatically, so we do not
    # have to specify it.
    data = pickle.load(f)
```

也可以看看:

模 copyreg
　　Pickle接口构造函数注册用于扩展类型。

模 pickletools
　　用于处理和分析腌制数据的工具。

模 shelve
　　对象的索引数据库; 用途pickle。

模 copy
　　浅而深的对象复制。

模 marshal
　　内置类型的高性能序列化。

脚注

[1] 不要将此与marshal模块混淆

[2] 这就是为什么lambda不能腌制功能的原因：所有 lambda功能都有相同的名称：<lambda>。

[3] 提出的例外可能是一个ImportError或一个， AttributeError但它可能是其他的东西。

[4] 该copy模块使用此协议进行浅层和深层复制操作。

[5] 对于现有的限制是由所产生的酸菜和实例序列分隔。因此，如果在持久性ID

12.2。copyreg- 注册pickle支持功能

源代码： Lib / copyreg.py

该copyreg模块提供了一种定义酸洗特定对象时使用的函数的方法。的pickle和copy酸洗/复制这些对象时模块使用这些功能。该模块提供有关不是类的对象构造函数的配置信息。这样的构造函数可能是工厂函数或类实例。

copyreg.**constructor**（*object*）

声明*对象*是一个有效的构造函数。如果*对象*不可调用（因此无法用作构造函数），则引发TypeError。

copyreg.**pickle**（*类型*，*函数*，*构造函数=无*）

声明该*函数*应该用作类型*类型的*对象的“减少”函数。 *函数*应返回包含两个或三个元素的字符串或元组。

可选的*构造函数*参数（如果提供的话）是一个可调用的对象，它可以在pickling时用*函数*返回的参数元组调用时重建对象。 TypeError如果*object*是一个类或者*构造函数*不可调用，将会引发它 。

有关*函数*和*构造函数*pickle的接口的更多详细信息，请参阅模块。请注意，pickler对象或其子类的 属性也可用于声明约简函数。dispatch_tablepickle.Pickler

12.2.1。示例

下面的例子想展示如何注册一个pickle函数以及如何使用它：

```
>>> import copyreg, copy, pickle
>>> class C(object):
...     def __init__(self, a):
...         self.a = a
...
>>> def pickle_c(c):
...     print("pickling a C instance...")
...     return C, (c.a,)
...
>>> copyreg.pickle(C, pickle_c)
>>> c = C(1)
>>> d = copy.copy(c)
pickling a C instance...
>>> p = pickle.dumps(c)
pickling a C instance...
```

12.3。 shelve- Python对象持久性

源代码： Lib / shelve.py

“货架”是一个持久的，类似字典的对象。与“dbm”数据库的不同之处在于，货架上的值（不是关键字！）本质上可以是任意的Python对象 - pickle模块可以处理的任何东西。这包括大多数类实例，递归数据类型和包含大量共享子对象的对象。键是普通的字符串。

shelve.open（*filename*，*flag ='c'*，*protocol = None*，*writeback = False* ）

打开一个持久字典。指定的文件名是底层数据库的基本文件名。作为副作用，可以将扩展名添加到文件名中，并且可以创建多个文件。默认情况下，打开底层数据库文件以进行读取和写入。可选的*标志*参数与*标志*参数的含义相同dbm.open()。

默认情况下，版本3酱菜用于序列化值。pickle协议的版本可以用*协议*参数指定。

由于Python的语义，一个书架无法知道何时修改了可变的持久字典条目。默认情况下，修改后的对象*仅*在分配给货架*时才*写入（请参阅示例）。如果可选*写回*参数设置为True，访问的所有项目也缓存在内存中，并在写回sync()和 close()；这可以更容易地修改持久性字典中的可变条目，但是，如果访问了很多条目，它可能会消耗大量的缓存内存，并且它可以使关闭操作非常缓慢，因为所有访问的条目都被写回（没有办法确定哪些访问条目是可变的，哪些实际上是变异的）。

> **注意：** 不要依赖自动关闭的搁板；总是close()在你不再需要的时候明确调用 ，或者shelve.open()用作上下文管理器：
>
> ```
> with shelve.open('spam') as db:
>     db['eggs'] = 'eggs'
> ```

警告： 由于该shelve模块由后台支持pickle，因此从不受信任的源加载架子是不安全的。像pickle一样，加载一个架子可以执行任意代码。

货架对象支持字典支持的所有方法。这简化了从基于字典的脚本到需要持久存储的脚本的过渡。

还支持另外两种方法：

Shelf.sync（）

如果搁架已打开且*写回*设置为，则将高速缓存中的所有条目*写回*True。如果可行，还清空缓存并同步磁盘上的持久字典。这在货架关闭时自动调用close()。

Shelf.close（）

同步并关闭持久性*字典*对象。在一个封闭的架子上操作将失败，一个ValueError。

> **也可以看看：** 持久的字典配方 ，广泛支持存储格式，并具有原生字典的速度。

12.3.1。限制

- 选择使用哪个数据库包（如dbm.ndbm或 dbm.gnu ）取决于哪个接口可用。因此直接使用打开数据库是不安全的dbm。数据库也（不幸）受限于dbm（如果使用的话）- 这意味着存储在数据库中的对象（腌制表示）应该相当小，并且在极少数情况下，重要的冲突可能会导致数据库拒绝更新。
- 该shelve模块不支持对搁置对象的*并发*读取/写入访问。（多个同时读取访问是安全的。）当一个程序有一个可写入的架子时，其他程序不应该打开它来读取或写入。Unix文件锁定可以用来解决这个问题，但是这在Unix版本中是不同的，并且需要关于所使用的数据库实现的知识。

class shelve.Shelf（*dict*，*protocol = None*，*writeback = False*，*keyencoding ='utf-8'*）

其中的一个子类collections.abc.MutableMapping在*字典*对象中存储pickle值。

默认情况下，版本3酱菜用于序列化值。pickle协议的版本可以用*协议*参数指定。请参阅pickle文档以了解有关咸菜协议的讨论。

如果*写回*参数是True，对象将保存所有访问条目的缓存，并在同步和关闭时间将它们写回*字典*。这允许对可变条目进行自然操作，但可能消耗更多内存并使同步和关闭需要很长时间。

所述*keyencoding*参数是用于它们与下面的字典使用之前进行编码密钥的编码。

一个Shelf对象也可以用作上下文管理器，在这种情况下，它会在with块结束时自动关闭。

*在版本3.2中更改：*添加了*keyencoding*参数; 以前，密钥始终以UTF-8编码。

*在版本3.4中更改：*添加了上下文管理器支持。

class shelve.BsdDbShelf（*dict*，*protocol = None*，*writeback = False*，*keyencoding ='utf-8'*）

子类的Shelf暴露first()，next()， previous()，last()和set_location()是第三方适用于哪些bsddb从模块pybsddb而不是在其他的数据库模块。传递给构造函数的*dict*对象必须支持这些方法。这通常是通过调用一个完成的 bsddb.hashopen()，bsddb.btopen()或bsddb.rnopen()。可选*协议*，*回写*和*键编码*参数与*该类*具有相同的解释Shelf。

class shelve.DbfilenameShelf（*filename*，*flag ='c'*，*protocol = None*，*writeback = False*）

它的一个子类Shelf接受一个*文件名*而不是一个类似dict的对象。底层文件将使用打开dbm.open()。默认情况下，该文件将被创建并打开以供读取和写入。可选的*标志*参数与该open()功能具有相同的解释。可选的*协议*和*写回*参数与Shelf该类具有相同的解释。

12.3.2。示例

总结界面（key是一个字符串，data是一个任意的对象）：

```
import shelve
```

```
d = shelve.open(filename)  # open -- file may get suffix added by low-level
                           # library

d[key] = data              # store data at key (overwrites old data if
                           # using an existing key)
data = d[key]              # retrieve a COPY of data at key (raise KeyError
                           # if no such key)
del d[key]                 # delete data stored at key (raises KeyError
                           # if no such key)

flag = key in d            # true if the key exists
klist = list(d.keys())     # a list of all existing keys (slow!)

# as d was opened WITHOUT writeback=True, beware:
d['xx'] = [0, 1, 2]        # this works as expected, but...
d['xx'].append(3)          # *this doesn't!* -- d['xx'] is STILL [0, 1, 2]!

# having opened d without writeback=True, you need to code carefully:
temp = d['xx']             # extracts the copy
temp.append(5)             # mutates the copy
d['xx'] = temp             # stores the copy right back, to persist it

# or, d=shelve.open(filename,writeback=True) would let you just code
# d['xx'].append(5) and have it work as expected, BUT it would also
# consume more memory and make the d.close() operation slower.

d.close()                  # close it
```

也可以看看:

模 dbm
通用接口dbm风格的数据库。

模 pickle
对象序列化使用shelve。

12.4。 marshal- 内部Python对象序列化

该模块包含可以以二进制格式读取和写入Python值的函数。该格式是针对Python的，但与机器体系结构问题无关（例如，您可以将Python值写入PC上的文件，将文件传输到Sun，然后再读回到那里）。该格式的细节是故意没有记录的; 它可能会在Python版本之间改变（尽管它很少）。[1]

这不是一个通用的“持久性”模块。对于通过RPC调用的Python对象的一般持久性和传输，请参阅模块pickle和 shelve。该marshal模块主要用于支持读取和编写. pyc文件的Python模块的“伪编译”代码。因此，如果需要，Python维护者保留修改编组格式的权利，这些编码格式采用后向不兼容的方式。如果你正在序列化和反序列化Python对象，可以使用pickle模块 - 性能可比，版本独立性得到保证，并且pickle支持比marshal更广泛的对象。

警告： 该marshal模块并非旨在防止错误或恶意构建的数据。切勿解组从不受信任或未经身份验证的源收到的数据。

并非所有的Python对象都支持; 一般来说，只有对象的值独立于特定的Python调用才能被该模块写入和读取。支持以下类型：布尔值，整数，浮点数，复数，字符串，字节，字节数组，字符串，列表，集合，frozensets，字典和代码对象，其中应该理解元组，列表，集合，frozensets只有支持其中所包含的价值，才支持字典。在单身None ， Ellipsis并且StopIteration还可以编组和解组。对于小于3的格式*版本*，递归列表，集合和字典不能写入（见下文）。

有读取/写入文件的函数以及对类似字节的对象进行操作的函数。

该模块定义了这些功能：

marshal. dump（*值*，*文件*[，*版本*]）

在打开的文件上写入值。该值必须是受支持的类型。该文件必须是可写入的二进制文件。

如果该值具有（或包含具有）不受支持类型的对象， ValueError则会引发异常 - 但垃圾数据也会写入该文件。该对象不会被正确地读回load()。

该*版本*的说法表示数据格式dump应使用（见下文）。

marshal. load（*文件*）

从打开的文件中读取一个值并返回。如果没有有效的读取值（例如，因为数据有不同的Python版本的不兼容元帅格式），提高EOFError ，ValueError或TypeError。该文件必须是可读的二进制文件。

注意： 如果包含不受支持的类型的对象被编组dump() ， load()将替代None解组类型。

marshal. dumps（*value*[，*version*]）

返回将被写入文件的字节对象。该值必须是受支持的类型。养异常，如果值具有（或包含具有的对象）不支持的类型。 dump(value, file)ValueError

该*版本*的说法表示数据格式`dumps`应使用（见下文）。

marshal.**loads**（*字节*）

将类似字节的对象转换为一个值。如果没有找到有效的价值，提高 `EOFError` ，`ValueError` 或`TypeError`。输入中的额外字节被忽略。

另外，定义了以下常量：

marshal.**version**

指示模块使用的格式。版本0是历史格式，版本1共享interned字符串，版本2使用浮点数字的二进制格式。版本3增加了对对象实例化和递归的支持。目前的版本是4。

脚注

[1] [illegible]

12.5。 dbm- 与Unix“数据库”的接口

源代码： Lib / dbm / __ init__.py

dbm是DBM数据库变体的通用接口 - dbm.gnu或者dbm.ndbm。如果没有安装这些模块，dbm.dumb则将使用模块中缓慢而简单的实现。Oracle Berkeley DB 有第三方接口。

*异常*dbm.error

包含可由每个受支持模块引发的异常的元组，其中包含也dbm.error称为第一项的唯一异常- 后者在dbm.error引发时使用。

dbm.whichdb（*文件名*）

这个函数尝试猜测哪些可用的几个简单的数据库模块- dbm.gnu，dbm.ndbm或dbm.dumb-应该用来打开一个指定的文件。

返回以下值之一：None如果文件由于不可读或不存在而无法打开; ''如果文件格式不能被猜测，则为空字符串（）; 或包含所需模块名称的字符串，如'dbm.ndbm'或'dbm.gnu'。

dbm.open（*file*，*flag ='r'*，*mode = 0o666*）

打开数据库文件*的文件*，并返回相应的对象。

如果数据库文件已经存在，whichdb()则使用该函数确定其类型并使用适当的模块; 如果它不存在，则使用上面列出的可导入的第一个模块。

可选的*标志*参数可以是：

值	含义
'r'	打开仅用于读取的现有数据库（默认）
'w'	打开现有的数据库进行读写
'c'	打开数据库进行读写，如果不存在则创建它
'n'	总是创建一个新的空的数据库，打开阅读和写作

可选*模式*参数是文件的Unix模式，仅在需要创建数据库时使用。它默认为八进制0o666（并且会被主要的umask修改）。

返回的对象open()支持与字典相同的基本功能; 键和它们的对应的值可以被存储，检索和删除，并且in操作者和keys()方法是可用的，以及get()和setdefault()。

*改变在3.2版本：*get()和setdefault()现在在所有数据库模块中可用。

键和值始终以字节形式存储。这意味着当使用字符串时，它们在被存储之前被隐式转换为默认编码。

这些对象还支持在with声明中使用，并在完成时自动关闭它们。

*在版本3.4中进行了更改：*为由上下文管理协议返回的对象添加了本机支持open()。

以下示例记录了一些主机名和相应的标题，然后打印出数据库的内容：

```
import dbm

# Open database, creating it if necessary.
with dbm.open('cache', 'c') as db:

    # Record some values
    db[b'hello'] = b'there'
    db['www.python.org'] = 'Python Website'
    db['www.cnn.com'] = 'Cable News Network'

    # Note that the keys are considered bytes now.
    assert db[b'www.python.org'] == b'Python Website'
    # Notice how the value is now in bytes.
    assert db['www.cnn.com'] == b'Cable News Network'

    # Often-used methods of the dict interface work too.
    print(db.get('python.org', b'not present'))

    # Storing a non-string key or value will raise an exception (most
    # likely a TypeError).
    db['www.yahoo.com'] = 4

# db is automatically closed when leaving the with statement.
```

也可以看看:

模 shelve
存储非字符串数据的持久性模块。

以下各节介绍各个子模块。

12.5.1。 dbm.gnu- DBM的GNU的重新解释

源代码： Lib / dbm / gnu.py

该模块与模块非常相似dbm，但使用GNU库 gdbm来提供一些附加功能。请注意，由dbm.gnu和创建的文件格式dbm.ndbm不兼容。

该dbm.gnu模块为GNU DBM库提供了一个接口。 dbm.gnu.gdbm对象的行为就像映射（字典），除了在存储之前键和值总是转换为字节。打印gdbm 对象不会打印键和值，并且不支持items()和values()方法。

*异常*dbm.gnu.error
引发dbm.gnu特定的错误，例如I / O错误。 KeyError引发一般映射错误，如指定不正确的键。

dbm.gnu.open（*filename* [，*flag* [，*mode*]] ）
打开一个gdbm数据库并返回一个gdbm对象。该*文件名* 参数是数据库文件的名称。

可选的*标志*参数可以是：

值	含义
'r'	打开仅用于读取的现有数据库（默认）
'w'	打开现有的数据库进行读写
'c'	打开数据库进行读写，如果不存在则创建它
'n'	总是创建一个新的空的数据库，打开阅读和写作

以下附加字符可以附加到该标志以控制数据库的打开方式：

值	含义
'f'	以快速模式打开数据库。写入数据库将不会同步。
's'	同步模式。这将导致数据库的更改立即写入文件。
'u'	不要锁定数据库。

并非所有标志对于所有版本都有效gdbm。模块常量 open_flags是一串支持的标志字符。error如果指定了无效标志，则会引发异常 。

可选*模式*参数是文件的Unix模式，仅在需要创建数据库时使用。它默认为八进制0o666。

除了类似字典的方法外，gdbm对象还有以下方法：

gdbm.**firstkey**( )
: 使用此方法和nextkey()方法可以遍历数据库中的每个键 。遍历按gdbm内部散列值排序，不会按键值排序。此方法返回开始键。

gdbm.**nextkey**(*重点*)
: 返回遍历中*关键字的键*。以下代码打印数据库中的每个键db ，而不必在内存中创建一个包含它们的列表：

```
k = db.firstkey()
while k != None:
    print(k)
    k = db.nextkey(k)
```

gdbm.**reorganize**( )
: 如果您执行了大量删除操作并希望缩小gdbm文件使用的空间，则此例程将重新组织数据库。 gdbm 对象不会缩短数据库文件的长度，除非使用此重组; 否则，删除的文件空间将被保留并在新（键值）对添加时重新使用。

gdbm.**sync**( )
: 当数据库以快速模式打开时，此方法强制任何未写入的数据写入磁盘。

gdbm.**close**( )

关闭gdbm数据库。

12.5.2。 dbm.ndbm- 基于ndbm的接口

源代码： Lib / dbm / ndbm.py

该dbm.ndbm模块为Unix“（n）dbm”库提供了一个接口。Dbm对象的行为与映射（字典）相似，只是键和值始终以字节形式存储。打印dbm对象不会打印键和值，并且不支持items()和values()方法。

该模块可以与“经典”ndbm接口或GNU GDBM兼容性接口一起使用。在Unix上，**配置**脚本将尝试找到合适的头文件来简化构建这个模块。

*异常*dbm.ndbm.error
引发dbm.ndbm特定的错误，例如I / O错误。KeyError引发一般映射错误，如指定不正确的键。

dbm.ndbm.library
ndbm使用的实现库的名称。

dbm.ndbm.open（*filename*[，*flag*[，*mode*]]）
打开一个dbm数据库并返回一个ndbm对象。该*文件名*参数是数据库文件名（不带.dir或.pag扩展）。

可选的*标志*参数必须是以下值之一：

值	含义
'r'	打开仅用于读取的现有数据库（默认）
'w'	打开现有的数据库进行读写
'c'	打开数据库进行读写，如果不存在则创建它
'n'	总是创建一个新的空的数据库，打开阅读和写作

可选*模式*参数是文件的Unix模式，仅在需要创建数据库时使用。它默认为八进制0o666（并且会被主要的umask修改）。

除了类似字典的方法之外，ndbm对象还提供以下方法：

ndbm.close（）
关闭ndbm数据库。

12.5.3。 dbm.dumb- 便携式DBM实施

源代码： Lib / dbm / dumb.py

注意： 该dbm.dumb模块的目的是作为最后的手段后备的 dbm模块时，一个更强大的模块不可用。该dbm.dumb 模块不是为了速度而编写的，并没有像其他数据库模块那样大量使用。

该dbm.dumb模块提供了一个完全用Python编写的持久字典式界面。不像其他模块，例如dbm.gnu不需要外部库。与其他持久映射一样，键和值始终以字节形式存储。

该模块定义了以下内容：

*异常*dbm.dumb.**error**

引发dbm.dumb特定的错误，例如I / O错误。 KeyError引发一般映射错误，如指定不正确的键。

dbm.dumb.**open**(*filename* [, *flag* [, *mode*]])

打开一个dumbdbm数据库并返回一个dumbdbm对象。该*文件名*参数是数据库文件的基本名称（没有任何具体的扩展）。当创建一个数据库dumbdbm，用文件.dat和.dir扩展名被创建。

可选的*标志*参数仅支持语义'c'和'n'值。其他值将默认为始终打开以进行更新的数据库，并且如果该数据库不存在，则会创建它。

可选*模式*参数是文件的Unix模式，仅在需要创建数据库时使用。它默认为八进制0o666（并且会被主要的umask修改）。

在版本3.5中更改： open()标志具有该值时总是创建一个新的数据库'n'。

从版本3.6开始弃用，将在版本3.8中删除： 创建数据库入'r'和'w'模式。在'r'模式下修改数据库 。

除了 collections.abc.MutableMapping该类dumbdbm提供的方法之外，对象还提供以下方法：

dumbdbm.**sync**( )

同步磁盘上的目录和数据文件。该方法由该Shelve.sync()方法调用。

dumbdbm.**close**( )

关闭dumbdbm数据库。

12.6。 sqlite3- 用于SQLite数据库的DB-API 2.0接口

源代码： Lib / sqlite3 /

SQLite是一个C库，它提供了一个轻量级的基于磁盘的数据库，它不需要单独的服务器进程，并允许使用SQL查询语言的非标准变体访问数据库。一些应用程序可以使用SQLite进行内部数据存储。也可以使用SQLite对应用程序进行原型设计，然后将代码移植到更大的数据库，如PostgreSQL或Oracle。

sqlite3模块由GerhardHäring编写。它提供了一个符合上述DB-API 2.0规范的SQL接口**PEP 249**。

要使用该模块，您必须首先创建一个Connection代表数据库的对象。这里的数据将被存储在example.db文件中：

```
import sqlite3
conn = sqlite3.connect('example.db')
```

您还可以提供特殊名称:memory:以在RAM中创建数据库。

一旦你有了Connection ，你可以创建一个Cursor 对象并调用它的execute()方法来执行SQL命令：

```
c = conn.cursor()

# Create table
c.execute('''CREATE TABLE stocks
             (date text, trans text, symbol text, qty real, price real)''')

# Insert a row of data
c.execute("INSERT INTO stocks VALUES ('2006-01-05','BUY','RHAT',100,35.14)")

# Save (commit) the changes
conn.commit()

# We can also close the connection if we are done with it.
# Just be sure any changes have been committed or they will be lost.
conn.close()
```

您保存的数据是持久性数据，可在以后的会话中使用：

```
import sqlite3
conn = sqlite3.connect('example.db')
c = conn.cursor()
```

通常你的SQL操作需要使用Python变量的值。你不应该使用Python的字符串操作来组装你的查询，因为这样做是不安全的; 它会使您的程序容易受到SQL注入攻击（有关可能出错的幽默示例，请参阅https://xkcd.com/327/ ）。

而是使用DB-API的参数替换。将?作为占位符，无论你想使用的值，然后提供值的元组作为第二个参数光标的execute()方法。（其他数据库模块可能使用不同的占位符，例如%s或:1）。例如：

```
# Never do this -- insecure!
symbol = 'RHAT'
c.execute("SELECT * FROM stocks WHERE symbol = '%s'" % symbol)

# Do this instead
t = ('RHAT',)
c.execute('SELECT * FROM stocks WHERE symbol=?', t)
print(c.fetchone())

# Larger example that inserts many records at a time
purchases = [('2006-03-28', 'BUY', 'IBM', 1000, 45.00),
             ('2006-04-05', 'BUY', 'MSFT', 1000, 72.00),
             ('2006-04-06', 'SELL', 'IBM', 500, 53.00),
            ]
c.executemany('INSERT INTO stocks VALUES (?,?,?,?,?)', purchases)
```

要在执行SELECT语句后检索数据，可以将游标作为迭代器，调用游标的fetchone()方法来检索单个匹配的行，或者调用fetchall()获取匹配行的列表。

这个例子使用迭代器形式：

```
>>> for row in c.execute('SELECT * FROM stocks ORDER BY price'):
        print(row)

('2006-01-05', 'BUY', 'RHAT', 100, 35.14)
('2006-03-28', 'BUY', 'IBM', 1000, 45.0)
('2006-04-06', 'SELL', 'IBM', 500, 53.0)
('2006-04-05', 'BUY', 'MSFT', 1000, 72.0)
```

也可以看看:

https://github.com/ghaering/pysqlite
pysqlite网页 - sqlite3是在名为“pysqlite”的外部开发的。

https://www.sqlite.org
SQLite网页; 该文档描述了支持的SQL方言的语法和可用数据类型。

http://www.w3schools.com/sql/
学习SQL语法的教程，参考和示例。

PEP 249 - 数据库API规范2.0
PEP由Marc-AndréLemburg撰写。

12.6.1。模块函数和常量

sqlite3.**version**
此模块的版本号，作为字符串。这不是SQLite库的版本。

sqlite3.version_info
该模块的版本号，作为整数的元组。这不是SQLite库的版本。

sqlite3.sqlite_version
运行时SQLite库的版本号，作为字符串。

sqlite3.sqlite_version_info
运行时SQLite库的版本号，作为整数的元组。

sqlite3.PARSE_DECLTYPES
该常数用于与函数的*detect_types*参数 一起使用connect()。

设置它会使sqlite3模块解析它返回的每一列的声明类型。它会解析出声明类型的第一个单词，即对于“整数主键”，它将解析出“整数”，或者对于“编号（10）”，它将解析出“编号”。然后，对于该列，它将查看转换器字典并使用在那里注册的转换器函数。

sqlite3.PARSE_COLNAMES
该常数用于与函数的*detect_types*参数 一起使用connect()。

设置这使得SQLite接口解析它返回的每一列的列名。它会查找一个形成[mytype]的字符串，然后决定'mytype'是列的类型。它会尝试在转换器字典中找到'mytype'条目，然后使用在那里找到的转换器函数返回值。找到的列名Cursor.description 只是列名的第一个单词，也就是说，如果您在SQL中使用类似的名称， 那么我们将解析出所有内容，直到列名称的第一个空格为止：列名称将简单地为“x”。’as "x [datetime]"’

sqlite3.connect（*database*[，*timeout*，*detect_types*，*isolation_level*，*check_same_thread*，*factory*，*cached_statements*，*uri*]）
打开到SQLite数据库文件*数据库*的连接。您可以使用":memory:"打开数据库连接到驻留在RAM而不是*磁盘*上的数据库。

当数据库被多个连接访问时，其中一个进程修改了数据库，SQLite数据库被锁定，直到该事务被提交。该*超时*参数指定连接应该多长时间等待锁消失，直到引发异常。超时参数的默认值是5.0（五秒）。

有关*isolation_level*参数，请参阅对象的 isolation_level属性Connection。

SQLite本机仅支持TEXT，INTEGER，REAL，BLOB和NULL类型。如果你想使用其他类型，你必须自己添加对它们的支持。该 *detect_types*参数和使用自定义**转换器**与模块级的注册register_converter()功能，让你轻松做到这一点。

*detect_types*默认为0（即关闭，没有类型检测），您可以将其设置为任意组合PARSE_DECLTYPES并PARSE_COLNAMES打开类型检测。

默认情况下，*check_same_thread*是True并且只有创建线程可以使用连接。如果设置False，则返回的连接可以在多个线程之间共享。当使用同一连接使用多个线程时，用户应该序列化写入操作以避免数据损坏。

默认情况下，sqlite3模块将其Connection类用于连接调用。但是，您可以继承这个Connection类，并 connect()通过为*工厂*参数提供您的类来使用您的类。

有关详细信息，请参阅本手册的SQLite和Python类型部分。

该sqlite3模块在内部使用一个语句缓存来避免SQL解析开销。如果要显式设置为连接缓存的语句数，可以设置*cached_statements*参数。目前实施的默认设置是缓存100条语句。

如果*uri*为true，则*数据库将*被解释为URI。这使您可以指定选项。例如，要以只读模式打开数据库，您可以使用：

```
db = sqlite3.connect('file:path/to/database?mode=ro', uri=True)
```

有关此功能的更多信息（包括可识别选项的列表）可以在SQLite URI文档中找到。

*在版本3.4中更改：*添加了*uri*参数。

sqlite3.**register_converter**(*typename*, *callable*)

注册一个可调用的字符串，将数据库中的字符串转换为自定义的Python类型。可调用将为类型为*typename*的所有数据库值调用。赋予参数*detect_types*中的connect()该类型检测是如何工作的功能。请注意，查询中*typename*和类型的名称必须匹配！

sqlite3.**register_adapter**(*type*, *callable*)

注册可调用以将自定义Python类型*类型*转换为SQLite支持的类型之一。可*调用*可*调用*可接受的Python值作为单个参数，并且必须返回以下类型的值：int，float，str或bytes。

sqlite3.**complete_statement**(*sql*)

返回True如果字符串*SQL*包含由分号终止一个或多个完整的SQL语句。它不验证SQL在语法上是否正确，只是没有未关闭的字符串文本，并且语句以分号结尾。

这可以用来为SQLite构建一个shell，如下例所示：

```
# A minimal SQLite shell for experiments

import sqlite3

con = sqlite3.connect(":memory:")
con.isolation_level = None
cur = con.cursor()

buffer = ""

print("Enter your SQL commands to execute in sqlite3.")
print("Enter a blank line to exit.")

while True:
    line = input()
    if line == "":
        break
    buffer += line
    if sqlite3.complete_statement(buffer):
        try:
            buffer = buffer.strip()
            cur.execute(buffer)

            if buffer.lstrip().upper().startswith("SELECT"):
```

```
                print(cur.fetchall())
            except sqlite3.Error as e:
                print("An error occurred:", e.args[0])
            buffer = ""

    con.close()
```

sqlite3.enable_callback_tracebacks（*标志*）

默认情况下，你不会在用户定义的函数，聚合，转换器，授权者回调等中获得任何回溯。如果你想调试它们，你可以调用此*标志*设置为的函数True。之后，你会从回调中获得回溯sys.stderr。用于False再次禁用该功能。

12.6.2。连接对象

*类*sqlite3.Connection

SQLite数据库连接具有以下属性和方法：

isolation_level

获取或设置当前的隔离级别。None用于自动提交模式或“延迟”，“立即”或“独占”之一。有关更详细的解释，请参阅控制交易部分 。

in_transaction

True如果一个事务处于活动状态（有未提交的更改），False否则。只读属性。

3.2版本中的新功能

cursor（*factory = Cursor*）

游标方法接受一个可选的参数*工厂*。如果提供，它必须是可返回的实例Cursor 或其子类。

commit（）

此方法提交当前事务。如果您不调用此方法，则自从上次调用以来执行的任何操作commit()都无法从其他数据库连接看到。如果您想知道为什么您没有看到您写入数据库的数据，请检查您是否忘记调用此方法。

rollback（）

此方法回滚自上次调用以来对数据库所做的任何更改 commit()。

close（）

这将关闭数据库连接。请注意，这不会自动调用commit()。如果您只是在不commit()先调用的情况下关闭数据库连接，则更改将会丢失！

execute（*sql*[，*parameters*]）

这是一个非标准的快捷方式，通过调用cursor()方法创建游标对象，execute()使用给定的*参数*调用游标的 方法，并返回游标。

executemany（*sql*[，*parameters*]）

这是一个非标准的快捷方式，通过调用cursor()方法创建游标对象，executemany()使用给定的*参数*调用游标的 方法，并返回游标。

executescript（*sql_script*）

这是一个非标准的快捷方式，它通过调用cursor()方法创建游标对象，executescript()使用给定的*sql_script*调用游标的 方法并返回游标。

create_function（*name*，*num_params*，*func*）

创建一个可以从SQL语句中以后使用下面的功能名称的用户定义函数*的名称*。*num_params*是函数接受的参数的数量（如果*num_params*是-1，函数可以接受任意数量的参数），*func*是一个可调用的Python，它被称为SQL函数。

该函数可以返回SQLite支持的任何类型：bytes，str，int，float和None。

例：

```
import sqlite3
import hashlib

def md5sum(t):
    return hashlib.md5(t).hexdigest()

con = sqlite3.connect(":memory:")
con.create_function("md5", 1, md5sum)
cur = con.cursor()
cur.execute("select md5(?)", (b"foo",))
print(cur.fetchone()[0])
```

create_aggregate（*name*，*num_params*，*aggregate_class*）

创建用户定义的聚合函数。

聚合类必须实现一个step方法，该方法接受参数*num_params*的数量（如果*num_params*是-1，该函数可以接受任意数量的参数）以及finalize将返回聚合的最终结果的方法。

该finalize方法可以返回SQLite支持的任何类型：bytes，str，int，float和None。

例：

```
import sqlite3

class MySum:
    def __init__(self):
        self.count = 0

    def step(self, value):
        self.count += value

    def finalize(self):
        return self.count

con = sqlite3.connect(":memory:")
con.create_aggregate("mysum", 1, MySum)
```

```
cur = con.cursor()
cur.execute("create table test(i)")
cur.execute("insert into test(i) values (1)")
cur.execute("insert into test(i) values (2)")
cur.execute("select mysum(i) from test")
print(cur.fetchone()[0])
```

create_collation (*名称*, *可调用*)

用指定*名称*创建一个排序规则并可*调用*。可调用将传递两个字符串参数。如果第一个的顺序低于第二个，则返回-1，如果顺序相等，则返回0;如果第一个的顺序高于第二个，则返回1。请注意，这会控制排序（SQL中的ORDER BY），因此您的比较不会影响其他SQL操作。

请注意，可调用函数将以Python字节串的形式获取其参数，通常以UTF-8编码。

以下示例显示了“错误方式”排序的自定义归类：

```
import sqlite3

def collate_reverse(string1, string2):
    if string1 == string2:
        return 0
    elif string1 < string2:
        return 1
    else:
        return -1

con = sqlite3.connect(":memory:")
con.create_collation("reverse", collate_reverse)

cur = con.cursor()
cur.execute("create table test(x)")
cur.executemany("insert into test(x) values (?)", [("a",), ("b",)])
cur.execute("select x from test order by x collate reverse")
for row in cur:
    print(row)
con.close()
```

要移除排序规则，请create_collation使用None可调用方式进行调用：

```
con.create_collation("reverse", None)
```

interrupt ( )

您可以从不同的线程调用此方法以中止可能在连接上执行的任何查询。该查询将中止，调用者将得到一个异常。

set_authorizer (*authorizer_callback*)

这个例程注册一个回调。每次尝试访问数据库中的一列表时，都会调用该回调。SQLITE_OK如果允许访问，SQLITE_DENY则应该返回回调 ，如果整个SQL语句应该中止并出现错误，SQLITE_IGNORE并且该列应被视为NULL值。这些常量在sqlite3模块中可用 。

回调的第一个参数表示要授权哪种操作。第二个和第三个参数将是参数或None 取决于第一个参数。第四个参数是数据库的名称（“main”，“temp”等）（如果适用）。第五个参数是负责访问尝试的最内部触发器或视图的名称，或者 None如果此访问尝试直接来自输入SQL代码。

请参考SQLite文档，了解第一个参数的可能值以及第二个和第三个参数的含义，具体取决于第一个参数。 sqlite3模块中提供了所有必需的常量。

set_progress_handler（*handler*，*n* ）

这个例程注册一个回调。该回调函数针对 SQLite虚拟机的每*n*个指令进行调用。如果您想在长时间运行期间从SQLite调用，例如更新GUI，这很有用。

如果要清除以前安装的任何进度处理程序，请使用Nonefor *处理程序*调用该方法。

从处理函数返回一个非零值将终止当前正在执行的查询并导致它引发OperationalError异常。

set_trace_callback（*trace_callback* ）

注册*trace_callback*以针对SQLite后端实际执行的每个SQL语句调用。

传递给回调函数的唯一参数是正在执行的语句（字符串）。回调的返回值被忽略。请注意，后端不仅运行传递给Cursor.execute() 方法的语句。其他来源包括Python模块的事务管理和当前数据库中定义的触发器的执行。

None作为*trace_callback*传递将禁用跟踪回调。

3.3版本的新功能

enable_load_extension（*启用*）

此例程允许/禁止SQLite引擎从共享库加载SQLite扩展。SQLite扩展可以定义新的函数，聚合或全新的虚拟表实现。一个众所周知的扩展是与SQLite一起发布的全文搜索扩展。

可装载的扩展名默认是禁用的。见[1]。

3.2版本中的新功能

```
import sqlite3

con = sqlite3.connect(":memory:")

# enable extension loading
con.enable_load_extension(True)

# Load the fulltext search extension
con.execute("select load_extension('./fts3.so')")

# alternatively you can load the extension using an API call:
# con.load_extension("./fts3.so")

# disable extension loading again
con.enable_load_extension(False)
```

```
# example from SQLite wiki
con.execute("create virtual table recipe using fts3(name, ingredients)")
con.executescript("""
    insert into recipe (name, ingredients) values ('broccoli stew', 'broccoli
    insert into recipe (name, ingredients) values ('pumpkin stew', 'pumpkin or
    insert into recipe (name, ingredients) values ('broccoli pie', 'broccoli c
    insert into recipe (name, ingredients) values ('pumpkin pie', 'pumpkin sug
    """)
for row in con.execute("select rowid, name, ingredients from recipe where name
    print(row)
```

load_extension(*路径*)

此例程从共享库中加载SQLite扩展。您必须先启用扩展加载，enable_load_extension()然后才能使用此例程。

可装载的扩展名默认是禁用的。见[1]。

3.2版本中的新功能

row_factory

您可以将此属性更改为可接受的游标，将游标和原始行作为元组接受，并返回实际结果行。这样，您可以实现更高级的返回结果方式，例如返回也可以按名称访问列的对象。

例：

```
import sqlite3

def dict_factory(cursor, row):
    d = {}
    for idx, col in enumerate(cursor.description):
        d[col[0]] = row[idx]
    return d

con = sqlite3.connect(":memory:")
con.row_factory = dict_factory
cur = con.cursor()
cur.execute("select 1 as a")
print(cur.fetchone()["a"])
```

如果返回一个元组不够，而你想要对列进行基于名称的访问，则应该考虑设置row_factory为高度优化的sqlite3.Row类型。Row提供基于索引和不区分大小写的基于名称的访问，几乎没有内存开销。它可能会比您自己的基于自定义字典的方法或甚至基于db_row的解决方案更好。

text_factory

使用此属性，您可以控制为TEXT 数据类型返回哪些对象。默认情况下，此属性设置为str， sqlite3模块将返回Unicode对象TEXT。如果您想要返回字节串，可以将其设置为bytes。

您还可以将其设置为接受单个字符串参数并返回结果对象的任何其他可调用对象。

有关说明，请参阅以下示例代码：

```
import sqlite3

con = sqlite3.connect(":memory:")
cur = con.cursor()

AUSTRIA = "\xd6sterreich"

# by default, rows are returned as Unicode
cur.execute("select ?", (AUSTRIA,))
row = cur.fetchone()
assert row[0] == AUSTRIA

# but we can make sqlite3 always return bytestrings ...
con.text_factory = bytes
cur.execute("select ?", (AUSTRIA,))
row = cur.fetchone()
assert type(row[0]) is bytes
# the bytestrings will be encoded in UTF-8, unless you stored garbage in the
# database ...
assert row[0] == AUSTRIA.encode("utf-8")

# we can also implement a custom text_factory ...
# here we implement one that appends "foo" to all strings
con.text_factory = lambda x: x.decode("utf-8") + "foo"
cur.execute("select ?", ("bar",))
row = cur.fetchone()
assert row[0] == "barfoo"
```

total_changes

返回自数据库连接打开以来已修改，插入或删除的数据库行总数。

iterdump ( )

返回一个以SQL文本格式转储数据库的迭代器。在保存内存数据库以供日后恢复时很有用。该函数提供了与**sqlite3** shell中的.dump命令相同的功能。

例：

```
# Convert file existing_db.db to SQL dump file dump.sql
import sqlite3

con = sqlite3.connect('existing_db.db')
with open('dump.sql', 'w') as f:
    for line in con.iterdump():
        f.write('%s\n' % line)
```

12.6.3。游标对象

*类*sqlite3.Cursor

一个Cursor实例具有以下属性和方法。

execute（*sql*[，*parameters*]）

执行一条SQL语句。SQL语句可能是参数化的（即占位符而不是SQL文字）。该sqlite3模块支持两种占位符：问号（qmark样式）和命名占位符（命名样式）。

以下是两种样式的示例：

```
import sqlite3

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute("create table people (name_last, age)")

who = "Yeltsin"
age = 72

# This is the qmark style:
cur.execute("insert into people values (?, ?)", (who, age))

# And this is the named style:
cur.execute("select * from people where name_last=:who and age=:age", {"who":

print(cur.fetchone())
```

execute()只会执行一条SQL语句。如果你试图用它来执行多个语句，它会引发一个Warning。使用 executescript()，如果你想用一个调用执行多个SQL语句。

executemany（*sql*，*seq_of_parameters*）

针对在序列*seq_of_parameters*中找到的所有参数序列或映射执行SQL命令。该sqlite3模块还允许使用迭代器产生参数而不是序列。

```
import sqlite3

class IterChars:
    def __init__(self):
        self.count = ord('a')

    def __iter__(self):
        return self

    def __next__(self):
        if self.count > ord('z'):
            raise StopIteration
        self.count += 1
        return (chr(self.count - 1),) # this is a 1-tuple

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute("create table characters(c)")

theIter = IterChars()
cur.executemany("insert into characters(c) values (?)", theIter)

cur.execute("select c from characters")
print(cur.fetchall())
```

以下是使用生成器的较简单示例：

```
import sqlite3
import string

def char_generator():
    for c in string.ascii_lowercase:
        yield (c,)

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute("create table characters(c)")

cur.executemany("insert into characters(c) values (?)", char_generator())

cur.execute("select c from characters")
print(cur.fetchall())
```

executescript (*sql_script*)

这是一次执行多个SQL语句的非标准便利方法。它首先发布一个COMMIT语句，然后执行它作为参数获取的SQL脚本。

*sql_script*可以是一个实例str。

例：

```
import sqlite3

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.executescript("""
    create table person(
        firstname,
        lastname,
        age
    );

    create table book(
        title,
        author,
        published
    );

    insert into book(title, author, published)
    values (
        'Dirk Gently''s Holistic Detective Agency',
        'Douglas Adams',
        1987
    );
    """)
```

fetchone ( )

获取查询结果集的下一行，返回单个序列，或者None没有更多数据可用时。

fetchmany (*size = cursor.arraysize*)

获取查询结果的下一组行，并返回一个列表。没有更多行可用时返回空列表。

每次调用要获取的行数由*size*参数指定。如果没有给出，游标的数组大小决定了要获取的行数。该方法应该尝试获取大小参数所指示的行数。如果由于指定的行数不可用而无法执行此操作，则可能会返回更少的行。

请注意，*大小*参数涉及性能方面的考虑因素。为了获得最佳性能，通常最好使用arraysize属性。如果使用*size*参数，那么最好从一次fetchmany()调用到下一次调用保持相同的值。

fetchall ( )

获取查询结果的所有（剩余）行，并返回一个列表。请注意，游标的arraysize属性可能会影响此操作的性能。没有行可用时返回空列表。

close ( )

现在关闭光标（而不是每当__del__被调用时）。

从这一点开始，光标将无法使用; 一ProgrammingError ，如果任何操作试图用光标将引发异常。

rowcount

虽然模块的Cursor类sqlite3实现了这个属性，但数据库引擎自己对确定“受影响的行”/“所选择的行”的支持是古怪的。

对于executemany()陈述，修改的数量被总结成rowcount。

按照Python DB API Spec的要求，rowcount如果executeXX()游标上没有执行任何操作，或者上一次操作的行数不能被界面确定，则属性“为-1 ”。这包括SELECT 语句，因为我们无法确定查询产生的行数，直到获取所有行。

使用3.6.5之前的SQLite版本，rowcount如果您没有任何条件，则设置为0 。 DELETE FROM table

lastrowid

此只读属性提供最后修改行的rowid。只有在您使用该方法发布INSERT或REPLACE声明时 才会设置execute()。比其他操作INSERT或 REPLACE或者当executemany()被调用时，lastrowid被设置为None。

如果INSERT或者REPLACE语句未能插入先前成功的rowid则返回。

*在版本3.6中更改：*增加了对REPLACE语句的支持。

arraysize

读/写属性，用于控制返回的行数fetchmany()。默认值是1，这意味着每次调用都会获取一行。

description

此只读属性提供最后一个查询的列名称。为了与Python DB API保持兼容，它会为每个元组的最后六项所在的每列返回一个7元组None。

它被设置为SELECT没有任何匹配行的语句。

connection

这个只读属性提供Connection 了Cursor对象使用的SQLite数据库。一个Cursor通过调用创建的对象con.cursor()将有一个 connection引用属性*CON*：

```
>>> con = sqlite3.connect(":memory:")
>>> cur = con.cursor()
>>> cur.connection == con
True
```

12.6.4。行对象

*类*sqlite3. Row

甲Row实例用作一个高度优化的 row_factory用于Connection对象。它试图模仿大部分功能中的元组。

它支持按列名和索引进行映射访问，迭代，表示，相等性测试和len()。

如果两个Row对象具有完全相同的列并且它们的成员相等，则它们相等。

keys ( )

此方法返回列名称的列表。在查询之后，它立即成为每个元组的第一个成员Cursor.description。

*在版本3.5中进行了更改：*添加了对切片的支持。

我们假设我们按照上面给出的例子初始化一个表格：

```
conn = sqlite3.connect(":memory:")
c = conn.cursor()
c.execute('''create table stocks
(date text, trans text, symbol text,
 qty real, price real)''')
c.execute("""insert into stocks
          values ('2006-01-05','BUY','RHAT',100,35.14)""")
conn.commit()
c.close()
```

现在我们插入Row：

```
>>> conn.row_factory = sqlite3.Row
>>> c = conn.cursor()
>>> c.execute('select * from stocks')
<sqlite3.Cursor object at 0x7f4e7dd8fa80>
>>> r = c.fetchone()
>>> type(r)
<class 'sqlite3.Row'>
>>> tuple(r)
('2006-01-05', 'BUY', 'RHAT', 100.0, 35.14)
>>> len(r)
5
```

```
>>> r[2]
'RHAT'
>>> r.keys()
['date', 'trans', 'symbol', 'qty', 'price']
>>> r['qty']
100.0
>>> for member in r:
...     print(member)
...
2006-01-05
BUY
RHAT
100.0
35.14
```

12.6.5。例外

*异常*sqlite3.Warning
一个子类Exception。

*异常*sqlite3.Error
此模块中其他例外的基类。它是一个子类Exception。

*异常*sqlite3.DatabaseError
针对与数据库相关的错误引发异常。

*异常*sqlite3.IntegrityError
当数据库的关系完整性受到影响时引发异常，例如外键检查失败。它是一个子类DatabaseError。

*异常*sqlite3.ProgrammingError
编程错误引发的异常，例如表未找到或已存在的表，SQL语句中的语法错误，指定的参数错误数等等。它是。的一个子类DatabaseError。

12.6.6。SQLite和Python类型

12.6.6.1。简介

SQLite的原生支持以下几种类型：NULL，INTEGER， REAL，TEXT，BLOB。

因此可以将以下Python类型发送给SQLite，而不会有任何问题：

Python类型	型QLite类
None	NULL
int	INTEGER
float	REAL
str	TEXT
bytes	BLOB

这是默认情况下SQLite类型转换为Python类型的方式：

SQLite类型	Python类型
NULL	None
INTEGER	int
REAL	float
TEXT	取决于text_factory， str默认情况下
BLOB	bytes

sqlite3模块的类型系统可以通过两种方式进行扩展：可以通过对象适配将其他Python类型存储在SQLite数据库中，并且可以让sqlite3模块通过转换器将SQLite类型转换为不同的Python类型。

12.6.6.2。使用适配器在SQLite数据库中存储其他Python类型

如前所述，SQLite本身只支持一组有限的类型。要将其他Python类型与SQLite一起使用，必须**将**它们**调整**为sqlite3模块支持的SQLite类型之一：NoneType，int，float，str，bytes之一。

有两种方法可以使sqlite3模块将自定义的Python类型改为支持的类型之一。

12.6.6.2.1。让你的对象适应自己

如果你自己写课程，这是一个很好的方法。假设你有这样的课程：

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y
```

现在您想要将该点存储在单个SQLite列中。首先，您必须首先选择一种支持的类型来表示该点。我们只需使用str并使用分号分隔坐标。然后你需要给你的班级一个必须返回转换后的值的方法。参数*协议*将会是。 __conform__(self, protocol)PrepareProtocol

```
import sqlite3

class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __conform__(self, protocol):
        if protocol is sqlite3.PrepareProtocol:
            return "%f;%f" % (self.x, self.y)

con = sqlite3.connect(":memory:")
cur = con.cursor()

p = Point(4.0, -3.2)
cur.execute("select ?", (p,))
print(cur.fetchone()[0])
```

12.6.6.2.2。注册可调用的适配器

另一种可能性是创建一个将类型转换为字符串表示并将函数注册的函数register_adapter()。

```
import sqlite3

class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

def adapt_point(point):
    return "%f;%f" % (point.x, point.y)

sqlite3.register_adapter(Point, adapt_point)

con = sqlite3.connect(":memory:")
cur = con.cursor()

p = Point(4.0, -3.2)
cur.execute("select ?", (p,))
print(cur.fetchone()[0])
```

该sqlite3模块有两个用于Python内置datetime.date和datetime.datetime类型的默认适配器。现在让我们假设我们想存储datetime.datetime不是ISO表示的对象，而是作为一个Unix时间戳。

```
import sqlite3
import datetime
import time

def adapt_datetime(ts):
    return time.mktime(ts.timetuple())

sqlite3.register_adapter(datetime.datetime, adapt_datetime)

con = sqlite3.connect(":memory:")
cur = con.cursor()

now = datetime.datetime.now()
cur.execute("select ?", (now,))
print(cur.fetchone()[0])
```

12.6.6.3。将SQLite值转换为自定义Python类型

编写适配器可让您将自定义Python类型发送到SQLite。但为了让它变得非常有用，我们需要让Python到SQLite来Python的往返工作。

输入转换器。

让我们回到Point课堂上。我们将通过分号分隔的x和y坐标存储为SQLite中的字符串。

首先，我们将定义一个转换器函数，该函数接受字符串作为参数并Point从中构造一个对象。

> **注意：** 转换器函数**总是**被一个bytes对象调用，不管在哪种数据类型下你发送值到SQLite。

```
def convert_point(s):
    x, y = map(float, s.split(b";"))
    return Point(x, y)
```

现在您需要让sqlite3模块知道您从数据库中选择的内容实际上是一个点。有两种方法可以做到这一点：

- 隐式地通过声明的类型
- 显式地通过列名称

这两种方法在部分中描述的模块功能和常数，在用于常量的条目PARSE_DECLTYPES和PARSE_COLNAMES。

以下示例说明了这两种方法。

```
import sqlite3

class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __repr__(self):
        return "(%f;%f)" % (self.x, self.y)

def adapt_point(point):
    return ("%f;%f" % (point.x, point.y)).encode('ascii')

def convert_point(s):
    x, y = list(map(float, s.split(b";")))
    return Point(x, y)

# Register the adapter
sqlite3.register_adapter(Point, adapt_point)

# Register the converter
sqlite3.register_converter("point", convert_point)

p = Point(4.0, -3.2)

#########################
# 1) Using declared types
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_DECLTYPES)
cur = con.cursor()
cur.execute("create table test(p point)")

cur.execute("insert into test(p) values (?)", (p,))
cur.execute("select p from test")
print("with declared types:", cur.fetchone()[0])
cur.close()
con.close()

#######################
# 1) Using column names
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_COLNAMES)
cur = con.cursor()
cur.execute("create table test(p)")
```

```
cur.execute("insert into test(p) values (?)", (p,))
cur.execute('select p as "p [point]" from test')
print("with column names:", cur.fetchone()[0])
cur.close()
con.close()
```

12.6.6.4。默认适配器和转换器

日期时间模块中有日期和日期时间类型的默认适配器。它们将作为ISO日期/ ISO时间戳发送到SQLite。

默认转换器的名称为“date”， `datetime.date`名称为“timestamp” `datetime.datetime`。

这样，在大多数情况下，您可以使用Python中的日期/时间戳，而不需要额外的操作。适配器的格式也与实验性SQLite日期/时间函数兼容。

以下示例演示了这一点。

```
import sqlite3
import datetime

con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_DECLTYPES|sqlite3.PARSE_
cur = con.cursor()
cur.execute("create table test(d date, ts timestamp)")

today = datetime.date.today()
now = datetime.datetime.now()

cur.execute("insert into test(d, ts) values (?, ?)", (today, now))
cur.execute("select d, ts from test")
row = cur.fetchone()
print(today, "=>", row[0], type(row[0]))
print(now, "=>", row[1], type(row[1]))

cur.execute('select current_date as "d [date]", current_timestamp as "ts [timestamp]"'
row = cur.fetchone()
print("current_date", row[0], type(row[0]))
print("current_timestamp", row[1], type(row[1]))
```

如果存储在SQLite中的时间戳记的分数部分长于6个数字，则其值将被时间戳转换器截断为微秒精度。

12.6.7。控制交易

默认情况下， `sqlite3`模块在数据修改语言（ DML ）语句（即 INSERT/ UPDATE/ DELETE/ REPLACE ）之前隐式打开事务 。

你可以控制`BEGIN`sqlite3通过 调用*isolation_level*参数`connect()`或通过`isolation_level`连接属性隐式执行哪种语句（或者根本不执行）。

如果您想要**自动提交模式**，请设置isolation_level为None。

否则，将其保留为默认值，这将导致一个简单的“BEGIN”语句，或将其设置为SQLite支持的隔离级别之一：“DEFERRED”，“IMMEDIATE”或“EXCLUSIVE”。

当前事务状态通过Connection.in_transaction连接对象的属性公开 。

在版本3.6中更改： sqlite3用于在DDL语句之前隐式提交打开的事务。这已不再是这种情况。

12.6.8。sqlite3高效使用

12.6.8.1。使用快捷方式

使用非标准的execute() ，executemany()并且 executescript()该方法的Connection对象，您的代码可以更简洁，因为你不必创建（通常是多余的）书面Cursor明确对象。相反，Cursor 对象是隐式创建的，这些快捷方法返回游标对象。这样，您可以执行一个SELECT语句并直接使用该Connection对象上的一次调用直接对其进行迭代。

```
import sqlite3

persons = [
    ("Hugo", "Boss"),
    ("Calvin", "Klein")
    ]

con = sqlite3.connect(":memory:")

# Create the table
con.execute("create table person(firstname, lastname)")

# Fill the table
con.executemany("insert into person(firstname, lastname) values (?, ?)", persons)

# Print the table contents
for row in con.execute("select firstname, lastname from person"):
    print(row)

print("I just deleted", con.execute("delete from person").rowcount, "rows")
```

12.6.8.2。按名称而不是索引访问列

该sqlite3模块的一个有用功能是sqlite3.Row设计用作行工厂的内置 类。

使用此类包装的行可以通过索引（如元组）访问，也可以通过名称不区分大小写：

```
import sqlite3

con = sqlite3.connect(":memory:")
con.row_factory = sqlite3.Row

cur = con.cursor()
cur.execute("select 'John' as name, 42 as age")
```

```
for row in cur:
    assert row[0] == row["name"]
    assert row["name"] == row["nAmE"]
    assert row[1] == row["age"]
    assert row[1] == row["AgE"]
```

12.6.8.3。使用连接作为上下文管理器

连接对象可以用作自动提交或回滚事务的上下文管理器。如果发生异常，交易将回滚; 否则，交易承诺：

```
import sqlite3

con = sqlite3.connect(":memory:")
con.execute("create table person (id integer primary key, firstname varchar unique)")

# Successful, con.commit() is called automatically afterwards
with con:
    con.execute("insert into person(firstname) values (?)", ("Joe",))

# con.rollback() is called after the with block finishes with an exception, the
# exception is still raised and must be caught
try:
    with con:
        con.execute("insert into person(firstname) values (?)", ("Joe",))
except sqlite3.IntegrityError:
    print("couldn't add Joe twice")
```

12.6.9。常见问题

12.6.9.1。多线程

较旧的SQLite版本在共享线程之间的连接时遇到了问题。这就是为什么Python模块不允许在线程之间共享连接和游标的原因。如果你仍然尝试这样做，你将在运行时得到一个异常。

唯一的例外是调用该`interrupt()`方法，这只有在从不同的线程调用时才有意义。

脚注

[1] *(1, 2)* 的sqlite3的模块没有默认装载的扩展支持内置，因为一些平台（特别是Mac OS X的）具有被没有这个功能编译SQLite库的版本。要获得可加载的扩展支持，您必须传递--enable-loadable-extensions选项configure。

13.数据压缩和归档

本章介绍的模块支持使用zlib，gzip，bzip2和lzma算法进行数据压缩以及创建ZIP和tar格式的归档。另请参阅 模块提供的归档操作shutil。

13.1。zlib- 与gzip兼容的压缩

对于需要数据压缩的应用程序，此模块中的功能允许使用zlib库进行压缩和解压缩。zlib库在http://www.zlib.net有自己的主页。Python模块和早于1.1.3的zlib库的版本之间存在已知的不兼容性; 1.1.3存在安全漏洞，所以我们推荐使用1.1.4或更高版本。

zlib的函数有很多选项，并且经常需要按特定顺序使用。本文档并不试图涵盖所有的排列组合; 请参阅http://www.zlib.net/manual.html上的zlib手册获取权威信息。

为了读取和写入.gz文件，请参阅gzip模块。

这个模块中可用的异常和功能是：

*异常*zlib.error
    压缩和解压缩错误引发异常。

zlib.adler32（*data*[，*value*]）
    计算的阿德勒-32校验和*数据*。（Adler-32校验和与CRC32几乎一样可靠，但可以更快地计算）。结果是一个无符号的32位整数。如果存在*值*，则将其用作校验和的起始值; 否则，使用默认值1。传入*值*允许通过多个输入的并置计算运行校验和。该算法不具有密码强度，不应用于认证或数字签名。由于该算法被设计用作校验和算法，因此不适合用作通用哈希算法。

    *在版本3.0中更改：*始终返回一个无符号值。要在所有Python版本和平台上生成相同的数值，请使用。adler32(data) & 0xffffffff

zlib.compress（*data*，*level = -1*）
    压缩*数据中*的字节，返回包含压缩数据的字节对象。*级别*是从或0到9或-1控制压缩*级别*的整数; 1（Z_BEST_SPEED）最快，产生的压缩最少，9（Z_BEST_COMPRESSION）最慢，并且产生最多。0（Z_NO_COMPRESSION）没有压缩。默认值是-1（Z_DEFAULT_COMPRESSION）。Z_DEFAULT_COMPRESSION表示速度和压缩之间的默认折衷（当前等同于级别6）。error发生任何错误时引发异常。

    *在版本3.6中进行了更改：*现在可以将*级别*用作关键字参数。

zlib.compressobj（*level = -1*，*method = DEFLATED*，*wbits = MAX_WBITS*，*memLevel = DEF_MEM_LEVEL*，*strategy = Z_DEFAULT_STRATEGY*[，*zdict*]）
    返回压缩对象，用于压缩一次无法放入内存的数据流。

    *级别*是压缩级别 - 从或0到的整数。（Z_BEST_SPEED）的值是最快的并且产生最小的压缩，而（Z_BEST_COMPRESSION）的值是最慢的并且产生最多的。（Z_NO_COMPRESSION）没有压缩。默认值是（Z_DEFAULT_COMPRESSION）。Z_DEFAULT_COMPRESSION表示速度和压缩之间的默认折衷（当前等同于级别6）。9-1190-1

    *方法*是压缩算法。目前，唯一支持的值是 DEFLATED。

所述*WBITS*参数控制所述历史缓冲区（或“窗口大小”）压缩数据时所使用的大小，以及是否一个头和尾被包括在输出中。它可能需要几个范围的值，默认为15（MAX_WBITS）：

- +9到+15：窗口大小的基二对数，因此范围介于512和32768之间。较大的值会产生更好的压缩，但会增加内存使用量。结果输出将包含一个zlib特定的标题和预告片。
- -9到-15：使用*wbits*的绝对值作为窗口大小对数，同时生成没有标头或尾随校验和的原始输出流。
- +25到+31 = 16 +（9到15）：将值的低4位用作窗口大小对数，同时在输出中包含基本的**gzip**头和尾部校验和。

所述*memLevel*参数控制的存储器用于内部压缩状态的量。有效值范围从1至9。较高的值使用更多的内存，但更快并产生较小的输出。

*策略*用于调整压缩算法。可能的值是 Z_DEFAULT_STRATEGY，Z_FILTERED，Z_HUFFMAN_ONLY，Z_RLE（zlib的1.2.0.1）和Z_FIXED（zlib的1.2.2.2）。

*zdict*是一个预定义的压缩字典。这是一系列bytes包含子序列的字节（如对象），这些子序列预期会在要压缩的数据中频繁出现。那些预计最常见的子序列应该出现在字典的末尾。

*在版本3.3中进行了更改：*添加了*zdict*参数和关键字参数支持。

zlib.**crc32**（*data*[，*value*]）

计算的CRC（循环冗余校验）校验*数据*。结果是一个无符号的32位整数。如果存在*值*，则将其用作校验和的起始值; 否则，使用默认值0。传入*值*允许通过多个输入的并置计算运行校验和。该算法不具有密码强度，不应用于认证或数字签名。由于该算法被设计用作校验和算法，因此不适合用作通用哈希算法。

*在版本3.0中更改：*始终返回一个无符号值。要在所有Python版本和平台上生成相同的数值，请使用。crc32(data) & 0xffffffff

zlib.**decompress**（*data*，*wbits = MAX_WBITS*，*bufsize = DEF_BUF_SIZE*）

解压缩*数据中*的字节，返回包含未压缩数据的字节对象。所述*WBITS*参数取决于的格式*数据*，并在下面进一步讨论。如果给出*bufsize*，则将其用作输出缓冲区的初始大小。error发生任何错误时引发异常。

该*WBITS*参数控制历史缓冲区的大小（或“窗口大小”），什么头和尾格式的预期。它与参数类似compressobj()，但接受更多的值范围：

- +8到+15：窗口大小的基二对数。输入必须包含zlib标题和预告片。
- 0：从zlib标题自动确定窗口大小。仅在zlib 1.2.3.5以后才支持。
- -8到-15：使用*wbits*的绝对值作为窗口大小的对数。输入必须是没有标题或预告片的原始流。
- +24到+31 = 16 +（8到15）：使用该值的低4位作为窗口大小对数。输入必须包含gzip标题和预告片。
- +40到+47 = 32 +（8到15）：使用该值的低4位作为窗口大小对数，并自动接受zlib或gzip格式。

在解压缩流时，窗口大小不得小于最初用于压缩流的大小; 使用太小的值可能会导致error异常。默认的*wbits*值对应于最大的窗口大小，并且需要包含zlib头部和尾部。

*bufsize*是用于保存解压缩数据的缓冲区的初始大小。如果需要更多的空间，缓冲区的大小将根据需要增加，因此您不必完全正确地获取该值; 调整它只会节省几个电话malloc()。

在版本3.6中更改： *wbits*和*bufsize*可以用作关键字参数。

zlib.**decompressobj**(*wbits = MAX_WBITS* [, *zdict*])

返回一个解压缩对象，用于解压一次无法放入内存的数据流。

该*WBITS*参数控制历史缓冲区的大小（或“窗口大小”），什么头和尾格式的预期。它与解压缩（）所描述的含义相同。

所述*zdict*参数指定的预定义压缩词典。如果提供，它必须与生成要解压缩的数据的压缩器所使用的字典相同。

> **注意：** 如果*zdict*是一个可变对象（如a `bytearray`），则不能在调用`decompressobj()`和第一次调用解压缩`decompress()`方法之间修改其内容。

版本3.3中更改： 添加了*zdict*参数。

压缩对象支持以下方法：

Compress.**compress**(*数据*)

压缩*数据*，返回一个包含压缩的数据在数据的至少一部分的字节对象*数据*。这些数据应连接到前面调用该`compress()`方法所产生的输出。一些输入可能会保存在内部缓冲区中以备后续处理。

Compress.**flush**([*模式*])

处理所有挂起的输入，并返回包含剩余压缩输出的字节对象。 *模式*可以从常数被选择`Z_NO_FLUSH`，`Z_PARTIAL_FLUSH`，`Z_SYNC_FLUSH`，`Z_FULL_FLUSH`，`Z_BLOCK`（ZLIB 1.2.3.4），或`Z_FINISH`，默认为`Z_FINISH`。除了`Z_FINISH`所有常量允许压缩更多的数据字节串，同时`Z_FINISH`完成压缩流并防止压缩更多数据。打完电话后`flush()`与*模式*设置为`Z_FINISH`，该`compress()`方法不能被再次调用; 唯一现实的行动是删除对象。

Compress.**copy**( )

返回压缩对象的副本。这可以用来有效地压缩一组共享公共初始前缀的数据。

解压缩对象支持以下方法和属性：

Decompress.**unused_data**

包含压缩数据末尾的任何字节的字节对象。也就是说，`b""`直到包含压缩数据的最后一个字节可用为止。如果整个字节串包含压缩数据，则这是`b""`一个空字节对象。

Decompress.**unconsumed_tail**

一个字节对象，其中包含上次`decompress()`调用未消耗的任何数据，因为它超出了未压缩数据缓冲区的限制。这些数据还没有被zlib机器看到，因此您必须将它（可能连接到更多的数据）反馈给后续的`decompress()`方法调用以获得正确的输出。

Decompress.**eof**

指示是否已到达压缩数据流的末尾的布尔值。

这使得区分正确形成的压缩流和不完整或截断的压缩流成为可能。

3.3版本的新功能

Decompress.**decompress**(*data*, *max_length = 0*)

解压缩*数据*，返回一个字节对象，该对象包含与*字符串*中至少部分数据对应的未压缩数据。这些数据应连接到前面调用该decompress()方法所产生的输出。一些输入数据可能会保存在内部缓冲区中以备后续处理。

如果可选参数*max_length*不为零，则返回值不会超过*max_length*。这可能意味着并非所有的压缩输入都可以被处理；未使用的数据将被存储在属性中unconsumed_tail。decompress()如果要继续解压缩，则必须将此字符串传递给后续调用。如果*max_length*为零，则整个输入被解压缩，并且unconsumed_tail是空的。

*在版本3.6中更改：*max_length可以用作关键字参数。

Decompress.**flush**([*length*])

处理所有挂起的输入，并返回包含剩余未压缩输出的字节对象。调用之后flush()，该decompress()方法不能再被调用；唯一现实的行动是删除对象。

可选参数*长度*设置输出缓冲区的初始大小。

Decompress.**copy**()

返回解压缩对象的副本。这可以用来在数据流的中途保存解压缩器的状态，以加速随后在随后的点对流的随机搜索。

有关正在使用的zlib库版本的信息可通过以下常量获得：

zlib.**ZLIB_VERSION**

用于构建模块的zlib库的版本字符串。这可能与运行时实际使用的zlib库不同，后者可用ZLIB_RUNTIME_VERSION。

zlib.**ZLIB_RUNTIME_VERSION**

解释器实际加载的zlib库的版本字符串。

3.3版本的新功能

也可以看看:

模 gzip

读写**gzip** -format文件。

http://www.zlib.net

zlib库的主页。

http://www.zlib.net/manual.html

zlib手册解释了库的许多功能的语义和用法。

13.2。 gzip- 支持gzip文件

源代码： Lib / gzip.py

这个模块提供了一个简单的界面来压缩和解压缩文件，就像GNU程序**gzip**和**gunzip**一样。

数据压缩由zlib模块提供。

该gzip模块提供了GzipFile类，以及 open() ， compress()和decompress()方便的功能。本GzipFile类读取和写入**gzip的** -format文件，自动压缩或解压缩数据，使得它看起来像一个普通的文件对象。

请注意，该模块不支持**gzip**和**gunzip**程序可以解压缩的其他文件格式，例如**压缩**和**打包**产生的文件格式 。

该模块定义了以下项目：

gzip. open （ *filename* ， *mode ='rb'* ， *compresslevel = 9* ， *encoding = None* ， *errors = None* ， *newline = None* ）

以二进制或文本模式打开gzip压缩文件，返回文件对象。

该*文件名*参数可以是一个实际的文件名（一个str或 bytes对象），或者现有文件对象从读取或写入到。

所述*模式*参数可以是任何的'r'，'rb'，'a'，'ab'， 'w'，'wb'，'x'或'xb'二进制模式，或者'rt'， 'at'，'wt'，或'xt'为文本模式。默认是'rb'。

所述*compresslevel*参数是从0到9的整数，作为用于 GzipFile构造函数。

对于二进制模式，这个函数相当于GzipFile 构造函数：。在这种情况下， 不得提供*编码*，*错误*和*换行*参数。GzipFile(filename, mode, compresslevel)

对于文本模式，GzipFile创建一个对象，并用io.TextIOWrapper指定的编码，错误处理行为和行结束包装在一个 实例中。

*在版本3.3中进行了更改：*增加了对作为文件对象的文件*名的*支持，对文本模式的支持以及*编码*，*错误*和*换行*参数的支持。

*改变在3.4版本：*为增加的支持'x'，'xb'和'xt'模式。

*在版本3.6中更改：*接受类似路径的对象。

class gzip. GzipFile （ *filename = None* ， *mode = None* ， *compresslevel = 9* ， *fileobj = None* ， *mtime = None* ）

GzipFile类的构造函数，它模拟文件对象的大部分方法，但方法除外truncate() 。至少*fileobj*和*filename中的一个*必须被赋予一个不平凡的值。

新的类实例基于*fileobj*，它可以是常规文件， io.BytesIO对象或模拟文件的任何其他对象。它默认为None，在这种情况下打开*文件名*以提供文件对象。

当*fileobj*不是时None，*filename*参数仅用于包含在**gzip**文件头中，其中可能包含未压缩文件的原始文件名。如果可辨别，它默认为*fileobj*的文件名; 否则，它默认为空字符串，在这种情况下，原始文件名不包含在标题中。

该*模式*参数可以是任意的'r'，'rb'，'a'，'ab'，'w'，'wb'，'x'，或者'xb'，根据文件是否被读取或写入。如果可辨别，缺省值是*fileobj*的模式; 否则，默认是'rb'。

请注意，该文件始终以二进制模式打开。要以文本模式打开一个压缩文件，使用open()（或包装你GzipFile有 io.TextIOWrapper）。

所述*compresslevel*参数是从一个整数0，以9控制压缩的水平; 1是最快的并且产生最小的压缩，并且9是最慢的并且产生最大的压缩。 0 没有压缩。默认是9。

的*修改时间*参数是压缩时将被写入到最后的修改时间字段中的数据流中的可选的数字时间戳记。它只能在压缩模式下提供。如果省略或None使用当前时间。查看mtime属性以获取更多详细信息。

调用GzipFile对象的close()方法不会关闭 *fileobj*，因为您可能希望在压缩数据之后附加更多的材质。这还允许您将io.BytesIO打开的对象作为*fileobj传递*，并使用io.BytesIO对象的getvalue()方法检索得到的内存缓冲区 。

GzipFile支持io.BufferedIOBase接口，包括迭代和with语句。只有这个 truncate()方法没有被执行。

GzipFile 还提供以下方法和属性：

peek（*n*）

在不提前文件位置的情况下读取*n个*未压缩的字节。对压缩流最多进行一次单独读取以满足呼叫。返回的字节数可能比请求的更多或更少。

> **注意：** 虽然调用peek()不会更改文件的位置GzipFile，但它可能会改变底层文件对象的位置（例如，如果GzipFile是使用*fileobj*参数构造的 *话*）。

3.2版本中的新功能

mtime

解压缩时，可以从该属性读取最近读取的标题中最后修改时间字段的值，作为整数。读取任何标题之前的初始值是None。

所有**gzip**压缩流都需要包含此时间戳字段。一些程序，如**gunzip**，使用时间戳。格式与返回的对象的返回值time.time()和st_mtime属性相同 os.stat()。

*在版本3.1中*进行了*更改：* with添加了对该语句的支持，以及 *mtime*构造函数参数和mtime属性。

*在版本3.2中进行了更改：添加了*对零填充和不可见文件的支持。

*在版本3.3中更改：*该io.BufferedIOBase.read1()方法现在已实施。

*在版本3.4中进行了更改：*增加了对模式'x'和'xb'模式的支持。

*在版本3.5中进行了更改：*增加了对编写任意 字节类对象的支持。该read()方法现在接受一个参数 None。

*在版本3.6中更改：*接受类似路径的对象。

gzip.**compress**(*data*, *compresslevel = 9*)

压缩*数据*，返回bytes包含压缩数据的对象。 *compresslevel*与GzipFile上面的构造函数具有相同的含义。

3.2版本中的新功能

gzip.**decompress**(*数据*)

解压缩*数据*，返回bytes包含未压缩数据的对象。

3.2版本中的新功能

13.2.1。用法示例

如何读取压缩文件的示例：

```
import gzip
with gzip.open('/home/joe/file.txt.gz', 'rb') as f:
    file_content = f.read()
```

如何创建压缩的GZIP文件的示例：

```
import gzip
content = b"Lots of content here"
with gzip.open('/home/joe/file.txt.gz', 'wb') as f:
    f.write(content)
```

如何GZIP压缩现有文件的示例：

```
import gzip
import shutil
with open('/home/joe/file.txt', 'rb') as f_in:
    with gzip.open('/home/joe/file.txt.gz', 'wb') as f_out:
        shutil.copyfileobj(f_in, f_out)
```

如何GZIP压缩二进制字符串的示例：

```
import gzip
s_in = b"Lots of content here"
s_out = gzip.compress(s_in)
```

也可以看看:

模 zlib

基本数据压缩模块需要支持**gzip**文件格式。

13.3。bz2- 支持bzip2压缩

源代码： Lib / bz2.py

该模块提供了一个全面的接口，用于使用bzip2压缩算法压缩和解压缩数据。

该bz2模块包含：

- 读取和写入压缩文件的open()函数和BZ2File类。
- 在BZ2Compressor和BZ2Decompressor类增量（DE）压缩。
- 该compress()和decompress()功能一次性的（de）压缩。

该模块中的所有类都可以安全地从多个线程访问。

13.3.1。（德）压缩文件

bz2.open（*filename*，*mode ='r'*，*compresslevel = 9*，*encoding = None*，*errors = None*，*newline = None* ）

以二进制或文本模式打开一个bzip2压缩文件，返回一个文件对象。

与构造函数一样BZ2File，*filename*参数可以是实际的文件名（a str或bytes对象），也可以是要读取或写入的现有文件对象。

所述*模式*参数可以是任何的'r'，'rb'，'w'，'wb'，'x'，'xb'，'a'或'ab'二进制模式，或者'rt'，'wt'，'xt'，或'at'为文本模式。默认是'rb'。

所述*compresslevel*参数是从1到9的整数，作为用于 BZ2File构造函数。

对于二进制模式，这个函数相当于BZ2File 构造函数：。在这种情况下，不得提供*编码*，*错误*和*换行*参数。BZ2File(filename, mode, compresslevel=compresslevel)

对于文本模式，BZ2File创建一个对象，并用io.TextIOWrapper指定的编码，错误处理行为和行结束包装在一个 实例中。

3.3版本的新功能

*改变在3.4版本：*将'x'加入（独家创作）模式。

*在版本3.6中更改：*接受类似路径的对象。

class bz2.BZ2File（*filename*，*mode ='r'*，*buffering = None*，*compresslevel = 9* ）

以二进制模式打开一个bzip2压缩文件。

如果*filename*是a str或bytesobject，则直接打开指定的文件。否则，*文件名*应该是一个文件对象，它将用于读取或写入压缩数据。

所述*模式*参数可以是'r'用于读取（默认值），'w'用于改写，'x'专用创建，或'a'为附加。这些可以等价表示为'rb'，'wb'，'xb'和'ab'分别。

如果*filename*是文件对象（而不是实际的文件名），'w'则不会截断该文件，而是等同于该文件'a'。

该*缓冲*参数将被忽略。它的使用已被弃用。

如果*mode*是'w'或'a'，*compresslevel*可以是一个介于两者之间的数字，1并9指定压缩级别：1产生最小压缩，9（默认）产生最大压缩。

如果*模式*是'r'，输入文件可能是多个压缩流的串联。

BZ2File提供io.BufferedIOBase除了detach()和之外指定的所有成员 truncate()。 with支持迭代和声明。

BZ2File 还提供了以下方法：

peek（[*n*]）

返回缓冲的数据而不提前文件位置。至少有一个字节的数据将被返回（除非在EOF处）。返回的确切字节数是未指定的。

> **注意：** 虽然调用peek()不会更改文件的位置BZ2File，但它可能会更改基础文件对象的位置（例如，如果BZ2File通过传递文件对象的*文件名来*构造它）。

3.3版本的新功能

*在版本3.1中*进行了*更改：* with添加了对语句的支持。

改变在3.3版本： 该 fileno()， readable()， seekable()， writable()， read1() 以及 readinto() 加入方法。

在版本3.3中更改： 支持添加了*文件名*作为文件对象而不是实际的文件名。

改变在3.3版本： 将'a'添加（追加）模式下，读取多流文件的支持一起。

改变在3.4版本： 将'x'加入（独家创作）模式。

在版本3.5中改变： 该read()方法现在接受一个参数 None。

在版本3.6中更改： 接受类似路径的对象。

13.3.2。增量（去）压缩

class bz2.BZ2Compressor（*compresslevel = 9* ）

创建一个新的压缩机对象。该对象可用于增量压缩数据。对于一次压缩，请使用该compress()功能。

compresslevel，如果给出，必须是介于1和之间的数字9。默认是9。

compress（*数据*）

向压缩机对象提供数据。如果可能，返回压缩数据块，否则返回空字节串。

当您完成向压缩机提供数据时，请调用该 `flush()` 方法完成压缩过程。

`flush ( )`

完成压缩过程。返回内部缓冲区中的压缩数据。

在调用此方法后，可能不会使用压缩器对象。

类 bz2. BZ2Decompressor

创建一个新的解压缩器对象。这个对象可以用来递增地解压缩数据。对于一次压缩，请使用该`decompress()`功能。

> **注意：** 这个类不透明地处理包含多个压缩流的输入，不像`decompress()`和`BZ2File`。如果您需要解压缩多流输入`BZ2Decompressor`，则必须为每个流使用新的解压缩器。

`decompress ( data , max_length = -1 )`

解压缩*数据*（类似字节的对象），以字节形式返回未压缩的数据。一些*数据*可能会在内部进行缓冲，以便稍后调用`decompress()`。返回的数据应与之前任何调用的输出连接`decompress()`。

如果*max_length*非负，则返回 解压缩数据的最多*max_length*个字节。如果达到此限制并可以生成更多输出，则该`needs_input`属性将设置为`False`。在这种情况下，下一个呼叫到 `decompress()`可以提供*数据*作为`b''`以获得更多的输出。

如果所有输入数据都被解压缩并返回（因为这小于*max_length*个字节，或者因为*max_length*为负数），该`needs_input`属性将被设置为`True`。

试图在流结束后解压缩数据引发*EOFError*。在流结束后找到的任何数据都将被忽略并保存在`unused_data`属性中。

*在版本3.5中更改：*添加了*max_length*参数。

`eof`

`True` 如果已到达流结束标记。

3.3版本的新功能

`unused_data`

在压缩流结束后找到数据。

如果在到达流的末尾之前访问此属性，则其值将为`b''`。

`needs_input`

`False`如果该`decompress()`方法可以在需要新的未压缩输入之前提供更多的解压缩数据。

3.5版本中的新功能。

13.3.3。一次性（去）压缩

bz2.compress（*data*，*compresslevel = 9* ）

压缩*数据*。

compresslevel，如果给出，必须是介于1和之间的数字9。默认是9。

对于增量压缩，请BZ2Compressor改用。

bz2.decompress（*数据*）

解压缩*数据*。

如果*数据*是多个压缩流的串联，则解压缩所有流。

对于增量解压缩，请BZ2Decompressor改用。

*在版本3.3中更改：*添加了对多流输入的支持。

13.4。 lzma- 使用LZMA算法进行压缩

3.3版本的新功能

源代码： Lib / lzma.py

该模块提供使用LZMA压缩算法压缩和解压缩数据的类和便利功能。还包括一个文件接口，支持**xz**实用程序使用的.xz传统.lzma文件格式 以及原始压缩流。

该模块提供的接口与模块非常相似bz2 。但是，请注意，LZMAFile是*不是*线程安全的，不像bz2.BZ2File ，所以如果你需要使用一个单一的LZMAFile从多个线程情况下，有必要用锁来保护它。

*异常*lzma.LZMAError
当在压缩或解压缩过程中，或在初始化压缩器/解压缩器状态时发生错误时，会引发此异常。

13.4.1。读取和写入压缩文件

lzma.open（*filename*，*mode =“rb”*，***，*format = None*，*check = -1*，*preset = None*，*filters = None*，*encoding = None*，*errors = None*，*newline = None* ）
以二进制或文本模式打开一个LZMA压缩文件，返回一个文件对象。

该*文件名*参数可以是一个实际的文件名（给定为一个 str ，bytes或路径状物体），在这种情况下，指定的文件被打开，或者它可以是现有的文件对象从读取或写入。

所述*模式*参数可以是任何的"r" ，"rb" ，"w" ，"wb" ， "x" ，"xb" ，"a"或"ab"二进制模式，或者"rt" ， "wt" ，"xt" ，或"at"为文本模式。默认是"rb"。

打开文件进行读取时，*格式*和*过滤器*参数与for具有相同的含义LZMADecompressor。在这种情况下，不应使用*检查*和*预设*参数。

打开文件进行写入时，*格式*，*检查*，*预设*和 *过滤*参数的含义与以前相同LZMACompressor。

对于二进制模式，这个函数相当于LZMAFile 构造函数：。在这种情况下，不得提供*编码*，*错误*和*换行*参数。LZMAFile(filename, mode, ...)

对于文本模式，LZMAFile创建一个对象，并用io.TextIOWrapper指定的编码，错误处理行为和行结束包装在一个 实例中。

*改变在3.4版本：*为增加的支持"x" ，"xb"和"xt"模式。

*在版本3.6中更改：*接受类似路径的对象。

class lzma.LZMAFile（*filename = None*，*mode =“r”*，***，*format = None*，*check = -1*，*preset = None*，*filters = None* ）
以二进制模式打开LZMA压缩文件。

一个LZMAFile可以打包一个已经打开的文件对象，或者直接在一个已命名的文件上操作。的*文件名*参数指定任一文件对象来包装，或要打开的文件的名称（作为str， bytes或路径状物体）。封装现有文件对象时，封装后的封装文件将不会 LZMAFile关闭。

所述*模式*参数可以是"r"用于读取（默认值），"w"用于改写，"x"专用创建，或"a"为附加。这些可以等价表示为"rb"，"wb"，"xb"和"ab" 分别。

如果*filename*是文件对象（而不是实际的文件名），"w"则不会截断该文件，而是等同于该文件"a"。

打开文件进行读取时，输入文件可能是多个单独压缩流的串联。这些透明解码为单个逻辑流。

打开文件进行读取时，*格式*和*过滤器*参数与for具有相同的含义LZMADecompressor。在这种情况下，不应使用*检查*和*预设*参数。

打开文件进行写入时，*格式*，*检查*，*预设*和 *过滤*参数的含义与以前相同LZMACompressor。

LZMAFile支持io.BufferedIOBase除了detach()和指定的所有成员 truncate()。 with支持迭代和声明。

还提供了以下方法：

peek（*size = -1* ）

返回缓冲的数据而不提前文件位置。至少一个字节的数据将被返回，除非EOF已经达到。返回的字节的确切数量是未指定的（*大小*参数被忽略）。

> **注意：** 虽然调用peek()不会更改文件的位置LZMAFile，但它可能会更改基础文件对象的位置（例如，如果LZMAFile通过传递文件对象的*文件名来*构造它）。

*在版本3.4中*进行了*更改*：增加了对模式"x"和"xb"模式的支持。

在版本3.5中改变：该read()方法现在接受一个参数 None。

在版本3.6中更改：接受类似路径的对象。

13.4.2。压缩和解压缩内存中的数据

class lzma.LZMACompressor（*format = FORMAT_XZ*，*check = -1*，*preset = None*，*filters = None* ）

创建一个压缩器对象，可用于增量压缩数据。

有关压缩单个数据块的更方便的方法，请参阅 compress()。

该*格式*参数指定应该用什么容器格式。可能的值是：

- FORMAT_XZ：.xz容器格式。
  这是默认格式。
- FORMAT_ALONE：传统的.lzma容器格式。

这种格式更受限于.xz-它不支持完整性检查或多个过滤器。

- FORMAT_RAW：原始数据流，不使用任何容器格式。
  此格式说明符不支持完整性检查，并且要求您始终指定自定义过滤器链（用于压缩和解压缩）。另外，以这种方式压缩的数据不能使用FORMAT_AUTO（参见LZMADecompressor）进行解压缩。

所述*检查*参数指定在所述压缩数据以包括完整性校验的类型。解压缩时使用此检查，以确保数据未被损坏。可能的值是：

- CHECK_NONE：没有完整性检查。这是默认（和唯一可接受的值）FORMAT_ALONE和FORMAT_RAW。
- CHECK_CRC32：32位循环冗余校验。
- CHECK_CRC64：64位循环冗余校验。这是默认的FORMAT_XZ。
- CHECK_SHA256：256位安全散列算法。

如果指定的检查不受支持，LZMAError则会引发。

可以将压缩设置指定为预设压缩级别（使用*预设*参数），或详细指定为自定义滤镜链（使用*filters*参数）。

的*预设*参数（如果提供的话）应之间的整数0和 9-ED OR（含），任选地与恒定PRESET_EXTREME。如果既未*预设*也未*过滤*，默认行为是使用PRESET_DEFAULT（预设级别6）。较高的预设会产生较小的输出，但会使压缩过程变慢。

注意: 除了CPU密集度更高之外，预设更高的压缩也需要更多的内存（并产生需要更多内存来解压缩的输出）。以预设9为例，LZMACompressor对象的开销 可高达800 MiB。出于这个原因，通常最好坚持使用默认预设。

该*过滤器*参数（如果提供的话）应当是一种过滤器链说明符。有关详情，请参阅指定自定义过滤器链。

compress（*数据*）
压缩*数据*（一个bytes对象），返回bytes 包含至少部分输入的压缩数据的对象。一些*数据*可能会在内部进行缓冲，以便稍后调用 compress()和flush()。返回的数据应与之前任何调用的输出连接compress()。

flush（）
完成压缩过程，返回bytes包含存储在压缩机内部缓冲区中的任何数据的对象。

此方法被调用后，压缩机不能使用。

class lzma.LZMADecompressor（*format = FORMAT_AUTO*，*memlimit = None*，*filters = None*）
创建一个解压缩器对象，可用于增量式解压缩数据。

有关一次解压缩整个压缩流的更方便的方法，请参阅decompress()。

该*格式*参数指定要使用的容器格式。默认是FORMAT_AUTO，它可以解压缩.xz和.lzma文件。其他可能的值是FORMAT_XZ，FORMAT_ALONE和FORMAT_RAW。

的*MEMLIMIT*参数指定的内存，该解压缩器可使用量的限制（以字节为单位）。当使用该参数时，LZMAError如果在给定的内存限制内无法解压缩输入，则解压缩将失败。

该*过滤器*参数指定用于创建被减压的流过滤器链。如果*格式*是 FORMAT_RAW ，则此参数是必需的，但不应用于其他格式。有关过滤器链的更多信息，请参阅指定自定义过滤器链。

> **注意：** 这个类不透明地处理包含多个压缩流的输入，不像decompress()和LZMAFile。要使用解压多流输入LZMADecompressor，您必须为每个流创建一个新的解压缩器。

decompress（*data*，*max_length = -1* ）

解压缩*数据*（类似字节的对象），以字节形式返回未压缩的数据。一些*数据*可能会在内部进行缓冲，以便稍后调用decompress()。返回的数据应与之前任何调用的输出连接decompress()。

如果*max_length*非负，则返回 解压缩数据的最多*max_length*个字节。如果达到此限制并可以生成更多输出，则该needs_input属性将设置为False。在这种情况下，下一个呼叫到 decompress()可以提供*数据*作为b''以获得更多的输出。

如果所有输入数据都被解压缩并返回（因为这小于*max_length*个字节，或者因为*max_length*为负数），该needs_input属性将被设置为True。

试图在流结束后解压缩数据引发*EOFError*。在流结束后找到的任何数据都将被忽略并保存在unused_data属性中。

*在版本3.5中更改：*添加了*max_length*参数。

check

输入流使用的完整性检查的ID。这可能是 CHECK_UNKNOWN直到有足够的输入已被解码，以确定哪些完整性检查它的用途。

eof

True 如果已到达流结束标记。

unused_data

在压缩流结束后找到数据。

在流到达之前，这将是b""。

needs_input

False如果该decompress()方法可以在需要新的未压缩输入之前提供更多的解压缩数据。

3.5版本中的新功能。

lzma.compress（*data*，*format = FORMAT_XZ*，*check = -1*，*preset = None*，*filters = None* ）

压缩*数据*（一个bytes对象），将压缩数据作为bytes对象返回 。

请参阅LZMACompressor上面的*格式*描述，*检查*， *预设*和*过滤*参数。

lzma.decompress（*data*，*format* = *FORMAT_AUTO*，*memlimit* = *None*，*filters* = *None*）

解压缩*数据*（一个bytes对象），将未压缩的数据作为bytes对象返回。

如果*数据*是多个不同压缩流的连接，则解压缩所有这些流，然后返回结果的连接。

请参阅LZMADecompressor上面的*格式*说明， *memlimit*和*过滤器*参数。

13.4.3。杂项

lzma.is_check_supported（*检查*）

如果此系统支持给定的完整性检查，则返回true。

CHECK_NONE并CHECK_CRC32始终得到支持。 CHECK_CRC64并且CHECK_SHA256如果你正在使用的版本可能不可用**liblzma**是用有限的功能集编译。

13.4.4。指定自定义过滤器链

过滤器链说明符是一个字典序列，其中每个字典包含单个过滤器的ID和选项。每个字典都必须包含密钥"id"，并且可能包含额外的密钥以指定依赖于过滤器的选项。有效的过滤器ID如下所示：

- 压缩过滤器：
    - FILTER_LZMA1（用于FORMAT_ALONE）
    - FILTER_LZMA2（用于FORMAT_XZ和FORMAT_RAW）
- Delta滤波器：
    - FILTER_DELTA
- 分支呼叫跳转（BCJ）过滤器：
    - FILTER_X86
    - FILTER_IA64
    - FILTER_ARM
    - FILTER_ARMTHUMB
    - FILTER_POWERPC
    - FILTER_SPARC

过滤器链最多可以包含4个过滤器，并且不能为空。链中的最后一个过滤器必须是压缩过滤器，而其他过滤器必须是delta或BCJ过滤器。

压缩过滤器支持以下选项（在代表过滤器的字典中指定为附加条目）：

- preset：一个压缩预设，用作未明确指定的选项的默认值来源。
- dict_size：字节大小（字节）。这应该在4 KiB和1.5 GiB（含）之间。
- lc：文字上下文位数。
- lp：文字位置位数。总和不得超过4。 lc + lp
- pb：位数位数; 最多不得超过4个。
- mode：MODE_FAST或MODE_NORMAL。
- nice_len：什么应该被视为一个“好长度”的比赛。这应该是273或更少。

- mf：用什么比赛发现者- ，MF_HC3 ，MF_HC4 ， MF_BT2 ，MF_BT3或MF_BT4。
- depth：匹配查找器使用的最大搜索深度。0（默认）意味着根据其他过滤器选项自动选择。

Delta滤波器存储字节之间的差异，在某些情况下为压缩器产生更多的重复输入。它支持一个选项， dist。这表示要减去的字节之间的距离。默认值为1，即取相邻字节之间的差异。

BCJ过滤器旨在应用于机器代码。它们将代码中的相关分支，调用和跳转转换为绝对寻址，目的是增加压缩器可以利用的冗余。这些过滤器支持一个选项， start_offset。这指定了应映射到输入数据开头的地址。默认值是0。

13.4.5。示例

读取压缩文件：

```
import lzma
with lzma.open("file.xz") as f:
    file_content = f.read()
```

创建一个压缩文件：

```
import lzma
data = b"Insert Data Here"
with lzma.open("file.xz", "w") as f:
    f.write(data)
```

压缩内存中的数据：

```
import lzma
data_in = b"Insert Data Here"
data_out = lzma.compress(data_in)
```

增量压缩：

```
import lzma
lzc = lzma.LZMACompressor()
out1 = lzc.compress(b"Some data\n")
out2 = lzc.compress(b"Another piece of data\n")
out3 = lzc.compress(b"Even more data\n")
out4 = lzc.flush()
# Concatenate all the partial results:
result = b"".join([out1, out2, out3, out4])
```

将压缩数据写入已经打开的文件：

```
import lzma
with open("file.xz", "wb") as f:
    f.write(b"This data will not be compressed\n")
    with lzma.open(f, "w") as lzf:
        lzf.write(b"This *will* be compressed\n")
    f.write(b"Not compressed\n")
```

使用自定义过滤器链创建压缩文件：

```
import lzma
my_filters = [
    {"id": lzma.FILTER_DELTA, "dist": 5},
    {"id": lzma.FILTER_LZMA2, "preset": 7 | lzma.PRESET_EXTREME},
]
with lzma.open("file.xz", "w", filters=my_filters) as f:
    f.write(b"blah blah blah")
```

13.5。zipfile- 使用ZIP档案

源代码： Lib / zipfile.py

ZIP文件格式是一种常用的归档和压缩标准。此模块提供了创建，读取，写入，追加和列出ZIP文件的工具。此模块的任何高级用法都需要了解PKZIP应用说明中定义的格式。

此模块目前不处理多磁盘ZIP文件。它可以处理使用ZIP64扩展名的ZIP文件（即大小超过4 GiB的ZIP文件）。它支持在ZIP压缩文件中解密加密文件，但它目前不能创建加密文件。解密过程非常缓慢，因为它在本地Python而不是C中实现。

该模块定义了以下项目：

*异常*zipfile.BadZipFile
错误的ZIP文件引发的错误。

3.2版本中的新功能

*异常*zipfile.BadZipfile
BadZipFile与旧Python版本兼容的别名。

自3.2版以来已弃用。

*异常*zipfile.LargeZipFile
ZIP文件需要ZIP64功能但尚未启用时引发的错误。

类 zipfile.ZipFile
用于读写ZIP文件的类。有关构造函数的详细信息，请参见ZipFile Objects部分 。

类 zipfile.PyZipFile
用于创建包含Python库的ZIP存档的类。

*类*zipfile.ZipInfo（*文件名= 'NONAME'，DATE_TIME =（1980，1，1，0，0，0）*）
用于表示关于档案成员的信息的类。这个类的实例由对象的 方法getinfo()和infolist()方法返回ZipFile。该zipfile模块的大多数用户不需要创建这些，但只能使用由该模块创建的那些用户。*文件名*应该是档案成员的全名，并且 *date_time*应该是包含六个字段的元组，其描述对文件的最后修改的时间; 这些字段在ZipInfo对象中进行了描述 。

zipfile.is_zipfile（*文件名*）
返回True如果*文件名*是基于它的幻数有效的ZIP文件，否则回报False。 *文件名*也可以是文件或类似文件的对象。

*在版本3.1中更改：*支持文件和文件类对象。

zipfile.ZIP_STORED
未压缩的归档成员的数字常量。

zipfile.ZIP_DEFLATED
常用ZIP压缩方法的数字常量。这需要 zlib模块。

zipfile.ZIP_BZIP2
BZIP2压缩方法的数字常量。这需要 bz2模块。

3.3版本的新功能

zipfile.ZIP_LZMA
LZMA压缩方法的数字常量。这需要 lzma模块。

3.3版本的新功能

> **注意：** ZIP文件格式规范包含自2001年以来对bzip2压缩的支持，以及自2006年以来对LZMA压缩的支持。但是，某些工具（包括较老的Python版本）不支持这些压缩方法，可能会拒绝完全处理ZIP文件，或者无法提取单个文件。

也可以看看:

PKZIP应用笔记
由Phil Katz撰写的关于ZIP文件格式的文档，所使用的格式和算法的创建者。

Info-ZIP主页
有关Info-ZIP项目的ZIP归档程序和开发库的信息。

13.5.1。ZipFile对象

class zipfile.ZipFile（*file*，*mode ='r'*，*compression = ZIP_STORED*，*allowZip64 = True*）

打开一个ZIP文件，其中*文件*可以是文件（字符串），文件类型对象或路径类对象的路径。该*模式*参数应该是'r'读取现有的文件，'w'以截断并写入新文件，'a'追加到现有的文件，或'x'专门创建并写入新文件。如果*模式*是，'x'并且*文件*引用了现有文件，FileExistsError则会引发a。如果*模式*是，'a'并且*文件*引用了现有的ZIP文件，则会向其中添加其他文件。如果*文件*不会引用ZIP文件，那么会在文件中追加新的ZIP归档文件。这是为了将ZIP压缩文件添加到其他文件（如python.exe）。如果*模式*是，'a'并且该文件根本不存在，则会创建它。如果*模式*是'r'或者'a'，文件应该是可搜索的。*压缩*是书写时存档，并应使用ZIP压缩方法ZIP_STORED，ZIP_DEFLATED，ZIP_BZIP2或ZIP_LZMA; 无法识别的值会导致NotImplementedError提高。如果ZIP_DEFLATED，ZIP_BZIP2或者ZIP_LZMA被指定但相应的模块（zlib，bz2或lzma）不可用，则RuntimeError上升。默认是ZIP_STORED。如果*allowZip64*是True（默认的）zipfile将创建zip文件大于4 GiB时使用ZIP64扩展的ZIP文件。如果它是假的，zipfile则会在ZIP文件需要ZIP64扩展时引发异常。

如果文件是使用mode创建的'w'，'x'或者'a'然后closed不向档案添加任何文件，则将为该文件写入适用于空档案的相应ZIP结构。

ZipFile也是一个上下文管理器，因此支持该 with语句。在这个例子中，*myzip*在with语句套件完成后关闭 - 即使发生异常：

```
with ZipFile('spam.zip', 'w') as myzip:
    myzip.write('eggs.txt')
```

*3.2版*新增功能：添加了ZipFile用作上下文管理器的功能。

*在3.3版本改变：*增加了对支持bzip2和lzma压缩。

*在版本3.4中更改：*默认情况下启用ZIP64扩展。

*在版本3.5中进行了更改：*增加了对写入不可查看流的支持。增加了对'x'模式的支持。

*在版本3.6中更改：*以前，RuntimeError无法识别的压缩值引发了一个纯*文本*。

*改变在3.6.2版本：*该*文件*参数接受路径状物体。

ZipFile.**close**()

关闭档案文件。您必须close()在退出程序之前拨打电话，否则不会写入重要记录。

ZipFile.**getinfo**(*名字*)

ZipInfo用关于存档成员*名称的*信息返回一个对象 。调用getinfo()存档中当前不包含的名称将引发一个KeyError。

ZipFile.**infolist**()

返回包含ZipInfo每个档案成员的对象的列表。如果打开了现有的存档，则这些对象与磁盘上实际ZIP文件中的条目顺序相同。

ZipFile.**namelist**()

按名称返回存档成员列表。

ZipFile.**open**(*name*，*mode='r'*，*pwd = None*，***，*force_zip64 = False*)

以二进制文件对象的形式访问存档的成员。 *名称*可以是档案中的文件名称或ZipInfo 对象。所述*模式*参数，如果包括的话，必须是'r'（默认值）或'w'。 *pwd*是用于解密加密ZIP文件的密码。

open()也是上下文管理者，因此支持这样的 with陈述：

```
with ZipFile('spam.zip') as myzip:
    with myzip.open('eggs.txt') as myfile:
        print(myfile.read())
```

随着*模式*'r'的类文件对象（ ZipExtFile ）是只读，并提供了以下方法： read()， readline()， readlines()，__iter__()，__next__()。这些对象可以独立于ZipFile进行操作。

随着mode='w'，一个可写的文件句柄被返回，它支持该 write()方法。当可写文件句柄打开时，尝试读取或写入ZIP文件中的其他文件将引发一个 ValueError。

在写入文件时，如果文件大小事先未知，但可能会超过2 GiB，请通过force_zip64=True以确保文件头格式能够支持大文件。如果事先知道文件大小，则ZipInfo使用file_sizeset 构造一个对象，并将其用作*名称*参数。

注意： 的open()，read()和extract()方法可利用一个文件名或ZipInfo对象。尝试阅读包含具有重复名称的成员的ZIP文件时，您会明白这一点。

*在3.6版中更改：*删除了对mode='U'。使用io.TextIOWrapper在读压缩文本文件的通用换行模式。

*在版本3.6中更改：*open()现在可以使用该mode='w'选项将文件写入存档。

*在版本3.6中更改：*调用open()一个封闭的ZipFile将引发一个ValueError。以前，RuntimeError有人提出了。

ZipFile.**extract**(*member*, *path* = None, *pwd* = None)

将存档中的成员提取到当前工作目录; *成员*必须是其全名或ZipInfo对象。其文件信息尽可能准确地提取。*路径*指定要提取到的不同目录。*成员*可以是文件名或ZipInfo对象。*pwd*是用于加密文件的密码。

返回创建的规范化路径（目录或新文件）。

注意： 如果成员的文件名是绝对路径，驱动/ UNC SharePoint和领导（回）斜线将被剥离，如：///foo/bar成为 foo/bar在Unix，并C:\foo\bar成为foo\bar在Windows上。"."成员文件名中的所有组件都将被删除，例如：../../foo../../ba..r成为foo../ba..r。在Windows非法字符（:，<，>，|，"，?，并*用下划线代替）（_）。

*在版本3.6中更改：*调用extract()一个封闭的ZipFile将引发一个 ValueError。以前，RuntimeError有人提出了。

*改变在3.6.2版本：*该*路径*参数接受路径状物体。

ZipFile.**extractall**(*path* = None, *members* = None, *pwd* = None)

将存档中的所有成员提取到当前工作目录。*路径*指定要提取到的不同目录。*成员*是可选的，并且必须是返回列表的子集namelist()。*pwd*是用于加密文件的密码。

警告： 切勿在未经事先检查的情况下从不受信任的来源提取档案。有可能文件是在*路径*之外创建的，例如具有"/"以两个点开头的绝对文件名或具有两个点的文件名的成员"."。这个模块试图阻止这一点。见extract()注。

*在版本3.6中更改：*调用extractall()一个封闭的ZipFile将引发一个 ValueError。以前，RuntimeError有人提出了。

*改变在3.6.2版本：*该*路径*参数接受路径状物体。

ZipFile.**printdir**()

打印存档的目录到sys.stdout。

ZipFile.**setpassword**(*pwd*)

将*pwd*设置为默认密码以提取加密文件。

ZipFile.**read**(*名称*, *pwd* = None)

返回档案中文件*名称*的字节。 *name*是档案中文件的名称或ZipInfo对象。档案必须打开才能阅读或追加。*pwd*是用于加密文件的密码，如果指定，它将覆盖使用默认密码设置setpassword()。呼吁read()对使用比其他压缩方法的ZipFile ZIP_STORED，ZIP_DEFLATED，ZIP_BZIP2或ZIP_LZMA将引发NotImplementedError。如果相应的压缩模块不可用，也会出现错误。

*在版本3.6中更改：*调用read()一个封闭的ZipFile将引发一个ValueError。以前，RuntimeError有人提出了。

ZipFile.**testzip**()

阅读档案中的所有文件，并检查它们的CRC和文件头。返回第一个错误文件的名称，否则返回None。

*在版本3.6中更改：*调用testfile()一个封闭的ZipFile将引发一个 ValueError。以前，RuntimeError有人提出了。

ZipFile.**write**(*文件名*，*arcname =无*，*compress_type =无*)

写命名的文件*的文件名*到归档，给它存档名称 *arcname*（默认情况下，这将是相同*的文件名*，但没有一个驱动器号和与领先的路径分隔符删除）。如果给定，*compress_type*将为新条目 覆盖为*压缩*参数给构造函数指定的值。档案必须以模式打开'w'，'x'或者'a'。

> **注意：** ZIP文件没有官方文件名编码。如果你有unicode文件名，你必须在将它们传递给你所需的编码之前将它们转换为字节串write()。WinZip将所有文件名解释为在CP437中编码，也称为DOS Latin。

> **注意：** 档案名称应该与档案根目录相关，也就是说，它们不应该以路径分隔符开头。

> **注意：** 如果arcname（或者filename如果arcname未给出）包含空字节，则归档文件的名称将在空字节处截断。

*在版本3.6中更改：*调用write()使用模式'r'或关闭的ZipFile创建的ZipFile将引发一个ValueError。以前，RuntimeError有人提出了。

ZipFile.**writestr**(*zinfo_or_arcname*，*data*[，*compress_type*])

将字符串*数据*写入存档; *zinfo_or_arcname*是它将在档案中给出的文件名称，或者是一个ZipInfo实例。如果它是一个实例，至少必须给出文件名，日期和时间。如果是名称，则日期和时间设置为当前日期和时间。档案必须以模式打开'w'，'x'或'a'。

如果给定，*compress_type*将覆盖 为新条目的构造函数提供的*压缩*参数的值，或*覆盖 zinfo_or_arcname*（如果是ZipInfo实例）的值。

> **注意：** 当将ZipInfo实例作为*zinfo_or_arcname*参数传递时，所使用的压缩方法将是 给定实例的*compress_type*成员中指定的压缩方法ZipInfo。默认情况下， ZipInfo构造函数将此成员设置为ZIP_STORED。

*改变在3.2版本：*该*compress_type*参数。

*在版本3.6中更改：*调用writestr()使用模式'r'或关闭的ZipFile创建的ZipFile将引发一个ValueError。以前，RuntimeError有人提出了。

以下数据属性也可用：

ZipFile.**filename**
ZIP文件的名称。

ZipFile.**debug**
要使用的调试输出的级别。这可以从0（默认，无输出）到3（最多输出）设置。调试信息被写入 sys.stdout。

ZipFile.**comment**
与ZIP文件关联的评论文本。如果将注释分配给ZipFile使用模式创建的 实例'w'，'x'或者'a'这应该是不超过65535字节的字符串。比此更长的评论在close()被调用时会在书面存档中被截断。

13.5.2。PyZipFile对象

该PyZipFile构造函数将相同的参数 ZipFile构造器，以及一个附加参数，*优化*。

class zipfile.**PyZipFile**（*file*，*mode ='r'*，*compression = ZIP_STORED*，*allowZip64 = True*，*optimize = -1* ）
*新版本3.2：*在*优化*参数。

*在版本3.4中更改：*默认情况下启用ZIP64扩展。

实例除了这些ZipFile对象之外还有一个方法：

writepy（*路径名*，*basename =''*，*filterfunc = None* ）
搜索文件*.py并将相应的文件添加到存档。

如果没有给出*optimize*参数，PyZipFile或者-1相应的文件是*.pyc文件，则在必要时进行编译。

如果将*优化*参数设置PyZipFile为0，1或者 2仅将具有该优化级别的文件（请参阅compile()）添加到归档中，则在必要时进行编译。

如果*路径名*是一个文件，则文件名必须以.py，并且只有（相应的*.pyc）文件被添加到顶层（无路径信息）。如果*路径*是不是结束一个文件 .py，一个RuntimeError将提高。如果它是一个目录，并且该目录不是一个包目录，则所有文件 *.pyc都将添加到顶层。如果该目录是一个包目录，则所有*.pyc这些目录都将作为文件路径添加到包名下，如果有任何子目录是包目录，则所有这些都将递归添加。

*basename*仅供内部使用。

filterfunc，如果给出的话，必须是一个采用单个字符串参数的函数。在将其添加到存档之前，它将通过每个路径（包括每个单独的完整文件路径）。如果*filterfunc*返回一个假

值，则不会添加路径，并且如果它是目录，则其内容将被忽略。例如，如果我们的测试文件都在test目录中或者以字符串开头test_，我们可以使用 *filterfunc*来排除它们：

```
>>> zf = PyZipFile('myprog.zip')
>>> def notests(s):
...     fn = os.path.basename(s)
...     return (not (fn == 'test' or fn.startswith('test_')))
>>> zf.writepy('myprog', filterfunc=notests)
```

该writepy()方法使用以下文件名进行归档：

```
string.pyc                      # Top level name
test/__init__.pyc               # Package directory
test/testall.pyc                # Module test.testall
test/bogus/__init__.pyc         # Subpackage directory
test/bogus/myfile.pyc           # Submodule test.bogus.myfile
```

*在新版本3.4：*在*filterfunc*参数。

*改变在3.6.2版本：*该*路径*参数接受路径状物体。

13.5.3。ZipInfo对象

ZipInfo类的实例由对象的方法getinfo()和 infolist()方法返回ZipFile。每个对象存储有关ZIP存档的单个成员的信息。

有一个classmethod ZipInfo为文件系统文件创建一个实例：

classmethod ZipInfo.**from_file**(*filename*，*arcname* = *None*)

ZipInfo为文件系统上的文件构建一个实例，准备将其添加到zip文件。

*文件名*应该是文件系统上文件或目录的路径。

如果指定了*arcname*，则将其用作存档中的名称。如果未指定*arcname*，则名称将与*文件名*相同，但会删除任何驱动器号和前导路径分隔符。

3.6版本中的新功能。

*改变在3.6.2版本：*该*文件名*参数接受路径状物体。

实例具有以下方法和属性：

ZipInfo.**is_dir**( )

True如果此存档成员是目录，则返回。

这使用条目的名称：目录应始终以/。

3.6版本中的新功能。

ZipInfo.**filename**

归档文件的名称。

ZipInfo.**date_time**

上次修改归档成员的时间和日期。这是一个包含六个值的元组：

指数	值
0	年份（> = 1980年）
1	月份（基于单位）
2	一个月中的日（基于一）
3	小时（从零开始）
4	分钟（从零开始）
5	秒（零基）

注意： 1980年以前，ZIP文件格式不支持时间戳。

ZipInfo.**compress_type**

归档成员的压缩类型。

ZipInfo.**comment**

评论个人档案成员。

ZipInfo.**extra**

扩展字段数据。在PKZIP应用笔记包含包含在该字符串数据的内部结构的一些意见。

ZipInfo.**create_system**

创建ZIP档案的系统。

ZipInfo.**create_version**

创建ZIP压缩文件的PKZIP版本。

ZipInfo.**extract_version**

PKZIP版本需要提取归档。

ZipInfo.**reserved**

必须为零。

ZipInfo.**flag_bits**

ZIP标志位。

ZipInfo.**volume**

文件头的卷号。

ZipInfo.**internal_attr**

内部属性。

ZipInfo.**external_attr**

外部文件属性。

ZipInfo.header_offset
字节偏移到文件头。

ZipInfo.CRC
未压缩文件的CRC-32。

ZipInfo.compress_size
压缩数据的大小。

ZipInfo.file_size
未压缩文件的大小。

13.5.4。命令行界面

该zipfile模块提供了一个简单的命令行界面来与ZIP档案进行交互。

如果您想创建一个新的ZIP存档，请在-c 选项后指定其名称，然后列出应包含的文件名：

```
$ python -m zipfile -c monty.zip spam.txt eggs.txt
```

传递一个目录也是可以接受的：

```
$ python -m zipfile -c monty.zip life-of-brian_1979/
```

如果要将ZIP压缩文件解压缩到指定的目录中，请使用以下-e选项：

```
$ python -m zipfile -e monty.zip target-dir/
```

有关ZIP归档文件的列表，请使用以下-l选项：

```
$ python -m zipfile -l monty.zip
```

13.5.4.1。命令行选项

-l <zipfile>
列出zipfile文件。

-c <zipfile> <source1> ... <sourceN>
从源文件创建zipfile。

-e <zipfile> <output_dir>
将zipfile解压缩到目标目录中。

-t <zipfile>
测试zip文件是否有效。

13.6。 tarfile- 读写tar档案文件

源代码： Lib / tarfile.py

该tarfile模块可以读取和写入tar档案，包括使用gzip，bz2和lzma压缩的档案。使用该zipfile模块读取或写入.zip文件，或shutil中的更高级别的函数。

一些事实和数字：

- 读取和写入gzip，bz2并lzma压缩归档如果各个模块是可用的。
- 读/写支持POSIX.1-1988（ustar）格式。
- 对GNU tar格式的读/写支持，包括长*名称*和长*链接*扩展，只读支持*稀疏*扩展的所有变体，包括稀疏文件的恢复。
- 读/写支持POSIX.1-2001（pax）格式。
- 处理目录，常规文件，硬链接，符号链接，fifo，字符设备和块设备，并能够获取和恢复文件信息，如时间戳，访问权限和所有者。

*版本3.3中更改：*增加了对lzma压缩的支持。

tarfile.open（*name = None*，*mode ='r'*，*fileobj = None*，*bufsize = 10240*，*** kwargs* ）

TarFile为路径*名称*返回一个对象。有关TarFile允许的对象和关键字参数的详细信息，请参阅TarFile对象。

*模式*必须是表单的字符串'filemode[:compression]'，它默认为'r'。以下是模式组合的完整列表：

模式	行动
'r' or 'r:*'	透明压缩打开（推荐）。
'r:'	打开阅读专门没有压缩。
'r:gz'	用gzip压缩打开。
'r:bz2'	用bzip2压缩打开阅读。
'r:xz'	打开阅读lzma压缩。
'x' 要么 'x:'	创建一个tar文件而不压缩。FileExistsError如果它已经存在，则引发异常。
'x:gz'	用gzip压缩创建一个tar文件。FileExistsError如果它已经存在，则引发异常。
'x:bz2'	用bzip2压缩创建一个tar文件。FileExistsError如果它已经存在，则引发异常。
'x:xz'	用lzma压缩创建一个tar文件。FileExistsError如果它已经存在，则引发异常。
'a' or 'a:'	打开后不需要压缩。如果该文件不存在，则会创建该文件。
'w' or 'w:'	打开未压缩的文字。
'w:gz'	打开gzip压缩文字。

模式	行动
'w:bz2'	打开bzip2压缩文字。
'w:xz'	打开lzma压缩文字。

请注意'a:gz'，'a:bz2'或'a:xz'不可能。如果*模式*不适合打开某个（压缩）的文件进行读取， ReadError则会引发。使用*模式* 'r'来避免这种情况。如果不支持压缩方法，CompressionError则会引发。

如果指定了*fileobj*，那么它将用作 以二进制模式打开的*名称*文件对象的替代方案。它应该在位置0。

对于模式'w:gz'，'r:gz'，'w:bz2'，'r:bz2'，'x:gz'，'x:bz2'，tarfile.open()接受关键字参数 *compresslevel*（默认9）来指定该文件的压缩级别。

对于特殊用途，对于第二格式*模式*：'filemode|[compression]'。 tarfile.open()将返回一个TarFile 对象，它将数据作为一个块流进行处理。文件不会随意查找。如果给定，*fileobj*可能是任何具有read()或write()方法的对象 （取决于*模式*）。*bufsize* 指定块大小，默认为字节。将此变体与例如套接字文件对象或磁带设备结合使用。但是，这样的对象是有限的，因为它不允许随机访问，请参阅示例。目前可能的模式：20 * 512sys.stdinTarFile

模式	行动
'r|*'	透明压缩打开一个tar块*流*供阅读。
'r|'	打开一个未压缩的焦油块*流*供阅读。
'r|gz'	打开一个gzip压缩*流*供阅读。
'r|bz2'	打开一个bzip2压缩*流*供阅读。
'r|xz'	打开一个lzma压缩*流*供阅读。
'w|'	写一个未压缩的*流*。
'w|gz'	打开gzip压缩*流*进行写入。
'w|bz2'	打开一个bzip2压缩*流*进行写入。
'w|xz'	打开一个lzma压缩*流*进行写入。

*改变在3.5版本：*将'x'加入（独家创作）模式。

*改变在3.6版本：*该*名称*参数接受路径状物体。

*类*tarfile.TarFile

阅读和编写tar档案的类。不要直接使用这个类：tarfile.open()改为使用。请参阅TarFile对象。

tarfile.is_tarfile(*名字*)

True如果*名称*是一个tar档案文件，则返回该tarfile 模块可以读取的内容。

该tarfile模块定义了以下例外情况：

*异常*tarfile.TarError
所有tarfile例外的基类。

*异常*tarfile.ReadError
当tar档案被打开时引发，或者不能被tarfile模块处理 或者以某种方式无效。

*异常*tarfile.CompressionError
当不支持压缩方法或无法正确解码数据时引发。

*异常*tarfile.StreamError
针对流式TarFile 物体的典型局限性而提出。

*异常*tarfile.ExtractError
在使用时引发*非致命*错误TarFile.extract()，但仅 在使用时引发。 TarFile.errorlevel== 2

*异常*tarfile.HeaderError
TarInfo.frombuf()如果获取的缓冲区无效，则会引发此问题。

以下常量在模块级别可用：

tarfile.ENCODING
默认字符编码：'utf-8'在Windows上， sys.getfilesystemencoding()否则返回值 。

以下每个常量都定义了tarfile模块能够创建的tar归档格式 。有关详细信息，请参阅支持的焦油格式部分

tarfile.USTAR_FORMAT
POSIX.1-1988（ustar）格式。

tarfile.GNU_FORMAT
GNU tar格式。

tarfile.PAX_FORMAT
POSIX.1-2001（pax）格式。

tarfile.DEFAULT_FORMAT
创建档案的默认格式。这是目前GNU_FORMAT。

也可以看看:

模 zipfile
zipfile标准模块的文档。

归档操作
由标准shutil模块提供的更高级归档工具的文档。

GNU tar手册，基本Tar格式
tar归档文件的文档，包括GNU tar扩展。

13.6.1。TarFile对象

该TarFile对象提供了一个tar档案的接口。tar档案是一系列块。一个档案成员（一个存储文件）由一个头块和数据块组成。可以将文件存储在tar归档文件中多次。每个存档成员都由一个TarInfo 对象表示，有关详细信息，请参阅TarInfo对象。

一个TarFile对象可以在with 语句中用作上下文管理器。块完成后它会自动关闭。请注意，如果发生例外情况，开放撰写的档案将不会最终确定; 只有内部使用的文件对象将被关闭。有关用例，请参阅 示例部分。

*3.2版*新增功能：增加了对上下文管理协议的支持。

*类*tarfile. TarFile（*名=无*，*模式= 'R'*，*FileObj文件=无*，*格式= DEFAULT_FORMAT*，*tarinfo = TarInfo*，*解除引用=假*，*ignore_zeros =假*，*编码= ENCODING*，*误差= 'surrogateescape'*，*pax_headers =无*，*调试= 0*，*errorlevel = 0* ）

以下所有参数都是可选的，并且可以作为实例属性来访问。

*name*是存档的路径名。*名称*可能是一个类似于路径的对象。如果给出*fileobj*，它可以被忽略。在这种情况下，如果文件对象name存在，则使用该文件对象的属性。

*模式*是'r'从现有存档读取'a'数据，将数据附加到现有文件，'w'创建覆盖现有文件的新文件，或'x'仅在尚不存在的情况下创建新文件。

如果给出*fileobj*，它将用于读取或写入数据。如果可以确定，*模式*会被*fileobj*的模式覆盖。将从位置0使用*fileobj*。

> **注意：** 关闭时*fileobj*未TarFile关闭。

*格式*控制存档格式。它必须是常量之一 USTAR_FORMAT ，GNU_FORMAT或者PAX_FORMAT是在模块级定义的。

该*tarinfo*参数可以用来替换缺省TarInfo使用不同的一类。

如果*解除引用*，则为False档案添加符号和硬链接。如果是True ，请将目标文件的内容添加到存档中。这对不支持符号链接的系统没有影响。

如果*ignore_zeros*是False ，*则将*空块视为归档的结尾。如果是这样True ，跳过空（和无效）块，尽量争取尽可能多的成员。这仅用于阅读级联或损坏的档案。

*调试*可以设置从0（无调试消息）到3（所有调试消息）。消息被写入sys.stderr。

如果错误*级别*是0，则使用时将忽略所有错误TarFile.extract()。不过，在调试启用时，它们在调试输出中显示为错误消息。如果1所有*致命*错误都作为OSError 例外提出。如果2所有*非致命*错误都作为TarError 例外提出。

该*编码*和*错误*参数定义用于读取或写入档案，以及如何转换错误将要处理的字符编码。默认设置适用于大多数用户。有关详细信息，请参阅Unicode问题部分。

所述*pax_headers*参数是将要添加作为PAX全局头如果串的一个可选的字典*的格式*是PAX_FORMAT。

*在3.2版本中更改：*使用'surrogateescape'作为默认的*错误*说法。

*改变在3.5版本：*将'x'加入（独家创作）模式。

*改变在3.6版本：*该*名称*参数接受路径状物体。

classmethod TarFile.open(...)

替代构造函数。这个tarfile.open()函数实际上是这个classmethod的一个捷径。

TarFile.getmember(*名字*)

返回一个TarInfo成员*名称*的对象。如果在存档中找不到*名称*，KeyError则会提出。

> **注意：** 如果一个成员在存档中多次出现，则认为其最后一次出现是最新的版本。

TarFile.getmembers()

将档案成员作为TarInfo对象列表返回。该列表与归档中的成员具有相同的顺序。

TarFile.getnames()

返回成员名单。它与返回的列表具有相同的顺序getmembers()。

TarFile.list(*verbose = True*, *, *members = None*)

打印目录到sys.stdout。如果*详细*是False，则仅打印成员的名称。如果是True，则产生与**ls -l**类似的输出。如果给出可选*成员*，它必须是返回列表的子集getmembers()。

*版本3.5中已更改：*添加了*成员*参数。

TarFile.next()

TarInfo当TarFile打开阅读时，将档案的下一个成员作为对象 返回。None如果没有更多可用，则返回。

TarFile.extractall(*path =“。”*, *members = None*, *, *numeric_owner = False*)

将存档中的所有成员提取到当前工作目录或目录*路径*。如果给出可选*成员*，它必须是返回列表的子集getmembers()。在提取所有成员之后设置所有者，修改时间和权限等目录信息。这样做是为了解决两个问题：每次在其中创建文件时，都会重置目录的修改时间。而且，如果一个目录的权限不允许写入，解压文件将失败。

如果*numeric_owner*是True，则tarfile中的uid和gid数字用于设置提取文件的所有者/组。否则，使用tarfile中的命名值。

> **警告：** 切勿在未经事先检查的情况下从不受信任的来源提取档案。有可能文件是在*路径*之外创建的，例如具有"/"以两个点开头的绝对文件名或具有两个点的文件名的成员".."。

*版本3.5中已更改：*添加了*numeric_owner*参数。

*改变在3.6版本：*该*路径*参数接受路径状物体。

TarFile.**extract**(*member*，*path* =“”，*set_attrs* = *True*，*，*numeric_owner* = *False*)

使用其全名将存档中的成员提取到当前工作目录。其文件信息尽可能准确地提取。*成员* 可能是文件名或TarInfo对象。您可以使用*路径*指定不同的目录。*路径*可能是一个类似于路径的对象。文件属性（owner，mtime，mode）被设置，除非*set_attrs*为false。

如果*numeric_owner*是True，则tarfile中的uid和gid数字用于设置提取文件的所有者/组。否则，使用tarfile中的命名值。

> **注意：** 该extract()方法不考虑几个提取问题。在大多数情况下，您应该考虑使用该extractall()方法。

> **警告：** 请参阅警告extractall()。

*版本3.2中已更改：*添加了*set_attrs*参数。

*版本3.5中已更改：*添加了*numeric_owner*参数。

*改变在3.6版本：*该*路径*参数接受路径状物体。

TarFile.**extractfile**(*成员*)

从存档中提取成员作为文件对象。*成员*可能是文件名或TarInfo对象。如果*成员*是常规文件或链接，io.BufferedReader则返回一个 对象。否则，None返回。

*版本3.3中更改：*返回一个io.BufferedReader对象。

TarFile.**add**(*name*，*arcname* = *None*，*recursive* = *True*，*exclude* = *None*，*，*filter* = *None*)

将文件*名*添加到存档。*名称*可以是任何类型的文件（目录，fifo，符号链接等）。如果给定，则*arcname*为档案中的文件指定替代名称。目录默认递归添加。这可以通过设置*递归*来 避免False。如果给出了*排除*，它必须是一个函数，它接受一个文件名参数并返回一个布尔值。根据这个值，相应的文件被排除（True）或添加（False）。如果指定了*过滤器*，它必须是关键字参数。它应该是一个接受TarInfo对象参数并返回已更改TarInfo对象的函数。如果它返回 None该TarInfo对象将从档案中排除。见实施例中的示例。

*在版本3.2中更改：*添加了*过滤器*参数。

*自从3.2版本不推荐使用：*在*排除*参数已被弃用，请使用*过滤器*参数来代替。

TarFile.**addfile**(*tarinfo*，*fileobj* = *None*)

将TarInfo对象*tarinfo*添加到存档。如果给出*fileobj*，它应该是一个二进制文件，并tarinfo.size从中读取字节并将其添加到存档中。您可以TarInfo直接创建对象，也可以使用gettarinfo()。

TarFile.**gettarinfo**(*name* = *None*，*arcname* = *None*，*fileobj* = *None*)

根据现有文件TarInfo的结果os.stat()或等效项创建一个对象。该文件可以按*名称命名*，也可以用文件描述符 指定为文件对象 *fileobj*。 *名称*可能是一个类似于路径的对象。如果给

定，*arcname*为档案中的文件指定替代名称，否则，该名称取自*fileobj*的 name属性或*名称*参数。该名称应该是一个文本字符串。

您可以TarInfo在添加它之前修改其中的一些属性addfile()。如果文件对象不是位于文件开头的普通文件对象，则size可能需要修改等属性。这是对象的情况，例如GzipFile。的name也可以被修饰，在这种情况下*arcname* 可能是一个虚设字符串。

*改变在3.6版本：*该*名称*参数接受路径状物体。

TarFile. close ( )
关闭TarFile。在写入模式下，两个完成零块被附加到存档。

TarFile. pax_headers
包含pax全局标题的键值对的字典。

13.6.2。TarInfo对象

一个TarInfo对象表示a中的一个成员TarFile。除了存储文件的所有必需属性（如文件类型，大小，时间，权限，所有者等）之外，它还提供了一些有用的方法来确定它的类型。它*不*包含该文件的数据本身。

TarInfo对象由TarFile方法 返回getmember()，getmembers()并且gettarinfo()。

class tarfile. TarInfo (*name =“”*)
创建一个TarInfo对象。

classmethod TarInfo. frombuf (*buf*，*encoding*，*errors*)
TarInfo从字符串缓冲区*buf*创建并返回一个对象。

HeaderError如果缓冲区无效则引发。

classmethod TarInfo. fromtarfile (*tarfile*)
从TarFile对象*tarfile*中读取下一个成员，并将其作为TarInfo对象返回。

TarInfo. tobuf (*format = DEFAULT_FORMAT*，*encoding = ENCODING*，*errors ='surrogateescape'*)
从一个TarInfo对象创建一个字符串缓冲区。有关参数的信息，请参阅TarFile该类的构造函数。

*在3.2版本中更改：*使用’surrogateescape’作为默认的*错误*说法。

一个TarInfo对象具有以下公共数据属性：

TarInfo. name
档案成员的名称。

TarInfo. size
字节大小。

TarInfo.**mtime**
上次修改时间。

TarInfo.**mode**
权限位。

TarInfo.**type**
文件类型。 *类型*通常是这些常量之一：REGTYPE， AREGTYPE，LNKTYPE，SYMTYPE，DIRTYPE， FIFOTYPE，CONTTYPE，CHRTYPE，BLKTYPE， GNUTYPE_SPARSE。要TarInfo更方便地确定对象的类型，请使用is*()下面的方法。

TarInfo.**linkname**
目标文件名称的名称，它仅存TarInfo在于类型LNKTYPE和对象中SYMTYPE。

TarInfo.**uid**
最初存储此成员的用户的用户标识。

TarInfo.**gid**
最初存储此成员的用户的组ID。

TarInfo.**uname**
用户名。

TarInfo.**gname**
团队名字。

TarInfo.**pax_headers**
包含关联的pax扩展标题的键值对的字典。

一个TarInfo对象还提供了一些方便的查询方法：

TarInfo.**isfile**( )
True如果Tarinfo对象是常规文件，则返回。

TarInfo.**isreg**( )
和...一样isfile()。

TarInfo.**isdir**( )
True如果它是目录，则返回。

TarInfo.**issym**( )
True如果它是符号链接，则返回。

TarInfo.**islnk**( )
True如果它是一个硬链接，则返回。

TarInfo.**ischr**( )
True如果它是一个字符设备，则返回。

TarInfo.**isblk**( )
    True如果它是块设备则返回。

TarInfo.**isfifo**( )
    True如果它是FIFO，则返回。

TarInfo.**isdev**( )
    True如果是字符设备，块设备或FIFO之一，则返回。

13.6.3。命令行界面

3.4版新增功能

该tarfile模块提供了一个简单的命令行界面来与tar档案进行交互。

如果你想创建一个新的tar档案，在-c 选项后面指定它的名字，然后列出应该包含的文件名：

```
$ python -m tarfile -c monty.tar  spam.txt eggs.txt
```

传递一个目录也是可以接受的：

```
$ python -m tarfile -c monty.tar life-of-brian_1979/
```

如果您想将tar归档文件解压缩到当前目录中，请使用以下-e选项：

```
$ python -m tarfile -e monty.tar
```

您也可以通过传递目录名称将tar归档文件解压缩到其他目录中：

```
$ python -m tarfile -e monty.tar  other-dir/
```

有关tar归档文件的列表，请使用以下-l选项：

```
$ python -m tarfile -l monty.tar
```

13.6.3.1。命令行选项

-l <tarfile>
--list <tarfile>
    列出tarfile中的文件。

-c <tarfile> <source1> ... <sourceN>
--create <tarfile> <source1> ... <sourceN>
    从源文件创建tarfile。

-e <tarfile> [<output_dir>]
--extract <tarfile> [<output_dir>]

如果未指定*output_dir*，则将tarfile解压缩到当前目录中。

`-t <tarfile>`
`--test <tarfile>`
测试tarfile是否有效。

`-v, --verbose`
详细输出。

13.6.4。示例

如何提取整个tar档案到当前工作目录：

```
import tarfile
tar = tarfile.open("sample.tar.gz")
tar.extractall()
tar.close()
```

如何`TarFile.extractall()`使用生成器函数而不是列表提取tar存档的子集：

```
import os
import tarfile

def py_files(members):
    for tarinfo in members:
        if os.path.splitext(tarinfo.name)[1] == ".py":
            yield tarinfo

tar = tarfile.open("sample.tar.gz")
tar.extractall(members=py_files(tar))
tar.close()
```

如何从文件名列表中创建一个未压缩的tar归档文件：

```
import tarfile
tar = tarfile.open("sample.tar", "w")
for name in ["foo", "bar", "quux"]:
    tar.add(name)
tar.close()
```

使用`with`声明的同一个例子：

```
import tarfile
with tarfile.open("sample.tar", "w") as tar:
    for name in ["foo", "bar", "quux"]:
        tar.add(name)
```

如何读取gzip压缩的tar档案并显示一些成员信息：

```
import tarfile
tar = tarfile.open("sample.tar.gz", "r:gz")
```

```
for tarinfo in tar:
    print(tarinfo.name, "is", tarinfo.size, "bytes in size and is", end="")
    if tarinfo.isreg():
        print("a regular file.")
    elif tarinfo.isdir():
        print("a directory.")
    else:
        print("something else.")
tar.close()
```

如何使用以下*过滤器*参数创建归档并重置用户信息`TarFile.add()`：

```
import tarfile
def reset(tarinfo):
    tarinfo.uid = tarinfo.gid = 0
    tarinfo.uname = tarinfo.gname = "root"
    return tarinfo
tar = tarfile.open("sample.tar.gz", "w:gz")
tar.add("foo", filter=reset)
tar.close()
```

13.6.5。支持的tar格式

有三种可以使用该`tarfile`模块创建的tar格式：

- POSIX.1-1988 ustar格式（`USTAR_FORMAT`）。它支持的文件名长度最多为256个字符，链接名称最多为100个字符。最大文件大小为8 GiB。这是一个古老而有限但广泛支持的格式。

- GNU tar格式（`GNU_FORMAT`）。它支持长文件名和链接名称，大于8 GiB的文件和稀疏文件。它是GNU / Linux系统上的事实标准。`tarfile`完全支持长名称的GNU tar扩展，稀疏文件支持是只读的。

- POSIX.1-2001 pax格式（`PAX_FORMAT`）。它是最灵活的格式，几乎没有限制。它支持长文件名和链接名称，大文件和便携式存储路径名。但是，今天并非所有的tar实现都能够正确处理pax归档。

  在*PAX*格式的扩展现有*的ustar*格式。它使用额外的头部来存储不能以其他方式存储的信息。有两种类型的pax头文件：扩展头文件只影响后续文件头文件，全局头文件对整个存档文件有效并影响所有后续文件。出于便携性原因，pax标头中的所有数据都以*UTF-8*编码。

还有一些可以读取但不能创建的tar格式变体：

- 古代的V7格式。这是Unix第七版的第一个tar格式，只存储常规文件和目录。名称不能超过100个字符，没有用户/组名称信息。某些档案在包含非ASCII字符的字段中错误计算标题校验和。
- SunOS tar扩展格式。这种格式是POSIX.1-2001 pax格式的变体，但不兼容。

13.6.6。Unicode问题

tar格式最初被设想为在磁带驱动器上进行备份，主要侧重于保存文件系统信息。现在，tar档案通常用于文件分发和在网络上交换档案。原始格式（这是所有其他格式的基础）的一个问题是，没有支持不同字符编码的概念。例如，在*UTF-8*系统上创建的普通tar归档文件如果包含非*ASCII文本*，则无法在*Latin-1*系统上正确读取字符。文本元数据（如文件名，链接名称，用户/组名称）将显示损坏。不幸的是，没有办法自动检测存档的编码。pax格式旨在解决此问题。它使用通用字符编码*UTF-8*存储非ASCII元数据。

字符转换的细节tarfile由类的*编码*和*错误*关键字参数控制 TarFile。

*编码*定义了用于归档中元数据的字符编码。默认值是sys.getfilesystemencoding()或'ascii'作为后备。根据档案是读取还是写入，元数据必须被解码或编码。如果*编码*设置不当，此转换可能会失败。

所述*误差*参数定义字符被如何处理不能被转换。错误处理程序部分列出了可能的值。默认方案是'surrogateescape' Python也用于其文件系统调用的，参见文件名，命令行参数和环境变量。

在PAX_FORMAT档案的情况下，通常不需要*编码*，因为所有的元数据都是使用*UTF-8*存储的。*编码*仅用于二进制pax头文件解码或存储代理字符串的极少数情况。

14.文件格式

本章描述的模块解析各种各样的文件格式，它们不是标记语言，并且与电子邮件无关。

14.1。 csv- CSV文件读取和写入

源代码： Lib / csv.py

所谓的CSV（逗号分隔值）格式是电子表格和数据库最常用的导入和导出格式。在尝试以标准化的方式描述格式之前，CSV格式已经使用了多年 **RFC 4180**。缺乏明确的标准意味着不同应用程序生成和使用的数据中经常存在细微的差异。这些差异可能会令人讨厌从多个来源处理CSV文件。尽管分隔符和引用字符有所不同，但整体格式足够相似，可以编写一个可以高效处理这些数据的模块，从编程器中隐藏读取和写入数据的细节。

该csv模块实现了以CSV格式读取和写入表格数据的类。它允许程序员说出“用Excel首选的格式写这些数据”，或者“从Excel生成的这个文件中读取数据”，而不知道Excel使用的CSV格式的确切细节。程序员还可以描述其他应用程序可以理解的CSV格式，或者定义他们自己的专用CSV格式。

该csv模块reader和writer对象读取和写入序列。程序员也可以使用DictReader和DictWriter类以字典形式读写数据。

也可以看看:

PEP 305 - CSV文件API
Python增强建议提出了Python的这一增加。

14.1.1。模块内容

该csv模块定义了以下功能：

csv.reader（ *csvfile* ， *dialect ='excel'* ， *** fmtparams* ）
返回将在给定的*csv文件中*遍历行的reader对象。 *csvfile*可以是任何支持迭代器协议的对象，每次__next__()调用它的方法时都会返回一个字符串- 文件对象和列表对象都适用。如果*csvfile*是一个文件对象，它应该打开newline=''。[1]可以给出 可选的 *方言*参数，用于定义特定CSV方言的一组参数。它可能是Dialect该类的子类的实例或list_dialects()函数返回的一个字符串 。其他可选的*fmtparams*可以给出关键字参数来覆盖当前方言中的单个格式参数。有关方言和格式参数的完整详细信息，请参见方言和格式参数一节。

从csv文件读取的每一行都以字符串列表形式返回。除非QUOTE_NONNUMERIC指定了格式选项（在这种情况下未加引号的字段转换为浮点数），否则不会执行自动数据类型转换。

一个简短的用法示例：

```
>>> import csv
>>> with open('eggs.csv', newline='') as csvfile:
...     spamreader = csv.reader(csvfile, delimiter=' ', quotechar='|')
...     for row in spamreader:
...         print(', '.join(row))
```

```
Spam, Spam, Spam, Spam, Spam, Baked Beans
Spam, Lovely Spam, Wonderful Spam
```

csv.**writer**(*csvfile*, *dialect='excel'*, ** *fmtparams*)

返回一个编写器对象，负责将用户的数据转换为给定类文件对象上的分隔字符串。 *csvfile* 可以是具有write()方法的任何对象 。如果*csvfile*是一个文件对象，应该用newline='' [1] 打开 。 可以给出可选的*方言*参数，该参数用于定义特定CSV方言的一组参数。它可能是Dialect该类的子类的实例 或list_dialects()函数返回的一个字符串 。其他可选的*fmtparams*关键字参数可以用来覆盖当前方言中的单个格式参数。有关方言和格式参数的完整详细信息，请参见部分方言和格式参数。为了尽可能简化与实现DB API的模块的接口，该值None被写为空字符串。虽然这不是可逆的转换，但它可以更轻松地将SQL NULL数据值转储到CSV文件，而无需预处理从cursor.fetch*调用返回的数据。所有其他非字符串数据str()在写入之前都会被字符串化。

一个简短的用法示例：

```
import csv
with open('eggs.csv', 'w', newline='') as csvfile:
    spamwriter = csv.writer(csvfile, delimiter=' ',
                            quotechar='|', quoting=csv.QUOTE_MINIMAL)
    spamwriter.writerow(['Spam'] * 5 + ['Baked Beans'])
    spamwriter.writerow(['Spam', 'Lovely Spam', 'Wonderful Spam'])
```

csv.**register_dialect**(*name*[, *dialect*[, ** *fmtparams*]])

将*方言*与*名字联系起来*。 *名称*必须是字符串。方言可以通过传递子类Dialect或*fmtparams*关键字参数或两者来指定，其中关键字参数覆盖方言的参数。有关方言和格式参数的完整详细信息，请参见方言和格式参数一节。

csv.**unregister_dialect**(*名字*)

从方言注册表中删除与*姓名*关联的方言。Error如果*名称*不是已注册的方言名称，则引发An 。

csv.**get_dialect**(*名字*)

返回与*名称*关联的方言。Error如果*名称*不是已注册的方言名称，则引发 An 。这个函数返回一个不可变的 Dialect。

csv.**list_dialects**()

返回所有注册方言的名称。

csv.**field_size_limit**([*new_limit*])

返回解析器允许的当前最大字段大小。如果给出*new_limit*，则这成为新的限制。

该csv模块定义了以下类：

class csv.**DictReader**(*f*, *fieldnames = None*, *restkey = None*, *restval = None*, *dialect='excel'*, * *args*, ** *kwds*)

创建一个像普通阅读器一样运行的对象，但将每行中的信息映射到OrderedDict 由可选的*fieldnames*参数提供的键。

的*字段名*的参数是一个序列。如果省略*字段*名称，则文件*f*第一行中的值将用作字段名称。无论字段名是如何确定的，有序字典都会保留它们的原始顺序。

如果一行的字段数大于字段名，剩余的数据将放入一个列表中并与*restkey*指定的字段*名*（默认为None）一起存储。如果非空白行的字段少于字段名称，缺少的值将填入None。

所有其他可选参数或关键字参数都传递给底层 reader实例。

*在版本3.6中更改：*返回的行现在是类型OrderedDict。

一个简短的用法示例：

```
>>> import csv
>>> with open('names.csv', newline='') as csvfile:
...     reader = csv.DictReader(csvfile)
...     for row in reader:
...         print(row['first_name'], row['last_name'])
...
Eric Idle
John Cleese

>>> print(row)
OrderedDict([('first_name', 'John'), ('last_name', 'Cleese')])
```

class csv. DictWriter (*f* , *fieldnames* , *restval =''* , *extrasaction ='raise'* , *dialect ='excel'* , ** args* , *** kwds*)

创建一个像普通作家一样操作的对象，但将字典映射到输出行。的*字段名*的参数是一个sequence标识，其中在传递给字典值的顺序按键的writerow()方法被写入到文件 *F*。可选的*restval*参数指定字典在*字段*名称中缺少键时要写入的值。如果传递给writerow()方法的字典包含在*字段名中*找不到的键 ，则可选的*extrasaction*参数指示要执行的操作。如果设置为'raise'，ValueError 则会引发默认值a 。如果它设置为'ignore'，字典中的额外值将被忽略。任何其他可选参数或关键字参数都会传递给底层 writer实例。

请注意，与DictReader类不同，该*字段的fieldnames*参数DictWriter不是可选的。由于Python的dict 对象没有排序，因此没有足够的信息来推断行应该写入文件*f*的顺序。

一个简短的用法示例：

```
import csv

with open('names.csv', 'w', newline='') as csvfile:
    fieldnames = ['first_name', 'last_name']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

    writer.writeheader()
    writer.writerow({'first_name': 'Baked', 'last_name': 'Beans'})
    writer.writerow({'first_name': 'Lovely', 'last_name': 'Spam'})
    writer.writerow({'first_name': 'Wonderful', 'last_name': 'Spam'})
```

*类*csv. Dialect

的Dialect类是依赖于主要用于它的属性，这是用来定义一个特定的参数的容器类 reader或writer实例。

*类*csv. excel

在excel类定义的Excel生成CSV文件的通常的性质。它用方言名称注册'excel'。

*类*csv. excel_tab

所述excel_tab类定义Excel生成的制表符分隔的文件的通常的性质。它用方言名称注册'excel-tab'。

*类*csv. unix_dialect

的unix_dialect类定义在UNIX系统上，即，使用生成的CSV文件的通常性质'\n'如线路终端机和引用的所有字段。它用方言名称注册'unix'。

3.2版本中的新功能

*类*csv. Sniffer

本Sniffer类用来推断一个CSV文件的格式。

本Sniffer类提供了两个方法：

sniff（*sample*，*delimiters* = *None*）

分析给定*样本*并返回Dialect反映所发现参数的子类。如果给出了可选的*delimiters*参数，则将其解释为包含可能的有效分隔符字符的字符串。

has_header（*样本*）

分析示例文本（推测为CSV格式），并 True在第一行显示为一系列列标题时返回。

使用示例Sniffer：

```
with open('example.csv', newline='') as csvfile:
    dialect = csv.Sniffer().sniff(csvfile.read(1024))
    csvfile.seek(0)
    reader = csv.reader(csvfile, dialect)
    # ... process CSV file contents here ...
```

该csv模块定义了以下常量：

csv. QUOTE_ALL

指示writer对象引用所有字段。

csv. QUOTE_MINIMAL

指示writer对象只引用那些包含特殊字符，如字段*分隔符*，*quotechar*或任何字符*lineterminator*。

csv. QUOTE_NONNUMERIC

指示writer对象引用所有非数字字段。

指示读者将所有未引用的字段转换为*float*类型。

csv. QUOTE_NONE

指示writer对象永远不会引用字段。当输出数据中出现当前 *分隔符*时，它前面是当前的*escapechar* 字符。如果没有设置*escapechar*，Error那么如果遇到任何需要转义的字符，

作者将会提出。

指示reader不对引号字符进行特殊处理。

该csv模块定义了以下例外情况：

*异常*csv.Error
当检测到错误时由任何功能引发。

14.1.2。方言和格式参数

为了更容易指定输入和输出记录的格式，特定的格式参数被组合为方言。方言是Dialect具有一组特定方法和单一validate()方法的类的子类。在创建reader或 writer对象时，程序员可以将该类的字符串或子类指定Dialect为方言参数。除了或代替*方言*参数之外，程序员还可以指定单独的格式化参数，这些参数的名称与Dialect该类下面定义的属性名称相同。

方言支持以下属性：

Dialect.delimiter
用于分隔字段的单字符字符串。它默认为','。

Dialect.doublequote
控制出现在字段内部的*quotechar*实例应如何引用。何时True，角色加倍。当 False时，*escapechar*作为前缀*quotechar*。它默认为True。

在输出时，如果*双引号*是False没有*escapechar*设置， Error如果需要进行上调*quotechar*是在现场发现的。

Dialect.escapechar
由作家使用的一个字符串转换为逃避*分隔符*，如果*报价*被设置为QUOTE_NONE与*quotechar*如果*双引号*是 False。阅读时，*escapechar*从下列字符中删除任何特殊含义。它默认为None禁用转义。

Dialect.lineterminator
用于终止由生成的行的字符串writer。它默认为'\r\n'。

> **注意：** 这reader是硬编码识别任一'\r'或'\n'作为行结束，并忽略*lineterminator*。这种行为在未来可能会改变。

Dialect.quotechar
一个字符的字符串，用于引用包含特殊字符（如*分隔符*或*引号*）或包含*换行符的*字段。它默认为'"'。

Dialect.quoting
控制引号应由作者生成并由读者识别。它可以采用任何QUOTE_*常量（请参见模块内容部分），并且默认为QUOTE_MINIMAL。

Dialect.skipinitialspace

何时True，忽略*分隔符*后面的空格。默认是False。

Dialect.**strict**
什么时候出现错误的CSV输入True异常Error。默认是False。

14.1.3。读者对象

Reader对象（函数DictReader返回的实例和对象 reader() ）具有以下公用方法：

csvreader.**__next__ ( )**
将读者的可迭代对象的下一行作为列表（如果返回对象reader() ）或字典（如果它是 DictReader 实例）返回，根据当前的方言进行解析。通常你应该称之为next(reader)。

读者对象具有以下公共属性：

csvreader.**dialect**
解析器使用的方言的只读描述。

csvreader.**line_num**
从源迭代器读取的行数。这与返回的记录数量不同，因为记录可以跨越多行。

DictReader对象具有以下公共属性：

csvreader.**fieldnames**
如果在创建对象时未作为参数传递，则在首次访问或从文件读取第一条记录时初始化此属性。

14.1.4。作家对象

Writer对象（ DictWriter由writer()函数返回的实例和对象）具有以下公共方法。甲*行*必须是可迭代字符串或数字用于Writer对象和一个字典映射字段名到字符串或数字（通过使它们通过 str() 第一）为DictWriter对象。请注意，复数数字是由parens包围的。这可能会导致读取CSV文件的其他程序出现问题（假设它们完全支持复数）。

csvwriter.**writerow (*row*)**
将*行*参数写入作者的文件对象，根据当前的方言格式化。

*在版本3.5中进行了更改：*添加了对任意迭代的支持。

csvwriter.**writerows (*行*)**
将所有*行*参数（如上所述的*行*对象列表）写入作者的文件对象，根据当前的方言格式化。

编写器对象具有以下公共属性：

csvwriter.**dialect**
作者正在使用的方言的只读说明。

DictWriter对象具有以下公用方法：

DictWriter.writeheader( )
用字段名写一行（如构造函数中指定的那样）。

3.2版本中的新功能

14.1.5。示例

读取CSV文件的最简单示例：

```
import csv
with open('some.csv', newline='') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

用另一种格式读取文件：

```
import csv
with open('passwd', newline='') as f:
    reader = csv.reader(f, delimiter=':', quoting=csv.QUOTE_NONE)
    for row in reader:
        print(row)
```

相应的最简单的写作示例是：

```
import csv
with open('some.csv', 'w', newline='') as f:
    writer = csv.writer(f)
    writer.writerows(someiterable)
```

由于open()用于打开CSV文件进行读取，因此该文件将默认使用系统默认编码解码为unicode（请参阅参考资料locale.getpreferredencoding()）。要使用不同的编码解码文件，请使用encodingopen参数：

```
import csv
with open('some.csv', newline='', encoding='utf-8') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

这同样适用于写入除系统默认编码以外的内容：在打开输出文件时指定编码参数。

注册一种新的方言：

```
import csv
csv.register_dialect('unixpwd', delimiter=':', quoting=csv.QUOTE_NONE)
with open('passwd', newline='') as f:
    reader = csv.reader(f, 'unixpwd')
```

稍微更高级的读者使用 - 捕捉和报告错误：

```
import csv, sys
filename = 'some.csv'
with open(filename, newline='') as f:
    reader = csv.reader(f)
    try:
        for row in reader:
            print(row)
    except csv.Error as e:
        sys.exit('file {}, line {}: {}'.format(filename, reader.line_num, e))
```

虽然模块不直接支持分析字符串，但可以轻松完成：

```
import csv
for row in csv.reader(['one,two,three']):
    print(row)
```

脚注

[1] *(1, 2)* 如果newline=''没有指定，嵌入引用字段内换行符将不会被正确地解释，并且在使用平台\r\n上写一个额外linendings \r将被添加。newline='' 由于csv模块执行自己的（通用）换行处理，因此指定它应该始终安全 。

14.2。configparser- 配置文件解析器

源代码： Lib / configparser.py

该模块提供了ConfigParser实现基本配置语言的类，该语言提供了类似于Microsoft Windows INI文件中的结构的结构。您可以使用它来编写可以由最终用户轻松定制的Python程序。

注意： 这个库并*没有*解释或写在INI语法的Windows注册表扩展版本中使用的值类型的前缀。

也可以看看:

模 shlex
支持创建可用作应用程序配置文件的备用格式的Unix shell-like迷你语言。

模 json
json模块实现了也可用于此目的的JavaScript语法子集。

14.2.1。快速入门

让我们看一个非常基本的配置文件，如下所示：

```
[DEFAULT]
ServerAliveInterval = 45
Compression = yes
CompressionLevel = 9
ForwardX11 = yes

[bitbucket.org]
User = hg

[topsecret.server.com]
Port = 50022
ForwardX11 = no
```

下一节将介绍INI文件的结构。本质上，文件由多个部分组成，每个部分包含具有值的键。configparser类可以读写这些文件。首先以编程方式创建上述配置文件。

```
>>> import configparser
>>> config = configparser.ConfigParser()
>>> config['DEFAULT'] = {'ServerAliveInterval': '45',
...                      'Compression': 'yes',
...                      'CompressionLevel': '9'}
>>> config['bitbucket.org'] = {}
>>> config['bitbucket.org']['User'] = 'hg'
>>> config['topsecret.server.com'] = {}
>>> topsecret = config['topsecret.server.com']
>>> topsecret['Port'] = '50022'     # mutates the parser
>>> topsecret['ForwardX11'] = 'no'  # same here
>>> config['DEFAULT']['ForwardX11'] = 'yes'
```

```
>>> with open('example.ini', 'w') as configfile:
...   config.write(configfile)
...
```

正如你所看到的，我们可以像配置字典一样处理配置解析器。稍后会有一些区别，但这种行为非常接近您期望从字典中获得的结果。

现在我们已经创建并保存了一个配置文件，让我们回过头来读一读它所保存的数据。

```
>>> import configparser
>>> config = configparser.ConfigParser()
>>> config.sections()
[]
>>> config.read('example.ini')
['example.ini']
>>> config.sections()
['bitbucket.org', 'topsecret.server.com']
>>> 'bitbucket.org' in config
True
>>> 'bytebong.com' in config
False
>>> config['bitbucket.org']['User']
'hg'
>>> config['DEFAULT']['Compression']
'yes'
>>> topsecret = config['topsecret.server.com']
>>> topsecret['ForwardX11']
'no'
>>> topsecret['Port']
'50022'
>>> for key in config['bitbucket.org']: print(key)
...
user
compressionlevel
serveraliveinterval
compression
forwardx11
>>> config['bitbucket.org']['ForwardX11']
'yes'
```

正如我们上面看到的，API非常简单。唯一的魔法DEFAULT部分涉及为所有其他部分提供默认值的部分[1]。还要注意，部分中的键不区分大小写，并以小写形式存储[1]。

14.2.2。支持的数据类型

配置解析器不会猜测配置文件中值的数据类型，因此始终将它们作为字符串存储在内部。这意味着如果你需要其他数据类型，你应该自己转换：

```
>>> int(topsecret['Port'])
50022
>>> float(topsecret['CompressionLevel'])
9.0
```

由于这个任务非常常见，配置解析器提供了一系列方便的getter方法来处理整数，浮点数和布尔值。最后一个是最有趣的，因为简单地将价值传递给bool()自己bool('False')仍然没有好处True。这就是配置解析器也提供的原因getboolean()。此方法不区分大小写，并从'yes'/'no'，'on'/'off'，'true'/'false'和'1'/'0' [1]中识别布尔值。例如：

```
>>> topsecret.getboolean('ForwardX11')
False
>>> config['bitbucket.org'].getboolean('ForwardX11')
True
>>> config.getboolean('bitbucket.org', 'Compression')
True
```

除此之外getboolean()，配置解析器还提供同等getint()和 getfloat()方法。您可以注册自己的转换器并自定义提供的转换器。[1]

14.2.3。回退值

与字典一样，您可以使用节的get()方法来提供回退值：

```
>>> topsecret.get('Port')
'50022'
>>> topsecret.get('CompressionLevel')
'9'
>>> topsecret.get('Cipher')
>>> topsecret.get('Cipher', '3des-cbc')
'3des-cbc'
```

请注意，默认值优先于回退值。例如，在我们的例子中，'CompressionLevel'键只在该'DEFAULT'部分中指定。如果我们试图从该部分中获取它'topsecret.server.com'，即使我们指定了回退，我们也将始终获得默认值：

```
>>> topsecret.get('CompressionLevel', '3')
'9'
```

还有一件事要注意的是，解析器级别的get()方法提供了一个自定义的，更复杂的接口，为了向后兼容性而进行维护。使用此方法时，可通过fallback关键字参数提供回退值：

```
>>> config.get('bitbucket.org', 'monster',
...            fallback='No such things as monsters')
'No such things as monsters'
```

相同的fallback参数可以与被使用 getint()，getfloat()和 getboolean()方法，例如：

```
>>> 'BatchMode' in topsecret
False
>>> topsecret.getboolean('BatchMode', fallback=True)
True
>>> config['DEFAULT']['BatchMode'] = 'no'
>>> topsecret.getboolean('BatchMode', fallback=True)
False
```

14.2.4。支持的INI文件结构

一个配置文件由多个部分组成，每个部分由一个[section]标题引导，然后是由特定字符串分隔的键/值条目（=或:默认为[1]）。默认情况下，部分名称区分大小写，但键不是 [1]。从键和值中删除前导空白和尾随空白。值可以省略，在这种情况下，键/值分隔符也可以省略。值也可以跨越多行，只要它们比值的第一行缩进更深。根据解析器的模式，空行可能被视为多行值的一部分或被忽略。

配置文件可以包括注释，通过特定的字符（前缀#和;默认[1] ）。评论可能会出现在其他空行上，可能会缩进。[1]

例如：

```
[Simple Values]
key=value
spaces in keys=allowed
spaces in values=allowed as well
spaces around the delimiter = obviously
you can also use : to delimit keys from values

[All Values Are Strings]
values like this: 1000000
or this: 3.14159265359
are they treated as numbers? : no
integers, floats and booleans are held as: strings
can use the API to get converted values directly: true

[Multiline Values]
chorus: I'm a lumberjack, and I'm okay
    I sleep all night and I work all day

[No Values]
key_without_value
empty string value here =

[You can use comments]
# like this
; or this

# By default only in an empty line.
# Inline comments can be harmful because they prevent users
# from using the delimiting characters as parts of values.
# That being said, this can be customized.

    [Sections Can Be Indented]
        can_values_be_as_well = True
        does_that_mean_anything_special = False
        purpose = formatting for readability
        multiline_values = are
            handled just fine as
            long as they are indented
            deeper than the first line
            of a value
        # Did I mention we can indent comments, too?
```

14.2.5。值的插值

在核心功能之上，ConfigParser支持插值。这意味着值可以在从get()呼叫返回之前进行预处理。

*类*configparser.**BasicInterpolation**

默认实现使用ConfigParser。它使值能够包含引用同一节中的其他值的格式字符串或特殊默认节中的值[1]。初始化时可以提供其他默认值。

例如：

```
[Paths]
home_dir: /Users
my_dir: %(home_dir)s/lumberjack
my_pictures: %(my_dir)s/Pictures
```

在上面的例子中，ConfigParser用*内插*设置为 BasicInterpolation()将解析%(home_dir)s到的值 home_dir （ /Users 在这种情况下 ）。 %(my_dir)s 实际上会解决/Users/lumberjack。所有内插都是按需完成的，因此引用链中使用的键不必在配置文件中以任何特定顺序指定。

随着interpolation设置None，解析器仅返回 %(my_dir)s/Pictures作为的价值my_pictures和 %(home_dir)s/lumberjack作为价值my_dir。

*类*configparser.**ExtendedInterpolation**

插值的替代处理程序，用于实现更高级的语法，例如用于zc.buildout。扩展插值${section:option}用于表示来自外部部分的值。插值可以跨越多个层次。为方便起见，如果section:省略该 部分，插值默认为当前部分（可能还有特殊部分的默认值）。

例如，上面使用基本插值指定的配置在扩展插值中看起来像这样：

```
[Paths]
home_dir: /Users
my_dir: ${home_dir}/lumberjack
my_pictures: ${my_dir}/Pictures
```

其他部分的值也可以被提取：

```
[Common]
home_dir: /Users
library_dir: /Library
system_dir: /System
macports_dir: /opt/local

[Frameworks]
Python: 3.2
path: ${Common:system_dir}/Library/Frameworks/

[Arthur]
nickname: Two Sheds
last_name: Jackson
```

```
my_dir: ${Common:home_dir}/twosheds
my_pictures: ${my_dir}/Pictures
python_dir: ${Frameworks:path}/Python/Versions/${Frameworks:Python}
```

14.2.6。映射协议访问

3.2版本中的新功能

映射协议访问是一种功能的通用名称，可以像使用字典一样使用自定义对象。在情况下 configparser ，映射接口实现使用 parser['section']['option']符号。

parser['section']特别是在解析器中返回该部分数据的代理。这意味着这些值不会被复制，而是根据需要从原始解析器中获取。更重要的是，当代码段的值发生变化时，它们实际上在原始解析器中发生了变异。

configparser对象的行为尽可能接近实际字典。映射界面完整，并遵守 MutableMappingABC。但是，应该考虑到一些差异：

- 默认情况下，部分中的所有密钥都可以不区分大小写的方式访问 [1]。例如，只产生'ed选项键名称。这意味着默认情况下的小写键。同时，对于拥有密钥的部分，这两个表达式都会返回：for option in parser["section"]optionxform'a' True

  ```
  "a" in parser["section"]
  "A" in parser["section"]
  ```

- 所有部分也包含DEFAULTSECT值，这意味着 .clear()部分可能不会将部分明显留空。这是因为默认值不能从节中删除（因为技术上他们不在那里）。如果它们在部分中被覆盖，则删除会导致默认值再次可见。试图删除一个默认值会导致一个KeyError。

- DEFAULTSECT 不能从解析器中移除：

  - 试图删除它引发ValueError，
  - parser.clear() 保持完好，
  - parser.popitem() 从不返回它。

- parser.get(section, option, **kwargs)- 第二个参数**不是** 回退值。不过请注意，部分级别的get()方法与映射协议和传统的configparser API兼容。

- parser.items() 与映射协议兼容（返回包含 DEFAULTSECT 的 *section_name* ，*section_proxy*对的列表 ）。但是，这个方法也可以用参数调用：。后者调用返回的列表*选项*，*值*在指定的对，与膨胀的所有内插（除非 被提供）。parser.items(section, raw, vars) sectionraw=True

映射协议在现有的传统API之上实现，因此覆盖原始接口的子类仍然应该具有按预期工作的映射。

14.2.7。自定义解析器行为

使用INI格式的应用程序几乎与应用程序一样多。 configparser为支持最大的INI样式提供了很大的支持。默认功能主要由历史背景决定，很可能您想要自定义一些功能。

改变特定配置解析器工作方式的最常见方式是使用以下__init__()选项：

- *默认值*，默认值：None

  该选项接受一个最初放在该DEFAULT部分中的键值对的字典。这为支持简洁的配置文件提供了一种优雅的方式，这些配置文件没有指定与记录的默认值相同的值。

  提示：如果要为特定部分指定默认值，请read_dict()在读取实际文件之前使用 。

- *dict_type*，默认值：collections.OrderedDict

  此选项对映射协议的行为方式以及写入的配置文件的外观有着重大影响。使用默认的有序字典，每个部分都按照添加到解析器的顺序存储。部分内的选项也一样。

  另一种字典类型可用于例如在回写时对部分和选项进行排序。出于性能原因，您也可以使用常规字典。

  请注意：有一些方法可以在单个操作中添加一组键值对。在这些操作中使用常规字典时，密钥的顺序可能是随机的。例如：

```
>>> parser = configparser.ConfigParser()
>>> parser.read_dict({'section1': {'key1': 'value1',
...                                'key2': 'value2',
...                                'key3': 'value3'},
...                   'section2': {'keyA': 'valueA',
...                                'keyB': 'valueB',
...                                'keyC': 'valueC'},
...                   'section3': {'foo': 'x',
...                                'bar': 'y',
...                                'baz': 'z'}
... })
>>> parser.sections()
['section3', 'section2', 'section1']
>>> [option for option in parser['section3']]
['baz', 'foo', 'bar']
```

  在这些操作中，您还需要使用有序字典：

```
>>> from collections import OrderedDict
>>> parser = configparser.ConfigParser()
>>> parser.read_dict(
...   OrderedDict((
...     ('s1',
...      OrderedDict((
...        ('1', '2'),
...        ('3', '4'),
...        ('5', '6'),
...      ))
...     ),
...     ('s2',
...      OrderedDict((
...        ('a', 'b'),
...        ('c', 'd'),
...        ('e', 'f'),
...      ))
```

```
...       ),
...     ))
... )
>>> parser.sections()
['s1', 's2']
>>> [option for option in parser['s1']]
['1', '3', '5']
>>> [option for option in parser['s2'].values()]
['b', 'd', 'f']
```

- *allow_no_value*，默认值：False

  已知一些配置文件包含没有值的设置，但其中符合其支持的语法configparser。构造函数的 *allow_no_value*参数可以用来表示应该接受这样的值：

```
>>> import configparser

>>> sample_config = """
... [mysqld]
...   user = mysql
...   pid-file = /var/run/mysqld/mysqld.pid
...   skip-external-locking
...   old_passwords = 1
...   skip-bdb
...   # we don't need ACID today
...   skip-innodb
... """
>>> config = configparser.ConfigParser(allow_no_value=True)
>>> config.read_string(sample_config)

>>> # Settings with values are treated as before:
>>> config["mysqld"]["user"]
'mysql'

>>> # Settings without values provide None:
>>> config["mysqld"]["skip-bdb"]

>>> # Settings which aren't specified still raise an error:
>>> config["mysqld"]["does-not-exist"]
Traceback (most recent call last):
  ...
KeyError: 'does-not-exist'
```

- *分隔符*，默认值：('=', ':')

  分隔符是一些子字符串，用于从一个分节中的值中分隔键。一行上首次出现的分隔子字符串被视为分隔符。这意味着值（但不是键）可以包含分隔符。

  另请参阅*space_around_delimiters*参数 ConfigParser.write()。

- *comment_prefixes*，默认值：('#', ';')

- *inline_comment_prefixes*，默认值：None

注释前缀是指示配置文件中有效注释开始的字符串。*comment_prefixes*仅用于其他空行（可选择缩进），而*inline_comment_prefixes*可以在每个有效值之后使用（例如段名称，选项和空行）。默认情况下，内联注释被禁用，'#' 并且';' 用作整行注释的前缀。

*在版本3.2中更改：*在先前版本的configparser行为匹配 comment_prefixes=('#',';')和 inline_comment_prefixes=(';',)。

请注意，配置解析器不支持转义注释前缀，因此使用*inline_comment_prefixes*可能会阻止用户使用用作注释前缀的字符来指定选项值。如有疑问，请避免设置*inline_comment_prefixes*。在任何情况下，以多行值形式存储注释前缀字符的唯一方法是插入前缀，例如：

```
>>> from configparser import ConfigParser, ExtendedInterpolation
>>> parser = ConfigParser(interpolation=ExtendedInterpolation())
>>> # the default BasicInterpolation could be used as well
>>> parser.read_string("""
... [DEFAULT]
... hash = #
...
... [hashes]
... shebang =
...   ${hash}!/usr/bin/env python
...   ${hash} -*- coding: utf-8 -*-
...
... extensions =
...   enabled_extension
...   another_extension
...   #disabled_by_comment
...   yet_another_extension
...
... interpolation not necessary = if # is not at line start
... even in multiline values = line #1
...   line #2
...   line #3
... """)
>>> print(parser['hashes']['shebang'])

#!/usr/bin/env python
# -*- coding: utf-8 -*-
>>> print(parser['hashes']['extensions'])

enabled_extension
another_extension
yet_another_extension
>>> print(parser['hashes']['interpolation not necessary'])
if # is not at line start
>>> print(parser['hashes']['even in multiline values'])
line #1
line #2
line #3
```

- *严格的*，默认值：True

  当设置为True，解析器将不允许任何部分或选项重复而从单个源读取（使用read_file()，read_string()或read_dict()）。建议在新应用程序中使用严格的解析

器。

*在版本3.2中更改：*在先前版本的configparser行为匹配 strict=False。

- *empty_lines_in_values*，默认值：True

  在配置解析器中，只要它们的缩进比保存它们的键更多，值就可以跨越多行。默认情况下，解析器也会让空行成为值的一部分。同时，键可以任意缩进以提高可读性。因此，当配置文件变得庞大而复杂时，用户很容易失去对文件结构的跟踪。举个例子：

  ```
  [Section]
  key = multiline
    value with a gotcha

   this = is still a part of the multiline value of 'key'
  ```

  如果用户使用比例字体来编辑文件，这可能是特别有问题的。这就是为什么当你的应用程序不需要空行的值时，你应该考虑禁止它们。这将使空行每次分割键。在上面的例子中，它会产生两个键，key和this。

- *default_section*，默认值：configparser.DEFAULTSECT（即：“DEFAULT”）

  允许为其他部分或插值目的使用特殊部分默认值的约定是该库的一个强大的概念，它允许用户创建复杂的声明性配置。通常会调用此部分，“DEFAULT”但可以将其自定义为指向任何其他有效的部分名称。一些典型值包括：“general”或“common”。提供的名称用于在从任何源读取时识别缺省段，并在将配置写回到文件时使用。它的当前值可以使用该parser_instance.default_section属性检索， 并且可以在运行时修改（即将文件从一种格式转换为另一种格式）。

- *插值*，默认值：configparser.BasicInterpolation

  插值行为可以通过*插值*参数提供自定义处理程序来定制。None可以用来完全关闭插值，ExtendedInterpolation()提供了一个更高级的变种灵感zc.buildout。更多关于专用文档部分的主题。 RawConfigParser有一个默认值None。

- *转换器*，默认值：未设置

  配置解析器提供执行类型转换的选项值获取器。默认情况下getint()，getfloat()和getboolean()实现。如果需要其他getter，用户可以在子类中定义它们，或者传递一个字典，其中每个键都是转换器的名称，每个值都是可执行的，用于实现所述转换。例如，传递将添加 解析器对象和所有节代理。换句话说，可以同时编写 和 。{'decimal': decimal.Decimal} getdecimal() parser_instance.getdecimal('section', 'key', fallback=0) parser_instance['section'].getdecimal('key', 0)

  如果转换器需要访问解析器的状态，它可以作为一个配置解析器子类的方法来实现。如果此方法的名称以该名称开头get，则它将在所有节代理上以dict兼容形式提供（请参阅getdecimal()上面的示例）。

通过覆盖这些解析器属性的默认值可以实现更高级的定制。默认值是在类上定义的，所以它们可以被子类或属性赋值覆盖。

configparser.BOOLEAN_STATES

默认情况下使用时getboolean()，配置解析器考虑以下值True：'1'，'yes'，'true'，'on'和以下值False：'0'，'no'，'false'，'off'。您可以通过指定字符串及其布尔结果的自定义字典来覆盖此内容。例如：

```
>>> custom = configparser.ConfigParser()
>>> custom['section1'] = {'funky': 'nope'}
>>> custom['section1'].getboolean('funky')
Traceback (most recent call last):
...
ValueError: Not a boolean: nope
>>> custom.BOOLEAN_STATES = {'sure': True, 'nope': False}
>>> custom['section1'].getboolean('funky')
False
```

其他典型的布尔对包括accept/ reject或 enabled/ disabled。

configparser.optionxform（*可选*）

此方法在每次读取，获取或设置操作时转换选项名称。默认将名称转换为小写。这也意味着当配置文件被写入时，所有密钥都将是小写字母。如果不适合，则覆盖此方法。例如：

```
>>> config = """
... [Section1]
... Key = Value
...
... [Section2]
... AnotherKey = Value
... """
...
>>> typical = configparser.ConfigParser()
>>> typical.read_string(config)
>>> list(typical['Section1'].keys())
['key']
>>> list(typical['Section2'].keys())
['anotherkey']
>>> custom = configparser.RawConfigParser()
>>> custom.optionxform = lambda option: option
>>> custom.read_string(config)
>>> list(custom['Section1'].keys())
['Key']
>>> list(custom['Section2'].keys())
['AnotherKey']
```

configparser.SECTCRE

一个编译的正则表达式，用于解析节标题。默认匹配[section]名称"section"。空格被认为是部分名称的一部分，因此将被视为名称的一部分。如果不合适，则覆盖此属性。例如：[larch]" larch "

```
>>> config = """
... [Section 1]
... option = value
...
... [ Section 2 ]
... another = val
```

```
...     """
>>> typical = ConfigParser()
>>> typical.read_string(config)
>>> typical.sections()
['Section 1', '  Section 2  ']
>>> custom = ConfigParser()
>>> custom.SECTCRE = re.compile(r"\[ *(?P<header>[^]]+?) *\]")
>>> custom.read_string(config)
>>> custom.sections()
['Section 1', 'Section 2']
```

注意： 虽然ConfigParser对象也使用OPTCRE属性来识别选项行，但不建议重写它，因为这会干扰构造器选项*allow_no_value*和*分隔符*。

14.2.8。传统API示例

主要是因为向后兼容性问题，configparser 还提供了带有明确get/ set方法的传统API 。尽管下面概述的方法有有效的用例，但对于新项目，映射协议访问是首选。传统的API有时更先进，低级和彻底违反直觉。

写入配置文件的示例：

```
import configparser

config = configparser.RawConfigParser()

# Please note that using RawConfigParser's set functions, you can assign
# non-string values to keys internally, but will receive an error when
# attempting to write to a file or when you get it in non-raw mode. Setting
# values using the mapping protocol or ConfigParser's set() does not allow
# such assignments to take place.
config.add_section('Section1')
config.set('Section1', 'an_int', '15')
config.set('Section1', 'a_bool', 'true')
config.set('Section1', 'a_float', '3.1415')
config.set('Section1', 'baz', 'fun')
config.set('Section1', 'bar', 'Python')
config.set('Section1', 'foo', '%(bar)s is %(baz)s!')

# Writing our configuration file to 'example.cfg'
with open('example.cfg', 'w') as configfile:
    config.write(configfile)
```

再次读取配置文件的示例：

```
import configparser

config = configparser.RawConfigParser()
config.read('example.cfg')

# getfloat() raises an exception if the value is not a float
# getint() and getboolean() also do this for their respective types
a_float = config.getfloat('Section1', 'a_float')
```

```
an_int = config.getint('Section1', 'an_int')
print(a_float + an_int)

# Notice that the next output does not interpolate '%(bar)s' or '%(baz)s'.
# This is because we are using a RawConfigParser().
if config.getboolean('Section1', 'a_bool'):
    print(config.get('Section1', 'foo'))
```

要获得插值，请使用ConfigParser：

```
import configparser

cfg = configparser.ConfigParser()
cfg.read('example.cfg')

# Set the optional *raw* argument of get() to True if you wish to disable
# interpolation in a single get operation.
print(cfg.get('Section1', 'foo', raw=False))  # -> "Python is fun!"
print(cfg.get('Section1', 'foo', raw=True))   # -> "%(bar)s is %(baz)s!"

# The optional *vars* argument is a dict with members that will take
# precedence in interpolation.
print(cfg.get('Section1', 'foo', vars={'bar': 'Documentation',
                                       'baz': 'evil'}))

# The optional *fallback* argument can be used to provide a fallback value
print(cfg.get('Section1', 'foo'))
      # -> "Python is fun!"

print(cfg.get('Section1', 'foo', fallback='Monty is not.'))
      # -> "Python is fun!"

print(cfg.get('Section1', 'monster', fallback='No such things as monsters.'))
      # -> "No such things as monsters."

# A bare print(cfg.get('Section1', 'monster')) would raise NoOptionError
# but we can also use:

print(cfg.get('Section1', 'monster', fallback=None))
      # -> None
```

这两种类型的ConfigParsers都有默认值。如果使用的选项未在其他地方定义，它们将用于插值。

```
import configparser

# New instance with 'bar' and 'baz' defaulting to 'Life' and 'hard' each
config = configparser.ConfigParser({'bar': 'Life', 'baz': 'hard'})
config.read('example.cfg')

print(config.get('Section1', 'foo'))     # -> "Python is fun!"
config.remove_option('Section1', 'bar')
config.remove_option('Section1', 'baz')
print(config.get('Section1', 'foo'))     # -> "Life is hard!"
```

14.2.9。ConfigParser对象

class configparser.ConfigParser（*defaults = None*，*dict_type = collections.OrderedDict*，*allow_no_value = False*，*delimiters = ('='，':')*，*comment_prefixes = ('#'，';')*，*inline_comment_prefixes = None*，*strict = True*，*empty_lines_in_values = True*，*default_section = configparser.DEFAULTSECT*，*interpolation = BasicInterpolation（）*，*转换器= {}* ）

主要配置解析器。当*默认*给出，它被初始化到内在默认的字典。当给出*dict_type时*，它将用于为部分列表，部分内的选项以及默认值创建字典对象。

当给定*分隔符时*，它被用作从值中分割键的子字符串集合。当给出*comment_prefixes时*，它将用作在其他空行中添加注释的子字符串集合。评论可以缩进。当给出*inline_comment_prefixes时*，它将用作在非空行中加注释前缀的子字符串集合。

当*strict*是True（缺省值）时，解析器将不允许在从单个源（文件，字符串或字典）中读取任何段或选项时重复提高DuplicateSectionError或 DuplicateOptionError。当*empty_lines_in_values*是False （默认：）*的*True，每个空行标记选项的结束。否则，多行选项的内部空行将保留为值的一部分。当*allow_no_value*为True（默认：）*的*False，不接受值的选项被接受; 为这些保留的值是，None并且它们被序列化而没有尾随分隔符。

当给出*default_section时*，它指定特殊部分的名称，其中包含其他部分的缺省值和插值目的（通常为named "DEFAULT"）。可以使用default_section实例属性在运行时检索和更改此值。

插值行为可以通过*插值*参数提供自定义处理程序来定制。None可以用来完全关闭插值，ExtendedInterpolation()提供了一个更高级的变种灵感zc.buildout。更多关于专用文档部分的主题。

插值中使用的所有选项名称将通过该optionxform()方法传递，就像任何其他选项名称引用一样。例如，使用默认实现optionxform()（将选项名称转换为小写），这些值和等价。foo %(bar)s foo %(BAR)s

当给出*转换器时*，它应该是一个字典，其中每个键表示类型转换器的名称，每个值都是可调用的，实现从字符串到所需数据类型的转换。每个转换器都get*()在解析器对象和节代理上获得自己的相应方法。

*版本3.1中更改：*默认的*dict_type*是collections.OrderedDict。

在版本3.2中进行了更改：添加了allow_no_value，*分隔符*，*comment_prefixes*，*strict*，*empty_lines_in_values*，*default_section*和*插值*。

*在3.5版本中改为：*该*转换器*加入争论。

defaults（）

返回包含实例范围默认值的字典。

sections（）

返回可用部分的列表; 在*默认的部分*不包括在列表中。

add_section (*部分*)

将名为 *section 的节*添加到实例中。如果给定名称的部分已经存在，DuplicateSectionError则提出。如果传递了 *默认的段*名称，ValueError则会引发。该部分的名称必须是字符串; 如果没有，TypeError则会提出。

*在版本3.2中更改：*引发非字符串部分名称TypeError。

has_section (*部分*)

指示命名 *节*是否存在于配置中。该*默认部分*没有被确认。

options (*部分*)

返回指定*部分中*可用选项的列表。

has_option (*部分*，*选项*)

如果给定*部分*存在并包含给定*选项*，则返回 True; 否则返回False。如果指定的 *部分*是None空字符串，则假定为DEFAULT。

read (*文件名*，*编码=无*)

尝试读取和解析文件名列表，返回已成功解析的文件名列表。

如果*文件名*是字符串或类似路径的对象，则它被视为单个文件名。如果以文件名命名的*文件*无法打开，则该文件将被忽略。这样做的目的是为了指定一个潜在的配置文件位置列表（例如，当前目录，用户的主目录以及某个系统范围的目录），并且读取列表中的所有现有配置文件。

如果没有指定的文件存在，则该ConfigParser 实例将包含一个空数据集。要求从文件加载初始值的应用程序应read_file()在调用read()任何可选文件之前加载所需的一个或多个文件：

```
import configparser, os

config = configparser.ConfigParser()
config.read_file(open('defaults.cfg'))
config.read(['site.cfg', os.path.expanduser('~/.myapp.cfg')],
            encoding='cp1250')
```

*新版本3.2：*该*编码*参数。以前，所有文件都是使用默认编码读取的open()。

*在新版本3.6.1：*该*文件名*参数接受路径状物体。

read_file (*f*, *source = None*)

从*f中*读取和分析配置数据，这些数据必须是可迭代的，产生Unicode字符串（例如以文本模式打开的文件）。

可选参数*源*指定正在读取的文件的名称。如果没有给出，*f*有一个name属性，用于 *源*; 默认是'<???>'。

*版本3.2中的新功能：*取代readfp()。

read_string (*string*, *source ='<string>'*)

从字符串解析配置数据。

可选参数*source*指定传递字符串的特定于上下文的名称。如果没有给出，'<string>'使用。这通常应该是文件系统路径或URL。

3.2版本中的新功能

read_dict（*dictionary*，*source*='<dict>'）

从提供字典式items()方法的任何对象加载配置。键是部分名称，值是包含键和值的字典，应该出现在部分中。如果使用的字典类型保留顺序，则会按顺序添加分区和它们的键。值自动转换为字符串。

可选参数*source*指定传递的字典的特定于上下文的名称。如果没有给出，<dict>使用。

这个方法可以用来在解析器之间复制状态。

3.2版本中的新功能

get（*section*，*option*，*，*raw* = *False*，*vars* = *None*[，*fallback*]）

获取指定*部分*的*选项*值。如果提供了*vars*，它必须是字典。该*选项*在*变量*（如果提供），*部分*和*DEFAULTSECT*中以该顺序查找。如果未找到密钥并提供*回退*，则将其用作回退值。 可以作为*后备*值提供。None

'%'除非*原始*参数为true ，否则所有插值都会在返回值中扩展。插值键的值以与选项相同的方式查找。

*在版本3.2中更改：*参数*raw*，*vars*和*fallback*仅用于关键字，以保护用户免于尝试使用第三个参数作为*回*退回退（尤其是在使用映射协议时）。

getint（*section*，*option*，*，*raw* = *False*，*vars* = *None*[，*fallback*]）

将指定*部分中*的*选项*强制为 整数的简便方法。请参阅*原料*，*变量*和 *后备的*解释。get()

getfloat（*section*，*option*，*，*raw* = *False*，*vars* = *None*[，*fallback*]）

将指定*部分中*的*选项*强制 为浮点数的简便方法。请参阅*原料*， *变量*和*后备的*解释。get()

getboolean（*section*，*option*，*，*raw* = *False*，*vars* = *None*[，*fallback*]）

将指定*部分中*的*选项*强制 为布尔值的简便方法。需要注意的是该选项的接受值是，，，和，导致该方法返回，和，，，和，这导致它返回。这些字符串值以不区分大小写的方式进行检查。任何其他价值都会导致它升高 。请参阅*原料*，*变量*和 *后备的*解释。'1''yes''true''on'True'0''no''false''off'FalseValueErrorget()

items（*raw* = *False*，*vars* = *None*）

items（*section*，*raw* = *False*，*vars* = *None*）

当没有给出*部分时*，返回*section_name*， *section_proxy*对的列表，包括DEFAULTSECT。

否则，返回给定*部分*中选项的*名称*，*值*对的列表。可选参数的含义与该方法相同 。 get()

set (*section* , *option* , *value*)

如果给定部分存在，请将给定选项设置为指定值; 否则提出NoSectionError。 *选项*和*值*必须是字符串; 如果没有， TypeError则会提出。

write (*fileobject* , *space_around_delimiters = True*)

将配置表示写入指定的文件对象，该对象必须以文本模式打开（接受字符串）。这种表示可以通过未来的read()呼叫来解析。如果 *space_around_delimiters*为true，则键和值之间的分隔符将被空格包围。

remove_option (*部分* , *选项*)

从指定*部分*删除指定的*选项*。如果该部分不存在，请提出。如果该选项存在被删除，则返回; 否则返回 。 NoSectionErrorTrueFalse

remove_section (*部分*)

从配置中删除指定的*部分*。如果该部分实际存在，则返回True。否则返回False。

optionxform (*可选*)

转换在输入文件中找到的选项名称*选项*或由客户端代码传递到应在内部结构中使用的表单。默认实现返回一个小写版本的 *选项*; 子类可能会覆盖此，或者客户端代码可以在实例上设置此名称的属性以影响此行为。

您不需要将解析器的子类用于使用此方法，也可以将其设置在实例上，并将其设置为接受字符串参数并返回字符串的函数。 str例如，将其设置为使选项名称区分大小写：

```
cfgparser = ConfigParser()
cfgparser.optionxform = str
```

请注意，在读取配置文件时，选项名称周围的空白将在optionxform()调用之前被删除。

readfp (*fp* , *filename = None*)

*自3.2版弃用：*read_file()代之以使用。

*在版本3.2中改变：*readfp()现在迭代*fp*而不是调用fp.readline()。

对于readfp()使用不支持迭代的参数调用的现有代码，以下生成器可用作文件类对象的包装：

```
def readline_generator(fp):
    line = fp.readline()
    while line:
        yield line
        line = fp.readline()
```

而不是parser.readfp(fp)使用 parser.read_file(readline_generator(fp))。

configparser.MAX_INTERPOLATION_DEPTH

get()当*原始*参数为false时递归插值的最大深度。这仅在使用默认*插值*时有关。

14.2.10。RawConfigParser对象

class configparser.RawConfigParser(*defaults* = None, *dict_type* = collections.OrderedDict, *allow_no_value* = False, *, *delimiters* = ('=', ':'), *comment_prefixes* = ('#', ';'), *inline_comment_prefixes* = None, *strict* = True, *empty_lines_in_values* = True, *default_section* = configparser.DEFAULTSECT[, *interpolation*])

ConfigParser插值的传统变体默认情况下禁用，不安全add_section和set方法。

> **注意:** 考虑使用ConfigParser它来检查内部存储的值的类型。如果你不想插值，你可以使用ConfigParser(interpolation=None)。

add_section(*部分*)

将名为*section*的*节*添加到实例中。如果给定名称的部分已经存在，DuplicateSectionError则提出。如果传递了*默认的段*名称，ValueError则会引发。

类型*部分*不检查，它可以让用户创建一个名为段非字符串。此行为不受支持，并可能导致内部错误。

set(*section*, *option*, *value*)

如果给定部分存在，请将给定选项设置为指定值; 否则提出NoSectionError。尽管可以使用 RawConfigParser（或ConfigParser将*原始*参数设置为true）用于非字符串值的*内部*存储，但只能使用字符串值实现全部功能（包括插值和输出到文件）。

该方法允许用户在内部为键分配非字符串值。此行为不受支持，尝试写入文件或以非原始模式获取时会导致错误。 **使用** 不允许进行这种分配**的映射协议API**。

14.2.11。例外

*异常*configparser.Error

所有其他configparser例外的基类。

*异常*configparser.NoSectionError

找不到指定的部分时引发异常。

*异常*configparser.DuplicateSectionError

如果add_section()使用已经存在的节的名称调用异常，或者在单个输入文件，字符串或字典中多次使用节时使用严格分析器调用异常。

*版本3.2中的新增内容：*可选source和lineno要__init__()添加的属性和参数。

*异常*configparser.DuplicateOptionError

如果单个选项在从单个文件，字符串或字典中读取期间出现两次，则由严格解析器引发的异常。这会捕获拼写错误和区分大小写相关的错误，例如，字典可能有两个键代表相同的

不区分大小写的配置键。

*异常*configparser.NoOptionError
当在指定的部分找不到指定的选项时引发异常。

*异常*configparser.InterpolationError
执行字符串插值时发生问题时引发异常的基类。

*异常*configparser.InterpolationDepthError
由于迭代次数超过而无法完成字符串插值时引发异常MAX_INTERPOLATION_DEPTH。子类InterpolationError。

*异常*configparser.InterpolationMissingOptionError
从值引用的选项不存在时引发异常。子类InterpolationError。

*异常*configparser.InterpolationSyntaxError
当进行替换的源文本不符合所需的语法时引发异常。子类InterpolationError。

*异常*configparser.MissingSectionHeaderError
尝试解析没有节标题的文件时引发异常。

*异常*configparser.ParsingError
当尝试解析文件时发生错误时引发异常。

*在版本3.2中进行了更改：*将filename属性和__init__()参数重命名以 source保持一致性。

脚注

[1] *(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)* 配置解析器允许重定制。如果您有兴趣更改脚注参考概述的行为，请参阅“自定义解析器行为”部分。

14.3。 netrc- netrc文件处理

源代码： Lib / netrc.py

本netrc类解析和封装了Unix的使用NETRC文件格式**的FTP**程序和其他FTP客户端。

class netrc. netrc（[*file*]）

一个netrc实例或子类实例封装来自netrc文件的数据。初始化参数（如果存在）指定要解析的文件。如果未提供参数，.netrc则会读取用户主目录中的文件。NetrcParseError诊断信息（包括文件名，行号和终止令牌）会引发解析错误。如果POSIX系统上没有指定参数，密码中存在.netrc文件将提出一个NetrcParseError，如果该文件的所有权或权限是不安全的（由除用户以外的其他用户身份运行进程所拥有的，或访问读或通过任何其他写用户）。这实现了与ftp和其他使用的程序相同的安全行为.netrc。

*在版本3.4中更改：*添加了POSIX权限检查。

*异常*netrc. NetrcParseError

netrc在源文本中遇到语法错误时由类引发的异常。此例外的实例提供了三个有趣的属性：msg错误的文本解释， filename源文件的名称，并lineno给出发现错误的行号。

14.3.1。netrc对象

一个netrc实例有以下方法：

netrc. authenticators（*主机*）

为*主机*返回一个三元组验证器。如果netrc文件不包含给定主机的条目，则返回与“默认”条目关联的元组。如果没有匹配的主机和默认条目可用，则返回。(login, account, password) None

netrc. __repr__（）

将类数据以netrc文件的格式转储为字符串。（这会丢弃评论并可能重新排列条目。）

的实例netrc有公开的实例变量：

netrc. hosts

将主机名字典映射到元组。“默认”条目（如果有的话）通过该名称表示为伪主机。(login, account, password)

netrc. macros

将宏名称字典映射到字符串列表。

注意： 密码仅限于ASCII字符集的一个子集。在密码中允许使用所有ASCII标点，但请注意，密码中不允许使用空格和不可打印的字符。这是.netrc文件解析的方式的一个限制，将来可能会被删除。

14.4。 xdrlib- 编码和解码XDR数据

源代码： Lib / xdrlib.py

该xdrlib模块支持外部数据表示标准，如下所述**RFC 1014**，由Sun Microsystems，Inc.于1987年6月编写。它支持RFC中描述的大多数数据类型。

该xdrlib模块定义了两个类，一个用于将变量打包到XDR表示中，另一个用于从XDR表示中解压缩。还有两个异常类。

*类*xdrlib. Packer
Packer是将数据打包到XDR表示中的类。将 Packer类实例化不带任何参数。

*类*xdrlib. Unpacker (*数据*)
Unpacker是从字符串缓冲区中解压XDR数据值的补充类。输入缓冲区是作为*数据*给出的。

也可以看看:

RFC 1014 - XDR：外部数据表示标准
该RFC定义了在模块最初编写时XDR的数据编码。它显然已经过时了**RFC 1832**。

RFC 1832 - XDR：外部数据表示标准
较新的RFC，提供了XDR的修订定义。

14.4.1。帕克对象

Packer 实例具有以下方法：

Packer. get_buffer ( )
以字符串形式返回当前包缓冲区。

Packer. reset ( )
将包缓冲区重置为空字符串。

通常，您可以通过调用适当的pack_type()方法来打包任何最常见的XDR数据类型。每种方法都只有一个参数，即要打包的值。下面简单数据类型的包装方法的支持： pack_uint() ，pack_int() ，pack_enum() ，pack_bool() ， pack_uhyper() ，和pack_hyper()。

Packer. pack_float (*价值*)
打包单精度浮点数*值*。

Packer. pack_double (*价值*)
打包双精度浮点数*值*。

以下方法支持打包字符串，字节和不透明数据：

Packer.pack_fstring（*n*，*s*）
包装固定长度的字符串*s*。 *n*是字符串的长度，但 *不*包含在数据缓冲区中。如果需要，字符串会填充空字节以保证4字节对齐。

Packer.pack_fopaque（*n*，*数据*）
包装一个固定长度的不透明数据流，类似于pack_fstring()。

Packer.pack_string（*s*）
包装一个可变长度的字符串*s*。首先将字符串的长度打包为无符号整数，然后将字符串数据打包 pack_fstring()。

Packer.pack_opaque（*数据*）
包装一个可变长度的不透明数据字符串，类似于pack_string()。

Packer.pack_bytes（*字节*）
打包一个可变长度的字节流，类似于pack_string()。

以下方法支持打包数组和列表：

Packer.pack_list（*list*，*pack_item*）
打包同类项目的*列表*。此方法对于具有不确定大小的列表非常有用; 即在整个列表走完之前，大小不可用。对于列表中的每个项目，1首先打包一个无符号整数，然后打包列表中的数据值。 *pack_item*是被调用来打包单个项目的函数。在列表的最后，0打包一个无符号整数。

例如，要打包整数列表，代码可能如下所示：

```
import xdrlib
p = xdrlib.Packer()
p.pack_list([1, 2, 3], p.pack_int)
```

Packer.pack_farray（*n*，*array*，*pack_item*）
打包同质项目的固定长度列表（*数组*）。 *n*是列表的长度; 它*没有*打包到缓冲区中，但是ValueError如果len(array)不等于*n*，则会引发异常。如上所述，*pack_item*是用于打包每个元素的函数。

Packer.pack_array（*list*，*pack_item*）
打包同义项目的可变长度*列表*。首先，列表的长度被打包为一个无符号整数，然后每个元素按pack_farray()上面的方式打包 。

14.4.2。解包器对象

该Unpacker课程提供以下方法：

Unpacker.reset（*数据*）
用给定的*数据*重新设置字符串缓冲区。

Unpacker.get_position ( )
返回数据缓冲区中的当前解包位置。

Unpacker.set_position (*位置*)
将数据缓冲区解包位置设置为*位置*。你应该小心使用get_position()和set_position()。

Unpacker.get_buffer ( )
以字符串形式返回当前的解包数据缓冲区。

Unpacker.done ( )
表示解包完成。Error如果所有数据都未解压，则引发异常。

另外，可以用a打包的每种数据类型都可以用a Packer打开Unpacker。解包方法的形式是unpack_type()，并且不需要任何参数。他们返回解压缩的对象。

Unpacker.unpack_float ( )
解包一个单精度浮点数。

Unpacker.unpack_double ( )
解压缩一个双精度浮点数，类似于 unpack_float()。

另外，以下方法解压字符串，字节和不透明数据：

Unpacker.unpack_fstring (*n*)
解压并返回一个固定长度的字符串。 *n*是预期的字符数。假定填充空字节以保证4字节对齐。

Unpacker.unpack_fopaque (*n*)
解包并返回一个固定长度的不透明数据流，类似于 unpack_fstring()。

Unpacker.unpack_string ( )
解压并返回一个可变长度的字符串。首先将字符串的长度解压缩为无符号整数，然后将字符串数据解压缩 unpack_fstring()。

Unpacker.unpack_opaque ( )
解压缩并返回一个可变长度的不透明数据字符串，类似于 unpack_string()。

Unpacker.unpack_bytes ( )
解压并返回一个可变长度的字节流，类似于 unpack_string()。

以下方法支持解包数组和列表：

Unpacker.unpack_list (*unpack_item*)
解包并返回同质项目列表。首先解压缩一个无符号整数标志，然后每次解压缩一个元素。如果该标志是 1 ，则该项目被解压并附加到列表中。一个标志0 表示列表的结尾。*unpack_item*是被调用来解包项目的函数。

Unpacker.unpack_farray (*n* , *unpack_item*)

解包并返回（作为列表）一个固定长度的同类物品数组。 *n* 是缓冲区中期望的列表元素的数量。如上所述，*unpack_item*是用于解包每个元素的函数。

Unpacker.unpack_array（*unpack_item* ）

解压缩并返回一个可变长度的同类项目*列表*。首先，将列表的长度解压缩为一个无符号整数，然后像unpack_farray()上面那样解压缩每个元素。

14.4.3。例外

这个模块中的异常被编码为类实例：

*异常*xdrlib.Error

基本的异常类。 Error具有msg包含错误描述的单个公共属性 。

*异常*xdrlib.ConversionError

类派生自Error。不包含额外的实例变量。

以下是您将如何捕获以下例外之一的示例：

```
import xdrlib
p = xdrlib.Packer()
try:
    p.pack_double(8.01)
except xdrlib.ConversionError as instance:
    print('packing the double failed:', instance.msg)
```

14.5。plistlib- 生成并解析Mac OS X .plist 文件

源代码： Lib / plistlib.py

该模块提供了一个接口，用于读写主要由Mac OS X使用的“属性列表”文件，并支持二进制文件和XML plist文件。

属性列表（.plist）文件格式是支持基本对象类型的简单序列化，如字典，列表，数字和字符串。通常顶级对象是一个字典。

要写出并解析plist文件，请使用dump()和 load()函数。

要在字节对象中使用plist数据，请使用dumps() 和loads()。

值可以是字符串，整数，浮点数，布尔值，元组，列表，字典（但只有字符串键）Data，bytes，bytesarray 或datetime.datetime对象。

*在版本3.4中更改：*新API，旧API已弃用。支持二进制格式的plists添加。

也可以看看:

PList手册页
Apple的文件格式文件。

该模块定义了以下功能：

plistlib.load（*fp*，*，*fmt = None*，*use_builtin_types = True*，*dict_type = dict* ）
阅读plist文件。*fp*应该是一个可读的二进制文件对象。返回解压后的根对象（通常是一个字典）。

该*FMT*是文件的格式和下面的值是有效的：

- None：自动检测文件格式
- FMT_XML：XML文件格式
- FMT_BINARY：二进制plist格式

如果*use_builtin_types*为true（缺省值），则二进制数据将作为实例返回bytes，否则将作为实例返回 Data。

该*dict_type*是用于那些从plist文件读取字典的类型。plist的确切结构可以通过使用来恢复collections.OrderedDict（尽管在plist文件中键的顺序不应该是重要的）。

该FMT_XML格式的XML数据使用Expat解析器进行分析xml.parsers.expat- 请参阅其文档以了解格式不正确的XML可能出现的异常。未知元素将被plist解析器忽略。

InvalidFileException 无法分析文件时，二进制格式的解析器会引发。

3.4版新增功能

plistlib.**loads**(*data*，***，*fmt = None*，*use_builtin_types = True*，*dict_type = dict*)
从字节对象加载plist。请参阅`load()`关于参数的解释。

3.4版新增功能

plistlib.**dump**(*值*，*fp*，***，*fmt = FMT_XML*，*sort_keys = True*，*skipkeys = False*)
将*值*写入plist文件。*Fp*应该是一个可写的二进制文件对象。

该*FMT*参数指定plist文件的格式，并且可以是以下值之一：

- `FMT_XML`：XML格式的plist文件
- `FMT_BINARY`：二进制格式的plist文件

当*sort_keys*为true（默认）时，词典的键将按排序顺序写入plist，否则它们将按字典的迭代顺序编写。

当*skipkeys*为false（缺省值）时，函数`TypeError` 在字典的键不是字符串时引发，否则跳过这些键。

`TypeError`如果对象是不受支持的类型或包含不受支持类型的对象的容器，则会引发A.

一个`OverflowError`将提高对于不能在（二进制）的plist文件来表示的整数值。

3.4版新增功能

plistlib.**dumps**(*value*，***，*fmt = FMT_XML*，*sort_keys = True*，*skipkeys = False*)
作为plist格式的字节对象返回*值*。有关`dump()`此函数的关键字参数的解释，请参阅文档。

3.4版新增功能

以下功能已弃用：

plistlib.**readPlist**(*pathOrFile*)
阅读plist文件。*pathOrFile*可以是文件名或（可读和二进制）文件对象。返回解压后的根对象（通常是一个字典）。

这个函数调用`load()`来完成实际的工作，参见关于关键字参数的解释的文档。 `that function`

> **注意：** 结果中的Dict值有一个`__getattr__`遵循的方法`__getitem_`。这意味着您可以使用属性访问来访问这些字典的项目。

自3.4版弃用：`load()`改为使用。

plistlib.**writePlist**(*rootObject*，*pathOrFile*)
将*rootObject*写入XML plist文件。*pathOrFile*可以是文件名或（可写和二进制）文件对象

自3.4版弃用：`dump()`改为使用。

plistlib.readPlistFromBytes（*数据*）
从字节对象读取plist数据。返回根对象。

请参阅load()关于参数的描述。

> **注意：** 结果中的Dict值有一个__getattr__遵循的方法__getitem_。这意味着您可以使用属性访问来访问这些字典的项目。

*自3.4版弃用：*loads()改为使用。

plistlib.writePlistToBytes（*rootObject*）
将*rootObject*作为XML plist格式的字节对象返回。

*自3.4版弃用：*dumps()改为使用。

以下课程可供选择：

Dict([dict]):
返回与字典*词典*相同值的扩展映射对象 。

该类是dict可以使用属性访问权限访问项目的子类。也就是说，aDict.key与aDict['key']获取，设置和删除映射中的项目相同 。

自3.0版以来已弃用。

*类*plistlib.Data（*数据*）
返回字节对象*数据*周围的“数据”包装器对象。这用于从plists转换成plists的函数来表示plists中<data>可用的类型。

它有一个属性，data可用于检索存储在其中的Python字节对象。

*自3.4版弃用：*bytes改为使用对象。

以下常量可用：

plistlib.FMT_XML
plist文件的XML格式。

3.4版新增功能

plistlib.FMT_BINARY
plist文件的二进制格式

3.4版新增功能

14.5.1。示例

生成一个plist：

```
pl = dict(
    aString = "Doodah",
    aList = ["A", "B", 12, 32.1, [1, 2, 3]],
    aFloat = 0.1,
    anInt = 728,
    aDict = dict(
        anotherString = "<hello & hi there!>",
        aThirdString = "M\xe4ssig, Ma\xdf",
        aTrueValue = True,
        aFalseValue = False,
    ),
    someData = b"<binary gunk>",
    someMoreData = b"<lots of binary gunk>" * 10,
    aDate = datetime.datetime.fromtimestamp(time.mktime(time.gmtime())),
)
with open(fileName, 'wb') as fp:
    dump(pl, fp)
```

解析plist：

```
with open(fileName, 'rb') as fp:
    pl = load(fp)
print(pl["aKey"])
```

15.加密服务

本章描述的模块实现各种密码学算法。它们由安装人员自行决定。在Unix系统上，该`crypt`模块也可能是可用的。这里有一个概述：

15.1。hashlib- 安全散列和消息摘要

源代码： Lib / hashlib.py

该模块实现了许多不同安全散列和消息摘要算法的通用接口。包括FIPS安全哈希算法SHA1，SHA224，SHA256，SHA384和SHA512（在FIPS 180-2中定义）以及RSA的MD5算法（在Internet中定义**RFC 1321**）。术语“安全散列”和“消息摘要”是可互换的。较早的算法被称为消息摘要。现代术语是安全散列。

> **注意：** 如果您需要adler32或crc32哈希函数，则它们在zlib模块中可用。

> **警告：** 一些算法已知散列冲突弱点，请参阅最后的“另请参见”部分。

15.1.1。散列算法

有一种为每种类型的*散列*命名的构造函数方法。所有返回一个具有相同简单接口的哈希对象。例如：用于sha256()创建一个SHA-256哈希对象。您现在可以使用该方法为类似字节的对象（通常bytes）提供此对象update()。在任何时候，您都可以使用or 方法向它提供输入到它的数据的连接*摘要*。digest() hexdigest()

> **注意：** 为了获得更好的多线程性能，Python GIL针对创建对象或更新时的大于2047字节的数据发布。

> **注意：** update()不支持将字符串对象送入，因为散列在字节而不是字符上工作。

散列算法，总是存在该模块中的构造是 sha1()，sha224()，sha256()，sha384()， sha512()，blake2b()，和blake2s()。 md5()通常也可以使用，但如果您使用的是罕见的“符合FIPS”的Python版本，它可能会丢失。根据Python在您的平台上使用的OpenSSL库，还可以使用其他算法。在大多数平台上 sha3_224()，sha3_256()，sha3_384()，sha3_512()， shake_128()，shake_256()也可提供。

新的3.6版： SHA3（Keccak）和SHAKE构造 sha3_224()， sha3_256()， sha3_384()，sha3_512()，shake_128()，shake_256()。

*新的3.6版：*blake2b()和blake2s()添加。

例如，要获取字节字符串的摘要：b'Nobody inspects the spammish repetition'

```
>>> import hashlib
>>> m = hashlib.sha256()
>>> m.update(b"Nobody inspects")
>>> m.update(b" the spammish repetition")
>>> m.digest()
b'\x03\x1e\xdd}Ae\x15\x93\xc5\xfe\\\x00o\xa5u+7\xfd\xdf\xf7\xbcN\x84:\xa6\xaf\x0c\x95\x0f
>>> m.digest_size
32
>>> m.block_size
64
```

更浓缩：

```
>>> hashlib.sha224(b"Nobody inspects the spammish repetition").hexdigest()
'a4337bc45a8fc544c03f52dc550cd6e1e87021bc896588bd79e901e2'
```

hashlib.**new**(*名称*[, *数据*])

是一个通用构造函数，它将所需算法的字符串名称作为其第一个参数。它也存在允许访问上面列出的散列以及OpenSSL库可能提供的任何其他算法。命名的构造函数比它更快new() 并且应该是首选。

使用new()OpenSSL提供的算法：

```
>>> h = hashlib.new('ripemd160')
>>> h.update(b"Nobody inspects the spammish repetition")
>>> h.hexdigest()
'cc4a5ce1b3df48aec5d22d1f16b894a0b894eccc'
```

Hashlib提供了以下常量属性：

hashlib.**algorithms_guaranteed**

包含散列算法名称的集合保证在所有平台上由该模块支持。请注意，尽管一些上游供应商提供了一个奇怪的“符合FIPS”的Python版本，但排除了它，“md5”仍在此列表中。

3.2版本中的新功能

hashlib.**algorithms_available**

包含正在运行的Python解释器中可用的哈希算法名称的集合。传递给这些名称将被识别new()。 algorithms_guaranteed将永远是一个子集。相同的算法可能会在不同的名称下出现多次（感谢OpenSSL）。

3.2版本中的新功能

以下值作为构造函数返回的哈希对象的常量属性提供：

hash.**digest_size**

结果散列的大小（以字节为单位）。

hash.**block_size**

散列算法的内部块大小（以字节为单位）。

哈希对象具有以下属性：

hash.**name**

该散列的规范名称，始终为小写，并始终适合作为new()创建此类型的另一个散列的参数。

*在版本3.4中进行了更改：*自从CPython诞生以来，名称属性一直存在于CPython中，但是直到Python 3.4没有正式指定，因此在某些平台上可能不存在。

哈希对象具有以下方法：

hash.**update**(*arg*)

用对象*arg*更新哈希对象，该对象必须可解释为字节缓冲区。重复的调用相当于所有参数串联的单个调用：相当于。 `m.update(a); m.update(b)` `m.update(a+b)`

*在版本3.1中进行了更改：*当使用OpenSSL提供的散列算法时，发布Python GIL以允许其他线程运行，同时对大于2047字节的数据进行散列更新。

hash.**digest**( )

返回传递给该`update()`方法的数据的摘要。这是一个字节大小的对象`digest_size`，可能包含从0到255的整个范围内的字节。

hash.**hexdigest**( )

就像`digest()`除了摘要是以双倍长度的字符串对象的形式返回的，只包含十六进制数字。这可以用于在电子邮件或其他非二进制环境中安全地交换该值。

hash.**copy**( )

返回散列对象的副本（“克隆”）。这可用于高效计算共享初始子字符串的数据摘要。

15.1.2。SHAKE变量长度摘要

的`shake_128()`和`shake_256()`算法提供可变长度的消化物与length_in_bits // 2到安全的128或256位。因此，他们的摘要方法需要一定的篇幅。最大长度不受SHAKE算法的限制。

shake.**digest**(*长度*)

返回传递给该`update()`方法的数据的摘要。这是一个字节大小的对象`length`，可能包含从0到255的整个范围内的字节。

shake.**hexdigest**(*长度*)

就像`digest()`除了摘要是以双倍长度的字符串对象的形式返回的，只包含十六进制数字。这可以用于在电子邮件或其他非二进制环境中安全地交换该值。

15.1.3。重要推导

密钥派生和密钥扩展算法是为安全密码散列而设计的。天真的算法`sha1(password)`不能抵抗强力攻击。一个好的密码哈希函数必须是可调的，速度慢，并且包含一个盐。

hashlib.**pbkdf2_hmac**(*hash_name*，*密码*，*盐*，*迭代*，*dklen* =*无*)

该函数提供了PKCS＃5基于密码的密钥导出函数2.它使用HMAC作为伪随机函数。

字符串*hash_name*是HMAC散列摘要算法的所需名称，例如'sha1'或'sha256'。*密码*和*salt*被解释为字节的缓冲区。应用程序和库应该将*密码*限制在合理的长度（例如1024）。*盐*应该来自合适来源的大约16或更多字节，例如`os.urandom()`。

的数目*迭代*应当基于散列算法和计算能力来选择。截至2013年，建议至少进行10万次以上的SHA-256迭代。

*dklen*是派生键的长度。如果*为dkLen*被`None`然后散列算法的摘要大小*hash_name*被使用，例如64，用于SHA-512。

```
>>> import hashlib, binascii
>>> dk = hashlib.pbkdf2_hmac('sha256', b'password', b'salt', 100000)
>>> binascii.hexlify(dk)
b'0394a2ede332c9a13eb82e9b24631604c31df978b4e2f0fbd2c549944f9d79a5'
```

3.4版新增功能

注意： OpenSSL提供了*pbkdf2_hmac*的快速实现。Python实现使用一个内联版本hmac。它大约慢了三倍，并且不释放GIL。

hashlib.scrypt（*密码*，*，*salt*，*n*，*r*，*p*，*maxmem = 0*，*dklen = 64*）

该函数提供了基于密码的基于密码的密钥派生函数 **RFC 7914**。

*密码*和*salt*必须是类似字节的对象。应用程序和库应该将*密码*限制在合理的长度（例如1024）。*盐*应该来自合适来源的大约16或更多字节，例如os.urandom()。

*n*是CPU /内存成本因子，*r*块大小，*p*并行化因子和*maxmem*限制内存（OpenSSL 1.1.0默认为32 MB）。*dklen*是派生键的长度。

可用性：OpenSSL 1.1+

3.6版本中的新功能。

15.1.4。BLAKE2

BLAKE2是RFC-7693中定义的加密哈希函数，具有两种风格：

- **BLAKE2b**针对64位平台进行了优化，并生成1到64个字节之间的任意大小的摘要，
- **BLAKE2s**针对8位到32位平台进行了优化，并生成1到32字节之间的任何大小的摘要。

BLAKE2支持**键控模式**（HMAC更快更简单的替换），**盐化哈希**，**个性化**和**树哈希**。

来自该模块的哈希对象遵循标准库hashlib对象的API。

15.1.4.1。创建哈希对象

新的哈希对象通过调用构造函数来创建：

hashlib.blake2b（*data = b''*，*digest_size = 64*，*key = b''*，*salt = b''*，*person = b''*，*fanout = 1*，*depth = 1*，*leaf_size = 0*，*node_offset = 0*，*node_depth = 0*，*inner_size = 0*，*last_node = False*）

hashlib.blake2s（*data = b''*，*digest_size = 32*，*key = b''*，*salt = b''*，*person = b''*，*fanout = 1*，*depth = 1*，*leaf_size = 0*，*node_offset = 0*，*node_depth = 0*，*inner_size = 0*，*last_node = False*）

这些函数返回用于计算BLAKE2b或BLAKE2s的相应散列对象。他们可以选择这些一般参数：

- *数据*：散列*数据的*初始块，必须可解释为字节缓冲区。
- *digest_size*：输出摘要的大小（以字节为单位）。
- *key*：用于密钥散列的密钥（BLAKE2b最多64个字节，BLAKE2最多32个字节）。

- *盐*：用于随机哈希的盐（对于BLAKE2b最多16个字节，对于BLAKE2最多8个字节）。
- *person*：个性化字符串（BLAKE2b最多16个字节，BLAKE2最多8个字节）。

下表显示了一般参数的限制（以字节为单位）：

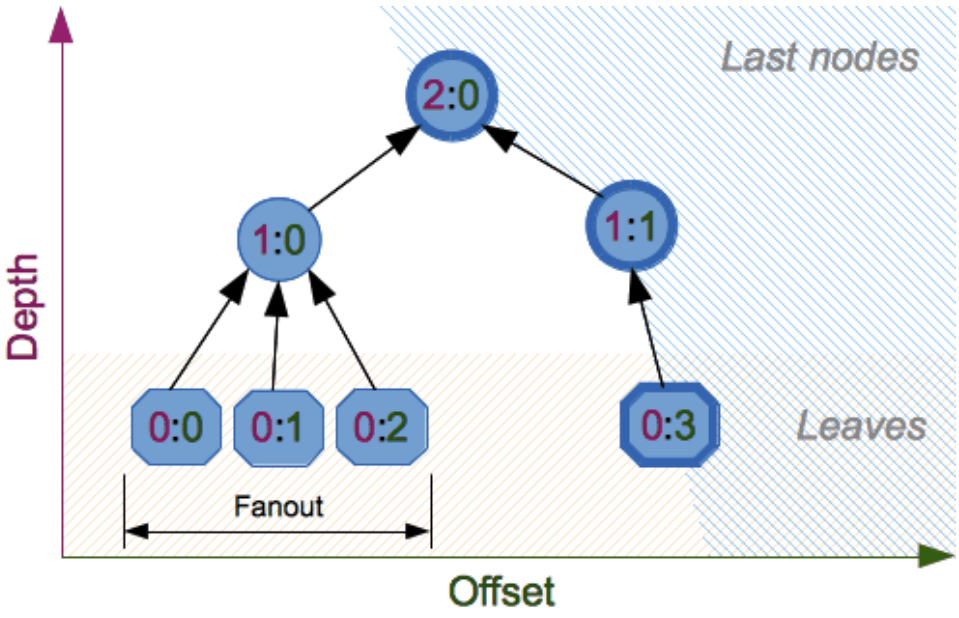
哈希	digest_size	LEN（钥匙）	LEN（盐）	LEN（人）
BLAKE2b	64	64	16	16
BLAKE2s	32	32	8	8

注意： BLAKE2规范为salt和个性化参数定义了恒定的长度，但是，为了方便起见，该实现接受任何大小的字节串，直到指定的长度。如果参数的长度小于指定的长度，则用零填充，例如，`b'salt'` 并且`b'salt\x00'` 是相同的值。（这不是*关键*的情况。）

这些大小可以作为模块常量使用，如下所述。

构造函数还接受以下树哈希参数：

- *扇出*：扇出（0至255,0如果无限制，则在顺序模式下为1）。
- *深度*：树的最大深度（1到255,255如果无限制，则在顺序模式下为1）。
- *leaf_size*：叶的最大字节长度（0到2 ** *32-1,0*如果无限制或在顺序模式下）。
- *node_offset*：节点偏移量（对于BLAKE2b，0到2 ** 64-1,0到2 ** 48-1对于BLAKE2s，0对于第一，最左边，叶子或顺序模式）。
- *node_depth*：节点深度（0到*255,0*树叶或顺序模式）。
- *inner_size*：内部摘要大小（BLAKE2b为0至64，BLAKE2为0至32，顺序模式为0）。
- *last_node*：布尔值，指示处理后的节点是否是最后一个节点（对于顺序模式为*False*）。



请参阅BLAKE2规范中的第2.10节，以全面查看树散列。

15.1.4.2。常量

blake2b. SALT_SIZE

blake2s. SALT_SIZE

盐长度（构造函数接受的最大长度）。

blake2b.PERSON_SIZE

blake2s.PERSON_SIZE

个性化字符串长度（构造函数接受的最大长度）。

blake2b.MAX_KEY_SIZE

blake2s.MAX_KEY_SIZE

最大密钥大小。

blake2b.MAX_DIGEST_SIZE

blake2s.MAX_DIGEST_SIZE

散列函数可输出的最大摘要大小。

15.1.4.3。示例

15.1.4.3.1。简单哈希

要计算一些数据的哈希值，您应该首先通过调用相应的构造函数（blake2b()或blake2s()）来构造一个哈希对象，然后通过调用update()该对象来更新数据，最后通过调用 digest()（或hexdigest()用于十六进制编码的字符串）。

```
>>> from hashlib import blake2b
>>> h = blake2b()
>>> h.update(b'Hello world')
>>> h.hexdigest()
'6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f33343a8c65551134ed1ae0f2b0
```

作为一个捷径，您可以传递第一个数据块作为第一个参数直接更新到构造函数（或者作为*data*关键字参数）：

```
>>> from hashlib import blake2b
>>> blake2b(b'Hello world').hexdigest()
'6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f33343a8c65551134ed1ae0f2b0
```

您可以hash.update()根据需要多次调用迭代更新哈希：

```
>>> from hashlib import blake2b
>>> items = [b'Hello', b' ', b'world']
>>> h = blake2b()
>>> for item in items:
...     h.update(item)
>>> h.hexdigest()
'6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f33343a8c65551134ed1ae0f2b0
```

15.1.4.3.2。使用不同的摘要大小

对于BLAKE2b，BLAKE2具有可配置的摘要大小，最大为64个字节，对于BLAKE2，最大为32个字节。例如，要用BLAKE2b替换SHA-1而不改变输出大小，我们可以告诉BLAKE2b产生20个字节的摘要：

```
>>> from hashlib import blake2b
>>> h = blake2b(digest_size=20)
>>> h.update(b'Replacing SHA1 with the more secure function')
>>> h.hexdigest()
'd24f26cf8de66472d58d4e1b1774b4c9158b1f4c'
>>> h.digest_size
20
>>> len(h.digest())
20
```

具有不同摘要大小的哈希对象具有完全不同的输出（较短的哈希*不是*较长哈希的前缀）；即使输出长度相同，BLAKE2b和BLAKE2s也会产生不同的输出：

```
>>> from hashlib import blake2b, blake2s
>>> blake2b(digest_size=10).hexdigest()
'6fa1d8fcfd719046d762'
>>> blake2b(digest_size=11).hexdigest()
'eb6ec15daf9546254f0809'
>>> blake2s(digest_size=10).hexdigest()
'1bf21a98c78a1c376ae9'
>>> blake2s(digest_size=11).hexdigest()
'567004bf96e4a25773ebf4'
```

15.1.4.3.3。键控哈希

键控哈希可用于身份验证，作为基于哈希的消息身份验证代码（HMAC）的更快更简单的替代方法。由于从BLAKE继承的不可分性属性，BLAKE2可以安全地用于前缀MAC模式。

此示例显示如何使用密钥获取带有密钥的消息（十六进制编码）的128位认证码：b'message data' b'pseudorandom key'

```
>>> from hashlib import blake2b
>>> h = blake2b(key=b'pseudorandom key', digest_size=16)
>>> h.update(b'message data')
>>> h.hexdigest()
'3d363ff7401e02026f4a4687d4863ced'
```

作为一个实际的例子，Web应用程序可以对发送给用户的cookies进行对称签名，并稍后验证它们以确保它们未被篡改：

```
>>> from hashlib import blake2b
>>> from hmac import compare_digest
>>>
>>> SECRET_KEY = b'pseudorandomly generated server secret key'
>>> AUTH_SIZE = 16
>>>
>>> def sign(cookie):
...     h = blake2b(digest_size=AUTH_SIZE, key=SECRET_KEY)
```

```
...     h.update(cookie)
...     return h.hexdigest().encode('utf-8')
>>>
>>> def verify(cookie, sig):
...     good_sig = sign(cookie)
...     return compare_digest(good_sig, sig)
>>>
>>> cookie = b'user-alice'
>>> sig = sign(cookie)
>>> print("{0},{1}".format(cookie.decode('utf-8'), sig))
user-alice,b'43b3c982cf697e0c5ab22172d1ca7421'
>>> verify(cookie, sig)
True
>>> verify(b'user-bob', sig)
False
>>> verify(cookie, b'0102030405060708090a0b0c0d0e0f00')
False
```

即使有本地密钥散列模式，BLAKE2当然也可以用于hmac模块的HMAC构建：

```
>>> import hmac, hashlib
>>> m = hmac.new(b'secret key', digestmod=hashlib.blake2s)
>>> m.update(b'message')
>>> m.hexdigest()
'e3c8102868d28b5ff85fc35dda07329970d1a01e273c37481326fe0c861c8142'
```

15.1.4.3.4。随机哈希

通过设置*salt*参数，用户可以将散列函数引入随机化。随机哈希可用于防止数字签名中使用的哈希函数的冲突攻击。

> 随机哈希是为消息准备人员生成全部或部分消息由第二方即消息签名人签名的情况而设计的。如果消息编写者能够找到密码散列函数冲突（即，产生相同散列值的两条消息），那么她可以准备消息的有意义的版本，以产生相同的散列值和数字签名，但具有不同的结果（例如，，转账1,000,000美元到一个账户，而不是10美元）。加密散列函数已被设计为具有抗碰撞性作为主要目标，但是目前对攻击加密散列函数的集中可能导致给定的加密散列函数提供比预期更小的抗碰撞抵抗。随机哈希通过减少准备人员在数字签名生成过程中生成两个或更多最终产生相同哈希值的消息的可能性，提供了签名者额外的保护 - 即使找到哈希函数的冲突是切实可行的。然而，当签名者准备好消息的所有部分时，随机哈希的使用可以减少由数字签名提供的安全性。
>
> （NIST SP-800-106“数字签名的随机哈希”）

在BLAKE2中，salt在初始化期间作为散列函数的一次性输入进行处理，而不是作为每个压缩函数的输入。

警告： 使用BLAKE2或任何其他通用加密散列函数（如SHA-256）进行*哈希散列*（或只是散列）不适合散列密码。有关更多信息，请参阅BLAKE2常见问题。

```
>>> import os
>>> from hashlib import blake2b
>>> msg = b'some message'
>>> # Calculate the first hash with a random salt.
>>> salt1 = os.urandom(blake2b.SALT_SIZE)
```

```
>>> h1 = blake2b(salt=salt1)
>>> h1.update(msg)
>>> # Calculate the second hash with a different random salt.
>>> salt2 = os.urandom(blake2b.SALT_SIZE)
>>> h2 = blake2b(salt=salt2)
>>> h2.update(msg)
>>> # The digests are different.
>>> h1.digest() != h2.digest()
True
```

15.1.4.3.5。个性化

有时候强制散列函数为不同的目的为相同的输入产生不同的摘要是有用的。引用Skein散列函数的作者：

> 我们建议所有应用程序设计人员认真考虑这样做;我们已经看到很多协议，其中在协议的一部分中计算的哈希可以用在完全不同的部分，因为两个哈希计算是在相似或相关数据上完成的，攻击者可以强制应用程序将哈希输入相同。对协议中使用的每个散列函数进行个性化处理，可以立即停止这种类型的攻击。
>
> （Skein Hash函数族，第21页）

BLAKE2可以通过将字节传递给*person*参数来进行个性化设置：

```
>>> from hashlib import blake2b
>>> FILES_HASH_PERSON = b'MyApp Files Hash'
>>> BLOCK_HASH_PERSON = b'MyApp Block Hash'
>>> h = blake2b(digest_size=32, person=FILES_HASH_PERSON)
>>> h.update(b'the same content')
>>> h.hexdigest()
'20d9cd024d4fb086aae819a1432dd2466de12947831b75c5a30cf2676095d3b4'
>>> h = blake2b(digest_size=32, person=BLOCK_HASH_PERSON)
>>> h.update(b'the same content')
>>> h.hexdigest()
'cf68fb5761b9c44e7878bfb2c4c9aea52264a80b75005e65619778de59f383a3'
```

个性化与键控模式一起也可用于从单个模式导出不同的密钥。

```
>>> from hashlib import blake2s
>>> from base64 import b64decode, b64encode
>>> orig_key = b64decode(b'Rm5EPJai72qcK3RGBpW3vPNfZy5OZothY+kHY6h21KM=')
>>> enc_key = blake2s(key=orig_key, person=b'kEncrypt').digest()
>>> mac_key = blake2s(key=orig_key, person=b'kMAC').digest()
>>> print(b64encode(enc_key).decode('utf-8'))
rbPb15S/Z9t+agffno5wuhB77VbRi6F9Iv2qIxU7WHw=
>>> print(b64encode(mac_key).decode('utf-8'))
G9GtHFE1YluXY1zWPlYk1e/nWfu0WSEb0KRcjhDeP/o=
```

15.1.4.3.6。树模式

下面是一个用两个叶节点对最小树进行散列的例子：

```
  10
 /  \
00  01
```

本示例使用64字节的内部摘要，并返回32字节的最终摘要：

```
>>> from hashlib import blake2b
>>>
>>> FANOUT = 2
>>> DEPTH = 2
>>> LEAF_SIZE = 4096
>>> INNER_SIZE = 64
>>>
>>> buf = bytearray(6000)
>>>
>>> # Left leaf
... h00 = blake2b(buf[0:LEAF_SIZE], fanout=FANOUT, depth=DEPTH,
...                leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...                node_offset=0, node_depth=0, last_node=False)
>>> # Right leaf
... h01 = blake2b(buf[LEAF_SIZE:], fanout=FANOUT, depth=DEPTH,
...                leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...                node_offset=1, node_depth=0, last_node=True)
>>> # Root node
... h10 = blake2b(digest_size=32, fanout=FANOUT, depth=DEPTH,
...                leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...                node_offset=0, node_depth=1, last_node=True)
>>> h10.update(h00.digest())
>>> h10.update(h01.digest())
>>> h10.hexdigest()
'3ad2a9b37c6070e374c7a8c508fe20ca86b6ed54e286e93a0318e95e881db5aa'
```

15.1.4.4。学分

BLAKE2 由 *Jean-Philippe Aumasson* ， *Samuel Neves* ， *Zooko Wilcox-O'Hearn* 和 *Christian Winnerlein设计*，基于 由*Jean-Philippe Aumasson*，*Luca Henzen*，*Willi Meier*和 *Raphael C. -W* 创建的SHA-3入围BLAKE。*潘*。

它使用由*Daniel* J.Bernstein设计的ChaCha密码的核心算法。

stdlib实现基于pyblake2模块。它由 *Dmitry Chestnykh*根据*Samuel Neves*编写的C实现书写。该文档是从pyblake2复制而来，由*Dmitry Chestnykh编写*。

C代码部分由*Christian Heimes*为Python重写。

以下公共领域专用适用于C哈希函数实现，扩展代码和本文档：

> 在法律允许的范围内，作者已将本软件的所有版权及相关权利和邻接权利授予全球公有领域。该软件的发布没有任何担保。
>
> 您应该已经收到CC0公共领域奉献的副本以及此软件。如果没有，请参阅 http://creativecommons.org/publicdomain/zero/1.0/。

根据Creative Commons Public Domain Dedication 1.0 Universal的规定，以下人员协助开发项目或为项目和公共领域做出贡献：

- *Alexandr Sokolovskiy*

也可以看看:

模 hmac
使用散列生成消息认证码的模块。

模 base64
另一种为非二进制环境编码二进制散列的方法。

https://blake2.net
官方BLAKE2网站。

http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf
关于安全散列算法的FIPS 180-2出版物。

https://en.wikipedia.org/wiki/Cryptographic_hash_function#Cryptographic_hash_algorithms
维基百科文章，提供有关哪些算法具有已知问题的信息，以及它们在使用中的含义。

https://www.ietf.org/rfc/rfc2898.txt
PKCS＃5：基于密码的密码规范2.0版

15.2。 hmac- 消息认证的键控哈希

源代码： Lib / hmac.py

该模块实现了如所描述的HMAC算法 **RFC 2104**。

hmac.new（*key*，*msg = None*，*digestmod = None* ）

返回一个新的hmac对象。 *密钥*是给出秘密密钥的字节或字节数组对象。如果*msg*存在，则进行方法调用update(msg)。 *digestmod*是HMAC对象使用的摘要名称，摘要构造函数或模块。它支持任何适合hashlib.new()和默认hashlib.md5构造函数的名称。

*版本3.4中更改：参数键*可以是字节或字节数组对象。参数*msg*可以是任何支持的类型hashlib。参数*digestmod*可以是散列算法的名称。

*自3.4版以后*不推荐使用：MD5作为*digestmod的*隐式默认摘要已弃用。

HMAC对象具有以下方法：

HMAC.update（*味精*）

用*msg*更新hmac对象。重复的调用相当于所有参数串联的单个调用： 相当于。 m.update(a)； m.update(b) m.update(a + b)

*在版本3.4中更改：参数msg*可以是任何类型支持的hashlib。

HMAC.digest（ ）

返回传递给该update()方法的字节的摘要。这个字节对象的长度与赋予构造函数的摘要的*digest_size*的长度相同。它可能包含非ASCII字节，包括NUL字节。

警告： 在digest()验证例程期间，当比较外部提供摘要的输出时，建议使用该compare_digest()功能而不是==操作员来降低定时攻击的脆弱性。

HMAC.hexdigest（ ）

就像digest()除了摘要以长度的两倍返回包含十六进制数字的长度。这可以用于在电子邮件或其他非二进制环境中安全地交换该值。

警告： 在hexdigest()验证例程期间，当比较外部提供摘要的输出时，建议使用该compare_digest()功能而不是==操作员来降低定时攻击的脆弱性。

HMAC.copy（ ）

返回hmac对象的副本（“克隆”）。这可用于有效计算共享初始子字符串的字符串摘要。

哈希对象具有以下属性：

HMAC.digest_size

生成的HMAC摘要的大小（以字节为单位）。

HMAC.**block_size**

散列算法的内部块大小（以字节为单位）。

3.4版新增功能

HMAC.**name**

这个HMAC的规范名称，总是小写，例如hmac-md5。

3.4版新增功能

该模块还提供以下帮助功能：

hmac.**compare_digest**(*a*, *b*)

返回。此功能采用旨在通过避免基于内容的短路行为来防止时序分析的方法，使其适用于密码学。 *a*和*b*必须是相同的类型：或者（仅用于ASCII，例如返回 ）或类似字节的对象。

a == bstrHMAC.hexdigest()

> **注意：** 如果*a*和*b*的长度不同，或者发生错误，定时攻击理论上可以揭示*a*和*b*类型和长度的信息- 但不是它们的值。

3.3版本的新功能

也可以看看:

模 hashlib

Python模块提供安全的散列函数。

15.3。secrets- 生成用于管理机密的安全随机数

3.6版本中的新功能。

源代码： Lib / secrets.py

该secrets模块用于生成适合管理密码，帐户身份验证，安全令牌和相关机密等数据的密码性强随机数。

特别是，secrets应优先使用模块中的默认伪随机数生成器，该random模块专为建模和仿真而设计，而不是安全或密码学。

也可以看看： **PEP 506**

15.3.1。随机数字

该secrets模块提供对操作系统提供的最安全的随机性来源的访问。

*类*secrets.SystemRandom
使用操作系统提供的最高质量源生成随机数的类。查看 random.SystemRandom更多详情。

secrets.choice(*序列*)
从非空序列中返回一个随机选择的元素。

secrets.randbelow(*n*)
返回范围[0，*n*）中的一个随机int 。

secrets.randbits(*k*)
用*k*个随机位返回一个int 。

15.3.2。生成令牌

该secrets模块提供了生成安全令牌的功能，适用于密码重置，难以猜测的URL等类似应用。

secrets.token_bytes([*nbytes* = *None*])
返回一个包含*nbytes*字节数的随机字节字符串。如果*nbytes*是None或不是，则使用合理的默认值。

```
>>> token_bytes(16)
b'\xebr\x17D*t\xae\xd4\xe3S\xb6\xe2\xebP1\x8b'
```

secrets.token_hex([*nbytes* = *None*])

以十六进制形式返回随机文本字符串。该字符串具有*nbytes* 随机字节，每个字节转换为两个十六进制数字。如果*nbytes*是 None或不是，则使用合理的默认值。

```
>>> token_hex(16)
'f9bf78b9a18ce6d46a0cd2b0b86df9da'
```

secrets.**token_urlsafe**([*nbytes = None*])

返回一个随机的URL安全文本字符串，包含*nbytes*随机字节。文本是Base64编码的，所以平均每个字节的结果大约为1.3个字符。如果*nbytes*是None或不是，则使用合理的默认值。

```
>>> token_urlsafe(16)
'Drmhze6EPcv0fN_81Bj-nA'
```

15.3.2.1。令牌应该使用多少字节？

为了防止 暴力攻击，令牌需要具有足够的随机性。不幸的是，随着计算机变得越来越强大，并且能够在更短的时间内做出更多猜测，所认为的充分必然会增加。截至2015年，相信32个字节（256位）的随机性足以满足secrets模块预期的典型使用情况。

对于那些想要管理自己的令牌长度的人，可以通过给int 各种token_*函数提供参数来明确指定令牌的多少随机性。这个参数被认为是随机使用的字节数。

否则，如果没有提供参数，或者参数是None ，token_*函数将使用合理的默认值。

注意： 该默认值随时可能发生变化，包括在维护版本中。

15.3.3。其他功能

secrets.**compare_digest**(*a*, *b*)

True如果字符串*a*和*b*相等则返回，否则False以减少定时攻击的风险的方式返回 。查看hmac.compare_digest()更多详情。

15.3.4。食谱和最佳做法

本节介绍用于secrets 管理基本安全级别的配方和最佳实践。

生成一个八字符的字母数字密码：

```
import string
alphabet = string.ascii_letters + string.digits
password = ''.join(choice(alphabet) for i in range(8))
```

注意： 应用程序不应 以可恢复的格式存储密码，无论是纯文本还是加密的。应该使用密码强的单向（不可逆）散列函数对它们进行腌制和散列。

生成至少包含一个小写字母，至少一个大写字母和至少三个数字的十个字符的字母数字密码：

```
import string
alphabet = string.ascii_letters + string.digits
while True:
    password = ''.join(choice(alphabet) for i in range(10))
    if (any(c.islower() for c in password)
            and any(c.isupper() for c in password)
            and sum(c.isdigit() for c in password) >= 3):
        break
```

生成XKCD风格的密码短语：

```
# On standard Linux systems, use a convenient dictionary file.
# Other platforms may need to provide their own word-list.
with open('/usr/share/dict/words') as f:
    words = [word.strip() for word in f]
    password = ' '.join(choice(words) for i in range(4))
```

生成一个难以猜到的临时URL，其中包含适用于密码恢复应用程序的安全令牌：

```
url = 'https://mydomain.com/reset=' + token_urlsafe()
```

16.通用操作系统服务

本章介绍的模块提供了（几乎）所有操作系统（如文件和时钟）上可用的操作系统功能的接口。这些接口通常是在Unix或C接口之后建模的，但它们也可以在大多数其他系统上使用。这里有一个概述：

16.1。 os- 其他操作系统接口

源代码： Lib / os.py

该模块提供了一种使用与操作系统相关的功能的便携方式。如果您只想读取或写入文件，请参阅open()，如果要操作路径，请参阅该os.path模块，并且如果要读取命令行上所有文件中的所有行，请参阅该fileinput 模块。有关创建临时文件和目录的信息，请参阅tempfile 模块，高级文件和目录处理请参阅shutil 模块。

关于这些功能的可用性的说明：

- Python的所有内置操作系统相关模块的设计都是这样的，只要具有相同的功能，它就使用相同的接口; 例如，该函数以相同的格式os.stat(path)返回有关*路径的*统计信息（恰好源于POSIX接口）。
- 通过该os模块也可以使用特定操作系统特有的扩展，但使用它们当然会对可移植性构成威胁。
- 所有接受路径或文件名的函数接受字节和字符串对象，并且如果返回路径或文件名，则会生成相同类型的对象。
- “可用性：Unix”说明意味着这个函数通常在Unix系统上找到。它没有在特定的操作系统上声称它的存在。
- 如果没有单独说明，那么声称“可用性：Unix”的所有功能都支持在基于Unix核心的Mac OS X上。

注意： 在OSError无效或无法访问的文件名和路径或具有正确类型但操作系统不接受的其他参数的情况下，此模块中的所有功能都会引发。

*异常*os.error
内置OSError异常的别名。

os.name
导入的操作系统相关模块的名称。下列名称当前已注册：'posix'，'nt'，'java'。

也可以看看： sys.platform具有更精细的粒度。 os.uname()给出系统相关的版本信息。该platform模块提供了系统身份的详细检查。

16.1.1。文件名，命令行参数和环境变量

在Python中，文件名，命令行参数和环境变量使用字符串类型表示。在某些系统上，在将这些字符串传递给操作系统之前，需要将这些字符串解码为字节和从字节解码。Python使用文件系统编码来执行此转换（请参阅 sys.getfilesystemencoding()）。

*版本3.1中更改：*在某些系统上，使用文件系统编码进行转换可能会失败。在这种情况下，Python使用surrogateescape编码错误处理程序，这意味着解码时将不可解码的字节替换为Unicode字符U + DCxx，并将这些字符再次转换为编码时的原始字节。

文件系统编码必须保证成功解码128以下的所有字节。如果文件系统编码无法提供此保证，则API函数可能会引发UnicodeErrors。

16.1.2。处理参数

这些功能和数据项提供信息并对当前过程和用户进行操作。

os.ctermid（）
返回进程控制终端对应的文件名。

可用性：Unix。

os.environ
甲映射表示字符串环境对象。例如， environ['HOME']是您的主目录（在某些平台上）的路径名，并且等同getenv("HOME")于C.

这种映射是在os模块首次导入时捕获的，通常在Python启动时作为处理的一部分site.py。在这段时间之后对环境所做的更改不会反映到os.environ，但通过os.environ直接修改所做的更改除外。

如果平台支持该putenv()功能，则该映射可用于修改环境以及查询环境。 putenv()将在修改映射时自动调用。

在Unix上，键和值使用sys.getfilesystemencoding()和 'surrogateescape'错误处理程序。使用environb，如果你想使用不同的编码。

注意： putenv()直接调用不会改变os.environ，所以最好修改os.environ。

注意： 在某些平台上，包括FreeBSD和Mac OS X，设置environ可能会导致内存泄漏。请参阅系统文档 putenv()。

如果putenv()未提供，则可以将此映射的修改副本传递给相应的流程创建功能，以使子流程使用修改后的环境。

如果平台支持该unsetenv()功能，则可以删除此映射中的项目以取消设置环境变量。unsetenv()当一个项目被删除从将被自动调用os.environ，并且当所述一个pop()或clear()方法被调用。

os.environb
字节版本environ：表示环境为字节字符串的映射对象。 environ并environb同步（修改environb更新environ，反之亦然）。

environb仅当supports_bytes_environ为True 时才可用。

3.2版本中的新功能

os.chdir（*路径*）
os.fchdir（*fd*）

os. getcwd ( )
这些功能在“ 文件和目录”中介绍。

os. fsencode (*文件名*)
编码路径类似 *的文件名*与文件系统的编码’surrogateescape’错误处理程序，或’strict’在Windows上; 回报bytes不变。

fsdecode() 是相反的功能。

3.2版本中的新功能

*在版本3.6中进行了更改：*添加了支持以接受实现该os.PathLike 接口的对象。

os. fsdecode (*文件名*)
使用错误处理程序或 在Windows上解码 *文件*系统编码的路径类 *文件名* ；回报不变。’surrogateescape’’strict’ str

fsencode() 是相反的功能。

3.2版本中的新功能

*在版本3.6中进行了更改：*添加了支持以接受实现该os.PathLike 接口的对象。

os. fspath (*路径*)
返回路径的文件系统表示。

如果str或者bytes被传入，则返回原样。否则将__fspath__()被调用，并且只要它是a str 或bytesobject ，它的值就会返回。在所有其他情况下，都会TypeError被提出。

3.6版本中的新功能。

*类*os. PathLike
表示文件系统路径的对象的抽象基类，例如pathlib.PurePath。

3.6版本中的新功能。

abstractmethod __fspath__ ( )
返回对象的文件系统路径表示。

该方法应该只返回一个str或一个bytes对象，并且首选项是str。

os. getenv (*键*, *默认=无*)
如果它存在，则返回环境变量*键*的值; 如果不存在，则 返回*默认值*。 *键*，*默认*和结果是str。

在Unix上，键和值用解码器sys.getfilesystemencoding() 和’surrogateescape’错误处理器解码。使用os.getenvb() ，如果你想使用不同的编码。

可用性：最受欢迎的Unix，Windows。

os. getenvb (*键* , *默认=无*)
如果它存在，则返回环境变量*键*的值; 如果不存在，则 返回*默认值*。*键* , *默认*和结果是字节。

`getenvb()`仅当`supports_bytes_environ` 为True 时才可用。

可用性：大多数Unix的口味。

3.2版本中的新功能

os. get_exec_path (*env =无*)
在启动进程时，返回将搜索命名可执行文件的目录列表，类似于shell。 *ENV* ，指定时，应该是一个环境变量字典来查找在PATH默认情况下，当*ENV*是`None` ， `environ`被使用。

3.2版本中的新功能

os. getegid ( )
返回当前进程的有效组标识。这对应于当前进程中正在执行的文件上的“set id”位。

可用性：Unix。

os. geteuid ( )
返回当前进程的有效用户标识。

可用性：Unix。

os. getgid ( )
返回当前进程的实际组ID。

可用性：Unix。

os. getgrouplist (*用户* , *组*)
返回*用户*所属组ID的列表。如果*组*不在列表中，则将其包含在内; 通常，*群组*被指定为来自*用户*的密码记录的群组ID字段。

可用性：Unix。

3.3版本的新功能

os. getgroups ( )
返回与当前进程关联的补充组标识的列表。

可用性：Unix。

注意： 在Mac OS X上， `getgroups()`行为与其他Unix平台有所不同。如果Python解释器是使用`10.5`或更早的部署目标构建的，则`getgroups()`返回与当前用户进程关联的有效组ID的列表; 此列表仅限于系统定义的条目数量（通常为16），并且可以通过调用修改（ `setgroups()`如果具有适当的特权）。如果使用大于的部署目标构建`10.5` ，则`getgroups()`返回与进程的有效用户标识关联的用户的当前组访问列表; 组访问列表可能

在进程的生命周期中发生变化，它不受调用的影响setgroups()，其长度不限于16个。部署目标值MACOSX_DEPLOYMENT_TARGET可以使用sysconfig.get_config_var()。

os.getlogin()
返回在进程的控制终端上登录的用户的名称。对于大多数用途而言，getpass.getuser()由于后者检查环境变量，因此更有用 LOGNAME 要么 USERNAME找出用户是谁，然后回退到pwd.getpwuid(os.getuid())[0]获取当前真实用户标识的登录名。

可用性：Unix，Windows。

os.getpgid(*pid*)
用进程号*pid*返回进程的进程组ID。如果*pid*为0，则返回当前进程的进程组标识。

可用性：Unix。

os.getpgrp()
返回当前进程组的ID。

可用性：Unix。

os.getpid()
返回当前进程ID。

os.getppid()
返回父进程ID。当父进程退出时，在Unix上，返回的id是init进程（1）中的一个，在Windows上它仍然是相同的id，可能已被其他进程重用。

可用性：Unix，Windows。

*在版本3.2中更改：*添加了对Windows的支持。

os.getpriority(*其中*, *谁*)
获取计划调度优先级。的值，*其*是一个 PRIO_PROCESS，PRIO_PGRP或者PRIO_USER，和*谁* 相对于解释*该*（为一个进程标识符 PRIO_PROCESS，用于处理组标识符PRIO_PGRP和用户的ID为 PRIO_USER）。*who*的零值表示（分别）调用进程，调用进程的进程组或调用进程的真实用户标识。

可用性：Unix。

3.3版本的新功能

os.PRIO_PROCESS
os.PRIO_PGRP
os.PRIO_USER
getpriority()和setpriority()函数的参数。

可用性：Unix。

3.3版本的新功能

os. getresuid ( )
返回一个元组（ruid，euid，suid），表示当前进程的真实，有效和保存的用户id。

可用性：Unix。

3.2版本中的新功能

os. getresgid ( )
返回一个元组（rgid，egid，sgid），表示当前进程的真实，有效和保存的组ID。

可用性：Unix。

3.2版本中的新功能

os. getuid ( )
返回当前进程的真实用户ID。

可用性：Unix。

os. initgroups (*username*，*gid*)
调用系统initgroups（）初始化组访问列表，其中包含指定用户名所属的所有组以及指定的组ID。

可用性：Unix。

3.2版本中的新功能

os. putenv (*key*，*value*)
将名为*key*的环境变量设置为字符串*值*。对环境的这种变化影响开始与子进程 os.system()， popen()或fork()和execv()。

可用性：最受欢迎的Unix，Windows。

> **注意：** 在某些平台上，包括FreeBSD和Mac OS X，设置environ可能会导致内存泄漏。请参阅putenv的系统文档。

何时putenv()被支持，对项目的分配os.environ会自动转换为相应的呼叫putenv()；但是，调用putenv()不会更新os.environ，所以实际上最好分配给项目os.environ。

os. setegid (*egid*)
设置当前进程的有效组ID。

可用性：Unix。

os. seteuid (*euid*)
设置当前进程的有效用户标识。

可用性：Unix。

os. setgid (*gid*)

设置当前进程的组ID。

可用性：Unix。

os.setgroups（*组*）
将与当前进程关联的补充组标识列表设置为 *组*。*组*必须是一个序列，并且每个元素必须是标识组的整数。此操作通常仅适用于超级用户。

可用性：Unix。

> **注意：** 在Mac OS X上，*组*的长度不得超过系统定义的最大有效组ID的最大数量，通常为16个。有关`getgroups()`通过调用setgroups（）可能不会返回相同组列表的情况，请参阅文档。

os.setpgrp（）
调用系统调用`setpgrp()`或根据实现哪个版本（如果有）。有关语义的信息，请参阅Unix手册。`setpgrp(0, 0)`

可用性：Unix。

os.setpgid（*pid*，*pgrp*）
调用系统调用`setpgid()`以将id为*pid*的进程的进程组标识设置为标识为*pgrp*的进程组。有关语义的信息，请参阅Unix手册。

可用性：Unix。

os.setpriority（*其中*，*谁*，*优先级*）
设置节目安排优先级。的值，*其*是一个`PRIO_PROCESS`，`PRIO_PGRP`或者`PRIO_USER`，和*谁*相对于解释*该*（为一个进程标识符`PRIO_PROCESS`，用于处理组标识符`PRIO_PGRP`和用户的ID为`PRIO_USER`）。*who*的零值表示（分别）调用进程，调用进程的进程组或调用进程的真实用户标识。*优先级*是-20至19范围内的值。默认优先级为0; 低优先级会导致更有利的调度。

可用性：Unix

3.3版本的新功能

os.setregid（*rgid*，*egid*）
设置当前进程的真实和有效的组ID。

可用性：Unix。

os.setresgid（*rgid*，*egid*，*sgid*）
设置当前进程的真实，有效和保存的组ID。

可用性：Unix。

3.2版本中的新功能

os.setresuid（*ruid*，*euid*，*suid*）

设置当前进程的真实，有效和保存的用户标识。

可用性：Unix。

3.2版本中的新功能

os.**setreuid**(*ruid*, *euid*)
设置当前进程的真实和有效的用户ID。

可用性：Unix。

os.**getsid**(*pid*)
调用系统调用`getsid()`。有关语义的信息，请参阅Unix手册。

可用性：Unix。

os.**setsid**()
调用系统调用`setsid()`。有关语义的信息，请参阅Unix手册。

可用性：Unix。

os.**setuid**(*uid*)
设置当前进程的用户标识。

可用性：Unix。

os.**strerror**(*代码*)
返回对应于错误代码错误消息*代码*。在提供未知错误编号的情况下`strerror()`返回的平台上。`NULL``ValueError`

os.**supports_bytes_environ**
`True`如果环境的本机操作系统类型是字节（例如`False`在Windows上）。

3.2版本中的新功能

os.**umask**(*掩码*)
设置当前的数字umask并返回先前的umask。

os.**uname**()
返回标识当前操作系统的信息。返回值是一个具有五个属性的对象：

- `sysname` - 操作系统名称
- `nodename` - 网络上的机器名称（实施定义）
- `release` - 操作系统版本
- `version` - 操作系统版本
- `machine` - 硬件标识符

为了向后兼容，这个目的也可迭代，表现得像含有五元组`sysname`，`nodename`， `release`，`version`，和`machine` 以该顺序。

一些系统截断nodename为8个字符或导致组件；获取主机名的更好方法是socket.gethostname()甚至是socket.gethostbyaddr(socket.gethostname())。

可用性：最近的Unix版本。

*在版本3.3中进行了更改：*返回类型从元组更改为具有命名属性的元组对象。

os.unsetenv(*重点*)

取消设置（删除）名为*key*的环境变量。对环境的这种变化影响开始与子进程os.system()，popen()或fork()和execv()。

何时unsetenv()被支持，删除项目os.environ被自动翻译成相应的呼叫unsetenv()；但是，调用unsetenv()不会更新os.environ，因此实际上最好删除项目os.environ。

可用性：最受欢迎的Unix，Windows。

16.1.3。文件对象创建

这个函数创建新的文件对象。（另请参阅open()打开文件描述符。）

os.fdopen(*fd*，**args*，***kwargs*)

返回连接到文件描述符*fd*的打开的文件对象。这是open()内置函数的别名，并接受相同的参数。唯一的区别是第一个参数fdopen()必须总是一个整数。

16.1.4。文件描述符操作

这些函数对使用文件描述符引用的I / O流进行操作。

文件描述符是对应于当前进程打开的文件的小整数。例如，标准输入通常是文件描述符0，标准输出是1，标准错误是2.然后为进程打开的其他文件将分配3,4,5等等。名称“文件描述符”有点欺骗性; 在Unix平台上，套接字和管道也被文件描述符引用。

该fileno()方法可用于在需要时获取与文件对象关联的文件描述符。请注意，直接使用文件描述符将绕过文件对象方法，忽略诸如内部数据缓冲等方面。

os.close(*fd*)

关闭文件描述符*fd*。

> **注意：** 此函数用于低级别I / O，并且必须将其应用于由os.open()或返回的文件描述符pipe()。关闭内置函数返回的“文件对象”open()，popen()或者fdopen()使用或者 使用它的close()方法。

os.closerange(*fd_low*，*fd_high*)

关闭所有文件描述符，从*fd_low*（含）到*fd_high*（独占），忽略错误。相当于（但快得多）：

```
for fd in range(fd_low, fd_high):
    try:
        os.close(fd)
    except OSError:
        pass
```

os.device_encoding(*fd*)
如果连接到终端，则返回描述与*fd*关联的设备的编码的字符串; 否则返回None。

os.dup(*fd*)
返回文件描述符*fd*的副本。新的文件描述符是 不可继承的。

在Windows上，当复制标准流（0：stdin，1：stdout，2：stderr）时，新文件描述符是可继承的。

*在版本3.4中更改：*新的文件描述符现在是不可继承的。

os.dup2(*fd*，*fd2*，*inheritable* = *True*)
将文件描述符*fd*复制到*fd2*，必要时先关闭后者。文件描述符*fd2*默认是可继承的，如果*可继承*则是非*可继承*的False。

*在版本3.4中进行了更改：*添加可选的*可继承*参数。

os.fchmod(*fd*，*mode*)
将由*fd*给出的文件的模式更改为数字*模式*。请参阅文档以chmod()了解可能的*模式*值。从Python 3.3开始，这相当于。os.chmod(fd, mode)

可用性：Unix。

os.fchown(*fd*，*uid*，*gid*)
将由*fd*给出的文件的所有者和组标识更改为数字*uid* 和*gid*。要使其中一个ID保持不变，请将其设置为-1。看 chown()。从Python 3.3开始，这相当于。os.chown(fd, uid, gid)

可用性：Unix。

os.fdatasync(*fd*)
用filedescriptor *fd*强制将文件写入磁盘。不强制更新元数据。

可用性：Unix。

> **注意：** 此功能在MacOS上不可用。

os.fpathconf(*fd*，*name*)
返回与打开文件相关的系统配置信息。*name* 指定要检索的配置值; 它可能是一个字符串，它是定义的系统值的名称; 这些名称在许多标准（POSIX.1，Unix 95，Unix 98等）中都有详细说明。一些平台也定义了其他名称。pathconf_names字典中给出了主机操作系统已知的名称。对于未包含在该映射中的配置变量，也会接受传递*名称*的整数。

如果*名称*是一个字符串并且未知，ValueError则会引发。如果主机系统不支持*名称*的特定值，即使它包含在内pathconf_names，OSError也会errno.EINVAL为错误号引发 一个值。

从Python 3.3开始，这相当于。os.pathconf(fd, name)

可用性：Unix。

os.**fstat**(*fd*)

获取文件描述符*fd*的状态。返回一个stat_result 对象。

从Python 3.3开始，这相当于os.stat(fd)。

> **也可以看看：** 该stat()功能。

os.**fstatvfs**(*fd*)

返回有关含有与文件描述符相关的文件的文件系统信息*FD*一样statvfs()。从Python 3.3开始，这相当于os.statvfs(fd)。

可用性：Unix。

os.**fsync**(*fd*)

用filedescriptor *fd*强制将文件写入磁盘。在Unix上，它调用本地fsync()函数; 在Windows上，MS _commit()功能。

如果你从一个缓冲的Python 文件对象 *f开始*，首先做 f.flush()，然后做os.fsync(f.fileno())，以确保与*f*相关的所有内部缓冲区都被写入磁盘。

可用性：Unix，Windows。

os.**ftruncate**(*fd*, *长度*)

截断与文件描述符*fd*相对应的文件，以使其*长度*至多为*长度*字节。从Python 3.3开始，这相当于。os.truncate(fd, length)

可用性：Unix，Windows。

*版本3.5中更改：*增加了对Windows的支持

os.**get_blocking**(*fd*)

获取文件描述符的阻塞模式：False如果O_NONBLOCK标志被设置，True则标志被清除。

另见set_blocking()和socket.socket.setblocking()。

可用性：Unix。

3.5版本中的新功能。

os.**isatty**(*fd*)

返回True如果文件描述符*FD*是开放的，连接到一个TTY（类似的）设备，否则False。

os.**lockf**(*fd*, *cmd*, *len*)

在打开的文件描述符上应用，测试或移除POSIX锁。 *fd*是一个打开的文件描述符。 *CMD*指定命令来使用-的一个F_LOCK ，F_TLOCK ， F_ULOCK或F_TEST。 *len*指定要锁定的文件部分。

可用性：Unix。

3.3版本的新功能

os.F_LOCK
os.F_TLOCK
os.F_ULOCK
os.F_TEST
指定lockf()将采取什么操作的标志。

可用性：Unix。

3.3版本的新功能

os.lseek（*fd*，*pos*，*how* ）
将文件描述符*fd*的当前位置设置为位置*pos*，修改*方法*如下：SEEK_SET或0设置相对于文件开头的位置; SEEK_CUR或者1将其设置为相对于当前位置; SEEK_END或者2将其设置为相对于文件的结尾。从头开始，以字节为单位返回新的光标位置。

os.SEEK_SET
os.SEEK_CUR
os.SEEK_END
lseek()函数的参数。它们的值分别为0,1和2。

*V3.3中的新增功能：*某些操作系统可能支持其他值，例如 os.SEEK_HOLE或os.SEEK_DATA。

os.open（*path*，*flags*，*mode = 0o777*，*，dir_fd = None* ）
根据*模式*打开文件*路径*并根据*标志*和可能的模式设置各种标志。在计算*模式下*，当前的umask值首先被屏蔽掉。返回新打开的文件的文件描述符。新的文件描述符是不可继承的。

有关标志和模式值的说明，请参阅C运行时文档; 标志常量（如O_RDONLY和O_WRONLY ）在os模块中定义。特别是在Windows O_BINARY上，需要添加 以二进制模式打开文件。

该函数可以使用*dir_fd*参数支持与目录描述符相关的*路径*。

*在版本3.4中更改：*新的文件描述符现在是不可继承的。

> **注意：** 此功能适用于低级别的I / O。对于正常使用，使用内置函数open() ，该函数返回带有 和方法（以及更多）的文件对象。要将文件描述符包装到文件对象中，请使用。read() write() fdopen()

*版本3.3中的新增功能：*dir_fd参数。

*版本3.5中更改：*如果系统调用中断且信号处理程序不引发异常，则该函数现在重试系统调用而不是引发 InterruptedError异常（请参阅**PEP 475**为理由）。

*在版本3.6中更改：*接受类似路径的对象。

以下常量是函数的*flags*参数的 选项open()。它们可以使用按位或运算符进行组合｜。其中一些不适用于所有平台。有关它们的可用性和使用的说明，请参阅Unix上的*open（2）*手册页或Windows 上的MSDN。

os.O_RDONLY
os.O_WRONLY
os.O_RDWR
os.O_APPEND
os.O_CREAT
os.O_EXCL
os.O_TRUNC
上述常量可在Unix和Windows上使用。

os.O_DSYNC
os.O_RSYNC
os.O_SYNC
os.O_NDELAY
os.O_NONBLOCK
os.O_NOCTTY
os.O_CLOEXEC
上述常量仅在Unix上可用。

*在版本3.3中更改：*添加O_CLOEXEC常量。

os.O_BINARY
os.O_NOINHERIT
os.O_SHORT_LIVED
os.O_TEMPORARY
os.O_RANDOM
os.O_SEQUENTIAL
os.O_TEXT
上述常量仅在Windows上可用。

os.O_ASYNC
os.O_DIRECT
os.O_DIRECTORY
os.O_NOFOLLOW
os.O_NOATIME
os.O_PATH
os.O_TMPFILE
os.O_SHLOCK
os.O_EXLOCK
上述常量是扩展名，如果它们没有被C库定义，则不存在。

*在3.4版中更改：*添加O_PATH支持它的系统。添加O_TMPFILE，仅适用于Linux Kernel 3.11或更新版本。

os.openpty()

打开一个新的伪终端对。分别为pty和tty 返回一对文件描述符 。新的文件描述符是不可继承的。对于（稍微）更便携的方法，请使用该模块。(master, slave) pty

可用性：一些Unix的味道。

*在版本3.4中更改：*新的文件描述符现在是不可继承的。

os.pipe()

创建一个管道。分别返回一对可用于读写的文件描述符。新的文件描述符是 不可继承的。(r, w)

可用性：Unix，Windows。

*在版本3.4中更改：*新的文件描述符现在是不可继承的。

os.pipe2(*标志*)

创建一个带有原子设置*标志*的管道。 可以通过将这些值中的一个或多个ORing在一起来构造*标志*： O_NONBLOCK，O_CLOEXEC。分别返回一对可用于读写的文件描述符。(r, w)

可用性：一些Unix的味道。

3.3版本的新功能

os.posix_fallocate(*fd*, *offset*, *len*)

确保为由*fd*指定的文件分配足够的磁盘空间，并 从*offset*开始并继续*len*字节。

可用性：Unix。

3.3版本的新功能

os.posix_fadvise(*fd*, *offset*, *len*, *advice*)

宣布打算以特定模式访问数据，从而允许内核进行优化。该建议适用于由*fd*指定的文件的区域，从*偏移量*开始 并继续为*len*字节。 *建议*是一个POSIX_FADV_NORMAL，POSIX_FADV_SEQUENTIAL，POSIX_FADV_RANDOM，POSIX_FADV_NOREUSE，POSIX_FADV_WILLNEED或POSIX_FADV_DONTNEED。

可用性：Unix。

3.3版本的新功能

os.POSIX_FADV_NORMAL
os.POSIX_FADV_SEQUENTIAL
os.POSIX_FADV_RANDOM
os.POSIX_FADV_NOREUSE
os.POSIX_FADV_WILLNEED
os.POSIX_FADV_DONTNEED

可以在*建议*中使用的标志posix_fadvise()指定了可能使用的访问模式。

可用性：Unix。

3.3版本的新功能

os.pread（*fd*，*buffersize*，*offset* ）
从文件描述符*fd中*读取*偏移*位置。它将读取*缓冲区大小*的字节数。文件偏移保持不变。

可用性：Unix。

3.3版本的新功能

os.pwrite（*fd*，*str*，*offset* ）
将*bytestring*写入一个文件描述符*fd*，从*offset开始*，保持文件偏移量不变。

可用性：Unix。

3.3版本的新功能

os.read（*fd*，*n* ）
从文件描述符*fd中*读取至多*n*个字节。返回包含读取字节的字节串。如果已经达到由*fd*引用的文件的末尾，则返回空字节对象。

> **注意：** 此函数用于低级别I / O，并且必须将其应用于由os.open()或返回的文件描述符pipe()。读取由内置函数返回的“文件对象”，open()或者通过 popen()或fdopen() ，或sys.stdin使用其 read()或readline()方法来读取。

*版本3.5中更改：*如果系统调用中断且信号处理程序不引发异常，则该函数现在重试系统调用而不是引发 InterruptedError异常（请参阅**PEP 475**为理由）。

os.sendfile（*out*，*in*，*offset*，*count* ）
os.sendfile（*out*，*in*，*offset*，*count*，[*headers*，] [*trailers*，] *flags = 0* ）
将*计数*字节从文件描述符*中*复制到文件描述符中，从*偏移处*开始。返回发送的字节数。EOF到达时返回0。

所有定义的平台都支持第一个函数表示法 sendfile()。

在Linux上，如果*偏移*给出None的字节从当前位置读出*的*和位置*的*更新。

第二壳体可在Mac OS X和FreeBSD，其中可以使用*报头*和 *拖车*是被之前和之后的数据从写入缓冲器的任意序列*中*被写入。它返回与第一种情况相同的结果。

在Mac OS X和FreeBSD上，*计数*值为0 指定发送，直到达到*in*的结尾。

所有平台都支持插槽为*出*文件描述符，有些平台允许其他类型（如普通文件，管道）也是如此。

跨平台的应用程序不应该使用*标题*，*尾部*和*标志* 参数。

可用性：Unix。

注意： 对于更高级别的包装sendfile()，请参阅 socket.socket.sendfile()。

3.3版本的新功能

os.set_blocking(*fd* , *blocking*)
设置指定文件描述符的阻止模式。O_NONBLOCK如果阻塞是设置 标志False，否则清除标志。

另见get_blocking()和socket.socket.setblocking()。

可用性：Unix。

3.5版本中的新功能。

os.SF_NODISKIO
os.SF_MNOWAIT
os.SF_SYNC
sendfile()函数的参数，如果实现支持它们。

可用性：Unix。

3.3版本的新功能

os.readv(*fd* , *buffers*)
从一个文件描述符*fd*读入一些可变字节的对象 *缓冲区*。readv()会将数据传输到每个缓冲区，直到它满了，然后移动到序列中的下一个缓冲区以保存其余数据。readv()返回读取的总字节数（可能小于所有对象的总容量）。

可用性：Unix。

3.3版本的新功能

os.tcgetpgrp(*fd*)
返回与*fd*给出的终端关联的进程组（由返回的开放文件描述符os.open()）。

可用性：Unix。

os.tcsetpgrp(*fd* , *pg*)
将与由*fd*（打开的文件描述符返回的os.open()）给出的终端关联的进程组设置为*pg*。

可用性：Unix。

os.ttyname(*fd*)
返回一个字符串，指定与文件描述符*fd*关联的终端设备。如果*fd*未与终端设备关联，则会引发异常。

可用性：Unix。

os.write(*fd* , *str*)
将*str中*的bytestring写入文件描述符*fd*。返回实际写入的字节数。

注意： 此函数用于低级别I / O，并且必须将其应用于由os.open()或返回的文件描述符pipe()。要编写由内置函数返回一个“文件对象”open()或popen()或 fdopen()或sys.stdout或sys.stderr，使用它的 write()方法。

*版本3.5中更改：*如果系统调用中断且信号处理程序不引发异常，则该函数现在重试系统调用而不是引发 InterruptedError异常（请参阅**PEP 475**为理由）。

os.**writev**(*fd*, *buffers*)

将*缓冲区*的内容写入文件描述符*fd*。*缓冲区*必须是一个类似字节的对象序列。缓冲区按阵列顺序处理。第一个缓冲区的全部内容在进入第二个之前写入，依此类推。操作系统可以对可以使用的缓冲区数量设置限制（sysconf（）值SC_IOV_MAX）。

writev() 将每个对象的内容写入文件描述符并返回写入的总字节数。

可用性：Unix。

3.3版本的新功能

16.1.4.1。查询终端的大小

3.3版本的新功能

os.**get_terminal_size**(*fd=STDOUT_FILENO*)

返回终端窗口的大小，作为类型的元组。(columns, lines) terminal_size

可选参数fd（默认STDOUT_FILENO或标准输出）指定应查询哪个文件描述符。

如果文件描述符未连接到终端，OSError 则引发。

shutil.get_terminal_size()是通常应该使用的高级功能，os.get_terminal_size是低级的实现。

可用性：Unix，Windows。

*类*os.**terminal_size**

元组的子类，持有终端窗口大小。(columns, lines)

columns

终端窗口的宽度，以字符为单位。

lines

终端窗口的高度，以字符表示。

16.1.4.2。文件描述符的继承

3.4版新增功能

文件描述符具有“可继承”标志，该标志指示文件描述符是否可以被子进程继承。从Python 3.4开始，由Python创建的文件描述符在默认情况下是非可继承的。

在UNIX上，执行新程序时，子进程中的非可继承文件描述符被关闭，其他文件描述符被继承。

在Windows上，除了标准流（文件描述符0,1和2：stdin，stdout和stderr）外，子进程中的非可继承句柄和文件描述符都是关闭的，它们始终是继承的。使用spawn*函数，所有可继承的句柄和所有可继承的文件描述符都被继承。使用该subprocess模块，除标准流之外的所有文件描述符都将关闭，并且只有*close_fds*参数为可继承的句柄才会继承 False。

os.get_inheritable(*fd*)
获取指定文件描述符的“可继承”标志（布尔值）。

os.set_inheritable(*fd*, *可继承*)
设置指定文件描述符的“可继承”标志。

os.get_handle_inheritable(*句柄*)
获取指定句柄的“可继承”标志（布尔值）。

可用性：Windows。

os.set_handle_inheritable(*句柄*, *可继承*)
设置指定句柄的“可继承”标志。

可用性：Windows。

16.1.5。文件和目录

在一些Unix平台上，许多这些功能都支持以下一个或多个功能：

- **指定文件描述符：** 对于某些函数，*path*参数不仅可以是一个给出路径名的字符串，还可以是一个文件描述符。然后该函数将对描述符引用的文件进行操作。（对于POSIX系统，Python将调用f...该函数的 版本。）

  您可以使用检查*路径*是否可以在您的平台上指定为文件描述符os.supports_fd。如果不可用，则使用它将引发一次NotImplementedError。

  如果该函数还支持*dir_fd*或*follow_symlinks*参数，则在将*路径*作为文件描述符提供时指定其中之一是错误的。

- **相对于目录描述符的路径：**如果*dir_fd*不是None，它应该是引用目录的文件描述符，并且操作的路径应该是相对的; 路径将是相对于该目录。如果路径是绝对路径，则忽略*dir_fd*。（对于POSIX系统，Python会调用...at或f...at函数的版本。）

  您可以使用检查您的平台是否支持 *dir_fd*os.supports_dir_fd。如果不可用，则使用它将引发一次 NotImplementedError。

- **不遵循符号链接：**如果*follow_symlinks*是 False，并且要操作的路径的最后一个元素是符号链接，则该函数将在符号链接本身上运行，而不是链接指向的文件。（对于POSIX系

统，Python将调用1...该函数的版本。）

您可以使用检查您的平台是否支持*follow_symlinks*os.supports_follow_symlinks。如果不可用，则使用它将引发一次NotImplementedError。

os.access（*path*，*mode*，***，*dir_fd = None*，*effective_ids = False*，*follow_symlinks = True*）

使用真正的uid / gid来测试对*路径的*访问。请注意，大多数操作将使用有效的uid / gid，因此可以在suid / sgid环境中使用此例程来测试调用用户是否具有指定的*路径*访问权限。*模式*应该是F_OK来测试是否存在*路径*，或者它可以是包容性的OR的一种或多种的R_OK，W_OK和X_OK测试权限。True如果访问被允许，False则返回，如果不是。有关更多信息，请参阅Unix手册页*访问权限（2）*。

此函数可以支持指定相对于目录描述符的路径，而不支持符号链接。

如果*effective_ids*是True，access()将使用有效的uid / gid而不是真正的uid / gid执行访问检查。您的平台可能不支持*effective_ids*；你可以检查它是否可用os.supports_effective_ids。如果不可用，则使用它将引发一次NotImplementedError。

注意： 使用access()来检查用户是否有权例如在实际使用之前打开文件open()，这会造成安全漏洞，因为用户可能利用检查和打开文件之间的短时间间隔来操纵它。最好使用EAFP技术。例如：

```
if os.access("myfile", os.R_OK):
    with open("myfile") as fp:
        return fp.read()
return "some default data"
```

写得更好：

```
try:
    fp = open("myfile")
except PermissionError:
    return "some default data"
else:
    with fp:
        return fp.read()
```

注意： 即使access()指示它们会成功，I / O操作也可能失败，特别是对于可能具有超出通常POSIX许可位模型的权限语义的网络文件系统的操作。

*版本3.3中已更改：*添加了*dir_fd*，*effective_ids*和*follow_symlinks*参数。

*在版本3.6中更改：*接受类似路径的对象。

os.F_OK
os.R_OK
os.W_OK
os.X_OK

作为*模式*参数传递的值分别access()用于测试*路径*的存在性，可读性，可写性和可执行性。

os.chdir(*路径*)
将当前工作目录更改为*路径*。

该功能可以支持指定文件描述符。描述符必须引用打开的目录，而不是打开的文件。

*3.3版*新增功能：增加了对在某些平台上将*路径*指定为文件描述符的支持。

*在版本3.6中更改：*接受类似路径的对象。

os.chflags(*path*, *flags*, *, *follow_symlinks = True*)
将*路径*的标志设置为数字*标志*。*标志*可以采用以下值（按照stat模块中的定义）的组合（按位或）：

- stat.UF_NODUMP
- stat.UF_IMMUTABLE
- stat.UF_APPEND
- stat.UF_OPAQUE
- stat.UF_NOUNLINK
- stat.UF_COMPRESSED
- stat.UF_HIDDEN
- stat.SF_ARCHIVED
- stat.SF_IMMUTABLE
- stat.SF_APPEND
- stat.SF_NOUNLINK
- stat.SF_SNAPSHOT

该功能可以支持不遵循符号链接。

可用性：Unix。

*新的3.3版：*该*follow_symlinks*参数。

*在版本3.6中更改：*接受类似路径的对象。

os.chmod(*path*, *mode*, *, *dir_fd = None*, *follow_symlinks = True*)
将*路径*的模式更改为数字*模式*。*模式*可能会采用以下值之一（如stat模块中定义的）或它们的按位或运算组合：

- stat.S_ISUID
- stat.S_ISGID
- stat.S_ENFMT
- stat.S_ISVTX
- stat.S_IREAD
- stat.S_IWRITE
- stat.S_IEXEC
- stat.S_IRWXU
- stat.S_IRUSR
- stat.S_IWUSR
- stat.S_IXUSR

- stat.S_IRWXG
- stat.S_IRGRP
- stat.S_IWGRP
- stat.S_IXGRP
- stat.S_IRWXO
- stat.S_IROTH
- stat.S_IWOTH
- stat.S_IXOTH

这个函数可以支持指定一个文件描述符， 相对于目录描述符的路径，而不是跟随符号链接。

> **注意：** 虽然Windows支持chmod()，但只能用它设置文件的只读标志（通过stat.S_IWRITE和stat.S_IREAD 常量或相应的整数值）。所有其他位都被忽略。

*3.3版新增功能：*增加了对指定*路径*作为打开文件描述符的支持，以及*dir_fd*和*follow_symlinks*参数的支持。

*在版本3.6中更改：*接受类似路径的对象。

os.chown(*path*, *uid*, *gid*, *, *dir_fd = None*, *follow_symlinks = True*)
将*路径*的所有者和组标识更改为数字*uid*和*gid*。要使其中一个ID保持不变，请将其设置为-1。

这个函数可以支持指定一个文件描述符， 相对于目录描述符的路径，而不是跟随符号链接。

请参阅shutil.chown()除了数字ID以外的接受名称的更高级别的函数。

可用性：Unix。

*V3.3中的新增功能：*增加了对指定*路径*的打开文件描述符以及*dir_fd*和*follow_symlinks*参数的支持。

*在版本3.6中更改：*支持类似路径的对象。

os.chroot(*路径*)
将当前进程的根目录更改为*路径*。

可用性：Unix。

*在版本3.6中更改：*接受类似路径的对象。

os.fchdir(*fd*)
将当前工作目录更改为由文件描述符*fd*表示的目录。描述符必须引用打开的目录，而不是打开的文件。从Python 3.3开始，这相当于os.chdir(fd)。

可用性：Unix。

os.getcwd()
返回表示当前工作目录的字符串。

os.getcwdb()
返回表示当前工作目录的字符串。

os.lchflags(*路径*, *标志*)
设置的标志*路径*的数字*标志*，如chflags()，但不遵循符号链接。从Python 3.3开始，这相当于。os.chflags(path, flags, follow_symlinks=False)

可用性：Unix。

*在版本3.6中更改：*接受类似路径的对象。

os.lchmod(*路径*, *模式*)
将*路径*的模式更改为数字*模式*。如果path是符号链接，则会影响符号链接而不是目标。请参阅文档以chmod()了解可能的*模式*值。从Python 3.3开始，这相当于。os.chmod(path, mode, follow_symlinks=False)

可用性：Unix。

*在版本3.6中更改：*接受类似路径的对象。

os.lchown(*路径*, *uid*, *gid*)
将*路径*的所有者和组标识更改为数字*uid*和*gid*。此功能不会遵循符号链接。从Python 3.3开始，这相当于。os.chown(path, uid, gid, follow_symlinks=False)

可用性：Unix。

*在版本3.6中更改：*接受类似路径的对象。

os.link(*src*, *dst*, *, *src_dir_fd* = *None*, *dst_dir_fd* = *None*, *follow_symlinks* = *True*)
创建一个指向名为*dst*的*src*的硬链接。

此函数可以支持指定*src_dir_fd*和/或*dst_dir_fd*来提供与目录描述符相关的路径，而不是遵循符号链接。

可用性：Unix，Windows。

*在版本3.2中更改：*添加了Windows支持。

*3.3版*新增功能：添加了*src_dir_fd*，*dst_dir_fd*和*follow_symlinks*参数。

*改变在3.6版本：*接受一个路径状物体的*SRC*和*DST*。

os.listdir(*path* ='。')
返回包含*路径*给定目录中条目名称的列表。该列表以任意顺序排列，并且不包含特殊条目'.'，'..'即使它们存在于目录中。

*路径*可能是一个类似于路径的对象。如果*路径*是类型的bytes（直接或间接通过PathLike接口），则返回的文件名也是类型的bytes；在所有其他情况下，它们将是类型的str。

这个函数也可以支持指定一个文件描述符；文件描述符必须引用一个目录。

注意： 要将str文件名编码为bytes，请使用fsencode()。

也可以看看： 该scandir()函数返回目录条目以及文件属性信息，为许多常见用例提供更好的性能。

*在3.2版本中更改：*该*路径*参数成为可选的。

*3.3版*新增功能：增加了对指定*路径*的打开文件描述符的支持。

*在版本3.6中更改：*接受类似路径的对象。

os.**lstat**(*path*, *, *dir_fd = None*)

lstat()在给定的路径上执行等价的系统调用。类似于stat()，但不遵循符号链接。返回一个 stat_result对象。

在不支持符号链接的平台上，这是一个别名 stat()。

从Python 3.3开始，这相当于。os.stat(path, dir_fd=dir_fd, follow_symlinks=False)

该功能还可以支持与目录描述符相关的路径。

也可以看看： 该stat()功能。

*在版本3.2中进行了更改：*添加了对Windows 6.0（Vista）符号链接的支持。

*在版本3.3中更改：*添加了*dir_fd*参数。

*改变在3.6版本：*接受一个路径状物体的*SRC*和*DST*。

os.**mkdir**(*path*, *mode = 0o777*, *, *dir_fd = None*)

使用数字模式*模式*创建一个名为*path*的目录。

如果该目录已经存在，FileExistsError则引发。

在某些系统上，*模式*被忽略。在使用它的地方，当前的umask值首先被屏蔽掉。如果除了最后9位（即*模式*的八进制表示的最后3位数）被设置，它们的含义是平台相关的。在某些平台上，它们被忽略，你应该chmod()明确地调用 它们来设置它们。

该功能还可以支持与目录描述符相关的路径。

也可以创建临时目录; 看 tempfile模块的tempfile.mkdtemp()功能。

*版本3.3中的新增功能：*dir_fd参数。

*在版本3.6中更改：*接受类似路径的对象。

os.**makedirs**(*name*, *mode = 0o777*, *exist_ok = False*)

递归目录创建功能。像mkdir()，但是使所有需要包含叶目录的中级目录。

所述*模式*参数被传递给mkdir(); 请参阅mkdir（）说明以了解它的解释。

如果*exist_ok*是False（缺省值），OSError则在目标目录已经存在的情况下会产生一个。

> **注意：** makedirs()如果要创建的路径元素包含pardir（例如，在UNIX系统上的“..”），将会变得混乱。

该函数正确处理UNC路径。

*新版本3.2：*在*exist_ok*参数。

在3.4.1版本中更改： Python的3.4.1之前，如果*exist_ok*是True与目录的存在，makedirs()仍然会产生一个错误，如果*模式*不匹配的现有目录的模式。由于这种行为不可能安全地实现，所以它在Python 3.4.1中被删除。见bpo-21082。

*在版本3.6中更改：*接受类似路径的对象。

os.**mkfifo**(*path*, *mode = 0o666*, ***, *dir_fd = None*)

使用数字模式*模式*创建一个名为*path*的FIFO（一个命名管道）。首先将当前的umask值从模式中屏蔽掉。

该功能还可以支持与目录描述符相关的路径。

FIFO是可以像普通文件一样访问的管道。FIFO存在，直到它们被删除（例如os.unlink()）。通常，FIFO被用作“客户端”和“服务器”类型进程之间的交会点：服务器打开FIFO进行读取，客户端打开它进行写入。请注意，mkfifo()不会打开FIFO - 它只会创建会合点。

可用性：Unix。

*版本3.3中*的*新增功能：*dir_fd参数。

*在版本3.6中更改：*接受类似路径的对象。

os.**mknod**(*path*, *mode = 0o600*, *device = 0*, ***, *dir_fd = None*)

创建一个名为*path*的文件系统节点（文件，设备专用文件或命名管道）。*模式*同时指定要使用的权限以及要创建的节点的类型，被组合（按位OR）与一个stat.S_IFREG，stat.S_IFCHR，stat.S_IFBLK，和stat.S_IFIFO（这些常数是在可用的stat）。对于stat.S_IFCHR和stat.S_IFBLK，*设备*定义新创建的设备专用文件（可能使用os.makedev()），否则将被忽略。

该功能还可以支持与目录描述符相关的路径。

可用性：Unix。

*版本3.3中*的*新增功能：*dir_fd参数。

*在版本3.6中更改：*接受类似路径的对象。

os.**major**(*设备*)

从原始设备号码（通常是st_dev或st_rdev来自stat）中提取设备主号码。

os.**minor**(*设备*)

从原始设备号码（通常是st_dev或st_rdev来自stat）中提取设备次要号码。

os.makedev(*主要*, *次要*)
从主要和次要设备编号构成原始设备编号。

os.pathconf(*路径*, *名称*)
返回与指定文件相关的系统配置信息。*name* 指定要检索的配置值; 它可能是一个字符串，它是定义的系统值的名称; 这些名称在许多标准（POSIX.1，Unix 95，Unix 98等）中都有详细说明。一些平台也定义了其他名称。pathconf_names字典中给出了主机操作系统已知的名称。对于未包含在该映射中的配置变量，也会接受传递*名称*的整数。

如果*名称*是一个字符串并且未知，ValueError则会引发。如果主机系统不支持*名称*的特定值，即使它包含在内pathconf_names，OSError也会errno.EINVAL为错误号引发 一个值。

该功能可以支持指定文件描述符。

可用性：Unix。

*在版本3.6中更改：*接受类似路径的对象。

os.pathconf_names
字典映射名通过接受pathconf()和fpathconf()用于通过主机操作系统这些名称定义的整数值。这可以用来确定系统已知的一组名称。

可用性：Unix。

os.readlink(*path*, *, *dir_fd = None*)
返回表示符号链接指向的路径的字符串。结果可能是绝对路径名或相对路径名; 如果它是相对的，则可以使用它转换为绝对路径名。os.path.join(os.path.dirname(path), result)

如果*路径*是一个字符串对象（直接或间接通过 PathLike接口），结果也将是一个字符串对象，并且该调用可能会引发一个UnicodeDecodeError。如果*路径*是一个字节对象（直接或间接），则结果将是一个字节对象。

该功能还可以支持与目录描述符相关的路径。

可用性：Unix，Windows

*在版本3.2中进行了更改：*添加了对Windows 6.0（Vista）符号链接的支持。

*版本3.3中的新增功能：*dir_fd参数。

*在版本3.6中更改：*接受类似路径的对象。

os.remove(*path*, *, *dir_fd = None*)
删除（删除）文件*路径*。如果*路径*是目录，OSError则引发。使用rmdir()删除目录。

该功能可以支持与目录描述符相关的路径。

在Windows上，尝试删除正在使用的文件会导致引发异常; 在Unix上，目录条目被删除，但分配给该文件的存储不可用，直到原始文件不再被使用。

这个功能在语义上与unlink()。

*版本3.3中的新增功能：*dir_fd参数。

*在版本3.6中更改：*接受类似路径的对象。

os.**removedirs**(*名字*)

递归地删除目录。像rmdir()除了如果叶目录被成功删除之外的工作，removedirs()试图连续删除*路径中*提到的每个父目录，直到引发错误（忽略，因为它通常意味着父目录不是空的）。例如，os.removedirs('foo/bar/baz')将首先删除该目录'foo/bar/baz'，然后删除'foo/bar'，'foo'如果它们是空的。OSError如果叶子目录无法成功删除则引发。

*在版本3.6中更改：*接受类似路径的对象。

os.**rename**(*src*, *dst*, *, *src_dir_fd = None*, *dst_dir_fd = None*)

将文件或目录*src*重命名为*dst*。如果*dst*是一个目录，OSError将会被提出。在Unix上，如果*dst*存在并且是一个文件，那么如果用户有权限，它将被静默地替换。如果*src*和*dst*位于不同的文件系统上，该操作可能会在某些Unix版本上失败。如果成功，重命名将是一个原子操作（这是POSIX要求）。在Windows上，如果*dst*已经存在，OSError即使它是文件也会被提升。

该函数可以支持指定*src_dir_fd*和/或*dst_dir_fd*来提供与目录描述符相关的路径。

如果您想跨平台覆盖目标，请使用replace()。

*新的3.3版：*该*src_dir_fd*和*dst_dir_fd*参数。

*改变在3.6版本：*接受一个路径状物体的*SRC*和*DST*。

os.**renames**(*旧的*, *新的*)

递归目录或文件重命名功能。像rename()创建新路径名所需的任何中间目录都是首先尝试的那样工作。重命名后，将使用旧的名称的最右边路径段对应的目录removedirs()。

> **注意：** 如果您缺少删除叶子目录或文件所需的权限，则此功能可能会失败，并生成新的目录结构。

*改变在3.6版本：*接受一个路径状物体为*老*和*新*。

os.**replace**(*src*, *dst*, *, *src_dir_fd = None*, *dst_dir_fd = None*)

将文件或目录*src*重命名为*dst*。如果*dst*是一个目录，OSError将会被提出。如果*dst*存在并且是一个文件，如果用户有权限，它将被静静地替换。如果*src*和*dst*位于不同的文件系统上，操作可能会失败。如果成功，重命名将是一个原子操作（这是POSIX要求）。

该函数可以支持指定*src_dir_fd*和/或*dst_dir_fd*来提供与目录描述符相关的路径。

3.3版本的新功能

*改变在3.6版本：*接受一个路径状物体的*SRC*和*DST*。

os.**rmdir**(*path*, *, *dir_fd = None*)

删除（删除）目录*路径*。仅当目录为空时才起作用，否则OSError会引发。为了移除整个目录树，shutil.rmtree()可以使用。

该功能可以支持与目录描述符相关的路径。

*版本3.3中的新增功能：*dir_fd参数。

*在版本3.6中更改：*接受类似路径的对象。

os.scandir(*path='.'*)

返回os.DirEntry对应于*路径*给定目录中条目的对象的迭代器。这些条目产生以任意顺序，特殊项目'.'和'..'不包括在内。

如果操作系统在扫描目录时提供该信息，则使用scandir()而不是listdir()可以显着提高也需要文件类型或文件属性信息的代码的性能os.DirEntry。所有os.DirEntry方法可以执行系统调用，但 is_dir() 和 is_file() 通常只需要一个系统调用的符号链接；os.DirEntry.stat() 总是需要在Unix上进行系统调用，但只需要一个用于Windows上的符号链接。

*路径*可能是一个类似于路径的对象。如果*路径*是类型的bytes（直接或间接通过PathLike接口），则 每个类型name和path属性os.DirEntry将是bytes；在所有其他情况下，它们将是类型的str。

所述scandir()迭代器支持上下文管理协议，并具有以下的方法：

scandir.close()

关闭迭代器和自由获取的资源。

当迭代器耗尽或垃圾收集时，或迭代过程中发生错误时，这会自动调用。不过，最好明确地调用它或使用该with 语句。

3.6版本中的新功能。

以下示例显示了一个简单的用法，scandir()可以显示给定*路径*中不以其开头的 所有文件（不包括目录）'.'。该entry.is_file()呼叫通常不会进行额外的系统调用：

```
with os.scandir(path) as it:
    for entry in it:
        if not entry.name.startswith('.') and entry.is_file():
            print(entry.name)
```

注意： 在基于Unix的系统上，scandir()使用系统的 opendir() 和 readdir() 函数。在Windows上，它使用Win32 FindFirstFileW 和 FindNextFileW 函数。

3.5版本中的新功能。

*3.6版*新增功能：增加了对上下文管理器协议和 close() 方法的支持。如果一个scandir()迭代器既没有被用尽也没有被显式关闭ResourceWarning，它的析构函数将被发射。

该函数接受一个类似于路径的对象。

*类*os.DirEntry

通过产生的对象scandir()来公开目录条目的文件路径和其他文件属性。

scandir()将尽可能多地提供这些信息，而无需进行额外的系统调用。当进行stat()或lstat()系统调用时，os.DirEntry对象将缓存结果。

os.DirEntry实例不打算存储在长寿命的数据结构中;如果您知道文件元数据已更改，或者自调用后经过很长时间scandir()，请致电os.stat(entry.path)以获取最新信息。

由于这些os.DirEntry方法可以进行操作系统调用，它们也可能会引发OSError。如果您需要对错误进行非常细致的控制，则可以OSError在调用其中一个os.DirEntry方法时进行捕捉，并根据需要进行处理。

要直接用作路径类对象，需要os.DirEntry 实现PathLike接口。

os.DirEntry实例的属性和方法如下所示：

name

条目的基本文件名，相对于*路径*参数。scandir()

该name属性将是bytes如果*路径*参数是类型的，否则。使用 解码字节的文件名。scandir() bytes str fsdecode()

path

条目的完整路径名：相当于其中*scandir_path*是*路径*参数。如果*路径*参数是绝对路径，则路径是绝对*路径*。os.path.join(scandir_path, entry.name)scandir()scandir()

该path属性将是bytes如果*路径*参数是类型的，否则。使用 解码字节的文件名。scandir() bytes str fsdecode()

inode()

返回条目的inode号码。

结果缓存在os.DirEntry对象上。使用 要赶上最新信息。os.stat(entry.path, follow_symlinks=False).st_ino

在第一个未缓存的调用中，系统调用在Windows上需要，而在Unix上不需要。

is_dir(*, *follow_symlinks = True*)

True如果此条目是指向目录的目录或符号链接，则返回;返回False如果条目或指向任何其他类型的文件，或者如果它不存在了。

如果*follow_symlinks*是False，则True仅在该条目是目录时才返回（不符合链接）;返回False如果条目是任何其他类型的文件，或者如果它不存在了。

结果缓存在os.DirEntry对象上，并有一个单独的缓存用于*follow_symlinks* True和False。请随身os.stat()携带stat.S_ISDIR()以获取最新信息。

在第一个未缓存的呼叫中，大多数情况下不需要系统调用。特别是，对于非符号链接，Windows或Unix都不需要系统调用，除非在某些Unix文件系统（如网络文件系

统）上返回。如果该条目是一个符号，一个系统调用将被要求遵循符号链接，除非*follow_symlinks*是。dirent.d_type == DT_UNKNOWNFalse

这种方法可以提高OSError ， PermissionError但是FileNotFoundError被捕获而不被提出。

is_file (*, *follow_symlinks = True*)

True如果此条目是指向文件的文件或符号链接，则返回; 返回False如果条目或指向一个目录或其他非文件条目，或者如果它不存在了。

如果*follow_symlinks*是False ， True只有当这个条目是一个文件（没有符号链接）时才返回; 返回False如果条目是一个目录或其他非文件条目，或者已经不存在了。

结果缓存在os.DirEntry对象上。缓存，系统调用以及引发的异常均按照is_dir()。

is_symlink ( )

返回True如果此项是一个符号链接（即使破碎）; 返回False如果条目指向一个目录或任何类型的文件，或者如果它不存在了。

结果缓存在os.DirEntry对象上。致电 os.path.islink()以获取最新信息。

在第一个未缓存的呼叫中，大多数情况下不需要系统调用。具体而言，Windows或Unix都不需要系统调用，除了某些Unix文件系统（如网络文件系统）返回的情况 。dirent.d_type == DT_UNKNOWN

这种方法可以提高OSError ， PermissionError但是FileNotFoundError被捕获而不被提出。

stat (*, *follow_symlinks = True*)

stat_result为此条目返回一个对象。该方法默认遵循符号链接; 统计符号链接添加follow_symlinks=False参数。

在Unix上，这个方法总是需要一个系统调用。在Windows上，如果*follow_symlinks*是True且该条目是符号链接，则只需要系统调用。

在Windows中， st_ino ， st_dev和st_nlink属性 stat_result总是设置为零。调用os.stat()以获取这些属性。

结果缓存在os.DirEntry对象上，并有一个单独的缓存用于*follow_symlinks* True和False。致电os.stat()以获取最新信息。

请注意，有几个属性和方法之间的一个很好的对应os.DirEntry和pathlib.Path。具体而言，该name属性与“ is_dir() ， is_file()” is_symlink() 和“ stat()方法”具有相同的含义。

3.5版本中的新功能。

*在版本3.6中更改：*增加了对PathLike接口的支持。增加了bytes对Windows上路径的支持。

os. stat (*path* , *, *dir_fd = None* , *follow_symlinks = True*)

获取文件或文件描述符的状态。stat()在给定的路径上执行等价的 系统调用。*路径*可以被指定为字符串或字节 - 直接或间接通过PathLike 接口 - 或作为打开的文件描述符。返回一个stat_result 对象。

这个函数通常遵循符号链接; 统计符号链接添加参数 follow_symlinks=False ，或使用lstat()。

该函数可以支持指定文件描述符， 而不支持符号链接。

例：

```
>>> import os
>>> statinfo = os.stat('somefile.txt')
>>> statinfo
os.stat_result(st_mode=33188, st_ino=7876932, st_dev=234881026,
st_nlink=1, st_uid=501, st_gid=501, st_size=264, st_atime=1297230295,
st_mtime=1297230027, st_ctime=1297230027)
>>> statinfo.st_size
264
```

也可以看看： fstat()和lstat()功能。

*3.3版*新增功能：添加了*dir_fd*和*follow_symlinks*参数，指定了文件描述符而不是路径。

*在版本3.6中更改：*接受类似路径的对象。

*类*os.stat_result

对象的属性大致对应于stat结构的成员 。它用于os.stat()， os.fstat()和os.lstat()。的结果。

属性：

st_mode
: 文件模式：文件类型和文件模式位（权限）。

st_ino
: Inode号码。

st_dev
: 该文件所在设备的标识符。

st_nlink
: 硬链接的数量。

st_uid
: 文件所有者的用户标识符。

st_gid
: 文件所有者的组标识符。

st_size

文件大小（以字节为单位），如果是常规文件或符号链接。符号链接的大小是它所包含的路径名的长度，而不是终止的空字节。

时间戳：

`st_atime`
最近访问的时间以秒表示。

`st_mtime`
最近的内容修改时间以秒表示。

`st_ctime`
取决于平台：

- Unix上最新的元数据更改时间，
- 在Windows上创建的时间，以秒表示。

`st_atime_ns`
最近访问的时间以纳秒表示，以整数表示。

`st_mtime_ns`
最新内容修改的时间以纳秒表示，以整数表示。

`st_ctime_ns`
取决于平台：

- Unix上最新的元数据更改时间，
- 在Windows上创建的时间，以纳秒为单位表示为整数。

另请参阅该`stat_float_times()`功能。

注意： 确切含义和分辨率`st_atime`， `st_mtime`以及`st_ctime`属性取决于操作系统和文件系统上。例如，在使用FAT或FAT32文件系统的Windows系统上，`st_mtime`具有2秒的分辨率，并且 `st_atime`仅具有1天的分辨率。详细信息请参阅您的操作系统文档。
同样，虽然`st_atime_ns`，`st_mtime_ns`和`st_ctime_ns`在纳秒总是表示，许多系统不提供纳秒的精度。在那些提供纳秒的精度系统，浮点对象用来存储`st_atime`，`st_mtime`以及`st_ctime` 无法保留这一切，因此会稍微不精确。如果您需要确切的时间戳你应该总是使用 `st_atime_ns`，`st_mtime_ns`和`st_ctime_ns`。

在某些Unix系统（如Linux）上，以下属性也可能可用：

`st_blocks`
为文件分配的512字节块的数量。`st_size`当文件有空洞时，这可能小于/ 512。

`st_blksize`
对于高效的文件系统I / O，“首选”块大小。以较小的块写入文件可能会导致低效的读取 - 修改 - 重写。

`st_rdev`

设备的类型，如果是inode设备。

st_flags
用户定义的文件标志。

在其他Unix系统上（例如FreeBSD），下列属性可能是可用的（但只有在root尝试使用时才可以填写）：

st_gen
文件生成号码。

st_birthtime
文件创建时间。

在Mac OS系统上，以下属性也可能可用：

st_rsize
文件的实际大小。

st_creator
文件的创建者。

st_type
文件类型。

在Windows系统上，以下属性也可用：

st_file_attributes
Windows文件属性：返回dwFileAttributes的BY_HANDLE_FILE_INFORMATION结构的 成员GetFileInformationByHandle()。查看模块中的FILE_ATTRIBUTE_* 常量stat。

标准模块stat定义了用于从stat结构中提取信息的函数和常量。（在Windows上，有些项目填充虚拟值。）

为了向后兼容，一个stat_result实例是也可作为至少10点的整数给出的最重要的（便携式）成员的元组可访问的stat结构中，在顺序 st_mode，st_ino，st_dev，st_nlink，st_uid，st_gid，st_size，st_atime， st_mtime，st_ctime。一些实现可能会在最后添加更多项目。为了与较老的Python版本兼容，stat_result作为元组访问总是返回整数。

*在3.3版本的新功能：*增加了st_atime_ns，st_mtime_ns和 st_ctime_ns会员。

*3.5版*新增功能：st_file_attributes在Windows上添加了成员。

os.stat_float_times([*newvalue*])
确定是否stat_result将时间戳记表示为浮动对象。如果*newvalue*是True，则将来的调用stat()返回float，如果是 False，则将来的调用返回int。如果省略了*newvalue*，则返回当前设置。

为了与较老的Python版本兼容，stat_result作为元组访问总是返回整数。

Python现在默认返回浮点值。无法正确使用浮点时间戳的应用程序可以使用此功能来恢复旧的行为。

时间戳的分辨率（即可能的最小分数）取决于系统。有些系统仅支持第二种解决方案 在这些系统上，分数将始终为零。

建议此设置仅在程序启动时在*__main__*模块中更改; 图书馆不应该改变这个设置。如果应用程序使用工作不正确的库（如果处理了浮点时间戳记），则此应用程序应关闭此功能，直到库更正为止。

自3.3版以来已弃用。

os.**statvfs**(*路径*)

statvfs()在给定路径上执行系统调用。返回值是一个对象，其属性描述了给定路径上的文件系统，并对应于成员statvfs结构，即： f_bsize，f_frsize，f_blocks，f_bfree，f_bavail，f_files，f_ffree，f_favail， f_flag，f_namemax。

为f_flag属性的位标记定义了两个模块级常量：如果ST_RDONLY设置了，则文件系统将以只读方式挂载，如果ST_NOSUID设置，setuid / setgid位的语义将被禁用或不支持。

额外的模块级常量是为基于GNU / glibc的系统定义的。这些都是ST_NODEV（禁止访问设备特殊文件）， ST_NOEXEC（禁止执行程序）， ST_SYNCHRONOUS （写在一次同步），ST_MANDLOCK（允许在一个FS强制锁），ST_WRITE（在文件/目录/符号链接写），ST_APPEND （追加专用文件），ST_IMMUTABLE（不可变文件），ST_NOATIME （不更新访问时间），ST_NODIRATIME（不更新目录访问时间），ST_RELATIME（更新相对于mtime / ctime的时间）。

该功能可以支持指定文件描述符。

可用性：Unix。

*在3.2版本中更改：*将ST_RDONLY和ST_NOSUID添加常数。

*3.3版*新增功能*：*增加了对指定*路径*的打开文件描述符的支持。

*改变在3.4版本：*该ST_NODEV，ST_NOEXEC，ST_SYNCHRONOUS， ST_MANDLOCK，ST_WRITE，ST_APPEND， ST_IMMUTABLE，ST_NOATIME，ST_NODIRATIME，和ST_RELATIME添加常数。

*在版本3.6中更改：*接受类似路径的对象。

os.**supports_dir_fd**

甲Set对象指示在其功能 os模块允许使用其的*dir_fd*参数。不同的平台提供不同的功能，并且可能在另一个平台上工作的选项可能不受支持。出于一致性的考虑，支持*dir_fd的*函数总是允许指定参数，但如果功能实际上不可用，则会引发异常。

要检查某个特定函数是否允许使用其*dir_fd* 参数，请使用该in运算符supports_dir_fd。作为一个例子，这个表达式决定了*dir_fd*参数是否os.stat() 是本地可用的：

```
os.stat in os.supports_dir_fd
```

目前*dir_fd*参数只适用于Unix平台; 他们都没有在Windows上工作。

3.3版本的新功能

os.supports_effective_ids

甲Set对象指示在其功能 os模块允许使用的*effective_ids*为参数 os.access()。如果本地平台支持它，则集合将包含os.access()，否则它将为空。

要检查您是否可以使用*effective_ids*参数 os.access()，请使用如下in操作符supports_effective_ids：

```
os.access in os.supports_effective_ids
```

目前*effective_ids*仅适用于Unix平台; 它不适用于Windows。

3.3版本的新功能

os.supports_fd

甲Set指示对象，其功能在 os模块许可证指定它们的*路径*参数作为打开文件描述符。不同的平台提供不同的功能，并且可能在另一个平台上工作的选项可能不受支持。为了保持一致性，支持*fd的*函数总是允许指定参数，但如果功能实际上不可用，则会引发异常。

要检查一个特定的函数是否允许为其*路径*参数指定一个打开的文件描述符，请使用该in运算符 supports_fd。作为一个例子，这个表达式决定了os.chdir()在本地平台上调用时是否接受打开的文件描述符：

```
os.chdir in os.supports_fd
```

3.3版本的新功能

os.supports_follow_symlinks

甲Set对象指示在其功能 os模块允许使用其的*follow_symlinks*参数。不同的平台提供不同的功能，并且可能在另一个平台上工作的选项可能不受支持。为了保持一致性，支持*follow_symlink的*函数总是允许指定参数，但如果功能实际上不可用，则会引发异常。

要检查某个特定函数是否允许使用其*follow_symlinks*参数，请使用该in运算符supports_follow_symlinks。作为一个例子，这个表达式决定了*follow_symlinks*参数是否os.stat()是本地可用的：

```
os.stat in os.supports_follow_symlinks
```

3.3版本的新功能

os.symlink(*src*, *dst*, *target_is_directory = False*, ***, *dir_fd = None*)

创建一个指向名为*dst的*src的符号链接。

在Windows上，符号链接表示文件或目录，并且不会动态变形为目标。如果目标存在，则会创建符合链接的类型以匹配。否则，如果*target_is_directory*是True或文件符号链接（默认），则符号链接将被创建为目录。在非Window平台上，*target_is_directory*被忽略。

在Windows 6.0（Vista）中引入了符号链接支持。 symlink()将NotImplementedError在6.0以前的Windows版本中引发。

该功能可以支持与目录描述符相关的路径。

> **注意：** 在Windows上，为了成功创建符号链接，需要*SeCreateSymbolicLinkPrivilege*。此权限通常不授予常规用户，但可用于可将权限提升到管理员级别的帐户。获得权限或以管理员身份运行应用程序是成功创建符号链接的方法。
> `OSError` 当函数被非特权用户调用时引发。

可用性：Unix，Windows。

*在版本3.2中进行了更改：*添加了对Windows 6.0（Vista）符号链接的支持。

*3.3版新增功能：*添加了*dir_fd*参数，现在允许 在非Windows平台上使用*target_is_directory*。

*改变在3.6版本：*接受一个路径状物体的*SRC*和*DST*。

os.**sync**（）
强制将所有内容写入磁盘。

可用性：Unix。

3.3版本的新功能

os.**truncate**（*路径*，*长度*）
截断对应于*路径*的文件，以使其*长度*最多为 *长度*字节。

该功能可以支持指定文件描述符。

可用性：Unix，Windows。

3.3版本的新功能

*版本3.5中更改：*增加了对Windows的支持

*在版本3.6中更改：*接受类似路径的对象。

os.**unlink**（*path*，*，*dir_fd = None* ）
删除（删除）文件*路径*。这个功能在语义上是相同的`remove()`；该`unlink`名称是其传统的Unix名称。请参阅文档以 `remove()`获取更多信息。

*版本3.3中的新增功能：*dir_fd参数。

*在版本3.6中更改：*接受类似路径的对象。

os.**utime**（*path*，*times = None*，*，[*ns*，] *dir_fd = None*，*follow_symlinks = True* ）
设置*路径*指定文件的访问和修改时间。

`utime()`需要两个可选参数，*时间*和*ns*。这些指定了在*路径上*设置的时间，并且使用如下：

- 如果指定了*ns*，则它必须是表单的2元组， 其中每个成员都是表示纳秒的int。
  `(atime_ns, mtime_ns)`

- 如果*时间*不是None，它必须是表单的2元组， 其中每个成员都是表示秒的int或float。(atime, mtime)
- 如果*时间*是None和*NS*是不确定的，这等同于指定 其中两个时间是当前时间。 ns=(atime_ns, mtime_ns)

对于*时间*和*ns*都指定元组是错误的。

是否为*路径*指定目录 取决于操作系统是否将目录实现为文件（例如Windows不）。请注意，此处设置的确切时间可能不会由后续stat()调用返回，具体取决于操作系统记录访问和修改时间的分辨率; 见stat()。保留精确时间的最佳方法是将 结果对象中的*st_atime_ns*和*st_mtime_ns*字段os.stat()与*ns*参数一起用于*utime*。

这个函数可以支持指定一个文件描述符， 相对于目录描述符的路径，而不是跟随符号链接。

*V3.3中新增：*支持为*路径*指定打开的文件描述符，以及*dir_fd*，*follow_symlinks*和*ns*参数。

*在版本3.6中更改：*接受类似路径的对象。

os.**walk**(*top*, *topdown = True*, *onerror = None*, *followlinks = False*)

通过自顶向下或自底向上走树来生成目录树中的文件名。对于根目录*顶部*（包括*顶部*本身）树中的每个目录 ，它会生成一个3元组。(dirpath, dirnames, filenames)

*dirpath*是一个字符串，即目录的路径。 *dirnames中*是子目录的名称列表中的*dirpath*（不包括'.'和'..'）。 *文件名*是*dirpath*中非目录文件名称的列表。请注意，列表中的名称不包含路径组件。要获得完整路径（从*顶部*开始）到*dirpath中*的文件或目录，请执行 。os.path.join(dirpath, name)

如果可选参数*topdown*被True指定或未指定，则目录的三元组在其任何子目录（*自上而下*生成目录）的三元组之前生成。如果*自上而下的*是False，其所有子目录的三元组（目录生成由下而上）后三的目录中生成。无论*topdown*的值如何，子目录的列表都会在目录及其子目录的元组生成之前检索。

当*自上而下的*是True，来电者可以修改*dirnames中*就地列表（可能使用del或切片分配），并且walk()只会递归到他们的名字留在子目录*dirnames中*; 这可以用来修剪搜索，强制访问的特定顺序，甚至可以通知 walk()调用者在walk()再次恢复之前创建或重命名的目录 。修改*dirnames中*，当*自上而下的*就是False对行走的行为没有影响，因为在自下而上的模式在目录*dirnames中*被之前生成*dirpath*生成本身。

默认情况下，来自scandir()调用的错误将被忽略。如果指定了可选参数*onerror*，它应该是一个函数; 它将被调用一个参数，一个OSError实例。它可以报告错误以继续步行，或者提出异常以中止步行。请注意，文件名可用作filename异常对象的属性。

默认情况下，walk()不会走向解析为目录的符号链接。设置*followlinks*来True访问目录指向符号链接，在支持它们的系统。

> **注意：** 请注意，设置*followlinks*到True可能导致无限递归如果一个链接指向其自身的父目录。walk() 不跟踪它已经访问过的目录。

注意： 如果你传递一个相对路径名，不要改变当前工作目录的恢复walk()。 walk()从不改变当前目录，并假定它的调用者也不会。

此示例显示非目录文件在起始目录下的每个目录中占用的字节数，但不显示在任何CVS子目录下：

```
import os
from os.path import join, getsize
for root, dirs, files in os.walk('python/Lib/email'):
    print(root, "consumes", end=" ")
    print(sum(getsize(join(root, name)) for name in files), end=" ")
    print("bytes in", len(files), "non-directory files")
    if 'CVS' in dirs:
        dirs.remove('CVS')  # don't visit CVS directories
```

在下一个示例（简单实现shutil.rmtree()）中，从下往上走树是至关重要的，rmdir()不允许在目录为空之前删除目录：

```
# Delete everything reachable from the directory named in "top",
# assuming there are no symbolic links.
# CAUTION:  This is dangerous!  For example, if top == '/', it
# could delete all your disk files.
import os
for root, dirs, files in os.walk(top, topdown=False):
    for name in files:
        os.remove(os.path.join(root, name))
    for name in dirs:
        os.rmdir(os.path.join(root, name))
```

*在版本3.5中更改：*此函数现在调用，os.scandir()而不是os.listdir()通过减少调用的数量来加快速度os.stat()。

*在版本3.6中更改：*接受类似路径的对象。

os.**fwalk**(*top='.'*, *topdown=True*, *onerror=None*, ***, *follow_symlinks=False*, *dir_fd=None*)

它的行为完全一样walk()，只是它产生了一个4元组，并且它支持。(dirpath, dirnames, filenames, dirfd) dir_fd

dirpath，*dirnames*和*文件名*与walk()输出相同，*dirfd*是引用目录*dirpath*的*文件描述符*。

此函数始终支持与目录描述符相关的路径，而不支持符号链接。然而需要注意的是，不同于其他功能，则fwalk()默认值 *follow_symlinks*是False。

注意： 由于fwalk()yield文件描述符，这些描述符只有在下一个迭代步骤之前才有效，因此dup()如果要延长它们的长度，则应该复制它们（例如，with）。

此示例显示非目录文件在起始目录下的每个目录中占用的字节数，但不显示在任何CVS子目录下：

```
import os
for root, dirs, files, rootfd in os.fwalk('python/Lib/email'):
    print(root, "consumes", end="")
    print(sum([os.stat(name, dir_fd=rootfd).st_size for name in files]),
          end="")
    print("bytes in", len(files), "non-directory files")
    if 'CVS' in dirs:
        dirs.remove('CVS')  # don't visit CVS directories
```

在下一个示例中，从下往上走树是非常重要的： rmdir() 不允许在目录为空之前删除目录：

```
# Delete everything reachable from the directory named in "top",
# assuming there are no symbolic links.
# CAUTION:  This is dangerous!  For example, if top == '/', it
# could delete all your disk files.
import os
for root, dirs, files, rootfd in os.fwalk(top, topdown=False):
    for name in files:
        os.unlink(name, dir_fd=rootfd)
    for name in dirs:
        os.rmdir(name, dir_fd=rootfd)
```

可用性：Unix。

3.3版本的新功能

*在版本3.6中更改：*接受类似路径的对象。

16.1.5.1。Linux扩展属性

3.3版本的新功能

这些功能仅在Linux上可用。

os. **getxattr** (*path* , *attribute* , * , *follow_symlinks = True*)

返回扩展文件系统属性值*属性*的 *路径*。*属性*可以是字节或str（直接或间接通过 PathLike接口）。如果它是str，则用文件系统编码进行编码。

该函数可以支持指定文件描述符， 而不支持符号链接。

*改变在3.6版本：*接受一个路径状物体的*路径*和*属性*。

os. **listxattr** (*path = None* , * , *follow_symlinks = True*)

返回*路径*上扩展文件系统属性的列表。列表中的属性表示为用文件系统编码解码的字符串。如果*路径*是None， listxattr() 将检查当前目录。

该函数可以支持指定文件描述符， 而不支持符号链接。

*在版本3.6中更改：*接受类似路径的对象。

os. **removexattr** (*path* , *attribute* , * , *follow_symlinks = True*)

从*路径中*删除扩展的文件系统属性*属性*。 *属性*应该是字节或str（直接或间接通过 接口）。如果它是一个字符串，则用文件系统编码进行编码。PathLike

该函数可以支持指定文件描述符， 而不支持符号链接。

*改变在3.6版本：*接受一个路径状物体的*路径*和*属性*。

os.setxattr（*path*，*attribute*，*value*，*flags = 0*，*，*follow_symlinks = True*）

将*路径*上的扩展文件系统属性*属性*设置为*值*。 *属性*必须是没有嵌入式NUL的字节或字符串（直接或间接通过接口）。如果它是一个str，它将用文件系统编码进行编码。 *标志*可能是或。如果给出并且该属性不存在，则会被提出。如果给出并且该属性已经存在，则该属性将不会被创建并且将被引发。PathLikeXATTR_REPLACEXATTR_CREATEXATTR_REPLACEEEXISTSXATTR_CREATEENODATA

该函数可以支持指定文件描述符， 而不支持符号链接。

> **注意：** Linux内核版本低于2.6.39的错误导致在某些文件系统上flags参数被忽略。

*改变在3.6版本：*接受一个路径状物体的*路径*和*属性*。

os.XATTR_SIZE_MAX

扩展属性值的最大大小可以是。目前，这是Linux上的64 KiB。

os.XATTR_CREATE

这是flags参数中可能的值setxattr()。它表示操作必须创建一个属性。

os.XATTR_REPLACE

这是flags参数中可能的值setxattr()。它表示操作必须替换现有的属性。

16.1.6。流程管理

这些功能可用于创建和管理进程。

各种exec*函数为加载到进程中的新程序提供参数列表。在每种情况下，这些参数中的第一个将作为自己的名称传递给新程序，而不是作为用户可能在命令行上输入的参数。对于C程序员来说，这是argv[0]传递给程序的main()。例如，只能在标准输出上打印; 似乎会被忽略。os.execv('/bin/echo', ['foo', 'bar'])barfoo

os.abort（）

SIGABRT为当前进程生成一个信号。在Unix上，默认行为是产生核心转储; 在Windows上，该进程立即返回一个退出代码3。请注意，调用此函数将不要求注册的Python的信号处理SIGABRT用signal.signal()。

os.execl（*path*，*arg0*，*arg1*，*...*）

os.execle（*path*，*arg0*，*arg1*，*...*，*env*）

os.execlp（*file*，*arg0*，*arg1*，*...*）

os.execlpe（*file*，*arg0*，*arg1*，*...*，*env*）

os.execv(*path*, *args*)

os.execve(*path*, *args*, *env*)

os.execvp(*文件*, *参数*)

os.execvpe(*file*, *args*, *env*)

这些功能都执行一个新程序，取代当前进程; 他们不回来。在Unix上，新的可执行文件被加载到当前进程中，并且具有与调用者相同的进程ID。错误将作为OSError例外报告 。

当前进程立即被替换。打开文件对象和描述符不会刷新，因此如果在这些打开的文件上可能存在缓冲数据，则应该在调用函数之前sys.stdout.flush()或os.fsync()之前 使用它们进行刷新 exec*。

exec*函数的“l”和“v”变体在传递命令行参数方面有所不同。如果在编写代码时参数的数量是固定的，则“l”变体可能是最容易使用的变体; 各个参数只是成为execl*() 功能的附加参数。当参数的数量是可变的时，“v”变体是好的，参数以列表或元组作为*args* 参数传递。在任何一种情况下，子进程的参数都应以正在运行的命令的名称开始，但这不是强制执行的。

其中包括一个“P”附近的端部（所述变体execlp()， execlpe()，execvp()，和execvpe()）将使用 PATH环境变量来定位程序*文件*。当环境正在被替换时（使用其中一个exec*e变体，在下一段中讨论），新的环境被用作PATH变量。其他变体，execl()，execle()，execv()，和execve()，不会使用PATH变量来定位可执行文件; *路径*必须包含适当的绝对或相对路径。

为execle()，execlpe()，execve()，和execvpe()（注意，在“e”的这些所有端），则*ENV*参数必须被用于定义新处理的环境变量（这些被用来代替当前进程的环境）的映射; 功能execl()， execlp()，execv()，以及execvp()所有会导致新进程继承当前进程的环境。

对于execve()某些平台，*路径*也可以被指定为打开的文件描述符。您的平台可能不支持此功能; 你可以检查它是否可用os.supports_fd。如果不可用，则使用它将引发一次NotImplementedError。

可用性：Unix，Windows。

*3.3版*新增功能*：*增加了对指定*路径* for 的打开文件描述符的支持execve()。

*在版本3.6中更改：*接受类似路径的对象。

os._exit(*n*)

退出状态为*n*的进程，无需调用清理处理程序，刷新stdio缓冲区等。

> **注意：** 标准的退出方式是sys.exit(n)。 _exit()通常应该只在儿童过程之后才能使用fork()。

以下退出代码已定义并可用于_exit()，但它们不是必需的。这些通常用于以Python编写的系统程序，例如邮件服务器的外部命令传递程序。

> **注意：** 其中一些可能不适用于所有的Unix平台，因为有一些变化。这些常量被定义在底层平台所定义的位置。

os.EX_OK
退出代码意味着没有发生错误。

可用性：Unix。

os.EX_USAGE
退出代码意味着命令被错误地使用，例如给出错误的参数数量。

可用性：Unix。

os.EX_DATAERR
退出代码意味着输入数据不正确。

可用性：Unix。

os.EX_NOINPUT
退出代码意味着输入文件不存在或不可读。

可用性：Unix。

os.EX_NOUSER
退出代码，意味着指定的用户不存在。

可用性：Unix。

os.EX_NOHOST
退出代码，意味着指定的主机不存在。

可用性：Unix。

os.EX_UNAVAILABLE
退出代码，这意味着所需的服务不可用。

可用性：Unix。

os.EX_SOFTWARE
退出代码意味着检测到内部软件错误。

可用性：Unix。

os.EX_OSERR
退出代码意味着检测到操作系统错误，例如无法分叉或创建管道。

可用性：Unix。

os.EX_OSFILE
退出代码意味着某些系统文件不存在，无法打开或出现其他类型的错误。

可用性：Unix。

os.EX_CANTCREAT

退出代码意味着无法创建用户指定的输出文件。

可用性：Unix。

os.EX_IOERR
退出代码意味着在某个文件上执行I / O时发生错误。

可用性：Unix。

os.EX_TEMPFAIL
退出代码意味着发生了临时故障。这表示可能不是真正的错误，例如在重试操作期间无法建立的网络连接。

可用性：Unix。

os.EX_PROTOCOL
退出代码意味着协议交换是非法的，无效的或不被理解的。

可用性：Unix。

os.EX_NOPERM
退出代码意味着没有足够的权限执行操作（但不适用于文件系统问题）。

可用性：Unix。

os.EX_CONFIG
退出代码意味着发生某种配置错误。

可用性：Unix。

os.EX_NOTFOUND
退出代码意味着类似“未找到条目”。

可用性：Unix。

os.fork()
叉一个孩子的过程。返回0孩子和父母的孩子的进程ID。如果发生错误OSError。

请注意，从线程使用fork()时，包括FreeBSD <= 6.3和Cygwin在内的一些平台已知问题。

警告： 请参阅ssl使用fork()使用SSL模块的应用程序。

可用性：Unix。

os.forkpty()
用一个新的伪终端作为孩子的控制终端，把一个子进程分叉开来。返回一对，其中*PID*是在孩子，家长新的子进程ID，并且*FD*是伪终端的主端的文件描述符。对于更便携的方法，请使用该 模块。如果发生错误。(pid, fd)OptyOSError

可用性：一些Unix的味道。

os.kill（*pid*，*sig*）
发送信号*sig*到进程*pid*。主机平台上可用的特定信号的常量在signal模块中定义。

Windows：signal.CTRL_C_EVENT和 signal.CTRL_BREAK_EVENT信号是特殊的信号，只能发送到共享公共控制台窗口的控制台进程，例如某些子进程。*sig的*任何其他值都将导致进程被TerminateProcess API无条件地终止，并且退出代码将被设置为*sig*。 Windows版本kill()还需要处理句柄才能被杀死。

另见signal.pthread_kill()。

3.2版新增功能： Windows支持。

os.killpg（*pgid*，*sig*）
将信号*sig*发送到进程组*pgid*。

可用性：Unix。

os.nice（*增量*）
为流程的“好”添加*增量*。回报新的美好。

可用性：Unix。

os.plock（*op*）
将程序段锁定到内存中。*op*（定义于<sys/lock.h>）的值确定哪些段被锁定。

可用性：Unix。

os.popen（*cmd*，*mode='r'*，*buffering=-1*）
从命令*cmd*打开管道。返回值是一个打开的文件对象，连接到管道，可以根据*模式*是'r'（默认）还是可以读取或写入'w'。该*缓冲*参数的含义与相应的参数内置相同的open()功能。返回的文件对象读取或写入文本字符串而不是字节。

该close方法返回None，如果子进程成功退出，或者子过程的返回码，如果有一个错误。在POSIX系统上，如果返回码是正值，它表示进程的返回值左移一个字节。如果返回码是负数，则处理过程由返回码的取反值给出的信号终止。（例如，如果子进程被终止，则返回值可能是。）在Windows系统上，返回值包含子进程的带符号整数返回码。-signal.SIGKILL

这是通过使用subprocess.Popen; 请参阅该类的文档以获取更多有效的方式来管理和与子流程进行通信。

os.spawnl（*模式*，*路径*，*...*）
os.spawnle（*mode*，*path*，*...*，*env*）
os.spawnlp（*模式*，*文件*，*...*）
os.spawnlpe（*mode*，*file*，*...*，*env*）
os.spawnv（*模式*，*路径*，*参数*）
os.spawnve（*mode*，*path*，*args*，*env*）

os. spawnvp（*模式*，*文件*，*参数*）

os. spawnvpe（*mode*，*file*，*args*，*env* ）

在新进程中执行程序*路径*。

（请注意，该subprocess模块提供了更强大的工具来生成新进程并检索其结果;使用该模块比使用这些函数更可取，尤其要检查使用子进程 模块部分替换旧函数。）

如果*模式*是P_NOWAIT，这个函数返回新进程的进程ID; 如果*模式*是P_WAIT，返回进程的退出代码，如果它正常退出，或者-signal，*信号*是杀死进程的信号。在Windows上，进程ID实际上是进程句柄，因此可以与该waitpid()函数一起使用。

spawn*函数的“l”和“v”变体在传递命令行参数方面有所不同。如果在编写代码时参数的数量是固定的，则“l”变体可能是最容易使用的变体; 各个参数只是成为spawnl*()功能的附加参数 。当参数的数量是可变的时，“v”变体是好的，参数以列表或元组作为*args*参数传递。无论哪种情况，子进程的参数都必须以正在运行的命令的名称开始。

其中包括一个第二“p”的附近的端部（所述变体spawnlp()， spawnlpe()，spawnvp()，和spawnvpe()）将使用 PATH环境变量来定位程序*文件*。当环境正在被替换时（使用其中一个spawn*e变体，在下一段中讨论），新的环境被用作PATH变量。其他变体，spawnl()，spawnle()，spawnv()，和spawnve()，不会使用 PATH变量来定位可执行文件; *路径*必须包含适当的绝对或相对路径。

为spawnle()，spawnlpe()，spawnve()，和spawnvpe()（注意，在“E”这些所有端），则*ENV*参数必须被用于定义新处理的环境变量的映射（它们被用来代替当前进程的环境中）; 功能 spawnl()，spawnlp()，spawnv()，以及spawnvp()所有会导致新进程继承当前进程的环境。请注意，*env*字典中的键和值必须是字符串; 无效的键或值将导致函数失败，返回值为127。

作为一个例子，下面的调用spawnlp()和spawnvpe()是等效的：

```
import os
os.spawnlp(os.P_WAIT, 'cp', 'cp', 'index.html', '/dev/null')

L = ['cp', 'index.html', '/dev/null']
os.spawnvpe(os.P_WAIT, 'cp', L, os.environ)
```

可用性：Unix，Windows。 spawnlp()，spawnlpe()，spawnvp() 而 spawnvpe()不是Windows上可用。 spawnle()并且 spawnve()在Windows上不是线程安全的; 我们建议您改用该 subprocess模块。

*在版本3.6中更改：*接受类似路径的对象。

os. P_NOWAIT

os. P_NOWAITO

功能族的*模式*参数的可能值spawn*。如果给出了这些值中的任何一个，那么spawn*()函数将在创建新进程后立即返回，并将进程ID作为返回值。

可用性：Unix，Windows。

os. P_WAIT

函数族的*模式*参数的可能值spawn*。如果这是以*模式*形式给出的，那么这些spawn*()函数将不会返回，直到新进程运行完成并且将返回运行成功的进程的退出代码，或者-signal信号终止进程。

可用性：Unix，Windows。

os.P_DETACH
os.P_OVERLAY
功能族的*模式*参数的可能值spawn*。这些便携式比上面列出的便携式更少。P_DETACH类似于P_NOWAIT，但是新进程从调用进程的控制台分离。如果P_OVERLAY使用，则当前进程将被替换; 该spawn*函数不会返回。

可用性：Windows。

os.startfile(*path*[,*operation*])
用相关的应用程序启动一个文件。

当没有指定*操作时*'open'，它的作用就像在Windows资源管理器中双击文件一样，或者将文件名作为参数传递给交互式命令shell中的**start**命令：文件以任何应用程序（如果有）的扩展名打开已关联的。

当给出另一个*操作时*，它必须是一个"命令动词"，它指定了该文件应该做什么。Microsoft记录的常见动词是'print'和'edit'（用于文件）以及'explore'和'find'（用于目录）。

startfile()相关应用程序启动后立即返回。没有选项可以等待应用程序关闭，也无法检索应用程序的退出状态。该*路径*参数是相对于当前目录。如果你想使用绝对路径，确保第一个字符不是斜线（'/'）; 底层的Win32 ShellExecute()函数不起作用，如果是。使用该os.path.normpath()函数确保路径为Win32正确编码。

为了减少解释器启动开销，Win32 ShellExecute()函数在第一次调用该函数之前不会解析。如果该功能无法解决，NotImplementedError将会被提出。

可用性：Windows。

os.system(*命令*)
在子shell中执行命令（一个字符串）。这是通过调用标准C函数来实现的system()，并且具有相同的限制。sys.stdin执行的命令的环境中不会反映更改等。如果*命令*产生任何输出，它将被发送到解释器标准输出流。

在Unix上，返回值是以指定格式编码的进程的退出状态wait()。请注意，POSIX没有指定C system()函数的返回值的含义，所以Python函数的返回值是依赖于系统的。

在Windows上，返回值是运行*命令*后由系统shell返回的值。该shell由Windows环境变量给出COMSPEC：通常是**cmd.exe**，它返回命令运行的退出状态; 在使用非本地shell的系统上，请查阅您的shell文档。

该subprocess模块提供了更强大的功能，用于产生新过程并检索其结果; 使用该模块优于使用此功能。有关一些有用的配方，请参阅文档中的"使用子流程模块"一节中的"替换旧功能"一节subprocess。

可用性：Unix，Windows。

os.times()
返回当前的全局流程时间。返回值是一个具有五个属性的对象：

- user - 用户时间
- system - 系统时间
- children_user - 所有子进程的用户时间
- children_system - 所有子进程的系统时间
- elapsed - 自过去的固定点以来的实时时间

为了向后兼容，该目的也表现得像含有五元组user，system，children_user，children_system，和elapsed以该顺序。

请参阅Unix手册页次 *（2）*或相应的Windows平台API文档。在Windows上，只有user和system已知; 其他属性为零。

可用性：Unix，Windows。

*在版本3.3中进行了更改：*返回类型从元组更改为具有命名属性的元组对象。

os.wait()
等待子进程完成，并返回一个包含它的pid和退出状态指示的元组：一个16位数字，其低字节是杀死进程的信号编号，其高字节是退出状态（如果信号数字为零）; 如果生成核心文件，则设置低字节的高位。

可用性：Unix。

os.waitid(*idtype*，*id*，*options*)
等待一个或多个子进程的完成。 *idtype*可以是P_PID，P_PGID或者P_ALL。 *id*指定要等待的pid。 *选项*是由一个或多个ORing构建的WEXITED， WSTOPPED或者WCONTINUED另外可以与OR进行WNOHANG或运算 WNOWAIT。返回值是表示包含在该数据对象siginfo_t的结构，即：si_pid，si_uid，si_signo，si_status， si_code或None，如果WNOHANG是指定的，没有子在一个可等待的状态。

可用性：Unix。

3.3版本的新功能

os.P_PID
os.P_PGID
os.P_ALL
这些是*idtype*中可能的值waitid()。它们会影响*id*的解释。

可用性：Unix。

3.3版本的新功能

os.WEXITED
os.WSTOPPED
os.WNOWAIT

可用于*选项*中的标志waitid()指定要等待的子信号。

可用性：Unix。

3.3版本的新功能

os.CLD_EXITED
os.CLD_DUMPED
os.CLD_TRAPPED
os.CLD_CONTINUED
这些是si_code返回结果 的可能值waitid()。

可用性：Unix。

3.3版本的新功能

os.waitpid(*pid*, *options*)
这个功能的细节在Unix和Windows上有所不同。

在Unix上：等待由进程标识符*pid*给出的子进程的完成，并返回包含进程标识符和退出状态指示（编码为wait()）的元组。调用的语义受整数*选项*值的影响，这些*选项*应该0用于正常操作。

如果*pid*大于0，则waitpid()请求该特定进程的状态信息。如果*pid*是0，则请求是针对当前进程的进程组中的任何子进程的状态。如果*pid*是-1，则该请求适用于当前进程的任何孩子。如果*pid*小于 -1，则为进程组中的任何进程请求状态-pid（*pid*的绝对值）。

一个OSError是提出了与errno的值时，系统调用返回-1。

在Windows上：等待进程句柄*pid*给出的进程完成，并返回包含*pid*的元组，并且其退出状态左移8位（移位使得跨平台使用该函数更容易）。一个*PID*小于或等于0已经在Windows上没有特殊的意义，并提出了一个例外。整数*选项*的值不起作用。*pid*可以指任何已知id的进程，不一定是子进程。 spawn* 调用的函数P_NOWAIT返回适当的进程句柄。

*版本3.5中更改：*如果系统调用中断且信号处理程序不引发异常，则该函数现在重试系统调用而不是引发 InterruptedError异常（请参阅**PEP 475**为理由）。

os.wait3(*选项*)
与之类似waitpid()，除了没有给出进程ID参数，并返回包含子进程ID，退出状态指示和资源使用信息的3元素元组。参考resource。 getrusage()有关资源使用信息的详细信息。选项参数与提供给waitpid()和的 参数相同wait4()。

可用性：Unix。

os.wait4(*pid*, *options*)
与waitpid()包含子进程ID，退出状态指示和资源使用信息的3元素元组类似，返回。参考resource。 getrusage()有关资源使用信息的详细信息。参数wait4()与提供的参数相同waitpid()。

可用性：Unix。

os.WNOHANG
waitpid()立即没有子进程状态时立即返回的选项。在这种情况下函数返回。(0, 0)

可用性：Unix。

os.WCONTINUED
此选项会导致子进程在自上次报告状态后已从作业控制停止继续进行报告。

可用性：一些Unix系统。

os.WUNTRACED
这个选项会导致子进程被报告，如果它们已经被停止，但是它们的当前状态自停止以来还没有被报告。

可用性：Unix。

下面的函数采取一个过程状态代码作为返回的 system() ，wait()或waitpid()作为一个参数。他们可能被用来确定一个过程的处置。

os.WCOREDUMP（*状态*）
返回True如果进程产生核心转储，否则返回False。

可用性：Unix。

os.WIFCONTINUED（*状态*）
返回True如果进程已经从作业控制停止持续，否则返回False。

可用性：Unix。

os.WIFSTOPPED（*状态*）
返回True如果进程已经停止，否则返回 False。

可用性：Unix。

os.WIFSIGNALED（*状态*）
返回True如果进程因信号退出，否则返回 False。

可用性：Unix。

os.WIFEXITED（*状态*）
返回True如果进程使用退出*出口（2）*系统调用，否则返回False。

可用性：Unix。

os.WEXITSTATUS（*状态*）
如果WIFEXITED(status)为true，则将整型参数返回到 *exit（2）*系统调用。否则，返回值是没有意义的。

可用性：Unix。

os.WSTOPSIG(*状态*)
返回导致进程停止的信号。

可用性：Unix。

os.WTERMSIG(*状态*)
返回导致进程退出的信号。

可用性：Unix。

16.1.7。调度程序的接口

这些功能控制操作系统如何为进程分配CPU时间。它们仅在某些Unix平台上可用。有关更多详细信息，请参阅您的Unix手册页。

3.3版本的新功能

如果操作系统支持以下调度策略，则会公开它们。

os.SCHED_OTHER
默认调度策略。

os.SCHED_BATCH
为尝试保留计算机其余部分的交互性的CPU密集型进程调度策略。

os.SCHED_IDLE
为极低优先级的后台任务调度策略。

os.SCHED_SPORADIC
针对零星服务器程序的调度策略。

os.SCHED_FIFO
先进先出调度策略。

os.SCHED_RR
循环调度策略。

os.SCHED_RESET_ON_FORK
该标志可以与任何其他调度策略进行或运算。当具有此标志的进程设置分支时，其子进程的调度策略和优先级将重置为默认值。

class os.sched_param(*sched_priority*)
此类表示中使用的可调谐调度参数 sched_setparam()，sched_setscheduler()和sched_getparam()。它是不可变的。

目前，只有一个可能的参数：

sched_priority

调度策略的调度优先级。

os.sched_get_priority_min(*政策*)
获取*策略*的最低优先级值。*策略*是以上调度策略常数之一。

os.sched_get_priority_max(*政策*)
获取*策略*的最大优先级值。*策略*是以上调度策略常数之一。

os.sched_setscheduler(*pid*, *policy*, *param*)
使用PID *pid*为进程设置调度策略。一个0 的*pid*意味着调用过程。*策略*是以上调度策略常数之一。*参数*是一个sched_param实例。

os.sched_getscheduler(*pid*)
使用PID *pid*返回进程的调度策略。一个0 的*pid*意味着调用过程。结果是上面的一个调度策略常量。

os.sched_setparam(*pid*, *param*)
使用PID *pid*为进程设置调度参数。一个0 的*pid*意味着调用过程。*参数*是一个sched_param实例。

os.sched_getparam(*pid*)
返回调度参数作为sched_param用于与PID的流程实例*的PID*。一个0 的*pid*意味着调用过程。

os.sched_rr_get_interval(*pid*)
返回以秒为单位的循环量子用于与PID的进程*PID*。一个0 的 *pid*意味着调用过程。

os.sched_yield()
自愿放弃CPU。

os.sched_setaffinity(*pid*, *mask*)
用PID *PID*（或当前进程，如果为零）限制进程到一组CPU。 *掩码*是代表进程应该限制到的一组CPU的整数的迭代。

os.sched_getaffinity(*pid*)
返回组CPU与PID的进程的*PID*（或如果零当前进程）被限制到。

16.1.8。其他系统信息

os.confstr(*名字*)
返回字符串值的系统配置值。*name*指定要检索的配置值; 它可能是一个字符串，它是定义的系统值的名称; 这些名称是在许多标准（POSIX，Unix 95，Unix 98等）中指定的。一些平台也定义了其他名称。主机操作系统已知的名称是confstr_names字典的键 。对于未包含在该映射中的配置变量，也会接受传递*名称*的整数。

如果由*名称*指定的配置值未定义，None则返回。

如果*名称*是一个字符串并且未知，ValueError则会引发。如果主机系统不支持*名称*的特定值，即使它包含在内confstr_names，OSError也会errno.EINVAL为错误号引发 一个值。

可用性：Unix。

os.confstr_names
字典映射名称confstr()由主机操作系统为这些名称定义的整数值接受。这可以用来确定系统已知的一组名称。

可用性：Unix。

os.cpu_count()
返回系统中的CPU数量。None如果未确定，则返回。

这个数字不等于当前进程可以使用的CPU数量。可用的CPU数量可以通过len(os.sched_getaffinity(0))

3.4版新增功能

os.getloadavg()
返回系统运行队列中最近1分钟，5分钟和15分钟平均的进程数量，OSError如果无法获得平均负载，则返回上升的进程数量。

可用性：Unix。

os.sysconf(*名字*)
返回整数值的系统配置值。如果由*名称*指定的配置值未定义，-1则返回。关于*名称*参数的注释也confstr()适用于此处; 提供有关已知名称信息的字典由以下给出sysconf_names。

可用性：Unix。

os.sysconf_names
字典映射名称sysconf()由主机操作系统为这些名称定义的整数值接受。这可以用来确定系统已知的一组名称。

可用性：Unix。

以下数据值用于支持路径操作操作。这些都是为所有平台定义的。

os.path模块中定义了更高级的路径名操作。

os.curdir
操作系统用来引用当前目录的常量字符串。这是'.'针对Windows和POSIX的。也可以通过os.path。

os.pardir
操作系统用来引用父目录的常量字符串。这是'..'针对Windows和POSIX的。也可以通过os.path。

os.sep

操作系统用来分隔路径名组件的字符。这适用'/'于POSIX和'\\' Windows。请注意，知道这不足以解析或连接路径名 - 使用 os.path.split()和os.path.join() - 但它偶尔是有用的。也可以通过os.path。

os.altsep
操作系统用来分隔路径名组件的替代字符，或者None只存在一个分隔符。这'/'在Windows系统中设置为 sep反斜杠。也可以通过 os.path。

os.extsep
将基本文件名与扩展名分开的字符; 例如，'.' in os.py。也可以通过os.path。

os.pathsep
操作系统通常使用的字符来分隔搜索路径组件（如in PATH），例如':' POSIX或';' Windows。也可以通过os.path。

os.defpath
如果环境没有 密钥exec*p*， spawn*p*则使用默认搜索路径'PATH'。也可以通过os.path。

os.linesep
用于在当前平台上分隔（或相反，终止）行的字符串。这可能是单个字符，例如'\n'对于POSIX，或者多个字符，例如'\r\n'对于Windows。在编写以文本模式打开的文件时（默认），不要使用 *os.linesep*作为行终止符; '\n'在所有平台上使用一个，而不是。

os.devnull
空设备的文件路径。例如：'/dev/null'用于POSIX，'nul'用于Windows。也可以通过os.path。

os.RTLD_LAZY
os.RTLD_NOW
os.RTLD_GLOBAL
os.RTLD_LOCAL
os.RTLD_NODELETE
os.RTLD_NOLOAD
os.RTLD_DEEPBIND
用于setdlopenflags()和 getdlopenflags()功能的标志。查看Unix手册页面 *dlopen（3）*了解不同标志的含义。

3.3版本的新功能

16.1.9。随机数字

os.getrandom（*size*，*flags = 0*）
获取随机字节的*大小*。该函数可以返回比请求更少的字节。

这些字节可用于种子用户空间随机数生成器或用于加密目的。

getrandom()依靠从设备驱动程序和其他环境噪声源收集到的熵。不必要的读取大量数据，将会对其他用户造成负面影响/dev/random和 /dev/urandom设备。

flags参数是一个可以包含零个或多个以下值的位掩码，它们组合在一起：os.GRND_RANDOMand GRND_NONBLOCK。

另请参阅Linux getrandom（）手册页。

可用性：Linux 3.17及更新版本。

3.6版本中的新功能。

os.urandom（*size*）

返回一串适合加密使用的随机字节*大小*。

该函数从OS特定的随机源中返回随机字节。返回的数据对于加密应用程序来说应该是不可预知的，尽管它的确切质量取决于操作系统的实现。

在Linux上，如果getrandom()系统调用可用，则将其用于阻塞模式：阻塞，直到系统随机熵池初始化（内核收集128位熵）。看到了**PEP 524**为理由。在Linux上，该getrandom()函数可用于以非阻塞模式（使用GRND_NONBLOCK标志）获取随机字节，或轮询直到系统随机熵池被初始化。

在类Unix系统上，从/dev/urandom 设备读取随机字节。如果/dev/urandom设备不可用或不可读， NotImplementedError则引发异常。

在Windows上，它将使用CryptGenRandom()。

> **也可以看看：** 该secrets模块提供更高级别的功能。有关您的平台提供的随机数生成器的简单易用的界面，请参阅random.SystemRandom。

*在3.6.0版中更改：*在Linux上，getrandom()现在用于阻止模式以提高安全性。

*在版本3.5.2中更改：*在Linux上，如果getrandom()系统调用块（urandom熵池尚未初始化），请重新阅读/dev/urandom。

*在版本3.5中进行了更改：*在Linux 3.17及更新的版本上，getrandom()现在可以使用系统调用。在OpenBSD 5.6及更新版本上，getentropy() 现在使用了C 函数。这些函数避免使用内部文件描述符。

os.GRND_NONBLOCK

默认情况下，当读取时/dev/random，getrandom()如果没有随机字节可用，/dev/urandom则在块读取时阻塞，如果熵池尚未初始化。

如果GRND_NONBLOCK标志已设置，则getrandom()在这些情况下不会阻止，而是立即提升BlockingIOError。

3.6版本中的新功能。

os.GRND_RANDOM

如果设置了该位，则从/dev/random池中而不是/dev/urandom池中抽取随机字节。

3.6版本中的新功能。

16.2。 io- 使用流的核心工具

源代码： Lib / io.py

16.2.1。概述

该io模块提供了Python用于处理各种类型I / O的主要工具。主要有三种类型的I / O：*文本I / O*，*二进制I / O* 和*原始I / O*。这些是通用类别，各种后备存储可用于其中的每一种。属于任何这些类别的具体对象称为文件对象。其他常见术语是*流* 和*文件类对象*。

与其类别无关，每个具体流对象还具有各种功能：可以是只读，只写或读写。它也可以允许任意随机访问（向前或向后寻找任何位置），或者只允许顺序访问（例如在套接字或管道的情况下）。

所有的流都注意你给他们的数据类型。例如，给二进制流str的write()方法提供一个对象将引发一个TypeError。所以会给文本流bytes的write()方法一个对象 。

在版本3.3中进行了更改： IOError以前用于筹集的操作现在已经增加OSError ，因为 IOError现在已成为别名OSError。

16.2.1.1。文本I /

文本I / O期望并产生str对象。这意味着只要后备存储本身由字节组成（例如在文件的情况下），就可以透明地编码和解码数据，以及平台特定的换行符的可选转换。

创建文本流的最简单方法是使用open()（可选）指定编码：

```
f = open("myfile.txt", "r", encoding="utf-8")
```

内存中的文本流也可用作StringIO对象：

```
f = io.StringIO("some initial text data")
```

文档流API在文档中有详细描述 TextIOBase。

16.2.1.2。二进制I /

二进制I / O（也称为*缓冲I / O*）需要 类似字节的对象并生成bytes 对象。不执行编码，解码或换行。这种类型的流可以用于各种非文本数据，并且还需要手动控制文本数据的处理。

创建一个二进制流的最简单的方法是open()用'b'在模式字符串：

```
f = open("myfile.jpg", "rb")
```

内存中的二进制流也可用作BytesIO对象：

```
f = io.BytesIO(b"some initial binary data: \x00\x01")
```

二进制流API在文档中详细描述 BufferedIOBase。

其他库模块可能会提供其他方式来创建文本或二进制流。看socket.socket.makefile()例如。

16.2.1.3。原始I /

原始I / O（也称为*无缓冲I / O*）通常用作二进制和文本流的低级构建块; 从用户代码直接操作原始流是很有用的。不过，您可以通过在缓冲禁用的情况下以二进制模式打开文件来创建原始流：

```
f = open("myfile.jpg", "rb", buffering=0)
```

原始流API在文档中详细描述RawIOBase。

16.2.2。高级模块接口

io.DEFAULT_BUFFER_SIZE
　包含模块的缓冲I / O类使用的默认缓冲区大小的int。 如果可能的话，open()使用文件的blksize（通过获得 os.stat() ）。

io.open（*file*，*mode ='r'*，*buffering = -1*，*encoding = None*，*errors = None*，*newline = None*，*closefd = True*，*opener = None* ）
　这是内建open()函数的别名。

*异常*io.BlockingIOError
　这是内建BlockingIOError 异常的兼容别名。

*异常*io.UnsupportedOperation
　例外的继承OSError和ValueError当不支持的操作上调用流时引发。

16.2.2.1。内存中的流

也可以使用一个str或类似字节的对象作为读写文件。对于字符串StringIO可以像在文本模式下打开的文件一样使用。 BytesIO可以像使用二进制模式打开的文件一样使用。两者都提供随机访问的完整读写功能。

也可以看看:

sys
　包含标准的IO流：sys.stdin ，sys.stdout ，和sys.stderr。

16.2.3。类层次结构

I / O流的实现被组织为类的层次结构。第一个 抽象基类（ ABCs ），用于指定不同类别的流，然后是提供标准流实现的具体类。

> **注意:** 抽象基类还提供了一些方法的默认实现，以帮助实现具体的流类。例如，BufferedIOBase提供的未优化的实现方式 readinto() 和readline()。

I / O层次结构的顶部是抽象基类IOBase。它定义了流的基本接口。但是，请注意，读取和写入流之间没有分离。UnsupportedOperation如果它们不支持给定的操作，则允许实施。

在RawIOBaseABC延伸IOBase。它处理字节读取和写入流。 FileIO子类RawIOBase 为机器文件系统中的文件提供接口。

在 BufferedIOBase 上一个原始字节流ABC涉及缓冲（ RawIOBase ）。它的子类， BufferedWriter， BufferedReader，和BufferedRWPair缓冲流是可读，可写，以及可读写的。 BufferedRandom 为随机访问流提供缓冲接口。另一个 BufferedIOBase子类BytesIO是内存中字节流。

的TextIOBaseABC，的另一亚类中IOBase，处理与其字节表示文本流，并处理的编码和解码，并从字符串。 TextIOWrapper，它扩展了它，是缓冲的原始流（ BufferedIOBase ）的缓冲文本接口。最后， StringIO是文本的内存流。

参数名称不是规范的一部分，只有参数 open() 意图用作关键字参数。

下表总结了该io模块提供的ABC ：

ABC	继承	存根方法	Mixin方法和属性
IOBase		fileno，seek和truncate	close，closed，__enter__，__exit__，flush，isatty，__iter__，__next__，readable，readline，readlines，seekable，tell，writable，和writelines
RawIOBase	IOBase	readinto 和 write	继承的IOBase方法read，和readall
BufferedIOBase	IOBase	detach，read， read1，和write	继承的IOBase方法readinto，和readinto1
TextIOBase	IOBase	detach，read，readline，和 write	继承的IOBase方法encoding， errors以及newlines

16.2.3.1。I / O基类

*类*io. IOBase

所有I / O类的抽象基类，作用于字节流。没有公共构造函数。

此类为许多派生类可以选择性覆盖的方法提供了空抽象实现; 默认实现代表无法读取，写入或查找的文件。

即使IOBase没有声明read()，readinto()或者write()因为它们的签名会有所不同，实现和客户端应该将这些方法视为接口的一部分。而且，实现可能会在他们不支持的操作被调用时引发ValueError（或UnsupportedOperation）。

用于从文件读取或写入文件的二进制数据的基本类型是 bytes。其他类似字节的对象也被接受为方法参数。在某些情况下，例如 需要readinto()的可写对象bytearray。文本I / O类使用str数据。

请注意，调用封闭流中的任何方法（即使是查询）都是未定义的。 ValueError在这种情况下，实现可能会提高。

IOBase（及其子类）支持迭代器协议，这意味着IOBase可以通过产生流中的行来迭代对象。根据流是二进制流（产生字节）还是文本流（产生字符串），行的定义略有不同。见readline()下文。

IOBase也是上下文管理器，因此支持该 with声明。在这个例子中，*文件*在with声明套件结束后关闭 - 即使发生异常：

```
with open('spam.txt', 'w') as file:
    file.write('Spam and eggs!')
```

IOBase 提供这些数据属性和方法：

close ( )
: 冲洗并关闭此流。如果该文件已关闭，则此方法无效。一旦文件关闭，对文件的任何操作（例如读或写）都会引发一次ValueError。

  为了方便，可以多次调用此方法; 然而，只有第一个电话会起作用。

closed
: True 如果流被关闭。

fileno ( )
: 返回流的底层文件描述符（整数）（如果存在）。一个OSError如果IO对象不使用文件描述符提高。

flush ( )
: 刷新流的写入缓冲区（如果适用）。这对于只读流和非阻塞流没有任何作用。

isatty ( )
: 返回True如果流是交互式（即，连接到一个终端/ TTY设备）。

readable ( )
: True如果可以从中读取流，则返回。如果False，read() 会提高OSError。

readline (*size = -1*)
: 从流中读取并返回一行。如果*大小*指定，顶多*大小*字节将被读取。

  行结束符总是b'\n'用于二进制文件; 对于文本文件，可以使用*换行符*参数来open()选择识别的行结束符。

readlines (*hint = -1*)
: 从流中读取并返回行列表。 可以指定*提示*来控制读取的行数：如果所有行的总大小（字节/字符）超过*提示*，则不会再读取行。

请注意，已经可以在不调用的情况下迭代文件对象。 `for line in file: ... file.readlines()`

seek（*offset*[，*whence*]）

将流位置更改为给定的字节*偏移量*。 *偏移*是相对于由指示的位置解释*何处*。*whence*的默认值是SEEK_SET。*从哪里来的*价值是：

- SEEK_SET或0- 流的开始（默认）；*偏移*应为零或正数
- SEEK_CUR或1- 当前流位置；*抵消*可能是负面的
- SEEK_END或2- 流的结尾；*抵消*通常是负面的

返回新的绝对位置。

*新的3.1版：*该SEEK_*常数。

*V3.3中的新增功能：*某些操作系统可能支持其他值，例如 os.SEEK_HOLE 或 os.SEEK_DATA。文件的有效值可能取决于它是以文本还是二进制模式打开。

seekable（）

True如果流支持随机访问，则返回。如果False， seek()，tell()并且truncate()将提高OSError。

tell（）

返回当前的流位置。

truncate（*size = None*）

将流大小调整为以字节为单位的给定*大小*（如果未指定*大小*，则将其*大小设置*为当前位置）。当前流的位置不变。此调整大小可以扩展或减少当前文件大小。在扩展的情况下，新文件区的内容取决于平台（在大多数系统中，附加字节是零填充的）。新的文件大小被返回。

在版本3.5中更改： Windows现在将扩展时填充文件。

writable（）

True如果流支持写入，则返回。如果False， write()并且truncate()会升高OSError。

writelines（*线*）

写入流列表。没有添加行分隔符，因此通常提供的每行都在末尾具有行分隔符。

__del__（）

准备对象销毁。 IOBase提供了调用实例close()方法的此方法的默认实现 。

*类*io.RawIOBase

原始二进制I / O的基类。它继承IOBase。没有公共构造函数。

原始二进制I / O通常提供对底层操作系统设备或API的低级别访问，并且不会尝试将其封装在高级基元中（这留给了缓冲I / O和文本I / O，稍后会在本文中进行描述页）。

除了属性和方法以外IOBase， RawIOBase还提供了以下方法：

read（*size = -1*）

从对象中读取*大小*字节并返回它们。为方便起见，如果*大小*未指定或-1，则返回EOF之前的所有字节。否则，只会有一次系统调用。比更少*的大小*，如果操作系统调用返回比较少的字节，可能会返回*大小*字节。

如果返回0字节，并且*大小*不为0，则表示文件结束。如果对象处于非阻塞模式并且没有可用字节， None则返回。

默认实现遵循readall()和 readinto()。

readall ( )

读取并返回流中的所有字节，直到EOF，如有必要，使用多个对流的调用。

readinto (*b*)

将字节读入预先分配的可写类 字节对象 *b*，并返回读取的字节数。如果对象处于非阻塞模式并且没有可用字节，None则返回。

write (*b*)

写给定类字节对象，*b*，以底层的原始流，并返回写入的字节的数目。这可能小于*b*的长度（以字节为单位），具体取决于基础原始流的细节，特别是在非阻塞模式下。 None如果原始流设置为不阻塞，并且没有单个字节可以很容易地写入它，则返回。调用者可以在此方法返回后释放或变异*b*，因此实现应该只 在方法调用期间访问*b*。

*类*io. BufferedIOBase

支持某种缓冲的二进制流的基类。它继承IOBase。没有公共构造函数。

主要的区别RawIOBase在于方法read()， readinto()并且write()将尝试（分别）读取尽可能多的输入，或者消耗所有给定的输出，但可能会造成多次系统调用。

另外，BlockingIOError如果基础原始数据流处于非阻塞模式并且不能提取或提供足够的数据，那么可以引发这些方法; 不像他们的RawIOBase同行，他们永远不会回来None。

此外，该read()方法没有遵循的默认实现readinto()。

一个典型的BufferedIOBase实现不应该从继承 RawIOBase的实现，但是一个包，喜欢BufferedWriter和BufferedReader这样做。

BufferedIOBase提供或覆盖这些方法和属性以及来自以下的方法和属性IOBase：

raw

处理的基础原始流（ RawIOBase实例） BufferedIOBase。这不是BufferedIOBaseAPI的一部分， 在某些实现中可能不存在。

detach ( )

将底层原始流从缓冲区中分离出来并返回。

在原始流被分离后，缓冲区处于不可用状态。

一些缓冲区，例如BytesIO，没有从此方法返回的单个原始流的概念。他们提高UnsupportedOperation。

版本3.1中的新功能。

read (*size = -1*)

读取并返回*大小*字节。如果参数被省略，None或者为负数，则读取并返回数据直到达到EOF。 bytes如果流已经在EOF中，则返回空 对象。

如果参数是肯定的，并且基础原始流不是交互式的，则可以发出多个原始读取以满足字节计数（除非首先达到EOF）。但对于交互式原始数据流，最多只会发布一次原始读取，而短暂结果并不意味着即将发生EOF。

BlockingIOError如果底层原始数据流处于非阻塞模式并且此刻没有可用数据，则会引发A.

read1 (*size = -1*)

阅读并返回多达*大小*字节，最多一个调用底层的原始数据流的read()（或readinto()）方法。如果你在一个BufferedIOBase 对象上实现自己的缓冲，这会很有用。

readinto (*b*)

将字节读入预先分配的，可写 的类似字节的对象 *b*并返回读取的字节数。

同样read()，可能会向基础原始流发布多个读取，除非后者是交互式的。

BlockingIOError如果底层原始数据流处于非阻塞模式并且此刻没有可用数据，则会引发A.

readinto1 (*b*)

将字节读入预先分配的可写类 字节对象 *b*，使用最多一次调用基础原始流的read()（或readinto()）方法。返回读取的字节数。

BlockingIOError如果底层原始数据流处于非阻塞模式并且此刻没有可用数据，则会引发A.

3.5版本中的新功能。

write (*b*)

写给定类字节对象，*b*，并返回写入的字节的数量（总是等于的长度*b*以字节为单位，因为如果写入失败的OSError将提高）。根据实际的实现情况，这些字节可能很容易写入基础流，或者由于性能和延迟原因而保存在缓冲区中。

处于非阻塞模式时，BlockingIOError如果需要将数据写入原始数据流，则会引发a，但无法接受所有没有阻塞的数据。

调用者可以在此方法返回后释放或变异*b*，因此实现应该只在方法调用期间访问*b*。

16.2.3.2。原始文件I /

class io. FileIO (*name*，*mode ='r'*，*closefd = True*，*opener = None*)

FileIO表示包含字节数据的OS级文件。它实现了RawIOBase接口（因此也实现了接口IOBase）。

这个*名字*可以是以下两件事之一：

- bytes表示将要打开的文件的路径的字符串或对象。在这种情况下，closefd必须是True（默认），否则会引发错误。
- 一个整数，表示生成的FileIO对象将访问的现有OS级文件描述符的编号。当FileIO对象关闭时，这个fd也将被关闭，除非*closefd* 被设置为False。

该*模式*可为'r'，'w'，'x'或'a'用于读取（默认），写作，创造独家或追加。如果文件在打开或写入时不存在，则该文件将被创建; 它会在写入时被截断。FileExistsError如果它在创建时已经存在，将会被提出。打开创建文件意味着写入，因此该模式的行为与之类似'w'。'+'将模式添加 到允许同时读取和写入的模式。

在read()（与积极的参数调用），readinto() 并write()在这个类的方法只会让一个系统调用。

可以通过传递可调用的*开罐器*来使用自定义*开罐器*。然后通过使用（*name*，*flags*）调用*opener*来获得文件对象的底层文件描述符。*开*叫*器*必须返回一个打开的文件描述符（ 作为*开*叫*者的*结果传递类似于传递的功能 ）。 os.openNone

新创建的文件是不可继承的。

有关open()使用*opener* 参数的示例，请参见内置函数。

*在3.3版本的改变：*在*揭幕战*中添加参数。该'x'模式已添加。

*在版本3.4中更改：*该文件现在是不可继承的。

除了从属性和方法IOBase以及 RawIOBase，FileIO提供以下数据属性：

mode
: 在构造函数中给出的模式。

name
: 文件名称。这是在构造函数中没有给出名称时该文件的文件描述符。

16.2.3.3。缓冲流

缓冲I / O流为I / O设备提供了比原始I / O更高级别的接口。

class io.BytesIO（[*initial_bytes*]）
: 使用内存中字节缓冲区的流实现。它继承 BufferedIOBase。该close()方法被调用时，缓冲区被丢弃 。

可选参数*initial_bytes*是一个类似字节的对象，它包含初始数据。

BytesIO提供或覆盖除了那些从这些方法BufferedIOBase和IOBase：

getbuffer（）
: 在缓冲区的内容上返回一个可读写的视图而不复制它们。另外，改变视图将透明地更新缓冲区的内容：

```
>>> b = io.BytesIO(b"abcdef")
>>> view = b.getbuffer()
>>> view[2:4] = b"56"
>>> b.getvalue()
b'ab56ef'
```

注意: 只要视图存在，BytesIO对象就不能调整大小或关闭。

3.2版本中的新功能

getvalue ( )

返回bytes包含缓冲区的全部内容。

read1 ( )

在BytesIO，这是一样的read()。

readinto1 ( )

在BytesIO，这是一样的readinto()。

3.5版本中的新功能。

*类*io. BufferedReader (*raw*, *buffer_size = DEFAULT_BUFFER_SIZE*)

缓冲区提供对可读，顺序RawIOBase对象的更高级访问 。它继承BufferedIOBase。从这个对象读取数据时，可能会从基础原始数据流请求更大量的数据，并将其保存在内部缓冲区中。缓存的数据可以在随后的读取中直接返回。

构造函数BufferedReader为给定的可读 *原始流*和*buffer_size*创建一个。如果*buffer_size*被省略， DEFAULT_BUFFER_SIZE则使用。

BufferedReader提供或覆盖除了那些从这些方法BufferedIOBase和IOBase ：

peek ([*size*])

返回流中的字节而不推进位置。对原始流最多进行一次读取以满足呼叫。返回的字节数可能少于或多于请求的数量。

read ([*size*])

读取并返回*大小*字节，或者如果*大小*未给出或为负数，则直到EOF或读取调用将在非阻塞模式下阻塞。

read1 (*size*)

只需在原始流上调用一次，即可读取并返回*大小*字节。如果至少有一个字节被缓存，则只返回缓冲的字节。否则，会创建一个原始流读取调用。

*类*io. BufferedWriter (*raw*, *buffer_size = DEFAULT_BUFFER_SIZE*)

提供对可写入，顺序RawIOBase对象的更高级别访问的缓冲区 。它继承BufferedIOBase。写入此对象时，通常将数据放入内部缓冲区中。缓冲区将RawIOBase 在各种条件下写入到底层对象，包括：

- 当缓冲区对于所有未决数据而言太小时;
- 何时flush()被调用;

- 当seek()被请求时（BufferedRandom对象）；
- 当BufferedWriter物体被关闭或毁坏时。

构造函数BufferedWriter为给定的可写 *原始*流创建一个。如果没有给出*buffer_size*，则默认为 DEFAULT_BUFFER_SIZE。

BufferedWriter提供或覆盖除了那些从这些方法BufferedIOBase和IOBase：

flush（）

强制缓冲区中保存的字节流入原始流。BlockingIOError如果原始流阻塞，则应该提出A.

write（*b*）

编写类字节对象，*b*，并返回写入的字节数。处于非阻塞模式时，BlockingIOError如果需要写出缓冲区但未加工的流阻止，则会引发a 。

*类*io.BufferedRandom（*raw*，*buffer_size = DEFAULT_BUFFER_SIZE*）

随机访问流的缓冲接口。它继承 BufferedReader和BufferedWriter进一步的支持 seek()和tell()功能。

构造函数为第一个参数中给出的可搜索原始流创建一个读取器和写入器。如果*buffer_size*被省略，则默认为 DEFAULT_BUFFER_SIZE。

BufferedRandom有能力做任何事情BufferedReader或 BufferedWriter能做。

*类*io.BufferedRWPair（*reader*，*writer*，*buffer_size = DEFAULT_BUFFER_SIZE*）

一个缓冲I / O对象，将两个单向RawIOBase 对象（一个可读，另一个可写）组合为单个双向端点。它继承BufferedIOBase。

*读取器*和*写入器*是RawIOBase分别是可读且可写的对象。如果*buffer_size*被省略，则默认为DEFAULT_BUFFER_SIZE。

BufferedRWPair实现BufferedIOBase除了detach()引发的 所有方法UnsupportedOperation。

> **警告：** BufferedRWPair不会尝试同步对其基础原始流的访问。你不应该把它和读者和作者一样传递给它; BufferedRandom改为使用。

16.2.3.4。文本I /

*类*io.TextIOBase

文本流的基类。这个类提供了一个基于字符和行的接口来流I / O。没有readinto()方法，因为Python的字符串是不可变的。它继承IOBase。没有公共构造函数。

TextIOBase提供或覆盖这些数据属性和方法以及来自以下方面的数据属性和方法IOBase：

encoding

用于将流的字节解码为字符串并将字符串编码为字节的编码名称。

errors

解码器或编码器的错误设置。

newlines
一个字符串，一个字符串元组，或者None，表示到目前为止翻译的换行符。根据实现和初始构造函数标志，这可能不可用。

buffer
基本的二进制缓冲区（一个BufferedIOBase实例） TextIOBase处理。这不是TextIOBaseAPI的一部分，在某些实现中可能不存在。

detach()
分离底层二进制缓冲区TextIOBase并将其返回。

底层缓冲区分离后，TextIOBase处于不可用状态。

一些TextIOBase实现，比如StringIO，可能没有底层缓冲的概念，并且调用这个方法会提高UnsupportedOperation。

版本3.1中的新功能。

read(*size*)
从流中读取并返回最多*大小的*字符作为单个字符 str。如果*大小*为负值None，则直到EOF。

readline(*size = -1*)
阅读直到换行或EOF并返回一个str。如果流已经在EOF中，则返回空字符串。

如果指定了*大小*，最多可以读取*大小*字符。

seek(*offset*[, *whence*])
将流位置更改为给定的*偏移量*。行为取决于*哪个*参数。*whence*的默认值是 SEEK_SET。

- SEEK_SET或者0：从流的开头寻找（默认）；*偏移量*必须是返回的数字TextIOBase.tell()或零。任何其他*偏移*值都会产生未定义的行为。
- SEEK_CUR或者1：“寻找”到当前位置；*偏移*必须为零，这是一个无操作（所有其他值不受支持）。
- SEEK_END或者2：寻求流的结束；*偏移量*必须为零（所有其他值不受支持）。

将新的绝对位置作为不透明数字返回。

*新的3.1版：*该SEEK_*常数。

tell()
将当前流位置作为不透明数字返回。该数字通常不代表底层二进制存储中的字节数。

write(*s*)
将字符串*s*写入流并返回写入的字符数。

class io.TextIOWrapper(*buffer*, *encoding = None*, *errors = None*, *newline = None*, *line_buffering = False*, *write_through = False*)
一个BufferedIOBase二进制流缓冲的文本流。它继承TextIOBase。

*编码*给出了流将被解码或编码的编码的名称。它默认为locale.getpreferredencoding(False)。

*错误*是一个可选字符串，指定如何处理编码和解码错误。如果存在编码错误（默认设置具有相同效果），则通过'strict'以引发ValueError异常None，或通过'ignore'以忽略错误。（请注意，忽略编码错误可能会导致数据丢失。）'replace'会导致将替换标记（例如'?'）插入存在格式错误的数据的位置。'backslashreplace'导致格式错误的数据被替换为反斜杠转义序列。写入时，'xmlcharrefreplace'（用适当的XML字符引用'namereplace'替换）或（用换\N{...}码序列替换）可以使用。任何其他已经注册的错误处理名称codecs.register_error()也是有效的。

*换行符*控制着行结束的处理方式。它可以是None，''，'\n'，'\r'，和'\r\n'。它的工作原理如下：

- 当从流中读取输入时，如果*换行符*是None，则启用通用换行符模式。输入中的行可以以'\n'，，'\r'或结束'\r\n'，并且'\n'在返回给调用者之前将这些行翻译成。如果是''，则启用通用换行符模式，但行结束符将返回给调用方未翻译。如果它具有任何其他合法值，则输入行仅由给定字符串终止，并且行尾以未翻译形式返回给调用者。
- 将输出写入流时，如果*换行符*是None，则'\n'写入的任何字符都将转换为系统默认行分隔符 os.linesep。如果*换行符*是''或'\n'，则不会进行翻译。如果*换行符*是任何其他合法值，则'\n'写入的任何字符都将转换为给定的字符串。

如果*line_buffering*为True，flush()则在写入调用包含换行符或回车时隐含。

如果*write_through*是True，write()则保证调用不被缓冲：写入TextIOWrapper 对象的任何数据立即被处理到其基础二进制*缓冲区*。

*改变在3.3版本：*该*write_through*参数已被添加。

*版本3.3中更改：*现在使用默认*编码*，locale.getpreferredencoding(False) 而不是locale.getpreferredencoding()。不要使用更改临时区域设置编码locale.setlocale()，请使用当前区域设置编码，而不是用户首选编码。

TextIOWrapper除了那些TextIOBase和它的父母之外还提供了一个属性：

line_buffering
线路缓冲是否启用。

class io.StringIO(*initial_value=''*, *newline='\n'*)
用于文本I / O的内存中的流。close()调用方法时，文本缓冲区被丢弃。

缓冲区的初始值可以通过提供*initial_value*来设置。如果启用了换行符转换，则换行符将被编码，就像通过 write()。该流位于缓冲区的开始处。

该*新行*的说法就像是的TextIOWrapper。默认情况下，只考虑\n字符作为行的末尾，并且不执行换行。如果*换行符*设置为None，则换行符会\n在所有平台上写入，但通用换行符解码在读取时仍然执行。

StringIO除了来自TextIOBase其父母的方法外，还提供了此方法：

getvalue()

返回一个str包含缓冲区的全部内容。换行符被解码read()，尽管流位置没有改变。

用法示例：

```
import io

output = io.StringIO()
output.write('First line.\n')
print('Second line.', file=output)

# Retrieve file contents -- this will be
# 'First line.\nSecond line.\n'
contents = output.getvalue()

# Close object and discard memory buffer --
# .getvalue() will now raise an exception.
output.close()
```

*类*io.IncrementalNewlineDecoder

帮助编解码器对通用换行符模式的换行符进行解码。它继承codecs.IncrementalDecoder。

16.2.4。性能

本节讨论提供的具体I / O实现的性能。

16.2.4.1。二进制I /

即使用户请求单个字节，通过读写大块数据，缓冲I / O也可以隐藏任何调用和执行操作系统无缓冲I / O例程的低效率。增益取决于操作系统和执行的I / O种类。例如，在一些现代操作系统（如Linux）上，无缓冲磁盘I / O可以像缓冲I / O一样快。但是，底线是缓冲I / O提供可预测的性能，无论平台和支持设备如何。因此，对二进制数据使用缓冲I / O而非无缓冲I / O几乎总是可取的。

16.2.4.2。文本I /

二进制存储（例如文件）上的文本I / O比同一存储上的二进制I / O慢得多，因为它需要使用字符编解码器在Unicode和二进制数据之间进行转换。这可能会变得很明显，处理大量日志文件等大量文本数据。此外， TextIOWrapper.tell()和TextIOWrapper.seek()都相当缓慢，由于使用的重建算法。

StringIO但是，它是一个本地内存unicode容器，并且会显示类似的速度BytesIO。

16.2.4.3。多线程

FileIO对象是线程安全的，以至于操作系统调用的程度（例如read(2)在Unix下），它们也是线程安全的。

二进制缓冲对象（的实例 BufferedReader ， BufferedWriter ， BufferedRandom 和 BufferedRWPair ）使用锁保护它们的内部结构; 因此从多个线程一次调用它们是安全的。

TextIOWrapper 对象不是线程安全的。

16.2.4.4。重入

二进制缓冲对象（实例BufferedReader ， BufferedWriter ，BufferedRandom和BufferedRWPair ）是不可重入。虽然在正常情况下不会发生可重入呼叫，但它们可能来自在signal处理程序中执行I / O。如果线程试图重新输入已经访问的缓冲对象，RuntimeError 则会引发a。请注意，这并不禁止不同的线程进入缓冲对象。

上述隐式扩展到文本文件，因为open()函数会将缓冲对象包装在一个TextIOWrapper。这包括标准流，因此也影响内置功能print()。

16.3。 time- 时间访问和转换

该模块提供各种与时间有关的功能。有关相关功能，另请参阅datetime和calendar模块。

尽管此模块始终可用，但并非所有平台上都提供了所有功能。本模块中定义的大多数函数调用具有相同名称的平台C库函数。查阅平台文档有时可能会有帮助，因为这些功能的语义在不同平台之间有所不同。

为了解释一些术语和惯例。

- 该*时期*是时间开始，在此点，并与平台有关。对于Unix，时代是1970年1月1日00:00:00（UTC）。要了解特定平台上的时代，请看 time.gmtime(0)。
- 术语*从epoch秒*是指从epoch的经过秒数的总数，通常会排除 闰秒。在所有符合POSIX标准的平台上，闰秒不包括在内。
- 本模块中的功能可能无法处理时代之前或未来的日期和时间。未来的截止点由C库确定;对于32位系统，通常在2038年。
- **2000年（Y2K）问题**：Python取决于平台的C库，它通常没有2000年问题，因为所有日期和时间都在内部表示为从时代开始的秒数。 strptime() 给定%y格式代码时，函数可以解析2位数年份。当解析2位数年份时，它们根据POSIX和ISO C标准转换：值69-99映射到1969-1999，值0-68映射到2000-2068。
- UTC的协调世界时（以前称为格林威治标准时间，或GMT）。UTC的缩写不是一个错误，而是英法之间的妥协。
- 夏令时是夏令时，是在一年中部分时间（通常）一小时调整时区的时间。DST规则是魔术（由当地法律决定），并且可能每年都会发生变化。C库有一个包含本地规则的表（通常从系统文件中读取灵活性），并且是这方面真正智慧的唯一来源。
- 各种实时函数的精度可能会低于表达其值或参数的单位所建议的精度。例如，在大多数Unix系统中，时钟“滴答”只有每秒50或100次。
- 在另一方面，精度time()和sleep()比其的Unix等效物更好：次被表示为浮点数， time() 返回可用最准确的时间（使用的Unix gettimeofday()如果有的话），以及sleep()将接受具有非零分数的时间（Unix的select()使用在可能的情况下执行此操作）。
- 如通过返回的时间值gmtime() ，localtime()和 strptime() ，并通过接受asctime() ，mktime()并且 strftime() ，是9个整数的序列。的返回值 gmtime() ，localtime()以及strptime()还提供属性的各个字段的名称。

  请参阅struct_time这些对象的描述。

  *版本3.3中已更改：*该struct_time类型已扩展为在平台支持相应成员时提供tm_gmtoff 和tm_zone属性 。 struct tm

  *改变在3.6版本：*该struct_time属性tm_gmtoff和tm_zone 现在可以在所有平台上。
- 使用以下功能在时间表示之间进行转换：

从	至	使用
自纪元以来的秒数	struct_time 以UTC表示	gmtime()
自纪元以来的秒数	struct_time 在当地时间	localtime()
struct_time 以UTC表示	自纪元以来的秒数	calendar.timegm()
struct_time 在当地时间	自纪元以来的秒数	mktime()

16.3.1。函数

time.**asctime**([*t*])

转换的元组或struct_time表示如通过返回的时间 gmtime()或localtime()到以下形式的字符串：。如果未提供*t*，则使用返回的当前时间。区域设置信息不被使用 。'Sun Jun 20 23:21:05 1993' localtime() asctime()

> **注意：** 与同名的C函数不同，asctime()不会添加尾随换行符。

time.**clock**()

在Unix上，将当前处理器时间返回为以秒为单位的浮点数。精确性，实际上"处理器时间"含义的定义取决于具有相同名称的C函数的精确度。

在Windows上，此函数根据Win32函数返回自第一次调用该函数以来的挂钟时间，作为浮点数 QueryPerformanceCounter()。分辨率通常优于1微秒。

*自3.3版弃用：*此功能的行为取决于平台：根据您的要求使用 perf_counter() 或 process_time()取而代之，以获得定义良好的行为。

time.**clock_getres**(*clk_id*)

返回指定时钟*clk_id*的分辨率（精度）。有关*clk_id*的可接受值列表，请参阅 时钟ID常量。

可用性：Unix。

3.3版本的新功能

time.**clock_gettime**(*clk_id*)

返回指定时钟*clk_id*的时间。有关*clk_id*的可接受值列表，请参阅 时钟ID常量。

可用性：Unix。

3.3版本的新功能

time.**clock_settime**(*clk_id*, *time*)

设置指定时钟*clk_id*的时间。目前， CLOCK_REALTIME是*clk_id*唯一可接受的值。

可用性：Unix。

3.3版本的新功能

time.ctime([*秒*])

将时间以秒为单位的时间转换为表示本地时间的字符串。如果没有提供*secs*，或者使用None返回的当前时间time()。 ctime(secs)相当于 asctime(localtime(secs))。区域设置信息不被使用ctime()。

time.get_clock_info(*名字*)

获取有关指定时钟的信息作为名称空间对象。支持的时钟名称和相应的函数来读取它们的值是：

- 'clock'：time.clock()
- 'monotonic'：time.monotonic()
- 'perf_counter'：time.perf_counter()
- 'process_time'：time.process_time()
- 'time'：time.time()

结果具有以下属性：

- *可调*：True如果时钟可以自动更改（例如通过NTP守护进程）或由系统管理员手动更改，False否则
- *实现*：用于获取时钟值的底层C函数的名称。请参阅时钟ID常量以获取可能的值。
- *单调*：True如果时钟不能倒退， False否则
- *分辨率*：以秒为单位的时钟分辨率（float）

3.3版本的新功能

time.gmtime([*秒*])

将从时期开始以秒为单位的时间转换为struct_timeUTC中dst标志始终为零的时间。如果没有提供*secs*，或者 使用None返回的当前时间time()。一秒钟的分数被忽略。参见上面的struct_time对象描述 。请参阅calendar.timegm()此功能的反面。

time.localtime([*秒*])

像gmtime()但转换到当地时间。如果没有提供*secs*，或者 使用None返回的当前时间time()。1当DST适用于给定时间时，dst标志被设置为。

time.mktime(*t*)

这是与之相反的功能localtime()。它的参数是 struct_time或满9元组（因为需要DST标志;使用-1 ，如果它是未知的DST标志），这表示在时间*本地*时间，而不是UTC。它返回一个浮点数，与之兼容time()。如果输入值不能被表示为有效时间， OverflowError或者ValueError将被提升（这取决于无效值是被Python还是底层C库捕获）。它可以产生时间的最早日期是平台相关的。

time.monotonic()

返回单调时钟的值（小数秒），即不能倒退的时钟。时钟不受系统时钟更新的影响。返回值的参考点未定义，因此只有连续调用结果之间的差异才有效。

在Windows之前的版本中，monotonic()检测到 GetTickCount()整数溢出（32位，在49.7天后翻转）。每次检测到溢出时，它将内部历元（参考时间）增加2^{32}。历元存储在本地进程状态中，因此在monotonic()运行超过49天的两个Python进程中，值可能不同。在更新版本的Windows和其他操作系统上，monotonic()是系统范围的。

3.3版本的新功能

*在版本3.5中更改：*该功能现在始终可用。

time.perf_counter ( )

返回性能计数器的值（以分数秒为单位），即具有最高可用分辨率的时钟以测量短时间。它包括睡眠时间和系统范围内的时间。返回值的参考点未定义，因此只有连续调用结果之间的差异才有效。

3.3版本的新功能

time.process_time ( )

返回当前进程的系统和用户CPU时间之和的值（小数秒）。它不包括睡眠时间。它在整个流程范围内定义。返回值的参考点未定义，因此只有连续调用结果之间的差异才有效。

3.3版本的新功能

time.sleep (*秒*)

暂停执行调用线程达给定的秒数。该参数可能是一个浮点数，以表示更准确的睡眠时间。实际的中断时间可能会少于所要求的时间，因为任何被捕获的信号都将终止该sleep()信号的捕获程序的后续执行。而且，由于系统中的其他活动的调度，中止时间可能比任意数量所要求的更长。

*在版本3.5中更改：*即使睡眠被信号中断，该函数现在也会睡眠至少*秒*，除非信号处理程序引发异常（请参阅 **PEP 475**为理由）。

time.strftime (*format* [, *t*])

根据*format* 参数的指定，将元组或struct_time时间转换为字符串返回 gmtime()或localtime()转换为字符串。如果未提供*t*，则使用返回的当前时间 。 *格式*必须是一个字符串。 如果*t*中的任何字段超出了允许的范围，则会引发此问题。localtime()ValueError

0是时间元组中任何位置的合法参数; 如果它通常是非法的，那么这个值就会被强制为一个正确的值。

以下指令可以嵌入*格式*字符串中。它们显示时没有可选的字段宽度和精度规格，并且被strftime()结果中的指示字符替换：

指示	含义	笔记
%a	区域设置的缩写星期几名称。	
%A	Locale的完整星期几名称。	
%b	区域设置的缩写月份名称。	
%B	区域设置的全月名称。	
%c	区域设置的适当日期和时间表示。	
%d	一个月的日期为十进制数[01,31]。	
%H	小时（24小时制）作为十进制数[00,23]。	
%I	小时（12小时制）作为十进制数[01,12]。	

指示	含义	笔记
%j	一年中的一天是十进制数[001,366]。	
%m	月为十进制数[01,12]。	
%M	分钟为十进制数[00,59]。	
%p	区域设置相当于AM或PM。	（1）
%S	秒作为十进制数[00,61]。	（2）
%U	一年中的星期数（星期日作为一周的第一天）作为十进制数[00,53]。第一个星期日前一年的所有日子都被认为是在第0周。	（3）
%w	星期几为十进制数[0（星期日），6]。	
%W	一年中的星期数（星期一作为一周的第一天）作为十进制数[00,53]。第一个星期一之前的新年的所有日子都被认为是在第0周。	（3）
%x	区域设置的适当日期表示。	
%X	区域设置适当的时间表示。	
%y	没有世纪作为十进制数的年份[00,99]。	
%Y	世纪作为十进制数字。	
%z	时区偏移量指示格式为+ HHMM或-HHMM的UTC / GMT的正负时差，其中H表示十进制小时数字，M表示十进制分钟数[-23：59，+23：59]。	
%Z	时区名称（如果不存在时区，则不显示字符）。	
%%	字面'%'字符。	

笔记：

1. 与该strptime()函数一起使用时，%p如果该%I指令用于解析小时，则该指令仅影响输出小时字段。
2. 范围确实是0到61; 值60在表示闰秒的时间戳中有效，并且61由于历史原因支持该值。
3. 当与使用strptime()功能，%U并且%W指定了一周，一年中的一天，在计算仅使用。

这里是一个例子，日期的格式与。中指定的格式兼容 **RFC 2822** Internet电子邮件标准。[1]

```
>>> from time import gmtime, strftime
>>> strftime("%a, %d %b %Y %H:%M:%S +0000", gmtime())
'Thu, 28 Jun 2001 14:17:15 +0000'
```

在某些平台上可能支持其他指令，但只有这里列出的指令具有ANSI C标准化的含义。要查看平台上支持的全套格式代码，请参阅*strftime（3）*文档。

在某些平台上，可选的字段宽度和精度规格可以按照'%'以下顺序紧跟在指令的首字母之后; 这也是不便携的。字段宽度通常为2，除了%j它是3的位置。

time.**strptime**(*string* [, *format*])

根据格式解析表示时间的字符串。返回值是struct_time由gmtime()or返回的localtime()。

该*格式*参数使用相同的指令那些由使用 strftime()；它默认为匹配返回的格式。如果*字符串*不能根据*格式*进行分析，或者在解析后有多余的数据，则会引发。当无法推断出更精确的值时，用于填写缺失数据的默认值为。这两个*字符串*和*格式*必须是字符串。"%a %b %d %H:%M:%S %Y"ctime()ValueError(1900, 1, 1, 0, 0, 0, 0, 1, -1)

例如：

```
>>> import time
>>> time.strptime("30 Nov 00", "%d %b %y")
time.struct_time(tm_year=2000, tm_mon=11, tm_mday=30, tm_hour=0, tm_min=0,
                 tm_sec=0, tm_wday=3, tm_yday=335, tm_isdst=-1)
```

对%Z指令的支持是基于包含的值tzname以及是否daylight为真。因此，除了识别始终知道的UTC和GMT（并且被认为是非夏时制时区）之外，它是平台特定的。

仅支持文档中指定的指令。由于 strftime()每个平台都可以实现，因此有时可以提供比列出的指令更多的指令。但是strptime()独立于任何平台，因此不一定支持所有没有记录为支持的指令。

*类*time.struct_time

类型的时间序列值的返回通过gmtime()， localtime()和strptime()。它是具有命名元组接口的对象：值可以通过索引和属性名称来访问。以下值存在：

指数	属性	值
0	tm_year	（例如，1993年）
1	tm_mon	范围[1,12]
2	tm_mday	范围[1，31]
3	tm_hour	范围[0,23]
4	tm_min	范围[0，59]
五	tm_sec	范围[0，61]; 参见**（2）**的 strftime()描述
6	tm_wday	范围[0，6]，星期一为0
7	tm_yday	范围[1,366]
8	tm_isdst	0,1或-1; 见下文
N / A	tm_zone	时区名称的缩写
N / A	tm_gmtoff	以秒为单位偏移UTC的东部

请注意，与C结构不同，月份值的范围是[1,12]，而不是[0,11]。

在呼叫中mktime()，tm_isdst夏令时有效时可以设置为1，否则为0。值为-1表示这是未知的，并且通常会导致正确的状态被填充。

当长度不正确的元组被传递给期望a的函数 struct_time或具有错误类型的元素时，TypeError会引发a。

time.**time**( )

自纪元起以秒为单位返回浮点数的时间。时代的具体日期和闰秒的处理与 平台有关。在Windows和大多数Unix系统上，时间是1970年1月1日，00:00:00（UTC），闰秒不计入自纪元以来的秒数。这通常被称为 Unix时间。要了解特定平台上的时代，请看 gmtime(0)。

请注意，即使时间总是以浮点数形式返回，但并非所有系统的时间精度都比1秒高。虽然此函数通常返回非递减值，但如果系统时钟已在两次调用之间回退，则它可以返回比先前调用更低的值。

返回的数字time()可以通过将其传递到gmtime()函数或在本地时间传递给函数 来转换为UTC中更常见的时间格式（即年，月，日，小时等） localtime()。在这两种情况下struct_time都会返回一个 对象，从中可以将日历日期的组件作为属性进行访问。

time.**tzset**( )

重置库例程使用的时间转换规则。环境变量TZ 指定了这是如何完成的。

可用性：Unix。

> **注意:** 虽然在许多情况下，改变了 TZ环境变量可能会影响函数的输出localtime()而不调用 tzset()，因此不应该依赖此行为。
> 该 TZ 环境变量不应该包含空格。

标准格式的 TZ 环境变量是（为清楚起见，添加了空白）：

```
std offset [dst [offset [,start[/time], end[/time]]]]
```

组件是：

std 和 dst
: 三个或更多的字母数字提供时区缩写。这些将会传播到time.tzname中

offset
: 偏移量的形式为：。这表示该值增加了当地时间到达UTC。如果前面有' - '，则时区在Prime Meridian的东边; 否则，它是西方的。如果在dst之后没有偏移，则夏季时间假定为比标准时间提前一个小时。 ± hh[:mm[:ss]]

start[/time], end[/time]
: 指示何时更改和返回DST。开始日期和结束日期的格式为以下之一：

  J*n*
  : 朱利安日*n*（1 <= *n* <= 365）。闰日不计算在内，所以在所有年份中，2月28日是第59天，3月1日是第60天。

  n
  : 基于零的Julian日（0 <= *n* <= 365）。闰日计算在内，可以参考2月29日。

  M*m*.*n*.*d*

的*d*“日天（0 <= *d* <= 6）周的*Ñ*的月*米*的一年（1 <= *Ñ* <= 5,1 <= *米* <= 12，其中每周5种手段“的最后*d*一天月*米*其可以在任一第四或第五周发生”）。1周是其中的第一个星期*d*“日天发生。第零天是星期天。

timeoffset除了不允许使用前导符号（'-'或'+'）以外，格式相同。如果没有给出时间，默认是02:00:00。

```
>>> os.environ['TZ'] = 'EST+05EDT,M4.1.0,M10.5.0'
>>> time.tzset()
>>> time.strftime('%X %x %Z')
'02:07:36 05/08/03 EDT'
>>> os.environ['TZ'] = 'AEST-10AEDT-11,M10.5.0,M3.5.0'
>>> time.tzset()
>>> time.strftime('%X %x %Z')
'16:08:12 05/08/03 AEST'
```

在许多Unix系统（包括* BSD，Linux，Solaris和Darwin）中，使用系统的zoneinfo（*tzfile（5）*）数据库来指定时区规则会更方便。为此，请设置 TZ环境变量转换为所需时区数据文件的路径，相对于系统'zoneinfo'时区数据库的根，通常位于/usr/share/zoneinfo。例如，'US/Eastern'，'Australia/Melbourne'，'Egypt'或'Europe/Amsterdam'。

```
>>> os.environ['TZ'] = 'US/Eastern'
>>> time.tzset()
>>> time.tzname
('EST', 'EDT')
>>> os.environ['TZ'] = 'Egypt'
>>> time.tzset()
>>> time.tzname
('EET', 'EEST')
```

16.3.2。时钟ID常量

这些常数是用来作为参数clock_getres()和 clock_gettime()。

time.CLOCK_HIGHRES

Solaris OS具有一个CLOCK_HIGHRES计时器，该计时器会尝试使用最佳硬件来源，并且可能会提供接近纳秒的分辨率。 CLOCK_HIGHRES是不可调节的高分辨率时钟。

可用性：Solaris。

3.3版本的新功能

time.CLOCK_MONOTONIC

无法设置的时钟，表示从某个未指定的起点开始的单调时间。

可用性：Unix。

3.3版本的新功能

time.CLOCK_MONOTONIC_RAW

与CLOCK_MONOTONIC基于硬件的原始时间类似，但可以访问不受NTP调整的时间。

可用性：Linux 2.6.28或更高版本。

3.3版本的新功能

time.CLOCK_PROCESS_CPUTIME_ID
来自CPU的高分辨率每进程定时器。

可用性：Unix。

3.3版本的新功能

time.CLOCK_THREAD_CPUTIME_ID
线程特定的CPU时钟。

可用性：Unix。

3.3版本的新功能

以下常量是唯一可以发送到的参数 clock_settime()。

time.CLOCK_REALTIME
系统范围的实时时钟。设置这个时钟需要适当的权限。

可用性：Unix。

3.3版本的新功能

16.3.3。时区常量

time.altzone
本地DST时区的偏移量，以UTC为单位的秒数（如果已定义）。如果当地DST时区在UTC以东（如西欧，包括英国），则这是负面的。只有daylight在非零时才使用。见下面的注释。

time.daylight
如果定义了DST时区，则为非零值。见下面的注释。

time.timezone
本地（非DST）时区的偏移量，以UTC为单位的秒数（西欧大部分地区为负值，美国为正值，英国为零）。见下面的注释。

time.tzname
两个字符串的元组：第一个是本地非DST时区的名称，第二个是本地DST时区的名称。如果没有定义DST时区，则不应使用第二个字符串。见下面的注释。

注意： 对于上述时区常数（altzone，daylight，timezone，和tzname），该值是由时区规则效果在模块加载时确定或最后一次tzset()被调用，并且可以是不正确的，在过去的时间。

建议使用tm_gmtoff和 tm_zone结果localtime()来获取时区信息。

也可以看看:

模 datetime
面向日期和时间的面向对象的接口。

模 locale
国际化服务。区域设置会影响许多格式说明的解释strftime()和strptime()。

模 calendar
一般日历相关功能。 timegm()是gmtime()这个模块的反面。

脚注

[1] 使用的%Z是现在已经过时，但%z展开成首选小时/分钟偏移不是所有ANSI C库支持逃生。此外，严格阅读原来的1982年**RFC 822**标准要求两位数的年份（%y而不是%Y），但实践在2000年之前移到了4位数的年份。之后，**RFC 822**变得过时了，4位数的年份首先被推荐**RFC 1123**，然后由其授权**RFC 2822**。

16.4。argparse- 用于命令行选项，参数和子命令的解析器

3.2版本中的新功能

源代码： Lib / argparse.py

该argparse模块可以轻松编写用户友好的命令行界面。该程序定义了它需要的参数，argparse 并将找出如何解析这些参数sys.argv。该argparse 模块还会自动生成帮助和用法消息，并在用户给出程序无效参数时发出错误。

教程

此页面包含API参考信息。有关Python命令行解析的更简单介绍，请参阅 argparse 教程。

16.4.1。示例

下面的代码是一个Python程序，它接受一个整数列表并产生总和或最大值：

```
import argparse

parser = argparse.ArgumentParser(description='Process some integers.')
parser.add_argument('integers', metavar='N', type=int, nargs='+',
                    help='an integer for the accumulator')
parser.add_argument('--sum', dest='accumulate', action='store_const',
                    const=sum, default=max,
                    help='sum the integers (default: find the max)')

args = parser.parse_args()
print(args.accumulate(args.integers))
```

假设上面的Python代码保存在一个名为的文件中prog.py，它可以在命令行运行并提供有用的帮助信息：

```
$ python prog.py -h
usage: prog.py [-h] [--sum] N [N ...]

Process some integers.

positional arguments:
 N           an integer for the accumulator

optional arguments:
 -h, --help  show this help message and exit
 --sum       sum the integers (default: find the max)
```

当使用适当的参数运行时，它会输出命令行整数的总和或最大值：

```
$ python prog.py 1 2 3 4
4

$ python prog.py 1 2 3 4 --sum
10
```

如果传入无效参数，它将发出错误：

```
$ python prog.py a b c
usage: prog.py [-h] [--sum] N [N ...]
```

```
prog.py: error: argument N: invalid int value: 'a'
```

以下部分将引导您完成此示例。

16.4.1.1。创建解析器

使用的第一步argparse是创建一个 ArgumentParser对象：

```
>>> parser = argparse.ArgumentParser(description='Process some integers.')
```

该ArgumentParser对象将包含将命令行解析为Python数据类型所需的全部信息。

16.4.1.2。添加参数

ArgumentParser通过调用add_argument()方法来完成关于程序参数的信息填充。通常，这些调用将告诉ArgumentParser如何在命令行上取出字符串并将它们转换为对象。该信息在parse_args()被调用时被存储和使用。例如：

```
>>> parser.add_argument('integers', metavar='N', type=int, nargs='+',
...                     help='an integer for the accumulator')
>>> parser.add_argument('--sum', dest='accumulate', action='store_const',
...                     const=sum, default=max,
...                     help='sum the integers (default: find the max)')
```

稍后，调用parse_args()将返回一个具有两个属性的对象，integers并且accumulate。该integers属性将是一个或多个整数的列表，如果在命令行中指定了该accumulate属性sum()，则该属性将是该函数，如果该属性不是该属性，则该属性将是函数。--summax()

16.4.1.3。解析参数

ArgumentParser通过该parse_args()方法解析参数 。这将检查命令行，将每个参数转换为适当的类型，然后调用相应的操作。在大多数情况下，这意味着一个简单的Namespace对象将由从命令行解析出来的属性建立起来：

```
>>> parser.parse_args(['--sum', '7', '-1', '42'])
Namespace(accumulate=<built-in function sum>, integers=[7, -1, 42])
```

在一个脚本中，parse_args()通常会被调用，不带任何参数，并且ArgumentParser会自动从中确定命令行参数sys.argv。

16.4.2。ArgumentParser对象

class argparse.ArgumentParser（*prog = None*，*usage = None*，*description = None*，*epilog = None*，*parents = []*，*formatter_class = argparse.HelpFormatter*，*prefix_chars ='-'*，*fromfile_prefix_chars = None*，*argument_default = None*，*conflict_handler ='error'*，*add_help = True*，*allow_abbrev = True* ）

创建一个新的ArgumentParser对象。所有参数都应该作为关键字参数传递。下面的每个参数都有自己的更详细的描述，但总之它们是：

- PROG -程序的名称（默认：sys.argv[0]）
- 用法 - 描述程序使用情况的字符串（默认值：从添加到解析器的参数生成）
- 描述 - 在参数帮助之前显示的文本（默认值：无）

- epilog - 在参数帮助后显示的文本（默认值：无）
- 父母 - ArgumentParser也应包含其参数的对象列表
- formatter_class - 用于自定义帮助输出的类
- prefix_chars - 前缀可选参数的字符集（默认值：'-'）
- fromfile_prefix_chars -该组文件前缀字符从额外的参数应该读（默认值：None）
- argument_default -为参数的全局默认值（默认值：None）
- conflict_handler - 解决冲突选项的策略（通常不需要）
- add_help -添加-h/--help选项解析器（默认值：True）
- allow_abbrev - 允许长选项缩写，如果缩写是明确的。（默认值：True）

*在版本3.5中进行了更改：*添加了*allow_abbrev*参数。

以下部分描述如何使用每个这些。

16.4.2.1。编

默认情况下，ArgumentParser对象用于sys.argv[0]确定如何在帮助消息中显示程序的名称。这个默认值几乎总是可取的，因为它会使帮助消息与命令行上的程序调用方式相匹配。例如，考虑myprogram.py使用以下代码命名的文件 ：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--foo', help='foo help')
args = parser.parse_args()
```

该程序的帮助将显示myprogram.py为程序名称（不管程序从何处被调用）：

```
$ python myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO   foo help
$ cd ..
$ python subdir/myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO   foo help
```

要更改此默认行为，可以使用以下prog=参数提供另一个值 ArgumentParser：

```
>>> parser = argparse.ArgumentParser(prog='myprogram')
>>> parser.print_help()
usage: myprogram [-h]

optional arguments:
 -h, --help  show this help message and exit
```

请注意，程序名称（无论从参数确定sys.argv[0]还是从 prog=参数确定）都可用于使用%(prog)s格式说明符来帮助消息。

```
>>> parser = argparse.ArgumentParser(prog='myprogram')
>>> parser.add_argument('--foo', help='foo of the %(prog)s program')
>>> parser.print_help()
usage: myprogram [-h] [--foo FOO]
```

```
optional arguments:
 -h, --help  show this help message and exit
 --foo FOO   foo of the myprogram program
```

16.4.2.2。用法

默认情况下，ArgumentParser根据它包含的参数计算使用情况消息：

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', nargs='?', help='foo help')
>>> parser.add_argument('bar', nargs='+', help='bar help')
>>> parser.print_help()
usage: PROG [-h] [--foo [FOO]] bar [bar ...]

positional arguments:
 bar          bar help

optional arguments:
 -h, --help   show this help message and exit
 --foo [FOO]  foo help
```

默认消息可以用usage=关键字参数覆盖：

```
>>> parser = argparse.ArgumentParser(prog='PROG', usage='%(prog)s [options]')
>>> parser.add_argument('--foo', nargs='?', help='foo help')
>>> parser.add_argument('bar', nargs='+', help='bar help')
>>> parser.print_help()
usage: PROG [options]

positional arguments:
 bar          bar help

optional arguments:
 -h, --help   show this help message and exit
 --foo [FOO]  foo help
```

该%(prog)s格式说明可填写程序名称您的使用情况的信息。

16.4.2.3。描述

大多数对ArgumentParser构造函数的调用将使用 description=关键字参数。这个观点简要介绍了程序的功能和工作原理。在帮助消息中，说明显示在命令行用法字符串和各种参数的帮助消息之间：

```
>>> parser = argparse.ArgumentParser(description='A foo that bars')
>>> parser.print_help()
usage: argparse.py [-h]

A foo that bars

optional arguments:
 -h, --help  show this help message and exit
```

默认情况下，描述将被行包裹，以便它符合给定的空间。要改变这种行为，请参阅formatter_class参数。

16.4.2.4。结语

一些程序喜欢在参数描述之后显示程序的附加描述。这样的文本可以使用epilog= 参数来指定ArgumentParser：

```
>>> parser = argparse.ArgumentParser(
...     description='A foo that bars',
...     epilog="And that's how you'd foo a bar")
>>> parser.print_help()
usage: argparse.py [-h]

A foo that bars

optional arguments:
 -h, --help  show this help message and exit

And that's how you'd foo a bar
```

与描述参数一样，epilog=文本在默认情况下是行包装的，但此行为可以使用formatter_class 参数进行调整ArgumentParser。

16.4.2.5。父母

有时，几个解析器共享一组通用参数。 可以使用具有所有共享参数并传递给parents=参数的单个解析器，而不是重复这些参数的定义ArgumentParser。该parents=参数获取ArgumentParser 对象列表，收集所有位置和可选操作，并将这些操作添加到ArgumentParser正在构建的对象中：

```
>>> parent_parser = argparse.ArgumentParser(add_help=False)
>>> parent_parser.add_argument('--parent', type=int)

>>> foo_parser = argparse.ArgumentParser(parents=[parent_parser])
>>> foo_parser.add_argument('foo')
>>> foo_parser.parse_args(['--parent', '2', 'XXX'])
Namespace(foo='XXX', parent=2)

>>> bar_parser = argparse.ArgumentParser(parents=[parent_parser])
>>> bar_parser.add_argument('--bar')
>>> bar_parser.parse_args(['--bar', 'YYY'])
Namespace(bar='YYY', parent=None)
```

请注意，大多数父解析器将指定add_help=False。否则， ArgumentParser将会看到两个-h/--help选项（一个在父项中，一个在子项中）并引发错误。

注意： 在传递它们之前，您必须完全初始化解析器parents=。如果在子解析器之后更改父解析器，那些更改将不会反映到子解析器中。

16.4.2.6。formatter_class

ArgumentParser对象允许通过指定备用格式类来自定义帮助格式。目前，有四个这样的类：

*类*argparse. RawDescriptionHelpFormatter
*类*argparse. RawTextHelpFormatter
*类*argparse. ArgumentDefaultsHelpFormatter
*类*argparse. MetavarTypeHelpFormatter

RawDescriptionHelpFormatter并RawTextHelpFormatter更好地控制文本描述的显示方式。默认情况下，ArgumentParser对象将命令行帮助消息中的描述和 结语文本换行：

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     description='''this description
...         was indented weird
...             but that is okay''',
...     epilog='''
...             likewise for this epilog whose whitespace will
...         be cleaned up and whose words will be wrapped
...         across a couple lines''')
>>> parser.print_help()
usage: PROG [-h]

this description was indented weird but that is okay

optional arguments:
 -h, --help  show this help message and exit

likewise for this epilog whose whitespace will be cleaned up and whose words
will be wrapped across a couple lines
```

RawDescriptionHelpFormatter作为formatter_class= 指示描述和epilog已经被正确格式化并且不应该被行包裹传递：

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.RawDescriptionHelpFormatter,
...     description=textwrap.dedent('''\
...         Please do not mess up this text!
...         --------------------------------
...             I have indented it
...             exactly the way
...             I want it
...         '''))
>>> parser.print_help()
usage: PROG [-h]

Please do not mess up this text!
--------------------------------
   I have indented it
   exactly the way
   I want it

optional arguments:
 -h, --help  show this help message and exit
```

RawTextHelpFormatter为各种帮助文本保留空格，包括参数说明。但是，多个新行被替换为一个。如果您希望保留多个空行，请在换行符之间添加空格。

ArgumentDefaultsHelpFormatter 自动为每个参数帮助消息添加有关默认值的信息：

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.ArgumentDefaultsHelpFormatter)
>>> parser.add_argument('--foo', type=int, default=42, help='FOO!')
>>> parser.add_argument('bar', nargs='*', default=[1, 2, 3], help='BAR!')
>>> parser.print_help()
usage: PROG [-h] [--foo FOO] [bar [bar ...]]

positional arguments:
 bar         BAR! (default: [1, 2, 3])
```

```
optional arguments:
 -h, --help  show this help message and exit
 --foo FOO   FOO! (default: 42)
```

MetavarTypeHelpFormatter为每个参数使用type参数的名称作为其值的显示名称（而不是 像常规格式化程序那样使用dest）：

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.MetavarTypeHelpFormatter)
>>> parser.add_argument('--foo', type=int)
>>> parser.add_argument('bar', type=float)
>>> parser.print_help()
usage: PROG [-h] [--foo int] float

positional arguments:
  float

optional arguments:
  -h, --help  show this help message and exit
  --foo int
```

16.4.2.7。prefix_chars

大多数命令行选项将-用作前缀，例如-f/--foo。需要支持不同或额外前缀字符的解析器，例如像+for 这样的选项/foo，可以使用prefix_chars=ArgumentParser构造函数的参数来指定它们：

```
>>> parser = argparse.ArgumentParser(prog='PROG', prefix_chars='-+')
>>> parser.add_argument('+f')
>>> parser.add_argument('++bar')
>>> parser.parse_args('+f X ++bar Y'.split())
Namespace(bar='Y', f='X')
```

该prefix_chars=参数默认为'-'。提供一组不包含的字符-将导致-f/--foo选项不被允许。

16.4.2.8。fromfile_prefix_chars

有时候，例如当处理特别长的参数列表时，将参数列表保存在文件中而不是在命令行输入它可能是有意义的。如果将fromfile_prefix_chars=参数赋予 ArgumentParser构造函数，那么以任何指定字符开头的参数将被视为文件，并将被它们包含的参数替换。例如：

```
>>> with open('args.txt', 'w') as fp:
...     fp.write('-f\nbar')
>>> parser = argparse.ArgumentParser(fromfile_prefix_chars='@')
>>> parser.add_argument('-f')
>>> parser.parse_args(['-f', 'foo', '@args.txt'])
Namespace(f='bar')
```

从文件中读取的参数必须默认为每行一个（但也可参见 convert_arg_line_to_args()），并且将它们视为与命令行中原始文件引用参数位于同一位置。所以在上面的例子中，表达式 被认为等同于表达式。['-f', 'foo', '@args.txt']['-f', 'foo', '-f', 'bar']

该fromfile_prefix_chars=参数默认为None，这意味着争论就永远不会文件引用处理。

16.4.2.9。argument_default

一般来说，参数默认值是通过传递一个默认值来指定的，add_argument()或者通过调用set_defaults()具有一组特定名称/值对的 方法来指定参数默认 值。然而，有时候，为参数指定单个解析器范围的默认值可能很有用。这可以通过传递 argument_default=关键字参数来完成ArgumentParser。例如，要全局禁止parse_args() 呼叫中的属性创建，我们提供argument_default=SUPPRESS：

```
>>> parser = argparse.ArgumentParser(argument_default=argparse.SUPPRESS)
>>> parser.add_argument('--foo')
>>> parser.add_argument('bar', nargs='?')
>>> parser.parse_args(['--foo', '1', 'BAR'])
Namespace(bar='BAR', foo='1')
>>> parser.parse_args([])
Namespace()
```

16.4.2.10。allow_abbrev

通常，当您将参数列表传递给an的 parse_args()方法时ArgumentParser ，它会识别长选项的缩写。

设置allow_abbrev为False以下功能可以禁用此功能：

```
>>> parser = argparse.ArgumentParser(prog='PROG', allow_abbrev=False)
>>> parser.add_argument('--foobar', action='store_true')
>>> parser.add_argument('--foonley', action='store_false')
>>> parser.parse_args(['--foon'])
usage: PROG [-h] [--foobar] [--foonley]
PROG: error: unrecognized arguments: --foon
```

3.5版本中的新功能。

16.4.2.11。conflict_handler

ArgumentParser对象不允许具有相同选项字符串的两个操作。默认情况下，ArgumentParser如果尝试使用已在使用的选项字符串创建参数，则会引发异常：

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-f', '--foo', help='old foo help')
>>> parser.add_argument('--foo', help='new foo help')
Traceback (most recent call last):
 ..
ArgumentError: argument --foo: conflicting option string(s): --foo
```

有时（例如，在使用父母时），使用相同的选项字符串覆盖任何较旧的参数可能会很有用。为了获得这种行为，该值 'resolve' 可以提供给conflict_handler=参数 ArgumentParser：

```
>>> parser = argparse.ArgumentParser(prog='PROG', conflict_handler='resolve')
>>> parser.add_argument('-f', '--foo', help='old foo help')
>>> parser.add_argument('--foo', help='new foo help')
>>> parser.print_help()
usage: PROG [-h] [-f FOO] [--foo FOO]

optional arguments:
 -h, --help  show this help message and exit
 -f FOO      old foo help
 --foo FOO   new foo help
```

请注意，ArgumentParser如果对象的所有选项字符串都被覆盖，对象只会删除一个操作。因此，在上面的示例中，旧-f/--foo 操作保留为-f操作，因为只有--foo选项字符串被覆盖。

16.4.2.12。add_help

默认情况下，ArgumentParser对象添加一个选项，该选项只显示解析器的帮助消息。例如，考虑一个名为myprogram.py包含以下代码的文件 ：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--foo', help='foo help')
args = parser.parse_args()
```

如果-h或--help在命令行中提供，则会打印参数帮助器的帮助：

```
$ python myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO   foo help
```

偶尔，禁用此帮助选项可能会有用。这可以通过False作为add_help=参数传递给 ArgumentParser ：

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> parser.add_argument('--foo', help='foo help')
>>> parser.print_help()
usage: PROG [--foo FOO]

optional arguments:
 --foo FOO  foo help
```

帮助选项通常是-h/--help。这是一个例外，如果prefix_chars=指定并且不包括- ，在这种情况下-h并且--help不是有效的选项。在这种情况下，第一个字符prefix_chars用于作为帮助选项的前缀：

```
>>> parser = argparse.ArgumentParser(prog='PROG', prefix_chars='+/')
>>> parser.print_help()
usage: PROG [+h]

optional arguments:
  +h, ++help  show this help message and exit
```

16.4.3。add_argument（ ）方法

ArgumentParser.add_argument（*name或flags ...* [, *action*] [, *nargs*] [, *const*] [, *default*] [, *type*] [, *choices*] [, *required*] [, *help*] [, *metavar*] [, *dest*] ）

定义如何解析单个命令行参数。下面的每个参数都有自己的更详细的描述，但总之它们是：

- 名称或标志 - 名称或选项字符串列表，例如foo 或。 -f, --foo
- action - 在命令行中遇到此参数时要采取的基本操作类型。
- nargs - 应该使用的命令行参数的数量。
- const - 一些动作和nargs选择所需的常量值。
- default - 如果参数在命令行中不存在，则生成的值。
- type - 命令行参数应该转换的类型。
- 选项 - 一个允许参数值的容器。
- 必需 - 是否可以省略命令行选项（仅限可选项）。
- 帮助 - 对参数做了什么的简要描述。
- metavar - 用法消息中参数的名称。

- dest - 要添加到返回对象的属性的名称 parse_args()。

以下部分描述如何使用每个这些。

16.4.3.1。名称或标志

该add_argument()方法必须知道是否需要一个可选参数（如-for或）--foo，或位置参数（如文件名列表）。add_argument()因此传递给它的第一个参数 必须是一系列标志或一个简单的参数名称。例如，可以创建一个可选参数，如下所示：

```
>>> parser.add_argument('-f', '--foo')
```

而一个位置参数可以创建如下：

```
>>> parser.add_argument('bar')
```

当parse_args()被调用时，可选参数将由-前缀标识，其余参数将被假定为位置：

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-f', '--foo')
>>> parser.add_argument('bar')
>>> parser.parse_args(['BAR'])
Namespace(bar='BAR', foo=None)
>>> parser.parse_args(['BAR', '--foo', 'FOO'])
Namespace(bar='BAR', foo='FOO')
>>> parser.parse_args(['--foo', 'FOO'])
usage: PROG [-h] [-f FOO] bar
PROG: error: the following arguments are required: bar
```

16.4.3.2。行动

ArgumentParser对象将命令行参数与操作相关联。这些操作可以完成任何与它们相关的命令行参数的任何操作，尽管大多数操作只是为返回的对象添加一个属性 parse_args()。该action关键字参数指定的命令行参数应该如何处理。提供的操作是：

- 'store' - 这只是存储参数的值。这是默认操作。例如：

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.parse_args('--foo 1'.split())
Namespace(foo='1')
```

- 'store_const' - 这存储由const关键字参数指定的值。该'store_const'操作最常用于指定某种标志的可选参数。例如：

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_const', const=42)
>>> parser.parse_args(['--foo'])
Namespace(foo=42)
```

- 'store_true'和'store_false' -这些是特殊情况下 'store_const'用于存储值True和False 分别。此外，他们创造的默认值False，并 True分别。例如：

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_true')
```

```
>>> parser.add_argument('--bar', action='store_false')
>>> parser.add_argument('--baz', action='store_false')
>>> parser.parse_args('--foo --bar'.split())
Namespace(foo=True, bar=False, baz=True)
```

- 'append' - 这存储一个列表，并将每个参数值附加到列表中。这对于允许多次指定选项很有用。用法示例：

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='append')
>>> parser.parse_args('--foo 1 --foo 2'.split())
Namespace(foo=['1', '2'])
```

- 'append_const' - 存储一个列表，并将const关键字参数指定的值附加到列表中。（请注意，const关键字参数默认为None。）'append_const' 当多个参数需要将常量存储到同一列表时，该操作通常很有用。例如：

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--str', dest='types', action='append_const', const=str)
>>> parser.add_argument('--int', dest='types', action='append_const', const=int)
>>> parser.parse_args('--str --int'.split())
Namespace(types=[<class 'str'>, <class 'int'>])
```

- 'count' - 这会计算关键字参数发生的次数。例如，这对增加冗长级别很有用：

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--verbose', '-v', action='count')
>>> parser.parse_args(['-vvv'])
Namespace(verbose=3)
```

- 'help' - 这将显示当前解析器中所有选项的完整帮助消息，然后退出。默认情况下，帮助操作会自动添加到解析器中。请参阅ArgumentParser有关如何创建输出的详细信息。

- 'version' - 这需要调用中的version=关键字参数 add_argument()，并打印版本信息并在调用时退出：

```
>>> import argparse
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--version', action='version', version='%(prog)s 2.0')
>>> parser.parse_args(['--version'])
PROG 2.0
```

您也可以通过传递一个Action子类或其他实现相同接口的对象来指定一个任意的动作。推荐的方法是扩展Action，覆盖__call__方法和可选的__init__方法。

自定义操作的示例：

```
>>> class FooAction(argparse.Action):
...     def __init__(self, option_strings, dest, nargs=None, **kwargs):
...         if nargs is not None:
...             raise ValueError("nargs not allowed")
...         super(FooAction, self).__init__(option_strings, dest, **kwargs)
...     def __call__(self, parser, namespace, values, option_string=None):
...         print('%r %r %r' % (namespace, values, option_string))
...         setattr(namespace, self.dest, values)
...
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action=FooAction)
>>> parser.add_argument('bar', action=FooAction)
```

```
>>> args = parser.parse_args('1 --foo 2'.split())
Namespace(bar=None, foo=None) '1' None
Namespace(bar='1', foo=None) '2' '--foo'
>>> args
Namespace(bar='1', foo='2')
```

有关更多详情，请参阅Action。

16.4.3.3。NARGS

ArgumentParser对象通常会将单个命令行参数与要执行的单个操作相关联。的nargs关键字参数与单个动作不同数量的命令行参数。支持的值是：

- N（一个整数）。 N命令行中的参数将汇集到一个列表中。例如：

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo', nargs=2)
  >>> parser.add_argument('bar', nargs=1)
  >>> parser.parse_args('c --foo a b'.split())
  Namespace(bar=['c'], foo=['a', 'b'])
  ```

  请注意，nargs=1生成一个项目的列表。这与项目自行生成的默认值不同。

- '?'。如有可能，将从命令行中消耗一个参数，并将其作为单个项目生成。如果没有命令行参数，则会生成默认值 。请注意，对于可选参数，还有一个额外的情况 - 选项字符串存在，但后面没有命令行参数。在这种情况下，const的值将被生成。一些例子来说明这一点：

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo', nargs='?', const='c', default='d')
  >>> parser.add_argument('bar', nargs='?', default='d')
  >>> parser.parse_args(['XX', '--foo', 'YY'])
  Namespace(bar='XX', foo='YY')
  >>> parser.parse_args(['XX', '--foo'])
  Namespace(bar='XX', foo='c')
  >>> parser.parse_args([])
  Namespace(bar='d', foo='d')
  ```

  更常见的用途之一nargs='?'是允许可选的输入和输出文件：

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('infile', nargs='?', type=argparse.FileType('r'),
  ...                     default=sys.stdin)
  >>> parser.add_argument('outfile', nargs='?', type=argparse.FileType('w'),
  ...                     default=sys.stdout)
  >>> parser.parse_args(['input.txt', 'output.txt'])
  Namespace(infile=<_io.TextIOWrapper name='input.txt' encoding='UTF-8'>,
            outfile=<_io.TextIOWrapper name='output.txt' encoding='UTF-8'>)
  >>> parser.parse_args([])
  Namespace(infile=<_io.TextIOWrapper name='<stdin>' encoding='UTF-8'>,
            outfile=<_io.TextIOWrapper name='<stdout>' encoding='UTF-8'>)
  ```

- '*'。所有存在的命令行参数都被收集到一个列表中。请注意，具有多个位置参数通常没有多大意义nargs='*'，但可以使用多个可选参数nargs='*'。例如：

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo', nargs='*')
  >>> parser.add_argument('--bar', nargs='*')
  >>> parser.add_argument('baz', nargs='*')
  ```

```
>>> parser.parse_args('a b --foo x y --bar 1 2'.split())
Namespace(bar=['1', '2'], baz=['a', 'b'], foo=['x', 'y'])
```

- '+'。就像'*'现在所有的命令行参数都被收集到一个列表中。此外，如果没有至少一个命令行参数，则会生成错误消息。例如：

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', nargs='+')
>>> parser.parse_args(['a', 'b'])
Namespace(foo=['a', 'b'])
>>> parser.parse_args([])
usage: PROG [-h] foo [foo ...]
PROG: error: the following arguments are required: foo
```

- argparse.REMAINDER。所有其余的命令行参数都被收集到一个列表中。这对于派发到其他命令行实用程序的命令行实用程序通常很有用：

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo')
>>> parser.add_argument('command')
>>> parser.add_argument('args', nargs=argparse.REMAINDER)
>>> print(parser.parse_args('--foo B cmd --arg1 XX ZZ'.split()))
Namespace(args=['--arg1', 'XX', 'ZZ'], command='cmd', foo='B')
```

如果nargs没有提供关键字参数，则消耗的参数数量由动作决定。通常这意味着一个命令行参数将被消耗，并且将生成单个项目（不是列表）。

16.4.3.4。常量

该const的参数add_argument()是用来装未在命令行中读取，但所需要的各种常数值ArgumentParser的动作。它最常见的两种用途是：

- 何时add_argument()用action='store_const'or 调用 action='append_const'。这些操作将该const值添加 到返回的对象的某个属性中 parse_args()。有关示例，请参阅操作说明。
- 何时add_argument()用选项字符串（如-for --foo）和nargs='?'。这会创建一个可选参数，后面跟零个或一个命令行参数。在解析命令行时，如果选项字符串遇到后面没有命令行参数，const则会取而代之的是值。有关示例，请参阅nargs说明。

使用'store_const'和'append_const'操作时，const 必须给出关键字参数。对于其他操作，它默认为None。

16.4.3.5。默认

所有可选参数和一些位置参数可以在命令行中省略。如果命令行参数不存在，那么其值缺省为的default关键字参数 add_argument()将None指定应使用的值。对于可选参数，default当选项字符串不存在于命令行时，将使用该值：

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=42)
>>> parser.parse_args(['--foo', '2'])
Namespace(foo='2')
>>> parser.parse_args([])
Namespace(foo=42)
```

如果该default值是一个字符串，则解析器会将该值解析为一个命令行参数。特别是， 在设置返回值的属性之前，解析器应用任何类型转换参数（如果提供的话） Namespace。否则，解析器按原样使用该值：

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--length', default='10', type=int)
>>> parser.add_argument('--width', default=10.5, type=int)
>>> parser.parse_args()
Namespace(length=10, width=10.5)
```

对于nargs等于?or的位置参数，当没有命令行参数存在时使用*该default值：

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', nargs='?', default=42)
>>> parser.parse_args(['a'])
Namespace(foo='a')
>>> parser.parse_args([])
Namespace(foo=42)
```

default=argparse.SUPPRESS如果命令行参数不存在，则提供不导致添加属性的原因：

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=argparse.SUPPRESS)
>>> parser.parse_args([])
Namespace()
>>> parser.parse_args(['--foo', '1'])
Namespace(foo='1')
```

16.4.3.6。键入

默认情况下，ArgumentParser对象以简单的字符串读取命令行参数。但是，通常命令行字符串应该被解释为另一种类型，如a float或int。所述 type的关键字参数add_argument()允许执行任何必要的类型检查和类型转换。常见的内置类型和函数可以直接用作type参数的值：

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', type=int)
>>> parser.add_argument('bar', type=open)
>>> parser.parse_args('2 temp.txt'.split())
Namespace(bar=<_io.TextIOWrapper name='temp.txt' encoding='UTF-8'>, foo=2)
```

有关何时将参数应用于默认参数的信息，请参阅default关键字参数 部分type。

为了方便使用多种类型的文件时，argparse模块提供工厂FILETYPE这需要的mode= ， bufsize= ， encoding=和 errors=对的参数open()功能。例如， FileType('w')可以用来创建一个可写文件：

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('bar', type=argparse.FileType('w'))
>>> parser.parse_args(['out.txt'])
Namespace(bar=<_io.TextIOWrapper name='out.txt' encoding='UTF-8'>)
```

type= 可以采用任何可以调用单个字符串参数的可调用方法，并返回转换后的值：

```
>>> def perfect_square(string):
...     value = int(string)
...     sqrt = math.sqrt(value)
...     if sqrt != int(sqrt):
...         msg = "%r is not a perfect square" % string
...         raise argparse.ArgumentTypeError(msg)
...     return value
...
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', type=perfect_square)
```

```
>>> parser.parse_args(['9'])
Namespace(foo=9)
>>> parser.parse_args(['7'])
usage: PROG [-h] foo
PROG: error: argument foo: '7' is not a perfect square
```

该选择关键字参数可以是用于类型检查，简单地核对值的范围更方便：

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', type=int, choices=range(5, 10))
>>> parser.parse_args(['7'])
Namespace(foo=7)
>>> parser.parse_args(['11'])
usage: PROG [-h] {5,6,7,8,9}
PROG: error: argument foo: invalid choice: 11 (choose from 5, 6, 7, 8, 9)
```

请参阅选项部分了解更多详情。

16.4.3.7。选择

一些命令行参数应该从一组受限制的值中选择。这些可以通过将容器对象作为*选择*关键字参数传递给 add_argument()。在解析命令行时，将检查参数值，如果参数不是可接受值之一，则会显示错误消息：

```
>>> parser = argparse.ArgumentParser(prog='game.py')
>>> parser.add_argument('move', choices=['rock', 'paper', 'scissors'])
>>> parser.parse_args(['rock'])
Namespace(move='rock')
>>> parser.parse_args(['fire'])
usage: game.py [-h] {rock,paper,scissors}
game.py: error: argument move: invalid choice: 'fire' (choose from 'rock',
'paper', 'scissors')
```

请注意，在 执行任何类型转换后，将检查包含在*选择*容器中的内容，以便*选择* 容器中的对象类型应与指定的类型匹配：

```
>>> parser = argparse.ArgumentParser(prog='doors.py')
>>> parser.add_argument('door', type=int, choices=range(1, 4))
>>> print(parser.parse_args(['3']))
Namespace(door=3)
>>> parser.parse_args(['4'])
usage: doors.py [-h] {1,2,3}
doors.py: error: argument door: invalid choice: 4 (choose from 1, 2, 3)
```

任何支持in操作符的对象都可以作为*选择* 值传递，因此所有dict对象，set对象，自定义容器等都是受支持的。

16.4.3.8。需要

通常情况下，argparse模块假定标志像-f和--bar 表示*可选*参数，这些参数在命令行中总是可以省略。要创建*所需*的选项，True可以为required= 关键字参数指定add_argument()：

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', required=True)
>>> parser.parse_args(['--foo', 'BAR'])
Namespace(foo='BAR')
>>> parser.parse_args([])
```

```
usage: argparse.py [-h] [--foo FOO]
argparse.py: error: option --foo is required
```

如示例所示，如果某个选项被标记为required， parse_args()则会在命令行中不存在该选项时报告错误。

注意： 所需的选项通常被认为是不好的形式，因为用户期望 *选项*是 *可选的*，因此应尽可能避免。

16.4.3.9。帮助

该help值是一个包含参数简要描述的字符串。当用户请求帮助时（通常通过使用-h或--help通过命令行），这些help描述将与每个参数一起显示：

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('--foo', action='store_true',
...                     help='foo the bars before frobbling')
>>> parser.add_argument('bar', nargs='+',
...                     help='one of the bars to be frobbled')
>>> parser.parse_args(['-h'])
usage: frobble [-h] [--foo] bar [bar ...]

positional arguments:
 bar     one of the bars to be frobbled

optional arguments:
 -h, --help  show this help message and exit
 --foo   foo the bars before frobbling
```

这些help字符串可以包含各种格式说明符，以避免重复诸如程序名称或参数默认值之类的内容。可用符包括节目名称，%(prog)s和大多数关键字参数 add_argument()，如%(default)s，%(type)s等：

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('bar', nargs='?', type=int, default=42,
...                     help='the bar to %(prog)s (default: %(default)s)')
>>> parser.print_help()
usage: frobble [-h] [bar]

positional arguments:
 bar     the bar to frobble (default: 42)

optional arguments:
 -h, --help  show this help message and exit
```

由于帮助字符串支持%格式化，如果您希望文字%出现在帮助字符串中，您必须将其转义为%%。

argparse支持通过将某些选项的帮助条目设置help为argparse.SUPPRESS：

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('--foo', help=argparse.SUPPRESS)
>>> parser.print_help()
usage: frobble [-h]

optional arguments:
  -h, --help  show this help message and exit
```

16.4.3.10。metavar

当ArgumentParser生成帮助消息时，需要一些方法来引用每个预期的参数。默认情况下，ArgumentParser对象使用dest值作为每个对象的“名称”。默认情况下，对于位置参数操作，dest值直接使用，对于可选参数操作，dest值是大写的。所以，一个单独的位置参数 dest='bar' 将被称为bar。 --foo 单个命令行参数后面的单个可选参数将被称为FOO。一个例子：

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.add_argument('bar')
>>> parser.parse_args('X --foo Y'.split())
Namespace(bar='X', foo='Y')
>>> parser.print_help()
usage:  [-h] [--foo FOO] bar

positional arguments:
 bar

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO
```

另一个名称可以用以下方式指定metavar：

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', metavar='YYY')
>>> parser.add_argument('bar', metavar='XXX')
>>> parser.parse_args('X --foo Y'.split())
Namespace(bar='X', foo='Y')
>>> parser.print_help()
usage:  [-h] [--foo YYY] XXX

positional arguments:
 XXX

optional arguments:
 -h, --help  show this help message and exit
 --foo YYY
```

请注意，metavar只会更改*显示的*名称 - parse_args()对象上属性的名称仍由dest值确定。

不同的值nargs可能会导致metavar被多次使用。提供一个元组来metavar为每个参数指定一个不同的显示：

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x', nargs=2)
>>> parser.add_argument('--foo', nargs=2, metavar=('bar', 'baz'))
>>> parser.print_help()
usage: PROG [-h] [-x X X] [--foo bar baz]

optional arguments:
 -h, --help     show this help message and exit
 -x X X
 --foo bar baz
```

16.4.3.11。DEST

大多数ArgumentParser操作都会添加一些值作为返回对象的属性parse_args()。该属性的名称由dest关键字参数确定 add_argument()。对于位置参数操作， dest通常作为第一个参数提供给 add_argument()：

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('bar')
>>> parser.parse_args(['XXX'])
Namespace(bar='XXX')
```

对于可选的参数操作，dest通常从选项字符串推断值。 通过取第一个长选项字符串并剥离初始 字符串来ArgumentParser生成值。如果没有提供长选项字符串，则将通过剥离初始字符从第一个短选项字符串派生。任何内部字符都将转换为字符以确保该字符串是有效的属性名称。下面的例子说明了这种行为：dest--dest--_

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('-f', '--foo-bar', '--foo')
>>> parser.add_argument('-x', '-y')
>>> parser.parse_args('-f 1 -x 2'.split())
Namespace(foo_bar='1', x='2')
>>> parser.parse_args('--foo 1 -y 2'.split())
Namespace(foo_bar='1', x='2')
```

dest 允许提供自定义属性名称：

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', dest='bar')
>>> parser.parse_args('--foo XXX'.split())
Namespace(bar='XXX')
```

16.4.3.12。动作类

Action类实现Action API，这是一个可调用的函数，它返回一个可从命令行处理参数的可调用对象。任何遵循此API的对象都可以作为action参数传递给 add_argument()。

class argparse.Action（*option_strings*，*dest*，*nargs* = *None*，*const* = *None*，*default* = *None*，*type* = *None*，*choices* = *None*，*required* = *False*，*help* = *None*，*metavar* = *None* ）

ArgumentParser使用Action对象来表示解析来自命令行的一个或多个字符串的单个参数所需的信息。Action类必须接受两个位置参数以及传递给ArgumentParser.add_argument() 除action它自身以外的任何关键字参数。

Action 的实例（或任何可调用的 action 参数的返回值）应该具有定义的属性“dest”，“option_strings”，“default”，“type”，“required”，“help”等。确定这些属性的最简单方法是调用Action.__init__。

动作实例应该是可调用的，所以子类必须重载该 __call__方法，该方法应该接受四个参数：

- parser - 包含此操作的ArgumentParser对象。
- namespace- Namespace 将被返回的对象 parse_args()。大多数操作都使用此对象添加属性setattr()。
- values - 关联的命令行参数，应用了任何类型的转换。类型转换用type关键字参数 指定add_argument()。
- option_string - 用于调用此操作的选项字符串。该option_string参数是可选的，如果该动作与位置参数相关联，则该参数将不存在。

该__call__方法可能会执行任意操作，但通常会在namespace基于dest和上设置属性values。

16.4.4。parse_args（）方法

ArgumentParser.parse_args(*args = None*, *namespace = None*)

将参数字符串转换为对象并将它们分配为命名空间的属性。返回填充的命名空间。

先前的调用add_argument()确切地确定创建了哪些对象以及如何分配对象。有关add_argument()详细信息，请参阅文档。

- args - 要解析的字符串列表。默认取自 sys.argv。
- 命名空间 - 一个获取属性的对象。默认值是一个新的空 Namespace对象。

16.4.4.1。选项值语法

该parse_args()方法支持多种指定选项值的方式（如果需要的话）。在最简单的情况下，该选项及其值作为两个单独的参数传递：

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x')
>>> parser.add_argument('--foo')
>>> parser.parse_args(['-x', 'X'])
Namespace(foo=None, x='X')
>>> parser.parse_args(['--foo', 'FOO'])
Namespace(foo='FOO', x=None)
```

对于长选项（名称长于单个字符的选项），该选项和值也可以作为单个命令行参数传递，=用于将它们分开：

```
>>> parser.parse_args(['--foo=FOO'])
Namespace(foo='FOO', x=None)
```

对于短期选项（选项只有一个字符长），选项和它的值可以连接起来：

```
>>> parser.parse_args(['-xX'])
Namespace(foo=None, x='X')
```

-只要使用一个前缀，几个简短的选项就可以结合在一起，只要最后一个选项（或者它们中没有一个）需要一个值：

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x', action='store_true')
>>> parser.add_argument('-y', action='store_true')
>>> parser.add_argument('-z')
>>> parser.parse_args(['-xyzZ'])
Namespace(x=True, y=True, z='Z')
```

16.4.4.2。无效的参数

在解析命令行时，parse_args()检查各种错误，包括不明确的选项，无效的类型，无效的选项，错误的位置参数数量等。遇到此类错误时，它会退出并将错误与使用消息一起打印出来：

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', type=int)
>>> parser.add_argument('bar', nargs='?')

>>> # invalid type
>>> parser.parse_args(['--foo', 'spam'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: argument --foo: invalid int value: 'spam'
```

```
>>> # invalid option
>>> parser.parse_args(['--bar'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: no such option: --bar

>>> # wrong number of arguments
>>> parser.parse_args(['spam', 'badger'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: extra arguments found: badger
```

16.4.4.3。包含　参数-

parse_args()每当用户明确犯了错误，该方法都会尝试给出错误，但有些情况本质上是不明确的。例如，命令行参数-1可能是尝试指定选项或试图提供位置参数。这里的parse_args()方法很谨慎：位置参数可能只有-在它们看起来像负数时才会开始，解析器中没有任何选项看起来像负数：

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x')
>>> parser.add_argument('foo', nargs='?')

>>> # no negative number options, so -1 is a positional argument
>>> parser.parse_args(['-x', '-1'])
Namespace(foo=None, x='-1')

>>> # no negative number options, so -1 and -5 are positional arguments
>>> parser.parse_args(['-x', '-1', '-5'])
Namespace(foo='-5', x='-1')

>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-1', dest='one')
>>> parser.add_argument('foo', nargs='?')

>>> # negative number options present, so -1 is an option
>>> parser.parse_args(['-1', 'X'])
Namespace(foo=None, one='X')

>>> # negative number options present, so -2 is an option
>>> parser.parse_args(['-2'])
usage: PROG [-h] [-1 ONE] [foo]
PROG: error: no such option: -2

>>> # negative number options present, so both -1s are options
>>> parser.parse_args(['-1', '-1'])
usage: PROG [-h] [-1 ONE] [foo]
PROG: error: argument -1: expected one argument
```

如果您的位置参数必须以开头-且看起来不像负数，那么您可以插入伪参数'--'，该伪参数parse_args()指示之后的所有内容都是位置参数：

```
>>> parser.parse_args(['--', '-f'])
Namespace(foo='-f', one=None)
```

16.4.4.4。参数缩写（前缀匹配）

如果缩写是明确的（该前缀匹配唯一选项），默认情况下该parse_args()方法允许将长选项缩写为前缀：

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-bacon')
>>> parser.add_argument('-badger')
>>> parser.parse_args('-bac MMM'.split())
Namespace(bacon='MMM', badger=None)
>>> parser.parse_args('-bad WOOD'.split())
Namespace(bacon=None, badger='WOOD')
>>> parser.parse_args('-ba BA'.split())
usage: PROG [-h] [-bacon BACON] [-badger BADGER]
PROG: error: ambiguous option: -ba could match -badger, -bacon
```

对于可能产生多个选项的参数会产生错误。通过将allow_abbrev设置为可禁用此功能False。

16.4.4.5。超越sys.argv

有时候可能有一个ArgumentParser解析其他的参数sys.argv。这可以通过传递一个字符串列表来完成parse_args()。这对于在交互式提示下进行测试很有用：

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument(
...     'integers', metavar='int', type=int, choices=range(10),
...     nargs='+', help='an integer in the range 0..9')
>>> parser.add_argument(
...     '--sum', dest='accumulate', action='store_const', const=sum,
...     default=max, help='sum the integers (default: find the max)')
>>> parser.parse_args(['1', '2', '3', '4'])
Namespace(accumulate=<built-in function max>, integers=[1, 2, 3, 4])
>>> parser.parse_args(['1', '2', '3', '4', '--sum'])
Namespace(accumulate=<built-in function sum>, integers=[1, 2, 3, 4])
```

16.4.4.6。名字空间对象

*类*argparse.Namespace

简单的类默认使用parse_args()创建一个对象来保存属性并返回它。

这个类是故意简单的，只是一个object具有可读字符串表示的子类。如果您更喜欢使用类似字典的属性视图，则可以使用标准的Python语言vars()：

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> args = parser.parse_args(['--foo', 'BAR'])
>>> vars(args)
{'foo': 'BAR'}
```

ArgumentParser将属性赋值给已经存在的对象而不是新Namespace对象也可能是有用的。这可以通过指定namespace=关键字参数来实现：

```
>>> class C:
...     pass
...
>>> c = C()
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.parse_args(args=['--foo', 'BAR'], namespace=c)
>>> c.foo
'BAR'
```

16.4.5。其他工具

16.4.5.1。子命令

ArgumentParser. add_subparsers ([*title*] [, *description*] [, *prog*] [, *parser_class*] [, *action*] [, *option_string*] [, *dest*] [, *help*] [, *metavar*])

许多程序分割它们的功能分成若干子命令，例如，该svn程序可以调用像子命令，和。当程序执行需要不同种类的命令行参数的几个不同功能时，以这种方式拆分功能可能是一个特别好的想法。 支持用该方法创建这样的子命令 。该方法通常不带参数调用，并返回一个特殊的操作对象。该对象有一个方法，它接受一个命令名和任何构造函数参数，并返回一个可以像平常一样修改的对象。 svn checkout svn update svn commit ArgumentParser add_subparsers() add_subparsers() add_parser() ArgumentParser ArgumentParser

参数描述：

- title - 帮助输出中的子分析器组的标题; 如果提供了描述，则默认为“子命令”，否则使用标题作为位置参数
- 描述 - 默认情况下帮助输出中的子分析器组的说明 None
- 将使用子命令帮助显示的prog - 使用信息，默认情况下程序的名称和子分析器参数前的任何位置参数
- parser_class - 将用于创建子分析器实例的类，默认情况下是当前分析器的类（ 例如，ArgumentParser ）
- 操作 - 在命令行中遇到此参数时要采取的基本操作类型
- dest - 存储子命令名称的属性名称; 默认情况下None不存储任何值
- 帮助 - 默认帮助帮助输出中的子分析器组None
- metavar - 字符串在帮助中显示可用的子命令; 默认情况下它是None和呈现形式为{cmd1，cmd2，..}的子命令

一些示例用法：

```
>>> # create the top-level parser
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', action='store_true', help='foo help')
>>> subparsers = parser.add_subparsers(help='sub-command help')
>>>
>>> # create the parser for the "a" command
>>> parser_a = subparsers.add_parser('a', help='a help')
>>> parser_a.add_argument('bar', type=int, help='bar help')
>>>
>>> # create the parser for the "b" command
>>> parser_b = subparsers.add_parser('b', help='b help')
>>> parser_b.add_argument('--baz', choices='XYZ', help='baz help')
>>>
>>> # parse some argument lists
>>> parser.parse_args(['a', '12'])
Namespace(bar=12, foo=False)
>>> parser.parse_args(['--foo', 'b', '--baz', 'Z'])
Namespace(baz='Z', foo=True)
```

请注意，返回的对象parse_args()将仅包含由命令行选择的主分析器和子分析器的属性（而不是任何其他子分析器）。因此，在上面的示例中，当a指定命令时，只有foo和bar属性存在，并且在b指定命令时，只有foo和 baz属性存在。

同样，当从分析器请求帮助消息时，只会打印该特定分析器的帮助。帮助消息将不包括父解析器或同级解析器消息。（但是，每个子分析器命令的帮助消息可以通过提供help=参数给出，add_parser()如上所示。）

```
>>> parser.parse_args(['--help'])
usage: PROG [-h] [--foo] {a,b} ...

positional arguments:
  {a,b}   sub-command help
    a     a help
    b     b help

optional arguments:
  -h, --help  show this help message and exit
  --foo   foo help

>>> parser.parse_args(['a', '--help'])
usage: PROG a [-h] bar

positional arguments:
  bar     bar help

optional arguments:
  -h, --help  show this help message and exit

>>> parser.parse_args(['b', '--help'])
usage: PROG b [-h] [--baz {X,Y,Z}]

optional arguments:
  -h, --help     show this help message and exit
  --baz {X,Y,Z}  baz help
```

该add_subparsers()方法还支持title和description 关键字参数。当出现任何一个时，子分析器的命令将出现在帮助输出中的他们自己的组中。例如：

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(title='subcommands',
...                                    description='valid subcommands',
...                                    help='additional help')
>>> subparsers.add_parser('foo')
>>> subparsers.add_parser('bar')
>>> parser.parse_args(['-h'])
usage:  [-h] {foo,bar} ...

optional arguments:
  -h, --help  show this help message and exit

subcommands:
  valid subcommands

  {foo,bar}   additional help
```

此外，还add_parser支持一个附加aliases参数，它允许多个字符串引用相同的子分析器。这个例子，就像svn别名co作为一个简写checkout：

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers()
>>> checkout = subparsers.add_parser('checkout', aliases=['co'])
>>> checkout.add_argument('foo')
>>> parser.parse_args(['co', 'bar'])
Namespace(foo='bar')
```

处理子命令的一种特别有效的方式是将该add_subparsers()方法的使用与调用结合起来，set_defaults()以便每个子分析器知道应该执行哪个Python函数。例如：

```
>>> # sub-command functions
>>> def foo(args):
...     print(args.x * args.y)
...
>>> def bar(args):
...     print('((%s))' % args.z)
...
>>> # create the top-level parser
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers()
>>>
>>> # create the parser for the "foo" command
>>> parser_foo = subparsers.add_parser('foo')
>>> parser_foo.add_argument('-x', type=int, default=1)
>>> parser_foo.add_argument('y', type=float)
>>> parser_foo.set_defaults(func=foo)
>>>
>>> # create the parser for the "bar" command
>>> parser_bar = subparsers.add_parser('bar')
>>> parser_bar.add_argument('z')
>>> parser_bar.set_defaults(func=bar)
>>>
>>> # parse the args and call whatever function was selected
>>> args = parser.parse_args('foo 1 -x 2'.split())
>>> args.func(args)
2.0
>>>
>>> # parse the args and call whatever function was selected
>>> args = parser.parse_args('bar XYZYX'.split())
>>> args.func(args)
((XYZYX))
```

这样，您可以parse_args()在参数分析完成后调用适当的函数。将功能与这样的动作相关联通常是处理每个子分析器的不同动作的最简单方法。但是，如果需要检查被调用的子分析器的名称，则该调用的dest关键字参数add_subparsers()将起作用：

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(dest='subparser_name')
>>> subparser1 = subparsers.add_parser('1')
>>> subparser1.add_argument('-x')
>>> subparser2 = subparsers.add_parser('2')
>>> subparser2.add_argument('y')
>>> parser.parse_args(['2', 'frobble'])
Namespace(subparser_name='2', y='frobble')
```

16.4.5.2。FileType对象

class argparse.FileType（*mode ='r'*，*bufsize = -1*，*encoding = None*，*errors = None* ）

该FileType工厂创建一个可以传递给类型参数的对象ArgumentParser.add_argument()。具有FileType对象作为其类型的参数将打开命令行参数作为具有所请求模式，缓冲区大小，编码和错误处理的文件（open()有关更多详细信息，请参阅该函数）：

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--raw', type=argparse.FileType('wb', 0))
>>> parser.add_argument('out', type=argparse.FileType('w', encoding='UTF-8'))
>>> parser.parse_args(['--raw', 'raw.dat', 'file.txt'])
Namespace(out=<_io.TextIOWrapper name='file.txt' mode='w' encoding='UTF-8'>, raw=<_io.FileI
```

FileType对象理解伪参数'-'并自动将其转换sys.stdin为可读FileType对象和 sys.stdout可写FileType对象：

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('infile', type=argparse.FileType('r'))
>>> parser.parse_args(['-'])
Namespace(infile=<_io.TextIOWrapper name='<stdin>' encoding='UTF-8'>)
```

在新版本3.4：在编码和错误关键字参数。

16.4.5.3。参数组

ArgumentParser.add_argument_group（*title = None*，*description = None* ）

默认情况下，ArgumentParser在显示帮助消息时将命令行参数分组为“位置参数”和“可选参数”。当比这个默认参数有一个更好的概念组参数时，可以使用该add_argument_group()方法创建适当的组 ：

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> group = parser.add_argument_group('group')
>>> group.add_argument('--foo', help='foo help')
>>> group.add_argument('bar', help='bar help')
>>> parser.print_help()
usage: PROG [--foo FOO] bar

group:
  bar    bar help
  --foo FOO  foo help
```

该add_argument_group()方法返回一个参数组对象，它有一个add_argument()像普通的方法ArgumentParser。当将参数添加到组中时，解析器将其视为与普通参数类似，但将参数显示在单独的组中以获取帮助消息。该add_argument_group()方法接受可用于自定义此显示的*标题*和*描述*参数：

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> group1 = parser.add_argument_group('group1', 'group1 description')
>>> group1.add_argument('foo', help='foo help')
>>> group2 = parser.add_argument_group('group2', 'group2 description')
>>> group2.add_argument('--bar', help='bar help')
>>> parser.print_help()
usage: PROG [--bar BAR] foo

group1:
  group1 description

  foo    foo help

group2:
  group2 description

  --bar BAR  bar help
```

请注意，任何不在用户定义组中的参数将返回到通常的“位置参数”和“可选参数”部分。

16.4.5.4。互斥

ArgumentParser.add_mutually_exclusive_group（*required = False* ）

创建一个互斥组。argparse将确保在命令行中仅存在互斥组中的一个参数：

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_mutually_exclusive_group()
>>> group.add_argument('--foo', action='store_true')
>>> group.add_argument('--bar', action='store_false')
>>> parser.parse_args(['--foo'])
Namespace(bar=True, foo=True)
>>> parser.parse_args(['--bar'])
Namespace(bar=False, foo=False)
>>> parser.parse_args(['--foo', '--bar'])
usage: PROG [-h] [--foo | --bar]
PROG: error: argument --bar: not allowed with argument --foo
```

该add_mutually_exclusive_group()方法还接受*必需的*参数，以指示至少需要一个互斥参数：

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_mutually_exclusive_group(required=True)
>>> group.add_argument('--foo', action='store_true')
>>> group.add_argument('--bar', action='store_false')
>>> parser.parse_args([])
usage: PROG [-h] (--foo | --bar)
PROG: error: one of the arguments --foo --bar is required
```

请注意，目前互斥的参数组不支持 *标题*和*描述*参数 add_argument_group()。

16.4.5.5。解析器默认值

ArgumentParser.**set_defaults**(***kwargs*)

大多数情况下，parse_args() 通过检查命令行参数和参数操作来完全确定返回的对象的属性。set_defaults()允许一些额外的属性被确定，而不需要检查命令行的添加：

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', type=int)
>>> parser.set_defaults(bar=42, baz='badger')
>>> parser.parse_args(['736'])
Namespace(bar=42, baz='badger', foo=736)
```

请注意，解析器级别的默认值总是覆盖参数级别的默认值：

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default='bar')
>>> parser.set_defaults(foo='spam')
>>> parser.parse_args([])
Namespace(foo='spam')
```

解析器级别的默认值在使用多个解析器时可能特别有用。请参阅该add_subparsers()方法以获取此类型的示例。

ArgumentParser.**get_default**(*dest*)

获取默认值的命名空间属性，为通过设置 add_argument()或通过 set_defaults()：

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default='badger')
>>> parser.get_default('foo')
'badger'
```

16.4.5.6。打印帮助

在大多数典型的应用程序中，parse_args()将负责格式化和打印任何使用或错误消息。但是，有几种格式化方法可用：

ArgumentParser.print_usage(*file* = *None*)

打印关于ArgumentParser应该如何在命令行上调用的简要说明。如果*文件*是None，sys.stdout假定。

ArgumentParser.print_help(*file* = *None*)

打印帮助信息，包括程序使用情况和关于使用注册的参数的信息ArgumentParser。如果*文件*是None，sys.stdout假定。

也有这些方法的变体，只是返回一个字符串，而不是打印它：

ArgumentParser.format_usage()

返回一个字符串，其中包含ArgumentParser应该如何在命令行上调用的简要说明。

ArgumentParser.format_help()

返回一个包含帮助信息的字符串，包括程序使用情况和有关使用注册的参数的信息ArgumentParser。

16.4.5.7。部分解析

ArgumentParser.parse_known_args(*args* = *None*，*namespace* = *None*)

有时脚本可能只解析一些命令行参数，将剩下的参数传递给另一个脚本或程序。在这些情况下，该parse_known_args()方法可能很有用。它的工作方式很像，parse_args()只是在出现额外参数时不会产生错误。相反，它会返回一个包含已填充名称空间和剩余参数字符串列表的两个项目元组。

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_true')
>>> parser.add_argument('bar')
>>> parser.parse_known_args(['--foo', '--badger', 'BAR', 'spam'])
(Namespace(bar='BAR', foo=True), ['--badger', 'spam'])
```

警告： 前缀匹配规则适用于parse_known_args()。解析器可能会使用一个选项，即使它只是其中一个已知选项的前缀，而不是将它留在剩余的参数列表中。

16.4.5.8。自定义文件解析

ArgumentParser.convert_arg_line_to_args(*arg_line*)

从文件读取的参数（请参阅构造函数的*fromfile_prefix_chars*关键字参数ArgumentParser）每行读取一个参数。convert_arg_line_to_args()可以替代更好的阅读。

该方法采用一个参数*arg_line*，它是从参数文件中读取的字符串。它返回从此字符串解析的参数列表。按顺序从参数文件中读取每行一次的方法。

这种方法的有用重写是将每个由空格分隔的单词作为参数。以下示例演示如何执行此操作：

```
class MyArgumentParser(argparse.ArgumentParser):
    def convert_arg_line_to_args(self, arg_line):
        return arg_line.split()
```

16.4.5.9。退出方法

ArgumentParser.exit（*status* = *0*，*message* = *None* ）

该方法终止程序，以指定的*状态*退出， 并且如果给定，则在此之前打印*消息*。

ArgumentParser.error（*消息*）

此方法将包含*消息*的使用消息打印到标准错误，并终止状态代码为2的程序。

16.4.6。升级optparse代码

最初，argparse模块试图保持兼容性optparse。但是，optparse很难透明地进行扩展，尤其是需要支持新的nargs=说明符和更好的使用消息所需的更改 。当大多数内容 optparse都被复制粘贴或猴子修补时，尝试维持向后兼容性似乎不再实际可行。

该argparse模块optparse 以多种方式改进了标准库模块，包括：

- 处理位置参数。
- 支持子命令。
- 允许像+和的替代选项前缀/。
- 处理零个或多个以及一个或多个样式的参数。
- 生成更多信息的使用信息。
- 为定制type和提供更简单的界面action。

从A部分升级路径optparse到argparse：

- 用optparse.OptionParser.add_option()呼叫 替换所有ArgumentParser.add_argument()呼叫。
- 更换用并添加额外 的定位参数调用。请记住，现在在上下文中调用的是之前称为的内容。(options, args) = parser.parse_args() args = parser.parse_args() ArgumentParser.add_argument() options argparse args
- 更换 optparse.OptionParser.disable_interspersed_args() 设定 nargs 一个位置参数来 argparse.REMAINDER，或使用parse_known_args()收集未解析字符串参数在一个单独的列表。
- callback_*使用type或action参数替换回调操作和关键字 参数。
- 将type关键字参数的字符串名称替换为相应的类型对象（例如int，float，complex等）。
- 更换 optparse.Values 用 Namespace 和 optparse.OptionError 与 optparse.OptionValueError 用 ArgumentError。
- 用隐式参数替换字符串，例如%default或%prog使用标准Python语法来使用字典格式化字符串，也就是 %(default)s和%(prog)s。
- 用 version 调用来替换OptionParser构造函数参数 。 parser.add_argument('--version', action='version', version='<the version>')

16.5。 getopt- 用于命令行选项的C风格解析器

源代码： Lib / getopt.py

> **注意：** 该getopt模块是用于命令行选项的解析器，其API旨在为C getopt()函数的用户所熟悉。不熟悉C getopt()函数的用户，或者希望编写更少代码并获得更好的帮助和错误消息的用户应考虑使用该 argparse模块。

该模块帮助脚本解析命令行参数sys.argv。它支持与Unix getopt()函数相同的约定（包括'-'和'--'形式的参数的特殊含义）。通过可选的第三个参数，也可以使用类似于GNU软件支持的长选项。

该模块提供了两个功能和一个例外：

getopt.getopt（*args*，*shortopts*，*longopts = []*）

分析命令行选项和参数列表。 *args*是要解析的参数列表，没有对正在运行的程序的引用。通常，这意味着sys.argv[1:]。*短片*是脚本想要识别的选项字母串，其中需要参数后跟冒号的选项（':'即，与Unix getopt()使用的格式相同）。

> **注意：** 与GNU不同getopt()，在非选项参数之后，所有进一步的参数也被认为是非选项。这与非GNU Unix系统的工作方式类似。

如果指定了*longopts*，则必须是具有应该支持的长选项名称的字符串列表。主要'--'字符不应包含在选项名称中。需要参数的长选项后面应该跟一个等号（'='）。可选参数不受支持。要只接受长选项，*短裤*应该是一个空字符串。只要提供的选项名称前缀与接受的选项之一完全匹配，就可以识别命令行上的长选项。例如，如果*longopts*是，该选项将匹配为，但不会唯一匹配，因此会被提出。['foo', 'frob']--fo--foo--fGetoptError

返回值由两个元素组成：第一个是成对的列表; 第二个是选项列表被剥离后留下的程序参数列表（这是一个后面的*args*片段）。每个返回的选项和值对具有作为其第一个元素的选项，其前缀为短选项（例如）的连字符或长选项的两个连字符（例如），选项参数作为其第二个元素或空字符串，如果该选项没有参数。这些选项按照与它们相同的顺序出现在列表中，从而允许多次出现。多头和空头期权可能会混合。(option, value)'-x''--long-option'

getopt.gnu_getopt（*args*，*shortopts*，*longopts = []*）

此功能的工作原理与getopt()默认情况下使用的GNU风格扫描模式不同。这意味着选项和非选项参数可能会混在一起。 getopt()只要遇到非选项参数，该函数立即停止处理选项。

如果选项字符串的第一个字符是'+'，或者是环境变量POSIXLY_CORRECT 被设置，则只要遇到非选项参数，选项处理就会停止。

*异常*getopt.GetoptError

当在参数列表中找到无法识别的选项或者需要参数的选项不存在时，会引发此问题。异常的参数是一个指示错误原因的字符串。对于长期选项，给一个不需要一个选项的参数也会

导致这个异常被提出。属性msg并opt给出错误信息和相关选项; 如果没有与异常相关的特定选项， opt则为空字符串。

*异常*getopt.error

别名GetoptError; 为了向后兼容。

仅使用Unix样式选项的示例：

```
>>> import getopt
>>> args = '-a -b -cfoo -d bar a1 a2'.split()
>>> args
['-a', '-b', '-cfoo', '-d', 'bar', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'abc:d:')
>>> optlist
[('-a', ''), ('-b', ''), ('-c', 'foo'), ('-d', 'bar')]
>>> args
['a1', 'a2']
```

使用长选项名称同样简单：

```
>>> s = '--condition=foo --testing --output-file abc.def -x a1 a2'
>>> args = s.split()
>>> args
['--condition=foo', '--testing', '--output-file', 'abc.def', '-x', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'x', [
...     'condition=', 'output-file=', 'testing'])
>>> optlist
[('--condition', 'foo'), ('--testing', ''), ('--output-file', 'abc.def'), ('-x', '')]
>>> args
['a1', 'a2']
```

在脚本中，典型用法如下所示：

```
import getopt, sys

def main():
    try:
        opts, args = getopt.getopt(sys.argv[1:], "ho:v", ["help", "output="])
    except getopt.GetoptError as err:
        # print help information and exit:
        print(err)  # will print something like "option -a not recognized"
        usage()
        sys.exit(2)
    output = None
    verbose = False
    for o, a in opts:
        if o == "-v":
            verbose = True
        elif o in ("-h", "--help"):
            usage()
            sys.exit()
        elif o in ("-o", "--output"):
            output = a
        else:
```

```
            assert False, "unhandled option"
    # ...

if __name__ == "__main__":
    main()
```

请注意，通过使用argparse模块，可以使用更少的代码和更多的信息帮助和错误消息来生成等效的命令行界面：

```
import argparse

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-o', '--output')
    parser.add_argument('-v', dest='verbose', action='store_true')
    args = parser.parse_args()
    # ... do something with args.output ...
    # ... do something with args.verbose ..
```

也可以看看:

模 argparse
    可选的命令行选项和参数解析库。

16.6。 logging- Python的日志工具

源代码： Lib / logging / __init__.py

该模块定义了为应用程序和库实现灵活事件记录系统的函数和类。

使用标准库模块提供的日志API的主要好处是，所有Python模块都可以参与日志记录，因此您的应用程序日志可以包含自己的消息，并与第三方模块的消息进行集成。

该模块提供了很多功能和灵活性。如果您对日志记录不熟悉，最好的办法是看看教程（参见右边的链接）。

重要

此页面包含API参考信息。有关更高级主题的教程信息和讨论，请参阅

- 基本教程
- 高级教程
- 记录食谱

下面列出了模块定义的基本类及其功能。

- 记录器公开应用程序代码直接使用的接口。
- 处理程序将日志记录（由记录器创建）发送到相应的目标。
- 过滤器提供更精细的设备来确定要输出的日志记录。
- 格式化程序指定最终输出中日志记录的布局。

16.6.1。记录器对象

记录器具有以下属性和方法。请注意，记录器不会直接实例化，而是始终通过模块级功能进行实例化 logging.getLogger(name)。多次调用getLogger()具有相同的名称将始终返回对同一个Logger对象的引用。

这name可能是一个由期限分隔的分层值，例如 foo.bar.baz （尽管它也可能只是简单foo的例如）。在分层列表中更靠后的记录器是列表中较高记录器的子记录器。例如，给定一个记录器使用的名称foo，记录仪用的名字foo.bar ， foo.bar.baz以及foo.bam是所有后代foo。记录器名称层次结构类似于Python包层次结构，如果使用推荐的结构按每个模块组织记录器，则记录器名称层次结构与此类似 logging.getLogger(__name__)。这是因为在模块中， __name__ Python包名称空间中的模块名称。

*类*logging.Logger

propagate

如果此属性的计算结果为true，除了附加到此记录器的任何处理程序外，记录到此记录程序的事件将传递给更高级别（祖先）记录程序的处理程序。消息直接传递给祖先记录器的处理程序 - 既不考虑祖先记录器的级别也不考虑过滤器。

如果它的计算结果为false，则日志消息不会传递给祖先记录器的处理程序。

构造函数将此属性设置为True。

注意： 如果将处理程序附加到记录器*及其*一个或多个其祖先，它可能会多次发出相同的记录。一般情况下，您不需要将处理程序附加到多个记录器 - 如果您只将它附加到记录器层次结构中最高的相应记录器，那么它会看到所有后裔记录器记录的所有事件，前提是它们的传播设置被设置为`True`。一种常见的情况是仅将处理程序附加到根记录器，并让传播处理其余部分。

`setLevel`（*level*）

将此记录器的阈值设置为*等级*。记录不如*级别*严重的消息将被忽略; 具有严重*级别*或更高*级别的*日志消息将由处理程序或处理程序服务此记录程序发出，除非处理程序的级别设置为比*级别*更高的严重*级别*。

创建记录器时，级别设置为`NOTSET`（当记录器是根记录器时会导致所有消息处理，或者当记录器是非根记录器时将其委派给父项）。请注意，根记录器是使用级别创建的`WARNING`。

术语“对父代的委托”意味着如果记录器具有NOTSET级别，则遍历其祖先记录器链，直至找到具有除NOTSET之外的级别的祖先，或者达到根。

如果发现祖先的级别不是NOTSET，则祖先的级别被视为祖先搜索开始时的记录器的有效级别，并且用于确定如何处理日志记录事件。

如果达到根目录，并且其级别为NOTSET，则将处理所有消息。否则，根的级别将被用作有效级别。

请参阅记录级别以获取级别列表。

*在版本3.2中进行了更改：*现在，*level*参数接受字符串表示形式，如'INFO'作为整数常量的替代形式，如`INFO`。但是，请注意，这些级别在内部存储为整数，例如`getEffectiveLevel()`和`isEnabledFor()`将返回/期望传递整数的方法。

`isEnabledFor`（*lvl*）

指示是否将由此记录器处理严重性*级别*为*lvl*的消息。这种方法首先检查由设置的模块级别级别`logging.disable(lvl)`，然后检查记录器的有效级别`getEffectiveLevel()`。

`getEffectiveLevel`（）

指示此记录器的有效等级。如果`NOTSET`已经设置了以外的值`setLevel()`，则返回该值。否则，层次结构将遍历到根，直到找到一个以外的值`NOTSET`，并返回该值。返回的值是一个整数，通常是其中的一个`logging.DEBUG`，`logging.INFO`等等。

`getChild`（*后缀*）

根据后缀确定，返回记录器，该记录器是此记录器的后代。因此，`logging.getLogger('abc').getChild('def.ghi')`将返回与返回的相同的记录器`logging.getLogger('abc.def.ghi')`。这是一种方便的方法，当使用eg `__name__`而不是字面字符串命名父记录器时很有用。

3.2版本中的新功能

`debug`（*msg*，**args*，***kwargs*）

DEBUG在此记录器上记录一级消息。该*MSG*是消息格式字符串，并且*ARGS*是被合并到参数 *MSG*使用字符串格式化操作。（请注意，这意味着您可以使用格式字符串中的关键字以及单个字典参数。）

在被检查的*kwargs中*有三个关键字参数： *exc_info*，*stack_info*和*extra*。

如果*exc_info*未评估为false，则会导致将异常信息添加到日志消息中。如果提供了异常元组（以返回的格式 `sys.exc_info()` ）或异常实例，则会使用它；否则，`sys.exc_info()`被调用来获取异常信息。

第二个可选的关键字参数是*stack_info*，默认为 `False`。如果为true，堆栈信息将添加到日志消息中，包括实际的日志记录调用。请注意，这与通过指定*exc_info*显示的堆栈信息*不同*：前者是从堆栈底部到当前线程中的日志记录调用的堆栈帧，而后者是关于已解除堆栈帧的信息，遇到异常，同时搜索异常处理程序。

您可以指定*stack_info*独立的*exc_info*，如只显示你是怎么在代码中的某个点，即使没有异常曾引起。堆栈框架按照以下标题行打印：

```
Stack (most recent call last):
```

这模仿了显示异常帧时使用的内容。`Traceback (most recent call last):`

第三个关键字参数是*额外的*，它可以用来传递一个字典，该字典用于使用用户定义的属性填充为记录事件创建的LogRecord的__dict__。这些自定义属性可以随意使用。例如，它们可以合并到记录的消息中。例如：

```
FORMAT = '%(asctime)-15s %(clientip)s %(user)-8s %(message)s'
logging.basicConfig(format=FORMAT)
d = {'clientip': '192.168.0.1', 'user': 'fbloggs'}
logger = logging.getLogger('tcpserver')
logger.warning('Protocol problem: %s', 'connection reset', extra=d)
```

会打印类似的东西

```
2006-02-08 22:20:02,165 192.168.0.1 fbloggs  Protocol problem: connection rese
```

在通过了词典的按键*额外*不应由记录系统所使用的密钥冲突。（有关`Formatter`日志记录系统使用哪些密钥的更多信息，请参阅文档。）

如果您选择在记录的消息中使用这些属性，则需要谨慎操作。例如，在上面的例子中，`Formatter`已经用LogRecord属性字典中的'clientip'和'user'设置了一个格式字符串。如果缺少这些信息，将不会记录该消息，因为会出现字符串格式异常。所以在这种情况下，你总是需要用这些键传递*额外的*字典。

虽然这可能很烦人，但此功能旨在用于特殊情况下，例如多线程服务器，其中相同的代码在许多环境中执行，并且产生的有趣情况取决于此环境（如远程客户端IP地址和经过身份验证用户名，在上面的例子中）。在这种情况下，专门的 `Formatter`s可能会与特定的`Handler`s 一起使用。

*：在3.2版本中的新*的*stack_info*加入参数。

*改变在3.5版本：*该*exc_info*现在参数可以接受的例外情况。

info (*msg* , ** args* , *** kwargs*)

INFO在此记录器上记录一级消息。参数被解释为debug()。

warning (*msg* , ** args* , *** kwargs*)

WARNING在此记录器上记录一级消息。参数被解释为debug()。

> **注意:** 有一种过时的方法warn在功能上与之相同warning。如warn已弃用，请勿使用它 - 请warning改用。

error (*msg* , ** args* , *** kwargs*)

ERROR在此记录器上记录一级消息。参数被解释为debug()。

critical (*msg* , ** args* , *** kwargs*)

CRITICAL在此记录器上记录一级消息。参数被解释为debug()。

log (*lvl* , *msg* , ** args* , *** kwargs*)

在此记录器上记录具有整数级别*lvl*的消息。其他参数解释为debug()。

exception (*msg* , ** args* , *** kwargs*)

ERROR在此记录器上记录一级消息。参数被解释为debug()。异常信息被添加到日志消息中。该方法只能从异常处理程序调用。

addFilter (*过滤器*)

将指定的过滤*器*过滤*器*添加到此记录器。

removeFilter (*过滤器*)

从此记录器中删除指定的过滤器*过滤*器。

filter (*记录*)

将记录器的过滤器应用于记录，并在记录要处理时返回真值。依次查阅过滤器，直到其中一个返回错误值。如果它们中没有一个返回错误值，则记录将被处理（传递给处理程序）。如果返回一个假值，则不会进一步处理该记录。

addHandler (*hdlr*)

将指定的处理程序*hdlr*添加到此记录器。

removeHandler (*hdlr*)

从此记录器中移除指定的处理程序*hdlr*。

findCaller (*stack_info = False*)

查找调用者的源文件名和行号。以4元素元组的形式返回文件名，行号，函数名称和堆栈信息。None除非*stack_info*是，否则将返回堆栈信息True。

handle (*记录*)

通过将记录传递给与此记录器及其祖先相关联的所有处理程序来处理记录（直到找到错误的*传播*值）。此方法用于从套接字接收的未识别记录以及本地创建的记录。记录

器级别过滤使用filter()。

makeRecord (*name* , *lvl* , *fn* , *lno* , *msg* , *args* , *exc_info* , *func = None* , *extra = None* , *sinfo = None*)

这是一个工厂方法，可以在子类中重写以创建专用LogRecord实例。

hasHandlers ( )

检查该记录器是否配置了任何处理程序。这是通过在记录器层次结构中寻找此记录器及其父母中的处理程序完成的。返回，True如果找到一个处理程序，否则False。只要发现'propagate'属性设置为false的记录器，该方法就停止搜索层次结构 - 这将是检查处理器存在的最后一个记录器。

3.2版本中的新功能

*在版本3.7中更改：*现在可以挑选和取消记录器。

16.6.2。日志级别

日志级别的数值在下表中给出。如果你想定义你自己的水平，并且需要它们具有相对于预定义水平的特定值，这些主要是感兴趣的。如果您使用相同的数值定义一个级别，它会覆盖预定义的值; 预定义的名称将丢失。

水平	数字值
CRITICAL	50
ERROR	40
WARNING	三十
INFO	20
DEBUG	10
NOTSET	0

16.6.3。处理程序对象

处理程序具有以下属性和方法。请注意，Handler 它不会直接实例化; 这个类作为更有用的子类的基础。但是，__init__()子类中的方法需要调用 Handler.__init__()。

*类*logging.Handler

__init__ (*level = NOTSET*)

Handler通过设置其级别来初始化实例，将过滤器列表设置为空列表并创建一个锁（使用createLock() ）来序列化对I / O机制的访问。

createLock ( )

初始化线程锁，该线程锁可用于序列化对可能不是线程安全的底层I / O功能的访问。

acquire ( )

获取创建的线程锁createLock()。

release ( )

释放通过获取的线程锁acquire()。

setLevel (*level*)

将此处理程序的阈值设置为*级别*。记录不比*级别*严重的消息将被忽略。当创建一个处理程序时，级别设置为NOTSET（这会导致处理所有消息）。

请参阅记录级别以获取级别列表。

*在版本3.2中进行了更改：*现在，*level*参数接受字符串表示形式，如'INFO'作为整数常量的替代形式，如INFO。

setFormatter (*fmt*)

Formatter将此处理程序设置为*fmt*。

addFilter (*过滤器*)

将指定的过滤*器*过滤*器*添加到此处理程序。

removeFilter (*过滤器*)

从此处理程序中删除指定的过滤*器*过滤*器*。

filter (*记录*)

如果要处理记录，则将此处理程序的过滤器应用于记录并返回真值。依次查阅过滤器，直到其中一个返回错误值。如果它们中没有一个返回错误值，则记录将被发射。如果返回一个假值，处理程序不会发出记录。

flush ( )

确保所有日志记录输出已被刷新。这个版本什么都不做，并且打算由子类来实现。

close ( )

整理处理程序使用的所有资源。该版本没有输出，但是从处理程序的内部列表中删除处理程序，处理程序在shutdown()被调用时关闭 。子类应该确保这个被覆盖的close()方法调用。

handle (*记录*)

根据可能已添加到处理程序的过滤器，有条件地发出指定的日志记录。采集/释放I / O线程锁包裹记录的实际排放。

handleError (*记录*)

在emit()呼叫期间遇到异常时，应该从处理程序调用此方法。如果模块级别的属性raiseExceptions是False，则异常将被忽略。这就是日志系统最需要的东西 - 大多数用户不会在意日志系统中的错误，他们更关心应用程序错误。但是，如果您愿意，可以用自定义处理程序替换它。指定的记录是发生异常时正在处理的记录。（默认值raiseExceptions是True，因为这在开发过程中更有用）。

format (*记录*)

格式化记录 - 如果设置了格式化程序，请使用它。否则，请使用该模块的默认格式化程序。

emit（*记录*）

做任何事情来实际记录指定的日志记录。此版本旨在由子类实现，因此引发了一个NotImplementedError。

有关作为标准包含的处理程序列表，请参阅logging.handlers。

16.6.4。格式化对象

Formatter对象具有以下属性和方法。他们负责将a转换LogRecord成（通常）一个可以被人类或外部系统解释的字符串。该基地 Formatter允许指定一个格式化字符串。如果没有提供，'%(message)s'则使用缺省值，该值仅包含日志记录调用中的消息。要在格式化输出中添加更多信息（如时间戳），请继续阅读。

格式化器可以与一个格式字符串，其利用所述的知识的被初始化LogRecord的属性-例如上文利用的事实，即该用户的消息和参数是提及的默认值预格式化成LogRecord的*消息*属性。这个格式字符串包含标准的Python%风格的映射键。有关字符串格式 的更多信息，请参见printf-style字符串格式部分。

a LogRecord中有用的映射键在LogRecord属性部分给出 。

class logging.Formatter（*fmt = None*，*datefmt = None*，*style ='%'*）

返回类的新实例Formatter。该实例初始化为整个消息的格式字符串，以及消息的日期/时间部分的格式字符串。如果没有指定*fmt*，'%(message)s'则使用。如果没有指定*datefmt*，则使用类似ISO8601（或RFC3339）的日期格式。有关formatTime()更多详情，请参阅 文档。

该*风格*参数可以是“%”之一，“{”或“$”，并确定如何格式字符串将其数据合并：使用%-formatting之一， str.format()或string.Template。请参阅在整个应用程序中使用特定的格式化样式， 以获取有关使用{ - 和$ 格式化日志消息的更多信息。

*在3.2版本中更改：*该*风格*中加入参数。

format（*记录*）

记录的属性字典被用作字符串格式化操作的操作数。返回结果字符串。格式化字典之前，执行一些准备步骤。 记录的*消息*属性使用*msg%args进行*计算。如果格式化字符串包含'(asctime)'，formatTime()则调用格式化事件时间。如果有异常信息，则使用formatException()消息格式化并附加到该消息。请注意格式化的异常信息缓存在属性*exc_text中*。这很有用，因为异常信息可以通过网络进行腌制和发送，但是如果您有多个异常信息，则应该小心Formatter定制异常信息格式的子类。在这种情况下，格式化程序完成其格式化后，必须清除缓存值，以便处理该事件的下一个格式化程序不使用缓存值，而是重新计算它。

如果堆栈信息可用，则会将其附加到异常信息之后，formatStack()并在必要时使用它进行转换。

formatTime（*record*，*datefmt = None*）

该方法应该format()由格式化程序调用，该程序要使用格式化的时间。在格式化程序中可以重写此方法以提供任何特定要求，但其基本行为如下所示：如果指定了

datefmt（一个字符串），它将用于 time.strftime()格式化记录的创建时间。否则，使用ISO8601（或RDC 3339）格式。结果字符串被返回。

该函数使用用户可配置的函数将创建时间转换为元组。默认情况下，time.localtime()使用；要为特定的格式化程序实例更改此converter属性，请将该特性设置为与time.localtime()或 具有相同签名的函数time.gmtime()。要为所有格式化程序更改它，例如，如果您希望以GMT格式显示所有记录时间，请converter 在Formatter该类中设置属性。

*版本3.3中的更改：*以前，类似默认ISO8601的格式是硬编码的，如下例所示：其中逗号前的部分由strptime格式的string（ ）处理，而逗号后的部分为毫秒值。由于strptime在毫秒中没有格式占位符，因此毫秒值将使用另一个格式字符串追加- 并且这两个格式字符串都已硬编码到此方法中。随着更改，这些字符串被定义为类级别的属性，在需要时可以在实例级别覆盖这些属性。属性的名称（用于strptime格式字符串）和（用于追加毫秒值）。2010-09-06 22:38:15,292’%Y-%m-%d %H:%M:%S’’%s,%03d’default_time_formatdefault_msec_format

formatException（*exc_info* ）

将指定的异常信息（由返回的标准异常元组sys.exc_info()）格式化为字符串。这个默认实现只是使用traceback.print_exception()。结果字符串被返回。

formatStack（*stack_info* ）

将指定的堆栈信息（作为返回的字符串traceback.print_stack()，但删除最后一个换行符）格式化 为字符串格式。这个默认实现只是返回输入值。

16.6.5。筛选对象

Filters可以被用于Handlers和Loggers用于比级别提供的更复杂的过滤。基本过滤器类只允许记录器层次结构中某个点以下的事件。例如，用'AB'初始化的过滤器将允许由记录器'A.B'，'ABC'，'ABCD'，'ABD'等记录的事件，但不允许'A.BB'，'BAB'等。如果初始化用空字符串传递所有事件。

class logging.Filter（*name* =''）

返回Filter类的一个实例。如果*名字*被指定，它的名字一个记录器，它与它的孩子在一起，将有允许通过过滤器的事件。如果*name*是空字符串，则允许每个事件。

filter（*记录*）

指定的记录是否被记录？对于no返回零，对于yes返回非零。如果认为合适，记录可以通过这种方法在原地进行修改。

请注意，在处理程序发出事件之前，会查看附加到处理程序的过滤器，而在将事件发送给处理程序之前debug()，每当记录事件（使用info()等）时都会查询连接到记录程序的过滤器。这意味着由后代记录器生成的事件不会被记录器的过滤器设置过滤，除非过滤器也已应用于这些后代记录器。

您实际上不需要子类Filter：您可以传递具有filter相同语义的方法的任何实例。

*在版本3.2中进行了更改：*您不需要创建专门的Filter类或使用其他类与filter方法：您可以使用函数（或其他可调用）作为过滤器。过滤逻辑将检查过滤器对象是否具有filter属性：如果它确实是，则它被假定为a Filter并且其filter()方法被调用。否则，它被认为是可调用的，并且将该记录作为单个参数进行调用。返回的值应该与返回的值一致 filter()。

虽然过滤器主要用于根据比级别更复杂的标准过滤记录，但他们可以看到每个由它们所连接的处理程序或记录程序处理的记录：如果您想要执行诸如计算多少个记录记录由特定的记录器或处理程序处理，或者在正在处理的LogRecord中添加，更改或删除属性。显然，改变LogRecord需要谨慎处理，但它确实允许将上下文信息注入日志中（请参阅使用过滤器来传递上下文信息）。

16.6.6。LogRecord对象

LogRecordLogger 每次记录事件时都会自动创建实例，并且可以通过makeLogRecord()（例如，通过线路接收的腌制事件）手动创建 实例。

class logging.LogRecord（*name*，*level*，*pathname*，*lineno*，*msg*，*args*，*exc_info*，*func* = *None*，*sinfo* = *None* ）

包含与正在记录的事件相关的所有信息。

主要信息被传入，msg并被args组合使用来创建记录的字段。 msg % argsmessage

参数：	• **name** - 用于记录由此LogRecord表示的事件的记录器的名称。请注意，该名称将始终具有此值，即使它可能由连接到其他（祖先）记录器的处理程序发出。 • **级别** - 日志事件的数字级别（DEBUG，INFO等之一）请注意，这将转换为LogRecord的*两个*属性： levelno数值和levelname相应级别名称。 • **路径名** - 进行日志记录调用的源文件的完整路径名。 • **lineno** - 进行日志记录调用的源文件中的行号。 • **msg** - 事件描述消息，可能是一个带有变量数据占位符的格式字符串。 • **args** - 要合并到*msg*参数中的变量数据以获取事件描述。 • **exc_info** - 包含当前异常信息的异常元组，或者None没有可用的异常信息。 • **func** - 调用日志记录调用的函数或方法的名称。 • **sinfo** - 一个文本字符串，表示当前线程中堆栈底部的堆栈信息，直到日志记录调用。

getMessage（ ）

LogRecord将任何用户提供的参数与消息合并后返回此实例的消息。如果日志调用的用户提供的消息参数不是字符串，str()则调用它将其转换为字符串。这允许使用用户定义的类作为消息，其__str__方法可以返回要使用的实际格式字符串。

*在版本3.2中进行了更改：*LogRecord通过提供用于创建记录的工厂，创建了一个更具可配置性的工具。工厂可以使用getLogRecordFactory()和设置setLogRecordFactory() （请参见工厂签名）。

此功能可用于在创建时将自己的值注入到LogRecord中。您可以使用以下模式：

```
old_factory = logging.getLogRecordFactory()

def record_factory(*args, **kwargs):
```

```
    record = old_factory(*args, **kwargs)
    record.custom_attribute = 0xdecafbad
    return record

logging.setLogRecordFactory(record_factory)
```

有了这种模式，多个工厂可以被链接在一起，只要它们不会覆盖对方的属性或无意中覆盖上面列出的标准属性，就不会有任何意外。

16.6.7。LogRecord属性

LogRecord有许多属性，其中大部分属性都是从构造函数的参数派生而来的。（请注意，名称并不总是与LogRecord构造函数参数和LogRecord属性完全对应。）这些属性可用于将记录中的数据合并到格式字符串中。下表按字母顺序列出了属性名称，它们的含义以及%-style格式字符串中相应的占位符。

如果您使用{} -formatting（`str.format()`），则可以将其 `{attrname}` 用作格式字符串中的占位符。如果您使用$ -formatting（`string.Template`），请使用表单`${attrname}`。当然，在这两种情况下，都要替换`attrname`为您要使用的实际属性名称。

在{}格式的情况下，可以通过将属性名称放在属性名称后面指定格式标记，并用冒号分隔。例如：一个占位符`{msecs:03d}`将会格式化一个毫秒的`4`as 值`004`。`str.format()`有关可用选项的完整详细信息，请参阅文档。

属性名称	格式	描述
ARGS	不需要自己设置格式。	参数元组被合并到`msg`生成中`message`，或者其值被用来合并的字典（当只有一个参数时，它是
asctime	`%(asctime)s`	`LogRecord`创建时的人类可读时间。默认情况下这是表单'2003-07-08 16:49:45,896'的格式
创建	`%(created)f`	当时间`LogRecord`被创建（通过返回的`time.time()`）。
exc_info	不需要自己设置格式。	异常元组（àla `sys.exc_info`）或者如果没有发生异常`None`。
文件名	`%(filename)s`	文件名部分`pathname`。
了funcName	`%(funcName)s`	包含日志记录调用的函数的名称。
levelname	`%(levelname)s`	文'DEBUG'，'INFO'，'WARNING'，'ERROR'，'CRITICAL'）。
levelno	`%(levelno)s`	对于消息数字记录级别（`DEBUG`，`INFO`，`WARNING`，`ERROR`，`CRITICAL`）。
LINENO	`%(lineno)d`	发出日志记录调用的源行号（如果可用）。
信息	`%(message)s`	记`msg % args`计算为。这是在`Formatter.format()`被调用时设置的
模	`%(module)s`	模块（名称部分`filename`）。
毫秒	`%(msecs)d`	`LogRecord`创建时间的毫秒部分。
味精	不需要自己设置格式。	在`args`生成`message`或任意对象的格式字符串（请参阅使用任意对象作为消息）。
名称	`%(name)s`	用于记录呼叫的记录器的名称。
路径	`%(pathname)s`	发出日志记录调用的源文件的完整路径名（如果
处理	`%(process)d`	进程ID（如果可用）。

processName	%(processName)s	进程名称（如果可用）。
relativeCreated	%(relativeCreated)d	[illegible]时间，LogRecord创
stack_info	你不需要自己设置格	[illegible]
线	%(thread)d	线程ID（如果可用）。
threadName	%(threadName)s	线程名称（如果可用）。

*在版本3.1中更改：processName*被添加。

16.6.8。LoggerAdapter对象

LoggerAdapter实例用于方便地将上下文信息传递到日志记录调用中。有关使用示例，请参阅有关向日志输出添加上下文信息的部分 。

*类*logging. LoggerAdapter（ *记录器*，*额外* ）

返回一个LoggerAdapter使用底层Logger实例和字典对象初始化的实例。

process（ *msg*，*kwargs* ）

修改传递给日志记录调用的消息和/或关键字参数以插入上下文信息。该实现将对象作为*额外*传递给构造函数，并使用'extra'关键字将其添加到*kwargs*。返回值是一个（*msg*，*kwargs*）元组，它具有传入的参数（可能是修改的）版本。

除了上述以外，LoggerAdapter支持下面的方法Logger：debug()，info()，warning()，error()，exception()，critical()，log()，isEnabledFor()，getEffectiveLevel()，setLevel()和 hasHandlers()。这些方法与其中的对应方具有相同的签名Logger，因此可以交换使用这两种类型的实例。

*改变在3.2版本：*在isEnabledFor()，getEffectiveLevel()，setLevel()和hasHandlers()方法加入LoggerAdapter。这些方法委托给底层记录器。

16.6.9。线程安全

日志记录模块旨在保证线程安全，而不需要客户完成任何特殊工作。它通过使用线程锁实现了这一点; 有一个锁来序列化访问模块的共享数据，并且每个处理程序还创建一个锁来序列化对其底层I / O的访问。

如果您正在使用signal 模块实现异步信号处理程序，则可能无法在此类处理程序中使用日志记录。这是因为threading模块中的锁实现并不总是可重入的，所以不能从这些信号处理程序中调用。

10年6月16日。模块级函数

除了上述类别之外，还有许多模块级别的功能。

logging. getLogger（ *name = None* ）

返回具有指定名称的记录器，或者如果名称是None，则返回记录器，该记录器是层次结构的根记录器。如果指定，名称通常是点分隔的分层名称，如*“a”*，*“a.b”*或*“abcd”*。这些名称的选择完全取决于正在使用日志记录的开发人员。

使用给定名称对此函数的所有调用都将返回相同的记录器实例。这意味着记录器实例永远不需要在应用程序的不同部分之间传递。

logging.**getLoggerClass**( )

返回标准Logger类或传递给的最后一个类 setLoggerClass()。可以在新的类定义中调用此函数，以确保安装自定义Logger类不会撤销其他代码已应用的自定义。例如：

```
class MyLogger(logging.getLoggerClass()):
    # ... override behaviour here
```

logging.**getLogRecordFactory**( )

返回一个用于创建一个可调用的对象LogRecord。

*版本3.2中的新功能：*此功能已提供，setLogRecordFactory()以便开发人员更好地控制如何LogRecord 构建表示记录事件。

查看setLogRecordFactory()更多关于工厂被调用的信息。

logging.**debug**(*msg*, ** args*, *** kwargs*)

DEBUG在根记录器上记录具有级别的消息。该*MSG*是消息格式字符串，并且*ARGS*是被合并到参数 *MSG*使用字符串格式化操作。（请注意，这意味着您可以使用格式字符串中的关键字以及单个字典参数。）

在被检查的*kwargs中*有三个关键字参数：*exc_info* ，如果它不计算为false，则会导致异常信息被添加到日志消息中。如果提供了一个异常元组（以返回的格式 sys.exc_info() ），则使用它; 否则，sys.exc_info() 被调用来获取异常信息。

第二个可选的关键字参数是*stack_info*，默认为 False。如果为true，堆栈信息将添加到日志消息中，包括实际的日志记录调用。请注意，这与通过指定*exc_info*显示的堆栈信息*不同*：前者是从堆栈底部到当前线程中的日志记录调用的堆栈帧，而后者是关于已解除堆栈帧的信息，遇到异常，同时搜索异常处理程序。

您可以指定*stack_info*独立的*exc_info*，如只显示你是怎么在代码中的某个点，即使没有异常曾引起。堆栈框架按照以下标题行打印：

```
Stack (most recent call last):
```

这模仿了显示异常帧时使用的内容。Traceback (most recent call last):

第三个可选的关键字参数是*额外的*，它可以用来传递一个字典，该字典用于使用用户定义的属性填充为记录事件创建的LogRecord的__dict__。这些自定义属性可以随意使用。例如，它们可以合并到记录的消息中。例如：

```
FORMAT = '%(asctime)-15s %(clientip)s %(user)-8s %(message)s'
logging.basicConfig(format=FORMAT)
```

```
d = {'clientip': '192.168.0.1', 'user': 'fbloggs'}
logging.warning('Protocol problem: %s', 'connection reset', extra=d)
```

会打印如下内容：

```
2006-02-08 22:20:02,165 192.168.0.1 fbloggs  Protocol problem: connection reset
```

在通过了词典的按键*额外*不应由记录系统所使用的密钥冲突。（有关Formatter日志记录系统使用哪些密钥的更多信息，请参阅文档。）

如果您选择在记录的消息中使用这些属性，则需要谨慎操作。例如，在上面的例子中，Formatter已经用LogRecord属性字典中的'clientip'和'user'设置了一个格式字符串。如果缺少这些信息，将不会记录该消息，因为会出现字符串格式异常。所以在这种情况下，你总是需要用这些键传递*额外的*字典。

虽然这可能很烦人，但此功能旨在用于特殊情况下，例如多线程服务器，其中相同的代码在许多环境中执行，并且产生的有趣情况取决于此环境（如远程客户端IP地址和经过身份验证用户名，在上面的例子中）。在这种情况下，专门的 Formatters可能会与特定的Handlers 一起使用。

*：在3.2版本中的新*的*stack_info*加入参数。

logging.**info**(*msg*, **args*, ***kwargs*)

INFO在根记录器上记录具有级别的消息。参数被解释为debug()。

logging.**warning**(*msg*, **args*, ***kwargs*)

WARNING在根记录器上记录具有级别的消息。参数被解释为debug()。

> **注意：** 有一个warn功能完全相同的过时功能warning。如warn已弃用，请勿使用它 - 请warning改用。

logging.**error**(*msg*, **args*, ***kwargs*)

ERROR在根记录器上记录具有级别的消息。参数被解释为debug()。

logging.**critical**(*msg*, **args*, ***kwargs*)

CRITICAL在根记录器上记录具有级别的消息。参数被解释为debug()。

logging.**exception**(*msg*, **args*, ***kwargs*)

ERROR在根记录器上记录具有级别的消息。参数被解释为debug()。异常信息被添加到日志消息中。这个函数只能从异常处理程序中调用。

logging.**log**(*level*, *msg*, **args*, ***kwargs*)

在根记录器上记录具有级别*级别*的消息。其他参数解释为debug()。

> **注意：** 上述模块级别的便利功能（委托给根记录器）调用basicConfig()以确保至少有一个处理程序可用。因此，除非在线程启动*之前*至少有一个处理程序已添加到根日志记录程序中，否则不应在线程中使用它们，*也不*应在2.7.1和3.2之前的Python版本中使用它们。在Python的早期版本中，由于线程安全性的缺陷，这可能会（在极少数情况下）导

致处理程序被多次添加到根记录器，这可能会导致同一事件发生多个消息。basicConfig()

logging.disable(*lvl* = *CRITICAL*)

为所有记录器提供优先级高于记录器自身等级的重写级别*lvl*。当需要暂时限制整个应用程序的日志记录输出时，此功能可能很有用。其效果是禁用所有严重*级别*为*lvl*及以下的日志记录调用，以便如果使用INFO值调用它，则所有INFO和DEBUG事件都将被丢弃，而严重级别为WARNING及以上的则会根据记录器的有效水平。如果logging.disable(logging.NOTSET)被调用，它会有效地消除这个覆盖级别，以便日志输出再次取决于各个日志记录器的有效级别。

请注意，如果您定义的任何自定义日志记录级别高于CRITICAL（不建议这样做），则您将无法依赖*lvl*参数的默认值，但必须明确提供适当的值。

*在3.7版本中更改：*将*LVL*参数被默认为级别CRITICAL。有关此更改的更多信息，请参阅问题＃28524。

logging.addLevelName(*lvl*, *levelName*)

在内部字典*中将*级别*lvl*与文本级别名称相关*联*，该内部字典用于将数字级别映射到文本表示，例如在Formatter格式化消息时。这个功能也可以用来定义你自己的水平。唯一的约束是所有使用的级别都必须使用这个函数进行注册，级别应该是正整数，并且应该按照严重性的增加顺序增加。

注意： 如果您正在考虑定义自己的关卡，请参阅关于自定义关卡的部分。

logging.getLevelName(*lvl*)

返回日志级别*lvl*的文本表示。如果水平是预定义的级别之一CRITICAL，ERROR，WARNING，INFO或者DEBUG那么你得到相应的字符串。如果您使用addLevelName()名称关联*级别*，则返回与*lvl*关联的名称。如果传入一个对应于其中一个定义的级别的数值，则返回相应的字符串表示。否则，返回字符串'Level%s'%lvl。

注意： 级别是内部整数（因为它们需要在日志逻辑中进行比较）。此函数用于通过%(levelname)s格式说明符在格式化日志输出中显示的整数级别和级别名称之间进行转换（请参阅LogRecord属性）。

*版本3.4中的更改：*在*3.4*以前的Python版本中，此函数也可以传递文本级别，并返回该级别的相应数值。这种未记录的行为被认为是一个错误，并且在Python 3.4中被删除，但由于保留了向后兼容性而在3.4.2中恢复。

logging.makeLogRecord(*attrdict*)

创建并返回LogRecord其属性由*attrdict*定义的新实例。此函数对于获取LogRecord通过套接字发送的pickled 属性字典并LogRecord在接收端重新构造为实例很有用。

logging.basicConfig(***kwargs*)

通过创建StreamHandler缺省值Formatter并将其添加到根记录器来进行日志记录系统的基本配置。的功能debug()，info()，warning()，error()和critical()将调用basicConfig()自动如果没有处理程序为根记录器限定。

如果根记录器已经为它配置了处理程序，该函数不会执行任何操作。

> **注意：** 在其他线程启动之前，应该从主线程调用该函数。在2.7.1和3.2之前的Python版本中，如果此函数是从多个线程中调用的，则可能（极少数情况下）处理程序将不止一次添加到根记录器，从而导致意外的结果，如消息在日志中被复制。

以下关键字参数均受支持。

格式	描述
filename	指定使用指定的文件名而不是StreamHandler创建FileHandler。
filemode	指定打开文件的模式，如果指定了filename（如果文件模式未指定，则默认为'a'）。
format	为处理程序使用指定的格式字符串。
datefmt	使用指定的日期/时间格式。
style	如果format指定，则使用此样式作为格式字符串。'%'，'{'或'$'之一用于%格式化，str.format()或者 string.Template分别为'%'，如果未指定，则默认为'%'。
level	将根记录器级别设置为指定级别。
stream	使用指定的流来初始化StreamHandler。请注意，此参数与'文件名'不兼容 - 如果两者都存在，ValueError则会引发a。
handlers	如果指定，这应该是已经创建的处理程序的迭代器，以添加到根记录器。任何尚未设置格式化程序的处理程序将被分配此函数中创建的默认格式化程序。请注意，该参数与'filename'或'stream'不兼容 - 如果两者都存在，ValueError则会引发a。

*改变在3.2版本：*将style加入争论。

*改变在3.3版本：*将handlers加入争论。加入额外的检查，以赶上其中不相容参数指定的情况下（例如， handlers连同stream或filename，或stream 连同filename）。

logging.**shutdown**( )

通过刷新和关闭所有处理程序，通知日志记录系统执行有序关闭。这应该在应用程序退出时调用，在此调用之后不应再使用日志记录系统。

logging.**setLoggerClass**(*克拉斯*)

在实例化记录器时告诉日志记录系统使用类*klass*。该类应该定义__init__()为只需要一个名称参数，并且__init__()应该调用Logger.__init__()。在需要使用自定义记录器行为的应用程序实例化任何记录器之前，通常会调用此函数。

logging.**setLogRecordFactory**(*工厂*)

设置一个可调用的用于创建一个LogRecord。

参数：	**工厂** - 工厂可调用来实例化日志记录。

*版本3.2中的新*功能：此功能已提供，getLogRecordFactory()以便开发人员更好地控制如何LogRecord构建表示记录事件。

工厂有以下签名：

factory(name, level, fn, lno, msg, args, exc_info, func=None, sinfo=None, **kwargs)

名称：	记录器名称。
水平：	日志级别（数字）。
FN：	调用日志记录的文件的完整路径名。
LNO：	执行日志记录调用的文件中的行号。
味精：	日志消息。
ARGS：	日志消息的参数。
exc_info：	一个异常元组，或None。
FUNC：	调用日志记录调用的函数或方法的名称。
sinfo：	traceback.print_stack()显示调用层次结构的堆栈追溯（如由其提供）。
kwargs：	其他关键字参数。

11年6月16日。模块级属性

logging.lastResort

通过此属性可以使用“最后处理程序”。这是一个StreamHandler写入sys.stderr级别WARNING，用于在没有任何日志记录配置的情况下处理日志记录事件。最终的结果是只打印消息 sys.stderr。这取代了以前的错误消息，说“没有处理程序可以找到记录器XYZ”。如果您因某种原因需要更早的行为，lastResort可以设置为None。

3.2版本中的新功能

12年6月16日。与警告模块集成

该captureWarnings()功能可用于logging 与warnings模块集成。

logging.captureWarnings(*捕获*)

此功能用于通过登录和关闭来捕获警告。

如果*捕获*是True，warnings模块发出的警告将被重定向到日志记录系统。具体来说，将使用格式化警告并将warnings.formatwarning()结果字符串记录到名为'py.warnings'严重性为的记录器WARNING。

如果*捕获*是False，则警告重定向到日志记录系统将停止，并且警告将被重定向到它们的原始目的地（即，之前有效的那些目标captureWarnings(True)被调用）。

也可以看看:

模 logging.config

记录模块的配置API。

模 `logging.handlers`
记录模块附带的有用处理程序。

PEP 282 - 记录系统
描述此功能以包含在Python标准库中的建议。

原始的Python日志包
这是`logging`包的原始来源。该站点提供的软件包版本适用于Python 1.5.2,2.1.x和2.2.x，它们不包括`logging`标准库中的软件包。

16.7。logging.config- 记录配置

源代码： Lib / logging / config.py

本节介绍用于配置日志记录模块的API。

重要

此页面仅包含参考信息。有关教程，请参阅

- 基本教程
- 高级教程
- 记录食谱

16.7.1。配置功能

以下功能配置日志记录模块。它们位于logging.config模块中。它们的使用是可选的 - 您可以使用这些函数或通过调用主API（定义在logging其自身中）并定义在logging或中声明的处理程序 来配置日志记录模块logging.handlers。

logging.config.**dictConfig**(*config*)

从字典中获取日志配置。该字典的内容在 下面的配置字典模式中进行了描述。

如果配置期间遇到错误，则该函数将引发一个ValueError，TypeError，AttributeError 或ImportError与适当的描述性信息。以下是可能会引发错误的（可能不完整的）条件列表：

- A level不是字符串，或者是不与实际日志记录级别对应的字符串。
- 一个propagate不是布尔值的值。
- 一个没有相应目的地的ID。
- 在增量呼叫期间找到不存在的处理程序ID。
- 无效的记录器名称。
- 无法解析为内部或外部对象。

解析由DictConfigurator类执行，其构造函数传递用于配置的字典，并具有一个configure()方法。该logging.config模块具有dictConfigClass 最初设置为的可调用属性DictConfigurator。您可以用dictConfigClass您自己的合适实施来替换它的价值。

dictConfig()调用dictConfigClass传递指定的字典，然后调用configure()返回对象上的方法来使配置生效：

```
def dictConfig(config):
    dictConfigClass(config).configure()
```

例如，一个子类DictConfigurator可以DictConfigurator.__init__()自己调用__init__()，然后设置自定义前缀，这将在随后的 configure()调用中使用。dictConfigClass将被绑定到这个新的子类，然后dictConfig()可以完全按照默认的，未定制的状态进行调用。

3.2版本中的新功能

logging.config.**fileConfig**(*fname*, *defaults = None*, *disable_existing_loggers = True*)

从configparser格式文件中读取日志配置。文件的格式应如 配置文件格式中所述。该功能可以从应用程序中多次调用，允许最终用户从各种预先配置的配置中进行选择（如果开发人员提供了一种机制来呈现选项并加载所选配置）。

参数：	• **fname** - 一个文件名，或一个类似文件的对象，或者一个从中派生的实例RawConfigParser。如果RawConfigParser传递了一个 -derived实例，它将按原样使用。否则，Configparser实例化a，并从传入的对象读取配置fname。如果它有一个readline() 方法，它被认为是一个类似文件的对象并且正在读取read_file()；否则，它被假定为一个文件名并传递给 read()。 • **默认值** - 可以在此参数中指定要传递给ConfigParser的默认值。 • **disable_existing_loggers** - 如果指定为False，进行此调用时存在的记录器保持启用状态。默认是True因为这会以向后兼容的方式启用旧行为。这种行为是为了禁用任何现有的记录器，除非它们或其祖先在记录配置中明确命名。

*版本3.4中更改：*RawConfigParser现在将一个子类的实例接受为值fname。这有利于：

- 在日志配置只是整个应用程序配置的一部分的情况下使用配置文件。
- 使用从文件读取的配置，然后在使用应用程序（例如，基于命令行参数或运行时环境的其他方面）进行修改之前将其传递给该配置fileConfig。

logging.config.**listen**(*port = DEFAULT_LOGGING_CONFIG_PORT*, *verify = None*)

启动指定端口上的套接字服务器，并侦听新配置。如果没有指定端口，DEFAULT_LOGGING_CONFIG_PORT则使用该模块的默认值。日志记录配置将作为适合由dictConfig()或 处理的文件发送fileConfig()。返回Thread可以调用start()以启动服务器的实例，以及join()在适当时可以调用的实例。要停止服务器，请致电stopListening()。

的verify参数，如果指定的话，应该是一个可调用哪些应该验证跨过插座接收到的字节是否有效，并应被处理。这可以通过对通过套接字发送的内容进行加密和/或签名来完成，使得verify可调用对象可以执行签名验证和/或解密。该verify调用被称为带一个参数-在各插座接收到的字节-和应返回的字节进行处理，或者None以指示该字节应被丢弃。返回的字节可能与传入的字节相同（例如，只有验证完成时），或者它们可能完全不同（可能是在执行解密时）。

要将配置发送到套接字，请读入配置文件，并将其作为一串字节序列发送到套接字，后面是以二进制格式打包的四字节长度字符串。 struct.pack('>L', n)

注意： 由于部分配置已通过 eval()，因此使用此功能可能会使其用户面临安全风险。虽然该函数仅绑定到一个套接字上localhost，并且不接受来自远程计算机的连接，但有些情况下，不可信的代码可以在调用的进程帐户下运行 listen()。具体来说，如果进程调用listen()在多用户机器上运行，而用户不能相互信任，那么恶意用户可以安排在受害用户的进程中运行基本上任意代码，只需连接受害者 listen()套接字并发送一个配置，该配置运行攻击者希望在受害者进程中执行的任何代码。如果使用默认端口，这很容易实现，但即使使用不同的端口也不困难）。为避免发生这种情况的风险，请使用 verify参数listen()来防止应用无法识别的配置。

*改变在3.4版本：*将verify加入争论。

注意： 如果您想将配置发送给不禁用现有记录器的侦听器，则需要使用JSON格式进行配置，这将dictConfig()用于配置。这种方法允许你指定disable_existing_loggers为False在您发送的配置。

logging.config.**stopListening**()

停止通过呼叫创建的侦听服务器listen()。这通常在调用join()返回值 之前调用listen()。

16.7.2。配置字典模式

描述日志记录配置需要列出要创建的各种对象以及它们之间的连接; 例如，您可以创建一个名为'console'的处理程序，然后说名为'startup'的记录程序将其消息发送到'console'处理程序。这些对象不限于由logging 模块提供的对象，因为您可能会编写自己的格式化程序或处理程序类。这些类的参数可能还需要包含外部对象，如sys.stderr。描述这些对象和连接的语法在 下面的对象连接中定义。

16.7.2.1。Dictionary Schema

传递给的字典dictConfig()必须包含以下键：

- *版本* - 被设置为表示模式版本的整数值。目前唯一有效的值是1，但拥有这个键可以使模式发展，同时仍然保持向后兼容性。

所有其他键都是可选的，但如果存在的话，它们将按照下面的描述进行解释。在下面提到'配置字典'的所有情况下，将检查特殊'()'键以查看是否需要自定义实例化。如果是这样，下面用户定义对象中描述的机制 用于创建一个实例; 否则，上下文用于确定要实例化的内容。

- *格式化程序* - 相应的值将是一个字典，其中每个键是一个格式化程序ID，每个值都是一个字典，用于描述如何配置相应的Formatter实例。

  配置字典是搜索键format和datefmt （默认值None），这些用于构造一个 Formatter实例。

- *过滤器* - 相应的值将是一个字典，其中每个键是一个过滤器ID，每个值是一个字典，描述如何配置相应的过滤器实例。

  搜索配置字典中的键name（默认为空字符串），并用于构造logging.Filter 实例。

- *处理程序* - 相应的值将是一个字典，其中每个键是一个处理程序ID，每个值是一个字典，描述如何配置相应的Handler实例。

  在配置字典中搜索以下关键字：

  - class（强制）。这是处理程序类的完全限定名称。
  - level（可选的）。处理程序的级别。
  - formatter（可选的）。此处理程序的格式化程序的ID。
  - filters（可选的）。此处理程序的过滤器的id列表。

所有*其他*键都作为关键字参数传递给处理程序的构造函数。例如，给出片段：

```
handlers:
  console:
    class : logging.StreamHandler
    formatter: brief
    level   : INFO
    filters: [allow_foo]
    stream  : ext://sys.stdout
  file:
    class : logging.handlers.RotatingFileHandler
    formatter: precise
    filename: logconfig.log
    maxBytes: 1024
    backupCount: 3
```

带id的处理程序`console`被实例化为a `logging.StreamHandler` ，`sys.stdout`用作基础流。带有id的处理程序`file`被实例化为一个 `logging.handlers.RotatingFileHandler`带有关键字参数的 。 `filename='logconfig.log', maxBytes=1024, backupCount=3`

- *记录器* - 相应的值将是一个字典，其中每个键是一个记录器名称，每个值是一个字典，用于描述如何配置相应的记录器实例。

  在配置字典中搜索以下关键字：

  - `level`（可选的）。记录器的级别。
  - `propagate`（可选的）。记录器的传播设置。
  - `filters`（可选的）。此记录器的过滤器的ID列表。
  - `handlers`（可选的）。此记录器的处理程序的ID列表。

  指定的记录器将根据指定的级别，传播，过滤器和处理程序进行配置。

- *根* - 这将是根记录器的配置。处理配置将与任何记录器一样，只是该`propagate`设置不适用。

- *增量* - 是否将配置解释为现有配置的增量配置。该值默认为 `False` ，这意味着指定的配置会使用与现有`fileConfig()` API 使用的语义相同的语义替换现有配置。

  如果指定的值是`True` ，则按照“ 增量配置 ”一节中的说明处理配置。

- *disable_existing_loggers* - 是否禁用任何现有的记录器。该设置镜像中的相同名称的参数`fileConfig()`。如果不存在，则此参数默认为`True`。如果*增量*为，则此值将被忽略`True`。

16.7.2.2。增量配置

为增量配置提供完全的灵活性是很困难的。例如，因为诸如过滤器和格式化程序之类的对象是匿名的，所以一旦设置了配置，在扩充配置时就不可能引用这样的匿名对象。

此外，一旦配置设置完成，就没有一种令人信服的情况，即在运行时任意更改记录器，处理程序，过滤器和格式化程序的对象图; 记录器和处理程序的详细程度可以通过设置级别来控制（并且在记录器的情况下，传播标志）。在多线程环境中，以安全的方式任意更改对象图是有问题的; 虽然并非不可能，但这些好处并不值得它增加实施的复杂性。

因此，当incremental配置字典的键存在并且是True ，该系统将完全忽略任何formatters和filters条目，并且仅处理与level 在设置handlers项，并且level与 propagate在设置loggers和root条目。

在配置dict中使用值可以让配置作为pickle字典通过线路发送到套接字侦听器。因此，长时间运行的应用程序的日志详细程度可以随着时间而改变，而不需要停止并重新启动应用程序。

16.7.2.3。对象连接

该模式描述了一组记录对象 - 记录器，处理程序，格式化程序，过滤器 - 它们在对象图中彼此连接。因此，模式需要表示对象之间的连接。例如，说一旦配置完毕，特定的记录器就会附加一个特定的处理程序。为了讨论的目的，我们可以说记录器代表源，而处理器则是两者之间连接的目的地。当然，在配置的对象中，这由记录器持有对处理程序的引用来表示。在配置字典中，这是通过给每个目标对象一个明确标识它的id，然后在源对象的配置中使用id来指示源和目标对象之间存在具有该id的连接来完成的。

因此，例如，请考虑以下YAML代码片段：

```
formatters:
  brief:
    # configuration for formatter with id 'brief' goes here
  precise:
    # configuration for formatter with id 'precise' goes here
handlers:
  h1: #This is an id
   # configuration of handler with id 'h1' goes here
   formatter: brief
  h2: #This is another id
   # configuration of handler with id 'h2' goes here
   formatter: precise
loggers:
  foo.bar.baz:
    # other configuration for logger 'foo.bar.baz'
    handlers: [h1, h2]
```

（注意：YAML在这里使用，因为它比字典的等效Python源代码更易读。）

记录器的ID是记录器名称，它将以编程方式用于获取对这些记录器的引用，例如foo.bar.baz。为格式化器和过滤器的ID可以是任何字符串值（如brief，precise上述的），并且它们是瞬时的，因为它们仅用于处理配置字典有意义并用于确定对象之间的连接，并且在任何地方当配置呼叫是不保持做完了。

上面的代码片断表明，名为logger的foo.bar.baz应该有两个处理程序，它们由处理程序ID h1和描述h2。格式化程序h1是由id描述的 brief，格式化程序h2是由id描述的 precise。

16.7.2.4。用户定义的对象

该模式支持处理程序，过滤器和格式化程序的用户定义对象。（对于不同的实例，记录器不需要具有不同的类型，因此在此配置模式中不支持用户定义的记录器类。）

要配置的对象由详细描述其配置的字典来描述。在某些地方，日志系统将能够从上下文中推断出一个对象是如何实例化的，但是当一个用户定义的对象被实例化时，系统将不知道如何执行此操作。为了为用户定义的对象实例化提供完全的灵活性，用户需要提供一个“工厂” - 一个可调用的对象，它用配置字典调用，并返回实例化的对象。这是通过一个绝对的进口通道来指示工厂在专用钥匙下可用'()'。这里有一个具体的例子：

```
formatters:
  brief:
    format: '%(message)s'
  default:
    format: '%(asctime)s %(levelname)-8s %(name)-15s %(message)s'
    datefmt: '%Y-%m-%d %H:%M:%S'
  custom:
      (): my.package.customFormatterFactory
      bar: baz
      spam: 99.9
      answer: 42
```

上面的YAML代码片段定义了三个格式化程序。第一个id brief是logging.Formatter带指定格式字符串的标准实例。第二个，使用id default，具有更长的格式，并且还明确定义了时间格式，并将导致logging.Formatter使用这两个格式字符串进行初始化。示于Python源形式中，brief和default 格式化器具有配置子字典：

```
{
  'format' : '%(message)s'
}
```

和：

```
{
  'format' : '%(asctime)s %(levelname)-8s %(name)-15s %(message)s',
  'datefmt' : '%Y-%m-%d %H:%M:%S'
}
```

因为这些字典不包含特殊键'()'，所以从上下文推断实例化：因此，logging.Formatter创建了标准实例。第三个格式化器的配置子字典（id custom）为：

```
{
  '()' : 'my.package.customFormatterFactory',
  'bar' : 'baz',
  'spam' : 99.9,
  'answer' : 42
}
```

这包含特殊的键'()'，这意味着用户定义的实例化是想要的。在这种情况下，将使用指定的工厂可调用函数。如果它是一个实际的可调用对象，它将被直接使用 - 否则，如果指定了一个字符串（如示例中那样），则实际的可调用对象将使用普通导入机制进行定位。将使用配置子字典中的**其余**项作为关键字参数调用可调用对象。在上面的例子中，带有id的格式化程序custom将被假定为由该调用返回：

```
my.package.customFormatterFactory(bar='baz', spam=99.9, answer=42)
```

该键'()'已被用作特殊键，因为它不是有效的关键字参数名称，因此不会与调用中使用的关键字参数的名称冲突。这'()'也是一个助记符，相应的值是可调用的。

16.7.2.5。访问外部对象

例如，有时配置需要引用配置外部的对象sys.stderr。如果使用Python代码构造配置字典，这很简单，但是当通过文本文件（例如JSON，YAML）提供配置时会出现问题。在文本文件中，没有标准的方法可以区分sys.stderr文字字符串'sys.stderr'。为了便于区分，配置系统在字符串值中查找某些特殊的前缀并专门处理它们。例如，如果文字字符串'ext://sys.stderr'作为配置中的值提供，那么ext://将剥离文本字符串 ，并使用普通导入机制处理值的其余部分。

这种前缀的处理以类似于协议处理的方式完成：有一种通用机制来查找与正则表达式相匹配的前缀^(?P<prefix>[a-z]+)://(?P<suffix>.*)$ ，如果该前缀prefix被识别，suffix则以前缀相关的方式处理该前缀，并且处理将替换字符串值。如果前缀未被识别，则字符串值将保持原样。

16.7.2.6。访问内部对象

除了外部对象外，有时还需要引用配置中的对象。这将由配置系统隐式地完成它所知道的事情。例如，字符串值'DEBUG'用于level在一个记录器或处理程序将自动转换为值logging.DEBUG，并且handlers，filters和formatter条目将一个对象ID和解析为相应的目标对象。

但是，对于logging模块未知的用户定义的对象，需要更通用的机制。例如，考虑一下logging.handlers.MemoryHandler，这需要target另一个处理程序委托给它的参数。由于系统已经知道这个类，因此在配置中，给定的target只需要是相关目标处理程序的对象标识，系统将从该标识解析为处理程序。但是，如果用户定义了my.package.MyHandler具有alternate处理程序的用户，那么配置系统不会知道alternate所指的处理程序。为了迎合这一点，一个通用的解析系统允许用户指定：

```
handlers:
  file:
    # configuration of file handler goes here

  custom:
    (): my.package.MyHandler
    alternate: cfg://handlers.file
```

文字字符串'cfg://handlers.file'将以类似方式解析为带有ext://前缀的字符串，但查看配置本身而不是导入名称空间。该机制允许通过点或索引进行访问，方式与提供的类似str.format。因此，给出以下片段：

```
handlers:
  email:
    class: logging.handlers.SMTPHandler
    mailhost: localhost
    fromaddr: my_app@domain.tld
    toaddrs:
      - support_team@domain.tld
```

```
    - dev_team@domain.tld
  subject: Houston, we have a problem.
```

在配置方面，该字符串'cfg://handlers'将解析为与关键的字典handlers，字符串'cfg://handlers.email 将解析为与关键的字典email中的handlers字典，等等。该字符串'cfg://handlers.email.toaddrs[1] 将解析为'dev_team.domain.tld'并且该字符串'cfg://handlers.email.toaddrs[0]'将解析为该值'support_team@domain.tld'。subject可以使用'cfg://handlers.email.subject'或等价地访问该值'cfg://handlers.email[subject]'。只有当密钥包含空格或非字母数字字符时，才需要使用后一种形式。如果索引值仅包含十进制数字，则将尝试使用相应的整数值进行访问，如果需要则返回字符串值。

给定一个字符串cfg://handlers.myhandler.mykey.123，这将解决config_dict['handlers']['myhandler']['mykey']['123']。如果字符串被指定为cfg://handlers.myhandler.mykey[123]，系统将尝试从中检索值config_dict['handlers']['myhandler']['mykey'][123]，config_dict['handlers']['myhandler']['mykey']['123']如果失败则返回。

16.7.2.7。导入解析和自定义导入器

默认情况下，导入解析使用内置__import__()函数进行导入。你可能想用自己的进口机制，以取代这样的：如果是这样，你可以替换importer的属性DictConfigurator或它的超的BaseConfigurator类。但是，由于通过描述符从类访问函数的方式，您需要小心。如果您使用Python可调用来执行导入，并且您希望在类级别而不是实例级别定义它，则需要将其包装staticmethod()。例如：

```
from importlib import import_module
from logging.config import BaseConfigurator

BaseConfigurator.importer = staticmethod(import_module)
```

staticmethod()如果您在配置器*实例*上设置可调用的导入，则不需要换行。

16.7.3。配置文件格式

理解的配置文件格式fileConfig()基于configparser功能。该文件必须包含称为的部分[loggers]，[handlers]并按[formatters]名称标识文件中定义的每种类型的实体。对于每个这样的实体，都有一个单独的部分来标识该实体如何配置。因此，对于log01在该[loggers]部分中命名的记录器，相关的配置细节保存在一个部分中[logger_log01]。同样，hand01在该[handlers]部分中调用的处理程序将其配置保存在一个名为section的部分中[handler_hand01]，而在该部分中调用的格式化程序将form01在被调用的[formatters]部分中指定其配置[formatter_form01]。根记录器配置必须在称为的部分中指定[logger_root]。

注意： 该fileConfig()API比旧dictConfig()的API，并且不提供功能覆盖记录的某些方面。例如，您不能使用配置Filter对象，这些对象提供超出简单整数级别的消息过滤fileConfig()。如果您需要Filter在日志记录配置中拥有实例，则需要使用dictConfig()。请注意，将来会增加对配置功能的增强功能dictConfig()，所以在方便的时候考虑过渡到这个更新的API是值得的。

下面给出了文件中这些部分的示例。

```
[loggers]
keys=root,log02,log03,log04,log05,log06,log07

[handlers]
keys=hand01,hand02,hand03,hand04,hand05,hand06,hand07,hand08,hand09

[formatters]
keys=form01,form02,form03,form04,form05,form06,form07,form08,form09
```

根记录器必须指定一个级别和一系列处理程序。根记录器部分的一个例子如下。

```
[logger_root]
level=NOTSET
handlers=hand01
```

该level条目可以是一个或 。仅对于根记录器而言，意味着将记录所有消息。级别值在 包的名称空间的上下文中使用。DEBUG, INFO, WARNING, ERROR, CRITICALNOTSETNOTSETeval()logging

该handlers条目是逗号分隔的处理程序名称列表，必须出现在该[handlers]部分中。这些名称必须出现在该 [handlers]部分中，并在配置文件中具有相应的部分。

对于根记录器以外的记录器，需要一些附加信息。以下示例说明了这一点。

```
[logger_parser]
level=DEBUG
handlers=hand01
propagate=1
qualname=compiler.parser
```

除了如果指定了非root用户记录器的级别以外，系统level和handlers条目将被解释为根记录器，但NOTSET系统将在层次结构的更高层次上咨询记录器以确定记录器的有效级别。所述propagate项被设置为1，以指示消息必须越往上记录器层次结构传播到处理程序从此记录，或0，以指示消息**不**传播到处理程序向上的层次结构。该 qualname条目是记录器的分层通道名称，也就是说应用程序用来获取记录器的名称。

指定处理程序配置的部分由以下示例。

```
[handler_hand01]
class=StreamHandler
level=NOTSET
formatter=form01
args=(sys.stdout,)
```

该class条目指示处理程序的类（如通过eval() 在logging包的命名空间）。这level被解释为记录器，并被NOTSET认为是'记录一切'。

该 formatter 条目表示该处理程序的格式化程序的键名。如果为空，logging._defaultFormatter则使用默认格式化程序（ ）。如果指定了名称，则它必须出现在该[formatters]部分中，并在配置文件中具有相应的部分。

args当eval()在logging 包名称空间的上下文中使用时，该条目是处理程序类的构造函数的参数列表。请参阅相关处理程序的构造函数或下面的示例，查看典型条目的构造方式。

```
[handler_hand02]
class=FileHandler
level=DEBUG
formatter=form02
args=('python.log', 'w')

[handler_hand03]
class=handlers.SocketHandler
level=INFO
formatter=form03
args=('localhost', handlers.DEFAULT_TCP_LOGGING_PORT)

[handler_hand04]
class=handlers.DatagramHandler
level=WARN
formatter=form04
args=('localhost', handlers.DEFAULT_UDP_LOGGING_PORT)

[handler_hand05]
class=handlers.SysLogHandler
level=ERROR
formatter=form05
args=(('localhost', handlers.SYSLOG_UDP_PORT), handlers.SysLogHandler.LOG_USER)

[handler_hand06]
class=handlers.NTEventLogHandler
level=CRITICAL
formatter=form06
args=('Python Application', '', 'Application')

[handler_hand07]
class=handlers.SMTPHandler
level=WARN
formatter=form07
args=('localhost', 'from@abc', ['user1@abc', 'user2@xyz'], 'Logger Subject')

[handler_hand08]
class=handlers.MemoryHandler
level=NOTSET
formatter=form08
target=
args=(10, ERROR)

[handler_hand09]
class=handlers.HTTPHandler
level=NOTSET
formatter=form09
args=('localhost:9022', '/log', 'GET')
```

指定格式化程序配置的部分如下所示。

```
[formatter_form01]
format=F1 %(asctime)s %(levelname)s %(message)s
```

```
datefmt=
class=logging.Formatter
```

该format条目是全盘格式化字符串，datefmt条目是strftime()兼容的日期/时间格式字符串。如果为空，则该包将替代ISO8601格式的日期/时间，这几乎等同于指定日期格式字符串。这种格式还指定了毫秒，这些毫秒附加到使用上述格式字符串的结果，并带有逗号分隔符。这种格式的示例时间是 。'%Y-%m-%d %H:%M:%S' 2003-01-23 00:29:50,411

该class条目是可选的。它指示格式化程序类的名称（作为虚线模块和类名称）。此选项对于实例化子Formatter类很有用 。 Formatter可以以扩展或压缩格式呈现异常回溯的子类 。

注意: 由于使用eval()如上所述，使用listen()通过套接字发送和接收配置会导致潜在的安全风险。风险限于多个不具有相互信任的用户在同一台机器上运行代码的情况; 请参阅listen()文档以获取更多信息。

也可以看看:

模 logging
记录模块的API参考。

模 logging.handlers
记录模块附带的有用处理程序。

16.8。 logging.handlers- 记录处理程序

源代码： Lib / logging / handlers.py

包中提供了以下有用的处理程序。需要注意的是三个处理程序（ StreamHandler ， FileHandler 和 NullHandler ）实际上是在定义logging模块本身，而是一直在与其他处理一起记录在这里。

重要

此页面仅包含参考信息。有关教程，请参阅

- 基本教程
- 高级教程
- 记录食谱

16.8.1。StreamHandler中

StreamHandler位于核心logging包中的类将日志输出发送到诸如*sys.stdout*，*sys.stderr*或任何类似文件的对象（或者更确切地说，任何支持的对象write()和flush()方法）的流。

class logging.StreamHandler（*stream = None* ）

返回类的新实例StreamHandler。如果指定了*流*，实例将使用它来记录输出; 否则， 将使用*sys.stderr*。

emit（*记录*）

如果指定了格式化程序，则用它来格式化记录。然后记录被写入带有终止符的流中。如果存在异常信息，则使用traceback.print_exception()并附加到流中进行格式化。

flush（）

通过调用其flush()方法刷新流。请注意，该 close()方法是继承的Handler ，因此没有输出，所以flush()有时可能需要显式调用。

*改变在3.2版本：*在StreamHandler类现在有一个terminator属性，默认值'\n'，这是写格式记录到一个流时使用的终止符。如果你不想这个换行符终止，你可以设置处理程序实例的terminator属性为空字符串。在早期版本中，终结符被硬编码为'\n'。

16.8.2。的FileHandler

的FileHandler类，位于核心logging包，发送日志输出到磁盘文件。它继承了输出功能StreamHandler。

class logging.FileHandler（*filename*，*mode ='a'*，*encoding = None*，*delay = False* ）

返回类的新实例FileHandler。指定的文件将被打开并用作日志记录的流。如果没有指定*模式*，'a'则使用。如果*编码*不是None ，则它用于使用该编码打开文件。如果*延迟*是真的，则文件打开被推迟到第一次呼叫emit()。默认情况下，文件无限增长。

*在版本3.6中更改：*与字符串值一样，Path对于*filename*参数也接受对象。

close（）

关闭文件。

emit（*记录*）
将记录输出到文件。

16.8.3。NullHandler

版本3.1中的新功能。

该NullHandler级，位于核心logging包，并没有做任何的格式或输出。它本质上是一个供图书馆开发人员使用的“无操作”处理程序。

*类*logging.NullHandler
返回类的新实例NullHandler。

emit（*记录*）
这种方法什么都不做。

handle（*记录*）
这种方法什么都不做。

createLock（）
此方法返回None锁，因为没有访问需要序列化的底层I / O。

有关如何使用的更多信息， 请参阅配置日志记录库NullHandler。

16.8.4。WatchedFileHandler

这个WatchedFileHandler类位于logging.handlers 模块中，FileHandler它监视它正在登录的文件。如果文件发生更改，则会使用文件名关闭并重新打开。

由于使用诸如执行日志文件旋转的*newsyslog*和 *logrotate*等程序，文件更改可能会发生。这个处理程序，打算在Unix / Linux下使用，监视该文件以查看自上次发出后是否发生了更改。（如果文件的设备或inode已更改，则认为文件已更改。）如果文件已更改，则旧文件流将关闭，并打开文件以获取新流。

这个处理程序不适合在Windows下使用，因为在Windows下打开日志文件不能被移动或重命名 - 日志打开带有排它锁的文件 - 因此不需要这样的处理程序。此外， Windows下不支持*ST_INO*；stat()始终为此值返回零。

class logging.handlers.WatchedFileHandler（*filename*，*mode ='a'*，*encoding = None*，*delay = False*）
返回类的新实例WatchedFileHandler。指定的文件将被打开并用作日志记录的流。如果没有指定*模式*，'a'则使用。如果*编码*不是None，则它用于使用该编码打开文件。如果*延迟*是真的，则文件打开被推迟到第一次呼叫emit()。默认情况下，文件无限增长。

*在版本3.6中更改：*与字符串值一样，Path对于*filename*参数也接受对象。

reopenIfNeeded（ ）

检查文件是否已更改。如果有，则现有流将被刷新并关闭，并且文件将再次打开，通常作为将记录输出到文件的先导。

3.6版本中的新功能。

emit（ *记录* ）

将记录输出到文件，但首先调用reopenIfNeeded()重新打开文件，如果它已更改。

16.8.5。BaseRotatingHandler

的BaseRotatingHandler类，位于在logging.handlers 模块，是用于旋转文件处理程序基类，RotatingFileHandler和TimedRotatingFileHandler。你不需要实例化这个类，但它有你需要覆盖的属性和方法。

class logging.handlers. BaseRotatingHandler（ *filename* ， *mode* ， *encoding* = *None* ， *delay* = *False* ）

参数如下FileHandler。属性是：

namer

如果此属性设置为可调用，则该rotation_filename() 方法委托给此可调用对象。传递给可调用对象的参数是传递给它们的参数rotation_filename()。

> **注意：** namer函数在翻转过程中被调用了很多次，所以它应该尽可能的简单和快速。每次给定输入时它也应该返回相同的输出，否则翻转行为可能无法按预期工作。

3.3版本的新功能

rotator

如果此属性设置为可调用，则该rotate()方法委托给此可调用对象。传递给可调用对象的参数是传递给它们的参数rotate()。

3.3版本的新功能

rotation_filename（ *default_name* ）

旋转时修改日志文件的文件名。

这提供了一个自定义的文件名可以提供。

默认实现调用处理程序的'namer'属性，如果它是可调用的，则将默认名称传递给它。如果该属性不可调用（默认为None），则该名称将返回原样。

参数：	**default_name** - 日志文件的默认名称。

3.3版本的新功能

rotate（ *source* ， *dest* ）

旋转时，旋转当前日志。

默认实现调用处理程序的'rotator'属性，如果它是可调用的，则将source和dest参数传递给它。如果该属性不可调用（默认为None），则只将该源重命名为目标。

参数：	• **源** - 源文件名。这通常是基本文件名，例如'test.log'。 • **dest** - 目标文件名。通常这是源代码的旋转方式，例如'test.log.1'。

3.3版本的新功能

这些属性存在的原因是为了节省您的子类 - 您可以对RotatingFileHandler和的 实例使用相同的可调用元素TimedRotatingFileHandler。如果namer或rotator可调用引发一个异常，这将在emit()调用期间以与任何其他异常相同的方式处理，即通过handleError()处理程序的方法处理。

如果您需要对旋转处理进行更重大的更改，则可以覆盖这些方法。

有关示例，请参阅使用旋转器和编号器来自定义日志旋转处理。

16.8.6。RotatingFileHandler

RotatingFileHandler位于logging.handlers 模块中的类支持磁盘日志文件的旋转。

class logging.handlers.RotatingFileHandler（*filename*，*mode ='a'*，*maxBytes = 0*，*backupCount = 0*，*encoding = None*，*delay = False* ）

返回类的新实例RotatingFileHandler。指定的文件将被打开并用作日志记录的流。如果没有指定*模式*，'a'则使用。如果*编码*不是None，则它用于使用该编码打开文件。如果*延迟*是真的，则文件打开被推迟到第一次呼叫emit()。默认情况下，文件无限增长。

您可以使用*maxBytes*和*backupCount*值来允许文件以 预定大小*翻转*。当即将超过大小时，文件将被关闭，并且将静默打开一个新文件以进行输出。只要当前日志文件的长度接近*maxBytes*，就会发生翻转; 但是如果*maxBytes*或*backupCount*中的任一个 为零，则不会发生翻转，因此您通常要将*backupCount*设置为至少1，并且具有非零的*maxBytes*。当*backupCount*非零时，系统将通过将扩展名'.1'，'.2'等附加到文件名来保存旧的日志文件。例如，使用*backupCount* 5和的基本文件名app.log，你会得到app.log， app.log.1，app.log.2，达app.log.5。正在写入的文件始终是app.log。当这个文件被填满时，它被关闭并重新命名为app.log.1，如果文件app.log.1， app.log.2等存在，那么它们被重命名为app.log.2， app.log.3分别等。

*在版本3.6中更改：*与字符串值一样，Path对于*filename*参数也接受对象。

doRollover（）

如上所述进行翻车。

emit（*记录*）

如前所述，将记录输出到文件，以适应翻滚。

16.8.7。TimedRotatingFileHandler

该TimedRotatingFileHandler级，位于在 logging.handlers模块，支持以一定的时间间隔的盘日志文件旋转。

class logging.handlers.TimedRotatingFileHandler（*filename*，*when ='h'*，*interval = 1*，*backupCount = 0*，*encoding = None*，*delay = False*，*utc = False*，*atTime = None* ）

返回类的新实例TimedRotatingFileHandler。指定的文件将被打开并用作日志记录的流。在旋转它也设置文件名后缀。旋转基于时间和 *间隔*的乘积而*发生*。

您可以使用*when*来指定*间隔*的类型。可能值列表如下。请注意，它们不区分大小写。

值	**间隔类型**	**如果/如何*atTime*使用**
'S'	秒	忽视
'M'	分钟	忽视
'H'	小时	忽视
'D'	天	忽视
'W0'-'W6'	平日（0 =星期一）	用于计算初始翻转时间
'midnight'	如果*atTime*未指定，则在午夜 *翻车*，否则在*atTime*时间	用于计算初始翻转时间

当使用以工作日为基础的轮换时，星期一指定'W0'，星期二指定'W1'，星期日指定'W6'。在这种情况下，不会使用为*时间间隔*传递的值 。

系统将通过将扩展名附加到文件名来保存旧的日志文件。扩展名基于日期和时间，使用strftime格式 %Y-%m-%d_%H-%M-%S或其前导部分，具体取决于翻转间隔。

第一次计算下一个翻转时间（创建处理程序时），现有日志文件的最后修改时间或当前时间用于计算下一次旋转的发生时间。

如果*utc*参数为true，则将使用UTC中的时间; 否则使用本地时间。

如果*backupCount*不为零，则至多*backupCount*文件将被保留，并且如果在发生滚动时会创建更多，则会删除最旧的一个。删除逻辑使用间隔来确定要删除哪些文件，因此更改间隔可能会使旧文件处于闲置状态。

如果*延迟*是真的，则文件打开被推迟到第一次呼叫 emit()。

如果*atTime*不是None，则必须是一个datetime.time指定发生翻滚发生的时间的实例，对于翻转设置为“在午夜”或“在特定工作日”发生的情况。请注意，在这些情况下，*atTime*值有效用于计算*初始* 翻转，并且后续翻转将通过正常间隔计算进行计算。

注意： 当初始化处理程序时，计算初始翻转时间。只有在发生翻转时才会计算后续翻转时间，并且只有在发出输出时才会发生翻转。如果不考虑这一点，可能会导致一些混淆。例如，如果设置了“每分钟”的时间间隔，那并不意味着您将始终看到日志文件的分隔时间（在文件名中）如果在应用程序执行期间，记录输出每分钟产生一次的频率更高，*那么*您可能会看到时间间隔为一分钟的日志文件。另一方面，如果记录消息每五分钟只输出一次（比如说），那么文件时间就会出现缺口，与没有输出（因此不发生翻转）的分钟相对应。

*在版本3.4中更改：*atTime参数已添加。

*在版本3.6中更改：*与字符串值一样，Path对于*filename*参数也接受对象。

doRollover（）
如上所述进行翻车。

emit（*记录*）
将记录输出到文件，以适应上述的翻转。

16.8.8。的SocketHandler

SocketHandler位于logging.handlers模块中的类将日志记录输出发送到网络套接字。基类使用TCP套接字。

*类*logging.handlers.SocketHandler（*主机*，*端口*）
返回SocketHandler该类的新实例，用于与其地址由*主机*和*端口*给定的远程计算机进行通信。

*版本3.4中更改：*如果port指定为None，则使用中的值创建Unix域套接字host- 否则将创建TCP套接字。

close（）
关闭插座。

emit（）
Pickles记录的属性字典并以二进制格式将其写入套接字。如果套接字发生错误，则以静默方式丢弃数据包。如果连接先前丢失，请重新建立连接。要将接收端的记录解开为a LogRecord，请使用该makeLogRecord()功能。

handleError（）
处理期间发生的错误emit()。最可能的原因是连接丢失。关闭套接字，以便我们可以重试下一个事件。

makeSocket（）
这是一种工厂方法，它允许子类定义他们想要的确切类型的套接字。默认实现创建一个TCP套接字（socket.SOCK_STREAM）。

makePickle（*记录*）
使用长度前缀以二进制格式浸泡记录的属性字典，并将其返回，以便通过套接字进行传输。

请注意，泡菜并不完全安全。如果您担心安全问题，则可能需要重写此方法以实现更安全的机制。例如，您可以使用HMAC签署酱菜，然后在接收端验证它们，或者您也可以在接收端禁用取消全局对象。

send（*包*）
发送一个pickled字符串*包*到套接字。此功能允许在网络繁忙时发生部分发送。

createSocket（）

尝试创建一个套接字; 失败时，使用指数退避算法。在初始失败时，处理程序将放弃尝试发送的消息。当后续消息由同一个实例处理时，它将不会尝试连接，直到一段时间过去。默认参数是初始延迟时间为1秒，如果在延迟之后仍然无法建立连接，则处理程序将每次最多延迟两倍延迟时间，最长为30秒。

此行为由以下处理程序属性控制：

- retryStart （初始延迟，默认为1.0秒）。
- retryFactor （乘数，默认为2.0）。
- retryMax （最大延迟时间，默认为30.0秒）。

这意味着，如果远程监听器在处理器被使用*后*启动，则可能会丢失消息（因为处理器甚至不会在延迟过去之前尝试连接，而只是在延迟期间悄悄丢弃消息）。

16.8.9。DatagramHandler

DatagramHandler位于logging.handlers 模块中的类继承自SocketHandler支持通过UDP套接字发送日志记录消息。

*类*logging.handlers.DatagramHandler（*主机*，*端口*）

返回DatagramHandler该类的新实例，用于与其地址由*主机*和*端口*给定的远程计算机进行通信。

*版本3.4中更改：*如果port指定为None ，则使用中的值创建Unix域套接字host- 否则将创建UDP套接字。

emit（）

Pickles记录的属性字典并以二进制格式将其写入套接字。如果套接字发生错误，则以静默方式丢弃数据包。要将接收端的记录解开为a LogRecord ，请使用该makeLogRecord() 功能。

makeSocket（）

工厂方法在SocketHandler这里被覆盖以创建一个UDP套接字（ socket.SOCK_DGRAM ）。

send（*s* ）

发送pickled字符串到套接字。

10年8月16日。SysLogHandler

的SysLogHandler类，位于在logging.handlers模块，支持发送记录消息到远程或本地Unix系统日志。

class logging.handlers.SysLogHandler（*address = ('localhost'，SYSLOG_UDP_PORT)*，*facility = LOG_USER*，*socktype = socket.SOCK_DGRAM* ）

返回的新实例SysLogHandler打算与远程Unix机器，其地址是通过提供给通信类*地址*的形式的元组。如果没有指定*地址*， 则使用。该地址用于打开套接字。提供元组的另一种方法是

将地址作为字符串提供，例如'/ dev / log'。在这种情况下，使用Unix域套接字将消息发送到系统日志。如果*设施*没有指定，则使用。打开的套接字类型取决于 *socktype*参数，该参数默认为，因此会打开一个UDP套接字。要打开TCP套接字（用于更新的syslog守护进程，如 rsyslog ），请指定值。 (host, port)('localhost', 514)(host, port)LOG_USERsocket.SOCK_DGRAMsocket.SOCK_STREAM

请注意，如果您的服务器不在UDP端口514上侦听，则 SysLogHandler可能看起来不起作用。在这种情况下，请检查您应该使用域套接字的地址 - 它依赖于系统。例如，在Linux上通常是'/ dev / log'，但在OS / X上它是'/ var / run / syslog'。您需要检查您的平台并使用适当的地址（如果您的应用程序需要在多个平台上运行，您可能需要在运行时检查该平台）。在Windows上，你几乎不得不使用UDP选项。

*在版本3.2中更改：*添加了*socktype*。

close ( )

关闭到远程主机的套接字。

emit (*记录*)

记录被格式化，然后发送到系统日志服务器。如果存在异常信息，*则不*会将其发送到服务器。

在3.2.1版本中更改：（参见：BPO-12168 ）。在早期版本中，发送到syslog守护进程的消息总是用NULL字节终止，因为这些后台程序的早期版本预期NUL终止的消息-尽管它不是在相关规范（RFC 5424 ）中。这些守护进程的更新版本并不期望NUL字节，但如果它在那里则将其剥离，甚至更近的守护进程（更紧密地遵循RFC 5424 ）通过NUL字节作为消息的一部分。

为了在面对所有这些不同的守护进程行为时更轻松地处理系统日志消息，通过使用类级属性可以配置NUL字节的附加配置 append_nul。这默认为True（保留现有的行为），但可以设置为False在SysLogHandler实例上，以便该实例*不*添加NUL终止符。

在3.3版本中进行了更改：（请参阅：bpo-12419。 ）在早期版本中，没有用于识别消息来源的“ident”或“tag”前缀的功能。现在可以使用类级别属性来指定它，默认为 ""保留现有行为，但可以在SysLogHandler实例上重写该实例，以便该实例将ident识别为每个处理的消息。请注意，提供的ident必须是文本，而不是字节，并且与消息完全相同。

encodePriority (*设施*, *优先*)

将设施和优先级编码为一个整数。您可以传递字符串或整数 - 如果传递字符串，则使用内部映射字典将它们转换为整数。

符号LOG_值SysLogHandler在sys/syslog.h头文件中定义和镜像定义的值。

优先级

名称（字符串）	符号价值
alert	LOG_ALERT
crit 要么 critical	LOG_CRIT

名称（字符串）	符号价值
debug	LOG_DEBUG
emerg 要么 panic	LOG_EMERG
err 要么 error	LOG_ERR
info	LOG_INFO
notice	LOG_NOTICE
warn 要么 warning	LOG_WARNING

设备

名称（字符串）	符号价值
auth	LOG_AUTH
authpriv	LOG_AUTHPRIV
cron	LOG_CRON
daemon	LOG_DAEMON
ftp	LOG_FTP
kern	LOG_KERN
lpr	LOG_LPR
mail	LOG_MAIL
news	LOG_NEWS
syslog	LOG_SYSLOG
user	LOG_USER
uucp	LOG_UUCP
local0	LOG_LOCAL0
local1	LOG_LOCAL1
local2	LOG_LOCAL2
local3	LOG_LOCAL3
local4	LOG_LOCAL4
local5	LOG_LOCAL5
local6	LOG_LOCAL6
local7	LOG_LOCAL7

mapPriority（*levelname*）

将日志记录级别名称映射到系统日志优先级名称。如果您使用的是自定义级别，或者默认算法不适合您的需要，则可能需要覆盖此设置。默认的算法映射DEBUG，INFO，WARNING，ERROR和 CRITICAL等效系统日志的名称，以及所有其他级别名称为“警告”。

16.8.11。NTEventLogHandler

该 模块NTEventLogHandler位于logging.handlers模块中，支持将日志消息发送到本地Windows NT，Windows 2000或Windows XP事件日志。在您使用它之前，您需要安装Mark Hammond的Win32 Python扩展。

*类*logging.handlers.NTEventLogHandler（*appname*，*dllname* = *None*，*logtype* ='*Application*'）

返回类的新实例NTEventLogHandler。该*应用程序的名字*被用作它出现在事件日志中定义的应用程序名称。使用此名称创建适当的注册表项。该*的Dllname*应该给一个.dll的完全合格的路径名或.exe包含消息定义在日志中保持（如果没有指定，'win32service.pyd'则使用-这是安装了Win32扩展，包含了一些基本的占位符消息定义需要注意的是使用这些占位符将使您的事件日志变大，因为整个消息源都保存在日志中。如果您希望更轻松的日志，则必须传入自己的.dll或.exe的名称，其中包含您想要的消息定义在事件日志中使用）。该 *日志类型*是一个'Application'，'System'或者'Security'，并且默认为'Application'。

close（）

此时，您可以从注册表中删除应用程序名称作为事件日志条目的来源。但是，如果您这样做，您将无法按照您在事件日志查看器中预期的方式查看事件 - 它需要能够访问注册表以获取.dll名称。目前的版本不这样做。

emit（*记录*）

确定消息ID，事件类别和事件类型，然后在NT事件日志中记录消息。

getEventCategory（*记录*）

返回记录的事件类别。如果您想指定自己的类别，请覆盖此选项。该版本返回0。

getEventType（*记录*）

返回记录的事件类型。如果你想指定你自己的类型，覆盖这个。该版本能够使用的处理器的类型映射属性，它是建立在一个映射__init__()到包含映射词典DEBUG，INFO，WARNING，ERROR和CRITICAL。如果您使用自己的级别，则需要重写此方法或在处理程序的*typemap*属性中放置合适的字典。

getMessageID（*记录*）

返回记录的消息ID。如果您使用的是自己的消息，则可以通过将*消息*传递给记录器作为ID而不是格式字符串来实现此目的。然后，在这里，您可以使用字典查找来获取消息ID。该版本返回1，这是基础消息ID win32service.pyd。

12年8月16日。SMTPHandler

该模块SMTPHandler位于logging.handlers模块中，支持通过SMTP将日志消息发送到电子邮件地址。

class logging.handlers.SMTPHandler（*mailhost*，*fromaddr*，*toaddrs*，*subject*，*credentials* = *None*，*secure* = *None*，*timeout* = *1.0* ）

返回类的新实例SMTPHandler。该实例使用电子邮件的发件人地址和主题行进行初始化。该*toaddrs*应该是一个字符串列表。要指定非标准SMTP端口，请使用（主机，端口）元组格式作为*mailhost*参数。如果使用字符串，则使用标准SMTP端口。如果您的SMTP服务器需要验证，您可以为*credentials*参数指定一个（用户名，密码）元组。

要指定使用安全协议（TLS），请将元组传递给 *安全*参数。这仅在提供认证凭证时使用。元组应该是一个空元组，或者是一个带有密钥文件名称的单值元组，或者是一个带有密钥文件和证书文件名称的2值元组。（这个元组传递给 `smtplib.SMTP.starttls()`方法。）

可以使用*timeout*参数指定与SMTP服务器通信的 *超时时间*。

*：在3.3版本中的新*的*超时*加入争论。

emit（*记录*）

格式化记录并将其发送到指定的收件人。

getSubject（*记录*）

如果要指定与记录相关的主题行，请覆盖此方法。

13年8月16日。MemoryHandler

该`MemoryHandler`级，位于在`logging.handlers`模块，支持在内存中的日志记录缓冲，定期将其冲洗到 *目标*处理程序。每当缓冲区已满时，或者发现某个严重程度或更高的事件时，都会发生刷新。

`MemoryHandler`是更一般的`BufferingHandler`，这是一个抽象类的一个子 类。这将记录记录缓冲在内存中。无论何时将每条记录添加到缓冲区中，都会通过调用`shouldFlush()`来查看缓冲区是否应该被刷新。如果它应该的话，那么`flush()`预计会冲洗。

class `logging.handlers.` BufferingHandler（*容量*）

使用指定容量的缓冲区初始化处理程序。

emit（*记录*）

将记录追加到缓冲区。如果`shouldFlush()`返回true，则调用`flush()`来处理缓冲区。

flush（）

你可以重写这个来实现自定义刷新行为。该版本只是将缓冲区清空。

shouldFlush（*记录*）

如果缓冲区达到容量，则返回true。可以重写此方法以实现自定义刷新策略。

class `logging.handlers.` MemoryHandler（*capacity*，*flushLevel* = *ERROR*，*target* = *None*，*flushOnClose* = *True* ）

返回类的新实例`MemoryHandler`。该实例使用*容量*的缓冲区大小进行初始化。如果没有指定*flushLevel*， `ERROR`则使用。如果没有*目标*指定，则目标将需要使用设置`setTarget()`此处理程序做任何事情之前有用。如果 *flushOnClose*指定为`False`，则在处理程序关闭时*不会*刷新缓冲区。如果未指定或指定为`True`，则处理程序关闭时，会先发生刷新缓冲区的先前行为。

*改变在3.6版本：*将*flushOnClose*加入参数。

close（）

调用`flush()`，将目标设置为`None`并清除缓冲区。

flush ( )

对于a来说MemoryHandler，冲洗意味着将缓冲的记录发送到目标（如果有的话）。发生这种情况时，缓冲区也会被清除。覆盖，如果你想要不同的行为。

setTarget (*目标*)

设置此处理程序的目标处理程序。

shouldFlush (*记录*)

检查缓冲区已满或*flushLevel*或更高的记录。

14年8月16日。HttpHandler的

该模块HTTPHandler位于logging.handlers模块中，支持使用GET或 POST语义将日志消息发送到Web服务器。

class logging.handlers. HTTPHandler (*host* , *url* , *method ='GET'* , *secure = False* , *credentials = None* , *context = None*)

返回类的新实例HTTPHandler。该*主机*可以是形式的host:port ，如果你需要使用特定的端口号。如果没有指定*方法*，GET则使用。如果*安全*，则将使用HTTPS连接。该*情境*参数可以设置为一个 ssl.SSLContext实例来配置用于HTTPS连接的SSL设置。如果指定了*凭证*，它应该是由用户标识和密码组成的2元组，它将使用基本身份验证放置在HTTP“授权”标头中。如果您指定了凭证，则还应该指定secure = True，以便您的用户标识和密码不会以明文形式通过线路传递。

*在3.5版本中改变：*在*上下文*中添加参数。

mapLogRecord (*记录*)

提供一个字典，基于该字典record进行URL编码并发送到Web服务器。默认实现只是返回 record.__dict__。如果只有一部分LogRecord要发送到Web服务器，或者需要更多特定的发送到服务器的内容，则可以覆盖此方法。

emit (*记录*)

将记录作为URL编码字典发送到Web服务器。该 mapLogRecord()方法用于将记录转换为要发送的字典。

> **注意：** 由于准备将它发送到Web服务器的记录与通用格式化操作不同，因此使用它 setFormatter()来指定 Formatterfor HTTPHandler不起作用。 format()该处理程序不是调用，而是调用 mapLogRecord()并urllib.parse.urlencode()以适合发送到Web服务器的形式对字典进行编码。

15年8月16日。QueueHandler

3.2版本中的新功能

QueueHandler位于logging.handlers模块中的类支持将日志消息发送到队列，例如在queue或multiprocessing模块中实现的那些队列 。

与QueueListener类一起，QueueHandler可用于让处理程序在与日志记录相关的单独线程上执行其工作。这在Web应用程序以及其他服务应用程序中非常重要，在这些应用程序中，服务客户端的线程需要尽快进行响应，而任何可能较慢的操作（例如发送电子邮件 SMTPHandler ）都是在单独的线程上完成的。

*类*logging.handlers.QueueHandler（*队列*）

返回类的新实例QueueHandler。该实例被初始化为用于发送消息的队列。队列可以是任何类似队列的对象; 它是按照原样使用的enqueue()，它需要知道如何向它发送消息。

emit（*记录*）

排队准备LogRecord的结果。

prepare（*记录*）

准备排队记录。该方法返回的对象被排队。

基本实现格式化记录以合并消息和参数，并就地从记录中删除不可打开的项目。

如果要将记录转换为字典或JSON字符串，或者发送记录的修改副本，同时保留原始记录，则可能需要覆盖此方法。

enqueue（*记录*）

使用队列排队队列中的记录put_nowait(); 如果要使用阻止行为或超时或自定义队列实现，则可能需要重写此操作。

16年8月16日。QueueListener

3.2版本中的新功能

QueueListener位于logging.handlers 模块中的类支持从队列中接收日志消息，例如在queue或multiprocessing模块中实现的日志消息。消息从内部线程的队列中接收，并在同一线程上传递给一个或多个处理程序进行处理。尽管 QueueListener它本身并不是一个处理程序，但它在这里被记录是因为它与之协同工作QueueHandler。

与QueueHandler类一起，QueueListener可用于让处理程序在与日志记录相关的单独线程上执行其工作。这在Web应用程序以及其他服务应用程序中非常重要，在这些应用程序中，服务客户端的线程需要尽快进行响应，而任何可能较慢的操作（例如发送电子邮件 SMTPHandler ）都是在单独的线程上完成的。

class logging.handlers.QueueListener（*queue*，** handlers*，*respect_handler_level = False* ）

返回类的新实例QueueListener。该实例被初始化为发送消息的队列以及将处理放置在队列中的条目的处理程序列表。队列可以是任何类似队列的对象; 它按原样传递给dequeue()需要知道如何从中获取消息的方法。如果respect_handler_level是True，则在决定是否将消息传递给该处理程序时，处理程序的级别受到尊重（与消息的级别相比）; 否则，行为与之前的Python版本一样 - 总是将每条消息传递给每个处理程序。

*改变在3.5版本：*将respect_handler_levels加入争论。

dequeue (*块*)

出列记录并将其返回，可选择阻止。

基础实现使用`get()`。如果您想要使用超时或使用自定义队列实现，则可能需要重写此方法。

prepare (*记录*)

准备处理记录。

这个实现只是返回传入的记录。如果需要在将记录传递给处理程序之前执行任何自定义编组或处理操作，则可能需要重写此方法。

handle (*记录*)

处理记录。

这只是循环处理程序提供他们的记录来处理。传递给处理程序的实际对象是从中返回的实际对象`prepare()`。

start ( )

启动监听器。

这将启动一个后台线程来监视LogRecords要处理的队列。

stop ( )

停止收听者。

这要求线程终止，然后等待它这样做。请注意，如果您在应用程序退出前未调用此功能，则可能会有一些记录仍留在队列中，这些记录将不会被处理。

enqueue_sentinel ( )

将标记写入队列以告知侦听器退出。这个实现使用`put_nowait()`。如果您想要使用超时或使用自定义队列实现，则可能需要重写此方法。

3.3版本的新功能

也可以看看:

模 `logging`
记录模块的API参考。

模 `logging.config`
记录模块的配置API。

16.9。 getpass- 便携式密码输入

源代码： Lib / getpass.py

该getpass模块提供两个功能：

getpass. **getpass** (*prompt ='Password ：'* , *stream = None*)

提示用户输入密码而不回显。用户会使用字符串*提示符*（默认为）*提示*。在Unix上，如果需要，使用替换错误处理程序将提示写入类文件对象*流*。 *流*默认为控制终端（ ）或者不可用（在Windows上忽略此参数）。'Password: '/dev/tty sys.stderr

如果无回声输入不可用getpass（ ）返回打印警告消息以进行*流式传输*并读取sys.stdin和发布GetPassWarning。

> **注意：** 如果您从IDLE内部调用getpass，则可以在您启动IDLE的终端中完成输入，而不是在空闲窗口本身。

*异常*getpass. **GetPassWarning**

UserWarning密码输入时发出的子类可能会被回显。

getpass. **getuser** ( )

返回用户的“登录名”。

这个函数检查环境变量 LOGNAME ， USER ， LNAME 和 USERNAME ，并按顺序返回第一个被设置为非空字符串的值。如果没有设置，则会在支持pwd模块的系统上返回密码数据库的登录名，否则会引发异常。

一般来说，这个函数应该优先于os.getlogin()。

16.10。curses- 字符单元显示的终端处理

该curses模块为curses库提供了一个接口，这是用于便携式高级终端处理的事实标准。

尽管curses在Unix环境中被广泛使用，但是Windows，DOS以及其他可能的系统也可以使用这些版本。此扩展模块旨在匹配ncurses的API，Linux上的开源curses库和Unix的BSD变体。

注意： 每当文档提到一个*字符时*，它可以被指定为一个整数，一个字符的Unicode字符串或一个字节的字节字符串。

每当文档提到一个*字符串时*，它可以被指定为一个Unicode字符串或一个字节字符串。

注意： 从版本5.4开始，ncurses库决定如何使用nl_langinfo函数解释非ASCII数据。这意味着您必须调用locale.setlocale()应用程序并使用系统的可用编码之一对Unicode字符串进行编码。本例使用系统的默认编码：

```
import locale
locale.setlocale(locale.LC_ALL, '')
code = locale.getpreferredencoding()
```

然后使用*代码*作为str.encode()调用的编码。

也可以看看:

模 curses.ascii
与ASCII字符一起工作的实用程序，无论您的区域设置如何。

模 curses.panel
面板堆栈扩展，为curses窗口添加深度。

模 curses.textpad
用于支持类**Emacs**绑定的curses的可编辑文本小部件 。

用Python编程Curses
由Andrew Kuchling和Eric Raymond撰写的关于使用curses与Python的教程材料。

Python源代码发行版中的Tools / demo /目录包含一些使用此模块提供的curses绑定的示例程序。

16.10.1。函数

该模块curses定义了以下例外情况：

*异常*curses.**error**
当curses库函数返回错误时引发异常。

注意： 只要函数或方法的*x*或*y*参数是可选的，它们就默认为当前的光标位置。每当*attr*是可选的，它默认为A_NORMAL。

该模块curses定义了以下功能：

curses.**baudrate**( )
以每秒位数的速度返回终端的输出速度。在软件终端仿真器上，它将具有固定的高价值。由于历史原因，在过去，它被用来编写时间延迟的输出回路，偶尔也会根据线路速度改变接口。

curses.beep( )
发出短暂的注意声音。

curses.can_change_color( )
根据程序员是否可以更改终端显示的颜色返回True或返回False。

curses.cbreak( )
进入cbreak模式。在cbreak模式下（有时称为“罕见”模式），关闭正常的tty行缓存，并可逐个读取字符。但是，与原始模式不同，特殊字符（中断，退出，暂停和流量控制）保留其对tty驱动程序和调用程序的影响。首先呼叫raw()然后cbreak()以终止模式离开终端。

curses.color_content(*color_number*)
返回颜色*color_number*中红色，绿色和蓝色（RGB）分量的*强度*，颜色颜色 *编号*必须介于0和之间COLORS。返回一个三元组，其中包含给定颜色的R，G，B值，该值位于 0（无分量）和1000（分量的最大量）之间。

curses.color_pair(*color_number*)
以指定的颜色返回显示文本的属性值。该属性值可以结合A_STANDOUT ， A_REVERSE以及其他A_*属性。pair_number()是这个函数的对应部分。

curses.curs_set(*可见性*)
设置光标状态。 *可见性*可以设置为0 ， ，1或者2不可见，正常或非常明显。如果终端支持请求的可见性，则返回前一个光标状态; 否则引发异常。在许多终端上，“可见”模式是下划线光标，“非常明显”模式是块光标。

curses.def_prog_mode( )
将当前终端模式保存为“程序”模式，即运行程序使用curses时的模式。（它的副本是“shell”模式，当程序不在curses时）。随后的调用reset_prog_mode()将恢复这种模式。

curses.def_shell_mode( )
将当前终端模式保存为“外壳”模式，即运行程序不使用curses时的模式。（当程序使用curses功能时，它的对应是“程序”模式。）随后的调用 reset_shell_mode()将恢复此模式。

curses.delay_output(*ms*)
在输出中插入*ms*毫秒暂停。

curses.doupdate( )
更新物理屏幕。curses库保留两个数据结构，一个表示当前的物理屏幕内容，一个虚拟屏幕表示所需的下一个状态。该doupdate()地更新物理屏幕相匹配的虚拟屏幕。

在noutrefresh()诸如addstr()已经在窗口上执行的写入操作之后，可以通过调用来更新虚拟屏幕。正常的refresh()呼叫只是noutrefresh()紧随其后doupdate()；如果您必须更新多个窗口，则可以通过noutrefresh()在所有窗口上发出呼叫，然后单个呼叫，来加速性能并降低屏幕闪烁doupdate()。

curses.echo( )
进入回声模式。在回显模式下，每输入一个字符，它都会回显到屏幕上。

curses.endwin( )
取消初始化库，并将终端返回到正常状态。

curses.erasechar( )
将用户当前的擦除字符作为一个字节的字节对象返回。在Unix操作系统下，这是curses程序的控制tty的一个属性，并不是由curses库本身设置的。

curses.filter( )

filter()如果使用该例程，则必须在调用之前initscr()调用该例程。其效果是，在这些呼叫期间，LINES设置为1;功能clear，cup，cud，cud1，cuu1，cuu，vpa被禁用;并将该home字符串设置为的值cr。其效果是光标限制在当前行，屏幕更新也是如此。这可以用于在不触摸屏幕的其余部分的情况下启用一次一个字符的行编辑。

curses.flash( )

闪烁屏幕。也就是说，将其改为反向视频，然后在短时间内将其改回。有些人更喜欢诸如“可见的钟声”来发出的可听见的注意力信号beep()。

curses.flushinp( )

刷新所有输入缓冲区。这抛弃了用户键入并且尚未被程序处理的任何类型的头文件。

curses.getmouse( )

在getch()返回KEY_MOUSE信号指示鼠标事件后，应该调用此方法来检索排队的鼠标事件，表示为5元组。*id*是用于区分多个设备的ID值，*x*，*y*，*z*是事件的坐标。（*z*当前未被使用。）*bstate*是一个整数值，其位将被设置为指示事件的类型，并且将是以下一个或多个常量的按位或运算，其中*n*是从1到4的按钮编号：，，，，，，，。(id, x, y, z, bstate)BUTTONn_PRESSEDBUTTONn_RELEASEDBUTTONn_CLICKEDBUTTONn_DOUBLE_CLICKEDBUTTONn_TRIPLE_CLICKEDBUTTON_SHIFTBUTTON_CTRLBUTTON_ALT

curses.getsyx( )

将虚拟屏幕光标的当前坐标作为元组返回。如果是，那么返回。(y, x)leaveokTrue(-1, -1)

curses.getwin(*文件*)

通过先前的putwin()呼叫读取存储在文件中的窗口相关数据。例程然后使用该数据创建并初始化新窗口，返回新的窗口对象。

curses.has_colors( )

True如果终端可以显示颜色，则返回;否则，返回False。

curses.has_ic( )

返回True如果终端有镶片和删除字符的能力。此功能仅出于历史原因，因为所有现代软件终端仿真器都具有此功能。

curses.has_il( )

True如果终端具有插入和删除行功能，或者可以使用滚动区域模拟它们，则返回。此功能仅出于历史原因，因为所有现代软件终端仿真器都具有此功能。

curses.has_key(*ch*)

取一个关键值*ch*，True如果当前终端类型识别出具有该值的关键字，则返回。

curses.halfdelay(*十分之一*)

用于半延时模式，类似于Cbreak模式，因为用户键入的字符可立即用于程序。但是，在阻塞*十分*之几秒后，如果没有输入任何内容，则引发异常。*十分之一*的值必须是介于1和之间的数字255。使用 nocbreak()留下半延迟模式。

curses.init_color(*color_number*, *r*, *g*, *b*)

改变颜色的定义，取出要改变的颜色的数字，后面跟着三个RGB值（红色，绿色和蓝色成分的数量）。*color_number*的值必须介于0和 之间COLORS。*r*，*g*，*b中的*每一个都必须是介于0和 之间的值1000。当init_color()使用时，屏幕上出现的所有颜色立即变为新的定义。此功能在大多数终端上不可用;只有在can_change_color()退货时才有效True。

curses.init_pair(*pair_number*, *fg*, *bg*)

更改颜色对的定义。它有三个参数：要更改的颜色对的编号，前景颜色编号和背景颜色编号。*pair_number*的值必须在1和 之间（颜色对以黑色连线为白色，不能更改）。*fg*和*bg*参数的值必须介于和之间。如果颜色对先前已初始化，则刷新屏幕并将该颜色对的所有出现更改为新定义。COLOR_PAIRS - 100COLORS

curses.**initscr**( )
初始化库。返回代表整个屏幕的窗口对象。

> **注意：** 如果在打开终端时出现错误，则底层curses库可能会导致解释器退出。

curses.**is_term_resized**(*nlines*, *ncols*)
True如果resize_term()将修改窗口结构，False则返回， 否则返回。

curses.**isendwin**( )
返回，True如果endwin()已被调用（即，curses库已被初始化）。

curses.**keyname**(*k*)
将编号为*k*的密钥的名称作为字节对象返回。生成可打印ASCII字符的键的名称是键的字符。控制键组合的名称是由插入符（b'^'）后跟相应的可打印ASCII字符组成的双字节字节对象。alt-key组合（128-255）的名称是一个字节对象，由前缀b'M-'和相应的ASCII字符的名称组成。

curses.**killchar**( )
将用户的当前行kill字符作为一个字节的字节对象返回。在Unix操作系统下，这是curses程序的控制tty的一个属性，并不是由curses库本身设置的。

curses.**longname**( )
返回包含描述当前终端的terminfo长名称字段的字节对象。详细描述的最大长度是128个字符。它只在呼叫之后被定义initscr()。

curses.**meta**(*标志*)
如果*标志*是True，则允许输入8位字符。如果 *标志*是False，只允许7位字符。

curses.**mouseinterval**(*间隔*)
设置按下和释放事件之间可以经过的最大时间（以毫秒为单位），以便将它们识别为点击，并返回上一个间隔值。默认值是200毫秒，或五分之一秒。

curses.**mousemask**(*mousemask*)
设置要报告的鼠标事件，并返回一个元组。 *availmask*指示可以报告哪些指定的鼠标事件; 完全失败后返回。 *oldmask*是给定窗口的鼠标事件掩码的前一个值。如果从未调用此函数，则不会报告任何鼠标事件。(availmask, oldmask) 0

curses.**napms**(*ms*)
睡眠*毫秒*毫秒。

curses.**newpad**(*nlines*, *ncols*)
创建并返回指向具有给定行数和列数的新Pad数据结构的指针。返回一个pad作为窗口对象。

衬垫就像一个窗口，除了它不受屏幕大小的限制，并且不一定与屏幕的特定部分相关联。当需要大窗口时，可以使用焊盘，并且一次只能在屏幕上显示一部分窗口。自动刷新垫（例如从滚动或回显输入）不会发生。的refresh()和noutrefresh() 一个垫的方法要求6个参数指定要显示的垫的一部分，并用于在显示屏幕上的位置。参数是*pminrow*，*pmincol*，*sminrow*，*smincol*，*smaxrow*，*smaxcol*；该*p* 参数指的是要显示的填充区域的左上角，并且*s*参数在屏幕上定义了一个剪贴框，在该屏幕内将显示填充区域。

curses.**newwin**(*nlines*, *ncols*)

curses.newwin (*nlines* , *ncols* , *begin_y* , *begin_x*)
返回一个新的窗口，其左上角在 ，其高度/宽度为 *nlines* / *ncols*。 (begin_y, begin_x)

默认情况下，窗口将从指定位置延伸到屏幕的右下角。

curses.nl ( )
进入换行模式。此模式将返回键转换为输入的换行符，并将换行符转换为输出的回车符和换行符。换行模式最初处于打开状态。

curses.nocbreak ( )
离开cbreak模式。使用线路缓冲恢复到正常的“烹饪”模式。

curses.noecho ( )
留下回声模式。输入字符的回显被关闭。

curses.nonl ( )
离开换行模式。在输入时禁止将换行转换为换行符，并禁止将换行符的低级转换转换为换行符/返回输出（但这不会改变其行为addch('\n')，它总是等同于虚拟屏幕上的返回和换行符）。通过翻译，诅咒有时可以加快垂直运动的速度; 此外，它将能够检测输入上的返回键。

curses.noqiflush ( )
当noqiflush()使用常规，与相关联的输入和输出队列的正常冲洗INTR，QUIT和SUSP字符将不被完成。noqiflush()如果您希望输出在处理程序退出后继续执行，就好像中断没有发生一样，您可能需要调用信号处理程序。

curses.noraw ( )
保持原始模式。使用线路缓冲恢复到正常的“烹饪”模式。

curses.pair_content (*pair_number*)
返回包含请求颜色对的颜色的元组。*pair_number*的值必须介于和之间。(fg, bg)1COLOR_PAIRS - 1

curses.pair_number (*attr*)
返回由属性值*attr*设置的颜色对的编号。 color_pair()是这个函数的对应部分。

curses.putp (*str*)
相当于; 为当前终端发出指定terminfo能力的值。请注意， 总是输出到标准输出。 tputs(str, 1, putchar) putp()

curses.qiflush ([*flag*])
如果*标志*是False，则效果与调用相同noqiflush()。如果 *标志*是True或者没有提供参数，当这些控制字符被读取时，队列将被刷新。

curses.raw ( )
进入原始模式。在原始模式下，中断，退出，暂停和流量控制键的正常线路缓冲和处理被关闭; 字符会逐个呈现给curses输入函数。

curses.reset_prog_mode ( )
如之前保存的那样，将终端恢复到“编程”模式 def_prog_mode()。

curses.reset_shell_mode ( )
如之前保存的，将终端恢复到“外壳”模式 def_shell_mode()。

curses.resetty ( )
将终端模式的状态恢复到上次调用时的状态 savetty()。

curses.resize_term (*nlines* , *ncols*)

使用后端功能resizeterm() ，执行大部分工作; 在调整窗口大小时， resize_term()空白填充扩展的区域。调用应用程序应该用适当的数据填写这些区域。该resize_term()函数尝试调整所有窗口的大小。但是，由于打击垫的调用惯例，不可能在没有与应用程序进行额外交互的情况下调整这些大小。

curses.resizeterm (*nlines* , *ncols*)

将标准窗口和当前窗口调整为指定尺寸，并调整由记录窗口尺寸的curses库（特别是SIGWINCH处理程序）使用的其他簿记数据。

curses.savetty ( )

将终端模式的当前状态保存在缓冲区中，可用于 resetty()。

curses.setsyx (*y* , *x*)

将虚拟屏幕光标设置为*y* ，*x*。如果*y*和*x*都是-1 ，则 leaveok设置True。

curses.setupterm (*term = None* , *fd = -1*)

初始化终端。 *term*是给出终端名称的字符串，或者None; 如果省略或None ，的值 TERM环境变量将被使用。*fd*是任何初始化序列将被发送到的文件描述符; 如果没有提供，或者将使用-1文件描述符for sys.stdout。

curses.start_color ( )

如果程序员想要使用颜色，并且在任何其他颜色操作例程被调用之前必须被调用。之后立即调用这个例程是一个好习惯initscr()。

start_color()初始化模块中的八种基本颜色（黑色，红色，绿色，黄色，蓝色，品红色，青色和白色）以及两个全局变量curses ，COLORS并且COLOR_PAIRS包含终端可以支持的最大颜色和颜色对数。它还将终端上的颜色恢复到刚刚打开终端时的值。

curses.termattrs ( )

返回终端支持的所有视频属性的逻辑OR。当诅咒程序需要完全控制屏幕外观时，此信息非常有用。

curses.termname ( )

返回环境变量的值 TERM，作为一个字节对象，截断为14个字符。

curses.tigetflag (*capname*)

以整数形式返回与terminfo能力名称*capname*对应的布尔能力的值。-1如果*capname*不是布尔功能，或者0终端描述中已取消或不存在，则返回值。

curses.tigetnum (*capname*)

以整数形式返回与terminfo能力名称*capname*对应的数字能力的值。-2如果*capname*不是数字能力，或者-1终端描述中被取消或缺失，则返回值。

curses.tigetstr (*capname*)

以字节对象的形式返回与terminfo能力名称*capname*对应的字符串能力的值。返回None如果*capname* 不是terminfo的“串能力”，或者被取消或者从终端描述不存在。

curses.tparm (*str*[, ...])

使用提供的参数实例化字节对象*str* ，其中*str*应该是从terminfo数据库获取的参数化字符串。例如 可能会导致，确切的结果取决于终端类型。 tparm(tigetstr("cup"), 5, 3)b'\033[6;4H'

curses.typeahead (*fd*)

指定文件描述符*fd*用于键入检查。如果*fd* 是-1 ，则不会进行事先检查。

curses库通过在更新屏幕时定期查找typeahead来进行“line-breakout优化”。如果找到输入，并且它来自tty，则当前更新将被推迟，直到刷新或再次调用doupdate，从而可以更快地响应事先键入的命令。这个函

数允许指定一个不同的文件描述符来进行typehead检查。

curses.**unctrl**(*ch*)
返回字符*ch*的可打印表示的字节对象。控制字符表示为脱字符，后跟字符，例如as b'^C'。打印字符保持原样。

curses.**ungetch**(*ch*)
推*ch*，所以下一个getch()会返回它。

> **注意：** 调用之前只能推送一个*ch*getch()。

curses.**update_lines_cols**( )
更新 LINES 和 COLS。用于检测手动调整屏幕大小。

3.5版本中的新功能。

curses.**unget_wch**(*ch*)
推*ch*，所以下一个get_wch()会返回它。

> **注意：** 调用之前只能推送一个*ch*get_wch()。

3.3版本的新功能

curses.**ungetmouse**(*id*，*x*，*y*，*z*，*bstate*)
将KEY_MOUSE事件推入输入队列，将给定的状态数据与它关联。

curses.**use_env**(*标志*)
如果使用，应该在调用之前调用此函数initscr()或调用newterm。当*flag*是时False，即使环境变量，也会使用terminfo数据库中指定的行和列的值LINES 和 COLUMNS （默认使用），或者如果curses正在窗口中运行（在这种情况下，默认行为是使用窗口大小if LINES 和 COLUMNS 没有设置）。

curses.**use_default_colors**( )
允许在支持此功能的终端上使用颜色的默认值。使用它来支持应用程序的透明度。默认颜色分配给颜色编号-1。调用此函数后，例如将颜色对*x*初始化为默认背景上的红色前景色。init_pair(x, curses.COLOR_RED, -1)

curses.**wrapper**(*func*，...)
初始化诅咒并调用另一个可调用对象*func*，它应该是您的curses使用应用程序的其余部分。如果应用程序引发异常，则此函数将在重新引发异常并生成回溯之前将终端恢复到正常状态。可调用对象*func*然后将主窗口'stdscr'作为其第一个参数，然后传递给其他任何参数wrapper()。在调用*func*之前，wrapper()打开cbreak模式，关闭echo，启用终端键盘，如果终端具有颜色支持，则初始化颜色。在退出时（无论是正常还是例外），它将恢复已烹饪模式，打开回音并禁用终端键盘。

16.10.2。窗口对象

窗口对象，返回的initscr()和newwin()上面，有以下几种方法和属性：

window.**addch**(*ch*[, *attr*])
window.**addch**(*y*, *x*, *ch*[, *attr*])
涂料角色*CH*在与属性*ATTR*，在该位置覆盖任何字符以前的画家。默认情况下，字符位置和属性是窗口对象的当前设置。(y, x)

> **注意：** 在窗口，子窗口或垫子外写字引发了一个问题curses.error。尝试写入窗口右下角，子窗口或打击垫会导致在打印字符后引发异常。

window.addnstr（*str*，*n*[，*attr*]）
window.addnstr（*y*，*x*，*str*，*n*[，*attr*]）
使用属性 *attr*最多绘制字符串*str*中的 *n个*字符，覆盖之前在显示器上的所有内容。（y，x）

window.addstr（*str*[，*attr*]）
window.addstr（*y*，*x*，*str*[，*attr*]）
涂料的字符串*STR*在与属性 *ATTR*，显示先前覆盖任何东西。（y，x）

> **注意：** 在窗口，子窗口或垫子外写字引发curses.error。尝试写入窗口右下角，子窗口或焊盘将导致在打印字符串后引发异常。

window.attroff（*attr*）
从应用于所有写入当前窗口的“背景”集中删除属性*attr*。

window.attron（*attr*）
从应用于所有写入的“背景”集添加属性*attr*到当前窗口。

window.attrset（*attr*）
将“背景”属性设置为*attr*。这组最初 0（没有属性）。

window.bkgd（*ch*[，*attr*]）
将窗口的背景属性设置为字符*ch*，属性为*attr*。然后，该更改将应用于该窗口中的每个字符位置：

- 窗口中每个字符的属性都会更改为新的背景属性。
- 无论何处出现前一个背景字符，都会更改为新的背景字符。

window.bkgdset（*ch*[，*attr*]）
设置窗口的背景。窗口的背景由一个字符和任何属性组合组成。背景的属性部分与写入窗口的所有非空字符组合（OR）。背景的字符和属性部分都与空白字符组合在一起。背景变成字符的属性，并通过任何滚动和插入/删除行/字符操作与字符一起移动。

window.border（[*ls*[，*rs*[，*ts*[，*bs*[，*tl*[，*tr*[，*bl*[，*br*]]]]]]]]）
在窗口的边缘周围绘制边框。每个参数指定用于边框特定部分的字符; 有关更多详细信息，请参阅下表。

> **注意：** 0任何参数的值都会导致该参数使用默认字符。关键字参数*不能*使用。该表中列出了默认值：

参数	描述	默认值
LS	左边	ACS_VLINE
RS	右边	ACS_VLINE
TS	最佳	ACS_HLINE
BS	底部	ACS_HLINE
TL	左上角	ACS_ULCORNER
TR	右上角	ACS_URCORNER
BL	左下角	ACS_LLCORNER
BR	右下角	ACS_LRCORNER

window.box（[*vertch*，*horch*]）

类似于border()，但*ls*和*rs*都是*垂直的*，*ts*和 *bs*都是*霍尔*。此功能始终使用默认的角落角色。

window.chgat(*attr*)
window.chgat(*num*, *attr*)
window.chgat(*y*, *x*, *attr*)
window.chgat(*y*, *x*, *num*, *attr*)
在当前光标位置或位置（如果提供）设置*num*字符的属性。如果没有给出*num*或者是，则将该行的所有字符的属性设置为该行的末尾。如果提供此功能，则将光标移动到位置。使用该方法将更改已更改的行，以便在下一次窗口刷新时重新显示内容。(y, x)-1(y, x)touchline()

window.clear( )
喜欢erase()，但也会导致整个窗口下次打电话时重新粉刷refresh()。

window.clearok(*标志*)
如果*标志*是True，下一个呼叫refresh()将完全清除窗口。

window.clrtobot( )
从光标擦除到窗口的末尾：删除光标下的所有行，然后执行相应的clrtoeol()操作。

window.clrtoeol( )
从光标擦除到行尾。

window.cursyncup( )
更新窗口所有祖先的当前光标位置，以反映窗口的当前光标位置。

window.delch([*y*, *x*])
删除任何字符。(y, x)

window.deleteln( )
删除光标下的行。以下所有行都向上移动一行。

window.derwin(*begin_y*, *begin_x*)
window.derwin(*nlines*, *ncols*, *begin_y*, *begin_x*)
“派生窗口”的缩写derwin()与调用相同 subwin()，只是*begin_y*和*begin_x*与窗口的原点相关，而不是相对于整个屏幕。为派生窗口返回一个窗口对象。

window.echochar(*ch*[, *attr*])
添加属性*attr*的字符*ch*，并立即调用 窗口。 refresh()

window.enclose(*y*, *x*)
测试给定窗口是否包含给定的一对屏幕相对字符单元坐标，返回True或False。确定屏幕窗口的哪个子集包含鼠标事件的位置很有用。

window.encoding
用于编码方法参数的编码（Unicode字符串和字符）。编码属性是在创建子窗口时从父窗口继承的，例如使用window.subwin()。默认情况下，使用区域设置编码（请参阅locale.getpreferredencoding()）。

3.3版本的新功能

window.erase( )
清除窗口。

window.getbegyx( )
返回左上角的坐标元组。(y, x)

window.getbkgd ( )
返回给定窗口的当前背景字符/属性对。

window.getch ([*y*, *x*])
获得一个角色。注意，返回的整数*并没有*一定要在ASCII范围：功能键，键盘按键等被比255更高的数字在无延迟模式为代表，返回-1如果没有输入，否则等待，直到按下一个键。

window.get_wch ([*y*, *x*])
获得广泛的性格。返回大多数键的字符，或功能键，小键盘键和其他特殊键的整数。在无延迟模式下，如果没有输入，则引发异常。

3.3版本的新功能

window.getkey ([*y*, *x*])
获取一个字符，返回一个字符串，而不是一个整数，像getch() 这样。功能键，小键盘键和其他特殊键返回包含键名称的多字节字符串。在无延迟模式下，如果没有输入，则引发异常。

window.getmaxyx ( )
返回窗口高度和宽度的元组。(y, x)

window.getparyx ( )
将此窗口的起始坐标作为元组返回到其父窗口。如果此窗口没有父项，则返回。(y, x) (-1, -1)

window.getstr ( )
window.getstr (*n*)
window.getstr (*y*, *x*)
window.getstr (*y*, *x*, *n*)
从用户读取一个字节对象，具有原始行编辑能力。

window.getyx ( )
返回当前光标位置相对于窗口左上角的元组。(y, x)

window.hline (*ch*, *n*)
window.hline (*y*, *x*, *ch*, *n*)
显示从字符*ch*开始的长度为*n*的水平线。(y, x)

window.idcok (*标志*)
如果*标志*是False，curses不再考虑使用终端的硬件插入/删除字符特征; 如果*标志*是True，则启用使用字符插入和删除。当首次初始化curses时，缺省情况下使用字符插入/删除。

window.idlok (*标志*)
如果是*国旗*True，curses将尝试使用硬件行编辑功能。否则，行插入/删除被禁用。

window.immedok (*标志*)
如果*标志*是True，窗口图像的任何改变自动导致窗口被刷新; 你不必再打电话给refresh()自己。但是，由于重复调用wrefresh，它可能会大大降低性能。该选项默认是禁用的。

window.inch ([*y*, *x*])
返回窗口中给定位置的字符。底部的8位是字符，高位是属性。

window.insch (*ch* [, *attr*])
window.insch (*y*, *x*, *ch* [, *attr*])
油漆字符*CH*在具有属性*ATTR*，从移动位置的线*X*由一个字符权。(y, x)

window.**insdelln**(*nlines*)

将*nlines*行插入当前行上方的指定窗口中。该*nlines*底线都将丢失。对于负*nline*，请删除从光标下方开始的*nlines*行，然后向上移动其余行。底部的*nlines*线被清除。当前光标位置保持不变。

window.**insertln**()

在光标下插入一个空行。以下所有行都向下移动一行。

window.**insnstr**(*str*, *n*[, *attr*])

window.**insnstr**(*y*, *x*, *str*, *n*[, *attr*])

在光标下方的字符前面插入一个字符串（可以放在行上的字符数量），最多可包含*n*个字符。如果*n*为零或负数，则插入整个字符串。光标右边的所有字符都右移，线上最右边的字符丢失。光标位置不变（如果指定，则移至*y*，*x*后）。

window.**insstr**(*str*[, *attr*])

window.**insstr**(*y*, *x*, *str*[, *attr*])

在光标之下的字符之前插入一个字符串（可以放在行上的字符数量）。光标右边的所有字符都右移，线上最右边的字符丢失。光标位置不变（如果指定，则移至*y*，*x*后）。

window.**instr**([*n*])

window.**instr**(*y*, *x*[, *n*])

返回从当前光标位置开始的窗口中提取的字符的字节对象，如果指定，则返回*y*，*x*。属性从角色中剥离。如果指定了*n*，则instr()最多返回*n*个字符的字符串（不包括尾部NUL）。

window.**is_linetouched**(*线*)

返回True如果指定的线路是自最后通话修改 refresh()；否则返回False。 curses.error 如果*行*对给定窗口无效，则引发异常。

window.**is_wintouched**()

返回True如果指定的窗口是因为在最后一次通话修改 refresh()；否则返回False。

window.**keypad**(*标志*)

如果*标志*是True，由某些键（键盘，功能键）生成的转义序列将被解释curses。如果*标志*是False，转义序列将保留在输入流中。

window.**leaveok**(*标志*)

如果*标志*是True，则光标保持在更新位置，而不是位于“光标位置”。这可以尽可能减少光标移动。如果可能的话，光标将变得不可见。

如果*标志*是False，更新后光标将始终处于“光标位置”。

window.**move**(*new_y*, *new_x*)

将光标移到。(new_y, new_x)

window.**mvderwin**(*y*, *x*)

将窗口移动到其父窗口中。窗口的屏幕相关参数不会改变。此例程用于在屏幕上相同的物理位置显示父窗口的不同部分。

window.**mvwin**(*new_y*, *new_x*)

移动窗口使其左上角位于。(new_y, new_x)

window.**nodelay**(*标志*)

如果*标志*是True，getch()将是非阻塞的。

window.**notimeout**(*标志*)

如果*标志*是True，转义序列不会超时。

如果*标志*是False几毫秒后，转义序列将不会被解释，并且将被保留在输入流中。

window.noutrefresh( )
标记刷新，但等待。该函数更新代表窗口所需状态的数据结构，但不强制更新物理屏幕。要做到这一点，请致电 doupdate()。

window.overlay(*destwin* [, *sminrow*, *smincol*, *dminrow*, *dmincol*, *dmaxrow*, *dmaxcol*])
将窗口覆盖在*destwin*的顶部。窗口不必是相同的大小，只有重叠区域被复制。该副本是非破坏性的，这意味着当前背景字符不会覆盖*destwin*的旧内容 。

要对复制区域进行细化控制，overlay()可以使用第二种形式 。*sminrow*和*smincol*是源窗口的左上角坐标，其他变量在目标窗口中标记一个矩形。

window.overwrite(*destwin* [, *sminrow*, *smincol*, *dminrow*, *dmincol*, *dmaxrow*, *dmaxcol*])
覆盖在*destwin*之上的*窗口*。窗口不必是相同的大小，在这种情况下，只有重叠区域被复制。该副本具有破坏性，这意味着当前背景字符将覆盖*destwin*的旧内容 。

要对复制区域进行细化控制，overwrite()可以使用第二种形式 。*sminrow*和*smincol*是源窗口的左上角坐标，其他变量标记目标窗口中的矩形。

window.putwin(*文件*)
将与窗口关联的所有数据写入提供的文件对象。这些信息可以在以后使用该getwin()功能进行检索。

window.redrawln(*beg*, *num*)
表明*NUM*屏线，开始行*乞讨*，被损坏，应在下一次完全重绘refresh()电话。

window.redrawwin( )
触摸整个窗口，使其在下次refresh()调用时完全重新绘制 。

window.refresh([*pminrow*, *pmincol*, *sminrow*, *smincol*, *smaxrow*, *smaxcol*])
立即更新显示（与以前的绘图/删除方法同步实际屏幕）。

这6个可选参数只能在窗口是使用创建的打击垫时指定newpad()。需要额外的参数来指示包含垫和屏幕的部分。*pminrow*和*pmincol*指定矩形的左上角显示在*打击*垫中。 *sminrow*， *smincol*，*smaxrow*和*smaxcol*指定要在屏幕上显示的矩形的边缘。由于矩形必须具有相同的大小，因此要在屏幕中显示的矩形的右下角是从屏幕坐标计算出来的。两个矩形必须完全包含在它们各自的结构中。的负值*pminrow*，*pmincol*，*sminrow*或*smincol*被视为零。

window.resize(*nlines*, *ncols*)
为curses窗口重新分配存储以将其大小调整为指定的值。如果任何一个维度大于当前值，则窗口的数据将填充具有当前背景再现（如设置bkgdset() ）的空白。

window.scroll([*lines = 1*])
通过*线条*向上滚动屏幕或滚动区域。

window.scrollok(*标志*)
控制当窗口的光标离开窗口边界或滚动区域时发生的情况，或者是作为底线上的换行动作的结果，或者是输入最后一行的最后一个字符。如果*标志*是False，光标留在最后一行。如果*标志*是True，窗口向上滚动一行。请注意，为了获得终端的物理滚动效果，还需要调用idlok()。

window.setscrreg(*顶部*, *底部*)
将滚动区域从行*顶部设置*到行*底部*。所有滚动操作都将在该地区进行。

window.standend ( )
关闭突出特性。在某些终端上，这具有关闭所有属性的副作用。

window.standout ( )
打开属性*A_STANDOUT*。

window.subpad (*begin_y* , *begin_x*)
window.subpad (*nlines* , *ncols* , *begin_y* , *begin_x*)
返回一个子窗口，其左上角在，宽度/高度为*ncols* / *nlines*。(begin_y, begin_x)

window.subwin (*begin_y* , *begin_x*)
window.subwin (*nlines* , *ncols* , *begin_y* , *begin_x*)
返回一个子窗口，其左上角在，宽度/高度为*ncols* / *nlines*。(begin_y, begin_x)

默认情况下，子窗口将从指定位置延伸到窗口的右下角。

window.syncdown ( )
触摸在任何祖先窗口中被触摸的窗口中的每个位置。这个例程被调用refresh() ，所以几乎不需要手动调用它。

window.syncok (*标志*)
如果*标志*是True ，则syncup()在窗口发生变化时自动调用*标志*。

window.syncup ( )
触摸窗口中已更改的窗口祖先中的所有位置。

window.timeout (*延迟*)
为窗口设置阻止或非阻止读取行为。如果*延迟*为负，则使用阻塞读取（将无限期等待输入）。如果 *延迟*为零，则使用非阻塞式读取，如果没有输入正在等待， getch()将返回-1。如果*延迟*为正， getch()则将阻塞*延迟*毫秒， -1如果在该时间结束时仍然没有输入，则返回。

window.touchline (*start* , *count* [, *changed*])
假装行*数*已经改变，从行*开始*。如果 提供*更改*，则指定受影响的行是否被标记为已更改（*更改*=True ）或未更改（*更改*=False ）。

window.touchwin ( )
为了绘制优化目的，假设整个窗口已经被更改。

window.untouchwin ( )
自上次调用以来，将窗口中的所有行标记为未更改 refresh()。

window.vline (*ch* , *n*)
window.vline (*y* , *x* , *ch* , *n*)
显示从字符*ch*开始的长度为*n*的垂直线。(y, x)

16.10.3。常量

该curses模块定义了以下数据成员：

curses.ERR
一些诅咒例程返回一个整数，例如 getch() ，ERR失败时返回 。

curses.OK

一些诅咒例程返回一个整数，例如 napms() ，OK成功时返回 。

curses.version

表示模块当前版本的字节对象。也可以作为 __version__。

一些常量可用于指定字符单元属性。可用的确切常数取决于系统。

属性	含义
A_ALTCHARSET	备用字符集模式
A_BLINK	闪烁模式
A_BOLD	粗体模式
A_DIM	暗淡模式
A_INVIS	不可见或空白模式
A_NORMAL	普通属性
A_PROTECT	保护模式
A_REVERSE	反向背景和前景色
A_STANDOUT	突出模式
A_UNDERLINE	下划线模式
A_HORIZONTAL	水平高光
A_LEFT	左突出显示
A_LOW	低亮点
A_RIGHT	右突出显示
A_TOP	最高的亮点
A_VERTICAL	垂直高亮
A_CHARTEXT	位掩码来提取一个字符

有几个常量可用于提取某些方法返回的相应属性。

位掩码	含义
A_ATTRIBUTES	位掩码来提取属性
A_CHARTEXT	位掩码来提取一个字符
A_COLOR	位掩码提取色彩对字段信

键由以名称开头的整数常量引用 KEY_。可用的确切键帽取决于系统。

关键常数	键
KEY_MIN	最小键值
KEY_BREAK	断开键（不可靠）
KEY_DOWN	向下箭头
KEY_UP	向上箭头
KEY_LEFT	左箭头
KEY_RIGHT	右箭头
KEY_HOME	主页键（向上+左箭头）
KEY_BACKSPACE	退格（不可靠）
KEY_F0	功能键。最多支持64个功能键。
KEY_Fn	功能键的值*n*
KEY_DL	删除线
KEY_IL	插入行
KEY_DC	删除字符
KEY_IC	插入字符或进入插入模式
KEY_EIC	退出插入字符模式

关键常数	键
KEY_CLEAR	清除屏幕
KEY_EOS	清除屏幕结束
KEY_EOL	清除到行尾
KEY_SF	向前滚动1行
KEY_SR	向后滚动1行（反向）
KEY_NPAGE	下一页
KEY_PPAGE	上一页
KEY_STAB	设置选项卡
KEY_CTAB	清除选项卡
KEY_CATAB	清除所有标签
KEY_ENTER	输入或发送（不可靠）
KEY_SRESET	软（部分）重置（不可靠）
KEY_RESET	重置或硬重置（不可靠）
KEY_PRINT	打印
KEY_LL	首页向下或向下（左下）
KEY_A1	键盘左上角
KEY_A3	键盘右上方
KEY_B2	键盘中心
KEY_C1	键盘左下方
KEY_C3	键盘右下方
KEY_BTAB	后退标签
KEY_BEG	Beg（开始）
KEY_CANCEL	取消
KEY_CLOSE	关
KEY_COMMAND	命令（命令）
KEY_COPY	复制
KEY_CREATE	创建
KEY_END	结束
KEY_EXIT	出口
KEY_FIND	找
KEY_HELP	帮帮我
KEY_MARK	标记
KEY_MESSAGE	信息
KEY_MOVE	移动
KEY_NEXT	下一个
KEY_OPEN	打开
KEY_OPTIONS	选项
KEY_PREVIOUS	上一个（上一个）
KEY_REDO	重做
KEY_REFERENCE	参考（参考）
KEY_REFRESH	刷新
KEY_REPLACE	更换
KEY_RESTART	重新开始
KEY_RESUME	恢复
KEY_SAVE	保存

关键常数	键
KEY_SBEG	Shifted Beg（开始）
KEY_SCANCEL	移动取消
KEY_SCOMMAND	转移命令
KEY_SCOPY	转移拷贝
KEY_SCREATE	转移创建
KEY_SDC	转移删除字符
KEY_SDL	移动删除线
KEY_SELECT	选择
KEY_SEND	转移结束
KEY_SEOL	转移清除线
KEY_SEXIT	转移出口
KEY_SFIND	转移查找
KEY_SHELP	转移帮助
KEY_SHOME	转移的家
KEY_SIC	转移输入
KEY_SLEFT	移动左箭头
KEY_SMESSAGE	转移消息
KEY_SMOVE	移动移动
KEY_SNEXT	向下移动
KEY_SOPTIONS	转移选项
KEY_SPREVIOUS	上移
KEY_SPRINT	转移打印
KEY_SREDO	转移重做
KEY_SREPLACE	移位替换
KEY_SRIGHT	移动右箭头
KEY_SRSUME	转移的简历
KEY_SSAVE	转移保存
KEY_SSUSPEND	转移暂停
KEY_SUNDO	移动撤消
KEY_SUSPEND	暂停
KEY_UNDO	解开
KEY_MOUSE	鼠标事件已发生
KEY_RESIZE	终端调整大小事件
KEY_MAX	最大键值

上VT100s和他们的软件仿真，如X终端仿真器，通常有至少四个功能键（ KEY_F1 ，KEY_F2 ， KEY_F3 ，KEY_F4 ）可用，并映射到箭头键 KEY_UP ，KEY_DOWN ，KEY_LEFT和KEY_RIGHT在明显的方式。如果您的机器具有PC键盘，则可以安全地使用箭头键和十二个功能键（较旧的PC键盘可能只有十个功能键）；此外，以下键盘映射也是标准配置：

键帽	不变
Insert	KEY_IC
Delete	KEY_DC
Home	KEY_HOME
End	KEY_END
Page Up	KEY_PPAGE
Page Down	KEY_NPAGE

下表列出了备用字符集中的字符。这些是从VT100终端继承的，并且通常可用于诸如X终端的软件仿真。当没有图形可用时，curses会回到粗糙的可打印ASCII近似值。

注意： 这些只有在`initscr()`被调用后才可用。

ACS代码	含义
ACS_BBSS	右上角的替代名称
ACS_BLOCK	实心方块
ACS_BOARD	董事会的正方形
ACS_BSBS	水平线的替代名称
ACS_BSSB	左上角的替代名称
ACS_BSSS	顶部T恤的替代名称
ACS_BTEE	底部三通
ACS_BULLET	子弹
ACS_CKBOARD	棋盘（点画）
ACS_DARROW	指向下的箭头
ACS_DEGREE	学位标志
ACS_DIAMOND	钻石
ACS_GEQUAL	大于或 - 等于对
ACS_HLINE	水平线
ACS_LANTERN	灯笼符号
ACS_LARROW	左箭头
ACS_LEQUAL	低于或相等到
ACS_LLCORNER	左下角
ACS_LRCORNER	右下角
ACS_LTEE	左发球
ACS_NEQUAL	不等号
ACS_PI	字母pi
ACS_PLMINUS	加号或减号
ACS_PLUS	大加号
ACS_RARROW	右箭头
ACS_RTEE	正确的发球台
ACS_S1	扫描线1
ACS_S3	扫描线3
ACS_S7	扫描线7
ACS_S9	扫描线9
ACS_SBBS	右下角的替代名称
ACS_SBSB	垂直线的替代名称
ACS_SBSS	右T恤的替代名称
ACS_SSBB	左下角的替代名称
ACS_SSBS	底部三通的替代名称
ACS_SSSB	左T恤的替代名称
ACS_SSSS	交叉名称或大加号
ACS_STERLING	英镑
ACS_TTEE	顶级球座
ACS_UARROW	向上箭头
ACS_ULCORNER	左上角

ACS_URCORNER	右上角
ACS_VLINE	垂线

下表列出了预定义的颜色：

不变	颜色
COLOR_BLACK	黑色
COLOR_BLUE	蓝色
COLOR_CYAN	青色（浅绿蓝色）
COLOR_GREEN	绿色
COLOR_MAGENTA	洋红色（紫红色）
COLOR_RED	红
COLOR_WHITE	白色
COLOR_YELLOW	黄色

16.11。curses.textpad-文本输入小部件的诅咒程序

该curses.textpad模块提供了一个Textbox处理curses窗口中的基本文本编辑的类，支持类似于Emacs（也是Netscape Navigator，BBedit 6.x，FrameMaker和许多其他程序）的键集。该模块还提供了一个矩形绘图功能，可用于框定文本框或用于其他目的。

该模块curses.textpad定义了以下功能：

curses.textpad.**rectangle**（*赢*，*ULY*，*ULX*，*LRY*，*LRX*）

绘制一个矩形。第一个参数必须是一个窗口对象; 其余的参数是相对于该窗口的坐标。第二个和第三个参数是要绘制的矩形左上角的y和x坐标; 第四个和第五个参数是右下角的y和x坐标。矩形将在终端上使用VT100 / IBM PC形式字符进行绘制（包括xterm和大多数其他软件终端仿真器）。否则，它将用ASCII破折号，垂直条和加号绘制。

16.11.1。文本框对象

你可以Textbox如下实例化一个对象：

class curses.textpad.**Textbox**（*win*）

返回一个文本框小部件对象。该*赢*的说法应该是一个诅咒 窗口对象，其中的文本框将被遏制。文本框的编辑光标最初位于包含窗口的左上角，并带有坐标。实例的标志最初处于打开状态。（0，0）stripspaces

Textbox 对象有以下方法：

edit（[*验证器*]）

这是您通常使用的入口点。它接受编辑键击，直到输入一个终止键击。如果 提供*验证*程序，它必须是一个函数。将按键输入的每个击键作为参数调用; 命令调度是在结果上完成的。此方法以字符串形式返回窗口内容; stripspaces属性是否影响窗口中的空白 。

do_command（*ch*）

处理单个命令按键。以下是支持的特殊按键：

按键	行动
Control-A	去窗口的左边缘。
Control-B	光标离开，如果合适的话，包装到上一行。
Control-D	删除光标下的字符。

按键	行动
Control-E	转到右边缘（strippaces off）或行尾（strippaces on）。
Control-F	正确的光标，适当时包装到下一行。
Control-G	终止，返回窗口内容。
Control-H	向后删除字符。
Control-J	终止，如果窗口是1行，否则插入换行符。
Control-K	如果行是空白的，删除它，否则清除行结束。
Control-L	刷新屏幕。
Control-N	光标下来; 向下移动一行。
Control-O	在光标位置插入空行。
Control-P	光标向上; 向上移动一行。

如果光标位于无法移动的边缘，则移动操作不会执行任何操作。在可能的情况下支持以下同义词：

不变	按键
KEY_LEFT	Control-B
KEY_RIGHT	Control-F
KEY_UP	Control-P
KEY_DOWN	Control-N
KEY_BACKSPACE	Control-h

所有其他击键都被视为一个命令来插入给定的字符并向右移动（使用换行符）。

gather ( )

以字符串形式返回窗口内容; 会员是否会影响窗口中的空白stripspaces。

stripspaces

该属性是一个控制窗口中空白解释的标志。打开时，每行上的尾随空白将被忽略; 任何将光标放在尾部空白上的光标移动都会转到该行的末尾，而在收集窗口内容时尾随空白会被剥离。

16.12。curses.ascii- 用于ASCII字符的实用程序

该curses.ascii模块为ASCII字符提供名称常量，并为各种ASCII字符类中的成员资格测试函数。提供的常量是控制字符的名称，如下所示：

名称	含义
NUL	
SOH	标题开始，控制台中断
STX	文字开始
ETX	文本结尾
EOT	传输结束
ENQ	询问，跟ACK流量控制
ACK	承认
BEL	钟
BS	退格
TAB	标签
HT	别名为TAB：“水平选项卡”
LF	换行
NL	别名为LF：“新行”
VT	垂直选项卡
FF	换页
CR	回车
SO	移出，开始替代字符集
SI	转入，恢复默认字符集
DLE	数据链接转义
DC1	XON，用于流量控制
DC2	设备控制2，块模式流量控
DC3	XOFF，用于流量控制
DC4	设备控制4
NAK	否定确认
SYN	同步空闲
ETB	结束传输块
CAN	取消
EM	媒体结束
SUB	替代
ESC	逃逸
FS	文件分隔符
GS	组分隔符

名称	含义
RS	记录分隔符，块模式终止符
US	单元分隔符
SP	空间
DEL	删除

请注意，其中许多这些在现代使用中没有什么实际意义。助记符来源于数字计算机之前的电传打字机惯例。

该模块提供以下功能，在标准C库中的功能上进行了图案化：

curses.ascii.**isalnum**(*c*)
检查ASCII字母数字字符; 它相当于。`isalpha(c) or isdigit(c)`

curses.ascii.**isalpha**(*c*)
检查一个ASCII字母字符; 它相当于。`isupper(c) or islower(c)`

curses.ascii.**isascii**(*c*)
检查符合7位ASCII集的字符值。

curses.ascii.**isblank**(*c*)
检查一个ASCII空白字符; 空间或水平标签。

curses.ascii.**iscntrl**(*c*)
检查ASCII控制字符（范围在0x00到0x1f或0x7f）。

curses.ascii.**isdigit**(*c*)
检查一个ASCII十进制数字，'0'通过'9'。这相当于。`c in string.digits`

curses.ascii.**isgraph**(*c*)
检查ASCII空间以外的任何可打印字符。

curses.ascii.**islower**(*c*)
检查ASCII小写字符。

curses.ascii.**isprint**(*c*)
检查包括空格在内的任何ASCII可打印字符。

curses.ascii.**ispunct**(*c*)
检查任何不是空格或字母数字字符的可打印ASCII字符。

curses.ascii.**isspace**(*c*)
检查ASCII空格字符; 空格，换行，回车，换页，水平标签，垂直标签。

curses.ascii.**isupper**(*c*)
检查一个ASCII大写字母。

curses.ascii.**isxdigit**(*c*)
检查ASCII十六进制数字。这相当于。`c in string.hexdigits`

curses.ascii.**isctrl**(*c*)
检查ASCII控制字符（序号值0到31）。

curses.ascii.**ismeta**(*c*)
检查一个非ASCII字符（序数值为0x80或更高）。

这些函数接受整数或单字符字符串; 当参数是一个字符串时，它首先使用内置函数进行转换`ord()`。

请注意，所有这些函数都会检查从传入的字符串的字符派生的有序位值; 他们实际上并不知道主机的字符编码。

以下两个函数采用单字符字符串或整数字节值; 他们返回相同类型的值。

curses.ascii.**ascii**(*c*)
返回对应于*c*的低7位的ASCII值。

curses.ascii.**ctrl**(*c*)
返回与给定字符相对应的控制字符（字符位值与0x1f按位结尾）。

curses.ascii.**alt**(*c*)
返回与给定的ASCII字符相对应的8位字符（字符位值与0x80位对齐）。

以下函数采用单字符字符串或整数值; 它返回一个字符串。

curses.ascii.**unctrl**(*c*)
返回ASCII字符的字符串表示形式*c*。如果*c*是可打印的，则该字符串是字符本身。如果字符是一个控制字符（0x00-0x1f），则该字符串包含一个插入符（'^'），后跟相应的大写字母。如果字符是ASCII删除（0x7f），则字符串是'^?'。如果字符设置了元位（0x80），则元位将被剥离，应用前面的规则并将其'!'前置到结果中。

curses.ascii.**controlnames**
一个33个元素的字符串数组，包含从0（NUL）到0x1f（US）的32个ASCII控制字符的ASCII助记符，以及SP空格字符的助记符。

16.13。curses.panel-一个面板堆栈扩展为诅咒

面板是具有深度附加功能的窗口，因此它们可以堆叠在一起，并且只显示每个窗口的可见部分。面板可以被添加，在堆栈中向上或向下移动，并被移除。

16.13.1。函数

该模块curses.panel定义了以下功能：

curses.panel.bottom_panel ( )
返回面板堆栈中的底部面板。

curses.panel.new_panel (*赢*)
返回一个面板对象，将其与给定的窗口*win*相关联。请注意，您需要保留显式引用的返回面板对象。如果您不这样做，面板对象将被垃圾收集并从面板堆栈中移除。

curses.panel.top_panel ( )
返回面板堆栈中的顶部面板。

curses.panel.update_panels ( )
在面板堆栈发生更改后更新虚拟屏幕。这不叫 curses.doupdate() ，所以你必须自己做。

16.13.2。面板对象

new_panel()上面返回的面板对象是具有堆叠顺序的窗口。总是有一个窗口与确定内容的面板相关联，而面板方法负责面板堆栈中窗口的深度。

面板对象有以下方法：

Panel.above ( )
返回当前面板上方的面板。

Panel.below ( )
返回当前面板下方的面板。

Panel.bottom ( )
将面板推到堆栈的底部。

Panel.hidden ( )
True如果面板隐藏（不可见）则返回，False否则返回。

Panel.hide ( )

隐藏面板。这不会删除对象，它只是使屏幕上的窗口不可见。

Panel.**move**(*y*, *x*)
将面板移动到屏幕坐标。(y, x)

Panel.**replace**(*赢*)
将与面板相关联的窗口更改为窗口*赢*。

Panel.**set_userptr**(*obj*)
将面板的用户指针设置为*obj*。这用于将任意数据与面板相关联，并且可以是任何Python对象。

Panel.**show**( )
显示面板（可能已被隐藏）。

Panel.**top**( )
将面板推到堆栈的顶部。

Panel.**userptr**( )
返回面板的用户指针。这可能是任何Python对象。

Panel.**window**( )
返回与面板关联的窗口对象。

16.14。platform- 访问底层平台的识别数据

源代码： Lib / platform.py

注意： 特定的平台按字母顺序列出，其中Linux包含在Unix部分。

16.14.1。跨平台

platform.**architecture**(*可执行文件= sys.executable* , *bits =''* , *linkage =''*)

查询给定的可执行文件（默认为Python解释器二进制文件）以获取各种体系结构信息。

返回包含有关位体系结构和用于可执行文件的链接格式信息的元组。这两个值都以字符串形式返回。(bits, linkage)

无法确定的值将返回给定参数预设。如果给出位''，则sizeof(pointer)（或者sizeof(long)在Python版本<1.5.2）被用作指示器以支持指针大小。

该功能依靠系统的file命令来完成实际工作。如果不是所有的Unix平台和一些非Unix平台都可以使用它，那么只有当可执行文件指向Python解释器时才可用。当不满足上述需求时使用合理的默认值。

注意： 在Mac OS X（也可能是其他平台）上，可执行文件可能是包含多种体系结构的通用文件。
为了获得当前解释器的“64位”，查询sys.maxsize属性更加可靠：

```
is_64bits = sys.maxsize > 2**32
```

platform.**machine**( )

返回机器类型，例如'i386'。如果无法确定值，则返回空字符串。

platform.**node**( )

返回计算机的网络名称（可能不完全合格！）。如果无法确定值，则返回空字符串。

platform.**platform**(*aliased = 0* , *terse = 0*)

使用尽可能多的有用信息返回标识底层平台的单个字符串。

输出的目的是为了*人类可读*，而不是机器可解析的。它可能在不同的平台上看起来不同，这是有意的。

如果*aliased*为true，则该函数将为各种平台使用别名，这些平台报告的系统名称与通用名称不同，例如，SunOS将报告为Solaris。该system_alias()功能用于实现这一点。

将*terse*设置为true会导致该函数仅返回识别平台所需的绝对最小信息。

platform.**processor**( )

返回（实际）处理器名称，例如'amdk6'。

如果无法确定值，则返回空字符串。请注意，许多平台不提供这些信息，或只是返回相同的值 machine()。NetBSD这样做。

platform.**python_build**( )

返回一个元组，将Python构建号和日期声明为字符串。(buildno, builddate)

platform.**python_compiler**( )

返回标识用于编译Python的编译器的字符串。

platform.**python_branch**( )

返回一个标识Python实现SCM分支的字符串。

platform.**python_implementation**( )

返回一个标识Python实现的字符串。可能的返回值是：'CPython'，'IronPython'，'Jython'，'PyPy'。

platform.**python_revision**( )

返回一个标识Python实现SCM修订版的字符串。

platform.**python_version**( )

以字符串形式返回Python版本'major.minor.patchlevel'。

请注意，与Python不同sys.version，返回的值将始终包含patchlevel（它默认为0）。

platform.**python_version_tuple**( )

将Python版本作为字符串的元组返回。(major, minor, patchlevel)

请注意，与Python不同sys.version，返回值将始终包含patchlevel（它默认为'0'）。

platform.**release**( )

返回系统的释放，例如'2.2.0'或者'NT'如果不能确定的值返回一个空字符串。

platform.**system**( )

返回系统/ OS的名称，例如'Linux'，'Windows'或'Java'。如果无法确定值，则返回空字符串。

platform.**system_alias**(*系统*, *版本*, *版本*)

返回别名到用于某些系统的常用市场名称。它还会在某些情况下对信息进行一些重新排序，否则会导致混淆。(system, release, version)

platform.**version**( )

返回系统的发行版本，例如。如果无法确定值，则返回空字符串。'#3 on degas'

platform.**uname**( )

相当便携的uname界面。返回一个namedtuple() 包含六个属性：system，node，release，version，machine，和processor。

请注意，这会processor在os.uname()结果中添加第六个属性（ ）。而且，属性名称对于前两个属性是不同的; os.uname()命名他们 sysname和nodename。

无法确定的条目设置为''。

*版本3.3中更改：*结果从元组更改为namedtuple。

16.14.2。Java平台

platform.java_ver（*release =''*，*vendor =''*，*vminfo =（''，''，''）*，*osinfo =（''，''，''）*）

Jython的版本界面。

返回一个元组，其中*vminfo*是一个元组，*osinfo*是一个元组 。无法确定的值被设置为作为参数给出的默认值（全部默认为）。(release, vendor, vminfo, osinfo)(vm_name, vm_release, vm_vendor)(os_name, os_version, os_arch)''

16.14.3。Windows平台

platform.win32_ver（*release =''*，*version =''*，*csd =''*，*ptype =''* ）

从Windows注册表获取更多版本信息，并返回一个 涉及操作系统版本，版本号，CSD级别（Service Pack）和操作系统类型（多/单处理器）的元组。(release, version, csd, ptype)

作为一个提示：*p-型*是在单处理器NT机器和多处理器机器。在*“免费”*指的是操作系统版本是免费的调试代码。它也可以声明*'检查'*，这意味着操作系统版本使用调试代码，即检查参数，范围等的代码。'Uniprocessor Free''Multiprocessor Free'

> **注意：** 这个函数最适合安装Mark Hammond的 win32all软件包，但也适用于Python 2.3及更高版本（在Python 2.6中添加了对此的支持）。它显然只能在Win32兼容平台上运行。

16.14.3.1。Win95 / 98特定

platform.popen（*cmd*，*mode ='r'*，*bufsize = -1* ）

便携式popen()界面。找到一个喜欢的工作popen实现win32pipe.popen()。在Windows NT上，win32pipe.popen() 应该工作; 在Windows 9x上，它由于MS C库中的错误而挂起。

*自3.3版弃用：*此功能已过时。使用该subprocess模块。特别检查用子流程模块部分替换旧功能。

16.14.4。Mac OS平台

platform.mac_ver（*release =''*，*versioninfo =（''，''，''）*，*machine =''* ）

获取Mac OS版本信息并将其作为元组返回，其中*versioninfo*是元组。 (release, versioninfo, machine) (version, dev_stage, non_release_version)

无法确定的条目设置为''。所有的元组条目都是字符串。

16.14.5。Unix平台

platform.dist (*distname =''* , *version =''* , *id =''* , *supported_dists =（'SuSE'，'debian'，'redhat'，'mandrake'，...）*)

这是另一个名字linux_distribution()。

*从版本3.5开始弃用，将在版本3.8中删除：*请参阅distro包之类的替代方法。

platform.linux_distribution (*distname =''* , *version =''* , *id =''* , *supported_dists =（'SuSE'，'debian'，'redhat'，'mandrake'，...）* , *full_distribution_name = 1*)

尝试确定Linux OS分发名称的名称。

supported_dists可能会给出定义要查找的Linux发行版的集合。它默认为由其发行文件名称标识的当前支持的Linux发行版的列表。

如果full_distribution_name为真（默认），则返回从操作系统读取的完整分发。否则，supported_dists使用从中取出的短名称。

返回(distname,version,id)默认为给定参数的参数的元组。 id是版本号后的括号中的项目。它通常是版本代号。

*从版本3.5开始弃用，将在版本3.8中删除：*请参阅distro包之类的替代方法。

platform.libc_ver (*可执行文件= sys.executable* , *lib =''* , *version =''* , *chunksize = 2048*)

尝试确定文件可执行文件（默认为Python解释器）所链接的libc版本。如果查找失败，则返回默认给定参数的字符串元组。 (lib, version)

请注意，此函数深入了解不同libc版本为可执行文件添加符号的方式，可能仅适用于使用**gcc**编译的可执行文件。

该文件被读出并在块扫描*CHUNKSIZE*字节。

16.15。 errno- 标准的errno系统符号

该模块提供标准errno系统符号。每个符号的值是相应的整数值。名字和描述都是借来的linux/include/errno.h，应该是非常全面的。

errno.errorcode
提供从errno值到底层系统中字符串名称映射的字典。例如，errno.errorcode[errno.EPERM]映射到'EPERM'。

要将数字错误代码转换为错误消息，请使用os.strerror()。

在以下列表中，当前平台上未使用的符号未由模块定义。定义的符号的具体列表可用errno.errorcode.keys()。可用的符号可以包括：

errno.EPERM
不允许操作

errno.ENOENT
无此文件或目录

errno.ESRCH
没有这样的过程

errno.EINTR
系统调用中断。

> **也可以看看:** 这个错误被映射到异常InterruptedError。

errno.EIO
I / O错误

errno.ENXIO
没有这样的设备或地址

errno.E2BIG
Arg列表太长

errno.ENOEXEC
执行格式错误

errno.EBADF
错误的文件号码

errno.ECHILD
没有儿童进程

errno.EAGAIN
再试一次

errno.ENOMEM
内存不足

errno.EACCES
没有权限

errno.EFAULT
地址不正确

errno.ENOTBLK
需要阻止设备

errno.EBUSY
设备或资源忙碌

errno.EEXIST
文件已存在

errno.EXDEV
跨设备链接

errno.ENODEV
无此设备

errno.ENOTDIR
不是目录

errno.EISDIR
是一个目录

errno.EINVAL
无效的论点

errno.ENFILE
文件表溢出

errno.EMFILE
打开的文件过多

errno.ENOTTY
不是打字机

errno.ETXTBSY
文本文件忙

errno.EFBIG
    文件过大

errno.ENOSPC
    设备上没有剩余空间

errno.ESPIPE
    非法寻求

errno.EROFS
    只读文件系统

errno.EMLINK
    链接太多

errno.EPIPE
    管道破损

errno.EDOM
    数学论证超出了func的领域

errno.ERANGE
    数学结果无法表示

errno.EDEADLK
    会发生资源死锁

errno.ENAMETOOLONG
    文件名太长

errno.ENOLCK
    没有可用的记录锁

errno.ENOSYS
    功能未实现

errno.ENOTEMPTY
    目录不为空

errno.ELOOP
    遇到太多符号链接

errno.EWOULDBLOCK
    操作会阻止

errno.ENOMSG
    没有期望类型的消息

errno.EIDRM
标识符已删除

errno.ECHRNG
频道数超出范围

errno.EL2NSYNC
2级不同步

errno.EL3HLT
3级停止

errno.EL3RST
3级重置

errno.ELNRNG
链接数量超出范围

errno.EUNATCH
协议驱动程序未附加

errno.ENOCSI
没有可用的CSI结构

errno.EL2HLT
2级停止

errno.EBADE
交换无效

errno.EBADR
无效的请求描述符

errno.EXFULL
交换完整

errno.ENOANO
没有阳极

errno.EBADRQC
无效的请求代码

errno.EBADSLT
插槽无效

errno.EDEADLOCK
文件锁定死锁错误

errno.EBFONT
错误的字体文件格式

errno.ENOSTR
设备不是流

errno.ENODATA
无可用数据

errno.ETIME
计时器已过期

errno.ENOSR
流出资源

errno.ENONET
机器不在网络上

errno.ENOPKG
软件包未安装

errno.EREMOTE
对象是远程的

errno.ENOLINK
链接已被切断

errno.EADV
广告错误

errno.ESRMNT
Srmount错误

errno.ECOMM
通讯发送错误

errno.EPROTO
协议错误

errno.EMULTIHOP
多跳试图

errno.EDOTDOT
RFS特定错误

errno.EBADMSG
不是数据信息

errno.EOVERFLOW
对于定义的数据类型，值太大

errno.ENOTUNIQ
名称在网络上不唯一

errno.EBADFD
文件描述符处于不良状态

errno.EREMCHG
远程地址已更改

errno.ELIBACC
无法访问所需的共享库

errno.ELIBBAD
访问损坏的共享库

errno.ELIBSCN
a.out中的.lib节已损坏

errno.ELIBMAX
试图链接太多共享库

errno.ELIBEXEC
无法直接执行共享库

errno.EILSEQ
非法字节序列

errno.ERESTART
应该重新启动中断的系统调用

errno.ESTRPIPE
流管道错误

errno.EUSERS
用户太多

errno.ENOTSOCK
套接字在非套接字上运行

errno.EDESTADDRREQ
目的地址需要

errno.EMSGSIZE
信息太长

errno.EPROTOTYPE
协议套接字错误类型

errno.ENOPROTOOPT
协议不可用

errno.EPROTONOSUPPORT
协议不支持

errno.ESOCKTNOSUPPORT
套接字类型不受支持

errno.EOPNOTSUPP
传输端点不支持操作

errno.EPFNOSUPPORT
协议族不受支持

errno.EAFNOSUPPORT
地址族不受协议支持

errno.EADDRINUSE
地址已在使用中

errno.EADDRNOTAVAIL
无法分配请求的地址

errno.ENETDOWN
网络已关闭

errno.ENETUNREACH
网络不可达

errno.ENETRESET
网络由于重置而断开连接

errno.ECONNABORTED
软件导致连接中止

errno.ECONNRESET
连接重置由对等

errno.ENOBUFS
没有可用的缓冲空间

errno.EISCONN
传输端点已连接

errno.ENOTCONN
传输端点未连接

errno.ESHUTDOWN
传输端点关闭后无法发送

errno.ETOOMANYREFS
引用太多：无法拼接

errno.ETIMEDOUT
连接超时

errno.ECONNREFUSED
拒绝连接

errno.EHOSTDOWN
主机关闭

errno.EHOSTUNREACH
没有到主机的路由

errno.EALREADY
操作已在进行中

errno.EINPROGRESS
正在进行中的操作

errno.ESTALE
陈旧的NFS文件句柄

errno.EUCLEAN
结构需要清洁

errno.ENOTNAM
不是XENIX命名的类型文件

errno.ENAVAIL
没有可用的XENIX信号量

errno.EISNAM
是一个命名的类型文件

errno.EREMOTEIO
远程I/O错误

errno.EDQUOT
超出配额

16.16。ctypes- 一个用于Python的外部函数库

ctypes是一个用于Python的外部函数库。它提供C兼容的数据类型，并允许在DLL或共享库中调用函数。它可以用来将这些库封装在纯Python中。

16.16.1。ctypes教程

注意：本教程中的代码示例doctest用于确保它们实际工作。由于一些代码示例在Linux，Windows或Mac OS X下表现不同，因此它们在注释中包含doctest指令。

注意：一些代码示例引用了ctypes c_int类型。在它是别名的平台上。所以，如果您希望印刷的话，您不应该感到困惑- 它们实际上是同一种类型。 sizeof(long) == sizeof(int)c_longc_longc_int

16.16.1.1。加载动态链接库

ctypes导出*cdll*以及Windows *windll*和*oledll* 对象，以加载动态链接库。

通过将它们作为这些对象的属性进行访问来加载库。*cdll* 加载使用标准cdecl调用约定导出函数的库，而*windll*库使用stdcall 调用约定调用函数。*oledll*也使用stdcall调用约定，并假定函数返回一个Windows HRESULT错误代码。错误代码用于OSError在函数调用失败时自动引发异常。

*在版本3.3中更改：*用于引发的Windows错误WindowsError，现在是其别名OSError。

以下是Windows的一些示例。请注意，msvcrtMS标准C库包含大多数标准C函数，并使用cdecl调用约定：

```
>>> from ctypes import *
>>> print(windll.kernel32)
<WinDLL 'kernel32', handle ... at ...>
>>> print(cdll.msvcrt)
<CDLL 'msvcrt', handle ... at ...>
>>> libc = cdll.msvcrt
>>>
```

Windows会.dll自动附加通常的文件后缀。

注意： 通过访问标准C库cdll.msvcrt将使用库的过时版本，该版本可能与Python正在使用的版本不兼容。在可能的情况下，使用本机Python功能，否则导入并使用该msvcrt模块。

在Linux上，需要指定*包含*加载库的扩展名的文件名，因此不能使用属性访问来加载库。要么使用LoadLibrary()dll加载器的方法，要么 通过调用构造函数创建CDLL实例来加载库：

```
>>> cdll.LoadLibrary("libc.so.6")
<CDLL 'libc.so.6', handle ... at ...>
>>> libc = CDLL("libc.so.6")
>>> libc
```

```
<CDLL 'libc.so.6', handle ... at ...>
>>>
```

16.16.1.2。从加载的dll访问函数

函数作为dll对象的属性被访问：

```
>>> from ctypes import *
>>> libc.printf
<_FuncPtr object at 0x...>
>>> print(windll.kernel32.GetModuleHandleA)
<_FuncPtr object at 0x...>
>>> print(windll.kernel32.MyOwnFunction)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "ctypes.py", line 239, in __getattr__
    func = _StdcallFuncPtr(name, self)
AttributeError: function 'MyOwnFunction' not found
>>>
```

请注意，win32系统DLL喜欢kernel32并user32经常导出ANSI以及UNICODE版本的函数。UNICODE版本的导出是W附加到名称上的，而ANSI版本导出时A 附加了名称。 win32 GetModuleHandle函数返回给定模块名称的 *模块句柄*，它具有以下C原型，并且使用一个宏来公开其中的一个，GetModuleHandle取决于是否定义了UNICODE：

```
/* ANSI version */
HMODULE GetModuleHandleA(LPCSTR lpModuleName);
/* UNICODE version */
HMODULE GetModuleHandleW(LPCWSTR lpModuleName);
```

*windll*不会尝试通过魔法选择其中之一，您必须通过指定GetModuleHandleA或GetModuleHandleW显式地访问您需要的版本，然后分别使用字节或字符串对象调用它。

有时，dll导出的函数名称不是有效的Python标识符，例如"??2@YAPAXI@Z"。在这种情况下，您必须使用 getattr() 来检索函数：

```
>>> getattr(cdll.msvcrt, "??2@YAPAXI@Z")
<_FuncPtr object at 0x...>
>>>
```

在Windows上，一些dll导出函数不是按名称，而是按序号。这些函数可以通过使用序号索引dll对象来访问：

```
>>> cdll.kernel32[1]
<_FuncPtr object at 0x...>
>>> cdll.kernel32[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "ctypes.py", line 310, in __getitem__
    func = _StdcallFuncPtr(name, self)
AttributeError: function ordinal 0 not found
>>>
```

16.16.1.3。调用函数

您可以像调用任何其他Python一样调用这些函数。这个例子使用这个time()函数，该函数返回自Unix纪元以来的系统时间（以秒为单位）以及GetModuleHandleA()函数，该函数返回一个win32模块句柄。

本示例使用NULL指针调用两个函数（None应该用作NULL指针）：

```
>>> print(libc.time(None))
1150640792
>>> print(hex(windll.kernel32.GetModuleHandleA(None)))
0x1d000000
>>>
```

注意： ctypes ValueError如果它检测到传递了无效的参数数目，则可能会在调用该函数后引发一次。这种行为不应该依赖。它在3.6.2中被弃用，并且将在3.7中被删除。

ValueError在调用stdcall具有cdecl调用约定的函数 时引发，反之亦然：

```
>>> cdll.kernel32.GetModuleHandleA(None)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Procedure probably called with not enough arguments (4 bytes missing)
>>>

>>> windll.msvcrt.printf(b"spam")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Procedure probably called with too many arguments (4 bytes in excess)
>>>
```

要找出正确的调用约定，必须查看C头文件或要调用的函数的文档。

在Windows上，ctypes当使用无效参数值调用函数时，使用win32结构化异常处理来防止一般保护错误导致崩溃：

```
>>> windll.kernel32.GetModuleHandleA(32)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OSError: exception: access violation reading 0x00000020
>>>
```

然而，有足够的方法来使Python崩溃ctypes，所以你应该小心。该faulthandler模块可以帮助调试崩溃（例如，由错误的C库调用产生的分段错误）。

None，整数，字节对象和（unicode）字符串是唯一可以在这些函数调用中直接用作参数的本机Python对象。 None作为C NULL指针传递，字节对象和字符串作为指针传递给包含其数据（或）的内存块。Python整数作为平台的默认C 类型传递 ，它们的值被屏蔽以适合C类型。 char *wchar_t *int

在继续使用其他参数类型调用函数之前，我们必须了解更多关于ctypes数据类型的信息。

16.16.1.4。基本数据类型

ctypes 定义了许多基本的C兼容数据类型：

ctypes类型	C型	Python类型
c_bool	_Bool	布尔（1）
c_char	char	1个字符的字节对象
c_wchar	wchar_t	1个字符的字符串
c_byte	char	INT
c_ubyte	unsigned char	INT
c_short	short	INT
c_ushort	unsigned short	INT
c_int	int	INT
c_uint	unsigned int	INT
c_long	long	INT
c_ulong	unsigned long	INT
c_longlong	__int64 要么 long long	INT
c_ulonglong	unsigned __int64 要么 unsigned long long	INT
c_size_t	size_t	INT
c_ssize_t	ssize_t 要么 Py_ssize_t	INT
c_float	float	浮动
c_double	double	浮动
c_longdouble	long double	浮动
c_char_p	char * （NUL终止）	字节对象或 None
c_wchar_p	wchar_t * （NUL终止）	字符串或 None
c_void_p	void *	int或 None

1. 构造函数接受任何具有真值的对象。

所有这些类型都可以通过使用正确类型和值的可选初始值设定项来调用它们来创建：

```
>>> c_int()
c_long(0)
>>> c_wchar_p("Hello, World")
c_wchar_p(140018365411392)
>>> c_ushort(-3)
c_ushort(65533)
>>>
```

由于这些类型是可变的，它们的值也可以在之后改变：

```
>>> i = c_int(42)
>>> print(i)
c_long(42)
>>> print(i.value)
```

```
42
>>> i.value = -99
>>> print(i.value)
-99
>>>
```

分配一个新的值，将指针类型的实例c_char_p， c_wchar_p以及c_void_p改变所述*存储器位置*它们指向，*而不是内容*的内存块（当然不是，因为Python字节对象是不可改变的）：

```
>>> s = "Hello, World"
>>> c_s = c_wchar_p(s)
>>> print(c_s)
c_wchar_p(139966785747344)
>>> print(c_s.value)
Hello World
>>> c_s.value = "Hi, there"
>>> print(c_s)              # the memory location has changed
c_wchar_p(139966783348904)
>>> print(c_s.value)
Hi, there
>>> print(s)                # first object is unchanged
Hello, World
>>>
```

但是，您应该小心，不要将它们传递给期望指向可变内存的函数。如果你需要可变内存块，ctypes有一个create_string_buffer()以各种方式创建它们的 函数。当前的内存块内容可以通过raw 属性访问（或更改）；如果您想以NUL结尾的字符串的形式访问它，请使用以下value 属性：

```
>>> from ctypes import *
>>> p = create_string_buffer(3)            # create a 3 byte buffer, initialized to N
>>> print(sizeof(p), repr(p.raw))
3 b'\x00\x00\x00'
>>> p = create_string_buffer(b"Hello")     # create a buffer containing a NUL termina
>>> print(sizeof(p), repr(p.raw))
6 b'Hello\x00'
>>> print(repr(p.value))
b'Hello'
>>> p = create_string_buffer(b"Hello", 10) # create a 10 byte buffer
>>> print(sizeof(p), repr(p.raw))
10 b'Hello\x00\x00\x00\x00\x00'
>>> p.value = b"Hi"
>>> print(sizeof(p), repr(p.raw))
10 b'Hi\x00lo\x00\x00\x00\x00\x00'
>>>
```

该create_string_buffer() 函数将取代该c_buffer() 函数（仍可作为别名使用），以及c_string() 早期版本的ctypes函数。要创建包含C类型的Unicode字符的可变内存块，请wchar_t使用该 create_unicode_buffer() 函数。

16.16.1.5。调用函数，继续

需要注意的是的printf打印到实际的标准输出通道，*不*给 sys.stdout ，所以这些例子只是在控制台提示符下运行，而不是从内*IDLE*或*PythonWin*的：

```
>>> printf = libc.printf
>>> printf(b"Hello, %s\n", b"World!")
Hello, World!
14
>>> printf(b"Hello, %S\n", "World!")
Hello, World!
14
>>> printf(b"%d bottles of beer\n", 42)
42 bottles of beer
19
>>> printf(b"%f bottles of beer\n", 42.5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ArgumentError: argument 2: exceptions.TypeError: Don't know how to convert parameter 2
>>>
```

如前所述，除整数，字符串和字节对象之外的所有Python类型都必须用相应的ctypes类型包装，以便它们可以转换为所需的C数据类型：

```
>>> printf(b"An int %d, a double %f\n", 1234, c_double(3.14))
An int 1234, a double 3.140000
31
>>>
```

16.16.1.6。用您自己的自定义数据类型调用函数

您还可以自定义ctypes参数转换以允许您自己的类的实例用作函数参数。 ctypes查找一个_as_parameter_属性并将其用作函数参数。当然，它必须是整数，字符串或字节之一：

```
>>> class Bottles:
...     def __init__(self, number):
...         self._as_parameter_ = number
...
>>> bottles = Bottles(42)
>>> printf(b"%d bottles of beer\n", bottles)
42 bottles of beer
19
>>>
```

如果您不想将实例的数据存储在_as_parameter_ 实例变量中，则可以定义一个property可以根据请求提供该属性的函数。

16.16.1.7。指定所需的参数类型（函数原型）

通过设置argtypes属性，可以指定从DLL导出的函数所需的参数类型。

argtypes必须是C数据类型的序列（该printf函数在这里可能不是一个好例子，因为它取决于格式字符串需要可变数量和不同类型的参数，另一方面，实验此功能非常方便）：

```
>>> printf.argtypes = [c_char_p, c_char_p, c_int, c_double]
>>> printf(b"String '%s', Int %d, Double %f\n", b"Hi", 10, 2.2)
String 'Hi', Int 10, Double 2.200000
37
>>>
```

指定格式可以防止不兼容的参数类型（就像C函数的原型一样），并尝试将参数转换为有效的类型：

```
>>> printf(b"%d %d %d", 1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ArgumentError: argument 2: exceptions.TypeError: wrong type
>>> printf(b"%s %d %f\n", b"X", 2, 3)
X 2 3.000000
13
>>>
```

如果你已经定义了你自己传递给函数调用的from_param()类，你必须实现一个类方法，以便它们能够按argtypes顺序使用它们。本from_param()类方法接收传给函数的Python对象，它做一个类型检测，或者是需要确保这个对象是可接受的，然后返回对象本身，它_as_parameter_不管你想传递的C函数属性，或在这种情况下的论点。同样，结果应该是整数，字符串，字节，ctypes实例或具有_as_parameter_属性的对象 。

16.16.1.8。返回类型

默认情况下，函数被假定为返回C int类型。其他返回类型可以通过设置restype函数对象的属性来指定。

这是一个更高级的例子，它使用了strchr函数，它需要一个字符串指针和一个字符，并返回一个指向字符串的指针：

```
>>> strchr = libc.strchr
>>> strchr(b"abcdef", ord("d"))
8059983
>>> strchr.restype = c_char_p    # c_char_p is a pointer to a string
>>> strchr(b"abcdef", ord("d"))
b'def'
>>> print(strchr(b"abcdef", ord("x")))
None
>>>
```

如果你想避免ord("x")上面的调用，你可以设置 argtypes属性，第二个参数将从单个字符的Python字节对象转换为C字符：

```
>>> strchr.restype = c_char_p
>>> strchr.argtypes = [c_char_p, c_char]
>>> strchr(b"abcdef", b"d")
'def'
>>> strchr(b"abcdef", b"def")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
ArgumentError: argument 2: exceptions.TypeError: one character string expected
>>> print(strchr(b"abcdef", b"x"))
None
>>> strchr(b"abcdef", b"d")
'def'
>>>
```

restype如果外部函数返回一个整数，您也可以使用可调用的Python对象（例如函数或类）作为属性。可调用函数将用C函数返回的*整数*调用，并且此调用的结果将用作函数调用的结果。这对检查错误返回值并自动引发异常很有用：

```
>>> GetModuleHandle = windll.kernel32.GetModuleHandleA
>>> def ValidHandle(value):
...     if value == 0:
...         raise WinError()
...     return value
...
>>>
>>> GetModuleHandle.restype = ValidHandle
>>> GetModuleHandle(None)
486539264
>>> GetModuleHandle("something silly")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in ValidHandle
OSError: [Errno 126] The specified module could not be found.
>>>
```

WinError是一个函数，它将调用Windows FormatMessage()API获取错误代码的字符串表示形式，并*返回*一个异常。 WinError需要一个可选的错误代码参数，如果没有人使用，它会调用GetLastError()来检索它。

请注意，通过errcheck属性可以使用更强大的错误检查机制; 详情请参阅参考手册。

16.16.1.9。传递指针（或者：通过引用传递参数）

有时C API函数需要一个*指向*数据类型的*指针*作为参数，可能写入相应的位置，或者数据太大而无法通过值传递。这也被称为*通过引用传递参数*。

ctypesbyref()通过引用导出用于传递参数的函数。这个pointer()函数可以达到同样的效果，但是pointer()由于它构造了一个真正的指针对象，因此做了很多工作，所以byref()如果你不需要Python本身的指针对象，使用它会更快。

```
>>> i = c_int()
>>> f = c_float()
>>> s = create_string_buffer(b'\000' * 32)
>>> print(i.value, f.value, repr(s.value))
0 0.0 b''
>>> libc.sscanf(b"1 3.14 Hello", b"%d %f %s",
...             byref(i), byref(f), s)
3
>>> print(i.value, f.value, repr(s.value))
```

```
1 3.1400001049 b'Hello'
>>>
```

16.16.1.10。结构和联合

结构和联合必须从导出Structure和Union 其中所定义的基类ctypes模块。每个子类都必须定义一个_fields_属性。 _fields_必须是包含*字段名称*和*字段类型*的*2元组*列表 。

字段类型必须是ctypes类型c_int或任何其他派生ctypes类型：结构，联合，数组，指针。

下面是一个POINT结构的简单示例，它包含两个名为*x*和*y*的整数 ，并且还显示了如何在构造函数中初始化结构：

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = [("x", c_int),
...                 ("y", c_int)]
...
>>> point = POINT(10, 20)
>>> print(point.x, point.y)
10 20
>>> point = POINT(y=5)
>>> print(point.x, point.y)
0 5
>>> POINT(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many initializers
>>>
```

但是，您可以构建更复杂的结构。通过使用结构作为字段类型，结构本身可以包含其他结构。

这是一个RECT结构，其中包含两个名为*upperleft*和 *lowerright*的*POINT*：

```
>>> class RECT(Structure):
...     _fields_ = [("upperleft", POINT),
...                 ("lowerright", POINT)]
...
>>> rc = RECT(point)
>>> print(rc.upperleft.x, rc.upperleft.y)
0 5
>>> print(rc.lowerright.x, rc.lowerright.y)
0 0
>>>
```

嵌套结构也可以通过几种方式在构造函数中初始化：

```
>>> r = RECT(POINT(1, 2), POINT(3, 4))
>>> r = RECT((1, 2), (3, 4))
```

字段描述符可以从*类中*检索，它们对于调试很有用，因为它们可以提供有用的信息：

```
>>> print(POINT.x)
<Field type=c_long, ofs=0, size=4>
>>> print(POINT.y)
<Field type=c_long, ofs=4, size=4>
>>>
```

警告: ctypes不支持通过值来传递具有位域的联合或结构。虽然这可能适用于32位x86，但图书馆无法保证在一般情况下工作。具有位域的联合和结构应始终通过指针传递给函数。

16.16.1.11。结构/联合对齐和字节顺序

默认情况下，Structure和Union字段的对齐方式与C编译器所做的相同。可以_pack_在子类定义中指定一个类属性来覆盖这种行为 。这必须设置为正整数并指定字段的最大对齐方式。这也是在MSVC中所做的。#pragma pack(n)

ctypes使用结构和联合的本地字节顺序。要建立与非本地字节顺序结构，你可以使用一个BigEndianStructure，LittleEndianStructure， BigEndianUnion，和LittleEndianUnion基类。这些类不能包含指针字段。

16.16.1.12。结构和联合中的位域

可以创建包含位域的结构和联合。位字段只能用于整数字段，位宽指定为_fields_元组中的第三项：

```
>>> class Int(Structure):
...     _fields_ = [("first_16", c_int, 16),
...                 ("second_16", c_int, 16)]
...
>>> print(Int.first_16)
<Field type=c_long, ofs=0:0, bits=16>
>>> print(Int.second_16)
<Field type=c_long, ofs=0:16, bits=16>
>>>
```

16.16.1.13。数组

数组是序列，包含固定数量的相同类型的实例。

推荐的创建数组类型的方法是将数据类型乘以正整数：

```
TenPointsArrayType = POINT * 10
```

下面是一个有点人为的数据类型的例子，它是一个包含4个POINT和其他内容的结构：

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = ("x", c_int), ("y", c_int)
...
>>> class MyStruct(Structure):
```

```
...     _fields_ = [("a", c_int),
...                 ("b", c_float),
...                 ("point_array", POINT * 4)]
>>>
>>> print(len(MyStruct().point_array))
4
>>>
```

实例以通常的方式创建，通过调用该类：

```
arr = TenPointsArrayType()
for pt in arr:
    print(pt.x, pt.y)
```

上面的代码打印了一系列的行，因为数组内容被初始化为零。0 0

也可以指定正确类型的初始化程序：

```
>>> from ctypes import *
>>> TenIntegers = c_int * 10
>>> ii = TenIntegers(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
>>> print(ii)
<c_long_Array_10 object at 0x...>
>>> for i in ii: print(i, end=" ")
...
1 2 3 4 5 6 7 8 9 10
>>>
```

16.16.1.14。指针

通过调用pointer()一个ctypes类型的函数 来创建指针实例：

```
>>> from ctypes import *
>>> i = c_int(42)
>>> pi = pointer(i)
>>>
```

指针实例有一个contents属性，它返回指针指向的i对象，即上面的对象：

```
>>> pi.contents
c_long(42)
>>>
```

请注意，ctypes没有OOR（原始对象返回），它会在您每次检索属性时构造一个新的等效对象：

```
>>> pi.contents is i
False
>>> pi.contents is pi.contents
False
>>>
```

c_int为指针的内容属性分配另一个实例会导致指针指向存储它的内存位置：

```
>>> i = c_int(99)
>>> pi.contents = i
>>> pi.contents
c_long(99)
>>>
```

指针实例也可以用整数索引：

```
>>> pi[0]
99
>>>
```

分配给整数索引会改变指向的值：

```
>>> print(i)
c_long(99)
>>> pi[0] = 22
>>> print(i)
c_long(22)
>>>
```

也可以使用不同于0的索引，但是您必须知道自己在做什么，就像在C中一样：您可以访问或更改任意内存位置。通常，如果您从C函数接收到指针，则只能使用此功能，并且您*知道*指针实际上指向的是数组而不是单个项目。

在幕后，pointer()函数不仅仅是创建指针实例，它必须首先创建指针*类型*。这是通过POINTER()接受任何ctypes类型的函数完成的，并返回一个新的类型：

```
>>> PI = POINTER(c_int)
>>> PI
<class 'ctypes.LP_c_long'>
>>> PI(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: expected c_long instead of int
>>> PI(c_int(42))
<ctypes.LP_c_long object at 0x...>
>>>
```

调用没有参数的指针类型将创建一个NULL指针。NULL指针有一个False布尔值：

```
>>> null_ptr = POINTER(c_int)()
>>> print(bool(null_ptr))
False
>>>
```

ctypes检查NULL何时取消引用指针（但取消引用无效的非NULL指针会导致Python崩溃）：

```
>>> null_ptr[0]
Traceback (most recent call last):
    ....
```

```
ValueError: NULL pointer access
>>>

>>> null_ptr[0] = 1234
Traceback (most recent call last):
    ....
ValueError: NULL pointer access
>>>
```

16.16.1.15。类型转换

通常，ctypes会进行严格的类型检查。这意味着，如果您 POINTER(c_int)在argtypes函数列表中或者在结构定义中作为成员字段的类型，则只接受完全相同类型的实例。这个规则有一些例外，其中ctypes接受其他对象。例如，您可以传递兼容的数组实例而不是指针类型。因此，POINTER(c_int)ctypes接受一个c_int数组：

```
>>> class Bar(Structure):
...     _fields_ = [("count", c_int), ("values", POINTER(c_int))]
...
>>> bar = Bar()
>>> bar.values = (c_int * 3)(1, 2, 3)
>>> bar.count = 3
>>> for i in range(bar.count):
...     print(bar.values[i])
...
1
2
3
>>>
```

另外，如果一个函数参数被显式声明为一个指针类型（如POINTER(c_int) ）in argtypes ，则可以将该指针类型的对象（ c_int在本例中）传递给该函数。 byref()在这种情况下，ctypes会自动应用所需的转换。

要将POINTER类型字段设置为NULL ，您可以分配None ：

```
>>> bar.values = None
>>>
```

有时你有不兼容类型的实例。在C中，您可以将一种类型转换为另一种类型。 ctypes提供cast()可以以相同方式使用的功能。Bar上面定义的结构接受 其字段的POINTER(c_int)指针或c_int数组values ，但不包含其他类型的实例：

```
>>> bar.values = (c_byte * 4)()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: incompatible types, c_byte_Array_4 instance instead of LP_c_long instance
>>>
```

对于这些情况，该cast()功能非常方便。

该cast()函数可用于将ctypes实例转换为指向不同ctypes数据类型的指针。 cast()需要两个参数，一个是或可以转换为某种类型的指针的ctypes对象，以及一个ctypes指针类型。它返回第二个参数的一个实例，它引用与第一个参数相同的内存块：

```
>>> a = (c_byte * 4)()
>>> cast(a, POINTER(c_int))
<ctypes.LP_c_long object at ...>
>>>
```

所以，cast()可以用来分配结构的values字段Bar：

```
>>> bar = Bar()
>>> bar.values = cast((c_byte * 4)(), POINTER(c_int))
>>> print(bar.values[0])
0
>>>
```

16.16.1.16。不完整类型

*不完全类型*是结构，联合或者其成员尚未指定的数组。在C中，它们由后面定义的前向声明指定：

```
struct cell; /* forward declaration */

struct cell {
    char *name;
    struct cell *next;
};
```

直接翻译成ctypes代码就是这样，但它不起作用：

```
>>> class cell(Structure):
...     _fields_ = [("name", c_char_p),
...                 ("next", POINTER(cell))]
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in cell
NameError: name 'cell' is not defined
>>>
```

因为新类在类声明本身中不可用。在，我们可以定义类并 在类声明之后的后面设置属性：
class cellctypescell_fields_

```
>>> from ctypes import *
>>> class cell(Structure):
...     pass
...
>>> cell._fields_ = [("name", c_char_p),
...                  ("next", POINTER(cell))]
>>>
```

让我们尝试一下。我们创建两个实例cell，并让它们指向彼此，最后遵循指针链几次：

```
>>> c1 = cell()
>>> c1.name = "foo"
>>> c2 = cell()
>>> c2.name = "bar"
>>> c1.next = pointer(c2)
>>> c2.next = pointer(c1)
>>> p = c1
>>> for i in range(8):
...     print(p.name, end=" ")
...     p = p.next[0]
...
foo bar foo bar foo bar foo bar
>>>
```

16.16.1.17。回调函数

ctypes允许从Python可调参数中创建C可调用函数指针。这些有时称为*回调函数*。

首先，您必须为回调函数创建一个类。该类知道调用约定，返回类型以及此函数将接收的参数的数量和类型。

该CFUNCTYPE()工厂函数使用回调函数创建类型cdecl调用约定。在Windows上，WINFUNCTYPE()工厂函数使用stdcall 调用约定为回调函数创建类型。

使用结果类型作为第一个参数调用这两个工厂函数，并将回调函数期望的参数类型作为剩余参数。

我将在这里展示一个使用标准C库qsort()函数的示例，该 函数用于借助回调函数对项目进行排序。 qsort()将用于对整数数组进行排序：

```
>>> IntArray5 = c_int * 5
>>> ia = IntArray5(5, 1, 7, 33, 99)
>>> qsort = libc.qsort
>>> qsort.restype = None
>>>
```

qsort()必须使用指向要排序的数据的指针，数据数组中的项目数量，一个项目的大小以及指向比较函数的指针调用回调。这个回调函数将被两个指向项目的指针调用，如果第一个项目小于第二个项目，它必须返回一个负整数，如果它们相等则返回一个零，否则返回一个正整数。

所以我们的回调函数接收指向整数的指针，并且必须返回一个整数。首先我们创建type回调函数：

```
>>> CMPFUNC = CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
>>>
```

要开始，下面是一个简单的回调，它显示了它传递的值：

```
>>> def py_cmp_func(a, b):
...     print("py_cmp_func", a[0], b[0])
```

```
...     return 0
...
>>> cmp_func = CMPFUNC(py_cmp_func)
>>>
```

结果：

```
>>> qsort(ia, len(ia), sizeof(c_int), cmp_func)
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 5 7
py_cmp_func 1 7
>>>
```

现在我们可以比较这两个项目并返回一个有用的结果：

```
>>> def py_cmp_func(a, b):
...     print("py_cmp_func", a[0], b[0])
...     return a[0] - b[0]
...
>>>
>>> qsort(ia, len(ia), sizeof(c_int), CMPFUNC(py_cmp_func))
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 1 7
py_cmp_func 5 7
>>>
```

正如我们可以轻松检查，我们的数组现在排序：

```
>>> for i in ia: print(i, end=" ")
...
1 5 7 33 99
>>>
```

注意： CFUNCTYPE()只要从C代码中使用对象，请确保保持对象的引用。ctypes没有，如果你不这样做，他们可能被垃圾收集，在回调时崩溃你的程序。

另外，请注意，如果在Python控制之外创建的线程（例如通过调用回调的外部代码）中调用回调函数，则ctypes会在每次调用时创建一个新的虚拟Python线程。这种行为在大多数情况下都是正确的，但这意味着即使这些调用是由同一个C线程创建的，存储在其中的值threading.local也*不会*在不同的回调中生存。

16.16.1.18。访问从dll导出的值

一些共享库不仅可以导出函数，还可以导出变量。Python库本身的一个例子是Py_OptimizeFlag，整数设置为0,1或2，具体取决于启动时给出的-O或-OO标志。

ctypes可以使用in_dll()类型的类方法访问像这样的值。 *pythonapi*是一个预定义的符号，可以访问Python C api：

```
>>> opt_flag = c_int.in_dll(pythonapi, "Py_OptimizeFlag")
>>> print(opt_flag)
c_long(0)
>>>
```

如果口译员已经开始使用-O，则样本将打印c_long(1)，或者c_long(2)如果-OO已经指定。

一个扩展的例子也演示了如何使用指针来访问PyImport_FrozenModules由Python导出的 指针。

引用该价值的文档：

> 该指针被初始化为指向一组 记录，由其成员全部为*NULL*或零的终止。导入冻结模块时，会在此表中进行搜索。第三方代码可以使用这个技巧来提供一个动态创建的冻结模块集合。 struct _frozen

所以操纵这个指针甚至可以证明是有用的。要限制示例大小，我们只显示如何读取此表格ctypes：

```
>>> from ctypes import *
>>>
>>> class struct_frozen(Structure):
...     _fields_ = [("name", c_char_p),
...                 ("code", POINTER(c_ubyte)),
...                 ("size", c_int)]
...
>>>
```

我们已经定义了数据类型，所以我们可以得到指向表的指针：struct _frozen

```
>>> FrozenTable = POINTER(struct_frozen)
>>> table = FrozenTable.in_dll(pythonapi, "PyImport_FrozenModules")
>>>
```

既然table是记录pointer数组struct_frozen，我们可以遍历它，但是我们必须确保我们的循环终止，因为指针没有大小。迟早它可能会因访问冲突或其他原因而崩溃，所以最好在我们敲入NULL条目时跳出循环：

```
>>> for item in table:
...     if item.name is None:
...         break
...     print(item.name.decode("ascii"), item.size)
...
_frozen_importlib 31764
_frozen_importlib_external 41499
__hello__ 161
__phello__ -161
__phello__.spam 161
>>>
```

标准Python有一个冻结模块和一个冻结包（由负大小成员表示）的事实并不为人所熟知，它仅用于测试。例如试用它。`import __hello__`

16.16.1.19。惊喜

在`ctypes`实际发生的情况之外，您可能会预料到某些方面存在一些优势。

考虑下面的例子：

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = ("x", c_int), ("y", c_int)
...
>>> class RECT(Structure):
...     _fields_ = ("a", POINT), ("b", POINT)
...
>>> p1 = POINT(1, 2)
>>> p2 = POINT(3, 4)
>>> rc = RECT(p1, p2)
>>> print(rc.a.x, rc.a.y, rc.b.x, rc.b.y)
1 2 3 4
>>> # now swap the two points
>>> rc.a, rc.b = rc.b, rc.a
>>> print(rc.a.x, rc.a.y, rc.b.x, rc.b.y)
3 4 3 4
>>>
```

嗯。我们当然期待印刷最后的声明。发生了什么？以下是上述行的步骤：`3 4 1 2rc.a, rc.b = rc.b, rc.a`

```
>>> temp0, temp1 = rc.b, rc.a
>>> rc.a = temp0
>>> rc.b = temp1
>>>
```

请注意，`temp0`并且`temp1`还在使用的内部缓冲器的对象`rc`上述对象。因此执行将缓冲区内容复制到缓冲区中。这又反过来改变了内容。所以，最后一项任务并没有预期的效果。`rc.a = temp0 temp0 rc temp1 rc.b = temp1`

请记住，从Structure，Unions和Arrays中检索子对象不会*复制*子对象，而是检索访问根对象的基础缓冲区的包装对象。

另一个可能与预期不同的例子是：

```
>>> s = c_char_p()
>>> s.value = "abc def ghi"
>>> s.value
'abc def ghi'
>>> s.value is s.value
False
>>>
```

为什么要打印False？ctypes实例是包含一个内存块和一些访问内存内容的描述符的对象。在内存块中存储Python对象不会存储对象本身，而是存储该对象contents。每次访问内容都会构造一个新的Python对象！

16.16.1.20。可变大小的数据类型

ctypes 为可变大小的数组和结构提供了一些支持。

该resize()函数可用于调整现有ctypes对象的内存缓冲区大小。该函数将对象作为第一个参数，并将请求的大小以字节为第二个参数。内存块不能小于由对象类型指定的自然内存块，ValueError如果尝试这样，则会引发a ：

```
>>> short_array = (c_short * 4)()
>>> print(sizeof(short_array))
8
>>> resize(short_array, 4)
Traceback (most recent call last):
    ...
ValueError: minimum size is 8
>>> resize(short_array, 32)
>>> sizeof(short_array)
32
>>> sizeof(type(short_array))
8
>>>
```

这很好，但是如何访问此数组中包含的其他元素？由于该类型仍然只知道大约4个元素，所以我们在访问其他元素时遇到错误：

```
>>> short_array[:]
[0, 0, 0, 0]
>>> short_array[7]
Traceback (most recent call last):
    ...
IndexError: invalid index
>>>
```

另一种使用可变大小数据类型的方法ctypes是使用Python的动态特性，并且在所需大小已知的情况下（根据具体情况）重新定义数据类型。

16.16.2。ctypes参考

16.16.2.1。查找共享库

以编译语言编程时，在编译/链接程序和程序运行时访问共享库。

该find_library()函数的目的是以类似于编译器或运行时加载程序的方式来定位库（在具有最近应加载的多个版本的共享库的平台上），而ctypes库加载程序的作用类似于程序运行，并直接调用运行时加载器。

该ctypes.util模块提供了一个功能，可以帮助确定要加载的库。

ctypes.util.find_library(*名字*)

尝试找到一个库并返回一个路径名。 *名字*是不一样的任何前缀库名*的lib*，标的相同.so，.dylib或版本号（这是用于POSIX链接器选项的形式-l）。如果没有找到库，则返回None。

确切的功能依赖于系统。

在Linux上，find_library()尝试运行外部程序（/sbin/ldconfig，gcc，objdump和ld）找到库文件。它返回库文件的文件名。

*在版本3.6中进行了更改：*在Linux上，LD_LIBRARY_PATH搜索库时使用环境变量的值，如果无法通过任何其他方式找到库。

这里有些例子：

```
>>> from ctypes.util import find_library
>>> find_library("m")
'libm.so.6'
>>> find_library("c")
'libc.so.6'
>>> find_library("bz2")
'libbz2.so.1.0'
>>>
```

在OS X上，find_library()尝试使用几个预定义的命名方案和路径来定位库，并在成功时返回完整的路径名：

```
>>> from ctypes.util import find_library
>>> find_library("c")
'/usr/lib/libc.dylib'
>>> find_library("m")
'/usr/lib/libm.dylib'
>>> find_library("bz2")
'/usr/lib/libbz2.dylib'
>>> find_library("AGL")
'/System/Library/Frameworks/AGL.framework/AGL'
>>>
```

在Windows上，find_library()搜索系统搜索路径，并返回完整的路径名，但由于没有预定义的命名方案，调用find_library("c")将失败并返回None。

如果包装共享库ctypes，它*可以*更好地确定在开发时共享库的名字，并硬编码到封装模块，而不是使用find_library()定位在运行时库。

16.16.2.2。加载共享库

有几种方法可以将共享库加载到Python进程中。一种方法是实例化以下类之一：

class ctypes.CDLL（*name*，*mode* = *DEFAULT_MODE*，*handle* = *None*，*use_errno* = *False*，*use_last_error* = *False* ）

这个类的实例代表加载的共享库。这些库中的函数使用标准的C调用约定，并假定返回int。

class ctypes.OleDLL（*name*，*mode* = *DEFAULT_MODE*，*handle* = *None*，*use_errno* = *False*，*use_last_error* = *False* ）

仅限Windows：此类的实例表示加载的共享库，这些库中的函数使用stdcall调用约定，并假定返回特定于Windows的HRESULT代码。 HRESULT 值包含指定函数调用失败还是成功的信息，以及其他错误代码。如果返回值表示失败，OSError则自动提升。

*在版本3.3中改变：*WindowsError曾经被提出。

class ctypes.WinDLL（*name*，*mode* = *DEFAULT_MODE*，*handle* = *None*，*use_errno* = *False*，*use_last_error* = *False* ）

仅Windows：此类的实例表示已加载的共享库，这些库中的函数使用stdcall调用约定，并假定int默认返回。

在Windows CE只标准调用约定时，为了方便， WinDLL并OleDLL使用标准调用约定在这个平台上。

在调用由这些库导出的任何函数之前释放Python 全局解释器锁，然后重新获取它。

class ctypes.PyDLL（*name*，*mode* = *DEFAULT_MODE*，*handle* = *None* ）

这个类的实例的行为与CDLL实例类似，只是在函数调用期间*没有*释放Python GIL ，并且在函数执行后检查了Python错误标志。如果设置了错误标志，则会引发Python异常。

因此，这仅用于直接调用Python C api函数。

所有这些类都可以通过调用至少一个参数（共享库的路径名）来实例化。如果您已经有一个已经加载的共享库的句柄，它可以作为handlenamed参数传递，否则使用底层平台dlopen或LoadLibrary 函数将库加载到进程中，并获得句柄。

该*模式*参数可用于指定库的加载方式。有关详细信息，请参阅*dlopen（3）*联机帮助页。在Windows上，*模式*被忽略。在POSIX系统上，RTLD_NOW总是被添加，并且是不可配置的。

该*use_errno*参数，当设置为true，使一个ctypes机制，允许访问该系统errno以安全的方式错误号。 ctypes维护系统errno 变量的线程局部副本; 如果您在函数调用与ctypes私有副本交换之前调用使用use_errno=True该errno值创建的外部函数，则在 函数调用后立即发生相同的情况。

该函数ctypes.get_errno()返回ctypes私人副本的值，该函数将ctypes私人副本ctypes.set_errno()更改为新值并返回前一个值。

的*use_last_error*参数，设置为true时，使能由所述管理Windows错误代码相同的机制GetLastError()和 SetLastError()Windows API函数； ctypes.get_last_error()并ctypes.set_last_error()用于请求和更改Windows错误代码的ctypes专用副本。

ctypes.RTLD_GLOBAL

用作*模式*参数的标志。在该标志不可用的平台上，它被定义为整数零。

ctypes.RTLD_LOCAL
用作*模式*参数的标志。在无法使用的平台上，它与*RTLD_GLOBAL*相同。

ctypes.DEFAULT_MODE
用于加载共享库的默认模式。在OSX 10.3上，这是*RTLD_GLOBAL*，否则它与*RTLD_LOCAL*相同。

这些类的实例没有公共方法。共享库导出的函数可以作为属性或索引进行访问。请注意，通过属性访问函数会缓存结果，因此每次重复访问它都会返回相同的对象。另一方面，通过索引访问它每次都会返回一个新对象：

```
>>> libc.time == libc.time
True
>>> libc['time'] == libc['time']
False
```

以下公共属性可用，它们的名称以下划线开头，不与导出的函数名称冲突：

PyDLL._handle
用于访问库的系统句柄。

PyDLL._name
在构造函数中传递的库的名称。

共享库也可以通过使用作为LibraryLoader类实例的预制对象之一加载，可以通过调用该LoadLibrary()方法，也可以通过检索该库作为加载器实例的属性。

class ctypes.LibraryLoader(*dlltype*)
加载共享库的类。*dlltype*应该是一个CDLL，PyDLL，WinDLL或OleDLL类型。

__getattr__()有特殊的行为：它允许通过访问共享库作为库加载器实例的属性来加载它。结果被缓存，所以重复的属性访问每次返回相同的库。

LoadLibrary(*名字*)
将共享库加载到进程中并返回。此方法始终返回库的新实例。

这些预制图书馆装载机是可利用的：

ctypes.cdll
创建CDLL实例。

ctypes.windll
仅限Windows：创建WinDLL实例。

ctypes.oledll
仅限Windows：创建OleDLL实例。

ctypes.pydll

创建PyDLL实例。

为了直接访问C Python api，可以使用一个随时可用的Python共享库对象：

ctypes.pythonapi
一个PyDLL将Python C API函数作为属性公开的实例。请注意，所有这些函数都假定返回C int，这当然不总是事实，所以您必须指定正确的restype属性才能使用这些函数。

16.16.2.3。外国函数

如前一节所述，外部函数可以作为加载共享库的属性来访问。默认以这种方式创建的函数对象接受任意数量的参数，接受任何ctypes数据实例作为参数，并返回由库加载器指定的默认结果类型。他们是一个私人课堂的例子：

*类*ctypes._FuncPtr
C可调用外部函数的基类。

外部函数的实例也是C兼容的数据类型; 它们代表C函数指针。

这种行为可以通过分配给外部函数对象的特殊属性来定制。

restype
指定一个ctypes类型来指定外部函数的结果类型。使用None的void，功能不返回任何东西。

可以分配一个不是ctypes类型的可调用Python对象，在这种情况下，假定该函数返回一个C int，并且可调用对象将用该整数调用，从而允许进一步处理或错误检查。不推荐使用此方法，以便更灵活地进行后处理或错误检查，并使用ctypes数据类型作为参数 restype并将可调用参数指定给参数errcheck。

argtypes
分配一个ctypes类型的元组来指定该函数接受的参数类型。使用stdcall调用约定的函数只能使用与此元组长度相同数量的参数来调用; 使用C调用约定的函数也接受其他未指定的参数。

当调用外部函数时，每个实际参数都会传递给 元组中from_param()项目的 类方法 argtypes ，此方法允许将实际参数调整为外部函数接受的对象。例如，元组中的c_char_p项目argtypes将使用ctypes转换规则将作为参数传递的字符串转换为字节对象。

新增功能：现在可以将项目放入不是ctypes类型的argtypes中，但每个项目都必须有一个from_param()返回可用作参数（整数，字符串，ctypes实例）的值的方法。这允许定义可以将自定义对象作为函数参数的适配器。

errcheck
为此属性分配一个Python函数或另一个可调用对象。可调用函数将被调用三个或更多参数：

callable（*结果*，*func*，*参数*）

*结果*是外部函数返回的内容，如restype属性所指定的那样 。

*func*是外部函数对象本身，这允许重用相同的可调用对象来检查或后处理几个函数的结果。

*参数*是一个包含最初传递给函数调用的参数的元组，这允许专门化所使用参数的行为。

该函数返回的对象将从外部函数调用中返回，但它也可以检查结果值并在外部函数调用失败时引发异常。

*异常*ctypes. ArgumentError
当外部函数调用不能转换传递的参数之一时引发此异常。

16.16.2.4。函数原型

外部函数也可以通过实例化函数原型来创建。函数原型与C中的函数原型相似; 他们描述了一个函数（返回类型，参数类型，调用约定），而没有定义实现。必须使用所需的结果类型和函数的参数类型来调用工厂函数。

ctypes. CFUNCTYPE（*restype*，* *argtypes*，*use_errno = False*，*use_last_error = False* ）
返回的函数原型创建使用标准C调用约定的函数。该功能将在通话过程中释放GIL。如果*use_errno*设置为true，系统errno变量的ctypes私人副本将 errno在调用之前和之后与实际值进行交换; *use_last_error*对Windows错误代码也一样。

ctypes. WINFUNCTYPE（*restype*，* *argtypes*，*use_errno = False*，*use_last_error = False* ）
仅适用于Windows：返回的函数原型创建使用stdcall调用约定的函数 ，除了在Windows CE上与之 WINFUNCTYPE()相同的地方CFUNCTYPE()。该功能将在通话过程中释放GIL。*use_errno*和*use_last_error*具有与上面相同的含义。

ctypes. PYFUNCTYPE（*restype*，* *argtypes* ）
返回的函数原型创建使用Python调用约定的函数。在通话过程中，该功能*不会*释放GIL。

这些工厂函数创建的函数原型可以以不同的方式实例化，具体取决于调用中参数的类型和数量：

prototype（*地址*）
返回指定地址处的外部函数，该地址必须是整数。

prototype（*可调用*）
从Python *可调用中*创建一个C可调用函数（回调函数）。

prototype（*func_spec*[，*paramflags*] ）
返回共享库导出的外部函数。*func_spec*必须是2元组。第一项是作为字符串的导出函数的名称，或者是作为小整数的导出函数的序数。第二项是共享库实例。(name_or_ordinal, library)

prototype (*vtbl_index* , *name* [, *paramflags* [, *iid*]])

返回将调用COM方法的外部函数。*vtbl_index*是虚函数表中的索引，一个小的非负整数。*名称*是COM方法的名称。*iid*是用于扩展错误报告的接口标识符的可选指针。

COM方法使用特殊的调用约定：除了在argtypes元组中指定的那些参数外，它们还需要一个指向COM接口的指针作为第一个参数。

可选的*paramflags*参数创建比上述功能更多功能的外部函数包装器。

*paramflags*必须是一个长度与*元素*长度相同的元组argtypes。

此元组中的每个项目都包含有关参数的更多信息，它必须是包含一个，两个或三个项目的元组。

第一个项目是一个包含参数方向标志组合的整数：

1
指定函数的输入参数。

2
输出参数。外部函数填充一个值。

4
输入参数，默认为整数零。

可选的第二项是作为字符串的参数名称。如果指定了这个，可以用命名参数调用外部函数。

可选的第三项是该参数的默认值。

此示例演示如何包装Windows MessageBoxW函数，以便它支持默认参数和命名参数。从Windows头文件的C声明是这样的：

```
WINUSERAPI int WINAPI
MessageBoxW(
    HWND hWnd,
    LPCWSTR lpText,
    LPCWSTR lpCaption,
    UINT uType);
```

这里是包装ctypes：

```
>>> from ctypes import c_int, WINFUNCTYPE, windll
>>> from ctypes.wintypes import HWND, LPCWSTR, UINT
>>> prototype = WINFUNCTYPE(c_int, HWND, LPCWSTR, LPCWSTR, UINT)
>>> paramflags = (1, "hwnd", 0), (1, "text", "Hi"), (1, "caption", "Hello from ctypes
>>> MessageBox = prototype(("MessageBoxW", windll.user32), paramflags)
```

在MessageBox国外功能现在可以被称为在这些方面：

```
>>> MessageBox()
>>> MessageBox(text="Spam, spam, spam")
>>> MessageBox(flags=2, text="foo bar")
```

第二个示例演示输出参数。win32 GetWindowRect 函数通过将它们复制到RECT调用者必须提供的结构中来检索指定窗口的尺寸 。这是C声明：

```
WINUSERAPI BOOL WINAPI
GetWindowRect(
     HWND hWnd,
     LPRECT lpRect);
```

这里是包装ctypes：

```
>>> from ctypes import POINTER, WINFUNCTYPE, windll, WinError
>>> from ctypes.wintypes import BOOL, HWND, RECT
>>> prototype = WINFUNCTYPE(BOOL, HWND, POINTER(RECT))
>>> paramflags = (1, "hwnd"), (2, "lprect")
>>> GetWindowRect = prototype(("GetWindowRect", windll.user32), paramflags)
>>>
```

带有输出参数的函数将自动返回输出参数值（如果有一个参数值），或者当有多个输出参数值时，包含输出参数值的元组将被自动返回，所以GetWindowRect函数现在会在调用时返回一个RECT实例。

输出参数可以与errcheck协议结合使用以进行进一步的输出处理和错误检查。win32 GetWindowRectAPI函数返回一个BOOL信号成功或失败，所以这个函数可以做错误检查，并且在API调用失败时引发异常：

```
>>> def errcheck(result, func, args):
...     if not result:
...         raise WinError()
...     return args
...
>>> GetWindowRect.errcheck = errcheck
>>>
```

如果该errcheck函数返回它接收不变的参数元组，则ctypes继续其在输出参数上执行的正常处理。如果你想返回一个窗口坐标而不是RECT实例的元组 ，你可以检索函数中的字段并返回它们，正常的处理将不再发生：

```
>>> def errcheck(result, func, args):
...     if not result:
...         raise WinError()
...     rc = args[1]
...     return rc.left, rc.top, rc.bottom, rc.right
...
>>> GetWindowRect.errcheck = errcheck
>>>
```

16.16.2.5。效用函数

ctypes.addressof (*obj*)
以整数形式返回内存缓冲区的地址。 *obj*必须是一个ctypes类型的实例。

ctypes.alignment (*obj_or_type*)
返回ctypes类型的对齐要求。*obj_or_type*必须是ctypes类型或实例。

ctypes.byref (*obj* [, *offset*])
返回一个*obj*的轻量级指针，它必须是一个ctypes类型的实例。 *偏移量*默认为零，并且必须是一个将被添加到内部指针值的整数。

`byref(obj, offset)` 对应于这个C代码：

```
(((char *)&obj) + offset)
```

返回的对象只能用作外部函数调用参数。它的行为类似`pointer(obj)`，但建设速度更快。

ctypes.cast (*obj* , *type*)
该函数与C中的转换运算符类似。它返回一个*类型*的新实例，它指向与*obj*相同的内存块。*type*必须是指针类型，*obj*必须是可以被解释为指针的对象。

ctypes.create_string_buffer (*init_or_size* , *size* = *None*)
这个函数创建一个可变的字符缓冲区。返回的对象是一个ctypes数组`c_char`。

*init_or_size*必须是指定数组大小的整数，或者将用于初始化数组项的字节对象。

如果一个字节对象被指定为第一个参数，那么缓冲区会比其长度大一个项目，以便数组中的最后一个元素是一个NUL终止字符。一个整数可以作为第二个参数传递，它允许指定数组的大小，如果不应该使用字节的长度。

ctypes.create_unicode_buffer (*init_or_size* , *size* = *None*)
这个函数创建一个可变的unicode字符缓冲区。返回的对象是一个ctypes数组`c_wchar`。

*init_or_size*必须是指定数组大小的整数，或者是用来初始化数组项的字符串。

如果一个字符串被指定为第一个参数，那么缓冲区会比字符串的长度大一个项目，以便数组中的最后一个元素是一个NUL终止字符。一个整数可以作为第二个参数传递，它允许指定数组的大小，如果不应该使用字符串的长度。

ctypes.DllCanUnloadNow ( )
仅Windows：这个函数是一个钩子，它允许实现具有ctypes的进程内COM服务器。它是从DllCanUnloadNow函数调用的_ctypes扩展dll导出的。

ctypes.DllGetClassObject ( )
仅Windows：这个函数是一个钩子，它允许实现具有ctypes的进程内COM服务器。它从`_ctypes`扩展dll导出的DllGetClassObject函数中调用。

ctypes.util.find_library (*名字*)

尝试找到一个库并返回一个路径名。 *名字*是库名称没有任何前缀一样lib，后缀的相似.so，.dylib或版本号（这是用于POSIX链接器选项的形式-l）。如果没有找到库，则返回None。

确切的功能依赖于系统。

ctypes.util.find_msvcrt()

仅Windows：返回Python使用的VC运行时库的文件名，以及扩展模块。如果库的名称无法确定，None则返回。

如果您需要释放内存（例如，通过调用扩展模块分配内存），则必须在分配内存的同一库中使用此函数。free(void *)

ctypes.FormatError([*code*])

仅适用于Windows：返回的错误代码的文本描述*代码*。如果未指定错误代码，则通过调用Windows API函数GetLastError来使用上一个错误代码。

ctypes.GetLastError()

仅Windows：返回调用线程中由Windows设置的最后一个错误代码。此函数直接调用Windows *GetLastError()*函数，它不会返回错误代码的ctypes-专用副本。

ctypes.get_errno()

返回errno调用线程中系统变量的ctypes-private副本的当前值。

ctypes.get_last_error()

仅Windows：LastError在调用线程中返回系统变量的ctypes-private副本的当前值。

ctypes.memmove(*dst*, *src*, *count*)

与标准C memmove库函数相同：将从*src*到*count*的字节*数*拷贝到*dst*。*dst*和*src*必须是可以转换为指针的整数或ctypes实例。

ctypes.memset(*dst*, *c*, *count*)

与标准C memset库函数相同：填充地址为*dst*的存储器块，其*计数*字节为*c*。*dst*必须是指定地址的整数或ctypes实例。

ctypes.POINTER(*type*)

这个工厂函数创建并返回一个新的ctypes指针类型。指针类型被缓存并在内部重用，所以重复调用这个函数很便宜。*类型*必须是ctypes类型。

ctypes.pointer(*obj*)

这个函数创建一个新的指针实例，指向*obj*。返回的对象是该类型的POINTER(type(obj))。

注意：如果你只是想将一个指向一个对象的指针传递给外部函数调用，你应该使用byref(obj)哪一个更快。

ctypes.resize(*obj*, *size*)

该函数调整*obj*的内部缓冲区，它必须是ctypes类型的一个实例。不可能使缓冲区小于对象类型的本地大小，如下所示sizeof(type(obj))，但可以放大缓冲区。

ctypes.set_errno(*价值*)

将errno调用线程中系统变量的ctypes-private副本的当前值设置为*值*并返回以前的值。

ctypes.set_last_error(*价值*)

仅限Windows：将LastError调用线程中系统变量的ctypes-private副本的当前值设置为*值*并返回以前的值。

ctypes.sizeof(*obj_or_type*)

返回ctypes类型或实例内存缓冲区的大小（以字节为单位）。与C sizeof运营商相同。

ctypes.string_at(*地址*, *大小*=*-1*)

此函数返回从内存地址*地址*开始的C字符串作为字节对象。如果指定了大小，则将其用作大小，否则将假定该字符串为零终止。

ctypes.WinError(*code* = *None*, *descr* = *None*)

仅Windows：这个函数可能是ctypes中命名最差的东西。它创建了一个OSError实例。如果未指定*代码*，GetLastError则调用该代码以确定错误代码。如果未指定*descr*，FormatError()则调用该错误以获取错误的文本描述。

*在版本3.3中更改：*WindowsError用于创建的实例。

ctypes.wstring_at(*地址*, *大小*=*-1*)

这个函数返回了从内存地址的宽字符串*地址*为一个字符串。如果指定了*大小*，则将其用作字符串的字符数，否则将假定该字符串为零终止。

16.16.2.6。数据类型

*类*ctypes._CData

这个非公共类是所有ctypes数据类型的公共基类。除此之外，所有ctypes类型实例都包含一个保存C兼容数据的内存块; 内存块的地址由addressof()辅助函数返回。另一个实例变量暴露为_objects; 这包含其他Python对象，在内存块包含指针的情况下需要保持活动状态。

ctypes数据类型的通用方法，这些都是类方法（确切地说，它们是元类的方法）：

from_buffer(*source*[, *offset*])

此方法返回共享*源*对象的缓冲区的ctypes实例。所述*源*对象必须支持可写缓冲器接口。可选的*offset*参数以字节为单位指定源缓冲区的偏移量; 默认值为零。如果源缓冲区不够大，ValueError则引发。

from_buffer_copy(*source*[, *offset*])

此方法创建一个ctypes实例，从必须可读的*源*对象缓冲区复制缓冲区。可选的*offset*参数以字节为单位指定源缓冲区的偏移量; 默认值为零。如果源缓冲区不够大，ValueError则引发。

from_address(*地址*)

此方法使用*地址*指定的内存返回一个ctypes类型实例，该*地址*必须是整数。

from_param (*obj*)

此方法将*obj*调整为ctypes类型。当外部函数的argtypes元组中存在该类型时，它将与在外部函数调用中使用的实际对象一起调用; 它必须返回一个可以用作函数调用参数的对象。

所有的ctypes数据类型都有这个classmethod的默认实现，如果这是一个类型的实例，通常会返回*obj*。有些类型也接受其他对象。

in_dll (*库*, *名称*)

此方法返回由共享库导出的ctypes类型实例。*name*是导出数据的符号的名称，*library*是加载的共享库。

ctypes数据类型的通用实例变量：

_b_base_

有时ctypes数据实例不拥有它们包含的内存块，而是共享基础对象的部分内存块。所述 _b_base_只读构件是根ctypes的对象拥有该存储器块。

_b_needsfree_

此只读变量在ctypes数据实例已分配内存块本身时为true，否则为false。

_objects

此成员是None包含需要保持活动状态的Python对象的字典或字典，以便内存块内容保持有效。该对象仅用于调试; 永远不要修改这本词典的内容。

16.16.2.7。基本数据类型

*类*ctypes._SimpleCData

这个非公共类是所有基本ctypes数据类型的基类。这里提到它是因为它包含基本ctypes数据类型的通用属性。 _SimpleCData是它的一个子类 _CData，所以它继承了它们的方法和属性。现在可以对不包含指针的ctypes数据类型进行酸洗。

实例具有单个属性：

value

该属性包含实例的实际值。对于整数和指针类型，它是一个整数，对于字符类型，它是单字节字节对象或字符串，对于字符指针类型，它是一个Python字节对象或字符串。

当value从ctypes实例中检索属性时，通常每次都会返回一个新对象。 ctypes*并没有*实现原来的目标回报率，总是一个新的对象被创建。所有其他ctypes对象实例也是如此。

基本数据类型在作为外部函数调用结果返回时，或者例如通过检索结构字段成员或数组项时，会透明地转换为本机Python类型。换句话说，如果一个外国函数有一个 restype的c_char_p，你总是会收到一个Python字节的对象，*不是*一个c_char_p实例。

基本数据类型的子类*不会*继承此行为。所以，如果一个外部函数restype是它的一个子类c_void_p，你将从函数调用中得到这个子类的一个实例。当然，你可以通过访问value属性来获得指针的值。

这些是基本的ctypes数据类型：

*类*ctypes.**c_byte**

表示C 数据类型，并将该值解释为小整数。构造函数接受一个可选的整数初始值设定项; 没有溢出检查完成。signed char

*类*ctypes.**c_char**

表示C char数据类型，并将该值解释为单个字符。构造函数接受一个可选的字符串初始值设定项，字符串的长度必须恰好为一个字符。

*类*ctypes.**c_char_p**

当它指向零终止的字符串时表示C 数据类型。对于也可能指向二进制数据的通用字符指针， 必须使用。该构造函数接受一个整数地址或一个字节对象。char *POINTER(c_char)

*类*ctypes.**c_double**

表示C double数据类型。构造函数接受一个可选的浮点初始值设定项。

*类*ctypes.**c_longdouble**

表示C 数据类型。构造函数接受一个可选的浮点初始值设定项。在它是别名的平台上。long doublesizeof(long double) == sizeof(double)c_double

*类*ctypes.**c_float**

表示C float数据类型。构造函数接受一个可选的浮点初始值设定项。

*类*ctypes.**c_int**

表示C 数据类型。构造函数接受一个可选的整数初始值设定项; 没有溢出检查完成。在它是别名的平台上。signed intsizeof(int) == sizeof(long)c_long

*类*ctypes.**c_int8**

表示C 8位数据类型。通常是别名 。signed intc_byte

*类*ctypes.**c_int16**

表示C 16位数据类型。通常是别名 。signed intc_short

*类*ctypes.**c_int32**

表示C 32位数据类型。通常是别名 。signed intc_int

*类*ctypes.**c_int64**

表示C 64位数据类型。通常是别名 。signed intc_longlong

*类*ctypes.**c_long**

表示C 数据类型。构造函数接受一个可选的整数初始值设定项; 没有溢出检查完成。signed long

*类*ctypes.**c_longlong**

表示C 数据类型。构造函数接受一个可选的整数初始值设定项; 没有溢出检查完成。 signed long long

*类*ctypes.**c_short**

表示C 数据类型。构造函数接受一个可选的整数初始值设定项; 没有溢出检查完成。 signed short

*类*ctypes.**c_size_t**

表示C size_t数据类型。

*类*ctypes.**c_ssize_t**

表示C ssize_t数据类型。

3.2版本中的新功能

*类*ctypes.**c_ubyte**

表示C 数据类型，它将该值解释为小整数。构造函数接受一个可选的整数初始值设定项; 没有溢出检查完成。 unsigned char

*类*ctypes.**c_uint**

表示C 数据类型。构造函数接受一个可选的整数初始值设定项; 没有溢出检查完成。在它是别名的平台上。 unsigned int sizeof(int) == sizeof(long) c_ulong

*类*ctypes.**c_uint8**

表示C 8位数据类型。通常是别名 。 unsigned int c_ubyte

*类*ctypes.**c_uint16**

表示C 16位数据类型。通常是别名 。 unsigned int c_ushort

*类*ctypes.**c_uint32**

表示C 32位数据类型。通常是别名 。 unsigned int c_uint

*类*ctypes.**c_uint64**

表示C 64位数据类型。通常是别名 。 unsigned int c_ulonglong

*类*ctypes.**c_ulong**

表示C 数据类型。构造函数接受一个可选的整数初始值设定项; 没有溢出检查完成。 unsigned long

*类*ctypes.**c_ulonglong**

表示C 数据类型。构造函数接受一个可选的整数初始值设定项; 没有溢出检查完成。 unsigned long long

*类*ctypes.**c_ushort**

表示C 数据类型。构造函数接受一个可选的整数初始值设定项; 没有溢出检查完成。 unsigned short

*类*ctypes.**c_void_p**

代表C 类型。该值表示为整数。构造函数接受一个可选的整数初始值设定项。 void *

*类*ctypes.c_wchar

表示C wchar_t数据类型，并将该值解释为单个字符的unicode字符串。构造函数接受一个可选的字符串初始值设定项，字符串的长度必须恰好为一个字符。

*类*ctypes.c_wchar_p

表示C 数据类型，它必须是一个指向零终止的宽字符字符串的指针。构造函数接受一个整数地址或一个字符串。wchar_t *

*类*ctypes.c_bool

表示C bool数据类型（更准确地说，_Bool来自C99）。其值可以是True或False，和构造函数接受具有真值的任何对象。

*类*ctypes.HRESULT

仅限Windows：表示一个HRESULT值，其中包含函数或方法调用的成功或错误信息。

*类*ctypes.py_object

表示C 数据类型。不带参数调用它会创建一个指针。PyObject *NULL PyObject *

该ctypes.wintypes模块提供了相当长的一段其他Windows特定的数据类型，例如HWND，WPARAM或DWORD。一些有用的结构像MSG或者RECT也被定义。

16.16.2.8。结构化数据类型

class ctypes.Union(**args*, ***kw*)

本地字节顺序的联合的抽象基类。

class ctypes.BigEndianStructure(**args*, ***kw*)

*大端*字节顺序结构的抽象基类。

class ctypes.LittleEndianStructure(**args*, ***kw*)

*小端*字节顺序结构的抽象基类。

具有非本地字节顺序的结构不能包含指针类型字段或包含指针类型字段的任何其他数据类型。

class ctypes.Structure(**args*, ***kw*)

*本地*字节顺序结构的抽象基类。

具体结构和联合类型必须通过继承这些类型之一来创建，并至少定义一个_fields_类变量。ctypes将创建描述符，允许通过直接属性访问来读写字段。这些是

fields

定义结构字段的序列。这些项目必须是2元组或3元组。第一个项目是字段的名称，第二个项目指定字段的类型; 它可以是任何ctypes数据类型。

对于整数类型字段c_int，可以给出第三个可选项。它必须是一个定义字段位宽的小正整数。

一个结构或联合中的字段名称必须是唯一的。这不会被检查，重复名称时只能访问一个字段。

可以在定义Structure子类的类语句*之后*定义_fields_类变量，这允许创建直接或间接引用其自身的数据类型：

```
class List(Structure):
    pass
List._fields_ = [("pnext", POINTER(List)),
                 ...
                ]
```

的_fields_类变量但是，必须被定义在第一次使用之前的类型（创建一个实例，sizeof()被称为其上，等等）。稍后分配给_fields_类变量将引发AttributeError。

可以定义结构类型的子子类，它们继承基类的字段加上_fields_在子子类中定义的字段（如果有的话）。

pack

一个可选的小整数，它允许覆盖实例中结构字段的对齐方式。 _pack_必须在_fields_分配时已经被定义，否则它将不起作用。

anonymous

列出未命名（匿名）字段名称的可选序列。 _anonymous_必须在_fields_分配时已经定义，否则将不起作用。

此变量中列出的字段必须是结构或联合类型字段。 ctypes将在结构类型中创建描述符，允许直接访问嵌套字段，而不需要创建结构或联合字段。

这是一个示例类型（Windows）：

```
class _U(Union):
    _fields_ = [("lptdesc", POINTER(TYPEDESC)),
                ("lpadesc", POINTER(ARRAYDESC)),
                ("hreftype", HREFTYPE)]

class TYPEDESC(Structure):
    _anonymous_ = ("u",)
    _fields_ = [("u", _U),
                ("vt", VARTYPE)]
```

该TYPEDESC结构描述了一个COM数据类型，该vt字段指定哪个联合字段是有效的。由于该u字段被定义为匿名字段，因此现在可以直接访问TYPEDESC实例之外的成员。td.lptdesc并且td.u.lptdesc 是等价的，但前者更快，因为它不需要创建临时联合实例：

```
td = TYPEDESC()
td.vt = VT_PTR
td.lptdesc = POINTER(some_type)
td.u.lptdesc = POINTER(some_type)
```

可以定义结构的子子类，它们继承基类的字段。如果子类定义具有单独的 _fields_变量，则在此指定的字段将附加到基类的字段中。

结构和联合构造函数接受位置和关键字参数。位置参数用于按照出现的顺序初始化成员字段_fields_。构造函数中的关键字参数被解释为属性赋值，因此它们将_fields_使用相同的名称进行初始化 ，或者为不存在的名称创建新的属性_fields_。

16.16.2.9。数组和指针

class ctypes. Array (**args*)

数组的抽象基类。

推荐的创建具体数组类型的方法是将任何ctypes数据类型乘以 正整数。或者，您可以继承此类型并定义_length_和_type_分类变量。数组元素可以使用标准的下标和切片访问来读写; 对于切片读取，所得到的物体是 *不*本身Array。

length

指定数组中元素数量的正整数。超出范围的下标导致一个 IndexError。将被退回len()。

type

指定数组中每个元素的类型。

数组子类的构造函数接受位置参数，用于按顺序初始化元素。

*类*ctypes. _Pointer

指针的私有抽象基类。

具体指针类型通过调用 POINTER() 将被指向的类型来创建; 这是通过自动完成的pointer()。

如果指针指向数组，则可以使用标准下标和切片访问来读取和写入其元素。指针对象没有大小，所以len()会提高TypeError。负指标将在指针*之前*从内存中读取（如C中所示），并且超出范围的下标可能会因访问冲突（如果幸运的话）而崩溃。

type

指定指向的类型。

contents

返回要指向点的对象。分配给该属性会将指针更改为指向指定的对象。

17.并发执行

本章描述的模块为并发执行代码提供支持。工具的适当选择取决于要执行的任务（CPU绑定与IO绑定）以及首选的开发风格（事件驱动的协作式多任务与抢先式多任务）。这里有一个概述：

以下是一些上述服务的支持模块：

17.1。 threading- 基于线程的并行处理

源代码： Lib / threading.py

该模块在较低级别的_thread模块上构建较高级别的线程接口。另请参阅queue模块。

该dummy_threading模块适用于threading因_thread缺失而无法使用的情况 。

> **注意:** 尽管它们未在下面列出，但camelCase本模块仍支持Python 2.x系列中用于此模块中某些方法和函数的名称。

该模块定义了以下功能：

threading.**active_count** ( )
返回Thread当前活着的对象数量。返回的计数等于返回的列表的长度enumerate()。

threading.**current_thread** ( )
返回当前Thread对象，对应于调用者的控制线程。如果调用者的控制线程不是通过threading模块创建的 ，则返回具有有限功能的虚拟线程对象。

threading.**get_ident** ( )
返回当前线程的'线程标识符'。这是一个非零整数。它的价值没有直接的意义; 它的目的是作为一个神奇的cookie用于例如索引线程特定数据的字典。当线程退出并创建另一个线程时，线程标识符可能会被回收。

3.3版本的新功能

threading.**enumerate** ( )
返回Thread当前活着的所有对象的列表。该列表包括守护进程线程，由其创建的虚拟线程对象 current_thread()以及主线程。它排除了尚未启动的已终止的线程和线程。

threading.**main_thread** ( )
返回主Thread对象。在正常情况下，主线程是Python解释器启动的线程。

3.4版新增功能

threading.**settrace** (*func*)
为从threading模块启动的所有线程设置跟踪功能。该*FUNC*将被传递给 sys.settrace()每个线程，它之前 run()被调用的方法。

threading.**setprofile** (*func*)
为从threading模块启动的所有线程设置配置文件功能。该*FUNC*将被传递给 sys.setprofile()每个线程，它之前 run()被调用的方法。

threading.**stack_size** ([*size*])

返回创建新线程时使用的线程堆栈大小。可选的 *size*参数指定要用于随后创建的线程的堆栈大小，并且必须为0（使用平台或配置的默认值）或至少32,768（32 KiB）的正整数值。如果未指定*大小*，则使用0。如果不支持更改线程堆栈大小，RuntimeError则会引发a。如果指定的堆栈大小无效，aValueError并且堆栈大小未修改。32 KiB是目前支持的最小堆栈大小值，以保证解释器本身具有足够的堆栈空间。请注意，某些平台可能对堆栈大小的值有特别的限制，例如要求最小堆栈大小> 32 KiB或需要以系统内存页大小的倍数进行分配 - 有关更多信息，请参阅平台文档（4 KiB页是常见的;对于堆栈大小，使用4096的倍数是没有更具体信息的建议方法）。可用性：Windows，带有POSIX线程的系统。

该模块还定义了以下常量：

threading.TIMEOUT_MAX
允许的最大值*的超时*阻断功能的（参数Lock.acquire()，RLock.acquire()，Condition.wait()等等）。指定超过此值的超时将引发一次OverflowError。

3.2版本中的新功能

该模块定义了许多类，这些类将在下面的章节中详细介绍。

这个模块的设计松散地基于Java的线程模型。但是，在Java为锁和条件变量生成每个对象的基本行为的地方，它们是Python中的单独对象。Python的Thread类支持Java的Thread类的一部分行为; 目前没有优先级，没有线程组，线程不能被销毁，停止，暂停，恢复或中断。Java的Thread类的静态方法在实现时映射到模块级函数。

下面描述的所有方法都是以原子方式执行的。

17.1.1。线程局部数据

线程本地数据是其值为线程特定的数据。要管理线程本地数据，只需创建一个local（或一个子类）的实例并在其上存储属性：

```
mydata = threading.local()
mydata.x = 1
```

实例的值对于单独的线程将会有所不同。

*类*threading.local
表示线程本地数据的类。

有关更多详细信息和广泛的示例，请参阅_threading_local模块的文档字符串。

17.1.2。线程对象

的Thread类表示在单独的控制线程中运行的活动。有两种方法可以指定活动：将可调用对象传递给构造函数，或者通过重写run()子类中的方法。在子类中不应该重写其他方法（构造函数除外）。换句话说，*仅*覆盖__init__()和run()这个类的方法。

一旦线程对象被创建，它的活动必须通过调用线程的start()方法来启动。这将run() 在一个单独的控制线程中调用该方法。

一旦线程的活动开始，线程被认为是“活着的”。当它的run()方法终止时，它会停止活动- 通常或者通过引发未处理的异常。该is_alive() 方法测试线程是否存在。

其他线程可以调用线程的join()方法。这会阻塞调用线程，直到其join()方法被调用的线程终止。

一个线程有一个名字。该名称可以传递给构造函数，并通过name属性读取或更改。

线程可以被标记为“守护线程”。这个标志的意义在于，只有守护进程线程退出时，整个Python程序才会退出。初始值是从创建线程继承的。该标志可以通过daemon属性或*守护进程*构造函数参数来设置。

> **注意：** 守护程序线程在关机时突然停止。他们的资源（如打开文件，数据库事务等）可能无法正确释放。如果你想让你的线程正常停止，使它们不是守护进程，并使用合适的信号机制，如Event。

有一个“主线程”对象; 这对应于Python程序中的初始控制线程。它不是一个守护线程。

有可能会创建“虚拟线程对象”。这些是与“外来线程”对应的线程对象，它们是在线程模块外部启动的控制线程，例如直接来自C代码。虚拟线程对象的功能有限; 他们总是被认为是活着和神圣的，而且不能被join()编辑。它们从不被删除，因为不可能检测到外来线程的终止。

class threading.Thread（*group = None*，*target = None*，*name = None*，*args = ( )*，*kwargs = {}*，*，*daemon = None* ）

应始终使用关键字参数调用此构造函数。参数是：

*组*应该是None; 在ThreadGroup实施课程时预留给将来的延期 。

*target*是要由run()方法调用的可调用对象。默认为None ，意味着什么都不叫。

*名称*是线程名称。默认情况下，一个唯一的名称由“Thread- *N* ”形式构成，其中*N*是小数字。

*args*是目标调用的参数元组。默认为 ()。

*kwargs*是目标调用的关键字参数字典。默认为 {}。

如果不是None ，*守护进程会*明确设置该线程是否为*守护进程*。如果None（默认），守护进程属性是从当前线程继承的。

如果子类重写构造函数，则必须确保Thread.__init__()在对线程执行任何其他操作之前调用基类构造函数（ ）。

*在版本3.3中进行了更改：*添加了*守护进程*参数。

start（ ）

开始线程的活动。

每个线程对象最多只能调用一次。它安排run()在独立的控制线程中调用对象的方法。

这个方法会RuntimeError在同一个线程对象上多次调用一次。

run ( )

表示线程活动的方法。

您可以在子类中重写此方法。标准run() 方法调用传递给对象构造函数的可调用对象作为*目标*参数，如果有的话，分别从*args*和*kwargs*参数中获取顺序和关键字参数。

join (*timeout* = *None*)

等到线程终止。这会阻塞调用线程，直到调用其join()方法的线程终止 - 通常或通过未处理的异常 - 或直到发生可选的超时。

如果*超时*参数不存在None，它应该是一个浮点数，以秒为单位指定该操作的超时值（或其分数）。一如join()往常返回None，您必须调用is_alive() after join()以决定是否发生超时 - 如果线程仍处于活动状态，则 join()呼叫超时。

当*timeout*参数不存在时None，该操作将阻塞，直到线程终止。

一个线程可以被join()多次编辑。

join()提出了RuntimeError如果试图加入当前线程因为这将导致死锁。 join()在线程启动之前它也是一个错误，并且尝试这样做会引发相同的异常。

name

一个字符串，仅用于识别目的。它没有语义。多个线程可以被赋予相同的名称。初始名称由构造函数设置。

getName ( )
setName ( )

旧的getter / setter API name; 改为直接将其用作属性。

ident

此线程的'线程标识符'或者None线程尚未启动。这是一个非零整数。查看get_ident()功能。当线程退出并创建另一个线程时，线程标识符可能会被回收。该标识符即使在线程退出后也可用。

is_alive ( )

返回线程是否存在。

该方法True在run()方法开始之前返回，直到run()方法终止。模块函数enumerate()返回所有活动线程的列表。

daemon

指示此线程是否为守护程序线程（True）或不（False）的布尔值。这必须在start()调用之前设置，否则RuntimeError会引发。它的初始值是从创建线程继承的; 主线程不是守护进程线程，因此在主线程中创建的所有线程均默认为 daemon= False。

当没有活动的非守护进程线程时，整个Python程序将退出。

isDaemon ( )
setDaemon ( )
旧的getter / setter API daemon; 改为直接将其用作属性。

CPython实现细节：在CPython中，由于全局解释器锁定，只有一个线程可以一次执行Python代码（即使某些面向性能的库可能会克服此限制）。如果您希望您的应用程序更好地利用多核机器的计算资源，建议您使用 multiprocessing或concurrent.futures.ProcessPoolExecutor。但是，如果要同时运行多个I / O限制任务，线程仍然是一个合适的模型。

17.1.3。锁定对象

原始锁定是一种同步原语，在锁定时不属于特定线程。在Python中，它是当前可用的最低级同步原语，由_thread 扩展模块直接实现。

原始锁处于“锁定”或“解锁”两种状态之一。它在解锁状态下创建。它有两个基本的方法，acquire()和 release()。状态解锁时，acquire() 将状态更改为锁定并立即返回。当状态被锁定时， acquire()阻塞直到release()在另一个线程中的呼叫将其更改为解锁，然后acquire()呼叫将其重置为锁定并返回。该release()方法只能在锁定状态下调用; 它将状态更改为解锁并立即返回。如果试图释放未锁定的锁， RuntimeError则会提出。

锁也支持上下文管理协议。

当多个线程在acquire()等待状态转为解锁状态时被阻塞时，当一个release() 呼叫将状态重置为解锁状态时，只有一个线程继续进行; 哪一个等待的线程没有被定义，并且可能会因实现而有所不同。

所有的方法都是自动执行的。

*类*threading. Lock
实现原始锁定对象的类。一旦一个线程获得一个锁，后续的获取它的尝试将被阻塞，直到它被释放; 任何线程都可能释放它。

请注意，Lock它实际上是一个工厂函数，它返回平台支持的具体Lock类的最有效版本的实例。

acquire (*blocking = True*，*timeout = -1*)
获取锁定，阻止或不阻止。

当调用*阻塞*参数设置为True（默认）时，阻塞直到锁定解锁，然后将其设置为锁定并返回True。

当调用*阻塞*参数设置为False，不要阻塞。如果*阻塞*设置的呼叫True会阻塞，False 立即返回; 否则，将锁设置为锁定并返回True。

当使用设置为正值的浮点*超时*参数进行调用*时*，只要锁定无法获取，最多会阻止*超时*指定的秒数。一个*超时*的参数-1 指定了一个无限的等待。 当*阻塞*为false 时，禁止指定*超时*。

返回值True是否成功获取锁定（ False如果没有）（例如，如果*超时*已过期）。

*改变在3.2版本：*该*超时*参数是新的。

*在版本3.2中更改：*锁定采集现在可以被POSIX上的信号中断。

release ( )

释放一个锁。这可以从任何线程中调用，而不仅仅是已经获得锁的线程。

锁定被锁定时，将其重置为解锁状态，然后返回。如果有其他线程被阻塞，等待锁被解锁，请准确地让其中的一个继续。

在解锁的锁上调用时，RuntimeError会引发a。

没有返回值。

17.1.4。RLock对象

可重入锁定是同一个原语，它可能被同一个线程多次获取。在内部，除了原始锁使用的锁定/解锁状态外，它还使用“拥有线程”和“递归级别”的概念。在锁定状态下，某个线程拥有该锁; 在解锁状态下，没有线程拥有它。

为了锁定锁定，一个线程调用它的acquire()方法; 这将在线程拥有锁定时返回。为了解锁，一个线程调用它的release()方法。 acquire()/ release() 调用对可以嵌套; 只有最后一对release()（ release()最外端的一对）重置锁以解锁，并允许另一个线程被阻止acquire()进行。

再入式锁也支持上下文管理协议。

*类*threading. RLock

这个类实现了可重入的锁对象。重入锁必须由获取它的线程释放。一旦一个线程获得了一个可重入的锁，同一个线程可以再次获得它而不会阻塞; 线程每次获取它时都必须释放一次。

请注意，RLock它实际上是一个工厂函数，它返回平台支持的具体RLock类的最高效版本的实例。

acquire (*blocking = True*，*timeout = -1*)

获取锁定，阻止或不阻止。

在没有参数的情况下调用时：如果此线程已拥有该锁，则递归级别递增1，并立即返回。否则，如果另一个线程拥有该锁，则阻塞直到该锁被解锁。一旦锁定被解锁（不属于任何线程），然后获取所有权，将递归级别设置为1，然后返回。如果有多个线程被阻塞，等待锁定解锁，则一次只能有一个线程能够获取该锁的所有权。在这种情况下没有返回值。

当使用设置为true 的*阻塞*参数进行调用时，请执行与不使用参数时调用相同的操作，并返回true。

当将*blocking*参数设置为false 时调用时，请勿阻止。如果没有参数的调用会被阻塞，立即返回false; 否则，与没有参数调用时一样，返回true。

当使用设置为正值的浮点*超时*参数进行调用*时*，只要锁定无法获取，最多会阻止*超时*指定的秒数。如果已获取锁，则返回true，如果超时已过，则返回false。

*改变在3.2版本：*该*超时*参数是新的。

release ( )

释放一个锁，递减递归级别。如果在递减之后它为零，则将锁重置为解锁（不属于任何线程），并且如果有其他线程被阻塞，等待锁解锁，请准确地让其中一个继续。如果在递减之后递归级别仍然不为零，则锁保持锁定并由调用线程拥有。

只有在调用线程拥有锁定时才调用此方法。 RuntimeError如果在解锁锁定时调用此方法，则会引发A.

没有返回值。

17.1.5。条件对象

一个条件变量总是与某种锁相关联; 这可以通过或默认创建一个。当多个条件变量必须共享相同的锁时，传入一个是很有用的。该锁是条件对象的一部分：您不必单独跟踪它。

条件变量服从上下文管理协议：使用该with语句在封闭块的持续时间内获取关联的锁。该acquire()和 release()方法也调用相关的锁的相应方法。

必须调用其他方法并保持关联的锁。该 wait()方法释放锁，然后阻塞，直到另一个线程通过调用notify()或 唤醒它notify_all()。一旦醒来， wait() 重新获得锁定并返回。也可以指定超时。

该notify()方法唤醒等待条件变量的线程之一，如果有任何正在等待的话。该notify_all() 方法唤醒等待条件变量的所有线程。

注意：notify()和notify_all()方法不释放锁; 这意味着被唤醒的一个或多个线程不会wait()立即从它们的调用中返回，而只是在调用notify()或notify_all() 最终放弃锁的所有权的线程时。

使用条件变量的典型编程风格使用锁来同步对某些共享状态的访问; 那些对特定的状态改变感兴趣的线程， wait() 直到他们看到所需的状态，而修改状态调用的线程notify() 或者notify_all()当他们改变状态时，这些线程 可能成为其中一个服务员的期望状态。例如，以下代码是具有无限缓冲区容量的通用生产者 - 消费者情况：

```
# Consume one item
with cv:
    while not an_item_is_available():
        cv.wait()
    get_an_available_item()

# Produce one item
with cv:
    make_an_item_available()
    cv.notify()
```

该while循环检查应用程序的条件是必要的，因为wait()可以任意长的时间后返回，并促使病情notify()通话可能不再成立。这是多线程编程所固有的。该 wait_for()方法可用于自动执行条件检查，并简化超时计算：

```
# Consume an item
with cv:
    cv.wait_for(an_item_is_available)
    get_an_available_item()
```

要选择notify()和notify_all() ，考虑一个状态变化是否可以只为一个或几个等待的线程有趣。例如，在典型的生产者 - 消费者情况下，向缓冲区添加一个项目只需要唤醒一个消费者线程。

class threading. Condition (*lock = None*)

这个类实现了条件变量对象。一个条件变量允许一个或多个线程等待，直到它们被另一个线程通知。

如果给出了*lock*参数而不是None ，它必须是一个Lock 或者一个RLock对象，并且它被用作底层锁。否则，将RLock创建一个新对象并将其用作基础锁。

*在版本3.3中更改：*从工厂功能更改为类。

acquire (**args*)

获取底层锁。该方法调用底层锁的相应方法; 返回值是该方法返回的值。

release ( )

释放底层的锁。该方法调用底层锁的相应方法; 没有返回值。

wait (*timeout = None*)

等到通知或直到发生超时。如果在调用此方法时调用线程未获取锁，RuntimeError则会引发a。

此方法释放底层锁，然后阻塞，直到它被另一个线程中的相同条件变量唤醒notify()或notify_all()调用，直到发生可选超时。一旦被唤醒或超时，它将重新获得锁定并返回。

如果*超时*参数不存在None ，它应该是一个浮点数，以秒为单位指定该操作的超时值（或其分数）。

当底层锁是an时RLock ，它不会使用它的release()方法释放，因为当递归获取多次时，这实际上可能不会解锁该锁。相反，使用RLock该类的内部接口，即使已递归获取多次，也可以真正解锁该接口。当锁重新获得时，另一个内部接口用于恢复递归级别。

返回值是True除非给定的*超时*过期，在这种情况下False。

*在版本3.2中更改：*以前，该方法总是返回None。

wait_for (*谓词*, *超时=无*)

等到条件评估为真。 *谓词*应该是可调用的，其结果将被解释为布尔值。一个*超时*可以提供给最长等待时间。

此实用程序方法可能会wait()重复调用，直到满足谓词或发生超时。返回值是谓词的最后一个返回值，并将评估 False该方法是否超时。

忽略超时功能，调用此方法大致等同于编写：

```
while not predicate():
    cv.wait()
```

因此，同样的规则适用于wait()：锁定必须在被调用时保持，并在返回时重新获取。谓词在持有锁的情况下进行评估。

3.2版本中的新功能

notify (*n = 1*)

默认情况下，唤醒等待此条件的一个线程（如果有）。如果在调用此方法时调用线程未获取锁， RuntimeError则会引发a。

此方法至多唤醒*n个*等待条件变量的线程; 如果没有线程正在等待，则它是无操作的。

如果至少有*n个*线程正在等待，那么当前实现将醒来*n 个*线程。但是，依靠这种行为是不安全的。未来，优化的实现可能偶尔唤醒超过 *n个*线程。

注意：一个唤醒的线程实际上并没有从它的wait() 调用返回，直到它可以重新获得锁。由于notify()不释放锁，因此其调用者应该。

notify_all ( )

唤醒等待这种情况的所有线程。这个方法的作用就像 notify()，但唤醒所有等待的线程而不是一个。如果在调用此方法时调用线程未获取锁， RuntimeError则会引发a。

17.1.6。semaphore对象。

这是计算机科学的历史上最古老的同步原语之一，由早期荷兰计算机科学家艾兹赫尔·戴克斯特拉（他使用的名称发明P()和V()替代acquire()和 release()）。

信号量管理一个内部计数器，每次acquire()调用都会减少内部计数器，并随每次 调用而递增release() 。柜台不能低于零; 当acquire() 发现它为零时，它会阻塞，等待其他线程调用release()。

信号量也支持上下文管理协议。

class threading. Semaphore (*value = 1*)

这个类实现信号量对象。信号量管理一个原子计数器，代表release()呼叫数量减去acquire()呼叫数量 加上一个初始值。该acquire()方法根据需要进行阻止，直到它可以返回而不使计数器为负。如果没有给出，*值*默认为1。

可选参数给出了内部计数器的初始*值*；它默认为1。如果给定的*值*小于0，ValueError则升高。

*在版本3.3中更改：*从工厂功能更改为类。

acquire（*blocking = True*，*timeout = None*）

获取信号量。

当没有参数被调用时：

- 如果内部计数器在输入时大于零，则将其减1并立即返回true。
- 如果内部计数器在输入时为零，则阻止直到通过呼叫唤醒 release()。一旦唤醒（且计数器大于0），将计数器减1并返回true。每次打电话都会醒来一个线程release()。不应该依赖线程被唤醒的顺序。

当调用*阻塞*设置为false时，不要阻塞。如果没有参数的调用会被阻塞，立即返回false；否则，与没有参数调用时一样，返回true。

当以*超时*以外的方式调用时None，它将最多*暂停*秒。如果获取在该时间间隔内未成功完成，则返回false。否则返回true。

*改变在3.2版本：*该*超时*参数是新的。

release（）

释放一个信号量，将内部计数器递增1。当它在入口处为零并且另一个线程正在等待它再次变得大于零时，唤醒该线程。

class threading. BoundedSemaphore（*value = 1*）

类实现有界信号量对象。有界信号量会检查以确保其当前值不超过其初始值。如果有，ValueError就提出来。在大多数情况下，信号量用于保护容量有限的资源。如果信号量被释放太多次，这是一个错误的迹象。如果没有给出，*值*默认为1。

*在版本3.3中更改：*从工厂功能更改为类。

17.1.6.1。Semaphore示例

信号量通常用于防范容量有限的资源，例如数据库服务器。在任何情况下，资源的大小是固定的，你应该使用有界的信号量。在产生任何工作线程之前，主线程会初始化信号量：

```
maxconnections = 5
# ...
pool_sema = BoundedSemaphore(value=maxconnections)
```

一旦产生，工作线程在需要连接到服务器时调用信号量的获取和释放方法：

```
with pool_sema:
    conn = connectdb()
    try:
        # ... use connection ...
    finally:
        conn.close()
```

有界信号量的使用降低了导致信号量被释放的程序错误超过它所获得的错误的可能性。

17.1.7。事件对象

这是线程间最简单的通信机制之一：一个线程表示一个事件，其他线程等待它。

事件对象管理一个内部标志，该标志可以通过该set()方法设置为true，并通过该 方法重置为false clear() 。该wait()方法阻塞直到标志为真。

*类*threading.Event

类实现事件对象。一个事件管理一个标志，该标志可以通过该set()方法设置为true，并通过该方法重置为false clear()。该wait()方法阻塞直到标志为真。国旗最初是假的。

*在版本3.3中更改：*从工厂功能更改为类。

is_set ( )

当且仅当内部标志为真时返回true。

set ( )

将内部标志设置为true。所有等待它成为真的线程都被唤醒。 wait()一旦标志为真，调用的线程根本不会阻塞。

clear ( )

将内部标志重置为false。随后，线程调用 wait()将被阻塞，直到set()被调用以将内部标志再次设置为真。

wait (*timeout = None*)

阻塞直到内部标志为真。如果内部标志在输入时为真，则立即返回。否则，阻塞直到另一个线程调用 set()将该标志设置为true，或者直到发生可选的超时。

如果超时参数不存在None，它应该是一个浮点数，以秒为单位指定该操作的超时值（或其分数）。

当且仅当内部标志已设置为true时（无论是在等待调用之前还是在等待开始之后），此方法都会返回true，因此True除非发生超时并且操作超时，否则它将始终返回。

*在版本3.1中更改：*以前，该方法始终返回None。

17.1.8。计时器对象

这个类表示一个只有经过一段时间才能运行的动作 - 一个计时器。 Timer是其子类，Thread 因此也可以作为创建自定义线程的示例。

定时器和线程一样，通过调用它们的start() 方法来启动。通过调用该cancel()方法可以停止定时器（在动作开始之前）。计时器在执行其动作之前等待的时间间隔可能与用户指定的时间间隔不完全相同。

例如：

```
def hello():
    print("hello, world")

t = Timer(30.0, hello)
t.start()  # after 30 seconds, "hello, world" will be printed
```

class threading.Timer (*interval* , *function* , *args = None* , *kwargs = None*)

创建一个计时器，它会在*时间间隔*秒后通过参数*args*和关键字参数*kwargs*运行*函数*。如果*参数*是（缺省值），则会使用空列表。如果*kwargs*是（默认），那么将会使用一个空的字典。NoneNone

*在版本3.3中更改：*从工厂功能更改为类。

cancel ( )

停止定时器，并取消定时器动作的执行。这只有在计时器仍处于等待阶段时才有效。

17.1.9。屏障对象

3.2版本中的新功能

该类提供了一个简单的同步原语，供固定数量的需要彼此等待的线程使用。每个线程都会尝试通过调用该wait()方法来传递障碍，并阻塞直到所有线程都进行了wait()调用。此时，线程被同时释放。

对于相同数量的线程，屏障可以重复使用任意次数。

作为一个例子，下面是一个简单的同步客户端和服务器线程的方法：

```
b = Barrier(2, timeout=5)

def server():
    start_server()
    b.wait()
    while True:
        connection = accept_connection()
        process_server_connection(connection)

def client():
    b.wait()
    while True:
        connection = make_connection()
        process_client_connection(connection)
```

*类*threading.Barrier (*派对* , *动作=无* , *超时=无*)

为*派对*的线程数量创建一个障碍对象。一个*动作*，当提供时，是一个可调用的线程在被释放时被一个线程调用。 如果没有为该方法指定任何值，则*timeout*是默认的超时值wait()。

wait (*timeout = None*)

通过障碍。当所有的线程派对都调用了这个函数时，它们都被同时释放。如果提供*超时*，则优先使用它提供给类构造函数的任何内容。

返回值是一个整数，范围从0到*party* - 1，对于每个线程都是不同的。这可以用来选择一个线程来做一些特殊的管家工作，例如：

```
i = barrier.wait()
if i == 0:
    # Only one thread needs to print this
    print("passed the barrier")
```

如果给构造函数提供了一个*动作*，其中一个线程在释放之前会调用它。如果此呼叫出现错误，屏障将进入分解状态。

如果通话超时，屏障就会进入中断状态。

BrokenBarrierError如果屏障在线程正在等待时被破坏或重置，则此方法可能会引发异常。

reset ( )

将屏障返回到默认的空状态。等待它的任何线程都会收到BrokenBarrierError异常。

请注意，如果有其他状态未知的线程，则使用此函数可能需要一些外部同步。如果障碍被破坏，那么离开它并创建一个新的可能会更好。

abort ( )

把障碍放入一个破碎的状态。这会导致任何活动的或将来的调用wait()失败BrokenBarrierError。例如，如果其中一个需要中止，以避免应用程序死锁。

最好简单地创建具有合理*超时*值的屏障 来自动防止出现错误的线程之一。

parties

穿过屏障所需的线程数。

n_waiting

当前正在屏障中等待的线程数量。

broken

一个布尔值，True如果障碍处于破碎状态。

*异常*threading. BrokenBarrierError

该对象被重置或损坏RuntimeError时，会引发 该异常（的一个子类）Barrier。

10年1月17日。在with语句中使用锁，条件和信号量

该模块提供的所有对象acquire()和 release()方法都可以用作with 语句的上下文管理器。该acquire()方法将在块被输入release()时调用，并在块退出时调用。因此，以下片段：

```
with some_lock:
    # do something...
```

相当于：

```
some_lock.acquire()
try:
    # do something...
finally:
    some_lock.release()
```

目前Lock，RLock，Condition， Semaphore，和BoundedSemaphore对象可以用作 with声明上下文管理。

17.2。multiprocessing- 基于过程的并行处理

源代码： Lib / multiprocessing /

17.2.1。简介

multiprocessing是一个使用类似于threading模块的API支持产卵过程的软件包。该multiprocessing包提供本地和远程并发性，通过使用子进程而不是线程有效地侧移 全局解释器锁。由于这个原因，该multiprocessing模块允许程序员充分利用给定机器上的多个处理器。它可以在Unix和Windows上运行。

该multiprocessing模块还引入了threading模块中没有模拟量的API 。这方面的一个主要例子是 Pool提供了一种方便的手段，即跨越多个输入值并行执行一个函数，在输入数据之间分配输入数据（数据并行性）。以下示例演示了在模块中定义这些函数的通用做法，以便子进程可以成功导入该模块。这个数据并行性的基本例子Pool ，

```
from multiprocessing import Pool

def f(x):
    return x*x

if __name__ == '__main__':
    with Pool(5) as p:
        print(p.map(f, [1, 2, 3]))
```

将打印到标准输出

```
[1, 4, 9]
```

17.2.1.1。本Process类

在multiprocessing ，进程是通过创建一个Process 对象，然后调用它的start()方法产生的。Process 遵循API的threading.Thread。一个多进程程序的简单例子是

```
from multiprocessing import Process

def f(name):
    print('hello', name)

if __name__ == '__main__':
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

为了显示所涉及的各个进程ID，下面是一个扩展示例：

```
from multiprocessing import Process
import os
```

```
def info(title):
    print(title)
    print('module name:', __name__)
    print('parent process:', os.getppid())
    print('process id:', os.getpid())

def f(name):
    info('function f')
    print('hello', name)

if __name__ == '__main__':
    info('main line')
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

有关该部件为什么必要的解释，请参阅编程指南。 if __name__ == '__main__'

17.2.1.2。上下文和启动方法

根据平台的不同，multiprocessing支持三种启动流程的方式。这些*启动方法*是

卵

父进程启动一个新的python解释器进程。子进程只会继承运行进程对象run()方法所需的资源。特别是，来自父进程的不必要的文件描述符和句柄将不会被继承。与使用*fork*或*forkserver*相比，使用此方法启动进程相当慢。

在Unix和Windows上可用。Windows上的默认值。

叉子

父进程os.fork()用来分叉Python解释器。子进程在开始时与父进程有效地相同。父进程的所有资源都由子进程继承。请注意，安全分叉多线程进程存在问题。

仅在Unix上提供。Unix上的默认值。

forkserver

当程序启动并选择*forkserver*启动方法时，服务器进程启动。从此，无论何时需要新的进程，父进程都会连接到服务器并请求它分叉一个新进程。fork服务器进程是单线程的，因此它可以安全使用os.fork()。没有不必要的资源被继承。

在支持在Unix管道上传递文件描述符的Unix平台上可用。

*在版本3.4中进行了更改：*在所有unix平台上添加了*spawn*，并为某些unix平台添加了*forkserver*。子进程不再继承Windows上的所有父代继承句柄。

在Unix上，使用*spawn*或*forkserver*启动方法也会启动一个*信号量跟踪器*进程，该进程跟踪由程序进程创建的未链接的已命名信号量。当所有进程都退出时，信号量跟踪器将取消链接任何剩余的信号量。通常应该没有，但如果一个进程被一个信号杀死了，那么可能会有一些“泄漏”的信号量。（取消链接命名的信号量是一个严重的问题，因为系统只允许有限的数量，并且在下一次重新启动之前它们不会自动断开链接。）

要选择在主模块set_start_method()的子句中使用的开始方法。例如：if __name__ == '__main__'

```
import multiprocessing as mp

def foo(q):
    q.put('hello')

if __name__ == '__main__':
    mp.set_start_method('spawn')
    q = mp.Queue()
    p = mp.Process(target=foo, args=(q,))
    p.start()
    print(q.get())
    p.join()
```

set_start_method() 不应该在程序中使用一次以上。

或者，您可以使用get_context()获取上下文对象。上下文对象与多处理模块具有相同的API，并允许在同一程序中使用多个启动方法。

```
import multiprocessing as mp

def foo(q):
    q.put('hello')

if __name__ == '__main__':
    ctx = mp.get_context('spawn')
    q = ctx.Queue()
    p = ctx.Process(target=foo, args=(q,))
    p.start()
    print(q.get())
    p.join()
```

请注意，与一个上下文相关的对象可能与另一个上下文的进程不兼容。特别是，使用*fork*上下文创建的锁不能传递到使用*spawn*或*forkserver*启动方法启动的进程 。

想要使用特定启动方法的库应该可以get_context()用来避免干扰库用户的选择。

17.2.1.3。在进程之间交换对象

multiprocessing 支持两种进程之间的通信通道：

队列

这个Queue类是一个近似的克隆queue.Queue。例如：

```
from multiprocessing import Process, Queue

def f(q):
    q.put([42, None, 'hello'])

if __name__ == '__main__':
```

```
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print(q.get())    # prints "[42, None, 'hello']"
    p.join()
```

队列是线程和进程安全的。

管道

该Pipe()函数返回一对由管道连接的连接对象，默认情况下为双工（双向）。例如：

```
from multiprocessing import Process, Pipe

def f(conn):
    conn.send([42, None, 'hello'])
    conn.close()

if __name__ == '__main__':
    parent_conn, child_conn = Pipe()
    p = Process(target=f, args=(child_conn,))
    p.start()
    print(parent_conn.recv())   # prints "[42, None, 'hello']"
    p.join()
```

通过返回的两个连接对象Pipe()表示管道的两端。每个连接对象都有send()和recv()方法（等等）。请注意，如果两个进程（或线程）试图同时读取或写入管道的*同*一端，则管道中的数据可能会损坏。当然，不会同时使用管道的不同端点进行腐蚀的风险。

17.2.1.4。进程之间的同步

multiprocessing包含来自所有同步基元的等价物threading。例如，可以使用锁来确保一次只有一个进程打印到标准输出：

```
from multiprocessing import Process, Lock

def f(l, i):
    l.acquire()
    try:
        print('hello world', i)
    finally:
        l.release()

if __name__ == '__main__':
    lock = Lock()

    for num in range(10):
        Process(target=f, args=(lock, num)).start()
```

如果不使用来自不同流程的锁定输出，则容易混淆。

17.2.1.5。在进程之间共享状态

如上所述，在进行并发编程时，通常最好尽可能避免使用共享状态。使用多个进程时尤其如此。

但是，如果您确实需要使用某些共享数据，则 multiprocessing可以提供一些方法。

共享内存

数据可以使用Value或 存储在共享内存映射中Array。例如，下面的代码

```
from multiprocessing import Process, Value, Array

def f(n, a):
    n.value = 3.1415927
    for i in range(len(a)):
        a[i] = -a[i]

if __name__ == '__main__':
    num = Value('d', 0.0)
    arr = Array('i', range(10))

    p = Process(target=f, args=(num, arr))
    p.start()
    p.join()

    print(num.value)
    print(arr[:])
```

将打印

```
3.1415927
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

在'd'与'i'创建时使用的参数num和arr被使用的那种的TypeCodes array模块：'d'表示一个双精度浮点数和'i'指示符号整数。这些共享对象将是进程和线程安全的。

为了更灵活地使用共享内存，可以使用multiprocessing.sharedctypes支持创建从共享内存分配的任意ctypes对象的 模块。

服务器进程

通过Manager()控制服务器进程返回的管理器对象，该进程包含Python对象并允许其他进程使用代理来操作它们。

通过返回的经理Manager()将支持类型 list ，dict ，Namespace ，Lock ， RLock ，Semaphore ，BoundedSemaphore ， Condition ，Event ，Barrier ， Queue ，Value和Array。例如，

```
from multiprocessing import Process, Manager
```

```
def f(d, l):
    d[1] = '1'
    d['2'] = 2
    d[0.25] = None
    l.reverse()

if __name__ == '__main__':
    with Manager() as manager:
        d = manager.dict()
        l = manager.list(range(10))

        p = Process(target=f, args=(d, l))
        p.start()
        p.join()

        print(d)
        print(l)
```

将打印

```
{0.25: None, 1: '1', '2': 2}
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

服务器进程管理器比使用共享内存对象更灵活，因为它们可以支持任意对象类型。此外，单个管理员可以通过网络在不同计算机上的进程共享。但是，它们比使用共享内存要慢。

17.2.1.6。使用　　工作人员

本Pool类代表工作进程池。它具有允许以几种不同方式将任务卸载到工作进程的方法。

例如：

```
from multiprocessing import Pool, TimeoutError
import time
import os

def f(x):
    return x*x

if __name__ == '__main__':
    # start 4 worker processes
    with Pool(processes=4) as pool:

        # print "[0, 1, 4,..., 81]"
        print(pool.map(f, range(10)))

        # print same numbers in arbitrary order
        for i in pool.imap_unordered(f, range(10)):
            print(i)

        # evaluate "f(20)" asynchronously
        res = pool.apply_async(f, (20,))      # runs in *only* one process
        print(res.get(timeout=1))             # prints "400"
```

```
        # evaluate "os.getpid()" asynchronously
        res = pool.apply_async(os.getpid, ()) # runs in *only* one process
        print(res.get(timeout=1))             # prints the PID of that process

        # launching multiple evaluations asynchronously *may* use more processes
        multiple_results = [pool.apply_async(os.getpid, ()) for i in range(4)]
        print([res.get(timeout=1) for res in multiple_results])

        # make a single worker sleep for 10 secs
        res = pool.apply_async(time.sleep, (10,))
        try:
            print(res.get(timeout=1))
        except TimeoutError:
            print("We lacked patience and got a multiprocessing.TimeoutError")

        print("For the moment, the pool remains available for more work")

    # exiting the 'with'-block has stopped the pool
    print("Now the pool is closed and no longer available")
```

请注意，池的方法只能由创建它的进程使用。

注意: 这个包中的功能要求__main__模块可以由孩子导入。编程指南 中介绍了这一点，但值得在此指出。这意味着一些示例（如multiprocessing.pool.Pool示例）在交互式解释器中不起作用。例如：

```
>>> from multiprocessing import Pool
>>> p = Pool(5)
>>> def f(x):
...     return x*x
...
>>> p.map(f, [1,2,3])
Process PoolWorker-1:
Process PoolWorker-2:
Process PoolWorker-3:
Traceback (most recent call last):
AttributeError: 'module' object has no attribute 'f'
AttributeError: 'module' object has no attribute 'f'
AttributeError: 'module' object has no attribute 'f'
```

（如果你尝试这样做，它实际上会输出三个以半随机方式交错的完整回溯，然后你可能不得不停止主处理。）

17.2.2。参考

该multiprocessing软件包主要复制threading模块的API。

17.2.2.1。Process和例外

class multiprocessing.Process (*group = None*，*target = None*，*name = None*，*args = ( )*，*kwargs = {}*，***，*daemon = None*)

过程对象表示在单独的过程中运行的活动。本 Process 类具有的所有方法等同 threading.Thread。

应始终使用关键字参数调用构造函数。*团队* 应该永远是None; 它仅仅是为了兼容而存在 threading.Thread。 *target*是要由run()方法调用的可调用对象。它默认为None ，意味着什么都不叫。*名称*是进程名称（ name详情请参阅）。 *args*是目标调用的参数元组。 *kwargs*是目标调用的关键字参数字典。如果提供，则仅关键字*守护进程*参数将进程daemon标志设置为True或False。如果None（默认），该标志将从创建过程继承。

默认情况下，没有参数传递给*目标*。

如果一个子类重写构造函数，它必须确保它Process.__init__()在进行任何其他操作之前调用基类构造函数（ ）。

*在版本3.3中进行了更改：*添加了*守护进程*参数。

run ( )

表示进程活动的方法。

您可以在子类中重写此方法。标准run() 方法调用传递给对象构造函数的可调用对象作为目标参数，如果有的话，分别从*args*和*kwargs*参数中获取顺序和关键字参数。

start ( )

开始流程的活动。

每个过程对象最多只能调用一次。它安排run()在独立的进程中调用对象的方法。

join ([*timeout*])

如果可选参数*timeout*是None（默认值），则该方法将阻塞，直到其join()方法被调用的进程终止。如果*超时*是一个正数，它最多会阻塞*超时*秒数。请注意，None如果方法终止或方法超时，该方法将返回。检查过程exitcode以确定它是否终止。

一个过程可以连接多次。

进程无法自行加入，因为这会导致死锁。尝试在启动之前加入进程是错误的。

name

该进程的名称。该名称是仅用于识别目的的字符串。它没有语义。多个进程可以被赋予相同的名称。

初始名称由构造函数设置。如果没有明确的名字被提供给构造函数，则构造表单'Process-N_1：N_2：... N_k'的名称，其中每个N_k是其父代的第N个孩子。

is_alive ( )

返回过程是否存在。

粗略地说，从该start() 方法返回的那一刻起，一个进程对象处于活动状态，直到子进程终止。

daemon

进程的守护进程标志，一个布尔值。这必须在start()调用之前设置 。

初始值是从创建过程继承的。

当进程退出时，它将尝试终止其所有守护进程的子进程。

请注意，守护进程不允许创建子进程。否则，守护进程会在其父进程退出时终止其子进程。另外，这些**不是** Unix守护进程或服务，它们是正常的进程，如果非守护进程退出，它们将被终止（而不是加入）。

除了 threading.ThreadAPI之外，Process对象还支持以下属性和方法：

pid

返回进程ID。在这个过程产生之前，这将是 None。

exitcode

孩子的退出代码。这将是None如果该过程还没有结束。负值-*N*表示孩子被信号*N*终止。

authkey

进程的身份验证密钥（一个字节字符串）。

当multiprocessing初始化主进程正在使用分配一个随机串os.urandom()。

当Process创建一个对象时，它将继承其父进程的身份验证密钥，但可以通过设置authkey为另一个字节字符串来更改它。

请参阅认证密钥。

sentinel

系统对象的数字句柄，当过程结束时它将变为“就绪”。

如果您想一次等待几个事件，则可以使用此值multiprocessing.connection.wait()。否则，调用join()更简单。

在Windows中，这是与使用的OS手柄WaitForSingleObject 和WaitForMultipleObjects家庭的API调用。在Unix上，这是一个可用于select模块基元的文件描述符。

3.3版本的新功能

terminate ( )

终止该过程。在Unix上，这是使用SIGTERM信号完成的；在Windows TerminateProcess()上使用。请注意，退出处理程序和最后的子句等将不会执行。

请注意，流程的后代流程*不会*终止 - 它们将简单地变成孤儿。

警告: 如果在关联进程正在使用管道或队列时使用此方法，那么管道或队列可能会损坏并可能会被其他进程无法使用。同样，如果进程获得了锁或信号量等，那么终

止它可能会导致其他进程死锁。

需要注意的是start()，join()，is_alive()， terminate()和exitcode方法只能由创建进程对象的过程调用。

一些方法的示例用法Process：

```
>>> import multiprocessing, time, signal
>>> p = multiprocessing.Process(target=time.sleep, args=(1000,))
>>> print(p, p.is_alive())
<Process(Process-1, initial)> False
>>> p.start()
>>> print(p, p.is_alive())
<Process(Process-1, started)> True
>>> p.terminate()
>>> time.sleep(0.1)
>>> print(p, p.is_alive())
<Process(Process-1, stopped[SIGTERM])> False
>>> p.exitcode == -signal.SIGTERM
True
```

*异常*multiprocessing.ProcessError

所有multiprocessing例外的基类。

*异常*multiprocessing.BufferTooShort

Connection.recv_bytes_into()当提供的缓冲区对象太小而不能读取消息时引发异常。

如果e是的一个实例BufferTooShort，然后e.args[0]会给出消息作为字节串。

*异常*multiprocessing.AuthenticationError

出现身份验证错误时引发。

*异常*multiprocessing.TimeoutError

当超时过期时由超时方法提出。

17.2.2.2。管道和队列

当使用多个进程时，通常使用消息传递进行进程之间的通信，并避免使用任何同步原语（如锁）。

对于传递消息，可以使用Pipe()（用于两个进程之间的连接）或队列（允许多个生产者和消费者）。

的Queue，SimpleQueue和JoinableQueue类型是多生产，多消费FIFO仿照队列queue.Queue标准库类。他们的区别在于Queue缺乏 task_done()和join()引入的Python 2.5的方法queue.Queue类。

如果使用，JoinableQueue则**必须**调用 JoinableQueue.task_done()从队列中移除的每个任务，否则用于计算未完成任务数的信号量最终可能会溢出，引发异常。

请注意，您也可以使用管理员对象创建共享队列 - 请参阅 经理。

注意： multiprocessing使用常规queue.Empty和 queue.Full例外来表示超时。它们在multiprocessing命名空间中不可用，因此您需要从中导入它们 queue。

注意： 当一个对象被放在一个队列中时，该对象被腌制，后面的线程将清理后的数据刷新到底层管道。这有一些令人惊讶的后果，但不应该导致任何实际困难 - 如果他们真的打扰你，那么你可以使用由经理创建的队列 。

1. 在一个空队列上放置一个对象之后，在队列的empty() 方法返回之前可能会有一个无限小的延迟，False并且get_nowait()可以在不提升的情况下返回queue.Empty。
2. 如果多个进程正在排列对象，则可能会在另一端无序地接收对象。然而，由相同过程入队的对象将始终按预期顺序相对于彼此。

警告： 如果一个进程正在使用打死Process.terminate()或os.kill() 当它试图使用Queue，则在队列中的数据可能会损坏。这可能会导致任何其他进程在尝试稍后使用队列时发生异常。

警告： 如上所述，如果子进程已将项目放入队列（并且未使用JoinableQueue.cancel_join_thread），则该进程将不会终止，直到所有缓冲项目已被刷新到管道。

这意味着如果你尝试加入这个过程，你可能会遇到死锁，除非你确定放入队列的所有物品都已被消耗掉。同样，如果子进程是非守护进程，那么父进程在尝试加入所有非守护进程子进程时可能会在退出时挂起。

请注意，使用管理器创建的队列不存在此问题。请参阅 编程准则。

有关使用队列进行进程间通信的示例，请参阅 示例。

multiprocessing.**Pipe**([*双工*])

返回一对的 代表的配管的端部的对象。(conn1, conn2) Connection

如果*双工*是True（默认），那么管道是双向的。如果 *双工*是False管道，那么管道是单向的：conn1只能用于接收消息，conn2并且只能用于发送消息。

class multiprocessing.**Queue**([*maxsize*])

返回使用管道和几个锁/信号量实现的进程共享队列。当一个进程首先将一个项目放入队列中时，启动一个进给线程，该进程线程将对象从缓冲区传送到管道中。

标准库模块的常见queue.Empty和queue.Full例外情况queue引发超时。

Queue实现queue.Queue除了task_done()和之外的 所有方法join()。

qsize()

返回队列的近似大小。由于多线程/多处理语义，这个数字是不可靠的。

请注意，NotImplementedError在Mac OS X等Unix平台上可能会出现这种情况，sem_getvalue()但未实现。

empty ( )

True如果队列为空False则返回，否则返回。由于多线程/多处理语义，这是不可靠的。

full ( )

True如果队列已满False则返回，否则返回。由于多线程/多处理语义，这是不可靠的。

put (*obj* [, *block* [, *timeout*]])

将obj放入队列中。如果可选参数*块*是True （默认）并且*超时*是None（默认），则在需要时禁止，直到有空闲插槽可用。如果*超时*是一个正数，它会阻止至多*超时*秒数，并queue.Full在该时间内没有空闲插槽时引发异常。否则（*块*是 False），如果空闲插槽立即可用，则在队列中放置一个项目，否则引发queue.Full异常（在这种情况下*超时*被忽略）。

put_nowait (*obj*)

相当于。put(obj, False)

get ([*block* [, *timeout*]])

从队列中移除并返回一个项目。如果可选的参数*块*为 True默认值，并且*超时*为None默认值，则在必要时阻塞，直到项目可用。如果*超时时间*为正数，则最多会阻止*超时*秒数， queue.Empty 如果在该时间内没有可用项目，则会引发异常。否则（块是 False），如果一个项立即可用，则返回一个项目，否则引发 queue.Empty异常（在这种情况下*超时*被忽略）。

get_nowait ( )

相当于get(False)。

multiprocessing.Queue有一些其他方法未找到 queue.Queue。大多数代码通常不需要这些方法：

close ( )

指示当前进程不会有更多数据放入此队列中。一旦将所有缓冲数据刷新到管道，后台线程将退出。这在队列被垃圾收集时自动调用。

join_thread ( )

加入后台线程。这只能在close()被调用后才能使用。它会阻塞，直到后台线程退出，确保缓冲区中的所有数据已被刷新到管道。

默认情况下，如果进程不是队列的创建者，那么在退出时它将尝试加入队列的后台线程。该过程可以调用 cancel_join_thread()做出join_thread()什么也不做。

cancel_join_thread ( )

防止join_thread()阻塞。特别是，这可以防止后台线程在进程退出时自动加入 - 请参阅join_thread()。

这个方法的名字可能更好 allow_exit_without_flush()。这很可能会导致入队数据丢失，并且几乎可以肯定不需要使用它。如果您需要当前进程立即退出而不等待将排入

队列的数据刷新到底层管道，并且您不关心丢失的数据，那才真正在那里。

> **注意：** 该类的功能需要在主机操作系统上运行共享信号量。没有一个，这个类中的功能将被禁用，并尝试实例化一个Queue将导致一个ImportError。有关更多信息，请参阅bpo-3770。对于下面列出的任何专用队列类型也是如此。

*类*multiprocessing.SimpleQueue

这是一个简化的Queue类型，非常接近锁定Pipe。

empty ( )

True如果队列为空False则返回，否则返回。

get ( )

从队列中移除并返回一个项目。

put (*item*)

将*项目*放入队列中。

class multiprocessing.JoinableQueue ([*maxsize*])

JoinableQueue，一个Queue子类，是另外有一个队列task_done()和join()方法。

task_done ( )

表明以前排队的任务已完成。由队列使用者使用。对于每个get()用于获取任务的对象，后续调用会task_done()告知队列该任务的处理已完成。

如果a join()当前处于阻塞状态，则在所有项目都处理完毕后（即task_done()接收到已put()进入队列的每个项目的呼叫），它将恢复。

提出了一个ValueError好象叫更多的时间比中放入队列中的项目。

join ( )

阻塞，直到队列中的所有项目都被获取并处理。

每当将项目添加到队列中时，未完成任务的数量就会增加。每当消费者打电话task_done()表明该物品已被检索并且所有工作都已完成时，计数就会减少。当未完成任务的计数降至零时， join()取消阻止。

17.2.2.3。杂项

multiprocessing.active_children ( )

返回当前进程的所有活着的孩子的列表。

调用它会产生“加入”已经完成的任何过程的副作用。

multiprocessing.cpu_count ( )

返回系统中的CPU数量。

这个数字不等于当前进程可以使用的CPU数量。可用的CPU数量可以通过len(os.sched_getaffinity(0))

可能会提高NotImplementedError。

也可以看看: os.cpu_count()

multiprocessing.**current_process**()

返回Process当前进程对应的对象。

一种类似物threading.current_thread()。

multiprocessing.**freeze_support**()

添加对何时使用的程序multiprocessing已被冻结以生成Windows可执行文件的支持。（已经用**py2exe**，**PyInstaller**和**cx_Freeze**进行过测试。）

需要在主模块的行后面直接调用该函数。例如：if __name__ == '__main__'

```
from multiprocessing import Process, freeze_support

def f():
    print('hello world!')

if __name__ == '__main__':
    freeze_support()
    Process(target=f).start()
```

如果该freeze_support()行被忽略，则尝试运行冻结的可执行文件将会引发RuntimeError。

freeze_support()在Windows以外的任何操作系统上调用时，调用都不起作用。另外，如果模块正在通过Windows上的Python解释器正常运行（该程序尚未被冻结），则freeze_support()不起作用。

multiprocessing.**get_all_start_methods**()

返回支持的启动方法列表，其中第一个是默认方法。可能的启动方法是'fork'，'spawn'并且'forkserver'。仅在Windows 'spawn'上可用。在Unix上'fork'并且'spawn'始终受支持，并且'fork'是默认设置。

3.4版新增功能

multiprocessing.**get_context**(*method = None*)

返回与multiprocessing模块具有相同属性的上下文对象。

如果*方法*是，None则返回默认上下文。否则，*方法*应该是'fork'，'spawn'，'forkserver'。ValueError如果指定的启动方法不可用，则会引发此问题。

3.4版新增功能

multiprocessing.**get_start_method**(*allow_none = False*)

返回用于启动进程的启动方法的名称。

如果start方法没有被修复，并且*allow_none*为false，那么start方法被固定为默认值并返回名称。如果start方法没有被修复，并且*allow_none* 为true，则None返回。

返回值可以是'fork'，'spawn'，'forkserver'或None。 'fork'在Unix上是默认的，而'spawn'在Windows上是默认的。

3.4版新增功能

multiprocessing.**set_executable**( )

设置启动子进程时使用的Python解释器的路径。（默认sys.executable使用）。嵌入者可能需要做一些类似的事情

```
set_executable(os.path.join(sys.exec_prefix, 'pythonw.exe'))
```

然后才能创建子进程。

*在版本3.4中进行了更改：*现在，在使用'spawn'启动方法的Unix上受支持。

multiprocessing.**set_start_method**(*方法*)

设置应该用来启动子进程的方法。 *方法*可以'fork'，'spawn'或者'forkserver'。

请注意，这应该最多调用一次，并且它应该在主模块的子句内受到保护。if __name__ == '__main__'

3.4版新增功能

注意： multiprocessing不包含的类似物 threading.active_count()，threading.enumerate()， threading.settrace()，threading.setprofile()，threading.Timer，或threading.local。

17.2.2.4。连接对象

连接对象允许发送和接收可选对象或字符串。他们可以被认为是面向消息的连接套接字。

连接对象通常使用 Pipe- 另请参阅 监听器和客户端。

*类*multiprocessing.connection.**Connection**

send(*obj*)

将对象发送到应该使用的连接的另一端recv()。

该对象必须是可挑选的。非常大的泡菜（大约32 MB +，但取决于操作系统）可能会引发ValueError异常。

recv( )

返回使用连接的另一端发送的对象 send()。阻止，直到有东西可以接收。EOFError如果没有什么可以接收而另一端关闭，就会引发 。

fileno ( )

返回连接使用的文件描述符或句柄。

close ( )

关闭连接。

当连接被垃圾收集时这被自动调用。

poll ([*timeout*])

返回是否有可供读取的数据。

如果没有指定*超时*，它将立即返回。如果 *超时*是一个数字，那么这指定了要阻止的最大时间（以秒为单位）。如果*超时*，None则使用无限超时。

请注意，多个连接对象可以通过使用轮询一次multiprocessing.connection.wait()。

send_bytes (*buffer* [, *offset* [, *size*]])

从类似字节的对象发送字节数据作为完整的消息。

如果给出*偏移量*，则数据从*缓冲区中的*该位置读取。如果 给出*大小*，那么将从缓冲区读取那么多字节。非常大的缓冲区（大约32 MB +，但取决于操作系统）可能会引发ValueError异常

recv_bytes ([*maxlength*])

将连接另一端发送的字节数据的完整消息作为字符串返回。阻止，直到有东西可以接收。提出EOFError，如果没有什么留下来接收和另一端已经关闭。

如果*最大长度*被指定并且所述消息是长于*最大长度*然后OSError升至并连接将不再是可读的。

*在版本3.3中更改：*此函数用于引发IOError，现在它是一个别名OSError。

recv_bytes_into (*buffer* [, *offset*])

将从连接另一端发送的完整字节数据消息读入*缓冲区*，并返回消息中的字节数。阻止，直到有东西可以接收。EOFError如果没有什么可以接收而另一端关闭，就会引发。

*缓冲区*必须是可写的类似字节的对象。如果 给出*偏移量*，那么消息将从该位置写入缓冲区。偏移量必须是小于*缓冲区*长度的非负整数（以字节为单位）。

如果缓冲区太短，则会BufferTooShort引发异常，并且完整的消息可用，e.args[0]因为e 异常实例在哪里。

*在版本3.3中更改：*现在可以使用Connection.send()和在进程之间传输连接对象本身Connection.recv()。

版本3.3中的新功能： Connection对象现在支持上下文管理协议 - 请参阅 上下文管理器类型。 __enter__()返回连接对象，并__exit__()调用close()。

例如：

```
>>> from multiprocessing import Pipe
>>> a, b = Pipe()
>>> a.send([1, 'hello', None])
>>> b.recv()
[1, 'hello', None]
>>> b.send_bytes(b'thank you')
>>> a.recv_bytes()
b'thank you'
>>> import array
>>> arr1 = array.array('i', range(5))
>>> arr2 = array.array('i', [0] * 10)
>>> a.send_bytes(arr1)
>>> count = b.recv_bytes_into(arr2)
>>> assert count == len(arr1) * arr1.itemsize
>>> arr2
array('i', [0, 1, 2, 3, 4, 0, 0, 0, 0, 0])
```

警告: 该Connection.recv()方法会自动取消接收的数据，这可能会带来安全风险，除非您可以信任发送该消息的进程。

因此，除非连接对象是使用Pipe()您制作的，否则只应 在执行某种验证之后使用recv()和send()方法。请参阅 认证密钥。

警告: 如果某个进程在尝试读取或写入管道时被终止，那么管道中的数据可能会被破坏，因为可能无法确定消息边界位于何处。

17.2.2.5。同步原语

通常，同步原语在多进程程序中并不像在多线程程序中那样必要。请参阅threading模块的文档。

请注意，也可以使用管理器对象创建同步基元 - 请参阅管理器。

class multiprocessing.Barrier (*parties* [, *action* [, *timeout*]])

屏障对象：一个克隆threading.Barrier。

3.3版本的新功能

class multiprocessing.BoundedSemaphore ([*value*])

一个有界的信号量对象：一个紧密的类比 threading.BoundedSemaphore。

与其相近的模拟存在单一差异：其acquire方法的第一个参数被命名为*block*，与之一致Lock.acquire()。

注意: 在Mac OS X上，这是无法区分的，Semaphore因为 sem_getvalue()没有在该平台上实现。

class multiprocessing.Condition ([*lock*])

一个条件变量：一个别名threading.Condition。

如果指定了*锁*，那么它应该是一个Lock或一个RLock 对象multiprocessing。

*在版本3.3中更改：*该wait_for()方法已添加。

*类*multiprocessing.Event

克隆的threading.Event。

*类*multiprocessing.Lock

非递归锁定对象：一个非常类似的threading.Lock。一旦进程或线程获得锁定，随后尝试从任何进程或线程获取它将被阻塞，直到它被释放; 任何进程或线程都可以释放它。threading.Lock适用于线程的概念和行为 在multiprocessing.Lock适用于进程或线程时在此处被复制 ，除非另有说明。

请注意，Lock它实际上是一个工厂函数，它返回multiprocessing.synchronize.Lock使用默认上下文初始化的实例。

Lock支持上下文管理器协议，因此可以在with语句中使用。

acquire (*block = True*，*timeout = None*)

获取锁定，阻止或不阻止。

将*block*参数设置为True（默认值）时，方法调用将被阻塞，直到锁处于解锁状态，然后将其设置为锁定并返回True。请注意，这第一个参数的名称不同于threading.Lock.acquire()。

将*块*参数设置为False，方法调用不会阻止。如果锁当前处于锁定状态，则返回False; 否则将锁设置为锁定状态并返回True。

当用正值，浮点值*超时*调用*时*，只要无法获取锁，就会阻塞*超时*指定的秒数。具有负值的调用*超时*值 相当于*超时*值为零。*超时*值为None（缺省值）的调用将 *超时时间*设置为无限。请注意，处理负值或*超时*None值 与实施的行为不同 。该*超时*参数有，如果没有实际意义*块*参数被设置为，并因此忽略。返回threading.Lock.acquire()FalseTrue如果已获取锁定或False超时时间已过。

release ( )

释放一个锁。这可以从任何进程或线程调用，不仅是最初获取锁的进程或线程。

threading.Lock.release()除了在解锁的锁上调用a时，行为与其中的行为相同ValueError。

*类*multiprocessing.RLock

递归锁定对象：一个紧密的模拟对象threading.RLock。递归锁必须由获取它的进程或线程释放。一旦进程或线程获得了递归锁定，相同的进程或线程可能会再次获取而不会阻塞; 该进程或线程必须每次释放一次它被获取。

请注意，RLock它实际上是一个工厂函数，它返回multiprocessing.synchronize.RLock使用默认上下文初始化的实例。

RLock支持上下文管理器协议，因此可以在with语句中使用。

acquire (*block = True*，*timeout = None*)

获取锁定，阻止或不阻止。

在*block*参数设置为的情况下调用时True，阻塞直到锁处于解锁状态（不属于任何进程或线程），除非该锁已被当前进程或线程所拥有。当前进程或线程接着获取锁的所有权（如果它还没有所有权），并且锁内的递归级别增加1，导致返回值为True。请注意，与实现相比，第一个参数的行为存在一些差异threading.RLock.acquire()，从参数本身的名称开始。

当调用*block*参数设置为时False，不要阻塞。如果锁已被另一个进程或线程获取（并因此被拥有），则当前进程或线程不会获得所有权，并且锁内的递归级别不会更改，导致返回值为False。如果锁处于未锁定状态，当前进程或线程将获得所有权，并递增递归级别，从而返回值为True。

*timeout*参数的使用和行为与in中的相同 Lock.acquire()。请注意，某些*超时*行为 与实施的行为有所不同threading.RLock.acquire()。

release ( )

释放一个锁，递减递归级别。如果在递减之后递归级别为零，则将锁重置为解锁（不属于任何进程或线程），并且如果有任何其他进程或线程被阻塞，等待锁解锁，则准许其中一个进程继续。如果在递减之后递归级别仍然不为零，则锁保持锁定并由调用进程或线程拥有。

只有在调用进程或线程拥有锁定时才调用此方法。一个AssertionError如果该方法是通过一个过程调用或线程以外的雇主或升高如果锁处于解锁（无主）的状态。请注意，在这种情况下引发的异常类型与实现的行为不同threading.RLock.release()。

class multiprocessing. Semaphore ([*value*])

信号量对象：一个近似的类比threading.Semaphore。

与其相近的模拟存在单一差异：其acquire方法的第一个参数被命名为*block*，与之一致 Lock.acquire()。

注意： 在Mac OS X上，sem_timedwait不受支持，因此acquire()使用超时调用将使用休眠循环模拟该函数的行为。

注意： 如果所产生的SIGINT信号Ctrl-C，而主线程是由呼叫阻塞到达 BoundedSemaphore.acquire()，Lock.acquire()，RLock.acquire()，Semaphore.acquire()，Condition.acquire() 或Condition.wait()则该呼叫将被立即中断和 KeyboardInterrupt将提高。

这与threading在等效阻塞调用进行时忽略SIGINT 的行为不同。

注意： 某些此软件包的功能需要在主机操作系统上运行共享信号量。没有一个，multiprocessing.synchronize模块将被禁用，并尝试导入它将导致一个ImportError。有关更多信息，请参阅 bpo-3770。

17.2.2.6。共享ctypes对象

可以使用可以由子进程继承的共享内存创建共享对象。

multiprocessing.Value（*typecode_or_type*，**args*，*lock = True*）

返回ctypes从共享内存分配的对象。默认情况下，返回值实际上是对象的同步包装器。对象本身可以通过a的*值*属性来访问Value。

*typecode_or_type*确定返回对象的类型：它是一个ctypes类型或array 模块使用的类型的一个字符类型。 **参数*传递给类型的构造函数。

如果*锁*是True（默认值），那么一个新的递归锁对象被创建同步访问值。如果*锁*是一个Lock或一个RLock对象，那么将用于同步对该值的访问。如果是*锁*，False那么访问返回的对象不会被锁自动保护，因此它不一定是“过程安全的”。

操作就像+=它涉及的读取和写入不是原子。因此，例如，如果你想以原子方式递增共享值，那么仅仅是做不到

```
counter.value += 1
```

假设关联的锁是递归的（它是默认的），你可以改为这样做

```
with counter.get_lock():
    counter.value += 1
```

请注意，*锁定*是仅有关键字的参数。

multiprocessing.Array（*typecode_or_type*，*size_or_initializer*，***，*lock = True*）

返回从共享内存分配的ctypes数组。默认情况下，返回值实际上是数组的同步包装器。

*typecode_or_type*确定返回数组元素的类型：它是模块使用的ctypes类型或单字符typecode array。如果*size_or_initializer*是一个整数，那么它决定了数组的长度，并且该数组最初将归零。否则，*size_or_initializer*是用于初始化数组的序列，其长度决定数组的长度。

如果*锁*是True（默认值），那么一个新的锁定对象被创建同步访问值。如果*锁*是一个Lock或 一个RLock对象，那么将用于同步对该值的访问。如果是*锁*，False那么访问返回的对象不会被锁自动保护，因此它不一定是“过程安全的”。

请注意，*锁定*是仅有关键字的参数。

请注意，一个ctypes.c_char具有*值*和*原始* 属性的数组允许用它来存储和检索字符串。

17.2.2.6.1。该multiprocessing.sharedctypes模块

该multiprocessing.sharedctypes模块提供了ctypes从共享内存中分配可由子进程继承的对象的功能 。

注意： 虽然可以将指针存储在共享内存中，但请记住这将指向特定进程的地址空间中的某个位置。但是，指针在第二个进程的上下文中很可能无效，并且试图从第二个进程取消引用该指针可能会导致崩溃。

multiprocessing.sharedctypes.RawArray（*typecode_or_type*，*size_or_initializer*）

返回从共享内存分配的ctypes数组。

*typecode_or_type*确定返回数组元素的类型：它是模块使用的ctypes类型或单字符typecode `array`。如果*size_or_initializer*是一个整数，那么它确定数组的长度，并且该数组将被初始化为零。否则，*size_or_initializer*是一个用于初始化数组的序列，其长度决定数组的长度。

请注意，设置和获取元素可能是非原子的 - 请使用它 `Array()` 来确保使用锁自动同步访问。

multiprocessing.sharedctypes.RawValue（*typecode_or_type*，**args*）

返回从共享内存分配的ctypes对象。

*typecode_or_type*确定返回对象的类型：它是一个ctypes类型或`array` 模块使用的类型的一个字符类型。 **参数*传递给类型的构造函数。

请注意，设置和获取该值可能是非原子的 - 使用它 `Value()` 来确保访问使用锁自动同步。

请注意，`ctypes.c_char`has `value`和`raw` 属性允许用它来存储和检索字符串 - 请参阅文档`ctypes`。

multiprocessing.sharedctypes.Array（*typecode_or_type*，*size_or_initializer*，*，*lock* = *True*）

相同`RawArray()`，除了取决于的值*锁*的过程安全的同步封装器可以返回来代替原始ctypes的阵列。

如果*锁*是`True`（默认值），那么一个新的锁定对象被创建同步访问值。如果*锁*是一个`Lock`或一个 `RLock`对象，那么将用于同步对该值的访问。如果是*锁*，`False`那么访问返回的对象不会被锁自动保护，因此它不一定是“过程安全的”。

请注意，*锁定*是仅有关键字的参数。

multiprocessing.sharedctypes.Value（*typecode_or_type*，**args*，*lock* = *True*）

相同`RawValue()`，除了取决于的值*锁*的过程安全的同步封装器可以返回来代替原始ctypes的对象。

如果*锁*是`True`（默认值），那么一个新的锁定对象被创建同步访问值。如果*锁*是一个`Lock`或 一个`RLock`对象，那么将用于同步对该值的访问。如果是*锁*，`False`那么访问返回的对象不会被锁自动保护，因此它不一定是“过程安全的”。

请注意，*锁定*是仅有关键字的参数。

multiprocessing.sharedctypes.copy（*obj*）

返回从ctypes对象*obj*的副本共享内存中分配的ctypes对象。

multiprocessing.sharedctypes.synchronized(*obj*[, *lock*])

为使用*锁*来同步访问的ctypes对象返回一个流程安全的包装器对象。如果*锁*是None（缺省值），然后一个multiprocessing.RLock对象被自动创建。

一个同步包装除了包装它的对象之外还有两个方法：get_obj()返回包装对象并get_lock()返回用于同步的锁对象。

请注意，通过包装器访问ctypes对象可能比访问原始ctypes对象要慢很多。

*在版本3.5中更改：*同步对象支持上下文管理器协议。

下表比较了用于从共享内存创建共享ctypes对象的语法与正常ctypes语法。（在表中MyStruct是。的一些子类ctypes.Structure。）

ctypes的	使用类型共享类型	使用typecode共享类型
c_double(2.4)	RawValue(c_double, 2.4)	RawValue('d', 2.4)
MyStruct(4, 6)	RawValue(MyStruct, 4, 6)	
(c_short * 7)()	RawArray(c_short, 7)	RawArray('h', 7)
(c_int * 3)(9,2,8)	RawArray(c_int, (9, 2,	RawArray('i', (9, 2, 8))

下面是一个例子，其中一些ctypes对象被一个子进程修改：

```
from multiprocessing import Process, Lock
from multiprocessing.sharedctypes import Value, Array
from ctypes import Structure, c_double

class Point(Structure):
    _fields_ = [('x', c_double), ('y', c_double)]

def modify(n, x, s, A):
    n.value **= 2
    x.value **= 2
    s.value = s.value.upper()
    for a in A:
        a.x **= 2
        a.y **= 2

if __name__ == '__main__':
    lock = Lock()

    n = Value('i', 7)
    x = Value(c_double, 1.0/3.0, lock=False)
    s = Array('c', b'hello world', lock=lock)
    A = Array(Point, [(1.875,-6.25), (-5.75,2.0), (2.375,9.5)], lock=lock)

    p = Process(target=modify, args=(n, x, s, A))
    p.start()
    p.join()

    print(n.value)
    print(x.value)
    print(s.value)
    print([(a.x, a.y) for a in A])
```

打印的结果是

```
49
0.1111111111111111
HELLO WORLD
[(3.515625, 39.0625), (33.0625, 4.0), (5.640625, 90.25)]
```

17.2.2.7。经理

经理提供了一种方法来创建可以在不同进程之间共享的数据，包括在不同机器上运行的进程之间通过网络进行共享。管理员对象控制管理*共享对象*的服务器进程 。其他进程可以通过使用代理来访问共享对象。

multiprocessing.Manager ( )

返回SyncManager可用于在进程之间共享对象的已启动对象。返回的管理器对象对应于衍生的子进程，并具有创建共享对象并返回相应代理的方法。

当垃圾收集或其父进程退出时，Manager进程将立即关闭。管理员类在multiprocessing.managers模块中定义 ：

class multiprocessing.managers.BaseManager ([*address* [, *authkey*]])

创建一个BaseManager对象。

一旦创建，应该调用start()或get_server().serve_forever()确保管理器对象引用已启动的管理器进程。

*地址*是管理进程侦听新连接的地址。如果*地址*是None那么任意一个被选择。

*authkey*是将用于检查到服务器进程的传入连接的有效性的身份验证密钥。如果 *AUTHKEY* 是None再current_process().authkey被使用。否则使用*authkey* ，它必须是一个字节字符串。

start ([*initializer* [, *initargs*]])

启动一个子流程来启动管理器。如果*初始化*程序不是 ，None 那么子进程将initializer(*initargs)在启动时调用。

get_server ( )

返回一个Server代表管理器控制下的实际服务器的对象。该Server对象支持该serve_forever()方法：

```
>>> from multiprocessing.managers import BaseManager
>>> manager = BaseManager(address=('', 50000), authkey=b'abc')
>>> server = manager.get_server()
>>> server.serve_forever()
```

Server另外还有一个address属性。

connect ( )

将本地管理器对象连接到远程管理器进程：

```
>>> from multiprocessing.managers import BaseManager
>>> m = BaseManager(address=('127.0.0.1', 5000), authkey=b'abc')
>>> m.connect()
```

shutdown ( )

停止经理使用的过程。这仅start()在用于启动服务器进程时才可用 。

这可以被多次调用。

register (*typeid的*[, *可调用*[, *的proxyType* [, *露出*[, *method_to_typeid* [, *create_method*]]]]])

可用于注册类型或可与经理类一起调用的类方法。

*typeid*是一个“类型标识符”，用于标识特定类型的共享对象。这必须是一个字符串。

*callable*是可调用的，用于为此类型标识创建对象。如果一个管理器实例将使用该connect()方法连接到服务器，或者如果 *create_method*参数是False可以留下的话None。

*proxytype*是其中的一个子类，BaseProxy用于使用此*typeid*为共享对象创建代理。如果None那么代理类是自动创建的。

*exposed*用于指定一个方法名称序列，该类型的代理应该被允许使用访问BaseProxy._callmethod()。（如果*暴露*在None随后 proxytype._exposed_被用于代替如果它存在）。在其中没有指定暴露列表中的情况下，共享对象的所有“公共方法”将是可访问的。（这里的“公共方法”是指任何具有__call__()方法且名称不以其开头的属性’_’。）

*method_to_typeid*是一个映射，用于指定那些应该返回代理的公开方法的返回类型。它将方法名称映射到typeid字符串。（如果*method_to_typeid*是None然后proxytype._method_to_typeid_，如果它存在来代替。）如果一个方法的名字不在此映射的关键或如果映射是None则该方法返回该对象将在值被复制。

*create_method*确定是否应该使用名称*typeid*创建一个方法，该方法可用于通知服务器进程创建新的共享对象并为其返回代理。默认情况下是True。

BaseManager 实例也有一个只读属性：

address

经理使用的地址。

*版本3.3中更改：*管理器对象支持上下文管理协议 - 请参阅 上下文管理器类型。__enter__()启动服务器进程（如果尚未启动），然后返回管理器对象。 __exit__()来电shutdown()。

在以前的版本__enter__()中，如果它尚未启动，则不会启动管理器的服务器进程。

*类*multiprocessing.managers.SyncManager

其中的一个子类BaseManager可用于进程的同步。这种类型的对象被返回multiprocessing.Manager()。

它的方法创建并返回一些常用数据类型的代理对象，以跨进程同步。这主要包括共享列表和字典。

Barrier (*parties* [, *action* [, *timeout*]])
创建一个共享threading.Barrier对象并为其返回一个代理。

3.3版本的新功能

BoundedSemaphore ([*value*])
创建一个共享threading.BoundedSemaphore对象并为其返回一个代理。

Condition ([*lock*])
创建一个共享threading.Condition对象并为其返回一个代理。

如果提供了*锁*，那么它应该是一个threading.Lock或一个threading.RLock对象的代理。

*在版本3.3中更改：*该wait_for()方法已添加。

Event ( )
创建一个共享threading.Event对象并为其返回一个代理。

Lock ( )
创建一个共享threading.Lock对象并为其返回一个代理。

Namespace ( )
创建一个共享Namespace对象并为其返回一个代理。

Queue ([*maxsize*])
创建一个共享queue.Queue对象并为其返回一个代理。

RLock ( )
创建一个共享threading.RLock对象并为其返回一个代理。

Semaphore ([*value*])
创建一个共享threading.Semaphore对象并为其返回一个代理。

Array (*typecode* , *sequence*)
创建一个数组并为其返回一个代理。

Value (*typecode* , *value*)
创建一个具有可写value属性的对象并返回它的代理。

dict ( )
dict (*映射*)
dict (*序列*)
创建一个共享dict对象并为其返回一个代理。

list ( )

list(*序列*)

创建一个共享list对象并为其返回一个代理。

*在版本3.6中更改：*共享对象能够嵌套。例如，共享容器对象（如共享列表）可以包含其他共享对象，这些共享对象将全部由该对象管理和同步SyncManager。

*类*multiprocessing.managers.Namespace

可以注册的类型SyncManager。

名称空间对象没有公共方法，但具有可写的属性。它的表示显示了它的属性值。

但是，当为名称空间对象使用代理时，以下列属性开头的属性'_'将成为代理的属性，而不是所指对象的属性：

```
>>> manager = multiprocessing.Manager()
>>> Global = manager.Namespace()
>>> Global.x = 10
>>> Global.y = 'hello'
>>> Global._z = 12.3    # this is an attribute of the proxy
>>> print(Global)
Namespace(x=10, y='hello')
```

17.2.2.7.1。自定义管理器

要创建自己的经理，需要创建一个子类BaseManager并使用register()该类方法向经理类注册新类型或可调用对象。例如：

```
from multiprocessing.managers import BaseManager

class MathsClass:
    def add(self, x, y):
        return x + y
    def mul(self, x, y):
        return x * y

class MyManager(BaseManager):
    pass

MyManager.register('Maths', MathsClass)

if __name__ == '__main__':
    with MyManager() as manager:
        maths = manager.Maths()
        print(maths.add(4, 3))         # prints 7
        print(maths.mul(7, 8))         # prints 56
```

17.2.2.7.2。使用远程管理器

可以在一台机器上运行管理服务器，并让客户机从其他机器上使用它（假设所涉及的防火墙允许）。

运行以下命令将为远程客户端可以访问的单个共享队列创建一个服务器：

```
>>> from multiprocessing.managers import BaseManager
>>> from queue import Queue
>>> queue = Queue()
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue', callable=lambda:queue)
>>> m = QueueManager(address=('', 50000), authkey=b'abracadabra')
>>> s = m.get_server()
>>> s.serve_forever()
```

一个客户端可以如下访问服务器：

```
>>> from multiprocessing.managers import BaseManager
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue')
>>> m = QueueManager(address=('foo.bar.org', 50000), authkey=b'abracadabra')
>>> m.connect()
>>> queue = m.get_queue()
>>> queue.put('hello')
```

另一个客户端也可以使用它：

```
>>> from multiprocessing.managers import BaseManager
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue')
>>> m = QueueManager(address=('foo.bar.org', 50000), authkey=b'abracadabra')
>>> m.connect()
>>> queue = m.get_queue()
>>> queue.get()
'hello'
```

本地进程也可以访问该队列，使用客户端上面的代码远程访问它：

```
>>> from multiprocessing import Process, Queue
>>> from multiprocessing.managers import BaseManager
>>> class Worker(Process):
...     def __init__(self, q):
...         self.q = q
...         super(Worker, self).__init__()
...     def run(self):
...         self.q.put('local hello')
...
>>> queue = Queue()
>>> w = Worker(queue)
>>> w.start()
>>> class QueueManager(BaseManager): pass
...
>>> QueueManager.register('get_queue', callable=lambda: queue)
>>> m = QueueManager(address=('', 50000), authkey=b'abracadabra')
>>> s = m.get_server()
>>> s.serve_forever()
```

17.2.2.8。代理对象

代理是一个对象，它*指的*是一个共享对象，它可能存在于不同的进程中。共享对象被认为是代理的*对象*。多个代理对象可能具有相同的对象。

代理对象具有调用其指示对象的相应方法的方法（尽管代理中并不需要指示对象的每种方法）。通过这种方式，代理可以像使用对象一样使用：

```
>>> from multiprocessing import Manager
>>> manager = Manager()
>>> l = manager.list([i*i for i in range(10)])
>>> print(l)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> print(repr(l))
<ListProxy object, typeid 'list' at 0x...>
>>> l[4]
16
>>> l[2:5]
[4, 9, 16]
```

注意，应用于`str()`代理将返回指示对象的表示，而应用`repr()`将返回代理的表示。

代理对象的一个重要特性是它们是可选择的，因此它们可以在进程之间传递。因此，指称对象可以包含 代理对象。这允许嵌套这些托管列表，字典和其他代理对象：

```
>>> a = manager.list()
>>> b = manager.list()
>>> a.append(b)         # referent of a now contains referent of b
>>> print(a, b)
[<ListProxy object, typeid 'list' at ...>] []
>>> b.append('hello')
>>> print(a[0], b)
['hello'] ['hello']
```

类似地，字典和列表代理可以嵌套在一起：

```
>>> l_outer = manager.list([ manager.dict() for i in range(2) ])
>>> d_first_inner = l_outer[0]
>>> d_first_inner['a'] = 1
>>> d_first_inner['b'] = 2
>>> l_outer[1]['c'] = 3
>>> l_outer[1]['z'] = 26
>>> print(l_outer[0])
{'a': 1, 'b': 2}
>>> print(l_outer[1])
{'c': 3, 'z': 26}
```

如果标准（非代理）`list`或`dict`对象包含在指示对象中，那些对这些可变值的修改将不会通过管理器传播，因为代理无法知道其中包含的值何时被修改。但是，将值存储在容器代理中（这会触发`__setitem__`代理对象上的代理）并通过管理器传播，因此为了有效修改此类项目，可以将修改的值重新分配给容器代理：

```
# create a list proxy and append a mutable object (a dictionary)
lproxy = manager.list()
lproxy.append({})
# now mutate the dictionary
```

```
d = lproxy[0]
d['a'] = 1
d['b'] = 2
# at this point, the changes to d are not yet synced, but by
# updating the dictionary, the proxy is notified of the change
lproxy[0] = d
```

这种方法可能不如在大多数使用情况下使用嵌套代理对象方便， 但也展示了对同步的控制级别。

注意: 代理类型multiprocessing无法支持按值进行比较。所以，例如，我们有：

```
>>> manager.list([1,2,3]) == [1,2,3]
False
```

在进行比较时，应该只使用指示对象的副本。

*类*multiprocessing.managers.BaseProxy

代理对象是的子类的实例BaseProxy。

_callmethod（*方法名*[，*ARGS*[，*kwds*]]）

调用并返回代理所指对象方法的结果。

如果proxy是代理人的对象是obj表达式

```
proxy._callmethod(methodname, args, kwds)
```

将评估表达

```
getattr(obj, methodname)(*args, **kwds)
```

在经理的过程中。

返回的值将是调用结果的副本或新共享对象的代理 - 请参阅*method_to_typeid* 参数的文档BaseManager.register()。

如果通话引发异常，则通过重新提出 _callmethod()。如果经理进程中出现其他异常，则将其转换为RemoteError异常并由其提出_callmethod()。

特别要注意的是，如果*methodname*尚未*公开*，将会引发异常。

使用的一个例子_callmethod()：

```
>>> l = manager.list(range(10))
>>> l._callmethod('__len__')
10
>>> l._callmethod('__getitem__', (slice(2, 7),)) # equivalent to l[2:7]
[2, 3, 4, 5, 6]
>>> l._callmethod('__getitem__', (20,))          # equivalent to l[20]
Traceback (most recent call last):
```

```
...
IndexError: list index out of range
```

_getvalue ( )
返回指称对象的副本。

如果指示物是不可抽出的，那么这将引发异常。

__repr__ ( )
返回代理对象的表示。

__str__ ( )
返回指示对象的表示。

17.2.2.8.1。清理

代理对象使用weakref回调函数，以便当它被垃圾回收后，它将从拥有其指示对象的管理器中注销自身。

当不再有任何代理引用共享对象时，会从管理进程中删除共享对象。

17.2.2.9。进程池

可以创建一个过程池，执行与Pool课程一起提交给它的任务。

class multiprocessing.pool. Pool ([*processes* [, *initializer* [, *initargs* [, *maxtasksperchild* [, *context*]]]]])
一个进程池对象，用于控制可提交作业的工作进程池。它支持超时和回调的异步结果，并具有并行映射实现。

*进程*是要使用的工作进程的数量。如果*过程*是 None然后通过返回的数字os.cpu_count() 被使用。

如果*初始化*程序不是，None则每个工作进程initializer(*initargs) 在启动时都会调用 。

*maxtasksperchild*是一个工作进程在退出之前可以完成的任务数量，*并由*新的工作进程替换，以便释放未使用的资源。默认的*maxtasksperchild*是None ，这意味着工作进程的生存时间与池一样长。

*上下文*可用于指定用于启动工作进程的上下文。通常使用函数multiprocessing.Pool() 或Pool() 上下文对象的方法创建池。在这两种情况*下*都可以适当设置。

请注意，池对象的方法只应由创建池的进程调用。

版本3.2中的新功能：maxtasksperchild

3.4版新增：上下文

注意： 一般情况下，工作进程在Pool池的工作队列的整个持续时间内都处于活动状态。在其他系统（如Apache，mod_wsgi等）中发现的释放工作人员资源的频繁模式是允许池中的工作人员在退出，清理和产生新的进程之前完成一定量的工作取代旧的。该*maxtasksperchild*参数来Pool自曝这种能力给最终用户。

apply（*func*[，*args*[，*kwds*]]）

使用参数*args*和关键字参数*kwds*调用*func*。它阻塞，直到结果准备就绪。鉴于这些块，更适合并行执行工作。此外，*func*只能在游泳池的其中一名工作人员*身上*执行。apply_async()

apply_async（*func*[，*args*[，*kwds*[，*callback*[，*error_callback*]]]]）

apply()返回结果对象的方法的变体。

如果指定了*回调*，那么它应该是一个可接受的参数。当结果变为就绪时，将对其应用*回调*，即除非调用失败，否则将 应用*error_callback*。

如果指定了*error_callback*，那么它应该是一个可接受一个参数的可调用函数。如果目标函数失败，则使用异常实例调用*error_callback*。

回调应该立即完成，否则处理结果的线程会被阻塞。

map（*func*，*iterable*[，*chunksize*]）

map()内置函数的并行等价物（尽管它只支持一个*可迭代的*参数）。它阻塞，直到结果准备就绪。

这种方法将迭代器切成许多块，并将它作为单独的任务提交给进程池。这些块的（近似）大小可以通过将*chunksize*设置为正整数来指定。

map_async（*func*，*iterable*[，*chunksize*[，*callback*[，*error_callback*]]]）

map()返回结果对象的方法的变体。

如果指定了*回调*，那么它应该是一个可接受的参数。当结果变为就绪时，将对其应用*回调*，即除非调用失败，否则将 应用*error_callback*。

如果指定了*error_callback*，那么它应该是一个可接受一个参数的可调用函数。如果目标函数失败，则使用异常实例调用*error_callback*。

回调应该立即完成，否则处理结果的线程会被阻塞。

imap（*func*，*iterable*[，*chunksize*]）

一个懒惰的版本map()。

所述*CHUNKSIZE*参数是与由所使用的一个map()方法。对于使用了一个较大的值很长iterables *CHUNKSIZE*可以使作业完成的**太多**比使用的默认值加快 1。

此外，如果*CHUNKSIZE*是1则next()通过返回的迭代器的方法imap()方法有一个可选的*超时*参数： next(timeout)将提高multiprocessing.TimeoutError如果结果不能内退回*超时*秒。

imap_unordered (*func* , *iterable* [, *chunksize*])

同样imap()，除了从返回的迭代结果的排序应该考虑随心所欲。（只有当只有一个工作进程时，才能保证“正确”的顺序。）

starmap (*func* , *iterable* [, *chunksize*])

就像map()除了*迭代*器的元素被期望是被解压为参数的迭代器之外。

因此，一个*可迭代*的结果。[(1,2), (3, 4)][func(1,2), func(3,4)]

3.3版本的新功能

starmap_async (*func* , *iterable* [, *chunksize* [, *callback* [, *error_callback*]]])

*iteables迭代*的迭代组合，starmap()并map_async()迭代 *迭代*，并调用*func*与解压缩的iterables。返回结果对象。

3.3版本的新功能

close ( )

防止将更多任务提交到池中。所有任务完成后，工作进程将退出。

terminate ( )

立即停止工作进程而不完成杰出的工作。当池对象被垃圾收集时terminate()会立即调用。

join ( )

等待工作进程退出。必须打电话close()或 terminate()在使用之前join()。

*3.3版新增功能：*池对象现在支持上下文管理协议 - 请参阅 上下文管理器类型。__enter__()返回池对象，并__exit__()调用terminate()。

*类*multiprocessing.pool.AsyncResult

由Pool.apply_async()和 返回的结果的类Pool.map_async()。

get ([*timeout*])

到达时返回结果。如果*超时*不是None，并且结果未在*超时*秒 内到达，multiprocessing.TimeoutError则会引发。如果远程调用引发异常，那么该异常将被重新调整get()。

wait ([*timeout*])

等到结果可用或*超时*秒数通过。

ready ( )

返回通话是否完成。

successful ( )

返回是否完成呼叫而不引发异常。AssertionError如果结果尚未准备好，将会升高。

以下示例演示了如何使用池：

```
from multiprocessing import Pool
import time

def f(x):
    return x*x

if __name__ == '__main__':
    with Pool(processes=4) as pool:          # start 4 worker processes
        result = pool.apply_async(f, (10,)) # evaluate "f(10)" asynchronously in a si
        print(result.get(timeout=1))         # prints "100" unless your computer is *v

        print(pool.map(f, range(10)))        # prints "[0, 1, 4,..., 81]"

        it = pool.imap(f, range(10))
        print(next(it))                      # prints "0"
        print(next(it))                      # prints "1"
        print(it.next(timeout=1))            # prints "4" unless your computer is *ver

        result = pool.apply_async(time.sleep, (10,))
        print(result.get(timeout=1))         # raises multiprocessing.TimeoutError
```

17.2.2.10。听众和客户

通常在进程之间传递消息是使用队列或使用 Connection返回的对象完成的 Pipe()。

但是，该multiprocessing.connection模块允许一些额外的灵活性。它基本上提供了一个高层面的面向消息的API来处理套接字或Windows命名管道。它还支持使用该模块的*摘要式身份验证*hmac，并支持同时轮询多个连接。

multiprocessing.connection.**deliver_challenge**(*连接*, *authkey*)

将随机生成的消息发送到连接的另一端并等待回复。

如果回复与使用*authkey*作为密钥的消息摘要匹配，则欢迎消息将发送到连接的另一端。否则 AuthenticationError会提出。

multiprocessing.connection.**answer_challenge**(*连接*, *authkey*)

接收消息，使用*authkey*作为密钥来*计算*消息的摘要，然后将摘要发回。

如果没有收到欢迎消息，则会 AuthenticationError提出。

multiprocessing.connection.**Client**(*address* [, *family* [, *authkey*]])

尝试建立到正在使用地址*地址*的侦听器的连接 ，返回一个Connection。

连接的类型由*族*参数决定，但通常可以省略，因为它通常可以从*地址*格式中推断出来 。（请参阅地址格式）

如果给出*authkey*而不是None，它应该是一个字节字符串，并将用作基于HMAC的认证质询的密钥。如果*authkey*为None，*则不*进行身份验证。 AuthenticationError如果身份验证失败，会引发此问题 请参阅认证密钥。

class multiprocessing.connection.Listener ([*address* [, *family* [, *backlog* [, *authkey*]]]])

绑定套接字或Windows命名管道的包装器，它正在侦听连接。

*地址*是监听器对象的绑定套接字或命名管道使用的地址。

> **注意：** 如果使用地址'0.0.0.0'，则该地址在Windows上不会是可连接的终点。如果您需要可连接的端点，则应使用“127.0.0.1”。

*family*是要使用的套接字（或命名管道）的类型。这可以是一个字符串'AF_INET'（对于TCP套接字），'AF_UNIX'（对于Unix域套接字）或'AF_PIPE'（对于Windows命名管道）。其中只有第一个保证可用。如果*家庭*是None从*地址*格式推断出*家庭*的话。如果*地址*也是 None默认选择。这个默认值是被认为是最快可用的家族。请参阅 地址格式。请注意，如果*系列*是 'AF_UNIX'和地址，None那么将使用创建的私有临时目录创建套接字tempfile.mkstemp()。

如果侦听器对象使用套接字，则在绑定后，*backlog*（默认为1）将传递给listen()套接字的方法。

如果给出*authkey*而不是None，它应该是一个字节字符串，并将用作基于HMAC的认证质询的密钥。如果*authkey*为None，*则不*进行身份验证。 AuthenticationError如果身份验证失败，会引发此问题 请参阅认证密钥。

accept ( )

接受监听器对象的绑定套接字或命名管道上的连接并返回一个Connection对象。如果尝试进行身份验证并失败，则会 AuthenticationError引发此问题。

close ( )

关闭侦听器对象的绑定套接字或命名管道。当收听器被垃圾收集时，这被自动调用。但明确地称它是明智的。

侦听器对象具有以下只读属性：

address

Listener对象正在使用的地址。

last_accepted

上次接受连接的地址。如果这不可用，那就是了None。

*版本3.3中的新功能：*侦听器对象现在支持上下文管理协议 - 请参阅 上下文管理器类型。__enter__()返回侦听器对象，然后__exit__()调用close()。

multiprocessing.connection.wait (*object_list* , *timeout* = *None*)

等待*object_list中*的对象准备就绪。返回*object_list*中已准备好的对象的列表。如果*超时*是一个浮点数，那么该呼叫将阻塞至多几秒钟。如果 *超时*是None那么它会阻止无限期。负超时等同于零超时。

对于Unix和Windows，如果是*object_list*，则对象可以出现在*object_list*中

- 一个可读的Connection对象;
- 一个连接和可读的socket.socket对象;要么
- 对象的sentinel属性 Process。

当有数据可以从中读取或另一端已关闭时，连接或套接字对象已准备就绪。

Unix：几乎相同 。不同的是，如果被一个信号中断，它可以以一个错误号码提升，而 不会 。 wait(object_list, timeout) select.select(object_list, [], [], timeout) select.select() OSError EINTR wait()

Windows：*object_list中*的项目必须是可等待的整型句柄（根据Win32函数文档使用的定义 WaitForMultipleObjects()），或者可以是带有fileno()返回套接字句柄或管道句柄的方法的对象。（请注意，管柄和套管手柄**不是**等待手柄。）

3.3版本的新功能

例子

以下服务器代码创建一个用作身份验证密钥的侦听器。然后等待连接并将一些数据发送到客户端：'secret password'

```
from multiprocessing.connection import Listener
from array import array

address = ('localhost', 6000)     # family is deduced to be 'AF_INET'

with Listener(address, authkey=b'secret password') as listener:
    with listener.accept() as conn:
        print('connection accepted from', listener.last_accepted)

        conn.send([2.25, None, 'junk', float])

        conn.send_bytes(b'hello')

        conn.send_bytes(array('i', [42, 1729]))
```

以下代码连接到服务器并从服务器接收一些数据：

```
from multiprocessing.connection import Client
from array import array

address = ('localhost', 6000)

with Client(address, authkey=b'secret password') as conn:
    print(conn.recv())                  # => [2.25, None, 'junk', float]

    print(conn.recv_bytes())            # => 'hello'

    arr = array('i', [0, 0, 0, 0, 0])
    print(conn.recv_bytes_into(arr))    # => 8
    print(arr)                          # => array('i', [42, 1729, 0, 0, 0])
```

以下代码用于wait()同时等待来自多个进程的消息：

```
import time, random
from multiprocessing import Process, Pipe, current_process
from multiprocessing.connection import wait

def foo(w):
    for i in range(10):
        w.send((i, current_process().name))
    w.close()

if __name__ == '__main__':
    readers = []

    for i in range(4):
        r, w = Pipe(duplex=False)
        readers.append(r)
        p = Process(target=foo, args=(w,))
        p.start()
        # We close the writable end of the pipe now to be sure that
        # p is the only process which owns a handle for it.  This
        # ensures that when p closes its handle for the writable end,
        # wait() will promptly report the readable end as being ready.
        w.close()

    while readers:
        for r in wait(readers):
            try:
                msg = r.recv()
            except EOFError:
                readers.remove(r)
            else:
                print(msg)
```

17.2.2.10.1。地址格式

- 一个'AF_INET'地址是以下形式的元组，其中 *主机名*是一个字符串，*端口*是一个整数。 (hostname, port)
- 一个'AF_UNIX'地址是代表文件系统中的文件名的字符串。
- 一个'AF_PIPE'地址是格式的字符串

  r'\\.\pipe*PipeName*'。要用于Client()连接到名为*ServerName*的远程计算机上的命名管道，应该使用表单的地址。r'*ServerName*\pipe*PipeName*'

请注意，任何以两个反斜杠开头的字符串默认为'AF_PIPE'地址而不是'AF_UNIX'地址。

17.2.2.11。验证码

当使用时Connection.recv，收到的数据会自动取消。不幸的是，从不可信来源取出数据是一种安全风险。因此Listener，请Client()使用该hmac模块提供摘要式身份验证。

认证密钥是一个可以被认为是密码的字节串：一旦连接建立，两端都会要求证明另一端知道认证密钥。（证明两端都使用相同的密钥并**没有**涉及发送通过连接键。）

如果请求身份验证但未指定身份验证密钥，current_process().authkey则使用返回值（请参阅Process）。该值将由Process当前进程创建的任何对象自动继承。这意味着（默认情况下）多

进程程序的所有进程将共享一个身份验证密钥，可以在设置它们之间的连接时使用该身份验证密钥。

合适的认证密钥也可以通过使用生成os.urandom()。

17.2.2.12。记录

有些支持日志记录功能。但请注意，logging 程序包不使用进程共享锁，因此可能（取决于处理程序类型）来自不同进程的消息混淆。

multiprocessing.get_logger ( )

返回使用的记录器multiprocessing。如有必要，将创建一个新的。

首次创建时，记录器具有级别logging.NOTSET并且没有默认处理程序。发送到此记录器的消息不会默认传播到根记录器。

请注意，在Windows上，子进程只会继承父进程记录器的级别 - 记录器的其他任何自定义都不会被继承。

multiprocessing.log_to_stderr ( )

该函数执行一个调用，get_logger()但除了返回由get_logger创建的记录器之外，它还添加了一个处理程序，该处理程序将输出发送到sys.stderr使用格式 。'[%(levelname)s/%(processName)s] %(message)s'

以下是打开日志记录的示例会话：

```
>>> import multiprocessing, logging
>>> logger = multiprocessing.log_to_stderr()
>>> logger.setLevel(logging.INFO)
>>> logger.warning('doomed')
[WARNING/MainProcess] doomed
>>> m = multiprocessing.Manager()
[INFO/SyncManager-...] child process calling self.run()
[INFO/SyncManager-...] created temp directory /.../pymp-...
[INFO/SyncManager-...] manager serving at '/.../listener-...'
>>> del m
[INFO/MainProcess] sending shutdown message to manager
[INFO/SyncManager-...] manager exiting with exitcode 0
```

有关日志级别的完整表格，请参阅logging模块。

17.2.2.13。该multiprocessing.dummy模块

multiprocessing.dummy复制API的multiprocessing但不过是threading模块的包装。

17.2.3。编程指导

使用时应遵守一定的准则和习惯用语 multiprocessing。

17.2.3.1。所有启动方法

以下内容适用于所有启动方法。

避免共享状态

尽可能避免在进程间转移大量数据。

可能最好坚持使用队列或管道进行进程之间的通信，而不是使用较低级别的同步原语。

Picklability

确保代理方法的参数是可挑选的。

代理线程安全

除非用锁保护它，否则不要使用来自多个线程的代理对象。

（使用*相同*代理的不同进程永远不会有问题。）

加入僵尸进程

在Unix上，当进程结束但尚未加入时，它变成僵尸。不应该有很多，因为每当新进程开始（或被 `active_children()` 调用）时，所有已完成的尚未加入的进程都将被加入。也调用完成的进程`Process.is_alive`将加入该进程。即使如此，明确加入您开始的所有流程也可能是一种很好的做法。

继承不如pickle / unpickle

当使用*spawn*或*forkserver*启动方法时，`multiprocessing`需要选择多种类型，以便子进程可以使用它们。但是，通常应避免使用管道或队列将共享对象发送到其他进程。相反，您应该安排程序，以便需要访问在别处创建的共享资源的进程可以从祖先进程继承它。

避免终止进程

使用该`Process.terminate` 方法停止进程可能会导致进程当前正在使用的任何共享资源（例如锁，信号量，管道和队列）被破坏或不可用于其他进程。

因此，最好只考虑 `Process.terminate`在不使用任何共享资源的进程上使用。

加入使用队列的进程

请记住，已将项目放入队列的进程将在终止之前等待，直到所有缓冲项目都由“馈线”线程馈送到基础管道。（子进程可以调用`Queue.cancel_join_thread` 队列的 方法来避免这种行为。）

这意味着无论什么时候使用队列，都需要确保放入队列中的所有项目在加入之前最终都会被删除。否则，您无法确定将项目放入队列的进程将终止。还要记住，非守护进程会自动加入。

一个会导致死锁的例子如下：

```
from multiprocessing import Process, Queue

def f(q):
    q.put('X' * 1000000)

if __name__ == '__main__':
    queue = Queue()
    p = Process(target=f, args=(queue,))
    p.start()
    p.join()                    # this deadlocks
    obj = queue.get()
```

这里的一个解决方法是交换最后两行（或者直接删除 p.join() 行）。

明确地将资源传递给子进程

在使用*fork* start方法的Unix上，子进程可以使用在父进程中使用全局资源创建的共享资源。但是，最好将该对象作为参数传递给子进程的构造函数。

除了使代码（可能）与Windows以及其他启动方法兼容外，这还可以确保只要子进程仍然存在，对象将不会在父进程中被垃圾回收。如果在父进程中垃圾收集对象时某些资源被释放，这可能很重要。

举例来说

```
from multiprocessing import Process, Lock

def f():
    ... do something using "lock" ...

if __name__ == '__main__':
    lock = Lock()
    for i in range(10):
        Process(target=f).start()
```

应改写为

```
from multiprocessing import Process, Lock

def f(l):
    ... do something using "l" ...

if __name__ == '__main__':
    lock = Lock()
    for i in range(10):
        Process(target=f, args=(lock,)).start()
```

谨防用sys.stdin“像对象的文件”

multiprocessing 原本被无条件地称为：

```
os.close(sys.stdin.fileno())
```

在该multiprocessing.Process._bootstrap()方法中 - 这导致了流程中的问题。这已被更改为：

```
sys.stdin.close()
sys.stdin = open(os.open(os.devnull, os.O_RDONLY), closefd=False)
```

它解决了进程之间相互冲突的基本问题，导致了错误的文件描述符错误，但却给应用程序带来了潜在的危险，这些应用程序将替换sys.stdin()为带有输出缓冲的“类文件对象”。这种危险是，如果多个进程调用 close()这个类文件对象，可能会导致相同的数据被多次刷新到对象，从而导致损坏。

如果您编写一个类似文件的对象并实现自己的缓存，那么无论何时添加到缓存中，都可以通过存储pid来使其更安全，并在pid更改时放弃缓存。例如：

```
@property
def cache(self):
    pid = os.getpid()
    if pid != self._pid:
        self._pid = pid
        self._cache = []
    return self._cache
```

有关更多信息，请参阅bpo-5155，bpo-5313和bpo-5331

17.2.3.2。该*菌种*和*forkserver*启动方法

有一些额外的限制不适用于*fork* start方法。

更多的可拾取性

确保所有参数Process.__init__()都是可选择的。此外，如果您继承了子类，Process那么确保在Process.start调用方法时可以选择实例。

全局变量

请记住，如果在子进程中运行的代码尝试访问全局变量，则它所看到的值（如果有）可能与Process.start被调用时父进程中的值不同。

但是，仅仅是模块级别常量的全局变量不会导致问题。

主模块安全导入

确保主模块可以通过新的Python解释器安全地导入，而不会引起意外的副作用（例如开始一个新过程）。

例如，使用运行以下模块的*spawn*或*forkserver* start方法会失败，并显示RuntimeError：

```
from multiprocessing import Process

def foo():
    print('hello')
```

```
p = Process(target=foo)
p.start()
```

相反，应该使用以下方式来保护程序的“切入点”：`if __name__ == '__main__':`

```
from multiprocessing import Process, freeze_support, set_start_method

def foo():
    print('hello')

if __name__ == '__main__':
    freeze_support()
    set_start_method('spawn')
    p = Process(target=foo)
    p.start()
```

（`freeze_support()`如果程序正常运行而不是冻结，可以省略该行。）

这允许新产生的Python解释器安全地导入模块，然后运行模块的`foo()`功能。

如果在主模块中创建池或管理器，则会应用类似的限制。

17.2.4。示例

演示如何创建和使用定制的管理人员和代理人：

```
from multiprocessing import freeze_support
from multiprocessing.managers import BaseManager, BaseProxy
import operator

##

class Foo:
    def f(self):
        print('you called Foo.f()')
    def g(self):
        print('you called Foo.g()')
    def _h(self):
        print('you called Foo._h()')

# A simple generator function
def baz():
    for i in range(10):
        yield i*i

# Proxy type for generator objects
class GeneratorProxy(BaseProxy):
    _exposed_ = ['__next__']
    def __iter__(self):
        return self
    def __next__(self):
        return self._callmethod('__next__')
```

```
# Function to return the operator module
def get_operator_module():
    return operator

##

class MyManager(BaseManager):
    pass

# register the Foo class; make `f()` and `g()` accessible via proxy
MyManager.register('Foo1', Foo)

# register the Foo class; make `g()` and `_h()` accessible via proxy
MyManager.register('Foo2', Foo, exposed=('g', '_h'))

# register the generator function baz; use `GeneratorProxy` to make proxies
MyManager.register('baz', baz, proxytype=GeneratorProxy)

# register get_operator_module(); make public functions accessible via proxy
MyManager.register('operator', get_operator_module)

##

def test():
    manager = MyManager()
    manager.start()

    print('-' * 20)

    f1 = manager.Foo1()
    f1.f()
    f1.g()
    assert not hasattr(f1, '_h')
    assert sorted(f1._exposed_) == sorted(['f', 'g'])

    print('-' * 20)

    f2 = manager.Foo2()
    f2.g()
    f2._h()
    assert not hasattr(f2, 'f')
    assert sorted(f2._exposed_) == sorted(['g', '_h'])

    print('-' * 20)

    it = manager.baz()
    for i in it:
        print('<%d>' % i, end=' ')
    print()

    print('-' * 20)

    op = manager.operator()
    print('op.add(23, 45) =', op.add(23, 45))
    print('op.pow(2, 94) =', op.pow(2, 94))
    print('op._exposed_ =', op._exposed_)

##
```

```
if __name__ == '__main__':
    freeze_support()
    test()
```

使用Pool：

```
import multiprocessing
import time
import random
import sys

#
# Functions used by test code
#

def calculate(func, args):
    result = func(*args)
    return '%s says that %s%s = %s' % (
        multiprocessing.current_process().name,
        func.__name__, args, result
        )

def calculatestar(args):
    return calculate(*args)

def mul(a, b):
    time.sleep(0.5 * random.random())
    return a * b

def plus(a, b):
    time.sleep(0.5 * random.random())
    return a + b

def f(x):
    return 1.0 / (x - 5.0)

def pow3(x):
    return x ** 3

def noop(x):
    pass

#
# Test code
#

def test():
    PROCESSES = 4
    print('Creating pool with %d processes\n' % PROCESSES)

    with multiprocessing.Pool(PROCESSES) as pool:
        #
        # Tests
        #

        TASKS = [(mul, (i, 7)) for i in range(10)] + \
```

```
        [(plus, (i, 8)) for i in range(10)]

    results = [pool.apply_async(calculate, t) for t in TASKS]
    imap_it = pool.imap(calculatestar, TASKS)
    imap_unordered_it = pool.imap_unordered(calculatestar, TASKS)

    print('Ordered results using pool.apply_async():')
    for r in results:
        print('\t', r.get())
    print()

    print('Ordered results using pool.imap():')
    for x in imap_it:
        print('\t', x)
    print()

    print('Unordered results using pool.imap_unordered():')
    for x in imap_unordered_it:
        print('\t', x)
    print()

    print('Ordered results using pool.map() --- will block till complete:')
    for x in pool.map(calculatestar, TASKS):
        print('\t', x)
    print()

    #
    # Test error handling
    #

    print('Testing error handling:')

    try:
        print(pool.apply(f, (5,)))
    except ZeroDivisionError:
        print('\tGot ZeroDivisionError as expected from pool.apply()')
    else:
        raise AssertionError('expected ZeroDivisionError')

    try:
        print(pool.map(f, list(range(10))))
    except ZeroDivisionError:
        print('\tGot ZeroDivisionError as expected from pool.map()')
    else:
        raise AssertionError('expected ZeroDivisionError')

    try:
        print(list(pool.imap(f, list(range(10)))))
    except ZeroDivisionError:
        print('\tGot ZeroDivisionError as expected from list(pool.imap())')
    else:
        raise AssertionError('expected ZeroDivisionError')

    it = pool.imap(f, list(range(10)))
    for i in range(10):
        try:
            x = next(it)
        except ZeroDivisionError:
```

```
                if i == 5:
                    pass
            except StopIteration:
                break
            else:
                if i == 5:
                    raise AssertionError('expected ZeroDivisionError')

        assert i == 9
        print('\tGot ZeroDivisionError as expected from IMapIterator.next()')
        print()

        #
        # Testing timeouts
        #

        print('Testing ApplyResult.get() with timeout:', end=' ')
        res = pool.apply_async(calculate, TASKS[0])
        while 1:
            sys.stdout.flush()
            try:
                sys.stdout.write('\n\t%s' % res.get(0.02))
                break
            except multiprocessing.TimeoutError:
                sys.stdout.write('.')
        print()
        print()

        print('Testing IMapIterator.next() with timeout:', end=' ')
        it = pool.imap(calculatestar, TASKS)
        while 1:
            sys.stdout.flush()
            try:
                sys.stdout.write('\n\t%s' % it.next(0.02))
            except StopIteration:
                break
            except multiprocessing.TimeoutError:
                sys.stdout.write('.')
        print()
        print()


if __name__ == '__main__':
    multiprocessing.freeze_support()
    test()
```

显示如何使用队列将任务提供给工作进程集合并收集结果的示例：

```
import time
import random

from multiprocessing import Process, Queue, current_process, freeze_support

#
# Function run by worker processes
#
```

```
def worker(input, output):
    for func, args in iter(input.get, 'STOP'):
        result = calculate(func, args)
        output.put(result)

#
# Function used to calculate result
#

def calculate(func, args):
    result = func(*args)
    return '%s says that %s%s = %s' % \
        (current_process().name, func.__name__, args, result)

#
# Functions referenced by tasks
#

def mul(a, b):
    time.sleep(0.5*random.random())
    return a * b

def plus(a, b):
    time.sleep(0.5*random.random())
    return a + b

#
#
#

def test():
    NUMBER_OF_PROCESSES = 4
    TASKS1 = [(mul, (i, 7)) for i in range(20)]
    TASKS2 = [(plus, (i, 8)) for i in range(10)]

    # Create queues
    task_queue = Queue()
    done_queue = Queue()

    # Submit tasks
    for task in TASKS1:
        task_queue.put(task)

    # Start worker processes
    for i in range(NUMBER_OF_PROCESSES):
        Process(target=worker, args=(task_queue, done_queue)).start()

    # Get and print results
    print('Unordered results:')
    for i in range(len(TASKS1)):
        print('\t', done_queue.get())

    # Add more tasks using `put()`
    for task in TASKS2:
        task_queue.put(task)

    # Get and print some more results
    for i in range(len(TASKS2)):
```

```
        print('\t', done_queue.get())

    # Tell child processes to stop
    for i in range(NUMBER_OF_PROCESSES):
        task_queue.put('STOP')


if __name__ == '__main__':
    freeze_support()
    test()
```

17.3。该concurrent包

目前，该软件包中只有一个模块：

- `concurrent.futures` - 启动并行任务

17.4。concurrent.futures- 启动并行任务

3.2版本中的新功能

源代码： Lib / concurrent / futures / thread.py 和Lib / concurrent / futures / process.py

该concurrent.futures模块为异步执行可调用对象提供了一个高级接口。

异步执行可以使用线程，使用 ThreadPoolExecutor 或单独的进程来执行ProcessPoolExecutor。两者都实现了由抽象Executor类定义的相同接口。

17.4.1。执行对象

*类*concurrent.futures.Executor

抽象类，提供异步执行调用的方法。它不应该直接使用，而应该通过其具体的子类来使用。

submit（*fn*，**args*，***kwargs* ）

将可调用函数*fn*安排为执行， 并返回一个表示可调用执行的对象。fn(*args **kwargs)Future

```
with ThreadPoolExecutor(max_workers=1) as executor:
    future = executor.submit(pow, 323, 1235)
    print(future.result())
```

map（*func*，** iterables*，*timeout = None*，*chunksize = 1* ）

与以下类似：map(func, *iterables)

- 在*iterables*收集立即而不是懒洋洋地;
- *func*是异步执行的，可以同时调用 *func*。

返回的迭代器引发一个concurrent.futures.TimeoutError if __next__()被调用，并且结果在原始调用的*超时*秒后不可用Executor.map()。 *超时*可以是一个int或一个浮点数。如果没有指定*超时*，或者 None等待时间没有限制。

如果*func*调用引发异常，那么当从迭代器中检索到该异常时将引发该异常。

在使用时ProcessPoolExecutor，此方法 会将*迭代器*切成许多块，并将其作为单独的任务提交给池。这些块的（近似）大小可以通过将*chunksize*设置为正整数来指定。对于很长的iterables，采用大值*CHUNKSIZE*可以显著改善性能相比的1.默认大小 ThreadPoolExecutor，*CHUNKSIZE*没有效果。

*版本3.5中已更改：*添加了*chunksize*参数。

shutdown（*wait = True* ）

执行者指示它应该释放当前挂起的期货执行完成时正在使用的任何资源。呼叫 Executor.submit()和Executor.map()关机后提出RuntimeError。

如果*等待*，True那么这个方法将不会返回，直到所有挂起的期货完成执行并且与执行器相关联的资源都被释放。如果*等待*，False那么这个方法将立即返回，当所有未完成的期货完成执行时，与执行器相关的资源将被释放。无论*等待*的价值如何，整个Python程序都将不会退出，直到所有挂起的期货都执行完毕。

如果使用with语句，可以避免必须显式调用此方法 ，该语句将关闭 Executor （等待Executor.shutdown()被调用*wait*设置为 True ）：

```
import shutil
with ThreadPoolExecutor(max_workers=4) as e:
    e.submit(shutil.copy, 'src1.txt', 'dest1.txt')
    e.submit(shutil.copy, 'src2.txt', 'dest2.txt')
    e.submit(shutil.copy, 'src3.txt', 'dest3.txt')
    e.submit(shutil.copy, 'src4.txt', 'dest4.txt')
```

17.4.2。ThreadPoolExecutor的

ThreadPoolExecutor是一个Executor使用线程池异步执行调用的子类。

当与a关联的可调用对象Future等待另一个结果时，可能发生死锁Future。例如：

```
import time
def wait_on_b():
    time.sleep(5)
    print(b.result())  # b will never complete because it is waiting on a.
    return 5

def wait_on_a():
    time.sleep(5)
    print(a.result())  # a will never complete because it is waiting on b.
    return 6


executor = ThreadPoolExecutor(max_workers=2)
a = executor.submit(wait_on_b)
b = executor.submit(wait_on_a)
```

和：

```
def wait_on_future():
    f = executor.submit(pow, 5, 2)
    # This will never complete because there is only one worker thread and
    # it is executing this function.
    print(f.result())

executor = ThreadPoolExecutor(max_workers=1)
executor.submit(wait_on_future)
```

class concurrent.futures. ThreadPoolExecutor（*max_workers* = *None*，*thread_name_prefix* =''）

一个Executor最多使用*max_workers* 线程池来异步执行调用的子类。

*在版本3.5中进行了更改：*如果*max_workers*是None或者没有给出，它将默认为机器上的处理器数乘以5，假设ThreadPoolExecutor通常用于重叠I / O而不是CPU工作，并且工人数应该高于工人的数量ProcessPoolExecutor。

*在3.6版本的新功能：*该*thread_name_prefix*加入参数允许用户控制由池更容易调试创建工作线程threading.Thread名。

17.4.2.1。ThreadPoolExecutor示例

```
import concurrent.futures
import urllib.request

URLS = ['http://www.foxnews.com/',
        'http://www.cnn.com/',
        'http://europe.wsj.com/',
        'http://www.bbc.co.uk/',
        'http://some-made-up-domain.com/']

# Retrieve a single page and report the URL and contents
def load_url(url, timeout):
    with urllib.request.urlopen(url, timeout=timeout) as conn:
        return conn.read()

# We can use a with statement to ensure threads are cleaned up promptly
with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
    # Start the load operations and mark each future with its URL
    future_to_url = {executor.submit(load_url, url, 60): url for url in URLS}
    for future in concurrent.futures.as_completed(future_to_url):
        url = future_to_url[future]
        try:
            data = future.result()
        except Exception as exc:
            print('%r generated an exception: %s' % (url, exc))
        else:
            print('%r page is %d bytes' % (url, len(data)))
```

17.4.3。ProcessPoolExecutor

本ProcessPoolExecutor类是Executor使用的过程池异步执行调用子类。 ProcessPoolExecutor使用该multiprocessing模块，该模块允许它旁听Global Interpreter Lock，但也意味着只有可执行对象才能被执行和返回。

该__main__模块必须可由工作程序子进程导入。这意味着ProcessPoolExecutor在交互式解释器中不起作用。

从提交给a的可调用对象调用Executor或Future方法ProcessPoolExecutor将导致死锁。

class concurrent.futures.ProcessPoolExecutor (*max_workers = None*)

Executor使用至多*max_workers*进程池异步执行调用的子类。如果*max_workers*是None或者没有给出，它将默认为机器上的处理器数量。如果*max_workers小于*或等于0，ValueError 则会引发a。

*版本3.3中更改：*其中一个工作进程突然终止时， BrokenProcessPool现在会引发错误。以前，行为是不确定的，但对执行者或其未来的操作往往会冻结或僵局。

17.4.3.1。ProcessPoolExecutor示例

```
import concurrent.futures
import math

PRIMES = [
    112272535095293,
    112582705942171,
    112272535095293,
    115280095190773,
    115797848077099,
    1099726899285419]

def is_prime(n):
    if n % 2 == 0:
        return False

    sqrt_n = int(math.floor(math.sqrt(n)))
    for i in range(3, sqrt_n + 1, 2):
        if n % i == 0:
            return False
    return True

def main():
    with concurrent.futures.ProcessPoolExecutor() as executor:
        for number, prime in zip(PRIMES, executor.map(is_prime, PRIMES)):
            print('%d is prime: %s' % (number, prime))

if __name__ == '__main__':
    main()
```

17.4.4。未来的对象

所述Future类封装一个可调用的异步执行。 Future实例由创建Executor.submit()。

*类*concurrent.futures.Future

封装可调用对象的异步执行。 Future 实例由Executor.submit()测试创建，不应直接创建。

cancel ( )

尝试取消通话。如果该呼叫当前正在执行且无法取消，则该方法将返回False，否则该呼叫将被取消并且该方法将返回True。

cancelled ( )

True如果通话成功取消，则返回。

running ( )

返回True如果当前正在执行的呼叫，无法取消。

done ( )

返回True如果调用成功取消或结束运行。

result (*timeout* = *None*)

返回通话返回的值。如果通话尚未完成，则此方法将等待*超时*秒数。如果通话没有在*超时*秒内 完成， concurrent.futures.TimeoutError则会提出a。*超时*可以是一个int或float。如果没有指定*超时*，或者None等待时间没有限制。

如果未来在完成前被取消，那么CancelledError 将会被提出。

如果调用提出，这个方法将引发同样的异常。

exception (*timeout* = *None*)

返回通话引发的异常。如果通话尚未完成，则此方法将等待*超时*秒数。如果通话没有在*超时*秒内 完成， concurrent.futures.TimeoutError则会提出a。 *超时*可以是一个int或float。如果没有指定*超时*，或者None等待时间没有限制。

如果未来在完成前被取消，那么CancelledError 将会被提出。

如果通话没有完成， None则返回。

add_done_callback (*fn*)

将可调用函数*fn*附加到未来。 当未来被取消或完成时，*fn*将被称为未来作为唯一的参数。

添加的可调用按照它们被添加的顺序调用，并且总是在属于添加它们的进程的线程中调用。如果可调用引发一个Exception子类，它将被记录并被忽略。如果可调用引发一个BaseException子类，则行为是未定义的。

如果未来已经完成或取消，*fn*将立即被调用。

以下Future方法适用于单元测试和 Executor实现。

set_running_or_notify_cancel ( )

这个方法只能Executor在执行与Future单元测试相关的工作之前由实现来调用。

如果方法返回False ， Future则被取消， Future.cancel()即被调用并返回*True*。任何等待Future完成（即通过 as_completed()或wait() ）的线程都会被唤醒。

如果方法返回，True那么这个Future未被取消并且已经处于运行状态，即调用 Future.running() 将返回*True*。

此方法只能被调用一次后不能被称为 Future.set_result() 或 Future.set_exception() 已被调用。

set_result（*结果*）

设置与相关的工作结果Future，以 *结果*。

这个方法只能用于Executor实现和单元测试。

set_exception（*例外*）

设置与Future该 *异常*相关的工作结果。 Exception

这个方法只能用于Executor实现和单元测试。

17.4.5。模块功能

concurrent.futures.wait（*fs*，*timeout = None*，*return_when = ALL_COMPLETED* ）

等待*fs*给出的Future实例（可能由不同Executor实例创建 ）完成。返回一个名为2的元组。第一组命名包含在等待完成之前完成（完成或被取消）的期货。第二套名称包含未完成的期货。 donenot_done

*超时*可用于控制返回之前等待的最大秒数。 *超时*可以是一个int或float。如果没有指定*超时*，或者None等待时间没有限制。

*return_when*指示此函数何时应返回。它必须是以下常量之一：

不变	描述
FIRST_COMPLETED	该功能将在任何未来完成或取消时返回。
FIRST_EXCEPTION	该函数将在任何未来通过引发异常完成时返回。如果没有未来引发异常，那么它相当于ALL_COMPLETED。
ALL_COMPLETED	该功能将在所有期货结束或被取消时返回。

concurrent.futures.as_completed（*fs*，*timeout = None* ）

返回由*fs*给出的Future实例（可能由不同Executor实例创建）的迭代器，这些实例在完成（完成或取消）时产生期货。由*fs*提供的任何重复的期货将被返还一次。任何之前完成的期货 都将被收回。返回的迭代器引发一个if 被调用，并且结果在原始调用的*超时*秒后不可用。 *超时*可以是一个int或float。如果 没有指定*超时*，或者等待时间没有限制。 as_completed() concurrent.futures.TimeoutError __next__() as_completed() None

也可以看看:

PEP 3148 - 期货 - 异步执行计算

描述此功能以包含在Python标准库中的建议。

17.4.6。异常类

*异常*concurrent.futures.CancelledError

未来取消时引发。

*异常*concurrent.futures.TimeoutError

未来的操作超过给定的超时时提高。

*异常*concurrent.futures.process.BrokenProcessPool

派生自此的RuntimeError异常类是在a的一个工作人员ProcessPoolExecutor以不干净的方式终止时（例如，如果它从外部被杀）而引发的。

3.3版本的新功能

17.5。 subprocess-子流程管理

源代码： Lib / subprocess.py

该subprocess模块允许你产生新的进程，连接到他们的输入/输出/错误管道，并获得他们的返回代码。该模块旨在替换几个较旧的模块和功能：

```
os.system
os.spawn*
```

有关如何使用subprocess模块来替换这些模块和功能的信息可以在以下各节中找到。

也可以看看： **PEP 324** - PEP提出了子流程模块

17.5.1。使用subprocess模块

建议调用子进程的方法是run() 对所有可以处理的用例使用该函数。对于更高级的用例，底层Popen接口可以直接使用。

该run()功能是在Python 3.5中添加的; 如果您需要保留与旧版本的兼容性，请参阅较旧的高级API部分。

subprocess.**run**(*args*，*，*stdin = None*，*input = None*，*stdout = None*，*stderr = None*，*shell = False*，*cwd = None*，*timeout = None*，*check = False*，*encoding = None*，*errors = None*)

运行*args*描述的命令。等待命令完成，然后返回一个CompletedProcess实例。

上面显示的参数仅仅是最常用的参数，在下面的常用参数（因此在缩写签名中使用关键字符号）中进行了描述。全功能签名与Popen构造函数大体相同- 除了*超时*，*输入*和*检查外*，该函数的所有参数都被传递给该接口。

这不会捕获默认情况下的stdout或stderr。要做到这一点，通过 PIPE对*标准输出*和/或*标准错误*的论点。

在*超时*参数传递给Popen.communicate()。如果超时过期，子进程将被终止并等待。该TimeoutExpired异常会被重新提出的子进程终止后。

该*输入*参数传递给Popen.communicate()，因此到子进程的标准输入。如果使用它，则必须是字节序列，或者如果指定了*encoding*或*errors*或者*universal_newlines*为true，则为字符串 。使用时，内部Popen对象会自动创建 stdin=PIPE，并且*stdin*参数也可能不会被使用。

如果*检查结果*为真，并且该进程以非零退出代码退出， CalledProcessError则会引发异常。该异常的属性包含参数，退出代码以及stdout和stderr（如果它们被捕获）。

如果指定了*编码*或*错误*，或者*universal_newlines*为true，则stdin，stdout和stderr的文件对象将在文本模式下使用指定的*编码*和*错误*或io.TextIOWrapper默认值打开。否则，文件对

象以二进制模式打开。

例子：

```
>>> subprocess.run(["ls", "-l"])  # doesn't capture output
CompletedProcess(args=['ls', '-l'], returncode=0)

>>> subprocess.run("exit 1", shell=True, check=True)
Traceback (most recent call last):
  ...
subprocess.CalledProcessError: Command 'exit 1' returned non-zero exit status 1

>>> subprocess.run(["ls", "-l", "/dev/null"], stdout=subprocess.PIPE)
CompletedProcess(args=['ls', '-l', '/dev/null'], returncode=0,
stdout=b'crw-rw-rw- 1 root root 1, 3 Jan 23 16:23 /dev/null\n')
```

3.5版本中的新功能。

*在版本3.6中更改：*添加了*编码*和*错误*参数

*类*subprocess.CompletedProcess

run()表示已完成的进程的返回值。

args

用于启动流程的参数。这可能是一个列表或一个字符串。

returncode

退出子进程的状态。通常，退出状态为0表示它已成功运行。

负值-N表示孩子被信号终止 N（仅限POSIX）。

stdout

从子进程中捕获stdout。一个字节序列，或者一个字符串，如果 run() 使用编码或错误进行调用。None如果stdout未被捕获。

如果你运行这个过程stderr=subprocess.STDOUT，stdout和stderr将会被合并到这个属性中，并且stderr会被执行 None。

stderr

从子进程中捕获stderr。一个字节序列，或者一个字符串，如果 run() 使用编码或错误进行调用。None如果stderr没有被捕获。

check_returncode()

如果returncode不为零，则提出一个CalledProcessError。

3.5版本中的新功能。

subprocess.DEVNULL

可以用作*标准输入*，*标准输出*或*标准错误*参数的特殊值，Popen并指示os.devnull 将使用特殊文件。

3.3版本的新功能

subprocess.PIPE
特殊值，可用作*标准输入*，*标准输出*或*标准错误*参数，Popen并指示应打开标准流的管道。最有用的Popen.communicate()。

subprocess.STDOUT
可以用作*stderr*参数的特殊值，Popen并指示标准错误应该与标准输出进入相同的句柄。

*异常*subprocess.SubprocessError
该模块的所有其他异常的基类。

3.3版本的新功能

*异常*subprocess.TimeoutExpired
SubprocessError在等待子进程时超时过期时引发的子类。

cmd
用于产生子进程的命令。

timeout
以秒为单位的超时。

output
子进程的输出，如果它被run()or 捕获check_output()。否则，None。

stdout
用于输出的别名，用于对称stderr。

stderr
子进程的Stderr输出，如果它被捕获run()。否则，None。

3.3版本的新功能

*在版本3.5中进行了更改：*添加了*标准输出*和*标准错误*属性

*异常*subprocess.CalledProcessError
SubprocessError当进程运行check_call()或check_output()返回非零退出状态时引发的 子类。

returncode
退出子进程的状态。如果过程由于信号而退出，则这将是负信号编号。

cmd
用于产生子进程的命令。

output
子进程的输出，如果它被run()or 捕获check_output()。否则，None。

stdout

用于输出的别名，用于对称stderr。

stderr

子进程的Stderr输出，如果它被捕获run()。否则，None。

*在版本3.5中进行了更改：*添加了*标准输出*和*标准错误*属性

17.5.1.1。常用参数

为了支持各种各样的用例，Popen构造函数（和便利函数）接受大量的可选参数。对于大多数典型的用例，这些参数中的很多可以安全地保留其默认值。最常见的论据是：

所有调用都需要*args*，并且应该是一个字符串或一系列程序参数。提供一系列参数通常是首选的，因为它允许模块处理任何所需的参数转义和引用（例如允许文件名中的空格）。如果传递一个字符串，则*shell*必须是True（见下文），否则字符串必须简单地命名要执行的程序而不指定任何参数。

stdin，*stdout*和*stderr*分别指定执行的程序的标准输入，标准输出和标准错误文件句柄。有效值是PIPE，DEVNULL现有文件描述符（正整数），现有文件对象和None。PIPE表示应该创建一个新的管道给孩子。 DEVNULL表示os.devnull将使用特殊文件。使用默认设置None，不会发生重定向; 孩子的文件句柄将从父类继承。此外，*stderr*可以是 STDOUT，这表示应该将stderr数据从子进程捕获到与*stdout*相同的文件句柄中。

如果指定了*编码*或*错误*，或者*universal_newlines*为true，则文本对象*stdin*，*stdout*和*stderr*将在文本模式下使用调用中指定的*编码*和*错误*或默认值打开io.TextIOWrapper。

对于*标准输入*，*输入*中的行尾字符'\n'将被转换为默认行分隔符os.linesep。对于*stdout*和*stderr*，*输出*中的所有行尾都将被转换为'\n'。有关更多信息，请参阅该构造函数io.TextIOWrapper的*newline*参数所在的类的文档None。

如果不使用文本模式，*stdin*，*stdout*和*stderr*将作为二进制流打开。不执行编码或行结束转换。

*3.6版*新增功能：增加了*编码*和*错误*参数。

> **注意：** 文件对象的newlines属性Popen.stdin， Popen.stdout并且Popen.stderr不会被Popen.communicate()方法更新。

如果*shell*是True，则指定的命令将通过shell执行。如果您主要将Python用于其在大多数系统shell中提供的增强控制流并且仍希望方便地访问其他shell功能，如shell管道，文件名通配符，环境变量扩展以及扩展~到用户的主目录。但是请注意，Python本身提供了很多贝壳般的功能实现（特别是glob， fnmatch，os.walk()，os.path.expandvars()， os.path.expanduser()，和shutil）。

*版本3.3中更改：*当*universal_newlines*是时True，类使用编码locale.getpreferredencoding(False)而不是locale.getpreferredencoding()。有

关io.TextIOWrapper此更改的更多信息，请参阅 班级。

> **注意：** 使用前阅读安全注意事项部分shell=True。

这些选项以及所有其他选项在Popen构造函数文档中有更详细的描述。

17.5.1.2。POPEN构造

该模块中的基础过程创建和管理由Popen该类处理。它提供了很大的灵活性，以便开发人员能够处理不属于便利功能范围的不常见情况。

class subprocess.Popen（*args*，*bufsize = -1*，*executable = None*，*stdin = None*，*stdout = None*，*stderr = None*，*preexec_fn = None*，*close_fds = True*，*shell = False*，*cwd = None*，*env = None*，*universal_newlines = False*，*startupinfo = None*，*creationflags = 0*，*restore_signals = True*，*start_new_session = False*，*pass_fds = ( )*，*，*encoding = None*，*errors = None* ）

在新过程中执行子程序。在POSIX上，类使用 os.execvp()类行为来执行子程序。在Windows上，该类使用Windows CreateProcess()功能。参数 Popen如下。

*参数*应该是一系列程序参数或者一个字符串。默认情况下，如果*args*是一个序列，*则要*执行的程序是*args中*的第一个项目。如果*args*是一个字符串，则解释是依赖于平台的，并在下面进行描述。查看*shell*和*可执行*参数以获取与默认行为的其他差异。除非另有说明，否则建议将*参数*作为序列传递。

在POSIX上，如果*args*是一个字符串，则将该字符串解释为要执行的程序的名称或路径。但是，只有在不向程序传递参数的情况下才能完成此操作。

> **注意：** shlex.split()在确定*参数*的正确标记时，尤其是在复杂情况下可能有用：
>
> ```
> >>> import shlex, subprocess
> >>> command_line = input()
> /bin/vikings -input eggs.txt -output "spam spam.txt" -cmd "echo '$MONEY'"
> >>> args = shlex.split(command_line)
> >>> print(args)
> ['/bin/vikings', '-input', 'eggs.txt', '-output', 'spam spam.txt', '-cmd', "echo
> >>> p = subprocess.Popen(args) # Success!
> ```
>
> 请特别注意，shell中由空白分隔的选项（例如*-input*）和参数（如*eggs.txt*）将放在单独的列表元素中，而在shell中使用时需要引用或反斜杠转义的参数（例如包含空格或上面显示的*echo*命令的文件名）是单个列表元素。

在Windows上，如果*args*是序列，则它将按照在Windows 上将*参数*序列转换为字符串中所述的方式转换为字符串。这是因为底层CreateProcess()在字符串上运行。

所述*壳*参数（默认为False）指定是否使用壳作为要执行的程序。如果*shell*是True，建议将*args*作为字符串传递，而不是作为序列传递。

在POSIX上shell=True，shell默认为/bin/sh。如果 *args*是一个字符串，则该字符串指定通过shell执行的命令。这意味着该字符串的格式必须与在shell提示符下键入时的格式完全相同。这包括，例如，在其中包含空格的引号或反斜线转义文件名。如果*args*是一个序列，则第一项指定命令字符串，并且任何附加项都将被视为shell本身的附加参数。也就是说，Popen相当于：

```
Popen(['/bin/sh', '-c', args[0], args[1], ...])
```

在Windows上shell=True，COMSPEC环境变量指定了默认的shell。您需要shell=True在Windows 上指定的唯一时间 是您希望执行的命令内置到shell中（例如**dir**或**copy**）。您不需要 shell=True运行批处理文件或基于控制台的可执行文件。

注意： 使用前阅读安全注意事项部分shell=True。

open()在创建stdin / stdout / stderr管道文件对象时，*bufsize*将作为该函数的相应参数提供：

- 0 意味着无缓冲（读写是一个系统调用，可以返回较短）
- 1意味着缓冲行（只有universal_newlines=True在文本模式下才可用）
- 任何其他积极的价值意味着使用大约那个大小的缓冲区
- 负bufsize（默认值）表示将使用io.DEFAULT_BUFFER_SIZE的系统默认值。

*在版本3.3.1中进行了更改：bufsize*现在默认为-1，默认情况下启用缓冲以匹配大多数代码所期望的行为。在Python 3.2.4和3.3.1之前的版本中，它错误地默认为0未缓冲并允许短读取。这是无意的，并不符合大多数代码预期的Python 2的行为。

该*可执行文件*参数指定的替代程序来执行。这是很少需要的。何时shell=False，*可执行文件*会将程序替换为由*args*指定的执行程序。但是，原始*参数*仍然传递给程序。大多数程序会将由*args*指定的程序视为命令名称，这可能与实际执行的程序不同。在POSIX上，*args*名称成为**ps**等实用程序中可执行文件的显示名称。如果shell=True在POSIX上，*可执行*参数指定了默认的替换shell /bin/sh。

stdin，*stdout*和*stderr分别*指定执行的程序的标准输入，标准输出和标准错误文件句柄。有效值是PIPE，DEVNULL现有文件描述符（正整数），现有文件对象和None。 PIPE 表示应该创建一个新的管道给孩子。 DEVNULL 表示os.devnull将使用特殊文件。使用默认设置None，不会发生重定向; 孩子的文件句柄将从父类继承。此外，*stderr*可以是 STDOUT，它表示应用程序的stderr数据应该被捕获到与stdout相同的文件句柄中。

如果将*preexec_fn*设置为可调用对象，则该对象将在子进程执行前被调用。（仅限POSIX）

警告： 该*preexec_fn*参数不是安全的线程的应用程序中存在使用。在调用exec之前，子进程可能会死锁。如果你必须使用它，保持它微不足道！最大限度地减少您调用的库的数量。

注意： 如果您需要修改儿童的环境，请使用*env* 参数，而不是在*preexec_fn中执行*。该*start_new_session*参数可以采用以前普遍使用的地点*preexec_fn*调用os.setsid（）的孩子。

如果*close_fds*为true，则将执行子进程之前的所有文件描述符0，1并且 2将被关闭。（仅限POSIX）。默认值因平台而异：在POSIX上始终为true。在Windows上，当*stdin* / *stdout* / *stderr*为true时，则为true None，否则为false。在Windows上，如果*close_fds*为true，那么子进程将不会继承任何句柄。请注意，在Windows上，您不能将*close_fds*设置为true，并通过设置*stdin*，*stdout*或*stderr*来重定向标准句柄。

*在版本3.2中更改：*close_fds的默认值已更改False为上述内容。

*pass_fds*是一个可选的文件描述符序列，用于在父代和子代之间保持开放。提供任何*pass_fds*强制 *close_fds*为True。（仅限POSIX）

*：在3.2版本中的新*的*pass_fds*加入参数。

如果*cwd*不是None，则在执行该子项之前，该函数将工作目录更改为 *cwd*。 *cwd*可以是一个str和 路径类似的对象。特别是， 如果可执行文件路径是相对路径，该函数将查找与*cwd相关*的可执行*文件*（或*args中*的第一项）。

*在版本3.6中更改：*cwd参数接受类似路径的对象。

如果*restore_signals*为true（缺省值），则Python设置为SIG_IGN的所有信号都将在子进程中恢复到执行前的SIG_DFL。目前这包括SIGPIPE，SIGXFZ和SIGXFSZ信号。（仅限POSIX）

*在版本3.2中进行了更改：*添加了*restore_signals*。

如果*start_new_session*为true，则将在执行子进程之前在子进程中进行setsid（）系统调用。（仅限POSIX）

*在版本3.2中进行了更改：*添加了*start_new_session*。

如果*env*不是None，它必须是一个为新进程定义环境变量的映射; 这些被用来代替继承当前进程环境的默认行为。

> **注意：** 如果指定，*env*必须提供程序执行所需的任何变量。在Windows上，为了运行并行程序集，指定的*env* **必须**包含有效的SystemRoot。

如果指定了*编码*或*错误*，则文件对象*stdin*，*stdout* 和*stderr*将在文本模式下以指定的编码和*错误*打开 ，如上面的常用已用参数中所述。如果 是*universal_newlines*True，它们将以默认编码的文本模式打开。否则，它们会以二进制流的形式打开。

*3.6版新增功能：*添加了*编码*和*错误*。

如果给出，*startupinfo*将是一个STARTUPINFO对象，它被传递给底层CreateProcess函数。*创造性标志*，如果给出，可以是CREATE_NEW_CONSOLE或 CREATE_NEW_PROCESS_GROUP。（仅限Windows）

通过with语句支持Popen对象作为上下文管理器：在退出时，关闭标准文件描述符，并等待进程。

```
with Popen(["ifconfig"], stdout=PIPE) as proc:
    log.write(proc.stdout.read())
```

*在版本3.2中更改：*添加了上下文管理器支持。

*在版本3.6中更改：*ResourceWarning如果子进程仍在运行，Popen析构函数现在会发出警告。

17.5.1.3。例外

在新程序开始执行之前，子进程中引发的异常将在父进程中重新提出。另外，异常对象将会有一个额外的属性被调用child_traceback，这是一个字符串，其中包含来自孩子视角的回溯信息。

最常见的例外是OSError。例如，尝试执行不存在的文件时会发生这种情况。应用程序应该准备OSError例外。

ValueError如果Popen使用无效参数调用A 将会被提出。

check_call()如果被调用的进程返回一个非零的返回码，check_output()将会引发CalledProcessError。

接受*超时*参数的所有函数和方法，如 在过程退出之前超时过期call()，Popen.communicate()则会引发TimeoutExpired。

在这个模块中定义的异常都从中继承SubprocessError。

*：在3.3版本中的新*的SubprocessError加入的基类。

17.5.2。安全考虑

与其他一些popen函数不同，这个实现永远不会隐式调用系统shell。这意味着所有字符（包括shell元字符）都可以安全地传递给子进程。如果通过明确调用shell，shell=True应用程序有责任确保所有空白和元字符都被正确引用以避免 shell注入 漏洞。

在使用时shell=True，shlex.quote()可以使用该函数正确地转义要用于构造shell命令的字符串中的空白和shell元字符。

17.5.3。Popen对象

Popen类的实例有以下方法：

Popen.poll()

检查子进程是否已终止。设置并返回 returncode属性。否则，退货None。

Popen.wait(*timeout=None*)

等待子进程终止。设置并返回 returncode属性。

如果进程在*超时*秒后没有终止，请引发 TimeoutExpired异常。捕获此异常并重试等待是安全的。

注意： 当使用stdout=PIPEor 时，这将会发生死锁， stderr=PIPE 并且子进程会向管道生成足够的输出，从而阻止等待OS管道缓冲区接受更多数据。使用Popen.communicate()管道时要避免这种情况。

注意： 该功能是使用忙循环（非阻塞呼叫和短暂休眠）实现的。使用asyncio模块进行异步等待：请参阅asyncio.create_subprocess_exec。

*在版本3.3中更改：*添加了*超时*。

*自3.4版弃用：*不要使用*endtime*参数。它在3.3被无意暴露，但没有记录，因为它打算私下供内部使用。改用*超时*。

Popen.**communicate**(*输入=无*, *超时=无*)

与流程进行交互：将数据发送到stdin。从stdout和stderr中读取数据，直到达到文件结尾。等待进程终止。可选的 *输入*参数应该是要发送到子进程的None数据，或者如果没有数据应该发送给子进程 。如果流以文本模式打开，则*输入*必须是字符串。否则，它必须是字节。

communicate()返回一个元组。如果在文本模式下打开流，数据将是字符串; 否则，字节。(stdout_data, stderr_data)

请注意，如果要将数据发送到进程的stdin，则需要使用创建Popen对象stdin=PIPE。同样，要获得除None结果元组以外的任何内容 ，您需要提供stdout=PIPE和/或 stderr=PIPE也可以。

如果进程在*超时*秒后没有终止， TimeoutExpired则会引发异常。捕获此异常并重试通信不会丢失任何输出。

如果超时过期，子进程不会被终止，因此为了正确地清理一个行为良好的应用程序，应该终止子进程并完成通信：

```
proc = subprocess.Popen(...)
try:
    outs, errs = proc.communicate(timeout=15)
except TimeoutExpired:
    proc.kill()
    outs, errs = proc.communicate()
```

注意： 读取的数据缓冲在内存中，所以如果数据量很大或无限，就不要使用这种方法。

*在版本3.3中更改：*添加了*超时*。

Popen.**send_signal**(*信号*)

将信号*发送*给孩子。

注意： 在Windows上，SIGTERM是别名 terminate()。可以将CTRL_C_EVENT和CTRL_BREAK_EVENT发送到使用*CREATE_NEW_PROCESS_GROUP*的*creationflags*参数开始的进程。

Popen.**terminate**()

停止孩子。在Posix操作系统上，该方法向孩子发送SIGTERM。在Windows上调用Win32 API函数TerminateProcess()来停止该子项。

Popen.kill()

杀死孩子。在Posix操作系统上，该函数向孩子发送SIGKILL。在Windows上kill()是别名terminate()。

以下属性也可用：

Popen.args

该*ARGS*因为它传递给参数Popen的程序参数的序列或者一个字符串- 。

3.3版本的新功能

Popen.stdin

如果*stdin*参数是PIPE，则该属性是可返回的可写流对象open()。如果 指定了*encoding*或*errors*参数或*universal_newlines*参数True，则该流为文本流，否则为字节流。如果*stdin* 参数不是PIPE，则该属性为None。

Popen.stdout

如果*stdout*参数是PIPE，则此属性是返回的可读流对象open()。从流读取提供了子进程的输出。如果指定了*encoding*或*errors*参数或*universal_newlines*参数True，则该流为文本流，否则为字节流。如果*stdout*参数不是 PIPE，则该属性为None。

Popen.stderr

如果*stderr*参数是PIPE，则此属性是返回的可读流对象open()。从流读取提供了子进程的错误输出。如果指定了*encoding*或*errors*参数或*universal_newlines*参数True，则该流为文本流，否则为字节流。如果*stderr*参数不是PIPE，则该属性为None。

警告: 使用communicate()而不是.stdin.write， .stdout.read或.stderr.read避免死锁由于任何其他操作系统管缓冲区填满并阻塞子进程。

Popen.pid

子进程的进程ID。

请注意，如果将*shell*参数设置为True，则这是生成的shell的进程ID。

Popen.returncode

孩子返回代码，由poll()和wait()（和间接地communicate()）设置。一个None值表示进程尚未结束。

负值-N表示孩子被信号终止 N（仅限POSIX）。

17.5.4。Windows Popen

该STARTUPINFO级和以下常量仅适用于Windows。

*类*subprocess.STARTUPINFO

部分支持Windows STARTUPINFO 结构用于Popen创建。

dwFlags
一个位域，用于确定STARTUPINFO 进程创建窗口时是否使用某些属性。

```
si = subprocess.STARTUPINFO()
si.dwFlags = subprocess.STARTF_USESTDHANDLES | subprocess.STARTF_USESHOWWINDOW
```

hStdInput
如果dwFlags指定STARTF_USESTDHANDLES ，则此属性是该进程的标准输入句柄。如果STARTF_USESTDHANDLES未指定，则标准输入的默认值是键盘缓冲区。

hStdOutput
如果dwFlags指定STARTF_USESTDHANDLES ，则此属性是该进程的标准输出句柄。否则，该属性将被忽略，标准输出的默认值是控制台窗口的缓冲区。

hStdError
如果dwFlags指定STARTF_USESTDHANDLES ，则此属性是该进程的标准错误句柄。否则，该属性将被忽略，标准错误的默认值是控制台窗口的缓冲区。

wShowWindow
如果dwFlags指定STARTF_USESHOWWINDOW ，则此属性可以是可nCmdShow 在ShowWindow 函数的参数中 指定的任何值，但SW_SHOWDEFAULT。否则，该属性将被忽略。

SW_HIDE为此属性提供。它在 Popen被调用时使用shell=True。

17.5.4.1。常量

该subprocess模块暴露下列常量。

subprocess.STD_INPUT_HANDLE
标准输入设备。最初，这是控制台输入缓冲区， CONIN$。

subprocess.STD_OUTPUT_HANDLE
标准输出设备。最初，这是活动控制台屏幕缓冲区，CONOUT$。

subprocess.STD_ERROR_HANDLE
标准的错误设备。最初，这是活动控制台屏幕缓冲区，CONOUT$。

subprocess.SW_HIDE
隐藏窗口。另一个窗口将被激活。

subprocess.STARTF_USESTDHANDLES
指定STARTUPINFO.hStdInput ， STARTUPINFO.hStdOutput和STARTUPINFO.hStdError属性包含的附加信息。

subprocess.STARTF_USESHOWWINDOW

指定该STARTUPINFO.wShowWindow属性包含附加信息。

subprocess.CREATE_NEW_CONSOLE
新进程有一个新的控制台，而不是继承其父控制台（默认）。

subprocess.CREATE_NEW_PROCESS_GROUP
一个参数指定一个新的进程组将被创建。该标志对于 在子进程上使用是必需的。 Popen creationflagsos.kill()

如果CREATE_NEW_CONSOLE被指定，该标志被忽略。

17.5.5。较旧的高级

在Python 3.5之前，这三个函数将高级API组成了子进程。您现在可以run()在很多情况下使用，但很多现有代码都会调用这些函数。

subprocess.call（*args*，*，*stdin = None*，*stdout = None*，*stderr = None*，*shell = False*，*cwd = None*，*timeout = None* ）
运行*args*描述的命令。等待命令完成，然后返回returncode属性。

这相当于：

```
run(...).returncode
```

（除了不支持*输入*和*检查*参数）

上面显示的参数仅仅是最常见的参数。完整的函数签名是大致相同的的Popen构造-该功能通过比其他所有提供的参数*超时*直接通过该接口。

注意： 请勿使用stdout=PIPE或stderr=PIPE使用此功能。子进程会阻塞它是否因管道未被读取而产生足够的输出到管道来填充OS管道缓冲区。

*在版本3.3中更改：*添加了*超时*。

subprocess.check_call（*args*，*，*stdin = None*，*stdout = None*，*stderr = None*，*shell = False*，*cwd = None*，*timeout = None* ）
带参数运行命令。等待命令完成。如果返回码为零，则返回，否则提升CalledProcessError。该 CalledProcessError对象将在returncode属性中具有返回码。

这相当于：

```
run(..., check=True)
```

（除了不支持*输入*参数）

上面显示的参数仅仅是最常见的参数。完整的函数签名是大致相同的的Popen构造-该功能通过比其他所有提供的参数*超时*直接通过该接口。

注意： 请勿使用stdout=PIPE或stderr=PIPE使用此功能。子进程会阻塞它是否因管道未被读取而产生足够的输出到管道来填充OS管道缓冲区。

*在版本3.3中更改：*添加了*超时*。

subprocess.check_output（*args*，*，*stdin = None*，*stderr = None*，*shell = False*，*cwd = None*，*encoding = None*，*errors = None*，*universal_newlines = False*，*timeout = None*）

用参数运行命令并返回其输出。

如果返回代码不为零，则会引发一次CalledProcessError。该 CalledProcessError对象将在returncode属性中具有返回码，并在 属性中包含任何输出 output。

这相当于：

```
run(..., check=True, stdout=PIPE).stdout
```

上面显示的参数仅仅是最常见的参数。全功能签名与大部分相同run()- 大多数参数直接传递到该界面。但是，input=None不支持显式传递继承父级的标准输入文件句柄。

默认情况下，此函数将以编码字节的形式返回数据。输出数据的实际编码可能取决于被调用的命令，因此解码到文本通常需要在应用程序级别进行处理。

此行为可以通过设置覆盖*universal_newlines*到 True如在上述常用的参数。

要在结果中捕获标准错误，请使用 stderr=subprocess.STDOUT：

```
>>> subprocess.check_output(
...     "ls non_existent_file; exit 0",
...     stderr=subprocess.STDOUT,
...     shell=True)
'ls: non_existent_file: No such file or directory\n'
```

版本3.1中的新功能。

*在版本3.3中更改：*添加了*超时*。

*在版本3.4中更改：*添加了对*输入*关键字参数的支持。

*在版本3.6中更改：*添加了*编码*和*错误*。详情请参阅run()。

17.5.6。用subprocess模块替换旧功能

在本节中，“a变成b”表示b可以用作a的替换。

注意： 如果执行的程序无法找到，本节中的所有“a”功能都将无效（或多或少）；OSError 代替“b” 代替。

另外，如果请求的操作产生非零返回码，则替换使用 check_output() 将失败CalledProcessError。输出仍然可用作output引发异常的 属性。

在以下示例中，我们假设相关功能已从subprocess模块导入。

17.5.6.1。替换/ bin / sh shell重新引号

```
output=`mycmd myarg`
```

变为：

```
output = check_output(["mycmd", "myarg"])
```

17.5.6.2。替换shell管道

```
output=`dmesg | grep hda`
```

变为：

```
p1 = Popen(["dmesg"], stdout=PIPE)
p2 = Popen(["grep", "hda"], stdin=p1.stdout, stdout=PIPE)
p1.stdout.close()  # Allow p1 to receive a SIGPIPE if p2 exits.
output = p2.communicate()[0]
```

启动p2后，p1.stdout.close（ ）调用非常重要，以便p1在p1之前退出时接收SIGPIPE。

或者，对于可信输入，shell自己的管线支持仍可以直接使用：

```
output=`dmesg | grep hda`
```

变为：

```
output=check_output("dmesg | grep hda", shell=True)
```

17.5.6.3。替换 os.system()

```
sts = os.system("mycmd" + " myarg")
# becomes
sts = call("mycmd" + " myarg", shell=True)
```

笔记：

- 通过shell调用程序通常不是必需的。

更现实的例子看起来像这样：

```
try:
    retcode = call("mycmd" + " myarg", shell=True)
    if retcode < 0:
        print("Child was terminated by signal", -retcode, file=sys.stderr)
    else:
        print("Child returned", retcode, file=sys.stderr)
```

```
except OSError as e:
    print("Execution failed:", e, file=sys.stderr)
```

17.5.6.4。取代os.spawn家庭

P_NOWAIT示例：

```
pid = os.spawnlp(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg")
==>
pid = Popen(["/bin/mycmd", "myarg"]).pid
```

P_WAIT示例：

```
retcode = os.spawnlp(os.P_WAIT, "/bin/mycmd", "mycmd", "myarg")
==>
retcode = call(["/bin/mycmd", "myarg"])
```

矢量例子：

```
os.spawnvp(os.P_NOWAIT, path, args)
==>
Popen([path] + args[1:])
```

环境示例：

```
os.spawnlpe(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg", env)
==>
Popen(["/bin/mycmd", "myarg"], env={"PATH": "/usr/bin"})
```

17.5.6.5。更换os.popen()，os.popen2()，os.popen3()

```
(child_stdin, child_stdout) = os.popen2(cmd, mode, bufsize)
==>
p = Popen(cmd, shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdin, child_stdout) = (p.stdin, p.stdout)
```

```
(child_stdin,
 child_stdout,
 child_stderr) = os.popen3(cmd, mode, bufsize)
==>
p = Popen(cmd, shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, stderr=PIPE, close_fds=True)
(child_stdin,
 child_stdout,
 child_stderr) = (p.stdin, p.stdout, p.stderr)
```

```
(child_stdin, child_stdout_and_stderr) = os.popen4(cmd, mode, bufsize)
==>
p = Popen(cmd, shell=True, bufsize=bufsize,
```

```
          stdin=PIPE, stdout=PIPE, stderr=STDOUT, close_fds=True)
(child_stdin, child_stdout_and_stderr) = (p.stdin, p.stdout)
```

返回码处理转换如下：

```
pipe = os.popen(cmd, 'w')
...
rc = pipe.close()
if rc is not None and rc >> 8:
    print("There were some errors")
==>
process = Popen(cmd, stdin=PIPE)
...
process.stdin.close()
if process.wait() != 0:
    print("There were some errors")
```

17.5.6.6。从popen2模块中替换功能

注意： 如果popen2函数的cmd参数是一个字符串，则该命令通过/ bin / sh执行。如果它是一个列表，则直接执行该命令。

```
(child_stdout, child_stdin) = popen2.popen2("somestring", bufsize, mode)
==>
p = Popen("somestring", shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdout, child_stdin) = (p.stdout, p.stdin)
```

```
(child_stdout, child_stdin) = popen2.popen2(["mycmd", "myarg"], bufsize, mode)
==>
p = Popen(["mycmd", "myarg"], bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdout, child_stdin) = (p.stdout, p.stdin)
```

popen2.Popen3和popen2.Popen4基本工作原理为 subprocess.Popen ，不同的是：

- Popen 如果执行失败会引发异常。
- 该*capturestderr*参数被替换为*标准错误*的说法。
- stdin=PIPE并且stdout=PIPE必须指定。
- popen2关闭默认情况下，所有文件描述符，但你必须指定 close_fds=True使用Popen ，以保证在所有平台上的或过去的Python版本此行为。

17.5.7。传统的Shell调用函数

该模块还提供了来自2.x commands模块的以下遗留功能 。这些操作隐式地调用系统外壳，上述关于安全性和异常处理一致性的保证都不适用于这些功能。

subprocess.**getstatusoutput**(*cmd*)

在shell中返回执行*cmd*。(exitcode, output)

在shell中执行字符串*cmd*Popen.check_output()并返回2元组。使用区域设置编码;有关更多详细信息，请参阅常用参数注释。(exitcode, output)

尾随的换行符从输出中剥离。该命令的退出代码可以解释为子进程的返回代码。例：

```
>>> subprocess.getstatusoutput('ls /bin/ls')
(0, '/bin/ls')
>>> subprocess.getstatusoutput('cat /bin/junk')
(1, 'cat: /bin/junk: No such file or directory')
>>> subprocess.getstatusoutput('/bin/junk')
(127, 'sh: /bin/junk: not found')
>>> subprocess.getstatusoutput('/bin/kill $$')
(-15, '')
```

可用性：POSIX和Windows

*在版本3.3.4中进行了更改：*添加了Windows支持。

该函数现在返回（exitcode，output）而不是（status，output），就像在Python 3.3.3和更早的版本中一样。看WEXITSTATUS()。

subprocess.**getoutput**(*cmd*)

在shell中返回执行*cmd的*输出（stdout和stderr）。

像getstatusoutput()，除了退出状态被忽略，返回值是一个包含命令输出的字符串。例：

```
>>> subprocess.getoutput('ls /bin/ls')
'/bin/ls'
```

可用性：POSIX和Windows

*版本3.3.4中已更改：已*添加Windows支持

17.5.8。笔记

17.5.8.1。在Windows上将参数序列转换为字符串

在Windows上，*参数*序列被转换为可以使用以下规则解析的字符串（它对应于MS C运行时使用的规则）：

1. 参数是由空白分隔的，它是一个空格或一个制表符。
2. 无论内部包含空格，用双引号括起来的字符串都被解释为单个参数。引用的字符串可以嵌入参数中。
3. 前面加一个反斜杠的双引号被解释为文字双引号。
4. 反斜杠从字面上解释，除非它们立即在双引号之前。
5. 如果反斜杠立即在双引号之前，则每对反斜杠都将被解释为文字反斜杠。如果反斜杠的数量是奇数，则最后一个反斜杠将转义下一个双引号，如规则3所述。

也可以看看:

shlex

提供解析和转义命令行功能的模块。

17.6。 sched- 事件调度程序

源代码： Lib / sched.py

该sched模块定义了一个实现通用事件调度器的类：

class sched. scheduler (*timefunc = time.monotonic* , *delayfunc = time.sleep*)

在scheduler类定义一个通用的接口，调度事件。它需要两个功能来实际处理“外部世界” - *timefunc* 应该可以在没有参数的情况下进行调用，并返回一个数字（任何单位的“时间”）。如果time.monotonic不可用，则*timefunc*默认为time.time。该*delayfunc*功能应该是可调用的一个参数，与输出兼容*timefunc* ，并应延迟许多时间单位。在每个事件运行后，*delayfunc*也将被参数调用，0以允许其他线程有机会在多线程应用程序中运行。

版本3.3中更改： *timefunc*和*delayfunc*参数是可选的。

版本3.3中更改： scheduler可以安全地在多线程环境中使用类。

例：

```
>>> import sched, time
>>> s = sched.scheduler(time.time, time.sleep)
>>> def print_time(a='default'):
...     print("From print_time", time.time(), a)
...
>>> def print_some_times():
...     print(time.time())
...     s.enter(10, 1, print_time)
...     s.enter(5, 2, print_time, argument=('positional',))
...     s.enter(5, 1, print_time, kwargs={'a': 'keyword'})
...     s.run()
...     print(time.time())
...
>>> print_some_times()
930343690.257
From print_time 930343695.274 positional
From print_time 930343695.275 keyword
From print_time 930343700.273 default
930343700.276
```

17.6.1。计划程序对象

scheduler 实例具有以下方法和属性：

scheduler. enterabs (*时间* , *优先级* , *动作* , *参数=（ ）* , *kwargs = {}*)

安排新的活动。在*时间*参数应该是一个数字类型与返回值兼容*timefunc*传递给构造函数。计划在同一*时间的*活动将按其*优先*顺序执行 。较低的数字表示较高的优先级。

执行事件意味着执行。 *参数*是一个持有*动作*位置参数的序列。 *kwargs*是持有该参数的字典*行动*。 action(*argument, **kwargs)

返回值是一个可用于事件后续取消的事件（请参阅参考资料cancel()）。

*版本3.3中更改：参数*参数是可选的。

*3.3版新增：kwargs*参数被添加。

scheduler.enter(*delay*，*priority*，*action*，*argument = ()*，*kwargs = {}*)

安排*延迟*更多时间单位的事件。除了相对时间以外，其他参数，效果和返回值与那些相同enterabs()。

*版本3.3中更改：参数*参数是可选的。

*3.3版新增：kwargs*参数被添加。

scheduler.cancel(*事件*)

从队列中删除事件。如果*事件*不是当前队列中的事件，则此方法将引发一个ValueError。

scheduler.empty()

如果事件队列为空，则返回true。

scheduler.run(*blocking = True*)

运行所有计划的事件。此方法将等待（使用delayfunc()传递给构造函数的函数）进行下一个事件，然后执行此操作，直到没有更多计划事件。

如果*阻塞*为假，则执行预定的事件，因为过期最快（如果有），然后返回调度程序中下一个预定呼叫的最后期限（如果有）。

无论是*动作*还是*delayfunc*可引发异常。无论哪种情况，调度程序都将保持一致的状态并传播异常。如果通过*操作*引发异常，则将来的呼叫将不会尝试该事件run()。

如果一系列事件的运行时间比下一个事件之前的可用时间长，那么调度器将会落后。没有事件会被丢弃; 调用代码负责取消不再相关的事件。

*新增3.3版本：阻止*参数。

scheduler.queue

只读属性按照它们将运行的顺序返回即将发生的事件列表。每个事件都显示为具有以下字段的命名元组：时间，优先级，动作，参数，kwargs。

17.7。queue-一个同步队列类

源代码： Lib / queue.py

该queue模块实现了多生产者，多用户队列。当信息必须在多个线程之间安全地交换时，它在线程编程中特别有用。该Queue模块中的类实现了所有必需的锁定语义。这取决于Python中线程支持的可用性; 看threading 模块。

该模块实现三种类型的队列，它们仅在检索条目的顺序上有所不同。在FIFO 队列中，添加的第一个任务是第一个检索的。在 LIFO队列中，最近添加的条目是第一个检索到的（像堆栈一样操作）。使用优先级队列，条目保持排序（使用heapq模块），并且首先检索最低值的条目。

在内部，模块使用锁来暂时阻止竞争线程; 然而，它不是为了处理线程内的重入而设计的。

该queue模块定义了以下类和例外：

class queue. Queue (*maxsize = 0*)

FIFO队列的构造器。 *maxsize*是一个整数，用于设置可以放入队列中的项目数的上限。一旦达到此大小，插入将会阻塞，直到消耗队列项目。如果 *maxsize*小于或等于零，则队列大小是无限的。

class queue. LifoQueue (*maxsize = 0*)

LIFO队列的构造器。 *maxsize*是一个整数，用于设置可以放入队列中的项目数的上限。一旦达到此大小，插入将会阻塞，直到消耗队列项目。如果 *maxsize*小于或等于零，则队列大小是无限的。

class queue. PriorityQueue (*maxsize = 0*)

优先队列的构造函数。 *maxsize*是一个整数，用于设置可以放入队列中的项目数的上限。一旦达到此大小，插入将会阻塞，直到消耗队列项目。如果 *maxsize*小于或等于零，则队列大小是无限的。

最低值的条目首先被检索（最低值的条目是返回的条目sorted(list(entries))[0] ）。条目的典型模式是以下形式的元组：。 (priority_number, data)

*异常*queue. Empty

在空对象上调用非阻塞get() （或 get_nowait() ）时引发异常Queue。

*异常*queue. Full

在对象已满时调用非阻塞put() （或 put_nowait() ）时引发异常Queue。

17.7.1。队列对象

队列对象（ Queue ，LifoQueue或PriorityQueue ）提供下面描述的公共方法。

Queue. qsize ( )

返回队列的近似大小。请注意，qsize（）> 0并不保证后续的get（）不会被阻塞，也不会保证put（）不会阻塞qsize（）<maxsize。

Queue.**empty（）**

True如果队列为空False则返回，否则返回。如果empty（）返回True它并不能保证后续的put（）调用不会被阻塞。同样，如果empty（）返回False它并不能保证后续调用get（）不会被阻塞。

Queue.**full（）**

True如果队列已满False则返回，否则返回。如果full（）返回True它并不能保证后续调用get（）不会被阻塞。同样，如果full（）返回False它并不能保证后续调用put（）不会被阻塞。

Queue.**put（*item*，*block* = *True*，*timeout* = *None*）**

将*项目*放入队列中。如果可选参数*块*为真并且*超时*为 None（默认），则在需要时禁止，直到有空闲插槽可用。如果 *超时*是一个正数，它会阻止至多*超时*秒数，并Full在该时间内没有空闲插槽时引发异常。否则（*块*为假），如果空闲插槽立即可用，则在队列中放置一个项目，否则引发Full异常（在这种情况下*超时*被忽略）。

Queue.**put_nowait（*item*）**

相当于。put(item, False)

Queue.**get（*block* = *True*，*timeout* = *None*）**

从队列中移除并返回一个项目。如果可选的参数*块*为true并且 *超时*为None（默认），则在必要时阻塞，直到项目可用。如果*超时时间*为正数，则最多会阻止*超时*秒数，Empty如果在该时间内没有可用项目，则会引发异常。否则（*block*为false），如果一个立即可用，则返回一个项目，否则引发Empty异常（在这种情况下*超时*被忽略）。

Queue.**get_nowait（）**

相当于get(False)。

提供了两种方法来支持跟踪入队任务是否已完全由守护进程消费者线程处理。

Queue.**task_done（）**

表明以前排队的任务已完成。由队列消费者线程使用。对于每个get()用于获取任务的对象，后续调用会 task_done()告知队列该任务的处理已完成。

如果a join()当前处于阻塞状态，则在所有项目都处理完毕后（即task_done()接收到已put()进入队列的每个项目的呼叫），它将恢复。

提出了一个ValueError好象叫更多的时间比中放入队列中的项目。

Queue.**join（）**

阻塞，直到队列中的所有项目都被获取并处理。

每当将项目添加到队列中时，未完成任务的数量就会增加。无论何时消费者线程调用task_done()以指示该项目已被检索并且所有工作都已完成，计数就会减少。当未完成任务的计数降至零时，join()取消阻止。

如何等待入队任务完成的示例：

```
def worker():
    while True:
        item = q.get()
        if item is None:
            break
        do_work(item)
        q.task_done()

q = queue.Queue()
threads = []
for i in range(num_worker_threads):
    t = threading.Thread(target=worker)
    t.start()
    threads.append(t)

for item in source():
    q.put(item)

# block until all tasks are done
q.join()

# stop workers
for i in range(num_worker_threads):
    q.put(None)
for t in threads:
    t.join()
```

也可以看看:

类 multiprocessing.Queue
用于多处理（而不是多线程）上下文中的队列类。

collections.deque是具有快速原子append()和 popleft()无需锁定的操作的无界队列的替代实现。

17.8。 dummy_threading- 替换threading模块

源代码： Lib / dummy_threading.py

该模块为模块提供重复的接口threading。当_thread模块没有在平台上提供时，它意味着被导入。

建议的用法是：

```
try:
    import threading
except ImportError:
    import dummy_threading as threading
```

当创建的线程阻塞等待另一个线程创建时，请注意不要使用此模块。这通常会在阻止I / O时发生。

17.9。 _thread- 低级线程

该模块提供了用于处理多个线程（也称为*轻量级进程*或*任务*）的低级原语- 多个控制线程共享其全局数据空间。为了同步，提供了简单的锁（也称为*互斥锁*或*二进制信号量*）。该threading模块提供了构建在该模块之上的更易于使用和更高级的线程化API。

该模块是可选的。它在Windows，Linux，SGI IRIX，Solaris 2.x以及具有POSIX线程（又名“pthread”）实现的系统上受支持。对于缺少_thread模块的系统，该 _dummy_thread模块可用。它复制了该模块的接口，可以用作插入式替换。

它定义了以下常量和函数：

*异常*_thread.error
引发线程特定的错误。

*在版本3.3中更改：*这是现在内置的同义词RuntimeError。

_thread.LockType
这是锁定对象的类型。

_thread.start_new_thread(*function*，*args*[，*kwargs*])
开始一个新的线程并返回它的标识符。该线程使用参数列表*args*（它必须是一个元组）执行函数 *函数*。可选的 *kwargs*参数指定关键字参数的字典。当函数返回时，该线程静静地退出。当函数以未处理的异常终止时，将打印一个堆栈跟踪，然后该线程退出（但其他线程继续运行）。

_thread.interrupt_main()
KeyboardInterrupt在主线程中引发异常。一个子线程可以使用这个函数来中断主线程。

_thread.exit()
引发SystemExit异常。当未被捕获时，这将导致线程无声地退出。

_thread.allocate_lock()
返回一个新的锁定对象。下面描述锁的方法。锁最初是解锁的。

_thread.get_ident()
返回当前线程的'线程标识符'。这是一个非零整数。它的价值没有直接的意义; 它的目的是作为一个神奇的cookie用于例如索引线程特定数据的字典。当线程退出并创建另一个线程时，线程标识符可能会被回收。

_thread.stack_size([*size*])
返回创建新线程时使用的线程堆栈大小。可选的 *size*参数指定要用于随后创建的线程的堆栈大小，并且必须为0（使用平台或配置的默认值）或至少32,768（32 KiB）的正整数值。如果未指定*大小*，则使用0。如果不支持更改线程堆栈大小，RuntimeError则会引发a。如果指定的堆栈大小无效，aValueError并且堆栈大小未修改。32 KiB是目前支持的最小堆栈大小值，以保证解释器本身具有足够的堆栈空间。请注意，某些平台可能对堆栈大小的值

有特别的限制，例如要求最小堆栈大小> 32 KiB或需要以系统内存页大小的倍数进行分配 - 有关更多信息，请参阅平台文档（4 KiB页是常见的;对于堆栈大小，使用4096的倍数是没有更具体信息的建议方法）。可用性：Windows，带有POSIX线程的系统。

_thread.TIMEOUT_MAX

*超时*参数 允许的最大值Lock.acquire()。指定超过此值的超时将引发一次OverflowError。

3.2版本中的新功能

锁对象具有以下方法：

lock.acquire（*waitflag = 1*，*timeout = -1* ）

如果没有任何可选的参数，这个方法无条件地获取锁，如果需要的话等待，直到它被另一个线程释放（一次只有一个线程可以获得一个锁 - 这就是它们存在的原因）。

如果存在整数*waitflag*参数，则操作取决于其值：如果为零，则只有在可以立即获取而不等待的情况下才会获取锁定，而如果为非零，则锁定将如上所述无条件获取。

如果浮点*超时*参数存在且为肯定的，则它指定返回之前的最长等待时间（以秒为单位）。负 *超时*参数指定无限等待。如果*waitflag*为零，则不能指定*超时*。

返回值True是否成功获取锁定，False如果不成功 。

*改变在3.2版本：*该*超时*参数是新的。

*在版本3.2中更改：*锁定采集现在可以被POSIX上的信号中断。

lock.release（ ）

释放锁定。锁定必须早已获得，但不一定是由同一个线程获取。

lock.locked（ ）

返回锁的状态：True是否已经被某个线程获取， False如果没有。

除了这些方法之外，还可以通过with语句使用锁对象 ，例如：

```
import _thread

a_lock = _thread.allocate_lock()

with a_lock:
    print("a_lock is locked while this executes")
```

注意事项：

- 线程奇怪地与中断交互：KeyboardInterrupt 异常将被任意线程接收。（当signal 模块可用时，中断总是进入主线程。）
- 调用sys.exit()或引发SystemExit异常等同于调用_thread.exit()。
- 无法acquire()在锁上中断方法 - 在 KeyboardInterrupt获取锁之后会发生异常。
- 当主线程退出时，系统定义其他线程是否存活。在大多数系统中，它们被杀死而没有执行try... finally子句或执行对象析构函数。

- 当主线程退出时，它不会执行任何通常的清理（除了`try`... `finally`子句被尊重），并且标准I / O文件不会被刷新。

17.10。 _dummy_thread- 替换_thread模块

源代码： Lib / _dummy_thread.py

该模块为模块提供重复的接口_thread。当_thread模块没有在平台上提供时，它意味着被导入。

建议的用法是：

```
try:
    import _thread
except ImportError:
    import _dummy_thread as _thread
```

当创建的线程阻塞等待另一个线程创建时，请注意不要使用此模块。这通常会在阻止I / O时发生。

18.进程间通信和网络

本章介绍的模块提供了不同进程进行通信的机制。

某些模块仅适用于同一台计算机上的两个进程，例如 `signal`和`mmap`。其他模块支持两个或多个进程可用于跨计算机进行通信的网络协议。

本章描述的模块列表是：

18.1。 socket- 低级网络接口

源代码： Lib / socket.py

该模块提供对BSD *套接字*接口的访问。它适用于所有现代Unix系统，Windows，MacOS以及其他平台。

> **注意:** 某些行为可能与平台有关，因为调用操作系统套接字API。

Python接口是套接字的Unix系统调用和库接口直接转换为Python的面向对象风格的socket()函数：该函数返回一个*套接字对象*，其方法实现各种套接字系统调用。参数类型稍高级别比在C接口：与read()和write()上Python文件操作，缓冲区分配上接收操作是自动的，并且缓冲长度是发送操作隐式的。

> **也可以看看:**
>
> **模** socketserver
> 简化写入网络服务器的类。
>
> **模** ssl
> 套接字对象的TLS / SSL封装器。

18.1.1。套接字系列

根据系统和构建选项，该模块支持各种插座系列。

特定套接字对象所需的地址格式将根据套接字对象创建时指定的地址族自动选择。套接字地址表示如下：

- AF_UNIX绑定到文件系统节点的套接字的地址用字符串表示，使用文件系统编码和'surrogateescape'错误处理程序（请参阅**PEP 383**）。Linux抽象名称空间中的地址作为类似字节的对象返回，其中包含初始空字节; 请注意，此名称空间中的套接字可以与正常的文件系统套接字进行通信，因此计划在Linux上运行的程序可能需要处理这两种类型的地址。将字符串或字节类型的对象作为参数传递时，可以用于任何类型的地址。

  *版本3.3中更改：*以前，AF_UNIX套接字路径被假定为使用UTF-8编码。

  *在版本3.5中更改：*现在可以接受可写入类似字节的对象。

- 一对用于地址族，其中*host*是表示Internet域表示法中的主机名的字符串，例如像IPv4地址，而*port*是整数。(host, port)AF_INET'daring.cwi.nl''100.50.200.5'

- 对于AF_INET6地址族，使用四元组，其中*flowinfo*和*scopeid*表示C中的成员 和成员。对于 模块方法，为了向后兼容，可以省略*flowinfo*和*scopeid*。但是，请注意，省略*scopeid*会导致操作范围内的IPv6地址时出现问题。(host, port, flowinfo, scopeid)sin6_flowinfosin6_scope_idstruct sockaddr_in6socket

- AF_NETLINK套接字表示成对。(pid, groups)

- 使用AF_TIPC 地址系列可以使用仅支持TIPC的Linux 。TIPC是一种开放的，基于非IP的网络协议，设计用于集群计算机环境。地址由一个元组表示，并且这些字段取决于地址类型。一般的元组形式是 ，其中：(addr_type, v1, v2, v3 [, scope])

    - *ADDR_TYPE*是一个TIPC_ADDR_NAMESEQ，TIPC_ADDR_NAME或TIPC_ADDR_ID。

    - *范围*是一个TIPC_ZONE_SCOPE，TIPC_CLUSTER_SCOPE和 TIPC_NODE_SCOPE。

    - 如果*addr_type*是TIPC_ADDR_NAME，则*v1*是服务器类型，*v2*是端口标识符，并且*v3*应该是0。

      如果*addr_type*是TIPC_ADDR_NAMESEQ，则*v1*是服务器类型，*v2* 是较低的端口号，而*v3*是较高的端口号。

      如果*addr_type*是TIPC_ADDR_ID，则*v1*是节点，*v2*是参考，并且*v3*应该设置为0。

- 元组用于地址族，其中*interface*是表示网络接口名称的字符串 。网络接口名称可用于接收来自该系列的所有网络接口的数据包。(interface,)AF_CAN’can0’’’

- 字符串或元组用于家庭 协议。该字符串是使用动态分配的ID的内核控件的名称。如果已知内核控制的ID和单元号，或者使用了注册的ID，则可以使用元组。(id, unit)SYSPROTO_CONTROLPF_SYSTEM

  3.3版本的新功能

- AF_BLUETOOTH 支持以下协议和地址格式：

    - BTPROTO_L2CAP接受其中是蓝牙地址作为字符串和是整数。(bdaddr, psm)bdaddrpsm

    - BTPROTO_RFCOMM 接受其中 是蓝牙地址作为字符串和是整数。(bdaddr, channel)bdaddrchannel

    - BTPROTO_HCI接受(device_id,)其中device_id是整数或与接口的蓝牙地址的字符串。（这取决于您的操作系统; NetBSD和DragonFlyBSD需要一个蓝牙地址，而其他所有内容都需要一个整数。）

      *在版本3.2中进行了更改：*添加了NetBSD和DragonFlyBSD支持。

    - BTPROTO_SCO以字符串格式接受包含蓝牙地址的 对象bdaddr在哪里。（例如）FreeBSD下不支持该协议。bdaddrbytesb’12:23:34:45:56:67’

- AF_ALG是一个基于Linux的内核密码学套接字接口。算法套接字配置有2到4个元素的元组，其中：(type, name [, feat [, mask]])

    - *类型*是算法类型为字符串，例如aead，hash， skcipher或rng。
    - *名称*是该算法的名称和操作模式为字符串，例如 sha256，hmac(sha256)，cbc(aes)或drbg_nopr_ctr_aes256。
    - *专长*和*面具*是无符号的32位整数。

  可用性Linux 2.6.38，某些算法类型需要更新的内核。

  3.6版本中的新功能。

- 某些其他地址系列（AF_PACKET，AF_CAN）支持特定的表示。

对于IPv4地址，接受两种特殊形式而不是主机地址：空字符串表示INADDR_ANY，字符串'<broadcast>'表示INADDR_BROADCAST。这种行为与IPv6不兼容，因此，如果您打算在您的Python程序中支持IPv6，您可能需要避免这些行为。

如果您在IPv4 / v6套接字地址的*主机*部分中使用主机名，程序可能会显示一个非确定性行为，因为Python使用DNS解析返回的第一个地址。根据DNS解析和/或主机配置的结果，套接字地址将以不同的方式解析为实际的IPv4 / v6地址。对于确定性行为，在*主机*部分使用数字地址。

所有错误都会引发异常 可以引发无效参数类型和内存不足情况的正常例外; 从Python 3.3开始，与套接字或地址语义相关的错误引发OSError或其子类之一（它们用于引发socket.error）。

通过支持非阻塞模式setblocking()。通过支持基于超时的这种推广 settimeout()。

18.1.2。模块内容

该模块socket导出以下元素。

18.1.2.1。例外

*异常*socket.error
不推荐使用的别名OSError。

*版本3.3中更改：*以下**PEP 3151**，这个班被**取而代之**OSError。

*异常*socket.herror
OSError此异常的一个子类是针对与地址有关的错误引发的，即针对在POSIX C API 中使用*h_errno*的函数（包括gethostbyname_ex()和）gethostbyaddr()。随附的值是表示由库调用返回的错误的对。 *h_errno*是一个数字值，而 *string*表示由C函数返回 的*h_errno*的描述。(h_errno, string) hstrerror()

*在3.3版中改变了：*这个类被做成了一个子类OSError。

*异常*socket.gaierror
OSError这个异常的一个子类是由getaddrinfo()与地址有关的错误引发的getnameinfo()。随附的值是表示由库调用返回的错误的对。 *字符串*表示由C函数返回的*错误*描述 。数字*错误*值将与本模块中定义的其中一个常量相匹配。(error, string) gai_strerror() EAI_*

*在3.3版中改变了：*这个类被做成了一个子类OSError。

*异常*socket.timeout
OSError 此类异常的一个子类是在通过先前调用 settimeout() （或隐式通过setdefaulttimeout()）启用超时的套接字上发生超时时引发的 。随附的值是一个字符串，其值现在总是“超时”。

*在3.3版中改变了：*这个类被做成了一个子类OSError。

18.1.2.2。常量

AF_ *和SOCK_ *常量现在AddressFamily和 SocketKind IntEnum集合。

3.4版新增功能

socket.AF_UNIX
socket.AF_INET
socket.AF_INET6

这些常量表示用于第一个参数的地址（和协议）族socket()。如果AF_UNIX常量没有被定义，那么这个协议是不受支持的。取决于系统，可能有更多的常量。

socket.SOCK_STREAM
socket.SOCK_DGRAM
socket.SOCK_RAW
socket.SOCK_RDM
socket.SOCK_SEQPACKET

这些常量表示用于第二个参数的套接字类型 socket()。取决于系统，可能有更多的常量。（仅SOCK_STREAM和SOCK_DGRAM似乎是普遍有用的。）

socket.SOCK_CLOEXEC
socket.SOCK_NONBLOCK

这两个常量（如果已定义）可以与套接字类型组合，并允许您以原子方式设置某些标志（从而避免可能的竞争条件和需要单独调用）。

> **也可以看看：** 安全文件描述符处理 以获得更全面的解释。

可用性：Linux> = 2.6.27。

3.2版本中的新功能

SO_*
socket.SOMAXCONN
MSG_*
SOL_*
SCM_*
IPPROTO_*
IPPORT_*
INADDR_*
IP_*
IPV6_*
EAI_*
AI_*
NI_*
TCP_*

这些形式的许多常量，在套接字和/或IP协议的Unix文档中记录，也在套接字模块中定义。它们通常用于 socket对象的方法setsockopt()和参数中getsockopt()。在大多数情况下，只有那些在Unix头文件中定义的符号才被定义;对于几个符号，提供了默认值。

改变在3.6版： SO_DOMAIN，SO_PROTOCOL，SO_PEERSEC，SO_PASSSEC，TCP_USER_TIMEOUT，TCP_CONGESTION添加。

*在版本3.6.5中更改：*在Windows上，如果运行时Windows支持TCP_FASTOPEN，TCP_KEEPCNT则会出现。

socket.AF_CAN
socket.PF_CAN

SOL_CAN_*
CAN_*

Linux文档中记录的许多这些形式的常量也在套接字模块中定义。

可用性：Linux> = 2.6.25。

3.3版本的新功能

socket.CAN_BCM
CAN_BCM_*

CAN协议系列中的CAN_BCM是广播管理器（BCM）协议。在Linux文档中记录的广播管理器常量也在套接字模块中定义。

可用性：Linux> = 2.6.25。

3.4版新增功能

socket.CAN_RAW_FD_FRAMES

在CAN_RAW插座中启用CAN FD支持。这是默认禁用的。这允许您的应用程序发送CAN和CAN FD帧; 但是，从插座读取数据时，您必须同时接受CAN和CAN FD帧。

这个常量记录在Linux文档中。

可用性：Linux> = 3.6。

3.5版本中的新功能。

socket.AF_RDS
socket.PF_RDS
socket.SOL_RDS
RDS_*

Linux文档中记录的许多这些形式的常量也在套接字模块中定义。

可用性：Linux> = 2.6.30。

3.3版本的新功能

socket.SIO_RCVALL
socket.SIO_KEEPALIVE_VALS
socket.SIO_LOOPBACK_FAST_PATH
RCVALL_*

Windows的常量WSAIoctl（）。常量用作ioctl()套接字对象方法的参数 。

*在版本3.6中更改：*SIO_LOOPBACK_FAST_PATH已添加。

TIPC_*

TIPC相关的常量，与C socket API输出的相匹配。有关更多信息，请参阅TIPC文档。

socket.AF_ALG
socket.SOL_ALG
ALG_*

Linux核心密码学常量。

可用性：Linux> = 2.6.38。

3.6版本中的新功能。

socket.AF_LINK
可用性：BSD，OSX。

3.4版新增功能

socket.has_ipv6
此常数包含一个布尔值，表示此平台是否支持IPv6。

socket.BDADDR_ANY
socket.BDADDR_LOCAL
这些是包含特殊含义的蓝牙地址的字符串常量。例如，BDADDR_ANY在指定绑定套接字时可用于指示任何地址 BTPROTO_RFCOMM。

socket.HCI_FILTER
socket.HCI_TIME_STAMP
socket.HCI_DATA_DIR
用于 BTPROTO_HCI 。 HCI_FILTER 不适用于 NetBSD 或 DragonFlyBSD 。 HCI_TIME_STAMP 并 HCI_DATA_DIR没有直接提供其FreeBSD，NetBSD的，或DragonFlyBSD。

18.1.2.3。函数

18.1.2.3.1。创建插座

以下函数都创建套接字对象。

socket.socket（*family = AF_INET*，*type = SOCK_STREAM*，*proto = 0*，*fileno = None* ）
使用给定的地址系列，套接字类型和协议号创建一个新套接字。地址家庭应该是AF_INET（默认），，，AF_INET6 或。套接字类型应该是（默认），或者可能是其他 常量之一。该协议数通常是零，并且可以省略或在其中的地址是家庭的情况下的协议应该是一个或。如果指定了*fileno*，则其他参数将被忽略，导致带有指定文件描述符的套接字返回。与*fileno*不同，它将返回相同的套接字而不是重复的。这可能有助于关闭使用分离的套接字 。AF_UNIX AF_CAN AF_RDS SOCK_STREAM SOCK_DGRAM SOCK_RAW SOCK_AF_CAN CAN_RAW CAN_BCM socket.fromfd() socket.close()

新创建的套接字是不可继承的。

*在版本3.3中进行了更改：*添加了AF_CAN系列。AF_RDS系列被添加。

*在版本3.4中更改：*添加了CAN_BCM协议。

*在版本3.4中更改：*返回的套接字现在是不可继承的。

socket.socketpair（[*family* [，*type* [，*proto*]]] ）
使用给定的地址系列，套接字类型和协议编号构建一对连接的套接字对象。地址族，套接字类型和协议号与上述socket()功能一样。 AF_UNIX 如果在平台上定义了默认系列，否则，默认是AF_INET。

新创建的套接字是不可继承的。

*在版本3.2中更改：*返回的套接字对象现在支持整个套接字API，而不是子集。

*在版本3.4中更改：*返回的套接字现在是不可继承的。

*版本3.5中已更改：*添加了Windows支持。

socket.create_connection（*address*[，*timeout*[，*source_address*]]）

连接到监听Internet *地址*（2元组）的TCP服务，然后返回套接字对象。这是一个更高层次的功能比：如果*主机*是一个非数字的主机名，它会尽力解决两个和，然后尝试连接到所有可能的地址，直到有一个连接成功。这使得编写兼容IPv4和IPv6的客户端变得容易。（host，port）socket.connect()AF_INETAF_INET6

尝试连接之前，传递可选的*timeout*参数将在套接字实例上设置超时。如果没有*超时*供应，全局默认超时设置由返回 getdefaulttimeout()使用。

如果提供，在连接之前，*source_address*必须是套接字绑定的2元组作为其源地址。如果主机或端口分别为“0”或0，则将使用操作系统默认行为。（host，port）

*在版本3.2中更改：*添加了*source_address*。

socket.fromfd（*fd*，*family*，*type*，*proto = 0*）

复制文件描述符*fd*（由文件对象的fileno()方法返回的整数），并根据结果构建一个套接字对象。地址族，套接字类型和协议号码与上述socket()功能相同。文件描述符应该引用一个套接字，但这不会被检查 - 如果文件描述符无效，对象的后续操作可能会失败。这个函数很少需要，但可以用来在传递给程序的套接字上获取或设置套接字选项作为标准输入或输出（例如由Unix inet守护进程启动的服务器）。假设套接字处于阻塞模式。

新创建的套接字是不可继承的。

*在版本3.4中更改：*返回的套接字现在是不可继承的。

socket.fromshare（*数据*）

从该socket.share()方法获得的数据实例化一个套接字。假设套接字处于阻塞模式。

可用性：Windows。

3.3版本的新功能

socket.SocketType

这是一个代表套接字对象类型的Python类型对象。它和type(socket(...))。一样。

18.1.2.3.2。其他功能

该socket模块还提供各种与网络相关的服务：

socket.getaddrinfo（*host*，*port*，*family = 0*，*type = 0*，*proto = 0*，*flags = 0*）

将*主机* / *端口*参数转换为5元组序列，其中包含创建连接到该服务的套接字的所有必要参数。*主机*是域名，IPv4 / v6地址的字符串表示None。*端口*是一个字符串服务名称，如'http'数字端口号或None。通过传递None作为*主机*和*端口*的值，您可以传递NULL给底层的C API。

该*家族*，*类型*和*原*参数可以以缩小返回的地址的列表中随意指定的。将零作为每个参数的值传递将选择全部范围的结果。该*标志*参数可以是一个或几个的AI_*常数，会影响结果的方式计算并返回。例如，AI_NUMERICHOST将禁用域名解析并在*主机*是域名时引发错误。

该函数返回具有以下结构的5元组列表：

(family, type, proto, canonname, sockaddr)

在这些元组中，*family*，*type*，*proto*都是整数，并且意味着被传递给socket()函数。 *canonname*是一个表示*主机*规范名称的字符串，如果它AI_CANONNAME是*flags*参数的一部分的话；否则*canonname*将为空。 *sockaddr*是一个描述套接字地址的元组，其格式取决于返回的*家族*（2元组，用于4元组），并且意在传递给该 方法。(address, port)AF_INET(address, port, flow info, scope id)AF_INET6socket.connect()

以下示例将假设的TCP连接的地址信息提取到example.org端口80（如果未启用IPv6，系统的结果可能会有所不同）：

```
>>> socket.getaddrinfo("example.org", 80, proto=socket.IPPROTO_TCP)
[(<AddressFamily.AF_INET6: 10>, <SocketType.SOCK_STREAM: 1>,
 6, '', ('2606:2800:220:1:248:1893:25c8:1946', 80, 0, 0)),
 (<AddressFamily.AF_INET: 2>, <SocketType.SOCK_STREAM: 1>,
 6, '', ('93.184.216.34', 80))]
```

*在版本3.2中更改：*现在可以使用关键字参数传递参数。

socket.getfqdn([*name*])

为*名称*返回完全限定的域名。如果*名称*被省略或为空，则将其解释为本地主机。要查找完全限定名称，将gethostbyaddr()检查返回的主机名，然后检查主机的别名（如果可用）。包含句点的名字被选中。如果没有完全限定的域名可用，则返回由gethostname()返回的主机名称。

socket.gethostbyname(*主机名*)

将主机名转换为IPv4地址格式。IPv4地址以字符串形式返回，例如'100.50.200.5'。如果主机名称本身是IPv4地址，则它将保持不变。查看gethostbyname_ex()更完整的界面。gethostbyname()不支持IPv6名称解析，getaddrinfo()应该用于IPv4 / v6双栈支持。

socket.gethostbyname_ex(*主机名*)

将主机名转换为IPv4地址格式，扩展接口。返回一个triple，其中*主机名*是响应给定*ip_address*的主要主机名，*aliaslist*是同一地址的备用主机名（可能为空）列表，*ipaddrlist*是同一主机上同一接口的IPv4地址列表（通常但不总是单个地址）。不支持IPv6名称解析，应该用于IPv4 / v6双栈支持。(hostname, aliaslist, ipaddrlist)gethostbyname_ex()getaddrinfo()

socket.gethostname()

返回一个字符串，其中包含Python解释器当前正在执行的计算机的主机名。

注意：gethostname()并不总是返回完全合格的域名; getfqdn()为此使用。

socket.gethostbyaddr(*ip_address*)

返回一个triple，其中*主机名*是响应给定*ip_address*的主要主机名，*aliaslist*是同一地址的备用主机名（可能为空）列表，*ipaddrlist*是同一接口的IPv4 / v6地址列表主机（很可能只包含一个地址）。要找到完全限定的域名，请使用该功能。支持IPv4和IPv6。(hostname, aliaslist, ipaddrlist)getfqdn()gethostbyaddr()

socket.getnameinfo(*sockaddr*, *flags*)

将套接字地址*sockaddr*转换为2元组。根据*标志*的设置，结果可以在*主机中*包含完全限定的域名或数字地址表示。同样，*端口*可以包含字符串端口名称或数字端口号。(host, port)

socket.**getprotobyname**(*protocolname*)

将Internet协议名称（例如，'icmp'）转换为适合作为（可选）第三个参数传递给socket() 函数的常量。这通常只需要以“原始”模式（SOCK_RAW）打开的套接字; 对于正常的套接字模式，如果协议被省略或为零，则自动选择正确的协议。

socket.**getservbyname**(*servicename* [, *protocolname*])

将Internet服务名称和协议名称转换为该服务的端口号。可选的协议名称，如果有，应该是'tcp'或'udp'，否则任何协议将匹配。

socket.**getservbyport**(*port* [, *protocolname*])

将Internet端口号和协议名称转换为该服务的服务名称。可选的协议名称，如果有，应该是'tcp'或'udp'，否则任何协议将匹配。

socket.**ntohl**(*x*)

将32位正整数从网络转换为主机字节顺序。在主机字节顺序与网络字节顺序相同的机器上，这是无操作的; 否则，它执行一个4字节的交换操作。

socket.**ntohs**(*x*)

将16位正整数从网络转换为主机字节顺序。在主机字节顺序与网络字节顺序相同的机器上，这是无操作的; 否则，它执行一个2字节的交换操作。

socket.**htonl**(*x*)

将32位正整数从主机转换为网络字节顺序。在主机字节顺序与网络字节顺序相同的机器上，这是无操作的; 否则，它执行一个4字节的交换操作。

socket.**htons**(*x*)

将16位正整数从主机转换为网络字节顺序。在主机字节顺序与网络字节顺序相同的机器上，这是无操作的; 否则，它执行一个2字节的交换操作。

socket.**inet_aton**(*ip_string*)

将IPv4地址从点分四字符串格式（例如'123.45.67.89'）转换为32位打包二进制格式，作为长度为4个字符的字节对象。在与使用标准C库的程序进行交谈并且需要类型对象时这很有用，该类型对于此函数返回的32位打包二进制文件的C类型。struct in_addr

inet_aton() 也接受少于三个点的字符串; 有关详细信息，请参阅Unix手册页*inet（3）*。

如果传递给该函数的IPv4地址字符串无效，OSError将会引发。请注意，究竟什么是有效的取决于底层的C实现inet_aton()。

inet_aton() 不支持IPv6，inet_pton() 应该用于IPv4 / v6双栈支持。

socket.**inet_ntoa**(*packed_ip*)

将32位打包的IPv4地址（类似字节的对象长度为4个字节）转换为其标准的虚线四字符串表示形式（例如'123.45.67.89'）。在与使用标准C库的程序进行交谈并且需要类型对象时这非常有用，该类型对象是该函数用作参数的32位打包二进制数据的C类型。struct in_addr

如果传递给该函数的字节序列长度不是4个字节，OSError则会引发。inet_ntoa() 不支持IPv6，inet_ntop() 应该用于IPv4 / v6双栈支持。

*在版本3.5中更改：*现在可以接受可写入类似字节的对象。

socket.**inet_pton**(*address_family*, *ip_string*)

将IP地址从其特定于家庭的字符串格式转换为打包的二进制格式。 inet_pton() 当库或网络协议调用类型（类似于 ）或类型的对象时非常有用。 struct in_addr inet_aton() struct in6_addr

*address_family*支持的值是当前AF_INET和 AF_INET6。如果IP地址字符串*ip_string*无效， OSError则会引发。请注意，究竟什么是有效的取决于*address_family*的值和底层实现 inet_pton()。

可用性：Unix（可能不是所有平台），Windows。

*在版本3.4中更改：*添加了Windows支持

socket.inet_ntop（*address_family*，*packed_ip* ）

将打包的IP地址（某些字节数的类似字节的对象）转换为其标准的特定于家族的字符串表示形式（例如'7.10.0.5'或'5aef:2b::8'）。 inet_ntop() 当库或网络协议返回类型（类似于）或 类型的对象时非常有用。 struct in_addr inet_ntoa() struct in6_addr

*address_family*支持的值是当前AF_INET和 AF_INET6。如果字节对象*packed_ip*不是指定地址系列的正确长度，ValueError则会引发。 OSError因呼叫错误而被提出inet_ntop()。

可用性：Unix（可能不是所有平台），Windows。

*在版本3.4中更改：*添加了Windows支持

*在版本3.5中更改：*现在可以接受可写入类似字节的对象。

socket.CMSG_LEN（*长度*）

返回辅助数据项的总长度，不带尾部填充，并带有给定*长度的*相关数据。该值通常可以用作recvmsg()接收单个辅助数据项的缓冲区大小，但是**RFC 3542**要求使用便携式应用程序CMSG_SPACE()，因此包含用于填充的空间，即使该项目将是缓冲区中的最后一个。引发OverflowError如果*长度*是值的允许范围之外。

可用性：大多数Unix平台，可能是其他平台

3.3版本的新功能

socket.CMSG_SPACE（*长度*）

返回需要的缓冲区大小recvmsg()以接收具有给定*长度的*关联数据的辅助数据项 以及任何尾部填充。接收多个项目所需的缓冲空间是CMSG_SPACE() 其相关数据长度值的总和。引发OverflowError如果*长度*是值的允许范围之外。

请注意，某些系统可能不支持此功能而支持辅助数据。另请注意，使用此功能的结果设置缓冲区大小可能无法精确限制可接收的辅助数据量，因为其他数据可能能够放入填充区域。

可用性：大多数Unix平台，可能是其他平台

3.3版本的新功能

socket.getdefaulttimeout（ ）

为新套接字对象返回以秒为单位的默认超时值（float）。值None表示新的套接字对象没有超时。首次导入套接字模块时，默认为None。

socket.setdefaulttimeout（*超时*）

为新套接字对象设置默认超时值（以秒为单位）。首次导入套接字模块时，默认为None。查看settimeout()可能的值及其各自的含义。

socket.sethostname(*名字*)

将机器的主机*名称*设置为*名称*。OSError如果您没有足够的权利，这将会引发一次。

可用性：Unix。

3.3版本的新功能

socket.if_nameindex()

返回网络接口信息列表（index int，name string）元组。OSError如果系统调用失败。

可用性：Unix。

3.3版本的新功能

socket.if_nametoindex(*if_name*)

返回与接口名称对应的网络接口索引号。OSError如果没有给定名称的接口存在。

可用性：Unix。

3.3版本的新功能

socket.if_indextoname(*if_index*)

返回接口索引号对应的网络接口名称。OSError如果没有给定索引的接口存在。

可用性：Unix。

3.3版本的新功能

18.1.3。socket对象

套接字对象有以下方法。除此之外makefile()，这些对应于适用于套接字的Unix系统调用。

*在版本3.2中更改：*添加了对上下文管理器协议的支持。退出上下文管理器相当于调用close()。

socket.accept()

接受连接。套接字必须绑定到地址并监听连接。返回值是一对，*conn*是一个*新的*套接字对象，可用于在连接上发送和接收数据，*地址*是绑定到连接另一端套接字的地址。(conn, address)

新创建的套接字是不可继承的。

*在版本3.4中更改：*套接字现在是不可继承的。

*在版本3.5中更改：*如果系统调用中断并且信号处理程序不引发异常，则此方法现在重试系统调用而不是引发InterruptedError异常（请参阅**PEP 475**为理由）。

socket.bind(*地址*)

将套接字绑定到*地址*。套接字不能被绑定。（*地址*的格式取决于地址族 - 参见上文。）

socket.close()

将套接字标记为关闭。底层系统资源（例如文件描述符）在关闭所有文件对象时makefile()也会关闭。一旦发生这种情况，套接字对象的所有未来操作都将失败。远程端将不会收到更多数据（在排队数据刷新后）。

套接字在被垃圾收集时会自动关闭，但建议close()他们明确地使用它们，或者with在它们周围使用 语句。

OSError如果在进行底层close()调用时发生错误，则现在引发 *版本3.6中的更改：*

> **注意：** close()释放与连接关联的资源，但不一定立即关闭连接。如果您想及时关闭连接，shutdown() 请先致电close()。

socket.**connect**(*地址*)

连接到*地址*的远程套接字。（ *地址*的格式取决于地址族 - 参见上文。 ）

如果连接被信号中断，则方法一直等到连接完成，或者socket.timeout如果信号处理程序没有引发异常并且套接字被阻塞或超时，则会引发超时。对于非阻塞套接字， InterruptedError如果连接被信号中断（或信号处理程序引发的异常），该方法会引发 异常。

*在版本3.5中进行了更改：*该方法现在等待连接完成，而不是 InterruptedError在连接被信号中断时引发异常，信号处理程序不会引发异常并且套接字被阻塞或超时（请参阅**PEP 475**为理由）。

socket.**connect_ex**(*地址*)

像connect(address)，但返回一个错误指示符，而不是引发由C级connect()调用返回的错误的异常（其他问题，例如“找不到主机”，仍然可能引发异常）。 0如果操作成功，则错误指示符为该errno变量的值，否则为 错误指示符。这对于支持异步连接很有用。

socket.**detach**( )

将套接字对象置于关闭状态而不实际关闭底层文件描述符。文件描述符被返回，并且可以被重新用于其他目的。

3.2版本中的新功能

socket.**dup**( )

重复套接字。

新创建的套接字是不可继承的。

*在版本3.4中更改：*套接字现在是不可继承的。

socket.**fileno**( )

返回套接字的文件描述符（一个小整数），或者在失败时返回-1。这对于有用select.select()。

在Windows下，此方法返回的小整数不能用于可以使用文件描述符（例如os.fdopen()）的地方。Unix没有这个限制。

socket.**get_inheritable**( )

获取套接字文件描述符或套接字句柄的可继承标志：True如果套接字可以在子进程中继承，False如果它不能。

3.4版新增功能

socket.**getpeername**( )

返回套接字连接的远程地址。例如，这对找出远程IPv4 / v6套接字的端口号很有用。（返回地址的格式取决于地址族 - 参见上文。）在某些系统上，此功能不受支持。

socket.getsockname()

返回套接字自己的地址。例如，这对于找出IPv4 / v6套接字的端口号很有用。（返回地址的格式取决于地址族 - 参见上文。）

socket.getsockopt(*level*, *optname*[, *buflen*])

返回给定套接字选项的值（请参阅Unix手册页 *getsockopt（2）*）。所需的符号常量（SO_*等）在本模块中定义。如果*buflen*不存在，则假定整数选项，并且该函数返回其整数值。如果存在*buflen*，它指定用于接收选项的缓冲区的最大长度，并且此缓冲区作为字节对象返回。这取决于调用者解码缓冲区的内容（请参阅可选的内置模块struct以解码编码为字节字符串的C结构）。

socket.gettimeout()

返回与套接字操作相关的超时秒数（浮点数），或者None没有设置超时时间。这反映了最后一次呼叫 setblocking()或settimeout()。

socket.ioctl(*控制*, *选项*)

平台：	视窗

该ioctl()方法是WSAIoctl系统界面的有限接口。有关更多信息，请参阅Win32文档。

在其他平台上，可以使用通用fcntl.fcntl()和fcntl.ioctl()功能; 他们接受一个socket对象作为他们的第一个参数。

目前，只有下面的控制代码的支持：SIO_RCVALL，SIO_KEEPALIVE_VALS，和SIO_LOOPBACK_FAST_PATH。

*在版本3.6中更改：*SIO_LOOPBACK_FAST_PATH已添加。

socket.listen([*backlog*])

启用服务器以接受连接。如果指定*backlog*，它必须至少为0（如果较低，则设置为0）; 它指定拒绝新连接之前系统允许的未接受连接数。如果未指定，则选择默认的合理值。

*改变在3.5版本：*该*积压*参数现在是可选的。

socket.makefile(*mode='r'*, *buffering=None*, ***, *encoding=None*, *errors=None*, *newline=None*)

返回与套接字关联的文件对象。确切的返回类型取决于给定的参数makefile()。这些参数的解释方式与内置open()函数的解释方式相同，只是唯一支持的*模式*值是'r'（默认）'w'和'b'。

套接字必须处于阻塞模式; 它可能有一个超时，但如果发生超时，文件对象的内部缓冲区可能会以不一致的状态结束。

关闭通过返回的文件对象makefile()将不会关闭原始套接字，除非所有其他文件对象已关闭并且socket.close()已在套接字对象上调用。

> **注意：** 在Windows上，makefile()不能在具有文件描述符的文件对象所在的位置使用由此创建的类文件对象，例如subprocess.Popen()。

socket.recv(*bufsize*[, *flags*])

从套接字接收数据。返回值是一个表示接收到的数据的字节对象。一次接收的最大数据量由*bufsize*指定。有关可选参数*标志*的含义，请参阅Unix手册页*recv（2）*; 它默认为零。

注意: 为了与硬件和网络现实最佳匹配，*bufsize*的值 应该是2的相对小的幂，例如4096。

*在版本3.5中更改：*如果系统调用中断并且信号处理程序不引发异常，则此方法现在重试系统调用而不是引发InterruptedError异常（请参阅**PEP 475**为理由）。

socket.**recvfrom**(*bufsize*[, *flags*])

从套接字接收数据。返回值是一对，其中*bytes*是表示接收数据的字节对象，*address*是发送数据的套接字的地址。有关可选参数*标志*的含义，请参阅Unix手册页 *recv（2）*；它默认为零。（*地址*的格式取决于地址族 - 参见上文。）(bytes, address)

*在版本3.5中更改：*如果系统调用中断并且信号处理程序不引发异常，则此方法现在重试系统调用而不是引发InterruptedError异常（请参阅**PEP 475**为理由）。

socket.**recvmsg**(*bufsize*[, *ancbufsize*[, *flags*]])

从套接字接收正常数据（最大*bufsize*字节）和辅助数据。所述*ancbufsize*参数设置在用于接收的辅助数据的内部缓冲区的字节大小; 它默认为0，这意味着不会收到辅助数据。可以使用CMSG_SPACE()或计算辅助数据的适当缓冲区大小 CMSG_LEN()，并且不适合缓冲区的项目可能会被截断或丢弃。该*标志*参数默认为0，有用法相同 recv()。

返回值是一个4元组：。该*数据*项是一个保持接收到的非辅助性的数据对象。该*ancdata*项是零个或多个元组的列表表示辅助数据（控制消息）接收：*cmsg_level*和 *cmsg_type*分别指定协议级和协议特定的类型整数，且*cmsg_data*是 保持相关联的数据对象。该*msg_flags* 项是按位或的各种标志指示在接收到的消息的条件; 详细信息请参阅您的系统文档。如果接收套接字未连接，则为*地址*(data, ancdata, msg_flags, address) bytes (cmsg_level, cmsg_type, cmsg_data) bytes是发送套接字的地址（如果可用）; 否则，其值不确定。

在某些系统，sendmsg()并且recvmsg()可以被用于在一个进程之间传递文件描述符AF_UNIX 插座。当使用这个工具时（它通常只限于 SOCK_STREAM套接字），recvmsg()将在其辅助数据中返回表单的项目，其中*fds*是代表新文件描述符的对象，作为本机C 类型的二进制数组。如果系统调用返回后引发异常，它将首先尝试关闭通过此机制接收的任何文件描述符。(socket.SOL_SOCKET, socket.SCM_RIGHTS, fds) bytes int recvmsg()

某些系统不会指示仅部分接收到的辅助数据项的截断长度。如果一个项目看起来超出了缓冲区的末尾，recvmsg()将会发出一个RuntimeWarning，并且会返回缓冲区内部的一部分，只要它在相关数据开始之前没有被截断。

在支持该SCM_RIGHTS机制的系统上，以下函数将接收*maxfds*文件描述符，返回消息数据和包含描述符的列表（同时忽略意外情况，例如接收到不相关的控制消息）。另见sendmsg()。

```
import socket, array

def recv_fds(sock, msglen, maxfds):
    fds = array.array("i")   # Array of ints
    msg, ancdata, flags, addr = sock.recvmsg(msglen, socket.CMSG_LEN(maxfds * fds.itemsi
    for cmsg_level, cmsg_type, cmsg_data in ancdata:
        if (cmsg_level == socket.SOL_SOCKET and cmsg_type == socket.SCM_RIGHTS):
            # Append data, ignoring any truncated integers at the end.
            fds.fromstring(cmsg_data[:len(cmsg_data) - (len(cmsg_data) % fds.itemsize)])
    return msg, list(fds)
```

可用性：大多数Unix平台，可能是其他平台

3.3版本的新功能

*在版本3.5中更改：*如果系统调用中断并且信号处理程序不引发异常，则此方法现在重试系统调用而不是引发InterruptedError异常（请参阅**PEP 475**为理由）。

socket.**recvmsg_into**(*buffers*[, *ancbufsize*[, *flags*]])

从套接字接收正常的数据和辅助数据，其行为如同 recvmsg()，但将非辅助数据分散到一系列缓冲区中，而不是返回新的字节对象。所述 *缓冲器*参数必须是可写入的出口缓冲器（例如对象的可迭代bytearray的对象）；这些数据将被连续的非辅助数据块填充，直到全部被写入或没有更多的缓冲区。操作系统可以对可以使用的缓冲区数量设置限制（ sysconf()值SC_IOV_MAX ）。该 *ancbufsize*和 *标志*参数的含义为相同recvmsg()。

返回值是一个4元组，其中*nbytes*是写入缓冲区的非辅助数据的总字节数，*ancdata*， *msg_flags*和*address*与for相同。(nbytes, ancdata, msg_flags, address)recvmsg()

例：

```
>>> import socket
>>> s1, s2 = socket.socketpair()
>>> b1 = bytearray(b'----')
>>> b2 = bytearray(b'0123456789')
>>> b3 = bytearray(b'--------------')
>>> s1.send(b'Mary had a little lamb')
22
>>> s2.recvmsg_into([b1, memoryview(b2)[2:9], b3])
(22, [], 0, None)
>>> [b1, b2, b3]
[bytearray(b'Mary'), bytearray(b'01 had a 9'), bytearray(b'little lamb---')]
```

可用性：大多数Unix平台，可能是其他平台

3.3版本的新功能

socket.**recvfrom_into**(*buffer*[, *nbytes*[, *flags*]])

从套接字接收数据，将其写入*缓冲区*而不是创建新的字节串。返回值是一对，其中*nbytes*是接收的字节数，*address*是发送数据的套接字的地址。有关可选参数*标志*的含义，请参阅Unix手册页*recv（2）*；它默认为零。（*地址*的格式 取决于地址族 - 参见上文。）(nbytes, address)

socket.**recv_into**(*buffer*[, *nbytes*[, *flags*]])

接收到*的nbytes*从插座字节，所述数据存储到缓冲器中，而不是创建一个新的字节串。如果未指定*nbytes*（或0），则接收达到给定缓冲区中可用的大小。返回接收的字节数。有关可选参数*标志*的含义，请参阅Unix手册页*recv（2）*；它默认为零。

socket.**send**(*bytes*[, *flags*])

将数据发送到套接字。套接字必须连接到远程套接字。可选的*flags*参数具有与recv()上面相同的含义。返回发送的字节数。应用程序负责检查所有数据是否已发送; 如果只传输了一些数据，则应用程序需要尝试传送剩余的数据。有关该主题的更多信息，请参阅Socket编程HOWTO。

*在版本3.5中更改：*如果系统调用中断并且信号处理程序不引发异常，则此方法现在重试系统调用而不是引发InterruptedError异常（请参阅**PEP 475**为理由）。

socket.**sendall**(*bytes*[, *flags*])

将数据发送到套接字。套接字必须连接到远程套接字。可选的*flags*参数具有与recv()上面相同的含义。与send()此不同，此方法继续从*字节*发送数据，直到发送所有数据或发生错误。 None成功返回。出错时，会引发异常，并且无法确定成功发送了多少数据（如果有）。

*在版本3.5中更改：*每次成功发送数据时，套接字超时不再被重置。套接字超时现在是发送所有数据的最大总持续时间。

*在版本3.5中更改：*如果系统调用中断并且信号处理程序不引发异常，则此方法现在重试系统调用而不是引发InterruptedError异常（请参阅**PEP 475**为理由）。

socket.sendto(*字节*, *地址*)

socket.sendto(*字节*, *标志*, *地址*)

将数据发送到套接字。由于目标套接字由*地址*指定，因此套接字不应连接到远程套接字。可选的*flags*参数具有与recv()上面相同的含义。返回发送的字节数。（*地址*的格式取决于地址族 - 参见上文。）

*在版本3.5中更改：*如果系统调用中断并且信号处理程序不引发异常，则此方法现在重试系统调用而不是引发InterruptedError异常（请参阅**PEP 475**为理由）。

socket.sendmsg(*buffers*[, *ancdata*[, *flags*[, *address*]]])

将普通数据和辅助数据发送到套接字，从一系列缓冲区收集非辅助数据并将其连接成一条消息。所述*缓冲器*参数指定为可迭代非辅助数据 等字节的对象（例如，bytes对象）；操作系统可以设置可以使用的缓冲区数量的限制（sysconf()值SC_IOV_MAX）。所述*ancdata*参数指定所述辅助数据（控制消息）作为零个或多个元组的迭代，其中*cmsg_level*和*cmsg_type*分别指定协议级和协议特定的类型整数，且*cmsg_data*(cmsg_level, cmsg_type, cmsg_data)是一个持有关联数据的类似字节的对象。请注意，某些系统（特别是无系统CMSG_SPACE()）可能支持每次呼叫仅发送一条控制消息。该*标志*参数默认为0，有用法相同 send()。如果提供了*地址*而不是None，它会设置消息的目标地址。返回值是发送的非辅助数据的字节数。

下面的函数发送文件描述符的列表*FDS*通过一个AF_UNIX插座，在其支持的系统 SCM_RIGHTS机制。另见recvmsg()。

```
import socket, array

def send_fds(sock, msg, fds):
    return sock.sendmsg([msg], [(socket.SOL_SOCKET, socket.SCM_RIGHTS, array.array("i",
```

可用性：大多数Unix平台，可能是其他平台

3.3版本的新功能

*在版本3.5中更改：*如果系统调用中断并且信号处理程序不引发异常，则此方法现在重试系统调用而不是引发InterruptedError异常（请参阅**PEP 475**为理由）。

socket.sendmsg_afalg([*msg*,]*, *op*[, *iv*[, *assoclen*[, *flags*]]])

sendmsg()for AF_ALGsocket的专用版本。设置模式，IV，AEAD关联的数据长度和AF_ALG插座的标志。

可用性：Linux> = 2.6.38

3.6版本中的新功能。

socket.sendfile(*文件*, *偏移量= 0*, *计数=无*)

使用高性能发送文件直到达到EOF os.sendfile并返回发送的字节总数。*文件*必须是以二进制模式打开的常规文件对象。如果 os.sendfile不可用（例如Windows）或*文件*不是常规文件send()将被使用。*偏移量*指示从何处开始读取文件。如果指定，*count*是传输字节的总数，而不是发送文件

直到达到EOF。返回时更新文件位置，或者在发生错误的情况下更新文件位置，在这种情况下file.tell()可用于计算发送的字节数。套接字必须是SOCK_STREAM类型的。非阻塞套接字不受支持。

3.5版本中的新功能。

socket.**set_inheritable**(*可继承*)

设置套接字文件描述符或套接字句柄的可继承标志。

3.4版新增功能

socket.**setblocking**(*标志*)

设置套接字的阻塞或非阻塞模式：如果*标志*为false，则套接字设置为非阻塞模式，否则设置为阻塞模式。

此方法是某些settimeout()呼叫的简写：

- sock.setblocking(True) 相当于 sock.settimeout(None)
- sock.setblocking(False) 相当于 sock.settimeout(0.0)

socket.**settimeout**(*价值*)

设置阻塞套接字操作的超时时间。所述*值*参数可以是表达秒的非负的浮点数，或None。如果给出非零值，则在操作完成之前，timeout如果超时时间*值*已过，则后续套接字操作将引发 异常。如果给出零，则套接字处于非阻塞模式。如果None给出，套接字被置于阻塞模式。

有关更多信息，请参阅套接字超时的说明。

socket.**setsockopt**(*level*，*optname*，*value：int*)

socket.**setsockopt**(*级别*，*optname*，*值：缓冲区*)

socket.**setsockopt**(*级别*，*optname*，*None*，*optlen：int*)

设置给定套接字选项的值（请参阅Unix手册页的*setsockopt（2）*）。所需的符号常量在socket模块（SO_*等）中定义。该值可以是一个整数，None或一个类字节对象表示的缓冲器。在后面的情况下，由调用者来确保字节串包含正确的位（请参阅可选的内置模块struct以将C结构编码为字节串）。当值设置为None，optlen参数是必需的。这相当于使用optval = NULL和optlen = optlen来调用setsockopt C函数。

*在版本3.5中更改：*现在可以接受可写入类似字节的对象。

*在版本3.6中进行了更改：添加了*setsockopt（level，optname，None，optlen：int）表单。

socket.**shutdown**(*如何*)

关闭一个或两个连接的一半。如果*怎么*是SHUT_RD，还接收被禁止。如果*怎么*是SHUT_WR，进一步将是不允许的。如果*怎么*是SHUT_RDWR，进一步的发送和接收被禁止。

socket.**share**(*process_id*)

复制套接字并准备与目标进程共享。目标进程必须提供*process_id*。然后可以使用某种形式的进程间通信将结果字节对象传递给目标进程，并且可以使用套接字重新创建fromshare()。一旦这个方法被调用，关闭套接字是安全的，因为操作系统已经为目标进程复制了它。

可用性：Windows。

3.3版本的新功能

请注意，没有方法read()或write()；使用 recv()和send()没有*标志*的说法来代替。

套接字对象也具有这些（只读）属性，这些属性与给予socket构造函数的值相对应。

socket.family
套接字系列。

socket.type
套接字类型。

socket.proto
套接字协议。

18.1.4。套接字超时的注意事项

套接字对象可以处于以下三种模式之一：阻塞，非阻塞或超时。套接字默认情况下始终以阻塞模式创建，但可以通过调用进行更改setdefaulttimeout()。

- 在*阻塞模式下*，操作阻塞直至完成或系统返回错误（如连接超时）。
- 在*非阻塞模式下*，如果无法立即完成操作，则操作失败（带有不幸系统相关的错误）：select可以使用函数 来知道何时以及是否可以读取或写入套接字。
- 在*超时模式下*，如果在套接字指定的超时（它们引发timeout异常）或系统返回错误时无法完成操作，则操作失败。

注意： 在操作系统级别，处于*超时模式的*套接字在内部设置为非阻塞模式。此外，阻塞和超时模式在文件描述符和引用相同网络端点的套接字对象之间共享。如果你决定使用fileno()套接字，这个实现细节可能会有明显的后果。

18.1.4.1。超时和connect方法

该connect()操作也受到超时设置的限制，并且通常建议settimeout() 在调用connect()或传递超时参数之前调用该操作 create_connection()。但是，无论是否有任何Python套接字超时设置，系统网络堆栈也可能会返回其自身的连接超时错误。

18.1.4.2。超时和accept方法

如果getdefaulttimeout()不是None，则该accept()方法返回的套接字将继承该超时。否则，行为取决于侦听套接字的设置：

- 如果侦听套接字处于*阻塞模式*或处于*超时模式*，则返回的套接字accept()处于*阻塞模式*；
- 如果侦听套接字处于*非阻塞模式*，则返回的套接字accept()是处于阻塞模式还是非阻塞模式取决于操作系统。如果您想确保跨平台行为，建议您手动覆盖此设置。

18.1.5。示例

以下是使用TCP / IP协议的四个最小示例程序：一个服务器，用于回显接收的所有数据（仅服务一个客户端）以及一个使用它的客户端。注意，服务器必须执行序列socket()， bind()，listen()，accept()（可能重复accept()以服务一个以上的客户端），而一个客户端只需要在序列socket()，

connect()。还要注意的是，服务器没有sendall()/recv()它正在侦听插座上，而在返回的新套接字accept()。

前两个示例仅支持IPv4。

```
# Echo server program
import socket

HOST = ''                 # Symbolic name meaning all available interfaces
PORT = 50007              # Arbitrary non-privileged port
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen(1)
    conn, addr = s.accept()
    with conn:
        print('Connected by', addr)
        while True:
            data = conn.recv(1024)
            if not data: break
            conn.sendall(data)
```

```
# Echo client program
import socket

HOST = 'daring.cwi.nl'    # The remote host
PORT = 50007              # The same port as used by the server
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b'Hello, world')
    data = s.recv(1024)
print('Received', repr(data))
```

接下来的两个例子与上述两个例子相同，但支持IPv4和IPv6。服务器端将听取可用的第一个地址族（它应该监听两者）。在大多数IPv6就绪系统中，IPv6将优先，服务器可能不接受IPv4流量。客户端将尝试连接到由于名称解析而返回的所有地址，并将流量发送到第一个成功连接的地址。

```
# Echo server program
import socket
import sys

HOST = None               # Symbolic name meaning all available interfaces
PORT = 50007              # Arbitrary non-privileged port
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC,
                              socket.SOCK_STREAM, 0, socket.AI_PASSIVE):
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
    except OSError as msg:
        s = None
        continue
    try:
        s.bind(sa)
        s.listen(1)
    except OSError as msg:
        s.close()
        s = None
        continue
    break
```

```
if s is None:
    print('could not open socket')
    sys.exit(1)
conn, addr = s.accept()
with conn:
    print('Connected by', addr)
    while True:
        data = conn.recv(1024)
        if not data: break
        conn.send(data)
```

```
# Echo client program
import socket
import sys

HOST = 'daring.cwi.nl'    # The remote host
PORT = 50007              # The same port as used by the server
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC, socket.SOCK_STREAM):
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
    except OSError as msg:
        s = None
        continue
    try:
        s.connect(sa)
    except OSError as msg:
        s.close()
        s = None
        continue
    break
if s is None:
    print('could not open socket')
    sys.exit(1)
with s:
    s.sendall(b'Hello, world')
    data = s.recv(1024)
print('Received', repr(data))
```

下一个例子展示了如何在Windows上使用原始套接字编写一个非常简单的网络嗅探器。该示例需要管理员权限来修改接口：

```
import socket

# the public network interface
HOST = socket.gethostbyname(socket.gethostname())

# create a raw socket and bind it to the public interface
s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_IP)
s.bind((HOST, 0))

# Include IP headers
s.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

# receive all packages
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)

# receive a package
print(s.recvfrom(65565))
```

```
# disabled promiscuous mode
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)
```

最后一个例子显示了如何使用套接字接口使用原始套接字协议与CAN网络进行通信。为了使用CAN与广播管理器协议，请使用以下方法打开一个套接字：

```
socket.socket(socket.AF_CAN, socket.SOCK_DGRAM, socket.CAN_BCM)
```

结合（后CAN_RAW）或连接（CAN_BCM）插槽，你可以使用socket.send()，而socket.recv()插座的对象像往常一样操作（及其对应）。

这个例子可能需要特殊的权限：

```
import socket
import struct


# CAN frame packing/unpacking (see 'struct can_frame' in <linux/can.h>)

can_frame_fmt = "=IB3x8s"
can_frame_size = struct.calcsize(can_frame_fmt)

def build_can_frame(can_id, data):
    can_dlc = len(data)
    data = data.ljust(8, b'\x00')
    return struct.pack(can_frame_fmt, can_id, can_dlc, data)

def dissect_can_frame(frame):
    can_id, can_dlc, data = struct.unpack(can_frame_fmt, frame)
    return (can_id, can_dlc, data[:can_dlc])


# create a raw socket and bind it to the 'vcan0' interface
s = socket.socket(socket.AF_CAN, socket.SOCK_RAW, socket.CAN_RAW)
s.bind(('vcan0',))

while True:
    cf, addr = s.recvfrom(can_frame_size)

    print('Received: can_id=%x, can_dlc=%x, data=%s' % dissect_can_frame(cf))

    try:
        s.send(cf)
    except OSError:
        print('Error sending CAN frame')

    try:
        s.send(build_can_frame(0x01, b'\x01\x02\x03'))
    except OSError:
        print('Error sending CAN frame')
```

在执行过程中执行多次延迟太小的示例可能会导致此错误：

```
OSError: [Errno 98] Address already in use
```

这是因为之前的执行已经使套接字处于一种TIME_WAIT 状态，并且不能立即重用。

有一个socket标志设置，以防止这一点， socket.SO_REUSEADDR ：

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind((HOST, PORT))
```

该SO_REUSEADDR标志告诉内核在TIME_WAIT状态中重用本地套接字 ，而不等待其自然超时过期。

也可以看看: 有关套接字编程的介绍（C语言），请参阅以下文章：

- Stuart Sechrest编写的*4.3BSD Interprocess通讯教程*
- Samuel J. Leffler等人*的"Advanced 4.3BSD Interprocess Communication Tutorial"*

无论是在"UNIX程序员手册"的补充文档1（PS1：7和PS1：8章节）中。各种与套接字相关的系统调用的特定于平台的参考材料也是关于套接字语义细节的宝贵信息来源。对于Unix，请参考手册页面;对于Windows，请参阅WinSock（或Winsock 2）规范。对于可用于IPv6的API，读者可能想参考**RFC 3493**标题为IPv6的基本套接字接口扩展。

18.2。 ssl- 套接字对象的TLS / SSL封装

源代码： Lib / ssl.py

此模块为客户端和服务器端的网络套接字提供对传输层安全性（通常称为“安全套接字层”）加密和对等身份验证功能的访问。该模块使用OpenSSL库。只要OpenSSL安装在该平台上，就可以在所有现代Unix系统，Windows，Mac OS X以及其他平台上使用。

注意： 某些行为可能与平台有关，因为调用操作系统套接字API。安装的OpenSSL版本也可能导致行为的变化。例如，TLSv1.1和TLSv1.2带有openssl 1.0.1版本。

警告： 未阅读安全注意事项，请勿使用此模块。这样做可能会导致错误的安全感，因为ssl模块的默认设置不一定适合您的应用程序。

本节记录ssl模块中的对象和功能; 有关TLS，SSL和证书的更多一般信息，读者可参阅底部“另请参见”部分的文档。

这个模块提供了一个ssl.SSLSocket从socket.socket类型派生出来的 类，它提供了一个类似套接字的包装器，它也使用SSL加密和解密通过套接字的数据。它支持其他方法，如getpeercert()检索连接另一端的证书，以及cipher()检索用于安全连接的密码。

对于更复杂的应用程序， ssl.SSLContext该类有助于管理设置和证书，然后可以通过该SSLContext.wrap_socket()方法创建的SSL套接字继承该设置和证书。

*在版本3.6中更改：*不推荐使用OpenSSL 0.9.8,1.0.0和1.0.1，不再支持。未来，ssl模块至少需要OpenSSL 1.0.2或1.1.0。

18.2.1。函数，常量和异常

*异常*ssl.SSLError

引发了底层SSL实现的错误（目前由OpenSSL库提供）。这表示在底层网络连接上叠加的更高级别的加密和认证层中存在一些问题。这个错误是一个子类型OSError。 SSLError实例的错误代码和消息 由OpenSSL库提供。

*在版本3.3中更改：*SSLError曾经是一个子类型socket.error。

library

一个字符串助记符指定在发生错误，如子模块的OpenSSL SSL，PEM或X509。可能值的范围取决于OpenSSL版本。

3.3版本的新功能

reason

例如，指定发生此错误的原因的字符串助记符CERTIFICATE_VERIFY_FAILED。可能值的范围取决于OpenSSL版本。

3.3版本的新功能

*异常*ssl.SSLZeroReturnError

SSLError尝试读取或写入时引发的子类以及SSL连接已被彻底关闭。请注意，这并不意味着底层传输（读取TCP）已关闭。

3.3版本的新功能

*异常*ssl.SSLWantReadError

尝试读取或写入数据时SSLError由非阻塞SSL套接字引发的子类，但在实现请求之前需要在底层TCP传输上接收更多数据。

3.3版本的新功能

*异常*ssl.SSLWantWriteError

尝试读取或写入数据时SSLError由非阻塞SSL套接字引发的子类，但在请求可以实现之前需要在底层TCP传输上发送更多数据。

3.3版本的新功能

*异常*ssl.SSLSyscallError

SSLError在尝试在SSL套接字上完成操作时遇到系统错误时引发的子类。不幸的是，没有简单的方法来检查原始的errno号码。

3.3版本的新功能

*异常*ssl.SSLEOFError

SSLErrorSSL连接突然终止时引发的子类。通常，遇到此错误时，不应尝试重新使用底层传输。

3.3版本的新功能

*异常*ssl.CertificateError

引发一个证书错误（如主机名不匹配）。但是OpenSSL检测到的证书错误引发了一个问题SSLError。

18.2.1.1。套接字创建

以下功能允许创建独立套接字。从Python 3.2开始，它可以更灵活地使用SSLContext.wrap_socket()。

ssl.wrap_socket(*sock*, *keyfile = None*, *certfile = None*, *server_side = False*, *cert_reqs = CERT_NONE*, *ssl_version = {see docs}*, *ca_certs = None*, *do_handshake_on_connect = True*, *suppress_ragged_eofs = True*, *ciphers = None*)

需要一个实例sock的socket.socket，并且返回的一个实例ssl.SSLSocket，的一个亚型socket.socket，它包装在SSL上下文底层套接字。 sock必须是一个SOCK_STREAM插座; 其他套接字类型不受支持。

对于客户端套接字，上下文结构是懒惰的; 如果底层套接字尚未连接，则将connect()在套接字上调用上下文构造。对于服务器端套接字，如果套接字没有远程对等端，则假定它是一个监听套接字，服务器端SSL封装将自动在通过该accept()方法接受的客户端连接上执行。 wrap_socket()可能会提高SSLError。

在keyfile和certfile参数指定包含证书被用来识别连接的本地端可选的文件。请参阅证书的讨论，以获取更多关于如何在证书中存储证书的信息certfile。

该参数server_side是一个布尔值，用于标识此套接字是否需要服务器端或客户端行为。

该参数cert_reqs指定是否需要从连接的另一端获得证书，以及是否在提供时验证证书。它必须是三个值之一CERT_NONE （忽略证书），CERT_OPTIONAL（不需要，但是如果提供CERT_REQUIRED验证）或（需要和验证）。如果此参数的值不是CERT_NONE，则该ca_certs参数必须指向一个CA证书文件。

该ca_certs文件包含一组连接的“证书颁发机构”证书，用于验证从连接的另一端传递的证书。有关如何在此文件中安排证书的更多信息，请参阅证书的讨论。

该参数ssl_version指定要使用哪个版本的SSL协议。通常，服务器选择特定的协议版本，并且客户端必须适应服务器的选择。大多数版本不能与其他版本互操作。如果未指定，则默认为 PROTOCOL_TLS; 它提供了与其他版本的最大兼容性。

下面是一个表格，显示客户端（侧面）中的哪些版本可以连接到服务器中的哪些版本（沿顶部）：

客户端 / **服务器**	**的SSLv2**	**在SSLv3**	**TLS [3]**	**使用TLSv1**	**TLSv1.1**	**TLSv1.2工作**
的SSLv2	是	没有	否[1]	没有	没有	没有
在SSLv3	没有	是	否[2]	没有	没有	没有
TLS（SSLv23）[3]	否[1]	否[2]	是	是	是	是
使用TLSv1	没有	没有	是	是	没有	没有
TLSv1.1	没有	没有	是	没有	是	没有
TLSv1.2工作	没有	没有	是	没有	没有	是

脚注

[1] *（1，2）* SSLContext禁用的SSLv2与OP_NO_SSLv2默认情况下。

[2] *（1，2）* SSLContext禁用的SSLv3与OP_NO_SSLv3默认。

[3] *（1，2）* TLS 1.3协议将提供与PROTOCOL_TLSOpenSSL中> = 1.1.1。对于TLS 1.3，没有专用的PROTOCOL常量。

注意： 哪些连接成功取决于OpenSSL的版本。例如，在OpenSSL 1.0.0之前，SSLv23客户端总是尝试SSLv2连接。

该*密码*参数设置此SSL对象可用的密码。它应该是OpenSSL密码列表格式的字符串。

该参数do_handshake_on_connect指定在执行a之后socket.connect()是否自动执行SSL握手，或者是否通过调用SSLSocket.do_handshake()方法明确调用应用程序。调用SSLSocket.do_handshake()明确地让程序控制握手中涉及的套接字I / O的阻塞行为。

该参数suppress_ragged_eofs指定该SSLSocket.recv()方法应如何 从连接的另一端发出意外的EOF信号。如果指定为True（缺省值），它将返回一个正常的EOF（一个空字节对象），以响应从底层套接字引发的意外的EOF错误; 如果False它会将异常提交给调用者。

*在版本3.2中更改：*新的可选参数*密码*。

18.2.1.2。创建上下文

便利功能可帮助创建SSLContext用于常见目的的对象。

ssl.create_default_context(*purpose* = *Purpose.SERVER_AUTH*，*cafile* = *None*，*capath* = *None*，*cadata* = *None*）

SSLContext为了给定的*目的*，返回一个具有默认设置的新对象。这些设置是由ssl模块选择的，并且通常表示比SSLContext直接调用构造函数时更高的安全级别。

cafile，*capath*，*cadata*代表可信的CA证书，用于信任证书验证，如SSLContext.load_verify_locations()。如果三者都是 None，则此功能可以选择相信系统的默认CA证书。

这些设置是：PROTOCOL_TLS，OP_NO_SSLv2并且 OP_NO_SSLv3带有不带RC4的高加密密码套件，并且没有未经身份验证的密码套件。传递SERVER_AUTH 为*目的*设置verify_mode到CERT_REQUIRED 与任一负载CA证书（当中的至少一个*凭证档案错误*，*capath*或 *cadata*给出），或者使用SSLContext.load_default_certs()以加载默认的CA证书。

> **注意：** 协议，选项，密码和其他设置可能会随时更改为更严格的值，而无需事先弃用。这些值表示兼容性和安全性之间的合理平衡。
> 如果您的应用程序需要特定设置，则应该自行创建 SSLContext并应用设置。

> **注意：** 如果您发现某些较旧的客户端或服务器尝试连接SSLContext由此函数创建的一个错误，表明它们出现“协议或密码套件不匹配”的错误，则可能是因为它们仅支持使用该函数排除的SSL3.0 OP_NO_SSLv3。SSL3.0被广泛认为是完全破碎的。如果您仍希望继续使用此功能，但仍允许使用SSL 3.0连接，则可以使用以下方式重新启用它们：
>
> ```
> ctx = ssl.create_default_context(Purpose.CLIENT_AUTH)
> ctx.options &= ~ssl.OP_NO_SSLv3
> ```

3.4版新增功能

在版本3.4.4中更改： RC4从默认密码字符串中删除。

在版本3.6中更改： ChaCha20 / Poly1305已添加到默认密码字符串。

3DES已从默认密码字符串中删除。

版本3.6.3中已更改： TLS 1.3密码套件TLS_AES_128_GCM_SHA256，TLS_AES_256_GCM_SHA384和TLS_CHACHA20_POLY1305_SHA256已添加到默认密码字符串。

18.2.1.3。随机生成

ssl.RAND_bytes(*num*)

返回*num*密码强的伪随机字节。SSLError如果PRNG没有接种足够的数据，或者当前的RAND方法不支持该操作，则引发一次。RAND_status()可用于检查PRNG的状态RAND_add()并可用于播种PRNG。

几乎所有的应用程序os.urandom()都是可取的

阅读维基百科文章Cryptographically secure pseudorandom number generator（CSPRNG），以获得密码生成器的要求。

3.3版本的新功能

ssl.RAND_pseudo_bytes(*num*)

返回（字节，is_cryptographic）：字节是*num*伪随机字节，is_cryptographic是True如果生成的字节是密码强的。SSLError如果当前RAND方法不支持该操作，则引发一个操作。

生成的伪随机字节序列如果长度足够，将是唯一的，但不一定是不可预测的。它们可以用于非加密目的，也可以用于密码协议中的某些目的，但通常不用于密钥生成等。

几乎所有的应用程序os.urandom()都是可取的

3.3版本的新功能

自从3.6版弃用： OpenSSL已弃用ssl.RAND_pseudo_bytes()，请ssl.RAND_bytes()改用。

ssl.RAND_status()

返回True如果SSL伪随机数生成器已经播种了“足够的”随机性，以及False其他。您可以使用ssl.RAND_egd()和ssl.RAND_add()增加伪随机数生成器的随机性。

ssl.RAND_egd(*路径*)

如果您在某处运行熵收集守护进程（EGD），并且*path*是打开它的套接字连接的路径名，那么将从套接字读取256个字节的随机数，并将其添加到SSL伪随机数生成器以增加生成的密钥的安全性。这通常只对没有更好随机源的系统是必需的。

请参阅http://egd.sourceforge.net/或http://prngd.sourceforge.net/了解熵收集守护进程的来源。

可用性：不适用于LibreSSL和OpenSSL> 1.1.0

ssl.RAND_add(*字节*，*熵*)

将给定的*字节*混合到SSL伪随机数生成器中。参数*熵*（一个浮点数）是包含在字符串中的熵的下限（所以你可以随时使用0.0）。看到有关熵源更多信息的**RFC 1750**。

*在版本3.5中更改：*现在可以接受可写入类似字节的对象。

18.2.1.4。证书处理

ssl.**match_hostname**（*cert*，*hostname*）

验证*证书*（以解码格式返回 SSLSocket.getpeercert()）匹配给定的*主机名*。应用的规则是用于检查HTTPS服务器身份的规则，如概述**RFC 2818**，**RFC 5280**和**RFC 6125**。除了HTTPS之外，该功能还适用于检查各种基于SSL的协议（如FTPS，IMAPS，POPS等）中服务器的身份。

CertificateError在失败时被提出。成功时，该函数不返回任何内容：

```
>>> cert = {'subject': ((('commonName', 'example.com'),),)}
>>> ssl.match_hostname(cert, "example.com")
>>> ssl.match_hostname(cert, "example.org")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/py3k/Lib/ssl.py", line 130, in match_hostname
ssl.CertificateError: hostname 'example.org' doesn't match 'example.com'
```

3.2版本中的新功能

*在版本3.3.3中更改：*现在的功能如下**RFC 6125**，第6.4.3节，在国际化域名（IDN）片段中既不匹配多个通配符（例如*.*.com或*a*.example.org）也不匹配通配符。IDN A标签www*.xn--pthon-kva.org仍然受支持，但x*.python.org不再匹配xn--tda.python.org。

*在版本3.5中进行了更改：*现在支持IP地址的匹配（如果存在于证书的subjectAltName字段中）。

ssl.**cert_time_to_seconds**（*cert_time*）

返回从Epoch开始的秒数，给定cert_time 表示来自以strptime格式（C区域设置）的证书的“notBefore”或“notAfter”日期的字符串。"%b %d %H:%M:%S %Y %Z"

这是一个例子：

```
>>> import ssl
>>> timestamp = ssl.cert_time_to_seconds("Jan  5 09:34:43 2018 GMT")
>>> timestamp
1515144883
>>> from datetime import datetime
>>> print(datetime.utcfromtimestamp(timestamp))
2018-01-05 09:34:43
```

“notBefore”或“notAfter”日期必须使用GMT（**RFC 5280**）。

*版本3.5中的更改：*将输入时间解释为输入字符串中'GMT'时区指定的UTC时间。以前使用过本地时区。返回一个整数（输入格式中不含小数）

ssl.**get_server_certificate**（*addr*，*ssl_version* = *PROTOCOL_TLS*，*ca_certs* = *None*）

给定addr一个受SSL保护的服务器的地址，作为（*主机名*，*端口号*）对，获取服务器的证书，并将其作为PEM编码的字符串返回。如果ssl_version指定，则使用该版本的SSL协议尝试连接到服务器。如果ca_certs指定，它应该是一个包含根证书列表的文件，与用于相同参数的格式相同wrap_socket()。该调用将尝试根据该组根证书验证服务器证书，如果验证尝试失败，将会失败。

*在版本3.3中更改：*此功能现在与IPv6兼容。

*在版本3.5中更改：*默认*ssl_version*从PROTOCOL_SSLv3更改 PROTOCOL_TLS为与现代服务器的最大兼容性。

ssl.DER_cert_to_PEM_cert（*DER_cert_bytes*）
给定一个证书作为DER编码的字节块，返回一个PEM编码的相同证书的字符串版本。

ssl.PEM_cert_to_DER_cert（*PEM_cert_string*）
给定一个证书作为ASCII PEM字符串，为相同的证书返回DER编码的字节序列。

ssl.get_default_verify_paths（）
返回一个带有OpenSSL默认cafile和capath路径的命名元组。路径与使用的路径相同SSLContext.set_default_verify_paths()。返回值是一个命名的元组DefaultVerifyPaths：

- cafile- 解析cafile的路径或者None文件不存在，
- capath- 已解析路径，或者None目录不存在，
- openssl_cafile_env - 指向cafile的OpenSSL环境密钥，
- openssl_cafile - 硬编码的cafile文件路径，
- openssl_capath_env - OpenSSL的环境密钥指向一个capath，
- openssl_capath - 通往capath目录的硬编码路径

可用性：LibreSSL忽略环境变量 openssl_cafile_env和openssl_capath_env

3.4版新增功能

ssl.enum_certificates（*store_name*）
从Windows的系统证书存储中检索证书。*store_name*可能是其中之一CA，ROOT或者MY。Windows也可能提供其他的证书存储。

该函数返回（cert_bytes，encoding_type，trust）元组列表。encoding_type指定cert_bytes的编码。它既可 x509_asn用于X.509 ASN.1数据，也可pkcs_7_asn用于PKCS＃7 ASN.1数据。Trust将证书的目的指定为一组OIDS，或者确切地说，True如果证书对于所有目的都是可信的。

例：

```
>>> ssl.enum_certificates("CA")
[(b'data...', 'x509_asn', {'1.3.6.1.5.5.7.3.1', '1.3.6.1.5.5.7.3.2'}),
 (b'data...', 'x509_asn', True)]
```

可用性：Windows。

3.4版新增功能

ssl.enum_crls（*store_name*）

从Windows的系统证书存储中检索CRL。*store_name*可能是其中之一CA，ROOT或者MY。Windows也可能提供其他的证书存储。

该函数返回（cert_bytes，encoding_type，trust）元组列表。encoding_type指定cert_bytes的编码。它既可x509_asn用于X.509 ASN.1数据，也可pkcs_7_asn用于PKCS＃7 ASN.1数据。

可用性：Windows。

3.4版新增功能

18.2.1.5。常量

所有常量现在enum.IntEnum或enum.IntFlag集合。

3.6版本中的新功能。

ssl.CERT_NONE

可能的值SSLContext.verify_mode，或cert_reqs 参数wrap_socket()。在这种模式下（默认），从套接字连接的另一端不需要证书。如果从另一端收到证书，则不会尝试对其进行验证。

请参阅下面关于安全性考虑的讨论。

ssl.CERT_OPTIONAL

可能的值SSLContext.verify_mode，或cert_reqs 参数wrap_socket()。在这种模式下，套接字连接的另一端不需要证书; 但如果提供了它们，将会尝试进行验证，并SSLError 在失败时进行验证。

使用此设置需要传递一组有效的CA证书，SSLContext.load_verify_locations()或者作为ca_certs参数的值传递给它wrap_socket()。

ssl.CERT_REQUIRED

可能的值SSLContext.verify_mode，或cert_reqs 参数wrap_socket()。在这种模式下，证书需要来自套接字连接的另一端; 一个SSLError 如果验证失败，如果没有提供证书将提高，或。

使用此设置需要传递一组有效的CA证书，SSLContext.load_verify_locations()或者作为ca_certs参数的值传递给它wrap_socket()。

*类*ssl.VerifyMode

enum.IntEnum CERT_ *常量的集合。

3.6版本中的新功能。

ssl.VERIFY_DEFAULT

可能值SSLContext.verify_flags。在这种模式下，证书撤销列表（CRL）不会被检查。默认情况下，OpenSSL既不要求也不验证CRL。

3.4版新增功能

ssl.VERIFY_CRL_CHECK_LEAF

可能值SSLContext.verify_flags。在这种模式下，只有对等证书是检查而非中间CA证书。该模式需要由对等证书的颁发者（其直接祖先CA）签署的有效CRL。如果没有正确的加载SSLContext.load_verify_locations，验证将失败。

3.4版新增功能

ssl.VERIFY_CRL_CHECK_CHAIN

可能值SSLContext.verify_flags。在此模式下，检查对等证书链中所有证书的CRL。

3.4版新增功能

ssl.VERIFY_X509_STRICT

SSLContext.verify_flags为破损的X.509证书禁用解决方法的可能值。

3.4版新增功能

ssl.VERIFY_X509_TRUSTED_FIRST

可能值SSLContext.verify_flags。它指示OpenSSL在构建信任链以验证证书时优先使用可信证书。该标志默认启用。

3.4.4版本的新功能。

*类*ssl.VerifyFlags

enum.IntFlag VERIFY_*常量的集合。

3.6版本中的新功能。

ssl.PROTOCOL_TLS

选择客户端和服务器都支持的最高协议版本。尽管有这个名字，这个选项可以选择“SSL”和“TLS”协议。

3.6版本中的新功能。

ssl.PROTOCOL_TLS_CLIENT

自动协商最高协议版本PROTOCOL_TLS，但仅支持客户端SSLSocket连接。该协议启用CERT_REQUIRED并check_hostname默认。

3.6版本中的新功能。

ssl.PROTOCOL_TLS_SERVER

自动协商最高协议版本PROTOCOL_TLS，但仅支持服务器端SSLSocket连接。

3.6版本中的新功能。

ssl.PROTOCOL_SSLv23

数据的别名：*PROTOCOL_TLS*。

*自从版本3.6开始弃用：*PROTOCOL_TLS改为使用。

ssl.PROTOCOL_SSLv2
选择SSL版本2作为通道加密协议。

如果使用该OPENSSL_NO_SSL2标志编译OpenSSL，则此协议不可用。

> **警告：** SSL版本2不安全。它的使用非常令人沮丧。

*自从3.6版弃用：*OpenSSL已经取消了对SSLv2的支持。

ssl.PROTOCOL_SSLv3
选择SSL版本3作为通道加密协议。

如果使用该OPENSSL_NO_SSLv3标志编译OpenSSL，则此协议不可用。

> **警告：** SSL版本3不安全。它的使用非常令人沮丧。

*自从3.6版弃用：*OpenSSL已弃用所有版本特定的协议。使用PROTOCOL_TLS带有标志的默认协议OP_NO_SSLv3。

ssl.PROTOCOL_TLSv1
选择TLS版本1.0作为通道加密协议。

*自从3.6版弃用：*OpenSSL已弃用所有版本特定的协议。使用PROTOCOL_TLS带有标志的默认协议OP_NO_SSLv3。

ssl.PROTOCOL_TLSv1_1
选择TLS版本1.1作为通道加密协议。仅适用于openssl 1.0.1+版本。

3.4版新增功能

*自从3.6版弃用：*OpenSSL已弃用所有版本特定的协议。使用PROTOCOL_TLS带有标志的默认协议OP_NO_SSLv3。

ssl.PROTOCOL_TLSv1_2
选择TLS 1.2版作为通道加密协议。这是最现代化的版本，如果双方都能说出来，这可能是最大限度保护的最佳选择。仅适用于openssl 1.0.1+版本。

3.4版新增功能

*自从3.6版弃用：*OpenSSL已弃用所有版本特定的协议。使用PROTOCOL_TLS带有标志的默认协议OP_NO_SSLv3。

ssl.OP_ALL
为其他SSL实现中存在的各种错误提供解决方法。该选项默认设置。它不一定将标志设置为OpenSSL的SSL_OP_ALL常量。

3.2版本中的新功能

ssl.OP_NO_SSLv2
阻止SSLv2连接。该选项仅适用于与PROTOCOL_TLS。它防止对等方选择SSLv2作为协议版本。

3.2版本中的新功能

自3.6版弃用： SSLv2已弃用

ssl.OP_NO_SSLv3
防止SSLv3连接。该选项仅适用于与PROTOCOL_TLS。它防止对等方选择SSLv3作为协议版本。

3.2版本中的新功能

自3.6版弃用： SSLv3已弃用

ssl.OP_NO_TLSv1
阻止TLSv1连接。该选项仅适用于与PROTOCOL_TLS。它防止对等方选择TLSv1作为协议版本。

3.2版本中的新功能

ssl.OP_NO_TLSv1_1
阻止TLSv1.1连接。该选项仅适用于与PROTOCOL_TLS。它防止对等方选择TLSv1.1作为协议版本。仅适用于openssl 1.0.1+版本。

3.4版新增功能

ssl.OP_NO_TLSv1_2
阻止TLSv1.2连接。该选项仅适用于与PROTOCOL_TLS。它可以防止对等方选择TLSv1.2作为协议版本。仅适用于openssl 1.0.1+版本。

3.4版新增功能

ssl.OP_NO_TLSv1_3
阻止TLSv1.3连接。该选项仅适用于与PROTOCOL_TLS。它防止对等方选择TLSv1.3作为协议版本。TLS 1.3可用于OpenSSL 1.1.1或更高版本。当针对较旧版本的OpenSSL编译Python时，标志默认为*0*。

3.6.3版本中的新功能。

ssl.OP_CIPHER_SERVER_PREFERENCE
使用服务器的密码排序首选项，而不是客户端的。此选项对客户端套接字和SSLv2服务器套接字没有影响。

3.3版本的新功能

ssl.OP_SINGLE_DH_USE
防止对不同SSL会话重复使用相同的DH密钥。这提高了前向保密性，但需要更多的计算资源。该选项仅适用于服务器套接字。

3.3版本的新功能

ssl.OP_SINGLE_ECDH_USE
防止重复使用相同的ECDH密钥以用于不同的SSL会话。这提高了前向保密性，但需要更多的计算资源。该选项仅适用于服务器套接字。

3.3版本的新功能

ssl.OP_NO_COMPRESSION
禁用SSL通道上的压缩。如果应用协议支持自己的压缩方案，这很有用。

该选项仅适用于OpenSSL 1.0.0及更高版本。

3.3版本的新功能

*类*ssl.Options
enum.IntFlag OP_*常量的集合。

ssl.OP_NO_TICKET
防止客户端请求会话票据。

3.6版本中的新功能。

ssl.HAS_ALPN
OpenSSL库是否具有内置的对*应用层协议协商*TLS扩展的支持，如下所述**RFC 7301**。

3.5版本中的新功能。

ssl.HAS_ECDH
OpenSSL库是否内置了对基于椭圆曲线的Diffie-Hellman密钥交换的支持。这应该是真实的，除非该分销商明确禁用了该功能。

3.3版本的新功能

ssl.HAS_SNI
OpenSSL库是否内置了对*服务器名称指示*扩展的支持（如定义在**RFC 6066**）。

3.2版本中的新功能

ssl.HAS_NPN
NPN草案规范中描述的OpenSSL库是否内置了对*Next Protocol Negotiation*的支持。如果为true，则可以使用该方法通知您想要支持的协议。SSLContext.set_npn_protocols()

3.3版本的新功能

ssl.HAS_TLSv1_3
OpenSSL库是否具有对TLS 1.3协议的内置支持。

3.6.3版本中的新功能。

ssl.CHANNEL_BINDING_TYPES

支持的TLS通道绑定类型列表。这个列表中的字符串可以用作参数SSLSocket.get_channel_binding()。

3.3版本的新功能

ssl.OPENSSL_VERSION

解释器加载的OpenSSL库的版本字符串：

```
>>> ssl.OPENSSL_VERSION
'OpenSSL 1.0.2k  26 Jan 2017'
```

3.2版本中的新功能

ssl.OPENSSL_VERSION_INFO

包含五个整数的元组，代表有关OpenSSL库的版本信息：

```
>>> ssl.OPENSSL_VERSION_INFO
(1, 0, 2, 11, 15)
```

3.2版本中的新功能

ssl.OPENSSL_VERSION_NUMBER

OpenSSL库的原始版本号，作为单个整数：

```
>>> ssl.OPENSSL_VERSION_NUMBER
268443839
>>> hex(ssl.OPENSSL_VERSION_NUMBER)
'0x100020bf'
```

3.2版本中的新功能

ssl.ALERT_DESCRIPTION_HANDSHAKE_FAILURE
ssl.ALERT_DESCRIPTION_INTERNAL_ERROR
ALERT_DESCRIPTION_*

来自警报描述 **RFC 5246**等。该 IANA TLS警报注册表 包含此列表，并引用到自己的意思被定义RFC文档。

用作回调函数的返回值 SSLContext.set_servername_callback()。

3.4版新增功能

*类*ssl.AlertDescription

enum.IntEnum 收集ALERT_DESCRIPTION_*常量。

3.6版本中的新功能。

Purpose.SERVER_AUTH

选项create_default_context()和 SSLContext.load_default_certs()。该值表示该上下文可用于对Web服务器进行身份验证（因此，它将用于创建客户端套接字）。

3.4版新增功能

Purpose.CLIENT_AUTH

选项create_default_context()和 SSLContext.load_default_certs()。该值表示该上下文可用于对Web客户端进行身份验证（因此，它将用于创建服务器端套接字）。

3.4版新增功能

*类*ssl.SSLErrorNumber

enum.IntEnum 收集SSL_ERROR_ *常量。

3.6版本中的新功能。

18.2.2。SSL套接字

class ssl.SSLSocket（*socket.socket* ）

SSL套接字提供了以下Socket对象的方法：

- accept()
- bind()
- close()
- connect()
- detach()
- fileno()
- getpeername()， getsockname()
- getsockopt()， setsockopt()
- gettimeout()，settimeout()， setblocking()
- listen()
- makefile()
- recv()，recv_into()（但flags不允许传递非零参数）
- send()，sendall()（具有相同的限制）
- sendfile()（但os.sendfile将仅用于纯文本套接字，否则send()将被使用）
- shutdown()

但是，由于SSL（和TLS）协议在TCP的顶端有其自己的组帧，SSL套接字抽象在某些方面可能与正常的OS级套接字的规范不同。尤其请参阅 关于非阻塞套接字的说明。

通常，SSLSocket不是直接创建，而是使用该 SSLContext.wrap_socket()方法。

*在版本3.5中更改：*sendfile()添加了该方法。

*改变在3.5版本：*该shutdown()不会每个字节接收或发送时间重置套接字超时。套接字超时现在为关闭的最长总持续时间。

*自从版本3.6开始不推荐使用：*SSLSocket直接创建实例已被弃用，用于SSLContext.wrap_socket()包装套接字。

SSL套接字还具有以下附加方法和属性：

SSLSocket.read（*len = 1024*，*buffer = None* ）

从SSL套接字读取*len*个字节的数据并将结果作为bytes实例返回。如果*缓冲器*被指定，然后读入到缓冲液代替，并返回读出的字节数。

提升SSLWantReadError或者SSLWantWriteError如果套接字是 非阻塞的并且读取会阻塞。

因为在任何时候都可以进行重新协商，所以调用read()也会导致写入操作。

*版本3.5中更改：*每次接收或发送字节时，套接字超时不再重置。套接字超时现在达到最大总持续时间以读取*len* 字节。

*自从3.6版弃用：*recv()代替使用read()。

SSLSocket.**write**(*buf*)

将*buf*写入SSL套接字并返回写入的字节数。所述 *的buf*参数必须是支撑所述缓冲器接口的对象。

引发SSLWantReadError或者SSLWantWriteError如果套接字是 非阻塞的并且写入将被阻塞。

因为在任何时候都可以进行重新协商，所以调用write()也会导致读取操作。

*版本3.5中更改：*每次接收或发送字节时，套接字超时不再重置。套接字超时现在是写入*buf的*最大总持续时间。

*自从3.6版弃用：*send()代替使用write()。

注意： 的read()和write()方法是读取和写入未加密的，应用级数据和解密/加密它以加密的，电线级数据的低级别的方法。这些方法需要一个活动的SSL连接，即握手已完成并且SSLSocket.unwrap()未被调用。

通常你应该使用套接字API方法 recv() ， send()而不是这些方法。

SSLSocket.**do_handshake**( )

执行SSL设置握手。

*在版本3.4中更改：*match_hostname()当check_hostname套接字的属性 context为true 时，握手方法也会执行。

*版本3.5中更改：*每次接收或发送字节时，套接字超时不再重置。套接字超时现在是握手最长的总持续时间。

SSLSocket.**getpeercert**(*binary_form = False*)

如果连接的另一端没有对等体的证书，则返回None。如果SSL握手尚未完成，请提出ValueError。

如果binary_form参数是False ，并且从对等端接收到证书，则此方法返回一个dict实例。如果证书未通过验证，则字典为空。如果证书已经过验证，它会返回一个包含几个密钥的字典，其中包括subject 证书颁发issuer 的主体和颁发证书的主体。如果证书包含“ *主题备用名称*扩展的实例（请参阅**RFC 3280** ），subjectAltName字典中也会有一个关键字。

的subject和issuer字段是包含在该证书的数据结构给出了各个场相对专有名称（的RDN）的序列的元组，并且每个RDN是名称-值对的序列。这是一个真实世界的例子：

```
{'issuer': ((('countryName', 'IL'),),
            (('organizationName', 'StartCom Ltd.'),),
            (('organizationalUnitName',
              'Secure Digital Certificate Signing'),),
            (('commonName',
              'StartCom Class 2 Primary Intermediate Server CA'),)),
 'notAfter': 'Nov 22 08:15:19 2013 GMT',
 'notBefore': 'Nov 21 03:09:52 2011 GMT',
 'serialNumber': '95F0',
 'subject': ((('description', '571208-SLe257oHY9fVQ07Z'),),
             (('countryName', 'US'),),
             (('stateOrProvinceName', 'California'),),
             (('localityName', 'San Francisco'),),
             (('organizationName', 'Electronic Frontier Foundation, Inc.'),),
             (('commonName', '*.eff.org'),),
             (('emailAddress', 'hostmaster@eff.org'),)),
 'subjectAltName': (('DNS', '*.eff.org'), ('DNS', 'eff.org')),
 'version': 3}
```

注意： 要验证特定服务的证书，可以使用该 match_hostname() 功能。

如果binary_form参数是True，并且提供了证书，则此方法将整个证书的DER编码形式作为一系列字节返回，或者None如果对等体未提供证书。对等体是否提供证书取决于SSL套接字的角色：

- 对于客户端SSL套接字，服务器将始终提供证书，无论是否需要验证;
- 对于服务器SSL套接字，客户端只会在服务器请求时提供证书; 因此getpeercert() 将返回 None，如果你使用CERT_NONE（而不是 CERT_OPTIONAL或CERT_REQUIRED）。

*在版本3.2中更改：*返回的字典包括其他项目，如issuer 和notBefore。

*在版本3.4中更改：*ValueError在握手未完成时引发。返回的字典包括额外采用X509v3扩展项目，如crlDistributionPoints，caIssuers和OCSP的URI。

SSLSocket.**cipher**()

返回包含正在使用的密码名称，定义其使用的SSL协议版本以及正在使用的密码位数的三值元组。如果没有建立连接，则返回None。

SSLSocket.**shared_ciphers**()

返回握手期间客户端共享的密码列表。返回列表的每个条目都是一个三值元组，其中包含密码的名称，定义其使用的SSL协议版本以及密码使用的密码位数。 shared_ciphers() 返回 None如果没有连接已经建立或者套接字的客户机套接字。

3.5版本中的新功能。

SSLSocket.**compression**()

返回用作字符串的压缩算法，或者None 连接未被压缩。

如果更高级别的协议支持自己的压缩机制，则可以使用OP_NO_COMPRESSION禁用SSL级别的压缩。

3.3版本的新功能

SSLSocket.**get_channel_binding**(*cb_type="tls-unique"*)

获取当前连接的通道绑定数据，作为字节对象。返回 None如果没有连接或握手尚未完成。

所述*cb_type*参数允许所需的信道绑定类型的选择。 CHANNEL_BINDING_TYPES列表中列出了有效的通道绑定类型 。目前只有'tls-unique'通道绑定，由...定义**RFC 5929**支持。 ValueError如果请求不支持的通道绑定类型，则会引发此问题。

3.3版本的新功能

SSLSocket.**selected_alpn_protocol**()

返回在TLS握手期间选择的协议。如果 SSLContext.set_alpn_protocols()未被调用，如果另一方不支持ALPN，如果此套接字不支持任何客户端建议的协议，或者握手尚未发生，None则返回。

3.5版本中的新功能。

SSLSocket.**selected_npn_protocol**()

返回在TLS / SSL握手期间选择的更高级协议。如果SSLContext.set_npn_protocols()没有被调用，或者对方不支持NPN，或者握手尚未发生，则返回None。

3.3版本的新功能

SSLSocket.**unwrap**()

执行SSL关闭握手，从基础套接字移除TLS层，并返回底层套接字对象。这可以用于从连接上的加密操作转到未加密。返回的套接字应始终用于与连接的另一端进一步通信，而不是原始套接字。

SSLSocket.**version**()

将连接协商的实际SSL协议版本作为字符串返回，或者None没有建立安全连接。在撰写本文时，可能的返回值包括"SSLv2"， "SSLv3"，"TLSv1"，"TLSv1.1"和"TLSv1.2"。最近的OpenSSL版本可能会定义更多的返回值。

3.5版本中的新功能。

SSLSocket.**pending**()

返回可用于读取的已解密字节数，等待连接。

SSLSocket.**context**

SSLContext这个SSL套接字所绑定的对象。如果SSL套接字是使用顶级wrap_socket()函数（而不是SSLContext.wrap_socket()）创建的，则这是为此SSL套接字创建的自定义上下文对象。

3.2版本中的新功能

SSLSocket.**server_side**

一个布尔值，True用于服务器端套接字和False客户端套接字。

3.2版本中的新功能

SSLSocket.**server_hostname**

服务器的主机名：strtype，或者None用于服务器端套接字，或者如果主机名未在构造器中指定。

3.2版本中的新功能

SSLSocket.**session**

在SSLSession此SSL连接。TLS握手执行后，会话可用于客户端和服务器端套接字。对于客户端套接字，可以do_handshake()在调用之前设置 会话以重用会话。

3.6版本中的新功能。

SSLSocket.**session_reused**

3.6版本中的新功能。

18.2.3。SSL上下文

3.2版本中的新功能

SSL上下文保存比单个SSL连接更长的各种数据，例如SSL配置选项，证书和私钥。它还管理服务器端套接字的SSL会话缓存，以加速来自相同客户端的重复连接。

class ssl.**SSLContext**(*protocol = PROTOCOL_TLS*)

创建一个新的SSL上下文。你可以传递*协议*，它必须是PROTOCOL_*本模块中定义的常量之一。 PROTOCOL_TLS目前推荐用于最大的互操作性和默认值。

> **也可以看看：** create_default_context()让ssl模块为特定目的选择安全设置。

*版本3.6中已更改：*使用安全的默认值创建上下文。选项 OP_NO_COMPRESSION，OP_CIPHER_SERVER_PREFERENCE，OP_SINGLE_DH_USE，OP_SINGLE_ECDH_USE，OP_NO_SSLv2（除了PROTOCOL_SSLv2），及OP_NO_SSLv3（除PROTOCOL_SSLv3）为默认设置。初始密码套件列表只包含HIGH 密码，没有NULL 密码，没有MD5 密码（除PROTOCOL_SSLv2）。

SSLContext 对象具有以下方法和属性：

SSLContext.**cert_store_stats**()

获取有关已加载的X.509证书数量的统计信息，以字典形式标记为CA证书和证书撤销列表的X.509证书数。

具有一个CA证书和另一个证书的上下文示例：

```
>>> context.cert_store_stats()
{'crl': 0, 'x509_ca': 1, 'x509': 2}
```

3.4版新增功能

SSLContext.load_cert_chain(*certfile*，*keyfile* = *None*，*password* = *None*)

加载私钥和相应的证书。所述*certFile中*字符串必须的路径PEM格式包含证书以及任何号码，以建立证书的真实性所需要的CA证书的单个文件。该*密钥文件*的字符串，如果存在的话，必须指向包含私有密钥的文件，否则私钥将采取*certFile中*也是如此。见的讨论 证书关于证书是如何被存储在信息*certFile中*。

该*密码*参数可以是一个函数来调用用于解密私钥获取密码。只有当私钥被加密并且密码是必要的时才会被调用。它将被调用，没有参数，它应该返回一个字符串，字节或bytearray。如果返回值是一个字符串，在使用它解密密钥之前，它将被编码为UTF-8。或者，字符串，字节或bytearray值可直接作为*密码*参数提供。如果私钥未加密并且不需要密码，它将被忽略。

如果未指定*password*参数并且需要密码，则将使用OpenSSL的内置密码提示机制以交互方式提示用户输入密码。

一个SSLError如果私钥不与证书相匹配上升。

*在版本3.3中更改：*新的可选参数*密码*。

SSLContext.load_default_certs(*purpose* = *Purpose.SERVER_AUTH*)

从默认位置加载一组默认的“证书颁发机构”（CA）证书。在Windows加载从CA证书CA和ROOT系统存储。在其他系统上调用 SSLContext.set_default_verify_paths()。将来，该方法也可能会从其他位置加载CA证书。

该*目的*标志指定类型的CA证书加载什么。默认设置Purpose.SERVER_AUTH加载证书，这些证书被标记和信任用于TLS Web服务器身份验证（客户端套接字）。 Purpose.CLIENT_AUTH在服务器端加载用于客户端证书验证的CA证书。

3.4版新增功能

SSLContext.load_verify_locations(*cafile* = *None*，*capath* = *None*，*cadata* = *None*)

加载一组“认证权威”一起使用时，以验证其他对等端的证书（CA）证书的verify_mode比其他 CERT_NONE。必须至少指定一个*cafile*或*capath*。

此方法还可以加载PEM或DER格式的证书吊销列表（CRL）。为了使用CRL，SSLContext.verify_flags 必须正确配置。

的*凭证档案错误*串，如果存在的话，是路径到PEM格式级联CA证书的文件。有关如何在此文件中安排证书的更多信息，请参阅证书的讨论 。

的*capath*串，如果存在的话，是路径到包含PEM格式几个CA证书，以下内容的一个目录的OpenSSL特定布局。

的*cadata*对象，如果存在的话，或者是一个或多个PEM编码证书的ASCII字符串或类字节对象 DER编码的证书。像使用*capath*一样，PEM编码证书周围的额外行被忽略，但至少必须存在一个证书。

*在版本3.4中更改：*新的可选参数*cadata*

SSLContext.**get_ca_certs**(*binary_form = False*)

获取加载的“证书颁发机构”（CA）证书列表。如果 binary_form参数是False每个列表项是一个类似输出的字典SSLSocket.getpeercert()。否则，该方法将返回DER编码证书的列表。除非通过SSL连接请求和加载证书，否则返回的列表不包含来自*Capath*的证书。

> **注意：** 在capath目录中的证书不会被加载，除非它们至少被使用过一次。

3.4版新增功能

SSLContext.**get_ciphers**( )

获取启用的密码列表。该列表按密码优先顺序排列。看SSLContext.set_ciphers()。

例：

```
>>> ctx = ssl.SSLContext(ssl.PROTOCOL_SSLv23)
>>> ctx.set_ciphers('ECDHE+AESGCM:!ECDSA')
>>> ctx.get_ciphers()  # OpenSSL 1.0.x
[{'alg_bits': 256,
  'description': 'ECDHE-RSA-AES256-GCM-SHA384 TLSv1.2 Kx=ECDH     Au=RSA  '
                 'Enc=AESGCM(256) Mac=AEAD',
  'id': 50380848,
  'name': 'ECDHE-RSA-AES256-GCM-SHA384',
  'protocol': 'TLSv1/SSLv3',
  'strength_bits': 256},
 {'alg_bits': 128,
  'description': 'ECDHE-RSA-AES128-GCM-SHA256 TLSv1.2 Kx=ECDH     Au=RSA  '
                 'Enc=AESGCM(128) Mac=AEAD',
  'id': 50380847,
  'name': 'ECDHE-RSA-AES128-GCM-SHA256',
  'protocol': 'TLSv1/SSLv3',
  'strength_bits': 128}]
```

在OpenSSL 1.1和更新版本中，密码字典包含更多字段::

```
>>> ctx.get_ciphers()  # OpenSSL 1.1+
[{'aead': True,
  'alg_bits': 256,
  'auth': 'auth-rsa',
  'description': 'ECDHE-RSA-AES256-GCM-SHA384 TLSv1.2 Kx=ECDH     Au=RSA  '
                 'Enc=AESGCM(256) Mac=AEAD',
  'digest': None,
  'id': 50380848,
  'kea': 'kx-ecdhe',
  'name': 'ECDHE-RSA-AES256-GCM-SHA384',
  'protocol': 'TLSv1.2',
  'strength_bits': 256,
  'symmetric': 'aes-256-gcm'},
 {'aead': True,
  'alg_bits': 128,
  'auth': 'auth-rsa',
  'description': 'ECDHE-RSA-AES128-GCM-SHA256 TLSv1.2 Kx=ECDH     Au=RSA  '
                 'Enc=AESGCM(128) Mac=AEAD',
```

```
 'digest': None,
 'id': 50380847,
 'kea': 'kx-ecdhe',
 'name': 'ECDHE-RSA-AES128-GCM-SHA256',
 'protocol': 'TLSv1.2',
 'strength_bits': 128,
 'symmetric': 'aes-128-gcm'}]
```

可用性：OpenSSL 1.0.2+

3.6版本中的新功能。

SSLContext.**set_default_verify_paths**( )

从构建OpenSSL库时定义的文件系统路径加载一组默认的“证书颁发机构”（CA）证书。不幸的是，没有简单的方法来知道这种方法是否成功：如果没有找到证书，则不返回错误。但是，当OpenSSL库作为操作系统的一部分提供时，它可能会被正确配置。

SSLContext.**set_ciphers**(*密码*)

设置使用此上下文创建的套接字的可用密码。它应该是OpenSSL密码列表格式的字符串。如果不能选择密码（因为编译时选项或其他配置禁止使用所有指定的密码），SSLError将会引发一个密码。

> **注意：** 当连接时，SSLSocket.cipher()SSL套接字的方法会给出当前选定的密码。

SSLContext.**set_alpn_protocols**(*协议*)

指定套接字应在SSL / TLS握手期间通告的协议。它应该是一个ASCII字符串列表，比如按照首选项排序。协议的选择将在握手过程中发生，并根据情况进行播放['http/1.1', 'spdy/2']**RFC 7301**。在握手成功后，该SSLSocket.selected_alpn_protocol()方法将返回商定的协议。

NotImplementedError如果HAS_ALPN是False，此方法将会引发。

OpenSSL 1.1.0至1.1.0e将中止握手并SSLError 在双方支持ALPN但不能达成协议时提高。1.1.0f +的行为如同1.0.2，SSLSocket.selected_alpn_protocol()返回None。

3.5版本中的新功能。

SSLContext.**set_npn_protocols**(*协议*)

指定套接字应在SSL / TLS握手期间通告的协议。它应该是一个字符串列表，例如，按照首选项排序。协议的选择将在握手期间进行，并将根据NPN草案规范进行播放。在握手成功后，该方法将返回商定的协议。['http/1.1', 'spdy/2']SSLSocket.selected_npn_protocol()

NotImplementedError如果HAS_NPN是False，此方法将会引发。

3.3版本的新功能

SSLContext.**set_servername_callback**(*server_name_callback*)

注册一个回调函数，当TLS客户端指定服务器名称指示时，SSL / TLS服务器收到TLS客户端Hello握手消息后将调用该函数。服务器名称指示机制在中指定**RFC 6066**第3节 - 服务器

名称指示。

每次只能设置一个回调SSLContext。如果*server_name_callback* 是None那么回调被禁用。随后调用此函数将禁用先前注册的回调。

回调函数*server_name_callback*将用三个参数调用; 第一个是ssl.SSLSocket第二个是表示客户端打算通信的服务器名称的字符串（或者None如果TLS客户端Hello不包含服务器名称），第三个参数是原始的SSLContext。服务器名称参数是IDNA解码的服务器名称。

一个典型的使用这个回调的是将改变ssl.SSLSocket的 SSLSocket.context属性类型的新对象 SSLContext表示相匹配的服务器名的证书链。

由于TLS连接的早期谈判阶段，只有有限的方法和属性可用，例如SSLSocket.selected_alpn_protocol()和SSLSocket.context。 SSLSocket.getpeercert()，SSLSocket.getpeercert()， SSLSocket.cipher()和SSLSocket.compress()方法要求TLS连接已经渐渐超越了TLS客户端问候，并因此将不包含返回有意义的值，也不能被安全地调用。

该*server_name_callback*函数必须返回None允许TLS协商继续。如果需要TLS失败，ALERT_DESCRIPTION_*可以返回一个常量 。其他返回值将导致TLS致命错误ALERT_DESCRIPTION_INTERNAL_ERROR。

如果服务器名称上存在IDNA解码错误，TLS连接将终止ALERT_DESCRIPTION_INTERNAL_ERROR并向客户端发送致命的TLS警报消息。

如果从*server_name_callback*函数引发异常，则TLS连接将终止并显示致命的TLS警报消息ALERT_DESCRIPTION_HANDSHAKE_FAILURE。

NotImplementedError如果OpenSSL库在构建时定义了OPENSSL_NO_TLSEXT，则会引发此方法。

3.4版新增功能

SSLContext.load_dh_params(*dhfile*)

加载Diffie-Hellman（DH）密钥交换的密钥生成参数。使用DH密钥交换以牺牲计算资源（服务器和客户端）为代价提高了前向保密性。所述*dhfile*参数应的路径包含PEM格式DH参数的文件。

该设置不适用于客户端套接字。您也可以使用该 OP_SINGLE_DH_USE选项来进一步提高安全性。

3.3版本的新功能

SSLContext.set_ecdh_curve(*curve_name*)

为基于椭圆曲线的Diffie-Hellman（ECDH）密钥交换设置曲线名称。ECDH比常规DH显着更快，而且可以说是安全的。所述*curve_name*参数应该是字符串描述公知的椭圆曲线，例如prime256v1用于一个广泛支持曲线。

该设置不适用于客户端套接字。您也可以使用该 OP_SINGLE_ECDH_USE选项来进一步提高安全性。

如果HAS_ECDH是，该方法不可用False。

3.3版本的新功能

也可以看看:

SSL / TLS和完美的前向保密
文森特伯纳特。

SSLContext.**wrap_socket**(*sock*, *server_side = False*, *do_handshake_on_connect = True*, *suppress_ragged_eofs = True*, *server_hostname = None*, *session = None*)

总结现有的Python插座*袜子*，并返回一个SSLSocket 对象。 *袜子*必须是一个SOCK_STREAM插座; 其他套接字类型不受支持。

返回的SSL套接字与上下文，其设置和证书相关联。参数*server_side*，*do_handshake_on_connect* 和*suppress_ragged_eofs*具有与顶级wrap_socket()函数中相同的含义。

在客户端连接上，可选参数*server_hostname*指定我们要连接的服务的主机名。这允许单个服务器使用不同的证书托管多个基于SSL的服务，这与HTTP虚拟主机非常相似。如果*server_side*为true，则指定*server_hostname*将引发一次。ValueError

会议，见session。

*在版本3.5中更改：*即使OpenSSL没有SNI，也始终允许传递server_hostname。

*在版本3.6中更改：会话*参数已添加。

SSLContext.**wrap_bio**(*incoming*, *outgoing*, *server_side = False*, *server_hostname = None*, *session = None*)

SSLObject通过包装*传入*和*传出*的BIO对象*来*创建一个新实例。SSL例程将从传入的BIO读取输入数据并将数据写入传出的BIO。

该*server_side*，*server_hostname*和*会话*参数具有相同的含义SSLContext.wrap_socket()。

*在版本3.6中更改：会话*参数已添加。

SSLContext.**session_stats**()

获取有关由此上下文创建或管理的SSL会话的统计信息。返回的字典将每条信息的名称映射到它们的数字值。例如，以下是创建上下文后会话缓存中的命中和未命中总数：

```
>>> stats = context.session_stats()
>>> stats['hits'], stats['misses']
(0, 0)
```

SSLContext.**check_hostname**

是否使用match_hostname()in 匹配peer cert的主机名SSLSocket.do_handshake()。上下文的verify_mode必须设置为CERT_OPTIONAL或 CERT_REQUIRED，你必须通过*server_hostname*到wrap_socket()为了匹配的主机名。

例：

```
import socket, ssl

context = ssl.SSLContext()
context.verify_mode = ssl.CERT_REQUIRED
context.check_hostname = True
context.load_default_certs()

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
ssl_sock = context.wrap_socket(s, server_hostname='www.verisign.com')
ssl_sock.connect(('www.verisign.com', 443))
```

3.4版新增功能

> **注意：** 该功能需要OpenSSL 0.9.8f或更新版本。

SSLContext.**options**

表示在此上下文中启用的一组SSL选项的整数。默认值是OP_ALL，但您可以指定其他选项，例如OP_NO_SSLv2通过将它们组合在一起。

> **注意：** 对于0.9.8m以上版本的OpenSSL，只能设置选项，而不能清除它们。试图清除一个选项（通过重置相应的位）将引发一个ValueError。

*版本3.6中更改：*SSLContext.options返回Options标志：

```
>>> ssl.create_default_context().options
<Options.OP_ALL|OP_NO_SSLv3|OP_NO_SSLv2|OP_NO_COMPRESSION: 2197947391>
```

SSLContext.**protocol**

在构建上下文时选择的协议版本。该属性是只读的。

SSLContext.**verify_flags**

证书验证操作的标志。您可以VERIFY_CRL_CHECK_LEAF通过将它们组合在一起来设置标志。默认情况下，OpenSSL既不需要也不需要验证证书撤销列表（CRL）。仅适用于openssl 0.9.8+版本。

3.4版新增功能

*版本3.6中更改：*SSLContext.verify_flags返回VerifyFlags标志：

```
>>> ssl.create_default_context().verify_flags
<VerifyFlags.VERIFY_X509_TRUSTED_FIRST: 32768>
```

SSLContext.**verify_mode**

是否尝试验证其他同行的证书以及验证是否失败时的行为方式。该属性必须是一个CERT_NONE，CERT_OPTIONAL或CERT_REQUIRED。

*在版本3.6中更改：*SSLContext.verify_mode返回VerifyMode枚举：

```
>>> ssl.create_default_context().verify_mode
<VerifyMode.CERT_REQUIRED: 2>
```

18.2.4。证书

一般而言，证书是公钥/私钥系统的一部分。在这个系统中，每个*委托人*（可能是一台机器，一个人或一个组织）被分配一个唯一的两部分加密密钥。密钥的一部分是公共密钥，称为*公钥*。另一部分保密，称为*私钥*。这两部分是相关的，因为如果您使用其中一个部分对消息进行加密，则可以使用其他部分对其进行解密，而**只能**使用其他部分进行解密。

证书包含有关两个委托人的信息。它包含*主题*的名称和主题的公钥。它还包含第二位负责*人发行人*的声明，该主体是他自称的主体，而且这确实是主体的公钥。发行人的声明与发行人的私钥签署，只有发行人知道。但是，任何人都可以通过查找发行人的公钥来验证发行人的声明，使用它解密声明，并将其与证书中的其他信息进行比较。证书还包含有效时间段的信息。这表示为两个字段，称为“notBefore”和“notAfter”。

在使用证书的Python中，客户端或服务器可以使用证书来证明他们是谁。网络连接的另一端也可能需要生成一个证书，并且该证书可以验证，以满足需要验证的客户端或服务器。如果验证失败，可以将连接尝试设置为引发异常。验证是由底层OpenSSL框架自动完成的; 该应用程序不需要关注其机制。但是，应用程序通常需要提供证书集以允许此过程发生。

Python使用文件来包含证书。他们应该被格式化为“PEM”（请参阅**RFC 1422**），它是一个用标题行和页脚行包装的base-64编码形式：

```
-----BEGIN CERTIFICATE-----
... (certificate in base64 PEM encoding) ...
-----END CERTIFICATE-----
```

18.2.4.1。证书链

包含证书的Python文件可以包含一系列证书，有时称为*证书链*。这个链应该与校长谁“是”客户端或服务器，然后该证书的颁发的证书，然后对发行人的证书的特定证书启动*该*证书，依此类推，直到链，直到你获得*自签名*证书，即具有相同主题和颁发者的*证书*，有时称为*根证书*。证书只能在证书文件中连接在一起。例如，假设我们从服务器证书到签署服务器证书的证书颁发机构的证书有三个证书链，并且颁发给颁发证书颁发机构证书的机构的根证书：

```
-----BEGIN CERTIFICATE-----
... (certificate for your server)...
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
... (the certificate for the CA)...
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
... (the root certificate for the CA's issuer)...
-----END CERTIFICATE-----
```

18.2.4.2。CA证书

如果您打算要求验证连接证书的另一端，则需要提供一个“CA证书”文件，并填写您愿意信任的每个颁发机构的证书链。同样，这个文件只包含连接在一起的这些链。为了验证，Python将使用它在匹配的文件中找到的第一个链。平台的证书文件可以通过调用使用SSLContext.load_default_certs()，这是自动完成的create_default_context()。

18.2.4.3。组合密钥和证书

私钥通常与证书存储在同一个文件中；在这种情况下，只有certfile参数SSLContext.load_cert_chain()和wrap_socket()需要传递。如果私钥与证书一起存储，它应该位于证书链中的第一个证书之前：

```
-----BEGIN RSA PRIVATE KEY-----
... (private key in base64 encoding) ...
-----END RSA PRIVATE KEY-----
-----BEGIN CERTIFICATE-----
... (certificate in base64 PEM encoding) ...
-----END CERTIFICATE-----
```

18.2.4.4。自签证书

如果要创建提供SSL加密连接服务的服务器，则需要获取该服务的证书。获取适当证书的方式很多，例如从认证机构购买证书。另一种常见做法是生成一个自签名证书。最简单的方法是使用OpenSSL包，使用如下所示：

```
% openssl req -new -x509 -days 365 -nodes -out cert.pem -keyout cert.pem
Generating a 1024 bit RSA private key
.......++++++
..............................++++++
writing new private key to 'cert.pem'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:MyState
Locality Name (eg, city) []:Some City
Organization Name (eg, company) [Internet Widgits Pty Ltd]:My Organization, Inc.
Organizational Unit Name (eg, section) []:My Group
Common Name (eg, YOUR name) []:myserver.mygroup.myorganization.com
Email Address []:ops@myserver.mygroup.myorganization.com
%
```

自签名证书的缺点是它是它自己的根证书，并且没有人会在已知（和可信）根证书的缓存中拥有它。

18.2.5。示例

18.2.5.1。测试SSL支持

要测试Python安装中是否存在SSL支持，用户代码应使用以下习惯用法：

```
try:
    import ssl
except ImportError:
    pass
else:
    ...  # do something that requires SSL support
```

18.2.5.2。客户端操作

本示例使用推荐的客户端套接字安全设置创建SSL上下文，其中包括自动证书验证：

```
>>> context = ssl.create_default_context()
```

如果您更喜欢自己调整安全设置，则可以从头创建一个上下文（但要注意，您可能无法正确设置设置）：

```
>>> context = ssl.SSLContext()
>>> context.verify_mode = ssl.CERT_REQUIRED
>>> context.check_hostname = True
>>> context.load_verify_locations("/etc/ssl/certs/ca-bundle.crt")
```

（这段代码假设您的操作系统放置了所有CA证书/etc/ssl/certs/ca-bundle.crt;如果没有，您将得到一个错误并且必须调整位置）

当您使用环境连接到服务器时，CERT_REQUIRED 验证服务器证书：它确保服务器证书使用其中一个CA证书进行签名，并检查签名的正确性：

```
>>> conn = context.wrap_socket(socket.socket(socket.AF_INET),
...                            server_hostname="www.python.org")
>>> conn.connect(("www.python.org", 443))
```

您可以取回证书：

```
>>> cert = conn.getpeercert()
```

目视检查显示证书确定了所需的服务（即HTTPS主机www.python.org）：

```
>>> pprint.pprint(cert)
{'OCSP': ('http://ocsp.digicert.com',),
 'caIssuers': ('http://cacerts.digicert.com/DigiCertSHA2ExtendedValidationServerCA.cr
 'crlDistributionPoints': ('http://crl3.digicert.com/sha2-ev-server-g1.crl',
                           'http://crl4.digicert.com/sha2-ev-server-g1.crl'),
 'issuer': ((('countryName', 'US'),),
            (('organizationName', 'DigiCert Inc'),),
            (('organizationalUnitName', 'www.digicert.com'),),
            (('commonName', 'DigiCert SHA2 Extended Validation Server CA'),)),
```

```
 'notAfter': 'Sep  9 12:00:00 2016 GMT',
 'notBefore': 'Sep  5 00:00:00 2014 GMT',
 'serialNumber': '01BB6F00122B177F36CAB49CEA8B6B26',
 'subject': ((('businessCategory', 'Private Organization'),),
             (('1.3.6.1.4.1.311.60.2.1.3', 'US'),),
             (('1.3.6.1.4.1.311.60.2.1.2', 'Delaware'),),
             (('serialNumber', '3359300'),),
             (('streetAddress', '16 Allen Rd'),),
             (('postalCode', '03894-4801'),),
             (('countryName', 'US'),),
             (('stateOrProvinceName', 'NH'),),
             (('localityName', 'Wolfeboro,'),),
             (('organizationName', 'Python Software Foundation'),),
             (('commonName', 'www.python.org'),)),
 'subjectAltName': (('DNS', 'www.python.org'),
                    ('DNS', 'python.org'),
                    ('DNS', 'pypi.org'),
                    ('DNS', 'docs.python.org'),
                    ('DNS', 'testpypi.org'),
                    ('DNS', 'bugs.python.org'),
                    ('DNS', 'wiki.python.org'),
                    ('DNS', 'hg.python.org'),
                    ('DNS', 'mail.python.org'),
                    ('DNS', 'packaging.python.org'),
                    ('DNS', 'pythonhosted.org'),
                    ('DNS', 'www.pythonhosted.org'),
                    ('DNS', 'test.pythonhosted.org'),
                    ('DNS', 'us.pycon.org'),
                    ('DNS', 'id.python.org')),
 'version': 3}
```

现在SSL通道已建立并且证书已通过验证，您可以继续与服务器通话：

```
>>> conn.sendall(b"HEAD / HTTP/1.0\r\nHost: linuxfr.org\r\n\r\n")
>>> pprint.pprint(conn.recv(1024).split(b"\r\n"))
[b'HTTP/1.1 200 OK',
 b'Date: Sat, 18 Oct 2014 18:27:20 GMT',
 b'Server: nginx',
 b'Content-Type: text/html; charset=utf-8',
 b'X-Frame-Options: SAMEORIGIN',
 b'Content-Length: 45679',
 b'Accept-Ranges: bytes',
 b'Via: 1.1 varnish',
 b'Age: 2188',
 b'X-Served-By: cache-lcy1134-LCY',
 b'X-Cache: HIT',
 b'X-Cache-Hits: 11',
 b'Vary: Cookie',
 b'Strict-Transport-Security: max-age=63072000; includeSubDomains',
 b'Connection: close',
 b'',
 b'']
```

请参阅下面关于安全性考虑的讨论。

18.2.5.3。服务器端操作

对于服务器操作，通常您需要拥有服务器证书和私钥，每个证书都在一个文件中。您将首先创建一个持有密钥和证书的上下文，以便客户可以检查您的真实性。然后你将打开一个套接字，将它绑定到一个端口，调用listen()它并开始等待客户端连接：

```
import socket, ssl

context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
context.load_cert_chain(certfile="mycertfile", keyfile="mykeyfile")

bindsocket = socket.socket()
bindsocket.bind(('myaddr.mydomain.com', 10023))
bindsocket.listen(5)
```

当客户端连接时，您将调用accept()套接字从另一端获取新套接字，并使用上下文的SSLContext.wrap_socket()方法为连接创建服务器端SSL套接字：

```
while True:
    newsocket, fromaddr = bindsocket.accept()
    connstream = context.wrap_socket(newsocket, server_side=True)
    try:
        deal_with_client(connstream)
    finally:
        connstream.shutdown(socket.SHUT_RDWR)
        connstream.close()
```

然后，您将从该数据中读取数据connstream并执行相关操作，直到完成客户端（或客户端已与您完成）为止：

```
def deal_with_client(connstream):
    data = connstream.recv(1024)
    # empty data means the client is finished with us
    while data:
        if not do_something(connstream, data):
            # we'll assume do_something returns False
            # when we're finished with client
            break
        data = connstream.recv(1024)
    # finished with client
```

然后返回监听新客户端连接（当然，真正的服务器可能会在单独的线程中处理每个客户端连接，或者将套接字置于非阻塞模式并使用事件循环）。

18.2.6。在非阻塞套接字的注意事项

在非阻塞模式下，SSL套接字的行为与普通套接字略有不同。使用非阻塞套接字时，需要注意以下几点：

- 如果I / O操作阻塞，大多数SSLSocket方法都会引发SSLWantWriteError或者SSLWantReadError而不是BlockingIOError。SSLWantReadError如果需要对底层套接字进

行读操作，并且SSLWantWriteError对底层套接字进行写操作，则会引发该操作。需要注意的是试图 *写*至SSL套接字可能需要*读取*从底层插座中，并试图*读取*从SSL套接字可能要求之前*写入*到底层插座。

*在版本3.5中更改：*在较早的Python版本中，该SSLSocket.send()方法返回零而不是提升SSLWantWriteError或 SSLWantReadError。

- 调用select()告诉您可以从（或写入）读取操作系统级套接字，但并不意味着上层SSL层有足够的数据。例如，只有部分SSL帧可能已经到达。因此，您必须准备好处理SSLSocket.recv() 和SSLSocket.send()失败，并在接到另一个电话后重试 select()。

- 相反，由于SSL层具有自己的组帧，因此SSL套接字可能仍然有数据可供读取而不 select() 知道它。因此，您应该先拨打电话 SSLSocket.recv()以排除任何可能存在的数据，然后在select()必要时仅阻止通话。

  （当然，类似的规定适用于使用其他基元 poll()或selectors模块中的基元）

- SSL握手本身将是非阻塞的：该 SSLSocket.do_handshake()方法必须重试直到它成功返回。这是一个select()用于等待套接字准备就绪的概要：

```
while True:
    try:
        sock.do_handshake()
        break
    except ssl.SSLWantReadError:
        select.select([sock], [], [])
    except ssl.SSLWantWriteError:
        select.select([], [sock], [])
```

也可以看看： 该asyncio模块支持非阻塞SSL套接字并提供更高级别的API。它使用selectors模块和句柄轮询事件SSLWantWriteError，SSLWantReadError以及 BlockingIOError例外。它也以异步方式运行SSL握手。

18.2.7。内存BIO支持

3.5版本中的新功能。

自从Python 2.6引入SSL模块以来，SSLSocket 该类提供了两个相关但不同的功能区域：

- SSL协议处理
- 网络IO

网络IO API与其提供的相同socket.socket，SSLSocket也从中继承。这使得SSL套接字可以用作常规套接字的直接替代品，使得向现有应用程序添加SSL支持变得非常容易。

SSL协议处理和网络IO的组合通常效果很好，但有些情况下却没有。一个例子是异步IO框架，它想要使用socket.socket 与内部OpenSSL套接字IO例程假定的“选择/轮询文件描述符”（基于准备就绪）模型不同的IO多路复用模型。这主要与Windows这样的平台无关，因为这种模式效率不高。为此，提供了一个缩小的SSLSocket被调用范围变体SSLObject。

*类*ssl.SSLObject

SSLSocket表示不包含任何网络IO方法的SSL协议实例的缩小范围变体。这个类通常由想通过内存缓冲区为SSL实现异步IO的框架作者使用。

该类在OpenSSL实现的低级SSL对象上实现接口。该对象捕获SSL连接的状态，但不提供任何网络IO本身。IO需要通过独立的“BIO”对象执行，这些对象是OpenSSL的IO抽象层。

一个SSLObject实例可以使用被创建 wrap_bio() 方法。此方法将创建 SSLObject实例并将其绑定到一对BIO。该*传入* BIO用于从Python的数据传递到所述SSL协议实例，而 *传出* BIO用于围绕传递数据的其他方式。

以下方法可用：

- context
- server_side
- server_hostname
- session
- session_reused
- read()
- write()
- getpeercert()
- selected_npn_protocol()
- cipher()
- shared_ciphers()
- compression()
- pending()
- do_handshake()
- unwrap()
- get_channel_binding()

与SSLSocket此相比，此对象缺少以下功能：

- 任何形式的网络IO; recv()并且send()只能读写底层MemoryBIO缓冲区。
- 没有*do_handshake_on_connect*机器。您必须始终手动调用do_handshake()才能开始握手。
- 没有处理*suppress_ragged_eofs*。所有违反协议的文件结束条件都会通过SSLEOFError异常报告。
- 方法unwrap()调用不会返回任何内容，这与返回底层套接字的SSL套接字不同。
- 该*server_name_callback*传递给回调 SSLContext.set_servername_callback()将得到一个SSLObject ，而不是一个实例SSLSocket的实例作为其第一个参数。

一些笔记涉及到使用SSLObject：

- 所有的IO SSLObject都是非阻塞的。这意味着，例如read() ，SSLWantReadError如果需要比传入BIO可用的更多数据，则会引发这种 情况。
- 没有wrap_bio()类似的模块级调用 wrap_socket()。一个SSLObject总是通过一个SSLContext。

SSLObject使用内存缓冲区与外部世界进行通信。该类MemoryBIO提供了一个可用于此目的的内存缓冲区。它包装一个OpenSSL内存BIO（基本IO）对象：

*类*ssl.MemoryBIO

一个可用于在Python和SSL协议实例之间传递数据的内存缓冲区。

pending
返回当前在内存缓冲区中的字节数。

eof
指示内存BIO在文件结束位置是否为当前的布尔值。

read (*n* = *-1*)
从内存缓冲区中最多读取*n*个字节。如果*n*未指定或为负，则返回所有字节。

write (*buf*)
将*buf*中的字节写入内存BIO。所述*的*buf参数必须是支持缓冲协议的对象。

返回值是写入的字节数，它总是等于*buf*的长度。

write_eof ( )
将EOF标记写入存储器BIO。在此方法被调用后，调用是非法的write()。 eof在读取了当前在缓冲区中的所有数据后，该属性将变为真。

18.2.8。SSL会话

3.6版本中的新功能。

*类*ssl.SSLSession
会话对象使用session。

id

time

timeout

ticket_lifetime_hint

has_ticket

18.2.9。安全考虑

18.2.9.1。最佳默认

对于**客户端使用**，如果您对安全策略没有任何特殊要求，强烈建议您使用该create_default_context()功能来创建SSL上下文。它将加载系统的可信CA证书，启用证书验证和主机名检查，并尝试选择合理安全的协议和密码设置。

例如，下面介绍如何使用smtplib.SMTP该类创建到SMTP服务器的可信安全连接：

```
>>> import ssl, smtplib
>>> smtp = smtplib.SMTP("mail.python.org", port=587)
>>> context = ssl.create_default_context()
```

```
>>> smtp.starttls(context=context)
(220, b'2.0.0 Ready to start TLS')
```

如果连接需要客户端证书，则可以添加该证书 SSLContext.load_cert_chain()。

相比之下，如果您通过SSLContext 自己调用构造函数来创建SSL上下文，则默认情况下将不会启用证书验证或主机名检查。如果您这样做，请阅读下面的段落以达到良好的安全级别。

18.2.9.2。手动设置

18.2.9.2.1。验证证书

SSLContext直接调用构造函数时， CERT_NONE是默认值。由于它不认证对方，所以它可能是不安全的，特别是在客户端模式下，大多数时候你想确保你正在与之通话的服务器的真实性。因此，在客户端模式下，强烈建议使用 CERT_REQUIRED。然而，这本身并不足够; 您还必须检查可以通过调用获得的服务器证书是否 SSLSocket.getpeercert()符合所需的服务。对于许多协议和应用程序，服务可以通过主机名来标识; 在这种情况下， match_hostname()可以使用该功能。SSLContext.check_hostname启用时会自动执行此常见检查。

在服务器模式下，如果要使用SSL层对客户端进行身份验证（而不是使用更高级别的身份验证机制），则还必须指定CERT_REQUIRED并类似地检查客户端证书。

> **注意：** 在客户端模式，CERT_OPTIONAL并且CERT_REQUIRED除非匿名密码被启用（它们被默认禁用）是等价的。

18.2.9.2.2。协议版本

SSL版本2和3被认为是不安全的，因此使用起来很危险。如果您希望客户端和服务器之间的兼容性最大，建议使用PROTOCOL_TLS_CLIENT或 PROTOCOL_TLS_SERVER作为协议版本。SSLv2和SSLv3默认是禁用的。

```
>>> client_context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
>>> client_context.options |= ssl.OP_NO_TLSv1
>>> client_context.options |= ssl.OP_NO_TLSv1_1
```

上面创建的SSL上下文仅允许TLSv1.2及更高版本（如果系统支持）连接到服务器。PROTOCOL_TLS_CLIENT 意味着缺省情况下证书验证和主机名检查。您必须将证书加载到上下文中。

18.2.9.2.3。密码选择

如果您有高级安全要求，则可以通过该SSLContext.set_ciphers()方法对协商SSL会话时启用的密码进行微调 。从Python 3.2.3开始，ssl模块在默认情况下会禁用某些弱密码，但您可能想要进一步限制密码选择。请务必阅读关于密码列表格式的 OpenSSL文档。如果要检查给定密码列表是否启用了哪些密码，请在系统上使用 SSLContext.get_ciphers()或执行该命令。 openssl ciphers

18.2.9.3。多处理

如果将此模块用作多处理应用程序的一部分（例如使用multiprocessingor concurrent.futures模块），请注意OpenSSL的内部随机数生成器无法正确处理分叉的进程。如果应用程序使用任何SSL功能，则必须更改父进程的PRNG状态os.fork()。任何成功的调用RAND_add()，RAND_bytes()或者 RAND_pseudo_bytes()是足够的。

18.2.10。LibreSSL支持

LibreSSL是OpenSSL 1.0.1的一个分支。ssl模块对LibreSSL的支持有限。当使用LibreSSL编译ssl模块时，某些功能不可用。

- LibreSSL> = 2.6.1不再支持NPN。这些方法 SSLContext.set_npn_protocols()和SSLSocket.selected_npn_protocol()不可用。
- SSLContext.set_default_verify_paths() 忽略env变量 SSL_CERT_FILE 和 SSL_CERT_PATH 虽然 get_default_verify_paths()仍然报道。

也可以看看:

类 socket.socket
基础socket类的文档

SSL / TLS强加密：简介
从Apache HTTP Server文档介绍

RFC 1422：Internet电子邮件的隐私增强：第二部分：基于证书的密钥管理
史蒂夫肯特

RFC 4086：安全性的随机性要求
Donald E.，Jeffrey I. Schiller

RFC 5280：Internet X.509公钥基础设施证书和证书撤销列表（CRL）配置文件
D.库珀

RFC 5246：传输层安全（TLS）协议版本1.2
T. Dierks et。人。

RFC 6066：传输层安全性（TLS）扩展
D.东湖

IANA TLS：传输层安全性（TLS）参数
IANA

RFC 7525：安全使用传输层安全性（TLS）和数据报传输层安全性（DTLS）的建议
IETF

Mozilla的服务器端TLS建议
Mozilla的

18.3。 select- 等待I / O完成

该模块提供存取select()和poll()可用的功能在大多数操作系统中，devpoll()在Solaris和衍生物可用，epoll()在Linux 2.5+可用且 kqueue()可用于大多数BSD。请注意，在Windows上，它只适用于套接字; 在其他操作系统上，它也适用于其他文件类型（特别是在Unix上，它在管道上工作）。它不能用于常规文件，以确定自上次读取文件后文件是否已增长。

注意： 该selectors模块允许基于select模块基元的高级和高效的I / O复用。鼓励用户使用selectors模块，除非他们想要精确控制所使用的操作系统级基元。

该模块定义了以下内容：

*异常*select. error
不推荐使用的别名OSError。

*版本3.3中更改：*以下**PEP 3151**，这个班被**取而代之**OSError。

select. devpoll（）
（仅在Solaris和衍生产品上受支持。）返回一个/dev/poll 轮询对象; 有关devpoll对象支持的方法，请参见下面的/ dev / poll轮询对象部分。

devpoll()对象链接到实例化时允许的文件描述符的数量。如果你的程序降低了这个值，devpoll()将会失败。如果你的程序增加了这个值，devpoll()可能会返回一个不完整的活动文件描述符列表。

新的文件描述符是不可继承的。

3.3版本的新功能

*在版本3.4中更改：*新的文件描述符现在是不可继承的。

select. epoll（ *sizehint = -1*，*flags = 0* ）
（仅在Linux 2.5.44及更新版本上受支持。）返回边缘轮询对象，该边缘轮询对象可用作I / O事件的边缘或层次触发接口。*sizehint*和*标志*被弃用并完全忽略。

请参阅下面的边缘和级别触发轮询（epoll）对象部分，了解针对对象支持的方法。

epoll对象支持上下文管理协议：在with语句中使用时，新的文件描述符在块的末尾自动关闭。

新的文件描述符是不可继承的。

*在版本3.3中更改：*添加了*flags*参数。

*在版本3.4中进行了更改：*with添加了对语句的支持。新的文件描述符现在是不可继承的。

*：自从3.4版本不推荐使用该标志*的参数。 select.EPOLL_CLOEXEC现在默认使用。使用os.set_inheritable()使文件描述符可继承。

select.poll()

（不受所有操作系统支持。）返回一个轮询对象，该对象支持注册和取消注册文件描述符，然后轮询它们以查找I / O事件; 有关轮询对象支持的方法，请参见下面的轮询对象部分。

select.kqueue()

（仅在BSD上支持。）返回一个内核队列对象; 有关kqueue对象支持的方法，请参见下面的Kqueue Objects部分 。

新的文件描述符是不可继承的。

*在版本3.4中更改：*新的文件描述符现在是不可继承的。

select.kevent(*ident*，*filter* = *KQ_FILTER_READ*，*flags* = *KQ_EV_ADD*，*fflags* = *0*，*data* = *0*，*udata* = *0*)

（仅在BSD上支持。）返回内核事件对象; 有关kevent对象支持的方法，请参见下面的Kevent对象部分 。

select.select(*RLIST*，*wlist*，*的Xlist*[，*超时*])

这是Unix `select()`系统调用的直接接口。前三个参数是“可等待对象”的序列：表示文件描述符的整数或带有无参数方法的对象，`fileno()`返回这样一个整数：

- *rlist*：等到准备好阅读
- *wlist*：等到准备写作
- *xlist*：等待“异常情况”（请参阅手册页，了解您的系统考虑的这种情况）

空序列是允许的，但三个空序列的接受是平台相关的。（已知在Unix上工作，但在Windows上工作。）可选的*超时*参数指定*超时*为浮点数，以秒为单位。当省略*超时*参数时，该功能将阻塞，直到至少有一个文件描述符就绪。超时值为零指定轮询并且永不阻止。

返回值是准备好的对象列表的三倍：前三个参数的子集。如果在没有文件描述符准备就绪的情况下达到超时，则返回三个空列表。

间序列中的可接受的对象类型的Python 文件对象（例如`sys.stdin`通过返回，或物体`open()`或`os.popen()`）中，由返回的插座对象`socket.socket()`。你也可以自己定义一个*包装*类，只要它有一个合适的`fileno()`方法（它真的返回一个文件描述符，而不仅仅是一个随机整数）。

> **注意：** Windows上的文件对象不可接受，但套接字是。在Windows上，底层`select()`函数由WinSock库提供，并且不处理源自WinSock的文件描述符。

*在版本3.5中进行了更改：*除非信号处理程序引发异常，否则此函数现在会在被信号中断时重新计算超时值（请参阅 **PEP 475**为理由），而不是提高`InterruptedError`。

select.PIPE_BUF

当管道已被报告为可以写入时，可以写入而不阻塞到管道的最小字节数`select()`，`poll()`或此模块中的另一个接口。这不适用于其他类型的文件类对象，例如套接字。

这个值由POSIX保证至少为512.可用性：Unix。

3.2版本中的新功能

18.3.1。/dev/poll轮询对象

Solaris和衍生品都有/dev/poll。虽然select()是O（最高文件描述符）并且poll()是O（文件描述符的数量），但是/dev/pollO（活动文件描述符）。

/dev/poll行为非常接近标准poll() 对象。

devpoll.close( )
关闭轮询对象的文件描述符。

3.4版新增功能

devpoll.closed
True 如果轮询对象关闭。

3.4版新增功能

devpoll.fileno( )
返回轮询对象的文件描述符编号。

3.4版新增功能

devpoll.register(*fd*[, *eventmask*])
使用轮询对象注册文件描述符。未来对该poll()方法的调用 将检查文件描述符是否有任何挂起的I / O事件。 *fd*可以是一个整数，也可以是一个带有fileno()返回整数的方法的对象。文件对象实现fileno()，所以它们也可以用作参数。

*eventmask*是一个可选的位掩码，用于描述要检查的事件类型。常量与poll() 对象相同。默认值是常量的组合POLLIN， POLLPRI和POLLOUT。

> **警告：** 注册已经注册的文件描述符不是错误，但结果是未定义的。适当的操作是先取消注册或修改它。与此相比，这是一个重要的区别poll()。

devpoll.modify(*fd*[, *eventmask*])
这个方法的unregister()后面跟着一个 register()。明确地做同样的事情（有点）更有效率。

devpoll.unregister(*fd*)
删除正在由轮询对象跟踪的文件描述符。就像这个 register()方法一样，*fd*可以是一个整数或一个带有fileno()返回整数的方法的对象 。

试图删除从未注册的文件描述符将被安全忽略。

devpoll.poll([*timeout*])

轮询注册文件描述符的集合，并返回包含2元组的可能为空的列表，以便为要报告的事件或错误描述符。*fd*是文件描述符，*事件*是一个位掩码，为该描述符的报告事件设置位 - 用于等待输入，指示描述符可以写入，等等。一个空列表表示该呼叫超时，并且没有文件描述符有任何要报告的事件。如果给出*超时*，它指定系统在返回之前等待事件的时间长度（以毫秒为单位）。如果省略*超时*，则为-1，否则该调用将阻塞，直到该轮询对象发生事件。(fd, event)POLLINPOLLOUTNone

*在版本3.5中进行了更改：*除非信号处理程序引发异常，否则此函数现在会在被信号中断时重新计算超时值（请参阅 **PEP 475**为理由），而不是提高 InterruptedError。

18.3.2。边缘和水平触发轮询（epoll）对象

http://linux.die.net/man/4/epoll

eventmask

不变	含义
EPOLLIN	可供阅读
EPOLLOUT	可用于写入
EPOLLPRI	用于读取的紧急数据
EPOLLERR	在assoc上发生错误情况。FD
EPOLLHUP	挂起发生在assoc上。FD
EPOLLET	设置Edge Trigger行为，默认是Level Trigger行为
EPOLLONESHOT	设置一次性行为。在一个事件被拉出后，fd在内部被禁用
EPOLLEXCLUSIVE	当关联的fd发生事件时，只唤醒一个epoll对象。默认情况下（如果这个标志没有设置）是唤醒所有在fd上轮询的epoll对象。
EPOLLRDHUP	流套接字对等关闭连接或关闭写入连接的一半。
EPOLLRDNORM	相当于 EPOLLIN
EPOLLRDBAND	优先数据带可以被读取。
EPOLLWRNORM	相当于 EPOLLOUT
EPOLLWRBAND	可以写入优先数据。
EPOLLMSG	忽略。

epoll.close()
关闭epoll对象的控制文件描述符。

epoll.closed
True 如果epoll对象关闭。

epoll.fileno()
返回控制fd的文件描述符编号。

epoll.**fromfd**(*fd*)

从给定的文件描述符创建一个epoll对象。

epoll.**register**(*fd*[, *eventmask*])

用epoll对象注册一个fd描述符。

epoll.**modify**(*fd*, *eventmask*)

修改注册的文件描述符。

epoll.**unregister**(*fd*)

从epoll对象中删除已注册的文件描述符。

epoll.**poll**(*timeout* = *-1*, *maxevents* = *-1*)

等待事件。超时秒数（浮点数）

*在版本3.5中进行了更改：*除非信号处理程序引发异常，否则此函数现在会在被信号中断时重新计算超时值（请参阅 **PEP 475**为理由），而不是提高 InterruptedError。

18.3.3。轮询对象

poll()大多数Unix系统支持的系统调用为同时为多个客户提供服务的网络服务器提供了更好的可扩展性。 poll()缩放比较好，因为系统调用只需要列出感兴趣的文件描述符，同时select()构建位图，为感兴趣的fds打开位，然后必须再次线性扫描整个位图。 select()是O（最高文件描述符），而 poll()O（文件描述符的数量）。

poll.**register**(*fd*[, *eventmask*])

使用轮询对象注册文件描述符。未来对该poll()方法的调用 将检查文件描述符是否有任何挂起的I / O事件。 *fd*可以是一个整数，也可以是一个带有fileno()返回整数的方法的对象。文件对象实现fileno()，所以它们也可以用作参数。

*eventmask*是一个可选位掩码描述要检查事件的类型，可以是常量的组合POLLIN， POLLPRI和POLLOUT在如下表所述。如果未指定，则使用的默认值将检查所有3种类型的事件。

不变	含义
POLLIN	有数据需要阅读
POLLPRI	有紧急的数据要阅读
POLLOUT	准备输出：写入不会被阻止
POLLERR	某种错误情况
POLLHUP	挂了
POLLRDHUP	流套接字对等关闭连接，或关闭写入连接的一半
POLLNVAL	无效的请求：描述符未打开

注册已经注册的文件描述符并不是错误，并且具有与仅仅注册一次描述符相同的效果。

poll.modify（*fd*，*eventmask*）

修改已经注册的fd。这与之相同 。尝试修改从未注册的文件描述符会导致引发errno异常 。 register(fd, eventmask) OSError ENOENT

poll.unregister（*fd*）

删除正在由轮询对象跟踪的文件描述符。就像这个 register() 方法一样，*fd*可以是一个整数或一个带有fileno()返回整数的方法的对象 。

试图删除从未注册的文件描述符会导致引发 KeyError异常。

poll.poll（[*timeout*]）

轮询注册文件描述符的集合，并返回包含2元组的可能为空的列表，以便为要报告的事件或错误描述符。*fd*是文件描述符，*事件*是一个位掩码，为该描述符的报告事件设置位 - 用于等待输入，指示描述符可以写入，等等。一个空列表表示该呼叫超时，并且没有文件描述符有任何要报告的事件。如果给出*超时*，它指定系统在返回之前等待事件的时间长度（以毫秒为单位）。如果*超时*被忽略，否定，或者，该调用将被阻塞，直到该轮询对象发生事件。 (fd, event) POLLIN POLLOUT None

*在版本3.5中进行了更改：*除非信号处理程序引发异常，否则此函数现在会在被信号中断时重新计算超时值（请参阅 **PEP 475**为理由），而不是提高 InterruptedError。

18.3.4。Kqueue对象

kqueue.close（）

关闭kqueue对象的控制文件描述符。

kqueue.closed

True 如果kqueue对象关闭。

kqueue.fileno（）

返回控制fd的文件描述符编号。

kqueue.fromfd（*fd*）

从给定的文件描述符创建一个kqueue对象。

kqueue.control（*变更表*，*max_events*[，*超时=无*]）→EVENTLIST

低级接口kevent

- changelist必须是kevent对象或者可迭代的对象 None
- max_events必须是0或正整数
- 以秒为单位的超时时间

*在版本3.5中进行了更改：*除非信号处理程序引发异常，否则此函数现在会在被信号中断时重新计算超时值（请参阅 **PEP 475**为理由），而不是提高 InterruptedError。

18.3.5。KEVENT对象

https://www.freebsd.org/cgi/man.cgi?query=kqueue&sektion=2

kevent.ident

用于识别事件的值。解释取决于过滤器，但通常是文件描述符。在构造函数中，ident可以是一个int或一个带有fileno()方法的对象。kevent在内部存储整数。

kevent.filter

内核过滤器的名称。

不变	含义
KQ_FILTER_READ	获取描述符并在有数据可读时返回
KQ_FILTER_WRITE	获取描述符并在有可用数据写入时返回
KQ_FILTER_AIO	AIO请求
KQ_FILTER_VNODE	在*fflag*中观看一个或多个请求的事件时返回
KQ_FILTER_PROC	注意进程ID上的事件
KQ_FILTER_NETDEV	监视网络设备上的事件[在Mac OS X上不可用]
KQ_FILTER_SIGNAL	每当观看的信号被传送到进程时，都会返回
KQ_FILTER_TIMER	建立一个任意的计时器

kevent.flags

筛选器操作。

不变	含义
KQ_EV_ADD	添加或修改事件
KQ_EV_DELETE	从队列中移除一个事件
KQ_EV_ENABLE	Permitscontrol（ ）返回事件
KQ_EV_DISABLE	Disablesevent
KQ_EV_ONESHOT	第一次发生后删除事件
KQ_EV_CLEAR	检索事件后重置状态
KQ_EV_SYSFLAGS	内部事件
KQ_EV_FLAG1	内部事件
KQ_EV_EOF	过滤特定的EOF状况
KQ_EV_ERROR	查看返回值

kevent.fflags

过滤特定的标志。

KQ_FILTER_READ和 KQ_FILTER_WRITE过滤器标志：

不变	含义
KQ_NOTE_LOWAT	套接字缓冲区的低水印

不变	含义

KQ_FILTER_VNODE 过滤器标志：

不变	含义
KQ_NOTE_DELETE	*unlink（）*被调用
KQ_NOTE_WRITE	发生了写入
KQ_NOTE_EXTEND	该文件已被扩展
KQ_NOTE_ATTRIB	一个属性被改变了
KQ_NOTE_LINK	链接计数已更改
KQ_NOTE_RENAME	该文件被重命名
KQ_NOTE_REVOKE	对该文件的访问被撤销

KQ_FILTER_PROC 过滤器标志：

不变	含义
KQ_NOTE_EXIT	该过程已经退出
KQ_NOTE_FORK	该过程称为*fork（）*
KQ_NOTE_EXEC	该过程已经执行了一个新过程
KQ_NOTE_PCTRLMASK	内部过滤器标志
KQ_NOTE_PDATAMASK	内部过滤器标志
KQ_NOTE_TRACK	遵循*fork（）*过程
KQ_NOTE_CHILD	在*NOTE_TRACK*的子进程上 *返回*
KQ_NOTE_TRACKERR	无法附属于孩子

KQ_FILTER_NETDEV 过滤器标志（在Mac OS X上不可用）：

不变	含义
KQ_NOTE_LINKUP	链接已启动
KQ_NOTE_LINKDOWN	链接已关闭
KQ_NOTE_LINKINV	链接状态无效

kevent.data

过滤特定的数据。

kevent.udata

用户定义的值。

18.4。selectors- 高级I / O多路复用

3.4版新增功能

源代码： Lib / selectors.py

18.4.1。简介

该模块允许基于select模块原语构建的高级和高效的I / O复用 。鼓励用户改用这个模块，除非他们想要精确控制所使用的操作系统级基元。

它定义了一个BaseSelector抽象基类以及几个具体实现（KqueueSelector，EpollSelector...），可用于等待多个文件对象的I / O准备就绪通知。在下文中，“文件对象”是指具有fileno()方法或原始文件描述符的任何对象 。查看文件对象。

DefaultSelector是当前平台上最高效的实现的别名：这应该是大多数用户的默认选择。

注意： 支持的文件对象类型取决于平台：在Windows上，支持套接字，但不支持管道，而在Unix上，支持套接字（也可能支持某些其他类型，例如fifos或特殊文件设备）。

也可以看看：

select
低级I / O复用模块。

18.4.2。类

类层次结构：

```
BaseSelector
+-- SelectSelector
+-- PollSelector
+-- EpollSelector
+-- DevpollSelector
+-- KqueueSelector
```

在下面的内容中，*事件*是一个按位掩码，指示在给定的文件对象上应该等待哪些I / O事件。它可以是以下模块常量的组合：

不变	含义
EVENT_READ	可供阅读
EVENT_WRITE	可用于写入

*类*selectors. SelectorKey

阿SelectorKey是一个namedtuple用于将一个文件对象到其底层的文件描述符相关联，所选择的事件掩码和附加数据。它由几种BaseSelector 方法返回。

fileobj
文件对象注册。

fd
基础文件描述符。

events
此文件对象上必须等待的事件。

data
与此文件对象关联的可选不透明数据：例如，可用于存储每个客户端的会话ID。

*类*selectors.BaseSelector
A BaseSelector用于等待多个文件对象上的I / O事件准备就绪。它支持文件流注册，注销以及等待这些流上的I / O事件的方法，并具有可选的超时时间。它是一个抽象的基类，所以不能实例化。使用 DefaultSelector替代，或一个SelectSelector ， KqueueSelector等等。如果你想专门使用的实现，你的平台支持它。 BaseSelector其具体实现支持 上下文管理器协议。

abstractmethod register (*fileobj* , *events* , *data = None*)
注册一个文件对象进行选择，监视它以查找I / O事件。

*fileobj*是要监视的文件对象。它可以是一个整型文件描述符或一个带有fileno()方法的对象。 *事件*是监视*事件*的按位掩码。 *数据*是一个不透明的对象。

这将返回一个新SelectorKey实例，或者ValueError在无效事件掩码或文件描述符的情况下引发a ，或者 KeyError文件对象已经注册。

abstractmethod unregister (*fileobj*)
从选择中取消注册文件对象，将其从监视中移除。文件对象在关闭之前应该被注销。

*fileobj*必须是先前注册的文件对象。

这会返回关联的SelectorKey实例，或者KeyError如果*fileobj*未注册，*则会*引发一次 。ValueError如果*fileobj*无效（例如，它没有fileno() 方法或其fileno()方法具有无效的返回值），它会引发 。

modify (*fileobj* , *events* , *data = None*)
更改已注册的文件对象的监控事件或附加数据。

除了可以更高效地实施之外，这相当于BaseSelector.unregister(fileobj)()紧随其后。BaseSelector.register(fileobj, events, data)()

这会返回一个新SelectorKey实例，或者ValueError在无效事件掩码或文件描述符的情况下引发a ，或者 KeyError文件对象未被注册。

abstractmethod select (*timeout = None*)

等到某些已注册的文件对象变为就绪状态，或超时过期。

如果，这指定了最长等待时间，以秒为单位。如果该调用不会阻塞，并且将报告当前准备好的文件对象。如果*超时*是，调用将阻塞，直到一个监视的文件对象就绪。`timeout > 0`timeout <= 0None

这将返回一个元组列表，每个准备好的文件对象都有一个元组列表。(key, events)

*key*是与SelectorKey就绪文件对象相对应的实例。 *事件*是在这个文件对象上准备好的*事件*的掩码。

> **注意:** 如果当前进程收到信号，则此方法可以在任何文件对象准备就绪或超时已过之前返回：在这种情况下，将返回空列表。

*在版本3.5中更改：*如果信号处理程序没有引发异常，则选择器现在在被信号中断时重新计算重新计算的超时值（请参阅 **PEP 475**的基本原理），而不是在超时之前返回一个空的事件列表。

close ( )

关闭选择器。

这必须被调用以确保任何底层资源都被释放。选择器关闭后不得使用。

get_key (*fileobj*)

返回与注册文件对象关联的密钥。

这将返回与SelectorKey此文件对象关联的实例，或者KeyError在文件对象未注册时引发。

abstractmethod get_map ( )

返回文件对象到选择键的映射。

这将返回一个Mapping将注册文件对象映射到其关联SelectorKey 实例的实例。

*类*selectors. DefaultSelector

默认的选择器类，使用当前平台上可用的最有效的实现。这应该是大多数用户的默认选择。

*类*selectors. SelectSelector

select.select()基于选择器。

*类*selectors. PollSelector

select.poll()基于选择器。

*类*selectors. EpollSelector

select.epoll()基于选择器。

fileno ( )

这将返回基础select.epoll()对象使用的文件描述符 。

*类*selectors.DevpollSelector

select.devpoll()基于选择器。

fileno ( )

这将返回基础select.devpoll()对象使用的文件描述符 。

3.5版本中的新功能。

*类*selectors.KqueueSelector

select.kqueue()基于选择器。

fileno ( )

这将返回基础select.kqueue()对象使用的文件描述符 。

18.4.3。示例

这是一个简单的echo服务器实现：

```
import selectors
import socket

sel = selectors.DefaultSelector()

def accept(sock, mask):
    conn, addr = sock.accept()  # Should be ready
    print('accepted', conn, 'from', addr)
    conn.setblocking(False)
    sel.register(conn, selectors.EVENT_READ, read)

def read(conn, mask):
    data = conn.recv(1000)  # Should be ready
    if data:
        print('echoing', repr(data), 'to', conn)
        conn.send(data)  # Hope it won't block
    else:
        print('closing', conn)
        sel.unregister(conn)
        conn.close()

sock = socket.socket()
sock.bind(('localhost', 1234))
sock.listen(100)
sock.setblocking(False)
sel.register(sock, selectors.EVENT_READ, accept)

while True:
    events = sel.select()
    for key, mask in events:
        callback = key.data
        callback(key.fileobj, mask)
```

18.5。asyncio- 异步I / O，事件循环，协程和任务

3.4版新增功能

源代码： Lib / asyncio /

此模块提供了使用协同程序编写单线程并发代码的基础结构，在套接字和其他资源上多路复用I / O访问，运行网络客户端和服务器以及其他相关基元。以下是包装内容的更详细清单：

- 具有各种系统特定实现的可插入事件循环；
- 运输和协议抽象（类似于 Twisted中的）；
- 对TCP，UDP，SSL，子流程管道，延迟调用等的具体支持（有些可能与系统有关）；
- 一个Future模仿concurrent.futures 模块中的类的类，但适用于事件循环;
- 协程和任务基于（ yield fromPEP 380），以帮助以顺序方式编写并发代码;
- 取消对Futures和协程的支持;
- 在单个线程中的协同程序之间使用的同步原语，模仿threading模块中的同步原语；
- 一个将工作传递给线程池的接口，当你绝对肯定不得不使用阻塞I / O调用的库时。

异步编程比传统的“顺序”编程更复杂：请参阅使用asyncio开发的页面，其中列出了常见的陷阱并解释了如何避免它们。在开发过程中启用调试模式以检测常见问题。

目录：

也可以看看： 该`asyncio`模块设计于**PEP 3156**。有关运输和协议的动机入门，请参阅**PEP 3153**。

18.5.1。基本事件循环

源代码： Lib / asyncio / events.py

事件循环是由中提供的中央执行设备asyncio。它提供了多种设施，包括：

- 注册，执行和取消延迟呼叫（超时）。
- 为各种通信创建客户端和服务器传输。
- 启动子流程和相关传输以与外部程序进行通信。
- 将昂贵的函数调用委托给一个线程池。

*类*asyncio.BaseEventLoop

这个类是一个实现细节。它是一个子类， AbstractEventLoop可能是基类中的具体事件循环实现asyncio。它不应该直接使用; AbstractEventLoop改为使用。 BaseEventLoop不应该被第三方代码分类; 内部接口不稳定。

*类*asyncio.AbstractEventLoop

事件循环的抽象基类。

这个类不是线程安全的。

18.5.1.1。运行事件循环

AbstractEventLoop.run_forever()

运行直到stop()被调用。如果在stop()被调用之前 run_forever()被调用，这将轮询I / O选择器一次，超时时间为零，运行响应I / O事件（以及已经调度的事件）计划的所有回调，然后退出。如果stop()在run_forever()运行时被调用，这将运行当前批次的回调，然后退出。请注意，在这种情况下，回调计划的回调将不会运行; 他们会在下次run_forever()调用时运行。

在版本3.5.1中更改。

AbstractEventLoop.run_until_complete(*未来*)

运行直到Future完成。

如果参数是一个协程对象，它将被包装 ensure_future()。

返回未来的结果，或引发异常。

AbstractEventLoop.is_running()

返回事件循环的运行状态。

AbstractEventLoop.stop()

停止运行事件循环。

这会导致run_forever()在下一个合适的机会退出（有关更多详细信息，请参阅此处）。

在版本3.5.1中更改。

AbstractEventLoop.**is_closed**()

True如果事件循环已关闭，则返回。

3.4.2版本的新功能。

AbstractEventLoop.**close**()

关闭事件循环。该循环一定不能运行。未完成的回调将会丢失。

这会清除队列并关闭执行程序，但不会等待执行程序完成。

这是幂等和不可逆的。此后不应该调用其他方法。

coroutine AbstractEventLoop.**shutdown_asyncgens**()

将所有当前打开的异步生成器对象安排为通过aclose()调用关闭。调用此方法后，只要重新生成新的异步生成器，事件循环就会发出警告。应该用于可靠地完成所有预定的异步生成器。例：

```
try:
    loop.run_forever()
finally:
    loop.run_until_complete(loop.shutdown_asyncgens())
    loop.close()
```

3.6版本中的新功能。

18.5.1.2。调用

大多数asyncio功能不接受关键字。如果您想将关键字传递给回调，请使用functools.partial()。例如，会打电话。loop.call_soon(functools.partial(print, "Hello", flush=True))print("Hello", flush=True)

注意： functools.partial()比lambda函数更好，因为asyncio可以检查functools.partial()对象以调试模式显示参数，而lambda函数的表示效果较差。

AbstractEventLoop.**call_soon**(*callback*, **args*)

尽快安排回拨。call_soon()当控制返回到事件循环时，回调将在返回后调用。

这将作为FIFO队列运行，回调会按照它们的注册顺序调用。每个回调将被调用一次。

回调后的任何位置参数将在调用时传递给回调函数。

asyncio.Handle返回的实例可用于取消回调。

使用functools.partial将关键字传递给回调。

AbstractEventLoop.**call_soon_threadsafe**(*callback*, **args*)

喜欢call_soon()，但线程安全。

请参阅 文档的并发性和多线程部分。

18.5.1.3。延迟调用

事件循环有自己的计算超时的内部时钟。使用哪个时钟取决于（平台特定的）事件循环实现; 理想情况下，它是一个单调的时钟。这通常会是一个不同的时钟time.time()。

注意: 超时（相对*延迟*或绝对*时间*）不应超过一天。

AbstractEventLoop.call_later(*延迟*, *回调*, **args*)

安排在给定*延迟*秒后（无论是int还是float）调用*回调*。

asyncio.Handle返回的实例可用于取消回调。

*回调*将在每次通话时被调用一次call_later()。如果两个回调计划的时间完全相同，那么它将不会被首先调用。

调用时，可选的位置*参数*将传递给回调函数。如果你想用一些命名参数调用回调，使用闭包或者functools.partial()。

使用functools.partial将关键字传递给回调。

AbstractEventLoop.call_at(*when*, *callback*, **args*)

当（int或float）使用与之相同的时间参考时， 安排在给定的绝对时间戳处 *调用回调*。AbstractEventLoop.time()

此方法的行为与call_later()。相同。

asyncio.Handle返回的实例可用于取消回调。

使用functools.partial将关键字传递给回调。

AbstractEventLoop.time()

float根据事件循环的内部时钟，将当前时间作为一个值返回。

也可以看看: 该asyncio.sleep()功能。

18.5.1.4。期货

AbstractEventLoop.create_future()

创建一个asyncio.Future附加到循环的对象。

这是在asyncio中创建期货的首选方式，因为事件循环实现可以提供Future类的替代实现（具有更好的性能或工具）。

3.5.2版本的新功能。

18.5.1.5。任务

AbstractEventLoop.create_task（*coro*）

计划协同对象的执行：将来包装它。返回一个Task对象。

第三方事件循环可以使用它们自己的子类来Task实现互操作性。在这种情况下，结果类型是。的一个子类 Task。

这个方法是在Python 3.4.2中添加的。使用该async()函数也支持较老的Python版本。

3.4.2版本的新功能。

AbstractEventLoop.set_task_factory（*工厂*）

设置将被使用的任务工厂 AbstractEventLoop.create_task()。

如果*工厂*是None默认的任务工厂，则会设置。

如果*工厂*是*可调用的*，它应该有一个签名匹配 ，其中*循环*将是对活动事件循环的引用，*coro*将是一个协程对象。可调用对象必须返回兼容对象。(loop, coro)asyncio.Future

3.4.4版本的新功能。

AbstractEventLoop.get_task_factory（）

返回任务工厂，或者None如果默认的工具正在使用中。

3.4.4版本的新功能。

18.5.1.6。创建连接

coroutine AbstractEventLoop.create_connection（*protocol_factory*，*host = None*，*port = None*，*，ssl = None*，*family = 0*，*proto = 0*，*flags = 0*，*sock = None*，*local_addr = None*，*server_hostname = None*）

创建到给定互联网*主机*和*端口*：套接字系列AF_INET或 AF_INET6根据*主机*（或*家庭*，如果指定），套接字类型的流传输连接SOCK_STREAM。 *protocol_factory*必须是可返回协议实例的可调用对象。

这种方法是一个协程，它会尝试在后台建立连接。成功时，协程返回一对。(transport, protocol)

基本操作的时间顺序如下：

1. 连接已建立，并 创建一个传输来表示它。
2. *protocol_factory*不带参数被调用，并且必须返回一个 协议实例。
3. 协议实例绑定到传输，并connection_made()调用其 方法。
4. 该协程与该 对成功返回。(transport, protocol)

创建的传输是一个实现相关的双向流。

注意： *protocol_factory*可以是任何类型的可调用的，不一定是类。例如，如果您想使用预先创建的协议实例，则可以通过。`lambda: my_protocol`

改变连接创建方式的选项：

- *ssl*：如果给定而不是false，则会创建一个SSL / TLS传输（默认情况下会创建一个纯TCP传输）。如果*ssl*是一个`ssl.SSLContext`对象，则使用此上下文创建传输; 如果*ssl*是`True`，则使用具有一些未指定默认设置的上下文。

  也可以看看： SSL / TLS安全考虑

- *server_hostname*仅用于与*ssl*一起使用，并设置或覆盖目标服务器证书将与之匹配的主机名。默认情况下，使用*主机*参数的值。如果*主机*为空，则没有默认值，并且必须为*server_hostname*传递一个值。如果*server_hostname*是空字符串，则禁用主机名匹配（这是一个严重的安全风险，允许中间人攻击）。

- *family*，*proto*，*flags*是可选的地址族，协议和标志将通过getaddrinfo（）传递给*主机*解析。如果给出，这些应该都是来自相应`socket`模块常量的整数。

- 如果有的话，*袜子*应该是现有的，已经连接的`socket.socket`物体，供交通工具使用。如果给出了*袜子*，则不应指定*主机*，*端口*，*家族*，*原型*，*标志*和*local_addr*。

- *local_addr*（如果有）是一个用于在本地绑定套接字的元组。该*local_host*和*LOCAL_PORT*中查找使用的getaddrinfo（），类似于*主机*和*端口*。`(local_host, local_port)`

*版本3.5中更改：*在Windows上`ProactorEventLoop`，现在支持SSL / TLS。

也可以看看： 该`open_connection()`函数可用于获取一对（`StreamReader`，`StreamWriter`）而不是协议。

coroutine `AbstractEventLoop.`**`create_datagram_endpoint`**`(`*protocol_factory*`,` *local_addr = None*`,` *remote_addr = None*`,` ***`,` *family = 0*`,` *proto = 0*`,` *flags = 0*`,` *reuse_address = None*`,` *reuse_port = None*`,` *allow_broadcast = None*`,` *sock = None*`)`

创建数据报连接：套接字系列`AF_INET`或`AF_INET6`根据*主机*（或*家庭*，如果指定），套接字类型`SOCK_DGRAM`。*protocol_factory*必须是可返回协议实例的可调用对象。

这种方法是一个协程，它会尝试在后台建立连接。成功时，协程返回一对。`(transport, protocol)`

选项更改连接的创建方式：

- *local_addr*（如果有）是一个用于在本地绑定套接字的元组。该*local_host*和*LOCAL_PORT*都抬起头来使用。`(local_host, local_port)` `getaddrinfo()`
- *remote_addr*（如果给出）是用于将套接字连接到远程地址的元组。在*远程主机*和*REMOTE_PORT*都抬起头来使用。`(remote_host, remote_port)` `getaddrinfo()`
- *家庭*，*原*，*标志*是可选的家庭地址，协议和标志，通过传递`getaddrinfo()`的*主机*解决方案。如果给出，这些应该都是来自相应`socket`模块常量的整数。

- *reuse_address*告诉内核在TIME_WAIT状态下重用本地套接字，而无需等待其自然超时过期。如果未指定，则会True在UNIX上自动设置为。
- *reuse_port*通知内核允许将此端点绑定到与其他现有端点绑定到的端口相同的端口，只要它们在创建时都设置了此标志即可。Windows和某些UNIX不支持此选项。如果SO_REUSEPORT常量没有定义，那么这个功能是不支持的。
- *allow_broadcast*通知内核允许此端点发送消息到广播地址。
- *袜子*可以选择性地指定，以便使用预先存在的，已经连接的socket.socket物体供运输工具使用。如果指定，*LOCAL_ADDR*和*REMOTE_ADDR*应该被忽略（必须None）。

在Windows上ProactorEventLoop，此方法不受支持。

请参阅UDP回显客户端协议和 UDP回显服务器协议示例。

改变在3.4.4版本：在家庭，原，标志，reuse_address，reuse_port， allow_broadcast*和*袜子*添加参数。

*协程*AbstractEventLoop.**create_unix_connection**（*protocol_factory*，*path*，*，*ssl* = *None*，*sock* = *None*，*server_hostname* = *None* ）

创建UNIX连接：套接字系列AF_UNIX，套接字类型SOCK_STREAM。所述AF_UNIX插座家庭使用同一台机器上有效的进程之间进行通信。

这种方法是一个协程，它会尝试在后台建立连接。成功时，协程返回一对。(transport, protocol)

*path*是UNIX域套接字的名称，除非 指定了*sock*参数，否则它是必需的。摘要支持UNIX套接字，str和 bytes路径。

请参阅AbstractEventLoop.create_connection()参数的方法。

可用性：UNIX。

18.5.1.7。创建监听连接

coroutine AbstractEventLoop.**create_server**（*protocol_factory*，*host* = *None*，*port* = *None*，*，*family* = *socket.AF_UNSPEC*，*flags* = *socket.AI_PASSIVE*，*sock* = *None*，*backlog* = *100*，*ssl* = *None*，*reuse_address* = *None*，*reuse_port* = *None* ）

创建SOCK_STREAM绑定到*主机*和*端口*的TCP服务器（套接字类型）。

返回一个Server对象，它的sockets属性包含创建的套接字。使用该Server.close()方法停止服务器：关闭侦听套接字。

参数：

- 该*主机*参数可以是一个字符串，在这种情况下，TCP服务器绑定到*主机*和*端口*。该*主机*参数也可以是串序列，并在这种情况下，TCP服务器被绑定到序列的所有主机。如果*主机*是空字符串，或者None假定所有接口都将返回多个套接字列表（很可能是一个用于IPv4，另一个用于IPv6）。
- *家庭*可以设置为socket.AF_INET或 AF_INET6迫使插座使用IPv4或IPv6。如果未设置，则将从主机确定（默认为socket.AF_UNSPEC）。

- *标志*是一个位掩码getaddrinfo()。
- 可以选择指定*袜子*以便使用预先存在的套接字对象。如果指定，应该省略*主机*和*端口*（必须是 None ）。
- *积压*是传递给队列的最大连接数 listen()（默认为100）。
- 可以将*ssl*设置为SSLContext通过接受的连接启用SSL。
- *reuse_address*告诉内核在TIME_WAIT状态下重用本地套接字，而无需等待其自然超时过期。如果未指定，则会True在UNIX上自动设置为。
- *reuse_port*通知内核允许将此端点绑定到与其他现有端点绑定到的端口相同的端口，只要它们在创建时都设置了此标志即可。Windows上不支持此选项。

这个方法是一个协程。

*版本3.5中更改：*在Windows上ProactorEventLoop，现在支持SSL / TLS。

也可以看看： 该函数start_server()创建一个（ StreamReader， StreamWriter ）对，并用该对函数返回一个函数。

*改变在3.5.1版本：*该*主机*参数现在可以串序列。

*协程*AbstractEventLoop.**create_unix_server**（*protocol_factory*，*path = None*，*，sock = None*，*backlog = 100*，*ssl = None* ）

类似于AbstractEventLoop.create_server()，但特定于套接字系列AF_UNIX。

这个方法是一个协程。

可用性：UNIX。

*协程*BaseEventLoop.**connect_accepted_socket**（*protocol_factory*，*sock*，*，ssl = None* ）

处理接受的连接。

这由服务器使用，它接受asyncio之外的连接，但使用asyncio来处理它们。

参数：

- *sock*是从accept 调用返回的先前存在的套接字对象。
- 可以将*ssl*设置为SSLContext通过接受的连接启用SSL。

这个方法是一个协程。完成后，协程返回一对。(transport, protocol)

3.5.3版本的新功能。

18.5.1.8。观察文件描述符

在Windows上SelectorEventLoop，只支持套接字句柄（例如：不支持管道文件描述符）。

在Windows上ProactorEventLoop，这些方法不受支持。

AbstractEventLoop.**add_reader**（*fd*，*callback*，** args* ）

开始观察文件描述符以获取可用性，然后使用指定的参数调用 *回调*。

使用functools.partial将关键字传递给回调。

AbstractEventLoop.remove_reader(*fd*)

停止观看文件描述符的可用性。

AbstractEventLoop.add_writer(*fd*, *callback*, **args*)

开始观察文件描述符的写入可用性，然后用指定的参数调用 *回调*。

使用functools.partial将关键字传递给回调。

AbstractEventLoop.remove_writer(*fd*)

停止观看文件描述符以获取写入可用性。

该手表读取事件文件描述符 的例子使用了低级别的AbstractEventLoop.add_reader()方法来注册一个套接字的文件描述符。

18.5.1.9。低级套接字操作

*协同程序*AbstractEventLoop.sock_recv(*sock*, *nbytes*)

从套接字接收数据。模仿阻塞 socket.socket.recv()方法。

返回值是一个表示接收到的数据的字节对象。一次接收的最大数据量由*nbytes*指定。

通过SelectorEventLoop事件循环，套接字*袜*必须是非阻塞的。

这个方法是一个协程。

*协程*AbstractEventLoop.sock_sendall(*袜子*, *数据*)

将数据发送到套接字。模仿阻塞 socket.socket.sendall()方法。

套接字必须连接到远程套接字。此方法继续从*数据*发送数据，直到发送所有数据或发生错误。 None成功返回。出错时，会引发异常，并且无法确定接收端成功处理了多少数据（如果有）。

通过SelectorEventLoop事件循环，套接字*袜*必须是非阻塞的。

这个方法是一个协程。

*协程*AbstractEventLoop.sock_connect(*sock*, *address*)

连接到*地址*的远程套接字。模仿阻塞socket.socket.connect()方法。

通过SelectorEventLoop事件循环，套接字*袜*必须是非阻塞的。

这个方法是一个协程。

*在版本3.5.2中更改：*address不再需要解决。 sock_connect 将尝试通过调用来检查*地址*是否已经解决 socket.inet_pton()。如果不是， AbstractEventLoop.getaddrinfo()将用于解析 *地址*。

也可以看看： AbstractEventLoop.create_connection() 和 asyncio.open_connection()。

*协同程序*AbstractEventLoop.**sock_accept**(*sock*)

接受连接。阻塞后建模 socket.socket.accept()。

套接字必须绑定到地址并监听连接。返回值是一对，*conn* 是一个*新的*套接字对象，可用于在连接上发送和接收数据，*地址*是绑定到连接另一端套接字的地址。(conn, address)

插座*袜子*必须是非阻塞的。

这个方法是一个协程。

也可以看看： AbstractEventLoop.create_server()和start_server()。

18.5.1.10。解析主机名

*协程*AbstractEventLoop.**getaddrinfo**(*主机*，*端口*，*，*family = 0*，*type = 0*，*proto = 0*，*flags = 0*)

这个方法是一个协程，类似于 socket.getaddrinfo()函数但是非阻塞。

*协程*AbstractEventLoop.**getnameinfo**(*sockaddr*，*flags = 0*)

这个方法是一个协程，类似于 socket.getnameinfo()函数但是非阻塞。

18.5.1.11。连接管

在Windows上SelectorEventLoop，这些方法不受支持。使用ProactorEventLoop支持Windows上的管道。

*协程*AbstractEventLoop.**connect_read_pipe**(*protocol_factory*，*pipe*)

在eventloop中注册读取管道。

*protocol_factory*应该用Protocol 接口实例化对象。 *管道*是一个类似文件的对象。返回对，其中*传输*支持 接口。(transport, protocol)ReadTransport

通过SelectorEventLoop事件循环，*管道*被设置为非阻塞模式。

这个方法是一个协程。

*协程*AbstractEventLoop.**connect_write_pipe**(*protocol_factory*，*pipe*)

在eventloop中注册写入管道。

*protocol_factory*应该用BaseProtocol 接口实例化对象。*管道*是类文件对象。返回对，其中*传输*支持 接口。(transport, protocol)WriteTransport

通过SelectorEventLoop事件循环，*管道*被设置为非阻塞模式。

这个方法是一个协程。

也可以看看： 该AbstractEventLoop.subprocess_exec()和AbstractEventLoop.subprocess_shell()方法。

18.5.1.12。UNIX信号

可用性：仅适用于UNIX。

AbstractEventLoop.**add_signal_handler**(*signum*, *callback*, **args*)

为信号添加处理程序。

ValueError如果信号编号无效或不可捕捉，请升起。提高RuntimeError如果有问题设立的处理程序。

使用functools.partial将关键字传递给回调。

AbstractEventLoop.**remove_signal_handler**(*sig*)

删除信号处理程序。

True如果信号处理程序被移除，返回，False如果没有。

也可以看看： 该signal模块。

18.5.1.13。执行者

在Executor（线程池或进程池）中调用一个函数。默认情况下，事件循环使用线程池执行器（ThreadPoolExecutor）。

*协程*AbstractEventLoop.**run_in_executor**(*executor*, *func*, **args*)

安排一个*func*在指定的执行者中被调用。

在*执行*参数应该是一个Executor 实例。如果*执行者*是，则使用缺省*执行*程序None。

使用functools.partial将关键字传递给* func *。

这个方法是一个协程。

*在版本3.5.3中进行了更改：*BaseEventLoop.run_in_executor()不再配置 max_workers它创建的线程池执行程序，而是将其留给线程池执行程序（ThreadPoolExecutor）以设置默认值。

AbstractEventLoop.**set_default_executor**(*执行者*)

设置使用的默认执行器run_in_executor()。

18.5.1.14。错误处理

允许定制如何在事件循环中处理异常。

AbstractEventLoop.set_exception_handler(*处理程序*)

将*处理程序*设置为新事件循环异常处理程序。

如果*处理程序*是None，则会设置默认的异常处理程序。

如果*handler*是一个可调用对象，它应该有一个匹配的签名，其中 对活动事件循环的引用将是一个对象（有关上下文的详细信息，请参阅文档）。(loop, context) loop context dict call_exception_handler()

AbstractEventLoop.get_exception_handler( )

返回异常处理程序，或者None如果默认的处理程序正在使用中。

3.5.2版本的新功能。

AbstractEventLoop.default_exception_handler(*上下文*)

缺省异常处理程序

这在发生异常并且没有设置异常处理程序时调用，并且可以由想要推迟为默认行为的自定义异常处理程序调用。

*上下文*参数与in中的含义相同 call_exception_handler()。

AbstractEventLoop.call_exception_handler(*上下文*)

调用当前事件循环异常处理程序。

*上下文*是一个dict包含以下键的对象（以后可能会引入新的键）：

- 'message'：错误信息;
- 'exception'（可选）：异常对象;
- 'future'（可选）：asyncio.Futureinstance;
- 'handle'（可选）：asyncio.Handleinstance;
- '协议'（可选）：协议实例;
- 'transport'（可选）：传输实例;
- 'socket'（可选）：socket.socket实例。

> **注意：** 注意：此方法不应在子类事件循环中重载。对于任何自定义异常处理，请使用set_exception_handler()方法。

18.5.1.15。调试模式

AbstractEventLoop.get_debug( )

获取bool事件循环的调试模式( )。

默认值是True如果环境变量 PYTHONASYNCIODEBUG被设置为非空字符串，False 否则。

3.4.2版本的新功能。

AbstractEventLoop.set_debug(*enabled : bool*)

设置事件循环的调试模式。

3.4.2版本的新功能。

也可以看看： asyncio的调试模式。

18.5.1.16。服务器

*类*asyncio.Server

服务器在套接字上侦听。

由该AbstractEventLoop.create_server()方法和 start_server()函数创建的对象。不要直接实例化类。

close()

停止服务：关闭侦听套接字并将sockets 属性设置为None。

表示现有传入客户端连接的套接字保持打开状态。

服务器异步关闭，使用wait_closed() 协程等待服务器关闭。

coroutine wait_closed()

等到该close()方法完成。

这个方法是一个协程。

sockets

socket.socket服务器正在侦听的对象列表，或者 None服务器是否关闭。

18.5.1.17。处理

*类*asyncio.Handle

回调包装对象返回的AbstractEventLoop.call_soon()，AbstractEventLoop.call_soon_threadsafe()，AbstractEventLoop.call_later()，和AbstractEventLoop.call_at()。

cancel()

取消通话。如果回调已被取消或执行，则此方法不起作用。

18.5.1.18。事件循环示例

18.5.1.18.1。Hello World with call_soon()

使用该AbstractEventLoop.call_soon()方法安排回叫的示例。回调显示，然后停止事件循环：'Hello World'

```
import asyncio

def hello_world(loop):
    print('Hello World')
    loop.stop()

loop = asyncio.get_event_loop()

# Schedule a call to hello_world()
loop.call_soon(hello_world, loop)

# Blocking call interrupted by loop.stop()
loop.run_forever()
loop.close()
```

也可以看看： 在世界您好协同程序示例使用一个协程。

18.5.1.18.2。用call_later（ ）显示当前日期

回叫示例每秒显示当前日期。回调使用该AbstractEventLoop.call_later()方法在5秒内重新安排自己，然后停止事件循环：

```
import asyncio
import datetime

def display_date(end_time, loop):
    print(datetime.datetime.now())
    if (loop.time() + 1.0) < end_time:
        loop.call_later(1, display_date, end_time, loop)
    else:
        loop.stop()

loop = asyncio.get_event_loop()

# Schedule the first call to display_date()
end_time = loop.time() + 5.0
loop.call_soon(display_date, end_time, loop)

# Blocking call interrupted by loop.stop()
loop.run_forever()
loop.close()
```

也可以看看： 在显示当前日期协程的示例使用协程。

18.5.1.18.3。观察读取事件的文件描述符

等到文件描述符使用AbstractEventLoop.add_reader()方法接收到一些数据 ，然后关闭事件循环：

```
import asyncio
try:
    from socket import socketpair
```

```
except ImportError:
    from asyncio.windows_utils import socketpair

# Create a pair of connected file descriptors
rsock, wsock = socketpair()
loop = asyncio.get_event_loop()

def reader():
    data = rsock.recv(100)
    print("Received:", data.decode())
    # We are done: unregister the file descriptor
    loop.remove_reader(rsock)
    # Stop the event loop
    loop.stop()

# Register the file descriptor for read event
loop.add_reader(rsock, reader)

# Simulate the reception of data from the network
loop.call_soon(wsock.send, 'abc'.encode())

# Run the event loop
loop.run_forever()

# We are done, close sockets and the event loop
rsock.close()
wsock.close()
loop.close()
```

也可以看看： 该寄存器使用协议示例等待数据的开放套接字使用由该 `AbstractEventLoop.create_connection()` 方法创建的低级协议 。

该寄存器是一个开放套接字，用于等待数据使用流示例使用由 `open_connection()` 协同函数创建的高级流。

18.5.1.18.4。为SIGINT和SIGTERM设置信号处理程序

为信号注册处理程序SIGINT并SIGTERM使用该 `AbstractEventLoop.add_signal_handler()` 方法：

```
import asyncio
import functools
import os
import signal

def ask_exit(signame):
    print("got signal %s: exit" % signame)
    loop.stop()

loop = asyncio.get_event_loop()
for signame in ('SIGINT', 'SIGTERM'):
    loop.add_signal_handler(getattr(signal, signame),
                            functools.partial(ask_exit, signame))

print("Event loop running forever, press Ctrl+C to interrupt.")
```

```
print("pid %s: send SIGINT or SIGTERM to exit." % os.getpid())
try:
    loop.run_forever()
finally:
    loop.close()
```

此示例仅适用于UNIX。

18.5.2。事件循环

源代码： Lib / asyncio / events.py

18.5.2.1。事件循环函数

以下功能是访问全局策略方法的便捷捷径。请注意，这提供了访问默认策略的权限，除非通过set_event_loop_policy()在执行过程中早些时候调用替代策略。

asyncio.get_event_loop ( )
相当于打电话get_event_loop_policy().get_event_loop()。

asyncio.set_event_loop (*循环*)
相当于打电话get_event_loop_policy().set_event_loop(loop)。

asyncio.new_event_loop ( )
相当于打电话get_event_loop_policy().new_event_loop()。

18.5.2.2。可用的事件循环

asyncio目前提供了两个事件循环的实现： SelectorEventLoop和ProactorEventLoop。

*类*asyncio.SelectorEventLoop
基于selectors模块的事件循环。子类 AbstractEventLoop。

使用平台上最有效的选择器。

在Windows上，只支持套接字（例如：不支持管道）：请参阅select的MSDN文档。

*类*asyncio.ProactorEventLoop
使用“I / O完成端口”又名IOCP的Windows Proactor事件循环。子类AbstractEventLoop。

可用性：Windows。

> **也可以看看：** 有关I / O完成端口的MSDN文档。

ProactorEventLoop在Windows 上使用示例：

```
import asyncio, sys

if sys.platform == 'win32':
    loop = asyncio.ProactorEventLoop()
    asyncio.set_event_loop(loop)
```

18.5.2.3。平台支持

该asyncio模块设计为便携式，但每个平台仍有细微差别，可能不支持所有asyncio功能。

18.5.2.3.1。窗口

Windows事件循环的常见限制：

- create_unix_connection()并且create_unix_server()不受支持：套接字系列socket.AF_UNIX特定于UNIX
- add_signal_handler()并且remove_signal_handler()不支持
- EventLoopPolicy.set_child_watcher()不受支持。ProactorEventLoop支持子流程。它只有一个实现来监视子进程，不需要配置它。

SelectorEventLoop具体限制：

- SelectSelector仅用于支持插座，并且仅限于512个插座。
- add_reader()并且add_writer()只接受插座文件描述符
- 不支持管道（例如：connect_read_pipe()，connect_write_pipe()）
- 子流程不受支持（例如：subprocess_exec()，subprocess_shell()）

ProactorEventLoop具体限制：

- create_datagram_endpoint()（UDP）不受支持
- add_reader()并且add_writer()不支持

Windows上单调时钟的分辨率通常约为15.6毫秒。最佳分辨率是0.5毫秒。分辨率取决于硬件（HPET的可用性）和Windows配置。请参阅asyncio延迟呼叫。

*在版本3.5中更改：*ProactorEventLoop现在支持SSL。

18.5.2.3.2。Mac OS

从小牛（Mac OS 10.9）开始，只有PTY等字符设备才得到很好的支持。在Mac OS 10.5及更早版本中，它们完全不受支持。

在Mac OS 10.6，10.7和10.8，默认的事件循环是SelectorEventLoop其使用selectors.KqueueSelector。selectors.KqueueSelector不支持这些版本的字符设备。在SelectorEventLoop可以与被用于SelectSelector或PollSelector以支持这些版本的Mac OS X上的实施例字符设备：

```
import asyncio
import selectors

selector = selectors.SelectSelector()
loop = asyncio.SelectorEventLoop(selector)
asyncio.set_event_loop(loop)
```

18.5.2.4。事件循环策略和默认策略

事件循环管理通过*策略*模式抽象出来，为定制平台和框架提供最大的灵活性。在整个流程执行过程中，一个全局策略对象根据调用上下文来管理流程可用的事件循环。策略是实现

AbstractEventLoopPolicy接口的对象 。

对于大多数用户来说asyncio ，政策永远不必明确处理，因为默认的全球政策是足够的（见下文）。

模块级功能 get_event_loop()并set_event_loop()提供对由默认策略管理的事件循环的方便访问。

18.5.2.5。事件循环策略接口

事件循环策略必须实现以下接口：

*类*asyncio.AbstractEventLoopPolicy
事件循环策略。

get_event_loop ( )
获取当前上下文的事件循环。

返回实现AbstractEventLoop 接口的事件循环对象。

如果没有为当前上下文设置事件循环，并且当前策略未指定创建一个事件循环，则引发异常。它永远不会回来None。

set_event_loop (*循环*)
将当前上下文的事件循环设置为*循环*。

new_event_loop ( )
根据此策略的规则创建并返回一个新的事件循环对象。

如果需要将此循环设置为当前上下文的事件循环，则set_event_loop()必须明确调用。

默认策略将上下文定义为当前线程，并管理与每个线程交互的事件循环asyncio。如果当前线程尚未拥有与之关联的事件循环，则默认策略的 get_event_loop()方法在从主线程调用时创建一个，但在RuntimeError其他情况下引发。

18.5.2.6。访问全局循环策略

asyncio.get_event_loop_policy ( )
获取当前的事件循环策略。

asyncio.set_event_loop_policy (*政策*)
设置当前事件循环策略。如果*策略*是None ，则恢复默认策略。

18.5.2.7。定制事件循环策略

要实现新的事件循环策略，建议您对具体的默认事件循环策略进行子类化，DefaultEventLoopPolicy 并覆盖要更改行为的方法，例如：

```
class MyEventLoopPolicy(asyncio.DefaultEventLoopPolicy):

    def get_event_loop(self):
        """Get the event loop.

        This may be None or an instance of EventLoop.
        """
        loop = super().get_event_loop()
        # Do something with loop ...
        return loop

asyncio.set_event_loop_policy(MyEventLoopPolicy())
```

18.5.3。任务和协程

源代码： Lib / asyncio / tasks.py

源代码： Lib / asyncio / coroutines.py

18.5.3.1。协同程序

使用的协程asyncio可以使用该语句或使用生成器来实现 。将在Python 3.5中添加类型的协程，并建议如果没有必要支持旧的Python版本。 async defasync def

应该使用基于生成器的协程来修饰@asyncio.coroutine ，尽管这不是严格执行的。装饰器可以与协程兼容，也可以作为文档。基于生成器的协程使用中引入的语法async defyield from**PEP 380** ，而不是原始yield语法。

“协同程序”一词，就像“发生器”一样，用于两个不同的（尽管相关的）概念：

- 定义协程的函数（使用或装饰的函数定义）。如果需要消歧，我们将称之为*协同函数*（返回）。 async def@asyncio.coroutine iscoroutinefunction()True
- 通过调用协程函数获得的对象。该对象表示将最终完成的计算或I / O操作（通常是组合）。如果需要消歧，我们将其称为*协程对象*（ iscoroutine()返回True ）。

协程可以做的事情是：

- result = await future或者- 暂停协程，直到将来完成，然后返回未来的结果，或者引发将被传播的异常。（如果未来取消，它会引发一个例外。）注意任务是期货，关于期货的一切都适用于任务。result = yield from futureCancelledError
- result = await coroutine或者- 等待另一个协程生成一个结果（或引发一个将被传播的异常）。该表达式必须是一个*呼叫*到另一个协程。 result = yield from coroutinecoroutine
- return expression- 产生一个结果给正在等待这个使用await或的协程。 yield from
- raise exception- 在等待这个使用await或的协程中引发一个异常。 yield from

调用协程不会启动其代码运行 - 调用返回的协程对象在您计划执行之前不会执行任何操作。有两种基本的方法可以启动它：从另一个协同程序调用或假设另一个协程已经运行！或者使用函数或方法安排它的执行。 await coroutineyield from coroutine ensure_future()AbstractEventLoop.create_task()

协程（和任务）只能在事件循环运行时运行。

@asyncio.coroutine

装饰者标记基于生成器的协程。这使得生成器可以调用协程，并且还可以通过协程来调用生成器 ，例如使用 表达式。 yield fromasync defasync defawait

没有必要自己装饰协程。 async def

如果发生器在销毁之前未被排出，则会记录一条错误消息。请参阅检测从未计划的协同程序。

注意: 在本文档中，某些方法被记录为协程，即使它们是纯Python函数返回的Future。这是有意在未来调整这些功能的实施的自由。如果需要在回调式代码中使用这样的函数，请将其结果包括在内ensure_future()。

18.5.3.1.1。例如：Hello World协程

协程显示示例："Hello World"

```
import asyncio

async def hello_world():
    print("Hello World!")

loop = asyncio.get_event_loop()
# Blocking call which returns when the hello_world() coroutine is done
loop.run_until_complete(hello_world())
loop.close()
```

也可以看看: 带有call_soon（）示例的Hello World使用该AbstractEventLoop.call_soon()方法安排回调。

18.5.3.1.2。例如：协程显示当前日期

使用该sleep()功能，协程在5秒钟内每秒显示当前日期的示例：

```
import asyncio
import datetime

async def display_date(loop):
    end_time = loop.time() + 5.0
    while True:
        print(datetime.datetime.now())
        if (loop.time() + 1.0) >= end_time:
            break
        await asyncio.sleep(1)

loop = asyncio.get_event_loop()
# Blocking call which returns when the display_date() coroutine is done
loop.run_until_complete(display_date(loop))
loop.close()
```

也可以看看: 在显示当前日期与call_later（）的示例使用与回调AbstractEventLoop.call_later()方法。

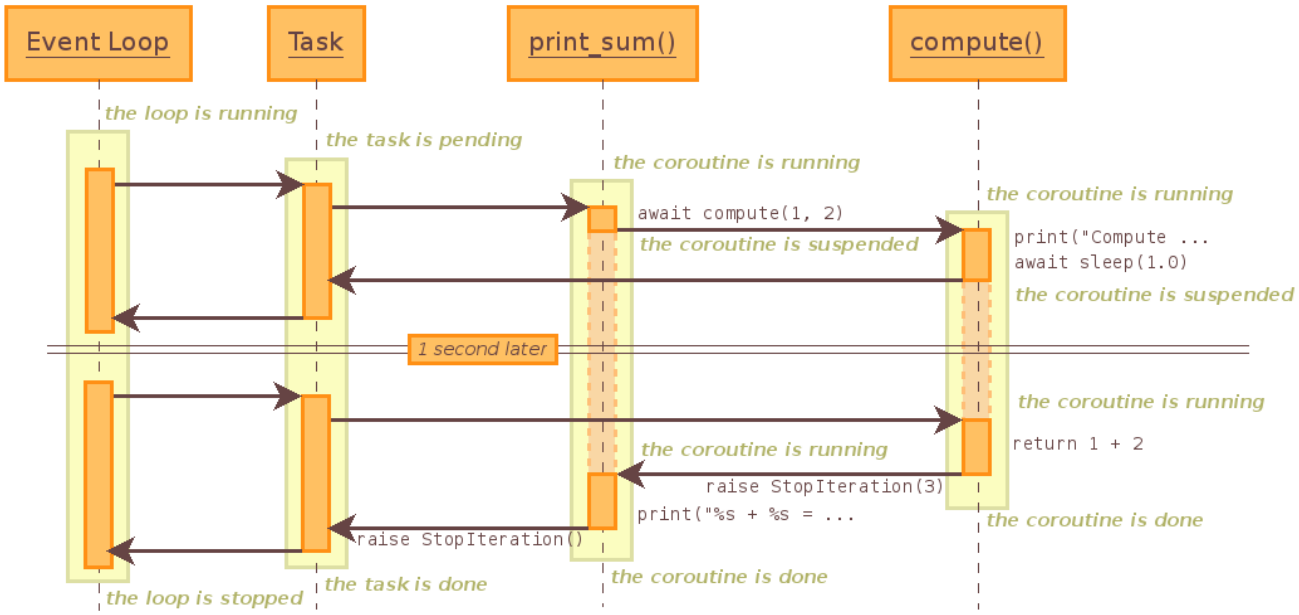
18.5.3.1.3。例子：链接程序

示例链接协程：

```
import asyncio

async def compute(x, y):
    print("Compute %s + %s ..." % (x, y))
    await asyncio.sleep(1.0)
    return x + y

async def print_sum(x, y):
    result = await compute(x, y)
    print("%s + %s = %s" % (x, y, result))

loop = asyncio.get_event_loop()
loop.run_until_complete(print_sum(1, 2))
loop.close()
```

compute()被链接到print_sum()：print_sum()协程compute()在返回结果之前等待完成。

示例的序列图：



当“任务” AbstractEventLoop.run_until_complete()获取协程对象而不是任务时，由该方法创建。

该图显示了控制流程，并没有描述内部工作情况。例如，睡眠协程创建一个内部未来，用于AbstractEventLoop.call_later()在1秒内唤醒任务。

18.5.3.2。InvalidStateError

*异常*asyncio.InvalidStateError

在这种状态下不允许操作。

18.5.3.3。TimeoutError

*异常*asyncio.TimeoutError

该操作超出了规定的期限。

注意: 这个异常与内建TimeoutError异常不同！

18.5.3.4。未来

class asyncio. Future (***, *loop = None*)

这个类*几乎*兼容concurrent.futures.Future。

区别：

- result()并且exception()在未来尚未完成时不要超时参数并引发异常。
- 注册的回调函数add_done_callback()总是通过事件循环来调用call_soon()。
- 这个类与包中的wait()和 as_completed()函数 不兼容concurrent.futures。

这个类不是线程安全的。

cancel ( )

取消未来并安排回调。

如果未来已经完成或取消，请返回False。否则，将未来的状态更改为取消，安排回调并返回 True。

cancelled ( )

True如果未来取消，则返回。

done ( )

True如果未来完成，则返回。

完成意味着结果/例外可用，或未来被取消。

result ( )

返回这个未来代表的结果。

如果未来取消，则引发CancelledError。如果未来的结果尚不可用，则会提出InvalidStateError。如果未来完成并设置了异常，则会引发此异常。

exception ( )

返回在未来设置的例外。

None只有在未来完成时才会返回异常（或者如果没有设置异常）。如果未来取消，则引发 CancelledError。如果未来还没有完成，就会引发 InvalidStateError。

add_done_callback (*fn*)

添加一个回调，在未来完成时运行。

回调被调用一个参数 - 未来的对象。如果将来调用时已经完成，则回调计划为call_soon()。

使用 functools.partial 将参数传递给回调。例如，会打电话。`fut.add_done_callback(functools.partial(print, "Future:", flush=True))` `print("Future:", fut, flush=True)`

`remove_done_callback(fn)`

从“完成时调用”列表中删除所有回调实例。

返回删除的回调数。

`set_result(结果)`

标记未来并设定结果。

如果在调用此方法时未来已完成，则引发 `InvalidStateError`。

`set_exception(例外)`

标记未来完成并设置例外。

如果在调用此方法时未来已完成，则引发 `InvalidStateError`。

18.5.3.4.1。示例：使用run_until_complete（）的未来

结合a `Future`和协程函数的例子：

```
import asyncio

async def slow_operation(future):
    await asyncio.sleep(1)
    future.set_result('Future is done!')

loop = asyncio.get_event_loop()
future = asyncio.Future()
asyncio.ensure_future(slow_operation(future))
loop.run_until_complete(future)
print(future.result())
loop.close()
```

协程函数负责计算（需要1秒），并将结果存储到将来。该 `run_until_complete()` 方法等待未来的完成。

注意： 该`run_until_complete()`方法在内部使用`add_done_callback()`未来完成时通知的 方法。

18.5.3.4.2。示例：使用run_forever（）的未来

前面的例子可以用不同的 `Future.add_done_callback()` 方法来描述控制流的方法：

```
import asyncio

async def slow_operation(future):
    await asyncio.sleep(1)
```

```
    future.set_result('Future is done!')

def got_result(future):
    print(future.result())
    loop.stop()

loop = asyncio.get_event_loop()
future = asyncio.Future()
asyncio.ensure_future(slow_operation(future))
future.add_done_callback(got_result)
try:
    loop.run_forever()
finally:
    loop.close()
```

在这个例子中，未来用于链接slow_operation()到 got_result() ：何时slow_operation()完成，got_result()是否与结果一起被调用。

18.5.3.5。任务

class asyncio. Task (*coro* , * , *loop = None*)

安排协同程序的执行：将来包装它。任务是。的一个子类Future。

任务负责在事件循环中执行协程对象。如果未来包装的协同程序产生，则任务暂停包装的协程的执行并等待未来的完成。未来完成后，包装的协程的执行会随着结果或未来的例外而重新开始。

事件循环使用协作式调度：事件循环一次只能运行一项任务。如果其他事件循环在不同的线程中运行，其他任务可以并行运行。当任务等待未来完成时，事件循环执行一项新任务。

取消任务与取消未来不同。调用cancel()会抛出一个 CancelledError包裹的协程。cancelled()只有True在包装的协程没有捕获 CancelledError异常或引发 CancelledError异常时才会返回。

如果挂起的任务被销毁，其包装的协程的执行没有完成。这可能是一个错误，并且会记录一条警告：请参阅挂起的任务被销毁。

不要直接创建Task实例：使用ensure_future() 函数或AbstractEventLoop.create_task()方法。

这个类不是线程安全的。

classmethod all_tasks (*loop = None*)

为事件循环返回一组所有任务。

默认情况下，返回当前事件循环的所有任务。

classmethod current_task (*loop = None*)

在事件循环中返回当前正在运行的任务None。

默认情况下返回当前事件循环的当前任务。

None在不被调用的情况下被返回Task。

cancel ( )

请求此任务取消自己。

这样CancelledError可以在事件循环的下一个循环中安排一个被抛入包装的协程。协程然后有机会使用try / except / finally来清理甚至拒绝请求。

与Future.cancel()此不同的是，这并不能保证该任务将被取消：该异常可能被捕获并采取行动，推迟取消任务或完全阻止取消。该任务也可能会返回一个值或引发一个不同的异常。

此方法被调用后立即cancelled()不会返回True（除非任务已被取消）。当包装的协程结束时，任务将被标记为取消 CancelledError（即使 cancel()未被调用）。

get_stack (*, *limit* = *None*)

返回此任务协同程序的堆栈帧列表。

如果协程没有完成，则返回暂停的协议栈。如果协程已成功完成或取消，则返回空列表。如果协程由异常终止，则返回追溯帧列表。

帧总是从最旧到最新排列。

可选限制提供了返回帧的最大数量; 默认情况下会返回所有可用的帧。它的含义根据是否返回堆栈或回溯而有所不同：返回堆栈的最新帧，但返回最早的回溯帧。（这与追溯模块的行为相匹配。）

由于我们无法控制的原因，只有一个堆栈帧返回一个挂起的协程。

print_stack (*, *limit* = *None*, *file* = *None*)

打印此任务的协同程序的堆栈或回溯。

对于由get_stack（）检索的帧，这会产生与回溯模块类似的输出。limit参数被传递给get_stack（）。文件参数是写入输出的I / O流; 默认输出写入sys.stderr。

18.5.3.5.1。示例：并行执行任务

并行执行3个任务（A，B，C）的示例：

```
import asyncio

async def factorial(name, number):
    f = 1
    for i in range(2, number+1):
        print("Task %s: Compute factorial(%s)..." % (name, i))
        await asyncio.sleep(1)
        f *= i
    print("Task %s: factorial(%s) = %s" % (name, number, f))
```

```
loop = asyncio.get_event_loop()
loop.run_until_complete(asyncio.gather(
    factorial("A", 2),
    factorial("B", 3),
    factorial("C", 4),
))
loop.close()
```

输出：

```
Task A: Compute factorial(2)...
Task B: Compute factorial(2)...
Task C: Compute factorial(2)...
Task A: factorial(2) = 2
Task B: Compute factorial(3)...
Task C: Compute factorial(3)...
Task B: factorial(3) = 6
Task C: Compute factorial(4)...
Task C: factorial(4) = 24
```

创建任务时会自动计划执行任务。所有任务完成后，事件循环停止。

18.5.3.6。任务功能

注意： 在下面的函数中，可选的*循环*参数允许显式设置基础任务或协程使用的事件循环对象。如果没有提供，则使用默认的事件循环。

asyncio.**as_completed**(*fs* , * , *loop = None* , *timeout = None*)

返回一个迭代器，其值在等待时是Future 实例。

asyncio.TimeoutError如果在所有期货完成之前发生超时，则引发。

例：

```
for f in as_completed(fs):
    result = yield from f  # The 'yield from' may raise
    # Use result
```

注意： 期货f不一定是fs的成员。

asyncio.**ensure_future**(*coro_or_future* , * , *loop = None*)

计划协同对象的执行：将来包装它。返回一个Task对象。

如果参数是a Future ，则直接返回。

3.4.4版本的新功能。

*在版本3.5.1中更改：*该函数接受任何等待对象。

也可以看看： 该AbstractEventLoop.create_task()方法。

asyncio. **async** (*coro_or_future* , * , *loop = None*)

已弃用的别名ensure_future()。

自3.4.4版以来已弃用。

asyncio. **wrap_future** (*future* , * , *loop = None*)

将concurrent.futures.Future物体包裹在物体中Future 。

asyncio. **gather** (* *coros_or_futures* , *loop = None* , *return_exceptions = False*)

返回给定协同对象或未来的未来聚合结果。

所有期货必须共享相同的事件循环。如果所有任务都成功完成，则返回的未来结果是结果列表（按照原始序列的顺序，不一定是结果到达的顺序）。如果*return_exceptions*为true，则将任务中的异常视为成功结果，并收集到结果列表中; 否则，第一个引发的异常将立即传播到返回的未来。

取消：如果外部未来取消，所有儿童（尚未完成）也将被取消。如果有任何孩子被取消，则将其视为升起CancelledError- 在这种情况下外部未来*不会*被取消。（这是为了防止取消一个孩子导致其他孩子被取消。）

asyncio. **iscoroutine** (*obj*)

返回True如果*OBJ*是一个协程对象，其可以基于一个发生器或协程。 async def

asyncio. **iscoroutinefunction** (*func*)

返回True如果*FUNC*被确定为是一个协程功能，其可以是装饰发生器功能或 功能。 async def

asyncio. **run_coroutine_threadsafe** (*coro* , *loop*)

将协程对象提交给给定的事件循环。

返回a concurrent.futures.Future来访问结果。

这个函数是从不同于事件循环运行的线程调用的。用法：

```
# Create a coroutine
coro = asyncio.sleep(1, result=3)
# Submit the coroutine to a given loop
future = asyncio.run_coroutine_threadsafe(coro, loop)
# Wait for the result with an optional timeout argument
assert future.result(timeout) == 3
```

如果协程中出现异常，则会通知返回的未来。它也可以用来取消事件循环中的任务：

```
try:
    result = future.result(timeout)
except asyncio.TimeoutError:
    print('The coroutine took too long, cancelling the task...')
    future.cancel()
except Exception as exc:
    print('The coroutine raised an exception: {!r}'.format(exc))
```

```
else:
    print('The coroutine returned: {!r}'.format(result))
```

请参阅 文档的并发性和多线程部分。

> **注意：** 与模块中的其他函数不同， `run_coroutine_threadsafe()` 需要显式传递*循环*参数。

3.5.1版本的新功能。

*协程*asyncio.**sleep**(*延迟*, *结果=无*, ***, *循环=无*)

创建一个在给定时间之后完成的协程（以秒为单位）。如果提供了*结果*，则在协程完成时将其发送给调用者。

睡眠的解决取决于事件循环的粒度。

这个函数是一个协程。

*协程*asyncio.**shield**(*arg*, ***, *loop = None*)

等待未来，避免取消。

该声明：

```
res = yield from shield(something())
```

与声明完全相同：

```
res = yield from something()
```

*除非*包含它的协程被取消，否则运行中的任务`something()`不会被取消。从这个角度来看`something()`，取消没有发生。但它的调用者仍然被取消，所以yield-from表达式仍然存在`CancelledError`。注意：如果`something()`通过其他方式取消，这仍将取消`shield()`。

如果你想完全忽略取消（不推荐），你可以结合`shield()`try / except从句，如下所示：

```
try:
    res = yield from shield(something())
except CancelledError:
    res = None
```

*协程*asyncio.**wait**(*期货*, ***, *循环=无*, *超时=无*, *return_when = ALL_COMPLETED*)

等待序列*期货*给出的期货和协同对象 完成。协程将包裹在任务中。返回两组 `Future`:(完成，待定）。

序列*期货*不能为空。

*超时*可用于控制返回之前等待的最大秒数。 *超时*可以是一个int或float。如果没有指定*超时*，或者`None`等待时间没有限制。

*return_when*指示此函数何时应返回。它必须是`concurrent.futures`模块的以下常量之一：

不变	描述
FIRST_COMPLETED	该功能将在任何未来完成或取消时返回。
FIRST_EXCEPTION	该函数将在任何未来通过引发异常完成时返回。如果没有未来引发异常，那么它相当于ALL_COMPLETED。
ALL_COMPLETED	该功能将在所有期货结束或被取消时返回。

这个函数是一个协程。

用法：

```
done, pending = yield from asyncio.wait(fs)
```

> **注意：** 这不会提高asyncio.TimeoutError！在超时发生时未完成的期货在第二组中返回。

*协程*asyncio.wait_for(*fut*, *timeout*, *, *loop = None*)

等待单个Future或协程对象 完成超时。如果*超时*是None，块等到将来完成。

协程将被包裹Task。

返回Future或协程的结果。发生超时时，它会取消该任务并引发asyncio.TimeoutError。为避免任务取消，请将其包装shield()。

如果等待取消，未来的*fut*也会被取消。

这个函数是一个协程，用法如下：

```
result = yield from asyncio.wait_for(fut, 60.0)
```

*在版本3.4.3中更改：*如果等待取消，未来的*fut*现在也被取消。

18.5.4。传输和协议（基于回调的API）

源代码： Lib / asyncio / transports.py

源代码： Lib / asyncio / protocols.py

18.5.4.1。运输

传输是asyncio为了抽象各种通信信道而提供的类。你通常不会自己实例化一个交通工具; 相反，您将调用一个AbstractEventLoop方法来创建传输并尝试启动底层通信通道，并在成功时回复您。

通信通道建立后，传输总是与协议实例配对。然后协议可以为各种目的调用运输方法。

asyncio目前实现了TCP，UDP，SSL和子流程管道的传输。交通工具上可用的方法取决于交通工具的种类。

运输类不是线程安全的。

*在版本3.6中更改：*套接字选项TCP_NODELAY现在默认设置。

18.5.4.1.1。BaseTransport

*类*asyncio.BaseTransport

运输基础类。

close()

关闭运输。如果传输具有传出数据的缓冲区，则缓存的数据将被异步刷新。将不会收到更多数据。在刷新所有缓冲数据之后，协议的connection_lost()方法将None以其参数被调用 。

is_closing()

True如果运输工具正在关闭或关闭，则返回。

3.5.1版本的新功能。

get_extra_info(*名称*，*默认=无*)

返回可选的运输信息。 *name*是表示要获取的特定于传输的信息的字符串，*默认值*是在信息不存在时返回的值。

此方法允许传输实现轻松地公开特定于频道的信息。

- 插座：
  - 'peername'：套接字所连接的远程地址，结果socket.socket.getpeername()（None出错）
  - 'socket'：socket.socket实例
  - 'sockname'：套接字自己的地址，结果 socket.socket.getsockname()

- SSL套接字：
  - 'compression'：压缩算法用作字符串，或者None连接未被压缩; 的结果ssl.SSLSocket.compression()
  - 'cipher'：包含正在使用的密码名称的三值元组，定义其使用的SSL协议版本以及正在使用的密码位数; 的结果 ssl.SSLSocket.cipher()
  - 'peercert'：同行证书; 的结果 ssl.SSLSocket.getpeercert()
  - 'sslcontext'：ssl.SSLContext实例
  - 'ssl_object'：ssl.SSLObject或ssl.SSLSocket 实例
- 管：
  - 'pipe'：管道对象
- 子：
  - 'subprocess'：subprocess.Popen实例

set_protocol (*协议*)

设置一个新的协议。切换协议只应在两个协议都记录在案才能支持交换机时完成。

3.5.3版本的新功能。

get_protocol ( )

返回当前协议。

3.5.3版本的新功能。

版本3.5.1中已更改：'ssl_object'信息已添加到SSL套接字中。

18.5.4.1.2。ReadTransport

*类*asyncio. ReadTransport

只读传输接口。

pause_reading ( )

暂停运输的接收端。 在调用data_received()之前，没有数据会传递给协议的方法resume_reading()。

resume_reading ( )

恢复接收端。 data_received()如果某些数据可用于读取，该协议的方法将再次被调用。

18.5.4.1.3。WriteTransport

*类*asyncio. WriteTransport

只写传输接口。

abort ( )

立即关闭运输工具，无需等待挂起的操作完成。缓冲的数据将会丢失。将不会收到更多数据。该协议的connection_lost()方法最终将None以其参数被调用。

can_write_eof ( )

True如果运输支持write_eof()，False则返回，否则返回。

get_write_buffer_size()

返回传输器使用的输出缓冲区的当前大小。

get_write_buffer_limits()

获取写流控制的*高*限制和*低*限制限制。返回一个元组，其中*低位*和*高位*是正数字节。(low, high)

使用set_write_buffer_limits()设置限制。

3.4.2版本的新功能。

set_write_buffer_limits(*高=无*，*低=无*)

设置写流量控制的*高*限制和*低*限制限制。

当调用协议pause_writing()和resume_writing()方法时，这两个值（以字节数衡量）控制。如果规定，低水限制必须小于或等于高水限制。不论*高*或*低*都可能是负面的。

pause_writing()当缓冲区大小变得大于或等于*高*值时被调用。如果写入已暂停，resume_writing()则在缓冲区大小小于或等于*低值*时调用。

默认值是特定于实现的。如果只给出高水限制，则低水限制默认为特定于实施的值小于或等于高水限制。将*高位*设置为零也将*低位*设置为零，并pause_writing()在缓冲区变为非空时导致被调用。将*低位*设置为零将导致 resume_writing()仅在缓冲区为空时调用。对任一限制使用零通常是次优的，因为它减少了同时进行I / O和计算的机会。

使用get_write_buffer_limits()得到限制。

write(*数据*)

将一些*数据*字节写入传输。

这种方法不会阻止; 它缓冲数据并安排它被异步发送出去。

writelines(*list_of_data*)

将一个列表（或任何可迭代的）数据字节写入传输。这在功能上等同于调用write()迭代产生的每个元素，但可以更有效地实现。

write_eof()

刷新缓冲数据后关闭传输的写入结束。数据仍可能被接收。

NotImplementedError如果传输（例如SSL）不支持半关闭，则可以引发此方法。

18.5.4.1.4。DatagramTransport

DatagramTransport.sendto(*data*，*addr = None*)

将*数据*字节发送给由*addr*（依赖于传输的目标地址）给定的远程对等点。如果*addr*是None，则将数据发送到创建传输时指定的目标地址。

这种方法不会阻止; 它缓冲数据并安排它被异步发送出去。

DatagramTransport. abort ( )

立即关闭运输工具，无需等待挂起的操作完成。缓冲的数据将会丢失。将不会收到更多数据。该协议的connection_lost()方法最终将None以其参数被调用。

18.5.4.1.5。BaseSubprocessTransport

*类*asyncio. BaseSubprocessTransport

get_pid ( )

以整数形式返回子进程进程标识。

get_pipe_transport (*fd*)

返回与整型文件描述符*fd*对应的通信管道的传输：

- 0：标准输入（*标准输入*）的可读流式传输，或者None如果子流程未使用stdin=PIPE
- 1：标准输出（*标准输出*）的可写流式传输，或者None如果子流程不是使用stdout=PIPE
- 2：标准错误（*stderr*）的可写流式传输，或者None如果子进程没有被创建stderr=PIPE
- 其他*fd*：None

get_returncode ( )

返回子进程返回码作为一个整数，或者None如果它没有返回，类似于subprocess.Popen.returncode属性。

kill ( )

杀死子进程，如subprocess.Popen.kill()。

在POSIX系统上，该函数向子进程发送SIGKILL。在Windows上，这个方法是一个别名terminate()。

send_signal (*信号*)

将*信号*编号发送到子过程，如在 subprocess.Popen.send_signal()。

terminate ( )

如要求子进程停止subprocess.Popen.terminate()。此方法是该方法的别名close()。

在POSIX系统上，此方法将SIGTERM发送到子进程。在Windows上，调用Windows API函数TerminateProcess ( ) 来停止子进程。

close ( )

terminate()如果子进程尚未返回，则通过调用方法来请求子进程停止，并关闭所有管道的传输（*stdin*，*stdout*和*stderr*）。

18.5.4.2。协议

asyncio提供您可以继承的基类来实现您的网络协议。这些类与传输一起使用（见下文）：协议解析传入数据并要求写入传出数据，而传输负责实际的I / O和缓冲。

当继承协议类时，建议您重写某些方法。这些方法是回调函数：它们将在某些事件中由传输器调用（例如，在收到某些数据时）。除非你正在实施交通工具，否则你不应该自己打电话给他们。

注意： 所有的回调都有默认的实现，都是空的。因此，您只需要针对您感兴趣的事件实施回调。

18.5.4.2.1。协议类

*类*asyncio. Protocol
用于实现流协议的基类（用于例如TCP和SSL传输）。

*类*asyncio. DatagramProtocol
实现数据报协议的基类（用于例如UDP传输）。

*类*asyncio. SubprocessProtocol
用于实现与子进程通信的基类（通过一组单向管道）。

18.5.4.2.2。连接回调

这些回调可能会被调用Protocol ，DatagramProtocol 并且SubprocessProtocol实例：

BaseProtocol. connection_made（*运输*）
在建立连接时调用。

该*传输*参数是代表连接的传输。如果需要，您有责任将它存储在某个地方（例如作为属性）。

BaseProtocol. connection_lost（*exc* ）
当连接丢失或关闭时调用。

该参数是一个异常对象或None。后者意味着收到一个正常的EOF，或者连接在连接的这一侧被中止或关闭。

connection_made()并且connection_lost() 在每次成功连接时被调用一次。所有其他回调函数将在这两种方法之间调用，这样可以在协议实现中更轻松地进行资源管理。

以下回调只能在SubprocessProtocol 实例上调用：

SubprocessProtocol. pipe_data_received（*fd* ，*data* ）
当子进程将数据写入其stdout或stderr管道时调用。 *fd*是管道的整数文件描述符。 *数据*是包含数据的非空字节对象。

SubprocessProtocol. pipe_connection_lost（*fd* ，*exc* ）

当与子进程通信的其中一个管道关闭时调用。 *fd*是已关闭的整数文件描述符。

SubprocessProtocol.process_exited()

当子进程退出时调用。

18.5.4.2.3。流媒体协议

以下回调在Protocol实例上被调用：

Protocol.data_received(*数据*)

当收到一些数据时调用。 *数据*是包含传入数据的非空字节对象。

> **注意：** 数据是否被缓冲，分块或重新组装取决于传输。一般来说，你不应该依赖特定的语义，而是使你的解析具有通用性和灵活性。但是，数据总是以正确的顺序接收。

Protocol.eof_received()

当另一端发信号时调用它不会发送更多数据（例如通过调用write_eof()，如果另一端也使用asyncio）。

此方法可能会返回一个错误值（包括None），在这种情况下，传输将自行关闭。相反，如果此方法返回真值，关闭传输将取决于协议。由于默认实现返回None，它隐式关闭连接。

> **注意：** 某些传输（如SSL）不支持半关闭连接，在这种情况下，从此方法返回true将不会阻止关闭连接。

data_received()在连接过程中可以调用任意次数。然而，eof_received()最多data_received()只能调用一次，如果调用它，将不会在它之后调用。

状态机：

```
开始 - > connection_made() [ - > data_received()*] [ - > eof_received() ？] - >
connection_lost()- >结束
```

18.5.4.2.4。数据报协议

以下回调在DatagramProtocol实例上调用。

DatagramProtocol.datagram_received(*data*，*addr*)

在收到数据报时调用。 *数据*是包含传入数据的字节对象。 *addr*是发送数据的对端的地址;确切的格式取决于运输。

DatagramProtocol.error_received(*exc*)

当以前的发送或接收操作产生一个时调用 OSError。 *exc*是OSError例子。

当传输（例如UDP）检测到无法将数据报传送给其接收者时，此方法在罕见情况下被调用。但在许多情况下，无法传递的数据报将被无声丢弃。

18.5.4.2.5。流量控制回调

这些回调可能会被调用Protocol ， DatagramProtocol并且SubprocessProtocol实例：

BaseProtocol.**pause_writing**( )
当交通缓冲区超过高水位标志时调用。

BaseProtocol.**resume_writing**( )
当运输缓冲液在低水位标记下方流失时调用。

pause_writing()并且resume_writing()调用是成对的 - pause_writing()在缓冲区严格超过高水位标记时调用一次（即使后续写入增加了缓冲区大小甚至更多），并且resume_writing()当缓冲区大小达到低位标记时最终调用一次。

注意： 如果缓冲区大小等于高水位标记， pause_writing()则不被调用 - 它必须严格超过。相反，resume_writing()当缓冲区大小等于或低于低水位标记时调用。这些最终条件对于确保任何一个商标为零时预期的结果都很重要。

注意： 在BSD系统（OS X，FreeBSD等）上不支持流量控制DatagramProtocol，因为写入过多数据包导致的发送失败不容易被检测到。套接字始终显示为“就绪”并且多余的数据包被丢弃; 一个 OSErrorerrno errno.ENOBUFS可能会或可能不会被提出; 如果它被提出，它将被报告， DatagramProtocol.error_received()但被忽略。

18.5.4.2.6。协程和协议

协程可以使用协议方法进行调度ensure_future()，但不能保证执行顺序。协议不知道在协议方法中创建的协程，因此不会等待它们。

要有可靠的执行顺序，请在协程中使用流对象。例如， 协程可用于等待写缓冲区刷新。 yield fromStreamWriter.drain()

18.5.4.3。协议示例

18.5.4.3.1。TCP回显客户端协议

使用该AbstractEventLoop.create_connection()方法的TCP回显客户端，发送数据并等待连接关闭：

```
import asyncio

class EchoClientProtocol(asyncio.Protocol):
    def __init__(self, message, loop):
        self.message = message
        self.loop = loop

    def connection_made(self, transport):
        transport.write(self.message.encode())
```

```
        print('Data sent: {!r}'.format(self.message))

    def data_received(self, data):
        print('Data received: {!r}'.format(data.decode()))

    def connection_lost(self, exc):
        print('The server closed the connection')
        print('Stop the event loop')
        self.loop.stop()

loop = asyncio.get_event_loop()
message = 'Hello World!'
coro = loop.create_connection(lambda: EchoClientProtocol(message, loop),
                              '127.0.0.1', 8888)
loop.run_until_complete(coro)
loop.run_forever()
loop.close()
```

事件循环运行两次。该 run_until_complete() 方法是优选的此短例如，如果服务器不监听，而不必编写短协程来处理异常和停止正在运行的循环，提高异常。在run_until_complete()退出时，循环不再运行，因此在发生错误时不需要停止循环。

也可以看看: 使用流 示例的TCP回显客户端使用该asyncio.open_connection()函数。

18.5.4.3.2。TCP回显服务器协议

使用TCP回显服务器的AbstractEventLoop.create_server()方法，发送回收到的数据并关闭连接：

```
import asyncio

class EchoServerClientProtocol(asyncio.Protocol):
    def connection_made(self, transport):
        peername = transport.get_extra_info('peername')
        print('Connection from {}'.format(peername))
        self.transport = transport

    def data_received(self, data):
        message = data.decode()
        print('Data received: {!r}'.format(message))

        print('Send: {!r}'.format(message))
        self.transport.write(data)

        print('Close the client socket')
        self.transport.close()

loop = asyncio.get_event_loop()
# Each client connection will create a new protocol instance
coro = loop.create_server(EchoServerClientProtocol, '127.0.0.1', 8888)
server = loop.run_until_complete(coro)

# Serve requests until Ctrl+C is pressed
print('Serving on {}'.format(server.sockets[0].getsockname()))
```

```
try:
    loop.run_forever()
except KeyboardInterrupt:
    pass

# Close the server
server.close()
loop.run_until_complete(server.wait_closed())
loop.close()
```

Transport.close()可以在WriteTransport.write()即使数据尚未在套接字上发送后立即调用 ：两种方法都是异步的。不需要，因为这些传输方法不是协同程序。yield from

也可以看看： 使用流 示例的TCP回显服务器使用该asyncio.start_server()函数。

18.5.4.3.3。UDP回显客户端协议

UDP回应客户端使用该AbstractEventLoop.create_datagram_endpoint() 方法，当我们收到答案时发送数据并关闭传输：

```
import asyncio

class EchoClientProtocol:
    def __init__(self, message, loop):
        self.message = message
        self.loop = loop
        self.transport = None

    def connection_made(self, transport):
        self.transport = transport
        print('Send:', self.message)
        self.transport.sendto(self.message.encode())

    def datagram_received(self, data, addr):
        print("Received:", data.decode())

        print("Close the socket")
        self.transport.close()

    def error_received(self, exc):
        print('Error received:', exc)

    def connection_lost(self, exc):
        print("Socket closed, stop the event loop")
        loop = asyncio.get_event_loop()
        loop.stop()

loop = asyncio.get_event_loop()
message = "Hello World!"
connect = loop.create_datagram_endpoint(
    lambda: EchoClientProtocol(message, loop),
    remote_addr=('127.0.0.1', 9999))
transport, protocol = loop.run_until_complete(connect)
loop.run_forever()
```

```
transport.close()
loop.close()
```

18.5.4.3.4。UDP回显服务器协议

使用UDP回显服务器的AbstractEventLoop.create_datagram_endpoint() 方法，发送回收到的数据：

```
import asyncio

class EchoServerProtocol:
    def connection_made(self, transport):
        self.transport = transport

    def datagram_received(self, data, addr):
        message = data.decode()
        print('Received %r from %s' % (message, addr))
        print('Send %r to %s' % (message, addr))
        self.transport.sendto(data, addr)

loop = asyncio.get_event_loop()
print("Starting UDP server")
# One protocol instance will be created to serve all client requests
listen = loop.create_datagram_endpoint(
    EchoServerProtocol, local_addr=('127.0.0.1', 9999))
transport, protocol = loop.run_until_complete(listen)

try:
    loop.run_forever()
except KeyboardInterrupt:
    pass

transport.close()
loop.close()
```

18.5.4.3.5。使用协议注册一个打开的套接字以等待数据

等到套接字使用AbstractEventLoop.create_connection()带有协议的方法接收数据 ，然后关闭事件循环

```
import asyncio
try:
    from socket import socketpair
except ImportError:
    from asyncio.windows_utils import socketpair

# Create a pair of connected sockets
rsock, wsock = socketpair()
loop = asyncio.get_event_loop()

class MyProtocol(asyncio.Protocol):
    transport = None
```

```
    def connection_made(self, transport):
        self.transport = transport

    def data_received(self, data):
        print("Received:", data.decode())

        # We are done: close the transport (it will call connection_lost())
        self.transport.close()

    def connection_lost(self, exc):
        # The socket has been closed, stop the event loop
        loop.stop()

# Register the socket to wait for data
connect_coro = loop.create_connection(MyProtocol, sock=rsock)
transport, protocol = loop.run_until_complete(connect_coro)

# Simulate the reception of data from the network
loop.call_soon(wsock.send, 'abc'.encode())

# Run the event loop
loop.run_forever()

# We are done, close sockets and the event loop
rsock.close()
wsock.close()
loop.close()
```

也可以看看： 该手表读取事件文件描述符的例子使用了低级别的 `AbstractEventLoop.add_reader()` 方法来注册一个套接字的文件描述符。

该寄存器是一个开放套接字，用于等待数据使用流示例使用由 `open_connection()` 协同函数创建的高级流。

18.5.5。Streams（基于协议的API）

源代码： Lib / asyncio / streams.py

18.5.5.1。流函数

注意： 这个模块中的顶层函数只是作为方便包装器而已。这里没有什么特别的，如果他们没有做到你想要的东西，可以随意复制他们的代码。

*协程*asyncio.open_connection（*host = None*，*port = None*，***，*loop = None*，*limit = None*，*** kwds* ）

用于create_connection()返回（读取器，写入器）对的包装器。

读者返回的是一个StreamReader实例; 作家是一个StreamWriter实例。

参数是AbstractEventLoop.create_connection()除*protocol_factory*外的所有常用参数；最常见的是位置主机和端口，以及各种可选的关键字参数。

其他可选的关键字参数是*循环*（设置要使用的事件循环实例）和*限制*（设置传递给它的缓冲区限制 StreamReader ）。

这个函数是一个协程。

*协同程序*asyncio.start_server（*client_connected_cb*，*host = None*，*port = None*，***，*loop = None*，*limit = None*，*** kwds* ）

启动套接字服务器，并为每个连接的客户端回拨一个回叫。返回值与create_server()。相同。

该*client_connected_cb*参数被称为有两个参数：*client_reader*，*client_writer*。*client_reader*是一个 StreamReader对象，而*client_writer*是一个 StreamWriter对象。所述*client_connected_cb*参数可以是一个普通的回调函数或一个协程函数；如果它是一个协程函数，它会自动转换成一个Task。

create_server()除了*protocol_factory*之外，其余的参数都是通常的参数；最常见的是位置*主机*和*端口*，以及各种可选的关键字参数。

其他可选的关键字参数是*循环*（设置要使用的事件循环实例）和*限制*（设置传递给它的缓冲区限制 StreamReader ）。

这个函数是一个协程。

*协程*asyncio.open_unix_connection（*path = None*，***，*loop = None*，*limit = None*，*** kwds* ）

用于create_unix_connection()返回（读取器，写入器）对的包装器。

请参阅open_connection()有关退货价值和其他详细信息。

这个函数是一个协程。

可用性：UNIX。

*协程*asyncio.start_unix_server（*client_connected_cb*，*path* = *None*，*，loop* = *None*，*limit* = *None*，** *kwds* ）

启动一个UNIX Domain Socket服务器，并为每个连接的客户端回拨一个回叫。

请参阅start_server()有关退货价值和其他详细信息。

这个函数是一个协程。

可用性：UNIX。

18.5.5.2。StreamReader的

class asyncio.StreamReader（*limit* = *None*，*loop* = *None* ）

这个类不是线程安全的。

exception（ ）

获得例外。

feed_eof（ ）

确认EOF。

feed_data（*数据*）

将*数据*字节送入内部缓冲区。任何等待数据的操作都将被恢复。

set_exception（*exc* ）

设置例外。

set_transport（*运输*）

设置运输。

*协程*read（*n* = *-1* ）

最多可读取*n*个字节。如果未提供*n*或设置为-1，则读取直到EOF并返回所有读取字节。

如果收到EOF并且内部缓冲区为空，则返回一个空bytes对象。

这个方法是一个协程。

coroutine readline（ ）

阅读一行，其中“行”是以字节结尾的一系列字节\n。

如果接收到EOF并且\n未找到，则该方法将返回部分读取字节。

如果收到EOF并且内部缓冲区为空，则返回一个空bytes对象。

这个方法是一个协程。

coroutine readexactly (*n*)

完全读取*n*个字节。 IncompleteReadError如果在读取*n*之前已达到流的末尾，则引发IncompleteReadError.partial该异常，异常的 属性包含部分读取字节。

这个方法是一个协程。

*协程*readuntil (*分隔符*= *b'\n'*)

从流中读取数据直到separator找到。

成功时，数据和分隔符将从内部缓冲区中删除（消耗）。返回的数据将在最后包含分隔符。

配置的流限制用于检查结果。限制设置可以返回的数据的最大长度，不包括分隔符。

如果发生EOF并且仍未找到完整的分隔符， IncompleteReadError则会引发异常，并且将重置内部缓冲区。该 IncompleteReadError.partial属性可能部分包含分隔符。

如果数据由于超限而无法读取， LimitOverrunError则会引发异常，并且数据将保留在内部缓冲区中，因此可以再次读取。

3.5.2版本的新功能。

at_eof ( )

True如果缓冲区为空feed_eof()并被调用，则返回。

18.5.5.3。StreamWriter的

*类*asyncio. StreamWriter (*传输*, *协议*, *阅读器*, *循环*)

包装运输。

这暴露了write() ， writelines() ， can_write_eof() ， write_eof() ，get_extra_info()和close()。它添加了 drain()返回一个可选项Future ，您可以等待流量控制。它还添加了Transport直接引用的传输属性。

这个类不是线程安全的。

transport

运输。

can_write_eof ( )

True 如果运输支持 write_eof() ， False 则返回， 否则返回。看WriteTransport.can_write_eof()。

close ( )

关闭运输：请参阅BaseTransport.close()。

coroutine drain ( )

让底层传输的写入缓冲区有机会被刷新。

预期的用途是写：

```
w.write(data)
yield from w.drain()
```

当传输缓冲区的大小达到高水位限制（协议暂停）时，阻塞，直到缓冲区的大小降低到低水位限制并且协议恢复。当没有什么可以等待时，收益率就会立即持续。

从中drain()产生可以让循环调度写入操作并刷新缓冲区。当可能大量的数据被写入传输时，它尤其应该被使用，并且协程在调用之间不会产生write()。

这个方法是一个协程。

get_extra_info(*名称*, *默认=无*)

返回可选的运输信息：参见 BaseTransport.get_extra_info()。

write(*数据*)

将一些*数据*字节写入传输：请参阅 WriteTransport.write()。

writelines(*数据*)

将一个列表（或任何可迭代的）数据字节写入transport：see WriteTransport.writelines()。

write_eof()

冲洗缓冲数据后关闭传输的写入结束：参见WriteTransport.write_eof()。

18.5.5.4。StreamReaderProtocol

class asyncio.StreamReaderProtocol(*stream_reader*, *client_connected_cb = None*, *loop = None*)

微不足道的帮手类以适应Protocol和 StreamReader。子类Protocol。

*stream_reader*是一个StreamReader实例， 当进行连接时，*client_connected_cb*是一个用（stream_reader，stream_writer）调用的可选函数，*loop*是要使用的事件循环实例。

（这是一个辅助类，而不是将StreamReader它自己作为 Protocol子类，因为它StreamReader具有其他潜在用途，并防止StreamReader意外调用该协议不适当方法的用户。）

18.5.5.5。IncompleteReadError

*异常*asyncio.IncompleteReadError

不完整的读取错误，子类EOFError。

expected

预期字节总数（ int ）。

partial
在到达流结束之前读取字节串（ bytes ）。

18.5.5.6。LimitOverrunError

*异常*asyncio.LimitOverrunError
在查找分隔符时达到了缓冲区限制。

consumed
要消耗的字节总数。

18.5.5.7。流示例

18.5.5.7.1。使用流的TCP回显客户端

使用该asyncio.open_connection()函数的TCP回显客户端：

```
import asyncio

@asyncio.coroutine
def tcp_echo_client(message, loop):
    reader, writer = yield from asyncio.open_connection('127.0.0.1', 8888,
                                                        loop=loop)

    print('Send: %r' % message)
    writer.write(message.encode())

    data = yield from reader.read(100)
    print('Received: %r' % data.decode())

    print('Close the socket')
    writer.close()

message = 'Hello World!'
loop = asyncio.get_event_loop()
loop.run_until_complete(tcp_echo_client(message, loop))
loop.close()
```

也可以看看： 的TCP回波客户端协议 的示例使用AbstractEventLoop.create_connection()方法。

18.5.5.7.2。使用流的TCP回显服务器

使用该asyncio.start_server()函数的TCP回显服务器：

```
import asyncio
```

```
@asyncio.coroutine
def handle_echo(reader, writer):
    data = yield from reader.read(100)
    message = data.decode()
    addr = writer.get_extra_info('peername')
    print("Received %r from %r" % (message, addr))

    print("Send: %r" % message)
    writer.write(data)
    yield from writer.drain()

    print("Close the client socket")
    writer.close()

loop = asyncio.get_event_loop()
coro = asyncio.start_server(handle_echo, '127.0.0.1', 8888, loop=loop)
server = loop.run_until_complete(coro)

# Serve requests until Ctrl+C is pressed
print('Serving on {}'.format(server.sockets[0].getsockname()))
try:
    loop.run_forever()
except KeyboardInterrupt:
    pass

# Close the server
server.close()
loop.run_until_complete(server.wait_closed())
loop.close()
```

也可以看看: 的TCP回声服务器协议 的示例使用AbstractEventLoop.create_server()方法。

18.5.5.7.3。获取HTTP标头

查询在命令行上传递的URL的HTTP标头的简单示例：

```
import asyncio
import urllib.parse
import sys

@asyncio.coroutine
def print_http_headers(url):
    url = urllib.parse.urlsplit(url)
    if url.scheme == 'https':
        connect = asyncio.open_connection(url.hostname, 443, ssl=True)
    else:
        connect = asyncio.open_connection(url.hostname, 80)
    reader, writer = yield from connect
    query = ('HEAD {path} HTTP/1.0\r\n'
             'Host: {hostname}\r\n'
             '\r\n').format(path=url.path or '/', hostname=url.hostname)
    writer.write(query.encode('latin-1'))
    while True:
        line = yield from reader.readline()
        if not line:
```

```
            break
        line = line.decode('latin1').rstrip()
        if line:
            print('HTTP header> %s' % line)

    # Ignore the body, close the socket
    writer.close()

url = sys.argv[1]
loop = asyncio.get_event_loop()
task = asyncio.ensure_future(print_http_headers(url))
loop.run_until_complete(task)
loop.close()
```

用法：

```
python example.py http://example.com/path/page.html
```

或使用HTTPS：

```
python example.py https://example.com/path/page.html
```

18.5.5.7.4。注册一个开放的套接字以等待数据使用流

协程等待套接字使用该open_connection()函数接收数据 ：

```
import asyncio
try:
    from socket import socketpair
except ImportError:
    from asyncio.windows_utils import socketpair

@asyncio.coroutine
def wait_for_data(loop):
    # Create a pair of connected sockets
    rsock, wsock = socketpair()

    # Register the open socket to wait for data
    reader, writer = yield from asyncio.open_connection(sock=rsock, loop=loop)

    # Simulate the reception of data from the network
    loop.call_soon(wsock.send, 'abc'.encode())

    # Wait for data
    data = yield from reader.read(100)

    # Got data, we are done: close the socket
    print("Received:", data.decode())
    writer.close()

    # Close the second socket
    wsock.close()

loop = asyncio.get_event_loop()
```

```
loop.run_until_complete(wait_for_data(loop))
loop.close()
```

也可以看看: 该寄存器使用协议示例等待数据的开放套接字使用由该 `AbstractEventLoop.create_connection()` 方法创建的低级协议 。

该手表读取事件文件描述符的例子使用了低级别的 `AbstractEventLoop.add_reader()` 方法来注册一个套接字的文件描述符。

18.5.6。子流程

源代码： Lib / asyncio / subprocess.py

18.5.6.1。Windows事件循环

在Windows上，默认的事件循环SelectorEventLoop不支持子进程。 ProactorEventLoop应该用来代替。在Windows上使用它的示例：

```
import asyncio, sys

if sys.platform == 'win32':
    loop = asyncio.ProactorEventLoop()
    asyncio.set_event_loop(loop)
```

也可以看看： 可用的事件循环和平台支持。

18.5.6.2。创建一个子进程：使用进程的高级

*协程*asyncio. create_subprocess_exec (**args*，*stdin = None*，*stdout = None*，*stderr = None*，*loop = None*，*limit = None*，*** kwds*)

创建一个子进程。

该*限制*参数设置传递给缓冲区限制 StreamReader 。查看AbstractEventLoop.subprocess_exec()其他参数。

返回一个Process实例。

这个函数是一个协程。

*协程*asyncio. create_subprocess_shell (*cmd*，*stdin = None*，*stdout = None*，*stderr = None*，*loop = None*，*limit = None*，*** kwds*)

运行shell命令*cmd*。

该*限制*参数设置传递给缓冲区限制 StreamReader 。查看AbstractEventLoop.subprocess_shell()其他参数。

返回一个Process实例。

应用程序有责任确保所有空白字符和元字符适当引用以避免shell注入 漏洞。该shlex.quote()函数可用于正确地转义将用于构造shell命令的字符串中的空白和shell元字符。

这个函数是一个协程。

使用AbstractEventLoop.connect_read_pipe()和 AbstractEventLoop.connect_write_pipe()方法来连接管道。

18.5.6.3。创建子进程：使用subprocess.Popen低级API

使用subprocess模块异步运行子进程。

*协程*AbstractEventLoop.subprocess_exec(*protocol_factory*, **args*, *stdin = subprocess.PIPE*, *stdout = subprocess.PIPE*, *stderr = subprocess.PIPE*, ***kwargs*)

从一个或多个字符串参数（字符串或编码为文件系统编码的字节字符串）创建一个子进程，其中第一个字符串指定要执行的程序，其余字符串指定程序的参数。（因此，sys.argv假设它是一个Python脚本，这些字符串参数一起构成了程序的 值。）这与subprocess.Popen使用shell = False调用的标准库类以及作为第一个参数传递的字符串列表类似; 然而，在哪里Popen需要一个参数是一个字符串列表，subprocess_exec()需要多个字符串参数。

该*protocol_factory*必须实例化的一个子 asyncio.SubprocessProtocol类。

其他参数：

- *stdin*：表示要使用连接到子进程的标准输入流的管道的文件类对象connect_write_pipe()，或者常量 subprocess.PIPE（缺省值）。默认情况下，将创建并连接一个新的管道。
- *stdout*：表示要使用连接到子进程标准输出流的管道的文件类对象connect_read_pipe()，或常量 subprocess.PIPE（缺省值）。默认情况下，将创建并连接一个新的管道。
- *stderr*：表示要用子连接到子进程的标准错误流的管道的类文件对象connect_read_pipe()，或者其中一个常量 subprocess.PIPE（缺省值）或subprocess.STDOUT。默认情况下，将创建并连接一个新的管道。当subprocess.STDOUT指定时，子进程的标准错误流将连接到与标准输出流相同的管道。
- subprocess.Popen 除了*bufsize*，*universal_newlines*和 *shell*之外，所有其他关键字参数都会传递给没有解释的参数，这些参数根本不应指定。

返回一对，其中*transport*是一个实例。(transport, protocol)BaseSubprocessTransport

这个方法是一个协程。

请参阅subprocess.Popen该类的构造函数以获取参数。

*协程*AbstractEventLoop.subprocess_shell(*protocol_factory*, *cmd*, ***, *stdin = subprocess.PIPE*, *stdout = subprocess.PIPE*, *stderr = subprocess.PIPE*, ***kwargs*)

使用平台的“shell”语法，从*cmd*创建子*进程*，该子*进程*是字符串或编码为文件系统编码的字节字符串。这与使用的标准库subprocess.Popen类相似 shell=True。

该*protocol_factory*必须实例化的一个子 asyncio.SubprocessProtocol类。

请参阅subprocess_exec()有关其余参数的更多详细信息。

返回一对，其中*transport*是一个实例。(transport, protocol)BaseSubprocessTransport

应用程序有责任确保所有空白字符和元字符适当引用以避免shell注入 漏洞。 该 shlex.quote() 函数可用于正确地转义将用于构造shell命令的字符串中的空白和shell元字符。

这个方法是一个协程。

也可以看看: 该AbstractEventLoop.connect_read_pipe()和 AbstractEventLoop.connect_write_pipe()方法。

18.5.6.4。常量

asyncio.subprocess.PIPE

可以用作and 和*stdin* ， *stdout*或*stderr*参数的特殊值可以打开标准流的管道。 create_subprocess_shell() create_subprocess_exec()

asyncio.subprocess.STDOUT

可以用作特殊值*标准错误*参数 create_subprocess_shell()和create_subprocess_exec() ，并表示标准误差应进入同一句柄作为标准输出。

asyncio.subprocess.DEVNULL

特殊值可用作and 和的*stdin* ， *stdout*或*stderr*参数，表示将使用特殊文件。 create_subprocess_shell() create_subprocess_exec() os.devnull

18.5.6.5。进程

*类*asyncio.subprocess.Process

由create_subprocess_exec()该 create_subprocess_shell()函数创建的子进程。

Process该类的API 被设计为与该类的API接近subprocess.Popen ，但是存在一些差异：

- 没有明确的poll()方法
- 该communicate()和 wait()方法不采取*超时*参数：使用wait_for()功能
- 该*universal_newlines*不支持的参数（只字节字符串支持）
- 在wait()所述的方法Process ，而类是异步 wait()的方法Popen 类作为繁忙的循环中实现。

这个类不是线程安全的。另请参阅 子流程和线程部分。

coroutine wait ( )

等待子进程终止。设置并返回returncode 属性。

这个方法是一个协程。

注意: 当使用stdout=PIPEor 时，这将会发生死锁， stderr=PIPE并且子进程会向管道生成足够的输出，从而阻止等待OS管道缓冲区接受更多数据。使用communicate()管道来避免这种情况时使用该 方法。

*协程*communicate (*输入=无*)

与流程进行交互：将数据发送到stdin。从stdout和stderr中读取数据，直到达到文件结尾。等待进程终止。可选的*输入*参数应该是要发送到子进程的None数据，或者如果没有数据应该发送给子进程。*输入*的类型必须是字节。

communicate()返回一个元组。(stdout_data, stderr_data)

如果一个BrokenPipeError或ConnectionResetError书写时引发异常*输入*到标准输入，异常被忽略。它在所有数据写入标准输入之前退出时发生。

请注意，如果要将数据发送到进程的stdin，则需要使用创建Process对象stdin=PIPE。同样，要获得除None结果元组以外的任何内容，您需要提供stdout=PIPE 和/或stderr=PIPE也可以。

这个方法是一个协程。

> **注意:** 读取的数据缓冲在内存中，所以如果数据量很大或无限，就不要使用这种方法。

*在版本3.4.2中更改：*该方法现在忽略BrokenPipeError并 ConnectionResetError。

send_signal (*信号*)

将信号*发送*到子进程。

> **注意：** 在 Windows 上，SIGTERM 是一个别名 terminate()。 CTRL_C_EVENT 并 CTRL_BREAK_EVENT 可以发送到以包含的 *creationflags* 参数 开始的进程 CREATE_NEW_PROCESS_GROUP。

terminate ( )

停止孩子。在Posix操作系统上，该方法发送signal.SIGTERM 给孩子。在Windows上调用Win32 API函数 TerminateProcess()来停止该子项。

kill ( )

杀死孩子。在Posix操作系统上，函数发送SIGKILL给孩子。在Windows上kill()是别名terminate()。

stdin

标准输入流（ StreamWriter ），None如果进程是使用创建的stdin=None。

stdout

标准输出流（ StreamReader ），None如果进程是使用stdout=None。

stderr

标准错误流（ StreamReader ），None如果进程是使用创建的stderr=None。

> **警告:** 使用communicate()方法而不是.stdin.write，.stdout.read或.stderr.read 避免因暂停读取或写入并阻止子进程流死锁。

pid

进程的标识符。

请注意，对于该create_subprocess_shell() 函数创建的进程，此属性是衍生shell的进程标识符。

returncode

进程退出时返回代码。一个None值表示进程尚未结束。

负值-N表示该孩子被信号终止 N（仅限Unix）。

18.5.6.6。子进程和线程

asyncio支持从不同线程运行子进程，但有一些限制：

- 事件循环必须在主线程中运行
- 在执行其他线程的子进程之前，必须在主线程中实例化子监视器。调用 get_child_watcher() 主线程中的函数来实例化子监视器。

本asyncio.subprocess.Process类不是线程安全的。

也可以看看： asyncio部分中的并发和多线程。

18.5.6.7。子的例子

18.5.6.7.1。使用传输和协议的子进程

用于获取子流程输出并等待子流程退出的子流程协议示例。子进程由该AbstractEventLoop.subprocess_exec()方法创建：

```
import asyncio
import sys

class DateProtocol(asyncio.SubprocessProtocol):
    def __init__(self, exit_future):
        self.exit_future = exit_future
        self.output = bytearray()

    def pipe_data_received(self, fd, data):
        self.output.extend(data)

    def process_exited(self):
        self.exit_future.set_result(True)

@asyncio.coroutine
def get_date(loop):
    code = 'import datetime; print(datetime.datetime.now())'
    exit_future = asyncio.Future(loop=loop)

    # Create the subprocess controlled by the protocol DateProtocol,
    # redirect the standard output into a pipe
```

```
    create = loop.subprocess_exec(lambda: DateProtocol(exit_future),
                                  sys.executable, '-c', code,
                                  stdin=None, stderr=None)
    transport, protocol = yield from create

    # Wait for the subprocess exit using the process_exited() method
    # of the protocol
    yield from exit_future

    # Close the stdout pipe
    transport.close()

    # Read the output which was collected by the pipe_data_received()
    # method of the protocol
    data = bytes(protocol.output)
    return data.decode('ascii').rstrip()

if sys.platform == "win32":
    loop = asyncio.ProactorEventLoop()
    asyncio.set_event_loop(loop)
else:
    loop = asyncio.get_event_loop()

date = loop.run_until_complete(get_date(loop))
print("Current date: %s" % date)
loop.close()
```

18.5.6.7.2。使用流的子流程

使用Process该类来控制子流程和StreamReader从标准输出读取的类的示例。子进程由create_subprocess_exec()函数创建：

```
import asyncio.subprocess
import sys

@asyncio.coroutine
def get_date():
    code = 'import datetime; print(datetime.datetime.now())'

    # Create the subprocess, redirect the standard output into a pipe
    create = asyncio.create_subprocess_exec(sys.executable, '-c', code,
                                            stdout=asyncio.subprocess.PIPE)
    proc = yield from create

    # Read one line of output
    data = yield from proc.stdout.readline()
    line = data.decode('ascii').rstrip()

    # Wait for the subprocess exit
    yield from proc.wait()
    return line

if sys.platform == "win32":
    loop = asyncio.ProactorEventLoop()
    asyncio.set_event_loop(loop)
```

```
else:
    loop = asyncio.get_event_loop()

date = loop.run_until_complete(get_date())
print("Current date: %s" % date)
loop.close()
```

18.5.7。同步原语

源代码： Lib / asyncio / locks.py

锁：

- Lock
- Event
- Condition

信号灯：

- Semaphore
- BoundedSemaphore

ASYNCIO锁API被设计成接近的类threading 模块（ Lock ， Event ， Condition ， Semaphore ， BoundedSemaphore ），但它没有*超时*参数。该 asyncio.wait_for() 功能可用于在超时后取消任务。

18.5.7.1。锁定

18.5.7.1.1。锁定

class asyncio. Lock （ *, *loop = None* ）

原始锁定对象。

原始锁定是同步原语，锁定时不属于特定的协同程序。原始锁定处于“锁定”或“解锁”两种状态之一。

它在解锁状态下创建。它有两个基本的方法，acquire() 和release()。状态解锁后，acquire（ ）会将状态更改为锁定并立即返回。当状态被锁定时，获取（ ）阻塞，直到在另一个协同程序中调用release（ ）将其更改为解锁，然后acquire（ ）调用将其重置为锁定并返回。release（ ）方法只能在锁定状态下调用; 它将状态更改为解锁并立即返回。如果试图释放未锁定的锁，RuntimeError则会提出。

当acquire（ ）等待状态转为解锁状态时，多个协程被阻塞，当release（ ）调用将状态重置为解锁状态时，只有一个协程会继续; 正在处理在acquire（ ）中被阻塞的第一个协程。

acquire()是一个协程，应该用。 yield from

锁也支持上下文管理协议。 应该用作上下文管理器表达式。 (yield from lock)

这个类不是线程安全的。

用法：

```
lock = Lock()
...
yield from lock
```

```
try:
    ...
finally:
    lock.release()
```

上下文管理器用法

```
lock = Lock()
...
with (yield from lock):
    ...
```

可以测试锁定对象的锁定状态：

```
if not lock.locked():
    yield from lock
else:
    # lock is acquired
    ...
```

locked ( )

True如果获取锁，则返回。

coroutine acquire ( )

获取一个锁。

此方法阻塞，直到解锁，然后将其设置为锁定并返回True。

这个方法是一个协程。

release ( )

释放一个锁。

锁定被锁定时，将其重置为解锁状态，然后返回。如果有其他协程被阻塞，等待锁被解锁，请准确地让其中一个进行。

在解锁的锁上调用时，RuntimeError会引发a。

没有返回值。

18.5.7.1.2。事件

class asyncio. Event (*, *loop = None*)

一个事件实现，异步等同于threading.Event。

类实现事件对象。一个事件管理一个标志，该标志可以通过该set()方法设置为true，并通过该方法重置为false clear() 。该wait()方法阻塞直到标志为真。国旗最初是假的。

这个类不是线程安全的。

clear ( )

将内部标志重置为false。随后，调用协程 wait() 将被阻塞，直到set()被调用以将内部标志再次设置为真。

is_set ( )

True当且仅当内部标志为真时才返回。

set ( )

将内部标志设置为true。所有等待它成为真实的协程都被唤醒。 wait() 一旦标志成立就调用的协程不会阻止。

coroutine wait ( )

阻塞直到内部标志为真。

如果内部标志在输入时为真，则True立即返回。否则，阻塞直到另一个协程调用set()将标志设置为true，然后返回True。

这个方法是一个协程。

18.5.7.1.3。条件

class asyncio. Condition (*lock = None* , *** , *loop = None*)

条件实现，异步等同于 threading.Condition。

这个类实现了条件变量对象。条件变量允许一个或多个协同程序等待，直到它们被另一个协程通知。

如果给出了*lock*参数而不是None，它必须是一个Lock 对象，并且它被用作底层锁。否则，将Lock创建一个新对象并将其用作基础锁。

这个类不是线程安全的。

coroutine acquire ( )

获取底层锁。

此方法阻塞，直到解锁，然后将其设置为锁定并返回True。

这个方法是一个协程。

notify (*n = 1*)

默认情况下，唤醒等待这种情况的一个协程，如果有的话。如果在调用此方法时调用的协程没有获得锁，RuntimeError则会引发a。

这种方法至多会唤醒*n个*等待条件变量的协程; 如果没有协同程序在等待，那么它是不可操作的。

> **注意:** 一个唤醒的协程实际上并没有从它的wait() 调用中返回，直到它可以重新获得锁。由于notify()不释放锁，因此其调用者应该。

locked ( )

True如果获取了底层锁，则返回。

notify_all ( )

唤醒所有等待这种情况的协同程序。这种方法的作用类似于 notify() 唤醒所有等待的协程，而不是一个。如果在调用此方法时调用的协程没有获得锁， RuntimeError则会引发a。

release ( )

释放底层的锁。

锁定被锁定时，将其重置为解锁状态，然后返回。如果有其他协程被阻塞，等待锁被解锁，请准确地让其中一个进行。

在解锁的锁上调用时，RuntimeError会引发a。

没有返回值。

coroutine wait ( )

等到通知。

如果在调用此方法时调用的协程没有获得锁，RuntimeError则会引发a。

此方法释放底层锁，然后阻塞，直到它被另一个协程中的相同条件变量唤醒notify()或notify_all()调用。一旦醒来，它重新获得锁定并返回True。

这个方法是一个协程。

*协程*wait_for (*谓词*)

等到谓词变为真。

谓词应该是可调用的，其结果将被解释为布尔值。最终的谓词值是返回值。

这个方法是一个协程。

18.5.7.2。信号量

18.5.7.2.1。信号量

class asyncio. Semaphore (*value = 1* , *, loop = None*)

信号量实现。

信号量管理一个内部计数器，每次acquire()调用都会减少内部计数器，并随每次 调用而递增release()。柜台不能低于零; 当acquire()发现它为零时，它会阻塞，等待其他协同程序调用release()。

信号量也支持上下文管理协议。

可选参数给出了内部计数器的初始值; 它默认为1。如果给定的值小于0 ， ValueError 则提高。

这个类不是线程安全的。

coroutine acquire ( )

获取信号量。

如果内部计数器在输入时大于零，则将其减1并True立即返回。如果进入时为零，则阻止，等待其他协程调用release()使其大于0，然后返回True。

这个方法是一个协程。

locked ( )

True如果不能立即获取信号量，则返回。

release ( )

释放一个信号量，将内部计数器递增1。当入口处为零并且另一个协程正在等待它再次变得大于零时，唤醒该协程。

18.5.7.2.2。BoundedSemaphore

class asyncio. BoundedSemaphore (*value = 1* , *** , *loop = None*)

有界的信号量实现。从...继承Semaphore。

这就提出了ValueError在release()它是否会增加超过初始值的值。

18.5.8。队列

源代码： Lib / asyncio / queues.py

队列：

- Queue
- PriorityQueue
- LifoQueue

ASYNCIO队列API被设计成接近的类queue 模块（ Queue ，PriorityQueue ， LifoQueue ），但它没有*超时*参数。该 asyncio.wait_for() 功能可用于在超时后取消任务。

18.5.8.1。队列

class asyncio. Queue （ *maxsize = 0* ，*，loop = None* ）

队列，用于协调生产者和消费者协同程序。

如果*maxsize*小于或等于零，则队列大小是无限的。如果它是一个大于整数的整数0，那么当队列达到*最大值*时会阻塞，直到一个项目被删除。 yield from put() get()

与标准库不同queue ，您可以可靠地知道该队列的大小qsize() ，因为您的单线程asyncio应用程序在调用qsize()和在队列上执行操作之间不会中断。

这个类不是线程安全的。

*在版本3.4.4中更改：*新建join()和task_done()方法。

empty （ ）

True如果队列为空False则返回，否则返回。

full （ ）

True如果maxsize队列中有项目，则返回。

> **注意：** 如果Queue初始化为maxsize=0（默认），那么 full()永远不会True。

coroutine get （ ）

从队列中移除并返回一个项目。如果队列为空，请等到项目可用。

这个方法是一个协程。

> **也可以看看：** 该empty()方法。

get_nowait （ ）

从队列中移除并返回一个项目。

如果一个项目立即可用，则返回一个项目，否则提出 QueueEmpty。

coroutine join ( )

阻塞，直到队列中的所有项目都被获取并处理。

每当将项目添加到队列中时，未完成任务的数量就会增加。无论何时消费者线程调用 task_done() 以指示该项目已被检索并且所有工作都已完成，计数就会减少 。当未完成任务的计数降至零时， join() 取消阻止。

这个方法是一个协程。

3.4.4版本的新功能。

*协程*put (*item*)

将一个项目放入队列中。如果队列已满，请在添加项目之前等待空闲插槽可用。

这个方法是一个协程。

也可以看看： 该full()方法。

put_nowait (*item*)

将物品放入队列中不受阻塞。

如果没有空闲插槽立即可用，请提高QueueFull。

qsize ( )

队列中的项目数量。

task_done ( )

表明以前排队的任务已完成。

由队列使用者使用。对于每个get()用于获取任务的对象，后续调用会task_done()告知队列该任务的处理已完成。

如果a join()当前处于阻塞状态，则在所有项目都处理完毕后（即task_done()接收到已put()进入队列的每个项目的呼叫），它将恢复。

ValueError如果被调用次数多于放入队列中的项目，则引发。

3.4.4版本的新功能。

maxsize

队列中允许的项目数量。

18.5.8.2。PriorityQueue中

*类*asyncio. PriorityQueue

一个子类Queue; 以优先顺序检索条目（最低优先）。

条目通常是以下形式的元组：（优先级编号，数据）。

18.5.8.3。LifoQueue

*类*asyncio.LifoQueue

它的一个子类Queue首先检索最近添加的条目。

18.5.8.3.1。例外

*异常*asyncio.QueueEmpty

当get_nowait()方法在Queue空的对象上调用 时引发异常。

*异常*asyncio.QueueFull

put_nowait()在Queue已满的对象上调用该方法 时引发异常。

18.5.9。用asyncio开发

异步编程不同于传统的“顺序”编程。本页列出了常见陷阱并解释如何避免它们。

18.5.9.1。asyncio的调试模式

执行asyncio已经写入性能。为了简化异步代码的开发，您可能希望启用*调试模式*。

要为应用程序启用所有调试检查：

- 通过设置环境变量来全局启用asyncio调试模式 PYTHONASYNCIODEBUG到1或通过呼叫AbstractEventLoop.set_debug()。
- 将 asyncio 记录器的日志级别设置为 logging.DEBUG 。例如，logging.basicConfig(level=logging.DEBUG)在启动时调用 。
- 配置warnings模块以显示ResourceWarning 警告。例如，使用-WdefaultPython 的命令行选项来显示它们。

示例调试检查：

- 定义了日志协程，但从未“放弃”
- call_soon()call_at()如果从错误的线程调用它们，方法会引发异常。
- 记录选择器的执行时间
- 日志回调需要执行超过100毫秒。该 AbstractEventLoop.slow_callback_duration属性是“慢”回调的最小持续时间（秒）。
- ResourceWarning当传输和事件循环没有明确关闭时，会发出警告。

也可以看看： 该AbstractEventLoop.set_debug()方法和asyncio记录器。

18.5.9.2。取消

在经典编程中取消任务并不常见。在异步编程中，不仅它是常见的，而且你必须准备好你的代码来处理它。

可以用他们的Future.cancel() 方法明确取消期货和任务。 wait_for()当超时发生时，该功能取消等待的任务。还有许多其他情况可以间接取消任务。

如果将来被取消，不要打电话set_result()或说明set_exception()方法Future：它会失败，例外。例如，写：

```
if not fut.cancelled():
    fut.set_result('done')
```

不要直接调用未来 set_result() 的 set_exception() 方法或调用未来的 方法 AbstractEventLoop.call_soon()：未来可以在其方法被调用之前取消。

如果你等待未来，如果未来取消以避免无用的操作，应该及早检查。例：

```
@coroutine
def slow_operation(fut):
    if fut.cancelled():
        return
    # ... slow computation ...
    yield from fut
    # ...
```

该shield()功能也可以用来忽略取消。

18.5.9.3。并发和多线程

一个事件循环在一个线程中运行，并执行同一线程中的所有回调和任务。当一个任务在事件循环中运行时，没有其他任务在同一个线程中运行。但是当任务使用时，任务被暂停，事件循环执行下一个任务。 yield from

要安排来自不同线程的回调， AbstractEventLoop.call_soon_threadsafe()应该使用该 方法。例：

```
loop.call_soon_threadsafe(callback, *args)
```

大多数asyncio对象不是线程安全的。如果您访问事件循环外部的对象，则只应该担心。例如，要取消未来，请勿直接调用其Future.cancel()方法，但是：

```
loop.call_soon_threadsafe(fut.cancel)
```

为了处理信号并执行子进程，事件循环必须在主线程中运行。

要从不同的线程调度协程对象， run_coroutine_threadsafe()应该使用该 功能。它返回一个concurrent.futures.Future来访问结果：

```
future = asyncio.run_coroutine_threadsafe(coro_func(), loop)
result = future.result(timeout)  # Wait for the result with a timeout
```

该AbstractEventLoop.run_in_executor()方法可与线程池执行程序一起使用，以在不同线程中执行回调以不阻塞事件循环的线程。

也可以看看: 该同步原语部分描述的方式来同步任务。

该子进程和线程部分列出ASYNCIO限制运行在不同的线程子进程。

18.5.9.4。正确处理阻塞函数

阻止函数不应该直接调用。例如，如果某个功能阻塞1秒，其他任务会延迟1秒，这会对反应性产生重要影响。

对于网络和子进程， asyncio模块提供协议等高级API 。

执行程序可用于在不同线程或甚至不同进程中运行任务，以不阻塞事件循环的线程。查看AbstractEventLoop.run_in_executor()方法。

也可以看看： 该延迟调用部分详细介绍了事件循环是如何处理的时间。

18.5.9.5。记录

该asyncio模块记录与信息logging的记录器模块'asyncio'。

该asyncio模块的默认日志级别是logging.INFO。对于那些不想从asyncio日志级别获得这种冗长的人可以改变。例如，要将级别更改为logging.WARNING：

```
logging.getLogger('asyncio').setLevel(logging.WARNING)
```

18.5.9.6。检测从未计划的协程对象

当一个协程函数被调用并且它的结果没有被传递给ensure_future()或传递给该AbstractEventLoop.create_task()方法时，协程对象的执行永远不会被调度，这可能是一个错误。 启用asyncio的调试模式 以记录警告来检测它。

错误示例：

```
import asyncio

@asyncio.coroutine
def test():
    print("never scheduled")

test()
```

在调试模式下输出：

```
Coroutine test() at test.py:3 was never yielded from
Coroutine object created at (most recent call last):
  File "test.py", line 7, in <module>
    test()
```

修复方法是使用协程对象调用ensure_future()函数或 AbstractEventLoop.create_task()方法。

也可以看看： 待处理任务被销毁。

18.5.9.7。检测不会消耗的异常

Python通常会调用sys.excepthook()未处理的异常。如果 Future.set_exception()被调用，但异常不会被消耗， sys.excepthook()则不会被调用。相反，当垃圾收集器删除将来时发出日志，并在引发异常的地方回溯。

未处理的异常示例：

```
import asyncio

@asyncio.coroutine
def bug():
    raise Exception("not consumed")

loop = asyncio.get_event_loop()
asyncio.ensure_future(bug())
loop.run_forever()
loop.close()
```

输出：

```
Task exception was never retrieved
future: <Task finished coro=<coro() done, defined at asyncio/coroutines.py:139> excep
Traceback (most recent call last):
  File "asyncio/tasks.py", line 237, in _step
    result = next(coro)
  File "asyncio/coroutines.py", line 141, in coro
    res = func(*args, **kw)
  File "test.py", line 5, in bug
    raise Exception("not consumed")
Exception: not consumed
```

启用asyncio的调试模式以获取创建任务的追踪。在调试模式下输出：

```
Task exception was never retrieved
future: <Task finished coro=<bug() done, defined at test.py:3> exception=Exception('no
source_traceback: Object created at (most recent call last):
  File "test.py", line 8, in <module>
    asyncio.ensure_future(bug())
Traceback (most recent call last):
  File "asyncio/tasks.py", line 237, in _step
    result = next(coro)
  File "asyncio/coroutines.py", line 79, in __next__
    return next(self.gen)
  File "asyncio/coroutines.py", line 141, in coro
    res = func(*args, **kw)
  File "test.py", line 5, in bug
    raise Exception("not consumed")
Exception: not consumed
```

有不同的选项来解决这个问题。第一个选项是将协程链连接到另一个协程，并使用经典尝试/除外：

```
@asyncio.coroutine
def handle_exception():
    try:
        yield from bug()
    except Exception:
        print("exception consumed")
```

```
loop = asyncio.get_event_loop()
asyncio.ensure_future(handle_exception())
loop.run_forever()
loop.close()
```

另一种选择是使用该AbstractEventLoop.run_until_complete() 功能：

```
task = asyncio.ensure_future(bug())
try:
    loop.run_until_complete(task)
except Exception:
    print("exception consumed")
```

也可以看看： 该Future.exception()方法。

18.5.9.8。链协程正确

当一个协程函数调用其他协程函数和任务时，它们应该被显式链接。否则，执行不保证是连续的。yield from

使用不同错误asyncio.sleep()模拟缓慢操作的示例：

```
import asyncio

@asyncio.coroutine
def create():
    yield from asyncio.sleep(3.0)
    print("(1) create file")

@asyncio.coroutine
def write():
    yield from asyncio.sleep(1.0)
    print("(2) write into file")

@asyncio.coroutine
def close():
    print("(3) close file")

@asyncio.coroutine
def test():
    asyncio.ensure_future(create())
    asyncio.ensure_future(write())
    asyncio.ensure_future(close())
    yield from asyncio.sleep(2.0)
    loop.stop()

loop = asyncio.get_event_loop()
asyncio.ensure_future(test())
loop.run_forever()
print("Pending tasks at exit: %s" % asyncio.Task.all_tasks(loop))
loop.close()
```

预期产出：

```
(1) create file
(2) write into file
(3) close file
Pending tasks at exit: set()
```

实际产出：

```
(3) close file
(2) write into file
Pending tasks at exit: {<Task pending create() at test.py:7 wait_for=<Future pending c
Task was destroyed but it is pending!
task: <Task pending create() done at test.py:5 wait_for=<Future pending cb=[Task._wake
```

之前停止循环create()完成后，close()之前一直叫write()，而协同程序功能进行了顺序叫：create()，write()，close()。

要修正这个例子，任务必须标记为：yield from

```
@asyncio.coroutine
def test():
    yield from asyncio.ensure_future(create())
    yield from asyncio.ensure_future(write())
    yield from asyncio.ensure_future(close())
    yield from asyncio.sleep(2.0)
    loop.stop()
```

或者没有asyncio.ensure_future()：

```
@asyncio.coroutine
def test():
    yield from create()
    yield from write()
    yield from close()
    yield from asyncio.sleep(2.0)
    loop.stop()
```

18.5.9.9。待处理任务被破坏

如果挂起的任务被销毁，其包装的协程的执行没有完成。这可能是一个错误，因此会记录一条警告。

日志示例：

```
Task was destroyed but it is pending!
task: <Task pending coro=<kill_me() done, defined at test.py:5> wait_for=<Future pendi
```

启用asyncio的调试模式以获取创建任务的追踪。登录调试模式示例：

```
Task was destroyed but it is pending!
source_traceback: Object created at (most recent call last):
  File "test.py", line 15, in <module>
    task = asyncio.ensure_future(coro, loop=loop)
task: <Task pending coro=<kill_me() done, defined at test.py:5> wait_for=<Future pend
```

也可以看看： 检测从未计划的协程对象。

18.5.9.10。关闭传输和事件循环

当不再需要传输时，请调用其`close()`方法释放资源。事件循环也必须显式关闭。

如果传输或事件循环没有被显式关闭，那么 `ResourceWarning`会在其析构函数中发出警告。默认情况下， `ResourceWarning`警告被忽略。asyncio的调试模式解释了如何显示它们。

18.6。 asyncore- 异步套接字处理程序

源代码： Lib / asyncore.py

*自从3.6版弃用：*请asyncio改用。

注意： 该模块仅用于向后兼容。对于我们推荐使用的新代码asyncio。

该模块为编写异步套接字服务客户端和服务器提供基本的基础结构。

只有两种方法可以让单个处理器上的程序“一次完成多件事”。多线程编程是最简单也是最流行的方法，但还有另一种非常不同的技术，它可以让您几乎具有多线程的所有优点，而无需实际使用多线程。如果你的程序在很大程度上是I / O限制的，那才真正实用。如果你的程序是处理器绑定的，那么抢先调度的线程可能就是你真正需要的。然而，网络服务器很少受处理器限制。

如果您的操作系统select()在其I / O库中支持系统调用（几乎全部都是这样），那么您可以使用它来一次处理多个通信通道; 在你的I / O发生在“背景”时做其他工作。尽管这种策略看起来很奇怪和复杂，特别是一开始，它比多线程编程在许多方面更易于理解和控制。该asyncore模块为您解决了许多棘手的问题，从而迅速构建复杂的高性能网络服务器和客户端。对于“会话式”应用程序和协议，伴随asynchat 模块是非常宝贵的。

这两个模块的基本思想是创建一个或多个网络 *通道*，类asyncore.dispatcher和 实例asynchat.async_chat。创建频道会将它们添加到全局地图中，loop()如果您没有提供自己的*地图*，则会 使用该全局*地图*。

一旦创建了初始频道，调用该loop()函数就会激活频道服务，直到最后一个频道（包括在异步服务期间添加到地图的任何频道）关闭为止。

asyncore.loop（[*timeout*[，*use_poll*[，*map*[，*count*]]]]）

输入一个轮询循环，在计数通过或所有打开的通道关闭后终止。所有参数都是可选的。的*计数*参数默认为None，导致在环路，只有当所有的频道都被关闭终止。的*超时*参数设置为适当的超时参数select()或poll() 呼叫，以秒为单位; 默认值是30秒。该*use_poll*参数，如果属实，表明poll()应优先使用select()（默认为False）。

该*地图*参数是一个字典，字典的项目是观看渠道。由于频道关闭，他们将从他们的地图中删除。如果*地图*被省略，则使用全局地图。信道（的实例 asyncore.dispatcher，asynchat.async_chat和亚类的化合物）可自由地在地图上混合。

*类*asyncore.dispatcher

的dispatcher类是一个低级别的插座对象的薄包装。为了使它更有用，它有几个用于事件处理的方法，这些方法是从异步循环中调用的。否则，它可以被当作普通的非阻塞套接字对象。

在特定时间或特定连接状态下触发低级别事件会告诉异步循环发生某些更高级别的事件。例如，如果我们要求一个套接字连接到另一个主机，我们知道当套接字第一次变为可写入时已经建立了连接（此时您知道您可以写入并期望成功）。隐含的高级事件是：

事件	描述
handle_connect()	由第一次读取或写入事件所暗示
handle_close()	隐含在没有可用数据的读取事件中
handle_accepted()	监听套接字上的读事件暗示了这一点

在异步处理期间，每个映射通道readable()和 writable()方法都用于确定是否应将通道套接字添加到读取和写入事件的select()ed或 poll()ed 通道列表中。

因此，这组通道事件大于基本套接字事件。可以在子类中重写的全套方法如下：

handle_read ( )

当异步循环检测到read()通道套接字上的调用将成功时调用。

handle_write ( )

当异步循环检测到可写的可写套接字时调用。通常这种方法将实现必要的性能缓冲。例如：

```
def handle_write(self):
    sent = self.send(self.buffer)
    self.buffer = self.buffer[sent:]
```

handle_expt ( )

当套接字连接有带外（OOB）数据时调用。这几乎不会发生，因为OOB得到了很少的支持，很少使用。

handle_connect ( )

当活动开启者的套接字实际建立连接时调用。例如，可以发送“欢迎”横幅，或者发起与远程端点的协议协商。

handle_close ( )

当插座关闭时调用。

handle_error ( )

当引发异常时调用，但不处理。默认版本打印浓缩回溯。

handle_accept ( )

当可以与发出connect() 本地端点呼叫的新远程端点建立连接时，在监听通道（被动开启者）上调用。在版本3.2中弃用; handle_accepted()改为使用 。

自3.2版以来已弃用。

handle_accepted (*sock* , *addr*)

当与已发出connect() 本地端点呼叫的新远程端点建立连接时，在监听通道（被动开启器）上调用。 *sock*是一个*新的*套接字对象，可用于在连接上发送和接收数据，*addr*是绑定到连接另一端套接字的地址。

3.2版本中的新功能

readable ( )

每次调用异步循环以确定是否应将通道的套接字添加到可能发生读取事件的列表中。默认方法简单地返回True，表示默认情况下，所有通道都会对读取事件感兴趣。

writable ( )

每次在异步循环周围调用以确定是否应将通道的套接字添加到发生写入事件的列表中。默认方法会简单地返回True，表明默认情况下，所有通道都会对写入事件感兴趣。

另外，每个通道都委托或扩展了许多套接字方法。其中大部分与他们的套接字伙伴几乎完全相同。

create_socket (*family = socket.AF_INET*, *type = socket.SOCK_STREAM*)

这与创建普通套接字相同，并将使用相同的选项进行创建。 socket有关创建套接字的信息，请参阅文档。

*版本3.3中更改：家族*和*类型*参数可以省略。

connect (*地址*)

与普通的套接字对象一样，*地址*是一个元组，主机要连接到的第一个元素，第二个元素是端口号。

send (*数据*)

将*数据*发送到套接字的远程端点。

recv (*buffer_size*)

从套接字的远程端点读取大多数*buffer_size*字节。一个空字节对象意味着该通道已经从另一端关闭。

请注意，即使 或已经报告套接字准备好读取，recv() 也可能会升高。BlockingIOErrorselect.select()select.poll()

listen (*积压*)

听取与插座的连接。的*积压*参数指定排队的最大连接数和应至少为1; 最大值取决于系统（通常为5）。

bind (*地址*)

将套接字绑定到*地址*。套接字不能被绑定。（*地址*格式取决于地址族 - 请参阅 socket文档以获取更多信息。）要将套接字标记为可重用（设置SO_REUSEADDR选项），请调用dispatcher对象的set_reuse_addr()方法。

accept ( )

接受连接。套接字必须绑定到地址并监听连接。返回值可以是None一对或一对 ，其中*conn*是可用于在连接上发送和接收数据的*新*套接字对象，*地址*是绑定到连接另一端套接字的地址。当返回这意味着连接并没有发生，在这种情况下，服务器应该只是忽略此事件，并保持侦听进一步的连接。(conn, address)None

close ( )

关闭插座。对套接字对象的所有未来操作都将失败。远程端点将不会收到更多数据（排队后的数据被刷新）。垃圾收集时，套接字会自动关闭。

*类*asyncore.**dispatcher_with_send**

一个dispatcher子类，增加了简单的缓冲输出能力，对于简单的客户非常有用。用于更复杂的用法 asynchat.async_chat。

*类*asyncore.**file_dispatcher**

file_dispatcher接受一个文件描述符或文件对象以及一个可选的map参数，并封装它以便与 poll() or loop()函数一起使用。如果提供了一个文件对象或任何一个 fileno()方法，该方法将被调用并传递给 file_wrapper构造函数。可用性：UNIX。

*类*asyncore.**file_wrapper**

file_wrapper接受一个整型文件描述符并调用os.dup()复制句柄，以便独立于file_wrapper关闭原始句柄。该类实现了足够的方法来模拟file_dispatcher该类使用的套接字。可用性：UNIX。

18.6.1。asyncore基本HTTP客户端示例

这是一个非常基本的HTTP客户端，它使用dispatcher该类来实现其套接字处理：

```
import asyncore

class HTTPClient(asyncore.dispatcher):

    def __init__(self, host, path):
        asyncore.dispatcher.__init__(self)
        self.create_socket()
        self.connect( (host, 80) )
        self.buffer = bytes('GET %s HTTP/1.0\r\nHost: %s\r\n\r\n' %
                            (path, host), 'ascii')

    def handle_connect(self):
        pass

    def handle_close(self):
        self.close()

    def handle_read(self):
        print(self.recv(8192))

    def writable(self):
        return (len(self.buffer) > 0)

    def handle_write(self):
        sent = self.send(self.buffer)
        self.buffer = self.buffer[sent:]


client = HTTPClient('www.python.org', '/')
asyncore.loop()
```

18.6.2。asyncore基本回显服务器示例

这是一个基本的回显服务器，它使用`dispatcher`该类来接受连接并将传入连接分派给处理程序：

```
import asyncore

class EchoHandler(asyncore.dispatcher_with_send):

    def handle_read(self):
        data = self.recv(8192)
        if data:
            self.send(data)

class EchoServer(asyncore.dispatcher):

    def __init__(self, host, port):
        asyncore.dispatcher.__init__(self)
        self.create_socket()
        self.set_reuse_addr()
        self.bind((host, port))
        self.listen(5)

    def handle_accepted(self, sock, addr):
        print('Incoming connection from %s' % repr(addr))
        handler = EchoHandler(sock)

server = EchoServer('localhost', 8080)
asyncore.loop()
```

18.7。asynchat- 异步套接字命令/响应处理程序

源代码： Lib / asynchat.py

*自从3.6版弃用：*请asyncio改用。

注意： 该模块仅用于向后兼容。对于我们推荐使用的新代码asyncio。

该模块建立在asyncore基础架构之上，简化了异步客户端和服务器，并且更容易处理元素被任意字符串终止或者长度可变的协议。 asynchat定义async_chat你的子类的抽象类，提供collect_incoming_data()和 found_terminator()方法的实现。它使用相同的异步环路asyncore，和两种类型的信道的，asyncore.dispatcher 并且asynchat.async_chat，可以自由地在信道映射进行混合。通常，asyncore.dispatcher服务器通道asynchat.async_chat会在接收传入连接请求时生成新的 通道对象。

*类*asynchat.async_chat

这个类是一个抽象的子类asyncore.dispatcher。为了使实际使用中你必须继承的代码async_chat，提供有意义collect_incoming_data()和found_terminator()方法。asyncore.dispatcher可以使用这些方法，尽管在消息/响应上下文中并不都是有意义的。

像asyncore.dispatcher，async_chat定义了一组事件，这些事件是在select()调用之后通过分析套接字条件生成的。一旦轮询循环开始，async_chat对象的方法就由事件处理框架调用，而程序员不需要采取任何行动。

可以修改两个类属性以提高性能，甚至可以节省内存。

ac_in_buffer_size

异步输入缓冲区大小（默认4096）。

ac_out_buffer_size

异步输出缓冲区大小（默认4096）。

与之不同的是asyncore.dispatcher，async_chat您可以定义*生产者*的FIFO队列。制作人只需要一种方法，它应该返回要在频道上传输的数据。生产者通过使其方法返回空字节对象来指示耗尽（*即*它不包含更多数据）。此时，对象将生产者从队列中移除，并开始使用下一个生产者（如果有的话）。当生产者队列为空时，该 方法什么都不做。您使用通道对象的方法来描述如何识别来自远程端点的传入传输的末端或重要断点。more()more()async_chathandle_write()set_terminator()

要建立一个正常运作的async_chat子类你的输入法collect_incoming_data()和found_terminator()必须处理通道异步接收数据。方法如下所述。

async_chat.close_when_done（）

推动None生产者队列。当这个制作者从队列中弹出时，它会导致频道关闭。

async_chat.collect_incoming_data(*数据*)

调用*数据*保存任意数量的接收数据。必须重写的默认方法会引发NotImplementedError异常。

async_chat.discard_buffers()

在紧急情况下，此方法将丢弃保存在输入和/或输出缓冲区以及生产者队列中的所有数据。

async_chat.found_terminator()

当传入的数据流符合由设置的终止条件时调用set_terminator()。必须重写的默认方法会引发NotImplementedError异常。缓冲的输入数据应该通过实例属性可用。

async_chat.get_terminator()

返回该通道的当前终止符。

async_chat.push(*数据*)

将数据推入通道的队列以确保传输。这就是让通道将数据写入网络所需要做的一切，尽管例如可以使用自己的生产者以更复杂的方式实现加密和分块。

async_chat.push_with_producer(*生产者*)

获取一个生产者对象并将其添加到与该通道关联的生产者队列中。当所有当前推送的生产者都用尽时，通道将通过调用其生产者的more()方法并将数据发送到远程端点来消费该生产者的数据。

async_chat.set_terminator(*term*)

设置要在通道上识别的终止条件。term可以是三种类型的值中的任何一种，对应于处理输入协议数据的三种不同方式。

术语	描述
串	将found_terminator()在输入流中找到字符串时调用
整数	found_terminator()当收到指定数量的字符时将会打电话
None	该频道将继续永久收集数据

请注意，终止符后面的任何数据将在found_terminator()调用后供通道读取。

18.7.1。asynchat示例

以下部分示例显示了如何使用HTTP请求读取async_chat。Web服务器可能会http_request_handler为每个传入的客户端连接创建一个对象。请注意，最初将通道终止符设置为与HTTP标头末尾的空行匹配，并且标志表示正在读取标头。

一旦头部被读取，如果请求的类型是POST（表示输入流中还有其他数据），则Content-Length:头部用于设置数字终结符以从通道读取适量的数据。

handle_request()在设置了通道终止符后，所有相关输入都被编组后，调用该方法None以确保忽略Web客户端发送的任何无关数据。

```
import asynchat

class http_request_handler(asynchat.async_chat):

    def __init__(self, sock, addr, sessions, log):
        asynchat.async_chat.__init__(self, sock=sock)
        self.addr = addr
        self.sessions = sessions
        self.ibuffer = []
        self.obuffer = b""
        self.set_terminator(b"\r\n\r\n")
        self.reading_headers = True
        self.handling = False
        self.cgi_data = None
        self.log = log

    def collect_incoming_data(self, data):
        """Buffer the data"""
        self.ibuffer.append(data)

    def found_terminator(self):
        if self.reading_headers:
            self.reading_headers = False
            self.parse_headers(b"".join(self.ibuffer))
            self.ibuffer = []
            if self.op.upper() == b"POST":
                clen = self.headers.getheader("content-length")
                self.set_terminator(int(clen))
            else:
                self.handling = True
                self.set_terminator(None)
                self.handle_request()
        elif not self.handling:
            self.set_terminator(None)  # browsers sometimes over-send
            self.cgi_data = parse(self.headers, b"".join(self.ibuffer))
            self.handling = True
            self.ibuffer = []
            self.handle_request()
```

18.8。signal- 为异步事件设置处理程序

该模块提供了在Python中使用信号处理程序的机制。

18.8.1。一般规则

该signal.signal()功能允许在收到信号时定义自定义处理程序。安装了少量的缺省处理程序：SIGPIPE被忽略（因此管道和套接字上的写入错误可以作为普通的Python异常报告）并被SIGINT转换为KeyboardInterrupt异常。

一旦设置了特定信号的处理程序，直到它被显式重置（Python模拟BSD样式接口，而不管底层实现如何）时，它仍保持安装状态，除了底层实现之后的处理程序以外 SIGCHLD。

18.8.1.1。执行Python信号处理程序

Python信号处理程序不会在低级（C）信号处理程序中执行。相反，低级信号处理程序会设置一个标志，告诉 虚拟机稍后执行相应的Python信号处理程序（例如在下一个字节码指令处）。这有以下后果：

- 捕捉类似的SIGFPE或 SIGSEGV由C代码中的无效操作引起的同步错误是没有意义的。Python将从信号处理程序返回到C代码，这可能会再次提高相同的信号，导致Python显然挂起。从Python 3.3开始，您可以使用该faulthandler模块报告同步错误。
- 无论接收到什么信号，纯粹以C语言实现的长时间运行计算（例如大量文本的正则表达式匹配）都可以连续运行一段任意时间。计算结束时将调用Python信号处理程序。

18.8.1.2。信号和线程

即使在另一个线程中收到信号，Python信号处理程序也总是在主Python线程中执行。这意味着信号不能用作线程间通信的手段。您可以改用threading模块中的同步原语。

此外，只有主线程才可以设置新的信号处理程序。

18.8.2。模块内容

*在3.5版本中改变了：*信号（SIG *），处理器（ SIG_DFL ， SIG_IGN ）和sigmask（ ， SIG_BLOCK ， SIG_UNBLOCK ）SIG_SETMASK下面列出相关的常数，变成 enums。 getsignal() ， pthread_sigmask() ， sigpending()和 sigwait()函数返回人类可读的 enums。

signal模块中定义的变量是：

signal.SIG_DFL
    这是两种标准信号处理选项之一; 它将简单地执行信号的默认功能。例如，在大多数系统中，缺省操作SIGQUIT是转储核心并退出，而默认操作SIGCHLD是简单地忽略它。

signal.SIG_IGN

这是另一个标准的信号处理程序，它将简单地忽略给定的信号。

SIG*
所有的信号编号都是象征性地定义的。例如，挂断信号被定义为signal.SIGHUP;变量名称与C程序中使用的名称相同，如下所示<signal.h>。'signal()'的Unix手册页列出了现有的信号（在某些系统上，这是*信号（2）*，在其他系统 上是*信号（7）*）。请注意，并非所有系统都定义了同一组信号名称;只有由系统定义的名称才由该模块定义。

signal.CTRL_C_EVENT
与Ctrl+C击键事件相对应的信号。这个信号只能用于os.kill()。

可用性：Windows。

3.2版本中的新功能

signal.CTRL_BREAK_EVENT
与Ctrl+Break击键事件相对应的信号。这个信号只能用于os.kill()。

可用性：Windows。

3.2版本中的新功能

signal.NSIG
比最高信号数量多一个。

signal.ITIMER_REAL
实时减少间隔计时器，并SIGALRM在到期时交付。

signal.ITIMER_VIRTUAL
仅在进程正在执行时递减间隔计时器，并在到期时递送SIGVTALRM。

signal.ITIMER_PROF
当进程执行时以及代表进程执行系统时，减少间隔计时器。加上ITIMER_VIRTUAL，这个计时器通常用于分析应用程序在用户和内核空间中所花费的时间。SIGPROF在到期时交付。

signal.SIG_BLOCK
指示该信号被阻止的*how*参数的可能值pthread_sigmask()。

3.3版本的新功能

signal.SIG_UNBLOCK
指示要解除信号的*how*参数的可能值pthread_sigmask()。

3.3版本的新功能

signal.SIG_SETMASK
指示信号掩码将被替换的*how*参数的可能值pthread_sigmask()。

3.3版本的新功能

该signal模块定义了一个例外：

*异常*signal.ItimerError

被提出来表示底层setitimer()或 getitimer()实现中的错误。如果传递无效的间隔计时器或负时间，则会出现此错误setitimer()。这个错误是一个子类型OSError。

*版本3.3中的新增功能：*此错误曾经是一个子类型IOError，现在是它的别名OSError。

该signal模块定义了以下功能：

signal.alarm(*时间*)

如果*时间*为非零，该函数请求一个SIGALRM信号被发送到处理*时间*秒。任何先前预定的警报都将被取消（任何时候只能安排一个警报）。然后返回的值是任何先前设置的警报传递之前的秒数。如果*时间*为零，则不安排警报，并且任何预定警报都将被取消。如果返回值为零，则当前不安排警报。（请参阅Unix手册页*警报（2）*。）可用性：Unix。

signal.getsignal(*signalnum*)

返回信号*signalnum*的当前信号处理程序。返回的值可能是一个可调用的Python对象，或者是一个特殊值 signal.SIG_IGN，signal.SIG_DFL或者None。这里 signal.SIG_IGN意味着信号先前被忽略， signal.SIG_DFL意味着处理信号的默认方式以前在使用中，并且None意味着以前的信号处理程序不是从Python安装的。

signal.pause()

使进程进入休眠状态，直到收到信号; 然后会调用适当的处理程序。什么都不返回 不在Windows上。（请参阅Unix手册页 *信号（2）*。）

又见sigwait()，sigwaitinfo()，sigtimedwait()和 sigpending()。

signal.pthread_kill(*thread_id*，*signalnum*)

将信号*signalnum*发送到线程*thread_id*，与调用方在同一进程中的另一个线程。目标线程可以执行任何代码（Python或不）。但是，如果目标线程正在执行Python解释器，Python信号处理程序将由主线程执行。因此，向特定Python线程发送信号的唯一要点是强制正在运行的系统调用失败InterruptedError。

使用threading.get_ident()或对象的ident 属性threading.Thread为*thread_id*获取合适的值。

如果*signalnum*为0，则不发送信号，但仍然执行错误检查; 这可以用来检查目标线程是否仍在运行。

可用性：Unix（有关更多信息，请参见手册页*pthread_kill（3）*）。

另见os.kill()。

3.3版本的新功能

signal.pthread_sigmask(*how*，*mask*)

获取和/或更改调用线程的信号掩码。信号掩码是当前被呼叫者阻止传送的一组信号。将旧信号掩码作为一组信号返回。

调用的行为是依赖于价值*如何*，如下所示。

- SIG_BLOCK：阻塞信号的集合是当前集合和*掩码*参数的联合。
- SIG_UNBLOCK：*掩码*中的信号将从当前的一组阻塞信号中移除。允许尝试解除未被阻止的信号。
- SIG_SETMASK：阻塞信号组被设置为*掩码*参数。

*掩码*是一组信号编号（例如{ signal.SIGINT， signal.SIGTERM}）。使用一个完整的屏蔽在内的所有信号。range(1, signal.NSIG)

例如，读取调用线程的信号掩码。signal.pthread_sigmask(signal.SIG_BLOCK, [])

可用性：Unix。有关更多信息，请参见手册页*sigprocmask（3）*和*pthread_sigmask（3）*。

另见pause()，sigpending()并且sigwait()。

3.3版本的新功能

signal.**setitimer**(*which*, *seconds*[, *interval*])

给定的间隔计时器集（之一signal.ITIMER_REAL，signal.ITIMER_VIRTUAL或signal.ITIMER_PROF通过指定），*其*之后触发*秒*（浮子被接受时，从不同的alarm()），并且每一个后*间隔*秒。通过指定的时间间隔定时器，*其*可以通过设置秒至零被清除。

当间隔计时器触发时，将向该进程发送一个信号。发送的信号取决于正在使用的定时器; signal.ITIMER_REAL将交付SIGALRM，signal.ITIMER_VIRTUAL发送SIGVTALRM并signal.ITIMER_PROF交付SIGPROF。

旧值作为元组返回：（delay，interval）。

试图通过一个无效的间隔计时器将导致一个ItimerError。可用性：Unix。

signal.**getitimer**(*哪*)

返回由指定的给定的时间间隔定时器的当前值，*其*。可用性：Unix。

signal.**set_wakeup_fd**(*fd*)

将唤醒文件描述符设置为*fd*。当接收到一个信号时，信号编号作为一个字节写入fd。库可以使用此功能来唤醒投票或选择呼叫，从而使信号得到完全处理。

返回旧的唤醒fd（如果文件描述符唤醒未启用，则返回-1）。如果*fd*为-1，则禁用文件描述符唤醒。如果不是-1，那么*fd*必须是非阻塞的。它是由图书馆中删除任何字节*FD*调用大选投票前或再次选择。

例如用于解码信号编号列表。struct.unpack('%uB' % len(data), data)

当启用线程时，只能从主线程调用此函数; 试图从其他线程调用它会导致引发ValueError异常。

*版本3.5中更改：*在Windows上，该函数现在还支持套接字句柄。

signal.**siginterrupt**(*signalnum*, *标志*)

更改系统调用重启行为：如果*标志*为False，系统调用将在信号*signalnum*中断时重新启动，否则系统调用将被中断。什么都不返回 可用性：Unix（有关更多信息，请参见手册页*siginterrupt（3）*）。

请注意，安装信号处理程序signal()将通过以给定信号siginterrupt()的真实*标志*值隐式调用来重置重启动行为为可中断。

signal.**signal**（*signalnum*，*handler*）

将信号*signalnum*的处理程序设置为函数*处理程序*。*处理程序*可以是一个可调用的Python对象，它带有两个参数（见下文），或者一个特殊值signal.SIG_IGN或signal.SIG_DFL。之前的信号处理程序将被返回（参见getsignal()上面的描述）。（请参阅Unix手册页*信号（2）*。）

当启用线程时，只能从主线程调用此函数; 试图从其他线程调用它会导致引发ValueError异常。

该*处理程序*有两个参数调用：信号编号和当前堆栈帧（None或框架对象;有关框架对象的描述，请参阅类型层次结构中的描述或查看inspect模块中的属性描述）。

在 Windows 中，signal()只能叫SIGABRT，SIGFPE，SIGILL，SIGINT，SIGSEGV，SIGTERM，或SIGBREAK。A ValueError会在其他情况下被提出。请注意，并非所有系统都定义了同一组信号名称; AttributeError如果信号名称没有定义为SIG*模块级别常量，则会引发一次。

signal.**sigpending**（）

检查待传递给调用线程的一组信号（即阻塞时产生的信号）。返回待处理信号的集合。

可用性：Unix（有关更多信息，请参阅手册页*sigpending（2）*）。

另见pause()，pthread_sigmask()并且sigwait()。

3.3版本的新功能

signal.**sigwait**（*sigset*）

挂起调用线程的执行，直到传递信号集*sigset中*指定的信号之一。该功能接受信号（将其从待处理的信号列表中移除）并返回信号编号。

可用性：Unix（有关更多信息，请参见*manwait（3）*手册页）。

又见pause()，pthread_sigmask()，sigpending()，sigwaitinfo()和sigtimedwait()。

3.3版本的新功能

signal.**sigwaitinfo**（*sigset*）

挂起调用线程的执行，直到传递信号集*sigset中*指定的信号之一。该功能接受信号并将其从待处理的信号列表中移除。如果*sigset中的*一个信号已经等待调用线程，该函数将立即返回关于该信号的信息。信号处理程序不针对传送的信号进行调用。InterruptedError如果该函数被一个不在*sigset中*的信号中断，该函数会产生一个。

返回值是表示包含在该数据对象 siginfo_t的结构，即：si_signo，si_code， si_errno，si_pid，si_uid，si_status， si_band。

可用性：Unix（有关更多信息，请参见手册页*sigwaitinfo（2）*）。

另见pause()，sigwait()并且sigtimedwait()。

3.3版本的新功能

*在版本3.5中更改：*如果函数被*sigset*中的信号中断并且信号处理程序不引发异常，现在重试该函数（请参阅**PEP 475**为理由）。

signal.**sigtimedwait**(*sigset*, *timeout*)

喜欢sigwaitinfo()，但需要额外的*超时*参数指定超时。如果指定*超时*0，则执行轮询。None如果发生超时，则返回。

可用性：Unix（有关更多信息，请参见手册页*sigtimedwait（2）*）。

另见pause()，sigwait()并且sigwaitinfo()。

3.3版本的新功能

*在版本3.5中进行了更改：*如果被*sigset*中的信号中断，并且信号处理程序不引发异常，现在重新计算该函数的重计算*超时值*（请参阅**PEP 475**为理由）。

18.8.3。示例

这是一个简单的示例程序。它使用该alarm()函数来限制等待打开文件的时间; 如果该文件适用于可能未开启的串行设备，这通常会导致os.open()无限期挂起，这很有用。解决方法是在打开文件之前设置5秒钟的警报; 如果操作时间太长，则会发送警报信号，并且处理程序引发异常。

```
import signal, os

def handler(signum, frame):
    print('Signal handler called with signal', signum)
    raise OSError("Couldn't open device!")

# Set the signal handler and a 5-second alarm
signal.signal(signal.SIGALRM, handler)
signal.alarm(5)

# This open() may hang indefinitely
fd = os.open('/dev/ttyS0', os.O_RDWR)

signal.alarm(0)          # Disable the alarm
```

18.9。 mmap- 内存映射文件支持

内存映射文件对象的行为既像bytearray和类似 文件的对象。您可以在bytearray预期的大多数地方使用mmap对象; 例如，您可以使用该re 模块搜索内存映射文件。您也可以通过执行来更改单个字节，或者通过分配给片来更改子序列：。您还可以读取和写入从当前文件位置开始的数据，并通过文件将数据写入不同的位置。obj[index] = 97obj[i1:i2] = b'...'seek()

内存映射文件由mmap构造函数创建，在Unix和Windows上不同。在任何情况下，您都必须为打开的文件提供文件描述符以进行更新。如果您希望映射现有的Python文件对象，请使用其fileno()方法获取*fileno*参数的正确值 。否则，您可以使用该os.open()函数直接返回文件描述符（该文件在完成时仍需要关闭）来打开该文件 。

注意： 如果你想创建一个可写入的缓冲文件的内存映射，你应该flush()首先使用该文件。这对确保对缓冲区的本地修改实际上可用于映射是必需的。

对于构造函数的Unix和Windows版本，可以将*访问*指定为可选的关键字参数。*访问*接受以下三个值之一：ACCESS_READ，ACCESS_WRITE或ACCESS_COPY 分别指定只读，直写或写时复制内存。*访问*可以在Unix和Windows上使用。如果未指定*访问权限*，则Windows mmap将返回一个直写映射。所有三种访问类型的初始内存值都来自指定的文件。分配给 ACCESS_READ内存映射引发TypeError异常。分配给ACCESS_WRITE内存映射影响内存和底层文件。赋值给一个ACCESS_COPY内存映射会影响内存，但不会更新基础文件。

要映射匿名内存，应该将-1作为文件no与长度一起传递。

class mmap.**mmap**（*fileno*，*length*，*tagname = None*，*access = ACCESS_DEFAULT* [，*offset*]）

（Windows版本）映射文件句柄*fileno*指定的文件的*长度*字节，并创建一个mmap对象。如果*长度*大于文件的当前大小，则文件将扩展为包含*长度*字节。如果*长度*是，映射的最大长度是文件的当前大小，但是如果文件为空，则Windows引发异常（您无法在Windows上创建空映射）。0

标记名（如果指定而不是None）是一个为映射提供标记名称的字符串。Windows允许您针对同一个文件有许多不同的映射。如果您指定现有标签的名称，则会打开该标签，否则会创建该名称的新标签。如果省略此参数或者None创建没有名称的映射。避免使用标签参数将有助于在Unix和Windows之间保持代码的可移植性。

*偏移量*可以被指定为非负整数偏移量。mmap引用将相对于文件开头的偏移量。*偏移量*默认为0. *偏移量*必须是ALLOCATIONGRANULARITY的倍数。

class mmap.**mmap**（*fileno*，*length*，*flags = MAP_SHARED*，*prot = PROT_WRITE* | *PROT_READ*，*access = ACCESS_DEFAULT* [，*offset*]）

（Unix版本）映射文件描述符*fileno*指定的文件的*长度*字节，并返回一个mmap对象。如果*长度*是，则映射的最大长度将是调用时文件的当前大小 。0mmap

*标志*指定映射的性质。MAP_PRIVATE创建一个私人写时复制映射，所以对mmap对象内容的更改对于此进程是私有的，并MAP_SHARED创建一个映射，该映射与映射该文件的相同区域

的所有其他进程共享。默认值是MAP_SHARED。

如果指定*prot*，则提供期望的内存保护; 两个最有用的值是PROT_READ和PROT_WRITE，到指定的页面可以被读取或写入。 *prot*默认为 。PROT_READ | PROT_WRITE

*访问*可以被指定代替*标志*和*prot*作为可选的关键字参数。指定*标志*，*prot*和 *访问*都是错误的。有关如何使用此参数的信息，请参阅上面的*访问*说明。

*偏移量*可以被指定为非负整数偏移量。mmap引用将相对于文件开头的偏移量。*偏移量* 默认为0. *偏移量*必须是PAGESIZE或ALLOCATIONGRANULARITY的倍数。

为了确保创建的内存映射的有效性，由描述符*fileno*指定的*文件*在内部自动与Mac OS X和OpenVMS上的物理后备存储同步。

这个例子展示了一个简单的使用方法mmap：

```
import mmap

# write a simple example file
with open("hello.txt", "wb") as f:
    f.write(b"Hello Python!\n")

with open("hello.txt", "r+b") as f:
    # memory-map the file, size 0 means whole file
    mm = mmap.mmap(f.fileno(), 0)
    # read content via standard file methods
    print(mm.readline())  # prints b"Hello Python!\n"
    # read content via slice notation
    print(mm[:5])  # prints b"Hello"
    # update content using slice notation;
    # note that new content must have same size
    mm[6:] = b" world!\n"
    # ... and read again using standard file methods
    mm.seek(0)
    print(mm.readline())  # prints b"Hello  world!\n"
    # close the map
    mm.close()
```

mmap也可以在with 声明中用作上下文管理器：

```
import mmap

with mmap.mmap(-1, 13) as mm:
    mm.write(b"Hello world!")
```

*3.2版本中的新功能：*支持上下文管理器。

下一个示例演示如何创建匿名映射并在父进程和子进程之间交换数据：

```
import mmap
import os

mm = mmap.mmap(-1, 13)
mm.write(b"Hello world!")
```

```
pid = os.fork()

if pid == 0:  # In a child process
    mm.seek(0)
    print(mm.readline())

    mm.close()
```

内存映射文件对象支持以下方法：

close ( )

关闭mmap。随后调用该对象的其他方法将导致引发ValueError异常。这不会关闭打开的文件。

closed

True 如果文件关闭。

3.2版本中的新功能

find (*sub* [, *start* [, *end*]])

返回找到*子*序列*子*对象的最低索引，使得*子*包含在[*start* , *end*] 范围内。可选参数*开始*和*结束*被解释为切片符号。返回-1失败。

*在版本3.5中更改：*现在可以接受可写入类似字节的对象。

flush ([*offset* [, *size*]])

将对文件的内存中副本所做的更改刷新到磁盘。如果不使用此调用，则无法保证在对象销毁前写回更改。如果指定了*偏移量*和*大小*，则只有给定范围的字节的更改才会刷新到磁盘; 否则，映射的整个范围被刷新。

（Windows版本）返回的非零值表示成功; 零表示失败。

（Unix版本）返回零值表示成功。呼叫失败时引发异常。

move (*dest* , *src* , *count*)

将从偏移量*src*开始的*计数*字节复制到目标索引 *dest*。如果mmap是使用创建的，则调用移动会引发异常。 ACCESS_READTypeError

read ([*n*])

从当前文件位置开始返回一个bytes包含*n*个字节的文件。如果参数被省略None或为负，则将当前文件位置的所有字节返回到映射的结尾。文件位置更新为指向返回的字节之后。

*版本3.3中更改：*参数可以省略或None。

read_byte ( )

以整数形式返回当前文件位置的一个字节，并将文件位置前进1。

readline ( )

返回单行，从当前文件位置开始，直到下一个换行符。

resize (*newsize*)

如果有的话，调整地图和底层文件的大小。如果mmap是使用ACCESS_READor 创建的ACCESS_COPY，则调整地图的大小会引发TypeError异常。

rfind (*sub* [, *start* [, *end*]])

返回找到*子*序列*子*对象的最高索引，使得*子*包含在[*start* , *end*] 范围内。可选参数*开始*和*结束*被解释为切片符号。返回-1失败。

*在版本3.5中更改：*现在可以接受可写入类似字节的对象。

seek (*pos* [, *whence*])

设置文件的当前位置。 *此*参数是可选的，默认为os.SEEK_SET或0（绝对文件定位）；其他值是os.SEEK_CUR或1（相对于当前位置寻找）和os.SEEK_END或2（相对于文件结尾寻找）。

size ( )

返回文件的长度，该长度可能大于内存映射区域的大小。

tell ( )

返回文件指针的当前位置。

write (*字节*)

将字节以*字节*为单位写入文件指针当前位置的内存中，并返回写入的字节数（从不小于 len(bytes)，因为如果写入失败，ValueError则会引发a）。文件位置被更新为指向写入的字节之后。如果mmap是使用创建的ACCESS_READ，那么写入它将引发TypeError异常。

*在版本3.5中更改：*现在可以接受可写入类似字节的对象。

*在版本3.6中更改：*现在返回写入的字节数。

write_byte (*字节*)

将整数*字节*写入文件指针当前位置的内存中; 该文件位置被提前1。如果mmap是使用创建的ACCESS_READ，那么写入它将引发TypeError异常。

19.互联网数据处理

本章介绍支持Internet上常用处理数据格式的模块。

19.1。 email- 电子邮件和MIME处理软件包

源代码： Lib / email / __ init__.py

该email软件包是用于管理电子邮件消息的库。它*没有*专门设计用于发送电子邮件到SMTP（**RFC 2821**），NNTP或其他服务器; 这些是诸如smtplib和的模块的功能 nntplib。该email软件包尽可能符合RFC标准，并提供支持**RFC 5233**和**RFC 6532**以及与MIME相关的RFC**RFC 2045**，**RFC 2046**，**RFC 2047**，**RFC 2183**，和**RFC 2231**。

电子邮件包的整体结构可以分为三个主要组件，以及控制其他组件行为的第四个组件。

该软件包的核心组件是一个表示电子邮件的“对象模型”。应用程序主要通过在message子模块中定义的对象模型接口与应用程序进行交互。应用程序可以使用此API来询问有关现有电子邮件的问题，构建新电子邮件，或添加或删除自己使用相同对象模型界面的电子邮件子组件。也就是说，根据电子邮件及其MIME子组件的性质，电子邮件对象模型是所有提供EmailMessage API的对象的树结构。

该软件包的另外两个主要组件是parser和generator。解析器获取电子邮件消息的序列化版本（字节流）并将其转换为EmailMessage对象树 。生成器 EmailMessage将其转换回串行化的字节流。（解析器和生成器也处理文本字符流，但不鼓励使用这种用法，因为以这种或那种方式结束消息是无效的。

控制组件是policy模块。每一个EmailMessage ，每一个 ，每generator一个 parser都有一个policy控制其行为的关联对象。通常情况下，应用程序只需要在EmailMessage创建时指定策略，可以通过直接实例化EmailMessage 创建新电子邮件，也可以通过使用a解析输入流parser。但是当使用a序列化消息时，可以更改策略generator。例如，这允许从磁盘解析通用电子邮件消息，但在将其发送到电子邮件服务器时使用标准SMTP设置对其进行序列化。

电子邮件软件包会尽最大努力从应用程序中隐藏各种治理RFC的细节。从概念上讲，应用程序应该能够将电子邮件视为unicode文本和二进制附件的结构化树，而不必担心在序列化时如何表示这些内容。然而，在实践中，通常至少需要注意一些关于MIME消息及其结构的规则，特别是MIME“内容类型”的名称和性质以及它们如何识别多部分文档。大多数情况下，只有更复杂的应用程序才需要这种知识，即使如此，它也只应该是所讨论的高级结构，而不是这些结构如何表示的细节。由于MIME内容类型在现代互联网软件（不仅仅是电子邮件）中被广泛使用，

以下部分描述了该email软件包的功能。我们从message对象模型开始，它是应用程序将使用的主要接口，并使用parser和generator组件进行跟踪 。然后我们介绍 policy控制，完成对图书馆主要组成部分的处理。

接下来的三节将介绍该软件包可能引发的异常情况以及parser可能检测到的缺陷（不符合RFC）。然后，我们覆盖headerregistry和 contentmanager子组件，这对于做头和有效载荷的更详细的操作分别提供工具。这两个组件都包含与消费和生成非重要消息相关的功能，并且还记录了它们的可扩展性API，这些对于高级应用程序来说都很有用。

以下是使用前几节中介绍的API基本部分的一组示例。

上述代表电子邮件包的现代（适用于Unicode的）API。其余的部分，从Message 课程开始，涵盖了compat32更直接处理电子邮件如何表示的细节的传统API。该 compat32API并*没有*隐藏应用程序的RFC的细节，但对于那些需要在该级别运行的应用程序，它们可以成为有用的工具。本文档对于compat32为了向后兼容的原因仍在使用API的应用程序也是相关的。

*改变在3.6版本：*文档重组和改写，以促进新 EmailMessage/ EmailPolicy API。

该内容email包的文档：

传统API：

也可以看看:

模 smtplib
SMTP（简单邮件传输协议）客户端

模 poplib
POP（邮局协议）客户端

模 imaplib
IMAP（Internet消息访问协议）客户端

模 nntplib
NNTP（网络新闻传输协议）客户端

模 mailbox
使用各种标准格式创建，读取和管理磁盘上的邮件集合的工具。

模 smtpd
SMTP服务器框架（主要用于测试）

19.1.1。email.message：代表电子邮件

源代码： Lib / email / message.py

3.6版新增功能： [1]

email包中的中心类是EmailMessage 从email.message模块导入的类。它是email对象模型的基类。 EmailMessage提供了设置和查询标题字段，访问消息体以及创建或修改结构化消息的核心功能。

电子邮件消息由*标题*和*有效负载*（也称为*内容*）组成。标题是**RFC 5322**或**RFC 6532**样式字段名称和值，其中字段名称和值由冒号分隔。冒号不是字段名称或字段值的一部分。有效载荷可以是简单的文本消息或二进制对象，或者是具有它们自己的一组头标和它们自己的有效载荷的子消息的结构化序列。后一种有效载荷由具有MIME类型的消息（例如 *multipart / *或 *message / rfc822 ）表示*。

由一个EmailMessage对象提供的概念模型是一个有序的字典的头部，加上一个*有效载荷*，代表着这个模型 消息的 **RFC 5322**正文，可能是子EmailMessage 对象列表。除了用于访问标题名称和值的常规字典方法之外，还有一些方法用于访问来自标题的专用信息（例如MIME内容类型），用于在有效负载上操作，用于生成消息的序列化版本，以及递归地遍历对象树。

类似EmailMessage字典的接口由标头名称索引，标头名称必须是ASCII值。字典的值是带有一些额外方法的字符串。标题以保存案例的形式存储并返回，但字段名称不区分大小写。与真正的字典不同，有一个按键的顺序，并且可以有重复的按键。提供了其他方法来处理具有重复键的标题。

对于MIME容器文档（如*multipart / *和*message / rfc822* 消息对象），*有效*内容是字符串或字节对象（对于简单消息对象或EmailMessage对象列表）。

class email.message.EmailMessage（*policy = default* ）

如果指定*策略*，则使用它指定的规则来更新和序列化消息的表示。如果*政策*未设置，使用的 default政策，这是继电子邮件RFC的规则，除了行结束（而不是RFC规定\r\n，它使用Python标准\n行尾）。欲了解更多信息，请参阅 policy文档。

as_string（*unixfrom = False*，*maxheaderlen = None*，*policy = None* ）

将整个消息以字符串的形式展平。当可选的 *unixfrom*为true时，信封头将包含在返回的字符串中。 *unixfrom*默认为False。为了向后兼容基Message类*maxheaderlen*是可以接受的，但默认为None，这意味着默认情况下，行长度由 max_line_length策略控制。该*策略*的参数可被用于覆盖从消息实例中获得的默认策略。这可以用来控制该方法生成的一些格式，因为指定的*策略*将被传递给该方法Generator。

EmailMessage 如果需要填充默认值以完成对字符串的转换（例如，可能会生成或修改MIME边界），则消除该消息可能会触发更改。

请注意，此方法作为一种便利提供，可能不是在应用程序中序列化消息的最有用方式，尤其是在处理多个消息时。查看 email.generator.Generator用于序列化消息的更

灵活的API。还注意，该方法仅限于生产序列化为“7位干净”当消息 utf8是False，这是缺省值。

在版本3.6中进行了更改： 未指定*maxheaderlen*时的默认行为从默认值更改为0，从而默认为策略中的*max_line_length*值。

__str__（）

等同于*as_string（policy = self.policy.clone（utf8 = True））*。允许 str(msg)以可读格式生成包含序列化消息的字符串。

*在版本3.4中更改：*该方法已更改为使用utf8=True，从而生成一个类似 **RFC 6531**的消息表示，而不是直接别名as_string()。

as_bytes（*unixfrom = False，policy = None*）

将整个消息作为字节对象返回展平。当可选的 *unixfrom*为true时，信封头将包含在返回的字符串中。 *unixfrom*默认为False。该*策略*的参数可被用于覆盖从消息实例中获得的默认策略。这可以用来控制该方法生成的一些格式，因为指定的*策略*将被传递给该方法 BytesGenerator。

EmailMessage 如果需要填充默认值以完成对字符串的转换（例如，可能会生成或修改MIME边界），则消除该消息可能会触发更改。

请注意，此方法作为一种便利提供，可能不是在应用程序中序列化消息的最有用方式，尤其是在处理多个消息时。查看 email.generator.BytesGenerator用于序列化消息的更灵活的API。

__bytes__（）

相当于as_bytes()。允许bytes(msg)生成包含序列化消息的字节对象。

is_multipart（）

返回True如果消息的有效载荷子的列表EmailMessage对象，否则返回False。当is_multipart()返回False时，有效载荷应该是一个字符串对象（其可能被一个CTE编码的二进制有效负载）。请注意， is_multipart() 返回 True 并不一定意味着“msg.get_content_maintype（）=='multipart'”将返回True。例如，当is_multipart返回类型True时 将返回。EmailMessagemessage/rfc822

set_unixfrom（*unixfrom*）

将消息的信封头设置为*unixfrom*，它应该是一个字符串。（请参阅mboxMessage此标题的简要说明。）

get_unixfrom（）

返回消息的信封头。默认为None如果信封头从未设置。

以下方法实现类似映射的接口以访问消息的标题。请注意，这些方法和普通映射（即字典）界面之间存在一些语义差异。例如，在字典中没有重复的键，但这里可能有重复的消息标题。另外，在字典中，对返回的键没有保证的顺序keys()，但是在EmailMessage 对象中，标题总是按照它们出现在原始消息中的顺序返回，或者在稍后将它们添加到消息中。任何删除并重新添加的标题总是附加到标题列表的末尾。

这些语义差异是有意的，并且在最常见的用例中偏向于方便。

请注意，在所有情况下，消息中出现的任何信封标题都不包含在映射界面中。

__len__()
返回标题的总数，包括重复项。

__contains__(*名字*)
如果消息对象具有名为*name*的字段，则返回true 。匹配完成时不考虑大小写，*名称*不包含尾部冒号。用于in操作员。例如：

```
if 'message-id' in myMessage:
   print('Message-ID:', myMessage['message-id'])
```

__getitem__(*名字*)
返回指定标题字段的值。 *名称*不包含冒号字段分隔符。如果标题丢失，None则返回; a KeyError永远不会升起。

请注意，如果命名字段在消息头中多次出现，那么将返回那些字段值中的哪些值是未定义的。使用该get_all()方法获取所有名为*name*的现存头的值。

使用标准（非compat32）策略，返回值是一个子类的实例email.headerregistry.BaseHeader。

__setitem__(*name*, *val*)
将字段添加到带有字段*名称*和值*val*的消息中。该字段被添加到消息的现有标题的末尾。

请注意，这并*不能*覆盖或删除任何现有的头名称相同。如果要确保新标题是带有字段名称*名称*的消息中唯一存在的标题，请先删除该字段，例如：

```
del msg['subject']
msg['subject'] = 'Python roolz!'
```

如果policy定义某些标题是唯一的（如标准策略所做的那样），那么ValueError当尝试为已经存在的标题分配值时，此方法可能会引发一次。这种行为是为了一致性的缘故，但不要依赖它，因为我们可能会选择使这种分配在将来自动删除现有标题。

__delitem__(*名字*)
从消息的标题中删除所有出现的具有名称*名称*的字段。如果名称字段不存在于标题中，则不会引发异常。

keys()
返回所有消息标题字段名称的列表。

values()
返回所有消息字段值的列表。

items()

返回包含所有消息的字段标题和值的2元组列表。

get（*名称*，*failobj =无*）

返回指定标题字段的值。这与`__getitem__()`除了如果命名标题丢失（*failobj*缺省为）时返回可选*failobj*相同 。None

以下是一些额外有用的标题相关方法：

get_all（*名称*，*failobj =无*）

返回名为*name*的字段的所有值的列表。如果消息中没有这样的标题，则返回*failobj*（默认为 None）。

add_header（*_name*，*_value*，*** _params*）

扩展标题设置。`__setitem__()` 除了可以提供额外的头部参数作为关键字参数之外，此方法类似。 *_name*是要添加的标题字段，而*_value*是标题的 *主要*值。

对于关键字参数字典*_params*中的每个项目，都将该关键字作为参数名称，并将下划线转换为破折号（因为破折号在Python标识符中是非法的）。通常，该参数将被添加，key="value"除非该值为None，在这种情况下，只有该键将被添加。

如果该值包含非ASCII字符，则可以通过将值指定为格式中的三元组来明确控制字符集和语言，其中是指定用于对值进行编码的字符集的字符串，通常可以设置为或空字符串（请参阅(CHARSET, LANGUAGE, VALUE)CHARSETLANGUAGENone**RFC 2231**用于其他可能性），并且VALUE是包含非ASCII码点的字符串值。如果三元组没有传递，并且该值包含非ASCII字符，则会自动编入英寸**RFC 2231**格式使用CHARSET的utf-8和LANGUAGE的None。

这里是一个例子：

```
msg.add_header('Content-Disposition', 'attachment', filename='bud.gif')
```

这将添加一个看起来像的标题

```
Content-Disposition: attachment; filename="bud.gif"
```

使用非ASCII字符的扩展接口的示例：

```
msg.add_header('Content-Disposition', 'attachment',
               filename=('iso-8859-1', '', 'Fußballer.ppt'))
```

replace_header（*_name*，*_value*）

替换标题。替换消息中找到的与*_name*匹配的第一个标头，保留原始标头的标头顺序和字段名称大小写。如果找不到匹配的标题，请提出一个 KeyError。

get_content_type（）

返回消息的内容类型，强制为*maintype* / *subtype*形式的 *小写*。如果 消息中没有*Content-Type*标题，则返回由返回的值 `get_default_type()`。如果*Content-Type*标头无效，则返回text/plain。

（根据 **RFC 2045**，消息始终有一个默认类型， `get_content_type()` 将始终返回一个值。 **RFC 2045**将消息的默认类型定义为 *text / plain*，除非它出现在 *multipart / digest* 容器中，在这种情况下，它将是 *message / rfc822*。如果 *Content-Type*头的类型规范无效，**RFC 2045**规定默认类型为 *text / plain*。 ）

`get_content_maintype ( )`

返回消息的主要内容类型。这是 返回的字符串的*maintype*部分`get_content_type()`。

`get_content_subtype ( )`

返回消息的子内容类型。这是 返回的字符串的*子类型*部分`get_content_type()`。

`get_default_type ( )`

返回默认的内容类型。除了作为*多部分/摘要*容器的子*部分的*消息之外， 大多数消息具有默认的*文本/纯文本*内容类型。这样的子部分有一个默认的内容类型的*消息/ rfc822*。

`set_default_type ( ctype )`

设置默认的内容类型。 *ctype*应该是 *text / plain*或*message / rfc822*，尽管这不是强制执行的。缺省内容类型不存储在 *Content-Type* 标头中，因此只有 在消息中`get_content_type`不存在*Content-Type*标头时才会影响方法的返回值。

`set_param ( param , value , header ='Content-Type' , requote = True , charset = None , language ='' , replace = False )`

在*Content-Type*标题中设置一个参数。如果该参数已经存在于标题中，则将其值替换为*值*。当*header*是`Content-Type`（缺省值）并且消息中还不存在标题时，添加它，将其值设置为 *text / plain*，并追加新的参数值。可选 *标题*指定*Content-Type*的替代标题。

如果该值包含非ASCII字符，则可以使用可选的*字符集*和*语言* 参数明确指定*字符集*和*语言*。可选*语言*指定**RFC 2231**语言，默认为空字符串。这两个*字符集*和*语言*应弦。默认是使用`utf8` *的字符集*，并`None`为*语言*。

如果*替换*为`False`（默认），则标题将移动到标题列表的末尾。如果*更换*就是`True`，头部将在地方进行更新。

不推荐使用带对象的*requote*参数`EmailMessage`。

请注意，标题的现有参数值可以通过`params`标题值的属性进行访问（例如，`msg['Content-Type'].params['charset']`。

在版本3.4中更改：`replace`添加了关键字。

`del_param ( param , header ='content-type' , requote = True )`

从*Content-Type* 标题中完全删除给定的参数。标题将在没有参数或其值的情况下被重写。可选*标题*指定*Content-Type*的替代方案 。

不推荐使用带对象的*requote*参数`EmailMessage`。

`get_filename ( failobj =无 )`

返回消息filename的*Content-Disposition*头的参数 值。如果标题没有filename参数，则此方法回退到查找*Content-Type*标题name上的参数。如果两者都未找到，或者标题丢失，则返回*failobj*。返回的字符串将始终不加引号 。 email.utils.unquote()

get_boundary (*failobj =无*)

返回消息boundary的*Content-Type*头的参数 值，或者如果头丢失或没有参数，则*返回* *failobj*boundary。返回的字符串将始终不加引号email.utils.unquote()。

set_boundary (*边界*)

将*Content-Type*标头的boundary参数设置为 *边界*。 将在必要时总是引用*边界*。如果消息对象没有*Content-Type*头部，则引发A.set_boundary() HeaderParseError

请注意，使用此方法与删除旧的*Content-Type*标题并添加一个具有新边界的新标题有细微差别 add_header() ，因为这会set_boundary()保留标题列表中*Content-Type*标题的顺序。

get_content_charset (*failobj =无*)

返回*Content-Type*头部的charset参数，强制为小写。如果没有*Content-Type*标头，或者该标头没有参数，则返回*failobj*。 charset

get_charsets (*failobj =无*)

返回包含消息中字符集名称的列表。如果消息是*多部分*，那么列表将包含有效载荷中每个子部分的一个元素，否则它将是长度为1的列表。

列表中的每个项目都是一个字符串，它是表示子部分charset的*Content-Type*标题中的参数值 。如果子部分没有*Content-Type* 标头，没有charset参数，或者不是*文本*主MIME类型，则返回列表中的该项目将为*failobj*。

is_attachment ( )

返回True ，如果有一个*内容处置*头和它的（不区分大小写）值attachment ，False否则。

在版本3.4.2中更改： is_attachment现在是一种方法，而不是一个属性，与之一致 is_multipart()。

get_content_disposition ( )

返回消息的*Content-Disposition*头的小写值（不带参数）， 如果它有一个或者None。此方法的可能值为*内联*，*附件*或None 后续消息**RFC 2183**。

3.5版本中的新功能。

以下方法涉及询问和操纵消息的内容（有效载荷）。

walk ( )

该walk()方法是一个通用的生成器，可用于以深度优先遍历顺序遍历消息对象树的所有部分和子部分。您通常会将其walk()用作for循环中的迭代器; 每次迭代都会返回下一个子部分。

以下是一个打印多部分消息结构的每个部分的MIME类型的示例：

```
>>> for part in msg.walk():
...     print(part.get_content_type())
multipart/report
text/plain
message/delivery-status
text/plain
text/plain
message/rfc822
text/plain
```

walk迭代is_multipart()返回的任何部分的子部分 True，即使 可能会返回。我们可以在我们的例子中看到这一点，通过使用调试帮助函数：msg.get_content_maintype() == 'multipart'False_structure

```
>>> for part in msg.walk():
...     print(part.get_content_maintype() == 'multipart',
...           part.is_multipart())
True True
False False
False True
False False
False False
False True
False False
>>> _structure(msg)
multipart/report
    text/plain
    message/delivery-status
        text/plain
        text/plain
    message/rfc822
        text/plain
```

这里的message部分不是multiparts，但它们确实包含子部分。is_multipart()返回True并walk下降到子部分。

get_body（*preferencelist = ('related'，'html'，'plain')*）

返回最适合成为消息“正文”的MIME部分。

*首选项列表*必须是来自集合related， html和和的字符串序列plain，并指示返回部分的内容类型的首选顺序。

开始寻找与该get_body方法被调用的对象的候选匹配 。

如果related未包含在*首选项列表中*，则考虑（子）部分与首选项匹配时作为候选项遇到的任何相关联的根部分（或根部分的子部分）。

遇到a时multipart/related，请检查start参数，如果找到具有匹配的*Content-ID*的零件，请仅在查找候选匹配项时考虑它。否则，只考虑第一个（默认根）部分multipart/related。

如果某个零件具有*Content-Disposition*标题，则只有标题值为的情况下才考虑该零件的候选匹配inline。

如果没有任何候选人匹配偏好列表中的任何 *偏好设置*，则返回None。

注：（1）对于大多数应用程序的唯一*preferencelist*组合，真正意义是('plain',)，和默认。（2）因为匹配从调用的对象开始，调用a 将返回对象本身，除非 *preferencelist* 具有非默认值。（3）未指定*Content-Type*或其 *Content-Type*标头无效的消息（或消息部分）将视为它们是类型的，这可能偶尔会导致返回意外的结果。('html', 'plain') ('related', 'html', 'plain') get_body get_body multipart/related text/plain get_body

iter_attachments( )

返回消息中不是候选"正文"部分的所有直接子部分的迭代器。即，跳过的每一个的第一次出现text/plain，text/html，multipart/related，或 multipart/alternative（除非它们被明确地标记为经由附件*内容处置：附件*），并返回所有剩余部分。当直接应用于a时 multipart/related，将迭代器返回到除根部分以外的所有相关部分（即：start参数指向的部分，或者第一部分，如果没有start参数或start 参数不匹配 *Content-ID* 的任何部分）。当直接应用于一个multipart/alternative或一个非multipart，返回一个空的迭代器。

iter_parts( )

将消息的所有直接子部分返回一个迭代器，对于非消息来说这将是空的multipart。（另见 walk()）

get_content(**args*, *content_manager = None*, *** kw*)

调用*content_manager*的get_content()方法，将self作为消息对象传递，并将任何其他参数或关键字作为附加参数传递。如果 未指定*content_manager*，请使用当前指定的内容。content_manager policy

set_content(**args*, *content_manager = None*, *** kw*)

调用*content_manager*的set_content()方法，将self作为消息对象传递，并将任何其他参数或关键字作为附加参数传递。如果 未指定*content_manager*，请使用当前指定的内容。content_manager policy

make_related(*boundary = None*)

将非multipart消息转换为multipart/related消息，将任何现有的*Content-* Header和有效载荷转换为（新的）第一部分multipart。如果*边界*指定，使用它作为在所述多部分的边界的字符串，否则离开边界在需要时被自动创建（例如，当该消息是序列化）。

make_alternative(*boundary = None*)

将non multipart或a multipart/related转换为a multipart/alternative，将任何现有的*Content-* header和有效载荷转换为（新的）第一部分multipart。如果 *边界*指定，使用它作为在所述多部分的边界的字符串，否则离开边界在需要时被自动创建（例如，当该消息是序列化）。

make_mixed(*boundary = None*)

将非a multipart，a multipart/related 或 a multipart-alternative 转换为a multipart/mixed，将任何现有的 *Content-* header和有效载荷转换为（新的）第一部

分 multipart。如果*边界*指定，使用它作为在所述多部分的边界的字符串，否则离开边界在需要时被自动创建（例如，当该消息是序列化）。

add_related (**args*, *content_manager = None*, ***kw*)

如果消息是a multipart/related，则创建一个新的消息对象，将所有参数传递给它的 set_content() 方法，并将 attach() 其传递给 multipart。如果消息是非消息 multipart，则调用make_related()，然后按上述方式继续。如果该消息是其他类型的 multipart，则提出一个 TypeError。如果未指定*content_manager*，请使用当前 content_manager指定的内容policy。如果添加的部分没有*Content-Disposition*标头，则添加一个值inline。

add_alternative (**args*, *content_manager = None*, ***kw*)

如果消息是a multipart/alternative，则创建一个新的消息对象，将所有参数传递给它的set_content()方法，并将 attach()其传递给multipart。如果该消息是一个非multipart或multipart/related，呼叫 make_alternative()然后如上述进行。如果该消息是其他类型的 multipart，则提出一个 TypeError。如果未指定*content_manager*，请使用当前content_manager指定的内容policy。

add_attachment (**args*, *content_manager = None*, ***kw*)

如果消息是a multipart/mixed，则创建一个新的消息对象，将所有参数传递给它的 set_content() 方法，并将 attach() 其传递给 multipart。如果该消息是非 multipart，，multipart/related或 multipart/alternative，呼叫make_mixed()，然后按上述进行。如果未指定*content_manager*，请使用当前content_manager 指定的内容 policy。如果添加的部分没有 *Content-Disposition* 标头，则添加一个值 attachment。此方法可以用于显式附件（*Content-Disposition：附件*和inline附件（*Content-Disposition：inline*）），方法是将适当的选项传递给content_manager。

clear ( )

删除有效载荷和所有标题。

clear_content ( )

删除有效载荷和所有Content-标题，将所有其他标题保持原样并按原始顺序排列。

EmailMessage 对象具有以下实例属性：

preamble

MIME文档的格式允许标题后面的空白行和第一个多段边界字符串之间的文本。通常，这个文本在MIME感知的邮件阅读器中是不可见的，因为它不属于标准的MIME装甲。但是，查看邮件的原始文本时，或者在非MIME的阅读器中查看邮件时，可以看到该文本。

在*序言*属性包含MIME文件这导致额外的装甲文本。当在Parser标题后面但在第一个边界字符串之前发现一些文本时，它会将该文本分配给该消息的*前导*属性。当 Generator 写出MIME消息的纯文本表示，并且它发现消息具有*前导*属性时，它将在头和第一个边界之间的区域中写入该文本。请参阅email.parser并 email.generator了解详情。

请注意，如果消息对象没有前导码，则*前导码*属性将为None。

epilogue

的*尾声*属性作用方式与相同的*前导码*属性，不同之处在于它包含的最后一个边界和消息的结束之间出现的文本。与之一样preamble，如果没有结语文本，这个属性将会是None。

defects

该*缺陷*属性包含的所有解析此消息时发现的问题的列表。请参阅有关email.errors可能的解析缺陷的详细说明。

class email.message.MIMEPart (*policy = default*)

这个类表示MIME消息的一个子部分。EmailMessage除了在被调用时不添加*MIME-Version*标头以外，它与之相同 set_content() ，因为子部分不需要它们自己的*MIME-Version*标头。

脚注

[1] 最初作为临时模块添加到3.4中。旧邮件类的文档已移至 email.message.Message：使用compat32 API表示电子邮件。

19.1.2。email.parser：解析电子邮件

源代码： Lib / email / parser.py

消息对象结构可以通过以下两种方式之一来创建：它们可以通过创建EmailMessage 对象，使用字典接口添加标头，以及使用set_content()和相关方法添加有效载荷来创建，也可以通过解析序列化的电子邮件消息表示。

该email软件包提供了一个可理解大多数电子邮件文档结构（包括MIME文档）的标准分析器。您可以将解析器传递给字节，字符串或文件对象，解析器将返回EmailMessage对象结构的根 实例。对于简单的非MIME消息，这个根对象的负载可能是一个包含消息文本的字符串。为MIME消息，根对象将返回True其is_multipart() 方法，以及子部分可以通过操纵有效载荷的方法，如被访问get_body()， iter_parts()和 walk()。

实际上有两个可用的解析器接口，即Parser API和增量FeedParserAPI。的Parser，如果你有邮件的整个文本在内存中，或如果整个消息住在文件系统上的文件API是最有用的。 FeedParser当您从一个可能阻塞等待更多输入的流中读取消息时（例如从套接字读取电子邮件消息）更合适。该 FeedParser消息可以逐渐消耗和分析消息，并且只在关闭解析器时才返回根对象。

请注意，解析器可以以有限的方式进行扩展，当然，您可以从头开始完全实现自己的解析器。连接email包的捆绑解析器和 EmailMessage类的所有逻辑都包含在policy 类中，因此定制解析器可以通过实现适当policy 方法的自定义版本以任何方式创建消息对象树。

19.1.2.1。FeedParser API

的BytesFeedParser，从导入的email.feedparser模块，提供了一个API有利于电子邮件消息，如增量分析从可阻断（如插座）一个源读取的电子邮件消息的文本时是必要的。该BytesFeedParser过程可以被用于解析完全包含在一个电子邮件消息类字节对象，字符串或文件，但该BytesParserAPI可以是用于这种用途的情况下更为方便。两个解析器API的语义和结果是相同的。

这个BytesFeedParserAPI很简单，你创建一个实例，为它提供一堆字节，直到没有更多的字节供给它为止，然后关闭解析器以检索根消息对象。这BytesFeedParser在解析符合标准的消息时非常准确，并且在解析不符合规定的消息方面做得非常好，提供了有关消息如何被破坏的信息。它将填充消息对象的 defects属性以及它在消息中找到的任何问题的列表。查看email.errors模块以获取它可找到的缺陷列表。

以下是以下API的API BytesFeedParser：

class email.parser.BytesFeedParser（*_factory = None*，*，policy = policy.compat32*）

创建一个BytesFeedParser实例。可选*_factory*是无参数可调用的; 如果未指定，则使用message_factory该*策略*。每当需要新的消息对象时调用 *_factory*。

如果指定了*策略*，则使用它指定的规则来更新消息的表示。如果未设置*策略*，请使用该compat32策略，该策略与Python 3.2版本的电子邮件包保持向后兼容，并提供 Message默认

工厂。所有其他政策都提供EmailMessage默认的*_factory*。有关其他*策略*控件的更多信息，请参阅 policy文档。

注意：**应始终指定策略关键字**；email.policy.default在未来的Python版本中，默认设置将变为。

3.2版本中的新功能

*在版本3.3中进行了更改：*添加了*策略*关键字。

在版本3.6中更改： *_factory*默认为该策略message_factory。

feed（*数据*）

为解析器提供更多数据。 *数据*应该是包含一行或多行的类似字节的对象。线可以是部分的，解析器会将这些局部线合适地拼接在一起。这些行可以具有三种公共行结束中的任何一行：回车，换行，回车和换行（甚至可以混合使用）。

close（）

完成所有以前提供的数据的解析并返回根消息对象。它是未定义的，如果feed()在调用此方法后调用它，会发生什么情况。

class email.parser.FeedParser（*_factory = None*，*，policy = policy.compat32*）

像BytesFeedParser除了feed()方法的输入 必须是字符串一样工作。这是效用有限，因为对于这样的消息是有效的唯一办法是，它仅包含ASCII文本，或者，如果utf8是 True，没有二进制附件。

*在版本3.3中进行了更改：*添加了*策略*关键字。

19.1.2.2。解析器

BytesParser从email.parser模块导入的类提供了一个API，当消息的完整内容可用于类似字节的对象或文件时，该类可用于解析消息。该 email.parser模块还提供Parser对于解析字符串，只有头部的解析器，BytesHeaderParser并且 HeaderParser，可如果你只是在邮件的标题感兴趣的使用。 BytesHeaderParser并且HeaderParser 在这些情况下可以更快，因为它们不会尝试解析消息体，而是将有效负载设置为原始主体。

*类*email.parser.BytesParser（*_class = None*，*，policy = policy.compat32*）

创建一个BytesParser实例。该*_class*和*政策* 参数具有相同的含义和语义的*_factory* 和*政策*的争论BytesFeedParser。

注意：**应始终指定策略关键字**；email.policy.default在未来的Python版本中，默认设置将变为。

*在版本3.3中进行了更改：*删除了在2.4中已弃用的*严格*参数。添加了 *策略*关键字。

在版本3.6中更改： *_class*默认为该策略message_factory。

parse（*fp*，*headersonly = False*）

读取二进制文件对象*fp中的*所有数据，解析结果字节，并返回消息对象。 *fp*必须同时支持readline()和read() 方法。

包含在*fp中*的字节必须格式化为一个块**RFC 5322** （或者，如果utf8是True，**RFC 6532**）风格的标题和标题延续线，可选地以信封标题开头。标题块由数据结尾或空行结束。标题块之后是消息的正文（可能包含MIME编码的子部分，包括具有 *Content-Transfer-Encoding的子*部分8bit。

可选的*headersonly*是一个标志，指定是否在阅读头文件后停止解析。默认值是False，表示它解析文件的全部内容。

parsebytes（*bytes*，*headersonly = False* ）

与该parse()方法类似，除了它需要一个类似字节的对象而不是类似文件的对象。调用此方法 类字节对象相当于包装*字节*在 BytesIO第一实例和调用parse()。

可选的*headeronly*与parse()方法一样。

3.2版本中的新功能

*类*email.parser. BytesHeaderParser（*_class = None*，*，policy = policy.compat32* ）

完全一样BytesParser，只是使用了*headeronly的*默认值True。

3.3版本的新功能

*类*email.parser. Parser（*_class = None*，*，policy = policy.compat32* ）

这个类并行BytesParser，但处理字符串输入。

*版本3.3中更改：*删除*严格的*参数。添加了*策略*关键字。

*在版本3.6中更改：*_class默认为该策略message_factory。

parse（*fp*，*headersonly = False* ）

从文本模式文件对象*fp中*读取所有数据，解析生成的文本并返回根消息对象。 *fp*必须同时支持文件类对象readline()和 read()方法。

除了文本模式要求之外，此方法的运行方式与此类似 BytesParser.parse()。

parsestr（*text*，*headersonly = False* ）

与该parse()方法类似，除了它需要一个字符串对象而不是类似文件的对象。调用一个字符串这个方法相当于环绕*文本*在StringIO一审和调用parse()。

可选的*headeronly*与parse()方法一样。

*类*email.parser. HeaderParser（*_class = None*，*，policy = policy.compat32* ）

完全一样Parser，只是使用了*headeronly的*默认值True。

由于从字符串或文件对象创建消息对象结构是一项常见任务，因此提供了四个函数作为方便。它们在顶层email包名称空间中可用。

email. message_from_bytes（*s*，*_class = None*，*，policy = policy.compat32* ）

从类似字节的对象中返回消息对象结构。这相当于BytesParser().parsebytes(s)。可选的 *_class*和 *strict*与BytesParser类构造函数一样被解释。

3.2版本中的新功能

*版本3.3中更改：*删除*严格的*参数。添加了*策略*关键字。

message_from_binary_file(fp, _class=None, *, policy=policy.compat32)

从打开的二进制文件对象中返回消息对象结构树。这相当于BytesParser().parse(fp)。 *_class*和 *policy*与BytesParser类构造函数一起解释。

3.2版本中的新功能

*版本3.3中更改：*删除*严格的*参数。添加了*策略*关键字。

email.message_from_string(*s*, *_class = None*, *, *policy = policy.compat32*)

从字符串中返回一个消息对象结构。这相当于 Parser().parsestr(s)。 *_class*和*policy*与Parser类构造函数一起解释。

*版本3.3中更改：*删除*严格的*参数。添加了*策略*关键字。

email.message_from_file(*fp*, *_class = None*, *, *policy = policy.compat32*)

从打开的文件对象中返回消息对象结构树。这相当于Parser().parse(fp)。 *_class*和*policy*与Parser类构造函数一起解释。

*版本3.3中更改：*删除*严格的*参数。添加了*策略*关键字。

在版本3.6中更改： *_class*默认为该策略message_factory。

下面是一个如何message_from_bytes()在交互式Python提示符下使用的例子：

```
>>> import email
>>> msg = email.message_from_bytes(myBytes)
```

19.1.2.3。其他注释

以下是关于解析语义的一些注释：

- 大多数非*多部分*类型的消息被解析为具有字符串有效载荷的单个消息对象。这些对象将返回False了 is_multipart() ，并且 iter_parts()将产生一个空列表。
- 所有*多部分*类型的消息都将被解析为一个容器消息对象，其中包含有关其负载的子消息对象列表。外容器消息将返回True了 is_multipart() ，并且 iter_parts()将产生的子部件的列表。
- 大多数内容类型为*message / *的*消息*（如 *message / delivery-status*和*message / rfc822* ）也将被解析为包含长度为1的列表有效内容的容器对象。它们的 is_multipart()方法将返回True。产生的单个元素iter_parts() 将是一个子消息对象。
- 一些非符合标准的消息可能不会对他们的内部一致的*多*-edness。这样的消息可能有一个*类型*为*multipart*的 *Content-Type*头，但是它们的 is_multipart()方法可能会返回False。

如果这些消息被解析FeedParser，他们将MultipartInvariantViolationDefect在其*缺陷*属性列表中包含该类的 一个实例 。详情请参阅email.errors。

19.1.3。email.generator：生成MIME文件

源代码： Lib / email / generator.py

最常见的任务之一是生成由消息对象结构表示的电子邮件消息的平面（序列化）版本。如果要通过smtplib.SMTP.sendmail()或nntplib模块发送消息，或者在控制台上打印消息，则需要执行此操作。采用消息对象结构并生成序列化表示是生成器类的工作。

与email.parser模块一样，您不仅限于捆绑生成器的功能，你可以自己从头开始编写一个。然而，捆绑的生成器知道如何以符合标准的方式生成大多数电子邮件，应该很好地处理MIME和非MIME电子邮件消息，并且设计为使面向字节的解析和生成操作反转，转换policy用于两者。也就是说，通过BytesParser类解析序列化的字节流，然后使用BytesGenerator应该产生与输入[1]相同的输出来重新生成序列化的字节流。（另一方面，在EmailMessage由程序构造的情况下使用生成器 可能导致EmailMessage对象的变化，因为缺省值被填充。）

的Generator类可以被用来压平的消息转换为文本（而不是二进制）序列化表示，但由于Unicode的不能直接表示二进制数据，该消息是必要性转化入东西只包含ASCII字符，采用了标准的电子邮件RFC内容传输编码技术对电子邮件进行编码，以便通过不是“8位清洁”的通道进行传输。

class email.generator.BytesGenerator（*outfp*，*mangle_from_* = *None*，*maxheaderlen* = *None*，***，*policy* = *None*）

返回一个BytesGenerator对象，该对象将提供给该flatten()方法的任何消息或提供给该方法的任何surrogateescape编码文本write()写入文件类对象 *outfp*。 *outfp*必须支持write接受二进制数据的方法。

如果可选*mangle_from_*是True，把一个>角色在身体以实际字符串开头的任何行的前面，也就是 后面在一行的开头空间。 *mangle_from_* 默认为*策略* 设置的值（用于 策略和所有其他策略）。 *mangle_from_*用于消息以unix mbox格式存储时使用（请参阅和为什么内容长度格式不正确）。 ″From ″Frommangle_from_Truecompat32Falsemailbox

如果*maxheaderlen*不是None，则重新折叠比*maxheaderlen*长的任何标题行，或者如果0不重新包装任何标题。如果 *manheaderlen*是None（默认），则根据*策略*设置封装标题和其他消息行。

如果*政策*指定，使用该策略来控制消息生成。如果 *策略*是None（默认），则使用与Message或EmailMessage 传递给的对象关联的策略 flatten来控制消息生成。请参阅email.policy有关*策略*控制的详细信息。

3.2版本中的新功能

*在版本3.3中进行了更改：*添加了*策略*关键字。

在版本3.6中更改： *mangle_from_* 和*maxheaderlen*参数的默认行为是遵循策略。

flatten（*msg*，*unixfrom* = *False*，*linesep* = *None*）

将以*msg*为根的消息对象结构的文本表示形式打印到BytesGenerator 创建实例时指定的输出文件。

如果该policy选项cte_type 是8bit（默认），则将原始解析消息中未修改的任何头文件复制到输出中，并将原始位置设置为高位的任何字节复制，并保留非ASCII *内容传输编码*任何身体部位都有它们。如果cte_type是7bit，则使用ASCII兼容的*Content-Transfer-Encoding*将字节设置为根据需要设置的高位。也就是说，将具有非ASCII *Content-Transfer-Encoding*（*Content-Transfer-Encoding：8位*）的部分转换为ASCII兼容的 *Content-Transfer-Encoding*，并使用MIME在头中编码RFC无效的非ASCII字节unknown-8bit 字符集，从而使它们符合RFC标准。

如果*unixfrom*是True，打印由Unix的邮箱格式（见用于信封头定界符mailbox之前第一的）根消息对象的 **RFC 5322**标头。如果根对象没有信封头，则制作一个标准头。默认是False。请注意，对于子部件，不会打印任何信封头。

如果*linesep*不是None，则将其用作扁平邮件所有行之间的分隔符。如果*linesep*是None（默认），请使用*策略中*指定的值。

clone（*fp*）

BytesGenerator使用完全相同的选项设置返回此实例的独立克隆，并将*fp*作为新的*outfp*。

write（*s*）

编码*小号*使用的ASCII编解码器和surrogateescape错误处理程序，并把它传递给*写*的方法*outfp*传递给 BytesGenerator的构造。

为了方便起见，EmailMessage 提供了方法 as_bytes() 和 bytes(aMessage)（aka __bytes__()），它们简化了消息对象的序列化二进制表示的生成。有关更多细节，请参阅email.message。

由于字符串不能表示二进制数据，因此Generator该类必须将其平滑的任何消息中的任何二进制数据转换为ASCII兼容格式，方法是将它们转换为兼容ASCII的 *Content-Transfer_Encoding*。使用电子邮件RFC的术语，您可以将其视为Generator序列化为不是“8位清理”的I / O流。换句话说，大多数应用程序都希望使用BytesGenerator，而不是Generator。

class email.generator.Generator（*outfp*，*mangle_from_* = *None*，*maxheaderlen* = *None*，*，policy* = *None*）

返回一个Generator对象，将写提供到的任何消息flatten() 的方法，或提供给任何文本write() 的方法，所述类文件对象 *outfp*。 *outfp*必须支持write接受字符串数据的 方法。

如果可选*mangle_from_*是True，把一个>角色在身体以实际字符串开头的任何行的前面，也就是 后面在一行的开头空间。 *mangle_from_* 默认为*策略* 设置的值（用于 策略和所有其他策略）。 *mangle_from_*用于消息以unix mbox格式存储时使用（请参阅和为什么内容长度格式不正确）。″From ″Frommangle_from_Truecompat32Falsemailbox

如果*maxheaderlen*不是None，则重新折叠比*maxheaderlen*长的任何标题行，或者如果0不重新包装任何标题。如果 *manheaderlen*是None（默认），则根据*策略*设置封装标题和其他消息行。

如果*政策*指定，使用该策略来控制消息生成。如果 *策略*是None（默认），则使用与Message或EmailMessage 传递给的对象关联的策略 flatten来控制消息生成。请参阅email.policy有关*策略*控制的详细信息。

*在版本3.3中进行了更改：*添加了*策略*关键字。

*在版本3.6中更改：*mangle_from_ 和*maxheaderlen*参数的默认行为是遵循策略。

flatten（*msg*，*unixfrom = False*，*linesep = None* ）

将以*msg*为根的消息对象结构的文本表示形式打印到Generator 创建实例时指定的输出文件。

如果该policy选项cte_type 是8bit，则生成该消息，就好像该选项已设置为7bit。（这是必需的，因为字符串不能表示非ASCII字节。）使用与ASCII兼容的*Content-Transfer-Encoding*根据需要将高位设置为高位。也就是说，将具有非ASCII *Cotnent-Transfer-Encoding*（*Content-Transfer-Encoding：8位*）的部分转换为ASCII兼容的*Content-Transfer-Encoding*，并且使用MIME unknown-8bit字符集在头中编码RFC无效的非ASCII字节，从而使它们符合RFC标准。

如果*unixfrom*是True，打印由Unix的邮箱格式（见用于信封头定界符mailbox之前第一的）根消息对象的 **RFC 5322**标头。如果根对象没有信封头，则制作一个标准头。默认是False。请注意，对于子部件，不会打印任何信封头。

如果*linesep*不是None，则将其用作扁平邮件所有行之间的分隔符。如果*linesep*是None（默认），请使用*策略中*指定的值。

*在版本3.2中进行了更改：*增加了对重新编码8bit消息体的支持以及 *linesep*参数。

clone（*fp* ）

Generator使用完全相同的选项返回此实例的独立克隆，并将*fp*作为新的*outfp*。

write（*s* ）

写*小号*的*写*的方法*outfp*传递给 Generator的构造。这为Generator在print() 函数中使用的实例提供了足够类似文件的API。

为了方便起见，EmailMessage提供了方法 as_string()和str(aMessage)（aka __str__()），它们简化了消息对象的格式化字符串表示的生成。有关更多细节，请参阅 email.message。

该email.generator模块还提供了派生类， DecodedGenerator它类似于Generator基类，不同之处在于非*文本*部分不是序列化的，而是在输出流中由从部分信息填充的模板派生的字符串表示。

class email.generator.DecodedGenerator（*outfp*，*mangle_from_ = None*，*maxheaderlen = None*，*fmt = None*，*，*policy = None* ）

就像Generator，除了传递给消息的任何子部分Generator.flatten()，如果子部分是主类型 *文本*，打印子部分的解码有效负载，如果主类型不是*文本*，而不是打印它填充字符串*fmt*使用来自零件的信息并打印所得到的填充字符串。

要填写*fmt*，请执行，其中 是由以下键和值组成的字典：fmt % part_infopart_info

- `type`- 非文本部分的完整MIME类型
- `maintype`- 非文本部分的主要MIME类型
- `subtype`- 非文本部分的子MIME类型
- `filename`- 非文本部分的文件名
- `description`- 与非文本部分关联的描述
- `encoding`- 非文本部分的内容传输编码

如果*fmt*是`None`，请使用以下默认*fmt*：

> “[消息的非文本（%（类型）s）部分省略，文件名%（文件名）s]”

可选*_mangle_from_*和*maxheaderlen*与 `Generator`基类相同。

脚注

[1] 此声明假定您使用适当的设置 `unixfrom`，并且没有`policy`要求自动调整的设置（例如，`refold_source`必须是`none`，这 *不是*默认设置）。它也不是100%正确的，因为如果消息不符合标准，偶尔会在解析错误恢复期间丢失关于确切原始文本的信息。在可能的情况下修复后面这些边缘情况是目标。

19.1.4。email.policy：策略对象

3.3版本的新功能

源代码： Lib / email / policy.py

该email软件包的主要重点是处理各种电子邮件和MIME RFC所述的电子邮件。但是，电子邮件消息的一般格式（每个由一个名称后跟一个冒号后跟一个值，整个块后跟一个空行和一个任意'body'组成的头部字段块）是一种已经找到的格式电子邮件领域之外的实用程序。其中一些用途与主要电子邮件RFC相当接近，有些则没有。即使在使用电子邮件时，有时也希望打破对RFC的严格遵守，例如生成可与不遵循标准的电子邮件服务器进行互操作的电子邮件，或实施您想要以违反方式使用的扩展程序标准。

策略对象为电子邮件包提供了处理所有这些不同用例的灵活性。

一个Policy对象封装了一组属性和方法，用于控制电子邮件包在使用过程中各个组件的行为。Policy实例可以传递给电子邮件包中的各种类和方法来改变默认行为。下面介绍可设置的值及其默认值。

电子邮件包中有所有类都使用默认策略。对于所有的parser类和相关的便利函数，对于Message类，这是Compat32 策略，通过其相应的预定义实例compat32。此策略提供了与电子邮件包的Python3.3之前版本完全向后兼容（在某些情况下，包括缺陷兼容性）。

*策略*关键字to的 默认值EmailMessage是EmailPolicy策略，通过它的预定义实例default。

当 创建一个Message或一个EmailMessage对象时，它会获取一个策略。如果消息是由a创建的parser，则传递给解析器的策略将成为其创建的消息使用的策略。如果消息是由程序创建的，则可以在创建策略时指定该策略。当消息传递给a时 generator，生成器默认使用消息中的策略，但您也可以将特定策略传递给生成器，以覆盖存储在消息对象中的策略。

在将来的Python版本中，类和解析器便利函数的*policy*关键字的默认值**将会发生变化**。因此，在调用模块中描述的任何类和函数时 **，应始终明确指定要使用的策略**。email.parserparser

本文档的第一部分介绍了抽象基类的功能Policy，该 抽象基类定义了所有策略对象包括的常用功能compat32。这包括由电子邮件包内部调用的某些钩子方法，自定义策略可以重写以获得不同的行为。第二部分描述具体类，EmailPolicy并Compat32分别实现提供标准行为和向后兼容行为和功能的钩子。

Policy实例是不可变的，但它们可以被克隆，接受与类构造函数相同的关键字参数，并返回一个新的 Policy实例，该实例是原始的副本，但指定的属性值已更改。

例如，可以使用以下代码从磁盘上的文件中读取电子邮件并将其传递到sendmailUnix系统上的系统程序：

```
>>> from email import message_from_binary_file
>>> from email.generator import BytesGenerator
>>> from email import policy
>>> from subprocess import Popen, PIPE
>>> with open('mymsg.txt', 'rb') as f:
```

```
...     msg = message_from_binary_file(f, policy=policy.default)
>>> p = Popen(['sendmail', msg['To'].addresses[0]], stdin=PIPE)
>>> g = BytesGenerator(p.stdin, policy=msg.policy.clone(linesep='\r\n'))
>>> g.flatten(msg)
>>> p.stdin.close()
>>> rc = p.wait()
```

这里我们告诉BytesGenerator在创建要输入的二进制字符串时使用RFC正确的行分隔符sendmail's stdin，其中默认策略将使用\n行分隔符。

某些电子邮件包方法接受*策略*关键字参数，允许为该方法重写策略。例如，以下代码使用前面示例中*msg*对象的as_bytes()方法，并使用运行它的平台的本机行分隔符将消息写入文件：

```
>>> import os
>>> with open('converted.txt', 'wb') as f:
...     f.write(msg.as_bytes(policy=msg.policy.clone(linesep=os.linesep)))
17
```

策略对象也可以使用添加运算符进行组合，从而生成一个策略对象，其设置是总计对象的非默认值的组合：

```
>>> compat_SMTP = policy.compat32.clone(linesep='\r\n')
>>> compat_strict = policy.compat32.clone(raise_on_defect=True)
>>> compat_strict_SMTP = compat_SMTP + compat_strict
```

此操作不可交换; 也就是添加对象的顺序。为了显示：

```
>>> policy100 = policy.compat32.clone(max_line_length=100)
>>> policy80 = policy.compat32.clone(max_line_length=80)
>>> apolicy = policy100 + policy80
>>> apolicy.max_line_length
80
>>> apolicy = policy80 + policy100
>>> apolicy.max_line_length
100
```

class email.policy.Policy (** *kw*)

这是所有策略类的抽象基类。它为几个简单的方法提供默认实现，以及不可变性属性，clone()方法和构造函数语义的实现。

策略类的构造函数可以传递各种关键字参数。可以指定的参数是此类上的任何非方法属性，以及具体类上的任何其他非方法属性。在构造函数中指定的值将覆盖相应属性的默认值。

该类定义了以下属性，因此可以在任何策略类的构造函数中传递以下值：

max_line_length

序列化输出中任何行的最大长度，不包括行结束符。默认值是78**RFC 5322**。值为 0或None表示根本不应换行。

linesep

在串行输出中用于终止行的字符串。默认是\n因为这是Python使用的内部行尾规则，尽管\r\nRFC是必需的。

cte_type
控制可能或需要使用的内容传输编码的类型。可能的值是：

7bit	所有数据必须是“7位清理”（仅限ASCII）。这意味着必要的数据将使用quoted-printable或base64编码进行编码。
8bit	数据不限于7位清理。标题中的数据仍然需要仅用于ASCII，因此将被编码（请参阅 fold_binary()和utf8下面的例外情况），但正文部分可能使用8bitCTE。

一个cte_type价值8bit仅适用于BytesGenerator，不 Generator，因为字符串不能包含二进制数据。如果 Generator是在指定的策略操作 cte_type=8bit，它会表现得好像cte_type是7bit。

raise_on_defect
如果True遇到的任何缺陷都会作为错误提出。如果 False（默认），缺陷将传递给register_defect()方法。

mangle_from_
如果在身体中True以*“From”*开始的行通过放在>它们的前面而逃脱。当消息被生成器序列化时使用此参数。默认：False。

*新的3.5版：*该*mangle_from_*参数。

message_factory
用于构造新的空消息对象的工厂函数。构建消息时由解析器使用。默认为None，在这种情况下Message使用。

3.6版本中的新功能。

以下Policy方法旨在由使用电子邮件库的代码通过自定义设置创建策略实例：

clone（ ** *kw* ）
返回Policy属性与当前实例具有相同值的新实例，除非通过关键字参数为这些属性赋予了新值。

其余Policy方法由电子邮件包代码调用，并且不打算由使用电子邮件包的应用程序调用。自定义策略必须实现所有这些方法。

handle_defect（ *obj*，*缺陷* ）
处理在*obj*上发现的*缺陷*。当电子邮件包调用这个方法时，*缺陷*将永远是一个子类 。Defect

默认实现检查raise_on_defect标志。如果是的话True，*缺陷*会作为例外提出。如果是False（默认），则传递*obj*和*缺陷*register_defect()。

register_defect（ *obj*，*缺陷* ）

在*obj*上注册*缺陷*。在电子邮件包中，*缺陷*将永远是一个子类。Defect

默认实现调用*obj*属性的append方法。当电子邮件包调用时，*obj*通常会有一个具有方法的属性。与电子邮件包一起使用的自定义对象类型（例如自定义对象）也应提供此类属性，否则分析的消息中的缺陷将引发意外错误。defectshandle_defectdefectsappendMessage

header_max_count（*名字*）

返回名为*name*的头部允许的最大数量。

当标题被添加到EmailMessage 或Message对象时调用。如果返回的值不是0或者None，并且已经有多个标题*名称*大于或等于返回值 的标题，ValueError则会引发a。

因为默认行为Message.__setitem__是将值附加到标题列表中，所以很容易在不实现它的情况下创建重复标题。该方法允许某些头部受限于可以通过Message编程添加到头部的头部实例的数量。（解析器没有观察到这个限制，它将忠实地产生与正在解析的消息中存在的头相同数量的头。）

None所有标题名称的默认实现都会返回。

header_source_parse（*源代码*）

电子邮件包以字符串列表的形式调用此方法，每个字符串以在被解析的源文件中找到的行分隔字符结尾。第一行包含字段标题名称和分隔符。源中的所有空白都将保留。该方法应该返回要存储在其中的 元组来表示解析的头。(name, value)Message

如果实现希望保持与现有电子邮件包策略的兼容性，*名称*应该是大小写保留的名称（所有字符直到:“分隔符”），而*值*应该是展开的值（所有行分隔符都被删除，但空格保持不变），没有领先的空白。

*源代码行*可能包含代理转换的二进制数据。

没有默认的实现

header_store_parse（*名称*，*值*）

当应用程序以Message编程方式修改（而不是Message由分析器创建）时，电子邮件包使用应用程序提供的名称和值调用此方法。该方法应该返回要存储在元素中的元组来表示标题。(name, value)Message

如果实现希望保持与现有电子邮件包策略的兼容性，那么*名称*和*值*应该是不改变传入参数内容的字符串或字符串子类。

没有默认的实现

header_fetch_parse（*名称*，*值*）

电子邮件包使用当前存储在应用程序请求该头部时的*名称*和*值*调用此方法Message，并且该方法返回的任何内容都是作为正在检索的头部值传递回应用程序的内容。请注意，可能有多个标题存储在同一个名称中Message;该方法将传递指定要返回给应用程序的标头的特定名称和值。

*值*可能包含代理转义的二进制数据。该方法返回的值中不应有代理转义的二进制数据。

没有默认的实现

fold (*名称* , *值*)

电子邮件包使用当前存储在给定标题中的*名称*和*值*调用此方法Message。该方法应该返回一个字符串，该字符串通过用*值*组合*名称*并在适当的位置插入 字符来正确表示标题“已折叠”（根据策略设置）。看到linesepRFC 5322 讨论了折叠电子邮件标题的规则。

*值*可能包含代理转义的二进制数据。该方法返回的字符串中不应有代理转义的二进制数据。

fold_binary (*名称* , *值*)

fold()与之相同，除了返回的值应该是字节对象而不是字符串。

*值*可能包含代理转义的二进制数据。这些可以在返回的字节对象中转换回二进制数据。

class email.policy. EmailPolicy (** *kw*)

此具体Policy提供的行为旨在完全符合当前的电子邮件RFC。这些包括（但不限于）RFC 5322，RFC 2047和当前的MIME RFC。

该策略添加了新的标题解析和折叠算法。标题不是简单的字符串，而是str具有取决于字段类型的属性的子类。解析和折叠算法完全实现 RFC 2047和RFC 5322。

该message_factory 属性的默认值是EmailMessage。

除了上面列出的适用于所有策略的可设置属性之外，此策略还添加了以下附加属性：

3.6版新增功能：[1]

utf8

如果False，跟着RFC 5322，通过将它们编码为“编码字”来支持标题中的非ASCII字符。如果True，跟着 RFC 6532并使用utf-8编码标题。以这种方式格式化的邮件可能会传递到支持SMTPUTF8扩展名的SMTP服务器（RFC 6531）。

refold_source

如果Message对象中标题的值源自a parser（与程序设置相反），则此属性指示在将消息转换回序列化形式时，生成器是否应该重新生成该值。可能的值是：

none	所有来源值都使用原始折叠
long	具有长于任何行的源值 max_line_length将被重新折叠
all	所有的价值都被重新折叠。

默认是long。

header_factory

可调用有两个参数，name并且value，在这里 name是一个首部字段名称和value是未折叠头字段值，并返回表示该标题的字符串的子类。提供了默认header_factory（请参阅参考资料headerregistry），它支持各种地址和日期的自定义分析**RFC 5322** 标头字段类型以及主要的MIME标头字段类型。将来会增加对其他自定义分析的支持。

content_manager

至少有两个方法的对象：get_content 和 set_content。当 get_content() 或 set_content()一个的方法 EmailMessage对象被调用时，它调用此对象的相应方法，传递消息对象作为第一个参数，并传递给它作为附加参数，任何参数或关键字。默认情况下content_manager设置为 raw_data_manager。

3.4版新增功能

该类提供了以下抽象方法的以下具体实现Policy：

header_max_count（*名字*）

返回max_count用于使用给定名称表示标题的专用类的属性值。

header_source_parse（*源代码*）

该名称被解析为一切直到':'并且未修改返回。该值通过从第一行剩余部分剥离前导空白字符，将所有后续行连接在一起并剥离任何尾部回车符或换行符来确定。

header_store_parse（*名称*，*值*）

该名称将保持不变。如果输入值有一个name 属性，并且匹配忽略大小写的*名称*，则返回值不变。否则，*名称*和*值*将传递给 header_factory，并将生成的头对象作为值返回。在这种情况下，ValueError如果输入值包含CR或LF字符，则会引发a。

header_fetch_parse（*名称*，*值*）

如果该值有一个name属性，它将返回到未修改。否则，将*名称*和删除了任何CR或LF字符的*值*传递给该header_factory对象，并返回生成的头对象。任何代理转义的字节都会变成unicode未知字符的字形。

fold（*名称*，*值*）

标题折叠由refold_source策略设置控制。当且仅当它没有一个name属性（具有name属性意味着它是某种类型的头对象）时，该值才被认为是“源值”。如果源值需要根据策略重新折叠，则通过将*名称*和*值*以及删除的任何CR和LF字符传递给该对象，将其转换为标题对象header_factory。通过fold使用当前策略调用它的方法来完成标题对象的折叠。

源值被分成几行使用splitlines()。如果该值不被重新折叠，则会使用linesep来自策略的行重新加入行 并返回。例外是包含非ASCII二进制数据的行。在这种情况下，无论refold_source设置如何，该值都会重新折叠，这会导致使用unknown-8bitcharset对二进制数据进行CTE编码。

fold_binary（*名称*，*值*）

一样的fold()，如果cte_type是7bit，除了返回的值是字节。

如果cte_type是8bit，则将非ASCII二进制数据转换回字节。refold_header无论二进制数据是由单字节字符还是多字节字符组成，都无法知道二进制数据的标题，因为无

法知道二进制数据是由单字节字符还是多字节字符组成的。

EmailPolicy提供适用于特定应用程序域的以下默认值实例。请注意，将来这些实例（尤其是HTTP实例）的行为可能会进行调整，以便更加符合与其域相关的RFC。

email.policy.**default**

EmailPolicy所有默认值均未更改的实例。该策略使用标准的Python \n行结束符而不是RFC正确的 \r\n。

email.policy.**SMTP**

适合按照电子邮件RFC来序列化消息。喜欢default，但linesep设置为\r\n，符合RFC。

email.policy.**SMTPUTF8**

一样SMTP，除了utf8是True。用于将消息序列化到消息存储而不使用头中的编码字。如果发件人地址或收件人地址具有非ASCII字符（该smtplib.SMTP.send_message()方法会自动处理该字符），则只应用于SMTP传输。

email.policy.**HTTP**

适用于序列化HTTP头中使用的头。就像 SMTP那个max_line_length设置为None（无限制）一样。

email.policy.**strict**

便利实例。 default除了raise_on_defect设置为相同外 True。这使得任何政策都可以通过写作而变得严格：

```
somepolicy + policy.strict
```

通过所有这些EmailPolicies，电子邮件包的有效API从Python 3.2 API以下列方式更改：

- Message在解析头文件的结果上设置头文件并创建头文件对象。
- 从Message解析头中的结果中获取头值，并创建并返回头对象。
- 任何标题对象或由于策略设置而重新折叠的任何标题都会使用完全实现RFC折叠算法的算法进行折叠，包括了解需要和允许的编码字的位置。

从应用程序视图来看，这意味着通过该EmailMessage对象获得的任何标题 都是具有额外属性的标题对象，其字符串值是标题的完全解码的统一码值。同样，可以使用unicode字符串为标题分配新值或创建新标题，并且策略将负责将unicode字符串转换为正确的RFC编码格式。

标题对象及其属性在中描述 headerregistry。

class email.policy.**Compat32**(***kw*)

这具体Policy是向后兼容性政策。它复制Python 3.2中电子邮件包的行为。该 policy模块还定义了此类的一个实例， compat32用作默认策略。因此电子邮件包的默认行为是保持与Python 3.2的兼容性。

以下属性的值与Policy默认值不同 ：

mangle_from_

默认是True。

该类提供了以下抽象方法的以下具体实现Policy：

header_source_parse（*源代码*）
: 该名称被解析为一切直到':'并且未修改返回。该值通过从第一行剩余部分剥离前导空白字符，将所有后续行连接在一起并剥离任何尾部回车符或换行符来确定。

header_store_parse（*名称*，*值*）
: 名称和值未经修改即返回。

header_fetch_parse（*名称*，*值*）
: 如果该值包含二进制数据，则Header使用unknown-8bit字符集将其转换为 对象。否则，它将不被修改。

fold（*名称*，*值*）
: 标题使用Header折叠算法进行折叠，折叠算法保留值中的现有换行符，并将每个结果行包装为max_line_length。非ASCII二进制数据使用unknown-8bitcharset 进行CTE编码。

fold_binary（*名称*，*值*）
: 标题使用Header折叠算法进行折叠，折叠算法保留值中的现有换行符，并将每个结果行包装为max_line_length。如果cte_type是 7bit ，非ASCII二进制数据使用unknown-8bit 字符集进行CTE编码。否则，将使用原始源标题，其现有的换行符和它可能包含的任何（RFC无效的）二进制数据。

email.policy. compat32
: Compat32在Python 3.2中提供向后兼容电子邮件包行为的实例。

脚注

[1] 最初作为临时特征添加在3.3中。

19.1.5。email.errors：异常和缺陷类

源代码： Lib / email / errors.py

以下异常类在email.errors模块中定义：

*异常*email.errors.MessageError
这是email包可以引发的所有异常的基类。它来自标准Exception类并且没有定义其他方法。

*异常*email.errors.MessageParseError
这是该类提出的异常的基 Parser类。它来源于 MessageError。这个类也被解析器在内部使用headerregistry。

*异常*email.errors.HeaderParseError
解析时引发一些错误条件 消息的 **RFC 5322**标题，这个类是派生自MessageParseError。set_boundary()如果在调用方法时内容类型未知，该 方法将引发此错误。 Header可能会因为某些base64解码错误而引发此错误，并且当尝试创建一个看起来包含嵌入标头的标头时（也就是说，有一条应该是没有前导空白并且看起来像头）。

*异常*email.errors.BoundaryError
已弃用且不再使用。

*异常*email.errors.MultipartConversionError
当有效载荷被添加到使用的Message对象时引发add_payload()，但有效载荷已经是标量，并且消息的*Content-Type*主类型不是 *多部分*或缺失。 MultipartConversionError乘法继承自MessageError内置TypeError。

由于Message.add_payload()已被弃用，这种例外在实践中很少引起。但是，如果attach()方法在派生自MIMENonMultipart （eg MIMEImage ）的类的实例上调用， 也可能会引发异常。

以下是FeedParser 解析消息时可以找到的缺陷列表。请注意，将缺陷添加到发现问题的消息中，例如，如果嵌套在*multipart / alternative*内的消息 具有格式错误的头，则嵌套的消息对象将具有缺陷，但包含的消息不会。

所有的缺陷类别都是从中分类的email.errors.MessageDefect。

- NoBoundaryInMultipartDefect- 声称是多部分的消息，但没有*边界*参数。
- StartBoundaryNotFoundDefect- *Content-Type*头中声明的起始边界 从未找到。
- CloseBoundaryNotFoundDefect - 找到了起始边界，但没有找到相应的边界。

  3.3版本的新功能

- FirstHeaderLineIsContinuationDefect - 该消息有一个延续线作为其第一个标题行。

- `MisplacedEnvelopeHeaderDefect` - 在标题块的中间找到“Unix From”标题。
- `MissingHeaderBodySeparatorDefect` - 解析没有前导空格但不包含':'的标题时发现一行。解析继续假定该行代表正文的第一行。

  3.3版本的新功能

- `MalformedHeaderDefect` - 发现头部缺少冒号，或者是其他格式不正确。

  *自3.3版弃用：*此缺陷尚未用于多个Python版本。

- `MultipartInvariantViolationDefect`- 一条消息声称是 *多部分*，但没有发现子部分。请注意，当消息具有此缺陷时，`is_multipart()`即使其内容类型声称为*多部分*，其方法也可能返回false 。
- `InvalidBase64PaddingDefect` - 解码base64编码字节块时，填充不正确。添加了足够的填充以执行解码，但解码后的字节可能无效。
- `InvalidBase64CharactersDefect` - 解码base64编码字节块时，遇到base64字母以外的字符。这些字符将被忽略，但结果解码的字节可能无效。

19.1.6。email.headerregistry：自定义标题对象

源代码： Lib / email / headerregistry.py

3.6版新增功能：[1]

标题由自定义的子类表示str。用于表示给定的报头中的特定类是由确定 header_factory的policy在效果被创建的头时。本部分记录了header_factory电子邮件包执行的具体 操作 符合**RFC 5322**标准的电子邮件消息，它不仅为各种标题类型提供定制的标题对象，而且还为应用程序添加其自定义标题类型提供了扩展机制。

当使用派生自的任何策略对象时 EmailPolicy ，所有标题都由其生成 HeaderRegistry并BaseHeader作为它们的最后一个基类。每个头类都有一个由头的类型决定的额外基类。例如，许多头文件都将该类 UnstructuredHeader作为其他基类。头文件的专用第二类由头文件的名称决定，使用存储在头文件中的查找表HeaderRegistry。所有这些都是针对典型应用程序透明地进行管理的，但提供了用于修改默认行为以供更复杂的应用程序使用的接口。

下面的部分首先记录头基类及其属性，随后是用于修改行为的API HeaderRegistry ，最后是用于表示从结构头解析的数据的支持类。

*类*email.headerregistry.BaseHeader（*名称*，*值*）

*姓名*和*价值*BaseHeader从 header_factory呼叫传递到。任何头对象的字符串值是完全解码为Unicode 的*值*。

这个基类定义了以下只读属性：

name

标题的名称（':'之前的字段部分）。这恰恰是在传递值 header_factory呼吁*名称*；那就是保存案件。

defects

HeaderDefect实例元组报告解析期间发现的任何RFC合规性问题。电子邮件软件包试图完成关于检测合规性问题。请参阅errors 模块以了解可能会报告的缺陷类型。

max_count

这种类型的标头的最大数量可以相同 name。None意味着无限的价值。BaseHeader这个属性的值是None；预计专门的头类将根据需要覆盖该值。

BaseHeader 还提供以下方法，该方法由电子邮件库代码调用，通常不应由应用程序调用：

fold（*，*政策*）

linesep 根据需要返回包含字符的字符串，以根据*策略*正确折叠标题 。甲cte_type的8bit将被视为好像它是7bit ，由于头部可以不包含任意的二进制数据。如果utf8是False ，则非ASCII数据**RFC 2047**编码。

BaseHeader本身不能用于创建标题对象。它定义了一个协议，每个专用头与之协作以产生头对象。具体来说，BaseHeader要求专业课提供一个classmethod()名称parse。这个方法被调用如下：

```
parse(string, kwds)
```

kwds是一个包含一个预初始化密钥的字典，defects。 defects是一个空的列表。解析方法应该将任何检测到的缺陷追加到这个列表中。返回时，kwds字典*必须*包含至少键decoded和值的值defects。 decoded 应该是头部的字符串值（也就是说，头部值完全解码为unicode）。解析方法应该假定该*字符串*可能包含内容传输编码部分，但应该正确处理所有有效的Unicode字符，以便它可以解析未编码的标头值。

BaseHeader的__new__随后创建报头的实例，并调用其 init方法。如果专门课程init 希望设置超出BaseHeader自身提供的属性的其他属性，则只需提供一种方法 。这样的init方法应该是这样的：

```
def init(self, *args, **kw):
    self._myattr = kw.pop('myattr')
    super().init(*args, **kw)
```

也就是说，专用类放入kwds 字典中的任何额外内容都应该被移除和处理，并且kw（和args）的其余内容 传递给该BaseHeader init方法。

*类*email.headerregistry.UnstructuredHeader

“非结构化”标题是默认的标题类型 **RFC 5322**。任何没有指定语法的头都被视为非结构化。非结构化标题的经典示例是 *Subject*标题。

在 **RFC 5322中**，非结构化标题是ASCII字符集中任意文本的运行。 然而， **RFC 2047**有一个**RFC 5322**兼容机制，用于在报头值内将非ASCII文本编码为ASCII字符。当一个包含编码字的*值*被传递给构造函数时，UnstructuredHeader解析器会将这些编码的字转换为unicode，然后是非结构化文本的 **RFC 2047**规则。解析器使用启发式来尝试解码某些不符合规范的编码字。在这种情况下注册缺陷，以及在编码字或非编码文本内出现无效字符等问题。

此标题类型不提供其他属性。

*类*email.headerregistry.DateHeader

RFC 5322为电子邮件标题中的日期指定了一种非常具体的格式。该DateHeader分析器承认，日期格式，并承认一些在某些时候“在野外”发现的变异形式。

此标题类型提供了以下附加属性：

datetime

如果标题值可以被识别为一种或另一种形式的有效日期，则该属性将包含datetime 表示该日期的实例。如果输入日期的时区被指定为-0000（表示它是UTC，但不包含关于源时区的信息），那么datetime将是一个天真的datetime。如果找到一个特定的时区偏移量（包括 *+0000* ），然后 datetime 将包含意识到 datetime ，使用datetime.timezone记录时区偏移。

decoded标题的值由格式化 datetime根据RFC 5322规则; 也就是说，它被设置为：

```
email.utils.format_datetime(self.datetime)
```

创建时DateHeader，*值*可能是 datetime实例。这意味着，例如，下面的代码是有效的，并且可以预期：

```
msg['Date'] = datetime(2011, 7, 15, 21)
```

因为这datetime太天真了，它将被解释为UTC时间戳，并且结果值将有一个时区-0000。更有用的是使用localtime()该utils模块的功能 ：

```
msg['Date'] = utils.localtime()
```

本示例使用当前时区偏移量将日期标题设置为当前时间和日期。

*类*email.headerregistry.AddressHeader

地址标题是最复杂的结构化标题类型之一。的AddressHeader类提供的通用接口的任何地址报头。

此标题类型提供了以下附加属性：

groups

Group编码在标题值中找到的地址和组的对象元组。地址不属于基团的一部分是在该列表中作为单地址表示 Groups，其display_name是None。

addresses

Address对象的元组，用于对标题值中的所有单独地址进行编码。如果标题值包含任何组，则该组中的单个地址将包含在组中出现该值的地方（即地址列表被“展平”为一维列表）。

decoded标题的值将所有编码字解码为unicode。 idna编码的域名也被解码为unicode。该decoded值由属性元素join的 str值设置。 groups', '

可以使用任意组合的列表Address和Group对象来设置地址头的值。 Group对象，其display_name是None将被解释为单个地址，这允许通过使用从所获得的列表中完整组被复制的地址列表groups的源标题的属性。

*类*email.headerregistry.SingleAddressHeader

它的一个子类AddressHeader添加了一个附加属性：

address

由标题值编码的单个地址。如果标题值实际上包含多个地址（这在默认情况下会违反RFC policy），那么访问此属性将导致a ValueError。

上面的许多类也有一个Unique变体（例如， UniqueUnstructuredHeader）。唯一的区别是在Unique 变体max_count中设置为1。

*类*email.headerregistry.MIMEVersionHeader

对于*MIME-Version*头，实际上只有一个有效值，那就是1.0。为了将来打样，这个头类支持其他有效的版本号。如果版本号每个都有一个有效值**RFC 2045**，那么头对象将具有None以下属性的非值：

version
: 版本号作为字符串，删除任何空格和/或注释。

major
: 主版本号为整数

minor
: 小版本号作为整数

*类*email.headerregistry.ParameterizedMIMEHeader
: MIME标题全部以前缀“Content-”开头。每个特定的标题都有一个特定的值，在该标题的类下描述。有些还可以获取具有通用格式的补充参数列表。该类用作所有带参数的MIME头的基础。

params
: 一个字典将参数名称映射到参数值。

*类*email.headerregistry.ContentTypeHeader
: 一ParameterizedMIMEHeader，处理类*的Content-Type*头。

content_type
: 内容类型字符串，在表单中maintype/subtype。

maintype

subtype

*类*email.headerregistry.ContentDispositionHeader
: 一ParameterizedMIMEHeader，处理该类*内容处置*头。

content-disposition
: inline并且attachment是常用的唯一有效值。

*类*email.headerregistry.ContentTransferEncoding
: 处理*Content-Transfer-Encoding*标头。

cte
: 有效值是7bit，8bit，base64，和 quoted-printable。看到**RFC 2045**获取更多信息。

class email.headerregistry.HeaderRegistry（*base_class = BaseHeader*，*default_class = UnstructuredHeader*，*use_default_map = True* ）
: 这是EmailPolicy默认使用的工厂。 HeaderRegistry使用*base_class*和从它所拥有的注册表中检索的专用类，动态构建用于创建头实例的类。当给定的标题名称没有出现在注册表中时，由*default_class*指定的类将用作专用类。当*use_default_map*是True（缺省值）时，标

头名称到类的标准映射将在初始化期间被复制到注册表中。 *base_class*始终是生成的类 `__bases__`列表中的最后一个类。

默认映射是：

学科：	UniqueUnstructuredHeader
日期：	UniqueDateHeader
怨恨日：	DateHeader
原稿-日期：	UniqueDateHeader
发件人：	UniqueSingleAddressHeader
重发发送者：	SingleAddressHeader
至：	UniqueAddressHeader
怨恨到：	AddressHeader
抄送：	UniqueAddressHeader
重新发送-CC：	AddressHeader
从：	UniqueAddressHeader
怨恨，从：	AddressHeader
回复：	UniqueAddressHeader

`HeaderRegistry` 有以下方法：

`map_to_type`（*self*，*name*，*cls* ）

*name*是要映射的头的名称。它将在注册表中转换为小写字母。 *cls*是与*base_class*一起使用的专用类，用于创建用于实例化与*name*匹配的头的类。

`__getitem__`（*名字*）

构造并返回一个类来处理创建*名称*标题。

`__call__`（*名称*，*值*）

从注册表中检索与*名称*关联的专用头（如果*名称*没有出现在注册表中，则使用*default_class*），并使用*base_class*进行组合，以生成类，调用构造类的构造函数，传递相同的参数列表，并最终返回类由此创建的实例。

以下类是用于表示从结构化头文件中解析出的数据的类，通常可以由应用程序用来构造分配给特定头文件的结构化值。

class `email.headerregistry.Address`（*display_name =''*，*username =''*，*domain =''*，*addr_spec = None* ）

该类用于表示电子邮件地址。地址的一般形式是：

```
[display_name] <username@domain>
```

要么：

```
username@domain
```

每个部分必须符合特定的语法规则 **RFC 5322**。

为了方便，可以指定*addr_spec*而不是*用户名*和 *域*，在这种情况下，将从*addr_spec*中分析*用户名*和*域*。一个*addr_spec*必须是正确的RFC引号的字符串; 如果不是会引发错误。Unicode字符是允许的，并且在序列化时将被属性编码。然而，每上述RFC Unicode是 *未*在地址的用户名部分允许的。 Address

display_name
: 地址的显示名称部分（如果有）已删除全部引用。如果该地址没有显示名称，则该属性将为空字符串。

username
: username地址的一部分，删除了所有引用。

domain
: domain地址的一部分。

addr_spec
: 该username@domain地址的部分，正确地引述用作裸地址（上面显示的第二种形式）。该属性不可变。

__str__（）
: 该str对象的值是根据引用的地址 **RFC 5322**规则，但不包含任何非ASCII字符的内容传输编码。

为了支持SMTP（**RFC 5321**）Address处理一种特殊情况：如果 username和domain都是空字符串（或None），Address则为字符串值<>。

class email.headerregistry. Group（*display_name = None*，*addresses = None* ）
用于表示地址组的类。地址组的一般形式是：

```
display_name: [address-list];
```

为了便于处理由组和单个地址组成的地址列表，Group通过将*display_name*设置为 None并提供单个地址列表作为*地址*，还可以使用a 表示单个地址不属于某个组。

display_name
: 该display_name组的。如果是None且只有一个Address输入addresses，则Group表示一个地址不在一个组中。

addresses
: Address表示组中地址的可能为空的对象元组。

__str__（）
: a的str值Group根据。格式化**RFC 5322**，但没有任何非ASCII字符的内容传输编码。如果 display_name没有，并且列表中有单个Address，则 addresses该str值将str与该单

个相同Address。

脚注

[1] 最初作为临时模块添加到3.3中

19.1.7。email.contentmanager：管理MIME内容

源代码： Lib / email / contentmanager.py

3.6版新增功能： [1]

*类*email.contentmanager. ContentManager

内容管理器的基类。提供了标准的注册表机制来注册MIME内容等表示之间的转换器，以及在get_content与set_content调度方法。

get_content（*msg*，**args*，***kw*）

查一查基础上的处理函数mimetype的*味精*（见下段），调用它，穿过所有的参数，并返回调用的结果。期望的是处理程序将从*msg中*提取有效载荷，并返回一个对关于提取的数据的信息进行编码的对象。

要查找处理程序，请在注册表中查找以下注册表键，并找到第一个键：

- 表示完整MIME类型的字符串（maintype/subtype）
- 代表的字符串 maintype
- 空的字符串

如果这些键都不生成处理程序，请KeyError为完整的MIME类型提出一个。

set_content（*msg*，*obj*，**args*，***kw*）

如果maintype是multipart，提出一个TypeError；否则查找基于类型的处理函数*OBJ*（见下段），呼吁clear_content()对 *味精*，并调用处理函数，通过所有的参数。期望的是处理程序将转换并将*obj存*储到 *msg中*，也可能对*msg*进行其他更改，例如添加各种MIME头以对解释存储的数据所需的信息进行编码。

要找到处理程序，请获取*obj*（）的类型，然后在注册表中查找以下键，并找到第一个键：typ = type(obj)

- 类型本身（typ）
- 该类型的完全限定名（）。typ.__module__ + '.' + typ.__qualname__
- 类型的品名（typ.__qualname__）
- 该类型的名称（typ.__name__）。

如果以上任何一项都不匹配，则对MRO（typ.__mro__）中的每种类型重复上述所有检查。最后，如果没有其他键产生处理程序，请检查键的处理程序None。如果没有处理程序None，KeyError则为该类型的完全限定名称引发一个。

如果不存在，也添加*MIME-Version*标头（另请参阅MIMEPart）。

add_get_handler（*key*，*handler*）

将函数*处理程序*记录为*键*的处理程序。有关*密钥*的可能值，请参阅get_content()。

add_set_handler（*typekey*，*handler*）

记录*处理程序*作为在传递类型匹配*typekey*的对象时调用的函数set_content()。有关*typekey*的可能值，请参阅set_content()。

19.1.7.1。内容管理器实例

目前该电子邮件包仅提供一个具体的内容管理器 raw_data_manager，尽管将来可能会添加更多。 raw_data_manager是 由其content_manager提供的 EmailPolicy及其衍生物。

email.contentmanager.**raw_data_manager**

该内容管理器仅提供超出其Message自身提供的最小接口：它仅处理文本，原始字节字符串和Message对象。不过，与基本API相比，它提供了显着的优势：get_content在文本部分将返回unicode字符串，而不需要应用程序手动解码它，set_content提供了丰富的选项，用于控制添加到零件的头文件和控制内容传输编码，并且它使得能够使用各种add_方法，从而简化了多部分消息的创建。

email.contentmanager.**get_content**（*msg*，*errors='replace'*）

以字符串（对于text零件），EmailMessage对象（对于message/rfc822 零件）或bytes对象（对于所有其他非多部件类型）返回零件的有效载荷。提出一个KeyError如果呼吁一个multipart。如果部件是一个 text部分和*错误*被指定时，解码所述有效载荷为Unicode时使用它作为错误处理程序。默认的错误处理程序是 replace。

email.contentmanager.**set_content**（*msg*，*<'str'>*，*subtype="plain"*，*charset='utf-8'cte = None*，*disposition = None*，*filename = None*，*cid = None*，*params = None*，*headers = None*）

email.contentmanager.**set_content**（*MSG*，*<'字节'>*，*maintype*，*亚型*，*CTE = "BASE64"*，*处置=无*，*文件名=无*，*CID =无*，*PARAMS =无*，*标题=无*）

email.contentmanager.**set_content**（*msg*，*<'EmailMessage'>*，*cte = None*，*disposition = None*，*filename = None*，*cid = None*，*params = None*，*headers = None*）

email.contentmanager.**set_content**（*msg*，*<'list'>*，*subtype='mixed'*，*disposition = None*，*filename = None*，*cid = None*，*params = None*，*headers = None*）

添加标题和有效载荷到*msg*：

添加一个*Content-Type*标头maintype/subtype 。

- 为str，设置MIME maintype到text，并设置子类型*亚型*如果指定它，或者plain，如果它不是。
- 对于bytes，使用指定的*主类型*和*子类型*，或者TypeError如果未指定它们，则引发它们。
- 对于EmailMessage对象，请将maintype message设置为，如果指定了子类型，或者将子*类型*设置为子*类型*，则将子*类型*设置为子*类型*rfc822。如果*子类型*是 partial，则引发错误（bytes对象必须用于构造message/partial零件）。
- 对于*<'list'>*，它应该是一个EmailMessage对象列表， 如果指定了，则设置maintype 为multipart，并将其设置subtype为*子类型*，mixed如果不是。如果*<'list'>*中的消息部分具有*MIME-Version*标头，请删除它们。

如果提供*字符集*（仅适用于str），请使用指定的字符集将字符串编码为字节。默认是utf-8。如果指定的*字符集*是标准MIME字符集名称的已知别名，请改为使用标准字符集。

如果设置了*cte*，则使用指定的内容传输编码对有效载荷进行编码，并将*Content-Transfer-Encoding*标头设置为该值。对于可能的值*CTE*是quoted-printable，base64，7bit，8bit，和binary。如果输入不能在指定的编码被编码（例如，指定*的CTE*的7bit为包含非ASCII值的输入），养ValueError。

- 对于str对象，如果未设置*cte*，请使用启发式来确定最紧凑的编码。
- 因为EmailMessage，per**RFC 2046**，如果产生错误*CTE*的quoted-printable或者base64被请求*亚型*rfc822，和任何*CTE*比其他7bit的*亚型*external-body。对于message/rfc822，8bit如果未指定*cte*，请使用。对于所有其他的*子类型*值，请使用7bit。

注意： 一个*CTE*的binary实际上并没有正常工作呢。EmailMessage被修改的对象set_content是正确的，但BytesGenerator没有正确序列化它。

如果*设置了处置*，则将其用作*Content-Disposition*标题的值。如果未指定，并且指定了*文件名*，则添加具有该值的标头attachment。如果未指定*处置*并且未指定*文件名*，则不要添加标题。为唯一有效的值*配置*是attachment和inline。

如果指定了*文件名*，则将其用作*Content-Disposition*标题的filename参数值。

如果指定了*cid*，请添加一个*Content-ID*标头，并将*cid*作为其值。

如果*PARAMS*指定，迭代其items方法和使用所得的对设置在附加参数*的Content-Type*首部。(key, value)

如果指定了*标题*并且是表单的字符串列表或对象列表（通过具有属性与字符串区分），请将标题添加到*消息*。headername: headervalueheadername

脚注

[1] 最初作为临时模块添加到3.4中

19.1.8。email：例子

以下是一些如何使用该email软件包读取，写入和发送简单电子邮件以及更复杂的MIME消息的示例。

首先，让我们看看如何创建和发送简单的文本消息（文本内容和地址都可能包含unicode字符）：

```
# Import smtplib for the actual sending function
import smtplib

# Import the email modules we'll need
from email.message import EmailMessage

# Open the plain text file whose name is in textfile for reading.
with open(textfile) as fp:
    # Create a text/plain message
    msg = EmailMessage()
    msg.set_content(fp.read())

# me == the sender's email address
# you == the recipient's email address
msg['Subject'] = 'The contents of %s' % textfile
msg['From'] = me
msg['To'] = you

# Send the message via our own SMTP server.
s = smtplib.SMTP('localhost')
s.send_message(msg)
s.quit()
```

解析RFC822头文件可以很容易地通过使用parser模块中的类来完成：

```
# Import the email modules we'll need
from email.parser import BytesParser, Parser
from email.policy import default

# If the e-mail headers are in a file, uncomment these two lines:
# with open(messagefile, 'rb') as fp:
#     headers = BytesParser(policy=default).parse(fp)

#  Or for parsing headers in a string (this is an uncommon operation), use:
headers = Parser(policy=default).parsestr(
        'From: Foo Bar <user@example.com>\n'
        'To: <someone_else@example.com>\n'
        'Subject: Test message\n'
        '\n'
        'Body would go here\n')

#  Now the header items can be accessed as a dictionary:
print('To: {}'.format(headers['to']))
print('From: {}'.format(headers['from']))
print('Subject: {}'.format(headers['subject']))
```

```
# You can also access the parts of the addresses:
print('Recipient username: {}'.format(headers['to'].addresses[0].username))
print('Sender name: {}'.format(headers['from'].addresses[0].display_name))
```

以下是如何发送包含可能驻留在目录中的一系列家庭照片的MIME消息的示例：

```
# Import smtplib for the actual sending function
import smtplib

# And imghdr to find the types of our images
import imghdr

# Here are the email package modules we'll need
from email.message import EmailMessage

# Create the container email message.
msg = EmailMessage()
msg['Subject'] = 'Our family reunion'
# me == the sender's email address
# family = the list of all recipients' email addresses
msg['From'] = me
msg['To'] = ', '.join(family)
msg.preamble = 'Our family reunion'

# Open the files in binary mode.  Use imghdr to figure out the
# MIME subtype for each specific image.
for file in pngfiles:
    with open(file, 'rb') as fp:
        img_data = fp.read()
    msg.add_attachment(img_data, maintype='image',
                                 subtype=imghdr.what(None, img_data))

# Send the email via our own SMTP server.
with smtplib.SMTP('localhost') as s:
    s.send_message(msg)
```

以下是如何将目录的全部内容作为电子邮件发送的示例：[1]

```
#!/usr/bin/env python3

"""Send the contents of a directory as a MIME message."""

import os
import smtplib
# For guessing MIME type based on file name extension
import mimetypes

from argparse import ArgumentParser

from email.message import EmailMessage
from email.policy import SMTP


def main():
    parser = ArgumentParser(description="""\
Send the contents of a directory as a MIME message.
```

```
Unless the -o option is given, the email is sent by forwarding to your local
SMTP server, which then does the normal delivery process.  Your local machine
must be running an SMTP server.
""")
    parser.add_argument('-d', '--directory',
                        help="""Mail the contents of the specified directory,
                        otherwise use the current directory.  Only the regular
                        files in the directory are sent, and we don't recurse to
                        subdirectories.""")
    parser.add_argument('-o', '--output',
                        metavar='FILE',
                        help="""Print the composed message to FILE instead of
                        sending the message to the SMTP server.""")
    parser.add_argument('-s', '--sender', required=True,
                        help='The value of the From: header (required)')
    parser.add_argument('-r', '--recipient', required=True,
                        action='append', metavar='RECIPIENT',
                        default=[], dest='recipients',
                        help='A To: header value (at least one required)')
    args = parser.parse_args()
    directory = args.directory
    if not directory:
        directory = '.'
    # Create the message
    msg = EmailMessage()
    msg['Subject'] = 'Contents of directory %s' % os.path.abspath(directory)
    msg['To'] = ', '.join(args.recipients)
    msg['From'] = args.sender
    msg.preamble = 'You will not see this in a MIME-aware mail reader.\n'

    for filename in os.listdir(directory):
        path = os.path.join(directory, filename)
        if not os.path.isfile(path):
            continue
        # Guess the content type based on the file's extension.  Encoding
        # will be ignored, although we should check for simple things like
        # gzip'd or compressed files.
        ctype, encoding = mimetypes.guess_type(path)
        if ctype is None or encoding is not None:
            # No guess could be made, or the file is encoded (compressed), so
            # use a generic bag-of-bits type.
            ctype = 'application/octet-stream'
        maintype, subtype = ctype.split('/', 1)
        with open(path, 'rb') as fp:
            msg.add_attachment(fp.read(),
                               maintype=maintype,
                               subtype=subtype,
                               filename=filename)
    # Now send or store the message
    if args.output:
        with open(args.output, 'wb') as fp:
            fp.write(msg.as_bytes(policy=SMTP))
    else:
        with smtplib.SMTP('localhost') as s:
            s.send_message(msg)
```

```
if __name__ == '__main__':
    main()
```

以下是如何将MIME消息解压缩到文件目录中的示例：

```
#!/usr/bin/env python3

"""Unpack a MIME message into a directory of files."""

import os
import email
import mimetypes

from email.policy import default

from argparse import ArgumentParser


def main():
    parser = ArgumentParser(description="""\
Unpack a MIME message into a directory of files.
""")
    parser.add_argument('-d', '--directory', required=True,
                        help="""Unpack the MIME message into the named
                        directory, which will be created if it doesn't already
                        exist.""")
    parser.add_argument('msgfile')
    args = parser.parse_args()

    with open(args.msgfile, 'rb') as fp:
        msg = email.message_from_binary_file(fp, policy=default)

    try:
        os.mkdir(args.directory)
    except FileExistsError:
        pass

    counter = 1
    for part in msg.walk():
        # multipart/* are just containers
        if part.get_content_maintype() == 'multipart':
            continue
        # Applications should really sanitize the given filename so that an
        # email message can't be used to overwrite important files
        filename = part.get_filename()
        if not filename:
            ext = mimetypes.guess_extension(part.get_content_type())
            if not ext:
                # Use a generic bag-of-bits extension
                ext = '.bin'
            filename = 'part-%03d%s' % (counter, ext)
        counter += 1
        with open(os.path.join(args.directory, filename), 'wb') as fp:
            fp.write(part.get_payload(decode=True))
```

```
if __name__ == '__main__':
    main()
```

以下是如何使用替代纯文本版本创建HTML消息的示例。为了让事情更有趣，我们在html部分中包含了一个相关图像，并且保存了我们要发送到磁盘的副本以及发送它。

```
#!/usr/bin/env python3

import smtplib

from email.message import EmailMessage
from email.headerregistry import Address
from email.utils import make_msgid

# Create the base text message.
msg = EmailMessage()
msg['Subject'] = "Ayons asperges pour le déjeuner"
msg['From'] = Address("Pepé Le Pew", "pepe", "example.com")
msg['To'] = (Address("Penelope Pussycat", "penelope", "example.com"),
             Address("Fabrette Pussycat", "fabrette", "example.com"))
msg.set_content("""\
Salut!

Cela ressemble à un excellent recipie[1] déjeuner.

[1] http://www.yummly.com/recipe/Roasted-Asparagus-Epicurious-203718

--Pepé
""")

# Add the html version.  This converts the message into a multipart/alternative
# container, with the original text message as the first part and the new html
# message as the second part.
asparagus_cid = make_msgid()
msg.add_alternative("""\
<html>
  <head></head>
  <body>
    <p>Salut!</p>
    <p>Cela ressemble à un excellent
        <a href="http://www.yummly.com/recipe/Roasted-Asparagus-Epicurious-203718">
            recipie
        </a> déjeuner.
    </p>
    <img src="cid:{asparagus_cid}" />
  </body>
</html>
""".format(asparagus_cid=asparagus_cid[1:-1]), subtype='html')
# note that we needed to peel the <> off the msgid for use in the html.

# Now add the related image to the html part.
with open("roasted-asparagus.jpg", 'rb') as img:
    msg.get_payload()[1].add_related(img.read(), 'image', 'jpeg',
                                     cid=asparagus_cid)

# Make a local copy of what we are going to send.
with open('outgoing.msg', 'wb') as f:
```

```
    f.write(bytes(msg))

# Send the message via local SMTP server.
with smtplib.SMTP('localhost') as s:
    s.send_message(msg)
```

如果我们发送了最后一个例子的消息，下面是我们可以处理它的一种方法：

```
import os
import sys
import tempfile
import mimetypes
import webbrowser

# Import the email modules we'll need
from email import policy
from email.parser import BytesParser

# An imaginary module that would make this work and be safe.
from imaginary import magic_html_parser

# In a real program you'd get the filename from the arguments.
with open('outgoing.msg', 'rb') as fp:
    msg = BytesParser(policy=policy.default).parse(fp)

# Now the header items can be accessed as a dictionary, and any non-ASCII will
# be converted to unicode:
print('To:', msg['to'])
print('From:', msg['from'])
print('Subject:', msg['subject'])

# If we want to print a preview of the message content, we can extract whatever
# the least formatted payload is and print the first three lines.  Of course,
# if the message has no plain text part printing the first three lines of html
# is probably useless, but this is just a conceptual example.
simplest = msg.get_body(preferencelist=('plain', 'html'))
print()
print(''.join(simplest.get_content().splitlines(keepends=True)[:3]))

ans = input("View full message?")
if ans.lower()[0] == 'n':
    sys.exit()

# We can extract the richest alternative in order to display it:
richest = msg.get_body()
partfiles = {}
if richest['content-type'].maintype == 'text':
    if richest['content-type'].subtype == 'plain':
        for line in richest.get_content().splitlines():
            print(line)
        sys.exit()
    elif richest['content-type'].subtype == 'html':
        body = richest
    else:
        print("Don't know how to display {}".format(richest.get_content_type()))
        sys.exit()
elif richest['content-type'].content_type == 'multipart/related':
```

```
    body = richest.get_body(preferencelist=('html'))
    for part in richest.iter_attachments():
        fn = part.get_filename()
        if fn:
            extension = os.path.splitext(part.get_filename())[1]
        else:
            extension = mimetypes.guess_extension(part.get_content_type())
        with tempfile.NamedTemporaryFile(suffix=extension, delete=False) as f:
            f.write(part.get_content())
            # again strip the <> to go from email form of cid to html form.
            partfiles[part['content-id'][1:-1]] = f.name
else:
    print("Don't know how to display {}".format(richest.get_content_type()))
    sys.exit()
with tempfile.NamedTemporaryFile(mode='w', delete=False) as f:
    # The magic_html_parser has to rewrite the href="cid:..." attributes to
    # point to the filenames in partfiles.  It also has to do a safety-sanitize
    # of the html.  It could be written using html.parser.
    f.write(magic_html_parser(body.get_content(), partfiles))
webbrowser.open(f.name)
os.remove(f.name)
for fn in partfiles.values():
    os.remove(fn)

# Of course, there are lots of email messages that could break this simple
# minded program, but it will handle the most common ones.
```

截至提示时，上面的输出是：

```
To: Penelope Pussycat <penelope@example.com>, Fabrette Pussycat <fabrette@example.com>
From: Pepé Le Pew <pepe@example.com>
Subject: Ayons asperges pour le déjeuner

Salut!

Cela ressemble à un excellent recipie[1] déjeuner.
```

脚注

[1] 感谢Matthew Dixon Cowles提供的原创灵感和例子。

19.1.9。email.message.Message：使用compat32API表示电子邮件

该Message班是非常相似的 EmailMessage类，而不受类添加的方法，并与其他一些方法的默认行为是稍有不同。我们还在这里记录了一些方法，尽管EmailMessage该类支持，但除非您正在处理遗留代码，否则不推荐这些方法 。

两类的哲学和结构是相同的。

本文档描述了默认（for Message ）策略下的行为Compat32。如果您打算使用其他政策，则应该使用该EmailMessage班级。

电子邮件消息由*标题*和*有效内容组成*。标题必须是 **RFC 5233**样式名称和值，其中字段名称和值由冒号分隔。冒号不是字段名称或字段值的一部分。有效载荷可以是简单的文本消息或二进制对象，或者是具有它们自己的一组头标和它们自己的有效载荷的子消息的结构化序列。后一种有效载荷由具有MIME类型的消息（例如 *multipart / *或 message / rfc822 ）*表示*。

由一个Message对象提供的概念模型是一个有序的标题字典，它具有额外的方法来访问头文件中的专用信息，访问有效负载，生成消息的序列化版本，以及递归地遍历对象树。请注意，支持重复标题，但必须使用特殊方法才能访问它们。

所述Message伪字典由标题名称，它必须是ASCII值索引。字典的值是应该只包含ASCII字符的字符串; 对于非ASCII输入有一些特殊处理，但并不总是产生正确的结果。标题以保存案例的形式存储并返回，但字段名称不区分大小写。也可能有一个信封头，也称为*Unix-From*头或From_头。对于MIME容器文档（例如*multipart / *和message / rfc822* ），*有效*内容是字符串或字节（对于简单消息对象或Message对象 列表 ）。

这里是这个Message类的方法：

class email.message.Message（*policy = compat32* ）

如果指定*策略*（它必须是policy 类的一个实例），请使用它指定的规则来更新和序列化消息的表示形式。如果未设置*策略*，请使用compat32策略，该策略与Python 3.2版本的电子邮件包保持向后兼容。欲了解更多信息，请参阅 policy文档。

*改变在3.3版本：*该*政策*加入关键字参数。

as_string（*unixfrom = False*，*maxheaderlen = 0*，*policy = None* ）

将整个消息以字符串的形式展平。当可选的*unixfrom* 为true时，信封头将包含在返回的字符串中。 *unixfrom*默认为False。出于向后兼容性的原因， *maxheaderlen*默认为0，所以如果你想要一个不同的值，你必须显式地覆盖它（策略中为*max_line_length*指定的值将被这个方法忽略）。该*策略*的参数可被用于覆盖从消息实例中获得的默认策略。这可以用来控制该方法生成的一些格式，因为指定的*策略*将被传递给该方法Generator。

Message如果需要填充默认值以完成对字符串的转换（例如，可能会生成或修改MIME边界），则消除该消息可能会触发更改。

请注意，此方法是为了方便而提供的，可能并不总是按照您想要的方式格式化消息。例如，默认情况下，它不会执行From以unix mbox格式所需的行开头的行。为了更灵活，实例化一个 Generator实例并flatten()直接使用它的 方法。例如：

```
from io import StringIO
from email.generator import Generator
fp = StringIO()
g = Generator(fp, mangle_from_=True, maxheaderlen=60)
g.flatten(msg)
text = fp.getvalue()
```

如果消息对象包含未根据RFC标准进行编码的二进制数据，则不符合规范的数据将被unicode“未知字符”代码点替换。（另见as_bytes()和 BytesGenerator。）

*在版本3.4中更改：*添加了*策略*关键字参数。

__str__()

相当于as_string()。允许str(msg)生成包含格式化消息的字符串。

as_bytes(*unixfrom=False*，*policy=None*)

将整个消息作为字节对象返回展平。当可选的 *unixfrom*为true时，信封头将包含在返回的字符串中。 *unixfrom*默认为False。该*策略*的参数可被用于覆盖从消息实例中获得的默认策略。这可以用来控制该方法生成的一些格式，因为指定的*策略*将被传递给该方法 BytesGenerator。

Message如果需要填充默认值以完成对字符串的转换（例如，可能会生成或修改MIME边界），则消除该消息可能会触发更改。

请注意，此方法是为了方便而提供的，可能并不总是按照您想要的方式格式化消息。例如，默认情况下，它不会执行From以unix mbox格式所需的行开头的行。为了更灵活，实例化一个 BytesGenerator实例并flatten()直接使用它的 方法。例如：

```
from io import BytesIO
from email.generator import BytesGenerator
fp = BytesIO()
g = BytesGenerator(fp, mangle_from_=True, maxheaderlen=60)
g.flatten(msg)
text = fp.getvalue()
```

3.4版新增功能

__bytes__()

相当于as_bytes()。允许bytes(msg)生成包含格式化消息的字节对象。

3.4版新增功能

is_multipart()

返回True如果消息的有效载荷子的列表Message对象，否则返回False。当is_multipart()返回时False，有效负载应该是一个字符串对象（可能是CTE编码的二进制有效负载）（注意，is_multipart()返回True并不一定意味

着“msg.get_content_maintype（ ）==‘multipart’”将返回True。例如，is_multipart将返回True当 Message是类型时）message/rfc822。

set_unixfrom（*unixfrom* ）

将消息的信封头设置为*unixfrom*，它应该是一个字符串。

get_unixfrom（ ）

返回消息的信封头。默认为None如果信封头从未设置。

attach（*有效载荷*）

将给定的*有效负载*添加到当前有效负载，该有效负载必须是调用前的对象None列表Message。调用之后，有效负载将始终是Message对象列表。如果你想将有效负载设置为标量对象（例如字符串），请set_payload()改为使用 。

这是一种传统方法。在 EmailMessage类上它的功能被替换set_content()为相关的make和add方法。

get_payload（*i = None*，*decode = False* ）

返回当前的有效载荷，这将是一个列表 Message，当对象is_multipart()是True时，或者一个字符串is_multipart()是False。如果有效载荷是一个列表，并且您改变了列表对象，那么您可以修改消息的有效载荷。

使用可选参数*i*，get_payload()将返回有效负载的第*i*个元素，如果is_multipart()是从零开始计数True。一个IndexError如果将提高*我*是大于或等于在所述有效载荷的项目数小于0或。如果有效负载是一个字符串（即 is_multipart()是False ）并且*我*给出，一个TypeError上升。

根据*Content-Transfer-Encoding* 标头，可选*解码*是指示是否应解码有效载荷的标志。当和消息不是多部分时，如果该头部的值为或，则有效载荷将被解码。如果使用其他编码，或*Content-Transfer-Encoding* 标头丢失，则有效负载将按原样返回（未解码）。在所有情况下，返回值都是二进制数据。如果消息是多部分并且*解码*标记是，则返回。如果有效负载是base64，并且它没有完全形成（缺少填充，base64字母以外的字符），则将在消息的缺陷属性中添加适当的缺陷（ True quoted-printable base64 True None InvalidBase64PaddingDefect 或者 InvalidBase64CharactersDefect，分别）。

当*解码*为False（默认）时，主体将作为字符串返回，而不解码*Content-Transfer-Encoding*。但是，对于8位的*Content-Transfer-Encoding*，尝试使用错误处理程序使用*Content-Type*标头charset指定 的原始字节进行解码。如果未指定，或者电子邮件包未识别给定，则使用默认的ASCII字符集对正文进行解码。 replace charset charset

这是一种传统方法。在 EmailMessage 类上它的功能被替换为 get_content() 和 iter_parts()。

set_payload（*有效载荷*，*字符集=无* ）

将整个消息对象的有效载荷设置为*有效载荷*。确保有效载荷不变量是客户的责任。可选*字符集*设置消息的默认字符集; set_charset()详情请参阅。

这是一种传统方法。在 EmailMessage类的功能所取代set_content()。

set_charset（*charset*）

将有效负载的字符集设置为*charset*，它可以是 Charset 实例（请参阅email.charset），命名字符集的字符串或None。如果它是一个字符串，它将被转换为一个Charset实例。如果*charset* 是None，则该charset参数将从*Content-Type*头中删除（该消息不会被修改）。其他任何东西都会生成一个TypeError。

如果没有现有的*MIME-Version*标头，则会添加一个。如果没有现有的*Content-Type*标头，则会添加一个值为*text / plain*的值。*Content-Type*头是否 已经存在，其charset 参数将被设置为*charset.output_charset*。如果 *charset.input_charset* 和*charset.output_charset*不同，负载将被重新编码到*output_charset*。如果没有现有的*Content-Transfer-Encoding*头部，则有效载荷将在需要时使用指定的进行传输编码Charset，并且将添加具有适当值的头部。如果一个*Content-Transfer-Encoding*头已经存在，假定已经使用该*Content-Transfer-Encoding*正确地编码了有效载荷，并且没有被修改。

这是一种传统方法。在 EmailMessage类上，它的功能被方法的*charset*参数 替代email.emailmessage.EmailMessage.set_content()。

get_charset（）

返回Charset与消息有效负载相关的实例。

这是一种传统方法。在 EmailMessage课堂上它总是返回 None。

以下方法实现了一个用于访问消息的类似映射的接口 **RFC 2822**标头。请注意，这些方法和普通映射（即字典）界面之间存在一些语义差异。例如，在字典中没有重复的键，但这里可能有重复的消息标题。另外，在字典中，对返回的键没有保证的顺序keys()，但是在Message对象中，标题总是按照它们在原始消息中出现的顺序返回，或者在稍后添加到消息中。任何删除然后重新添加的标题总是附加到标题列表的末尾。

这些语义差异是有意的，并且偏向于最大的方便。

请注意，在所有情况下，消息中出现的任何信封标题都不包含在映射界面中。

在从字节生成的模型中，任何（与RFC相违背的）包含非ASCII字节的头值在通过此接口检索时将被表示为Header具有*未知8位字*符集的对象。

__len__（）

返回标题的总数，包括重复项。

__contains__（*名字*）

如果消息对象具有名为*name*的字段，则返回true 。匹配不区分大小写，*名称*不应包含尾部冒号。用于in操作员，例如：

```
if 'message-id' in myMessage:
   print('Message-ID:', myMessage['message-id'])
```

__getitem__（*名字*）

返回指定标题字段的值。 *名称*不应包含冒号字段分隔符。如果标题丢失，None则返回; a KeyError永远不会升起。

请注意，如果命名字段在消息头中多次出现，那么将返回那些字段值中的哪些值是未定义的。使用该get_all()方法获取所有现存命名标题的值。

__setitem__ (*name* , *val*)

将字段添加到带有字段*名称*和值*val*的消息中。该字段附加到消息现有字段的末尾。

请注意，这并*不能*覆盖或删除任何现有的头名称相同。如果要确保新标题是带有字段名称*名称*的消息中唯一存在的标题，请先删除该字段，例如：

```
del msg['subject']
msg['subject'] = 'Python roolz!'
```

__delitem__ (*名字*)

从消息的标题中删除所有出现的具有名称*名称*的字段。如果名称字段不存在于标题中，则不会引发异常。

keys ( )

返回所有消息标题字段名称的列表。

values ( )

返回所有消息字段值的列表。

items ( )

返回包含所有消息的字段标题和值的2元组列表。

get (*名称* , *failobj =无*)

返回指定标题字段的值。这与__getitem__()除了如果命名标题丢失（缺省为）时返回可选*failobj*相同 None。

以下是一些其他有用的方法：

get_all (*名称* , *failobj =无*)

返回名为*name*的字段的所有值的列表。如果消息中没有这样的标题，则返回*failobj*（默认为 None）。

add_header (*_name* , *_value* , ** *_params*)

扩展标题设置。__setitem__() 除了可以提供额外的头部参数作为关键字参数之外，此方法类似。 *_name*是要添加的标题字段，而*_value*是标题的 *主要*值。

对于关键字参数字典*_params*中的每个项目，都将该关键字作为参数名称，并将下划线转换为破折号（因为破折号在Python标识符中是非法的）。通常，该参数将被添加，key="value"除非该值为None，在这种情况下，只有该键将被添加。如果该值包含非ASCII字符，则可将其指定为格式中的三元组 ，其中是指定用于对值进行编码的字符集的字符串，通常可以设置为空字符串（请参阅(CHARSET, LANGUAGE, VALUE) CHARSET LANGUAGE None**RFC 2231**用于其他可能性），并且VALUE是包含非ASCII码点的字符串值。如果三元组没有传递，并且该值包含非ASCII字符，则会自动编入英寸**RFC 2231**格式使用CHARSET 的utf-8和LANGUAGE的None。

这是一个例子：

```
msg.add_header('Content-Disposition', 'attachment', filename='bud.gif')
```

这将添加一个看起来像的标题

```
Content-Disposition: attachment; filename="bud.gif"
```

非ASCII字符的示例：

```
msg.add_header('Content-Disposition', 'attachment',
               filename=('iso-8859-1', '', 'Fußballer.ppt'))
```

哪产生

```
Content-Disposition: attachment; filename*="iso-8859-1''Fu%DFballer.ppt"
```

replace_header (*_name* , *_value*)

替换标题。替换消息中找到的与*_name*匹配的第一个标头，保留标头顺序和字段名称大小写。如果找不到匹配的标题，KeyError则会引发a。

get_content_type ( )

返回消息的内容类型。将返回的字符串强制为*maintype* / *subtype*形式的*小写字母*。如果消息中没有 *Content-Type*头，则get_default_type()返回给定的默认类型。由于根据 **RFC 2045**，消息始终有一个默认类型，get_content_type() 将始终返回一个值。

RFC 2045将消息的默认类型定义为 *text* / *plain* ， 除非它出现在 *multipart* / *digest*容器中，在这种情况下，它将是 *message* / *rfc822*。如果 *Content-Type*头的类型规范无效， **RFC 2045**规定默认类型为 *text* / *plain*。

get_content_maintype ( )

返回消息的主要内容类型。这是 返回的字符串的*maintype*部分get_content_type()。

get_content_subtype ( )

返回消息的子内容类型。这是 返回的字符串的*子类型*部分get_content_type()。

get_default_type ( )

返回默认的内容类型。除了作为*多部分*/*摘要*容器的子*部分的*消息之外， 大多数消息具有默认的*文本*/*纯文本*内容类型。这样的子部分有一个默认的内容类型的*消息*/ *rfc822*。

set_default_type (*ctype*)

设置默认的内容类型。 *ctype*应该是 *text* / *plain*或*message* / *rfc822* ，尽管这不是强制执行的。默认内容类型不存储在 *Content-Type*标头中。

get_params (*failobj* = *None* , *header* ='content-type' , *unquote* = *True*)

以列表形式返回消息的*Content-Type*参数。返回列表的元素是键/值对的2元组，在'='符号上分割。左侧'='是键，右侧是值。如果'='参数中没有任何符号，则值为空字符串，否则该值get_param()如上所述，如果可选*unquote*为 True （默认值），则为未引用。

如果没有*Content-Type*标头，可选的*failobj*是要返回的对象 。可选*标题*是要搜索的标题而不是*Content-Type*。

这是一种传统方法。在 EmailMessage类中，它的功能由标头访问方法返回的各个标头对象的*params*属性替代。

get_param (*param* , *failobj* = *None* , *header* ='content-type' , *unquote* = *True*)

以字符串形式返回*Content-Type*头的参数 *param*的值。如果消息没有*Content-Type* 头或者没有这样的参数，则返回*failobj* (默认为None) 。

可选*标题 (* 如果给出) 指定要使用的消息标题而不是 *Content-Type*。

参数键始终不区分大小写。返回值可以是一个字符串，也可以是一个三元组 (如果参数是) **RFC 2231** 编码。当它是一个三元组时，值的元素就是这种形式 。请注意这两个和 可能 ，在这种情况下，你应该考虑 在被编码字符集。你通常可以忽略 。(CHARSET, LANGUAGE, VALUE) CHARSET LANGUAGE None VALUE us-ascii LANGUAGE

如果你的应用程序不关心参数是否被编码为in **RFC 2231中** ，您可以通过调用来email.utils.collapse_rfc2231_value()传递参数值 ，从中传入返回值get_param()。当值是一个元组时，这将返回一个适当解码的Unicode字符串，或者如果原始字符串不是，则返回原始字符串。例如：

```
rawparam = msg.get_param('foo')
param = email.utils.collapse_rfc2231_value(rawparam)
```

在任何情况下，VALUE除非设置*unquote*，否则参数值 (返回的字符串或3元组中的 项目) 始终未加引号False。

这是一种传统方法。在 EmailMessage类中，它的功能由标头访问方法返回的各个标头对象的*params*属性替代。

set_param (*param* , *value* , *header* ='Content-Type' , *requote* = *True* , *charset* = *None* , *language* ='' , *replace* = *False*)

在*Content-Type*标题中设置一个参数。如果参数已经存在于标题中，它的值将被替换为 *值*。如果*Content-Type*头还没有被定义为这个消息，它将被设置为*text / plain*，并且新的参数值将被附加到每个**RFC 2045**。

可选*标题*指定*Content-Type*的替代标题， 除非可选的重新*标记*为False (默认为True) ，否则将根据需要引用所有参数。

如果指定了可选的*字符集*，参数将按照。编码**RFC 2231**。可选*语言*指定RFC 2231语言，默认为空字符串。这两个*字符集*和*语言* 应弦。

如果*替换*为False (默认) ，则标题将移动到标题列表的末尾。如果*更换*就是True ，头部将在地方进行更新。

在版本3.4中更改： replace添加了关键字。

del_param (*param* , *header* ='content-type' , *requote* = *True*)

从*Content-Type* 标题中完全删除给定的参数。标题将在没有参数或其值的情况下被重写。除非所有的值将被引用作为必要*重新报价*为 False（默认为True）。可选*标题*指定*Content-Type*的替代方案。

set_type（*type*，*header ='Content-Type'*，*requote = True* ）

设置*Content-Type* 标题的主类型和子*类型*。*类型*必须是*maintype / subtype*形式的字符串，否则ValueError会引发a。

此方法替换*Content-Type*标头，保留所有参数。如果重新编号是，则会*保留*False现有标题的引用，否则参数将被引用（默认值）。

可以在*标题*参数中指定一个替代标*头*。当 *Content-Type*头被设置时， 还添加了*MIME-Version*头。

这是一种传统方法。在 EmailMessage类上它的功能被替换为make_和add_方法。

get_filename（*failobj =无*）

返回消息filename的*Content-Disposition*头的参数 值。如果标题没有filename参数，则此方法回退到查找*Content-Type*标题name上的参数。如果两者都未找到，或者标题丢失，则返回*failobj*。返回的字符串将始终不加引号 。 email.utils.unquote()

get_boundary（*failobj =无*）

返回消息boundary的*Content-Type*头的参数 值，或者如果头丢失或没有参数，则*返回failobj*boundary。返回的字符串将始终不加引号email.utils.unquote()。

set_boundary（*边界*）

将*Content-Type*标头的boundary参数设置为 *边界*。 将在必要时总是引用*边界*。如果消息对象没有*Content-Type*头部，则引发A.set_boundary() HeaderParseError

请注意，使用这种方法比删除旧的稍微不同的 *内容类型*头并添加一个新通过的新的边界add_header() ，因为set_boundary()保留的顺序*的Content-Type*标头的列表标题。但是，它并*没有*保留这可能是存在于原始任何连续行*的Content-Type*首部。

get_content_charset（*failobj =无*）

返回*Content-Type*头部的charset参数，强制为小写。如果没有*Content-Type*标头，或者该标头没有参数，则返回*failobj*。 charset

请注意，此方法与get_charset()返回Charset消息正文的默认编码的 实例不同。

get_charsets（*failobj =无*）

返回包含消息中字符集名称的列表。如果消息是*多部分*，那么列表将包含有效载荷中每个子部分的一个元素，否则它将是长度为1的列表。

列表中的每个项目都是一个字符串，它是表示子部分charset的*Content-Type*标题中的参数值 。但是，如果子部分没有 *Content-Type*标头，没有charset参数，或者不是*文本*主MIME类型，则返回列表中的该项目将为*failobj*。

get_content_disposition（ ）

返回消息的*Content-Disposition*头的小写值（不带参数），如果它有一个或者None。此方法的可能值为*内联*，*附件*或None 后续消息**RFC 2183**。

3.5版本中的新功能。

walk（）

该walk()方法是一个通用的生成器，可用于以深度优先遍历顺序遍历消息对象树的所有部分和子部分。您通常会将其walk()用作for循环中的迭代器;每次迭代都会返回下一个子部分。

以下是一个打印多部分消息结构的每个部分的MIME类型的示例：

```
>>> for part in msg.walk():
...     print(part.get_content_type())
multipart/report
text/plain
message/delivery-status
text/plain
text/plain
message/rfc822
text/plain
```

walk迭代is_multipart()返回的任何部分的子部分 True，即使 可能会返回。我们可以在我们的例子中看到这一点，通过使用调试帮助函数：msg.get_content_maintype() == 'multipart'False_structure

```
>>> for part in msg.walk():
...     print(part.get_content_maintype() == 'multipart',
...           part.is_multipart())
True True
False False
False True
False False
False False
False True
False False
>>> _structure(msg)
multipart/report
    text/plain
message/delivery-status
    text/plain
    text/plain
message/rfc822
    text/plain
```

这里的message部分不是multiparts，但它们确实包含子部分。is_multipart()返回True并walk下降到子部分。

Message 对象还可以选择包含两个实例属性，这些属性可以在生成MIME消息的纯文本时使用。

preamble

MIME文档的格式允许标题后面的空白行和第一个多段边界字符串之间的文本。通常，这个文本在MIME感知的邮件阅读器中是不可见的，因为它不属于标准的MIME装甲。但是，查看邮件的原始文本时，或者在非MIME的阅读器中查看邮件时，可以看到该文本。

在*序言*属性包含MIME文件这导致额外的装甲文本。当在Parser标题后面但在第一个边界字符串之前发现一些文本时，它会将该文本分配给该消息的*前导*属性。当 Generator写出MIME消息的纯文本表示，并且它发现消息具有*前导*属性时，它将在头和第一个边界之间的区域中写入该文本。请参阅email.parser并 email.generator了解详情。

请注意，如果消息对象没有前导码，则*前导码*属性将为None。

epilogue

的*尾声*属性作用方式与相同的*前导码*属性，不同之处在于它包含的最后一个边界和消息的结束之间出现的文本。

您不需要将尾声设置为空字符串，以便 Generator在文件末尾打印换行符。

defects

该*缺陷*属性包含的所有解析此消息时发现的问题的列表。请参阅有关email.errors可能的解析缺陷的详细说明。

10年1月19日。email.mime：从头开始创建电子邮件和MIME对象

源代码： Lib / email / mime /

该模块是legacy（Compat32）电子邮件API的一部分。其功能部分取代contentmanager了新的API，但在某些应用程序中，这些类可能仍然有用，即使在非传统代码中也是如此。

通常，通过将文件或某些文本传递给解析器来获取消息对象结构，解析器解析文本并返回根消息对象。但是，您也可以从头开始构建完整的消息结构，或者Message手动构建单个 对象。实际上，您也可以使用现有的结构并添加新Message对象，将它们移动等。这为切割和切割MIME消息提供了一个非常方便的界面。

您可以通过创建Message 实例，手动添加附件和所有适当的标题来创建新的对象结构。尽管如此，该email包提供了一些方便的子类来使事情变得更简单。

这里是类：

class email.mime.base.MIMEBase（*_maintype*，*_subtype*，*，policy = compat32*，*** _params* ）

模块： email.mime.base

这是所有MIME特定子类的基类 Message。通常你不会创建特定的实例MIMEBase，尽管你可以。 MIMEBase 主要作为更具体的MIME感知子类的便利基类提供。

*_maintype*是*Content-Type*主要类型（例如*文本*或*图像*），*_subtype*是*Content-Type*次要类型（例如*plain*或*gif*）。 *_params*是一个参数键/值字典，并直接传递给Message.add_header。

如果*政策*指定，（默认为 compat32政策），它会被传递给 Message。

该MIMEBase班总是增加了一个*内容类型*头（基于*_maintype*，*_subtype*和*_params*）和*MIME-版本*头（始终设置为1.0）。

*在版本3.6中更改：*添加了仅*策略*关键字参数。

*类*email.mime.nonmultipart.MIMENonMultipart

模块： email.mime.nonmultipart

它的一个子MIMEBase类，是不是*多部分的*MIME消息的中间基类。这个类的主要目的是防止使用attach()只对*多部分*消息有意义的 方法 。如果attach() 被调用，MultipartConversionError则会引发异常。

class email.mime.multipart.MIMEMultipart（*_subtype ='mixed'*，*boundary = None*，*_subparts = None*，*，policy = compat32*，*** _ params* ）

模块： email.mime.multipart

它的一个子MIMEBase类是*多部分* MIME消息的中间基类。可选*_subtype* 默认为*混合*，但可用于指定消息的子类型。甲*内容类型*的报头*的multipart / _subtype* 将被添加到该消息对象。一个*MIME-版*头也将增加。

可选*边界*是多部分边界字符串。当None（缺省值）时，边界在需要时计算（例如，当消息被序列化时）。

*_subparts*是有效负载的一系列初始子部分。必须可以将此序列转换为列表。您始终可以使用该Message.attach方法将新的子部件附加到消息中。

可选*策略*参数默认为compat32。

*Content-Type*头部的其他参数取自关键字参数，或传递到*_params*参数（它是一个关键字字典）。

*在版本3.6中更改：*添加了仅*策略*关键字参数。

*类*email.mime.application.MIMEApplication（*_data*，*_subtype = '八位字节流'*，*_encoder = email.encoders.encode_base64*，*，政策= compat32*，*** _ PARAMS* ）

模块： email.mime.application

的一个子类MIMENonMultipart中， MIMEApplication类被用来表示主要类型的MIME消息对象*应用*。 *_data*是一个包含原始字节数据的字符串。可选*_subtype*指定MIME子类型，并默认为 *八位字节流*。

可选的*_encoder*是一个可调用的函数（即函数），它将执行传输数据的实际编码。这个可调用的参数有一个参数，这就是MIMEApplication实例。它应该使用 get_payload() 并将 set_payload() 有效载荷更改为编码形式。它还应该根据需要将任何*Content-Transfer-Encoding*或其他标题添加到消息对象。默认编码是base64。查看 email.encoders模块以获取内置编码器的列表。

可选*策略*参数默认为compat32。

*_params*直接传递给基类构造函数。

*在版本3.6中更改：*添加了仅*策略*关键字参数。

*类*email.mime.audio.MIMEAudio（*_audiodata*，*_subtype = None*，*_encoder = email.encoders.encode_base64*，*，policy = compat32*，*** _ params* ）

模块： email.mime.audio

的一个子类MIMENonMultipart中， MIMEAudio类用于创建主要类型的MIME消息对象 *的音频*。*_audiodata*是一个包含原始音频数据的字符串。如果这个数据可以被标准Python模块解码sndhdr，那么子类型将自动包含在*Content-Type*头中。否则，您可以通过*_subtype* 参数显式指定音频子类型。如果无法猜测次要类型，并且未提供*_subtype*，则会TypeError提出。

可选的*_encoder*是一个可调用的函数（即函数），它将执行用于传输的音频数据的实际编码。这个可调用的参数有一个参数，这就是MIMEAudio实例。它应该使用 get_payload() 并将 set_payload() 有效载荷更改为编码形式。它还应该根据需要将任何*Content-Transfer-*

*Encoding*或其他标题添加到消息对象。默认编码是base64。查看 email.encoders模块以获取内置编码器的列表。

可选*策略*参数默认为compat32。

*_params*直接传递给基类构造函数。

*在版本3.6中更改：*添加了仅*策略*关键字参数。

class email.mime.image.MIMEImage（*_imagedata*，*_subtype = None*，*_encoder = email.encoders.encode_base64*，*，policy = compat32*，*** _ params* ）

模块： email.mime.image

的一个子类MIMENonMultipart中， MIMEImage类用于创建主要类型的MIME消息对象 *图像*。*_imagedata*是一个包含原始图像数据的字符串。如果这个数据可以被标准Python模块解码imghdr，那么子类型将自动包含在*Content-Type*头中。否则，您可以通过*_subtype* 参数显式指定图像子类型。如果无法猜测次要类型，并且未提供*_subtype*，则会TypeError提出。

可选的*_encoder*是一个可调用的函数（即函数），它将执行传输图像数据的实际编码。这个可调用的参数有一个参数，这就是MIMEImage实例。它应该使用 get_payload()并将set_payload()有效载荷更改为编码形式。它还应该根据需要将任何*Content-Transfer-Encoding*或其他标题添加到消息对象。默认编码是base64。查看 email.encoders模块以获取内置编码器的列表。

可选*策略*参数默认为compat32。

*_params*直接传递给MIMEBase 构造函数。

*在版本3.6中更改：*添加了仅*策略*关键字参数。

class email.mime.message.MIMEMessage（*_msg*，*_subtype ='rfc822'*，*，policy = compat32* ）

模块： email.mime.message

的一个子类MIMENonMultipart中， MIMEMessage类用于创建主要类型的MIME对象 *消息*。*_msg*用作有效负载，并且必须是类Message（或其子类）的实例，否则TypeError会引发a。

可选*_subtype*设置消息的子类型; 它默认为 *rfc822*。

可选*策略*参数默认为compat32。

*在版本3.6中更改：*添加了仅*策略*关键字参数。

class email.mime.text.MIMEText（*_text*，*_subtype ='plain'*，*_charset = None*，*，policy = compat32* ）

模块： email.mime.text

的一个子类MIMENonMultipart中， MIMEText类用于创建主要类型的MIME对象 *文本*。*_text*是有效负载的字符串。 *_subtype*是次要类型，默认为*plain*。 *_charset*是文本的字符集，并

作为参数传递给 MIMENonMultipart构造函数; us-ascii如果字符串仅包含ascii代码点， 则默认为默认值，否则为默认值utf-8。该*_CHARSET*参数接受字符串或一个 Charset实例。

除非显式设置*_charset*参数None ，否则创建的MIMEText对象将同时具有带参数的*Content-Type* 标 头 charset 和 *Content-Transfer-Encoding* 标 头 。 这 意 味 着 set_payload 即 使 在 set_payload 命令中传递了字符集，后续调用也不会导致编码的有效内容。您可以通过删除Content-Transfer-Encoding标题来“重置”此行为 ，之后一个set_payload调用将自动编码新的有效内容（并添加一个新的 *Content-Transfer-Encoding*标头）。

可选*策略*参数默认为compat32。

*在版本3.5中更改：*_charset也接受Charset实例。

*在版本3.6中更改：*添加了仅*策略*关键字参数。

11年1月19日。email.header：国际化标题

源代码： Lib / email / header.py

该模块是legacy（Compat32）电子邮件API的一部分。在当前API中，头文件的编码和解码由类的字典式API透明地处理EmailMessage。除了在遗留代码中使用外，此模块还可用于需要完全控制编码标头时使用的字符集的应用程序。

本节中的其余部分是该模块的原始文档。

RFC 2822是描述电子邮件格式的基本标准。它来自老年人**RFC 822**标准在大多数电子邮件仅由ASCII字符组成的时候被广泛使用。 **RFC 2822**是一个规范，其中假定电子邮件仅包含7位ASCII字符。

当然，由于电子邮件已经在全球范围内部署，它已经变得国际化，因此现在可以在电子邮件中使用特定于语言的字符集。基本标准仍然要求仅使用7位ASCII字符传输电子邮件消息，因此已经编写了大量的RFC，描述了如何将包含非ASCII字符的电子邮件编码成 **RFC 2822**兼容格式。这些RFC包括**RFC 2045**，**RFC 2046**， **RFC 2047**，和**RFC 2231**。该email软件包在其模块email.header和email.charset模块中支持这些标准。

如果要在电子邮件标题中包含非ASCII字符，请在 *Subject*或*To*字段中说明，应该使用 Header该类并将Message 对象中的字段分配给实例，Header而不是使用字符串作为标题值。 Header从email.header模块导入类。例如：

```
>>> from email.message import Message
>>> from email.header import Header
>>> msg = Message()
>>> h = Header('p\xf6stal', 'iso-8859-1')
>>> msg['Subject'] = h
>>> msg.as_string()
'Subject: =?iso-8859-1?q?p=F6stal?=\n\n'
```

请注意，我们是如何让*主题*字段包含非ASCII字符的？我们通过创建Header实例并传入字符串编码的字符集来实现此目的。当后续 Message实例变平时，*主题* 字段正确**RFC 2047**编码。支持MIME的邮件阅读器将使用嵌入的ISO-8859-1字符显示此头文件。

这里是Header课程描述：

*类*email.header.Header（*S =无*，*字符集=无*，*maxlinelen =无*，*HEADER_NAME = NONE*，*continuation_ws = ' '误差='严格'* ）

创建一个可以包含不同字符集中的字符串的符合MIME的标头。

可选*s*是初始标题值。如果None（默认），则未设置初始标题值。您可以稍后使用append()方法调用追加到标题。 *s*可能是bytesor的 一个实例str，但请参阅append()语义的文档。

可选的*字符集有*两个目的：它 与方法的*字符集*参数具有相同的含义append()。它还为append()忽略*字符集*参数的所有后续调用设置默认字符集。如果 构造函数中没有提供

charset（缺省值），则us-ascii 字符集既用作*s*的初始字符集又用作后续append()调用的默认字符集。

最大行长度可以通过*maxlinelen*明确指定。为了将第一行分割为一个较短的值（为了说明没有包含在*s*中的字段标题，例如*Subject*），传入*header_name*中字段的名称。默认*maxlinelen*是76，而对于默认值*HEADER_NAME*是None，这意味着它不考虑为一个长，分割报头的第一行。

可选*continuation_ws*必须是**符合RFC 2822**的折叠空白字符，通常是空格或硬标签字符。这个角色将被添加到延续线。 *continuation_ws* 默认为单个空格字符。

可选*错误*直接传递给append()方法。

append（*s*，*charset* = *None*，*errors* ='strict'）

将字符串*s*附加到MIME头。

可选的*字符集*，如果有的话，应该是一个Charset 实例（参见email.charset ）或字符集的名称，它将被转换为Charset实例。值None（默认值）表示使用构造函数中给出的*字符集*。

*s*可能是bytes或的一个实例str。如果它是一个实例bytes，则*charset*是该字节字符串的编码，UnicodeError如果字符串不能用该字符集解码，则会产生一个字符串。

如果*s*是一个实例str，则*charset*是一个提示，指定字符串中字符的字符集。

在任何情况下，当制作一个 **使用RFC 2822**的标头 **RFC 2047**规则中，字符串将使用charset的输出编解码器进行编码。如果字符串不能使用输出编解码器进行编码，则会引发UnicodeError。

如果*s*是一个字节字符串，则可选*错误*作为decode调用的错误参数传递。

encode（*splitchars* =';，\ *t*'，*maxlinelen* = *None*，*linesep* ='\ *n*'）

将消息头编码为符合RFC的格式，可能包装长行并将非ASCII部分封装在base64或quoted-printable编码中。

可选*splitchars*是一个包含字符的字符串，在正常的标题封装期间，字符应该被分割算法赋予额外的权重。这是非常粗略的支持**RFC 2822**的'更高级语法断点'：在行拆分过程中，首选splitchar的拆分点是首选，字符在它们出现在字符串中的顺序上是首选。空格和制表符可以包含在字符串中，以指示当其他拆分字符不出现在正在拆分的行中时，是否应该优先考虑拆分作为拆分点。Splitchars不会影响**RFC 2047**编码线。

maxlinelen，如果给定，则覆盖实例的最大行长度值。

*linesep*指定用于分隔折叠标题行的字符。它默认为Python应用程序代码（ \n ）的最有用的值，但\r\n可以指定为了生成具有符合RFC标准的行分隔符的标头。

*在版本3.2中更改：*添加了*linesep*参数。

本Header类还提供了一些方法，以支持标准的运营商和内置函数。

__str__（）

Header使用无限长的线条返回字符串的近似值。所有作品都使用指定的编码转换为unicode并适当地连接在一起。'unknown-8bit'使用'replace'错误处理程序将任何带charset的块解码为ASCII 。

*在版本3.2中*进行了*更改：*为'unknown-8bit'字符集添加了处理。

__eq__ (*其他*)
该方法允许您比较两个Header实例是否相等。

__ne__ (*其他*)
该方法允许您比较两个Header不平等的实例。

该email.header模块还提供以下便利功能。

email.header. decode_header (*header*)
在不转换字符集的情况下解码消息标题值。标题值位于*标题中*。

该函数返回一个包含头部每个解码部分的对列表。 *字符集*是用于报头的非编码的部分，否则包含已编码的字符串中指定的字符集的名称的小写字符串。(decoded_string, charset) None

这是一个例子：

```
>>> from email.header import decode_header
>>> decode_header('=?iso-8859-1?q?p=F6stal?=')
[(b'p\xf6stal', 'iso-8859-1')]
```

email.header. make_header (*decoded_seq* , *maxlinelen =无* , *HEADER_NAME = NONE* , *continuation_ws =""*)
Header从返回的成对序列中创建一个实例 decode_header()。

decode_header()接受一个标题值字符串并返回一组字符串对，其中*charset*是*字符集*的名称。(decoded_string, charset)

该函数使用这些序列中的一个并返回一个 Header实例。可选*maxlinelen* ， *信头* ，和*continuation_ws*如在Header构造函数。

12年1月19日。email.charset：表示字符集

源代码： Lib / email / charset.py

该模块是legacy（Compat32）电子邮件API的一部分。在新API中只使用别名表。

本节中的其余部分是该模块的原始文档。

该模块提供了一个Charset用于表示电子邮件中的字符集和字符集转换的类，以及用于操作此注册表的字符集注册表和一些便利方法。实例Charset在email包中的其他几个模块中使用。

从email.charset模块中导入此类。

class email.charset.Charset（*input_charset = DEFAULT_CHARSET*）

将字符集映射到其电子邮件属性。

该课程提供有关特定字符集对电子邮件的要求的信息。考虑到可用的编解码器的可用性，它还提供了在字符集之间转换的便利例程。鉴于字符集，它将尽最大努力提供有关如何以符合RFC的方式在电子邮件中使用该字符集的信息。

在电子邮件标题或正文中使用时，某些字符集必须用quoted-printable或base64编码。某些字符集必须完全转换，并且不允许在电子邮件中使用。

可选的*input_charset*如下所述; 它总是被强制为小写。在被别名标准化之后，它也被用作查找字符集的注册表以找出用于字符集的标题编码，主体编码和输出转换编解码器。例如，如果 *input_charset*是iso-8859-1，则标题和正文将使用quoted-printable进行编码，并且不需要输出转换编解码器。如果 *input_charset*是euc-jp，那么标题将用base64进行编码，不会对body进行编码，但输出文本将从euc-jp 字符集转换为iso-2022-jp字符集。

Charset 实例具有以下数据属性：

input_charset

指定的初始字符集。常用的别名被转换为他们的*官方*电子邮件名称（例如latin_1转换为 iso-8859-1）。默认为7位us-ascii。

header_encoding

如果字符集必须在电子邮件头中使用之前进行编码，则此属性将设置为Charset.QP（对于引用可打印）Charset.BASE64（对于base64编码），或Charset.SHORTEST对于最短的QP或BASE64编码。否则，它会None。

body_encoding

与*header_encoding*相同，但描述了邮件消息正文的编码，确实可能与标题编码不同。Charset.SHORTEST不允许*body_encoding*。

output_charset

一些字符集必须先转换后才能用于电子邮件标题或正文。如果*input_charset*是其中的一个，则此属性将包含将要转换为的字符集输出的名称。否则，它会None。

input_codec
: 用于将*input_charset*转换为Unicode 的Python编解码器的名称。如果不需要转换编解码器，则此属性为 None。

output_codec
: 用于将Unicode转换为*output_charset*的Python编解码器的名称 。如果不需要转换编解码器，则该属性将具有与*input_codec*相同的值。

Charset 实例还有以下方法：

get_body_encoding ( )
: 返回用于正文编码的内容传输编码。

  这要么是字符串，quoted-printable要么base64取决于所使用的编码，或者它是一个函数，在这种情况下，您应该使用一个参数调用函数，Message对象将被编码。然后该函数应该将*Content-Transfer-Encoding* 头本身设置为适当的。

  返回字符串quoted-printable ，如果*body_encoding*是QP ，返回字符串base64如果*body_encoding*是BASE64 ，并返回字符串7bit ，否则。

get_output_charset ( )
: 返回输出字符集。

  这是*output_charset*属性（如果不是）None ，否则它是*input_charset*。

header_encode (*字符串*)
: 标头编码*字符串*。

  编码类型（base64或quoted-printable）将基于 *header_encoding*属性。

header_encode_lines (*string* , *maxlengths*)
: 通过首先将*字符串*转换为字节来对其进行标头编码。

  这与header_encode()除了字符串符合*maxlengths*参数所给出的最大行长度相似之外，该参数必须是一个迭代器：从此迭代器返回的每个元素将提供下一个最大行长度。

body_encode (*字符串*)
: 对字符串*字符串*进行身体编码。

  编码类型（base64或quoted-printable）将基于 *body_encoding*属性。

本Charset类还提供了一些方法来支持标准操作和内置功能。

__str__ ( )
: 将*input_charset*作为强制转换为小写字符串的形式返回。 __repr__() 是一个别名__str__()。

__eq__ (*其他*)
: 该方法允许您比较两个Charset实例是否相等。

__ne__（*其他*）

该方法允许您比较两个Charset不平等的实例。

该email.charset模块还提供了以下功能，用于向全局字符集，别名和编解码器注册表中添加新条目：

email.charset.add_charset（*charset*，*header_enc* = *None*，*body_enc* = *None*，*output_charset* = *None* ）

将字符属性添加到全局注册表。

*字符集*是输入字符集，并且必须是字符集的规范名称。

可选*header_enc*和*body_enc*或者是Charset.QP带引号的可打印，Charset.BASE64为base64编码， Charset.SHORTEST争取在最短的引用可打印或base64编码的，或者None因为没有编码。 SHORTEST仅对*header_enc*有效 。默认值是None无编码。

可选的*output_charset*是输出应该在其中的字符集。当Charset.convert()调用方法时，转换将从输入字符集继续到Unicode，再到输出字符集。默认是以与输入相同的字符集输出。

无论*input_charset*和*output_charset*必须在模块的字符的Unicode编码解码器条目设置到编解码器的映射; 用于add_codec()添加模块不知道的编解码器。有关codecs更多信息，请参阅模块的文档。

全局字符集注册表保存在模块全局字典中 CHARSETS。

email.charset.add_alias（*别名*，*规范*）

添加一个字符集别名。 *别名*是别名，例如latin-1。 *canonical*是字符集的规范名称，例如iso-8859-1。

全局字符集别名注册表保存在模块全局字典中 ALIASES。

email.charset.add_codec（*charset*，*codecname* ）

添加一个将给定字符集中的字符映射到Unicode的编解码器。

*字符集*是*字符集*的规范名称。*codecname*是一个Python编解码器的名称，以适合的第二个参数str的 encode()方法。

13年1月19日。email.encoders：编码器

源代码： Lib / email / encoders.py

该模块是legacy（Compat32）电子邮件API的一部分。在新的API中，功能由该方法的*cte*参数提供set_content()。

本节中的其余部分是该模块的原始文档。

Message从头开始创建对象时，通常需要对通过兼容邮件服务器传输的有效负载进行编码。对于包含二进制数据的*image* / *和*text* / *类型的消息尤其如此。

该email软件包在其encoders模块中提供了一些便利的编码 。这些编码器实际使用 MIMEAudio和 MIMEImage 类构造函数提供的默认编码。所有的编码器函数只需要一个参数，即消息对象进行编码。他们通常会提取有效负载，对其进行编码，并将有效负载重置为新编码的值。他们还应该适当地设置*Content-Transfer-Encoding*标头。

请注意，这些功能对于多部分消息没有意义。它们必须应用于各个子部分，并且会 TypeError 传递一个类型为multipart的消息。

以下是提供的编码功能：

email.encoders.encode_quopri（*味精*）
: 将有效载荷编码为quoted-printable形式，并将 *Content-Transfer-Encoding*标头设置为 quoted-printable [1]。当大多数有效负载是正常可打印数据，但包含一些不可打印字符时，这是一种很好的编码方式。

email.encoders.encode_base64（*味精*）
: 将有效载荷编码为base64格式，并将 *Content-Transfer-Encoding*标头设置为base64。当您的大部分有效内容是不可打印的数据时，这是一种很好的编码方式，因为它比引用打印的形式更紧凑。base64编码的缺点是它使得文本不可读。

email.encoders.encode_7or8bit（*味精*）
: 这实际上并不修改消息的有效负载，但它确实根据有效负载数据将 *Content-Transfer-Encoding*标头设置为7bit或8bit适当。

email.encoders.encode_noop（*味精*）
: 这什么都不做; 它甚至不设置 *Content-Transfer-Encoding*标头。

脚注

[1] 请注意，编码encode_quopri()还会对数据中的所有制表符和空格字符进行编码。

14年1月19日。email.utils：杂项工具

源代码： Lib / email / utils.py

email.utils 模块中提供了一些有用的实用程序：

email.utils.**localtime**(*dt =无*)
将当地时间作为知晓的日期时间对象返回。如果没有参数调用，则返回当前时间。否则，*dt*参数应该是一个 datetime实例，并根据系统时区数据库将其转换为本地时区。如果*DT*是天真的（即dt.tzinfo是None），它被认为是在本地时间。在这种情况下，*isdst*的正值或零值会导致localtime最初假定夏令时（例如，夏令时）对于指定时间是或不是（分别）有效。为负值 *isdst*导致localtime试图神圣的夏季时间是否在指定的时间效应。

3.3版本的新功能

email.utils.**make_msgid**(*idstring = None*，*domain = None*)
返回适合于的字符串 *符合***RFC 2822**的 *Message-ID*标头。可选的 *idstring*（如果给出）是用于加强消息ID唯一性的字符串。如果给定可选*域*，则在'@'之后提供msgid的部分。默认是本地主机名。通常不需要重写此默认值，但在某些情况下可能会有用，例如构建跨多个主机使用一致域名的分布式系统。

*在版本3.2中更改：*添加了*域*关键字。

其余功能是legacy（Compat32）电子邮件API的一部分。没有必要直接将这些与新的API一起使用，因为它们提供的解析和格式化由新API的标题解析机制自动完成。

email.utils.**quote**(*str*)
将*str中的*反斜杠替换为两个反斜杠的新字符串，并将双引号替换为反斜杠双引号。

email.utils.**unquote**(*str*)
返回一个不带*引号*的*str*的新字符串。如果*str*结束并以双引号开头，则会被剥离。同样，如果*str*结束并以尖括号开始，它们将被剥离。

email.utils.**parseaddr**(*地址*)
解析地址 - 应该是某个包含地址的字段（例如“*收件人*”或“*抄送*”）的值，并将其解析为其*真实姓名*和 *电子邮件地址*部分。返回该信息的元组，除非解析失败，在这种情况下返回2元组。('', '')

email.utils.**formataddr**(*pair*，*charset ='utf-8'*)
与此相反parseaddr()，这需要表单的2元组并返回适合于*To*或 *Cc*标题的字符串值。如果*pair*的第一个元素为false，则第二个元素将不加修改地返回。(realname, email_address)

可选*字符集*是将用于字符集的字符集**RFC 2047** 编码的realnameifrealname包含非ASCII字符。可以是一个str或一个 实例Charset。默认为utf-8。

*在版本3.3中更改：*添加了*字符集*选项。

email.utils.**getaddresses**(*fieldvalues*)

此方法返回由返回的表单的2元组列表parseaddr()。 *fieldvalues*是可能返回的一系列头字段值 Message.get_all。下面是一个简单的例子，它可以获取消息的所有收件人：

```
from email.utils import getaddresses

tos = msg.get_all('to', [])
ccs = msg.get_all('cc', [])
resent_tos = msg.get_all('resent-to', [])
resent_ccs = msg.get_all('resent-cc', [])
all_recipients = getaddresses(tos + ccs + resent_tos + resent_ccs)
```

email.utils.**parsedate**(*日期*)

尝试根据中的规则解析日期 **RFC 2822**。但是，一些邮件程序不遵循指定的格式，因此parsedate()在这种情况下尝试正确猜测。 *日期*是一个字符串，包含一个**RFC 2822等** 日期。如果成功解析日期，则返回可直接传递给的9元组；否则将被退回。请注意，结果元组的索引6,7和8不可用。"Mon, 20 Nov 1995 19:12:08 -0500"parsedate() time.mktime() None

email.utils.**parsedate_tz**(*日期*)

执行与之相同的功能parsedate()，但返回一个None或一个10元组; 前9个元素组成一个可以直接传递的元组，time.mktime()第十个元素 是日期时区与UTC（这是格林威治标准时间的正式期限）[1]的偏移量。如果输入字符串没有时区，则返回的元组的最后一个元素是None。请注意，结果元组的索引6,7和8不可用。

email.utils.**parsedate_to_datetime**(*日期*)

的倒数format_datetime()。执行相同的功能 parsedate()，但成功返回a datetime。如果输入日期的时区为-0000，那么这datetime将是一个天真的 datetime，如果日期符合RFC，它将代表UTC的时间，但没有指示日期来自的消息的实际源时区。如果输入日期有任何其他有效的时区偏移量，datetime则会知道datetime相应的a。 timezone tzinfo

3.3版本的新功能

email.utils.**mktime_tz**(*元组*)

将返回的10元组parsedate_tz()转换为UTC时间戳（自Epoch以来的秒数）。如果元组中的时区项目是None假设当地时间。

email.utils.**formatdate**(*timeval = None*，*localtime = False*，*usegmt = False*)

按照每个返回一个日期字符串 **RFC 2822**，例如：

```
Fri, 09 Nov 2001 01:08:47 -0000
```

可选*的timeval*如果给定的是作为接受一个浮点时间值 time.gmtime()和time.localtime()，否则就使用当前时间。

可选*本地时间*是一个标志，当True，解释*timeval中*，并返回相对于本地时区而不是UTC，适当服用夏令时考虑的日期。默认值False是使用UTC。

可选的*usegmt*是一个标志，当True输出带有时区的日期字符串作为ascii字符串时GMT，而不是数字-0000。这对于某些协议（如HTTP）是必需的。这仅适用于*本地*时间 False。默

认是False。

email.utils.format_datetime(*dt*, *usegmt = False*)

喜欢formatdate，但输入是一个datetime实例。如果它是一个天真的日期时间，它被假定为“没有关于源时区的信息的UTC”，并且常规-0000用于时区。如果它是知道的datetime，则使用数字时区偏移量。如果它是一个零偏移的意识到的时区，那么可以将*usegmt*设置为True，在这种情况下，GMT将使用该字符串而不是数字时区偏移量。这提供了一种生成标准符合HTTP日期标头的方法。

3.3版本的新功能

email.utils.decode_rfc2231(*s*)

解码字符串*小号*根据**RFC 2231**。

email.utils.encode_rfc2231(*s*, *charset = None*, *language = None*)

编码字符串*小号*根据**RFC 2231**。可选的*字符集*和 *语言*（如果给出）是要使用的字符集名称和语言名称。如果两者都没有给出，则 *s*按原样返回。如果给出*字符集*但*语言* 不是，则使用空字符串对*语言进行*编码。

email.utils.collapse_rfc2231_value(*value*, *errors ='replace'*, *fallback_charset ='us-ascii'*)

当一个头参数被编入时 **RFC 2231**格式， Message.get_param可以返回包含字符集，语言和值的3元组。 collapse_rfc2231_value()将其变成一个unicode字符串。可选的*错误*传递给*错误*的参数str的 encode()方法; 它默认为'replace'。可选 *fallback_charset*指定要使用的字符集，如果是 **RFC 2231**头文件不为Python所知; 它默认为'us-ascii'。

为了方便起见，如果传递的*值*collapse_rfc2231_value()不是元组，它应该是一个字符串，并且不加引号就返回。

email.utils.decode_params(*params*)

根据。解码参数列表 **RFC 2231**。 *params*是包含表单元素的2元组序列。(content-type, string-value)

脚注

[1] 请注意，时区偏移的符号time.timezone与同一时区的变量符号相反；后面的变量在遵循POSIX标准时遵循这个模块**RFC 2822**。

15年1月19日。email.iterators：迭代器

源代码： Lib / email / iterators.py

使用该Message.walk方法在消息对象树上迭代相当容易 。该 email.iterators模块为消息对象树提供了一些有用的更高级别的迭代。

email.iterators.**body_line_iterator**(*msg*，*decode = False*)

这遍历*msg的*所有子部分中的所有有效负载，逐行返回字符串有效负载。它跳过所有的子部分头文件，并跳过任何不是Python字符串的有效负载的子部分。这有点相当于从文件中读取消息的平面文本表示readline()，跳过所有中间头文件。

可选*解码*传递给Message.get_payload。

email.iterators.**typed_subpart_iterator**(*msg*，*maintype ='text'*，*subtype = None*)

这遍历*msg的*所有子部分，仅返回匹配由*maintype*和*subtype*指定的MIME类型的*子部分*。

请注意，*子类型*是可选的; 如果省略，则子部分MIME类型匹配仅与主类型匹配。 *maintype* 也是可选的; 它默认为 *文本*。

因此，默认typed_subpart_iterator()返回MIME类型为*text / *的*每个子部分。

以下功能已被添加为有用的调试工具。它应该 *不*被认为是支持公共接口包的一部分。

email.iterators.**_structure**(*msg*，*fp = None*，*level = 0*，*include_default = False*)

打印消息对象结构的内容类型的缩进表示。例如：

```
>>> msg = email.message_from_file(somefile)
>>> _structure(msg)
multipart/mixed
    text/plain
    text/plain
    multipart/digest
        message/rfc822
            text/plain
        message/rfc822
            text/plain
        message/rfc822
            text/plain
        message/rfc822
            text/plain
        message/rfc822
            text/plain
    text/plain
```

可选的*fp*是一个类似文件的对象来打印输出。它必须适合Python的print()功能。 *级别*在内部使用。 *include_default*，如果为true，则也输出默认类型。

19.2。 json- JSON编码器和解码器

源代码： Lib / json / __ init__.py

JSON（JavaScript对象表示法），由 **RFC 7159**（过时了**RFC 4627**）和 ECMA-404，是一种轻量级的数据交换格式，它受 JavaScript对象字面值语法的启发（尽管它不是JavaScript的严格子集）[1]。

json公开了标准库marshal和pickle模块用户熟悉的API 。

编码基本的Python对象层次结构：

```
>>> import json
>>> json.dumps(['foo', {'bar': ('baz', None, 1.0, 2)}])
'["foo", {"bar": ["baz", null, 1.0, 2]}]'
>>> print(json.dumps("\"foo\bar"))
"\"foo\bar"
>>> print(json.dumps('\u1234'))
"\u1234"
>>> print(json.dumps('\\'))
"\\"
>>> print(json.dumps({"c": 0, "b": 0, "a": 0}, sort_keys=True))
{"a": 0, "b": 0, "c": 0}
>>> from io import StringIO
>>> io = StringIO()
>>> json.dump(['streaming API'], io)
>>> io.getvalue()
'["streaming API"]'
```

紧凑编码：

```
>>> import json
>>> json.dumps([1, 2, 3, {'4': 5, '6': 7}], separators=(',', ':'))
'[1,2,3,{"4":5,"6":7}]'
```

漂亮的印刷：

```
>>> import json
>>> print(json.dumps({'4': 5, '6': 7}, sort_keys=True, indent=4))
{
    "4": 5,
    "6": 7
}
```

解码JSON：

```
>>> import json
>>> json.loads('["foo", {"bar":["baz", null, 1.0, 2]}]')
['foo', {'bar': ['baz', None, 1.0, 2]}]
>>> json.loads('"\\"foo\\bar"')
'"foo\x08ar'
```

```
>>> from io import StringIO
>>> io = StringIO('["streaming API"]')
>>> json.load(io)
['streaming API']
```

专注于JSON对象解码：

```
>>> import json
>>> def as_complex(dct):
...     if '__complex__' in dct:
...         return complex(dct['real'], dct['imag'])
...     return dct
...
>>> json.loads('{"__complex__": true, "real": 1, "imag": 2}',
...     object_hook=as_complex)
(1+2j)
>>> import decimal
>>> json.loads('1.1', parse_float=decimal.Decimal)
Decimal('1.1')
```

扩展JSONEncoder：

```
>>> import json
>>> class ComplexEncoder(json.JSONEncoder):
...     def default(self, obj):
...         if isinstance(obj, complex):
...             return [obj.real, obj.imag]
...         # Let the base class default method raise the TypeError
...         return json.JSONEncoder.default(self, obj)
...
>>> json.dumps(2 + 1j, cls=ComplexEncoder)
'[2.0, 1.0]'
>>> ComplexEncoder().encode(2 + 1j)
'[2.0, 1.0]'
>>> list(ComplexEncoder().iterencode(2 + 1j))
['[2.0', ', 1.0', ']']
```

json.tool在shell中使用来验证和漂亮打印：

```
$ echo '{"json":"obj"}' | python -m json.tool
{
    "json": "obj"
}
$ echo '{1.2:3.4}' | python -m json.tool
Expecting property name enclosed in double quotes: line 1 column 2 (char 1)
```

有关详细文档，请参阅命令行界面。

注意： JSON是YAML 1.2的一个子集。由该模块的默认设置（特别是默认*分隔符*值）生成的JSON 也是YAML 1.0和1.1的子集。因此该模块也可以用作YAML串行器。

19.2.1。基本用法

json.dump（*obj*，*fp*，***，*skipkeys = False*，*ensure_ascii = True*，*check_circular = True*，*allow_nan = True*，*cls = None*，*indent = None*，*separators = None*，*default = None*，*sort_keys = False*，*** kw* ）

使用此转换表，将 *obj*作为JSON格式的流序列化为*fp*（.write()支持 文件类对象）。

如果*skipkeys*为真（默认：False），那么字典键是一个基本类型（不str，int，float，bool， None）将被跳过，而不是养TypeError。

该json模块总是会产生str对象，而不是 bytes对象。因此，fp.write()必须支持str 输入。

如果*ensure_ascii*为true（缺省值），则输出将保证所有传入的非ASCII字符都被转义。如果*ensure_ascii*为false，则这些字符将按原样输出。

如果*check_circular*为false（默认值：）True，那么容器类型的循环引用检查将被跳过，并且循环引用将导致OverflowError（或更糟糕）。

如果*allow_nan*是假的（默认：True），那么这将是一个 ValueError序列化超出范围的float值（nan， inf，-inf在JSON规范的严格遵守）。如果*allow_nan*是真实的，其JavaScript当量（NaN， Infinity，-Infinity）会被使用。

如果*indent*是一个非负整数或字符串，那么JSON数组元素和对象成员将与该缩进级别相匹配。缩进级别为0，否定或""仅插入换行符。 None（默认）选择最紧凑的表示形式。使用正整数缩进缩进每个级别的许多空间。如果*缩进*是一个字符串（如"\t"），则该字符串用于缩进每个级别。

*在版本3.2中更改：*除了整数外，还允许字符串用于*缩进*。

如果指定，*分隔符*应该是一个 元组。默认值是，如果*缩进*是和 其他。要获得最紧凑的JSON表示，您应该指定消除空白。 (item_separator， key_separator)(', ', ': ')None(',', ': ')(',', ':')

*改变在3.4版本：*使用如果默认*缩进*不是。 (',', ': ')None

如果指定，则*默认值*应为针对不能以其他方式进行序列化的对象调用的函数。它应该返回一个JSON可编码版本的对象或引发一个TypeError。如果未指定，TypeError 则引发。

如果*sort_keys*为true（默认值：）False，则字典的输出将按键排序。

要使用自定义JSONEncoder子类（例如覆盖该 default()方法来序列化其他类型的子类），请使用*cls* kwarg 指定它；否则JSONEncoder使用。

*在版本3.6中更改：*所有可选参数现在仅为关键字。

json.dumps（*obj*，***，*skipkeys = False*，*ensure_ascii = True*，*check_circular = True*，*allow_nan = True*，*cls = None*，*indent = None*，*separators = None*，*default = None*，*sort_keys = False*，*** kw* ）

将*obj*序列*化为*str使用此转换表格式化的JSON 。这些论点的含义与之相同 dump()。

注意： 不像pickle和marshal，JSON是不是一个框架式协议，因此试图序列反复调用多个对象 dump() 使用相同的*FP*将导致一个无效的JSON文件。

注意： JSON的键/值对中的键始终是该类型的键str。当字典转换为JSON时，字典中的所有键都被强制转换为字符串。因此，如果字典被转换成JSON然后又回到字典中，字典可能不等于原来的字典。也就是说，如果x具有非字符串键。 loads(dumps(x)) != x

json.load（*fp*，*，*cls = None*，*object_hook = None*，*parse_float = None*，*parse_int = None*，*parse_constant = None*，*object_pairs_hook = None*，** kw*）

使用此转换表，将 *fp*（包含JSON文档的.read() 支持类文件对象）反序列*化为* Python对象。

*object_hook*是一个可选函数，它将被解码的任何对象字面值的结果调用（a dict）。*object_hook*的返回值 将被用来代替dict。此功能可用于实现自定义解码器（例如JSON-RPC 类提示）。

*object_pairs_hook*是一个可选函数，它将被调用，并且任何对象文本的结果都使用有序列表对进行解码。将使用*object_pairs_hook*的返回值来代替 dict。此功能可用于实现自定义解码器，这些解码器依赖于解码键和值对collections.OrderedDict()的顺序（例如， 会记住插入顺序）。如果 还定义了*object_hook*，则*object_pairs_hook*优先。

*在版本3.1中更改：*添加了对*object_pairs_hook的*支持。

如果指定了*parse_float*，将会调用每个JSON浮点数的字符串进行解码。默认情况下，这相当于float(num_str)。这可以用来为JSON浮点数使用另一种数据类型或解析器（例如decimal.Decimal）。

如果指定了*parse_int*，将使用每个JSON int的字符串进行解码。默认情况下，这相当于int(num_str)。这可以用来为JSON整数使用另一种数据类型或解析器（例如float）。

parse_constant：如果指定，将与下列字符串之一叫'-Infinity'，'Infinity'，'NaN'。如果遇到无效的JSON号码，这可以用来引发异常。

*在版本3.1中更改：parse_constant*不再被'null'，'true'，'false'调用。

要使用自定义JSONDecoder子类，请使用cls kwarg 指定它; 否则JSONDecoder使用。其他关键字参数将传递给该类的构造函数。

如果被反序列化的数据不是有效的JSON文档，JSONDecodeError则会引发a。

*在版本3.6中更改：*所有可选参数现在仅为关键字。

json.loads（*s*，*，*encoding = None*，*cls = None*，*object_hook = None*，*parse_float = None*，*parse_int = None*，*parse_constant = None*，*object_pairs_hook = None*，** kw*）

使用此转换表，将*s*（a 或包含JSON文档的实例）反序列化为Python对象。strbytesbytearray

其他参数的含义与中相同load()，除了被忽略和弃用的*编码*外。

如果被反序列化的数据不是有效的JSON文档， `JSONDecodeError`则会引发a。

*在版本3.6中更改：*s现在可以是type `bytes`或`bytearray`。输入编码应该是UTF-8，UTF-16或UTF-32。

19.2.2。编码器和解码器

class `json.`JSONDecoder（*，object_hook = None，parse_float = None，parse_int = None，parse_constant = None，strict = True，object_pairs_hook = None* ）

简单的JSON解码器。

默认情况下，在解码中执行以下翻译：

JSON	蟒蛇
目的	字典
排列	名单
串	海峡
数字（int）	INT
数量（实际）	浮动
真正	真正
假	假
空值	没有

它也理解`NaN`，`Infinity`并且`-Infinity`作为它们相应的`float`值，它超出了JSON规范。

如果指定了*object_hook*，将会调用每个JSON对象的解码结果，并使用它的返回值代替给定的值 `dict`。这可以用来提供自定义的反序列化（例如，支持JSON-RPC类提示）。

如果指定*object_pairs_hook*，将调用每个JSON对象的结果，并使用有序的对列表进行解码。将使用*object_pairs_hook*的返回值 来代替`dict`。此功能可用于实现自定义解码器，这些解码器依赖于解码键和值对`collections.OrderedDict()`的顺序（例如， 会记住插入顺序）。如果 还定义了*object_hook*，则*object_pairs_hook*优先。

*在版本3.1中更改：*添加了对*object_pairs_hook*的支持。

如果指定了*parse_float*，将会调用每个JSON浮点数的字符串进行解码。默认情况下，这相当于`float(num_str)`。这可以用来为JSON浮点数使用另一种数据类型或解析器（例如`decimal.Decimal`）。

如果指定了*parse_int*，将使用每个JSON int的字符串进行解码。默认情况下，这相当于`int(num_str)`。这可以用来为JSON整数使用另一种数据类型或解析器（例如`float`）。

parse_constant：如果指定，将与下列字符串之一叫`'-Infinity'`，`'Infinity'`，`'NaN'`。如果遇到无效的JSON号码，这可以用来引发异常。

如果*strict*为假（`True`是默认值），那么控制字符将被允许在字符串内。此上下文中的控制字符是字符代码在0-31范围内的字符，包括'`\t`'（制表符），'`\n`'，'`\r`'和'`\0`'。

如果被反序列化的数据不是有效的JSON文档，`JSONDecodeError`则会引发a。

*在版本3.6中更改：*所有参数现在仅为关键字。

`decode`（*s*）

返回*s*的Python表示（`str`包含JSON文档的实例）。

`JSONDecodeError` 如果给定的JSON文档无效，将会引发此问题。

`raw_decode`（*s*）

解码来自JSON文档*小号*（一个`str`与JSON文档开始），并在返回的Python表示的2元组以及索引*小号*文档结束位置。

这可以用来从一个字符串解码JSON文档，该字符串可能在最后有多余的数据。

class `json.`JSONEncoder（**，skipkeys = False，ensure_ascii = True，check_circular = True，allow_nan = True，sort_keys = False，indent = None，separators = None，default = None*）

用于Python数据结构的可扩展JSON编码器。

默认支持以下对象和类型：

蟒蛇	**JSON**
字典	目的
列表，元组	排列
海峡	串
int，float，int-和float派生的枚举	数
真正	真正
假	假
没有	空值

*在版本3.4中更改：*添加了对int和float派生的Enum类的支持。

为了扩展这个以识别其他对象，子类并`default()`使用另一个方法实现一个 方法，`o`如果可能的话，该方法返回一个可序列化对象，否则它应该调用超类实现（提升`TypeError`）。

如果*skipkeys*为false（默认值），那么它就是一个`TypeError`尝试钥匙编码不在`str`，`int`，`float`或`None`。如果*skipkeys*为true，则简单地跳过这些项目。

如果*ensure_ascii*为true（缺省值），则输出将保证所有传入的非ASCII字符都被转义。如果*ensure_ascii*为false，则这些字符将按原样输出。

如果*check_circular*为true（默认值），那么在编码期间将检查列表，字典和自定义编码对象的循环引用，以防止无限递归（这会导致`OverflowError`）。否则，不会进行这种检查。

如果*allow_nan*为真（默认值），然后NaN，Infinity和 -Infinity将被编码为这样。这种行为不符合JSON规范，但与大多数基于JavaScript的编码器和解码器一致。否则，这将是一个ValueError编码这样的浮游物。

如果*sort_keys*为true（默认值：）False，则字典的输出将按键排序; 这对于回归测试非常有用，可以确保JSON序列化可以在日常的基础上进行比较。

如果*indent*是一个非负整数或字符串，那么JSON数组元素和对象成员将与该缩进级别相匹配。缩进级别为0，否定或""仅插入换行符。 None（默认）选择最紧凑的表示形式。使用正整数缩进缩进每个级别的许多空间。如果*缩进*是一个字符串（如"\t"），则该字符串用于缩进每个级别。

*在版本3.2中更改：*除了整数外，还允许字符串用于*缩进*。

如果指定，*分隔符*应该是一个 元组。默认值是，如果*缩进*是和 其他。要获得最紧凑的JSON 表示，您应该指定消除空白。 (item_separator, key_separator) (', ', ': ') None (',', ': ') (',', ':')

*改变在3.4版本：*使用如果默认*缩进*不是。 (',', ': ') None

如果指定，则*默认值*应为针对不能以其他方式进行序列化的对象调用的函数。它应该返回一个JSON可编码版本的对象或引发一个TypeError。如果未指定，TypeError 则引发。

*在版本3.6中更改：*所有参数现在仅为关键字。

default(*o*)

在子类中实现此方法，以便返回*o*的可序列化对象，或调用基本实现（以提高a TypeError）。

例如，要支持任意迭代器，可以像这样实现默认值：

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return json.JSONEncoder.default(self, o)
```

encode(*o*)

返回Python数据结构的JSON字符串表示形式*o*。例如：

```
>>> json.JSONEncoder().encode({"foo": ["bar", "baz"]})
'{"foo": ["bar", "baz"]}'
```

iterencode(*o*)

对给定对象*o进行*编码，并将每个字符串表示视为可用。例如：

```
for chunk in json.JSONEncoder().iterencode(bigobject):
    mysocket.write(chunk)
```

19.2.3。例外

*异常*json. JSONDecodeError（ *msg* ，*doc* ，*pos* ）

ValueError具有以下附加属性的子类：

msg

未格式化的错误消息。

doc

正在解析的JSON文档。

pos

解析失败的*文档*的起始索引。

lineno

与*pos*对应的行。

colno

对应于*pos*的列。

3.5版本中的新功能。

19.2.4。标准兼容性和互操作性

JSON格式由指定 **RFC 7159**和 ECMA-404。本节详细介绍了该模块对RFC的遵从级别。为了简单起见，JSONEncoder和JSONDecoder亚类和除了明确提及的那些其它参数，都没有考虑。

该模块不严格遵守RFC，实现了一些有效的JavaScript但不是有效的JSON的扩展。尤其是：

- 无限和NaN数值被接受和输出;
- 对象中的重复名称将被接受，并且只使用最后一个名称 - 值对的值。

由于RFC允许RFC兼容的解析器接受不符合RFC的输入文本，因此该模块的解串器在技术上符合RFC默认设置。

19.2.4.1。字符编码

RFC要求使用UTF-8，UTF-16或UTF-32来表示JSON，UTF-8是最大互操作性的推荐默认值。

根据RFC的规定，尽管不是必需的，但该模块的序列化*程序*默认情况下会设置 *ensure_ascii* = *True*，从而转义输出，以便生成的字符串仅包含ASCII字符。

除了*ensure_ascii*参数外，该模块严格按照Python对象之间的转换进行定义 ，因此不会直接解决字符编码问题。Unicode strings

RFC禁止将字节顺序标记（BOM）添加到JSON文本的开头，并且此模块的串行器不会将BOM添加到其输出中。RFC允许但不要求JSON解串器忽略其输入中的初始BOM。`ValueError` 当存在初始BOM时，该模块的解串器会产生一个。

RFC没有明确禁止包含与有效的Unicode字符不对应的字节序列的JSON字符串（例如不成对的UTF-16替代品），但是它确实注意到它们可能导致互操作性问题。默认情况下，该模块接受并输出（当存在于原始中 `str` ）用于这种序列的代码点。

19.2.4.2。无限和NaN数值

RFC不允许表示无限或NaN号码值。尽管如此，在默认情况下，该模块能够接受和输出 `Infinity` ， `-Infinity`以及`NaN`就好像它们是有效的JSON数字面值：

```
>>> # Neither of these calls raises an exception, but the results are not valid JSON
>>> json.dumps(float('-inf'))
'-Infinity'
>>> json.dumps(float('nan'))
'NaN'
>>> # Same when deserializing
>>> json.loads('-Infinity')
-inf
>>> json.loads('NaN')
nan
```

在序列化程序中，可以使用*allow_nan*参数来改变这种行为。在反序列化器中，可以使用*parse_constant*参数来改变这种行为。

19.2.4.3。对象中的重复名称

RFC指定JSON对象中的名称应该是唯一的，但不要求JSON对象中的重复名称应该如何处理。默认情况下，该模块不会引发异常; 相反，它忽略了给定名称以外的所有名称 - 值对：

```
>>> weird_json = '{"x": 1, "x": 2, "x": 3}'
>>> json.loads(weird_json)
{'x': 3}
```

该*object_pairs_hook*参数可以用来改变这种行为。

19.2.4.4。顶级非对象，非数组值

旧版本指定的JSON旧版本 **RFC 4627**要求JSON文本的顶级值必须是JSON对象或数组（Python`dict`或`list` ），并且不能是JSON null，布尔值，数字或字符串值。 **RFC 7159**删除了这个限制，而且这个模块没有，也没有在其串行器或解串器中实现这个限制。

无论如何，为了获得最大的互操作性，您可能希望自己自愿遵守限制。

19.2.4.5。实施限制

一些JSON解串器实现可能会对以下内容设置限制：

- 接受的JSON文本的大小
- JSON对象和数组的最大嵌套级别
- JSON数字的范围和精度
- JSON字符串的内容和最大长度

除了相关的Python数据类型本身或Python解释器本身以外，本模块不会强加任何限制。

序列化为JSON时，请注意可能会使用JSON的应用程序中的任何此类限制。特别是，JSON数字被反序列化为IEEE 754双精度数字并且因此受制于该表示的范围和精度限制是很常见的。当序列化`int`极大数量的Python 值时，或者在序列化“奇异”数字类型的实例时， 这尤其重要`decimal.Decimal`。

19.2.5。命令行界面

源代码： Lib / json / tool.py

该`json.tool`模块提供了一个简单的命令行界面来验证和漂亮地打印JSON对象。

如果未指定可选参数`infile`和`outfile`参数，`sys.stdin`并`sys.stdout`分别使用：

```
$ echo '{"json": "obj"}' | python -m json.tool
{
    "json": "obj"
}
$ echo '{1.2:3.4}' | python -m json.tool
Expecting property name enclosed in double quotes: line 1 column 2 (char 1)
```

*在版本3.5中更改：*输出现在与输入顺序相同。使用该`--sort-keys`选项按字母顺序排序词典的输出。

19.2.5.1。命令行选项

`infile`

要验证或漂亮打印的JSON文件：

```
$ python -m json.tool mp_films.json
[
    {
        "title": "And Now for Something Completely Different",
        "year": 1971
    },
    {
        "title": "Monty Python and the Holy Grail",
        "year": 1975
    }
]
```

如果没有指定*infile*，请阅读`sys.stdin`。

outfile
: 写的输出*INFILE*给定*OUTFILE*。否则，写入sys.stdout。

--sort-keys
: 按键排序字典输出。

  3.5版本中的新功能。

-h, --help
: 显示帮助信息。

脚注

[1] 正如RFC 7159勘误中所述，JSON允许在字符串中使用文字U + 2028（LINE分隔符）和U + 2029（PARAGRAPH SEPARATOR）字符，而JavaScript（从ECMAScript版本1开始）则不允许。

19.3。 mailcap- Mailcap文件处理

源代码： Lib / mailcap.py

Mailcap文件用于配置感知MIME的应用程序（如邮件阅读器和Web浏览器）如何对具有不同MIME类型的文件作出反应。（名称“mailcap”源自短语“邮件功能”）。例如，一个mailcap文件可能包含一行。然后，如果用户遇到具有MIME类型*video / mpeg*的电子邮件消息或Web文档，将被替换为文件名（通常属于临时文件），并且**xmpeg**程序可以自动启动以查看该文件。

```
video/mpeg; xmpeg %s%s
```

mailcap格式记录在 **RFC 1524**，“用于多媒体邮件格式信息的用户代理配置机制”，但不是互联网标准。但是，大多数Unix系统都支持mailcap文件。

mailcap.findmatch（*caps*，*MIMEtype*，*key ='view'*，*filename ='/ dev / null'*，*plist = []*）

返回一个2元组; 第一个元素是包含要执行的命令行的字符串（可以传递给它 os.system() ），第二个元素是给定MIME类型的mailcap条目。如果找不到匹配的MIME类型，则返回。 (None, None)

*键*是所需字段的名称，它表示要执行的活动的类型; 默认值是'view'，因为在最常见的情况下，您只需查看MIME类型数据的主体。如果您想要创建给定MIME类型的新体或者更改现有正文数据，其他可能的值可能是“撰写”和“编辑”。看到**RFC 1524**中有关这些字段的完整列表。

*filename*是要%s在命令行中替换的文件名; 默认值’/dev/null’几乎肯定不是你想要的，所以通常你会通过指定文件名覆盖它。

*plist*可以是包含命名参数的列表; 默认值只是一个空列表。列表中的每个条目都必须是一个包含参数名称，等号（’=’）和参数值的字符串。Mailcap条目可以包含名称参数，例如%{foo}，它将被名为'foo'的参数的值替换。例如，如果命令行在mailcap文件中，并且*plist*设置为 ，则生成的命令行将为 。 showpartial %{id} %{number} %{total} [’id=1’, ’number=2’, ’total=3’]’showpartial 1 2 3’

在mailcap文件中，可以选择指定“test”字段来测试某些外部条件（例如机器体系结构或所用的窗口系统）以确定是否应用该mailcap行。 findmatch() 将自动检查这些条件并在检查失败时跳过条目。

mailcap.getcaps（）

将MIME类型的字典映射到mailcap文件条目列表。该字典必须传递给该findmatch()函数。一个条目被存储为一个词典列表，但不需要知道这个表示的细节。

信息来源于系统中找到的所有mailcap文件。在用户的mailcap文件的设置$HOME/.mailcap将覆盖在系统mailcap文件中的设置/etc/mailcap，/usr/etc/mailcap和/usr/local/etc/mailcap。

一个示例用法：

```
>>> import mailcap
>>> d = mailcap.getcaps()
>>> mailcap.findmatch(d, 'video/mpeg', filename='tmp1223')
('xmpeg tmp1223', {'view': 'xmpeg %s'})
```

19.4。mailbox- 以各种格式操作邮箱

源代码： Lib / mailbox.py

该模块定义了两类，Mailbox并且Message，用于访问和操作的磁盘上的邮箱和它们所包含的消息。 Mailbox提供从键到消息的字典式映射。 Message扩展email.message模块的 Message类与格式特定的状态和行为。支持的邮箱格式是Maildir，mbox，MH，Babyl和MMDF。

也可以看看:

模 email
表示和操纵消息。

19.4.1。Mailbox对象

*类*mailbox.Mailbox
邮箱，可能会被检查和修改。

所述Mailbox类定义了一个接口，并且不旨在被实例化。相反，特定于格式的子类应该从中继承， Mailbox并且您的代码应该实例化特定的子类。

该Mailbox接口是类似于字典的，与对应于消息的小键。密钥由Mailbox它们将被使用的实例发布，并且仅对该Mailbox 实例有意义。即使相应的消息被修改，一个密钥也会继续识别消息，例如通过用另一个消息替换它。

消息可以被添加到Mailbox使用set-类似方法实例add()，并使用去除del语句或设定等的方法remove()和discard()。

Mailbox接口语义在一些值得注意的方面与字典语义不同。每次请求消息时，都会Message根据邮箱的当前状态生成新的表示（通常为实例）。同样，当消息被添加到 Mailbox实例时，所提供的消息表示的内容被复制。在任何情况下都不是对Mailbox实例保留的消息表示的引用。

默认Mailbox迭代器遍历消息表示，而不是默认字典迭代器的键。而且，在迭代过程中修改邮箱是安全和明确的。迭代器创建后添加到邮箱的消息将不会被迭代器看到。在迭代器产生它们之前从邮箱中删除的邮件将被静静地跳过，尽管KeyError如果随后删除相应的消息，使用迭代器中的密钥可能会导致 异常。

警告: 修改可能被其他进程同时更改的邮箱时要非常谨慎。用于此类任务的最安全邮箱格式是Maildir; 尽量避免使用mbox等单文件格式进行并发写入。如果您要修改邮箱，则*必须在*读取文件中的任何邮件或通过添加或删除邮件进行任何更改*之前*调用lock()和unlock()方法*来*锁定邮箱。无法锁定邮箱会导致丢失邮件或损坏整个邮箱。

Mailbox 实例具有以下方法：

add (*消息*)

将*邮件*添加到邮箱并返回已分配给它的密钥。

参数*消息*可以是Message实例， email.message.Message实例，字符串，字节字符串或类似文件的对象（应该以二进制模式打开）。如果*消息*是适当的格式特定的Message子类的mboxMessage实例（例如，如果它是一个 实例并且这是一个mbox实例），则使用其格式特定的信息。否则，将使用针对格式特定信息的合理默认值。

*在版本3.2中进行了更改：*添加了对二进制输入的支持。

remove (*重点*)
__delitem__ (*重点*)
discard (*重点*)

从邮箱中删除与*密钥*对应的消息。

如果不存在这样的消息，KeyError则如果该方法被调用为remove()或者__delitem__()但如果该方法被调用时不引发异常则会引发异常discard()。 discard()如果底层邮箱格式支持其他进程的并发修改，则该行为可能是首选。

__setitem__ (*键* , *消息*)

用消息替换对应于*密钥*的*消息*。KeyError如果没有消息已经与*密钥*对应，则引发 异常。

同样add()，参数*消息*可以是Message 实例，email.message.Message实例，字符串，字节字符串或文件类对象（应该以二进制模式打开）。如果 *消息*是适当的格式特定的Message子类的mboxMessage实例（例如，如果它是一个实例并且这是一个 mbox实例），则使用其格式特定的信息。否则，当前对应于*密钥*的消息的格式特定信息保持不变。

iterkeys ( )
keys ( )

如果调用as iterkeys() ，则返回所有键的迭代器;如果调用as ，则返回键列表keys()。

itervalues ( )
__iter__ ( )
values ( )

如果调用为itervalues()或__iter__() ，则返回所有消息的表示形式的迭代器，或者如果调用为 ，则返回此类表示形式的列表values()。 Message除非在Mailbox 实例初始化时指定了自定义消息工厂，否则这些消息将表示为适当的格式特定的子类的实例。

> **注意:** __iter__()字典的行为不同于字典，它们遍历密钥。

iteritems ( )
items ( )

通过（*键* , *消息* ）对返回一个迭代器，其中*key*是一个键， *message*是一个消息表示形式，如果调用为，则iteritems()返回一个这样的对的列表items()。 Message除非在

Mailbox实例初始化时指定了自定义消息工厂，否则这些消息将表示为适当的格式特定的 子类的实例。

get（*键*，*默认=无*）
__getitem__（*重点*）

返回对应于*键*的消息的表示。如果不存在这样的消息，*默认情况下*，如果该方法被调用的返回 get()以及KeyError如果该方法被称为引发异常__getitem__()。该消息表示为适当格式特定Message子类的实例，除非在Mailbox实例初始化时指定了自定义消息工厂。

get_message（*重点*）

将对应于*key*的消息的表示形式作为特定格式的Message子类的实例返回，或者KeyError如果不存在此类消息，则引发异常。

get_bytes（*重点*）

返回对应于*键*的消息的字节表示，或者KeyError如果不存在这样的消息则引发异常。

3.2版本中的新功能

get_string（*重点*）

返回对应于*键*的消息的字符串表示形式，或者KeyError如果不存在此类消息，则引发异常。通过处理该消息email.message.Message将其转换为7位干净表示。

get_file（*重点*）

返回与*密钥*对应的消息的类文件表示，或者KeyError如果不存在此类消息则引发异常。类文件对象的行为就好像以二进制模式打开一样。该文件在不再需要时应该关闭。

*在版本3.2中更改：*文件对象确实是一个二进制文件; 以前它在文本模式下被错误地返回。此外，类文件对象现在支持上下文管理协议：您可以使用with语句自动关闭它。

> **注意：** 与消息的其他表示形式不同，类似文件的表示形式不一定独立于Mailbox创建它们的实例或底层邮箱。每个子类都提供更具体的文档。

__contains__（*重点*）

True如果*键*对应于消息，False则返回，否则返回。

__len__（）

返回邮箱中的邮件数量。

clear（）

删除邮箱中的所有邮件。

pop（*键*，*默认=无*）

返回与*密钥*对应的消息表示并删除该消息。如果不存在这样的消息，则返回*默认值*。该消息表示为适当格式特定Message子类的实例， 除非在Mailbox实例初始化时指定了自定义消息工厂。

popitem ( )

返回任意（*密钥*，*消息*）对，其中*密钥*是密钥， *消息*是消息表示，并删除相应的消息。如果邮箱为空，则引发KeyError异常。该消息表示为适当格式特定Message子类的实例， 除非在Mailbox实例初始化时指定了自定义消息工厂。

update (*arg*)

参数*ARG*应该是一个*键*-到-*消息*映射或（可迭代*键*，*消息*）对。更新邮箱，以便对于每个给定的 *密钥*和*消息*，将与*密钥*相对应的消息设置为 *消息*，就像通过使用一样__setitem__()。同样__setitem__() ，每个*密钥*必须已经对应邮箱中的消息，否则KeyError会引发异常，所以通常*arg*不是Mailbox实例。

> **注意:** 与字典不同，不支持关键字参数。

flush ( )

将任何挂起的更改写入文件系统。对于某些Mailbox 子类，变更总是立即写入，flush()并且什么也不做，但是您仍应养成调用此方法的习惯。

lock ( )

获取邮箱上的独家咨询锁，以便其他进程知道不要修改它。ExternalClashError如果锁不可用，则引发An 。使用的特定锁定机制取决于邮箱格式。在对其内容进行任何修改之前，您应*始终*锁定邮箱。

unlock ( )

释放邮箱上的锁，如果有的话。

close ( )

冲洗邮箱，如有必要将其解锁，然后关闭所有打开的文件。对于某些Mailbox子类，此方法什么都不做。

19.4.1.1。 Maildir

*类*mailbox.Maildir (*dirname*，*factory = None*，*create = True*)

Mailbox Maildir格式邮箱的子类。参数 *工厂*是一个可调用的对象，它接受类似文件的消息表示（其行为如同以二进制模式打开）并返回一个自定义表示。如果*工厂*是None ，MaildirMessage则用作默认消息表示。如果*创建*是True，如果它不存在的邮箱已创建。

由于历史原因，*dirname*被命名为这样而不是*路径*。

Maildir是一种基于目录的邮箱格式，为qmail邮件传输代理发明，现在已被其他程序广泛支持。Maildir邮箱中的邮件存储在通用目录结构中的单独文件中。这种设计允许Maildir邮箱被多个不相关的程序访问和修改，而不会造成数据损坏，因此不需要文件锁定。

Maildir的邮箱包含三个子目录，分别为： tmp， new，和cur。消息暂时在tmp子目录中创建，然后移至new子目录以完成交付。邮件用户代理可能随后将邮件移动到 cur子目录，并将有关邮件状态的信息存储在附加到其文件名的特殊“信息”部分中。

由Courier邮件传输代理引入的样式文件夹也受支持。如果'.'是其名称中的第一个字符，主邮箱的任何子目录都被视为一个文件夹 。文件夹名称由Maildir没有前导的表示'.'。每个文件夹本身都是Maildir邮箱，但不应包含其他文件夹。相反，逻辑嵌套表示使用'.'分隔层，例如"Archived.2005.07"。

注意： Maildir规范要求':'在某些消息文件名中使用冒号（ ）。但是，某些操作系统不允许在文件名中使用此字符。如果您希望在此类操作系统上使用类似Maildir的格式，则应该指定另一个字符来代替。感叹号（'!'）是一种流行的选择。例如：

```
import mailbox
mailbox.Maildir.colon = '!'
```

该colon属性也可以基于每个实例来设置。

MaildirMailbox除了以下内容，实例还具有所有的方法：

list_folders ( )
返回所有文件夹的名称列表。

get_folder (*文件夹*)
返回一个Maildir表示名称为*folder*的文件夹的实例 。一NoSuchMailboxError ，如果文件夹不存在异常。

add_folder (*文件夹*)
创建一个名称为*folder的文件夹*并返回一个Maildir 代表它的实例。

remove_folder (*文件夹*)
删除名称为*文件夹的文件夹*。如果该文件夹包含任何消息，NotEmptyError则会引发异常并且该文件夹不会被删除。

clean ( )
从过去36小时内未访问过的邮箱中删除临时文件。Maildir规范说邮件阅读程序偶尔会这样做。

一些Mailbox方法的实施Maildir值得特别注意：

add (*消息*)
__setitem__ (*键* , *消息*)
update (*arg*)

警告： 这些方法根据当前的进程ID生成唯一的文件名。在使用多个线程时，可能会发生未检测到的名称冲突，并导致邮箱损坏，除非协调线程以避免使用这些方法同时操作同一个邮箱。

flush ( )
对Maildir邮箱的所有更改都立即应用，因此此方法不会执行任何操作。

lock ( )
unlock ( )

Maildir邮箱不支持（或需要）锁定，所以这些方法什么也不做。

close ( )
Maildir 实例不保留任何打开的文件，并且底层邮箱不支持锁定，所以此方法不会执行任何操作。

get_file (*重点*)
根据主机平台的不同，当返回的文件保持打开状态时，可能无法修改或删除基础消息。

也可以看看:

来自qmail的maildir手册页
格式的原始规范。

使用maildir格式
Maildir由其发明人注释。包含更新的名称创建方案和“信息”语义的详细信息。

来自Courier的maildir手册页
格式的另一个规范。介绍支持文件夹的常见扩展名。

19.4.1.2。 mbox

*类*mailbox. mbox (*路径* , *工厂=无* , *创建= True*)
Mailbox用于mbox格式邮箱的子类。参数*工厂*是一个可调用的对象，它接受类似文件的消息表示（其行为如同以二进制模式打开）并返回一个自定义表示。如果 *工厂*是None ，mboxMessage则用作默认消息表示。如果*创建*是True ，如果它不存在的邮箱已创建。

mbox格式是在Unix系统上存储邮件的经典格式。mbox邮箱中的所有邮件都存储在一个文件中，每个邮件的开头由前五个字符为“From”的行指示。

存在几种mbox格式的变体来解决原来的缺陷。为了兼容性，mbox实现了原始格式，有时称为*mboxo*。这意味着*Content-Length*标题（如果存在的话）被忽略，并且在消息体中的行的开始处任何出现的“From”都被转换为“> From”，当存储消息时，尽管出现“> From “在阅读邮件时不会转换为”发件人“。

一些Mailbox方法的实施mbox值得特别注意：

get_file (*重点*)
使用文件调用后flush()或close()在 mbox实例可能产生不可预测的结果或引发异常。

lock ( )
unlock ( )
使用三种锁定机制 - 点锁定和（如果可用） flock()和lockf()系统调用。

也可以看看:

来自qmail的mbox手册页

格式及其变化的规范。

从锡的mbox手册页
格式的另一个规范，详细介绍了锁定。

在Unix上配置Netscape Mail：为什么内容长度格式不好
使用原始mbox格式而不是变体的一个参数。

“mbox”是几个互不兼容邮箱格式的家族
mbox变化的历史。

19.4.1.3。 MH

*类*mailbox.MH（*路径*，*工厂=无*，*创建= True* ）

Mailbox MH格式邮箱的子类。参数*工厂*是一个可调用的对象，它接受类似文件的消息表示（其行为如同以二进制模式打开）并返回一个自定义表示。如果*工厂*是None，MHMessage则用作默认消息表示。如果*创建*是True，如果它不存在的邮箱已创建。

MH是一种基于目录的邮箱格式，是为邮件用户代理MH Message Handling System发明的。MH邮箱中的每封邮件都驻留在自己的文件中。除邮件外，MH邮箱还可能包含其他MH邮箱（称为*文件夹*）。文件夹可能无限期地嵌套。MH邮箱也支持*序列*，这些*序列*被命名为用于在逻辑上对消息进行分组而不移动到子文件夹的列表。序列.mh_sequences在每个文件夹中调用的文件中定义。

本MH类操纵MH邮箱，但它并不试图模拟所有的**MH**的行为。特别是，它不会修改，也不受**mh**用于存储其状态和配置的文件context或.mh_profile文件的影响。

MH Mailbox除了以下内容，实例还具有所有的方法：

list_folders（ ）
返回所有文件夹的名称列表。

get_folder（*文件夹*）
返回MH表示名称为*文件夹*的文件夹的实例 。—NoSuchMailboxError，如果文件夹不存在异常。

add_folder（*文件夹*）
创建一个名称为*folder的文件夹*并返回一个MH代表它的实例。

remove_folder（*文件夹*）
删除名称为*文件夹的文件夹*。如果该文件夹包含任何消息，NotEmptyError则会引发异常并且该文件夹不会被删除。

get_sequences（ ）
返回映射到键列表的序列名称字典。如果没有序列，则返回空字典。

set_sequences（*序列*）
根据序列重新定义邮箱中存在的*序列*，映射到键列表的名称字典（如返回的）get_sequences()。

pack ( )
根据需要重命名邮箱中的邮件以消除编号中的空白。序列表中的条目会相应更新。

> **注意:** 已经发行的密钥通过此操作失效，不应随后使用。

一些Mailbox方法的实施MH值得特别注意：

remove (*重点*)
__delitem__ (*重点*)
discard (*重点*)
这些方法立即删除该消息。不使用通过在其名称前面加逗号来标记消息以供删除的MH惯例。

lock ( )
unlock ( )
使用三种锁定机制 - 点锁定和（如果可用） flock() 和lockf() 系统调用。对于MH邮箱，锁定邮箱意味着锁定.mh_sequences文件，并且仅在影响它们的任何操作期间锁定单个邮件文件。

get_file (*重点*)
根据主机平台的不同，当返回的文件保持打开状态时，可能无法删除底层消息。

flush ( )
对MH邮箱的所有更改都立即应用，因此此方法不起作用。

close ( )
MH实例不保留任何打开的文件，所以此方法等同于unlock()。

也可以看看:

nmh - 消息处理系统
nmh的主页，原始**mh**的更新版本。

MH＆nmh：用户和程序员的电子邮件
有关**mh**和**nmh的** GPL授权书籍，以及关于邮箱格式的一些信息。

19.4.1.4。 Babyl

*类*mailbox.Babyl (*路径*, *工厂=无*, *创建= True*)
Mailbox Babyl格式邮箱的子类。参数 *工厂*是一个可调用的对象，它接受类似文件的消息表示（其行为如同以二进制模式打开）并返回一个自定义表示。如果 *工厂*是None ，BabylMessage则用作默认消息表示。如果*创建*是True，如果它不存在的邮箱已创建。

Babyl是由Emacs附带的Rmail邮件用户代理使用的单一文件邮箱格式。消息的开头由包含两个字符Control-Underscore（'\037'）和Control-L（'\014'）的行表示。消息的结尾由下一条消息的开始指示，或者在最后一条消息的情况下，包含一个Control-Underscore（'\037'）字符的行。

Babyl邮箱中的邮件有两组标题，原始标题和所谓的可见标题。可见标题通常是原始标题的一个子集，已被重新格式化或删节以获得更多吸引力。Babyl邮箱中的每封邮件也都有一个附带的*标签*列表 或用于记录有关邮件的额外信息的短字符串，并且邮箱中找到的所有用户定义标签列表都保存在Babyl选项部分。

BabylMailbox除了以下内容，实例还具有所有的方法：

get_labels ( )
返回邮箱中使用的所有用户定义标签的名称列表。

> **注意:** 检查实际消息以确定邮箱中存在哪些标签，而不是查阅Babyl选项部分中的标签列表，但每当邮箱被修改时都会更新Babyl部分。

一些Mailbox方法的实施Babyl值得特别注意：

get_file (*重点*)
在Babyl邮箱中，邮件的标题不会与邮件正文连续存储。要生成类似文件的表示形式，标题和正文将一起复制到io.BytesIO实例中，该实例具有与文件相同的API。因此，文件类对象确实独立于底层邮箱，但与字符串表示形式相比，不会节省内存。

lock ( )
unlock ( )
使用三种锁定机制 - 点锁定和（如果可用） flock()和lockf()系统调用。

也可以看看:

版本5 Babyl文件的格式
Babyl格式的规范。

用Rmail阅读邮件
Rmail手册提供关于Babyl语义的一些信息。

19.4.1.5。 MMDF

*类*mailbox.MMDF (*路径* , *工厂=无* , *创建= True*)
MailboxMMDF格式邮箱的子类。参数*工厂*是一个可调用的对象，它接受类似文件的消息表示（其行为如同以二进制模式打开）并返回一个自定义表示。如果 *工厂*是None ，MMDFMessage则用作默认消息表示。如果*创建*是True，如果它不存在的邮箱已创建。

MMDF是为多渠道备忘录分发机构（邮件传输代理）发明的单一文件邮箱格式。每条消息的格式与mbox消息相同，但在包含四个Control-A（'\001'）字符的行之前和之后括起来。与mbox格式一样，每条消息的开头都用前五个字符为“From”的行来表示，但由于额外的消息分隔符行阻止了，所以当存储消息时，附加出现的“From”不会转换为“> From”将这种情况误认为后续消息的开始。

一些Mailbox方法的实施MMDF值得特别注意：

get_file (*重点*)

使用文件调用后flush()或close()在 MMDF实例可能产生不可预测的结果或引发异常。

lock ( )
unlock ( )

使用三种锁定机制 - 点锁定和（如果可用） flock()和lockf()系统调用。

也可以看看:

从锡的mmdf手册页
新闻阅读器锡的文档中的MMDF格式规范。

MMDF
描述多通道备忘录分发工具的维基百科文章。

19.4.2。 Message对象

class mailbox. Message (*message* = *None*)

email.message模块的 一个子类Message。 mailbox.Message添加邮箱格式特定状态和行为的子类。

如果省略*消息*，则新实例将以默认的空状态创建。如果*消息*是一个email.message.Message实例，则其内容被复制; 此外，如果*消息*是Message实例，则尽可能转换任何特定于格式的信息。如果*消息*是一个字符串，一个字节字符串或一个文件，它应该包含一个**符合RFC 2822**的消息，它被读取和解析。文件应以二进制模式打开，但为了向后兼容，文本模式文件被接受。

子类提供的特定于格式的状态和行为有所不同，但通常情况下，只有不支持特定邮箱的属性（尽管推测这些属性是特定于特定邮箱格式的）。例如，单个文件邮箱格式的文件偏移量和基于目录的邮箱格式的文件名不会保留，因为它们仅适用于原始邮箱。但是，诸如消息是否被用户读取或标记为重要的状态被保留，因为它适用于消息本身。

没有要求Message使用Mailbox实例来表示使用实例检索的消息。在某些情况下，生成Message表示所需的时间和内存可能不可接受。对于这种情况，Mailbox实例还提供字符串和文件类型的表示，并且可以在Mailbox实例初始化时指定自定义消息工厂。

19.4.2.1。 MaildirMessage

class mailbox. MaildirMessage (*message* = *None*)

具有Maildir特定行为的消息。参数*消息*的含义与Message构造函数相同。

通常，邮件用户代理应用程序会在用户第一次打开并关闭邮箱后，将new子目录中的所有邮件移动 到cur子目录，并记录邮件是否旧，无论它们是否已被实际读取。每条消息cur都有一个“信息”部分添加到其文件名中，以存储有关其状态的信息。（某些邮件阅读器也可能为信息添加“信息”部分 new。）“信息”部分可以采取两种形式之一：它可以包含“2”，后跟一系列标准化标记（例如“2，FR “）或者它可能包含”1“，后面跟着所谓的实验信息。Maildir消息的标准标志如下：

旗	含义	说明
d	草案	在构图下
F	标记	标记为重要
P	通过	转发，重发或反弹
[R	回答	已回复
小号	看	读
Ť	丢弃	标记为随后删除

MaildirMessage 实例提供以下方法：

get_subdir ( )

返回“新”（如果消息应该存储在new 子目录中）或“cur”（如果消息应该存储在cur 子目录中）。

> **注意：** 消息通常是从移动new到cur后其邮箱被访问，无论是否该消息已被阅读。msg如果是，则 读取消息。 ″S″ in msg.get_flags() True

set_subdir (*subdir*)

设置消息应存储在的子目录中。参数*subdir* 必须是“新”或“当前”。

get_flags ( )

返回一个字符串，指定当前设置的标志。如果该消息与标准的Maildir格式符合，其结果是在零字母顺序或每一个的发生级联 'D'，'F'，'P'，'R'，'S'，和'T'。如果没有设置标志或“info”包含实验性语义，则返回空字符串。

set_flags (*标志*)

设置由标志指定的*标志*并取消设置所有其他*标志*。

add_flag (*标志*)

设置标志指定的*标志*而不更改其他标志。要一次添加多个标志，*标志*可能是一串多于一个字符。当前的“信息”被覆盖，不管它是否包含实验信息而不是标记。

remove_flag (*标志*)

取消设置标志指定的*标志*而不更改其他标志。要一次删除多个标志，*标志*可能是一串多于一个字符。如果“信息”包含实验信息而不是标记，则当前的“信息”不会被修改。

get_date ( )

将消息的交付日期作为表示自时代以来秒数的浮点数返回。

set_date (*日期*)

将消息的交付日期设置为*日期*，这是自纪元以来表示秒数的浮点数。

get_info ( )

返回包含消息“信息”的字符串。这对于访问和修改实验性的“信息”（即不是标志列表）很有用。

set_info (*info*)

将“info”设置为*info*，它应该是一个字符串。

当一个MaildirMessage是基于一个创建的实例 mboxMessage或MMDFMessage实例中，*状态*和*X-状态*标头省略，下面的转换发生：

结果状态	态oxMessage或MMDFMessage 状
“cur”子目录	O标志
F旗	F旗
R标志	一R旗
S旗	R标志
T旗	D标志

当基于MaildirMessage实例创建 MHMessage实例时，会发生以下转换：

结果状态	MHMessage 州
“cur”子目录	“看不见的”序列
“cur”子目录和S标志	没有“看不见”的序
F旗	标记”序列
R标志	“回答”了序列

当基于MaildirMessage实例创建 BabylMessage实例时，会发生以下转换：

结果状态	BabylMessage 州
“cur”子目录	“看不见”的标签
“cur”子目录和S标志	没有“看不见”的标签
P旗	“转发”或“重发”标签
R标志	“回答”了标签
T旗	“已删除”标签

19.4.2.2。 mboxMessage

class mailbox. mboxMessage (*message = None*)

具有特定于mbox的行为的消息。参数*消息*的含义与Message构造函数相同。

mbox邮箱中的邮件一起存储在单个文件中。发件人的信封地址和发送时间通常存储在一个以“From”开头的行中，该行用于指示邮件的开始，尽管mbox实现中该数据的确切格式有很大差异。指示消息状态的标志，例如它是否被读取或标记为重要，通常存储在 *Status*和*X-Status*标题中。

用于mbox消息的常规标志如下所示：

旗	含义	说明
[R	读	读
Ø	旧	以前由MUA检测到
d	删除	标记为随后删除
F	标记	标记为重要
一个	回答	已回复

“R”和“O”标志存储在*状态*标题中，“D”，“F”和“A”标志存储在*X状态*标题中。标志和标题通常以提到的顺序出现。

mboxMessage 实例提供以下方法：

get_from ( )

返回一个字符串，表示标记mbox邮箱中消息开头的“From”行。排除领先的“From”和尾随的换行符。

set_from (*from_* , *time_* = *None*)

将“From”行设置为*from_*，应该指定不带前导“From”或尾随换行符。为了方便，*time_*可以被指定，并且将被适当地格式化并被附加到*from_*。如果 指定了*time_*，它应该是一个 time.struct_time 实例，适合传递给 time.strftime() 或者 True （使用 time.gmtime() ）的元组。

get_flags ( )

返回一个字符串，指定当前设置的标志。如果该消息与传统格式符合，其结果是在每一个的零个或一个出现的以下顺序级联’R’，’O’，’D’，’F’，和’A’。

set_flags (*标志*)

设置由标志指定的*标志*并取消设置所有其他*标志*。参数*标志*应当在零个或多个的每一个的任何顺序串联’R’，’O’，’D’，’F’，和’A’。

add_flag (*标志*)

设置标志指定的*标志*而不更改其他标志。要一次添加多个标志，*标志*可能是一串多于一个字符。

remove_flag (*标志*)

取消设置标志指定的*标志*而不更改其他标志。要一次删除多个标志，*标志*可能是一串多于一个字符。

当基于mboxMessage实例创建 MaildirMessage实例时，将根据MaildirMessage实例的交付日期生成“发件人”行 ，并进行以下转换：

结果状态	MaildirMessage 州
R标志	S旗
O标志	“cur”子目录
D标志	T旗
F旗	F旗
一只旗	R标志

当基于mboxMessage实例创建 MHMessage实例时，会发生以下转换：

结果状态	MHMessage 州
R标志和O标志	没有“看不见”的序列
O标志	“看不见”序列
F旗	“标记”序列
一只旗	“回答”了序列

当基于mboxMessage实例创建 BabylMessage实例时，会发生以下转换：

结果状态	BabylMessage 州
R标志和O标志	没有“看不见”的标签
O标志	“看不见”的标签
D标志	“已删除”标签
一只旗	“回答”了标签

当基于Message实例创建MMDFMessage 实例时，“From”行被复制，所有标志直接对应：

结果状态	MMDFMessage 州
R标志	R标志
O标志	O标志
D标志	D标志
F旗	F旗
一只旗	一只旗

19.4.2.3。 MHMessage

class mailbox. MHMessage (*message = None*)

具有MH特定行为的消息。参数*消息*的含义与Message构造函数相同。

MH消息不支持传统意义上的标记或标志，但它们确实支持序列，它们是任意消息的逻辑分组。某些邮件阅读程序（尽管不是标准的**mh**和 **nmh**）使用序列的方式与标志与其他格式的使用方式大致相同，如下所示：

序列	说明
看不见	没有阅读，但以前由MUA检测到
回答	已回复
已标记	标记为重要

MHMessage 实例提供以下方法：

get_sequences ( )

返回包含此消息的序列名称列表。

set_sequences (*序列*)

设置包含此消息的序列列表。

add_sequence (*序列*)

将*序列*添加到包含此消息的序列列表中。

remove_sequence (*序列*)

从包含此消息的序列列表中删除*序列*。

当基于MHMessage实例创建 MaildirMessage实例时，会发生以下转换：

结果状态	MaildirMessage 州
“看不见的”序列	没有S标志
“回答”了序列	R标志
“标记”序列	F旗

当一个MHMessage是基于一个创建的实例 mboxMessage或MMDFMessage实例中，*状态* 和*X-状态*标头省略，下面的转换发生：

结果状态	mboxMessage或MMDFMessage 状态
“看不见的”序列	没有R标志
“回答”了序列	A旗
“标记”序列	F旗

当基于MHMessage实例创建 BabylMessage实例时，会发生以下转换：

结果状态	BabylMessage 州
“看不见的”序列	“看不见”的标签
“回答”了序列	“回答”了标签

19.4.2.4。 BabylMessage

class mailbox.BabylMessage (*message* = *None*)

具有Babyl特定行为的消息。参数*消息*的含义与Message构造函数相同。

某些消息标签，称为*属性*，按照惯例定义为具有特殊含义。属性如下：

标签	说明
看不见	没有阅读，但以前由MUA检测到
删除	标记为随后删除
提交	复制到另一个文件或邮箱
回答	已回复
转发	转发
编辑	由用户修改
愤恨	愤恨

默认情况下，Rmail仅显示可见标题。该BabylMessage 级，不过，使用原来的头，因为他们都比较齐全。如果需要，可以明确地访问可见标题。

BabylMessage 实例提供以下方法：

get_labels ( )

返回消息上的标签列表。

set_labels (*标签*)

将消息上的*标签*列表设置为*标签*。

add_label (*标签*)

将*标签*添加到邮件标签列表中。

remove_label (*标签*)

从邮件标签列表中删除*标签*。

get_visible ( )

返回一个Message实例，其头文件是消息的可见头文件，其正文为空。

set_visible (*可见*)

设置消息的可见标题是相同的标题 *消息*。 *可见*参数应该是Message 实例，email.message.Message实例，字符串或文件类对象（应该在文本模式下打开）。

update_visible ( )

当BabylMessage实例的原始标题被修改时，可见标题不会自动修改为对应。此方法更新可见标题如下：与相应的原始标题的每个可见首部设置为原始标题的值，没有相应的原始标题的每个可见头被除去，和任何的*日期*，*从*，*回复到*， *要*，*CC*和*主题*都存在于原始标题中，但不可见标题添加到可见标题中。

当基于BabylMessage实例创建 MaildirMessage实例时，会发生以下转换：

结果状态	MaildirMessage 州
“看不见”的标签	没有S标志
“已删除”标签	T旗
“回答”了标签	R标志
“转发”标签	P旗

当一个BabylMessage是基于一个创建的实例 mboxMessage或MMDFMessage实例中，*状态*和*X-状态*标头省略，下面的转换发生：

结果状态	mboxMessage或MMDFMessage 状态
“看不见”的标签	没有R标志
“已删除”标签	D标志
“回答”了标签	一只旗

当基于BabylMessage实例创建 MHMessage实例时，会发生以下转换：

结果状态	MHMessage 州
“看不见”的标签	“看不见的”序列
“回答”了标签	“回答”了序列

19.4.2.5。 MMDFMessage

class mailbox. MMDFMessage (*message* = *None*)

具有MMDF特定行为的消息。参数*消息*的含义与Message构造函数相同。

与mbox邮箱中的邮件一样，MMDF邮件与发件人地址和交货日期一起存储在以“From”开头的起始行中。同样，指示消息状态的标志通常存储在*Status*和*X-Status*标头中。

MMDF消息的常规标志与mbox消息的常规标志相同，如下所示：

旗	含义	说明
[R	读	读
Ø	旧	以前由MUA检测到
d	删除	标记为随后删除
F	标记	标记为重要
一个	回答	已回复

“R”和“O”标志存储在*状态*标题中，“D”，“F”和“A”标志存储在*X状态*标题中。标志和标题通常以提到的顺序出现。

MMDFMessage实例提供以下方法，它们与以下方法相同mboxMessage ：

get_from ( )

返回一个字符串，表示标记mbox邮箱中消息开头的“From”行。排除领先的“From”和尾随的换行符。

set_from (*from_* , *time_* = *None*)

将“From”行设置为*from_*，应该指定不带前导“From”或尾随换行符。为了方便，*time_*可以被指定，并且将被适当地格式化并被附加到*from_*。如果 指定了*time_*，它应该是一个 time.struct_time 实例，适合传递给 time.strftime() 或者 True （使用 time.gmtime() ）的元组。

get_flags ( )

返回一个字符串，指定当前设置的标志。如果该消息与传统格式符合，其结果是在每一个的零个或一个出现的以下顺序级联'R'，'O'，'D'，'F'，和'A'。

set_flags (*标志*)

设置由标志指定的*标志*并取消设置所有其他*标志*。参数*标志*应当在零个或多个的每一个的任何顺序串联'R'，'O'，'D'，'F'，和'A'。

add_flag (*标志*)

设置标志指定的*标志*而不更改其他标志。要一次添加多个标志，*标志*可能是一串多于一个字符。

remove_flag (*标志*)

取消设置标志指定的*标志*而不更改其他标志。要一次删除多个标志，*标志*可能是一串多于一个字符。

当基于MMDFMessage实例创建 MaildirMessage实例时，将根据MaildirMessage实例的交付日期生成“发件人”行 ，并进行以下转换：

结果状态	MaildirMessage **州**
R标志	S旗
O标志	“cur”子目录
D标志	T旗
F旗	F旗
一只旗	R标志

当基于MMDFMessage实例创建 MHMessage实例时，会发生以下转换：

结果状态	MHMessage **州**
R标志和O标志	没有“看不见”的序列
O标志	“看不见的”序列
F旗	“标记”序列
一只旗	“回答”了序列

当基于MMDFMessage实例创建 BabylMessage实例时，会发生以下转换：

结果状态	BabylMessage **州**
R标志和O标志	没有“看不见”的标签
O标志	“看不见”的标签
D标志	“已删除”标签
一只旗	“回答”了标签

当基于MMDFMessage实例创建 mboxMessage实例时，“From”行被复制，所有标志直接对应：

结果状态	mboxMessage **州**
R标志	R标志
O标志	O标志
D标志	D标志
F旗	F旗
一只旗	一只旗

19.4.3。例外

以下异常类在mailbox模块中定义：

*异常*mailbox. Error

所有其他模块特定例外的基础类。

*异常*mailbox.NoSuchMailboxError

在希望找到邮箱但未找到邮箱时引发，例如在实例化具有Mailbox不存在的路径（并且*创建*参数设置为False）的 子类时，或打开不存在的文件夹时引发。

*异常*mailbox.NotEmptyError

当邮箱不是空的，但预计是，例如删除包含邮件的文件夹时引发。

*异常*mailbox.ExternalClashError

当程序控制范围之外的某些与邮箱相关的情况导致无法继续时，例如无法获取另一个程序已经拥有锁的锁，或者存在唯一生成的文件名时，就会引发此问题。

*异常*mailbox.FormatError

当文件中的数据无法分析时引发，MH 例如实例试图读取损坏的.mh_sequences文件。

19.4.4。示例

将邮箱中的所有邮件的主题打印出来似乎很有趣的简单示例：

```
import mailbox
for message in mailbox.mbox('~/mbox'):
    subject = message['subject']       # Could possibly be None.
    if subject and 'python' in subject.lower():
        print(subject)
```

要将Babyl邮箱中的所有邮件复制到MH邮箱，请转换所有可以转换的格式特定信息：

```
import mailbox
destination = mailbox.MH('~/Mail')
destination.lock()
for message in mailbox.Babyl('~/RMAIL'):
    destination.add(mailbox.MHMessage(message))
destination.flush()
destination.unlock()
```

本示例将来自多个邮件列表的邮件分类到不同的邮箱中，小心避免由于其他程序的并发修改导致邮件损坏，由于程序中断而导致邮件丢失或由于邮箱中的邮件格式不正确而提前终止：

```
import mailbox
import email.errors

list_names = ('python-list', 'python-dev', 'python-bugs')

boxes = {name: mailbox.mbox('~/email/%s' % name) for name in list_names}
inbox = mailbox.Maildir('~/Maildir', factory=None)

for key in inbox.iterkeys():
    try:
        message = inbox[key]
    except email.errors.MessageParseError:
        continue                # The message is malformed. Just leave it.
```

```
    for name in list_names:
        list_id = message['list-id']
        if list_id and name in list_id:
            # Get mailbox to use
            box = boxes[name]

            # Write copy to disk before removing original.
            # If there's a crash, you might duplicate a message, but
            # that's better than losing a message completely.
            box.lock()
            box.add(message)
            box.flush()
            box.unlock()

            # Remove original message
            inbox.lock()
            inbox.discard(key)
            inbox.flush()
            inbox.unlock()
            break               # Found destination, so stop looking.

for box in boxes.itervalues():
    box.close()
```

19.5。 mimetypes- 将文件名映射到MIME类型

源代码： Lib / mimetypes.py

该mimetypes模块在文件名或URL与与文件扩展名关联的MIME类型之间进行转换。从文件名到MIME类型以及从MIME类型到文件扩展名提供转换; 编码不适用于后面的转换。

该模块提供了一个类和许多便利功能。这些功能是这个模块的正常接口，但是一些应用程序也可能对这个类感兴趣。

下面介绍的功能为该模块提供了主要接口。如果模块尚未初始化，他们将init()依靠init()设置的信息进行调用。

mimetypes.guess_type（*url*，*strict* = *True* ）

根据由*url*给出的文件名或URL猜测文件的类型。返回值是一个元组，如果类型不能被猜测（丢失或未知的后缀）或表单的字符串（可用于MIME *内容类型*头），则*类型*为元组。(type, encoding)None’type/subtype’

*编码*是None无编码或用于编码的程序的名称（例如**压缩**或**gzip**）。编码是适合用作一个*内容编码*报头，**不**作为 *内容传送编码*标头。映射是由表驱动的。编码后缀区分大小写。类型后缀首先是区分大小写，然后区分大小写。

可选的*strict*参数是一个标志，指定已知MIME类型的列表是否仅限于在IANA注册的官方类型。当*严格*为True（默认值），只有IANA类型的支持; 当 *严格的*是False，一些额外的非标准，但常用的MIME类型也被识别。

mimetypes.guess_all_extensions（*type*，*strict* = *True* ）

猜扩展基于其MIME类型给出一个文件，*类型*。返回值是提供所有可能的文件扩展名的字符串列表，包括前导点（’.’）。延伸部不保证已与任何特定的数据流相关联，但将被映射到MIME类型*类型*通过guess_type()。

可选的*严格*参数与该guess_type()函数具有相同的含义。

mimetypes.guess_extension（*type*，*strict* = *True* ）

猜测给出的扩展基于其MIME类型的文件，*类型*。返回值是一个给出文件扩展名的字符串，包括前导点（’.’）。扩展不能保证已与任何特定的数据流相关联，但将被映射到MIME类型*类型*通过 guess_type()。如果没有扩展名可以猜出*类型*，None则返回。

可选的*严格*参数与该guess_type()函数具有相同的含义。

一些额外的功能和数据项目可用于控制模块的行为。

mimetypes.init（*files* = *None* ）

初始化内部数据结构。如果给定，*文件*必须是一系列文件名，应该用来扩充默认的类型映射。如果省略，则使用的文件名取自knownfiles; 在Windows上，加载当前的注册表设置。在文件中命名的每个文件或者 优先于在它之前命名的*文件*knownfiles。 init()允许重复呼叫 。

指定*文件*的空列表将防止应用系统默认值：只有已知值将从内置列表中呈现。

*在版本3.2中更改：*以前，Windows注册表设置被忽略。

mimetypes.**read_mime_types**(*文件名*)

加载文件*filename*中给出的类型映射（如果存在）。类型映射作为字典映射文件扩展名（包括前导点（'.'））返回到表单的字符串'type/subtype'。如果文件的文件*名*不存在或无法读取，None则返回。

mimetypes.**add_type**(*type*, *ext*, *strict = True*)

根据MIME类型添加映射*类型*的扩展名*EXT*。当扩展名已知时，新的类型将取代旧的类型。当类型已知时，扩展名将被添加到已知扩展名列表中。

当*严格*为True（默认值），映射将被添加到官方的MIME类型，否则非标准的。

mimetypes.**inited**

指示全局数据结构是否已被初始化的标志。这是设置为True通过init()。

mimetypes.**knownfiles**

通常安装的类型映射文件名称列表。这些文件通常以mime.types不同的软件包命名 并安装在不同的位置。

mimetypes.**suffix_map**

字典映射后缀到后缀。这用于识别编码文件和类型由相同扩展名表示的编码文件。例如，.tgz扩展名被映射为.tar.gz 允许单独识别编码和类型。

mimetypes.**encodings_map**

字典映射文件扩展名到编码类型。

mimetypes.**types_map**

将MIME类型的字典映射文件扩展名。

mimetypes.**common_types**

字典映射文件扩展名到非标准的，但通常发现的MIME类型。

模块的示例用法：

```
>>> import mimetypes
>>> mimetypes.init()
>>> mimetypes.knownfiles
['/etc/mime.types', '/etc/httpd/mime.types', ... ]
>>> mimetypes.suffix_map['.tgz']
'.tar.gz'
>>> mimetypes.encodings_map['.gz']
'gzip'
>>> mimetypes.types_map['.tgz']
'application/x-tar-gz'
```

19.5.1。MimeTypes对象

的MimeTypes类可以是用于其中可能希望多于一个MIME类型的数据库应用程序有用的; 它提供了一个类似于mimetypes模块的接口 。

class mimetypes.MimeTypes (*filenames = ( )* , *strict = True*)

这个类表示一个MIME类型的数据库。默认情况下，它提供与该模块其余部分相同的数据库访问权限。初始数据库是模块提供的副本，可以通过mime.types使用read() or readfp() 方法将附加样式文件加载到数据库中进行扩展 。如果不需要默认数据，映射字典也可以在加载附加数据之前清除。

可选的*文件名*参数可用于使其他文件被加载到默认数据库的“顶部”。

suffix_map

字典映射后缀到后缀。这用于识别编码文件和类型由相同扩展名表示的编码文件。例如，.tgz扩展名被映射为.tar.gz 允许单独识别编码和类型。这最初是suffix_map模块中定义的全局副本。

encodings_map

字典映射文件扩展名到编码类型。这最初是encodings_map模块中定义的全局副本。

types_map

包含两个字典的元组，将文件扩展名映射到MIME类型：第一个字典用于非标准类型，第二个字典用于标准类型。它们由common_types和 初始化types_map。

types_map_inv

包含两个字典的元组，将MIME类型映射到文件名扩展名列表：第一个字典用于非标准类型，第二个字典用于标准类型。它们由common_types和初始化 types_map。

guess_extension (*type* , *strict = True*)

与guess_extension()函数类似，使用存储为对象一部分的表。

guess_type (*url* , *strict = True*)

与guess_type()函数类似，使用存储为对象一部分的表。

guess_all_extensions (*type* , *strict = True*)

与guess_all_extensions()函数类似，使用存储为对象一部分的表。

read (*filename* , *strict = True*)

从名为*filename*的文件加载MIME信息。这用于readfp()解析文件。

如果*严格的*是True，信息将被添加到标准类型列表，其他非标准类型的列表。

readfp (*fp* , *strict = True*)

从打开的文件*fp*中加载MIME类型信息。该文件必须具有标准mime.types文件的格式。

如果*严格的*是True，信息将被添加到标准类型列表中，其他非标准类型的列表。

read_windows_registry (*strict = True*)

从Windows注册表中加载MIME类型信息。可用性：Windows。

如果*严格的*是`True`，信息将被添加到标准类型列表中，其他非标准类型的列表。

3.2版本中的新功能

19.6。base64- Base16，Base32，Base64，Base85数据编码

源代码： Lib / base64.py

该模块提供将二进制数据编码为可打印的ASCII字符并将这些编码解码回二进制数据的功能。它为中指定的编码提供编码和解码功能 **RFC 3548**定义了Base16，Base32和Base64算法，以及事实上的标准Ascii85和Base85编码。

该 **RFC 3548**编码适用于编码二进制数据，以便它可以通过电子邮件安全发送，用作URL的一部分，或作为HTTP POST请求的一部分包含在其中。编码算法与 **uuencode**程序不同。

这个模块提供了两个接口。现代接口支持将类似字节的对象编码为ASCII `bytes`，并解码类似字节的对象或包含ASCII的字符串`bytes`。两个base-64字母都在中定义**支持RFC 3548**（正常，URL和文件系统安全）。

传统界面不支持字符串解码，但它确实提供了用于编码和解码文件对象的功能。它仅支持Base64标准字母表，并且每按76个字符添加新行**RFC 2045**。请注意，如果你正在寻找**RFC 2045**支持您可能想要查看`email` 包。

*在版本3.3中进行了更改：*现在接口的解码功能现在可接受仅ASCII字符的Unicode字符串。

*版本3.4中更改：*此模块中的所有编码和解码功能现在都可以接受任何类似字节的对象。增加了Ascii85 / Base85支持。

现代界面提供：

base64.b64encode（*s*，*altchars* = *None* ）

编码类字节对象 *小号*使用Base64和返回编码 `bytes`。

可选的*altchars*必须是长度至少为2 的字节状对象（忽略附加字符），它指定+和/字符的替代字母。这允许应用程序例如生成URL或文件系统安全的Base64字符串。默认值是None，使用标准的Base64字母表。

base64.b64decode（*s*，*altchars* = *None*，*validate* = *False* ）

解码Base64编码类字节对象或ASCII字符串 *小号*并返回解码`bytes`。

可选的*altchars*必须是一个类似字节的对象或至少长度为2的ASCII字符串（忽略附加字符），它指定使用替代字母表而不是+和/字符。

一`binascii.Error`，如果引发异常*小号*不正确填充。

如果*validate*是`False`（缺省值），则在填充检查之前，将丢弃正常的base-64字母表和替代字母表中的字符。如果*验证*是`True`，则输入中的这些非字母字符将导致a `binascii.Error`。

base64.standard_b64encode（*s* ）

编码字节状物体 *小号*使用标准的Base64字母，并返回该编码bytes。

base64.standard_b64decode（*s*）

解码字节状物品或ASCII字符串*小号*使用标准的Base64字母，并返回解码bytes。

base64.urlsafe_b64encode（*s*）

编码字节状物体 *小号*使用URL-和文件系统安全的字母表，其中替代-，而不是+和_，而不是/标准的Base64字母，并返回编码bytes。结果仍然可以包含=。

base64.urlsafe_b64decode（*s*）

解码字节状物品或ASCII字符串*小号* 使用URL-和文件系统安全的字母表，其中替代-，而不是+和_，而不是 /标准的Base64字母，并返回解码 bytes。

base64.b32encode（*s*）

编码类字节对象 *小号*使用Base32并返回所述经编码bytes。

base64.b32decode（*s*，*casefold = False*，*map01 = None*）

解码Base32编码类字节对象或ASCII字符串*小号*并返回解码bytes。

可选*casefold*是一个标志，指定小写字母是否可以作为输入。为了安全起见，默认是False。

RFC 3548允许将数字0（零）可选地映射到字母O（哦），并且可选地将数字1（一）映射到字母I（眼睛）或字母L（el）。可选参数 *map01*如果不是None，指定数字1应该映射到哪个字母（当 *map01*不是None，数字0总是映射到字母O）。出于安全目的，默认值是None0，因此输入中不允许使用0和1。

binascii.Error如果*s*未正确填充或者输入中存在非字母字符，则会引发A.

base64.b16encode（*s*）

编码类字节对象 *小号*使用Base16并返回所述经编码bytes。

base64.b16decode（*s*，*casefold = False*）

解码Base16编码类字节对象或ASCII字符串*小号*并返回解码bytes。

可选*casefold*是一个标志，指定小写字母是否可以作为输入。为了安全起见，默认是False。

binascii.Error如果*s*未正确填充或者输入中存在非字母字符，则会引发A.

base64.a85encode（*b*，*，foldspaces = False*，*wrapcol = 0*，*pad = False*，*adobe = False*）

编码类字节对象 *b*使用ASCII85并返回所述经编码bytes。

*foldspaces*是一个可选标志，它使用特殊的短序列'y'而不是4个连续的空格（ASCII 0x20），由'btoa'支持。“标准”Ascii85编码不支持此功能。

*wrapcol*控制输出是否应该b'\n'添加新行（）字符。如果这不是零，则每个输出行最多只有很多字符。

*鍵盘*控制在编码之前输入是否填充到4的倍数。请注意，该btoa实施始终填充。

*adobe*控制编码的字节序列是否使用<~ and 框架，~>由Adobe实现使用。

3.4版新增功能

base64.a85decode（*b*，*，*foldspaces = False*，*adobe = False*，*ignorechars = b'\ t \ n \ r \ v'）

解码Ascii85编码的类似字节的对象或ASCII字符串*b*并返回解码bytes。

*折叠空间*是一个标志，指定是否应将'y'短序列作为4个连续空格（ASCII 0x20）的缩写。“标准”Ascii85编码不支持此功能。

*adobe*控制输入序列是否采用Adobe Ascii85格式（即用<~和~>构成）。

*ignorechars*应该是一个类似字节的对象或ASCII字符串，其中包含要从输入忽略的字符。这应该只包含空白字符，并且默认情况下包含ASCII中的所有空白字符。

3.4版新增功能

base64.b85encode（*b*，*pad = False*）

使用base85 编码类似字节的对象 *b*（用于例如git风格的二进制差异）并返回编码bytes。

如果*pad*为真，则输入将被填充，b'\0' 所以其长度在编码之前是4个字节的倍数。

3.4版新增功能

base64.b85decode（*b*）

解码base85编码的类字节对象或ASCII字符串*b*并返回解码bytes。如有必要，填充被隐式删除。

3.4版新增功能

传统界面：

base64.decode（*输入*，*输出*）

解码二进制*输入*文件的内容并将生成的二进制数据写入*输出*文件。*输入*和*输出*必须是文件对象。*输入*将被读取直到input.readline()返回一个空字节对象。

base64.decodebytes（*s*）

解码类字节的对象 *s*，它必须包含一行或多行base64编码数据，并返回解码bytes。

版本3.1中的新功能。

base64.decodestring（*s*）

不推荐使用的别名decodebytes()。

自3.1版以来已弃用。

base64.encode（*输入*，*输出*）

编码二进制*输入*文件的内容并将生成的base64编码数据写入*输出*文件。*输入*和*输出*必须是文件对象。*输入*将被读取直到input.read()返回一个空字节对象。在每输出76个字节后encode()插入一个换行符（b'\n'），并确保输出始终以换行符结束，如**RFC 2045**（MIME）。

base64.**encodebytes**(*s*)

编码类字节对象 *小号*，它可以包含任意的二进制数据，并返回bytes包含base64编码数据，用换行（b'\n'每76个字节的输出的后插入），并且有确保一个尾随换行，按照**RFC 2045**（MIME）。

版本3.1中的新功能。

base64.**encodestring**(*s*)

不推荐使用的别名encodebytes()。

自3.1版以来已弃用。

模块的示例用法：

```
>>> import base64
>>> encoded = base64.b64encode(b'data to be encoded')
>>> encoded
b'ZGF0YSBOobyBiZSBlbmNvZGVk'
>>> data = base64.b64decode(encoded)
>>> data
b'data to be encoded'
```

也可以看看:

模 binascii

支持包含ASCII到二进制和二进制到ASCII转换的模块。

RFC 1521 - MIME（多用途Internet邮件扩展）第一部分：指定和描述Internet消息体格式的机制

第5.2节“Base64 Content-Transfer-Encoding”提供了base64编码的定义。

19.7。binhex- 编码和解码binhex4文件

源代码： Lib / binhex.py

该模块以binhex4格式对文件进行编码和解码，格式允许以ASCII格式表示Macintosh文件。只处理数据分支。

该`binhex`模块定义了以下功能：

`binhex.`**binhex**（*输入*，*输出*）
将文件名*输入*的二进制文件转换为binhex文件*输出*。的 *输出*参数可以是一个文件名或文件对象（支撑的任何对象`write()`和`close()`方法）。

`binhex.`**hexbin**（*输入*，*输出*）
解码binhex文件*输入*。*输入*可能是文件名或文件类对象的支持`read()`和`close()`方法。生成的文件被写入名为*output*的文件，除非参数是`None`在这种情况下从binhex文件读取输出文件名。

以下例外也被定义：

异常`binhex.`**Error**
当某些东西不能使用binhex格式进行编码时（例如，文件名太长而无法放入文件名字段），或输入的binhex数据编码不正确时引发异常。

也可以看看:

模 `binascii`
支持包含ASCII到二进制和二进制到ASCII转换的模块。

19.7.1。笔记

有一个替代的，更强大的编码器和解码器接口，请参阅源代码以获取详细信息。

如果您在非Macintosh平台上编码或解码文本文件，它们仍将使用旧的Macintosh新行约定（回车作为行尾）。

19.8。binascii- 二进制和ASCII之间的转换

该binascii模块包含许多在二进制和各种ASCII编码二进制表示之间转换的方法。通常情况下，你不会直接使用这些功能，但使用的包装模块一样uu， base64或者binhex相反。该binascii模块包含用C语言编写的低级函数，以获得更高级别模块使用的更高速度。

注意： a2b_*函数接受只包含ASCII字符的Unicode字符串。其它功能只接受字节状物体（如bytes，bytearray以及支持所述缓冲协议其他对象）。

*在版本3.3中更改：*现在只有ASCII码的Unicode字符串被a2b_*函数接受。

该binascii模块定义了以下功能：

binascii.**a2b_uu**（*字符串*）

将一行uuencoded数据转换回二进制并返回二进制数据。行通常包含45（二进制）字节，除了最后一行。行数据后面可能会出现空格。

binascii.**b2a_uu**（*数据*）

将二进制数据转换为一行ASCII字符，返回值是转换后的行，包括换行符char。*数据*的长度最多应为45。

binascii.**a2b_base64**（*字符串*）

将一段base64数据转换回二进制并返回二进制数据。一次可以传递多个行。

binascii.**b2a_base64**（*data*，*，*newline* = *True*）

将二进制数据转换为base64编码中的一行ASCII字符。返回值是转换后的行，如果*换行符*为true，则包括换行符字符。该功能的输出符合**RFC 3548**。

*在版本3.6中更改：*添加了新*行*参数。

binascii.**a2b_qp**（*data*，*header* = *False*）

将一段引用可打印的数据转换回二进制并返回二进制数据。一次可以传递多个行。如果可选参数*头存*在且为true，则下划线将被解码为空格。

binascii.**b2a_qp**（*data*，*quotetabs* = *False*，*istext* = *True*，*header* = *False*）

将二进制数据转换为带引号的可打印编码中的ASCII字符行。返回值是转换后的行。如果可选参数*quotetabs*存在且为true，则*所有制表*符和空格都将被编码。如果可选参数*istext*存在并且为true，则不对新行进行编码，但将对后面的空白进行编码。如果可选参数*头存*在且为true，则根据RFC1522将空格编码为下划线。如果可选参数*头存*在且为假，则换行符也将被编码; 否则换行转换可能会破坏二进制数据流。

binascii.**a2b_hqx**（*字符串*）

将binhex4格式的ASCII数据转换为二进制，而不进行RLE解压缩。该字符串应该包含完整数量的二进制字节，或者（在binhex4数据的最后一部分的情况下）剩余的位为零。

binascii.**rledecode_hqx**(*数据*)

根据binhex4标准对数据执行RLE解压缩。该算法0x90在一个字节之后用作重复指示符，然后是一个计数。计数值0指定一个字节值0x90。该例程返回解压缩的数据，除非数据输入数据以孤立重复指示符结束，在这种情况下Incomplete引发异常。

*在版本3.2中更改：*仅接受字节串或字节数组对象作为输入。

binascii.**rlecode_hqx**(*数据*)

对*数据*执行binhex4样式的RLE压缩并返回结果。

binascii.**b2a_hqx**(*数据*)

执行hexbin4二进制转ASCII转换并返回结果字符串。参数应该已经是RLE编码的，并且长度可以被3除尽（可能除了最后一个片段）。

binascii.**crc_hqx**(*数据*, *值*)

计算的一个16位的CRC值*的数据*，从*值*作为初始CRC，并返回结果。这使用CRC-CCITT多项式 $x^{16} + x^{12} + x^5 + 1$，通常表示为0x1021。该CRC以binhex4格式使用。

binascii.**crc32**(*data*[, *value*])

计算*数据*的32位校验和CRC-32，从最初的CRC *值*开始*计算*。默认的初始CRC是零。该算法与ZIP文件校验和一致。由于该算法被设计用作校验和算法，因此不适合用作通用哈希算法。使用方法如下：

```
print(binascii.crc32(b"hello world"))
# Or, in two pieces:
crc = binascii.crc32(b"hello")
crc = binascii.crc32(b" world", crc)
print('crc32 = {:#010x}'.format(crc))
```

*在版本3.0中更改：*结果始终未签名。要在所有Python版本和平台上生成相同的数值，请使用。crc32(data) & 0xffffffff

binascii.**b2a_hex**(*数据*)

binascii.**hexlify**(*数据*)

返回二进制*数据*的十六进制表示。*数据的*每个字节都 被转换成相应的2位十六进制表示。返回的字节对象因此是*数据*长度的两倍。

binascii.**a2b_hex**(*hexstr*)

binascii.**unhexlify**(*hexstr*)

返回由十六进制字符串*hexstr*表示的二进制数据。这个函数是与之相反的b2a_hex()。*hexstr*必须包含偶数个十六进制数字（可以是大写或小写），否则 Error会引发异常。

*异常*binascii.**Error**

异常引发的错误。这些通常是编程错误。

*异常*binascii.**Incomplete**

不完整的数据引发异常。这些通常不是编程错误，但可以通过读取更多的数据并再次尝试来处理。

也可以看看:

模 base64
支持基于16,32,64和85的RFC兼容base64风格编码。

模 binhex
支持Macintosh上使用的binhex格式。

模 uu
支持Unix上使用的UU编码。

模 quopri
支持MIME电子邮件中使用的引用打印编码。

19.9。 quopri- 对MIME quoted-printable数据进行编码和解码

源代码： Lib / quopri.py

该模块执行quoted-printable传输编码和解码，如定义 **RFC 1521**：“MIME（多用途互联网邮件扩展）第一部分：指定和描述互联网消息体格式的机制”。带引号的可打印编码设计用于非可印刷字符相对较少的数据; base64如果发送图形文件时有很多这样的字符，则通过模块可用的base64编码方案 更加紧凑。

quopri. decode（ *input* ， *output* ， *header = False* ）

解码*输入*文件的内容并将结果解码的二进制数据写入*输出*文件。*输入*和*输出*必须是二进制文件对象。如果可选参数*标题*存在且为true，则下划线将被解码为空格。这用于解码“Q”编码的头文件**RFC 1522**：“MIME（多用途Internet邮件扩展）第二部分：非ASCII文本的邮件标题扩展”。

quopri. encode（ *输入* ， *输出* ， *quotetabs* ， *header = False* ）

对*输入*文件的内容进行编码，并将生成的引用可打印数据写入*输出*文件。*输入*和*输出*必须是 二进制文件对象。*quotetabs*，一个非可选标志，用于控制是否对嵌入的空格和制表符进行编码; 当它为真时，它编码这样的嵌入空白，并且当它错误时，它们就不会被编码。请注意，出现在行尾的空格和制表符总是按照编码进行编码**RFC 1521**。 *头*是一个标志，它控制空格是否按照下划线编码**RFC 1522**。

quopri. decodestring（ *s* ， *header = False* ）

就像decode()，它接受一个源bytes并返回相应的解码bytes。

quopri. encodestring（ *s* ， *quotetabs = False* ， *header = False* ）

就像encode()，它接受一个源bytes并返回相应的编码bytes。默认情况下，它向函数的*quotetabs*参数发送一个 False值。 encode()

也可以看看:

模 base64

对MIME base64数据进行编码和解码

19.10。 uu- 编码和解码uuencode文件

源代码： Lib / uu.py

该模块以uuencode格式对文件进行编码和解码，允许任意二进制数据通过仅ASCII连接进行传输。无论在哪里需要一个文件参数，这些方法都会接受一个类似文件的对象。为了向后兼容，还接受包含路径名的字符串，并打开相应的文件以便读取和写入; 路径名'-'被理解为意味着标准输入或输出。但是，此接口已弃用; 调用者最好打开文件本身，并确保在需要时该模式在Windows 上'rb'或'wb'在Windows上。

此代码由Lance Ellinghouse提供，并由Jack Jansen修改。

该uu模块定义了以下功能：

uu.encode（*in_file*，*out_file*，*name = None*，*mode = None* ）

将Uuencode文件*in_file*放入文件*out_file中*。uuencoded文件将具有指定*名称*和*模式*的标题作为解码文件结果的默认值。缺省默认值取自*in_file中*，或'-'和0o666分别。

uu.decode（*in_file*，*out_file = None*，*mode = None*，*quiet = False* ）

该调用解码uuencoded文件*in_file*，将结果放在文件 *out_file上*。如果*out_file*是路径名，则*mode*在必须创建文件时使用来设置权限位。*out_file*和*mode的*默认值来自uuencode头文件。但是，如果头中指定的文件已经存在，uu.Error则会引发a。

decode()如果输入是由不正确的uuencoder产生的，并且Python可以从该错误中恢复，则可以向标准错误发出警告。将*安静*设置 为真实值会使此警告消声。

*异常*uu.Error

它的子类Exception可以uu.decode()在各种情况下引发，如上面所描述的，但也包括格式错误的头文件或截断的输入文件。

也可以看看:

模 binascii

支持包含ASCII到二进制和二进制到ASCII转换的模块。

20.结构化标记处理工具

Python支持各种模块来处理各种形式的结构化数据标记。这包括使用标准通用标记语言（SGML）和超文本标记语言（HTML）的模块以及使用可扩展标记语言（XML）的多个接口。

20.1。html- 超文本标记语言支持

源代码： Lib / html / __ init__.py

该模块定义了用于操作HTML的实用程序。

html. escape（ *s* , *quote* = *True* ）

转换角色&，<并>在串*小号*到HTML安全序列。如果您需要显示可能包含HTML中的此类字符的文本，请使用此选项。如果可选标志*引用*为真，则字符（ ″ ）和（ ’ ）也会被翻译; 这有助于包含在由引号分隔的HTML属性值中，如in 。<a href="...">

3.2版本中的新功能

html. unescape（ *s* ）

转换所有命名和数字字符引用（ 例如> ， > ，&x3e;在字符串 ）*小号*到对应的Unicode字符。此函数使用由HTML 5标准定义的有效和无效字符引用的规则，而。list of HTML 5 named character references

3.4版新增功能

html包中的子模块是：

- html.parser - 具有宽松解析模式的HTML / XHTML解析器
- html.entities - HTML实体定义

20.2。html.parser- 简单的HTML和XHTML分析器

源代码： Lib / html / parser.py

该模块定义了一个类HTMLParser，它用作解析HTML（超文本标记语言）和XHTML格式的文本文件的基础。

class html.parser.HTMLParser（ ***，*convert_charrefs = True* ）

创建一个能解析无效标记的解析器实例。

如果*convert_charrefs*是True（默认），则所有字符引用（除了script/ style元素中的那些）都会自动转换为相应的Unicode字符。

一个HTMLParser实例被送到HTML数据和遇到的开始标记，结束标记，文本，注释和其他标记元素时，调用处理方法。用户应该继承HTMLParser并覆盖其方法来实现所需的行为。

这个解析器不会检查结束标签是否匹配开始标签，也不会检查通过关闭外部元素隐式关闭的元素的结束标签处理程序。

*在版本3.4中进行了更改：*添加了*convert_charrefs*关键字参数。

*在版本3.5中更改：*参数*convert_charrefs*的默认值是现在True。

20.2.1。HTML解析器应用程序示例

作为一个基本示例，下面是一个简单的HTML解析器，它使用 HTMLParser该类在遇到它们时打印出开始标记，结束标记和数据：

```
from html.parser import HTMLParser

class MyHTMLParser(HTMLParser):
    def handle_starttag(self, tag, attrs):
        print("Encountered a start tag:", tag)

    def handle_endtag(self, tag):
        print("Encountered an end tag :", tag)

    def handle_data(self, data):
        print("Encountered some data  :", data)

parser = MyHTMLParser()
parser.feed('<html><head><title>Test</title></head>'
            '<body><h1>Parse me!</h1></body></html>')
```

输出将是：

```
Encountered a start tag: html
Encountered a start tag: head
Encountered a start tag: title
Encountered some data  : Test
Encountered an end tag : title
Encountered an end tag : head
Encountered a start tag: body
Encountered a start tag: h1
Encountered some data  : Parse me!
Encountered an end tag : h1
Encountered an end tag : body
Encountered an end tag : html
```

20.2.2。HTMLParser方法

HTMLParser 实例具有以下方法：

HTMLParser.**feed**(*数据*)
将一些文本提供给解析器。它是由完整的元素组成的，不完整的数据被缓冲，直到更多的数据被馈送或被 close() 调用。 *数据*必须是str。

HTMLParser.**close**()
强制处理所有缓冲数据，就好像它后面跟着一个文件结束标记。该方法可以由派生类重新定义，以在输入末尾定义附加处理，但重新定义的版本应始终调用HTMLParser基类方法close()。

HTMLParser.**reset**()
重置实例。丢失所有未处理的数据。这在实例化时被隐含地调用。

HTMLParser.**getpos**()
返回当前行号和偏移量。

HTMLParser.**get_starttag_text**()
返回最近打开的开始标签的文本。这通常不是结构化处理所需要的，但对于处理HTML“部署”或重新生成输入时只需要很少的修改（可以保留属性之间的空白等）可能很有用。

当遇到数据或标记元素时，将调用以下方法，并且这些方法旨在在子类中重写。基类实现什么都不做（除了handle_startendtag()）：

HTMLParser.**handle_starttag**(*tag*, *attrs*)
调用此方法来处理标记（例如）的开始。

的*标签*参数被转换为较低的情况下，标记的名称。该*ATTRS* 参数是一个列表包含标签的发现里面的属性对括号。该*名称*将被转换为小写，并且该*值中的*引号已被删除，并且字符和实体引用已被替换。(name, value)<>

例如，对于标签，这个方法将被称为。<A HREF="https://www.cwi.nl/">handle_starttag('a', [('href', 'https://www.cwi.nl/')])

来自所有实体的引用html.entities在属性值中被替换。

HTMLParser.handle_endtag(*标签*)

调用此方法来处理元素的结束标记（例如 ）。

的*标签*参数被转换为较低的情况下，标记的名称。

HTMLParser.handle_startendtag(*tag*, *attrs*)

与此类似handle_starttag()，但在解析器遇到XHTML样式的空标记（）时调用。这个方法可能被需要这个特定词汇信息的子类覆盖；默认实现只是调用和。handle_starttag()handle_endtag()

HTMLParser.handle_data(*数据*)

这种方法被调用来处理任意数据（例如，文本节点和内容<script>...</script>和<style>...</style>）。

HTMLParser.handle_entityref(*名字*)

调用此方法来处理表单&name;（例如>）的命名字符引用，其中*name*是一般实体引用（例如'gt'）。如果*convert_charrefs*为，则此方法从不调用True。

HTMLParser.handle_charref(*名字*)

调用此方法来处理表单&#NNN;和的十进制和十六进制数字字符引用&#xNNN;。例如，十进制等价物>是is >，而十六进制是>；在这种情况下，该方法将收到'62'或'x3E'。如果*convert_charrefs*为，则此方法从不调用True。

HTMLParser.handle_comment(*数据*)

遇到注释时会调用此方法（例如<!--comment-->）。

例如，注释将导致使用参数调用此方法。<!-- comment -->' comment '

Internet Explorer条件注释（condcoms）的内容也将发送到此方法，因此，此方法将收到。<!--[if IE 9]>IE9-specific content<![endif]-->'[if IE 9]>IE9-specific content<![endif]'

HTMLParser.handle_decl(*decl*)

调用此方法来处理HTML doctype声明（例如）。<!DOCTYPE html>

的*DECL*参数将是内部的声明中的全部内容<!...>的标记（例如）。'DOCTYPE html'

HTMLParser.handle_pi(*数据*)

在遇到处理指令时调用的方法。该*数据*参数将包含整个处理指令。例如，对于处理指令，此方法将被称为。它打算被派生类覆盖；基类实现什么都不做。<?proc color='red'>handle_pi("proc color='red'")

> **注意：** 该HTMLParser级用来处理指令的SGML语法规则。使用尾随的XHTML处理指令'?'将导致该*数据*'?'包含在*数据*中。

HTMLParser.unknown_decl(*数据*)

当解析器读取无法识别的声明时调用此方法。

该*数据*参数将是内声明的全部内容<![...]>标记。被派生类覆盖有时很有用。基类实现什么都不做。

20.2.3。示例

下面的类实现了一个解析器，将用于说明更多示例：

```
from html.parser import HTMLParser
from html.entities import name2codepoint

class MyHTMLParser(HTMLParser):
    def handle_starttag(self, tag, attrs):
        print("Start tag:", tag)
        for attr in attrs:
            print("     attr:", attr)

    def handle_endtag(self, tag):
        print("End tag  :", tag)

    def handle_data(self, data):
        print("Data     :", data)

    def handle_comment(self, data):
        print("Comment  :", data)

    def handle_entityref(self, name):
        c = chr(name2codepoint[name])
        print("Named ent:", c)

    def handle_charref(self, name):
        if name.startswith('x'):
            c = chr(int(name[1:], 16))
        else:
            c = chr(int(name))
        print("Num ent  :", c)

    def handle_decl(self, data):
        print("Decl     :", data)

parser = MyHTMLParser()
```

解析文档类型：

```
>>> parser.feed('<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" '
...             '"http://www.w3.org/TR/html4/strict.dtd">')
Decl     : DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4
```

解析具有几个属性和标题的元素：

```
>>> parser.feed('<img src="python-logo.png" alt="The Python logo">')
Start tag: img
```

```
     attr: ('src', 'python-logo.png')
     attr: ('alt', 'The Python logo')
>>>
>>> parser.feed('<h1>Python</h1>')
Start tag: h1
Data     : Python
End tag  : h1
```

内容script和style元素按原样返回，无需进一步解析：

```
>>> parser.feed('<style type="text/css">#python { color: green }</style>')
Start tag: style
     attr: ('type', 'text/css')
Data     : #python { color: green }
End tag  : style

>>> parser.feed('<script type="text/javascript">'
...              'alert("<strong>hello!</strong>");</script>')
Start tag: script
     attr: ('type', 'text/javascript')
Data     : alert("<strong>hello!</strong>");
End tag  : script
```

解析评论：

```
>>> parser.feed('<!-- a comment -->'
...              '<!--[if IE 9]>IE-specific content<![endif]-->')
Comment  :  a comment
Comment  : [if IE 9]>IE-specific content<![endif]
```

解析命名和数字字符引用并将它们转换为正确的字符（注意：这3个引用全部相同'>'）：

```
>>> parser.feed('&gt;&#62;&#x3E;')
Named ent: >
Num ent  : >
Num ent  : >
```

将不完整的块送入feed()作品，但 handle_data()可能会多次调用（除非将*convert_charrefs*设置为True）：

```
>>> for chunk in ['<sp', 'an>buff', 'ered ', 'text</s', 'pan>']:
...     parser.feed(chunk)
...
Start tag: span
Data     : buff
Data     : ered
Data     : text
End tag  : span
```

解析无效的HTML（例如未加引号的属性）也适用：

```
>>> parser.feed('<p><a class=link href=#main>tag soup</p ></a>')
Start tag: p
Start tag: a
```

```
     attr: ('class', 'link')
     attr: ('href', '#main')
Data     : tag soup
End tag  : p
End tag  : a
```

20.3。 html.entities- HTML一般实体的定义

源代码： Lib / html / entities.py

这个模块定义了4个词典， html5 ， name2codepoint ， codepoint2name ，和entitydefs。

html.entities. **html5**

将名为字符引用[1]的 HTML5映射到等效Unicode字符的字典，例如。请注意，后缀分号包含在名称（例如）中，但是某些名称即使没有分号也会被标准接受：在这种情况下，名称存在且不存在。另见。 html5['gt;'] == '>''gt;'';' html.unescape()

3.3版本的新功能

html.entities. **entitydefs**

将XHTML 1.0实体定义映射到ISO Latin-1中的替换文本的字典。

html.entities. **name2codepoint**

一个将HTML实体名称映射到Unicode代码点的字典。

html.entities. **codepoint2name**

将Unicode代码点映射到HTML实体名称的字典。

脚注

[1] 请参阅https://www.w3.org/TR/html5/syntax.html#named-character-references

20.4。XML处理模块

源代码： Lib / xml /

用于处理XML的Python接口被分组在`xml`包中。

警告: XML模块对于错误或恶意构建的数据不安全。如果您需要解析不可信或未经身份验证的数据，请参阅XML漏洞以及 defusedxml和defusedexpat Packages部分。

需要注意的是，`xml`包中的模块要求至少有一个符合SAX的XML解析器可用。Expat解析器包含在Python中，因此`xml.parsers.expat`模块将始终可用。

`xml.dom`和`xml.sax`包的文档是DOM和SAX接口的Python绑定的定义。

XML处理子模块是：

- `xml.etree.ElementTree`：ElementTree API，一个简单且轻量级的XML处理器

- `xml.dom`：DOM API定义
- `xml.dom.minidom`：一个最小的DOM实现
- `xml.dom.pulldom`：支持构建部分DOM树

- `xml.sax`：SAX2基类和便利功能
- `xml.parsers.expat`：Expat分析器绑定

20.4.1。XML漏洞

XML处理模块对于恶意构建的数据不安全。攻击者可以滥用XML功能来执行拒绝服务攻击，访问本地文件，生成到其他机器的网络连接或绕过防火墙。

下表概述了已知的攻击以及各种模块是否易受攻击。

类	萨克斯	etree	命名dom	pulldom	XMLRPC
十亿笑	**弱势**	**弱势**	**弱势**	**弱势**	**弱势**
二次爆炸	**弱势**	**弱势**	**弱势**	**弱势**	**弱势**
外部实体扩张	**弱势**	安全（1）	安全（2）	**弱势**	安全
DTD检索	**弱势**	安全	安全	**弱势**	安全
减压弹	安全	安全	安全	安全	**弱势**

1. `xml.etree.ElementTree`不会扩展外部实体并`ParserError`在实体出现时引发 。
2. `xml.dom.minidom` 不会扩展外部实体，只是简单地返回未扩展的实体。
3. `xmlrpclib` 不会扩展外部实体并将其忽略。

十亿笑/指数实体扩张

该十亿笑攻击-也被称为指数实体膨胀-使用嵌套实体的多个级别。每个实体多次引用另一个实体，最终的实体定义包含一个小字符串。指数扩展会导致几千兆字节的文本并消耗大量内存和CPU时间。

二次爆炸实体扩展
: 二次爆炸攻击类似于十亿笑攻击; 它也滥用实体扩张。它不是嵌套的实体，而是一遍又一遍地重复一个带有几千个字符的大型实体。这种攻击并不像指数案例那样高效，但它避免了触发解析器禁止深层嵌套实体的对策。

外部实体扩张
: 实体声明可以包含多个文本以供替换。他们也可以指向外部资源或本地文件。XML解析器访问资源并将内容嵌入到XML文档中。

DTD检索
: 一些XML库像Python `xml.dom.pulldom`从远程或本地位置检索文档类型定义。该功能与外部实体扩展问题具有相似的含义。

减压弹
: 解压缩炸弹（又名ZIP炸弹）适用于所有可解析压缩XML流的XML库，如压缩的HTTP流或LZMA压缩文件。对于攻击者来说，它可以将传输的数据量减少三个或更多。

PyPI上的defusedxml文档提供了有关具有示例和引用的所有已知攻击媒介的更多信息。

20.4.2。在defusedxml和defusedexpat包

defusedxml是一个纯Python包，包含所有stdlib XML解析器的子类，用于防止任何潜在的恶意操作。对于解析不可信XML数据的任何服务器代码，建议使用此包。该软件包还附带了有关更多XML漏洞（如XPath注入）的示例漏洞利用和扩展文档。

defusedexpat提供了一个修改的libexpat和一个修补 `pyexpat`模块，用于应对实体扩展DoS攻击。该`defusedexpat`模块仍然允许一个理智的和可配置的实体扩展。这些修改可能包含在某些未来版本的Python中，但不会包含在Python的任何bug修复版本中，因为它们会破坏向后兼容性。

20.5。 xml.etree.ElementTree- ElementTree XML

源代码： Lib / xml / etree / ElementTree.py

该xml.etree.ElementTree模块实现了一个简单高效的API来解析和创建XML数据。

*在版本3.3中进行了更改：*只要可用，该模块将使用快速实施。该xml.etree.cElementTree模块已弃用。

警告: 该xml.etree.ElementTree模块对恶意构建的数据不安全。如果您需要解析不可信或未经身份验证的数据，请参阅XML漏洞。

20.5.1。教程

这是一个简短的教程xml.etree.ElementTree（ ET简而言之）。目标是展示模块的一些构建模块和基本概念。

20.5.1.1。XML树和元素

XML是一种固有的分层数据格式，用树来表示它是最自然的方式。 ET有两个类用于此目的 - ElementTree将整个XML文档表示为树，并 Element表示此树中的单个节点。与整个文档的交互（读写文件）通常在ElementTree关卡上完成。与单个XML元素及其子元素的交互是在Element关卡上完成的。

20.5.1.2。解析

我们将使用以下XML文档作为本节的示例数据：

```
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank>1</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank>4</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
    <country name="Panama">
        <rank>68</rank>
```

```
        <year>2011</year>
        <gdppc>13600</gdppc>
        <neighbor name="Costa Rica" direction="W"/>
        <neighbor name="Colombia" direction="E"/>
    </country>
</data>
```

我们可以通过从文件中读取来导入这些数据：

```
import xml.etree.ElementTree as ET
tree = ET.parse('country_data.xml')
root = tree.getroot()
```

或直接从一个字符串：

```
root = ET.fromstring(country_data_as_string)
```

fromstring()将XML从字符串中直接解析到一个Element，这是解析树的根元素。其他解析函数可能会创建一个ElementTree。检查文档以确保。

作为一个Element，root有一个标签和一个属性字典：

```
>>> root.tag
'data'
>>> root.attrib
{}
```

它还有我们可以迭代的子节点：

```
>>> for child in root:
...     print(child.tag, child.attrib)
...
country {'name': 'Liechtenstein'}
country {'name': 'Singapore'}
country {'name': 'Panama'}
```

儿童是嵌套的，我们可以通过索引访问特定的子节点：

```
>>> root[0][1].text
'2008'
```

注意： 并非XML输入的所有元素都将最终作为解析树的元素。目前，该模块跳过输入中的任何XML注释，处理指令和文档类型声明。尽管如此，使用此模块的API构建的树而不是从XML文本解析可以在其中有注释和处理指令；它们将在生成XML输出时包含在内。可以通过将自定义TreeBuilder实例传递给XMLParser 构造函数来访问文档类型声明。

20.5.1.3。Pull API进行非阻塞解析

此模块提供的大多数解析函数都要求在返回任何结果之前一次读取整个文档。可以逐渐使用XMLParser并向其馈送数据，但它是一个推送API，它在回调目标上调用方法，这对于大多数需

求而言太低级别且不方便。有时，用户真正想要的是能够在不阻碍操作的情况下逐步解析XML，同时享受完全构建的Element对象的便利性。

这是做这件事最有力的工具XMLPullParser。它不需要阻止读取来获取XML数据，而是通过XMLPullParser.feed()调用递增地提供数据。要获取解析的XML元素，请调用XMLPullParser.read_events()。这里是一个例子：

```
>>> parser = ET.XMLPullParser(['start', 'end'])
>>> parser.feed('<mytag>sometext')
>>> list(parser.read_events())
[('start', <Element 'mytag' at 0x7fa66db2be58>)]
>>> parser.feed(' more text</mytag>')
>>> for event, elem in parser.read_events():
...     print(event)
...     print(elem.tag, 'text=', elem.text)
...
end
```

明显的用例是以非阻塞方式运行的应用程序，其中从套接字接收XML数据或从某个存储设备逐步读取XML数据。在这种情况下，阻止读取是不可接受的。

因为它非常灵活，所以XMLPullParser使用简单的用例可能会很不方便。如果您不介意在读取XML数据时阻止应用程序，但仍希望具有增量分析功能，请查看iterparse()。当您阅读大型XML文档并且不想将其全部保存在内存中时，它会很有用。

20.5.1.4。寻找有趣的元素

Element有一些有用的方法可以递归地遍历它下面的所有子树（它的子节点，它们的子节点等等）。例如，Element.iter()：

```
>>> for neighbor in root.iter('neighbor'):
...     print(neighbor.attrib)
...
{'name': 'Austria', 'direction': 'E'}
{'name': 'Switzerland', 'direction': 'W'}
{'name': 'Malaysia', 'direction': 'N'}
{'name': 'Costa Rica', 'direction': 'W'}
{'name': 'Colombia', 'direction': 'E'}
```

Element.findall()仅找到具有当前元素的直接子元素的标签的元素。Element.find()找到具有特定标签的*第一个*孩子，并Element.text访问该元素的文本内容。Element.get()访问元素的属性：

```
>>> for country in root.findall('country'):
...     rank = country.find('rank').text
...     name = country.get('name')
...     print(name, rank)
...
Liechtenstein 1
Singapore 4
Panama 68
```

通过使用XPath，可以查找哪些元素的更复杂的规范。

20.5.1.5。修改XML文件

ElementTree提供了一种构建XML文档并将其写入文件的简单方法。该ElementTree.write()方法用于此目的。

一旦创建完成，Element可以通过直接更改其字段（如Element.text），添加和修改属性（Element.set()方法）以及添加新的子项（例如with Element.append()）来操纵对象。

假设我们想为每个国家的军衔添加一个军衔，并向updated 军衔元素添加一个属性：

```
>>> for rank in root.iter('rank'):
...     new_rank = int(rank.text) + 1
...     rank.text = str(new_rank)
...     rank.set('updated', 'yes')
...
>>> tree.write('output.xml')
```

我们的XML现在看起来像这样：

```
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank updated="yes">2</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank updated="yes">5</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
    <country name="Panama">
        <rank updated="yes">69</rank>
        <year>2011</year>
        <gdppc>13600</gdppc>
        <neighbor name="Costa Rica" direction="W"/>
        <neighbor name="Colombia" direction="E"/>
    </country>
</data>
```

我们可以使用删除元素Element.remove()。假设我们想要移除排名高于50的所有国家：

```
>>> for country in root.findall('country'):
...     rank = int(country.find('rank').text)
...     if rank > 50:
...         root.remove(country)
...
>>> tree.write('output.xml')
```

我们的XML现在看起来像这样：

```
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank updated="yes">2</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank updated="yes">5</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
</data>
```

20.5.1.6。构建XML文档

该SubElement()函数还提供了一种为给定元素创建新子元素的简便方法：

```
>>> a = ET.Element('a')
>>> b = ET.SubElement(a, 'b')
>>> c = ET.SubElement(a, 'c')
>>> d = ET.SubElement(c, 'd')
>>> ET.dump(a)
<a><b /><c><d /></c></a>
```

20.5.1.7。用命名空间解析

如果XML输入具有名称空间，标签与形式的前缀属性prefix:sometag得到扩展到 {uri}sometag 其中*的前缀*是由完全代替*URI*。另外，如果有一个默认名称空间，那么这个完整的URI会被预先添加到所有的非前缀标签中。

这是一个XML示例，它包含两个名称空间，一个名称前缀为“fictional”，另一个名称空间作为默认名称空间：

```
<?xml version="1.0"?>
<actors xmlns:fictional="http://characters.example.com"
        xmlns="http://people.example.com">
    <actor>
        <name>John Cleese</name>
        <fictional:character>Lancelot</fictional:character>
        <fictional:character>Archie Leach</fictional:character>
    </actor>
    <actor>
        <name>Eric Idle</name>
        <fictional:character>Sir Robin</fictional:character>
        <fictional:character>Gunther</fictional:character>
        <fictional:character>Commander Clement</fictional:character>
```

```
    </actor>
</actors>
```

搜索和浏览此XML示例的一种方法是将URI手动添加到a `find()` 或xpath中的每个标记或属性 `findall()` ：

```
root = fromstring(xml_text)
for actor in root.findall('{http://people.example.com}actor'):
    name = actor.find('{http://people.example.com}name')
    print(name.text)
    for char in actor.findall('{http://characters.example.com}character'):
        print(' |-->', char.text)
```

搜索命名空间XML示例的更好方法是使用自己的前缀创建一个字典并将其用于搜索函数中：

```
ns = {'real_person': 'http://people.example.com',
      'role': 'http://characters.example.com'}

for actor in root.findall('real_person:actor', ns):
    name = actor.find('real_person:name', ns)
    print(name.text)
    for char in actor.findall('role:character', ns):
        print(' |-->', char.text)
```

这两种方法都输出：

```
John Cleese
 |--> Lancelot
 |--> Archie Leach
Eric Idle
 |--> Sir Robin
 |--> Gunther
 |--> Commander Clement
```

20.5.1.8。其他资源

有关教程和其他文档的链接，请参阅http://effbot.org/zone/element-index.htm。

20.5.2。XPath支持

此模块为XPath表达式在树中定位元素提供了有限的支持 。目标是支持缩写语法的一小部分; 完整的XPath引擎不在该模块的范围内。

20.5.2.1。示例

下面是一个演示模块的一些XPath功能的示例。我们将使用Parsing XML部分中的 `countrydata`XML文档 ：

```
import xml.etree.ElementTree as ET
```

```
root = ET.fromstring(countrydata)

# Top-level elements
root.findall(".")

# All 'neighbor' grand-children of 'country' children of the top-level
# elements
root.findall("./country/neighbor")

# Nodes with name='Singapore' that have a 'year' child
root.findall(".//year/..[@name='Singapore']")

# 'year' nodes that are children of nodes with name='Singapore'
root.findall(".//*[@name='Singapore']/year")

# All 'neighbor' nodes that are the second child of their parent
root.findall(".//neighbor[2]")
```

20.5.2.2。支持的XPath语法

句法	含义
tag	选择具有给定标签的所有子元素。例如，spam选择名为的所有子元素spam，并spam/egg选择所有名为egg子级的 所有孙子spam。
*	选择所有子元素。例如，*/egg 选择所有名为的孙子egg。
.	选择当前节点。这在路径开始时最有用，表明它是相对路径。
//	选择当前元素下所有级别的所有子元素。例如，.//egg选择egg整个树中的所有元素。
..	选择父元素。返回None如果路径试图到达开始元素的祖先（元素find被称为上）。
[@attrib]	选择具有给定属性的所有元素。
[@attrib='value']	选择给定属性具有给定值的所有元素。该值不能包含引号。
[tag]	选择所有具有指定子项的元素 tag。只支持直接的孩子。
[tag='text']	选择所有具有名为tag其完整文本内容（包括后代）的孩子等于给定的元素 text。
[position]	选择位于给定位置的元素。位置可以是整数（1是第一个），表达式last()（对于最后一个），或者是last()-1（倒数第二个）等位置的位置。

谓词（方括号内的表达式）必须以标记名称，星号或其他谓词开头。 position谓词必须以标签名称开头。

20.5.3。参考

20.5.3.1。函数

xml.etree.ElementTree.Comment（*text = None*）

注释元素工厂。该工厂函数创建一个特殊元素，该元素将被标准串行器序列化为XML注释。注释字符串可以是字符串或Unicode字符串。 *文本*是包含注释字符串的字符串。返回表示评论的元素实例。

请注意，XMLParser跳过输入中的注释，而不是为它们创建注释对象。一个ElementTree只包含注释节点，如果他们已经被插入到使用的一个树Element的方法。

xml.etree.ElementTree.**dump**(*elem*)

将元素树或元素结构写入sys.stdout。该功能只能用于调试。

确切的输出格式是依赖于实现的。在这个版本中，它被写成普通的XML文件。

*elem*是元素树或个别元素。

xml.etree.ElementTree.**fromstring**(*文本*)

从字符串常量中解析XML部分。和...一样XML()。 *文本* 是一个包含XML数据的字符串。返回一个Element实例。

xml.etree.ElementTree.**fromstringlist**(*sequence*，*parser* = *None*)

从一系列字符串片段中解析XML文档。 *序列*是包含XML数据片段的列表或其他序列。 *解析器*是一个可选的解析器实例。如果没有给出， XMLParser 则使用标准解析器。返回一个Element实例。

3.2版本中的新功能

xml.etree.ElementTree.**iselement**(*元素*)

检查对象是否是有效的元素对象。 *元素*是一个元素实例。如果这是一个元素对象，则返回一个真值。

xml.etree.ElementTree.**iterparse**(*source*，*events* = *None*，*parser* = *None*)

逐步将XML部分解析为元素树，并向用户报告发生了什么。 *源*是 包含XML数据的文件名或文件对象。 *事件*是一系列要报告的事件。支持的事件为字符串"start"，"end"，"start-ns"和 "end-ns"（以下简称“NS”事件被用来获得详细的命名空间信息）。如果省略*事件*，则仅"end"报告事件。 *解析器*是一个可选的解析器实例。如果没有给出，XMLParser则使用标准 解析器。 *解析器*必须是其子类， XMLParser并且只能使用默认值TreeBuilder作为目标。返回提供的迭代器(event, elem) 对。

请注意，虽然iterparse()增量构建树，但会在*源*（或其名称的文件）上发出阻止读取。因此，它不适用于无法进行阻塞读取的应用程序。有关完全非阻塞解析，请参阅XMLPullParser。

> **注意：** iterparse()只保证它在发出“开始”事件时已经看到了开始标记的“>”字符，因此定义了属性，但是文本和尾部属性的内容在此处未定义。这同样适用于元素儿童; 他们可能会或可能不会在场。
> 如果您需要完全填充的元素，请查找“结束”事件。

*：自从3.4版本不推荐使用*的*解析器*的说法。

xml.etree.ElementTree.**parse**(*source*，*parser* = *None*)

将XML部分解析为元素树。 *源*是包含XML数据的文件名或文件对象。 *解析器*是一个可选的解析器实例。如果没有给出，XMLParser则使用标准解析器。返回一个 ElementTree实例。

xml.etree.ElementTree.ProcessingInstruction(*目标*, *文本=无*)

PI元件工厂。这个工厂函数创建一个特殊的元素，它将被序列化为XML处理指令。 *target*是包含PI目标的字符串。 *文本*是包含PI内容的字符串（如果给出）。返回一个表示处理指令的元素实例。

请注意，XMLParser跳过输入中的处理指令而不是为它们创建注释对象。一个 ElementTree只包含处理指令节点，如果他们已经被插入到使用的一个树 Element的方法。

xml.etree.ElementTree.register_namespace(*前缀*, *uri*)

注册一个名称空间前缀。注册表是全局的，任何现有的给定前缀或名称空间URI映射都将被删除。 *前缀*是一个名称空间前缀。 *uri*是一个命名空间uri。如果可能的话，该命名空间中的标签和属性将与给定的前缀一起序列化。

3.2版本中的新功能

xml.etree.ElementTree.SubElement(*parent*, *tag*, *attrib* = *{}*, ** *extra*)

子元素工厂。此函数创建一个元素实例，并将其附加到现有元素。

元素名称，属性名称和属性值可以是字节串或Unicode字符串。 *父母*是父母的元素。 *标记*是子元素名称。 *attrib*是一个可选字典，包含元素属性。 *extra*包含其他属性，作为关键字参数给出。返回一个元素实例。

xml.etree.ElementTree.tostring(*element*, *encoding =“us-ascii”*, *method =“xml”*, *, *short_empty_elements = True*)

生成一个XML元素的字符串表示，包括所有子元素。 *元素*是一个Element实例。 *encoding* [1]是输出编码（默认是US-ASCII）。使用encoding="unicode"生成一个Unicode字符串（否则生成一个字节字符串）。 *方法*是任一"xml"，"html"或"text"（默认为"xml"）。 *short_empty_elements*与in中的含义相同ElementTree.write()。返回包含XML数据的（可选）编码的字符串。

版本3.4中的新增功能：short_empty_elements参数。

xml.etree.ElementTree.tostringlist(*element*, *encoding =“us-ascii”*, *method =“xml”*, *, *short_empty_elements = True*)

生成一个XML元素的字符串表示，包括所有子元素。 *元素*是一个Element实例。 *encoding* [1]是输出编码（默认是US-ASCII）。使用encoding="unicode"生成一个Unicode字符串（否则生成一个字节字符串）。 *方法*是任一"xml"，"html"或"text"（默认为"xml"）。 *short_empty_elements*与in中的含义相同ElementTree.write()。返回包含XML数据的（可选）编码字符串的列表。除此之外，它不保证任何特定的顺序。 b"".join(tostringlist(element)) == tostring(element)

3.2版本中的新功能

版本3.4中的新增功能：short_empty_elements参数。

xml.etree.ElementTree.XML(*文本*, *解析器=无*)

从字符串常量中解析XML部分。这个函数可以用来在Python代码中嵌入“XML文字”。 *文本*是一个包含XML数据的字符串。 *解析器*是一个可选的解析器实例。如果没有给出，

XMLParser则使用标准 解析器。返回一个Element实例。

xml.etree.ElementTree.XMLID（*文本*，*解析器=无*）

从字符串常量中解析XML部分，并返回一个从元素ID：s映射到元素的字典。 *文本*是一个包含XML数据的字符串。 *解析器*是一个可选的解析器实例。如果没有给出，XMLParser则使用标准 解析器。返回包含Element实例和字典的元组 。

20.5.3.2。Element对象

class xml.etree.ElementTree.Element（*tag*，*attrib = {}*，*** extra* ）

元素类。该类定义了Element接口，并提供了该接口的参考实现。

元素名称，属性名称和属性值可以是字节串或Unicode字符串。 *标签*是元素名称。 *attrib*是一个可选字典，包含元素属性。 *extra*包含其他属性，作为关键字参数给出。

tag

一个字符串，用于标识此元素表示的数据类型（换句话说，元素类型）。

text
tail

这些属性可以用来保存与元素相关的附加数据。它们的值通常是字符串，但可能是任何特定于应用程序的对象。如果元素是从XML文件创建的，则*text*属性包含元素的开始标记与其第一个子标记或结束标记之间的文本，或者None，*tail*属性包含元素的结束标记和下一个标记之间的文本，或者None。对于XML数据

```
<a><b>1<c>2<d/>3</c></b>4</a>
```

的*一个*元件具有None两个*文本*和*尾*属性，所述*b*元件具有*文本* *"1"*和*尾部* *"4"*，所述*Ç*元件具有*文本* *"2"*和*尾* None，并且*d*元件具有*文本* None和*尾巴* *"3"*。

收集的元件的内文，见itertext()，例如*""*.join(element.itertext())。

应用程序可以在这些属性中存储任意对象

attrib

包含元素属性的字典。请注意，虽然 *attrib*值始终是一个真正可变的Python字典，但ElementTree实现可能会选择使用另一个内部表示，并且只有在有人要求时才创建字典。要利用这些实现，请尽可能使用下面的字典方法。

以下类似字典的方法适用于元素属性。

clear（）

重置一个元素。此函数删除所有子元素，清除所有属性，并将文本和尾部属性设置为None。

get（*键*，*默认=无*）

获取名为*key*的元素属性。

返回属性值，如果未找到该属性，则返回*默认*值。

items ( )
以（名称，值）对的序列形式返回元素属性。属性以任意顺序返回。

keys ( )
以列表形式返回元素属性名称。名称以任意顺序返回。

set (*key*, *value*)
将元素上的属性*键*设置为*值*。

以下方法适用于元素的子元素（子元素）。

append (*子元素*)
将元素*子元素*添加到此元素内部子元素列表的末尾。TypeError如果*子元素*不是，则 引发Element。

extend (*子元素*)
从零个或多个元素的序列对象追加*子*元素。TypeError如果一个子元素不是一个子元素，则引发Element。

3.2版本中的新功能

find (*match*, *namespaces = None*)
查找第一个子元素匹配*匹配*。 *匹配*可以是标签名称或路径。返回一个元素实例或None。 *名称空间*是从名称空间前缀到全名的可选映射。

findall (*match*, *namespaces = None*)
按标记名称或路径查找所有匹配的子元素 。返回包含文档顺序中所有匹配元素的列表。 *名称空间*是从名称空间前缀到全名的可选映射。

findtext (*match*, *default = None*, *namespaces = None*)
查找第一个子元素匹配*匹配的*文本。 *匹配*可以是标签名称或路径。返回第一个匹配元素的文本内容，如果没有找到元素，则返回*默认值*。请注意，如果匹配元素没有文本内容，则返回空字符串。*名称空间*是从名称空间前缀到全名的可选映射。

getchildren ( )
*自3.2版弃用：*使用list(elem)或迭代。

getiterator (*tag = None*)
*自3.2版弃用：*请改用方法Element.iter()。

insert (*索引*, *子元素*)
在此元素的给定位置插入*子*元素。TypeError如果*子元素*不是，则引发 Element。

iter (*tag = None*)
创建一个以当前元素为根的树型迭代器。迭代器以文档（深度优先）顺序遍历此元素及其下的所有元素。如果*标签*不是None或者'*'，则只有标签等于*标签的*元素从迭代器返回。如果在迭代期间修改了树结构，结果是未定义的。

3.2版本中的新功能

iterfind (*match* , *namespaces* = *None*)

按标记名称或路径查找所有匹配的子元素 。返回按文档顺序产生所有匹配元素的迭代。*名称空间*是从名称空间前缀到全名的可选映射。

3.2版本中的新功能

itertext ( )

创建一个文本迭代器。迭代器按照文档顺序遍历此元素和所有子元素，并返回所有内部文本。

3.2版本中的新功能

makeelement (*tag* , *attrib*)

创建与此元素相同类型的新元素对象。不要调用这个方法，SubElement()而是使用工厂函数。

remove (*子元素*)

从元素中删除*子*元素。与find *方法不同，此方法根据实例标识比较元素，而不是标签值或内容。

Element 对象还支持下列序列类型的方法来与子元素的工作：__delitem__() ，__getitem__() ，__setitem__() ， __len__()。

警告：没有子元素的元素将测试为False。这种行为在未来的版本中会改变。改用特定len(elem)或测试。elem is None

```
element = root.find('foo')

if not element:  # careful!
    print("element not found, or element has no subelements")

if element is None:
    print("element not found")
```

20.5.3.3。ElementTree的对象

class xml.etree.ElementTree. ElementTree (*element* = *None* , *file* = *None*)

ElementTree包装类。该类表示整个元素层次结构，并为标准XML序列化和额外支持。

*元素*是根元素。如果给定，树会用XML *文件*的内容进行初始化。

_setroot (*元素*)

替换此树的根元素。这会丢弃树的当前内容，并用给定的元素替换它。小心使用。 *元素*是一个元素实例。

find (*match* , *namespaces* = *None*)

相同Element.find() ，从树的根开始。

findall (*match* , *namespaces* = *None*)

相同Element.findall() ，从树的根开始。

findtext (*match* , *default* = *None* , *namespaces* = *None*)

相同Element.findtext() ，从树的根开始。

getiterator (*tag* = *None*)

*自3.2版弃用：*请改用方法ElementTree.iter()。

getroot ( )

返回此树的根元素。

iter (*tag* = *None*)

创建并返回根元素的树型迭代器。迭代器按照节的顺序遍历树中的所有元素。 *标签*是要查找的标签（默认是返回所有元素）。

iterfind (*match* , *namespaces* = *None*)

相同Element.iterfind() ，从树的根开始。

3.2版本中的新功能

parse (*source* , *parser* = *None*)

将外部XML部分加载到此元素树中。 *源*是一个文件名或文件对象。 *解析器*是一个可选的解析器实例。如果没有给出， XMLParser则使用标准解析器。返回节根元素。

write (*file* , *encoding* ="us-ascii" , *xml_declaration* = *None* , *default_namespace* = *None* , *method* ="xml" , * , *short_empty_elements* = *True*)

将元素树作为XML写入文件。 *文件*是一个文件名，或者是一个为写入而打开的 文件对象。 *encoding* [1]是输出编码（默认是US-ASCII）。 *xml_declaration*控制是否应该将XML声明添加到文件中。使用False了永远， True为始终， None 仅当不US-ASCII或UTF-8或Unicode（默认为None）。 *default_namespace*设置默认的XML名称空间（对于“xmlns”）。 *方法*是任一"xml" ， "html"或"text" （默认为 "xml" ）。仅关键字*short_empty_elements*参数控制不包含内容的元素的格式。如果True （默认），它们作为单个自封闭标签发射，否则它们作为一对开始/结束标签发射。

输出是一个字符串（ str ）或二进制（ bytes ）。这由*编码*参数控制。如果*编码*是"unicode" ，则输出是一个字符串; 否则，它是二元的。请注意，如果它是一个打开的文件对象，这可能与*文件*类型冲突; 确保你不要尝试写入一个字符串到二进制流，反之亦然。

*版本3.4中的新增功能：*short_empty_elements参数。

这是将要被操纵的XML文件：

```
<html>
    <head>
        <title>Example page</title>
    </head>
    <body>
        <p>Moved to <a href="http://example.org/">example.org</a>
```

```
        or <a href="http://example.com/">example.com</a>.</p>
    </body>
</html>
```

更改第一段中每个链接的属性“目标”的示例：

```
>>> from xml.etree.ElementTree import ElementTree
>>> tree = ElementTree()
>>> tree.parse("index.xhtml")
<Element 'html' at 0xb77e6fac>
>>> p = tree.find("body/p")      # Finds first occurrence of tag p in body
>>> p
<Element 'p' at 0xb77ec26c>
>>> links = list(p.iter("a"))    # Returns list of all links
>>> links
[<Element 'a' at 0xb77ec2ac>, <Element 'a' at 0xb77ec1cc>]
>>> for i in links:              # Iterates through all found links
...     i.attrib["target"] = "blank"
>>> tree.write("output.xhtml")
```

20.5.3.4。QName对象

class xml.etree.ElementTree.QName（*text_or_uri*，*tag* = *None* ）

QName包装。这可以用来包装QName属性值，以便在输出上获得正确的名称空间处理。*text_or_uri*是一个包含QName值的字符串，格式为{uri} local，或者，如果给出tag参数，则为QName的URI部分。如果给出*标签*，则第一个参数被解释为URI，并且该参数被解释为本地名称。 QName实例是不透明的。

20.5.3.5。TreeBuilder对象

class xml.etree.ElementTree.TreeBuilder（*element_factory* = *None* ）

通用元素结构构建器。该构建器将一系列开始，数据和结束方法调用转换为格式良好的元素结构。您可以使用此类使用自定义XML解析器构建元素结构，或使用其他类似XML的格式的解析器。 当给定时，*element_factory*必须是一个可调用的接受两个位置参数：一个标签和一个属性字典。预计会返回一个新的元素实例。

close（ ）

刷新构建器缓冲区，并返回顶层文档元素。返回一个Element实例。

data（*数据*）

将文本添加到当前元素。 *数据*是一个字符串。这应该是一个字节串或一个Unicode字符串。

end（*标签*）

关闭当前元素。 *标签*是元素名称。返回关闭的元素。

start（*tag*，*attrs* ）

打开一个新元素。 *标签*是元素名称。 *attrs*是一个包含元素属性的字典。返回打开的元素。

另外，一个自定义TreeBuilder对象可以提供以下方法：

doctype (*name* , *pubid* , *system*)

处理文档类型声明。 *名称*是文档的名称。 *pubid*是公共标识符。 *系统*是系统标识符。此方法在默认TreeBuilder类中不存在。

3.2版本中的新功能

20.5.3.6。XMLParser对象

class xml.etree.ElementTree. XMLParser (*html = 0* , *target = None* , *encoding = None*)

这个类是模块的低级构建块。它 xml.parsers.expat用于高效的基于事件的XML解析。它可以用该feed()方法递增地提供XML数据，并且解析事件被转换为推送API - 通过调用*目标*对象上的回调。如果省略*目标*，TreeBuilder则使用标准。该*HTML*论点是历史上用于向后兼容性和现在已经过时。如果给出*编码* [1]，则该值将覆盖在XML文件中指定的编码。

*：自从3.4版本不推荐使用*的*HTML*参数。其余参数应通过关键字传递，以准备移除*html*参数。

close ( )

完成将数据提供给解析器。返回调用施工期间传递close()的*目标*方法的结果；默认情况下，这是顶级文档元素。

doctype (*name* , *pubid* , *system*)

*自3.2版弃用：*TreeBuilder.doctype()在自定义TreeBuilder目标上定义方法。

feed (*数据*)

将数据提供给解析器。 *数据*是编码数据。

XMLParser.feed()调用*目标*的方法，用于每个开口标签，其对于每个闭合标签方法，和数据由方法处理。 调用 *目标*的方法。不仅可以用于构建树型结构。这是计算XML文件最大深度 的 示 例 ： start(tag, attrs_dict) end(tag) data(data) XMLParser.close() close() XMLParser

```
>>> from xml.etree.ElementTree import XMLParser
>>> class MaxDepth:                     # The target object of the parser
...     maxDepth = 0
...     depth = 0
...     def start(self, tag, attrib):   # Called for each opening tag.
...         self.depth += 1
...         if self.depth > self.maxDepth:
...             self.maxDepth = self.depth
...     def end(self, tag):             # Called for each closing tag.
...         self.depth -= 1
...     def data(self, data):
...         pass            # We do not need to do anything with data.
...     def close(self):    # Called when all data has been parsed.
...         return self.maxDepth
...
>>> target = MaxDepth()
```

```
>>> parser = XMLParser(target=target)
>>> exampleXml = """
... <a>
...   <b>
...   </b>
...   <b>
...     <c>
...       <d>
...       </d>
...     </c>
...   </b>
... </a>"""
>>> parser.feed(exampleXml)
>>> parser.close()
4
```

20.5.3.7。XMLPullParser对象

class xml.etree.ElementTree.XMLPullParser (*events = None*)

拉解析器适用于非阻塞应用程序。它的输入端API 与其类似XMLParser，但不是将调用推送到回调目标，而是XMLPullParser收集解析事件的内部列表并让用户读取它。*事件*是一系列要报告的事件。支持的事件为字符串"start"，"end"，"start-ns"和"end-ns"（以下简称“NS”事件被用来获得详细的命名空间信息）。如果省略*事件*，则仅"end"报告事件。

feed (*数据*)

将给定的字节数据送入解析器。

close ( )

向解析器发出数据流已终止的信号。不像 XMLParser.close()，这个方法总是返回None。解析器关闭时尚未检索到的任何事件仍然可以使用read_events()。

read_events ( )

将迭代器返回到馈送给解析器的数据中遇到的事件。迭代器产生对，其中*event*是表示事件类型的字符串（例如），*elem*是遇到的对象。(event, elem)"end"Element

之前通话中提供的事件read_events()将不会再次提交。仅当从迭代器中检索事件时才会从内部队列中消耗事件，因此多个读取器并行迭代迭代器read_events()会产生不可预知的结果。

注意： XMLPullParser只保证它在发出“开始”事件时已经看到了开始标记的“>”字符，因此定义了属性，但是文本和尾部属性的内容在此处未定义。这同样适用于元素儿童; 他们可能会或可能不会在场。
如果您需要完全填充的元素，请查找“结束”事件。

3.4版新增功能

20.5.3.8。例外

*类*xml.etree.ElementTree.ParseError

解析失败时，此模块中的各种解析方法引发XML解析错误。此异常的实例的字符串表示形式将包含用户友好的错误消息。另外，它将具有以下可用属性：

`code`

来自expat分析器的数字错误代码。请参阅`xml.parsers.expat`错误代码及其含义列表的文档 。

`position`

*行的列*元组，*列*号，指定发生错误的位置。

脚注

[1] *(1，2，3，4)* 包括在XML输出的编码串应符合适当的标准。例如，“UTF-8”是有效的，但“UTF8”不是。见https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl 和https://www.iana.org/assignments/character-sets/character-sets.xhtml。

20.6。 xml.dom- 文档对象模型

源代码： Lib / xml / dom / __ init__.py

文档对象模型或“DOM”是万维网联盟（W3C）的跨语言API，用于访问和修改XML文档。DOM实现将XML文档呈现为树结构，或者允许客户端代码从头构建这样的结构。然后通过提供众所周知的接口的一组对象来访问该结构。

DOM对随机访问应用程序非常有用。SAX只允许您一次查看文档的一个位。如果您正在查看一个SAX元素，则无法访问其他元素。如果您正在查看文本节点，则无法访问包含元素。当你编写一个SAX应用程序时，你需要跟踪自己代码中某个地方的程序在文档中的位置。SAX不会为你做。另外，如果你需要在XML文档中向前看，你只是运气不好。

某些应用程序在事件驱动模型中无法访问树是根本不可能的。当然，你可以在SAX事件中自己构建某种树，但是DOM允许你避免编写代码。DOM是XML数据的标准树表示。

文档对象模型由W3C分阶段定义，或者在其术语中定义为“级别”。API的Python映射基本上基于DOM Level 2推荐。

通常，DOM应用程序首先将一些XML解析为DOM。如何实现这一点完全没有被DOM Level 1覆盖，而Level 2只提供有限的改进：有一个DOMImplementation对象类提供对Document创建方法的访问，但没有办法在实现中访问XML读取器/解析器/文档构建器独立的方式。在没有现有Document对象的情况下，也没有明确的方式来访问这些方法 。在Python中，每个DOM实现都会提供一个函数getDOMImplementation()。DOM Level 3添加了一个Load / Store规范，该规范为读者定义了一个接口，但是这在Python标准库中尚不可用。

一旦有了DOM文档对象，就可以通过其属性和方法访问XML文档的各个部分。这些属性在DOM规范中定义; 这部分参考手册描述了Python中规范的解释。

W3C提供的规范定义了用于Java，ECMAScript和OMG IDL的DOM API。这里定义的Python映射很大程度上基于IDL版本的规范，但不需要严格的遵从（尽管实现可以自由支持来自IDL的严格映射）。有关映射要求的详细讨论，请参阅一致性部分 。

也可以看看:

文档对象模型（DOM）Level 2规范
Python DOM API基于的W3C建议。

文档对象模型（DOM）1级规范
W3C支持的W3C推荐xml.dom.minidom。

Python语言映射规范
这指定了从OMG IDL到Python的映射。

20.6.1。模块内容

将xml.dom包含以下功能：

xml.dom.registerDOMImplementation(*名称*, *工厂*)
注册*工厂*名称为功能*名称*。工厂函数应该返回一个实现DOMImplementation 接口的对象。工厂函数可以每次都返回相同的对象，或者针对每次调用返回一个新对象，以适合特定的实现（例如，如果该实现支持一些定制）。

xml.dom.getDOMImplementation(*name = None*, *features = ()*)
返回一个合适的DOM实现。该*名称可能*是众所周知的，也可能是DOM实现的模块名称None。如果不是None，则导入相应的模块并DOMImplementation在导入成功时返回一个对象。如果没有给出名称，并且环境变量 PYTHON_DOM 被设置，这个变量用来查找实现。

如果没有给出名称，这将检查可用实现以找到具有所需功能集的实现。如果没有找到实现，请举一个 ImportError。要素列表必须是在可用对象上传递给方法 的一系列对。(feature, version) hasFeature() DOMImplementation

还提供了一些便利常数：

xml.dom.EMPTY_NAMESPACE
用于指示没有名称空间与DOM中的节点关联的值。这通常作为namespaceURI节点的*名称*，或者作为*名称*空间特定方法的*namespaceURI*参数使用。

xml.dom.XML_NAMESPACE
与保留前缀关联的名称空间URI xml，由XML中的Namespaces定义（第4节）。

xml.dom.XMLNS_NAMESPACE
由文档对象模型（DOM）Level 2 Core Specification（第1.1.8节）定义的名称空间声明的名称空间URI。

xml.dom.XHTML_NAMESPACE
由XHTML 1.0定义的XHTML名称空间的URI：可扩展超文本标记语言（3.1.1节）。

另外，还xml.dom包含一个基Node类和DOM异常类。该Node模块提供的类没有实现DOM规范定义的任何方法或属性; 具体的DOM实现必须提供这些。Node作为该模块的一部分提供的类确实提供了用于nodeType具体Node对象上的属性的常量；它们位于类内而不是模块级，以符合DOM规范。

20.6.2。在DOM对象

DOM的权威性文档是W3C的DOM规范。

请注意，DOM属性也可以作为节点来操作，而不是简单的字符串。然而，你必须这样做是非常罕见的，所以这种用法还没有记录。

接口	部分	目的
DOMImplementation	DOMImplementation对象	接口的底层实现。
Node	节点对象	文档中大多数对象的基本接口
NodeList	NodeList对象	一系列节点的接口。
DocumentType	DocumentType对象	关于处理文档所需声明的信息

接口	部分	目的
Document	文档对象	表示整个文档的对象。
Element	元素对象	文档层次结构中的元素节点。
Attr	属性对象	元素节点上的属性值节点。
Comment	评论对象	在源文档中表示评论。
Text	文本和CDATASection对象	包含文档中文本内容的节点。
ProcessingInstruction	ProcessingInstruction对象	处理指令表示。

附加部分描述了在Python中使用DOM定义的异常。

20.6.2.1。DOMImplementation对象

该DOMImplementation接口为应用程序提供了一种确定其正在使用的DOM中特定功能的可用性的方法。DOM Level 2增加了使用该功能创建新对象Document和 DocumentType对象的功能DOMImplementation。

DOMImplementation.hasFeature（*功能*，*版本*）

如果实现了一对字符串*功能*和 *版本*标识的功能，则返回true 。

DOMImplementation.createDocument（*namespaceUri*，*qualifiedName*，*doctype* ）

返回一个新的Document对象（ DOM的根），一个Element具有给定*namespaceUri*和*qualifiedName*的子 对象。的 *文档类型*必须是一个DocumentType由创建的对象createDocumentType()，或None。在Python DOM API中，前两个参数也可以None是为了表示不Element 创建子对象。

DOMImplementation.createDocumentType（*qualifiedName*，*publicId*，*systemId* ）

返回一个DocumentType封装给定的*qualifiedName*，*publicId*和*systemId*字符串的新对象 ，表示包含在XML文档类型声明中的信息。

20.6.2.2。节点对象

XML文档的所有组件都是其子类Node。

Node.nodeType

表示节点类型的整数。对于类型的符号常量是在Node对象：ELEMENT_NODE，ATTRIBUTE_NODE ， TEXT_NODE ， CDATA_SECTION_NODE ， ENTITY_NODE ，PROCESSING_INSTRUCTION_NODE，COMMENT_NODE， DOCUMENT_NODE，DOCUMENT_TYPE_NODE，NOTATION_NODE。这是一个只读属性。

Node.parentNode

当前节点的父级或None文档节点。该值始终是一个Node对象或None。对于Element节点，这将是父元素，除了根元素，在这种情况下，它将是Document对象。对于Attr节点，这总是None。这是一个只读属性。

Node.attributes

一个NamedNodeMap属性对象。只有元素才具有实际的价值; 其他人提供None这个属性。这是一个只读属性。

Node.previousSibling
紧跟在同一父节点之前的节点。例如，在*自身*元素的开始标记之前有一个结束标记的元素。当然，XML文档不仅仅包含元素，所以以前的兄弟可以是文本，评论或其他东西。如果此节点是父项的第一个子项，则此属性将为 None。这是一个只读属性。

Node.nextSibling
紧跟在这个父节点之后的节点。另见 previousSibling。如果这是父项的最后一个子项，则此属性将为None。这是一个只读属性。

Node.childNodes
此节点中包含的节点列表。这是一个只读属性。

Node.firstChild
节点的第一个孩子，如果有的话，或者None。这是一个只读属性。

Node.lastChild
节点的最后一个孩子，如果有的话，或者None。这是一个只读属性。

Node.localName
下面的部分是tagName冒号，如果有的话，否则整个tagName。该值是一个字符串。

Node.prefix
tagName如果有冒号，冒号前面的部分，否则为空字符串。该值是一个字符串，或None。

Node.namespaceURI
与元素名称关联的名称空间。这将是一个字符串或 None。这是一个只读属性。

Node.nodeName
这对每种节点类型都有不同的含义; 有关详细信息，请参阅DOM规范。您始终可以从另一个属性获取您在此获得的信息，例如tagName元素name 属性或属性属性。对于所有节点类型，此属性的值将是一个字符串或None。这是一个只读属性。

Node.nodeValue
这对每种节点类型都有不同的含义; 有关详细信息，请参阅DOM规范。情况类似于nodeName。该值是一个字符串或None。

Node.hasAttributes( )
如果节点具有任何属性，则返回true。

Node.hasChildNodes( )
如果节点有任何子节点，则返回true。

Node.isSameNode(*其他*)
如果*other*引用与此节点相同的节点，则返回true 。这对于使用任何类型的代理体系结构的DOM实现特别有用（因为多个对象可以引用同一个节点）。

> **注意：** 这是基于建议的DOM Level 3 API，它仍处于“工作草案”阶段，但这个特定的界面似乎没有争议。来自W3C的改变不一定会影响Python DOM接口中的这种方法（尽管任何新的W3C API也会被支持）。

Node.appendChild(*newChild*)
将新的子节点添加到子节点列表末尾的此节点，并返回*newChild*。如果节点已经在树中，它将首先被移除。

Node.insertBefore(*newChild*, *refChild*)
在现有的孩子之前插入一个新的孩子节点。它必须是 *refChild*是这个节点的孩子的情况; 如果没有，ValueError则会提出。 *newChild*返回。如果*refChild*是None，它会在儿童列表的末尾插入*newChild*。

Node.removeChild(*oldChild*)
删除一个子节点。 *oldChild*必须是该节点的子节点; 如果没有， ValueError则会提出。 *oldChild*成功返回。如果*oldChild* 不会被进一步使用，它的unlink()方法应该被调用。

Node.replaceChild(*newChild*, *oldChild*)
用新节点替换现有节点。*oldChild*必须是 这个节点的子节点; 如果没有，ValueError则会提出。

Node.normalize( )
加入相邻的文本节点，以便所有文本段都作为单个Text实例存储 。这为许多应用程序简化了从DOM树处理文本。

Node.cloneNode(*深*)
克隆这个节点。设置*深层*意味着克隆所有子节点。这将返回克隆。

20.6.2.3。NodeList对象

A NodeList表示一系列节点。这些对象以两种方式使用DOM中的核心建议：一个Element对象提供一个作为其子节点的名单，以及getElementsByTagName()和 getElementsByTagNameNS()方法Node返回的对象与此接口来表示查询结果。

DOM Level 2建议为这些对象定义了一个方法和一个属性：

NodeList.item(*i*)
如果还有一个，则返回序列中的第*i*项None。指数*i*不得小于零或大于或等于序列的长度。

NodeList.length
序列中的节点数量。

另外，Python DOM接口要求提供一些额外的支持来允许将NodeList对象用作Python序列。所有 NodeList实现必须包含对__len__()和的 支持 __getitem__(); 这允许迭代NodeListin for语句并适当支持len()内置函数。

如果一个DOM实现支持文档的修改，那么 NodeList 实现也必须支持 __setitem__() 和 __delitem__()方法。

20.6.2.4。DocumentType对象

关于由文档声明的符号和实体的信息（包括解析器使用它并可以提供信息的外部子集）可以从DocumentType对象中获得。在DocumentType为一个文件可以从Document对象的doctype 属性; 如果DOCTYPE文档没有声明，则将文档的doctype属性设置为None而不是此接口的实例。

DocumentType是专业化的Node，并增加了以下属性：

DocumentType.publicId
: 文档类型定义的外部子集的公共标识符。这将是一个字符串或None。

DocumentType.systemId
: 文档类型定义的外部子集的系统标识符。这将作为一个字符串的URI，或None。

DocumentType.internalSubset
: 从文档中提供完整内部子集的字符串。这不包括包含子集的括号。如果文档没有内部子集，应该是None。

DocumentType.name
: DOCTYPE声明中给出的根元素的名称（如果存在）。

DocumentType.entities
: 这是NamedNodeMap给出外部实体的定义。对于不止一次定义的实体名称，只提供第一个定义（其他则根据XML建议的要求被忽略）。这可能是 None由于解析器没有提供信息，或者没有定义实体。

DocumentType.notations
: 这是NamedNodeMap给出符号的定义。对于不止一次定义的记法名称，只提供第一个定义（其他则根据XML建议的要求而被忽略）。这可能是 None由于解析器没有提供信息，或者没有定义符号。

20.6.2.5。文档对象

A Document表示整个XML文档，包括其组成元素，属性，处理指令，注释等。请记住它从中继承属性Node。

Document.documentElement
: 文档的唯一根元素。

Document.createElement(*tagName*)
: 创建并返回一个新的元素节点。元素在创建时未插入到文档中。您需要使用其他方法之一（如insertBefore()或）来显式插入它appendChild()。

Document.createElementNS(*namespaceURI*，*tagName*)

用命名空间创建并返回一个新元素。该*标签名*可以有一个前缀。元素在创建时未插入到文档中。您需要使用其他方法之一（如insertBefore()或）来显式插入它 appendChild()。

Document.createTextNode（*数据*）

创建并返回包含作为参数传递的数据的文本节点。与其他创建方法一样，这个方法不会将节点插入树中。

Document.createComment（*数据*）

创建并返回包含作为参数传递的数据的注释节点。与其他创建方法一样，这个方法不会将节点插入树中。

Document.createProcessingInstruction（*目标*，*数据*）

创建并返回包含作为参数传递的*目标*和 *数据*的处理指令节点。与其他创建方法一样，这个方法不会将节点插入树中。

Document.createAttribute（*名字*）

创建并返回一个属性节点。此方法不会将属性节点与任何特定元素相关联。您必须setAttributeNode()在适当的Element对象上使用新创建的属性实例。

Document.createAttributeNS（*namespaceURI*，*qualifiedName* ）

用命名空间创建并返回一个属性节点。该*标签名*可以有一个前缀。此方法不会将属性节点与任何特定元素相关联。您必须setAttributeNode()在适当的 Element对象上使用新创建的属性实例。

Document.getElementsByTagName（*tagName* ）

搜索所有具有特定元素类型名称的后代（直接子代，子代孩子等）。

Document.getElementsByTagNameNS（*namespaceURI*，*localName* ）

使用特定的命名空间URI和本地名搜索所有后代（直接的孩子，儿童的孩子等）。localname是前缀后的名称空间的一部分。

20.6.2.6。Element对象

Element是它的一个子类Node，所以继承了该类的所有属性。

Element.tagName

元素类型名称。在使用命名空间的文档中，它可能有冒号。该值是一个字符串。

Element.getElementsByTagName（*tagName* ）

与Document课堂中的等同方法相同。

Element.getElementsByTagNameNS（*namespaceURI*，*localName* ）

与Document课堂中的等同方法相同。

Element.hasAttribute（*名字*）

如果元素具有按*名称*命名的属性，则返回true 。

Element.hasAttributeNS（*namespaceURI*，*localName* ）
如果元素具有由*namespaceURI*和 *localName*命名的属性，则返回true 。

Element.getAttribute（*名字*）
以字符串形式返回按*名称*命名的属性的值。如果不存在这样的属性，则返回空字符串，就好像该属性没有值。

Element.getAttributeNode（*attrname* ）
返回Attr由*attrname*命名的属性的节点。

Element.getAttributeNS（*namespaceURI*，*localName* ）
以字符串形式返回由*namespaceURI*和*localName*命名的属性的值。如果不存在这样的属性，则返回空字符串，就好像该属性没有值。

Element.getAttributeNodeNS（*namespaceURI*，*localName* ）
在给定*namespaceURI*和*localName的情况下*，将属性值作为节点返回。

Element.removeAttribute（*名字*）
按名称删除属性。如果没有匹配的属性， NotFoundErr则引发a。

Element.removeAttributeNode（*oldAttr* ）
如果存在，请从属性列表中删除并返回*oldAttr*。如果*oldAttr*不存在，NotFoundErr则提出。

Element.removeAttributeNS（*namespaceURI*，*localName* ）
按名称删除属性。请注意，它使用localName而不是qname。如果没有匹配属性，则不会引发异常。

Element.setAttribute（*名称*，*值*）
从字符串中设置一个属性值。

Element.setAttributeNode（*newAttr* ）
将新的属性节点添加到元素，如果name属性匹配，则在必要时替换现有的属性。如果发生替换，则将返回旧的属性节点。如果*newAttr*已被使用， InuseAttributeErr将会被提出。

Element.setAttributeNodeNS（*newAttr* ）
添加一个新的属性节点到元素中，如果namespaceURI和localName属性匹配，则在必要时替换现有的属性。如果发生替换，则将返回旧的属性节点。如果*newAttr* 已被使用，InuseAttributeErr将会被提出。

Element.setAttributeNS（*namespaceURI*，*qname*，*value* ）
给定一个*namespaceURI*和一个*qname* ，从一个字符串中设置一个属性值。请注意，qname是整个属性名称。这与以上不同。

20.6.2.7。Attr

Attr从继承Node ，所以继承了它的所有属性。

Attr.name
属性名称。在使用命名空间的文档中，它可能包含冒号。

Attr.localName
如果有冒号，冒号后面的部分名称，否则为整个名称。这是一个只读属性。

Attr.prefix
冒号前面的部分名称，如果有的话，否则为空字符串。

Attr.value
属性的文本值。这是该nodeValue属性的同义词 。

20.6.2.8。的NamedNodeMap对象

NamedNodeMap并*没有*继承Node。

NamedNodeMap.length
属性列表的长度。

NamedNodeMap.item(*索引*)
返回具有特定索引的属性。您获取属性的顺序是任意的，但对于DOM的生命周期将保持一致。每个项目是一个属性节点。通过value属性获取它的值。

也有实验方法给这个类更多的映射行为。您可以使用它们，也可以getAttribute*()在Element对象上使用标准化的方法族。

20.6.2.9。评论对象

Comment代表XML文档中的评论。它是子类 Node ，但不能有子节点。

Comment.data
评论的内容为一个字符串。该属性包含前导<!--和尾随之间的所有字符-->，但不包括它们。

20.6.2.10。文本和CDATASection对象

该Text接口表示在XML文档中的文本。如果解析器和DOM实现支持DOM的XML扩展，那么封装在CDATA标记节中的部分文本将存储在CDATASection对象中。这两个接口是相同的，但为nodeType属性提供不同的值 。

这些接口扩展了Node接口。他们不能有子节点。

Text.data
文本节点的内容为字符串。

注意: CDATASection节点的使用并不表示节点代表完整的CDATA标记部分，只是节点的内容是CDATA部分的一部分。单个CDATA部分可以由文档树中的多个节点表示。无法确定两个相

邻CDATASection节点是否代表不同的CDATA标记节。

20.6.2.11。ProcessingInstruction对象

表示XML文档中的处理指令; 这从Node接口继承 并且不能有子节点。

ProcessingInstruction.**target**
处理指令的内容直至第一个空白字符。这是一个只读属性。

ProcessingInstruction.**data**
处理指令的内容在第一个空白字符之后。

20.6.2.12。例外

DOM Level 2建议定义了一个例外，DOMException以及许多允许应用程序确定发生的错误类型的常量。DOMException实例携带一个code为特定异常提供适当值的属性。

Python DOM接口提供了常量，但也扩展了一组异常，以便DOM中定义的每个异常代码都存在特定的异常。这些实现必须引发适当的特定异常，每个异常都带有相应的code 属性值。

*异常*xml.dom.**DOMException**
用于所有特定DOM异常的基本异常类。这个异常类不能直接实例化。

*异常*xml.dom.**DomstringSizeErr**
当指定范围的文本不适合字符串时引发。这不是已知用于Python DOM实现，但可能从未使用Python编写的DOM实现中获得。

*异常*xml.dom.**HierarchyRequestErr**
当尝试插入不允许节点类型的节点时引发。

*异常*xml.dom.**IndexSizeErr**
当方法的索引或大小参数为负值或超出允许值时引发。

*异常*xml.dom.**InuseAttributeErr**
尝试插入Attr已存在于文档中其他位置的节点时引发。

*异常*xml.dom.**InvalidAccessErr**
如果基础对象不支持参数或操作，则引发。

*异常*xml.dom.**InvalidCharacterErr**
如果字符串参数包含在XML 1.0建议中使用的上下文中不允许的字符，则会引发此异常。例如，尝试Element在元素类型名称中创建具有空格的节点会导致此错误的发生。

*异常*xml.dom.**InvalidModificationErr**
当尝试修改节点的类型时引发。

*异常*xml.dom.**InvalidStateErr**

当尝试使用未定义或不再可用的对象时引发。

*异常*xml.dom.NamespaceErr
如果尝试以XML 建议中的名称空间不允许的方式更改任何对象，则会引发此异常。

*异常*xml.dom.NotFoundErr
在引用的上下文中不存在节点时异常。例如，NamedNodeMap.removeNamedItem()如果传入的节点不存在于地图中， 则会引发此问题。

*异常*xml.dom.NotSupportedErr
当实现不支持请求类型的对象或操作时引发。

*异常*xml.dom.NoDataAllowedErr
如果为不支持数据的节点指定了数据，则会引发此问题。

*异常*xml.dom.NoModificationAllowedErr
引发尝试修改不允许修改的对象（例如对于只读节点）。

*异常*xml.dom.SyntaxErr
当指定了无效或非法字符串时引发。

*异常*xml.dom.WrongDocumentErr
当节点插入到与当前所属不同的文档中时引发，并且实现不支持将节点从一个文档迁移到另一个文档。

DOM建议中定义的异常代码根据此表映射到上述异常：

不变	例外
DOMSTRING_SIZE_ERR	DomstringSizeErr
HIERARCHY_REQUEST_ERR	HierarchyRequestErr
INDEX_SIZE_ERR	IndexSizeErr
INUSE_ATTRIBUTE_ERR	InuseAttributeErr
INVALID_ACCESS_ERR	InvalidAccessErr
INVALID_CHARACTER_ERR	InvalidCharacterErr
INVALID_MODIFICATION_ERR	InvalidModificationErr
INVALID_STATE_ERR	InvalidStateErr
NAMESPACE_ERR	NamespaceErr
NOT_FOUND_ERR	NotFoundErr
NOT_SUPPORTED_ERR	NotSupportedErr
NO_DATA_ALLOWED_ERR	NoDataAllowedErr
NO_MODIFICATION_ALLOWED_ERR	NoModificationAllowedErr
SYNTAX_ERR	SyntaxErr
WRONG_DOCUMENT_ERR	WrongDocumentErr

20.6.3。一致性

本节介绍Python DOM API，W3C DOM推荐和Python的OMG IDL映射之间的一致性要求和关系。

20.6.3.1。类型映射

根据下表将DOM规范中使用的IDL类型映射到Python类型。

IDL类型	Python类型
boolean	bool 要么 int
int	int
long int	int
unsigned int	int
DOMString	str 要么 bytes
null	None

20.6.3.2。访问器方法

从OMG IDL到Python attribute的映射以Java映射的方式为IDL 声明定义访问器函数 。映射IDL声明

```
readonly attribute string someValue;
         attribute string anotherValue;
```

产生三种存取器函数：一个“获取”方法someValue （ _get_someValue() ），和“获取”和“设置”为方法anotherValue （ _get_anotherValue()和_set_anotherValue() ）。映射，尤其是不需要的IDL属性是访问正常的Python属性： object.someValue是*不是*工作需要，并且可能引发AttributeError。

但是，Python DOM API *确实*需要正常的属性访问。这意味着由Python IDL编译器生成的典型代理不可能工作，并且如果通过CORBA访问DOM对象，则可能需要在客户端上包装对象。虽然这确实需要对CORBA DOM客户端进行一些额外的考虑，但具有从Python使用DOM到CORBA的经验的实施者并不认为这是一个问题。所声明的属性readonly可能不会限制所有DOM实现中的写入访问。

在Python DOM API中，访问函数不是必需的。如果提供，它们应该采用由Python IDL映射定义的形式，但这些方法被认为是不必要的，因为可以直接从Python访问这些属性。不应该为readonly属性提供“设置”访问器。

IDL定义并未完全体现W3C DOM API的要求，例如某些对象的概念，如返回值为getElementsByTagName()“live”。Python DOM API不需要实施来执行这些要求。

20.7。 xml.dom.minidom- 最小的DOM实现

源代码： Lib / xml / dom / minidom.py

xml.dom.minidom是文档对象模型接口的最小实现，其API类似于其他语言。它的目标是比完整的DOM更简单，也更小。不熟悉DOM的用户应考虑使用该 xml.etree.ElementTree模块进行XML处理。

警告： 该xml.dom.minidom模块对恶意构建的数据不安全。如果您需要解析不可信或未经身份验证的数据，请参阅XML漏洞。

通常，DOM应用程序首先将一些XML解析为DOM。有了 xml.dom.minidom ，这是通过解析函数完成的：

```
from xml.dom.minidom import parse, parseString

dom1 = parse('c:\\temp\\mydata.xml')  # parse an XML file by name

datasource = open('c:\\temp\\mydata.xml')
dom2 = parse(datasource)  # parse an open file

dom3 = parseString('<myxml>Some data<empty/> some more data</myxml>')
```

该parse()函数可以采用文件名或打开的文件对象。

xml.dom.minidom.**parse**(*filename_or_file* , *parser = None* , *bufsize = None*)

Document从给定的输入中返回a 。*filename_or_file*可以是文件名，也可以是文件类对象。*解析器*（如果有的话）必须是SAX2解析器对象。该函数将更改解析器的文档处理程序并激活命名空间支持; 其他解析器配置（如设置实体解析器）必须事先完成。

如果你在一个字符串中有XML，你可以使用这个parseString()函数：

xml.dom.minidom.**parseString**(*string* , *parser = None*)

返回一个Document代表*字符串的字符串*。此方法io.StringIO为该字符串创建一个 对象并将其传递给该字符串parse()。

这两个函数都会返回一个Document代表文档内容的对象。

什么parse()和parseString()功能做的是一个“DOM生成器”，可以从任何SAX解析器解析接受事件并将它们转换成DOM树连接XML解析器。这些功能的名称可能是误导性的，但在学习接口时很容易理解。文档的解析将在这些函数返回之前完成; 只是这些函数本身不提供解析器实现。

您也可以Document通过调用“DOM实现”对象上的方法来创建一个。您可以通过调用包或 模块中的getDOMImplementation()函数来获取此对象 。一旦你有了，你可以添加子节点来填充DOM：xml.domxml.dom.minidomDocument

```
from xml.dom.minidom import getDOMImplementation
```

```
impl = getDOMImplementation()

newdoc = impl.createDocument(None, "some_tag", None)
top_element = newdoc.documentElement
text = newdoc.createTextNode('Some textual content.')
top_element.appendChild(text)
```

一旦有了DOM文档对象，就可以通过其属性和方法访问XML文档的各个部分。这些属性在DOM规范中定义。文档对象的主要属性是 documentElement属性。它为您提供了XML文档中的主要元素：包含所有其他元素的元素。这是一个示例程序：

```
dom3 = parseString("<myxml>Some data</myxml>")
assert dom3.documentElement.tagName == "myxml"
```

当您完成DOM树时，您可以选择调用该 unlink() 方法来鼓励早期清理不需要的对象。unlink()是xml.dom.minidomDOM API的一个特定扩展，它呈现节点及其后代实质上是无用的。否则，Python的垃圾收集器将最终处理树中的对象。

也可以看看:

文档对象模型（DOM）1级规范
W3C支持的W3C推荐xml.dom.minidom。

20.7.1。DOM对象

Python的DOM API的定义是作为 xml.dom 模块文档的一部分给出的。本节列出了API和xml.dom.minidom。

Node.**unlink**( )
打破DOM内的内部引用，以便在没有循环GC的情况下在Python版本上进行垃圾回收。即使循环GC可用，使用它也可以更快地提供大量内存，因此，在不再需要DOM对象时立即调用它是很好的做法。这只需要在Document对象上调用，但可以在子节点上调用以放弃该节点的子节点。

您可以避免使用该with 语句显式调用此方法。下面的代码将自动取消关联*DOM*当 with退出块：

```
with xml.dom.minidom.parse(datasource) as dom:
    ... # Work with dom.
```

Node.**writexml**(*writer*, *indent*=“”, *addindent*=“”, *newl*=“”)
将XML写入作者对象。作者应该有一个write()匹配文件对象接口的方法。的*缩进*的参数是当前节点的缩进。该*addindent*参数是增量压痕要用于当前的子节点。该*的NewL*参数指定用于终止换行符的字符串。

对于Document节点，可以使用附加的关键字参数*编码*来指定XML标头的编码字段。

Node.**toxml**(*encoding* = *None*)

返回包含DOM节点表示的XML的字符串或字节字符串。

使用显式*编码* [1]参数，结果是指定编码中的字节字符串。没有*编码*参数时，结果是一个Unicode字符串，并且结果字符串中的XML声明不指定编码。使用UTF-8以外的编码对此字符串进行编码可能不正确，因为UTF-8是XML的默认编码。

Node.**toprettyxml**(*indent=""*, *newl=""*, *encoding=""*)

返回文档的漂亮版本。*indent*指定缩进字符串，并默认为制表符; *newl*指定在每行末尾发射的字符串，默认为`\n`。

的*编码*参数的行为等的相应的参数 `toxml()`。

20.7.2。DOM示例

这个示例程序是一个简单程序的相当现实的例子。在这种特殊情况下，我们没有充分利用DOM的灵活性。

```
import xml.dom.minidom

document = """\
<slideshow>
<title>Demo slideshow</title>
<slide><title>Slide title</title>
<point>This is a demo</point>
<point>Of a program for processing slides</point>
</slide>

<slide><title>Another demo slide</title>
<point>It is important</point>
<point>To have more than</point>
<point>one slide</point>
</slide>
</slideshow>
"""

dom = xml.dom.minidom.parseString(document)

def getText(nodelist):
    rc = []
    for node in nodelist:
        if node.nodeType == node.TEXT_NODE:
            rc.append(node.data)
    return ''.join(rc)

def handleSlideshow(slideshow):
    print("<html>")
    handleSlideshowTitle(slideshow.getElementsByTagName("title")[0])
    slides = slideshow.getElementsByTagName("slide")
    handleToc(slides)
    handleSlides(slides)
    print("</html>")

def handleSlides(slides):
    for slide in slides:
```

```
        handleSlide(slide)

def handleSlide(slide):
    handleSlideTitle(slide.getElementsByTagName("title")[0])
    handlePoints(slide.getElementsByTagName("point"))

def handleSlideshowTitle(title):
    print("<title>%s</title>" % getText(title.childNodes))

def handleSlideTitle(title):
    print("<h2>%s</h2>" % getText(title.childNodes))

def handlePoints(points):
    print("<ul>")
    for point in points:
        handlePoint(point)
    print("</ul>")

def handlePoint(point):
    print("<li>%s</li>" % getText(point.childNodes))

def handleToc(slides):
    for slide in slides:
        title = slide.getElementsByTagName("title")[0]
        print("<p>%s</p>" % getText(title.childNodes))

handleSlideshow(dom)
```

20.7.3。minidom和DOM标准

该xml.dom.minidom模块本质上是一个DOM 1.0兼容的DOM，具有一些DOM 2功能（主要是名称空间功能）。

在Python中使用DOM接口非常简单。以下映射规则适用：

- 接口通过实例对象访问。应用程序不应该自己实例化类; 他们应该使用Document对象上可用的创建者函数。派生接口支持来自基本接口的所有操作（和属性）以及任何新的操作。
- 操作被用作方法。由于DOM只使用in 参数，参数按正常顺序传递（从左到右）。没有可选参数。 void操作返回None。
- IDL属性映射到实例属性。为了与Python的OMG IDL语言映射兼容， foo还可以通过访问器方法_get_foo()和访问属性_set_foo()。 readonly 属性不能改变; 这不是在运行时强制执行的。
- 类型，，，和 所有映射到Python的整数对象。 short intunsigned intunsigned long longboolean
- 该类型DOMString映射到Python字符串。 xml.dom.minidom支持字节或字符串，但通常会产生字符串。类型的值DOMString也可以None在nullW3C的DOM规范中允许具有IDL 值的地方。
- const 声明映射到各自范围内的变量（例如 xml.dom.minidom.Node.PROCESSING_INSTRUCTION_NODE）; 他们不能改变。
- DOMException目前不支持xml.dom.minidom。相反， xml.dom.minidom使用标准的Python例外，例如 TypeError和AttributeError。
- NodeList对象是使用Python的内置列表类型实现的。这些对象提供了DOM规范中定义的接口，但是对于早期版本的Python，它们不支持官方API。然而，它们比W3C建议中定义

的接口更“Pythonic”。

以下接口在以下方面没有实现`xml.dom.minidom`：

- `DOMTimeStamp`
- `DocumentType`
- `DOMImplementation`
- `CharacterData`
- `CDATASection`
- `Notation`
- `Entity`
- `EntityReference`
- `DocumentFragment`

其中大多数反映了XML文档中的信息，这对大多数DOM用户来说并不普遍。

脚注

[1] [illegible] https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl https://www.iana.org/assignments/character-sets/character-sets.xhtml。

20.8。 xml.dom.pulldom- 支持构建部分DOM树

源代码： Lib / xml / dom / pulldom.py

该xml.dom.pulldom模块提供了一个“pull parser”，在需要的时候，它也可以被要求产生DOM可访问的文档片段。基本概念涉及从传入XML流中提取“事件”并对其进行处理。与SAX一起使用事件驱动的处理模型和回调函数相比，拉分析器的用户负责显式地从流中提取事件，循环这些事件直到处理完成或出现错误情况。

警告: 该xml.dom.pulldom模块对恶意构建的数据不安全。如果您需要解析不可信或未经身份验证的数据，请参阅XML漏洞。

例：

```
from xml.dom import pulldom

doc = pulldom.parse('sales_items.xml')
for event, node in doc:
    if event == pulldom.START_ELEMENT and node.tagName == 'item':
        if int(node.getAttribute('price')) > 50:
            doc.expandNode(node)
            print(node.toxml())
```

event 是一个常数，可以是下列之一：

- START_ELEMENT
- END_ELEMENT
- COMMENT
- START_DOCUMENT
- END_DOCUMENT
- CHARACTERS
- PROCESSING_INSTRUCTION
- IGNORABLE_WHITESPACE

node 是类型的对象 xml.dom.minidom.Document ， xml.dom.minidom.Element 或者 xml.dom.minidom.Text。

由于文档被视为一个“扁平”的事件流，文档“树”被隐式遍历，并且无论它们在树中的深度如何，都会找到所需的元素。换句话说，不需要考虑递归搜索文档节点等分层问题，但是如果元素的上下文很重要，那么可能需要维护一些与上下文相关的状态（即记住文档中的文档的位置在任何给定的位置）或使用该DOMEventStream.expandNode()方法并切换到DOM相关处理。

class xml.dom.pulldom.PullDom（*documentFactory = None* ）

子类xml.sax.handler.ContentHandler。

class xml.dom.pulldom.SAX2DOM（*documentFactory = None* ）

子类xml.sax.handler.ContentHandler。

xml.dom.pulldom.parse（*stream_or_string*，*parser* = *None*，*bufsize* = *None*）

DOMEventStream从给定的输入中返回a 。*stream_or_string*可以是文件名，也可以是文件类对象。*解析器*，如果给出，必须是一个 XMLReader对象。该函数将更改解析器的文档处理程序并激活命名空间支持; 其他解析器配置（如设置实体解析器）必须事先完成。

如果你在一个字符串中有XML，你可以使用这个parseString()函数：

xml.dom.pulldom.parseString（*string*，*parser* = *None*）

返回一个DOMEventStream表示（Unicode）*字符串的*。

xml.dom.pulldom.default_bufsize

*bufsize*参数的缺省值parse()。

在调用之前可以更改此变量的值，parse()新值将生效。

20.8.1。DOMEventStream对象

*类*xml.dom.pulldom.DOMEventStream（*流*，*解析器*，*bufsize*）

getEvent（）

返回一个包含元组*事件*和当前*节点*作为 xml.dom.minidom.Document如果事件等于START_DOCUMENT ， xml.dom.minidom.Element 如果事件等于START_ELEMENT 或END_ELEMENT或者xml.dom.minidom.Text如果事件等于 CHARACTERS。除非expandNode()被调用，否则当前节点不包含有关其子节点的信息 。

expandNode（*节点*）

将*节点的*所有子*节点*扩展为*节点*。例：

```
from xml.dom import pulldom

xml = '<html><title>Foo</title> <p>Some text <div>and more</div></p> </html>'
doc = pulldom.parseString(xml)
for event, node in doc:
    if event == pulldom.START_ELEMENT and node.tagName == 'p':
        # Following statement only prints '<p/>'
        print(node.toxml())
        doc.expandNode(node)
        # Following statement prints node with all its children '<p>Some text
        print(node.toxml())
```

reset（）

20.9。 xml.sax- 支持SAX2分析器

源代码： Lib / xml / sax / __ init__.py

该xml.sax软件包提供了许多实现Python简单API（SAX）接口的模块。程序包本身提供SAX异常和SAX API用户最常使用的便利功能。

警告： 该xml.sax模块对恶意构建的数据不安全。如果您需要解析不可信或未经身份验证的数据，请参阅 XML漏洞。

便利功能是：

xml.sax.make_parser（*parser_list = []* ）

创建并返回一个SAX XMLReader对象。找到的第一个解析器将被使用。如果提供了*parser_list*，它必须是一个字符串序列，这些字符串命名具有名为函数的模块create_parser()。*parser_list中*列出的模块将在解析器默认列表中的模块之前使用。

xml.sax.parse（*filename_or_stream*，*handler*，*error_handler = handler.ErrorHandler（）*）

创建一个SAX解析器并用它来解析文档。以*filename_or_stream形式*传入的文档 可以是文件名或文件对象。该*处理器* 的参数必须是一个SAX ContentHandler实例。如果 给出*error_handler*，则它必须是SAX ErrorHandler 实例; 如果省略， SAXParseException将会引发所有错误。没有回报价值; 所有的工作必须由传入的*处理程序*来完成。

xml.sax.parseString（*string*，*handler*，*error_handler = handler.ErrorHandler（）*）

与作为参数收到parse()的缓冲区*字符串*类似，但解析。 *字符串*必须是一个str实例或类似字节的对象。

*在版本3.5中进行了更改：*添加了对str实例的支持。

典型的SAX应用程序使用三种对象：读者，处理程序和输入源。在这种情况下，“读者”是解析器的另一个术语，即某些代码从输入源读取字节或字符，并产生一系列事件。事件然后被分配给处理程序对象，即读者调用处理程序上的方法。因此，SAX应用程序必须获取reader对象，创建或打开输入源，创建处理程序并将这些对象连接在一起。作为准备的最后一步，读者被调用来解析输入。在解析过程中，处理程序对象的方法基于来自输入数据的结构和语法事件来调用。

对于这些对象，只有接口是相关的; 它们通常不是由应用程序本身实例化的。由于Python没有明确的接口概念，因此它们被正式引入为类，但是应用程序可能会使用不从所提供的类继承的实现。的 InputSource ，Locator， Attributes ，AttributesNS ，和XMLReader接口的模块中定义 xml.sax.xmlreader 。 处理程序接口在中定义 xml.sax.handler 。 为了方便，InputSource（通常直接实例化）和处理程序类也可从 xml.sax。这些接口如下所述。

除这些类外，xml.sax还提供以下异常类。

*异常*xml.sax.SAXException（*味精*，*例外=无*）

封装XML错误或警告。该类可以包含来自XML解析器或应用程序的基本错误或警告信息：它可以被分类以提供附加功能或添加本地化。请注意，虽然ErrorHandler界面中定义的处理程序 会接收此异常的实例，但实际并不需要引发异常 - 它也可用作信息容器。

实例化时，*msg*应该是错误的可读描述。可选的*异常*参数（如果给出）应该是None解析代码捕获的异常，并作为信息传递。

这是其他SAX异常类的基类。

*异常*xml.sax.SAXParseException（*msg*，*异常*，*定位器*）

SAXException抛出解析错误的子类。这个类的实例被传递给SAX ErrorHandler接口的方法来提供关于分析错误的信息。该类支持SAX Locator接口以及 SAXException接口。

*异常*xml.sax.SAXNotRecognizedException（*味精*，*例外=无*）

SAXException当SAX XMLReader遇到无法识别的特征或属性时引发的 子类。SAX应用程序和扩展可能会将此类用于类似目的。

*异常*xml.sax.SAXNotSupportedException（*味精*，*例外=无*）

要求SAXExceptionSAX XMLReader启用不受支持的功能，或将属性设置为实现不支持的值时引发的 子类。SAX应用程序和扩展可能会将此类用于类似目的。

也可以看看:

SAX：XML的简单API

本网站是SAX API定义的焦点。它提供了Java实现和在线文档。实现和历史信息的链接也可用。

模 xml.sax.handler

应用程序提供对象的接口定义。

模 xml.sax.saxutils

用于SAX应用程序的便利功能。

模 xml.sax.xmlreader

解析器提供的对象的接口定义。

20.9.1。SAXException对象

该SAXException异常类支持下列方法：

SAXException.getMessage（）

返回描述错误情况的可读信息。

SAXException.getException（）

返回一个封装的异常对象，或者None。

20.10。 xml.sax.handler-基本类SAX处理程序

源代码： Lib / xml / sax / handler.py

SAX API定义了四种类型的处理程序：内容处理程序，DTD处理程序，错误处理程序和实体解析程序。应用程序通常只需要实现那些他们感兴趣的事件的接口; 他们可以在单个对象或多个对象中实现接口。处理程序实现应该从模块中提供的基类继承 xml.sax.handler ，以便所有方法都获得默认实现。

*类*xml.sax.handler.ContentHandler

这是SAX中的主要回调接口，也是应用程序最重要的接口。该界面中的事件顺序反映文档中信息的顺序。

*类*xml.sax.handler.DTDHandler

处理DTD事件。

该接口仅指定基本解析所需的那些DTD事件（未解析的实体和属性）。

*类*xml.sax.handler.EntityResolver

用于解析实体的基本界面。如果你创建了一个实现这个接口的对象，然后用你的解析器注册这个对象，解析器会调用你的对象中的方法来解析所有的外部实体。

*类*xml.sax.handler.ErrorHandler

解析器用于向应用程序显示错误和警告消息的接口。该对象的方法控制错误是立即转换为异常还是以其他方式处理。

除了这些类以外， xml.sax.handler还为特征和属性名称提供了符号常量。

xml.sax.handler.feature_namespaces

值： "http://xml.org/sax/features/namespaces"
true：执行名称空间处理。
false：（可选）不执行命名空间处理（隐含命名空间前缀;默认）。
访问:(解析）只读;（不解析）读/写

xml.sax.handler.feature_namespace_prefixes

值： "http://xml.org/sax/features/namespace-prefixes"
true：报告用于名称空间声明的原始前缀名称和属性。
false：不报告用于名称空间声明的属性，也可以不报告原始的前缀名称（默认）。
访问:(解析）只读;（不解析）读/写

xml.sax.handler.feature_string_interning

值： "http://xml.org/sax/features/string-interning"
true：所有元素名称，前缀，属性名称，名称空间URI和本地名称都使用内置的intern函数来实现。
false：名称不一定是实习的，尽管它们可能是（默认）。
访问:(解析）只读;（不解析）读/写

xml.sax.handler.**feature_validation**

值：*"http://xml.org/sax/features/validation"*
true：报告所有验证错误（隐含外部一般实体和外部参数实体）。
false：不报告验证错误。
访问:(解析）只读;（不解析）读/写

xml.sax.handler.**feature_external_ges**

值：*"http://xml.org/sax/features/external-general-entities"*
true：包含所有外部一般（文本）实体。
假：不包括外部一般实体。
访问:(解析）只读;（不解析）读/写

xml.sax.handler.**feature_external_pes**

值：*"http://xml.org/sax/features/external-parameter-entities"*
true：包含所有外部参数实体，包括外部DTD子集。
false：不包含任何外部参数实体，即使是外部DTD子集。
访问:(解析）只读;（不解析）读/写

xml.sax.handler.**all_features**
所有功能列表。

xml.sax.handler.**property_lexical_handler**

值：*"http://xml.org/sax/properties/lexical-handler"*
数据类型：xml.sax.sax2lib.LexicalHandler（在Python 2中不支持）
描述：用于注释等词汇事件的可选扩展处理程序。
访问：读/写

xml.sax.handler.**property_declaration_handler**

值：*"http://xml.org/sax/properties/declaration-handler"*
数据类型：xml.sax.sax2lib.DeclHandler（在Python 2中不支持）
描述：用于DTD相关事件的可选扩展处理程序，而不是符号和未分离的实体。
访问：读/写

xml.sax.handler.**property_dom_node**

值：*"http://xml.org/sax/properties/dom-node"*
数据类型：org.w3c.dom.Node（在Python 2中不支持）
描述：解析时，如果这是一个DOM迭代器，则访问当前DOM节点; 当不解析时，用于迭代的根DOM节点。
访问:(解析）只读;（不解析）读/写

xml.sax.handler.**property_xml_string**

值：*"http://xml.org/sax/properties/xml-string"*
数据类型：字符串
描述：作为当前事件源的文字字符串。
访问权限：只读

xml.sax.handler.**all_properties**
所有已知属性名称的列表。

20.10.1。ContentHandler对象

用户需要继承ContentHandler以支持他们的应用程序。解析器在输入文档中的相应事件中调用以下方法：

ContentHandler.**setDocumentLocator**(*locator*)
由解析器调用，为应用程序提供一个用于定位文档事件来源的定位器。

强烈建议SAX解析器（尽管不是绝对必要）提供定位器：如果这样做，它必须在调用DocumentHandler接口中的任何其他方法之前通过调用此方法将定位器提供给应用程序。

定位器允许应用程序确定任何文档相关事件的结束位置，即使解析器未报告错误。通常情况下，应用程序将使用此信息报告自己的错误（例如与应用程序业务规则不匹配的字符内容）。由定位器返回的信息可能不足以用于搜索引擎。

请注意，只有在调用此界面中的事件时，定位器才会返回正确的信息。应用程序不应该尝试在任何其他时间使用它。

ContentHandler.**startDocument**( )
接收文档开始的通知。

SAX解析器只会在此接口或DTDHandler中的任何其他方法（除外setDocumentLocator()）之前调用此方法一次。

ContentHandler.**endDocument**( )
接收文档结束的通知。

SAX解析器只会调用一次该方法，它将是解析过程中调用的最后一个方法。解析器不应该调用此方法，直到它放弃解析（由于错误不可恢复）或到达输入的结尾。

ContentHandler.**startPrefixMapping**(*前缀*, *uri*)
开始前缀-URI命名空间映射的范围。

来自此事件的信息对于正常的命名空间处理不是必需的：SAX XML阅读器将在feature_namespaces启用功能（默认）时自动替换元素和属性名称的前缀。

然而，有些情况下，当应用程序需要在字符数据或属性值中使用前缀时，它们无法安全地自动扩展; 如果需要的话startPrefixMapping()，endPrefixMapping()事件提供信息给应用程序以扩展这些上下文中的前缀。

需要注意的是startPrefixMapping()，并endPrefixMapping()不能保证事件相对正确嵌套到每个-其他：所有 startPrefixMapping()事件都将相应发生之前 startElement()的事件，以及所有endPrefixMapping()相应的会后发生的事件endElement()的事件，但他们的顺序是不能保证。

ContentHandler.endPrefixMapping(*前缀*)

结束前缀-URI映射的范围。

详情请参阅startPrefixMapping()。该事件将始终发生在相应的endElement()事件之后，但事件的顺序 endPrefixMapping()不另外得到保证。

ContentHandler.startElement(*name*, *attrs*)

在非命名空间模式下指示元素的开始。

的*名称*参数包含的元素类型为字符串的原始XML 1.0的名称和*ATTRS*参数保存的目的Attributes 接口（参见Attributes接口包含该元素的属性）。作为*attrs*传递的对象可以被解析器重用; 坚持引用它并不是保持属性副本的可靠方法。要保留属性的副本，请使用*attrs*对象的copy()方法。

ContentHandler.endElement(*名字*)

以非命名空间模式表示元素的结尾。

该*名称*参数中包含的元素类型的名称，就像用 startElement()事件。

ContentHandler.startElementNS(*name*, *qname*, *attrs*)

在名称空间模式中指定元素的开始。

*name*参数将元素类型的*名称*包含为元组，*qname*参数包含源文档中使用的原始XML 1.0名称，*attrs*参数包含接口的实例（请参阅 AttributesNS接口），其中包含元件。如果没有名称空间与元素相关联，则*名称*的*uri*组件将会是。作为*attrs*传递的对象可以被解析器重用; 坚持引用它并不是保持属性副本的可靠方法。要保留属性的副本，请使用*attrs*对象的方法。(uri, localname)AttributesNSNonecopy()

解析器可以将*qname*参数设置为None，除非该 feature_namespace_prefixes功能被激活。

ContentHandler.endElementNS(*name*, *qname*)

以命名空间模式标记元素的结尾。

的*名称*参数包含元素类型的名称，就像用 startElementNS()方法，同样地*的QName*参数。

ContentHandler.characters(*内容*)

接收字符数据的通知。

解析器将调用此方法来报告每个字符数据块。SAX解析器可能会将所有连续的字符数据返回到单个块中，或者它们可能会将其分割为多个块; 但是，任何单个事件中的所有字符必须来自相同的外部实体，以便定位器提供有用的信息。

*内容*可能是一个字符串或字节实例; 该expat读写器模块总是会产生字符串。

> **注意：** 由Python XML Special Interest Group提供的较早的SAX 1接口为此方法使用了更类似于Java的接口。由于大多数使用Python的解析器都没有利用旧版界面，所以选择了更简单的签名来代替它。要将旧代码转换为新界面，请使用*内容*而不是使用旧的*偏移量*和*长度*参数对*内容*进行分片。

ContentHandler.ignorableWhitespace（*空格*）

接收元素内容中可忽略空白的通知。

验证解析器必须使用此方法报告每个可忽略的空白块（请参阅W3C XML 1.0建议的第2.10节）：如果非验证解析器能够解析和使用内容模型，则也可以使用此方法。

SAX解析器可能会将单个块中的所有连续空白值返回，或者可能会将其分割为多个块; 但是，任何单个事件中的所有字符必须来自相同的外部实体，以便定位器提供有用的信息。

ContentHandler.processingInstruction（*目标*，*数据*）

接收处理指令的通知。

解析器将为找到的每个处理指令调用一次该方法：请注意，处理指令可能发生在主文档元素之前或之后。

SAX解析器不应使用此方法报告XML声明（XML 1.0，第2.8节）或文本声明（XML 1.0，第4.3.1节）。

ContentHandler.skippedEntity（*名字*）

接收跳过实体的通知。

解析器将为每个跳过的实体调用一次该方法。未验证的处理器可能会在实体没有看到声明时跳过实体（例如，因为实体是在外部DTD子集中声明的）。所有处理器可能会跳过外部实体，具体取决于feature_external_ges该feature_external_pes属性的值。

20.10.2。DTDHandler对象

DTDHandler 实例提供以下方法：

DTDHandler.notationDecl（*name*，*publicId*，*systemId* ）

处理符号声明事件。

DTDHandler.unparsedEntityDecl（*name*，*publicId*，*systemId*，*ndata* ）

处理未解析的实体声明事件。

20.10.3。EntityResolver的对象

EntityResolver.resolveEntity（*publicId*，*systemId* ）

解析实体的系统标识符，并返回要读取的系统标识符作为字符串或要读取的InputSource。缺省实现返回*systemId*。

20.10.4。ErrorHandler对象

具有此接口的对象用于从中接收错误和警告信息XMLReader。如果你创建了一个实现这个接口的对象，然后注册这个对象 XMLReader，解析器会调用你的对象中的方法来报告所有的警告和错

误。有三个级别的错误可用：警告，（可能）可恢复的错误和不可恢复的错误。所有方法都SAXParseException只有一个参数。错误和警告可能会通过引发传入的异常对象转换为异常。

ErrorHandler.error(*例外*)

当解析器遇到可恢复的错误时调用。如果此方法不引发异常，则解析可能会继续，但应用程序不应该期望进一步的文档信息。允许解析器继续可能允许在输入文档中发现其他错误。

ErrorHandler.fatalError(*例外*)

当解析器遇到错误时调用它无法从中恢复;当此方法返回时，预计解析将终止。

ErrorHandler.warning(*例外*)

当解析器向应用程序提供次要警告信息时调用。当此方法返回时，分析预计会继续，并且文档信息将继续传递给应用程序。在此方法中引发异常将导致解析结束。

20.11。 xml.sax.saxutils- SAX实用程序

源代码： Lib / xml / sax / saxutils.py

该模块xml.sax.saxutils包含许多在创建SAX应用程序时通常很有用的类和函数，无论是直接使用还是作为基类。

xml.sax.saxutils.escape（*data*，*entities = {}* ）

转义'&'，'<'和'>'一串数据。

您可以通过传递字典作为可选*实体*参数来转义其他数据字符串。键和值必须全部是字符串;每个键将被替换为其对应的值。字符'&'，'<'并且'>'总是逃脱，即使*实体*提供。

xml.sax.saxutils.unescape（*data*，*entities = {}* ）

Unescape '&'，'<'和'>'一串数据。

通过传递字典作为可选*实体*参数，您可以忽略其他数据字符串。键和值必须全部是字符串;每个键将被替换为其对应的值。 '&amp'，'<'并且'>'始终未转义，即使提供了*实体*。

xml.sax.saxutils.quoteattr（*data*，*entities = {}* ）

类似escape()，也准备*数据*被用作一个属性值。返回值是带有任何额外所需替换的*数据*的引用版本。 quoteattr()将根据*数据*内容选择一个引号字符，试图避免编码字符串中的任何引号字符。如果单引号和双引号字符都在*数据中*，则双引号字符将被编码，*数据*将用双引号括起来。结果字符串可以直接用作属性值：

```
>>> print("<element attr=%s>" % quoteattr("ab ' cd \" ef"))
<element attr="ab ' cd &quot; ef">
```

当使用引用具体语法为HTML或任何SGML生成属性值时，此函数很有用。

class xml.sax.saxutils.XMLGenerator（*out = None*，*encoding ='iso-8859-1'*，*short_empty_elements = False* ）

该类ContentHandler通过将SAX事件写回XML文档来实现接口。换句话说，使用一个XMLGenerator作为内容处理程序将重现正在解析的原始文档。*out*应该是一个类似文件的对象，默认为*sys.stdout*。*编码*是默认的输出流的编码'iso-8859-1'。*short_empty_elements*控制不包含内容的元素的格式：if False（缺省值）它们作为一对开始/结束标签True发出，如果设置为它们作为单个自闭合标签发出。

*新版本3.2：*在*short_empty_elements*参数。

class xml.sax.saxutils.XMLFilterBase（*base* ）

这个类被设计为位于XMLReader客户端应用程序的事件处理程序之间。默认情况下，它没有做任何事情，只是将请求传递给读写器并将事件传递给未经修改的处理程序，但子类可以覆盖特定的方法以在事件流或配置请求通过时修改它们。

xml.sax.saxutils.prepare_input_source(*source*, *base*='')

该函数接受一个输入源和一个可选的基本URL，并返回一个完全解析的InputSource对象，以供阅读。输入源可以以字符串，文件类对象或InputSource对象的形式给出; 解析器将使用此函数为其方法实现多态*源*参数parse()。

20.12。 xml.sax.xmlreader- XML解析器的接口

源代码： Lib / xml / sax / xmlreader.py

SAX解析器实现XMLReader接口。它们在一个Python模块中实现，它必须提供一个函数create_parser()。这个函数被调用 xml.sax.make_parser()没有参数来创建一个新的分析器对象。

*类*xml.sax.xmlreader.XMLReader

可由SAX解析器继承的基类。

*类*xml.sax.xmlreader.IncrementalParser

在某些情况下，最好不要一次解析输入源，而是在文档可用时向其提供大块文件。请注意，读者通常不会读取整个文件，而是以块读取; parse() 在整个文档被处理之前仍然不会返回。所以如果阻塞行为parse()不可取，应该使用这些接口。

当解析器被实例化时，它已经准备好立即开始接收来自feed方法的数据。解析完成后，通过调用关闭重置方法，必须调用解析器以使分析器准备好接收新数据，无论是从feed还是使用parse方法。

请注意，在解析期间*不得*调用这些方法，即在调用parse之后并返回之前。

默认情况下，为了方便SAX 2.0驱动程序编写器，类还使用IncrementalParser接口的feed，close和reset方法实现XMLReader接口的解析方法。

*类*xml.sax.xmlreader.Locator

用于将SAX事件与文档位置相关联的界面。只有在调用DocumentHandler方法时，定位器对象才会返回有效的结果; 在任何其他时间，结果都是不可预测的。如果信息不可用，方法可能会返回None。

class xml.sax.xmlreader.InputSource (*system_id = None*)

封装XMLReader读取实体所需的信息。

该类可以包括关于公共标识符，系统标识符，字节流（可能具有字符编码信息）和/或实体的字符流的信息。

应用程序将创建此类的对象以用于该 XMLReader.parse() 方法并从EntityResolver.resolveEntity返回。

一个InputSource属于应用程序时，XMLReader不允许修改InputSource从应用程序传递给它的对象，尽管它可能使拷贝和修改的。

class xml.sax.xmlreader.AttributesImpl (*attrs*)

这是Attributes接口的实现（请参见属性接口一节 ）。这是一个类似字典的对象，表示startElement()调用中的元素属性。除了最有用的字典操作之外，它还支持接口描述的其

他许多方法。这个类的对象应该由读者实例化; *attrs*必须是一个类似字典的对象，包含从属性名称到属性值的映射。

class xml.sax.xmlreader.AttributesNSImpl（*attrs*，*qnames* ）

命名空间感知变体AttributesImpl ，将被传递给 startElementNS()。它是从派生而来的AttributesImpl ，但将属性名称理解为*namespaceURI*和 *localname*的二元组。另外，它提供了许多预期在原始文档中出现的合格名称的方法。该类实现了该 AttributesNS接口（请参见属性NS接口一节）。

20.12.1。XMLReader对象

该XMLReader接口支持以下方法：

XMLReader.parse（*源*）

处理输入源，产生SAX事件。的*源*对象可以是一个系统标识符（字符串标识输入源-通常是文件名或URL），一个类文件对象，或一个InputSource对象。当 parse()返回时，输入被完全处理，并且解析器对象可以被丢弃或复位。

*在版本3.5中进行了更改：*添加了对字符流的支持。

XMLReader.getContentHandler（ ）

返回当前ContentHandler。

XMLReader.setContentHandler（*处理程序*）

设置当前ContentHandler。如果ContentHandler设置为no ，则内容事件将被丢弃。

XMLReader.getDTDHandler（ ）

返回当前DTDHandler。

XMLReader.setDTDHandler（*处理程序*）

设置当前DTDHandler。如果no DTDHandler设置，则DTD事件将被丢弃。

XMLReader.getEntityResolver（ ）

返回当前EntityResolver。

XMLReader.setEntityResolver（*处理程序*）

设置当前EntityResolver。如果EntityResolver设置为no ，则尝试解析外部实体将导致打开实体的系统标识符，如果不可用，将失败。

XMLReader.getErrorHandler（ ）

返回当前ErrorHandler。

XMLReader.setErrorHandler（*处理程序*）

设置当前的错误处理程序。如果ErrorHandler 设置为“否”，则将引发错误作为例外，并打印警告。

XMLReader.setLocale（*locale* ）

允许应用程序为错误和警告设置区域设置。

SAX解析器不需要为错误和警告提供本地化; 如果他们不能支持所请求的语言环境，则必须提出SAX异常。应用程序可能会在解析过程中请求区域设置更改。

XMLReader.getFeature(*featurename*)
返回功能要素*名称*的当前设置。如果该特征未被识别， SAXNotRecognizedException则引发。众所周知的功能名称在模块中列出xml.sax.handler。

XMLReader.setFeature(*featurename*, *value*)
将*featurename*到*值*。如果该特征未被识别， SAXNotRecognizedException则引发。如果解析器不支持该功能或其设置，则会引发*SAXNotSupportedException*。

XMLReader.getProperty(*propertyname*)
返回属性*propertyname*的当前设置。如果财产不被承认，SAXNotRecognizedException则会提出A. 熟知的属性名称在模块中列出xml.sax.handler。

XMLReader.setProperty(*propertyname*, *value*)
将*属性名称*设置为*值*。如果财产不被承认， SAXNotRecognizedException则会被提出。如果解析器不支持该属性或其设置，则会引发*SAXNotSupportedException*。

20.12.2。IncrementalParser对象

IncrementalParser提供以下附加方法的实例：

IncrementalParser.feed(*数据*)
处理大量*数据*。

IncrementalParser.close( )
假设文档结束。这将检查只能在最后检查的格式良好条件，调用处理程序，并可以清理在分析过程中分配的资源。

IncrementalParser.reset( )
调用close之后调用此方法来重置解析器，以便它可以解析新文档。调用解析或关闭后调用结果而不调用重置是未定义的。

20.12.3。定位器对象

Locator提供这些方法的实例：

Locator.getColumnNumber( )
返回当前事件开始的列号。

Locator.getLineNumber( )
返回当前事件开始的行号。

Locator.getPublicId ( )
返回当前事件的公共标识符。

Locator.getSystemId ( )
返回当前事件的系统标识符。

20.12.4。InputSource的对象

InputSource.setPublicId (*id*)
设置这个的公共标识符InputSource。

InputSource.getPublicId ( )
返回这个的公共标识符InputSource。

InputSource.setSystemId (*id*)
设置这个的系统标识符InputSource。

InputSource.getSystemId ( )
返回这个的系统标识符InputSource。

InputSource.setEncoding (*编码*)
设置这个的字符编码InputSource。

编码必须是XML编码声明可接受的字符串（请参阅XML建议的4.3.3节）。

InputSource如果InputSource字符流还包含字符流，则该属性的编码属性将被忽略。

InputSource.getEncoding ( )
获取此InputSource的字符编码。

InputSource.setByteStream (*bytefile*)
为此输入源设置字节流（二进制文件）。

如果还有指定的字符流，则SAX解析器将忽略此操作，但它将使用字节流来优先打开URI连接本身。

如果应用程序知道字节流的字符编码，则应使用setEncoding方法对其进行设置。

InputSource.getByteStream ( )
获取该输入源的字节流。

getEncoding方法将返回此字节流的字符编码，或者None未知。

InputSource.setCharacterStream (*charfile*)
为此输入源设置字符流（一个文本文件）。

如果指定了字符流，则SAX解析器将忽略任何字节流，并且不会尝试打开与系统标识符的URI连接。

InputSource.getCharacterStream()
获取此输入源的字符流。

20.12.5。该Attributes接口

Attributes对象实现了的一部分映射协议，包括方法copy()， get()，__contains__()，items()，keys()，和values()。还提供了以下方法：

Attributes.getLength()
返回属性的数量。

Attributes.getNames()
返回属性的名称。

Attributes.getType(*名字*)
返回属性*名称*的类型，通常是'CDATA'。

Attributes.getValue(*名字*)
返回属性*名称*的值。

20.12.6。该AttributesNS接口

此接口是接口的子类型Attributes（请参见属性接口一节）。该接口支持的所有方法也可用于AttributesNS对象。

以下方法也可用：

AttributesNS.getValueByQName(*名字*)
返回合格名称的值。

AttributesNS.getNameByQName(*名字*)
返回一对合格的*名字*。(namespace, localname)

AttributesNS.getQNameByName(*名字*)
返回一对的限定名称。(namespace, localname)

AttributesNS.getQNames()
返回所有属性的限定名称。

20.13。 xml.parsers.expat- 使用Expat进行快速XML解析

警告: 该pyexpat模块对恶意构建的数据不安全。如果您需要解析不可信或未经身份验证的数据，请参阅 XML漏洞。

该 xml.parsers.expat 模块是Expat非验证XML解析器的Python接口。该模块提供了一个xmlparser表示XML解析器当前状态的单个扩展类型 。在xmlparser创建对象后，可以将对象的各种属性设置为处理函数。然后，当XML文档被提供给解析器时，处理函数被调用用于XML文档中的字符数据和标记。

该模块使用该pyexpat模块提供对Expat解析器的访问。直接使用pyexpat模块已被弃用。

该模块提供一个例外和一个类型对象：

*异常*xml.parsers.expat.ExpatError
当Expat报告错误时引发异常。有关解释Expat错误的更多信息，请参见ExpatError异常一节 。

*异常*xml.parsers.expat.error
别名ExpatError。

xml.parsers.expat.XMLParserType
ParserCreate()函数返回值的类型。

该xml.parsers.expat模块包含两个功能：

xml.parsers.expat.ErrorString（ *errno* ）
返回给定错误号*errno*的说明性字符串。

xml.parsers.expat.ParserCreate（ *encoding = None*，*namespace_separator = None* ）

创建并返回一个新的xmlparser对象。 *编码*（如果指定）必须是一个字符串，用于命名XML数据使用的编码。Expat不支持Python那样多的编码，编码的编码不能扩展; 它支持UTF-8，UTF-16，ISO-8859-1（Latin1）和ASCII。如果给出 *编码* [1]，它将覆盖文档的隐式或显式编码。

Expat可以选择为您执行XML名称空间处理，通过为*namespace_separator*提供值来启用它。该值必须是一个字符的字符串; 一个 ValueError如果字符串具有非法长度将被升高（None 被认为是相同的遗漏）。当启用名称空间处理时，属于名称空间的元素类型名称和属性名称将被展开。元素名称传递给元素处理器 StartElementHandler 以及EndElementHandler将命名空间URI的串联，命名空间分隔符，和名称的本地部分。如果名称空间分隔符是零字节（chr(0)），则名称空间URI和本地部分将连接在一起，而不用任何分隔符。

例如，如果*namespace_separator*被设置为空格字符（ ）并且解析了以下文档：’ ’

```
<?xml version="1.0"?>
<root xmlns    = "http://default-namespace.org/"
      xmlns:py = "http://www.python.org/ns/">
  <py:elem1 />
  <elem2 xmlns="" />
</root>
```

StartElementHandler 将为每个元素接收以下字符串：

```
http://default-namespace.org/ root
http://www.python.org/ns/ elem1
elem2
```

由于Expat使用的库的限制，返回pyexpat的xmlparser实例只能用于解析单个XML文档。调用ParserCreate每个文档以提供唯一的解析器实例。

也可以看看:

Expat XML解析器
Expat项目的主页。

20.13.1。XMLParser对象

xmlparser 对象有以下方法：

xmlparser.Parse (*data* [, *isfinal*])
分析字符串*数据*的内容，调用适当的处理函数来处理解析的数据。 *isfinal*必须对这种方法最后调用真实的; 它允许解析片段中的单个文件，而不是提交多个文件。 *数据*可以是任何时候的空字符串。

xmlparser.ParseFile (*文件*)
从目标*文件中*解析XML数据读取。 *文件*只需要提供read(nbytes)方法，当没有更多数据时返回空字符串。

xmlparser.SetBase (*基地*)
设置声明中用于解析系统标识符中的相对URI的基数。解析相关标识符被留给应用：此值将通过作为传递*基础*参数的 ExternalEntityRefHandler() ， NotationDeclHandler() 和 UnparsedEntityDeclHandler()功能。

xmlparser.GetBase ( )
返回包含基本由先前的呼叫建立一个字符串SetBase() ，或者None如果 SetBase()没有被调用。

xmlparser.GetInputContext ( )
以字符串形式返回生成当前事件的输入数据。数据在包含文本的实体的编码中。在事件处理程序未处于活动状态时调用时，返回值为None。

xmlparser.ExternalEntityParserCreate (*context* [, *encoding*])

创建一个“子”解析器，它可以用来解析由父解析器解析的内容引用的外部解析实体。该*情境*参数应该是传递给字符串ExternalEntityRefHandler()处理函数，如下所述。孩子分析器与创建 ordered_attributes和specified_attributes设置这个解析器的值。

xmlparser.SetParamEntityParsing(*标志*)

控制参数实体的解析（包括外部DTD子集）。可能的*标志*值是XML_PARAM_ENTITY_PARSING_NEVER，XML_PARAM_ENTITY_PARSING_UNLESS_STANDALONE并且XML_PARAM_ENTITY_PARSING_ALWAYS。如果设置标志成功，则返回true。

xmlparser.UseForeignDTD([*flag*])

使用*flag*的真实值（默认值）调用此函数将导致Expat为所有参数调用ExternalEntityRefHandlerwith None以允许加载备用DTD。如果文档不包含文档类型声明中，ExternalEntityRefHandler仍然会被调用，但StartDoctypeDeclHandler并EndDoctypeDeclHandler不会被调用。

为*标志*传递一个错误的值将取消之前通过一个真值的调用，但是否则不起作用。

此方法只能在调用Parse() or ParseFile() 方法之前调用; 在调用其中任何一个被调用的原因后ExpatError，将其code设置为的属性进行调用errors.codes[errors.XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING]。

xmlparser 对象具有以下属性：

xmlparser.buffer_size

所用缓冲区的大小buffer_text为true时。通过为该属性分配新的整数值可以设置新的缓冲区大小。当大小改变时，缓冲区将被刷新。

xmlparser.buffer_text

将其设置为true将导致xmlparser对象缓冲由Expat返回的文本内容，以避免多次调用CharacterDataHandler()回调函数。这可以大大提高性能，因为Expat通常会在每行结尾处将字符数据分块为块。此属性默认为false，可随时更改。

xmlparser.buffer_used

如果buffer_text启用，则存储在缓冲区中的字节数。这些字节表示UTF-8编码的文本。错误时该属性没有有意义的解释buffer_text。

xmlparser.ordered_attributes

将此属性设置为非零整数会导致将属性报告为列表而不是字典。属性以文档文本中的顺序显示。对于每个属性，都会显示两个列表条目：属性名称和属性值。（此模块的旧版本也使用此格式。）默认情况下，此属性为false; 它可能会随时更改。

xmlparser.specified_attributes

如果设置为非零整数，解析器将只报告在文档实例中指定的那些属性，而不报告从属性声明派生的属性。设置此应用程序时，需要特别小心地根据需要使用声明中提供的附加信息，以符合XML处理器行为标准。默认情况下，此属性为false; 它可能会随时更改。

以下属性包含与xmlparser对象遇到的最近错误相关的值，并且只有在调用Parse()或ParseFile()引发 xml.parsers.expat.ExpatError异常后才具有正确的值。

xmlparser.ErrorByteIndex
发生错误的字节索引。

xmlparser.ErrorCode
指定问题的数字代码。该值可以传递给 ErrorString() 函数，或者与errors对象中定义的其中一个常量进行比较 。

xmlparser.ErrorColumnNumber
发生错误的列号。

xmlparser.ErrorLineNumber
发生错误的行号。

以下属性包含与xmlparser对象中当前分析位置相关的值。在回调报告解析事件期间，它们指示生成事件的第一个字符序列的位置。当在回调之外调用时，指示的位置将刚过最后的分析事件（不管是否存在关联的回调）。

xmlparser.CurrentByteIndex
解析器输入中的当前字节索引。

xmlparser.CurrentColumnNumber
解析器输入中的当前列号。

xmlparser.CurrentLineNumber
解析器输入中的当前行号。

这是可以设置的处理程序列表。要在xmlparser对象*o*上设置处理程序 ，请使用。 *handlername* 必须从下面的列表中获取，*func*必须是一个可接受的对象，接受正确数量的参数。参数都是字符串，除非另有说明。 o.handlername = func

xmlparser.XmlDeclHandler (*版本* , *编码* , *独立*)
在分析XML声明时调用。XML声明是XML建议的适用版本的声明（可选），文档文本的编码以及可选的“独立”声明。 *版本*和*编码*将字符串和*独立*将是1如果文件被宣布独立，0如果宣布不必须是独立的，或者-1如果省略了独立的条款。这只适用于Expat版本1.95.0或更新的版本。

xmlparser.StartDoctypeDeclHandler (*doctypeName* , *systemId* , *publicId* , *has_internal_subset*)
当Expat开始解析文档类型声明（ ）时调用。该*doctypeName*正好提供所呈现。所述*的systenId*和 *publicId*如果指定，或参数给系统和公共标识符 ，如果删去。 如果文档包含内部文档声明子集， *has_internal_subset*将为true。 这需要Expat版本1.2或更新版本。 <!DOCTYPE ... None

xmlparser.EndDoctypeDeclHandler ( )
在Expat完成时解析文档类型声明时调用。这需要Expat版本1.2或更新版本。

xmlparser.ElementDeclHandler (*名称* , *型号*)
为每个元素类型声明调用一次。 *name*是元素类型的名称，*model*是内容模型的表示。

xmlparser.AttlistDeclHandler(*elname*, *attname*, *type*, *default*, *required*)

针对元素类型的每个声明的属性进行调用。如果一个属性列表声明声明了三个属性，那么这个处理器将被调用三次，每个属性一次。 *elname*是宣言适用于和元素的名称*attname*是声明的属性的名称。 属性类型是作为 *类型* 传递的字符串；可能的值是'CDATA'，'ID'，'IDREF'，... *默认*给出了使用的属性的默认值时，没有文档实例指定的属性，或者None如果没有默认值（#IMPLIED值）。如果该属性需要在文档实例中给出，则为 *必需*将是真实的。这需要Expat版本1.95.0或更新版本。

xmlparser.StartElementHandler(*名称*, *属性*)

调用每个元素的开始。 *name*是包含元素名称的字符串，*attributes*是元素属性。如果ordered_attributes属实，这是一个列表（请参阅 ordered_attributes完整说明）。否则，它是一个将名称映射到值的字典。

xmlparser.EndElementHandler(*名字*)

调用每个元素的结尾。

xmlparser.ProcessingInstructionHandler(*目标*, *数据*)

为每个处理指令调用。

xmlparser.CharacterDataHandler(*数据*)

调用字符数据。这将被称为正常字符数据，CDATA标记的内容和可忽略的空白。必须区分这些情况下，应用程序可以使用StartCdataSectionHandler， EndCdataSectionHandler以及ElementDeclHandler回调收集所需信息。

xmlparser.UnparsedEntityDeclHandler(*entityName*, *base*, *systemId*, *publicId*, *notationName*)

调用未解析的（NDATA）实体声明。这只适用于Expat库的1.2版本; 对于更新的版本，请EntityDeclHandler改用。（Expat库中的基本功能已被宣告已过时。）

xmlparser.EntityDeclHandler(*entityName*, *is_parameter_entity*, *value*, *base*, *systemId*, *publicId*, *notationName*)

请求所有实体声明。对于参数和内部实体， *值*将是一个给出实体声明内容的字符串; 这将None用于外部实体。该*notationName*参数将 None用于解析实体和符号的未解析实体的名称。 如果实体是参数实体，则 *is_parameter_entity* 将为true; 对于一般实体，*is_parameter_entity*将为true（大多数应用程序只需要关注一般实体）。这仅适用于Expat库的版本1.95.0。

xmlparser.NotationDeclHandler(*notationName*, *base*, *systemId*, *publicId*)

请求标记声明。 *notationName*，*base*和*systemId*以及 *publicId*是字符串（如果有的话）。如果公共标识符被省略， *publicId*将会是None。

xmlparser.StartNamespaceDeclHandler(*前缀*, *uri*)

当元素包含名称空间声明时调用。在StartElementHandler调用放置声明的元素之前处理名称空间声明。

xmlparser.EndNamespaceDeclHandler(*前缀*)

当包含名称空间声明的元素到达结束标记时调用。对于元素中的每个名称空间声明，调用该名称空间声明时会调用该名称空间声明， StartNamespaceDeclHandler以指示每个名称空间声明范围的开始。调用这个处理函数是EndElementHandler在元素结束的相应位置之后进行的。

xmlparser.CommentHandler（*数据*）

打电话征求意见。 *数据*是评论的文本，不包括前导'<!--'和尾随'-->'。

xmlparser.StartCdataSectionHandler（）

在CDATA部分开始时调用。这和EndCdataSectionHandler 需要能够识别CDATA部分的语法开始和结束。

xmlparser.EndCdataSectionHandler（）

在CDATA部分结束时调用。

xmlparser.DefaultHandler（*数据*）

调用XML文档中没有指定适用处理程序的任何字符。这意味着字符是可以被报告的构造的一部分，但是没有提供处理程序。

xmlparser.DefaultHandlerExpand（*数据*）

这DefaultHandler()与内部实体的扩展一样，但并不妨碍扩展。实体引用不会传递给默认处理程序。

xmlparser.NotStandaloneHandler（）

如果XML文档没有被声明为独立文档，则调用它。这发生在有外部子集或对参数实体的引用时，但XML声明没有在XML声明中独立设置yes。如果此处理程序返回0，则解析器将引发 XML_ERROR_NOT_STANDALONE错误。如果未设置此处理程序，解析器不会为此情况引发异常。

xmlparser.ExternalEntityRefHandler（*context*，*base*，*systemId*，*publicId*）

请求参考外部实体。 *基地*是目前的基地，由以前的呼叫设置SetBase()。公共和系统标识符 *systemId*和*publicId*是字符串，如果有的话; 如果没有给出公共标识符，*publicId*将会是None。的*上下文*值是不透明的，并且如下所述仅应使用。

对于要分析的外部实体，必须实现此处理程序。它负责创建子分析器，使用ExternalEntityParserCreate(context)适当的回调对其进行初始化，然后解析该实体。该处理程序应返回一个整数；如果它返回0，解析器将引发XML_ERROR_EXTERNAL_ENTITY_HANDLING错误，否则解析将继续。

如果未提供此处理程序，则通过DefaultHandler回调报告外部实体（如果提供）。

20.13.2。ExpatError异常

ExpatError 例外有许多有趣的属性：

ExpatError.code

Expat的特定错误的内部错误编号。该 errors.messages辞典这些错误号码映射到外籍人士的错误消息。例如：

```
from xml.parsers.expat import ParserCreate, ExpatError, errors

p = ParserCreate()
try:
    p.Parse(some_xml_document)
except ExpatError as err:
    print("Error:", errors.messages[err.code])
```

该errors模块还提供错误消息常量和codes将这些消息映射回错误代码的字典，请参见下文。

ExpatError.**lineno**

检测到错误的行号。第一行编号1。

ExpatError.**offset**

将字符偏移到发生错误的行中。第一列是编号0。

20.13.3。示例

下面的程序定义了三个处理程序，只是打印出他们的参数。

```
import xml.parsers.expat

# 3 handler functions
def start_element(name, attrs):
    print('Start element:', name, attrs)
def end_element(name):
    print('End element:', name)
def char_data(data):
    print('Character data:', repr(data))

p = xml.parsers.expat.ParserCreate()

p.StartElementHandler = start_element
p.EndElementHandler = end_element
p.CharacterDataHandler = char_data

p.Parse("""<?xml version="1.0"?>
<parent id="top"><child1 name="paul">Text goes here</child1>
<child2 name="fred">More text</child2>
</parent>""", 1)
```

这个程序的输出是：

```
Start element: parent {'id': 'top'}
Start element: child1 {'name': 'paul'}
Character data: 'Text goes here'
End element: child1
Character data: '\n'
Start element: child2 {'name': 'fred'}
```

```
Character data: 'More text'
End element: child2
Character data: '\n'
End element: parent
```

20.13.4。内容模型描述

内容模型使用嵌套元组进行描述。每个元组包含四个值：类型，量词，名称和孩子的元组。儿童仅仅是附加的内容模型描述。

前两个字段的值是在xml.parsers.expat.model模块中定义的常量 。这些常量可以分为两组：模型类型组和量词组。

模型类型组中的常量是：

xml.parsers.expat.model.XML_CTYPE_ANY
由模型名称命名的元素被声明为具有内容模型 ANY。

xml.parsers.expat.model.XML_CTYPE_CHOICE
指定的元素允许从多个选项中进行选择; 这用于内容模型，如。(A | B | C)

xml.parsers.expat.model.XML_CTYPE_EMPTY
声明为EMPTY具有此模型类型的元素。

xml.parsers.expat.model.XML_CTYPE_MIXED

xml.parsers.expat.model.XML_CTYPE_NAME

xml.parsers.expat.model.XML_CTYPE_SEQ
代表一系列相继模型的模型用这种模型类型表示。这用于模型，如。(A, B, C)

量词组中的常量是：

xml.parsers.expat.model.XML_CQUANT_NONE
没有给出修饰词，因此它可以只出现一次A。

xml.parsers.expat.model.XML_CQUANT_OPT
该模型是可选的：它可以出现一次或根本不出现A?。

xml.parsers.expat.model.XML_CQUANT_PLUS
模型必须发生一次或多次（如A+）。

xml.parsers.expat.model.XML_CQUANT_REP
该模型必须出现零次或多次，如A*。

20.13.5。Expat错误常量

xml.parsers.expat.errors 模块中提供了以下常量。这些常量在解释ExpatError发生错误时引发的异常对象的某些属性时非常有用。由于出于向后兼容性的原因，常量的值是错误*消息*而不是数字错误*代码*，因此您可以通过比较其 code 属性与值来执行此操作。errors.codes[errors.XML_ERROR_*CONSTANT_NAME*]

该errors模块具有以下属性：

xml.parsers.expat.errors.**codes**
将数字错误代码映射到其字符串描述的字典。

3.2版本中的新功能

xml.parsers.expat.errors.**messages**
字典将字符串描述映射到它们的错误代码。

3.2版本中的新功能

xml.parsers.expat.errors.**XML_ERROR_ASYNC_ENTITY**

xml.parsers.expat.errors.**XML_ERROR_ATTRIBUTE_EXTERNAL_ENTITY_REF**
属性值中的实体引用引用外部实体而不是内部实体。

xml.parsers.expat.errors.**XML_ERROR_BAD_CHAR_REF**
字符引用涉及XML中非法的字符（例如字符0或'&#0;'）。

xml.parsers.expat.errors.**XML_ERROR_BINARY_ENTITY_REF**
一个实体引用指的是一个用符号声明的实体，所以不能被解析。

xml.parsers.expat.errors.**XML_ERROR_DUPLICATE_ATTRIBUTE**
一个属性在开始标记中多次使用。

xml.parsers.expat.errors.**XML_ERROR_INCORRECT_ENCODING**

xml.parsers.expat.errors.**XML_ERROR_INVALID_TOKEN**
当输入字节无法正确分配给字符时引发;例如，0UTF-8输入流中的NUL字节（值）。

xml.parsers.expat.errors.**XML_ERROR_JUNK_AFTER_DOC_ELEMENT**
在文档元素之后发生了除空白外的其他内容。

xml.parsers.expat.errors.**XML_ERROR_MISPLACED_XML_PI**
XML声明被发现在输入数据的开始之外的某处。

xml.parsers.expat.errors.**XML_ERROR_NO_ELEMENTS**
该文档不包含任何元素（XML要求所有文档只包含一个顶层元素）..

xml.parsers.expat.errors.**XML_ERROR_NO_MEMORY**
Expat无法在内部分配内存。

xml.parsers.expat.errors.**XML_ERROR_PARAM_ENTITY_REF**

找不到参数实体引用。

xml.parsers.expat.errors.XML_ERROR_PARTIAL_CHAR
在输入中找到了不完整的字符。

xml.parsers.expat.errors.XML_ERROR_RECURSIVE_ENTITY_REF
实体参考包含对同一实体的另一个参考;可能通过不同的名称，并可能间接。

xml.parsers.expat.errors.XML_ERROR_SYNTAX
遇到了一些未指定的语法错误。

xml.parsers.expat.errors.XML_ERROR_TAG_MISMATCH
结束标记与最内层的开始标记不匹配。

xml.parsers.expat.errors.XML_ERROR_UNCLOSED_TOKEN
某些标记（例如开始标记）在流结束或遇到下一个标记之前未关闭。

xml.parsers.expat.errors.XML_ERROR_UNDEFINED_ENTITY
有人提到一个没有定义的实体。

xml.parsers.expat.errors.XML_ERROR_UNKNOWN_ENCODING
Expat不支持文档编码。

xml.parsers.expat.errors.XML_ERROR_UNCLOSED_CDATA_SECTION
CDATA标记部分未关闭。

xml.parsers.expat.errors.XML_ERROR_EXTERNAL_ENTITY_HANDLING

xml.parsers.expat.errors.XML_ERROR_NOT_STANDALONE
解析器确定文档不是“独立的”，虽然它声明自己在XML声明中，并且NotStandaloneHandler已经设置并返回0。

xml.parsers.expat.errors.XML_ERROR_UNEXPECTED_STATE

xml.parsers.expat.errors.XML_ERROR_ENTITY_DECLARED_IN_PE

xml.parsers.expat.errors.XML_ERROR_FEATURE_REQUIRES_XML_DTD
请求的操作需要编译DTD支持，但是Expat已配置为不支持DTD。这不应该由xml.parsers.expat模块的标准构建报告。

xml.parsers.expat.errors.XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING
解析开始后，请求行为更改，只能在分析开始之前对其进行更改。这是（目前）只能通过UseForeignDTD()。

xml.parsers.expat.errors.XML_ERROR_UNBOUND_PREFIX
在启用名称空间处理时找到未声明的前缀。

xml.parsers.expat.errors.XML_ERROR_UNDECLARING_PREFIX
该文档试图删除与前缀关联的名称空间声明。

xml.parsers.expat.errors.XML_ERROR_INCOMPLETE_PE
一个参数实体包含不完整的标记。

xml.parsers.expat.errors.XML_ERROR_XML_DECL
该文件完全没有文件元素。

xml.parsers.expat.errors.XML_ERROR_TEXT_DECL
解析外部实体中的文本声明时出错。

xml.parsers.expat.errors.XML_ERROR_PUBLICID
在公共ID中发现不允许的字符。

xml.parsers.expat.errors.XML_ERROR_SUSPENDED
请求的操作是在暂停的解析器上进行的，但不允许。这包括尝试提供额外的输入或停止解析器。

xml.parsers.expat.errors.XML_ERROR_NOT_SUSPENDED
解析器未被暂停时尝试恢复解析器。

xml.parsers.expat.errors.XML_ERROR_ABORTED
这不应该报告给Python应用程序。

xml.parsers.expat.errors.XML_ERROR_FINISHED
请求的操作是在解析输入完成的解析器上完成的，但是不允许。这包括尝试提供额外的输入或停止解析器。

xml.parsers.expat.errors.XML_ERROR_SUSPEND_PE

脚注

[1] 如XML输出中包含的编码字符串应符合相应的标准。例如，"UTF-8"是有效的，但是"UTF8"不是。参见https://www.w3.org/TR/2006/REC-xml-20060816/#NT-EncodingDecl和https://www.iana.org/assignments/character-sets/character-sets.xhtml。

21.互联网协议和支持

本章描述的模块实现互联网协议并支持相关技术。它们都是用Python实现的。这些模块中的大多数需要存在依赖于系统的模块`socket`，目前在大多数流行的平台上都支持该模块。这里是一个概述：

21.1。webbrowser- 方便的Web浏览器控制器

源代码： Lib / webbrowser.py

该webbrowser模块提供了一个高级界面，允许向用户显示基于Web的文档。在大多数情况下，只需open()从该模块调用该 函数就可以做正确的事情。

在Unix下，图形浏览器在X11下更受欢迎，但如果图形浏览器不可用或X11显示器不可用，则将使用文本模式浏览器。如果使用文本模式浏览器，则调用进程将阻塞，直到用户退出浏览器。

如果环境变量 BROWSER存在，它被解释为os.pathsep浏览器的 分离列表，以尝试超越平台默认值。当列表部分的值包含字符串时%s，它将被解释为一个文字浏览器命令行，用于替换参数URL %s; 如果该部分不包含%s，则将其简单解释为要启动的浏览器的名称。[1]

对于非Unix平台，或者当Unix上有远程浏览器时，控制过程不会等待用户完成浏览器，但允许远程浏览器在显示器上维护自己的窗口。如果远程浏览器在Unix上不可用，控制过程将启动一个新的浏览器并等待。

脚本**webbrowser**可以用作模块的命令行界面。它接受一个URL作为参数。它接受以下可选参数：-n如果可能，在新的浏览器窗口中打开URL; -t在新的浏览器页面（“标签”）中打开该URL。这些选项自然是相互排斥的。用法示例：

```
python -m webbrowser -t "http://www.python.org"
```

以下例外被定义：

*异常*webbrowser.Error
发生浏览器控制错误时引发异常。

定义了以下功能：

webbrowser.open（*url*，*new = 0*，*autoraise = True* ）
使用默认浏览器显示*网址*。如果*new*为0，则尽可能在相同的浏览器窗口中打开该*URL*。如果*新*值为1，则尽可能打开新的浏览器窗口。如果*新*值为2，则尽可能打开新的浏览器页面（“选项卡”）。如果*autoraise*是True，窗口如有可能提高（注意，在许多窗口管理器，这将无论发生此变量的设置）。

请注意，在某些平台上，尝试使用此功能打开文件名可能会起作用并启动操作系统的关联程序。但是，这既不支持也不便携。

webbrowser.open_new（*url* ）
打开*URL*在默认浏览器的一个新窗口，如果可能的话，否则，打开 *URL*中唯一的浏览器窗口。

webbrowser.open_new_tab（*url* ）
如果可能，在默认浏览器的新页面（“选项卡”）中打开*url*，否则相当于open_new()。

webbrowser.get（*使用=无*）

返回控制器对象为浏览器类型*使用*。如果*使用*IS None，返回控制器适合于呼叫者的环境中的默认浏览器。

webbrowser.register（*名称*，*构造函数*，*实例=无*）

注册浏览器类型*名称*。一旦浏览器类型被注册，该 get()功能可以返回该浏览器类型的控制器。如果*情况*不提供，或者是None，*构造函数*将被称为不带参数需要时创建一个实例。如果提供了*实例*，*构造函数*将永远不会被调用，并且可能是None。

这个入口点只有在你打算设置 BROWSER 变量或调用get()非空参数匹配您声明的处理程序的名称。

许多浏览器类型都是预定义的。该表给出了可以传递给get()函数的类型名称以及控制器类的相应实例，这些都是在本模块中定义的。

类型名称	班级名称	笔记
'mozilla'	Mozilla('mozilla')	
'firefox'	Mozilla('mozilla')	
'netscape'	Mozilla('netscape')	
'galeon'	Galeon('galeon')	
'epiphany'	Galeon('epiphany')	
'skipstone'	BackgroundBrowser('skipstone')	
'kfmclient'	Konqueror()	（1）
'konqueror'	Konqueror()	（1）
'kfm'	Konqueror()	（1）
'mosaic'	BackgroundBrowser('mosaic')	
'opera'	Opera()	
'grail'	Grail()	
'links'	GenericBrowser('links')	
'elinks'	Elinks('elinks')	
'lynx'	GenericBrowser('lynx')	
'w3m'	GenericBrowser('w3m')	
'windows-default'	WindowsDefault	（2）
'macosx'	MacOSX('default')	（3）
'safari'	MacOSX('safari')	（3）
'google-chrome'	Chrome('google-chrome')	
'chrome'	Chrome('chrome')	
'chromium'	Chromium('chromium')	
'chromium-browser'	Chromium('chromium-browser')	

笔记：

1. “Konqueror”是Unix的KDE桌面环境的文件管理器，只有在KDE运行时才有意义。可靠地检测KDE的一些方法会很好; 该KDEDIR变量是不够的。还要注意，即使在KDE 2中使用

konqueror命令时也使用名称“kfm” - 实施选择运行Konqueror的最佳策略。
2. 仅在Windows平台上。
3. 仅限于Mac OS X平台。

*3.3版新增功能：增加了*对Chrome / Chromium的支持。

这里有一些简单的例子：

```
url = 'http://docs.python.org/'

# Open URL in a new tab, if a browser window is already open.
webbrowser.open_new_tab(url)

# Open URL in new window, raising the window if possible.
webbrowser.open_new(url)
```

21.1.1。浏览器控制器对象

浏览器控制器提供了这些方法，它们可以并行执行三个模块级便利功能：

controller.**open**(*url*，*new = 0*，*autoraise = True*)
使用该控制器处理的浏览器显示*网址*。如果*新*值为1，则尽可能打开新的浏览器窗口。如果*新*值为2，则尽可能打开新的浏览器页面（“选项卡”）。

controller.**open_new**(*url*)
打开*网址*在此控制器，处理可能的话，浏览器的一个新窗口，否则，打开*URL*中唯一的浏览器窗口。别名 `open_new()`。

controller.**open_new_tab**(*url*)
如果可能，在由该控制器处理的浏览器的新页面（“选项卡”）中打开*url*，否则相当于 `open_new()`。

脚注

[1] 在没有完整路径的情况下命名的可执行文件将在中给出的目录中进行搜索 PATH 环境变量。

21.2。 cgi- 通用网关接口支持

源代码： Lib / cgi.py

通用网关接口（CGI）脚本的支持模块。

该模块定义了许多供Python编写的CGI脚本使用的实用程序。

21.2.1。简介

CGI脚本由HTTP服务器调用，通常用于处理通过HTML <FORM>或<ISINDEX>元素提交的用户输入。

CGI脚本通常位于服务器的特殊cgi-bin目录中。HTTP服务器在脚本的shell环境中放置有关请求的各种信息（例如客户端的主机名，请求的URL，查询字符串以及许多其他好东西），执行脚本并将脚本的输出发送回客户。

脚本的输入也连接到客户端，有时表单数据是这样读取的; 在其他时候，表单数据通过URL的“查询字符串”部分传递。本模块旨在处理不同情况，并为Python脚本提供更简单的接口。它还提供了许多帮助调试脚本的实用程序，最新的功能是支持从表单上传文件（如果浏览器支持的话）。

CGI脚本的输出应由两部分组成，用空行分隔。第一部分包含许多标题，告诉客户跟随什么样的数据。生成最小标题部分的Python代码如下所示：

```
print("Content-Type: text/html")    # HTML is following
print()                             # blank line, end of headers
```

第二部分通常是HTML，它允许客户端软件以头部，内嵌图像等方式显示格式良好的文本。下面是打印简单HTML代码的Python代码：

```
print("<TITLE>CGI script output</TITLE>")
print("<H1>This is my first CGI script</H1>")
print("Hello, world!")
```

21.2.2。使用cgi模块

从写作开始。 import cgi

当你写一个新的脚本时，考虑添加这些行：

```
import cgitb
cgitb.enable()
```

这将激活一个特殊的异常处理程序，如果发生任何错误，它将在Web浏览器中显示详细的报告。如果您不想将程序的内容显示给脚本的用户，您可以将这些报告保存到文件中，代码如

下：

```
import cgitb
cgitb.enable(display=0, logdir="/path/to/logdir")
```

在脚本开发过程中使用此功能非常有用。生成的报告cgitb提供的信息可以为您节省大量时间来追踪错误。您可以随时在cgitb测试脚本时删除该行，并确信它能正常工作。

要获取提交的表单数据，请使用FieldStorage该类。如果表单包含非ASCII字符，请使用*encoding*关键字参数设置为为文档定义的编码值。它通常包含在HTML文档的HEAD部分中的META标记中或 *Content-Type*标题中）。它从标准输入或环境读取表单内容（取决于根据CGI标准设置的各种环境变量的值）。由于它可能会消耗标准输入，因此应仅实例化一次。

该FieldStorage实例可以像Python字典一样编入索引。它允许in运营商进行成员资格测试，并支持标准字典方法keys()和内置函数 len()。包含空字符串的表单字段将被忽略，不会出现在字典中; 为了保留这些值，在创建 实例时为可选的*keep_blank_values*关键字参数提供一个真正的值 FieldStorage。

例如，下面的代码（其假定 *的Content-Type*首部和空行已经打印）检查该字段name和addr都设定为一个非空字符串：

```
form = cgi.FieldStorage()
if "name" not in form or "addr" not in form:
    print("<H1>Error</H1>")
    print("Please fill in the name and addr fields.")
    return
print("<p>name:", form["name"].value)
print("<p>addr:", form["addr"].value)
...further form processing here...
```

在这里，通过访问的字段form[key]本身就是实例 FieldStorage（或者MiniFieldStorage取决于表单编码）。 value实例的属性产生字段的字符串值。该getvalue()方法直接返回该字符串值; 它也接受一个可选的第二个参数作为默认返回，如果请求的键不存在。

如果提交的表单数据包含多个具有相同名称的字段，则检索到的对象form[key]不是一个FieldStorage或 MiniFieldStorage实例，而是这些实例的列表。同样，在这种情况下，form.getvalue(key)会返回一个字符串列表。如果您希望这种可能性（当您的HTML表单包含多个具有相同名称的字段时），请使用getlist()总是返回值列表的方法（这样您就不需要特别处理单个项目的情况）。例如，此代码连接任意数量的用逗号分隔的用户名字段：

```
value = form.getlist("username")
usernames = ",".join(value)
```

如果一个字段表示上传的文件，则通过value属性或getvalue() 方法访问该值 将以字节的形式读取内存中的整个文件。这可能不是你想要的。您可以通过测试filename属性或file 属性来测试上传的文件 。然后，您可以从file 属性读取数据，然后将其作为FieldStorage实例的垃圾回收的一部分自动关闭（ read() and readline()方法将返回字节）：

```
fileitem = form["userfile"]
if fileitem.file:
    # It's an uploaded file; count lines
```

```
    linecount = 0
    while True:
        line = fileitem.file.readline()
        if not line: break
        linecount = linecount + 1
```

FieldStorage对象还支持在with 声明中使用，它将在完成时自动关闭它们。

如果在获取上传文件的内容时遇到错误（例如，当用户通过单击“上一步”或“取消”按钮中断表单提交时）done，该字段的对象属性将设置为值-1。

文件上传草案标准考虑了从一个字段上传多个文件的可能性（使用递归*多部分/ *编码）。发生这种情况时，该项目将成为类似字典的FieldStorage项目。这可以通过测试它的type属性来确定，该属性应该是 *multipart / form-data*（或者可能是与*multipart / *匹配的另一个MIME类型）。在这种情况下，它可以像顶级窗体对象一样递归迭代。

当以“旧”格式（作为查询字符串或作为*application / x-www-form-urlencoded*类型的单个数据部分）提交*表单时*，这些项目实际上就是该类的实例MiniFieldStorage。在这种情况下， list ， file ，和filename属性始终None。

通过POST提交的也包含查询字符串的表单将包含两者 FieldStorage和MiniFieldStorage项目。

*在版本3.4中更改：*该file属性在创建FieldStorage实例的垃圾回收时自动关闭。

*在版本3.5中进行了更改：*增加了对FieldStorage类的上下文管理协议的支持 。

21.2.3。高级接口

上一节解释了如何使用FieldStorage该类读取CGI表单数据 。本节介绍了添加到该类中的更高级别的接口，以便以更易读和直观的方式进行操作。该接口不会使前面部分中描述的技术过时 - 例如，它们仍然可以有效处理文件上载。

界面由两个简单的方法组成。使用这些方法，您可以以通用的方式处理表单数据，而无需担心是否只有一个或多个值被张贴在一个名称下。

在前面的章节中，您已经学会在您希望用户使用同一个名称发布多个值时随时编写以下代码：

```
item = form.getvalue("item")
if isinstance(item, list):
    # The user is requesting more than one item.
else:
    # The user is requesting only one item.
```

例如，当表单包含一组具有相同名称的多个复选框时，这种情况很常见：

```
<input type="checkbox" name="item" value="1" />
<input type="checkbox" name="item" value="2" />
```

但是，在大多数情况下，表单中只有一个表单控件具有特定名称，因此您只需要一个与此名称关联的值。所以你写一个脚本包含例如这个代码：

```
user = form.getvalue("user").upper()
```

代码的问题是，你永远不应该期望客户端将为你的脚本提供有效的输入。例如，如果一个好奇的用户将另一个user=foo对添加到查询字符串中，那么脚本会崩溃，因为在这种情况下，getvalue("user")方法调用将返回一个列表而不是一个字符串。调用upper()列表上的方法无效（因为列表没有这个名称的方法）并导致 AttributeError异常。

因此，读取表单数据值的适当方式是始终使用代码来检查获取的值是单个值还是值列表。这很烦人，导致可读性较差的脚本。

更方便的方法是使用的方法getfirst() 和getlist()由该更高级别的接口提供的。

FieldStorage.**getfirst**(*名称*, *默认=无*)

此方法始终只返回与表单字段*名称*关联的一个值。该方法只会返回第一个值，以防在此名称下发布更多值。请注意，接收值的顺序可能因浏览器而异，因此不应计算在内。[1] 如果不存在这样的表单域或值，则该方法返回由可选参数*default*指定的*值*。None如果未指定，此参数默认为。

FieldStorage.**getlist**(*名字*)

此方法始终返回与表单字段*名称*关联的值列表。如果没有这样的表单字段或值存在对于该方法返回一个空列表 *名称*。如果只有一个这样的值存在，它将返回一个包含一个项目的列表。

使用这些方法，您可以编写精巧的代码：

```
import cgi
form = cgi.FieldStorage()
user = form.getfirst("user", "").upper()    # This way it's safe.
for item in form.getlist("item"):
    do_something(item)
```

21.2.4。函数

如果你想要更多的控制，或者如果你想在其他情况下使用本模块中实现的一些算法，这些都很有用。

cgi.**parse**(*fp = None*, *environ = os.environ*, *keep_blank_values = False*, *strict_parsing = False*)

在环境中或从文件解析查询（文件默认为 sys.stdin）。该*keep_blank_values*和*strict_parsing*参数传递给urllib.parse.parse_qs()不变。

cgi.**parse_qs**(*qs*, *keep_blank_values = False*, *strict_parsing = False*)

该功能在本模块中已弃用。urllib.parse.parse_qs() 改为使用。这里只是为了向后兼容而维护它。

cgi.**parse_qsl**(*qs*, *keep_blank_values = False*, *strict_parsing = False*)

该功能在本模块中已弃用。urllib.parse.parse_qsl() 改为使用。这里只是为了向后兼容而维护它。

cgi.parse_multipart（*fp*，*pdict*）

解析*multipart / form-data*类型的输入（用于文件上传）。参数是输入文件的*fp*，包含*Content-Type*头中其他参数的字典的*pdict*。

就像urllib.parse.parse_qs()键是字段名一样返回字典，每个值都是该字段的值列表。这很容易使用，但如果您预计会上传兆字节，则不会太好 - 在这种情况下，请使用FieldStorage类更灵活的类。

请注意，这不会分析嵌套的多部分 - FieldStorage用于此。

cgi.parse_header（*字符串*）

将MIME头（如*Content-Type*）解析为主值和参数字典。

cgi.test（）

强大的测试CGI脚本，可用作主程序。写入最小的HTTP标题并格式化以HTML形式提供给脚本的所有信息。

cgi.print_environ（）

用HTML格式化shell环境。

cgi.print_form（*form*）

用HTML格式化表单。

cgi.print_directory（）

在HTML中格式化当前目录。

cgi.print_environ_usage（）

在HTML中打印有用（由CGI使用）环境变量的列表。

cgi.escape（*s*，*quote = False*）

转换角色'&'，'<'并'>'在串*小号*到HTML安全序列。如果您需要显示可能包含HTML中的此类字符的文本，请使用此选项。如果可选的标志*引用*为真，则引号mark（"）也被翻译;这有助于包含在由双引号分隔的HTML属性值中，如in 。请注意，单引号永远不会翻译。<a href="...">

*自3.2版弃用：*此功能是不安全的，因为默认情况下，*quote*为false，因此不推荐使用。html.escape()改为使用。

21.2.5。关心安全

有一条重要的规则：如果你调用一个外部程序（通过 os.system()或os.popen()函数或其他具有类似功能的程序），请确保不要将从客户端收到的任意字符串传递给shell。这是一个众所周知的安全漏洞，Web上任何地方的聪明黑客都可以利用容易理解的CGI脚本来调用任意shell命令。即使部分URL或字段名称也不可信，因为请求不必来自您的表单！

为了安全起见，如果您必须将从窗体获得的字符串传递给shell命令，则应确保该字符串仅包含字母数字字符，破折号，下划线和句点。

21.2.6。在Unix系统上安装CGI脚本

阅读您的HTTP服务器的文档，并与您的本地系统管理员联系以查找应安装CGI脚本的目录; 通常这是在cgi-bin服务器树中的一个目录中。

确保你的脚本可以被“其他”读取和执行; Unix文件模式应该是0o755八进制（使用）。确保脚本的第一行包含从第1列开始，随后是Python解释器的路径名，例如：chmod 0755 filename#!

```
#!/usr/local/bin/python
```

确保Python解释器存在并可由“其他人”执行。

确保脚本需要读取或写入的任何文件可分别通过“其他”读取或写入 - 它们的模式应该是0o644可读和0o666可写的。这是因为，出于安全原因，HTTP服务器以“nobody”用户身份执行脚本，没有任何特殊权限。它只能读取（写入，执行）每个人都可以读取（写入，执行）的文件。执行时的当前目录也不同（通常是服务器的cgi-bin目录），而且这组环境变量也与您登录时得到的不同。特别是，不要指望shell的搜索路径对于可执行文件（ PATH ）或Python模块搜索路径（ PYTHONPATH ）被设置为任何有趣的东西。

如果您需要从不在Python默认模块搜索路径中的目录加载模块，则可以在导入其他模块之前更改脚本中的路径。例如：

```
import sys
sys.path.insert(0, "/usr/home/joe/lib/python")
sys.path.insert(0, "/usr/local/lib/python")
```

（这样，最后插入的目录将被首先搜索！）

非Unix系统的说明会有所不同; 检查你的HTTP服务器的文档（它通常会有一个关于CGI脚本的部分）。

21.2.7。测试你的CGI脚本

不幸的是，当从命令行尝试CGI脚本时，通常不会运行CGI脚本，而从命令行完美运行的脚本在从服务器运行时可能会神秘失败。还有一个原因是为什么你仍然应该从命令行测试脚本：如果它包含语法错误，Python解释器根本不会执行它，并且HTTP服务器很可能会向客户端发送一个神秘错误。

假设你的脚本没有语法错误，但它不起作用，你别无选择，只能阅读下一节。

21.2.8。调试CGI脚本

首先，检查一下安装错误 - 仔细阅读上面关于安装CGI脚本的部分可以节省很多时间。如果您想知道您是否正确理解了安装过程，请尝试安装此模块文件（ cgi.py ）的副本作为CGI脚本。当作为脚本调用时，该文件将以HTML形式转储其环境和表单的内容。给它正确的模式等，并发送一个请求。如果它安装在标准cgi-bin目录中，应该可以通过在表单的浏览器中输入URL来发送请求：

```
http://yourhostname/cgi-bin/cgi.py?name=Joe+Blow&addr=At+Home
```

如果这给404类型的错误，服务器找不到脚本 - 也许你需要将它安装在不同的目录中。如果它提供了另一个错误，那么在尝试继续之前应该修复一个安装问题。如果您获得了格式良好的环境和表单内容列表（在本例中，这些字段应该被列为值为“在家”和“名称”值为“Joe Blow”的“地址”），cgi.py脚本已经安装正确。如果您对自己的脚本遵循相同的过程，现在应该可以对其进行调试。

下一步可能是从脚本调用cgi模块的test()函数：用单个语句替换它的主代码

```
cgi.test()
```

这应该产生与安装cgi.py文件本身相同的结果 。

当一个普通的Python脚本引发一个未处理的异常（无论出于何种原因：模块名称中的拼写错误，无法打开的文件等）时，Python解释器会打印出一个很好的回溯并退出。虽然Python解释器在您的CGI脚本引发异常时仍然会执行此操作，但最有可能的是，回溯将最终放入其中一个HTTP服务器的日志文件中，或者完全丢弃。

幸运的是，一旦您设法让您的脚本执行*一些*代码，您就可以使用该cgitb模块轻松发送回溯至Web浏览器。如果你还没有这样做，只需添加行：

```
import cgitb
cgitb.enable()
```

到脚本的顶部。然后尝试再次运行它; 发生问题时，您应该看到详细的报告，可能会导致崩溃的原因。

如果您怀疑导入cgitb模块时可能存在问题，则可以使用更强大的方法（仅使用内置模块）：

```
import sys
sys.stderr = sys.stdout
print("Content-Type: text/plain")
print()
...your code here...
```

这依赖于Python解释器来打印回溯。输出内容类型设置为纯文本，禁用所有HTML处理。如果您的脚本正常工作，则原始HTML将由客户端显示。如果引发异常，很可能在打印出前两行后显示回溯。由于没有进行HTML解释，回溯将是可读的。

21.2.9。常见问题和解决方案

- 大多数HTTP服务器会缓存CGI脚本的输出，直到脚本完成。这意味着在脚本运行时不可能在客户端的显示器上显示进度报告。
- 检查上面的安装说明。
- 检查HTTP服务器的日志文件。（在单独的窗口中可能会有用！）`tail -f logfile`
- 首先通过做类似的事情来检查脚本是否有语法错误 。`python script.py`
- 如果您的脚本没有任何语法错误，请尝试添加到脚本的顶部。`import cgitb; cgitb.enable()`

- 调用外部程序时，确保可以找到它们。通常，这意味着使用绝对路径名 - PATH 通常不会在CGI脚本中设置为非常有用的值。
- 在读取或编写外部文件时，请确保可以在运行CGI脚本的用户标识下读取或写入它们：这通常是运行Web服务器的用户标识，或者是Web服务器suexec功能的一些明确指定的用户标识。
- 不要试图给一个CGI脚本一个set-uid模式。这在大多数系统上不起作用，并且也是安全责任。

脚注

[1] 虽然HTTP规范建议浏览器发送请求的顺序，但知道从合格的浏览

21.3。 cgitb- CGI脚本的跟踪管理器

源代码： Lib / cgitb.py

该cgitb模块为Python脚本提供了一个特殊的异常处理程序。（它的名字有点令人误解，它最初设计为在CGI脚本中显示HTML的广泛跟踪信息，后来被普遍化为以纯文本形式显示这些信息）。激活此模块后，如果发生未捕获的异常，将显示详细的格式化报告。该报告包含一个回溯，显示每个级别的源代码片段，以及当前运行的函数的参数值和局部变量值，以帮助您调试问题。或者，您可以将此信息保存到文件中，而不是将其发送到浏览器。

要启用此功能，只需将其添加到CGI脚本的顶部：

```
import cgitb
cgitb.enable()
```

enable()函数的选项控制报表是否显示在浏览器中，以及报表是否记录到文件中供以后分析。

cgitb.enable（*display = 1*，*logdir = None*，*context = 5*，*format =“html”*）

该函数cgitb通过设置值来使模块接管解释器对异常的默认处理sys.excepthook。

可选参数*显示*默认为1并且可以设置0为禁止将回溯发送到浏览器。如果参数*logdir*存在，则追溯报告将写入文件。*logdir*的值 应该是放置这些文件的目录。可选的参数 *上下文*是在回溯中的当前源代码行周围显示的上下文的行数; 这默认为5。如果可选参数*格式*为"html"，则输出格式为HTML。任何其他值强制纯文本输出。默认值是"html"。

cgitb.handler（*info =无*）

此函数使用默认设置处理异常（即，在浏览器中显示报告，但不记录到文件）。这可以在您发现异常并希望使用报告时使用cgitb。可选的 *info*参数应该是包含异常类型，异常值和跟踪对象的3元组，就像返回的元组一样 sys.exc_info()。如果未提供*info*参数，则从中获取当前异常sys.exc_info()。

21.4。wsgiref- WSGI实用程序和参考实现

Web服务器网关接口（WSGI）是Web服务器软件和用Python编写的Web应用程序之间的标准接口。拥有标准接口可以轻松使用支持WSGI和多个不同Web服务器的应用程序。

只有Web服务器和编程框架的作者需要知道WSGI设计的每个细节和角落案例。您不需要了解WSGI的每个细节，只需安装WSGI应用程序或使用现有框架编写Web应用程序即可。

wsgiref是WSGI规范的参考实现，可用于将WSGI支持添加到Web服务器或框架。它提供了用于操纵WSGI环境变量和响应头的实用程序，用于实现WSGI服务器的基类，为WSGI应用程序提供服务的演示HTTP服务器，以及用于检查WSGI服务器和应用程序是否符合WSGI规范的验证工具**PEP 3333**）。

有关WSGI的更多信息，请参阅https://wsgi.readthedocs.org/以及指向教程和其他资源的链接。

21.4.1。wsgiref.util- WSGI环境实用程序

该模块提供了多种用于WSGI环境的实用程序功能。WSGI环境是一个包含HTTP请求变量的字典，如下所述**PEP 3333**。所有采用 *environ* 参数的函数都需要提供符合WSGI的字典; 请参见**PEP 3333**的详细规范。

wsgiref.util.**guess_scheme**（*environ*）

wsgi.url_scheme通过检查*environ*字典中的HTTPS环境变量来返回是否应该是“http”或“https”的猜测。返回值是一个字符串。

创建包装CGI的网关或类似CGI的协议（如FastCGI）时，此功能很有用。通常，HTTPS当通过SSL接收请求时，提供此类协议的服务器将包含一个值为“1”“是”或“开” 的变量。所以，如果找到这样的值，这个函数返回“https”，否则返回“http”。

wsgiref.util.**request_uri**（*environ*，*include_query* = *True*）

使用在“URL Reconstruction”部分找到的算法返回完整的请求URI，可选地包括查询字符串**PEP 3333**。如果 *include_query*为false，则查询字符串不包含在生成的URI中。

wsgiref.util.**application_uri**（*environ*）

类似request_uri()，除了PATH_INFO和 QUERY_STRING变量被忽略。结果是请求所处理的应用程序对象的基本URI。

wsgiref.util.**shift_path_info**（*environ*）

将单个名称从中PATH_INFO切换SCRIPT_NAME并返回名称。该*ENVIRON字典改性*就地; 如果您需要保留原件PATH_INFO或SCRIPT_NAME完好无损，请使用副本。

如果没有剩余路径段PATH_INFO，None则返回。

通常，此例程用于处理请求URI路径的每个部分，例如将路径视为一系列字典键。此例程修改传入的环境以使其适用于调用位于目标URI处的另一个WSGI应用程序。例如，如果WSGI应用程序位于/foo，请求URI路径为/foo/bar/baz，并且WSGI应用程序处于/foo调

用状态shift_path_info()，则它将收到字符串“bar”，并且环境将更新为适合传递给WSGI应用程序/foo/bar。也就是说，SCRIPT_NAME将会从 /foo变为/foo/bar ，并将PATH_INFO从/bar/baz变为 /baz。

当PATH_INFO只是一个“/”时，这个例程返回一个空字符串并附加一个尾部斜线SCRIPT_NAME，即使通常忽略空路段，并且SCRIPT_NAME通常不以斜线结尾。这是有意的行为，以确保应用程序可以区分以使用此例程进行对象遍历时/x结束的URI之间的差异/x/。

wsgiref.util.**setup_testing_defaults**(*environ*)

更新*ENVIRON*琐碎的默认设置进行测试。

这个程序增加了对WSGI所需的各种参数，包括HTTP_HOST， SERVER_NAME，SERVER_PORT，REQUEST_METHOD，SCRIPT_NAME， PATH_INFO，和所有的**PEP 3333**-定义的wsgi.*变量。它只提供默认值，并不会替换这些变量的任何现有设置。

此例程旨在使WSGI服务器和应用程序的单元测试更容易设置虚拟环境。它不应该被实际的WSGI服务器或应用程序使用，因为数据是假的！

用法示例：

```
from wsgiref.util import setup_testing_defaults
from wsgiref.simple_server import make_server

# A relatively simple WSGI application. It's going to print out the
# environment dictionary after being updated by setup_testing_defaults
def simple_app(environ, start_response):
    setup_testing_defaults(environ)

    status = '200 OK'
    headers = [('Content-type', 'text/plain; charset=utf-8')]

    start_response(status, headers)

    ret = [("%s: %s\n" % (key, value)).encode("utf-8")
           for key, value in environ.items()]
    return ret

with make_server('', 8000, simple_app) as httpd:
    print("Serving on port 8000...")
    httpd.serve_forever()
```

除上述环境功能外，该wsgiref.util模块还提供了这些杂项实用程序：

wsgiref.util.**is_hop_by_hop**(*header_name*)

如果'header_name'是一个HTTP / 1.1“逐跳”标头，则返回true，如定义 **RFC 2616**。

*类*wsgiref.util.**FileWrapper**(*类文件*, *BLKSIZE = 8192*)

将文件类对象转换为迭代器的包装器。所得到的对象支持__getitem__()和__iter__()迭代的风格，对于与Python 2.1和Jython兼容。随着对象被迭代，可选*blksize*参数将被重复传递给*类文件*对象的read()方法以获取要产生的字节*串*。当read() 返回一个空字节串时，迭代结束并且不可恢复。

如果*filelike*有一个close()方法，返回的对象也会有一个 close()方法，并且在调用时会调用*类文件*对象的close() 方法。

用法示例：

```
from io import StringIO
from wsgiref.util import FileWrapper

# We're using a StringIO-buffer for as the file-like object
filelike = StringIO("This is an example file-like object"*10)
wrapper = FileWrapper(filelike, blksize=5)

for chunk in wrapper:
    print(chunk)
```

21.4.2。 wsgiref.headers- WSGI响应头文件

这个模块提供了一个类，Headers用于使用类似映射的接口方便地处理WSGI响应头。

class wsgiref.headers.Headers（[*headers*] ）

创建一个类似映射的对象封装*标头*，它必须是标头名称/值元组列表，如描述**PEP 3333**。*标题*的默认值是一个空列表。

Headers对象支持典型映射操作，包括 __getitem__() ， get() ， __setitem__() ， setdefault() ， __delitem__()和__contains__()。对于这些方法中的每一个，关键是标头名称（不区分大小写处理），并且该值是与该标头名称关联的第一个值。设置标题将删除该标题的所有现有值，然后在包装标题列表的末尾添加一个新值。通常会保留标题的现有订单，并将新标题添加到包装列表的末尾。

与字典不同，Headers当您尝试获取或删除不在包装标题列表中的密钥时，对象不会引发错误。获取不存在的头只是返回None，并且删除不存在的头不会执行任何操作。

Headers对象也支持keys() ， values()和 items()方法。该名单由归国keys() ，并items()可以包括相同的密钥不止一次，如果有一个多值的头。的 len()一个的Headers目的是相同的其长度 items() ，这是一样的缠绕头列表的长度。实际上，该items()方法只是返回包装标题列表的副本。

调用bytes()一个Headers对象返回一个格式化的字节串，适合作为HTTP响应头传输。每个标题都放在一行上，并用冒号和空格分隔。每行都以回车符和换行符结尾，字符串以空行结束。

除了映射接口和格式化功能外，Headers 对象还具有以下用于查询和添加多值标题的方法，以及用于添加具有MIME参数的标题的方法：

get_all（*名字*）

返回指定标题的所有值的列表。

返回的列表将按照它们出现在原始标题列表中的顺序排序，或者添加到此实例中，并可能包含重复项。任何删除并重新插入的字段都会附加到标题列表中。如果给定名称不存在字段，则返回空列表。

add_header（*名称*，*值*，**_params*）

添加一个（可能是多值的）头，其中包含通过关键字参数指定的可选MIME参数。

*名称*是要添加的标题字段。关键字参数可用于为标题字段设置MIME参数。每个参数必须是字符串或None。参数名称中的下划线将转换为破折号，因为破折号在Python标识符中是非法的，但许多MIME参数名称都包含破折号。如果参数值是一个字符串，则将其添加到表单中的标题值参数中name="value"。如果是None，则只添加参数名称。（这用于没有值的MIME参数。）示例用法：

```
h.add_header('content-disposition', 'attachment', filename='bud.gif')
```

以上将添加一个如下所示的标题：

```
Content-Disposition: attachment; filename="bud.gif"
```

*在版本3.5中更改：*headers参数是可选的。

21.4.3。wsgiref.simple_server- 一个简单的WSGI HTTP服务器

这个模块实现了一个简单的HTTP服务器（基于http.server）为WSGI应用程序提供服务。每个服务器实例在给定主机和端口上提供单个WSGI应用程序。如果您想在单个主机和端口上提供多个应用程序，则应该创建一个WSGI应用程序，PATH_INFO该应用程序可以解析以选择为每个请求调用哪个应用程序。（例如，使用shift_path_info()来自的功能 wsgiref.util。）

wsgiref.simple_server.make_server（*主机*，*端口*，*应用程序*，*server_class = WSGIServer*，*handler_class = WSGIRequestHandler*）

创建一个监听*主机*和*端口*的新WSGI服务器，接受*应用程序*的连接。返回值是提供的*server_class*的一个实例，并将使用指定的*handler_class*处理请求。*应用程序*必须是WSGI应用程序对象，如所定义的**PEP 3333**。

用法示例：

```
from wsgiref.simple_server import make_server, demo_app

with make_server('', 8000, demo_app) as httpd:
    print("Serving HTTP on port 8000...")

    # Respond to requests until process is killed
    httpd.serve_forever()

    # Alternative: serve one request, then exit
    httpd.handle_request()
```

wsgiref.simple_server.demo_app（*environ*，*start_response*）

该函数是一个小而完整的WSGI应用程序，它返回一个包含消息“Hello world！”的文本页面和一个在*environ*参数中提供的键/值对列表。这对验证WSGI服务器（如wsgiref.simple_server）是否能够正确运行简单的WSGI应用程序很有用。

*类*wsgiref.simple_server.WSGIServer（*server_address*，*RequestHandlerClass*）

创建一个WSGIServer实例。 *server_address*应该是一个（host,port）元组，并且*RequestHandlerClass*应该http.server.BaseHTTPRequestHandler是用于处理请求的子类。

您通常不需要调用此构造函数，因为该make_server()函数可以处理您的所有细节。

WSGIServer是它的一个子类http.server.HTTPServer，所以它的所有方法（如serve_forever()和handle_request()）都可用。WSGIServer还提供了这些WSGI特定的方法：

set_app（*应用程序*）

将可调用*应用程序*设置为将接收请求的WSGI应用程序。

get_app（）

返回当前设置的可调用应用程序。

但是，通常情况下，您不需要像set_app()通常所称的那样使用这些附加方法make_server()，并且get_app()主要是为了请求处理程序实例的存在。

class wsgiref.simple_server.WSGIRequestHandler（*request*，*client_address*，*server*）

为给定*请求*（即套接字），*client_address*（一个（host,port）元组）和*server*（WSGIServer实例）创建一个HTTP处理程序。

你不需要直接创建这个类的实例。它们会根据WSGIServer对象的需要自动创建。但是，您可以继承这个类并将其作为*handler_class提供*给make_server()函数。一些可能相关的用于覆盖子类的方法：

get_environ（）

返回包含请求的WSGI环境的字典。默认实现复制WSGIServer对象的base_environdictionary属性的内容，然后添加从HTTP请求派生的各种头文件。对这个方法的每次调用都应该返回一个新的字典，其中包含所有相关的CGI环境变量 **PEP 3333**。

get_stderr（）

返回应该用作wsgi.errors流的对象。默认实现只是返回sys.stderr。

handle（）

处理HTTP请求。默认实现使用wsgiref.handlers类创建一个处理程序实例来实现实际的WSGI应用程序接口。

21.4.4。wsgiref.validate- WSGI一致性检查器

在创建新的WSGI应用程序对象，框架，服务器或中间件时，使用新的代码一致性验证可能很有用wsgiref.validate。该模块提供了一个函数，用于创建验证WSGI服务器或网关与WSGI应用程序对象之间通信的WSGI应用程序对象，以检查双方是否符合协议。

请注意，此实用程序不保证完整 符合 **PEP 3333**；这个模块没有错误并不一定意味着错误不存在。但是，如果此模块确实产生错误，那么几乎可以确定服务器或应用程序不是100％兼容的。

该模块基于paste.lintIan Bicking的“Python Paste”库中的模块。

wsgiref.validate.validator（*应用程序*）

包装*应用程序*并返回一个新的WSGI应用程序对象。返回的应用程序会将所有请求转发给原始*应用程序*，并将检查*应用程序*和调用它的服务器是否符合WSGI规范和RFC 2616。

任何检测到的不合格都会导致产生AssertionError；但请注意，如何处理这些错误取决于服务器。例如，wsgiref.simple_server基于其他服务器的服务器 wsgiref.handlers（不会覆盖错误处理方法来执行其他操作）只会输出错误已发生的消息，并将追溯转储到sys.stderr其他某个错误流。

该包装也可以使用该warnings模块生成输出以指示有问题但可能实际上不会被禁止的行为**PEP 3333**。除非它们被使用Python命令行选项或抑制warningsAPI，任何这样的警告将被写入 sys.stderr（*未* wsgi.errors，除非它们碰巧是相同的对象）。

用法示例：

```
from wsgiref.validate import validator
from wsgiref.simple_server import make_server

# Our callable object which is intentionally not compliant to the
# standard, so the validator is going to break
def simple_app(environ, start_response):
    status = '200 OK'  # HTTP Status
    headers = [('Content-type', 'text/plain')]  # HTTP Headers
    start_response(status, headers)

    # This is going to break because we need to return a list, and
    # the validator is going to inform us
    return b"Hello World"

# This is the application wrapped in a validator
validator_app = validator(simple_app)

with make_server('', 8000, validator_app) as httpd:
    print("Listening on port 8000....")
    httpd.serve_forever()
```

21.4.5。wsgiref.handlers-服务器/网关基础类

该模块为实现WSGI服务器和网关提供基本处理程序类。这些基类处理与WSGI应用程序通信的大部分工作，只要它们具有类CGI环境以及输入，输出和错误流。

*类*wsgiref.handlers.CGIHandler

通过基于CGI的调用sys.stdin，sys.stdout，sys.stderr和 os.environ。当你有一个WSGI应用程序并且想把它作为一个CGI脚本运行时，这很有用。简单地调用CGIHandler().run(app)，app你希望调用的WSGI应用程序对象在哪里。

这个类是它的一个子类，BaseCGIHandler它设置wsgi.run_once为true，wsgi.multithreadfalse和wsgi.multiprocesstrue，并且总是使用sys并os获得必要的CGI流和环境。

*类*wsgiref.handlers.IISCGIHandler

CGIHandler未经设置配置allowPathInfo选项（IIS> = 7）或配置数据库allowPathInfoForScriptMappings（IIS <7）时，用于在Microsoft IIS Web服务器上部署时使用的特殊替代方案。

默认情况下，IIS提供了一个PATH_INFO是复制了SCRIPT_NAME在前面，给这一希望实现路由WSGI应用程序的问题。这个处理程序去除任何这样的重复路径。

IIS可以被配置为传递正确PATH_INFO，但是这会导致另一个PATH_TRANSLATED错误。幸运的是，这个变量很少被使用，并且不被WSGI保证。但是，在IIS <7上，只能在虚拟主机级别进行设置，影响所有其他脚本映射，其中许多脚本映射在暴露给该PATH_TRANSLATED错误时会中断。由于这个原因，IIS <7几乎从未部署修复程序。（甚至IIS7也很少使用它，因为它仍然没有UI。）

CGI代码无法判断是否设置了该选项，因此提供了单独的处理程序类。它以与CGIHandler调用相同的方式使用，您希望调用IISCGIHandler().run(app)哪个 appWSGI应用程序对象。

3.2版本中的新功能

class wsgiref.handlers.BaseCGIHandler（*stdin*，*stdout*，*stderr*，*environ*，*multithread* = *True*，*multiprocess* = *False*）

与此类似CGIHandler，但不是使用sys和 os模块，而是显式指定CGI环境和I / O流。在*多线程*和*多进程*值被用来设置 wsgi.multithread和wsgi.multiprocess标志由处理实例中运行的所有应用程序。

这个类是SimpleHandler打算与HTTP“原始服务器”以外的软件一起使用的子类。如果您正在编写使用Status:头来发送HTTP状态的网关协议实现（例如CGI，FastCGI，SCGI等），那么您可能希望将其替换为子类SimpleHandler。

class wsgiref.handlers.SimpleHandler（*stdin*，*stdout*，*stderr*，*environ*，*multithread* = *True*，*multiprocess* = *False*）

BaseCGIHandler与HTTP源服务器类似，但设计用于HTTP源服务器。如果你正在编写一个HTTP服务器实现，你可能想要继承它而不是BaseCGIHandler。

这个类是一个子类BaseHandler。它覆盖了 __init__()，get_stdin()，get_stderr()，add_cgi_vars()，_write()，和_flush()方法以支持明确设置环境，通过构造流。所提供的环境和流被存储在stdin，stdout，stderr，和 environ属性。

*标准输出*的write()方法应该完整地写入每个块，就像。 io.BufferedIOBase

*类*wsgiref.handlers.BaseHandler

这是运行WSGI应用程序的抽象基类。每个实例将处理单个HTTP请求，但原则上，您可以创建一个可用于多个请求的子类。

BaseHandler 实例只有一种用于外部使用的方法：

run (*app*)
运行指定的WSGI应用程序*app*。

所有其他BaseHandler方法在运行应用程序的过程中都由此方法调用，因此主要用于定制过程。

必须在子类中覆盖以下方法：

_write (*数据*)
缓冲字节*数据*以传输给客户端。如果这种方法实际传输数据，这没关系; BaseHandler只是在底层系统实际上具有这种区别时，才将写入和刷新操作分开以提高效率。

_flush ( )
强制缓冲数据传输到客户端。如果这种方法是无操作的（即，如果_write()实际发送数据），那没关系。

get_stdin ( )
返回适合用作wsgi.input当前正在处理的请求的输入流对象。

get_stderr ( )
返回适合用作wsgi.errors当前正在处理的请求的输出流对象。

add_cgi_vars ( )
将当前请求的CGI变量插入到environ属性中。

以下是您可能希望覆盖的一些其他方法和属性。然而，这个列表只是一个总结，并不包括可以被覆盖的每个方法。在尝试创建自定义BaseHandler 子类之前，您应该查阅文档字符串和源代码以获取更多信息。

定制WSGI环境的属性和方法：

wsgi_multithread
要用于wsgi.multithread环境变量的值。它默认为true BaseHandler，但在其他子类中可能有不同的默认值（或由构造函数设置）。

wsgi_multiprocess
要用于wsgi.multiprocess环境变量的值。它默认为true BaseHandler，但在其他子类中可能有不同的默认值（或由构造函数设置）。

wsgi_run_once
要用于wsgi.run_once环境变量的值。它默认为false BaseHandler，但CGIHandler默认设置为true。

os_environ
要包含在每个请求的WSGI环境中的默认环境变量。默认情况下，这是导入os.environ时 的副本wsgiref.handlers，但子类可以在类或实例级别创建它们自己的副本。请注意，字典应该被认为是只读的，因为默认值是在多个类和实例之间共享的。

server_software

如果origin_server设置了该属性，则使用此属性的值来设置默认的SERVER_SOFTWARE WSGI环境变量，并且还可以Server:在HTTP响应中设置默认标题。对于不是HTTP原始服务器的处理程序（例如BaseCGIHandler和CGIHandler），它将被忽略。

*版本3.3中更改：*术语“Python”替换为实现特定的术语，如“CPython”，“Jython”等。

get_scheme()

返回用于当前请求的URL方案。默认实现使用guess_scheme()函数从wsgiref.util当前请求的environ变量中猜测该方案是“http”还是“https”。

setup_environ()

将该environ属性设置为完全填充的WSGI环境。默认实现使用上述所有方法和属性，再加上get_stdin()，get_stderr()以及add_cgi_vars()方法和wsgi_file_wrapper属性。SERVER_SOFTWARE如果不存在，它也会插入一个密钥，只要该origin_server属性是一个真实值并且该server_software属性已设置。

用于自定义异常处理的方法和属性：

log_exception(*exc_info*)

将*exc_info*元组记录到服务器日志中。*exc_info*是一个元组。默认实现简单地将回溯写入请求的流并刷新它。子类可以重写此方法来更改格式或重定目标输出，将回溯邮件发送给管理员，或者其他任何可能被认为合适的操作。(type, value, traceback) wsgi.errors

traceback_limit

缺省log_exception()方法输出的回溯中包含的最大帧数。如果None包含所有框架。

error_output(*environ*, *start_response*)

此方法是一个WSGI应用程序，为用户生成错误页面。只有在标题发送到客户端之前发生错误时才会调用它。

此方法可以使用当前的错误信息sys.exc_info()，并在调用时将该信息传递给*start_response*（如“错误处理”一节中所述**PEP 3333**）。

默认的实现只是使用的error_status，error_headers和error_body属性生成输出页面。子类可以覆盖它以产生更多的动态错误输出。

但是，请注意，从安全角度不建议将诊断信息吐出给任何旧用户; 理想情况下，您应该做一些特殊的事情来启用诊断输出，这就是为什么默认实现不包含任何内容。

error_status

用于错误响应的HTTP状态。这应该是一个状态字符串，如下定义**PEP 3333**；它默认为500代码和消息。

error_headers

用于错误响应的HTTP标头。这应该是WSGI响应头（元组）的列表，如下所述(name, value) **PEP 3333**。默认列表只是将内容类型设置为text/plain。

error_body
: 错误响应正文。这应该是一个HTTP响应主体字节串。它默认为纯文本，“发生服务器错误。请联系管理员。“

方法和属性 **PEP 3333**的“可选平台特定文件处理”功能：

wsgi_file_wrapper
: 一个wsgi.file_wrapper工厂，或None。该属性的默认值是wsgiref.util.FileWrapper该类。

sendfile ( )
: 重写以实现平台特定的文件传输。只有当应用程序的返回值是由wsgi_file_wrapper属性指定的类的实例时才调用此方法。如果它能够成功传输文件，它应该返回一个真值，以便默认的传输代码不会被执行。此方法的默认实现仅返回一个假值。

其他方法和属性：

origin_server
: 如果处理程序的该属性应该设置为一个真正的价值_write()和 _flush()被用来直接传达给客户，而不是通过想要在一个特殊的HTTP状态一类CGI网关协议 Status:报头。

  该属性的默认值为true BaseHandler，但在BaseCGIHandler和中为false CGIHandler。

http_version
: 如果origin_server为true，则使用此字符串属性来设置为客户端设置的响应的HTTP版本。它默认为"1.0"。

wsgiref.handlers. read_environ ( )
: 将CGI变量转换os.environ为PEP 3333“unicode”字节中的字节，并返回一个新的字典。该功能使用 CGIHandler，并IISCGIHandler在地方直接使用的 os.environ，这不一定是WSGI兼容上使用Python 3所有平台和Web服务器-尤其是那些在OS的实际环境为Unicode（如Windows），或那些那里的环境字节，但Python用来解码它的系统编码不是ISO-8859-1（例如使用UTF-8的Unix系统）。

  如果您正在实现您自己的基于CGI的处理程序，则可能需要使用此例程，而不是直接复制值os.environ 。

  3.2版本中的新功能

21.4.6。示例

这是一个工作的“Hello World”WSGI应用程序：

```
from wsgiref.simple_server import make_server

# Every WSGI application must have an application object - a callable
# object that accepts two arguments. For that purpose, we're going to
# use a function (note that you're not limited to a function, you can
# use a class for example). The first argument passed to the function
```

```
# is a dictionary containing CGI-style environment variables and the
# second variable is the callable object (see PEP 333).
def hello_world_app(environ, start_response):
    status = '200 OK'  # HTTP Status
    headers = [('Content-type', 'text/plain; charset=utf-8')]  # HTTP Headers
    start_response(status, headers)

    # The returned object is going to be printed
    return [b"Hello World"]

with make_server('', 8000, hello_world_app) as httpd:
    print("Serving on port 8000...")

    # Serve until process is killed
    httpd.serve_forever()
```

21.5。 urllib- URL处理模块

源代码： Lib / urllib /

urllib 是一个收集多个模块以处理URL的软件包：

- urllib.request 用于打开和阅读网址
- urllib.error 包含由...提出的例外 urllib.request
- urllib.parse 用于解析URL
- urllib.robotparser用于解析robots.txt文件

21.6。urllib.request- 用于打开URL的可扩展库

源代码： Lib / urllib / request.py

该urllib.request模块定义了帮助在复杂世界中打开URL（主要是HTTP）的函数和类 - 基本和摘要式身份验证，重定向，cookie等等。

也可以看看： 该请求包 被推荐用于更高级别的HTTP客户端接口。

该urllib.request模块定义了以下功能：

urllib.request.urlopen（*url*，*data* = *None*，[*timeout*，] *，*cafile* = *None*，*capath* = *None*，*cadefault* = *False*，*context* = *None* ）

打开URL *url*，它可以是一个字符串，也可以是一个 Request对象。

*数据*必须是指定要发送到服务器的附加数据的对象，或者None如果不需要这样的数据。详情请参阅Request 。

urllib.request模块使用HTTP / 1.1并Connection:close在其HTTP请求中包含标头。

可选的*timeout*参数指定阻止诸如连接尝试（如果未指定，将使用全局默认超时设置）等操作的超时（以秒为单位）。这实际上只适用于HTTP，HTTPS和FTP连接。

如果指定了*上下文*，则它必须是 ssl.SSLContext 描述各种SSL选项的实例。查看 HTTPSConnection 更多细节。

可选的*cafile*和*capath*参数为HTTPS请求指定一组可信的CA证书。 *cafile*应指向包含一系列CA证书的单个文件，而*capath*应指向散列证书文件的目录。更多信息可以在中找到 ssl.SSLContext.load_verify_locations()。

该*cadefault*参数被忽略。

这个函数总是返回一个可以作为上下文管理器的对象， 并且具有类似的方法

- geturl() - 返回检索的资源的URL，通常用于确定是否遵循重定向
- info()- 以email.message_from_string()实例的形式返回页面的元信息（如标题）（请参阅 快速参考HTTP标题）
- getcode() - 返回响应的HTTP状态码。

对于HTTP和HTTPS URL，此函数返回http.client.HTTPResponse稍微修改的 对象。除上述三种新方法之外，msg属性还包含与属性相同的信息reason - 服务器返回的原因短语 - 而不是文档中指定的响应标头 HTTPResponse。

对于传统URLopener和FancyURLopener类明确处理的FTP，文件和数据URL和请求 ，此函数返回一个urllib.response.addinfourl对象。

引发URLError协议错误。

请注意，None如果没有处理程序处理请求，可能会返回（尽管默认安装的全局OpenerDirector使用 UnknownHandler来确保永远不会发生这种情况）。

另外，如果检测到代理设置（例如，当一个*_proxy 环境变量像http_proxy已设置），ProxyHandler默认安装并确保通过代理处理请求。

urllib.urlopenPython 2.6及更早版本的遗留功能已停止使用; urllib.request.urlopen()对应于旧的 urllib2.urlopen。代理处理通过传递字典参数完成urllib.urlopen，可以通过使用ProxyHandler对象来获得 。

*在版本3.2中更改：*添加了*cafile*和*capath*。

*在版本3.2中更改：*如果可能，现在支持HTTPS虚拟主机（即，如果 ssl.HAS_SNI为true）。

*3.2版新增功能：数据*可以是一个可迭代的对象。

*在版本3.3中更改：*添加了*cadefault*。

*在版本3.4.3 中*进行了*更改：*已添加*上下文*。

自 3.6 版弃用：cafile，*capath* 和 *cadefault* 已被弃用，以支持*上下文*。请ssl.SSLContext.load_cert_chain()改用，或者让我们 ssl.create_default_context()为您选择系统的可信CA证书。

urllib.request.**install_opener**(*opener*)

安装一个OpenerDirector实例作为默认的全局开启器。如果你想让urlopen使用那个开罐器，只需要安装一个开罐器; 否则，简单地调用OpenerDirector.open()代替 urlopen()。代码不检查真实的 OpenerDirector，任何具有合适接口的类都可以工作。

urllib.request.**build_opener**([*handler*, ...])

返回一个OpenerDirector实例，按照给定的顺序链接处理程序。*处理程序* s可以是实例BaseHandler或子类BaseHandler（在这种情况下，必须可以在不带任何参数的情况下调用构造函数）。以下类的实例将在前面*的处理程序*S，除非*处理器*小号含有它们，将它们的它们的实例或亚类：ProxyHandler（如果检测到代理设置），，UnknownHandler，HTTPHandler，HTTPDefaultErrorHandler，HTTPRedirectHandler，FTPHandler，。FileHandlerHTTPErrorProcessor

如果Python安装有SSL支持（即，如果ssl模块可以导入），HTTPSHandler也将被添加。

一个BaseHandler子类，还可以改变其handler_order 属性，修改其在处理程序列表中的位置。

urllib.request.**pathname2url**(*路径*)

将*路径*的本地语法的路径名*路径*转换为URL的路径组件中使用的表单。这不会产生完整的网址。返回值将使用该quote()函数引用。

urllib.request.**url2pathname**(*路径*)

将路径组件*路径*从百分比编码的URL 转换为路径的本地语法。这不接受完整的网址。此功能用于 unquote() 解码*路径*。

urllib.request. getproxies ( )

此帮助函数将代理服务器URL映射的方案字典返回。它以<scheme>_proxy大小写敏感的方式首先扫描所有操作系统中名为变量的环境，并在无法找到它时从Mac OS X系统配置Mac OS X和Windows系统注册表Windows中查找代理信息。如果小写和大写环境变量都存在（并且不同意），则小写是首选。

> **注意:** 如果REQUEST_METHOD设置了环境变量（通常表示脚本正在CGI环境中运行），则环境变量HTTP_PROXY （大写 _PROXY ）将被忽略。这是因为该变量可以由客户端使用“Proxy：”HTTP标头注入。如果您需要在CGI环境中使用HTTP代理，请ProxyHandler明确使用 ，或确保变量名是小写（或至少_proxy后缀）。

提供以下课程：

class urllib.request. Request (*url*，*data = None*，*headers = {}*，*origin_req_host = None*，*unverifiable = False*，*method = None*)

这个类是一个URL请求的抽象。

*网址*应该是包含有效网址的字符串。

*数据*必须是指定发送到服务器的附加数据的对象，或者None如果不需要这样的数据。目前HTTP请求是唯一使用*数据的*请求。支持的对象类型包括字节，文件类对象和迭代器。如果没有 Content-Length ，也没有Transfer-Encoding报头字段已经被设置，HTTPHandler将根据的类型来设置这些标题*数据*。 Content-Length将用于发送字节对象，同时按照中的规定Transfer-Encoding: chunked**RFC 7230**第3.3.1节将用于发送文件和其他可迭代数据。

对于HTTP POST请求方法，*数据*应该是标准*application / x-www-form-urlencoded*格式的缓冲区。该 urllib.parse.urlencode() 函数采用2元组的映射或序列，并以此格式返回ASCII字符串。在用作*数据*参数之前，它应该被编码为字节。

*标题*应该是一个字典，并将被视为 add_header() 与每个键和值作为参数调用。这通常用于“欺骗” User-Agent头部值，浏览器用它来标识自己 - 一些HTTP服务器只允许来自普通浏览器而不是脚本的请求。例如，Mozilla Firefox可能会将自己标识为，而 默认的用户代理字符串是 （在Python 2.6上）。 "Mozilla/5.0 (X11; U; Linux i686) Gecko/20071127 Firefox/2.0.0.11"urllib"Python-urllib/2.6"

Content-Type如果*数据* 参数存在，应该包含适当的头文件。如果未提供此标题且*数据* 不是无，则将作为默认值添加。Content-Type: application/x-www-form-urlencoded

最后两个参数只对正确处理第三方HTTP cookies感兴趣：

origin_req_host 应该是根据定义的原始事务的请求主机**RFC 2965**。它默认为http.cookiejar.request_host(self)。这是用户发起的原始请求的主机名或IP地址。例如，如果请求是针对HTML文档中的图像的，则该请求应该是包含图像的页面请求的请求主机。

*无法验证*应该表明请求是否无法验证，如RFC 2965所定义。它默认为False。无法验证的请求是其用户没有选择批准的URL。例如，如果请求是针对HTML文档中的图像，并且用户无权批准自动获取图像，则应该是正确的。

*方法*应该是一个字符串，表示将使用的HTTP请求方法（例如'HEAD'）。如果提供，它的值将存储在 method 属性中并被使用 get_method()。默认值是'GET'如果*数据*是None或'POST'以其他方式。子类可以通过method在类中设置属性来指示不同的默认方法。

> **注意：** 如果数据对象无法多次传送其内容（例如文件或可以产生一次内容的迭代文件），并且重新请求HTTP重定向或身份验证请求，则请求将无法按预期工作。该*数据*被发送到头部后马上HTTP服务器。在图书馆没有100个继续期待的支持。

版本3.3中已更改： Request.method参数已添加到Request类中。

在版本3.4中更改： Request.method可以在课程级别指定默认值。

在版本3.6中进行了更改： 如果Content-Length未提供错误，并且*数据*既不None是字节对象，也不会引发错误。回退使用分块传输编码。

*类*urllib.request.OpenerDirector

该OpenerDirector课程通过BaseHandler链接打开URL。它管理处理程序的链接，并从错误中恢复。

*类*urllib.request.BaseHandler

这是所有注册处理程序的基类 - 并且只处理简单的注册机制。

*类*urllib.request.HTTPDefaultErrorHandler

一个为HTTP错误响应定义默认处理程序的类; 所有的回应都会变成HTTPError例外。

*类*urllib.request.HTTPRedirectHandler

一个类来处理重定向。

class urllib.request.HTTPCookieProcessor（*cookiejar* = *None*）

一个类来处理HTTP Cookie。

*类*urllib.request.ProxyHandler（*代理=无*）

导致请求通过代理。如果提供了*代理服务器*，它必须是一个字典映射协议名称到代理服务器的URL。缺省值是从环境变量中读取代理列表 <protocol>_proxy。如果未设置代理环境变量，则在Windows环境中，代理设置可从注册表的"Internet设置"部分获得，而在Mac OS X环境中，将从OS X系统配置框架中检索代理信息。

禁用自动检测代理传递空字典。

该 no_proxy环境变量可用于指定不应通过代理达到的主机; 如果设置，它应该是逗号分隔的主机名后缀列表，可选地 :port 附加后缀，例如 cern.ch,ncsa.uiuc.edu,some.host:8080。

> **注意：** HTTP_PROXY如果REQUEST_METHOD设置了变量，将被忽略; 请参阅文档getproxies()。

*类*urllib.request.HTTPPasswordMgr

保持映射数据库。(realm, uri) -> (user, password)

*类*urllib.request.HTTPPasswordMgrWithDefaultRealm

保持映射数据库。一个领域被认为是一个全面的领域，如果没有其他领域适合，就会搜索这个领域。(realm, uri) -> (user, password)None

*类*urllib.request.HTTPPasswordMgrWithPriorAuth

它的一个变种HTTPPasswordMgrWithDefaultRealm也有一个映射数据库。BasicAuth处理程序可以使用它来确定何时立即发送身份验证凭证，而不是先等待响应。uri -> is_authenticated401

3.5版本中的新功能。

class urllib.request.AbstractBasicAuthHandler(*password_mgr = None*)

这是一个mixin类，可以帮助远程主机和代理进行HTTP身份验证。*password_mgr*，如果给出，应该是兼容的HTTPPasswordMgr；有关必须支持的接口的信息，请参阅HTTPPasswordMgr对象部分。如果*passwd_mgr*还提供is_authenticated和update_authenticated方法（参见HTTPPasswordMgrWithPriorAuth对象），则处理程序将使用该is_authenticated结果中的某个给定URI以确定是否要发送的认证证书与请求。如果为URI is_authenticated返回True，则发送凭证。如果is_authenticated是False，凭证不发送，然后如果一个401收到响应后，请求将与身份验证凭证一起重新发送。如果验证成功，update_authenticated则调用以设置is_authenticated TrueURI，以便后续对URI或其任何超级URI的请求将自动包含验证凭证。

*3.5版*新增is_authenticated功能*：*增加了支持。

class urllib.request.HTTPBasicAuthHandler(*password_mgr = None*)

处理远程主机的身份验证。*password_mgr*，如果给出，应该是兼容的HTTPPasswordMgr；有关必须支持的接口的信息，请参阅HTTPPasswordMgr对象部分。HTTPBasicAuthHandler会ValueError在出现错误的身份验证方案时引发。

class urllib.request.ProxyBasicAuthHandler(*password_mgr = None*)

处理与代理的身份验证。*password_mgr*，如果给出，应该是兼容的HTTPPasswordMgr；有关必须支持的接口的信息，请参阅HTTPPasswordMgr对象部分。

class urllib.request.AbstractDigestAuthHandler(*password_mgr = None*)

这是一个mixin类，可以帮助远程主机和代理进行HTTP身份验证。*password_mgr*，如果给出，应该是兼容的HTTPPasswordMgr；有关必须支持的接口的信息，请参阅HTTPPasswordMgr对象部分。

class urllib.request.HTTPDigestAuthHandler(*password_mgr = None*)

处理远程主机的身份验证。*password_mgr*，如果给出，应该是兼容的HTTPPasswordMgr；有关必须支持的接口的信息，请参阅HTTPPasswordMgr对象部分。当同时添加摘要式身份验证处理程序和基本身份验证处理程序时，首先会尝试摘要式身份验证。如果摘要认证再次返回40x响应，则将其发送到基本认证处理程序以处理。此处理程序方法将ValueError在出现Digest或Basic以外的身份验证方案时引发。

*在版本3.3中更改：*引发ValueError不支持的身份验证方案。

class urllib.request.ProxyDigestAuthHandler（*password_mgr = None* ）

处理与代理的身份验证。*password_mgr*，如果给出，应该是兼容的HTTPPasswordMgr; 有关必须支持的接口的信息，请参阅HTTPPasswordMgr对象部分 。

*类*urllib.request.HTTPHandler

一个处理HTTP URL打开的类。

*类*urllib.request.HTTPSHandler（*debuglevel = 0*，*context = None*，*check_hostname = None* ）

一个处理HTTPS URL打开的类。 *上下文*和*check_hostname* 的含义与中相同http.client.HTTPSConnection。

*在版本3.2中更改：*添加了*上下文*和*check_hostname*。

*类*urllib.request.FileHandler

打开本地文件。

*类*urllib.request.DataHandler

打开数据网址。

3.4版新增功能

*类*urllib.request.FTPHandler

打开FTP URL。

*类*urllib.request.CacheFTPHandler

打开FTP URL，保持打开的FTP连接缓存，以尽量减少延迟。

*类*urllib.request.UnknownHandler

一个全能的类来处理未知的URL。

*类*urllib.request.HTTPErrorProcessor

处理HTTP错误响应。

21.6.1。请求对象

以下方法描述了Request公共接口，所有可以在子类中被覆盖。它还定义了几个公共属性，客户可以使用它们来检查解析的请求。

Request.full_url

传递给构造函数的原始URL。

在版本3.4中更改。

Request.full_url是一个具有setter，getter和deleter的属性。获取 full_url返回带有片段的原始请求URL（如果存在）。

Request.type
URI方案。

Request.host
URI权限（通常是主机），但也可能包含由冒号分隔的端口。

Request.origin_req_host
请求的原始主机，不带端口。

Request.selector
URI路径。如果Request使用代理，则选择器将是传递给代理的完整URL。

Request.data
请求的实体主体，None如果没有指定。

*在版本3.4中*更改：Request.data现在更改值将删除“Content-Length”标题（如果它以前已设置或计算）。

Request.unverifiable
布尔值，表示请求是否是RFC 2965定义的不可验证的。

Request.method
要使用的HTTP请求方法。默认情况下，它的值是None，这意味着get_method()它将对要使用的方法进行正常的计算。它的值可以通过get_method()在Request子类中设置类级别来提供默认值，或者通过*方法*参数将值传递给Request构造函数来设置（因此覆盖默认计算）。

3.3版本的新功能

*在版本3.4中更改：*现在可以在子类中设置默认值; 以前它只能通过构造函数参数来设置。

Request.get_method()
返回一个指示HTTP请求方法的字符串。如果Request.method不是None，则返回它的值，否则返回'GET'如果Request.data是None，或者'POST'不是。这只对HTTP请求有意义。

版本3.3中更改： get_method现在查看的值Request.method。

Request.add_header(*key*, *val*)
向请求添加另一个标头。除了HTTP处理程序之外，所有处理程序都会忽略标题，并将它们添加到发送到服务器的标题列表中。请注意，不能有多个具有相同名称的标题，并且以后的调用将覆盖以前的调用，以防万一*键发生*冲突。目前，这并不是HTTP功能的损失，因为所有在不止一次使用时具有含义的标题具有仅使用一个标题获得相同功能的（标题特定的）方式。

Request.add_unredirected_header(*key*, *header*)
添加不会添加到重定向请求的标头。

Request.has_header(*header*)
返回实例是否具有指定的头（同时检查常规和未重定向）。

Request.remove_header（*header*）

从请求实例中删除已命名的头文件（来自常规和未重定向的头文件）。

3.4版新增功能

Request.get_full_url（）

返回构造函数中给出的URL。

在版本3.4中更改。

返回 Request.full_url

Request.set_proxy（*主机*，*类型*）

通过连接到代理服务器来准备请求。该*主机*和*类型*将取代这些实例，并且该实例的选择将是在构造函数中给出的原始URL。

Request.get_header（*header_name*，*default* = *None*）

返回给定标题的值。如果标题不存在，则返回默认值。

Request.header_items（）

返回请求标头的元组列表（header_name，header_value）。

*在版本3.4中进行了更改：*自3.3版本以来已弃用的请求方法add_data，has_data，get_data，get_type，get_host，get_selector，get_origin_req_host和is_unverifiable。

21.6.2。OpenerDirector对象

OpenerDirector 实例具有以下方法：

OpenerDirector.add_handler（*处理程序*）

*处理程序*应该是一个实例BaseHandler。搜索以下方法并添加到可能的链中（请注意，HTTP错误是一种特殊情况）。

- protocol_open()- 表示处理程序知道如何打开*协议*URL。
- http_error_type()- 表示处理程序知道如何使用HTTP错误代码*类型*处理HTTP错误。
- protocol_error()- 表示处理程序知道如何处理来自（非http）*协议的*错误。
- protocol_request()- 表示处理程序知道如何预处理*协议*请求。
- protocol_response()- 表示处理程序知道如何后处理*协议*响应。

OpenerDirector.open（*url*，*data* = *None* [，*timeout*]）

打开给定的*URL*（可以是请求对象或字符串），可以传递给定的*数据*。引发的参数，返回值和异常与那些urlopen()（它只是open() 在当前安装的全局中调用方法OpenerDirector）相同。可选的*timeout*参数指定阻止诸如连接尝试（如果未指定，将使用全局默认超时设置）等操作的超时（以秒为单位）。超时功能实际上仅适用于HTTP，HTTPS和FTP连接）。

OpenerDirector.error（*proto*，** args*）

处理给定协议的错误。这将使用给定的参数（特定于协议）调用给定协议的注册错误处理程序。HTTP协议是一种特殊情况，它使用HTTP响应代码来确定特定的错误处理程序; 请参阅http_error_*() 处理程序类的方法。

所引发的返回值和异常与那些返回值和异常相同urlopen()。

OpenerDirector对象分三个阶段打开URL：

在每个阶段中调用这些方法的顺序是通过对处理程序实例进行排序来确定的。

1. 每个使用类似方法的处理程序都会protocol_request()调用该方法来预处理请求。
2. protocol_open()调用名为like的方法来处理请求。当处理程序返回非None 值（即响应）或引发异常（通常 URLError）时，此阶段结束。允许异常传播。

   实际上，上述算法首先被尝试用于名为的方法 default_open()。如果所有这些方法都返回None，则对于名为like的方法重复该算法protocol_open()。如果所有这些方法都返回 None，则对于名为的方法重复该算法 unknown_open()。

   请注意，这些方法的实现可能涉及调用父 OpenerDirector实例open()和 error()方法。
3. 每个使用类似方法的处理程序都会protocol_response()调用该方法来后处理响应。

21.6.3。BaseHandler对象

BaseHandler对象提供了一些直接有用的方法，以及其他旨在被派生类使用的方法。这些用于直接使用：

BaseHandler.**add_parent**(*导演*)
添加导演作为父项。

BaseHandler.**close**( )
删除任何父母。

以下属性和方法只能由派生自的类使用 BaseHandler。

> **注意：** 该惯例已被采用，定义的子类 protocol_request()或protocol_response()方法被命名 *Processor; 所有其他人都被命名*Handler。

BaseHandler.**parent**
一个有效的OpenerDirector，可以用来打开使用不同的协议，或处理错误。

BaseHandler.**default_open**(*req*)
这个方法*没有*被定义BaseHandler，但是如果他们想要捕获所有的URL，子类应该定义它。

这个方法，如果实施，将由父母调用 OpenerDirector。它应该返回一个类似于文件的对象，如open() of OpenerDirector或者返回值中所述None。它应该提高URLError，除非发生真正的例外事件（例如，MemoryError不应映射到 URLError）。

此方法将在任何协议特定的打开方法之前调用。

BaseHandler.protocol_open（*req*）

这个方法*没有*被定义BaseHandler，但是如果他们想要使用给定的协议处理URL，子类应该定义它。

该方法（如果已定义）将由父级调用OpenerDirector。返回值应该与for相同default_open()。

BaseHandler.unknown_open（*req*）

这个方法*没有*定义BaseHandler，但是子类应该定义它，如果他们想捕获所有没有特定注册处理器的URL来打开它。

这个方法如果被实现的话，将会被调用。返回值应该与for相同。parent OpenerDirectordefault_open()

BaseHandler.http_error_default（*req*，*fp*，*code*，*msg*，*hdrs*）

这个方法*没有*在中定义BaseHandler，但是如果它们打算提供一个捕获所有其他未处理的HTTP错误的子类，它应该覆盖它。它会被OpenerDirector错误自动调用，通常不会在其他情况下调用。

*req*将是一个Request对象，*fp*将是一个带有HTTP错误体的文件类对象，*代码*将是错误的三位代码，*msg*将是代码的用户可见解释，而*hdrs*将是一个映射对象与错误的标题。

返回的值和异常应该与那些相同urlopen()。

BaseHandler.http_error_nnn（*req*，*fp*，*code*，*msg*，*hdrs*）

*nnn*应该是一个三位数的HTTP错误代码。这个方法也没有被定义BaseHandler，但是如果它存在的话，在一个子类的一个实例上，当代码为*nnn*的HTTP错误发生时，它将被调用。

子类应该重写此方法来处理特定的HTTP错误。

提出的参数，返回值和异常应该与for相同http_error_default()。

BaseHandler.protocol_request（*req*）

这个方法*没有*在中定义BaseHandler，但是如果他们想要预处理给定协议的请求，子类就应该定义它。

该方法（如果已定义）将由父级调用OpenerDirector。*请求*将成为一个Request对象。返回值应该是一个Request对象。

BaseHandler.protocol_response（*req*，*回复*）

这个方法*没有*在中定义BaseHandler，但是如果子类想要后处理给定协议的响应，子类就应该定义它。

该方法（如果已定义）将由父级调用OpenerDirector。*请求*将成为一个Request对象。*响应*将是一个实现与返回值相同的接口的对象urlopen()。返回值应该实现与返回值相同的接口urlopen()。

21.6.4。HTTPRedirectHandler对象

注意： 某些HTTP重定向需要来自此模块的客户端代码的操作。如果是这样的话，HTTPError就提出来了。看到**RFC 2616**详细介绍了各种重定向代码的精确含义。

HTTPError如果HTTPRedirectHandler带有一个不是HTTP，HTTPS或FTP URL的重定向URL，那么作为安全考虑引发一个异常。

HTTPRedirectHandler.**redirect_request**(*req*, *fp*, *code*, *msg*, *hdrs*, *newurl*)

返回Request或None响应重定向。http_error_30*()当从服务器收到重定向时，这由方法的默认实现调用。如果应该发生重定向，则返回一个新的Request以允许http_error_30*()执行重定向到*newurl*。否则，HTTPError如果没有其他处理程序尝试处理此URL，则返回，或者None如果您不能，则返回另一个处理程序。

> **注意：** 此方法的默认实现不严格遵循 **RFC 2616**指出，301和302对POST请求的响应不得在没有用户确认的情况下自动重定向。实际上，浏览器确实允许自动重定向这些响应，将POST更改为a GET，并且默认实现会重现此行为。

HTTPRedirectHandler.**http_error_301**(*req*, *fp*, *code*, *msg*, *hdrs*)

重定向到Location:或URI:网址。OpenerDirector当获得HTTP“永久移动”响应时，父级调用此方法。

HTTPRedirectHandler.**http_error_302**(*req*, *fp*, *code*, *msg*, *hdrs*)

同样的http_error_301()，但呼吁'找到'的回应。

HTTPRedirectHandler.**http_error_303**(*req*, *fp*, *code*, *msg*, *hdrs*)

同样的http_error_301()，但呼吁'看到其他'的回应。

HTTPRedirectHandler.**http_error_307**(*req*, *fp*, *code*, *msg*, *hdrs*)

与之相同http_error_301()，但呼吁“临时重定向”响应。

21.6.5。HTTPCookieProcessor对象

HTTPCookieProcessor 实例有一个属性：

HTTPCookieProcessor.**cookiejar**

在http.cookiejar.CookieJar其中存储Cookie。

21.6.6。ProxyHandler对象

ProxyHandler.**protocol_open**(*请求*)

对于在构造函数中给出的*代理*字典中具有代理的每个 *协议*，ProxyHandler都会有一个方法。该方法将通过调用修改通过代理的请求，并调用链中的下一个处理程序以实际执行协议。protocol_open()request.set_proxy()

21.6.7。HTTPPasswordMgr对象

这些方法可用于HTTPPasswordMgr和 HTTPPasswordMgrWithDefaultRealm对象。

HTTPPasswordMgr. add_password (*realm* , *uri* , *user* , *passwd*)

*uri*可以是单个URI，也可以是一系列URI。*realm* , *user*和 *passwd*必须是字符串。当给定*领域的*认证和给定URI的任何给定的超级URI时，这导致用作认证令牌。(user, passwd)

HTTPPasswordMgr. find_user_password (*realm* , *authuri*)

获取给定领域和URI的用户/密码（如果有的话）。如果没有匹配的用户/密码，此方法将返回 。(None, None)

对于HTTPPasswordMgrWithDefaultRealm对象，None如果给定的*领域*没有匹配的用户/密码，则搜索*领域*。

21.6.8。HTTPPasswordMgrWithPriorAuth对象

此密码管理器扩展HTTPPasswordMgrWithDefaultRealm为支持始终发送验证凭证的跟踪URI。

HTTPPasswordMgrWithPriorAuth. add_password (*realm* , *uri* , *user* , *passwd* , *is_authenticated = False*)

realm , *uri* , *user* , *passwd* 等 HTTPPasswordMgr.add_password() 。*is_authenticated*is_authenticated为给定的URI或URI列表设置标志的初始值。如果*is_authenticated*被指定为True，则*领域*被忽略。

HTTPPasswordMgr. find_user_password (*领域* , *authuri*)

与HTTPPasswordMgrWithDefaultRealm对象相同

HTTPPasswordMgrWithPriorAuth. update_authenticated (*self* , *uri* , *is_authenticated = False*)

更新is_authenticated给定*URI*或URI列表的标志。

HTTPPasswordMgrWithPriorAuth. is_authenticated (*self* , *authuri*)

返回is_authenticated给定URI 的标志的当前状态。

21.6.9。AbstractBasicAuthHandler对象

AbstractBasicAuthHandler. http_error_auth_reqed (*authreq* , *host* , *req* , *headers*)

通过获取用户/密码对来处理认证请求，然后重新尝试请求。 *authreq*应该是请求中包含有关域的信息的头的名称，*host*指定要认证的URL和路径，*req*应该是（失败的）Request对象，并且 *头*应该是错误头。

*主机*是权威（例如"python.org"）或包含权威组件（例如"http://python.org/"）的URL。在任一情况下，权限不能包含userinfo的组分（所以，"python.org"和

"python.org:80"都很好，"joe:password@python.org"是不是）。

10年6月21日。HTTPBasicAuthHandler对象

HTTPBasicAuthHandler.http_error_401（*req*，*fp*，*code*，*msg*，*hdrs* ）

如果可用，请重新尝试带有认证信息的请求。

11年6月21日。ProxyBasicAuthHandler对象

ProxyBasicAuthHandler.http_error_407（*req*，*fp*，*code*，*msg*，*hdrs* ）

如果可用，请重新尝试带有认证信息的请求。

12年6月21日。AbstractDigestAuthHandler对象

AbstractDigestAuthHandler.http_error_auth_reqed（*authreq*，*host*，*req*，*headers* ）

*AUTHREQ*应该在哪里被包括在所述请求有关领域中的信息的标题的名称，*主机*应该验证到主机，*REQ* 应该是（失败）Request对象，以及*标头*应该是错误标题。

13年6月21日。HTTPDigestAuthHandler对象

HTTPDigestAuthHandler.http_error_401（*req*，*fp*，*code*，*msg*，*hdrs* ）

如果可用，请重新尝试带有认证信息的请求。

14年6月21日。ProxyDigestAuthHandler对象

ProxyDigestAuthHandler.http_error_407（*req*，*fp*，*code*，*msg*，*hdrs* ）

如果可用，请重新尝试带有认证信息的请求。

15年6月21日。HTTPHandler对象

HTTPHandler.http_open（*req* ）

发送HTTP请求，可以是GET或POST，具体取决于 req.has_data()。

16年6月21日。HTTPSHandler对象

HTTPSHandler.https_open（*req* ）

发送HTTPS请求，可以是GET或POST，具体取决于 req.has_data()。

17年6月21日。FileHandler对象

FileHandler.file_open（*req*）

如果没有主机名或主机名是本地打开文件'localhost'。

*在版本3.2中更改：*此方法仅适用于本地主机名。当给出远程主机名时，会产生一个URLError。

18年6月21日。DataHandler的对象

DataHandler.data_open（*req*）

阅读数据网址。这种URL包含URL本身编码的内容。数据URL语法在中指定**RFC 2397**。这个实现忽略了base64编码数据URL中的空白，所以URL可以被包装在任何源文件中。但即使有些浏览器不介意在base64编码数据URL末尾丢失的填充，此实现将引发ValueError这种情况。

19年6月21日。FTPHandler对象

FTPHandler.ftp_open（*req*）

打开*req*指示的FTP文件。登录始终使用空的用户名和密码完成。

20年6月21日。CacheFTPHandler对象

CacheFTPHandler对象是FTPHandler具有以下附加方法的对象：

CacheFTPHandler.setTimeout（*t*）

将连接超时设置为*t*秒。

CacheFTPHandler.setMaxConns（*m*）

将最大数量的缓存连接设置为*m*。

21年6月21日。UnknownHandler对象

UnknownHandler.unknown_open（）

引发URLError异常。

22年6月21日。HTTPErrorProcessor对象

HTTPErrorProcessor.http_response（）

处理HTTP错误响应。

对于200个错误代码，响应对象立即返回。

对于非200错误代码，这只是将作业传递给 protocol_error_code() 处理程序方法 OpenerDirector.error() 。最终，如果没有其他处理程序处理该错误，HTTPDefaultErrorHandler则会引发一次 HTTPError。

HTTPErrorProcessor.https_response( )

处理HTTPS错误响应。

行为与之相同http_response()。

23年6月21日。示例

除下面的例子外，HOWTO在使用urllib包获取互联网资源时给出了更多的例子 。

这个例子获取python.org主页面并显示它的前300个字节。

```
>>> import urllib.request
>>> with urllib.request.urlopen('http://www.python.org/') as f:
...     print(f.read(300))
...
b'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">\n\n\n<html
xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">\n\n<head>\n
<meta http-equiv="content-type" content="text/html; charset=utf-8" />\n
<title>Python Programming '
```

请注意，urlopen返回一个字节对象。这是因为urlopen无法自动确定从HTTP服务器接收到的字节流的编码。通常，一旦程序确定或猜测到合适的编码，程序就会将返回的字节对象解码为字符串。

以下W3C文档https://www.w3.org/International/O-charset列出了（X）HTML或XML文档可以指定其编码信息的各种方式。

由于python.org网站使用其元标记中指定的*utf-8*编码，因此我们将使用相同的解码字节对象。

```
>>> with urllib.request.urlopen('http://www.python.org/') as f:
...     print(f.read(100).decode('utf-8'))
...
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtm
```

不使用上下文管理器方法也可以实现相同的结果 。

```
>>> import urllib.request
>>> f = urllib.request.urlopen('http://www.python.org/')
>>> print(f.read(100).decode('utf-8'))
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtm
```

在下面的例子中，我们正在向CGI的stdin发送数据流并读取它返回给我们的数据。请注意，此示例仅在Python安装支持SSL时起作用。

```
>>> import urllib.request
>>> req = urllib.request.Request(url='https://localhost/cgi-bin/test.cgi',
...                       data=b'This data is passed to stdin of the CGI')
>>> with urllib.request.urlopen(req) as f:
...     print(f.read().decode('utf-8'))
...
Got Data: "This data is passed to stdin of the CGI"
```

上例中使用的示例CGI的代码是：

```
#!/usr/bin/env python
import sys
data = sys.stdin.read()
print('Content-type: text/plain\n\nGot Data: "%s"' % data)
```

以下是使用以下方法执行PUT请求的示例Request：

```
import urllib.request
DATA = b'some data'
req = urllib.request.Request(url='http://localhost:8080', data=DATA,method='PUT')
with urllib.request.urlopen(req) as f:
    pass
print(f.status)
print(f.reason)
```

使用基本HTTP认证：

```
import urllib.request
# Create an OpenerDirector with support for Basic HTTP Authentication...
auth_handler = urllib.request.HTTPBasicAuthHandler()
auth_handler.add_password(realm='PDQ Application',
                          uri='https://mahler:8092/site-updates.py',
                          user='klem',
                          passwd='kadidd!ehopper')
opener = urllib.request.build_opener(auth_handler)
# ...and install it globally so it can be used with urlopen.
urllib.request.install_opener(opener)
urllib.request.urlopen('http://www.example.com/login.html')
```

build_opener()提供了许多默认的处理程序，包括a ProxyHandler。默认情况下，ProxyHandler使用名为的环境变量<scheme>_proxy，其中<scheme>涉及的URL方案在哪里。例如，http_proxy读取环境变量以获取HTTP代理的URL。

本示例将默认值替换为ProxyHandler使用编程提供的代理URL的默认值，并添加了代理授权支持ProxyBasicAuthHandler。

```
proxy_handler = urllib.request.ProxyHandler({'http': 'http://www.example.com:3128/'})
proxy_auth_handler = urllib.request.ProxyBasicAuthHandler()
proxy_auth_handler.add_password('realm', 'host', 'username', 'password')

opener = urllib.request.build_opener(proxy_handler, proxy_auth_handler)
# This time, rather than install the OpenerDirector, we use it directly:
opener.open('http://www.example.com/login.html')
```

添加HTTP标头：

对构造函数使用*headers*参数`Request`，或者：

```
import urllib.request
req = urllib.request.Request('http://www.example.com/')
req.add_header('Referer', 'http://www.python.org/')
# Customize the default User-Agent header value:
req.add_header('User-Agent', 'urllib-example/0.1 (Contact: . . .)')
r = urllib.request.urlopen(req)
```

`OpenerDirector`自动为每个*用户*添加一个*User-Agent*头`Request`。要改变这一点：

```
import urllib.request
opener = urllib.request.build_opener()
opener.addheaders = [('User-agent', 'Mozilla/5.0')]
opener.open('http://www.example.com/')
```

另外，请记住，传递给（或 ）时会添加一些标准标题（*Content-Length*， *Content-Type*和*Host*）。`Request``urlopen()``OpenerDirector.open()`

以下是使用该`GET`方法检索包含参数的URL 的示例会话：

```
>>> import urllib.request
>>> import urllib.parse
>>> params = urllib.parse.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> url = "http://www.musi-cal.com/cgi-bin/query?%s" % params
>>> with urllib.request.urlopen(url) as f:
...     print(f.read().decode('utf-8'))
...
```

以下示例使用该`POST`方法。请注意，urlencode的params输出在作为数据发送到urlopen之前被编码为字节：

```
>>> import urllib.request
>>> import urllib.parse
>>> data = urllib.parse.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> data = data.encode('ascii')
>>> with urllib.request.urlopen("http://requestb.in/xrbl82xr", data) as f:
...     print(f.read().decode('utf-8'))
...
```

以下示例使用明确指定的HTTP代理覆盖环境设置：

```
>>> import urllib.request
>>> proxies = {'http': 'http://proxy.example.com:8080/'}
>>> opener = urllib.request.FancyURLopener(proxies)
>>> with opener.open("http://www.python.org") as f:
...     f.read().decode('utf-8')
...
```

以下示例完全不使用代理，覆盖环境设置：

```
>>> import urllib.request
>>> opener = urllib.request.FancyURLopener({})
>>> with opener.open("http://www.python.org/") as f:
...     f.read().decode('utf-8')
...
```

24年6月21日。旧版界面

以下函数和类从Python 2模块移植 `urllib`（而不是`urllib2`）。他们可能会在将来某个时候被弃用。

`urllib.request.` **`urlretrieve`** `(` *url*，*filename* = *None*，*reporthook* = *None*，*data* = *None* `)`

将由URL表示的网络对象复制到本地文件。如果URL指向本地文件，则除非提供文件名，否则不会复制该对象。返回一个元组，其中*文件名*是可以找到该对象的本地文件名，并且*标头*是返回的对象的方法（对于远程对象）。例外情况与之相同。`(filename, headers) info() urlopen() urlopen()`

第二个参数（如果存在）指定要复制到的文件位置（如果不存在，则该位置将是具有生成名称的临时文件）。第三个参数（如果存在的话）是一个钩子函数，在建立网络连接时会被调用一次，之后每个块被读取一次。该钩子将传递三个参数; 到目前为止传输的块的数量，以字节为单位的块大小以及文件的总大小。第三个参数可能`-1`在较旧的FTP服务器上，它们不响应检索请求而返回文件大小。

以下示例说明了最常见的使用场景：

```
>>> import urllib.request
>>> local_filename, headers = urllib.request.urlretrieve('http://python.org/')
>>> html = open(local_filename)
>>> html.close()
```

如果*URL*使用`http:`方案标识符，则可以给出可选的*数据*参数来指定`POST`请求（通常请求类型是`GET`）。的*数据*参数必须是在标准bytes对象*应用程序/ x-WWW窗体-urlencoded*格式; 看`urllib.parse.urlencode()`功能。

`urlretrieve()` `ContentTooShortError`当它检测到可用数据量小于预期量（这是*Content-Length*报头报告的大小）时 将会增加。例如，当下载被中断时，会发生这种情况。

该*内容长度*被视为一个下界：如果有更多的数据读取，读取urlretrieve更多的数据，但如果较少的数据是可用的，它引发异常。

在这种情况下，您仍可以检索下载的数据，并将其存储在`content`异常实例的 属性中。

如果未提供*Content-Length*标头，则urlretrieve无法检查其下载的数据的大小，只是将其返回。在这种情况下，您只需假定下载成功。

`urllib.request.` **`urlcleanup`** `( )`

清除先前调用可能遗留的临时文件`urlretrieve()`。

*类*urllib.request.URLopener（*代理=无*，*** x509* ）

自3.3版以来已弃用。

打开和阅读URL的基类。除非你需要使用比其他方案来支持开放的对象http:，ftp:或者file:，你可能想使用FancyURLopener。

默认情况下，URLopener该类将发送一个*User-Agent*标头urllib/*VVV*，其中*VVV*是urllib版本号。应用程序可以通过子类定义它们自己的*User-Agent*头， URLopener或者在子类定义FancyURLopener中将类属性 version设置为合适的字符串值。

可选的*代理*参数应该是字典映射方案名称到代理URL，其中空字典将代理完全关闭。它的默认值是None，在这种情况下，如果存在的话，将使用环境代理设置，如urlopen()上面的定义中所讨论的。

在*x509*中收集的其他关键字参数可用于在使用该https:方案时验证客户端。支持关键字*key_file*和*cert_file*以提供SSL密钥和证书; 两者都需要支持客户端身份验证。

URLopener OSError如果服务器返回错误代码，则对象将引发异常。

open（*fullurl*，*data = None* ）

使用适当的协议打开*fullurl*。此方法设置缓存和代理信息，然后使用其输入参数调用相应的open方法。如果该方案未被识别， open_unknown()则被调用。该*数据*参数的含义相同*数据*的说法 urlopen()。

open_unknown（*fullurl*，*data = None* ）

可覆盖的界面打开未知的URL类型。

retrieve（*url*，*filename = None*，*reporthook = None*，*data = None* ）

检索*url*的内容并将其放在*文件名中*。返回值是由本地文件名和email.message.Message包含响应头的对象（对于远程URL）或None（对于本地URL）组成的元组 。调用者必须打开并读取*文件名*的内容。如果没有给出*文件名*并且URL指向本地文件，则返回输入文件名。如果URL非本地和 *文件名*没有给出，则该文件的输出tempfile.mktemp()与输入URL的最后一个路径成分的后缀匹配的后缀。如果有*报告*它必须是一个接受三个数字参数的函数：块号，读入的最大块大小和下载的总大小（如果未知，则为-1）。它将在开始时以及从网络读取每个数据块后调用一次。 对于本地URL，*reporthook*被忽略。

如果*URL*使用http:方案标识符， 则可以给出可选的*数据*参数来指定POST请求（通常请求类型是GET）。的*数据*参数必须在标准 *应用程序/ x-WWW窗体-urlencoded*格式; 看 urllib.parse.urlencode()功能。

version

指定开启器对象的用户代理的变量。为了 urllib告诉服务器它是一个特定的用户代理，在调用基础构造函数之前，在子类中将其设置为类变量或构造函数中。

class urllib.request.FancyURLopener（... ）

自3.3版以来已弃用。

FancyURLopener子类URLopener为下列HTTP响应代码提供默认处理：301,302,303,307和401.对于上面列出的30x响应代码，*Location*标头用于获取实际的URL。对于401响应代码（需要认证），执行基本的HTTP认证。对于30x响应代码，递归受限于*maxtries*属性的值，默认值为10。

对于所有其他响应代码，将http_error_default()调用该方法，您可以在子类中重写以适当地处理错误。

> **注意：** 根据信 未经用户确认，不得自动重定向对POST请求的 **RFC 2616,301**和302响应。实际上，浏览器确实允许自动重定向这些响应，将POST更改为GET，并urllib重现此行为。

构造函数的参数与那些参数相同URLopener。

> **注意：** 执行基本身份验证时，FancyURLopener实例将调用其prompt_user_passwd()方法。默认实现向用户询问控制终端上的所需信息。如果需要，子类可以重写此方法以支持更适当的行为。

本FancyURLopener类提供了应该被重载，以提供适当的行为一个额外的方法：

prompt_user_passwd(*主机*, *领域*)

返回在指定的安全领域中对给定主机上的用户进行身份验证所需的信息。返回值应该是一个元组，可用于基本认证。(user, password)

实施提示在终端上提供这些信息; 应用程序应该重写此方法以在本地环境中使用适当的交互模型。

25年6月21日。urllib.request限制

- 目前，仅支持以下协议：HTTP（版本0.9和1.0），FTP，本地文件和数据URL。

  *在版本3.4中进行了更改：*添加了对数据URL的支持。

- 缓存功能urlretrieve()已被禁用，直到有人找到时间来破解到期时间标题的正确处理。

- 应该有一个函数来查询一个特定的URL是否在缓存中。

- 为了向后兼容，如果URL看起来指向本地文件，但文件无法打开，则使用FTP协议重新解释URL。这有时会导致混淆错误消息。

- 的urlopen()和urlretrieve()功能可能会导致任意长时间的延迟，而等待网络连接被建立。这意味着使用这些函数而不使用线程来构建交互式Web客户端是很困难的。

- 由urlopen() or 返回的urlretrieve()数据是服务器返回的原始数据。这可能是二进制数据（如图像），纯文本或（例如）HTML。HTTP协议在应答头中提供了类型信息，可以通过查看*Content-Type*头来检查。如果返回的数据是HTML，则可以使用该模块html.parser解析它。

- 处理FTP协议的代码不能区分文件和目录。当尝试读取指向无法访问的文件的URL时，这可能会导致意外的行为。如果URL以a结尾/，则假定引用一个目录并将相应地处理。但是，如果尝试读取文件导致550错误（意思是无法找到URL或者由于权限原因而无法访问该URL），那么该路径将被视为目录以处理指定目录时的情况由一个URL但尾随/已经被关闭了。当您尝试获取读取权限使其无法访问的文件时，这可能会导致误导性结果; FTP代码将尝试读取它，以550错误失败，然后执行不可读文件的目录列表。如果需要细粒度控制，请考虑使用ftplib模块，子类FancyURLopener或更改 *_urlopener*以满足您的需求。

21.7。urllib.response- urllib使用的响应类

该urllib.response模块定义了定义像接口这样的最小文件的函数和类，包括read()和readline()。典型的响应对象是一个addinfourl实例，它定义了一个info()方法并返回标题和一个geturl()返回url 的方法。该模块定义的功能由模块内部使用 urllib.request。

21.8。urllib.parse- 将URL解析为组件

源代码： Lib / urllib / parse.py

此模块定义了一个标准接口，用于打破组件中的统一资源定位符（URL）字符串（寻址方案，网络位置，路径等），将组件返回到URL字符串中，并将“相对URL”转换为给出“基本URL”的绝对URL。

该模块已被设计为与相对统一资源定位符上的Internet RFC相匹配。它支持下列URL方案：`file`，`ftp`，`gopher`，`hdl`，`http`，`https`，`imap`，`mailto`，`mms`，`news`，`nntp`，`prospero`，`rsync`，`rtsp`，`rtspu`，`sftp`，`shttp`，`sip`，`sips`，`snews`，`svn`，`svn+ssh`，`telnet`，`wais`，`ws`，`wss`。

该`urllib.parse`模块定义的功能分为两大类：URL解析和URL引用。这些在下面的章节中有详细介绍。

21.8.1。URL解析

URL解析功能专注于将URL字符串拆分为其组件，或将URL组件组合成URL字符串。

`urllib.parse.`**`urlparse`**`(`*`urlstring`*`,` *`scheme=''`*`,` *`allow_fragments=True`*`)`

将URL解析为六个组件，返回一个6元组。这对应于URL的一般结构：`scheme://netloc/path;parameters?query#fragment`。每个元组项都是一个字符串，可能是空的。组件不会在较小的部分中分解（例如，网络位置是单个字符串），并且%escapes不会扩展。上面显示的分隔符不是结果的一部分，除了*路径*组件中的前导斜杠（如果存在）保留。例如：

```
>>> from urllib.parse import urlparse
>>> o = urlparse('http://www.cwi.nl:80/%7Eguido/Python.html')
>>> o
ParseResult(scheme='http', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> o.scheme
'http'
>>> o.port
80
>>> o.geturl()
'http://www.cwi.nl:80/%7Eguido/Python.html'
```

遵循语法规范 **RFC 1808中**，urlparse只有在通过'//'正确引入时才能识别netloc。否则，输入被假定为相对URL，因此以路径组件开始。

```
>>> from urllib.parse import urlparse
>>> urlparse('//www.cwi.nl:80/%7Eguido/Python.html')
ParseResult(scheme='', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> urlparse('www.cwi.nl/%7Eguido/Python.html')
ParseResult(scheme='', netloc='', path='www.cwi.nl/%7Eguido/Python.html',
```

```
          params='', query='', fragment='')
>>> urlparse('help/Python.html')
ParseResult(scheme='', netloc='', path='help/Python.html', params='',
          query='', fragment='')
```

该*方案*论证给出了默认的寻址方案，将仅用于当URL不指定一个。它应该与*urlstring*类型相同（文本或字节），但默认值''始终是允许的，并且会自动转换为b''适当的值。

如果*allow_fragments*参数为false，则不识别片段标识符。相反，它们被解析为路径，参数或查询组件的一部分，并且fragment被设置为返回值中的空字符串。

返回值实际上是一个子类的实例tuple。该类具有以下额外的只读便利属性：

属性	指数	值	价值如果不存在
scheme	0	URL方案说明符	*方案*参数
netloc	1	网络位置部分	空字符串
path	2	分层路径	空字符串
params	3	最后一个路径元素的参数	空字符串
query	4	查询组件	空字符串
fragment	五	片段标识符	空字符串
username		用户名	None
password		密码	None
hostname		主机名（小写）	None
port		端口号为整数（如果存在）	None

如果在URL中指定了无效的端口，则读取该port属性将引发一次ValueError。有关结果对象的更多信息，请参见结构化分析结果部分 。

netloc属性中无与伦比的方括号将引发一个 ValueError。

*在版本3.2中进行了更改：*增加了IPv6 URL解析功能。

*在版本3.3中进行了更改：*现在针对所有URL方案（除非*allow_fragment*为false）对片段进行分析，依照**RFC 3986**。以前，存在支持片段的方案的白名单。

*在版本3.6中更改：现在出现*超出范围的端口号ValueError，而不是返回None。

urllib.parse.parse_qs（*qs*，*keep_blank_values = False*，*strict_parsing = False*，*encoding ='utf-8'*，*errors ='replace'*）

解析作为字符串参数给出的查询字符串（类型为 *application / x-www-form-urlencoded的数据*）。数据作为字典返回。字典键是唯一的查询变量名称，值是每个名称的值列表。

可选参数*keep_blank_values*是一个标志，指示百分比编码查询中的空白值是否应视为空白字符串。真值表示空白字符应保留为空白字符串。默认值为false表示空白值将被忽略并视为未被包含。

可选参数*strict_parsing*是一个标志，指示如何解析错误。如果为false（默认值），则错误将被忽略。如果属实，错误会引发`ValueError`异常。

可选的*编码*和*错误*参数指定如何将百分比编码的序列解码为Unicode字符，如`bytes.decode()`方法所接受。

使用该`urllib.parse.urlencode()`函数（将`doseq`参数设置为`True`）将这些字典转换为查询字符串。

*在版本3.2中更改：*添加*编码*和*错误*参数。

urllib.parse.**parse_qsl**(*qs*, *keep_blank_values = False*, *strict_parsing = False*, *encoding ='utf-8'*, *errors ='replace'*)

解析作为字符串参数给出的查询字符串（类型为 *application / x-www-form-urlencoded的数据*）。数据以名称，值对的列表形式返回。

可选参数*keep_blank_values*是一个标志，指示百分比编码查询中的空白值是否应视为空白字符串。真值表示空白字符应保留为空白字符串。默认值为false表示空白值将被忽略并视为未被包含。

可选参数*strict_parsing*是一个标志，指示如何解析错误。如果为false（默认值），则错误将被忽略。如果属实，错误会引发`ValueError`异常。

可选的*编码*和*错误*参数指定如何将百分比编码的序列解码为Unicode字符，如`bytes.decode()`方法所接受。

使用该`urllib.parse.urlencode()`函数将这些对列表转换为查询字符串。

*在版本3.2中更改：*添加*编码*和*错误*参数。

urllib.parse.**urlunparse**(*部分*)

从返回的元组构造一个URL `urlparse()`。该*部分*参数可以是任何六个项目迭代。如果最初解析的URL具有不必要的分隔符（例如，?带有空查询; RFC声明它们是等同的），则这可能会导致稍微不同的但等同的URL。

urllib.parse.**urlsplit**(*urlstring*, *scheme =''*, *allow_fragments = True*)

这与`urlparse()`URL相似，但不会将参数分开。通常应该使用它，而不是`urlparse()`如果允许将参数应用于URL的*路径*部分的每个段的更新近的URL语法（参见**RFC 2396**）。需要单独的功能来分隔路径段和参数。该函数返回一个5元组:(地址方案，网络位置，路径，查询，片段标识符）。

返回值实际上是一个子类的实例`tuple`。该类具有以下额外的只读便利属性：

属性	指数	值	价值如果不存在
scheme	0	URL方案说明符	*方案*参数
netloc	1	网络位置部分	空字符串
path	2	分层路径	空字符串
query	3	查询组件	空字符串

属性	指数	值	价值如果不存在
fragment	4	片段标识符	空字符串
username		用户名	None
password		密码	None
hostname		主机名（小写）	None
port		端口号为整数（如果存在）	None

如果在URL中指定了无效的端口，则读取该port属性将引发一次ValueError。有关结果对象的更多信息，请参见结构化分析结果部分 。

netloc属性中无与伦比的方括号将引发一个 ValueError。

*在版本3.6中更改：现在出现*超出范围的端口号ValueError，而不是返回None。

urllib.parse.**urlunsplit**（*部分*）

将返回的元组元素合并urlsplit()为一个完整的URL作为字符串。该*部分*参数可以是任何五个项目的迭代。如果最初解析的URL具有不必要的分隔符（例如，带有空查询的？; RFC声明它们是等同的），则这可能会导致稍微不同的但等同的URL。

urllib.parse.**urljoin**（*base*，*url*，*allow_fragments* = *True*）

通过将“基本URL”（*base*）与另一个URL（*url*）组合起来构建完整（“绝对”）URL 。非正式地说，它使用基本URL的组件，特别是寻址方案，网络位置和路径（的一部分）来提供相关URL中缺失的组件。例如：

```
>>> from urllib.parse import urljoin
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html', 'FAQ.html')
'http://www.cwi.nl/%7Eguido/FAQ.html'
```

该*allow_fragments*参数具有相同的含义，默认为 urlparse()。

注意： 如果*url*是绝对URL（即以//or 开头scheme://），则*url*的主机名和/或方案将出现在结果中。例如：

```
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html',
...         '//www.python.org/%7Eguido')
'http://www.python.org/%7Eguido'
```

如果您不想要这种行为，请使用和预处理*url*，urlsplit()并 urlunsplit()删除可能的*方案*和*netloc*部分。

*在版本3.5中进行了更改：*更新了行为以匹配在中定义的语义**RFC 3986**。

urllib.parse.**urldefrag**（*url*）

如果*url*包含片段标识符，则返回 不带片段标识符的*URL*的修改版本，并将片段标识符作为单独的字符串返回。如果*url*中没有片段标识符，则返回未修改的*url*和一个空字符串。

返回值实际上是一个子类的实例tuple。该类具有以下额外的只读便利属性：

属性	指数	值	价值如果不存在
url	0	没有片段的网址	空字符串
fragment	1	片段标识符	空字符串

有关结果对象的更多信息，请参见结构化分析结果部分。

*版本3.2中更改：*结果是结构化对象而不是简单的2元组。

21.8.2。解析ASCII编码字节

URL解析函数最初设计为仅对字符串进行操作。在实践中，能够正确处理引用和编码的URL作为ASCII字节序列很有用。因此，URL此模块中的所有功能解析上操作bytes ，并 bytearray在除对象str的对象。

如果str传入数据，结果将只包含 str数据。如果bytes或bytearray数据被传入，结果将只包含bytes数据。

尝试在单个函数调用中混合str数据bytes或者 bytearray在单个函数调用中调用数据会导致TypeError引发，同时尝试传入非ASCII字节值将触发UnicodeDecodeError。

为了支持str和 之间的结果对象的更容易的转换， bytes来自URL解析函数的所有返回值提供了一种encode()方法（当结果包含str 数据时）或decode()方法（当结果包含bytes 数据时）。这些方法的签名与相应的方法str和bytes方法的签名相匹配（除了默认编码'ascii'不是'utf-8'）。每个产生一个包含bytes数据（对于 encode()方法）或str数据（对于 decode()方法）的相应类型的值。

在调用URL解析方法之前，需要对可能包含非ASCII数据的引用可能不正确的URL进行操作的应用程序需要从字节到字符执行它们自己的解码。

本节中描述的行为仅适用于URL解析功能。在生成或消费字节序列时，URL引用函数使用自己的规则，详见个别URL引用函数的文档。

版本3.2中更改： URL解析函数现在接受ASCII编码的字节序列

21.8.3。结构化分析结果

从结果对象urlparse() ，urlsplit() 和 urldefrag()功能是子类的tuple类型。这些子类添加了这些功能文档中列出的属性，前一节中介绍的编码和解码支持以及其他方法：

urllib.parse.SplitResult.**geturl**()

将原始URL的重新组合版本作为字符串返回。这可能与原始URL有所不同，因为该方案可能会归一化为小写字母，并且可能会删除空的组件。具体来说，将删除空参数，查询和片段标识符。

对于urldefrag()结果，只有空的片段标识符将被删除。对于urlsplit()和urlparse()结果，所有指出的更改将由此方法返回的URL进行。

如果通过原始解析函数返回，则此方法的结果保持不变：

```
>>> from urllib.parse import urlsplit
>>> url = 'HTTP://www.Python.org/doc/#'
>>> r1 = urlsplit(url)
>>> r1.geturl()
'http://www.Python.org/doc/'
>>> r2 = urlsplit(r1.geturl())
>>> r2.geturl()
'http://www.Python.org/doc/'
```

以下类在对str对象进行操作时提供了结构化分析结果的实现：

class urllib.parse.DefragResult（*url*，*fragment* ）

urldefrag()包含str 数据的结果的具体类。该encode()方法返回一个DefragResultBytes 实例。

3.2版本中的新功能

*类*urllib.parse.ParseResult（*方案*，*netloc*，*路径*，*参数*，*查询*，*片段* ）

urlparse()包含str 数据的结果的具体类。该encode()方法返回一个ParseResultBytes 实例。

*类*urllib.parse.SplitResult（*方案*，*netloc*，*路径*，*查询*，*片段* ）

urlsplit()包含str 数据的结果的具体类。该encode()方法返回一个SplitResultBytes 实例。

以下类上操作时提供解析结果的实现方式bytes或bytearray对象：

class urllib.parse.DefragResultBytes（*url*，*fragment* ）

urldefrag()包含bytes 数据的结果的具体类。该decode()方法返回一个DefragResult 实例。

3.2版本中的新功能

*类*urllib.parse.ParseResultBytes（*方案*，*netloc*，*路径*，*参数*，*查询*，*片段* ）

urlparse()包含bytes 数据的结果的具体类。该decode()方法返回一个ParseResult 实例。

3.2版本中的新功能

*类*urllib.parse.SplitResultBytes（*方案*，*netloc*，*路径*，*查询*，*片段* ）

urlsplit()包含bytes 数据的结果的具体类。该decode()方法返回一个SplitResult 实例。

3.2版本中的新功能

21.8.4。URL引用

URL引用函数专注于获取程序数据，并通过引用特殊字符并适当地编码非ASCII文本来使其可以安全地用作URL组件。他们还支持颠倒这些操作，以便从URL组件的内容中重新创建原始数据（如果该任务尚未被上述URL解析功能所覆盖）。

urllib.parse. quote (*string* , *safe ='/'* , *encoding = None* , *errors = None*)

使用转义替换*字符串*中的特殊字符%xx。字母，数字和字符'_.-'从未被引用。默认情况下，此功能用于引用URL的路径部分。可选的*安全*参数指定不应引用的附加ASCII字符 - 其默认值为'/'。

*字符串*可能是a str或a bytes。

可选的*编码*和*错误*参数指定如何处理非ASCII字符，如str.encode()方法所接受。 *编码*默认为'utf-8'。 *错误*默认为'strict'，意味着不受支持的字符引发 UnicodeEncodeError。如果*字符串*是a 或者是a ，则不得提供*编码*和*错误*。 bytesTypeError

请注意，相当于。 quote(string, safe, encoding, errors)quote_from_bytes(string.encode(encoding, errors), safe)

例如：收益率。 quote('/El Niño/')'/El%20Ni%C3%B1o/'

urllib.parse. quote_plus (*string* , *safe =''* , *encoding = None* , *errors = None*)

类似quote()，但也可以用加号替换空格，这是在构建查询字符串以进入URL时引用HTML表单值所需的。除非包含在*保险箱中*，否则原始字符串中的加号会被转义。它也没有*安全的*默认值'/'。

例如：收益率。 quote_plus('/El Niño/')'%2FEl+Ni%C3%B1o%2F'

urllib.parse. quote_from_bytes (*bytes* , *safe ='/'*)

像quote()，但接受一个bytes对象而不是一个 str，并且不执行字符串到字节的编码。

例如：quote_from_bytes(b'a&\xef')收益率'a%26%EF'。

urllib.parse. unquote (*string* , *encoding ='utf-8'* , *errors ='replace'*)

用%xx它们的单字符替换换码。可选的*编码*和*错误*参数指定如何将百分比编码的序列解码为Unicode字符，如bytes.decode()方法所接受。

*字符串*必须是str。

*编码*默认为'utf-8'。 *错误*默认为'replace'，意味着无效序列被占位符替换。

例如：unquote('/El%20Ni%C3%B1o/')收益率。'/El Niño/'

urllib.parse. unquote_plus (*string* , *encoding ='utf-8'* , *errors ='replace'*)

喜欢unquote()，但也可以用空格替换加号，以取消查询HTML表单值。

*字符串*必须是str。

例如：unquote_plus('/El+Ni%C3%B1o/')收益率。'/El Niño/'

urllib.parse. unquote_to_bytes (*字符串*)

用%xx它们的单字节等效替换转义符，然后返回一个 bytes对象。

*字符串*可能是a str或a bytes。

如果它是a `str`，则*字符串*中未转义的非ASCII字符将被编码为UTF-8字节。

例如：`unquote_to_bytes('a%26%EF')`收益率`b'a&\xef'`。

urllib.parse.**urlencode**(*query*，*doseq = False*，*safe =''*，*encoding = None*，*errors = None*，*quote_via = quote_plus*)

将映射对象或可能包含的两元素元组或序列，`str`或者`bytes`对象转换为百分比编码的ASCII文本字符串。如果结果字符串被用作POST 函数的*数据*`urlopen()`，那么它应该被编码为字节，否则会导致一个`TypeError`。

生成的字符串是一系列`key=value`由'&'字符分隔的对，其中*键*和*值*都使用*quote_via* 函数引用。默认情况下，`quote_plus()`用于引用值，这意味着空格被引用为'+'字符，'/'字符编码为`%2F`，它遵循GET请求（`application/x-www-form-urlencoded`）的标准。另一个可以作为*quote_via*传递的*函数*是`quote()`，它将对空格`%20` 进行编码并且不编码'/'字符。为了最大限度地控制所引用内容，请使用 `quote`并指定一个*安全*值。

当使用两元素元组作为*查询*参数时，每个元组的第一个元素是一个键，第二个元素是一个值。value元素本身可以是一个序列，在这种情况下，如果可选参数*doseq*的计算结果为`True`，则为该键的值序列的每个元素生成`key=value`分隔的单个 对'&'。编码字符串中参数的顺序将与序列中参数元组的顺序相匹配。

在*安全*，*编码*和*错误*参数传递到 *quote_via*（该*编码*和*错误*当查询元素是参数仅传递`str`）。

为了反转这个编码过程，`parse_qs()`并`parse_qsl()`在本模块中提供将查询字符串解析为Python数据结构。

请参阅urllib示例以了解如何使用urlencode方法为URL生成查询字符串或为POST生成数据。

*在版本3.2中更改：*查询参数支持字节和字符串对象。

*3.5版本中的新功能：*quote_via参数。

也可以看看:

RFC 3986 - 统一资源标识符
这是目前的标准（STD66）。对urllib.parse模块的任何更改都应符合此。可以观察到某些偏差，主要是出于向后兼容的目的以及主要浏览器中通常观察到的某些事实上的解析要求。

RFC 2732 - URL中文字IPv6地址的格式。
这指定了IPv6 URL的解析要求。

RFC 2396 - 统一资源标识符（URI）：通用语法
描述统一资源名称（URN）和统一资源定位符（URL）的通用语法要求的文档。

RFC 2368 - mailto URL方案。
解析对mailto URL方案的要求。

RFC 1808 - 相对统一资源定位符

此征求意见包括加入绝对和相对URL的规则，其中包括处理边界案件的“异常示例”的相当数量。

RFC 1738 - 统一资源定位符（URL）

这指定绝对URL的形式语法和语义。

21.9。urllib.error- 由urllib.request引发的异常类

源代码： Lib / urllib / error.py

该urllib.error模块为异常引发的异常定义了异常类urllib.request。基本的异常类是URLError。

以下例外情况urllib.error适当提出：

*异常*urllib.error.URLError

处理程序遇到问题时会引发此异常（或派生的异常）。它是一个子类OSError。

reason

这个错误的原因。它可以是一个消息字符串或另一个异常实例。

*在版本3.3中更改：*URLError已经取得了一个子类OSError而不是IOError。

*异常*urllib.error.HTTPError

尽管是一个异常（的一个子类URLError），但 HTTPError它也可以作为非例外的文件返回值（与返回值相同urlopen()）。这在处理异常的HTTP错误时很有用，例如验证请求。

code

RFC 2616中定义的HTTP状态代码。此数值对应于在中找到的代码字典中找到的值http.server.BaseHTTPRequestHandler.responses。

reason

这通常是解释此错误原因的字符串。

headers

导致该问题的HTTP请求的HTTP响应标头 HTTPError。

3.4版新增功能

*异常*urllib.error.ContentTooShortError（*味精*，*内容*）

当urlretrieve()函数检测到下载的数据量小于预期量（由*Content-Length*标题给出）时，会引发此异常。该content属性存储下载的（并且假定被截断的）数据。

21.10。urllib.robotparser-解析器的robots.txt

源代码： Lib / urllib / robotparser.py

该模块提供了一个类，RobotFileParser它回答关于特定用户代理是否可以在发布该robots.txt文件的网站上获取URL的问题。有关robots.txt文件结构的更多详细信息，请参阅http://www.robotstxt.org/orig.html。

class urllib.robotparser.RobotFileParser（*url =''*）

这个类提供了一些方法来读取，解析和回答有关*url*robots.txt文件的问题。

set_url（*url*）

设置引用robots.txt文件的URL。

read（）

读取robots.txtURL并将其提供给解析器。

parse（*线*）

解析行参数。

can_fetch（*useragent*，*url*）

返回True是否允许*useragent*根据解析 文件中包含的规则获取*url*robots.txt。

mtime（）

返回robots.txt文件上次获取的时间。这对于需要robots.txt定期检查新文件的长时间运行的网络蜘蛛非常有用。

modified（）

设置robots.txt文件上次获取到当前时间的时间。

crawl_delay（*useragent*）

返回的值Crawl-delay从参数robots.txt为*用户代理*的问题。如果没有这样的参数，或者它不适用于指定的*useragent*，或者robots.txt此参数的条目具有无效语法，则返回None。

3.6版本中的新功能。

request_rate（*useragent*）

返回的内容Request-rate从参数robots.txt作为命名的元组。如果没有这样的参数，或者它不适用于 指定的*useragent*，或者此参数的条目具有无效语法，则返回。RequestRate(requests, seconds)robots.txtNone

3.6版本中的新功能。

以下示例演示了RobotFileParser该类的基本用法：

```
>>> import urllib.robotparser
>>> rp = urllib.robotparser.RobotFileParser()
>>> rp.set_url("http://www.musi-cal.com/robots.txt")
>>> rp.read()
>>> rrate = rp.request_rate("*")
>>> rrate.requests
3
>>> rrate.seconds
20
>>> rp.crawl_delay("*")
6
>>> rp.can_fetch("*", "http://www.musi-cal.com/cgi-bin/search?city=San+Francisco")
False
>>> rp.can_fetch("*", "http://www.musi-cal.com/")
True
```

21.11。http- HTTP模块

源代码： Lib / http / __ init__.py

http 是一个收集多个用于使用超文本传输协议的模块的软件包：

- http.client是一个低级别的HTTP协议客户端; 用于高级URL开放使用urllib.request
- http.server 包含基于的基本HTTP服务器类 socketserver
- http.cookies 使用cookie实施状态管理的实用程序
- http.cookiejar 提供cookie的持久性

http也是一个通过http.HTTPStatusenum 定义了许多HTTP状态代码和相关消息的模块：

*类*http.HTTPStatus

3.5版本中的新功能。

它的一个子类enum.IntEnum定义了一组HTTP状态码，原因短语和用英文写成的长描述。

用法：

```
>>> from http import HTTPStatus
>>> HTTPStatus.OK
<HTTPStatus.OK: 200>
>>> HTTPStatus.OK == 200
True
>>> http.HTTPStatus.OK.value
200
>>> HTTPStatus.OK.phrase
'OK'
>>> HTTPStatus.OK.description
'Request fulfilled, document follows'
>>> list(HTTPStatus)
[<HTTPStatus.CONTINUE: 100>, <HTTPStatus.SWITCHING_PROTOCOLS: 101>, ...]
```

21.11.1。HTTP状态码

支持的， IANA注册的 状态代码http.HTTPStatus是：

码	枚举名称	细节
100	CONTINUE	HTTP / 1.1 **RFC 7231**，第6.2.1节
101	SWITCHING_PROTOCOLS	HTTP / 1.1 **RFC 7231**，第6.2.2节
102	PROCESSING	WebDAV的 **RFC 2518**，第10.1节
200	OK	HTTP / 1.1 **RFC 7231**，第6.3.1节
201	CREATED	HTTP / 1.1 **RFC 7231**，第6.3.2节
202	ACCEPTED	HTTP / 1.1 **RFC 7231**，第6.3.3节
203	NON_AUTHORITATIVE_INFORMATION	HTTP / 1.1 **RFC 7231**，第6.3.4节
204	NO_CONTENT	HTTP / 1.1 **RFC 7231**，第6.3.5节

205	RESET_CONTENT	HTTP / 1.1 **RFC 7231**，第6.3.6节
206	PARTIAL_CONTENT	HTTP / 1.1 **RFC 7233**，第4.1节
207	MULTI_STATUS	WebDAV的 **RFC 4918**，第11.1节
208	ALREADY_REPORTED	WebDAV绑定扩展 **RFC 5842**，第7.1节（实验）
226	IM_USED	三角洲编码在HTTP中 **RFC 3229**，第10.4.1节
300	MULTIPLE_CHOICES	HTTP / 1.1 **RFC 7231**，第6.4.1节
301	MOVED_PERMANENTLY	HTTP / 1.1 **RFC 7231**，第6.4.2节
302	FOUND	HTTP / 1.1 **RFC 7231**，第6.4.3节
303	SEE_OTHER	HTTP / 1.1 **RFC 7231**，第6.4.4节
304	NOT_MODIFIED	HTTP / 1.1 **RFC 7232**，第4.1节
305	USE_PROXY	HTTP / 1.1 **RFC 7231**，第6.4.5节
307	TEMPORARY_REDIRECT	HTTP / 1.1 **RFC 7231**，第6.4.7节
308	PERMANENT_REDIRECT	永久重定向 **RFC 7238**，第3部分（实验）
400	BAD_REQUEST	HTTP / 1.1 **RFC 7231**，第6.5.1节
401	UNAUTHORIZED	HTTP / 1.1身份验证 **RFC 7235**，第3.1节
402	PAYMENT_REQUIRED	HTTP / 1.1 **RFC 7231**，第6.5.2节
403	FORBIDDEN	HTTP / 1.1 **RFC 7231**，第6.5.3节
404	NOT_FOUND	HTTP / 1.1 **RFC 7231**，第6.5.4节
405	METHOD_NOT_ALLOWED	HTTP / 1.1 **RFC 7231**，第6.5.5节
406	NOT_ACCEPTABLE	HTTP / 1.1 **RFC 7231**，第6.5.6节
407	PROXY_AUTHENTICATION_REQUIRED	HTTP / 1.1身份验证 **RFC 7235**，第3.2节
408	REQUEST_TIMEOUT	HTTP / 1.1 **RFC 7231**，第6.5.7节
409	CONFLICT	HTTP / 1.1 **RFC 7231**，第6.5.8节
410	GONE	HTTP / 1.1 **RFC 7231**，第6.5.9节
411	LENGTH_REQUIRED	HTTP / 1.1 **RFC 7231**，第6.5.10节
412	PRECONDITION_FAILED	HTTP / 1.1 **RFC 7232**，第4.2节
413	REQUEST_ENTITY_TOO_LARGE	HTTP / 1.1 **RFC 7231**，第6.5.11节
414	REQUEST_URI_TOO_LONG	HTTP / 1.1 **RFC 7231**，第6.5.12节
415	UNSUPPORTED_MEDIA_TYPE	HTTP / 1.1 **RFC 7231**，第6.5.13节
416	REQUEST_RANGE_NOT_SATISFIABLE	HTTP / 1.1范围请求 **RFC 7233**，第4.4节
417	EXPECTATION_FAILED	HTTP / 1.1 **RFC 7231**，第6.5.14节
422	UNPROCESSABLE_ENTITY	WebDAV的 **RFC 4918**，第11.2节
423	LOCKED	WebDAV的 **RFC 4918**，第11.3节
424	FAILED_DEPENDENCY	WebDAV的 **RFC 4918**，第11.4节
426	UPGRADE_REQUIRED	HTTP / 1.1 **RFC 7231**，第6.5.15节
428	PRECONDITION_REQUIRED	其他HTTP状态码 **RFC 6585**
429	TOO_MANY_REQUESTS	其他HTTP状态码 **RFC 6585**
431	REQUEST_HEADER_FIELDS_TOO_LARGE	其他HTTP状态码 **RFC 6585**

500	INTERNAL_SERVER_ERROR	HTTP / 1.1 **RFC 7231**，第6.6.1节
501	NOT_IMPLEMENTED	HTTP / 1.1 **RFC 7231**，第6.6.2节
502	BAD_GATEWAY	HTTP / 1.1 **RFC 7231**，第6.6.3节
503	SERVICE_UNAVAILABLE	HTTP / 1.1 **RFC 7231**，第6.6.4节
504	GATEWAY_TIMEOUT	HTTP / 1.1 **RFC 7231**，第6.6.5节
505	HTTP_VERSION_NOT_SUPPORTED	HTTP / 1.1 **RFC 7231**，第6.6.6节
506	VARIANT_ALSO_NEGOTIATES	HTTP中的透明内容协商 **RFC 2295**，第8.1节（实验）
507	INSUFFICIENT_STORAGE	WebDAV的 **RFC 4918**，第11.5节
508	LOOP_DETECTED	WebDAV绑定扩展 **RFC 5842**，第7.2节（实验）
510	NOT_EXTENDED	一个HTTP扩展框架 **RFC 2774**，第7部分（实验）
511	NETWORK_AUTHENTICATION_REQUIRED	其他HTTP状态码 **RFC 6585**，第6部分

为了保持向后兼容性，枚举值也`http.client`以常量的形式出现在模块中。枚举名称等于常量名称（即`http.HTTPStatus.OK`也可用作为 `http.client.OK`）。

21.12。http.client- HTTP协议客户端

源代码： Lib / http / client.py

该模块定义了实现HTTP和HTTPS协议客户端的类。通常不会直接使用 - 该模块urllib.request使用它来处理使用HTTP和HTTPS的URL。

也可以看看： 该请求包 被推荐用于更高级别的HTTP客户端接口。

注意： HTTPS支持仅在Python支持SSL（通过ssl模块）编译时才可用。

该模块提供以下类别：

class http.client.HTTPConnection（*host*，*port* = *None*，[*timeout*，] *source_address* = *None* ）

一个HTTPConnection实例代表一个HTTP服务器的事务。它应该被实例化，传递一个主机和可选的端口号。如果没有传递端口号，则从主机字符串中提取端口（如果它具有表单）host:port，否则使用默认的HTTP端口（80）。如果给出可选的*timeout*参数，阻塞操作（如连接尝试）将在几秒后超时（如果未给出，则使用全局默认超时设置）。可选的*source_address*参数可以是一个（主机，端口）的元组，作为HTTP连接的源地址。

例如，以下调用都会创建在同一主机和端口连接到服务器的实例：

```
>>> h1 = http.client.HTTPConnection('www.python.org')
>>> h2 = http.client.HTTPConnection('www.python.org:80')
>>> h3 = http.client.HTTPConnection('www.python.org', 80)
>>> h4 = http.client.HTTPConnection('www.python.org', 80, timeout=10)
```

*在版本3.2中更改：*添加了*source_address*。

*改变在3.4版本：*在*严格的*参数被删除。不再支持HTTP 0.9风格的“简单回复”。

class http.client.HTTPSConnection（*host*，*port* = *None*，*key_file* = *None*，*cert_file* = *None*，[*timeout*，] *source_address* = *None*，*，*context* = *None*，*check_hostname* = *None* ）

它的一个子类HTTPConnection使用SSL与安全服务器进行通信。默认端口是443。如果指定了*上下文*，则它必须是ssl.SSLContext描述各种SSL选项的实例。

请阅读安全注意事项以了解有关最佳做法的更多信

*在版本3.2中进行了更改：添加了*source_address，*context*和*check_hostname*。

*在版本3.2中更改：*如果可能，此类现在支持HTTPS虚拟主机（即，如果ssl.HAS_SNI为true）。

*改变在3.4版本：*在*严格的*参数被删除。不再支持HTTP 0.9样式的“简单响应”。

*在版本3.4.3中更改：*此类现在默认执行所有必需的证书和主机名检查。为了恢复到之前未经验证的行为， `ssl._create_unverified_context()` 可以将其传递给*上下文*参数。

自从版本3.6开始弃用： *key_file* 和 *cert_file* 不赞成使用*上下文*。请 `ssl.SSLContext.load_cert_chain()` 改用，或者让我们 `ssl.create_default_context()` 为您选择系统的可信CA证书。

所述*check_hostname*参数也过时; 应该使用*上下文*的 `ssl.SSLContext.check_hostname` 属性。

class `http.client.` HTTPResponse (*sock* , *debuglevel = 0* , *method = None* , *url = None*)

成功连接后返回实例的类。没有直接由用户实例化。

*改变在3.4版本：*在*严格的*参数被删除。不再支持HTTP 0.9样式的“简单响应”。

以下例外情况适当提出：

异常 `http.client.` HTTPException

此模块中其他例外的基类。它是一个子类 `Exception`。

异常 `http.client.` NotConnected

一个子类`HTTPException`。

异常 `http.client.` InvalidURL

`HTTPException`如果给出端口并且是非数字或空的，则产生子类。

异常 `http.client.` UnknownProtocol

一个子类`HTTPException`。

异常 `http.client.` UnknownTransferEncoding

一个子类`HTTPException`。

异常 `http.client.` UnimplementedFileMode

一个子类`HTTPException`。

异常 `http.client.` IncompleteRead

一个子类`HTTPException`。

异常 `http.client.` ImproperConnectionState

一个子类`HTTPException`。

异常 `http.client.` CannotSendRequest

一个子类`ImproperConnectionState`。

异常 `http.client.` CannotSendHeader

一个子类`ImproperConnectionState`。

异常 `http.client.` ResponseNotReady

一个子类ImproperConnectionState。

*异常*http.client.BadStatusLine
一个子类HTTPException。如果服务器响应我们不了解的HTTP状态代码，就会引发此问题。

*异常*http.client.LineTooLong
一个子类HTTPException。如果从服务器收到HTTP协议中的过长行，则会引发此问题。

*异常*http.client.RemoteDisconnected
的子类ConnectionResetError和BadStatusLine。通过提出HTTPConnection.getresponse()当试图读取在没有从连接读取数据的响应的结果，表明该远端已经关闭了连接。

*3.5版本中的新功能：*以前曾被提出过。BadStatusLine('')

这个模块中定义的常量是：

http.client.HTTP_PORT
HTTP协议的默认端口（始终80）。

http.client.HTTPS_PORT
HTTPS协议的默认端口（始终443）。

http.client.responses
该字典将HTTP 1.1状态代码映射到W3C名称。

例如：http.client.responses[http.client.NOT_FOUND]是。'Not Found'

请参阅HTTP状态代码，以获取此模块中可用作常量的HTTP状态代码列表。

21.12.1。HTTPConnection对象

HTTPConnection 实例具有以下方法：

HTTPConnection.request（*method*，*url*，*body* = *None*，*headers* = *{}*，***，*encode_chunked* = *False* ）
这将使用HTTP请求方法*方法*和选择器*url*向服务器发送请求。

如果指定了*body*，则在完成标题后发送指定的数据。它可以是str，一个类字节对象，一个开放的文件对象，或一个可迭代bytes。如果*body* 是一个字符串，则它被编码为ISO-8859-1，这是HTTP的默认值。如果它是一个类似字节的对象，则按原样发送字节。如果它是文件对象，则发送文件的内容; 这个文件对象应该至少支持该read()方法。如果文件对象是一个实例io.TextIOBase，则该read() 方法返回的数据将被编码为ISO-8859-1，否则返回的数据按 read()原样发送。如果*身体* 是一个可迭代的，迭代器的元素按原样发送，直到迭代器耗尽。

该*头*参数应该是额外的HTTP标头的映射与发送请求。

如果*标题*既不包含内容长度也不包含传输编码，但是有一个请求体，其中一个标题字段将自动添加。如果 *体*是None ，在Content-Length头被设置为0对于期望的本体方法（ PUT ，POST ，和PATCH ）。如果 *body*是一个字符串或者不是一个文件的类似字节的对象，则Content-Length头部被设置为它的长度。任何其他类型的*主体*（通常是文件和可迭代的）都将进行块编码，并且Transfer-Encoding标头将自动设置而不是Content-Length。

该*encode_chunked*如果在指定传输编码参数是唯一的相关*头*。如果*encode_chunked*为False ，则HTTPConnection对象假定所有编码均由调用代码处理。如果是True ，则主体将被块编码。

> **注意：** 分块传输编码已添加到HTTP协议版本1.1。除非已知HTTP服务器处理HTTP 1.1，否则调用者必须指定Content-Length，或者必须传递 str不是文件的字节类对象作为主体表示。

*3.2版本中的新功能：*body现在可以是一个可迭代的。

*在版本3.6中更改：*如果*标题*中未设置Content-Length和Transfer-Encoding ，则文件和可迭代*主体*对象现在是块编码的。该*encode_chunked*加入争论。没有尝试确定文件对象的内容长度。

HTTPConnection.**getresponse**( )

在发送请求以获取服务器的响应之后应该调用它。返回一个HTTPResponse实例。

> **注意：** 请注意，您必须先阅读整个响应，然后才能向服务器发送新请求。

*在版本3.5中进行了更改：*如果ConnectionError引发了子类或子类，则在HTTPConnection发送新请求时，该 对象将准备好重新连接。

HTTPConnection.**set_debuglevel**(*level*)

设置调试级别。默认的调试级别是0 ，意味着不打印调试输出。任何大于的值0都会导致所有当前定义的调试输出被打印到标准输出。将debuglevel 被传递到任何新HTTPResponse创建的对象。

版本3.1中的新功能。

HTTPConnection.**set_tunnel**(*host*, *port = None*, *headers = None*)

设置HTTP连接隧道的主机和端口。这允许通过代理服务器运行连接。

主机和端口参数指定隧道连接的端点（即包含在CONNECT请求中的地址，*而不是*代理服务器的地址）。

headers参数应该是通过CONNECT请求发送的额外HTTP头的映射。

例如，要通过在端口8080上本地运行的HTTPS代理服务器进行隧道传输，我们会将代理的地址传递给HTTPSConnection 构造函数，以及我们最终想要访问该set_tunnel()方法的主机地址：

```
>>> import http.client
>>> conn = http.client.HTTPSConnection("localhost", 8080)
```

```
>>> conn.set_tunnel("www.python.org")
>>> conn.request("HEAD","/index.html")
```

3.2版本中的新功能

HTTPConnection.**connect**( )

连接到创建对象时指定的服务器。默认情况下，如果客户端尚未建立连接，则在发出请求时会自动调用此方法。

HTTPConnection.**close**( )

关闭与服务器的连接。

作为使用上述request()方法的替代方法，您还可以通过使用以下四个函数逐步发送您的请求。

HTTPConnection.**putrequest**(*method*，*url*，*skip_host* = *False*，*skip_accept_encoding* = *False*)

这应该是连接到服务器之后的第一个调用。它向包含*方法*字符串，*url*字符串和HTTP版本（HTTP/1.1）的服务器发送一行信息。要禁用自动发送Host:或Accept-Encoding:标题（例如接受附加内容编码），请指定*skip_host*或*skip_accept_encoding* 并使用非False值。

HTTPConnection.**putheader**(*header*，*argument*[，...])

发送 **RFC 822**风格的头部到服务器。它发送一行到服务器，包括标题，冒号和空格以及第一个参数。如果给出更多参数，则会发送连续行，每行包含一个制表符和一个参数。

HTTPConnection.**endheaders**(*message_body* = *None*，*，encode_chunked* = *False*)

向服务器发送一个空行，表示标题的结尾。可选的*message_body*参数可用于传递与请求关联的消息主体。

如果*encode_chunked*是True，则*message_body*的每次迭代的结果 将按照中指定的块编码 **RFC 7230** ，第 3.3.1 节。数据如何编码取决于 *message_body* 的类型 。如果*message_body*实现了缓冲区接口，则编码将导致一个块。如果 *MESSAGE_BODY*是collections.Iterable，每次迭代 *MESSAGE_BODY*将导致一大块。如果 *message_body* 是一个 文件对象，每次调用.read()都会导致一个块。该方法在 *message_body*之后立即自动用信号通知块编码数据的结束。

> **注意：** 由于分块编码规范，由块编码器忽略由迭代器体产生的空块。这是为了避免由于编码格式不正确而导致目标服务器读取请求的提前终止。

*3.6版新功能：分*块编码支持。该*encode_chunked*加入参数。

HTTPConnection.**send**(*数据*)

将数据发送到服务器。只有在endheaders()方法被调用之后并且被调用之前，才应该直接使用 getresponse()它。

21.12.2。HTTPResponse对象

一个HTTPResponse实例包装来自服务器的HTTP响应。它提供对请求标题和实体主体的访问。响应是一个可迭代的对象，可以在with语句中使用。

*改变在3.5版本：*该io.BufferedIOBase接口现在已经实现和所有的读者操作的支持。

HTTPResponse.read（[*amt*]）
读取并返回响应主体，或直到下一个*AMT*字节。

HTTPResponse.readinto（*b*）
将响应主体的下一个len（b）个字节读入缓冲区*b*。返回读取的字节数。

3.3版本的新功能

HTTPResponse.getheader（*名称*，*默认=无*）
返回标题*名称*的值，如果没有标题匹配*名称*，则返回*默认值*。如果名称*名称*不止一个标题，则返回由'，'连接的所有值。如果'default'是除了单个字符串之外的任何可迭代的元素，则其元素类似地以逗号连接返回。

HTTPResponse.getheaders（）
返回（标题，值）元组列表。

HTTPResponse.fileno（）
返回fileno底层套接字。

HTTPResponse.msg
甲http.client.HTTPMessage包含响应标头实例。 http.client.HTTPMessage是的一个子类email.message.Message。

HTTPResponse.version
服务器使用的HTTP协议版本。HTTP / 1.0为10，HTTP / 1.1为11。

HTTPResponse.status
服务器返回的状态码。

HTTPResponse.reason
服务器返回的原因短语。

HTTPResponse.debuglevel
一个调试钩子。如果debuglevel大于零，则在响应被读取和解析时，消息将被打印到标准输出。

HTTPResponse.closed
是True，如果流被关闭。

21.12.3。示例

以下是使用该GET方法的示例会话：

```
>>> import http.client
>>> conn = http.client.HTTPSConnection("www.python.org")
>>> conn.request("GET", "/")
>>> r1 = conn.getresponse()
>>> print(r1.status, r1.reason)
200 OK
>>> data1 = r1.read()  # This will return entire content.
>>> # The following example demonstrates reading data in chunks.
>>> conn.request("GET", "/")
>>> r1 = conn.getresponse()
>>> while not r1.closed:
...     print(r1.read(200))  # 200 bytes
b'<!doctype html>\n<!--[if"...
...
>>> # Example of an invalid request
>>> conn.request("GET", "/parrot.spam")
>>> r2 = conn.getresponse()
>>> print(r2.status, r2.reason)
404 Not Found
>>> data2 = r2.read()
>>> conn.close()
```

以下是使用该HEAD方法的示例会话。请注意，该 HEAD方法从不返回任何数据。

```
>>> import http.client
>>> conn = http.client.HTTPSConnection("www.python.org")
>>> conn.request("HEAD", "/")
>>> res = conn.getresponse()
>>> print(res.status, res.reason)
200 OK
>>> data = res.read()
>>> print(len(data))
0
>>> data == b''
True
```

以下是一个示例会话，显示如何POST请求：

```
>>> import http.client, urllib.parse
>>> params = urllib.parse.urlencode({'@number': 12524, '@type': 'issue', '@action': '
>>> headers = {"Content-type": "application/x-www-form-urlencoded",
...            "Accept": "text/plain"}
>>> conn = http.client.HTTPConnection("bugs.python.org")
>>> conn.request("POST", "", params, headers)
>>> response = conn.getresponse()
>>> print(response.status, response.reason)
302 Found
>>> data = response.read()
>>> data
b'Redirecting to <a href="http://bugs.python.org/issue12524">http://bugs.python.org/is
>>> conn.close()
```

客户端请求与请求非常相似。区别仅在于HTTP服务器允许通过请求创建资源的服务器端。应该注意的是，自定义HTTP方法+也通过发送适当的+方法属性来处理。这里是一个示例会话，它

显示了如何 使用http.client 执行请求： HTTP PUT POST PUT urllib.request.Request PUT

```
>>> # This creates an HTTP message
>>> # with the content of BODY as the enclosed representation
>>> # for the resource http://localhost:8080/file
...
>>> import http.client
>>> BODY = "***filecontents***"
>>> conn = http.client.HTTPConnection("localhost", 8080)
>>> conn.request("PUT", "/file", BODY)
>>> response = conn.getresponse()
>>> print(response.status, response.reason)
200, OK
```

21.12.4。HTTPMessage对象

一个 http.client.HTTPMessage 实例包含来自 HTTP 响应的头文件。它使用 email.message.Message该类来实现。

21.13。ftplib- FTP协议客户端

源代码： Lib / ftplib.py

这个模块定义了类FTP和一些相关的项目。的 FTP类实现FTP协议的客户端。您可以使用它来编写执行各种自动FTP作业的Python程序，例如镜像其他FTP服务器。它也被模块urllib.request用来处理使用FTP的URL。有关FTP（文件传输协议）的更多信息，请参阅InternetRFC 959。

以下是使用该ftplib模块的示例会话：

```
>>> from ftplib import FTP
>>> ftp = FTP('ftp.debian.org')     # connect to host, default port
>>> ftp.login()                     # user anonymous, passwd anonymous@
'230 Login successful.'
>>> ftp.cwd('debian')               # change into "debian" directory
>>> ftp.retrlines('LIST')           # list directory contents
-rw-rw-r--    1 1176     1176         1063 Jun 15 10:18 README
...
drwxr-sr-x    5 1176     1176         4096 Dec 19  2000 pool
drwxr-sr-x    4 1176     1176         4096 Nov 17  2008 project
drwxr-xr-x    3 1176     1176         4096 Oct 10  2012 tools
'226 Directory send OK.'
>>> ftp.retrbinary('RETR README', open('README', 'wb').write)
'226 Transfer complete.'
>>> ftp.quit()
```

该模块定义了以下项目：

class ftplib.FTP（*host =''*，*user =''*，*passwd =''*，*acct =''*，*timeout = None*，*source_address = None* ）

返回FTP该类的新实例。当给出*主机时*，进行方法调用connect(host)。当给出*用户时*，另外进行方法调用（其中*passwd*和 *acct*在未给出时默认为空字符串）。可选的*timeout* 参数指定阻止诸如连接尝试之类的操作的超时时间（以秒为单位）（如果未指定，则将使用全局默认超时设置）。 *source_address*是套接字绑定到连接前的源地址的2元组。

login(user, passwd, acct)(host, port)

本FTP类支持with的语句，如：

```
>>> from ftplib import FTP
>>> with FTP("ftp1.at.proftpd.org") as ftp:
...     ftp.login()
...     ftp.dir()
...
'230 Anonymous login ok, restrictions apply.'
dr-xr-xr-x   9 ftp      ftp           154 May  6 10:43 .
dr-xr-xr-x   9 ftp      ftp           154 May  6 10:43 ..
dr-xr-xr-x   5 ftp      ftp          4096 May  6 10:43 CentOS
dr-xr-xr-x   3 ftp      ftp            18 Jul 10  2008 Fedora
>>>
```

在版本3.2中进行了更改： with添加了对语句的支持。

在版本3.3中更改： 添加了*source_address*参数。

class ftplib.FTP_TLS（*host =''*，*user =''*，*passwd =''*，*acct =''*，*keyfile = None*，*certfile = None*，*context = None*，*timeout = None*，*source_address = None* ）

阿FTP如描述这增加了TLS支持FTP子类 **RFC 4217**。照常连接到端口21，在认证之前隐式保护FTP控制连接。保护数据连接需要用户通过调用prot_p()方法明确要求它。 *上下文*是一个ssl.SSLContext允许将SSL配置选项，证书和私钥绑定到一个（可能长寿命）的结构中的对象。请阅读最佳实践的安全考虑事项。

*keyfile*和*certfile*是*上下文*的传统替代方案- 它们可以指向PEM格式的私钥和证书链文件（分别为SSL连接）。

3.2版本中的新功能

在版本3.3中更改： 添加了*source_address*参数。

在3.4版本中更改： 类现在支持主机名检查 ssl.SSLContext.check_hostname和*服务器名称指示*（见 ssl.HAS_SNI ）。

自版本3.6起弃用： *keyfile* 和 *certfile* 不赞成使用*上下文*。请ssl.SSLContext.load_cert_chain()改用，或者让我们 ssl.create_default_context()为您选择系统的可信CA证书。

以下是使用FTP_TLS该类的示例会话：

```
>>> ftps = FTP_TLS('ftp.pureftpd.org')
>>> ftps.login()
'230 Anonymous user logged in'
>>> ftps.prot_p()
'200 Data protection level set to "private"'
>>> ftps.nlst()
['6jack', 'OpenBSD', 'antilink', 'blogbench', 'bsdcam', 'clockspeed', 'djbdns-jedi
```

*异常*ftplib.error_reply

从服务器收到意外回复时引发异常。

*异常*ftplib.error_temp

当接收到表示暂时错误的错误代码（400-499范围内的响应代码）时引发异常。

*异常*ftplib.error_perm

当接收到表示永久性错误的错误代码（响应代码在500-599范围内）时收到异常。

*异常*ftplib.error_proto

从服务器接收到不符合文件传输协议响应规范的回复时引发异常，即从范围为1-5的数字开始。

ftplib.all_errors

由于FTP FTP连接的问题（与调用者编程错误相对），实例方法可能引发的所有异常（作为元组）的集合。这组包括上面列出的四个例外以及OSError。

也可以看看:

模 netrc
解析器的.netrc文件格式。该文件.netrc通常由FTP客户端在提示用户之前加载用户认证信息。

21.13.1。FTP对象

有两种方法可用，一种用于处理文本文件，另一种用于处理二进制文件。这些命令用于lines文本版本或binary二进制版本所使用的命令。

FTP 实例具有以下方法：

FTP. set_debuglevel（*level*）
设置实例的调试级别。这将控制打印的调试输出量。缺省值，0不会产生调试输出。一个值1会产生适量的调试输出，通常每个请求只有一行。值2或更高值会产生最大量的调试输出，记录控制连接上发送和接收的每条线路。

FTP. connect（*host =''*，*port = 0*，*timeout = None*，*source_address = None*）
连接到给定的主机和端口。默认端口号是21由FTP协议规范指定的。很少需要指定不同的端口号。每个实例只能调用一次该函数; 如果在创建实例时提供主机，则根本不应该调用它。所有其他方法只能在连接完成后使用。可选的*timeout*参数指定连接尝试的超时时间（以秒为单位）。如果没有*超时*通过，则将使用全局默认超时设置。 *source_address*是套接字绑定到连接前的源地址的2元组。（host，port）

*在版本3.3中更改：*添加了*source_address*参数。

FTP. getwelcome（）
返回服务器发送的回复初始连接的欢迎消息。（此消息有时包含可能与用户有关的免责声明或帮助信息。）

FTP. login（*user ='anonymous'*，*passwd =''*，*acct =''*）
以给定的*用户*身份登录。将*passwd文件*和*ACCT*参数是可选的，默认为空字符串。如果没有指定*用户*，则默认为'anonymous'。如果*用户*是'anonymous'，则默认*密码*是'anonymous@'。连接建立后，每个实例只能调用一次该函数; 如果在创建实例时提供主机和用户，则根本不应该调用它。大多数FTP命令仅在客户端登录后才被允许。*acct*参数提供“会计信息”; 很少有系统实施这个。

FTP. abort（）
中止正在进行的文件传输。使用它并不总是奏效，但值得一试。

FTP. sendcmd（*cmd*）
发送一个简单的命令字符串到服务器并返回响应字符串。

FTP.voidcmd(*cmd*)

发送一个简单的命令字符串到服务器并处理响应。如果接收到与成功相对应的响应码（范围在200-299范围内），则不返回任何内容。提高error_reply其他。

FTP.retrbinary(*cmd*, *callback*, *blocksize = 8192*, *rest = None*)

以二进制传输模式检索文件。 *cmd*应该是一个合适的 RETR命令：。该*回调*函数被调用用于接收，与单个字节参数给出数据块的每个数据块。可选的*blocksize*参数指定为了进行实际传输而创建的低级套接字对象（它也将传递给*回调*的数据块的最大大小）读取的最大块大小。选择合理的默认值。*休息*意味着与该方法中的相同。'RETR filename'transfercmd()

FTP.retrlines(*cmd*, *callback = None*)

以ASCII传输模式检索文件或目录列表。 *CMD*应适当RETR命令（见retrbinary()）或一个命令，例如 LIST或NLST（通常只是字符串'LIST'）。 LIST检索文件列表和关于这些文件的信息。 NLST检索文件名列表。该*回调*函数被调用用含有与剥离尾部CRLF行字符串参数的每一行。默认*回调* 打印该行sys.stdout。

FTP.set_pasv(*val*)

如果*val*为true，则启用"被动"模式，否则禁用被动模式。被动模式默认打开。

FTP.storbinary(*cmd*, *fp*, *blocksize = 8192*, *callback = None*, *rest = None*)

以二进制传输模式存储文件。 *cmd*应该是一个合适的 STOR命令：。*fp*是一个文件对象（以二进制模式打开），使用它的 方法以块大小*块的*形式读取直到EOF ，以提供要存储的数据。该*块大小*参数默认为8192。 *回调*是一个可选的单参数调用它被发送后调用数据的每块上。 *休息*意味着与该方法中的相同。"STOR filename"read()transfercmd()

*在版本3.2中更改：*添加了*rest*参数。

FTP.storlines(*cmd*, *fp*, *callback = None*)

以ASCII传输模式存储文件。 *cmd*应该是一个合适的 STOR命令（参见参考资料storbinary()）。从文件对象 *fp*（以二进制模式打开）使用其readline() 方法提供要存储的数据读取直到EOF 。 *回调*函数是一个可选的单个参数，在每一行发送后都会调用它。

FTP.transfercmd(*cmd*, *rest = None*)

通过数据连接启动传输。如果传输是有效的，发送一个 EPRT或 PORT命令，并通过指定的传输命令*CMD*，并接受该连接。如果服务器是被动的，则发送EPSV或PASV 命令，连接到它并启动传输命令。无论哪种方式，返回连接的套接字。

如果给出可选*休息*，则将REST命令发送到服务器，并将*休息*作为参数传递 。 *休息*通常是一个字节偏移到请求的文件，告诉服务器重新启动发送文件的字节在请求的偏移量，跳过最初的字节。但请注意，RFC 959仅要求 *其余*字符包含ASCII码33到ASCII码126的可打印范围内的字符transfercmd()。因此，该方法将*rest*转换 为字符串，但不对字符串的内容执行检查。如果服务器不能识别该REST命令，error_reply则会引发异常。如果发生这种情况，只需拨打transfercmd()没有 *休息* 论据。

FTP.ntransfercmd(*cmd*, *rest = None*)

喜欢transfercmd() ，但返回数据连接的元组和数据的预期大小。如果无法计算预期的大小，None 将按预期的大小返回。 *cmd*和*休息*意味着与在中相同的东西transfercmd()。

FTP.mlsd(*path=""*, *facts=[]*)

使用MLSD命令以标准格式列出目录（**RFC 3659**）。如果省略了*路径*，则假定当前目录。*事实*是表示所需信息类型的字符串列表（例如）。返回一个生成器对象，为每个在路径中找到的文件生成一个包含两个元素的元组。第一个元素是文件名，第二个是包含文件名相关信息的字典。这本词典的内容可能会受到*事实*参数的限制，但服务器不能保证返回所有请求的事实。`["type", "size", "perm"]`

3.3版本的新功能

FTP.nlst(*argument*[, ...])

返回该NLST命令返回的文件名列表。可选*参数*是要列出的目录（默认为当前服务器目录）。可以使用多个参数将非标准选项传递给NLST命令。

> **注意：** 如果您的服务器支持该命令，则mlsd()可以提供更好的API。

FTP.dir(*argument*[, ...])

生成该LIST命令返回的目录列表，将其打印到标准输出。可选*参数*是要列出的目录（默认为当前服务器目录）。可以使用多个参数将非标准选项传递给LIST命令。如果最后一个参数是一个函数，它用作*回调*函数retrlines()；默认打印到 sys.stdout。此方法返回None。

> **注意：** 如果您的服务器支持该命令，则mlsd()可以提供更好的API。

FTP.rename(*fromname*, *toname*)

将服务器上的文件名*fromname*重命名为*toname*。

FTP.delete(*文件名*)

从服务器中删除名为*filename*的文件。如果成功，则返回响应的文本，否则会引发error_perm权限错误或 error_reply其他错误。

FTP.cwd(*路径名*)

设置服务器上的当前目录。

FTP.mkd(*路径名*)

在服务器上创建一个新目录。

FTP.pwd()

返回服务器上当前目录的路径名。

FTP.rmd(*dirname*)

删除服务器上名为*dirname*的目录。

FTP.size(*文件名*)

请求服务器上名为*filename*的文件的大小。成功时，文件的大小以整数None形式返回，否则返回。请注意，该SIZE命令不是标准化的，但受许多常用服务器实现的支持。

FTP.quit()

发送QUIT命令到服务器并关闭连接。这是关闭连接的“礼貌”方式，但如果服务器对QUIT命令作出错误响应，则可能会引发异常。这意味着调用close()方法，该 方法将使FTP实例无用于后续调用（请参见下文）。

FTP.close()
单方面关闭连接。这不应该应用于已经关闭的连接，例如成功调用之后quit()。在这个调用之后，FTP实例不应该再被使用（在调用之后close()或者quit()你不能通过发布另一种login()方法重新打开连接）。

21.13.2。FTP_TLS对象

FTP_TLS类继承FTP，定义这些额外的对象：

FTP_TLS.ssl_version
要使用的SSL版本（默认为ssl.PROTOCOL_SSLv23）。

FTP_TLS.auth()
根据ssl_version属性中指定的内容，使用TLS或SSL设置安全的控制连接。

*在3.4版本中更改：*该方法现在支持主机名检查 ssl.SSLContext.check_hostname和*服务器名称指示*（见 ssl.HAS_SNI）。

FTP_TLS.ccc()
将控制通道恢复为纯文本。这对利用知道如何在不打开固定端口的情况下使用非安全FTP处理NAT的防火墙非常有用。

3.3版本的新功能

FTP_TLS.prot_p()
建立安全的数据连接。

FTP_TLS.prot_c()
设置明文数据连接。

21.14。poplib- POP3协议的客户端

源代码： Lib / poplib.py

这个模块定义了一个类，POP3它封装了一个到POP3服务器的连接，并实现了协议的定义**RFC 1939**。该 POP3级支持从最小的和可选的命令集 **RFC 1939**。该POP3班还支持STLS中介绍的命令**RFC 2595**在已建立的连接上启用加密通信。

另外，这个模块提供了一个类POP3_SSL，它提供了对连接到使用SSL作为底层协议层的POP3服务器的支持。

请注意，POP3尽管得到广泛支持，但已经过时。POP3服务器的实现质量差异很大，而且太多的差劲。如果您的邮件服务器支持IMAP，那么最好使用 imaplib.IMAP4该类，因为IMAP服务器往往会更好地实现。

该poplib模块提供了两个类：

*类*poplib.POP3（*主机*，*端口*=*POP3_PORT*[，*超时*]）

这个类实现了实际的POP3协议。连接在实例初始化时创建。如果省略*端口*，则使用标准的POP3端口（110）。可选的*timeout*参数指定连接尝试的超时（以秒为单位）（如果未指定，则将使用全局默认超时设置）。

class poplib.POP3_SSL（*host*，*port* = *POP3_SSL_PORT*，*keyfile* = *None*，*certfile* = *None*，*timeout* = *None*，*context* = *None*）

这是POP3通过SSL加密套接字连接到服务器的子类。如果未指定*端口* 995，则使用标准的POP3-over-SSL端口。 *超时*工作在POP3构造函数中。 *上下文*是一个可选ssl.SSLContext对象，允许将SSL配置选项，证书和私钥捆绑到一个（可能长寿命）的结构中。请阅读 最佳实践的安全考虑事项。

*keyfile*和*certfile*是*上下文*的传统替代方案- 它们可以分别指向PEM格式的私钥和证书链文件，以用于SSL连接。

*在版本3.2中更改：*添加了*上下文*参数。

*在3.4版本中更改：*类现在支持主机名检查 ssl.SSLContext.check_hostname和*服务器名称指示*（见 ssl.HAS_SNI）。

自版本3.6起弃用： *keyfile* 和 *certfile* 不赞成使用*上下文*。请ssl.SSLContext.load_cert_chain()改用，或者让我们 ssl.create_default_context()为您选择系统的可信CA证书。

一个例外被定义为poplib模块的属性：

*异常*poplib.error_proto

从该模块发出的任何错误都会引发异常（模块发生的错误socket 不会被捕获）。异常的原因以字符串形式传递给构造函数。

也可以看看:

模 imaplib
标准的Python IMAP模块。

有关Fetchmail的常见问题
如果您需要编写基于POP协议的应用程序，则**fetchmail** POP / IMAP客户端的常见问题可收集有关POP3服务器变体和RFC不合规性的信息，这些信息可能很有用。

21.14.1。POP3对象

所有的POP3命令都用同名的方法表示，小写; 大多数返回由服务器发送的响应文本。

一个POP3实例有以下方法：

POP3.set_debuglevel(*level*)
设置实例的调试级别。这将控制打印的调试输出量。缺省值，0不会产生调试输出。一个值1会产生适量的调试输出，通常每个请求只有一行。值2或更高值会产生最大量的调试输出，记录控制连接上发送和接收的每条线路。

POP3.getwelcome()
返回由POP3服务器发送的问候字符串。

POP3.capa()
按照中的指定查询服务器的功能 **RFC 2449**。返回表单中的字典。{'name': ['param'...]}

3.4版新增功能

POP3.user(*用户名*)
发送用户命令，响应应该指示需要密码。

POP3.pass_(*密码*)
发送密码，回复包括邮件数量和邮箱大小。注意：服务器上的邮箱被锁定直到quit()被调用。

POP3.apop(*用户*, *秘密*)
使用更安全的APOP认证登录到POP3服务器。

POP3.rpop(*用户*)
使用RPOP身份验证（类似于UNIX r命令）登录到POP3服务器。

POP3.stat()
获取邮箱状态。结果是一个2个整数的元组：。(message count, mailbox size)

POP3.list([*which*])
请求消息列表，结果在表格中。如果*它*被设置，它是列出的消息。(response, ['mesg_num octets', ...], octets)

POP3.retr(*哪*)

检索整个消息号，*其*，并设置其可见标志。结果在表格中。(response, ['line', ...], octets)

POP3.dele(*哪*)

标志的消息号，*其*为删除。在大多数服务器上，直到QUIT（主要的例外是Eudora QPOP，它故意通过在任何断开连接上执行挂起删除操作时故意违反RFC），删除操作才会实际执行。

POP3.rset()

删除邮箱的任何删除标记。

POP3.noop()

没做什么。可能被用作保持活力。

POP3.quit()

签署：提交更改，解锁邮箱，删除连接。

POP3.top(*其中*, *多少*)

检索消息首部加上*howmuch*的消息号的报头之后的消息的线*哪个*。结果在表格中。(response, ['line', ...], octets)

与RETR命令不同，此方法使用的POP3 TOP命令不会设置消息的可见标志; 不幸的是，TOP在RFC中没有详细说明，并且在非品牌服务器中经常被破坏。在信任它之前，手动测试您将使用的POP3服务器。

POP3.uidl(*which* = *None*)

返回消息摘要（唯一标识符）列表。如果指定了*哪个*，则结果在表单中包含该消息的唯一标识，否则结果为列表。'response mesgnum uid(response, ['mesgnum uid', ...], octets)

POP3.utf8()

尝试切换到UTF-8模式。如果成功则返回服务器响应，否则返回error_proto。指定在**RFC 6856**。

3.5版本中的新功能。

POP3.stls(*context* = *None*)

按照中所述在活动连接上启动TLS会话 **RFC 2595**。这仅在用户认证之前被允许

*上下文*参数是一个ssl.SSLContext允许将SSL配置选项，证书和私钥绑定到单个（可能长寿命）的结构中的对象。请阅读 最佳实践的安全考虑事项。

此方法支持通过ssl.SSLContext.check_hostname和*服务器名称指示进行*主机名检查（请参阅 ssl.HAS_SNI）。

3.4版新增功能

POP3_SSL没有其他方法的实例。此子类的接口与其父类相同。

21.14.2。POP3示例

这是一个最小的例子（没有错误检查），打开邮箱并检索并打印所有消息：

```
import getpass, poplib

M = poplib.POP3('localhost')
M.user(getpass.getuser())
M.pass_(getpass.getpass())
numMessages = len(M.list()[1])
for i in range(numMessages):
    for j in M.retr(i+1)[1]:
        print(j)
```

在模块的最后，有一个测试部分，其中包含更广泛的使用示例。

21.15。 imaplib- IMAP4协议客户端

源代码： Lib / imaplib.py

该模块定义了三个类， IMAP4 ， IMAP4_SSL和 IMAP4_stream ，其中封装到IMAP4服务器的连接，并执行如在所限定的IMAP4rev1客户端协议的一个大的子集 **RFC 2060** 。 它向后兼容IMAP4（ **RFC 1730** ）服务器，但请注意，该STATUS命令在IMAP4中不受支持。

imaplib模块提供了三个类， IMAP4它是基类：

class imaplib. IMAP4 （ *host =''* , *port = IMAP4_PORT* ）

该类实现了实际的IMAP4协议。创建连接并在实例初始化时确定协议版本（ IMAP4或IMAP4rev1 ）。如果*主机*没有指定，''（本地主机）被使用。如果 省略*端口*，则使用标准IMAP4端口（ 143 ）。

该IMAP4课程支持该with声明。当像这样使用LOGOUT时，当with语句退出时，IMAP4 命令会自动发出 。例如：

```
>>> from imaplib import IMAP4
>>> with IMAP4("domain.org") as M:
...     M.noop()
...
('OK', [b'Nothing Accomplished. d25if65hy903weo.87'])
```

*在版本3.5中*进行了*更改*：with添加了对语句的支持。

三个例外被定义为IMAP4该类的属性：

*异常*IMAP4. error

任何错误都会引发异常。异常的原因以字符串形式传递给构造函数。

*异常*IMAP4. abort

IMAP4服务器错误导致引发此异常。这是一个子类 IMAP4.error。请注意，关闭实例并实例化一个新实例通常会允许从此异常中恢复。

*异常*IMAP4. readonly

当可写邮箱的状态由服务器更改时引发此异常。这是一个子类IMAP4.error。某些其他客户端现在具有写入权限，并且需要重新打开邮箱才能重新获得写入权限。

还有一个安全连接的子类：

class imaplib. IMAP4_SSL （ *host =''* , *port = IMAP4_SSL_PORT* , *keyfile = None* , *certfile = None* , *ssl_context = None* ）

这是IMAP4通过SSL加密套接字连接而来的子类（要使用此类，您需要一个使用SSL支持编译的套接字模块）。如果*主机*没有指定，''（本地主机）被使用。如果省略*端口*，则使用标准的IMAP4-SSL端口（ 993 ）。 *ssl_context*是一个ssl.SSLContext允许将SSL配置选

项，证书和私钥绑定到单个（可能长寿命）的结构中的对象。请阅读最佳实践的安全考虑事项。

*密钥文件*和*certFile*中是一个传统的替代*ssl_context* -他们可以指向该SSL连接PEM格式的私钥和证书链文件。请注意，*keyfile* / *certfile*参数与*ssl_context*互斥，ValueError如果*keyfile* / *certfile*与*ssl_context*一起提供，*则会引发*a。

*版本3.3中更改：*添加了*ssl_context*参数。

*在3.4版本中更改：*类现在支持主机名检查 ssl.SSLContext.check_hostname和*服务器名称指示*（见 ssl.HAS_SNI ）。

自从版本3.6开始弃用： *keyfile* 和 *certfile* 不推荐使用 *ssl_context* 。 请 ssl.SSLContext.load_cert_chain()改用，或者让我们 ssl.create_default_context()为您选择系统的可信CA证书。

第二个子类允许由子进程创建连接：

class imaplib.IMAP4_stream（ *command* ）

这是衍生自一个子类IMAP4 ，其连接到 stdin/stdout通过将创建的文件描述符*命*令到subprocess.Popen()。

定义了以下效用函数：

imaplib.Internaldate2tuple（ *datestr* ）

解析IMAP4 INTERNALDATE字符串并返回相应的本地时间。返回值是一个time.struct_time元组，或者 None如果字符串格式错误。

imaplib.Int2AP（ *num* ）

使用set [A.. P]中的字符将整数转换为字符串表示形式。

imaplib.ParseFlags（ *flagstr* ）

将IMAP4 FLAGS响应转换为单个标志的元组。

imaplib.Time2Internaldate（ *date_time* ）

将*date_time*转换为IMAP4 INTERNALDATE表示。返回值是以下形式的字符串:（包括双引号）。所述*DATE_TIME*参数可以是数（整数或浮点数）表示纪元以来秒（通过与返回的），一个9元组表示的本地时间的一个实例（如通过返回 ），的感知实例 ，或一个双引号字符串。在最后一种情况下，假定它已经是正确的格式。 "DD-Mmm-YYYY HH:MM:SS +HHMM"time.time()time.struct_timetime.localtime()datetime.datetime

请注意，IMAP4邮件号码随邮箱更改而变化; 特别是在EXPUNGE命令执行删除操作后，剩余的消息将被重新编号。因此，使用UID命令来代替使用UID是非常明智的。

在模块的最后，有一个测试部分，其中包含更广泛的使用示例。

也可以看看： 描述协议的文件，以及实施它的服务器的源代码和二进制文件均可在华盛顿大学*IMAP信息中心*（ https://www.washington.edu/imap/ ）找到。

21.15.1。IMAP4对象

所有IMAP4rev1命令都由相同名称的方法表示，无论是大写还是小写。

所有命令的参数都被转换为字符串，除了AUTHENTICATE，以及最后一个参数APPEND作为IMAP4文字传递。如有必要（该字符串包含IMAP4协议敏感字符，不包含括号或双引号），每个字符串都被引用。但是，该命令的*密码*参数LOGIN始终是引用的。如果你想避免引用一个参数字符串（例如：*flags*参数 STORE），那么将该字符串括在括号内（例如：）r'(\Deleted)'。

每个命令都会返回一个元组：其中*type*通常是 或，而*data*是来自命令响应的文本，或者来自命令的命令结果。每个*数据*都是一个字符串或一个元组。如果是一个元组，那么第一部分是响应的头部，第二部分包含数据（即'literal'值）。(type, [data, ...])'OK''NO'

以下命令的*message_set*选项是一个字符串，指定要对其执行操作的一条或多条消息。它可以是简单的消息号（'1'），消息号的范围（'2:4'），也可以是由逗号（'1:3,6:9'）分隔的一组非连续的范围。范围可以包含一个星号来表示无限上限（'3:*'）。

一个IMAP4实例有以下方法：

IMAP4.append（*邮箱*，*国旗*，*DATE_TIME*，*消息*）
将*邮件*追加到指定的邮箱。

IMAP4.authenticate（*机制*，*authobject*）
验证命令 - 需要响应处理。

*机制*指定要使用哪种认证机制 - 它应该出现在capabilities表单中的实例变量中AUTH=mechanism。

*authobject*必须是可调用的对象：

```
data = authobject(response)
```

它将被调用来处理服务器连续响应; 它传递的*响应*参数将是bytes。它应该返回bytes *的数据* 将被base64编码并发送到服务器。None如果*应该发送客户端中止响应，它应该返回 。

*在版本3.5中进行了更改：*字符串用户名和密码现在被编码为utf-8而不是被限制为ASCII。

IMAP4.check（）
服务器上的检查点邮箱。

IMAP4.close（）
关闭当前选定的邮箱。已删除的邮件将从可写邮箱中删除。这是以前推荐的命令LOGOUT。

IMAP4.copy（*message_set*，*new_mailbox*）
将*message_set*消息复制到*new_mailbox的末尾*。

IMAP4.create（*邮箱*）
创建名为*mailbox的新邮箱*。

IMAP4.delete（*邮箱*）

删除旧邮箱命名*邮箱*。

IMAP4.deleteacl（*邮箱*，*谁*）

删除为邮箱上的用户设置的ACL（删除所有权限）。

IMAP4.enable（*能力*）

启用*功能*（请参阅**RFC 5161**）。大多数功能不需要启用。目前仅UTF8=ACCEPT支持该功能（请参阅**RFC 6855**）。

*新的3.5版：*该enable()方法本身，**RFC 6855**支持。

IMAP4.expunge（）

永久删除所选邮箱中的已删除邮件。EXPUNGE 为每个已删除的消息生成一个响应。返回的数据包含EXPUNGE 收到订单的消息编号列表。

IMAP4.fetch（*message_set*，*message_parts* ）

获取（部分）消息。 *message_parts*应该是括在括号内的消息部分名称的字符串，例如：。返回的数据是消息部分包络和数据的元组。"(UID BODY[TEXT])"

IMAP4.getacl（*邮箱*）

获取*邮箱*的ACLs 。该方法是非标准的，但由服务器支持。Cyrus

IMAP4.getannotation（*邮箱*，*条目*，*属性*）

检索*邮箱*的指定ANNOTATIONs 。该方法是非标准的，但由服务器支持。Cyrus

IMAP4.getquota（*root* ）

获取quota *root*的资源使用情况和限制。此方法是rfc2087中定义的IMAP4 QUOTA扩展的一部分。

IMAP4.getquotaroot（*邮箱*）

获取quota roots指定*邮箱*的列表。此方法是rfc2087中定义的IMAP4 QUOTA扩展的一部分。

IMAP4.list（[*directory*[，*pattern*]] ）

列出*目录*匹配*模式中的*邮箱名称。 *目录*默认为顶层邮件文件夹，并且*模式*默认为匹配任何内容。返回的数据包含LIST响应列表。

IMAP4.login（*用户*，*密码*）

使用明文密码识别客户端。该*密码*将被引用。

IMAP4.login_cram_md5（*用户*，*密码*）

CRAM-MD5识别客户端以保护密码时强制使用身份验证。只有当服务器CAPABILITY响应包含该短语时才会起作用AUTH=CRAM-MD5。

IMAP4.logout（）

关闭连接到服务器。返回服务器BYE响应。

IMAP4.lsub（*directory*='""'，*pattern*='*'）

以目录匹配模式列出订阅的邮箱名称。*目录*默认为顶级目录，并且*模式*默认为匹配任何邮箱。返回的数据是消息部分包络和数据的元组。

IMAP4.myrights（*邮箱*）

显示我的邮箱ACL（即我拥有的邮箱权限）。

IMAP4.namespace（）

返回RFC2342中定义的IMAP名称空间。

IMAP4.noop（）

发送NOOP到服务器。

IMAP4.open（*主机*，*端口*）

在*主机上*打开套接字到*端口*。该方法由构造函数隐式调用。通过这种方法建立的连接对象将在被使用，，，和方法。您可以重写此方法。IMAP4 IMAP4.read() IMAP4.readline() IMAP4.send() IMAP4.shutdown()

IMAP4.partial（*message_num*，*message_part*，*start*，*length*）

获取消息的截断部分。返回的数据是消息部分包络和数据的元组。

IMAP4.proxyauth（*用户*）

假设*用户*身份验证。允许授权管理员代理进入任何用户的邮箱。

IMAP4.read（*size*）

从远程服务器读取*大小*字节。您可以重写此方法。

IMAP4.readline（）

从远程服务器读取一行。您可以重写此方法。

IMAP4.recent（）

提示服务器进行更新。None如果没有新消息，则返回数据，否则返回值RECENT。

IMAP4.rename（*oldmailbox*，*newmailbox*）

将名为*oldmailbox的*邮箱重命名为*newmailbox*。

IMAP4.response（*代码*）

返回响应*代码的*数据，如果收到，或None。返回给定的代码，而不是通常的类型。

IMAP4.search（*charset*，*criterion*[，...]）

搜索邮箱中的匹配邮件。*字符集*可能是None，在这种情况下，CHARSET将在服务器的请求中指定no。IMAP协议要求至少指定一个标准; 服务器返回错误时将引发异常。如果使用该命令启用了功能，则*字符集*必须是。NoneUTF8=ACCEPTenable()

例：

```
# M is a connected IMAP4 instance...
typ, msgnums = M.search(None, 'FROM', '"LDJ"')

# or:
typ, msgnums = M.search(None, '(FROM "LDJ")')
```

IMAP4.**select**(*mailbox='INBOX'*, *readonly=False*)

选择一个邮箱。返回的数据是*邮箱中*的邮件数（EXISTS响应）。默认*邮箱*是'INBOX'。如果 设置了*只读*标志，则不允许对邮箱进行修改。

IMAP4.**send**(*数据*)

发送data到远程服务器。您可以重写此方法。

IMAP4.**setacl**(*邮箱*, *谁*, *什么*)

设置ACL的*邮箱*。该方法是非标准的，但由Cyrus服务器支持。

IMAP4.**setannotation**(*邮箱*, *条目*, *属性*[, ...])

设置ANNOTATIONS代表*邮箱*。该方法是非标准的，但由Cyrus服务器支持。

IMAP4.**setquota**(*root*, *limits*)

设置quota *根目录*的资源*限制*。此方法是rfc2087中定义的IMAP4 QUOTA扩展的一部分。

IMAP4.**shutdown**( )

关闭连接建立在open。这个方法被隐含地称为IMAP4.logout()。您可以重写此方法。

IMAP4.**socket**( )

返回用于连接到服务器的套接字实例。

IMAP4.**sort**(*sort_criteria*, *charset*, *search_criterion* [, ...])

该sort命令是search对结果进行排序语义的变体。返回的数据包含空格分隔的匹配消息编号列表。

Sort在*search_criterion*参数前有两个参数; *sort_criteria*的加括号列表和搜索*字符集*。请注意，不同的是search，搜索*字符集*参数是强制性的。还有一个对应于命令的方式 对应。该命令首先使用charset参数在邮箱中搜索与给定搜索条件相匹配的邮件，以便在搜索条件中解释字符串。然后它返回匹配消息的数量。uid sort sort uid search search sort

这是一个IMAP4rev1扩展命令。

IMAP4.**starttls**(*ssl_context=None*)

发送一个STARTTLS命令。该*ssl_context*参数是可选的，应该是一个ssl.SSLContext对象。这将启用IMAP连接上的加密。请阅读最佳实践的安全考虑事项。

3.2版本中的新功能

*在3.4版本中更改：*该方法现在支持主机名检查 ssl.SSLContext.check_hostname和*服务器名称指示*（见 ssl.HAS_SNI ）。

IMAP4.**status**(*邮箱*, *名称*)

请求*邮箱的*命名状态条件。

IMAP4.store（*message_set*，*command*，*flag_list*）
更改标记邮箱中邮件的处置。 *命令*由第6.4.6节指定作为“FLAGS”，“+ FLAGS”或“-FLAGS”之一的 **RFC 2060**，可选地带有后缀“.SILENT”。

例如，要在所有消息上设置删除标志：

```
typ, data = M.search(None, 'ALL')
for num in data[0].split():
   M.store(num, '+FLAGS', '\\Deleted')
M.expunge()
```

注意： 创建包含']'的标志（例如：“[test]”）违反 **RFC 3501**（IMAP协议）。但是，imaplib在历史上允许创建此类标签，而流行的IMAP服务器（如Gmail）接受并生成此类标志。有非Python程序也可以创建这样的标签。虽然它是RFC违规，IMAP客户端和服务器应该是严格的，但imaplib仍然允许为了向后兼容的原因创建这些标签，并且从python 3.6开始，如果它们是从服务器发送的，则处理它们，因为这改善了现实世界的兼容性。

IMAP4.subscribe（*邮箱*）
订阅新邮箱。

IMAP4.thread（*threading_algorithm*，*charset*，*search_criterion* [，...]）
该thread命令是search结果的线程语义的变体。返回的数据包含空格分隔的线程成员列表。

线程成员由零个或多个消息编号组成，由空格分隔，表示连续的父代和子代。

线程在*search_criterion*参数前有两个参数; 一个 *threading_algorithm*，以及搜索*字符集*。请注意，不同的是 search，搜索*字符集*参数是强制性的。还有一个 对应于命令的方式对应。该命令首先使用charset参数在邮箱中搜索与给定搜索条件相匹配的邮件，以便在搜索条件中解释字符串。然后它根据指定的线程算法返回匹配的消息。uid threadthreaduid searchsearchthread

这是一个IMAP4rev1扩展命令。

IMAP4.uid（*command*，*arg* [，...]）
使用UID标识的消息执行命令参数，而不是消息编号。返回适合命令的响应。至少必须提供一个参数; 如果没有提供，则服务器将返回错误并引发异常。

IMAP4.unsubscribe（*邮箱*）
取消订阅旧邮箱。

IMAP4.xatom（*name* [，...]）
作为CAPABILITY回应，允许服务器通知简单的扩展命令。

以下属性在以下实例上定义IMAP4：

IMAP4.PROTOCOL_VERSION

CAPABILITY服务器响应中最新支持的协议。

IMAP4.debug
整数值来控制调试输出。初始化值取自模块变量Debug。大于三的值跟踪每条命令。

IMAP4.utf8_enabled
通常False为布尔值，但设置为True如果 enable()为该UTF8=ACCEPT 功能成功发出命令。

3.5版本中的新功能。

21.15.2。IMAP4示例

这是一个最小的例子（没有错误检查），打开邮箱并检索并打印所有消息：

```
import getpass, imaplib

M = imaplib.IMAP4()
M.login(getpass.getuser(), getpass.getpass())
M.select()
typ, data = M.search(None, 'ALL')
for num in data[0].split():
    typ, data = M.fetch(num, '(RFC822)')
    print('Message %s\n%s\n' % (num, data[0][1]))
M.close()
M.logout()
```

21.16。nntplib- NNTP协议客户端

源代码： Lib / nntplib.py

该模块定义了NNTP实现网络新闻传输协议客户端的类。它可以用来实现新闻阅读器或海报，或自动新闻处理器。它与。兼容**RFC 3977** 以及更旧的**RFC 977**和**RFC 2980**。

以下是如何使用它的两个小例子。列出关于新闻组的一些统计信息并打印最近10篇文章的主题：

```
>>> s = nntplib.NNTP('news.gmane.org')
>>> resp, count, first, last, name = s.group('gmane.comp.python.committers')
>>> print('Group', name, 'has', count, 'articles, range', first, 'to', last)
Group gmane.comp.python.committers has 1096 articles, range 1 to 1096
>>> resp, overviews = s.over((last - 9, last))
>>> for id, over in overviews:
...     print(id, nntplib.decode_header(over['subject']))
...
1087 Re: Commit privileges for Łukasz Langa
1088 Re: 3.2 alpha 2 freeze
1089 Re: 3.2 alpha 2 freeze
1090 Re: Commit privileges for Łukasz Langa
1091 Re: Commit privileges for Łukasz Langa
1092 Updated ssh key
1093 Re: Updated ssh key
1094 Re: Updated ssh key
1095 Hello fellow committers!
1096 Re: Hello fellow committers!
>>> s.quit()
'205 Bye!'
```

要从二进制文件发布文章（假设文章有有效的标题，并且您有权在特定新闻组上发布文章）：

```
>>> s = nntplib.NNTP('news.gmane.org')
>>> f = open('article.txt', 'rb')
>>> s.post(f)
'240 Article posted successfully.'
>>> s.quit()
'205 Bye!'
```

模块本身定义了以下类：

class nntplib.NNTP（*host*，*port* = *119*，*user* = *None*，*password* = *None*，*readermode* = *None*，*usenetrc* = *False* [，*timeout*] ）

返回一个新NNTP对象，代表与主机*主机*上运行的NNTP服务器的连接，监听端口*端口*。可以为套接字连接指定可选的*超时*值。如果提供了可选的*用户*和*密码*，或者如果存在合适的凭证，/.netrc并且可选标志*usenetrc* 为true，则使用and 命令来标识和验证服务器的用户。如果可选标志*readermode*为true，那么aAUTHINFO USERAUTHINFO PASSmode reader命令在执行认证之前发送。如果您连接到本地计算机上的NNTP服务器并且打算调用特定于阅读器的命令（例如，），则有时需要阅读器模式group。如果你意想不到

NNTPPermanentError的话，你可能需要设置*readermode*。本NNTP类支持with无条件消耗声明OSError例外，完成后关闭NNTP连接，例如：

```
>>> from nntplib import NNTP
>>> with NNTP('news.gmane.org') as n:
...     n.group('gmane.comp.python.committers')
...
('211 1755 1 1755 gmane.comp.python.committers', 1755, 1, 1755, 'gmane.comp.python
>>>
```

*在版本3.2中更改：*usenetrc现在False是默认情况下。

*在版本3.3中进行了更改：*with添加了对语句的支持。

class nntplib.NNTP_SSL（*host*，*port* = *563*，*user* = *None*，*password* = *None*，*ssl_context* = *None*，*readermode* = *None*，*usenetrc* = *False* [，*timeout*] ）

返回一个新NNTP_SSL对象，表示与在主机*主机*上运行的NNTP服务器的加密连接，并在端口*端口*侦听。 NNTP_SSL对象与对象具有相同的方法 NNTP。如果*端口*被省略，端口563（NNTPS）被使用。 *ssl_context*也是可选的，并且是一个SSLContext对象。请阅读最佳实践的安全考虑事项。所有其他参数的行为与for相同NNTP。

请注意，SSL-on-563不鼓励每个 **RFC 4642**，支持STARTTLS，如下所述。但是，有些服务器只支持前者。

3.2版本中的新功能

*在3.4版本中更改：*类现在支持主机名检查 ssl.SSLContext.check_hostname和*服务器名称指示*（见 ssl.HAS_SNI ）。

*异常*nntplib.NNTPError

从标准异常派生Exception，这是nntplib模块引发的所有异常的基类。这个类的实例具有以下属性：

response

服务器的响应（如果可用）作为str对象。

*异常*nntplib.NNTPReplyError

从服务器收到意外回复时引发异常。

*异常*nntplib.NNTPTemporaryError

收到400-499范围内的响应代码时引发异常。

*异常*nntplib.NNTPPermanentError

收到500-599范围内的响应代码时引发异常。

*异常*nntplib.NNTPProtocolError

从服务器收到的回复不是以范围1-5中的数字开头时引发的异常。

*异常*nntplib.NNTPDataError

响应数据中出现错误时引发异常。

21.16.1。NNTP对象

当连接时，NNTP和NNTP_SSL对象支持下列方法和属性。

21.16.1.1。属性

NNTP.nntp_version

表示服务器支持的NNTP协议版本的整数。实际上，这应该2用于服务器广告 **RFC 3977**合规性和1其他。

3.2版本中的新功能

NNTP.nntp_implementation

描述NNTP服务器的软件名称和版本的字符串，或者None如果未由服务器通告。

3.2版本中的新功能

21.16.1.2。方法

该*响应*被返回的几乎所有方法的返回元组的第一项是服务器的响应：一个字符串开头的三位数代码。如果服务器的响应表明有错误，则该方法引发上述例外之一。

以下许多方法都采用可选的仅含关键字的参数*文件*。当*文件*被提供的参数，它必须是一个文件对象 二进制写入打开，或者磁盘上的文件的名称被写入。然后该方法将把服务器返回的任何数据（响应行和终止点除外）写入文件; 该方法通常返回的任何行，元组或对象的列表都将为空。

*在版本3.2中*进行了*更改：*以下许多方法已经过重新设计和修正，这使得它们与3.1版本不兼容。

NNTP.quit()

发送一个QUIT命令并关闭连接。一旦调用了这个方法，就不应该调用NNTP对象的其他方法。

NNTP.getwelcome()

返回服务器发送的回复初始连接的欢迎消息。（此消息有时包含可能与用户有关的免责声明或帮助信息。）

NNTP.getcapabilities()

返回 作为dict实例映射功能名称（可能为空）的值列表的 **RFC 3977**功能 。在不了解CAPABILITIES 命令的旧式服务器上，将返回空字典。

```
>>> s = NNTP('news.gmane.org')
>>> 'POST' in s.getcapabilities()
True
```

3.2版本中的新功能

NNTP.**login**(*user = None*, *password = None*, *usenetrc = True*)

AUTHINFO用用户名和密码发送命令。如果*用户*和*密码*为，None并且*usenetrc*为true，~/.netrc那么将尽可能使用凭据。

除非故意延迟，否则通常在NNTP对象初始化期间执行登录，并且不需要单独调用此功能。要强制认证延迟，创建对象时不能设置*用户名*或*密码*，并且必须将*usenetrc*设置为False。

3.2版本中的新功能

NNTP.**starttls**(*ssl_context = None*)

发送一个STARTTLS命令。这将启用NNTP连接上的加密。该*ssl_context*参数是可选的，应该是一个ssl.SSLContext对象。请阅读最佳实践的安全考虑事项。

请注意，在传输认证信息后可能无法完成此操作，并且如果可能，在NNTP对象初始化过程中会进行身份验证。请参阅NNTP.login()有关抑制此行为的信息。

3.2版本中的新功能

*在3.4版本中更改：*该方法现在支持主机名检查ssl.SSLContext.check_hostname和*服务器名称指示*（见ssl.HAS_SNI）。

NNTP.**newgroups**(*日期*, *, *文件=无*)

发送一个NEWGROUPS命令。该*日期*参数应该是一个datetime.date或datetime.datetime对象。返回一对，其中*组*是一个列表，表示自给定*日期*以来新增的组。如果提供*文件*，但是，*组*将是空的。(response, groups)

```
>>> from datetime import date, timedelta
>>> resp, groups = s.newgroups(date.today() - timedelta(days=3))
>>> len(groups)
85
>>> groups[0]
GroupInfo(group='gmane.network.tor.devel', last='4', first='1', flag='m')
```

NNTP.**newnews**(*组*, *日期*, *, *文件=无*)

发送一个NEWNEWS命令。在这里，*组*是一个组名，或者'*'，*日期*的含义与以前相同newgroups()。返回一对 邮件，其中*articles*是邮件ID列表。(response, articles)

NNTP服务器管理员经常禁用此命令。

NNTP.**list**(*group_pattern = None*, *, *file = None*)

发送LIST或命令。返回一个对，其中*列表*是表示所有可用于此NNTP服务器的组的元组列表的元组列表，可选地匹配模式字符串*group_pattern*。每个元组都有一个表单，其中*group*是一个组名，*last*和*first*分别是最后一个和第一个文章编号，而*标志*通常取这些值中的一个：LIST ACTIVE(response, list)(group, last, first, flag)

- y：允许来自同行的本地帖子和文章。
- m：该组已被审核，所有帖子都必须获得批准。
- n：不允许本地发布，只有来自同行的文章。

- `j`：来自同行的文章被提交给垃圾组。
- `x`：没有本地发布，并且来自同行的文章被忽略。
- `=foo.bar`：文章是在`foo.bar`小组中提交的。

如果*标志*有另一个值，那么新闻组的状态应该被认为是未知的。

该命令可以返回非常大的结果，特别是如果 未指定*group_pattern*。除非您确实需要刷新结果，否则最好将结果离线进行缓存。

*在版本3.2中进行了更改：*添加了*group_pattern*。

NNTP.**descriptions**(*grouppattern*)

发送一个命令，其中*grouppattern*是一个在中指定的自变量字符串`LIST NEWSGROUPS`**RFC 3977**（它基本上与DOS或UNIX shell通配符字符串相同）。返回一对，其中*描述* 是将组名称映射到文本描述的字典。`(response, descriptions)`

```
>>> resp, descs = s.descriptions('gmane.comp.python.*')
>>> len(descs)
295
>>> descs.popitem()
('gmane.comp.python.bio.general', 'BioPython discussion list (Moderated)')
```

NNTP.**description**(*组*)

获取一个组的描述*组*。如果多个组匹配（如果“组”是真正的自变量字符串），则返回第一个匹配。如果没有组匹配，则返回一个空字符串。

这消除了来自服务器的响应代码。如果需要回应代码，请使用`descriptions()`。

NNTP.**group**(*名字*)

发送一个`GROUP`命令，其中*name*是组名称。如果存在该组，则选择该组作为当前组。返回一个元组，其中*count*是组 中的（估计）数量的文章，*首先*是组中的第一个文章编号，*last* 是组中的最后一个文章编号，*name* 是组名称。`(response, count, first, last, name)`

NNTP.**over**(*message_spec*, *, *file = None*)

在旧服务器上发送`OVER`命令或`XOVER`命令。 *message_spec*可以是表示消息ID的字符串，也可以是指示当前组中文章范围的数字元组，也可以是指示从当前组中的*第*一篇文章开始到最后一篇文章的文章范围的元组，或者选择当前组中的最新文章。`(first, last)``(first, None)``None`

返回一对。 *概览*是元组列表， 每个由*message_spec*选择的文章都有一个。每个*概述*都是具有相同数量项目的字典，但是这个数字取决于服务器。这些项目可以是消息标题（关键字是小写的标题名称），也可以是元数据项目（该关键字是元数据名称的前缀）。NNTP规范保证以下内容：`(response, overviews)``(article_number, overview)`":"

- 在`subject`，`from`，`date`，`message-id`和`references` 头
- 的`:bytes`元数据：字节的整个生制品中的号码（包括标题和正文）
- 的`:lines`元数据：线中相应体的数量

每个项目的值是一个字符串，或者`None`如果不存在。

`decode_header()`当它们可能包含非ASCII字符时，建议在标题值上使用该函数：

```
>>> _, _, first, last, _ = s.group('gmane.comp.python.devel')
>>> resp, overviews = s.over((last, last))
>>> art_num, over = overviews[0]
>>> art_num
117216
>>> list(over.keys())
['xref', 'from', ':lines', ':bytes', 'references', 'date', 'message-id', 'subject']
>>> over['from']
'=?UTF-8?B?Ik1hcnRpbiB2LiBMw7Z3aXMi?= <martin@v.loewis.de>'
>>> nntplib.decode_header(over['from'])
'"Martin v. Löwis" <martin@v.loewis.de>'
```

3.2版本中的新功能

NNTP.**help**(*, *file = None*)

发送一个HELP命令。返回一对，其中*列表*是帮助字符串的列表。(response, list)

NNTP.**stat**(*message_spec = None*)

发送一个STAT命令，其中*message_spec*是一个消息ID（包含在'<'和中'>'）或当前组中的物品编号。如果*message_spec*被省略或者None当前组中的当前文章被考虑。返回一个三元组 ，其中*number*是商品编号，*id*是消息编号。(response, number, id)

```
>>> _, _, first, last, _ = s.group('gmane.comp.python.devel')
>>> resp, number, message_id = s.stat(first)
>>> number, message_id
(9099, '<20030112190404.GE29873@epoch.metaslash.com>')
```

NNTP.**next**( )

发送一个NEXT命令。返回stat()。

NNTP.**last**( )

发送一个LAST命令。返回stat()。

NNTP.**article**(*message_spec = None*, *, *file = None*)

发送一个ARTICLE命令，其中*message_spec*的含义与以前相同stat()。返回一个元组，其中*info* 是一个具有三个属性的*数字*， *message_id*和*行*（按该顺序）。 *数*是该组中的物品号码（或0，如果信息不可用），*MESSAGE_ID*消息ID作为字符串，和*线*的行的列表（而不终止新行），其包括原始消息包括标题和主体。(response, info)namedtuple

```
>>> resp, info = s.article('<20030112190404.GE29873@epoch.metaslash.com>')
>>> info.number
0
>>> info.message_id
'<20030112190404.GE29873@epoch.metaslash.com>'
>>> len(info.lines)
65
>>> info.lines[0]
b'Path: main.gmane.org!not-for-mail'
>>> info.lines[1]
b'From: Neal Norwitz <neal@metaslash.com>'
```

```
>>> info.lines[-3:]
[b'There is a patch for 2.3 as well as 2.2.', b'', b'Neal']
```

NNTP.**head**(*message_spec = None*, *, *file = None*)

与之相同article()，但发送HEAD命令。的*线*返回（或写入*文件*）将仅包含消息头，而不是体。

NNTP.**body**(*message_spec = None*, *, *file = None*)

与之相同article()，但发送BODY命令。的*线*返回（或写入*文件*）将仅包含消息体，而不是头。

NNTP.**post**(*数据*)

使用该POST命令发布文章。的*数据*参数可以是一个文件对象打开的二进制读数，或（被张贴代表制品的原始线）字节对象的任何可迭代。它应该代表一个格式良好的新闻文章，包括必需的标题。该 post()方法自动转义以行开始.并添加终止行。

如果方法成功，则返回服务器的响应。如果服务器拒绝发布，NNTPReplyError则会引发a。

NNTP.**ihave**(*message_id*, *data*)

发送IHAVE命令。*message_id*是发送到服务器的消息的标识符（用'<'和括起来'>'）。该*数据*参数和返回值是一样的post()。

NNTP.**date**( )

返回一对。 *日期*是一个 包含服务器当前日期和时间的对象。(response, date)datetime

NNTP.**slave**( )

发送一个SLAVE命令。返回服务器的*响应*。

NNTP.**set_debuglevel**(*level*)

设置实例的调试级别。这将控制打印的调试输出量。缺省值，0不会产生调试输出。值 1会产生适量的调试输出，通常每个请求或响应只有一行。值2或更高值会产生最大数量的调试输出，记录连接上发送和接收的每条线路（包括消息文本）。

以下是在中定义的可选NNTP扩展 **RFC 2980**。其中一些已被更新的命令所取代**RFC 3977**。

NNTP.**xhdr**(*hdr*, *str*, *, *file = None*)

发送XHDR命令。的*HDR*参数是一个报头的关键字，例如'subject'。该*海峡*的说法应该具有的形式'first-last'，其中*第一*和*最后一个*是第一个和最后的文章编号，以搜索。返回一对，其中*list*是对的列表，其中*id*是商品编号（作为字符串），*text*是该文章请求的标题的文本。如果提供了*文件*参数，则命令的输出将 存储在文件中。如果*file*是一个字符串，那么该方法将打开一个具有该名称的文件，写入并关闭它。如果(response, list)(id, text)XHDR*文件*是一个文件对象，那么它将开始调用write()它来存储命令输出的行。如果提供了*文件*，则返回的*列表*是一个空列表。

NNTP.**xover**(*开始*, *结束*, *, *文件=无*)

发送XOVER命令。 *开始*和*结束*是分隔要选择的文章范围的文章编号。返回值与for相同over()。建议使用over()，因为它会自动使用较新的OVER命令（如果可用）。

NNTP.xpath（*id*）

返回一个对，其中*path*是具有消息ID *标识*的文章的目录路径。大多数情况下，这个扩展没有被NNTP服务器管理员启用。(resp, path)

自版本3.3起弃用： XPATH扩展未被有效使用。

21.16.2。效用函数

该模块还定义了以下实用程序功能：

nntplib.decode_header（*header_str*）

解码标头值，取消转义任何转义的非ASCII字符。 *header_str*必须是一个str对象。非转义的值被返回。建议使用此功能以可读的形式显示一些标题：

```
>>> decode_header("Some subject")
'Some subject'
>>> decode_header("=?ISO-8859-15?Q?D=E9buter_en_Python?=")
'Débuter en Python'
>>> decode_header("Re: =?UTF-8?B?cHJvYmzDqG1lIGRlIG1hdHJpY2U=?=")
'Re: problème de matrice'
```

21.17。 smtplib- SMTP协议客户端

源代码： Lib / smtplib.py

该smtplib模块定义了一个SMTP客户端会话对象，可用于使用SMTP或ESMTP侦听器守护程序将邮件发送到任何Internet计算机。有关SMTP和ESMTP操作的详细信息，请参阅**RFC 821**（简单邮件传输协议）和**RFC 1869**（SMTP服务扩展）。

class smtplib. SMTP（ *host =''*，*port = 0*，*local_hostname = None*，[*timeout*，] *source_address = None* ）

一个SMTP实例封装了一个SMTP连接。它具有支持完整的SMTP和ESMTP操作的方法。如果给出了可选的主机和端口参数，则connect()在初始化期间使用这些参数调用SMTP 方法。如果指定， 则使用*local_hostname*作为HELO / EHLO命令中本地主机的FQDN。否则，使用本地主机名 socket.getfqdn()。如果connect()调用返回成功代码以外的任何其他内容，SMTPConnectError则引发。可选的 *timeout*参数指定阻止诸如连接尝试（如果未指定，将使用全局默认超时设置）等操作的超时（以秒为单位）。如果超时过期，socket.timeout被提出。可选的source_address参数允许绑定到具有多个网络接口的计算机中的某个特定源地址，和/或某些特定的源TCP端口。它需要一个2元组（主机，端口），以便套接字在连接之前绑定为其源地址。如果省略（或者如果主机或端口分别为''和/或0），则将使用操作系统默认行为。

对于正常使用，您只需要初始化/连接 sendmail() ，和quit()方法。下面是一个例子。

该SMTP课程支持该with声明。当像这样使用QUIT时，with语句退出时会自动发出SMTP 命令 。例如：

```
>>> from smtplib import SMTP
>>> with SMTP("domain.org") as smtp:
...     smtp.noop()
...
(250, b'Ok')
>>>
```

*在版本3.3中进行了更改：*with添加了对语句的支持。

*在版本3.3中更改：*添加了source_address参数。

*3.5版*新增功能：SMTPUTF8扩展（**RFC 6531**）现在支持。

class smtplib. SMTP_SSL（ *host =''*，*port = 0*，*local_hostname = None*，*keyfile = None*，*certfile = None*，[*timeout*，] *context = None*，*source_address = None* ）

一个SMTP_SSL实例的行为与实例的行为完全相同 SMTP。 SMTP_SSL应该用于需要从连接开始使用SSL并且使用starttls()不合适的情况。如果*主机*没有指定，则使用本地主机。如果*端口*为零，则使用标准的SMTP-over-SSL端口（465）。可选参数*local_hostname*，*timeout*和*source_address*与它们在SMTP类中的含义相同。 *上下文*也是可选的，可以包含SSLContext和允许配置安全连接的各个方面。请阅读最佳实践的安全考虑事项。

*密钥文件*和*certfile*是*上下文*的传统替代方案，并且可以指向用于SSL连接的PEM格式的私钥和证书链文件。

*在版本3.3中更改：*添加了*上下文*。

*在版本3.3中更改：*添加了source_address参数。

*在3.4版本中更改：*类现在支持主机名检查 ssl.SSLContext.check_hostname和*服务器名称指示*（见 ssl.HAS_SNI ）。

自版本3.6起弃用： *keyfile* 和 *certfile* 不赞成使用*上下文*。请 ssl.SSLContext.load_cert_chain() 改用，或者让我们 ssl.create_default_context() 为您选择系统的可信CA证书。

class smtplib.LMTP（*host =''*，*port = LMTP_PORT*，*local_hostname = None*，*source_address = None* ）

LMTP协议与ESMTP非常相似，主要基于标准的SMTP客户端。对LMTP使用Unix套接字是很常见的，所以我们的 connect() 方法必须支持以及常规的主机：端口服务器。可选参数local_hostname和source_address与它们在SMTP类中的含义相同。要指定一个Unix套接字，您必须使用绝对路径作为*主机*，以'/'开头。

使用常规的SMTP机制支持身份验证。使用Unix套接字时，LMTP通常不支持或不需要任何身份验证，但您的里程可能会有所不同。

一个很好的例外选择也被定义为：

*异常*smtplib.SMTPException

这个子类OSError是这个模块提供的所有其他异常的基本异常类。

在版本3.4中更改： SMTPException成为了OSError

*异常*smtplib.SMTPServerDisconnected

当服务器意外断开连接时，或者在将SMTP连接到服务器之前尝试使用实例时，会引发此异常。

*异常*smtplib.SMTPResponseException

包含SMTP错误代码的所有异常的基类。当SMTP服务器返回错误代码时，会在某些情况下生成这些例外情况。错误代码存储在错误的smtp_code属性中，并且 smtp_error属性设置为错误消息。

*异常*smtplib.SMTPSenderRefused

发件人地址被拒绝。除了所有SMTPResponseException例外设置的属性之外，这会将 “发件人”设置为SMTP服务器拒绝的字符串。

*异常*smtplib.SMTPRecipientsRefused

所有收件人地址被拒绝 每个收件人的错误都可以通过该属性进行访问，该属性recipients与SMTP.sendmail() 回报完全一样。

*异常*smtplib.SMTPDataError

SMTP服务器拒绝接受消息数据。

*异常*smtplib.SMTPConnectError
在建立与服务器的连接期间发生错误。

*异常*smtplib.SMTPHeloError
服务器拒绝了我们的HELO消息。

*异常*smtplib.SMTPNotSupportedError
服务器不支持所尝试的命令或选项。

3.5版本中的新功能。

*异常*smtplib.SMTPAuthenticationError
SMTP认证出错了。很可能服务器不接受提供的用户名/密码组合。

也可以看看:

RFC 821 - 简单邮件传输协议
SMTP的协议定义。本文档涵盖了SMTP的型号，操作步骤和协议详细信息。

RFC 1869 - SMTP服务扩展
SMTP的ESMTP扩展的定义。这描述了一个用新命令扩展SMTP的框架，支持动态发现服务器提供的命令，并定义了一些附加命令。

21.17.1。SMTP对象

一个SMTP实例有以下方法：

SMTP.set_debuglevel(*level*)
设置调试输出级别。值为1或True为*级别将*导致调试消息的连接以及发送到服务器和从服务器接收的所有消息。的2值*电平*在这些消息的结果被加时间戳。

*在版本3.5中更改：*添加了调试级别2。

SMTP.docmd(*cmd*, *args=''*)
发送命令*cmd*到服务器。可选参数*args*简单地连接到命令，由空格分隔。

这将返回一个由数字响应代码和实际响应行组成的2元组（多行响应连接成一个长行）。

在正常操作中，不需要明确调用此方法。它用于实现其他方法，可能对测试私有扩展有用。

如果在等待答复时到服务器的连接丢失，SMTPServerDisconnected则会引发。

SMTP.connect(*host='localhost'*, *port=0*)
连接到给定端口上的主机。默认值是通过标准SMTP端口（25）连接到本地主机。如果主机名以冒号（':'）后跟一个数字结尾，则该后缀将被剥离，并将该数字解释为要使用的端

口号。如果在实例化过程中指定了主机，则此方法由构造函数自动调用。返回服务器在其连接响应中发送的响应代码和消息的2元组。

SMTP.helo（*name =''*）

使用自己的身份识别SMTP服务器HELO。主机名参数默认为本地主机的完全限定域名。服务器返回的消息存储为helo_resp对象的属性。

在正常操作中，不需要明确调用此方法。 sendmail()必要时它将被隐式调用。

SMTP.ehlo（*name =''*）

使用自己的身份识别ESMTP服务器EHLO。主机名参数默认为本地主机的完全限定域名。检查ESMTP选项的响应并将其存储以供使用has_extn()。还设置了几个信息属性：服务器返回的消息存储为ehlo_resp属性，does_esmtp 根据服务器是否支持ESMTP esmtp_features 将其设置为true或false，并且 将是包含此服务器支持的SMTP服务扩展名的字典，以及他们的参数（如果有的话）。

除非您希望has_extn()在发送邮件之前使用它，否则不需要明确调用此方法。 sendmail()必要时它将被隐式调用 。

SMTP.ehlo_or_helo_if_needed（）

此方法调用ehlo()和或者helo()如果之前没有EHLO或HELO命令此会话。它EHLO 首先尝试ESMTP 。

SMTPHeloError

服务器没有正确回复HELO问候。

SMTP.has_extn（*名字*）

返回True如果*名称*是在设置服务器返回的SMTP服务扩展的，False否则。案例被忽略。

SMTP.verify（*地址*）

使用SMTP检查此服务器上地址的有效性VRFY。返回由代码250和完整数组成的元组**RFC 822**地址（包括人名），如果用户地址有效。否则，返回400或更大的SMTP错误代码和错误字符串。

> **注意：** 许多网站禁用SMTP VRFY以阻止垃圾邮件发送者。

SMTP.login（*user*，*password*，*，*initial_response_ok = True*）

登录需要认证的SMTP服务器。参数是用于验证的用户名和密码。如果之前没有 EHLO或HELO命令此会话，则此方法EHLO 首先尝试使用ESMTP 。如果认证成功，此方法将正常返回，或者可能引发以下例外情况：

SMTPHeloError

服务器没有正确回复HELO问候。

SMTPAuthenticationError

服务器不接受用户名/密码组合。

SMTPNotSupportedError

该AUTH命令不受服务器支持。

SMTPException
没有找到合适的验证方法。

smtplib如果支持的每种身份验证方法都按照服务器的支持发布，则会依次尝试。请参阅有关auth() 支持的身份验证方法的列表。 *initial_response_ok*被传递给auth()。

可选的关键字参数*initial_response_ok*指定对于支持它的认证方法，是否在中指定了“初始响应”**RFC 4954**可以与AUTH命令一起发送，而不需要挑战/响应。

*在版本3.5中更改：*SMTPNotSupportedError可能会引发，并添加了 *initial_response_ok*参数。

SMTP.auth（*mechanism*，*authobject*，*，*initial_response_ok = True* ）
SMTP AUTH为指定的认证*机制*发出命令 ，并通过*authobject*处理质询响应。

*机制*指定哪个认证机制将用作该AUTH命令的参数; 有效值是那些在auth元素中列出的值esmtp_features。

*authobject*必须是一个可调用的对象，并带有可选的单个参数：

data = authobject（challenge = None）

如果可选关键字参数*initial_response_ok*为true， authobject()则将首先调用不带参数。它可以返回 **RFC 4954** “初始响应”字节将被编码并通过如下AUTH命令发送。如果authobject()不支持初始响应（例如，因为它需要一个挑战），它应该None在被调用时返回 challenge=None。如果 *initial_response_ok*为false，则authobject()不会首先调用None。

如果初始响应检查返回None，或者*initial_response_ok*为false，authobject()则会调用它来处理服务器的质询响应; 它通过的*挑战*论证将是a bytes。它应该返回bytes *的数据*将被base64编码并发送到服务器。

的 SMTP 类提供 authobjects 的 CRAM-MD5 ， PLAIN 和 LOGIN 机制；它们被命名为SMTP.auth_cram_md5 ， SMTP.auth_plain ，和SMTP.auth_login分别。它们都要求将实例的属性user和password属性SMTP设置为适当的值。

用户代码通常不需要auth直接调用，而是可以调用login()方法，该方法将依次按照列出的顺序尝试上述每种机制。 auth被暴露以促进不直接支持（或尚未）支持的认证方法的实现smtplib。

3.5版本中的新功能。

SMTP.starttls（*keyfile = None*，*certfile = None*，*context = None* ）
将SMTP连接置于TLS（传输层安全性）模式。随后的所有SMTP命令都将被加密。你应该再打电话ehlo() 。

如果提供*密钥文件*和*certfile*，则会将其传递给socket 模块的ssl()功能。

可选的*上下文*参数是一个ssl.SSLContext对象; 这是使用密钥文件和证书文件的替代方法，如果指定*密钥文件*和*证书*文件都 应该None。

如果之前没有EHLO或HELO命令此会话，则此方法EHLO首先尝试使用ESMTP 。

SMTPHeloError
服务器没有正确回复HELO问候。

SMTPNotSupportedError
该服务器不支持STARTTLS扩展名。

RuntimeError
您的Python解释器不支持SSL / TLS支持。

*在版本3.3中更改：*添加了*上下文*。

*在3.4版本中更改：*该方法现在支持主机名检查与 SSLContext.check_hostname和*服务器名称指标*（见 HAS_SNI ）。

*在版本3.5中更改：*缺少STARTTLS支持的错误现在是 SMTPNotSupportedError子类而不是基础 SMTPException。

SMTP.sendmail（*from_addr*，*to_addrs*，*msg*，*mail_options = []*，*rcpt_options = []* ）
发送邮件。所需的参数是**RFC 822**从 - 地址字符串，一个列表**RFC 822**地址字符串（一个裸体字符串将被视为一个带有1个地址的列表）和一个消息字符串。调用者可以传递一个ESMTP选项列表（例如8bitmime ）作为 *mail_options*在命令中*使用*。应该与所有 命令一起使用的ESMTP选项（如命令）可以作为 *rcpt_options*传递。（如果您需要使用不同的ESMTP选项不同的收件人必须使用低级别的方法，例如 ，和发送消息。 ）MAIL FROMDSNRCPTmail()rcpt()data()

> **注意：** 的*from_addr*和*to_addrs*参数被用于构建由传输代理使用的消息信封。 sendmail不以任何方式修改消息标题。

*msg*可能是一个字符串，它包含ASCII范围内的字符或一个字节字符串。使用ascii编解码器将字符串编码为字节，并将孤立字符\r 和\n字符转换为\r\n字符。字节字符串不被修改。

如果之前没有EHLO或HELO命令此会话，则此方法EHLO首先尝试使用ESMTP 。如果服务器执行ESMTP，则将消息大小和每个指定的选项传递给它（如果选项位于服务器通告的功能集中）。如果EHLO失败，HELO则会尝试并取消ESMTP选项。

如果邮件被至少一个收件人接受，此方法将正常返回。否则会引发异常。也就是说，如果此方法不引发异常，则有人应该收到您的邮件。如果此方法不引发异常，它将返回一个字典，每个收件人都有一个条目被拒绝。每个条目都包含SMTP错误代码的元组以及服务器发送的伴随错误消息。

如果SMTPUTF8包含在*mail_options中*，并且服务器支持它， *from_addr*和*to_addrs*可能包含非ASCII字符。

此方法可能会引发以下例外情况：

SMTPRecipientsRefused
所有收件人都被拒绝。没有人收到邮件。 recipients 异常对象的属性是一个带有关于被拒绝的收件人的信息的字典（例如至少有一个收件人被接受时返回的信息）。

SMTPHeloError

服务器没有正确回复HELO问候。

SMTPSenderRefused
服务器不接受*from_addr*。

SMTPDataError
服务器回复了一个意外的错误代码（拒绝收件人除外）。

SMTPNotSupportedError
SMTPUTF8在*mail_options*中给出，但不受服务器支持。

除非另有说明，否则即使发生异常，连接也会打开。

*在版本3.2中更改：*msg可能是一个字节字符串。

*在版本3.5中进行了更改：*SMTPUTF8添加了支持，并且SMTPNotSupportedError如果SMTPUTF8指定但可能会引发服务器不支持它。

SMTP.send_message(*msg*，*from_addr = None*，*to_addrs = None*，*mail_options = []*，*rcpt_options = []*)
这是使用sendmail()由email.message.Message对象表示的消息进行调用的便捷方法。这些参数的含义*与之*相同sendmail()，只是*msg*是一个Message 对象。

如果*from_addr*是None或*to_addrs*是None，则send_message用从*msg*头部提取的地址填充那些参数，如**RFC 5322**：*from_addr*设置为*发件人* 字段（如果存在），否则设置为*发件人*字段。*to_addrs*结合来自 *msg*的 *To*， *Cc*和 *Bcc*字段的值（如果有）。如果消息中只出现一组 *Resent-* *标头，则将忽略常规标头，并使用 *Resent-* *标头。如果消息包含多个 *Resent-* *标头集合，*则会*引发a，因为无法明确检测最近一组 *Resent-*标头。ValueError

send_message序列化*MSG*使用 BytesGenerator与\r\n作为*linesep*，并调用sendmail()来发送所产生的消息。无论*from_addr*和*to_addrs*的值*如何*，send_message都不会传输任何 可能出现在*msg中的密件抄送*或*重新发送的密件抄送*标题。如果*from_addr*和*to_addrs中的*任何地址包含非ASCII字符，并且服务器不支持通告，则会发生错误。否则，序列化与它的克隆与设置属性，并与SMTPUTF8 SMTPNotSupported Message policy utf8 True SMTPUTF8 BODY=8BITMIME 被添加到*mail_options*。

3.2版本中的新功能

*3.5版新增功能：*支持国际化地址（SMTPUTF8）。

SMTP.quit()
终止SMTP会话并关闭连接。返回SMTP QUIT命令的结果。

对应于标准SMTP / ESMTP命令低级别的方法HELP， RSET，NOOP，MAIL，RCPT，和DATA也被支持。通常这些不需要直接调用，所以它们在这里没有记录。有关详细信息，请查阅模块代码。

21.17.2。SMTP示例

本示例提示用户输入邮件信封（'收件人'和'发件人'地址）中需要的地址以及要发送的邮件。请注意，消息中包含的标题必须包含在输入的消息中; 这个例子不做任何处理**RFC 822**标头。特别是，“收件人”和“发件人”地址必须明确包含在邮件标题中。

```
import smtplib

def prompt(prompt):
    return input(prompt).strip()

fromaddr = prompt("From: ")
toaddrs  = prompt("To: ").split()
print("Enter message, end with ^D (Unix) or ^Z (Windows):")

# Add the From: and To: headers at the start!
msg = ("From: %s\r\nTo: %s\r\n\r\n"
       % (fromaddr, ", ".join(toaddrs)))
while True:
    try:
        line = input()
    except EOFError:
        break
    if not line:
        break
    msg = msg + line

print("Message length is", len(msg))

server = smtplib.SMTP('localhost')
server.set_debuglevel(1)
server.sendmail(fromaddr, toaddrs, msg)
server.quit()
```

注意： 一般来说，您会希望使用该`email`包的功能来构建电子邮件，然后您可以通过该电子邮件发送`send_message()`; 看**电子邮件：例子**。

21.18。 smtpd- SMTP服务器

源代码： Lib / smtpd.py

该模块提供了几个类来实现SMTP（电子邮件）服务器。

也可以看看： 所述aiosmtpd包是该模块的推荐更换。它基于asyncio并提供更直接的API。smtpd应被视为弃用。

有几个服务器实现存在; 一个是通用的无所作为的实现，可以被覆盖，另外两个提供特定的邮件发送策略。

另外，SMTPChannel可以扩展为与SMTP客户端实现非常特定的交互行为。

代码支持 **RFC 5321**，加上**RFC 1870** SIZE和**RFC 6531** SMTPUTF8扩展。

21.18.1。SMTPSERVER对象

*类*smtpd.SMTPServer（*localaddr*，*remoteaddr*，*data_size_limit = 33554432*，*地图=无*，*enable_SMTPUTF8 =假*，*decode_data =假*）

创建一个SMTPServer绑定到本地地址*localaddr*的新对象 。它会将*remoteaddr*视为上游SMTP中继。无论 *localaddr*和*remoteaddr*应该是（主机，端口）元组。该对象继承自asyncore.dispatcher，并将自身插入asyncore实例化的事件循环中。

*data_size_limit*指定将在DATA命令中接受的最大字节数。的值None或0表示没有限制。

*map*是用于连接的套接字映射（最初为空的字典是合适的值）。如果未指定，asyncore 则使用全局套接字映射。

*enable_SMTPUTF8*决定SMTPUTF8扩展名（如定义在**RFC 6531**）应该被启用。默认是False。 时True ， SMTPUTF8被接受作为参数的MAIL 指令和当存在时被传递给process_message() 在所述 kwargs['mail_options'] 列表中。 *decode_data* 和 *enable_SMTPUTF8* 不能同时设置True。

*decode_data*指定是否应使用UTF-8解码SMTP事务的数据部分。当*decode_data*是False（默认）时，服务器通告8BITMIME 扩展名（**RFC 6152**），接受BODY=8BITMIME参数的MAIL命令，并且当存在时其传递到process_message() 在kwargs['mail_options']列表中。*decode_data*和 *enable_SMTPUTF8* 不能同时设置True。

process_message（*peer*，*mailfrom*，*rcpttos*，*data*，*** kwargs* ）

引发NotImplementedError异常。在子类中重写此操作以对此消息执行一些有用的操作。不管在构造函数中传递的是什么，*remoteaddr*都将作为_remoteaddr 属性提供。*peer*是远程主机的地址，*mailfrom*是信封创建者，*rcpttos*是信封收件人，*data*是包含电子邮件内容的字符串（应该在**RFC 5321** 格式）。

如果*decode_data*构造函数关键字设置为True，*data* 参数将是一个unicode字符串。如果它设置为False，它将是一个字节对象。

*kwargs*是包含附加信息的字典。如果decode_data=True作为init参数给出，则它是空的，否则它包含以下键：

mail_options：
所有接收到的参数列表MAIL （元素都是大写字符串;例如：）。
['BODY=8BITMIME', 'SMTPUTF8']

rcpt_options：
与*mail_options*相同，但是用于RCPT命令。目前没有选项被支持，所以现在这将永远是一个空列表。 RCPT TO

实现process_message应该使用**kwargs 签名来接受任意的关键字参数，因为未来的功能增强可能会将密钥添加到kwargs字典中。

返回None请求正常响应; 否则返回所需的响应字符串250 Ok**RFC 5321**格式。

channel_class
在子类中覆盖此项以使用自定义SMTPChannel来管理SMTP客户端。

*在新版本3.4：在地图*构造函数的参数。

*版本3.5中已更改：localaddr*和*remoteaddr*现在可能包含IPv6地址。

*新的3.5版：该decode_data*和*enable_SMTPUTF8*构造函数的参数，以及 *kwargs*参数process_message()时*decode_data*是 False。

*在版本3.6中更改：decode_data*现在False是默认情况。

21.18.2。调试服务器对象

class smtpd.DebuggingServer（*localaddr*，*remoteaddr* ）
创建一个新的调试服务器。参数是按照SMTPServer。消息将被丢弃，并打印在stdout上。

21.18.3。PureProxy对象

class smtpd.PureProxy（*localaddr*，*remoteaddr* ）
创建一个新的纯代理服务器。参数是按照SMTPServer。一切都会被传送到*remoteaddr*。请注意，运行这是一个很好的机会让你成为一个开放的继电器，所以请小心。

21.18.4。MailmanProxy对象

class smtpd.MailmanProxy（*localaddr*，*remoteaddr* ）
创建一个新的纯代理服务器。参数是按照SMTPServer。除非本地邮递员配置知道地址，否则一切都会被转发到*remoteaddr*，在这种情况下，邮件将通过邮递员处理。请注意，运行

这是一个很好的机会让你成为一个开放的继电器，所以请小心。

21.18.5。SMTPChannel对象

class smtpd.SMTPChannel（*server*，*conn*，*addr*，*data_size_limit = 33554432*，*map = None*，*enable_SMTPUTF8 = False*，*decode_data = False* ）

创建一个SMTPChannel管理服务器和单个SMTP客户端之间通信的新对象。

*conn*和*addr*按照下面描述的实例变量。

*data_size_limit*指定将在DATA命令中接受的最大字节数。的值None或0表示没有限制。

*enable_SMTPUTF8*决定SMTPUTF8扩展名（如定义在**RFC 6531** ）应该被启用。默认是False。 *decode_data*和 *enable_SMTPUTF8*不能同时设置True。

可以在*map*中指定字典以避免使用全局套接字映射。

*decode_data*指定是否应使用UTF-8解码SMTP事务的数据部分。默认是False。*decode_data*和*enable_SMTPUTF8*不能同时设置True。

要使用自定义SMTPChannel实现，您需要覆盖 SMTPServer.channel_class 您的SMTPServer。

*改变在3.5版本：*该*decode_data*和*enable_SMTPUTF8*添加参数。

在版本3.6中更改： *decode_data*现在False是默认情况。

将SMTPChannel具有以下实例变量：

smtp_server
: 拥有SMTPServer这个频道。

conn
: 保存连接到客户端的套接字对象。

addr
: 保存客户端的地址，返回的第二个值 socket.accept

received_lines
: 保存从客户端收到的线串（使用UTF-8解码）的列表。这些"\r\n"行将其行结尾转换为"\n"。

smtp_state
: 保持通道的当前状态。这将是 COMMAND最初的，然后DATA在客户端发送“DATA”行之后。

seen_greeting
: 保存包含客户端在其“HELO”中发送的问候语的字符串。

`mailfrom`
保存包含客户端“MAIL FROM：”行中标识的地址的字符串。

`rcpttos`
保存包含来自客户端的“RCPT TO：”行中标识的地址的字符串列表。

`received_data`
保存包含客户端在DATA状态期间发送的所有数据的字符串，直至但不包括终止"\r\n.\r\n"。

`fqdn`
保存返回的服务器的完全限定域名 `socket.getfqdn()`。

`peer`
拥有客户端对等的名称作为被退回`conn.getpeername()` 地方`conn`是`conn`。

在`SMTPChannel`由调用名为方法操作`smtp_<command>` 时从所述客户端的命令行的接收。内置于基 `SMTPChannel`类中的是处理以下命令的方法（并对它们进行适当的响应）：

命令	所采取的行动
HELO	接受客户的问候并存入 `seen_greeting`。将服务器设置为基本命令模式。
EHLO	接受客户的问候并存入 `seen_greeting`。将服务器设置为扩展命令模式。
NOOP	不采取行动。
放弃	干净地关闭连接。
邮件	接受“MAIL FROM：”语法并将提供的地址存储为 `mailfrom`。在扩展命令模式下，接受 **RFC 1870** SIZE属性并基于 *data_size_limit*的值适当地作出响应。
RCPT	接受“RCPT TO：”语法并将提供的地址存储在`rcpttos`列表中。
RSET	重置`mailfrom`，`rcpttos`和 `received_data`，而不是问候。
数据	将内部状态设置为`DATA`并存储来自客户端的剩余行，`received_data`直到"\r\n.\r\n"接收到终止符。
帮帮我	返回关于命令语法的最少信息
VRFY	返回代码252（服务器不知道该地址是否有效）
EXPN	报告该命令未实现。

21.19。 telnetlib- Telnet客户端

源代码： Lib / telnetlib.py

该telnetlib模块提供了一个Telnet实现Telnet协议的类。看到有关该协议的详细信息，请参阅 **RFC 854**。另外，它为协议字符（见下面）和telnet选项提供了符号常量。远程登录选项的符号名称遵循其中的定义arpa/telnet.h，并TELOPT_删除了前导符。对于传统上未包含的选项的符号名称arpa/telnet.h，请参阅模块源本身。

telnet命令的符号常量为：IAC，DONT，DO，WONT，WILL，SE（子协商结束），NOP（无操作），DM（数据标记），BRK（中断），IP（中断过程），AO中止输出），AYT（你在那里），EC（擦除字符），EL（擦除线），GA（前进），SB（子协商开始）。

class telnetlib.Telnet（*host* = *None*，*port* = *0*[，*timeout*]）

Telnet代表到Telnet服务器的连接。该实例最初并未默认连接; 该open()方法必须用于建立连接。或者，也可以将主机名和可选端口号传递给构造函数，在这种情况下，将在构造函数返回之前建立与服务器的连接。可选的 *timeout*参数指定阻止诸如连接尝试（如果未指定，将使用全局默认超时设置）等操作的超时（以秒为单位）。

不要重新打开已连接的实例。

这个类有很多read_*()方法。请注意，EOFError当读取连接结束时会引发其中的一些问题，因为由于其他原因它们可能会返回空字符串。请参阅下面的个别说明。

一个Telnet对象是一个上下文管理器，可以用在一个 with语句中。当with块结束，则close()方法被称为：

```
>>> from telnetlib import Telnet
>>> with Telnet('localhost', 23) as tn:
...     tn.interact()
...
```

*在版本3.6中更改：*添加了上下文管理器支持

也可以看看:

RFC 854 - Telnet协议规范
Telnet协议的定义。

21.19.1。Telnet对象

Telnet 实例具有以下方法：

Telnet.read_until（*预计*，*超时=无*）

阅读到一个给定的字节串，*预计*，当遇到或直到*超时*秒过去了。

如果找不到匹配项，则返回可用的内容，可能为空字节。EOFError如果连接关闭并且没有熟化数据可用，则提升。

Telnet.read_all（）
读取所有数据，直到EOF为字节; 阻塞直到连接关闭。

Telnet.read_some（）
除非命中EOF，否则至少读取一个字节的已煮熟数据。b''如果EOF被击中，则返回。如果没有数据立即可用，则阻止。

Telnet.read_very_eager（）
阅读所有可以在I / O（热切）中不受阻塞的内容。

EOFError如果连接关闭并且没有可用的已烹饪数据，则升起 b''如果没有可用的数据，则返回。除非在IAC序列中，否则不要阻塞。

Telnet.read_eager（）
阅读随时可用的数据。

EOFError如果连接关闭并且没有可用的已烹饪数据，则升起 b''如果没有可用的数据，则返回。除非在IAC序列中，否则不要阻塞。

Telnet.read_lazy（）
处理并返回队列中已有的数据（懒惰）。

EOFError如果连接关闭且没有可用数据，则提升。b''如果没有可用的数据，则返回 。除非在IAC序列中，否则不要阻塞。

Telnet.read_very_lazy（）
返回已熟练队列中的任何数据（非常懒）。

EOFError如果连接关闭且没有可用数据，则提升。b''如果没有可用的数据，则返回 。这个方法永远不会阻塞

Telnet.read_sb_data（）
返回SB / SE对之间收集的数据（子选项开始/结束）。回调应该在用SE命令调用时访问这些数据。这个方法永远不会阻塞

Telnet.open（*host*，*port = 0*[，*timeout*]）
连接到主机。可选的第二个参数是端口号，默认为标准Telnet端口（23）。可选的*timeout*参数指定阻止诸如连接尝试（如果未指定，将使用全局默认超时设置）等操作的超时（以秒为单位）。

不要尝试重新打开已连接的实例。

Telnet.msg（*msg*，* *args*）
当调试级别为>0 时打印调试消息。如果存在额外的参数，则使用标准字符串格式操作符将其替换为消息。

Telnet.set_debuglevel(*debuglevel*)
设置调试级别。*debuglevel*的值越高，获得的调试输出就越多（on sys.stdout）。

Telnet.close()
关闭连接。

Telnet.get_socket()
返回内部使用的套接字对象。

Telnet.fileno()
返回内部使用的套接字对象的文件描述符。

Telnet.write(*缓冲区*)
将一个字节字符串写入套接字，使IAC字符加倍。这可以阻止连接被阻止。OSError如果连接关闭，可能会引发。

*版本3.3中更改：*此方法用于引发socket.error，现在它是一个别名OSError。

Telnet.interact()
交互功能，模拟一个非常笨的Telnet客户端。

Telnet.mt_interact()
多线程版本interact()。

Telnet.expect(*list*, *timeout* = *None*)
阅读，直到正则表达式列表中的一个匹配。

第一个参数是正则表达式列表，可以是编译的（正则表达式对象）也可以是未编译的（字节字符串）。可选的第二个参数是以秒为单位的超时值; 默认是无限期阻止。

返回三个元素的元组：第一个正则表达式匹配的列表中的索引; 匹配对象返回; 并读取字节直到包括匹配。

如果找到文件结尾并且没有读取字节，请进行提升EOFError。否则，不匹配时，返回其中*的数据*是迄今收到的字节（可能为空字节，如果超时发生）。(-1, None, data)

如果正则表达式以贪婪匹配（例如.*）结束，或者如果多个表达式可以匹配相同的输入，则结果是非确定性的，并且可能取决于I / O时序。

Telnet.set_option_negotiation_callback(*回调*)
每次在输入流上读取telnet选项时，使用以下参数调用此*回调*（如果已设置）：callback（telnet套接字，命令（DO / DONT / WILL / WONT），选项）。之后telnetlib不会执行其他操作。

21.19.2。Telnet示例

一个说明典型用途的简单例子：

```
import getpass
import telnetlib

HOST = "localhost"
user = input("Enter your remote account: ")
password = getpass.getpass()

tn = telnetlib.Telnet(HOST)

tn.read_until(b"login: ")
tn.write(user.encode('ascii') + b"\n")
if password:
    tn.read_until(b"Password: ")
    tn.write(password.encode('ascii') + b"\n")

tn.write(b"ls\n")
tn.write(b"exit\n")

print(tn.read_all().decode('ascii'))
```

21.20。 uuid- 根据RFC 4122的UUID对象

源代码： Lib / uuid.py

该模块提供不可变的UUID对象（ UUID类）和功能uuid1() ， uuid3() ， uuid4() ， uuid5()如在指定用于生成版本1，3，4和5点的UUIDRFC 4122。

如果你想要的只是一个唯一的ID，你应该打电话uuid1()或 uuid4()。请注意，这uuid1()可能会危害隐私，因为它会创建一个包含计算机网络地址的UUID。 uuid4()创建一个随机的UUID。

class uuid.UUID（ *hex = None* ，*bytes = None* ，*bytes_le = None* ，*fields = None* ，*int = None* ，*version = None* ）

从32个十六进制数字的字符串中创建一个UUID，一个16字节的字符串作为*字节*参数，一个16字节的字符串，以little-endian顺序作为*bytes_le*参数，一个由6个整数组成的元组（32位*time_low*，16- 作为*字段*参数，或者作为*int*参数的单个128位整数，作为*字段*参数的位时间*_mid_* ，16位*time_hi_version* ，8位*clock_seq_hi_variant* ，8位 *clock_seq_low* ，48位 *节点*）。当给出一串十六进制数字时，大括号，连字符和URN前缀都是可选的。例如，这些表达式都产生相同的UUID：

```
UUID('{12345678-1234-5678-1234-567812345678}')
UUID('12345678123456781234567812345678')
UUID('urn:uuid:12345678-1234-5678-1234-567812345678')
UUID(bytes=b'\x12\x34\x56\x78'*4)
UUID(bytes_le=b'\x78\x56\x34\x12\x34\x12\x78\x56' +
              b'\x12\x34\x56\x78\x12\x34\x56\x78')
UUID(fields=(0x12345678, 0x1234, 0x5678, 0x12, 0x34, 0x567812345678))
UUID(int=0x12345678123456781234567812345678)
```

正好一个*十六进制*，*字节*，*bytes_le*，*字段*，或*INT*必须给予。该*版本的*参数是可选的; 如果给定的，所得到的UUID将具有其变体和版本号根据RFC 4122设置，从而覆盖在给定的比特*十六进制*，*字节*，*bytes_le*，*字段*，或*INT*。

通过比较它们的UUID.int属性来比较UUID对象 。与非UUID对象的比较引发了一个问题TypeError。

str(uuid) 12345678-1234-5678-1234-567812345678以32位十六进制数字表示UUID 的形式返回一个字符串 。

UUID 实例具有这些只读属性：

UUID.bytes

UUID作为一个16字节的字符串（包含以big-endian字节顺序的六个整数字段）。

UUID.bytes_le

UUID作为16字节的字符串（以 little-endian字节顺序包含*time_low* ， *time_mid*和*time_hi_version* ）。

UUID.**fields**

UUID的六个整数字段的元组，它们也可用作六个单独的属性和两个派生属性：

领域	含义
time_low	UUID的前32位
time_mid	UUID的接下来的16位
time_hi_version	UUID的接下来的16位
clock_seq_hi_variant	UUID的接下来的8位
clock_seq_low	UUID的接下来的8位
node	UUID的最后48位
time	60位时间戳
clock_seq	14位序列号

UUID.**hex**

UUID作为32个字符的十六进制字符串。

UUID.**int**

UUID是一个128位整数。

UUID.**urn**

UUID作为RFC4122中规定的URN。

UUID.**variant**

UUID变体，它确定UUID的内部布局。这将是一个常量RESERVED_NCS，RFC_4122，RESERVED_MICROSOFT，或RESERVED_FUTURE。

UUID.**version**

UUID版本号（1到5，仅在变体时才有意义RFC_4122）。

该uuid模块定义了以下功能：

uuid.**getnode**()

获取硬件地址为48位正整数。这是第一次运行，它可能会启动一个单独的程序，可能会很慢。如果所有尝试获取硬件地址都失败，我们选择一个随机的48位数字，其第8位设置为1，如RFC 4122中推荐的那样。“硬件地址”表示网络接口的MAC地址，以及具有多个网络接口可以返回其中任何一个的MAC地址。

uuid.**uuid1**(*node = None*, *clock_seq = None*)

从主机ID，序列号和当前时间生成一个UUID。如果 没有给出*节点*，getnode()则用于获取硬件地址。如果 给出*clock_seq*，它将用作序列号; 否则选择一个随机的14位序列号。

uuid.**uuid3**(*名称空间*, *名称*)

根据名称空间标识（这是一个UUID）和一个名称（它是一个字符串）的MD5散列生成一个UUID。

uuid.**uuid4**()
生成一个随机的UUID。

uuid.**uuid5**(*名称空间*, *名称*)
根据名称空间标识（这是一个UUID）和名称（它是一个字符串）的SHA-1散列生成一个UUID。

该uuid模块定义了以下用于uuid3()或的名称空间标识符 uuid5()。

uuid.**NAMESPACE_DNS**
当指定此名称空间时，*名称*字符串是完全限定的域名。

uuid.**NAMESPACE_URL**
当这个名字空间被指定时，*名字*字符串就是一个URL。

uuid.**NAMESPACE_OID**
当这个名字空间被指定时，*名字*字符串就是一个ISO OID。

uuid.**NAMESPACE_X500**
当指定此名称空间时，*名称*字符串是DER中的X.500 DN或文本输出格式。

该uuid模块为该variant属性的可能值定义了以下常量：

uuid.**RESERVED_NCS**
保留用于NCS兼容性。

uuid.**RFC_4122**
指定在中给出的UUID布局 **RFC 4122**。

uuid.**RESERVED_MICROSOFT**
保留用于Microsoft兼容性。

uuid.**RESERVED_FUTURE**
留作未来定义。

也可以看看:

RFC 4122 - 通用唯一标识符（UUID）URN命名空间
此规范为UUID定义了统一资源名称命名空间，UUID的内部格式以及生成UUID的方法。

21.20.1。示例

以下是uuid模块典型用法的一些示例：

```
>>> import uuid

>>> # make a UUID based on the host ID and current time
>>> uuid.uuid1()
```

```
UUID('a8098c1a-f86e-11da-bd1a-00112444be1e')

>>> # make a UUID using an MD5 hash of a namespace UUID and a name
>>> uuid.uuid3(uuid.NAMESPACE_DNS, 'python.org')
UUID('6fa459ea-ee8a-3ca4-894e-db77e160355e')

>>> # make a random UUID
>>> uuid.uuid4()
UUID('16fd2706-8baf-433b-82eb-8c7fada847da')

>>> # make a UUID using a SHA-1 hash of a namespace UUID and a name
>>> uuid.uuid5(uuid.NAMESPACE_DNS, 'python.org')
UUID('886313e1-3b8a-5372-9b90-0c9aee199e5d')

>>> # make a UUID from a string of hex digits (braces and hyphens ignored)
>>> x = uuid.UUID('{00010203-0405-0607-0809-0a0b0c0d0e0f}')

>>> # convert a UUID to a string of hex digits in standard form
>>> str(x)
'00010203-0405-0607-0809-0a0b0c0d0e0f'

>>> # get the raw 16 bytes of the UUID
>>> x.bytes
b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t\n\x0b\x0c\r\x0e\x0f'

>>> # make a UUID from a 16-byte string
>>> uuid.UUID(bytes=x.bytes)
UUID('00010203-0405-0607-0809-0a0b0c0d0e0f')
```

21.21。socketserver- 一个网络服务器的框架

源代码： Lib / socketserver.py

该socketserver模块简化了编写网络服务器的任务。

有四个基本的具体服务器类：

*类*socketserver.TCPServer（*server_address*，*RequestHandlerClass*，*bind_and_activate* = *True*）

这使用Internet TCP协议，该协议在客户端和服务器之间提供连续的数据流。如果*bind_and_activate*为true，构造函数会自动尝试调用server_bind()和server_activate()。其他参数传递给BaseServer基类。

*类*socketserver.UDPServer（*server_address*，*RequestHandlerClass*，*bind_and_activate* = *True*）

这使用数据报，这些数据报是离散的信息包，可能无序到达或在运输途中丢失。参数与“相同”TCPServer。

*类*socketserver.UnixStreamServer（*server_address*，*RequestHandlerClass*，*bind_and_activate* = *True*）

*类*socketserver.UnixDatagramServer（*server_address*，*RequestHandlerClass*，*bind_and_activate* = *True*）

这些较少使用的类与TCP和UDP类相似，但使用Unix域套接字; 它们在非Unix平台上不可用。参数与“相同”TCPServer。

这四个类*同步*处理请求; 每个请求必须完成才能开始下一个请求。如果每个请求需要很长时间才能完成，这是不合适的，因为它需要大量计算，或者因为它返回了很多客户端处理缓慢的数据。解决方案是创建一个单独的进程或线程来处理每个请求; 在ForkingMixIn和ThreadingMixIn混合型类可用于支持异步行为。

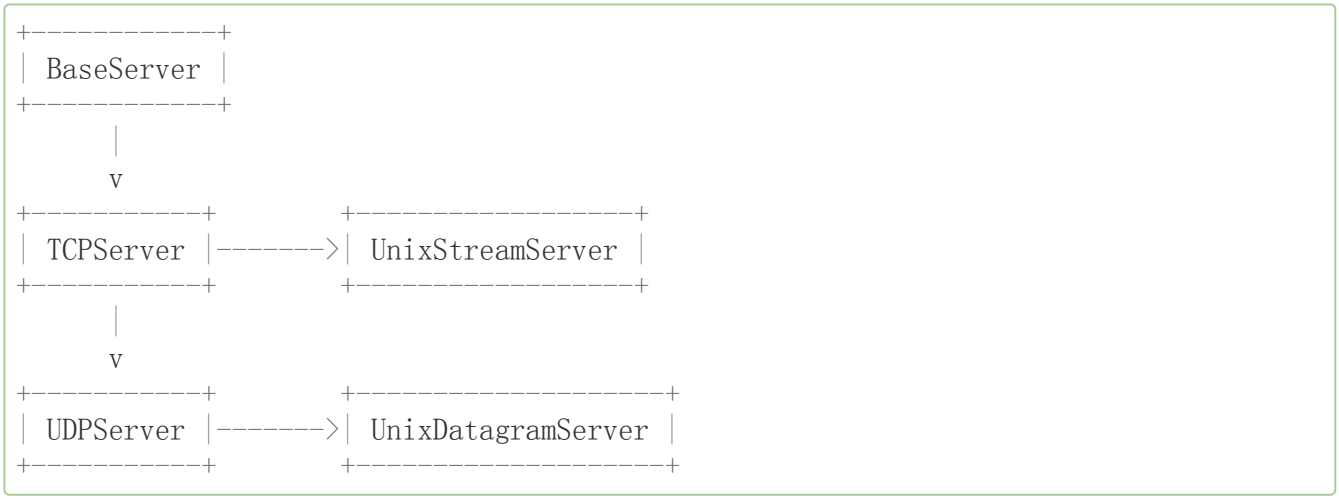
创建一个服务器需要几个步骤。首先，您必须创建一个请求处理程序类，方法是继承BaseRequestHandler该类并重写其handle()方法; 此方法将处理传入的请求。其次，您必须实例化其中一个服务器类，并将其传递给服务器的地址和请求处理程序类。建议在with声明中使用服务器。然后调用服务器对象的handle_request()or serve_forever()方法来处理一个或多个请求。最后，调用server_close()关闭套接字（除非你使用了一个with语句）。

从ThreadingMixIn线程连接行为继承时，应明确声明您希望线程在突然关闭时的行为方式。在ThreadingMixIn类定义的属性*daemon_threads*，这表明服务器是否应该等待线程终止。如果您希望线程自主行为，则应该明确设置该标志; 默认值是False，这意味着只有退出创建的所有线程后，Python才会ThreadingMixIn退出。

无论使用何种网络协议，服务器类都具有相同的外部方法和属性。

21.21.1。服务器创建说明

继承图中有五个类，其中四个代表四种类型的同步服务器：



请注意，UnixDatagramServer源自而UDPServer不是源自 UnixStreamServerIP和Unix流服务器之间的唯一区别是地址系列，这在两个Unix服务器类中都是重复的。

*类*socketserver.**ForkingMixIn**
*类*socketserver.**ThreadingMixIn**

使用这些混合类可以创建每种类型服务器的分叉和线程版本。例如，ThreadingUDPServer创建如下：

```
class ThreadingUDPServer(ThreadingMixIn, UDPServer):
    pass
```

混合类首先出现，因为它覆盖了中定义的方法 UDPServer。设置各种属性也会改变底层服务器机制的行为。

ForkingMixIn而下面提到的分叉类仅在支持的POSIX平台上可用fork()。

*类*socketserver.**ForkingTCPServer**
*类*socketserver.**ForkingUDPServer**
*类*socketserver.**ThreadingTCPServer**
*类*socketserver.**ThreadingUDPServer**

这些类是使用混合类预定义的。

要实现一个服务，你必须从中派生一个类BaseRequestHandler 并重新定义它的handle()方法。然后，您可以通过将其中一个服务器类与请求处理程序类相结合来运行各种版本的服务。对于数据报或流服务，请求处理程序类必须不同。这可以通过使用处理程序子类StreamRequestHandler或隐藏 DatagramRequestHandler。

当然，你还是要用你的头！例如，如果服务包含可以由不同请求修改的内存中的状态，则使用分叉服务器是没有意义的，因为子进程中的修改将永远达不到父进程中保留的初始状态并传递给每个子进程。在这种情况下，您可以使用线程服务器，但您可能必须使用锁来保护共享数据的完整性。

另一方面，如果您正在构建一个HTTP服务器，其中所有数据都存储在外部（例如，在文件系统中），则同时执行的类将基本上使服务“聋”，同时处理一个请求 - 可能用于很长一段时间，如果客户端收到所请求的所有数据的速度很慢。这里的线程或分叉服务器是合适的。

在某些情况下，可能需要同步处理部分请求，但要根据请求数据完成分叉子处理。这可以通过使用同步服务器并在请求处理程序类handle()方法中执行显式分支来实现。

在既不支持线程也不支持线程的环境中处理多个同时发生的请求的另一种方法fork()是维护一个部分完成的请求的明确表，并selectors用于决定下一个请求的哪个请求（或者是否处理新的传入请求）。这对于每个客户端可能长时间连接的流服务（如果线程或子进程无法使用）尤其重要。请参阅 asyncore另一种管理此方法。

21.21.2。服务器对象

*类*socketserver. BaseServer（*server_address*，*RequestHandlerClass* ）

这是模块中所有服务器对象的超类。它定义了下面给出的接口，但不实现大部分方法，这些方法在子类中完成。两个参数被存储在相应的 server_address和RequestHandlerClass属性。

fileno（ ）

返回服务器正在侦听的套接字的整数文件描述符。这个功能最常被传递到selectors，以允许在同一个进程中监视多个服务器。

handle_request（ ）

处理单个请求。这个函数调用下面的方法依次是：get_request()，verify_request()，和 process_request()。如果用户提供 handle()的处理程序类的handle_error()方法引发异常，则将调用服务器的方法。如果在timeout 几秒钟内没有收到请求，handle_timeout()则会被调用并handle_request() 返回。

serve_forever（*poll_interval = 0.5* ）

处理请求直到有明确的shutdown()请求。轮询每个*poll_interval*秒关闭。忽略该timeout属性。它还会调用service_actions()子类或mixin可用于提供特定于特定服务的操作。例如， ForkingMixIn该类service_actions()用来清理僵尸子进程。

*在版本3.3中更改：*添加service_actions了对该serve_forever方法的调用。

service_actions（ ）

这在serve_forever()循环中被调用。子类或mixin类可以重写此方法以执行特定于特定服务的操作，例如清理操作。

3.3版本的新功能

shutdown（ ）

告诉serve_forever()循环停止并等到它结束。

server_close（ ）

清理服务器。可能会被覆盖。

address_family

服务器套接字所属的协议族。常见的例子是socket.AF_INET和socket.AF_UNIX。

RequestHandlerClass

用户提供的请求处理程序类; 将为每个请求创建此类的一个实例。

server_address
服务器正在侦听的地址。地址格式因协议族而异; socket有关详细信息，请参阅该模块的文档。对于Internet协议，这是一个元组，其中包含一个给出地址的字符串和一个整数端口号：例如。('127.0.0.1', 80)

socket
服务器将侦听传入请求的套接字对象。

服务器类支持以下类变量：

allow_reuse_address
服务器是否允许重用地址。这默认为 False ，并且可以在子类中设置来更改策略。

request_queue_size
请求队列的大小。如果处理单个请求需要很长时间，则在服务器繁忙时到达的任何请求都将放入队列中，直至request_queue_size请求。一旦队列满了，来自客户端的进一步请求将会出现"连接被拒绝"错误。默认值通常是5，但这可以由子类覆盖。

socket_type
服务器使用的套接字的类型; socket.SOCK_STREAM并且 socket.SOCK_DGRAM是两个共同的价值。

timeout
超时持续时间，以秒为单位或者None如果不需要超时。如果handle_request()在超时期限内未收到任何传入请求， handle_timeout()则调用该方法。

有多种服务器方法可以被基类服务器类的子类覆盖，比如TCPServer ：这些方法对服务器对象的外部用户无用。

finish_request (*request* , *client_address*)
实际上通过实例化RequestHandlerClass和调用它的handle()方法来处理请求。

get_request ( )
必须接受来自套接字的请求，并返回包含 要用于与客户端通信的*新*套接字对象的2元组以及客户端的地址。

handle_error (*request* , *client_address*)
如果实例的handle() 方法RequestHandlerClass引发异常，则调用此函数。缺省操作是将追溯打印到标准错误并继续处理更多请求。

*在版本3.6中更改：*现在只调用派生自Exception 类的异常。

handle_timeout ( )
当该timeout属性被设置None为非超时值并超时时间已过且未收到请求时，将调用此函数。forking服务器的默认动作是收集已退出的任何子进程的状态，而在线程服务器中，此方法不执行任何操作。

process_request (*request* , *client_address*)

调用finish_request()来创建一个实例 RequestHandlerClass。如果需要，这个函数可以创建一个新的进程或线程来处理请求; 这个ForkingMixIn和 ThreadingMixIn类是这样做的。

server_activate ( )

由服务器的构造函数调用以激活服务器。TCP服务器的默认行为只是listen() 在服务器的套接字上调用。可能会被覆盖。

server_bind ( )

由服务器的构造函数调用以将套接字绑定到所需的地址。可能会被覆盖。

verify_request (*request* , *client_address*)

必须返回一个布尔值; 如果值是True ，请求将被处理，如果是False ，请求将被拒绝。该功能可以被覆盖以实现服务器的访问控制。默认实现总是返回True。

在版本3.6中进行了更改： 添加了对上下文管理器协议的支持。退出上下文管理器相当于调用server_close()。

21.21.3。请求处理程序对象

*类*socketserver. BaseRequestHandler

这是所有请求处理程序对象的超类。它定义了下面给出的接口。一个具体的请求处理子类必须定义一个新的handle()方法，并且可以重载其他任何方法。为每个请求创建一个新的子类实例。

setup ( )

在handle()方法执行任何初始化操作之前调用。默认实现什么都不做。

handle ( )

该功能必须完成服务请求所需的全部工作。默认实现什么都不做。有几个实例属性可用; 该请求可用作为self.request; 客户地址为self.client_address; 和服务器实例self.server ，以防万一它需要访问每个服务器的信息。

self.request数据报或流服务的类型不同。对于流服务， self.request是一个套接字对象; 对于数据报服务来说， self.request是一对字符串和套接字。

finish ( )

在handle()执行所需清理操作的方法后调用。默认实现什么都不做。如果setup() 引发异常，则不会调用该函数。

*类*socketserver. StreamRequestHandler

*类*socketserver. DatagramRequestHandler

这些 BaseRequestHandler 子类重写 setup() 和 finish() 方法，提供self.rfile 和self.wfile属性。该self.rfile和self.wfile属性可以被读取或写入，分别获得请求的数据或者数据返回给客户端。

这 rfile 两个类的属性都支持 io.BufferedIOBase 可读接口，并 DatagramRequestHandler.wfile支持 io.BufferedIOBase可写接口。

*在版本3.6中更改：*StreamRequestHandler.wfile还支持 io.BufferedIOBase可写接口。

21.21.4。示例

21.21.4.1。 socketserver.TCPServer示例

这是服务器端：

```
import socketserver

class MyTCPHandler(socketserver.BaseRequestHandler):
    """
    The request handler class for our server.

    It is instantiated once per connection to the server, and must
    override the handle() method to implement communication to the
    client.
    """

    def handle(self):
        # self.request is the TCP socket connected to the client
        self.data = self.request.recv(1024).strip()
        print("{} wrote:".format(self.client_address[0]))
        print(self.data)
        # just send back the same data, but upper-cased
        self.request.sendall(self.data.upper())

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999

    # Create the server, binding to localhost on port 9999
    with socketserver.TCPServer((HOST, PORT), MyTCPHandler) as server:
        # Activate the server; this will keep running until you
        # interrupt the program with Ctrl-C
        server.serve_forever()
```

另一个请求处理程序类，它使用流（通过提供标准文件接口来简化通信的文件类对象）：

```
class MyTCPHandler(socketserver.StreamRequestHandler):

    def handle(self):
        # self.rfile is a file-like object created by the handler;
        # we can now use e.g. readline() instead of raw recv() calls
        self.data = self.rfile.readline().strip()
        print("{} wrote:".format(self.client_address[0]))
        print(self.data)
        # Likewise, self.wfile is a file-like object used to write back
        # to the client
        self.wfile.write(self.data.upper())
```

区别在于readline()第二个处理程序中的调用会recv()多次调用 ，直到遇到换行符为止，而recv()第一个处理程序中的单个调用将仅返回一次sendall()调用中从客户端发送的内容。

这是客户端：

```
import socket
import sys

HOST, PORT = "localhost", 9999
data = " ".join(sys.argv[1:])

# Create a socket (SOCK_STREAM means a TCP socket)
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
    # Connect to server and send data
    sock.connect((HOST, PORT))
    sock.sendall(bytes(data + "\n", "utf-8"))

    # Receive data from the server and shut down
    received = str(sock.recv(1024), "utf-8")

print("Sent:     {}".format(data))
print("Received: {}".format(received))
```

示例的输出应该如下所示：

服务器：

```
$ python TCPServer.py
127.0.0.1 wrote:
b'hello world with TCP'
127.0.0.1 wrote:
b'python is nice'
```

客户：

```
$ python TCPClient.py hello world with TCP
Sent:     hello world with TCP
Received: HELLO WORLD WITH TCP
$ python TCPClient.py python is nice
Sent:     python is nice
Received: PYTHON IS NICE
```

21.21.4.2。socketserver.UDPServer示例

这是服务器端：

```
import socketserver

class MyUDPHandler(socketserver.BaseRequestHandler):
    """
    This class works similar to the TCP handler class, except that
    self.request consists of a pair of data and client socket, and since
    there is no connection the client address must be given explicitly
```

```
    when sending data back via sendto().
    """

    def handle(self):
        data = self.request[0].strip()
        socket = self.request[1]
        print("{} wrote:".format(self.client_address[0]))
        print(data)
        socket.sendto(data.upper(), self.client_address)

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999
    with socketserver.UDPServer((HOST, PORT), MyUDPHandler) as server:
        server.serve_forever()
```

这是客户端：

```
import socket
import sys

HOST, PORT = "localhost", 9999
data = " ".join(sys.argv[1:])

# SOCK_DGRAM is the socket type to use for UDP sockets
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# As you can see, there is no connect() call; UDP has no connections.
# Instead, data is directly sent to the recipient via sendto().
sock.sendto(bytes(data + "\n", "utf-8"), (HOST, PORT))
received = str(sock.recv(1024), "utf-8")

print("Sent:     {}".format(data))
print("Received: {}".format(received))
```

示例的输出应该与TCP服务器示例完全一样。

21.21.4.3。异步混合

要构建异步处理程序，请使用ThreadingMixIn和 ForkingMixIn类。

该ThreadingMixIn班的一个例子：

```
import socket
import threading
import socketserver

class ThreadedTCPRequestHandler(socketserver.BaseRequestHandler):

    def handle(self):
        data = str(self.request.recv(1024), 'ascii')
        cur_thread = threading.current_thread()
        response = bytes("{}: {}".format(cur_thread.name, data), 'ascii')
        self.request.sendall(response)

class ThreadedTCPServer(socketserver.ThreadingMixIn, socketserver.TCPServer):
```

```
    pass

def client(ip, port, message):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
        sock.connect((ip, port))
        sock.sendall(bytes(message, 'ascii'))
        response = str(sock.recv(1024), 'ascii')
        print("Received: {}".format(response))

if __name__ == "__main__":
    # Port 0 means to select an arbitrary unused port
    HOST, PORT = "localhost", 0

    server = ThreadedTCPServer((HOST, PORT), ThreadedTCPRequestHandler)
    with server:
        ip, port = server.server_address

        # Start a thread with the server -- that thread will then start one
        # more thread for each request
        server_thread = threading.Thread(target=server.serve_forever)
        # Exit the server thread when the main thread terminates
        server_thread.daemon = True
        server_thread.start()
        print("Server loop running in thread:", server_thread.name)

        client(ip, port, "Hello World 1")
        client(ip, port, "Hello World 2")
        client(ip, port, "Hello World 3")

        server.shutdown()
```

示例的输出应该如下所示：

```
$ python ThreadedTCPServer.py
Server loop running in thread: Thread-1
Received: Thread-2: Hello World 1
Received: Thread-3: Hello World 2
Received: Thread-4: Hello World 3
```

所述ForkingMixIn类被以相同的方式使用的，所不同的是，服务器将生成一个新的过程为每个请求。仅在支持的POSIX平台上可用fork()。

21.22。 http.server- HTTP服务器

源代码： Lib / http / server.py

该模块定义了用于实现HTTP服务器（Web服务器）的类。

一个类HTTPServer是，一个socketserver.TCPServer子类。它创建并侦听HTTP套接字，将请求分派给处理程序。创建和运行服务器的代码如下所示：

```
def run(server_class=HTTPServer, handler_class=BaseHTTPRequestHandler):
    server_address = ('', 8000)
    httpd = server_class(server_address, handler_class)
    httpd.serve_forever()
```

*类*http.server.HTTPServer（*server_address*，*RequestHandlerClass* ）

TCPServer该类通过将服务器地址存储为名为server_nameand的 实例变量来构建类server_port。服务器可以由处理程序访问，通常通过处理程序的server实例变量访问。

在HTTPServer必须赋予*RequestHandlerClass*上实例化，其中该模块提供了三种不同的变体：

class http.server.BaseHTTPRequestHandler（*request*，*client_address*，*server* ）

这个类用于处理到达服务器的HTTP请求。它本身不能响应任何实际的HTTP请求; 它必须被分类以处理每个请求方法（例如GET或POST）。 BaseHTTPRequestHandler提供了许多类和实例变量，以及子类使用的方法。

处理程序将解析请求和头文件，然后调用特定于请求类型的方法。方法名称由请求构成。例如，对于请求方法SPAM，该do_SPAM()方法将被调用，不带任何参数。所有相关信息都存储在处理程序的实例变量中。子类不应该需要重写或扩展该__init__()方法。

BaseHTTPRequestHandler有以下实例变量：

client_address

包含引用客户地址的表单元组。(host, port)

server

包含服务器实例。

close_connection

在handle_one_request()返回之前应该设置的布尔值，指示是否可以预期另一个请求，或者应该关闭连接。

requestline

包含HTTP请求行的字符串表示形式。终止CRLF被剥离。这个属性应该被设置handle_one_request()。如果没有处理有效的请求行，则应将其设置为空字符串。

command

包含命令（请求类型）。例如，'GET'。

path
包含请求路径。

request_version
包含来自请求的版本字符串。例如，'HTTP/1.0'。

headers
保存由MessageClass类变量指定的类的实例。这个实例解析并管理HTTP请求中的头文件。该parse_headers()函数从 http.client用于分析所述报头和它要求的HTTP请求提供一个有效的**RFC 2822**风格的标题。

rfile
一个io.BufferedIOBase输入流，准备从可选输入数据的开始阅读。

wfile
包含用于将响应写回客户端的输出流。写入此流时必须正确遵守HTTP协议，以实现与HTTP客户端的成功互操作。

*版本3.6中更改：*这是一个io.BufferedIOBase流。

BaseHTTPRequestHandler 具有以下属性：

server_version
指定服务器软件版本。你可能想重写这个。格式是多个以空格分隔的字符串，其中每个字符串的格式为名称[/版本]。例如，'BaseHTTP/0.2'。

sys_version
包含Python系统版本，可以通过 version_string方法和server_version类变量使用。例如，'Python/1.4'。

error_message_format
指定应由send_error()方法用于为客户端构建错误响应的格式字符串。该字符串默认情况下会responses根据传递给的状态代码填充变量send_error()。

error_content_type
指定发送给客户端的错误响应的Content-Type HTTP标头。默认值是'text/html'。

protocol_version
这指定了响应中使用的HTTP协议版本。如果设置为 'HTTP/1.1'，服务器将允许HTTP持久连接;但是，您的服务器*必须*在其所有对客户端的响应中包含一个准确的Content-Length 标题（使用send_header()）。为了向后兼容，该设置默认为'HTTP/1.0'。

MessageClass
指定一个email.message.Message类似的类来解析HTTP标头。通常情况下，这不会被覆盖，并且默认为 http.client.HTTPMessage。

responses

该属性包含错误代码整数到包含短消息和长消息的两元素元组的映射。例如，。的*短消息*通常被用作所述*消息*中的错误响应密钥，并且*longmessage*作为*解释*键。它被使用和方法。{code: (shortmessage, longmessage)} send_response_only() send_error()

一个BaseHTTPRequestHandler实例有以下方法：

handle ( )

调用handle_one_request()一次（或者，如果启用了持续连接，则会多次）来处理传入的HTTP请求。你永远不需要重写它; 相反，实施适当的do_*() 方法。

handle_one_request ( )

此方法将解析并将请求分发给适当的 do_*()方法。你永远不需要重写它。

handle_expect_100 ( )

当符合HTTP / 1.1的服务器收到一个 请求头时，它会回应一个后面的头。如果服务器不希望客户端继续，则可以重写此方法以引发错误。例如，服务器可以选择发送作为响应标题和。 Expect: 100-continue100 Continue200 OK417 Expectation Failedreturn False

3.2版本中的新功能

send_error (*code*，*message* = *None*，*explain* = *None*)

将完整的错误回复发送并记录到客户端。数字*代码* 指定HTTP错误代码，并将*消息*作为可选的简短的人为可读的错误描述。的*解释*参数可以用来提供有关错误的更详细的信息; 它将使用该error_message_format属性进行格式化，并在完整的标题集之后作为响应主体发送。该responses 属性保存*消息*的默认值并*解释*如果没有提供值将被使用; 对于未知代码，两者的默认值都是字符串???。身体将是空的，如果该方法是HEAD或响应代码是下列之一：1xx ， ，204 No Content205 Reset Content ，。 304 Not Modified

*在版本3.4中更改：*错误响应包含一个Content-Length标头。添加了*解释*参数。

send_response (*code*，*message* = *None*)

将响应头添加到头缓冲区并记录接受的请求。HTTP响应行写入内部缓冲区，然后写入*服务器*和*日期*标题。这两个头文件的值分别从version_string() 和date_time_string()方法中获取。如果服务器不打算使用该send_header()方法发送任何其他头文件，send_response()则应在后面跟一个end_headers() 调用。

*版本3.3中更改：*标题存储在内部缓冲区中，end_headers() 需要明确调用。

send_header (*关键字*，*值*)

将HTTP头添加到一个内部缓冲区，当调用end_headers()或者flush_headers()被调用时将被写入输出流。*关键字*应该指定的标题关键字，具有*值*指定其值。请注意，在send_header调用完成之后， end_headers()务必调用以完成操作。

*在版本3.2中更改：*标题存储在内部缓冲区中。

send_response_only (*code*，*message* = *None*)

仅发送响应标头，用于响应由服务器发送到客户端的目的。标题不缓冲，并直接发送stream.If的输出*消息*没有被指定，对应的响应的HTTP消息*代码*被发送。100 Continue

3.2版本中的新功能

end_headers ( )

在头缓冲区和调用中添加一个空行（表示响应中HTTP头的结尾）flush_headers()。

*在版本3.2中进行了更改：*将缓冲的标头写入输出流。

flush_headers ( )

最后将头部发送到输出流并刷新内部头部缓冲区。

3.3版本的新功能

log_request (*code* ='-', *size* ='-')

记录已接受（成功）的请求。*代码*应该指定与响应关联的数字HTTP代码。如果响应的大小可用，那么它应该作为*大小*参数传递。

log_error (...)

当请求无法完成时记录错误。默认情况下，它将消息传递给log_message()，因此它采用相同的参数（*格式*和附加值）。

log_message (*格式*, ...)

将任意消息记录到sys.stderr。这通常被重写以创建自定义错误记录机制。的*格式*的参数是一个标准的printf风格格式的字符串，其中所述附加参数 log_message()被应用作为输入的格式。客户端IP地址和当前日期和时间以每个记录的消息为前缀。

version_string ( )

返回服务器软件的版本字符串。这是server_version和sys_version属性的组合 。

date_time_string (*timestamp = None*)

返回由*时间戳记*给出的日期和时间（必须是None或以格式返回的格式time.time()），格式化为邮件标题。如果*时间戳*被省略，它使用当前的日期和时间。

结果看起来像。'Sun, 06 Nov 1994 08:49:37 GMT'

log_date_time_string ( )

返回格式化为日志记录的当前日期和时间。

address_string ( )

返回客户端地址。

*在版本3.3中更改：*以前，执行名称查找。为避免名称解析延迟，它现在总是返回IP地址。

class http.server.SimpleHTTPRequestHandler (*request*, *client_address*, *server*)

该类提供当前目录和下面的文件，直接将目录结构映射到HTTP请求。

很多工作，例如解析请求，都是由基类完成的 BaseHTTPRequestHandler。这个类实现 do_GET() 和do_HEAD()功能。

以下定义为以下类级别的属性 SimpleHTTPRequestHandler：

server_version

这将是，其中在模块级被定义。"SimpleHTTP/" + __version__ __version__

extensions_map

字典映射后缀为MIME类型。缺省值由空字符串表示，并被认为是 application/octet-stream。该映射不区分大小写使用，因此只应包含小写的键。

的SimpleHTTPRequestHandler类定义了下列方法：

do_HEAD ( )

这个方法服务于'HEAD'请求类型：它发送它将发送的等价GET请求头。有关do_GET() 可能的标题的更完整说明，请参阅该方法。

do_GET ( )

通过将请求解释为相对于当前工作目录的路径，该请求被映射到本地文件。

如果请求被映射到目录，则检查目录中名为index.html or 的文件index.htm（按该顺序）。如果找到，则返回文件的内容; 否则通过调用该list_directory()方法生成目录列表。此方法用于 os.listdir()扫描目录，404如果listdir()失败则返回错误响应。

如果请求被映射到一个文件，它将被打开并返回内容。任何OSError在打开的pool异常映射到一个404，错误。否则，通过调用方法来猜测内容类型，该方法依次使用 *extensions_map*变量。'File not found' guess_type()

将'Content-type:'输出一个包含猜测内容类型的'Content-Length:'标题，后跟一个包含文件大小的'Last-Modified:'标题和一个包含文件修改时间的 标题。

然后跟着一个空白行表示标题的结尾，然后输出文件的内容。如果文件的MIME类型text/以文本模式打开文件开头；否则使用二进制模式。

例如用法，请参阅模块中test()函数调用的http.server实现。

该SimpleHTTPRequestHandler班可在下列方式使用以创建相对于当前目录下的文件服务一个非常基本的web服务器：

```
import http.server
import socketserver

PORT = 8000

Handler = http.server.SimpleHTTPRequestHandler

with socketserver.TCPServer(("", PORT), Handler) as httpd:
    print("serving at port", PORT)
    httpd.serve_forever()
```

http.server也可以使用-m 带有参数的解释器的开关直接调用。与前面的例子类似，这为相对于当前目录的文件提供服务：port number

```
python -m http.server 8000
```

默认情况下，服务器将自身绑定到所有接口。该选项-b/--bind 指定了它应该绑定的特定地址。例如，以下命令只会导致服务器绑定到本地主机：

```
python -m http.server 8000 --bind 127.0.0.1
```

--bind引入了*3.4版本中的新功能：*参数。

class http.server.CGIHTTPRequestHandler（*request*，*client_address*，*server* ）

该类用于从当前目录和下面提供CGI脚本的文件或输出。请注意，将HTTP层次结构映射到本地目录结构完全如SimpleHTTPRequestHandler。

> **注意：** CGI脚本运行的脚本CGIHTTPRequestHandler无法执行重定向（HTTP代码302），因为代码200（脚本输出如下）在执行CGI脚本之前发送。这会抢占状态码。

然而，该类将运行CGI脚本，而不是将其作为文件提供，如果它猜测它是CGI脚本。只有基于目录的CGI被使用 - 其他常见的服务器配置是将特殊扩展视为表示CGI脚本。

该do_GET()和do_HEAD()功能被修改为运行CGI脚本和服务输出，而不是服务文件，如果该请求导致下面的地方cgi_directories道路。

在CGIHTTPRequestHandler定义了以下数据成员：

cgi_directories

这个默认和描述的目录被视为包含CGI脚本。['/cgi-bin', '/htbin']

在CGIHTTPRequestHandler定义了以下方法：

do_POST（ ）

该方法服务于'POST'请求类型，只允许CGI脚本。错误501，“只能POST到CGI脚本”，在尝试POST到非CGI URL时输出。

请注意，出于安全原因，CGI脚本将以U nobody用户nobody运行。CGI脚本的问题将被转换为错误403。

CGIHTTPRequestHandler可以通过传递--cgi选项在命令行中启用：

```
python -m http.server --cgi 8000
```

21.23。http.cookies- HTTP状态管理

源代码： Lib / http / cookies.py

该http.cookies模块定义了用于抽象cookie（HTTP状态管理机制）概念的类。它支持简单的仅字符串的cookie，并提供了一个抽象，让任何可序列化的数据类型作为cookie值。

该模块以前严格应用了。中描述的解析规则 **RFC 2109**和**RFC 2068**规范。自从发现MSIE 3.0x不遵循这些规范中概述的字符规则，并且当处理Cookie时，许多当前的浏览器和服务器也放宽了解析规则。因此，所使用的解析规则不太严格。

的字符集，string.ascii_letters，string.digits和 !#$%&'*+-.^_`|~:表示所述一组由该模块中的Cookie名称（如允许有效字符key）。

*在版本3.3中更改：*允许'：'作为有效的Cookie名称字符。

注意： 在遇到无效cookie时，CookieError如果您的cookie数据来自浏览器，则应始终准备无效数据并CookieError解析。

*异常*http.cookies.CookieError
由于例外而失败 **RFC 2109**无效：不正确的属性，错误的 *Set-Cookie*头等。

class http.cookies.BaseCookie（[*input*]）
这个类是一个类似字典的对象，其键是字符串，其值是Morsel实例。请注意，将某个键设置为某个值后，该值将首先转换为Morsel包含键和值的值。

如果给出*输入*，则将其传递给load()方法。

class http.cookies.SimpleCookie（[*input*]）
这个类派生自BaseCookie和覆盖value_decode()，value_encode()并str()分别是身份和身份。

也可以看看:

模 http.cookiejar
针对Web *客户端的*HTTP cookie处理。在http.cookiejar与 http.cookies模块不互相依赖。

RFC 2109 - HTTP状态管理机制
这是该模块实施的州管理规范。

21.23.1。Cookie对象

BaseCookie.value_decode（*val*）

从字符串表示中返回一个解码值。返回值可以是任何类型。这个方法没有做任何事 BaseCookie- 它存在，所以它可以被覆盖。

BaseCookie.value_encode（*val*）

返回一个编码值。*val*可以是任何类型，但返回值必须是字符串。这个方法没有做任何事 BaseCookie- 它存在，所以它可以被覆盖。

一般来说，它应该是这样的情况，value_encode()并且 value_decode()在*value_decode*的范围上是*相反的*。

BaseCookie.output（*attrs = None*，*header ='Set-Cookie：'*，*sep ='\ r \ n'*）

返回适合作为HTTP标头发送的字符串表示。*ATTRS*和 *报头*被发送到每一个Morsel 的 output()方法。*sep*用于将标头连接在一起，默认情况下是组合'\r\n' （CRLF）。

BaseCookie.js_output（*attrs =无*）

返回一个嵌入式JavaScript代码片段，如果在支持JavaScript的浏览器上运行，则会像HTTP头文件发送一样运行。

*attrs*的含义与英文中的相同output()。

BaseCookie.load（*rawdata*）

如果*rawdata*是一个字符串，则将其解析为一个字符串HTTP_COOKIE并将其中的值添加为Morsels。如果是字典，则相当于：

```
for k, v in rawdata.items():
    cookie[k] = v
```

21.23.2。一口对象

*类*http.cookies.Morsel

摘要一个键/值对，它有一些 **RFC 2109**属性。

Morsels是类似字典的对象，其键集是不变的 - 是有效的 **RFC 2109**属性，它们是

- expires
- path
- comment
- domain
- max-age
- secure
- version
- httponly

该属性httponly指定cookie仅在HTTP请求中传输，并且无法通过JavaScript访问。这旨在缓解某些形式的跨站点脚本。

这些键不区分大小写，它们的默认值是''。

*改变在3.5版本中：*__eq__()现在需要key和value 考虑。

Morsel.value

Cookie的价值。

*自3.5版以来已弃用：*分配给value；set()改为使用。

Morsel.coded_value

Cookie的编码值 - 这是应该发送的内容。

*自3.5版以来已弃用：*分配给coded_value；set()改为使用。

Morsel.key

Cookie的名称。

*自3.5版以来已弃用：*分配给key；set()改为使用。

Morsel.set（*key*，*value*，*coded_value* ）

设置*关键字*，*值*和*coded_value*属性。

*自3.5版弃用：*未记录的*LegalChars*参数将被忽略，并将在未来版本中删除。

Morsel.isReservedKey（*K* ）

*K*是否是a的一组键Morsel。

Morsel.output（*attrs* = *None*，*header* ='*Set-Cookie*：' ）

返回Morsel的字符串表示形式，适合作为HTTP头发送。默认情况下，除非给出*attrs*，否则包含所有属性，在这种情况下，它应该是要使用的属性列表。*标题*是默认的 "Set-Cookie:"。

Morsel.js_output（*attrs* =*无* ）

返回一个可嵌入的JavaScript代码片段，如果在支持JavaScript的浏览器上运行，就会像HTTP头文件发送一样。

*attrs*的含义与英文中的相同output()。

Morsel.OutputString（*attrs* =*无* ）

返回表示Morsel的字符串，不包含任何周围的HTTP或JavaScript。

*attrs*的含义与英文中的相同output()。

Morsel.update（*值* ）

用字典值中的值更新Morsel字典中的*值*。如果*值*词典中的任何键不是有效的，则引发错误**RFC 2109**属性。

*在版本3.5中更改：*无效键发生错误。

Morsel.copy（*价值* ）

返回Morsel对象的浅表副本。

*在版本3.5中更改：*返回一个Morsel对象而不是字典。

Morsel.setdefault(*key* , *value* = *None*)

如果密钥无效，则引发错误 **RFC 2109**属性，否则表现相同dict.setdefault()。

21.23.3。示例

以下示例演示如何使用该http.cookies模块。

```
>>> from http import cookies
>>> C = cookies.SimpleCookie()
>>> C["fig"] = "newton"
>>> C["sugar"] = "wafer"
>>> print(C) # generate HTTP headers
Set-Cookie: fig=newton
Set-Cookie: sugar=wafer
>>> print(C.output()) # same thing
Set-Cookie: fig=newton
Set-Cookie: sugar=wafer
>>> C = cookies.SimpleCookie()
>>> C["rocky"] = "road"
>>> C["rocky"]["path"] = "/cookie"
>>> print(C.output(header="Cookie:"))
Cookie: rocky=road; Path=/cookie
>>> print(C.output(attrs=[], header="Cookie:"))
Cookie: rocky=road
>>> C = cookies.SimpleCookie()
>>> C.load("chips=ahoy; vienna=finger") # load from a string (HTTP header)
>>> print(C)
Set-Cookie: chips=ahoy
Set-Cookie: vienna=finger
>>> C = cookies.SimpleCookie()
>>> C.load('keebler="E=everybody; L=\\"Loves\\"; fudge=\\012;";')
>>> print(C)
Set-Cookie: keebler="E=everybody; L=\"Loves\"; fudge=\012;"
>>> C = cookies.SimpleCookie()
>>> C["oreo"] = "doublestuff"
>>> C["oreo"]["path"] = "/"
>>> print(C)
Set-Cookie: oreo=doublestuff; Path=/
>>> C = cookies.SimpleCookie()
>>> C["twix"] = "none for you"
>>> C["twix"].value
'none for you'
>>> C = cookies.SimpleCookie()
>>> C["number"] = 7 # equivalent to C["number"] = str(7)
>>> C["string"] = "seven"
>>> C["number"].value
'7'
>>> C["string"].value
'seven'
>>> print(C)
Set-Cookie: number=7
Set-Cookie: string=seven
```

21.24。http.cookiejar- HTTP客户端的Cookie处理

源代码： Lib / http / cookiejar.py

该http.cookiejar模块定义了用于自动处理HTTP Cookie的类。对于访问需要小型数据片断的*网站（cookie*），它可以通过来自Web服务器的HTTP响应在客户端机器上设置，然后在稍后的HTTP请求中返回到服务器。

普通的Netscape cookie协议和通过定义的协议 **RFC 2965**被处理。RFC 2965处理默认关闭。根据实际的“策略”， **RFC 2109** cookie被解析为Netscape cookies，随后被视为Netscape或RFC 2965 cookie。请注意，互联网上绝大多数的cookies都是Netscape cookies。http.cookiejar试图遵循的事实上的Netscape cookie协议（其从载于原始的Netscape规范基本上不同），包括采取注意到的max-age和port与RFC 2965引入cookie的属性。

注意： 在*Set-Cookie*和 *Set-Cookie2*头文件（例如。domain和expires）中找到的各种命名参数通常称为*属性*。为了区分它们与Python属性，本模块的文档使用术语 *cookie-attribute*代替。

该模块定义了以下例外情况：

*异常*http.cookiejar. LoadError

FileCookieJar在未能从文件加载cookie 时引发此异常的实例。 LoadError是的一个子类OSError。

在版本3.3中更改： LoadError是OSError取而代之的 一个子类IOError。

提供以下课程：

class http.cookiejar. CookieJar（*policy* = *None* ）

*策略*是实现CookiePolicy接口的对象。

在CookieJar类存储的HTTP cookies。它从HTTP请求中提取cookie，并在HTTP响应中返回它们。CookieJar实例会在必要时自动过期包含cookie。子类还负责存储和检索文件或数据库中的Cookie。

class http.cookiejar. FileCookieJar（*filename*，*delayload* = *None*，*policy* = *None* ）

*策略*是实现CookiePolicy接口的对象。对于其他参数，请参阅相应属性的文档。

A CookieJar可以从磁盘上的文件加载Cookie，也可以将Cookie保存到磁盘上的文件。在调用or 方法之前，**不会**从指定的文件加载 Cookies 。该类的子类在FileCookieJar子类和与Web浏览器的合作中进行了介绍。 load() revert()

*类*http.cookiejar. CookiePolicy

这个类负责决定每个cookie是否被接受/返回到服务器。

class http.cookiejar.DefaultCookiePolicy（*blocked_domains* = *None*，*allowed_domains* = *None*，*netscape* = *True*，*rfc2965* = *False*，*rfc2109_as_netscape* = *None*，*hide_cookie2* = *False*，*strict_domain* = *False*，*strict_rfc2965_unverifiable* = *True*，*strict_ns_unverifiable* = *False*，*strict_ns_domain* = *DefaultCookiePolicy.DomainLiberal*，*strict_ns_set_initial_dollar* = *False*，*strict_ns_set_path* = *False* ）

构造函数参数只能作为关键字参数传递。 *blocked_domains*是一系列域名，我们从不接受cookies，也不返回cookie。如果不是，则为*allowed_domains*None，这是我们接受和返回cookie的唯一域的序列。对于所有其他参数，请参阅文档CookiePolicy和DefaultCookiePolicy对象。

DefaultCookiePolicy实现Netscape和RFC 2965 cookie的标准接受/拒绝规则。默认情况下，根据RFC 2965规则处理RFC 2109 cookie（即，在版本Cookie属性为1 的*Set-Cookie*头中接收*的Cookie*）。但是，如果关闭RFC 2965处理或rfc2109_as_netscape将TrueRFC 2109 cookie由CookieJar实例“降级”为Netscape cookie，则通过将实例的version属性设置Cookie为0. DefaultCookiePolicy还提供了一些参数以允许对策略进行一些微调。

*类*http.cookiejar.Cookie

这个类代表Netscape，RFC 2109和RFC 2965 cookies。预计用户不会http.cookiejar构建自己的Cookie 实例。相反，如果必要的话，叫make_cookies()上一个 CookieJar实例。

也可以看看:

模 urllib.request
通过自动cookie处理打开URL。

模 http.cookies
HTTP cookie类，主要用于服务器端代码。在 http.cookiejar与http.cookies模块不互相依赖。

https://curl.haxx.se/rfc/cookie_spec.html
原始Netscape cookie协议的规范。虽然这仍然是主要的协议，但所有主要浏览器（和 http.cookiejar ）实现的'Netscape cookie协议'只与所勾画出的相似cookie_spec.html。

RFC 2109 - HTTP状态管理机制
已过时RFC 2965.使用版本= 1的*Set-Cookie*。

RFC 2965 - HTTP状态管理机制
修复了错误的Netscape协议。使用*Set-Cookie2*代替*Set-Cookie*。没有被广泛使用。

http://kristol.org/cookie/errata.html
未完成的勘误符合RFC 2965。

RFC 2964 - 使用HTTP状态管理

21.24.1。CookieJar和FileCookieJar对象

CookieJar对象支持用于遍历包含对象的迭代器协议Cookie。

CookieJar 有以下方法：

CookieJar.add_cookie_header(*请求*)

添加正确的*Cookie*标头以*请求*。

如果策略允许（即在rfc2965与hide_cookie2所述属性CookieJar的CookiePolicy实例分别是真和假），则*COOKIE2*头也被添加在适当的时候。

所述*请求*对象（通常是一个urllib.request..Request实例）必须支持方法get_full_url()，get_host()，get_type()，unverifiable()，has_header()，get_header()，header_items()，add_unredirected_header()和origin_req_host属性由如记录urllib.request。

*版本3.3中更改：请求*对象需要origin_req_host属性。取消对已弃用方法的依赖性get_origin_req_host()。

CookieJar.extract_cookies(*回应*，*请求*)

从HTTP *响应中*提取cookie 并将其存储在CookieJar策略允许的地方。

该CookieJar会寻找容许*的Set-Cookie*和 *设置*，*COOKIE2*在头*响应*参数，存储cookies适当（受CookiePolicy.set_ok()方法的批准）。

该*响应*对象（到一个呼叫的通常的结果 urllib.request.urlopen()，或类似的）应支持的info() 方法，它返回一个email.message.Message实例。

所述*请求*对象（通常是一个urllib.request.Request实例）必须支持方法get_full_url()，get_host()， unverifiable()，和origin_req_host属性，通过如记录urllib.request。该请求用于设置cookie属性的默认值，并用于检查是否允许设置cookie。

*版本3.3中更改：请求*对象需要origin_req_host属性。取消对已弃用方法的依赖性get_origin_req_host()。

CookieJar.set_policy(*政策*)

设置CookiePolicy要使用的实例。

CookieJar.make_cookies(*回应*，*请求*)

Cookie从*响应*对象中提取对象的返回序列。

请参阅文档以extract_cookies()获取*响应*和*请求*参数所需的接口。

CookieJar.set_cookie_if_ok(*cookie*，*请求*)

设置一个Cookie如果政策说可以这样做。

CookieJar.set_cookie(*cookie*)

设置一个Cookie，不用检查政策，看看是否应该设置。

CookieJar.clear([*domain*[，*path*[，*name*]]])

清除一些饼干。

如果没有参数调用，请清除所有的cookie。如果只有一个参数，则只有属于该*域的* cookie才会被删除。如果给出两个参数，则删除属于指定*域*和URL *路径的* Cookie 。如果有三个

参数，那么具有指定*域*，*路径*和 *名称*的cookie将被删除。

KeyError如果不存在匹配的cookie，则引发。

CookieJar.clear_session_cookies( )

丢弃所有会话cookie。

放弃所有包含的具有真实discard属性的cookie（通常是因为它们具有no max-age或expirescookie属性或明确的discardcookie属性）。对于交互式浏览器，会话结束通常对应于关闭浏览器窗口。

请注意，该save()方法无论如何不会保存会话cookie，除非您通过传递*true_default_discard*参数来另外提问。

FileCookieJar 实现以下附加方法：

FileCookieJar.save(*filename = None*，*ignore_discard = False*，*ignore_expires = False*)

将Cookie保存到文件。

这个基类引发了NotImplementedError。子类可能会使此方法未实现。

*文件名*是保存cookie的文件的名称。如果没有指定*filename*，self.filename则使用（默认值是传递给构造函数的值，如果有的话）；如果self.filename是None， ValueError被提出。

ignore_discard：保存即使被设置为丢弃的Cookie。*ignore_expires*：保存即使过期的cookies

如果文件已经存在，则该文件被覆盖，从而清除它包含的所有cookie。保存的cookie可以在以后使用load() or revert()方法恢复。

FileCookieJar.load(*filename = None*，*ignore_discard = False*，*ignore_expires = False*)

从文件加载cookie。

旧饼干保留，除非被新加载的饼干覆盖。

参数是至于save()。

指定的文件必须采用该类LoadError可以理解的格式，否则 会被引发。另外，OSError可能会提出，例如，如果该文件不存在。

*在版本3.3中更改：*IOError曾经被引用，它现在是一个别名OSError。

FileCookieJar.revert(*filename = None*，*ignore_discard = False*，*ignore_expires = False*)

清除所有Cookie并从保存的文件重新加载Cookie。

revert()可以提出同样的例外load()。如果出现故障，对象的状态不会改变。

FileCookieJar 实例具有以下公共属性：

FileCookieJar.**filename**
在其中保留Cookie的默认文件的文件名。该属性可能被分配给。

FileCookieJar.**delayload**
如果为true，则从磁盘中缓慢加载cookie。不应将此属性分配给。这只是一个暗示，因为这只会影响性能，而不会影响行为（除非磁盘上的cookie正在改变）。一个CookieJar对象可能会忽略它。FileCookieJar标准库中包含的任何类都不会延迟加载cookie。

21.24.2。FileCookieJar子类和与浏览器的合作

CookieJar提供以下子类用于阅读和写作。

class http.cookiejar.**MozillaCookieJar**（*filename*，*delayload* = *None*，*policy* = *None*）
一个FileCookieJar可以从Mozilla cookies.txt文件格式（也被Lynx和Netscape浏览器使用）加载和保存cookie到磁盘的软件。

> **注意：** 这会丢失有关RFC 2965 cookie的信息，以及有关更新或非标准cookie属性的信息，例如port。

> **警告：** 如果您的Cookie存在丢失/损坏不便的情况（有一些细微之处可能会导致文件在加载/保存往返时发生轻微变化），请在保存之前备份您的Cookie。

另外请注意，Mozilla运行时保存的cookie会被Mozilla破坏。

class http.cookiejar.**LWPCookieJar**（*filename*，*delayload* = *None*，*policy* = *None*）
A FileCookieJar可以以与libwww-perl库的Set-Cookie3文件格式兼容的格式加载和保存cookie到磁盘。如果您想将cookie存储在可读的文件中，这很方便。

21.24.3。的CookiePolicy对象

实现CookiePolicy接口的对象具有以下方法：

CookiePolicy.**set_ok**（*cookie*，*请求*）
返回指示是否应该从服务器接受cookie的布尔值。

*cookie*是一个Cookie实例。*请求*是实现由文档定义的接口的对象CookieJar.extract_cookies()。

CookiePolicy.**return_ok**（*cookie*，*请求*）
返回指示cookie是否应返回给服务器的布尔值。

*cookie*是一个Cookie实例。*请求*是实现由文档定义的接口的对象CookieJar.add_cookie_header()。

CookiePolicy.domain_return_ok(*域*, *请求*)

如果cookie不应该返回，则返回false。

这种方法是一种优化。它不需要检查每个具有特定域的cookie（这可能涉及读取许多文件）。从所有工作中回归真实domain_return_ok()并path_return_ok()离开return_ok()。

如果domain_return_ok()cookie域返回true， path_return_ok()则调用cookie路径。否则，path_return_ok()并且return_ok()永远不会调用该Cookie域。如果path_return_ok()返回true，return_ok()则与Cookie对象本身一起调用以进行全面检查。否则， return_ok()永远不会调用该cookie路径。

请注意，domain_return_ok()每个*Cookie*域都会调用它，而不仅仅是*请求*域。例如，如果请求域是".example.com"，"www.example.com"则可以使用两者调用该函数"www.example.com"。同样如此path_return_ok()。

所述*请求*参数是如记录return_ok()。

CookiePolicy.path_return_ok(*路径*, *请求*)

给定cookie路径，如果不应该返回cookie，则返回false。

请参阅文档domain_return_ok()。

除了实现上述方法外，CookiePolicy接口的实现 还必须提供以下属性，指示应使用哪些协议以及如何使用。所有这些属性都可以分配给。

CookiePolicy.netscape

实施Netscape协议。

CookiePolicy.rfc2965

实施RFC 2965协议。

CookiePolicy.hide_cookie2

不要将*Cookie2*头添加到请求中（该头的存在向服务器表明我们理解RFC 2965 Cookie）。

定义一个CookiePolicy类最有用的方法是通过子类化DefaultCookiePolicy并覆盖上面的一些或全部方法。 CookiePolicy本身可能被用作'空白策略'来允许设置和接收任何和所有的cookies（这不太可能有用）。

21.24.4。DefaultCookiePolicy对象

实现接受和返回cookie的标准规则。

覆盖了RFC 2965和Netscape cookies。RFC 2965处理默认关闭。

提供自己策略的最简单方法是在添加自己的附加检查之前，重写此类并在重写的实现中调用其方法：

```
import http.cookiejar
class MyCookiePolicy(http.cookiejar.DefaultCookiePolicy):
```

```
    def set_ok(self, cookie, request):
        if not http.cookiejar.DefaultCookiePolicy.set_ok(self, cookie, request):
            return False
        if i_dont_want_to_store_this_cookie(cookie):
            return False
        return True
```

除了实现CookiePolicy 界面所需的功能之外，该类还允许您阻止和允许域设置和接收Cookie。还有一些严格的开关，可以让你稍微收紧一些松散的Netscape协议规则（以阻止一些良性cookie为代价）。

提供域黑名单和白名单（默认情况下都是关闭的）。只有不在黑名单中并且出现在白名单中的域（如果白名单处于活动状态）参与cookie设置并返回。使用*blocked_domains* 构造函数参数 blocked_domains() 和 set_blocked_domains() 方法（以及*allowed_domains*的相应参数和方法）。如果您设置了白名单，则可以将其设置为关闭None。

块中的域或允许不以点开头的列表必须等于要匹配的cookie域。例如，"example.com"匹配黑名单条目"example.com"，但"www.example.com"不匹配。以点开头的域也与更具体的域匹配。例如，两者 "www.example.com" 和 "www.coyote.example.com" 匹配 ".example.com"（但"example.com"本身不）。IP地址是一个例外，并且必须完全匹配。例如，如果blocked_domains包含"192.168.1.2" and ".168.1.2"，则192.168.1.2被阻塞，但193.168.1.2不是。

DefaultCookiePolicy 实现以下附加方法：

DefaultCookiePolicy.blocked_domains()
返回阻塞域的序列（作为元组）。

DefaultCookiePolicy.set_blocked_domains(*blocked_domains*)
设置阻止的域的顺序。

DefaultCookiePolicy.is_blocked(*域*)
返回*域名*是否在黑名单上，用于设置或接收Cookie。

DefaultCookiePolicy.allowed_domains()
返回None或允许的域序列（作为元组）。

DefaultCookiePolicy.set_allowed_domains(*allowed_domains*)
设置允许的域的序列，或None。

DefaultCookiePolicy.is_not_allowed(*域*)
返回*域名*是否不在白名单上，用于设置或接收cookie。

DefaultCookiePolicy 实例具有以下属性，这些属性都是从具有相同名称的构造函数参数中初始化的，并且可以全部分配给它们。

DefaultCookiePolicy.rfc2109_as_netscape
如果为true，则通过将CookieJar实例的版本属性设置为0，请求实例将RFC 2109 cookie（即通过版本cookie属性为1 的*Set-Cookie*头中收到的cookie）降级到Netscape

cookie `Cookie`。默认值为`None`，在这种情况下，RFC 2109 cookie会被降级，当且仅当RFC 2965处理被关闭。因此，默认情况下RFC 2109 cookie会降级。

一般严格开关：

`DefaultCookiePolicy.strict_domain`
不要让站点设置两个组成结构域与像国家代码顶级域名`.co.uk`，`.gov.uk`，`.co.nz`.etc。这远远不够完美，并不能保证工作！

RFC 2965协议严格性开关：

`DefaultCookiePolicy.strict_rfc2965_unverifiable`
按照关于无法验证的事务的RFC 2965规则（通常，不可验证的事务是由重定向或托管在另一个站点上的映像的请求导致的事务）。如果这是错误的，基于可验证性，cookies *不会*被阻塞

Netscape协议严格切换：

`DefaultCookiePolicy.strict_ns_unverifiable`
将RFC 2965规则应用于不可核实的交易，甚至应用于Netscape Cookie。

`DefaultCookiePolicy.strict_ns_domain`
标志表明对于Netscape cookie的域匹配规则有多严格。请参阅下面的可接受值。

`DefaultCookiePolicy.strict_ns_set_initial_dollar`
忽略Set-Cookie中的Cookie：名称以名称开头的标头`'$'`。

`DefaultCookiePolicy.strict_ns_set_path`
不允许设置路径不匹配请求URI的Cookie。

`strict_ns_domain`是旗帜的集合。它的值由或者一起构造（例如，`DomainStrictNoDots|DomainStrictNonDomain`意味着两个标志都被设置）。

`DefaultCookiePolicy.DomainStrictNoDots`
在设置cookie时，'主机前缀'不能包含一个点（例如`www.foo.bar.com`，由于`.bar.com`包含一个点而`www.foo`不能设置cookie）。

`DefaultCookiePolicy.DomainStrictNonDomain`
没有明确指定`domain`cookie属性的cookie只能返回到与设置cookie的域相同的域（例如，`spam.example.com`不会返回`example.com`没有`domain`cookie属性的cookie）。

`DefaultCookiePolicy.DomainRFC2965Match`
设置cookie时，需要完整的RFC 2965域名匹配。

提供以下属性是为了方便起见，它们是上述标志最有用的组合：

`DefaultCookiePolicy.DomainLiberal`
相当于0（即所有上述Netscape域严格标志关闭）。

`DefaultCookiePolicy.DomainStrict`

相当于DomainStrictNoDots|DomainStrictNonDomain。

21.24.5。Cookie对象

Cookie实例的Python属性大致对应于各种Cookie标准中指定的标准Cookie属性。通信不是一一对应的，因为复杂的默认值分配规则很复杂，因为max-age和expires cookie属性包含等效信息，并且由于RFC 2109 cookie可能会http.cookiejar从版本1 “降级” 到版本0（Netscape）饼干。

对于这些属性的分配不应该是必要的，除非在极少数情况下CookiePolicy。班级不强制内部一致性，所以如果你这样做，你应该知道你在做什么。

Cookie.**version**
整数或None。Netscape cookies有version0. RFC 2965和RFC 2109 cookie的versioncookie属性为1.但是，请注意，http.cookiejar可能会将RFC 2109 cookie降级为Netscape cookie，在这种情况下version为0。

Cookie.**name**
Cookie名称（一个字符串）。

Cookie.**value**
Cookie值（字符串）或None。

Cookie.**port**
代表一个端口或一组端口的字符串（例如'80'或'80,8080'），或者None。

Cookie.**path**
Cookie路径（例如字符串’/acme/rocket_launchers’）。

Cookie.**secure**
True如果cookie只能通过安全连接返回。

Cookie.**expires**
自epoch开始，整数失效日期或以秒为单位None。另见该is_expired()方法。

Cookie.**discard**
True如果这是一个会话cookie。

Cookie.**comment**
来自服务器的解释此cookie功能的字符串注释，或None。

Cookie.**comment_url**
链接到来自服务器的解释此cookie功能的评论的URL，或None。

Cookie.**rfc2109**
True如果此cookie是作为RFC 2109 cookie收到的（即cookie到达*Set-Cookie*标头，并且该标头中的Version cookie属性值为1）。提供此属性是因为http.cookiejar可能会将RFC 2109 cookie降级为Netscape cookie，在这种情况下version为0。

Cookie.port_specified
True如果一个端口或一组端口被服务器明确指定（在 *Set-Cookie* / *Set-Cookie2*头文件中）。

Cookie.domain_specified
True 如果域是由服务器明确指定的。

Cookie.domain_initial_dot
True如果服务器明确指定的域以点（'.'）开头。

Cookie可能会有其他非标准Cookie属性。这些可以通过以下方法访问：

Cookie.has_nonstandard_attr（*名字*）
如果cookie具有指定的cookie属性，则返回true。

Cookie.get_nonstandard_attr（*名称*，*默认=无*）
如果cookie具有指定的cookie属性，则返回其值。否则，返回 *默认值*。

Cookie.set_nonstandard_attr（*名称*，*值*）
设置指定的cookie属性的值。

的Cookie类也定义了下面的方法：

Cookie.is_expired（*now = None* ）
True如果cookie已超过服务器请求的时间，则应该过期。如果*现在*给出（从纪元开始以秒计），返回cookie是否在指定时间到期。

21.24.6。示例

第一个例子显示了最常用的用法http.cookiejar：

```
import http.cookiejar, urllib.request
cj = http.cookiejar.CookieJar()
opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

这个例子说明了如何使用你的Netscape，Mozilla或者Lynx cookies（假定用于定位cookies文件的Unix / Netscape约定）来打开一个URL：

```
import os, http.cookiejar, urllib.request
cj = http.cookiejar.MozillaCookieJar()
cj.load(os.path.join(os.path.expanduser("~"), ".netscape", "cookies.txt"))
opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

下一个例子说明了使用DefaultCookiePolicy。打开RFC 2965 cookie，在设置和返回Netscape cookies时对域进行更严格的限制，并阻止某些域设置cookie或让它们返回：

```
import urllib.request
from http.cookiejar import CookieJar, DefaultCookiePolicy
policy = DefaultCookiePolicy(
    rfc2965=True, strict_ns_domain=Policy.DomainStrict,
    blocked_domains=["ads.net", ".ads.net"])
cj = CookieJar(policy)
opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

21.25。 xmlrpc- XMLRPC服务器和客户端模块

XML-RPC是一种远程过程调用方法，它使用通过HTTP传递的XML作为传输。通过它，客户端可以使用远程服务器上的参数调用方法（服务器通过URI命名）并获取结构化数据。

xmlrpc是一个收集实现XML-RPC的服务器和客户端模块的包。这些模块是：

- xmlrpc.client
- xmlrpc.server

21.26。 xmlrpc.client- XML-RPC客户端访问

源代码： Lib / xmlrpc / client.py

XML-RPC是一种远程过程调用方法，它使用通过HTTP（S）传递的XML作为传输。通过它，客户端可以使用远程服务器上的参数调用方法（服务器通过URI命名）并获取结构化数据。该模块支持编写XML-RPC客户端代码; 它处理所有在线上一致的Python对象和XML之间的转换细节。

警告： 该xmlrpc.client模块对恶意构建的数据不安全。如果您需要解析不可信或未经身份验证的数据，请参阅 XML漏洞。

*在版本3.5中更改：*对于HTTPS URI，xmlrpc.client现在默认执行所有必需的证书和主机名检查。

class xmlrpc.client.ServerProxy（*uri*，*transport* = *None*，*encoding* = *None*，*verbose* = *False*，*allow_none* = *False*，*use_datetime* = *False*，*use_builtin_types* = *False*，*，*context* = *None* ）

*在3.3版本中更改：*将*use_builtin_types*加入标志。

甲ServerProxy实例是管理与远程XML-RPC服务器的通信的对象。所需的第一个参数是一个URI（统一资源指示符），通常是服务器的URL。可选的第二个参数是一个传输工厂实例; 默认情况下它是 SafeTransporthttps：URL的内部Transport实例，否则是内部HTTP实例。可选的第三个参数是一个编码，默认为UTF-8。可选的第四个参数是一个调试标志。

以下参数控制使用返回的代理实例。如果*allow_none*为true，则Python常量None将被转换为XML; 默认行为是为了None引发一个TypeError。这是XML-RPC规范的常用扩展，但并不受所有客户端和服务器的支持; 有关说明，请参阅 http://ontosys.com/xml-rpc/extensions.php 。所述*use_builtin_types*标志可用于引起日期/时间值被表示为datetime.datetime对象和要呈现为二进制数据bytes对象; 此标志默认为false。datetime.datetime，bytes并且bytearray可以将对象传递给调用。过时的*use_datetime*标志*与之*类似*use_builtin_types*，但它仅适用于日期/时间值。

HTTP和HTTPS传输都支持HTTP基本认证的URL语法扩展：http://user:pass@host:port/path。该user:pass部分将被base64编码为HTTP'授权'标头，并在调用XML-RPC方法时作为连接过程的一部分发送到远程服务器。如果远程服务器需要基本身份验证用户和密码，则只需使用它。如果提供了HTTPS URL，*上下文*可以ssl.SSLContext并配置基础HTTPS连接的SSL设置。

返回的实例是一个代理对象，其中的方法可用于在远程服务器上调用相应的RPC调用。如果远程服务器支持introspection API，则该代理还可用于向远程服务器查询其支持的方法（服务发现）并获取其他服务器关联的元数据。

可以整合的类型（例如，可以通过XML进行编组）的类型包括以下内容（除非另有说明，否则它们被解组为相同的Python类型）：

XML-RPC类型	Python类型
boolean	bool
int，i1，i2，i4，i8或biginteger	int范围从-2147483648到2147483647.值得到<int>标记。
double 要么 float	float。值得到<double>标签。
string	str
array	list或tuple包含适合的元素。数组返回为 lists。
struct	dict。键必须是字符串，值可以是任何适合的类型。用户定义的类的对象可以传入; 只有他们的 __dict__属性被传送。
dateTime.iso8601	DateTime或者datetime.datetime。返回的类型取决于*use_builtin_types*和*use_datetime*标志的值 。
base64	Binary，bytes或者 bytearray。返回的类型取决于*use_builtin_types*标志的值。
nil	该None常数。仅当*allow_none*为真时才允许传递 。
bigdecimal	decimal.Decimal。仅返回类型。

这是XML-RPC支持的全套数据类型。方法调用还可能引发一个特殊的Fault实例，用于发送XML-RPC服务器错误，或 ProtocolError用于在HTTP / HTTPS传输层中发出错误信号。双方Fault并ProtocolError从被称为基类派生 Error。请注意，xmlrpc客户端模块当前不编组内置子类的实例。

传递字符串时，XML等特殊字符（例如<，>）& 将被自动转义。但是，调用者有责任确保该字符串不含XML中不允许使用的字符，例如ASCII值介于0和31之间的控制字符（当然除了tab，newline和回车）; 如果不这样做将导致XML-RPC请求不是格式良好的XML。如果您必须通过XML-RPC传递任意字节，请使用下面描述的包装类bytes或bytearray类Binary。

Server被保留作为ServerProxy向后兼容的别名。新的代码应该使用ServerProxy。

*在版本3.5中更改：*添加了*上下文*参数。

*在版本3.6中进行了更改：*添加了对带有前缀（例如ex:nil）的类型标记的支持。新增支持unmarsalling为NUMERICS使用的Apache XML-RPC实现其他类型的：i1，i2，i8，biginteger，float和bigdecimal。有关说明，请参阅http://ws.apache.org/xmlrpc/types.html。

也可以看看:

XML-RPC HOWTO
用几种语言描述XML-RPC操作和客户端软件。几乎包含XML-RPC客户端开发人员需要了解的所有内容。

XML-RPC自省
介绍用于内省的XML-RPC协议扩展。

XML-RPC规范
官方规格。

非官方的XML-RPC勘误表

Fredrik Lundh的“非官方勘误表”，旨在澄清XML-RPC规范中的某些细节，并暗示在设计自己的XML-RPC实现时使用“最佳实践”。“

21.26.1。ServerProxy对象

甲ServerProxy实例具有对应于由XML-RPC服务器接受每个远程过程调用的方法。调用该方法将执行由名称和参数签名调度的RPC（例如，可以使用多个参数签名重载相同的方法名称）。该RPC完成通过返回一个值，在符合的类型或它们可以是返回的数据 Fault或ProtocolError对象指示错误。

支持XML内省API的服务器支持在保留system属性下分组的一些常用方法：

ServerProxy.system.**listMethods**（）

此方法返回一个字符串列表，XML-RPC服务器支持的每个（非系统）方法都有一个字符串列表。

ServerProxy.system.**methodSignature**（*名字*）

此方法接受一个参数，即由XML-RPC服务器实现的方法的名称。它为此方法返回一组可能的签名。签名是一个类型数组。这些类型中的第一个是方法的返回类型，其余的是参数。

由于允许多个签名（即重载），因此此方法返回签名列表而不是单例。

签名本身仅限于方法所期望的顶级参数。例如，如果一个方法需要一个结构数组作为参数，并且它返回一个字符串，那么它的签名就是“string，array”。如果期望三个整数并返回一个字符串，则其签名是“string，int，int，int”。

如果没有为该方法定义签名，则返回一个非数组值。在Python中，这意味着返回值的类型将不是列表。

ServerProxy.system.**methodHelp**（*名字*）

此方法接受一个参数，即由XML-RPC服务器实现的方法的名称。它返回一个描述该方法用法的文档字符串。如果没有这样的字符串可用，则返回空字符串。文档字符串可能包含HTML标记。

*在版本3.5中进行了更改：*ServerProxy支持关闭底层传输的上下文管理器协议的实例。

下面是一个工作示例。服务器代码：

```
from xmlrpc.server import SimpleXMLRPCServer

def is_even(n):
    return n % 2 == 0

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(is_even, "is_even")
server.serve_forever()
```

上述服务器的客户端代码：

```
import xmlrpc.client

with xmlrpc.client.ServerProxy("http://localhost:8000/") as proxy:
    print("3 is even: %s" % str(proxy.is_even(3)))
    print("100 is even: %s" % str(proxy.is_even(100)))
```

21.26.2。日期时间对象

*类*xmlrpc.client.DateTime

这个类可以在历元以来的秒数，时间元组，ISO 8601 时间/日期字符串或 datetime.datetime 实例中初始化。它具有以下方法，主要用于编组/解编码的内部使用：

decode (*字符串*)

接受一个字符串作为实例的新时间值。

encode (*out*)

将此DateTime项目的XML-RPC编码写入*输出*流对象。

它还通过丰富的比较和__repr__()方法支持某些Python的内置操作符。

下面是一个工作示例。服务器代码：

```
import datetime
from xmlrpc.server import SimpleXMLRPCServer
import xmlrpc.client

def today():
    today = datetime.datetime.today()
    return xmlrpc.client.DateTime(today)

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(today, "today")
server.serve_forever()
```

上述服务器的客户端代码：

```
import xmlrpc.client
import datetime

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")

today = proxy.today()
# convert the ISO8601 string to a datetime object
converted = datetime.datetime.strptime(today.value, "%Y%m%dT%H:%M:%S")
print("Today: %s" % converted.strftime("%d.%m.%Y, %H:%M"))
```

21.26.3。二进制对象

*类*xmlrpc.client.**Binary**

这个类可以从字节数据（可能包括NUL）初始化。Binary对象内容的主要访问由一个属性提供：

data

由Binary实例封装的二进制数据。数据是作为一个bytes对象提供的。

Binary 对象具有以下方法，主要用于编组/解编码的内部使用：

decode（*字节*）

接受一个base64 bytes对象并将其解码为实例的新数据。

encode（*out*）

将此二进制项的XML-RPC base 64编码写入*输出*流对象。

按照RFC 2045第6.8节的规定，编码后的数据每隔76个字符就会换行，这在编写XML-RPC规范时是事实上的标准base64规范。

它还支持某些Python的内置操作符__eq__() 和__ne__()方法。

二进制对象的示例用法。我们将通过XMLRPC传输图像：

```
from xmlrpc.server import SimpleXMLRPCServer
import xmlrpc.client

def python_logo():
    with open("python_logo.jpg", "rb") as handle:
        return xmlrpc.client.Binary(handle.read())

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(python_logo, 'python_logo')

server.serve_forever()
```

客户端获取图像并将其保存到文件中：

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
with open("fetched_python_logo.jpg", "wb") as handle:
    handle.write(proxy.python_logo().data)
```

21.26.4。故障对象

*类*xmlrpc.client.**Fault**

一个Fault对象封装了XML-RPC错误标签的内容。故障对象具有以下属性：

faultCode

指示故障类型的字符串。

faultString
包含与故障关联的诊断消息的字符串。

在下面的例子中，我们将Fault通过返回一个复杂的类型对象来故意造成一个。服务器代码：

```
from xmlrpc.server import SimpleXMLRPCServer

# A marshalling error is going to occur because we're returning a
# complex number
def add(x, y):
    return x+y+0j

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(add, 'add')

server.serve_forever()
```

上述服务器的客户端代码：

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
try:
    proxy.add(2, 5)
except xmlrpc.client.Fault as err:
    print("A fault occurred")
    print("Fault code: %d" % err.faultCode)
    print("Fault string: %s" % err.faultString)
```

21.26.5。ProtocolError对象

*类*xmlrpc.client.ProtocolError
甲ProtocolError对象描述底层传输层中的协议错误（例如404“未找到”错误，如果由URI指定的服务器不存在）。它具有以下属性：

url
触发错误的URI或URL。

errcode
错误代码。

errmsg
错误消息或诊断字符串。

headers
包含触发错误的HTTP / HTTPS请求标头的字典。

在下面的例子中，我们将ProtocolError 通过提供一个无效的URI 来故意造成一个错误：

```
import xmlrpc.client

# create a ServerProxy with a URI that doesn't respond to XMLRPC requests
proxy = xmlrpc.client.ServerProxy("http://google.com/")

try:
    proxy.some_method()
except xmlrpc.client.ProtocolError as err:
    print("A protocol error occurred")
    print("URL: %s" % err.url)
    print("HTTP/HTTPS headers: %s" % err.headers)
    print("Error code: %d" % err.errcode)
    print("Error message: %s" % err.errmsg)
```

21.26.6。MultiCall对象

该MultiCall对象提供了一种将多个调用封装到远程服务器到单个请求的方法[1]。

*类*xmlrpc.client.**MultiCall**(*服务器*)

创建一个用于boxcar方法调用的对象。*服务器*是呼叫的最终目标。可以对结果对象进行调用，但它们会立即返回None，并只将MultiCall对象名称和参数存储在 对象中。调用对象本身会导致所有存储的调用都作为单个system.multicall请求传输。这个调用的结果是一个发生器；遍历这个生成器会产生单独的结果。

这个类的用法示例如下。服务器代码：

```
from xmlrpc.server import SimpleXMLRPCServer

def add(x, y):
    return x + y

def subtract(x, y):
    return x - y

def multiply(x, y):
    return x * y

def divide(x, y):
    return x // y

# A simple server with simple arithmetic functions
server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_multicall_functions()
server.register_function(add, 'add')
server.register_function(subtract, 'subtract')
server.register_function(multiply, 'multiply')
server.register_function(divide, 'divide')
server.serve_forever()
```

上述服务器的客户端代码：

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
multicall = xmlrpc.client.MultiCall(proxy)
multicall.add(7, 3)
multicall.subtract(7, 3)
multicall.multiply(7, 3)
multicall.divide(7, 3)
result = multicall()

print("7+3=%d, 7-3=%d, 7*3=%d, 7//3=%d" % tuple(result))
```

21.26.7。便利功能

xmlrpc.client. dumps (*params* , *methodname* = *None* , *methodresponse* = *None* , *encoding* = *None* , *allow_none* = *False*)

将*params*转换为XML-RPC请求。或者如果*方法*响应 为真，则将*其*转换为响应。*参数*可以是参数的元组或Fault异常类的实例 。如果*methodresponse*为true，则只能返回一个值，这意味着*params*的长度必须为1. *编码* (如果提供) 是在生成的XML中使用的编码; 默认是UTF-8 。 Python 的 None 值不能用于标准的XML-RPC; 允许通过扩展来使用它 ，为*allow_none*提供一个真正的值。

xmlrpc.client. loads (*data* , *use_datetime* = *False* , *use_builtin_types* = *False*)

将XML-RPC请求或响应转换为Python对象，a 。 *params*是一个论元; *methodname*是一个字符串，或者 如果数据包中没有方法名称。如果XML-RPC数据包代表故障条件，则此函数将引发异常。所述*use_builtin_types*标志可用于引起日期/时间值被表示为对象和要呈现为二进制数据对象；此标志默认为false 。 (params, methodname) None Fault datetime.datetime bytes

过时的*use_datetime*标志与*use_builtin_types*类似，但它仅适用于日期/时间值。

*在3.3版本中更改：*将*use_builtin_types*加入标志。

21.26.8。客户端使用示例

```
# simple test program (from the XML-RPC specification)
from xmlrpc.client import ServerProxy, Error

# server = ServerProxy("http://localhost:8000") # local server
with ServerProxy("http://betty.userland.com") as proxy:

    print(proxy)

    try:
        print(proxy.examples.getStateName(41))
    except Error as v:
        print("ERROR", v)
```

要通过HTTP代理访问XML-RPC服务器，您需要定义一个自定义传输。以下示例显示了如何：

```
import http.client
import xmlrpc.client

class ProxiedTransport(xmlrpc.client.Transport):

    def set_proxy(self, host, port=None, headers=None):
        self.proxy = host, port
        self.proxy_headers = headers

    def make_connection(self, host):
        connection = http.client.HTTPConnection(*self.proxy)
        connection.set_tunnel(host, headers=self.proxy_headers)
        self._connection = host, connection
        return connection

transport = ProxiedTransport()
transport.set_proxy('proxy-server', 8080)
server = xmlrpc.client.ServerProxy('http://betty.userland.com', transport=transport)
print(server.examples.getStateName(41))
```

21.26.9。客户端和服务器使用示例

请参阅SimpleXMLRPCServer示例。

脚注

[1] 这种方法首先在关于xmlrpc.com的讨论中提出。

21.27。xmlrpc.server- 基本的XML-RPC服务器

源代码： Lib / xmlrpc / server.py

该xmlrpc.server模块为使用Python编写的XML-RPC服务器提供了一个基本的服务器框架。服务器可以独立使用 SimpleXMLRPCServer ，也可以嵌入CGI环境中使用CGIXMLRPCRequestHandler。

警告: 该xmlrpc.server模块对恶意构建的数据不安全。如果您需要解析不可信或未经身份验证的数据，请参阅 XML漏洞。

class xmlrpc.server.SimpleXMLRPCServer（*addr*，*requestHandler* = *SimpleXMLRPCRequestHandler*，*logRequests* = *True*，*allow_none* = *False*，*encoding* = *None*，*bind_and_activate* = *True*，*use_builtin_types* = *False* ）

创建一个新的服务器实例。该类提供了可以由XML-RPC协议调用的函数注册方法。所述*requestHandler* 参数应至少为请求处理程序实例的工厂；它默认为 SimpleXMLRPCRequestHandler 。该*地址*和*requestHandler* 参数传递给socketserver.TCPServer构造函数。如果*logRequests* 为true（缺省值），则会记录请求;将此参数设置为false将关闭日志记录。该*ALLOW_NONE*和*编码*参数传递到xmlrpc.client并控制从服务器返回的XML-RPC响应。该*bind_and_activate*参数控制是否 server_bind()和server_activate()由构造函数立即调用;它默认为true。将其设置为false允许代码在绑定地址之前操作*allow_reuse_address*类变量。所述*use_builtin_types*参数被传递给 loads()时被接收的日期/时间值或二进制数据哪种类型的处理功能和控制;它默认为false。

*在3.3版本中更改：*将*use_builtin_types*加入标志。

class xmlrpc.server.CGIXMLRPCRequestHandler（*allow_none* = *False*，*encoding* = *None*，*use_builtin_types* = *False* ）

创建一个新的实例来处理CGI环境中的XML-RPC请求。该 *ALLOW_NONE*和*编码*参数传递到xmlrpc.client 并控制从服务器返回的XML-RPC响应。所述*use_builtin_types*参数被传递给 loads()时被接收的日期/时间值或二进制数据哪种类型的处理功能和控制;它默认为false。

*在3.3版本中更改：*将*use_builtin_types*加入标志。

*类*xmlrpc.server.SimpleXMLRPCRequestHandler

创建一个新的请求处理器实例。此请求处理程序支持POST 请求并修改日志记录，以便*遵守*SimpleXMLRPCServer构造函数参数的*logRequests*参数 。

21.27.1。SimpleXMLRPCServer对象

在SimpleXMLRPCServer基于类 socketserver.TCPServer ，并提供创建简单的手段，独立的XML-RPC服务器。

SimpleXMLRPCServer.**register_function**(*函数*, *名称=无*)

注册一个可以响应XML-RPC请求的函数。如果给出*名称*，它将是与*函数*关联的方法名称，否则 function.__name__将被使用。 *名称*可以是普通字符串或Unicode字符串，也可以包含Python标识符中不合法的字符，包括句点字符。

SimpleXMLRPCServer.**register_instance**(*instance*, *allow_dotted_names = False*)

注册一个用来暴露尚未使用的方法名称的对象register_function()。如果*实例*包含_dispatch()方法，则使用请求的方法名称和请求中的参数调用它。它的API是 (注意*params*不代表可变参数列表)。如果它调用一个底层函数来执行它的任务，那么这个函数被称为 扩展参数列表。作为结果，返回值 将返回给客户端。如果*实例*没有方法，则会搜索与请求的方法的名称匹配的属性。 def _dispatch(self, method, params) func(*params) _dispatch() _dispatch()

如果可选的*allow_dotted_names*参数为true并且该实例没有_dispatch()方法，那么如果请求的方法名称包含句点，则会单独搜索方法名称的每个组件，结果会执行简单的分层搜索。然后使用请求中的参数调用此搜索中找到的值，并将返回值传回客户端。

> **警告:** 启用*allow_dotted_names*选项允许入侵者访问您的模块的全局变量，并可能允许入侵者在您的机器上执行任意代码。只能在安全的封闭网络上使用此选项。

SimpleXMLRPCServer.**register_introspection_functions**( )

注册XML-RPC内省功能 system.listMethods , system.methodHelp 和 system.methodSignature。

SimpleXMLRPCServer.**register_multicall_functions**( )

注册XML-RPC多重函数system.multicall。

SimpleXMLRPCRequestHandler.**rpc_paths**

一个属性值，必须是列出用于接收XML-RPC请求的URL的有效路径部分的元组。发布到其他路径的请求将导致404“无此页”HTTP错误。如果此元组为空，则所有路径都将被视为有效。默认值是。('/', '/RPC2')

21.27.1.1。SimpleXMLRPCServer示例

服务器代码：

```
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler

# Restrict to a particular path.
class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

# Create server
with SimpleXMLRPCServer(("localhost", 8000),
```

```
                        requestHandler=RequestHandler) as server:
    server.register_introspection_functions()

    # Register pow() function; this will use the value of
    # pow.__name__ as the name, which is just 'pow'.
    server.register_function(pow)

    # Register a function under a different name
    def adder_function(x,y):
        return x + y
    server.register_function(adder_function, 'add')

    # Register an instance; all the methods of the instance are
    # published as XML-RPC methods (in this case, just 'mul').
    class MyFuncs:
        def mul(self, x, y):
            return x * y

    server.register_instance(MyFuncs())

    # Run the server's main loop
    server.serve_forever()
```

以下客户端代码将调用上述服务器提供的方法：

```
import xmlrpc.client

s = xmlrpc.client.ServerProxy('http://localhost:8000')
print(s.pow(2,3))  # Returns 2**3 = 8
print(s.add(2,3))  # Returns 5
print(s.mul(5,2))  # Returns 5*2 = 10

# Print list of available methods
print(s.system.listMethods())
```

Lib/xmlrpc/server.py模块中包含的以下示例显示了允许虚线名称和注册多路呼叫功能的服务器。

警告： 启用*allow_dotted_names*选项允许入侵者访问您的模块的全局变量，并可能允许入侵者在您的机器上执行任意代码。只能在安全的封闭网络中使用此示例。

```
import datetime

class ExampleService:
    def getData(self):
        return '42'

    class currentTime:
        @staticmethod
        def getCurrentTime():
            return datetime.datetime.now()

with SimpleXMLRPCServer(("localhost", 8000)) as server:
    server.register_function(pow)
    server.register_function(lambda x,y: x+y, 'add')
```

```
    server.register_instance(ExampleService(), allow_dotted_names=True)
    server.register_multicall_functions()
    print('Serving XML-RPC on localhost port 8000')
    try:
        server.serve_forever()
    except KeyboardInterrupt:
        print("\nKeyboard interrupt received, exiting.")
        sys.exit(0)
```

这个ExampleService演示可以从命令行调用：

```
python -m xmlrpc.server
```

与上述服务器交互的客户端包含在 *Lib / xmlrpc / client.py中*：

```
server = ServerProxy("http://localhost:8000")

try:
    print(server.currentTime.getCurrentTime())
except Error as v:
    print("ERROR", v)

multi = MultiCall(server)
multi.getData()
multi.pow(2,9)
multi.add(1,2)
try:
    for response in multi():
        print(response)
except Error as v:
    print("ERROR", v)
```

这个与演示XMLRPC服务器交互的客户端可以被调用为：

```
python -m xmlrpc.client
```

21.27.2。CGIXMLRPCRequestHandler

该CGIXMLRPCRequestHandler班可用于处理发送到Python的CGI脚本XML-RPC请求。

CGIXMLRPCRequestHandler.**register_function**(*函数*, *名称=无*)

注册一个可以响应XML-RPC请求的函数。如果 给出*名称*，它将是与函数关联的方法名称，否则 将使用*函数.__ name__*。*名称*可以是普通字符串或Unicode字符串，也可以包含Python标识符中不合法的字符，包括句点字符。

CGIXMLRPCRequestHandler.**register_instance**(*实例*)

注册一个用来暴露尚未使用的方法名称的对象register_function()。如果实例包含_dispatch()方法，则使用请求的方法名称和请求中的参数调用该方法; 返回值作为结果返回给客户端。如果实例没有_dispatch()方法，则会搜索与请求方法的名称匹配的属性; 如果请求的方法名称包含句点，则会单独搜索方法名称的每个组件，从而执行简单的分层搜索。然后使用请求中的参数调用此搜索中找到的值，并将返回值传回客户端。

CGIXMLRPCRequestHandler.register_introspection_functions()
注册XML-RPC内省功能system.listMethods，system.methodHelp和system.methodSignature。

CGIXMLRPCRequestHandler.register_multicall_functions()
注册XML-RPC multicall函数system.multicall。

CGIXMLRPCRequestHandler.handle_request(*request_text* = *None*)
处理XML-RPC请求。如果给出*request_text*，它应该是由HTTP服务器提供的POST数据，否则将使用stdin的内容。

例：

```
class MyFuncs:
    def mul(self, x, y):
        return x * y


handler = CGIXMLRPCRequestHandler()
handler.register_function(pow)
handler.register_function(lambda x,y: x+y, 'add')
handler.register_introspection_functions()
handler.register_instance(MyFuncs())
handler.handle_request()
```

21.27.3。记录XMLRPC服务器

这些类扩展上述类以响应HTTP GET请求提供HTML文档。服务器可以独立使用DocXMLRPCServer，也可以嵌入CGI环境中使用DocCGIXMLRPCRequestHandler。

class xmlrpc.server.DocXMLRPCServer(*addr*, *requestHandler* = *DocXMLRPCRequestHandler*, *logRequests* = *True*, *allow_none* = *False*, *encoding* = *None*, *bind_and_activate* = *True*, *use_builtin_types* = *True*)
创建一个新的服务器实例。所有参数的含义与以前相同SimpleXMLRPCServer; *requestHandler*默认为DocXMLRPCRequestHandler。

*在3.3版本中更改：*将*use_builtin_types*加入标志。

*类*xmlrpc.server.DocCGIXMLRPCRequestHandler
创建一个新的实例来处理CGI环境中的XML-RPC请求。

*类*xmlrpc.server.DocXMLRPCRequestHandler
创建一个新的请求处理器实例。此请求处理程序支持XML-RPC POST请求，文档GET请求以及修改日志记录，以便*遵守*DocXMLRPCServer构造函数参数的*logRequests*参数。

21.27.4。DocXMLRPCServer对象

本DocXMLRPCServer类源自SimpleXMLRPCServer 并提供创建自我记录的手段，独立的XML-RPC服务器。HTTP POST请求被处理为XML-RPC方法调用。通过生成pydoc样式的HTML文档来处理HTTP GET请求。这允许服务器提供自己的基于Web的文档。

DocXMLRPCServer.set_server_title(*server_title*)
设置生成的HTML文档中使用的标题。这个标题将在HTML“title”元素中使用。

DocXMLRPCServer.set_server_name(*server_name*)
设置在生成的HTML文档中使用的名称。该名称将显示在“h1”元素内生成的文档的顶部。

DocXMLRPCServer.set_server_documentation(*server_documentation*)
设置在生成的HTML文档中使用的描述。此说明在文档中以服务器名称下方的段落显示。

21.27.5。DocCGIXMLRPCRequestHandler

的DocCGIXMLRPCRequestHandler类源自 CGIXMLRPCRequestHandler与提供了创建自文档，XML-RPC CGI脚本的装置。HTTP POST请求被处理为XML-RPC方法调用。通过生成pydoc样式的HTML文档来处理HTTP GET请求。这允许服务器提供自己的基于Web的文档。

DocCGIXMLRPCRequestHandler.set_server_title(*server_title*)
设置生成的HTML文档中使用的标题。这个标题将在HTML“title”元素中使用。

DocCGIXMLRPCRequestHandler.set_server_name(*server_name*)
设置在生成的HTML文档中使用的名称。该名称将显示在“h1”元素内生成的文档的顶部。

DocCGIXMLRPCRequestHandler.set_server_documentation(*server_documentation*)
设置在生成的HTML文档中使用的描述。此说明在文档中以服务器名称下方的段落显示。

21.28。 ipaddress- IPv4 / IPv6操作库

源代码： Lib / ipaddress.py

ipaddress 提供了创建，操作和操作IPv4和IPv6地址和网络的功能。

这个模块中的函数和类可以很简单地处理与IP地址相关的各种任务，包括检查两个主机是否在同一个子网上，遍历特定子网中的所有主机，检查字符串是否代表有效IP地址或网络定义等。

这是完整的模块API参考 - 有关概述和介绍，请参阅 ipaddress模块的介绍。

3.3版本的新功能

21.28.1。便利工厂功能

该ipaddress模块提供工厂功能来方便地创建IP地址，网络和接口：

ipaddress. **ip_address** (*地址*)

根据作为参数传递的IP地址返回一个IPv4Address或一个IPv6Address对象。可以提供IPv4或IPv6地址; 整数小于2 ** 32默认会被认为是IPv4。ValueError如果*地址*不代表有效的IPv4或IPv6地址，则引发A.

```
>>> ipaddress.ip_address('192.168.0.1')
IPv4Address('192.168.0.1')
>>> ipaddress.ip_address('2001:db8::')
IPv6Address('2001:db8::')
```

ipaddress. **ip_network** (*address* , *strict = True*)

根据作为参数传递的IP地址返回一个IPv4Network或一个IPv6Network对象。 *地址*是表示IP网络的字符串或整数。可以提供IPv4或IPv6网络; 整数小于2 ** 32默认会被认为是IPv4。 *严格* 传递给IPv4Network或IPv6Network构造函数。ValueError如果*地址*不代表有效的IPv4或IPv6地址，或者网络设置了主机位，则引发A.

```
>>> ipaddress.ip_network('192.168.0.0/28')
IPv4Network('192.168.0.0/28')
```

ipaddress. **ip_interface** (*地址*)

根据作为参数传递的IP地址返回一个IPv4Interface或一个IPv6Interface对象。 *地址*是表示IP地址的字符串或整数。可以提供IPv4或IPv6地址; 整数小于2 ** 32默认会被认为是IPv4。ValueError如果*地址*不代表有效的IPv4或IPv6地址，则引发A.

这些便利功能的一个缺点是需要同时处理IPv4和IPv6格式，这意味着错误消息提供了关于精确错误的最少信息，因为功能并不知道IPv4或IPv6格式是否有意使用。通过直接调用适当的特定于版本的类构造函数可以获得更详细的错误报告。

21.28.2。IP地址

21.28.2.1。地址对象

在IPv4Address和IPv6Address对象有许多共同的属性。一些仅对IPv6地址有意义的属性也由IPv4Address对象实现，以便更容易编写能够正确处理两个IP版本的代码。地址对象是可 散列的，因此它们可以在字典中用作键。

*类*ipaddress.IPv4Address（*地址*）

构建一个IPv4地址。一AddressValueError，如果被提升 *的地址*是不是有效的IPv4地址。

以下内容构成一个有效的IPv4地址：

1. 以小数点符号表示的字符串，由包含范围0-255中的四个十进制整数组成，由点（例如192.168.0.1）分隔。每个整数表示地址中的八位字节（字节）。只有小于8的值才能容忍0（因为在这种字符串的十进制和八进制解释之间没有歧义）。
2. 一个适合32位的整数。
3. 一个整数，打包到一个bytes长度为4 的对象中（最重要的八位字节在前）。

```
>>> ipaddress.IPv4Address('192.168.0.1')
IPv4Address('192.168.0.1')
>>> ipaddress.IPv4Address(3232235521)
IPv4Address('192.168.0.1')
>>> ipaddress.IPv4Address(b'\xC0\xA8\x00\x01')
IPv4Address('192.168.0.1')
```

version

适当的版本号：4适用于IPv4，6适用于IPv6。

max_prefixlen

此版本的地址表示中的总位数：32用于IPv4，128用于IPv6。

该前缀定义了一个地址中的前导比特的数量，这些比特用于确定地址是否是网络的一部分。

compressed

exploded

以点分十进制表示法的字符串表示形式。前导零不会包含在表示中。

由于IPv4没有为八位字节设置为零的地址定义简写符号，因此这两个属性始终与str(addr) IPv4地址相同。公开这些属性可以更轻松地编写可处理IPv4和IPv6地址的显示代码。

packed

这个地址的二进制表示 - bytes适当长度的对象（最重要的八位字节在前）。这是4个字节的IPv4和16个字节的IPv6。

reverse_pointer

IP地址的反向DNS PTR记录的名称，例如：

```
>>> ipaddress.ip_address("127.0.0.1").reverse_pointer
'1.0.0.127.in-addr.arpa'
>>> ipaddress.ip_address("2001:db8::1").reverse_pointer
'1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.8.b.d.0.1.0.0.2.ip6.arpa'
```

这是可用于执行PTR查找的名称，而不是解析的主机名本身。

3.5版本中的新功能。

is_multicast

True如果地址被保留用于多播使用。看到 **RFC 3171**（用于IPv4）或**RFC 2373**（用于IPv6）。

is_private

True如果地址分配给专用网络。请参阅 iana-ipv4-special-registry（用于IPv4）或iana-ipv6-special-registry （用于IPv6）。

is_global

True如果地址分配给公共网络。请参阅 iana-ipv4-special-registry（用于IPv4）或iana-ipv6-special-registry （用于IPv6）。

3.4版新增功能

is_unspecified

True如果地址未指定。看到**RFC 5735**（用于IPv4）或**RFC 2373**（用于IPv6）。

is_reserved

True 如果地址是IETF保留的。

is_loopback

True 如果这是一个回送地址。看到**RFC 3330**（用于IPv4）或**RFC 2373**（用于IPv6）。

is_link_local

True如果该地址被保留用于链路本地使用。看到 **RFC 3927**。

*类*ipaddress.IPv6Address（*地址*）

构建一个IPv6地址。一AddressValueError，如果被提升 *的地址*是不是一个有效的IPv6地址。

以下内容构成一个有效的IPv6地址：

1. 由八组四个十六进制数字组成的字符串，每组代表16位。这些组由冒号分隔。这描述了一个*分解*（长手形）符号。字符串也可以通过各种方式*压缩*（简写符号）。看到**RFC 4291**了解详情。例如，"0000:0000:0000:0000:0000:0abc:0007:0def"可以压缩到"::abc:7:def"。
2. 一个适合128位的整数。
3. 一个整数，打包成长bytes度为16的大对象的对象。

```
>>> ipaddress.IPv6Address('2001:db8::1000')
IPv6Address('2001:db8::1000')
```

compressed

地址表示的缩写形式，省略了组中的前导零以及最长的完全由零组成的组的序列折叠为单个空组。

这也是str(addr)IPv6地址返回的值。

exploded

地址表示的长形式，包括所有前导零和全部由零组成的组。

有关以下属性，请参阅该IPv4Address课程的相应文档 ：

packed

reverse_pointer

version

max_prefixlen

is_multicast

is_private

is_global

is_unspecified

is_reserved

is_loopback

is_link_local

新版本3.4： is_global

is_site_local

True如果地址是为站点本地使用而保留的。请注意，站点本地地址空间已被弃用**RFC 3879**。使用 is_private 测试如果这个地址是唯一的本地地址的空间所界定**RFC 4193**。

ipv4_mapped

对于看起来是IPv4映射地址地址（从开始 ::FFFF/96 ），此属性将报告嵌入的IPv4地址。对于任何其他地址，该属性将是None。

sixtofour

对于看起来像6to4地址（开始2002::/16 ）的地址 **RFC 3056** ，此属性将报告嵌入的IPv4地址。对于任何其他地址，该属性将是None。

teredo

对于看起来像Teredo地址（开始于2001::/32）的地址，定义如下**RFC 4380**，此属性将报告嵌入式IP地址对。对于任何其他地址，该属性将是。(server, client)None

21.28.2.2。转换为字符串和整数

要与网络接口（如套接字模块）进行互操作，地址必须转换为字符串或整数。这是使用str()和int()内建函数处理的：

```
>>> str(ipaddress.IPv4Address('192.168.0.1'))
'192.168.0.1'
>>> int(ipaddress.IPv4Address('192.168.0.1'))
3232235521
>>> str(ipaddress.IPv6Address('::1'))
'::1'
>>> int(ipaddress.IPv6Address('::1'))
1
```

21.28.2.3。运营商

地址对象支持一些运营商。除非另有说明，否则运营商只能在兼容对象（即IPv4与IPv4，IPv6与IPv6）之间应用。

21.28.2.3.1。比较运算符

地址对象可以与通常的一组比较运算符进行比较。一些例子：

```
>>> IPv4Address('127.0.0.2') > IPv4Address('127.0.0.1')
True
>>> IPv4Address('127.0.0.2') == IPv4Address('127.0.0.1')
False
>>> IPv4Address('127.0.0.2') != IPv4Address('127.0.0.1')
True
```

21.28.2.3.2。算术运算符

整数可以添加到地址对象或从地址对象中减去。一些例子：

```
>>> IPv4Address('127.0.0.2') + 3
IPv4Address('127.0.0.5')
>>> IPv4Address('127.0.0.2') - 3
IPv4Address('126.255.255.255')
>>> IPv4Address('255.255.255.255') + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ipaddress.AddressValueError: 4294967296 (>= 2**32) is not permitted as an IPv4 address
```

21.28.3。IP网络定义

在IPv4Network和IPv6Network对象提供用于定义和检查IP网络定义的机制。网络定义由一个*掩码*和一个*网络地址组成*，因此定义了一个IP地址范围，这些IP地址在与掩码进行掩码（二进制AND）时等于网络地址。例如，与掩模的网络定义255.255.255.0和所述网络地址192.168.1.0由IP地址的在包容范围192.168.1.0到192.168.1.255。

21.28.3.1。前缀，网络掩码和主机掩码

有几种等同的方法来指定IP网络掩码。甲*前缀* /<nbits>是表示许多高阶位是如何在网络掩码中设置的符号。甲*净掩模*是与一些数量的设置高阶比特的IP地址。因此，前缀/24相当于255.255.255.0IPv4中的网络掩码或ffff:ff00::IPv6中的网络掩码。另外，*主机掩码*是*网络掩码*的逻辑反转，并且有时用于（例如在Cisco访问控制列表中）表示网络掩码。主机掩码相当于/24IPv4 0.0.0.255。

21.28.3.2。网络对象

由地址对象实现的所有属性也由网络对象实现。另外，网络对象实现附加属性。所有这些都是IPv4Network和之间的共同点IPv6Network，所以为了避免重复，他们只是记录在案IPv4Network。网络对象是可散列的，因此它们可以在字典中用作键。

class ipaddress. IPv4Network（*address*，*strict* = *True*）

构建IPv4网络定义。*地址*可以是以下之一：

1. 由斜杠（/）分隔的由IP地址和可选掩码组成的字符串。IP地址是网络地址，掩码可以是单个数字，也就是说它是*前缀*或IPv4地址的字符串表示形式。如果是后者，掩模被解释为*网络掩码*如果它与非零字段开始，或作为*主机掩模*如果它与一个零场开始时，一个全零掩码的一个例外，其被视为一个*网罩*。如果没有提供掩码，则认为是/32。

   例如，以下*地址*规范是等效的：192.168.1.0/24，192.168.1.0/255.255.255.0和192.168.1.0/0.0.0.255。

2. 一个适合32位的整数。这相当于一个单一地址的网络，网络地址是*地址*和掩码/32。

3. 一个整数，打包成长bytes度为4 的对象，大端。解释类似于整数*地址*。

4. 地址描述和网络掩码的二元组，其中地址描述是一个字符串，一个32位整数，一个4字节的打包整数或一个现有的IPv4Address对象; 并且网络掩码是表示前缀长度的整数（例如24）或表示前缀掩码的字符串（例如255.255.255.0）。

一AddressValueError，如果被提升*的地址*是不是有效的IPv4地址。NetmaskValueError如果掩码对IPv4地址无效，则引发A.

如果*严格*是True与主机位在所提供的地址来设置，然后ValueError上升。否则，主机位被屏蔽掉以确定合适的网络地址。

除非另有说明，否则TypeError如果参数的IP版本不兼容，所有接受其他网络/地址对象的网络方法都会引发self。

*在版本3.5中进行了更改：*为*地址*构造函数参数添加了二元组形式。

version

max_prefixlen
    请参阅相应的属性文档 IPv4Address。

is_multicast

is_private

is_unspecified

is_reserved

is_loopback

is_link_local
    如果网络地址和广播地址都属实，则这些属性对于整个网络都是正确的。

network_address
    网络的网络地址。网络地址和前缀长度一起唯一地定义一个网络。

broadcast_address
    网络的广播地址。发送到广播地址的数据包应由网络上的每台主机接收。

hostmask
    主机掩码作为IPv4Address对象。

netmask
    网罩，作为一个IPv4Address对象。

with_prefixlen

compressed

exploded
    网络的字符串表示形式，掩码为前缀表示法。

    with_prefixlen并且compressed总是一样的 str(network)。 exploded采用分解形式的网络地址。

with_netmask
    网络的字符串表示形式，掩码表示为网络掩码表示法。

with_hostmask
    网络的字符串表示形式，掩码用主机掩码表示法表示。

num_addresses
    网络中的地址总数。

prefixlen

网络前缀的长度，以位为单位。

hosts ( )

返回网络中可用主机的迭代器。可用主机是属于网络的所有IP地址，除了网络地址本身和网络广播地址。对于掩码长度为31的网络，网络地址和网络广播地址也包含在结果中。

```
>>> list(ip_network('192.0.2.0/29').hosts())
[IPv4Address('192.0.2.1'), IPv4Address('192.0.2.2'),
 IPv4Address('192.0.2.3'), IPv4Address('192.0.2.4'),
 IPv4Address('192.0.2.5'), IPv4Address('192.0.2.6')]
>>> list(ip_network('192.0.2.0/31').hosts())
[IPv4Address('192.0.2.0'), IPv4Address('192.0.2.1')]
```

overlaps (*其他*)

True如果该网络被部分地或全部地包含在*其他*或 *其他*完全包含该网络。

address_exclude (*网络*)

计算从该网络中移除给定*网络*所产生的网络定义 。返回网络对象的迭代器。ValueError如果*网络*没有完全包含在这个网络中，则引发。

```
>>> n1 = ip_network('192.0.2.0/28')
>>> n2 = ip_network('192.0.2.1/32')
>>> list(n1.address_exclude(n2))
[IPv4Network('192.0.2.8/29'), IPv4Network('192.0.2.4/30'),
 IPv4Network('192.0.2.2/31'), IPv4Network('192.0.2.0/32')]
```

subnets (*prefixlen_diff = 1 , new_prefix = None*)

根据参数值，加入的子网将进行当前的网络定义。 *prefixlen_diff*是我们的前缀长度应该增加的量。 *new_prefix*是子网所需的新前缀; 它必须大于我们的前缀。必须设置*prefixlen_diff*和*new_prefix*中的一个和唯一一个。返回网络对象的迭代器。

```
>>> list(ip_network('192.0.2.0/24').subnets())
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/25')]
>>> list(ip_network('192.0.2.0/24').subnets(prefixlen_diff=2))
[IPv4Network('192.0.2.0/26'), IPv4Network('192.0.2.64/26'),
 IPv4Network('192.0.2.128/26'), IPv4Network('192.0.2.192/26')]
>>> list(ip_network('192.0.2.0/24').subnets(new_prefix=26))
[IPv4Network('192.0.2.0/26'), IPv4Network('192.0.2.64/26'),
 IPv4Network('192.0.2.128/26'), IPv4Network('192.0.2.192/26')]
>>> list(ip_network('192.0.2.0/24').subnets(new_prefix=23))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    raise ValueError('new prefix must be longer')
ValueError: new prefix must be longer
>>> list(ip_network('192.0.2.0/24').subnets(new_prefix=25))
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/25')]
```

supernet (*prefixlen_diff = 1 , new_prefix = None*)

包含此网络定义的超网，取决于参数值。 *prefixlen_diff*是我们的前缀长度应该减少的量。 *new_prefix*是超网所需的新前缀; 它必须小于我们的前缀。必须设置*prefixlen_diff*和*new_prefix中的*一个和唯一一个。返回单个网络对象。

```
>>> ip_network('192.0.2.0/24').supernet()
IPv4Network('192.0.2.0/23')
>>> ip_network('192.0.2.0/24').supernet(prefixlen_diff=2)
IPv4Network('192.0.0.0/22')
>>> ip_network('192.0.2.0/24').supernet(new_prefix=20)
IPv4Network('192.0.0.0/20')
```

compare_networks（*其他*）

比较这个网络与*其他*。在这个比较中只考虑网络地址; 主机位不是。返回要么-1， 0或1。

```
>>> ip_network('192.0.2.1/32').compare_networks(ip_network('192.0.2.2/32'))
-1
>>> ip_network('192.0.2.1/32').compare_networks(ip_network('192.0.2.0/32'))
1
>>> ip_network('192.0.2.1/32').compare_networks(ip_network('192.0.2.1/32'))
0
```

class ipaddress. IPv6Network（*address*，*strict* = *True* ）

构建IPv6网络定义。 *地址*可以是以下之一：

1. 一个由IP地址和可选前缀长度组成的字符串，用斜杠（ / ）分隔。IP地址是网络地址，前缀长度必须是单个数字，即*前缀*。如果没有提供前缀长度，则认为是/128。

   请注意，当前扩展的网络掩码不受支持。这意味着 2001:db00::0/24一个有效的论证，而2001:db00::0/ffff:ff00:: 不是。

2. 一个适合128位的整数。这相当于一个单一地址的网络，网络地址是*地址*和掩码/128。

3. 一个整数，打包成长bytes度为16的大对象的对象。解释类似于整数*地址*。

4. 地址描述和网络掩码的二元组，其中地址描述是一个字符串，一个128位整数，一个16字节的打包整数或一个现有的IPv6Address对象; 并且网络掩码是表示前缀长度的整数。

一AddressValueError，如果被提升*的地址*是不是一个有效的IPv6地址。 NetmaskValueError如果掩码对IPv6地址无效，则引发A.

如果*严格*是True与主机位在所提供的地址来设置，然后ValueError上升。否则，主机位被屏蔽掉以确定合适的网络地址。

*在版本3.5中进行了更改：*为*地址*构造函数参数添加了二元组形式。

version

max_prefixlen

is_multicast

is_private

is_unspecified

is_reserved

is_loopback

is_link_local

network_address

broadcast_address

hostmask

netmask

with_prefixlen

compressed

exploded

with_netmask

with_hostmask

num_addresses

prefixlen

hosts ( )

返回网络中可用主机的迭代器。可用主机是属于网络的所有IP地址，除了子网路由器任播地址。对于掩码长度为127的网络，子网路由器任播地址也包含在结果中。

overlaps (*其他*)

address_exclude (*网络*)

subnets (*prefixlen_diff = 1* , *new_prefix = None*)

supernet (*prefixlen_diff = 1* , *new_prefix = None*)

compare_networks (*其他*)

请参阅相应的属性文档 IPv4Network。

is_site_local

如果网络地址和广播地址都为真，则整个网络的这些属性都是正确的。

21.28.3.3。运营商

网络对象支持一些运营商。除非另有说明，否则运营商只能在兼容对象（即IPv4与IPv4，IPv6与IPv6）之间应用。

21.28.3.3.1。逻辑运算符

网络对象可以与通常的一组逻辑运算符进行比较。网络对象首先按网络地址排序，然后按网络掩码排序。

21.28.3.3.2。迭代

可以迭代网络对象以列出属于网络的所有地址。对于迭代，返回*所有*主机，包括不可用主机（对于可用主机，请使用该`hosts()`方法）。一个例子：

```
>>> for addr in IPv4Network('192.0.2.0/28'):
...     addr
...
IPv4Address('192.0.2.0')
IPv4Address('192.0.2.1')
IPv4Address('192.0.2.2')
IPv4Address('192.0.2.3')
IPv4Address('192.0.2.4')
IPv4Address('192.0.2.5')
IPv4Address('192.0.2.6')
IPv4Address('192.0.2.7')
IPv4Address('192.0.2.8')
IPv4Address('192.0.2.9')
IPv4Address('192.0.2.10')
IPv4Address('192.0.2.11')
IPv4Address('192.0.2.12')
IPv4Address('192.0.2.13')
IPv4Address('192.0.2.14')
IPv4Address('192.0.2.15')
```

21.28.3.3.3。网络作为地址容器

网络对象可以充当地址容器。一些例子：

```
>>> IPv4Network('192.0.2.0/28')[0]
IPv4Address('192.0.2.0')
>>> IPv4Network('192.0.2.0/28')[15]
IPv4Address('192.0.2.15')
>>> IPv4Address('192.0.2.6') in IPv4Network('192.0.2.0/28')
True
>>> IPv4Address('192.0.3.6') in IPv4Network('192.0.2.0/28')
False
```

21.28.4。接口对象

接口对象是可散列的，所以它们可以用作字典中的键。

*类*ipaddress.IPv4Interface(*地址*)

构建一个IPv4接口。*地址*的含义与构造函数中的一样IPv4Network，除了任意主机地址总是被接受。

IPv4Interface是它的一个子类IPv4Address，所以它继承了该类的所有属性。另外，以下属性可用：

ip

地址（IPv4Address）没有网络信息。

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.ip
IPv4Address('192.0.2.5')
```

network

该IPv4Network接口所属的网络（）。

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.network
IPv4Network('192.0.2.0/24')
```

with_prefixlen

带前缀表示法的掩码的接口的字符串表示形式。

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_prefixlen
'192.0.2.5/24'
```

with_netmask

与网络接口的字符串表示形式作为网络掩码。

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_netmask
'192.0.2.5/255.255.255.0'
```

with_hostmask

与网络接口的字符串表示形式作为主机掩码。

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_hostmask
'192.0.2.5/0.0.0.255'
```

*类*ipaddress.IPv6Interface(*地址*)

构建一个IPv6接口。*地址*的含义与构造函数中的一样IPv6Network，除了任意主机地址总是被接受。

IPv6Interface是它的一个子类IPv6Address，所以它继承了该类的所有属性。另外，以下属性可用：

ip

network

with_prefixlen

with_netmask

with_hostmask

请参阅相应的属性文档 IPv4Interface。

21.28.4.1。运营商

接口对象支持一些操作符。除非另有说明，否则运营商只能在兼容对象（即IPv4与IPv4，IPv6与IPv6）之间应用。

21.28.4.1.1。逻辑运算符

接口对象可以与通常的一组逻辑运算符进行比较。

为了进行相等比较（==和!=），IP地址和网络必须相同才能使对象相同。接口不会与任何地址或网络对象进行比较。

对于排序（<，>等），规则是不同的。可以比较具有相同IP版本的接口和地址对象，并且地址对象将始终在接口对象之前进行排序。两个接口对象首先通过它们的网络进行比较，如果它们相同，则通过它们的IP地址进行比较。

21.28.5。其他模块级别函数

该模块还提供以下模块级功能：

ipaddress.**v4_int_to_packed**(*地址*)

在网络（大端）顺序中将地址表示为4个打包字节。 *地址*是IPv4 IP地址的整数表示。ValueError如果整数是负数或太大而不能成为IPv4 IP地址，则引发A.

```
>>> ipaddress.ip_address(3221225985)
IPv4Address('192.0.2.1')
>>> ipaddress.v4_int_to_packed(3221225985)
b'\xc0\x00\x02\x01'
```

ipaddress.**v6_int_to_packed**(*地址*)

以网络（big-endian）顺序将地址表示为16个打包字节。 *地址*是IPv6 IP地址的整数表示。ValueError如果整数为负数或太大而不能成为IPv6 IP地址，则会引发A.

ipaddress.**summarize_address_range**(*第一*, *最后*)

给定第一个和最后一个IP地址，返回总结网络范围的迭代器。 *首先*是第一个IPv4Address或IPv6Address在范围内，*最后*是最后一个IPv4Address 或IPv6Address范围内。 TypeError如果*第一个*或*最后一个*不是IP地址或不是相同版本，则引发 A. ValueError如果*最后一个*不大于*第一个*或者如果 *第一个*地址版本不是4或6，则引发A.

```
>>> [ipaddr for ipaddr in ipaddress.summarize_address_range(
...     ipaddress.IPv4Address('192.0.2.0'),
...     ipaddress.IPv4Address('192.0.2.130'))]
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/31'), IPv4Network('192.0.2.
```

ipaddress.**collapse_addresses**(*地址*)

返回折叠IPv4Network或 IPv6Network对象的迭代器。 *地址*是一个迭代器 IPv4Network或IPv6Network对象。 TypeError如果*地址*包含混合版本对象，则引发A.

```
>>> [ipaddr for ipaddr in
... ipaddress.collapse_addresses([ipaddress.IPv4Network('192.0.2.0/25'),
... ipaddress.IPv4Network('192.0.2.128/25')])]
[IPv4Network('192.0.2.0/24')]
```

ipaddress.**get_mixed_type_key**(*obj*)

返回适合在网络和地址之间进行排序的密钥。地址和网络对象默认无法排序; 他们根本不同，所以表达：

```
IPv4Address('192.0.2.0') <= IPv4Network('192.0.2.0/24')
```

没有意义。然而，有些时候，您可能希望ipaddress对这些内容进行排序。如果你需要这样做，你可以使用这个函数作为*关键*参数sorted()。

*obj*是网络或地址对象。

21.28.6。自定义例外

为了支持来自类构造函数的更具体的错误报告，模块定义了以下例外：

*异常*ipaddress.**AddressValueError**(*ValueError*)

与地址有关的任何值错误。

*异常*ipaddress.**NetmaskValueError**(*ValueError*)

与网络掩码相关的任何值错误。

22.多媒体服务

本章介绍的模块实现了主要用于多媒体应用的各种算法或接口。它们由安装人员自行决定。这里有一个概述：

22.1。audioop- 操作原始音频数据

该audioop模块包含对声音片段的一些有用操作。它对由8,16,24或32位宽的有符号整数样本组成的声音片段进行操作，存储在类似字节的对象中。所有标量项都是整数，除非另有规定。

*在版本3.4中进行了更改：*添加了对24位样本的支持。所有函数现在都可以接受任何字节类的对象。字符串输入现在会导致立即错误。

该模块提供对a-LAW，u-LAW和Intel / DVI ADPCM编码的支持。

一些更复杂的操作只需要16位采样，否则采样大小（以字节为单位）始终是操作的参数。

该模块定义了以下变量和功能：

*异常*audioop.error
所有错误都会引发此异常，例如每个样本的未知字节数等等。

audioop.add（*fragment1*，*fragment2*，*width*）
返回一个片段，它是作为参数传递的两个样本的添加。*宽度*以字节为单位，无论是在样品宽度1，2，3或4。两个片段应具有相同的长度。溢出时截断样本。

audioop.adpcm2lin（*adpcmfragment*，*width*，*state*）
将Intel / DVI ADPCM编码片段解码为线性片段。有关lin2adpcm()ADPCM编码的详细信息，请参阅说明。返回样本宽度指定*宽度*的元组。(sample, newstate)

audioop.alaw2lin（*片段*，*宽度*）
将a-LAW编码中的声音片段转换为线性编码的声音片段。a-LAW编码总是使用8位样本，因此*宽度*仅指此处输出片段的样本宽度。

audioop.avg（*片段*，*宽度*）
返回片段中所有样本的平均值。

audioop.avgpp（*片段*，*宽度*）
返回片段中所有样本的平均峰 - 峰值。没有过滤完成，所以这个例程的用处是有问题的。

audioop.bias（*片段*，*宽度*，*偏向*）
返回一个片段，该片段是在每个样本中添加了偏差的原始片段。样品环绕在溢出情况下。

audioop.byteswap（*片段*，*宽度*）
“Byteswap”片段中的所有样本并返回修改的片段。将big-endian样本转换为little-endian，反之亦然。

3.4版新增功能

audioop.cross（*片段*，*宽度*）
返回作为参数传递的片段中零交叉的数量。

audioop.**findfactor**(*片段*, *参考*)

返回一个因子*F*，使其最小化，也就是说，返回您应该乘以*参考*的因子，使其尽可能匹配以便*片段化*。这些片段都应该包含2个字节的样本。 `rms(add(fragment, mul(reference, -F)))`

这个程序所用的时间与成正比`len(fragment)`。

audioop.**findfit**(*片段*, *参考*)

尝试尽可能将*参考*与*片段*的一部分（应该是更长的片段）进行匹配。这是（从概念上）通过从片段中取出*片段*来完成的，`findfactor()`用于计算最佳匹配并最小化结果。这些片段都应该包含2个字节的样本。返回一个元组，其中*偏移量*是（整数）偏移量到 最佳匹配开始的*片段*，*factor*是（浮点）因子。`(offset, factor)` `findfactor()`

audioop.**findmax**(*片段*, *长度*)

搜索具有最大能量的长度*长度*采样*片段*（不是字节！），即返回*i*，其 最大值。这些片段都应该包含2个字节的样本。 `rms(fragment[i*2:(i+length)*2])`

该程序需要的时间与成比例`len(fragment)`。

audioop.**getsample**(*片段*, *宽度*, *索引*)

从片段中返回样本*索引*的值。

audioop.**lin2adpcm**(*片段*, *宽度*, *状态*)

将采样转换为4位Intel / DVI ADPCM编码。ADPCM编码是一种自适应编码方案，其中每个4位数是一个样本与下一个样本之间的差异，除以（变化）步骤。英特尔/ DVI ADPCM算法已被IMA选用，因此它很可能成为标准。

*state*是一个包含编码器*状态*的元组。编码器返回一个元组 ，并且新*状态*应该被传递给下一个。在最初的调用中，可以作为状态传递。 *adpcmfrag*是ADPCM编码的片段，每个字节包装2个4位值。`(adpcmfrag, newstate)` `lin2adpcm()` `None`

audioop.**lin2alaw**(*片段*, *宽度*)

将音频片段中的采样转换为a-LAW编码，并将其作为字节对象返回。a-LAW是一种音频编码格式，您只需使用8位样本即可获得约13位的动态范围。它由Sun音频硬件等使用。

audioop.**lin2lin**(*fragment*, *width*, *newwidth*)

在1，2，3和4字节格式之间转换采样。

> **注意：** 在某些音频格式中，如.WAV文件，16,24和32位采样被签名，但8位采样未经签名。因此，当转换为这些格式的8位宽采样时，还需要将128添加到结果中：
>
> ```
> new_frames = audioop.lin2lin(frames, old_width, 1)
> new_frames = audioop.bias(new_frames, 1, 128)
> ```
>
> 反过来，在从8位到16位，24位或32位宽度采样转换时，也必须采用相同的方法。

audioop.**lin2ulaw**(*片段*, *宽度*)

将音频片段中的样本转换为u-LAW编码，并将其作为字节对象返回。u-LAW是一种音频编码格式，您只需使用8位采样即可获得约14位的动态范围。它由Sun音频硬件等使用。

audioop.**max**(*片段*, *宽度*)
返回片段中所有样本的*绝对值*的最大值。

audioop.**maxpp**(*片段*, *宽度*)
返回声音片段中的最大峰 - 峰值。

audioop.**minmax**(*片段*, *宽度*)
返回由声音片段中所有样本的最小值和最大值组成的元组。

audioop.**mul**(*片段*, *宽度*, *因子*)
返回原始片段中所有样本乘以浮点值*因子的*片段。溢出时截断样本。

audioop.**ratecv**(*片段*, *宽度*, *nchannels*, *inrate*, *outrate*, *state*[, *weightA*[, *weightB*]])
转换输入片段的帧频。

*state*是包含转换器*状态*的元组。转换器返回一个元组，并且*newstate*应该被传递给下一个调用。最初的呼叫应该作为状态传递。(newfragment, newstate)ratecv()None

的*weightA*和*weightB*参数是一个简单的数字滤波器和默认参数以1及0分别。

audioop.**reverse**(*片段*, *宽度*)
反转片段中的样本并返回修改后的片段。

audioop.**rms**(*片段*, *宽度*)
返回片段的均方根，即sqrt(sum(S_i^2)/n)。

这是衡量音频信号功率的指标。

audioop.**tomono**(*片段*, *宽度*, *lfactor*, *rfactor*)
将立体声片段转换为单声道片段。左声道乘以 *lfactor*，右声道乘以*rfactor*，然后再添加两个声道以提供单声道信号。

audioop.**tostereo**(*片段*, *宽度*, *lfactor*, *rfactor*)
从单片段生成立体声片段。每对在立体声片段的样品被从样品单声道，由此左声道采样乘以计算*lfactor*通过和右信道样本*rfactor*。

audioop.**ulaw2lin**(*片段*, *宽度*)
将u-LAW编码中的声音片段转换为线性编码的声音片段。u-LAW编码总是使用8位样本，因此*宽度*仅指此处输出片段的样本宽度。

请注意，诸如mul()或max()不区分单声道和立体声片段的操作，即所有样本均被视为相同。如果这是一个问题，立体声片段应该先被分成两个单声道片段，然后重新组合。这里是一个如何做到这一点的例子：

```
def mul_stereo(sample, width, lfactor, rfactor):
    lsample = audioop.tomono(sample, width, 1, 0)
    rsample = audioop.tomono(sample, width, 0, 1)
    lsample = audioop.mul(lsample, width, lfactor)
    rsample = audioop.mul(rsample, width, rfactor)
    lsample = audioop.tostereo(lsample, width, 1, 0)
    rsample = audioop.tostereo(rsample, width, 0, 1)
    return audioop.add(lsample, rsample, width)
```

如果您使用ADPCM编码器来构建网络数据包，并且希望您的协议是无状态的（即能够承受数据包丢失），则不仅应该传输数据，还应该传输状态。请注意，您应该将*初始*状态（传递给您的那个状态`lin2adpcm()`）发送给解码器，而不是最终状态（由编码器返回）。如果要使用`struct.Struct`以二进制形式存储状态，则可以将第一个元素（预测值）编码为16位，将第二个（delta索引）编码为8。

ADPCM编码器从未与其他ADPCM编码器一起使用过，仅针对他们自己。很可能是因为我错误地解释了标准，在这种情况下，他们将不能与各自的标准互操作。

这些`find*()`例程一看起来可能有点有趣。它们主要是为了做回声消除。这样做的一个相当快速的方法是从输出采样中选出最有活力的一块，在输入采样中找到它，并从输入采样中减去整个输出采样：

```
def echocancel(outputdata, inputdata):
    pos = audioop.findmax(outputdata, 800)    # one tenth second
    out_test = outputdata[pos*2:]
    in_test = inputdata[pos*2:]
    ipos, factor = audioop.findfit(in_test, out_test)
    # Optional (for better cancellation):
    # factor = audioop.findfactor(in_test[ipos*2:ipos*2+len(out_test)],
    #              out_test)
    prefill = '\0'*(pos+ipos)*2
    postfill = '\0'*(len(inputdata)-len(prefill)-len(outputdata))
    outputdata = prefill + audioop.mul(outputdata, 2, -factor) + postfill
    return audioop.add(inputdata, outputdata, 2)
```

22.2。aifc- 读写AIFF和AIFC文件

源代码： Lib / aifc.py

该模块提供对读写AIFF和AIFF-C文件的支持。AIFF是音频交换文件格式，一种用于将数字音频样本存储在文件中的格式。AIFF-C是格式的更新版本，包括压缩音频数据的能力。

音频文件有许多描述音频数据的参数。采样率或帧率是声音每秒采样的次数。通道数表明音频是单声道，立体声还是四路。每个帧每个通道包含一个采样。样本大小是每个样本的大小（以字节为单位）。因此一个帧由字节组成，一秒钟的音频由字节组成。`nchannels * samplesizenchannels * samplesize * framerate`

例如，CD质量音频的采样大小为两个字节（16位），使用两个通道（立体声），帧率为44,100帧/秒。这给出了一个4字节（2 * 2）的帧大小，而第二个值占用了2 * 2 * 44100字节（176,400字节）。

模块`aifc`定义了以下功能：

aifc.**open**（*文件*，*模式=无*）

打开AIFF或AIFF-C文件并使用下面描述的方法返回对象实例。参数*文件*是命名文件或文件对象的字符串。*模式*必须是`'r'`或者`'rb'`当文件必须打开以供阅读时，`'w'`或者`'wb'`当文件必须打开以供写入时。如果省略，`file.mode`则在使用时使用，否则`'rb'`使用。当用于书写，文件对象应该是可寻址的，除非你事先知道你要多少个样品中总编写和使用`writeframesraw()`和`setnframes()`。该`open()`函数可用于`with`声明中。当`with`块完成时，`close()`方法被调用。

在版本3.4中进行了更改：`with`添加了对语句的支持。

`open()`打开文件读取时返回的对象具有以下方法：

aifc.**getnchannels**（）

返回音频通道的数量（1个用于单声道，2个用于立体声）。

aifc.**getsampwidth**（）

返回单个样本的字节大小。

aifc.**getframerate**（）

返回采样率（每秒音频帧数）。

aifc.**getnframes**（）

返回文件中的音频帧数。

aifc.**getcomptype**（）

返回描述音频文件中使用的压缩类型的长度为4的字节数组。对于AIFF文件，返回的值是`b'NONE'`。

aifc.**getcompname**（）

将可转换的字节数组返回到音频文件中使用的压缩类型的可读描述。对于AIFF文件，返回的值是。b'not compressed'

aifc.**getparams**( )

返回a ，等同于方法的输出 。namedtuple() (nchannels, sampwidth, framerate, nframes, comptype, compname) get*()

aifc.**getmarkers**( )

返回音频文件中的标记列表。标记由三个元素的元组组成。第一个是标记ID（一个整数），第二个是从数据的起始位置开始的标记位置（一个整数），第三个是标记的名称（一个字符串）。

aifc.**getmark**(*id*)

getmarkers()按照给定*id*中的标记 返回元组。

aifc.**readframes**(*nframes*)

从音频文件中读取并返回下一个*nframes*帧。返回的数据是一个字符串，包含每个帧的所有通道的未压缩样本。

aifc.**rewind**( )

倒回读指针。下一步readframes()将从头开始。

aifc.**setpos**(*pos*)

寻找到指定的帧号。

aifc.**tell**( )

返回当前帧号。

aifc.**close**( )

关闭AIFF文件。调用此方法后，该对象不能再使用。

open()打开文件以便写入时返回的对象具有上述所有方法，除了readframes()和setpos()。另外还有以下方法。这些get*()方法只能在相应的set*()方法被调用后调用。在第一个writeframes()或之前 writeframesraw() ，必须填写除帧数之外的所有参数。

aifc.**aiff**( )

创建一个AIFF文件。默认情况下是创建一个AIFF-C文件，除非文件的名称结束，'.aiff'在这种情况下，默认是AIFF文件。

aifc.**aifc**( )

创建一个AIFF-C文件。默认情况下是创建一个AIFF-C文件，除非文件的名称结束，'.aiff'在这种情况下，默认是AIFF文件。

aifc.**setnchannels**(*nchannels*)

指定音频文件中的通道数量。

aifc.**setsampwidth**(*宽度*)

指定音频采样的大小（以字节为单位）。

aifc.**setframerate**(*费率*)
以每秒帧数指定采样频率。

aifc.**setnframes**(*nframes*)
指定要写入音频文件的帧数。如果此参数未设置或未正确设置，则该文件需要支持查找。

aifc.**setcomptype**(*type*, *name*)
指定压缩类型。如果未指定，音频数据将不会被压缩。在AIFF文件中，压缩是不可能的。名称参数应该是压缩型的字节阵列的人类可读的描述中，类型参数应该是长度为4的一个字节阵列目前使下面的压缩类型的支持：b'NONE'，b'ULAW'，b'ALAW'，b'G722'。

aifc.**setparams**(*nchannels*, *sampwidth*, *帧率*, *comptype*, *COMPNAME*)
一次设置所有上述参数。参数是一个由各种参数组成的元组。这意味着可以使用getparams()调用的结果 作为参数setparams()。

aifc.**setmark**(*id*, *pos*, *name*)
用给定的id（大于0）添加一个标记，并在给定的位置添加给定的名字。这个方法可以在之前的任何时候调用close()。

aifc.**tell**()
返回输出文件中的当前写入位置。与...结合使用setmark()。

aifc.**writeframes**(*数据*)
将数据写入输出文件。此方法只能在音频文件参数设置完成后调用。

*版本3.4中更改：*现在接受任何类似字节的对象。

aifc.**writeframesraw**(*数据*)
就像writeframes()，除了音频文件的头部没有更新。

*版本3.4中更改：*现在接受任何类似字节的对象。

aifc.**close**()
关闭AIFF文件。该文件的标题被更新以反映音频数据的实际大小。调用此方法后，该对象不能再使用。

22.3。 sunau- 读取和写入Sun AU文件

源代码： Lib / sunau.py

该sunau模块为Sun AU声音格式提供了一个方便的界面。请注意，该模块与模块aifc和接口兼容 wave。

一个音频文件由一个头部和数据组成。标题的字段是：

领域	内容
魔术词	四个字节. snd。
标题大小	标题的大小，包括信息，以字节为单位。
数据大小	数据的物理大小，以字节为单位。
编码	指定音频样本的编码方式。
采样率	每秒采样。
通道数量	样本中的通道数量。
信息	描述音频文件的ASCII字符串（用空字节填

除信息字段外，所有标题字段的大小均为4个字节。它们都是以big-endian字节顺序编码的32位无符号整数。

该sunau模块定义了以下功能：

sunau. open （ *文件* ， *模式* ）

如果*file*是一个字符串，则按该名称打开该文件，否则将其视为可搜索的文件类对象。 *模式* 可以是任何

’r’

只读模式。

’w’

只写模式。

请注意，它不允许读/写文件。

甲*模式*的’r’返回的AU_read对象，而一个*模式*的’w’或’wb’返回的AU_write对象。

sunau. openfp （ *文件* ， *模式* ）

open()保持向后兼容性的同义词。

该sunau模块定义了以下例外情况：

*异常*sunau. Error

由于Sun AU规范或实施缺陷而导致不可能发生的错误时引发错误。

该sunau模块定义了以下数据项目：

sunau. AUDIO_FILE_MAGIC

每个有效的Sun AU文件都以一个整数开始，以大端形式存储。这是. snd解释为整数的字符串。

sunau. AUDIO_FILE_ENCODING_MULAW_8

sunau.AUDIO_FILE_ENCODING_LINEAR_8
sunau.AUDIO_FILE_ENCODING_LINEAR_16
sunau.AUDIO_FILE_ENCODING_LINEAR_24
sunau.AUDIO_FILE_ENCODING_LINEAR_32
sunau.AUDIO_FILE_ENCODING_ALAW_8
AU模块支持的编码字段的值。

sunau.AUDIO_FILE_ENCODING_FLOAT
sunau.AUDIO_FILE_ENCODING_DOUBLE
sunau.AUDIO_FILE_ENCODING_ADPCM_G721
sunau.AUDIO_FILE_ENCODING_ADPCM_G722
sunau.AUDIO_FILE_ENCODING_ADPCM_G723_3
sunau.AUDIO_FILE_ENCODING_ADPCM_G723_5
来自AU头部的编码字段的其他已知值，但是该模块不支持这些值。

22.3.1。AU_read对象

open()如上所述，AU_read对象具有以下方法：

AU_read.close( )
关闭流，并使实例无法使用。（这在删除时自动调用。）

AU_read.getnchannels( )
返回音频通道数（1个用于单声道，2个用于立体声）。

AU_read.getsampwidth( )
以字节为单位返回采样宽度

AU_read.getframerate( )
返回采样频率。

AU_read.getnframes( )
返回音频帧的数量。

AU_read.getcomptype( )
返回压缩类型。支持的压缩类型是'ULAW'，'ALAW'和'NONE'。

AU_read.getcompname( )
人类可读的版本getcomptype()。支持的类型有各自的名称，和。'CCITT G.711 u-law''CCITT G.711 A-law''not compressed'

AU_read.getparams( )
返回a，等同于方法的输出。namedtuple() (nchannels, sampwidth, framerate, nframes, comptype, compname)get*()

AU_read.readframes(*n*)

读取和返回至多*n*帧音频，作为bytes对象。数据将以线性格式返回。如果原始数据采用u-LAW格式，则将被转换。

AU_read.**rewind**( )
将文件指针倒回到音频流的开头。

以下两种方法定义了它们之间兼容的术语“位置”，并且在其他方面与实施相关。

AU_read.**setpos**(*pos*)
将文件指针设置为指定的位置。只有返回的值 tell() 应该用于*pos*。

AU_read.**tell**( )
返回当前文件指针位置。请注意，返回的值与文件中的实际位置无关。

下面两个函数是为了兼容而定义的 aifc ，不要做任何有趣的事情。

AU_read.**getmarkers**( )
退货None。

AU_read.**getmark**(*id*)
引发错误。

22.3.2。AU_write对象

open() 如上所述，AU_write对象具有以下方法：

AU_write.**setnchannels**(*n*)
设置通道数量。

AU_write.**setsampwidth**(*n*)
设置样本宽度（以字节为单位）。

*版本3.4中更改：*增加了对24位样本的支持。

AU_write.**setframerate**(*n*)
设置帧频。

AU_write.**setnframes**(*n*)
设置帧数。这可以在以后更改，如果更多的帧被写入。

AU_write.**setcomptype**(*type* , *name*)
设置压缩类型和说明。只有’NONE’和’ULAW’支持输出。

AU_write.**setparams**(*元组*)
的*元组*应该是，凭有效的值的方法。设置所有参数。(nchannels, sampwidth, framerate, nframes, comptype, compname) set*()

AU_write.**tell**( )

返回文件中的当前位置，对于AU_read.tell()和AU_read.setpos()方法使用相同的免责声明。

AU_write.**writeframesraw**(*数据*)

编写音频帧，而不更正*nframes*。

*版本3.4中更改：*现在接受任何类似字节的对象。

AU_write.**writeframes**(*数据*)

写音频帧并确保*nframes*是正确的。

*版本3.4中更改：*现在接受任何类似字节的对象。

AU_write.**close**( )

确保*nframes*是正确的，然后关闭文件。

这个方法在删除时被调用。

请注意，在调用writeframes() 或设置任何参数后无效 writeframesraw()。

22.4。wave- 读写WAV文件

源代码： Lib / wave.py

该wave模块为WAV声音格式提供了一个方便的界面。它不支持压缩/解压缩，但它支持单声道/立体声。

该wave模块定义了以下功能和异常：

wave.open（*文件*，*模式=无*）

如果*file*是一个字符串，则按该名称打开该文件，否则将其视为文件类对象。 *模式*可以是：

'rb'

只读模式。

'wb'

只写模式。

请注意，它不允许读/写WAV文件。

甲*模式*的'rb'返回一个Wave_read对象，而一个*模式*的 'wb'返回一个Wave_write对象。如果省略*模式*并将文件类对象作为*文件*传递，file.mode则将其用作*模式*的默认值。

如果你传入一个类似文件的对象，当它的close()方法被调用时，wave对象不会关闭它；关闭文件对象是调用者的责任。

该 open() 函数可用于 with 声明中。当 with 块完成时，Wave_read.close() 或 Wave_write.close()方法被调用。

*在版本3.4中进行了更改：*添加了对不可查找文件的支持。

wave.openfp（*文件*，*模式*）

open()保持向后兼容性的同义词。

*异常*wave.Error

当某件事情不可能发生时会引发错误，因为它违反了WAV规范或触及实施缺陷。

22.4.1。Wave_read对象

返回的Wave_read对象open()具有以下方法：

Wave_read.close（）

如果流被打开wave，请关闭流，并使实例无法使用。这是在对象收集时自动调用的。

Wave_read.getnchannels（）

返回音频通道数（1单声道，2立体声）。

Wave_read.**getsampwidth**( )
以字节为单位返回采样宽度

Wave_read.**getframerate**( )
返回采样频率。

Wave_read.**getnframes**( )
返回音频帧的数量。

Wave_read.**getcomptype**( )
返回压缩类型（'NONE'是唯一受支持的类型）。

Wave_read.**getcompname**( )
人类可读的版本getcomptype()。通常 平行。'not compressed''NONE'

Wave_read.**getparams**( )
返回a ，等同于方法的输出 。namedtuple() (nchannels, sampwidth, framerate, nframes, comptype, compname)get*()

Wave_read.**readframes**(*n*)
读取和返回至多*n*帧音频，作为bytes对象。

Wave_read.**rewind**( )
将文件指针倒回到音频流的开头。

为了与aifc 模块兼容定义以下两种方法，并且不要做任何有趣的事情。

Wave_read.**getmarkers**( )
退货None。

Wave_read.**getmark**(*id*)
引发错误。

以下两种方法定义了它们之间兼容的术语“位置”，并且在其他方面与实施相关。

Wave_read.**setpos**(*pos*)
将文件指针设置为指定的位置。

Wave_read.**tell**( )
返回当前文件指针位置。

22.4.2。Wave_write对象

对于可搜索的输出流，wave头将自动更新以反映实际写入的帧数。对于不可检测的数据流，写入第一帧数据时， *nframes*值必须准确。通过调用或调用之前将要写入的帧数，然后使用写入帧数据，或通过调用所有要写入的帧数据，可以实现精确的*nframes*值 。在后一种情况下，将

计算数据中的帧数，并在写入帧数据之前相应地设置 *n* 帧。setnframes() setparams() close() writeframesraw() writeframes() writeframes()

返回的Wave_write对象open()具有以下方法：

*在版本3.4中进行了更改：*添加了对不可查找文件的支持。

Wave_write.close()
确保*nframes*是正确的，并关闭文件，如果它被打开 wave。这个方法在对象收集时被调用。如果输出流不可*查找*并且*nframes*与实际写入的帧数不匹配，则会引发异常。

Wave_write.setnchannels(*n*)
设置通道数量。

Wave_write.setsampwidth(*n*)
将示例宽度设置为*n*个字节。

Wave_write.setframerate(*n*)
将帧速率设置为*n*。

*在版本3.2中更改：*此方法的非整数输入四舍五入到最接近的整数。

Wave_write.setnframes(*n*)
将帧数设置为*n*。如果实际写入的帧数不同（如果输出流不可搜索，则此更新尝试将引发错误），这将稍后更改。

Wave_write.setcomptype(*type*, *name*)
设置压缩类型和说明。目前，只NONE支持压缩类型 ，意味着不压缩。

Wave_write.setparams(*元组*)
的*元组*应该是，凭有效的值的方法。设置所有参数。(nchannels, sampwidth, framerate, nframes, comptype, compname) set*()

Wave_write.tell()
返回文件中的当前位置，对于Wave_read.tell()和Wave_read.setpos()方法使用相同的免责声明 。

Wave_write.writeframesraw(*数据*)
编写音频帧，而不更正*nframes*。

*版本3.4中更改：*现在接受任何类似字节的对象。

Wave_write.writeframes(*数据*)
写音频帧并确保*nframes*是正确的。如果输出流不可查找，并且在写入*数据*之后已写入的帧的总数与先前为*nframe*设置的值不匹配，*则会引发错误*。

*版本3.4中更改：*现在接受任何类似字节的对象。

请注意，在调用writeframes() 或之后设置任何参数都是无效的writeframesraw()，并且任何尝试都会引发 wave.Error。

22.5。 chunk- 读取IFF分块数据

源代码： Lib / chunk.py

该模块提供读取使用EA IFF 85块的文件的界面。 [1] 该格式至少用于音频交换文件格式（AIFF / AIFF-C）和真实媒体文件格式（RMFF）。WAVE音频文件格式密切相关，也可以使用此模块进行读取。

块具有以下结构：

抵消	长度	内容
0	4	块ID
4	4	以大端字节顺序的块大小，不包括头
8	*ñ*	数据字节，其中*n*是前面字段中给出的大小
8 + *n*	0或1	如果*n*是奇数并且使用块对齐，则需要填充字节

ID是一个4字节的字符串，用于标识块的类型。

大小字段（一个32位值，使用big-endian字节顺序编码）给出了块数据的大小，不包括8字节的头。

通常一个IFF类型的文件由一个或多个块组成。Chunk这里定义的类的建议用法是在每个块的开始处实例化一个实例，并从实例中读取，直到它到达结尾，之后可以实例化一个新实例。在文件结尾处，创建新实例将失败并出现EOFError异常。

class chunk. Chunk（*file*，*align* = *True*，*bigendian* = *True*，*inclheader* = *False* ）

表示块的类。该*文件*的说法，预计是一个类似文件的对象。这个类的一个实例是特别允许的。唯一需要的方法是read()。如果方法 seek()和tell()存在并没有引发异常，它们也可使用。如果这些方法存在并引发异常，则预计它们不会改变对象。如果可选参数*align*为true，则假定块在2字节边界上对齐。如果*对齐*错误，则不假定对齐。默认值是true。如果可选参数 *bigendian*是假的，块大小被假定为小端顺序。这是WAVE音频文件所必需的。默认值是true。如果可选参数*inclheader*为true，则*块头*中给出的大小包括头的大小。默认值是false。

一个Chunk对象支持以下方法：

getname（ ）

返回块的名称（ID）。这是块的前4个字节。

getsize（ ）

返回块的大小。

close（ ）

关闭并跳到块的末尾。这不会关闭底层文件。

OSError如果在调用该close()方法后调用其余方法，则会引发该 方法。在Python 3.3之前，他们曾经提出过IOError，现在是一个别名OSError。

isatty( )

退货False。

seek(*pos*，*whence = 0*)

设置块的当前位置。的*何处*参数是可选的，缺省值为0（绝对文件定位）；其他值是1（相对于当前位置2查找）和（相对于文件末尾查找）。没有返回值。如果底层文件不允许搜索，则只允许前向搜索。

tell( )

将当前位置返回到块中。

read(*size = -1*)

从块中读取大部分*大小的*字节（如果在获取*大小*字节之前读取到块的末尾，则读取更少）。如果*size*参数为负值或省略，请读取所有数据直到块结束。当块立即遇到时，返回一个空字节对象。

skip( )

跳到块的末尾。所有进一步的调用read()块将返回b''。如果您对块的内容不感兴趣，应该调用此方法，以便文件指向下一个块的开始。

脚注

[1] “EA IFF 85”交换格式文件标准，Jerry Morrison，Electronic Arts，1985年1月。

22.6。colorsys- 颜色系统之间的转换

源代码： Lib / colorsys.py

该colorsys模块定义了计算机显示器和其他三个坐标系统：YIQ，HLS（色调亮度饱和度）和HSV（色相饱和度值）中使用的RGB（红绿蓝）色彩空间中表达的颜色之间的颜色值双向转换。所有这些色彩空间中的坐标都是浮点值。在YIQ空间中，Y坐标在0和1之间，但I和Q坐标可以是正值或负值。在所有其他空间中，坐标都在0和1之间。

也可以看看： 有关色彩空间的更多信息，请参见 http://www.poynton.com/ColorFAQ.html和 https://www.cambridgeincolour.com/tutorials/color-spaces.htm。

该colorsys模块定义了以下功能：

colorsys.**rgb_to_yiq**(*r*, *g*, *b*)
将颜色从RGB坐标转换为YIQ坐标。

colorsys.**yiq_to_rgb**(*y*, *i*, *q*)
将颜色从YIQ坐标转换为RGB坐标。

colorsys.**rgb_to_hls**(*r*, *g*, *b*)
将颜色从RGB坐标转换为HLS坐标。

colorsys.**hls_to_rgb**(*h*, *l*, *s*)
将颜色从HLS坐标转换为RGB坐标。

colorsys.**rgb_to_hsv**(*r*, *g*, *b*)
将颜色从RGB坐标转换为HSV坐标。

colorsys.**hsv_to_rgb**(*h*, *s*, *v*)
将颜色从HSV坐标转换为RGB坐标。

例：

```
>>> import colorsys
>>> colorsys.rgb_to_hsv(0.2, 0.4, 0.4)
(0.5, 0.5, 0.4)
>>> colorsys.hsv_to_rgb(0.5, 0.5, 0.4)
(0.2, 0.4, 0.4)
```

22.7。 imghdr- 确定图像的类型

源代码： Lib / imghdr.py

该imghdr模块确定包含在文件或字节流中的图像类型。

该imghdr模块定义了以下功能：

imghdr.what（*文件名*，*h* =*无*）

测试由*文件名*命名的*文件中*包含的图像数据，并返回描述图像类型的字符串。如果提供了可选的*h*，则忽略*文件名*，并假定*h*包含要测试的字节流。

*在版本3.6中更改：*接受类似路径的对象。

以下图像类型被识别，如下所示，返回值来自what()：

值	图像格式
'rgb'	SGI ImgLib文件
'gif'	GIF 87a和89a文件
'pbm'	便携式位图文件
'pgm'	便携式灰度图文件
'ppm'	便携式Pixmap文件
'tiff'	TIFF文件
'rast'	Sun光栅文件
'xbm'	X位图文件
'jpeg'	据IF或Exif格式的JPEG数
'bmp'	BMP文件
'png'	便携式网络图形
'webp'	WebP文件
'exr'	OpenEXR文件

*新的3.5版：*在*EXR*和*WEBP*添加格式。

您可以扩展imghdr可通过附加到此变量来识别的文件类型列表：

imghdr.tests

执行单个测试的功能列表。每个函数都有两个参数：字节流和打开的文件类对象。当what()用字节流调用时，文件类对象将会是None。

如果测试成功或测试None失败，测试函数应返回描述图像类型的字符串。

例：

```
>>> import imghdr
>>> imghdr.what('bass.gif')
'gif'
```

22.8。 sndhdr- 确定声音文件的类型

源代码： Lib / sndhdr.py

在sndhdr提供了试图确定的声音数据，其是在一个文件中的类型的实用功能。当这些功能能够确定什么类型的声音数据被存储在一个文件中，他们返回一个 namedtuple()包含五个属性：（，filetype， framerate，nchannels，）。 nframes sampwidth为值*类型*指示数据的类型和将是字符串中的一个'aifc'，'aiff'，'au'，'hcom'，'sndr'，'sndt'，'voc'，'wav'，'8svx'，'sb'，'ub'，或'ul'。该*sampling_rate*位将是实际值或0如果不明或难以解码。同样，*频道*或者是频道的数量或者0如果无法确定或值难以解码。*帧*的值将是帧数或-1。元组中的最后一项，*bits_per_sample*，或者是以位'A'为单位的样本大小，或者是A-LAW或者'U'u-LAW。

sndhdr.what（*文件名*）

确定存储在该文件的声音数据的类型*的文件名*使用 whathdr()。如果成功，则返回如上所述的namedtuple，否则 None返回。

*版本3.5中更改：*结果从元组更改为namedtuple。

sndhdr.whathdr（*文件名*）

根据文件头确定存储在文件中的声音数据的类型。文件的名称由*文件名*给出。这个函数返回一个如上所述的成功的namedtuple，或者None。

*版本3.5中更改：*结果从元组更改为namedtuple。

22.9。ossaudiodev- 访问OSS兼容音频设备

该模块允许您访问OSS（开放式声音系统）音频接口。OSS适用于广泛的开源和商业Unices，是Linux和FreeBSD最新版本的标准音频接口。

*在版本3.3中进行了更改：*此模块中的操作现在引发引发的OSError地方IOError 。

也可以看看:

打开声音系统程序员指南
OSS C API的官方文档

该模块定义了由OSS设备驱动程序提供的大量常量; 见<sys/soundcard.h>在Linux或者FreeBSD的列表项。

ossaudiodev 定义了以下变量和函数：

*异常*ossaudiodev.OSSAudioError
这个例外在某些错误中引发。参数是一个描述出错的字符串。

（如果ossaudiodev从系统调用收到错误时，例如 open() ，write() ，或ioctl() ，它提出了OSError通过直接检测到的错误ossaudiodev结果OSSAudioError。 ）

（为了向后兼容，异常类也可用 ossaudiodev.error。 ）

ossaudiodev.open（*模式*）
ossaudiodev.open（*设备*，*模式*）
打开音频设备并返回OSS音频设备对象。此对象支持许多类文件的方法，如read() ，write()和 fileno() （虽然有读常规的Unix之间的细微差别/写语义和那些OSS音频设备）。它还支持许多音频特定的方法; 请参阅下面的方法的完整列表。

*设备*是要使用的音频设备文件名。如果没有指定，则该模块首先查找环境变量AUDIODEV供设备使用。如果找不到，则返回/dev/dsp。

*模式*是'r'只读（记录）访问，'w'只写（回放）访问和'rw'两者之一。由于许多声卡只允许一个进程一次打开录像机或播放器，因此只有为需要的活动打开设备是一个好主意。此外，一些声卡是半双工的：它们可以打开以便读取或写入，但不能同时打开。

注意不寻常的调用语法：第一个参数是可选的，第二个参数是必需的。这是与替代旧linuxaudiodev模块兼容的历史工件ossaudiodev。

ossaudiodev.openmixer（[*device*]）
打开混音器设备并返回OSS混音器设备对象。 *设备*是要使用的混合器设备文件名。如果没有指定，则该模块首先查找环境变量MIXERDEV供设备使用。如果找不到，则返回/dev/mixer。

22.9.1。音频设备对象

在您可以写入或读取音频设备之前，您必须按正确的顺序调用三种方法：

1. setfmt() 设置输出格式
2. channels() 设置通道数
3. speed() 设置采样率

或者，您可以使用该setparameters()方法一次设置全部三个音频参数。这样更方便，但在所有情况下可能都不那么灵活。

通过open()定义以下方法和（只读）属性返回的音频设备对象：

oss_audio_device.close（）
明确关闭音频设备。当您完成写入或读取音频设备时，应该明确地关闭它。封闭的设备不能再次使用。

oss_audio_device.fileno（）
返回与设备关联的文件描述符。

oss_audio_device.read（*size*）
从音频输入中读取*大小*字节并将它们作为Python字符串返回。与大多数Unix设备驱动程序不同，处于阻止模式（默认设置）的OSS音频设备将阻塞，read()直到整个请求的数据量可用。

oss_audio_device.write（*数据*）
将类似字节的对象*数据*写入音频设备并返回写入的字节数。如果音频设备处于阻止模式（默认设置），则始终写入整个数据（同样，这与通常的Unix设备语义不同）。如果设备处于非阻塞模式，则可能不会写入某些数据 - 请参阅writeall()。

*在版本3.5中更改：*现在可以接受可写入类似字节的对象。

oss_audio_device.writeall（*数据*）
将类似字节的对象*数据*写入音频设备：等到音频设备能够接受数据，写入尽可能多的数据，然后重复*数据*直到*数据*完全写入。如果设备处于阻止模式（默认设置），则与此效果相同 write()；writeall()仅在非阻塞模式下有用。没有返回值，因为写入的数据量总是等于所提供的数据量。

*在版本3.5中更改：*现在可以接受可写入类似字节的对象。

*在版本3.2中更改：*音频设备对象也支持上下文管理协议，即它们可以在with语句中使用。

以下方法分别映射到一个ioctl()系统调用。的对应是显然的：例如，setfmt()对应于所述SNDCTL_DSP_SETFMT的ioctl，和sync()到SNDCTL_DSP_SYNC（咨询OSS文档时这可能是有用的）。如果底层 ioctl()失败，他们都会升起OSError。

oss_audio_device.nonblock（）
将设备置于非阻塞模式。一旦进入非阻塞模式，无法将其返回到阻塞模式。

oss_audio_device. getfmts ( )

返回声卡支持的音频输出格式的掩码。OSS支持的一些格式是：

格式	描述
AFMT_MU_LAW	对数编码（由Sun .au 文件使用/dev/audio ）
AFMT_A_LAW	对数编码
AFMT_IMA_ADPCM	由交互式多媒体协会定义的4：1压缩格式
AFMT_U8	无符号的8位音频
AFMT_S16_LE	已签名的16位音频，小端字节顺序（由英特尔处理器使用）
AFMT_S16_BE	已签名的16位音频，大端字节顺序（68k，PowerPC，Sparc使用）
AFMT_S8	签名，8位音频
AFMT_U16_LE	无符号的16位小端音频
AFMT_U16_BE	无符号的16位大端音频

请查阅OSS文档以获取音频格式的完整列表，并注意大多数设备仅支持这些格式的一部分。一些较旧的设备仅支持AFMT_U8; 今天使用的最常见的格式是 AFMT_S16_LE。

oss_audio_device. setfmt (*格式*)

尝试将当前音频格式设置为*格式* - 请参阅getfmts()列表。返回设备设置为的音频格式，可能不是所请求的格式。也可用于返回当前的音频格式 - 通过传递“音频格式”来完成此操作AFMT_QUERY。

oss_audio_device. channels (*nchannels*)

将输出通道的数量设置为*nchannels*。值1表示单声道，2立体声。某些设备可能有两个以上的通道，而一些高端设备可能不支持单声道。返回设备设置的通道数量。

oss_audio_device. speed (*samplerate*)

尝试将音频采样率设置为每秒采样率*采样率*。返回实际设置的比率。大多数声音设备不支持任意采样率。常见的价格是：

率	描述
8000	违约率 /dev/audio
11025	语音录制
22050	
44100	CD质量音频（16位/采样和2通道）
96000	DVD质量音频（24位/采样）

oss_audio_device. sync ( )

等到声音设备播放其缓冲区中的每个字节。（当设备关闭时，会发生这种情况。）OSS文档建议关闭并重新打开设备，而不是使用sync()。

oss_audio_device.**reset**（）

立即停止播放或录制，并将设备恢复到可以接受命令的状态。OSS文档建议在呼叫后关闭并重新打开设备`reset()`。

oss_audio_device.**post**（）

告诉驱动程序，输出中可能会出现暂停，使设备可以更智能地处理暂停。在等待用户输入之前或在执行磁盘I / O之前，您可以在播放专辑效果之后使用此功能。

以下便利方法结合了几个ioctl，或一个ioctl和一些简单的计算。

oss_audio_device.**setparameters**（*格式*，*nchannels*，*采样率*[，*严格=假*]）

在一次方法调用中设置关键音频采样参数 - 采样格式，通道数量和采样率。 *格式*，*nchannels*，和*采样率*应该在中的规定`setfmt()`，`channels()`，和 `speed()` 方法。如果*严格*为真，则`setparameters()`检查每个参数是否实际设置为所请求的值，`OSSAudioError`如果不是则引发 。返回一个元组（*格式*，*nchannels*， *采样率*表示实际上是由设备驱动器（即，相同的返回值设置的参数值）`setfmt()`， `channels()`和`speed()`）。

例如，

```
(fmt, channels, rate) = dsp.setparameters(fmt, channels, rate)
```

相当于

```
fmt = dsp.setfmt(fmt)
channels = dsp.channels(channels)
rate = dsp.rate(rate)
```

oss_audio_device.**bufsize**（）

以样本形式返回硬件缓冲区的大小。

oss_audio_device.**obufcount**（）

返回硬件缓冲区中尚未播放的样本数。

oss_audio_device.**obuffree**（）

返回可以排队到硬件缓冲区中的样本数量，以便不被阻塞地播放。

音频设备对象还支持多种只读属性：

oss_audio_device.**closed**

布尔值，指示设备是否已关闭。

oss_audio_device.**name**

包含设备文件名称的字符串。

oss_audio_device.**mode**

该文件的I / O模式，要么`"r"`，`"rw"`或`"w"`。

22.9.2。混音器设备对象

混音器对象提供了两种类似文件的方法：

oss_mixer_device.**close**()
该方法关闭打开的混音器设备文件。在该文件关闭后再次尝试使用混音器将会引发一次OSError。

oss_mixer_device.**fileno**()
返回打开的混音器设备文件的文件句柄编号。

*在版本3.2中更改：*混音器对象也支持上下文管理协议。

其余的方法特定于音频混合：

oss_mixer_device.**controls**()
此方法返回一个指定可用混音器控件的位掩码（“Control”是特定的可混音“通道”，如SOUND_MIXER_PCMor SOUND_MIXER_SYNTH）。该位掩码表示所有可用混音器控件的子集 - SOUND_MIXER_*在模块级定义的常量。例如，要确定当前的混音器对象是否支持PCM混音器，请使用以下Python代码：

```
mixer=ossaudiodev.openmixer()
if mixer.controls() & (1 << ossaudiodev.SOUND_MIXER_PCM):
    # PCM is supported
    ... code ...
```

对于大多数用途来说，SOUND_MIXER_VOLUME（主音量）和 SOUND_MIXER_PCM控件应该就足够了 - 但是在选择混音器控件时，使用混音器的代码应该是灵活的。例如，在重症超声SOUND_MIXER_VOLUME中不存在。

oss_mixer_device.**stereocontrols**()
返回指示立体声混音器控件的位掩码。如果设置了一位，则相应的控制是立体声; 如果未设置，则控制器为单声道或不支持混音器（与controls()确定哪一个配合使用）。

有关controls()从位掩码获取数据的示例，请参阅该函数的代码示例。

oss_mixer_device.**reccontrols**()
返回指定可用于记录的混音器控件的位掩码。有关controls()从位掩码读取的示例，请参阅代码示例。

oss_mixer_device.**get**(*控制*)
返回给定混音器控件的音量。返回的卷是一个2元组 (left_volume,right_volume)。音量指定为从0（无声）到100（全音量）的数字。如果控件是单音的，则2元组仍然返回，但两个音量都是相同的。

OSSAudioError如果指定了无效控件，或者指定 OSError了不支持的控件，则引发。

oss_mixer_device.**set**(*控制*, *(左, 右)*)
将给定混音器控制的音量设置为(left,right)。 left并且 right必须是整数并且在0（无声）和100（全音量）之间。成功时，新卷将作为2元组返回。请注意，这可能与指定的音

量不完全相同，因为某些声卡混音器的分辨率有限。

OSSAudioError如果指定了无效的混音器控制，或者指定的音量超出范围，则会引发。

oss_mixer_device.**get_recsrc**( )

此方法返回一个位掩码，指示当前正在将哪些控件用作记录源。

oss_mixer_device.**set_recsrc**(*bitmask*)

调用此功能指定一个录音源。如果成功，则返回指示新录制源（或多个源）的位掩码; OSError如果指定了无效源，则会引发此问题。将当前录音源设置为麦克风输入：

```
mixer.setrecsrc (1 << ossaudiodev.SOUND_MIXER_MIC)
```

23.国际化

本章描述的模块可帮助您编写独立于语言和语言环境的软件，方法是提供选择程序消息中使用的语言的机制，或者通过剪裁输出以匹配本地约定。

本章描述的模块列表是：

23.1。gettext- 多语言国际化服务

源代码： Lib / gettext.py

该gettext模块为您的Python模块和应用程序提供国际化（I18N）和本地化（L10N）服务。它支持GNU gettext消息目录API和更适合Python文件的更高级别的基于类的API。下面介绍的接口允许您使用一种自然语言编写模块和应用程序消息，并提供用不同自然语言运行的翻译消息目录。

还给出了一些关于本地化Python模块和应用程序的提示。

23.1.1。GNU 的gettext API

该gettext模块定义了以下API，它与GNU **gettext** API 非常相似。如果您使用此API，则会影响整个应用程序的全局翻译。如果您的应用程序是单语言的，通常这就是您想要的，并且语言的选择取决于用户的区域设置。如果您正在本地化Python模块，或者您的应用程序需要即时切换语言，则可能需要使用基于类的API。

gettext.bindtextdomain（*domain*，*localedir* = *None* ）

将*域*绑定到locale目录*localedir*。更具体地说， gettext将.mo使用路径（在Unix上）查找给定域的二进制文件：localedir/language/LC_MESSAGES/domain.mo，其中 在环境变量中搜索*语言*LANGUAGE， LC_ALL， LC_MESSAGES，和 LANG 分别。

如果*localedir*被省略None，则返回当前的*域*绑定。[1]

gettext.bind_textdomain_codeset（*域*，*codeset* =*无* ）

结合*域*到*代码集*，改变字节串的由返回的编码lgettext()，ldgettext()，lngettext() 和ldngettext()功能。如果*代码集*被省略，则返回当前绑定。

gettext.textdomain（*domain* = *None* ）

更改或查询当前全局域。如果*域*为None，则返回当前全局域，否则全局域设置 为返回的*域*。

gettext.gettext（*消息* ）

根据当前全局域，语言和区域设置目录返回*消息*的本地化转换。该函数通常_()与本地名称空间中的别名一样（请参见下面的示例）。

gettext.dgettext（*域*，*消息* ）

喜欢gettext()，但在指定的*域中*查看消息。

gettext.ngettext（*单数*，*复数*，*n* ）

喜欢gettext()，但考虑复数形式。如果找到翻译，则将复数公式应用于*n*，并返回结果信息（某些语言具有两个以上的复数形式）。如果没有找到翻译，如果*n*是1，则返回 *单数*；否则返回*复数*。

复制公式从目录标题中获取。它是一个C或Python表达式，它有一个自由变量*n*；表达式评估为目录中复数的索引。有关在文件中 使用的精确语法.po以及各种语言的公式，请参阅GNU gettext文档。

gettext.**dngettext**(*域*, *单数*, *复数*, *n*)

喜欢ngettext()，但在指定的*域中*查看消息。

gettext.**lgettext**(*消息*)

gettext.**ldgettext**(*域*, *消息*)

gettext.**lngettext**(*单数*, *复数*, *n*)

gettext.**ldngettext**(*域*, *单数*, *复数*, *n*)

相当于无相应的功能l前缀（gettext()，dgettext()，ngettext()和dngettext()），但翻译返回如在优选系统编码进行编码，如果没有其他的编码被显式地设置有字节串bind_textdomain_codeset()。

> **警告：** 在Python 3中应该避免使用这些函数，因为它们会返回编码字节。使用替代方法返回Unicode字符串会更好，因为大多数Python应用程序都希望将字符串替换为字符串而不是字节。此外，如果翻译后的字符串存在编码问题，则可能会出现意想不到的Unicode相关异常。l*()由于它们固有的问题和限制，这些函数可能会在未来的Python版本中被弃用。

请注意，GNU **gettext**也定义了一种dcgettext()方法，但这被认为没有用，所以它目前未实现。

以下是此API的典型用法示例：

```
import gettext
gettext.bindtextdomain('myapplication', '/path/to/my/language/directory')
gettext.textdomain('myapplication')
_ = gettext.gettext
# ...
print(_('This is a translatable string.'))
```

23.1.2。基于类的

该gettext模块的基于类的API 为您提供比GNU **gettext** API 更多的灵活性和更大的便利。这是本地化您的Python应用程序和模块的推荐方式。 gettext定义了一个实现GNU .mo格式文件解析的“翻译”类，并且有返回字符串的方法。这个“翻译”类的实例也可以把它们自己安装在内置的名字空间中作为函数 _()。

gettext.**find**(*domain*, *localedir* = *None*, *languages* = *None*, *all* = *False*)

该功能实现标准.mo文件搜索算法。它需要一个*域*，与需要的相同textdomain()。可选*localedir*与bindtextdomain() 可选*语言一样*是一个字符串列表，其中每个字符串都是一个语言代码。

如果未指定*localedir*，则使用默认的系统区域设置目录。 [2] 如果没有给出*语言*，则搜索以下环境变量：LANGUAGE， LC_ALL， LC_MESSAGES，和 LANG。返回非空值的第一个用于 *语言* 变量。环境变量应该包含一个以冒号分隔的语言列表，这些语言将在冒号上分开以产生预期的语言代码字符串列表。

find() 然后扩展和规范化语言，然后遍历它们，搜索由这些组件构建的现有文件：

localedir/*language*/LC_MESSAGES/*domain*.mo

存在的第一个这样的文件名由返回find()。如果没有找到这样的文件，则None返回。如果*全部*给出，它将按照它们出现在语言列表或环境变量中的顺序返回所有文件名的列表。

gettext.**translation**(*domain*，*localedir* = *None*，*languages* = *None*，*class_* = *None*，*fallback* = *False*，*codeset* = *None*)

返回Translations基于*域*，*localedir*和*语言*的实例，这些实例首先传递find()给获取关联.mo文件路径的列表。.mo缓存具有相同文件名的实例。实例化的实际类或者是*class_*，如果提供的话，否则GNUTranslations。该类的构造函数必须采用单个文件对象参数。如果提供，*代码集*将更改用于对lgettext()和lngettext() 方法中的已翻译字符串进行编码的字符集 。

如果找到多个文件，则以后的文件将用作早期文件的后备文件。允许设置回退，copy.copy()用于从缓存中克隆每个翻译对象; 实际的实例数据仍然与缓存共享。

如果没有.mo找到文件，这个功能引起了OSError，如果 *回退*是假的（这是默认值），并返回 NullTranslations如果实例的*回退*是真实的。

*在版本3.3中更改：*IOError用于提高而不是OSError。

gettext.**install**(*domain*，*localedir* = *None*，*codeset* = *None*，*names* = *None*)

这将_()根据传递给该函数的*域*，*localedir*和*代码集*在Python的builtins名称空间中 安装该函数 translation()。

有关*names*参数，请参阅翻译对象install()方法的描述。

如下所示，通常在应用程序中标记可作为翻译候选对象的字符串，方法是将它们包装在_()函数调用中，如下所示：

```
print(_('This string will be translated.'))
```

为了方便起见，您希望将_()函数安装在Python的builtins名称空间中，以便在应用程序的所有模块中轻松访问它。

23.1.2.1。本NullTranslations类

翻译类实际上是将原始源文件消息字符串转换为翻译的消息字符串。所有翻译类使用的基类是NullTranslations; 这提供了您可以用来编写自己的专业翻译课程的基本界面。以下是这些方法NullTranslations：

class gettext.**NullTranslations**(*fp* = *None*)

获取可选的文件对象 *fp*，该对象被基类忽略。初始化派生类设置的“protected”实例变量 *_info* 和 *_charset* ，以及通过设置的 *_fallback*`add_fallback()` 。然后，它调用 `self._parse(fp)` 如果*FP*不是 `None`。

`_parse(` *fp* `)`

在基类中没有操作，该方法使用文件对象*fp*，并从文件读取数据，初始化其消息目录。如果您的消息目录文件格式不受支持，则应该重写此方法来解析您的格式。

`add_fallback(` *回退* `)`

将*后备*作为当前翻译对象的后备对象添加。如果翻译对象无法为给定的消息提供翻译，则应该查阅翻译对象。

`gettext(` *消息* `)`

如果已设置回退，则转发`gettext()`至回退。否则，返回*消息*。在派生类中重写。

`ngettext(` *单数*，*复数*，*n* `)`

如果已设置回退，则转发`ngettext()`至回退。否则，如果*n*是1 ，则返回*单数*；否则返回*复数*。在派生类中重写。

`lgettext(` *消息* `)`

`lngettext(` *单数*，*复数*，*n* `)`

等同于`gettext()`和`ngettext()` ，但如果没有明确设置编码，则翻译将作为首选系统编码中编码的字节字符串返回`set_output_charset()`。在派生类中重写。

> **警告:** 在Python 3中应该避免使用这些方法。请参阅该`lgettext()`函数的警告 。

`info( )`

返回“受保护的” `_info`变量。

`charset( )`

返回消息目录文件的编码。

`output_charset( )`

返回用于在`lgettext()` 和中返回已翻译消息的编码`lngettext()`。

`set_output_charset(` *charset* `)`

更改用于返回已翻译消息的编码。

`install(` *names = None* `)`

该方法将安装`gettext()`到内置名称空间中，并将其绑定到`_`。

如果给出了*names*参数，那么它必须是一个包含您想要在内建命名空间中安装的函数名称的序列`_()`。支持的名称是`'gettext'`，`'ngettext'`， `'lgettext'`和`'lngettext'`。

请注意，这只是一种方法，尽管这是最方便的方式，可以使`_()`您的应用程序可以使用该功能。因为它影响整个应用程序全局，特别是内置命名空间，本地化模块不应该安装`_()`。相反，他们应该使用此代码使其`_()`模块可用：

```
import gettext
t = gettext.translation('mymodule', ...)
_ = t.gettext
```

这_()只会放在模块的全局名称空间中，因此只会影响此模块中的调用。

23.1.2.2。本GNUTranslations类

该gettext模块提供了一个来自以下的附加类 NullTranslations：GNUTranslations。这个类覆盖了 _parse()以big-endian和little-endian格式读取GNU **gettext**格式.mo文件。

GNUTranslations将可选的元数据从翻译目录中解析出来。GNU **gettext**约定包含元数据作为空字符串的翻译。这个元数据在**RFC 822**风格 对，并且应该包含密钥。如果找到密钥，则该属性用于初始化"受保护的"实例变量，如果未找到则默认为该实例变量 。如果指定了字符集编码，则使用此编码将从目录中读取的所有消息ID和消息字符串转换为Unicode，否则将采用ASCII编码。 key: valueProject-Id-VersionContent-TypecharsetcharsetNone

由于消息标识也被读为Unicode字符串，因此所有*gettext()方法都将假定消息标识为Unicode字符串，而不是字节字符串。

整组键/值对被放置在字典中并被设置为"受保护的" _info实例变量。

如果.mo文件的幻数无效，主版本号是意外的，或者如果在读取文件时发生其他问题，则GNUTranslations可以引发实例化 类OSError。

*类*gettext.GNUTranslations

基类实现覆盖了以下方法：

gettext（*消息*）

在目录中查找*消息* ID并返回相应的消息字符串，作为Unicode字符串。如果*消息*标识的目录中没有条目 ，并且已设置回退，则查找会转发到回退的gettext()方法。否则，返回*消息* ID。

ngettext（*单数*，*复数*，*n* ）

做一个消息ID的复数形式查找。 *单数*用作目录中查找消息的消息ID，而*n*用于确定使用哪种复数形式。返回的消息字符串是一个Unicode字符串。

如果在目录中找不到消息标识，并且指定了回退，则将请求转发到回退的ngettext()方法。否则，当*n*是1时，返回*单数*，并且在所有其他情况下返回*复数*。

这里是一个例子：

```
n = len(os.listdir('.'))
cat = GNUTranslations(somefile)
message = cat.ngettext(
    'There is %(num)d file in this directory',
    'There are %(num)d files in this directory',
    n) % {'num': n}
```

lgettext（*消息*）

lngettext（*单数*，*复数*，*n*）

等同于gettext()和ngettext()，但如果没有明确设置编码，则翻译将作为首选系统编码中编码的字节字符串返回 set_output_charset()。

> **警告:** 在Python 3中应该避免使用这些方法。请参阅该lgettext()函数的警告。

23.1.2.3。Solaris消息目录支持

Solaris操作系统定义了它自己的二进制.mo文件格式，但由于此格式没有文档可用，目前尚不支持。

23.1.2.4。Catalog构造函数

GNOME使用gettextJames Henstridge 的模块版本，但是这个版本有一个稍微不同的API。其记录的使用情况是：

```
import gettext
cat = gettext.Catalog(domain, localedir)
_ = cat.gettext
print(_('hello world'))
```

为了与这个旧模块兼容，该函数Catalog()是上述函数的别名translation()。

这个模块和Henstridge之间的一个区别是：他的目录对象支持通过映射API进行访问，但这似乎是未使用的，所以目前不支持。

23.1.3。国际化您的程序和模块

国际化（I18N）是指使程序知道多种语言的操作。本地化（L10N）是指您的程序一经国际化就适应当地语言和文化习惯。为了为您的Python程序提供多语言消息，您需要执行以下步骤：

1. 通过专门标记可翻译字符串来准备程序或模块
2. 在标记文件上运行一套工具以生成原始消息目录
3. 创建消息目录的语言特定翻译
4. 使用该gettext模块以便消息字符串被正确转换

为了准备I18N的代码，您需要查看文件中的所有字符串。任何需要翻译的字符串应该通过包装来标记_('...')- 也就是对该函数的调用_()。例如：

```
filename = 'mylog.txt'
message = _('writing a log message')
fp = open(filename, 'w')
fp.write(message)
fp.close()
```

在这个例子中，字符串被标记为翻译的候选字符串，而不是。'writing a log message''mylog.txt''w'

有几种工具可以提取用于翻译的字符串。原始GNU **gettext**仅支持C或C ++源代码，但其扩展版本**xgettext**扫描用多种语言（包括Python）编写的代码以查找标记为可翻译的字符串。Babel是一个Python国际化库，包含一个`pybabel`脚本来提取和编译消息目录。弗朗索瓦·皮纳得的程序调用**XPOT**做类似的工作，可作为他的一部分，PO-utils软件包。

（Python还包含这些程序的纯Python版本，称为 **pygettext.py**和**msgfmt.py** ;一些Python发行版将为您安装它们 。**pygettext.py**与**xgettext**类似 ，但只能理解Python源代码，不能处理其他编程语言。如C或C ++ **pygettext.py**支持类似的命令行界面 **了xgettext** ;关于它的使用细节，运行。 **msgfmt.py**与GNU二进制兼容 **的msgfmt**有了这两个方案，则可能不需要GNU。**gettext的**包使您的Python应用程序国际化。 ）`pygettext.py --help`

xgettext，**pygettext**和类似的工具生成 `.po`消息目录的文件。它们是结构化的可读文件，它包含源代码中的每个标记字符串，以及这些字符串的翻译版本的占位符。

`.po`然后将这些文件的副本交给为每种支持的自然语言编写翻译的个人翻译人员。他们将完整的特定于语言的版本作为一个`<language-name>.po`文件发送回`.mo`使用**msgfmt**程序编译为机器可读的二进制目录文件。这些`.mo`文件`gettext`在运行时被 模块用于实际的翻译处理。

`gettext`在代码中如何使用模块取决于您是将单个模块还是整个应用程序国际化。接下来的两节将讨论每个案例。

23.1.3.1。本地化你的模块

如果您正在本地化您的模块，则必须注意不要进行全局更改，例如对内置名称空间进行更改。你不应该使用GNU `gettext` API，而应该使用基于类的API。

假设您的模块被称为“垃圾邮件”，并且模块的各种自然语言翻译`.mo`文件驻留`/usr/share/locale`在GNU **gettext**格式中。以下是您将放在模块顶部的内容：

```
import gettext
t = gettext.translation('spam', '/usr/share/locale')
_ = t.gettext
```

23.1.3.2。本地化应用程序

如果您正在本地化应用程序，则可以将该`_()`函数全局安装到内置命名空间中，通常位于应用程序的主驱动程序文件中。这将让所有应用程序特定的文件都可以使用， `_('...')`而无需将其明确安装到每个文件中。

在简单情况下，您只需将以下代码添加到应用程序的主驱动程序文件中：

```
import gettext
gettext.install('myapplication')
```

如果你需要设置locale目录，你可以将它传递给 `install()`函数：

```
import gettext
gettext.install('myapplication', '/usr/share/locale')
```

23.1.3.3。即时更改语言

如果您的程序需要同时支持多种语言，则可能需要创建多个翻译实例，然后明确切换它们，如下所示：

```
import gettext

lang1 = gettext.translation('myapplication', languages=['en'])
lang2 = gettext.translation('myapplication', languages=['fr'])
lang3 = gettext.translation('myapplication', languages=['de'])

# start by using language1
lang1.install()

# ... time goes by, user selects language 2
lang2.install()

# ... more time goes by, user selects language 3
lang3.install()
```

23.1.3.4。延期翻译

在大多数编码情况下，字符串在被编码的地方被翻译。然而，有时候，您需要标记字符串进行翻译，但推迟实际翻译。一个典型的例子是：

```
animals = ['mollusk',
           'albatross',
           'rat',
           'penguin',
           'python', ]
# ...
for a in animals:
    print(a)
```

在这里，您想要将animals列表中的字符串标记为可翻译的，但您实际上并不想在打印之前翻译它们。

以下是您可以处理这种情况的一种方法：

```
def _(message): return message

animals = [_('mollusk'),
           _('albatross'),
           _('rat'),
           _('penguin'),
           _('python'), ]

del _

# ...
for a in animals:
    print(_(a))
```

这是可行的，因为虚拟定义_()只是简单地返回字符串。这个伪定义将暂时覆盖_()内置命名空间的任何定义（直到del命令）。小心，但如果你有一个_()在本地命名空间的先前定义。

请注意，第二次使用_()不会将“a”标识为可转换为**gettext**程序，因为该参数不是字符串文字。

下面的例子是处理这个问题的另一种方法：

```
def N_(message): return message

animals = [N_('mollusk'),
           N_('albatross'),
           N_('rat'),
           N_('penguin'),
           N_('python'), ]

# ...
for a in animals:
    print(_(a))
```

在这种情况下，您正在使用该函数标记可翻译的字符串，该函数 N_()不会与任何定义冲突_()。但是，您需要教您的消息提取程序以查找标有的可翻译字符串N_()。**xgettext**，**pygettext**，和**xpot**都通过使用命令行开关来支持这一点。这里的选择完全是任意的; 它可能一样容易。pybabel extract-kN_()MarkThisStringForTranslation()

23.1.4。致谢

以下人员为此模块的创建提供了代码，反馈，设计建议，以前的实施和宝贵的经验：

- 彼得芬克
- 詹姆斯亨斯特里奇
- Juan DavidIbáñezPalomar
- Marc-AndréLemburg
- Martin vonLöwis
- 弗朗索瓦·皮纳德
- 巴里华沙
- 古斯塔沃尼迈耶

脚注

[1] 缺省的语言环境目录是依赖于系统的; 例如，在RedHat Linux上/usr/share/locale，但是在Solaris上/usr/lib/locale。该gettext模块不会尝试支持这些系统相关的默认值; 相反，它的默认值是sys.prefix/share/locale。出于这个原因，最好bindtextdomain()在应用程序的开始时使用明确的绝对路径进行调用。

[2] 请参阅bindtextdomain()上面的脚注。

23.2。 locale- 国际化服务

源代码： Lib / locale.py

该locale模块打开对POSIX语言环境数据库和功能的访问。POSIX语言环境机制允许程序员处理应用程序中的某些文化问题，而不需要程序员知道执行该软件的每个国家的所有具体情况。

该locale模块在_locale模块的顶部实现，如果可用，该模块又使用ANSI C语言环境实现。

该locale模块定义了以下异常和功能：

*异常*locale. Error

当传递的区域设置setlocale()无法识别时引发异常。

locale. setlocale (*category* , *locale = None*)

如果*区域设置*是给定的而不是None，则setlocale()修改该*类别*的区域设置。可用的类别在下面的数据描述中列出。*语言环境*可以是字符串，也可以是两个字符串（语言代码和编码）的迭代。如果它是可迭代的，则使用区域别名引擎将其转换为区域设置名称。一个空字符串指定用户的默认设置。如果区域设置的修改失败，Error则会引发异常 。如果成功，则返回新的区域设置。

如果*区域设置*被省略None，则返回*类别*的当前设置。

setlocale()在大多数系统上不是线程安全的。应用程序通常以呼叫开始

```
import locale
locale.setlocale(locale.LC_ALL, '')
```

这将所有类别的语言环境设置为用户的默认设置（通常在。中指定） LANG环境变量）。如果此后区域设置未更改，则使用多线程不应导致问题。

locale. localeconv ( )

以字典形式返回本地约定的数据库。该字典具有以下字符串作为键：

类别	键	含义
LC_NUMERIC	'decimal_point'	小数点字符。
	'grouping'	数字序列指定了'thousands_sep'期望的相对位置 。如果序列被终止CHAR_MAX，则不执行进一步的分组。如果序列以a结尾，0则重复使用最后一个组的大小。
	'thousands_sep'	组间使用的字符。
LC_MONETARY	'int_curr_symbol'	国际货币符号。
	'currency_symbol'	当地的货币符号。

类别	键	含义
	'p_cs_precedes/n_cs_precedes'	货币符号是否在价值之前（对于正值和负值）。
	'p_sep_by_space/n_sep_by_space'	货币符号是否与空格分开（对于正值和负值）。
	'mon_decimal_point'	用于货币价值的小数点。
	'frac_digits'	货币值局部格式化中使用的小数位数。
	'int_frac_digits'	货币价值的国际格式化中使用的小数位数。
	'mon_thousands_sep'	用于货币值的组分隔符。
	'mon_grouping'	等同'grouping'于货币价值。
	'positive_sign'	用于注释正货币价值的符号。
	'negative_sign'	用于注释负面货币价值的符号。
	'p_sign_posn/n_sign_posn'	符号的位置（对于正值和负值），见下文。

可以将所有数值设置为CHAR_MAX指示在此语言环境中没有指定值。

对于可能的值'p_sign_posn'，并'n_sign_posn'在下面给出。

值	说明
0	货币和价值被圆括号包围。
1	该符号应该在值和货币符号之前。
2	标志应该遵循价值和货币符号。
3	该符号应该立即在值的前面。
4	该标志应该立即跟随该值。
CHAR_MAX	此区域设置中未指定任何内容。

如果函数的地址为非ASCII或长于1个字节，并且语言环境与语言环境不同，则该函数临时将LC_CTYPE语言环境LC_NUMERIC 设置为解码的字符decimal_point和thousands_sep字节字符串。此临时更改影响其他线程。 LC_NUMERICLC_CTYPE

*在版本3.6.5中进行了更改：*在某些情况下，此函数现在将LC_CTYPE语言环境临时设置为LC_NUMERIC区域设置。

locale.nl_langinfo（*可选*）

以字符串形式返回一些特定于区域的信息。此功能在所有系统上都不可用，并且可能的选项集也可能因平台而异。可能的参数值是数字，区域设置模块中的符号常量可用。

该nl_langinfo()功能接受下列其中一个键。大多数描述取自GNU C库中的相应描述。

locale.CODESET
获取具有所选语言环境中使用的字符编码名称的字符串。

locale.D_T_FMT
获取可用作格式字符串的字符串，以便time.strftime()以特定于语言环境的方式表示日期和时间。

locale.D_FMT
获取可用作格式字符串的字符串，以便time.strftime()以特定于语言环境的方式表示日期。

locale.T_FMT
获取可用作格式字符串的字符串，以便time.strftime()以特定于语言环境的方式表示时间。

locale.T_FMT_AMPM
获取格式字符串time.strftime()以表示am / pm格式的时间。

DAY_1 ... DAY_7
获取一周中第n天的名称。

> **注意:** 这遵循美国的DAY_1星期日惯例，而不是国际惯例（ISO 8601），星期一是本周的第一天。

ABDAY_1 ... ABDAY_7
获取一周中第n天的缩写名称。

MON_1 ... MON_12
获取第n个月的名称。

ABMON_1 ... ABMON_12
获取第n个月的缩写名称。

locale.RADIXCHAR
获取基数字符（小数点，小数点逗号等）。

locale.THOUSEP
获取数千个分隔符（三位数组）。

locale.YESEXPR
获取可用于正则表达式的正则表达式，以识别对是/否问题的肯定回答。

> **注意:** 该表达式的语法适合regex()C库中的函数，这可能与使用的语法不同re。

locale.NOEXPR
获取可用于正则表达式（3）的正则表达式来识别对“是/否”问题的否定响应。

locale.CRNCYSTR

获取货币符号，前面加上“-”（如果符号应该出现在值前面），“+”如果符号出现在值后面，或者“。”如果符号应该代替基数字符。

locale.ERA

获取一个表示当前语言环境中使用的时代的字符串。

大多数语言环境不定义此值。确定这个值的区域设置的一个例子是日本的。在日本，日期的传统表示包括与当时的皇帝统治相对应的时代的名称。

通常不需要直接使用该值。E在其格式字符串中指定修饰符会导致time.strftime()函数使用此信息。没有指定返回字符串的格式，因此您不应该在不同系统上假设它的知识。

locale.ERA_D_T_FMT

获取格式字符串，以便time.strftime()以特定于语言环境的基于时代的方式表示日期和时间。

locale.ERA_D_FMT

获取一个格式字符串，以便time.strftime()以特定于语言环境的基于时代的方式表示日期。

locale.ERA_T_FMT

获取一个格式字符串，以便time.strftime()以特定于语言环境的时代为基础表示时间。

locale.ALT_DIGITS

获取最多100个用于表示值0到99的值的表示。

locale.getdefaultlocale([*envvars*])

尝试确定默认的区域设置并将它们作为表单的元组返回。(language code, encoding)

根据POSIX，一个未 使用便携式语言环境调用运行的程序。调用让它使用默认的区域设置setlocale(LC_ALL, '')'C'setlocale(LC_ALL, '')LANG变量。由于我们不想干扰当前的区域设置，因此我们以上述方式模拟了行为。

为了保持与其他平台的兼容性，不仅是LANG变量被测试，但是作为envvars参数给出的变量列表。第一个被定义的将被使用。 *envvars*默认使用GNU gettext中使用的搜索路径; 它必须始终包含变量名称'LANG'。GNU的gettext的搜索路径包括'LC_ALL'，'LC_CTYPE'，'LANG'和'LANGUAGE'，在这个顺序。

除了代码'C'，语言代码对应于**RFC 1766**。 *语言代码*和*编码*可能是None如果他们的值不能确定。

locale.getlocale(*category = LC_CTYPE*)

将给定语言环境类别的当前设置作为包含*语言代码（编码）的*序列返回 。*类别*可能是其中的一个LC_*值LC_ALL。它默认为LC_CTYPE。

除了代码'C'，语言代码对应于**RFC 1766**。 *语言代码*和*编码*可能是None如果他们的值不能确定。

locale.**getpreferredencoding** (*do_setlocale = True*)

根据用户首选项返回用于文本数据的编码。用户首选项在不同的系统上表达方式不同，并且可能在某些系统上以编程方式不可用，所以此函数仅返回猜测。

在某些系统上，需要调用setlocale()以获取用户首选项，所以此函数不是线程安全的。如果调用setlocale不是必需或不需要的，应该将*do_setlocale*设置为False。

locale.**normalize** (*localename*)

返回给定语言环境名称的规范化语言环境代码。返回的区域设置代码被格式化以供使用setlocale()。如果规范化失败，则原始名称将保持不变。

如果给定的编码未知，则该函数默认为区域设置代码的默认编码setlocale()。

locale.**resetlocale** (*category = LC_ALL*)

将*类别*的区域设置设置为默认设置。

默认设置由调用确定getdefaultlocale()。 *类别*默认为LC_ALL。

locale.**strcoll** (*string1*，*string2*)

根据当前LC_COLLATE设置比较两个字符串。与其他任何比较函数一样，返回一个负值或一个正值，或者0根据*string1*在*string2*之前或之后进行整理还是等于它。

locale.**strxfrm** (*字符串*)

将字符串转换为可用于区分意识比较的字符串。例如，相当于。当重复比较相同的字符串时可以使用此函数，例如，在整理一系列字符串时。 strxfrm(s1) < strxfrm(s2) strcoll(s1, s2) < 0

locale.**format** (*format*，*val*，*grouping = False*，*monetary = False*)

格式化的数*VAL*根据当前LC_NUMERIC设置。格式遵循%运营商的惯例。对于浮点值，小数点在适当的情况下被修改。如果*分组*是真的，也将分组考虑在内。

如果*货币*为真，则转换使用货币千位分隔符和分组字符串。

请注意，该函数仅适用于一个％char说明符。对于整个格式字符串，请使用format_string()。

locale.**format_string** (*format*，*val*，*grouping = False*)

如过程中那样处理格式化说明符，但会考虑当前的语言环境设置。 format % val

locale.**currency** (*val*，*symbol = True*，*grouping = False*，*international = False*)

将数字格式*VAL*根据当前LC_MONETARY设置。

如果*符号*为true，则返回的字符串包含货币符号，这是默认值。如果*分组*为真（这不是默认值），则使用该值完成分组。如果*国际*是真的（这不是默认），则使用国际货币符号。

请注意，此功能不适用于'C'语言环境，因此您必须首先设置语言环境setlocale()。

locale.**str**(*float*)

使用与内置函数相同的格式格式化浮点数 str(float)，但考虑小数点。

locale.**delocalize**(*字符串*)

按照LC_NUMERIC设置将字符串转换为标准化数字字符串 。

3.5版本中的新功能。

locale.**atof**(*字符串*)

按照LC_NUMERIC 设置将字符串转换为浮点数。

locale.**atoi**(*字符串*)

按照LC_NUMERIC约定将字符串转换为整数。

locale.**LC_CTYPE**

用于字符类型函数的语言环境类别。根据这个类别的设置，模块string处理大小写的功能会改变它们的行为。

locale.**LC_COLLATE**

语言环境类别用于排序字符串。功能strcoll()和 strxfrm()对的locale模块受到影响。

locale.**LC_TIME**

区域设置类别用于格式化时间。该功能time.strftime() 遵循这些惯例。

locale.**LC_MONETARY**

用于格式化货币值的区域设置类别。可用的选项可从该localeconv()功能获得。

locale.**LC_MESSAGES**

用于消息显示的区域设置类别。Python目前不支持应用程序特定的区域识别消息。操作系统显示的消息（如返回的消息）os.strerror()可能会受此类别的影响。

locale.**LC_NUMERIC**

用于格式化数字的区域设置类别。功能format() ， atoi() ，atof()和str()的的locale模块是由类的影响。所有其他数字格式化操作不受影响。

locale.**LC_ALL**

所有区域设置的组合。如果在区域设置更改时使用此标志，则尝试设置所有类别的区域设置。如果任何类别都失败，则根本不会更改类别。使用此标志检索区域设置时，会返回指示所有类别设置的字符串。该字符串可以稍后用于恢复设置。

locale.**CHAR_MAX**

这是一个用于返回的不同值的符号常量 localeconv()。

例：

```
>>> import locale
>>> loc = locale.getlocale()  # get current locale
# use German locale; name might vary with platform
>>> locale.setlocale(locale.LC_ALL, 'de_DE')
```

```
>>> locale.strcoll('f\xe4n', 'foo')  # compare a string containing an umlaut
>>> locale.setlocale(locale.LC_ALL, '')    # use user's preferred locale
>>> locale.setlocale(locale.LC_ALL, 'C')   # use default (C) locale
>>> locale.setlocale(locale.LC_ALL, loc)   # restore saved locale
```

23.2.1。背景，细节，提示，提示和注意事项

C标准将语言环境定义为程序范围的属性，可能相对较昂贵地进行更改。最重要的是，一些实现被破坏，频繁的区域设置更改可能会导致核心转储。这使得语言环境使用起来有点痛苦。

最初，当程序启动时，语言环境是C语言环境，而不管用户的首选语言环境是什么。有一个例外： LC_CTYPE在启动时更改类别以将当前语言环境编码设置为用户的首选语言环境编码。该程序必须明确地说，它希望用户通过调用其他类别的首选区域设置。 setlocale(LC_ALL, '')

调用setlocale()一些库函数通常是一个坏主意，因为它会影响整个程序的副作用。保存和恢复它几乎同样糟糕：它是昂贵的并且会影响在设置恢复之前碰巧运行的其他线程。

如果在为一般用途编写模块时需要受区域设置影响的独立版本的操作（例如某些使用的格式time.strftime()），则必须找到一种不使用标准库例程的方法。更好的是让自己确信使用区域设置是可以的。只有作为最后的手段，你应该证明你的模块与非C本地设置不兼容。

只有这样，才能根据现场执行数字运算是使用由该模块定义的特殊功能：atof()，atoi()，format()，str()。

根据语言环境，无法执行大小写转换和字符分类。对于（Unicode）文本字符串，这些仅根据字符值完成，而对于字节字符串，转换和分类根据字节的ASCII值以及设置了高位的字节完成（即，非ASCII字节）永远不会被转换或被视为字母类的一部分，如字母或空格。

23.2.2。对于嵌入Python的扩展编写器和程序

扩展模块不应该调用setlocale()，除非找出当前语言环境。但是由于返回值只能用于便携式恢复它，这不是非常有用（除了可能找出区域是否是C）。

当Python代码使用locale模块更改语言环境时，这也会影响嵌入应用程序。如果嵌入式应用程序不希望发生这种情况，它应该_locale从config.c文件中的内置模块表中删除扩展模块（完成所有工作），并确保该_locale模块不能作为共享库访问。

23.2.3。访问消息目录

locale.gettext(*味精*)

locale.dgettext(*域*，*味精*)

locale.dcgettext(*域*，*味精*，*类别*)

locale.textdomain(*域*)

locale.bindtextdomain(*domain* , *dir*)

locale模块在提供此接口的系统上公开C库的gettext接口。它包括的功能gettext()，dgettext()，dcgettext()，textdomain()，bindtextdomain()，和bind_textdomain_codeset()。它们与gettext模块中的相同功能类似，但使用C库的二进制格式用于消息目录，以及C库的搜索算法用于查找消息目录。

Python应用程序通常不需要调用这些函数，而应该使用它gettext。一种已知的例外是用另外的C库在内部调用链接的应用程序 gettext()或dcgettext()。对于这些应用程序，可能需要绑定文本域，以便这些库可以正确定位其消息目录。

24.计划框架

本章描述的模块是基本上决定程序结构的框架。目前这里描述的模块都是面向编写命令行界面的。

本章描述的完整模块列表是：

24.1。 turtle- 龟图形

源代码： Lib / turtle.py

24.1.1。简介

龟图形是向孩子们介绍编程的一种流行方式。它是Wally Feurzig和Seymour Papert于1966年开发的原始Logo编程语言的一部分。

想象一下，机器人乌龟从xy平面（0,0）开始。之后，给它命令，并且它在它所面对的方向上移动（在屏幕上！）15个像素，当它移动时绘制一条线。给它的命令 ，它顺时针旋转25度。

```
import turtle turtle.forward(15) turtle.right(25)
```

通过将这些和类似命令组合在一起，可以轻松绘制复杂的形状和图片。

该turtle模块是从Python标准分发版到Python 2.5版的同名模块的扩展重新实现。

它试图保持旧乌龟模块的优点，并且（几乎）100％兼容它。这首先意味着学习程序员可以交互式地使用所有命令，类和方法-n。

乌龟模块以面向对象和面向过程的方式提供乌龟图形基元。由于它tkinter用于底层图形，因此需要安装Tk支持的Python版本。

面向对象的接口基本上使用两个+两个类：

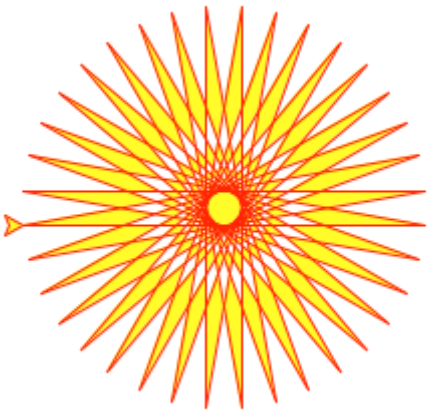
1. 在TurtleScreen类定义图形窗口作为用于绘制海龟游乐场。它的构造函数需要一个tkinter.Canvas或一个 ScrolledCanvas参数。它应该在用作turtle某些应用程序的一部分时使用。

   该函数Screen()返回一个TurtleScreen子类的单例对象 。当turtle作为独立的图形工具使用时，应该使用该函数 。作为一个单例对象，从它的类继承是不可能的。

   TurtleScreen / Screen的所有方法也作为函数存在，即作为面向过程的接口的一部分。

龟星

龟可以使用重复简单移动的程序绘制复杂的形状。



```
from turtle import *
color('red', 'yellow')
begin_fill()
while True:
    forward(200)
    left(170)
    if abs(pos()) < 1:
        break
end_fill()
done()
```

2. RawTurtle（别名：）RawPen定义了在a上绘制的Turtle对象TurtleScreen。它的构造函数需要Canvas，ScrolledCanvas或TurtleScreen作为参数，所以RawTurtle对象知道在哪里绘制。

派生自RawTurtle的是子类Turtle（别名：），该子类使用自动创建Pen的“ Screen实例”（如果尚未存在）。

RawTurtle / Turtle的所有方法也作为函数存在，即面向过程的接口的一部分。

过程接口提供了从类Screen和方法派生的函数Turtle。它们与相应的方法具有相同的名称。每当从Screen方法派生的函数被调用时，屏幕对象就会自动创建。无论何时调用Turtle方法派生的任何函数，都会自动创建一个（未命名的）乌龟对象。

要在屏幕上使用多个海龟，必须使用面向对象的界面。

注意: 在以下文档中给出了函数的参数列表。当然，方法有另外的第一个参数*self*，在这里省略。

24.1.2。可用的龟和屏幕方法概述

24.1.2.1。海龟方法

龟的议案

移动并绘制

forward() | fd()
backward() | bk() | back()
right() | rt()
left() | lt()
goto() | setpos() | setposition()
setx()
sety()
setheading() | seth()
home()
circle()
dot()
stamp()
clearstamp()
clearstamps()
undo()
speed()

告诉乌龟的状态

position() | pos()
towards()
xcor()
ycor()
heading()
distance()

设置和测量

degrees()
radians()

笔控制

绘图状态

pendown()| pd()|down()
penup()| pu()|up()
pensize() | width()
pen()
isdown()

颜色控制

color()
pencolor()
fillcolor()

填充

filling()
begin_fill()
end_fill()

更多的绘图控制

reset()
clear()
write()

海龟状态

能见度

showturtle() | st()
hideturtle() | ht()
isvisible()

出现

shape()
resizemode()
shapesize() | turtlesize()
shearfactor()
settiltangle()
tiltangle()
tilt()
shapetransform()
get_shapepoly()

使用事件

onclick()
onrelease()
ondrag()

特殊的乌龟方法

begin_poly()
end_poly()
get_poly()
clone()
getturtle() | getpen()
getscreen()
setundobuffer()

undobufferentries()

24.1.2.2。TurtleScreen / Screen的方法

窗口控制

bgcolor()
bgpic()
clear() | clearscreen()
reset() | resetscreen()
screensize()
setworldcoordinates()

动画控制

delay()
tracer()
update()

使用屏幕事件

listen()
onkey() | onkeyrelease()
onkeypress()
onclick() | onscreenclick()
ontimer()
mainloop() | done()

设置和特殊方法

mode()
colormode()
getcanvas()
getshapes()
register_shape() | addshape()
turtles()
window_height()
window_width()

输入法

textinput()
numinput()

特定于屏幕的方法

bye()
exitonclick()
setup()
title()

24.1.3。RawTurtle / Turtle和相应函数的方法

本节中的大部分示例都涉及一个叫做Turtle的实例 `turtle`。

24.1.3.1。海龟运动

turtle.**forward**(*距离*)
turtle.**fd**(*距离*)

参数：	**距离** - 一个数字（整数或浮点数）

将乌龟向前移动指定的*距离*，朝着乌龟朝前的方向移动。

```
>>> turtle.position()
(0.00,0.00)
>>> turtle.forward(25)
>>> turtle.position()
(25.00,0.00)
>>> turtle.forward(-75)
>>> turtle.position()
(-50.00,0.00)
```

turtle.**back**(*距离*)
turtle.**bk**(*距离*)
turtle.**backward**(*距离*)

参数：	**距离** - 一个数字

将乌龟向后移动*距离*，与乌龟朝向的方向相反。不要改变乌龟的标题。

```
>>> turtle.position()
(0.00,0.00)
>>> turtle.backward(30)
>>> turtle.position()
(-30.00,0.00)
```

turtle.**right**(*角度*)
turtle.**rt**(*角度*)

参数：	**角度** - 一个数字（整数或浮点数）

以*角度*单位向右转动龟。（单位默认为度数，但可以通过degrees()和radians()功能进行设置。）角度取决于乌龟模式，请参阅mode()。

```
>>> turtle.heading()
22.0
>>> turtle.right(45)
>>> turtle.heading()
337.0
```

turtle.**left**(*角度*)
turtle.**lt**(*角度*)

参数：	**角度** - 一个数字（整数或浮点数）

将*角度*单位转向左边的龟。（单位默认为度数，但可以通过degrees()和radians()功能进行设置。）角度取决于乌龟模式，请参阅mode()。

```
>>> turtle.heading()
22.0
>>> turtle.left(45)
>>> turtle.heading()
67.0
```

turtle.**goto**(*x*, *y=无*)
turtle.**setpos**(*x*, *y=无*)
turtle.**setposition**(*x*, *y=无*)

参数：	• **x** - 一个数字或一对数字 • **y** - 一个数字或None

如果*y*是None，*x*必须是一对坐标或一个Vec2D（例如返回的pos()）。

将龟移到绝对位置。如果笔落下，画线。不要改变乌龟的方向。

```
>>> tp = turtle.pos()
>>> tp
(0.00,0.00)
>>> turtle.setpos(60,30)
>>> turtle.pos()
(60.00,30.00)
>>> turtle.setpos((20,80))
>>> turtle.pos()
(20.00,80.00)
>>> turtle.setpos(tp)
>>> turtle.pos()
(0.00,0.00)
```

turtle.**setx**(*x*)

参数：	**x** - 一个数字（整数或浮点数）

将乌龟的第一个坐标设置为*x*，保持第二个坐标不变。

```
>>> turtle.position()
(0.00,240.00)
>>> turtle.setx(10)
>>> turtle.position()
(10.00,240.00)
```

turtle.**sety**(*y*)

参数：	**y** - 一个数字（整数或浮点数）

将龟的第二个坐标设置为*y*，保持第一个坐标不变。

```
>>> turtle.position()
(0.00,40.00)
```

```
>>> turtle.sety(-10)
>>> turtle.position()
(0.00,-10.00)
```

turtle.**setheading**(*to_angle*)
turtle.**seth**(*to_angle*)

参数：	**to_angle** - 数字（整数或浮点数）

将乌龟的方向设置为*to_angle*。以下是一些常用的度数方向：

标准模式	标志模式
0 - 东	0 - 北
90 - 北	90 - 东
180 - 西	180 - 南
270 - 南	270 - 西

```
>>> turtle.setheading(90)
>>> turtle.heading()
90.0
```

turtle.**home**( )

将乌龟移动到原点 - 坐标（0,0） - 并将其标题设置为其起始方向（取决于模式，请参阅参考资料mode() ）。

```
>>> turtle.heading()
90.0
>>> turtle.position()
(0.00,-10.00)
>>> turtle.home()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
```

turtle.**circle**(*radius*，*extent* = *None*，*steps* = *None*)

参数：	• **半径** - 一个数字 • **程度** - 一个数字（或None） • **步骤** - 整数（或None）

绘制一个给定*半径*的圆。该中心是龟的左*半径*单位; *范围* - 一个角度 - 确定绘制圆的哪一部分。如果*范围*没有给出，绘制整个圆。如果*范围*不是整圆，则圆弧的一个端点是当前笔的位置。如果*半径*为正，则逆时针绘制弧线，否则沿顺时针方向绘制弧线。最后，龟的方向改变了*程度*。

由于圆是用刻有正多边形的近似值，所以*步*数决定了要使用的步数。如果没有给出，它会自动计算。可用于绘制正多边形。

```
>>> turtle.home()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
>>> turtle.circle(50)
>>> turtle.position()
(-0.00,0.00)
>>> turtle.heading()
0.0
>>> turtle.circle(120, 180)  # draw a semicircle
>>> turtle.position()
(0.00,240.00)
>>> turtle.heading()
180.0
```

turtle.**dot**(*size = None*, **color*)

参数：	• **大小** - 一个整数> = 1（如果给出） • **颜色** - 颜色字符串或数字颜色元组

用*颜色*画出一个直径*大小的*圆点。如果没有给出*尺寸*，则使用pensize + 4和2 * pensize的最大值。

```
>>> turtle.home()
>>> turtle.dot()
>>> turtle.fd(50); turtle.dot(20, "blue"); turtle.fd(50)
>>> turtle.position()
(100.00,-0.00)
>>> turtle.heading()
0.0
```

turtle.**stamp**( )

在当前乌龟位置上将海龟形状的副本印到画布上。返回该邮票的stamp_id，可以通过调用将其删除clearstamp(stamp_id)。

```
>>> turtle.color("blue")
>>> turtle.stamp()
11
>>> turtle.fd(50)
```

turtle.**clearstamp**(*stampid*)

参数：	**stampid** - 一个整数，必须是先前stamp()调用的返回值

用给定的*stampid*删除邮票。

```
>>> turtle.position()
(150.00,-0.00)
>>> turtle.color("blue")
>>> astamp = turtle.stamp()
>>> turtle.fd(50)
>>> turtle.position()
(200.00,-0.00)
```

```
>>> turtle.clearstamp(astamp)
>>> turtle.position()
(200.00,-0.00)
```

turtle.**clearstamps**(*n=无*)

参数：	**n** - 整数（或None）

删除全部或姓氏/ *ñ*龟的邮票。如果*n*是None，删除所有的邮票，如果*n* > 0删除前*n*个邮票，否则如果*n* <0删除最后*n*个邮票。

```
>>> for i in range(8):
...     turtle.stamp(); turtle.fd(30)
13
14
15
16
17
18
19
20
>>> turtle.clearstamps(2)
>>> turtle.clearstamps(-2)
>>> turtle.clearstamps()
```

turtle.**undo**()

撤消（重复）最后的乌龟动作。可用撤销操作的数量由不可扩展器的大小决定。

```
>>> for i in range(4):
...     turtle.fd(50); turtle.lt(80)
...
>>> for i in range(8):
...     turtle.undo()
```

turtle.**speed**(*速度=无*)

参数：	**速度** - 范围为0..10的整数或**速度**串（见下文）

将乌龟的速度设置为0..10范围内的整数值。如果没有参数，则返回当前速度。

如果输入是大于10或小于0.5的数字，则速度设置为0.将速度串映射为speedvalues，如下所示：

- “最快”：0
- “快”：10
- “正常”：6
- “慢”：3
- “最慢”：1

从1到10的速度强制执行越来越快的画线和龟转动的动画。

注意：*速度* = 0意味着*没有*动画发生。向前/向后使乌龟跳跃，同样向左/向右使乌龟立即转动。

```
>>> turtle.speed()
3
>>> turtle.speed('normal')
>>> turtle.speed()
6
>>> turtle.speed(9)
>>> turtle.speed()
9
```

24.1.3.2。告诉乌龟的状态

turtle.**position**( )

turtle.**pos**( )

返回龟的当前位置（x，y）（作为Vec2D矢量）。

```
>>> turtle.pos()
(440.00,-0.00)
```

turtle.**towards**(*x*，*y*=*无*)

参数：	• **x** - 一个数字或一对数字或一个乌龟实例的向量 • **y** - 如果*x*是数字，则为数字，否则None

返回从乌龟位置到由（x，y），矢量或其他乌龟指定的位置之间的角度。这取决于乌龟的开始取向，取决于模式 - “标准”/“世界”或“标志”）。

```
>>> turtle.goto(10, 10)
>>> turtle.towards(0,0)
225.0
```

turtle.**xcor**( )

返回乌龟的x坐标。

```
>>> turtle.home()
>>> turtle.left(50)
>>> turtle.forward(100)
>>> turtle.pos()
(64.28,76.60)
>>> print(round(turtle.xcor(), 5))
64.27876
```

turtle.**ycor**( )

返回乌龟的y坐标。

```
>>> turtle.home()
>>> turtle.left(60)
>>> turtle.forward(100)
>>> print(turtle.pos())
(50.00,86.60)
>>> print(round(turtle.ycor(), 5))
86.60254
```

turtle.**heading**( )

返回乌龟的当前标题（值取决于乌龟模式，请参阅 `mode()`）。

```
>>> turtle.home()
>>> turtle.left(67)
>>> turtle.heading()
67.0
```

turtle.**distance**(*x*，*y*=*无*)

参数：	• **x** - 一个数字或一对数字或一个乌龟实例的向量 • **y** - 如果*x*是数字，则为数字，否则None

以龟步单位返回从乌龟到（x，y），给定向量或给定的其他乌龟的距离。

```
>>> turtle.home()
>>> turtle.distance(30,40)
50.0
>>> turtle.distance((30,40))
50.0
>>> joe = Turtle()
>>> joe.forward(77)
>>> turtle.distance(joe)
77.0
```

24.1.3.3。测量设置

turtle.**degrees**(*fullcircle = 360.0*)

参数：	**全圆** - 一个数字

设置角度测量单位，即设置一整圈的“度数”。默认值是360度。

```
>>> turtle.home()
>>> turtle.left(90)
>>> turtle.heading()
90.0

Change angle measurement unit to grad (also known as gon,
grade, or gradian and equals 1/100-th of the right angle.)
>>> turtle.degrees(400.0)
>>> turtle.heading()
100.0
>>> turtle.degrees(360)
>>> turtle.heading()
90.0
```

turtle.**radians**( )

将角度测量单位设置为弧度。相当于 `degrees(2*math.pi)`。

```
>>> turtle.home()
>>> turtle.left(90)
>>> turtle.heading()
90.0
>>> turtle.radians()
>>> turtle.heading()
1.5707963267948966
```

24.1.3.4。笔控制

24.1.3.4.1。绘图状态

turtle.**pendown**( )
turtle.**pd**( )
turtle.**down**( )
拉下笔 - 移动时绘图。

turtle.**penup**( )
turtle.**pu**( )
turtle.**up**( )
拉起笔 - 移动时无图。

turtle.**pensize**(*width = None*)
turtle.**width**(*width = None*)

参数：	**宽度** - 一个正数

将线条粗细设置为*宽度*或将其返回。如果resizemode被设置为“auto”并且turtleshape是一个多边形，那么该多边形将以相同的线条粗细绘制。如果没有给出参数，则返回当前的pensize。

```
>>> turtle.pensize()
1
>>> turtle.pensize(10)   # from here on lines of width 10 are drawn
```

turtle.**pen**(*pen = None*，*** pendict*)

参数：	• **笔** - 包含部分或全部下列键的字典 • **pendict** - 一个或多个关键字参数，下面列出的关键字为关键字

使用以下键/值对在“笔字典”中返回或设置笔的属性：

- “显示”：真/假
- “pendown”：真/假
- “pencolor”：颜色字符串或颜色元组
- “fillcolor”：颜色字符串或颜色元组
- “pensize”：正数
- “速度”：范围为0..10的数字
- “resizemode”：“auto”或“user”或“noresize”
- “stretchfactor”:(正数，正数)

- “大纲”：正数
- “倾斜”：数字

该字典可以用作后续调用pen()以恢复前一笔状态的参数。此外，这些属性中的一个或多个可以作为关键字参数提供。这可以用于在一个语句中设置多个笔属性。

```
>>> turtle.pen(fillcolor="black", pencolor="red", pensize=10)
>>> sorted(turtle.pen().items())
[('fillcolor', 'black'), ('outline', 1), ('pencolor', 'red'),
 ('pendown', True), ('pensize', 10), ('resizemode', 'noresize'),
 ('shearfactor', 0.0), ('shown', True), ('speed', 9),
 ('stretchfactor', (1.0, 1.0)), ('tilt', 0.0)]
>>> penstate=turtle.pen()
>>> turtle.color("yellow", "")
>>> turtle.penup()
>>> sorted(turtle.pen().items())[:3]
[('fillcolor', ''), ('outline', 1), ('pencolor', 'yellow')]
>>> turtle.pen(penstate, fillcolor="green")
>>> sorted(turtle.pen().items())[:3]
[('fillcolor', 'green'), ('outline', 1), ('pencolor', 'red')]
```

turtle.**isdown**( )

True如果笔停止，False则返回。

```
>>> turtle.penup()
>>> turtle.isdown()
False
>>> turtle.pendown()
>>> turtle.isdown()
True
```

24.1.3.4.2。颜色控制

turtle.**pencolor**(**args*)

返回或设置pencolor。

允许四种输入格式：

pencolor()

返回当前的pencolor作为颜色指定字符串或作为元组（参见示例）。可用作另一种颜色/ pencolor / fillcolor调用的输入。

pencolor(colorstring)

设置pencolor到 *colorstring* ，这是一个Tk的颜色指定的字符串，例如"red"，"yellow"，或"#33cc8c"。

pencolor((r, g, b))

将pencolor设置为由*r*，*g*和 *b*的元组表示的RGB颜色。*r*，*g*和*b*中的每一个都必须在0..colormode范围内，其中colormode是1.0或255（请参阅参考资料colormode()）。

pencolor(r, g, b)

将pencolor设置为由*r*，*g*和*b*表示的RGB颜色。*r*，*g*和*b*中的每一个都必须在0 ... colormode范围内。

如果turtleshape是多边形，则使用新设置的pencolor绘制该多边形的轮廓。

```
>>> colormode()
1.0
>>> turtle.pencolor()
'red'
>>> turtle.pencolor("brown")
>>> turtle.pencolor()
'brown'
>>> tup = (0.2, 0.8, 0.55)
>>> turtle.pencolor(tup)
>>> turtle.pencolor()
(0.2, 0.8, 0.5490196078431373)
>>> colormode(255)
>>> turtle.pencolor()
(51.0, 204.0, 140.0)
>>> turtle.pencolor('#32c18f')
>>> turtle.pencolor()
(50.0, 193.0, 143.0)
```

turtle.**fillcolor**(**args*)

返回或设置fillcolor。

允许四种输入格式：

`fillcolor()`

可能以元组格式返回当前的fillcolor作为颜色指定字符串（请参阅示例）。可用作另一种颜色/ pencolor / fillcolor调用的输入。

`fillcolor(colorstring)`

设置填充颜色到 *colorstring* ，这是一个Tk的颜色指定的字符串，例如"red"，"yellow"，或"#33cc8c"。

`fillcolor((r, g, b))`

将fillcolor设置为由*r*，*g*和 *b*的元组表示的RGB颜色。*r*，*g*和*b*中的每一个都必须在0..colormode范围内，其中colormode是1.0或255（请参阅参考资料`colormode()`）。

`fillcolor(r, g, b)`

将fillcolor设置为由*r*，*g*和*b*表示的RGB颜色。*r*，*g*和*b*中的每一个都 必须在0 ... colormode范围内。

如果turtleshape是多边形，则使用新设置的fillcolor绘制该多边形的内部。

```
>>> turtle.fillcolor("violet")
>>> turtle.fillcolor()
'violet'
>>> col = turtle.pencolor()
>>> col
(50.0, 193.0, 143.0)
>>> turtle.fillcolor(col)
>>> turtle.fillcolor()
(50.0, 193.0, 143.0)
>>> turtle.fillcolor('#ffffff')
>>> turtle.fillcolor()
(255.0, 255.0, 255.0)
```

turtle.**color**(**args*)

返回或设置pencolor和fillcolor。

允许多种输入格式。他们使用0到3个参数如下：

color()

返回当前pencolor和当前填充颜色为一对颜色指定字符串或元组通过与返回的pencolor()和 fillcolor()。

color(colorstring)，color((r,g,b))，color(r,g,b)

输入如in pencolor()，将fillcolor和pencolor设置为给定值。

color(colorstring1, colorstring2)，color((r1,g1,b1), (r2,g2,b2))

如果使用其他输入格式，则等效于pencolor(colorstring1)并且fillcolor(colorstring2)类似。

如果turtleshape是多边形，则使用新设置的颜色绘制该多边形的轮廓和内部。

```
>>> turtle.color("red", "green")
>>> turtle.color()
('red', 'green')
>>> color("#285078", "#a0c8f0")
>>> color()
((40.0, 80.0, 120.0), (160.0, 200.0, 240.0))
```

另请参阅：屏幕方法colormode()。

24.1.3.4.3。填充

turtle.**filling**()

返回fillstate（True如果填充，False否则）。

```
>>> turtle.begin_fill()
>>> if turtle.filling():
...     turtle.pensize(5)
... else:
...     turtle.pensize(3)
```

turtle.**begin_fill**()

在绘制要填充的形状之前调用。

turtle.**end_fill**()

填写最后一次调用之后绘制的形状begin_fill()。

```
>>> turtle.color("black", "red")
>>> turtle.begin_fill()
>>> turtle.circle(80)
>>> turtle.end_fill()
```

24.1.3.4.4。更多的绘图控制

turtle.**reset**( )

从屏幕上删除乌龟的图纸，重新将乌龟居中并将变量设置为默认值。

```
>>> turtle.goto(0,-22)
>>> turtle.left(100)
>>> turtle.position()
(0.00,-22.00)
>>> turtle.heading()
100.0
>>> turtle.reset()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
```

turtle.**clear**( )

从屏幕上删除乌龟的图纸。不要移动乌龟。乌龟的状态和位置以及其他乌龟的图纸不受影响。

turtle.**write**(*arg*，*move = False*，*align =“left”*，*font =（“Arial”，8，“normal”）*)

参数：	• **arg** - 要写入TurtleScreen的对象 • **移动** - 真/假 • **对齐** - 其中一个字符串“左”，“中”或右“ • **font** - 一个三元组（fontname，fontsize，fonttype）

写文本 - *arg*的字符串表示- 根据*align*（“left”，“center”或right“）和给定字体在当前海龟位置写入。如果*移动*为真，笔将移动到文本的右下角。默认情况下，*移动*是False。

```
>>> turtle.write("Home = ", True, align="center")
>>> turtle.write((0,0), True)
```

24.1.3.5。龟状态

24.1.3.5.1。可见性

turtle.**hideturtle**( )

turtle.**ht**( )

使乌龟无形。在做复杂绘图的过程中这样做是个好主意，因为隐藏龟可以显着提高绘图速度。

```
>>> turtle.hideturtle()
```

turtle.**showturtle**( )

turtle.**st**( )

使乌龟可见。

```
>>> turtle.showturtle()
```

turtle.**isvisible**( )

True如果乌龟显示，False如果隐藏，则返回。

```
>>> turtle.hideturtle()
>>> turtle.isvisible()
False
>>> turtle.showturtle()
>>> turtle.isvisible()
True
```

24.1.3.5.2。外观

turtle.**shape**(*name* = *None*)

参数：	**名称** - 一个有效形状**名称**的字符串

将龟形状设置为具有给定*名称的*形状，或者如果没有给出名称，则返回当前形状的名称。带有*名称*的形状必须存在于TurtleScreen的形状字典中。最初有以下多边形形状：“箭头”，“龟”，“圆”，“方形”，“三角形”，“经典”。要了解如何处理形状，请参阅屏幕方法register_shape()。

```
>>> turtle.shape()
'classic'
>>> turtle.shape("turtle")
>>> turtle.shape()
'turtle'
```

turtle.**resizemode**(*rmode* = *None*)

参数：	**rmode** - 其中一个字符串“auto”，“user”，“noresize”

将resizemode设置为其中一个值：“auto”，“user”，“noresize”。如果*RMODE* 没有给出，返回当前resizemode。不同的调整大小模式有以下效果：

- “auto”：根据pensize的值调整乌龟的外观。
- “用户”：根据由设置的stretchfactor和outlinewidth（outline）的值来调整龟的外观shapesize()。
- “noresize”：没有适应龟的外观发生。

resizemode（“user”）shapesize()在与参数一起使用时被调用。

```
>>> turtle.resizemode()
'noresize'
>>> turtle.resizemode("auto")
>>> turtle.resizemode()
'auto'
```

turtle.**shapesize**(*stretch_wid* = *None*，*stretch_len* = *None*，*outline* = *None*)
turtle.**turtlesize**(*stretch_wid* = *None*，*stretch_len* = *None*，*outline* = *None*)

参数：	• **stretch_wid** - 正数 • **stretch_len** - 正数

	• **大纲** - 正数

返回或设置笔的属性x / y-stretchfactors和/或大纲。将调整大小模式设置为“用户”。当且仅当resizemode设置为“user”时，乌龟将根据其拉伸因子展开显示：*stretch_wid*拉伸因子垂直于其方向，*stretch_len*拉伸因子在其方向方向上，*轮廓线*决定了形状轮廓的宽度。

```
>>> turtle.shapesize()
(1.0, 1.0, 1)
>>> turtle.resizemode("user")
>>> turtle.shapesize(5, 5, 12)
>>> turtle.shapesize()
(5, 5, 12)
>>> turtle.shapesize(outline=8)
>>> turtle.shapesize()
(5, 5, 8)
```

turtle.**shearfactor**(*剪切=无*)

参数：	**剪切**数（可选）

设置或返回当前的剪切因子。根据给定的剪切因子剪切剪切龟形，这是剪切角的切线。不要*不*改变乌龟的标题（移动方向）。如果没有给出剪切：返回当前的剪切因子，即剪切角的切线，平行于龟头方向的线被剪切。

```
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.shearfactor(0.5)
>>> turtle.shearfactor()
0.5
```

turtle.**tilt**(*角度*)

参数：	**角度** - 一个数字

旋转由turtleshape *角度*从当前的倾斜角度，但*不*改变乌龟的标题（移动方向）。

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.tilt(30)
>>> turtle.fd(50)
>>> turtle.tilt(30)
>>> turtle.fd(50)
```

turtle.**settiltangle**(*角度*)

参数：	**角度** - 一个数字

无论当前的倾斜角度如何，旋转龟形指向*角度*指定的方向。 *不要*改变乌龟的朝向（移动方向）。

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
```

```
>>> turtle.settiltangle(45)
>>> turtle.fd(50)
>>> turtle.settiltangle(-45)
>>> turtle.fd(50)
```

自3.1版以来已弃用。

turtle.**tiltangle**(*angle=None*)

参数：	**角度** - 数字（可选）

设置或返回当前的倾斜角度。如果给出了角度，则不管当前的倾斜角度如何，都将乌龟形状旋转指向角度指定的方向。不要*不*改变乌龟的标题（移动方向）。如果没有给出角度：返回当前的倾斜角度，即海龟的方向与海龟的方向（其运动方向）之间的角度。

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.tilt(45)
>>> turtle.tiltangle()
45.0
```

turtle.**shapetransform**(*t11=无*, *t12=无*, *t21=无*, *t22=无*)

参数：	• **t11** - 一个数字（可选） • **t12** - 一个数字（可选） • **t21** - 一个数字（可选） • **t12** - 一个数字（可选）

设置或返回乌龟形状的当前转换矩阵。

如果没有给出矩阵元素，则将变换矩阵作为4元素的元组返回。否则，设置给定元素并根据由第一行t11，t12和第二行t21,22组成的矩阵变换龟形。行列式t11 * t22-t12 * t21不能为零，否则会引发错误。根据给定的矩阵修改stretchfactor，shearfactor和tiltangle。

```
>>> turtle = Turtle()
>>> turtle.shape("square")
>>> turtle.shapesize(4,2)
>>> turtle.shearfactor(-0.5)
>>> turtle.shapetransform()
(4.0, -1.0, -0.0, 2.0)
```

turtle.**get_shapepoly**( )

将当前形状多边形返回为坐标对的元组。这可以用来定义一个新形状或复合形状的组件。

```
>>> turtle.shape("square")
>>> turtle.shapetransform(4, -1, 0, 2)
>>> turtle.get_shapepoly()
((50, -20), (30, 20), (-50, 20), (-30, -20))
```

24.1.3.6。使用事件

turtle.**onclick**(*fun*, *btn = 1*, *add = None*)

参数：	• **fun** - 一个带有两个参数的函数，这些参数将与画布上单击点的坐标一起调用 • **num** - 鼠标按钮的数量，默认为1（鼠标左键） • **添加** - True或False- 如果True，将添加新的绑定，否则它将替换以前的绑定

绑定*的乐趣*就这个龟鼠标点击事件。如果*好玩的*是None，现有绑定被删除。匿名乌龟的例子，即程序方式：

```
>>> def turn(x, y):
...     left(180)
...
>>> onclick(turn)  # Now clicking into the turtle will turn it.
>>> onclick(None)  # event-binding will be removed
```

turtle.**onrelease**(*fun*, *btn = 1*, *add = None*)

参数：	• **fun** - 一个带有两个参数的函数，这些参数将与画布上单击点的坐标一起调用 • **num** - 鼠标按钮的数量，默认为1（鼠标左键） • **添加** - True或False- 如果True，将添加新的绑定，否则它将替换以前的绑定

绑定*的乐趣*就这个龟鼠标按钮释放事件。如果*好玩的*是 None，现有绑定被删除。

```
>>> class MyTurtle(Turtle):
...     def glow(self,x,y):
...         self.fillcolor("red")
...     def unglow(self,x,y):
...         self.fillcolor("")
...
>>> turtle = MyTurtle()
>>> turtle.onclick(turtle.glow)     # clicking on turtle turns fillcolor red,
>>> turtle.onrelease(turtle.unglow) # releasing turns it to transparent.
```

turtle.**ondrag**(*fun*, *btn = 1*, *add = None*)

参数：	• **fun** - 一个带有两个参数的函数，这些参数将与画布上单击点的坐标一起调用 • **num** - 鼠标按钮的数量，默认为1（鼠标左键） • **添加** - True或False- 如果True，将添加新的绑定，否则它将替换以前的绑定

绑定*的乐趣*就这个龟鼠标移动事件。如果*好玩的*是None，现有绑定被删除。

备注：在乌龟上的每一个鼠标移动事件序列都在该乌龟的鼠标点击事件之前。

```
>>> turtle.ondrag(turtle.goto)
```

随后，点击并拖动乌龟将在屏幕上移动，从而生成手绘图（如果笔落下）。

24.1.3.7。特殊海龟方法

turtle.**begin_poly**( )

开始记录多边形的顶点。当前的乌龟位置是多边形的第一个顶点。

turtle.**end_poly**( )

停止记录多边形的顶点。当前乌龟位置是多边形的最后一个顶点。这将与第一个顶点连接。

turtle.**get_poly**( )

返回最后记录的多边形。

```
>>> turtle.home()
>>> turtle.begin_poly()
>>> turtle.fd(100)
>>> turtle.left(20)
>>> turtle.fd(30)
>>> turtle.left(60)
>>> turtle.fd(50)
>>> turtle.end_poly()
>>> p = turtle.get_poly()
>>> register_shape("myFavouriteShape", p)
```

turtle.**clone**( )

创建并返回具有相同位置，标题和龟属性的龟的克隆。

```
>>> mick = Turtle()
>>> joe = mick.clone()
```

turtle.**getturtle**( )

turtle.**getpen**( )

返回Turtle对象本身。只有合理的使用：作为返回“匿名乌龟”的函数：

```
>>> pet = getturtle()
>>> pet.fd(50)
>>> pet
<turtle.Turtle object at 0x...>
```

turtle.**getscreen**( )

返回TurtleScreen乌龟正在绘制的对象。然后可以为该对象调用TurtleScreen方法。

```
>>> ts = turtle.getscreen()
>>> ts
<turtle._Screen object at 0x...>
>>> ts.bgcolor("pink")
```

turtle.**setundobuffer**(*size*)

参数：	**大小** - 一个整数或None

设置或禁用不中断器。如果*size*是一个整数，则会安装给定大小的空的undobuffer。 *大小*给出了可以通过undo()方法/函数撤消的最大数量的龟行动。如果*大小*是 None ，则禁用不可用的缓冲区。

```
>>> turtle.setundobuffer(42)
```

turtle.**undobufferentries**( )

返回undobuffer中的条目数。

```
>>> while undobufferentries():
...     undo()
```

24.1.3.8。复合形状

要使用由多个不同颜色的多边形组成的复合乌龟形状，您必须Shape明确地使用助手类，如下所述：

1. 创建一个类型为“compound”的空Shape对象。
2. 使用该addcomponent()方法，根据需要向该对象添加尽可能多的组件 。

   例如：

```
>>> s = Shape("compound")
>>> poly1 = ((0,0),(10,-5),(0,10),(-10,-5))
>>> s.addcomponent(poly1, "red", "blue")
>>> poly2 = ((0,0),(10,-5),(-10,-5))
>>> s.addcomponent(poly2, "blue", "red")
```

3. 现在将Shape添加到屏幕的shapelist中并使用它：

```
>>> register_shape("myshape", s)
>>> shape("myshape")
```

注意： 的Shape类是由内部使用register_shape()以不同的方式方法。*只有*在使用上图所示的复合形状时，应用程序员才能处理Shape类！

24.1.4。TurtleScreen /屏幕及相应功能的方法

本节中的大多数示例都涉及一个名为的TurtleScreen实例 screen。

24.1.4.1。窗口控制

turtle.**bgcolor**(**args*)

参数：	**参数** - 一个颜色字符串或范围为0..colormode的三个数字或这种数字的三元组

设置或返回TurtleScreen的背景颜色。

```
>>> screen.bgcolor("orange")
>>> screen.bgcolor()
'orange'
>>> screen.bgcolor("#800080")
>>> screen.bgcolor()
(128.0, 0.0, 128.0)
```

turtle.**bgpic**(*picname = None*)

参数：	**picname** - 一个字符串，一个gif文件的名字"nopic"，或者None

设置当前backgroundimage的背景图片或返回名称。如果*picname* 是文件名，请将相应的图像设置为背景。如果*图片名称*是 "nopic"，删除背景图片，如果存在。如果*picname*是None，则返回当前backgroundimage的文件名。

```
>>> screen.bgpic()
'nopic'
>>> screen.bgpic("landscape.gif")
>>> screen.bgpic()
"landscape.gif"
```

turtle.**clear**( )

turtle.**clearscreen**( )

从TurtleScreen中删除所有图纸和所有海龟。将现在空的TurtleScreen重置为初始状态：白色背景，无背景图像，无事件绑定和追踪。

> **注意：** 此TurtleScreen方法仅在名称下作为全局函数提供clearscreen。全局函数clear是从Turtle方法派生出来的一个不同的函数clear。

turtle.**reset**( )

turtle.**resetscreen**( )

将屏幕上的所有海龟重置为其初始状态。

> **注意：** 此TurtleScreen方法仅在名称下作为全局函数提供resetscreen。全局函数reset是从乌龟方法派生的另一个函数reset。

turtle.**screensize**(*canvwidth = None*，*canvheight = None*，*bg = None*)

参数：	• **canvwidth** - 正整数，画布的新宽度（以像素为单位） • **canvheight** - 正整数，以像素为单位的画布新高度 • **bg** - colorstring或color-tuple，新的背景颜色

如果没有给出参数，则返回当前值（画布宽度，画布高度）。否则调整海龟画的画布大小。不要改变绘图窗口。要观察画布的隐藏部分，请使用滚动条。使用这种方法，可以使之前画布外部的绘图部分可见。

```
>>> screen.screensize()
(400, 300)
>>> screen.screensize(2000,1500)
>>> screen.screensize()
(2000, 1500)
```

例如寻找一只错误逃脱的乌龟;-)

turtle.**setworldcoordinates**(*LLX*，*LLY*，*URX*，*URY*)

参数：	• **llx** - 一个数字，画布左下角的x坐标 • **lly** - 画布左下角的数字，y坐标 • **urx** - 一个数字，画布右上角的x坐标 • **ury** - 画布右上角的数字，y坐标

设置用户定义的坐标系并在必要时切换到“世界”模式。这执行一个`screen.reset()`。如果模式“世界”已经激活，则根据新坐标重绘所有图纸。

注意：在用户定义的坐标系中，角度可能会出现扭曲。

```
>>> screen.reset()
>>> screen.setworldcoordinates(-50,-7.5,50,7.5)
>>> for _ in range(72):
...     left(10)
...
>>> for _ in range(8):
...     left(45); fd(2)   # a regular octagon
```

24.1.4.2。动画控制

turtle.**delay**(*延迟=无*)

参数：	**延迟** - 正整数

设置或返回以毫秒为单位的绘图*延迟*。（这大致是两次连续画布更新之间的时间间隔。）绘图延迟越长，动画越慢。

可选参数：

```
>>> screen.delay()
10
>>> screen.delay(5)
>>> screen.delay()
5
```

turtle.**tracer**(*n =无*, *延迟=无*)

参数：	• **n** - 非负整数 • **延迟** - 非负整数

打开/关闭乌龟动画并为更新图纸设置延迟。如果 给出*n*，则仅实际执行每个第n个常规屏幕更新。（可用于加速绘制复杂图形。）当不带参数调用时，返回当前存储的n值。第二个参数设置延迟值（请参阅 `delay()` ）。

```
>>> screen.tracer(8, 25)
>>> dist = 2
>>> for i in range(200):
...     fd(dist)
...     rt(90)
...     dist += 2
```

turtle.**update**( )

执行TurtleScreen更新。在示踪剂关闭时使用。

另见RawTurtle / Turtle方法speed()。

24.1.4.3。使用屏幕事件

turtle.**listen**(*xdummy = None*, *ydummy = None*)

将注意力集中在TurtleScreen（为了收集关键事件）。提供虚拟参数是为了能够传递listen()给onclick方法。

turtle.**onkey**(*fun*, *key*)
turtle.**onkeyrelease**(*fun*, *key*)

参数：	• **有趣** - 一个没有参数或功能的函数None • **键** - 字符串：键（例如“a”）或键符号（例如“空格”）

绑定*开心*到键表面释放事件。如果*好玩的*是None，事件绑定被删除。备注：为了能够注册关键事件，TurtleScreen必须有重点。（见方法listen()）

```
>>> def f():
...     fd(50)
...     lt(60)
...
>>> screen.onkey(f, "Up")
>>> screen.listen()
```

turtle.**onkeypress**(*fun*, *key = None*)

参数：	• **有趣** - 一个没有参数或功能的函数None • **键** - 字符串：键（例如“a”）或键符号（例如“空格”）

如果给出了密钥，则为按键按键事件绑定*乐趣*;如果没有给出密钥，则按键按下事件。备注：为了能够注册关键事件，TurtleScreen必须具有焦点。（见方法listen()）

```
>>> def f():
...     fd(50)
...
>>> screen.onkey(f, "Up")
>>> screen.listen()
```

turtle.**onclick**(*fun*, *btn = 1*, *add = None*)
turtle.**onscreenclick**(*fun*, *btn = 1*, *add = None*)

参数：	• **fun** - 一个带有两个参数的函数，这些参数将与画布上单击点的坐标一起调用 • **num** - 鼠标按钮的数量，默认为1（鼠标左键） • **添加** - True或False- 如果True，将添加新的绑定，否则它将替换以前的绑定

绑定*的乐趣*在此屏幕上的鼠标点击事件。如果*好玩的*是None，现有绑定被删除。

示例名为turtleScreen的实例screen和名为turtle的Turtle实例：

```
>>> screen.onclick(turtle.goto) # Subsequently clicking into the TurtleScreen wil
>>>                            # make the turtle move to the clicked point.
>>> screen.onclick(None)        # remove event binding again
```

注意： 此TurtleScreen方法仅在名称下作为全局函数提供onscreenclick。全局函数onclick是从乌龟方法派生的另一个函数onclick。

turtle.**ontimer**(*fun*, *t = 0*)

参数：	• **有趣** - 一个没有参数的函数 • **t** - 数字> = 0

安装一个在*t*毫秒后调用*乐趣*的定时器。

```
>>> running = True
>>> def f():
...     if running:
...         fd(50)
...         lt(60)
...         screen.ontimer(f, 250)
>>> f()   ### makes the turtle march around
>>> running = False
```

turtle.**mainloop**( )

turtle.**done**( )

开始事件循环 - 调用Tkinter的mainloop函数。必须是乌龟图形程序中的最后一个语句。必须*不能*交互使用海龟作图的-如果脚本是从IDLE内-n模式（无子）运行使用。

```
>>> screen.mainloop()
```

24.1.4.4。输入法

turtle.**textinput**(*标题*, *提示*)

参数：	• **标题** - 字符串 • **提示** - 字符串

弹出一个用于输入字符串的对话窗口。参数标题是对话窗口的标题，提示是主要描述要输入什么信息的文本。返回字符串输入。如果该对话框被取消，则返回None。

```
>>> screen.textinput("NIM", "Name of first player:")
```

turtle.**numinput**(*title*, *prompt*, *default = None*, *minval = None*, *maxval = None*)

参数：	• **标题** - 字符串 • **提示** - 字符串 • **默认** - 数字（可选） • **minval** - 数字（可选） • **maxval** - 数字（可选）

弹出一个对话窗口输入一个数字。标题是对话窗口的标题，提示是主要描述输入什么数字信息的文本。默认值：默认值，最小值：输入的最小值，最大值：输入的最大值输入的数字必须在minval .. maxval范围内（如果给出）。如果没有，则发出提示并且对话框保持打开状态以进行更正。返回数字输入。如果该对话框被取消，则返回None。

```
>>> screen.numinput("Poker", "Your stakes:", 1000, minval=10, maxval=10000)
```

24.1.4.5。设置和特殊方法

turtle.**mode**(*mode = None*)

参数：	**模式** - 其中一个字符串“标准”，“标识”或“世界”

设置乌龟模式（“标准”，“标志”或“世界”）并执行重置。如果没有给出模式，则返回当前模式。

模式“标准”与旧版本兼容turtle。模式“logo”与大多数Logo乌龟图形兼容。模式“世界”使用用户定义的“世界坐标”。**注意**：在这种模式下，如果x/y单位比例不等于1，则角度会出现扭曲。

模式	**最初的海龟标题**	**积极的角度**
“标准”	向右（东）	逆时针
“商标”	向上（北）	顺时针

```
>>> mode("logo")   # resets turtle heading to north
>>> mode()
'logo'
```

turtle.**colormode**(*cmode = None*)

参数：	**cmode** - 其中一个值为1.0或255

返回colormode或将其设置为1.0或255.随后 颜色三元组的*r*，*g*，*b*值必须在0 .. *cmode*范围内。

```
>>> screen.colormode(1)
>>> turtle.pencolor(240, 160, 80)
Traceback (most recent call last):
     ...
TurtleGraphicsError: bad color sequence: (240, 160, 80)
>>> screen.colormode()
1.0
>>> screen.colormode(255)
>>> screen.colormode()
255
>>> turtle.pencolor(240,160,80)
```

turtle.**getcanvas**( )

返回此TurtleScreen的画布。对于知道如何处理Tkinter帆布的内部人士非常有用。

```
>>> cv = screen.getcanvas()
>>> cv
<turtle.ScrolledCanvas object ...>
```

turtle.**getshapes**( )

返回所有当前可用乌龟形状的名称列表。

```
>>> screen.getshapes()
['arrow', 'blank', 'circle', ..., 'turtle']
```

turtle.**register_shape**(*名称*, *形状=无*)

turtle.**addshape**(*名称*, *形状=无*)

有三种不同的方法可以调用这个函数：

1. *name*是gif文件的名称，*形状*是None：安装相应的图像形状。

   ```
   >>> screen.register_shape("turtle.gif")
   ```

   注意: 转动龟时图像形状*不*旋转，因此它们不显示龟的标题！

2. *name*是一个任意字符串，*shape*是坐标对的元组：安装相应的多边形形状。

   ```
   >>> screen.register_shape("triangle", ((5,-3), (0,5), (-5,-3)))
   ```

3. *name*是一个任意字符串，shape是一个（复合）Shape 对象：安装相应的复合形状。

将龟形添加到TurtleScreen的形状列表中。只有这样才能注册的形状可以通过发出命令来使用shape(shapename)。

turtle.**turtles**( )

返回屏幕上的乌龟列表。

```
>>> for turtle in screen.turtles():
...     turtle.color("red")
```

turtle.**window_height**( )

返回乌龟窗口的高度。

```
>>> screen.window_height()
480
```

turtle.**window_width**( )

返回乌龟窗口的宽度。

```
>>> screen.window_width()
640
```

24.1.4.6。具体到屏幕的方法，而不是从TurtleScreen继承

turtle.bye（）

关闭turtlegraphics窗口。

turtle.exitonclick（）

将bye（）方法绑定到屏幕上的鼠标点击。

如果配置字典中的值“using_IDLE”是False（默认值），那么也输入mainloop。备注：如果使用带-n开关的IDLE（无子处理），则应将此值设置为Truein turtle.cfg。在这种情况下，IDLE自己的主循环也对客户端脚本有效。

turtle.setup（*width = _CFG [“width”]，height = _CFG [“height”]，startx = _CFG [“leftright”]，starty = _CFG [“topbottom”]*）

设置主窗口的大小和位置。参数的默认值存储在配置字典中，可以通过turtle.cfg文件进行更改。

参数：	• **宽度** - 如果是整数，则以像素为单位，如果是浮点，屏幕的一部分; 默认为屏幕的50％ • **高度** - 如果是整数，高度以像素为单位，如果是浮点数，则为屏幕的一小部分; 默认是屏幕的75％ • **startx** - 如果为正值，**则以**屏幕左边缘的像素为起始位置，如果右边缘为负值，如果None，中心窗口水平 • **starty** - 如果是正数，则从屏幕顶部边缘开始以像素为单位，如果底部边缘为负数，如果None，垂直居中窗口

```
>>> screen.setup (width=200, height=200, startx=0, starty=0)
>>>              # sets window to 200x200 pixels, in upper left of screen
>>> screen.setup(width=.75, height=0.5, startx=None, starty=None)
>>>              # sets window to 75% of screen by 50% of screen and centers
```

turtle.title（*titlestring*）

参数：	**titlestring** - 显示在乌龟图形窗口的标题栏中的字符串

将turtle窗口的标题设置为*titlestring*。

```
>>> screen.title("Welcome to the turtle zoo!")
```

24.1.5。公共类

class turtle.RawTurtle（*canvas*）

class turtle.RawPen（*canvas*）

参数：	**画布** - a tkinter.Canvas，a ScrolledCanvas或a TurtleScreen

创建一只乌龟。龟具有上述所有方法作为“乌龟/ RawTurtle的方法”。

*类*turtle.Turtle

RawTurtle的子类，具有相同的界面，但是Screen第一次需要时会自动创建一个默认 对象。

class turtle.TurtleScreen（*cv*）

参数：	**cv** - a tkinter.Canvas

提供setbg()如上所述的面向屏幕的方法等。

*类*turtle.Screen

TurtleScreen的子类，添加了四种方法。

*类*turtle.ScrolledCanvas（*主*）

参数：	**主** - 一些Tkinter小部件包含ScrolledCanvas，即添加了滚动条的Tkinter-canvas

由类Screen使用，从而自动为龟提供一个ScrolledCanvas作为操场。

class turtle.Shape（*type_*，*data*）

参数：	**type_** - 其中一个字符串“polygon”，“image”，“compound”

数据结构建模形状。双方必须遵循这个规范：(type_, data)

类型_	***数据***
“多边形”	多边形元组，即坐标对的元组
“图片”	一张图片（这种形式只能在内部使用！）
“复合”	None（必须使用该addcomponent()方法构造复合形状）

addcomponent（*poly*，*fill*，*outline* = *None*）

参数：	• **poly** - 一个多边形，即数字对的元组 • **填充** - *聚合物*将填充的颜色 • **轮廓** - poly的轮廓颜色（如果有的话）

例：

```
>>> poly = ((0,0),(10,-5),(0,10),(-10,-5))
>>> s = Shape("compound")
>>> s.addcomponent(poly, "red", "blue")
>>> # ... add more components and then use register_shape()
```

请参阅复合形状。

class turtle.Vec2D（*x*，*y*）

一个二维矢量类，用作实现龟图形的辅助类。对乌龟图形程序也可能有用。派生自元组，所以一个向量就是一个元组！

提供（对于*a*，*b*矢量，*k*数字）：

- a + b 矢量添加
- a - b 矢量减法
- a * b 内在产品
- k * a并与标量相乘a * k
- abs(a) a的绝对值
- a.rotate(angle) 回转

24.1.6。帮助和配置

24.1.6.1。如何使用帮助

Screen和Turtle类的公共方法通过文档大量记录。所以这些可以通过Python帮助工具作为在线帮助使用：

- 使用IDLE时，工具提示将显示函数/方法调用中键入的文档字符串的签名和第一行。
- 调用help()方法或函数显示文档字符串：

```
>>> help(Screen.bgcolor)
Help on method bgcolor in module turtle:

bgcolor(self, *args) unbound turtle.Screen method
    Set or return backgroundcolor of the TurtleScreen.

    Arguments (if given): a color string or three numbers
    in the range 0..colormode or a 3-tuple of such numbers.


      >>> screen.bgcolor("orange")
      >>> screen.bgcolor()
      "orange"
      >>> screen.bgcolor(0.5,0,0.5)
      >>> screen.bgcolor()
      "#800080"

>>> help(Turtle.penup)
Help on method penup in module turtle:

penup(self) unbound turtle.Turtle method
    Pull the pen up -- no drawing when moving.

    Aliases: penup | pu | up

    No argument

    >>> turtle.penup()
```

- 从方法派生的函数的文档有一个修改后的形式：

```
>>> help(bgcolor)
Help on function bgcolor in module turtle:

bgcolor(*args)
    Set or return backgroundcolor of the TurtleScreen.
```

```
    Arguments (if given): a color string or three numbers
    in the range 0..colormode or a 3-tuple of such numbers.

    Example::

      >>> bgcolor("orange")
      >>> bgcolor()
      "orange"
      >>> bgcolor(0.5,0,0.5)
      >>> bgcolor()
      "#800080"

>>> help(penup)
Help on function penup in module turtle:

penup()
    Pull the pen up -- no drawing when moving.

    Aliases: penup | pu | up

    No argument

    Example:
    >>> penup()
```

这些修改后的文档字符串会与导入时从方法派生的函数定义一起自动创建。

24.1.6.2。docstrings翻译成不同的语言

有一个实用工具可以创建一个字典，其中的关键字是方法名称，其值是Screen和Turtle类的公共方法的文档字符串。

turtle.**write_docstringdict**(*filename="turtle_docstringdict"*)

参数：	**文件名** - 一个字符串，用作文件名

使用给定的文件名创建并将docstring-dictionary写入Python脚本。这个函数必须显式调用（它不会被turtle图形类使用）。docstring字典将被写入Python脚本。它旨在作为将文档翻译成不同语言的模板。 *filename*.py

如果您（或您的学生）希望以turtle您的母语使用在线帮助，则必须翻译文档并将结果文件保存为例如turtle_docstringdict_german.py。

如果您的turtle.cfg文件中有适当的条目，则此字典将在导入时读入，并将替换原始的英文文档字符串。

在写这篇文章时，有德语和意大利语的docstring字典。（要求，请以glingl @ AON 。在。）

24.1.6.3。如何配置屏幕和海龟

内置的默认配置模仿旧乌龟模块的外观和行为，以保持最佳的兼容性。

如果您想要使用不同的配置，以更好地反映此模块的功能或更适合您需求的配置，例如在教室中使用，您可以准备一个配置文件turtle.cfg，该配置文件将在导入时读取，并根据其设置。

内置配置将对应于以下turtle.cfg：

```
width = 0.5
height = 0.75
leftright = None
topbottom = None
canvwidth = 400
canvheight = 300
mode = standard
colormode = 1.0
delay = 10
undobuffersize = 1000
shape = classic
pencolor = black
fillcolor = black
resizemode = noresize
visible = True
language = english
exampleturtle = turtle
examplescreen = screen
title = Python Turtle Graphics
using_IDLE = False
```

选定条目的简短说明：

- 前四行对应于该Screen.setup() 方法的参数。
- 第5行和第6行对应于方法的参数 Screen.screensize()。
- *形状*可以是任何内置形状，例如：箭头，乌龟等。欲了解更多信息，请尝试help(shape)。
- 如果你不想使用fillcolor（即使乌龟透明），你必须写（但所有非空字符串不得在cfg文件中有引号）。fillcolor = ""
- 如果你想反映乌龟的状态，你必须使用。resizemode = auto
- 如果你设置了例如docstringdict 将在导入时加载（如果存在于导入路径中，例如在与导入路径相同的目录中）。language = italianturtle_docstringdict_italian.pyturtle
- 条目*exampleturtle*和*examplescreen*定义了这些对象在文档中出现的名称。方法文档字符串到函数文档字符串的转换将从文档字符串中删除这些名称。
- *using_IDLE*：True如果您经常使用IDLE及其-n开关（“无子进程”），请将其设置为。这将阻止exitonclick()进入主循环。

在存储turtle.cfg目录中可以有一个文件，而turtle在当前工作目录中可以有一个文件。后者将覆盖第一个的设置。

该Lib/turtledemo目录包含一个turtle.cfg文件。您可以将其作为示例进行研究，并在运行演示时查看其效果（最好不要在演示查看器中）。

24.1.7。 turtledemo- 演示脚本

该turtledemo软件包包含一组演示脚本。这些脚本可以使用提供的演示查看器运行和查看，如下所示：

```
python -m turtledemo
```

或者，您可以单独运行演示脚本。例如，

```
python -m turtledemo.bytedesign
```

该`turtledemo`包目录包含：

- 演示查看器`__main__.py`可用于查看脚本的源代码并同时运行它们。
- 演示`turtle` 模块不同功能的多个脚本。可以通过示例菜单访问示例。它们也可以独立运行。
- 一个`turtle.cfg`其作为如何编写和使用这些文件的示例文件。

演示脚本是：

名称	描述	特征
bytedesign	复杂的古典乌龟图形模式	`tracer()`，延迟， `update()`
混沌	[illegible]计算有	世界坐标
时钟	显示计算机时间的模拟时钟	乌龟像钟表的手，在时间
colormixer	用r，g，b进行实验	`ondrag()`
森林	3个宽度优先的树	随机
fractalcurves	希尔伯特和科赫曲线	递归
lindenmayer	民族数学（印度科兰人）	L-系统
minimal_hanoi	河内塔	[illegible]矩形海龟（形状，
NIM	[illegible]三根棍子对着电脑玩古典的nim游	海龟作为[illegible]）事件驱动
涂料	超极简主义绘画程序	`onclick()`
和平	初级	乌龟：外观和动画
彭罗斯	用风筝和飞镖进行非周期性的平铺	`stamp()`
planet_and_moon	引力系统的模拟	复合形状， `Vec2D`
round_dance	跳舞的乌龟沿着相反的方向成对旋转	[illegible]
sorting_animate	可视化演示不同的分类方法	简单对齐，随机化
树	器个（图形）广度优先树（使用生成	`clone()`
two_canvases	简单的设计	两个画布上的海龟
维基百科	来自维基百科上关于乌龟图形的文章	`clone()`， `undo()`
银洋	另一个基本例子	`circle()`

玩的开心！

24.1.8。因为Python 2.6的变化

- 的方法`Turtle.tracer()`，`Turtle.window_width()`和 `Turtle.window_height()`已被淘汰。具有这些名称和功能的方法现在仅可用作方法`Screen`。从这些派生出来的功能仍然可用。（事实上，在Python 2.6中，这些方法仅仅是相应的`TurtleScreen`/ 方法的重复 `Screen`。）
- 该方法`Turtle.fill()`已被淘汰。其行为`begin_fill()`和`end_fill()` 变化略有改变：现在每个灌装流程都必须通过`end_fill()`电话完成 。
- 一种方法`Turtle.filling()`已被添加。它返回一个布尔值：`True`如果填充过程正在进行，`False`否则。这个行为对应`fill()`于Python 2.6中没有参数的调用。

24.1.9。自Python 3.0以来的变化

- 的方法Turtle.shearfactor()，Turtle.shapetransform()并 Turtle.get_shapepoly()已被添加。因此，全系列的正则线性变换现在可用于变换龟的形状。 Turtle.tiltangle()在功能上得到了增强：它现在可以用于获取或设置倾斜角度。 Turtle.settiltangle()已被弃用。
- 该方法Screen.onkeypress()已被添加为 Screen.onkey()实际上将操作绑定到keyrelease事件的补充。因此后者有一个别名：Screen.onkeyrelease()。
- 该方法 Screen.mainloop()已添加。所以当只使用Screen和Turtle对象时，不能另外导入mainloop()。
- 两种输入方式已添加Screen.textinput()和 Screen.numinput()。这些弹出式输入对话框分别返回字符串和数字。
- 两个示例脚本tdemo_nim.py以及tdemo_round_dance.py 已添加到Lib/turtledemo目录中。

24.2。 cmd- 支持面向行的命令解释器

源代码： Lib / cmd.py

本Cmd类为编写面向行的命令解释一个简单的框架。这些对于测试线束，管理工具和原型来说通常很有用，这些工具和原型稍后将被包装在更复杂的界面中。

class cmd. Cmd（ *completekey ='tab'*，*stdin = None*，*stdout = None* ）

一个Cmd实例或子类实例是一个面向行的解释器框架。没有充分的理由来实例化Cmd自己; 相反，它作为您自己定义的解释器类的超类很有用，以继承Cmd方法和封装操作方法。

可选参数*completekey*是readline完成键的名称; 它默认为Tab。如果*completekey*不None和readline是可用的，命令完成自动完成。

可选参数*stdin*和*stdout*指定了Cmd实例或子类实例将用于输入和输出的输入和输出文件对象。如果没有指定，他们将默认为sys.stdin和 sys.stdout。

如果你想要使用一个给定的*stdin*，确保将实例的use_rawinput属性设置 为False，否则*stdin*将被忽略。

24.2.1。Cmd对象

一个Cmd实例有以下方法：

Cmd. cmdloop（ *intro = None* ）

反复发出提示，接受输入，从收到的输入中解析出一个初始前缀，并分派给操作方法，将其余的行作为参数传递给它们。

可选参数是在第一个提示之前发布的横幅或介绍字符串（这将覆盖intro类属性）。

如果readline模块被加载，输入将自动继承 **类似bash**的历史列表编辑（例如，Control-P滚动回到最后一个命令，Control-N转到下一个命令，以Control-F 非破坏性的方式向右Control-B移动光标，破坏性地等）。

输入的文件结束被作为字符串传回'EOF'。

一个解释器实例会识别一个命令名称，foo当且仅当它有一个方法do_foo()。作为一种特殊情况，以字符开头的行将'?'被分派给该方法do_help()。作为另一种特殊情况，以字符开头的行将'!'被分派给方法do_shell()（如果定义了这种方法）。

当postcmd()方法返回一个真值时，这个方法会返回。该*站*参数postcmd()是命令的相应的返回值do_*()的方法。

如果启用了完成，完成命令将自动完成，并通过调用complete_foo()参数*text*，*line*，*begidx*和*endidx*来完成命令args 。 *text*是我们试图匹配的字符串前缀：所有返回的匹配都必须以它开头。*行*是删除前导空白的当前输入行，*begidx*和*endidx* 是前缀文本的开始和结束索引，可用于根据参数位于哪个位置来提供不同的完成。

所有Cmd继承预定义的子类do_help()。此方法使用参数'bar'调用，调用相应的方法help_bar()，如果不存在，则打印文档字符串（ do_bar()如果可用）。如果没有参数，则do_help()列出所有可用的帮助主题（即所有具有相应 help_*()方法或具有文档字符串的命令的命令），并列出所有未记录的命令。

Cmd.onecmd(*str*)

解释参数，就好像它是根据提示输入的一样。这可能会被忽略，但通常不需要; 请参阅precmd()和postcmd()有用的执行钩子的方法。返回值是一个标志，指示解释器是否应停止解释命令。如果*str*do_*()命令有一个方法 ，则返回该方法的返回值，否则返回该方法的返回值。 default()

Cmd.emptyline( )

在响应提示输入空行时调用的方法。如果此方法未被覆盖，则重复输入的最后一个非空命令。

Cmd.default(*线*)

在命令前缀未被识别时在输入行上调用的方法。如果这个方法没有被覆盖，它会输出一个错误信息并返回。

Cmd.completedefault(*文本*，*行*，*begidx*，*endidx*)

当没有特定于命令的complete_*()方法可用时，调用方法来完成输入行 。默认情况下，它返回一个空列表。

Cmd.precmd(*线*)

钩方法执行的命令行之前*行*解释，但会产生和发出的输入提示之后。这个方法是一个存根Cmd; 它存在被子类覆盖。返回值用作将由方法执行的命令onecmd(); 该 precmd()实现可以重新写入命令或简单地返回*线*不变。

Cmd.postcmd(*停止*，*行*)

Hook方法在命令调度完成后执行。这个方法是一个存根Cmd; 它存在被子类覆盖。 *line*是执行的命令行，*stop*是一个标志，表示在调用之后执行是否会终止postcmd(); 这将是方法的返回值onecmd()。该方法的返回值将被用作与*停止*对应的内部标志的新值; 返回错误将导致解释继续。

Cmd.preloop( )

钩子方法cmdloop()被调用时执行一次。这个方法是一个存根Cmd; 它存在被子类覆盖。

Cmd.postloop( )

Hook方法在cmdloop()即将返回时执行一次。这个方法是一个存根Cmd; 它存在被子类覆盖。

Cmd子类的实例具有一些公共实例变量：

Cmd.prompt

发出提示以请求输入。

Cmd.identchars

接受命令前缀的字符串。

Cmd.lastcmd
最后看到非空命令前缀。

Cmd.cmdqueue
排队的输入行列表。cmdloop()当需要新输入时，会检查cmdqueue列表；如果它不是空的，它的元素将按顺序处理，就像在提示符处输入一样。

Cmd.intro
作为介绍或横幅发布的字符串。可以通过给cmdloop()方法一个参数来覆盖 。

Cmd.doc_header
如果帮助输出包含用于记录的命令的部分，则发出头文件。

Cmd.misc_header
如果帮助输出具有用于其他帮助主题的部分（即，help_*()没有相应 do_*()方法的方法），则发出头文件。

Cmd.undoc_header
如果帮助输出具有未记录命令的部分（即，do_*()没有相应help_*() 方法的方法），则发出头文件。

Cmd.ruler
用于在帮助消息标题下绘制分隔线的字符。如果为空，则不绘制标尺线。它默认为'='。

Cmd.use_rawinput
一面旗帜，默认为真。如果为true，则cmdloop()用于input()显示提示并阅读下一个命令;如果错误，sys.stdout.write() 并被sys.stdin.readline()使用。（这意味着通过readline在支持它的系统上导入，解释器将自动支持类**Emacs**的行编辑和命令历史击键。）

24.2.2。Cmd示例

该cmd模块主要用于构建定制shell，以便用户以交互方式处理程序。

本节介绍如何围绕turtle模块中的一些命令构建shell的简单示例。

基本的乌龟命令，如forward()添加到Cmd名为的方法的 子类do_forward()。该参数被转换为一个数字并发送到乌龟模块。文档字符串用在shell提供的帮助工具中。

该示例还包括一个基本的记录和回放工具，该工具使用precmd()负责将输入转换为小写并将命令写入文件的方法实现。该do_playback()方法读取文件并将录制的命令添加到cmdqueue即时回放中：

```
import cmd, sys
from turtle import *

class TurtleShell(cmd.Cmd):
    intro = 'Welcome to the turtle shell.   Type help or ? to list commands.\n'
    prompt = '(turtle) '
```

```
    file = None

    # ----- basic turtle commands -----
    def do_forward(self, arg):
        'Move the turtle forward by the specified distance:  FORWARD 10'
        forward(*parse(arg))
    def do_right(self, arg):
        'Turn turtle right by given number of degrees:  RIGHT 20'
        right(*parse(arg))
    def do_left(self, arg):
        'Turn turtle left by given number of degrees:  LEFT 90'
        left(*parse(arg))
    def do_goto(self, arg):
        'Move turtle to an absolute position with changing orientation.  GOTO 100 200'
        goto(*parse(arg))
    def do_home(self, arg):
        'Return turtle to the home position:  HOME'
        home()
    def do_circle(self, arg):
        'Draw circle with given radius an options extent and steps:  CIRCLE 50'
        circle(*parse(arg))
    def do_position(self, arg):
        'Print the current turtle position:  POSITION'
        print('Current position is %d %d\n' % position())
    def do_heading(self, arg):
        'Print the current turtle heading in degrees:  HEADING'
        print('Current heading is %d\n' % (heading(),))
    def do_color(self, arg):
        'Set the color:  COLOR BLUE'
        color(arg.lower())
    def do_undo(self, arg):
        'Undo (repeatedly) the last turtle action(s):  UNDO'
    def do_reset(self, arg):
        'Clear the screen and return turtle to center:  RESET'
        reset()
    def do_bye(self, arg):
        'Stop recording, close the turtle window, and exit:  BYE'
        print('Thank you for using Turtle')
        self.close()
        bye()
        return True

    # ----- record and playback -----
    def do_record(self, arg):
        'Save future commands to filename:  RECORD rose.cmd'
        self.file = open(arg, 'w')
    def do_playback(self, arg):
        'Playback commands from a file:  PLAYBACK rose.cmd'
        self.close()
        with open(arg) as f:
            self.cmdqueue.extend(f.read().splitlines())
    def precmd(self, line):
        line = line.lower()
        if self.file and 'playback' not in line:
            print(line, file=self.file)
        return line
    def close(self):
        if self.file:
```

```
            self.file.close()
            self.file = None

def parse(arg):
    'Convert a series of zero or more numbers to an argument tuple'
    return tuple(map(int, arg.split()))

if __name__ == '__main__':
    TurtleShell().cmdloop()
```

下面是一个示例会话，其中显示了帮助功能，使用空行重复命令以及简单的录制和播放功能：

```
Welcome to the turtle shell.   Type help or ? to list commands.

(turtle) ?

Documented commands (type help <topic>):
========================================
bye     color    goto     home  playback  record  right
circle  forward  heading  left  position  reset   undo

(turtle) help forward
Move the turtle forward by the specified distance:  FORWARD 10
(turtle) record spiral.cmd
(turtle) position
Current position is 0 0

(turtle) heading
Current heading is 0

(turtle) reset
(turtle) circle 20
(turtle) right 30
(turtle) circle 40
(turtle) right 30
(turtle) circle 60
(turtle) right 30
(turtle) circle 80
(turtle) right 30
(turtle) circle 100
(turtle) right 30
(turtle) circle 120
(turtle) right 30
(turtle) circle 120
(turtle) heading
Current heading is 180

(turtle) forward 100
(turtle)
(turtle) right 90
(turtle) forward 100
(turtle)
(turtle) right 90
(turtle) forward 400
(turtle) right 90
(turtle) forward 500
```

```
(turtle) right 90
(turtle) forward 400
(turtle) right 90
(turtle) forward 300
(turtle) playback spiral.cmd
Current position is 0 0

Current heading is 0

Current heading is 180

(turtle) bye
Thank you for using Turtle
```

24.3。 shlex- 简单的词法分析

源代码： Lib / shlex.py

本shlex类可以很容易地编写简单的语法类似的Unix外壳的词法分析器。这对编写微型语言（例如，在Python应用程序的运行控制文件中）或解析带引号的字符串通常很有用。

该shlex模块定义了以下功能：

shlex. **split** (*s* , *comments = False* , *posix = True*)

拆分字符串*小号*使用shell的语法。如果*注释*是False （默认），则给定字符串中的注释解析将被禁用（将实例的commenters属性 设置shlex为空字符串）。该函数默认在POSIX模式下运行，但如果*posix*参数为false ，则使用非POSIX模式。

> **注意：** 由于split()功能实例化一个shlex 实例，传递None用于*小号*将读取的字符串从标准输入分裂。

shlex. **quote** (*s*)

返回字符串s的shell转义版本。返回的值是一个字符串，可安全地用作shell命令行中的一个标记，用于不能使用列表的情况。

这个习语会是不安全的：

```
>>> filename = 'somefile; rm -rf ~'
>>> command = 'ls -l {}'.format(filename)
>>> print(command)  # executed by a shell: boom!
ls -l somefile; rm -rf ~
```

quote() 让你插上安全漏洞：

```
>>> command = 'ls -l {}'.format(quote(filename))
>>> print(command)
ls -l 'somefile; rm -rf ~'
>>> remote_command = 'ssh home {}'.format(quote(command))
>>> print(remote_command)
ssh home 'ls -l '"'"'somefile; rm -rf ~'"'"''
```

引用与UNIX shell兼容，并且split()：

```
>>> remote_command = split(remote_command)
>>> remote_command
['ssh', 'home', "ls -l 'somefile; rm -rf ~'"]
>>> command = split(remote_command[-1])
>>> command
['ls', '-l', 'somefile; rm -rf ~']
```

3.3版本的新功能

该shlex模块定义了以下类：

class shlex. shlex（*instream = None*，*infile = None*，*posix = False*，*punctuation_chars = False* ）

一个shlex实例或子类实例是一个词法分析器对象。初始化参数（如果存在）指定从哪里读取字符。它必须是一个文件- /流状物体 read()和readline()方法，或字符串。如果没有提供任何参数，则会从中提取输入sys.stdin。第二个可选参数是文件名字符串，它设置infile属性的初始值。如果*instream* 参数省略或等于sys.stdin，则此第二个参数默认为“stdin”。的*POSIX*参数定义的操作模式：当*POSIX*是不正确的（默认），shlex实例将在兼容模式运行。在POSIX模式下运行时， shlex将尝试尽可能接近POSIX shell解析规则。该*punctuation_chars*参数提供了一种方法，使行为更接近它们如何真正的解析。这可能需要很多值：默认值False，保留在Python 3.5及更早版本中看到的行为。如果设置为True，则();<>|&更改字符的分析 ：这些字符（视为标点符号）的任何运行都会作为单个标记返回。如果设置为非空字符串，则这些字符将用作标点字符。 wordchars出现在*punctuation_chars*中的属性中的 任何字符都将从中删除wordchars。有关更多信息，请参阅 提高与Shell的兼容性。

*在3.6版本中更改：*将*punctuation_chars*加入参数。

也可以看看:

模 configparser
解析器提供类似于Windows .ini文件的配置文件。

24.3.1。shlex对象

一个shlex实例有以下方法：

shlex. get_token（ ）
返回一个令牌。如果令牌已经堆叠使用push_token()，将令牌从堆栈中弹出。否则，从输入流中读取一个。如果读取遇到立即文件结束，eof则返回（''非POSIX模式和NonePOSIX模式下的空字符串（ ）。

shlex. push_token（*str* ）
将参数推入令牌堆栈。

shlex. read_token（ ）
阅读原始令牌。忽略回推堆栈，不解释源请求。（这通常不是一个有用的入口点，并且仅在此处为了完整而记录。）

shlex. sourcehook（*文件名*）
当shlex检测到一个源请求时（见source 下面），这个方法被赋予下面的标记作为参数，并且期望返回一个由文件名和打开的文件类对象组成的元组。

通常，这种方法首先将任何引号从参数中删除。如果结果是绝对路径名，或者以前的源请求没有生效，或者以前的源是流（如sys.stdin），则结果将保留。否则，如果结果是一个

相对路径名，紧接在源包含堆栈之前的文件的名称的目录部分被预置（这种行为就像C预处理器处理的方式）。#include "file.h"

操作的结果被视为文件名，并作为元组的第一个元素返回，并open()调用它以产生第二个组件。（注意：这与实例初始化中的参数顺序相反！）

该钩子已公开，因此您可以使用它来实现目录搜索路径，添加文件扩展名以及其他名称空间黑客。没有相应的'关闭'钩子，但shlex实例close()在返回EOF时会调用源输入流的 方法。

要更加明确地控制源码堆栈，请使用push_source()和 pop_source()方法。

shlex.**push_source**(*newstream*, *newfile* = *None*)

将输入源流推入输入堆栈。如果指定了文件名参数，稍后将在错误消息中使用。这与该方法内部使用的sourcehook()方法相同。

shlex.**pop_source**()

弹出输入堆栈中最后推送的输入源。当词法分析器在叠加输入流上达到EOF时，这与内部使用的方法相同。

shlex.**error_leader**(*infile* = *None*, *lineno* = *None*)

此方法以Unix C编译器错误标签的格式生成错误消息引导程序; 格式是，其中用当前源文件的名称和当前输入行号替换（可选参数可用于覆盖它们）。'"%s", line %d: '%s%d

提供这种便利是为了鼓励shlex用户以Emacs和其他Unix工具所理解的标准可解析格式生成错误消息。

shlex子类的实例具有一些公共实例变量，这些变量既可以控制词法分析，也可以用于调试：

shlex.**commenters**

被识别为评论初学者的字符串。从注释初学者到行尾的所有字符都被忽略。包括'#'默认情况下。

shlex.**wordchars**

将积累成多字符标记的字符串。默认情况下，包含所有ASCII字母数字和下划线。在POSIX模式下，Latin-1集中的重音字符也包含在内。如果 punctuation_chars不为空，则~-./*?= 可能出现在文件名规范和命令行参数中的字符也将包含在此属性中，并且出现在其中的任何字符punctuation_chars将在其出现 时被删除wordchars。

shlex.**whitespace**

将被视为空白并跳过的字符。空格限制令牌。默认情况下，包括空格，制表符，换行符和回车符。

shlex.**escape**

将被视为逃脱的字符。这将仅用于POSIX模式，并且仅包括'\'默认值。

shlex.**quotes**

将被视为字符串的字符。令牌会累积，直到再次遇到相同的引号（因此，不同的引号类型会像在shell中一样保护彼此）。缺省情况下，包含ASCII单引号和双引号。

shlex.escapedquotes

quotes那里的字符将解释在中定义的转义字符 escape。这仅在POSIX模式下使用，并且仅包括'"'默认值。

shlex.whitespace_split

如果True，令牌只会被分割成空格。例如，这对解析命令行很有用，以shlex类似于shell参数的方式获取令牌。如果这个属性是True， punctuation_chars将不起作用，并且分割只会发生在空格上。在使用时punctuation_chars，它旨在提供更接近shell实现的解析，建议保留whitespace_split为False（默认值）。

shlex.infile

当前输入文件的名称，最初在类实例化时设置或由后来的源请求堆叠。在构建错误消息时检查这一点可能很有用。

shlex.instream

此shlex实例正在读取字符的输入流。

shlex.source

该属性是None默认的。如果您为其分配字符串，则该字符串将被识别为与source各种shell中的关键字类似的词法级别包含请求 。也就是说，紧随其后的标记将作为文件名打开，输入将从该流中取出，直到EOF，此时close()将调用该流的方法，输入源将再次成为原始输入流。源请求可以被堆叠到任意深度级别。

shlex.debug

如果此属性是数字1或更多，则shlex 实例将在其行为上输出详细的进度输出。如果你需要使用它，你可以阅读模块源代码来了解详细信息。

shlex.lineno

源代码行数（迄今为止看到的换行数加1）。

shlex.token

令牌缓冲区。在捕捉异常时检查这一点可能很有用。

shlex.eof

用于确定文件结束的令牌。这将被设置为空字符串（''），非POSIX模式和NonePOSIX模式。

shlex.punctuation_chars

将被视为标点的字符。标点字符的运行将作为单个标记返回。但是，请注意，不会执行语义有效性检查：例如，'>>>'可能会作为标记返回，即使它可能不被shell识别。

3.6版本中的新功能。

24.3.2。解析规则

在非POSIX模式下运行时，shlex会尝试遵守以下规则。

- 引号字符在单词中不被识别（Do"Not"Separate被解析为单个单词Do"Not"Separate）；

- 转义字符不被识别;
- 用引号将字符括起来保留引号内所有字符的字面值;
- 关闭引号分开单词（ "Do"Separate被解析为"Do"和 Separate ）;
- 如果whitespace_split是False ，任何未声明为单词字符，空格或引号的字符将作为单字符标记返回。如果是这样True ， shlex只会将空格分割成空格;
- EOF用空字符串（''）表示。
- 即使引用了空字符串也不可能解析。

在POSIX模式下运行时， shlex会尝试遵守以下解析规则。

- 行情被剥离出来，不要分开单词（ "Do"Not"Separate"被解析为单个单词DoNotSeparate ）;
- 非引号转义字符（例如'\'）保留下一个字符的字面值;
- 将引号中的字符括起来 escapedquotes （例如"'"）保留引号内所有字符的字面值;
- 将引号中的字符括起来 escapedquotes （例如'"'）保留引号内的所有字符的字面值，除了在中提到的字符escape。转义字符只有在后面跟着使用的引号或转义字符本身时才保留其特殊含义。否则，转义字符将被视为正常字符。
- EOF通过一个None值发送信号;
- 引用空字符串（''）是允许的。

24.3.3。改进了与贝壳的兼容性

3.6版本中的新功能。

的shlex类提供兼容性与由普通Unix外壳等进行解析bash ， dash和sh。要利用这种兼容性，请punctuation_chars在构造函数中指定参数。这默认为False ，它保留了3.6之前的行为。但是，如果它设置为True ，则字符的解析 ();<>|& 会发生变化：这些字符的任何运行都将作为单个标记返回。虽然缺少一个完整的shell解析器（这对于标准库来说已经超出了范围，但由于shell的多样性），它确实允许您比其他方式更容易地执行命令行处理。为了说明，您可以在以下代码片段中看到不同之处：

```
>>> import shlex
>>> text = "a && b; c && d || e; f >'abc'; (def \"ghi\")"
>>> list(shlex.shlex(text))
['a', '&', '&', 'b', ';', 'c', '&', '&', 'd', '|', '|', 'e', ';', 'f', '>',
"'abc'", ';', '(', 'def', '"ghi"', ')']
>>> list(shlex.shlex(text, punctuation_chars=True))
['a', '&&', 'b', ';', 'c', '&&', 'd', '||', 'e', ';', 'f', '>', "'abc'",
';', '(', 'def', '"ghi"', ')']
```

当然，令牌将被返回，这对shell无效，并且您需要对返回的令牌执行自己的错误检查。

True您可以传递具有特定字符的字符串，而不是将其作为punctuation_chars参数的值传递，这将用于确定哪些字符构成标点符号。例如：

```
>>> import shlex
>>> s = shlex.shlex("a && b || c", punctuation_chars="|")
>>> list(s)
['a', '&', '&', 'b', '||', 'c']
```

注意： 当punctuation_chars指定时，该wordchars 属性会增加字符~-./*?=。这是因为这些字符可能出现在文件名（包括通配符）和命令行参数（例如--color=auto ）中。因此：

```
>>> import shlex
>>> s = shlex.shlex('~/a && b-c --color=auto || d *.py?',
...                 punctuation_chars=True)
>>> list(s)
['~/a', '&&', 'b-c', '--color=auto', '||', 'd', '*.py?']
```

为了获得最佳效果，punctuation_chars 应该与之一起设置 posix=True。（请注意，这是默认值 shlex 的 posix=False。）

25.带有Tk的图形用户界面

Tk / Tcl一直是Python不可分割的一部分。它提供了一个健壮且独立于平台的窗口工具包，可供Python程序员使用该`tkinter`包及其扩展`tkinter.tix`和`tkinter.ttk`模块使用。

该`tkinter`软件包是Tcl / Tk之上的一个面向对象的薄层。要使用`tkinter`，您不需要编写Tcl代码，但需要查阅Tk文档以及偶尔使用的Tcl文档。 `tkinter`是一组将Tk小部件实现为Python类的包装器。另外，内部模块`_tkinter`提供了一个线程安全机制，允许Python和Tcl进行交互。

`tkinter`最主要的优点是速度很快，而且通常与Python捆绑在一起。虽然其标准文档很薄弱，但有很好的材料可供使用，其中包括：参考资料，教程，书籍和其他资料。`tkinter`也因过时的外观和感觉而出名，这在Tk8.5中得到了极大的改善。不过，您可能会感兴趣的还有许多其他GUI库。有关替代方法的更多信息，请参阅其他图形用户界面包部分。

25.1。 tkinter- Tcl / Tk的Python界面

源代码： Lib / tkinter / __ init__.py

的tkinter封装（“TK接口”）是标准的Python接口Tk的GUI工具包。Tk和tkinter在大多数Unix平台以及Windows系统上均可用。（Tk本身不是Python的一部分;它保存在ActiveState中。）

从命令行运行应打开一个窗口，演示一个简单的Tk接口，让您知道已正确安装在您的系统上，并且还显示安装了哪个版本的Tcl / Tk，因此您可以阅读特定于此的Tcl / Tk文档版。 python -m tkinter tkinter

也可以看看： Tkinter文档：

Python Tkinter资源
Python Tkinter主题指南提供了大量有关使用Python中的Tk以及链接到Tk上的其他信息源的信息。

TKDocs
广泛的教程和一些小部件的更友好的小部件页面。

Tkinter参考：Python的GUI
在线参考资料。

Tkinter文档来自effbot
由effbot.org支持的tkinter的在线参考。

编程Python
由马克卢茨书，有优良的Tkinter报道。

繁忙的Python开发人员的现代Tkinter
Mark Rozerman撰写关于用Python和Tkinter构建有吸引力的现代图形用户界面。

Python和Tkinter编程
由约翰格雷森预订（国际标准书号1-884777-81-3）。

Tcl / Tk文档：

Tk命令
大多数命令都可以作为tkinter或tkinter.ttk类来使用。更改'8.6'以匹配您的Tcl / Tk安装版本。

Tcl / Tk最近的手册页
最新的Tcl / Tk手册在www.tcl.tk上。

ActiveState Tcl主页
Tk / Tcl开发主要发生在ActiveState上。

Tcl和Tk工具包
Tcl的发明人John Ousterhout预定。

Tcl和Tk中的实用编程
布伦特韦尔奇的百科全书。

25.1.1。Tkinter模块

大部分时间tkinter都是你真正需要的，但也有一些额外的模块可供选择。Tk接口位于名为的二进制模块中_tkinter。该模块包含Tk的底层接口，不应直接由应用程序员使用。它通常是共享库（或DLL），但在某些情况下可能会与Python解释器静态链接。

除了Tk接口模块之外，tkinter还包括许多Python模块，tkinter.constants是最重要的模块之一。导入tkinter将自动导入tkinter.constants，因此，通常，使用Tkinter只需要一个简单的导入语句：

```
import tkinter
```

或者更经常地：

```
from tkinter import *
```

class tkinter.Tk（*screenName = None*，*baseName = None*，*className ='Tk'*，*useTk = 1* ）

将Tk类实例化没有参数。这会创建一个Tk的顶层小部件，它通常是应用程序的主窗口。每个实例都有自己关联的Tcl解释器。

tkinter.Tcl（*screenName = None*，*baseName = None*，*className ='Tk'*，*useTk = 0* ）

该Tcl()函数是一个工厂函数Tk，除了不初始化Tk子系统外，它创建的对象非常类似于该类创建的对象。在不希望创建无关的顶层窗口的环境中，或者在无法（如没有X服务器的Unix / Linux系统）环境中驱动Tcl解释器时，这通常很有用。由对象创建的Tcl()对象可以通过调用其loadtk()方法创建Toplevel窗口（并初始化Tk子系统）。

其他提供Tk支持的模块包括：

tkinter.scrolledtext
内置垂直滚动条的文本小部件。

tkinter.colorchooser
对话框让用户选择一种颜色。

tkinter.commondialog
在此处列出的其他模块中定义的对话框的基类。

tkinter.filedialog
常用对话框允许用户指定要打开或保存的文件。

tkinter.font
实用程序来帮助处理字体。

tkinter.messagebox
访问标准的Tk对话框。

tkinter.simpledialog
基本对话框和便利功能。

tkinter.dnd

拖放支持tkinter。这是实验性的，并且在用Tk DND代替时应该被弃用。

turtle
在Tk窗口中的乌龟图形。

25.1.2。Tkinter的救生衣

本节不是为Tk或Tkinter的详尽教程而设计的。相反，它旨在作为一个制止差距，为系统提供一些入门指导。

积分：

- Tk由John Ousterhout在伯克利书写。
- Tkinter由Steen Lumholt和Guido van Rossum撰写。
- 这个救生圈是由弗吉尼亚大学的马特康威编写的。
- HTML渲染和一些自由编辑是由Ken Manheimer的FrameMaker版本生成的。
- Fredrik Lundh详细阐述并修改了类接口描述，以使它们与Tk 4.2一致。
- Mike Clarkson将文档转换为LaTeX，并编译了参考手册的用户界面章节。

25.1.2.1。如何使用本节

本部分分为两部分：前半部分（大致）包含背景材料，后半部分可作为便于参考的键盘。

当试图回答“我该怎么做”这个形式的问题时，通常最好找出如何在直接Tk中做“无用”，然后将其转换回相应的tkinter呼叫。Python程序员通常可以通过查看Tk文档来猜测正确的Python命令。这意味着为了使用Tkinter，你必须知道一些关于Tk的知识。这个文件不能完成这个角色，所以我们能做的最好的事情就是向您提供最好的文档。以下是一些提示：

- 作者强烈建议获取Tk手册页的副本。具体来说，manN目录中的手册页是最有用的。该man3手册页介绍了C接口Tk库，因此不为脚本编写者特别有用。
- Addison-Wesley出版了一本名为Tcl的书和John Ousterhout的Tk Toolkit（ISBN 0-201-63337-X），这是对新手Tcl和Tk的一个很好的介绍。这本书并非详尽无遗，对于许多细节，它都遵循手册页。
- tkinter/__init__.py对大多数人来说是最后的选择，但如果没有其他任何意义的话，它可能是一个很好的去处。

25.1.2.2。简单的Hello World程序

```
import tkinter as tk

class Application(tk.Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.pack()
        self.create_widgets()

    def create_widgets(self):
        self.hi_there = tk.Button(self)
        self.hi_there["text"] = "Hello World\n(click me)"
        self.hi_there["command"] = self.say_hi
        self.hi_there.pack(side="top")
```

```
        self.quit = tk.Button(self, text="QUIT", fg="red",
                              command=root.destroy)
        self.quit.pack(side="bottom")

    def say_hi(self):
        print("hi there, everyone!")

root = tk.Tk()
app = Application(master=root)
app.mainloop()
```

25.1.3。A（非常）快速浏览Tcl /

类层次结构看起来很复杂，但实际上，应用程序员几乎总是引用层次结构底部的类。

笔记：

- 提供这些类是为了在一个命名空间下组织某些功能。它们不是要独立实例化的。
- 该Tk班，就是要在应用程序中只有一次实例化。应用程序员不需要明确地实例化一个实例，只要有任何其他类实例化，系统就会创建一个。
- 这个Widget类不意味着被实例化，它只是意味着子类化才能生成“真实”的小部件（在C++中，这被称为“抽象类”）。

为了使用这个参考资料，有时候您需要知道如何阅读Tk的简短段落以及如何识别Tk命令的各个部分。（请参阅将Tk的基本Tk映射到Tkinter中， tkinter以了解下面的内容。）

Tk脚本是Tcl程序。像所有的Tcl程序一样，Tk脚本只是由空格分隔的令牌列表。传统知识小部件就是它的*类*的*选项*，帮助配置，和*行动*，使人们做有用的事情。

要在Tk中创建一个小部件，该命令始终是以下形式：

```
classCommand newPathname options
```

classCommand
: 表示要制作哪种小部件（按钮，标签，菜单...）

newPathname
: 是这个小部件的新名称。Tk中的所有名称都必须是唯一的。为了实现这一点，Tk中的小部件以*路径名*命名，就像文件系统中的文件一样。顶级小部件*root*称为.（句号），子句由更多句点分隔。例如，.myApp.controlPanel.okButton可能是小部件的名称。

选项
: 配置小部件的外观，并在某些情况下，它的行为。选项以标志和值列表的形式出现。标志前面有一个'-'，就像Unix shell命令标志一样，并且如果值超过一个单词，则会将值放在引号中。

例如：

```
button   .fred   -fg red -text "hi there"
   ^       ^     \_____________________/
   |       |                |
```

```
 class    new            options
command  widget  (-opt val -opt val ...)
```

一旦创建，小部件的路径名变成新命令。这个新的 *Widget命令*是程序员获取新Widget执行某些*操作*的句柄。在C语言中，你可以用C ++表示它为someAction（fred，someOptions），你可以将它表示为fred.someAction（someOptions），在Tk中，你会说：

```
.fred someAction someOptions
```

请注意，对象名称.fred以点开头。

正如你所期望的那样，*someAction*的合法值将取决于widget的类：如果fred是一个按钮（fred会变灰），但如果fred是标签则不起作用（Tk不支持禁用标签）。.fred disable

*someOptions*的合法价值取决于行动。某些操作（如 disable，不需要参数），其他操作（如文本输入框的delete 命令）需要参数来指定要删除的文本范围。

25.1.4。基本映射传统知识纳入Tkinter的

Tk中的类命令对应于Tkinter中的类构造函数。

```
button .fred                =====>  fred = Button()
```

对象的主对象隐含在创建时给它的新名称中。在Tkinter中，明确指定了主人。

```
button .panel.fred          =====>  fred = Button(panel)
```

Tk中的配置选项在带有数值的紧跟标签列表中给出。在Tkinter中，选项在实例构造函数中被指定为关键字参数，在配置调用中被指定为keyword-args，或者在字典样式中被指定为已建立实例的实例索引。请参见 设置选项中的设置选项部分。

```
button .fred -fg red        =====>  fred = Button(panel, fg="red")
.fred configure -fg red     =====>  fred["fg"] = red
                            OR ==>  fred.config(fg="red")
```

在Tk中，要在小部件上执行操作，请使用小部件名称作为命令，然后使用操作名称，可能带有参数（选项）。在Tkinter中，您可以调用类实例上的方法来调用窗口小部件上的操作。给定小部件可以执行的动作（方法）列在中 tkinter/__init__.py。

```
.fred invoke                =====>  fred.invoke()
```

要给封装器（几何管理器）提供一个小部件，可以使用可选参数调用封装。在Tkinter中，Pack类拥有所有这些功能，并且各种形式的pack命令都作为方法实现。所有的小部件 tkinter都是从Packer中分类出来的，因此继承了所有的打包方法。有关tkinter.tix表格几何管理器的其他信息，请参阅模块文档。

```
pack .fred -side left       =====>  fred.pack(side="left")
```

25.1.5。Tk和Tkinter如何相关

从上到下：

你的应用在这里（Python）
: 一个Python应用程序发出`tkinter`呼叫。

tkinter（Python包）
: 这个调用（例如，创建一个按钮小部件）是在`tkinter`用Python编写的包中实现的。这个Python函数将解析命令和参数，并将它们转换为一种形式，使它们看起来好像它们来自Tk脚本而不是Python脚本。

_tkinter（C）
: 这些命令及其参数将传递给C函数 `_tkinter`- 注意下划线 - 扩展模块。

Tk Widgets（C和Tcl）
: 这个C函数可以调用其他C模块，包括组成Tk库的C函数。Tk在C和一些Tcl中实现。Tk小部件的Tcl部分用于将某些默认行为绑定到小部件，并在`tkinter` 导入Python 包的位置执行一次。（用户从来没有看到这个阶段）。

Tk（C）
: Tk Widgets的Tk部分实现了最终的映射.....

Xlib（C）
: Xlib库在屏幕上绘制图形。

25.1.6。方便的参考

25.1.6.1。设置选项

选项控制一些东西，比如一个小部件的颜色和边框宽度。选项可以通过三种方式设置：

在创建对象时，使用关键字参数

```
fred = Button(self, fg="red", bg="blue")
```

创建对象后，将选项名称视为字典索引

```
fred["fg"] = "red"
fred["bg"] = "blue"
```

使用config（）方法更新对象创建后的多个attrs

```
fred.config(fg="red", bg="blue")
```

有关给定选项及其行为的完整说明，请参阅相关Widget的Tk手册页。

请注意，手册页列出了每个窗口小部件的“标准选项”和“小部件特定选项”。前者是许多小部件通用的选项列表，后者是特定小部件特有的选项。标准选项记录在*选项（3）*手册页上。

本文档没有区分标准和特定于窗口小部件的选项。有些选项不适用于某些类型的小部件。给定小部件是否响应特定选项取决于小部件的类别; 按钮有一个command选项，标签没有。

给定小部件支持的选项列在该小部件的手册页中，或者可以在运行时通过调用config()没有参数的方法或通过调用该keys()小部件上的方法来查询。这些调用的返回值是一个字典，其中的键是作为字符串（例如，'relief'）的选项的名称，其值是5元组。

一些选项，如bg长名称的常见选项的同义词（bg是“背景”的简写）。传递config()方法时，速记选项的名称将返回2元组，而不是5元组。传回的2元组将包含同义词的名称和“真实”选项（如）。('bg', 'background')

指数	含义	例
0	选项名称	'relief'
1	数据库查找的选项名称	'relief'
2	选项类用于数据库查找	'Relief'
3	默认值	'raised'
4	当前值	'groove'

例：

```
>>> print(fred.config())
{'relief': ('relief', 'relief', 'Relief', 'raised', 'groove')}
```

当然，打印的字典将包括所有可用选项及其值。这只是作为一个例子。

25.1.6.2。包装工

封隔器是Tk的几何管理机制之一。几何管理器用于指定小部件在其容器中的定位的相对位置 - 它们是相互的*主人*。相较于较为繁琐*砂矿*（用于不常用，我们在这里不介绍），封隔器采用定性关系规范- *以上*，*以左*，*填充*，等等-和工作的一切，以确定确切位置坐标为你。

任何*去*控件的大小由内部“从控件”的大小决定。打包器用于控制从站窗口小部件在其打包的主站内出现的位置。您可以将窗口小部件封装到框架中，并将框架封装到其他框架中，以实现您所需的布局。此外，这种安排是动态调整的，以适应配置的增量变化，一旦打包完成。

请注意，窗口小部件在用几何管理器指定几何图形之前不会显示。忽略几何规格是一个常见的早期错误，然后在创建小部件时会感到惊讶，但没有任何内容出现。例如，小部件只有在pack()应用打包器的方法后才会出现 。

可以使用关键字选项/值对调用pack（ ）方法，该对控制了窗口小部件出现在其容器中的位置，以及在主应用程序窗口调整大小时的行为方式。这里有些例子：

```
fred.pack()                     # defaults to side = "top"
fred.pack(side="left")
fred.pack(expand=1)
```

25.1.6.3。封装选项

有关打包机的更多信息及其可以选择的选项，请参阅John Ousterhout书中的手册页和第183页。

锚
: 锚点类型。表示包装员将每个奴隶放入包裹的位置。

扩大
: 布尔值0或1。

填
: 合法值：'x'，'y'，'both'，'none'。

ipadx和ipady
: 距离 - 指定从属部件每边的内部填充。

padx和pady
: 距离 - 指定从属小部件每侧的外部填充。

侧
: 合法的值是：'left'，'right'，'top'，'bottom'。

25.1.6.4。耦合小部件变量

某些小部件（如文本输入小部件）的当前值设置可以通过使用特殊选项直接连接到应用程序变量。这些选项包括variable，textvariable，onvalue，offvalue，和 value。这种连接可以两种方式工作：如果变量因任何原因而变化，它所连接的小部件将被更新以反映新值。

不幸的是，在当前的实现中tkinter，不可能通过variable或textvariable选项将任意Python变量移交给小部件。唯一的种变量此作品是由一个叫做类变量，在子类中定义的变量tkinter。

目前已经定义的变量的许多有用的子类： StringVar，IntVar，DoubleVar，和 BooleanVar。要读取此变量的当前值，请调用其get()上的 方法，并更改其调用该set() 方法的值。如果你遵循这个协议，那么这个小部件将总是跟踪这个变量的值，而不需要你做进一步的干预。

例如：

```
class App(Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.pack()

        self.entrythingy = Entry()
        self.entrythingy.pack()

        # here is the application variable
        self.contents = StringVar()
        # set it to some value
        self.contents.set("this is a variable")
        # tell the entry widget to watch this variable
        self.entrythingy["textvariable"] = self.contents

        # and here we get a callback when the user hits return.
        # we will have the program print out the value of the
        # application variable when the user hits return
```

```
        self.entrythingy.bind('<Key-Return>',
                              self.print_contents)

    def print_contents(self, event):
        print("hi. contents of entry is now ---->",
              self.contents.get())
```

25.1.6.5。窗口管理器

在Tk中，有一个实用程序命令wm，用于与窗口管理器进行交互。该wm命令的选项允许您控制诸如标题，位置，图标位图等之类的内容。在中tkinter，这些命令已经作为Wm类中的方法实现。Toplevel小部件从Wm类中继承，因此可以Wm直接调用这些方法。

要查看包含给定窗口小部件的顶层窗口，通常可以引用窗口小部件的主窗口。当然，如果这个小部件已经被封装在一个框架内，那么这个主框架将不会代表一个顶层窗口。要访问包含任意小部件的顶级窗口，可以调用该_root()方法。该方法以下划线开头，表示这个函数是实现的一部分，而不是Tk功能的接口。

以下是一些典型用法的例子：

```
import tkinter as tk

class App(tk.Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.pack()

# create the application
myapp = App()

#
# here are method calls to the window manager class
#
myapp.master.title("My Do-Nothing Application")
myapp.master.maxsize(1000, 400)

# start the program
myapp.mainloop()
```

25.1.6.6。Tk选项数据类型

锚

合法的值是罗盘上的点："n"，"ne"，"e"，"se"，"s"，"sw"，"w"，"nw"，和也"center"。

位图

有八个内置名为位图：'error'，'gray25'，'gray50'，'hourglass'，'info'，'questhead'，'question'，'warning'。要指定一个X位图文件名，给出该文件的完整路径，前面带有一个@，如"@/usr/contrib/bitmap/gumby.bit"。

布尔

您可以传递整数0或1或字符串"yes"或"no"。

回电话
这是任何不带参数的Python函数。例如：

```
def print_it():
    print("hi there")
fred["command"] = print_it
```

颜色
可以给出颜色作为rgb.txt文件中X颜色的名称，或以4位"#RGB"，8位"#RRGGBB"，12位"#RRRGGGBBB"或16位"#RRRRGGGGBBBB"范围代表RGB值的字符串，其中R，G，B代表任何合法十六进制数字 详情请参阅Ousterhout书籍的第160页。

光标
cursorfont.h可以使用标准X游标名称，而不使用 XC_前缀。例如，要获得手形光标（ XC_hand2 ），请使用字符串"hand2"。你也可以指定你自己的位图和掩码文件。参见Ousterhout书中的第179页。

距离
屏幕距离可以在像素或绝对距离中指定。像素以数字和绝对距离作为字符串给出，尾随字符表示单位：c厘米，i英寸，m毫米，p打印点。例如，3.5英寸表示为"3.5i"。

字形
Tk使用列表字体名称格式，例如。正数的字体大小以点为单位; 负数的尺寸以像素为单位进行测量。{courier 10 bold}

几何
这是一种形式的字符串widthxheight，其中宽度和高度以大多数小部件的像素为单位进行度量（以小部件显示文本的字符为单位）。例如：。fred["geometry"] = "200x100"

辩解
合法的值是字符串："left"，"center"，"right"，和 "fill"。

地区
这是一个由四个空格分隔的元素构成的字符串，每个元素都是合法的距离（参见上文）。例如：and 和 都是合法区域。"2 3 4 5""3i 2i 4.5i 2i""3c 2c 4c 10.43c"

浮雕
确定小部件的边框样式。合法的值是： "raised"，"sunken"，"flat"，"groove"，和"ridge"。

scrollcommand
这几乎总是set()一些滚动条小部件的方法，但可以是任何采用单个参数的小部件方法。

包：
必须是一个："none"，"char"或"word"。

25.1.6.7。绑定和事件

通过widget命令的绑定方法，您可以监视某些事件并在发生该事件类型时触发回调函数。绑定方法的形式是：

```
def bind(self, sequence, func, add=''):
```

哪里：

序列
: 是一个表示目标事件类型的字符串。（有关详细信息，请参阅John Ousterhout书中的约束手册页和第201页）。

FUNC
: 是一个Python函数，带有一个参数，在事件发生时被调用。一个Event实例将作为参数传递。（以这种方式部署的函数通常称为*回调*函数。）

加
: 是可选的，''或者'+'。传递一个空字符串表示这个绑定是要替换这个事件关联的任何其他绑定。传递'+'意味着此函数将被添加到绑定到此事件类型的函数列表中。

例如：

```
def turn_red(self, event):
    event.widget["activeforeground"] = "red"

self.button.bind("<Enter>", self.turn_red)
```

请注意事件的窗口小部件字段在`turn_red()`回调中如何被访问 。该字段包含捕获X事件的小部件。下表列出了您可以访问的其他事件字段以及它们在Tk中的表示方式，这在引用Tk手册页时非常有用。

TK	Tkinter事件字段	TK	Tkinter事件字段
%f	焦点	%s	烧焦
%h	高度	%t	send_event
%k	关键代码	%A	键符
%s	州	%N	keysym_num
%t	时间	%T	类型
%w	宽度	%W	小部件
%x	x	%X	x_root
%y	y	%Y	y_root

25.1.6.8。索引参数

许多小部件需要传递“索引”参数。它们用于指向Text小部件中的特定位置，或指向Entry小部件中的特定字符或指向Menu小部件中的特定菜单项。

条目小部件索引（索引，视图索引等）
: 条目窗口小部件具有涉及正在显示的文本中的字符位置的选项。您可以使用这些`tkinter`函数来访问文本小部件中的这些特殊点：

文本小部件索引
: Text小部件的索引符号非常丰富，最好在Tk手册页中进行描述。

菜单索引（menu.invoke（），menu.entryconfig（）等）
: 一些菜单选项和方法可以操作特定的菜单条目。无论何时，对于选项或参数都需要菜单索引，您可以通过：

- 一个整数，它指向窗口小部件中的条目的数字位置，从顶部开始计数，从0开始;

- 字符串"active"，它是指当前在光标下的菜单位置;
- "last"引用最后一个菜单项的字符串;
- 通过前面的整数@，如在@6，其中整数被解释为y像素菜单的坐标系统中的坐标;
- 该字符串"none"表示根本没有菜单条目，通常与menu.activate（ ）一起使用来停用所有条目，最后，
- 一个与菜单条目标签相匹配的文本字符串，从菜单顶部扫描到底部。请注意，此索引类型的所有其它后认为，这意味着对于菜单项的匹配标记的last，active或者none可以被解释为上述的文字，而不是。

25.1.6.9。图像

可以通过相应的子类创建不同格式的图像tkinter.Image：

- BitmapImage 用于XBM格式的图像。
- PhotoImage用于PGM，PPM，GIF和PNG格式的图像。后者支持从Tk 8.6开始。

任何类型的图像都是通过file或data 选项创建的（其他选项也可用）。

这个图像对象可以用于任何一个image小部件支持的选项（例如标签，按钮，菜单）。在这些情况下，Tk将不会保留对图像的引用。当删除对图像对象的最后一个Python引用时，图像数据也会被删除，并且无论图像在哪里使用，Tk都会显示一个空框。

也可以看看: 该枕头包增加了对格式，如BMP，JPEG，TIFF和WebP的，等等的支持。

25.1.7。文件处理程序

Tk允许你注册和取消注册一个回调函数，当一个文件描述符可能有I / O时，这个回调函数将从Tk mainloop调用。每个文件描述符只能注册一个处理程序。示例代码：

```
import tkinter
widget = tkinter.Tk()
mask = tkinter.READABLE | tkinter.WRITABLE
widget.tk.createfilehandler(file, mask, callback)
...
widget.tk.deletefilehandler(file)
```

此功能在Windows上不可用。

由于您不知道有多少字节可供读取，因此您可能不想使用BufferedIOBaseor 或方法，因为这些字节会坚持读取预定义的字节数。对于套接字，or 方法可以正常工作; 对于其他文件，请使用原始读取或。 TextIOBase read() readline() recv() recvfrom() os.read(file.fileno(), maxbytecount)

Widget.tk.createfilehandler（*file*，*mask*，*func* ）

注册文件处理程序回调函数*func*。该*文件*的参数可以是与一个对象fileno()的方法（例如，文件或套接字对象），或一个整数文件描述符。的*掩模*参数是任何以下三个常量的或运算组合。回调被调用如下：

```
callback(file, mask)
```

Widget.tk.deletefilehandler(*文件*)

取消注册文件处理程序。

tkinter.READABLE
tkinter.WRITABLE
tkinter.EXCEPTION

*掩码*参数中使用的常量。

25.2。 tkinter.ttk- Tk主题小部件

源代码： Lib / tkinter / ttk.py

该tkinter.ttk模块提供对Tk 8.5中引入的Tk主题小部件集的访问。如果Python尚未针对Tk 8.5进行编译，则在安装了*Tile的情况下*仍然可以访问该模块。前一种使用Tk 8.5的方法提供了额外的好处，包括X11下的消除锯齿字体渲染和窗口透明度（需要X11上的合成窗口管理器）。

其基本思想tkinter.ttk是尽可能将实现窗口小部件行为的代码与实现其外观的代码分离开来。

也可以看看:

Tk Widget造型支持
介绍主题支持Tk的文档

25.2.1。使用TTK

要开始使用Ttk，请导入其模块：

```
from tkinter import ttk
```

要覆盖基本的Tk小部件，导入应该遵循Tk导入：

```
from tkinter import *
from tkinter.ttk import *
```

该代码将导致几个tkinter.ttk小部件（ Button ， Checkbutton ， Entry ， Frame ， Label ， LabelFrame ，Menubutton ，PanedWindow ， Radiobutton ，Scale和Scrollbar ）来自动更换Tk部件。

这有直接的好处，可以使用新的小部件，从而提供跨平台更好的外观和感觉; 但是，替换小部件并不完全兼容。主要区别在于，小部件选项（例如“fg”，“bg”和与小部件样式相关的其他小部件选项不再存在于Ttk小部件中。相反，使用这个ttk.Style类来改进样式效果。

也可以看看:

转换现有应用程序以使用Tile小部件
一本关于在移动应用程序以使用新窗口小部件时遇到的差异的专着（使用Tcl术语）。

25.2.2。TTK窗口小部件

TTK配备了17只小部件，其中十一个在Tkinter的已经存在： Button ，Checkbutton ，Entry ， Frame ， Label ，LabelFrame ，Menubutton ，PanedWindow ， Radiobutton ，Scale和Scrollbar。

其他六个都是新的Combobox，Notebook，Progressbar， Separator，Sizegrip和Treeview。而它们都是子类Widget。

使用Ttk小部件为应用程序提供了改进的外观和感觉。如上所述，样式编码方式存在差异。

Tk代码：

```
l1 = tkinter.Label(text="Test", fg="black", bg="white")
l2 = tkinter.Label(text="Test", fg="black", bg="white")
```

Ttk代码：

```
style = ttk.Style()
style.configure("BW.TLabel", foreground="black", background="white")

l1 = ttk.Label(text="Test", style="BW.TLabel")
l2 = ttk.Label(text="Test", style="BW.TLabel")
```

有关TtkStyling的更多信息，请参阅Style类文档。

25.2.3。窗口小部件

ttk.Widget 定义Tk主题小部件支持的标准选项和方法，不应该直接实例化。

25.2.3.1。标准选项

所有ttk小组件均接受以下选项：

选项	描述
类	指定窗口类。该类用于查询窗口其他选项的选项数据库，确定窗口的默认绑定标签并选择窗口小部件的默认布局和样式。该选项是只读的，只能在窗口创建时指定。
光标	指定要用于小部件的鼠标光标。如果设置为空字符串（默认值），则光标将继承父窗口小部件。
takefocus	确定在键盘遍历期间窗口是否接受焦点。0,1或返回一个空字符串。如果返回0，则意味着在键盘遍历期间窗口应该被完全跳过。如果为1，则意味着只要窗口可见，该窗口就会收到输入焦点。而一个空字符串意味着遍历脚本决定是否关注窗口。
样式	可用于指定自定义小部件样式。

25.2.3.2。可滚动的小部件选项

由滚动条控制的小部件支持以下选项。

选项	描述

选项	描述
xscrollcommand	用于与水平滚动条进行通信。 当小部件窗口中的视图改变时，小部件将根据滚动命令生成一个Tcl命令。 通常这个选项由Scrollbar.set()一些滚动条的方法组成 。这将导致滚动条在窗口中的视图改变时被更新。
yscrollcommand	用于与垂直滚动条进行通信。有关更多信息，请参阅上文。

25.2.3.3。标签选项

标签，按钮和其他类似按钮的小部件支持以下选项。

选项	描述
文本	指定要在小部件内显示的文本字符串。
textvariable	指定一个名称，其值将用于代替文本选项资源。
强调	如果设置，则指定字符的索引（从0开始）以在文本字符串中加下划线。下划线字符用于助记符激活。
图片	指定要显示的图像。这是一个或多个元素的列表。第一个元素是默认图像名称。列表的其余部分是否定义了一系列statespec / value对Style.map()，指定不同的图像以便在控件处于特定状态或状态组合时使用。列表中的所有图像应具有相同的大小。
复合	指定在文本和图像选项都存在的情况下如何显示相对于文本的图像。有效值是： • 文本：仅显示文本 • 图像：仅显示图像 • 顶部，底部，左侧，右侧：分别显示图像上方，下方，左侧或右侧的图像。 • none：默认值。显示图像如果存在，否则显示文本。
宽度	如果大于零，则指定为文本标签分配多少字符宽度，如果小于零，则指定最小宽度。如果为零或未指定，则使用文本标签的自然宽度。

25.2.3.4。兼容性选项

选项	描述
州	可以设置为“正常”或“禁用”来控制“禁用”状态位。这是一个只写选项：设置它会更改小部件状态，但该Widget.state()方法不会影响此选项。

25.2.3.5。Widget国家

小部件状态是独立状态标志的位图。

旗	描述
活性	鼠标光标位于窗口小部件上并按下鼠标按钮将导致一些操作发生
残	在程序控制下，Widget被禁用
焦点	小工具有键盘焦点
压制	小工具被按下
选	“开”，“真”或“当前”，例如Checkbuttons和单选按钮
背景	Windows和Mac有一个“活动”或前景窗口的概念。该*背景*状态设置为背景的窗口小部件，并清除了那些在前台窗口
只读	小部件不应允许用户修改
备用	特定于窗口小部件的替代显示格式
无效	小部件的值是无效的

状态规范是一系列状态名称，可选地以感叹号作为前缀，指示该位关闭。

25.2.3.6。ttk.Widget

除了下面介绍的方法外，还ttk.Widget支持方法tkinter.Widget.cget()和tkinter.Widget.configure()。

*类*tkinter.ttk.Widget

identify（*x*，*y*）

返回位于所述元素的名称*X ÿ*，或空字符串，如果点不任何元件之内。

*x*和*y*是相对于小部件的像素坐标。

instate（*statespec*，*callback = None*，* *args*，** *kw* ）

测试小部件的状态。如果未指定回调，则返回True小部件状态是否与*statespec*匹配，False否则返回。如果指定了回调，那么如果小部件状态匹配*statespec，则会*使用args调用回调 。

state（*statespec = None* ）

修改或查询小部件状态。如果指定了*statespec，*则根据它设置小部件状态并返回一个新的*statespec，*指出哪些标志已更改。如果没有指定*statespec*，则返回当前启用的状态标志。

*statespec*通常是一个列表或元组。

25.2.4。组合框

该ttk.Combobox小部件将文本字段与值的下拉列表组合在一起。这个小部件是的一个子类Entry。

除了从继承的方法Widget：Widget.cget()，Widget.configure()，Widget.identify()，Widget.instate()和Widget.state()，然后从以下继承Entry：Entry.bbox()，Entry.delete()，Entry.icursor()，Entry.index()，Entry.insert()，Entry.selection()，Entry.xview()，它有一些其它方法中，在所描述ttk.Combobox。

25.2.4.1。选项

该小部件接受以下特定选项：

选项	描述
exportselection	布尔值。如果设置，窗口部件选择链接到窗口管理器选择（例如，可以通过调用Misc.selection_get返回）。
辩解	指定文本在窗口小部件中的排列方式。“左”，“中”或“右”之一。
高度	以行为单位指定弹出式列表框的高度。
postcommand	在显示值之前立即调用的脚本（可能注册了Misc.register）。它可以指定显示哪些值。
州	“正常”，“只读”或“禁用”之一。在“只读”状态下，该值可能不会被直接编辑，用户只能从下拉列表中选择值。在“正常”状态下，文本字段可直接编辑。在“禁用”状态下，不可能进行交互。
textvariable	指定一个名称，其值与小部件值链接。每当与该名称关联的值发生更改时，窗口小部件值就会更新，反之亦然。看tkinter.StringVar。
值	指定要在下拉列表框中显示的值列表。
宽度	以小部件字体的平均大小字符指定指示输入窗口的所需宽度的整数值。

25.2.4.2。虚拟事件

当用户从值列表中选择一个元素时，组合框部件会生成一个**<< ComboboxSelected >>**虚拟事件。

25.2.4.3。ttk.Combobox

*类*tkinter.ttk.Combobox

current（*newindex =无*）

如果指定了*newindex*，则将combobox值设置为元素位置*newindex*。否则，返回当前值的索引或-1（如果当前值不在值列表中）。

get（）

返回组合框的当前值。

set（*价值*）

将组合框的值设置为*值*。

25.2.5。笔记本

Ttk Notebook小部件管理一组窗口并一次显示一个窗口。每个子窗口都与一个选项卡关联，用户可以选择该选项卡来更改当前显示的窗口。

25.2.5.1。选项

该小部件接受以下特定选项：

选项	描述
高度	如果存在且大于零，则指定窗格区域的所需高度（不包括内部填充或制表符）。否则，将使用所有窗格的最大高度。
填充	指定要在笔记本外围添加的额外空间量。填充是最多四个长度规格的列表，左上方右上方。如果指定的元素少于四个，则底部默认为顶部，右侧默认为左侧，顶部默认为左侧。
宽度	如果存在且大于零，则指定窗格区域的所需宽度（不包括内部填充）。否则，将使用所有窗格的最大宽度。

25.2.5.2。标签选项

还有标签的特定选项：

选项	描述
州	“正常”，“禁用”或“隐藏”。如果“禁用”，则选项卡不可选。如果“隐藏”，则标签不显示。
黏	指定子窗口在窗格区域内的位置。值是一个包含零个或多个字符“n”，“s”，“e”或“w”的字符串。根据`grid()`几何管理器，每个字母指的是子窗口将粘贴的一侧（北，南，东或西）。
填充	指定笔记本和此窗格之间要添加的额外空间量。语法与此小部件使用的选项填充相同。
文本	指定要在选项卡中显示的文本。
图片	指定要在选项卡中显示的图像。请参阅中所述的选项图像`Widget`。
复合	指定如何显示相对于文本的图像，在存在选项文本和图像的情况下。有关合法值，请参阅 标签选项。
强调	指定要在文本字符串中下划线的字符的索引（从0开始）。带下划线的字符在`Notebook.enable_traversal()`被调用时用于助记符激活。

25.2.5.3。标签标识符

存在于几种方法中的tab_id `ttk.Notebook`可以采取以下任何形式：

- 介于零和选项卡数之间的整数
- 子窗口的名称
- 表单“@ x，y”的位置规范，用于标识选项卡
- 字符串“current”，标识当前选择的选项卡
- 文字字符串“结束”，它返回选项卡的数量（仅适用于 Notebook.index() ）

25.2.5.4。虚拟活动

这个小部件在选择新选项卡后生成一个**<< NotebookTabChanged >>**虚拟事件。

25.2.5.5。ttk.Notebook

*类*tkinter.ttk.Notebook

add (*孩子*，***千瓦*)

向笔记本添加一个新选项卡。

如果窗口当前由笔记本管理但隐藏，则会恢复到之前的位置。

请参阅选项卡选项以获取可用选项的列表。

forget (*tab_id*)

删除由*tab_id*指定的选项卡，取消*映射*和取消管理关联的窗口。

hide (*tab_id*)

隐藏由*tab_id*指定的选项卡。

该选项卡不会显示，但关联的窗口仍由笔记本管理，并记住其配置。隐藏的选项卡可以使用该add()命令进行恢复。

identify (*x* , *y*)

返回位置*x*，*y*处的制表符元素的名称，如果没有，则返回空字符串。

index (*tab_id*)

返回由*tab_id*指定的选项卡的数字索引，如果*tab_id*是字符串“end”，则*返回*选项卡总数。

insert (*pos* , *child* , ** *kw*)

在指定的位置插入窗格。

*pos*是字符串“结束”，一个整数索引或管理的孩子的名字。如果*孩子*已经被笔记本管理，将其移动到指定的位置。

请参阅选项卡选项以获取可用选项的列表。

select (*tab_id = None*)

选择指定的*tab_id*。

关联的子窗口将被显示，并且之前选择的窗口（如果不同）未被映射。如果省略*tab_id*，则返回当前选定窗格的窗口小部件名称。

`tab`（*tab_id*，*option = None*，*** kw* ）

查询或修改特定*tab_id*的选项。

如果没有给出*kw*，则返回标签选项值的字典。如果 指定了*选项*，则返回该*选项*的值。否则，将选项设置为相应的值。

`tabs`（ ）

返回笔记本管理的窗口列表。

`enable_traversal`（ ）

为包含此笔记本的顶层窗口启用键盘遍历。

这将扩展包含笔记本的顶级窗口的绑定，如下所示：

- `Control-Tab`：选择当前选择的选项卡之后的选项卡。
- `Shift-Control-Tab`：选择当前选定的标签之前的标签。
- `Alt-K`：其中*K*是任何选项卡的助记符（带下划线）的字符，将选择该选项卡。

可以启用单个顶层中的多个笔记本进行遍历，包括嵌套笔记本。但是，笔记本遍历只有在所有窗格都有作为主人的笔记本时才能正常工作。

25.2.6。进度条

该`ttk.Progressbar`小部件显示长时间运行的状态。它可以以两种模式操作：1）确定模式，显示相对于要完成的工作总量完成的量;以及2）提供动画显示以使用户知道工作正在进行的不确定模式。

25.2.6.1。选项

该小部件接受以下特定选项：

选项	描述
东方	“水平”或“垂直”之一。指定进度条的方向。
长度	指定进度条的长轴的长度（如果是水平则为宽度，如果为垂直则为高度）。
模式	“确定性”或“不确定性”之一。
最大	指定最大值的数字。默认为100。
值	进度条的当前值。在“确定”模式下，这表示完成的工作量。在“不确定”模式下，它被解释为模*最大值*; 也就是说，进度条在其值*最大*增加时完成一个“循环” 。
变量	与选项值链接的名称。如果指定，则修改后者时，进度栏的值会自动设置为该名称的值。

选项	描述
相	只读选项。只要该值大于0并且在确定模式下小于最大值，该部件就会周期性地增加此选项的值。该选项可以被当前主题使用以提供额外的动画效果。

25.2.6.2。ttk.Progressbar

*类*tkinter.ttk.Progressbar

start（*interval = None* ）

开始自动增量模式：计划一个重复计时器事件，该事件调用 Progressbar.step() 每个*间隔*毫秒。如果省略，则 *时间间隔*默认为50毫秒。

step（*金额=无*）

按*数量*增加进度条的值。

*金额*默认为1.0，如果省略。

stop（ ）

停止自动增量模式：取消Progressbar.start()此进度条启动的任何循环计时器事件 。

25.2.7。分隔符

该ttk.Separator小部件显示水平或垂直分隔栏。

除了继承的方法之外，没有其他方法ttk.Widget。

25.2.7.1。选项

该小部件接受以下特定选项：

选项	描述
东方	“水平”或“垂直”之一。指定分隔符的方向。

25.2.8。Sizegrip

的ttk.Sizegrip微件（也称为成长盒）允许用户通过按下并拖动把手来调整含顶层窗口。

这个小部件既没有特定的选项，也没有特定的方法，除了从中继承ttk.Widget。

25.2.8.1。平台特定的注释

- 在MacOS X上，顶层窗口默认会自动包含一个内置的大小夹点。添加一个Sizegrip是无害的，因为内置的抓地力只会掩盖小部件。

25.2.8.2。错误

- 如果包含顶层的位置是相对于屏幕的右侧或底部（例如...）指定的，则Sizegrip窗口小部件将不调整窗口大小。
- 此小部件仅支持“东南”调整大小。

25.2.9。树形

该ttk.Treeview小部件显示项目的分层集合。每个项目都有一个文本标签，一个可选图像和一个可选的数据值列表。数据值在树标签之后的连续列中显示。

数据值的显示顺序可以通过设置widget选项来控制displaycolumns。树小部件也可以显示列标题。列可以通过小部件选项列中列出的数字或符号名称访问。请参阅列标识符。

每个项目都由一个唯一的名称标识。如果它们不是由调用者提供的，该小部件将生成项目ID。有一个杰出的根项目，命名{}。根项目本身不显示; 其子女出现在层次结构的顶层。

每个项目还有一个标签列表，可用于将事件绑定与单个项目相关联，并控制项目的外观。

Treeview小部件支持水平和垂直滚动，根据“可滚动小部件选项”中介绍的选项以及方法Treeview.xview()和Treeview.yview()。

25.2.9.1。选项

该小部件接受以下特定选项：

选项	描述
列	列标识符列表，指定列数及其名称。
displaycolumns	一列列标识符（符号或整数索引），用于指定显示哪些数据列及其出现顺序或字符串“#all”。
高度	指定应该可见的行数。注意：请求的宽度由列宽的总和确定。
填充	指定小部件的内部填充。填充是最多四个长度规格的列表。
选择模式	控制内置类绑定如何管理选择。“扩展”，“浏览”或“无”之一。如果设置为“扩展”（默认），则可以选择多个项目。如果“浏览”，则一次只能选择一个项目。如果“无”，选择将不会改变。 请注意，无论此选项的值如何，应用程序代码和标签绑定都可以设置所需的选择。

选项	描述
显示	包含零个或多个以下值的列表，指定要显示的树的哪些元素。 • 树：显示列＃0中的树标签。 • 标题：显示标题行。 默认是“树状标题”，即显示所有元素。 **注意**：即使未指定show =“tree”，列＃0始终指向树列。

25.2.9.2。项目选项

可以为插入和项目小部件命令中的项目指定以下项目选项。

选项	描述
文本	要为项目显示的文本标签。
图片	Tk图像，显示在标签的左侧。
值	与该项目关联的值列表。 每个项目都应该具有与小部件选项列相同数量的值。如果数值少于列数，则其余数值假定为空。如果有多个列的值，则会忽略额外的值。
打开	真/假值指示项目的孩子是否应该显示或隐藏。
标签	与此商品相关的标签列表。

25.2.9.3。标签选项

可以在标签上指定以下选项：

选项	描述
前景	指定文本前景色。
背景	指定单元格或项目背景颜色。
字形	指定绘制文本时使用的字体。
图片	指定项目图像，以防项目的图像选项为空。

25.2.9.4。列标识符

列标识符采用以下任何一种形式：

- 列列表选项中的符号名称。
- 一个整数n，指定第n个数据列。
- 格式为#n的字符串，其中n是一个整数，指定第n个显示列。

笔记：

- 商品的选项值可能以不同于其存储顺序的顺序显示。
- 即使未指定show =“tree”，列＃0始终指向树列。

数据列号是项目选项值列表中的索引; 显示列号是树中显示值的列号。树标签显示在＃0列中。如果未设置选项显示列，则数据列n显示在列＃n + 1中。同样，**第＃0列总是指向树列**。

25.2.9.5。虚拟活动

Treeview小部件生成以下虚拟事件。

事件	描述
<< TreeviewSelect >>	每当选择更改时生成。
<< TreeviewOpen >>	在设置要打开的焦点项目之前生成= True。
<< TreeviewClose >>	在将焦点项目设置为打开= False之后生成。

该Treeview.focus()和Treeview.selection()方法可用于确定受影响的项目或项目。

25.2.9.6。ttk.Treeview

*类*tkinter.ttk.Treeview

bbox（*item*，*column* = *None* ）

以（x，y，width，height）形式返回指定*项目*的边界框（相对于树视图窗口小部件的窗口）。

如果指定了*列*，则返回该单元格的边界框。如果该 *项目*不可见（即，如果它是封闭项目的后代或滚动离屏），则返回空字符串。

get_children（*item* = *None* ）

返回属于*项目*的子*项*列表。

如果未指定*项目*，则返回根子项。

set_children（*item*，* *newchildren* ）

用*新生儿*替换*项目*的孩子。

在*项目*中出现的不在*新生儿身上的*儿童从树上分离。在没有项目*newchildren*可能的祖先 *项目*。请注意，不指定*newchildren*会导致分离 *项目*的子项。

column（*列*，*选项*= *None*，** *kw* ）

查询或修改指定*列*的选项。

如果没有给出*kw*，则返回列选项值的字典。如果 *选项*指定然后该值*选项*返回。否则，将选项设置为相应的值。

有效的选项/值是：

- ID
  返回列名称。这是一个只读选项。
- 锚点：标准Tk锚点值之一。
  指定该列中的文本应该如何与单元格对齐。
- 最小宽度：宽度
  列的最小宽度（以像素为单位）。当小部件被调整大小或用户拖动列时，treeview小部件不会使该列的大小小于此选项指定的大小。
- 拉伸：真/假
  指定在调整窗口小部件时是否应调整列的宽度。
- 宽度：宽度
  列的宽度（以像素为单位）。

要配置树列，请使用column =“＃0”

`delete`（*项目*）

删除所有指定的*项目*及其所有后代。

根项目不能被删除。

`detach`（*项目*）

取消链接树中的所有指定*项目*。

项目及其所有后代仍然存在，可能会重新插入树中的其他位置，但不会显示。

根项目不能分离。

`exists`（*item*）

`True`如果指定的*项目*存在于树中，则返回。

`focus`（*item = None*）

如果*项目*被指定，将重点项目*的项目*。否则，返回当前焦点项目，如果没有，则返回“'。

`heading`（*列*，*选项= None*，*** kw*）

查询或修改指定*列*的标题选项。

如果没有给出*kw*，则返回标题选项值的字典。如果 *选项*指定然后该值*选项*返回。否则，将选项设置为相应的值。

有效的选项/值是：

- 文本：文本
  要在列标题中显示的文本。
- image：imageName
  指定要显示在列标题右侧的图像。

- 主播：主播
  指定标题文本应如何对齐。标准Tk锚定值之一。
- 命令：回调
  按下标题标签时要调用的回调。

要配置树列标题，请使用column =“＃0”进行调用。

identify（ *component* ，*x* ，*y* ）
在*x*和*y*给定的点下返回指定*组件*的描述，如果在该位置不存在此*组件*，则返回空字符串。

identify_row（ *y* ）
返回位置*y*处的项目的项目ID 。

identify_column（ *x* ）
返回位置*x*的单元格的数据列标识符。

树列有ID＃0。

identify_region（ *x* ，*y* ）
返回以下之一：

地区	含义
标题	树标题区。
分隔器	两列标题之间的空格。
树	树区。
细胞	一个数据单元。

可用性：Tk 8.6。

identify_element（ *x* ，*y* ）
返回位置*x*，*y*处的元素。

可用性：Tk 8.6。

index（ *item* ）
返回其父*项*的子*项*列表中*项目*的整数索引。

insert（ *parent* ，*index* ，*iid = None* ，*** kw* ）
创建一个新项目并返回新创建项目的项目标识符。

*父*项是父项的项目标识，或者是用于创建新顶级项目的空字符串。*索引*是一个整数，或者值“end”，指定在父节点的孩子列表中插入新项目的位置。如果*index*小于或等于零，则新节点插入在开始处; 如果*index*大于或等于当前孩子的数量，则将其插入到最后。如果指定了*iid*，则将其用作项目标识符; *iid*不能在树中存在。否则，会生成一个新的唯一标识符。

请参阅项目选项以获取可用点的列表。

item (*item* , *option = None* , *** kw*)
查询或修改指定*项目*的选项。

如果没有给出选项，则返回具有该项目的选项/值的字典。如果*选项*被指定，则返回该选项的值。否则，将选项设置为由*kw*给出的相应值。

move (*item* , *parent* , *index*)
移动*项目*定位*指数*在*父母*的孩子的名单。

将某个项目移到其子项下是非法的。如果*索引*小于或等于零，则将*项目*移至开头; 如果大于或等于子女人数，则移至最后。如果*项目* 被分离，则重新附加。

next (*item*)
返回*项目*下一个兄弟的标识符，如果*item*是其父*项*的最后一个子*项*，则返回'' 。

parent (*item*)
返回*项目*父*项*的ID，如果*项目*位于层次结构的顶层，则返回'' 。

prev (*item*)
返回*项目*前一个兄弟的标识符，或者如果*item*是其父*项*的第一个子*项*，则返回'' 。

reattach (*item* , *parent* , *index*)
别名`Treeview.move()`。

see (*item*)
确保*项目*可见。

将所有*项目*的祖先打开选项设置为`True`，并在必要时滚动窗口小部件，以使*项目*位于树的可见部分内。

selection (*selop = None* , *items = None*)
如果未指定*selop*，则返回所选项目。否则，它将按照以下选择方法进行操作。

自版本3.6起弃用，将在版本3.8中删除：`selection()`不推荐使用更改选择状态。改为使用以下选择方法。

selection_set (**项目*)
*项目*成为新的选择。

*在版本3.6中更改：项目*可以作为单独的参数传递，而不仅仅作为单个元组传递。

selection_add (**项目*)
将*项目*添加到选择。

*在版本3.6中更改：项目*可以作为单独的参数传递，而不仅仅作为单个元组传递。

selection_remove (**项目*)
从选择中删除*项目*。

*在版本3.6中更改：项目*可以作为单独的参数传递，而不仅仅作为单个元组传递。

selection_toggle (*项目*)
切换中的每个项目的选择状态*的项目*。

*在版本3.6中更改：项目*可以作为单独的参数传递，而不仅仅作为单个元组传递。

set (*item* , *column* = *None* , *value* = *None*)
使用一个参数，返回指定*项目*的列/值对的字典。使用两个参数，返回指定*列*的当前值。使用三个参数，将给定*项目*中给定*列*的值设置 为指定*值*。

tag_bind (*tagname* , *sequence* = *None* , *callback* = *None*)
将给定事件*序列*的回调绑定到标记*标记名*。当一个事件被传递给一个项目时，将调用每个项目标签选项的回调。

tag_configure (*tagname* , *option* = *None* , ** *kw*)
查询或修改指定*标记名*的选项。

如果没有给出*kw*，则返回*标记名*选项设置的字典 。如果指定了*选项*，则返回 指定*标记名的该选项*的值。否则，将选项设置为给定*标记名*的相应值。

tag_has (*tagname* , *item* = *None*)
如果指定了*项目*，则根据指定的*项目*是否具有给定的*标记名*返回1或0 。否则，返回具有指定标签的所有项目的列表。

可用性：Tk 8.6

xview (**args*)
查询或修改树视图的水平位置。

yview (**args*)
查询或修改树视图的垂直位置。

10年2月25日。Ttk造型

每个窗口小部件ttk都分配了一种样式，它指定组成窗口小部件的元素集以及它们的排列方式，以及元素选项的动态和默认设置。默认情况下，样式名称与小部件的类名称相同，但可能会被小部件样式选项覆盖。如果您不知道小部件的类名称，请使用方法 Misc.winfo_class() （somewidget.winfo_class（ ））。

也可以看看:

Tcl'2004会议介绍
本文档解释了主题引擎的工作原理

*类*tkinter.ttk.Style
这个类用于操作样式数据库。

configure (*style* , *query_opt* = *None* , ** *kw*)

查询或设置*样式中*指定选项的默认值。

*kw*中的每个键都是一个选项，每个值都是标识该选项值的字符串。

例如，要将每个默认按钮更改为带有一些填充和不同背景颜色的平面按钮：

```
from tkinter import ttk
import tkinter

root = tkinter.Tk()

ttk.Style().configure("TButton", padding=6, relief="flat",
   background="#ccc")

btn = ttk.Button(text="Sample")
btn.pack()

root.mainloop()
```

map（*style*，*query_opt* = *None*，** *kw* ）

在*样式中*查询或设置指定选项的动态值。

*kw*中的每个键都是一个选项，每个值应该是一个列表或一个元组（通常），其中包含按元组，列表或其他首选项组合的statespecs。statepec是一个或多个状态的复合，然后是一个值。

一个例子可能会让它更容易理解：

```
import tkinter
from tkinter import ttk

root = tkinter.Tk()

style = ttk.Style()
style.map("C.TButton",
    foreground=[('pressed', 'red'), ('active', 'blue')],
    background=[('pressed', '!disabled', 'black'), ('active', 'white')]
    )

colored_btn = ttk.Button(text="Test", style="C.TButton").pack()

root.mainloop()
```

请注意，如果顺序更改为前景选项，则选项的（州，值）顺序顺序很重要，例如，当窗口小部件处于活动或按下状态时，结果将为蓝色前景。`[('active', 'blue'), ('pressed', 'red')]`

lookup（*style*，*option*，*state* = *None*，*default* = *None* ）

返回指定的值*选项*中*的风格*。

如果指定了*状态*，则预期它是一个或多个状态的序列。如果设置了*默认*参数，则在未找到选项说明的情况下将其用作后退值。

要检查Button默认使用的字体：

```
from tkinter import ttk

print(ttk.Style().lookup("TButton", "font"))
```

layout (*style* , *layoutspec* = *None*)

为给定*样式*定义小部件布局。如果省略*layoutspec*，则返回给定样式的布局规范。

如果指定，*layoutspec*应该是一个列表或其他序列类型（不包括字符串），其中每个项目应该是一个元组，第一个项目是布局名称，第二个项目应该具有Layouts中描述的格式。

要了解格式，请参阅以下示例（不打算执行任何有用的操作）：

```
from tkinter import ttk
import tkinter

root = tkinter.Tk()

style = ttk.Style()
style.layout("TMenubutton", [
   ("Menubutton.background", None),
   ("Menubutton.button", {"children":
       [("Menubutton.focus", {"children":
           [("Menubutton.padding", {"children":
               [("Menubutton.label", {"side": "left", "expand": 1})]
           })]
       })]
   }),
])

mbtn = ttk.Menubutton(text='Text')
mbtn.pack()
root.mainloop()
```

element_create (*elementname* , *etype* , ** args* , *** kw*)

在当前主题创建一个给定的一个新的元素，*VLAN的*预计将是另一个，“从”，“图像”或“VSAPI”。后者仅适用于Windows XP和Vista的Tk 8.6a，在此不再赘述。

如果使用“image”，则*args*应该包含默认图像名称，后跟statespec / value对（这是imagespec），*kw*可以有以下选项：

- 边界=填充
  填充是最多四个整数的列表，分别指定左边界，右边界，右边界和下边界。
- 高度=高度
  指定元素的最小高度。如果小于零，则基础图像的高度将用作默认值。
- 填充=填充
  指定元素的内部填充。如果未指定，则默认为边框的值。

- 粘=规范

  指定图像在最终宗地内的放置方式。spec包含零个或多个字符“n”，“s”，“w”或“e”。

- 宽度=宽度

  指定元素的最小宽度。如果小于零，则基础图像的宽度将用作默认值。

如果“from”用作*etype*的值， `element_create()` 则会克隆现有元素。*args*预计包含一个主题名称，元素将从中克隆，并且可以从中克隆一个元素。如果未指定要克隆的元素，则将使用空元素。*kw*被丢弃。

`element_names ( )`

返回当前主题中定义的元素列表。

`element_options ( elementname )`

返回*元素名称*的选项列表。

`theme_create ( 主题名 , 父=无 , 设置=无 )`

创建一个新的主题。

如果*themename*已经存在，那是错误的。如果指定了*父项*，则新主题将继承父主题的样式，元素和布局。如果存在*设置*，则预计它们具有与用于的相同的语法 `theme_settings()`。

`theme_settings ( 主题名称 , 设置 )`

暂时将当前主题设置为主题*名称*，应用指定的 *设置*，然后恢复以前的主题。

在每个键*的设置*是一个样式，每个值可以包含键“配置”，“图”，“布局”和“元件创建”和他们预计由方法规定为具有相同的格式 `Style.configure()` ， `Style.map()` ， `Style.layout()` 和 `Style.element_create()` 分别。

作为一个例子，让我们稍微改变默认主题的Combobox：

```
from tkinter import ttk
import tkinter

root = tkinter.Tk()

style = ttk.Style()
style.theme_settings("default", {
   "TCombobox": {
       "configure": {"padding": 5},
       "map": {
           "background": [("active", "green2"),
                          ("!disabled", "green4")],
           "fieldbackground": [("!disabled", "green3")],
           "foreground": [("focus", "OliveDrab1"),
                          ("!disabled", "OliveDrab2")]
       }
   }
})
```

```
combo = ttk.Combobox().pack()

root.mainloop()
```

theme_names（ ）

返回所有已知主题的列表。

theme_use（*themename* = *None* ）

如果*THEMENAME*没有给出，返回在使用中的主题。否则，将当前主题设置为*themename*，刷新所有小部件并发出<< ThemeChanged >>事件。

25.2.10.1。布局

布局可以是None，如果不需要任何选项，或者指定如何排列元素的选项字典。布局机制使用包几何管理器的简化版本：给定初始空洞，每个元素被分配一个包裹。有效的选项/值是：

- 方：哪边

  指定放置元件的腔的哪一侧; 顶部，右侧，底部或左侧之一。如果省略，该元件占据整个空腔。

- 粘性：nswe

  指定元素放置在其分配的宗地内的位置。

- 单位：0或1

  如果设置为1，则将元素及其所有后代作为单个元素处理，以用于Widget.identify()等。它用于像滚动条大拇指一样的东西。

- 儿童：[sublayout ...]

  指定放置在元素内的元素列表。每个元素都是一个元组（或其他序列类型），其中第一个项目是布局名称，另一个是布局。

25.3。 tkinter.tix- Tk的扩展小部件

源代码： Lib / tkinter / tix.py

*自从版本3.6开始弃用：*此Tk扩展未被维护，不应在新代码中使用。 tkinter.ttk改为使用 。

的tkinter.tix（TK接口扩展）模块提供一个额外的一套丰富的小部件。尽管标准Tk库有许多有用的小部件，但它们还远远没有完成。该tkinter.tix库提供大部分是从标准Tk的缺少通常所需部件的：HList， ComboBox，Control（又名SPINBOX）和滚动部件的分类。 tkinter.tix还包括有更多的窗口小部件是在宽范围的应用中通常是有用的：NoteBook ， FileEntry ， PanedWindow等; 其中有40多个。

使用所有这些新的小部件，您可以将新的交互技术引入到应用程序中，创建更有用且更直观的用户界面。您可以通过选择最合适的小部件来设计您的应用程序，以满足应用程序和用户的特殊需求。

也可以看看:

Tix主页
主页为Tix。这包括指向其他文档和下载的链接。

Tix人页
在线版本的手册页和参考资料。

Tix编程指南
程序员参考资料的在线版本。

Tix开发应用程序
Tix应用程序用于开发Tix和Tkinter程序。潮汐应用程序在Tk或Tkinter下工作，并包含**TixInspect**，一名检查员可远程修改和调试Tix / Tk / Tkinter应用程序。

25.3.1。使用蒂克斯

class tkinter.tix.Tk（*screenName = None*，*baseName = None*，*className ='Tix'* ）
Tix的Toplevel小部件主要代表应用程序的主窗口。它有一个关联的Tcl解释器。

中类tkinter.tix模块子类中的类 tkinter。前者导入后者，所以要tkinter.tix 与Tkinter一起使用，您只需导入一个模块。在一般情况下，你可以导入tkinter.tix，并更换到顶层通话 tkinter.Tk用tix.Tk：

```
from tkinter import tix
from tkinter.constants import *
root = tix.Tk()
```

要使用tkinter.tix，您必须安装Tix小部件，通常与安装Tk小部件一起安装。要测试您的安装，请尝试以下操作：

```
from tkinter import tix
root = tix.Tk()
root.tk.eval('package require Tix')
```

如果失败了，那么您的Tk安装问题必须在继续之前解决。使用环境变量TIX_LIBRARY指向安装蒂克斯库目录，并确保您有动态对象库（ tix8183.dll或libtix8183.so ）包含您的Tk的动态对象库（ 相同的目录tk8183.dll或 libtk8183.so ）。具有动态对象库的目录还应该有一个名为pkgIndex.tcl（区分大小写）的文件，其中包含以下行：

```
package ifneeded Tix 8.1 [list load "[file join $dir tix8183.dll]" Tix]
```

25.3.2。Tix

Tix 为tkinter剧目引入了40多个小部件类。

25.3.2.1。基本小工具

*类*tkinter.tix.**Balloon**

一个气球，超过一个部件弹出提供帮助。当用户将光标移动到已绑定了气球窗口小部件的窗口小部件中时，屏幕上将显示一个带有描述性消息的小型弹出式窗口。

*类*tkinter.tix.**ButtonBox**

所述ButtonBox 插件创建按钮的框，诸如通常用于。 Ok Cancel

*类*tkinter.tix.**ComboBox**

该组合框 控件类似于MS Windows中的组合框控件。用户可以通过键入条目子组件或从列表框子选择组件来选择一个选项。

*类*tkinter.tix.**Control**

该控制 窗口小部件也被称为SpinBox窗口小部件。用户可以通过按两个箭头按钮或直接在输入中输入数值来调整数值。新值将根据用户定义的上限和下限进行检查。

*类*tkinter.tix.**LabelEntry**

该LabelEntry 部件包的入口小部件和一个标签为一体的大型部件。它可以用来简化“输入表单”类型界面的创建。

*类*tkinter.tix.**LabelFrame**

该LabelFrame 部件包框控件和标签为一个大型部件。要在LabelFrame小部件内创建小部件，可以创建与小部件相关的新小部件，并在小部件 frame内部管理它们frame。

*类*tkinter.tix.**Meter**

该仪表控件可以用来显示后台作业可能需要很长的时间来执行进度。

*类*tkinter.tix.**OptionMenu**

该OptionMenu 创建一个选项菜单按钮。

*类*tkinter.tix.**PopupMenu**

该PopupMenu的 插件可被用作一个替换的tk_popup命令。 Tix PopupMenu小部件的优点是它需要较少的应用程序代码来操作。

*类*tkinter.tix.Select

在选择控件是按钮subwidgets的容器。它可以用来为用户提供无线电盒或复选框式的选择选项。

*类*tkinter.tix.StdButtonBox

所述StdButtonBox 插件是一组用于基序状对话框标准按钮。

25.3.2.2。文件选择器

*类*tkinter.tix.DirList

该DirList 插件播放目录，原有的目录和子目录的列表视图。用户可以选择列表中显示的目录之一或更改为其他目录。

*类*tkinter.tix.DirTree

该DirTree 插件播放目录，原有的目录和子目录的树视图。用户可以选择列表中显示的目录之一或更改为其他目录。

*类*tkinter.tix.DirSelectDialog

该DirSelectDialog 控件呈现在对话窗口中的文件系统的目录。用户可以使用此对话窗口浏览文件系统以选择所需的目录。

*类*tkinter.tix.DirSelectBox

这DirSelectBox与标准的Motif（TM）目录选择框类似。它通常用于用户选择一个目录。DirSelectBox将最近选定的目录存储到ComboBox小部件中，以便可以再次快速选择它们。

*类*tkinter.tix.ExFileSelectBox

该ExFileSelectBox 部件通常嵌入在tixExFileSelectDialog部件。它为用户选择文件提供了一个便利的方法。 ExFileSelectBox小部件的样式与 MS Windows 3.1上的标准文件对话框非常相似。

*类*tkinter.tix.FileSelectBox

所述FileSelectBox 是类似于标准基序（TM）文件选择框。它通常用于用户选择文件。FileSelectBox将大多数最近选择的文件存储到ComboBox小部件中，以便可以再次快速选择它们。

*类*tkinter.tix.FileEntry

该FileEntry的 插件可被用于输入一个文件名。用户可以手动键入文件名。或者，用户可以按下位于该条目旁边的按钮小部件，这将弹出一个文件选择对话框。

25.3.2.3。分层列表框

*类*tkinter.tix.HList

所述HList插件可以用于显示具有分级结构的任何数据，例如，文件系统的目录树。这些列表条目是缩进的，并通过分支线根据它们在层次结构中的位置进行连接。

*类*tkinter.tix.CheckList

该检查表 插件播放由用户选择的项目的列表。CheckList与Tk checkbutton或radiobutton小部件类似，除了它能够处理比checkbutton或radiobutton更多的项目。

*类*tkinter.tix.Tree

该树插件可用于以树的形式显示的阶层数据。用户可以通过打开或关闭部分树来调整树的视图。

25.3.2.4。表格列表框

*类*tkinter.tix.TList

所述的TList插件可用于以表格的形式来显示数据。TList小部件的列表条目 与Tk列表框小部件中的条目类似。主要区别是（1）TList小部件可以以二维格式显示列表条目，（2）可以使用图形图像以及列表条目的多种颜色和字体。

25.3.2.5。经理部件

*类*tkinter.tix.PanedWindow

该PanedWindow 插件允许用户以交互方式操纵几个窗格的大小。窗格可以垂直或水平排列。用户通过在两个窗格之间拖动调整大小手柄来更改窗格的大小。

*类*tkinter.tix.ListNoteBook

该ListNoteBook 小部件是非常相似的TixNoteBook小工具：它可以用来在使用笔记本的比喻有限的空间来显示多个窗口。笔记本被分成一堆页面（窗口）。一次只能显示其中的一个页面。用户可以通过在子hlist小组件中选择所需页面的名称来浏览这些页面。

*类*tkinter.tix.NoteBook

该笔记本 插件可用于在使用笔记本比喻的有限空间来显示多个窗口。笔记本被分成一堆页面。一次只能显示其中的一个页面。用户可以通过选择NoteBook小部件顶部的可视“标签”浏览这些页面。

25.3.2.6。图像类型

该tkinter.tix模块增加了：

- 所有图像映射功能tkinter.tix和tkinter小部件可以从XPM文件创建彩色图像。
- 复合图像类型可用于创建由多条水平线组成的图像; 每行由从左到右排列的一系列项目（文本，位图，图像或空间）组成。例如，可以使用复合图像在Tk Button 小部件中同时显示位图和文本字符串。

25.3.2.7。杂项小工具

*类*tkinter.tix.InputOnly

的InputOnly 窗口小部件是接受来自用户的，其可以与完成的输入bind 命令（仅限Unix）。

25.3.2.8。表格几何管理器

此外，tkinter.tix增强tkinter我们提供：

*类*tkinter.tix.Form
该表几何经理基于对所有Tk组件连接的规则。

25.3.3。TIX命令

*类*tkinter.tix.tixCommand
所述TIX命令提供访问的其他元素Tix的内部状态和 Tix应用程序上下文。这些方法操纵的大多数信息都与应用程序的整体或屏幕或显示有关，而不是与特定的窗口有关。

要查看当前设置，常见用法是：

```
from tkinter import tix
root = tix.Tk()
print(root.tix_configure())
```

tixCommand.tix_configure(*cnf = None*, *** kw*)
查询或修改Tix应用程序上下文的配置选项。如果未指定任何选项，则返回字典中的所有可用选项。如果选项没有指定值，那么该方法将返回一个描述一个名称选项的列表（如果没有指定任何选项，此列表将与返回值的相应子列表相同）。如果指定了一个或多个选项 - 值对，则该方法将给定的选项修改为具有给定的值; 在这种情况下，该方法返回一个空字符串。选项可以是任何配置选项。

tixCommand.tix_cget(*可选*)
返回给出的配置选项的当前值*的选项*。选项可以是任何配置选项。

tixCommand.tix_getbitmap(*名字*)
找到名称的位图文件name.xpm或name位于其中一个位图目录中（请参阅tix_addbitmapdir()方法）。通过使用 tix_getbitmap()，您可以避免在应用程序中硬编码位图文件的路径名。成功时，它将返回位图文件的完整路径名，并以该字符为前缀@。返回的值可以用来配置bitmapTk和Tix小部件的选项。

tixCommand.tix_addbitmapdir(*目录*)
Tix维护一个目录列表，这些目录tix_getimage()和 tix_getbitmap()方法将搜索图像文件。标准位图目录是$TIX_LIBRARY/bitmaps。该tix_addbitmapdir()方法将*目录*添加到此列表中。通过使用此方法，还可以使用tix_getimage()or tix_getbitmap()方法来定位应用程序的图像文件。

tixCommand.tix_filedialog([*dlgclass*])
返回可能在此应用程序的不同调用中共享的文件选择对话框。此方法将在第一次调用时创建文件选择对话框小部件。此对话将由所有后续呼叫返回tix_filedialog()。可选的dlgclass参数可以作为字符串传递，以指定需要什么类型的文件选择对话框小部件。可能的选择是tix，FileSelectDialog或者tixExFileSelectDialog。

tixCommand.tix_getimage(*self*, *name*)

找到名称的图像文件name.xpm，name.xbm或 name.ppm位于其中一个位图目录中（请参阅 tix_addbitmapdir() 上面的 方法）。如果存在多个具有相同名称（但扩展名不同）的文件，则根据X显示器的深度选择图像类型：在单色显示器上选择xbm图像，并在彩色显示器上选择彩色图像。通过使用tix_getimage()，您可以避免在应用程序中硬编码图像文件的路径名。成功时，此方法将返回新创建的图像的名称，该名称可用于配置imageTk和Tix小部件的选项。

tixCommand.tix_option_get(*名字*)

获取由Tix方案机制维护的选项。

tixCommand.tix_resetoptions(*newScheme*, *newFontSet* [, *newScmPrio*])

将Tix应用程序的方案和字体集分别重置为*newScheme*和 *newFontSet*。这只会影响这次调用后创建的小部件。因此，最好在Tix应用程序中创建任何小部件之前调用resetoptions方法。

可以给出可选参数*newScmPrio*来重置由Tix方案设置的Tk选项的优先级。

由于Tk处理X选项数据库的方式，Tix已经导入并定位后，不可能使用该tix_config()方法重置颜色方案和字体集。相反，tix_resetoptions() 必须使用该方法。

25.4。 tkinter.scrolledtext- 滚动文本小部件

源代码： Lib / tkinter / scrolledtext.py

该tkinter.scrolledtext模块提供了一个相同名称的类，它实现了一个基本文本小部件，该小部件具有一个垂直滚动条，用于执行“正确的事情”。使用ScrolledText该类比直接设置文本小部件和滚动条要容易得多。构造函数与tkinter.Text该类的相同。

文本组件和滚动条都在挤在一起Frame ，和的方法Grid和Pack几何管理者都是从收购Frame对象。这允许ScrolledText小部件直接用于实现最正常的几何管理行为。

如果需要更具体的控制，可以使用以下属性：

ScrolledText.**frame**
    围绕文本和滚动条小部件的框架。

ScrolledText.**vbar**
    滚动条小部件。

25.5。IDLE

源代码： Lib / idlelib /

IDLE是Python的集成开发和学习环境。

IDLE具有以下功能：

- 使用tkinterGUI工具包编码为100％纯Python
- 跨平台：在Windows，Unix和Mac OS X上大体相同
- Python shell窗口（交互式解释器），具有代码输入，输出和错误消息的颜色
- 多窗口文本编辑器，多重撤销，Python着色，智能缩进，调用提示，自动完成和其他功能
- 在任何窗口中搜索，在编辑器窗口中进行替换，并搜索多个文件（grep）
- 具有持久性断点的调试器，步进以及查看全局和本地命名空间
- 配置，浏览器和其他对话框

25.5.1。菜单

IDLE有两种主要的窗口类型，Shell窗口和Editor窗口。可以同时拥有多个编辑器窗口。输出窗口（例如用于编辑/在文件中查找）是编辑窗口的子类型。它们目前与编辑器窗口具有相同的顶部菜单，但是具有不同的默认标题和上下文菜单。

IDLE的菜单根据当前选择的窗口动态更改。下面记录的每个菜单都指示与哪个窗口类型相关联。

25.5.1.1。文件菜单（外壳和编辑器）

新文件
: 创建一个新的文件编辑窗口。

打开…
: 使用打开对话框打开现有文件。

最近的文件
: 打开最近的文件列表。点击一个打开它。

打开模块...
: 打开一个现有的模块（搜索sys.path）。

类浏览器
: 以树状结构显示当前编辑器文件中的函数，类和方法。在shell中，首先打开一个模块。

路径浏览器
: 以树结构显示sys.path目录，模块，函数，类和方法。

保存
: 将当前窗口保存到关联的文件（如果有的话）。自打开或上次保存以来更改的Windows在窗口标题之前和之后都有*。如果没有关联的文件，请改为另存为。

另存为…
用另存为对话框保存当前窗口。保存的文件成为该窗口的新关联文件。

保存副本为...
将当前窗口保存到不同的文件而不更改关联的文件。

打印窗口
将当前窗口打印到默认打印机。

关
关闭当前窗口（要求保存如果未保存）。

出口
关闭所有窗口并退出IDLE（要求保存未保存的窗口）。

25.5.1.2。编辑菜单（外壳和编辑器）

解开
撤消对当前窗口的最后更改。最多可以撤消1000次更改。

重做
将最后撤销的更改重做到当前窗口。

切
将选择复制到系统范围的剪贴板; 然后删除选择。

复制
将选择复制到系统范围的剪贴板中。

糊
将系统范围剪贴板的内容插入当前窗口。

剪贴板功能也可在上下文菜单中使用。

全选
选择当前窗口的全部内容。

找…
用许多选项打开一个搜索对话框

再次查找
重复上次搜索，如果有的话。

查找选择
搜索当前选择的字符串，如果有的话。

在文件中查找...
打开文件搜索对话框。将结果放入新的输出窗口。

更换…
打开一个搜索和替换对话框。

去线
将光标移至要求的行号并使该行可见。

显示完成
　　打开一个可滚动的列表，允许选择关键字和属性。请参阅下面“提示”部分的“完成”。

展开Word
　　展开您输入的前缀以匹配同一窗口中的完整单词; 重复以获得不同的扩展。

显示来电提示
　　在函数的未闭括号之后，用函数参数提示打开一个小窗口。

显示周围的包袱
　　突出显示周围的圆括号。

25.5.1.3。格式菜单（仅限编辑器窗口）

缩进区域
　　按缩进宽度右移所选行（默认4个空格）。

Dedent地区
　　移动缩进宽度左侧的所选行（默认4个空格）。

评论区域
　　在所选行的前面插入##。

取消注释区域
　　删除选定行中的前导＃或##。

Tabify Region
　　将*领先*的空间转变为标签。（注意：我们推荐使用4个空格块缩进Python代码。）

解除地区
　　将*所有*标签转换为正确数量的空格。

切换标签
　　打开一个对话框，在使用空格和制表符的缩进之间切换。

新的缩进宽度
　　打开一个对话框来改变缩进宽度。Python社区接受的默认值是4个空格。

格式段落
　　在注释块或多行字符串或字符串中的选定行中重新格式化当前空行分隔的段落。段落中的所有行将被格式化为小于N列，其中N默认为72。

去除尾部空白
　　通过将str.rstrip应用于每行（包括多行字符串中的行），删除行的最后一个非空白字符后的尾随空格和其他空白字符。

25.5.1.4。运行菜单（仅限编辑器窗口）

Python Shell
　　打开或唤醒Python Shell窗口。

检查模块
　　检查编辑器窗口中当前打开的模块的语法。如果模块未保存，IDLE将提示用户保存或自动保存，如在“空闲设置”对话框的“常规”选项卡中所选。如果出现语法错误，则在编辑器窗口

中显示大概的位置。

运行模块
: 做检查模块（上图）。如果没有错误，请重新启动外壳以清理环境，然后执行该模块。输出显示在Shell窗口中。请注意，输出需要使用print或write。执行完成后，Shell将保留焦点并显示提示。此时，可以交互式地探索执行结果。这与使用命令行执行文件类似。
`python -i file`

25.5.1.5。外壳菜单（仅限Shell窗口）

查看上次重新启动
: 将Shell窗口滚动到最后一个Shell重新启动。

重新启动Shell
: 重新启动外壳以清洁环境。

中断执行
: 停止正在运行的程序。

25.5.1.6。调试菜单（仅适用于Shell窗口）

转到文件/行
: 看在当前行。用光标，上面一行代表文件名和行号。如果找到，请打开文件（如果尚未打开），并显示该行。使用它可以查看异常回溯中引用的源代码行和查找文件中找到的代码行。在Shell窗口和Output窗口的上下文菜单中也可用。

调试器（切换）
: 激活时，在Shell中输入或从编辑器运行的代码将在调试器下运行。在编辑器中，可以使用上下文菜单设置断点。此功能仍然不完整，有点实验性。

堆栈查看器
: 显示树部件中最后一个异常的栈跟踪，可以访问本地和全局。

自动打开堆栈查看器
: 在未处理的异常情况下自动打开堆栈查看器。

25.5.1.7。选项菜单（外壳和编辑器）

配置IDLE
: 打开一个配置对话框，并为以下内容更改首选项：字体，缩进，键绑定，文本颜色主题，启动窗口和大小，其他帮助源和扩展（请参阅下文）。在OS X上，通过选择应用程序菜单中的首选项来打开配置对话框。要在较旧的IDLE中使用新的内置颜色主题（IDLE Dark），请将其保存为新的自定义主题。

  非默认用户设置保存在用户主目录中的.idlerc目录中。通过编辑或删除.idlerc中的一个或多个文件来解决由不良用户配置文件引起的问题。

代码上下文（切换）（仅限编辑器窗口）
: 在编辑窗口的顶部打开一个窗格，其中显示了在窗口顶部滚动的代码的块上下文。

25.5.1.8。窗口菜单（外壳和编辑器）

缩放高度
: 在正常大小和最大高度之间切换窗口。初始大小默认为40行80个字符，除非在配置IDLE对话框的常规选项卡上更改。

该菜单的其余部分列出了所有打开的窗口的名称; 选择一个将其带到前台（如果需要，可以对其进行解密）。

25.5.1.9。帮助菜单（Shell和编辑器）

关于IDLE
: 显示版本，版权，许可证，学分等等。

IDLE帮助
: 显示IDLE的帮助文件，详细说明菜单选项，基本编辑和导航以及其他提示。

Python文档
: 访问本地Python文档（如果已安装），或者启动Web浏览器并打开显示最新Python文档的docs.python.org。

龟演示
: 用示例python代码和龟图来运行turtledemo模块。

可以在此处添加其他帮助源，并在常规选项卡下的配置IDLE对话框中添加。

25.5.1.10。上下文菜单

在窗口中右键单击打开上下文菜单（按住Control键并单击OS X）。上下文菜单在编辑菜单上也有标准的剪贴板功能。

切
: 将选择复制到系统范围的剪贴板; 然后删除选择。

复制
: 将选择复制到系统范围的剪贴板中。

糊
: 将系统范围剪贴板的内容插入当前窗口。

编辑器窗口也具有断点功能。设置了断点的行被特别标记。在调试器下运行时，断点只有效果。文件的断点保存在用户的.idlerc目录中。

设置断点
: 在当前行上设置一个断点。

清除断点
: 清除该行的断点。

Shell和Output窗口具有以下内容。

转到文件/行
与“调试”菜单中相同。

25.5.2。编辑和导航

在本节中，'C'表示Control Windows和Unix上的Command密钥，以及Mac OSX上的密钥。

- Backspace删除到左侧; Del删除到右侧
- C-Backspace删除剩下的单词; C-Del删除右侧的单词
- 方向键和Page Up/ Page Down或四处移动
- C-LeftArrow并C-RightArrow通过文字移动
- Home/ End去开始/结束行
- C-Home/ C-End去文件的开始/结束
- 一些有用的Emacs绑定从Tcl / Tk继承而来：
    - C-a 行首
    - C-e 行结束
    - C-k 杀死行（但不放在剪贴板中）
    - C-l 插入点周围的中心窗口
    - C-b 后退一个字符而不删除（通常您也可以使用光标键进行此操作）
    - C-f 前进一个字符而不删除（通常您也可以使用光标键进行此操作）
    - C-p 往上走一条线（通常你也可以用光标键来做这个）
    - C-d 删除下一个字符

标准键绑定（如C-c复制和C-v粘贴）可能会起作用。在“配置IDLE”对话框中选择键绑定。

25.5.2.1。自动缩进

在块打开语句之后，下一行缩进4个空格（在Python Shell窗口中由一个选项卡）。在某些关键字（中断，返回等）后，下一行是缩进的。在领先的缩进中，Backspace如果他们在那里删除多达4个空格。Tab插入空格（在Python Shell窗口的一个选项卡中），数字取决于缩进宽度。目前，由于Tcl / Tk限制，选项卡限制为四个空格。

另请参阅编辑菜单中的缩进/缩进区域命令。

25.5.2.2。完成

完成提供了函数，类和类的属性，既有内置的也有用户定义的。还为文件名提供完成。

AutoCompleteWindow（ACW）将在“。”后面的预定义延迟（默认值为两秒）后打开。或者（在一个字符串中）输入os.sep。如果在其中一个字符（加上零个或更多其他字符）之后键入了一个选项卡，如果发现可能的延续，ACW将立即打开。

如果输入的字符只有一个可能的完成，a `Tab`将提供该完成而不打开ACW。

'显示完成'将强制打开完成窗口，默认情况下 `C-space`将打开完成窗口。在一个空字符串中，这将包含当前目录中的文件。在空白行中，它将包含当前命名空间中的内置和用户定义的函数和类，以及任何导入的模块。如果输入了一些字符，ACW将尝试更具体。

如果键入一串字符，则ACW选择将跳转到与这些字符最接近的条目。输入a `tab`将导致在编辑器窗口或Shell中输入最长的非歧义匹配。`tab`连续两次将提供当前的ACW选择，将返回或双击。光标键，Page Up / Down，鼠标选择和滚轮都在ACW上运行。

可以通过在'。'后面输入隐藏名称的开始来访问“隐藏”属性，例如'_'。这允许访问具有`__all__`设置的模块 或者类别私有属性。

完成和'扩展字'设施可以节省大量的输入！

完成目前仅限于名称空间中的完成。编辑器窗口中的名称不通过`__main__`且`sys.modules`不会被找到。用导入运行该模块一次以纠正这种情况。请注意，IDLE本身在sys.modules中放置了很多模块，默认情况下可以找到很多模块，例如re模块。

如果您不喜欢ACW弹出窗口，只需延长延迟时间或禁用扩展。

25.5.2.3。Calltips

当*可访问* 函数`(`的名称后面显示一个提示时，会显示一个提示。名称表达式可能包含点和下标。一个calltip会一直保留，直到它被点击，光标移出参数区域或键入。当光标位于定义的参数部分时，菜单或快捷方式显示一个提示。`)`

calltip由函数签名和文档字符串的第一行组成。对于没有可访问签名的内建函数，calltip由第五行或第一个空白行上的所有行组成。这些细节可能会改变。

这组*可访问的*函数取决于哪些模块已导入用户进程，包括由Idle本身导入的模块，以及自上次重新启动以来运行了哪些定义。

例如，重新启动Shell并输入`itertools.count(`。出现提示是因为Idle将itertools导入用户进程以供自己使用。（这可能会改变。）输入`turtle.write(`并没有显示。闲置不导入乌龟。菜单或快捷方式也不做任何事情。输入 ，然后将工作。`import turtle turtle.write(`

在编辑器中，导入语句在运行文件之前不起作用。有人可能希望在顶部写入导入语句之后运行文件，或者在编辑之前立即运行现有文件。

25.5.2.4。Python Shell窗口

- `C-c` 中断执行命令
- `C-d`发送文件结束; 如果在`>>>`提示符下键入，则关闭窗口
- `Alt-/` （扩展词）对于减少键入也很有用

  命令历史

- Alt-p检索与您输入的内容相匹配的先前命令。在OS X上使用C-p。
- Alt-n接下来检索。在OS X上使用C-n。
- Return 而在任何先前的命令中检索该命令

25.5.2.5。文字颜色

闲置默认为白色文字为黑色，但为具有特殊含义的文字添加颜色。对于shell，这些是shell输出，shell错误，用户输出和用户错误。对于Python代码，在shell提示符下或在编辑器中，这些都是关键字，内置类和函数名，名以下class和 def，字符串和评论。对于任何文本窗口，这些是光标（如果存在），找到的文本（如果可能）和选定的文本。

文字着色是在背景中完成的，因此偶尔可以看到无色文字。要更改配色方案，请使用“配置IDLE”对话框的“突出显示”选项卡。编辑器中的调试器断点行标记以及弹出窗口和对话框中的文本不是用户可配置的。

25.5.3。启动和代码执行

在使用该-s选项启动时，IDLE将执行由环境变量引用的文件IDLESTARTUP 要么PYTHONSTARTUP。IDLE首先检查IDLESTARTUP; 如果IDLESTARTUP存在，则引用的文件被运行。如果IDLESTARTUP不存在，则IDLE检查 PYTHONSTARTUP。由这些环境变量引用的文件是存储IDLE shell中频繁使用的函数或执行导入语句以导入公共模块的便利位置。

另外，Tk如果它存在，也会加载启动文件。请注意，Tk文件无条件加载。这个附加文件.Idle.py在用户的主目录中查找。此文件中的语句将在Tk命名空间中执行，因此该文件对于导入要从IDLE的Python shell中使用的函数没有用处。

25.5.3.1。命令行用法

```
idle.py [-c command] [-d] [-e] [-h] [-i] [-r file] [-s] [-t title] [-] [arg] ...

-c command  run command in the shell window
-d          enable debugger and open shell window
-e          open editor window
-h          print help message with legal combinations and exit
-i          open shell window
-r file     run file in shell window
-s          run $IDLESTARTUP or $PYTHONSTARTUP first, in shell window
-t title    set title of shell window
-           run stdin in shell (- must be last option before args)
```

如果有参数：

- 如果-，-c或r时，所有的参数都放在 sys.argv[1:...]和sys.argv[0]被设置为’’，’-c’或’-r’。即使在“选项”对话框中设置了默认值，也不会打开编辑器窗口。
- 否则，参数是打开的文件进行编辑并 sys.argv反映传递给IDLE自身的参数。

25.5.3.2。启动失败

IDLE使用套接字在IDLE GUI过程和用户代码执行过程之间进行通信。无论Shell何时启动或重新启动，都必须建立连接。（后者由分隔线表示'RESTART'）。如果用户进程无法连接到GUI进程，则会显示一个Tk错误框，其中包含指示用户在此处的“无法连接”消息。然后它退出。

失败的一个常见原因是与标准库模块名称相同的用户编写的文件，例如*random.py*和*tkinter.py*。当这样的文件与即将运行的文件位于同一目录中时，IDLE无法导入stdlib文件。目前的修复是重命名用户文件。

虽然不如过去普遍，但防病毒或防火墙程序可能会阻止连接。如果程序不能被允许连接，那么必须关闭IDLE才能工作。允许这种内部连接是安全的，因为在外部端口上不可见数据。类似的问题是阻止连接的网络错误配置。

Python安装问题偶尔会停止IDLE：多个版本可能会发生冲突，或者单个安装可能需要管理员访问权限。如果撤消冲突，或不能或不想以管理员身份运行，则可能最容易彻底删除Python并重新开始。

僵尸pythonw.exe进程可能是一个问题。在Windows上，使用任务管理器检测并停止一个。有时由程序崩溃或键盘中断（control-C）启动的重新启动可能无法连接。关闭Shell菜单上的错误框或重新启动Shell可能会解决一个临时问题。

当IDLE首次启动时，它会尝试读取~/ .idlerc /（~是主目录）中的用户配置文件。如果出现问题，应显示错误消息。如果不考虑随机磁盘故障，可以通过从不使用选项下的配置对话框手动编辑文件来避免这种情况。一旦发生，解决方案可能是删除一个或多个配置文件。

如果IDLE没有消息退出，并且它不是从控制台启动的，请尝试从控制台启动（并查看是否显示消息。python -m idlelib）

25.5.3.3。空闲 - 控制台差异

除极少数情况外，使用IDLE执行Python代码的结果与在控制台窗口中执行相同的代码相同。但是，不同的界面和操作偶尔会影响可见结果。例如，sys.modules从更多条目开始。

IDLE还替换sys.stdin，sys.stdout以及sys.stderr从Shell窗口获取输入并将输出发送到Shell窗口的对象。当Shell有重点时，它控制着键盘和屏幕。这通常是透明的，但直接访问键盘和屏幕的功能将不起作用。如果sys是带复位importlib.reload(sys)，闲置的更改都将丢失之类的东西input，raw_input和 print将无法正常工作。

利用IDLE的Shell，您可以输入，编辑和回顾完整的报表。一些控制台一次只能使用一条物理线路。IDLE exec用来运行每个语句。因此，'__builtins__'总是为每个语句定义。

25.5.3.4。开发tkinter应用程序

为了促进tkinter程序的开发，IDLE有意与标准Python不同。在标准Python中输入并且不显示任何内容 在IDLE中输入相同的内容，并出现一个tk窗口。在标准的Python中，还必须进入 才能看到窗口。IDLE在背景中具有相同的功能，每秒约20次，大约每50毫秒。接下来进入。再次，标准Python中没有任何明显的变化，直到进入。import tkinter as tk; root = tk.Tk() root.update() b = tk.Button(root, text='button'); b.pack() root.update()

大多数tkinter程序运行`root.mainloop()`，通常在tk应用程序被销毁之前不会返回。如果程序是使用IDLE编辑器运行的，或者使用IDLE编辑器运行该程序，则在 返回之前不会显示shell提示符，此时没有任何可交互的内容。`python -i>>>mainloop()`

从IDLE编辑器运行tkinter程序时，可以注释掉主循环调用。然后，立即得到一个shell提示，并可以与实时应用程序进行交互。只需要记住在标准Python中运行时重新启用主循环呼叫。

25.5.3.5。没有子　　运行

默认情况下，IDLE通过使用内部回送接口的套接字在单独的子进程中执行用户代码。此连接不是外部可见的，并且没有数据发送到互联网或从互联网接收数据。如果防火墙软件无论如何抱怨，你可以忽略它。

如果尝试使套接字连接失败，空闲将通知您。这样的故障有时是暂时的，但如果持续存在，则问题可能是防火墙阻塞连接或配置特定系统错误。在问题解决之前，可以使用-n命令行开关运行Idle。

如果使用-n命令行开关启动IDLE，它将在单个进程中运行，并且不会创建运行RPC Python执行服务器的子进程。如果Python无法在平台上创建子进程或RPC套接字接口，这可能很有用。但是，在这种模式下，用户代码与IDLE本身并不是隔离的。另外，选择运行/运行模块（F5）时，环境不会重新启动。如果您的代码已被修改，您必须重新加载（）受影响的模块并重新导入任何特定项目（例如从foo import baz），如果更改生效。由于这些原因，如果可能的话，最好使用缺省子进程运行IDLE。

自3.4版以来已弃用。

25.5.4。帮助和偏好

25.5.4.1。其他帮助来源

IDLE包含一个名为“Python Docs”的帮助菜单条目，该条目将打开广泛的帮助来源，包括docs.python.org上提供的教程。可以随时使用“配置IDLE”对话框从帮助菜单中添加或删除选定的URL。有关更多信息，请参阅IDLE帮助菜单中的IDLE帮助选项。

25.5.4.2。设置首选项

字体首选项，高亮显示，按键和常规首选项可通过选项菜单上的配置IDLE进行更改。密钥可以是用户定义的; IDLE附带四个内置密钥集。另外，用户可以在键标签下的配置IDLE对话框中创建一个自定义键集。

25.5.4.3。扩展

IDLE包含一个扩展工具。扩展的首选项可以使用首选项对话框的扩展选项卡进行更改。有关更多信息，请参阅idlelib目录中config-extensions.def的开头部分。当前唯一的默认扩展名是zzdummy，这也是一个用于测试的示例。

25.6。其他图形用户界面包

主要的跨平台（Windows，Mac OS X，类Unix）GUI工具包可用于Python：

也可以看看:

PyGObject
PyGObject为使用GObject的 C库提供了自检绑定 。其中一个库是GTK + 3小部件集。GTK +配备了比Tkinter更多的小部件。一个在线的 Python GTK + 3教程 可用。

PyGTK的
PyGTK为GTK + 2的旧版本库提供了绑定。它提供了一个面向对象的接口，它比C接口稍高一些。还有对GNOME的绑定 。在线教程可用。

PyQt的
PyQt是一个绑定到Qt工具包的**sip**包装。Qt是一个广泛的C ++ GUI应用程序开发框架，可用于Unix，Windows和Mac OS X. **sip**是一种为Python类生成C ++库绑定的工具，专门为Python设计。

PySide
PySide是一个由诺基亚提供的Qt工具包的更新绑定。与PyQt相比，其许可方案对非开源应用程序更友好。

wxPython的
wxPython是一个用于Python的跨平台GUI工具包，它是围绕流行的wxWidgets（以前称为wxWindows）C ++工具包构建的。它通过尽可能使用每个平台的本地窗口小部件（类Unix系统上的GTK +）为Windows，Mac OS X和Unix系统上的应用程序提供本机外观和感觉。除了大量的小部件外，wxPython还提供了用于在线文档和上下文相关帮助，打印，HTML查看，低级设备上下文绘制，拖放，系统剪贴板访问，基于XML的资源格式等的类，包括一个不断增长的用户贡献模块库。

PyGTK，PyQt和wxPython都具有比Tkinter更现代的外观和感觉以及更多的部件。另外，还有许多用于Python的其他GUI工具包，包括跨平台和特定于平台的。请参阅Python Wiki中的GUI编程页面以获取更完整的列表，以及链接到不同GUI工具包进行比较的文档。

26.开发工具

本章介绍的模块可帮助您编写软件。例如， `pydoc`模块需要一个模块并根据模块的内容生成文档。的`doctest`和`unittest`模块包含用于编写单元测试自动行使代码并确认该期望的输出产生框架。 **2to3**可以将Python 2.x源代码转换为有效的Python 3.x代码。

本章描述的模块列表是：

26.1。 typing- 支持类型提示

3.5版本中的新功能。

源代码： Lib / typing.py

> **注意:** 打字模块暂时包含在标准库中 。如果核心开发人员认为有必要，可能会添加新功能，甚至可能会在次要版本之间改变API。

该模块支持按照指定的类型提示 **PEP 484**和**PEP 526**。最根本的支持包含类型的`Any` ，`Union` ，`Tuple` ，`Callable` ，`TypeVar` ，和 `Generic`。完整的规格请参阅**PEP 484**。有关提示类型的简化介绍，请参阅**PEP 483**。

下面的函数接受并返回一个字符串，并注释如下：

```
def greeting(name: str) -> str:
    return 'Hello ' + name
```

在函数中`greeting` ，参数`name`预期是类型 `str`和返回类型`str`。亚型被接受为参数。

26.1.1。类型别名

类型别名通过将类型分配给别名来定义。在这个例子中， `Vector`与`List[float]`将被视为可互换同义词：

```
from typing import List
Vector = List[float]

def scale(scalar: float, vector: Vector) -> Vector:
    return [scalar * num for num in vector]

# typechecks; a list of floats qualifies as a Vector.
new_vector = scale(2.0, [1.0, -4.2, 5.4])
```

类型别名可用于简化复杂类型签名。例如：

```
from typing import Dict, Tuple, List

ConnectionOptions = Dict[str, str]
Address = Tuple[str, int]
Server = Tuple[Address, ConnectionOptions]

def broadcast_message(message: str, servers: List[Server]) -> None:
    ...

# The static type checker will treat the previous type signature as
# being exactly equivalent to this one.
def broadcast_message(
        message: str,
```

```
        servers: List[Tuple[Tuple[str, int], Dict[str, str]]]) -> None:
    ...
```

请注意，None作为类型提示是一种特殊情况，并被替换为 type(None)。

26.1.2。NEWTYPE

使用NewType()助手函数来创建不同的类型：

```
from typing import NewType

UserId = NewType('UserId', int)
some_id = UserId(524313)
```

静态类型检查器会将新类型视为它是原始类型的子类。这对于帮助捕捉逻辑错误非常有用：

```
def get_user_name(user_id: UserId) -> str:
    ...

# typechecks
user_a = get_user_name(UserId(42351))

# does not typecheck; an int is not a UserId
user_b = get_user_name(-1)
```

您仍然可以int对类型的变量执行所有操作UserId，但结果将始终是类型的int。这可以让你通过UserId任何int可能的 地方，但是会阻止你无意中UserId以无效的方式创建一个：

```
# 'output' is of type 'int', not 'UserId'
output = UserId(23413) + UserId(54341)
```

请注意，这些检查仅通过静态类型检查程序强制执行。在运行时，该语句将创建一个函数，该函数立即返回您传递它的任何参数。这意味着该表达式不会创建一个新的类或引入超出常规函数调用的开销。Derived = NewType('Derived', Base)DerivedDerived(some_value)

更确切地说，表达式在运行时总是如此。some_value is Derived(some_value)

这也意味着无法创建子类型，Derived 因为它是运行时的标识函数，而不是实际的类型：

```
from typing import NewType

UserId = NewType('UserId', int)

# Fails at runtime and does not typecheck
class AdminUserId(UserId): pass
```

但是，可以NewType()基于“派生”创建一个NewType：

```
from typing import NewType

UserId = NewType('UserId', int)
```

```
ProUserId = NewType('ProUserId', UserId)
```

和类型检查ProUserId将按预期工作。

看到 **PEP 484**了解更多详情。

> **注意:** 回想一下，使用类型别名声明两种类型彼此*等效*。这样做将会使静态类型检查治疗为*完全等同*于在所有情况下。当您想简化复杂类型签名时，这很有用。Alias = OriginalAliasOriginal
>
> 相反，NewType声明一种类型是另一种类型的*子类型*。这样做会使静态类型检查器作为一个*子类*来处理，这意味着类型值不能用在需要类型值的地方 。当您想以最小的运行时间成本防止逻辑错误时，这非常有用。 Derived = NewType('Derived', Original)DerivedOriginalOriginalDerived

3.5.2版本的新功能。

26.1.3。可赎回

预期特定签名的回调函数的框架可能会使用类型暗示。Callable[[Arg1Type, Arg2Type], ReturnType]

例如：

```
from typing import Callable

def feeder(get_next_item: Callable[[], str]) -> None:
    # Body

def async_query(on_success: Callable[[int], None],
                on_error: Callable[[int, Exception], None]) -> None:
    # Body
```

可以通过用文字省略号替换类型提示中的参数列表来声明可调用的返回类型，而无需指定调用签名。Callable[..., ReturnType]

26.1.4。泛型

由于无法以通用方式静态推断有关保存在容器中的对象的类型信息，因此抽象基类已扩展为支持订阅以表示容器元素的预期类型。

```
from typing import Mapping, Sequence

def notify_by_email(employees: Sequence[Employee],
                    overrides: Mapping[str, str]) -> None: ...
```

泛型可以通过使用可用于打字的新工厂进行参数化TypeVar。

```
from typing import Sequence, TypeVar

T = TypeVar('T')      # Declare type variable

def first(l: Sequence[T]) -> T:   # Generic function
    return l[0]
```

26.1.5。用户自定义泛型类型

用户定义的类可以定义为泛型类。

```
from typing import TypeVar, Generic
from logging import Logger

T = TypeVar('T')

class LoggedVar(Generic[T]):
    def __init__(self, value: T, name: str, logger: Logger) -> None:
        self.name = name
        self.logger = logger
        self.value = value

    def set(self, new: T) -> None:
        self.log('Set ' + repr(self.value))
        self.value = new

    def get(self) -> T:
        self.log('Get ' + repr(self.value))
        return self.value

    def log(self, message: str) -> None:
        self.logger.info('%s: %s', self.name, message)
```

Generic[T]作为基类定义该类LoggedVar采用单个类型参数T。这也可以T作为类体内的类型。

所述Generic基类使用一个元类定义 __getitem__() ，使得LoggedVar[t]是作为一种类型的有效的：

```
from typing import Iterable

def zero_all_vars(vars: Iterable[LoggedVar[int]]) -> None:
    for var in vars:
        var.set(0)
```

泛型类型可以有任意数量的类型变量，并且类型变量可能会受到限制：

```
from typing import TypeVar, Generic
...

T = TypeVar('T')
S = TypeVar('S', int, str)
```

```
class StrangePair(Generic[T, S]):
    ...
```

每个类型的变量参数Generic必须是不同的。因此这是无效的：

```
from typing import TypeVar, Generic
...

T = TypeVar('T')

class Pair(Generic[T, T]):   # INVALID
    ...
```

您可以使用多重继承Generic：

```
from typing import TypeVar, Generic, Sized

T = TypeVar('T')

class LinkedList(Sized, Generic[T]):
    ...
```

从泛型类继承时，某些类型变量可能是固定的：

```
from typing import TypeVar, Mapping

T = TypeVar('T')

class MyDict(Mapping[str, T]):
    ...
```

在这种情况下MyDict有一个参数，T。

Any对每个位置使用不指定类型参数的通用类。在下面的例子中，MyIterable不是泛型的，而是隐式地继承自Iterable[Any]：

```
from typing import Iterable

class MyIterable(Iterable): # Same as Iterable[Any]
```

用户定义的通用类型别名也受支持。例子：

```
from typing import TypeVar, Iterable, Tuple, Union
S = TypeVar('S')
Response = Union[Iterable[S], int]

# Return type here is same as Union[Iterable[str], int]
def response(query: str) -> Response[str]:
    ...

T = TypeVar('T', int, float, complex)
Vec = Iterable[Tuple[T, T]]
```

```
def inproduct(v: Vec[T]) -> T: # Same as Iterable[Tuple[T, T]]
    return sum(x*y for x, y in v)
```

元类使用的Generic是其子类abc.ABCMeta。通用类可以通过包含抽象方法或属性成为ABC，并且泛型类也可以具有ABCs作为基类而不存在元类冲突。通用元类不受支持。参数化泛型的结果被缓存，并且打字模块中的大部分类型都是可散列的，并且可以相互平等。

26.1.6。该Any类型

一种特殊的类型是Any。静态类型检查器会将每种类型视为与每种类型兼容Any并Any兼容。

这意味着可以对on类型的值执行任何操作或方法调用，Any并将其分配给任何变量：

```
from typing import Any

a = None    # type: Any
a = []      # OK
a = 2       # OK

s = ''      # type: str
s = a       # OK

def foo(item: Any) -> int:
    # Typechecks; 'item' could be any type,
    # and that type might have a 'bar' method
    item.bar()
    ...
```

请注意，将类型值分配给Any更精确的类型时，不会执行 类型检查。例如，静态类型检查器在分配a时s即使s被声明为类型str并且int在运行时接收到值，也不会报告错误！

此外，没有返回类型或参数类型的所有函数将默认默认使用Any：

```
def legacy_parser(text):
    ...
    return data

# A static type checker will treat the above
# as having the same signature as:
def legacy_parser(text: Any) -> Any:
    ...
    return data
```

当您需要动态混合和静态类型代码时，此行为Any可用作*逃生舱口*。

将行为Any与行为进行对比object。类似于Any，每个类型都是一个子类型object。然而，与Any中，反之则不然：object是*不是*所有其他类型的子类型。

这意味着，当一个值object的类型是，类型检查器会拒绝几乎所有的操作，并将它分配给一个更专用类型的变量（或使用它作为返回值）是一个类型错误。例如：

```
def hash_a(item: object) -> int:
    # Fails; an object does not have a 'magic' method.
    item.magic()
    ...

def hash_b(item: Any) -> int:
    # Typechecks
    item.magic()
    ...

# Typechecks, since ints and strs are subclasses of object
hash_a(42)
hash_a("foo")

# Typechecks, since Any is compatible with all types
hash_b(42)
hash_b("foo")
```

使用object以指示值可以是任何类型的类型安全方式。使用Any以指示值是动态类型。

26.1.7。类，函数和装饰器

该模块定义了以下类，函数和装饰器：

*类*typing.TypeVar

类型变量。

用法：

```
T = TypeVar('T')  # Can be anything
A = TypeVar('A', str, bytes)  # Must be str or bytes
```

类型变量的存在主要是为了静态类型检查器的好处。它们用作泛型类型的参数以及泛型函数定义。有关泛型类型的更多信息，请参见Generic类。通用函数的工作如下：

```
def repeat(x: T, n: int) -> Sequence[T]:
    """Return a list containing n references to x."""
    return [x]*n

def longest(x: A, y: A) -> A:
    """Return the longest of two strings."""
    return x if len(x) >= len(y) else y
```

后一个例子的签名本质上是和的重载。另请注意，如果参数是某些子类的实例，则返回类型仍然是普通的。(str, str) -> str(bytes, bytes) -> bytesstrstr

在运行时，会提高。在一般情况下，并且不应该与类型使用。isinstance(x, T)TypeErrorisinstance()issubclass()

类型变量可能被标记为协变或通过传递covariant=True或逆变 contravariant=True。看到**PEP 484**了解更多详情。默认情况下，类型变量是不变的。或者，类型变量可以指定使用

上限bound=<type>。这意味着，对于类型变量替换（显式或隐式）的实际类型必须是边界类型的子类，请参阅**PEP 484**。

*类*typing.Generic

泛型类型的抽象基类。

泛型类型通常是通过从一个具有一个或多个类型变量的类的实例继承来声明的。例如，一个通用的映射类型可能被定义为：

```
class Mapping(Generic[KT, VT]):
    def __getitem__(self, key: KT) -> VT:
        ...
        # Etc.
```

这个类然后可以使用如下：

```
X = TypeVar('X')
Y = TypeVar('Y')

def lookup_name(mapping: Mapping[X, Y], key: X, default: Y) -> Y:
    try:
        return mapping[key]
    except KeyError:
        return default
```

class typing.Type（*Generic [CT_co]*）

一个带有注解的变量C可以接受一个类型的值C。与此相反，带注释的变量Type[C]可以接受是类本身的值-即，它将接受*类对象*的 C。例如：

```
a = 3         # Has type 'int'
b = int       # Has type 'Type[int]'
c = type(a)   # Also has type 'Type[int]'
```

请注意，这Type[C]是协变的：

```
class User: ...
class BasicUser(User): ...
class ProUser(User): ...
class TeamUser(User): ...

# Accepts User, BasicUser, ProUser, TeamUser, ...
def make_new_user(user_class: Type[User]) -> User:
    # ...
    return user_class()
```

Type[C]covariant 这个事实意味着所有的子类都 C应该实现相同的构造函数签名和类方法签名C。类型检查器应该标记违反这个的情况，但是也应该允许子类中的构造函数调用与指定的基类中的构造函数调用相匹配。如果需要类型检查器来处理这种特殊情况，可能会在将来的修订版本中发生变化 **PEP 484**。

唯一的合法参数Type是类，类的联合和 Any。例如：

```
def new_non_team_user(user_class: Type[Union[BaseUser, ProUser]]): ...
```

Type[Any]相当于Type它的等价物type，它是Python的元类层次结构的根。

3.5.2版本的新功能。

class typing. Iterable (*Generic [T_co]*)
通用版本的collections.abc.Iterable。

class typing. Iterator (*Iterable [T_co]*)
通用版本的collections.abc.Iterator。

class typing. Reversible (*Iterable [T_co]*)
通用版本的collections.abc.Reversible。

*类*typing. SupportsInt
一种抽象方法的ABC __int__。

*类*typing. SupportsFloat
一种抽象方法的ABC __float__。

*类*typing. SupportsComplex
一种抽象方法的ABC __complex__。

*类*typing. SupportsBytes
一种抽象方法的ABC __bytes__。

*类*typing. SupportsAbs
带有一个抽象方法的ABC，__abs__它的返回类型是协变的。

*类*typing. SupportsRound
带有一个抽象方法的ABC，__round__ 它的返回类型是协变的。

class typing. Container (*Generic [T_co]*)
通用版本的collections.abc.Container。

*类*typing. Hashable
别名 collections.abc.Hashable

*类*typing. Sized
别名 collections.abc.Sized

class typing. Collection (*Sized , Iterable [T_co] , Container [T_co]*)
通用版本的 collections.abc.Collection

3.6版本中的新功能。

class typing. AbstractSet (*Sized , Collection [T_co]*)

通用版本的collections.abc.Set。

class typing.MutableSet (*AbstractSet [T]*)
通用版本的collections.abc.MutableSet。

class typing.Mapping (*Sized , Collection [KT] , Generic [VT_co]*)
通用版本的collections.abc.Mapping。

class typing.MutableMapping (*Mapping [KT , VT]*)
通用版本的collections.abc.MutableMapping。

class typing.Sequence (*可逆[T_co] , 集合[T_co]*)
通用版本的collections.abc.Sequence。

class typing.MutableSequence (*Sequence [T]*)
通用版本的collections.abc.MutableSequence。

class typing.ByteString (*Sequence [int]*)
通用版本的collections.abc.ByteString。

这种类型代表的类型bytes , bytearray和memoryview。

作为这种类型的缩写 , bytes可以用于注释上述任何类型的参数。

*类*typing.Deque (*deque , MutableSequence [T]*)
通用版本的collections.deque。

3.6.1版本的新功能。

class typing.List (*list , MutableSequence [T]*)
通用版本list。用于注释返回类型。注释它优选使用抽象集合类型 , 如参数Mapping , Sequence或AbstractSet。

这种类型可以使用如下：

```
T = TypeVar('T', int, float)

def vec2(x: T, y: T) -> List[T]:
    return [x, y]

def keep_positives(vector: Sequence[T]) -> List[T]:
    return [item for item in vector if item > 0]
```

*类*typing.Set (*set , MutableSet [T]*)
通用版本的builtins.set。

class typing.FrozenSet (*frozenset , AbstractSet [T_co]*)
通用版本的builtins.frozenset。

class typing. MappingView (*Sized , Iterable [T_co]*)

通用版本的collections.abc.MappingView。

*类*typing. KeysView (*MappingView [KT_co] , AbstractSet [KT_co]*)

通用版本的collections.abc.KeysView。

*类*typing. ItemsView (*MappingView , Generic [KT_co , VT_co]*)

通用版本的collections.abc.ItemsView。

*类*typing. ValuesView (*MappingView [VT_co]*)

通用版本的collections.abc.ValuesView。

class typing. Awaitable (*Generic [T_co]*)

通用版本的collections.abc.Awaitable。

class typing. Coroutine (*Awaitable [V_co] , Generic [T_co T_contra , V_co]*)

通用版本的collections.abc.Coroutine。类型变量的方差和顺序对应于那些变量和顺序Generator，例如：

```
from typing import List, Coroutine
c = None # type: Coroutine[List[str], str, int]
...
x = c.send('hi') # type: List[str]
async def bar() -> None:
    x = await c # type: int
```

class typing. AsyncIterable (*Generic [T_co]*)

通用版本的collections.abc.AsyncIterable。

*类*typing. AsyncIterator (*AsyncIterable [T_co]*)

通用版本的collections.abc.AsyncIterator。

class typing. ContextManager (*Generic [T_co]*)

通用版本的contextlib.AbstractContextManager。

3.6版本中的新功能。

class typing. AsyncContextManager (*Generic [T_co]*)

带有异步抽象__aenter__()和__aexit__() 方法的ABC 。

3.6版本中的新功能。

class typing. Dict (*dict , MutableMapping [KT , VT]*)

通用版本的dict。这种类型的用法如下：

```
def get_position_in_index(word_list: Dict[str, int], word: str) -> int:
    return word_list[word]
```

class typing. DefaultDict (*collections.defaultdict , MutableMapping [KT , VT]*)

通用版本的collections.defaultdict。

3.5.2版本的新功能。

*类*typing. Counter (*collections.Counter , Dict [T , int]*)

通用版本的collections.Counter。

3.6.1版本的新功能。

*类*typing. ChainMap (*collections.ChainMap , MutableMapping [KT , VT]*)

通用版本的collections.ChainMap。

3.6.1版本的新功能。

class typing. Generator (*Iterator [T_co] , Generic [T_co , T_contra , V_co]*)

生成器可以用通用类型进行注释。例如：Generator[YieldType, SendType, ReturnType]

```
def echo_round() -> Generator[int, float, str]:
    sent = yield 0
    while sent >= 0:
        sent = yield round(sent)
    return 'Done'
```

请注意，与打字模块中的许多其他泛型不同，它们SendType 的Generator行为是不一致的，而不是共变或不变的。

如果您的发电机只能屈服值，设置SendType和 ReturnType到None：

```
def infinite_stream(start: int) -> Generator[int, None, None]:
    while True:
        yield start
        start += 1
```

或者，注释您的生成器的返回类型为Iterable[YieldType]或者Iterator[YieldType]：

```
def infinite_stream(start: int) -> Iterator[int]:
    while True:
        yield start
        start += 1
```

*类*typing. AsyncGenerator (*AsyncIterator [T_co] , 泛型[T_co , T_contra]*)

异步生成器可以使用通用类型进行注释。例如：AsyncGenerator[YieldType, SendType]

```
async def echo_round() -> AsyncGenerator[int, float]:
    sent = yield 0
    while sent >= 0.0:
        rounded = await round(sent)
        sent = yield rounded
```

与普通生成器不同，异步生成器不能返回值，所以没有ReturnType类型参数。与之相反Generator，这种 SendType行为是矛盾的。

如果您的发生器只能产生数值，请将其设置SendType为 None：

```
async def infinite_stream(start: int) -> AsyncGenerator[int, None]:
    while True:
        yield start
        start = await increment(start)
```

或者，注释您的生成器的返回类型为AsyncIterable[YieldType]或者AsyncIterator[YieldType]：

```
async def infinite_stream(start: int) -> AsyncIterator[int]:
    while True:
        yield start
        start = await increment(start)
```

3.5.4版本的新功能。

*类*typing.**Text**

Text是一个别名str。它提供了为Python 2代码提供向前兼容的路径：在Python 2中，它Text是一个别名 unicode。

使用Text以指示值必须包含的方式，与两者的Python 2和Python 3兼容Unicode字符串：

```
def add_unicode_checkmark(text: Text) -> Text:
    return text + u' \u2713'
```

3.5.2版本的新功能。

*类*typing.**io**

I / O流类型的包装器名称空间。

这定义了通用类型IO[AnyStr]和亚类TextIO 和BinaryIO，从导出IO[str]和IO[bytes]分别。这些代表了返回的I / O流的类型 open()。

这些类型也直接访问的typing.IO， typing.TextIO和typing.BinaryIO。

*类*typing.**re**

正则表达式匹配类型的包装器名称空间。

这定义了类型别名，Pattern以及Match哪些对应于来自re.compile()和 的返回类型re.match()。这些类型的（和对应的功能）是通用的AnyStr，并且可以通过书面提出具体Pattern[str]，Pattern[bytes]，Match[str]，或 Match[bytes]。

这些类型也可以直接作为typing.Pattern 和typing.Match。

*类*typing.**NamedTuple**

namedtuple的键入版本。

用法：

```
class Employee(NamedTuple):
    name: str
    id: int
```

这相当于：

```
Employee = collections.namedtuple('Employee', ['name', 'id'])
```

要给字段一个默认值，你可以在类体中赋值给它：

```
class Employee(NamedTuple):
    name: str
    id: int = 3

employee = Employee('Guido')
assert employee.id == 3
```

具有默认值的字段必须位于没有默认值的字段后面。

生成的类有两个额外的属性：_field_types，给一个字典映射字段名称类型，和_field_defaults一个字典映射字段名称为默认值。（字段名称在_fields属性中，属于已命名 的API的一部分。）

NamedTuple 子类也可以有文档和方法：

```
class Employee(NamedTuple):
    """Represents an employee."""
    name: str
    id: int = 3

    def __repr__(self) -> str:
        return f'<Employee {self.name}, id={self.id}>'
```

向后兼容的用法：

```
Employee = NamedTuple('Employee', [('name', str), ('id', int)])
```

*在版本3.6中更改：*增加了对**PEP 526**变量注释语法。

*在版本3.6.1中进行了更改：*添加了对默认值，方法和文档字符串的支持。

typing.NewType(*典型*)

向类型检查器指示不同类型的帮助器函数，请参阅NewType。在运行时它将返回一个返回其参数的函数。用法：

```
UserId = NewType('UserId', int)
first_user = UserId(1)
```

3.5.2版本的新功能。

typing.cast(*typ*, *val*)

将值转换为类型。

这将返回值不变。对于类型检查器，这表示返回值具有指定类型，但在运行时我们故意不检查任何内容（我们希望它尽可能快）。

typing.get_type_hints(*obj*[, *globals*[, *locals*]])

返回包含函数，方法，模块或类对象的类型提示的字典。

这通常与之相同obj.__annotations__。另外，编码为字符串文字的前向引用是通过在globals和locals命名空间中对它们进行评估来处理的。如果需要，Optional[t]如果设置了默认值，则添加函数和方法注释None。对于一类C，返回通过合并所有的构造一个字典__annotations__沿着 C.__mro__以相反的顺序。

@typing.overload

所述@overload装饰允许描述支持参数类型的多个不同的组合的功能和方法。一系列@overload修饰后的定义必须紧跟一个非@overload修饰定义（对于相同的功能/方法）。该@overload-decorated定义仅用于类型检查的好处，因为它们将通过非覆盖@overload-decorated定义，而后者则是在运行时使用，但应该由一个类型检查被忽略。在运行时，@overload直接调用-decorated函数会引发 NotImplementedError。重载的一个例子给出了比使用联合或类型变量表示更精确的类型：

```
@overload
def process(response: None) -> None:
    ...
@overload
def process(response: int) -> Tuple[int, str]:
    ...
@overload
def process(response: bytes) -> str:
    ...
def process(response):
    <actual implementation>
```

看到 **PEP 484**了解详细信息并与其他打字语义进行比较。

@typing.no_type_check

Decorator指出注释不是类型提示。

这用作类或函数装饰器。对于一个类，它递归地应用于该类中定义的所有方法（但不适用于在其超类或子类中定义的方法）。

这会使函数发生变化。

@typing.no_type_check_decorator

装饰者给另一个装饰者的no_type_check()效果。

这将包装装饰函数的东西包装在装饰函数中no_type_check()。

typing.Any

指示无约束类型的特殊类型。

- 每种类型都兼容Any。
- Any 与各种类型兼容。

typing. Union
联盟类型; 指X或Y.Union[X, Y]

要定义联合，请使用例如。细节：Union[int, str]

- 参数必须是类型，并且必须至少有一个。
- 工会工会变平，例如：

```
Union[Union[int, str], float] == Union[int, str, float]
```

- 单一论点的工会消失，例如：

```
Union[int] == int  # The constructor actually returns int
```

- 冗余参数被跳过，例如：

```
Union[int, str, int] == Union[int, str]
```

- 比较联合时，参数顺序被忽略，例如：

```
Union[int, str] == Union[str, int]
```

- 当一个类及其子类存在时，后者被跳过，例如：

```
Union[int, object] == object
```

- 你不能继承或实例化一个联合。
- 你不能写Union[X][Y]。
- 您可以Optional[X]用作简写。Union[X, None]

typing. Optional
可选类型。

Optional[X]相当于。Union[X, None]

请注意，这与可选参数的概念不同，后者是具有缺省值的参数。具有默认值的可选参数不需要Optional在其类型注释中使用限定符（尽管如果默认值是推断的None）。Optional如果None允许显式值，则强制参数可能仍然有一个类型。

typing. Tuple
元组类型; 是包含第一个X类型和第二个Y类型的两个元素的元组的类型。Tuple[X, Y]

例子：是两个元素的元组对应于类型变量T1和T2。 是一个int，一个浮点数和一个字符串的元组。Tuple[T1, T2]Tuple[int, float, str]

要指定同类型的可变长度元组，请使用文字省略号，例如。一个平面 相当于，然后是平面。Tuple[int, ...]TupleTuple[Any, ...]tuple

typing.Callable

可调用类型; 是（int） - > str的函数。Callable[[int], str]

订阅语法必须始终与两个值一起使用：参数列表和返回类型。参数列表必须是类型列表或省略号; 返回类型必须是单一类型。

没有语法表示可选参数或关键字参数; 这种函数类型很少用作回调类型。 （文字省略号）可用于键入提示可调参数并返回任意数量的参数 。一个平面相当于，然后 是 平面。Callable[..., ReturnType]ReturnTypeCallableCallable[..., Any]collections.abc.Callable

typing.ClassVar

用于标记类变量的特殊类型构造。

正如介绍的那样 **PEP 526**，一个包装在ClassVar中的变量注释，表明给定的属性是用作类变量，不应该在该类的实例上设置。用法：

```
class Starship:
    stats: ClassVar[Dict[str, int]] = {} # class variable
    damage: int = 10                     # instance variable
```

ClassVar 只接受类型，不能进一步订阅。

ClassVar本身不是一个类，不应该与isinstance()or 一起使用issubclass()。 ClassVar不会更改Python运行时行为，但可以由第三方类型检查程序使用。例如，类型检查器可能会将以下代码标记为错误：

```
enterprise_d = Starship(3000)
enterprise_d.stats = {} # Error, setting class variable on instance
Starship.stats = {}     # This is OK
```

3.5.3版本的新功能。

typing.AnyStr

AnyStr是一个类型变量，定义为 。AnyStr = TypeVar('AnyStr', str, bytes)

它被用于可以接受任何类型的字符串而不允许混合不同类型的字符串的函数。例如：

```
def concat(a: AnyStr, b: AnyStr) -> AnyStr:
    return a + b

concat(u"foo", u"bar")  # Ok, output has type 'unicode'
concat(b"foo", b"bar")  # Ok, output has type 'bytes'
concat(u"foo", b"bar")  # Error, cannot mix unicode and bytes
```

typing.TYPE_CHECKING

假定True由第三方静态类型检查器的特殊常量。它False在运行时。用法：

```
if TYPE_CHECKING:
    import expensive_mod

def fun(arg: 'expensive_mod.SomeType') -> None:
    local_var: expensive_mod.AnotherType = other_fun()
```

请注意，第一个类型注释必须用引号括起来，使其成为“前向引用”，以隐藏`expensive_mod`解释器运行时的引用。不评估局部变量的类型注释，因此第二个注释不需要用引号括起来。

3.5.2版本的新功能。

26.2。 pydoc- 文档生成器和在线帮助系统

源代码： Lib / pydoc.py

该pydoc模块自动从Python模块生成文档。文档可以作为控制台上的文本页面呈现，提供给Web浏览器或保存到HTML文件。

对于模块，类，函数和方法，所显示的文档是从对象的docstring（即__doc__属性）派生的，并且是对其可记录成员的递归派生。如果没有文档字符串，则pydoc尝试从源文件中的类，函数或方法的定义上方或模块的顶部（请参阅参考资料）中的注释行块获取说明inspect.getcomments()。

内置函数help()调用交互式解释器中的联机帮助系统，该解释器用于pydoc在控制台上以文本形式生成其文档。通过在操作系统的命令提示符处将**pydoc**作为脚本运行，也可以从Python解释器外部查看相同的文本文档。例如，跑步

```
pydoc sys
```

在shell提示符下将显示sys模块上的文档，其格式类似于Unix **man**命令所显示的手册页。**pydoc**的参数可以是函数，模块或包的名称，也可以是包中模块或模块中的类，方法或函数的虚线引用。如果**pydoc**的参数看起来像一个路径（即它包含操作系统的路径分隔符，例如Unix中的斜杠），并且引用现有的Python源文件，则会为该文件生成文档。

> **注意：** 为了查找对象及其文档，pydoc导入要记录的模块。因此，模块级别的任何代码都将在此时执行。当一个文件被调用为一个脚本而不是仅仅被导入时，使用一个警卫来执行代码。 `if __name__ == '__main__':`

将输出打印到控制台时，**pydoc会**尝试对输出进行分页以便于阅读。如果PAGER环境变量已设置， **pydoc**将使用其值作为分页程序。

-w在参数前指定一个标志将导致HTML文档被写出到当前目录中的文件中，而不是在控制台上显示文本。

-k在参数前面指定一个标志将以类似于Unix **man**命令的方式再次搜索所有可用模块的提纲行，以获得作为参数给定的关键字。模块的摘要行是其文档字符串的第一行。

您也可以使用**pydoc**在本地机器上启动HTTP服务器，该服务器将为访问Web浏览器提供文档。**pydoc -p 1234** 将在端口1234上启动HTTP服务器，允许您http://localhost:1234/在首选Web浏览器中浏览文档。指定0为端口号将选择一个任意未使用的端口。

pydoc -b将启动服务器并另外打开Web浏览器到模块索引页面。每个提供服务的页面在顶部都有一个导航栏，您可以*获取*有关单个项目的帮助，在其摘要行中*搜索*包含关键字的所有模块，然后转至*模块索引*，*主题*和 *关键字*页面。

当**pydoc**生成文档时，它使用当前的环境和路径来查找模块。因此， 如果启动Python解释器并键入，则调用**pydoc垃圾邮件**文档就是您将获得的模块版本。 `import spam`

核心模块模块文件被假定为居住在 https://docs.python.org/X.Y/library/ 哪里X以及Y是Python解释器的主要和次要版本号。这可以通过设置覆盖PYTHONDOCS 环境变量添加到不同的URL或包含“库参考手册”页面的本地目录。

*在版本3.2中更改：*添加了该-b选项。

*在3.3版本中更改：*将-g取出的命令行选项。

在版本3.4中进行了更改： pydoc 现在使用 inspect.signature() 而不是 inspect.getfullargspec()从可调参数中提取签名信息。

26.3。 doctest- 测试交互式Python示例

源代码： Lib / doctest.py

该doctest模块搜索看起来像交互式Python会话的文本片段，然后执行这些会话以验证它们完全按照所示方式工作。有几种常用的方法可以使用doctest：

- 通过验证所有交互式示例仍按记录工作来检查模块的文档是否是最新的。
- 通过验证测试文件或测试对象中的交互式示例如预期工作来执行回归测试。
- 为包编写教程文档，用输入输出示例大量地说明。根据实例或说明文本是否被强调，这具有“识字测试”或“可执行文档”的味道。

这是一个完整但很小的示例模块：

```
"""
This is the "example" module.

The example module supplies one function, factorial().  For example,

>>> factorial(5)
120
"""

def factorial(n):
    """Return the factorial of n, an exact integer >= 0.

    >>> [factorial(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> factorial(30)
    265252859812191058636308480000000
    >>> factorial(-1)
    Traceback (most recent call last):
        ...
    ValueError: n must be >= 0

    Factorials of floats are OK, but the float must be an exact integer:
    >>> factorial(30.1)
    Traceback (most recent call last):
        ...
    ValueError: n must be exact integer
    >>> factorial(30.0)
    265252859812191058636308480000000

    It must also not be ridiculously large:
    >>> factorial(1e100)
    Traceback (most recent call last):
        ...
    OverflowError: n too large
    """

    import math
    if not n >= 0:
        raise ValueError("n must be >= 0")
```

```
    if math.floor(n) != n:
        raise ValueError("n must be exact integer")
    if n+1 == n:  # catch a value like 1e300
        raise OverflowError("n too large")
    result = 1
    factor = 2
    while factor <= n:
        result *= factor
        factor += 1
    return result


if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

如果您example.py直接从命令行运行，那么doctest 可以发挥它的魔力：

```
$ python example.py
$
```

没有输出！这很正常，而且这意味着所有的例子都有效。传递 -v给脚本，并doctest打印它正在尝试的详细日志，并在最后打印摘要：

```
$ python example.py -v
Trying:
    factorial(5)
Expecting:
    120
ok
Trying:
    [factorial(n) for n in range(6)]
Expecting:
    [1, 1, 2, 6, 24, 120]
ok
```

以此类推，最终结局如下：

```
Trying:
    factorial(1e100)
Expecting:
    Traceback (most recent call last):
        ...
    OverflowError: n too large
ok
2 items passed all tests:
   1 tests in __main__
   8 tests in __main__.factorial
9 tests in 2 items.
9 passed and 0 failed.
Test passed.
$
```

这就是你需要知道的开始有效使用doctest！跳入。以下部分提供完整的详细信息。请注意，标准Python测试套件和库中有很多文档测试的例子。在标准测试文件中可以找到特别有用的示例

Lib/test/test_doctest.py。

26.3.1。简单用法：检查Docstrings中的示例

开始使用doctest的最简单的方法（但不一定是您继续这样做的方式）是结束每个模块M：

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

doctest然后在模块中检查文档字符串M。

将模块作为脚本运行会导致文档字符串中的示例得到执行和验证：

```
python M.py
```

这将不会显示任何东西，除非一个例子失败，在这种情况下，失败的例子和失败的原因被打印到标准输出，并且输出的最后一行是，其中*N*是数字失败的例子。 ***Test Failed*** N failures.

-v改为使用开关运行它：

```
python M.py -v
```

并且所有尝试过的示例的详细报告都会打印到标准输出，并在最后列出各种摘要。

您可以通过传递 verbose=True 来强制详细模式 testmod() ，或通过传递来禁止它 verbose=False。在任何一种情况下， sys.argv都不会被检查testmod()（如此通过-v或没有影响）。

还有一个用于运行的命令行快捷方式testmod()。您可以指示Python解释器直接从标准库运行doctest模块，并在命令行上传递模块名称：

```
python -m doctest -v example.py
```

这将example.py作为独立模块导入并testmod()在其上运行 。请注意，如果文件是软件包的一部分并从该软件包导入其他子模块，则可能无法正常工作。

有关更多信息testmod()，请参阅基本API一节。

26.3.2。简单用法：检查文本文件中的示例

doctest的另一个简单应用是在文本文件中测试交互式示例。这可以通过以下testfile()功能完成：

```
import doctest
doctest.testfile("example.txt")
```

该短脚本执行并验证文件中包含的任何交互式Python示例example.txt。文件内容被视为一个巨大的文档字符串; 该文件不需要包含Python程序！例如，也许example.txt包含这个：

```
The ``example`` module
======================

Using ``factorial``
-------------------

This is an example text file in reStructuredText format.  First import
``factorial`` from the ``example`` module:

    >>> from example import factorial

Now use it:

    >>> factorial(6)
    120
```

运行doctest.testfile("example.txt")然后在这个文档中找到错误：

```
File "./example.txt", line 14, in example.txt
Failed example:
    factorial(6)
Expected:
    120
Got:
    720
```

与之一样testmod()，testfile()除非例子失败，否则不会显示任何内容。如果一个例子失败了，那么失败的例子和失败的原因将被打印到标准输出中，格式为 testmod()。

默认情况下，testfile()查找调用模块目录中的文件。有关可用于指示其在其他位置查找文件的可选参数的说明，请参见Basic API一节。

就像testmod()，testfile()可以通过-v命令行开关或可选的关键字参数 *verbose*来设置详细程度 。

还有一个用于运行的命令行快捷方式testfile()。您可以指示Python解释器直接从标准库运行doctest模块，并在命令行上传递文件名：

```
python -m doctest -v example.txt
```

因为文件名并不以此结束.py，因此doctest推断它必须与其一起运行testfile()，而不是testmod()。

有关更多信息testfile()，请参阅基本API一节。

26.3.3。工作

本节将详细介绍doctest如何工作：查看它的文档字符串，它如何查找交互式示例，它使用的执行上下文，它如何处理异常以及如何使用选项标志来控制其行为。这是编写doctest示例时需要

了解的信息; 有关在这些示例上实际运行doctest的信息，请参阅以下各节。

26.3.3.1。哪些Docstrings被检查？

模块docstring，以及所有函数，类和方法文档字符串被搜索。导入到模块中的对象不被搜索。

另外，如果M.__test__存在且“为真”，则它必须是字典，并且每个条目将（字符串）名称映射到函数对象，类对象或字符串。从中找到的函数和类对象文档字符串M.__test__被搜索，字符串被视为文档字符串。在输出，一键K在 M.__test__出现与名称

```
<name of M>.__test__.K
```

找到的任何类都以相似的方式递归搜索，以测试其包含的方法和嵌套类中的文档字符串。

CPython实现细节：在版本3.4之前，用C语言编写的扩展模块没有通过doctest完全搜索。

26.3.3.2。Docstring示例如何被认可？

在大多数情况下，交互式控制台会话的复制和粘贴工作正常，但doctest并不试图精确模拟任何特定的Python shell。

```
>>> # comments are ignored
>>> x = 12
>>> x
12
>>> if x == 13:
...     print("yes")
... else:
...     print("no")
...     print("NO")
...     print("NO!!!")
...
no
NO
NO!!!
>>>
```

任何期望的输出必须紧跟在 包含代码的最后一行或一行之后，并且预期的输出（如果有的话）扩展到下一行 或全空白行。’>>> ’’... ’’>>> ’

细则：

- 预期的输出不能包含全空白行，因为这样的行被用来表示预期输出的结束。如果预期的输出包含空白行，请<BLANKLINE>在doctest示例中输入空行。
- 所有硬标签字符都被扩展为空格，使用8列制表位。测试代码生成的输出中的选项卡不会被修改。由于示例输出*中的*任何硬标签*都是*展开的，这意味着如果代码输出包含硬标签，则doctest可以通过的唯一方式是如果 NORMALIZE_WHITESPACE选项或指令 有效。或者，可以重写测试以捕获输出并将其作为测试的一部分与预期值进行比较。源代码中对制表符的处理是通过反复试验得出的，并且已被证明是处理它们的最不容易出错的方式。通过编写自定义DocTestParser类，可以使用不同的算法来处理选项卡。

- 输出到标准输出被捕获，但不输出到标准错误（异常追溯通过不同的方式捕获）。
- 如果在交互式会话中通过反斜线继续行，或者出于任何其他原因使用反斜杠，则应该使用原始文档字符串，该字符串将按照键入时的方式保存反斜杠：

```
>>> def f(x):
...     r'''Backslashes in a raw docstring: m\n'''
>>> print(f.__doc__)
Backslashes in a raw docstring: m\n
```

否则，反斜杠将被解释为字符串的一部分。例如，`\n`以上将被解释为一个换行符。或者，您可以在doctest版本中将每个反斜杠加倍（并且不使用原始字符串）：

```
>>> def f(x):
...     '''Backslashes in a raw docstring: m\\n'''
>>> print(f.__doc__)
Backslashes in a raw docstring: m\n
```

- 起始栏无关紧要：

```
>>> assert "Easy!"
     >>> import math
         >>> math.floor(1.9)
         1
```

并且从开始示例的初始行中出现的预期输出中删除了许多主要的空白字符。'`>>> `'

26.3.3.3。什么是执行上下文？

默认情况下，每次`doctest`发现一个文档字符串进行测试，它采用的是 *浅拷贝*的`M`的全局，使运行测试不会改变模块真实的全局，因此，在一个测试`M`不能离开屑不小心让另外一个背后测试工作。这意味着示例可以自由使用任何在顶层定义的`M`名称，以及在运行的文档字符串中定义的名称。示例无法看到其他文档中定义的名称。

你可以通过强制使用自己的字典作为执行上下文 `globs=your_dict`来`testmod()`或`testfile()`替代。

26.3.3.4。什么是例外？

没问题，只要回溯是该示例生成的唯一输出：只需粘贴回溯。[1]由于回溯包含可能快速变化的细节（例如，确切的文件路径和行号），所以这是doctest很难灵活接受的一种情况。

简单的例子：

```
>>> [1, 2, 3].remove(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

该文档测试成功，如果`ValueError`提出，详情如图所示。`list.remove(x): x not in list`

预期的异常输出必须以追溯标题开头，该标题可以是以下两行中的任一行，缩写与示例的第一行相同：

```
Traceback (most recent call last):
Traceback (innermost last):
```

traceback头后面跟着一个可选的traceback堆栈，其内容被doctest忽略。回溯堆栈通常被忽略，或者从交互式会话逐字复制。

跟踪堆栈后面是最有趣的部分：包含异常类型和细节的行。这通常是追溯的最后一行，但如果异常具有多行详细信息，则可以跨越多行：

```
>>> raise ValueError('multi\n    line\ndetail')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: multi
    line
detail
```

最后三行（以开始`ValueError`）与异常的类型和细节进行比较，其余部分将被忽略。

最佳做法是省略追溯堆栈，除非它为示例增加了重要的文档值。所以最后一个例子可能更好，因为：

```
>>> raise ValueError('multi\n    line\ndetail')
Traceback (most recent call last):
    ...
ValueError: multi
    line
detail
```

请注意，回溯处理非常特别。特别是，在改写的例子中，使用`...`独立于doctest的`ELLIPSIS`选项。这个例子中的省略号可以省略，或者可以是三个（或三百个）逗号或数字，或者Monty Python skit的缩进记录。

一些细节你应该阅读一次，但不需要记住：

- Doctest无法猜测您的预期输出是来自异常追溯还是来自普通打印。因此，例如，预计会传递一个示例，无论是否实际提出，或者该示例仅打印该追溯文本。实际上，普通输出很少以追溯标题行开始，所以这不会产生实际问题。`ValueError: 42 is prime``ValueError`
- 回溯堆栈的每一行（如果存在）必须比示例的第一行缩进得更远，*或者*以非字母数字字符开始。追溯标题后面的第一行缩写相同，并以字母数字开头，作为异常详细信息的开始。当然这对于真正的回溯来说是正确的。
- 当`IGNORE_EXCEPTION_DETAIL`指定doctest选项时，将忽略最左侧冒号后面的所有内容以及异常名称中的所有模块信息。
- 交互式shell省略了一些`SyntaxError`s 的追溯标题行。但doctest使用traceback标题行来区分异常和非异常。因此，在极少数情况下，如果您需要测试一个`SyntaxError`省略traceback头的测试，则需要手动将traceback头行添加到测试示例中。

- 对于某些SyntaxErrors，Python使用^标记来显示语法错误的字符位置：

```
>>> 1 1
  File "<stdin>", line 1
    1 1
      ^
SyntaxError: invalid syntax
```

由于显示错误位置的行出现在异常类型和细节之前，因此它们不会被doctest检查。例如，即使将^标记放在错误的位置，也会通过以下测试：

```
>>> 1 1
  File "<stdin>", line 1
    1 1
    ^
SyntaxError: invalid syntax
```

26.3.3.5。选项标志

许多选项标志控制着doctest行为的各个方面。这些标志的符号名称作为模块常量提供，可以 按位或运算并传递给各种函数。这些名称也可以在doctest指令中使用，并可以通过-o选项传递给doctest命令行界面。

*在新版本3.4：*在-o命令行选项。

第一组选项定义测试语义，控制doctest如何确定实际输出是否与示例预期输出相匹配的方面：

doctest.DONT_ACCEPT_TRUE_FOR_1

默认情况下，如果预期的输出块只包含1，只是含有实际输出块1或仅True被认为是一个匹配，并类似地用于0对False。当DONT_ACCEPT_TRUE_FOR_1指定时，不允许替换。缺省行为迎合了Python将许多函数的返回类型从整数更改为布尔值; 希望“小整数”输出的doctests在这些情况下仍然有效。这个选项可能会消失，但不会持续数年。

doctest.DONT_ACCEPT_BLANKLINE

默认情况下，如果预期的输出块包含仅包含字符串的行<BLANKLINE>，则该行将匹配实际输出中的空行。由于真正的空行界定了预期的输出，因此这是沟通预期空行的唯一方式。什么时候 DONT_ACCEPT_BLANKLINE被指定，这个替代是不允许的。

doctest.NORMALIZE_WHITESPACE

指定时，所有空白（空格和换行符）都被视为相等。预期输出中的任何空白序列都将与实际输出中的任何空白序列相匹配。默认情况下，空白必须完全匹配。NORMALIZE_WHITESPACE当预期输出的行很长时，并且您想要在源代码中的多行中包装它时，它特别有用。

doctest.ELLIPSIS

指定时，...预期输出中的省略号标记（）可以匹配实际输出中的任何子字符串。这包括跨越行边界的子字符串和空的子字符串，所以最好保持简单的使用。复杂的用途可能会导致相同类型的“oops，它匹配得太多了！”.*在正则表达式中很容易出现意外。

doctest.IGNORE_EXCEPTION_DETAIL

指定时，即使异常详细信息不匹配，如果引发了期望类型的异常，那么期望异常的示例也会通过。例如，如果引发的实际异常是预期的例子，但会失败，例如，如果 引发。ValueError: 42ValueError: 3*14TypeError

它也会忽略Python 3 doctest报告中使用的模块名称。因此，无论测试是在Python 2.7还是Python 3.2（或更高版本）下运行，这两种变体都可以与指定的标志一起使用：

```
>>> raise CustomError('message')
Traceback (most recent call last):
CustomError: message

>>> raise CustomError('message')
Traceback (most recent call last):
my_module.CustomError: message
```

请注意，ELLIPSIS也可以用于忽略异常消息的详细信息，但根据是否将模块详细信息作为异常名称的一部分进行打印，此类测试可能仍会失败。使用IGNORE_EXCEPTION_DETAIL和来自Python 2.3的细节也是编写文档测试的唯一明确方式，它不关心异常细节，但仍然在Python 2.3或更低版本中继续传递（这些版本不支持doctest指令并将它们忽略为不相关的注释）。例如：

```
>>> (1, 2)[3] = 'moo'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object doesn't support item assignment
```

虽然Python 2.4中的细节更改为“不”而不是“不”，但在Python 2.3以及更高版本的Python版本中通过了指定的标志。

*在版本3.2中进行了更改：*IGNORE_EXCEPTION_DETAIL现在也忽略了与包含被测异常的模块有关的任何信息。

doctest.SKIP

指定时，请不要运行该示例。这在doctest示例既可用作文档也可用作测试用例的情况下非常有用，应将其用于文档目的，但不应进行检查。例如，该示例的输出可能是随机的; 或者该示例可能依赖于测试驱动程序无法使用的资源。

SKIP标志也可用于临时“注释”示例。

doctest.COMPARISON_FLAGS

将上面的所有比较标志掩盖起来。

第二组选项控制如何报告测试失败：

doctest.REPORT_UDIFF

指定时，涉及多行预期和实际输出的故障将使用统一差异显示。

doctest.REPORT_CDIFF

指定时，涉及多行预期输出和实际输出的故障将使用上下文差异显示。

doctest.REPORT_NDIFF
指定时，difflib.Differ使用与常用ndiff.py实用程序相同的算法计算差异。这是标记线内和线间差异的唯一方法。例如，如果预期输出的一行包含数字1，其中实际输出包含字母l，则会插入一行，并在其中插入用于标记不匹配列位置的插入符号。

doctest.REPORT_ONLY_FIRST_FAILURE
指定时，显示每个doctest中的第一个失败示例，但禁止所有其他示例的输出。这将防止doctest报告因早期故障而中断的正确示例; 但它也可能隐藏不正确的例子，不依靠第一次失败而失败。当 REPORT_ONLY_FIRST_FAILURE指定时，剩余的示例仍在运行，并仍然计入报告的故障总数; 只有输出被抑制。

doctest.FAIL_FAST
指定时，在第一个失败示例之后退出，不要尝试运行其余示例。因此，报告的故障数最多为1.在调试期间，此标志可能很有用，因为第一次故障后的示例甚至不会产生调试输出。

doctest命令行接受该选项-f作为简写。-o FAIL_FAST

3.4版新增功能

doctest.REPORTING_FLAGS
将上面的所有报告标记掩盖起来。

还有一种方法可以注册新的选项标志名称，但除非您打算doctest通过子类扩展内部函数，否则这种方法并不有用。

doctest.register_optionflag(*名字*)
用给定名称创建一个新选项标志，并返回新标志的整数值。 register_optionflag()可用于子类化 OutputChecker 或 DocTestRunner 创建您的子类支持的新选项。register_optionflag()应该总是使用以下习惯用法来调用：

```
MY_FLAG = register_optionflag('MY_FLAG')
```

26.3.3.6。指令

Doctest指令可用于修改单个示例的选项标志。Doctest指令是遵循示例源代码的特殊Python注释：

```
指令             :: =“#”“doctest：”directive_options    :: =    （“，”）\ *
 directive_option     :: =   on_or_off            :: =“+”\ |  “ - ”
 directive_option_name :: =“DONT_ACCEPT_BLANKLINE”\ |  “NORMALIZE_WHITESPACE”\ | .
directive_options
directive_optiondirective_optionon_or_off  directive_option_name
```

+or -和指令选项名称之间不允许有空格。指令选项名称可以是上面解释的任何选项标志名称。

一个例子的doctest指令修改了doctest的这个例子的行为。使用+启用这个名字的行为，或-将其禁用。

例如，这个测试通过：

```
>>> print(list(range(20))) # doctest: +NORMALIZE_WHITESPACE
[0,  1,  2,  3,  4,  5,  6,  7,  8,  9,
10,  11,  12,  13,  14,  15,  16,  17,  18,  19]
```

如果没有指令，它会失败，这是因为实际输出在单个数字列表元素之前没有两个空格，并且因为实际输出在单行上。这个测试也通过了，并且还需要一个指令来做到这一点：

```
>>> print(list(range(20))) # doctest: +ELLIPSIS
[0, 1, ..., 18, 19]
```

多条指令可用于单条物理线路，用逗号分隔：

```
>>> print(list(range(20))) # doctest: +ELLIPSIS, +NORMALIZE_WHITESPACE
[0,    1, ...,   18,    19]
```

如果单个示例使用多个指令注释，则将它们合并：

```
>>> print(list(range(20))) # doctest: +ELLIPSIS
...                        # doctest: +NORMALIZE_WHITESPACE
[0,    1, ...,   18,    19]
```

如前例所示，您可以将...行添加到仅包含指令的示例中。当一个例子对于指令很容易适合同一行时太长了，这会很有用：

```
>>> print(list(range(5)) + list(range(10, 20)) + list(range(30, 40)))
... # doctest: +ELLIPSIS
[0, ..., 4, 10, ..., 19, 30, ..., 39]
```

请注意，由于默认情况下所有选项都被禁用，并且指令仅适用于它们出现的示例，因此启用选项（通过+指令）通常是唯一有意义的选择。但是，选项标志也可以传递给运行doctests的函数，建立不同的默认值。在这种情况下，通过-指令禁用选项可能很有用。

26.3.3.7。警告

doctest严格要求在预期产出中要求完全匹配。如果即使单个字符不匹配，测试也会失败。这可能会让你感到惊讶，因为你确切地知道Python做了什么，并且不能保证输出。例如，在打印字典时，Python不保证键值对将以任何特定的顺序打印，因此像

```
>>> foo()
{"Hermione": "hippogryph", "Harry": "broomstick"}
```

很脆弱！一种解决方法是做

```
>>> foo() == {"Hermione": "hippogryph", "Harry": "broomstick"}
True
```

代替。另一个是要做的

```
>>> d = sorted(foo().items())
>>> d
[('Harry', 'broomstick'), ('Hermione', 'hippogryph')]
```

还有其他的，但你明白了。

另一个不好的想法是打印嵌入对象地址的东西，比如

```
>>> id(1.0) # certain to fail some of the time
7948648
>>> class C: pass
>>> C()   # the default repr() for instances embeds an address
<__main__.C instance at 0x00AC18F0>
```

该ELLIPSIS指令为最后一个示例提供了一个很好的方法：

```
>>> C() #doctest: +ELLIPSIS
<__main__.C instance at 0x...>
```

浮点数也受到跨平台的小输出变化的影响，因为Python遵循平台C库进行浮点格式化，而C库在质量上差别很大。

```
>>> 1./7  # risky
0.14285714285714285
>>> print(1./7) # safer
0.142857142857
>>> print(round(1./7, 6)) # much safer
0.142857
```

表格I/2.**J中的数字在所有平台上都是安全的，而且我通常会编写一些doctest的例子来生成这种格式的数字：

```
>>> 3./4  # utterly safe
0.75
```

简单的分数对于人们来说也更容易理解，并且这使得更好的文档。

26.3.4。基本

功能testmod()和testfile()提供了一个简单的界面，文档测试，应该是足够了最基本的用途。有关这两个函数的不太正式的介绍，请参见简单用法：检查文档字符串 和简单用法中的示例：检查文本文件中的示例。

doctest.testfile（*文件名*，*module_relative =真*，*名=无*，*包=无*，*水珠=无*，*冗长=无*，*报告=真*，*optionflags = 0*，*extraglobs =无*，*raise_on_error =假*，*解析器= DocTestParser（）*，*编码=无*）

除*文件名*以外的所有参数都是可选的，并应以关键字形式指定。

在名为*filename*的文件中测试示例。返回。(failure_count, test_count)

可选参数*module_relative*指定应如何解释文件名：

- 如果*module_relative*是`True`（缺省值），则*filename*指定一个与操作系统无关的模块相对路径。默认情况下，这个路径是相对于调用模块的目录; 但是如果指定了*package*参数，那么它与该包相关。为了确保OS独立性，*文件名*应该使用 `/`字符来分隔路径段，并且可能不是绝对路径（即它可能不以开头`/`）。
- 如果*module_relative*是`False`，则*filename*指定一个OS特定的路径。路径可以是绝对的或相对的; 相对路径相对于当前工作目录被解析。

可选参数*名称*给出测试的名称; 默认情况下，或者`None`， `os.path.basename(filename)`被使用。

可选参数*包*是一个Python包或一个Python包的名称，其目录应该用作模块相关文件名的基本目录。如果未指定包，则调用模块的目录将用作模块相关文件名的基本目录。如果*module_relative*是指定*包*，那是错误的。`False`

可选参数*globs*在执行示例时给出了一个用作全局变量的字典。这个词典的一个新的浅拷贝是为doctest创建的，所以它的例子从一个干净的石板开始。默认情况下，或者如果`None`使用新的空字典。

可选参数*extraglobs*给出了一个合并到用于执行示例的全局变量中的字典。这适用于`dict.update()`：如果*globs*和 *extraglobs*具有共同的键，则*extraglobs*中的关联值出现在组合字典中。默认情况下，或者如果`None`，不使用额外的全局变量。这是一个允许doctests参数化的高级功能。例如，可以为基类编写一个doctest，使用该类的通用名称，然后通过传递将泛型名称映射到要测试的子类的*extraglobs*字典来测试任意数量的子类。

可选参数*详细信息*如果为true，则会打印大量内容，如果为false，则仅打印失败; 默认情况下，或者如果`None`，当且仅当'`-v`'在时才是`sys.argv`。

可选的参数*报告*在最后打印摘要时为真，否则在最后不打印任何内容。在详细模式下，摘要是详细的，否则摘要非常简短（实际上，如果所有测试都通过，则为空）。

可选参数*optionflags*（默认值0）采用选项标志的 按位或。请参见选项标志部分。

可选参数*raise_on_error*默认为false。如果为true，则在例子中发生第一次失败或意外异常时引发异常。这样可以对故障进行事后调试。默认行为是继续运行示例。

可选参数*解析器*指定`DocTestParser`应该用于从文件中提取测试的一个（或子类）。它默认为一个普通的解析器（即，`DocTestParser()`）。

可选的参数*编码*指定应该用于将文件转换为unicode的编码。

`doctest.testmod`(*m = None*，*name = None*，*globs = None*，*verbose = None*，*report = True*，*optionflags = 0*，*extraglobs = None*，*raise_on_error = False*，*exclude_empty = False*)

所有参数都是可选的，除*m*外的所有参数都应以关键字形式指定。

在从*m开始*可以访问的函数和类中的docstrings中的测试示例（或者`__main__`如果*m*没有提供或者是`None`，则为module），以`m.__doc__`。开头。

还有可以从字典中获得的测试例子`m.__test__`，如果它存在与否`None`。 `m.__test__`将名称（字符串）映射到函数，类和字符串; 函数和类docstrings搜索的例子; 字符串被直接搜索，就好像它们是docstrings一样。

仅搜索附加到属于模块*m*的对象的文档字符串。

返回。`(failure_count, test_count)`

可选参数*名称*给出模块的名称; 默认情况下，或者`None`，`m.__name__`被使用。

可选参数*exclude_empty*默认为false。如果属实，则没有找到doctests的对象将被排除在考虑之外。默认值是向后兼容的黑客，使代码仍然使用`doctest.master.summarize()`连同`testmod()`继续得到输出，没有测试对象。新构造函数的*exclude_empty*参数`DocTestFinder`默认为true。

可选参数*extraglobs*，*verbose*，*report*，*optionflags*，*raise_on_error*和*globs*与`testfile()`上述函数相同，只是*globs*默认为`m.__dict__`。

`doctest.`run_docstring_examples(*f*, *globs*, *verbose* = *False*, *name* =*"NoName"*, *compileflags* = *None*, *optionflags* = *0*)

与对象*f*相关的测试例子; 例如，*f*可以是字符串，模块，函数或类对象。

字典参数*glob*的浅拷贝用于执行上下文。

失败消息中使用可选参数*名称*，缺省值为`"NoName"`。

如果可选参数*verbose*为true，则即使没有失败，也会生成输出。默认情况下，仅在发生示例故障时才会生成输出。

可选参数*compileflags*给出了运行示例时应由Python编译器使用的一组标志。默认情况下，或者如果`None`，推导的标志对应于在*globs*中找到的一组未来特征。

可选参数*optionflags*与`testfile()`上面的函数一样。

26.3.5。单元测试API

随着您的文档测试模块集合的增长，您需要一种系统地运行所有文档测试的方法。 `doctest`提供了两个函数，可用于`unittest`从包含doctests的模块和文本文件创建测试套件。要与`unittest`测试发现集成，`load_tests()`在您的测试模块中包含一个函数：

```
import unittest
import doctest
import my_module_with_doctests

def load_tests(loader, tests, ignore):
    tests.addTests(doctest.DocTestSuite(my_module_with_doctests))
    return tests
```

有两个主要的功能可以用`unittest.TestSuite`文本文件和模块创建实例：

doctest.DocFileSuite(*paths, module_relative=True, package=None, setUp=None, tearDown=None, globs=None, optionflags=0, parser=DocTestParser(), encoding=None)

将doctest测试从一个或多个文本文件转换为 unittest.TestSuite。

返回的unittest.TestSuite内容将由unittest框架运行并在每个文件中运行交互式示例。如果任何文件中的示例失败，则合成的单元测试将失败，并failureException 引发异常，显示包含测试的文件的名称和一个（有时是近似的）行号。

将一个或多个路径（字符串）传递给要检查的文本文件。

选项可以作为关键字参数提供：

可选参数*module_relative*指定 应如何解释*路径中*的文件名：

- 如果*module_relative*是True（缺省值），则*路径*中的每个文件名 指定与操作系统无关的模块相对路径。默认情况下，这个路径是相对于调用模块的目录; 但是如果 指定了*package*参数，那么它与该包相关。为确保操作系统无关性，每个文件名应使用/字符来分隔路径段，并且可能不是绝对路径（即可能不以其开头 / ）。
- 如果*module_relative*是False，则*路径*中的每个文件名都指定一个OS特定的路径。路径可以是绝对的或相对的; 相对路径相对于当前工作目录被解析。

可选参数*包*是Python包或Python包的名称，其目录应该用作*路径中*与模块相关的文件名的基本目录。如果未指定包，则调用模块的目录将用作模块相关文件名的基本目录。如果*module_relative*是 指定*包*，那是错误的。False

可选参数*setUp*指定测试套件的设置函数。这在每个文件中运行测试之前被调用。该*的setUp*函数将被传递一个DocTest对象。setUp函数可以在测试的*globs*属性通过时访问测试全局变量。

可选参数*tearDown*指定测试套件的拆卸函数。这是在每个文件中运行测试后调用的。在*拆卸*会被传递给一个DocTest对象。setUp函数可以在测试的*globs*属性通过时访问测试全局变量。

可选参数*globs*是包含测试的初始全局变量的字典。每个测试都会创建一本新字典。默认情况下，*globs*是一个新的空字典。

可选参数*optionflags*指定测试的默认doctest选项，由各个选项标记组合或创建。请参见选项标志部分 。请参阅set_unittest_reportflags()下面的功能以更好地设置报告选项。

可选参数*解析器*指定DocTestParser应该用于从文件中提取测试的一个（或子类）。它默认为一个普通的解析器（即，DocTestParser()）。

可选的参数*编码*指定应该用于将文件转换为unicode的编码。

全局__file__添加到提供给从文本文件加载的文档测试中使用的全局变量DocFileSuite()。

doctest.DocTestSuite(*module=None, globs=None, extraglobs=None, test_finder=None, setUp=None, tearDown=None, checker=None*)

将模块的doctest测试转换为一个模块unittest.TestSuite。

返回的unittest.TestSuite是由unittest框架运行并在模块中运行每个doctest。如果有任何文档测试失败，则合成的单元测试失败，并failureException引发异常，显示包含测试的文件的名称和一个（有时是近似的）行号。

可选参数*模块*提供要测试的模块。它可以是一个模块对象或一个（可能点缀的）模块名称。如果未指定，则使用调用此函数的模块。

可选参数*globs*是包含测试的初始全局变量的字典。每个测试都会创建一本新字典。默认情况下，*globs*是一个新的空字典。

可选参数*extraglobs*指定一组额外的全局变量，这是合并成的*水珠*。默认情况下，不使用额外的全局变量。

可选参数*test_finder*是DocTestFinder用于从模块中提取doctests 的对象（或插入替换）。

可选参数*setUp*，*tearDown*和*optionflags*与DocFileSuite()上面的函数相同。

此功能使用与搜索技术相同的功能testmod()。

*在版本3.5中更改：*如果*模块不*包含文档字符串而不是引发，则DocTestSuite()返回空白。unittest.TestSuiteValueError

在幕后，DocTestSuite()创建unittest.TestSuite出的doctest.DocTestCase实例，DocTestCase是的子类unittest.TestCase。DocTestCase这里没有记录（这是一个内部细节），但是研究它的代码可以回答关于unittest集成确切细节的问题。

同样，DocFileSuite()创建unittest.TestSuite出的 doctest.DocFileCase实例，DocFileCase是的子类DocTestCase。

因此创建unittest.TestSuite运行实例的 两种方式DocTestCase。这对于一个微妙的原因很重要：当你doctest自己运行 函数时，可以doctest通过将选项标志传递给doctest函数来直接控制正在使用的选项。但是，如果你正在编写一个unittest框架，unittest最终控制何时以及如何运行测试。框架作者通常希望控制 doctest报表选项（可能例如由命令行选项指定），但是没有办法将选项传递unittest给 doctest测试运行者。

出于这个原因，doctest还支持通过此功能支持doctest 特定于unittest支持的报告标志的概念：

doctest.set_unittest_reportflags(*标志*)

设置doctest要使用的报告标志。

参数*标志*采用选项标志的按位或。请参见选项标志部分。只能使用“报告标志”。

这是一个模块全局设置，并影响模块运行的所有将来的doctests unittest：在 实例构建时查看为测试用例指定的选项标记的runTest()方法。如果没有指定报告的标志（这是典型的和预期的情况下），的报告标志是 按位或运算进入选项标志，因此增强选项标志传递给创建运行文档测试实例。如果时指定的任何报告的标志 例如构建，的 报告标志被忽略。DocTestCaseDocTestCasedoctestunittestDocTestRunnerDocTestCasedoctestunittest

unittest在函数被调用之前生效的报告标志的值由函数返回。

26.3.6。高级

基本的API是一个简单的包装，旨在使doctest易于使用。它相当灵活，应该满足大多数用户的需求; 但是，如果您需要对测试进行更精细的控制，或者希望扩展doctest的功能，那么您应该使用高级API。

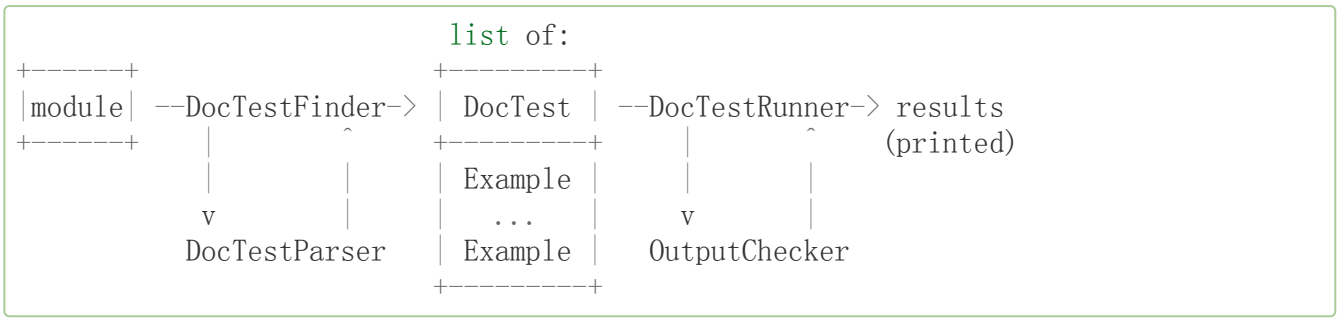
高级API围绕两个容器类进行，这两个容器类用于存储从doctest案例中提取的交互式示例：

- Example：一个Python 语句，与它的预期输出配对。
- DocTest：Examples 的集合，通常从单个文档字符串或文本文件中提取。

定义其他处理类来查找，分析和运行，并检查doctest示例：

- DocTestFinder：查找给定模块中的所有文档字符串，并使用 DocTestParsera DocTest从包含交互式示例的每个文档字符串中创建一个。
- DocTestParser：DocTest从字符串中创建一个对象（例如对象的文档字符串）。
- DocTestRunner：执行a中的例子DocTest，并使用一个OutputChecker来验证它们的输出。
- OutputChecker：将doctest示例中的实际输出与预期输出进行比较，并确定它们是否匹配。

下图总结了这些处理类之间的关系：



26.3.6.1。DocTest对象

*类*doctest.DocTest（*例子*，*globs*，*名称*，*文件名*，*lineno*，*docstring* ）

应该在单个命名空间中运行的doctest示例的集合。构造函数参数用于初始化相同名称的属性。

DocTest定义了以下属性。它们由构造函数初始化，不应该直接修改。

examples

Example编码应该由此测试运行的各个交互式Python示例的对象列表。

globs

应该运行示例的名称空间（又称全局变量）。这是一个将名称映射到值的字典。globs在测试运行之后，示例所做的任何对名称空间的更改（例如绑定新变量）都会反映出来。

name

一个字符串名称标识DocTest。通常，这是测试从中提取的对象或文件的名称。

filename

这DocTest是从中提取的文件的名称; 或者 None如果文件名是未知的，或者如果DocTest没有从文件中提取。

lineno

行号在filename哪里DocTest开始，或 None行号是否不可用。该行号相对于文件的开头是从零开始的。

docstring

从中提取测试None的字符串，或者字符串不可用，或者测试未从字符串中提取。

26.3.6.2。示例对象

class doctest. Example (*source* , *want* , *exc_msg = None* , *lineno = 0* , *indent = 0* , *options = None*)

一个交互式示例，由Python语句及其预期输出组成。构造函数参数用于初始化相同名称的属性。

Example定义了以下属性。它们由构造函数初始化，不应该直接修改。

source

包含示例源代码的字符串。这个源代码由一个Python语句组成，并且总是以换行符结尾; 构造函数在必要时添加一个换行符。

want

运行示例源代码的预期输出（来自标准输出，或者异常情况下的回溯）。 want除非没有输出，否则以换行符结束，在这种情况下，它是一个空字符串。构造函数在必要时添加一个换行符。

exc_msg

该示例生成的异常消息，如果该示例预计会生成异常; 或者None如果不希望产生异常。该异常消息与返回值进行比较 traceback.format_exception_only()。 exc_msg除非是换行符，否则以换行符结尾None。如果需要，构造函数会添加一个换行符。

lineno

包含示例开始处的示例的字符串中的行号。该行号相对于包含字符串的开头是从零开始的。

indent

包含字符串中的示例缩进，即示例第一个提示之前的空格字符数。

options

从选项标记到Trueor的字典映射False，用于覆盖此示例的默认选项。任何未包含在此字典中的选项标志都保留默认值（由DocTestRunners 指定 optionflags ）。默认情况下，不设置任何选项。

26.3.6.3。DocTestFinder对象

class doctest. DocTestFinder（ *verbose = False*，*parser = DocTestParser（ ）*，*recurse = True*，*exclude_empty = True* ）

一个处理类，用于DocTest从文档字符串及其包含对象的文档字符串中提取与给定对象相关的s。 DocTests可以从模块，类，函数，方法，静态方法，类方法和属性中提取。

可选参数*verbose*可用于显示查找器搜索的对象。它默认为False（不输出）。

可选参数*解析器*指定DocTestParser用于从文档字符串中提取文档测试的对象（或插入替换）。

如果可选参数*recurse*为false，那么DocTestFinder.find() 将只检查给定的对象，而不检查任何包含的对象。

如果可选参数*exclude_empty*为false， DocTestFinder.find()则将包含具有空文档字符串的对象的测试。

DocTestFinder 定义了以下方法：

find（ *obj [，name] [，module] [，globs] [，extraglobs]* ）

返回DocTest由*obj*的文档字符串或其包含的任何对象的文档字符串定义的s 的列表。

可选参数*名称*指定对象的名称; 这个名字将被用来为返回的DocTests 构造名字。如果没有指定*名称*，则obj.__name__使用。

可选参数*模块*是包含给定对象的模块。如果模块没有被指定或者是None，则测试发现者将尝试自动确定正确的模块。使用该对象的模块：

- 作为默认命名空间，如果没有指定*globs*。
- 阻止DocTestFinder从其他模块导入的对象中提取DocTests。（包含模块以外的*模块的*包含对象 将被忽略。）
- 查找包含该对象的文件的名称。
- 帮助查找文件中对象的行号。

如果*模块*是False，则不会尝试找到该模块。这是很晦涩的，主要用于测试doctest本身：如果*module*是False，或者是None但不能自动找到，那么所有对象都被认为属于（不存在的）模块，因此所有包含的对象将（递归地）被搜索为doctests。

对于每个全局DocTest通过组合形成*水珠*和 *extraglobs*（在绑定*extraglobs*倍率绑定在*水珠*）。为每个字典创建一个新的globals字典的浅表副本DocTest。如果没有指定*globs*，则默认为模块的*__dict__*（如果指定）或{}以其他方式。如果未指定*extraglobs*，则默认为{}。

26.3.6.4。DocTestParser对象

*类*doctest. DocTestParser

一个处理类，用于从字符串中提取交互式示例，并使用它们创建DocTest对象。

DocTestParser 定义了以下方法：

get_doctest（*字符串*，*globs*，*名称*，*文件名*，*lineno* ）

从给定的字符串中提取所有doctest示例，并将它们收集到一个 DocTest对象中。

globs，*name*，*filename*和*lineno*是新DocTest对象的属性 。请参阅文档以DocTest获取更多信息。

get_examples（*string*，*name* ='<string>' ）

从给定的字符串中提取所有doctest示例，并将它们作为Example对象列表返回。行号是从0开始的。可选参数 *名称*是标识此字符串的名称，仅用于错误消息。

parse（*string*，*name* ='<string>' ）

将给定的字符串分成示例和干预文本，并将它们作为交替Examples和字符串的列表返回。 Examples的行号 是基于0的。可选参数*名称*是标识此字符串的名称，仅用于错误消息。

26.3.6.5。DocTestRunner对象

*类*doctest. DocTestRunner（*checker* = *None*，*verbose* = *None*，*optionflags* = *0* ）

处理类用于执行和验证a中的交互式示例 DocTest。

预期产出与实际产出之间的比较由a OutputChecker。这种比较可以用许多选项标志来定制;有关更多信息，请参阅选项标志部分。如果选项标志不足，则可以通过OutputChecker向构造函数传递一个子类来定制比较。

测试运行者的显示输出可以通过两种方式进行控制。首先，可以传递一个输出函数TestRunner.run()；这个函数将会被显示的字符串调用。它默认为sys.stdout.write。如果捕获的输出不充分，则显示输出也可以通过继承DocTestRunner，并覆盖方法定制report_start()，report_success()，report_unexpected_exception()，和report_failure()。

可选的关键字参数*检查器*指定OutputChecker 应该用于比较预期输出与doctest示例的实际输出的对象（或插入替换）。

可选的关键字参数*verbose*控制着DocTestRunner冗长。如果*详细*是True，则会在每个示例运行时打印信息。如果*详细*是False，则只打印故障。如果*verbose*未指定，或者None使用详细输出，则使用命令行开关-v。

可选的关键字参数*optionflags*可用于控制测试运行器如何将预期输出与实际输出进行比较，以及它如何显示故障。有关更多信息，请参见选项标志部分。

DocTestParser 定义了以下方法：

report_start（*out*，*test*，*example* ）

报告测试运行人员即将处理给出的示例。提供此方法以允许其子类DocTestRunner定制其输出;它不应该直接调用。

*例子*就是要处理的例子。 *测试*是*包含示例*的测试 。 *out*是传递给的输出函数DocTestRunner.run()。

report_success (*out* , *test* , *example* , *got*)

报告给出的示例已成功运行。提供此方法以允许其子类DocTestRunner定制其输出; 它不应该直接调用。

*例子*就是要处理的例子。 *得到的*是实例的实际输出。 *测试*是包含*示例*的测试。 *out*是传递给的输出函数DocTestRunner.run()。

report_failure (*out* , *test* , *example* , *got*)

报告给出的例子失败。提供此方法以允许其子类DocTestRunner定制其输出; 它不应该直接调用。

*例子*就是要处理的例子。 *得到的*是实例的实际输出。 *测试*是包含*示例*的测试。 *out*是传递给的输出函数DocTestRunner.run()。

report_unexpected_exception (*out* , *test* , *example* , *exc_info*)

报告给出的示例引发了意外的异常。提供此方法以允许其子类DocTestRunner定制其输出; 它不应该直接调用。

*例子*就是要处理的例子。*exc_info*是包含有关意外异常（由返回的sys.exc_info()）的信息的元组。 *测试*是包含*示例*的测试。 *out*是传递给的输出函数DocTestRunner.run()。

run (*test* , *compileflags = None* , *out = None* , *clear_globs = True*)

运行在实施例中*测试*（一个DocTest对象），并使用写入器功能显示结果*出来*。

这些示例在命名空间中运行test.globs。如果*clear_globs*为true（缺省值），那么该名称空间将在测试运行后清除，以帮助进行垃圾回收。如果您想在测试完成后检查名称空间，请使用*clear_globs = False*。

*compileflags*给出了运行示例时Python编译器应该使用的一组标志。如果未指定，则它将默认为适用于*globs*的future-import标志集。

每个示例的输出都使用DocTestRunner输出检查器进行检查，并且结果由这些DocTestRunner.report_*()方法进行格式化。

summarize (*verbose = None*)

打印由此DocTestRunner运行的所有测试用例的摘要，并返回一个指定的元组。TestResults(failed, attempted)

可选的*详细*参数控制摘要的详细程度。如果没有指定DocTestRunner详细程度，则使用冗长度。

26.3.6.6。OutputChecker对象

*类*doctest.OutputChecker

用于检查doctest示例的实际输出是否与预期输出匹配的类。 OutputChecker定义了两种方法： check_output()，它比较给定的一对输出，如果匹配则返回真；并output_difference()返回一个描述两个输出之间差异的字符串。

`OutputChecker` 定义了以下方法：

`check_output`（*want*，*got*，*optionflags*）

`True`如果示例（*got*）的实际输出与预期输出（*想要*）匹配，则返回。如果这些字符串相同，则始终认为它们匹配; 但取决于测试运行器使用的选项标志，还可以使用几种非精确匹配类型。有关选项标志的更多信息，请参见选项标志部分 。

`output_difference`（*例如*，*got*，*optionflags*）

返回一个字符串，描述给定示例（*示例*）的预期输出与实际输出（*获得*）之间的差异。 *optionflags*是用来比较*想要*和*得到*的选项标志的集合。

26.3.7。调试

Doctest提供了几种调试doctest示例的机制：

- 几个函数将doctests转换为可执行的Python程序，可以在Python调试器下运行`pdb`。
- 的`DebugRunner`类是的一个子类`DocTestRunner`的是提高用于第一故障的例子的异常，包含有关实施例的信息。该信息可用于对示例执行事后调试。
- `unittest` 通过 `DocTestSuite()` 支持由 `debug()` 所定义的方法生成的案例 `unittest.TestCase`。
- 您可以`pdb.set_trace()`在doctest示例中添加调用，并在执行该行时放入Python调试器。然后你可以检查变量的当前值，等等。例如，假设`a.py` 只包含这个模块docstring：

```
"""
>>> def f(x):
...     g(x*2)
>>> def g(x):
...     print(x+3)
...     import pdb; pdb.set_trace()
>>> f(3)
9
"""
```

然后，一个交互式Python会话可能如下所示：

```
>>> import a, doctest
>>> doctest.testmod(a)
--Return--
> <doctest a[1]>(3)g()->None
-> import pdb; pdb.set_trace()
(Pdb) list
  1     def g(x):
  2         print(x+3)
  3  ->     import pdb; pdb.set_trace()
[EOF]
(Pdb) p x
6
(Pdb) step
--Return--
```

```
> <doctest a[0]>(2)f()->None
-> g(x*2)
(Pdb) list
  1     def f(x):
  2  ->     g(x*2)
[EOF]
(Pdb) p x
3
(Pdb) step
--Return--
> <doctest a[2]>(1)?()->None
-> f(3)
(Pdb) cont
(0, 3)
>>>
```

将doctests转换为Python代码的函数，并可能在调试器下运行综合代码：

doctest.**script_from_examples**(*s*)

将带有示例的文本转换为脚本。

参数*s*是一个包含doctest示例的字符串。该字符串被转换为Python脚本，其中*s*中的doctest示例转换为常规代码，其他所有内容都转换为Python注释。生成的脚本作为字符串返回。例如，

```
import doctest
print(doctest.script_from_examples(r"""
    Set x and y to 1 and 2.
    >>> x, y = 1, 2

    Print their sum:
    >>> print(x+y)
    3
"""))
```

显示：

```
# Set x and y to 1 and 2.
x, y = 1, 2
#
# Print their sum:
print(x+y)
# Expected:
## 3
```

该函数在其他函数的内部使用（请参见下文），但是当您想要将交互式Python会话转换为Python脚本时，该函数也很有用。

doctest.**testsource**(*模块*, *名称*)

将对象的doctest转换为脚本。

参数*模块*是一个模块对象，或者一个模块的虚线名称，包含其文档感兴趣的对象。参数*名称*是具有感兴趣的doctests的对象的名称（在模块内）。结果是一个字符串，包含对象的文

档字符串转换为Python脚本， `script_from_examples()` 如上所述。例如，如果模块`a.py` 包含顶级函数`f()` ，那么

```
import a, doctest
print(doctest.testsource(a, "a.f"))
```

打印函数`f()`的文档字符串的脚本版本，将文档转换为代码，其余部分放在注释中。

`doctest.` debug（*module*，*name*，*pm = False* ）

调试对象的doctests。

该*模块*和*名称*参数是相同的功能 `testsource()` 之上。已命名对象的文档字符串的合成Python脚本被写入临时文件，然后该文件在Python调试器的控制下运行`pdb`。

`module.__dict__`本地和全局执行上下文都使用浅表副本。

可选参数*pm*控制是否使用验尸调试。如果*pm* 具有真值，则脚本文件将直接运行，并且仅当脚本通过引发未处理的异常终止时才会涉及调试器。如果确实如此，则通过`pdb.post_mortem()`从未处理的异常中传递回溯对象来调用验尸调试。如果*pm*没有被指定，或者是false，那么通过传递一个适当的`exec()`调用来从脚本开始运行脚本`pdb.run()`。

`doctest.` debug_src（*src*，*pm = False*，*globs =无* ）

用字符串调试doctests。

这与`debug()`上面的函数类似，只是通过*src*参数直接指定了包含doctest示例的字符串。

可选参数*pm*与`debug()`上面的函数具有相同的含义。

可选的参数*globs*给出了一个字典，用作本地和全局执行上下文。如果未指定，或者`None`使用空字典。如果指定，则使用字典的浅表副本。

该`DebugRunner`级和特殊的例外可能提高，最感兴趣的测试框架的作者，并且只在这里勾勒。查看源代码，尤其`DebugRunner`是docstring（这是一个doctest！）以获取更多详细信息：

类`doctest.` DebugRunner（*checker = None*，*verbose = None*，*optionflags = 0* ）

只要`DocTestRunner`遇到故障，它的一个子类就会引发异常。如果发生意外异常，则会引发`UnexpectedException`异常，包含测试，示例和原始异常。如果输出不匹配，则会`DocTestFailure`引发异常，包含测试，示例和实际输出。

有关构造函数参数和方法的信息，请参阅高级API`DocTestRunner`一节中的文档。

`DebugRunner`实例可能会引发两个例外情况：

异常`doctest.` DocTestFailure（*测试*，*例子*，*得到* ）

`DocTestRunner`表示doctest示例的实际输出与预期输出不符的异常。构造函数参数用于初始化相同名称的属性。

`DocTestFailure` 定义了以下属性：

`DocTestFailure.` test

DocTest示例失败时正在运行的对象。

DocTestFailure.example
的Example失败。

DocTestFailure.got
示例的实际输出。

*异常*doctest.UnexpectedException(*test*，*example*，*exc_info*)
一个异常DocTestRunner提示表示doctest示例引发了意外异常。构造函数参数用于初始化相同名称的属性。

UnexpectedException 定义了以下属性：

UnexpectedException.test
DocTest示例失败时正在运行的对象。

UnexpectedException.example
的Example失败。

UnexpectedException.exc_info
包含有关意外异常的信息的元组，返回的是 sys.exc_info()。

26.3.8。肥皂盒

正如引言中提到的，doctest已经发展到三个主要用途：

1. 检查文档字符串中的示例。
2. 回归测试。
3. 可执行文档/文字测试。

这些用途具有不同的要求，区分它们很重要。特别是，用不明确的测试用例填充文档字符串会导致错误的文档。

在编写文档字符串时，请小心选择文档字符串示例。有一件艺术需要学习 - 起初可能并不自然。示例应该为文档增加真正的价值。一个很好的例子往往可以说很多话。如果谨慎处理，这些示例对您的用户来说将是非常宝贵的，并且会随着时间的推移和事情的变化而回报多次收集它们所需的时间。我仍然惊讶于我的一个doctest 示例在“无害”更改后停止工作的频率。

Doctest也是回归测试的绝佳工具，特别是如果你不吝啬解释性文本。通过插入散文和例子，跟踪实际正在测试的内容以及为什么更容易。当一个测试失败时，好的散文可以使得更容易找出问题所在，以及应该如何解决问题。的确，您可以在基于代码的测试中编写大量的评论，但很少有程序员会这样做。许多人已经发现使用doctest方法会导致更清晰的测试。也许这只是因为doctest使编写散文比编写代码容易一些，而在代码中编写注释有点困难。我认为它比以上更深入：编写基于doctest测试的自然态度是，您想解释软件的优点，并用实例来说明它们。这反过来自然会导致以最简单的功能开始的测试文件，并在逻辑上进展到复杂性和边缘情况。一个连贯的叙述是结果，而不是一组孤立的函数，它们似乎随机地测试孤立的功能位。这是一种不同的态度，产生不同的结果，模糊了测试和解释之间的区别。

回归测试最好局限于专用对象或文件。有几种组织测试的选项：

- 将包含测试用例的文本文件编写为交互式示例，并使用`testfile()`或测试这些文件`DocFileSuite()`。这是推荐的，尽管对于从一开始就使用doctest设计的新项目来说，这是最容易做到的。
- 定义名为`_regrtest_topic`包含单个文档字符串的函数，其中包含指定主题的测试用例。这些功能可以包含在与模块相同的文件中，或者分离到单独的测试文件中。
- 定义`__test__`从回归测试主题到包含测试用例的文档字符串的字典映射。

当您将测试放入模块中时，模块本身可以成为测试运行者。当测试失败时，您可以安排测试运行者在调试问题时仅重新运行失败的doctest。这是一个这样的测试运行者的最小例子：

```
if __name__ == '__main__':
    import doctest
    flags = doctest.REPORT_NDIFF|doctest.FAIL_FAST
    if len(sys.argv) > 1:
        name = sys.argv[1]
        if name in globals():
            obj = globals()[name]
        else:
            obj = __test__[name]
        doctest.run_docstring_examples(obj, globals(), name=name,
                                       optionflags=flags)
    else:
        fail, total = doctest.testmod(optionflags=flags)
        print("{} failures out of {} tests".format(fail, total))
```

脚注

[1] 若支持包含预期输出和异常的测试试图猜测一个结束和另一个开始的地方太容易出

26.4。unittest- 单元测试框架

源代码： Lib / unittest / __ init__.py

（如果您已经熟悉测试的基本概念，则可能需要跳至断言方法列表。）

该unittest单元测试框架最初是由JUnit的启发，也有类似的味道在其他语言主要单元测试框架。它支持测试自动化，共享测试的设置和关闭代码，将测试集合到集合中，以及测试独立于报告框架。

为了实现这一点，unittest以面向对象的方式支持一些重要的概念：

测试夹具
: 甲*测试夹具*表示执行一个或多个测试，以及任何相联清除动作所需要的制剂。例如，这可能涉及创建临时或代理数据库，目录或启动服务器进程。

测试用例
: 甲*测试用例*是测试的各个单元。它检查对特定输入集的特定响应。 unittest提供了一个基类， TestCase它可以用来创建新的测试用例。

测试套件
: 一个*测试套件*是测试案例，测试套件，或两者的集合。它用于聚合应该一起执行的测试。

测试赛跑者
: 甲*测试运行*是编排的测试的执行，并提供结果给用户的部件。跑步者可以使用图形界面，文本界面或返回特殊值来指示执行测试的结果。

也可以看看:

模 doctest
: 另一种测试支持模块具有非常不同的风味。

简单的Smalltalk测试：使用模式
: Kent Beck关于使用共享模式测试框架的原始论文unittest。

鼻子和py.test
: 第三方单元测试框架具有编写测试的轻量级语法。例如，。 `assert func(10) == 42`

Python测试工具分类
: Python测试工具的广泛列表，包括功能测试框架和模拟对象库。

在Python邮件列表中测试
: 用Python讨论测试和测试工具的特殊兴趣小组。

Tools/unittestgui/unittestgui.pyPython源代码发布中的脚本是用于测试发现和执行的GUI工具。这主要是为了便于新手进行单元测试。对于生产环境，建议测试由Buildbot，Jenkins 或 Hudson等持续集成系统驱动 。

26.4.1。基本示例

该unittest模块为构建和运行测试提供了丰富的工具集。本节表明，一小部分工具足以满足大多数用户的需求。

以下是测试三种字符串方法的简短脚本：

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)
```

```
if __name__ == '__main__':
    unittest.main()
```

一个测试用例是通过子类创建的unittest.TestCase。三个单独的测试用名称以字母开头的方法来定义 test。这个命名约定告诉测试运行者哪些方法代表测试。

每个测试的关键在于打电话assertEqual()来检查预期的结果; assertTrue()或assertFalse() 验证条件; 或者assertRaises()验证是否引发了特定的异常。使用这些方法代替 assert语句，因此测试运行者可以累积所有测试结果并生成报告。

该setUp()和tearDown()方法允许你定义将在每次测试方法之后执行的指令。在组织测试代码部分详细介绍了它们。

最后一个块显示了一个运行测试的简单方法。 unittest.main() 为测试脚本提供了一个命令行界面。当从命令行运行时，上面的脚本会产生如下所示的输出：

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

将-v选项传递给测试脚本将指示unittest.main() 启用更高级别的详细程度，并生成以下输出：

```
test_isupper (__main__.TestStringMethods) ... ok
test_split (__main__.TestStringMethods) ... ok
test_upper (__main__.TestStringMethods) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

以上示例显示了最常用的unittest功能，足以满足许多日常测试需求。文档的其余部分从最基本的原则中探索完整的功能集。

26.4.2。命令行界面

可以从命令行使用unittest模块来运行模块，类或甚至单个测试方法的测试：

```
python -m unittest test_module1 test_module2
python -m unittest test_module.TestClass
python -m unittest test_module.TestClass.test_method
```

您可以使用模块名称和完全限定的类或方法名称的任意组合来传递列表。

测试模块也可以通过文件路径指定：

```
python -m unittest tests/test_something.py
```

这使您可以使用shell文件名完成来指定测试模块。指定的文件必须仍然可以作为模块导入。通过删除'.py'并将路径分隔符转换为'.'来将路径转换为模块名称。如果你想执行一个不可导入的测试文件作为模块，你应该直接执行文件。

您可以通过传入-v标志来运行更详细的测试（更高级的冗长）：

```
python -m unittest -v test_module
```

当没有参数执行时测试发现已启动：

```
python -m unittest
```

有关所有命令行选项的列表：

```
python -m unittest -h
```

*在版本3.2中更改：*在早期版本中，只能运行单个测试方法，而不能运行模块或类。

26.4.2.1。命令行选项

unittest支持这些命令行选项：

`-b`, `--buffer`
标准输出和标准错误流在测试运行期间被缓冲。丢弃通过测试期间的输出。输出在测试失败或错误时正常回显并被添加到失败消息中。

`-c`, `--catch`
`Control-C`在测试运行期间，等待当前测试结束，然后报告所有结果。第二个`Control-C`提出了正常的`KeyboardInterrupt`例外。

有关提供此功能的功能，请参阅信号处理。

`-f`, `--failfast`
停止第一次错误或失败的测试运行。

`--locals`
在回溯中显示局部变量。

*新版本3.2：*命令行选项`-b`，`-c`并`-f`添加。

*3.5版新增功能：*命令行选项`--locals`。

命令行也可以用于测试发现，用于运行项目中的所有测试或者只是一个子集。

26.4.3。测试发现

3.2版本中的新功能

Unittest支持简单的测试发现。为了与测试发现兼容，所有测试文件都必须是可从项目顶层目录导入的模块或 包（包括名称空间包）（这意味着它们的文件名必须是有效的标识符）。

测试发现在中实现`TestLoader.discover()`，但也可以从命令行使用。基本的命令行用法是：

```
cd project_directory
python -m unittest discover
```

注意： 作为捷径，`python -m unittest`等同于`python -m unittest discover`。如果您想传递参数以测试发现，则必须明确使用子命令。`discover`

该`discover`子命令有以下选项：

`-v`, `--verbose`
详细输出

`-s`, `--start-directory` `directory`
目录开始发现（`.`默认）

`-p`, `--pattern` `pattern`
匹配测试文件的模式（`test*.py`默认）

`-t`, `--top-level-directory` `directory`
项目的顶级目录（默认为开始目录）

该`-s`，`-p`和`-t`选项可以作为位置参数按顺序传递。以下两条命令行是等效的：

```
python -m unittest discover -s project_directory -p "*_test.py"
python -m unittest discover project_directory "*_test.py"
```

除了作为路径之外，还可以传递包名称`myproject.subpackage.test`作为起始目录。随后将导入您提供的包名称，并将其在文件系统上的位置用作起始目录。

警告： 测试发现通过导入它们来加载测试。一旦测试发现已经从您指定的开始目录中找到所有测试文件，它会将路径转换为要导入的软件包名称。例如`foo/bar/baz.py`将被导入为`foo.bar.baz`。

如果您有一个全局安装的软件包，并尝试在另一个软件包副本上进行测试发现，那么导入*可能*发生在错误的地方。如果发生这种情况，测试发现会警告您并退出。

如果您将开始目录作为软件包名称而不是目录的路径提供，那么discover会假定从其导入的位置是您想要的位置，因此您将不会收到警告。

测试模块和软件包可以通过load_tests协议定制测试加载和发现。

*版本3.4中更改：*测试发现支持命名空间包。

26.4.4。组织测试代码

单元测试的基本构件是*测试用例* - 必须设置和检查正确性的单个场景。在中 unittest ，测试用例由 unittest.TestCase实例表示。要制作自己的测试用例，您必须编写TestCase或使用子类 FunctionTestCase。

TestCase实例的测试代码应该完全自包含，以便它可以单独运行或与任意数量的其他测试用例组合运行。

最简单的TestCase子类将简单地实现一个测试方法（即一个名称以该方法开头的方法test ），以执行特定的测试代码：

```
import unittest

class DefaultWidgetSizeTestCase(unittest.TestCase):
    def test_default_widget_size(self):
        widget = Widget('The widget')
        self.assertEqual(widget.size(), (50, 50))
```

请注意，为了测试某些东西，我们使用基类assert*() 提供的方法之一TestCase。如果测试失败，将会产生一个异常并附带解释性消息，并将unittest 测试用例标识为*失败*。任何其他异常将被视为*错误*。

测试可能很多，并且它们的设置可能是重复的。幸运的是，我们可以通过实现一个叫做的方法来分解设置代码，setUp()测试框架会自动调用我们运行的每一个单独的测试：

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def test_default_widget_size(self):
        self.assertEqual(self.widget.size(), (50,50),
                         'incorrect default size')

    def test_widget_resize(self):
        self.widget.resize(100,150)
        self.assertEqual(self.widget.size(), (100,150),
                         'wrong size after resize')
```

注意： 各种测试的运行顺序通过对测试方法名称与字符串的内置排序进行排序来确定。

如果该setUp()方法在测试运行时引发异常，则框架将认为测试发生了错误，并且测试方法将不会执行。

同样，我们可以提供一种tearDown()方法，在测试方法运行后进行整理：

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def tearDown(self):
        self.widget.dispose()
```

如果setUp()成功， tearDown()将运行测试方法是否成功。

测试代码的这种工作环境称为 *测试夹具*。一个新的TestCase实例被创建为一个独特的测试夹具，用于执行每个单独的测试方法。因此 *TestCase.setUp*， *TestCase.tearDown*，和 *测试用例.__ init__* 将每个测试被调用一次。

建议您使用TestCase实现根据测试的功能将测试分组在一起。 unittest为此提供了一个机制：*测试套件*，由下式表示unittest的 TestSuite类。在大多数情况下，调用unittest.main()会做正确的事情并为您收集所有模块的测试用例并执行它们。

但是，如果您想自定义测试套件的构建，则可以自行完成：

```
def suite():
    suite = unittest.TestSuite()
    suite.addTest(WidgetTestCase('test_default_widget_size'))
    suite.addTest(WidgetTestCase('test_widget_resize'))
    return suite

if __name__ == '__main__':
    runner = unittest.TextTestRunner()
    runner.run(suite())
```

您可以将测试用例和测试套件的定义放置在与测试代码相同的模块中（例如widget.py），但将测试代码放置在单独的模块中有几个优点，例如 test_widget.py：

- 测试模块可以从命令行单独运行。
- 测试代码可以更容易地从提供的代码中分离出来。
- 更改测试代码以适应其测试代码的诱惑没有很好的理由。
- 测试代码应该比它测试的代码更加频繁地被修改。
- 测试过的代码可以更容易地重构。
- 用C编写的模块测试必须在不同的模块中，为什么不一致呢？
- 如果测试策略发生变化，则不需要更改源代码。

26.4.5。重新使用旧的测试代码

有些用户会发现他们有现成的测试代码，他们想从中运行unittest ，而不是将每个旧的测试函数都转换为TestCase子类。

出于这个原因，unittest提供一个FunctionTestCase类。这个子类TestCase可以用来包装现有的测试函数。还可以提供设置和拆卸功能。

鉴于以下测试功能：

```
def testSomething():
    something = makeSomething()
    assert something.name is not None
    # ...
```

可以使用可选的设置和拆卸方法创建如下的等效测试用例实例：

```
testcase = unittest.FunctionTestCase(testSomething,
                                     setUp=makeSomethingDB,
                                     tearDown=deleteSomethingDB)
```

注意： 尽管FunctionTestCase可以用来将现有的测试库快速转换为unittest基于系统，但不建议采用这种方法。花时间设置适当的TestCase 子类将使未来的测试重构变得更容易。

在某些情况下，现有的测试可能是使用该doctest 模块编写的。如果是这样，doctest提供一个DocTestSuite可以unittest.TestSuite从现有doctest测试中自动构建实例 的类。

26.4.6。跳过测试和预期的失败

版本3.1中的新功能。

Unittest支持跳过单独的测试方法甚至整个测试类。此外，它支持将测试标记为“预期失败”，这是一种被破坏并将失败的测试，但不应被视为失败 TestResult。

跳过测试只是使用装饰器 或其一个条件变体的问题。 skip()

基本跳过看起来像这样：

```
class MyTestCase(unittest.TestCase):

    @unittest.skip("demonstrating skipping")
```

```
    def test_nothing(self):
        self.fail("shouldn't happen")

    @unittest.skipIf(mylib.__version__ < (1, 3),
                     "not supported in this library version")
    def test_format(self):
        # Tests that work for only a certain version of the library.
        pass

    @unittest.skipUnless(sys.platform.startswith("win"), "requires Windows")
    def test_windows_support(self):
        # windows specific testing code
        pass
```

这是以详细模式运行上述示例的输出：

```
test_format (__main__.MyTestCase) ... skipped 'not supported in this library version'
test_nothing (__main__.MyTestCase) ... skipped 'demonstrating skipping'
test_windows_support (__main__.MyTestCase) ... skipped 'requires Windows'

----------------------------------------------------------------------
Ran 3 tests in 0.005s

OK (skipped=3)
```

类可以像方法一样跳过：

```
@unittest.skip("showing class skipping")
class MySkippedTestCase(unittest.TestCase):
    def test_not_run(self):
        pass
```

TestCase.setUp()也可以跳过测试。当需要设置的资源不可用时，这非常有用。

预期的故障使用expectedFailure()装饰器。

```
class ExpectedFailureTestCase(unittest.TestCase):
    @unittest.expectedFailure
    def test_fail(self):
        self.assertEqual(1, 0, "broken")
```

通过制作一个装饰器来调用自己的跳过装饰器很容易，该装饰器skip()在需要跳过时调用 测试。除非传递的对象具有特定的属性，否则此装饰器将跳过测试：

```
def skipUnlessHasattr(obj, attr):
    if hasattr(obj, attr):
        return lambda func: func
    return unittest.skip("{!r} doesn't have {!r}".format(obj, attr))
```

以下装饰器实现测试跳过和预期失败：

@unittest.**skip**(*原因*)
    无条件地跳过装饰测试。 *理由*应该描述为什么测试被跳过。

@unittest.**skipIf**(*条件*, *原因*)
    如果*条件*为真，跳过装饰测试。

@unittest.**skipUnless**(*条件*, *原因*)
    除非*条件*为真，否则跳过装饰测试。

@unittest.**expectedFailure**
    将测试标记为预期的失败。如果运行时测试失败，测试不算作失败。

*异常*unittest.**SkipTest**(*原因*)
    这个例外引起了跳过测试。

    通常你可以使用TestCase.skipTest()或者跳过一个装饰器，而不是直接引用它。

跳过的测试不会有setUp()或tearDown()绕过它们。被跳过的班级不会有setUpClass()或tearDownClass()运行。跳过的模块不会有setUpModule()或tearDownModule()运行。

26.4.7。使用子测试区分测试迭代

3.4版新增功能

当你的一些测试仅仅因为一些非常小的差异而不同时，例如一些参数，unittest允许你使用subTest()上下文管理器在测试方法体内区分它们。

例如，下面的测试：

```
class NumbersTest(unittest.TestCase):

    def test_even(self):
        """
        Test that numbers between 0 and 5 are all even.
        """
        for i in range(0, 6):
            with self.subTest(i=i):
                self.assertEqual(i % 2, 0)
```

会产生以下输出：

```
======================================================================
FAIL: test_even (__main__.NumbersTest) (i=1)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0

======================================================================
FAIL: test_even (__main__.NumbersTest) (i=3)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0

======================================================================
FAIL: test_even (__main__.NumbersTest) (i=5)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0
```

如果不使用子测试，执行会在第一次失败后停止，并且错误将不太容易诊断，因为值i 不会显示：

```
======================================================================
FAIL: test_even (__main__.NumbersTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0
```

26.4.8。类和函数

本节将深入介绍API unittest。

26.4.8.1。测试用例

class unittest. TestCase (*methodName ='runTest'*)

TestCase类的实例表示unittest宇宙中的逻辑测试单元。此类旨在用作基类，具体测试由具体的子类实现。该类实现了测试运行器所需的接口，以允许它驱动测试，以及测试代码可用于检查和报告各种故障的方法。

每个实例都TestCase将运行一个基本方法：名为*methodName*的方法。在大多数用途中TestCase，您既不会更改*methodName*也不会重新实现默认runTest()方法。

*在版本3.2中更改：*TestCase可以在不提供*methodName*的情况下成功实例化 。这使得TestCase 从交互式解释器进行试验变得更加容易。

TestCase 实例提供了三组方法：一组用于运行测试，另一组用于由测试实现检查条件和报告失败，还有一些查询方法允许收集有关测试本身的信息。

第一组中的方法（运行测试）是：

setUp ( )

调用方法来准备测试夹具。这在调用测试方法之前立即调用; 除AssertionError或SkipTest ，通过该方法产生的任何异常都将被认为是错误的，而不是一个检测失败。默认实现什么都不做。

tearDown ( )

在调用测试方法后立即调用方法并记录结果。即使测试方法引发异常，也会调用此方法，因此在子类中的实现可能需要特别小心检查内部状态。任何异常，比其他 AssertionError或SkipTest ，通过该方法提出将被视为附加的错误，而不是一个测试失败（从而增加报告的错误的总数）。 setUp()不管测试方法的结果如何，只有成功时才会调用此方法。默认实现什么都不做。

setUpClass ( )

在单个类中的测试之前调用的类方法运行。 setUpClass被称为类作为唯一的参数，并且必须被修饰为classmethod() ：

```
@classmethod
def setUpClass(cls):
    ...
```

有关更多详细信息，请参阅Class和Module Fixtures。

3.2版本中的新功能

tearDownClass ( )

在单个类中的测试之后调用的类方法已经运行。 tearDownClass被称为类作为唯一的参数，并且必须被修饰为classmethod() ：

```
@classmethod
def tearDownClass(cls):
    ...
```

有关更多详细信息，请参阅Class和Module Fixtures。

3.2版本中的新功能

run (*结果=无*)

运行测试，将结果收集到TestResult作为*结果*传递的对象中。如果省略*结果*或None创建临时结果对象（通过调用defaultTestResult() 方法）并使用。结果对象返回给run()调用者。

简单地调用TestCase 实例可能会产生同样的效果。

*在版本3.3中更改：*以前的版本run没有返回结果。调用一个实例也没有。

skipTest (*原因*)

在测试方法中调用它或setUp()跳过当前测试。有关更多信息，请参阅跳过测试和预期故障。

版本3.1中的新功能。

subTest (*msg = None* , *** params*)

返回一个上下文管理器，它执行封装的代码块作为子测试。 *msg*和*params*是可选的，每当子测试失败时显示任意值，允许您清楚地识别它们。

测试用例可以包含任意数量的子测试声明，并且可以任意嵌套。

请参阅使用子测试区分测试迭代以获取更多信息。

3.4版新增功能

debug ( )

运行测试而不收集结果。这允许将测试引发的异常传播给调用者，并且可以用来支持在调试器下运行测试。

本TestCase类提供了一些断言方法来检查并报告故障。下表列出了最常用的方法（有关更多断言方法，请参阅下表）：

方法	检查	新的
assertEqual(a, b)	a == b	
assertNotEqual(a, b)	a != b	
assertTrue(x)	bool(x) is True	
assertFalse(x)	bool(x) is False	
assertIs(a, b)	a is b	3.1
assertIsNot(a, b)	a is not b	3.1
assertIsNone(x)	x is None	3.1
assertIsNotNone(x)	x is not None	3.1
assertIn(a, b)	a in b	3.1
assertNotIn(a, b)	a not in b	3.1
assertIsInstance(a, b)	isinstance(a, b)	3.2
assertNotIsInstance(a, b)	not isinstance(a, b)	3.2

所有的assert方法都会接受一个*msg*参数，如果指定了这个参数，它将被用作失败时的错误消息（另请参见longMessage）。请注意，*味精*关键字参数可以传递给assertRaises()，assertRaisesRegex()，assertWarns()，assertWarnsRegex()只有当它们被用作上下文管理器。

assertEqual（*第一*，*第二*，*味精=无*）

测试*第一*和*第二*是相等的。如果这些值不相等，则测试将失败。

另外，如果*第一个*和*第二个*是完全相同的类型，并且列表，元组，dict，set，frozenset或str中的某一个或子类与addTypeEqualityFunc()类型特定的相等函数一起注册的任何类型将被调用以生成更有用的默认错误信息（另请参阅特定于类型的方法列表）。

*在版本3.1中更改：*添加了特定于类型的相等函数的自动调用。

*在版本3.2中进行了更改：*assertMultiLineEqual()添加为用于比较字符串的默认类型相等函数。

assertNotEqual（*第一*，*第二*，*味精=无*）

测试*第一*和*第二*不相等。如果这些值确实相等，则测试将失败。

assertTrue（*expr*，*msg = None*）
assertFalse（*expr*，*msg = None*）

测试*expr*是否为真（或false）。

请注意，这相当于而不是（用于后者）。当更具体的方法可用时（例如，而不是），这种方法也应该避免，因为它们在失败的情况下提供更好的错误消息。bool(expr) is Trueexpr is TrueassertIs(expr, True)assertEqual(a, b)assertTrue(a == b)

assertIs（*第一*，*第二*，*味精=无*）
assertIsNot（*第一*，*第二*，*味精=无*）

测试*第一个*和*第二个*评估（或不评估）到同一个对象。

版本3.1中的新功能。

assertIsNone（*expr*，*msg = None*）
assertIsNotNone（*expr*，*msg = None*）

测试*expr*是否（或不）None。

版本3.1中的新功能。

assertIn（*第一*，*第二*，*味精=无*）
assertNotIn（*第一*，*第二*，*味精=无*）

*首先*测试（或不）在*第二*。

版本3.1中的新功能。

assertIsInstance（*obj*，*cls*，*msg = None*）
assertNotIsInstance（*obj*，*cls*，*msg = None*）

测试*obj*是否（或不是）*cls的*一个实例（可以是一个类或一个类的元组，可以支持`isinstance()`）。要检查确切的类型，请使用。`assertIs(type(obj), cls)`

3.2版本中的新功能

还可以使用以下方法检查异常，警告和日志消息的生成：

方法	检查	新的
`assertRaises(exc, fun, *args, **kwds)`	`fun(*args, **kwds)`引发了*不满*	
`assertRaisesRegex(exc, r, fun, *args, **kwds)`	`fun(*args, **kwds)`引发*exc*并且消息匹配正则表达式*r*	3.1
`assertWarns(warn, fun, *args, **kwds)`	`fun(*args, **kwds)`引发*警告*	3.2
`assertWarnsRegex(warn, r, fun, *args, **kwds)`	`fun(*args, **kwds)`引发*警告*并且消息匹配正则表达式*r*	3.2
`assertLogs(logger, level)`	该`with`块以最低*级别*登录*记录器*	3.4

assertRaises（*异常*，*可调用*，**args*，***kwds*）
assertRaises（*例外*，*msg =无*）

测试在*调用callable*时使用传递给它的任何位置或关键字参数时会引发异常`assertRaises()`。如果发生*异常*，则测试通过;如果发生另一个异常，则为错误;如果未发生异常，则发生错误。要捕获任何一组异常，包含异常类的元组可以作为*例外*传递。

如果只给出*异常*和可能的*msg*参数，则返回一个上下文管理器，以便测试下的代码可以内联写入而不是作为函数写入：

```
with self.assertRaises(SomeException):
    do_something()
```

当用作上下文管理器时，`assertRaises()`接受附加关键字参数*msg*。

上下文管理器将把捕获到的异常对象存储在它的`exception`属性中。如果打算对引发的异常执行附加检查，这可能很有用：

```
with self.assertRaises(SomeException) as cm:
    do_something()

the_exception = cm.exception
self.assertEqual(the_exception.error_code, 3)
```

*在版本3.1中进行了更改：*添加了`assertRaises()`用作上下文管理器的功能。

*在版本3.2中更改：*添加了该`exception`属性。

*在版本3.3中更改：*用作上下文管理器时添加了*msg*关键字参数。

assertRaisesRegex（*异常*，*正则表达式*，*可调用*，**args*，***kwds*）
assertRaisesRegex（*例外*，*正则表达式*，*msg =无*）

像`assertRaises()`但也测试*正则表达式*匹配的引发异常的字符串表示。*正则表达式*可能是一个正则表达式对象，或者是一个包含正则表达式的字符串，适合使用`re.search()`。例子：

```
self.assertRaisesRegex(ValueError, "invalid literal for.*XYZ'$",
                       int, 'XYZ')
```

要么：

```
with self.assertRaisesRegex(ValueError, 'literal'):
   int('XYZ')
```

*新版本3.1：*名称下`assertRaisesRegexp`。

*在版本3.2中更改：*重命名为`assertRaisesRegex()`。

*在版本3.3中更改：*用作上下文管理器时添加了*msg*关键字参数。

assertWarns（*警告*，*可调用*，**args*，***kwds*）
assertWarns（*警告*，*味精=无*）

测试在*调用callable*时使用传递给它的任何位置或关键字参数 会触发警告assertWarns()。如果*警告*被触发，则测试通过，否则失败。任何异常都是错误。要捕获任何一组警告，包含警告类的元组可以作为*警告*传递。

如果只给出*警告*和可能的*msg*参数，则返回一个上下文管理器，以便可以将内联代码写入测试代码，而不是作为函数：

```
with self.assertWarns(SomeWarning):
    do_something()
```

当用作上下文管理器时，assertWarns()接受附加关键字参数*msg*。

上下文管理器将存储在其陷入警告对象 warning的属性，以及源线，引发了警告filename和lineno属性。如果打算对捕获的警告执行附加检查，这可能很有用：

```
with self.assertWarns(SomeWarning) as cm:
    do_something()

self.assertIn('myfile.py', cm.filename)
self.assertEqual(320, cm.lineno)
```

无论调用何种警告过滤器，该方法都可以工作。

3.2版本中的新功能

*在版本3.3中更改：*用作上下文管理器时添加了*msg*关键字参数。

assertWarnsRegex (*警告*，*正则表达式*，*可调用*，** args*，*** kwds*)
assertWarnsRegex (*警告*，*正则表达式*，*味精=无*)

像assertWarns()但也测试*正则表达式*匹配触发警告的消息。 *正则表达式*可能是一个正则表达式对象，或者是一个包含正则表达式的字符串，适合使用re.search()。例：

```
self.assertWarnsRegex(DeprecationWarning,
                      r'legacy_function\(\) is deprecated',
                      legacy_function, 'XYZ')
```

要么：

```
with self.assertWarnsRegex(RuntimeWarning, 'unsafe frobnicating'):
    frobnicate('/etc/passwd')
```

3.2版本中的新功能

*在版本3.3中更改：*用作上下文管理器时添加了*msg*关键字参数。

assertLogs (*logger = None*，*level = None*)

上下文管理器用于测试记录*器*或其子项之一是否至少记录了一条消息，并至少具有给定的 *级别*。

如果给出，*记录器*应该是一个logging.Logger对象或 str给出记录器的名称。缺省值是根记录器，它将捕获所有消息。

如果给定，*级别*应该是数字日志记录级别或其等效字符串（例如，"ERROR"或 logging.ERROR）。默认是logging.INFO。

如果with 块内至少有一条消息与*记录器*和*级别*条件匹配，则测试通过，否则失败。

上下文管理器返回的对象是一个记录帮助程序，用于跟踪匹配的日志消息。它有两个属性：

records

logging.LogRecord匹配日志消息的对象列表。

output

str具有匹配消息格式化输出的对象列表。

例：

```
with self.assertLogs('foo', level='INFO') as cm:
   logging.getLogger('foo').info('first message')
   logging.getLogger('foo.bar').error('second message')
```

```
self.assertEqual(cm.output, ['INFO:foo:first message',
                             'ERROR:foo.bar:second message'])
```

3.4版新增功能

还有其他方法用于执行更具体的检查，例如：

方法	检查	新的
assertAlmostEqual(a, b)	round(a-b, 7) == 0	
assertNotAlmostEqual(a, b)	round(a-b, 7) != 0	
assertGreater(a, b)	a > b	3.1
assertGreaterEqual(a, b)	a >= b	3.1
assertLess(a, b)	a < b	3.1
assertLessEqual(a, b)	a <= b	3.1
assertRegex(s, r)	r.search(s)	3.1
assertNotRegex(s, r)	not r.search(s)	3.2
assertCountEqual(a, b)	*a*和*b*在相同数字中具有相同的元素，而不管它们的顺序如何	3.2

assertAlmostEqual (*第一*，*第二*，*places = 7*，*msg = None*，*delta = None*)
assertNotAlmostEqual (*第一*，*第二*，*places = 7*，*msg = None*，*delta = None*)

测试*第一*和*第二*近似（或不近似）通过计算差，四舍五入到小数点的给定数目相等*的地方*（默认7），并与零进行比较。请注意，这些方法将数值四舍五入到给定的*小数位数*（即round()函数）而不是*有效数字*。

如果提供*增量*而不是*位置*，则*第一个*和*第二个*之间的差值必须小于或等于（或大于）*delta*。

提供*三角洲*和*地方*提出了一个TypeError。

*在版本3.2中更改：*assertAlmostEqual()自动考虑几乎相等的比较相等的对象。 assertNotAlmostEqual()如果对象相等则自动失败。添加了*delta*关键字参数。

assertGreater (*第一*，*第二*，*味精=无*)
assertGreaterEqual (*第一*，*第二*，*味精=无*)
assertLess (*第一*，*第二*，*味精=无*)
assertLessEqual (*第一*，*第二*，*味精=无*)

根据方法名称测试*第一个*分别是>，> =，<或<= *第二个*。如果不是，测试将失败：

```
>>> self.assertGreaterEqual(3, 4)
AssertionError: "3" unexpectedly not greater than or equal to "4"
```

版本3.1中的新功能。

assertRegex (*文本*，*正则表达式*，*msg =无*)
assertNotRegex (*文本*，*正则表达式*，*msg =无*)

测试*正则表达式*搜索匹配（或不匹配）*文本*。如果失败，错误消息将包括模式和*文本*（或模式和意外匹配的*文本*部分）。 *正则表达式*可能是一个正则表达式对象，或者是一个包含正则表达式的字符串，适合使用re.search()。

*新版本3.1：*名称下assertRegexpMatches。

*在版本3.2中更改：*该方法assertRegexpMatches()已重命名为 assertRegex()。

*3.2版新增功能：*assertNotRegex()。

*3.5版本中的新功能：*该名称assertNotRegexpMatches是不推荐的别名assertNotRegex()。

assertCountEqual (*第一*，*第二*，*味精=无*)

无论顺序如何，*首先*测试序列都包含与*第二个*相同的元素。如果没有，则会生成列出序列之间差异的错误消息。

比较*第一*和 *第二*时，重复元素*不会*被忽略。它验证每个元素在两个序列中是否具有相同的计数。等价于： 但也适用于不可对象序列。assertEqual(Counter(list(first)), Counter(list(second)))

3.2版本中的新功能

该assertEqual()方法将相同类型的对象的相等性检查分派给不同类型特定的方法。这些方法已经用于大多数内置类型，但也可以使用addTypeEqualityFunc()以下方法注册新方法：

addTypeEqualityFunc（*typeobj*，*function*）

注册一个名为by的类型特定方法assertEqual()来检查两个完全相同的*typeobj*（而不是子类）对象是否相等。 *函数必须像两个位置参数一样，第三个msg = None关键字参数*assertEqual()。self.failureException(msg)当检测到前两个参数之间的不平等时，它必须提高 - 可能提供有用的信息并解释错误消息中的细节不等式。

版本3.1中的新功能。

assertEqual()下表中汇总了自动使用的特定于类型的方法 列表。请注意，通常不需要直接调用这些方法。

方法	用于比较	新的
assertMultiLineEqual(a, b)	字符串	3.1
assertSequenceEqual(a, b)	序列	3.1
assertListEqual(a, b)	名单	3.1
assertTupleEqual(a, b)	元组	3.1
assertSetEqual(a, b)	设置或frozensets	3.1
assertDictEqual(a, b)	http://stardict.sourceforge.net/Dictionaries.php下载	3.1

assertMultiLineEqual（*第一*，*第二*，*味精=无*）

测试了多串*首先*是等于字符串*第二*。如果不等于两个字符串的差异，则突出显示差异将包含在错误消息中。比较字符串时默认使用此方法assertEqual()。

版本3.1中的新功能。

assertSequenceEqual（*第一*，*第二*，*msg =无*，*seq_type =无*）

测试两个序列是否相等。如果提供了*seq_type*，则 *第一个*和*第二个必须是seq_type的*实例，*否则*将引发失败。如果序列不同，则会构造一个错误消息，显示两者之间的差异。

这个方法不是直接被调用assertEqual()，而是用来实现assertListEqual()和 assertTupleEqual()。

版本3.1中的新功能。

assertListEqual（*第一*，*第二*，*味精=无*）
assertTupleEqual（*第一*，*第二*，*味精=无*）

测试两个列表或元组是否相等。如果不是，则会构建一条仅显示两者之间差异的错误消息。如果两个参数中的任何一个参数都属于错误类型，也会引发错误。在比较列表或元组时，默认使用这些方法assertEqual()。

版本3.1中的新功能。

assertSetEqual（*第一*，*第二*，*味精=无*）

测试两组相等。如果不是，则会构造一个错误消息，列出各组之间的差异。比较设置或frozensets时默认使用此方法assertEqual()。

如果*第一个*或*第二个*没有set.difference()方法，则失败。

版本3.1中的新功能。

assertDictEqual（*第一*，*第二*，*味精=无*）

测试两个字典是否相同。如果不是，则会构造一个错误消息，显示字典中的差异。默认情况下会使用此方法比较调用中的字典assertEqual()。

版本3.1中的新功能。

最后TestCase提供以下方法和属性：

fail（*msg =无*）

用*msg*或None错误信息无条件地表示测试失败。

failureException

这个类属性给出了测试方法引发的异常。如果一个测试框架需要使用一个专门的异常，可能需要携带更多的信息，那么它必须将这个异常小类化，以便与框架“公平”。这个属性的初始值是 AssertionError。

longMessage

此类属性确定将自定义失败消息作为msg参数传递给发生故障的assertXYY调用时发生的情况。 True是默认值。在这种情况下，自定义消息被附加到标准失败消息的末尾。设置False为时，自定义消息将替换标准消息。

通过在调用assert方法之前True或False之前分配一个实例属性self.longMessage，可以在各个测试方法中重写类设置。

在每次测试呼叫之前，课程设置会重置。

版本3.1中的新功能。

maxDiff

该属性通过assert方法控制差异输出的最大长度，该方法报告差异的差异。它默认为80 * 8个字符。受此属性影响的Assert方法 assertSequenceEqual() （包括委托给它的所有序列比较方法）assertDictEqual() 和 assertMultiLineEqual()。

设置maxDiff为None意味着没有最大长度的差异。

3.2版本中的新功能

测试框架可以使用以下方法收集测试信息：

countTestCases( )

返回此测试对象表示的测试数量。对于 TestCase情况下，这将永远是1。

defaultTestResult( )

返回应该用于此测试用例类的测试结果类的实例（如果没有其他结果实例提供给该 run()方法）。

对于TestCase情况下，这将永远是一个实例 TestResult; 子类TestCase应该根据需要覆盖它。

id( )

返回一个标识特定测试用例的字符串。这通常是测试方法的全名，包括模块和类名。

shortDescription( )

返回测试的描述，或者None没有提供描述。此方法的默认实现返回测试方法的文档字符串的第一行（如果可用）或None。

*在版本3.1中进行了更改：*在3.1中，即使在存在文档字符串的情况下，也更改为将测试名称添加到简短描述中。这导致了unittest扩展的兼容性问题，并且将测试名称添加到了 TextTestResultPython 3.2中。

addCleanup(*函数*, ** args*, *** kwargs*)

添加一个函数来tearDown()清理测试过程中使用的资源。函数将按照它们添加的顺序（LIFO）以相反的顺序调用。它们被调用时添加了任何参数和关键字参数 addCleanup()。

如果setUp()失败，意味着tearDown()没有被调用，那么添加的任何清理函数仍将被调用。

版本3.1中的新功能。

doCleanups( )

此方法在之后tearDown()或之后被无条件地调用， setUp()如果setUp()引发异常。

它负责调用所添加的所有清理函数 addCleanup()。如果你需要清除函数被调用 *之前*到tearDown()，那么你可以调用doCleanups() 自己。

doCleanups() 一次只能抛出一堆清理函数，因此可以随时调用它。

版本3.1中的新功能。

class unittest.FunctionTestCase(*testFunc*, *setUp = None*, *tearDown = None*, *description = None*)

该类实现了TestCase允许测试运行器驱动测试的接口部分，但不提供测试代码可用于检查和报告错误的方法。这用于使用旧版测试代码创建测试用例，从而将其集成到 unittest基于测试框架的测试中。

26.4.8.1.1。弃用别名

由于历史原因，其中一些TestCase方法有一个或多个现在已被弃用的别名。下表列出了正确的名称及其弃用的别名：

方法名称	弃用的别名	弃用的别名
assertEqual()	failUnlessEqual	的assertEquals
assertNotEqual()	failIfEqual	assertNotEquals
assertTrue()	failUnless	断言_
assertFalse()	failIf	
assertRaises()	failUnlessRaises	
assertAlmostEqual()	failUnlessAlmostEqual	assertAlmostEquals
assertNotAlmostEqual()	failIfAlmostEqual	assertNotAlmostEquals
assertRegex()		assertRegexpMatches
assertNotRegex()		assertNotRegexpMatches
assertRaisesRegex()		assertRaisesRegexp

*自3.1版弃用：*第二列中列出的失败*别名。

*自3.2版弃用：*第三列中列出的断言*别名。

自3.2版以来已弃用： assertRegexpMatches 并 assertRaisesRegexp 已重命名为 assertRegex() 和 assertRaisesRegex()。

从版本3.5开始弃用： assertNotRegexpMatches赞成的名称assertNotRegex()。

26.4.8.2。分组测试

class unittest. TestSuite (*tests = ( )*)

这个类代表了单个测试用例和测试套件的集合。该类提供了测试运行器所需的接口，以使其可以像其他任何测试用例一样运行。运行TestSuite实例与遍历套件相同，单独运行每个测试。

如果给出了*测试*，它必须是可用于最初构建套件的单个测试用例或其他测试套件的迭代。还提供了其他方法，以便稍后将测试用例和套件添加到集合中。

TestSuite对象的行为与TestCase对象非常相似，只是它们并未实际执行测试。相反，它们用于将测试聚合到应该一起运行的测试组中。一些额外的方法可用于向TestSuite实例添加测试：

addTest (*测试*)

添加TestCase或TestSuite套件。

addTests (*测试*)

将迭代TestCase和TestSuite 实例中的所有测试添加到此测试套件。

这相当于迭代*测试*，调用addTest()每个元素。

TestSuite共享以下方法TestCase：

run (*结果*)

运行与此套件相关的测试，将结果收集到作为*结果*传递的测试结果对象中。请注意，与之不同TestCase.run()，TestSuite.run()需要传入结果对象。

debug ( )

运行与此套件相关的测试而不收集结果。这允许将测试引发的异常传播给调用者，并且可以用来支持在调试器下运行测试。

countTestCases ( )

返回此测试对象表示的测试数量，包括所有单个测试和子套件。

__iter__ ( )

按a分组的测试TestSuite总是通过迭代进行访问。子类可以懒惰地提供覆盖测试__iter__()。请注意，可以在单个套件上多次调用此方法（例如，在计算测试或比较相等时），因此TestSuite.run()对于每次调

用迭代，重复迭代返回的测试必须相同。之后TestSuite.run()，调用者不应该依赖此方法返回的测试，除非调用者使用覆盖TestSuite._removeTestAtIndex()保留测试引用的子类。

*在版本3.2中更改：*在早期版本中，TestSuite直接访问测试而不是迭代，因此覆盖__iter__()不足以提供测试。

*在版本3.4中更改：*在早期版本中TestSuite，每个TestCase之后都保存引用 TestSuite.run()。子类可以通过重写来恢复该行为TestSuite._removeTestAtIndex()。

在TestSuite对象的典型用法中，该run()方法由一个TestRunner而不是由最终用户测试工具调用。

26.4.8.3。加载和运行测试

*类*unittest.TestLoader

所述TestLoader类被用来创建从类和模块的测试套件。通常，不需要创建该类的实例; 该 unittest模块提供了一个可以共享的实例 unittest.defaultTestLoader。但是，使用子类或实例可以自定义一些可配置的属性。

TestLoader 对象具有以下属性：

errors

加载测试时遇到的非致命错误列表。在任何时候都不由加载器重置。致命错误通过相关的一种方法向发起方提出异常来发出信号。非致命错误也可以通过合成测试来指示，它会在运行时增加原始错误。

3.5版本中的新功能。

TestLoader 对象有以下方法：

loadTestsFromTestCase (*testCaseClass*)

返回一个套件中包含的所有测试用例TestCase派生 testCaseClass。

为每个由名为的方法创建一个测试用例实例 getTestCaseNames()。默认情况下，这些是以test。开头的方法名称。如果不getTestCaseNames()返回任何方法，但runTest()实现了该方法，则会为该方法创建一个测试用例。

loadTestsFromModule (*module*，*pattern = None*)

返回给定模块中包含的所有测试用例套件。此方法在*模块*中搜索派生的类，TestCase并为为该类定义的每个测试方法创建类的实例。

> **注意：** 虽然使用TestCase衍生类的层次结构可以方便地共享fixtures和helper函数，但是在基类上定义测试方法并不打算直接实例化，但这种方法并不能很好地发挥作用。但是，如果灯具不同并在子类中定义，则这样做会很有用。

如果一个模块提供了一个load_tests函数，它将被调用来加载测试。这允许模块自定义测试加载。这是load_tests协议。的*图案*参数作为第三个参数来传递load_tests。

*在版本3.2中更改：*支持load_tests添加。

*在版本3.5中进行了更改：不合法的*和非官方的*use_load_tests*默认参数已被弃用并被忽略，尽管它仍被接受用于向后兼容。该方法现在还接受仅作为第三个参数传递给的关键字参数 *模式*load_tests。

loadTestsFromName (*name*，*module = None*)

返回给定字符串说明符的所有测试用例。

指定器*名称*是“带点名称”，其可以解决要么模块，测试用例类，测试用例类内的测试法， TestSuite实例或它返回一个可调用对象 TestCase或TestSuite实例。这些检查按照此处列出的顺序应用; 也就是说，可能的测试用例类中的方法将被选为“测试用例类中的测试方法”，而不是“可调用对象”。

例如，如果您的模块SampleTests包含带有三种测试方法（，和）的 TestCase派生类，则说明符会导致此方法返回一个将运行所有三种测试方法的套件。使用说明符 会导致它返回一个只运行测试方法的测试套件。说明符可以引用尚未导入的模块和包；他们将作为副作用进口。SampleTestCase test_one() test_two() test_three()’SampleTests.SampleTestCase’’SampleTests.SampleTestCase.test_two’test_two()

该方法可以选择性地解析相对于给定*模块的名称*。

*在版本3.5中更改：*如果在遍历*名称*时发生ImportError或AttributeError发生，则会返回运行时引发该错误的综合测试。这些错误包含在self.errors累积的错误中。

loadTestsFromNames（*names*，*module* = None ）

类似于loadTestsFromName()，但是采用一系列名称而不是单一名称。返回值是一个测试套件，它支持为每个名称定义的所有测试。

getTestCaseNames（*testCaseClass* ）

返回在*testCaseClass*中找到的方法名称的排序顺序; 这应该是一个子类TestCase。

discover（*start_dir*，*pattern* ='test * .py'，*top_level_dir* = None ）

通过从指定的开始目录递归到子目录中查找所有测试模块，并返回包含它们的TestSuite对象。只有与*模式*匹配的测试文件才会被加载。（使用shell风格模式匹配。）只有可导入的模块名称（即有效的Python标识符）才会被加载。

所有测试模块必须可以从项目的顶层导入。如果起始目录不是顶层目录，则顶层目录必须单独指定。

如果导入模块失败，例如由于语法错误，那么这将被记录为单个错误，并且发现将继续。如果导入失败是由于SkipTest引发，它将被记录为跳过而不是错误。

如果找到一个包（包含一个名为文件的目录__init__.py），则会检查该包是否有load_tests函数。如果存在，则会被调用 。测试发现需要注意的是，即使load_tests函数本身调用，也只能确保在调用期间仅对包进行检查 。package.load_tests(loader, tests, pattern)loader.discover

如果load_tests存在，那么发现并*没有*递归放入包中，load_tests负责加载包中的所有测试。

该模式故意不作为加载程序属性存储，以便程序包可以继续自行发现。存储*top_level_dir*，因此load_tests不需要将此参数传入 loader.discover()。

*start_dir*可以是虚线模块名称以及目录。

3.2版本中的新功能

*在版本3.4中更改：*SkipTest在导入时引发的模块被记录为跳过，而不是错误。发现适用于名称空间包。即使基础文件系统的排序不依赖于文件名，路径也会在导入之前进行排序，以便执行顺序相同。

*版本3.5中已更改：*现在检查发现的包load_tests，无论其路径是否匹配*模式*，因为包名称不可能匹配默认模式。

a的以下属性TestLoader可以通过子类或实例上的赋值来配置：

testMethodPrefix

提供方法名称前缀的字符串，将被解释为测试方法。默认值是'test'。

这影响getTestCaseNames()和所有的loadTestsFrom*() 方法。

sortTestMethodsUsing

函数用于在对它们进行排序时比较方法名称 getTestCaseNames()以及所有loadTestsFrom*()方法。

suiteClass

从测试列表构造测试套件的可调用对象。在结果对象上不需要任何方法。默认值是 TestSuite类。

这影响了所有的loadTestsFrom*()方法。

*类*unittest.TestResult

该类用于编译有关哪些测试已成功并失败的信息。

一个TestResult对象存储一组测试的结果。在 TestCase和TestSuite班保证结果正确记录; 测试作者不需要担心记录测试的结果。

构建在最上面的测试框架unittest可能需要访问TestResult为报告目的运行一组测试所生成的 对象; 为此目的TestResult，TestRunner.run()方法返回一个实例 。

TestResult 实例具有以下在检查运行一组测试的结果时将会感兴趣的属性：

errors

包含TestCase实例和字符串的2元组的列表，其中包含格式化的回溯。每个元组代表一个引发意外异常的测试。

failures

包含TestCase实例和字符串的2元组的列表，其中包含格式化的回溯。每个元组代表一个测试，其中使用这些TestCase.assert*()方法显式地发送了失败信号。

skipped
包含TestCase实例和字符串的2元组的列表，其中包含跳过测试的原因。

版本3.1中的新功能。

expectedFailures
包含TestCase实例和字符串的2元组的列表，其中包含格式化的回溯。每个元组代表测试用例的预期失败。

unexpectedSuccesses
包含TestCase标记为预期失败但成功的实例的列表。

shouldStop
设置为True执行测试时应该停止stop()。

testsRun
迄今为止运行的测试总数。

buffer
如果设置为true，sys.stdout并且sys.stderr将在之间进行缓冲 startTest()和stopTest()被调用。收集到的输出只会回显到真实状态sys.stdout，sys.stderr如果测试失败或错误。任何输出也附加到失败/错误消息。

3.2版本中的新功能

failfast
如果设置为true，stop()则会在第一次失败或错误时调用，从而停止测试运行。

3.2版本中的新功能

tb_locals
如果设置为true，则局部变量将显示在回溯中。

3.5版本中的新功能。

wasSuccessful()
返回True如果所有测试跑这么远都过去了，否则返回 False。

*在版本3.4中更改：*返回False是否有unexpectedSuccesses 来自expectedFailure()装饰器标记的测试。

stop()
可以调用此方法来指示正在运行的测试集应通过设置shouldStop属性来中止True。 TestRunner对象应该尊重这个标志并返回而不需要运行任何附加测试。

例如，TextTestRunner当用户通过键盘发出中断时，该类将使用此功能来停止测试框架。提供TestRunner实现的交互式工具可以以类似的方式使用它。

TestResult该类的以下方法用于维护内部数据结构，并可以在子类中进行扩展以支持其他报告要求。这在构建支持交互式报告而运行测试的工具时特别有用。

startTest(*测试*)
当测试用例*测试*即将运行时调用。

stopTest(*测试*)
无论结果如何，在*测试用例测试*执行后调用。

startTestRun()
在执行任何测试之前调用一次。

版本3.1中的新功能。

stopTestRun()
所有测试执行完毕后调用一次。

版本3.1中的新功能。

addError (*test* , *err*)

当测试用例*测试*引发意外的异常时调用。*err*是由以下形式返回的形式的元组sys.exc_info() ：。(type, value, traceback)

默认实现将元组附加到实例的属性，其中*formatted_err*是从*err*派生的格式化回溯。(test, formatted_err) errors

addFailure (*test* , *err*)

当测试用例*测试*指示失败时调用。*err*是由以下形式返回的形式的元组sys.exc_info() ：。(type, value, traceback)

默认实现将元组附加到实例的属性，其中*formatted_err*是从*err*派生的格式化回溯。(test, formatted_err) failures

addSuccess (*测试*)

当测试用例*测试*成功时调用。

默认实现什么都不做。

addSkip (*测试* , *原因*)

当测试用例*测试*被跳过时调用。 *原因*是测试给跳过的原因。

默认实现将元组附加到实例的属性。(test, reason) skipped

addExpectedFailure (*test* , *err*)

当测试用例*测试*失败时调用，但用expectedFailure()装饰器标记 。

默认实现将元组附加到实例的属性，其中*formatted_err* 是从*err*派生的格式化回溯。(test, formatted_err) expectedFailures

addUnexpectedSuccess (*测试*)

当测试用例*测试*用expectedFailure()装饰器标记 但成功时调用。

默认实现将测试附加到实例的 unexpectedSuccesses属性。

addSubTest (*测试* , *子测试* , *结果*)

当分测试结束时调用。 *测试*是与测试方法相对应的测试用例。 *subtest*是TestCase描述子测试的自定义 实例。

如果*结果*是None ，分测试成功了。否则，如果*结果*是由以下形式返回的表单的元组，则会失败sys.exc_info() ：。(type, value, traceback)

当结果成功时，默认实现不执行任何操作，并将子测试失败记录为正常失败。

3.4版新增功能

*类*unittest. TextTestResult (*流* , *描述* , *详细*)

一个具体的实现TestResult使用的 TextTestRunner。

*3.2版本中的新功能：*此类是以前命名的_TextTestResult。旧名称仍作为别名存在，但不推荐使用。

unittest. defaultTestLoader

TestLoader该类的实例打算共享。如果不需要自定义，TestLoader则可以使用此实例，而不是重复创建新实例。

class unittest. TextTestRunner (*stream = None* , *descriptions = True* , *verbosity = 1* , *failfast = False* , *buffer = False* , *resultclass = None* , *warnings = None* , *** , *tb_locals = False*)

将结果输出到流的基本测试运行器实现。如果*stream* 是None默认值， sys.stderr则用作输出流。这个类有几个可配置参数，但实质上非常简单。运行测试套件的图形应用程序应提供替代实现。**kwargs当功能添加到unittest时，这些实现应该接受构造跑步者的接口。

默认情况下，这个亚军显示DeprecationWarning ， PendingDeprecationWarning ， ResourceWarning 和ImportWarning即使他们是默认被忽略掉。弃用的单元测试方法导致的弃用警告也是特殊情况，并且当警告过滤器是'default'或时'always' ，它们将仅出现一次每个模块，以避免太多警告消息。这种行为可以使用Python的覆盖-Wd或-Wa选项（见警告控制 ）和离开 *的警告*来None。

*在版本3.2中更改：*添加了warnings参数。

*在版本3.2中更改：*默认流设置为sys.stderr实例化时间而不是导入时间。

*版本3.5中已更改：*添加了tb_locals参数。

_makeResult（）

此方法返回的实例TestResult所使用run()。它不打算直接调用，但可以在子类中重写以提供自定义TestResult。

_makeResult()将TextTestRunner构造函数中传递的类或可调用实例化为resultclass参数。TextTestResult如果没有resultclass提供，它默认为。结果类用以下参数实例化：

```
stream, descriptions, verbosity
```

run（*测试*）

该方法是*TextTestRunner*的主要公共接口。这个方法需要一个TestSuite或TestCase实例。TestResult通过调用创建A，_makeResult()并运行测试并将结果打印到stdout。

unittest.main（*模块='__main__'，defaultTest =无，argv的=无，TestRunner的=无，testLoader = unittest.defaultTestLoader，出口=真，冗长= 1，FAILFAST =无，catchbreak =无，缓冲=无，警告=无*）

一个命令行程序，从*模块*中加载一组测试并运行它们; 这主要是为了使测试模块方便地执行。此函数最简单的用法是在测试脚本的末尾包含以下行：

```
if __name__ == '__main__':
    unittest.main()
```

您可以通过传递详细信息参数来运行更详细的信息测试：

```
if __name__ == '__main__':
    unittest.main(verbosity=2)
```

所述*defaultTest*参数是一个单一的测试中的名称或测试名称可迭代如果经由未指定试验名称来运行*的argv*。如果未指定或None没有通过*argv*提供测试名称，则会运行*模块*中找到的所有测试。

该*argv的*参数可以是传递给程序的选项列表，第一个元素是程序名称。如果未指定或使用None的值sys.argv。

所述*的TestRunner*参数可以是一个测试运行类或它的一个已创建的实例。默认情况下，带有退出代码的main调用sys.exit()指示测试成功或失败。

该*testLoader*参数必须是一个TestLoader实例，并默认为defaultTestLoader。

main通过传递参数支持交互式解释器的使用exit=False。这会在标准输出中显示结果而不调用sys.exit()：

```
>>> from unittest import main
>>> main(module='test_module', exit=False)
```

的*故障快速转移*，*catchbreak*和*缓冲*参数具有与相同名称的相同的效果的命令行选项。

该*警告*参数指定的警告过滤器 应该在运行测试中使用。如果没有指定它，它将保留，None如果一个-W选项传递给**python**（请参阅警告控制），否则它将被设置为'default'。

调用main实际上会返回TestProgram该类的一个实例。这将测试结果存储为result属性。

*在3.1版本中更改：*将*退出*添加参数。

*在版本3.2改变：*该*冗长*，*FAILFAST*，*catchbreak*，*缓冲液*和*警告*加入参数。

*在3.4版本中更改：*将*defaultTest*参数改为同时接受测试名称的迭代。

26.4.8.3.1。load_tests协议

3.2版本中的新功能

模块或包可以定制在正常测试运行期间如何从它们加载测试，或者通过实现一个名为的函数来测试发现load_tests。

如果一个测试模块定义了load_tests它，它将被 TestLoader.loadTestsFromModule()以下参数调用：

```
load_tests(loader, standard_tests, pattern)
```

*模式*从哪里直接通过loadTestsFromModule。它默认为None。

它应该返回一个TestSuite。

*loader*是*加载*的实例TestLoader。 *standard_tests*是默认从模块加载的测试。测试模块通常只想添加或移除标准测试集中的测试。加载包作为测试发现的一部分时使用第三个参数。

load_tests从特定的一组TestCase类中加载测试的典型函数 可能如下所示：

```
test_cases = (TestCase1, TestCase2, TestCase3)

def load_tests(loader, tests, pattern):
    suite = TestSuite()
    for test_class in test_cases:
        tests = loader.loadTestsFromTestCase(test_class)
        suite.addTests(tests)
    return suite
```

如果在包含软件包的目录中启动发现，无论是从命令行还是通过调用TestLoader.discover()，__init__.py都会检查软件包 load_tests。如果该函数不存在，发现将会递归到包中，就像它只是另一个目录一样。否则，发现软件包的测试将被留下load_tests，并用以下参数调用它：

```
load_tests(loader, standard_tests, pattern)
```

这应该返回TestSuite代表包中的所有测试。（ standard_tests只包含从中收集的测试__init__.py。 ）

因为模式被传递到load_tests包中可以自由地继续（并可能修改）测试发现。 load_tests测试软件包的“无功能”功能看起来像：

```
def load_tests(loader, standard_tests, pattern):
    # top level directory cached on loader instance
    this_dir = os.path.dirname(__file__)
    package_tests = loader.discover(start_dir=this_dir, pattern=pattern)
    standard_tests.addTests(package_tests)
    return standard_tests
```

*在版本3.5中更改：*由于不匹配默认模式的软件包名称，发现不再检查匹配*模式*的软件包名称。

26.4.9。类和模块夹具

类和模块级别的固定装置在中实现TestSuite。当测试套件遇到来自新类的测试时，则tearDownClass() 从前一个类（如果有的话）被调用，然后setUpClass()从新类中调用 。

类似地，如果测试来自与先前测试不同的模块， tearDownModule则从之前的模块开始运行，然后 setUpModule从新模块运行。

在所有测试运行最后tearDownClass并且 tearDownModule运行之后。

请注意，共享装置在测试并行化等[潜在]功能方面表现不佳，并且它们会中断测试隔离。他们应该小心使用。

由unittest测试加载器创建的测试的默认排序是将来自相同模块和类的所有测试分组在一起。这将导致 setUpClass/setUpModule（等）每个类和模块只被调用一次。如果随机化订单，以便来自不同模块和类的测试彼此相邻，则可以在单次测试中多次调用这些共享夹具功能。

共享灯具不适用于具有非标准订购的套件。一个BaseTestSuite仍然存在不希望支持共享夹具框架。

如果在共享夹具功能之一中发生任何异常，则测试报告为错误。因为没有相应的测试实例，所以创建一个_ErrorHolder对象（与a具有相同的接口 TestCase）来表示错误。如果你只是使用标准的unittest测试运行器，那么这个细节并不重要，但如果你是一个框架作者，它可能是相关的。

26.4.9.1。setUpClass和tearDownClass

这些必须作为类方法来实现：

```
import unittest
```

```
class Test(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        cls._connection = createExpensiveConnectionObject()

    @classmethod
    def tearDownClass(cls):
        cls._connection.destroy()
```

如果您希望setUpClass和tearDownClass呼吁然后在基类必须调用由他们自己。这些实现 TestCase是空的。

如果在此期间发生异常setUpClass ，则类中的测试未运行tearDownClass且未运行。被跳过的班级不会有setUpClass或tearDownClass运行。如果异常是一个 SkipTest异常，那么该类将被报告为被跳过而不是作为错误。

26.4.9.2。setUpModule和tearDownModule

这些应该作为功能来实现：

```
def setUpModule():
    createConnection()

def tearDownModule():
    closeConnection()
```

如果在某个时候发生异常setUpModule ，则模块中的所有测试都不会运行，并且tearDownModule不会运行。如果异常是一个 SkipTest异常，那么该模块将被报告为被跳过而不是错误。

10年4月26日。信号处理

3.2版本中的新功能

单元测试的-c/--catch命令行选项以及catchbreak参数可以unittest.main()在测试运行期间提供更友好的control-C处理。在启用catch break行为的情况下，control-C将允许当前正在运行的测试完成，然后测试运行将结束并报告迄今为止的所有结果。第二个控件c会KeyboardInterrupt以通常的方式提出。

control-c处理信号处理程序试图保持与安装自己的signal.SIGINT处理程序的代码或测试的兼容性。如果unittest处理程序被调用但*不是*已安装的signal.SIGINT处理程序，即它已被替换为被测系统并委托给它，则它会调用默认处理程序。这通常是代码的预期行为，该代码取代已安装的处理程序并委托给它。对于需要unittestcontrol-c处理禁用的单个测试，removeHandler() 可以使用装饰器。

为了在测试框架中启用control-c处理功能，框架作者有几个实用功能。

unittest.**installHandler**( )
: 安装control-c处理程序。当signal.SIGINT收到a时（通常是响应用户按下control-c）所有已注册的结果已stop()被调用。

unittest.**registerResult**(*结果*)
: TestResult为control-c处理注册一个对象。注册结果会存储对它的弱引用，所以它不会阻止垃圾收集结果。

  TestResult如果未启用control-c处理，则注册对象没有副作用，因此测试框架可以无条件地注册它们创建的所有结果，而不管是否启用处理。

unittest.**removeResult**(*结果*)
: 删除注册结果。一旦结果被删除，那么 stop()将不再调用该结果对象来响应control-c。

unittest.**removeHandler**(*function = None*)
: 当没有参数的情况下调用这个函数会删除control-c处理程序，如果它已经安装。这个函数也可以用作测试装饰器在测试执行时临时移除该处理器：

  ```
  @unittest.removeHandler
  def test_signal_handling(self):
      ...
  ```

26.5。unittest.mock- 模拟对象库

3.3版本的新功能

源代码： Lib / unittest / mock.py

unittest.mock是一个用Python进行测试的库。它允许您用模拟对象替换测试系统的某些部分，并断言它们是如何使用的。

unittest.mock提供了一个核心Mock类，不需要在整个测试套件中创建大量存根。执行一个动作后，你可以断言使用哪些方法/属性以及它们被调用的参数。您还可以指定返回值并以正常方式设置所需的属性。

另外，mock提供了一个patch()装饰器，用于处理测试范围内的修补模块和类级别属性，以及sentinel创建独特对象。请参阅快速指南对如何使用一些例子Mock，MagicMock和 patch()。

模拟是非常容易使用，并设计用于unittest。模拟基于'动作 - >断言'模式，而不是'记录 - >重播'被许多模拟框架使用。

有一个unittest.mock早期版本的Python的后端，可以在PyPI上模拟。

26.5.1。快速指南

Mock并且MagicMock对象在您访问它们时创建所有属性和方法，并存储它们如何使用的详细信息。您可以对它们进行配置，指定返回值或限制可用的属性，然后声明它们的使用方式：

```
>>> from unittest.mock import MagicMock
>>> thing = ProductionClass()
>>> thing.method = MagicMock(return_value=3)
>>> thing.method(3, 4, 5, key='value')
3
>>> thing.method.assert_called_with(3, 4, 5, key='value')
```

side_effect 允许您执行副作用，包括在调用模拟时引发异常：

```
>>> mock = Mock(side_effect=KeyError('foo'))
>>> mock()
Traceback (most recent call last):
 ...
KeyError: 'foo'
```

```
>>> values = {'a': 1, 'b': 2, 'c': 3}
>>> def side_effect(arg):
...     return values[arg]
...
>>> mock.side_effect = side_effect
>>> mock('a'), mock('b'), mock('c')
(1, 2, 3)
>>> mock.side_effect = [5, 4, 3, 2, 1]
```

```
>>> mock(), mock(), mock()
(5, 4, 3)
```

Mock有许多其他方式可以配置它并控制其行为。例如，*spec*参数配置模拟从另一个对象获取其规格。尝试访问该规范中不存在的属性或方法将会失败，并带有一个AttributeError。

的patch()装饰/上下文管理器可以很容易地嘲笑类或对象模块中的下测试。您指定的对象将在测试过程中替换为模拟（或其他对象），并在测试结束时恢复：

```
>>> from unittest.mock import patch
>>> @patch('module.ClassName2')
... @patch('module.ClassName1')
... def test(MockClass1, MockClass2):
...     module.ClassName1()
...     module.ClassName2()
...     assert MockClass1 is module.ClassName1
...     assert MockClass2 is module.ClassName2
...     assert MockClass1.called
...     assert MockClass2.called
...
>>> test()
```

注意： 当您嵌套修补程序修饰符时，这些修改将以与它们应用的顺序相同的顺序（装饰器应用的正常*Python*顺序）传递到装饰函数中。这意味着从下往上，所以在上面的例子中，模拟module.ClassName1是先传入的。

随着patch()它的事项，你在他们抬头的命名空间修补的对象。这通常很简单，但要获得快速指南，请参阅补丁的位置。

以及装饰器patch()可以用作with语句中的上下文管理器：

```
>>> with patch.object(ProductionClass, 'method', return_value=None) as mock_method:
...     thing = ProductionClass()
...     thing.method(1, 2, 3)
...
>>> mock_method.assert_called_once_with(1, 2, 3)
```

也patch.dict()可以在范围内在字典中设置值，并在测试结束时将字典恢复到原始状态：

```
>>> foo = {'key': 'value'}
>>> original = foo.copy()
>>> with patch.dict(foo, {'newkey': 'newvalue'}, clear=True):
...     assert foo == {'newkey': 'newvalue'}
...
>>> assert foo == original
```

Mock支持嘲讽Python 魔术方法。使用魔术方法的最简单方法是在MagicMock课堂上。它允许你做这样的事情：

```
>>> mock = MagicMock()
>>> mock.__str__.return_value = 'foobarbaz'
>>> str(mock)
```

```
'foobarbaz'
>>> mock.__str__.assert_called_with()
```

模拟允许您将函数（或其他模拟实例）分配给魔术方法，并且它们将被适当地调用。这个MagicMock类只是一个模拟变体，它具有为你预先创建的所有魔法方法（无论如何，所有有用的方法）。

以下是在普通的Mock类中使用魔术方法的一个例子：

```
>>> mock = Mock()
>>> mock.__str__ = Mock(return_value='wheeeeee')
>>> str(mock)
'wheeeeee'
```

为了确保测试中的模拟对象与它们要替换的对象具有相同的API，可以使用自动指定。自动指定可以通过*autospec*参数来修补，或者通过 create_autospec() 函数来完成。自动指定会创建与被替换的对象具有相同属性和方法的模拟对象，并且任何函数和方法（包括构造函数）都具有与实际对象相同的调用签名。

这确保了如果错误地使用你的mock，将会像生产代码一样失败：

```
>>> from unittest.mock import create_autospec
>>> def function(a, b, c):
...     pass
...
>>> mock_function = create_autospec(function, return_value='fishy')
>>> mock_function(1, 2, 3)
'fishy'
>>> mock_function.assert_called_once_with(1, 2, 3)
>>> mock_function('wrong arguments')
Traceback (most recent call last):
 ...
TypeError: <lambda>() takes exactly 3 arguments (1 given)
```

create_autospec() 也可以用于复制__init__方法签名的类，以及复制方法签名的可调用对象__call__。

26.5.2。模拟类

Mock是一个灵活的模拟对象，旨在替代整个代码中使用存根和测试双打。嘲笑是可调用的，当你访问它们时创建属性作为新的嘲笑[1]。访问相同的属性将始终返回相同的模拟。嘲笑记录你如何使用它们，让你可以断言你的代码对他们做了什么。

MagicMock是Mock预先创建并准备使用的所有魔术方法的子类。也有不可调用的变体，当你嘲笑不可调用的对象时很有用： NonCallableMock和NonCallableMagicMock

的patch() 装饰可以很容易地暂时代替与特定模块中的类Mock对象。默认情况下patch() 会MagicMock为你创建一个。您可以指定一个Mock使用*new_callable*参数的替代类patch()。

class unittest.mock. Mock (*spec = None*，*side_effect = None*，*return_value = DEFAULT*，*wraps = None*，*name = None*，*spec_set = None*，*unsafe = False*，*** kwargs*)

创建一个新的Mock对象。Mock需要几个可选参数来指定Mock对象的行为：

- *spec*：这可以是一个字符串列表或一个现有的对象（一个类或实例），充当模拟对象的规范。如果你传递一个对象，那么通过调用对象上的dir（不包括不受支持的魔术属性和方法）来形成一个字符串列表。访问不在此列表中的任何属性将引发一个AttributeError。

  如果*spec*是一个对象（而不是字符串列表），则 __class__返回spec对象的类。这允许mocks通过isinstance()测试。

- *spec_set*：*规范*的更严格的变体。如果使用，试图*设置*或获取不在作为*spec_set*传递的对象上的模拟属性 将引发一个AttributeError。

- *side_effect*：每当调用Mock时被调用的函数。查看side_effect属性。用于引发异常或动态更改返回值。该函数的调用方式与模拟的参数相同，除非它返回DEFAULT，否则此函数的返回值将用作返回值。

  或者，*side_effect*可以是一个异常类或实例。在这种情况下，当模拟被调用时会引发异常。

  如果*side_effect*是可迭代的，那么每次调用模拟将返回来自迭代器的下一个值。

  通过将*side_effect*设置为可以清除*side_effect*None。

- *return_value*：调用模拟时返回的值。默认情况下，这是一个新的模拟（在第一次访问时创建）。查看 return_value属性。

- *不安全*：默认情况下，如果任何以*assert*或 *assret*开头的属性都会引发一个AttributeError。通过unsafe=True 将允许访问这些属性。

  3.5版本中的新功能。

- *包装*：*包装*模拟对象的项目。如果*包装*没有None调用，Mock会将调用传递给包装对象（返回真实结果）。模拟上的属性访问将返回一个Mock对象，该对象封装了包装对象的相应属性（因此尝试访问不存在的属性将引发AttributeError）。

  如果模拟具有显式的*return_value*设置，则调用不会传递给包装的对象，而是返回*return_value*。

- *名称*：如果模拟名称，那么它将用于模拟的repr。这对调试很有用。这个名字传播给孩子们。

Mocks也可以使用任意关键字参数来调用。这些将被用于在模拟创建后设置属性。详情请参阅 configure_mock()方法。

assert_called(**args*, ***kwargs*)

断言该模拟被调用至少一次。

```
>>> mock = Mock()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called()
```

3.6版本中的新功能。

assert_called_once (**args*, ***kwargs*)

断言模拟只被调用一次。

```
>>> mock = Mock()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_once()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_once()
Traceback (most recent call last):
...
AssertionError: Expected 'method' to have been called once. Called 2 times.
```

3.6版本中的新功能。

assert_called_with (**args*, ***kwargs*)

这种方法是一种方便的方式来断言调用是以特定的方式进行的：

```
>>> mock = Mock()
>>> mock.method(1, 2, 3, test='wow')
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_with(1, 2, 3, test='wow')
```

assert_called_once_with (**args*, ***kwargs*)

断言该模拟只被调用一次，并且该调用与指定的参数一致。

```
>>> mock = Mock(return_value=None)
>>> mock('foo', bar='baz')
>>> mock.assert_called_once_with('foo', bar='baz')
>>> mock('other', bar='values')
>>> mock.assert_called_once_with('other', bar='values')
Traceback (most recent call last):
  ...
AssertionError: Expected 'mock' to be called once. Called 2 times.
```

assert_any_call (**args*, ***kwargs*)

断言模拟已被指定参数调用。

断言如果通过模拟已经*永远*被调用，不像 assert_called_with() 并且 assert_called_once_with() ，只有当呼叫最近的一次传球，并且在的情况下，assert_called_once_with()它也必须是唯一的呼叫。

```
>>> mock = Mock(return_value=None)
>>> mock(1, 2, arg='thing')
>>> mock('some', 'thing', 'else')
>>> mock.assert_any_call(1, 2, arg='thing')
```

assert_has_calls (*调用*, *any_order = False*)

断言模拟已被指定的调用调用。该mock_calls列表被检查通话。

如果*any_order*为false（默认值），那么调用必须是顺序的。在指定的呼叫之前或之后可以有额外的呼叫。

如果*any_order*是真的，那么这些调用可以以任何顺序，但它们必须全部出现在中mock_calls。

```
>>> mock = Mock(return_value=None)
>>> mock(1)
>>> mock(2)
>>> mock(3)
>>> mock(4)
>>> calls = [call(2), call(3)]
>>> mock.assert_has_calls(calls)
>>> calls = [call(4), call(2), call(3)]
>>> mock.assert_has_calls(calls, any_order=True)
```

assert_not_called ( )

断言模拟从未被调用过。

```
>>> m = Mock()
>>> m.hello.assert_not_called()
>>> obj = m.hello()
>>> m.hello.assert_not_called()
Traceback (most recent call last):
  ...
AssertionError: Expected 'hello' to not have been called. Called 1 times.
```

3.5版本中的新功能。

reset_mock (*, *return_value* = *False*, *side_effect* = *False*)

reset_mock方法重置模拟对象上的所有调用属性：

```
>>> mock = Mock(return_value=None)
>>> mock('hello')
>>> mock.called
True
>>> mock.reset_mock()
>>> mock.called
False
```

*在版本3.6中更改：*为reset_mock函数添加了两个仅用于关键字的参数。

当你想要做一系列重复使用同一个对象的断言时，这可能很有用。请注意，*不会*清除返回值或默认情况下使用普通分配设置的任何子属性。如果你想重置 *return_value*或者，然后传递相应的参数。孩子嘲笑和返回值模拟（如果有的话）也重置。reset_mock() side_effectside_effectTrue

> **注意：** *return_value*，并且side_effect是仅关键字参数。

mock_add_spec (*spec*, *spec_set* = *False*)

给模拟添加一个规范。*spec*可以是对象或字符串列表。只有*规范中的*属性才能从模拟中获取为属性。

如果*spec_set*为true，则只能设置spec中的属性。

attach_mock (*模拟*, *属性*)

附上一个模拟作为这个属性，替换它的名字和父母。呼叫连接的模拟将被记录在method_calls和mock_calls这一个属性。

configure_mock (** *kwargs*)

通过关键字参数在模拟上设置属性。

属性加返回值和副作用可以使用标准点符号在子模块上设置，并在方法调用中解开字典：

```
>>> mock = Mock()
>>> attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> mock.configure_mock(**attrs)
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
  ...
KeyError
```

在mock的构造函数调用中可以实现同样的事情：

```
>>> attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> mock = Mock(some_attribute='eggs', **attrs)
>>> mock.some_attribute
'eggs'
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
  ...
KeyError
```

configure_mock() 存在是为了使模拟创建后更容易进行配置。

__dir__ ( )

Mock对象将结果限制dir(some_mock)为有用的结果。对于具有*spec*的模拟，这包括模拟的所有允许的属性。

查看FILTER_DIR过滤功能，以及如何将其关闭。

_get_child_mock (** *kw*)

创建子模型以获取属性并返回值。默认情况下，子模拟器将与父模块的类型相同。Mock的子类可能想要覆盖这个来自定义子模拟的方式。

对于不可调用的嘲笑，将使用可调用变体（而不是任何自定义子类）。

called

代表模拟对象是否被调用的布尔值：

```
>>> mock = Mock(return_value=None)
>>> mock.called
False
>>> mock()
>>> mock.called
True
```

call_count

一个整数，告诉你模拟对象被调用了多少次：

```
>>> mock = Mock(return_value=None)
>>> mock.call_count
0
>>> mock()
>>> mock()
>>> mock.call_count
2
```

return_value

将其设置为配置通过调用模拟返回的值：

```
>>> mock = Mock()
>>> mock.return_value = 'fish'
>>> mock()
'fish'
```

默认返回值是一个模拟对象，你可以用正常的方式来配置它：

```
>>> mock = Mock()
>>> mock.return_value.attribute = sentinel.Attribute
>>> mock.return_value()
<Mock name='mock()()' id='...'>
>>> mock.return_value.assert_called_with()
```

`return_value` 也可以在构造函数中设置：

```
>>> mock = Mock(return_value=3)
>>> mock.return_value
3
>>> mock()
3
```

side_effect

这可以是调用模拟时要调用的函数，可以是可迭代的，也可以是引发的异常（类或实例）。

如果你传入一个函数，它将被调用与模拟相同的参数，除非函数返回`DEFAULT`单例，否则对模拟的调用将返回函数返回的任何内容。如果函数返回，`DEFAULT`则模拟将返回其正常值（来自`return_value`）。

如果你传入一个迭代器，它将被用来检索一个迭代器，它必须在每次调用时产生一个值。该值可以是要引发的异常实例，也可以是从模拟调用返回的值（DEFAULT处理与函数案例相同）。

引发异常的模拟示例（用于测试API的异常处理）：

```
>>> mock = Mock()
>>> mock.side_effect = Exception('Boom!')
>>> mock()
Traceback (most recent call last):
  ...
Exception: Boom!
```

使用side_effect返回一系列值：

```
>>> mock = Mock()
>>> mock.side_effect = [3, 2, 1]
>>> mock(), mock(), mock()
(3, 2, 1)
```

使用可调用的：

```
>>> mock = Mock(return_value=3)
>>> def side_effect(*args, **kwargs):
...     return DEFAULT
...
>>> mock.side_effect = side_effect
>>> mock()
3
```

side_effect可以在构造函数中设置。下面是一个例子，它为模拟函数调用的值添加一个值并返回它：

```
>>> side_effect = lambda value: value + 1
>>> mock = Mock(side_effect=side_effect)
>>> mock(3)
4
>>> mock(-8)
-7
```

设置side_effect以None将其清除：

```
>>> m = Mock(side_effect=KeyError, return_value=3)
>>> m()
Traceback (most recent call last):
 ...
KeyError
>>> m.side_effect = None
>>> m()
3
```

call_args

这要么是None（如果模拟没有被调用），要么是上次调用模拟的参数。这将以一个元组的形式出现：第一个成员是调用模拟（或空元组）的任何有序参数，第二个成员是任何关键字参数（或空字典）。

```
>>> mock = Mock(return_value=None)
>>> print(mock.call_args)
None
>>> mock()
>>> mock.call_args
call()
>>> mock.call_args == ()
True
>>> mock(3, 4)
>>> mock.call_args
call(3, 4)
>>> mock.call_args == ((3, 4),)
True
>>> mock(3, 4, 5, key='fish', next='w00t!')
>>> mock.call_args
call(3, 4, 5, key='fish', next='w00t!')
```

call_args，以及列表中的成员call_args_list， method_calls并且mock_calls是call对象。这些都是元组，因此可以将它们解压缩以获取各个参数并进行更复杂的断言。见 电话元组。

call_args_list

这是按顺序对模拟对象进行的所有调用的列表（因此列表的长度是调用次数）。在进行任何呼叫之前，它是一个空的列表。该 call对象可用于方便地构建与之比较的调用列表call_args_list。

```
>>> mock = Mock(return_value=None)
>>> mock()
>>> mock(3, 4)
>>> mock(key='fish', next='w00t!')
>>> mock.call_args_list
[call(), call(3, 4), call(key='fish', next='w00t!')]
>>> expected = [(), ((3, 4),), ({'key': 'fish', 'next': 'w00t!'},)]
>>> mock.call_args_list == expected
True
```

成员call_args_list是call对象。这些可以解压为元组来获得单个参数。见 电话元组。

method_calls

除了跟踪对自己的调用外，mock还会跟踪对方法和属性以及*它们的*方法和属性的调用：

```
>>> mock = Mock()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.property.method.attribute()
<Mock name='mock.property.method.attribute()' id='...'>
```

```
>>> mock.method_calls
[call.method(), call.property.method.attribute()]
```

成员method_calls是call对象。这些可以解压为元组来获得单个参数。见 电话元组。

mock_calls

mock_calls记录对模拟对象，其方法，魔术方法*和*返回值模拟的*所有*调用。

```
>>> mock = MagicMock()
>>> result = mock(1, 2, 3)
>>> mock.first(a=3)
<MagicMock name='mock.first()' id='...'>
>>> mock.second()
<MagicMock name='mock.second()' id='...'>
>>> int(mock)
1
>>> result(1)
<MagicMock name='mock()()' id='...'>
>>> expected = [call(1, 2, 3), call.first(a=3), call.second(),
... call.__int__(), call()(1)]
>>> mock.mock_calls == expected
True
```

成员mock_calls是call对象。这些可以解压为元组来获得单个参数。见 电话元组。

__class__

通常，__class__对象的属性将返回其类型。对于具有a的模拟对象spec，请__class__返回spec类。这允许模拟对象通过isinstance()他们正在替换/伪装的对象的测试：

```
>>> mock = Mock(spec=3)
>>> isinstance(mock, int)
True
```

__class__是可分配的，这允许模拟通过 isinstance()检查而不强制你使用规范：

```
>>> mock = Mock()
>>> mock.__class__ = dict
>>> isinstance(mock, dict)
True
```

class unittest.mock.**NonCallableMock**(*spec = None*，*wraps = None*，*name = None*，*spec_set = None*，*** kwargs*)

一个不可调用的版本Mock。Mock除了*return_value*和*side_effect*之外，构造函数参数具有相同的含义， 这对于不可调用的模拟没有意义。

模拟使用类或实例作为spec或 spec_set能够通过isinstance()测试的对象：

```
>>> mock = Mock(spec=SomeClass)
>>> isinstance(mock, SomeClass)
True
>>> mock = Mock(spec_set=SomeClass())
```

```
>>> isinstance(mock, SomeClass)
True
```

这些Mock类支持嘲讽魔法方法。有关完整详情，请参阅魔术方法。

模拟类和patch()装饰器都采用任意关键字参数进行配置。对于patch()装饰器，关键字被传递给正在创建的模拟的构造函数。关键字参数用于配置模拟的属性：

```
>>> m = MagicMock(attribute=3, other='fish')
>>> m.attribute
3
>>> m.other
'fish'
```

儿童模拟的返回值和副作用可以用相同的方式设置，使用虚线符号。由于您不能直接在调用中使用虚线名称，因此必须创建一个字典并使用**以下命令对其进行解压缩：

```
>>> attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> mock = Mock(some_attribute='eggs', **attrs)
>>> mock.some_attribute
'eggs'
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
  ...
KeyError
```

使用*spec*（或*spec_set*）创建的可调用模拟会在匹配对模拟的调用时检查规范对象的签名。因此，它可以匹配实际调用的参数，而不管它们是通过位置还是名称：

```
>>> def f(a, b, c): pass
...
>>> mock = Mock(spec=f)
>>> mock(1, 2, c=3)
<Mock name='mock()' id='140161580456576'>
>>> mock.assert_called_with(1, 2, 3)
>>> mock.assert_called_with(a=1, b=2, c=3)
```

这适用于assert_called_with()，assert_called_once_with()，assert_has_calls()和assert_any_call()。当Autospeccing，它也适用于方法模拟对象调用。

*在版本3.4中进行了更改：*在指定和自动引用的模拟对象上添加了签名反省。

class unittest.mock.PropertyMock(**args*, ***kwargs*)

模拟意图用作类的属性或其他描述符。PropertyMock提供__get__()和__set__()方法，以便您可以在提取时指定返回值。

PropertyMock从对象中获取一个实例调用模拟，没有参数。设置它将调用设置值的模拟。

```
>>> class Foo:
...     @property
...     def foo(self):
```

```
...         return 'something'
...     @foo.setter
...     def foo(self, value):
...         pass
...
>>> with patch('__main__.Foo.foo', new_callable=PropertyMock) as mock_foo:
...     mock_foo.return_value = 'mockity-mock'
...     this_foo = Foo()
...     print(this_foo.foo)
...     this_foo.foo = 6
...
mockity-mock
>>> mock_foo.mock_calls
[call(), call(6)]
```

由于存储模拟属性的方式，您不能直接将其附加 PropertyMock到模拟对象。相反，您可以将其附加到模拟类型对象：

```
>>> m = MagicMock()
>>> p = PropertyMock(return_value=3)
>>> type(m).foo = p
>>> m.foo
3
>>> p.assert_called_once_with()
```

26.5.2.1。调用

模拟对象是可调用的。该调用将返回设置为return_value属性的值 。默认返回值是一个新的Mock对象; 它是在第一次访问返回值时创建的（显式地或通过调用Mock） - 但它被存储并且每次都返回相同的值。

对该对象的调用将被记录在诸如call_args和的属性中call_args_list。

如果side_effect已设定，则在通话录音后将被呼叫，因此如果side_effect引发异常，通话仍将被录制。

进行模拟时最简单的方法是调用 side_effect异常类或实例：

```
>>> m = MagicMock(side_effect=IndexError)
>>> m(1, 2, 3)
Traceback (most recent call last):
  ...
IndexError
>>> m.mock_calls
[call(1, 2, 3)]
>>> m.side_effect = KeyError('Bang!')
>>> m('two', 'three', 'four')
Traceback (most recent call last):
  ...
KeyError: 'Bang!'
>>> m.mock_calls
[call(1, 2, 3), call('two', 'three', 'four')]
```

如果side_effect是一个函数，那么无论函数返回的是什么调用模拟返回。该side_effect函数被调用与模拟相同的参数。这允许您根据输入动态地改变呼叫的返回值：

```
>>> def side_effect(value):
...     return value + 1
...
>>> m = MagicMock(side_effect=side_effect)
>>> m(1)
2
>>> m(2)
3
>>> m.mock_calls
[call(1), call(2)]
```

如果你想模拟仍然返回默认返回值（一个新的模拟）或任何设置的返回值，那么有两种方法可以做到这一点。要么 mock.return_value从内部side_effect返回，要么返回DEFAULT：

```
>>> m = MagicMock()
>>> def side_effect(*args, **kwargs):
...     return m.return_value
...
>>> m.side_effect = side_effect
>>> m.return_value = 3
>>> m()
3
>>> def side_effect(*args, **kwargs):
...     return DEFAULT
...
>>> m.side_effect = side_effect
>>> m()
3
```

要删除一个side_effect，并返回到默认行为，请将其设置 side_effect为None：

```
>>> m = MagicMock(return_value=6)
>>> def side_effect(*args, **kwargs):
...     return 3
...
>>> m.side_effect = side_effect
>>> m()
3
>>> m.side_effect = None
>>> m()
6
```

该side_effect也可以是任何可迭代的对象。对模拟的重复调用将从迭代中返回值（直到迭代器耗尽且a StopIteration被引发）：

```
>>> m = MagicMock(side_effect=[1, 2, 3])
>>> m()
1
>>> m()
2
>>> m()
3
```

```
>>> m()
Traceback (most recent call last):
  ...
StopIteration
```

如果迭代的任何成员都是异常，它们将被提出而不是返回：

```
>>> iterable = (33, ValueError, 66)
>>> m = MagicMock(side_effect=iterable)
>>> m()
33
>>> m()
Traceback (most recent call last):
 ...
ValueError
>>> m()
66
```

26.5.2.2。删除属性

模拟对象按需创建属性。这允许他们假装成任何类型的对象。

你可能想要一个模拟对象返回False到一个hasattr()调用，或者 AttributeError当一个属性被获取时引发。你可以通过提供spec一个模拟对象来做到这一点，但这并不总是方便的。

您通过删除它们来“阻止”属性。一旦删除，访问属性将引发一个AttributeError。

```
>>> mock = MagicMock()
>>> hasattr(mock, 'm')
True
>>> del mock.m
>>> hasattr(mock, 'm')
False
>>> del mock.f
>>> mock.f
Traceback (most recent call last):
    ...
AttributeError: f
```

26.5.2.3。模拟名称和名称属性

由于“name”是Mock构造函数的参数，如果你希望你的模拟对象具有“name”属性，你不能在创建时传递它。有两种选择。一种选择是使用 configure_mock() ：

```
>>> mock = MagicMock()
>>> mock.configure_mock(name='my_name')
>>> mock.name
'my_name'
```

更简单的选项是在模拟创建后简单地设置“name”属性：

```
>>> mock = MagicMock()
>>> mock.name = "foo"
```

26.5.2.4。将Mocks附加为属性

当你将模拟附加为另一个模拟的属性（或作为返回值）时，它就成为该模拟的“孩子”。呼吁给孩子记录在method_calls和mock_calls父母的属性。这对于配置子模块然后将它们附加到父模块或者将模拟模块附加到记录所有对子模块的调用的父节点是很有用的，并且允许您对mock之间的调用顺序进行断言：

```
>>> parent = MagicMock()
>>> child1 = MagicMock(return_value=None)
>>> child2 = MagicMock(return_value=None)
>>> parent.child1 = child1
>>> parent.child2 = child2
>>> child1(1)
>>> child2(2)
>>> parent.mock_calls
[call.child1(1), call.child2(2)]
```

这是一个例外，如果模拟名称。这可以让你防止“父母”，如果由于某种原因你不希望它发生。

```
>>> mock = MagicMock()
>>> not_a_child = MagicMock(name='not-a-child')
>>> mock.attribute = not_a_child
>>> mock.attribute()
<MagicMock name='not-a-child()' id='...'>
>>> mock.mock_calls
[]
```

为您创建的模拟patch()会自动给出名称。要将具有名称的嘲笑附加到父项，请使用以下attach_mock()方法：

```
>>> thing1 = object()
>>> thing2 = object()
>>> parent = MagicMock()
>>> with patch('__main__.thing1', return_value=None) as child1:
...     with patch('__main__.thing2', return_value=None) as child2:
...         parent.attach_mock(child1, 'child1')
...         parent.attach_mock(child2, 'child2')
...         child1('one')
...         child2('two')
...
>>> parent.mock_calls
[call.child1('one'), call.child2('two')]
```

[1] 提出的例外是神奇的方法和属性（那些具有前导和尾随双下划线的）。模拟方法和属性魔术方法支持，请参阅魔术方法。

26.5.3。修补程序

修补程序修饰器仅用于修补它们修饰的功能范围内的对象。即使发生异常，它们也会自动为您处理未调整的问题。所有这些函数也可以用于语句或类装饰器。

26.5.3.1。补丁

注意： patch()是直接使用。关键是要在正确的命名空间中进行修补。请参阅要修补的部分。

unittest.mock.**patch**(*target*，*new = DEFAULT*，*spec = None*，*create = False*，*spec_set = None*，*autospec = None*，*new_callable = None*，*** kwargs*)

patch()充当函数装饰器，类装饰器或上下文管理器。在函数的主体或声明中，*目标*是用一个*新*对象修补的。当函数/ with语句退出时，修补程序被撤消。

如果*新*被省略，则目标被替换为a MagicMock。如果patch()用作装饰器，而省略*new*，则创建的模拟作为额外参数传入装饰函数。如果patch()用作上下文管理器，则创建的模拟将由上下文管理器返回。

*目标*应该是表单中的一个字符串'package.module.ClassName'。该*目标*是进口和指定对象与被替换的*新*对象，所以*目标*必须是您所呼叫的环境可导入patch()从。当装饰功能被执行时，目标被导入，而不是在装饰时。

该*规范*和*spec_set*关键字参数传递给MagicMock，如果补丁创建一个给你。

此外，您可以传递spec=True或spec_set=True，这会导致修补程序作为spec / spec_set对象传递到被模拟的对象中。

*new_callable*允许你指定一个不同的类或可调用的对象，这个对象将被调用来创建*新的*对象。默认MagicMock使用。

*autospec*是更强大的*规范*形式。如果你设置了，那么模拟将被替换的对象的规范创建。模拟的所有属性也将具有被替换对象的相应属性的规格。被嘲笑的方法和函数将检查它们的参数，并且如果它们被错误的签名调用，则会引发它们。为了模拟替换一个类，它们的返回值（'实例'）将具有与该类相同的规格。请参阅功能和 Autospeccing。autospec=True TypeError create_autospec()

而不是autospec=True你可以通过autospec=some_object使用任意对象作为规范而不是被替换的规范。

默认情况下，patch()将无法替换不存在的属性。如果传入create=True并且该属性不存在，则补丁将在调用修补功能时为您创建属性，然后再次将其删除。这对于您的生产代码在运行时创建的属性编写测试非常有用。它默认关闭，因为它可能很危险。开启它后，您可以针对实际不存在的API编写通过测试！

注意：
*在版本3.5中进行了更改：*如果您正在将模块内置到模块中，那么您无需通过create=True，它将默认添加。

修补程序可以用作TestCase类装饰器。它通过装饰课堂中的每个测试方法来工作。当您的测试方法共享一个通用的贴片集时，这将减少样板代码。patch()通过查找以开头的方法名称来查找测试patch.TEST_PREFIX。默认情况下'test'，这与unittest查找测试的方式相匹配。您可以通过设置指定替代前缀patch.TEST_PREFIX。

可以使用with语句将补丁用作上下文管理器。这里修补适用于with语句之后的缩进块。如果使用“as”，那么修补后的对象将被绑定到“as”之后的名称; 如果patch()为你创建一个模拟对象非常有用。

patch()采用任意关键字参数。这些将被传递给施工中的Mock（或*new_callable*）。

patch.dict(...)，patch.multiple(...)和patch.object(...)可用于备用的用例。

patch() 作为函数装饰器，为您创建模拟并将其传递到装饰函数中：

```
>>> @patch('__main__.SomeClass')
... def function(normal_argument, mock_class):
...     print(mock_class is SomeClass)
...
>>> function(None)
True
```

修补类用*实例*替换类。如果类在被测代码中被实例化，那么它将成为 将被使用的模拟。MagicMock return_value

如果该类多次实例化，则可以使用 side_effect每次都返回一个新的模拟。或者，您可以将*return_value*设置为任何您想要的值。

要在修补类的*实例*方法上配置返回值，您必须在修补类中执行此操作return_value。例如：

```
>>> class Class:
...     def method(self):
...         pass
...
>>> with patch('__main__.Class') as MockClass:
...     instance = MockClass.return_value
...     instance.method.return_value = 'foo'
...     assert Class() is instance
...     assert Class().method() == 'foo'
...
```

如果您使用*spec*或*spec_set*并且patch()正在替换一个*类*，那么创建的模拟的返回值将具有相同的规格。

```
>>> Original = Class
>>> patcher = patch('__main__.Class', spec=True)
>>> MockClass = patcher.start()
>>> instance = MockClass()
>>> assert isinstance(instance, Original)
>>> patcher.stop()
```

该*new_callable*要使用替代类的默认参数是有用MagicMock的创建模拟。例如，如果你想要NonCallableMock使用一个：

```
>>> thing = object()
>>> with patch('__main__.thing', new_callable=NonCallableMock) as mock_thing:
...     assert thing is mock_thing
...     thing()
...
Traceback (most recent call last):
  ...
TypeError: 'NonCallableMock' object is not callable
```

另一个用例可能是用一个io.StringIO实例替换一个对象：

```
>>> from io import StringIO
>>> def foo():
...     print('Something')
...
>>> @patch('sys.stdout', new_callable=StringIO)
... def test(mock_stdout):
...     foo()
...     assert mock_stdout.getvalue() == 'Something\n'
...
>>> test()
```

什么时候patch()为你创建一个模拟器，通常你需要做的第一件事就是配置模拟器。一些配置可以在修补程序的调用中完成。您传递给调用的任意任意关键字将用于在创建的模拟上设置属性：

```
>>> patcher = patch('__main__.thing', first='one', second='two')
>>> mock_thing = patcher.start()
>>> mock_thing.first
'one'
>>> mock_thing.second
'two'
```

以及在创建模拟属性的属性，如 return_value和side_effect ，孩子嘲笑也可以配置。这些在语法上不是直接作为关键字参数传入的有效方法，但是这些作为关键字的字典仍然可以通过以下方式扩展为patch()调用**：

```
>>> config = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> patcher = patch('__main__.thing', **config)
>>> mock_thing = patcher.start()
>>> mock_thing.method()
3
>>> mock_thing.other()
Traceback (most recent call last):
  ...
KeyError
```

26.5.3.2。patch.object

patch. object （ *target* ， *attribute* ， *new = DEFAULT* ， *spec = None* ， *create = False* ， *spec_set = None* ， *autospec = None* ， *new_callable = None* ， *** kwargs* ）

使用模拟对象修补对象（ *目标* ）上的指定成员（ *属性* ）。

patch.object() 可以用作装饰器，类装饰器或上下文管理器。参数*new*，*spec*，*create*，*spec_set*，*autospec* 和 *new_callable* 具有与 for 相同的含义 patch()。像 patch()，patch.object() 需要任意关键字参数配置它所创建的模拟对象。

当作为一类装饰用patch.object() 荣誉patch.TEST_PREFIX 为选择哪些方法来包装。

您可以patch.object() 使用三个参数或两个参数进行调用。三个参数表单将要修补的对象，属性名称和要替换属性的对象。

当使用两个参数表单调用时，可以省略替换对象，并为您创建一个模拟并作为额外参数传递给装饰函数：

```
>>> @patch.object(SomeClass, 'class_method')
... def test(mock_method):
...     SomeClass.class_method(3)
...     mock_method.assert_called_with(3)
...
>>> test()
```

规范，*创造*和其他论点patch.object() 具有相同的含义patch()。

26.5.3.3。patch.dict

patch.dict（*in_dict*，*values = ( )*，*clear = False*，*** kwargs* ）

修补字典或字典（如对象），并在测试后将字典恢复到原始状态。

*in_dict*可以是字典或像容器一样的映射。如果它是一个映射，那么它必须至少支持获取，设置和删除项目以及迭代键。

*in_dict*也可以是一个字符串，指定字典的名称，然后通过导入它来获取它。

*值*可以是字典中设置的值的字典。*值* 也可以是一对可迭代的。(key, value)

如果*清除*为真，那么在设置新值之前字典将被清除。

patch.dict() 也可以使用任意关键字参数调用以设置字典中的值。

patch.dict() 可以用作上下文管理器，装饰器或类装饰器。当作为一类装饰用patch.dict() 荣誉 patch.TEST_PREFIX为选择哪些方法来包装。

patch.dict() 可以用于将成员添加到字典中，或者只是让测试更改字典，并确保在测试结束时字典被恢复。

```
>>> foo = {}
>>> with patch.dict(foo, {'newkey': 'newvalue'}):
...     assert foo == {'newkey': 'newvalue'}
...
>>> assert foo == {}
```

```
>>> import os
>>> with patch.dict('os.environ', {'newkey': 'newvalue'}):
```

```
...     print(os.environ['newkey'])
...
newvalue
>>> assert 'newkey' not in os.environ
```

可以在patch.dict()调用中使用关键字来设置字典中的值：

```
>>> mymodule = MagicMock()
>>> mymodule.function.return_value = 'fish'
>>> with patch.dict('sys.modules', mymodule=mymodule):
...     import mymodule
...     mymodule.function('some', 'args')
...
'fish'
```

patch.dict()可以与字典一起使用，就像实际上不是字典的对象。他们至少必须支持项目获取，设置，删除以及迭代或成员测试。这对应于魔术方法__getitem__()，__setitem__()，__delitem__()，要么__iter__()或__contains__()。

```
>>> class Container:
...     def __init__(self):
...         self.values = {}
...     def __getitem__(self, name):
...         return self.values[name]
...     def __setitem__(self, name, value):
...         self.values[name] = value
...     def __delitem__(self, name):
...         del self.values[name]
...     def __iter__(self):
...         return iter(self.values)
...
>>> thing = Container()
>>> thing['one'] = 1
>>> with patch.dict(thing, one=2, two=3):
...     assert thing['one'] == 2
...     assert thing['two'] == 3
...
>>> assert thing['one'] == 1
>>> assert list(thing) == ['one']
```

26.5.3.4。patch.multiple

patch.**multiple**(*target*，*spec = None*，*create = False*，*spec_set = None*，*autospec = None*，*new_callable = None*，*** kwargs*)

在一次调用中执行多个修补程序。它需要修补对象（作为对象或字符串以通过导入获取对象）和修补程序的关键字参数：

```
with patch.multiple(settings, FIRST_PATCH='one', SECOND_PATCH='two'):
    ...
```

使用DEFAULT，如果你想为价值patch.multiple()为您创建嘲笑。在这种情况下，创建的模拟会通过关键字传递到装饰函数中，并且当patch.multiple()用作上下文管理器时会返回

字典。

patch.multiple() 可以用作装饰器，类装饰器或上下文管理器。参数*spec*，*spec_set*，*create*，*autospec*和 *new_callable*具有与for相同的含义patch()。这些参数将应用于*所有*通过patch.multiple()。

当作为一类装饰用patch.multiple()荣誉patch.TEST_PREFIX 为选择哪些方法来包装。

如果你想patch.multiple() 为你创建模拟，那么你可以使用它 DEFAULT 作为值。如果您patch.multiple()用作装饰器，那么创建的模拟会通过关键字传递到装饰函数中。

```
>>> thing = object()
>>> other = object()
```

```
>>> @patch.multiple('__main__', thing=DEFAULT, other=DEFAULT)
... def test_function(thing, other):
...     assert isinstance(thing, MagicMock)
...     assert isinstance(other, MagicMock)
...
>>> test_function()
```

patch.multiple()可以与其他patch装饰器嵌套，但在通过以下方法创建的任何标准参数*之后*，将关键字传递给关键字patch()：

```
>>> @patch('sys.exit')
... @patch.multiple('__main__', thing=DEFAULT, other=DEFAULT)
... def test_function(mock_exit, other, thing):
...     assert 'other' in repr(other)
...     assert 'thing' in repr(thing)
...     assert 'exit' in repr(mock_exit)
...
>>> test_function()
```

如果patch.multiple()用作上下文管理器，则上下文管理器返回的值是一个字典，其中创建的模拟按名称键入：

```
>>> with patch.multiple('__main__', thing=DEFAULT, other=DEFAULT) as values:
...     assert 'other' in repr(values['other'])
...     assert 'thing' in repr(values['thing'])
...     assert values['thing'] is thing
...     assert values['other'] is other
...
```

26.5.3.5。修补方法：启动和停止

所有的修补程序都有start()和stop()方法。这些使得在setUp方法中进行修补变得更简单，或者您希望在没有嵌套装饰器或语句的情况下执行多个修补程序。

要使用它们patch()，patch.object()或者patch.dict()照常使用它们并保留对返回patcher对象的引用。然后您可以打电话start()将补丁放置到位并stop()解除补丁。

如果你正在使用patch()为你创建一个模拟，那么它将被调用返回patcher.start。

```
>>> patcher = patch('package.module.ClassName')
>>> from package import module
>>> original = module.ClassName
>>> new_mock = patcher.start()
>>> assert module.ClassName is not original
>>> assert module.ClassName is new_mock
>>> patcher.stop()
>>> assert module.ClassName is original
>>> assert module.ClassName is not new_mock
```

一个典型的用例可能是在a的setUp 方法中执行多个补丁TestCase：

```
>>> class MyTest(TestCase):
...     def setUp(self):
...         self.patcher1 = patch('package.module.Class1')
...         self.patcher2 = patch('package.module.Class2')
...         self.MockClass1 = self.patcher1.start()
...         self.MockClass2 = self.patcher2.start()
...
...     def tearDown(self):
...         self.patcher1.stop()
...         self.patcher2.stop()
...
...     def test_something(self):
...         assert package.module.Class1 is self.MockClass1
...         assert package.module.Class2 is self.MockClass2
...
>>> MyTest('test_something').run()
```

警告: 如果您使用这种技术，您必须确保修补程序通过调用“撤消” stop。这可能比你想象的要烦琐，因为如果在那里引发异常，setUp那么tearDown就不会被调用。unittest.TestCase.addCleanup()使这更容易：

```
>>> class MyTest(TestCase):
...     def setUp(self):
...         patcher = patch('package.module.Class')
...         self.MockClass = patcher.start()
...         self.addCleanup(patcher.stop)
...
...     def test_something(self):
...         assert package.module.Class is self.MockClass
...
```

作为额外的奖励，您不再需要保留对该patcher 对象的引用。

也可以停止使用已经开始的所有补丁 patch.stopall()。

patch.**stopall**()

停止所有活动的补丁。只停止以开始的补丁start。

26.5.3.6。补丁内建

你可以修补模块内的任何内建。以下示例补丁内置`ord()`：

```
>>> @patch('__main__.ord')
... def test(mock_ord):
...     mock_ord.return_value = 101
...     print(ord('c'))
...
>>> test()
101
```

26.5.3.7。TEST_PREFIX

所有的修补程序都可以用作类装饰器。当以这种方式使用它们时，它们将每个测试方法都包装在类中。修补程序认识到以'test'测试方法开始的方法。这与`unittest.TestLoader`默认查找测试方法的方式相同 。

您可能希望为测试使用不同的前缀。您可以通过设置通知修补程序不同的前缀`patch.TEST_PREFIX`：

```
>>> patch.TEST_PREFIX = 'foo'
>>> value = 3
>>>
>>> @patch('__main__.value', 'not three')
... class Thing:
...     def foo_one(self):
...         print(value)
...     def foo_two(self):
...         print(value)
...
>>>
>>> Thing().foo_one()
not three
>>> Thing().foo_two()
not three
>>> value
3
```

26.5.3.8。嵌套修补程序装饰器

如果你想执行多个补丁，那么你可以简单地堆叠装饰器。

您可以使用以下模式堆叠多个修补程序装饰器：

```
>>> @patch.object(SomeClass, 'class_method')
... @patch.object(SomeClass, 'static_method')
... def test(mock1, mock2):
...     assert SomeClass.static_method is mock1
...     assert SomeClass.class_method is mock2
...     SomeClass.static_method('foo')
```

```
...     SomeClass.class_method('bar')
...     return mock1, mock2
...
>>> mock1, mock2 = test()
>>> mock1.assert_called_once_with('foo')
>>> mock2.assert_called_once_with('bar')
```

请注意，装饰器从底部向上应用。这是Python应用装饰器的标准方式。传入您的测试函数的创建模拟的顺序与此顺序相匹配。

26.5.3.9。在哪里修补

patch()通过（暂时）改变*名称*指向另一个*名称*的对象来工作。可以有许多名称指向任何单个对象，因此为了修补工作，您必须确保您修补被测系统使用的名称。

基本的原则是你修补一个物体被*查找的地方*，这不一定是它定义的地方。几个例子将有助于澄清这一点。

想象一下，我们有一个我们想用以下结构测试的项目：

```
a.py
    -> Defines SomeClass

b.py
    -> from a import SomeClass
    -> some_function instantiates SomeClass
```

现在我们想测试，some_function但我们想嘲笑SomeClass使用 patch()。问题是，当我们导入模块b时，我们将不得不这样做，然后SomeClass从模块a 导入。如果我们用它patch()来模拟，a.SomeClass那么它对我们的测试没有任何影响; 模块b已经有了对*真实*的引用SomeClass，它看起来像我们的补丁没有效果。

关键是要修补SomeClass它使用的地方（或查找它的位置）。在这种情况下，some_function实际上将SomeClass在模块b中查找，我们已经在其中导入它。修补应该看起来像：

```
@patch('b.SomeClass')
```

但是，考虑替代模块b执行和使用的替代方案。这两种进口形式都很常见。在这种情况下，我们想要修补的课程正在模块中查找，因此我们必须修补：from a import SomeClass import a some_function a.SomeClass a.SomeClass

```
@patch('a.SomeClass')
```

26.5.3.10。修补描述符和代理对象

这两个补丁和patch.object正确地修补和恢复描述：类方法，静态方法和属性。你应该将这些修补到*类*而不是实例上。它们还可以与代理属性访问的*一些*对象一起使用，如django设置对象。

26.5.4。MagicMock和魔术方法支持

26.5.4.1。嘲讽魔术方法

Mock支持嘲笑Python协议方法，也被称为“魔术方法”。这允许模拟对象替换实现Python协议的容器或其他对象。

因为神奇的方法与普通的方法不同[2]，所以这种支持已经被特别实施。这意味着只支持特定的魔法方法。支持的列表*几乎*包括所有这些。如果有任何缺失，请告诉我们。

您可以通过将您感兴趣的方法设置为函数或模拟实例来嘲笑魔术方法。如果您使用的是功能，那么它*必须*采取self的第一个参数[3] 。

```
>>> def __str__(self):
...     return 'fooble'
...
>>> mock = Mock()
>>> mock.__str__ = __str__
>>> str(mock)
'fooble'
```

```
>>> mock = Mock()
>>> mock.__str__ = Mock()
>>> mock.__str__.return_value = 'fooble'
>>> str(mock)
'fooble'
```

```
>>> mock = Mock()
>>> mock.__iter__ = Mock(return_value=iter([]))
>>> list(mock)
[]
```

对此的一个用例是用于模拟在with语句中用作上下文管理器的对象 ：

```
>>> mock = Mock()
>>> mock.__enter__ = Mock(return_value='foo')
>>> mock.__exit__ = Mock(return_value=False)
>>> with mock as m:
...     assert m == 'foo'
...
>>> mock.__enter__.assert_called_with()
>>> mock.__exit__.assert_called_with(None, None, None)
```

调用魔术方法不会出现在中method_calls ，但它们被记录在中mock_calls。

注意： 如果你使用*spec*关键字参数来创建一个模拟，那么试图设置一个不符合规范的魔法方法会引发一个AttributeError。

支持的魔术方法的完整列表是：

- __hash__ ， __sizeof__ ， __repr__和__str__
- __dir__ ， __format__和__subclasses__
- __floor__ ， __trunc__和__ceil__

- 比较：__lt__，__gt__，__le__，__ge__，__eq__和__ne__
- 集装箱方法：__getitem__，__setitem__，__delitem__，__contains__，__len__，__iter__，__reversed__和__missing__
- 上下文管理器：__enter__和__exit__
- 一元数值方法：__neg__，__pos__和__invert__
- 该数值方法（包括右手和就地变种）__add__，__sub__，__mul__，__matmul__，__div__，__truediv__，__floordiv__，__mod__，__divmod__，__lshift__，__rshift__，__and__，__xor__，__or__，和__pow__
- 数字转换方法：__complex__，__int__，__float__和__index__
- 描述符方法：__get__，__set__和__delete__
- 酸洗：__reduce__，__reduce_ex__，__getinitargs__，__getnewargs__，__getstate__和__setstate__

的确存在，但下面的方法*不*支持的，因为它们无论是在通过模拟使用，无法动态设置，或可能会导致问题：

- __getattr__，__setattr__，__init__和__new__
- __prepare__，__instancecheck__，__subclasscheck__，__del__

26.5.4.2。Magic

有两种MagicMock变体：MagicMock和NonCallableMagicMock。

class unittest.mock.**MagicMock**(**args*, ***kw*)

MagicMock是Mock大多数魔术方法的默认实现的子类。您可以使用，MagicMock而无需自己配置魔术方法。

构造函数参数的含义与Mock。

如果您使用*spec*或*spec_set*参数，则*只会*创建spec中存在的魔术方法。

class unittest.mock.**NonCallableMagicMock**(**args*, ***kw*)

一个不可调用的版本MagicMock。

MagicMock除了*return_value*和 *side_effect*之外，构造函数参数的含义与此相同，在不可调用的模拟中没有意义。

神奇的方法是用MagicMock对象设置的，所以你可以配置它们并以通常的方式使用它们：

```
>>> mock = MagicMock()
>>> mock[3] = 'fish'
>>> mock.__setitem__.assert_called_with(3, 'fish')
>>> mock.__getitem__.return_value = 'result'
>>> mock[2]
'result'
```

默认情况下，许多协议方法都需要返回特定类型的对象。这些方法预先配置了一个默认返回值，这样如果您对返回值不感兴趣，就可以使用这些方法，而无需执行任何操作。如果您想更改默认值，您仍然可以手动*设置*返回值。

方法及其默认值：

- __lt__ : 未实现
- __gt__ : 未实现
- __le__ : 未实现
- __ge__ : 未实现
- __int__ : 1
- __contains__ : 假
- __len__ : 0
- __iter__ : iter ([])
- __exit__ : 假
- __complex__ : 1j
- __float__ : 1.0
- __bool__ 真的
- __index__ : 1
- __hash__ : 模拟的默认哈希
- __str__ : 模拟的默认str
- __sizeof__ : 模拟的默认sizeof

例如：

```
>>> mock = MagicMock()
>>> int(mock)
1
>>> len(mock)
0
>>> list(mock)
[]
>>> object() in mock
False
```

这两种方法的平等，__eq__()和__ne__()，是特殊的。side_effect除非您更改其返回值以返回其他值，否则使用该属性对身份进行默认的等式比较 ：

```
>>> MagicMock() == 3
False
>>> MagicMock() != 3
True
>>> mock = MagicMock()
>>> mock.__eq__.return_value = True
>>> mock == 3
True
```

返回值MagicMock.__iter__()可以是任何可迭代对象，不需要是迭代器：

```
>>> mock = MagicMock()
>>> mock.__iter__.return_value = ['a', 'b', 'c']
>>> list(mock)
['a', 'b', 'c']
>>> list(mock)
['a', 'b', 'c']
```

如果返回值*是*一个迭代器，那么迭代一次就会消耗它，随后的迭代将产生一个空列表：

```
>>> mock.__iter__.return_value = iter(['a', 'b', 'c'])
>>> list(mock)
['a', 'b', 'c']
>>> list(mock)
[]
```

MagicMock除了一些模糊和过时的配置之外，还配置了所有支持的魔法方法。你仍然可以设置这些，如果你想。

默认支持但未设置的魔术方法MagicMock是：

- __subclasses__
- __dir__
- __format__
- __get__ ，__set__和__delete__
- __reversed__ 和 __missing__
- __reduce__ ， __reduce_ex__ ， __getinitargs__ ， __getnewargs__ ， __getstate__ 和 __setstate__
- __getformat__ 和 __setformat__

[2] 魔法方法*应该*在课堂而不是实例上查找。不同版本的Python在应用此规则时不一致。支持的协议方法应该适用于所有支持的Python版本。

[3] 该功能基本上与班级相连，但每个Mock 实例都与其他人保持隔离。

26.5.5。助手

26.5.5.1。哨兵

unittest.mock. **sentinel**

该sentinel对象提供了一种为测试提供独特对象的便捷方式。

当您按名称访问属性时，会按需创建属性。访问相同的属性将始终返回相同的对象。返回的对象具有合理的repr，以便测试失败消息可读。

这些sentinel属性在它们是copied或时不会保留它们的身份 pickled。

有时候在测试时需要测试一个特定的对象作为参数传递给另一个方法，或者返回。创建命名的哨兵对象来测试它是很常见的。 sentinel提供了一种创建和测试这种对象身份的便捷方式。

在这个例子中，我们使用补丁method来返回sentinel.some_object ：

```
>>> real = ProductionClass()
>>> real.method = Mock(name="method")
>>> real.method.return_value = sentinel.some_object
>>> result = real.method()
>>> assert result is sentinel.some_object
>>> sentinel.some_object
sentinel.some_object
```

26.5.5.2。默认

unittest.mock.DEFAULT

该DEFAULT对象是一个预先创建的哨兵（实际上 sentinel.DEFAULT）。它可以被side_effect 函数用来指示应该使用正常的返回值。

26.5.5.3。调用

unittest.mock.call(**args*, ***kwargs*)

call()是制作简单断言，对于比较辅助对象 call_args，call_args_list， mock_calls和method_calls。call()也可以搭配使用assert_has_calls()。

```
>>> m = MagicMock(return_value=None)
>>> m(1, 2, a='foo', b='bar')
>>> m()
>>> m.call_args_list == [call(1, 2, a='foo', b='bar'), call()]
True
```

call.call_list()

对于表示多个呼叫的呼叫对象，call_list() 返回所有中间呼叫以及最终呼叫的列表。

call_list对于“链接呼叫”做出断言特别有用。链接的呼叫是在一行代码上进行多次呼叫。这导致mock_calls模拟中有多个条目。手动构建调用序列可能很乏味。

call_list() 可以从相同的链接呼叫构建呼叫序列：

```
>>> m = MagicMock()
>>> m(1).method(arg='foo').other('bar')(2.0)
<MagicMock name='mock().method().other()()' id='...'>
>>> kall = call(1).method(arg='foo').other('bar')(2.0)
>>> kall.call_list()
[call(1),
 call().method(arg='foo'),
 call().method().other('bar'),
 call().method().other()(2.0)]
>>> m.mock_calls == kall.call_list()
True
```

一个call对象可以是（位置参数，关键字参数）或（名称，位置参数，关键字参数）的元组，取决于它的构造方式。当你自己构建他们这是不是特别有趣，但是call 那是在对象Mock.call_args，Mock.call_args_list和 Mock.mock_calls属性可以内省获得在它们所包含的各个参数。

将call在对象Mock.call_args和Mock.call_args_list 是二元组（位置指定参数时，关键字参数），而call在对象Mock.mock_calls，与那些沿着你自己构建，是（名称，位置指定参数时，关键字参数）三元组。

你可以使用他们的“元组”来抽出更多复杂的内省和断言的各个参数。位置参数是一个元组（如果没有位置参数，则为空元组）并且关键字参数是字典：

```
>>> m = MagicMock(return_value=None)
>>> m(1, 2, 3, arg='one', arg2='two')
```

```
>>> kall = m.call_args
>>> args, kwargs = kall
>>> args
(1, 2, 3)
>>> kwargs
{'arg2': 'two', 'arg': 'one'}
>>> args is kall[0]
True
>>> kwargs is kall[1]
True
```

```
>>> m = MagicMock()
>>> m.foo(4, 5, 6, arg='two', arg2='three')
<MagicMock name='mock.foo()' id='...'>
>>> kall = m.mock_calls[0]
>>> name, args, kwargs = kall
>>> name
'foo'
>>> args
(4, 5, 6)
>>> kwargs
{'arg2': 'three', 'arg': 'two'}
>>> name is m.mock_calls[0][0]
True
```

26.5.5.4。create_autospec

unittest.mock. create_autospec (*spec* , *spec_set* = *False* , *instance* = *False* , ** *kwargs*)

使用另一个对象作为规格创建一个模拟对象。模拟上的属性将使用*spec*对象上的相应属性作为规格。

被模拟的函数或方法将检查它们的参数，以确保它们被正确的签名调用。

如果*spec_set*被True然后尝试设置不规范对象上存在会引发一个属性AttributeError。

如果一个类被用作spec，那么模拟（类的实例）的返回值将具有相同的规格。您可以通过传递将类用作实例对象的规范instance=True。如果模拟的实例可调用，返回的模拟只能被调用。

create_autospec() 还会传递给传递给创建的模拟构造函数的任意关键字参数。

见Autospeccing有关如何使用自动speccing结合实例 create_autospec() 和*autospec*参数patch()。

26.5.5.5。任何

unittest.mock. ANY

有时候，你可能需要对模拟调用中的*某些*参数作出断言，但不是关心一些论点，就是想单独将它们拉出来，call_args并对它们做出更复杂的断言。

要忽略某些参数，您可以传递比较等于*一切的*对象 。无论通过什么，呼吁 assert_called_with()和 assert_called_once_with()将会成功。

```
>>> mock = Mock(return_value=None)
>>> mock('foo', bar=object())
>>> mock.assert_called_once_with('foo', bar=ANY)
```

ANY也可以用于与呼叫列表进行比较，如 mock_calls：

```
>>> m = MagicMock(return_value=None)
>>> m(1)
>>> m(1, 2)
>>> m(object())
>>> m.mock_calls == [call(1), call(1, 2), ANY]
True
```

26.5.5.6。FILTER_DIR

unittest.mock. FILTER_DIR

FILTER_DIR是一个模块级变量，用于控制模拟对象响应的方式dir()（仅适用于Python 2.6或更新的版本）。缺省值是True，它使用下面描述的过滤来仅显示有用的成员。如果您不喜欢此过滤功能，或者需要将其关闭以进行诊断，请进行设置。mock.FILTER_DIR = False

通过过滤，dir(some_mock)仅显示有用的属性，并包含通常不会显示的任何动态创建的属性。如果模拟是用*spec*（或*autospec*当然）创建的，则显示原始的所有属性，即使它们尚未被访问：

```
>>> dir(Mock())
['assert_any_call',
 'assert_called_once_with',
 'assert_called_with',
 'assert_has_calls',
 'attach_mock',
 ...
>>> from urllib import request
>>> dir(Mock(spec=request))
['AbstractBasicAuthHandler',
 'AbstractDigestAuthHandler',
 'AbstractHTTPHandler',
 'BaseHandler',
 ...
```

许多不太有用的（私有的Mock而不是被模拟的东西）下划线和双下划线前缀属性已经从调用 dir()a 的结果中过滤掉了Mock。如果您不喜欢这种行为，您可以通过设置模块级别开关将其关闭 FILTER_DIR：

```
>>> from unittest import mock
>>> mock.FILTER_DIR = False
>>> dir(mock.Mock())
['_NonCallableMock__get_return_value',
 '_NonCallableMock__get_side_effect',
```

```
 '_NonCallableMock__return_value_doc',
 '_NonCallableMock__set_return_value',
 '_NonCallableMock__set_side_effect',
 '__call__',
 '__class__',
 ...
```

或者，您可以使用vars(my_mock)（实例成员）和 dir(type(my_mock))（类型成员）绕过过滤，而不考虑 mock.FILTER_DIR。

26.5.5.7。mock_open

unittest.mock. mock_open（ *mock = None*，*read_data = None* ）

> 一个辅助函数来创建一个模拟来替换使用open()。它适用于open()直接调用或用作上下文管理器。
>
> 该*模拟*参数为模拟对象进行配置。如果None（默认）然后MagicMock将为您创建一个，API限于标准文件句柄上可用的方法或属性。
>
> *的read_data*是为一个字符串read()， readline()和readlines()文件句柄的方法来返回。调用这些方法将从*read_data*获取数据，直到数据 耗尽。这些方法的模拟非常简单：每次调用*模拟时*，*read_data*都会重新开始。如果您需要更多地控制您提供给测试代码的数据，则需要自行定制此模拟。当这一点不足时，PyPI上的一个内存中文件系统软件包可以提供一个真实的文件系统进行测试。

*在版本3.4中更改：*添加readline()和readlines()支持。 read()改变模拟使用*read_data*，而不是在每次调用时都返回它。

*在版本3.5中更改：read_data*现在每次调用*模拟时*都会重置。

使用open()上下文管理器是确保文件句柄正确关闭并且变得常见的好方法：

```
with open('/some/path', 'w') as f:
    f.write('something')
```

问题是，即使你模拟出调用open()它是*返回的对象*是用作上下文管理器（并具有__enter__()和 __exit__()调用）。

用a MagicMock来嘲讽上下文管理者是非常普遍和足够的，以至于辅助函数是有用的。

```
>>> m = mock_open()
>>> with patch('__main__.open', m):
...     with open('foo', 'w') as h:
...         h.write('some stuff')
...
>>> m.mock_calls
[call('foo', 'w'),
 call().__enter__(),
 call().write('some stuff'),
 call().__exit__(None, None, None)]
>>> m.assert_called_once_with('foo', 'w')
```

```
>>> handle = m()
>>> handle.write.assert_called_once_with('some stuff')
```

并阅读文件：

```
>>> with patch('__main__.open', mock_open(read_data='bibble')) as m:
...     with open('foo') as h:
...         result = h.read()
...
>>> m.assert_called_once_with('foo')
>>> assert result == 'bibble'
```

26.5.5.8。Autospeccing

Autospeccing基于现有spec的模拟功能。它将mock的api限制为原始对象的api（spec），但是它是递归的（延迟实现），因此mock的属性只与spec的属性具有相同的api。另外，嘲笑的函数/方法与原始函数/方法具有相同的调用签名，因此TypeError如果它们被错误地调用，则会引发它们。

在我解释自动搜索如何工作之前，这里就是为什么需要它。

Mock是一个非常强大和灵活的对象，但在用于从被测系统中剔除对象时会遇到两个缺陷。其中一个缺陷是Mockapi 特有的，另一个是使用模拟对象时更常见的问题。

首先是具体的问题Mock。 Mock有两个非常方便的断言方法：assert_called_with()和assert_called_once_with()。

```
>>> mock = Mock(name='Thing', return_value=None)
>>> mock(1, 2, 3)
>>> mock.assert_called_once_with(1, 2, 3)
>>> mock(1, 2, 3)
>>> mock.assert_called_once_with(1, 2, 3)
Traceback (most recent call last):
 ...
AssertionError: Expected 'mock' to be called once. Called 2 times.
```

因为嘲笑按需自动创建属性，并允许用任意参数调用它们，所以如果拼错这些断言方法之一，那么断言就消失了：

```
>>> mock = Mock(name='Thing', return_value=None)
>>> mock(1, 2, 3)
>>> mock.assret_called_once_with(4, 5, 6)
```

由于输入错误，您的测试可能会无声无息地通过。

第二个问题是更一般的嘲弄。如果你重构了一些你的代码，重命名成员等等，那么仍然使用*旧api*但是使用mock而不是真实对象的代码的任何测试仍然 会通过。这意味着即使代码已损坏，您的测试也可以通过。

请注意，这是您需要集成测试以及单元测试的另一个原因。独立测试一切都很好，但如果你不测试你的单元是如何“连接在一起”的，测试可能会遇到的错误仍然存在很大的空间。

mock已经提供了一个功能来帮助解决这个问题，称为speccing。如果您使用类或实例作为spec模拟，那么您只能访问真实类中存在的模拟属性：

```
>>> from urllib import request
>>> mock = Mock(spec=request.Request)
>>> mock.assret_called_with
Traceback (most recent call last):
 ...
AttributeError: Mock object has no attribute 'assret_called_with'
```

该规范仅适用于模拟本身，所以我们仍然有与模拟上的任何方法相同的问题：

```
>>> mock.has_data()
<mock.Mock object at 0x...>
>>> mock.has_data.assret_called_with()
```

自动指定解决了这个问题。你可以传递autospec=True给 patch()/ patch.object()或使用该create_autospec()函数来创建一个具有规范的模拟。如果您使用autospec=True参数，patch()那么被替换的对象将被用作spec对象。由于specively是“懒惰地”完成的（规范是在模拟的属性被访问时创建的），所以您可以将它用于非常复杂或深度嵌套的对象（如导入导入模块的模块），而不会对性能造成太大影响。

这里有一个它正在使用的例子：

```
>>> from urllib import request
>>> patcher = patch('__main__.request', autospec=True)
>>> mock_request = patcher.start()
>>> request is mock_request
True
>>> mock_request.Request
<MagicMock name='request.Request' spec='Request' id='...'>
```

你可以看到request.Request有一个规范。 request.Request在构造函数中有两个参数（其中一个是*self*）。如果我们尝试错误地调用它，会发生什么情况：

```
>>> req = request.Request()
Traceback (most recent call last):
 ...
TypeError: <lambda>() takes at least 2 arguments (1 given)
```

该规范也适用于实例化的类（即specible mocks的返回值）：

```
>>> req = request.Request('foo')
>>> req
<NonCallableMagicMock name='request.Request()' spec='Request' id='...'>
```

Request对象不可调用，所以实例化我们嘲笑的返回值request.Request是一个不可调用的模拟。根据规范，我们断言中的任何拼写错误都会引发错误：

```
>>> req.add_header('spam', 'eggs')
<MagicMock name='request.Request().add_header()' id='...'>
>>> req.add_header.assret_called_with
```

```
Traceback (most recent call last):
 ...
AttributeError: Mock object has no attribute 'assret_called_with'
>>> req.add_header.assert_called_with('spam', 'eggs')
```

在很多情况下，您只需添加autospec=True到现有的 patch() 调用中，然后就可以防止由于拼写错误和api更改而导致的错误。

除了使用*autospec*通过patch()有一个 create_autospec() 用于创建autospecced直接嘲笑：

```
>>> from urllib import request
>>> mock_request = create_autospec(request)
>>> mock_request.Request('foo', 'bar')
<NonCallableMagicMock name='mock.Request()' spec='Request' id='...'>
```

然而，这并不是没有限制和限制，这就是为什么它不是默认行为。为了知道spec对象上有哪些属性可用，autospec必须对规范进行反思（访问属性）。当你在模拟上遍历属性时，原始对象的相应遍历正在发生。如果您的任何特定对象具有可触发代码执行的属性或描述符，那么您可能无法使用自动指定。另一方面，设计你的对象以便内省是安全的[4]。

更严重的问题是，通常在__init__()方法中创建属性并且根本不存在于类中。 *autospec*无法知道任何动态创建的属性并将api限制为可见属性。

```
>>> class Something:
...   def __init__(self):
...     self.a = 33
...
>>> with patch('__main__.Something', autospec=True):
...   thing = Something()
...   thing.a
...
Traceback (most recent call last):
  ...
AttributeError: Mock object has no attribute 'a'
```

解决这个问题有几种不同的方法。最简单但不一定是最不讨厌的方法是在创建之后简单地在模拟上设置所需的属性。仅仅因为*autospec*不允许你获取规范中不存在的属性，它并不妨碍你设置它们：

```
>>> with patch('__main__.Something', autospec=True):
...   thing = Something()
...   thing.a = 33
...
```

*spec*和*autospec*都有一个更激进的版本，它*可以*防止你设置不存在的属性。如果你想确保你的代码只*设置*有效的属性，这很有用，但显然它可以防止这种特殊情况：

```
>>> with patch('__main__.Something', autospec=True, spec_set=True):
...   thing = Something()
...   thing.a = 33
...
Traceback (most recent call last):
```

```
 ...
AttributeError: Mock object has no attribute 'a'
```

解决此问题的最佳方法可能是将类属性添加为初始化的实例成员的默认值`__init__()`。请注意，如果您只设置默认属性，`__init__()`然后通过类属性提供它们（当然实例之间共享）也更快。例如

```
class Something:
    a = 33
```

这带来了另一个问题。提供None成员的默认值是相对常见的，这些成员稍后将成为不同类型的对象。 None作为一个规范将是无用的，因为它不会让你访问*任何*属性或方法。由于None是*永远*将是有用的天赋，而且很可能表明，通常一些其他类型的，autospec不使用规范的被设置为成员的成员None。这些只是普通的嘲笑（好 - MagicMocks）：

```
>>> class Something:
...     member = None
...
>>> mock = create_autospec(Something)
>>> mock.member.foo.bar.baz()
<MagicMock name='mock.member.foo.bar.baz()' id='...'>
```

如果修改您的生产类别以添加默认值不符合您的喜好，那么还有更多选项。其中之一就是将实例用作规范而不是类。另一种是创建生产类的子类，并将默认值添加到子类中而不影响生产类。这两个要求您使用替代对象作为规格。谢天谢地`patch()`支持这一点 - 您可以简单地通过替代对象作为*autospec*参数：

```
>>> class Something:
...   def __init__(self):
...     self.a = 33
...
>>> class SomethingForTest(Something):
...   a = 33
...
>>> p = patch('__main__.Something', autospec=SomethingForTest)
>>> mock = p.start()
>>> mock.a
<NonCallableMagicMock name='Something.a' spec='int' id='...'>
```

[4] 这只适用于类或已经实例化的对象。调用一个模拟类创建一个模拟实例*不会*创建一个真实的实例。只有属性查找 - 以及调用`dir()` - 完成。

26.6。unittest.mock- 开始

3.3版本的新功能

26.6.1。使用模拟

26.6.1.1。模拟修补方法

Mock对象的常见用途包括：

- 修补方法
- 记录方法调用对象

您可能需要替换对象上的某个方法，以检查是否由系统的另一部分使用正确的参数调用它：

```
>>> real = SomeClass()
>>> real.method = MagicMock(name='method')
>>> real.method(3, 4, 5, key='value')
<MagicMock name='method()' id='...'>
```

一旦我们的模拟已经被使用（real.method在这个例子中），它就有了方法和属性，可以让你断言它是如何被使用的。

注意： 在大多数这些例子中Mock，MagicMock类和类是可以互换的。因为这MagicMock是更有能力的类，所以默认情况下使用它是明智的。

一旦模拟被称为它的called属性设置为 True。更重要的是，我们可以使用assert_called_with()or assert_called_once_with()方法来检查它是否被正确的参数调用。

此示例测试调用ProductionClass().method结果是否调用该something方法：

```
>>> class ProductionClass:
...     def method(self):
...         self.something(1, 2, 3)
...     def something(self, a, b, c):
...         pass
...
>>> real = ProductionClass()
>>> real.something = MagicMock()
>>> real.method()
>>> real.something.assert_called_once_with(1, 2, 3)
```

26.6.1.2。模拟方法调用对象

在最后一个例子中，我们直接在对象上修补了一个方法来检查它是否被正确调用。另一个常见用例是将一个对象传递给一个方法（或被测系统的某个部分），然后检查它是否以正确的方式使用。

简单的ProductionClass下面有一个closer方法。如果它被一个对象调用close，那么它会调用它。

```
>>> class ProductionClass:
...     def closer(self, something):
...         something.close()
...
```

所以为了测试它，我们需要用一个close方法传入一个对象并检查它是否被正确调用。

```
>>> real = ProductionClass()
>>> mock = Mock()
>>> real.closer(mock)
>>> mock.close.assert_called_with()
```

我们不需要做任何工作来为我们的模拟提供'接近'的方法。访问关闭创建它。所以，如果'close'尚未被调用，那么在测试中访问它会创建它，但assert_called_with() 会引发失败异常。

26.6.1.3。嘲笑类

一个常见的用例是模拟被测试代码实例化的类。当你修补一个班级时，那个班级被替换为一个模拟。实例通过*调用类*创建。这意味着你通过查看模拟类的返回值来访问“模拟实例”。

在下面的例子中，我们有一个函数some_function实例化Foo 并调用其上的方法。呼吁用模拟来patch() 替换这个类Foo 。该Foo 实例是调用模拟的结果，因此通过修改模拟进行配置return_value。

```
>>> def some_function():
...     instance = module.Foo()
...     return instance.method()
...
>>> with patch('module.Foo') as mock:
...     instance = mock.return_value
...     instance.method.return_value = 'the result'
...     result = some_function()
...     assert result == 'the result'
```

26.6.1.4。命名你的模拟

给你的嘲笑一个名字是有用的。该名称显示在模拟的repr中，并且当模拟出现在测试失败消息中时可能会有所帮助。该名称也传播到模拟的属性或方法：

```
>>> mock = MagicMock(name='foo')
>>> mock
<MagicMock name='foo' id='...'>
>>> mock.method
<MagicMock name='foo.method' id='...'>
```

26.6.1.5。跟踪所有呼叫

通常你想跟踪一个方法的多次调用。该 mock_calls属性记录所有对模拟儿童属性的调用 - 以及他们的子女。

```
>>> mock = MagicMock()
>>> mock.method()
<MagicMock name='mock.method()' id='...'>
>>> mock.attribute.method(10, x=53)
<MagicMock name='mock.attribute.method()' id='...'>
>>> mock.mock_calls
[call.method(), call.attribute.method(10, x=53)]
```

如果你做出断言mock_calls并且任何意外的方法被调用，那么断言将失败。这很有用，因为除了声明您预期的呼叫已完成之外，您还将检查它们是否按正确的顺序制作，并且没有额外的呼叫：

您可以使用该call对象构造列表以与mock_calls以下内容进行比较

```
>>> expected = [call.method(), call.attribute.method(10, x=53)]
>>> mock.mock_calls == expected
True
```

26.6.1.6。设置返回值和属性

在模拟对象上设置返回值非常简单：

```
>>> mock = Mock()
>>> mock.return_value = 3
>>> mock()
3
```

当然你也可以在模拟方法上做同样的事情：

```
>>> mock = Mock()
>>> mock.method.return_value = 3
>>> mock.method()
3
```

返回值也可以在构造函数中设置：

```
>>> mock = Mock(return_value=3)
>>> mock()
3
```

如果你需要模拟一个属性设置，就这样做：

```
>>> mock = Mock()
>>> mock.x = 3
>>> mock.x
3
```

有时你想模拟一个更复杂的情况，例如 。如果我们希望这个调用返回一个列表，那么我们必须配置嵌套调用的结果。 mock.connection.cursor().execute("SELECT 1")

我们可以用call这样的方式在这样的“链接调用”中构建一组调用，以便随后轻松断言：

```
>>> mock = Mock()
>>> cursor = mock.connection.cursor.return_value
>>> cursor.execute.return_value = ['foo']
>>> mock.connection.cursor().execute("SELECT 1")
['foo']
>>> expected = call.connection.cursor().execute("SELECT 1").call_list()
>>> mock.mock_calls
[call.connection.cursor(), call.connection.cursor().execute('SELECT 1')]
>>> mock.mock_calls == expected
True
```

这是呼叫.call_list()将我们的呼叫对象变成表示链接呼叫的呼叫列表。

26.6.1.7。用模拟　　异常

一个有用的属性是side_effect。如果将此设置为异常类或实例，则在调用模拟时会引发异常。

```
>>> mock = Mock(side_effect=Exception('Boom!'))
>>> mock()
Traceback (most recent call last):
  ...
Exception: Boom!
```

26.6.1.8。副作用函数和迭代器

side_effect也可以设置为一个函数或一个迭代器。 side_effect作为迭代器的用例 是您的模拟将被多次调用的地方，并且您希望每个调用返回不同的值。当您设置 side_effect为可迭代时，对模拟的每次调用都会返回来自迭代器的下一个值：

```
>>> mock = MagicMock(side_effect=[4, 5, 6])
>>> mock()
4
>>> mock()
5
>>> mock()
6
```

对于更高级的用例，例如根据调用的模拟动态改变返回值， side_effect可以是一个函数。该函数将被调用与模拟相同的参数。无论函数返回的是什么调用返回的内容：

```
>>> vals = {(1, 2): 1, (2, 3): 2}
>>> def side_effect(*args):
...     return vals[args]
...
>>> mock = MagicMock(side_effect=side_effect)
>>> mock(1, 2)
```

```
1
>>> mock(2, 3)
2
```

26.6.1.9。从现有对象创建模拟

过度使用嘲讽的一个问题是，它将您的测试与您的模拟实现相结合，而不是真正的代码。假设你有一个实现的类some_method。在另一个课程的测试中，你提供了这个对象的模拟，*也*提供了some_method。如果以后你重构第一课，那么它就不再有了some_method- 那么即使你的代码现在被破坏了，你的测试也会继续通过！

Mock允许您使用*spec*关键字参数提供一个对象作为模拟*规范*。访问规范对象上不存在的模拟方法/属性将立即引发属性错误。如果您更改了规范的实现，那么使用该类的测试将立即开始失败，而无需在这些测试中实例化该类。

```
>>> mock = Mock(spec=SomeClass)
>>> mock.old_method()
Traceback (most recent call last):
   ...
AttributeError: object has no attribute 'old_method'
```

不管某些参数是作为位置参数还是命名参数传递，使用规范还可以更智能地匹配对模拟的调用：

```
>>> def f(a, b, c): pass
...
>>> mock = Mock(spec=f)
>>> mock(1, 2, 3)
<Mock name='mock()' id='140161580456576'>
>>> mock.assert_called_with(a=1, b=2, c=3)
```

如果你希望这个更聪明的匹配也可以在模拟上使用方法调用，你可以使用自动指定。

如果你想要一个更强大的规范形式来防止设置任意属性以及获取它们，那么你可以使用*spec_set*而不是*spec*。

26.6.2。修补程序装饰器

注意： 随着patch()它的事项，你在他们抬头的命名空间修补的对象。这通常很简单，但要获得快速指南，请参阅补丁的位置。

测试中的一个常见需求是修补类属性或模块属性，例如修补内建或修补模块中的类以测试其实例化。模块和类实际上是全球性的，因此修补它们必须在测试之后撤消，否则修补程序将持续到其他测试并导致难以诊断问题。

mock为此提供了三个方便的装饰器：patch() ，patch.object()和 patch.dict()。 patch采用表单的单个字符串 package.module.Class.attribute来指定您正在修补的属性。它也可以选择一个你想要的属性（或类或其他）被替换的值。'patch.object'需要一个对象和你想修补的属性的名称，以及可选的修补值。

patch.object：

```
>>> original = SomeClass.attribute
>>> @patch.object(SomeClass, 'attribute', sentinel.attribute)
... def test():
...     assert SomeClass.attribute == sentinel.attribute
...
>>> test()
>>> assert SomeClass.attribute == original
```

```
>>> @patch('package.module.attribute', sentinel.attribute)
... def test():
...     from package.module import attribute
...     assert attribute is sentinel.attribute
...
>>> test()
```

如果你正在修补一个模块（包括builtins），那么使用它patch() 来代替patch.object()：

```
>>> mock = MagicMock(return_value=sentinel.file_handle)
>>> with patch('builtins.open', mock):
...     handle = open('filename', 'r')
...
>>> mock.assert_called_with('filename', 'r')
>>> assert handle == sentinel.file_handle, "incorrect file handle returned"
```

模块名称可以是'虚线'，package.module如果需要，可以采用以下格式：

```
>>> @patch('package.module.ClassName.attribute', sentinel.attribute)
... def test():
...     from package.module import ClassName
...     assert ClassName.attribute == sentinel.attribute
...
>>> test()
```

一个很好的模式是实际装饰测试方法本身：

```
>>> class MyTest(unittest.TestCase):
...     @patch.object(SomeClass, 'attribute', sentinel.attribute)
...     def test_something(self):
...         self.assertEqual(SomeClass.attribute, sentinel.attribute)
...
>>> original = SomeClass.attribute
>>> MyTest('test_something').test_something()
>>> assert SomeClass.attribute == original
```

如果你想用Mock打补丁，只能使用patch()一个参数（或者patch.object()带两个参数）。该模拟将为您创建并传递到测试函数/方法中：

```
>>> class MyTest(unittest.TestCase):
...     @patch.object(SomeClass, 'static_method')
...     def test_something(self, mock_method):
...         SomeClass.static_method()
...         mock_method.assert_called_with()
```

```
...
>>> MyTest('test_something').test_something()
```

您可以使用以下模式堆叠多个修补程序装饰器：

```
>>> class MyTest(unittest.TestCase):
...     @patch('package.module.ClassName1')
...     @patch('package.module.ClassName2')
...     def test_something(self, MockClass2, MockClass1):
...         self.assertIs(package.module.ClassName1, MockClass1)
...         self.assertIs(package.module.ClassName2, MockClass2)
...
>>> MyTest('test_something').test_something()
```

当您嵌套修补程序修饰符时，这些修改将以与它们应用的顺序相同的顺序（装饰器应用的正常*Python*顺序）传递到装饰函数中。这意味着从下往上，所以在上面的例子中，模拟test_module.ClassName2是先传入的。

也patch.dict()可以在范围内在字典中设置值，并在测试结束时将字典恢复到原始状态：

```
>>> foo = {'key': 'value'}
>>> original = foo.copy()
>>> with patch.dict(foo, {'newkey': 'newvalue'}, clear=True):
...     assert foo == {'newkey': 'newvalue'}
...
>>> assert foo == original
```

patch，patch.object并且patch.dict都可以用作上下文管理器。

在你用来patch()为你创建一个模拟的地方，你可以使用with语句的“as”形式获得对模拟的引用：

```
>>> class ProductionClass:
...     def method(self):
...         pass
...
>>> with patch.object(ProductionClass, 'method') as mock_method:
...     mock_method.return_value = None
...     real = ProductionClass()
...     real.method(1, 2, 3)
...
>>> mock_method.assert_called_with(1, 2, 3)
```

作为替代方案patch，patch.object并且patch.dict可以用作类装饰器。以这种方式使用时，它与将装饰器单独应用于名称以“test”开头的每个方法相同。

26.6.3。更多示例

以下是一些稍微更高级的场景的更多示例。

26.6.3.1。嘲笑链式呼叫

一旦你理解了这个return_value属性，嘲笑链式调用实际上是直接的。当第一次调用模拟器时，或者return_value在调用模拟器之前获取模拟器时，Mock会创建一个新模型。

这意味着您可以通过询问return_value模拟来查看从对调用对象的调用返回的对象是如何使用的：

```
>>> mock = Mock()
>>> mock().foo(a=2, b=3)
<Mock name='mock().foo()' id='...'>
>>> mock.return_value.foo.assert_called_with(a=2, b=3)
```

从这里开始，配置一个简单的步骤，然后对链接的调用进行断言。当然，另一种选择是首先以更易测试的方式编写代码。

所以，假设我们有一些代码看起来有点像这样：

```
>>> class Something:
...     def __init__(self):
...         self.backend = BackendProvider()
...     def method(self):
...         response = self.backend.get_endpoint('foobar').create_call('spam', 'eggs')
...         # more code
```

假设这BackendProvider已经很好的测试了，我们该如何测试 method() ？具体来说，我们要测试代码段以正确的方式使用响应对象。 # more code

由于这个调用链是由一个实例属性组成的，我们可以backend在一个Something实例上修补这个属性。在这种特殊情况下，我们只对最后一次调用的返回值感兴趣， start_call所以我们没有太多配置要做。让我们假设它返回的对象是'类文件'，所以我们将确保我们的响应对象使用内置open()作为它的对象spec。

为此，我们创建一个模拟实例作为我们的模拟后端，并为其创建一个模拟响应对象。要将响应设置为最终的返回值， start_call我们可以这样做：

```
mock_backend.get_endpoint.return_value.create_call.return_value.start_call.return_valu
```

我们可以通过使用该configure_mock() 方法以更好的方式直接为我们设置返回值：

```
>>> something = Something()
>>> mock_response = Mock(spec=open)
>>> mock_backend = Mock()
>>> config = {'get_endpoint.return_value.create_call.return_value.start_call.return_va
>>> mock_backend.configure_mock(**config)
```

有了这些，我们猴子就修补了“模拟后端”，并且可以做出真正的呼叫：

```
>>> something.backend = mock_backend
>>> something.method()
```

使用mock_calls我们可以用单个断言来检查链接的调用。链接呼叫是一行代码中的多个呼叫，因此会有几个条目mock_calls。我们可以用来call.call_list()为我们创建这个电话列表：

```
>>> chained = call.get_endpoint('foobar').create_call('spam', 'eggs').start_call()
>>> call_list = chained.call_list()
>>> assert mock_backend.mock_calls == call_list
```

26.6.3.2。部分模拟

在一些测试中，我想模拟一个调用来datetime.date.today() 返回一个已知日期，但我不想阻止测试中的代码创建新的日期对象。不幸的datetime.date是用C语言编写的，所以我不能只用猴子修补静态date.today()方法。

我找到了一个简单的方法来完成这个工作，它包括用模拟有效地包装日期类，但是将构造函数的调用传递给真正的类（并返回实际的实例）。

在这里用来骂出被测模块类。然后 将模拟日期类中的属性设置为返回实际日期的lambda函数。当模拟日期类被调用时，真正的日期将被构造并返回。 patch decoratordateside_effectside_effect

```
>>> from datetime import date
>>> with patch('mymodule.date') as mock_date:
...     mock_date.today.return_value = date(2010, 10, 8)
...     mock_date.side_effect = lambda *args, **kw: date(*args, **kw)
...
...     assert mymodule.date.today() == date(2010, 10, 8)
...     assert mymodule.date(2009, 6, 8) == date(2009, 6, 8)
...
```

请注意，我们不会datetime.date全局打补丁，我们会date在*使用*它的模块中打补丁。看哪里补丁。

当date.today()被调用时返回一个已知日期，但对date(...)构造函数的调用 仍然返回正常日期。如果没有这个，你可能会发现自己必须使用与被测代码完全相同的算法来计算预期结果，这是一种典型的测试反模式。

对日期构造函数的调用被记录在对你的测试有用的mock_date属性（ call_count和朋友）中。

本博客文章讨论了处理嘲讽日期或其他内置类的另一种方法。

26.6.3.3。嘲笑一个生成器方法

Python生成器是一种函数或方法，它yield在迭代[1]时使用该语句返回一系列值。

调用生成器方法/函数以返回生成器对象。这是生成器对象，然后迭代。迭代的协议方法是__iter__()，所以我们可以使用a来嘲笑它MagicMock。

下面是一个带有作为生成器实现的“iter”方法的示例类：

```
>>> class Foo:
...     def iter(self):
...         for i in [1, 2, 3]:
...             yield i
...
>>> foo = Foo()
>>> list(foo.iter())
[1, 2, 3]
```

我们如何嘲笑这个课程，特别是它的“iter”方法？

要配置从迭代返回的值（在调用中隐含 `list` ），我们需要配置调用返回的对象`foo.iter()`。

```
>>> mock_foo = MagicMock()
>>> mock_foo.iter.return_value = iter([1, 2, 3])
>>> list(mock_foo.iter())
[1, 2, 3]
```

[1] 还有发生器表达式和更高级的发生器使用，但我们在这里并不关心它们。对发电机的一个很好的介绍以及它们的强大程度是：系统程序员的发电机技巧。

26.6.3.4。对每个测试方法应用相同的补丁

如果你想为多种测试方法提供多个补丁，显而易见的方法是将补丁装饰器应用于每种方法。这可能会感觉不必要的重复。对于Python 2.6或更新版本，您可以使用`patch()`（以各种形式）作为类装饰器。这会将修补程序应用于该类上的所有测试方法。测试方法通过名称以`test`下列方法开头的方法来标识：

```
>>> @patch('mymodule.SomeClass')
... class MyTest(TestCase):
...
...     def test_one(self, MockSomeClass):
...         self.assertIs(mymodule.SomeClass, MockSomeClass)
...
...     def test_two(self, MockSomeClass):
...         self.assertIs(mymodule.SomeClass, MockSomeClass)
...
...     def not_a_test(self):
...         return 'something'
...
>>> MyTest('test_one').test_one()
>>> MyTest('test_two').test_two()
>>> MyTest('test_two').not_a_test()
'something'
```

管理修补程序的另一种方法是使用修补程序方法：启动和停止。这允许你修补进入你`setUp`和`tearDown`方法。

```
>>> class MyTest(TestCase):
...     def setUp(self):
...         self.patcher = patch('mymodule.foo')
...         self.mock_foo = self.patcher.start()
...
```

```
...     def test_foo(self):
...         self.assertIs(mymodule.foo, self.mock_foo)
...
...     def tearDown(self):
...         self.patcher.stop()
...
>>> MyTest('test_foo').run()
```

如果您使用这种技术，您必须确保修补程序通过调用“撤消” `stop`。这可能比你想象的要烦琐，因为如果在setUp中引发异常，那么不会调用tearDown。 `unittest.TestCase.addCleanup()` 使这更容易：

```
>>> class MyTest(TestCase):
...     def setUp(self):
...         patcher = patch('mymodule.foo')
...         self.addCleanup(patcher.stop)
...         self.mock_foo = patcher.start()
...
...     def test_foo(self):
...         self.assertIs(mymodule.foo, self.mock_foo)
...
>>> MyTest('test_foo').run()
```

26.6.3.5。嘲弄未绑定的方法

今天写测试时，我需要修补一个*未绑定的方法*（在类上而不是在实例上修补方法）。我需要自我作为第一个参数传入，因为我想断言哪些对象正在调用此特定方法。问题是你不能用这个模拟补丁，因为如果你用一个模拟来替换一个未绑定的方法，那么当它从实例中获取时它不会成为一个绑定的方法，所以它不会自我传入。解决方法是使用真正的函数修改未绑定的方法。该 `patch()` 装饰使得它如此简单修补了方法与模拟是不必创建一个真正的功能变得令人讨厌。

如果你传递`autospec=True`给补丁，那么它会用一个*真实的*函数对象进行补丁 。这个函数对象与它正在替换的函数对象具有相同的签名，但委托给引擎盖下的模拟器。您仍然可以像以前一样使用自动创建的模型。它的意思是，如果你用它来修补一个未绑定的类的方法，模拟函数将被转化为一个绑定的方法，如果它是从一个实例中获取的。它会`self`作为第一个参数传入，这正是我想要的：

```
>>> class Foo:
...   def foo(self):
...     pass
...
>>> with patch.object(Foo, 'foo', autospec=True) as mock_foo:
...   mock_foo.return_value = 'foo'
...   foo = Foo()
...   foo.foo()
...
'foo'
>>> mock_foo.assert_called_once_with(foo)
```

如果我们不使用，`autospec=True`那么未绑定的方法将用Mock实例修补，而不是使用`self`。

26.6.3.6。用模拟检查多个调用

mock有一个很好的API用于断言你的模拟对象是如何使用的。

```
>>> mock = Mock()
>>> mock.foo_bar.return_value = None
>>> mock.foo_bar('baz', spam='eggs')
>>> mock.foo_bar.assert_called_with('baz', spam='eggs')
```

如果你的模拟只被调用一次，你可以使用该 assert_called_once_with() 方法，也断言这call_count是一个。

```
>>> mock.foo_bar.assert_called_once_with('baz', spam='eggs')
>>> mock.foo_bar()
>>> mock.foo_bar.assert_called_once_with('baz', spam='eggs')
Traceback (most recent call last):
    ...
AssertionError: Expected to be called once. Called 2 times.
```

双方assert_called_with并assert_called_once_with做出断言*最近*通话。如果您的模拟将被多次调用，并且您想对*所有*可以使用的调用 作出断言call_args_list：

```
>>> mock = Mock(return_value=None)
>>> mock(1, 2, 3)
>>> mock(4, 5, 6)
>>> mock()
>>> mock.call_args_list
[call(1, 2, 3), call(4, 5, 6), call()]
```

该call助手可以很容易地做出这些调用断言。您可以建立预期呼叫列表并将其与之进行比较call_args_list。这看起来非常类似于call_args_list：

```
>>> expected = [call(1, 2, 3), call(4, 5, 6), call()]
>>> mock.call_args_list == expected
True
```

26.6.3.7。应对可变参数

另一种情况是罕见的，但可以咬你，是当你的模拟被调用的可变参数。 call_args 并call_args_list存储*对*参数的*引用*。如果参数被测试代码改变了，那么当模拟被调用时，你不能再断言什么是值。

以下是一些显示问题的示例代码。想象一下'mymodule'中定义的以下函数：

```
def frob(val):
    pass

def grob(val):
    "First frob and then clear val"
    frob(val)
    val.clear()
```

当我们试图用正确的参数来测试这个grob调用时frob，看看会发生什么：

```
>>> with patch('mymodule.frob') as mock_frob:
...     val = {6}
...     mymodule.grob(val)
...
>>> val
set()
>>> mock_frob.assert_called_with({6})
Traceback (most recent call last):
    ...
AssertionError: Expected: (({6},), {})
Called with: ((set(),), {})
```

一种可能性是模拟复制你传入的参数。如果你做了依赖于对象标识进行相等的断言，那么这可能会导致问题。

这是一个使用该side_effect 功能的解决方案。如果你为side_effect模拟提供了一个函数，那么 side_effect将被调用与模拟相同的参数。这给了我们一个机会来复制参数并将它们存储起来用于以后的断言。在这个例子中，我使用*另一个*模拟来存储参数，以便我可以使用模拟方法来执行断言。再次，帮助函数为我设置了这个。

```
>>> from copy import deepcopy
>>> from unittest.mock import Mock, patch, DEFAULT
>>> def copy_call_args(mock):
...     new_mock = Mock()
...     def side_effect(*args, **kwargs):
...         args = deepcopy(args)
...         kwargs = deepcopy(kwargs)
...         new_mock(*args, **kwargs)
...         return DEFAULT
...     mock.side_effect = side_effect
...     return new_mock
...
>>> with patch('mymodule.frob') as mock_frob:
...     new_mock = copy_call_args(mock_frob)
...     val = {6}
...     mymodule.grob(val)
...
>>> new_mock.assert_called_with({6})
>>> new_mock.call_args
call({6})
```

copy_call_args被调用的模拟将被调用。它返回一个我们做断言的新模拟。该side_effect函数创建参数的副本并将其new_mock与副本一起呼叫。

注意： 如果你的模拟只会被使用一次，那么在他们被调用的地方有一个更简单的方法来检查参数。你可以简单地在side_effect函数内部进行检查。

```
>>> def side_effect(arg):
...     assert arg == {6}
...
>>> mock = Mock(side_effect=side_effect)
```

```
>>> mock({6})
>>> mock(set())
Traceback (most recent call last):
    ...
AssertionError
```

另一种方法是创建Mock或 MagicMock复制（使用copy.deepcopy() ）参数的子类。这是一个示例实现：

```
>>> from copy import deepcopy
>>> class CopyingMock(MagicMock):
...     def __call__(self, *args, **kwargs):
...         args = deepcopy(args)
...         kwargs = deepcopy(kwargs)
...         return super(CopyingMock, self).__call__(*args, **kwargs)
...
>>> c = CopyingMock(return_value=None)
>>> arg = set()
>>> c(arg)
>>> arg.add(1)
>>> c.assert_called_with(set())
>>> c.assert_called_with(arg)
Traceback (most recent call last):
    ...
AssertionError: Expected call: mock({1})
Actual call: mock(set())
>>> c.foo
<CopyingMock name='mock.foo' id='...'>
```

当你子类Mock或MagicMock所有动态创建的属性，并且return_value将自动使用你的子类。这意味着一个CopyingMock遗嘱的所有孩子也会有这种类型CopyingMock。

26.6.3.8。嵌套修补程序

使用补丁作为上下文管理器是很好的，但是如果你做了多个补丁，你可以用嵌套的语句进行嵌套，进一步向右进一步缩进：

```
>>> class MyTest(TestCase):
...
...     def test_foo(self):
...         with patch('mymodule.Foo') as mock_foo:
...             with patch('mymodule.Bar') as mock_bar:
...                 with patch('mymodule.Spam') as mock_spam:
...                     assert mymodule.Foo is mock_foo
...                     assert mymodule.Bar is mock_bar
...                     assert mymodule.Spam is mock_spam
...
>>> original = mymodule.Foo
>>> MyTest('test_foo').test_foo()
>>> assert mymodule.Foo is original
```

使用unittest cleanup函数和补丁方法：启动和停止我们可以在没有嵌套缩进的情况下实现相同的效果。一个简单的辅助方法，create_patch将补丁放在适当位置，并为我们返回创建的模

拟：

```
>>> class MyTest(TestCase):
...
...     def create_patch(self, name):
...         patcher = patch(name)
...         thing = patcher.start()
...         self.addCleanup(patcher.stop)
...         return thing
...
...     def test_foo(self):
...         mock_foo = self.create_patch('mymodule.Foo')
...         mock_bar = self.create_patch('mymodule.Bar')
...         mock_spam = self.create_patch('mymodule.Spam')
...
...         assert mymodule.Foo is mock_foo
...         assert mymodule.Bar is mock_bar
...         assert mymodule.Spam is mock_spam
...
>>> original = mymodule.Foo
>>> MyTest('test_foo').run()
>>> assert mymodule.Foo is original
```

26.6.3.9。用MagicMock嘲笑字典

你可能想模拟一个字典或其他容器对象，记录对它的所有访问，同时使它仍然像字典一样。

我们可以这样做MagicMock，它将像字典一样工作，并side_effect用于将字典访问委托给我们控制的实际底层字典。

当__getitem__()和__setitem__()我们的方法MagicMock称为（正常字典访问），那么side_effect被称为与键（在的情况下，__setitem__该值也是如此）。我们也可以控制返回的内容。

在使用完之后，MagicMock我们可以使用属性call_args_list来声明字典的使用方式：

```
>>> my_dict = {'a': 1, 'b': 2, 'c': 3}
>>> def getitem(name):
...      return my_dict[name]
...
>>> def setitem(name, val):
...     my_dict[name] = val
...
>>> mock = MagicMock()
>>> mock.__getitem__.side_effect = getitem
>>> mock.__setitem__.side_effect = setitem
```

注意： 使用的另一种方法MagicMock是使用Mock并仅提供您特别需要的魔术方法：

```
>>> mock = Mock()
>>> mock.__getitem__ = Mock(side_effect=getitem)
>>> mock.__setitem__ = Mock(side_effect=setitem)
```

一个*第三*选择是使用MagicMock，但传递dict的*规范*（或*spec_set*）参数，以便MagicMock创建只有现有的字典魔术方法：

```
>>> mock = MagicMock(spec_set=dict)
>>> mock.__getitem__.side_effect = getitem
>>> mock.__setitem__.side_effect = setitem
```

有了这些副作用函数，这个函数mock就像一个普通字典一样记录访问权限。KeyError如果您尝试访问不存在的密钥，它甚至会引发一次。

```
>>> mock['a']
1
>>> mock['c']
3
>>> mock['d']
Traceback (most recent call last):
    ...
KeyError: 'd'
>>> mock['b'] = 'fish'
>>> mock['d'] = 'eggs'
>>> mock['b']
'fish'
>>> mock['d']
'eggs'
```

使用它之后，可以使用正常的模拟方法和属性来对访问进行断言：

```
>>> mock.__getitem__.call_args_list
[call('a'), call('c'), call('d'), call('b'), call('d')]
>>> mock.__setitem__.call_args_list
[call('b', 'fish'), call('d', 'eggs')]
>>> my_dict
{'a': 1, 'c': 3, 'b': 'fish', 'd': 'eggs'}
```

26.6.3.10。模拟子类及其属性

有多种原因可能会导致子类化Mock。一个原因可能是添加辅助方法。这是一个愚蠢的例子：

```
>>> class MyMock(MagicMock):
...     def has_been_called(self):
...         return self.called
...
>>> mymock = MyMock(return_value=None)
>>> mymock
<MyMock id='...'>
>>> mymock.has_been_called()
False
>>> mymock()
>>> mymock.has_been_called()
True
```

Mock实例的标准行为是属性和返回值mock与它们所访问的模拟类型相同。这确保Mock属性Mocks和MagicMock属性是MagicMocks [2]。因此，如果您要继承以添加辅助方法，那么它们也可用于您的子类实例的属性和返回值模拟。

```
>>> mymock.foo
<MyMock name='mock.foo' id='...'>
>>> mymock.foo.has_been_called()
False
>>> mymock.foo()
<MyMock name='mock.foo()' id='...'>
>>> mymock.foo.has_been_called()
True
```

有时这很不方便。例如，一个用户正在创建一个Twisted适配器。将它应用于属性也会导致错误。

Mock（所有风格）都使用一种方法_get_child_mock来创建属性和返回值的这些“子嘲讽”。您可以通过重写此方法来防止您的子类用于属性。签名是它需要任意的关键字参数（**kwargs），然后传递给模拟构造函数：

```
>>> class Subclass(MagicMock):
...     def _get_child_mock(self, **kwargs):
...         return MagicMock(**kwargs)
...
>>> mymock = Subclass()
>>> mymock.foo
<MagicMock name='mock.foo' id='...'>
>>> assert isinstance(mymock, Subclass)
>>> assert not isinstance(mymock.foo, Subclass)
>>> assert not isinstance(mymock(), Subclass)
```

[2] 这能够调用的是不可调用的模拟。属性使用可调用的变体，因为否则不可调用的嘲笑还能够调用的是不。

26.6.3.11。用patch.dict嘲笑导入

模拟可能很困难的一种情况是在函数内部有一个本地导入的地方。这些很难模拟，因为它们没有使用我们可以修补的模块名称空间中的对象。

一般来说，本地进口应避免。它们有时被用来防止循环依赖，*通常*有更好的方法来解决问题（重构代码），或者通过延迟导入来防止“前期成本”。这也可以通过比无条件本地导入更好的方式来解决（将模块存储为类或模块属性，并且仅在首次使用时执行导入）。

除此之外，还有一种方法mock可以影响导入的结果。导入从字典中提取*对象*sys.modules。请注意，它会提取一个*对象*，该*对象*不一定是模块。第一次导入模块会导致模块对象被放入*sys.modules中*，所以通常在导入某个模块时会返回。然而，情况并非如此。

这意味着你可以patch.dict()用来*暂时*放置一个模拟器sys.modules。当这个补丁程序处于活动状态时，任何进口都将获取模拟。当补丁完成时（装饰函数退出，with语句体完成或被patcher.stop()调用），那么之前的任何内容都将被安全地恢复。

这是一个嘲笑'欺骗'模块的例子。

```
>>> mock = Mock()
>>> with patch.dict('sys.modules', {'fooble': mock}):
...     import fooble
...     fooble.blob()
...
<Mock name='mock.blob()' id='...'>
>>> assert 'fooble' not in sys.modules
>>> mock.blob.assert_called_once_with()
```

正如你可以看到成功，但在退出时没有“欺骗”。 `import fooble` `sys.modules`

这也适用于以下形式：`from module import name`

```
>>> mock = Mock()
>>> with patch.dict('sys.modules', {'fooble': mock}):
...     from fooble import blob
...     blob.blip()
...
<Mock name='mock.blob.blip()' id='...'>
>>> mock.blob.blip.assert_called_once_with()
```

稍微更多的工作，你也可以模拟软件包导入：

```
>>> mock = Mock()
>>> modules = {'package': mock, 'package.module': mock.module}
>>> with patch.dict('sys.modules', modules):
...     from package.module import fooble
...     fooble()
...
<Mock name='mock.module.fooble()' id='...'>
>>> mock.module.fooble.assert_called_once_with()
```

26.6.3.12。跟踪调用顺序和较少的详细调用声明

本`Mock`类可以跟踪*订单*上通过你的模拟对象方法的调用`method_calls`属性。这不允许您跟踪单独的模拟对象之间的调用顺序，但我们可以使用它`mock_calls`来实现相同的效果。

因为嘲笑跟踪对子模拟器的调用`mock_calls` ，并且访问模拟的任意属性创建子模拟器，所以我们可以从父模拟器创建单独的模拟器。然后，将这些孩子模拟的电话按顺序记录在`mock_calls`父母的姓名中：

```
>>> manager = Mock()
>>> mock_foo = manager.foo
>>> mock_bar = manager.bar
```

```
>>> mock_foo.something()
<Mock name='mock.foo.something()' id='...'>
>>> mock_bar.other.thing()
<Mock name='mock.bar.other.thing()' id='...'>
```

```
>>> manager.mock_calls
[call.foo.something(), call.bar.other.thing()]
```

然后，我们可以通过比较mock_calls经理模拟中的属性来断言包括订单在内的调用：

```
>>> expected_calls = [call.foo.something(), call.bar.other.thing()]
>>> manager.mock_calls == expected_calls
True
```

如果patch正在创建，并且到位，那么您可以使用该attach_mock()方法将它们附加到经理模拟。附加电话后将记录在mock_calls经理。

```
>>> manager = MagicMock()
>>> with patch('mymodule.Class1') as MockClass1:
...     with patch('mymodule.Class2') as MockClass2:
...         manager.attach_mock(MockClass1, 'MockClass1')
...         manager.attach_mock(MockClass2, 'MockClass2')
...         MockClass1().foo()
...         MockClass2().bar()
...
<MagicMock name='mock.MockClass1().foo()' id='...'>
<MagicMock name='mock.MockClass2().bar()' id='...'>
>>> manager.mock_calls
[call.MockClass1(),
 call.MockClass1().foo(),
 call.MockClass2(),
 call.MockClass2().bar()]
```

如果已经有很多电话，但你只对它们的特定顺序感兴趣，那么另一种方法是使用该assert_has_calls()方法。这需要一个调用列表（由该call对象构造）。如果这个序列的调用是在 mock_calls那个assert成功。

```
>>> m = MagicMock()
>>> m().foo().bar().baz()
<MagicMock name='mock().foo().bar().baz()' id='...'>
>>> m.one().two().three()
<MagicMock name='mock.one().two().three()' id='...'>
>>> calls = call.one().two().three().call_list()
>>> m.assert_has_calls(calls)
```

即使链接的调用m.one().two().three()不是对模拟进行的唯一调用，断言仍然成功。

有时候一个模拟可能会有几个呼叫，而你只关心一些呼叫。你甚至可能不关心订单。在这种情况下，您可以传递any_order=True给assert_has_calls：

```
>>> m = MagicMock()
>>> m(1), m.two(2, 3), m.seven(7), m.fifty('50')
(...)
>>> calls = [call.fifty('50'), call(1), call.seven(7)]
>>> m.assert_has_calls(calls, any_order=True)
```

26.6.3.13。更复杂的参数匹配

使用相同的基本概念，因为ANY我们可以实现匹配器对作为模拟参数的对象执行更复杂的断言。

假设我们期望某个对象被传递给一个默认模式，默认情况下根据对象标识（这是用户定义的类的Python默认值）进行比较。要使用，`assert_called_with()`我们需要传入完全相同的对象。如果我们只对这个对象的某些属性感兴趣，那么我们可以创建一个匹配器来为我们检查这些属性。

你可以在这个例子中看到一个'标准'呼叫`assert_called_with`是不够的：

```
>>> class Foo:
...     def __init__(self, a, b):
...         self.a, self.b = a, b
...
>>> mock = Mock(return_value=None)
>>> mock(Foo(1, 2))
>>> mock.assert_called_with(Foo(1, 2))
Traceback (most recent call last):
    ...
AssertionError: Expected: call(<__main__.Foo object at 0x...>)
Actual call: call(<__main__.Foo object at 0x...>)
```

我们`Foo`班的比较功能可能如下所示：

```
>>> def compare(self, other):
...     if not type(self) == type(other):
...         return False
...     if self.a != other.a:
...         return False
...     if self.b != other.b:
...         return False
...     return True
...
```

而一个可以使用比较函数进行相等操作的匹配器对象如下所示：

```
>>> class Matcher:
...     def __init__(self, compare, some_obj):
...         self.compare = compare
...         self.some_obj = some_obj
...     def __eq__(self, other):
...         return self.compare(self.some_obj, other)
...
```

把所有这些放在一起：

```
>>> match_foo = Matcher(compare, Foo(1, 2))
>>> mock.assert_called_with(match_foo)
```

该`Matcher`实例化与我们的比较功能和`Foo`我们要进行比较的对象。在`assert_called_with`该`Matcher`平等的方法将被调用，它模拟被称为与反对我们同创建了匹配的一个对象进行比较。如果他们匹配，然后 `assert_called_with`通过，如果他们没有`AssertionError`提出：

```
>>> match_wrong = Matcher(compare, Foo(3, 4))
>>> mock.assert_called_with(match_wrong)
Traceback (most recent call last):
```

```
    ...
AssertionError: Expected: ((<Matcher object at 0x...>,), {})
Called with: ((<Foo object at 0x...>,), {})
```

稍微调整一下，你可以让比较函数AssertionError直接提升 并提供更有用的失败信息。

从版本1.5开始，Python测试库PyHamcrest提供了类似的功能，这在其相等匹配器（hamcrest.library.integration.match_equality）的形式中可能很有用。

26.7。2to3 - 自动化的Python 2到3代码翻译

2to3是一个Python程序，它读取Python 2.x源代码并应用一系列*修复*程序将其转换为有效的Python 3.x代码。标准库包含一组丰富的修复程序，可处理几乎所有的代码。`lib2to3`但是，2to3支持库是一个灵活的通用库，因此可以为2to3编写自己的修补程序。 `lib2to3`也可以适用于需要自动编辑Python代码的自定义应用程序。

26.7.1。使用2to3的

通常将Python解释器作为脚本安装2to3。它也位于`Tools/scripts`Python根目录中。

2to3的基本参数是要转换的文件或目录的列表。这些目录是Python源代码的递归遍历。

以下是一个示例Python 2.x源文件，其中`example.py`：

```
def greet(name):
    print "Hello, {0}!".format(name)
print "What's your name?"
name = raw_input()
greet(name)
```

它可以在命令行上通过2to3转换为Python 3.x代码：

```
$ 2to3 example.py
```

针对原始源文件的差异被打印出来。2to3也可以将所需的修改写回源文件。（原始文件的备份除非另有`-n`说明）。使用以下`-w`标志启用更改：

```
$ 2to3 -w example.py
```

改造后，`example.py`看起来像这样：

```
def greet(name):
    print("Hello, {0}!".format(name))
print("What's your name?")
name = input()
greet(name)
```

在整个翻译过程中都会保留评论和确切的缩进。

默认情况下，2to3运行一组预定义的修复程序。该`-l`标志列出了所有可用的修补程序。可以给出一组明确的修复程序来运行`-f`。同样，`-x`显式禁用修复程序。以下示例只运行`imports`和`has_key`修复程序：

```
$ 2to3 -f imports -f has_key example.py
```

该命令运行除修复程序以外的每个`apply`修补程序：

```
$ 2to3 -x apply example.py
```

一些修补程序是*显式的*，意味着它们不是默认运行的，并且必须在命令行上列出才能运行。在此，除了默认的idioms修复程序之外，还会运行修复程序：

```
$ 2to3 -f all -f idioms example.py
```

注意如何传递all启用所有默认修复程序。

有时候2to3会在您的源代码中找到需要更改的地方，但2to3无法自动修复。在这种情况下，2to3会在文件的diff下面打印警告。您应该解决此警告，以获得兼容的3.x代码。

2to3也可以重构doctests。要启用此模式，请使用该-d 标志。请注意，*只有* doctests将被重构。这也不要求该模块是有效的Python。例如，像reST文档中的doctest样例也可以用这个选项重构。

该-v选项可以输出关于翻译过程的更多信息。

由于某些打印语句可以被解析为函数调用或语句，因此2to3不能总是读取包含打印功能的文件。当2to3检测到编译器指令时，它会修改其内部语法以作为函数进行解释。此更改也可以通过标志手动启用。用于 在已经对其打印语句进行转换的代码上运行修复程序。 from __future__ import print_functionprint()-p-p

的-o或--output-dir选项允许处理后的输出文件的备用目录的规范将被写入。 -n使用此标志时需要该 标志，因为备份文件在不覆盖输入文件时没有意义。

3.2.3版-o中的新增功能*：*添加了该选项。

该-W或--write-unchanged-files标志告诉2to3的总是写输出文件即使没有更改所需的文件。这是非常有用的， -o这样整个Python源代码树就可以从一个目录翻译到另一个目录。此选项意味着该-w标志，因为否则将无法使用。

3.2.3版本的新增功能：-W添加了该标志。

该--add-suffix选项指定一个字符串以附加到所有输出文件名。 -n指定此标志时需要该标志，因为写入不同的文件名时不需要备份。例：

```
$ 2to3 -n -W --add-suffix=3 example.py
```

将导致example.py3写入一个已命名的转换文件。

3.2.3版--add-suffix中的新增功能*：*添加了该选项。

将整个项目从一个目录树转换为另一个目录树：

```
$ 2to3 --output-dir=python3-version/mycode -W -n python2-version/mycode
```

26.7.2。固定器

转换代码的每个步骤都封装在修复程序中。该命令列出它们。如上述记载，每一个都可以接通和关断个别。他们在这里更详细地描述。2to3 -l

apply

删除的使用apply()。例如转换为。apply(function, *args, **kwargs)function(*args, **kwargs)

asserts

unittest用正确的方法替换已弃用的方法名称。

从	至
failUnlessEqual(a, b)	assertEqual(a, b)
assertEquals(a, b)	assertEqual(a, b)
failIfEqual(a, b)	assertNotEqual(a, b)
assertNotEquals(a, b)	assertNotEqual(a, b)
failUnless(a)	assertTrue(a)
assert_(a)	assertTrue(a)
failIf(a)	assertFalse(a)
failUnlessRaises(exc, cal)	assertRaises(exc, cal)
failUnlessAlmostEqual(a, b)	assertAlmostEqual(a, b)
assertAlmostEquals(a, b)	assertAlmostEqual(a, b)
failIfAlmostEqual(a, b)	assertNotAlmostEqual(a, b)
assertNotAlmostEquals(a, b)	assertNotAlmostEqual(a, b)

basestring

转换basestring为str。

buffer

转换buffer为memoryview。这个修复程序是可选的，因为memoryviewAPI是类似的，但不完全相同buffer。

dict

修复字典迭代方法。 dict.iteritems()被转换为 dict.items() ，dict.iterkeys()对 dict.keys() ，和 dict.itervalues()到dict.values()。同样， dict.viewitems() ，dict.viewkeys()和dict.viewvalues()分别被转换到dict.items() ，dict.keys()和dict.values()。它还包装的现有用途dict.items() ， dict.keys()以及dict.values()在一个呼叫list。

except

转换为。except X, Texcept X as T

exec

将exec语句转换为exec()函数。

execfile
删除的使用execfile()。该参数execfile()被包裹在调用open() ，compile()和exec()。

exitfunc
更改sys.exitfunc使用atexit 模块的分配。

filter
filter()在list呼叫中包装使用情况。

funcattrs
修复已重命名的功能属性。例如，my_function.func_closure 转换为my_function.__closure__。

future
删除语句。from __future__ import new_feature

getcwdu
重命名os.getcwdu()为os.getcwd()。

has_key
变化dict.has_key(key)来。key in dict

idioms
这个可选的修补程序执行多个转换，使Python代码更具惯用性。键入比较类似并 转换为。成为。这个修复者也试图 在适当的地方使用。例如，这个块type(x) is SomeClass type(x) == SomeClass isinstance(x, SomeClass) while 1 while True sorted()

```
L = list(some_iterable)
L.sort()
```

改为

```
L = sorted(some_iterable)
```

import
检测同级进口并将它们转换为相对进口。

imports
处理模块在标准库中重命名。

imports2
处理其他模块在标准库中重命名。imports仅因技术限制而与定影剂分开。

input
转换input(prompt)为eval(input(prompt))。

intern
转换intern()为sys.intern()。

isinstance

修复了第二个参数中的重复类型isinstance()。例如，转换为并转换为 。 isinstance(x, (int, int))isinstance(x, int)isinstance(x, (int, float, int))isinstance(x, (int, float))

itertools_imports

删除的进口itertools.ifilter() ， itertools.izip()以及 itertools.imap() 。 进口itertools.ifilterfalse()也改为itertools.filterfalse()。

itertools

变化的使用itertools.ifilter() ，itertools.izip()以及 itertools.imap()它们内置的等价物。 itertools.ifilterfalse()改为itertools.filterfalse()。

long

重命名long为int。

map

包裹map()在list呼叫中。它也改变 来。使用禁用此修复程序。map(None, x)list(x)from future_builtins import map

metaclass

将旧的元类语法（在类体中）转换为new（ ）。 __metaclass__ = MetaclassX(metaclass=Meta)

methodattrs

修复旧的方法属性名称。例如，meth.im_func转换为meth.__func__。

ne

将旧的不等号语法转换<>为!=。

next

将迭代器next()方法的使用转换为 next()函数。它还将next()方法重命名为 __next__()。

nonzero

重命名__nonzero__()为__bool__()。

numliterals

将八进制文字转换为新的语法。

operator

将调用operator模块中的各种功能转换为其他但等效的功能调用。在需要时，import 添加适当的语句，例如。进行以下映射：import collections

从	至
operator.isCallable(obj)	hasattr(obj, '__call__')
operator.sequenceIncludes(obj)	operator.contains(obj)
operator.isSequenceType(obj)	isinstance(obj, collections.Sequence)

从	至
operator.isMappingType(obj)	isinstance(obj, collections.Mapping)
operator.isNumberType(obj)	isinstance(obj, numbers.Number)
operator.repeat(obj, n)	operator.mul(obj, n)
operator.irepeat(obj, n)	operator.imul(obj, n)

paren
: 在列表解析中需要的地方添加额外的括号。例如，变成。[x for x in 1, 2][x for x in (1, 2)]

print
: 将print语句转换为print()函数。

raise
: 转换为，并转换为。如果是一个元组，翻译将是不正确的，因为替换元组的例外在3.0中已经被删除。raise E, Vraise E(V)raise E, V, Traise E(V).with_traceback(T)E

raw_input
: 转换raw_input()为input()。

reduce
: 处理移动reduce()到functools.reduce()。

reload
: 转换reload()为imp.reload()。

renames
: 变化sys.maxint来sys.maxsize。

repr
: 用repr()函数替换反向代码。

set_literal
: set用设置的文字替换构造函数的使用。这个修复器是可选的。

standarderror
: 重命名StandardError为Exception。

sys_exc
: 更改过时sys.exc_value，sys.exc_type， sys.exc_traceback使用sys.exc_info()。

throw
: 修复生成器throw()方法中的API更改。

tuple_params
: 删除隐式元组参数解包。此修复程序插入临时变量。

types
: 修复了删除types 模块中某些成员的代码。

unicode
: 重命名unicode为str。

urllib
: 处理的重命名urllib ，并urllib2在urllib 包。

ws_comma
: 从逗号分隔的项目中删除多余的空格。这个修复器是可选的。

xrange
: 重命名xrange() ，以range()与现有的包装range() 用电话list。

xreadlines
: 变化来。 for x in file.xreadlines() for x in file

zip
: zip() 在list呼叫中包装使用情况。这在出现时被禁用 。 from future_builtins import zip

26.7.3。 lib2to3- 2to3的图书馆

源代码： Lib / lib2to3 /

> **注意：** 该lib2to3API应该被认为是不稳定的，并且可能在未来发生剧烈变化。

26.8。 test- 用于Python的回归测试包

注意： 该test软件包仅供Python内部使用。这是为Python的核心开发人员的利益记录的。任何在Python标准库之外使用这个包都是不鼓励的，因为这里提到的代码可以在Python的发行版之间改变或删除，恕不另行通知。

该 test 软件包包含所有Python的回归测试以及模块 test.support 和 test.regrtest 。 test.support用于在test.regrtest推动测试套件的同时增强您 的测试。

test软件包中名称以开头的每个模块test_都是特定模块或功能的测试套件。所有新的测试应该使用unittest或doctest模块编写。一些较旧的测试是使用“传统”测试样式来编写的，该样式比较输出的打印结果 sys.stdout; 这种测试风格被认为已被弃用。

也可以看看:

模 unittest
编写PyUnit回归测试。

模 doctest
嵌入在文档字符串中的测试。

26.8.1。编写单元测试test包

最好使用unittest模块的测试遵循一些准则。一种是通过启动测试模块来命名测试模块，并以被测模块test_的名称结束测试模块。测试模块中的测试方法应该以测试方法test_的描述开始并结束。这是必需的，以便测试驱动程序将这些方法识别为测试方法。此外，该方法不应包含任何文档字符串。应该使用评论（如）为测试方法提供文档。这样做是因为文档字符串如果存在，就会被打印出来，因此没有说明正在运行的测试。 # Tests function returns only True or False

通常使用基本的样板文件：

```
import unittest
from test import support

class MyTestCase1(unittest.TestCase):

    # Only use setUp() and tearDown() if necessary

    def setUp(self):
        ... code to execute in preparation for tests ...

    def tearDown(self):
        ... code to execute to clean up after tests ...

    def test_feature_one(self):
        # Test feature one.
        ... testing code ...
```

```
    def test_feature_two(self):
        # Test feature two.
        ... testing code ...

    ... more test methods ...

class MyTestCase2(unittest.TestCase):
    ... same structure as MyTestCase1 ...

... more test classes ...

if __name__ == '__main__':
    unittest.main()
```

此代码模式允许测试套件`test.regrtest`作为支持`unittest`CLI 的脚本或通过 CLI 自行运行。`python -m unittest`

回归测试的目标是试图破解代码。这导致了一些指导方针要遵循：

- 测试套件应该执行所有的类，函数和常量。这不仅包括要呈现给外部世界的外部API，还包括“私人”代码。
- Whitebox测试（在测试写入时检查正在测试的代码）是首选。Blackbox测试（仅测试已发布的用户界面）不够完整，无法确保所有边界和边缘案例都经过测试。
- 确保所有可能的值都经过测试，包括无效值。这确保不仅所有有效值都可以接受，而且不正确的值也可以正确处理。
- 尽可能多地排除代码路径。测试发生分支的位置，从而定制输入以确保通过代码获取许多不同的路径。
- 为测试代码发现的任何错误添加显式测试。这将确保如果将来代码被更改，错误不会再次出现。
- 确保在测试后清理（如关闭并删除所有临时文件）。
- 如果测试取决于操作系统的特定条件，则在尝试进行测试之前验证已存在的条件。
- 尽可能少地导入模块，尽快完成。这样可以最大限度地减少测试的外部依赖性，并最大限度地减少导入模块副作用时可能出现的异常行为。
- 尝试最大化代码重用。有时候，测试会随着使用什么类型的输入而变化。通过用指定输入的类继承基本测试类来最小化代码重复：

```
class TestFuncAcceptsSequencesMixin:

    func = mySuperWhammyFunction

    def test_func(self):
        self.func(self.arg)

class AcceptLists(TestFuncAcceptsSequencesMixin, unittest.TestCase):
    arg = [1, 2, 3]
```

```
class AcceptStrings(TestFuncAcceptsSequencesMixin, unittest.TestCase):
    arg = 'abc'

class AcceptTuples(TestFuncAcceptsSequencesMixin, unittest.TestCase):
    arg = (1, 2, 3)
```

使用这种模式时，请记住所有继承的类 unittest.TestCase都作为测试运行。上例中的Mixin类没有任何数据，因此不能自行运行，因此它不会从中继承unittest.TestCase。

也可以看看:

测试驱动开发
Kent Beck在编写代码之前编写的一本书。

26.8.2。使用命令行界面运行测试

该test软件包可以作为脚本运行，以驱动Python的回归测试套件，这要归功于以下-m选项：**python -m test**。在引擎盖下，它使用test.regrtest；通话**蟒蛇-m test.regrtest**在以前的Python版本使用仍然有效。自行运行脚本会自动开始运行test包中的所有回归测试。它通过查找名称以开头的包中的所有模块test_，导入它们并执行函数（ test_main()如果存在）或通过unittest.TestLoader.loadTestsFromModule（如果test_main不存在）加载测试来完成此操作。要执行的测试名称也可以传递给脚本。指定一个回归测试（**python -m test test_spam**）将使输出最小化，并仅打印测试是否通过。

test直接运行允许测试使用哪些资源可供设置。您可以通过使用-u命令行选项来完成此操作。指定all该-u选项的值将启用所有可能的资源：**python -m test -uall**。如果只需要一种资源（更常见的情况），则可能会列出不需要的以逗号分隔的资源列表 all。命令**python -m test -uall，-audio，-largefile** 将test使用除资源audio和 largefile资源以外的所有资源运行。有关所有资源和更多命令行选项的列表，请运行**python -m test -h**。

执行回归测试的其他一些方法取决于正在执行测试的平台。在Unix上，您可以在构建Python的顶级目录中运行**make test**。在Windows上，从您的目录执行**rt.bat**PCBuild将运行所有的回归测试。

26.9。test.support- Python测试套件的实用程序

该test.support模块为Python的回归测试套件提供支持。

注意： test.support不是公共模块。这里记录下来帮助Python开发人员编写测试。该模块的API可能会发生变化，而不会在发布之间产生向后兼容性问题。

该模块定义了以下例外情况：

*异常*test.support.TestFailed

测试失败时会引发异常。这不利于 unittest基于测试和unittest.TestCase断言的方法。

*异常*test.support.ResourceDenied
子类unittest.SkipTest。当资源（如网络连接）不可用时引发。由requires() 功能引发。

该test.support模块定义了以下常量：

test.support.verbose
True当详细输出被启用时。当需要关于运行测试的更多详细信息时应该检查。*详细*由...设置 test.regrtest。

test.support.is_jython
True 如果正在运行的解释器是Jython。

test.support.TESTFN
设置为可安全使用的名称作为临时文件的名称。任何创建的临时文件应该关闭并取消链接（删除）。

该test.support模块定义了以下功能：

test.support.forget（*module_name*）
删除名为*module_name*的模块，sys.modules并删除模块的所有字节编译文件。

test.support.is_resource_enabled（*资源*）
True如果*资源*已启用且可用，则返回。可用资源列表仅在test.regrtest执行测试时设置。

test.support.requires（*资源*，*msg =无*）
提高ResourceDenied如果*资源*不可用。*msg*是ResourceDenied如果它被引发的参数。始终返回 True如果调用由它的功能__name__是'__main__'。在执行测试时使用test.regrtest。

test.support.findfile（*文件名*，*子目录=无*）
将路径返回到名为*filename的文件*。如果找不到匹配，则返回*文件名*。这不等于失败，因为它可能是文件的路径。

设置*子目录*指示用于查找文件的相对路径，而不是直接查看路径目录。

test.support.run_unittest（**类*）
执行unittest.TestCase传递给该函数的子类。该函数扫描以前缀开头的方法的类test_并单独执行测试。

将字符串作为参数传递也是合法的;这些应该是关键 sys.modules。每个关联的模块将被扫描 unittest.TestLoader.loadTestsFromModule()。这通常在以下test_main()函数中看到：

```
def test_main():
    support.run_unittest(__name__)
```

这将运行在命名模块中定义的所有测试。

test.support.run_doctest（*模块*，*详细=无*）

doctest.testmod()在给定的*模块*上运行。返回 。(failure_count, test_count)

如果*详细程度*是None ，doctest.testmod()则将冗长设置为verbose。否则，它运行的详细程度设置为 None。

test.support.**check_warnings**(**filters*，*quiet = True*)

这是一个便捷的包装warnings.catch_warnings() ，可以更容易地测试警告是否正确引发。这大致等同于warnings.catch_warnings(record=True)使用 warnings.simplefilter()设置来调用always并使用选项来自动验证记录的结果。

check_warnings接受表单的2元组作为位置参数。如果提供了一个或多个*过滤器*，或者如果可选关键字参数*quiet*是，它会检查以确保警告符合预期：每个指定的过滤器必须至少匹配由所附代码引发的警告之一或者测试失败，并且如果发现任何与指定过滤器不匹配的警告，则测试失败。要禁用这些检查中的第一项，请将*安静*设置为。("message regexp", WarningCategory)FalseTrue

如果没有指定参数，则默认为：

```
check_warnings(("", Warning), quiet=True)
```

在这种情况下，所有警告都会被捕获，并且不会引发错误。

在进入上下文管理器时，WarningRecorder返回一个实例。潜在的警告列表来自catch_warnings()于记录器对象的 warnings属性。为了方便起见，还可以通过记录器对象直接访问表示最近警告的对象的属性（请参见下面的示例）。如果没有发出警告，则表示警告的对象的任何属性将返回None。

记录器对象也有一个reset()方法，清除警告列表。

上下文管理器被设计为像这样使用：

```
with check_warnings(("assertion is always true", SyntaxWarning),
                    ("", UserWarning)):
    exec('assert(False, "Hey!")')
    warnings.warn(UserWarning("Hide me!"))
```

在这种情况下，如果没有提出任何警告或者提出了其他警告，check_warnings()则会引发错误。

当测试需要更深入地看待警告时，而不是仅仅检查它们是否发生，可以使用类似这样的代码：

```
with check_warnings(quiet=True) as w:
    warnings.warn("foo")
    assert str(w.args[0]) == "foo"
    warnings.warn("bar")
    assert str(w.args[0]) == "bar"
    assert str(w.warnings[0].args[0]) == "foo"
    assert str(w.warnings[1].args[0]) == "bar"
    w.reset()
    assert len(w.warnings) == 0
```

这里所有警告都会被捕获，并且测试代码会直接测试捕获的警告。

*在版本3.2中更改：*新的可选参数*过滤器*和*安静*。

test.support.**captured_stdin**()

test.support.**captured_stdout**()

test.support.**captured_stderr**()

一个上下文管理器，用临时io.StringIO对象替换指定的流。

用于输出流的示例：

```
with captured_stdout() as stdout, captured_stderr() as stderr:
    print("hello")
    print("error", file=sys.stderr)
assert stdout.getvalue() == "hello\n"
assert stderr.getvalue() == "error\n"
```

使用输入流的示例：

```
with captured_stdin() as stdin:
    stdin.write('hello\n')
    stdin.seek(0)
    # call test code that consumes from sys.stdin
    captured = input()
self.assertEqual(captured, "hello")
```

test.support.**temp_dir**(*path = None*, *quiet = False*)

上下文管理器，用于在*路径*上创建临时目录并生成目录。

如果*路径*是None，则使用创建的临时目录 tempfile.mkdtemp()。如果*安静的*是False，上下文管理引发的错误的异常。否则，如果指定了*路径*并且无法创建，则只会发出警告。

test.support.**change_cwd**(*path*, *quiet = False*)

上下文管理器，将当前工作目录临时更改为*路径*并生成目录。

如果*安静的*是False，上下文管理引发的错误的异常。否则，它只发出警告并保持当前工作目录不变。

test.support.**temp_cwd**(*name ='tempcwd'*, *quiet = False*)

临时管理器，临时创建一个新目录并更改当前工作目录（CWD）。

上下文管理器在临时更改当前工作目录之前，在具有名称*名称*的当前目录中创建一个临时目录。如果*名称*是None，则使用创建的临时目录tempfile.mkdtemp()。

如果*安静*，False并且无法创建或更改CWD，则会引发错误。否则，只会发出警告并使用原始的CWD。

test.support.**temp_umask**(*umask*)

临时设置进程umask的上下文管理器。

test.support.**can_symlink**()

True如果操作系统支持符号链接，False 则返回，否则返回。

@test.support.**skip_unless_symlink**

运行需要支持符号链接的测试的装饰器。

@test.support.**anticipate_failure**(*条件*)

装饰者有条件地标记测试 unittest.expectedFailure()。任何对这个装饰器的使用应该有一个相关的注释来标识相关的跟踪器问题。

@test.support.**run_with_locale**(*catstr*, **locales*)

用于在不同语言环境中运行函数的装饰器，并在完成后正确重置该装饰器。 *catstr*是语言环境类别，例如字符串（例如"LC_ALL"）。传递的*语言环境*将按顺序尝试，并使用第一个有效的语言环境。

test.support.**make_bad_fd**()

通过打开和关闭一个临时文件并返回它的描述符来创建一个无效的文件描述符。

test.support.**import_module**(*name*, *deprecated = False*)

该函数导入并返回指定的模块。与正常导入不同，unittest.SkipTest如果模块无法导入，此功能将会提升。

如果模块和封装弃用消息这个导入期间抑制*弃用*是True。

版本3.1中的新功能。

test.support.**import_fresh_module**(*name*, *fresh = ( )*, *blocked = ( )*, *deprecated = False*)

此函数通过sys.modules在导入之前删除命名模块来导入并返回指定Python模块的新副本。请注意，与此不同reload()，原始模块不受此操作的影响。

*fresh*是sys.modules在进行导入之前从缓存中删除的附加模块名称的迭代。

*被阻止的*是None 在导入期间在模块高速缓存中替换的模块名称的迭代，以确保尝试导入它们ImportError。

在开始导入之前保存已命名的模块以及在*新鲜*和已*阻止*参数中命名的所有模块，并sys.modules在新导入完成时重新插入 。

如果模块和封装弃用消息这个导入期间抑制*弃用*是True。

ImportError如果无法导入已命名的模块，则会引发此功能。

使用示例：

```
# Get copies of the warnings module for testing without affecting the
# version being used by the rest of the test suite. One copy uses the
# C implementation, the other is forced to use the pure Python fallback
# implementation
```

```
py_warnings = import_fresh_module('warnings', blocked=['_warnings'])
c_warnings = import_fresh_module('warnings', fresh=['_warnings'])
```

版本3.1中的新功能。

test.support.**bind_port**(*袜子*，*主机= HOST*)

将套接字绑定到空闲端口并返回端口号。依靠临时端口来确保我们使用未绑定的端口。这很重要，因为许多测试可能同时运行，特别是在buildbot环境中。如果sock.familyis AF_INET和sock.typeis SOCK_STREAM和套接字有 SO_REUSEADDR或SO_REUSEPORT设置了此方法，则此方法会引发异常 。测试不应该为TCP / IP套接字设置这些套接字选项。设置这些选项的唯一情况是通过多个UDP套接字测试多播。

此外，如果SO_EXCLUSIVEADDRUSE套接字选项可用（即在Windows上），它将在套接字上设置。这将防止其他人在测试期间绑定到我们的主机/端口。

test.support.**find_unused_port**(*family = socket.AF_INET*，*socktype = socket.SOCK_STREAM*)

返回应该适合绑定的未使用端口。这是通过创建一个临时套接字来实现的，该套接字具有与sock参数相同的系列和类型（默认为AF_INET ， SOCK_STREAM ），并将其0.0.0.0与端口设置为0 的指定主机地址（默认为）绑定，从OS中获取未使用的临时端口。然后关闭并删除临时套接字，并返回临时端口。

这种方法或者bind_port()应该用于任何在测试期间需要将服务器套接字绑定到特定端口的测试。使用哪一个取决于调用代码是创建一个python套接字，还是一个未使用的端口需要在构造器中提供或传递给外部程序（即-acceptopenssl的s_server模式的参数）。总是喜欢bind_port()在 find_unused_port()这里成为可能。不鼓励使用硬编码端口，因为它可能会使多个测试实例无法同时运行，这对buildbots来说是一个问题。

test.support.**load_package_tests**(*pkg_dir*，*loader*，*standard_tests*，*pattern*)

用于测试包的协议的通用实现。 *pkg_dir*是包的根目录; *loader*，*standard_tests*和*pattern*是期望的参数 。在简单情况下，测试包 可以是以下内容 ：unittest load_testsload_tests__init__.py

```
import os
from test.support import load_package_tests

def load_tests(*args):
    return load_package_tests(os.path.dirname(__file__), *args)
```

test.support.**detect_api_mismatch**(*ref_api*，*other_api*，*，*ignore = ( )*)

返回在*other_api*上*找不到*的*ref_api*的属性，函数或方法的*集合*，除了在*ignore*中指定的此检查中要忽略的项目的已定义列表之外。

默认情况下，这将跳过以'_'开头的私有属性，但包含所有魔术方法，即以'__'开头和结尾的魔术方法。

3.5版本中的新功能。

test.support.check__all__（*test_case*，*module*，*name_of_module* = *None*，*extra* = *（ ）*，*blacklist* = *（ ）*）

断言*模块*的__all__变量包含所有公共名称。

模块的公共名称（其API）会根据它们是否与公共名称约定匹配并在*模块*中定义而自动检测。

所述*name_of_module*参数可以指定（为一个字符串或它们的元组）的API可以在顺序被定义为被检测为一个公共API什么模块（一个或多个）。其中一种情况是，*模块*从其他模块（可能是C后端（像csv它和它的_csv））中导入其公共API的一部分。

该*额外的*参数可以是一组本来不会自动为“公共”检测到的姓名，喜欢的对象没有正确的__module__ 属性。如果提供，它将被添加到自动检测到的。

该*黑名单*参数可以是一组不得被视为公共API，即使他们的名字表明否则的一部分名字。

使用示例：

```
import bar
import foo
import unittest
from test import support

class MiscTestCase(unittest.TestCase):
    def test__all__(self):
        support.check__all__(self, foo)

class OtherTestCase(unittest.TestCase):
    def test__all__(self):
        extra = {'BAR_CONST', 'FOO_CONST'}
        blacklist = {'baz'}  # Undocumented name.
        # bar imports part of its API from _bar.
        support.check__all__(self, bar, ('bar', '_bar'),
                             extra=extra, blacklist=blacklist)
```

3.6版本中的新功能。

该test.support模块定义了以下类：

class test.support.TransientResource（*exc*，*** kwargs*）

实例是在引发ResourceDenied指定的异常类型时引发的上下文管理器。任何关键字参数都被视为属性/值对，以便与with语句中引发的任何异常进行比较 。只有在所有对匹配都正确的情况下才会ResourceDenied引发异常。

*类*test.support.EnvironmentVarGuard

用于临时设置或取消设置环境变量的类。实例可以用作上下文管理器并具有用于查询/修改底层的完整字典界面os.environ。从上下文管理器退出后，通过此实例完成的对环境变量的所有更改都将回滚。

*版本3.1中更改：*添加字典界面。

EnvironmentVarGuard.set（*envvar*，*value*）

暂时将环境变量设置envvar为的值 value。

EnvironmentVarGuard.unset（*envvar*）

暂时取消设置环境变量envvar。

*类*test.support.SuppressCrashReport

一个上下文管理器用于试图防止崩溃对话框弹出的测试，预计会导致子进程崩溃。

在Windows上，它使用SetErrorMode禁用Windows错误报告对话框 。

在UNIX上，resource.setrlimit()用于将resource.RLIMIT_CORE软限制设置 为0以防止创建coredump文件。

在两个平台上，旧值都会被恢复__exit__()。

*类*test.support.WarningsRecorder

用于记录单元测试警告的类。有关check_warnings()更多详细信息，请参阅上述文档

*类*test.support.FakePath（*路径*）

简单的路径类对象。它实现了__fspath__() 刚刚返回*路径*参数的方法。如果*路径*是一个例外，它将被提升__fspath__()。

27.调试和分析

这些库可以帮助您进行Python开发：调试器使您能够逐步完成代码，分析堆栈帧并设置断点等，并且分析器运行代码并为您提供执行时间的详细分类，使您能够识别程序中的瓶颈。

27.1。bdb- 调试器框架

源代码： Lib / bdb.py

该bdb模块处理基本的调试器功能，如设置断点或通过调试器管理执行。

以下例外被定义：

*异常*bdb.BdbQuit
Bdb班级提出的用于退出调试器的异常。

该bdb模块还定义了两个类：

class bdb.Breakpoint（*self*，*file*，*line*，*temporary = 0*，*cond = None*，*funcname = None*）
该类实现临时断点，忽略计数，禁用和（重新）启用以及条件。

断点由数通过一个名为列表索引bpbynumber 和通过对。前者指向一个类的单个实例。后者指向这种情况的列表，因为每行可能有多个断点。(file, line)bplistBreakpoint

创建断点时，其关联的文件名应为规范形式。如果定义了一个*funcname*，当该函数的第一行被执行时，将会计算一个断点命中。一个条件断点总是计数一个命中。

Breakpoint 实例具有以下方法：

deleteMe（）
从与文件/行关联的列表中删除断点。如果它是该位置的最后一个断点，它也会删除文件/行的条目。

enable（）
将断点标记为已启用。

disable（）
将断点标记为禁用。

bpformat（）
返回一个字符串，其中包含有关该断点的所有信息，并且格式良好：

- 断点编号。
- 如果它是暂时的或不是。
- 它的文件，行位置。
- 导致休息的条件。
- 如果它必须被忽略N次。
- 断点击数。

3.2版本中的新功能

bpprint（*out = None*）

bpformat()将文件的输出打印*出来*，如果是 None ，则输出到标准输出。

class bdb. Bdb (*skip = None*)

的Bdb类作为一个通用的Python调试基类。

该课程负责跟踪设施的细节; 派生类应该实现用户交互。标准调试器类（ pdb.Pdb ）是一个例子。

在*跳跃*的说法，如果有，一定要全域式的模块名称模式的迭代。调试器不会进入与匹配其中一种模式的模块相关的帧。框架是否被认为起源于某个模块是由__name__框架全局决定的。

*新的3.1版：*在*跳跃*的说法。

Bdb通常不需要重写以下方法。

canonic (*文件名*)

用于获取规范形式的文件名的辅助方法，即以规范化（不区分大小写的文件系统）绝对路径为例，删除周围的尖括号。

reset ( )

设置botframe ， stopframe ， returnframe并 quitting用准备开始调试属性值。

trace_dispatch (*frame* , *event* , *arg*)

该功能作为调试帧的跟踪功能进行安装。它的返回值是新的跟踪函数（在大多数情况下，也就是它本身）。

默认实现决定如何分派帧，具体取决于即将执行的事件类型（作为字符串传递）。 *事件*可以是以下之一：

- "line"：一行新的代码将被执行。
- "call"：函数即将被调用，或输入另一个代码块。
- "return"：函数或其他代码块即将返回。
- "exception"：发生异常。
- "c_call"：AC功能即将被调用。
- "c_return"：AC功能已返回。
- "c_exception"：AC函数引发了一个异常。

对于Python事件，调用专用函数（见下文）。对于C事件，不采取任何行动。

该*ARG*参数取决于前面的事件。

有关sys.settrace()跟踪功能的更多信息，请参阅文档。有关代码和框架对象的更多信息，请参阅 标准类型层次结构。

dispatch_line (*框架*)

如果调试器应停在当前行上，则调用该 user_line() 方法（应在子类中重写）。 BdbQuit如果Bdb.quitting标志已设置（可以从中设置user_line() ），引发异常。 trace_dispatch()在该范围内返回对进一步跟踪的方法的引用 。

dispatch_call (*frame* , *arg*)

如果调试器应停止此函数调用，请调用 user_call() 方法（应在子类中重写）。BdbQuit如果Bdb.quitting标志已设置（可以从中设置user_call()），引发异常。trace_dispatch()在该范围内返回对进一步跟踪的方法的引用。

dispatch_return(*frame*, *arg*)

如果调试器应停止此函数返回，请调用该 user_return() 方法（应在子类中重写）。BdbQuit如果Bdb.quitting标志已设置（可以从中设置user_return()），引发异常。trace_dispatch()在该范围内返回对进一步跟踪的方法的引用。

dispatch_exception(*frame*, *arg*)

如果调试器应该停止在这个异常处，调用 user_exception() 方法（应该在子类中重写）。BdbQuit如果Bdb.quitting标志已设置（可以从中设置user_exception()），引发异常。trace_dispatch()在该范围内返回对进一步跟踪的方法的引用。

通常派生类不会覆盖以下方法，但如果他们想重新定义停止点和断点的定义，它们可能会出现。

stop_here(*框架*)

该方法检查*框架*是否botframe在调用堆栈的下方。 botframe是开始调试的框架。

break_here(*框架*)

此方法检查文件名中是否存在断点以及属于*帧的*行，或至少在当前函数中是否存在断点。如果断点是临时断点，则此方法将其删除。

break_anywhere(*框架*)

此方法检查当前帧的文件名中是否存在断点。

派生类应该重写这些方法来获得对调试器操作的控制权。

user_call(*frame*, *argument_list*)

从dispatch_call()被调用函数内部任何地方有可能需要中断时调用此方法。

user_line(*框架*)

这种方法从dispatch_line()任何一个stop_here()或break_here()收益时 被调用True。

user_return(*frame*, *return_value*)

这个方法dispatch_return()在stop_here() yield 时被调用True。

user_exception(*frame*, *exc_info*)

这个方法dispatch_exception()在stop_here()yield 时 被调用True。

do_clear(*arg*)

处理临时断点时如何删除断点。

该方法必须由派生类实现。

派生类和客户端可以调用以下方法来影响步进状态。

set_step ( )
在一行代码后停止。

set_next (*框架*)
在给定帧中或下方的下一行停止。

set_return (*框架*)
从给定帧返回时停止。

set_until (*框架*)
当到达当前行的行不超过或从当前帧返回时停止。

set_trace ([*frame*])
从*框架*开始调试。如果未指定*帧*，则调用从调用者的帧开始。

set_continue ( )
只在断点或完成时停止。如果没有断点，请将系统跟踪功能设置为None。

set_quit ( )
将该quitting属性设置为True。这会BdbQuit在下一次调用其中一种dispatch_*()方法时引发。

派生类和客户端可以调用以下方法来操作断点。如果出现问题，或者None一切正常，这些方法将返回一个包含错误消息的字符串。

set_break (*filename* , *lineno* , *temporary = 0* , *cond* , *funcname*)
设置一个新的断点。如果*lineno*行不存在作为参数传递的 *文件名*，则返回错误消息。的*文件名* 应为规范形式，如在所描述的canonic()方法。

clear_break (*文件名* , *lineno*)
删除*文件名*和*lineno中*的断点。如果没有设置，则返回错误消息。

clear_bpbynumber (*arg*)
删除具有索引断点*精氨酸*的 Breakpoint.bpbynumber。如果*arg*不是数字或超出范围，则返回错误消息。

clear_all_file_breaks (*文件名*)
删除*文件名中的*所有断点。如果没有设置，则返回错误消息。

clear_all_breaks ( )
删除所有现有的断点。

get_bpbynumber (*arg*)
返回给定数字指定的断点。如果*arg*是一个字符串，它将被转换为一个数字。如果*arg*是非数字字符串，如果给定断点从不存在或已被删除， ValueError则会引发a。

3.2版本中的新功能

get_break (*文件名* , *lineno*)

检查是否有一个断点*LINENO*的*文件名*。

get_breaks (*文件名* , *lineno*)
返回所有断点*LINENO*的*文件名*，或一个空列表，如果没有设置。

get_file_breaks (*文件名*)
返回*文件名中的*所有断点，如果没有设置，则返回空列表。

get_all_breaks ( )
返回设置的所有断点。

派生类和客户端可以调用以下方法来获取表示堆栈跟踪的数据结构。

get_stack (*f* , *t*)
获取一个帧的记录列表，以及所有更高（呼叫）和更低帧的记录以及更高部分的大小。

format_stack_entry (*frame_lineno* , *lprefix* =': ')
返回一个字符串，其中包含由元组标识的关于堆栈条目的信息 ：(frame, lineno)

- 包含该帧的文件名的规范形式。
- 函数名称或"<lambda>"。
- 输入参数。
- 返回值。
- 代码行（如果存在）。

客户端可以调用以下两种方法来使用调试器来调试以字符串形式给出的语句。

run (*cmd* , *globals* = *None* , *locals* = *None*)
调试通过exec()函数执行的语句。 *全局变量* 默认为__main__.__dict__ ，*局部变量*默认为*全局变量*。

runeval (*expr* , *globals* = *None* , *locals* = *None*)
调试通过eval()函数执行的表达式。 *全局*和 *当地人*具有相同的含义在run()。

runctx (*cmd* , *globals* , *locals*)
为了向后兼容。调用run()方法。

runcall (*func* , * *args* , ** *kwds*)
调试一个函数调用，并返回其结果。

最后，模块定义了以下功能：

bdb. checkfuncname (*b* , *frame*)
检查我们是否应该在这里休息，这取决于断点*b的*设置方式。

如果它是通过行号设置的，它将检查是否b.line与也作为参数传递的帧中的一样。如果断点是通过函数名设置的，那么我们必须检查我们是否在正确的框架中（正确的功能），以及我们是否在第一个可执行行中。

bdb.effective(*文件*, *行*, *框架*)

确定在这行代码中是否存在有效（有效）断点。返回断点的元组以及指示是否可以删除临时断点的布尔值。如果没有匹配的断点，则返回。(None, None)

bdb.set_trace()

Bdb从调用者的框架开始调试实例。

27.2。faulthandler- 转储Python回溯

3.3版本的新功能

该模块包含显式转储Python回溯函数，出现故障，超时后或用户信号的函数。调用faulthandler.enable()安装故障处理程序为SIGSEGV，SIGFPE， SIGABRT，SIGBUS，和SIGILL信号。您也可以在启动时通过设置启用它们PYTHONFAULTHANDLER环境变量或通过使用命令行选项。-X faulthandler

错误处理程序与系统错误处理程序（如Apport或Windows错误处理程序）兼容。如果该sigaltstack()功能可用，该模块为信号处理程序使用替代堆栈。这允许它甚至在堆栈溢出时转储回溯。

故障处理程序在灾难性情况下调用，因此只能使用信号安全功能（例如，它无法在堆上分配内存）。由于这个限制，回溯转储与普通的Python回溯相比是最小的：

- 仅支持ASCII。的backslashreplace错误处理程序是用来对编码。
- 每个字符串限制为500个字符。
- 仅显示文件名，功能名称和行号。（没有源代码）
- 它限制为100帧和100个线程。
- 顺序相反：最近的呼叫显示在第一位。

默认情况下，Python追溯被写入sys.stderr。要查看回溯，应用程序必须在终端中运行。日志文件也可以传递给faulthandler.enable()。

该模块以C语言实现，因此追溯可以在崩溃时或Python死锁时转储。

27.2.1。倾倒追踪

faulthandler.**dump_traceback**(*file = sys.stderr*, *all_threads = True*)

将所有线程的追溯转储到*文件中*。如果*all_threads*是 False，只转储当前线程。

*在版本3.5中进行了更改：*增加了对将文件描述符传递给此函数的支持。

27.2.2。故障处理程序状态

faulthandler.**enable**(*file = sys.stderr*, *all_threads = True*)

启用故障处理程序：安装的处理程序SIGSEGV， SIGFPE，SIGABRT，SIGBUS和SIGILL 信号倾倒Python的回溯。如果*all_threads*是True，生产每一个正在运行的线程回溯。否则，只转储当前线程。

该*文件*必须保持打开状态，直到故障处理程序被禁用：请参阅 文件描述符的问题。

*在版本3.5中进行了更改：*增加了对将文件描述符传递给此函数的支持。

*在版本3.6中更改：*在Windows上，还安装了Windows异常处理程序。

faulthandler.disable ( )

禁用故障处理程序：卸载安装的信号处理程序 enable()。

faulthandler.is_enabled ( )

检查错误处理程序是否已启用。

27.2.3。在超时后转储追踪

faulthandler.dump_traceback_later (*timeout*，*repeat = False*，*file = sys.stderr*，*exit = False*)

转储全部线程的回溯，一个超时后*超时*秒，或每*超时*秒，如果*重复*的True。如果*退出*是True，调用 _exit()与状态= 1倾销回溯之后。（注意 _exit()立即退出进程，这意味着它不会执行任何清理，如刷新文件缓冲区。）如果该函数被调用两次，则新调用将替换先前的参数并重置超时。计时器具有亚秒级分辨率。

该*文件*必须保持打开状态，直到回溯被转储或被 cancel_dump_traceback_later()调用：请参阅文件描述符的问题。

该函数是使用看门狗线程实现的，因此如果Python在禁用线程的情况下编译则不可用。

*在版本3.5中进行了更改：*增加了对将文件描述符传递给此函数的支持。

faulthandler.cancel_dump_traceback_later ( )

取消最后一次呼叫dump_traceback_later()。

27.2.4。转储用户信号的追踪

faulthandler.register (*signum*，*file = sys.stderr*，*all_threads = True*，*chain = False*)

注册用户信号：为*signum*信号安装处理程序，将所有线程或当前线程（如果*all_threads*）的回溯转储False到*文件中*。如果链是，请调用前面的处理程序True。

该*文件*，直到该信号被注销必须保持开放的 unregister()看到：问题与文件描述符。

在Windows上不可用。

*在版本3.5中进行了更改：*增加了对将文件描述符传递给此函数的支持。

faulthandler.unregister (*signum*)

取消注册用户信号：卸载安装的*Signum*信号的处理程序register()。 True如果信号被注册，False则返回， 否则返回。

在Windows上不可用。

27.2.5。问题与文件描述符

enable() ，dump_traceback_later()并register()保持他们的文件描述符*文件*的说法。如果文件已关闭且其文件描述符被新文件重新使用，或者如果os.dup2()用于替换文件描述符，则追溯将被写入不同的文件。每次更换文件时再次调用这些函数。

27.2.6。示例

在启用或不启用错误处理程序的情况下，Linux上的段错误示例：

```
$ python3 -c "import ctypes; ctypes.string_at(0)"
Segmentation fault

$ python3 -q -X faulthandler
>>> import ctypes
>>> ctypes.string_at(0)
Fatal Python error: Segmentation fault

Current thread 0x00007fb899f39700 (most recent call first):
  File "/home/python/cpython/Lib/ctypes/__init__.py", line 486 in string_at
  File "<stdin>", line 1 in <module>
Segmentation fault
```

27.3。 pdb- Python调试器

源代码： Lib / pdb.py

该模块pdb为Python程序定义了一个交互式源代码调试器。它支持在源代码级设置（条件）断点和单步执行，检查堆栈帧，源代码列表，以及在任何栈帧的上下文中评估任意Python代码。它也支持验尸调试，可以在程序控制下调用。

调试器是可扩展的 - 它实际上被定义为类Pdb。这是目前没有记录，但通过阅读来源容易理解。扩展接口使用模块bdb和cmd。

调试器的提示符是(Pdb)。在调试器的控制下运行程序的典型用法是：

```
>>> import pdb
>>> import mymodule
>>> pdb.run('mymodule.test()')
> <string>(0)?()
(Pdb) continue
> <string>(1)?()
(Pdb) continue
NameError: 'spam'
> <string>(1)?()
(Pdb)
```

*在版本3.3中更改：*通过readline模块的制表符完成可用于命令和命令参数，例如当前全局名称和本地名称是作为p命令的参数提供的。

pdb.py也可以作为调试脚本来调用其他脚本。例如：

```
python3 -m pdb myscript.py
```

当作为脚本调用时，如果正在调试的程序异常退出，pdb将自动进入事后调试。在验尸调试后（或程序正常退出后），pdb将重新启动程序。自动重新启动会保留pdb的状态（如断点），并且在大多数情况下，比程序退出时退出调试器更有用。

*3.2版新增功能：*pdb.py现在接受-c执行命令的选项，如同在.pdbrc文件中给出的一样，请参阅调试器命令。

从正在运行的程序中插入调试器的典型用法是插入

```
import pdb; pdb.set_trace()
```

在你想要进入调试器的位置。然后，您可以按照此语句遍历代码，并在没有调试器的情况下使用该continue命令继续运行。

检查崩溃程序的典型用法是：

```
>>> import pdb
>>> import mymodule
```

```
>>> mymodule.test()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "./mymodule.py", line 4, in test
    test2()
  File "./mymodule.py", line 3, in test2
    print(spam)
NameError: spam
>>> pdb.pm()
> ./mymodule.py(3)test2()
-> print(spam)
(Pdb)
```

该模块定义了以下功能; 每个都以一种稍微不同的方式进入调试器：

pdb.run(*语句*, *globals* = *None*, *locals* = *None*)

在调试器控制下执行*语句*（以字符串或代码对象的形式给出）。调试器提示出现在任何代码执行之前; 你可以设置断点和类型continue，或者你可以使用step或者next（所有这些命令在下面解释）通过语句。可选的*全局变量*和*局部*变量指定代码执行的环境; 默认情况下__main__使用模块的字典。（请参阅内置 exec()或eval()功能的说明。）

pdb.runeval(*表达式*, *globals* = *None*, *locals* = *None*)

在调试器控制下评估*表达式*（以字符串或代码对象的*形式*给出）。当runeval()返回时，它返回表达式的值。否则这个功能类似于run()。

pdb.runcall(*函数*, **args*, ***kwds*)

用给定的参数调用*函数*（函数或方法对象，而不是字符串）。当runcall()返回时，它返回无论函数调用返回。只要输入该函数，调试器提示符就会出现。

pdb.set_trace()

在调用堆栈帧输入调试器。即使代码没有被调试（例如断言失败），这对于在程序中的给定点处对断点进行硬编码也很有用。

pdb.post_mortem(*traceback* = *None*)

输入给定*追踪*对象的验尸调试。如果没有 给出*回溯*，它将使用当前正在处理的异常（如果要使用默认值，则必须处理异常）。

pdb.pm()

输入发现的回溯的验尸调试 sys.last_traceback。

这些run*函数set_trace()是用于实例化 Pdb类并调用相同名称的方法的别名。如果你想访问更多的功能，你必须自己去做：

class pdb.Pdb(*completekey='tab'*, *stdin* = *None*, *stdout* = *None*, *skip* = *None*, *nosigint* = *False*, *readrc* = *True*)

Pdb 是调试器类。

该*completekey*，*标准输入*和*标准输出*参数被传递到底层cmd.Cmd阶级; 看到那里的描述。

在*跳跃*的说法，如果有，一定要全域式的模块名称模式的迭代。调试器不会进入与匹配其中一种模式的模块相关的帧。[1]

默认情况下，Ctrl-C当您发出continue命令时，Pdb为SIGINT信号设置一个处理程序（当用户在控制台上按下时发送）。这允许您通过按下再次进入调试器Ctrl-C。如果您希望Pdb不要触及SIGINT处理程序，请将*nosigint*设置为true。

该*readrc*参数默认为真，控制是否将PDB从文件系统加载.pdbrc文件。

使用*跳过*启用跟踪的示例调用：

```
import pdb; pdb.Pdb(skip=['django.*']).set_trace()
```

*新的3.1版：*在*跳跃*的说法。

*新版本3.2：*在*nosigint*参数。以前，一个SIGINT处理程序从未由Pdb设置过。

*改变在3.6版本：*该*readrc*参数。

run（*语句*，*globals* = *None*，*locals* = *None*）
runeval（*表达式*，*globals* = *None*，*locals* = *None*）
runcall（*函数*，* *args*，** *kwds*）
set_trace（）
请参阅上述功能的文档。

27.3.1。调试器命令

下面列出了调试器识别的命令。大多数命令可以缩写为一个或两个字母，如图所示; 例如h(elp)意味着可以h或help可以用来输入帮助命令（但不是he 或者hel，H或者Help或HELP）。命令的参数必须用空格（空格或制表符）分隔。可选参数[]在命令语法中用方括号（）括起来; 方括号不能输入。命令语法中的替代方法由垂直条（|）分隔。

输入一个空白行重复输入的最后一个命令。例外：如果最后一条命令是一条list命令，则会列出接下来的11条线。

调试器无法识别的命令被假定为Python语句，并在被调试程序的上下文中执行。Python语句也可以带有感叹号（！）前缀。这是检查被调试程序的强大方法; 甚至可以改变一个变量或调用一个函数。当在这样的语句中发生异常时，会打印异常名称，但调试器的状态不会更改。

调试器支持别名。别名可以有一个参数，使其对被检查的上下文具有一定的适应性。

多个命令可以在一行中输入，由一个单独的行分隔;;。（;不使用单个函数，因为它是传递给Python解析器的行中多个命令的分隔符。）没有智能应用于分隔命令; 输入在第一;;对被分割，即使它位于引用字符串的中间。

如果文件.pdbrc存在于用户的主目录或当前目录中，则它将被读入并执行，就好像它已在调试器提示符下键入一样。这对别名特别有用。如果两个文件都存在，则主目录中的文件首先被读取，并且在那里定义的别名可以被本地文件覆盖。

在版本3.2中更改：.pdbrc现在可以包含继续调试的命令，如 continue或next。以前，这些命令不起作用。

h(elp) [command]
: 如果没有参数，请打印可用命令的列表。用*命令*作为参数，打印关于该命令的帮助。 显示完整的文档（模块的文档字符串）。由于*命令* 参数必须是标识符，因此必须输入以获取有关该命令的帮助。help pdbpdbhelp exec！

w(here)
: 打印堆栈跟踪，最近一帧在底部。箭头表示当前帧，它决定了大多数命令的上下文。

d(own) [count]
: 移动当前帧*计数*（默认的）水平在堆栈跟踪下（到较新的帧）。

u(p) [count]
: 将当前帧*计数*（默认的）水平，直至在堆栈跟踪（到旧帧）。

b(reak) [([filename:]lineno | function) [, condition]]
: 使用*lineno*参数，在当前文件中设置一个中断。使用 *函数*参数，在该函数中的第一个可执行语句处设置一个中断。行号可以用文件名和冒号作为前缀，以指定另一个文件中的断点（可能是尚未加载的文件）。该文件被搜索sys.path。请注意，每个断点都分配了一个其他所有断点命令引用的数字。

  如果存在第二个参数，那么它是一个表达式，它必须在断点被赋予之前评估为true。

  如果没有参数，请列出所有中断，包括每个断点，断点已被命中的次数，当前忽略计数以及相关条件（如果有）。

tbreak [([filename:]lineno | function) [, condition]]
: 临时断点，在首次命中时会自动删除。参数与之相同break。

cl(ear) [filename:lineno | bpnumber [bpnumber ...]]
: 使用*文件名：lineno*参数，清除此行的所有断点。用空格分隔的断点数列表清除这些断点。没有参数，清除所有的中断（但首先要求确认）。

disable [bpnumber [bpnumber ...]]
: 禁用以空格分隔的断点编号列表给出的断点。禁用断点意味着它不会导致程序停止执行，但与清除断点不同，它将保留在断点列表中并可以（重新）启用。

enable [bpnumber [bpnumber ...]]
: 启用指定的断点。

ignore bpnumber [count]
: 设置给定断点编号的忽略计数。如果省略计数，忽略计数设置为0.当忽略计数为零时，断点变为活动状态。非零时，每次到达断点时都会减少计数，并且不会禁用断点，并且任何关联的条件计算结果为true。

condition bpnumber [condition]

为断点设置一个新*条件*，该断点在判断断点前必须评估为true。如果*条件*不存在，则删除任何现有条件; 即断点是无条件的。

commands [bpnumber]
指定断点编号*bpnumber*的命令列表。命令本身出现在以下几行。输入一行只是 end为了终止命令。一个例子：

```
(Pdb) commands 1
(com) p some_variable
(com) end
(Pdb)
```

要从断点删除所有命令，请键入命令并立即使用end; 也就是说，不给命令。

没有*bpnumber*参数，命令引用最后一个断点集。

您可以使用断点命令再次启动程序。只需使用continue命令或步骤或恢复执行的任何其他命令即可。

指定恢复执行的任何命令（当前的继续，步骤，下一个，返回，跳转，退出和它们的缩写）终止命令列表（就像该命令紧接着结束）。这是因为，无论何时恢复执行（即使是简单的下一步或步骤），都可能遇到另一个断点 - 它们可能有自己的命令列表，从而导致关于执行哪个列表的歧义。

如果在命令列表中使用'silent'命令，则不会打印有关在断点处停止的常用消息。这对于打印特定消息然后继续的断点可能是理想的。如果其他命令都不打印任何内容，则不会看到已达到断点的迹象。

s(tep)
执行当前行，在第一个可能的场合停止（无论是在被调用的函数中，还是在当前函数的下一行中）。

n(ext)
继续执行，直到达到当前函数中的下一行或返回。（之间的差next和step是step一个被调用的函数内停止，而next 执行在（几乎）全速，仅在当前函数的下一行停止调用的函数。）

unt(il) [lineno]
如果没有参数，则继续执行，直到达到数字大于当前数字的行。

使用行号，继续执行，直到达到数字大于或等于数字的行。在这两种情况下，当前帧返回时也停止。

*在版本3.2中更改：*允许提供明确的行号。

r(eturn)
继续执行，直到当前函数返回。

c(ont(inue))
继续执行，仅在遇到断点时停止。

j(ump) lineno

设置将要执行的下一行。仅在最底部的框架中可用。这可以让您跳回来并再次执行代码，或者跳转到跳过您不想运行的代码。

应该注意的是，并不是所有的跳转都是允许的 - 例如跳到`for`循环中或跳出 `finally`子句是不可能的。

l(ist) [first[, last]]

列出当前文件的源代码。如果没有参数，请在当前行的周围列出11行或继续之前的列表。使用. as参数，列出当前行周围的11行。用一个参数，在该行列出11行。有两个参数，列出给定的范围; 如果第二个参数小于第一个，则它被解释为一个计数。

当前帧中的当前行通过->。如果正在调试异常，那么异常最初引发或传播的行将由>>与之相反的行指明。

*新版本3.2：*该>>标记。

ll | longlist

列出当前函数或帧的所有源代码。有趣的线条被标记为`list`。

3.2版本中的新功能

a(rgs)

打印当前函数的参数列表。

p expression

评估当前上下文中的*表达式*并打印其值。

> **注意：** `print()`也可以使用，但不是调试器命令 - 它执行Python `print()`函数。

pp expression

和`p`命令一样，除了表达式的值是用`pprint`模块打印的。

whatis expression

打印*表达式*的类型。

source expression

尝试获取给定对象的源代码并显示它。

3.2版本中的新功能

display [expression]

如果表达式的值发生变化，则每次在当前帧中停止执行时显示该值。

如果没有表达式，请列出当前帧的所有显示表达式。

3.2版本中的新功能

undisplay [expression]

不要在当前帧中再显示表达式。如果没有表达式，请清除当前帧的所有显示表达式。

3.2版本中的新功能

interact
: 启动一个交互式解释器（使用code模块），其全局名称空间包含在当前作用域中找到的所有（全局和本地）名称。

3.2版本中的新功能

alias [name [command]]
: 创建一个别名叫做*名称*，执行*命令*。该命令*不得*包含在引号中。可替换参数可以用%1，%2等等来表示，而%*被所有参数替代。如果未给出命令，则会显示当前*名称的*别名。如果没有给出参数，则会列出所有别名。

别名可以嵌套并可以包含任何可以在pdb提示符下合法键入的内容。请注意，内部pdb命令*可以*被别名覆盖。然后隐藏这样的命令，直到别名被删除。别名被递归地应用于命令行的第一个单词; 该行中的所有其他单词都是单独存在的。

作为一个例子，这里有两个有用的别名（特别是当放在.pdbrc文件中时）：

```
# Print instance variables (usage "pi classInst")
alias pi for k in %1.__dict__.keys(): print("%1.",k,"=",%1.__dict__[k])
# Print instance variables in self
alias ps pi self
```

unalias name
: 删除指定的别名。

! statement
: 在当前堆栈帧的上下文中执行（单行）*语句*。除非语句的第一个单词类似于调试器命令，否则感叹号可以省略。要设置一个全局变量，你可以在赋值命令global的前面添加一行语句，例如：

```
(Pdb) global list_options; list_options = ['-l']
(Pdb)
```

run [args ...]
restart [args ...]
: 重新启动调试的Python程序。如果提供了一个参数，它将被拆分shlex并且结果将被用作新的参数sys.argv。历史记录，断点，动作和调试器选项均被保留。 restart是一个别名run。

q(uit)
: 退出调试器。正在执行的程序被中止。

脚注

[1] 框架是否被认为起源于某个模块是由__name__框架全局决定的。

27.4。Python的廓

源代码： Lib / profile.py和Lib / pstats.py

27.4.1。轮廓仪简介

cProfile并profile提供Python程序的*确定性分析*。一个*配置文件*是一组统计数据，描述如何经常和执行的程序多久各个部分。这些统计数据可以通过pstats模块格式化成报告。

Python标准库提供了相同概要分析界面的两种不同实现：

1. cProfile建议大多数用户使用; 这是一个C扩展，具有合理的开销，使其适合分析长时间运行的程序。基于lsprofBrett Rosen和Ted Czotter的贡献。
2. profile，它是一个纯粹的Python模块，其接口被模仿 cProfile，但是它为剖析程序增加了大量开销。如果您试图以某种方式扩展探查器，则使用此模块可能会更轻松。最初由Jim Roskind设计和撰写。

注意： 分析器模块旨在为给定程序提供执行配置文件，而不是用于基准测试（为此，有timeit相当准确的结果）。这特别适用于将Python代码与C代码进行基准比较：分析器为Python代码引入了开销，但不会为C级函数开销，因此C代码看起来比任何Python代码都要快。

27.4.2。即时用户手册

本节提供给“不想阅读手册”的用户。它提供了一个非常简短的概述，并允许用户快速执行现有应用程序的分析。

要分析一个只有一个参数的函数，你可以这样做：

```
import cProfile
import re
cProfile.run('re.compile("foo|bar")')
```

（使用profile而不是cProfile如果后者无法使用您的系统上。）

上述操作将运行re.compile()并打印如下所示的配置文件结果：

```
         197 function calls (192 primitive calls) in 0.002 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.001    0.001 <string>:1(<module>)
        1    0.000    0.000    0.001    0.001 re.py:212(compile)
        1    0.000    0.000    0.001    0.001 re.py:268(_compile)
        1    0.000    0.000    0.000    0.000 sre_compile.py:172(_compile_charset)
        1    0.000    0.000    0.000    0.000 sre_compile.py:201(_optimize_charset)
```

```
     4    0.000    0.000    0.000    0.000 sre_compile.py:25(_identityfunction)
   3/1    0.000    0.000    0.000    0.000 sre_compile.py:33(_compile)
```

第一行表示197个电话被监控。在这些调用中，192是*原始的*，这意味着调用不是通过递归来引发的。下一行：表示最右列中的文本字符串用于对输出进行排序。列标题包括：`Ordered by: standard name`

ncalls
: 为通话的数量。

tottime
: 在给定函数中花费的总时间（并且不包括调用子函数的时间）

percall
: 是`tottime`除以的商`ncalls`

cumtime
: 是在这个和所有子函数（从调用到退出）中累积的时间。*即使*是递归函数，这个数字也是准确的。

percall
: 是`cumtime`除以原始调用的商

文件名：LINENO（功能）
: 提供每个功能的相应数据

如果第一列中有两个数字（例如`3/1`），则意味着该函数被递归。第二个值是原始呼叫的数量，前者是呼叫的总数。请注意，当函数不递归时，这两个值是相同的，并且只打印单个数字。

除了在配置文件运行结束时打印输出，您可以通过为`run()`函数指定文件名将结果保存到文件中：

```
import cProfile
import re
cProfile.run('re.compile("foo|bar")', 'restats')
```

该`pstats.Stats`班从文件中读取配置文件的结果，并格式化它们以不同的方式。

该文件`cProfile`也可以作为脚本来调用另一个脚本。例如：

```
python -m cProfile [-o output_file] [-s sort_order] myscript.py
```

`-o` 将配置文件结果写入文件而不是stdout

`-s`指定一个`sort_stats()`排序值来排序输出。这仅适用于`-o`未提供的情况。

该`pstats`模块的`Stats`类具有多种方法来操作和打印保存到配置文件结果文件中的数据：

```
import pstats
p = pstats.Stats('restats')
p.strip_dirs().sort_stats(-1).print_stats()
```

该`strip_dirs()`方法从所有模块名称中删除了无关路径。该`sort_stats()`方法根据打印的标准模块/行/名称字符串对所有条目进行排序。该 `print_stats()`方法打印出所有的统计数据。您可

以尝试以下排序调用：

```
p.sort_stats('name')
p.print_stats()
```

第一次调用实际上将按函数名称对列表进行排序，第二次调用将打印统计信息。以下是一些有趣的电话实验：

```
p.sort_stats('cumulative').print_stats(10)
```

这会按照函数中的累积时间排序配置文件，然后仅打印十条最重要的行。如果您想了解哪些算法需要花费时间，那么您将使用上述行。

如果你想看看哪些函数循环了很多，并花费很多时间，你会这样做：

```
p.sort_stats('time').print_stats(10)
```

根据每个功能花费的时间进行排序，然后打印前十个功能的统计数据。

你也可以尝试：

```
p.sort_stats('file').print_stats('__init__')
```

这将按文件名对所有统计信息进行排序，然后仅打印出类init方法的统计信息（因为它们拼写__init__在其中）。作为最后一个例子，你可以尝试：

```
p.sort_stats('time', 'cumulative').print_stats(.5, 'init')
```

该行使用时间主键和累积时间的辅助键对统计信息进行排序，然后打印出一些统计信息。具体而言，列表首先被降至.5其原始大小的50％（re：），然后仅init保留行，并打印该子子列表。

如果您想知道哪些函数称为上述函数，您现在可以（p仍然按照最后一个标准排序）：

```
p.print_callers(.5, 'init')
```

并且您将获得每个列出功能的呼叫者列表。

如果你需要更多的功能，你将不得不阅读手册，或猜测以下功能的作用：

```
p.print_callees()
p.add('restats')
```

作为脚本调用，该pstats模块是用于阅读和检查配置文件转储的统计浏览器。它有一个简单的面向行的界面（使用实现cmd）和交互式帮助。

27.4.3。profile和cProfile模块参考

两个profile和cProfile模块提供以下功能：

profile.run(*command*, *filename* = *None*, *sort* = *-1*)

该函数接受一个可以传递给该exec()函数的参数，以及一个可选的文件名。在所有情况下，这个例程执行：

```
exec(command, __main__.__dict__, __main__.__dict__)
```

并从执行中收集分析统计数据。如果没有文件名，则该函数自动创建一个Stats实例并打印一个简单的性能分析报告。如果指定了排序值，则将其传递给此Stats实例以控制结果的排序方式。

profile.runctx(*command*, *globals*, *locals*, *filename* = *None*, *sort* = *-1*)

该函数类似于run()使用添加的参数为*命令*字符串提供全局变量和局部变量字典。该例程执行：

```
exec(command, globals, locals)
```

并收集上述run()功能中的分析统计信息。

class profile.Profile(*timer* = *None*, *timeunit* = *0.0*, *subcalls* = *True*, *builtins* = *True*)

这个类通常只在需要比cProfile.run()函数提供更精确的分析控制时才使用。

自定义计时器可用于测量通过*timer*参数运行代码需要多长时间。这必须是一个返回表示当前时间的单个数字的函数。如果数字是一个整数，那么*timeunit*指定一个乘数，它指定每个时间单位的持续时间。例如，如果计时器返回以数千秒计量的时间，则时间单位将是.001。

直接使用Profile该类可以在不将配置文件数据写入文件的情况下格式化配置文件结果：

```
import cProfile, pstats, io
pr = cProfile.Profile()
pr.enable()
# ... do something ...
pr.disable()
s = io.StringIO()
sortby = 'cumulative'
ps = pstats.Stats(pr, stream=s).sort_stats(sortby)
ps.print_stats()
print(s.getvalue())
```

enable()

开始收集分析数据。

disable()

停止收集分析数据。

create_stats()

停止收集性能分析数据并将内部结果记录为当前配置文件。

print_stats(*sort* = *-1*)

Stats根据当前配置文件创建一个对象并将结果输出到stdout。

dump_stats（*文件名*）
将当前配置*文件*的结果写入文件*名*。

run（*cmd*）
通过分析cmd exec()。

runctx（*cmd*，*globals*，*locals*）
通过exec()指定的全局和本地环境对cmd进行分析。

runcall（*func*，**args*，*** kwargs*）
轮廓 func(*args, **kwargs)

27.4.4。该Stats级

分析器数据分析使用Stats该类进行。

class pstats. Stats（**filenames或profile*，*stream = sys.stdout*）
此类构造函数根据*文件名*（或文件名列表）或实例创建“统计对象” Profile实例。输出将打印到由流指定的*流*。

由上述构造函数选择的文件必须由相应版本的profileor 创建cProfile。具体来说，该分析器的未来版本*没有*文件兼容性保证，并且与其他分析器生成的文件没有兼容性。如果提供了多个文件，则所有相同功能的统计数据将被合并，以便可以在单个报告中考虑多个进程的总体视图。如果需要将其他文件与现有Stats对象中的数据结合使用，则add()可以使用该 方法。

可以将一个cProfile.Profile 或一个profile.Profile对象用作配置文件数据源，而不是从文件读取配置文件数据。

Stats 对象有以下方法：

strip_dirs（）
Stats该类的此方法从文件名中删除所有前导路径信息。它在缩小打印输出尺寸以适应（接近）80列时非常有用。此方法修改对象，并且剥离的信息丢失。执行剥离操作后，该对象被视为以“随机”顺序输入条目，因为它刚好在对象初始化和加载之后。如果strip_dirs()导致两个函数名无法区分（它们位于同一文件名的同一行，并具有相同的函数名），那么这两个条目的统计信息将累积到一个条目中。

add（**文件名*）
Stats该类的此方法将其他分析信息累加到当前分析对象中。它的参数应该引用由相应版本的profile.run() or 创建的文件名cProfile.run()。同名（re：文件，行，名称）函数的统计信息会自动累积到单个函数统计信息中。

dump_stats（*文件名*）
将加载到Stats对象中的数据保存到名为 *filename的文件中*。如果该文件不存在，则会创建该文件，如果该文件已存在，则会覆盖该文件。这相当于profile.Profile和

cProfile.Profile类上的同名方法。

sort_stats（*键）

该方法Stats通过根据提供的标准对对象进行排序来修改对象。参数通常是标识排序基础的字符串（例如：'time'或'name'）。

当提供多于一个密钥时，当在它们之前选择的所有密钥相等时，附加密钥用作次要标准。例如，将根据其函数名称对所有条目进行排序，并通过按文件名排序来解析所有关系（相同的函数名称）。sort_stats('name', 'file')

缩写可以用于任何键名，只要缩写是明确的。以下是目前定义的关键字：

有效的Arg	含义
'calls'	通话计数
'cumulative'	累计时间
'cumtime'	累计时间
'file'	文件名
'filename'	文件名
'module'	文件名
'ncalls'	通话计数
'pcalls'	原始呼叫计数
'line'	电话号码
'name'	函数名称
'nfl'	名/文件/行
'stdname'	标准名称
'time'	内部时间
'tottime'	内部时间

请注意，统计信息的所有排序均按降序排列（首先放置耗时最多的项目），其中名称，文件和行号搜索按升序排列（按字母顺序排列）。之间的微妙的区别'nfl'和'stdname'是标准名称是一种名称为打印，这意味着该嵌入行号获取一种奇怪的方式进行比较。例如，第3行，第20行和第40行（如果文件名相同）将以字符串顺序20,3和40出现。相反，'nfl'对行号进行数字比较。其实sort_stats('nfl')也是一样的。sort_stats('name', 'file', 'line')

对于向后兼容性的原因，数值参数-1，0，1，和2被允许的。它们被解释为'stdname'，'calls'，'time'，和'cumulative'分别。如果使用这种旧样式格式（数字），则只会使用一个排序键（数字键），而其他参数将被忽略。

reverse_order（）

这个Stats类的方法颠倒了对象内基本列表的排序。请注意，默认情况下，升序和降序的选择是基于所选择的排序键进行的。

print_stats（*限制）

这个Stats类的方法按照profile.run()定义中的描述打印出一个报告。

打印顺序sort_stats()取决于在对象上执行的最后一个 操作（需要注意add()和 注意strip_dirs()）。

提供的参数（如果有的话）可用于将列表限制为重要条目。最初，该列表被视为完整的配置功能集。每个限制条件可以是整数（用于选择行数）或包含0.0和1.0之间的小数部分（用于选择行的百分比），也可以是将被解释为正则表达式的字符串（用于模式匹配标准名称这是打印）。如果提供了几个限制，则按顺序应用它们。例如：

```
print_stats(.1, 'foo:')
```

首先将打印限制为列表的前10%，然后仅打印属于文件名的部分的功能.*foo:。相比之下，命令：

```
print_stats('foo:', .1)
```

将列表限制为具有文件名的所有函数.*foo:，然后继续仅打印其中的前10%。

print_callers(*限制)

这个Stats类的方法打印一个调用配置文件数据库中所有调用每个函数的函数的列表。排序与提供的顺序相同print_stats()，并且限制参数的定义也是相同的。每个呼叫者都在自己的线路上报告。格式根据生成统计信息的分析器略有不同：

- 随着profile，数字显示在括号中每个呼叫者后，显示此特定呼叫了多少次提出。为方便起见，第二个非括号的数字重复在右侧的函数中花费的累积时间。
- 同时cProfile，每个调用者前面都有三个数字：该特定调用的次数，以及该特定调用者调用当前函数所花费的总时间和累计时间。

print_callees(*限制)

这个Stats类的方法打印一个由指定函数调用的所有函数的列表。除了调用方向的这种逆转（re：称为vs被调用）之外，参数和顺序与print_callers()方法相同。

27.4.5。什么是确定性分析？

*确定性分析*旨在反映所有*函数调用*，*函数返回*和*异常*事件都受到监视，并且对这些事件之间的间隔（在此期间用户的代码正在执行）进行精确定时。相比之下，*统计分析*（这不是由该模块完成的）随机采样有效指令指针，并推断出在哪里花费时间。后一种技术传统上涉及较少的开销（因为代码不需要检测），但仅提供时间花费在何处的相对指示。

在Python中，由于在执行期间有一个解释器处于活动状态，因此不需要使用检测代码来执行确定性分析。Python会为每个事件自动提供一个*钩子*（可选的回调函数）。此外，Python的解释性质往往会增加执行的开销，确定性分析往往只会在典型应用程序中增加小的处理开销。其结果是确定性分析并不昂贵，但它提供了有关Python程序执行的大量运行时统计信息。

通话计数统计可用于识别代码中的错误（令人惊讶的计数），并识别可能的内联扩展点（高通话计数）。内部时间统计可以用来识别应该仔细优化的“热循环”。应使用累积时间统计来识别算法选择中的高级错误。请注意，在此分析器中对累计时间的异常处理允许将算法的递归实现的统计信息直接与迭代实现进行比较。

27.4.6。限制

一个限制与时间信息的准确度有关。确定性分析器涉及准确性存在根本问题。最明显的限制是底层的“时钟”仅以约0.001秒的速率（典型值）滴答。因此，没有测量结果比底层时钟更准确。如果进行了足够的测量，那么“错误”将趋于平均。不幸的是，消除第一个错误会导致第二个错误。

第二个问题是，从事件发布到分析器调用获取时间的“调用需要一段时间”实际上*会获得*时钟状态。类似地，从获取时钟值（然后松鼠消失）的时间退出事件探查器时，存在一定的滞后，直到用户的代码再次执行。因此，多次调用或调用多个函数的函数通常会累积该错误。以这种方式积累的误差通常小于时钟的精度（小于一个时钟刻度），但它*可以*累积并变得非常重要。

profile与低开销相比， 问题更重要cProfile。由于这个原因，profile为给定的平台提供了一种校准方法，以便可以将这个错误概率地（平均地）移除。在分析器校准后，它会更准确（以最小二乘的方式），但它有时会产生负数（当通话计数特别低时，概率神对你起作用:-)。）不要*不*通过配置文件中的负数惊慌。它们应该*只*在您校准过轮廓仪时出现，并且结果实际上比没有校准时好。

27.4.7。校准

profile模块的分析器从每个事件处理时间中减去一个常量，以补偿调用时间函数的开销，并解除结果。默认情况下，常量为0.可以使用以下过程为给定平台获取更好的常量（请参阅 限制）。

```
import profile
pr = profile.Profile()
for i in range(5):
    print(pr.calibrate(10000))
```

该方法执行由参数给出的Python调用的次数，直接并且再次在分析器下测量两者的时间。然后计算每个事件探查器隐藏的开销，并将其作为一个浮点数返回。例如，在运行Mac OS X的1.8Ghz Intel Core i5上，使用Python的time.clock（ ）作为定时器，神奇的数字约为4.04e-6。

这个练习的目的是得到一个相当一致的结果。如果您的计算机速度*非常*快，或者您的计时器功能的分辨率较差，则可能必须通过100000甚至1000000才能获得一致的结果。

当你有一个一致的答案时，有三种方法可以使用它：

```
import profile

# 1. Apply computed bias to all Profile instances created hereafter.
profile.Profile.bias = your_computed_bias

# 2. Apply computed bias to a specific Profile instance.
pr = profile.Profile()
pr.bias = your_computed_bias

# 3. Specify computed bias in instance constructor.
pr = profile.Profile(bias=your_computed_bias)
```

如果你有选择，你最好选择一个较小的常数，然后你的结果“不经常”在配置文件统计中显示为负值。

27.4.8。使用自定义计时器

如果您想更改当前时间的确定方式（例如，强制使用挂钟时间或流逝的处理时间），请将所需的计时函数传递给Profile类构造函数：

```
pr = profile.Profile(your_time_func)
```

然后生成的分析器将会调用 your_time_func 。根据您使用的 profile.Profile 还是 cProfile.Profile， your_time_func返回值的解释不同：

profile.Profile
: your_time_func应该返回一个单一的数字，或者是一个总数是当前时间的数字列表（比如 os.times() 返回的）。如果函数返回单个时间数字，或者返回的数字列表的长度为2，那么您将得到特别快速的派遣例程。

  被警告您应该校准您选择的定时器功能的探查器类别（请参阅校准）。对于大多数机器来说，返回单个整数值的计时器将在分析过程中提供低成本方面的最佳结果。（ os.times() 是*非常*糟糕的，因为它返回浮点值的元组）。如果您想以最简洁的方式替换更好的计时器，可以派生出一个班级并硬连线替代派遣方法，以最佳方式处理您的计时器呼叫以及适当的校准常数。

cProfile.Profile
: your_time_func应该返回一个数字。如果它返回整数，也可以用第二个参数调用类构造函数，该参数指定一个单位时间的实际持续时间。例如，如果 your_integer_time_func返回时间以千位秒为单位进行计算，则您将按Profile如下所示构造实例：

  ```
  pr = cProfile.Profile(your_integer_time_func, 0.001)
  ```

  由于cProfile.Profile班级无法校准，因此应小心使用自定义计时器功能，并且应尽可能快。为了使用定制定时器获得最佳结果，可能需要在内部_lsprof模块的C源代码中对其进行硬编码。

Python 3.3增加了几个新功能 time ，可用于精确测量流程或挂钟时间。例如，请参阅 time.perf_counter() 。

27.5。 timeit- 测量小代码片段的执行时间

源代码： Lib / timeit.py

这个模块提供了一个简单的方法来计算一小段Python代码。它既有命令行界面，也有可调用的界面 。它避免了测量执行时间的一些常见陷阱。另请参阅Tim Peters在O'Reilly出版的*Python Cookbook*的“算法”一章中的介绍。

27.5.1。基本示例

以下示例显示了如何使用命令行界面 比较三个不同的表达式：

```
$ python3 -m timeit '"-".join(str(n) for n in range(100))'
10000 loops, best of 3: 30.2 usec per loop
$ python3 -m timeit '"-".join([str(n) for n in range(100)])'
10000 loops, best of 3: 27.5 usec per loop
$ python3 -m timeit '"-".join(map(str, range(100)))'
10000 loops, best of 3: 23.2 usec per loop
```

这可以通过以下Python接口来实现：

```
>>> import timeit
>>> timeit.timeit('"-".join(str(n) for n in range(100))', number=10000)
0.3018611848820001
>>> timeit.timeit('"-".join([str(n) for n in range(100)])', number=10000)
0.2727368790656328
>>> timeit.timeit('"-".join(map(str, range(100)))', number=10000)
0.23702679807320237
```

但请注意，`timeit`只有在使用命令行界面时，才会自动确定重复次数。在“ 示例”部分中，您可以找到更多高级示例。

27.5.2。Python接口

该模块定义了三个便利功能和一个公开课：

`timeit.timeit`（*stmt ='pass'*，*setup ='pass'*，*timer = <default timer>*，*number = 1000000*，*globals = None* ）

`Timer`使用给定的语句，*设置*代码和 *计时器*函数创建一个实例，并`timeit()`使用*数字*执行运行它的方法。可选的*全局变量*参数指定要在其中执行代码的名称空间。

*在版本3.5中更改：*添加了可选的*全局变量*参数。

`timeit.repeat`（*stmt ='pass'*，*setup ='pass'*，*timer = <default timer>*，*repeat = 3*，*number = 1000000*，*globals = None* ）

Timer使用给定的语句，*设置*代码和 *计时器*函数创建一个实例，并repeat()使用给定的*重复*次数和*次数*执行来运行它的方法。可选的*全局变量*参数指定要在其中执行代码的名称空间。

*在版本3.5中更改：*添加了可选的*全局变量*参数。

timeit.**default_timer**( )

默认的定时器，总是time.perf_counter()。

*版本3.3中更改：*time.perf_counter()现在是默认计时器。

class timeit.**Timer**(*stmt='pass'*, *setup='pass'*, *timer=<timer function>*, *globals=None*)

小代码片段的定时执行速度类。

构造函数接受一个语句来定时，另外一个用于设置的语句和一个定时器函数。两个语句默认为'pass'；定时器功能与平台有关（请参阅模块doc字符串）。 *stmt*和*setup*也可能包含多个由；或换行符分隔的语句，只要它们不包含多行字符串文字。该语句将默认在timeit的名称空间内执行; 这个行为可以通过传递一个名称空间给*全局变量*来控制。

要度量第一条语句的执行时间，请使用该timeit() 方法。该repeat()和autorange()方法是方便的方法来调用timeit()多次。

整个定时执行运行中排除了*设置*的执行时间。

该*语句*和*设置*参数也可以采取的是不带参数的可调用的对象。这将在一个定时器函数中嵌入对它们的调用，然后由它执行timeit()。请注意，由于额外的函数调用，在这种情况下计时开销稍大。

*在版本3.5中更改：*添加了可选的*全局变量*参数。

timeit(*number=1000000*)

主要语句的时间*编号*执行。这会执行一次设置语句，然后返回多次执行主语句所需的时间，以秒为单位以浮点形式进行度量。论证是循环的次数，默认为100万。主语句，设置语句和要使用的计时器函数传递给构造函数。

> **注意：** 默认情况下，在计时期间timeit()临时关闭垃圾回收。这种方法的优点是它使独立的时间更具可比性。这个缺点是GC可能是被测功能性能的重要组成部分。如果是这样，GC可以重新启用为*安装*字符串中的第一条语句。例如：
>
> ```
> timeit.Timer('for i in range(10): oct(i)', 'gc.enable()').timeit()
> ```

autorange(*callback=None*)

自动确定调用次数timeit()。

这是一个方便的函数，它会timeit()重复调用，以使总时间>= 0.2秒，并返回最终（循环数，循环次数）。它 timeit()以*数字*设置连续幂10（10,100,1000，...），最多可达10亿次，直到所需时间至少为0.2秒，或达到最大值。

如果*回调*被给出而不是None ，它将在每次试验后用两个参数被调用：。
callback(number, time_taken)

3.6版本中的新功能。

repeat (*repeat = 3* , *number = 1000000*)

通话timeit()几次。

这是一个方便的函数，可以timeit()重复调用，返回结果列表。第一个参数指定要调用多少次timeit()。第二个参数指定*数字*参数timeit()。

> **注意:** 计算结果向量的平均值和标准偏差并报告这些信息是很有吸引力的。但是，这不是很有用。在典型情况下，最低值为您的机器运行给定代码片段的速度提供了一个下限; 结果向量中较高的值通常不是由Python的速度变化引起的，而是由其他进程干扰您的定时精度造成的。因此min() ，结果可能是您应该感兴趣的唯一数字。之后，您应该查看整个向量并应用常识而不是统计。

print_exc (*file = None*)

助手从定时代码打印回溯。

典型用途：

```
t = Timer(...)       # outside the try/except
try:
    t.timeit(...)    # or t.repeat(...)
except Exception:
    t.print_exc()
```

与标准回溯相比，优势在于编译模板中的源代码行将被显示。可选的*文件*参数指示回溯发送的位置; 它默认为sys.stderr。

27.5.3。命令行界面

当从命令行调用程序时，将使用以下形式：

```
python -m timeit [-n N] [-r N] [-u U] [-s S] [-t] [-c] [-h] [statement ...]
```

在了解以下选项的情况下：

-n N, --number=N
多少次执行'声明'

-r N, --repeat=N
重复定时器的次数（默认值3）

-s S, --setup=S
语句最初执行一次（默认pass）

-p, --process

测量处理时间，而不是挂钟时间，time.process_time() 而不是使用time.perf_counter()默认值

3.3版本的新功能

-t, --time
使用time.time()（不推荐）

-u, --unit=U

指定定时器输出的时间单位; 可以选择usec，msec或sec

3.5版本中的新功能。

-c, --clock
使用time.clock()（不推荐）

-v, --verbose
打印原始时间结果; 重复更多的数字精度

-h, --help
打印一条简短的使用信息并退出

可以通过将每行指定为单独的语句参数来给出多行语句; 通过在引号中引入参数并使用前导空格，可以缩进行。多个-s选项的处理方式相似。

如果-n没有给出，则通过尝试连续幂为10来计算适当数量的循环，直到总时间至少为0.2秒。

default_timer()测量可能会受到同一台机器上运行的其他程序的影响，所以当需要准确定时时最好的做法是重复几次定时并使用最佳时间。该-r 选项对此很有帮助; 在大多数情况下，3次重复的默认值可能就足够了。您可以使用time.process_time()来衡量CPU时间。

注意: 执行通过语句会产生一定的基线开销。这里的代码并不试图隐藏它，但你应该知道它。基线开销可以通过调用没有参数的程序来衡量，并且它可能在Python版本中有所不同。

27.5.4。示例

可以提供一个在开始时只执行一次的设置语句：

```
$ python -m timeit -s 'text = "sample string"; char = "g"'  'char in text'
10000000 loops, best of 3: 0.0877 usec per loop
$ python -m timeit -s 'text = "sample string"; char = "g"'  'text.find(char)'
1000000 loops, best of 3: 0.342 usec per loop
```

```
>>> import timeit
>>> timeit.timeit('char in text', setup='text = "sample string"; char = "g"')
0.41440500499993504
>>> timeit.timeit('text.find(char)', setup='text = "sample string"; char = "g"')
1.7246671520006203
```

使用Timer该类及其方法也可以做到这一点：

```
>>> import timeit
>>> t = timeit.Timer('char in text', setup='text = "sample string"; char = "g"')
>>> t.timeit()
0.3955516149999312
>>> t.repeat()
[0.40193588800002544, 0.3960157959998014, 0.39594301399984033]
```

以下示例显示如何计算包含多行的表达式。在这里我们比较使用hasattr()和try/ except 或测试丢失和现在对象属性的成本：

```
$ python -m timeit 'try:' '  str.__bool__' 'except AttributeError:' '  pass'
100000 loops, best of 3: 15.7 usec per loop
$ python -m timeit 'if hasattr(str, "__bool__"): pass'
100000 loops, best of 3: 4.26 usec per loop

$ python -m timeit 'try:' '  int.__bool__' 'except AttributeError:' '  pass'
1000000 loops, best of 3: 1.43 usec per loop
$ python -m timeit 'if hasattr(int, "__bool__"): pass'
100000 loops, best of 3: 2.23 usec per loop
```

```
>>> import timeit
>>> # attribute is missing
>>> s = """\
... try:
...     str.__bool__
... except AttributeError:
...     pass
... """
>>> timeit.timeit(stmt=s, number=100000)
0.9138244460009446
>>> s = "if hasattr(str, '__bool__'): pass"
>>> timeit.timeit(stmt=s, number=100000)
0.5829014980008651
>>>
>>> # attribute is present
>>> s = """\
... try:
...     int.__bool__
... except AttributeError:
...     pass
... """
>>> timeit.timeit(stmt=s, number=100000)
0.04215312199994514
>>> s = "if hasattr(int, '__bool__'): pass"
>>> timeit.timeit(stmt=s, number=100000)
0.08588060699912603
```

要让timeit模块访问您定义的函数，您可以传递一个包含导入语句的 *设置*参数：

```
def test():
    """Stupid test function"""
    L = [i for i in range(100)]
```

```
if __name__ == '__main__':
    import timeit
    print(timeit.timeit("test()", setup="from __main__ import test"))
```

另一个选择是传递`globals()`给 *globals*参数，这会导致代码在当前的全局名称空间内执行。这比单独指定导入更方便：

```
def f(x):
    return x**2
def g(x):
    return x**4
def h(x):
    return x**8

import timeit
print(timeit.timeit('[func(42) for func in (f,g,h)]', globals=globals()))
```

27.6。 trace-跟踪或跟踪的Python语句执行

源代码： Lib / trace.py

该trace模块允许您跟踪程序执行，生成带注释的语句覆盖列表，打印在程序运行期间执行的调用者/被调用者关系和列表函数。它可以在另一个程序或命令行中使用。

也可以看看:

Coverage.py
一种流行的第三方覆盖工具，可提供HTML输出以及分支机构覆盖等高级功能。

27.6.1。命令行用法

该trace模块可以从命令行调用。它可以如此简单

```
python -m trace --count -C . somefile.py ...
```

以上将执行somefile.py并生成执行过程中导入到当前目录中的所有Python模块的注释列表。

--help
显示使用情况并退出。

--version
显示模块的版本并退出。

27.6.1.1。主要选项

调用时必须至少指定以下选项之一 trace。该--listfuncs选项与--trace和选项是互斥的--count。何时 --listfuncs提供，既不被接受--count也不 --trace被接受，反之亦然。

-c，--count
在程序完成时产生一组带注释的列表文件，显示每个语句执行的次数。另见 --coverdir，--file和 --no-report下面。

-t，--trace
在执行时显示行。

-l，--listfuncs
显示通过运行程序执行的功能。

-r，--report
从使用--countand --file选项的早期程序运行中生成注释列表 。这不会执行任何代码。

-T，--trackcalls

显示通过运行程序暴露的调用关系。

27.6.1.2。修饰符

-f，--file=<file>
要积累的文件的名称将在多次跟踪运行中计数。应该与--count选项一起使用。

-C，--coverdir=<dir>
报告文件所在的目录。覆盖率报告 package.module写入文件。 *dir/package/module*.cover

-m，--missing
当生成带注释的列表时，标记未被执行的行 >>>>>>。

-s，--summary
在使用--count或时--report ，为每个处理的文件写一个简短的总结到stdout。

-R，--no-report
不要生成带注释的列表。如果您打算使用几次运行--count ，然后在最后生成一组注释列表，这非常有用。

-g，--timing
在每行的前面加上自程序启动以来的时间。仅在追踪时使用。

27.6.1.3。过滤器

这些选项可能会重复多次。

--ignore-module=<mod>
忽略每个给定的模块名称及其子模块（如果它是一个包）。参数可以是用逗号分隔的名称列表。

--ignore-dir=<dir>
忽略指定目录和子目录中的所有模块和程序包。该参数可以是一个由目录分隔的目录列表 os.pathsep。

27.6.2。编程接口

class trace.Trace（*count = 1*，*trace = 1*，*countfuncs = 0*，*countcallers = 0*，*ignoremods = ( )*，*ignoredirs = ( )*，*infile = None*，*outfile = None*，*timing = False* ）
创建一个对象来跟踪单个语句或表达式的执行。所有参数都是可选的。 *计数*可以计算行号。 *跟踪*启用行跟踪。 *countfuncs*可以列出运行期间调用的函数。 *计数器*支持呼叫关系跟踪。 *ignoremods*是要忽略的模块或软件包的列表。 *ignoredirs*是应该忽略模块或包的目录列表。 *infile*是从中读取存储计数信息的文件的名称。 *outfile*是在其中写入更新的计数信息的文件的名称。 *定时*启用相对于何时开始显示跟踪的时间戳。

run（*cmd* ）

执行命令并使用当前跟踪参数从执行中收集统计信息。 *cmd*必须是一个字符串或代码对象，适合传入exec()。

runctx (*cmd* , *globals = None* , *locals = None*)

在已定义的全局和本地环境中，执行命令并使用当前跟踪参数从执行中收集统计信息。如果未定义，*全局变量*和*本地变量*默认为空字典。

runfunc (*func* , ** args* , *** kwds*)

使用当前跟踪参数在对象控制下调用给定参数的*func*Trace。

results ( )

返回一个CoverageResults包含所有以前调用的累积结果的对象run ，runctx而runfunc 对于给定的Trace实例。不重置累积的跟踪结果。

*类*trace. CoverageResults

覆盖率结果的容器，由...创建Trace.results()。不应该由用户直接创建。

update (*其他*)

合并来自另一个CoverageResults对象的数据。

write_results (*show_missing = True* , *summary = False* , *coverdir = None*)

撰写报道结果。设置*show_missing*以显示没有匹配的行。设置*摘要*以在输出中包含每个模块的覆盖率摘要。 *coverdir*指定将输出覆盖率结果文件的目录。如果None ，每个源文件的结果都放在其目录中。

演示使用编程接口的简单示例：

```
import sys
import trace

# create a Trace object, telling it what to ignore, and whether to
# do tracing or line-counting or both.
tracer = trace.Trace(
    ignoredirs=[sys.prefix, sys.exec_prefix],
    trace=0,
    count=1)

# run the new command using the given tracer
tracer.run('main()')

# make a report, placing output in the current directory
r = tracer.results()
r.write_results(show_missing=True, coverdir=".")
```

27.7。tracemalloc- 跟踪内存分配

3.4版新增功能

源代码： Lib / tracemalloc.py

tracemalloc模块是一个跟踪由Python分配的内存块的调试工具。它提供了以下信息：

- 回溯分配对象的位置
- 统计每个文件名和每行号分配的内存块数：总大小，分配的内存块的数量和平均大小
- 计算两个快照之间的差异以检测内存泄漏

为了跟踪大部分由Python分配的内存块，应该尽早通过设置来启动模块 PYTHONTRACEMALLOC环境变量1，或通过使用命令行选项。可以在运行时调用该函数来开始跟踪Python内存分配。 -X tracemalloc tracemalloc.start()

默认情况下，分配的内存块的跟踪仅存储最近的帧（ 1帧 ）。要在启动时存储25帧：设置 PYTHONTRACEMALLOC环境变量25，或使用 命令行选项。 -X tracemalloc=25

27.7.1。示例

27.7.1.1。显示顶层10

显示分配最多内存的10个文件：

```
import tracemalloc

tracemalloc.start()

# ... run your application ...

snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('lineno')

print("[ Top 10 ]")
for stat in top_stats[:10]:
    print(stat)
```

Python测试套件输出示例：

```
[ Top 10 ]
<frozen importlib._bootstrap>:716: size=4855 KiB, count=39328, average=126 B
<frozen importlib._bootstrap>:284: size=521 KiB, count=3199, average=167 B
/usr/lib/python3.4/collections/__init__.py:368: size=244 KiB, count=2315, average=108
/usr/lib/python3.4/unittest/case.py:381: size=185 KiB, count=779, average=243 B
/usr/lib/python3.4/unittest/case.py:402: size=154 KiB, count=378, average=416 B
/usr/lib/python3.4/abc.py:133: size=88.7 KiB, count=347, average=262 B
<frozen importlib._bootstrap>:1446: size=70.4 KiB, count=911, average=79 B
<frozen importlib._bootstrap>:1454: size=52.0 KiB, count=25, average=2131 B
```

```
<string>:5: size=49.7 KiB, count=148, average=344 B
/usr/lib/python3.4/sysconfig.py:411: size=48.0 KiB, count=1, average=48.0 KiB
```

我们可以看到Python 从模块中加载了数据（字节码和常量），并且分配了模块来构建 类型。4855 KiBcollections244 KiBnamedtuple

查看Snapshot.statistics()更多选项。

27.7.1.2。计算差异

拍两张快照并显示差异：

```
import tracemalloc
tracemalloc.start()
# ... start your application ...

snapshot1 = tracemalloc.take_snapshot()
# ... call the function leaking memory ...
snapshot2 = tracemalloc.take_snapshot()

top_stats = snapshot2.compare_to(snapshot1, 'lineno')

print("[ Top 10 differences ]")
for stat in top_stats[:10]:
    print(stat)
```

运行Python测试套件的一些测试之前/之后的输出示例：

```
[ Top 10 differences ]
<frozen importlib._bootstrap>:716: size=8173 KiB (+4428 KiB), count=71332 (+39369), a
/usr/lib/python3.4/linecache.py:127: size=940 KiB (+940 KiB), count=8106 (+8106), ave
/usr/lib/python3.4/unittest/case.py:571: size=298 KiB (+298 KiB), count=589 (+589), a
<frozen importlib._bootstrap>:284: size=1005 KiB (+166 KiB), count=7423 (+1526), avera
/usr/lib/python3.4/mimetypes.py:217: size=112 KiB (+112 KiB), count=1334 (+1334), ave
/usr/lib/python3.4/http/server.py:848: size=96.0 KiB (+96.0 KiB), count=1 (+1), averag
/usr/lib/python3.4/inspect.py:1465: size=83.5 KiB (+83.5 KiB), count=109 (+109), avera
/usr/lib/python3.4/unittest/mock.py:491: size=77.7 KiB (+77.7 KiB), count=143 (+143),
/usr/lib/python3.4/urllib/parse.py:476: size=71.8 KiB (+71.8 KiB), count=969 (+969), a
/usr/lib/python3.4/contextlib.py:38: size=67.2 KiB (+67.2 KiB), count=126 (+126), ave
```

我们可以看到Python已经加载了模块数据（字节码和常量），而且这些数据比在测试之前加载的时间多得多，而且之前的快照已经完成。同样，该 模块已经缓存了Python源代码来格式化回溯，所有这些都是自上一次快照以来的回溯。8173 KiB4428 KiBlinecache940 KiB

如果系统的可用内存很少，则可以使用该Snapshot.dump()方法在磁盘上将快照写入磁盘，从而脱机分析快照。然后使用该 Snapshot.load()方法重新加载快照。

27.7.1.3。获取内存块的追踪

代码显示最大内存块的回溯：

```
import tracemalloc

# Store 25 frames
tracemalloc.start(25)

# ... run your application ...

snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('traceback')

# pick the biggest memory block
stat = top_stats[0]
print("%s memory blocks: %.1f KiB" % (stat.count, stat.size / 1024))
for line in stat.traceback.format():
    print(line)
```

Python测试套件输出示例（回溯限制为25帧）：

```
903 memory blocks: 870.1 KiB
  File "<frozen importlib._bootstrap>", line 716
  File "<frozen importlib._bootstrap>", line 1036
  File "<frozen importlib._bootstrap>", line 934
  File "<frozen importlib._bootstrap>", line 1068
  File "<frozen importlib._bootstrap>", line 619
  File "<frozen importlib._bootstrap>", line 1581
  File "<frozen importlib._bootstrap>", line 1614
  File "/usr/lib/python3.4/doctest.py", line 101
    import pdb
  File "<frozen importlib._bootstrap>", line 284
  File "<frozen importlib._bootstrap>", line 938
  File "<frozen importlib._bootstrap>", line 1068
  File "<frozen importlib._bootstrap>", line 619
  File "<frozen importlib._bootstrap>", line 1581
  File "<frozen importlib._bootstrap>", line 1614
  File "/usr/lib/python3.4/test/support/__init__.py", line 1728
    import doctest
  File "/usr/lib/python3.4/test/test_pickletools.py", line 21
    support.run_doctest(pickletools)
  File "/usr/lib/python3.4/test/regrtest.py", line 1276
    test_runner()
  File "/usr/lib/python3.4/test/regrtest.py", line 976
    display_failure=not verbose)
  File "/usr/lib/python3.4/test/regrtest.py", line 761
    match_tests=ns.match_tests)
  File "/usr/lib/python3.4/test/regrtest.py", line 1563
    main()
  File "/usr/lib/python3.4/test/__main__.py", line 3
    regrtest.main_in_temp_cwd()
  File "/usr/lib/python3.4/runpy.py", line 73
    exec(code, run_globals)
  File "/usr/lib/python3.4/runpy.py", line 160
    "__main__", fname, loader, pkg_name)
```

我们可以看到，内存最多的是在分配importlib模块加载从模块的数据（字节码和常量）。回溯是最近加载的数据的位置：在模块的 行上。如果加载新模块，回溯可能会改变。 870.1 KiB importlib import pdb doctest

27.7.1.4。漂亮的顶部

代码显示10行，分配最多的内存与一个漂亮的输出，忽略和文件：<frozen importlib._bootstrap><unknown>

```
import linecache
import os
import tracemalloc

def display_top(snapshot, key_type='lineno', limit=10):
    snapshot = snapshot.filter_traces((
        tracemalloc.Filter(False, "<frozen importlib._bootstrap>"),
        tracemalloc.Filter(False, "<unknown>"),
    ))
    top_stats = snapshot.statistics(key_type)

    print("Top %s lines" % limit)
    for index, stat in enumerate(top_stats[:limit], 1):
        frame = stat.traceback[0]
        # replace "/path/to/module/file.py" with "module/file.py"
        filename = os.sep.join(frame.filename.split(os.sep)[-2:])
        print("#%s: %s:%s: %.1f KiB"
              % (index, filename, frame.lineno, stat.size / 1024))
        line = linecache.getline(frame.filename, frame.lineno).strip()
        if line:
            print('    %s' % line)

    other = top_stats[limit:]
    if other:
        size = sum(stat.size for stat in other)
        print("%s other: %.1f KiB" % (len(other), size / 1024))
    total = sum(stat.size for stat in top_stats)
    print("Total allocated size: %.1f KiB" % (total / 1024))

tracemalloc.start()

# ... run your application ...

snapshot = tracemalloc.take_snapshot()
display_top(snapshot)
```

Python测试套件输出示例：

```
Top 10 lines
#1: Lib/base64.py:414: 419.8 KiB
    _b85chars2 = [(a + b) for a in _b85chars for b in _b85chars]
#2: Lib/base64.py:306: 419.8 KiB
    _a85chars2 = [(a + b) for a in _a85chars for b in _a85chars]
#3: collections/__init__.py:368: 293.6 KiB
    exec(class_definition, namespace)
#4: Lib/abc.py:133: 115.2 KiB
    cls = super().__new__(mcls, name, bases, namespace)
#5: unittest/case.py:574: 103.1 KiB
    testMethod()
#6: Lib/linecache.py:127: 95.4 KiB
```

```
    lines = fp.readlines()
#7: urllib/parse.py:476: 71.8 KiB
    for a in _hexdig for b in _hexdig}
#8: <string>:5: 62.0 KiB
#9: Lib/_weakrefset.py:37: 60.0 KiB
    self.data = set()
#10: Lib/base64.py:142: 59.8 KiB
    _b32tab2 = [a + b for a in _b32tab for b in _b32tab]
6220 other: 3602.8 KiB
Total allocated size: 5303.1 KiB
```

查看Snapshot.statistics()更多选项。

27.7.2。API

27.7.2.1。函数

tracemalloc.**clear_traces**( )

清除由Python分配的内存块的痕迹。

另见stop()。

tracemalloc.**get_object_traceback**(*obj*)

获取分配Python对象*obj*的回溯。返回一个Traceback实例，或者None如果tracemalloc 模块没有跟踪内存分配或者没有跟踪对象的分配。

另请参阅gc.get_referrers()和sys.getsizeof()功能。

tracemalloc.**get_traceback_limit**( )

获取存储在跟踪追踪中的最大帧数。

该tracemalloc模块必须跟踪内存分配以获得限制，否则会引发异常。

该限制由start()功能设置。

tracemalloc.**get_traced_memory**( )

获取tracemalloc模块作为元组跟踪的内存块的当前大小和峰值大小 ：。(current: int, peak: int)

tracemalloc.**get_tracemalloc_memory**( )

获取tracemalloc用于存储内存块跟踪的模块的内存使用量（以字节为单位）。退货int。

tracemalloc.**is_tracing**( )

True如果该tracemalloc模块正在跟踪Python内存分配，False否则。

另请参阅start()和stop()功能。

tracemalloc.**start**(*nframe : int = 1*)

开始跟踪Python内存分配：在Python内存分配器上安装钩子。收集到的追踪痕迹将限于 *nframe* 帧。默认情况下，内存块的跟踪仅存储最近的帧：限制为1。*nframe*必须大于或等于1。

存储多个1帧仅用于计算按分组'traceback'或统计累积统计信息的统计信息：请参阅Snapshot.compare_to()和Snapshot.statistics()方法。

存储更多帧会增加tracemalloc模块的内存和CPU开销 。使用该get_tracemalloc_memory()功能来测量tracemalloc模块使用了多少内存。

该 PYTHONTRACEMALLOC环境变量（ PYTHONTRACEMALLOC=NFRAME ）和 命令行选项可用于在启动时启动跟踪。 -X tracemalloc=NFRAME

另见stop() ， is_tracing()和get_traceback_limit() 功能。

tracemalloc.stop ( )

停止跟踪Python内存分配：卸载Python内存分配器上的钩子。还清除Python分配的所有以前收集的内存块的痕迹。

调用take_snapshot()函数在清除轨迹之前拍摄轨迹快照。

另见start() ， is_tracing()和clear_traces() 功能。

tracemalloc.take_snapshot ( )

拍摄由Python分配的内存块的轨迹快照。返回一个新的 Snapshot实例。

快照不包括在tracemalloc模块开始追踪内存分配之前分配的内存块 。

痕迹追溯限于get_traceback_limit() 帧。使用函数的*nframe*参数start() 来存储更多的帧。

该tracemalloc模块必须跟踪内存分配以拍摄快照，请参阅该start()功能。

另请参阅该get_object_traceback()功能。

27.7.2.2。DomainFilter

*类*tracemalloc.DomainFilter (*包含：bool* , *domain：int*)

通过地址空间（域）过滤内存块的痕迹。

3.6版本中的新功能。

inclusive

如果*包*是True（包括），搭配内存块的地址空间分配domain。

如果*包*是False（排除），搭配内存块不在地址空间分配domain。

domain

内存块的地址空间（ int ）。只读属性。

27.7.2.3。过滤器

class tracemalloc.Filter（*包括：bool*，*filename_pattern：str*，*lineno：int = None*，*all_frames：bool = False*，*domain：int = None* ）

过滤内存块的痕迹。

请参阅*filename_pattern*fnmatch.fnmatch()语法的函数。该文件的扩展名被替换。'.pyc''.py'

例子：

- Filter(True, subprocess.__file__)只包含subprocess模块的痕迹
- Filter(False, tracemalloc.__file__)不包括tracemalloc模块的痕迹
- Filter(False, "<unknown>") 排除空回溯

*改变在3.5版本：*将'.pyo'文件扩展名不再与更换'.py'。

*版本3.6中已更改：*添加了该domain属性。

domain

内存块（int或None）的地址空间。

inclusive

如果*包含*为True（包含），则只匹配在名称与filename_pattern行号 匹配的文件中分配的内存块lineno。

如果*包*是False（排除），忽略一个文件名匹配分配的内存块filename_pattern的行号lineno。

lineno

行号（int）的过滤器。如果*lineno*是None，则过滤器匹配任何行号。

filename_pattern

过滤器的文件名模式（str）。只读属性。

all_frames

如果*all_frames*是True，则检查回溯的所有帧。如果 *all_frames*是False，则只检查最近的帧。

如果追溯限制为，则此属性不起作用1。查看 get_traceback_limit()功能和Snapshot.traceback_limit 属性。

27.7.2.4。框架

*类*tracemalloc.Frame

追溯的框架。

该Traceback班是一个序列Frame实例。

filename
文件名（str）。

lineno
行号（int）。

27.7.2.5。快照

*类*tracemalloc.Snapshot
Python分配的内存块的快照的快照。

该take_snapshot()函数创建一个快照实例。

compare_to（*old_snapshot：Snapshot*，*key_type：str*，*cumulative：bool = False*）
用旧快照计算差异。将统计信息作为按*key_type*StatisticDiff分组的实例的排序列表。

请参阅*key_type*和*累积*参数的Snapshot.statistics()方法。

结果从最大的排序，以最小的：绝对值StatisticDiff.size_diff，StatisticDiff.size的绝对值StatisticDiff.count_diff，Statistic.count然后通过StatisticDiff.traceback。

dump（*文件名*）
将快照写入文件。

使用load()重新加载快照。

filter_traces（*过滤器*）
Snapshot使用已过滤的traces 序列创建新实例，过滤*器*是实例DomainFilter和 列表Filter。如果*filters*是一个空列表，则返回一个Snapshot带有trace的副本的新 实例。

全部包含过滤器一次应用，如果没有包含过滤器匹配，则会忽略跟踪。如果至少有一个独占过滤器匹配，则跟踪将被忽略。

*在版本3.6中更改：*DomainFilter实例现在也在*过滤器中*被接受。

classmethod load（*filename*）
从文件加载快照。

另见dump()。

statistics（*key_type：str*，*cumulative：bool = False*）
将统计信息作为按*key_type*Statistic分组的实例的排序列表：

为key_type	描述
'filename'	文件名

为key_type	描述
'lineno'	文件名和行号
'traceback'	追溯

如果*累计*的True ，累积的大小和轨迹的追踪，不仅最近的帧的所有帧的内存块的数量。累积模式只能用于*key_type*等于 'filename' 和'lineno' 。

结果按以下顺序从最大到最小排序： Statistic.size ， Statistic.count 然后按Statistic.traceback。

traceback_limit

回溯中存储的最大帧数traces ：get_traceback_limit()拍摄快照时的结果。

traces

由Python分配的所有内存块的跟踪：Trace实例序列 。

该序列具有未定义的顺序。使用该Snapshot.statistics() 方法获取排序的统计列表。

27.7.2.6。统计

*类*tracemalloc.Statistic

内存分配统计。

Snapshot.statistics()返回Statistic实例列表。

另见StatisticDiff课程。

count

内存块的数量（ int ）。

size

内存块的总大小（以字节为单位int ）。

traceback

追溯内存块被分配的地方， Traceback 实例。

27.7.2.7。StatisticDiff

*类*tracemalloc.StatisticDiff

旧Snapshot实例和新实例之间内存分配的统计差异 。

Snapshot.compare_to()返回StatisticDiff 实例列表。另见Statistic课程。

count

新快照（ int ）中的内存块数量：0是否已在新快照中释放内存块。

count_diff

旧快照和新快照之间的内存块数量差异（ int ）：0是否已在新快照中分配内存块。

size
新快照中的内存块总大小（以字节为单位int）（ ）： 0如果内存块已在新快照中释放。

size_diff
新旧快照之间的内存块总大小（以字节为单位）的差异（ int ）：0如果内存块已在新快照中分配。

traceback
追溯内存块分配的地方， Traceback 实例。

27.7.2.8。跟踪

*类*tracemalloc. Trace
内存块的跟踪。

该Snapshot.traces属性是一系列Trace 实例。

size
内存块的大小（以字节为单位int）。

traceback
追溯内存块被分配的地方， Traceback 实例。

27.7.2.9。追踪

*类*tracemalloc. Traceback
Frame从最近帧到最早帧排序的实例序列。

回溯包含至少一1帧。如果tracemalloc模块无法获取帧，则使用"<unknown>"行号处的文件名0。

当拍摄快照时，痕迹的追溯限于 get_traceback_limit()帧。查看take_snapshot()功能。

该Trace.traceback属性是实例的一个Traceback 实例。

format（*限制=无*）
将traceback格式化为带有换行符的行列表。使用该 linecache模块从源代码中检索行。如果 *限制*设置，只能格式化*限*最近的帧。

与traceback.format_tb()函数类似，但 format()不包括换行符。

例：

```
print("Traceback (most recent call first):")
for line in traceback:
    print(line)
```

输出：

```
Traceback (most recent call first):
  File "test.py", line 9
    obj = Object()
  File "test.py", line 12
    tb = tracemalloc.get_object_traceback(f())
```

28.软件包装和分销

这些库可帮助您发布和安装Python软件。虽然这些模块被设计为与Python包索引结合使用 ，但它们也可以与本地索引服务器一起使用，或者根本不使用任何索引服务器。

28.1。distutils- 构建和安装Python模块

该distutils软件包为构建和安装其他模块到Python安装提供支持。新模块可以是100％纯Python，也可以是用C语言编写的扩展模块，也可以是Python包的集合，包括Python和C编码的模块。

大多数Python用户*不*希望直接使用此模块，而是使用由Python Packaging Authority维护的跨版本工具。特别是， setuptools是一个增强的替代方案，distutils它提供了：

- 支持声明项目依赖关系
- 用于配置在源版本中包含哪些文件的其他机制（包括用于与版本控制系统集成的插件）
- 声明项目“入口点”的能力，这可以作为应用程序插件系统的基础
- 能够在安装时自动生成Windows命令行可执行文件，而不需要预编译它们
- 在所有支持的Python版本中一致的行为

即使脚本本身只导入，推荐的pip安装程序也会运行所有 setup.py脚本。有关更多信息，请参阅Python打包用户指南。 setuptoolsdistutils

对于需要更深入了解当前包装和分发系统细节的包装工具作者和用户的好处，distutils基于遗留的用户文档和API参考仍可用：

- 安装Python模块（旧版本）
- 分发Python模块（旧版本）

28.2。ensurepip- 引导pip安装程序

3.4版新增功能

该ensurepip软件包支持将pip 安装程序引导到现有的Python安装或虚拟环境中。这种引导方式反映了这样pip一个事实，即具有其自身发布周期的独立项目，最新的可用稳定版本与CPython参考解释器的维护和功能版本捆绑在一起。

在大多数情况下，Python的最终用户不需要直接调用该模块（ pip默认情况下应该引导），但是如果pip在安装Python时（或创建虚拟环境时）跳过安装，或者在显式卸载后跳过安装 pip。

注意： 该模块*不能*访问互联网。引导所需的所有组件pip都包含在软件包的内部部分中。

也可以看看：

安装Python模块
用于安装Python包的最终用户指南

PEP 453：在Python安装中显式引导pip
该模块的原始原理和规范。

28.2.1。命令行界面

使用解释器的-m开关调用命令行界面。

最简单的调用是：

```
python -m ensurepip
```

pip如果尚未安装此调用，则会进行安装，否则不会执行任何操作。为确保安装的版本pip 至少与捆绑版本一样近ensurepip，请传递 --upgrade选项：

```
python -m ensurepip --upgrade
```

默认情况下，pip将安装到当前虚拟环境（如果有一个活动）或系统站点程序包（如果没有活动虚拟环境）。安装位置可以通过两个额外的命令行选项来控制：

- --root <dir>：pip相对于给定的根目录而不是当前活动的虚拟环境（如果有的话）的根目录或当前Python安装的默认根目录进行安装。
- --user：安装pip到用户站点软件包目录中，而不是全局用于当前Python安装（在活动虚拟环境中不允许使用此选项）。

默认情况下，脚本pipX和pipX.Y将要安装（其中XY代表的Python版本用于调用ensurepip）。安装的脚本可以通过两个额外的命令行选项来控制：

- --altinstall：如果请求备用安装，pipX 则*不会*安装该脚本。
- --default-pip：如果请求“默认点”安装，

pip 除了两个常规脚本之外，还将安装脚本。

提供两种脚本选择选项都会触发异常。

*版本3.6.3中已更改：*如果命令失败，则退出状态为非零。

28.2.2。模块

ensurepip 公开了两种编程使用功能：

ensurepip.version（ ）

返回一个字符串，指定在引导环境时将要安装的pip的捆绑版本。

ensurepip.bootstrap（*root = None*，*upgrade = False*，*user = False*，*altinstall = False*，*default_pip = False*，*verbosity = 0* ）

引导pip到当前或指定的环境中。

*root*指定要安装的替代根目录。如果*root*是None，则安装使用当前环境的默认安装位置。

*升级*指示是否将早期版本的现有安装升级pip到捆绑版本。

*用户*表示是否使用用户方案而不是全局安装。

默认情况下，脚本pipX和pipX.Y将要安装（其中XY代表的Python的当前版本）。

如果*altinstall*被设置，那么pipX将*不会*安装。

如果设置了*default_pip*，则pip除了两个常规脚本之外，还将安装它。

设置*altinstall*和*default_pip*都会触发 ValueError。

*详细程度*控制sys.stdout自举操作的输出级别。

> **注意：** 引导过程有两个副作用sys.path和 os.environ。在子进程中调用命令行界面可以避免这些副作用。

> **注意：** 自举过程可以安装其他模块所需的额外模块 pip，但是其他软件不应该假设这些依赖项默认存在（因为可能会在将来的版本中删除依赖项pip）。

28.3。 venv- 创建虚拟环境

3.3版本的新功能

源代码： Lib / venv /

该venv模块为创建具有自己的站点目录的轻型“虚拟环境”提供支持，可选地与系统站点目录隔离。每个虚拟环境都有自己的Python二进制文件（允许创建具有各种Python版本的环境），并且可以在其站点目录中拥有自己的一套独立的已安装Python包。

看到 **PEP 405**获取关于Python虚拟环境的更多信息。

> **注意：** 该pyvenv脚本自Python 3.6起弃用，以支持使用， 以避免任何潜在的混淆，因为虚拟环境将基于哪个Python解释器。 python3 -m venv

28.3.1。创建虚拟环境

通过执行命令完成虚拟环境的创建venv：

```
python3 -m venv /path/to/new/virtual/environment
```

运行此命令将创建目标目录（创建任何不存在的父目录），并pyvenv.cfg使用home指向运行该命令的Python安装的密钥在其中放置一个文件。它还创建了一个bin（或Scripts在Windows上）子目录，其中包含python二进制文件的副本（或者在Windows中为二进制文件）。它也创建一个（最初是空的）lib/pythonX.Y/site-packages 子目录（在Windows上，这是Lib\site-packages）。如果指定了现有的目录，它将被重新使用。

自从3.6版弃用： pyvenv是为Python 3.3和3.4创建虚拟环境的推荐工具，在Python 3.6中已弃用。

在版本3.5中进行了更改： venv现在建议使用这种方法来创建虚拟环境。

> **也可以看看：** Python打包用户指南：创建和使用虚拟环境

在Windows上，venv按如下所示调用该命令：

```
c:\>c:\Python35\python -m venv c:\path\to\myenv
```

或者，如果您为Python安装配置了变量PATH和PATHEXT变量：

```
c:\>python -m venv c:\path\to\myenv
```

该命令（如果运行-h）将显示可用选项：

```
usage: venv [-h] [--system-site-packages] [--symlinks | --copies] [--clear]
            [--upgrade] [--without-pip]
```

```
            ENV_DIR [ENV_DIR ...]

Creates virtual Python environments in one or more target directories.

positional arguments:
  ENV_DIR               A directory to create the environment in.

optional arguments:
  -h, --help            show this help message and exit
  --system-site-packages
                        Give the virtual environment access to the system
                        site-packages dir.
  --symlinks            Try to use symlinks rather than copies, when symlinks
                        are not the default for the platform.
  --copies              Try to use copies rather than symlinks, even when
                        symlinks are the default for the platform.
  --clear               Delete the contents of the environment directory if it
                        already exists, before environment creation.
  --upgrade             Upgrade the environment directory to use this version
                        of Python, assuming Python has been upgraded in-place.
  --without-pip         Skips installing or upgrading pip in the virtual
                        environment (pip is bootstrapped by default)

Once an environment has been created, you may wish to activate it, e.g. by
sourcing an activate script in its bin directory.
```

*在版本3.4中更改：*默认情况下安装pip，添加--without-pip 和--copies 选项

*在版本3.4中更改：*在早期版本中，如果目标目录已经存在，则会引发错误，除非提供了--clear or --upgrade选项。

创建的pyvenv.cfg文件还包含 include-system-site-packages密钥，设置为true如果venv使用该--system-site-packages选项运行，false否则。

除非提供该--without-pip选项，ensurepip否则将被引导pip进入虚拟环境。

可以给出多条路径venv，在这种情况下，根据给定的选项，将在每条提供的路径上创建相同的虚拟环境。

一旦创建了虚拟环境，就可以在虚拟环境的二进制目录中使用脚本“激活”它。脚本的调用是特定于平台的：

平台	贝壳	激活虚拟环境的命令
POSIX	庆典/ zsh中	$ source <venv> / bin / activate
	鱼	$。<VENV> /bin/activate.fish
	CSH / tcsh的	$ source <venv> /bin/activate.csh
视窗	CMD.EXE	C：\> <venv> \ Scripts \ activate.bat
	电源外壳	PS C:\> <venv> \ Scripts \ Activate.ps1

你并不特别*需要*激活一个环境; 激活只是预先将虚拟环境中的二进制文件目录到您的路径，让“蟒蛇”调用虚拟环境的Python解释器，你可以在无需使用他们的完整路径运行安装脚本。但是，安装在虚拟环境中的所有脚本都应该可以运行而不激活它，并且可以自动运行虚拟环境的Python。

您可以通过在shell中键入“停用”来停用虚拟环境。确切的机制是特定于平台的：例如，Bash激活脚本定义了“停用”功能，而在Windows上，有一些独立的脚本调用deactivate.bat并Deactivate.ps1在创建虚拟环境时安装。

*3.4版新增功能：*fish和csh激活脚本。

注意: 虚拟环境是一个Python环境，Python安装到其中的Python解释器，库和脚本与安装在其他虚拟环境中的Python解释器，库和脚本是隔离的，并且（默认情况下）安装在“系统”Python中的任何库，即安装的作为您的操作系统的一部分。

虚拟环境是一个目录树，其中包含Python可执行文件和其他指示它是虚拟环境的文件。

常见的安装工具，如Setuptools和pip工作的预期与虚拟环境。换句话说，当虚拟环境处于活动状态时，他们将Python软件包安装到虚拟环境中，而不需要被明确地告知。

当虚拟环境是活动的（即，虚拟环境的Python解释正在运行），属性sys.prefix和sys.exec_prefix点到虚拟环境的基本目录，而sys.base_prefix并 sys.base_exec_prefix指向其用于创建虚拟环境中的非虚拟环境Python安装。如果虚拟环境是不活动的，则sys.prefix 是相同的 sys.base_prefix 并且 sys.exec_prefix 是相同的sys.base_exec_prefix（它们都指向到非虚拟环境Python安装）。

当虚拟环境处于活动状态时，任何更改安装路径的选项都将从所有distutils配置文件中忽略，以防止项目无意中安装在虚拟环境之外。

在命令外壳中工作时，用户可以通过activate在虚拟环境的可执行文件目录中运行脚本（准确的文件名与外壳相关）来使虚拟环境处于活动状态，该脚本将虚拟环境的可执行文件的目录预装到PATH正在运行的外壳的环境变量中。在其他情况下不需要激活虚拟环境 - 安装到虚拟环境中的脚本具有指向虚拟环境的Python解释器的“shebang”行。这意味着脚本将与该解释器一起运行，而不考虑其值PATH。在Windows上，如果您安装了用于Windows的Python启动程序，则支持“shebang”行处理（这已添加到3.3中的Python中 - 请参阅**PEP 397**了解更多详情）。因此，在Windows资源管理器窗口中双击已安装的脚本应该使用正确的解释器运行脚本，而不需要在其中引用其虚拟环境PATH。

28.3.2。API

上面描述的高级方法使用了一个简单的API，它为第三方虚拟环境创建者提供了根据他们的需求定制环境创建的机制EnvBuilder。

class venv.EnvBuilder（*system_site_packages = False*，*clear = False*，*symlinks = False*，*upgrade = False*，*with_pip = False*，*prompt = None* ）

该EnvBuilder班接受对实例以下关键字参数：

- system_site_packages- 一个布尔值，指示系统Python站点包应该可供环境使用（默认为False）。
- clear - 一个布尔值，如果为true，则会在创建环境之前删除任何现有目标目录的内容。
- symlinks- 一个布尔值，指示是否尝试符号链接Python二进制文件（以及任何必需的DLL或其他二进制文件，例如pythonw.exe），而不是复制。默认为TrueLinux和Unix

系统，但False在Windows上。

- upgrade- 一个布尔值，如果为true，则将用正在运行的Python升级现有环境 - 用于在Python就地升级（默认为False）时使用。
- with_pip - 一个布尔值，如果为true，则确保pip安装在虚拟环境中。这ensurepip与--default-pip选项一起使用。
- prompt- 在激活虚拟环境后使用的字符串（默认为None表示将使用环境的目录名称）。

*在版本3.4中更改：*添加了with_pip参数

*3.6版*新增功能：增加了prompt参数

第三方虚拟环境工具的创作者可以自由使用提供的EnvBuilder类作为基类。

返回的env-builder是一个具有以下方法的对象create：

create（*env_dir*）

此方法根据需要参数指定要包含虚拟环境的目标目录的路径（绝对或相对于当前目录）。该create方法将在指定的目录中创建环境，或引发适当的异常。

该类的create方法EnvBuilder说明了可用于子类定制的钩子：

```
def create(self, env_dir):
    """
    Create a virtualized Python environment in a directory.
    env_dir is the target directory to create an environment in.
    """
    env_dir = os.path.abspath(env_dir)
    context = self.ensure_directories(env_dir)
    self.create_configuration(context)
    self.setup_python(context)
    self.setup_scripts(context)
    self.post_setup(context)
```

每一种方法ensure_directories()，create_configuration()，setup_python()，setup_scripts()并且post_setup()可以被覆盖。

ensure_directories（*env_dir*）

创建环境目录和所有必需的目录，并返回一个上下文对象。这只是属性（如路径）的持有者，供其他方法使用。这些目录已被允许存在，只要其中一个clear或被upgrade指定为允许在现有环境目录上进行操作即可。

create_configuration（*上下文*）

pyvenv.cfg在环境中创建配置文件。

setup_python（*上下文*）

在环境中创建Python可执行文件（以及Windows下的DLL）的副本。在POSIX系统中，如果一个特定的可执行文件 python3.x使用，符号链接python和python3将创建指向该可执行文件，除非已存在具有这些名称的文件。

setup_scripts（*上下文*）

将适合该平台的激活脚本安装到虚拟环境中。

post_setup（*上下文*）

一种占位符方法，可以在第三方实现中重写，以在虚拟环境中预安装包或执行其他后创建步骤。

此外，EnvBuilder提供这种工具方法，可以从被称为setup_scripts()或post_setup()在子类中，以协助安装自定义脚本到虚拟环境中。

install_scripts（*上下文*，*路径*）

*路径*是应包含子目录“common”，“posix”，“nt”的目录的路径，每个目录都包含指向环境中bin目录的脚本。os.name经过一些文本替换占位符后，“common”的内容和相应的目录被复制：

- __VENV_DIR__ 被替换为环境目录的绝对路径。
- __VENV_NAME__ 被替换为环境名称（环境目录的最终路径段）。
- __VENV_PROMPT__ 被提示符替换（环境名称由括号括起来，并带有下面的空格）
- __VENV_BIN_NAME__被替换为bin目录的名称（bin或者Scripts）。
- __VENV_PYTHON__ 被替换为环境可执行文件的绝对路径。

允许目录存在（用于在现有环境正在升级时）。

还有一个模块级的便利功能：

venv.create（*env_dir*，*system_site_packages = False*，*clear = False*，*symlinks = False*，*with_pip = False* ）

EnvBuilder用给定的关键字参数创建一个参数，并create()用*env_dir*参数调用它的 方法。

*在版本3.4中更改：*添加了with_pip参数

28.3.3。延伸 的例子EnvBuilder

以下脚本展示了如何EnvBuilder通过实现将setuptools和pip安装到创建的虚拟环境中的子类来进行扩展：

```
import os
import os.path
from subprocess import Popen, PIPE
import sys
from threading import Thread
from urllib.parse import urlparse
from urllib.request import urlretrieve
import venv

class ExtendedEnvBuilder(venv.EnvBuilder):
    """
    This builder installs setuptools and pip so that you can pip or
    easy_install other packages into the created virtual environment.

    :param nodist: If True, setuptools and pip are not installed into the
                   created virtual environment.
    :param nopip: If True, pip is not installed into the created
                  virtual environment.
    :param progress: If setuptools or pip are installed, the progress of the
```

```
                    installation can be monitored by passing a progress
                    callable.  If specified, it is called with two
                    arguments: a string indicating some progress, and a
                    context indicating where the string is coming from.
                    The context argument can have one of three values:
                    'main', indicating that it is called from virtualize()
                    itself, and 'stdout' and 'stderr', which are obtained
                    by reading lines from the output streams of a subprocess
                    which is used to install the app.

                    If a callable is not specified, default progress
                    information is output to sys.stderr.
    """

    def __init__(self, *args, **kwargs):
        self.nodist = kwargs.pop('nodist', False)
        self.nopip = kwargs.pop('nopip', False)
        self.progress = kwargs.pop('progress', None)
        self.verbose = kwargs.pop('verbose', False)
        super().__init__(*args, **kwargs)

    def post_setup(self, context):
        """
        Set up any packages which need to be pre-installed into the
        virtual environment being created.

        :param context: The information for the virtual environment
                        creation request being processed.
        """
        os.environ['VIRTUAL_ENV'] = context.env_dir
        if not self.nodist:
            self.install_setuptools(context)
        # Can't install pip without setuptools
        if not self.nopip and not self.nodist:
            self.install_pip(context)

    def reader(self, stream, context):
        """
        Read lines from a subprocess' output stream and either pass to a progress
        callable (if specified) or write progress information to sys.stderr.
        """
        progress = self.progress
        while True:
            s = stream.readline()
            if not s:
                break
            if progress is not None:
                progress(s, context)
            else:
                if not self.verbose:
                    sys.stderr.write('.')
                else:
                    sys.stderr.write(s.decode('utf-8'))
                sys.stderr.flush()
        stream.close()

    def install_script(self, context, name, url):
        _, _, path, _, _, _ = urlparse(url)
```

```
        fn = os.path.split(path)[-1]
        binpath = context.bin_path
        distpath = os.path.join(binpath, fn)
        # Download script into the virtual environment's binaries folder
        urlretrieve(url, distpath)
        progress = self.progress
        if self.verbose:
            term = '\n'
        else:
            term = ''
        if progress is not None:
            progress('Installing %s ...%s' % (name, term), 'main')
        else:
            sys.stderr.write('Installing %s ...%s' % (name, term))
            sys.stderr.flush()
        # Install in the virtual environment
        args = [context.env_exe, fn]
        p = Popen(args, stdout=PIPE, stderr=PIPE, cwd=binpath)
        t1 = Thread(target=self.reader, args=(p.stdout, 'stdout'))
        t1.start()
        t2 = Thread(target=self.reader, args=(p.stderr, 'stderr'))
        t2.start()
        p.wait()
        t1.join()
        t2.join()
        if progress is not None:
            progress('done.', 'main')
        else:
            sys.stderr.write('done.\n')
        # Clean up - no longer needed
        os.unlink(distpath)

    def install_setuptools(self, context):
        """
        Install setuptools in the virtual environment.

        :param context: The information for the virtual environment
                        creation request being processed.
        """
        url = 'https://bitbucket.org/pypa/setuptools/downloads/ez_setup.py'
        self.install_script(context, 'setuptools', url)
        # clear up the setuptools archive which gets downloaded
        pred = lambda o: o.startswith('setuptools-') and o.endswith('.tar.gz')
        files = filter(pred, os.listdir(context.bin_path))
        for f in files:
            f = os.path.join(context.bin_path, f)
            os.unlink(f)

    def install_pip(self, context):
        """
        Install pip in the virtual environment.

        :param context: The information for the virtual environment
                        creation request being processed.
        """
        url = 'https://raw.github.com/pypa/pip/master/contrib/get-pip.py'
        self.install_script(context, 'pip', url)
```

```
def main(args=None):
    compatible = True
    if sys.version_info < (3, 3):
        compatible = False
    elif not hasattr(sys, 'base_prefix'):
        compatible = False
    if not compatible:
        raise ValueError('This script is only for use with '
                         'Python 3.3 or later')
    else:
        import argparse

        parser = argparse.ArgumentParser(prog=__name__,
                                         description='Creates virtual Python '
                                                     'environments in one or '
                                                     'more target '
                                                     'directories.')
        parser.add_argument('dirs', metavar='ENV_DIR', nargs='+',
                            help='A directory in which to create the
                                 'virtual environment.')
        parser.add_argument('--no-setuptools', default=False,
                            action='store_true', dest='nodist',
                            help="Don't install setuptools or pip in the "
                                 "virtual environment.")
        parser.add_argument('--no-pip', default=False,
                            action='store_true', dest='nopip',
                            help="Don't install pip in the virtual "
                                 "environment.")
        parser.add_argument('--system-site-packages', default=False,
                            action='store_true', dest='system_site',
                            help='Give the virtual environment access to the '
                                 'system site-packages dir.')
        if os.name == 'nt':
            use_symlinks = False
        else:
            use_symlinks = True
        parser.add_argument('--symlinks', default=use_symlinks,
                            action='store_true', dest='symlinks',
                            help='Try to use symlinks rather than copies, '
                                 'when symlinks are not the default for '
                                 'the platform.')
        parser.add_argument('--clear', default=False, action='store_true',
                            dest='clear', help='Delete the contents of the '
                                               'virtual environment '
                                               'directory if it already '
                                               'exists, before virtual '
                                               'environment creation.')
        parser.add_argument('--upgrade', default=False, action='store_true',
                            dest='upgrade', help='Upgrade the virtual '
                                                 'environment directory to '
                                                 'use this version of '
                                                 'Python, assuming Python '
                                                 'has been upgraded '
                                                 'in-place.')
        parser.add_argument('--verbose', default=False, action='store_true',
                            dest='verbose', help='Display the output '
                                                 'from the scripts which '
                                                 'install setuptools and pip.')
```

```
        options = parser.parse_args(args)
        if options.upgrade and options.clear:
            raise ValueError('you cannot supply --upgrade and --clear together.')
        builder = ExtendedEnvBuilder(system_site_packages=options.system_site,
                                     clear=options.clear,
                                     symlinks=options.symlinks,
                                     upgrade=options.upgrade,
                                     nodist=options.nodist,
                                     nopip=options.nopip,
                                     verbose=options.verbose)
        for d in options.dirs:
            builder.create(d)

if __name__ == '__main__':
    rc = 1
    try:
        main()
        rc = 0
    except Exception as e:
        print('Error: %s' % e, file=sys.stderr)
    sys.exit(rc)
```

这个脚本也可以在线下载。

28.4。zipapp- 管理可执行的python zip档案

3.5版本中的新功能。

源代码： Lib / zipapp.py

该模块提供了一些工具来管理创建包含Python代码的zip文件，这些代码可以 由Python解释器直接执行。该模块提供了 命令行界面和Python API。

28.4.1。基本示例

以下示例显示了如何使用命令行界面 从包含Python代码的目录创建可执行档案。运行时，存档将执行存档中main模块的功能myapp。

```
$ python -m zipapp myapp -m "myapp:main"
$ python myapp.pyz
<output from myapp>
```

28.4.2。命令行界面

当从命令行调用程序时，将使用以下形式：

```
$ python -m zipapp source [options]
```

如果*source*是一个目录，则会从*源*内容创建一个存档 。如果*source*是一个文件，它应该是一个档案，并且它将被复制到目标档案（或者如果指定了-info选项，它将显示其shebang行的内容）。

以下选项是可以理解的：

-o <output>, --output=<output>
将输出写入名为*output*的文件。如果未指定此选项，则输出文件名将与输入*源*相同，并.pyz添加扩展名。如果给出了明确的文件名，它将按原样使用（因此，.pyz如果需要，应该包含扩展名）。

如果*源*是归档文件，则必须指定输出文件名（在这种情况下，*输出*不能与*源文件*相同）。

-p <interpreter>, --python=<interpreter>
#!在指定*解释器*的档案中添加一行作为要运行的命令。另外，在POSIX上，使存档可执行。缺省情况是不写任何#!行，也不使文件成为可执行文件。

-m <mainfn>, --main=<mainfn>
将__main__.py文件写入执行*mainfn*的存档。所述 *mainfn*参数应具有的形式是“pkg.mod：FN”，其中“pkg.mod”是在归档中的包/模块，和“FN”是给定模块中的调用。该__main__.py文件将执行该可调用。

--main 复制档案时无法指定。

--info
显示嵌入到档案中的解释器，用于诊断目的。在这种情况下，任何其他选项都会被忽略，并且SOURCE必须是存档，而不是目录。

-h, --help
打印一条简短的使用信息并退出。

28.4.3。Python

该模块定义了两个便利功能：

zipapp.create_archive(*source*，*target* = *None*，*interpreter* = *None*，*main* = *None*)
从*源*创建应用程序存档。来源可以是以下任何一种：

- 目录的名称或pathlib.Path引用目录的对象，在这种情况下，将从该目录的内容创建新的应用程序归档。
- 现有应用程序档案文件的名称或pathlib.Path 引用这样一个文件的对象，在这种情况下，文件被复制到目标（修改它以反映为*解释器* 参数给出的值）。.pyz如果需要，文件名应包含扩展名。
- 打开文件对象以字节模式读取。文件的内容应该是应用程序归档文件，并假定文件对象位于归档的开始位置。

所述*目标*参数确定所生成存档将被写入：

- 如果它是文件或pathlb.Path对象的名称，则归档文件将写入该文件。
- 如果它是一个打开的文件对象，则归档文件将被写入该文件对象，该文件对象必须打开才能以字节模式写入。
- 如果目标被省略（或None ），则源必须是目录，目标将是与源相同名称的文件，并.pyz添加扩展名。

该*解释*参数指定与归档将执行Python解释器的名称。它在档案开始时被写为“shebang”行。在POSIX上，这将由操作系统解释，并在Windows上由Python启动程序处理。省略*解释器会*导致没有任何文本行被写入。如果指定了解释器，并且目标是文件名，则将设置目标文件的可执行位。

的*主要*参数指定将被用作归档主程序调用的名称。只有当源是一个目录，并且源__main__.py文件尚未包含时才能指定它 。的*主要*论点应采取的形式“pkg.module：调用”和存档将通过导入“pkg.module”，并执行给定的调用不带参数运行。如果源是一个目录并且不包含文件，那么忽略*main*是错误的__main__.py，否则生成的存档将不可执行。

如果为*源*或*目标*指定了文件对象，则在调用create_archive之后调用者有责任关闭它。

当复制现有的归档，只有提供的文件对象需要 read和readline ，或write方法。从目录创建档案时，如果目标是文件对象，则它将被传递给zipfile.ZipFile该类，并且必须提供该类所需的方法。

zipapp.get_interpreter(*档案*)

#!在归档开始处返回行中指定的解释器。如果没有#!线路，则返回None。该*归档文件*的参数可以是文件名或文件对象打开以字节模式下读取。它被认为是在档案的开始。

28.4.4。示例

将一个目录打包到一个存档中，然后运行它。

```
$ python -m zipapp myapp
$ python myapp.pyz
<output from myapp>
```

使用create_archive()函数可以完成相同的操作：

```
>>> import zipapp
>>> zipapp.create_archive('myapp.pyz', 'myapp')
```

要使应用程序在POSIX上直接可执行，请指定要使用的解释器。

```
$ python -m zipapp myapp -p "/usr/bin/env python"
$ ./myapp.pyz
<output from myapp>
```

要替换现有存档上的shebang行，请使用以下create_archive()函数创建修改后的存档：

```
>>> import zipapp
>>> zipapp.create_archive('old_archive.pyz', 'new_archive.pyz', '/usr/bin/python3')
```

要更新文件，使用BytesIO 对象在内存中进行替换，然后覆盖源。请注意，在覆盖文件时出现错误会导致原始文件丢失的风险。此代码不能防止此类错误，但生产代码应该这样做。另外，这种方法只适用于档案适合内存的情况：

```
>>> import zipapp
>>> import io
>>> temp = io.BytesIO()
>>> zipapp.create_archive('myapp.pyz', temp, '/usr/bin/python2')
>>> with open('myapp.pyz', 'wb') as f:
>>>     f.write(temp.getvalue())
```

28.4.5。指定解释器

请注意，如果您指定解释器并分发应用程序归档文件，则需要确保所使用的解释器是可移植的。适用于Windows的Python启动程序支持最常见的POSIX #!行形式，但还有其他问题需要考虑：

- 如果您使用“/ usr / bin / env python”（或其他形式的“python”命令，例如“/ usr / bin / python”），则需要考虑您的用户可能拥有Python 2或Python 3作为它们的默认值，并编写代码以在两个版本下工作。
- 如果您使用明确的版本，例如“/ usr / bin / env python3”，那么您的应用程序将不适用于没有该版本的用户。（如果你还没有使你的代码与Python 2兼容，这可能就是你想要的）。

- 没有办法说“python XY或更高版本”，所以要小心使用像“/ usr / bin / env python3.4”这样的确切版本，因为您需要为Python 3.5的用户更改您的shebang行，例如。

通常，您应该使用“/ usr / bin / env python2”或“/ usr / bin / env python3”，具体取决于您的代码是否为Python 2或Python 3编写。

28.4.6。使用zipapp创建独立应用程序

使用该zipapp模块，可以创建自包含的Python程序，这些程序可以分发给只需要在其系统上安装合适版本的Python的最终用户。这样做的关键是将应用程序的所有依赖关系与应用程序代码捆绑到归档中。

创建独立存档的步骤如下所示：

1. 正常情况下在目录中创建应用程序，以便您拥有myapp 包含__main__.py文件的目录以及任何支持的应用程序代码。

2. myapp使用pip 将所有应用程序的依赖项安装到目录中：

   ```
   $ python -m pip install -r requirements.txt --target myapp
   ```

   （假设你在一个requirements.txt 文件中有你的项目需求- 如果没有，你可以在pip命令行上手动列出依赖关系）。

3. 或者，删除.dist-info目录中由pip创建的 myapp目录。这些包含用于管理包的pip元数据，因为您不会再使用pip，所以它们不是必需的 - 尽管如果您离开它们将不会造成任何伤害。

4. 使用以下方式打包应用

   ```
   $ python -m zipapp -p "interpreter" myapp
   ```

这将生成一个独立的可执行文件，它可以在任何机器上运行，并提供适当的解释器。有关 详细信息，请参阅指定解释器。它可以作为单个文件发送给用户。

在Unix上，该myapp.pyz文件是可执行的。.pyz如果您喜欢“普通”命令名称，则可以重命名文件以删除扩展名。在Windows中，myapp.pyz[w]文件是由如下事实可执行的Python解释器注册.pyz和.pyzw安装时的文件扩展名。

28.4.6.1。制作一个Windows可执行文件

在Windows上，.pyz扩展的注册是可选的，此外，某些地方不能“透明地”识别注册的扩展（最简单的例子是 subprocess.run(['myapp'])找不到您的应用程序 - 您需要明确指定扩展）。

因此，在Windows上，通常最好从zipapp创建一个可执行文件。虽然它确实需要C编译器，但这相对容易。基本的方法依赖于zipfiles可以具有任意数据前缀，并且Windows exe文件可以附加任意数据。因此，通过创建合适的启动器并将.pyz文件粘贴到文件的最后，最终生成一个运行应用程序的单个文件可执行文件。

一个合适的发射器可以像下面这样简单：

```
#define Py_LIMITED_API 1
#include "Python.h"

#define WIN32_LEAN_AND_MEAN
#include <windows.h>

#ifdef WINDOWS
int WINAPI wWinMain(
    HINSTANCE hInstance,      /* handle to current instance */
    HINSTANCE hPrevInstance,  /* handle to previous instance */
    LPWSTR lpCmdLine,         /* pointer to command line */
    int nCmdShow              /* show state of window */
)
#else
int wmain()
#endif
{
    wchar_t **myargv = _alloca((__argc + 1) * sizeof(wchar_t*));
    myargv[0] = __wargv[0];
    memcpy(myargv + 1, __wargv, __argc * sizeof(wchar_t *));
    return Py_Main(__argc+1, myargv);
}
```

如果您定义了WINDOWS预处理器符号，则会生成一个GUI可执行文件，如果没有它，则会生成一个控制台可执行文件。

要编译可执行文件，可以使用标准的MSVC命令行工具，也可以利用distutils知道如何编译Python源代码的事实：

```
>>> from distutils.ccompiler import new_compiler
>>> import distutils.sysconfig
>>> import sys
>>> import os
>>> from pathlib import Path

>>> def compile(src):
>>>     src = Path(src)
>>>     cc = new_compiler()
>>>     exe = src.stem
>>>     cc.add_include_dir(distutils.sysconfig.get_python_inc())
>>>     cc.add_library_dir(os.path.join(sys.base_exec_prefix, 'libs'))
>>>     # First the CLI executable
>>>     objs = cc.compile([str(src)])
>>>     cc.link_executable(objs, exe)
>>>     # Now the GUI executable
>>>     cc.define_macro('WINDOWS')
>>>     objs = cc.compile([str(src)])
>>>     cc.link_executable(objs, exe + 'w')

>>> if __name__ == "__main__":
>>>     compile("zastub.c")
```

由此产生的启动程序使用“有限ABI”，因此它将随任何版本的Python 3.x一起运行。它只需要Python（ python3.dll ）在用户的身上PATH。

对于完全独立的发行版，您可以随附您的应用程序来分发启动器，并与Python“嵌入式”发行版捆绑在一起。这将在具有适当架构（32位或64位）的任何PC上运行。

28.4.6.2。警告

将应用程序捆绑到单个文件中的过程存在一些限制。在大多数情况下，即使不是全部，也可以在不需要对应用程序进行重大更改的情况下解决。

1. 如果您的应用程序依赖于包含C扩展名的包，则该包不能从zip文件运行（这是操作系统限制，因为可执行代码必须存在于OS加载程序的文件系统中才能加载）。在这种情况下，您可以从zip文件中排除该依赖项，并且要求您的用户安装它，或者将其与zip文件一起发布，并向您的代码添加代码以__main__.py包含包含解压缩模块的目录sys.path。在这种情况下，您需要确保为目标体系结构提供适当的二进制文件（并且可能会sys.path根据用户的机器选择在运行时添加的正确版本）。
2. 如果您按照上述步骤发布Windows可执行文件，则需要确保用户具有python3.dllPATH（这不是安装程序的默认行为），或者应该将应用程序与嵌入式发行版捆绑在一起。
3. 上面的推荐启动器使用Python嵌入API。这意味着在你的应用程序中，sys.executable将是你的应用程序，而*不是*传统的Python解释器。您的代码及其依赖性需要为这种可能性做好准备。例如，如果您的应用程序使用该 multiprocessing 模块，则需要调用multiprocessing.set_executable()以使模块知道在哪里找到标准Python解释器。

28.4.7。Python Zip应用程序存档格式

__main__.py从2.6版开始，Python就能够执行包含文件的zip文件。为了由Python执行，应用程序归档文件必须是标准的压缩文件，其中包含__main__.py将作为应用程序入口点运行的文件。像任何Python脚本一样，脚本的父级（在本例中为zip文件）将被放置 sys.path ，因此可以从zip文件导入更多模块。

zip文件格式允许将任意数据预置为zip文件。zip应用程序格式使用此功能将一个标准POSIX“shebang”行添加到文件（ #!/path/to/interpreter ）中。

形式上，Python zip应用程序格式是：

1. 一个可选的shebang行，包含字符b'#!'后跟一个解释器名称，然后是一个换行符（ b'\n' ）字符。解释器名称可以是OS“shebang”处理可接受的任何东西，也可以是Windows上的Python启动器。解释器应该在Windows上以UTF-8编码，sys.getfilesystemencoding()在POSIX上编码。
2. 标准zipfile数据，由zipfile模块生成。zipfile内容*必须*包含一个名为__main__.py（它必须位于zipfile的“根目录”中 - 即它不能位于子目录中）的文件。zipfile数据可以被压缩或解压缩。

如果应用程序归档文件具有shebang行，那么它可能会在POSIX系统上设置可执行位，以允许它直接执行。

没有要求使用本模块中的工具来创建应用程序归档 - 该模块很方便，但以任何方式创建的上述格式的归档均可供Python使用。

29. Python运行时服务

本章描述的模块提供了与Python解释器及其与环境的交互相关的各种服务。这里有一个概述：

29.1。 sys- 系统特定的参数和功能

该模块提供对由解释器使用或维护的一些变量以及与解释器强烈交互的函数的访问。它始终可用。

sys. abiflags

在使用标准configure 脚本构建Python的POSIX系统上，它包含由指定的ABI标志PEP 3149。

3.2版本中的新功能

sys. argv

传递给Python脚本的命令行参数列表。argv[0]是脚本名称（操作系统依赖于它是否是完整的路径名）。如果命令是使用-c解释器的命令行选项执行的，argv[0]则设置为字符串'-c'。如果没有脚本名称传递给Python解释器，argv[0]则为空字符串。

要遍历标准输入或命令行上给出的文件列表，请参阅fileinput模块。

sys. base_exec_prefix

在Python启动时，在site.py运行之前设置为与之相同的值 exec_prefix。如果不在虚拟环境中运行 ，则值将保持不变; 如果 site.py发现虚拟环境正在使用，则值prefix和值exec_prefix将被更改为指向虚拟环境，base_prefix而且 base_exec_prefix将保持指向基本Python安装（虚拟环境创建者）。

3.3版本的新功能

sys. base_prefix

在Python启动时，在site.py运行之前设置为与之相同的值 prefix。如果不在虚拟环境中运行，则值将保持不变; 如果site.py发现虚拟环境正在使用，则值prefix和值exec_prefix将被更改为指向虚拟环境，base_prefix而且 base_exec_prefix将保持指向基本Python安装（虚拟环境创建者）。

3.3版本的新功能

sys. byteorder

本地字节顺序的指示符。这将'big'在大端（最重要的字节在先）平台和'little'小端（最不重要的字节在先）平台上具有价值。

sys. builtin_module_names

一个字符串元组，给出了编译到这个Python解释器中的所有模块的名称。（此信息不能以其他方式提供 - modules.keys()仅列出导入的模块。）

sys. call_tracing（*func*，*args* ）

呼叫func(*args)，同时启用跟踪。跟踪状态被保存，并在之后恢复。这旨在从检查点的调试器调用，以递归调试其他代码。

sys. copyright

包含与Python解释器相关的版权的字符串。

sys._clear_type_cache()
清除内部类型缓存。类型缓存用于加速属性和方法查找。*仅*在参考泄漏调试期间使用该函数删除不必要的引用。

该功能只能用于内部和专门用途。

sys._current_frames()
返回一个字典，该函数被调用时，将每个线程的标识符映射到当前在该线程中处于活动状态的最上面的堆栈帧。请注意，traceback模块中的函数可以根据给定的帧构建调用堆栈。

这对于调试死锁非常有用：该函数不需要死锁线程的合作，只要这些线程的调用堆栈保持死锁状态，它们就会被冻结。在调用代码检查帧时，为非死锁线程返回的帧可能与该线程的当前活动无关。

该功能只能用于内部和专门用途。

sys._debugmallocstats()
打印低级别信息以了解CPython内存分配器的状态。

如果Python配置了-with-pydebug，它还会执行一些昂贵的内部一致性检查。

3.3版本的新功能

CPython实现细节：该函数特定于CPython。确切的输出格式在这里没有定义，并且可能会改变。

sys.dllhandle
整数指定Python DLL的句柄。可用性：Windows。

sys.displayhook(*价值*)
如果*值*是不是None，这个函数打印repr(value)到 sys.stdout，并保存*价值*在builtins._。如果repr(value)不能sys.stdout.encoding用sys.stdout.errors错误处理程序（可能是'strict'）进行编码，则sys.stdout.encoding使用'backslashreplace'错误处理程序对其进行编码。

sys.displayhook被称为评估 在交互式Python会话中输入的表达式的结果。这些值的显示可以通过分配另一个单参数函数来定制sys.displayhook。

伪代码：

```
def displayhook(value):
    if value is None:
        return
    # Set '_' to None to avoid recursion
    builtins._ = None
    text = repr(value)
    try:
        sys.stdout.write(text)
    except UnicodeEncodeError:
```

```
        bytes = text.encode(sys.stdout.encoding, 'backslashreplace')
        if hasattr(sys.stdout, 'buffer'):
            sys.stdout.buffer.write(bytes)
        else:
            text = bytes.decode(sys.stdout.encoding, 'strict')
            sys.stdout.write(text)
    sys.stdout.write("\n")
    builtins._ = value
```

*在版本3.2中更改：*使用'backslashreplace'错误处理程序UnicodeEncodeError。

sys.dont_write_bytecode

如果这是真的，Python将不会尝试.pyc在源模块的导入上编写文件。该值最初设置为True或 False取决于-B命令行选项和 PYTHONDONTWRITEBYTECODE 环境变量，但您可以自己设置它来控制字节码文件生成。

sys.excepthook（*类型*，*值*，*追溯*）

这个函数打印出一个给定的回溯和例外sys.stderr。

当引发异常并且未捕获到异常时，解释器将调用 sys.excepthook三个参数，即异常类，异常实例和跟踪对象。在交互式会话中，这恰好在控制返回提示之前发生; 在一个Python程序中，这个就在程序退出之前发生。可以通过为其分配另外三个参数的函数来自定义这些顶级异常的处理sys.excepthook。

sys.__displayhook__

sys.__excepthook__

这些对象包含的原始值displayhook ，并excepthook 在程序的开始。它们被保存起来，displayhook并且 excepthook可以在它们碰巧被破碎物体取代的情况下被恢复。

sys.exc_info（ ）

这个函数返回一个包含三个值的元组，该元组提供有关当前正在处理的异常的信息。返回的信息既针对当前线程也针对当前堆栈帧。如果当前堆栈帧没有处理异常，则从调用堆栈帧或其调用者那里获取信息，直到找到处理异常的堆栈帧为止。这里，“处理异常”定义为“执行except子句”。对于任何堆栈帧，只能访问有关当前处理的异常的信息。

如果堆栈中的任何位置没有异常处理，None则返回包含三个值的元组。否则，返回的值是。它们的含义是：*type*获取正在处理的异常的类型（的一个子类）；*值*获取异常实例（异常类型的一个实例）；*回溯*得到一个回溯对象（参见参考手册），它在最初发生异常的地方封装了调用堆栈。(type, value, traceback)BaseException

sys.exec_prefix

一个字符串，其中给出了特定于站点的目录前缀，其中安装了与平台相关的Python文件; 默认情况下，这也是'/usr/local'。这可以在构建时使用**configure**脚本的--exec-prefix参数 进行设置。具体而言，所有配置文件（例如 头文件）都安装在该目录中 ，并且共享库模块安装在其中 ，例如 ，*XY* 是 Python 的版本号 。pyconfig.h*exec_prefix*/lib/python*X.Y*/config*exec_prefix*/lib/python*X.Y*/lib-dynload3.2

注意： 如果虚拟环境有效，则此值将更改为site.py指向虚拟环境。Python安装的值仍然可用，通过 base_exec_prefix。

sys.**executable**

一个字符串，给出Python解释器的可执行二进制文件的绝对路径，在这种情况下是合理的。如果Python无法检索到其可执行文件的真实路径，sys.executable将是一个空字符串或None。

sys.**exit**([*arg*])

从Python退出。这是通过引发SystemExit 异常来实现的，所以最终语句的子句指定的清理操作try 得到遵守，并且可以在外层拦截退出尝试。

可选参数*arg*可以是一个给出退出状态的整数（默认为零）或其他类型的对象。如果它是一个整数，零被认为是“成功终止”，并且任何非零值被shell等认为是“异常终止”。大多数系统要求它在0-127范围内，否则会产生未定义的结果。一些系统具有为特定退出代码分配特定含义的惯例，但这些通常是欠发达的; Unix程序通常使用2作为命令行语法错误，1使用其他类型的错误。如果传递另一种类型的对象，None则相当于传递零，并且打印其他任何对象stderr并产生退出代码1.特别是，sys.exit("some error message") 是发生错误时快速退出程序的一种方法。

由于exit()最终“唯一”引发了一个异常，它只会在主线程调用时退出进程，并且异常不会被拦截。

*版本3.6中更改：*如果在Python解释程序捕获后发生了清理错误SystemExit（例如标准流中的缓冲数据刷新错误），则退出状态将更改为120。

sys.**flags**

该结构序列 *标志*暴露的命令行标志的状态。属性是只读的。

属性	旗
debug	-d
inspect	-i
interactive	-i
optimize	-O 要么 -OO
dont_write_bytecode	-B
no_user_site	-s
no_site	-S
ignore_environment	-E
verbose	-v
bytes_warning	-b
quiet	-q
hash_randomization	-R

*在版本3.2中更改：*添加quiet新-q标志的属性。

*在新版本3.2.3：*该hash_randomization属性。

*在版本3.3中更改：*删除过时的division_warning属性。

sys.**float_info**

一个保存关于浮点类型信息的结构序列。它包含有关精度和内部表示的低级信息。这些值对应float.h于在'C'编程语言的标准头文件中定义的各种浮点常量; 有关详细信息，请参见1999 ISO / IEC C标准[C99]的第5.2.4.2.2节 “浮动类型的特性”。

属性	float.h宏	说明
epsilon	DBL_EPSILON	1和大于1的最小值之间的差值可以表示为浮点数
dig	DBL_DIG	浮点数可以忠实地表示的最大小数位数; 见下文
mant_dig	DBL_MANT_DIG	float precision：浮点数的基数radix 位数
max	DBL_MAX	最大可表示的有限浮点数
max_exp	DBL_MAX_EXP	最大整数e，这radix**(e-1)是一个可表示的有限浮点数
max_10_exp	DBL_MAX_10_EXP	最大整数e，使其10**e处于可表示的有限浮点范围内
min	DBL_MIN	最小正归一化浮点数
min_exp	DBL_MIN_EXP	最小整数e，这radix**(e-1)是一个标准化的浮点数
min_10_exp	DBL_MIN_10_EXP	最小整数e，这10**e是一个标准化的浮点数
radix	FLT_RADIX	指数表示的基数
rounds	FLT_ROUNDS	整数常量表示用于算术运算的舍入模式。这反映了解释器启动时系统FLT_ROUNDS宏的价值。有关可能值及其含义的解释，请参阅C99标准的5.2.4.2.2节。

该属性sys.float_info.dig需要进一步解释。如果 s 任何字符串表示具有最高sys.float_info.dig有效数字的十进制 数字，则转换s为浮点数然后再返回将返回表示相同十进制值的字符串：

```
>>> import sys
>>> sys.float_info.dig
15
>>> s = '3.14159265358979'    # decimal string with 15 significant digits
>>> format(float(s), '.15g')  # convert to float and back -> same value
'3.14159265358979'
```

但对于超过sys.float_info.dig有效数字的字符串，情况并非总是如此：

```
>>> s = '9876543211234567'    # 16 significant digits is too many!
>>> format(float(s), '.16g')  # conversion changes value
'9876543211234568'
```

sys.float_repr_style

一个字符串，指示repr()函数如何为浮动操作。如果字符串具有值，'short'那么对于有限的浮点数x，则repr(x)旨在生成具有该属性的短字符串。这是Python 3.1和更高版本中的常见行为。否则，它的价值 和行为方式与3.1版之前的Python版本相同。float(repr(x)) == xfloat_repr_style'legacy'repr(x)

版本3.1中的新功能。

sys.getallocatedblocks()

不管大小如何，都返回当前由解释器分配的内存块数量。此功能主要用于跟踪和调试内存泄漏。由于解释器的内部缓存，结果可能因呼叫而异；你可能需要打电话_clear_type_cache()并gc.collect()获得更可预测的结果。

如果Python构建或实现无法合理计算此信息，getallocatedblocks()则允许返回0。

3.4版新增功能

sys.getcheckinterval()

返回解释器的"检查间隔"; 见setcheckinterval()。

*自3.2版弃用：*getswitchinterval()代之以使用。

sys.getdefaultencoding()

返回Unicode实现使用的当前默认字符串编码的名称。

sys.getdlopenflags()

返回用于dlopen()调用的标志的当前值 。标志值的符号名称可以在os模块中找到（RTLD_xxx常量，例如 os.RTLD_LAZY ）。可用性：Unix。

sys.getfilesystemencoding()

返回用于在Unicode文件名和字节文件名之间转换的编码名称。为了获得最佳兼容性，str应该用于所有情况下的文件名，尽管也支持将文件名表示为字节。接受或返回文件名的函数应该支持str或字节，并在内部转换为系统的首选表示。

该编码始终是ASCII兼容的。

os.fsencode()并os.fsdecode()应该用来确保使用正确的编码和错误模式。

- 在Mac OS X上，编码是'utf-8'。
- 在Unix上，编码是区域设置编码。
- 在Windows上，编码可能是'utf-8'或者'mbcs'，取决于用户配置。

*在版本3.2中更改：*getfilesystemencoding()结果不再可用None。

在版本3.6中更改： Windows不再保证返回'mbcs'。看到**PEP 529**和_enablelegacywindowsfsencoding()更多信息。

sys.getfilesystemencodeerrors()

返回用于在Unicode文件名和字节文件名之间转换的错误模式的名称。编码名称从中返回getfilesystemencoding()。

os.fsencode()并os.fsdecode()应该用来确保使用正确的编码和错误模式。

3.6版本中的新功能。

sys.getrefcount(*object*)

返回*对象*的引用计数。返回的计数通常比您预期的要高，因为它包含（临时）参考作为参数getrefcount()。

sys.getrecursionlimit()

返回递归限制的当前值，即Python解释器堆栈的最大深度。此限制可防止无限递归导致C堆栈溢出并导致Python崩溃。它可以通过设置 setrecursionlimit()。

sys.getsizeof(*object*[, *default*])

以字节为单位返回对象的大小。该对象可以是任何类型的对象。所有内置对象都会返回正确的结果，但这不一定适用于第三方扩展，因为它是特定于实现的。

只考虑直接归因于该对象的内存消耗，而不考虑其引用的对象的内存消耗。

如果给定，如果对象不提供检索大小的方法，将返回*默认值*。否则TypeError会被提出。

getsizeof()调用对象的__sizeof__方法并在垃圾收集器管理该对象时添加额外的垃圾回收器开销。

请参阅递归sizeof recipe ，以getsizeof()递归方式查找容器及其所有内容的大小。

sys.getswitchinterval()

返回解释器的“线程切换间隔”; 见 setswitchinterval()。

3.2版本中的新功能

sys._getframe([*深*])

从调用堆栈中返回一个框架对象。如果给出了可选的整数*深度*，则将多个调用的帧对象返回到堆栈顶部以下。如果这比调用堆栈更深，ValueError则会提升。*深度*的默认值为零，返回调用堆栈顶部的帧。

CPython实现细节：该函数仅用于内部和专用目的。不能保证在Python的所有实现中都存在。

sys.getprofile()

获取由设置的分析器功能setprofile()。

sys.gettrace()

获取由设置的跟踪功能settrace()。

CPython实现细节：该gettrace()函数仅用于实现调试器，分析器，覆盖工具等。它的行为是实现平台的一部分，而不是语言定义的一部分，因此可能不适用于所有Python实现。

sys.getwindowsversion()

返回描述当前正在运行的Windows版本的命名元组。所命名的元素包括*major*，*minor*，*build*，*platform*，*service_pack*，*service_pack_minor*，*service_pack_major*，*suite_mask*，*product_type*和*platform_version*。*service_pack*包含一个字符串，*platform_version*是一个三元组，其他所有值都是整数。组件也可以按名称访问，所以`sys.getwindowsversion()[0]`相当于`sys.getwindowsversion().major`。为了与先前版本兼容，只有前5个元素可通过索引检索。

*平台*将会。`2 (VER_PLATFORM_WIN32_NT)`

*product_type*可能是以下值之一：

不变	含义
`1 (VER_NT_WORKSTATION)`	该系统是一个工作站。
`2 (VER_NT_DOMAIN_CONTROLLER)`	该系统是一个域控制器。
`3 (VER_NT_SERVER)`	该系统是一个服务器，但不是域控制器。

此函数包装Win32 `GetVersionEx()`函数; 请参阅Microsoft文档`OSVERSIONINFOEX()`以获取有关这些字段的更多信息。

*platform_version*返回当前操作系统的准确主要版本，次要版本和内部版本号，而不是为进程模拟的版本。它旨在用于日志记录而不是用于特征检测。

可用性：Windows。

*在版本3.2中更改：*更改为命名元组并添加了*service_pack_minor*，*service_pack_major*，*suite_mask*和*product_type*。

*在版本3.6中更改：*添加了*platform_version*

sys.**get_asyncgen_hooks**()

返回一个*asyncgen_hooks*对象，该对象类似于*（firstiter，finalizer）*`namedtuple`形式，其中*firstiter*和*finalizer*需要是一个或多个函数，它们将异步生成器迭代器作为参数，并用于计划异步生成器的完成通过事件循环。`None`

*3.6版新增功能：*请参阅**PEP 525**更多细节。

> **注意：** 这个功能是临时添加的（请参阅 详情请参阅 **PEP 411。** ）

sys.**get_coroutine_wrapper**()

返回`None`或由...设置的包装器`set_coroutine_wrapper()`。

*3.5版本中的新功能：*请参阅**PEP 492**了解更多详情。

> **注意：** 这个功能是临时添加的（请参阅 **PEP 411** ）。仅用于调试目的。

sys.**hash_info**

一个结构顺序给数字散列执行参数。有关数字类型散列的更多详细信息，请参阅 数字类型的散列。

属性	说明
width	用于散列值的位宽
modulus	用于数字哈希方案的素数模数P.
inf	哈希值返回正无穷大
nan	哈希值返回一个南
imag	乘数用于复数的虚部
algorithm	str，字节和内存视图散列算法的名称
hash_bits	哈希算法的内部输出大小
seed_bits	散列算法的种子密钥的大小

3.2版本中的新功能

*在版本3.4中进行了更改：*添加了*算法*，*hash_bits*和*seed_bits*

sys.hexversion

版本号编码为单个整数。每个版本都有保证，包括适当支持非生产版本。例如，要测试Python解释器的版本是否至少为1.5.2，请使用：

```
if sys.hexversion >= 0x010502F0:
    # use some advanced feature
    ...
else:
    # use an alternative implementation or warn the user
    ...
```

这被称为，hexversion因为只有在将其传递给内置hex()函数时才会看起来很有意义。该结构体序列 sys.version_info可被用于相同信息的一个更人性化的编码。

更多的细节hexversion可以在API和ABI版本控制中找到。

sys.implementation

包含有关当前正在运行的Python解释器的实现信息的对象。所有Python实现中都需要以下属性。

*name*是实现的标识符，例如'cpython'。实际的字符串由Python实现定义，但保证为小写。

*版本*是一个命名的元组，格式与 sys.version_info。它代表了Python *实现*的版本。这与当前正在运行的解释器符合的特定版本的Python *语言*具有不同的含义，它sys.version_info表示。例如，对于PyPy 1.8 sys.implementation.version可能，而会。对于CPython，它们具有相同的价值，因为它是参考实现。sys.version_info(1, 8, 0, 'final', 0)sys.version_infosys.version_info(2, 7, 2, 'final', 0)

*hexversion*是十六进制格式的实现版本，例如 sys.hexversion。

*cache_tag*是导入机器在缓存模块的文件名中使用的标签。按照惯例，它将是实现名称和版本的组合，例如'cpython-33'。但是，如果合适，Python实现可能会使用其他值。如果cache_tag设置为None，则表示应禁用模块缓存。

sys.implementation可能包含特定于Python实现的其他属性。这些非标准属性必须以下划线开头，这里不再描述。无论其内容如何，sys.implementation在解释器运行期间以及实施版本之间都不会改变。（但是，它可能会在Python语言版本之间改变。）请参阅**PEP 421**了解更多信息。

3.3版本的新功能

sys.int_info

一个结构序列，保存有关Python整数内部表示的信息。属性是只读的。

属性	说明
bits_per_digit	每位数字中保存的位数。Python整数存储在内部的基地2**int_info.bits_per_digit
sizeof_digit	用于表示数字的C类型的字节大小

版本3.1中的新功能。

sys.__interactivehook__

当该属性存在时，当解释器以交互模式启动时，会自动调用其值（不带参数）。这是在完成之后完成的PYTHONSTARTUP文件被读取，以便您可以在此处设置该钩子。该site模块 设置此。

3.4版新增功能

sys.intern(*字符串*)

在“interned”字符串的表格中输入*字符串*并返回字符串 - *字符串*本身或副本。实习字符串对于在字典查找中获得一些性能很有用 - 如果字典中的密钥被实施并且查找密钥被实施，则可以通过指针比较而不是字符串比较来完成关键比较（哈希后）。通常情况下，Python程序中使用的名称会自动实现，并且用于保存模块，类或实例属性的字典具有实际的密钥。

Interned字符串不是不朽的; 你必须保持对intern()周围回报价值的参考才能从中受益。

sys.is_finalizing()

True如果Python解释器正在 关闭，False则返回，否则返回。

3.5版本中的新功能。

sys.last_type
sys.last_value
sys.last_traceback

这三个变量并不总是被定义的; 它们在未处理异常时设置，并且解释器打印错误消息和堆栈回溯。它们的预期用途是允许交互式用户导入调试器模块并进行事后调试，而不必重新执行导致错误的命令。（典型用法是输入验尸调试器; 有关详细信息，请参阅模块。）import pdb; pdb.pm() pdb

变量的含义与exc_info()上面返回值的含义相同 。

sys.maxsize

给出最大值的整数可以是一个类型的变量Py_ssize_t。它通常位于32位平台和64位平台上。2**31 - 12**63 - 1

sys.maxunicode

一个给出最大Unicode码位值的整数，即1114111（0x10FFFF十六进制）。

*版本3.3中更改：*之前**PEP 393**，sys.maxunicode曾经是0xFFFF 或者0x10FFFF，取决于指定Unicode字符是否存储为UCS-2或UCS-4的配置选项。

sys.meta_path

元路径查找器对象的列表，find_spec()调用其 方法以查看其中一个对象是否可以找到要导入的模块。该 find_spec()方法至少调用正在导入的模块的绝对名称。如果要导入的模块包含在包中，则父包的__path__ 属性作为第二个参数传入。该方法返回一个 模块规格，或者None如果找不到该模块。

也可以看看:

importlib.abc.MetaPathFinder
抽象基类定义上的查找器对象的接口 meta_path。

importlib.machinery.ModuleSpec
find_spec()应该返回实例的具体类 。

*版本3.4中已更改：*模块规范是在Python 3.4中引入的，由 **PEP 451**。早期版本的Python寻找一种称为的方法 find_module()。如果meta_path条目没有find_spec()方法，这仍称为后备。

sys.modules

这是一个将模块名称映射到已经加载的模块的字典。这可以被操纵来强制重新加载模块和其他技巧。但是，替换字典并不一定按预期工作，并且从字典中删除基本项目可能会导致Python失败。

sys.path

指定模块搜索路径的字符串列表。从环境变量初始化PYTHONPATH，加上一个依赖于安装的默认值。

正如在程序启动时初始化的，这个列表的第一项path[0]是包含用于调用Python解释器的脚本的目录。如果脚本目录不可用（例如，如果解释器是交互式调用的，或者脚本是从标准输入中读取的），path[0]则为空字符串，它指示Python首先在当前目录中搜索模块。请注意，脚本目录 插入作为结果插入的条目*之前*PYTHONPATH。

一个程序可以根据自己的目的自由修改这个列表。只应添加字符串和字节sys.path; 导入时忽略所有其他数据类型。

也可以看看: 模块site介绍如何使用.pth文件进行扩展 sys.path。

sys.path_hooks

可以使用路径参数尝试为路径创建查找器的可调用列表 。如果可以创建查找器，则将由可调用对象返回，否则提升ImportError。

最初在中指定 **PEP 302**。

sys.path_importer_cache

作为查找器对象缓存的字典。键是已传递到的路径， sys.path_hooks值是找到的查找器。如果路径是有效的文件系统路径，但未找到查找器， sys.path_hooks则会None存储路径。

最初在中指定 **PEP 302**。

*在版本3.3中更改：*None存储，而不是imp.NullImporter找不到查找器时。

sys.platform

例如，该字符串包含一个平台标识符，可用于将平台特定的组件附加到该平台标识符sys.path。

对于Unix系统，除了在Linux上，这是因为返回由小写的操作系统名称与版本的第一部分由返回 追加，例如或者，*在当时的Python建时*。除非您想测试特定的系统版本，否则建议使用以下习惯用法：uname -suname -r'sunos5''freebsd8'

```
if sys.platform.startswith('freebsd'):
    # FreeBSD-specific code here...
elif sys.platform.startswith('linux'):
    # Linux-specific code here...
```

对于其他系统，其值为：

系统	platform **值**
Linux的	'linux'
视窗	'win32'
在Windows / Cygwin的	'cygwin'
Mac OS X	'darwin'

*版本3.3中更改：*在Linux上，sys.platform不再包含主要版本。它总是'linux'，而不是'linux2'或'linux3'。由于较早的Python版本包含版本号，因此建议始终使用startswith上面提供的习惯用法。

也可以看看： os.name具有更粗糙的粒度。 os.uname()给出系统相关的版本信息。该platform模块提供了系统身份的详细检查。

sys.prefix

一个字符串，它给出了特定于站点的目录前缀，其中安装了与平台无关的Python文件; 默认情况下，这是字符串'/usr/local'。这可以在构建时使用**configure**脚本的--prefix 参数进行设置。例如，Python库模块的主要集合安装在目录中， 而独立于平台的头文件（除

except 之 外 ） 都 存 储 在 该 目 录 中 ， 其 中 *XY* 是 Python 的 版 本 号 。 *prefix*/lib/python*X.Y*pyconfig.h *prefix*/include/python*X.Y*3.2

> **注意：** 如果虚拟环境有效，则此值将更改为site.py指向虚拟环境。Python安装的值仍然可用，通过base_prefix。

sys.ps1

sys.ps2

指定解释器的主要和次要提示的字符串。这些仅在解释器处于交互模式时才被定义。他们在这种情况下的初始值是和。如果将非字符串对象分配给任一变量，则每次解释器准备读取新的交互式命令时都会对其进行重新评估; 这可以用来实现动态提示。'>>> ''... ' str()

sys.setcheckinterval（*间隔*）

设置解释器的“检查间隔”。此整数值决定解释器检查周期性事件的频率，例如线程切换和信号处理程序。默认值是100，这意味着检查每100个Python虚拟指令执行。将其设置为更大的值可能会增加使用线程的程序的性能。将其设置为值<=0会检查每条虚拟指令，从而最大限度地提高响应性和开销。

*自3.2版弃用：*此功能不再有效果，因为线程切换和异步任务的内部逻辑已被重写。setswitchinterval()改为使用 。

sys.setdlopenflags（*n* ）

设置解释器用于dlopen()调用的标志，例如解释器何时加载扩展模块。除此之外，这将在导入模块时启用对符号的惰性解析（如果调用为） sys.setdlopenflags(0)。要通过扩展模块共享符号，请拨打电话 sys.setdlopenflags(os.RTLD_GLOBAL)。标志值的符号名称可以在os模块中找到（ RTLD_xxx常量，例如 os.RTLD_LAZY ）。

可用性：Unix。

sys.setprofile（*profilefunc* ）

设置系统的配置文件功能，它允许您在Python中实现Python源代码分析器。有关Python分析器的更多信息，请参阅Python Profiler一章。系统的配置文件函数与系统的跟踪函数类似（请参阅参考资料settrace() ），但是它会被调用不同的事件，例如，不会为每个已执行的代码行调用它（仅在调用和返回时调用，但会报告返回事件即使设置了例外）。该函数是线程特定的，但探查器没有办法知道线程之间的上下文切换，所以在多线程的情况下使用它是没有意义的。而且，它的返回值没有被使用，所以它可以简单地返回None。

配置文件函数应该有三个参数：*frame*，*event*和 *arg*。*帧*是当前的堆栈帧。 *事件*是一个字符串：'call'， 'return'，'c_call'，'c_return'，或'c_exception'。*参数*取决于事件类型。

这些事件具有以下含义：

'call'
一个函数被调用（或其他一些代码块输入）。配置文件功能被调用; *arg*是None。

'return'

函数（或其他代码块）即将返回。配置文件功能被调用; *arg*是将返回的值，或者None如果事件是由引发异常引起的。

'c_call'
AC功能即将被调用。这可能是一个扩展功能或内置。 *arg*是C函数对象。

'c_return'
AC功能已返回。*arg*是C函数对象。

'c_exception'
AC功能引发了一个异常。 *arg*是C函数对象。

sys.**setrecursionlimit**(*限制*)
将Python解释器堆栈的最大深度设置为*限制*。此限制可防止无限递归导致C堆栈溢出并导致Python崩溃。

尽可能高的限制取决于平台。当用户需要深度递归的程序和支持更高限制的平台时，用户可能需要设置更高的限制。这应该谨慎处理，因为太高的限制可能会导致崩溃。

如果在当前递归深度新限制太低， RecursionError则会引发异常。

改变在3.5.1版本：—RecursionError，如果新限制，在当前递归深度过低，现在出现了异常。

sys.**setswitchinterval**(*间隔*)
设置解释器的线程切换间隔（以秒为单位）。这个浮点值决定了分配给并发运行的Python线程的“时间片”的理想持续时间。请注意，实际值可能更高，特别是如果使用长时间运行的内部函数或方法。另外，在时间间隔结束时哪个线程将被调度，这是操作系统的决定。解释器没有自己的调度器。

3.2版本中的新功能

sys.**settrace**(*tracefunc*)
设置系统的跟踪功能，它允许你在Python中实现一个Python源代码调试器。该函数是特定于线程的; 为了支持多线程的调试器，它必须使用settrace()正在调试的每个线程进行注册。

跟踪函数应该有三个参数：*frame*，*event*和 *arg*。*帧*是当前的堆栈帧。 *事件*是一个字符串：'call'，'line'，'return'或'exception'。 *参数*取决于事件类型。

只要输入新的本地范围，就会调用跟踪功能（*事件*设置为'call'）; 它应该返回一个对使用该范围的本地跟踪函数的引用，或者None如果范围不应该被跟踪。

本地跟踪函数应该返回一个对自身的引用（或者引用另一个函数用于在该范围内进一步跟踪），或者None关闭该范围内的跟踪。

这些事件具有以下含义：

'call'
一个函数被调用（或其他一些代码块输入）。全局跟踪功能被调用; *arg*是None; 返回值指定本地跟踪功能。

'line'
解释器即将执行一行新的代码或重新执行循环的条件。局部跟踪功能被调用; *arg*是None; 返回值指定新的本地跟踪功能。请参阅有关 Objects/lnotab_notes.txt如何工作的详细说明。

'return'
函数（或其他代码块）即将返回。局部跟踪功能被调用; *arg*是将返回的值，或者None如果事件是由引发异常引起的。跟踪函数的返回值被忽略。

'exception'
发生异常。局部跟踪功能被调用; *arg*是一个元组; 返回值指定新的本地跟踪功能。(exception, value, traceback)

请注意，随着异常传播到调用者链中，'exception'每个级别都会生成一个 事件。

有关代码和框架对象的更多信息，请参阅标准类型层次结构。

CPython实现细节：该settrace()函数仅用于实现调试器，分析器，覆盖工具等。它的行为是实现平台的一部分，而不是语言定义的一部分，因此可能不适用于所有Python实现。

sys.set_asyncgen_hooks(*firstiter*, *finalizer*)
接受两个可选的关键字参数，这些参数是可接受的，它们接受一个 异步生成器迭代器作为参数。所述*firstiter* 当异步发电机被重复首次调用将被调用。当异步生成器即将被垃圾收集时，*终结器*将被调用。

*3.6版新增功能：*请参阅**PEP 525**更多细节，和用于的参考例 *终结*方法见实施asyncio.Loop.shutdown_asyncgens在 LIB / ASYNCIO / base_events.py

> **注意：** 这个功能是临时添加的（请参阅 详情请参阅 **PEP 411。** ）

sys.set_coroutine_wrapper(*包装*)
允许截取协程对象的创建（仅限由函数创建的对象;用或不会截取的生成 器）。 async def types.coroutine() asyncio.coroutine()

该*包装*参数必须是：

- 一个可接受一个参数的调用（一个协程对象）;
- None，重置包装。

如果调用两次，则新包装将替换前一个。该函数是特定于线程的。

可调用的*包装器*无法直接或间接定义新的协同程序：

```
def wrapper(coro):
    async def wrap(coro):
        return await coro
    return wrap(coro)
sys.set_coroutine_wrapper(wrapper)

async def foo():
    pass

# The following line will fail with a RuntimeError, because
```

```
# ``wrapper`` creates a ``wrap(coro)`` coroutine:
foo()
```

另见get_coroutine_wrapper()。

*3.5版本中的新功能：*请参阅**PEP 492**了解更多详情。

> **注意：** 这个功能是临时添加的（请参阅 **PEP 411** ）。仅用于调试目的。

sys._enablelegacywindowsfsencoding()

与默认的文件系统编码和错误模式分别更改为“mbcs”和“替换”，以便与3.6之前的Python版本保持一致。

这相当于定义了 PYTHONLEGACYWINDOWSFSENCODING 环境变量在启动Python之前。

可用性：Windows

*3.6版新增功能：*请参阅**PEP 529**了解更多详情。

sys.stdin
sys.stdout
sys.stderr

解释器用于标准输入，输出和错误的文件对象：

- stdin用于所有交互式输入（包括呼叫 input()）；
- stdout用于输出print()和表达 语句以及提示input()；
- 解释器自己的提示及其错误消息将转到stderr。

这些流是常规文本文件，如open()函数返回的那些文件。他们的参数选择如下：

- 字符编码是平台相关的。在Windows下，如果流是交互式的（即，如果它的isatty()方法返回True），则使用控制台代码页，否则使用ANSI代码页。在其他平台下，使用区域设置编码（请参阅locale.getpreferredencoding()）。

  但是，在所有平台下，您可以通过设置该值来覆盖此值 PYTHONIOENCODING 启动Python之前的环境变量。

- 交互时，标准流是行缓冲的。否则，它们像常规文本文件一样被块缓冲。您可以使用-u命令行选项覆盖此值。

> **注意：** 要向标准流写入或读取二进制数据，请使用基础二进制buffer对象。例如，要写入字节stdout，请使用sys.stdout.buffer.write(b'abc')。
> 但是，如果您正在编写一个库（并且不控制其代码将在哪个上下文中执行），请注意，标准流可能会替换为io.StringIO不支持该buffer属性的文件类对象。

sys.__stdin__
sys.__stdout__
sys.__stderr__

这些对象包含的原始值stdin，stderr并 stdout在程序的开始。它们在定稿过程中使用，并且可以用于打印到实际的标准流，而不管该 sys.std*对象是否已被重定向。

它也可用于将实际文件恢复到已知工作文件对象，以防被已损坏的对象覆盖。但是，执行此操作的首选方法是在替换之前显式保存前一个流，然后恢复保存的对象。

> **注意：** 在某些情况下stdin，stdout和stderr以及原始值__stdin__，__stdout__并且__stderr__可以 None。对于没有连接到控制台的Windows GUI应用程序和使用**pythonw**启动的Python应用程序，情况通常如此。

sys.thread_info

一个结构序列，保存有关线程实现的信息。

属性	说明
name	线程实现的名称： • 'nt'：Windows线程 • 'pthread'：POSIX线程 • 'solaris'：Solaris线程
lock	锁执行的名称： • 'semaphore'：锁使用信号量 • 'mutex+cond'：一个锁使用一个互斥锁和一个条件变量 • None 如果这个信息是未知的
version	线程库的名称和版本。它是一个字符串，或者None如果这些信息是未知的。

3.3版本的新功能

sys.tracebacklimit

当此变量设置为整数值时，它确定发生未处理的异常时打印的追溯信息的最大级别数。默认是1000。设置为0或更低时，所有回溯信息都会被抑制，并且只会打印异常类型和值。

sys.version

一个字符串，其中包含Python解释器的版本号以及所使用的内部版本号和编译器的附加信息。交互式解释器启动时显示该字符串。不要从中提取版本信息，而是使用模块version_info提供的功能 platform。

sys.api_version

此解释器的C API版本。在调试Python和扩展模块之间的版本冲突时，程序员可能会觉得这很有用。

sys.version_info

包含版本号的五个组件的元组：*major*，*minor*， *micro*，*releaselevel*和*serial*。除*releaselevel*之外的所有值都是整数；释放水平'alpha'，'beta'，'candidate'，或

'final'。 version_info对应于Python版本2.0 的值是。组件也可以通过名称访问，所以相当于 等等。 (2, 0, 0, 'final', 0) sys.version_info[0] sys.version_info.major

*版本3.1中更改：*添加了命名组件属性。

sys.**warnoptions**

这是警告框架的实施细节; 不要修改这个值。 warnings有关警告框架的更多信息，请参阅模块。

sys.**winver**

用于在Windows平台上形成注册表项的版本号。这将作为字符串资源1000存储在Python DLL中。该值通常是前三个字符version。它在sys 模块中用于提供信息; 修改此值不会影响Python使用的注册表项。可用性：Windows。

sys.**_xoptions**

通过-X命令行选项传递的各种实现特定标志的字典。选项名称或者映射到它们的值，如果明确给出，或者给定True。例：

```
$ ./python -Xa=b -Xc
Python 3.2a3+ (py3k, Oct 16 2010, 20:14:50)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys._xoptions
{'a': 'b', 'c': True}
```

CPython实现细节：这是访问传递选项的CPython特定方式 -X。其他实现可以通过其他方式导出它们，或根本不导入它们。

3.2版本中的新功能

引文

[C99]	ISO / IEC 9899：1999。“编程语言 - C”。该标准的公共草案可在http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf获得。

29.2。sysconfig- 提供对Python配置信息的访问

3.2版本中的新功能

源代码： Lib / sysconfig.py

该sysconfig模块提供对Python配置信息的访问，如安装路径列表和与当前平台相关的配置变量。

29.2.1。配置变量

一个Python分布包含Makefile和pyconfig.h 头文件是必要建立于对Python二进制文件本身和第三方的C扩展使用编译distutils。

sysconfig将这些文件中的所有变量放在可以使用get_config_vars()or 访问的字典中get_config_var()。

请注意，在Windows中，它是一个更小的集合。

sysconfig.**get_config_vars**(**args*)

如果没有参数，则返回与当前平台相关的所有配置变量的字典。

使用参数，返回查找配置变量字典中每个参数所得到的值列表。

对于每个参数，如果未找到该值，则返回None。

sysconfig.**get_config_var**(*名字*)

返回单个变量*名称*的值。相当于 get_config_vars().get(name)。

如果找不到*名字*，请返回None。

使用示例：

```
>>> import sysconfig
>>> sysconfig.get_config_var('Py_ENABLE_SHARED')
0
>>> sysconfig.get_config_var('LIBDIR')
'/usr/local/lib'
>>> sysconfig.get_config_vars('AR', 'CXX')
['ar', 'g++']
```

29.2.2。安装路径

Python使用的安装方案因平台和安装选项而异。这些方案sysconfig根据返回的值存储在唯一标识符下os.name。

使用distutils或基于Distutils的系统安装的每个新组件都将遵循相同的方案在正确的位置复制其文件。

Python目前支持七种方案：

- *posix_prefix*：适用于Linux或Mac OS X等Posix平台的方案。这是安装Python或组件时使用的默认方案。
- *posix_home*：在安装时使用*家庭*选项时使用的POSIX平台的方案。当通过具有特定归属前缀的Distutils安装组件时使用此方案。
- *posix_user*：用于通过Distutils安装组件并使用*用户*选项时使用的Posix平台的方案。该方案定义位于用户主目录下的路径。
- *nt*：适用于Windows平台的NT平台。
- *nt_user*：使用*用户*选项时NT平台的方案。

每个方案本身都由一系列路径组成，每条路径都有一个唯一的标识符。Python目前使用八条路径：

- *stdlib*：包含非特定于平台的标准Python库文件的目录。
- *platstdlib*：包含特定于平台的标准Python库文件的目录。
- *platlib*：特定于站点的平台特定文件的目录。
- *purelib*：特定于站点的非平台特定文件的目录。
- *包括*：非平台特定头文件的目录。
- *platinclude*：平台特定的头文件的目录。
- *脚本*：脚本文件的目录。
- *数据*：数据文件的目录。

sysconfig 提供了一些功能来确定这些路径。

sysconfig.get_scheme_names ( )
返回包含当前支持的所有方案的元组 sysconfig。

sysconfig.get_path_names ( )
返回包含当前支持的所有路径名的元组 sysconfig。

sysconfig.get_path (*name* [, *scheme* [, *vars* [, *expand*]]])
从名为*scheme*的安装方案中返回与路径*名*相对应的安装路径。

*名称*必须是返回列表中的值get_path_names()。

sysconfig为每个平台存储与每个路径名相对应的安装路径，其中要展开的变量。例如 ，*nt*方案的*stdlib*路径是：。 {base}/Lib

get_path()将使用返回的变量get_config_vars() 来扩展路径。所有变量都具有每个平台的默认值，因此可以调用此函数并获取默认值。

如果提供*方案*，它必须是返回列表中的值 get_scheme_names()。否则，将使用当前平台的默认方案。

如果提供了*vars*，它必须是一个将更新字典返回值的变量字典get_config_vars()。

如果*expand*设置为False，则路径将不会使用变量进行扩展。

如果找不到*名字*，请返回None。

sysconfig.get_paths([*scheme*[, *vars*[, *expand*]]])
返回包含与安装方案对应的所有安装路径的字典。查看get_path()更多信息。

如果*方案*没有提供，将使用当前平台的默认方案。

如果*瓦尔*提供，它必须是变量，将更新用来扩展路径的字典词典。

如果*expand*设置为false，则路径将不会展开。

如果*方案*不是现有的方案，get_paths()将提出一个 KeyError。

29.2.3。其他功能

sysconfig.get_python_version()
MAJOR.MINOR以字符串形式返回Python版本号。类似于 。'%d.%d' % sys.version_info[:2]

sysconfig.get_platform()
返回一个标识当前平台的字符串。

这主要用于区分平台特定的构建目录和平台特定的构建分布。通常包括操作系统名称和版本以及体系结构（由提供os.uname()），尽管包含的确切信息取决于操作系统; 例如对于IRIX，体系结构并不特别重要（IRIX仅在SGI硬件上运行），但对于Linux，内核版本并不特别重要。

返回值的示例：

- Linux的i586的
- linux-alpha（？）
- 从Solaris 2.6 sun4u的
- IRIX-5.3
- irix64-6.2

Windows将返回以下之一：

- win-amd64（AMD64上的64位Windows（又名x86_64，Intel64，EM64T等）
- win-ia64（Itanium上的64位Windows）
- win32（所有其他 - 特别是，sys.platform返回）

Mac OS X可以返回：

- MacOSX的-10.6-PPC
- MacOSX的-10.4 PPC64
- MacOSX的-10.3-I386
- MacOSX的-10.4发

对于其他非POSIX平台，目前只是返回sys.platform。

sysconfig.is_python_build()

返回True如果正在运行的Python解释器是从源建成并正在使用其内置的位置上运行，而不是从例如运行产生的位置或通过二进制安装程序安装。make install

sysconfig.**parse_config_h**(*fp*[, *vars*])

解析一个config.h风格的文件。

*fp*是指向config.h类文件的类文件对象。

包含名称/值对的字典被返回。如果一个可选字典作为第二个参数传入，它将被用来代替一个新的字典，并使用文件中读取的值进行更新。

sysconfig.**get_config_h_filename**()

返回的路径pyconfig.h。

sysconfig.**get_makefile_filename**()

返回的路径Makefile。

29.2.4。使用sysconfig的脚本

您可以使用sysconfigPython的*-m*选项作为脚本使用：

```
$ python -m sysconfig
Platform: "macosx-10.4-i386"
Python version: "3.2"
Current installation scheme: "posix_prefix"

Paths:
        data = "/usr/local"
        include = "/Users/tarek/Dev/svn.python.org/py3k/Include"
        platinclude = "."
        platlib = "/usr/local/lib/python3.2/site-packages"
        platstdlib = "/usr/local/lib/python3.2"
        purelib = "/usr/local/lib/python3.2/site-packages"
        scripts = "/usr/local/bin"
        stdlib = "/usr/local/lib/python3.2"

Variables:
        AC_APPLE_UNIVERSAL_BUILD = "0"
        AIX_GENUINE_CPLUSPLUS = "0"
        AR = "ar"
        ARFLAGS = "rc"
        ...
```

这个调用将在标准输出打印返回的信息 get_platform() ， get_python_version() ， get_path() 和 get_config_vars()。

29.3。builtins- 内置对象

该模块提供对Python所有“内置”标识符的直接访问; 例如，builtins.open是内置函数的全名open()。有关文档，请参阅内置函数和内置常量。

这个模块通常不会被大多数应用程序明确地访问，但是可以在提供与内置值相同名称的对象的模块中使用，但也需要该名称的内置。例如，在想要实现open()包装内置函数的 open()模块中，可以直接使用此模块：

```
import builtins

def open(path):
    f = builtins.open(path, 'r')
    return UpperCaser(f)

class UpperCaser:
    '''Wrapper around a file that converts output to upper-case.'''

    def __init__(self, f):
        self._f = f

    def read(self, count=-1):
        return self._f.read(count).upper()

    # ...
```

作为一个实现细节，大多数模块的名称__builtins__都是作为其全局变量的一部分提供的。该值__builtins__通常是该模块或该模块__dict__属性的值。由于这是一个实现细节，它可能不会被替代的Python实现使用。

29.4。 __main__ - 顶层脚本环境

'__main__' 是顶级代码执行的范围的名称。'__main__' 从标准输入，脚本或交互式提示读取时，模块的__name__设置为等于。

一个模块可以通过检查它是否在主要范围内运行来发现它是否在主要范围内运行__name__，这允许在模块作为脚本运行时有条件地执行代码的通用习惯用法，或者在导入时使用但不是它们：python -m

```
if __name__ == "__main__":
    # execute only if run as a script
    main()
```

对于一个包来说，通过包含一个__main__.py模块可以达到同样的效果 ，当模块运行时，其内容将被执行-m。

29.5。 warnings- 警告控制

源代码： Lib / warnings.py

警告消息通常是在有用的情况下发出的，以提醒用户程序中的某些条件，其中该条件（通常）不能保证引发异常并终止程序。例如，当程序使用过时的模块时，可能会发出警告。

Python 程序员通过调用 warn() 本模块中定义的函数来发出警告。（C 程序员使用 PyErr_WarnEx()；有关详细信息，请参阅 异常处理 ）。

警告信息通常会写入sys.stderr，但其配置可以灵活地进行更改，从忽略所有警告到将其变为例外。警告的处置可能因警告类别（见下文），警告消息的文本以及发出警告的来源位置而异。通常会抑制针对相同源位置的特定警告的重复。

警告控制分为两个阶段：首先，每发出一次警告，就确定是否应该发布消息; 接下来，如果要发布消息，则使用用户可设置的钩子将其格式化并打印。

警告过滤器控制是否发出警告消息，这是一系列匹配规则和操作。可以通过调用将规则添加到过滤器中，filterwarnings()并通过调用将其重置为默认状态resetwarnings()。

警告消息的打印通过调用完成showwarning()，可能会被覆盖; 该函数的默认实现通过调用来格式化消息formatwarning()，这也可供自定义实现使用。

也可以看看： logging.captureWarnings() 允许您使用标准日志记录基础结构处理所有警告。

29.5.1。警告类别

有许多代表警告类别的内置异常。此分类对于能够过滤出多组警告很有用。目前定义了以下警告类别类别：

类	描述
Warning	这是所有警告类别类的基类。它是一个子类
UserWarning	默认的类别warn()。
DeprecationWarning	有被忽略（除非由__main__模块触发）。
SyntaxWarning	基本类别，用于警告关于可疑语法特征的警告。
RuntimeWarning	有关可疑运行时功能的警告的基本类别。
FutureWarning	优于构造的警告的基本类别将在未来语义上发生变
PendingDeprecationWarning	基本类别，用于警告将来不推荐使用的功能（默认情
ImportWarning	在导入模块的过程中触发警告的基本类别（默认情况
UnicodeWarning	与Unicode相关的警告的基本类别。
BytesWarning	与bytes和有关的警告的基本类别 bytearray。
ResourceWarning	与资源使用有关的警告的基本类别。

虽然这些技术上是内置的例外，但它们在这里被记录，因为它们在概念上属于警告机制。

用户代码可以通过继承其中一个标准警告类别来定义其他警告类别。警告类别必须始终是`Warning`该类的一个子类。

29.5.2。警告过滤器

警告过滤器控制警告是否被忽略，显示或转化为错误（引发异常）。

从概念上讲，警告过滤器维护过滤器规格的有序列表; 任何特定的警告依次与列表中的每个过滤规范进行匹配，直到找到匹配; 比赛决定比赛的处置。每个条目都是表单（*action*， *message*，*category*，*module*，*lineno*）的元组，其中：

- *操作*是以下字符串之一：

值	性格
`"error"`	将匹配的警告转化为例外
`"ignore"`	从不打印匹配的警告
`"always"`	始终打印匹配的警告
`"default"`	打印出现警告的每个位置的首次匹配警告
`"module"`	打印出现警告的每个模块的首次匹配警告
`"once"`	不管位置如何，只打印首次出现的匹配警告

- *消息*是一个包含正则表达式的字符串，警告消息的开头必须匹配。该表达式被编译为始终不区分大小写。
- *category*是一个类（其子类`Warning`），为了匹配，警告类别必须是子类。
- *模块*是一个包含模块名称必须匹配的正则表达式的字符串。该表达式被编译为区分大小写。
- *lineno*是发生警告的行号必须匹配的整数，或者`0`匹配所有行号。

由于`Warning`该类是从内置类派生的`Exception` ，为了将警告转化为错误，我们只需提出`category(message)`。

警告过滤器由`-W`传递给Python解释器命令行的选项初始化。解释者将所有`-W`选项的参数保存在没有解释的情况下`sys.warnoptions`; 该 `warnings`模块在首次导入时解析这些数据（在打印消息后忽略无效选项`sys.stderr`）。

29.5.2.1。默认警告过滤器

默认情况下，Python会安装多个警告过滤器，这些过滤器可以通过传递给`-W`和调用 的命令行选项来覆盖`filterwarnings()`。

- `DeprecationWarning`并且`PendingDeprecationWarning`，和 `ImportWarning`被忽略。
- `BytesWarning`被忽略，除非该`-b`选项给出一次或两次; 在这种情况下，这个警告要么被打印（`-b`），要么变成异常（`-bb`）。
- `ResourceWarning` 被忽略，除非Python是在调试模式下构建的。

在版本 3.2 中更改： DeprecationWarning 现在除了默认情况下被忽略 PendingDeprecationWarning。

29.5.3。暂时抑制警告

如果您使用的代码会引发警告（例如不推荐使用的函数），但不希望看到警告，则可以使用 catch_warnings上下文管理器来抑制警告：

```
import warnings

def fxn():
    warnings.warn("deprecated", DeprecationWarning)

with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    fxn()
```

在上下文管理器中，所有的警告都将被忽略。这使您可以使用已知不推荐的代码，而不必查看警告，同时不抑制其他可能不知道其使用弃用代码的代码的警告。注意：这只能在单线程应用程序中保证。如果两个或更多线程同时使用catch_warnings上下文管理器，则行为是未定义的。

29.5.4。测试警告

要测试代码引发的警告，请使用catch_warnings上下文管理器。有了它，您可以暂时改变警告筛选器以方便您的测试。例如，执行以下操作捕获所有引发的警告以检查：

```
import warnings

def fxn():
    warnings.warn("deprecated", DeprecationWarning)

with warnings.catch_warnings(record=True) as w:
    # Cause all warnings to always be triggered.
    warnings.simplefilter("always")
    # Trigger a warning.
    fxn()
    # Verify some things
    assert len(w) == 1
    assert issubclass(w[-1].category, DeprecationWarning)
    assert "deprecated" in str(w[-1].message)
```

也可以通过使用error而不是 使所有警告成为例外always。有一点需要注意的是，如果由于once/ default规则已经引发了警告，那么不管设置了哪些过滤器，除非与警告相关的警告注册表已被清除，否则警告将不会再被看到。

一旦上下文管理器退出，警告过滤器将恢复到输入上下文时的状态。这可以防止测试在测试之间以意外的方式更改警告过滤器，并导致不确定的测试结果。 showwarning()模块中的功能也恢复到其原始值。注意：这只能在单线程应用程序中保证。如果两个或更多线程同时使用 catch_warnings上下文管理器，则行为是未定义的。

在测试引发同类警告的多个操作时，重要的是要以确认每个操作引发新警告的方式对它们进行测试（例如，将警告设置为例外并检查操作是否引发异常，检查长度警告列表在每次操作后继续增加，或者在每次新操作之前从警告列表中删除以前的条目）。

29.5.5。更新Python新版本的代码

只有开发者感兴趣的警告在默认情况下被忽略。因此，您应该确保通过可见的通常被忽略的警告来测试您的代码。您可以通过传递`-Wd`给解释器（这是速记）从命令行执行此操作。这将启用默认处理所有警告，包括默认情况下忽略的警告。要改变对遇到的警告采取什么行动，只需更改传递给参数的参数，例如。有关可能的更多细节，请参阅 标志。`-W default` `-W -W error` `-W`

要以编程方式执行相同的操作`-Wd`，请使用：

```
warnings.simplefilter('default')
```

确保尽快执行此代码。这可以防止注册已经引发的警告，从而意外地影响将来的警告如何被处理。

默认情况下会忽略某些警告，以防止用户看到仅由开发人员感兴趣的警告。由于您不一定能够控制用户使用哪些解释器来运行其代码，因此可能会在您的发布周期之间发布新版本的Python。新的解释器版本可能会在代码中触发新的警告，而这些警告在旧版本的解释器中不存在，例如`DeprecationWarning`您正在使用的模块。尽管作为开发人员希望得到您的代码使用已弃用模块的通知，但对用户而言，此信息本质上是噪声，并且不会给他们带来任何好处。

在运行测试的同时，该`unittest`模块也进行了更新以使用`'default'`过滤器。

29.5.6。可用函数

warnings.**warn**（*message*，*category* = *None*，*stacklevel* = *1*，*source* = *None* ）

发出警告，或者忽略它或引发异常。该*类别*参数，如果给定的，必须是一个警告类别类（见上文）；它默认为`UserWarning`。或者，*消息*可以是`Warning`实例，在这种情况下，*类别*将被忽略`message.__class__`并将被使用。在这种情况下，消息文本将会是`str(message)`。如果发出的特定警告被上述警告过滤器更改为错误，则此功能会引发异常。该*stacklevel*参数可以使用Python编写包装函数，如下所示：

```
def deprecation(message):
    warnings.warn(message, DeprecationWarning, stacklevel=2)
```

这使得警告指`deprecation()`的是调用者，而不是`deprecation()`它本身的源头（因为后者会打败警告信息的目的）。

来源，如果提供的话，是发射出来的被毁坏物体`ResourceWarning`。

*在版本3.6中更改：*添加了*源*参数。

warnings.**warn_explicit**（*message*，*category*，*filename*，*lineno*，*module* = *None*，*registry* = *None*，*module_globals* = *None*，*source* = *None* ）

这是一个低层次的接口，可以warn()显式传递消息，类别，文件名和行号，以及可选的模块名称和注册表（应该是__warningregistry__模块的字典）。模块名称默认为.py剥离的文件名；如果没有通过注册表，警告从不被抑制。*消息*必须是字符串，并且*类别*Warning或*消息*的子类可以是Warning实例，在这种情况下，*类别*将被忽略。

module_globals（如果提供）应该是发出警告的代码所使用的全局名称空间。（此参数用于支持显示zip文件或其他非文件系统导入源中找到的模块的源代码）。

来源，如果提供的话，是发射出来的被毁坏物体 ResourceWarning。

*在版本3.6中更改：*添加*源*参数。

warnings.showwarning（*message*，*category*，*filename*，*lineno*，*file = None*，*line = None* ）

给文件写一个警告。默认实现调用 并将结果字符串写入*文件*，该*文件*默认为。您可以通过分配给任何可调用的函数替换此函数。*行*是要包含在警告消息中的一行源代码; 如果*线路*没有提供，将尝试读取指定的行*名*和*LINENO*。 formatwarning(message, category, filename, lineno, line) sys.stderr warnings.showwarning showwarning()

warnings.formatwarning（*message*，*category*，*filename*，*lineno*，*line = None* ）

以标准方式格式化警告。这将返回一个可能包含嵌入换行符并以换行符结尾的字符串。*行*是要包含在警告消息中的一行源代码; 如果*线路*没有提供， formatwarning()将尝试读取指定的行*名*和 *LINENO*。

warnings.filterwarnings（*action*，*message =''*，*category = Warning*，*module =''*，*lineno = 0*，*append = False* ）

将条目插入警告筛选器规范列表中。该条目默认插入在前面; 如果 *append*为true，则在最后插入。这将检查参数的类型，编译*消息*和*模块*正则表达式，并将它们作为元组插入警告过滤器列表中。如果两个条目都匹配一个特定的警告，那么靠近列表前面的条目会覆盖列表中稍后的条目。省略参数默认为匹配所有内容的值。

warnings.simplefilter（*action*，*category = Warning*，*lineno = 0*，*append = False* ）

将简单条目插入警告筛选器规范列表中。函数参数的意义是相同的 filterwarnings()，但不需要正则表达式，因为只要类别和行号匹配，插入的过滤器就会始终匹配任何模块中的任何消息。

warnings.resetwarnings（ ）

重置警告过滤器。这将放弃所有先前调用的效果 filterwarnings()，包括-W命令行选项和调用的效果simplefilter()。

29.5.7。可用的上下文管理器

class warnings.catch_warnings（***，*record = False*，*module = None* ）

一个上下文管理器，复制并在退出时恢复警告过滤器和showwarning()函数。如果*记录*参数是False（默认），则上下文管理器None在输入时返回。如果*记录*是True，则会返回一个列表，该列表逐渐由自定义showwarning()函数（也抑制输出到sys.stdout）所见的对象填充。列表中的每个对象都具有与参数名称相同的属性 showwarning()。

该*模块*参数采用，将被用来代替当您导入返回该模块的模块`warnings` ，其过滤器将被保护。这个参数主要用于测试`warnings` 模块本身。

> **注意：** 该`catch_warnings`经理的工作方式代替，再后来恢复模块的 `showwarning()`功能和规格的过滤器的内部列表。这意味着上下文管理器正在修改全局状态，因此不是线程安全的。

29.6。contextlib- 用于with语境的实用程序

源代码： Lib / contextlib.py

该模块为涉及该with 语句的常见任务提供实用程序。有关更多信息，请参阅上下文管理器类型和 使用语句上下文管理器。

29.6.1。公用事业

提供的功能和类：

*类*contextlib. AbstractContextManager

实现和的类 的抽象基类。为一个默认的实现提供一种返回 而是默认返回一个抽象方法，该方法。另请参阅上下文管理器类型的定义。object.__enter__() object.__exit__() object.__enter__() self object.__exit__() None

3.6版本中的新功能。

@contextlib. contextmanager

此功能是装饰，可用于定义一个工厂函数with声明上下文管理，而无需创建一个类或者单独__enter__()和__exit__()方法。

一个简单的例子（这不建议作为生成HTML的真正方法！）：

```
from contextlib import contextmanager

@contextmanager
def tag(name):
    print("<%s>" % name)
    yield
    print("</%s>" % name)

>>> with tag("h1"):
...     print("foo")
...
<h1>
foo
</h1>
```

被调用的函数在调用时必须返回一个生成器引发器。此迭代器必须产生恰好一个值，该值将绑定到with语句as子句中的目标（如果有的话）。

在发生器产生的点上，嵌套在with 语句中的块被执行。该模块退出后，发电机恢复运行。如果块中发生未处理的异常，则在发生产量的位置将其重新发送到生成器中。因此，你可以使用一个 try... except... finally语句来捕获错误（如果有的话），或确保一些清理发生。如果一个异常仅仅为了记录它或者执行一些操作而被捕获（而不是完全压制它），那么生成器必须重新评估该异常。否则，生成器上下文管理器将向with语句指示该异常已被处理，并且执行将随着语句紧随该with语句而继续。

contextmanager() 使用ContextDecorator这样它会创建上下文管理器可以作为装饰，以及在with声明。当作为装饰器使用时，每个函数调用都会隐式创建一个新的生成器实例（这样可以创建其他“一次性”上下文管理器，contextmanager() 以满足上下文管理器支持多个调用以便用作装饰器的要求）。

*在版本3.2中更改：*使用ContextDecorator。

contextlib.**closing**(*东西*)

返回一个上下文管理器，在完成该块后关闭*事物*。这基本相当于：

```
from contextlib import contextmanager

@contextmanager
def closing(thing):
    try:
        yield thing
    finally:
        thing.close()
```

并让你编写这样的代码：

```
from contextlib import closing
from urllib.request import urlopen

with closing(urlopen('http://www.python.org')) as page:
    for line in page:
        print(line)
```

而不需要明确关闭page。即使发生错误， page.close() 当with块被退出时也会被调用。

contextlib.**suppress**(**例外*)

返回一个上下文管理器，用于抑制任何指定的异常（如果它们出现在with语句的主体中），然后用with语句结束后的第一个语句继续执行。

和其他任何完全抑制异常的机制一样，这个上下文管理器应该只用于覆盖非常具体的错误，在那里默默地继续执行程序是正确的。

例如：

```
from contextlib import suppress

with suppress(FileNotFoundError):
    os.remove('somefile.tmp')

with suppress(FileNotFoundError):
    os.remove('someotherfile.tmp')
```

此代码相当于：

```
try:
    os.remove('somefile.tmp')
except FileNotFoundError:
```

```
    pass

try:
    os.remove('someotherfile.tmp')
except FileNotFoundError:
    pass
```

这个上下文管理器是可重入的。

3.4版新增功能

contextlib.**redirect_stdout**(*new_target*)

上下文管理器用于临时重定向sys.stdout到另一个文件或类似文件的对象。

该工具增加了将输出硬连线到stdout的现有函数或类的灵活性。

例如，help()通常的输出发送到*sys.stdout*。您可以通过将输出重定向到一个io.StringIO对象来捕获该输出 ：

```
f = io.StringIO()
with redirect_stdout(f):
    help(pow)
s = f.getvalue()
```

要将输出发送help()到磁盘上的文件，请将输出重定向到常规文件：

```
with open('help.txt', 'w') as f:
    with redirect_stdout(f):
        help(pow)
```

要将输出发送help()到*sys.stderr*：

```
with redirect_stdout(sys.stderr):
    help(pow)
```

请注意，全局副作用sys.stdout意味着此上下文管理器不适合在库代码和大多数线程应用程序中使用。它也不会影响子过程的输出。但是，它对许多实用程序脚本仍然是一个有用的方法。

这个上下文管理器是可重入的。

3.4版新增功能

contextlib.**redirect_stderr**(*new_target*)

类似于redirect_stdout()但重定向 sys.stderr到另一个文件或类似文件的对象。

这个上下文管理器是可重入的。

3.5版本中的新功能。

*类*contextlib.**ContextDecorator**

基类使上下文管理器也可用作装饰器。

继承的上下文管理器ContextDecorator必须执行 __enter__并且__exit__正常。 __exit__即使在作为装饰器使用时也会保留其可选的异常处理。

ContextDecorator被使用contextmanager() ，所以你自动获得这个功能。

例子ContextDecorator ：

```
from contextlib import ContextDecorator

class mycontext(ContextDecorator):
    def __enter__(self):
        print('Starting')
        return self

    def __exit__(self, *exc):
        print('Finishing')
        return False

>>> @mycontext()
... def function():
...     print('The bit in the middle')
...
>>> function()
Starting
The bit in the middle
Finishing

>>> with mycontext():
...     print('The bit in the middle')
...
Starting
The bit in the middle
Finishing
```

对于以下形式的任何构造，此更改只是语法糖：

```
def f():
    with cm():
        # Do stuff
```

ContextDecorator 让你改为写：

```
@cm()
def f():
    # Do stuff
```

它清楚地表明，cm适用于整个功能，而不仅仅是一个部分（并且节省缩进级别也很好）。

已有基类的现有上下文管理器可以ContextDecorator作为混合类来扩展：

```
from contextlib import ContextDecorator

class mycontext(ContextBaseClass, ContextDecorator):
    def __enter__(self):
        return self
```

```
    def __exit__(self, *exc):
        return False
```

注意： 由于装饰函数必须能够被多次调用，所以底层的上下文管理器必须支持在多个 with 语句中使用。如果不是这种情况，那么with应该使用函数内部具有显式语句的原始构造。

3.2版本中的新功能

*类*contextlib.ExitStack

上下文管理器，旨在使编程方便地组合其他上下文管理器和清理函数，尤其是那些可选或由输入数据驱动的清理函数。

例如，一组文件可以很容易地用一个语句处理，如下所示：

```
with ExitStack() as stack:
    files = [stack.enter_context(open(fname)) for fname in filenames]
    # All opened files will automatically be closed at the end of
    # the with statement, even if attempts to open files later
    # in the list raise an exception
```

每个实例都维护一堆已注册的回调，当实例关闭时（在with语句结尾显式或隐式地）以相反顺序调用。请注意， 当上下文堆栈实例被垃圾收集时，*不会*隐式地调用回调。

使用这种堆栈模型，__init__可以正确处理在其方法中获取资源的上下文管理器（如文件对象）。

由于已注册的回调以相反的注册顺序调用，因此最终表现得好像多个嵌套with 语句已与注册回调集一起使用。这甚至延伸到异常处理 - 如果内部回调抑制或替换异常，则外部回调将基于更新后的状态传递参数。

这是一个相对较低级别的API，负责正确解开退出回调堆栈的细节。它为更高级别的上下文管理器提供了一个合适的基础，这些管理器以特定于应用程序的方式操作退出堆栈

3.3版本的新功能

enter_context（*cm*）

输入一个新的上下文管理器并将其__exit__()方法添加到回调堆栈中。返回值是上下文管理器自己的__enter__()方法的结果。

这些上下文管理器可能会将异常直接作为with语句的一部分直接使用，从而抑制异常。

push（*退出*）

将上下文管理器的__exit__()方法添加到回调堆栈中。

至于__enter__是*不是*调用，这种方法可以用来覆盖的部分__enter__()与上下文经理自己的实现 __exit__()方法。

如果传递了不是上下文管理器的对象，则此方法假定它是具有与上下文管理器__exit__()方法相同签名的回调，并将其直接添加到回调堆栈中。

通过返回真值，这些回调可以像上下文管理器__exit__()方法一样抑制异常。

从函数返回传入的对象，允许将此方法用作函数装饰器。

callback(*callback*, **args*, ***kwds*)

接受一个任意的回调函数和参数并将其添加到回调堆栈中。

与其他方法不同，以这种方式添加的回调无法抑制异常（因为它们从未传递异常详细信息）。

传入的回调从函数返回，允许此方法用作函数装饰器。

pop_all()

将回调堆栈转移到新ExitStack实例并返回。这个操作不会调用回调 - 相反，当新堆栈关闭时（明确或隐含地在with语句结尾处），它们将被调用。

例如，一组文件可以打开为“全部或全部”操作，如下所示：

```
with ExitStack() as stack:
    files = [stack.enter_context(open(fname)) for fname in filenames]
    # Hold onto the close method, but don't call it yet.
    close_files = stack.pop_all().close
    # If opening any file fails, all previously opened files will be
    # closed automatically. If all files are opened successfully,
    # they will remain open even after the with statement ends.
    # close_files() can then be invoked explicitly to close them all.
```

close()

立即展开回调堆栈，以相反的注册顺序调用回调。对于注册的任何上下文管理器和退出回调，传入的参数将指示没有发生异常。

29.6.2。示例和食谱

本节介绍一些有效使用由提供的工具的示例和配方contextlib。

29.6.2.1。支持可变数量的上下文管理器

主要用例ExitStack是类文档中给出的用例：在单个with语句中支持可变数量的上下文管理器和其他清理操作。可变性可能来自需要由用户输入驱动的上下文管理器的数量（例如打开用户指定的文件集合），或者来自某些上下文管理器是可选的：

```
with ExitStack() as stack:
    for resource in resources:
        stack.enter_context(resource)
    if need_special_resource():
        special = acquire_special_resource()
```

```
        stack.callback(release_special_resource, special)
    # Perform operations that use the acquired resources
```

如图所示，ExitStack还使得使用with 语句来管理任何本身不支持上下文管理协议的资源变得非常容易。

29.6.2.2。简化对单个可选上下文管理器的支持

在单个可选上下文管理器的特定情况下，ExitStack 实例可以用作“无所事事”上下文管理器，允许在不影响源代码的整体结构的情况下轻松地省略上下文管理器：

```
def debug_trace(details):
    if __debug__:
        return TraceContext(details)
    # Don't do anything special with the context in release mode
    return ExitStack()

with debug_trace():
    # Suite is traced in debug mode, but runs normally otherwise
```

29.6.2.3。捕获__enter__方法的异常

偶尔需要从__enter__ 方法实现中捕获异常，*而不会*无意中从with语句主体或上下文管理器的__exit__ 方法中捕获异常。通过使用ExitStack上下文管理协议中的步骤可以稍微分开以便允许：

```
stack = ExitStack()
try:
    x = stack.enter_context(cm)
except Exception:
    # handle __enter__ exception
else:
    with stack:
        # Handle normal case
```

实际上，需要这样做的可能表明底层API应该提供直接资源管理界面以便与 try/ except/ finally语句一起使用，但并非所有API在这方面都设计良好。当上下文管理器是唯一提供的资源管理API时，ExitStack可以更轻松地处理无法在with语句中直接处理的各种情况 。

29.6.2.4。在__enter__实施中清理

如文档中所述ExitStack.push() ，如果后面的__enter__()实现步骤失败，则此方法可用于清理已分配的资源。

下面是一个为上下文管理器执行此操作的示例，该上下文管理器接受资源获取和释放函数以及可选的验证函数，并将它们映射到上下文管理协议：

```
from contextlib import contextmanager, AbstractContextManager, ExitStack

class ResourceManager(AbstractContextManager):
```

```
    def __init__(self, acquire_resource, release_resource, check_resource_ok=None):
        self.acquire_resource = acquire_resource
        self.release_resource = release_resource
        if check_resource_ok is None:
            def check_resource_ok(resource):
                return True
        self.check_resource_ok = check_resource_ok

    @contextmanager
    def _cleanup_on_error(self):
        with ExitStack() as stack:
            stack.push(self)
            yield
            # The validation check passed and didn't raise an exception
            # Accordingly, we want to keep the resource, and pass it
            # back to our caller
            stack.pop_all()

    def __enter__(self):
        resource = self.acquire_resource()
        with self._cleanup_on_error():
            if not self.check_resource_ok(resource):
                msg = "Failed validation for {!r}"
                raise RuntimeError(msg.format(resource))
        return resource

    def __exit__(self, *exc_details):
        # We don't need to duplicate any of our resource release logic
        self.release_resource()
```

29.6.2.5。替换任何使用try-finally和标记变量

有时你会看到一个模式是一个try-finally带有标志变量的语句，用于指示是否finally应该执行该子句的主体。以最简单的形式（不能仅仅通过使用一个except子句来处理），它看起来像这样：

```
cleanup_needed = True
try:
    result = perform_operation()
    if result:
        cleanup_needed = False
finally:
    if cleanup_needed:
        cleanup_resources()
```

与任何try基于声明的代码一样，这可能会导致开发和检查出现问题，因为安装程序代码和清理代码可能最终由任意长的代码段分隔开来。

ExitStack可以在with语句结尾处注册执行的回调，然后再决定跳过执行该回调：

```
from contextlib import ExitStack

with ExitStack() as stack:
```

```
    stack.callback(cleanup_resources)
    result = perform_operation()
    if result:
        stack.pop_all()
```

这可以使预期的清理行为在前面显示出来，而不需要单独的标志变量。

如果一个特定的应用程序使用这种模式很多，它可以通过一个小的辅助类来进一步简化：

```
from contextlib import ExitStack

class Callback(ExitStack):
    def __init__(self, callback, *args, **kwds):
        super(Callback, self).__init__()
        self.callback(callback, *args, **kwds)

    def cancel(self):
        self.pop_all()

with Callback(cleanup_resources) as cb:
    result = perform_operation()
    if result:
        cb.cancel()
```

如果资源清理尚未整齐地捆绑到一个独立的函数中，那么仍然可以使用装饰器形式ExitStack.callback()来提前声明资源清理：

```
from contextlib import ExitStack

with ExitStack() as stack:
    @stack.callback
    def cleanup_resources():
        ...
    result = perform_operation()
    if result:
        stack.pop_all()
```

由于装饰者协议的工作方式，这种声明的回调函数不能接受任何参数。相反，任何要释放的资源必须作为闭包变量来访问。

29.6.2.6。使用上下文管理器作为函数装饰器

ContextDecorator可以在普通with语句中使用上下文管理器，也可以作为函数装饰器使用。

例如，用记录器打包函数或语句组有时很有用，它可以跟踪输入时间和退出时间。继承并不是为任务编写函数装饰器和上下文管理器，而是ContextDecorator在单一定义中提供两种功能：

```
from contextlib import ContextDecorator
import logging

logging.basicConfig(level=logging.INFO)

class track_entry_and_exit(ContextDecorator):
```

```
    def __init__(self, name):
        self.name = name

    def __enter__(self):
        logging.info('Entering: %s', self.name)

    def __exit__(self, exc_type, exc, exc_tb):
        logging.info('Exiting: %s', self.name)
```

这个类的实例可以用作上下文管理器：

```
with track_entry_and_exit('widget loader'):
    print('Some time consuming activity goes here')
    load_widget()
```

并且作为一个函数装饰器：

```
@track_entry_and_exit('widget loader')
def activity():
    print('Some time consuming activity goes here')
    load_widget()
```

请注意，使用上下文管理器作为函数装饰器时还有一个额外的限制：无法访问返回值`__enter__()`。如果需要该值，则仍然需要使用明确的`with`语句。

也可以看看:

PEP 343 - “with”声明
Python `with` 语句的规范，背景和示例。

29.6.3。单次使用，可重复使用和重入式上下文管理器

大多数上下文管理器的编写方式意味着它们只能在`with`声明中有效地使用一次。这些单次使用的上下文管理器必须在每次使用时重新创建 - 试图再次使用它们将触发异常或以其他方式无法正常工作。

这种常见的限制意味着通常建议直接在`with`使用它们的语句的标题中创建上下文管理器（如上面所有使用示例所示）。

文件是一个有效的单一使用上下文管理器的例子，因为第一个`with`语句将关闭文件，阻止使用该文件对象的任何进一步的IO操作。

创建使用的上下文管理器`contextmanager()`也是单用途上下文管理器，如果试图再次使用它们，将会抱怨底层的生成器无法生成。

```
>>> from contextlib import contextmanager
>>> @contextmanager
... def singleuse():
...     print("Before")
...     yield
...     print("After")
```

```
...
>>> cm = singleuse()
>>> with cm:
...     pass
...
Before
After
>>> with cm:
...     pass
...
Traceback (most recent call last):
    ...
RuntimeError: generator didn't yield
```

29.6.3.1。重入上下文管理器

更复杂的上下文管理者可能是“重入”的。这些上下文管理器不仅可以在多个with语句中使用，还可以*在*with已经使用相同上下文管理器的语句中使用。

threading.RLock 是一个可重入上下文管理器的一个例子，因为是 suppress() 和 redirect_stdout()。这是一个非常简单的可重入使用示例：

```
>>> from contextlib import redirect_stdout
>>> from io import StringIO
>>> stream = StringIO()
>>> write_to_stream = redirect_stdout(stream)
>>> with write_to_stream:
...     print("This is written to the stream rather than stdout")
...     with write_to_stream:
...         print("This is also written to the stream")
...
>>> print("This is written directly to stdout")
This is written directly to stdout
>>> print(stream.getvalue())
This is written to the stream rather than stdout
This is also written to the stream
```

现实世界中的重入例子更可能涉及多个函数互相调用，因此比这个例子要复杂得多。

还要注意的是，重入和线程安全*不是*一回事。 redirect_stdout()例如，绝对不是线程安全的，因为它通过绑定sys.stdout 到不同的流来对系统状态进行全局修改。

29.6.3.2。可重复使用的上下文管理

与单用和可重入上下文管理器不同的是，“可重用”上下文管理器（或者，因为可重入上下文管理器也是可重用的，所以完全明确地说，“可重用，但不可重入”的上下文管理器）。这些上下文管理器支持多次使用，但如果特定的上下文管理器实例已经在包含with语句中使用过，则会失败（或者不能正常工作）。

threading.Lock是可重用但不可重入的上下文管理器的示例（对于可重入锁，则需要改为使用threading.RLock）。

可重复使用但不可重入的上下文管理器的另一个例子是 ExitStack，当它离开任何语句时调用*所有*当前已注册的回调，而不管这些回调是在哪里添加的：

```
>>> from contextlib import ExitStack
>>> stack = ExitStack()
>>> with stack:
...     stack.callback(print, "Callback: from first context")
...     print("Leaving first context")
...
Leaving first context
Callback: from first context
>>> with stack:
...     stack.callback(print, "Callback: from second context")
...     print("Leaving second context")
...
Leaving second context
Callback: from second context
>>> with stack:
...     stack.callback(print, "Callback: from outer context")
...     with stack:
...         stack.callback(print, "Callback: from inner context")
...         print("Leaving inner context")
...     print("Leaving outer context")
...
Leaving inner context
Callback: from inner context
Callback: from outer context
Leaving outer context
```

正如示例中的输出所示，跨多个with语句重复使用单个堆栈对象可正常工作，但试图嵌套它们将导致堆栈在最内部的with语句末尾被清除，这不太可能是理想的行为。

使用单独的ExitStack实例而不是重复使用单个实例可避免该问题：

```
>>> from contextlib import ExitStack
>>> with ExitStack() as outer_stack:
...     outer_stack.callback(print, "Callback: from outer context")
...     with ExitStack() as inner_stack:
...         inner_stack.callback(print, "Callback: from inner context")
...         print("Leaving inner context")
...     print("Leaving outer context")
...
Leaving inner context
Callback: from inner context
Leaving outer context
Callback: from outer context
```

29.7。abc-抽象基类

源代码： Lib / abc.py

这个模块提供了在Python中定义抽象基类（ABCs）的基础结构，如下所述**PEP 3119**；请参阅PEP为什么将其添加到Python。（也可以看看**PEP 3141**和numbers关于基于ABC的数字的类型层次的 模块。）

该collections模块有一些来自ABCs的具体类; 当然，这些可以进一步推导出来。此外，collections.abc子模块还有一些可用于测试类或实例是否提供特定接口的ABCs，例如它是可散列的还是映射。

这个模块提供了ABCMeta用于定义ABCs 的元类和一个辅助类ABC来通过继承来定义ABCs：

*类*abc.ABC

ABCMeta作为元类的辅助类。通过这个类，可以通过简单地ABC 避免有时令人困惑的元类使用而创建抽象基类，例如：

```
from abc import ABC

class MyABC(ABC):
    pass
```

请注意，由于多重继承可能导致元类冲突，因此类型ABC仍然是ABCMeta继承关系，因此ABC需要关于元类使用的常规预防措施。也可以通过传递metaclass关键字并ABCMeta直接使用来定义抽象基类，例如：

```
from abc import ABCMeta

class MyABC(metaclass=ABCMeta):
    pass
```

3.4版新增功能

*类*abc.ABCMeta

用于定义抽象基类（ABC）的元类。

使用这个元类来创建一个ABC。ABC可以直接分类，然后作为混合课。您也可以将不相关的具体类（甚至内置类）和不相关的ABCs注册为“虚拟子类” - 这些及其后代将被内置issubclass()函数视为注册ABC的子类，但注册ABC不会显示（方法解析顺序），注册ABC定义的方法实现也不会被调用（甚至不能 super()）。[1]

使用元类创建的类ABCMeta具有以下方法：

register(*子类*)

将*子类*注册为此ABC的“虚拟子类”。例如：

```
from abc import ABC

class MyABC(ABC):
    pass

MyABC.register(tuple)

assert issubclass(tuple, MyABC)
assert isinstance((), MyABC)
```

*版本3.3中更改：*返回已注册的子类，以允许用作类装饰器。

*在版本3.4中更改：*要检测呼叫`register()`，您可以使用该`get_cache_token()`功能。

您也可以在抽象基类中重写此方法：

__subclasshook__（*子类*）

（必须定义为类方法。）

检查*子类*是否被认为是该ABC的一个子类。这意味着您可以自定义`issubclass`进一步的行为，而无需调用`register()`您想考虑ABC的子类的每个类。（这个类的方法是从`__subclasscheck__()`ABC 的方法中调用 的。）

这个方法应该返回`True`，`False`或者`NotImplemented`。如果它返回`True`，则该*子类*被认为是该ABC的一个子类。如果它返回`False`，则该*子类*不被视为该ABC的子类，即使它通常是一个。如果它返回 `NotImplemented`，子类检查将继续使用通常的机制。

为了演示这些概念，请看这个ABC定义的例子：

```
class Foo:
    def __getitem__(self, index):
        ...
    def __len__(self):
        ...
    def get_iterator(self):
        return iter(self)

class MyIterable(ABC):

    @abstractmethod
    def __iter__(self):
        while False:
            yield None

    def get_iterator(self):
        return self.__iter__()

    @classmethod
    def __subclasshook__(cls, C):
        if cls is MyIterable:
            if any("__iter__" in B.__dict__ for B in C.__mro__):
                return True
        return NotImplemented

MyIterable.register(Foo)
```

ABC MyIterable将标准迭代方法定义 __iter__() 为抽象方法。这里给出的实现仍然可以从子类中调用。该get_iterator()方法也是MyIterable抽象基类的一部分，但它不必在非抽象派生类中重写。

__subclasshook__() 这里定义的类方法表示，任何__iter__()在其 __dict__ （或者在其基类之一中，通过__mro__列表访问的方法）的类都被认为是一个类MyIterable。

最后，最后一行生成Foo一个虚拟子类MyIterable，尽管它没有定义一个__iter__()方法（它使用旧式迭代协议，按照__len__()和 定义__getitem__()）。请注意，这不会get_iterator 作为一种方法提供Foo，因此它是单独提供的。

该abc模块还提供以下装饰器：

@abc.abstractmethod

指示抽象方法的装饰器。

使用这个装饰器需要该类的元类是ABCMeta 或来源于它。ABCMeta除非所有抽象方法和属性都被覆盖，否则具有派生元类的类 不能实例化。抽象方法可以使用任何普通的“超级”调用机制来调用。 abstractmethod()可以用来声明属性和描述符的抽象方法。

不支持动态地将抽象方法添加到类中，或者尝试在创建方法或类时修改抽象状态。将abstractmethod()仅影响使用常规继承派生的子类; 用ABC register()方法注册的“虚拟子类”不受影响。

当abstractmethod()与其他方法描述符结合使用时，应将其作为最内层的装饰器应用，如以下用法示例所示：

```
class C(ABC):
    @abstractmethod
    def my_abstract_method(self, ...):
        ...
    @classmethod
    @abstractmethod
    def my_abstract_classmethod(cls, ...):
        ...
    @staticmethod
    @abstractmethod
    def my_abstract_staticmethod(...):
        ...

    @property
    @abstractmethod
    def my_abstract_property(self):
        ...
    @my_abstract_property.setter
    @abstractmethod
    def my_abstract_property(self, val):
        ...

    @abstractmethod
    def _get_x(self):
        ...
    @abstractmethod
```

```
    def _set_x(self, val):
        ...
    x = property(_get_x, _set_x)
```

为了正确地与抽象基类机器进行互操作，描述符必须将自身标识为抽象使用 __isabstractmethod__。一般而言，True 如果任何用于组成描述符的方法都是抽象的，则该属性应该是。例如，Python的内置属性等价于：

```
class Descriptor:
    ...
    @property
    def __isabstractmethod__(self):
        return any(getattr(f, '__isabstractmethod__', False) for
                   f in (self._fget, self._fset, self._fdel))
```

注意： 与Java抽象方法不同，这些抽象方法可能有一个实现。这个实现可以通过 super() 覆盖它的类的机制来调用。这可以作为使用协作多重继承的框架中的超级调用的终点。

该abc模块还支持以下传统装饰器：

@abc.**abstractclassmethod**

3.2版本中的新功能

*自从3.3版本不推荐使用：*现在可以使用classmethod用 abstractmethod() ，使这个装饰是多余的。

内置的子类classmethod() ，表示抽象类方法。否则它类似于abstractmethod()。

这种特殊情况已被废弃，因为classmethod()装饰器现在在应用于抽象方法时被正确识别为抽象：

```
class C(ABC):
    @classmethod
    @abstractmethod
    def my_abstract_classmethod(cls, ...):
        ...
```

@abc.**abstractstaticmethod**

3.2版本中的新功能

*自从3.3版本不推荐使用：*现在可以使用staticmethod用 abstractmethod() ，使这个装饰是多余的。

内置的子类staticmethod() ，表示抽象静态方法。否则它类似于abstractmethod()。

这种特殊情况已被废弃，因为staticmethod()装饰器现在在应用于抽象方法时被正确识别为抽象：

```
class C(ABC):
    @staticmethod
```

```
    @abstractmethod
    def my_abstract_staticmethod(...):
        ...
```

@abc.abstractproperty

自从 3.3 版本不推荐使用： 现在可以使用 property ， property.getter() ， property.setter()和property.deleter()用 abstractmethod() ，使这个装饰是多余的。

内置的子类property()，表示抽象属性。

这种特殊情况已被废弃，因为property()装饰器现在在应用于抽象方法时被正确识别为抽象：

```
class C(ABC):
    @property
    @abstractmethod
    def my_abstract_property(self):
        ...
```

上面的例子定义了一个只读属性; 您还可以通过适当地将一个或多个基础方法标记为抽象来定义读写抽象属性：

```
class C(ABC):
    @property
    def x(self):
        ...

    @x.setter
    @abstractmethod
    def x(self, val):
        ...
```

如果只有一些组件是抽象的，那么只有那些组件需要更新才能在子类中创建具体属性：

```
class D(C):
    @C.x.setter
    def x(self, val):
        ...
```

该abc模块还提供以下功能：

abc.get_cache_token ( )

返回当前的抽象基类缓存标记。

标记是一个不透明的对象（支持相等性测试），标识虚拟子类的抽象基类缓存的当前版本。令牌随着ABCMeta.register()任何ABC上的每次呼叫而改变。

3.4版新增功能

脚注

[1] C ++程序员应该注意到Python的虚拟基类概念与C ++不同。

29.8。atexit- 退出处理程序

该atexit模块定义了注册和取消注册清理功能的函数。如此注册的功能在正常解释器终止时自动执行。 以与其注册*相反的*顺序atexit运行这些功能; 如果你注册，以及在解释终止时间，他们将在顺序运行，，。ABCCBA

注意：当程序被未被Python处理的信号杀死，检测到Python致命内部错误或被调用时，不会调用通过此模块注册的函数os._exit()。

atexit.**register**(*func* , ** args* , *** kwargs*)

注册*func*作为函数在终止时执行。任何要传递给*func的*可选参数都必须作为参数传递给register()。可以多次注册相同的函数和参数。

在正常的程序结束时（例如，如果sys.exit()被调用或主模块的执行完成），所有注册的函数将按照先进先出的顺序调用。假定较低级别的模块通常会在高级模块之前导入，因此必须稍后进行清理。

如果在执行退出处理程序期间发生异常，则会打印回溯（除非SystemExit引发）并且保存异常信息。在所有退出处理程序有机会运行之后，最后的异常将被重新提出。

这个函数返回*func*，这使得它可以用作装饰器。

atexit.**unregister**(*func*)

从解释器关闭时运行的函数列表中删除*func*。在调用之后unregister()，当解释器关闭时，*func*保证不被调用，即使它被注册了一次以上。 unregister()如果*func*之前没有注册，*它将*默默无闻。

也可以看看:

模 readline

atexit读取和写入readline历史文件的有用示例。

29.8.1。atexit示例

下面的简单示例演示了模块在导入时如何从文件初始化计数器，并在程序终止时自动保存计数器的更新值，而不依赖应用程序在终止时显式调用该模块。

```
try:
    with open("counterfile") as infile:
        _count = int(infile.read())
except FileNotFoundError:
    _count = 0

def incrcounter(n):
    global _count
    _count = _count + n
```

```
def savecounter():
    with open("counterfile", "w") as outfile:
        outfile.write("%d" % _count)

import atexit
atexit.register(savecounter)
```

定位和关键字参数也可以传递给`register()`被调用的被注册函数：

```
def goodbye(name, adjective):
    print('Goodbye, %s, it was %s to meet you.' % (name, adjective))

import atexit
atexit.register(goodbye, 'Donny', 'nice')

# or:
atexit.register(goodbye, adjective='nice', name='Donny')
```

用作装饰者：

```
import atexit

@atexit.register
def goodbye():
    print("You are now leaving the Python sector.")
```

这只适用于可以不带参数调用的函数。

29.9。 traceback- 打印或检索堆栈追溯

源代码： Lib / traceback.py

该模块提供了一个标准接口来提取，格式化和打印Python程序的堆栈跟踪。它完全模仿Python解释器在打印堆栈跟踪时的行为。当您想要在程序控制下打印堆栈跟踪时，例如在解释器周围的“包装器”中，这很有用。

该模块使用追溯对象 - 这是存储在sys.last_traceback变量中并作为第三项返回 的对象类型sys.exc_info()。

该模块定义了以下功能：

traceback.print_tb（*tb*，*limit* = *None*，*file* = *None* ）

如果*限制*为正值，则打印以*限制*跟踪对象*tb*（从调用者的帧开始）的堆栈跟踪条目。否则，打印最后的 条目。如果省略*限制*或者打印所有条目。如果*文件*被省略或者输出到了；否则它应该是一个打开的文件或文件类对象来接收输出。abs(limit)NoneNone sys.stderr

*在版本3.5中进行了更改：*添加了负面*限制*支持。

traceback.print_exception（*etype*，*value*，*tb*，*limit* = *None*，*file* = *None*，*chain* = *True* ）

从traceback对象*tb*打印异常信息和堆栈跟踪条目 到*文件*。这与print_tb()以下方面有所不同：

- 如果*tb*不是None，它会打印一个标题Traceback (most recent call last):
- 它在堆栈跟踪之后打印异常*etype*和*值*
- 如果*类型（值）*是SyntaxError且*值*具有适当的格式，则会打印语法错误发生的行，并在其中指示错误的大概位置。

可选的*限制*参数与for的含义相同print_tb()。如果*chain*为true（默认值），则会打印链式异常（异常的 属性__cause__或__context__属性），就像解释器打印未处理的异常时一样。

*改变在3.5版本：*该*VLAN的*参数将被忽略，并从类型推断*值*。

traceback.print_exc（*limit* = *None*，*file* = *None*，*chain* = *True* ）

这是一个简写。print_exception(*sys.exc_info(), limit, file, chain)

traceback.print_last（*limit* = *None*，*file* = *None*，*chain* = *True* ）

这是一个简写。一般而言，只有在例外达到交互式提示后才能使用（请参阅 ）。print_exception(sys.last_type, sys.last_value, sys.last_traceback, limit, file, chain)sys.last_type

traceback.print_stack（*f* =*无*，*限制*=*无*，*文件*=*无*）

如果*限制*为正，则打印以*限制*堆栈跟踪条目（从调用点开始）。否则，打印最后的 条目。如果省略*限制*或者打印所有条目。可选的*f*参数可用于指定要启动的备用堆栈帧。可选*文件*

参数与for具有相同的含义。 abs(limit) Noneprint_tb()

*在版本3.5中进行了更改：*添加了负面*限制*支持。

traceback. extract_tb (*tb* , *limit = None*)

返回从回溯对象*tb*中提取的“预处理”堆栈跟踪条目的列表。这对堆栈跟踪的替代格式非常有用。可选的*限制*参数与for的含义相同 print_tb()。“预处理”堆栈跟踪条目是一个4元组（*文件名*，*行号*，*函数名称*，*文本*），表示通常为堆栈跟踪打印的信息。该*文本*是一个带有前导和尾随空白字符的字符串; 如果源不可用，它是None。

traceback. extract_stack (*f =无* , *限制=无*)

从当前堆栈帧中提取原始回溯。返回值与格式相同extract_tb()。可选的*f*和*限制*参数与for具有相同的含义print_stack()。

traceback. format_list (*extracted_list*)

给出extract_tb() or extract_stack()返回的元组列表，返回一个准备打印的字符串列表。结果列表中的每个字符串对应于参数列表中具有相同索引的项目。每个字符串以换行符结束; 这些字符串也可以包含内部换行符，对于那些源文本行不是的项目 None。

traceback. format_exception_only (*etype* , *value*)

格式化追溯的异常部分。的参数是异常类型和值，如由下式给出 sys. last_type 和 sys. last_value。返回值是一个字符串列表，每个字符串都以换行符结尾。通常，该列表包含一个字符串; 但是，对于SyntaxError 例外情况，它包含几行（打印时）显示有关语法错误发生位置的详细信息。指示发生异常的消息是列表中总是最后一个字符串。

traceback. format_exception (*etype* , *value* , *tb* , *limit = None* , *chain = True*)

格式化堆栈跟踪和异常信息。参数与相应的参数具有相同的含义print_exception()。返回值是一串字符串，每个字符串以换行符结尾，一些字符串包含内部换行符。当这些行连接并打印时，打印的文本与打印的文本完全相同print_exception()。

*改变在3.5版本：*该*VLAN的*参数将被忽略，并从类型推断*值*。

traceback. format_exc (*limit = None* , *chain = True*)

这就像是print_exc(limit)返回一个字符串，而不是打印到一个文件。

traceback. format_tb (*tb* , *limit = None*)

速记。format_list(extract_tb(tb, limit))

traceback. format_stack (*f =无* , *限制=无*)

速记。format_list(extract_stack(f, limit))

traceback. clear_frames (*tb*)

通过调用每个帧对象的方法清除回溯*tb*中所有堆栈帧的局部变量clear()。

3.4版新增功能

traceback. walk_stack (*f*)

f.f_back从给定帧开始跟随堆栈，产生每个帧的帧和行号。如果*f*是None，则使用当前堆栈。这个帮助器用于StackSummary.extract()。

3.5版本中的新功能。

traceback.walk_tb（*tb*）
在tb_next产生每个帧的帧和行号之后进行回溯。这个帮助器用于StackSummary.extract()。

3.5版本中的新功能。

该模块还定义了以下类：

29.9.1。TracebackException对象

3.5版本中的新功能。

TracebackException对象是根据实际例外创建的，以轻量级方式捕获数据以供以后打印。

*类*traceback.TracebackException（*exc_type*，*exc_value*，*exc_traceback*，*，*limit* = *None*，*lookup_lines* = *True*，*capture_locals* = *False*）
捕获一个异常以便以后呈现。*限制*，*lookup_lines*和 *capture_locals*与StackSummary该类相同。

请注意，当捕捉到当地人时，它们也会显示在回溯中。

__cause__
一个TracebackException原始的__cause__。

__context__
一个TracebackException原始的__context__。

__suppress_context__
__suppress_context__来自原始异常的值。

stack
一个StackSummary代表回溯。

exc_type
原始回溯类。

filename
对于语法错误 - 发生错误的文件名。

lineno
对于语法错误 - 发生错误的行号。

text

对于语法错误 - 发生错误的文本。

offset
对于语法错误 - 发生错误的文本的偏移量。

msg
对于语法错误 - 编译器错误消息。

classmethod from_exception (*exc* , * , *limit* = *None* , *lookup_lines* = *True* , *capture_locals* = *False*)
捕获一个异常以便以后呈现。*限制* ，*lookup_lines*和 *capture_locals*与StackSummary该类相同。

请注意，当捕捉到当地人时，它们也会显示在回溯中。

format (*, *chain* = *True*)
格式化例外。

如果*链条*不是True，__cause__并且__context__不会被格式化。

返回值是一个字符串的生成器，每个字符串以换行符结尾，一些包含内部换行符。print_exception() 是这个方法的一个包装，它只是将行打印到一个文件中。

指示发生异常的消息始终是输出中的最后一个字符串。

format_exception_only ( )
格式化回溯的异常部分。

返回值是一个字符串生成器，每个字符串都以换行符结尾。

通常，发生器发出单个字符串; 但是，对于 SyntaxError例外情况，它会发出几行（打印时）显示有关语法错误发生位置的详细信息。

指示发生异常的消息始终是输出中的最后一个字符串。

29.9.2。StackSummary对象

3.5版本中的新功能。

StackSummary 对象表示准备格式化的调用堆栈。

*类*traceback. StackSummary

classmethod extract (*frame_gen* , * , *limit* = *None* , *lookup_lines* = *True* , *capture_locals* = *False*)
StackSummary从帧生成器构造一个对象（比如由walk_stack() or 返回walk_tb() ）。

如果提供了*限制*，则只能从*frame_gen*获取多个帧。如果*lookup_lines*是False，返回的FrameSummary 对象将不会读取它们的行，从而导致创建StackSummary更便宜的代价

（如果实际上可能没有格式化，这可能很有价值）。如果*capture_locals*是True每个局部变量FrameSummary被捕获为对象表示。

classmethod from_list（*a_list*）

StackSummary从提供的旧式元组列表构造一个对象。每个元组应该是一个4元组，文件名为lineno，name，line作为元素。

format（）

返回准备打印的字符串列表。结果列表中的每个字符串都对应于堆栈中的单个帧。每个字符串以换行符结束; 这些字符串也可以包含内部换行符，用于具有源文本行的项目。

对于同一帧和线条的长序列，显示前几个重复，然后显示一个汇总行，说明进一步重复的确切数量。

*在版本3.6中更改：*重复帧的长序列现在缩写。

29.9.3。FrameSummary对象

3.5版本中的新功能。

FrameSummary 对象表示回溯中的单个帧。

class traceback.FrameSummary（*filename*，*lineno*，*name*，*lookup_line* = *True*，*locals* = *None*，*line* = *None*）

表示正在格式化或打印的回溯或堆栈中的单个帧。它可以有一个包含在其中的帧本地化字符串版本。如果*lookup_line*是False，那么源代码FrameSummary在查询line 属性之前不会查找（当将其转换为元组时，也会发生这种情况）。 line可能会直接提供，并且会阻止行查找。*当地人*是一个可选的局部变量的字典，并且如果提供的变量的表示被存储在摘要供以后显示。

29.9.4。追溯示例

这个简单的例子实现了一个基本的读取 - 评估 - 打印循环，与标准的Python交互式解释器循环相似（但不如此有用）。有关解释器循环的更完整实现，请参阅code 模块。

```
import sys, traceback

def run_user_code(envdir):
    source = input(">>> ")
    try:
        exec(source, envdir)
    except Exception:
        print("Exception in user code:")
        print("-"*60)
        traceback.print_exc(file=sys.stdout)
        print("-"*60)

envdir = {}
```

```
while True:
    run_user_code(envdir)
```

以下示例演示了打印和格式化异常和回溯的不同方法：

```
import sys, traceback

def lumberjack():
    bright_side_of_death()

def bright_side_of_death():
    return tuple()[0]

try:
    lumberjack()
except IndexError:
    exc_type, exc_value, exc_traceback = sys.exc_info()
    print("*** print_tb:")
    traceback.print_tb(exc_traceback, limit=1, file=sys.stdout)
    print("*** print_exception:")
    # exc_type below is ignored on 3.5 and later
    traceback.print_exception(exc_type, exc_value, exc_traceback,
                              limit=2, file=sys.stdout)
    print("*** print_exc:")
    traceback.print_exc(limit=2, file=sys.stdout)
    print("*** format_exc, first and last line:")
    formatted_lines = traceback.format_exc().splitlines()
    print(formatted_lines[0])
    print(formatted_lines[-1])
    print("*** format_exception:")
    # exc_type below is ignored on 3.5 and later
    print(repr(traceback.format_exception(exc_type, exc_value,
                                          exc_traceback)))
    print("*** extract_tb:")
    print(repr(traceback.extract_tb(exc_traceback)))
    print("*** format_tb:")
    print(repr(traceback.format_tb(exc_traceback)))
    print("*** tb_lineno:", exc_traceback.tb_lineno)
```

示例的输出结果如下所示：

```
*** print_tb:
  File "<doctest...>", line 10, in <module>
    lumberjack()
*** print_exception:
Traceback (most recent call last):
  File "<doctest...>", line 10, in <module>
    lumberjack()
  File "<doctest...>", line 4, in lumberjack
    bright_side_of_death()
IndexError: tuple index out of range
*** print_exc:
Traceback (most recent call last):
  File "<doctest...>", line 10, in <module>
    lumberjack()
  File "<doctest...>", line 4, in lumberjack
```

```
    bright_side_of_death()
IndexError: tuple index out of range
*** format_exc, first and last line:
Traceback (most recent call last):
IndexError: tuple index out of range
*** format_exception:
['Traceback (most recent call last):\n',
 '  File "<doctest...>", line 10, in <module>\n    lumberjack()\n',
 '  File "<doctest...>", line 4, in lumberjack\n    bright_side_of_death()\n',
 '  File "<doctest...>", line 7, in bright_side_of_death\n    return tuple()[0]\n',
 'IndexError: tuple index out of range\n']
*** extract_tb:
[<FrameSummary file <doctest...>, line 10 in <module>>,
 <FrameSummary file <doctest...>, line 4 in lumberjack>,
 <FrameSummary file <doctest...>, line 7 in bright_side_of_death>]
*** format_tb:
['  File "<doctest...>", line 10, in <module>\n    lumberjack()\n',
 '  File "<doctest...>", line 4, in lumberjack\n    bright_side_of_death()\n',
 '  File "<doctest...>", line 7, in bright_side_of_death\n    return tuple()[0]\n']
*** tb_lineno: 10
```

以下示例显示了打印和格式化堆栈的不同方法：

```
>>> import traceback
>>> def another_function():
...     lumberstack()
...
>>> def lumberstack():
...     traceback.print_stack()
...     print(repr(traceback.extract_stack()))
...     print(repr(traceback.format_stack()))
...
>>> another_function()
  File "<doctest>", line 10, in <module>
    another_function()
  File "<doctest>", line 3, in another_function
    lumberstack()
  File "<doctest>", line 6, in lumberstack
    traceback.print_stack()
[('<doctest>', 10, '<module>', 'another_function()'),
 ('<doctest>', 3, 'another_function', 'lumberstack()'),
 ('<doctest>', 7, 'lumberstack', 'print(repr(traceback.extract_stack()))')]
['  File "<doctest>", line 10, in <module>\n    another_function()\n',
 '  File "<doctest>", line 3, in another_function\n    lumberstack()\n',
 '  File "<doctest>", line 8, in lumberstack\n    print(repr(traceback.format_stack()))\n']
```

最后一个例子演示了最后几个格式化函数：

```
>>> import traceback
>>> traceback.format_list([('spam.py', 3, '<module>', 'spam.eggs()'),
...                        ('eggs.py', 42, 'eggs', 'return "bacon"')])
['  File "spam.py", line 3, in <module>\n    spam.eggs()\n',
 '  File "eggs.py", line 42, in eggs\n    return "bacon"\n']
>>> an_error = IndexError('tuple index out of range')
```

```
>>> traceback.format_exception_only(type(an_error), an_error)
['IndexError: tuple index out of range\n']
```

29.10。 __future__-未来的语句定义

源代码： Lib / __ future__.py

__future__ 是一个真正的模块，并有三个目的：

- 为了避免混淆分析导入语句的现有工具并期望找到它们导入的模块。
- 为确保将来的语句在2.1之前的版本中运行，至少会产生运行时异常（导入__future__将失败，因为在2.1之前没有该名称的模块）。
- 记录何时引入了不兼容的变更，以及何时将变为强制变更。这是可执行文档的一种形式，可以通过导入__future__并检查其内容以编程方式进行检查。

每个陈述__future__.py的形式如下：

```
FeatureName = _Feature(OptionalRelease, MandatoryRelease,
                       CompilerFlag)
```

其中，通常*OptionalRelease*小于*MandatoryRelease*，并且都是与以下形式相同的5元组sys.version_info：

```
(PY_MAJOR_VERSION, # the 2 in 2.1.0a3; an int
 PY_MINOR_VERSION, # the 1; an int
 PY_MICRO_VERSION, # the 0; an int
 PY_RELEASE_LEVEL, # "alpha", "beta", "candidate" or "final"; string
 PY_RELEASE_SERIAL # the 3; an int
)
```

*OptionalRelease*记录功能被接受的第一个版本。

在*MandatoryRelease*尚未发生的情况下，*MandatoryRelease*预测该特征将成为该语言的一部分。

其他*强制性发布*记录当该特征成为该语言的一部分时; 在此处或之后的发行版中，模块不再需要将来的语句来使用有问题的功能，但可能会继续使用此类导入。

*强制释放*也可能是None，这意味着计划中的功能被删除。

类的实例_Feature有两个相应的方法，getOptionalRelease()并且getMandatoryRelease()。

*CompilerFlag*是应该在内建函数的第四个参数中传递的（位域）标志，compile()以在动态编译的代码中启用该功能。该标志存储在实例的compiler_flag 属性中_Feature。

不会从中删除任何功能说明__future__。自Python 2.1推出以来，使用这种机制已经找到了进入该语言的途径：

特征	可选	强制性的	影响
nested_scopes	2.1.0b1	2.2	**PEP 227**： *静态嵌套示波器*
发电机	2.2.0a1	2.3	**PEP 255**： *简单的发电机*
师	2.2.0a2	3.0	**PEP 238**： *更改部门操作员*

特征	可选	强制性的	影响
absolute_import	2.5.0a1	3.0	**PEP 328**： *进口：多线和绝对/相对*
with_statement	2.5.0a1	2.6	**PEP 343**： *“带”声明*
print_function	2.6.0a2	3.0	**PEP 3105**： *使打印功能*
unicode_literals	2.6.0a2	3.0	**PEP 3112**： *Python 3000中的字节文字*
generator_stop	3.5.0b1	3.7	**PEP 479**： *发电机内的StopIteration处理*

也可以看看:

未来的发言
编译器如何处理未来的导入。

29.11。 gc- 垃圾收集器接口

该模块为可选的垃圾收集器提供了一个接口。它提供了禁用收集器，调整收集频率并设置调试选项的功能。它还提供对收集器找到但不能释放的不可访问对象的访问。由于收集器补充了Python中已经使用的引用计数，所以如果您确定程序没有创建引用循环，则可以禁用收集器。通过调用可以禁用自动收集 gc.disable() 。调试泄漏的程序调用 gc.set_debug(gc.DEBUG_LEAK)。请注意，这包括将 gc.DEBUG_SAVEALL垃圾收集的对象保存在gc.garbage中以供检查。

该gc模块提供以下功能：

gc.enable ( )
启用自动垃圾收集。

gc.disable ( )
禁用自动垃圾收集。

gc.isenabled ( )
如果启用自动收集，则返回true。

gc.collect (*generation = 2*)
没有参数，运行一个完整的集合。可选参数的*生成*可以是一个整数，指定要收集哪一代（从0到2）。ValueError如果世代号无效，则引发A. 返回找到的不可到达对象的数量。

每当完整收集或收集最高一代（2）时，为许多内置类型维护的空闲列表都会被清除。由于特定的实现，并非所有空闲列表中的项目都可以被释放，特别是float。

gc.set_debug (*标志*)
设置垃圾收集调试标志。调试信息将被写入sys.stderr。请参阅下面的调试标志列表，这些标志可以使用位操作进行组合来控制调试。

gc.get_debug ( )
返回当前设置的调试标志。

gc.get_objects ( )
返回收集器跟踪的所有对象的列表，不包括返回的列表。

gc.get_stats ( )
从解释器启动后返回包含收集统计信息的三个每代字典的列表。未来键的数量可能会发生变化，但目前每个字典将包含以下项目：

- collections 是这一代人收集的次数;
- collected 是这代人收集的物体总数;
- uncollectable是garbage 在这一代人内被发现无法收集（因此被移到列表中）的对象总数。

3.4版新增功能

gc.set_threshold(*threshold0*[, *threshold1*[, *threshold2*]])
设置垃圾收集阈值（收集频率）。将*threshold0*设置为零将禁用收集。

气相色谱将物体分成三代，具体取决于它们存活了多少次收集扫描。新的对象被放置在最年轻的一代（一代0）。如果一个物体在一个集合中存活，它就会被移入下一代老一代。由于世代2是最古老的一代，所以这一代的物品在收藏后仍然存在。为了决定何时运行，收集器会跟踪自上次收集以来的数字对象分配和释放。当分配数量减去释放次数超过*阈值0时*，开始收集。最初只0检查一代。如果世代0以来已经超过*阈值1*次检查1已经过检查，然后检查一代1。同样，*阈值2* 控制生成1之前生成的集合的数量2。

gc.get_count()
将当前集合计数作为一个元组返回。(count0, count1, count2)

gc.get_threshold()
将当前收集阈值作为一个元组返回。(threshold0, threshold1, threshold2)

gc.get_referrers(**objs*)
返回直接引用任何objs的对象列表。这个函数只会定位那些支持垃圾收集的容器; 不会发现引用其他对象但不支持垃圾回收的扩展类型。

请注意，已经被解除引用的对象，但是生活在周期中并且尚未被垃圾收集器收集的对象可以列在所产生的引用者中。要仅获取当前活动对象，请collect()在调用之前调用get_referrers()。

使用返回的物品时必须小心，get_referrers()因为其中一些物品可能仍在施工中，因此处于暂时无效的状态。get_referrers()除了调试以外，请避免使用其他用途。

gc.get_referents(**objs*)
返回任何参数直接引用的对象列表。返回的对象是由参数的C级tp_traverse方法（如果有的话）访问的那些对象，并且可能不是实际可以直接访问的所有对象。tp_traverse方法仅由支持垃圾收集的对象支持，并且只需要访问可能涉及周期的对象。因此，例如，如果一个整数可以直接从参数中访问，那么整数对象可能会或可能不会出现在结果列表中。

gc.is_tracked(*obj*)
True如果对象当前由垃圾回收器跟踪，则返回; False否则返回。作为一般规则，不跟踪原子类型的实例，并且非原子类型（容器，用户定义的对象...）的实例。但是，为了抑制简单实例的垃圾回收器占用空间（例如只包含原子键和值的字符串），可以存在一些类型特定的优化：

```
>>> gc.is_tracked(0)
False
>>> gc.is_tracked("a")
False
>>> gc.is_tracked([])
True
>>> gc.is_tracked({})
False
>>> gc.is_tracked({"a": 1})
False
```

```
>>> gc.is_tracked({"a": []})
True
```

版本3.1中的新功能。

为只读访问提供了以下变量（您可以改变这些值但不应重新绑定它们）：

gc.garbage
收集器发现无法释放但无法释放的对象列表（不可收集的对象）。从Python 3.4开始，除了使用具有非NULL tp_del插槽的C扩展类型的实例时，此列表大部分时间应为空。

如果DEBUG_SAVEALL设置，则所有不可访问的对象将被添加到此列表中而不是释放。

*在版本3.2中更改：*如果解释器关闭时该列表非空， ResourceWarning则发出a，默认情况下为无提示。如果 DEBUG_UNCOLLECTABLE设置，另外打印所有不可收集的对象。

*版本3.4中更改：*以下**PEP 442**，一个__del__()方法的对象不会再终结gc.garbage了。

gc.callbacks
垃圾收集器在收集之前和之后将调用的回调列表。回调将被调用两个参数， *阶段*和*信息*。

*阶段*可以是以下两个值之一：

“开始”：垃圾收集即将开始。

“停止”：垃圾收集已完成。

*info*是一个为回调提供更多信息的词典。以下键目前定义：

“一代”：收集最古老的一代。

“收集”：当*阶段*是“停止”时，成功收集的对象数量。

“无法收集”：当*阶段*是“停止”时，无法收集并投入的对象数量garbage。

应用程序可以添加自己的回调到这个列表。主要使用案例是：

收集有关垃圾收集的统计信息，例如收集各个世代的频率以及收集需要多长时间。

允许应用程序在出现时标识并清除自己的不可收集类型garbage。

3.3版本的新功能

提供以下常量用于set_debug()：

gc.DEBUG_STATS
在收集期间打印统计。调整收集频率时，此信息可能很有用。

gc.DEBUG_COLLECTABLE
打印找到可收集对象的信息。

gc.DEBUG_UNCOLLECTABLE

打印找到的不可收集对象的信息（不可获取但不能被收集器释放的对象）。这些对象将被添加到garbage列表中。

*在版本3.2中更改：*如果解释器关闭，也打印garbage列表 的内容（如果它不为空）。

gc.DEBUG_SAVEALL

设置时，找到的所有不可访问的对象将被追加到*垃圾中*，而不是被释放。这对调试泄漏程序很有用。

gc.DEBUG_LEAK

收集器需要打印有关泄漏程序（等于）的信息的调试标志。DEBUG_COLLECTABLE | DEBUG_UNCOLLECTABLE | DEBUG_SAVEALL

29.12。 inspect- 检查活物

源代码： Lib / inspect.py

该inspect模块提供了几个有用的功能来帮助获取有关活动对象的信息，例如模块，类，方法，函数，回溯，框架对象和代码对象。例如，它可以帮助您检查类的内容，检索方法的源代码，提取并格式化函数的参数列表，或者获取显示详细回溯所需的所有信息。

这个模块提供了四种主要的服务：类型检查，获取源代码，检查类和函数以及检查解释器堆栈。

29.12.1。类型和成员

该getmembers()函数检索诸如类或模块之类的对象的成员。主要提供名称以“is”开头的函数作为第二个参数的方便选择getmembers()。他们还帮助您确定何时可以期望找到以下特殊属性：

类型	属性	描述
模	__doc__	文档字符串
	__文件__	文件名（内置模块缺失）
类	__doc__	文档字符串
	__名称__	这个类被定义的名称
	__qualname__	合格的名字
	__module__	在这个类中定义的模块的名称
方法	__doc__	文档字符串
	__名称__	这个方法被定义的名称
	__qualname__	合格的名字
	__func__	包含方法实现的函数对象
	__自__	该方法绑定到的实例，或 None
功能	__doc__	文档字符串
	__名称__	这个函数被定义的名字
	__qualname__	合格的名字
	__码__	包含已编译函数字节码的代码对象
	__defaults__	用于位置或关键字参数的任何默认值的元组
	__kwdefaults__	为关键字参数映射任何默认值
	__globals__	在其中定义了该函数的全局名称空间
	__annotations__	参数名称映射到注释; "return"键保留用于返回注释
追溯	tb_frame	在这个级别的框架对象
	tb_lasti	字节码中最后一次尝试指令的索引
	tb_lineno	Python源代码中的当前行号
	tb_next	下一个内部追踪对象（由此级别调用）

类型	属性	描述
帧	f_back	下一个外框对象（这个框架的调用者）
	f_builtins	建立了这个框架所看到的名字空间
	f码	代码对象在这个框架中被执行
	f_globals	全局名称空间由此框架看到
	f_lasti	字节码中最后一次尝试指令的索引
	f_lineno	Python源代码中的当前行号
	f_locals	本框架所看到的本地名称空间
	f_restricted	如果帧处于受限执行模式，则为0或1
	f_trace	追踪此帧的功能，或 None
码	co_argcount	参数数量（不包括仅关键字参数，*或**参
	co_code	原始编译的字节码字符串
	co_cellvars	单元格变量的名称元组（由包含范围引用
	co_consts	字节码中使用的常量元组
	co_filename	此代码对象在其中创建的文件的名称
	co_firstlineno	Python源代码中的第一行数
	co_flags	CO_*标志的位图，在这里阅读更多
	co_lnotab	将行号编码为字节码索引
	co_freevars	自由变量名称的元组（通过函数的闭包来引用
	co_kwonlyargcount	仅包含关键字的参数（不包括** arg）
	co_name	这个代码对象被定义的名称
	co_names	局部变量名称的元组
	co_nlocals	局部变量的数量
	co_stacksize	所需的虚拟机堆栈空间
	co_varnames	参数名称和局部变量的元组
发电机	__名称__	名称
	__qualname__	合格的名字
	gi_frame	帧
	gi_running	发电机正在运转吗？
	gi_code	码
	gi_yieldfrom	对象被迭代 ，或 yield fromNone
协程	__名称__	名称
	__qualname__	合格的名字
	cr_await	对象正在等待，或 None
	cr_frame	帧
	cr_running	协程运行？
	cr_code	码
内置	__doc__	文档字符串
	__名称__	此功能或方法的原始名称
	__qualname__	合格的名字

	__自__	一个方法绑定到的实例，或者 None

*在版本3.5中更改：*向生成器添加__qualname__和gi_yieldfrom属性。

__name__现在可以从函数名称中设置生成器的属性，而不是代码名称，现在可以修改它。

inspect.**getmembers**(*object*[, *predicate*])

返回按名称排序的（名称，值）对列表中的所有对象成员。如果提供了可选*谓词*参数，则只包含谓词返回真值的成员。

> **注意：** getmembers()当参数是一个类并且这些属性已经在元类的自定义中列出时，将只返回元类中定义的类属性__dir__()。

inspect.**getmodulename**(*路径*)

返回由文件*路径*命名的模块的名称，不包括封装包的名称。文件扩展名将根据所有条目进行检查importlib.machinery.all_suffixes()。如果匹配，则返回最终路径组件，并删除扩展名。否则，None返回。

请注意，此函数*仅为*实际Python模块返回一个有意义的名称 - 可能引用Python包的路径仍将返回None。

*在版本3.3中更改：*该函数直接基于importlib。

inspect.**ismodule**(*object*)

如果对象是模块，则返回true。

inspect.**isclass**(*object*)

如果对象是一个类，无论是内置的还是用Python代码创建的，则返回true。

inspect.**ismethod**(*object*)

如果对象是用Python编写的绑定方法，则返回true。

inspect.**isfunction**(*object*)

如果对象是Python函数，则返回true，该函数包含由lambda表达式创建的函数。

inspect.**isgeneratorfunction**(*object*)

如果对象是Python生成器函数，则返回true。

inspect.**isgenerator**(*object*)

如果对象是一个生成器，则返回true。

inspect.**iscoroutinefunction**(*object*)

如果对象是协程函数（使用语法定义的函数），则返回true。 async def

3.5版本中的新功能。

inspect.**iscoroutine**(*object*)

如果对象是由函数创建 的协程，则返回true 。 async def

3.5版本中的新功能。

inspect.**isawaitable**(*object*)

如果对象可用于await表达式，则返回true 。

也可以用来区分基于生成器的协程和常规生成器：

```
def gen():
    yield
@types.coroutine
def gen_coro():
    yield

assert not isawaitable(gen())
assert isawaitable(gen_coro())
```

3.5版本中的新功能。

inspect.**isasyncgenfunction**(*object*)

如果对象是异步生成器函数，则返回true ，例如：

```
>>> async def agen():
...     yield 1
...
>>> inspect.isasyncgenfunction(agen)
True
```

3.6版本中的新功能。

inspect.**isasyncgen**(*object*)

如果对象是一个返回true 异步发电机迭代 通过创建异步发电机的功能。

3.6版本中的新功能。

inspect.**istraceback**(*object*)

如果对象是回溯，则返回true。

inspect.**isframe**(*object*)

如果对象是一个框架，则返回true。

inspect.**iscode**(*object*)

如果对象是代码，则返回true。

inspect.**isbuiltin**(*object*)

如果对象是内置函数或绑定的内置方法，则返回true。

inspect.**isroutine**(*object*)

如果对象是用户定义的或内置的函数或方法，则返回true。

inspect.**isabstract**(*object*)

如果对象是抽象基类，则返回true。

inspect.**ismethoddescriptor**(*object*)

如果对象是一个方法描述符返回true，但如果 ismethod() ， isclass() ， isfunction()或者 isbuiltin() 是真实的。

例如，这是真的int.__add__。一个通过这个测试的对象有一个__get__()方法，但不是一个__set__() 方法，但是除此之外，这组属性是变化的。一个 __name__属性通常是合理的，__doc__通常是。

通过也通过其他测试之一的描述符实现的方法从测试中返回false ismethoddescriptor() ，仅仅因为其他测试承诺更多 - 例如，可以指望在__func__对象通过时具有 属性（ etc ） ismethod()。

inspect.**isdatadescriptor**(*object*)

如果对象是数据描述符，则返回true。

数据描述符有一个__get__和一个__set__方法。示例是属性（在Python中定义），getset和成员。后两种是在C中定义的，并且这些类型有更多特定的测试可用，这在Python实现中是健壮的。典型地，数据描述符也将具有__name__与__doc__属性（属性，getsets，和成员同时具有这些属性的），但是这不被保证。

inspect.**isgetsetdescriptor**(*object*)

如果对象是getset描述符，则返回true。

CPython实现细节： getsets是通过PyGetSetDef结构在扩展模块中定义的属性 。对于没有这种类型的Python实现，此方法将始终返回False。

inspect.**ismemberdescriptor**(*object*)

如果对象是成员描述符，则返回true。

CPython实现细节：成员描述符是通过PyMemberDef结构在扩展模块中定义的属性 。对于没有这种类型的Python实现，此方法将始终返回False。

29.12.2。检索源代码

inspect.**getdoc**(*object*)

获取一个对象的文档字符串，并进行清理cleandoc()。如果未提供对象的文档字符串，并且该对象是类，方法，属性或描述符，则从继承层次结构中检索文档字符串。

*在版本3.5中进行了更改：*如果不覆盖文档字符串，它们现在会继承。

inspect.**getcomments**(*object*)

在单个字符串中返回紧接在对象源代码之前（对于类，函数或方法）或Python源文件顶部（如果对象是模块）的任何注释行。如果对象的源代码不可用，则返回None。如果该对象已在C或交互式shell中定义，则可能发生这种情况。

inspect.**getfile**(*object*)

返回定义了对象的（文本或二进制）文件的名称。TypeError如果对象是内置的模块，类或函数，这将失败。

inspect.**getmodule**(*object*)

尝试猜测某个对象是在哪个模块中定义的。

inspect.**getsourcefile**(*object*)

返回定义了对象的Python源文件的名称。TypeError如果对象是内置的模块，类或函数，这将失败。

inspect.**getsourcelines**(*object*)

返回一个对象的源代码行和起始行号的列表。参数可以是模块，类，方法，函数，追溯，框架或代码对象。源代码作为与对象相对应的行列表返回，行号指示在原始源文件中找到第一行代码的位置。一个OSError如果源代码不能被检索上升。

*在版本3.3中进行了更改：*OSError而不是IOError现在是前者的别名。

inspect.**getsource**(*object*)

返回一个对象的源代码文本。参数可以是模块，类，方法，函数，追溯，框架或代码对象。源代码作为单个字符串返回。一个OSError如果源代码不能被检索上升。

*在版本3.3中进行了更改：*OSError而不是IOError现在是前者的别名。

inspect.**cleandoc**(*doc*)

清除缩排以排列代码块的文档字符串中的缩进。

从第一行删除所有主要的空白字符。任何可从第二行向前均匀删除的空白字符将被删除。随后删除开头和结尾的空行。此外，所有选项卡都扩展为空格。

29.12.3。使用Signature对象　　可调用的对象

3.3版本的新功能

Signature对象表示可调用对象的调用签名及其返回注释。要检索签名对象，请使用该signature()函数。

inspect.**signature**(*callable*, *, *follow_wrapped* = *True*)

返回Signature给定的对象callable：

```
>>> from inspect import signature
>>> def foo(a, *, b:int, **kwargs):
...     pass

>>> sig = signature(foo)

>>> str(sig)
'(a, *, b:int, **kwargs)'
```

```
>>> str(sig.parameters['b'])
'b:int'

>>> sig.parameters['b'].annotation
<class 'int'>
```

接受各种各样的python callables，从普通函数和类到 functools.partial()对象。

ValueError如果不提供签名，并且 TypeError不支持该类型的对象，则引发。

3.5 版新增： follow_wrapped 参数。通过特定 False 的签名 callable（callable.__wrapped__不会用于打开装饰的可卡因）。

注意： 在Python的某些实现中，某些可调用对象可能不是内省的。例如，在CPython中，C中定义的一些内置函数不提供关于它们参数的元数据。

*类*inspect.Signature（*参数= None，*，return_annotation = Signature.empty* ）

Signature对象表示函数的调用签名及其返回注释。对于函数接受的每个参数，它将一个Parameter对象存储 在其parameters集合中。

可选*参数*参数是一系列Parameter 对象，经过验证，可以检查是否没有重名的参数，并且参数的顺序是正确的，即首先是位置 - 然后是位置 - 或 - 关键字，而参数默认值遵循没有默认值的参数。

可选的*return_annotation*参数可以是一个任意的Python对象，是可调用对象的“返回”注释。

签名对象是*不可变的*。使用Signature.replace()作出修改后的副本。

*在版本3.5中更改：*签名对象是可挑选和可排序的。

empty

一种特殊的级别标记，用于指定缺少返回注释。

parameters

参数名称到相应Parameter对象的有序映射 。

return_annotation

可调用的“返回”注释。如果可调用对象没有“返回”注释，则将此属性设置为Signature.empty。

bind（**args*，***kwargs* ）

从位置和关键字参数创建一个映射到参数。返回BoundArguments如果*args并**kwargs匹配签名，或者提出一个TypeError。

bind_partial（**args*，***kwargs* ）

以与之相同的方式工作Signature.bind() ，但允许省略一些必需的参数（模仿functools.partial()行为）。返回BoundArguments ，或者TypeError如果传递的参数与签名不匹配，则返回a 。

replace（**[，parameters][，return_annotation]* ）

根据调用的实例替换创建新的Signature实例。可以传递不同的parameters和/或return_annotation覆盖基本签名的相应属性。要从复制的Signature中删除return_annotation，请传入Signature.empty。

```
>>> def test(a, b):
...     pass
>>> sig = signature(test)
>>> new_sig = sig.replace(return_annotation="new return anno")
>>> str(new_sig)
"(a, b) -> 'new return anno'"
```

classmethod from_callable(*obj*, *, *follow_wrapped* = *True*)

Signature为给定的可调用对象返回一个（或其子类）对象obj。通过follow_wrapped=False签名obj而不打开__wrapped__链条。

该方法简化了以下子类Signature：

```
class MySignature(Signature):
    pass
sig = MySignature.from_callable(min)
assert isinstance(sig, MySignature)
```

3.5版本中的新功能。

class inspect.Parameter(*name*, *kind*, *, *default* = *Parameter.empty*, *annotation* = *Parameter.empty*)

参数对象是*不可变的*。您可以使用Parameter.replace()创建修改的副本来代替修改Parameter对象。

*在版本3.5中更改：*参数对象是可选择的并且可哈希的。

empty

一种特殊的级别标记，用于指定缺省值和注释的缺失。

name

参数的名称作为字符串。该名称必须是有效的Python标识符。

CPython实现细节： CPython.0在用于实现理解和生成器表达式的代码对象上生成表单的隐式参数名称。
*在版本3.6中进行了更改：*这些参数名称由此模块公开为类似名称implicit0。

default

参数的默认值。如果该参数没有默认值，则该属性设置为Parameter.empty。

annotation

参数的注释。如果该参数没有注释，则该属性设置为Parameter.empty。

kind

描述参数值如何绑定到参数。可能的值（可通过Parameter，如Parameter.KEYWORD_ONLY）：

名称	含义
POSITIONAL_ONLY	值必须作为位置参数提供。 Python没有明确的定义位置参数的语法，但是许多内置和扩展模块函数（特别是只接受一个或两个参数的函数）接受它们。
POSITIONAL_OR_KEYWORD	值可以作为关键字或位置参数提供（这是用Python实现的函数的标准绑定行为。）
VAR_POSITIONAL	位置参数的元组没有绑定到任何其他参数。这对应*args于Python函数定义中的参数。
KEYWORD_ONLY	值必须作为关键字参数提供。仅限关键字参数是在Python函数定义中出现*或*args输入的参数。
VAR_KEYWORD	没有绑定到任何其他参数的关键字参数的字典。这对应**kwargs于Python函数定义中的 参数。

例如：打印没有默认值的所有关键字参数：

```
>>> def foo(a, b, *, c, d=10):
...     pass

>>> sig = signature(foo)
>>> for param in sig.parameters.values():
...     if (param.kind == param.KEYWORD_ONLY and
...                         param.default is param.empty):
...         print('Parameter:', param)
Parameter: c
```

replace (**[, name] [, kind] [, default] [, annotation]*)

根据被调用的实例替换创建一个新的Parameter实例。要覆盖某个Parameter属性，请传递相应的参数。要从参数中删除默认值或/和注释，请传递Parameter.empty。

```
>>> from inspect import Parameter
>>> param = Parameter('foo', Parameter.KEYWORD_ONLY, default=42)
>>> str(param)
'foo=42'

>>> str(param.replace()) # Will create a shallow copy of 'param'
'foo=42'

>>> str(param.replace(default=Parameter.empty, annotation='spam'))
"foo:'spam'"
```

*在版本3.4中更改：*在Python 3.3参数对象被允许name设置为None如果它们kind被设置为POSITIONAL_ONLY。这不再被允许。

*类*inspect.BoundArguments

a Signature.bind()或Signature.bind_partial()呼叫的结果。保存参数映射到函数的参数。

arguments

collections.OrderedDict参数'名称到参数'值的有序，可变映射（ ）。仅包含显式绑定的参数。 arguments意志的变化反映在args和 kwargs。

应该与Signature.parameters任何参数处理目的一起使用。

> **注意：** 其中Signature.bind()或 Signature.bind_partial()依赖于默认值的参数被跳过。但是，如果需要，请使用BoundArguments.apply_defaults()添加它们。

args

位置参数值的元组。从arguments属性动态计算 。

kwargs

关键字参数值的字典。从arguments属性动态计算 。

signature

对父Signature对象的引用。

apply_defaults ( )

为缺少的参数设置默认值。

对于变量位置参数（ *args ），默认是一个空元组。

对于变量关键字参数（ **kwargs ），默认是一个空字典。

```
>>> def foo(a, b='ham', *args): pass
>>> ba = inspect.signature(foo).bind('spam')
>>> ba.apply_defaults()
>>> ba.arguments
OrderedDict([('a', 'spam'), ('b', 'ham'), ('args', ())])
```

3.5版本中的新功能。

的args和kwargs特性可被用于调用功能：

```
def test(a, *, b):
    ...

sig = signature(test)
ba = sig.bind(10, b=20)
test(*ba.args, **ba.kwargs)
```

也可以看看:

PEP 362 - 功能签名对象。
详细的规范，实现细节和示例。

29.12.4。类和函数

inspect.getclasstree（*classes*，*unique* = *False* ）

将给定的类列表排列成嵌套列表的层次结构。在出现嵌套列表的情况下，它包含从其条目紧接在列表之前的类派生的类。每个条目都是一个2元组，其中包含一个类和其基类的元组。如果*唯一*参数为true，则给定列表中的每个类的返回结构中只会出现一个条目。否则，使用多重继承的类和它们的后代将多次出现。

inspect.getargspec（*func* ）

获取Python函数参数的名称和默认值。一个 命名的元组 被返回。*args*是参数名称的列表。*可变参数*和*关键字* 是和参数或的名称。*默认值*是默认参数值的元组，或者没有默认参数;如果这个元组有*n* 个元素，它们对应于*args* 中列出的最后 *n* 个元素。 ArgSpec(args, varargs, keywords, defaults)***NoneNone

*自从3.0版本不推荐使用：*使用getfullargspec()一个更新的API，它通常是一个简易替换，而且正确处理功能注释，只有关键字参数。

或者，使用signature()和 Signature对象，为可调参数提供更加结构化的自检API。

inspect.getfullargspec（*func* ）

获取Python函数参数的名称和默认值。一个 命名的元组被返回：

FullArgSpec(args, varargs, varkw, defaults, kwonlyargs, kwonlydefaults, annotations)

*args*是位置参数名称的列表。 *varargs*是*参数*的名称，*或者None是否接受任意的位置参数。 *varkw*是**参数的名称，或者None是否接受任意的关键字参数。 *默认值*是与最后*n*个位置参数相对应的默认参数值的*n*元组，或者是否没有定义这样的默认值。 *kwonlyargs*是关键字参数名称的列表。 *kwonlydefaults*是一个将参数名称从*kwonlyargs*映射 到缺省值的字典，如果没有参数提供。 None *注释*是将参数名称映射到注释的字典。该特殊键"return"用于报告函数返回值注释（如果有的话）。

请注意，signature()并 签名对象提供可调用的自省推荐的API，并支持其他行为（如位置-only参数），其在扩展模块的API有时会遇到。该功能主要用于需要保持与Python 2 inspect模块API 兼容性的代码。

*在版本3.4中更改：*此函数现在基于signature()，但仍然忽略 __wrapped__属性，并在绑定方法的签名输出中包含已绑定的第一个参数。

*在版本3.6中进行了更改：*此方法以前被记录为signature()在Python 3.5中被弃用 ，但该决定已被逆转，以便为单源Python 2/3代码从遗留getargspec() API 迁移而恢复明确支持的标准接口 。

inspect.getargvalues（*框架*）

获取有关传递到特定框架的参数的信息。一个 命名的元组 被返回。*args*是参数名称的列表。 *可变参数*和*关键字* 是的名称和参数或。 *当地人*是给定框架的当地人字典。ArgInfo(args, varargs, keywords, locals)***None

注意： 这个函数被无意中标记为Python 3.5中的弃用。

inspect.**formatargspec**(*args*[, *varargs*, *varkw*, *defaults*, *kwonlyargs*, *kwonlydefaults*, *annotations*[, *formatarg*, *formatvarargs*, *formatvarkw*, *formatvalue*, *formatreturns*, *formatannotations*]])

从返回的值格式化漂亮的参数规范 `getfullargspec()`。

前七个参数是（`args`，`varargs`，`varkw`，`defaults`，`kwonlyargs`，`kwonlydefaults`，`annotations`）。

其他六个参数分别是调用参数名称，*参数名称，**参数名称，默认值，返回注释和单个注释到字符串的函数。

例如：

```
>>> from inspect import formatargspec, getfullargspec
>>> def f(a: int, b: float):
...     pass
...
>>> formatargspec(*getfullargspec(f))
'(a: int, b: float)'
```

*自3.5版以来不推荐使用：*使用`signature()`和 签名对象，这为可调参数提供了更好的内省API。

inspect.**formatargvalues**(*args*[, *varargs*, *varkw*, *locals*, *formatarg*, *formatvarargs*, *formatvarkw*, *formatvalue*])

根据返回的四个值格式化漂亮的参数规范 `getargvalues()`。格式*参数是相应的可选格式函数，被称为将名称和值转换为字符串。

注意： 这个函数被无意中标记为Python 3.5中的弃用。

inspect.**getmro**(*cls*)

以方法解析顺序返回cls基类的元组，包括cls。这个元组中不会出现多次类。请注意，方法解析顺序取决于cls的类型。除非使用非常独特的用户定义的元类型，否则cls将是元组的第一个元素。

inspect.**getcallargs**(*func*, **args*, ***kwds*)

将*args*和*kwds*绑定到Python函数或方法*func*的参数名称，就好像它们是用它们调用的一样。对于绑定方法，还将第一个参数（通常是named `self`）绑定到关联的实例。返回一个字典，将参数名称（包括参数*和 **参数的名称，如果有的话）映射到*args*和*kwds*的值。在错误地调用*func*的情况下，即每当由于签名不兼容而引发异常时，会引发相同类型和相同或类似消息的异常。例如：`func(*args, **kwds)`

```
>>> from inspect import getcallargs
>>> def f(a, b=1, *pos, **named):
...     pass
>>> getcallargs(f, 1, 2, 3) == {'a': 1, 'named': {}, 'b': 2, 'pos': (3,)}
```

```
True
>>> getcallargs(f, a=2, x=4) == {'a': 2, 'named': {'x': 4}, 'b': 1, 'pos': ()}
True
>>> getcallargs(f)
Traceback (most recent call last):
...
TypeError: f() missing 1 required positional argument: 'a'
```

3.2版本中的新功能

*自从3.5版本不推荐使用：*使用Signature.bind()和Signature.bind_partial()替代。

inspect.**getclosurevars**(*func*)

获取Python函数或方法*func*中的外部名称引用到其当前值的映射。一个 命名的元组 被返回。*nonlocals*将引用名称映射到词法封闭变量，将*全局变量*映射到函数的模块全局变量，并将*嵌入*的内置映射到函数体中可见的内建变量。*unbound*是函数中引用的名称集，根据当前模块全局变量和内置变量，根本无法解析该名称。ClosureVars(nonlocals, globals, builtins, unbound)

TypeError如果*func*不是Python函数或方法，则会引发此问题。

3.3版本的新功能

inspect.**unwrap**(*func*，*，stop = None*)

获取由*func*包装的对象。它遵循__wrapped__ 返回链中最后一个对象的属性链。

*stop*是一个可选的回调函数，接受包装器链中的一个对象作为其唯一参数，如果回调函数返回一个真值，则允许解除包装过早终止。如果回调从不返回真值，则链中的最后一个对象将照常返回。例如，signature()如果链中的任何对象具有已__signature__定义的属性，则 使用它来停止展开。

ValueError 如果遇到循环，则会提高。

3.4版新增功能

29.12.5。解释器堆栈

当以下函数返回“帧记录”时，每条记录都是一个 命名的元组 。该元组包含框架对象，文件名，当前行的行号，函数名称，源代码中的上下文行列表以及该列表中当前行的索引。FrameInfo(frame, filename, lineno, function, code_context, index)

*在版本3.5中更改：*返回一个命名的元组而不是元组。

注意： 保持对框架对象的引用，如框架记录的第一个元素中所记录的这些函数返回，可能会导致程序创建引用循环。一旦创建了一个引用循环，即使启用了Python的可选循环检测器，所有可以从形成循环的对象访问的所有对象的生命周期也可能变得更长。如果必须创建这样的周期，重要的是要确保它们被明确分解以避免对象的延迟破坏和增加的内存消耗。

虽然周期检测器会捕获这些，但通过删除finally子句中的周期，可以确定帧（和局部变量）的破坏 。如果在编译或使用Python时禁用循环检测器，这也很重要gc.disable()。例如：

```
def handle_stackframe_without_leak():
    frame = inspect.currentframe()
    try:
        # do something with the frame
    finally:
        del frame
```

如果要保留框架（例如稍后打印回溯），还可以使用该frame.clear()方法中断参考周期。

大多数这些函数支持的可选*上下文*参数指定要返回的上下文的行数，它们以当前行为中心。

inspect.**getframeinfo**(*frame*, *context = 1*)

获取关于帧或追溯对象的信息。一个命名的元组 被返回。Traceback(filename, lineno, function, code_context, index)

inspect.**getouterframes**(*frame*, *context = 1*)

获取一个帧和所有外部帧的帧记录列表。这些框架表示导致创建*框架*的调用。返回列表中的第一个条目表示*框架*；最后一个条目表示*帧*的最外层调用。

在版本3.5中更改： 返回命名元组 列表。FrameInfo(frame, filename, lineno, function, code_context, index)

inspect.**getinnerframes**(*traceback*, *context = 1*)

获取追踪帧和所有内部帧的帧记录列表。这些帧表示由于*帧*而产生的呼叫。列表中的第一个条目表示*回溯*；最后一个条目表示引发异常的位置。

在版本3.5中更改： 返回命名元组 列表。FrameInfo(frame, filename, lineno, function, code_context, index)

inspect.**currentframe**()

返回调用者的堆栈帧的帧对象。

CPython实现细节：该函数依赖于解释器中的Python堆栈框架支持，但不能保证在所有Python实现中都存在。如果在没有Python堆栈框架支持的实现中运行，则此函数返回None。

inspect.**stack**(*context = 1*)

返回调用者堆栈的帧记录列表。返回列表中的第一个条目表示调用者；最后一个条目代表堆叠上最外面的呼叫。

在版本3.5中更改： 返回命名元组 列表。FrameInfo(frame, filename, lineno, function, code_context, index)

inspect.**trace**(*context = 1*)

返回当前帧和引发当前正在处理的异常的帧之间堆栈的帧记录列表。列表中的第一个条目表示调用者；最后一个条目表示引发异常的位置。

在版本3.5中更改： 返回命名元组 列表。FrameInfo(frame, filename, lineno, function, code_context, index)

29.12.6。静态获取属性

既getattr()和hasattr()提取或检查属性的存在时可以触发代码的执行。描述符，如属性，将被调用__getattr__()并且__getattribute__() 可能被调用。

对于需要被动内省的情况，如文档工具，这可能不方便。getattr_static()具有相同的签名，getattr() 但避免在获取属性时执行代码。

inspect.getattr_static (*obj* , *attr* , *default = None*)

不通过描述符协议触发动态查找来检索属性，__getattr__()或__getattribute__()。

注意：此函数可能无法检索getattr可以获取的所有属性（如动态创建的属性），并可能找到getattr无法使用的属性（如引发AttributeError的描述符）。它也可以返回描述符对象而不是实例成员。

如果实例__dict__被其他成员（例如属性）遮蔽，则该函数将无法找到实例成员。

3.2版本中的新功能

getattr_static() 不会解析描述符，例如在C中实现的对象上的槽描述符或getset描述符。将返回描述符对象而不是基础属性。

您可以使用如下代码处理这些问题。请注意，对于任意getset描述符，调用这些描述符可能会触发代码执行：

```
# example code for resolving the builtin descriptor types
class _foo:
    __slots__ = ['foo']

slot_descriptor = type(_foo.foo)
getset_descriptor = type(type(open(__file__)).name)
wrapper_descriptor = type(str.__dict__['__add__'])
descriptor_types = (slot_descriptor, getset_descriptor, wrapper_descriptor)

result = getattr_static(some_object, 'foo')
if type(result) in descriptor_types:
    try:
        result = result.__get__()
    except AttributeError:
        # descriptors can raise AttributeError to
        # indicate there is no underlying value
        # in which case the descriptor itself will
        # have to do
        pass
```

29.12.7。发生器和协程的现状

在实现协程调度程序和生成器的其他高级用途时，确定生成器当前是否正在执行，是否正在等待启动或恢复或执行或已经终止会很有用。getgeneratorstate()允许容易地确定发电机的当前状态。

inspect.getgeneratorstate(*发电机*)

获取生成器迭代器的当前状态。

可能的状态是：

- GEN_CREATED：等待开始执行。
- GEN_RUNNING：目前正在由解释器执行。
- GEN_SUSPENDED：目前暂停在产量表达式。
- GEN_CLOSED：执行已完成。

3.2版本中的新功能

inspect.getcoroutinestate(*coroutine*)

获取协程对象的当前状态。该函数旨在与由函数创建的协程对象一起使用，但将接受任何具有和 属性的协程类对象。`async def`cr_runningcr_frame

可能的状态是：

- CORO_CREATED：等待开始执行。
- CORO_RUNNING：目前正在由解释器执行。
- CORO_SUSPENDED：目前暂停等待表达。
- CORO_CLOSED：执行已完成。

3.5版本中的新功能。

发电机当前的内部状态也可以查询。这对于测试目的非常有用，以确保内部状态按预期进行更新：

inspect.getgeneratorlocals(*发电机*)

获取*生成器*中实时局部变量到其当前值的映射。返回的字典从变量名称映射到值。这相当于调用`locals()`发生器的主体，所有相同的注意事项都适用。

如果*生成器*是当前没有关联帧的生成器，则返回空字典。 `TypeError`如果*生成器*不是Python生成器对象则引发 。

CPython实现细节：该函数依赖于生成器公开Python堆栈框架进行自省，但不能保证在所有Python实现中都是这种情况。在这种情况下，该函数将始终返回一个空字典。
3.3版本的新功能

inspect.getcoroutinelocals(*coroutine*)

该功能与此类似`getgeneratorlocals()`，但适用于由函数创建的协程对象。`async def`

3.5版本中的新功能。

29.12.8。代码对象位标志

Python代码对象有一个`co_flags`属性，它是以下标志的位图：

inspect.CO_OPTIMIZED

代码对象使用快速本地进行优化。

inspect.CO_NEWLOCALS
如果设置，f_locals则在执行代码对象时，将为框架创建一个新的字典。

inspect.CO_VARARGS
该代码对象具有可变的位置参数（*args类似于）。

inspect.CO_VARKEYWORDS
代码对象有一个可变的关键字参数（**kwargs-like）。

inspect.CO_NESTED
当代码对象是嵌套函数时，该标志被设置。

inspect.CO_GENERATOR
当代码对象是生成器函数时，该标志被设置，即当代码对象被执行时返回生成器对象。

inspect.CO_NOFREE
如果没有空闲或单元变量，则标志被设置。

inspect.CO_COROUTINE
当代码对象是协程函数时，标志被设置。当代码对象被执行时，它返回一个协程对象。看到**PEP 492**了解更多详情。

3.5版本中的新功能。

inspect.CO_ITERABLE_COROUTINE
该标志用于将发生器转换为基于生成器的协程。具有此标志的发电机对象可用于 await表达式，并可协同处理对象。看到yield from**PEP 492**了解更多详情。

3.5版本中的新功能。

inspect.CO_ASYNC_GENERATOR
当代码对象是异步生成器函数时，该标志被设置。当代码对象被执行时，它返回一个异步生成器对象。看到**PEP 525**更多细节。

3.6版本中的新功能。

注意： 这些标志特定于CPython，并且可能不会在其他Python实现中定义。此外，标志是一个实现细节，可以在未来的Python版本中删除或弃用。建议使用inspect模块中的公共API来满足任何内省需求。

29.12.9。命令行界面

该inspect模块还从命令行提供了基本的内省功能。

默认情况下，接受模块的名称并打印该模块的来源。可以通过附加冒号和目标对象的限定名称来打印模块内的类或函数。

--details

打印有关指定对象而不是源代码的信息

29.13。 site- 特定于站点的配置钩子

源代码： Lib / site.py

该模块在初始化期间自动导入。自动导入可以使用解释器的-S选项来抑制。

除非-S被使用，否则导入这个模块会将特定于站点的路径附加到模块搜索路径并添加一些内置函数。在这种情况下，可以安全地导入此模块，而不会自动修改模块搜索路径或添加到内置模块。要显式触发通常的站点特定的添加，请调用该site.main()函数。

*在版本3.3中更改：*即使在使用时也导入用于触发路径操作的模块 -S。

它首先从头部和尾部构建四个目录。对于头部，它使用sys.prefix和sys.exec_prefix; 空头被跳过。对于尾部，它使用空字符串，然后 lib/site-packages（在Windows上）或 （在Unix和Macintosh上）。对于每个不同的头尾组合，它会查看它是否指向现有目录，如果是，则将其添加到并检查新添加的配置文件路径。 lib/python*X.Y*/site-packages sys.path

*在3.5版中更改：*对“site-python”目录的支持已被删除。

如果名为“pyvenv.cfg”的文件存在于sys.executable上的一个目录中，则sys.prefix和sys.exec_prefix将设置为该目录，并且还检查站点包（sys.base_prefix和sys.base_exec_prefix将始终为Python安装的“真实”前缀）。如果“pyvenv.cfg”（引导配置文件）包含设置为“false”（不区分大小写）以外的任何内容的密钥“include-system-site-packages”，系统级前缀仍将被搜索到站点-packages; 否则他们不会。

路径配置文件是一个文件，其名称具有以上形式 并存在于上述四个目录之一中; 其内容是添加到其他项目（每行一个）。永远不会添加非现有项目，也不会检查项目是指目录而不是文件。没有项目被添加到一次以上。以空格开头的行和行将被跳过。以（后跟空格或制表符）开头的行被执行。 *name*.pth sys.path sys.path sys.path # import

例如，假设sys.prefix并sys.exec_prefix设置为 /usr/local。然后Python XY库被安装进去 。假设这有一个子目录有三个subsubdirectories， ，和，和两个路径的配置文件，和。假设 包含以下内容： /usr/local/lib/python*X.Y*/usr/local/lib/python*X.Y*/site-packages foo bar spam foo.pth bar.pth foo.pth

```
# foo package configuration

foo
bar
bletch
```

并bar.pth包含：

```
# bar package configuration

bar
```

然后，按以下sys.path顺序添加以下版本特定的目录 ：

```
/usr/local/lib/pythonX.Y/site-packages/bar
/usr/local/lib/pythonX.Y/site-packages/foo
```

请注意，bletch因为它不存在而被省略; 该bar 目录在目录之前，foo因为bar.pth之前按字母顺序排列foo.pth; 并且spam因为在任一路径配置文件中都没有提及而被省略。

在这些路径操作之后，尝试导入名为的模块sitecustomize，该模块 可以执行任意站点特定的自定义。它通常由站点包目录中的系统管理员创建。如果此导入失败并出现ImportError异常，则会被忽略。如果启动Python时没有可用的输出流，就像pythonw.exe在Windows上一样（默认情况下用于启动IDLE），尝试的输出将sitecustomize被忽略。任何异常ImportError都会导致流程的沉默或者神秘的失败。

之后，尝试导入名为的模块usercustomize，如果ENABLE_USER_SITE属实，则该模块可以执行任意用户特定的自定义 。该文件旨在创建在用户site-packages目录中（见下文），sys.path除非被禁用，否则该文件是其中的一部分-s。一个ImportError会被默默地忽略。

请注意，对于一些非Unix系统，sys.prefix并且sys.exec_prefix是空的，和路径的操作会被跳过; 然而进口 sitecustomize和usercustomize仍然是企图。

29.13.1。Readline配置

在支持的系统上，如果Python以交互模式启动且没有选项，则readline此模块还将导入和配置rlcompleter模块 。默认行为是启用制表符完成并 用作历史保存文件。要禁用它，删除（或重写）的属性，你在或模块或您的-S~/.python_historysys.__interactivehook__sitecustomizeusercustomizePYTHONSTARTUP文件。

*在版本3.4中更改：*激活rlcompleter和历史记录是自动完成的。

29.13.2。模块内容

site.PREFIXES
站点包目录的前缀列表。

site.ENABLE_USER_SITE
显示用户站点包目录状态的标志。 True意味着它已启用并被添加到sys.path。 False意味着它已被用户请求禁用（使用-s或 PYTHONNOUSERSITE）。 None意味着它出于安全原因（用户或组ID与有效ID不匹配）或由管理员禁用。

site.USER_SITE
运行Python的用户站点包的路径。可以，None如果 getusersitepackages()还没有被调用。默认值是 针对UNIX和非框架Mac OS X构建的，适用于Mac框架构建以及 Windows。这个目录是一个站点目录，这意味着它中的 文件将被处理。~/.local/lib/python*X.Y*/site-packages~/Library/Python/*X.Y*/lib/python/site-packages*%APPDATA%*\Python\Python*XY*\site-packages.pth

site.USER_BASE

用户站点包的基本目录路径。可以，None如果 getuserbase() 还没有被调用。默认值~/.local用于UNIX和Mac OS X非框架构建， Mac框架构建和 Windows。Distutils使用此值计算用户安装方案的脚本，数据文件，Python模块等的安装目录。也可以看看~/Library/Python/*X.Y*/*%APPDATA%*\Python PYTHONUSERBASE。

site.**main**()

将所有标准站点特定的目录添加到模块搜索路径。当这个模块被导入时，这个函数会自动调用，除非Python解释器是用-S标志启动的。

*在版本3.3中更改：*此函数用于无条件地调用。

site.**addsitedir**(*sitedir*, *known_paths = None*)

将一个目录添加到sys.path并处理其.pth文件。通常用于sitecustomize或usercustomize（见上）。

site.**getsitepackages**()

返回包含所有全局站点包目录的列表。

3.2版本中的新功能

site.**getuserbase**()

返回用户基目录的路径，USER_BASE。如果尚未初始化，该功能也会设置它，尊重PYTHONUSERBASE。

3.2版本中的新功能

site.**getusersitepackages**()

返回用户特定的site-packages目录的路径， USER_SITE。如果尚未初始化，该功能也会设置它，尊重PYTHONNOUSERSITE和USER_BASE。

3.2版本中的新功能

该site模块还提供了一种从命令行获取用户目录的方法：

```
$ python3 -m site --user-site
/home/user/.local/lib/python3.3/site-packages
```

如果它没有参数被调用，它将打印sys.path标准输出中的内容 ，接着是值 USER_BASE的目录和目录是否存在，然后是相同的东西 USER_SITE，最后是值ENABLE_USER_SITE。

--user-base

打印用户基目录的路径。

--user-site

打印用户site-packages目录的路径。

如果同时给出了这两个选项，则会打印用户群和用户站点（始终按此顺序），并以os.pathsep。分隔。

如果提供了任何选项，则脚本将以下列值之一退出：0如果用户site-packages目录已启用，用户1是否已禁用该用户，2如果由于安全原因或由管理员禁用该用户，并且值较大如果有错误，则为2。

也可以看看： **PEP 370** - 每个用户网站包目录

29.14。 fpectl- 浮点异常控制

注意： 该fpectl模块不是默认生成的，其使用不受鼓励，并且除专家之外可能是危险的。有关更多详细信息，另请参阅限制和其他限制条件部分 。

大多数计算机按照所谓的IEEE-754标准执行浮点运算。在任何实际的计算机上，某些浮点运算产生的结果不能表示为正常的浮点值。例如，尝试

```
>>> import math
>>> math.exp(1000)
inf
>>> math.exp(1000) / math.exp(1000)
nan
```

（上面的例子可以在很多平台上工作，DEC Alpha可能是一个例外。）“Inf”是IEEE-754中一个特殊的非数字值，代表“无穷大”，“nan”代表“不是数字。 “请注意，除了非数字结果之外，当您要求Python执行这些计算时，没有什么特别的事情发生。这实际上是IEEE-754标准中规定的默认行为，如果它适用于您，请立即停止阅读。

在某些情况下，在发生错误操作的地方提出异常并停止处理会更好。该 fpectl模块用于这种情况。它提供了对来自多个硬件制造商的浮点单元的控制，允许用户在发生SIGFPE任何IEEE-754除零，溢出或无效操作异常时开启生成过程。与插入到您的Python系统的C代码中的一对包装宏一起SIGFPE被捕获并转换为Python FloatingPointError异常。

该fpectl模块定义了以下功能并可能引发给定的异常情况：

fpectl.turnon_sigfpe ( )
    打开一代SIGFPE，并建立一个适当的信号处理程序。

fpectl.turnoff_sigfpe ( )
    重置浮点异常的默认处理。

*异常*fpectl.FloatingPointError
    在turnon_sigfpe()执行完之后，一个浮点操作会引发IEEE-754异常部分中的一个被Zero，Overflow或Invalid操作，然后引发这个标准的Python异常。

29.14.1。示例

以下示例演示如何启动和测试fpectl模块的操作 。

```
>>> import fpectl
>>> import fpetest
>>> fpectl.turnon_sigfpe()
>>> fpetest.test()
overflow        PASS
FloatingPointError: Overflow
```

```
div by 0        PASS
FloatingPointError: Division by zero
  [ more output from test elided ]
>>> import math
>>> math.exp(1000)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FloatingPointError: in math_1
```

29.14.2。限制和其他因素

设置给定的处理器来捕捉IEEE-754浮点错误当前需要在每个体系结构的基础上定制代码。您可能需要修改 `fpectl`以控制您的特定硬件。

将 IEEE-754 异常转换为 Python 异常需要使用包装宏 ， `PyFPE_START_PROTECT` 并 `PyFPE_END_PROTECT`以适当的方式将其插入代码中。Python本身已被修改以支持该`fpectl`模块，但数值分析师感兴趣的许多其他代码却没有。

该`fpectl`模块不是线程安全的。

也可以看看： 源代码分发中的一些文件可能对了解更多关于这个模块如何操作的内容感兴趣。包含文件`Include/pyfpe.h`在一定程度上讨论了该模块的实现。`Modules/fpetestmodule.c`给出了几个使用的例子。可以在其中找到许多其他示例 `Objects/floatobject.c`。

30.自定义Python解释器

本章描述的模块允许编写类似于Python的交互式解释器的接口。如果你想要一个除了Python语言之外还支持一些特殊功能的Python解释器，你应该看看这个`code`模块。（该`codeop`模块是较低级别的，用于支持编译可能不完整的Python代码块。）

本章描述的完整模块列表是：

30.1。code-解释器基类

源代码： Lib / code.py

该code模块提供了一些工具来实现Python中的read-eval-print循环。包括两个类和便利功能，可用于构建提供交互式解释器提示的应用程序。

class code. InteractiveInterpreter（ *locals = None* ）

这个类处理解析和解释器状态（用户的名字空间）；它不处理输入缓冲或提示或输入文件命名（文件名总是显式传递）。可选的*locals*参数指定将在其中执行代码的字典; 它默认为一个新创建的字典，其密钥'__name__'设置为'__console__'和键'__doc__'设置为None。

class code. InteractiveConsole（ *locals = None*，*filename =“<console>”* ）

密切模拟交互式Python解释器的行为。该类基础上InteractiveInterpreter，并使用熟悉的提示增加了 sys.ps1与sys.ps2和输入缓冲。

code. interact（ *banner = None*，*readfunc = None*，*local = None*，*exitmsg = None* ）

便捷功能运行读取评估打印循环。这将创建一个新实例InteractiveConsole并设置*readfunc*作为InteractiveConsole.raw_input()方法（如果提供）。如果提供了*local*，它将被传递给InteractiveConsole构造函数，作为解释器循环的缺省名称空间。 interact() 然后运行实例的方法，并使用*横幅*和*exitmsg*作为横幅和退出消息来使用（如果提供）。控制台对象在使用后被丢弃。

*在版本3.6中更改：*增加了*exitmsg*参数。

code. compile_command（ *source*，*filename =“<input>”*，*symbol =“single”* ）

这个函数对于想要模拟Python的解释器主循环（又名读 - 评估 - 打印循环）的程序很有用。棘手的部分是确定用户何时输入了可以通过输入更多文本（而不是完整命令或语法错误）完成的不完整命令。这个功能*几乎*总是与真正的解释器主循环做出相同的决定。

*源*是源字符串; *filename*是从中读取源的可选文件名，默认为'<input>'；和*符号*是可选的语法开始符号，这应该是'single'（缺省值）或'eval'。

如果命令是完整且有效的，则返回一个代码对象（与之相同）；如果命令不完整; 提出 如果命令是完整的，包含语法错误，或者提高或者如果命令包含一个无效的文字。compile(source, filename, symbol)NoneSyntaxErrorOverflowErrorValueError

30.1.1。交互式解释器对象

InteractiveInterpreter. runsource（ *source*，*filename =“<input>”*，*symbol =“single”* ）

在解释器中编译并运行一些源代码。参数与之相同 compile_command()；*文件名*的默认值是'<input>'，对于 *符号*是'single'。有几件事会发生：

- 输入不正确; compile_command()提出了一个例外（ SyntaxError或OverflowError ）。语法回溯将通过调用该showsyntaxerror()方法来打印。 runsource() 返回False。

- 输入不完整，需要更多输入; compile_command() 返回None。 runsource()返回True。
- 输入完成; compile_command()返回一个代码对象。该代码通过调用runcode() （它除了处理运行时异常外SystemExit ）来执行。 runsource()返回False。

返回值可用于决定是使用sys.ps1还是sys.ps2 提示下一行。

InteractiveInterpreter.runcode（*代码*）

执行一个代码对象。当发生异常时，showtraceback()被调用来显示回溯。除了SystemExit允许传播的所有异常都被捕获。

关于的一个注意事项KeyboardInterrupt：此例外情况可能发生在此代码的其他地方，并且可能无法始终捕获。来电者应该准备好处理它。

InteractiveInterpreter.showsyntaxerror（*filename = None* ）

显示刚发生的语法错误。这不会显示堆栈跟踪，因为没有语法错误。如果给出了*文件名*，它会填充到异常中，而不是Python解析器提供的默认文件名，因为'<string>'从字符串读取时它总是使用。输出由该write()方法编写。

InteractiveInterpreter.showtraceback（ ）

显示刚发生的异常。我们删除第一个堆栈项，因为它在解释器对象实现中。输出由该write()方法编写。

*在版本3.5中进行了更改：*显示完整链式回溯，而不是仅显示主要回溯。

InteractiveInterpreter.write（*数据*）

将一个字符串写入标准错误流（ sys.stderr ）。派生类应该覆盖它以根据需要提供适当的输出处理。

30.1.2。交互式控制台对象

该InteractiveConsole班是的一个子类 InteractiveInterpreter，因此提供了解释对象以及以下附加的所有方法。

InteractiveConsole.interact（*banner = None*，*exitmsg = None* ）

密切模拟交互式Python控制台。可选的*横幅*参数指定在首次交互之前要打印的横幅; 默认情况下，它会打印一个与标准Python解释器打印的横幅相似的横幅，然后是括号内的控制台对象的类名称（以免将其与真正的解释器混淆 - 因为它太接近了！）。

可选的*exitmsg*参数指定退出时打印的退出消息。通过空字符串来抑制退出消息。如果没有给出*exitmsg*，或者 None打印默认消息。

*在版本3.4中更改：*要禁止打印任何横幅，请传递空字符串。

*在版本3.6中更改：*退出时打印退出消息。

InteractiveConsole.push（*线*）

将一行源文本推送给解释器。该行不应该有换行符; 它可能有内部换行符。该行被附加到一个缓冲区，并且解释器的runsource()方法被调用，并将缓冲区的连接内容作为源进行调用。如果这表示该命令已执行或无效，则重置缓冲区; 否则，该命令是不完整的，并且缓冲

区保持原样，因为它是在追加行后。返回值是True如果需要更多输入，False如果以某种方式处理该行（这与之相同runsource()）。

InteractiveConsole.**resetbuffer**()

从输入缓冲区中移除任何未处理的源文本。

InteractiveConsole.**raw_input**(*prompt=""*)

编写提示并阅读一行。返回的行不包括尾随换行符。当用户输入EOF按键序列时，EOFError会产生。基本实现读取sys.stdin;一个子类可以用不同的实现来替换它。

30.2。 codeop- 编译Python代码

源代码： Lib / codeop.py

该codeop模块提供了用于模拟Python read-eval-print循环的实用程序，就像code模块中所做的那样。因此，您可能不想直接使用该模块; 如果你想在你的程序中包含这样一个循环，你可能需要使用该code 模块。

这项工作有两部分：

1. 能够判断一行输入是否完成了Python语句：简而言之，告诉是否打印下一个' >>>'或'...'。
2. 记住用户已经输入了哪些未来的陈述，因此随后的输入可以用这些陈述进行编译。

该codeop模块提供了一种做这些事情的方式，以及一种做这些事情的方式。

要做前者：

codeop. compile_command (*source* , *filename* ="<input>" , *symbol* ="single")

尝试编译*源*代码，如果*源代码*是有效的Python代码，它应该是一串Python代码并返回一个代码对象。在这种情况下，代码对象的文件名属性将是*文件名*，默认为 '<input>'。返回None如果*来源*是*不是*有效的Python代码，但有效的Python代码前缀。

如果*源*存在问题，则会引发异常。 SyntaxError 如果存在无效的Python语法，OverflowError或者ValueError存在无效的文字，则会引发此问题。

所述*符号*参数确定是否*源*被编译为一个声明（'single'作为，缺省值）或表达式（'eval'）。任何其他价值都会导致ValueError提高。

> **注意：** 解析器可能（但不太可能）在到达源代码结束之前停止解析并获得成功的结果; 在这种情况下，可以忽略尾随符号而不是引起错误。例如，后跟两个换行符的反斜线后面可能会出现任意垃圾。一旦解析器的API更好，这将被修复。

*类*codeop. Compile

这个类的实例具有__call__()与内置函数签名相同的方法compile()，但不同之处在于，如果实例编译包含__future__语句的程序文本，那么实例会“记住”，并用生效的语句编译所有后续程序文本。

*类*codeop. CommandCompiler

这个类别的实例具有__call__()与签名相同的方法 compile_command(); 不同之处在于，如果实例编译包含__future__语句的程序文本，那么该实例会'记住'并且使用有效的语句编译所有随后的程序文本。

31.导入模块

本章介绍的模块提供了导入其他Python模块和钩子以定制导入过程的新方法。

本章描述的完整模块列表是：

31.1。zipimport- 从Zip存档导入模块

该模块增加了从ZIP格式档案中导入Python模块（*.py，*.pyc）和软件包的功能。通常不需要zipimport明确使用该模块; 内置import机制会自动使用它作为sys.pathZIP归档路径的项目。

通常，sys.path是作为字符串的目录名称列表。该模块还允许将一个项目sys.path作为一个命名ZIP文件存档的字符串。ZIP归档文件可以包含一个子目录结构以支持包导入，并且归档中的路径可以被指定为仅从子目录导入。例如，路径example.zip/lib/只能从lib/存档中的子目录导入。

任何文件可能存在于ZIP压缩文件，但只有文件.py和.pyc可供导入。动态模块（.pyd，.so）的ZIP导入是不允许的。请注意，如果存档仅包含.py文件，则Python不会尝试通过添加相应.pyc文件来修改存档，这意味着如果ZIP存档不包含.pyc文件，导入可能会非常慢。

目前不支持带归档评论的ZIP归档。

也可以看看:

PKZIP应用笔记
由Phil Katz撰写的关于ZIP文件格式的文档，所使用的格式和算法的创建者。

PEP 273 - 从Zip档案导入模块
由James C. Ahlstrom撰写，他也提供了一个实现。Python 2.3遵循PEP 273中的规范，但使用Just van Rossum编写的使用PEP 302中描述的导入钩子的实现。

PEP 302 - 新的进口挂钩
PEP添加有助于此模块工作的导入挂钩。

该模块定义了一个例外：

*异常*zipimport.ZipImportError
zipimporter对象引发异常。这是它的一个子类ImportError，所以它也可以被捕获ImportError。

31.1.1。zipimporter对象

zipimporter是导入ZIP文件的类。

class zipimport.zipimporter(*archivepath*)
创建一个新的zipimporter实例。*archivepath*必须是ZIP文件的路径，或ZIP文件中的特定路径。例如，*archivepath*的foo/bar.zip/lib将查找模块在libZIP文件内的目录foo/bar.zip（假设它存在）。

ZipImportError如果*archivepath*未指向有效的ZIP存档，则会引发此问题。

find_module(*fullname*[, *path*])

搜索由*全名*指定的模块。*完整名称*必须是完全限定的（虚线）模块名称。它返回zipimporter实例本身，如果找到该模块，或者None它不是。可选的 *路径*参数被忽略 - 它在那里与进口协议兼容。

get_code（*全名*）
返回指定模块的代码对象。ZipImportError如果找不到该模块，请升级 。

get_data（*路径名*）
返回与*路径名*关联的数据。提高OSError如果没有找到该文件。

*在版本3.3中更改：*IOError用于提高而不是OSError。

get_filename（*全名*）
__file__如果指定的模块被导入，返回值将被设置为。ZipImportError如果找不到该模块，请升级。

版本3.1中的新功能。

get_source（*全名*）
返回指定模块的源代码。ZipImportError如果找不到该模块，None则进行提升 ，如果存档确实包含该模块，则返回 ，但没有来源。

is_package（*全名*）
返回True如果由指定的模块*全名*是一个包。ZipImportError如果找不到该模块，请升级 。

load_module（*全名*）
加载由*全名*指定的模块。*完整名称*必须是完全限定的（虚线）模块名称。它返回导入的模块，或者ZipImportError如果找不到则引发 。

archive
导入程序的关联ZIP文件的文件名，没有可能的子路径。

prefix
ZIP文件中用于搜索模块的子路径。这是zipimporter对象的空字符串，它指向ZIP文件的根目录。

当与斜线组合时，archive和prefix属性等于给予构造函数的原始*归档路径*参数zipimporter。

31.1.2。示例

这里是一个从ZIP压缩文件导入模块的例子 - 注意 zipimport模块没有明确使用。

```
$ unzip -l example.zip
Archive:  example.zip
  Length      Date    Time    Name
 --------    ----    ----    ----
     8467  11-26-02 22:30   jwzthreading.py
```

```
 --------                    -------
     8467                    1 file
$ ./python
Python 2.3 (#1, Aug 1 2003, 19:54:32)
>>> import sys
>>> sys.path.insert(0, 'example.zip')  # Add .zip file to front of path
>>> import jwzthreading
>>> jwzthreading.__file__
'example.zip/jwzthreading.py'
```

31.2。pkgutil- 包扩展实用程序

源代码： Lib / pkgutil.py

该模块为导入系统提供实用程序，特别是软件包支持。

class pkgutil.ModuleInfo（*module_finder*，*name*，*ispkg* ）

一个namedtuple，它包含模块信息的简要概述。

3.6版本中的新功能。

pkgutil.extend_path（*路径*，*名称*）

扩展组成包的模块的搜索路径。预期用途是将以下代码放入包中__init__.py：

```
from pkgutil import extend_path
__path__ = extend_path(__path__, __name__)
```

这将添加到软件包的__path__所有以该软件包sys.path命名的目录的子目录上。如果要将单个逻辑包的不同部分分配为多个目录，这非常有用。

它还会查找与*名称*参数匹配的 *.pkg文件。该功能与文件类似（请参阅 模块以获取更多信息），不同之处在于它不包含以特殊字符开头的行。一个文件在面值上是可信的：除了检查重复项之外，文件中找到的所有项 都被添加到路径中，而不管它们是否存在于文件系统中。（这是一个功能。）**.pthsiteimport*.pkg*.pkg

如果输入路径不是列表（如冻结包的情况那样），它将保持不变。输入路径未被修改; 将返回一个扩展副本。项目只在最后附加到副本。

假定sys.path是一个序列。 sys.path 那些不是指代现有目录的字符串的项目将被忽略。sys.path 当用作文件名时导致错误的Unicode项可能会导致此函数引发异常（符合os.path.isdir() 行为）。

class pkgutil.ImpImporter（*dirname = None* ）

PEP 302 Finder，它包装了Python的“经典”导入算法。

如果*dirname*是一个字符串，则a创建 **PEP 302**查找程序来搜索该目录。如果 *dirname*是None，a创建 **PEP 302**查找程序，用于搜索当前内容sys.path，以及任何冻结或内置的模块。

请注意，ImpImporter目前不支持在展示位置上使用sys.meta_path。

*自3.3版弃用：*此模拟不再需要，因为标准导入机制现在完全符合PEP 302并可用importlib。

*类*pkgutil.ImpLoader（*全名*，*文件*，*文件名*，*等等*）

Loader包装了Python的“经典”导入算法。

*自3.3版弃用：*此模拟不再需要，因为标准导入机制现在完全符合PEP 302并可用importlib。

pkgutil.**find_loader**(*全名*)

为给定的*全名*检索模块加载器。

这是一个向后兼容的包装器， importlib.util.find_spec()可以将大多数故障转换为ImportError并仅返回加载器而不是完整的 ModuleSpec。

*版本3.3中更改：*更新为直接基于，importlib而不是依赖于包内部PEP 302导入仿真。

*版本3.4中更改：*更新为基于**PEP 451**

pkgutil.**get_importer**(*path_item*)

检索给定*path_item*的查找器。

sys.path_importer_cache如果它是由路径挂接新创建的，则返回的查找程序将被缓存。

如果需要重新扫描，sys.path_hooks则可以手动清除缓存（或部分缓存）。

*版本3.3中更改：*更新为直接基于，importlib而不是依赖于包内部PEP 302导入仿真。

pkgutil.**get_loader**(*module_or_name*)

获取*module_or_name*的加载器对象。

如果模块或包装可以通过正常的进口机制进行访问，则返回该机械相关部件周围的包装。None如果无法找到或导入模块，则返回 。如果已命名的模块尚未导入，则会导入其包含的包（如果有），以便建立包__path__。

*版本3.3中更改：*更新为直接基于，importlib而不是依赖于包内部PEP 302导入仿真。

*版本3.4中更改：*更新为基于**PEP 451**

pkgutil.**iter_importers**(*fullname=''*)

给定模块名称的产量查找器对象。

如果fullname包含'.'，则查找器将用于包含完整名称的包，否则它们将全部是已注册的顶级查找器（即，在sys.meta_path和sys.path_hooks上）。

如果指定的模块位于包中，则该包将作为调用此函数的副作用导入。

如果未指定模块名称，则会生成所有顶级查找器。

*版本3.3中更改：*更新为直接基于，importlib而不是依赖于包内部PEP 302导入仿真。

pkgutil.**iter_modules**(*path=None*, *prefix=''*)

*路径*ModuleInfo上的所有子模块的产量，或者，如果 *路径*是，所有顶级模块都打开。None sys.path

*路径*应该是None要查找模块的路径或列表。

*前缀*是在输出上的每个模块名称的前面输出的字符串。

> **注意：** 只适用于定义 方法的查找器iter_modules()。这个接口是非标准的，所以该模块也提供了importlib.machinery.FileFinder和的 实现zipimport.zipimporter。

*版本3.3中更改：*更新为直接基于，importlib而不是依赖于包内部PEP 302导入仿真。

pkgutil.**walk_packages**(*path = None*, *prefix =''*, *onerror = None*)

ModuleInfo所有模块在*路径上*递归 产生，或者如果*路径*是None所有可访问的模块。

*路径*应该是None要查找模块的路径或列表。

*前缀*是在输出上的每个模块名称的前面输出的字符串。

请注意，此函数必须导入给定*路径*上的所有*包*（*不是*所有模块！），以便访问该属性以查找子模块。__path__

*onerror*是一个函数，如果在尝试导入程序包时发生任何异常，则会使用一个参数（正在导入的程序包的名称）调用它。如果未提供*onerror*函数，ImportError则会捕获并忽略这些函数，而传播所有其他异常，从而终止搜索。

例子：

```
# list all modules python can access
walk_packages()

# list all submodules of ctypes
walk_packages(ctypes.__path__, ctypes.__name__ + '.')
```

> **注意：** 只适用于定义 方法的查找器iter_modules()。这个接口是非标准的，所以该模块也提供了importlib.machinery.FileFinder和的 实现zipimport.zipimporter。

*版本3.3中更改：*更新为直接基于，importlib而不是依赖于包内部PEP 302导入仿真。

pkgutil.**get_data**(*包*, *资源*)

从包中获取资源。

这是加载器 get_data API 的包装器。所述 *包*参数应该是一个包的名称，在标准模块格式（foo.bar）。所述*资源*参数应该是在一个相对文件名的形式，使用/作为路径分隔。父目录名称 .. 是不允许的，也不是根目录名称（以a开头/）。

该函数返回一个二进制字符串，它是指定资源的内容。

对于位于已经导入的文件系统中的软件包，这大致相当于：

```
d = os.path.dirname(sys.modules[package].__file__)
data = open(os.path.join(d, resource), 'rb').read()
```

如果软件包无法找到或加载，或者它使用 不支持的加载程序get_data，则会None返回。尤其是装载机的 命名空间的包不支持 get_data。

31.3。modulefinder- 查找脚本使用的模块

源代码： Lib / modulefinder.py

该模块提供了一个ModuleFinder可用于确定由脚本导入的模块集的类。modulefinder.py也可以作为脚本运行，以Python脚本的文件名作为参数，之后将打印导入模块的报告。

modulefinder.AddPackagePath（*pkg_name*，*path*）
记录名称为*pkg_name*的包可以在指定的*路径中*找到。

modulefinder.ReplacePackage（*oldname*，*newname*）
允许指定名为*oldname*的模块实际上是名为*newname*的软件包 。

class modulefinder.ModuleFinder（*path = None*，*debug = 0*，*excludes = []*，*replace_paths = []*）
该类提供run_script()和report()方法，以确定该组由脚本导入模块。*路径*可以是搜索模块的目录列表; 如果未指定，sys.path则使用。 *调试*设置调试级别; 更高的值使得该类打印关于它在做什么的调试消息。*不包括*是要从分析中排除的模块名称的列表。*replace_paths*是将在模块路径中替换的元组列表。(oldpath, newpath)

report（）
打印报告到标准输出，列出由脚本及其路径导入的模块，以及缺少或似乎缺失的模块。

run_script（*路径名*）
分析必须包含Python代码的*路径名*文件的内容。

modules
字典映射模块名称到模块。请参阅 ModuleFinder的示例用法。

31.3.1。 使用示例ModuleFinder

稍后将分析的脚本（bacon.py）：

```
import re, itertools

try:
    import baconhameggs
except ImportError:
    pass

try:
    import guido.python.ham
except ImportError:
    pass
```

将输出bacon.py报告的脚本：

```
from modulefinder import ModuleFinder

finder = ModuleFinder()
finder.run_script('bacon.py')

print('Loaded modules:')
for name, mod in finder.modules.items():
    print('%s: ' % name, end='')
    print(','.join(list(mod.globalnames.keys())[:3]))

print('-'*50)
print('Modules not imported:')
print('\n'.join(finder.badmodules.keys()))
```

示例输出（可能因架构而异）：

```
Loaded modules:
_types:
copyreg:  _inverted_registry,_slotnames,__all__
sre_compile:  isstring,_sre,_optimize_unicode
_sre:
sre_constants:  REPEAT_ONE,makedict,AT_END_LINE
sys:
re:  __module__,finditer,_expand
itertools:
__main__:  re,itertools,baconhameggs
sre_parse:  _PATTERNENDERS,SRE_FLAG_UNICODE
array:
types:  __module__,IntType,TypeType
--------------------------------------------------
Modules not imported:
guido.python.ham
baconhameggs
```

31.4。runpy- 查找和执行Python模块

源代码： Lib / runpy.py

该runpy模块用于定位和运行Python模块，而无需先导入它们。它的主要用途是实现-m命令行开关，允许脚本使用Python模块命名空间而不是文件系统进行定位。

请注意，这*不是*一个沙盒模块 - 所有代码都在当前进程中执行，并且在函数返回后，任何副作用（例如其他模块的缓存导入）都将保留。

此外，runpy函数返回后，所执行代码定义的任何函数和类都不能保证能够正常工作。如果该限制对于给定的用例是不可接受的，importlib 则可能是比这个模块更合适的选择。

该runpy模块提供两个功能：

runpy.run_module（*mod_name*，*init_globals* = *None*，*run_name* = *None*，*alter_sys* = *False* ）

执行指定模块的代码并返回生成的模块全局变量字典。模块的代码首先使用标准导入机制（请参阅**PEP 302**），然后在新的模块名称空间中执行。

该*mod_name*参数应该是一个绝对的模块名称。如果模块名称引用的是包而不是普通模块，则会导入__main__该包，然后执行该包中的子模块，并返回生成的模块全局变量字典。

可选的字典参数*init_globals*可用于在代码执行之前预先填充模块的全局字典。提供的字典将不会被修改。如果在提供的字典中定义了以下任何特殊全局变量，则这些定义将被覆盖run_module()。

特殊的全局变量__name__，__spec__，__file__，__cached__，__loader__和__package__执行模块代码之前在全局词典设置（注意，这是一组变量最小-其他变量可以被隐式设置为一个解释实现细节）。

__name__如果此可选参数不是，则设置为*run_name*;否则 None，如果命名模块是包，则*返回*mod_name参数。mod_name + '.__main__'

__spec__将针对*实际*导入的模块进行适当设置（即__spec__.name始终为*mod_name*或 永远不会*运行*）。mod_name + '.__main__

__file__，__cached__，__loader__和__package__被 设置为正常基于模块规范。

如果参数*alter_sys*供给和的计算结果为True，然后sys.argv[0]用的值进行更新__file__，并 sys.modules[__name__]与正在执行的模块的临时模块的对象更新。双方sys.argv[0]并sys.modules[__name__] 恢复到原来的值在函数返回之前。

请注意，这种操作sys不是线程安全的。其他线程可能会看到部分初始化的模块，以及更改的参数列表。建议在sys从线程代码调用此函数时将模块保留为独立模式。

也可以看看： 该-m选项提供了与命令行相同的功能。

*在版本3.1中进行了更改：*通过查找__main__子模块增加了执行软件包的功能。

*在版本3.2中进行了更改：*添加了__cached__全局变量（请参阅**PEP 3147**）。

*在版本3.4中更改：已*更新以充分利用由添加的模块规格功能 **PEP 451**。这允许__cached__为以这种方式运行的模块进行正确设置，并确保真正的模块名称始终可以被访问__spec__.name。

runpy.run_path（*file_path*，*init_globals* = *None*，*run_name* = *None* ）

在指定的文件系统位置执行代码并返回生成的模块全局变量字典。与提供给CPython命令行的脚本名称一样，提供的路径可能引用Python源文件，编译的字节码文件或包含__main__ 模块的有效sys.path条目（例如包含顶级__main__.py文件的zip 文件）。

对于简单的脚本，指定的代码只是在新的模块名称空间中执行。对于有效的sys.path条目（通常是一个zip文件或目录），该条目首先添加到开头sys.path。该功能然后__main__使用更新的路径查找并执行模块。请注意，如果在指定位置没有此类模块，则没有针对调用__main__位于其他位置的现有条目的特殊保护sys.path。

可选的字典参数*init_globals*可用于在代码执行之前预先填充模块的全局字典。提供的字典将不会被修改。如果在提供的字典中定义了以下任何特殊全局变量，则这些定义将被覆盖run_path()。

特殊的全局变量__name__，__spec__，__file__，__cached__，__loader__和__package__执行模块代码之前在全局词典设置（注意，这是一组变量最小-其他变量可以被隐式设置为一个解释实现细节）。

__name__如果此可选参数不是 ，则设置为*run_name*。None’<run_path>’

如果所提供的路径直接引用的脚本文件（无论是作为源或作为预编译字节码），然后__file__将被设置为所提供的路径，并且__spec__，__cached__，__loader__和__package__将全部被设定为None。

如果提供的路径是对有效sys.path条目的引用，那么__spec__将对导入的__main__模块进行适当设置（即__spec__.name始终为__main__）。__file__，__cached__，__loader__和__package__将被设置为正常的基于模块的规格。

sys模块也进行了一些改动。首先，sys.path可以如上所述进行改变。sys.argv[0]用正在执行的模块的临时模块对象的值更新file_path并sys.modules[__name__]更新。sys在函数返回之前，所有对项目的修改都会被恢复。

请注意，与此不同run_module()，sys 在此函数中所做的更改不是可选的，因为这些调整对于允许执行sys.path条目至关重要。由于线程安全性限制仍然适用，因此在线程代码中使用此函数应该使用导入锁进行序列化或委托给单独的进程。

也可以看看： 命令行上的等效功能的接口选项（ ）。python path/to/script

3.2版本中的新功能

*在版本3.4中更改：已*更新以充分利用由添加的模块规格功能 **PEP 451**。这允许__cached__在__main__从有效的sys.path条目导入而不是直接执行的情况下正确设置。

也可以看看:

PEP 338 - 将模块作为脚本执行
由Nick Coghlan编写和实施的PEP。

PEP 366 - 主要模块显式相对导入
由Nick Coghlan编写和实施的PEP。

PEP 451 - 导入系统的ModuleSpec类型
由Eric Snow编写和实施的PEP

命令行和环境 - CPython命令行详细信息

该`importlib.import_module()`功能

31.5。importlib- 执行 import

版本3.1中的新功能。

源代码： Lib / importlib / __ init__.py

31.5.1。简介

这个importlib软件包的目的是双重的。一种是在Python源代码中提供import语句的实现（从而扩展为 __import__()函数）。这提供了一个import可以移植到任何Python解释器的实现。这也提供了一个比Python之外的编程语言更容易理解的实现。

二，要实现的组件import暴露在这个包中，使用户更容易创建自己的自定义对象（一般称为导入器）参与导入过程。

也可以看看:

导入声明
该import声明的语言参考。

包规范
包装的原始规格。自写这篇文档以来，一些语义已经发生了变化（例如，基于None in的重定向sys.modules）。

该__import__()功能
这个import语句是这个函数的语法糖。

PEP 235
在不区分大小写的平台上导入

PEP 263
定义Python源代码编码

PEP 302
新的导入钩子

PEP 328
进口：多线和绝对/相对

PEP 366
主模块显式相对导入

PEP 420
隐式名称空间包

PEP 451
导入系统的ModuleSpec类型

PEP 488
消除PYO文件

PEP 489

多相扩展模块初始化

PEP 3120
使用UTF-8作为默认源编码

PEP 3147
PYC存储库目录

31.5.2。函数

importlib.__import__（*名称*，*globals = None*，*locals = None*，*fromlist = （ ）*，*level = 0* ）

内置__import__()函数的实现。

> **注意：** 模块的编程导入应该使用import_module()而不是这个功能。

importlib.import_module（*名称*，*包=无* ）

导入一个模块。该*名*参数指定哪些模块在绝对或相对术语（例如无论是进口pkg.mod或..mod）。如果名称是用相对术语指定的，那么必须将*package*参数设置为用作解析包名称的锚点的包的名称（例如，将导入 ）。import_module('..mod', 'pkg.subpkg')pkg.mod

该import_module()函数充当简化的封装 importlib.__import__()。这意味着函数的所有语义都来自于importlib.__import__()。这两个函数最重要的区别是import_module()返回指定的包或模块（例如pkg.mod），同时__import__() 返回顶级包或模块（例如pkg）。

如果您正在动态导入自解释器开始执行以来创建的模块（例如，创建了Python源文件），则可能需要调用invalidate_caches()才能使新模块被导入系统注意到。

*在版本3.3中更改：*父包将自动导入。

importlib.find_loader（*名称*，*路径=无* ）

查找模块的加载程序，可选地在指定的*路径中*。如果该模块处于sys.modules，则sys.modules[name].__loader__返回（除非加载程序将被None设置或未被设置，在这种情况下 ValueError被提起）。否则，使用搜索sys.meta_path 完成。None如果没有发现加载程序则返回。

虚线的名称没有隐式导入其父项，因为需要加载它们，这可能不是所期望的。要正确导入子模块，您需要导入子模块的所有父包，并将正确的参数用于*路径*。

3.3版本的新功能

*在版本3.4中更改：*如果__loader__没有设置，ValueError就像设置属性时一样None。

*自3.4版弃用：*importlib.util.find_spec()改为使用。

importlib.invalidate_caches（ ）

使存储在中的查找程序的内部缓存失效 sys.meta_path。如果发现者实现了，invalidate_caches()那么它将被调用来执行失效。如果在程序运行时创建/安装了任何模块

以确保所有查找程序都会注意到新模块的存在，则应该调用此函数。

3.3版本的新功能

importlib.**reload**(*模块*)

重新加载以前导入的*模块*。参数必须是模块对象，所以它必须在之前成功导入。如果您已经使用外部编辑器编辑了模块源文件，并且希望在不离开Python解释器的情况下尝试新版本，这将非常有用。返回值是模块对象（如果重新导入会导致放置不同的对象，则模块对象可能不同`sys.modules`）。

何时`reload()`执行：

- 重新编译Python模块的代码并重新执行模块级代码，通过重新使用最初加载模块的加载程序来定义一组新的对象，这些对象将绑定到模块字典中的名称。`init`扩展模块的功能不是第二次调用。
- 与Python中的所有其他对象一样，旧对象只有在其引用计数降至零后才会回收。
- 模块名称空间中的名称将更新为指向任何新的或更改的对象。
- 对旧对象的其他引用（例如模块外部的名称）不会反弹以引用新对象，并且如果需要，则必须在它们出现的每个命名空间中进行更新。

还有一些其他警告：

当模块重新加载时，其字典（包含模块的全局变量）将被保留。名称的重新定义将覆盖旧的定义，所以这通常不是问题。如果新版本的模块未定义旧版本定义的名称，则旧定义将保留。如果该模块维护全局表或对象缓存，则该特性可用于模块的优势 - 如果需要`try`，可使用语句测试表的存在并跳过其初始化：

```
try:
    cache
except NameError:
    cache = {}
```

重新加载内置或动态加载的模块通常不是很有用。重装`sys`，`__main__`，`builtins`不建议和其他关键模块。在许多情况下，扩展模块并不是设计成多次初始化的，并且在重新加载时可能会以任意方式失败。

如果一个模块使用`from`... 从另一个模块导入对象`import`，则调用`reload()`另一个模块不会重新定义从其导入的对象 - 解决此问题的方法之一是重新执行`from`语句，另一个方法是使用`import` 限定名称（*module.name*）代替。

如果一个模块实例化一个类的实例，重新加载定义类的模块不会影响实例的方法定义 - 它们继续使用旧的类定义。派生类也是如此。

3.4版新增功能

31.5.3。 importlib.abc-导入相关的抽象基类

源代码： Lib / importlib / abc.py

该importlib.abc模块包含所有使用的核心抽象基类import。核心抽象基类的一些子类也被提供来帮助实现核心ABCs。

ABC层次结构：

```
object
 +-- Finder (deprecated)
 |    +-- MetaPathFinder
 |    +-- PathEntryFinder
 +-- Loader
      +-- ResourceLoader --------+
      +-- InspectLoader          |
           +-- ExecutionLoader --+
                                 +-- FileLoader
                                 +-- SourceLoader
```

*类*importlib.abc.Finder

代表查找器的抽象基类。

*自3.3版弃用：*MetaPathFinder或使用PathEntryFinder。

abstractmethod find_module（*fullname*，*path* = *None* ）

用于查找指定模块的加载程序的抽象方法。最初在中指定**PEP 302**，这种方法是为了在sys.meta_path基于路径的导入子系统中使用。

*在版本3.4中更改：*返回None时调用，而不是提高 NotImplementedError。

*类*importlib.abc.MetaPathFinder

表示元路径查找器的抽象基类。为了兼容性，这是一个子类Finder。

3.3版本的新功能

find_spec（*全名*，*路径*，*目标=无*）

一种为指定模块查找规范的抽象方法。如果这是顶级导入，*路径*将会是None。否则，这是搜索一个子包或模块，*路径*将是__path__父包的值。如果找不到规格，None则返回。传入时，target是一个模块对象，查找器可以使用该对象来更具教育性地猜测返回的规范。

3.4版新增功能

find_module（*全名*，*路径*）

传统方法，用于查找指定模块的加载程序。如果这是顶级导入，*路径*将会是None。否则，这是搜索一个子包或模块，*路径*将是__path__父包的值。如果找不到加载程序，None则返回。

如果find_spec()被定义，则提供向后兼容的功能。

*在版本3.4中更改：*返回None时调用，而不是提高 NotImplementedError。可以find_spec()用来提供功能。

*自3.4版弃用：*find_spec()改为使用。

invalidate_caches ( )

一个可选的方法，被调用时应该使查找器使用的任何内部缓存失效。importlib.invalidate_caches() 在使所有查找器的缓存无效时使用sys.meta_path。

*在版本3.4中更改：*返回None时调用，而不是NotImplemented。

*类*importlib.abc. PathEntryFinder

表示路径条目查找程序的抽象基类。虽然它有一些相似之处MetaPathFinder ，但 PathEntryFinder 意味着仅在基于路径的导入子系统中使用PathFinder。这个ABC只是Finder出于兼容性原因的一个子类。

3.3版本的新功能

find_spec (*全名*，*目标=无*)

一种为指定模块查找规范的抽象方法。取景器将仅在其分配的路径条目内搜索模块。如果找不到规格，None则返回。传入时，target 是一个模块对象，查找器可以使用该对象来更具教育性地猜测返回的规范。

3.4版新增功能

find_loader (*全名*)

传统方法，用于查找指定模块的加载程序。返回2元组，其中 是构成名称空间包的一部分的文件系统位置序列。加载器可以在指定文件系统位置对名称空间包的贡献时指定。一个空的列表可以用来表示加载器不是名称空间包的一部分。如果is 和 是空列表，则没有找到加载器或名称空间包的位置（即未找到该模块的任何内容）。(loader, portion)portionNoneportionportionloaderNoneportion

如果find_spec()被定义，则提供向后兼容的功能。

*版本3.4中更改：*返回而不是提高。使用时可提供的功能。(None, [])NotImplementedErrorfind_spec()

*自3.4版弃用：*find_spec()改为使用。

find_module (*全名*)

其具体实现Finder.find_module()等同于self.find_loader(fullname)[0]。

*自3.4版弃用：*find_spec()改为使用。

invalidate_caches ( )

一个可选的方法，被调用时应该使查找器使用的任何内部缓存失效。PathFinder.invalidate_caches() 在使所有缓存查找器的缓存无效时使用。

*类*importlib.abc. Loader

加载器的抽象基类。看到**PEP 302**用于装载机的确切定义。

create_module (*spec*)

导入模块时返回要使用的模块对象的方法。该方法可能会返回None ，表明应该发生默认的模块创建语义。

3.4版新增功能

*在3.5版本中进行了更改：*从Python 3.6开始，此方法在exec_module()定义时不会是可选的。

exec_module(*模块*)

抽象方法，用于在导入或重新加载模块时在其自己的名称空间中执行模块。exec_module()调用模块时应已初始化。当这个方法存在时，create_module()必须定义。

3.4版新增功能

*在版本3.6中更改：*create_module()也必须定义。

load_module(*全名*)

加载模块的传统方法。如果模块不能被加载，ImportError则被抬起，否则被加载的模块被返回。

如果所请求的模块已经存在sys.modules，则应该使用并重新加载该模块。否则，加载器应该创建一个新模块并sys.modules在任何加载开始之前将其插入，以防止导入引发递归。如果加载程序插入了一个模块并且加载失败，则加载程序必须从中删除它sys.modules; sys.modules在加载器开始执行之前已经存在的模块应该单独放置（参见importlib.util.module_for_loader()）。

加载器应该在模块上设置几个属性。（请注意，某些模块重新加载时，这些属性可能会发生变化）：

- __name__
  模块的名称。
- __file__
  模块数据存储位置的路径（未为内置模块设置）。
- __cached__
  应该存储模块的已编译版本的路径（当属性不合适时不会设置）。
- __path__
  指定包内搜索路径的字符串列表。该属性未在模块上设置。
- __package__
  模块/包的父包。如果模块是顶层的，那么它有一个空字符串的值。该importlib.util.module_for_loader()装饰可以处理的细节__package__。
- __loader__
  加载器用于加载模块。该importlib.util.module_for_loader()装饰可以处理的细节__package__。

何时exec_module()可用，则提供了向后兼容的功能。

*在3.4版本中更改：*提高ImportError调用，而不是时候NotImplementedError。提供功能时exec_module()可用。

*自3.4版弃用：*推荐用于加载模块的API是exec_module()（和create_module()）。加载器应该实现它而不是load_module（）。当执行exec_module（）时，导入机器负责load_module（）的所有其他职责。

module_repr（*模块*）

传统方法，在实现时计算并返回给定模块的repr，作为字符串。模块类型的默认repr（）将根据需要使用此方法的结果。

3.3版本的新功能

*在版本3.4中更改：*可选，而不是抽象方法。

*自3.4版弃用：*导入机器现在自动处理。

*类*importlib.abc. ResourceLoader

实现可选 的加载器的抽象基类**PEP 302**协议，用于从存储后端加载任意资源。

abstractmethod get_data（*路径*）

一种抽象方法，用于返回位于*路径中*的数据的字节。具有允许存储任意数据的文件类存储后端的加载器可以实现这种抽象方法，以直接访问存储的数据。OSError如果找不到*路径*将被提出。的*路径*预计使用模块来构建 __file__属性或从包装中的一个项目__path__。

*在版本3.4中更改：*引发，OSError而不是NotImplementedError。

*类*importlib.abc. InspectLoader

实现可选 的加载器的抽象基类用于检查模块的装载机的 **PEP 302**协议。

get_code（*全名*）

返回模块的代码对象，或者None如果模块没有代码对象（例如，对于内置模块就是这种情况）。 ImportError如果加载程序找不到所需的模块，请引发。

> **注意：** 虽然该方法有一个默认的实现，但是如果可能的话，建议它被覆盖。

*版本3.4中更改：*不再提供抽象和具体实现。

abstractmethod get_source（*fullname* ）

一种抽象方法来返回模块的来源。它使用通用换行符作为文本字符串返回，将所有可识别的行分隔符转换为'\n'字符。返回None 如果没有源可用（例如内置模块）。ImportError如果加载程序找不到指定的模块，则引发 。

*在版本3.4中更改：*引发，ImportError而不是NotImplementedError。

is_package（*全名*）

如果模块是包，则返回true值的抽象方法，否则返回false值。 ImportError如果加载程序找不到该模块，则会引发 此问题。

*在版本3.4中更改：*引发，ImportError而不是NotImplementedError。

static source_to_code（*data*，*path ='<string>'*）

从Python源代码创建一个代码对象。

的*数据*参数可以是任何的compile()功能支持（即字符串或字节）。该*路径*参数应该是“路径”的源代码源自，其中其可以是一个抽象的概念（例如，在压缩文件中的位置）。

通过后续的代码对象，可以通过运行在模块中执行它。exec(code, module.__dict__)

3.4版新增功能

*在版本3.5中更改：*使方法静态。

exec_module(*模块*)

执行Loader.exec_module()。

3.4版新增功能

load_module(*全名*)

执行Loader.load_module()。

*从版本3.4开始弃用：*exec_module()改为使用。

*类*importlib.abc.ExecutionLoader

一个抽象的基类继承了InspectLoader它，当它被实现时，可以帮助一个模块作为脚本来执行。ABC表示可选**PEP 302**协议。

abstractmethod get_filename(*fullname*)

一个抽象方法，用于返回__file__指定模块的值。如果没有路径可用，ImportError则引发。

如果源代码可用，那么该方法应该返回源文件的路径，而不管是否使用字节码来加载模块。

*在版本3.4中更改：*引发，ImportError而不是NotImplementedError。

class importlib.abc.FileLoader(*全名*, *路径*)

一个抽象基类，其从继承ResourceLoader和 ExecutionLoader，提供的具体实现ResourceLoader.get_data()和ExecutionLoader.get_filename()。

该*全称*参数是模块加载程序处理的完全解析名称。该*路径*参数是路径，该模块的文件。

3.3版本的新功能

name

加载器可以处理的模块的名称。

path

模块文件的路径。

load_module(*全名*)

呼叫超级的load_module()。

*自3.4版弃用：*Loader.exec_module()改为使用。

abstractmethod get_filename（*fullname*）

退货path。

abstractmethod get_data（*路径*）

作为二进制文件读取*路径*并从中返回字节。

*类*importlib.abc.SourceLoader

用于实现源文件（可选）和字节码文件加载的抽象基类。该类从两者继承，ResourceLoader并 ExecutionLoader要求执行：

- ResourceLoader.get_data()
- ExecutionLoader.get_filename()

  只应该返回源文件的路径; 不支持无源加载。

这个类定义的抽象方法是添加可选的字节码文件支持。不实现这些可选方法（或导致它们引发NotImplementedError）会导致加载器只能使用源代码。实现这些方法允许加载器使用源代码*和*字节代码文件; 它不允许 只提供字节码的*无源*加载。字节码文件是一种优化，可以通过删除Python编译器的解析步骤来加速加载，因此不会暴露字节码特定的API。

path_stats（*路径*）

可选的抽象方法，它返回一个dict包含有关指定路径的元数据。支持的字典键是：

- 'mtime'（强制）：表示源代码修改时间的整数或浮点数;
- 'size'（可选）：源代码的字节大小。

字典中的任何其他键都将被忽略，以允许将来的扩展。如果路径无法处理，OSError则引发。

3.3版本的新功能

*在版本3.4中更改：*升起OSError而不是NotImplementedError。

path_mtime（*路径*）

可选抽象方法，返回指定路径的修改时间。

*自3.3版弃用：*此方法已弃用，以支持path_stats()。您不必实现它，但它仍然可用于兼容性目的。提高OSError如果路径无法处理。

*在版本3.4中更改：*升起OSError而不是NotImplementedError。

set_data（*路径*，*数据*）

将指定字节写入文件路径的可选抽象方法。任何不存在的中间目录都将被自动创建。

由于路径为只读（errno.EACCES/ PermissionError），因此写入路径失败时，请勿传播异常。

*在版本3.4中更改：*NotImplementedError调用时不再提高。

get_code（*全名*）
具体实施InspectLoader.get_code()。

exec_module（*模块*）

具体实施Loader.exec_module()。

3.4版新增功能

load_module（*全名*）
具体实施Loader.load_module()。

*自3.4版弃用：*exec_module()改为使用。

get_source（*全名*）
具体实施InspectLoader.get_source()。

is_package（*全名*）
具体实施InspectLoader.is_package()。如果某个模块的文件路径（由提供的文件路径ExecutionLoader.get_filename()）是一个__init__在文件扩展名被删除**并且**模块名称本身没有结束时命名 的文件，则该模块被确定为一个包 __init__。

31.5.4。 importlib.machinery- 进口商和路径挂钩

源代码： Lib / importlib / machinery.py

该模块包含帮助import 查找和加载模块的各种对象。

importlib.machinery.SOURCE_SUFFIXES
表示源模块的识别文件后缀的字符串列表。

3.3版本的新功能

importlib.machinery.DEBUG_BYTECODE_SUFFIXES
表示非优化字节码模块的文件后缀的字符串列表。

3.3版本的新功能

*自3.5版以来已弃用：*BYTECODE_SUFFIXES改为使用。

importlib.machinery.OPTIMIZED_BYTECODE_SUFFIXES
代表优化字节码模块的文件后缀的字符串列表。

3.3版本的新功能

*自3.5版以来已弃用：*BYTECODE_SUFFIXES改为使用。

importlib.machinery.BYTECODE_SUFFIXES
代表字节码模块（包括前导点）的识别文件后缀的字符串列表。

3.3版本的新功能

*在版本3.5中更改：*该值不再依赖于__debug__。

importlib.machinery.EXTENSION_SUFFIXES

表示扩展模块的识别文件后缀的字符串列表。

3.3版本的新功能

importlib.machinery.all_suffixes()

返回表示由标准导入机制识别的模块的所有文件后缀的字符串组合列表。这是代码的助手，它只需要知道文件系统路径是否可能引用模块而不需要关于模块类型的任何细节（例如，inspect.getmodulename()）。

3.3版本的新功能

*类*importlib.machinery.BuiltinImporter

内置模块的导入器。列出所有已知的内置模块sys.builtin_module_names。这个类实现了importlib.abc.MetaPathFinder和 importlib.abc.InspectLoaderABCs。

只有类方法是由这个类定义的，以减轻实例化的需要。

*在版本3.5中更改：*作为一部分**PEP 489**，内建进口商现在执行 Loader.create_module()和Loader.exec_module()

*类*importlib.machinery.FrozenImporter

冷冻模块的进口商。这个类实现了 importlib.abc.MetaPathFinder 和importlib.abc.InspectLoaderABCs。

只有类方法是由这个类定义的，以减轻实例化的需要。

*类*importlib.machinery.WindowsRegistryFinder

Finder for Windows注册表中声明的模块。这个类实现了importlib.abc.MetaPathFinderABC。

只有类方法是由这个类定义的，以减轻实例化的需要。

3.3版本的新功能

*自从版本3.6开始弃用：*site改为使用配置。默认情况下，Python的未来版本可能无法启用此查找器。

*类*importlib.machinery.PathFinder

甲搜索用于sys.path与包__path__属性。这个类实现了importlib.abc.MetaPathFinderABC。

只有类方法是由这个类定义的，以减轻实例化的需要。

classmethod find_spec(*fullname*，*path* = *None*，*target* = *None*)

试图找到一种类方法规范为通过指定的模块*全名*上sys.path，或者，如果所定义，上*路径*。对于每个搜索到的路径条目，都会 sys.path_importer_cache被检查。如果找到一个非错误的对象，那么它将用作路径条目查找器来查找正在搜索的模块。如果未找到任何条目 sys.path_importer_cache，则sys.path_hooks搜索路径条目的查找器，并且如果找到，则sys.path_importer_cache与被查询关于该模块一起被存储。如果找不到查找器，则将None其存储在缓存中并返回。

3.4版新增功能

*在版本3.5中更改：*如果当前工作目录 - 由空字符串表示 - 不再有效，则None返回但不缓存值sys.path_importer_cache。

classmethod find_module（*fullname*，*path = None* ）

一个传统的包装find_spec()。

*自3.4版弃用：*find_spec()改为使用。

classmethod invalidate_caches（ ）

呼叫importlib.abc.PathEntryFinder.invalidate_caches()存储在中的所有查找器sys.path_importer_cache。

*在版本3.4中更改：*sys.path_hooks使用当前工作目录''（即空字符串）调用对象。

*类*importlib.machinery.FileFinder（*路径*，**loader_details* ）

importlib.abc.PathEntryFinder从文件系统获得缓存的具体实现。

该*路径*参数是其取景器是负责搜索的目录。

所述*loader_details*参数是的2项的元组的可变数目各自含有一个加载程序和文件的后缀序列加载器识别。预计装入程序是可接受的，它接受模块名称的两个参数和找到的文件的路径。

取景器将根据需要缓存目录内容，为每个模块搜索进行统计调用以验证缓存未过期。因为缓存过期依赖于操作系统的文件系统状态信息的粒度，所以存在潜在的争用条件：搜索模块，创建新文件，然后搜索新文件所代表的模块。如果操作发生的速度足以满足stat调用的粒度，那么模块搜索将失败。为防止这种情况发生，当您动态创建模块时，请务必致电importlib.invalidate_caches()。

3.3版本的新功能

path

查找器将搜索的路径。

find_spec（*全名*，*目标=无*）

试图找到该规范来处理*全名*内path。

3.4版新增功能

find_loader（*全名*）

试图找到装载机处理*全名*内path。

invalidate_caches ( )

清除内部缓存。

classmethod path_hook (* *loader_details*)

一个返回闭包的类方法sys.path_hooks。 FileFinder闭包使用直接给予闭包的路径参数和*loader_details* 间接返回一个实例。

如果闭包的参数不是现有目录， ImportError则会引发。

class importlib.machinery. SourceFileLoader (*全名* , *路径*)

具体实现的importlib.abc.SourceLoader通过继承importlib.abc.FileLoader和提供的其他方法的一些具体实现。

3.3版本的新功能

name

该加载器将处理的模块的名称。

path

源文件的路径。

is_package (*全名*)

如果path看起来是一个包，则返回true 。

path_stats (*路径*)

具体实施importlib.abc.SourceLoader.path_stats()。

set_data (*路径* , *数据*)

具体实施importlib.abc.SourceLoader.set_data()。

load_module (*name = None*)

importlib.abc.Loader.load_module()指定要加载的模块名称的具体实现是可选的。

*自从版本3.6开始弃用：*importlib.abc.Loader.exec_module()改为使用。

class importlib.machinery. SourcelessFileLoader (*全名* , *路径*)

其具体实现importlib.abc.FileLoader可以导入字节码文件（即不存在源代码文件）。

请注意，直接使用字节码文件（从而不是源代码文件）会禁止所有Python实现或更改字节码格式的Python新版本使用您的模块。

3.3版本的新功能

name

加载器将处理的模块的名称。

path

字节码文件的路径。

is_package (*全名*)
确定模块是否是基于的包path。

get_code (*全名*)
返回name从中创建的代码对象path。

get_source (*全名*)
None使用此加载程序时，返回的字节码文件没有源代码。

load_module (*name = None*)

importlib.abc.Loader.load_module()指定要加载的模块名称的具体实现是可选的。

自从版本3.6开始弃用： importlib.abc.Loader.exec_module()改为使用。

class importlib.machinery. ExtensionFileLoader (*全名* , *路径*)
importlib.abc.ExecutionLoader扩展模块的具体实现。

该*全称*参数指定的模块加载器是支持的名称。该*路径*参数是路径扩展模块的文件。

3.3版本的新功能

name
加载程序支持的模块的名称。

path
扩展模块的路径。

create_module (*spec*)
根据给定的规范创建模块对象 **PEP 489**。

3.5版本中的新功能。

exec_module (*模块*)
按照给定的模块对象初始化 **PEP 489**。

3.5版本中的新功能。

is_package (*全名*)
返回True如果文件路径指向一个封装的__init__ 模块基础上EXTENSION_SUFFIXES。

get_code (*全名*)
None作为扩展模块返回缺少代码对象。

get_source (*全名*)
None作为扩展模块返回没有源代码。

get_filename (*全名*)
退货path。

3.4版新增功能

class importlib.machinery.ModuleSpec（*name*，*loader*，***，*origin = None*，*loader_state = None*，*is_package = None* ）

模块的导入系统相关状态的规范。这通常暴露为模块的__spec__属性。在下面的描述中，括号中的名称给出了可直接在模块对象上使用的相应属性。例如。但请注意，虽然这些*值*通常是等价的，但它们可以不同，因为两个对象之间没有同步。因此可以在运行时更新模块，并且不会自动反映。module.__spec__.origin == module.__file__ __path__ __spec__.submodule_search_locations

3.4版新增功能

name

（__name__）

一个字符串，用于模块的完全限定名称。

loader

（__loader__）

用于加载的加载器。对于命名空间包应该设置为None。

origin

（__file__）

加载模块的位置名称，例如内置模块的“内建”和从源加载的模块的文件名。通常应该设置“原点”，但它可能是None（默认），表示它是未指定的。

submodule_search_locations

（__path__）

如果包装（None 否则）在哪里查找子模块的字符串列表。

loader_state

装载（或None）期间使用的额外模块特定数据的容器。

cached

（__cached__）

字符串表示编译模块应存储在哪里（或None）。

parent

（__package__）

（只读）模块作为子模块（或None）所属的软件包的完全限定名称。

has_location

指示模块的“origin”属性是否指向可加载位置的布尔值。

31.5.5。importlib.util- 进口商的实用程序代码

源代码： Lib / importlib / util.py

该模块包含帮助建造进口商的各种对象。

importlib.util.MAGIC_NUMBER

代表字节码版本号的字节。如果您需要加载/写入字节码的帮助，请考虑importlib.abc.SourceLoader。

3.4版新增功能

importlib.util.cache_from_source（*path*，*debug_override* = *None*，*，*optimization* = *None*）

返回 **PEP 3147** /**PEP 488**指向与源*路径*关联的字节编译文件的*路径*。例如，如果 *path*是Python 3.2/foo/bar/baz.py 的返回值 /foo/bar/__pycache__/baz.cpython-32.pyc 。该cpython-32 字符串来自当前的魔术标签（请参阅 get_tag()；如果sys.implementation.cache_tag未定义， NotImplementedError则会引发）。

在*优化*参数用于指定字节码文件的优化水平。空字符串表示没有优化，所以/foo/bar/baz.py 与*优化*的’’将导致字节码路径 /foo/bar/__pycache__/baz.cpython-32.pyc。 None导致使用interpter的优化级别。任何其他值的字符串表示正在使用，所以/foo/bar/baz.py 与*优化*的 2会导致字节码路径 /foo/bar/__pycache__/baz.cpython-32.opt-2.pyc。*优化*的字符串表示形式只能是字母数字形式，否则ValueError会引发。

该*debug_override*参数已被弃用，并且可以用于覆盖系统的价值__debug__。一个True值是设定相当于*优化*为空字符串。一个False值是一样的设置*优化*来1。如果两个*debug_override*的*优化*不None那么TypeError升高。

3.4版新增功能

*改变在3.5版本：*将*优化*添加参数和*debug_override*参数已被否决。

*在版本3.6中更改：*接受类似路径的对象。

importlib.util.source_from_cache（*路径*）

鉴于*通向* a**PEP 3147**文件名，返回关联的源代码文件路径。例如，如果*路径*是/foo/bar/__pycache__/baz.cpython-32.pyc返回的路径 /foo/bar/baz.py。*路径*不需要存在，但是如果它不符合**PEP 3147**或**PEP 488**格式，aValueError被提出。如果sys.implementation.cache_tag没有定义， NotImplementedError则提出。

3.4版新增功能

*在版本3.6中更改：*接受类似路径的对象。

importlib.util.decode_source（*source_bytes*）

解码代表源代码的给定字节，并将其作为带有通用换行符的字符串返回（如所需importlib.abc.InspectLoader.get_source()）。

3.4版新增功能

importlib.util.resolve_name（*名称*，*包*）

将相对模块名称解析为绝对模块名称。

如果**名称**没有前导点，则简单地返回**名称**。这允许使用，例如 不检查是否需要**包**参数。importlib.util.resolve_name('sys', __package__)

ValueError如果**名称**是相对模块名称，但包是假值（例如，None或空字符串），则会引发该错误。 ValueError也提出了一个相对的名字将逃离其包含的包（例如..bacon从spam包内请求）。

3.3版本的新功能

importlib.util.find_spec（*名称*，*包=无*）

查找模块的规格，可以选择相对于指定的**软件包**名称。如果该模块处于sys.modules，则sys.modules[name].__spec__返回（除非规范将被 None设置或未被设置，在这种情况下会ValueError被提出）。否则，使用搜索sys.meta_path完成。None如果没有发现规范则返回。

如果**名称**是子模块（包含一个点），则会自动导入父模块。

名称和**包装的**工作原理与之相同import_module()。

3.4版新增功能

importlib.util.module_from_spec（*spec*）

根据**规格**和 创建一个新模块spec.loader.create_module。

如果spec.loader.create_module 不返回None，则任何预先存在的属性都不会被重置。另外，AttributeError如果在模块上访问**规范**或设置属性时触发，则不会产生 。

此功能优于使用types.ModuleType创建新模块，因为**spec**用于在模块上尽可能多地设置导入控制的属性。

3.5版本中的新功能。

@importlib.util.module_for_loader

甲装饰为importlib.abc.Loader.load_module() 处理选择适当的模块的对象与加载。装饰的方法预计会有一个带有两个位置参数（例如）的调用签名，第二个参数将作为加载器要使用的模块**对象**。请注意，由于假设有两个参数，装饰器不能用于静态方法。load_module(self, module)

装饰方法将按照装载器的预期装入要装入的模块的**名称**。如果未找到该模块，则构建新模块 。无论模块来自哪里，都设置为**self**，并 根据返回的内容（如果可用）进行设置。这些

属性被无条件设置为支持重新加载。sys.modules __loader__ __package__ importlib.abc.InspectLoader.is_package()

如果装饰方法引发了异常并添加了模块 sys.modules，则模块将被移除以防止部分初始化的模块被留在中sys.modules。如果模块已经在sys.modules那里，它将被单独留下。

在版本3.3中更改： __loader__并__package__自动设置（如果可能）。

*改变在3.4版本：*集__name__，无条件地支持重装。__loader__ __package__

*自3.4版弃用：*导入机器现在直接执行此功能提供的所有功能。

@importlib.util.**set_loader**

一个装饰为importlib.abc.Loader.load_module() 设置__loader__ 属性返回的模块上。如果该属性已经设置，装饰器不做任何事情。假定包装方法（ie self）的第一个位置参数是__loader__应该设置的参数。

*改变在3.4版本：*设置__loader__如果设置为None，因为如果该属性不存在。

*自3.4版弃用：*进口机械自动处理。

@importlib.util.**set_package**

一个装饰为importlib.abc.Loader.load_module()设置 __package__属性返回的模块上。如果__package__ 已设置并且具有其他值None，则不会更改。

*自3.4版弃用：*进口机械自动处理。

importlib.util.**spec_from_loader**(*name*, *loader*, *, *origin* = *None*, *is_package* = *None*)

ModuleSpec基于加载程序创建实例的工厂函数。这些参数与ModuleSpec具有相同的含义。该功能使用可用的加载程序 API，例如 InspectLoader.is_package()填写规范中缺少的任何信息。

3.4版新增功能

importlib.util.**spec_from_file_location**(*名称*, *位置*, *, *loader* = *None*, *submodule_search_locations* = *None*)

工厂函数，用于ModuleSpec根据文件的路径创建实例。缺少的信息将通过使用加载器API以及模块将基于文件的含义填充到规范中。

3.4版新增功能

*在版本3.6中更改：*接受类似路径的对象。

class importlib.util.**LazyLoader**(*loader*)

推迟模块加载程序执行的类，直到模块具有访问的属性为止。

该类**仅**适用于定义exec_module()为需要使用哪个模块类型的装载机。出于同样的原因，加载器的 create_module()方法必须返回，None或者其__class__属性可以在不使用插槽的情

况下进行变异的类型。最后，替代放入对象的模块sys.modules将无法正常工作，因为无法安全地在整个解释器中正确替换模块引用; ValueError如果检测到这种替换，则会提高。

> **注意:** 对于启动时间很关键的项目，如果从未使用模块，该类可以潜在地降低加载模块的成本。对于那些启动时间是不是必需的，然后使用这个类的项目是 **严重**由于装载过程中产生的错误信息被气馁推迟，从而发生断章取义。

3.5版本中的新功能。

在 3.6 版本的改变： 开始呼吁 create_module() ，去除兼容性警告 importlib.machinery.BuiltinImporter和 importlib.machinery.ExtensionFileLoader。

classmethod factory (*loader*)

返回一个可调用的静态方法，该调用创建一个懒惰的加载器。这是为了在加载器按类而不是按实例传递的情况下使用。

```
suffixes = importlib.machinery.SOURCE_SUFFIXES
loader = importlib.machinery.SourceFileLoader
lazy_loader = importlib.util.LazyLoader.factory(loader)
finder = importlib.machinery.FileFinder(path, (lazy_loader, suffixes))
```

31.5.6。示例

31.5.6.1。以编程方式导入

要以编程方式导入模块，请使用importlib.import_module()。

```
import importlib

itertools = importlib.import_module('itertools')
```

31.5.6.2。检查是否可以导入模块

如果你需要找出一个模块是否可以在没有实际导入的情况下导入，那么你应该使用importlib.util.find_spec()。

```
import importlib.util
import sys

# For illustrative purposes.
name = 'itertools'

spec = importlib.util.find_spec(name)
if spec is None:
    print("can't find the itertools module")
else:
    # If you chose to perform the actual import ...
    module = importlib.util.module_from_spec(spec)
    spec.loader.exec_module(module)
```

```
    # Adding the module to sys.modules is optional.
    sys.modules[name] = module
```

31.5.6.3。直接导入源文件

要直接导入Python源文件，请使用以下配方（仅适用于Python 3.4和更新版本）：

```
import importlib.util
import sys

# For illustrative purposes.
import tokenize
file_path = tokenize.__file__
module_name = tokenize.__name__

spec = importlib.util.spec_from_file_location(module_name, file_path)
module = importlib.util.module_from_spec(spec)
spec.loader.exec_module(module)
# Optional; only necessary if you want to be able to import the module
# by name later.
sys.modules[module_name] = module
```

31.5.6.4。设置导入程序

对于深度自定义导入，您通常需要实现 导入程序。这意味着管理 事物的发现者和装载者两方面。对于发现者来说，有两种风格可供选择，具体取决于您的需求：元路径查找器或路径查找器。前者是你会穿什么sys.meta_path而后者则是创建使用什么路径进入钩上sys.path_hooks与工作sys.path条目可能创建一个取景器。这个例子将向您展示如何注册您自己的导入器，以便导入将使用它们（用于为您自己创建导入器，请阅读此包中定义的相应类的文档）：

```
import importlib.machinery
import sys

# For illustrative purposes only.
SpamMetaPathFinder = importlib.machinery.PathFinder
SpamPathEntryFinder = importlib.machinery.FileFinder
loader_details = (importlib.machinery.SourceFileLoader,
                  importlib.machinery.SOURCE_SUFFIXES)

# Setting up a meta path finder.
# Make sure to put the finder in the proper location in the list in terms of
# priority.
sys.meta_path.append(SpamMetaPathFinder)

# Setting up a path entry finder.
# Make sure to put the path hook in the proper location in the list in terms
# of priority.
sys.path_hooks.append(SpamPathEntryFinder.path_hook(loader_details))
```

31.5.6.5。近似 importlib.import_module()

导入本身是用Python代码实现的，可以通过importlib公开大部分导入机制。下面通过提供近似实现importlib.import_module()（Python 3.4和更新的importlib用法，Python 3.6和更新的代码的其他部分）来帮助说明importlib公开的各种API。

```
import importlib.util
import sys

def import_module(name, package=None):
    """An approximate implementation of import."""
    absolute_name = importlib.util.resolve_name(name, package)
    try:
        return sys.modules[absolute_name]
    except KeyError:
        pass

    path = None
    if '.' in absolute_name:
        parent_name, _, child_name = absolute_name.rpartition('.')
        parent_module = import_module(parent_name)
        path = parent_module.spec.submodule_search_locations
    for finder in sys.meta_path:
        spec = finder.find_spec(absolute_name, path)
        if spec is not None:
            break
    else:
        raise ImportError(f'No module named {absolute_name!r}')
    module = importlib.util.module_from_spec(spec)
    spec.loader.exec_module(module)
    sys.modules[absolute_name] = module
    if path is not None:
        setattr(parent_module, child_name, module)
    return module
```

32. Python语言服务

Python提供了许多模块来协助处理Python语言。这些模块支持标记化，解析，语法分析，字节码反汇编以及其他各种设施。

这些模块包括：

32.1。 parser- 访问Python解析树

该parser模块为Python的内部解析器和字节码编译器提供了一个接口。该接口的主要目的是允许Python代码编辑Python表达式的分析树并从中创建可执行代码。这比试图将任意Python代码片段解析并修改为字符串更好，因为解析是以与形成应用程序的代码相同的方式执行的。它也更快。

注意： 从Python 2.5开始，使用ast 模块在抽象语法树（AST）生成和编译阶段切入更为方便。

关于这个模块有几点需要注意，这对使用创建的数据结构很重要。这不是编辑Python代码的分析树的教程，但提供了使用parser模块的一些示例。

最重要的是，需要理解内部解析器处理的Python语法。有关语言语法的完整信息，请参阅Python语言参考。解析器本身是Grammar/Grammar根据标准Python发行版中文件中定义的语法规范创建的 。存储在由该模块创建的ST对象中的解析树是由expr()or suite()函数创建的内部解析器的实际输出，如下所述。ST创建的对象sequence2st()忠实地模拟这些结构。请注意，由于该语言的形式语法被修改，所以被认为“正确”的序列的值将因Python的不同版本而异。然而，作为源文本将代码从一个Python版本传输到另一个版本将始终允许在目标版本中创建正确的分析树，唯一的限制是迁移到较老版本的解释器将不支持更新的语言结构。解析树通常不是从一个版本到另一个版本兼容，而源代码始终是向前兼容的。

返回的序列的每个元素st2list()或st2tuple() 具有简单的形式。代表文法中非终端元素的序列总长度大于1。第一个元素是一个整数，用于标识语法中的产品。这些整数在C头文件Include/graminit.h和Python模块中 被赋予了符号名称symbol。序列中的每个附加元素都代表输入字符串中可识别的生产组件：它们总是与父代具有相同形式的序列。该结构的一个重要方面应该注意，用于标识父节点类型的关键字（如ifan中的关键字）if_stmt包含在节点树中，而不需要任何特殊处理。例如，if关键字由元组表示，其中是与所有标记关联的数值，包括用户定义的变量和函数名称。在请求行号信息时返回的另一种形式，同一个标记可能表示为，其中代表找到终端符号的行号。(1, 'if')1NAME(1, 'if', 12)12

终端元素的表示方式大致相同，但没有任何子元素，并添加了已识别的源文本。 if以上关键字的例子是有代表性的。C头文件Include/token.h和Python模块中定义了各种类型的终端符号token。

ST对象不需要支持这个模块的功能，但是有三个目的：允许应用程序分摊处理复杂分析树的代价，提供一个分析树表示，与Python相比，可以节省内存空间列表或元组表示，并且简化在C中创建附加模块来操纵分析树。一个简单的“包装”类可以在Python中创建，以隐藏ST对象的使用。

该parser模块为几个不同的目的定义函数。最重要的目的是创建ST对象并将ST对象转换为其他表示形式，例如解析树和编译代码对象，但也有函数用于查询由ST对象表示的解析树的类型。

也可以看看:

模 symbol

表示分析树内部节点的常用常量。

模 token
表示分析树的叶节点和用于测试节点值的函数的有用常量。

32.1.1。创建ST对象

ST对象可以从源代码或分析树中创建。从源创建ST对象时，将使用不同的函数来创建'eval'和'exec'表单。

parser.expr（*源*）
该expr()函数将参数*源*解析为它的输入。如果解析成功，则会创建ST对象来保存内部分析树表示形式，否则会引发适当的异常。compile(source, 'file.py', 'eval')

parser.suite（*源*）
该suite()函数将参数*源*解析为它的输入。如果解析成功，则会创建ST对象来保存内部分析树表示形式，否则会引发适当的异常。compile(source, 'file.py', 'exec')

parser.sequence2st（*序列*）
该函数接受一个表示为序列的分析树，并尽可能构建内部表示。如果它可以验证该树符合Python语法，并且所有节点都是Python主机版本中的有效节点类型，则会从内部表示形式创建一个ST对象并将其返回给被调用的对象。如果在创建内部表示时出现问题，或者树无法验证，ParserError则会引发异常。不应该假定以这种方式创建的ST对象能够正确编译;当ST对象被传递时，编译引发的正常异常仍可能启动compilest()。这可能表明问题与语法无关（例如MemoryError异常），但也可能由于解析结果等结构del f(0)，它转义了Python解析器，但是由字节码编译器检查。

表示终端令牌的序列可以表示为表单的两元素列表或者表单的三元素列表。如果第三个元素存在，则假定它是有效的行号。行号可以为输入树中的任何终端符号子集指定。(1, 'name')(1, 'name', 56)

parser.tuple2st（*序列*）
这与功能相同sequence2st()。此入口点保持向后兼容。

32.1.2。转换ST对象

无论用于创建它们的输入，ST对象都可以转换为表示为列表或元组树的解析树，也可以编译为可执行代码对象。可以使用或不使用行号信息来提取分析树。

parser.st2list（*st*，*line_info = False*，*col_info = False* ）
该函数在*st*中接受来自调用者的ST对象，并返回表示等效分析树的Python列表。得到的列表表示可用于检查或以列表形式创建新的分析树。只要内存可用于构建列表表示，该函数就不会失败。如果分析树仅用于检查，则st2tuple()应该使用分析树 来减少内存消耗和碎片。当需要列表表示时，此函数比检索元组表示并将其转换为嵌套列表快得多。

如果*line_info*为true，则所有终端令牌都将包含行号信息，作为表示令牌的列表的第三个元素。请注意，提供的行号指定了令牌*结束的行*。如果该标志为错误或省略，则该信息被省略。

parser. st2tuple (*st* , *line_info = False* , *col_info = False*)

该函数在*st中*接受来自调用者的ST对象，并返回表示等效分析树的Python元组。除了返回一个元组而不是一个列表之外，这个函数与之相同st2list()。

如果*line_info*为true，则所有终端令牌都将包含行号信息，作为表示令牌的列表的第三个元素。如果该标志为错误或省略，则该信息被省略。

parser. compilest (*st* , *filename ='<syntax-tree>'*)

可以在ST对象上调用Python字节编译器，以生成可用作对内置 函数exec()或eval()函数调用的一部分的代码对象。该函数为编译器提供接口，使用由*filename*参数指定的源文件名将内部分析树从*st*传递到解析器。为*文件名*提供的默认值 指示源是ST对象。

编译ST对象可能会导致与编译相关的异常; 一个例子可能是SyntaxError由分析树引起的：这个语句在Python的形式语法中被认为是合法的，但不是一个合法的语言结构。所述凸起此条件实际上是由Python字节编译器通常，这就是为什么它可以在此时通过提高生成模块。编译失败的大多数原因可以通过检查分析树来以编程方式进行诊断。 del f(0) SyntaxErrorparser

32.1.3。ST对象查询

提供了两个函数，它们允许应用程序确定ST是作为表达式还是套件创建的。既不这些功能可以被用来确定是否有ST，从源代码经由创建expr()或 suite()或从经由解析树sequence2st()。

parser. isexpr (*st*)

当*st*表示一个'eval'表单时，这个函数返回true，否则返回false。这很有用，因为通常使用现有的内置函数不能查询代码对象的信息。请注意，由此创建的代码对象compilest()也不能像这样查询，并且与由内置compile()函数创建的代码对象相同。

parser. issuite (*st*)

这个函数反映isexpr()了它会报告一个ST对象是否表示一个'exec'表单，通常称为“套件”。假设这个函数等价于是不安全的，因为将来可能会支持其他语法片段。 not isexpr(st)

32.1.4。异常和错误处理

解析器模块定义了一个异常，但也可能会从Python运行时环境的其他部分传递其他内置异常。查看每个函数以获取有关它可能引发的异常的信息。

*异常*parser. ParserError

解析器模块内发生故障时引发异常。这通常是针对验证失败产生的，而不是SyntaxError在正常解析期间产生的内置 。异常参数可以是描述失败原因的字符串，也可以是包含导致传递给解析树的故障的序列的元组sequence2st() 以及解释性字符串。调用sequence2st()需

要能够处理任何一种类型的异常，而对模块中其他功能的调用只需要知道简单的字符串值。

请注意，函数compilest()，expr()并suite()可能引发解析和编译过程通常引发的异常。这些措施包括内置的例外MemoryError， OverflowError，SyntaxError，和SystemError。在这些情况下，这些例外具有通常与其相关的所有含义。有关详细信息，请参阅每个功能的说明。

32.1.5。ST对象

ST对象之间支持有序和等式比较。酸洗ST对象（使用pickle模块）也被支持。

parser.STType
返回的对象的类型expr()，suite()和 sequence2st()。

ST对象具有以下方法：

ST.compile(*filename ='<syntax-tree>'*)
和...一样。compilest(st, filename)

ST.isexpr( )
和...一样isexpr(st)。

ST.issuite( )
和...一样issuite(st)。

ST.tolist(*line_info = False，col_info = False*)
和...一样。st2list(st, line_info, col_info)

ST.totuple(*line_info = False，col_info = False*)
和...一样。st2tuple(st, line_info, col_info)

32.1.6。例子：仿真 compile()

虽然许多有用的操作可能发生在解析和字节码生成之间，但最简单的操作是什么都不做。为此，使用parser模块产生中间数据结构就相当于代码

```
>>> code = compile('a + 5', 'file.py', 'eval')
>>> a = 5
>>> eval(code)
10
```

使用该parser模块的等效操作稍长一些，并允许将中间内部分析树保留为ST对象：

```
>>> import parser
>>> st = parser.expr('a + 5')
>>> code = st.compile('file.py')
>>> a = 5
```

```
>>> eval(code)
10
```

一个需要ST和代码对象的应用程序可以将这些代码打包成一些可用的函数：

```
import parser

def load_suite(source_string):
    st = parser.suite(source_string)
    return st, st.compile()

def load_expression(source_string):
    st = parser.expr(source_string)
    return st, st.compile()
```

32.2。ast- 抽象语法树

源代码： Lib / ast.py

该ast模块帮助Python应用程序处理Python抽象语法语法的树。抽象语法本身可能随着每个Python版本而改变; 这个模块有助于以编程方式找出当前语法的样子。

抽象语法树可以通过ast.PyCF_ONLY_AST作为标志传递给compile()内置函数或使用parse()本模块中提供的帮助程序来生成。结果将是类的所有继承对象的树ast.AST。可以使用内置compile()函数将抽象语法树编译为Python代码对象。

32.2.1。节点类

*类*ast.AST

这是所有AST节点类的基础。实际的节点类是从该Parser/Python.asdl文件派生的，该文件在下面再现。它们在_astC模块中定义并重新导出ast。

在抽象语法中为每个左侧符号定义了一个类（例如，ast.stmt或ast.expr）。另外，右边的每个构造函数都有一个类定义; 这些类继承了左侧树的类。例如，ast.BinOp继承自ast.expr。对于带有替代选项（又名“和”）的生产规则，左侧类是抽象的：只创建特定构造函数节点的实例。

_fields

每个具体类都有一个_fields给出所有子节点名称的属性。

具体类的每个实例对于每个子节点都有一个属性，类型在语法中定义。例如，ast.BinOp实例具有left类型的属性ast.expr。

如果这些属性在语法中标记为可选（使用问号），则值可能为None。如果这些属性可以有零个或多个值（用星号标记），则这些值表示为Python列表。在编译AST时，所有可能的属性都必须存在且具有有效值compile()。

lineno
col_offset

的实例ast.expr和ast.stmt亚类lineno和col_offset属性。这lineno是源文本的行数（1-索引，所以第一行是第一行），并且col_offset是生成该节点的第一个令牌的UTF-8字节偏移量。记录UTF-8偏移是因为解析器在内部使用UTF-8。

类的构造函数ast.T解析其参数如下：

- 如果有位置参数，则必须有多少项目T._fields; 他们将被分配为这些名称的属性。
- 如果有关键字参数，他们会将相同名称的属性设置为给定值。

例如，要创建并填充ast.UnaryOp节点，可以使用

```
node = ast.UnaryOp()
node.op = ast.USub()
```

```
node.operand = ast.Num()
node.operand.n = 5
node.operand.lineno = 0
node.operand.col_offset = 0
node.lineno = 0
node.col_offset = 0
```

或更紧凑

```
node = ast.UnaryOp(ast.USub(), ast.Num(5, lineno=0, col_offset=0),
                   lineno=0, col_offset=0)
```

32.2.2。抽象语法

抽象语法目前定义如下：

```
-- ASDL's 7 builtin types are:
-- identifier, int, string, bytes, object, singleton, constant
--
-- singleton: None, True or False
-- constant can be None, whereas None means "no value" for object.

module Python
{
    mod = Module(stmt* body)
        | Interactive(stmt* body)
        | Expression(expr body)

        -- not really an actual node but useful in Jython's typesystem.
        | Suite(stmt* body)

    stmt = FunctionDef(identifier name, arguments args,
                       stmt* body, expr* decorator_list, expr? returns)
          | AsyncFunctionDef(identifier name, arguments args,
                             stmt* body, expr* decorator_list, expr? returns)

          | ClassDef(identifier name,
             expr* bases,
             keyword* keywords,
             stmt* body,
             expr* decorator_list)
          | Return(expr? value)

          | Delete(expr* targets)
          | Assign(expr* targets, expr value)
          | AugAssign(expr target, operator op, expr value)
          -- 'simple' indicates that we annotate simple name without parens
          | AnnAssign(expr target, expr annotation, expr? value, int simple)

          -- use 'orelse' because else is a keyword in target languages
          | For(expr target, expr iter, stmt* body, stmt* orelse)
          | AsyncFor(expr target, expr iter, stmt* body, stmt* orelse)
          | While(expr test, stmt* body, stmt* orelse)
          | If(expr test, stmt* body, stmt* orelse)
          | With(withitem* items, stmt* body)
```

```
        | AsyncWith(withitem* items, stmt* body)

        | Raise(expr? exc, expr? cause)
        | Try(stmt* body, excepthandler* handlers, stmt* orelse, stmt* finalbody)
        | Assert(expr test, expr? msg)

        | Import(alias* names)
        | ImportFrom(identifier? module, alias* names, int? level)

        | Global(identifier* names)
        | Nonlocal(identifier* names)
        | Expr(expr value)
        | Pass | Break | Continue

        -- XXX Jython will be different
        -- col_offset is the byte offset in the utf8 string the parser uses
        attributes (int lineno, int col_offset)

        -- BoolOp() can use left & right?
expr = BoolOp(boolop op, expr* values)
      | BinOp(expr left, operator op, expr right)
      | UnaryOp(unaryop op, expr operand)
      | Lambda(arguments args, expr body)
      | IfExp(expr test, expr body, expr orelse)
      | Dict(expr* keys, expr* values)
      | Set(expr* elts)
      | ListComp(expr elt, comprehension* generators)
      | SetComp(expr elt, comprehension* generators)
      | DictComp(expr key, expr value, comprehension* generators)
      | GeneratorExp(expr elt, comprehension* generators)
      -- the grammar constrains where yield expressions can occur
      | Await(expr value)
      | Yield(expr? value)
      | YieldFrom(expr value)
      -- need sequences for compare to distinguish between
      -- x < 4 < 3 and (x < 4) < 3
      | Compare(expr left, cmpop* ops, expr* comparators)
      | Call(expr func, expr* args, keyword* keywords)
      | Num(object n) -- a number as a PyObject.
      | Str(string s) -- need to specify raw, unicode, etc?
      | FormattedValue(expr value, int? conversion, expr? format_spec)
      | JoinedStr(expr* values)
      | Bytes(bytes s)
      | NameConstant(singleton value)
      | Ellipsis
      | Constant(constant value)

      -- the following expression can appear in assignment context
      | Attribute(expr value, identifier attr, expr_context ctx)
      | Subscript(expr value, slice slice, expr_context ctx)
      | Starred(expr value, expr_context ctx)
      | Name(identifier id, expr_context ctx)
      | List(expr* elts, expr_context ctx)
      | Tuple(expr* elts, expr_context ctx)

      -- col_offset is the byte offset in the utf8 string the parser uses
      attributes (int lineno, int col_offset)
```

```
    expr_context = Load | Store | Del | AugLoad | AugStore | Param

    slice = Slice(expr? lower, expr? upper, expr? step)
          | ExtSlice(slice* dims)
          | Index(expr value)

    boolop = And | Or

    operator = Add | Sub | Mult | MatMult | Div | Mod | Pow | LShift
                 | RShift | BitOr | BitXor | BitAnd | FloorDiv

    unaryop = Invert | Not | UAdd | USub

    cmpop = Eq | NotEq | Lt | LtE | Gt | GtE | Is | IsNot | In | NotIn

    comprehension = (expr target, expr iter, expr* ifs, int is_async)

    excepthandler = ExceptHandler(expr? type, identifier? name, stmt* body)
                    attributes (int lineno, int col_offset)

    arguments = (arg* args, arg? vararg, arg* kwonlyargs, expr* kw_defaults,
                 arg? kwarg, expr* defaults)

    arg = (identifier arg, expr? annotation)
           attributes (int lineno, int col_offset)

    -- keyword arguments supplied to call (NULL identifier for **kwargs)
    keyword = (identifier? arg, expr value)

    -- import name with optional 'as' alias.
    alias = (identifier name, identifier? asname)

    withitem = (expr context_expr, expr? optional_vars)
}
```

32.2.3。ast助手

除了节点类之外，ast模块还定义了这些用于遍历抽象语法树的实用函数和类：

ast.**parse**(*source*，*filename='<unknown>'*，*mode='exec'*)

将源解析为AST节点。相当于。compile(source, filename, mode, ast.PyCF_ONLY_AST)

> **警告：** 由于Python AST编译器中的堆栈深度限制，可能会使Python解释器崩溃为足够大/复杂的字符串。

ast.**literal_eval**(*node_or_string*)

安全地评估表达式节点或包含Python文字或容器显示的字符串。提供的字符串或节点可能只包含以下Python文字结构：字符串，字节，数字，元组，列表，字典，集合，布尔值和None。

这可用于安全地评估包含来自不受信任来源的Python值的字符串，而无需自行解析值。它不能评估任意复杂的表达式，例如涉及操作符或索引。

> **警告：** 由于Python AST编译器中的堆栈深度限制，可能会使Python解释器崩溃为足够大/复杂的字符串。

*在版本3.2中更改：*现在允许字节和设置文字。

ast.get_docstring(*node*, *clean* = *True*)

返回给定的文档字符串*节点*（它必须是一个FunctionDef，ClassDef或Module节点），或None如果它没有文档字符串。如果*clean*是真的，用清理docstring的缩进inspect.cleandoc()。

ast.fix_missing_locations(*节点*)

当你编译一个节点树compile()，编译器期望lineno，并col_offset为支持他们的每一个节点的属性。填充生成的节点非常繁琐，所以这个助手通过将这些属性设置为父节点的值来递增地添加这些属性。它从*节点*开始递归地工作。

ast.increment_lineno(*节点*, *n* = *1*)

递增树中的每个节点的起始行号*节点*通过*Ñ*。这对于将代码移动到文件中的其他位置很有用。

ast.copy_location(*new_node*, *old_node*)

如果可能的话，将源位置（lineno和col_offset）从*old_node复制*到*new_node*，并返回*new_node*。

ast.iter_fields(*节点*)

为*节点*上存在的每个字段生成一个元组。(fieldname, value)node._fields

ast.iter_child_nodes(*节点*)

产生的所有直接子节点*的节点*，也就是在节点和在节点列表领域的所有项目的所有字段。

ast.walk(*节点*)

以*节点*（包括*节点*本身）的形式递归地产生树中的所有后代节点，并且没有指定顺序。如果你只想修改节点而不关心上下文，这很有用。

*类*ast.NodeVisitor

一个节点访问者基类，它遍历抽象语法树并为找到的每个节点调用访问者函数。该函数可能会返回该visit()方法转发的值。

这个类意图被子类化，子类添加访问者方法。

visit(*节点*)

访问一个节点。默认实现调用称为where 的方法，其中*classname*是节点类的名称，或者如果该方法不存在。self.visit_*classname*generic_visit()

generic_visit(*节点*)

该访问者调用visit()该节点的所有子节点。

请注意，除非访问者generic_visit()自己调用或访问它们，否则不会访问具有自定义访问者方法的节点的子节点。

NodeVisitor如果您想在遍历过程中将更改应用于节点，请不要使用它。对于这个特殊的访问者存在（ NodeTransformer ）允许修改。

*类*ast. NodeTransformer

一个NodeVisitor遍历抽象语法树并允许修改节点的子类。

该NodeTransformer会走的AST，并使用访问者方法的返回值来替换或删除旧节点。如果visitor方法的返回值是None，则该节点将从其位置移除，否则将被替换为返回值。返回值可能是原始节点，在这种情况下不会发生替换。

下面是一个示例转换器，它将所有名称查找（ foo ）重写为data['foo']：

```
class RewriteName(NodeTransformer):

    def visit_Name(self, node):
        return copy_location(Subscript(
            value=Name(id='data', ctx=Load()),
            slice=Index(value=Str(s=node.id)),
            ctx=node.ctx
        ), node)
```

请记住，如果您正在操作的节点具有子节点，则必须自己变换子节点或首先调用节点的generic_visit() 方法。

对于作为语句集合（适用于所有语句节点）的一部分的节点，访问者也可以返回节点列表，而不仅仅是单个节点。

通常你使用这样的变压器：

```
node = YourTransformer().visit(node)
```

ast. dump（ *node*，*annotate_fields = True*，*include_attributes = False* ）

返回*节点中*树的格式化转储。这主要用于调试目的。返回的字符串将显示字段的名称和值。这使代码无法评估，因此如果需要评估，必须将*annotate_fields*设置为False。行号和列偏移等属性默认不会被转储。如果需要， 可以将*include_attributes*设置为True。

也可以看看： Green Tree Snakes是一个外部文档资源，它在处理Python AST时有很好的细节。

32.3。 symtable- 访问编译器的符号表

源代码： Lib / symtable.py

符号表由AST编译器在字节码生成之前生成。符号表负责计算代码中每个标识符的范围。 symtable提供了一个界面来检查这些表格。

32.3.1。生成符号表

symtable. symtable (*代码* , *文件名* , *compile_type*)

返回 SymbolTablePython 源 *代码* 的顶层。 *filename* 是包含代码的文件的名称。 *compile_type*就像是*模式*参数compile()。

32.3.2。检查符号表

*类*symtable. SymbolTable

块的命名空间表。构造函数不公开。

get_type ( )

返回符号表的类型。可能的值是'class' ， 'module'和'function'。

get_id ( )

返回表格的标识符。

get_name ( )

返回表格的名称。如果表是用于类的话，这是类的名称，如果表是用于函数的函数的名称，或者 'top'表是全局的（ get_type()返回'module' ）。

get_lineno ( )

返回此表格中块的第一行的编号。

is_optimized ( )

返回True如果该表中的当地人可以优化。

is_nested ( )

返回True如果该块是一个嵌套类或功能。

has_children ( )

返回True如果块已经在它嵌套的命名空间。这些都可以用get_children()。

has_exec ( )

True如果该块使用，则返回exec。

get_identifiers ( )

返回此表中符号名称的列表。

lookup (*名字*)
在表中查找*名称*并返回一个Symbol实例。

get_symbols ( )
返回表中Symbol名称的实例列表。

get_children ( )
返回嵌套符号表的列表。

*类*symtable.Function
函数或方法的名称空间。这个类继承 SymbolTable。

get_parameters ( )
将包含参数名称的元组返回给此函数。

get_locals ( )
返回包含本函数名称的元组。

get_globals ( )
在此函数中返回包含全局变量名称的元组。

get_frees ( )
返回包含此函数中自由变量名称的元组。

*类*symtable.Class
类的名称空间。这个类继承SymbolTable。

get_methods ( )
返回包含类中声明的方法名称的元组。

*类*symtable.Symbol
SymbolTable与源中的标识符对应的条目。构造函数不公开。

get_name ( )
返回符号的名称。

is_referenced ( )
True如果符号在其块中使用，则返回。

is_imported ( )
True如果符号是从导入语句创建的，则返回。

is_parameter ( )
True如果符号是参数则返回。

is_global ( )
True如果符号是全局的，则返回。

is_declared_global ( )

True如果符号是用全局语句声明为全局的，则返回。

is_local ( )

True如果符号位于其块的本地，则返回。

is_free ( )

返回True如果符号在其块引用，但没有分配到。

is_assigned ( )

返回True如果符号在其块分配。

is_namespace ( )

True如果名称绑定引入新名称空间，则返回。

如果该名称用作函数或类声明的目标，则这是真实的。

例如：

```
>>> table = symtable.symtable("def some_func(): pass", "string", "exec")
>>> table.lookup("some_func").is_namespace()
True
```

请注意，单个名称可以绑定到多个对象。如果结果是True，该名称也可以绑定到其他对象，如int或list，但不会引入新的名称空间。

get_namespaces ( )

返回绑定到这个名字的名字空间列表。

get_namespace ( )

返回绑定到这个名字的名字空间。如果绑定了多个名称空间，ValueError则会引发。

32.4。 symbol- 用于Python解析树的常量

源代码： Lib / symbol.py

该模块提供表示分析树内部节点数值的常量。与大多数Python常量不同，它们使用小写名称。请参阅Grammar/GrammarPython发行版中的文件，了解语言语法上下文中名称的定义。名称映射到的特定数值可能会在Python版本之间发生变化。

该模块还提供了一个额外的数据对象：

symbol.sym_name
字典将本模块中定义的常量的数值映射回名称字符串，从而允许生成更多可读的解析树表示。

32.5。 token- 用于Python解析树的常量

源代码： Lib / token.py

该模块提供表示分析树（终端令牌）的叶节点的数值的常量。请参阅Grammar/Grammar Python发行版中的文件，了解语言语法上下文中名称的定义。名称映射到的特定数值可能会在Python版本之间发生变化。

该模块还提供了从数字代码到名称和一些功能的映射。这些函数镜像Python C头文件中的定义。

token.tok_name
字典将本模块中定义的常量的数值映射回名称字符串，从而允许生成更多可读的解析树表示。

token.ISTERMINAL（*x*）
终端令牌值返回true。

token.ISNONTERMINAL（*x*）
对于非终端标记值返回true。

token.ISEOF（*x*）
如果*x*是表示输入结束的标记，则返回true 。

令牌常量是：

token.ENDMARKER
token.NAME
token.NUMBER
token.STRING
token.NEWLINE
token.INDENT
token.DEDENT
token.LPAR
token.RPAR
token.LSQB
token.RSQB
token.COLON
token.COMMA
token.SEMI
token.PLUS
token.MINUS
token.STAR
token.SLASH
token.VBAR

token.AMPER
token.LESS
token.GREATER
token.EQUAL
token.DOT
token.PERCENT
token.LBRACE
token.RBRACE
token.EQEQUAL
token.NOTEQUAL
token.LESSEQUAL
token.GREATEREQUAL
token.TILDE
token.CIRCUMFLEX
token.LEFTSHIFT
token.RIGHTSHIFT
token.DOUBLESTAR
token.PLUSEQUAL
token.MINEQUAL
token.STAREQUAL
token.SLASHEQUAL
token.PERCENTEQUAL
token.AMPEREQUAL
token.VBAREQUAL
token.CIRCUMFLEXEQUAL
token.LEFTSHIFTEQUAL
token.RIGHTSHIFTEQUAL
token.DOUBLESTAREQUAL
token.DOUBLESLASH
token.DOUBLESLASHEQUAL
token.AT
token.ATEQUAL
token.RARROW
token.ELLIPSIS
token.OP
token.AWAIT
token.ASYNC
token.ERRORTOKEN
token.N_TOKENS
token.NT_OFFSET

*版本3.5中已更改：已*添加AWAIT和ASYNC令牌。从Python 3.7开始，“async”和“await”将被标记为NAME 标记，AWAIT并且ASYNC将被删除。

32.6。keyword- 测试Python关键字

源代码： Lib / keyword.py

该模块允许Python程序确定字符串是否为关键字。

keyword.iskeyword（s）

如果s是一个Python关键字，则返回true 。

keyword.kwlist

包含为解释器定义的所有关键字的序列。如果任何关键字被定义为在特定__future__语句生效时才处于活动状态，那么这些关键字也会包含在内。

32.7。 tokenize-分词器为Python源

源代码： Lib / tokenize.py

该tokenize模块为Python源代码提供了一个词法扫描器，并以Python实现。该模块中的扫描器也将评论作为标记返回，从而使其对于实现“漂亮打印机”（包括用于屏幕显示的着色器）非常有用。

为了简化标记流处理，使用通用标记类型返回所有运算符和 分隔符标记。确切的类型可以通过检查返回的指定元组的属性 来确定。 EllipsisOP exact_type tokenize.tokenize()

32.7.1。令牌输入

主要入口点是一个生成器：

tokenize.tokenize（*readline* ）

所述tokenize()发电机需要一个参数，*readline的*，它必须是一个可调用的对象，它提供相同的接口的 io.IOBase.readline()文件对象的方法。对函数的每次调用应该以字节形式返回一行输入。

生成器生成这些成员的5元组：令牌类型; 令牌字符串; 一个2元组ints，用于指定源中源标记开始的行和列; 一个2元组inint，用于指定令牌在源中结束的行和列; 和找到令牌的行。传递的行（最后一个元组项）是*逻辑*行; 包括延续线。5元组以字段名称作为命名元组返回：。 (srow, scol) (erow, ecol) type string start end line

返回的命名元组有一个名为的附加属性 exact_type ，其中包含token.OP令牌的确切运算符类型 。对于所有其他标记类型exact_type 等于指定的元组type字段。

*在版本3.1中更改：*添加了对命名元组的支持。

*在版本3.3中更改：*添加了对exact_type。

tokenize() 根据，通过查找UTF-8 BOM或编码cookie来确定文件的源编码 **PEP 263**。

所有来自token模块的常量也可以从tokenize三个额外的标记类型值中导出 ：

tokenize.COMMENT

令牌值用于表示注释。

tokenize.NL

令牌值用于指示非终止换行符。NEWLINE标记表示Python代码的逻辑行结束; 当一条逻辑代码行在多条物理线路上继续时，会生成NL令牌。

tokenize.ENCODING

标记值，表示用于将源字节解码为文本的编码。返回的第一个标记tokenize()将始终是一个ENCODING标记。

提供另一个功能来反转标记化过程。这对于创建令牌化脚本，修改令牌流和回写修改的脚本的工具很有用。

tokenize.untokenize（*可迭代*）

将令牌转换回Python源代码。该*迭代*必须具有至少两个元件，所述令牌类型和令牌字符串返回序列。任何其他序列元素都将被忽略。

重建的脚本作为单个字符串返回。结果保证将标记返回以匹配输入，以便转换无损并确保往返。保证仅适用于令牌类型和令牌字符串，因为令牌之间的间距（列位置）可能会发生变化。

它返回字节，使用ENCODING标记进行编码，这是第一个输出的标记序列tokenize()。

tokenize()需要检测它标记的源文件的编码。它使用的功能是可用的：

tokenize.detect_encoding（*readline*）

该detect_encoding()函数用于检测应该用于解码Python源文件的编码。它需要一个参数readline，就像tokenize()生成器一样。

它将最多调用readline两次，并返回已使用的编码（作为字符串）和已读入的所有行（未从字节解码）的列表。

它根据在中指定的UTF-8 BOM或编码cookie的存在来检测编码 **PEP 263**。如果BOM和cookie都存在但不同意，则会引发SyntaxError。请注意，如果找到BOM，'utf-8-sig'将作为编码返回。

如果未指定编码，则将'utf-8'返回默认值。

使用open()打开Python源文件：它使用 detect_encoding()来检测文件的编码。

tokenize.open（*文件名*）

使用检测到的编码以只读模式打开文件 detect_encoding()。

3.2版本中的新功能

*异常*tokenize.TokenError

当可能分成多行的文档字符串或表达式未在文件中的任何位置完成时引发，例如：

```
"""Beginning of
docstring
```

要么：

```
[1,
 2,
 3
```

请注意，未封闭的单引号字符串不会导致引发错误。它们被标记为ERRORTOKEN，然后标记它们的内容。

32.7.2。命令行用法

3.3版本的新功能

该tokenize模块可以作为命令行中的脚本执行。这很简单：

```
python -m tokenize [-e] [filename.py]
```

接受以下选项：

-h，--help
    显示此帮助信息并退出

-e，--exact
    使用确切的类型显示令牌名称

如果filename.py被指定，其内容被标记为stdout。否则，在stdin上执行标记。

32.7.3。示例

将浮点文字转换为Decimal对象的脚本重写器示例：

```
from tokenize import tokenize, untokenize, NUMBER, STRING, NAME, OP
from io import BytesIO

def decistmt(s):
    """Substitute Decimals for floats in a string of statements.

    >>> from decimal import Decimal
    >>> s = 'print(+21.3e-5*-.1234/81.7)'
    >>> decistmt(s)
    "print (+Decimal ('21.3e-5')*-Decimal ('.1234')/Decimal ('81.7'))"

    The format of the exponent is inherited from the platform C library.
    Known cases are "e-007" (Windows) and "e-07" (not Windows).  Since
    we're only showing 12 digits, and the 13th isn't close to 5, the
    rest of the output should be platform-independent.

    >>> exec(s)  #doctest: +ELLIPSIS
    -3.21716034272e-0...7

    Output from calculations with Decimal should be identical across all
    platforms.

    >>> exec(decistmt(s))
    -3.217160342717258261933904529E-7
    """
    result = []
    g = tokenize(BytesIO(s.encode('utf-8')).readline)  # tokenize the string
    for toknum, tokval, _, _, _ in g:
        if toknum == NUMBER and '.' in tokval:  # replace NUMBER tokens
            result.extend([
```

```
                (NAME, 'Decimal'),
                (OP, '('),
                (STRING, repr(tokval)),
                (OP, ')')
            ])
        else:
            result.append((toknum, tokval))
    return untokenize(result).decode('utf-8')
```

从命令行标记化的示例。剧本：

```
def say_hello():
    print("Hello, World!")

say_hello()
```

将被标记为以下输出，其中第一列是找到标记的行/列坐标的范围，第二列是标记的名称，最后一列是标记的值（如果有的话）

```
$ python -m tokenize hello.py
0,0-0,0:            ENCODING       'utf-8'
1,0-1,3:            NAME           'def'
1,4-1,13:           NAME           'say_hello'
1,13-1,14:          OP             '('
1,14-1,15:          OP             ')'
1,15-1,16:          OP             ':'
1,16-1,17:          NEWLINE        '\n'
2,0-2,4:            INDENT         '    '
2,4-2,9:            NAME           'print'
2,9-2,10:           OP             '('
2,10-2,25:          STRING         '"Hello, World!"'
2,25-2,26:          OP             ')'
2,26-2,27:          NEWLINE        '\n'
3,0-3,1:            NL             '\n'
4,0-4,0:            DEDENT         ''
4,0-4,9:            NAME           'say_hello'
4,9-4,10:           OP             '('
4,10-4,11:          OP             ')'
4,11-4,12:          NEWLINE        '\n'
5,0-5,0:            ENDMARKER      ''
```

确切的标记类型名称可以使用以下-e选项显示：

```
$ python -m tokenize -e hello.py
0,0-0,0:            ENCODING       'utf-8'
1,0-1,3:            NAME           'def'
1,4-1,13:           NAME           'say_hello'
1,13-1,14:          LPAR           '('
1,14-1,15:          RPAR           ')'
1,15-1,16:          COLON          ':'
1,16-1,17:          NEWLINE        '\n'
2,0-2,4:            INDENT         '    '
2,4-2,9:            NAME           'print'
2,9-2,10:           LPAR           '('
2,10-2,25:          STRING         '"Hello, World!"'
```

```
2,25-2,26:            RPAR           ')'
2,26-2,27:            NEWLINE        '\n'
3,0-3,1:              NL             '\n'
4,0-4,0:              DEDENT         ''
4,0-4,9:              NAME           'say_hello'
4,9-4,10:             LPAR           '('
4,10-4,11:            RPAR           ')'
4,11-4,12:            NEWLINE        '\n'
5,0-5,0:              ENDMARKER      ''
```

32.8。 tabnanny- 检测模糊缩进

源代码： Lib / tabnanny.py

目前这个模块的目的是作为一个脚本被调用。但是可以将它导入IDE并使用`check()` 下面描述的功能。

> **注意：** 此模块提供的API可能会在未来的版本中发生变化; 此类更改可能不会向后兼容。

tabnanny.**check**(*file_or_dir*)

如果*file_or_dir*是目录而不是符号链接，则递归地下降由*file_or_dir*命名的目录树，检查所有`.py` 文件。如果*file_or_dir*是一个普通的Python源文件，则会检查与空白相关的问题。使用该`print()`功能将诊断消息写入标准输出。

tabnanny.**verbose**

指示是否打印详细消息的标志。`-v`如果作为脚本被调用，这会增加该 选项。

tabnanny.**filename_only**

指示是否仅打印包含与空白相关的问题的文件的文件名的标志。`-q`如果调用脚本，则该选项设置为true 。

*异常*tabnanny.**NannyNag**

`process_tokens()`如果检测到模糊缩进，则引发。捕获并处理`check()`。

tabnanny.**process_tokens**(*令牌*)

该函数用于`check()`处理`tokenize`模块生成的令牌 。

也可以看看:

模 `tokenize`

用于Python源代码的词法扫描器。

32.9。 pyclbr- Python类浏览器支持

源代码： Lib / pyclbr.py

该pyclbr模块可用于确定有关模块中定义的类，方法和顶级函数的一些有限信息。所提供的信息足以实现传统的三窗格类浏览器。信息是从源代码中提取的，而不是通过导入模块，所以这个模块可以安全地用于不可信的代码。这个限制使得这个模块不能用于在Python中未实现的模块，包括所有标准和可选的扩展模块。

pyclbr. **readmodule** (*module* , *path = None*)

读取一个模块并返回一个字典映射类名到类描述符对象。参数*模块*应该是一个字符串模块的名称; 它可能是一个包中模块的名称。该 *路径*参数应该是一个序列，并且用于增加的值 sys.path ，这是用来定位模块的源代码。

pyclbr. **readmodule_ex** (*module* , *path = None*)

就像readmodule() ，但返回的字典除了将类名称映射到类描述符对象之外，还将顶级函数名称映射到函数描述符对象。此外，如果正在读取的模块是包，则’__path__’返回的字典中的键具有包含包搜索路径的列表的值。

32.9.1。类对象

Class用作字典中值的对象返回 readmodule()并readmodule_ex()提供以下数据属性：

Class. **module**

定义由类描述符描述的类的模块的名称。

Class. **name**

班级的名字。

Class. **super**

Class描述所描述类的直接基类的对象列表。被命名为超类但不被发现的类readmodule()被列为具有类名而不是作为Class 对象的字符串。

Class. **methods**

字典映射方法名称到行号。

Class. **file**

包含class定义类的语句的文件的名称。

Class. **lineno**

class指定文件中的语句 的行号file。

32.9.2。函数对象

Function用作返回字典中的值的对象 readmodule_ex() 提供以下属性：

Function.**module**
: 定义由功能描述符描述的功能的模块的名称。

Function.**name**
: 函数的名称。

Function.**file**
: 包含def定义函数的语句的文件的名称。

Function.**lineno**
: def指定文件中的语句 的行号file。

32.10。 py_compile- 编译Python源文件

源代码： Lib / py_compile.py

该py_compile模块提供了从源文件生成字节码文件的功能，以及将模块源文件作为脚本调用时使用的另一个功能。

虽然不是经常需要，但在安装共享使用模块时，此功能可能很有用，特别是如果某些用户可能没有权限在包含源代码的目录中写入字节代码缓存文件。

*异常*py_compile.PyCompileError

当尝试编译文件时发生错误时引发异常。

py_compile.compile（*file*，*cfile = None*，*dfile = None*，*doraise = False*，*optimize = -1* ）

将源文件编译为字节码并写出字节码高速缓存文件。源代码是从名为*file*的文件中加载的。字节码被写入*cfile*，默认为**PEP 3147** /**PEP 488**路径，结束于.pyc。例如，如果*文件*是/foo/bar/baz.py *cfile*，将默认/foo/bar/__pycache__/baz.cpython-32.pyc为Python 3.2。如果指定了*dfile*，则在代替*文件*时将其用作错误消息中的源文件的名称。如果*doraise*为true，PyCompileError则在编译*文件*时遇到错误时会引发a。如果*doraise*为false（缺省值），则会写入错误字符串sys.stderr，但不会引发异常。该函数返回字节编译文件的路径，即使用任何*cfile*值。

如果*cfile*成为（明确指定或计算的）的路径是符号链接或非常规文件，FileExistsError将会引发。这是一个警告，如果允许将字节编译文件写入这些路径，导入会将这些路径转换为常规文件。这是使用文件重命名来导入最终字节编译文件以防止并发文件写入问题的导入的副作用。

*优化*控制优化级别并传递给内置compile()函数。默认-1选择当前解释器的优化级别。

*在版本3.2中更改：*将*cfile*的默认值更改为**符合PEP 3147**标准。先前的默认值是*文件*+'c'（'o'如果已启用优化）。还添加了*优化*参数。

*在版本3.4中进行了更改：*更改了importlib用于字节码高速缓存文件写入的代码。这意味着文件创建/写入语义现在匹配什么importlib，例如权限，写入和移动语义等。还添加了FileExistsError当*cfile*是符号链接或非常规文件时引发的警告。

py_compile.main（*args = None* ）

编译几个源文件。在*args*中命名的文件（或者在命令行上，如果*args*是None）被编译，并且结果字节码以正常方式被缓存。该功能不搜索目录结构来定位源文件; 它只编译明确指定的文件。如果'-'是参数中唯一的参数，则文件列表取自标准输入。

*在版本3.2中更改：*添加了对'-'。

当该模块作为脚本运行时，该模块main()用于编译在命令行上命名的所有文件。如果其中一个文件无法编译，则退出状态为非零。

也可以看看:

模 `compileall`
工具来编译目录树中的所有Python源文件。

32.11。compileall- 字节编译Python库

源代码： Lib / compileall.py

该模块提供了一些实用功能来支持安装Python库。这些函数在目录树中编译Python源文件。此模块可用于在安装库时创建缓存的字节码文件，这使得即使对库目录没有写入权限的用户也可使用该模块。

32.11.1。命令行使用

该模块可以作为脚本（使用**python -m compileall**）来编译Python源代码。

`directory ...`
`file ...`
位置参数是要编译的文件或包含源文件的目录，递归遍历。如果没有给出参数，就像命令行一样。`-l <directories from sys.path>`

`-l`
不要递归到子目录中，只编译直接包含在指定或隐含目录中的源代码文件。

`-f`
即使时间戳是最新的，也强制重建。

`-q`
不要打印编译的文件列表。如果传递一次，错误消息仍将被打印。如果传递两次（`-qq`），则所有输出都被抑制。

`-d destdir`
预编译到正在编译的每个文件的路径的目录。这将出现在编译时间回溯中，并且也编译到字节码文件中，在字节码文件执行时源文件不存在的情况下，它将用于回溯和其他消息中。

`-x regex`
正则表达式用于搜索考虑编译的每个文件的完整路径，如果正则表达式产生匹配，则跳过文件。

`-i list`
读取文件`list`并将其包含的每一行添加到要编译的文件和目录列表中。如果`list`是`-`，则从中读取行`stdin`。

`-b`
将字节码文件写入其遗留位置和名称，这可能会覆盖由其他版本的Python创建的字节码文件。默认是将文件写入其中**PEP 3147**位置和名称，允许来自多个Python版本的字节码文件共存。

-r

控制子目录的最大递归级别。如果这是给出的，那么-l选项将不被考虑。 **python -m compileall <directory> -r 0**相当于 **python -m compileall <directory> -l**。

-j N

使用*N*个工人编译给定目录内的文件。如果0被使用，那么结果os.cpu_count() 将被使用。

*在3.2版本的改变：*增加了-i ，-b和-h选项。

*在3.5版本中改变：*增加了 -j ，-r和-qq选项。 -q选项已更改为多级值。 -b将永远产生一个字节码文件，结尾.pyc ，永远不会.pyo。

没有命令行选项来控制compile()函数使用的优化级别 ，因为Python解释器本身已经提供了选项：**python -O -m compileall**。

32.11.2。公共职能

compileall.compile_dir（*dir*，*maxlevels = 10*，*ddir = None*，*force = False*，*rx = None*，*quiet = 0*，*legacy = False*，*optimize = -1*，*workers = 1* ）

递归地下降由*dir*命名的目录树，.py 沿途编译所有文件。如果所有文件都成功编译，则返回一个真值，否则返回一个假值。

所述*maxlevels*参数用于限制递归的深度; 它默认为10。

如果给出了*ddir*，则它被*预*编译为每个被编译以用于编译时间回溯的文件的路径，并且还被编译到字节码文件中，在该文件中将用于回溯和其他消息，文件在执行字节码文件时不存在。

如果*强制*为真，即使时间戳是最新的，模块也会重新编译。

如果给出*rx*，则会在考虑编译的每个文件的完整路径上调用其搜索方法，如果它返回true值，则会跳过该文件。

如果*安静*是False或0（缺省值），文件名和其他信息被打印到标准输出。设置为1 ，仅打印错误。设置为2 ，所有输出被抑制。

如果*legacy*为true，则字节码文件将写入其传统位置和名称，这可能会覆盖由其他版本的Python创建的字节码文件。默认是将文件写入其中**PEP 3147**位置和名称，允许来自多个Python版本的字节码文件共存。

*优化*指定编译器的优化级别。它被传递给内置compile()函数。

参数*workers*指定并行编译文件的人数。默认情况是不使用多个工作人员。如果平台不能使用多个工人和*工人*参数，则顺序编译将用作后备。如果*工人*低于0 ，ValueError则会提高a。

*在版本3.2中进行了更改：*添加了*遗留*和*优化*参数。

*版本3.5中已更改：*添加了*工人*参数。

*在版本3.5中更改：安静*参数已更改为多级值。

*在3.5版本中改变：*在*传统的*参数只写出来.pyc的文件，没有.pyo文件，无论什么价值*优化*的。

*在版本3.6中更改：*接受类似路径的对象。

compileall.**compile_file**(*fullname*, *ddir = None*, *force = False*, *rx = None*, *quiet = 0*, *legacy = False*, *optimize = -1*)

用路径*全名*编译文件。如果文件编译成功，则返回一个真值，否则返回一个假值。

如果给出了*ddir*，则它将被编译为正在编译的文件的路径作为编译时间回溯的前缀，并且还会被编译到字节码文件中，在该文件中将用于回溯和其他消息，文件在执行字节码文件时不存在。

如果给出了*rx*，它的搜索方法会将完整路径名称传递给正在编译的文件，如果它返回一个真值，则该文件不会被编译并True返回。

如果*安静*是False或0（缺省值），文件名和其他信息被打印到标准输出。设置为1，仅打印错误。设置为2，所有输出被抑制。

如果*legacy*为true，则字节码文件将写入其传统位置和名称，这可能会覆盖由其他版本的Python创建的字节码文件。默认是将文件写入其中**PEP 3147**位置和名称，允许来自多个Python版本的字节码文件共存。

*优化*指定编译器的优化级别。它被传递给内置compile()函数。

3.2版本中的新功能

*在版本3.5中更改：安静*参数已更改为多级值。

*在3.5版本中改变：*在*传统的*参数只写出来.pyc的文件，没有.pyo文件，无论什么价值*优化*的。

compileall.**compile_path**(*skip_curdir = True*, *maxlevels = 0*, *force = False*, *quiet = 0*, *legacy = False*, *optimize = -1*)

字节编译所有.py找到的文件sys.path。如果所有文件都成功编译，则返回一个真值，否则返回一个假值。

如果*skip_curdir*为true（默认值），则当前目录不包含在搜索中。所有其他参数都传递给该compile_dir() 函数。请注意，与其他编译函数不同，maxlevels 默认为0。

*在版本3.2中进行了更改：*添加了*遗留*和*优化*参数。

*在版本3.5中更改：安静*参数已更改为多级值。

*在3.5版本中改变：*在*传统的*参数只写出来.pyc的文件，没有.pyo文件，无论什么价值*优化*的。

强制重新编译 子目录及其所有子目录中的所有.py文件Lib/：

```
import compileall

compileall.compile_dir('Lib/', force=True)

# Perform same compilation, excluding files in .svn directories.
import re
compileall.compile_dir('Lib/', rx=re.compile(r'[/\\][.]svn'), force=True)

# pathlib.Path objects can also be used.
import pathlib
compileall.compile_dir(pathlib.Path('Lib/'), force=True)
```

也可以看看:

模 py_compile
    字节编译单个源文件。

32.12。dis- 用于Python字节码的反汇编程序

源代码： Lib / dis.py

该dis模块通过反汇编来支持CPython 字节码的分析。该模块用作输入的CPython字节码在文件中定义Include/opcode.h并由编译器和解释器使用。

CPython实现细节： Bytecode是CPython解释器的实现细节。没有保证不会在不同版本的Python之间添加，删除或更改字节码。不应该认为这个模块的使用适用于Python VM或Python发行版。

*在版本3.6中更改：*每条指令使用2个字节。以前按指令变化的字节数。

示例：给定函数myfunc()：

```
def myfunc(alist):
    return len(alist)
```

以下命令可用于显示反汇编 myfunc()：

```
>>> dis.dis(myfunc)
  2           0 LOAD_GLOBAL              0 (len)
              2 LOAD_FAST                0 (alist)
              4 CALL_FUNCTION            1
              6 RETURN_VALUE
```

（“2”是行号）。

32.12.1。字节码分析

3.4版新增功能

字节码分析API允许将多段Python代码封装在 Bytecode对象中，以便轻松访问已编译代码的详细信息。

class dis.Bytecode（*x*，*，*first_line = None*，*current_offset = None*）

分析与函数，生成器，方法，源代码字符串或代码对象（由返回的compile()）对应的字节码。

这是围绕下面列出的许多函数的便捷包装，最值得注意的是get_instructions()，迭代Bytecode 实例会将字节码操作作为Instruction实例进行迭代。

如果*first_line*不是None，则表示应在反汇编代码中为第一个源代码行报告的行号。否则，源代码行信息（如果有）直接从反汇编的代码对象中获取。

如果*current_offset*不是None，则它指代反汇编代码中的指令偏移量。设置这意味着dis()将显示一个针对指定操作码的“当前指令”标记。

classmethod from_traceback (*tb*)

Bytecode从给定的回溯中构造一个实例，将*current_offset*设置 为负责异常的指令。

codeobj

编译后的代码对象。

first_line

代码对象的第一个源代码行（如果可用）

dis ( )

返回字节码操作的格式化视图（与打印的相同 dis.dis() ，但作为多行字符串返回）。

info ( )

返回格式化的多行字符串，其中包含有关代码对象的详细信息，例如code_info()。

例：

```
>>> bytecode = dis.Bytecode(myfunc)
>>> for instr in bytecode:
...     print(instr.opname)
...
LOAD_GLOBAL
LOAD_FAST
CALL_FUNCTION
RETURN_VALUE
```

32.12.2。分析功能

该dis模块还定义了以下分析功能，可直接将输入转换为所需的输出。如果只执行一个操作，它们可能很有用，因此中间分析对象不是有用的：

dis. code_info (*x*)

返回格式化的多行字符串，其中包含提供的函数，生成器，方法，源代码字符串或代码对象的详细代码对象信息。

请注意，代码信息字符串的确切内容具有很高的实现依赖性，并且可能会在Python VM或Python发行版中随意更改。

3.2版本中的新功能

dis. show_code (*x* , * , *file = None*)

将提供的函数，方法，源代码字符串或代码对象的详细代码对象信息打印到*文件*（或者sys.stdout如果 未指定*文件*）。

这是一个方便的简写，用于在解释器提示符下进行交互式探索。 print(code_info(x), file=file)

3.2版本中的新功能

*在版本3.4中进行了更改：*添加了*文件*参数。

dis.**dis**(*x = None*, ***, *file = None*)

拆卸*x*对象。 *x*可以表示模块，类，方法，函数，生成器，代码对象，源代码字符串或原始字节代码的字节序列。对于一个模块，它会分解所有功能。对于一个类，它会反汇编所有方法（包括类和静态方法）。对于代码对象或原始字节码序列，它会为每个字节码指令打印一行。首先将字符串编译为`compile()`在分解之前使用内置函数编写对象。如果未提供对象，则此功能将反汇编上一次回溯。

如果提供反汇编，则将其作为文本写入所提供的*文件*参数中，`sys.stdout`否则将反汇编写入*文件*。

*在版本3.4中进行了更改：*添加了*文件*参数。

dis.**distb**(*tb = None*, ***, *file = None*)

如果没有传递，则使用最后的回溯来反汇编回溯的堆栈顶层函数。指示引发异常的指令。

如果提供反汇编，则将其作为文本写入所提供的*文件*参数中，`sys.stdout`否则将反汇编写入*文件*。

*在版本3.4中进行了更改：*添加了*文件*参数。

dis.**disassemble**(*code*, *lasti = -1*, ***, *file = None*)

dis.**disco**(*code*, *lasti = -1*, ***, *file = None*)

反汇编一个代码对象，指出如果提供了*lasti*的最后一条指令。输出分为以下几列：

1. 行号，用于每行的第一条指令
2. 当前指令，表示为`-->`，
3. 带标签的说明`>>`，
4. 指令的地址，
5. 操作代码名称，
6. 操作参数和
7. 括号中的参数解释。

参数解释可识别本地和全局变量名称，常量值，分支目标和比较运算符。

如果提供反汇编，则将其作为文本写入所提供的*文件*参数中，`sys.stdout`否则将反汇编写入*文件*。

*在版本3.4中进行了更改：*添加了*文件*参数。

dis.**get_instructions**(*x*, ***, *first_line = None*)

返回所提供的函数，方法，源代码字符串或代码对象中的指令的迭代器。

迭代器生成一系列`Instruction`命名元组，给出所提供代码中每个操作的详细信息。

如果*first_line*不是`None`，则表示应在反汇编代码中为第一个源代码行报告的行号。否则，源代码行信息（如果有）直接从反汇编的代码对象中获取。

3.4版新增功能

dis.findlinestarts(*代码*)

这种发生器函数使用co_firstlineno和co_lnotab 代码对象的属性*代码*到发现其是在源代码的行开始的偏移量。它们以成对的方式生成。查看对象/ lnotab_notes.txt的格式以及如何对其进行解码。(offset, lineno) co_lnotab

*在版本3.6中更改：*行号可能会减少。之前，他们一直在增加。

dis.findlabels(*代码*)

检测作为跳转目标的代码对象*代码*中的所有偏移量，并返回这些偏移量的列表。

dis.stack_effect(*opcode*[, *oparg*])

使用参数*oparg*计算*opcode*的堆栈效应。

3.4版新增功能

32.12.3。Python的字节码指令

该get_instructions()函数和Bytecode类提供的字节码指令的详细Instruction情况：

*类*dis.Instruction

字节码操作的细节

opcode

用于操作的数字代码，对应于下面列出的操作码值和Opcode集合中的字节码值。

opname

用于操作的人类可读名称

arg

操作的数字参数（如果有的话），否则 None

argval

解析参数值（如果知道），否则与arg相同

argrepr

操作参数的可读描述

offset

在字节码序列内启动操作索引

starts_line

由此操作码开始的行（如果有的话），否则 None

is_jump_target

True 如果其他代码跳到这里，否则 False

3.4版新增功能

Python编译器当前生成以下字节码指令。

一般说明

NOP
不做任何代码。用作字节码优化器的占位符。

POP_TOP
删除顶部堆栈（TOS）项目。

ROT_TWO
交换两个最上面的堆栈项。

ROT_THREE
将第二个和第三个堆叠物品向上提升一个位置，从上至下移动到第三个位置。

DUP_TOP
复制堆栈顶部的引用。

3.2版本中的新功能

DUP_TOP_TWO
复制堆栈顶部的两个引用，使它们保持相同的顺序。

3.2版本中的新功能

一元操作

一元操作占据堆栈的顶部，应用操作，并将结果推回堆栈。

UNARY_POSITIVE
实现。`TOS = +TOS`

UNARY_NEGATIVE
实现。`TOS = -TOS`

UNARY_NOT
实现。`TOS = not TOS`

UNARY_INVERT
实现。`TOS = ~TOS`

GET_ITER
实现。`TOS = iter(TOS)`

GET_YIELD_FROM_ITER
如果TOS是生成器迭代器或协程对象，它将保持原样。否则，实现。`TOS = iter(TOS)`

3.5版本中的新功能。

二进制操作

二进制操作会从堆栈中删除堆栈顶部（TOS）和第二个最顶端堆栈项（TOS1）。他们执行操作，并将结果放回堆栈。

BINARY_POWER
    实现。TOS = TOS1 ** TOS

BINARY_MULTIPLY
    实现。TOS = TOS1 * TOS

BINARY_MATRIX_MULTIPLY
    实现。TOS = TOS1 @ TOS

    3.5版本中的新功能。

BINARY_FLOOR_DIVIDE
    实现。TOS = TOS1 // TOS

BINARY_TRUE_DIVIDE
    实现。TOS = TOS1 / TOS

BINARY_MODULO
    实现。TOS = TOS1 % TOS

BINARY_ADD
    实现。TOS = TOS1 + TOS

BINARY_SUBTRACT
    实现。TOS = TOS1 - TOS

BINARY_SUBSCR
    实现。TOS = TOS1[TOS]

BINARY_LSHIFT
    实现。TOS = TOS1 << TOS

BINARY_RSHIFT
    实现。TOS = TOS1 >> TOS

BINARY_AND
    实现。TOS = TOS1 & TOS

BINARY_XOR
    实现。TOS = TOS1 ^ TOS

BINARY_OR
    实现。TOS = TOS1 | TOS

就地操作

就地操作就像二元操作，因为它们删除TOS和TOS1，并将结果重新放回堆栈，但操作在TOS1支持它时在原地完成，并且生成的TOS可能（但没有是）原来的TOS1。

INPLACE_POWER
实现就地。TOS = TOS1 ** TOS

INPLACE_MULTIPLY
实现就地。TOS = TOS1 * TOS

INPLACE_MATRIX_MULTIPLY
实现就地。TOS = TOS1 @ TOS

3.5版本中的新功能。

INPLACE_FLOOR_DIVIDE
实现就地。TOS = TOS1 // TOS

INPLACE_TRUE_DIVIDE
实现就地。TOS = TOS1 / TOS

INPLACE_MODULO
实现就地。TOS = TOS1 % TOS

INPLACE_ADD
实现就地。TOS = TOS1 + TOS

INPLACE_SUBTRACT
实现就地。TOS = TOS1 - TOS

INPLACE_LSHIFT
实现就地。TOS = TOS1 << TOS

INPLACE_RSHIFT
实现就地。TOS = TOS1 >> TOS

INPLACE_AND
实现就地。TOS = TOS1 & TOS

INPLACE_XOR
实现就地。TOS = TOS1 ^ TOS

INPLACE_OR
实现就地。TOS = TOS1 | TOS

STORE_SUBSCR
实现。TOS1[TOS] = TOS2

DELETE_SUBSCR

实现。del TOS1[TOS]

协程操作码

GET_AWAITABLE

实现，其中 返回if 是具有CO_ITERABLE_COROUTINE标志的协程对象或生成器对象，或者解析 。 TOS = get_awaitable(TOS) get_awaitable(o) ooo.__await__

3.5版本中的新功能。

GET_AITER

实现。查看 有关详情TOS = get_awaitable(TOS.__aiter__()) GET_AWAITABLE get_awaitable

3.5版本中的新功能。

GET_ANEXT

实现PUSH(get_awaitable(TOS.__anext__()))。查看GET_AWAITABLE 有关详情get_awaitable

3.5版本中的新功能。

BEFORE_ASYNC_WITH

解决__aenter__和__aexit__从堆栈顶部的对象。推动__aexit__和结果__aenter__()的堆栈。

3.5版本中的新功能。

SETUP_ASYNC_WITH

创建一个新的框架对象。

3.5版本中的新功能。

其他操作码

PRINT_EXPR

为交互模式实现表达式语句。服务条款从堆栈中删除并打印。在非交互模式下，表达式语句以POP_TOP。结束。

BREAK_LOOP

由于break声明而终止循环。

CONTINUE_LOOP (*目标*)

由于continue陈述而继续循环。 *目标*是要跳转到的地址（这应该是一条FOR_ITER指令）。

SET_ADD (*i*)

电话。用于实现集合理解。 set.add(TOS1[-i], TOS)

LIST_APPEND (*i*)

电话。用于实现列表推导。 list.append(TOS[-i], TOS)

MAP_ADD (*i*)

电话。用于实现词典理解。 `dict.setitem(TOS1[-i], TOS, TOS1)`

版本3.1中的新功能。

对于所有的SET_ADD ，LIST_APPEND和MAP_ADD 指令，而增值或键/值对被弹出时，容器对象保持在栈上，以便它可用于循环的进一步迭代。

RETURN_VALUE

将TOS返回给函数的调用者。

YIELD_VALUE

弹出TOS并从发生器中产生它。

YIELD_FROM

弹出TOS并将其委托给发电机作为子控制器。

3.3版本的新功能

SETUP_ANNOTATIONS

检查是否`__annotations__`定义`locals()`，如果不是，则将其设置为空`dict`。此操作码仅在静态包含变量注释的类或模块主体时才会发出。

3.6版本中的新功能。

IMPORT_STAR

将所有不是'_'直接从模块TOS 开始的符号加载到本地名称空间。加载所有名称后，模块弹出。这个操作码实现。 `from module import *`

POP_BLOCK

从块堆栈中删除一个块。每帧有一堆块，表示嵌套循环，try语句等等。

POP_EXCEPT

从块堆栈中删除一个块。弹出的块必须是异常处理程序块，如输入除处理程序时隐式创建的那样。除了从帧堆栈弹出无关值外，最后三个弹出的值用于恢复异常状态。

END_FINALLY

终止一个finally条款。解释器回顾是否必须重新提出异常，或函数是否返回，并继续执行外部下一个块。

LOAD_BUILD_CLASS

推`builtins.__build_class__()`入堆栈。它后来被称为CALL_FUNCTION构建一个类。

SETUP_WITH (*delta*)

该操作码在块启动之前执行多个操作。首先，它`__exit__()`从上下文管理器加载并将其压入堆栈供以后使用WITH_CLEANUP。然后， `__enter__()`被调用，并且指向*delta*的finally块 被推入。最后，调用enter方法的结果被压入堆栈。下一个操作码将忽略它（ POP_TOP ），或将其存储在（a）变量（ ，）STORE_FAST ，STORE_NAME （或 UNPACK_SEQUENCE ）中。

3.2版本中的新功能

WITH_CLEANUP_START

with语句块退出时清理堆栈。TOS是上下文管理器的__exit__()绑定方法。在TOS下面有1-3个值，表示如何/为什么输入finally子句：

- SECOND = None
- （SECOND，THIRD）=（WHY_{RETURN,CONTINUE}），retval
- SECOND = WHY_*; 它下面没有retval
- （SECOND，THIRD，FOURTH）= exc_info（）

在最后一种情况下，被调用，否则 。将SECOND和结果调用到堆栈。TOS(SECOND, THIRD, FOURTH) TOS(None, None, None)

WITH_CLEANUP_FINISH

从堆栈中弹出异常类型和'exit'函数调用的结果。

如果堆栈表示一个异常，*并且*函数调用返回一个'true'值，那么这个信息会被“zapped”并被一个单独的代替 WHY_SILENCED以防止END_FINALLY重新引发异常。（但非本地gotos仍然会恢复。）

以下所有操作码都使用它们的参数。

STORE_NAME（*namei*）

实现。*namei*是代码对象属性 中*名称*的索引。编译器尝试使用 或者如果可能的话。name = TOS.co_names STORE_FAST STORE_GLOBAL

DELETE_NAME（*namei*）

实现，其中*namei*是 代码对象的索引属性。del name co_names

UNPACK_SEQUENCE（*count*）

解压缩到TOS *计*个人的价值观，这是放到堆栈从右到左。

UNPACK_EX（*计数*）

实现具有星标目标的赋值：将TOS中的迭代包解包为单个值，其中值的总数可以小于迭代中的项数：其中一个新值将是所有剩余项目的列表。

*计数*的低字节是列表值之前的值的数量，高字节是对它之后的值的数量进行*计数*。结果值从右到左放在堆栈中。

STORE_ATTR（*namei*）

实现，其中*namei*是名称的索引 。TOS.name = TOS1 co_names

DELETE_ATTR（*namei*）

实现，使用*namei*作为索引。del TOS.name co_names

STORE_GLOBAL（*namei*）

作为STORE_NAME，但将名称存储为全局名称。

DELETE_GLOBAL (*namei*)

作为DELETE_NAME，但删除全局名称。

LOAD_CONST (*consti*)

推co_consts[consti]入堆栈。

LOAD_NAME (*namei*)

将关联的值co_names[namei]推入堆栈。

BUILD_TUPLE (*count*)

创建一个从堆栈中消耗*计数*项的元组，并将生成的元组推入堆栈。

BUILD_LIST (*count*)

作为BUILD_TUPLE，但创建一个列表。

BUILD_SET (*count*)

作为BUILD_TUPLE，但创建一个集合。

BUILD_MAP (*count*)

将新的字典对象推入堆栈。弹出项目，以便字典包含*计数*条目： 。2 * count{..., TOS3: TOS2, TOS1: TOS}

*在版本3.5中进行了更改：*字典是从堆栈项创建的，而不是创建一个预先确定大小以容纳*计数*项的空字典。

BUILD_CONST_KEY_MAP (*count*)

BUILD_MAP专用于常量键的版本。 *计* 数值从栈中消耗。堆栈顶部的元素包含一个键元组。

3.6版本中的新功能。

BUILD_STRING (*count*)

连接来自堆栈的*计数*字符串并将结果字符串压入堆栈。

3.6版本中的新功能。

BUILD_TUPLE_UNPACK (*count*)

Pops 从堆栈中*计算*可迭代数据，并将它们加入到一个元组中，并推送结果。在元组显示中实现迭代解包。(*x, *y, *z)

3.5版本中的新功能。

BUILD_TUPLE_UNPACK_WITH_CALL (*count*)

这与调用语法相似BUILD_TUPLE_UNPACK，但用于调用语法。位置上的堆栈项目 应该是相应的可调用对象。f(*x, *y, *z)count + 1f

3.6版本中的新功能。

BUILD_LIST_UNPACK (*count*)

这BUILD_TUPLE_UNPACK与之类似，但推送一个列表而不是元组。在列表显示中实现迭代解包。`[*x, *y, *z]`

3.5版本中的新功能。

BUILD_SET_UNPACK (*count*)

这BUILD_TUPLE_UNPACK与之类似，但是推送了一个集合而不是元组。在集合显示中实现迭代解包。`{*x, *y, *z}`

3.5版本中的新功能。

BUILD_MAP_UNPACK (*count*)

Pops 从堆栈中*统计*映射，将它们合并到单个字典中，并推送结果。在字典显示中实现字典解包。`{**x, **y, **z}`

3.5版本中的新功能。

BUILD_MAP_UNPACK_WITH_CALL (*count*)

这与调用语法相似BUILD_MAP_UNPACK ，但用于调用语法。位置上的堆栈项目应该是相应的可调用对象。`f(**x, **y, **z)` `count + 2`

3.5版本中的新功能。

*版本3.6中更改：*可调用的位置是通过将opcode参数加2而不是在参数的第二个字节中对其进行编码来确定的。

LOAD_ATTR (*namei*)

用TOS替换TOS 。`getattr(TOS, co_names[namei])`

COMPARE_OP (*opname*)

执行布尔操作。操作名称可以在中找到 `cmp_op[opname]`。

IMPORT_NAME (*namei*)

导入模块 `co_names[namei]` 。 TOS 和 TOS1 被弹出并提供 *fromlist* 和 *level* *的*参数 `__import__()`。模块对象被压入堆栈。当前名称空间不受影响：对于正确的导入语句，后续STORE_FAST指令修改名称空间。

IMPORT_FROM (*namei*)

`co_names[namei]` 从TOS中找到的模块加载属性。产生的对象被压入堆栈，随后被STORE_FAST指令存储 。

JUMP_FORWARD (*delta*)

增量字节码计数器*增量*。

POP_JUMP_IF_TRUE (*目标*)

如果TOS为真，则将字节码计数器设置为*目标*。服务条款被弹出。

版本3.1中的新功能。

POP_JUMP_IF_FALSE（*目标*）
如果TOS为false，则将字节码计数器设置为*目标*。服务条款被弹出。

版本3.1中的新功能。

JUMP_IF_TRUE_OR_POP（*目标*）
如果TOS为真，则将字节码计数器设置为*目标*并将TOS保留在堆栈上。否则（TOS是错误的），TOS弹出。

版本3.1中的新功能。

JUMP_IF_FALSE_OR_POP（*目标*）
如果TOS为false，则将字节码计数器设置为*目标*并将TOS保留在堆栈上。否则（TOS为真），TOS弹出。

版本3.1中的新功能。

JUMP_ABSOLUTE（*目标*）
将字节码计数器设置为*目标*。

FOR_ITER（*delta*）
TOS是一个迭代器。调用它的`__next__()`方法。如果这产生一个新值，把它推到堆栈上（将迭代器留在它下面）。如果迭代器指示它已用尽TOS被弹出，并且字节码计数器增量为*增量*。

LOAD_GLOBAL（*namei*）
将指定的全局加载`co_names[namei]`到堆栈上。

SETUP_LOOP（*delta*）
将块的一个循环推入块堆栈。块从当前指令跨越*delta*字节的大小。

SETUP_EXCEPT（*delta*）
将try-except子句中的try块推入块堆栈。*delta* 指向第一个除了块。

SETUP_FINALLY（*delta*）
将try-except子句中的try块推入块堆栈。*delta* 指向finally块。

LOAD_FAST（*var_num*）
将对本地的引用`co_varnames[var_num]`推入堆栈。

STORE_FAST（*var_num*）
将TOS存储在本地`co_varnames[var_num]`。

DELETE_FAST（*var_num*）
删除本地`co_varnames[var_num]`。

STORE_ANNOTATION（*namei*）
将TOS存储为。`locals()['__annotations__'][co_names[namei]] = TOS`

3.6版本中的新功能。

LOAD_CLOSURE (*i*)
将参照推送到单元格的槽位*i*中的单元格，并释放可变的存储空间。变量的名称是 `co_cellvars[i]` 如果*我*小于*co_cellvars*的长度。否则是。 `co_freevars[i - len(co_cellvars)]`

LOAD_DEREF (*i*)
加载单元格的槽位*i*中的单元格并释放可变的存储空间。将参考推送到单元包含在堆栈上的对象。

LOAD_CLASSDEREF (*i*)
就像LOAD_DEREF在查询单元格之前首先检查本地字典。这用于在类体中加载自由变量。

3.4版新增功能

STORE_DEREF (*i*)
将TOS存储到单元格的槽位*i*中的单元格中并免费存储变量。

DELETE_DEREF (*i*)
清空单元格的槽位*i*中的单元格并释放可变的存储空间。由所使用的del语句。

3.2版本中的新功能

RAISE_VARARGS (*argc*)
引发一个例外。*argc*表示raise语句的参数数量，范围从0到3.处理程序将查找回溯为TOS2，参数为TOS1，异常为TOS。

CALL_FUNCTION (*argc*)
调用一个函数。 *argc*表示位置参数的数量。位置参数在堆栈上，最右边的参数在顶部。在参数下面，要调用的函数对象位于堆栈上。弹出所有函数参数，并将函数本身从堆栈中弹出，并推送返回值。

*在版本3.6中更改：*此操作码仅用于具有位置参数的调用。

CALL_FUNCTION_KW (*argc*)
调用一个函数。 *argc*表示参数的数量（位置和关键字）。堆栈顶部的元素包含关键字参数名称的元组。在元组下面，关键字参数在堆栈中，按照与元组相对应的顺序。在关键字参数下面，位置参数在堆栈上，最右边的参数在顶部。在参数下面，要调用的函数对象位于堆栈上。弹出所有函数参数，并将函数本身从堆栈中弹出，并推送返回值。

*版本3.6中更改：*关键字参数打包在一个元组而不是字典中， *argc*表示参数的总数

CALL_FUNCTION_EX (*标志*)
调用一个函数。 *标志*的最低位指示var-keyword参数是否放置在堆栈的顶部。在var-keyword参数下面，var-positional句参数在堆栈上。在参数下面，放置要调用的函数对象。弹出所有函数参数，并将函数本身从堆栈中弹出，并推送返回值。请注意，此操作码最多

可从堆栈中弹出三个项目。Var和位置和var关键字参数由BUILD_TUPLE_UNPACK_WITH_CALL和打包BUILD_MAP_UNPACK_WITH_CALL。

3.6版本中的新功能。

MAKE_FUNCTION（*argc*）

推入堆栈中的新函数对象。从底部到顶部，如果参数携带指定的标志值，则消耗的堆栈必须由值组成

- 0x01 按照位置顺序的默认参数对象的元组
- 0x02 一个关键字参数的默认值字典
- 0x04 一个注释字典
- 0x08 包含用于自由变量的单元的元组，从而形成闭包
- 与功能相关的代码（在TOS1）
- 该函数的合格名称（在TOS）

BUILD_SLICE（*argc*）

推入堆栈中的切片对象。 *argc*必须是2或3.如果是2， 则推送; 如果是3，则推。有关更多信息，请参阅内置功能。 slice(TOS1, TOS) slice(TOS2, TOS1, TOS) slice()

EXTENDED_ARG（*ext*）

前缀任何操作码的参数太大，无法放入默认的两个字节。 *ext*包含两个额外的字节，与后面的操作码参数一起构成一个四字节参数，*ext*是两个最重要的字节。

FORMAT_VALUE（*标志*）

用于实现格式化的文字字符串（f-字符串）。从堆栈中*弹出*一个可选的*fmt_spec*，然后是所需的*值*。 *标志*解释如下：

- (flags & 0x03) == 0x00：*值*按原样格式化。
- (flags & 0x03) == 0x01：呼吁str()对*价值*格式化之前。
- (flags & 0x03) == 0x02：呼吁repr()对*价值*格式化之前。
- (flags & 0x03) == 0x03：呼吁ascii()对*价值*格式化之前。
- (flags & 0x04) == 0x04：从堆栈中弹出*fmt_spec*并使用它，否则使用空的*fmt_spec*。

格式化使用PyObject_Format()。结果被压入堆栈。

3.6版本中的新功能。

HAVE_ARGUMENT

这不是一个真正的操作码。它确定了不使用参数的操作码之间的分界线以及那些（和分别）做的操作码之间的分界线。< HAVE_ARGUMENT >= HAVE_ARGUMENT

*在版本3.6中更改：*现在每条指令都有一个参数，但操作码 忽略它。之前，只有操作码有争论。< HAVE_ARGUMENT >= HAVE_ARGUMENT

32.12.4。操作码集合

这些集合用于字节码指令的自动内省：

dis.opname
操作名称的顺序，可使用字节码进行索引。

dis.opmap
字典映射操作名称到字节码。

dis.cmp_op
所有比较操作名称的顺序。

dis.hasconst
具有常数参数的字节码序列。

dis.hasfree
访问自由变量的字节码序列（注意，“自由”在这种情况下指的是在目前的范围由内部范围或在从这个范围内引用外部范围的名字引用的名称，它并 *没有*包括对全球或引用内置示波器）。

dis.hasname
通过名称访问属性的字节码序列。

dis.hasjrel
具有相对跳转目标的字节码序列。

dis.hasjabs
具有绝对跳转目标的字节码序列。

dis.haslocal
访问局部变量的字节码序列。

dis.hascompare
布尔运算的字节码序列。

32.13。pickletools- 咸菜开发者的工具

源代码： Lib / pickletools.py

这个模块包含了各种与pickle模块的细节有关的常量，关于实现的 一些冗长的评论以及用于分析pickle数据的一些有用的函数。这个模块的内容对于正在开发的Python核心开发人员非常有用pickle。该pickle模块的普通用户可能不会找到pickletools相关的 模块。

32.13.1。命令行用法

3.2版本中的新功能

从命令行调用时，将反汇编一个或多个pickle文件的内容。请注意，如果您想查看保存在pickle中的Python对象而不是pickle格式的细节，则可以使用它。但是，当您想要检查的pickle文件来自不可信源时，它是一个更安全的选项，因为它不会执行pickle字节码。 python -m pickletools-m pickle-m pickletools

例如，在文件中腌制一个元组：(1, 2) x.pickle

```
$ python -m pickle x.pickle
(1, 2)

$ python -m pickletools x.pickle
    0: \x80 PROTO      3
    2: K    BININT1    1
    4: K    BININT1    2
    6: \x86 TUPLE2
    7: q    BINPUT     0
    9: .    STOP
highest protocol among opcodes = 2
```

32.13.1.1。命令行选项

-a，--annotate
    用一个简短的操作码描述来标注每一行。

-o，--output=<file>
    应该写入输出的文件的名称。

-l，--indentlevel=<num>
    缩减新MARK水平的空白数量。

-m，--memo
    拆卸多个对象时，请在反汇编之间保留备忘录。

-p，--preamble=<preamble>

当指定多个pickle文件时，在每次反汇编之前打印给定的前导码。

32.13.2。编程接口

pickletools.**dis**（*pickle*，*out = None*，*memo = None*，*indentlevel = 4*，*annotate = 0* ）

输出咸菜类文件对象的符号反汇编*出来*，默认为sys.stdout。 *泡菜*可以是一个字符串或文件类的对象。 *备忘录*可以是一个Python字典，将用作pickle的备忘录; 它可以用来执行由同一个pickler创建的多个pickles的反汇编。连续的层次，由MARK流中的操作码表示，由*indentlevel*空格缩进。如果赋予非零值进行*注释*，则输出中的每个操作码都用一个简短描述进行注释。*注释*的值用作注释开始的列的提示。

*新版本3.2：*在*注释*的说法。

pickletools.**genops**（*pickle* ）

为pickle中的所有操作码提供一个迭代器，返回一个三元组序列。 *操作码*是一个类的实例；*arg*是操作码参数的解码值，作为Python对象; *pos*是这个操作码所在的位置。 *泡菜*可以是一个字符串或文件类的对象。(opcode, arg, pos) OpcodeInfo

pickletools.**optimize**（*picklestring* ）

消除未使用的PUT 操作码后，返回一个新的等效pickle字符串。优化的腌菜更短，传输时间更短，占用更少的存储空间，并且更有效地去除。

33.杂项服务

本章介绍的模块提供了所有Python版本中可用的杂项服务。这里有一个概述：

33.1。formatter- 通用输出格式

*自3.4版弃用：*由于缺乏使用，格式化程序模块已被弃用。

该模块支持两种接口定义，每种接口定义都有多种实现：*格式化*接口和*格式化*接口所需的*写入*接口。

格式化对象将格式化事件的抽象流转换为作者对象上的特定输出事件。格式化器管理多个堆栈结构以允许改变和恢复写入器对象的各种属性; 作家不需要能够处理相对的变化，也不需要任何种类的“变回”操作。可以通过格式器对象控制的特定作者属性是水平对齐，字体和左边距缩进。提供了一种支持向作者提供任意非排他性风格设置的机制。其他接口有助于格式化不可逆的事件，例如段落分隔。

编写器对象封装设备接口。支持抽象设备，如文件格式，以及物理设备。所提供的实现都适用于抽象设备。该接口提供了用于设置格式化对象管理和将数据插入到输出中的属性的机制。

33.1.1。格式化接口

创建格式化程序的接口取决于要实例化的特定格式化程序类。下面介绍的接口是所有格式化器在初始化后必须支持的必需接口。

一个数据元素在模块级定义：

formatter.AS_IS
: 可以在字体规范中使用的值传递给push_font() 下面描述的方法，或者作为任何其他push_property() 方法的新值。推送该AS_IS值允许pop_property() 调用相应的方法，而无需跟踪属性是否已更改。

以下属性是为格式化程序实例对象定义的：

formatter.writer
: 与格式化程序交互的编写器实例。

formatter.end_paragraph(*blanklines*)
: 关闭所有打开的段落，至少插入*blanklines*下一节之前。

formatter.add_line_break( )
: 如果尚不存在，请添加一个强行换行符。这不会打破逻辑段落。

formatter.add_hor_rule(**args*, ***kw*)
: 在输出中插入水平线。如果当前段落中有数据，则插入一个硬中断，但逻辑段落不会中断。参数和关键字被传递给作者的send_line_break() 方法。

formatter.add_flowing_data(*数据*)
: 提供应该用折叠空白格式化的数据。add_flowing_data()执行空白折叠时，也会考虑前面和后续调用的空白。传递给此方法的数据预计将由输出设备进行文字包装。请注意，由于

需要依赖设备和字体信息，所以任何字词封装仍然必须由写入器对象执行。

formatter.add_literal_data(*数据*)
提供应该无变化地传递给作者的数据。空格（包括换行符和制表符）在*数据*的值中被认为是合法的。

formatter.add_label_data(*格式*, *计数器*)
插入应该放置在当前左边距左侧的标签。这应该用于构建项目符号或编号列表。如果*格式*值是一个字符串，则将其解释为*计数器*的格式规范，该规格应为整数。这种格式的结果成为标签的价值；如果*格式*不是字符串，则直接将其用作标签值。标签值作为作者send_label_data()方法的唯一参数传递。非字符串标签值的解释取决于相关的作者。

格式规格是与计数器值一起用于计算标签值的字符串。格式字符串中的每个字符都被复制到标签值，并识别出一些字符以指示计数器值的变换。具体地，字符'1'表示计数器值格式化为阿拉伯数字，字符'A'和'a'表示在上部和下部的情况下，分别将计数器值的字母表示，并且'I'和'i'用罗马数字，在上部和下部壳体代表计数器值。请注意，字母和罗马变换要求计数器值大于零。

formatter.flush_softspace()
将从先前调用缓冲的任何挂起的空白发送 add_flowing_data()到关联的作者对象。这应该在对作者对象进行任何直接操作之前调用。

formatter.push_alignment(*align*)
将新的对齐设置推入对齐堆栈。这可能是 AS_IS如果不需要改变的话。如果对齐值从先前的设置改变，new_alignment()则使用*对齐*值调用写入者的方法。

formatter.pop_alignment()
恢复之前的对齐。

formatter.push_font(*（尺寸，斜体，粗体，电传类型）*)
更改编写器对象的部分或全部字体属性。未设置为的属性AS_IS设置为传入的值，而其他值保持其当前设置。作者的new_font()方法是用完全解析的字体规范调用的。

formatter.pop_font()
恢复以前的字体。

formatter.push_margin(*margin*)
将左边距缩进量增加1，将逻辑标签*边距*与新缩进量相关联。最初的保证金水平是0。逻辑标签的更改值必须为真值；除了AS_IS不足以改变余量的错误值。

formatter.pop_margin()
恢复以前的保证金。

formatter.push_style(**样式*)
按任意数量的任意样式规范。所有样式都按顺序压入样式堆栈。代表整个堆栈的元组（包括 AS_IS值）被传递给作者的new_styles()方法。

formatter. pop_style (*n = 1*)

弹出传递给的最后*n个*样式规范push_style()。表示修改后的堆栈的元组（包括AS_IS值）传递给作者的new_styles()方法。

formatter. set_spacing (*间距*)

设置作者的间距样式。

formatter. assert_line_data (*flag = 1*)

通知格式化程序数据已被添加到带外当前段落中。这应该在作家被直接操纵时使用。如果编写器操作在输出结束时产生了强行换行，则可选*标志*参数可以设置为false。

33.1.2。格式化程序实现

格式器对象的两个实现由该模块提供。大多数应用程序可以使用这些类中的一个，而无需修改或继承。

class formatter. NullFormatter (*writer = None*)

格式化程序什么都不做。如果*作者*被省略，NullWriter 则创建一个实例。NullFormatter实例没有编写任何方法 。如果实现一个写入器接口，实现应该继承这个类，但不需要继承任何实现。

*类*formatter. AbstractFormatter (*作家*)

标准格式化程序。这种实现已经被证明对许多作者具有广泛的适用性，并且可以在大多数情况下直接使用。它已被用于实现功能齐全的万维网浏览器。

33.1.3。Writer接口

创建编写器的接口取决于正在实例化的特定编写器类。下面介绍的接口是所有编写者在初始化后必须支持的必需接口。请注意，虽然大多数应用程序可以将该AbstractFormatter类用作格式化程序，但写入程序通常必须由应用程序提供。

writer. flush ( )

刷新任何缓冲的输出或设备控制事件。

writer. new_alignment (*align*)

设置对齐方式。该*对准*值可以是任何对象，但按照惯例是一个字符串或None，其中None指示所述写入器的“优选的”对准应该被使用。常规*对齐*值是'left'，'center'，'right'，和'justify'。

writer. new_font (*font*)

设置字体样式。*字体*的值将是None，表示应该使用设备的默认字体，或者表单的元组。大小将是一个字符串，指示应该使用的字体的大小; 特定的字符串及其解释必须由应用程序定义。的*斜体*，*粗体*和*电传*的值是布尔值指定这些字体属性的哪个应当被使用。(size, italic, bold, teletype)

writer. new_margin (*margin*，*level*)

将边距级别设置为整数*级别*，将逻辑标签设置为*边距*。逻辑标签的解释由作者自行决定; 对逻辑标签值的唯一限制是对于非零值的*电平*它不是一个错误的值。

writer.new_spacing（*间距*）
将间距样式设置为*间距*。

writer.new_styles（*风格*）
设置其他样式。的*样式*值是任意值的元组; 该值AS_IS应该被忽略。该*款式*元组可以被解释无论是作为一组或者根据应用和作家的要求执行堆栈。

writer.send_line_break（）
打破目前的路线。

writer.send_paragraph（*blankline*）
产生至少一个段分离*blankline*空行，或等同物。该*blankline*值将是一个整数。请注意，send_line_break()如果需要换行，实施将在此调用之前收到呼叫; 此方法不应包括结束段落的最后一行。它只负责段落之间的垂直间距。

writer.send_hor_rule（**args*，***kw*）
在输出设备上显示水平线。该方法的参数完全是应用程序和作者特定的，应该小心解释。该方法的实现可能会假定已经通过发出换行符send_line_break()。

writer.send_flowing_data（*数据*）
输出字符数据，可根据需要进行文字包装和重新流动。在对此方法的任何调用序列中，作者可能会认为多个空白字符的跨度已折叠为单个空格字符。

writer.send_literal_data（*数据*）
输出已经格式化显示的字符数据。一般来说，这应该被解释为意味着应该保留由换行符指示的换行符并且不应该引入新的换行符。与提供给send_formatted_data()界面的数据不同，数据可能包含嵌入的换行符和制表符。

writer.send_label_data（*数据*）
如果可能，请将*数据*设置为当前左边距的左侧。*数据*的价值 不受限制; 处理非字符串值完全取决于应用程序和写入器。这个方法只会在一行的开头被调用。

33.1.4。Writer实现

作者对象接口的三个实现作为本模块的示例提供。大多数应用程序需要从NullWriter类中派生新的作家 类。

*类*formatter.NullWriter
只提供界面定义的作者; 对任何方法都不采取任何行动。这应该是所有不需要继承任何实现方法的编写者的基类。

*类*formatter.AbstractWriter
一位作家，可以用于调试格式化程序，但不是其他的。每种方法通过在标准输出上打印其名称和参数来简单地宣布自己。

class formatter.DumbWriter（*file* = *None*，*maxcol* = *72* ）

其中上写入输出简单作家类文件对象中传递过来的*文件*，或者，如果*文件*被省略，在标准输出。输出只是简单地按照*maxcol*指定的列数进行*换行*。这门课适合回流一连串的段落。

34. MS Windows特定服务

本章介绍仅适用于MS Windows平台的模块。

34.1。msilib- 读写Microsoft Installer文件

源代码： Lib / msilib / __ init__.py

该msilib支持创建Microsoft安装程序（.msi）文件。由于这些文件通常包含嵌入式“cabinet”文件（.cab），因此它还公开了一个API来创建CAB文件。.cab目前尚未支持阅读文件; 读取.msi数据库的支持是可能的。

这个包旨在提供对.msi文件中所有表的完全访问，因此它是一个相当低级的API。这个包的两个主要应用是distutils命令bdist_msi和创建Python安装程序包本身（尽管目前使用不同版本的msilib）。

包内容大致可以分为四个部分：低级CAB例程，低级MSI例程，高级MSI例程和标准表结构。

msilib.FCICreate（*cabname*，*文件*）

创建一个名为*cabname*的新CAB文件。*文件*必须是元组列表，每个元组包含磁盘上文件的名称和CAB文件内的文件名称。

这些文件按照它们出现在列表中的顺序添加到CAB文件中。使用MSZIP压缩算法将所有文件添加到单个CAB文件中。

针对MSI创建的各个步骤的Python回调函数目前尚未公开。

msilib.UuidCreate（）

返回新唯一标识符的字符串表示形式。这包装Windows API函数UuidCreate()和UuidToString()。

msilib.OpenDatabase（*路径*，*坚持*）

通过调用MsiOpenDatabase返回一个新的数据库对象。*路径*是MSI文件的文件名; *坚持*可以是常数之一MSIDBOPEN_CREATEDIRECT，MSIDBOPEN_CREATE，MSIDBOPEN_DIRECT，MSIDBOPEN_READONLY，或MSIDBOPEN_TRANSACT，并且可以包括所述标志MSIDBOPEN_PATCHFILE。有关这些标志的含义，请参阅Microsoft文档; 根据标志，打开一个现有的数据库，或创建一个新的数据库。

msilib.CreateRecord（*count*）

通过调用返回一个新的记录对象MSICreateRecord()。*count*是记录的字段数。

msilib.init_database（*名称*，*模式*，*ProductName*，*ProductCode*，*ProductVersion*，*制造商*）

创建并返回一个新的数据库*名称*，使用*模式*初始化它，并设置属性*ProductName*，*ProductCode*，*ProductVersion*和*Manufacturer*。

*模式*必须是包含tables和_Validation_records属性的模块对象; 通常msilib.schema应该使用。

该函数返回时，数据库将只包含模式和验证记录。

msilib.add_data(*数据库*, *表格*, *记录*)

将所有*记录*添加到*数据库*中名为*table*的*表*中。

所述*表*参数必须是在MSI模式中的预定义的表中的一个，例如'Feature'，'File'，'Component'，'Dialog'，'Control'，等。

*记录*应该是一个元组列表，每个元组包含根据表的模式记录的所有字段。对于可选字段，None可以通过。

字段值可以是整数，字符串或二进制类的实例。

*类*msilib.Binary(*文件名*)

代表二进制表中的条目;插入这样一个对象使用 add_data()读取名为*文件名*的*文件*到表中。

msilib.add_tables(*数据库*, *模块*)

将所有表格内容从*模块*添加到*数据库*。*模块*必须包含一个属性*表*，其中列出了应为其添加内容的所有表以及每个具有实际内容的表的一个属性。

这通常用于安装序列表。

msilib.add_stream(*数据库*, *名称*, *路径*)

将文件*路径*添加到*数据库*_Stream表中，并使用流名称*名称*。

msilib.gen_uuid()

返回一个新的UUID，格式为MSI通常要求的格式（例如花括号，所有的十六进制都是大写）。

也可以看看: FCICreate UuidCreate UuidToString

34.1.1。数据库对象

Database.OpenView(*sql*)

通过调用返回一个视图对象MSIDatabaseOpenView()。*sql*是要执行的SQL语句。

Database.Commit()

通过调用提交当前事务中待处理的更改 MSIDatabaseCommit()。

Database.GetSummaryInformation(*count*)

通过调用返回一个新的摘要信息对象 MsiGetSummaryInformation()。 *count*是更新值的最大数量。

也可以看看: MSIDatabaseOpenView MSIDatabaseCommit MSIGetSummaryInformation

34.1.2。查看对象

View.Execute (*params*)

通过执行视图的SQL查询MSIViewExecute()。如果 *params*不是None，它是描述查询中参数标记的实际值的记录。

View.GetColumnInfo (*kind*)

通过调用返回描述视图列的记录 MsiViewGetColumnInfo()。*善良*可以是MSICOLINFO_NAMES或者 MSICOLINFO_TYPES。

View.Fetch ( )

通过调用返回查询的结果记录MsiViewFetch()。

View.Modify (*kind* , *data*)

通过调用修改视图 MsiViewModify()。 一*种*可以是一个 MSIMODIFY_SEEK , MSIMODIFY_REFRESH , MSIMODIFY_INSERT , MSIMODIFY_UPDATE , MSIMODIFY_ASSIGN , MSIMODIFY_REPLACE , MSIMODIFY_MERGE , MSIMODIFY_DELETE , MSIMODIFY_INSERT_TEMPORARY , MSIMODIFY_VALIDATE , MSIMODIFY_VALIDATE_NEW , MSIMODIFY_VALIDATE_FIELD，或MSIMODIFY_VALIDATE_DELETE。

*数据*必须是描述新数据的记录。

View.Close ( )

关闭视图，通过MsiViewClose()。

也可以看看: MsiViewExecute MSIViewGetColumnInfo MsiViewFetch MsiViewModify MsiViewClose

34.1.3。摘要信息对象

SummaryInformation.GetProperty (*field*)

通过返回摘要的属性MsiSummaryInfoGetProperty()。 *字段*是属性的名称，并且可以是一个常量 PID_CODEPAGE , PID_TITLE , PID_SUBJECT , PID_AUTHOR , PID_KEYWORDS , PID_COMMENTS , PID_TEMPLATE , PID_LASTAUTHOR , PID_REVNUMBER , PID_LASTPRINTED , PID_CREATE_DTM , PID_LASTSAVE_DTM , PID_PAGECOUNT , PID_WORDCOUNT , PID_CHARCOUNT , PID_APPNAME，或PID_SECURITY。

SummaryInformation.GetPropertyCount ( )

通过MsiSummaryInfoGetPropertyCount()。返回摘要属性的数量 。

SummaryInformation.SetProperty (*字段* , *值*)

通过设置属性MsiSummaryInfoSetProperty()。*字段*可以具有与in相同的值GetProperty() , *value*是该属性的新值。可能的值类型是整数和字符串。

SummaryInformation.Persist ( )

使用修改后的属性写入摘要信息流 MsiSummaryInfoPersist()。

也可以看看: MsiSummaryInfoGetProperty MsiSummaryInfoGetPropertyCount MsiSummaryInfoSetProperty MsiSummaryInfoPersist

34.1.4。记录对象

Record.GetFieldCount ( )
通过返回记录的字段数 MsiRecordGetFieldCount()。

Record.GetInteger (*field*)
尽可能以整数形式返回*字段*的值。 *字段*必须是整数。

Record.GetString (*field*)
尽可能将*字段*的值作为字符串返回。 *字段*必须是整数。

Record.SetString (*字段* , *值*)
通过将*字段*设置为*值*MsiRecordSetString()。 *字段*必须是整数; *值*一个字符串。

Record.SetStream (*字段* , *值*)
将*字段*设置为文件名为*value*的内容，直到 MsiRecordSetStream()。 *字段*必须是整数; *值*一个字符串。

Record.SetInteger (*字段* , *值*)
通过将*字段*设置为*值*MsiRecordSetInteger()。两个*字段*和 *值*必须是一个整数。

Record.ClearData ( )
通过将记录的所有字段设置为0 MsiRecordClearData()。

也可以看看: MsiRecordGetFieldCount MsiRecordSetString MsiRecordSetStream MsiRecordSetInteger MsiRecordClearData

34.1.5。错误

MSI功能的所有包装都会增加MSIError; 异常中的字符串将包含更多细节。

34.1.6。CAB对象

class msilib.CAB (*name*)
该类CAB代表一个CAB文件。在MSI构建过程中，文件将同时添加到Files表格和CAB文件中。然后，当添加完所有文件后，可以编写CAB文件，然后将其添加到MSI文件中。

*name*是MSI文件中CAB文件的名称。

append (*完整* , *文件* , *逻辑*)

添加与路径的文件*完整*的CAB文件，名称下 *的逻辑*。如果已经有一个名为*logical*的文件，则会创建一个新的文件名。

返回CAB文件中的文件的索引以及CAB文件内的文件的新名称。

commit（*数据库*）

生成一个CAB文件，将其作为一个流添加到MSI文件，将其放入 Media表中，并从磁盘中删除生成的文件。

34.1.7。目录对象

*类*msilib.Directory（*数据库*，*cab*，*basedir*，*physical*，*logical*，*default*[，*componentflags*]）

在目录表中创建一个新的目录。在目录的每个时间点都有一个当前组件，可以通过明确创建start_component()，也可以在第一次添加文件时隐式创建 。文件被添加到当前组件中，并被添加到cab文件中。要创建目录，需要指定（可以None）基本目录对象，物理目录的路径和逻辑目录名称。 *default* 指定目录表中的DefaultDir插槽。*componentflags*指定新组件获取的默认标志。

start_component（*component* = *None*，*feature* = *None*，*flags* = *None*，*keyfile* = *None*，*uuid* = *None*）

将一个条目添加到Component表中，并将此组件作为此目录的当前组件。如果没有给出组件名称，则使用目录名称。如果没有*功能*被赋予，则使用当前功能。如果没有 *标志*给出，将使用目录的缺省标志。如果没有 给出*密钥文件*，KeyPath在Component表中保留为空。

add_file（*file*，*src* = *None*，*version* = *None*，*language* = *None*）

将文件添加到目录的当前组件，如果没有当前组件，则启动一个新文件。默认情况下，源文件和文件表中的文件名称是相同的。如果指定了*src*文件，则会相对于当前目录进行解释。或者，可以为File表中的条目指定*版本*和*语言*。

glob（*pattern*，*exclude* = *None*）

按照glob模式中的指定将文件列表添加到当前组件。*排除*列表中可以排除单个文件。

remove_pyc（）

.pyc在卸载时删除文件。

也可以看看： 目录表 文件表 组件表 FeatureComponents表

34.1.8。特点

class msilib.Feature（*db*，*id*，*title*，*desc*，*display*，*level* = *1*，*parent* = *None*，*directory* = *None*，*attributes* = *0*）

Feature使用值*id*，*parent.id*， *标题*，*desc*，*显示*，*级别*，*目录*和*属性*向表中添加新记录。结果要素对象可以传递给start_component()方法 Directory。

set_current ( )

使此功能成为当前功能msilib。新组件会自动添加到默认功能中，除非明确指定了功能。

也可以看看: 功能表

34.1.9。GUI类

msilib提供了几个将GUI表包装在MSI数据库中的类。但是，没有提供标准的用户界面; 用于bdist_msi创建带用户界面的MSI文件来安装Python软件包。

class msilib. Control (*dlg* , *name*)

对话框控件的基类。*dlg*是控件所属的对话框对象，*name*是控件的名称。

event (*事件* , *参数* , *条件= 1* , *排序=无*)

ControlEvent为此控件输入表格。

mapping (*事件* , *属性*)

EventMapping为此控件输入表格。

condition (*行动* , *条件*)

ControlCondition为此控件输入表格。

*类*msilib. RadioButtonGroup (*dlg* , *name* , *property*)

创建一个名为*name*的单选按钮控件。*属性*是选择单选按钮时设置的安装程序属性。

add (*name* , *x* , *y* , *width* , *height* , *text* , *value = None*)

将名为*name*的单选按钮添加到组中，坐标为*x* , *y* ， *宽度* , *高度*和标签*文本*。如果*值*是None ，它默认为*名称*。

class msilib. Dialog (*db* , *name* , *x* , *y* , *w* , *h* , *attr* , *title* , *first* , *default* , *cancel*)

返回一个新的Dialog对象。 Dialog使用指定的坐标，对话框属性，标题，第一个，默认和取消控件的名称创建表中的条目。

control (*name* , *type* , *x* , *y* , *width* , *height* , *attributes* , *property* , *text* , *control_next* , *help*)

返回一个新的Control对象。 Control表格中的条目由指定的参数组成。

这是一种通用的方法; 对于特定的类型，提供了专门的方法。

text (*名称* , *x* , *y* , *宽度* , *高度* , *属性* , *文本*)

添加并返回一个Text控件。

bitmap (*名称* , *x* , *y* , *宽度* , *高度* , *文本*)

添加并返回一个Bitmap控件。

line (*名称* , *x* , *y* , *宽度* , *高度*)

添加并返回一个Line控件。

pushbutton (*name* , *x* , *y* , *width* , *height* , *attributes* , *text* , *next_control*)

添加并返回一个PushButton控件。

radiogroup (*name* , *x* , *y* , *width* , *height* , *attributes* , *property* , *text* , *next_control*)

添加并返回一个RadioButtonGroup控件。

checkbox (*name* , *x* , *y* , *width* , *height* , *attributes* , *property* , *text* , *next_control*)

添加并返回一个CheckBox控件。

也可以看看: 对话框表 控制表 控制类型 ControlCondition表 ControlEvent表 EventMapping表 RadioButton表

34.1.10。预计算表

msilib提供了几个仅包含模式和表定义的子包。目前，这些定义基于MSI 2.0版。

msilib. schema

这是MSI 2.0的标准MSI模式，其中*tables*变量提供表定义列表，而*_Validation_records*提供MSI验证数据。

msilib. sequence

该模块包含标准序列表的表格内容： *AdminExecuteSequence* ， *AdminUISequence* ， *AdvtExecuteSequence* ， *InstallExecuteSequence*和*InstallUISequence*。

msilib. text

此模块包含标准安装程序操作的UIText和ActionText表的定义。

34.2。msvcrt- 来自MS VC ++运行时的有用例程

这些功能可以访问Windows平台上的一些有用的功能。一些更高级别的模块使用这些函数来构建其服务的Windows实现。例如，该getpass模块在该getpass()函数的实现中使用它。

有关这些功能的更多文档可以在Platform API文档中找到。

该模块实现控制台I / O API的常规和宽字符变体。普通的API只处理ASCII字符，对于国际化应用程序的使用有限。应尽可能使用宽字符API。

*在版本3.3中进行了更改：*此模块中的操作现在引发引发的OSError地方IOError 。

34.2.1。文件操作

msvcrt.locking（*fd*，*mode*，*nbytes* ）

基于C运行时的文件描述符*fd*锁定文件的一部分。OSError失败时提高 。文件的锁定区域从当前文件位置延伸*nbytes*字节，并可能会延续到文件末尾。 *模式*必须是LK_*下面列出的常量之一。文件中的多个区域可能同时被锁定，但可能不会重叠。相邻区域不合并; 他们必须单独解锁。

msvcrt.LK_LOCK
msvcrt.LK_RLCK

锁定指定的字节。如果字节无法锁定，程序会在1秒后立即再次尝试。如果在尝试10次后，字节不能被锁定，OSError则会引发。

msvcrt.LK_NBLCK
msvcrt.LK_NBRLCK

锁定指定的字节。如果字节不能被锁定，OSError则引发。

msvcrt.LK_UNLCK

解锁必须先前锁定的指定字节。

msvcrt.setmode（*fd*，*flags* ）

设置文件描述符*fd*的行结束转换模式。要将其设置为文本模式，*标志*应该是os.O_TEXT; 对于二进制，它应该是 os.O_BINARY。

msvcrt.open_osfhandle（*句柄*，*标志*）

从文件句柄*句柄*创建一个C运行时文件描述符。该*标志* 参数应该是按位OR的os.O_APPEND ，os.O_RDONLY和os.O_TEXT。返回的文件描述符可以用作os.fdopen()创建文件对象的参数。

msvcrt.get_osfhandle（*fd* ）

返回文件描述符*fd*的文件*句柄*。OSError如果 *fd*无法识别则引发。

34.2.2。控制台I /

msvcrt.kbhit（）

如果按键等待读取，则返回true。

msvcrt.getch（）

读取按键并将结果字符作为字节字符串返回。没有任何东西被回应到控制台。如果按键不可用，此呼叫将被阻止，但不会等待Enter按下。如果按下的键是特殊功能键，则返回'\000'或'\xe0'；下一次调用将返回键码。该Control-C按键无法使用该功能来读取。

msvcrt.getwch（）

宽字符变体getch()，返回一个Unicode值。

msvcrt.getche（）

getch()与之类似，但如果它代表可打印字符，则按键将被回显。

msvcrt.getwche（）

宽字符变体getche()，返回一个Unicode值。

msvcrt.putch（*char*）

将字节字符串*char*打印到控制台而不缓冲。

msvcrt.putwch（*unicode_char*）

putch()接受Unicode值的宽字符变体。

msvcrt.ungetch（*char*）

导致字符串*char*被“推回”到控制台缓冲区中；它将成为由getch()或读取的下一个字符getche()。

msvcrt.ungetwch（*unicode_char*）

ungetch()接受Unicode值的宽字符变体。

34.2.3。其他功能

msvcrt.heapmin（）

强制malloc()堆清理自己并将未使用的块返回给操作系统。失败时，这会引发OSError。

34.3。 winreg- Windows注册表访问

这些函数将Windows注册表API暴露给Python。不使用整数作为注册表句柄，而是使用句柄对象来确保句柄正确关闭，即使程序员忽略了明确关闭它们。

*在版本3.3中进行了更改：*此模块中的几个函数用于引发一个 WindowsError ，它现在是一个别名OSError。

34.3.1。函数

该模块提供以下功能：

winreg.CloseKey（ *hkey* ）
关闭以前打开的注册表项。在*HKEY*参数指定以前打开的关键。

> **注意：** 如果*hkey*未使用此方法（或通过hkey.Close()）关闭，那么当*hkey*对象被Python销毁时它将被关闭。

winreg.ConnectRegistry（ *computer_name* ，*key* ）
建立到另一台计算机上的预定义注册表句柄的连接，并返回句柄对象。

*computer_name*是表单的远程计算机的名称 r"\\computername"。如果None使用本地计算机。

*键*是连接到的预定义句柄。

返回值是打开的键的句柄。如果该功能失败， OSError则会引发异常。

*版本3.3中更改：*请参阅上文。

winreg.CreateKey（ *key* ，*sub_key* ）
创建或打开指定的键，返回一个 句柄对象。

*键*是一个已经打开的键，或者是一个预定义的 HKEY_ *常量。

*sub_key*是一个字符串，用于命名此方法打开或创建的密钥。

如果*key*是预定义键之一，则*sub_key*可能是None。在这种情况下，返回的句柄是传递给该函数的相同的键句柄。

如果密钥已经存在，该功能将打开现有密钥。

返回值是打开的键的句柄。如果该功能失败， OSError则会引发异常。

*版本3.3中更改：*请参阅上文。

winreg.CreateKeyEx（ *key* ，*sub_key* ，*reserved = 0* ，*access = KEY_WRITE* ）

创建或打开指定的键，返回一个 句柄对象。

*键*是一个已经打开的键，或者是一个预定义的 HKEY_ *常量。

*sub_key*是一个字符串，用于命名此方法打开或创建的密钥。

*保留*是保留整数，并且必须为零。默认值为零。

*访问*是一个整数，它指定描述所需密钥安全访问的访问掩码。默认是KEY_WRITE。请参阅访问权限获取其他允许的值。

如果*key*是预定义键之一，则*sub_key*可能是None。在这种情况下，返回的句柄是传递给该函数的相同的键句柄。

如果密钥已经存在，该功能将打开现有密钥。

返回值是打开的键的句柄。如果该功能失败， OSError则会引发异常。

3.2版本中的新功能

*版本3.3中更改：*请参阅上文。

winreg.DeleteKey（*key*，*sub_key* ）

删除指定的密钥。

*键*是一个已经打开的键，或者是一个预定义的 HKEY_ *常量。

*sub_key*是一个字符串，它必须是由*key* 参数标识的*密钥*的子*密钥*。该值不能是None ，并且该键可能没有子键。

此方法不能使用子密钥删除密钥。

如果方法成功，则删除整个键（包括其所有值）。如果该方法失败，OSError则会引发异常。

*版本3.3中更改：*请参阅上文。

winreg.DeleteKeyEx（*key*，*sub_key*，*access = KEY_WOW64_64KEY*，*reserved = 0* ）

删除指定的密钥。

> **注意：** 该DeleteKeyEx()函数使用RegDeleteKeyEx Windows API函数实现，该函数特定于64位版本的Windows。请参阅RegDeleteKeyEx文档。

*键*是一个已经打开的键，或者是一个预定义的 HKEY_ *常量。

*sub_key*是一个字符串，它必须是由*key*参数标识的*密钥*的子 *密钥*。该值不能是None ，并且该键可能没有子键。

*保留*是保留整数，并且必须为零。默认值为零。

*访问*是一个整数，它指定描述所需密钥安全访问的访问掩码。默认是KEY_WOW64_64KEY。请参阅 访问权限获取其他允许的值。

此方法不能使用子密钥删除密钥。

如果方法成功，则删除整个键（包括其所有值）。如果该方法失败，OSError则会引发异常。

在不支持的Windows版本上，NotImplementedError引发。

3.2版本中的新功能

*版本3.3中更改：*请参阅上文。

winreg.DeleteValue（*key*，*value*）

从注册表项中删除一个命名值。

*键*是一个已经打开的键，或者是一个预定义的 HKEY_*常量。

*值*是标识要删除的值的字符串。

winreg.EnumKey（*key*，*索引*）

枚举打开的注册表项的子项，返回一个字符串。

*键*是一个已经打开的键，或者是一个预定义的 HKEY_*常量。

*index*是一个整数，用于标识要检索的键的索引。

该函数每次调用时都会检索一个子项的名称。通常会重复调用它，直到发生OSError异常，表示没有更多值可用。

*版本3.3中更改：*请参阅上文。

winreg.EnumValue（*key*，*索引*）

枚举打开的注册表项的值，返回一个元组。

*键*是一个已经打开的键，或者是一个预定义的 HKEY_*常量。

*index*是一个整数，用于标识要检索的值的索引。

该函数每次调用时都会检索一个子项的名称。通常会重复调用它，直到引发OSError异常，表示没有更多值。

结果是3个元素的元组：

指数	含义
0	一个标识值名称的字符串
1	包含值数据的对象，其类型取决于基础注册表类型
2	标识值数据类型的整数（请参阅文档中的表格 SetValueEx()）

*版本3.3中更改：*请参阅上文。

winreg.ExpandEnvironmentStrings（*str*）

扩展%NAME%字符串中的环境变量占位符，如 REG_EXPAND_SZ：

```
>>> ExpandEnvironmentStrings('%windir%')
'C:\\Windows'
```

winreg.FlushKey（*重点*）

将密钥的所有属性写入注册表。

*键*是一个已经打开的键，或者是一个预定义的 HKEY_ *常量。

不需要调用FlushKey()来更改密钥。注册表使用其懒惰的冲洗程序将注册表更改刷新到磁盘。在系统关闭时，注册表更改也会刷新到磁盘。不同的是CloseKey()，FlushKey()只有当所有的数据都被写入注册表时，该 方法才会返回。应用程序只能FlushKey()在绝对确定注册表更改在磁盘上时才应调用。

> **注意：** 如果您不知道是否需要FlushKey()拨打电话，那可能不是。

winreg.LoadKey（*key*，*sub_key*，*file_name*）

在指定的密钥下创建一个子密钥，并将来自指定文件的注册信息存储到该子密钥中。

*键*是由ConnectRegistry()常量HKEY_USERS或其中一个常量返回的句柄HKEY_LOCAL_MACHINE。

*sub_key*是标识要加载的子项的字符串。

*file_name*是从中加载注册表数据的文件的名称。该文件必须已经使用该SaveKey()功能创建。在文件分配表（FAT）文件系统下，文件名可能没有扩展名。

要在通话LoadKey()如果调用进程没有失败 SE_RESTORE_PRIVILEGE的特权。请注意，权限与权限不同 - 请参阅RegLoadKey文档以获取更多详细信息。

如果*key*是返回的句柄ConnectRegistry()，则*file_name*中指定的路径是相对于远程计算机的。

winreg.OpenKey（*key*，*sub_key*，*reserved = 0*，*access = KEY_READ*）
winreg.OpenKeyEx（*key*，*sub_key*，*reserved = 0*，*access = KEY_READ*）

打开指定的键，返回一个句柄对象。

*键*是一个已经打开的键，或者是一个预定义的 HKEY_ *常量。

*sub_key*是标识要打开的子键的字符串。

*保留*是保留整数，并且必须为零。默认值为零。

*访问*是一个整数，它指定描述所需密钥安全访问的访问掩码。默认是KEY_READ。请参阅访问权限获取其他允许的值。

结果是指定键的新句柄。

如果该功能失败，OSError则引发。

*在版本3.2中更改：*允许使用命名参数。

*版本3.3中更改：*请参阅上文。

winreg.QueryInfoKey（*重点*）
将元素的信息作为元组返回。

*键*是一个已经打开的键，或者是一个预定义的 HKEY_ *常量。

结果是3个元素的元组：

指数	含义
0	给出这个键所具有的子键数量的整数。
1	给出这个键值的个数的整数。
2	1601年1月1日以来的最后修改密钥（如果可用）的整数，为100纳秒。

winreg.QueryValue（*key*，*sub_key* ）
以字符串形式检索密钥的未命名值。

*键*是一个已经打开的键，或者是一个预定义的 HKEY_ *常量。

*sub_key*是一个字符串，它保存与该值关联的子键的名称。如果此参数为None或为空，则该函数将检索由SetValue()按键标识的*密钥*的方法设置的值。

注册表中的值具有名称，类型和数据组件。此方法检索具有NULL名称的键的第一个值的数据。但底层API调用不会返回该类型，因此QueryValueEx()如果可能的话，请始终使用 。

winreg.QueryValueEx（*key*，*value_name* ）
检索与打开的注册表项关联的指定值名称的类型和数据。

*键*是一个已经打开的键，或者是一个预定义的 HKEY_ *常量。

*value_name*是一个字符串，指示要查询的值。

结果是2个元素的元组：

指数	含义
0	注册表项的值。
1	给这个值注册表类型的整数（请参阅文档中的表SetValueEx() ）

winreg.SaveKey（*key*，*file_name* ）
将指定的密钥及其所有子密钥保存到指定的文件。

*键*是一个已经打开的键，或者是一个预定义的 HKEY_ *常量。

*file_name*是保存注册表数据的文件的名称。这个文件不可能已经存在。如果此文件名包含扩展名，则该方法不能在文件分配表（FAT）文件系统上使用该扩展名LoadKey() 。

如果*密钥*表示远程计算机上的密钥，则*file_name*描述的路径 是相对于远程计算机的。此方法的调用者必须拥有SeBackupPrivilege安全特权。请注意，权限与权限不同 - 请参阅用户权限和权限文档之间的 冲突 以获取更多详细信息。

该函数将*security_attributes的*NULL传递给API。

winreg. SetValue (*key* , *sub_key* , *type* , *value*)

将值与指定的键关联。

*键*是一个已经打开的键，或者是一个预定义的 HKEY_ *常量。

*sub_key*是一个字符串，用于命名与其关联的子键。

*type*是一个指定数据*类型*的整数。目前这个必须是 REG_SZ ，意味着只支持字符串。使用该 SetValueEx() 功能支持其他数据类型。

*value*是一个指定新值的字符串。

如果由*sub_key*参数指定的键不存在，则SetValue函数会创建它。

值的长度受可用内存的限制。长值（超过2048字节）应该以文件形式存储在配置注册表中。这有助于注册表高效地执行。

由*关键*参数标识的*关键字*必须已打开才能 KEY_SET_VALUE访问。

winreg. SetValueEx (*key* , *value_name* , *reserved* , *type* , *value*)

将数据存储在打开的注册表项的值字段中。

*键*是一个已经打开的键，或者是一个预定义的 HKEY_ *常量。

*value_name*是一个字符串，用于命名与该值关联的子键。

*保留*可以是任何东西 - 零总是传递给API。

*type*是一个指定数据*类型*的整数。请参阅 可用类型的值类型。

*value*是一个指定新值的字符串。

此方法还可以为指定的键设置附加值和类型信息。由关键参数标识的关键字必须已打开才能 KEY_SET_VALUE访问。

要打开密钥，请使用CreateKey()或OpenKey()方法。

值的长度受可用内存的限制。长值（超过2048字节）应该以文件形式存储在配置注册表中。这有助于注册表高效地执行。

winreg. DisableReflectionKey (*重点*)

禁用在64位操作系统上运行的32位进程的注册表反射。

*键*是一个已经打开的键，或者是一个预定义的HKEY_ *常量。

NotImplemented如果在32位操作系统上执行，通常会提高。

如果该键不在反射列表中，则该功能成功但不起作用。禁用键的反射不会影响任何子键的反射。

winreg.EnableReflectionKey（*重点*）
为指定的禁用密钥恢复注册表反射。

*键*是一个已经打开的键，或者是一个预定义的HKEY_ *常量。

NotImplemented如果在32位操作系统上执行，通常会提高。

恢复键的反射不影响任何子键的反射。

winreg.QueryReflectionKey（*重点*）
确定指定键的反射状态。

*键*是一个已经打开的键，或者是一个预定义的 HKEY_ *常量。

True如果禁用了反射，则返回。

NotImplemented如果在32位操作系统上执行，通常会提高。

34.3.2。常量

以下常量被定义用于许多_winreg功能。

34.3.2.1。HKEY_ *常量

winreg.HKEY_CLASSES_ROOT
从属于此项的注册表项定义文档的类型（或类）以及与这些类型关联的属性。Shell和COM应用程序使用存储在该密钥下的信息。

winreg.HKEY_CURRENT_USER
从属于此密钥的注册表项定义当前用户的首选项。这些首选项包括环境变量的设置，有关程序组，颜色，打印机，网络连接和应用程序首选项的数据。

winreg.HKEY_LOCAL_MACHINE
从属于此密钥的注册表项定义了计算机的物理状态，包括有关总线类型，系统内存以及安装的硬件和软件的数据。

winreg.HKEY_USERS
从属于此项的注册表项定义本地计算机上新用户的默认用户配置以及当前用户的用户配置。

winreg.HKEY_PERFORMANCE_DATA
从属于此密钥的注册表项允许您访问性能数据。数据实际上并不存储在注册表中;注册表功能使系统从其来源收集数据。

winreg.HKEY_CURRENT_CONFIG
包含有关本地计算机系统当前硬件配置文件的信息。

winreg.HKEY_DYN_DATA
此密钥不适用于98之后的Windows版本。

34.3.2.2。访问权限

有关更多信息，请参阅注册表项安全和访问。

winreg.KEY_ALL_ACCESS
结合 STANDARD_RIGHTS_REQUIRED ， ， KEY_QUERY_VALUE ， KEY_SET_VALUE ， KEY_CREATE_SUB_KEY ， KEY_ENUMERATE_SUB_KEYS ， KEY_NOTIFY 和 KEY_CREATE_LINK 访问权限。

winreg.KEY_WRITE
结合STANDARD_RIGHTS_WRITE，KEY_SET_VALUE和 KEY_CREATE_SUB_KEY访问权限。

winreg.KEY_READ
结合 STANDARD_RIGHTS_READ ， ，KEY_QUERY_VALUE ， KEY_ENUMERATE_SUB_KEYS 和 KEY_NOTIFY值。

winreg.KEY_EXECUTE
相当于KEY_READ。

winreg.KEY_QUERY_VALUE
需要查询注册表项的值。

winreg.KEY_SET_VALUE
创建，删除或设置注册表值所需。

winreg.KEY_CREATE_SUB_KEY
创建注册表项的子项所需。

winreg.KEY_ENUMERATE_SUB_KEYS
枚举注册表项的子项是必需的。

winreg.KEY_NOTIFY
需要为注册表项或注册表项的子项申请更改通知。

winreg.KEY_CREATE_LINK
保留供系统使用。

34.3.2.2.1。64位具体

有关更多信息，请参阅访问备用注册表视图。

winreg.KEY_WOW64_64KEY
表示64位Windows上的应用程序应该在64位注册表视图上运行。

winreg.KEY_WOW64_32KEY
表示64位Windows上的应用程序应该在32位注册表视图上运行。

34.3.2.3。值类型

有关更多信息，请参阅注册表值类型。

winreg.REG_BINARY
任何形式的二进制数据。

winreg.REG_DWORD
32位数字。

winreg.REG_DWORD_LITTLE_ENDIAN
小端格式的32位数字。相当于REG_DWORD。

winreg.REG_DWORD_BIG_ENDIAN
大端格式的32位数字。

winreg.REG_EXPAND_SZ
包含对环境变量（%PATH%）的引用的以空字符结尾的字符串。

winreg.REG_LINK
一个Unicode符号链接。

winreg.REG_MULTI_SZ
一系列以空字符结尾的字符串，由两个空字符终止。（Python自动处理该终止。）

winreg.REG_NONE
没有定义的值类型。

winreg.REG_QWORD
一个64位数字。

3.6版本中的新功能。

winreg.REG_QWORD_LITTLE_ENDIAN
小端格式的64位数字。相当于REG_QWORD。

3.6版本中的新功能。

winreg.REG_RESOURCE_LIST
设备驱动程序资源列表。

winreg.REG_FULL_RESOURCE_DESCRIPTOR
硬件设置。

winreg.REG_RESOURCE_REQUIREMENTS_LIST
硬件资源列表。

winreg.REG_SZ
以空字符结尾的字符串。

34.3.3。注册表句柄对象

该对象包装Windows HKEY对象，当对象被销毁时自动关闭它。为了保证清理，你可以调用Close()对象上的方法或CloseKey()函数。

该模块中的所有注册表函数都返回这些对象中的一个。

该模块中接受句柄对象的所有注册表函数也接受整数，但鼓励使用句柄对象。

处理对象提供了语义__bool__()- 因此

```
if handle:
    print("Yes")
```

Yes如果句柄当前有效（尚未关闭或分离），将打印。

该对象还支持比较语义，因此如果句柄对象都引用相同的基础Windows句柄值，则句柄对象将进行比较。

处理对象可以转换为整数（例如，使用内置 int()函数），在这种情况下返回基础Windows句柄值。您也可以使用该Detach()方法来返回整型句柄，并且还可以从句柄对象中断开Windows句柄。

PyHKEY.Close( )
关闭底层的Windows句柄。

如果句柄已关闭，则不会产生错误。

PyHKEY.Detach( )
从句柄对象分离Windows句柄。

结果是一个整数，它在分离之前保存句柄的值。如果手柄已经分离或关闭，则返回零。

调用此函数后，句柄实际上失效，但句柄未关闭。当需要底层Win32句柄在句柄对象的生命周期之外存在时，您可以调用此函数。

PyHKEY.__enter__( )

PyHKEY.__exit__(*exc_info)

HKEY对象实现__enter__()并 __exit__()支持with语句的上下文协议 ：

```
with OpenKey(HKEY_LOCAL_MACHINE, "foo") as key:
    ...  # work with key
```

将在控制离开块时自动关闭*鍵*with。

34.4。 winsound- Windows的声音播放界面

该winsound模块提供对Windows平台提供的基本声音播放设备的访问。它包含函数和几个常量。

winsound.Beep（*频率*，*持续时间*）

嘟嘟PC的扬声器。的*频率*参数指定的频率，以赫兹为单位的声音的，并且必须通过32767在范围37。在*时间* 参数指定毫秒的声音应该持续数。如果系统无法发出扬声器提示音RuntimeError。

winsound.PlaySound（*声音*，*旗帜*）

PlaySound()从平台API 调用底层函数。的 *声音*参数可以是文件名，一个系统声音别名，音频数据作为一个 像字节对象，或None。它的解释取决于*标志*的值，它可以是下面描述的常量的按位或运算组合。如果*声音*参数是 None ，则任何当前播放的波形声音都会停止。如果系统显示错误，RuntimeError则引发。

winsound.MessageBeep（*type = MB_OK* ）

MessageBeep()从平台API 调用底层函数。这会播放注册表中指定的声音。该*类型*参数指定哪个音质发挥; 可能的值是-1 ，MB_ICONASTERISK ， MB_ICONEXCLAMATION ，MB_ICONHAND ，MB_ICONQUESTION ，和MB_OK ，以下所有描述的。该值-1产生“简单的嘟嘟声”; 如果声音无法播放，这是最后的回退。如果系统显示错误，RuntimeError则引发。

winsound.SND_FILENAME

该*声音*参数是一个WAV文件的名称。不要与...一起使用 SND_ALIAS。

winsound.SND_ALIAS

该*声音*参数是从注册表中声音的关联名称。如果注册表中不包含此类名称，则除非SND_NODEFAULT另外指定，否则请播放系统默认声音 。如果没有登录默认声音，请提高RuntimeError。不要与...一起使用SND_FILENAME。

所有Win32系统至少支持以下内容; 大多数系统支持更多：

PlaySound() *名称*	**相应的控制面板声音名称**
'SystemAsterisk'	星号
'SystemExclamation'	感叹
'SystemExit'	退出Windows
'SystemHand'	关键停止
'SystemQuestion'	题

例如：

```
import winsound
# Play Windows exit sound.
winsound.PlaySound("SystemExit", winsound.SND_ALIAS)
```

```
# Probably play Windows default sound, if any is registered (because
# "*" probably isn't the registered name of any sound).
winsound.PlaySound("*", winsound.SND_ALIAS)
```

winsound.SND_LOOP
反复播放声音。该SND_ASYNC标志也必须用于避免阻塞。不能用于SND_MEMORY。

winsound.SND_MEMORY
的*声音*参数PlaySound()是WAV文件的存储器中的图像，作为一个 类字节对象。

> **注意：** 这个模块不支持异步播放内存映像，所以这个标志的组合SND_ASYNC将会提升RuntimeError。

winsound.SND_PURGE
停止播放指定声音的所有实例。

> **注意：** 现代Windows平台不支持此标志。

winsound.SND_ASYNC
立即返回，允许声音异步播放。

winsound.SND_NODEFAULT
如果找不到指定的声音，请不要播放系统默认声音。

winsound.SND_NOSTOP
不要中断当前播放的声音。

winsound.SND_NOWAIT
如果声音驱动程序正忙，请立即返回。

> **注意：** 现代Windows平台不支持此标志。

winsound.MB_ICONASTERISK
播放SystemDefault声音。

winsound.MB_ICONEXCLAMATION
播放SystemExclamation声音。

winsound.MB_ICONHAND
播放SystemHand声音。

winsound.MB_ICONQUESTION
播放SystemQuestion声音。

winsound.MB_OK
播放SystemDefault声音。

35. Unix特定服务

本章介绍的模块提供了Unix操作系统特有的功能的接口，或者在某些情况下提供给它的一些或多种变体。这里有一个概述：

35.1。posix- 最常见的POSIX系统调用

此模块提供对由C标准和POSIX标准（一个伪装的Unix界面）标准化的操作系统功能的访问。

不要直接导入此模块。 相反，导入模块os，该模块提供了该接口的*便携*版本。在Unix上，os模块提供了posix接口的超集。在非Unix操作系统上，该posix模块不可用，但通过该os接口始终可用子集。一旦os导入， 使用它就*不会有*性能损失posix。此外， os还提供了一些附加功能，例如putenv()在输入os.environ内容发生更改时自动调用 。

错误报告为例外; 通常的例外是针对类型错误给出的，而系统调用报告的错误会引发OSError。

35.1.1。大文件支持

多个操作系统（包括AIX，HP-UX，Irix的和Solaris），用于从一个C编程模型大于2吉布其中文件提供支撑 int和long是32位的值。这通常通过将相关大小和偏移量类型定义为64位值来实现。这些文件有时被称为*大文件*。

在Python的大小off_t大于a long并且类型可用并且至少与a一样大时，在Python中启用大文件支持。可能需要使用某些编译器标志来配置和编译Python以启用此模式。例如，对于最新版本的Irix，默认情况下它是启用的，但对于Solaris 2.6和2.7，您需要执行以下操作：long long off_t

```
CFLAGS="`getconf LFS_CFLAGS`" OPT="-g -O2 $CFLAGS" \
        ./configure
```

在支持大文件的Linux系统上，这可能起作用：

```
CFLAGS='-D_LARGEFILE64_SOURCE -D_FILE_OFFSET_BITS=64' OPT="-g -O2 $CFLAGS" \
        ./configure
```

35.1.2。着名的模块内容

除os模块文档中描述的许多功能外，还 posix定义了以下数据项目：

posix.**environ**

表示解释器启动时的字符串环境的字典。键和值在Unix上是字节而在Windows上是str。例如，environ[b'HOME']（environ['HOME']在Windows上）是您的主目录的路径名，相当于getenv("HOME")C.

修改此字典不会影响通过的字符串环境 execv()，popen()或system()；如果你需要改变环境，传递environ到execve()或添加变量赋值和导出报表的命令字符串 system()或popen()。

*在版本3.2中更改：*在Unix上，键和值是字节。

注意： 该os模块提供了一个替代实施，environ 其中更新修改环境。还要注意，更新os.environ会使这本字典过时。建议使用此 os模块版本直接访问 posix模块。

35.2。 pwd- 密码数据库

该模块提供对Unix用户帐户和密码数据库的访问。它适用于所有的Unix版本。

密码数据库条目以类似元组的对象形式报告，其属性对应于passwd结构的成员（下面的属性字段，请参阅<pwd.h>）：

指数	属性	含义
0	pw_name	登录名
1	pw_passwd	可选的加密密码
2	pw_uid	数字用户ID
3	pw_gid	数字组ID
4	pw_gecos	用户名或评论字段
五	pw_dir	用户主目录
6	pw_shell	用户命令解释器

uid和gid是整数，其他都是字符串。KeyError如果找不到要求的条目，则会提出。

注意： 在传统的Unix中，该字段pw_passwd通常包含用DES派生算法加密的密码（请参阅模块crypt）。然而大多数现代社会都使用所谓的*影子密码*系统。在这些 *unice中，pw_passwd* 字段只包含一个星号（'*'）或'x'加密密码存储在一个/etc/shadow不是全球可读的文件中的字母。*pw_passwd*字段是否包含有用的内容取决于系统。如果可用，spwd应在需要访问加密密码的地方使用该模块。

它定义了以下项目：

pwd.**getpwuid**(*uid*)
返回给定数字用户标识的密码数据库条目。

pwd.**getpwnam**(*名字*)
返回给定用户名的密码数据库条目。

pwd.**getpwall**()
以任意顺序返回所有可用密码数据库条目的列表。

也可以看看:

模 grp
组数据库的接口，与此类似。

模 spwd
影子密码数据库的界面，与此类似。

35.3。 spwd- 影子密码数据库

该模块提供对Unix影子密码数据库的访问。它可用于各种Unix版本。

您必须拥有足够的权限才能访问影子密码数据库（这通常意味着您必须是root用户）。

影子密码数据库条目被报告为元组对象，其属性对应于spwd结构的成员（下面的属性字段，请参阅<shadow.h>）：

指数	属性	含义
0	sp_namp	登录名
1	sp_pwdp	加密的密码
2	sp_lstchg	上次更改的日期
3	sp_min	变化之间的最短天数
4	sp_max	更改之间的最长天数
五	sp_warn	密码过期前的天数以警告用户
6	sp_inact	密码过期后直到帐户被禁用的天数
7	sp_expire	数1970-01-01至今帐户到期的天
8	sp_flag	保留的

sp_namp和sp_pwdp项目是字符串，其他所有项目都是整数。 KeyError如果找不到要求的条目，则会提出。

定义了以下功能：

spwd.getspnam（*名字*）
返回给定用户名的影子密码数据库条目。

*在版本3.6中更改：*引发，PermissionError而不是KeyError如果用户没有权限。

spwd.getspall（）
以任意顺序返回所有可用影子密码数据库条目的列表。

也可以看看:

模 grp
组数据库的接口，与此类似。

模 pwd
正常密码数据库的界面，与此类似。

35.4。 grp- 组数据库

该模块提供对Unix组数据库的访问。它适用于所有的Unix版本。

组数据库条目报告为元组对象，其属性对应于group结构的成员（ 下面的属性字段，请参阅<pwd.h>）：

指数	属性	含义
0	gr_name	组的名称
1	gr_passwd	（加密）组密码; 通常是空的
2	gr_gid	数字组ID
3	gr_mem	所有组成员的用户名

gid是一个整数，名称和密码是字符串，而成员列表是一个字符串列表。（ 请注意，根据密码数据库，大多数用户没有明确列出他们所在的组的成员，请检查两个数据库以获取完整的成员资格信息，还要注意以a gr_name开头+或-可能是YP / NIS参考并且可能无法通过getgrnam()或访问getgrgid()。 ）

它定义了以下项目：

grp.getgrgid（ *gid* ）
返回给定数字组ID的组数据库条目。KeyError 如果找不到要求的条目，则会提出。

*自从3.6版开始弃用：*自从Python 3.6以来，getgrgid()不赞成使用像float或string这样的非整数参数。

grp.getgrnam（ *名字* ）
返回给定组名称的组数据库条目。KeyError如果找不到要求的条目，则会提出。

grp.getgrall（ ）
以任意顺序返回所有可用组条目的列表。

也可以看看:

模 pwd
用户数据库的接口，与此类似。

模 spwd
影子密码数据库的界面，与此类似。

35.5。crypt- 检查Unix密码的功能

源代码： Lib / crypt.py

该模块实现*crypt（3）*例程的接口，该例程是基于修改的DES算法的单向散列函数; 有关更多详细信息，请参阅Unix手册页。可能的用途包括存储散列密码，以便您可以在不存储实际密码的情况下检查密码，或尝试使用字典破解Unix密码。

请注意，此模块的行为取决于正在运行的系统中*crypt（3）*例程的实际实现。因此，当前实现的任何可用扩展也将在此模块上提供。

35.5.1。散列方法

3.3版本的新功能

该crypt模块定义了散列方法列表（并非所有方法都可在所有平台上使用）：

crypt.METHOD_SHA512
一种模块化加密格式方法，包含16个字符的盐和86个字符的散列。这是最强的方法。

crypt.METHOD_SHA256
另一种模块化加密格式方法，包含16个字符的盐和43个字符的散列。

crypt.METHOD_MD5
另一种模块化加密格式方法，包含8个字符的盐和22个字符的散列。

crypt.METHOD_CRYPT
传统的方法有2个字符的盐和13个字符的散列。这是最弱的方法。

35.5.2。模块属性

3.3版本的新功能

crypt.methods
作为crypt.METHOD_*对象的可用密码哈希算法的列表 。该列表从最强到最弱排序。

35.5.3。模块功能

该crypt模块定义了以下功能：

crypt.crypt（*字*，*盐=无*）
*单词*通常是在提示或图形界面中键入的用户密码。可选的*salt*可以是从mksalt()其中一个crypt.METHOD_*值返回的字符串（尽管并非所有平台都可用），也可以是由该函数返回的完整加密密码，包括salt。如果没有提供*盐*，最强的方法将被使用（通过返回 methods()。

检查密码通常是通过将纯文本密码作为*单词*以及前一次crypt()调用的完整结果进行传递，该结果应该与此调用的结果相同。

盐（可能是2或16个字符的随机字符串，可能会加前缀 $digit$以指示方法），它将用于干扰加密算法。*salt中*的字符必须在集合中 [./a-zA-Z0-9]，但模块化加密格式（以前缀a为例）除外$digit$。

将散列密码作为字符串返回，该字符串将由与salt相同字母表中的字符组成。

由于一些*crypt（3）*扩展允许不同的值，*盐的*大小不同，建议在检查密码时使用完整的加密密码作为salt。

*在版本3.3中进行了更改：除盐的*crypt.METHOD_*字符串外，还接受值。

crypt.**mksalt**(*method* = *None*)

返回指定方法的随机生成的盐。如果没有 *方法*给出，如返回可用的最强方法 methods()被使用。

返回值是一个字符串，其长度为2个字符 crypt.METHOD_CRYPT，或者$digit$从该集合开始的19个字符和来自该集合的16个随机字符[./a-zA-Z0-9]，适合作为*salt*参数传递给crypt()。

3.3版本的新功能

35.5.4。示例

一个简单的例子说明了典型的使用情况（需要一个恒定时间的比较操作来限制对定时攻击的暴露， hmac.compare_digest()适用于这个目的）：

```
import pwd
import crypt
import getpass
from hmac import compare_digest as compare_hash

def login():
    username = input('Python login: ')
    cryptedpasswd = pwd.getpwnam(username)[1]
    if cryptedpasswd:
        if cryptedpasswd == 'x' or cryptedpasswd == '*':
            raise ValueError('no support for shadow passwords')
        cleartext = getpass.getpass()
        return compare_hash(crypt.crypt(cleartext, cryptedpasswd), cryptedpasswd)
    else:
        return True
```

要使用最强大的可用方法生成密码的哈希值并根据原始值进行检查：

```
import crypt
from hmac import compare_digest as compare_hash

hashed = crypt.crypt(plaintext)
```

```
if not compare_hash(hashed, crypt.crypt(plaintext, hashed)):
    raise ValueError("hashed version doesn't validate against original")
```

35.6。 termios- POSIX风格的tty控制

该模块提供了一个用于tty I / O控制的POSIX调用的接口。有关这些调用的完整说明，请参阅*termios（3）* Unix手册页。它仅适用于那些支持在安装期间配置的POSIX *termios*风格tty I / O控制的Unix版本 。

该模块中的所有函数都将文件描述符*fd*作为其第一个参数。这可以是整数文件描述符，例如返回的 `sys.stdin.fileno()` ，也可以是文件对象，如`sys.stdin`本身。

该模块还定义了使用此处提供的功能所需的所有常量; 这些名称与C中的对应名称相同。有关使用这些终端控制接口的更多信息，请参阅您的系统文档。

该模块定义了以下功能：

termios. **tcgetattr** (*fd*)
: 返回包含文件描述符*fd*的tty属性的列表，如下所示：其中，*cc*是tty特殊字符的列表（每个长度为1的字符串，除了具有索引的项目和定义这些字段时的整数）。标志和速度的解释以及*cc*数组中的索引必须使用模块中定义的符号常量来完成。[iflag, oflag, cflag, lflag, ispeed, ospeed, cc]VMINVTIMEtermios

termios. **tcsetattr** (*fd* , *when* , *attributes*)
: 设置文件描述符tty的属性*FD*从*属性*，这就好比通过返回一个列表tcgetattr()。在*当*参数确定时的属性被更改：TCSANOW立即改变，TCSADRAIN传输所有排队的输出之后改变，或者TCSAFLUSH传输所有排队的输出和废弃所有排队的输入后发生变化。

termios. **tcsendbreak** (*fd* , *持续时间*)
: 发送文件描述符*fd*中断。零*持续时间*发送一个中断0.25 -0.5秒; 非零*持续时间*具有系统依赖性意义。

termios. **tcdrain** (*fd*)
: 等到写入文件描述符*fd*的所有输出都被发送完毕。

termios. **tcflush** (*fd* , *队列*)
: 丢弃文件描述符*fd*上的排队数据。的*队列*选择器指定哪个队列：TCIFLUSH用于输入队列，TCOFLUSH用于输出队列，或TCIOFLUSH用于两个队列。

termios. **tcflow** (*fd* , *action*)
: 在文件描述符*fd*上挂起或恢复输入或输出。该*操作*参数可以是TCOOFF暂停输出，TCOON重启输出，TCIOFF暂停输入，或TCION重新启动输入。

也可以看看:

模 tty
: 常见终端控制操作的便捷功能。

35.6.1。示例

这里有一个函数，提示输入关闭回显的密码。请注意使用单独`tcgetattr()`调用和`try...finally`语句的技巧，以确保无论发生什么情况，都可以精确还原旧的tty属性：

```
def getpass(prompt="Password: "):
    import termios, sys
    fd = sys.stdin.fileno()
    old = termios.tcgetattr(fd)
    new = termios.tcgetattr(fd)
    new[3] = new[3] & ~termios.ECHO          # lflags
    try:
        termios.tcsetattr(fd, termios.TCSADRAIN, new)
        passwd = input(prompt)
    finally:
        termios.tcsetattr(fd, termios.TCSADRAIN, old)
    return passwd
```

35.7。 tty- 终端控制功能

源代码： Lib / tty.py

该tty模块定义了将tty置入cbreak和raw模式的函数。

因为它需要termios模块，所以它只能在Unix上工作。

该tty模块定义了以下功能：

tty. setraw (*fd* , *when = termios.TCSAFLUSH*)
将文件描述符*fd*的模式更改为raw。如果*何时*省略，则默认为termios.TCSAFLUSH，并传递给 termios.tcsetattr()。

tty. setcbreak (*fd* , *when = termios.TCSAFLUSH*)
将文件描述符*fd*的模式更改为cbreak。如果*何时*省略，则默认为termios.TCSAFLUSH，并传递给 termios.tcsetattr()。

也可以看看:

模 termios
低级终端控制界面。

35.8。pty- 伪终端实用程序

源代码： Lib / pty.py

该pty模块定义了处理伪终端概念的操作：启动另一个进程并能够以编程方式写入和读取其控制终端。

因为伪终端处理是高度依赖于平台的，所以有代码只对Linux执行。（Linux代码应该可以在其他平台上工作，但尚未经过测试。）

该pty模块定义了以下功能：

pty.**fork**（）
叉子。将孩子的控制终端连接到一个伪终端。返回值是。需要注意的是孩子得到*的*pid 0，*FD*是 *无效的*。父级的返回值是孩子的*PID*，*fd*是连接到孩子的控制终端（也是孩子的标准输入和输出）的文件描述符。(pid, fd)

pty.**openpty**（）
os.openpty()如果可能的话，打开一个新的伪终端对，或者用通用Unix系统的仿真代码。分别为主机和从机返回一对文件描述符 。(master, slave)

pty.**spawn**（*argv*[，*master_read*[，*stdin_read*]]）
产生一个进程，并将其控制终端与当前进程的标准io连接起来。这常常被用来挡住坚持从控制终端读取的程序。

函数*master_read*和*stdin_read*应该是从文件描述符中读取的函数。每次调用默认设置时都会尝试读取1024个字节。

*在版本3.4中更改：*spawn()现在从os.waitpid() 子进程返回状态值。

35.8.1。示例

下面的程序就像Unix命令*脚本（1）一样*，使用伪终端在“打字稿”中记录终端会话的所有输入和输出。

```
import argparse
import os
import pty
import sys
import time

parser = argparse.ArgumentParser()
parser.add_argument('-a', dest='append', action='store_true')
parser.add_argument('-p', dest='use_python', action='store_true')
parser.add_argument('filename', nargs='?', default='typescript')
options = parser.parse_args()

shell = sys.executable if options.use_python else os.environ.get('SHELL', 'sh')
```

```
filename = options.filename
mode = 'ab' if options.append else 'wb'

with open(filename, mode) as script:
    def read(fd):
        data = os.read(fd, 1024)
        script.write(data)
        return data

    print('Script started, file is', filename)
    script.write(('Script started on %s\n' % time.asctime()).encode())

    pty.spawn(shell, read)

    script.write(('Script done on %s\n' % time.asctime()).encode())
    print('Script done, file is', filename)
```

35.9。fcntl- fcntl和ioctl系统调用

该模块对文件描述符执行文件控制和I / O控制。它是fcntl()和ioctl()Unix例程的接口。有关这些调用的完整说明，请参阅*fcntl（2）*和 *ioctl（2）* Unix手册页。

该模块中的所有函数都将文件描述符*fd*作为其第一个参数。这可以是整数文件描述符（如返回的） sys.stdin.fileno()，也可以是一个io.IOBase对象（如sys.stdin 本身），它提供fileno()返回真实文件描述符的对象。

*在版本3.3中进行了更改：*此模块中的操作用于引发IOError他们现在引发的操作OSError。

该模块定义了以下功能：

fcntl.**fcntl**（*fd*，*cmd*，*arg = 0* ）

对文件描述符*fd*执行操作*cmd*（提供方法的文件对象也被接受）。用于*cmd*的值取决于操作系统，并可在模块中作为常量使用，使用与相关C头文件中使用的名称相同的名称。参数*arg*可以是整数值，也可以是 对象。对于整数值，此函数的返回值是C 调用的整数返回值。当参数是字节时，它表示一个二进制结构，例如由其创建 。二进制数据被复制到地址传递给C的缓冲区中fileno() fcntl bytes fcntl() struct.pack() fcntl()呼叫。成功调用后的返回值是缓冲区的内容，转换为bytes对象。返回的对象的长度将与*arg*参数的长度相同 。这限制为1024字节。如果操作系统在缓冲区中返回的信息大于1024字节，这很可能导致分段违规或更微妙的数据损坏。

如果fcntl()失败，OSError则提出。

fcntl.**ioctl**（*fd*，*request*，*arg = 0*，*mutate_flag = True* ）

该函数与函数相同fcntl()，只是参数处理更复杂。

所述*请求*参数被限制为可以容纳32位的值。用作*请求*参数的其他常量可以在termios模块中找到，其名称与相关C头文件中使用的名称相同。

参数*arg*可以是整数，支持只读缓冲区接口bytes的对象（如）或支持读写缓冲区接口（如bytearray）的对象之一。

除最后一种情况外，行为与fcntl() 功能一样。

如果传递了一个可变缓冲区，则行为由*mutate_flag*参数的值决定。

如果它是错误的，那么缓冲区的可变性将被忽略，行为与只读缓冲区相同，只是上面提到的1024字节限制是可以避免的 - 只要你传递的缓冲区至少和操作系统想要的一样长放在那里，事情应该工作。

如果*mutate_flag*为true（默认值），则缓冲区（实际上）传递给底层的ioctl()系统调用，后者的返回代码被传回给调用的Python，并且缓冲区的新内容反映了该动作ioctl()。这只是一个小小的简化，因为如果提供的缓冲区长度小于1024字节，它首先被复制到一个1024字节长的静态缓冲区，然后被传递ioctl()并复制回到提供的缓冲区。

如果ioctl()失败，OSError则会引发异常。

一个例子：

```
>>> import array, fcntl, struct, termios, os
>>> os.getpgrp()
13341
>>> struct.unpack('h', fcntl.ioctl(0, termios.TIOCGPGRP, "  "))[0]
13341
>>> buf = array.array('h', [0])
>>> fcntl.ioctl(0, termios.TIOCGPGRP, buf, 1)
0
>>> buf
array('h', [13341])
```

fcntl.**flock**(*fd*, *操作*)

对文件描述符*fd*执行锁定操作*操作*（提供方法的文件对象也被接受）。有关详细信息，请参见Unix手册 *（2）*。（在某些系统上，此功能是使用模拟的。）`fileno()` `fcntl()`

如果`flock()`失败，`OSError`则会引发异常。

fcntl.**lockf**(*fd*, *cmd*, *len = 0*, *start = 0*, *whence = 0*)

这本质上是`fcntl()`锁定调用的一个包装。 *fd*是要锁定或解锁的文件的文件描述符，*cmd*是以下值之一：

- LOCK_UN - 开锁
- LOCK_SH - 获得共享锁
- LOCK_EX - 获得独家锁定

当*cmd*为LOCK_SH或时LOCK_EX，它也可以按位与或，LOCK_NB以避免在锁定采集时被阻塞。如果LOCK_NB使用并且无法获取锁， `OSError`则会引发一个异常，并且该异常的*errno* 属性将设置为EACCES或EAGAIN（取决于操作系统;对于可移植性，请检查这两个值）。至少在某些系统上，LOCK_EX只能在文件描述符引用为写入而打开的文件时才能使用。

*LEN*是字节锁定的数目，*开始*是在其中锁启动时，相对于该字节偏移*何处*，并*从那里*是与`io.IOBase.seek()`，特别是：

- 0- 相对于文件的开始（`os.SEEK_SET`）
- 1- 相对于当前缓冲区的位置（`os.SEEK_CUR`）
- 2- 相对于文件的结尾（`os.SEEK_END`）

*开始*的默认值是0，这意味着从文件的开头开始。*len*的默认值是0，这意味着锁定到文件的末尾。*whence*的默认值也是0。

示例（全部在SVR4兼容系统上）：

```
import struct, fcntl, os

f = open(...)
rv = fcntl.fcntl(f, fcntl.F_SETFL, os.O_NDELAY)

lockdata = struct.pack('hhllhh', fcntl.F_WRLCK, 0, 0, 0, 0, 0)
rv = fcntl.fcntl(f, fcntl.F_SETLKW, lockdata)
```

请注意，在第一个示例中，返回值变量*rv*将保存一个整数值; 在第二个例子中它将保存一个`bytes`对象。*lockdata*变量的结构布局是依赖于系统的 - 因此使用这个`flock()`调用可能会更好。

也可以看看:

模 `os`
: 如果锁定标志`O_SHLOCK`和模块`O_EXLOCK`中存在`os`（仅在BSD上），该`os.open()`功能提供了`lockf()`和`flock()`功能的替代方法。

35.10。pipes- 与shell管道的接口

源代码： Lib / pipes.py

该pipes模块定义了一个类来抽象*管道*的概念 - 从一个文件到另一个文件的一系列转换器。

因为模块使用/ **bin / sh的**命令线，用于POSIX或兼容壳os.system()和os.popen()是必需的。

该pipes模块定义了以下类：

*类*pipes.Template
管道的抽象。

例：

```
>>> import pipes
>>> t = pipes.Template()
>>> t.append('tr a-z A-Z', '--')
>>> f = t.open('pipefile', 'w')
>>> f.write('hello world')
>>> f.close()
>>> open('pipefile').read()
'HELLO WORLD'
```

35.10.1。模板对象

以下方法的模板对象：

Template.reset（）
将管道模板恢复到其初始状态。

Template.clone（）
返回一个新的等效管道模板。

Template.debug（*标志*）
如果*标志*为真，则打开调试。否则，关闭调试。打开调试时，将打印要执行的命令，并给shell 命令更详细。set -x

Template.append（*cmd*，*kind*）
在最后附加一个新动作。在*CMD*变量必须是一个有效的Bourne shell命令。该*种*变量由两个字母组成。

第一个字母可以是'-'（这意味着命令读取其标准输入）'f'（这意味着命令读取命令行上的给定文件）或'.'（这意味着命令不读取任何输入，因此必须是第一个）。

类似地，第二个字母可以是'-'（这意味着命令写入标准输出）'f'（这意味着命令在命令行上写入文件）或'.'（这意味着该命令不会写入任何内容，因此必须是最后一个）。 ）

Template.prepend(*cmd* , *kind*)

在开始时添加一个新的动作。请参阅有关append()论据的解释。

Template.open(*文件* , *模式*)

返回一个类似文件的对象，打开*文件*，但是读取或写入管道。请注意，只有一个'r'，'w'可能会被给出。

Template.copy(*infile* , *outfile*)

通过管道将*infile*复制到*outfile*。

35.11。resource- 资源使用信息

该模块提供了测量和控制程序使用的系统资源的基本机制。

符号常量用于指定特定系统资源并请求有关当前进程或其子进程的使用信息。

一个OSError上引发的系统调用失败。

*异常*resource. error
不推荐使用的别名OSError。

*版本3.3中更改：*以下**PEP 3151**，这个班被**取而代之**OSError。

35.11.1。资源限制

使用setrlimit()下面描述的功能可以限制资源使用情况。每个资源都由一对限制控制：软限制和硬限制。软限制是电流限制，可能会随着时间的推移而降低或提高。软限制永远不会超过硬限制。硬限制可以降低到大于软限制的任何值，但不会升高。（只有拥有超级用户的有效UID的进程才能提高硬性限制。）

可以限制的具体资源依赖于系统。它们在*getrlimit（2）*手册页中进行了描述。下面列出的资源在底层操作系统支持时支持; 对于那些平台，在这个模块中没有定义操作系统无法检查或控制的资源。

resource. RLIM_INFINITY
常量用于表示无限资源的限制。

resource. getrlimit（*资源*）
返回具有当前软*资源*和硬*资源*限制的 元组。如果指定了无效资源，或者底层系统调用意外失败，则会引发。(soft, hard)ValueErrorerror

resource. setrlimit（*资源*，*限制*）
设定*资源*消耗的新限制。该*限制*参数必须是一个元组两个整数描述的新限制的。值 可以用来请求无限制的限制。(soft, hard)RLIM_INFINITY

引发ValueError，如果指定了无效的资源，如果新的软限制超过硬限制，或者如果一个进程试图提高其硬限制。指定RLIM_INFINITY该资源的硬限制或系统限制不是无限制时的限制将导致a ValueError。具有超级用户的有效UID的进程可以请求任何有效的限制值，包括无限制，但是ValueError 如果请求的限制超过系统限制，仍然会提高。

setrlimiterror如果底层的系统调用失败，也可能会引发。

resource. prlimit（*pid*，*resource*[，*limits*]）
组合setrlimit()并getrlimit()在一个功能中支持获取和设置任意进程的资源限制。如果*pid*为0，则该调用适用于当前进程。*资源*和 *限制*的含义与中相同setrlimit()，但 *限制*是可选的。

当*限制*没有给出函数返回的*资源*的过程中限制*PID*。当*限制*给出*的资源*的过程中限制设置并返回前资源限制。

ProcessLookupError当无法找到 *pid*PermissionError时以及用户没有CAP_SYS_RESOURCE进行该过程时引发。

可用性：使用glibc 2.13或更高版本的Linux 2.6.36或更高版本

3.4版新增功能

这些符号定义其消费可以使用控制资源 setrlimit()和getrlimit()功能说明如下。这些符号的值恰好是C程序使用的常量。

*getrlimit（2）*的Unix手册页列出了可用资源。请注意，并非所有系统都使用相同的符号或相同的值来表示相同的资源。该模块不会尝试屏蔽平台差异 - 在该平台上，该模块将无法使用未为平台定义的符号。

resource.RLIMIT_CORE
当前进程可以创建的核心文件的最大大小（以字节为单位）。如果需要更大的内核来包含整个过程映像，则可能会导致创建部分内核文件。

resource.RLIMIT_CPU
处理器可以使用的最大处理器时间量（以秒为单位）。如果超过此限制，SIGXCPU则会向该过程发送信号。（ signal有关如何捕获此信号并执行一些有用的操作的信息，请参阅模块文档，例如，将打开的文件刷新到磁盘。）

resource.RLIMIT_FSIZE
进程可能创建的文件的最大大小。

resource.RLIMIT_DATA
进程堆的最大大小（以字节为单位）。

resource.RLIMIT_STACK
当前进程调用堆栈的最大大小（以字节为单位）。这只会影响多线程进程中主线程的堆栈。

resource.RLIMIT_RSS
应该提供给流程的最大驻留集大小。

resource.RLIMIT_NPROC
当前进程可能创建的最大进程数。

resource.RLIMIT_NOFILE
当前进程的最大打开文件描述符数。

resource.RLIMIT_OFILE
BSD的名字RLIMIT_NOFILE。

resource.RLIMIT_MEMLOCK

可能被锁定在内存中的最大地址空间。

resource.RLIMIT_VMEM
该进程可能占用的最大映射内存区域。

resource.RLIMIT_AS
进程可能占用的地址空间的最大区域（以字节为单位）。

resource.RLIMIT_MSGQUEUE
可以分配给POSIX消息队列的字节数。

可用性：Linux 2.6.8或更高版本。

3.4版新增功能

resource.RLIMIT_NICE
这个过程的最高限额（以20 - rlim_cur计算）的上限。

可用性：Linux 2.6.12或更高版本。

3.4版新增功能

resource.RLIMIT_RTPRIO
实时优先级的上限。

可用性：Linux 2.6.12或更高版本。

3.4版新增功能

resource.RLIMIT_RTTIME
一个进程可以在实时调度下花费CPU时间的时间限制（以微秒为单位），而不会产生阻塞系统调用。

可用性：Linux 2.6.25或更高版本。

3.4版新增功能

resource.RLIMIT_SIGPENDING
进程可能排队的信号数量。

可用性：Linux 2.6.8或更高版本。

3.4版新增功能

resource.RLIMIT_SBSIZE
此用户的套接字缓冲区使用的最大大小（以字节为单位）。这限制了该用户随时可以保留的网络内存量，从而限制了mbuf的数量。

可用性：FreeBSD 9或更高版本。

3.4版新增功能

resource.RLIMIT_SWAP

交换空间的最大大小（以字节为单位），可由该用户标识的所有进程保留或使用。仅当vm.overcommit sysctl的位1置位时才会强制执行此限制。请参阅*调整（7）*以获得对此sysctl的完整描述。

可用性：FreeBSD 9或更高版本。

3.4版新增功能

resource.RLIMIT_NPTS

由此用户标识创建的伪终端的最大数量。

可用性：FreeBSD 9或更高版本。

3.4版新增功能

35.11.2。资源使用

这些函数用于检索资源使用信息：

resource.getrusage（*谁*）

此函数返回一个对象，该对象描述由*who*参数指定的当前进程或其子进程使用的资源。该*谁*参数应使用的一个指定RUSAGE_*下面介绍的常量。

每个返回值的字段都描述了特定系统资源的使用方式，例如，运行时间是用户模式或进程从主内存换出的次数。某些值取决于内部时钟节拍，例如过程正在使用的内存量。

为了向后兼容，返回值也可以作为16个元素的元组访问。

字段ru_utime和ru_stime返回值是浮点值，分别代表在用户模式下执行的时间量和在系统模式下执行的时间量。其余的值是整数。有关这些值的详细信息，请参阅*getrusage（2）*手册页。这里介绍一个简短的总结：

指数	领域	资源
0	ru_utime	用户模式下的时间（浮点数）
1	ru_stime	系统模式下的时间（浮点数）
2	ru_maxrss	最大驻地套装尺寸
3	ru_ixrss	共享内存大小
4	ru_idrss	未共享的内存大小
5	ru_isrss	未共享堆栈大小
6	ru_minflt	不需要I / O的页面错误
7	ru_majflt	需要I / O的页面错误
8	ru_nswap	换出次数

指数	领域	资源
9	ru_inblock	阻止输入操作
10	ru_oublock	块输出操作
11	ru_msgsnd	发送的消息
12	ru_msgrcv	收到的消息
13	ru_nsignals	收到的信号
14	ru_nvcsw	自愿上下文切换
15	ru_nivcsw	非自愿的上下文切换

ValueError如果指定了无效*who*参数，此函数将引发一次。 error在特殊情况下它也可能引发异常。

resource.getpagesize ( )

返回系统页面中的字节数。（这不必与硬件页面大小相同。）

以下RUSAGE_*符号被传递给该getrusage() 函数以指定应提供哪些处理信息。

resource.RUSAGE_SELF

传递以getrusage()请求调用进程占用的资源，这是进程中所有线程使用的资源的总和。

resource.RUSAGE_CHILDREN

传递以getrusage()请求已被终止和等待的调用进程的子进程占用的资源。

resource.RUSAGE_BOTH

传递给getrusage()请求当前进程和子进程消耗的资源。可能不适用于所有系统。

resource.RUSAGE_THREAD

传递给getrusage()请求当前线程使用的资源。可能不适用于所有系统。

3.2版本中的新功能

35.12。nis- Sun的NIS接口（黄页）

该nis模块为NIS库提供了一个简洁的包装器，可用于多个主机的集中管理。

由于NIS仅存在于Unix系统上，因此该模块仅适用于Unix。

该nis模块定义了以下功能：

nis.match（*key*，*mapname*，*domain = default_domain* ）
返回地图地图*名称*中的*键*的匹配*项*，如果没有则返回错误（ ）。两者都应该是字符串，*关键*是8位清理。返回值是任意字节数组（可能包含其他快乐）。nis.errorNULL

请注意，如果*mapname*是另一个名称的别名，则首先检查它。

该*域*参数允许覆盖用于查找NIS域。如果未指定，则查找位于默认的NIS域中。

nis.cat（*mapname*，*domain = default_domain* ）
返回一个字典映射*键*来*珍惜*这样。请注意，字典的键和值都是任意字节数组。match(key, mapname)==value

请注意，如果*mapname*是另一个名称的别名，则首先检查它。

该*域*参数允许覆盖用于查找NIS域。如果未指定，则查找位于默认的NIS域中。

nis.maps（*domain = default_domain* ）
返回所有有效地图的列表。

该*域*参数允许覆盖用于查找NIS域。如果未指定，则查找位于默认的NIS域中。

nis.get_default_domain（ ）
返回系统默认的NIS域。

该nis模块定义了以下例外情况：

*异常*nis.error
NIS函数返回错误代码时引发错误。

35.13。 syslog- Unix系统日志库例程

该模块为Unix syslog库例程提供了一个接口。请参阅Unix手册页以获取该syslog 设施的详细说明。

该模块包装系统syslog系列的例程。可以与系统日志服务器对话的纯Python库在logging.handlers模块中可用 SysLogHandler。

该模块定义了以下功能：

syslog.**syslog**（*消息*）

syslog.**syslog**（*优先*，*消息*）

将字符串*消息发送*到系统记录器。如有必要，可以添加尾随的换行符。每条消息都标有由*设施*和*级别*组成的优先 *级*。可选的*优先级*参数默认为LOG_INFO确定消息的优先级。如果设施未使用逻辑或（）进行*优先*编码，则使用该调用中给出的值。 LOG_INFO | LOG_USER openlog()

如果openlog()在调用之前没有被调用syslog()， openlog()将会被调用而没有参数。

syslog.**openlog**（[*ident*[，*logoption*[，*facility*]]]）

后续syslog()调用的记录选项可通过调用进行设置 openlog()。 如果日志当前未打开，syslog()将会调用openlog()不带参数。

可选的*ident*关键字参数是一个字符串，它被添加到每条消息中，并且默认sys.argv[0]使用剥离的前导路径组件。可选的*logoption*关键字参数（默认值为0）是一个位字段 - 请参阅下面的可能值进行组合。可选的*facility* 关键字参数（默认为LOG_USER）设置没有显式编码设施的消息的默认设施。

*在版本3.2中更改：*在以前的版本中，不允许使用关键字参数，并且*ident*是必需的。*ident*的默认值取决于系统库，通常python不是python程序文件的名称。

syslog.**closelog**（）

重置系统日志模块值并调用系统库closelog()。

这会导致模块的行为与最初导入时的行为相同。例如，openlog()将在第一次syslog()调用时调用（如果 openlog()尚未调用），并将*ident*和其他 openlog()参数重置为默认值。

syslog.**setlogmask**（*maskpri*）

将优先级掩码设置为*maskpri*并返回以前的掩码值。调用syslog()优先级不在*maskpri*中设置*将*被忽略。缺省是记录所有优先级。该函数LOG_MASK(pri) 为各个优先级*pri*计算掩码。该函数 LOG_UPTO(pri)为直到并包括*pri的*所有优先级计算掩码 。

该模块定义了以下常量：

优先级别（从高到低）：

LOG_EMERG，LOG_ALERT，LOG_CRIT，LOG_ERR， LOG_WARNING，LOG_NOTICE，LOG_INFO，LOG_DEBUG。

设备：

LOG_KERN ， LOG_USER ， LOG_MAIL ， LOG_DAEMON ， LOG_AUTH ， LOG_LPR ， LOG_NEWS ， LOG_UUCP ， LOG_CRON ，LOG_SYSLOG ，LOG_LOCAL0 向 LOG_LOCAL7 ，并且，如果所定义的 <syslog.h>， LOG_AUTHPRIV。

日志选项：

LOG_PID ， LOG_CONS ， LOG_NDELAY ，和 ，如果在定义的 <syslog.h> ， LOG_ODELAY ， LOG_NOWAIT ，和 LOG_PERROR。

35.13.1。示例

35.13.1.1。简单的例子

一组简单的例子：

```
import syslog

syslog.syslog('Processing started')
if error:
    syslog.syslog(syslog.LOG_ERR, 'Processing started')
```

设置某些日志选项的示例，这些将包括日志消息中的进程ID，并将消息写入用于邮件日志记录的目标设施：

```
syslog.openlog(logoption=syslog.LOG_PID, facility=syslog.LOG_MAIL)
syslog.syslog('E-mail processing initiated...')
```

36.被取代的模块

本章介绍的模块已被弃用，仅保留用于向后兼容。他们已被其他模块取代。

36.1。optparse- 用于命令行选项的解析器

源代码： Lib / optparse.py

*自从3.2版本不推荐使用：*该optparse模块已被弃用，也不会得到进一步发展; 发展将继续与argparse模块。

optparse是比旧getopt模块更方便，灵活且功能强大的库，用于解析命令行选项。 optparse使用更具说明性的命令行解析风格：您创建一个实例 OptionParser，使用选项填充它，并解析命令行。optparse允许用户使用传统的GNU / POSIX语法指定选项，并且还为您生成使用情况和帮助信息。

以下是optparse一个简单脚本中使用的示例：

```
from optparse import OptionParser
...
parser = OptionParser()
parser.add_option("-f", "--file", dest="filename",
                  help="write report to FILE", metavar="FILE")
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose", default=True,
                  help="don't print status messages to stdout")

(options, args) = parser.parse_args()
```

使用这几行代码，您的脚本的用户现在可以在命令行上执行“通常的事情”，例如：

```
<yourscript> --file=outfile -q
```

在分析命令行时，根据用户提供的命令行值optparse设置options返回的对象的属性parse_args()。当parse_args()从解析此命令行返回，options.filename将"outfile"和options.verbose会False。 optparse支持长期和短期期权，允许短期期权合并在一起，并允许期权以各种方式与其参数相关联。因此，以下命令行全部等同于上述示例：

```
<yourscript> -f outfile --quiet
<yourscript> --quiet --file outfile
<yourscript> -q -foutfile
<yourscript> -qfoutfile
```

另外，用户可以运行其中的一个

```
<yourscript> -h
<yourscript> --help
```

并optparse会打印出脚本选项的简要摘要：

```
Usage: <yourscript> [options]

Options:
```

```
-h, --help            show this help message and exit
-f FILE, --file=FILE  write report to FILE
-q, --quiet           don't print status messages to stdout
```

*yourscript*的值在运行时确定（通常来自 `sys.argv[0]`）。

36.1.1。背景

`optparse`明确地设计用于鼓励使用直接的常规命令行界面创建程序。为此，它只支持Unix下常用的最常用的命令行语法和语义。如果您不熟悉这些惯例，请阅读本节以熟悉它们。

36.1.1.1。术语

论据

在命令行上输入的字符串，并通过shell传递给`execl()` 或`execv()`。在Python中，参数是元素`sys.argv[1:]` （ `sys.argv[0]`是正在执行的程序的名称）。Unix shell也使用术语“单词”。

偶尔需要用一个参数列表来代替 `sys.argv[1:]`，所以你应该将“参数”理解为“ `sys.argv[1:]` 作为替代的一个元素 或其他列表 `sys.argv[1:]`”。

选项

用于提供额外信息来引导或定制程序执行的参数。选项有许多不同的语法; 传统的Unix语法是一个连字符（“ - ”），后跟单个字母，例如`-x`或`-F`。另外，传统的Unix语法允许将多个选项合并为一个参数，例如等同于。引入的GNU项目后跟一系列连字符分隔的单词，例如或。这是唯一提供的两种选项语法。`-x -F-xF----file--dry-runoptparse`

世界已经看到的一些其他选项语法包括：

- 一个连字符后面几个字母，例如`-pf`（这是*不*一样的多个选项合并为一个参数）
- 一个连字符后跟一个单词，例如`-file`（这在技术上等同于前面的语法，但它们通常不会在同一个程序中看到）
- 一个加号后跟单个字母，或几个字母或一个单词，例如 `+f`，`+rgb`
- 斜杠后跟一个字母，或几个字母，或一个字，例如`/f`， `/file`

这些选项语法不受支持`optparse`，并且它们永远不会。这是故意的：前三个在任何环境下都是非标准的，如果你专门针对VMS，MS-DOS和/或Windows，最后一个是唯一有意义的。

选项参数

一个选项后面的参数与该选项密切相关，并且在该选项时从参数列表中消耗。与`optparse`，选项参数可能会在他们的选项单独的参数：

```
-f foo
--file foo
```

或者包含在相同的论点中：

```
-ffoo
--file=foo
```

通常情况下，给定的选项需要参数或不参与。很多人都希望有一个“可选的选项参数”功能，这意味着如果他们看到它，某些选项将会有一个参数，如果它们不参与则不会。这是有点争议的，因为它使解析模糊：如果-a采取可选参数，并且-b完全是另一种选择，我们如何解释-ab？由于这种模糊性，optparse不支持此功能。

位置论证
在解析了选项后，即在选项及其参数已被解析并从参数列表中删除之后，在参数列表中剩余的东西。

必需的选项
一个必须在命令行上提供的选项; 请注意，“必需的选项”一词在英语中是自相矛盾的。optparse并不妨碍你实现所需的选项，但也不会给你太多的帮助。

例如，考虑这个假设的命令行：

```
prog -v --report report.txt foo bar
```

-v并且--report都是选择。假设只有--report 一个参数，report.txt是一个选项参数。 foo并且 bar是位置参数。

36.1.1.2。有什么选择？

选项用于提供额外的信息来调整或定制程序的执行。如果不清楚，选项通常是*可选的*。一个程序应该能够运行得很好，没有任何选择。（选择从Unix或GNU工具集的随机程序，它可以运行，而完全不带任何选项，仍然有意义吗？主要的例外是find，tar和 dd-所有这些都已经正确地批评自己的非标准语法突变oddballs和令人困惑的界面。）

很多人希望他们的计划有“必需的选择”。想想看。如果它是必需的，那么它*不是可选的*！如果你的程序为了成功运行而需要绝对的信息，那就是位置参数。

作为良好的命令行界面设计的例子，考虑cp 复制文件的谦虚工具。尝试复制文件时没有提供目的地和至少一个源文件没有多大意义。因此，cp如果你没有参数运行它就会失败。但是，它有一个灵活有用的语法，根本不需要任何选项：

```
cp SOURCE DEST
cp SOURCE ... DEST-DIR
```

你可以得到相当远的那一点。大多数cp实现提供了一系列选项来精确调整文件的复制方式：您可以保留模式和修改时间，避免遵循符号链接，在重新打开现有文件之前询问等等。但是这些都不会干扰核心任务 cp，即将一个文件复制到另一个文件，或将多个文件复制到另一个目录

36.1.1.3。什么是位置参数？

位置参数适用于您的程序完全需要运行的那些信息。

一个好的用户界面应该尽可能少的绝对要求。如果你的程序需要17个不同的信息，以成功运行，它没有多大关系*如何*你从用户大多数人的信息会放弃走开他们成功地运行该程序之前。无论用户界面是命令行，配置文件还是GUI：这都适用：如果您对用户提出了许多要求，他们中的大多数都会放弃。

简而言之，尽量减少用户绝对需要提供的信息量 - 尽可能使用合理的默认值。当然，你也想让你的程序合理灵活。这是什么选择。同样，无论它们是配置文件中的条目，GUI的“首选项”对话框中的小部件还是命令行选项，您实现的选项越多，程序的灵活性越高，以及更复杂其实施成为。当然，太多的灵活性也有缺点。太多的选项可能压倒用户，使你的代码难以维护。

36.1.2。教程

虽然optparse相当灵活且功能强大，但在大多数情况下使用也很简单。本节介绍任何optparse基于程序的通用代码模式。

首先，您需要导入OptionParser类; 然后，在主程序的早期，创建一个OptionParser实例：

```
from optparse import OptionParser
...
parser = OptionParser()
```

然后你可以开始定义选项。基本的语法是：

```
parser.add_option(opt_str, ...,
                  attr=value, ...)
```

每个选项都有一个或多个选项字符串（例如-f或）--file，以及几个选项属性，它们告诉optparse我们期望什么，以及在命令行中遇到该选项时该做什么。

通常，每个选项将有一个短选项字符串和一个长选项字符串，例如：

```
parser.add_option("-f", "--file", ...)
```

只要至少有一个选项字符串，您可以自由定义尽可能多的短选项字符串和尽可能多的长选项字符串（包括零）。

传递给的选项字符串OptionParser.add_option()对于由该调用定义的选项是有效的标签。为简洁起见，我们经常会提到 在命令行上*遇到一个选项*; 在现实中，optparse 遇到*选项字符串*并从中查找选项。

一旦定义了所有选项，optparse就要指示解析程序的命令行：

```
(options, args) = parser.parse_args()
```

（如果你喜欢，你可以传递一个自定义参数列表parse_args()，但这很少有必要：默认情况下它使用sys.argv[1:]。）

parse_args() 返回两个值：

- options，一个包含所有选项值的对象 - 例如，如果 --file使用单个字符串参数，options.file则将是由用户提供的文件名，或者None如果用户没有提供该选项
- args，解析选项后剩余的位置参数列表

本教程部分仅涉及四个最重要的选项属性： action，type，dest （目标），和help。其中，action最根本的是。

36.1.2.1。了解选项操作

optparse当命令行遇到选项时，操作会告诉您该做什么。有一组固定的操作被硬编码到optparse;添加新操作是扩展optparse一节中介绍的高级主题 。大多数操作都告诉optparse将某个值存储在某个变量中 - 例如，从命令行取一个字符串并将其存储在一个属性中options。

如果您未指定选项操作，则optparse默认为store。

36.1.2.2。商店行为

最常见的选项动作是store，它告诉optparse采取下一个参数（或当前参数的其余部分），确保它是正确的类型，并将其存储到您选择的目的地。

例如：

```
parser.add_option("-f", "--file",
                  action="store", type="string", dest="filename")
```

现在让我们编一个假的命令行并要求optparse解析它：

```
args = ["-f", "foo.txt"]
(options, args) = parser.parse_args(args)
```

当optparse看到选项字符串时-f，它会消耗下一个参数foo.txt，并存储它options.filename。所以，在此之后parse_args()，options.filename是"foo.txt"。

受支持的其他一些选项类型optparse是int和float。这是一个期望有一个整数参数的选项：

```
parser.add_option("-n", type="int", dest="num")
```

请注意，此选项没有长选项字符串，这是完全可以接受的。此外，没有明确的行动，因为默认是store。

我们来解析另一个虚假的命令行。这一次，我们会将选项参数与选项对齐：因为-n42（一个参数）等同于（两个参数），所以代码-n 42

```
(options, args) = parser.parse_args(["-n42"])
print(options.num)
```

将打印42。

如果您不指定类型，则optparse假定string。结合默认动作这一事实store，这意味着我们的第一个示例可以缩短很多：

```
parser.add_option("-f", "--file", dest="filename")
```

如果您不提供目的地，则optparse根据选项字符串计算合理的默认值：如果第一个长选项字符串是 --foo-bar，则默认目的地是foo_bar。如果没有长选项字符串，请optparse查看第一个短选项字符串：-fis 的默认目的地f。

optparse还包括内置的complex类型。扩展optparse一节介绍了添加类型。

36.1.2.3。处理布尔（标志）选项

标志选项 - 当一个特定的选项被看到时，将一个变量设置为真或假 - 很常见。 optparse通过两个独立的行动支持他们， store_true并且store_false。例如，您可能有一个verbose 打开-v和关闭的标志-q：

```
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose")
```

在这里，我们有两个不同的选项与相同的目的地，这是完全正确的。（这只是意味着您在设置默认值时必须小心 - 请参阅下文。）

当optparse 遇到-v 命令行时，它设置 options.verbose 为 True; 当它遇到时-q ，options.verbose被设置为False。

36.1.2.4。其他行动

其他一些支持的操作optparse是：

"store_const"
    存储一个常数值

"append"
    将此选项的参数附加到列表中

"count"
    增加一个计数器

"callback"
    调用指定的功能

这些在参考指南，参考指南和选项回调部分中有介绍。

36.1.2.5。默认值

以上所有示例都涉及在某些命令行选项可见时设置某个变量（“目标”）。如果这些选项从未见过会发生什么？由于我们没有提供任何默认值，它们都设置为None。这通常很好，但有时你想要更多的控制。 optparse可以为每个目标提供一个默认值，该值在解析命令行之前分配。

首先，考虑冗长/安静的例子。如果我们想要optparse设置 verbose为True除非-q被看到，那么我们可以这样做：

```
parser.add_option("-v", action="store_true", dest="verbose", default=True)
parser.add_option("-q", action="store_false", dest="verbose")
```

由于默认值适用于*目标*而不是任何特定的选项，并且这两个选项碰巧具有相同的目标，所以这完全等同：

```
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose", default=True)
```

考虑这个：

```
parser.add_option("-v", action="store_true", dest="verbose", default=False)
parser.add_option("-q", action="store_false", dest="verbose", default=True)
```

同样，缺省值verbose将为True：为任何特定目标提供的最后一个缺省值是重要的。

指定默认值的更清晰的set_defaults()方法是OptionParser 的方法，您可以在调用之前随时调用该方法parse_args()：

```
parser.set_defaults(verbose=True)
parser.add_option(...)
(options, args) = parser.parse_args()
```

和以前一样，为给定选项目标指定的最后一个值是计数的值。为了清楚起见，请尝试使用一种方法或另一种设置默认值，而不是两种。

36.1.2.6。生成帮助

optparse自动生成帮助和使用文本的能力对于创建用户友好的命令行界面非常有用。您所要做的就是help为每个选项提供一个值，并为整个程序提供一个简短的用法信息。这里有一个OptionParser，它提供了用户友好（记录）的选项：

```
usage = "usage: %prog [options] arg1 arg2"
parser = OptionParser(usage=usage)
parser.add_option("-v", "--verbose",
                  action="store_true", dest="verbose", default=True,
                  help="make lots of noise [default]")
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose",
                  help="be vewwy quiet (I'm hunting wabbits)")
parser.add_option("-f", "--filename",
                  metavar="FILE", help="write output to FILE")
parser.add_option("-m", "--mode",
                  default="intermediate",
                  help="interaction mode: novice, intermediate, "
                       "or expert [default: %default]")
```

如果在命令行上或者在命令行上optparse遇到，或者只是调用，则会将以下内容输出到标准输出：-h--helpparser.print_help()

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose         make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
```

```
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or
                        expert [default: intermediate]
```

（如果帮助输出由帮助选项触发，则optparse在打印帮助文本后退出。）

这里有很多事情可以帮助您optparse产生最好的帮助信息：

- 该脚本定义了它自己的使用信息：

  ```
  usage = "usage: %prog [options] arg1 arg2"
  ```

  optparse 将 %prog 用法字符串扩展为当前程序的名称，即 os.path.basename(sys.argv[0])。然后在详细的选项帮助之前打印展开的字符串。

  如果你不提供一个用法字符串，optparse使用一个平淡但合理的默认值，如果你的脚本没有使用任何位置参数，这很好。"Usage: %prog [options]"

- 每个选项都定义了一个帮助字符串，并且不用担心换行问题 - optparse照顾换行和帮助输出看起来不错。

- 带值的选项在自动生成的帮助消息中指示了这一事实，例如“模式”选项：

  ```
  -m MODE, --mode=MODE
  ```

  这里，“MODE”被称为元变量：它代表用户预期提供给-m/的参数--mode。默认情况下，optparse将目标变量名称转换为大写，并将其用于元变量。有时，这不是你想要的 - 例如，该--filename选项显式设置metavar="FILE"，导致这个自动生成的选项描述：

  ```
  -f FILE, --filename=FILE
  ```

  不过这不仅仅是节省空间很重要：手动编写的帮助文本使用元变量FILE来暗示用户，因为在半形式语法和非正式语义描述“将输出写入FILE”之间存在关联。这是一种简单而有效的方式，可以使您的帮助文本对最终用户更加清晰和有用。-f FILE

- 具有默认值的选项可以包含%default在帮助字符串中 - optparse将其替换str()为选项的默认值。如果选项没有默认值（或默认值为 None），则%default展开为none。

36.1.2.6.1。分组选项

在处理多种选项时，将这些选项分组以更好地帮助输出是很方便的。一个OptionParser可以包含多个选项组，每个选项组可以包含多个选项。

选项组使用该类获得OptionGroup：

*类*optparse.OptionGroup（*解析器*，*标题*，*描述=无*）

哪里

- 解析器是该OptionParser组将被插入的实例
- 标题是组标题
- 描述，可选，是对该组的长描述

OptionGroup继承自OptionContainer（like OptionParser），因此该add_option()方法可用于向该组添加选项。

一旦声明了所有选项，使用该OptionParser方法 add_option_group()将该组添加到先前定义的解析器中。

继续上一节中定义的解析器，将OptionGroup解析器添加 到解析器很容易：

```
group = OptionGroup(parser, "Dangerous Options",
                    "Caution: use these options at your own risk.  "
                    "It is believed that some of them bite.")
group.add_option("-g", action="store_true", help="Group option.")
parser.add_option_group(group)
```

这将导致以下帮助输出：

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose         make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or
                        expert [default: intermediate]

  Dangerous Options:
    Caution: use these options at your own risk.  It is believed that some
    of them bite.

    -g                  Group option.
```

更完整的示例可能涉及使用多个组：仍然扩展前面的示例：

```
group = OptionGroup(parser, "Dangerous Options",
                    "Caution: use these options at your own risk.  "
                    "It is believed that some of them bite.")
group.add_option("-g", action="store_true", help="Group option.")
parser.add_option_group(group)

group = OptionGroup(parser, "Debug Options")
group.add_option("-d", "--debug", action="store_true",
                 help="Print debug information")
group.add_option("-s", "--sql", action="store_true",
                 help="Print all SQL statements executed")
group.add_option("-e", action="store_true", help="Print every action done")
parser.add_option_group(group)
```

导致以下输出：

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
```

```
  -v, --verbose         make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or expert
                        [default: intermediate]

  Dangerous Options:
    Caution: use these options at your own risk.  It is believed that some
    of them bite.

    -g                  Group option.

  Debug Options:
    -d, --debug         Print debug information
    -s, --sql           Print all SQL statements executed
    -e                  Print every action done
```

另一个有趣的方法，特别是使用选项组进行编程时：

OptionParser.**get_option_group**(*opt_str*)

返回OptionGroup短或长选项字符串*opt_str*（例如'-o'或'--option'）所属的。如果没有这样的OptionGroup回报None。

36.1.2.7。打印版本字符串

与简短的使用字符串类似，optparse也可以为您的程序打印一个版本字符串。您必须提供字符串作为version OptionParser 的参数：

```
parser = OptionParser(usage="%prog [-f] [-q]", version="%prog 1.0")
```

%prog就像它在扩大一样usage。除此之外， version可以包含任何你喜欢的东西。当您提供它时，会optparse 自动--version为您的解析器添加一个选项。如果在命令行中遇到此选项，它会扩展您的version字符串（通过替换%prog），将其输出到stdout并退出。

例如，如果您的脚本被调用/usr/bin/foo：

```
$ /usr/bin/foo --version
foo 1.0
```

以下两种方法可用于打印和获取version字符串：

OptionParser.**print_version**(*file = None*)

将当前程序（ self.version ）的版本信息打印到 *文件*（默认stdout）。与之一样 print_usage() ，任何%progin的出现都 self.version被替换为当前程序的名称。如果 self.version是空的或未定义的，什么也不做。

OptionParser.**get_version**( )

print_version()与之相同，但返回版本字符串而不是打印它。

36.1.2.8。如何optparse处理错误

有两大类错误optparse需要担心：程序员错误和用户错误。程序员错误通常是错误的调用OptionParser.add_option()，例如无效的选项字符串，未知的选项属性，缺少的选项属性等。这些问题以常规方式处理：引发异常（optparse.OptionError或者TypeError）并让程序崩溃。

处理用户错误更为重要，因为无论您的代码如何稳定，它们都会保证发生。optparse可以自动检测一些用户错误，例如错误的选项参数（传递where 获取整数参数），缺少参数（在命令行末尾，其中接受任何类型的参数）。此外，您可以调用以指示应用程序定义的错误条件：-n 4x -n -n -n OptionParser.error()

```
(options, args) = parser.parse_args()
...
if options.a and options.b:
    parser.error("options -a and -b are mutually exclusive")
```

在任何一种情况下，都optparse以相同的方式处理错误：它将程序的使用消息和错误消息打印到标准错误，并以错误状态2退出。

考虑上面的第一个例子，用户传递4x给一个带有整数的选项：

```
$ /usr/bin/foo -n 4x
Usage: foo [options]

foo: error: option -n: invalid integer value: '4x'
```

或者，在用户未能传递任何值的情况下：

```
$ /usr/bin/foo -n
Usage: foo [options]

foo: error: -n option requires an argument
```

optparse生成的错误消息始终注意提及错误中涉及的选项; OptionParser.error()从应用程序代码调用时一定要这样做。

如果optparse的默认错误处理行为不能满足您的需求，您需要继承OptionParser并覆盖其exit()和/或error()方法。

36.1.2.9。把它放在一起

以下是optparse基于脚本的脚本通常的样子：

```
from optparse import OptionParser
...
def main():
    usage = "usage: %prog [options] arg"
    parser = OptionParser(usage)
    parser.add_option("-f", "--file", dest="filename",
                      help="read data from FILENAME")
```

```
    parser.add_option("-v", "--verbose",
                      action="store_true", dest="verbose")
    parser.add_option("-q", "--quiet",
                      action="store_false", dest="verbose")
    ...
    (options, args) = parser.parse_args()
    if len(args) != 1:
        parser.error("incorrect number of arguments")
    if options.verbose:
        print("reading %s..." % options.filename)
    ...

if __name__ == "__main__":
    main()
```

36.1.3。参考指南

36.1.3.1。创建解析器

使用的第一步optparse是创建一个OptionParser实例。

class optparse.OptionParser（...）

OptionParser构造函数没有必需的参数，但有许多可选的关键字参数。您应该始终将它们作为关键字参数传递，即不要依赖声明参数的顺序。

usage（默认值：）"%prog [options]"

当程序运行不正确或使用帮助选项时，可以使用该用法摘要进行打印。当optparse打印使用字符串时，它扩展 %prog为os.path.basename(sys.argv[0])（或者prog如果您传递了该关键字参数）。要抑制使用消息，请传递特殊值optparse.SUPPRESS_USAGE。

option_list（默认值：[]）

用于填充解析器的Option对象列表。这些选项 option_list是在任何选项standard_option_list（可以由OptionParser子类设置的类属性）之后，但在任何版本或帮助选项之前添加的。不推荐使用; add_option()改为创建解析器后使用。

option_class （默认：optparse.Option）

将选项添加到解析器中时使用的类add_option()。

version（默认值：None）

用户提供版本选项时要打印的版本字符串。如果您提供了真实值version，optparse则会自动添加带有单个选项字符串的版本选项--version。子字符串%prog的扩展与以前相同usage。

conflict_handler（默认值："error"）

指定将具有冲突选项字符串的选项添加到解析器时执行的操作; 请参阅选项之间的冲突部分。

description（默认值：None）

一段文字简要介绍了您的计划。 optparse重新格式化该段落以符合当前终端宽度，并在用户请求帮助时（在usage选项列表之前但之前）打印它。

formatter（默认：一个新的IndentedHelpFormatter）

optparse.HelpFormatter的一个实例，将用于打印帮助文本。 optparse为此提供了两个具体类：IndentedHelpFormatter和TitledHelpFormatter。

add_help_option（默认值：True）
如果为true，optparse则会向解析器添加一个帮助选项（带有选项字符串-h 和--help）。

prog
在扩展时使用的字符串%prog，usage而version不是os.path.basename(sys.argv[0])。

epilog（默认值：None）
在选项帮助后打印一段帮助文本。

36.1.3.2。 解析器

有几种方法可以用选项填充解析器。首选方法是使用OptionParser.add_option()，如教程部分所示。 add_option()可以通过以下两种方式之一进行调用：

- 传递一个Option实例（如返回的那样make_option()）
- 传递它可以接受的位置参数和关键字参数的任意组合make_option()（即选项构造函数），它将为您创建Option实例

另一种方法是将预先构建的Option实例列表传递给OptionParser构造函数，如下所示：

```
option_list = [
    make_option("-f", "--filename",
                action="store", type="string", dest="filename"),
    make_option("-q", "--quiet",
                action="store_false", dest="verbose"),
    ]
parser = OptionParser(option_list=option_list)
```

（make_option()是用于创建Option实例的工厂函数;目前它是Option构造函数的别名，将来的版本optparse可能会将Option拆分为多个类，make_option()并将选择正确的类实例化，不要直接实例化选项。

36.1.3.3。定义选项

每个选项实例代表一组同义的命令行选项字符串，例如-f和--file。您可以指定任意数量的短或长选项字符串，但必须至少指定一个总体选项字符串。

创建Option实例的规范方法是使用 add_option()方法OptionParser。

OptionParser.add_option(*可选*)
OptionParser.add_option(*opt_str*, *attr=value*, ...)
要用一个简短的选项字符串来定义一个选项：

```
parser.add_option("-f", attr=value, ...)
```

并且只用一个很长的选项字符串来定义一个选项：

```
parser.add_option("--foo", attr=value, ...)
```

关键字参数定义了新的Option对象的属性。最重要的选项属性是action，它很大程度上决定了哪些其他属性是相关或需要的。如果您传递不相关的选项属性或未能通过所需属性，optparse 则会引发一个OptionError异常来解释您的错误。

选项的*动作*决定了optparse它在命令行中遇到此选项时会发生什么。硬编码的标准选项操作 optparse是：

"store"
存储此选项的参数（默认）

"store_const"
存储一个常数值

"store_true"
存储一个真正的价值

"store_false"
存储一个虚假的值

"append"
将此选项的参数附加到列表中

"append_const"
将一个常数值附加到列表中

"count"
增加一个计数器

"callback"
调用指定的功能

"help"
打印包含所有选项和文档的使用消息

（如果您不提供操作，则默认为"store"。对于此操作，您还可以提供操作type和dest选项属性;请参阅标准选项操作。）

如您所见，大多数操作都涉及在某处存储或更新值。 optparse总是为此创建一个特殊对象，通常称为 options（它恰好是一个实例optparse.Values）。根据dest（目标）选项属性，选项参数（以及其他各种值）被存储为该对象的属性。

例如，当你打电话

```
parser.parse_args()
```

首先要做的optparse是创建options对象：

```
options = Values()
```

如果此解析器中的某个选项是使用定义的

```
parser.add_option("-f", "--file", action="store", type="string", dest="filename")
```

并且正在解析的命令行包含以下任一项：

```
-ffoo
-f foo
--file=foo
--file foo
```

那么optparse，看到这个选项，就会做相当于

```
options.filename = "foo"
```

在type和dest选项属性几乎一样重要action，但是action是唯一一个有意义*的所有*选项。

36.1.3.4。选项属性

以下选项属性可以作为关键字参数传递给 OptionParser.add_option()。如果传递与特定选项无关的选项属性，或者未能传递必需的选项属性，则会 optparse引发OptionError。

Option.**action**
（默认值："store"）

确定optparse在命令行中显示此选项时的行为; 可用的选项在这里记录。

Option.**type**
（默认值："string"）

该选项期望的参数类型（例如"string"或"int"）; 在这里记录可用的选项类型。

Option.**dest**
（默认：从选项字符串派生）

如果该选项的操作意味着在某处写入或修改了某个值，optparse则会告诉写入它的位置：dest命名在分析命令行时构建的options对象的属性optparse。

Option.**default**
如果该选项未在命令行中显示，则用于此选项目标的值。另见OptionParser.set_defaults()。

Option.**nargs**
（默认值：1）

type当看到这个选项时，应该消耗多少种类型的参数。如果> 1，optparse将存储值的元组dest。

Option.**const**
对于存储常量值的操作，要存储的常量值。

Option.**choices**

对于类型选项，"choice"用户可以从中选择字符串列表。

Option.**callback**

对于具有操作的选项，"callback"可以在看到此选项时调用可调用的选项。有关传递给可调用参数的参数的详细信息，请参见选项回调一节。

Option.**callback_args**

Option.**callback_kwargs**

callback在四个标准回调参数后传递的附加位置和关键字参数。

Option.**help**

在用户提供help选项（如--help）后列出所有可用选项时，可以为此选项打印帮助文本。如果没有提供帮助文本，该选项将被列出而没有帮助文本。要隐藏此选项，请使用特殊值optparse.SUPPRESS_HELP。

Option.**metavar**

（默认：从选项字符串派生）

在打印帮助文本时，可以使用选项参数。有关示例，请参见教程部分。

36.1.3.5。标准选项操作

各种选择行动都有稍微不同的要求和影响。大多数行为都有几个相关的选项属性，您可以指定它们来指导optparse行为; 有几个需要属性，您必须为使用该操作的任何选项指定属性。

- "store"[相关：type，dest，nargs，choices]

  该选项后面必须跟随一个参数，该参数将根据type并存储在一个值中转换为一个值dest。如果 nargs> 1，将从命令行消耗多个参数; 全部将根据type并转换dest为一个元组进行转换。请参阅 标准选项类型部分。

  如果choices提供（字符串的列表或元组），则类型默认为"choice"。

  如果type未提供，则默认为"string"。

  如果dest未提供，optparse则从第一个长选项字符串派生一个目标（例如，--foo-bar暗示 foo_bar）。如果没有长选项字符串，optparse则从第一个短选项字符串派生一个目标（例如，-f暗示f）。

  例：

  ```
  parser.add_option("-f")
  parser.add_option("-p", type="float", nargs=3, dest="point")
  ```

  分析命令行

  ```
  -f foo.txt -p 1 -3.5 4 -fbar.txt
  ```

  optparse 将设置

```
options.f = "foo.txt"
options.point = (1.0, -3.5, 4.0)
options.f = "bar.txt"
```

- "store_const"[要求：const; 相关： dest]

  该值const存储在dest。

  例：

```
parser.add_option("-q", "--quiet",
                  action="store_const", const=0, dest="verbose")
parser.add_option("-v", "--verbose",
                  action="store_const", const=1, dest="verbose")
parser.add_option("--noisy",
                  action="store_const", const=2, dest="verbose")
```

  如果--noisy被看到，optparse将会设置

```
options.verbose = 2
```

- "store_true"[相关：dest]

  一个特殊的情况下"store_const"存储一个真正的价值 dest。

- "store_false"[相关：dest]

  喜欢"store_true"，但存储一个虚假的价值。

  例：

```
parser.add_option("--clobber", action="store_true", dest="clobber")
parser.add_option("--no-clobber", action="store_false", dest="clobber")
```

- "append"[相关：type，dest， nargs，choices]

  该选项必须后跟一个参数，该参数将附加到列表中 dest。如果未提供默认值，dest则optparse在命令行上首次遇到此选项时，会自动创建一个空列表。如果nargs> 1，则会消耗多个参数，并且nargs 会附加一个长度元组dest。

  操作的默认值type和dest相同"store"。

  例：

```
parser.add_option("-t", "--tracks", action="append", type="int")
```

  如果-t3在命令行中看到，optparse是否等同于：

```
options.tracks = []
options.tracks.append(int("3"))
```

  如果稍后--tracks=4看到，它会：

```
options.tracks.append(int("4"))
```

该append动作调用append该选项当前值的方法。这意味着任何指定的默认值都必须有一个 append 方法。这也意味着，如果默认值非空，则缺省元素将出现在该选项的解析值中，并且在这些缺省值之后附加任何来自命令行的值：

```
>>> parser.add_option("--files", action="append", default=['~/.mypkg/defaults'])
>>> opts, args = parser.parse_args(['--files', 'overrides.mypkg'])
>>> opts.files
['~/.mypkg/defaults', 'overrides.mypkg']
```

- "append_const"[要求：const; 相关： dest]

  喜欢"store_const"，但const附加 价值dest; 与之一样"append"，dest默认为 None，并且在遇到第一次选项时自动创建一个空列表。

- "count"[相关：dest]

  增加存储在的整数dest。如果没有提供默认值，dest则在第一次增加之前将其设置为零。

  例：

  ```
  parser.add_option("-v", action="count", dest="verbosity")
  ```

  第一次-v在命令行中看到，optparse相当于：

  ```
  options.verbosity = 0
  options.verbosity += 1
  ```

  每一次随后发生的-v结果

  ```
  options.verbosity += 1
  ```

- "callback"[要求：callback; 相关： type，nargs，callback_args， callback_kwargs]

  通过调用指定的函数callback，它被称为

  ```
  func(option, opt_str, value, parser, *args, **kwargs)
  ```

  更多细节请参见选项回调部分。

- "help"

  打印当前选项解析器中所有选项的完整帮助信息。帮助信息由usage传递给OptionParser构造函数的help字符串和传递给每个选项的字符串构造而成。

  如果没有help为选项提供字符串，它仍将列在帮助消息中。要完全省略选项，请使用特殊值 optparse.SUPPRESS_HELP。

optparse自动help为所有OptionParser 添加一个选项，所以你通常不需要创建一个选项。

例：

```
from optparse import OptionParser, SUPPRESS_HELP

# usually, a help option is added automatically, but that can
# be suppressed using the add_help_option argument
parser = OptionParser(add_help_option=False)

parser.add_option("-h", "--help", action="help")
parser.add_option("-v", action="store_true", dest="verbose",
                  help="Be moderately verbose")
parser.add_option("--file", dest="filename",
                  help="Input file to read data from")
parser.add_option("--secret", help=SUPPRESS_HELP)
```

如果在命令行中optparse看到-h或者--help在命令行上看到，它会将如下帮助消息打印到stdout（假设 sys.argv[0]是"foo.py"）：

```
Usage: foo.py [options]

Options:
  -h, --help        Show this help message and exit
  -v                Be moderately verbose
  --file=FILENAME   Input file to read data from
```

打印完帮助信息后，将optparse终止您的过程 sys.exit(0)。

- "version"

  将提供给OptionParser的版本号打印到标准输出并退出。版本号实际上是通过print_version()OptionParser 的方法格式化和打印的。通常只有在 version参数提供给OptionParser构造函数时才有用。与 help选项一样，您很少创建version选项，因为optparse在需要时自动添加选项。

36.1.3.6。标准选项类型

optparse有五个内置选项类型："string"，"int"， "choice"，"float"和"complex"。如果您需要添加新的选项类型，请参阅扩展optparse部分。

字符串选项的参数不会以任何方式检查或转换：命令行上的文本按原样存储在目标中（或传递给回调函数）。

整数参数（类型"int"）被解析如下：

- 如果该号码开头0x，则将其解析为十六进制数字
- 如果数字以数字开头0，则将其解析为八进制数字
- 如果数字开头0b，它将被解析为一个二进制数字
- 否则，该数字将被解析为十进制数字

转换是通过调用int()适当的基地（2,8,10或16）来完成的。如果失败了，optparse尽管有更多有用的错误信息。

"float"并且"complex"选项参数可以直接使用float()和进行转换 complex() ，具有类似的错误处理。

"choice"选项是选项的子类型"string"。该 choices可选属性（字符串序列）定义了一组允许的选项参数。 optparse.check_choice()将用户提供的选项参数与此主列表进行比较，并OptionValueError在发出无效字符串时引发 。

36.1.3.7。解析参数

创建和填充OptionParser的要点是调用它的 parse_args()方法：

```
(options, args) = parser.parse_args(args=None, values=None)
```

输入参数在哪里

args
: 参数列表来处理（默认值：sys.argv[1:] ）

values
: 一个optparse.Values存储选项参数的对象（默认值：一个新的实例Values ） - 如果你给一个现有的对象，选项的默认值将不会被初始化

和返回值是

options
: values与之相同的对象，或者由其创建的optparse.Values实例optparse

args
: 处理所有选项后的剩余位置参数

最常见的用法是不提供关键字参数。如果您提供 values ，它将被重复setattr()调用修改（对于存储到选项目标的每个选项参数大致为一个）并返回 parse_args()。

如果parse_args()在参数列表中遇到任何错误，它会error()用适当的最终用户错误消息调用OptionParser的方法。这最终会终止您的进程，退出状态为2（命令行错误的传统Unix退出状态）。

36.1.3.8。查询和操作您的选项解析器

选项解析器的默认行为可以进行轻微的自定义，您也可以在选项解析器周围徘徊，看看有什么。OptionParser提供了几种方法来帮助你：

OptionParser. disable_interspersed_args ( )
: 将解析设置为在第一个非选项上停止。例如，如果-a和 -b都是不带参数的简单选项，optparse 通常会接受以下语法：

  ```
  prog -a arg1 -b arg2
  ```

并认为它等同于

```
prog -a -b arg1 arg2
```

要禁用此功能，请致电disable_interspersed_args()。这恢复了传统的Unix语法，其中选项解析在第一个非选项参数中停止。

如果你有一个命令处理器运行另一个具有自己的选项的命令，并且你想确保这些选项不会感到困惑，请使用它。例如，每个命令可能有一组不同的选项。

OptionParser.enable_interspersed_args()
将解析设置为不停止第一个非选项，允许使用命令参数散布开关。这是默认行为。

OptionParser.get_option(*opt_str*)
使用选项字符串*opt_str*返回选项实例，或者None如果没有选项具有该选项字符串。

OptionParser.has_option(*opt_str*)
如果OptionParser有选项字符串*opt_str*（例如-q或--verbose），则返回true。

OptionParser.remove_option(*opt_str*)
如果OptionParser具有与*opt_str*相对应的选项，则删除该选项。如果该选项提供了任何其他选项字符串，则所有这些选项字符串都将失效。如果*opt_str*不属于属于此的任何选项OptionParser，则引发ValueError。

36.1.3.9。选项之间的冲突

如果您不小心，可以很容易地定义带有冲突选项字符串的选项：

```
parser.add_option("-n", "--dry-run", ...)
...
parser.add_option("-n", "--noisy", ...)
```

（如果您已经使用一些标准选项定义了自己的OptionParser子类，则尤其如此。）

每次添加选项时，都会optparse检查与现有选项的冲突。如果它找到了，它会调用当前的冲突处理机制。您可以在构造函数中设置冲突处理机制：

```
parser = OptionParser(..., conflict_handler=handler)
```

或者单独拨打电话：

```
parser.set_conflict_handler(handler)
```

可用的冲突处理程序是：

"error"（默认）
假设选项冲突是编程错误并引发 OptionConflictError

"resolve"

智能解决选项冲突（见下文）

作为一个例子，让我们定义一个OptionParser智能地解决冲突并为其添加冲突的选项：

```
parser = OptionParser(conflict_handler="resolve")
parser.add_option("-n", "--dry-run", ..., help="do no harm")
parser.add_option("-n", "--noisy", ..., help="be noisy")
```

此时，optparse检测到以前添加的选项已在使用-n选项字符串。既然conflict_handler是这样"resolve"，它通过-n从早期选项的选项字符串列表中删除来解决这种情况。现在--dry-run是用户激活该选项的唯一方式。如果用户请求帮助，帮助信息将反映：

```
Options:
  --dry-run     do no harm
  ...
  -n, --noisy   be noisy
```

可以删除先前添加的选项的选项字符串，直到没有剩下的选项，并且用户无法从命令行调用该选项。在这种情况下，请optparse完全删除该选项，以便它不会显示在帮助文本或其他任何地方。继续使用我们现有的OptionParser：

```
parser.add_option("--dry-run", ..., help="new dry-run option")
```

此时，原始-n/ --dry-run选项不再可访问，因此optparse将其删除，留下以下帮助文本：

```
Options:
  ...
  -n, --noisy   be noisy
  --dry-run     new dry-run option
```

36.1.3.10。清理

OptionParser实例有几个循环引用。这对Python的垃圾回收器来说不应该是个问题，但是你可能希望在完成之后通过调用destroy()你的OptionParser来明确地断开循环引用。这对于长时间运行的应用程序特别有用，可以从OptionParser访问大对象图。

36.1.3.11。其他方法

OptionParser支持其他几种公共方法：

OptionParser. set_usage（*用法*）

根据上述关于usage 构造函数关键字参数的规则设置用法字符串。通过None设置默认使用字符串; 用于optparse.SUPPRESS_USAGE抑制使用消息。

OptionParser. print_usage（*file* = *None* ）

将当前程序的使用消息（ self.usage ）打印到*文件*（默认stdout）。字符串的任何出现%prog在self.usage 被替换为当前程序的名称。如果self.usage是空的或没有定义，什么也不做。

OptionParser.get_usage ( )

与print_usage()使用字符串相同，但返回使用字符串而不是打印它。

OptionParser.set_defaults (*dest* = *value* , ...)

一次为多个选项目标设置默认值。使用 set_defaults()是设置选项默认值的首选方式，因为多个选项可以共享相同的目标。例如，如果多个“模式”选项都设置相同的目的地，则其中任何一个都可以设置默认值，最后一个可以胜出：

```
parser.add_option("--advanced", action="store_const",
                  dest="mode", const="advanced",
                  default="novice")    # overridden below
parser.add_option("--novice", action="store_const",
                  dest="mode", const="novice",
                  default="advanced")  # overrides above setting
```

为了避免这种混淆，请使用set_defaults()：

```
parser.set_defaults(mode="advanced")
parser.add_option("--advanced", action="store_const",
                  dest="mode", const="advanced")
parser.add_option("--novice", action="store_const",
                  dest="mode", const="novice")
```

36.1.4。选项回调

当optparse内置动作和类型不足以满足您的需求时，您有两种选择：扩展optparse或定义回调选项。扩展optparse是更一般的，但对于很多简单情况来说却是矫枉过正的。通常一个简单的回调就是你所需要的。

定义回调选项有两个步骤：

- 使用该"callback"操作定义选项本身
- 写回调; 这是一个至少需要四个参数的函数（或方法），如下所述

36.1.4.1。定义回调选项

与往常一样，定义回调选项的最简单方法是使用该 OptionParser.add_option()方法。除此之外 action，您必须指定的唯一选项属性是callback要调用的函数：

```
parser.add_option("-c", action="callback", callback=my_callback)
```

callback是一个函数（或其他可调用对象），因此您必须my_callback()在创建此回调选项时已经定义。在这种简单的情况下，optparse甚至不知道是否-c接受任何参数，这通常意味着该选项不会引用任何参数 - 只需-c在命令行上存在 即可。但是，在某些情况下，您可能希望回调消耗任意数量的命令行参数。这是写回调变得棘手的地方; 本节稍后会介绍。

optparse总是将四个特定参数传递给您的回调函数，如果您通过callback_args和指定它们，它将只传递附加参数 callback_kwargs。因此，最小回调函数签名是：

```
def my_callback(option, opt, value, parser):
```

下面介绍回调的四个参数。

在定义回调选项时，您可以提供其他几个选项属性：

type
: 具有其通常的含义：与"store"or或"append"action一样，它指示optparse消费一个参数并将其转换为 type。但不是将转换后的值存储在任何地方，而是optparse将其传递给您的回调函数。

nargs
: 也有其通常的含义：如果它被提供并且> 1，optparse将消费nargs参数，每个参数都必须可转换为 type。然后它将转换值的元组传递给您的回调。

callback_args
: 一个额外的位置参数的元组传递给回调

callback_kwargs
: 一个额外的关键字参数字典传递给回调

36.1.4.2。如何调用回调

所有的回调被调用如下：

```
func(option, opt_str, value, parser, *args, **kwargs)
```

哪里

option
: 是调用回调的Option实例

opt_str
: 是在触发回调的命令行中看到的选项字符串。（如果使用了一个缩写的长选项，opt_str它将是完整的，规范的选项字符串 - 例如，如果用户将--foo命令行作为缩写--foobar，opt_str则将是 "--foobar"。）

value
: 是在命令行中看到的这个选项的参数。 optparse只有type设定了才会期待论据; 类型value将是选项类型所暗示的类型。如果type这个选项是 None（没有理由），那么value将是None。如果nargs > 1，value将是适当类型的值的元组。

parser
: 是驱动整个事物的OptionParser实例，主要是有用的，因为您可以通过其实例属性访问其他有趣的数据：

  parser.largs
  : 当前的剩余参数列表，即。已被使用但不是选项或选项参数的参数。随意修改parser.largs，例如添加更多参数。（这个列表将成为args第二个返回值parse_args()。）

  parser.rargs
  : 剩余参数的当前列表，即。同opt_str和 value（如适用）去除，并且只参数如下他们仍然存在。随意修改parser.rargs，例如通过消耗更多参数。

parser.values
默认存储选项值的对象（optparse.OptionValues的实例）。这可以让回调使用与其余部分相同的机制optparse来存储选项值; 你不需要混淆全局或封闭。您还可以访问或修改已在命令行中遇到的任何选项的值。

args
是通过callback_argsoption属性提供的任意位置参数的元组 。

kwargs
是通过提供的任意关键字参数的字典 callback_kwargs。

36.1.4.3。在回调中　　错误

OptionValueError如果该选项或其参数有任何问题，则应该提高回调函数。 optparse捕获此并终止程序，打印您提供给stderr的错误消息。您的信息应该清晰，简洁，准确，并提及有问题的选项。否则，用户将很难弄清楚他做错了什么。

36.1.4.4。回调示例1：简单回调

下面是一个不带任何参数的回调选项的例子，并简单地记录下该选项：

```
def record_foo_seen(option, opt_str, value, parser):
    parser.values.saw_foo = True

parser.add_option("--foo", action="callback", callback=record_foo_seen)
```

当然，你可以用这个"store_true"动作来做到这一点。

36.1.4.5。回调示例2：检查选项顺序

这里有一个稍微有趣的例子：记录下-a看到的事实，但如果它-b在命令行中出现，则会炸毁。

```
def check_order(option, opt_str, value, parser):
    if parser.values.b:
        raise OptionValueError("can't use -a after -b")
    parser.values.a = 1
...
parser.add_option("-a", action="callback", callback=check_order)
parser.add_option("-b", action="store_true", dest="b")
```

36.1.4.6。回调示例3：检查选项顺序（概括）

如果你想为几个类似的选项重新使用这个回调-b函数（设置一个标志，但是如果已经被看到就炸毁），它需要一些工作：错误信息和它设置的标志必须被概括。

```
def check_order(option, opt_str, value, parser):
    if parser.values.b:
        raise OptionValueError("can't use %s after -b" % opt_str)
    setattr(parser.values, option.dest, 1)
```

```
...
parser.add_option("-a", action="callback", callback=check_order, dest='a')
parser.add_option("-b", action="store_true", dest="b")
parser.add_option("-c", action="callback", callback=check_order, dest='c')
```

36.1.4.7。回调示例4：检查任意条件

当然，您可以在其中放置任何条件 - 您不仅限于检查已定义选项的值。例如，如果您在月球满了时不应调用选项，则只需执行以下操作：

```
def check_moon(option, opt_str, value, parser):
    if is_moon_full():
        raise OptionValueError("%s option invalid when moon is full"
                               % opt_str)
    setattr(parser.values, option.dest, 1)
...
parser.add_option("--foo",
                  action="callback", callback=check_moon, dest="foo")
```

（这个定义`is_moon_full()`留给读者练习。）

36.1.4.8。回调示例5：固定参数

当定义带有固定数量参数的回调选项时，情况会变得更有趣。指定回调选项需要参数类似于定义一个`"store"`或一个`"append"`选项：如果定义 `type` ，则该选项带有一个必须可转换为该类型的参数; 如果你进一步定义`nargs` ，那么该选项需要`nargs`参数。

以下是一个模拟标准`"store"`动作的例子：

```
def store_value(option, opt_str, value, parser):
    setattr(parser.values, option.dest, value)
...
parser.add_option("--foo",
                  action="callback", callback=store_value,
                  type="int", nargs=3, dest="foo")
```

请注意，`optparse`需要消耗3个参数并将它们转换为整数; 你所要做的就是存储它们。（或者其他;显然你不需要回调这个例子。）

36.1.4.9。回调示例6：变量参数

当你想要一个选项来获取可变数量的参数时，情况会变得多毛。对于这种情况，您必须编写回调`optparse`函数，因为它不提供任何内置功能。而且您必须处理`optparse`通常为您处理的常规Unix命令行解析的某些复杂问题。具体来说，回调应该执行裸`--`和`-`参数的传统规则：

- 无论是`--`或`-`可选项参数
- 裸`--`（如果不是一些选项的参数）：停止命令行处理并放弃`--`
- 裸露`-`（如果不是某些选项的参数）：暂停命令行处理，但保留`-`（附加到`parser.largs`）

如果你想要一个带有可变数量参数的选项，那么需要考虑几个微妙棘手的问题。您选择的具体实现将基于您愿意为您的应用程序进行哪些权衡（这就是为什么optparse不直接支持这种事情的原因）。

尽管如此，下面是一个带有可变参数的选项的回调函数：

```
def vararg_callback(option, opt_str, value, parser):
    assert value is None
    value = []

    def floatable(str):
        try:
            float(str)
            return True
        except ValueError:
            return False

    for arg in parser.rargs:
        # stop on --foo like options
        if arg[:2] == "--" and len(arg) > 2:
            break
        # stop on -a, but not on -3 or -3.0
        if arg[:1] == "-" and len(arg) > 1 and not floatable(arg):
            break
        value.append(arg)

    del parser.rargs[:len(value)]
    setattr(parser.values, option.dest, value)

...
parser.add_option("-c", "--callback", dest="vararg_attr",
                  action="callback", callback=vararg_callback)
```

36.1.5。扩展 optparse

由于optparse解释命令行选项的两个主要控制因素是每个选项的操作和类型，所以最可能的扩展方向是添加新操作和新类型。

36.1.5.1。添加新类型

要添加新的类型，你需要定义自己的子类optparse的 Option类。这个类有几个定义optparse类型的属性 ：TYPES和TYPE_CHECKER。

Option.TYPES

类型名称的元组; 在你的子类中，只需定义一个TYPES基于标准元组的新元组 。

Option.TYPE_CHECKER

字典映射类型名称到类型检查函数。类型检查函数具有以下签名：

```
def check_mytype(option, opt, value)
```

在哪里option是一个Option实例，opt是一个选项字符串（例如-f），并且value是命令行中必须检查并转换为所需类型的字符串。 check_mytype()应返回假设类型的对象mytype。类型检查函数返回的值将返回到返回的OptionValues实例中OptionParser.parse_args()，或作为value参数传递给回调 函数。

OptionValueError如果遇到任何问题，您的类型检查功能应该升高。 OptionValueError接受一个字符串参数，它被原样传递是OptionParser的error() 方法，这反过来会预先考虑节目名称和字符串"error:" 和结束处理之前打印一切到stderr。

下面是一个愚蠢的例子，演示了"complex"如何在命令行中添加一个选项类型来分析Python风格的复数。（这比以前更加恶劣，因为optparse1.3增加了对复杂数字的内置支持，但没关系。）

首先，必要的进口：

```
from copy import copy
from optparse import Option, OptionValueError
```

您需要首先定义类型检查器，因为它稍后会引用（在TYPE_CHECKER您的Option子类的 类属性中）：

```
def check_complex(option, opt, value):
    try:
        return complex(value)
    except ValueError:
        raise OptionValueError(
            "option %s: invalid complex value: %r" % (opt, value))
```

最后，Option子类：

```
class MyOption (Option):
    TYPES = Option.TYPES + ("complex",)
    TYPE_CHECKER = copy(Option.TYPE_CHECKER)
    TYPE_CHECKER["complex"] = check_complex
```

（如果我们不做出copy()的Option.TYPE_CHECKER，我们最终会修改TYPE_CHECKER的属性optparse的选项类。这是Python的，没有什么可以阻止你做，除了良好的礼仪和常识。）

而已！现在您可以编写一个使用新选项类型optparse的脚本，就像其他任何基于脚本的脚本一样，除非您必须指示OptionParser使用MyOption而不是Option：

```
parser = OptionParser(option_class=MyOption)
parser.add_option("-c", type="complex")
```

或者，您可以构建自己的选项列表并将其传递给OptionParser; 如果你不add_option()以上面的方式使用，你不需要告诉OptionParser使用哪个选项类：

```
option_list = [MyOption("-c", action="store", type="complex", dest="c")]
parser = OptionParser(option_list=option_list)
```

36.1.5.2。添加新的动作

添加新的动作有点棘手，因为您必须了解这个动作 optparse有几个分类：

“存储”操作
: 导致optparse将值存储到当前OptionValues实例的属性的操作; 这些选项需要将一个dest属性提供给Option构造函数。

“键入”操作
: 从命令行获取值并期望它具有某种特定类型的操作; 或者更确切地说，是一个可以转换成某种类型的字符串。这些选项需要type Option构造函数的一个属性。

这些重叠的集合：一些默认的“店”的行动是"store"，"store_const"，"append"，和"count"，而默认的“类型化”行动"store"，"append"和"callback"。

当你添加一个动作时，你需要通过将其列入至少一个以下的Option类属性中进行分类（所有都是字符串列表）：

Option.ACTIONS
: 所有操作必须在操作中列出。

Option.STORE_ACTIONS
: 此处还列出了“商店”操作。

Option.TYPED_ACTIONS
: 此处列出了“键入”操作。

Option.ALWAYS_TYPED_ACTIONS
: 总是采用某种类型的操作（即其选项总是取值）在此处另行列出。这样做的唯一影响是optparse 将默认类型，分配给"string"没有列出其操作的显式类型的选项ALWAYS_TYPED_ACTIONS。

为了实际执行您的新操作，您必须覆盖Option的 take_action()方法并添加一个识别您的操作的案例。

例如，让我们添加一个"extend"动作。这与标准"append"动作类似 ，但不是从命令行获取单个值并将其附加到现有列表，而是使用"extend"逗号分隔的单个字符串取多个值，并用它们扩展现有列表。也就是说，如果--names是"extend"类型选项，则是"string"命令行

```
--names=foo,bar --names blah --names ding,dong
```

会导致一个列表

```
["foo", "bar", "blah", "ding", "dong"]
```

我们再次定义一个Option的子类：

```
class MyOption(Option):

    ACTIONS = Option.ACTIONS + ("extend",)
    STORE_ACTIONS = Option.STORE_ACTIONS + ("extend",)
    TYPED_ACTIONS = Option.TYPED_ACTIONS + ("extend",)
    ALWAYS_TYPED_ACTIONS = Option.ALWAYS_TYPED_ACTIONS + ("extend",)
```

```
  def take_action(self, action, dest, opt, value, values, parser):
      if action == "extend":
          lvalue = value.split(",")
          values.ensure_value(dest, []).extend(lvalue)
      else:
          Option.take_action(
              self, action, dest, opt, value, values, parser)
```

说明的特点：

- "extend"双方预计，在命令行和重视的地方存储值，如此这般在这两个STORE_ACTIONS和TYPED_ACTIONS。
- 为了确保optparse将默认类型分配"string"给 "extend"操作，我们也将"extend"操作放入 ALWAYS_TYPED_ACTIONS。
- MyOption.take_action()只实现这一个新动作，并将控制权交给Option.take_action()标准optparse 动作。
- values是optparse_parser.Values类的一个实例，它提供了非常有用的ensure_value()方法。 ensure_value()实质上getattr()是安全阀; 它被称为

  ```
  values.ensure_value(attr, value)
  ```

  如果该attr属性values不存在或是None，则sure_value（ ）首先将其设置为value，然后返回'value。这对于像"extend"，，"append"和"count"所有这些操作非常方便，所有这些操作都将数据累加到一个变量中，并期望该变量具有某种类型（前两个是后者的整数）。使用 ensure_value()意味着使用您的操作的脚本不必担心为有问题的选项目标设置默认值; 他们可以保留默认值，None并ensure_value()在需要时处理。

36.2。 imp-进入导入内部

源代码： Lib / imp.py

*自3.4版以来已弃用：*此imp软件包正在等待弃用，以支持importlib。

该模块为用于实现该import语句的机制提供了一个接口 。它定义了以下常量和函数：

imp. get_magic ( ) ¶

返回用于识别字节编译代码文件（ .pyc文件）的魔术字符串值。（对于每个Python版本，此值可能不同。）

*自3.4版弃用：*importlib.util.MAGIC_NUMBER改为使用。

imp. get_suffixes ( )

返回3元组元组列表，每个元组描述一个特定类型的模块。每个三元组都有这样的形式，其中*后缀*是一个字符串，要附加到模块名称以形成要搜索的文件名，*模式*是要传递给内置函数以打开文件的模式字符串（可以用于文本文件或二进制文件），和*类型*是文件类型，其具有的一个值，或者，如下所述。(suffix, mode, type) open()'r''rb'PY_SOURCEPY_COMPILEDC_EXTENSION

*自3.3版弃用：*importlib.machinery改为使用定义的常量。

imp. find_module (*name* [, *path*])

尝试找到模块*名称*。如果省略了*路径*，或者搜索None给出的目录名称列表sys.path，但是首先搜索几个特殊的地方：函数试图找到具有给定名称（C_BUILTIN）的内置模块，然后是冻结模块（PY_FROZEN），以及在某些系统上也会查看其他地方（在Windows上，它会在注册表中查找可能指向特定文件的地方）。

否则，*路径*必须是目录名称列表; 每个目录都会搜索具有get_suffixes() 上述返回的任何后缀的文件。列表中无效的名称将被忽略（但所有列表项必须是字符串）。

如果搜索成功，则返回值是一个3元组元组：(file, pathname, description)

*文件*是一个位于开头的打开的文件对象，*pathname* 是找到的文件的路径名，*description*是一个3元元组，包含在通过get_suffixes()描述找到的模块类型返回的列表中。

如果模块不在文件中，则返回的*文件*是None， *pathname*是空字符串，*描述*元组的后缀和模式包含空字符串; 模块类型如上面的括号中所示。如果搜索不成功，ImportError则提出。其他例外表明争论或环境问题。

如果模块是一个包，*文件*是None，*pathname*是包路径，并且*描述*元组中的最后一个项是PKG_DIRECTORY。

此功能不处理分层模块名称（包含点的名称）。为了找到*PM*，即，子模块*中号*包的*P*，使用 find_module()和load_module()查找和负载包*P*，然后使用find_module()与*路径*设置为参数P.__path__。当*P*本身具有虚线名称时，递归应用此配方。

自3.3版弃用： `importlib.util.find_spec()` 除非需要Python 3.3兼容性，否则请使用 `importlib.find_loader()` 。例如前一种情况的用法，请参阅文档的“示例”部分 `importlib`。

imp.load_module(*名称*, *文件*, *路径名*, *说明*)

加载以前找到的模块`find_module()`（或通过其他方式执行的搜索生成兼容的结果）。此功能不仅仅是导入模块：如果模块已经导入，它将重新加载模块！该*名称*参数表示全模块名（包括包名，如果这是一个封装的子模块）。该*文件*的说法是一个开放的文件，*路径*是相应的文件名; 这些可以是None和''，分别，当模块是一个包或不被从文件中加载。该*描述*参数是一个元组，如将被退还`get_suffixes()`，说明什么样的模块必须加载。

如果加载成功，则返回值是模块对象; 否则，会引发异常（通常`ImportError`）。

重要提示：调用者负责关闭*文件*参数，如果不是None，即使发生异常。这最好使用`try`...`finally`语句完成。

*自从版本3.3开始不推荐使用：*如果之前与之联合`imp.find_module()`使用`importlib.import_module()`，则考虑使用，否则请使用您选择的替换件返回的加载程序`imp.find_module()`。如果你打电话`imp.load_module()`直接与文件路径参数和相关功能，然后使用的组合`importlib.util.spec_from_file_location()`和`importlib.util.module_from_spec()`。有关各种方法的详细信息，请参阅文档的“示例”部分`importlib`。

imp.new_module(*名字*)

返回一个名为*name*的新的空模块对象。该对象*未被*插入`sys.modules`。

自3.4版弃用：`importlib.util.module_from_spec()`改为使用。

imp.reload(*模块*)

重新加载以前导入的*模块*。参数必须是模块对象，所以它必须在之前成功导入。如果您已经使用外部编辑器编辑了模块源文件，并且希望在不离开Python解释器的情况下尝试新版本，这将非常有用。返回值是模块对象（与*模块*参数相同）。

何时`reload(module)`执行：

- 重新编译Python模块的代码并重新执行模块级代码，定义一组绑定到模块字典中名称的新对象。`init`扩展模块的功能不是第二次调用。
- 与Python中的所有其他对象一样，旧对象只有在其引用计数降至零后才会回收。
- 模块名称空间中的名称将更新为指向任何新的或更改的对象。
- 对旧对象的其他引用（例如模块外部的名称）不会反弹以引用新对象，并且如果需要，则必须在它们出现的每个命名空间中进行更新。

还有一些其他警告：

当模块重新加载时，其字典（包含模块的全局变量）将被保留。名称的重新定义将覆盖旧的定义，所以这通常不是问题。如果新版本的模块未定义旧版本定义的名称，则旧定义将保留。如果该模块维护全局表或对象缓存，则该特性可用于模块的优势 - 如果需要`try`，可使用语句测试表的存在并跳过其初始化：

```
try:
    cache
except NameError:
    cache = {}
```

它是合法的，虽然一般没有多大用处重装内置或动态加载的模块，除了sys，__main__和builtins。然而，在许多情况下，扩展模块并非设计为多次初始化，并且在重新加载时可能会以任意方式失败。

如果一个模块使用from..... 从另一个模块导入对象import，调用reload()另一个模块不会重新定义从其导入的对象 - 解决这个问题的方法之一是重新执行from语句，另一个是使用import限定名称（*module*。* name *）。

如果一个模块实例化一个类的实例，重新加载定义类的模块不会影响实例的方法定义 - 它们继续使用旧的类定义。派生类也是如此。

*在版本3.3中进行了更改：*依赖于两者__name__，__loader__并且在要重新加载的模块上定义而不是仅仅定义__name__。

*自3.4版弃用：*importlib.reload()改为使用。

以下功能便于处理 **PEP 3147**字节编译的文件路径。

3.2版本中的新功能

imp.cache_from_source（*path*，*debug_override* = *None* ）

返回 与源*路径*关联的字节编译文件的 **PEP 3147** *路径*。例如，如果 *path*是Python 3.2/foo/bar/baz.py的返回值/foo/bar/__pycache__/baz.cpython-32.pyc。该cpython-32字符串来自当前的魔术标签（请参阅 get_tag();如果sys.implementation.cache_tag未定义，NotImplementedError则会引发）。通过传入True或 False为 *debug_override*，您可以覆盖系统的值 __debug__，从而导致优化的字节码。

*路径*不需要存在。

*在版本3.3中更改：*如果sys.implementation.cache_tag是None，则会 NotImplementedError引发。

*自3.4版弃用：*importlib.util.cache_from_source()改为使用。

*改变在3.5版本：*该*debug_override*参数不再创建一个.pyo文件。

imp.source_from_cache（*路径*）

鉴于*通向* a**PEP 3147**文件名，返回关联的源代码文件路径。例如，如果*路径*是/foo/bar/__pycache__/baz.cpython-32.pyc返回的路径 /foo/bar/baz.py。 *路径*不需要存在，但是如果它不符合 **PEP 3147** 格式，aValueError 被提出。如果sys.implementation.cache_tag没有定义， NotImplementedError则提出。

*在版本3.3中更改：*未定义NotImplementedError时 提升sys.implementation.cache_tag。

*自3.4版弃用：*importlib.util.source_from_cache()改为使用。

imp.get_tag()

返回 **PEP 3147**魔术标记字符串与此版本的Python幻数匹配，如同返回get_magic()。

*自3.4版*sys.implementation.cache_tag开始*弃用：*直接从Python 3.3开始使用。

以下功能有助于与导入系统的内部锁定机制进行交互。锁定导入语义是一个实现细节，可能因发布版本而异。但是，Python确保循环导入没有任何死锁。

imp.lock_held()

返回，True如果全局导入锁定当前被保持，否则 False。在没有线程的平台上，总是返回False。

在具有线程的平台上，执行导入的线程首先保存全局导入锁，然后为其余导入设置每模块锁。这会阻止其他线程导入相同模块，直到原始导入完成，从而阻止其他线程查看由原始线程构建的不完整模块对象。循环导入是一个例外，它通过构建必须在某个时刻暴露一个不完整的模块对象。

*版本3.3中*已更改*：*大多数情况下，锁定方案已更改为每个模块锁定。全局导入锁定用于某些关键任务，例如初始化每个模块的锁定。

自3.4版以来已弃用。

imp.acquire_lock()

获取当前线程的解释程序全局导入锁。导入模块时应该使用此锁定导入钩子以确保线程安全。

一旦一个线程获得了导入锁定，同一个线程可以再次获得它而不会阻塞; 线程每次获取它时都必须释放一次。

在没有线程的平台上，这个函数什么都不做。

*版本3.3中*已更改*：*大多数情况下，锁定方案已更改为每个模块锁定。全局导入锁定用于某些关键任务，例如初始化每个模块的锁定。

自3.4版以来已弃用。

imp.release_lock()

释放解释器的全局导入锁定。在没有线程的平台上，这个函数什么都不做。

*版本3.3中*已更改*：*大多数情况下，锁定方案已更改为每个模块锁定。全局导入锁定用于某些关键任务，例如初始化每个模块的锁定。

自3.4版以来已弃用。

以下在本模块中定义的具有整数值的常量用于指示搜索结果find_module()。

imp.PY_SOURCE

该模块被发现为源文件。

自3.3版以来已弃用。

imp.PY_COMPILED
该模块被发现是一个编译的代码对象文件。

自3.3版以来已弃用。

imp.C_EXTENSION
该模块被发现是可动态加载的共享库。

自3.3版以来已弃用。

imp.PKG_DIRECTORY
该模块被发现为一个软件包目录。

自3.3版以来已弃用。

imp.C_BUILTIN
该模块被发现是一个内置模块。

自3.3版以来已弃用。

imp.PY_FROZEN
该模块被发现为冻结模块。

自3.3版以来已弃用。

class imp.NullImporter (*path_string*)
该NullImporter类型是**PEP 302**导入挂钩通过未能找到任何模块来处理非目录路径字符串。用现有目录或空字符串调用此类型引发ImportError。否则，NullImporter返回一个实例。

实例只有一个方法：

find_module (*fullname* [, *path*])
此方法始终返回None，表示找不到请求的模块。

*在版本3.3中更改：*None插入sys.path_importer_cache而不是实例NullImporter。

*自从3.4版本不推荐使用：*插入None到sys.path_importer_cache代替。

36.2.1。示例

以下函数模拟Python 1.4以前的标准导入语句（没有层次模块名称）。（这个*实现*在那个版本中不起作用，因为find_module()已经被扩展并且 load_module()已经被添加到1.4中。）

```
import imp
import sys

def __import__(name, globals=None, locals=None, fromlist=None):
    # Fast path: see if the module has already been imported.
    try:
```

```
        return sys.modules[name]
    except KeyError:
        pass

    # If any of the following calls raises an exception,
    # there's a problem we can't handle -- let the caller handle it.

    fp, pathname, description = imp.find_module(name)

    try:
        return imp.load_module(name, fp, pathname, description)
    finally:
        # Since we may exit via an exception, close fp explicitly.
        if fp:
            fp.close()
```

37.未记录的模块

以下是目前尚未记录的模块快速列表，但应记录在案。随时为他们提供文档！（通过电子邮件发送到文档@蟒蛇。组织。）

本章的想法和原创内容来自Fredrik Lundh的发布; 本章的具体内容已经大幅修改。

37.1。平台特定模块

这些模块用于实现该os.path模块，除此之外没有记录。几乎没有必要记录这些。

ntpath
    - os.path在Win32和Win64平台上实现。

posixpath
    - os.path在POSIX上实施。