Java 面试题解惑系列

作者：臧圩人

系列文章更新地址：

http://zangweiren.javaeye.com/

本文档更新至该系列中的第十一章，该系列文章仍在更新，由于文档制作存在一定滞后性，如需跟进最新系列文章请进入以上地址查看

制作时间：2008-09-11

JAVA 面试题解惑系列

（一）类的初始化顺序

关键字: java 面试题 初始化 发布时间：2008-06-26
作者：臧圩人（zangweiren）
网址：http://zangweiren.javaeye.com

>>>转载请注明出处！<<<

大家在去参加面试的时候，经常会遇到这样的考题：给你两个类的代码，它们之间是继承的关系，每个类里只有构造器方法和一些变量，构造器里可能还有一段代码对变量值进行了某种运算，另外还有一些将变量值输出到控制台的代码，然后让我们判断输出的结果。这实际上是在考查我们对于继承情况下类的初始化顺序的了解。

我们大家都知道，对于静态变量、静态初始化块、变量、初始化块、构造器，它们的初始化顺序依次是（静态变量、静态初始化块）>（变量、初始化块）>构造器。我们也可以通过下面的测试代码来验证这一点：
Java 代码

```
public class InitialOrderTest {

        // 静态变量
        public static String staticField = "静态变量";
        // 变量
        public String field = "变量";

        // 静态初始化块
        static {
            System.out.println(staticField);
            System.out.println("静态初始化块");
        }

        // 初始化块
        {
            System.out.println(field);
            System.out.println("初始化块");
        }

        // 构造器
        public InitialOrderTest() {
            System.out.println("构造器");
        }

        public static void main(String[] args) {
            new InitialOrderTest();
```

```
        }
    }
```

运行以上代码，我们会得到如下的输出结果：

1. 静态变量
2. 静态初始化块
3. 变量
4. 初始化块
5. 构造器

这与上文中说的完全符合。那么对于继承情况下又会怎样呢？我们仍然以一段测试代码来获取最终结果：
Java 代码 ：

```
class Parent {
            // 静态变量
            public static String p_StaticField = "父类--静态变量";
            // 变量
            public String p_Field = "父类--变量";

            // 静态初始化块
            static {
                  System.out.println(p_StaticField);
                System.out.println("父类--静态初始化块");
            }

            // 初始化块
            {
                System.out.println(p_Field);
                System.out.println("父类--初始化块");
            }

            // 构造器
            public Parent() {
                System.out.println("父类--构造器");
            }
    }

public class SubClass extends Parent {
            // 静态变量
            public static String s_StaticField = "子类--静态变量";
            // 变量
            public String s_Field = "子类--变量";
```

```
        // 静态初始化块
        static {
            System.out.println(s_StaticField);
            System.out.println("子类--静态初始化块");
        }
        // 初始化块
        {
            System.out.println(s_Field);
            System.out.println("子类--初始化块");
        }

        // 构造器
        public SubClass() {
            System.out.println("子类--构造器");
        }

        // 程序入口
        public static void main(String[] args) {
            new SubClass();
        }
    }
```

运行一下上面的代码，结果马上呈现在我们的眼前：

1. 父类--静态变量
2. 父类--静态初始化块
3. 子类--静态变量
4. 子类--静态初始化块
5. 父类--变量
6. 父类--初始化块
7. 父类--构造器
8. 子类--变量
9. 子类--初始化块
10. 子类--构造器

现在，结果已经不言自明了。大家可能会注意到一点，那就是，并不是父类完全初始化完毕后才进行子类的初始化，实际上子类的静态变量和静态初始化块的初始化是在父类的变量、初始化块和构造器初始化之前就完成了。

那么对于静态变量和静态初始化块之间、变量和初始化块之间的先后顺序又是怎样呢？是否静态变量总是先于静态初始化块，变量总是先于初始化块就被初始化了呢？实际上这取决于它们在类中出现的先后顺序。我们以静态变量和静态初始化块为例来进行说明。

同样，我们还是写一个类来进行测试：

Java 代码

```
public class TestOrder {
	// 静态变量
	public static TestA a = new TestA();

	// 静态初始化块
	static {
		System.out.println("静态初始化块");
	}

	// 静态变量
	public static TestB b = new TestB();

	public static void main(String[] args) {
		new TestOrder();
	}
}

class TestA {
	public TestA() {
		System.out.println("Test--A");
	}
}

class TestB {
	public TestB() {
		System.out.println("Test--B");
	}
}
```

运行上面的代码，会得到如下的结果：

1. Test--A
2. 静态初始化块
3. Test--B

大家可以随意改变变量 a、变量 b 以及静态初始化块的前后位置，就会发现输出结果随着它们在类中出现的前后顺序而改变，这就说明静态变量和静态初始化块是依照他们在类中的定义顺序进行初始化的。同样，变量和初始化块也遵循这个规律。

了解了继承情况下类的初始化顺序之后，如何判断最终输出结果就迎刃而解了。

（二）到底创建了几个 String 对象？

关键字: java 面试题 string 创建几个对象

作者：臧圩人（zangweiren） 发布时间：2008-06-30

网址：http://zangweiren.javaeye.com

我们首先来看一段代码：

Java 代码 ：

```
String str=new String("abc");
```

紧接着这段代码之后的往往是这个问题，那就是这行代码究竟创建了几个 String 对象呢？相信大家对这道题并不陌生，答案也是众所周知的，2 个。接下来我们就从这道题展开，一起回顾一下与创建 String 对象相关的一些 JAVA 知识。

我们可以把上面这行代码分成 String str、=、"abc"和 new String()四部分来看待。String str 只是定义了一个名为 str 的 String 类型的变量，因此它并没有创建对象；=是对变量 str 进行初始化，将某个对象的引用（或者叫句柄）赋值给它，显然也没有创建对象；现在只剩下 new String("abc")了。那么，new String("abc")为什么又能被看成"abc"和 new String()呢？我们来看一下被我们调用了的 String 的构造器：

Java 代码

```
public String(String original) {
        //other code ...
}
```

大家都知道，我们常用的创建一个类的实例（对象）的方法有以下两种：

1. 使用 new 创建对象。

2. 调用 Class 类的 newInstance 方法，利用反射机制创建对象。

我们正是使用 new 调用了 String 类的上面那个构造器方法创建了一个对象，并将它的引用赋值给了 str 变量。同时我们注意到，被调用的构造器方法接受的参数也是一个 String 对象，这个对象正是"abc"。由此我们又要引入另外一种创建 String 对象的方式的讨论——引号内包含文本。

这种方式是 String 特有的，并且它与 new 的方式存在很大区别。

Java 代码

```
String str="abc";
```

毫无疑问，这行代码创建了一个 String 对象。

Java 代码

```
String a="abc";
String b="abc";
```

那这里呢？答案还是一个。

Java 代码

```
String a="ab"+"cd";
```

再看看这里呢？答案仍是一个。有点奇怪吗？说到这里，我们就需要引入对字符串池相关知识的回顾了。

在 JAVA 虚拟机（JVM）中存在着一个字符串池，其中保存着很多 String 对象，并且可以被共享使用，因此它提高了效率。由于 String 类是 final 的，它的值一经创建就不可改变，因此我们不用担心 String 对象共享而带来程序的混乱。字符串池由 String 类维护，我们可以调用 intern()方法来访问字符串池。

我们再回头看看 String a="abc";，这行代码被执行的时候，JAVA 虚拟机首先在字符串池中查找是否已经存在了值为"abc"的这么一个对象，它的判断依据是 String 类 equals(Object obj)方法的返回值。如果有，则不再创建新的对象，直接返回已存在对象的引用；如果没有，则先创建这个对象，然后把它加入到字符串池中，再将它的引用返回。因此，我们不难理解前面三个例子中头两个例子为什么是这个答案了。

对于第三个例子：

Java 代码

```
String a="ab"+"cd";
```

由于常量的值在编译的时候就被确定了。在这里，"ab"和"cd"都是常量，因此变量 a 的值在编译时就可以确定。这行代码编译后的效果等同于：

Java 代码

```
String a="abcd";
```

因此这里只创建了一个对象"abcd"，并且它被保存在字符串池里了。

现在问题又来了，是不是所有经过“+”连接后得到的字符串都会被添加到字符串池中呢？我们都知道“==”可以用来比较两个变量，它有以下两种情况：

1. 如果比较的是两个基本类型（char，byte，short，int，long，float，double，boolean），则是判断它们的值是否相等。
2. 如果表较的是两个对象变量，则是判断它们的引用是否指向同一个对象。

下面我们就用“==”来做几个测试。为了便于说明，我们把指向字符串池中已经存在的对象也视为该对象被加入了字符串池：

Java 代码

```
public class StringTest {
	public static void main(String[] args) {
		String a = "ab";// 创建了一个对象，并加入字符串池中
		System.out.println("String a = \"ab\";");
		String b = "cd";// 创建了一个对象，并加入字符串池中
		System.out.println("String b = \"cd\";");
		String c = "abcd";// 创建了一个对象，并加入字符串池中

		String d = "ab" + "cd";
		// 如果 d 和 c 指向了同一个对象，则说明 d 也被加入了字符串池
		if (d == c) {
			System.out.println("\"ab\"+\"cd\" 创建的对象 \"加入了\" 字符串池中");
		}
		// 如果 d 和 c 没有指向了同一个对象，则说明 d 没有被加入字符串池
		else {
			System.out.println("\"ab\"+\"cd\" 创建的对象 \"没加入\" 字符串池中");
		}

		String e = a + "cd";
		// 如果 e 和 c 指向了同一个对象，则说明 e 也被加入了字符串池
		if (e == c) {
			System.out.println(" a  +\"cd\" 创建的对象 \"加入了\" 字符串池中");
		}
		// 如果 e 和 c 没有指向了同一个对象，则说明 e 没有被加入字符串池
```

```
            else {
                System.out.println(" a  +\"cd\" 创建的对象  \"没加入\" 字符串池中");
            }

            String f = "ab" + b;
            // 如果 f 和 c 指向了同一个对象，则说明 f 也被加入了字符串池
            if (f == c) {
                System.out.println("\"ab\"+ b   创建的对象  \"加入了\" 字符串池中");
            }
            // 如果 f 和 c 没有指向了同一个对象，则说明 f 没有被加入字符串池
            else {
                System.out.println("\"ab\"+ b   创建的对象  \"没加入\" 字符串池中");
            }

            String g = a + b;
            // 如果 g 和 c 指向了同一个对象，则说明 g 也被加入了字符串池
            if (g == c) {
                System.out.println(" a  + b   创建的对象  \"加入了\" 字符串池中");
            }
            // 如果 g 和 c 没有指向了同一个对象，则说明 g 没有被加入字符串池
            else {
                System.out.println(" a  + b   创建的对象  \"没加入\" 字符串池中");
            }
        }
}
```

运行结果如下：

1. String a = "ab";
2. String b = "cd";
3. "ab"+"cd" 创建的对象 "加入了" 字符串池中
4. a +"cd" 创建的对象 "没加入" 字符串池中
5. "ab"+ b 创建的对象 "没加入" 字符串池中
6. a + b 创建的对象 "没加入" 字符串池中

从上面的结果中我们不难看出，只有使用引号包含文本的方式创建的 String 对象之间使用“+”连接产生的新对象才会被加入字符串池中。对于所有包含 new 方式新建对象（包括 null）的“+”连接表达式，它所产生的新对象都不会被加入字符串池中，对此我们不再赘述。

但是有一种情况需要引起我们的注意。请看下面的代码：

Java 代码

```
public class StringStaticTest {
	// 常量 A
	public static final String A = "ab";

	// 常量 B
	public static final String B = "cd";

	public static void main(String[] args) {
		// 将两个常量用+连接对 s 进行初始化
		String s = A + B;
		String t = "abcd";
		if (s == t) {
			System.out.println("s 等于 t，它们是同一个对象");
		} else {
			System.out.println("s 不等于 t，它们不是同一个对象");
		}
	}
}
```

这段代码的运行结果如下：

- s 等于 t，它们是同一个对象

这又是为什么呢？原因是这样的，对于常量来讲，它的值是固定的，因此在编译期就能被确定了，而变量的值只有到运行时才能被确定，因为这个变量可以被不同的方法调用，从而可能引起值的改变。在上面的例子中，A 和 B 都是常量，值是固定的，因此 s 的值也是固定的，它在类被编译时就已经确定了。也就是说：

Java 代码

```
String s=A+B;
```

等同于：

Java 代码

```
String s="ab"+"cd";
```

我对上面的例子稍加改变看看会出现什么情况：

Java 代码

```
public class StringStaticTest {
	// 常量 A
	public static final String A;

	// 常量 B
	public static final String B;

	static {
		A = "ab";
		B = "cd";
	}

	public static void main(String[] args) {
		// 将两个常量用+连接对 s 进行初始化
		String s = A + B;
		String t = "abcd";
		if (s == t) {
			System.out.println("s 等于 t，它们是同一个对象");
		} else {
			System.out.println("s 不等于 t，它们不是同一个对象");
		}
	}
}
```

它的运行结果是这样：

- s 不等于 t，它们不是同一个对象

只是做了一点改动，结果就和刚刚的例子恰好相反。我们再来分析一下。A 和 B 虽然被定义为常量（只能被赋值一次），但是它们都没有马上被赋值。在运算出 s 的值之前，他们何时被赋值，以及被赋予什么样的值，都是个变数。因此 A 和 B 在被赋值之前，性质类似于一个变量。那么 s 就不能在编译期被确定，而只能在运行时被创建了。

由于字符串池中对象的共享能够带来效率的提高，因此我们提倡大家用引号包含文本的方式来创建 String 对象，实际上这也是我们在编程中常采用的。

接下来我们再来看看 intern()方法，它的定义如下：

Java 代码

```
public native String intern();
```

这是一个本地方法。在调用这个方法时，JAVA 虚拟机首先检查字符串池中是否已经存在与该对象值相等对象存在，如果有则返回字符串池中对象的引用；如果没有，则先在字符串池中创建一个相同值的 String 对象，然后再将它的引用返回。

我们来看这段代码：

Java 代码

```
public class StringInternTest {
	public static void main(String[] args) {
		// 使用 char 数组来初始化 a，避免在 a 被创建之前字符串池中已经存在了值为"abcd"的对象
		String a = new String(new char[] { 'a', 'b', 'c', 'd' });
		String b = a.intern();
		if (b == a) {
			System.out.println("b 被加入了字符串池中，没有新建对象");
		} else {
			System.out.println("b 没被加入字符串池中，新建了对象");
		}
	}
}
```

运行结果：

- b 没被加入字符串池中，新建了对象

如果 String 类的 intern()方法在没有找到相同值的对象时，是把当前对象加入字符串池中，然后返回它的引用的话，那么 b 和 a 指向的就是同一个对象；否则 b 指向的对象就是 JAVA 虚拟机在字符串池中新建的，只是它的值与 a 相同罢了。上面这段代码的运行结果恰恰印证了这一点。

最后我们再来说说 String 对象在 JAVA 虚拟机（JVM）中的存储，以及字符串池与堆（heap）和栈（stack）的关系。我们首先回顾一下堆和栈的区别：

- 栈（stack）：主要保存基本类型（或者叫内置类型）（char、byte、short、int、long、float、double、boolean）和对象的引用，数据可以共享，速度仅次于寄存器（register），快于堆。
- 堆（heap）：用于存储对象。

我们查看 String 类的源码就会发现，它有一个 value 属性，保存着 String 对象的值，类型是 char[]，这也正说明了字符串就是字符的序列。

当执行 String a="abc";时，JAVA 虚拟机会在栈中创建三个 char 型的值'a'、'b'和'c'，然后在堆中创建一个 String 对象，它的值（value）是刚才在栈中创建的三个 char 型值组成的数组{'a','b','c'}，最后这个新创建的 String 对象会被添加到字符串池中。如果我们接着执行 String b=new String("abc");代码，由于"abc"已经被创建并保存于字符串池中，因此 JAVA 虚拟机只会在堆中新创建一个 String 对象，但是它的值（value）是共享前一行代码执行时在栈中创建的三个 char 型值值'a'、'b'和'c'。

说到这里，我们对于篇首提出的 String str=new String("abc")为什么是创建了两个对象这个问题就已经相当明了了。

（三）变量（属性）的覆盖

关键字: java 面试题 继承 变量的覆盖 属性

作者：臧圩人（zangweiren） 发布时间：2008-07-03

网址：http://zangweiren.javaeye.com

>>>转载请注明出处！<<<

我们来看看这么一道题：

Java 代码

```
class ParentClass {
	public int i = 10;
}

public class SubClass extends ParentClass {
	public int i = 30;

	public static void main(String[] args) {
		ParentClass parentClass = new SubClass();
		SubClass subClass = new SubClass();
		System.out.println(parentClass.i + subClass.i);
	}
}
```

控制台的输出结果是多少呢？20？40？还是 60？

变量，或者叫做类的属性，在继承的情况下，如果父类和子类存在同名的变量会出现什么情况呢？这就是这道题要考查的知识点——变量（属性）的覆盖。

这个问题虽然简单，但是情况却比较复杂。因为我们不仅要考虑变量、静态变量和常量三种情况，还要考虑 private、friendly（即不加访问修饰符）、protected 和 public 四种访问权限下对属性的不同影响。

我们先从普通变量说起。依照我们的惯例，先来看一段代码：

Java 代码

```
class ParentClass {
	private String privateField = "父类变量--private";

	/* friendly */String friendlyField = "父类变量--friendly";

	protected String protectedField = "父类变量--protected";

	public String publicField = "父类变量--public";

	// private 的变量无法直接访问，因此我们给他增加了一个访问方法
```

```
	public String getPrivateFieldValue() {
		return privateField;
	}
}

public class SubClass extends ParentClass {
	private String privateField = "子类变量--private";

	/* friendly */String friendlyField = "子类变量--friendly";

	protected String protectedField = "子类变量--protected";

	public String publicField = "子类变量--public";

	// private 的变量无法直接访问，因此我们给他增加了一个访问方法
	public String getPrivateFieldValue() {
		return privateField;
	}

	public static void main(String[] args) {
		// 为了便于查阅，我们统一按照 private、friendly、protected、public 的顺序
		// 输出下列三种情况中变量的值

		// ParentClass 类型，ParentClass 对象
		ParentClass parentClass = new ParentClass();
		System.out.println("ParentClass parentClass = new ParentClass();");
		System.out.println(parentClass.getPrivateFieldValue());
		System.out.println(parentClass.friendlyField);
		System.out.println(parentClass.protectedField);
		System.out.println(parentClass.publicField);

		System.out.println();

		// ParentClass 类型，SubClass 对象
		ParentClass subClass = new SubClass();
```

```
                System.out.println("ParentClass subClass = new SubClass();");
                System.out.println(subClass.getPrivateFieldValue());
                System.out.println(subClass.friendlyField);
                System.out.println(subClass.protectedField);
                System.out.println(subClass.publicField);

                System.out.println();

                // SubClass 类型，SubClass 对象
                SubClass subClazz = new SubClass();
                System.out.println("SubClass subClazz = new SubClass();");
                System.out.println(subClazz.getPrivateFieldValue());
                System.out.println(subClazz.friendlyField);
                System.out.println(subClazz.protectedField);
                System.out.println(subClazz.publicField);
        }
}
```

这段代码的运行结果如下：

1. ParentClass parentClass = new ParentClass();
2. 父类变量--private
3. 父类变量--friendly
4. 父类变量--protected
5. 父类变量--public
6.
7. ParentClass subClass = new SubClass();
8. 子类变量--private
9. 父类变量--friendly
10. 父类变量--protected
11. 父类变量--public
12.
13. SubClass subClazz = new SubClass();
14. 子类变量--private
15. 子类变量--friendly

16. 子类变量--protected
17. 子类变量--public

从上面的结果中可以看出，private 的变量与其它三种访问权限变量的不同，这是由于方法的重写（override）而引起的。关于重写知识的回顾留给以后的章节，这里我们来看一下其它三种访问权限下变量的覆盖情况。

分析上面的输出结果就会发现，变量的值取决于我们定义的变量的类型，而不是创建的对象的类型。

在上面的例子中，同名的变量访问权限也是相同的，那么对于名称相同但是访问权限不同的变量，情况又会怎样呢？事实胜于雄辩，我们继续来做测试。由于 private 变量的特殊性，在接下来的实验中我们都把它排除在外，不予考虑。

由于上面的例子已经说明了，当变量类型是父类（ParentClass）时，不管我们创建的对象是父类（ParentClass）的还是子类（SubClass）的，都不存在属性覆盖的问题，因此接下来我们也只考虑变量类型和创建对象都是子类（SubClass）的情况。

Java 代码

```
class ParentClass {
	/* friendly */String field = "父类变量";
}

public class SubClass extends ParentClass {
	protected String field = "子类变量";

	public static void main(String[] args) {
		SubClass subClass = new SubClass();
		System.out.println(subClass.field);
	}
}
```

运行结果：

- 子类变量

Java 代码

```
class ParentClass {
        public String field = "父类变量";
}

public class SubClass extends ParentClass {
        protected String field = "子类变量";

        public static void main(String[] args) {
                SubClass subClass = new SubClass();
                System.out.println(subClass.field);
        }
}
```

运行结果：

- 子类变量

上面两段不同的代码，输出结果确是相同的。事实上，我们可以将父类和子类属性前的访问修饰符在 friendly、protected 和 public 之间任意切换，得到的结果都是相同的。也就是说访问修饰符并不影响属性的覆盖，关于这一点大家可以自行编写测试代码验证。

对于静态变量和常量又会怎样呢？我们继续来看：

Java 代码

```
class ParentClass {
        public static String staticField = "父类静态变量";

        public final String finalField = "父类常量";

        public static final String staticFinalField = "父类静态常量";
}

public class SubClass extends ParentClass {
        public static String staticField = "子类静态变量";

        public final String finalField = "子类常量";
```

```
        public static final String staticFinalField = "子类静态常量";

        public static void main(String[] args) {
                SubClass subClass = new SubClass();
                System.out.println(SubClass.staticField);
                // 注意，这里的 subClass 变量，不是 SubClass 类
                System.out.println(subClass.finalField);
                System.out.println(SubClass.staticFinalField);
        }
}
```

运行结果如下：

1. 子类静态变量
2. 子类常量
3. 子类静态常量

虽然上面的结果中包含“子类静态变量”和“子类静态常量”，但这并不表示父类的“静态变量”和“静态常量”可以被子类覆盖，因为它们都是属于类，而不属于对象。

上面的例子中，我们一直用对象来对变量（属性）的覆盖做测试，如果是基本类型的变量，结果是否会相同呢？答案是肯定的，这里我们就不再一一举例说明了。

最后，我们来做个总结。通过以上测试，可以得出一下结论：

1. 由于 private 变量受访问权限的限制，它不能被覆盖。
2. 属性的值取父类还是子类并不取决于我们创建对象的类型，而是取决于我们定义的变量的类型。
3. friendly、protected 和 public 修饰符并不影响属性的覆盖。
4. 静态变量和静态常量属于类，不属于对象，因此它们不能被覆盖。
5. 常量可以被覆盖。
6. 对于基本类型和对象，它们适用同样的覆盖规律。

我们再回到篇首的那道题，我想大家都已经知道答案了，输出结果应该是 40。

（四）final、finally 和 finalize 的区别

关键字: java 面试题 final finally finalize

作者：臧圩人（zangweiren）发布时间：2008-07-08

网址：http://zangweiren.javaeye.com

>>>转载请注明出处！<<<

final、finally 和 finalize 的区别是什么？

这是一道再经典不过的面试题了，我们在各个公司的面试题中几乎都能看到它的身影。final、finally 和 finalize 虽然长得像孪生三兄弟一样，但是它们的含义和用法却是大相径庭。这一次我们就一起来回顾一下这方面的知识。

final 关键字

我们首先来说说 final。它可以用于以下四个地方：

1. 定义变量，包括静态的和非静态的。
2. 定义方法的参数。
3. 定义方法。
4. 定义类。

我们依次来回顾一下每种情况下 final 的作用。首先来看第一种情况，如果 final 修饰的是一个基本类型，就表示这个变量被赋予的值是不可变的，即它是个常量；如果 final 修饰的是一个对象，就表示这个变量被赋予的引用是不可变的，这里需要提醒大家注意的是，不可改变的只是这个变量所保存的引用，并不是这个引用所指向的对象。在第二种情况下，final 的含义与第一种情况相同。实际上对于前两种情况，有一种更贴切的表述 final 的含义的描述，那就是，如果一个变量或方法参数被 final 修饰，就表示它只能被赋值一次，但是 JAVA 虚拟机为变量设定的默认值不记作一次赋值。

被 final 修饰的变量必须被初始化。初始化的方式有以下几种：

1. 在定义的时候初始化。
2. final 变量可以在初始化块中初始化，不可以在静态初始化块中初始化。
3. 静态 final 变量可以在静态初始化块中初始化，不可以在初始化块中初始化。
4. final 变量还可以在类的构造器中初始化，但是静态 final 变量不可以。

通过下面的代码可以验证以上的观点：

Java 代码

```
public class FinalTest {
	// 在定义时初始化
	public final int A = 10;

	public final int B;
	// 在初始化块中初始化
	{
		B = 20;
	}

	// 非静态 final 变量不能在静态初始化块中初始化
	// public final int C;
	// static {
	// C = 30;
	// }

	// 静态常量，在定义时初始化
	public static final int STATIC_D = 40;

	public static final int STATIC_E;
	// 静态常量，在静态初始化块中初始化
	static {
		STATIC_E = 50;
	}

	// 静态变量不能在初始化块中初始化
	// public static final int STATIC_F;
```

```
        // {
        // STATIC_F = 60;
        // }

        public final int G;

        // 静态 final 变量不可以在构造器中初始化
        // public static final int STATIC_H;

        // 在构造器中初始化
        public FinalTest() {
                G = 70;
                // 静态 final 变量不可以在构造器中初始化
                // STATIC_H = 80;

                // 给 final 的变量第二次赋值时，编译会报错
                // A = 99;
                // STATIC_D = 99;
        }

        // final 变量未被初始化，编译时就会报错
        // public final int I;

        // 静态 final 变量未被初始化，编译时就会报错
        // public static final int STATIC_J;
}
```

我们运行上面的代码之后出了可以发现 final 变量（常量）和静态 final 变量（静态常量）未被初始化时，编译会报错。

用 final 修饰的变量（常量）比非 final 的变量（普通变量）拥有更高的效率，因此我们在实际编程中应该尽可能多的用常量来代替普通变量，这也是一个很好的编程习惯。

当 final 用来定义一个方法时，会有什么效果呢？正如大家所知，它表示这个方法不可以被子类重写，但是它这不影响它被子类继承。我们写段代码来验证一下：

Java 代码

```
class ParentClass {
	public final void TestFinal() {
		System.out.println("父类--这是一个 final 方法");
	}
}

public class SubClass extends ParentClass {
	/**
	 * 子类无法重写（override）父类的 final 方法，否则编译时会报错
	 */
	// public void TestFinal() {
	// System.out.println("子类--重写 final 方法");
	// }

	public static void main(String[] args) {
		SubClass sc = new SubClass();
		sc.TestFinal();
	}
}
```

这里需要特殊说明的是，具有 private 访问权限的方法也可以增加 final 修饰，但是由于子类无法继承 private 方法，因此也无法重写它。编译器在处理 private 方法时，是按照 final 方法来对待的，这样可以提高该方法被调用时的效率。不过子类仍然可以定义同父类中的 private 方法具有同样结构的方法，但是这并不会产生重写的效果，而且它们之间也不存在必然联系。

最后我们再来回顾一下 final 用于类的情况。这个大家应该也很熟悉了，因为我们最常用的 String 类就是 final 的。由于 final 类不允许被继承，编译器在处理时把它的所有方法都当作 final 的，因此 final 类比普通类拥有更高的效率。而由关键字 abstract 定义的抽象类含有必须由继承自它的子类重载实现的抽象方法，因此无法同时用 final 和 abstract 来修饰同一个类。同样的道理，final 也不能用来修饰接口。 final 的类的所有方法都不能被重写，但这并不表示 final 的类的属性（变量）值也是不可改变的，要想做到 final 类的属性值不可改变，必须给它增加 final 修饰，请看下面的例子：

Java 代码

```
public final class FinalTest {

	int i = 10;
```

```
        public static void main(String[] args) {
                FinalTest ft = new FinalTest();
                ft.i = 99;
                System.out.println(ft.i);
        }
}
```

运行上面的代码试试看，结果是 99，而不是初始化时的 10。

finally 语句

接下来我们一起回顾一下 finally 的用法。这个就比较简单了，它只能用在 try/catch 语句中，并且附带着一个语句块，表示这段语句最终总是被执行。请看下面的代码：

Java 代码

```
public final class FinallyTest {
        public static void main(String[] args) {
                try {
                        throw new NullPointerException();
                } catch (NullPointerException e) {
                        System.out.println("程序抛出了异常");
                } finally {
                        System.out.println("执行了 finally 语句块");
                }
        }
}
```

运行结果说明了 finally 的作用：

1. 程序抛出了异常
2. 执行了 finally 语句块

请大家注意，捕获程序抛出的异常之后，既不加处理，也不继续向上抛出异常，并不是良好的编程习惯，它掩盖了程序执行中发生的错误，这里只是方便演示，请不要学习。

那么，有没有一种情况使 finally 语句块得不到执行呢？大家可能想到了 return、continue、break 这三个可以打乱代码顺序执行语句的规律。那我们就来试试看，这三个语句是否能影响 finally 语句块的执行：

Java 代码

```
public final class FinallyTest {

    // 测试 return 语句
    public ReturnClass testReturn() {
        try {
            return new ReturnClass();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            System.out.println("执行了 finally 语句");
        }
        return null;
    }

    // 测试 continue 语句
    public void testContinue() {
        for (int i = 0; i < 3; i++) {
            try {
                System.out.println(i);
                if (i == 1) {
                    continue;
                }
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                System.out.println("执行了 finally 语句");
            }
        }
    }

    // 测试 break 语句
    public void testBreak() {
```

```
            for (int i = 0; i < 3; i++) {
                try {
                    System.out.println(i);
                    if (i == 1) {
                        break;
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                } finally {
                    System.out.println("执行了 finally 语句");
                }
            }
        }

        public static void main(String[] args) {
            FinallyTest ft = new FinallyTest();
            // 测试 return 语句
            ft.testReturn();
            System.out.println();
            // 测试 continue 语句
            ft.testContinue();
            System.out.println();
            // 测试 break 语句
            ft.testBreak();
        }
}

class ReturnClass {
        public ReturnClass() {
            System.out.println("执行了 return 语句");
        }
}
```

上面这段代码的运行结果如下：

1. 执行了 return 语句

2. 执行了 finally 语句
3.
4. 0
5. 执行了 finally 语句
6. 1
7. 执行了 finally 语句
8. 2
9. 执行了 finally 语句
10.
11. 0
12. 执行了 finally 语句
13. 1
14. 执行了 finally 语句

很明显，return、continue 和 break 都没能阻止 finally 语句块的执行。从输出的结果来看，return 语句似乎在 finally 语句块之前执行了，事实真的如此吗？我们来想想看，return 语句的作用是什么呢？是退出当前的方法，并将值或对象返回。如果 finally 语句块是在 return 语句之后执行的，那么 return 语句被执行后就已经退出当前方法了，finally 语句块又如何能被执行呢？因此，正确的执行顺序应该是这样的：编译器在编译 return new ReturnClass();时，将它分成了两个步骤，new ReturnClass()和 return，前一个创建对象的语句是在 finally 语句块之前被执行的，而后一个 return 语句是在 finally 语句块之后执行的，也就是说 finally 语句块是在程序退出方法之前被执行的。同样，finally 语句块是在循环被跳过（continue）和中断（break）之前被执行的。

finalize 方法

最后，我们再来看看 finalize，它是一个方法，属于 java.lang.Object 类，它的定义如下：

Java 代码

```
protected void finalize() throws Throwable { }
```

众所周知，finalize()方法是 GC（garbage collector）运行机制的一部分，关于 GC 的知识我们将在后续的章节中来回顾。

在此我们只说说 finalize()方法的作用是什么呢？

finalize()方法是在 GC 清理它所从属的对象时被调用的，如果执行它的过程中抛出了无法捕

获的异常（uncaught exception），GC 将终止对改对象的清理，并且该异常会被忽略；直到下一次 GC 开始清理这个对象时，它的 finalize()会被再次调用。

请看下面的示例：

Java 代码

```
public final class FinallyTest {
        // 重写 finalize()方法
        protected void finalize() throws Throwable {
                System.out.println("执行了 finalize()方法");
        }

        public static void main(String[] args) {
                FinallyTest ft = new FinallyTest();
                ft = null;
                System.gc();
        }
}
```

运行结果如下：

- 执行了 finalize()方法

程序调用了 java.lang.System 类的 gc()方法，引起 GC 的执行，GC 在清理 ft 对象时调用了它的 finalize()方法，因此才有了上面的输出结果。调用 System.gc()等同于调用下面这行代码：

Java 代码

```
Runtime.getRuntime().gc();
```

调用它们的作用只是建议垃圾收集器（GC）启动，清理无用的对象释放内存空间，但是 GC 的启动并不是一定的，这由 JAVA 虚拟机来决定。直到 JAVA 虚拟机停止运行，有些对象的 finalize()可能都没有被运行过，那么怎样保证所有对象的这个方法在 JAVA 虚拟机停止运行之前一定被调用呢？答案是我们可以调用 System 类的另一个方法：

Java 代码

```
public static void runFinalizersOnExit(boolean value) {
        //other code
```

```
}
```

给这个方法传入 true 就可以保证对象的 finalize()方法在 JAVA 虚拟机停止运行前一定被运行了，不过遗憾的是这个方法是不安全的，它会导致有用的对象 finalize()被误调用，因此已经不被赞成使用了。

由于 finalize()属于 Object 类，因此所有类都有这个方法，Object 的任意子类都可以重写（override）该方法，在其中释放系统资源或者做其它的清理工作，如关闭输入输出流。

通过以上知识的回顾，我想大家对于 final、finally、finalize 的用法区别已经很清楚了。

（五）传了值还是传了引用？

关键字: java 面试题 值传递 引用传递

作者：臧圩人（zangweiren）发布时间：2008-07-13

网址：http://zangweiren.javaeye.com

>>>转载请注明出处！<<<

JAVA 中的传递都是值传递吗？有没有引用传递呢？

在回答这两个问题前，让我们首先来看一段代码：

Java 代码

```
public class ParamTest {
        // 初始值为 0
        protected int num = 0;

        // 为方法参数重新赋值
        public void change(int i) {
                i = 5;
        }

        // 为方法参数重新赋值
        public void change(ParamTest t) {
                ParamTest tmp = new ParamTest();
```

```
        tmp.num = 9;
        t = tmp;
    }

    // 改变方法参数的值
    public void add(int i) {
        i += 10;
    }

    // 改变方法参数属性的值
    public void add(ParamTest pt) {
        pt.num += 20;
    }

    public static void main(String[] args) {
        ParamTest t = new ParamTest();

        System.out.println("参数--基本类型");
        System.out.println("原有的值：" + t.num);
        // 为基本类型参数重新赋值
        t.change(t.num);
        System.out.println("赋值之后：" + t.num);
        // 为引用型参数重新赋值
        t.change(t);
        System.out.println("运算之后：" + t.num);

        System.out.println();

        t = new ParamTest();
        System.out.println("参数--引用类型");
        System.out.println("原有的值：" + t.num);
        // 改变基本类型参数的值
        t.add(t.num);
        System.out.println("赋引用后：" + t.num);
        // 改变引用类型参数所指向对象的属性值
```

```
                t.add(t);
                System.out.println("改属性后：" + t.num);
        }
}
```

这段代码的运行结果如下：

1. 参数--基本类型
2. 原有的值：0
3. 赋值之后：0
4. 运算之后：0
5.
6. 参数--引用类型
7. 原有的值：0
8. 赋引用后：0
9. 改属性后：20

从上面这个直观的结果中我们很容易得出如下结论：

1. 对于基本类型，在方法体内对方法参数进行重新赋值，并不会改变原有变量的值。
2. 对于引用类型，在方法体内对方法参数进行重新赋予引用，并不会改变原有变量所持有的引用。
3. 方法体内对参数进行运算，不影响原有变量的值。
4. 方法体内对参数所指向对象的属性进行运算，将改变原有变量所指向对象的属性值。

上面总结出来的不过是我们所看到的表面现象。那么，为什么会出现这样的现象呢？这就要说到值传递和引用传递的概念了。这个问题向来是颇有争议的。

大家都知道，在 JAVA 中变量有以下两种：

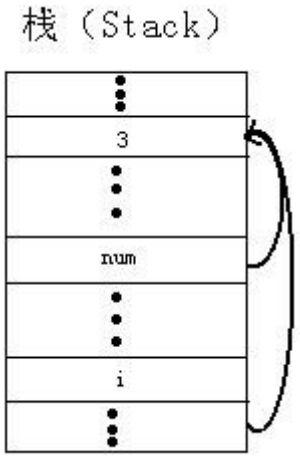
1. 基本类型变量，包括 char、byte、short、int、long、float、double、boolean。
2. 引用类型变量，包括类、接口、数组（基本类型数组和对象数组）。

当基本类型的变量被当作参数传递给方法时，JAVA 虚拟机所做的工作是把这个值拷贝了一份，然后把拷贝后的值传递到了方法的内部。因此在上面的例子中，我们回头来看看这个方法：

Java 代码

```
// 为方法参数重新赋值
public void change(int i) {
        i = 5;
}
```

在这个方法被调用时，变量 i 和 ParamTest 型对象 t 的属性 num 具有相同的值，却是两个不同变量。变量 i 是由 JAVA 虚拟机创建的作用域在 change(int i)方法内的局部变量，在这个方法执行完毕后，它的生命周期就结束了。在 JAVA 虚拟机中，它们是以类似如下的方式存储的：



很明显，在基本类型被作为参数传递给方式时，是值传递，在整个过程中根本没有牵扯到引用这个概念。这也是大家所公认的。对于布尔型变量当然也是如此，请看下面的例子：

Java 代码

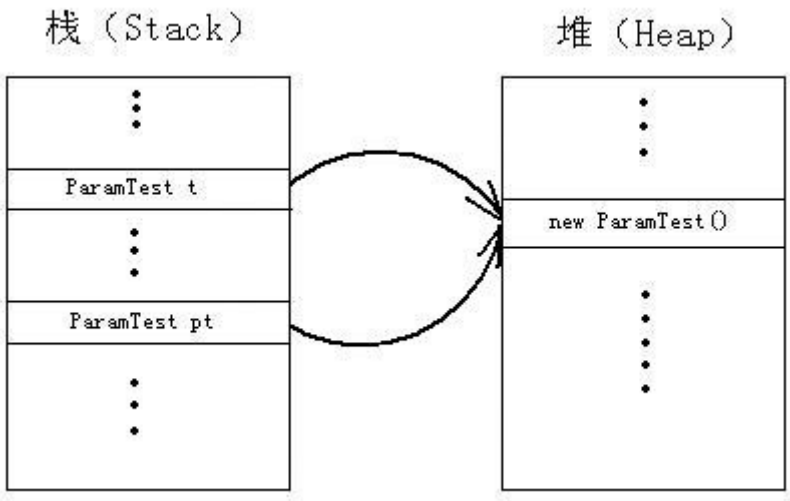
```
public class BooleanTest {
        // 布尔型值
        boolean bool = true;

        // 为布尔型参数重新赋值
        public void change(boolean b) {
                b = false;
        }

        // 对布尔型参数进行运算
        public void calculate(boolean b) {
                b = b && false;
                // 为了方便对比，将运算结果输出
                System.out.println("b 运算后的值：" + b);
        }

        public static void main(String[] args) {
```

```
        BooleanTest t = new BooleanTest();

        System.out.println("参数--布尔型");
        System.out.println("原有的值：" + t.bool);
        // 为布尔型参数重新赋值
        t.change(t.bool);
        System.out.println("赋值之后：" + t.bool);

        // 改变布尔型参数的值
        t.calculate(t.bool);
        System.out.println("运算之后：" + t.bool);
    }
}
```

输出结果如下：

1. 参数--布尔型
2. 原有的值：true
3. 赋值之后：true
4. b 运算后的值：false
5. 运算之后：true

那么当引用型变量被当作参数传递给方法时 JAVA 虚拟机又是怎样处理的呢？同样，它会拷贝一份这个变量所持有的引用，然后把它传递给 JAVA 虚拟机为方法创建的局部变量，从而这两个变量指向了同一个对象。在篇首所举的示例中，ParamTest 类型变量 t 和局部变量 pt 在 JAVA 虚拟机中是以如下的方式存储的：



有一种说法是当一个对象或引用类型变量被当作参数传递时，也是值传递，这个值就是对象的引用，因此 JAVA 中只有值传递，没有引用传递。还有一种说法是引用可以看作是对象的别名，当对象被当作参数传递给方法时，传递的是对象的引用，因此是引用传递。这两种观点各有支持者，但是前一种观点被绝大多数人所接受，其中有《Core Java》一书的作者，以及 JAVA 的创造者 James Gosling，而《Thinking in Java》一书的作者 Bruce Eckel 则站在了中立的立场上。

我个人认为值传递中的值指的是基本类型的数值，即使对于布尔型，虽然它的表现形式为 true 和 false，但是在栈中，它仍然是以数值形式保存的，即 0 表示 false，其它数值表示 true。而引用是我们用来操作对象的工具，它包含了对象在堆中保存地址的信息。即使在被作为参数传递给方法时，实际上传递的是它的拷贝，但那仍是引用。因此，用引用传递来区别与值传递，概念上更加清晰。

最后我们得出如下的结论：

1. 基本类型和基本类型变量被当作参数传递给方法时，是值传递。在方法实体中，无法给原变量重新赋值，也无法改变它的值。
2. 对象和引用型变量被当作参数传递给方法时，在方法实体中，无法给原变量重新赋值，但是可以改变它所指向对象的属性。至于到底它是值传递还是引用传递，这并不重要，重要的是我们要清楚当一个引用被作为参数传递给一个方法时，在这个方法体内会发生什么。

（六）字符串（String）杂谈

关键字: java 面试题 字符串 string

作者：臧圩人（zangweiren）发布时间：2008-07-18

网址：http://zangweiren.javaeye.com

上一次我们已经一起回顾了面试题中常考的到底创建了几个 String 对象的相关知识，这一次我们以几个常见面试题为引子，来回顾一下 String 对象相关的其它一些方面。

String 的 length()方法和数组的 length 属性

String 类有 length()方法吗？数组有 length()方法吗？

String 类当然有 length()方法了，看看 String 类的源码就知道了，这是这个方法的定义：

Java 代码

```
public int length() {
    return count;
}
```

String 的长度实际上就是它的属性--char 型数组 value 的长度。数组是没有 length()方法的，大家知道，在 JAVA 中，数组也被作为对象来处理，它的方法都继承自 Object 类。数组有一个属性 length，这也是它唯一的属性，对于所有类型的数组都是这样。

中文汉字在 char 中的保存一个中文汉字能保存在一个 char 类型里吗？

请看下面的例子：

Java 代码

```
public class ChineseTest {
    public static void main(String[] args) {
        // 将一个中文汉字赋值给一个 char 变量
        char a = '中';
        char b = '文';
        char c = '测';
        char d = '试';
        char e = '成';
        char f = '功';
        System.out.print(a);
        System.out.print(b);
        System.out.print(c);
        System.out.print(d);
        System.out.print(e);
```

```
                System.out.print(f);
        }
}
```

编译没有报错，运行结果：

1. 中文测试成功

答案就不用说了。为什么一个中文汉字可以保存在一个 char 变量里呢？因为在 JAVA 中，一个 char 是 2 个字节（byte），而一个中文汉字是一个字符，也是 2 个字节。而英文字母都是一个字节的，因此它也能保存到一个 byte 里，一个中文汉字却不能。请看：

Java 代码

```
public class ChineseTest {
        public static void main(String[] args) {
                // 将一个英文字母赋值给一个 byte 变量
                byte a = 'a';
                // 将一个中文汉字赋值给一个 byte 变量时，编译会报错
                // byte b = '中';

                System.out.println("byte a = " + a);
                // System.out.println("byte b = "+b);
        }
}
```

运行结果：

1. byte a = 97

正如大家所看到的那样，我们实际上是把字符'a'对应的 ASCII 码值赋值给了 byte 型变量 a。

让我们回过头来看看最初的例子，能不能将 a、b、c、d、e、f 拼接在一起一次输出呢？让我们试试看：

Java 代码

```
public class ChineseTest {
```

```
        public static void main(String[] args) {
                // 将一个中文汉字赋值给一个 char 变量
                char a = '中';
                char b = '文';
                char c = '测';
                char d = '试';
                char e = '成';
                char f = '功';
                System.out.print(a + b + c + d + e + f);
        }
}
```

运行结果：

- 156035

这显然不是我们想要的结果。只所以会这样是因为我们误用了“+”运算符，当它被用于字符串和字符串之间，或者字符串和其他类型变量之间时，它产生的效果是字符串的拼接；但当它被用于字符和字符之间时，效果等同于用于数字和数字之间，是一种算术运算。因此我们得到的“156035”是'中'、'文 '、'测'、'试'、'成'、'功'这六个汉字分别对应的数值算术相加后的结果。

字符串的反转输出

这也是面试题中常考的一道。我们就以一个包含了全部 26 个英文字母，同时又具有完整含义的最短句子作为例子来完成解答。先来看一下这个句子：

引用

```
A quick brown fox jumps over the lazy dog.（一只轻巧的棕色狐狸从那条懒狗身上跳了过去。）
```

最常用的方式就是反向取出每个位置的字符，然后依次将它们输出到控制台：

Java 代码

```
public class StringReverse {
        public static void main(String[] args) {
```

```
                // 原始字符串
                String s = "A quick brown fox jumps over the lazy dog.";
                System.out.println("原始的字符串：" + s);

                System.out.print("反转后字符串：");
                for (int i = s.length(); i > 0; i--) {
                        System.out.print(s.charAt(i - 1));
                }

                // 也可以转换成数组后再反转，不过有点多此一举
                char[] data = s.toCharArray();
                System.out.println();
                System.out.print("反转后字符串：");
                for (int i = data.length; i > 0; i--) {
                        System.out.print(data[i - 1]);
                }
        }
}
```

运行结果：

1. 原始的字符串：A quick brown fox jumps over the lazy dog.
2. 反转后字符串：.god yzal eht revo spmuj xof nworb kciuq A
3. 反转后字符串：.god yzal eht revo spmuj xof nworb kciuq A

以上两种方式虽然常用，但却不是最简单的方式，更简单的是使用现有的方法：

Java 代码

```
public class StringReverse {
        public static void main(String[] args) {
                // 原始字符串
                String s = "A quick brown fox jumps over the lazy dog.";
                System.out.println("原始的字符串：" + s);
```

```
                System.out.print("反转后字符串：");
                StringBuffer buff = new StringBuffer(s);
                // java.lang.StringBuffer 类的 reverse()方法可以将字符串反转
                System.out.println(buff.reverse().toString());
        }
}
```

运行结果：

1. 原始的字符串：A quick brown fox jumps over the lazy dog.
2. 反转后字符串：.god yzal eht revo spmuj xof nworb kciuq A

按字节截取含有中文汉字的字符串

要求实现一个按字节截取字符串的方法，比如对于字符串"我 ZWR 爱 JAVA"，截取它的前四位字节应该是"我 ZW"，而不是"我 ZWR"，同时要保证不会出现截取了半个汉字的情况。

英文字母和中文汉字在不同的编码格式下，所占用的字节数也是不同的，我们可以通过下面的例子来看看在一些常见的编码格式下，一个英文字母和一个中文汉字分别占用多少字节。

Java 代码

```
import java.io.UnsupportedEncodingException;

public class EncodeTest {
        /**
         * 打印字符串在指定编码下的字节数和编码名称到控制台
         *
         * @param s
         *          字符串
         * @param encodingName
         *          编码格式
         */
        public static void printByteLength(String s, String encodingName) {
                System.out.print("字节数：");
```

```
            try {
                System.out.print(s.getBytes(encodingName).length);
            } catch (UnsupportedEncodingException e) {
                e.printStackTrace();
            }
            System.out.println(";编码：" + encodingName);
    }

    public static void main(String[] args) {
            String en = "A";
            String ch = "人";

            // 计算一个英文字母在各种编码下的字节数
            System.out.println("英文字母：" + en);
            EncodeTest.printByteLength(en, "GB2312");
            EncodeTest.printByteLength(en, "GBK");
            EncodeTest.printByteLength(en, "GB18030");
            EncodeTest.printByteLength(en, "ISO-8859-1");
            EncodeTest.printByteLength(en, "UTF-8");
            EncodeTest.printByteLength(en, "UTF-16");
            EncodeTest.printByteLength(en, "UTF-16BE");
            EncodeTest.printByteLength(en, "UTF-16LE");

            System.out.println();

            // 计算一个中文汉字在各种编码下的字节数
            System.out.println("中文汉字：" + ch);
            EncodeTest.printByteLength(ch, "GB2312");
            EncodeTest.printByteLength(ch, "GBK");
            EncodeTest.printByteLength(ch, "GB18030");
            EncodeTest.printByteLength(ch, "ISO-8859-1");
            EncodeTest.printByteLength(ch, "UTF-8");
            EncodeTest.printByteLength(ch, "UTF-16");
            EncodeTest.printByteLength(ch, "UTF-16BE");
            EncodeTest.printByteLength(ch, "UTF-16LE");
```

```
        }
}
```

运行结果如下：

1. 英文字母：A
2. 字节数：1;编码：GB2312
3. 字节数：1;编码：GBK
4. 字节数：1;编码：GB18030
5. 字节数：1;编码：ISO-8859-1
6. 字节数：1;编码：UTF-8
7. 字节数：4;编码：UTF-16
8. 字节数：2;编码：UTF-16BE
9. 字节数：2;编码：UTF-16LE
10.
11. 中文汉字：人
12. 字节数：2;编码：GB2312
13. 字节数：2;编码：GBK
14. 字节数：2;编码：GB18030
15. 字节数：1;编码：ISO-8859-1
16. 字节数：3;编码：UTF-8
17. 字节数：4;编码：UTF-16
18. 字节数：2;编码：UTF-16BE
19. 字节数：2;编码：UTF-16LE

UTF-16BE 和 UTF-16LE 是 UNICODE 编码家族的两个成员。UNICODE 标准定义了
UTF-8、UTF-16、UTF-32 三种编码格式，共有
UTF-8、UTF-16、UTF-16BE、UTF-16LE、UTF-32、UTF-32BE、UTF-32LE 七种编码方案。
JAVA 所采用的编码方案是 UTF-16BE。从上例的运行结果中我们可以看出，
GB2312、GBK、GB18030 三种编码格式都可以满足题目的要求。下面我们就以 GBK 编码为例来进行解答。

如果我们直接按照字节截取会出现什么情况呢？我们来测试一下：

Java 代码

```
import java.io.UnsupportedEncodingException;

public class CutString {
    public static void main(String[] args) throws UnsupportedEncodingException {
        String s = "我 ZWR 爱 JAVA";
        // 获取 GBK 编码下的字节数据
        byte[] data = s.getBytes("GBK");
        byte[] tmp = new byte[6];
        // 将 data 数组的前六个字节拷贝到 tmp 数组中
        System.arraycopy(data, 0, tmp, 0, 6);
        // 将截取到的前六个字节以字符串形式输出到控制台
        s = new String(tmp);
        System.out.println(s);
    }
}
```

输出结果：

1. 我 ZWR?

在截取前六个字节时，第二个汉字“爱”被截取了一半，导致它无法正常显示了，这样显然是有问题的。

我们不能直接使用 String 类的 substring(int beginIndex, int endIndex)方法，因为它是按字符截取的。'我'和'Z'都被作为一个字符来看待，length 都是 1。实际上我们只要能区分开中文汉字和英文字母，这个问题就迎刃而解了，而它们的区别就是，中文汉字是两个字节，英文字母是一个字节。

Java 代码

```
import java.io.UnsupportedEncodingException;

public class CutString {

    /**
```

```
     * 判断是否是一个中文汉字
     *
     * @param c
     *          字符
     * @return true 表示是中文汉字，false 表示是英文字母
     * @throws UnsupportedEncodingException
     *          使用了 JAVA 不支持的编码格式
     */
    public static boolean isChineseChar(char c)
                    throws UnsupportedEncodingException {
            // 如果字节数大于 1，是汉字
            // 以这种方式区别英文字母和中文汉字并不是十分严谨，但在这个题目中，这样判断已经足够了
            return String.valueOf(c).getBytes("GBK").length > 1;
    }

    /**
     * 按字节截取字符串
     *
     * @param orignal
     *          原始字符串
     * @param count
     *          截取位数
     * @return 截取后的字符串
     * @throws UnsupportedEncodingException
     *          使用了 JAVA 不支持的编码格式
     */
    public static String substring(String orignal, int count)
                    throws UnsupportedEncodingException {
            // 原始字符不为 null，也不是空字符串
            if (orignal != null && !"".equals(orignal)) {
                    // 将原始字符串转换为 GBK 编码格式
                    orignal = new String(orignal.getBytes(), "GBK");
                    // 要截取的字节数大于 0，且小于原始字符串的字节数
                    if (count > 0 && count < orignal.getBytes("GBK").length) {
```

```
                        StringBuffer buff = new StringBuffer();
                        char c;
                        for (int i = 0; i < count; i++) {
                                // charAt(int index)也是按照字符来分解字符串的
                                c = orignal.charAt(i);
                                buff.append(c);
                                if (CutString.isChineseChar(c)) {
                                        // 遇到中文汉字，截取字节总数减 1
                                        --count;
                                }
                        }
                        return buff.toString();
                }
        }
        return orignal;
    }

    public static void main(String[] args) {
        // 原始字符串
        String s = "我 ZWR 爱 JAVA";
        System.out.println("原始字符串：" + s);
        try {
                System.out.println("截取前 1 位：" + CutString.substring(s, 1));
                System.out.println("截取前 2 位：" + CutString.substring(s, 2));
                System.out.println("截取前 4 位：" + CutString.substring(s, 4));
                System.out.println("截取前 6 位：" + CutString.substring(s, 6));
        } catch (UnsupportedEncodingException e) {
                e.printStackTrace();
        }
    }
}
```

运行结果：

1. 原始字符串：我 ZWR 爱 JAVA

2. 截取前 1 位：我

3. 截取前 2 位：我

4. 截取前 4 位：我 ZW

5. 截取前 6 位：我 ZWR 爱

(七）日期和时间的处理

关键字: java 面试题 日期 时间 转换

作者：臧圩人（zangweiren） 发布时间：2008-07-22

网址：http://zangweiren.javaeye.com

>>>转载请注明出处！<<<

日期和时间的处理不仅在面试题中会考到，在实际项目开发中也是我们经常需要处理的问题，似乎没有哪个项目可以避开它们，我们常常在处理用户的出生年月日、注册日期，订单的创建时间等属性时用到，由此可见其重要性。

java.util.Date 类

提到日期和时间，我想大家最先想到应该是 java.util.Date 类吧。Date 类可以精确到毫秒数，这个毫秒数是相对于格林威治标准时间 “1970-01-01 00:00:00.000 GMT”的差值。那么，什么是格林威治标准时间呢？要回答这个问题，我们需要先来了解一下世界时间标准方面的知识。

世界时间标准主要有 UTC，即 Coordinated Universal Time（中文名译作世界协调时间、世界统一时间或世界标准时间），以及 GMT，即 Greenwich Mean Time（中文名译作格林威治标准时间或格林威治平均时间）两种。严格来讲，UTC 比 GMT 更加精确一些，不过它们的差值不会超过 0.9 秒，如果超过了，将会为 UTC 增加闰秒以与 GMT，也就是地球自转周期保持一致。所以在日常使用中，我们可以把 UTC 和 GMT 一样看待。

日期和时间的表示是与我们所处的时区相关联的，如果我们不指定时区，那么它们将以系统默认的时区来显示。我们先来看看如何创建日期对象。Date 类有很多个构造器方法，大部分已经不被赞成使用了（Deprecated），不过还剩下两个可以使用的：

Java 代码

```
public Date() {
```

```
    this(System.currentTimeMillis());
}

public Date(long date) {
    //other code
}
```

第一个是无参构造器，使用系统当前时间的毫秒数来创建 Date 对象，它调用了 java.lang.System 类的 currentTimeMillis()来取得系统的当前时间的毫秒值。这是个本地方法，它的定义如下：

Java 代码

```
public static native long currentTimeMillis();
```

第二个构造器是根据给定的毫秒数来创建一个与之对应的 Date 对象，这个毫秒数决定了被创建对象的年、月、日、时、分、秒属性的值。

我们来看看日期和时间在默认时区下的显示效果：

Java 代码

```
import java.util.Date;

public class DateTest {
    public static void main(String[] args) {
        Date d = new Date();
        // 在默认时区下输出日期和时间值
        System.out.println(d);
    }
}
```

运行结果：

- Tue Jul 22 10:44:47 CST 2008

大家应该注意到了年份前的“CST”标识，它是 China Standard Time 的缩写，指的是中国标准时间，也就是我们常说的北京时间。它与 UTC 的时差是 UTC+8:00，就是说北京时间比世界标准时间早 8 个小时，如果世界标准时间是早上 1 点，北京时间就是早上 9 点。一般情况

下我们不需要关心时区问题。

在创建完 Date 对象之后，我们可以通过调用 getTime()方法来获得该对象的毫秒数值，调用 setTime(long time)方法来设置它的毫秒数值，从而影响年、月、日、时、分、秒这些属性。这两个方法的定义如下：

Java 代码

```
public long getTime() {
    //other code
}

public void setTime(long time) {
    //other code
}
```

既然 Date 对象可以表示盛相对于“1970-01-01 00:00:00.000 GMT”的毫秒数，我们自然可以通过这个值来比较两个日期的大小了，不过对于日期来讲，前后的说法应该更为恰当。而 Date 类已经为我们提供了这样的方法：

Java 代码

```
public boolean before(Date when) {
    //other code
}

public boolean after(Date when) {
    //other code
}

public int compareTo(Date anotherDate) {
    //other code
}
```

before()是判断当前日期是否在参数日期之前，即当前日期毫秒数小于参数日期毫秒数；after()是判断当前日期是否在参数日期之后，即当前日期毫秒数大于参数日期毫秒数。而 compareTo()是将当前日期与参数日期比较后，返回一个 int 型值，它的返回值有三种可能：-1、0 和 1。如果返回-1 则表示当前日期在参数日期之前；如果返回 0 则表示两个日期是同一时刻；返回 1 则表示当前日期在参数日期之后。虽然我们可以用 compareTo()方法来比较两个 Date 对象，但是它的设计实际是另有用途的，我们在后面的章节将会讲到。

下面我们就用一个示例来检验一下以上方法的用法：

Java 代码

```
import java.util.Date;

public class DateTest {
	public static void main(String[] args) {
		// 2008-08-08 20:00:00 对应的毫秒数
		long t2008 = 1218196800000L;
		// 1900-01-01 20:00:00 对应的毫秒数
		long t1900 = -2208945952000L;

		// 指定毫秒数创建 Date 对象
		Date d2008 = new Date(t2008);
		// 使用系统默认时间创建 Date 对象
		Date d1900 = new Date();
		// 通过设置毫秒数改变日期和时间
		d1900.setTime(t1900);

		System.out.println("调用方法：d1900.before(d2008)");
		System.out
				.print("比较结果：\"1900-01-01 20:00:00\"在\"2008-08-08
20:00:00\"");
		// 使用 before()方法比较
		if (d1900.before(d2008)) {
			System.out.println("之前");
		} else {
			System.out.println("之后");
		}

		System.out.println();

		System.out.println("调用方法：d2008.after(d1900)");
		System.out
				.print("比较结果：\"2008-08-08 20:00:00\"在\"1900-01-01
20:00:00\"");
```

```
                // 使用 after()方法比较
                if (d2008.after(d1900)) {
                        System.out.println("之后");
                } else {
                        System.out.println("之前");
                }

                System.out.println();

                System.out.println("调用方法：d1900.compareTo(d2008)");
                System.out
                                .print("比较结果：\"1900-01-01 20:00:00\"在\"2008-08-08
20:00:00\"");
                // 使用 compareTo()方法比较
                int i = d1900.compareTo(d2008);
                if (i == -1) {
                        System.out.println("之前");
                } else if (i == 1) {
                        System.out.println("之后");
                } else if (i == 0) {
                        System.out.println("是同一时刻");
                }
        }
}
```

运行结果：

1. 调用方法：d1900.before(d2008)
2. 比较结果："1900-01-01 20:00:00"在"2008-08-08 20:00:00"之前
3.
4. 调用方法：d2008.after(d1900)
5. 比较结果："2008-08-08 20:00:00"在"1900-01-01 20:00:00"之后
6.
7. 调用方法：d1900.compareTo(d2008)
8. 比较结果："1900-01-01 20:00:00"在"2008-08-08 20:00:00"之前

那么如果我们想直接获取或者改变年、月、日、时、分、秒等等这些属性的值时怎么办呢？Date 类当然有完成这些操作的方法，不过遗憾的是它们也都已经不被赞成使用了。我们必须换一个能够提供这些操作的类，这个类就是 java.util.Calendar。

公历历法 java.util.GregorianCalendar

Calendar 是一个抽象类，我们无法直接实例化它，它有一个具体子类实体类 java.util.GregorianCalendar，这个类实现的就是我们日常所用的公历历法，或者叫做阳历。我们可以直接使用 new 命令创建它的实例，或者使用 Calendar 类的这个方法来获得它实例：

Java 代码

```
public static Calendar getInstance(){
    //other code
}
```

采用上面这个方法时，我们创建的 Calendar 对象的日期和时间值是对象被创建时系统日期和时间值。当使用 new 命令时，我们有两种选择，一种是使用系统当前的日期和时间值初始化 GregorianCalendar 对象；另一种是通过给定年、月、日、时、分、秒等属性值来对其进行初始化。请看下面的例子：

Java 代码

```
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.GregorianCalendar;

public class DateTest {
        /**
         * 以一种较为友好的方式格式化日期时间值
         *
         * @param c
         *          日期时间对象
         * @return 格式化后的日期时间字符串
         */
        public static String toFriendlyString(Calendar c) {
                if (c != null) {
```

```
                DateFormat df = new SimpleDateFormat("yyyy 年 MM 月 dd 日
HH:mm:ss");
                return df.format(c.getTime());
            }
            return null;
        }

        public static void main(String[] args) {
            Calendar c1 = Calendar.getInstance();
            System.out.println("创建方式：Calendar.getInstance()");
            System.out.println("日期时间：" + DateTest.toFriendlyString(c1));
            System.out.println();

            Calendar c2 = new GregorianCalendar();
            System.out.println("创建方式：new GregorianCalendar()");
            System.out.println("日期时间：" + DateTest.toFriendlyString(c2));
            System.out.println();

            // 参数含义依次为：年、月、日
            Calendar c3 = new GregorianCalendar(2008, 8, 8);
            System.out.println("创建方式：new GregorianCalendar(2008, 8, 8)");
            System.out.println("日期时间：" + DateTest.toFriendlyString(c3));
            System.out.println();

            // 参数含义依次为：年、月、日、时、分
            Calendar c4 = new GregorianCalendar(2008, 8, 8, 6, 10);
            System.out.println("创建方式：new GregorianCalendar(2008, 8, 8, 6, 10)");
            System.out.println("日期时间：" + DateTest.toFriendlyString(c4));
            System.out.println();

            // 参数含义依次为：年、月、日、时、分、秒
            Calendar c5 = new GregorianCalendar(2008, 8, 8, 18, 10, 5);
            System.out.println("创建方式：new GregorianCalendar(2008, 8, 8, 18, 10,
5)");
            System.out.println("日期时间：" + DateTest.toFriendlyString(c5));
```

```
        }
}
```

运行结果如下：

1. 创建方式：Calendar.getInstance()
2. 日期时间：2008 年 07 月 22 日 11:54:48
3.
4. 创建方式：new GregorianCalendar()
5. 日期时间：2008 年 07 月 22 日 11:54:48
6.
7. 创建方式：new GregorianCalendar(2008, 8, 8)
8. 日期时间：2008 年 09 月 08 日 00:00:00
9.
10. 创建方式：new GregorianCalendar(2008, 8, 8, 6, 10)
11. 日期时间：2008 年 09 月 08 日 06:10:00
12.
13. 创建方式：new GregorianCalendar(2008, 8, 8, 18, 10, 5)
14. 日期时间：2008 年 09 月 08 日 18:10:05

为了便于阅读，我们增加一个 toFriendlyString(Calendar c)方法，它将日期时间值格式化为一种更加友好易懂的形式，我们将在接下来的内容中讲解它的实现原理。分析运行结果后，我们发现有两个地方需要

注意：

1. 在创建 GregorianCalendar 对象时，月份值都设定为 8，但打印结果都是 9 月份。这并不是我们的代码有问题，而是因为 JAVA 表示的月份是从 0 开始的，也就是说它用来表示月份的数值总是比实际月份值小 1。因此我们要表示 8 月份，就是应该设置 8-1=7 这个值。
2. GregorianCalendar 的小时数是 24 小时制的。

为了避免出现因为忘记处理 1 的差值而设置了错误的月份，也让代码看起来更加直观，推荐大家使用定义在 Calendar 类的的这些常量来代替直接用数字表示月份：

- 一月：Calendar.JANUARY = 0
- 二月：Calendar.FEBRUARY = 1
- 三月：Calendar.MARCH = 2
- 四月：Calendar.APRIL = 3

- 五月：Calendar.MAY = 4
- 六月：Calendar.JUNE = 5
- 七月：Calendar.JULY = 6
- 八月：Calendar.AUGUST = 7
- 九月：Calendar.SEPTEMBER = 8
- 十月：Calendar.OCTOBER = 9
- 十一月：Calendar.NOVEMBER = 10
- 十二月：Calendar.DECEMBER = 11

如果我们想要从 Calendar 对象获得各种属性的值，就需要调用它的 get(int field)方法，这个方法接收一个 int 型的参数，并且根据这个给定参数的值来返回相应的属性的值。该方法的定义如下：

Java 代码

```
public int get(int field){
    //other code
}
```

我们以一个示例来说明 get(int field)方法所能接受的一些常用参数的含义及用法：

Java 代码

```
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Calendar;

public class DateTest {
        /**
         * 以一种较为友好的方式格式化日期时间值
         *
         * @param c
         *          日期时间对象
         * @return 格式化后的日期时间字符串
         */
        public static String toFriendlyString(Calendar c) {
                if (c != null) {
                        DateFormat df = new SimpleDateFormat("yyyy 年 MM 月 dd 日
HH:mm:ss.SSS");
                        return df.format(c.getTime());
```

```
            }
            return null;
        }

        public static void main(String[] args) {
            Calendar c = Calendar.getInstance();
            System.out.println("当前时刻：" + DateTest.toFriendlyString(c));
            System.out.println();

            System.out.println("属性名称：Calendar.AM_PM");
            System.out.println("代表含义：上下午标识，上午返回 Calendar.AM=0，下午
返回
Calendar.PM=1");
            System.out.println("测试结果：" + c.get(Calendar.AM_PM));
            System.out.println();

            System.out.println("属性名称：Calendar.DATE");
            System.out.println("代表含义：一个月中的第几天，同
Calendar.DAY_OF_MONTH");
            System.out.println("测试结果：" + c.get(Calendar.DATE));
            System.out.println();

            System.out.println("属性名称：Calendar.DAY_OF_MONTH");
            System.out.println("代表含义：一个月中的第几天，同 Calendar.DATE");
            System.out.println("测试结果：" + c.get(Calendar.DAY_OF_MONTH));
            System.out.println();

            System.out.println("属性名称：Calendar.DAY_OF_WEEK");
            System.out.println("代表含义：一周中的第几天，对应星期几，第一天为星期
日，于
此类推。");
            System.out.println("星期日:Calendar.SUNDAY=1");
            System.out.println("星期一:Calendar.MONDAY=2");
            System.out.println("星期二:Calendar.TUESDAY=3");
            System.out.println("星期三:Calendar.WEDNESDAY=4");
```

```
                System.out.println("星期四:Calendar.THURSDAY=5");
                System.out.println("星期五:Calendar.FRIDAY=6");
                System.out.println("星期六:Calendar.SATURDAY=7");
                System.out.println("测试结果：" + c.get(Calendar.DAY_OF_WEEK));
                System.out.println();

                System.out.println("属性名称：Calendar.DAY_OF_WEEK_IN_MONTH");
                System.out.println("代表含义：这一天所对应的星期几在该月中是第几次出
现");
                System.out.println("测试结果：" +
c.get(Calendar.DAY_OF_WEEK_IN_MONTH));
                System.out.println();

                System.out.println("属性名称：Calendar.DAY_OF_YEAR");
                System.out.println("代表含义：一年中的第几天");
                System.out.println("测试结果：" + c.get(Calendar.DAY_OF_YEAR));
                System.out.println();

                System.out.println("属性名称：Calendar.HOUR");
                System.out.println("代表含义：12 小时制下的小时数，中午和午夜表示为 0");
                System.out.println("测试结果：" + c.get(Calendar.HOUR));
                System.out.println();

                System.out.println("属性名称：Calendar.HOUR_OF_DAY");
                System.out.println("代表含义：24 小时制下的小时数，午夜表示为 0");
                System.out.println("测试结果：" + c.get(Calendar.HOUR_OF_DAY));
                System.out.println();

                System.out.println("属性名称：Calendar.MILLISECOND");
                System.out.println("代表含义：毫秒数");
                System.out.println("测试结果：" + c.get(Calendar.MILLISECOND));
                System.out.println();

                System.out.println("属性名称：Calendar.MINUTE");
                System.out.println("代表含义：分钟");
```

```
                System.out.println("测试结果：" + c.get(Calendar.MINUTE));
                System.out.println();

                System.out.println("属性名称：Calendar.MONTH");
                System.out.println("代表含义：月份，从 0 到 11 表示 12 个月份，比实际月份
值小 1");
                System.out.println("测试结果：" + c.get(Calendar.MONTH));
                System.out.println();

                System.out.println("属性名称：Calendar.SECOND");
                System.out.println("代表含义：秒");
                System.out.println("测试结果：" + c.get(Calendar.SECOND));
                System.out.println();

                System.out.println("属性名称：Calendar.WEEK_OF_MONTH");
                System.out.println("代表含义：一个月中的第几个星期");
                System.out.println("测试结果：" + c.get(Calendar.WEEK_OF_MONTH));
                System.out.println();

                System.out.println("属性名称：Calendar.WEEK_OF_YEAR");
                System.out.println("代表含义：一年中的第几个星期");
                System.out.println("测试结果：" + c.get(Calendar.WEEK_OF_YEAR));
                System.out.println();

                System.out.println("属性名称：Calendar.YEAR");
                System.out.println("代表含义：年份");
                System.out.println("测试结果：" + c.get(Calendar.YEAR));
        }
}
```

运行结果如下：

1. 当前时刻：2008 年 07 月 22 日 13:16:07.421
2.
3. 属性名称：Calendar.AM_PM

4. 代表含义：上下午标识，上午返回 Calendar.AM=0，下午返回 Calendar.PM=1
5. 测试结果：1
6.
7. 属性名称：Calendar.DATE
8. 代表含义：一个月中的第几天，同 Calendar.DAY_OF_MONTH
9. 测试结果：22
10.
11. 属性名称：Calendar.DAY_OF_MONTH
12. 代表含义：一个月中的第几天，同 Calendar.DATE
13. 测试结果：22
14.
15. 属性名称：Calendar.DAY_OF_WEEK
16. 代表含义：一周中的第几天，对应星期几，第一天为星期日，于此类推。
17. 星期日:Calendar.SUNDAY=1
18. 星期一:Calendar.MONDAY=2
19. 星期二:Calendar.TUESDAY=3
20. 星期三:Calendar.WEDNESDAY=4
21. 星期四:Calendar.THURSDAY=5
22. 星期五:Calendar.FRIDAY=6
23. 星期六:Calendar.SATURDAY=7
24. 测试结果：3
25.
26. 属性名称：Calendar.DAY_OF_WEEK_IN_MONTH
27. 代表含义：这一天所对应的星期几在该月中是第几次出现
28. 测试结果：4
29.
30. 属性名称：Calendar.DAY_OF_YEAR
31. 代表含义：一年中的第几天
32. 测试结果：204
33.
34. 属性名称：Calendar.HOUR
35. 代表含义：12 小时制下的小时数，中午和午夜表示为 0
36. 测试结果：1

37.
38. 属性名称：Calendar.HOUR_OF_DAY
39. 代表含义：24 小时制下的小时数，午夜表示为 0
40. 测试结果：13
41.
42. 属性名称：Calendar.MILLISECOND
43. 代表含义：毫秒数
44. 测试结果：421
45.
46. 属性名称：Calendar.MINUTE
47. 代表含义：分钟
48. 测试结果：16
49.
50. 属性名称：Calendar.MONTH
51. 代表含义：月份，从 0 到 11 表示 12 个月份，比实际月份值小 1
52. 测试结果：6
53.
54. 属性名称：Calendar.SECOND
55. 代表含义：秒
56. 测试结果：7
57.
58. 属性名称：Calendar.WEEK_OF_MONTH
59. 代表含义：一个月中的第几个星期
60. 测试结果：4
61.
62. 属性名称：Calendar.WEEK_OF_YEAR
63. 代表含义：一年中的第几个星期
64. 测试结果：30
65.
66. 属性名称：Calendar.YEAR
67. 代表含义：年份
68. 测试结果：2008

其中 Calendar.DAY_OF_WEEK_IN_MONTH 代表的含义比较难理解一些，它表示“这一天所对应的星期几在该月中是第几次出现 ”。比如 2008 年 8 月 8 日是星期五，在它之前的 8 月 1 日也是星期五，因此它是 8 月份的第二个星期五。所以这时调用 get(Calendar.DAY_OF_WEEK_IN_MONTH)就会返回 2。这里存在一个简单易记的规律：对于每月的 1-7 号，它们一定占全了星期一到星期日，所以不管是它们中的哪一天，也不管这一天是星期几，它总是第一个，因此返回 1；8-14 号也同样占全了星期一到星期日，但由于 1-7 号的关系，对于它们总是返回 2；以此类推，15-21 号返回 3，22-28 号返回 4，29-31 号返回 5。

Calendar 对象和 Date 对象可以通过 Calendar 类的如下两个方法进行相互转换：

Java 代码

```
public final Date getTime() {
    //other code
}

public final void setTime(Date date) {
    //other code
}
```

日期格式化与解析

我们回头再来看看在上面的例子中定义的 toFriendlyString(Calendar c)方法，它将一个 Calendar 对象的日期时间值以一种很友好的方式来展现，使人们很容易看懂，也符合我们中国人的习惯。这完全得益于抽象类 DateFormat 以及它的子类实体类 SimpleDateFormat 的帮助。这两个类都位于 java.text 包中，是专门用于日期格式化和解析的类。而这两项工作的核心就是我们为此设定的 Pattern，我们可以称之为“日期格式表达式”。

理论上讲日期格式表达式包含全部 26 个英文字母的大小写，不过它们中的一些字母只是被预留了，并没有确切的含义。目前有效的字母及它们所代表的含义如下：

- G：年代标识，表示是公元前还是公元后
- y：年份
- M：月份
- d：日
- h：小时，从 1 到 12，分上下午
- H：小时，从 0 到 23
- m：分钟
- s：秒
- S：毫秒

- E：一周中的第几天，对应星期几，第一天为星期日，于此类推
- z：时区
- D：一年中的第几天
- F：这一天所对应的星期几在该月中是第几次出现
- w：一年中的第几个星期
- W：一个月中的第几个星期
- a：上午/下午标识
- k：小时，从 1 到 24
- K：小时，从 0 到 11，区分上下午

在日期格式表达式中出现的所有字母，在进行日期格式化操作后，都将被其所代表的含义对应的属性值所

替换，并且对某些字母来说，重复次数的不同，格式化后的结果也会有所不同。请看下面的例子：

Java 代码

```
import java.text.SimpleDateFormat;
import java.util.Date;

public class DateTest {
        public static void main(String[] args) {
                // 使用系统当前日期时间值创建一个 Date 对象
                Date now = new Date();

                // 创建一个日期格式表达式
                String pattern = "年代:G;年份:y;月份:M;日:d;时(1~12):h;时(0~23):H;分:m;秒:s;
毫秒:S;星期:E;上/下午:a;时区:z";
                // 使用日期格式表达式创建一个 SimpleDateFormat 对象
                SimpleDateFormat df = new SimpleDateFormat(pattern);
                // 调用 SimpleDateFormat 类的 format(Date date)方法对 Date 对象进行格式化，
并返回
格式化后的字符串。
                // 该方法继承自 java.text.DateFormat 类
                System.out.println("1 位： " + df.format(now));

                // 创建一个新的日期格式表达式
                pattern = "年代:GG;年份:yy;月份:MM;日:dd;时(1~12):hh;时(0~23):HH;分:mm;秒
```

```
:ss;毫秒:SS;星期:EE;上/下午:aa;时区:zz";
                // 调用 SimpleDateFormat 的 applyPattern(String pattern)方法用新创建的日期格
式
表达式替换其原有的
                df.applyPattern(pattern);
                System.out.println("2 位：" + df.format(now));

                pattern = "年代:GGG;年份:yyy;月份:MMM;日:ddd;时(1~12):hhh;时(0~23):HHH;
分
:mmm;秒:sss;毫秒:SSS;星期:EEE;上/下午:aaa;时区:zzz";
                df.applyPattern(pattern);
                System.out.println("3 位：" + df.format(now));

                pattern = "年代:GGGG;年份:yyyy;月份:MMMM;日:dddd;时(1~12):hhhh;时
(0~23):HHHH;分:mmmm;秒:ssss;毫秒:SSSS;星期:EEEE;上/下午:aaaa;时区:zzzz";
                df.applyPattern(pattern);
                System.out.println("4 位：" + df.format(now));

                pattern = "年代:GGGGG;年份:yyyyy;月份:MMMMM;日:ddddd;时(1~12):hhhhh;
时
(0~23):HHHHH;分:mmmmm;秒:sssss;毫秒:SSSSS;星期:EEEEE;上/下午:aaaaa;时区:zzzzz";
                df.applyPattern(pattern);
                System.out.println("5 位：" + df.format(now));

                pattern = "年代:GGGGGG;年份:yyyyyy;月份:MMMMMM;日:dddddd;时
(1~12):hhhhhh;时
(0~23):HHHHHH;分:mmmmmm;秒:ssssss;毫秒:SSSSSS;星期:EEEEEE;上/下午:aaaaaa;时
区:zzzzzz";
                df.applyPattern(pattern);
                System.out.println("6 位：" + df.format(now));
        }
}
```

输出结果如下：

1. 1 位：年代:公元;年份:08;月份:7;日:22;时(1~12):3;时(0~23):15;分:17;秒:49;毫秒:187;星期:

星期二;上/下午:下午;时区:CST

2. 2 位：年代:公元;年份:08;月份:07;日:22;时(1~12):03;时(0~23):15;分:17;秒:49;毫秒:187;星期

:星期二;上/下午:下午;时区:CST

3. 3 位：年代:公元;年份:08;月份:七月;日:022;时(1~12):003;时(0~23):015;分:017;秒:049;毫秒

:187;星期:星期二;上/下午:下午;时区:CST

4. 4 位：年代:公元;年份:2008;月份:七月;日:0022;时(1~12):0003;时(0~23):0015;分:0017;秒

:0049;毫秒:0187;星期:星期二;上/下午:下午;时区:中国标准时间

5. 5 位：年代:公元;年份:02008;月份:七月;日:00022;时(1~12):00003;时(0~23):00015;分:00017;秒

:00049;毫秒:00187;星期:星期二;上/下午:下午;时区:中国标准时间

6. 6 位：年代:公元;年份:002008;月份:七月;日:000022;时(1~12):000003;时(0~23):000015;分

:000017;秒:000049;毫秒:000187;星期:星期二;上/下午:下午;时区:中国标准时间

如果我们想输出原始的字母，而不是它们所代表含义的替换值，就需要用单引号将它们包含在内，对于预留字母也是如此，虽然它们没有确切的含义。一对单引号可以一次包含多个字母，而两个连续的单引号将输出一个单引号结果，双引号则需要转义后输出。对于 26 个字母之外的字符，可以放在一对单引号中，也可以直接书写。请看下面的例子：

Java 代码

```
import java.text.SimpleDateFormat;
import java.util.Date;

public class Test {
	public static void main(String[] args) {
		Date now = new Date();
		SimpleDateFormat df = new SimpleDateFormat(
				"'YEAR': yyyy 'MONTH:' "MM" 'DAY:' \"dd\" ");
		System.out.println(df.format(now));
	}
}
```

运行结果：

- YEAR: 2008 MONTH: '07' DAY: "22"

上面的一些例子中，我们将日期对象转换成一定格式的字符串输出，以得到符合我们习惯的较为友好的表现形式。我们还可以反过来，使用 DateFormat 类的 parse(String source)方法将具有一定格式的字符串转换为一个 Date 对象，前提是我们利用前面讲到日期格式表达式语法为其找到一个合适的 Pattern。例如：

Java 代码

```
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

public class DateTest {
	public static void main(String[] args) throws ParseException {
		String s = "2008-08-08";
		System.out.println("原始字符串：" + s);
		String pattern = "yyyy-MM-dd";
		System.out.println("对应表达式：" + pattern);
		SimpleDateFormat df = new SimpleDateFormat(pattern);
		Date date = df.parse(s);
		System.out.println("转换后的值：" + date);
		System.out.println();

		s = "05 年 2 月 12 日  18:04:33";
		System.out.println("原始字符串：" + s);
		pattern = "yy 年 M 月 d 日  HH:mm:ss";
		System.out.println("对应表达式：" + pattern);
		df.applyPattern(pattern);
		date = df.parse(s);
		System.out.println("转换后的值：" + date);
		System.out.println();

		s = "16/5/2004 20:7:2.050";
		System.out.println("原始字符串：" + s);
		pattern = "d/M/yyyy HH:m:s.SSS";
		System.out.println("对应表达式：" + pattern);
		df.applyPattern(pattern);
		date = df.parse(s);
```

```
                System.out.println("转换后的值：" + date);
        }
}
```

运行结果：

1. 原始字符串：2008-08-08
2. 对应表达式：yyyy-MM-dd
3. 转换后的值：Fri Aug 08 00:00:00 CST 2008
4.
5. 原始字符串：05 年 2 月 12 日 18:04:33
6. 对应表达式：yy 年 M 月 d 日 HH:mm:ss
7. 转换后的值：Sat Feb 12 18:04:33 CST 2005
8.
9. 原始字符串：16/5/2004 20:7:2.050
10. 对应表达式：d/M/yyyy HH:m:s.SSS
11. 转换后的值：Sun May 16 20:07:02 CST 2004

（八）聊聊基本类型（内置类型）

关键字: java 面试题 基本类型 int long boolean float double char

作者：臧圩人（zangweiren） 发布时间：2008-07-25

网址：http://zangweiren.javaeye.com

>>>转载请注明出处！<<<

基本类型，或者叫做内置类型，是 JAVA 中不同于类的特殊类型。它们是我们编程中使用最频繁的类型，因此面试题中也总少不了它们的身影，在这篇文章中我们将从面试中常考的几个方面来回顾一下与基本类型相关的知识。

基本类型共有九种，它们分别都有相对应的包装类。关于它们的详细信息请看下表：

序号	基本类型名称	包装类名称	二进制位数	取值范围
1.	byte	java.lang.Byte	8	-128 到 127
2.	short	java.lang.Short	16	-32768 到 32767
3.	int	java.lang.Integer	32	-2147483648 到 2147483647
4.	long	java.lang.Long	64	-92233720368477808 到 92233720368477807
5.	float	java.lang.Float	32	约-3.40282347E38 到 3.40282347E38 （有效小数位为 6-7）
6.	double	java.lang.Double	64	约-1.797693134862317E308 到 1.797693134862317E308 （有效小数位为 15）
7.	char	java.lang.Character	16	一个字符
8.	boolean	java.lang.Boolean	-	只能是 true 或者 false

对于基本类型 void 以及它的包装类 java.lang.Void，我们都无法直接进行操作。基本类型可以分为三类，字符类型 char，布尔类型 boolean 以及数值类型 byte、short、int、long、float、double。数值类型又可以分为整数类型 byte、short、int、long 和浮点数类型 float、double。JAVA 中的数值类型不存在无符号的，它们的取值范围是固定的，不会随着机器硬件环境或者操作系统的改变而改变。对于数值类型的基本类型的取值范围，我们无需强制去记忆，因为它们的值都已经以常量的形式定义在对应的包装类中了。请看下面的例子：

Java 代码

```
public class PrimitiveTypeTest {
        public static void main(String[] args) {
                // byte
                System.out.println("基本类型：byte 二进制位数：" + Byte.SIZE);
                System.out.println("包装类：java.lang.Byte");
                System.out.println("最小值：Byte.MIN_VALUE=" + Byte.MIN_VALUE);
                System.out.println("最大值：Byte.MAX_VALUE=" + Byte.MAX_VALUE);
                System.out.println();

                // short
                System.out.println("基本类型：short 二进制位数：" + Short.SIZE);
                System.out.println("包装类：java.lang.Short");
                System.out.println("最小值：Short.MIN_VALUE=" + Short.MIN_VALUE);
                System.out.println("最大值：Short.MAX_VALUE=" + Short.MAX_VALUE);
```

```
System.out.println();

// int
System.out.println("基本类型：int 二进制位数：" + Integer.SIZE);
System.out.println("包装类：java.lang.Integer");
System.out.println("最小值：Integer.MIN_VALUE=" + Integer.MIN_VALUE);
System.out.println("最大值：Integer.MAX_VALUE=" + Integer.MAX_VALUE);
System.out.println();

// long
System.out.println("基本类型：long 二进制位数：" + Long.SIZE);
System.out.println("包装类：java.lang.Long");
System.out.println("最小值：Long.MIN_VALUE=" + Long.MIN_VALUE);
System.out.println("最大值：Long.MAX_VALUE=" + Long.MAX_VALUE);
System.out.println();

// float
System.out.println("基本类型：float 二进制位数：" + Float.SIZE);
System.out.println("包装类：java.lang.Float");
System.out.println("最小值：Float.MIN_VALUE=" + Float.MIN_VALUE);
System.out.println("最大值：Float.MAX_VALUE=" + Float.MAX_VALUE);
System.out.println();

// double
System.out.println("基本类型：double 二进制位数：" + Double.SIZE);
System.out.println("包装类：java.lang.Double");
System.out.println("最小值：Double.MIN_VALUE=" + Double.MIN_VALUE);
System.out.println("最大值：Double.MAX_VALUE=" + Double.MAX_VALUE);
System.out.println();

// char
System.out.println("基本类型：char 二进制位数：" + Character.SIZE);
System.out.println("包装类：java.lang.Character");
// 以数值形式而不是字符形式将 Character.MIN_VALUE 输出到控制台
```

```
                System.out.println("最小值：Character.MIN_VALUE="
                                + (int) Character.MIN_VALUE);
                // 以数值形式而不是字符形式将 Character.MAX_VALUE 输出到控制台
                System.out.println("最大值：Character.MAX_VALUE="
                                + (int) Character.MAX_VALUE);
        }
}
```

运行结果：

```
1. 基本类型：byte 二进制位数：8
2. 包装类：java.lang.Byte
3. 最小值：Byte.MIN_VALUE=-128
4. 最大值：Byte.MAX_VALUE=127
5.
6. 基本类型：short 二进制位数：16
7. 包装类：java.lang.Short
8. 最小值：Short.MIN_VALUE=-32768
9. 最大值：Short.MAX_VALUE=32767
10.
11. 基本类型：int 二进制位数：32
12. 包装类：java.lang.Integer
13. 最小值：Integer.MIN_VALUE=-2147483648
14. 最大值：Integer.MAX_VALUE=2147483647
15.
16. 基本类型：long 二进制位数：64
17. 包装类：java.lang.Long
18. 最小值：Long.MIN_VALUE=-9223372036854775808
19. 最大值：Long.MAX_VALUE=9223372036854775807
20.
21. 基本类型：float 二进制位数：32
22. 包装类：java.lang.Float
23. 最小值：Float.MIN_VALUE=1.4E-45
24. 最大值：Float.MAX_VALUE=3.4028235E38
```

25.

26. 基本类型：double 二进制位数：64

27. 包装类：java.lang.Double

28. 最小值：Double.MIN_VALUE=4.9E-324

29. 最大值：Double.MAX_VALUE=1.7976931348623157E308

30.

31. 基本类型：char 二进制位数：16

32. 包装类：java.lang.Character

33. 最小值：Character.MIN_VALUE=0

34. 最大值：Character.MAX_VALUE=65535

Float 和 Double 的最小值和最大值都是以科学记数法的形式输出的，结尾的“E+数字”表示 E 之前的数字要乘以 10 的多少倍。比如 3.14E3 就是 3.14×1000=3140，3.14E-3 就是 3.14/1000=0.00314。

大家将运行结果与上表信息仔细比较就会发现 float、double 两种类型的最小值与 Float.MIN_VALUE、 Double.MIN_VALUE 的值并不相同，这是为什么呢？实际上 Float.MIN_VALUE 和 Double.MIN_VALUE 分别指的是 float 和 double 类型所能表示的最小正数。也就是说存在这样一种情况，0 到±Float.MIN_VALUE 之间的值 float 类型无法表示，0 到±Double.MIN_VALUE 之间的值 double 类型无法表示。这并没有什么好奇怪的，因为这些范围内的数值超出了它们的精度范围。

基本类型存储在栈中，因此它们的存取速度要快于存储在堆中的对应包装类的实例对象。从 Java5.0（1.5）开始，JAVA 虚拟机（Java Virtual Machine）可以完成基本类型和它们对应包装类之间的自动转换。因此我们在赋值、参数传递以及数学运算的时候像使用基本类型一样使用它们的包装类，但这并不意味着你可以通过基本类型调用它们的包装类才具有的方法。另外，所有基本类型（包括 void）的包装类都使用了 final 修饰，因此我们无法继承它们扩展新的类，也无法重写它们的任何方法。

各种数值类型之间的赋值与转换遵循什么规律呢？我们来看下面这个例子：

Java 代码

```
public class PrimitiveTypeTest {
        public static void main(String[] args) {
                // 给 byte 类型变量赋值时，数字后无需后缀标识
                byte byte_a = 1;
                // 编译器会做范围检查，如果赋予的值超出了范围就会报错
                // byte byte_b = 1000;
```

```
        // 把一个 long 型值赋值给 byte 型变量，编译时会报错，即使这个值没有超出 byte 类型的取值范围
        // byte byte_c = 1L;

        // 给 short 类型变量赋值时，数字后无需后缀标识
        short short_a = 1;
        // 编译器会做范围检查，如果赋予的值超出了范围就会报错
        // short short_b = 70000;
        // 把一个 long 型值赋值给 short 型变量，编译时会报错，即使这个值没有超出 short 类型的取值范围
        // byte short_c = 1L;

        // 给 short 类型变量赋值时，数字后无需后缀标识
        int int_a = 1;
        // 编译器会做范围检查，如果赋予的值超出了范围就会报错
        // int int_b = 2200000000;
        // 把一个 long 型值赋值给 int 型变量，编译时会报错，即使这个值没有超出 int 类型的取值范围
        // int int_c = 1L;

        // 可以把一个 int 型值直接赋值给 long 型变量，数字后无需后缀标识
        long long_a = 1;
        // 如果给 long 型变量赋予的值超出了 int 型值的范围，数字后必须加 L（不区分大小写）标识
        long long_b = 2200000000L;
        // 编译器会做范围检查，如果赋予的值超出了范围就会报错
        // long long_c = 9300000000000000000L;

        // 可以把一个 int 型值直接赋值给 float 型变量
        float float_a = 1;
        // 可以把一个 long 型值直接赋值给 float 型变量
        float float_b = 1L;
        // 没有 F（不区分大小写）后缀标识的浮点数默认为 double 型的，不能将它直接赋值给 float 型变量
        // float float_c = 1.0;
        // float 型数值需要有一个 F（不区分大小写）后缀标识
```

```
        float float_d = 1.0F;
        // 把一个 double 型值赋值给 float 型变量，编译时会报错，即使这个值没有超出 float 类型的取值范围
        // float float_e = 1.0D;
        // 编译器会做范围检查，如果赋予的值超出了范围就会报错
        // float float_f = 3.5000000E38F;

        // 可以把一个 int 型值直接赋值给 double 型变量
        double double_a = 1;
        // 可以把一个 long 型值直接赋值给 double 型变量
        double double_b = 1L;
        // 可以把一个 float 型值直接赋值给 double 型变量
        double double_c = 1F;
        // 不带后缀标识的浮点数默认为 double 类型的，可以直接赋值
        double double_d = 1.0;
        // 也可以给数字增加一个 D（不区分大小写）后缀标识，明确标出它是 double 类型的
        double double_e = 1.0D;
        // 编译器会做范围检查，如果赋予的值超出了范围就会报错
        // double double_f = 1.80000000000000000E308D;

        // 把一个 double 型值赋值给一个 byte 类型变量，编译时会报错，即使这个值没有超出 byte 类型的取值范围
        // byte byte_d = 1.0D;
        // 把一个 double 型值赋值给一个 short 类型变量，编译时会报错，即使这个值没有超出 short 类型的取值范围
        // short short_d = 1.0D;
        // 把一个 double 型值赋值给一个 int 类型变量，编译时会报错，即使这个值没有超出 int 类型的取值范围
        // int int_d = 1.0D;
        // 把一个 double 型值赋值给一个 long 类型变量，编译时会报错，即使这个值没有超出 long 类型的取值范围
        // long long_d = 1.0D;

        // 可以用字符初始化一个 char 型变量
        char char_a = 'a';
```

```
        // 也可以用一个 int 型数值初始化 char 型变量
        char char_b = 1;
        // 把一个 long 型值赋值给一个 char 类型变量，编译时会报错，即使这个值没有超出 char 类型的取值范围
        // char char_c = 1L;
        // 把一个 float 型值赋值给一个 char 类型变量，编译时会报错，即使这个值没有超出 char 类型的取值范围
        // char char_d = 1.0F;
        // 把一个 double 型值赋值给一个 char 类型变量，编译时会报错，即使这个值没有超出 char 类型的取值范围
        // char char_e = 1.0D;
        // 编译器会做范围检查，如果赋予的值超出了范围就会报错
        // char char_f = 70000;
    }
}
```

从上面的例子中我们可以得出如下几条结论：

1. 未带有字符后缀标识的整数默认为 int 类型；未带有字符后缀标识的浮点数默认为 double 类型。

2. 如果一个整数的值超出了 int 类型能够表示的范围，则必须增加后缀“L”（不区分大小写，建议用大写，因为小写的 L 与阿拉伯数字 1 很容易混淆），表示为 long 型。

3. 带有“F”（不区分大小写）后缀的整数和浮点数都是 float 类型的；带有“D”（不区分大小写）后缀的整数和浮点数都是 double 类型的。

4. 编译器会在编译期对 byte、short、int、long、float、double、char 型变量的值进行检查，如果超出了它们的取值范围就会报错。

5. int 型值可以赋给所有数值类型的变量；long 型值可以赋给 long、float、double 类型的变量；float 型值可以赋给 float、double 类型的变量；double 型值只能赋给 double 类型变量。

下图显示了几种基本类型之间的默认逻辑转换关系：

图中的实线表示无精度损失的转换，而虚线则表示这样的转换可能会损失一定的精度。如果我们想把一个能表示更大范围或者更高精度的类型，转换为一个范围更小或者精度更低的类型时，就需要使用强制类型转换（Cast）了。不过我们要尽量避免这种用法，因为它常常引发错误。请看下面的例子，如果不运行代码，你能预测它的结果吗？

Java 代码

```
public class PrimitiveTypeTest {
```

```
        public static void main(String[] args) {
                int a = 123456;
                short b = (short) a;
                // b 的值会是什么呢？
                System.out.println(b);
        }
}
```

运行结果：

1. -7616

运算符对基本类型的影响

当使用+、-、*、/、%运算符对基本类型进行运算时，遵循如下规则：

1. 只要两个操作数中有一个是 double 类型的，另一个将会被转换成 double 类型，并且结果也是 double 类型；
2. 否则，只要两个操作数中有一个是 float 类型的，另一个将会被转换成 float 类型，并且结果也是 float 类型；
3. 否则，只要两个操作数中有一个是 long 类型的，另一个将会被转换成 long 类型，并且结果也是 long 类型；
4. 否则，两个操作数（包括 byte、short、int、char）都将会被转换成 int 类型，并且结果也是 int 类型。

当使用+=、-=、*=、/=、%=、运算符对基本类型进行运算时，遵循如下规则：

- 运算符右边的数值将首先被强制转换成与运算符左边数值相同的类型，然后再执行运算，且运算结果与运算符左边数值类型相同。

了解了这些，我们就能解答下面这个常考的面试题了。请看：

引用

```
short s1=1;s1=s1+1;有什么错？ short s1=1;s1+=1;有什么错？
```

乍一看，觉得它们都应该没有错误，可以正常运行。我们来写个例子试试：

Java 代码

```
public class PrimitiveTypeTest {
        public static void main(String[] args) {
                short s1 = 1;
                // 这一行代码会报编译错误
                // s1 = s1 + 1;
                // 这一行代码没有报错
                s1 = 1 + 1;
                // 这一行代码也没有报错
                s1 += 1;
        }
}
```

从例子中我们可以看出结果了。利用上面列举的规律，也很容易解释。在 s1=s1+1;中，s1+1 运算的结果是 int 型，把它赋值给一个 short 型变量 s1，所以会报错；而在 s1+=1;中，由于是 s1 是 short 类型的，所以 1 首先被强制转换为 short 型，然后再参与运算，并且结果也是 short 类型的，因此不会报错。那么，s1=1+1;为什么不报错呢？这是因为 1+1 是个编译时可以确定的常量，“+”运算在编译时就被执行了，而不是在程序执行的时候，这个语句的效果等同于 s1=2，所以不会报错。前面讲过了，对基本类型执行强制类型转换可能得出错误的结果，因此在使用+=、 -=、*=、/=、%=等运算符时，要多加注意。

当使用“==”运算符在基本类型和其包装类对象之间比较时，遵循如下规则：

1. 只要两个操作数中有一个是基本类型，就是比较它们的数值是否相等。
2. 否则，就是判断这两个对象的内存地址是否相等，即是否是同一个对象。

下面的测试例子则验证了以上的规则：

Java 代码

```
public class EqualsTest {
        public static void main(String[] args) {
                // int 类型用 int 类型初始化
                int int_int = 0;
                // int 类型用 Integer 类型初始化
                int int_Integer = new Integer(0);
                // Integer 类型用 Integer 类型初始化
```

```
        Integer Integer_Integer = new Integer(0);
        // Integer 类型用 int 类型初始化
        Integer Integer_int = 0;

        System.out.println("int_int == int_Integer 结果是："
                + (int_int == int_Integer));
        System.out.println("Integer_Integer == Integer_int 结果是："
                + (Integer_Integer == Integer_int));
        System.out.println();
        System.out.println("int_int == Integer_Integer 结果是："
                + (int_int == Integer_Integer));
        System.out.println("Integer_Integer == int_int 结果是："
                + (Integer_Integer == int_int));
        System.out.println();

        // boolean 类型用 boolean 类型初始化
        boolean boolean_boolean = true;
        // boolean 类型用 Boolean 类型初始化
        boolean boolean_Boolean = new Boolean(true);
        // Boolean 类型用 Boolean 类型初始化
        Boolean Boolean_Boolean = new Boolean(true);
        // Boolean 类型用 boolean 类型初始化
        Boolean Boolean_boolean = true;

        System.out.println("boolean_boolean == boolean_Boolean 结果是："
                + (boolean_boolean == boolean_Boolean));
        System.out.println("Boolean_Boolean == Boolean_boolean 结果是："
                + (Boolean_Boolean == Boolean_boolean));
        System.out.println();
        System.out.println("boolean_boolean == Boolean_Boolean 结果是："
                + (boolean_boolean == Boolean_Boolean));
        System.out.println("Boolean_Boolean == boolean_boolean 结果是："
                + (Boolean_Boolean == boolean_boolean));
    }
}
```

运行结果：

1. int_int == int_Integer 结果是：true
2. Integer_Integer == Integer_int 结果是：false
3.
4. int_int == Integer_Integer 结果是：true
5. Integer_Integer == int_int 结果是：true
6.
7. boolean_boolean == boolean_Boolean 结果是：true
8. Boolean_Boolean == Boolean_boolean 结果是：false
9.
10. boolean_boolean == Boolean_Boolean 结果是：true
11. Boolean_Boolean == boolean_boolean 结果是：true

为了便于查看，上例中变量命名没有采用规范的方式，而是采用了“变量类型”+“_”+“初始化值类型”的方式。

Math.round()方法

java.lang.Math 类里有两个 round()方法，它们的定义如下：

Java 代码

```
public static int round(float a) {
      //other code
}

public static long round(double a) {
      //other code
}
```

它们的返回值都是整数，且都采用四舍五入法。运算规则如下：

1. 如果参数为正数，且小数点后第一位>=5，运算结果为参数的整数部分+1。

2. 如果参数为负数，且小数点后第一位>5，运算结果为参数的整数部分-1。

3. 如果参数为正数，且小数点后第一位<5；或者参数为负数，且小数点后第一位<=5，运算结果为参数的整数部分。

我们可以通过下面的例子来验证：

Java 代码

```
public class MathTest {
	public static void main(String[] args) {
		System.out.println("小数点后第一位=5");
		System.out.println("正数：Math.round(11.5)=" + Math.round(11.5));
		System.out.println("负数：Math.round(-11.5)=" + Math.round(-11.5));
		System.out.println();

		System.out.println("小数点后第一位<5");
		System.out.println("正数：Math.round(11.46)=" + Math.round(11.46));
		System.out.println("负数：Math.round(-11.46)=" + Math.round(-11.46));
		System.out.println();

		System.out.println("小数点后第一位>5");
		System.out.println("正数：Math.round(11.68)=" + Math.round(11.68));
		System.out.println("负数：Math.round(-11.68)=" + Math.round(-11.68));
	}
}
```

运行结果：

1. 小数点后第一位=5
2. 正数：Math.round(11.5)=12
3. 负数：Math.round(-11.5)=-11
4.
5. 小数点后第一位<5
6. 正数：Math.round(11.46)=11
7. 负数：Math.round(-11.46)=-11

8.
9. 小数点后第一位>5
10. 正数：Math.round(11.68)=12
11. 负数：Math.round(-11.68)=-12

根据上面例子的运行结果，我们还可以按照如下方式总结，或许更加容易记忆：

1. 参数的小数点后第一位<5，运算结果为参数整数部分。
2. 参数的小数点后第一位>5，运算结果为参数整数部分绝对值+1，符号（即正负）不变。
3. 参数的小数点后第一位=5，正数运算结果为整数部分+1，负数运算结果为整数部分。

但是上面的结论仍然不是很好记忆。我们来看看 round()方法的内部实现会给我们带来什么启发？我们来看这两个方法内部的代码：

Java 代码

```
public static int round(float a) {
        return (int)floor(a + 0.5f);
}

public static long round(double a) {
        return (long)floor(a + 0.5d);
}
```

看来它们都是将参数值+0.5 后交与 floor()进行运算，然后取返回值。那么 floor()方法的作用又是什么呢？它是取一个小于等于参数值的最大整数。比如经过 floor()方法运算后，如果参数是 10.2 则返回 10，13 返回 13，-20.82 返回-21，-16 返回-16 等等。既然是这样，我们就可以用一句话来概括 round()方法的运算效果了：

- Math 类的 round()方法的运算结果是一个<=(参数值+0.5)的最大整数。

switch 语句

哪些类型可以用于 switch 语句的判断呢？我们做个测试就知道了：

Java 代码

```
public class MathTest {
	// 枚举类型，Java5.0 以上版本可用
	static enum enum_e {
		A, B
	}

	public static void main(String[] args) {
		// byte
		byte byte_n = 0;
		switch (byte_n) {
		case 0:
			System.out.println("byte 可以用于 switch 语句");
			break;
		}

		// Byte 类
		Byte byte_m = 0;
		// 需要 Java5.0（1.5）以上版本支持
		switch (byte_m) {
		case 0:
			System.out.println("Byte 类可以用于 switch 语句");
			System.out.println();
			break;
		}

		// char
		char char_n = 0;
		switch (char_n) {
		case 0:
			System.out.println("char 可以用于 switch 语句");
			break;
		}

		// Character 类
```

```
Character char_m = 0;
// 需要 Java5.0（1.5）以上版本支持
switch (char_m) {
case 0:
        System.out.println("Character 类可以用于 switch 语句");
        System.out.println();
        break;
}

// short
short short_n = 0;
switch (short_n) {
case 0:
        System.out.println("short 可以用于 switch 语句");
        break;
}

// Short
Short short_m = 0;
// 需要 Java5.0（1.5）以上版本支持
switch (short_m) {
case 0:
        System.out.println("Short 类可以用于 switch 语句");
        System.out.println();
        break;
}

// int
int int_n = 0;
switch (int_n) {
case 0:
        System.out.println("int 可以用于 switch 语句");
        break;
}
```

```
// Integer 类
Integer int_m = 0;
// 需要 Java5.0（1.5）以上版本支持
switch (int_m) {
case 0:
        System.out.println("Integer 类可以用于 switch 语句");
        System.out.println();
        break;
}

// long
long long_n = 0;
// 编译错误，long 型不能用于 switch 语句
// switch (long_n) {
// case 0:
// System.out.println("long 可以用于 switch 语句");
// break;
// }

// Long 类
Long long_m = 0L;
// 编译错误，Long 类型不能用于 switch 语句
// switch (long_m) {
// case 0:
// System.out.println("Long 类可以用于 switch 语句");
// System.out.println();
// break;
// }

// float
float float_n = 0.0F;
// 编译错误，float 型不能用于 switch 语句
// switch (float_n) {
// case 0.0F:
// System.out.println("float 可以用于 switch 语句");
```

```
// break;
// }

// Float 类
Float float_m = 0.0F;
// 编译错误，Float 类型不能用于 switch 语句
// switch (float_m) {
// case 0.0F:
// System.out.println("Float 类可以用于 switch 语句");
// System.out.println();
// break;
// }

// double
double double_n = 0.0;
// 编译错误，double 型不能用于 switch 语句
// switch (double_n) {
// case 0.0:
// System.out.println("double 可以用于 switch 语句");
// break;
// }

// Double 类
Double double_m = 0.0;
// 编译错误，Double 类型不能用于 switch 语句
// switch (double_m) {
// case 0.0:
// System.out.println("Double 类可以用于 switch 语句");
// System.out.println();
// break;
// }

// boolean
boolean bool_b = true;
// 编译错误，boolean 型不能用于 switch 语句
```

```
// switch (bool_b) {
// case true:
// System.out.println("boolean 可以用于 switch 语句");
// break;
// }

// Boolean 类
Boolean bool_l = true;
// 编译错误，Boolean 类型不能用于 switch 语句
// switch (bool_l) {
// case true:
// System.out.println("Boolean 类可以用于 switch 语句");
// System.out.println();
// break;
// }

// String 对象
String string_s = "Z";
// 编译错误，long 型不能用于 switch 语句
// switch (string_s) {
// case "Z":
// System.out.println("String 可以用于 switch 语句");
// System.out.println();
// break;
// }

// enum（枚举类型，Java5.0 以上版本可用）
switch (MathTest.enum_e.A) {
case A:
        System.out.println("enum 可以用于 switch 语句-A");
        break;
case B:
        System.out.println("enum 可以用于 switch 语句-B");
        break;
}
```

```
        }
}
```

运行结果如下：

1. byte 可以用于 switch 语句
2. Byte 类可以用于 switch 语句
3.
4. char 可以用于 switch 语句
5. Character 类可以用于 switch 语句
6.
7. short 可以用于 switch 语句
8. Short 类可以用于 switch 语句
9.
10. int 可以用于 switch 语句
11. Integer 类可以用于 switch 语句
12.
13. enum 可以用于 switch 语句-A

结果已经出来了，我们来总结一下：

1. byte、char、short、int 四种基本类型以及它们的包装类（需要 Java5.0/1.5 以上版本支持）都可以用于 switch 语句。

2. long、float、double、boolean 四种基本类型以及它们的包装类（在 Java 所有版本中）都不能用于 switch 语句。

3. enum 类型，即枚举类型可以用于 switch 语句，但是要在 Java5.0（1.5）版本以上才支持。

4. 所有类型的对象（包括 String 类，但在 Java5.0/1.5 以上版本中，该项要排除 byte、char、short、int 四种基本类型对应的包装类）都不能用于 switch 语句。

（九）继承、多态、重载和重写

关键字: java 面试题 继承 多态 重载 重写

作者：臧圩人（zangweiren） 发布时间：2008-07-31

网址：http://zangweiren.javaeye.com

什么是多态？它的实现机制是什么呢？重载和重写的区别在那里？这就是这一次我们要回顾的四个十分重要的概念：继承、多态、重载和重写。

继承(inheritance)

简单的说，继承就是在一个现有类型的基础上，通过增加新的方法或者重定义已有方法（下面会讲到，这种方式叫重写）的方式，产生一个新的类型。继承是面向对象的三个基本特征--封装、继承、多态的其中之一，我们在使用 JAVA 时编写的每一个类都是在继承，因为在 JAVA 语言中，java.lang.Object 类是所有类最根本的基类（或者叫父类、超类），如果我们新定义的一个类没有明确地指定继承自哪个基类，那么 JAVA 就会默认为它是继承自 Object 类的。

我们可以把 JAVA 中的类分为以下三种：

1. 类：使用 class 定义且不含有抽象方法的类。
2. 抽象类：使用 abstract class 定义的类，它可以含有，也可以不含有抽象方法。
3. 接口：使用 interface 定义的类。

在这三种类型之间存在下面的继承规律：

- 类可以继承（extends）类，可以继承（extends）抽象类，可以继承（implements）接口。
- 抽象类可以继承（extends）类，可以继承（extends）抽象类，可以继承（implements）接口。
- 接口只能继承（extends）接口。

请注意上面三条规律中每种继承情况下使用的不同的关键字 extends 和 implements，它们是不可以随意替换的。大家知道，一个普通类继承一个接口后，必须实现这个接口中定义的所有方法，否则就只能被定义为抽象类。我在这里之所以没有对 implements 关键字使用“实现”这种说法是因为从概念上来说它也是表示一种继承关系，而且对于抽象类 implements 接口的情况下，它并不是一定要实现这个接口定义的任何方法，因此使用继承的说法更为合理一些。

以上三条规律同时遵守下面这些约束：

1. 类和抽象类都只能最多继承一个类，或者最多继承一个抽象类，并且这两种情况是互斥的，也就是说它们要么继承一个类，要么继承一个抽象类。

2. 类、抽象类和接口在继承接口时，不受数量的约束，理论上可以继承无限多个接口。当然，对于类来说，它必须实现它所继承的所有接口中定义的全部方法。

3. 抽象类继承抽象类，或者实现接口时，可以部分、全部或者完全不实现父类抽象类的抽象（abstract）方法，或者父类接口中定义的接口。

4. 类继承抽象类，或者实现接口时，必须全部实现父类抽象类的全部抽象（abstract）方法，或者父类接口中定义的全部接口。

继承给我们的编程带来的好处就是对原有类的复用（重用）。就像模块的复用一样，类的复用可以提高我们的开发效率，实际上，模块的复用是大量类的复用叠加后的效果。除了继承之外，我们还可以使用组合的方式来复用类。所谓组合就是把原有类定义为新类的一个属性，通过在新类中调用原有类的方法来实现复用。如果新定义的类型与原有类型之间不存在被包含的关系，也就是说，从抽象概念上来讲，新定义类型所代表的事物并不是原有类型所代表事物的一种，比如黄种人是人类的一种，它们之间存在包含与被包含的关系，那么这时组合就是实现复用更好的选择。下面这个例子就是组合方式的一个简单示例：

Java 代码

```
public class Sub {
	private Parent p = new Parent();

	public void doSomething() {
		// 复用 Parent 类的方法
		p.method();
		// other code
	}
}

class Parent {
	public void method() {
		// do something here
	}
}
```

当然，为了使代码更加有效，我们也可以在需要使用到原有类型（比如 Parent p）时，才对它进行初始化。

使用继承和组合复用原有的类，都是一种增量式的开发模式，这种方式带来的好处是不需要修改原有的代码，因此不会给原有代码带来新的 BUG，也不用因为对原有代码的修改而重新进行测试，这对我们的开发显然是有益的。因此，如果我们是在维护或者改造一个原有的系统或模块，尤其是对它们的了解不是很透彻的时候，就可以选择增量开发的模式，这不仅可以大大提高我们的开发效率，也可以规避由于对原有代码的修改而带来的风险。

多态(Polymorphism)

多态是又一个重要的基本概念，上面说到了，它是面向对象的三个基本特征之一。究竟什么是多态呢？我们先看看下面的例子，来帮助理解：

Java 代码

```
//汽车接口
interface Car {
	// 汽车名称
	String getName();

	// 获得汽车售价
	int getPrice();
}

// 宝马
class BMW implements Car {
	public String getName() {
		return "BMW";
	}

	public int getPrice() {
		return 300000;
	}
}

// 奇瑞 QQ
class CheryQQ implements Car {
	public String getName() {
```

```
            return "CheryQQ";
        }

        public int getPrice() {
            return 20000;
        }
}

// 汽车出售店
public class CarShop {
        // 售车收入
        private int money = 0;

        // 卖出一部车
        public void sellCar(Car car) {
            System.out.println("车型：" + car.getName() + "  单价：" + car.getPrice());
            // 增加卖出车售价的收入
            money += car.getPrice();
        }

        // 售车总收入
        public int getMoney() {
            return money;
        }

        public static void main(String[] args) {
            CarShop aShop = new CarShop();
            // 卖出一辆宝马
            aShop.sellCar(new BMW());
            // 卖出一辆奇瑞 QQ
            aShop.sellCar(new CheryQQ());
            System.out.println("总收入：" + aShop.getMoney());
        }
}
```

运行结果：

1. 车型：BMW 单价：300000
2. 车型：CheryQQ 单价：20000
3. 总收入：320000

继承是多态得以实现的基础。从字面上理解，多态就是一种类型（都是 Car 类型）表现出多种状态（宝马汽车的名称是 BMW，售价是 300000；奇瑞汽车的名称是 CheryQQ，售价是 2000）。将一个方法调用同这个方法所属的主体（也就是对象或类）关联起来叫做绑定，分前期绑定和后期绑定两种。下面解释一下它们的定义：

1. 前期绑定：在程序运行之前进行绑定，由编译器和连接程序实现，又叫做静态绑定。比如 static 方法和 final 方法，注意，这里也包括 private 方法，因为它是隐式 final 的。
2. 后期绑定：在运行时根据对象的类型进行绑定，由方法调用机制实现，因此又叫做动态绑定，或者运行时绑定。除了前期绑定外的所有方法都属于后期绑定。

多态就是在后期绑定这种机制上实现的。多态给我们带来的好处是消除了类之间的耦合关系，使程序更容易扩展。比如在上例中，新增加一种类型汽车的销售，只需要让新定义的类继承 Car 类并实现它的所有方法，而无需对原有代码做任何修改，CarShop 类的 sellCar(Car car)方法就可以处理新的车型了。新增代码如下：

Java 代码

```
// 桑塔纳汽车
class Santana implements Car {
	public String getName() {
		return "Santana";
	}

	public int getPrice() {
		return 80000;
	}
}
```

重载(overloading)和重写(overriding)

重载和重写都是针对方法的概念，在弄清楚这两个概念之前，我们先来了解一下什么叫方法

的型构（英文名是 signature，有的译作“签名”，虽然它被使用的较为广泛，但是这个翻译不准确的）。型构就是指方法的组成结构，具体包括方法的名称和参数，涵盖参数的数量、类型以及出现的顺序，但是不包括方法的返回值类型，访问权限修饰符，以及 abstract、static、final 等修饰符。比如下面两个就是具有相同型构的方法：

Java 代码

```
public void method(int i, String s) {
        // do something
}

public String method(int i, String s) {
        // do something
}
```

而这两个就是具有不同型构的方法：

Java 代码

```
public void method(int i, String s) {
        // do something
}

public void method(String s, int i) {
        // do something
}
```

了解完型构的概念后我们再来看看重载和重写，请看它们的定义：

- 重写，英文名是 overriding，是指在继承情况下，子类中定义了与其基类中方法具有相同型构的新方法，就叫做子类把基类的方法重写了。这是实现多态必须的步骤。
- 重载，英文名是 overloading，是指在同一个类中定义了一个以上具有相同名称，但是型构不同的方法。在同一个类中，是不允许定义多于一个的具有相同型构的方法的。

我们来考虑一个有趣的问题：构造器可以被重载吗？答案当然是可以的，我们在实际的编程中也经常这么做。实际上构造器也是一个方法，构造器名就是方法名，构造器参数就是方法参数，而它的返回值就是新创建的类的实例。但是构造器却不可以被子类重写，因为子类无法定义与基类具有相同型构的构造器。

（十）话说多线程

关键字: java 面试题 多线程 thread 线程池 synchronized 死锁

作者：臧圩人（zangweiren） 发布时间：2008-08-08

网址：http://zangweiren.javaeye.com

线程或者说多线程，是我们处理多任务的强大工具。线程和进程是不同的，每个进程都是一个独立运行的程序，拥有自己的变量，且不同进程间的变量不能共享；而线程是运行在进程内部的，每个正在运行的进程至少有一个线程，而且不同的线程之间可以在进程范围内共享数据。也就是说进程有自己独立的存储空间，而线程是和它所属的进程内的其他线程共享一个存储空间。线程的使用可以使我们能够并行地处理一些事情。线程通过并行的处理给用户带来更好的使用体验，比如你使用的邮件系统（outlook、Thunderbird、foxmail 等），你当然不希望它们在收取新邮件的时候，导致你连已经收下来的邮件都无法阅读，而只能等待收取邮件操作执行完毕。这正是线程的意义所在。

实现线程的方式

实现线程的方式有两种：

1. 继承 java.lang.Thread，并重写它的 run()方法，将线程的执行主体放入其中。
2. 实现 java.lang.Runnable 接口，实现它的 run()方法，并将线程的执行主体放入其中。

这是继承 Thread 类实现线程的示例：

Java 代码

```
public class ThreadTest extends Thread {
        public void run() {
                // 在这里编写线程执行的主体
                // do something
        }
}
```

这是实现 Runnable 接口实现多线程的示例：

Java 代码

```
public class RunnableTest implements Runnable {
        public void run() {
                // 在这里编写线程执行的主体
                // do something
        }
}
```

这两种实现方式的区别并不大。继承 Thread 类的方式实现起来较为简单，但是继承它的类就不能再继承别的类了，因此也就不能继承别的类的有用的方法了。而使用是想 Runnable 接口的方式就不存在这个问题了，而且这种实现方式将线程主体和线程对象本身分离开来，逻辑上也较为清晰，所以推荐大家更多地采用这种方式。

如何启动线程

我们通过以上两种方式实现了一个线程之后，线程的实例并没有被创建，因此它们也并没有被运行。我们要启动一个线程，必须调用方法来启动它，这个方法就是 Thread 类的 start()方法，而不是 run()方法（既不是我们继承 Thread 类重写的 run()方法，也不是实现 Runnable 接口的 run()方法）。run()方法中包含的是线程的主体，也就是这个线程被启动后将要运行的代码，它跟线程的启动没有任何关系。上面两种实现线程的方式在启动时会有所不同。

继承 Thread 类的启动方式：

Java 代码

```
public class ThreadStartTest {
        public static void main(String[] args) {
                // 创建一个线程实例
                ThreadTest tt = new ThreadTest();
                // 启动线程
                tt.start();
        }
}
```

实现 Runnable 接口的启动方式：

Java 代码

```
public class RunnableStartTest {
```

```
        public static void main(String[] args) {
                // 创建一个线程实例
                Thread t = new Thread(new RunnableTest());
                // 启动线程
                t.start();
        }
}
```

实际上这两种启动线程的方式原理是一样的。首先都是调用本地方法启动一个线程，其次是在这个线程里执行目标对象的 run()方法。那么这个目标对象是什么呢？为了弄明白这个问题，我们来看看 Thread 类的 run()方法的实现：

Java 代码

```
public void run() {
        if (target != null) {
                target.run();
        }
}
```

当我们采用实现 Runnable 接口的方式来实现线程的情况下，在调用 new Thread(Runnable target)构造器时，将实现 Runnable 接口的类的实例设置成了线程要执行的主体所属的目标对象 target，当线程启动时，这个实例的 run()方法就被执行了。当我们采用继承 Thread 的方式实现线程时，线程的这个 run()方法被重写了，所以当线程启动时，执行的是这个对象自身的 run()方法。总结起来就一句话，线程类有一个 Runnable 类型的 target 属性，它是线程启动后要执行的 run()方法所属的主体，如果我们采用的是继承 Thread 类的方式，那么这个 target 就是线程对象自身，如果我们采用的是实现 Runnable 接口的方式，那么这个 target 就是实现了 Runnable 接口的类的实例。

线程的状态

在 Java 1.4 及以下的版本中，每个线程都具有新建、可运行、阻塞、死亡四种状态，但是在 Java 5.0 及以上版本中，线程的状态被扩充为新建、可运行、阻塞、等待、定时等待、死亡六种。线程的状态完全包含了一个线程从新建到运行，最后到结束的整个生命周期。线程状态的具体信息如下：

1. NEW（新建状态、初始化状态）：线程对象已经被创建，但是还没有被启动时的状态。这段时间就是在我们调用 new 命令之后，调用 start()方法之前。

2. RUNNABLE（可运行状态、就绪状态）：在我们调用了线程的 start()方法之后线程所

处的状态。处于 RUNNABLE 状态的线程在 JAVA 虚拟机（JVM）上是运行着的，但是它可能还正在等待操作系统分配给它相应的运行资源以得以运行。

3. BLOCKED（阻塞状态、被中断运行）：线程正在等待其它的线程释放同步锁，以进入一个同步块或者同步方法继续运行；或者它已经进入了某个同步块或同步方法，在运行的过程中它调用了某个对象继承自 java.lang.Object 的 wait()方法，正在等待重新返回这个同步块或同步方法。

4. WAITING（等待状态）：当前线程调用了 java.lang.Object.wait()、java.lang.Thread.join()或者 java.util.concurrent.locks.LockSupport.park()三个中的任意一个方法，正在等待另外一个线程执行某个操作。比如一个线程调用了某个对象的 wait()方法，正在等待其它线程调用这个对象的 notify() 或者 notifyAll()（这两个方法同样是继承自 Object 类）方法来唤醒它；或者一个线程调用了另一个线程的 join()（这个方法属于 Thread 类）方法，正在等待这个方法运行结束。

5. TIMED_WAITING（定时等待状态）：当前线程调用了 java.lang.Object.wait(long timeout)、java.lang.Thread.join(long millis)、java.util.concurrent.locks.LockSupport.packNanos(long nanos)、java.util.concurrent.locks.LockSupport.packUntil(long deadline)四个方法中的任意一个，进入等待状态，但是与 WAITING 状态不同的是，它有一个最大等待时间，即使等待的条件仍然没有满足，只要到了这个时间它就会自动醒来。

6. TERMINATED（死亡状态、终止状态）：线程完成执行后的状态。线程执行完 run()方法中的全部代码，从该方法中退出，进入 TERMINATED 状态。还有一种情况是 run()在运行过程中抛出了一个异常，而这个异常没有被程序捕获，导致这个线程异常终止进入 TERMINATED 状态。

在 Java5.0 及以上版本中，线程的全部六种状态都以枚举类型的形式定义在 java.lang.Thread 类中了，代码如下：

Java 代码

```
public enum State {
        NEW,
        RUNNABLE,
        BLOCKED,
        WAITING,
        TIMED_WAITING,
        TERMINATED;
}
```

sleep()和 wait()的区别

sleep()方法和 wait()方法都成产生让当前运行的线程停止运行的效果，这是它们的共同点。下面我们来详细说说它们的不同之处。

sleep()方法是本地方法，属于 Thread 类，它有两种定义：

Java 代码

```
public static native void sleep(long millis) throws InterruptedException;

public static void sleep(long millis, int nanos) throws InterruptedException {
        //other code
}
```

其中的参数 millis 代表毫秒数（千分之一秒），nanos 代表纳秒数（十亿分之一秒）。这两个方法都可以让调用它的线程沉睡（停止运行）指定的时间，到了这个时间，线程就会自动醒来，变为可运行状态（RUNNABLE），但这并不表示它马上就会被运行，因为线程调度机制恢复线程的运行也需要时间。调用 sleep()方法并不会让线程释放它所持有的同步锁；而且在这期间它也不会阻碍其它线程的运行。上面的连个方法都声明抛出一个 InterruptedException 类型的异常，这是因为线程在 sleep()期间，有可能被持有它的引用的其它线程调用它的 interrupt()方法而中断。中断一个线程会导致一个 InterruptedException 异常的产生，如果你的程序不捕获这个异常，线程就会异常终止，进入 TERMINATED 状态，如果你的程序捕获了这个异常，那么程序就会继续执行 catch 语句块（可能还有 finally 语句块）以及以后的代码。

为了更好地理解 interrupt()效果，我们来看一下下面这个例子：

Java 代码

```
public class InterruptTest {
        public static void main(String[] args) {
                Thread t = new Thread() {
                        public void run() {
                                try {
                                        System.out.println("我被执行了-在 sleep()方法前");
                                        // 停止运行 10 分钟
                                        Thread.sleep(1000 * 60 * 60 * 10);
                                        System.out.println("我被执行了-在 sleep()方法后");
                                } catch (InterruptedException e) {
                                        System.out.println("我被执行了-在 catch 语句块中");
                                }
                                System.out.println("我被执行了-在 try{}语句块后");
                        }
                };
```

```
                // 启动线程
                t.start();
                // 在 sleep()结束前中断它
                t.interrupt();
        }
}
```

运行结果：

1. 我被执行了-在 sleep()方法前
2. 我被执行了-在 catch 语句块中
3. 我被执行了-在 try{}语句块后

wait()方法也是本地方法，属于 Object 类，有三个定义：

Java 代码

```
public final void wait() throws InterruptedException {
        //do something
}

public final native void wait(long timeout) throws InterruptedException;

public final void wait(long timeout, int nanos) throws InterruptedException {
        //do something
}
```

wari()和 wait(long timeout,int nanos)方法都是基于 wait(long timeout)方法实现的。同样地，timeout 代表毫秒数，nanos 代表纳秒数。当调用了某个对象的 wait()方法时，当前运行的线程就会转入等待状态（WAITING），等待别的线程再次调用这个对象的 notify()或者 notifyAll()方法（这两个方法也是本地方法）唤醒它，或者到了指定的最大等待时间，线程自动醒来。如果线程拥有某个或某些对象的同步锁，那么在调用了 wait()后，这个线程就会释放它持有的所有同步资源，而不限于这个被调用了 wait()方法的对象。wait()方法同样会被 Thread 类的 interrupt()方法中断，并产生一个 InterruptedException 异常，效果同 sleep()方法被中断一样。

实现同步的方式

同步是多线程中的重要概念。同步的使用可以保证在多线程运行的环境中，程序不会产生设计之外的错误结果。同步的实现方式有两种，同步方法和同步块，这两种方式都要用到 synchronized 关键字。

给一个方法增加 synchronized 修饰符之后就可以使它成为同步方法，这个方法可以是静态方法和非静态方法，但是不能是抽象类的抽象方法，也不能是接口中的接口方法。下面代码是一个同步方法的示例：

Java 代码

```
public synchronized void aMethod() {
        // do something
}

public static synchronized void anotherMethod() {
        // do something
}
```

线程在执行同步方法时是具有排它性的。当任意一个线程进入到一个对象的任意一个同步方法时，这个对象的所有同步方法都被锁定了，在此期间，其他任何线程都不能访问这个对象的任意一个同步方法，直到这个线程执行完它所调用的同步方法并从中退出，从而导致它释放了该对象的同步锁之后。在一个对象被某个线程锁定之后，其他线程是可以访问这个对象的所有非同步方法的。

同步块的形式虽然与同步方法不同，但是原理和效果是一致的。同步块是通过锁定一个指定的对象，来对同步块中包含的代码进行同步；而同步方法是对这个方法块里的代码进行同步，而这种情况下锁定的对象就是同步方法所属的主体对象自身。如果这个方法是静态同步方法呢？那么线程锁定的就不是这个类的对象了，也不是这个类自身，而是这个类对应的 java.lang.Class 类型的对象。同步方法和同步块之间的相互制约只限于同一个对象之间，所以静态同步方法只受它所属类的其它静态同步方法的制约，而跟这个类的实例（对象）没有关系。

下面这段代码演示了同步块的实现方式：

Java 代码

```
public void test() {
        // 同步锁
        String lock = "LOCK";
```

```
        // 同步块
        synchronized (lock) {
                // do something
        }

        int i = 0;
        // ...
}
```

对于作为同步锁的对象并没有什么特别要求，任意一个对象都可以。如果一个对象既有同步方法，又有同步块，那么当其中任意一个同步方法或者同步块被某个线程执行时，这个对象就被锁定了，其他线程无法在此时访问这个对象的同步方法，也不能执行同步块。

synchronized 和 Lock

Lock 是一个接口，它位于 Java 5.0 新增的 java.utils.concurrent 包的子包 locks 中。concurrent 包及其子包中的类都是用来处理多线程编程的。实现 Lock 接口的类具有与 synchronized 关键字同样的功能，但是它更加强大一些。 java.utils.concurrent.locks.ReentrantLock 是较常用的实现了 Lock 接口的类。下面是 ReentrantLock 类的一个应用实例：

Java 代码

```
private Lock lock = new ReentrantLock();

public void testLock() {
        // 锁定对象
        lock.lock();
        try {
                // do something
        } finally {
                // 释放对对象的锁定
                lock.unlock();
        }
}
```

lock()方法用于锁定对象，unlock()方法用于释放对对象的锁定，他们都是在 Lock 接口中定义的方法。位于这两个方法之间的代码在被执行时，效果等同于被放在 synchronized 同步块中。一般用法是将需要在 lock()和 unlock()方法之间执行的代码放在 try{}块中，并且在

finally{}块中调用 unlock()方法，这样就可以保证即使在执行代码抛出异常的情况下，对象的锁也总是会被释放，否则的话就会为死锁的产生增加可能。

使用 synchronized 关键字实现的同步，会把一个对象的所有同步方法和同步块看做一个整体，只要有一个被某个线程调用了，其他的就无法被别的线程执行，即使这些方法或同步块与被调用的代码之间没有任何逻辑关系，这显然降低了程序的运行效率。而使用 Lock 就能够很好地解决这个问题。我们可以把一个对象中按照逻辑关系把需要同步的方法或代码进行分组，为每个组创建一个 Lock 类型的对象，对实现同步。那么，当一个同步块被执行时，这个线程只会锁定与当前运行代码相关的其他代码最小集合，而并不影响其他线程对其余同步代码的调用执行。

关于死锁

死锁就是一个进程中的每个线程都在等待这个进程中的其他线程释放所占用的资源，从而导致所有线程都无法继续执行的情况。死锁是多线程编程中一个隐藏的陷阱，它经常发生在多个线程共用资源的时候。在实际开发中，死锁一般隐藏的较深，不容易被发现，一旦死锁现象发生，就必然会导致程序的瘫痪。因此必须避免它的发生。

程序中必须同时满足以下四个条件才会引发死锁：

1. 互斥（Mutual exclusion）：线程所使用的资源中至少有一个是不能共享的，它在同一时刻只能由一个线程使用。

2. 持有与等待（Hold and wait）：至少有一个线程已经持有了资源，并且正在等待获取其他的线程所持有的资源。

3. 非抢占式（No pre-emption）：如果一个线程已经持有了某个资源，那么在这个线程释放这个资源之前，别的线程不能把它抢夺过去使用。

4. 循环等待（Circular wait）：假设有 N 个线程在运行，第一个线程持有了一个资源，并且正在等待获取第二个线程持有的资源，而第二个线程正在等待获取第三个线程持有的资源，依此类推……第 N 个线程正在等待获取第一个线程持有的资源，由此形成一个循环等待。

线程池

线程池就像数据库连接池一样，是一个对象池。所有的对象池都有一个共同的目的，那就是为了提高对象的使用率，从而达到提高程序效率的目的。比如对于 Servlet，它被设计为多线程的（如果它是单线程的，你就可以想象，当 1000 个人同时请求一个网页时，在第一个人获得请求结果之前，其它 999 个人都在郁闷地等待），如果为每个用户的每一次请求都创建一个新的线程对象来运行的话，系统就会在创建线程和销毁线程上耗费很大的开销，大大降

低系统的效率。因此，Servlet 多线程机制背后有一个线程池在支持，线程池在初始化初期就创建了一定数量的线程对象，通过提高对这些对象的利用率，避免高频率地创建对象，从而达到提高程序的效率的目的。

下面实现一个最简单的线程池，从中理解它的实现原理。为此我们定义了四个类，它们的用途及具体实现如下：

1. Task（任务）：这是个代表任务的抽象类，其中定义了一个 deal()方法，继承 Task 抽象类的子类需要实现这个方法，并把这个任务需要完成的具体工作在 deal()方法编码实现。线程池中的线程之所以被创建，就是为了执行各种各样数量繁多的任务的，为了方便线程对任务的处理，我们需要用 Task 抽象类来保证任务的具体工作统一放在 deal()方法里来完成，这样也使代码更加规范。

Task 的定义如下：

Java 代码

```
public abstract class Task {
	public enum State {
		/* 新建  */NEW, /* 执行中  */RUNNING, /* 已完成  */FINISHED
	}

	// 任务状态
	private State state = State.NEW;

	public void setState(State state) {
		this.state = state;
	}

	public State getState() {
		return state;
	}

	public abstract void deal();
}
```

2. TaskQueue（任务队列）：在同一时刻，可能有很多任务需要执行，而程序在同一时刻

只能执行一定数量的任务，当需要执行的任务数超过了程序所能承受的任务数时怎么办呢？这就有了先执行哪些任务，后执行哪些任务的规则。TaskQueue 类就定义了这些规则中的一种，它采用的是 FIFO（先进先出，英文名是 First In First Out）的方式，也就是按照任务到达的先后顺序执行。

TaskQueue 类的定义如下：

Java 代码

```
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;


    public class TaskQueue {
        private List<Task> queue = new LinkedList<Task>();

        // 添加一项任务
        public synchronized void addTask(Task task) {
            if (task != null) {
                queue.add(task);
            }
        }

        // 完成任务后将它从任务队列中删除
        public synchronized void finishTask(Task task) {
            if (task != null) {
                task.setState(Task.State.FINISHED);
                queue.remove(task);
            }
        }

        // 取得一项待执行任务
        public synchronized Task getTask() {
            Iterator<Task> it = queue.iterator();
            Task task;
            while (it.hasNext()) {
                task = it.next();
```

```
                // 寻找一个新建的任务
                if (Task.State.NEW.equals(task.getState())) {
                    // 把任务状态置为运行中
                    task.setState(Task.State.RUNNING);
                    return task;
                }
            }
            return null;
        }
    }
```

3. addTask(Task task)方法用于当一个新的任务到达时，将它添加到任务队列中。这里使用了 LinkedList 类来保存任务到达的先后顺序。 finishTask(Task task)方法用于任务被执行完毕时，将它从任务队列中清除出去。getTask()方法用于取得当前要执行的任务。
TaskThread（执行任务的线程）：它继承自 Thread 类，专门用于执行任务队列中的待执行任务。

Java 代码

```
public class TaskThread extends Thread {
    // 该线程所属的线程池
    private ThreadPoolService service;

    public TaskThread(ThreadPoolService tps) {
        service = tps;
    }

    public void run() {
        // 在线程池运行的状态下执行任务队列中的任务
        while (service.isRunning()) {
            TaskQueue queue = service.getTaskQueue();
            Task task = queue.getTask();
            if (task != null) {
                task.deal();
```

```
                }
                queue.finishTask(task);
            }
        }
    }
```

4. ThreadPoolService（线程池服务类）：这是线程池最核心的一个类。它在被创建了时候就创建了几个线程对象，但是这些线程并没有启动运行，但调用了 start()方法启动线程池服务时，它们才真正运行。stop()方法可以停止线程池服务，同时停止池中所有线程的运行。而 runTask(Task task)方法是将一个新的待执行任务交与线程池来运行。

ThreadPoolService 类的定义如下：

Java 代码

```
import java.util.ArrayList;
import java.util.List;

public class ThreadPoolService {
    // 线程数
    public static final int THREAD_COUNT = 5;

    // 线程池状态
    private Status status = Status.NEW;

    private TaskQueue queue = new TaskQueue();

    public enum Status {
        /* 新建  */NEW, /* 提供服务中  */RUNNING, /* 停止服务
*/TERMINATED,
    }

    private List<Thread> threads = new ArrayList<Thread>();

    public ThreadPoolService() {
        for (int i = 0; i < THREAD_COUNT; i++) {
            Thread t = new TaskThread(this);
```

```
                threads.add(t);
            }
        }

        // 启动服务
        public void start() {
            this.status = Status.RUNNING;
            for (int i = 0; i < THREAD_COUNT; i++) {
                threads.get(i).start();
            }
        }

        // 停止服务
        public void stop() {
            this.status = Status.TERMINATED;
        }

        // 是否正在运行
        public boolean isRunning() {
            return status == Status.RUNNING;
        }

        // 执行任务
        public void runTask(Task task) {
            queue.addTask(task);
        }

        protected TaskQueue getTaskQueue() {
            return queue;
        }
    }
```

完成了上面四个类，我们就实现了一个简单的线程池。现在我们就可以使用它了，下面的代

码做了一个简单的示例：

Java 代码

```
public class SimpleTaskTest extends Task {
	@Override
	public void deal() {
		// do something
	}

	public static void main(String[] args) throws InterruptedException {
		ThreadPoolService service = new ThreadPoolService();
		service.start();
		// 执行十次任务
		for (int i = 0; i < 10; i++) {
			service.runTask(new SimpleTaskTest());
		}
		// 睡眠 1 秒钟，等待所有任务执行完毕
		Thread.sleep(1000);
		service.stop();
	}
}
```

当然，我们实现的是最简单的，这里只是为了演示线程池的实现原理。在实际应用中，根据情况的不同，可以做很多优化。比如：

- 调整任务队列的规则，给任务设置优先级，级别高的任务优先执行。
- 动态维护线程池，当待执行任务数量较多时，增加线程的数量，加快任务的执行速度；当任务较少时，回收一部分长期闲置的线程，减少对系统资源的消耗。

事实上 Java5.0 及以上版本已经为我们提供了线程池功能，无需再重新实现。这些类位于 java.util.concurrent 包中。

Executors 类提供了一组创建线程池对象的方法，常用的有一下几个：

Java 代码

```
public static ExecutorService newCachedThreadPool() {
```

```
        // other code
}

public static ExecutorService newFixedThreadPool(int nThreads) {
        // other code
}

public static ExecutorService newSingleThreadExecutor() {
        // other code
}
```

newCachedThreadPool()方法创建一个动态的线程池，其中线程的数量会根据实际需要来创建和回收，适合于执行大量短期任务的情况；newFixedThreadPool(int nThreads)方法创建一个包含固定数量线程对象的线程池，nThreads 代表要创建的线程数，如果某个线程在运行的过程中因为异常而终止了，那么一个新的线程会被创建和启动来代替它；而newSingleThreadExecutor()方法则只在线程池中创建一个线程，来执行所有的任务。

这三个方法都返回了一个 ExecutorService 类型的对象。实际上，ExecutorService 是一个接口，它的 submit()方法负责接收任务并交与线程池中的线程去运行。submit()方法能够接受 Callable 和 Runnable 两种类型的对象。它们的用法和区别如下：

1. Runnable 接口：继承 Runnable 接口的类要实现它的 run()方法，并将执行任务的代码放入其中，run()方法没有返回值。适合于只做某种操作，不关心运行结果的情况。

2. Callable 接口：继承 Callable 接口的类要实现它的 call()方法，并将执行任务的代码放入其中，call()将任务的执行结果作为返回值。适合于执行某种操作后，需要知道执行结果的情况。

无论是接收 Runnable 型参数，还是接收 Callable 型参数的 submit()方法，都会返回一个 Future（也是一个接口）类型的对象。该对象中包含了任务的执行情况以及结果。调用 Future 的 boolean isDone()方法可以获知任务是否执行完毕；调用 Object get()方法可以获得任务执行后的返回结果，如果此时任务还没有执行完，get()方法会保持等待，直到相应的任务执行完毕后，才会将结果返回。

我们用下面的一个例子来演示 Java5.0 中线程池的使用：

Java 代码

```
import java.util.concurrent.*;
```

```
public class ExecutorTest {
	public static void main(String[] args) throws InterruptedException,
			ExecutionException {
		ExecutorService es = Executors.newSingleThreadExecutor();
		Future fr = es.submit(new RunnableTest());// 提交任务

		Future fc = es.submit(new CallableTest());// 提交任务
		// 取得返回值并输出
		System.out.println((String) fc.get());

		// 检查任务是否执行完毕
		if (fr.isDone()) {
			System.out.println("执行完毕-RunnableTest.run()");
		} else {
			System.out.println("未执行完-RunnableTest.run()");
		}

		// 检查任务是否执行完毕
		if (fc.isDone()) {
			System.out.println("执行完毕-CallableTest.run()");
		} else {
			System.out.println("未执行完-CallableTest.run()");
		}

		// 停止线程池服务
		es.shutdown();
	}
}

class RunnableTest implements Runnable {
	public void run() {
		System.out.println("已经执行-RunnableTest.run()");
	}
}
```

```
class CallableTest implements Callable {
        public Object call() {
                System.out.println("已经执行-CallableTest.call()");
                return "返回值-CallableTest.call()";
        }
}
```

运行结果：

1. 已经执行-RunnableTest.run()
2. 已经执行-CallableTest.call()
3. 返回值-CallableTest.call()
4. 执行完毕-RunnableTest.run()
5. 执行完毕-CallableTest.run()

使用完线程池之后，需要调用它的 shutdown()方法停止服务，否则其中的所有线程都会保持运行，程序不会退出。

（十一）这些运算符你是否还记得？

关键字: java 面试题 自增 自减 位运算符

作者：臧圩人（zangweiren） 发布时间：2008-08-25

网址：http://zangweiren.javaeye.com

有些运算符在 JAVA 语言中存在着，但是在实际开发中我们或许很少用到它们，在面试题中却时常出现它们的身影，对于这些运算符的含义和用法，你是否还记得呢？

自增（++）和自减（--）运算符

我们先来回答几个问题吧：

Java 代码

```
int i = 0;
int j = i++;
int k = --i;
```

这段代码运行后，i 等于多少？j 等于多少？k 等于多少？太简单了？好，继续：

Java 代码

```
int i = 0;
int j = i++ + ++i;
int k = --i + i--;
```

代码执行后 i、j、k 分别等于多少呢？还是很简单？好，再继续：

Java 代码

```
int i=0;
System.out.println(i++);
```

这段代码运行后输出结果是什么？0？1？

Java 代码

```
float f=0.1F;
f++;
double d=0.1D;
d++;
char c='a';
c++;
```

上面这段代码可以编译通过吗？为什么？如果你能顺利回答到这里，说明你对自增和自减运算符的掌握已经很好了。

为了分析出上面提出的几个问题，我们首先来回顾一下相关知识：

- 自增（++）：将变量的值加 1，分前缀式（如++i）和后缀式（如 i++）。前缀式是先加 1 再使用；后缀式是先使用再加 1。
- 自减（--）：将变量的值减 1，分前缀式（如--i）和后缀式（如 i--）。前缀式是先减 1 再使用；后缀式是先使用再减 1。

在第一个例子中，int j=i++;是后缀式，因此 i 的值先被赋予 j，然后再自增 1，所以这行代码运行后，i=1、j=0；而 int k=--i;是前缀式，因此 i 先自减 1，然后再将它的值赋予 k，因此这行代码运行后，i=0、k=0。

在第二个例子中，对于 int j=i++ + ++i;，首先运行 i++，i 的值 0 被用于加运算（+），之后 i 自增值变为 1，然后运行++i，i 先自增变为 2，之后被用于加运算，最后将 i 两次的值相加的结果 0+2=2 赋给 j，因此这行代码运行完毕后 i=2、j=2；对于 int k=--i + i--;用一样的思路分析，具体过程在此不再赘述，结果应该是 i=0、k=2。

自增与自减运算符还遵循以下规律：

1. 可以用于整数类型 byte、short、int、long，浮点类型 float、double，以及字符串类型 char。

2. 在 Java5.0 及以上版本中，它们可以用于基本类型对应的包装器类 Byte、Short、Integer、Long、Float、Double、Character。

3. 它们的运算结果的类型与被运算的变量的类型相同。

下面的这个例子验证以上列出的规律，它可以编译通过并执行。

Java 代码

```
public class Test {
	public static void main(String[] args) {
		// 整型
		byte b = 0;
		b++;
		// 整型
		long l = 0;
		l++;
		// 浮点型
		double d = 0.0;
		d++;
		// 字符串
		char c = 'a';
		c++;
		// 基本类型包装器类
		Integer i = new Integer(0);
```

```
            i++;
        }
}
```

按位运算符

你还能说出来按位运算符一共有哪几种吗？对比下面的列表看看，有没有从你的记忆中消失了的：

1. 按位与运算（&）：二元运算符。当被运算的两个值都为 1 时，运算结果为 1；否则为 0。

2. 按位或运算（|）：二元运算符。当被运算的两个值都为 0 时，运算结果为 0；否则为 1。

3. 按位异或运算（^）：二元运算符。当被运算的两个值中任意一个为 1，另一个为 0 时，运算结果为 1；否则为 0。

4. 按位非运算（~）：一元运算符。当被运算的值为 1 时，运算结果为 0；当被运算的值为 0 时，运算结果为 1。

这里不像我们看到的逻辑运算符（与运算&&、或运算||、非运算！）操作的是布尔值 true 或 false，或者是一个能产生布尔值的表达式；“按位运算符”所指的“位”就是二进制位，因此它操作的是二进制的 0 和 1。在解释按位运算符的执行原理时，我们顺便说说它们和逻辑运算符的区别。

1. 逻辑运算符只能操作布尔值或者一个能产生布尔值的表达式；按位运算符能操作整型值，包括 byte、short、int、long，但是不能操作浮点型值（即 float 和 double），它还可以操作字符型（char）值。按位运算符不能够操作对象，但是在 Java5.0 及以上版本中，byte、short、int、long、char 所对应的包装器类是个例外，因为 JAVA 虚拟机会自动将它们转换为对应的基本类型的数据。

下面的例子验证了这条规律：

Java 代码

```
public class BitOperatorTest {
        public static void main(String[] args) {
                // 整型
                byte b1 = 10, b2 = 20;
```

```
                System.out.println("(byte)10 & (byte)20 = " + (b1 & b2));
                // 字符串型
                char c1 = 'a', c2 = 'A';
                System.out.println("(char)a | (char)A = " + (c1 | c2));
                // 基本类型的包装器类
                Long l1 = new Long(555), l2 = new Long(666);
                System.out.println("(Long)555 ^ (Long)666 = " + (l1 ^ l2));
                // 浮点型
                float f1 = 0.8F, f2 = 0.5F;
                // 编译报错，按位运算符不能用于浮点数类型
                // System.out.println("(float)0.8 & (float)0.5 = " + (f1 & f2));
        }
    }
```

运行结果：

- (byte)10 & (byte)20 = 0
- (char)a | (char)A = 97
- (Long)555 ^ (Long)666 = 177

2. 逻辑运算符的运算遵循短路形式，而按位运算符则不是。所谓短路就是一旦能够确定运算的结果，就不再进行余下的运算。

下面的例子更加直观地展现了短路与非短路的区别：

Java 代码

```
public class OperatorTest {
        public boolean leftCondition() {
                System.out.println("执行-返回值:false;方法:leftCondition()");
                return false;
        }

        public boolean rightCondition() {
                System.out.println("执行-返回值:true;方法:rightCondition()");
                return true;
```

```
        }

        public int leftNumber() {
            System.out.println("执行-返回值:0;方法:leftNumber()");
            return 0;
        }

        public int rightNumber() {
            System.out.println("执行-返回值:1;方法:rightNumber()");
            return 1;
        }

        public static void main(String[] args) {
            OperatorTest ot = new OperatorTest();

            if (ot.leftCondition() && ot.rightCondition()) {
                // do something
            }
            System.out.println();

            int i = ot.leftNumber() & ot.rightNumber();
        }
    }
```

运行结果：

- 执行-返回值:false;方法:leftCondition()
-
- 执行-返回值:0;方法:leftNumber()
- 执行-返回值:1;方法:rightNumber()

运行结果已经很明显地显示了短路和非短路的区别，我们一起来分析一下产生这个运

行结果的原因。当运行“ot.leftCondition() && ot.rightCondition()”时，由于方法leftCondition()返回了 false，而对于“&&”运算来说，必须要运算符两边的值都为 true 时，运算结果才为 true，因此这时候就可以确定，不论 rightCondition()的返回值是什么，“ot.leftCondition() && ot.rightCondition()”的运算值已经可以确定是 false，由于逻辑运算符是短路的形式，因此在这种情况下，rightCondition()方法就不再被运行了。

而对于“ot.leftNumber() & ot.rightNumber()”，由于“leftNumber()”的返回值是 0，对于按位运算符“&”来说，必须要运算符两边的值都是 1 时，运算结果才是 1，因此这时不管“rightNumber()”方法的返回值是多少，“ot.leftNumber() & ot.rightNumber()”的运算结果已经可以确定是 0，但是由于按位运算符是非短路的，所以 rightNumber()方法还是被执行了。这就是短路与非短路的区别。

移位运算符

移位运算符和按位运算符一样，同属于位运算符，因此移位运算符的位指的也是二进制位。它包括以下几种：

1. 左移位（<<）：将操作符左侧的操作数向左移动操作符右侧指定的位数。移动的规则是在二进制的低位补 0。

2. 有符号右移位（>>）：将操作符左侧的操作数向右移动操作符右侧指定的位数。移动的规则是，如果被操作数的符号为正，则在二进制的高位补 0；如果被操作数的符号为负，则在二进制的高位补 1。

3. 无符号右移位（>>>）：将操作符左侧的操作数向右移动操作符右侧指定的位数。移动的规则是，无论被操作数的符号是正是负，都在二进制位的高位补 0。

注意，移位运算符不存在“无符号左移位（<<<）”一说。与按位运算符一样，移位运算符可以用于 byte、short、 int、long 等整数类型，和字符串类型 char，但是不能用于浮点数类型 float、double；当然，在 Java5.0 及以上版本中，移位运算符还可用于byte、short、int、long、char 对应的包装器类。我们可以参照按位运算符的示例写一个测试程序来验证，这里就不再举例了。

与按位运算符不同的是，移位运算符不存在短路不短路的问题。

写到这里就不得不提及一个在面试题中经常被考到的题目：

引用

请用最有效率的方法计算出 2 乘以 8 等于几？

这里所谓的最有效率，实际上就是通过最少、最简单的运算得出想要的结果，而移位是计算机中相当基础的运算了，用它来实现准没错了。左移位 “<<”把被操作数每向左移动一位，效果等同于将被操作数乘以 2，而 2*8=（2*2*2*2），就是把 2 向左移位 3 次。因此最有效率的计算 2 乘以 8 的方法就是“2<<3”。

最后，我们再来考虑一种情况，当要移位的位数大于被操作数对应数据类型所能表示的最大位数时，结果会是怎样呢？比如，1<<35=?呢？

这里就涉及到移位运算的另外一些规则：

1. byte、short、char 在做移位运算之前，会被自动转换为 int 类型，然后再进行运算。

2. byte、short、int、char 类型的数据经过移位运算后结果都为 int 型。

3. long 经过移位运算后结果为 long 型。

4. 在左移位（<<）运算时，如果要移位的位数大于被操作数对应数据类型所能表示的最大位数，那么先将要求移位数对该类型所能表示的最大位数求余后，再将被操作数移位所得余数对应的数值，效果不变。比如 1<<35=1<<(35%32)=1<& lt;3=8。

5. 对于有符号右移位（>>）运算和无符号右移位（>>>）运算，当要移位的位数大于被操作数对应数据类型所能表示的最大位数时，那么先将要求移位数对该类型所能表示的最大位数求余后，再将被操作数移位所得余数对应的数值，效果不变。。比如 100>>35=100>>(35%32)=100>>3=12。

下面的测试代码验证了以上的规律：

Java 代码

```
public abstract class Test {
        public static void main(String[] args) {
                System.out.println("1 << 3 = " + (1 << 3));
                System.out.println("(byte) 1 << 35 = " + ((byte) 1 << (32 + 3)));
                System.out.println("(short) 1 << 35 = " + ((short) 1 << (32 + 3)));
                System.out.println("(char) 1 << 35 = " + ((char) 1 << (32 + 3)));
                System.out.println("1 << 35 = " + (1 << (32 + 3)));
                System.out.println("1L << 67 = " + (1L << (64 + 3)));
                // 此处需要 Java5.0 及以上版本支持
                System.out.println("new Integer(1) << 3 = " + (new Integer(1) << 3));
                System.out.println("10000 >> 3 = " + (10000 >> 3));
                System.out.println("10000 >> 35 = " + (10000 >> (32 + 3)));
                System.out.println("10000L >>> 67 = " + (10000L >>> (64 + 3)));
```

```
        }
}
```

运行结果：

1. 1 << 3 = 8
2. (byte) 1 << 35 = 8
3. (short) 1 << 35 = 8
4. (char) 1 << 35 = 8
5. 1 << 35 = 8
6. 1L << 67 = 8
7. new Integer(1) << 3 = 8
8. 10000 >> 3 = 1250
9. 10000 >> 35 = 1250
10. 10000L >>> 67 = 1250