开始，关系数据库理论逐渐成熟，随之涌现出大量商用的关系数据库。关系数据库理论经过30多年的发展已经相当完善，在大量数据的查找、排序操作上非常成熟且快速，并对数据库系统的并发、隔离有非常完善的解决方案。

面向对象数据库则是由面向对象编程语言催生的新型数据库，目前有些数据库系统如Oracle 11g等开始增加面向对象特性，但面向对象数据库还没有大规模地商业应用。

对于关系数据库而言，最基本的数据存储单元就是数据表，因此可以简单地把数据库想象成大量数据表的集合（当然，数据库绝不仅由数据表组成）。

数据表是存储数据的逻辑单元，可以把数据表想象成由行和列组成的表格，其中每一行也被称为一条记录，每一列也被称为一个字段。为数据库建表时，通常需要指定该表包含多少列，每列的数据类型信息，无须指定该数据表包含多少行——因为数据库表的行是动态改变的，每行用于保存一条用户数据。除此之外，还应该为每个数据表指定一个特殊列，该特殊列的值可以唯一地标识此行的记录，则该特殊列被称为主键列。

MySQL数据库的一个实例（Server Instance）可以同时包含多个数据库，MySQL使用如下命令来查看当前实例下包含多少个数据库：

```
show databases;
```

注意：

MySQL默认以分号作为每条命令的结束符，所以在每条MySQL命令结束后都应该输一个英文分号（;）。

如果用户需要创建新的数据库，则可以使用如下命令：

```
create database [IF NOT EXISTS] 数据库名;
```

如果用户需要删除指定数据库，则可以使用如下命令：

```
drop database 数据库名;
```

建立了数据库之后，如果想操作该数据库（例如为该数据库建表，在该数据库中执行查询等操作），则需要进入该数据库。进入指定数据库可以使用如下命令：

```
use 数据库名;
```

进入指定数据库后，如果需要查询该数据库下包含多少个数据表，则可以使用如下命令：

```
show tables;
```

如果想查看指定数据表的表结构（查看该表有多少列，每列的数据类型等信息），则可以使用如下命令：

```
desc 表名
```

图13.8显示了使用MySQL命令行客户端执行这些命令的效果。

正如在图13.8中看到的，MySQL的命令行客户端依次执行了show databases;、drop database abc;等命令，如果将多条MySQL命令写在一份SQL脚本文件里，然后将这份SQL脚本的内容一次复制到该窗口里，将可以看到该命令行客户端一次性执行所有SQL命令的效果——这种一次性执行多条SQL命令的方式也被称为导入SQL脚本。

提示：

本章的大量程序需要相应数据库的支持，因为本章的大部分程序都会提供对应的SQL脚本，因此运行这些程序之前，应该先向MySQL数据库中导入这些SQL脚本。

MySQL数据库安装成功后，在其安装目录下有一个bin路径（本书中该路径为D:\Program Files\MySQL\MySQL Server 5.7\bin），该路径下包含一个mysql命令，该命令用于启动MySQL命令行客户端。执行mysql命令的语法如下：

```
mysql -p 密码 -u 用户名 -h 主机名 --default-character-set=utf8
```

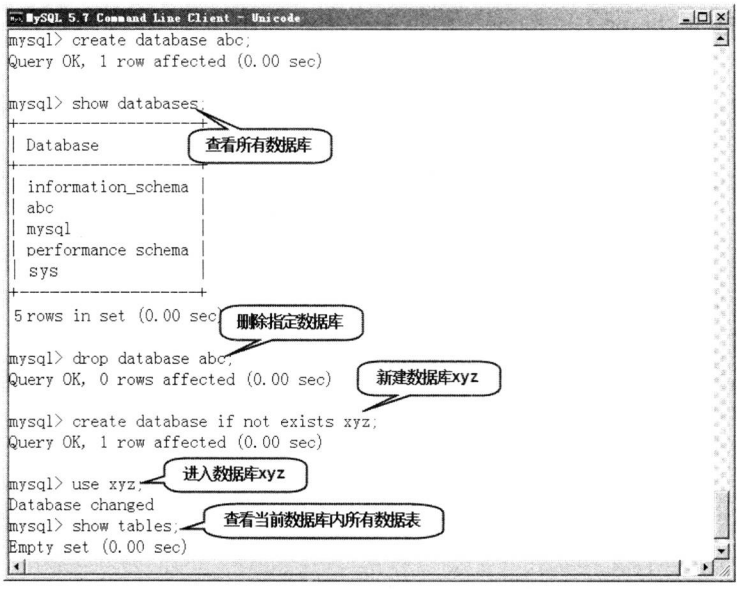


图 13.8　执行 MySQL 常用命令

执行上面命令可以连接远程主机的 MySQL 服务。为了保证有较好的安全性，执行上面命令时可以省略-p 后面的密码，执行该命令后系统会提示输入密码。

提示：

为了更方便地使用该命令，可以将该 MySQL 安装路径下的 bin 目录添加到系统 PATH 环境变量中。实际上，“开始”菜单中的“MySQL 5.7 Command Line Client - Unicode”菜单项就是一条 mysql 命令。

MySQL 数据库通常支持如下两种存储机制。

- MyISAM：这是 MySQL 早期默认的存储机制，对事务支持不够好。
- InnoDB：InnoDB 提供事务安全的存储机制。InnoDB 通过建立行级锁来保证事务完整性，并以 Oracle 风格的共享锁来处理 Select 语句。系统默认启动 InnoDB 存储机制，如果不想使用 InnoDB 表，则可以使用 skip-innodb 选项。

对比两种存储机制，不难发现 InnoDB 比 MyISAM 多了事务支持的功能，而事务支持是 Java EE 最重要的特性，因此通常推荐使用 InnoDB 存储机制。如果使用 5.0 以上版本的 MySQL 数据库系统，通常无须指定数据表的存储机制，因为系统默认使用 InnoDB 存储机制。如果需要在建表时显式指定存储机制，则可在标准建表语法的后面添加下面任意一句。

- ENGINE=MyISAM——强制使用 MyISAM 存储机制。
- ENGINE=InnoDB——强制使用 InnoDB 存储机制。

13.2.3　SQL 语句基础

SQL 的全称是 Structured Query Language，也就是结构化查询语言。SQL 是操作和检索关系数据库的标准语言，标准的 SQL 语句可用于操作任何关系数据库。

使用 SQL 语句，程序员和数据库管理员（DBA）可以完成如下任务。

- 在数据库中检索信息。
- 对数据库的信息进行更新。
- 改变数据库的结构。
- 更改系统的安全设置。
- 增加或回收用户对数据库、表的许可权限。

在上面 5 个任务中，一般程序员可以管理前 3 个任务，后面 2 个任务通常由 DBA 来完成。

标准的 SQL 语句通常可分为如下几种类型。

- 查询语句：主要由 select 关键字完成，查询语句是 SQL 语句中最复杂、功能最丰富的语句。

- DML（Data Manipulation Language，数据操作语言）语句：主要由 insert、update 和 delete 三个关键字完成。
- DDL（Data Definition Language，数据定义语言）语句：主要由 create、alter、drop 和 truncate 四个关键字完成。
- DCL（Data Control Language，数据控制语言）语句：主要由 grant 和 revoke 两个关键字完成。
- 事务控制语句：主要由 commit、rollback 和 savepoint 三个关键字完成。

SQL 语句的关键字不区分大小写，也就是说，create 和 CREATE 的作用完全一样。在上面 5 种 SQL 语句中，DCL 语句用于为数据库用户授权，或者回收指定用户的权限，通常无须程序员操作，所以本节不打算介绍任何关于 DCL 的知识。

在 SQL 命令中也可能需要使用标识符，标识符可用于定义表名、列名，也可用于定义变量等。这些标识符的命名规则如下。

- 标识符通常必须以字母开头。
- 标识符包括字母、数字和三个特殊字符（# _ $）。
- 不要使用当前数据库系统的关键字、保留字，通常建议使用多个单词连缀而成，单词之间以_分隔。
- 同一个模式下的对象不应该同名，这里的模式指的是外模式。

掌握了 SQL 的这些基础知识后，下面将分类介绍各种 SQL 语句。

注意：

truncate 是一个特殊的 DDL 语句，truncate 在很多数据库中都被归类为 DDL，它相当于先删除指定的数据表，然后再重建该数据表。如果使用 MySQL 的普通存储机制，truncate 确实是这样的。但如果使用 InnoDB 存储机制，则比较复杂，在 MySQL 5.0.3 之前，truncate 和 delete 完全一样；在 5.0.3 之后，truncate table 比 delete 效率高，但如果该表被外键约束所参照，则依然被映射成 delete 操作。当使用快速 truncate 时，该操作会重设自动增长计数器。在 5.0.13 之后，快速 truncate 总是可用，即比 delete 性能要好。关于 truncate 的用法，请参考本章后面内容。

13.2.4 DDL 语句

DDL 语句是操作数据库对象的语句，包括创建（create）、删除（drop）和修改（alter）数据库对象。

前面已经介绍过，最基本的数据库对象是数据表，数据表是存储数据的逻辑单元。但数据库里绝不仅包括数据表，数据库里可包含如表 13.1 所示的几种常见的数据库对象。

表 13.1 常见的数据库对象

对象名称	对应关键字	描　述
表	table	表是存储数据的逻辑单元，以行和列的形式存在；列就是字段，行就是记录
数据字典		就是系统表，存放数据库相关信息的表。系统表里的数据通常由数据库系统维护，程序员通常不应该手动修改系统表及系统表数据，只可查看系统表数据
约束	constraint	执行数据校验的规则，用于保证数据完整性的规则
视图	view	一个或者多个数据表里数据的逻辑显示。视图并不存储数据
索引	index	用于提高查询性能，相当于书的目录
函数	function	用于完成一次特定的计算，具有一个返回值
存储过程	procedure	用于完成一次完整的业务处理，没有返回值，但可通过传出参数将多个值传给调用环境
触发器	trigger	相当于一个事件监听器，当数据库发生特定事件后，触发器被触发，完成相应的处理

因为存在上面几种数据库对象，所以 create 后可以紧跟不同的关键字。例如，建表应使用 create table，建索引应使用 create index，建视图应使用 create view……drop 和 alter 后也需要添加类似的关键字来表示删除、修改哪种数据库对象。

提示：

因为函数、存储过程和触发器属于数据库编程内容，而且需要大量使用数据库特性，这已经超出了本书的介绍范围，故本章不打算介绍函数、存储过程和触发器编程。

1. 创建表的语法

标准的建表语句的语法如下：

```
create table [模式名.]表名
(
    # 可以有多个列定义
    columnName1 datatype [default expr] ,
    ...
)
```

上面语法中圆括号里可以包含多个列定义，每个列定义之间以英文逗号（,）隔开，最后一个列定义不需要使用英文逗号，而是直接以括号结束。

前面已经讲过，建立数据表只是建立表结构，就是指定该数据表有多少列，每列的数据类型，所以建表语句的重点就是圆括号里的列定义，列定义由列名、列类型和可选的默认值组成。

列定义有点类似于 Java 里的变量定义，与变量定义不同的是，列定义时将列名放在前面，列类型放在后面。如果要指定列的默认值，则使用 default 关键字，而不是使用等号（=）。

例如下面的建表语句：

```
create table test
(
    # 整型通常用 int
    test_id int,
    # 小数点数
    test_price decimal,
    # 普通长度文本，使用 default 指定默认值
    test_name varchar(255) default 'xxx',
    # 大文本类型
    test_desc text,
    # 图片
    test_img blob,
    test_date datetime
);
```

建表时需要指定每列的数据类型，不同数据库所支持的列类型不同，这需要查阅不同数据库的相关文档。MySQL 支持如表 13.2 所示的几种列类型。

表 13.2 MySQL 支持的列类型

列 类 型	说 明
tinyint/smallint/mediumint int(integer)/bigint	1 字节/2 字节/3 字节/4 字节/8 字节整数，又可分为有符号和无符号两种。这些整数类型的区别仅仅是表数范围不同
float/double	单精度、双精度浮点类型
decimal(dec)	精确小数类型，相对于 float 和 double 不会产生精度丢失的问题
date	日期类型，不能保存时间。把 java.util.Date 对象保存进 date 列时，时间部分将会丢失
time	时间类型，不能保存日期。把 java.util.Date 对象保存进 time 列时，日期部分将会丢失
datetime	日期、时间类型
timestamp	时间戳类型
year	年类型，仅仅保存时间的年份
char	定长字符串类型
varchar	可变长度字符串类型
binary	定长二进制字符串类型，它以二进制形式保存字符串

续表

列 类 型	说 明
varbinary	可变长度的二进制字符串类型，它以二进制形式保存字符串
tinyblob/blob mediumblob/longblob	1 字节/2 字节/3 字节/4 字节的二进制大对象，可用于存储图片、音乐等二进制数据，分别可存储：255B/64KB/16MB/4GB 的大小
tinytext/text mediumtext/longtext	1 字节/2 字节/3 字节/4 字节的文本对象，可用于存储超长长度的字符串，分别可存储：255B/64KB/16MB/4GB 大小的文本
enum('value1','value2',...)	枚举类型，该列的值只能是 enum 后括号里多个值的其中之一
set('value1','value2',...)	集合类型，该列的值可以是 set 后括号里多个值的其中几个

上面是比较常见的建表语句，这种建表语句只是创建一个空表，该表里没有任何数据。如果使用子查询建表语句，则可以在建表的同时插入数据。子查询建表语句的语法如下：

```
create table [模式名.]表名 [column[, column...]]
as subquery;
```

上面语法中新表的字段列表必须与子查询中的字段列表数量匹配，创建新表时的字段列表可以省略，如果省略了该字段列表，则新表的列名与选择结果完全相同。下面语句使用子查询来建表。

```
# 创建 hehe 数据表，该数据表和 user_inf 完全相同，数据也完全相同
create table hehe
as
select * from user_inf;
```

因为上面语句是利用子查询来建立数据表，所以执行该 SQL 语句要求数据库中已存在 user_inf 数据表（读者可向 test 数据库中导入 codes\13\13.2 目录下的 user_inf.sql 脚本后执行上面命令），否则程序将出现错误。

提示：

当数据表创建成功后，MySQL 使用 information_schema 数据库里的 TABLES 表来保存该数据库实例中的所有数据表，用户可通过查询 TABLES 表来获取该数据库的表信息。

2. 修改表结构的语法

修改表结构使用 alter table，修改表结构包括增加列定义、修改列定义、删除列、重命名列等操作。增加列定义的语法如下：

```
alter table 表名
add
(
    # 可以有多个列定义
    column_name1 datatype [default expr] ,
    ...
);
```

上面的语法格式中圆括号部分与建表语法的圆括号部分完全相同，只是此时圆括号里的列定义是追加到已有表的列定义后面。还有一点需要指出，如果只是新增一列，则可以省略圆括号，仅在 add 后紧跟一个列定义即可。为数据表增加字段的 SQL 语句如下：

```
# 为 hehe 数据表增加一个 hehe_id 字段，该字段的类型为 int
alter table hehe
add hehe_id int;
# 为 hehe 数据表增加 aaa、bbb 字段，两个字段的类型都为 varchar(255)
alter table hehe
add
(
    aaa varchar(255) default 'xxx',
    bbb varchar(255)
);
```

上面第二条 SQL 语句增加 aaa 字段时，为该字段指定默认值为'xxx'。值得指出的是，SQL 语句中

的字符串值不是用双引号引起，而是用单引号引起的。

增加字段时需要注意：如果数据表中已有数据记录，除非给新增的列指定了默认值，否则新增的数据列不可指定非空约束，因为那些已有的记录在新增列上肯定是空（实际上，修改表结构很容易失败，只要新增的约束与已有数据冲突，修改就会失败）。

修改列定义的语法如下：

```
alter table 表名
modify column_name datatype [default expr] [first|after col_name];
```

上面语法中 first 或者 after col_name 指定需要将目标修改到指定位置。

从上面修改语法中可以看出，该修改语句每次只能修改一个列定义，如下代码所示：

```
# 将 hehe 表的 hehe_id 列修改成 varchar(255)类型
alter table hehe
modify hehe_id varchar(255);
# 将 hehe 表的 bbb 列修改成 int 类型
alter table hehe
modify bbb int;
```

从上面代码中不难看出，使用 SQL 修改数据表里列定义的语法和为数据表只增加一个列定义的语法几乎完全一样，关键是增加列定义使用 add 关键字，而修改列定义使用 modify 关键字。还有一点需要指出，add 新增的列名必须是原表中不存在的，而 modify 修改的列名必须是原表中已存在的。

注意：

虽然 MySQL 的一个 modify 命令不支持一次修改多个列定义，但其他数据库如 Oracle 支持一个 modify 命令修改多个列定义，一个 modify 命令修改多个列定义的语法和一个 add 命令增加多个列定义的语法非常相似，也需要使用圆括号把多个列定义括起来。如果需要让 MySQL 支持一次修改多个列定义，则可在 alter table 后使用多个 modify 命令。

如果数据表里已有数据记录，则修改列定义非常容易失败，因为有可能修改的列定义规则与原有的数据记录不符合。如果修改数据列的默认值，则只会对以后的插入操作有作用，对以前已经存在的数据不会有任何影响。

从数据表中删除列的语法比较简单：

```
alter table 表名
drop column_name
```

删除列只要在 drop 后紧跟需要删除的列名即可。例如：

```
# 删除 hehe 表中的 aaa 字段
alter table hehe
drop aaa;
```

从数据表中删除列定义通常总是可以成功，删除列定义时将从每行中删除该列的数据，并释放该列在数据块中占用的空间。所以删除大表中的字段时需要比较长的时间，因为还需要回收空间。

上面介绍的这些增加列、修改列和删除列的语法是标准的 SQL 语法，对所有的数据库都通用。除此之外，MySQL 还提供了两种特殊的语法：重命名数据表和完全改变列定义。

重命名数据表的语法格式如下：

```
alter table 表名
rename to 新表名
```

如下 SQL 语句用于将 hehe 表命名为 wawa：

```
# 将 hehe 数据表重命名为 wawa
alter table hehe
rename to wawa;
```

MySQL 为 alter table 提供了 change 选项，该选项可以改变列名。change 选项的语法如下：

```
alter table 表名
```

```
change old_column_name new_column_name type [default expr] [first|after col_name]
```

对比 change 和 modify 两个选项，不难发现：change 选项比 modify 选项多了一个列名，因为 change 选项可以改变列名，所以它需要两个列名。一般而言，如果不需要改变列名，使用 alter table 的 modify 选项即可，只有当需要修改列名时才会使用 change 选项。如下语句所示：

```
# 将 wawa 数据表的 bbb 字段重命名为 ddd
alter table wawa
change bbb ddd int;
```

3. 删除表的语法

删除表的语法格式如下：

```
drop table 表名;
```

如下 SQL 语句将会把数据库中已有的 wawa 数据表删除：

```
# 删除数据表
drop table wawa;
```

删除数据表的效果如下。

- 表结构被删除，表对象不再存在。
- 表里的所有数据也被删除。
- 该表所有相关的索引、约束也被删除。

4. truncate 表

对于大部分数据库而言，truncate 都被当成 DDL 处理，truncate 被称为“截断”某个表——它的作用是删除该表里的全部数据，但保留表结构。相对于 DML 里的 delete 命令而言，truncate 的速度要快得多，而且 truncate 不像 delete 可以删除指定的记录，truncate 只能一次性删除整个表的全部记录。truncate 命令的语法如下：

```
truncate 表名
```

MySQL 对 truncate 的处理比较特殊——如果使用非 InnoDB 存储机制，truncate 比 delete 速度要快；如果使用 InnoDB 存储机制，在 MySQL 5.0.3 之前，truncate 和 delete 完全一样，在 5.0.3 之后，truncate table 比 delete 效率高，但如果该表被外键约束所参照，truncate 又变为 delete 操作。在 5.0.13 之后，快速 truncate 总是可用，即比 delete 性能要好。

13.2.5 数据库约束

前面创建的数据表仅仅指定了一些列定义，这仅仅是数据表的基本功能。除此之外，所有的关系数据库都支持对数据表使用约束，通过约束可以更好地保证数据表里数据的完整性。约束是在表上强制执行的数据校验规则，约束主要用于保证数据库里数据的完整性。除此之外，当表中数据存在相互依赖性时，可以保护相关的数据不被删除。

大部分数据库支持下面 5 种完整性约束。

- NOT NULL：非空约束，指定某列不能为空。
- UNIQUE：唯一约束，指定某列或者几列组合不能重复。
- PRIMARY KEY：主键，指定该列的值可以唯一地标识该条记录。
- FOREIGN KEY：外键，指定该行记录从属于主表中的一条记录，主要用于保证参照完整性。
- CHECK：检查，指定一个布尔表达式，用于指定对应列的值必须满足该表达式。

虽然大部分数据库都支持上面 5 种约束，但 MySQL 不支持 CHECK 约束，虽然 MySQL 的 SQL 语句也可以使用 CHECK 约束，但这个 CHECK 约束不会有任何作用。

虽然约束的作用只是用于保证数据表里数据的完整性，但约束也是数据库对象，并被存储在系统表中，也拥有自己的名字。根据约束对数据列的限制，约束分为如下两类。

- 单列约束：每个约束只约束一列。
- 多列约束：每个约束可以约束多个数据列。

为数据表指定约束有如下两个时机。

- 建表的同时为相应的数据列指定约束。
- 建表后创建，以修改表的方式来增加约束。

大部分约束都可以采用列级约束语法或者表级约束语法。下面依次介绍 5 种约束的建立和删除（约束通常无法修改）。

提示：

MySQL 使用 information_schema 数据库里的 TABLE_CONSTRAINTS 表来保存该数据库实例中所有的约束信息，用户可以通过查询 TABLE_CONSTRAINTS 表来获取该数据库的约束信息。

1. NOT NULL 约束

非空约束用于确保指定列不允许为空，非空约束是比较特殊的约束，它只能作为列级约束使用，只能使用列级约束语法定义。这里要介绍一下 SQL 中的 null 值，SQL 中的 null 不区分大小写。SQL 中的 null 具有如下特征。

- 所有数据类型的值都可以是 null，包括 int、float、boolean 等数据类型。
- 与 Java 类似的是，空字符串不等于 null，0 也不等于 null。

如果需要在建表时为指定列指定非空约束，只要在列定义后增加 not null 即可。建表语句如下：

```
create table hehe
(
    # 建立了非空约束，这意味着 hehe_id 不可以为 null
    hehe_id int not null,
    # MySQL 的非空约束不能指定名字
    hehe_name varchar(255) default 'xyz' not null,
    # 下面列可以为空，默认就是可以为空
    hehe_gender varchar(2) null
);
```

除此之外，也可以在使用 alter table 修改表时增加或者删除非空约束，SQL 命令如下：

```
# 增加非空约束
alter table hehe
modify hehe_gender varchar(2) not null
# 取消非空约束
alter table hehe
modify hehe_name varchar(2) null;
# 取消非空约束，并指定默认值
alter table hehe
modify hehe_name varchar(255) default 'abc' null;
```

2. UNIQUE 约束

唯一约束用于保证指定列或指定列组合不允许出现重复值。虽然唯一约束的列不可以出现重复值，但可以出现多个 null 值（因为在数据库中 null 不等于 null）。

同一个表内可建多个唯一约束，唯一约束也可由多列组合而成。当为某列创建唯一约束时，MySQL 会为该列相应地创建唯一索引。如果不给唯一约束起名，该唯一约束默认与列名相同。

唯一约束既可以使用列级约束语法建立，也可以使用表级约束语法建立。如果需要为多列建组合约束，或者需要为唯一约束指定约束名，则只能用表级约束语法。

当建立唯一约束时，MySQL 在唯一约束所在列或列组合上建立对应的唯一索引。

使用列级约束语法建立唯一约束非常简单，只要简单地在列定义后增加 unique 关键字即可。SQL 语句如下：

```
# 建表时创建唯一约束，使用列级约束语法建立约束
create table unique_test
(
    # 建立了非空约束，这意味着 test_id 不可以为 null
    test_id int not null,
```

```
    # unique 就是唯一约束，使用列级约束语法建立唯一约束
    test_name varchar(255) unique
);
```

如果需要为多列组合建立唯一约束，或者想自行指定约束名，则需要使用表级约束语法。表级约束语法格式如下：

```
[constraint 约束名] 约束定义
```

上面的表级约束语法格式既可放在 create table 语句中与列定义并列，也可放在 alter table 语句中使用 add 关键字来添加约束。SQL 语句如下：

```
# 建表时创建唯一约束，使用表级约束语法建立约束
create table unique_test2
(
    # 建立了非空约束，这意味着 test_id 不可以为 null
    test_id int not null,
    test_name varchar(255),
    test_pass varchar(255),
    # 使用表级约束语法建立唯一约束
    unique (test_name),
    # 使用表级约束语法建立唯一约束，而且指定约束名
    constraint test2_uk unique(test_pass)
);
```

上面的建表语句为 test_name、test_pass 分别建立了唯一约束，这意味着这两列都不能出现重复值。除此之外，还可以为这两列组合建立唯一约束，SQL 语句如下：

```
# 建表时创建唯一约束，使用表级约束语法建立约束
create table unique_test3
(
    # 建立了非空约束，这意味着 test_id 不可以为 null
    test_id int not null,
    test_name varchar(255),
    test_pass varchar(255),
    # 使用表级约束语法建立唯一约束，指定两列组合不允许重复
    constraint test3_uk unique(test_name,test_pass)
);
```

对于上面的 unique_test2 和 unique_test3 两个表，都是对 test_name、test_pass 建立唯一约束，其中 unique_test2 要求 test_name、test_pass 都不能出现重复值，而 unique_test3 只要求 test_name、test_pass 两列值的组合不能重复。

也可以在修改表结构时使用 add 关键字来增加唯一约束，SQL 语句如下：

```
# 增加唯一约束
alter table unique_test3
add unique(test_name, test_pass);
```

还可以在修改表时使用 modify 关键字，为单列采用列级约束语法来增加唯一约束，代码如下：

```
# 为 unique_test3 表的 test_name 列增加唯一约束
alter table unique_test3
modify test_name varchar(255) unique;
```

对于大部分数据库而言，删除约束都是在 alter table 语句后使用“drop constraint 约束名”语法来完成的，但 MySQL 并不使用这种方式，而是使用“drop index 约束名”的方式来删除约束。例如如下 SQL 语句：

```
# 删除 unique_test3 表上的 test3_uk 唯一约束
alter table unique_test3
drop index test3_uk;
```

3. PRIMARY KEY 约束

主键约束相当于非空约束和唯一约束，即主键约束的列既不允许出现重复值，也不允许出现 null 值；如果对多列组合建立主键约束，则多列里包含的每一列都不能为空，但只要求这些列组合不能重复。

主键列的值可用于唯一地标识表中的一条记录。

每一个表中最多允许有一个主键，但这个主键约束可由多个数据列组合而成，主键是表中能唯一确定一行记录的字段或字段组合。

建立主键约束时既可使用列级约束语法，也可使用表级约束语法。如果需要对多个字段建立组合主键约束，则只能使用表级约束语法。使用表级约束语法来建立约束时，可以为该约束指定约束名。但不管用户是否为该主键约束指定约束名，MySQL 总是将所有的主键约束命名为 PRIMARY。

提示：

MySQL 允许在建立主键约束时为该约束命名，但这个名字没有任何作用，这是为了保持与标准 SQL 的兼容性。大部分数据库都允许自行指定主键约束的名字，而且一旦指定了主键约束名，则该约束名就是用户指定的名字。

当创建主键约束时，MySQL 在主键约束所在列或列组合上建立对应的唯一索引。

创建主键约束的语法和创建唯一约束的语法非常像，一样允许使用列级约束语法为单独的数据列创建主键，如果需要为多列组合建立主键约束或者需要为主键约束命名，则应该使用表级约束语法来建立主键约束。与建立唯一约束不同的是，建立主键约束使用 primary key。

建表时创建主键约束，使用列级约束语法：

```
create table primary_test
(
    # 建立了主键约束
    test_id int primary key,
    test_name varchar(255)
);
```

建表时创建主键约束，使用表级约束语法：

```
create table primary_test2
(
    test_id int not null,
    test_name varchar(255),
    test_pass varchar(255),
    # 指定主键约束名为 test2_pk，对大部分数据库有效，但对 MySQL 无效
    # MySQL 数据库中该主键约束名依然是 PRIMARY
    constraint test2_pk primary key(test_id)
);
```

建表时创建主键约束，以多列建立组合主键，只能使用表级约束语法：

```
create table primary_test3
(
    test_name varchar(255),
    test_pass varchar(255),
    # 建立多列组合的主键约束
    primary key(test_name, test_pass)
);
```

如果需要删除指定表的主键约束，则在 alter table 语句后使用 drop primary key 子句即可。SQL 语句如下：

```
# 删除主键约束
alter table primary_test3
drop primary key;
```

如果需要为指定表增加主键约束，既可通过 modify 修改列定义来增加主键约束，这将采用列级约束语法来增加主键约束；也可通过 add 来增加主键约束，这将采用表级约束语法来增加主键约束。SQL 语句如下：

```
# 使用表级约束语法增加主键约束
alter table primary_test3
add primary key(test_name,test_pass);
```

如果只是为单独的数据列增加主键约束，则可使用 modify 修改列定义来实现，如下 SQL 语句所示：

```
# 使用列级约束语法增加主键约束
alter table primary_test3
modify test_name varchar(255) primary key;
```

注意：

不要连续执行上面两条 SQL 语句，因为上面两条 SQL 语句都是为 primary_test3 增加主键约束，而同一个表里最多只能有一个主键约束，所以连续执行上面两条 SQL 语句肯定出现错误。为了避免这个问题，可以在成功执行了第一条增加主键约束的 SQL 语句之后，先将 primary_test3 里的主键约束删除后再执行第二条增加主键约束的 SQL 语句。

很多数据库对主键列都支持一种自增长的特性——如果某个数据列的类型是整型，而且该列作为主键列，则可指定该列具有自增长功能。指定自增长功能通常用于设置逻辑主键列——该列的值没有任何物理意义，仅仅用于标识每行记录。MySQL 使用 auto_increment 来设置自增长，SQL 语句如下：

```
create table primary_test4
(
    # 建立主键约束，使用自增长
    test_id int auto_increment primary key,
    test_name varchar(255),
    test_pass varchar(255)
);
```

一旦指定了某列具有自增长特性，则向该表插入记录时可不为该列指定值，该列的值由数据库系统自动生成。

4. FOREIGN KEY 约束

外键约束主要用于保证一个或两个数据表之间的参照完整性，外键是构建于一个表的两个字段或者两个表的两个字段之间的参照关系。外键确保了相关的两个字段的参照关系：子（从）表外键列的值必须在主表被参照列的值范围之内，或者为空（也可以通过非空约束来约束外键列不允许为空）。

当主表的记录被从表记录参照时，主表记录不允许被删除，必须先把从表里参照该记录的所有记录全部删除后，才可以删除主表的该记录。还有一种方式，删除主表记录时级联删除从表中所有参照该记录的从表记录。

从表外键参照的只能是主表主键列或者唯一键列，这样才可保证从表记录可以准确定位到被参照的主表记录。同一个表内可以拥有多个外键。

建立外键约束时，MySQL 也会为该列建立索引。

外键约束通常用于定义两个实体之间的一对多、一对一的关联关系。对于一对多的关联关系，通常在多的一端增加外键列，例如老师－学生（假设一个老师对应多个学生，但每个学生只有一个老师，这是典型的一对多的关联关系）。为了建立他们之间的关联关系，可以在学生表中增加一个外键列，该列中保存此条学生记录对应老师的主键。对于一对一的关联关系，则可选择任意一方来增加外键列，增加外键列的表被称为从表，只要为外键列增加唯一约束就可表示一对一的关联关系了。对于多对多的关联关系，则需要额外增加一个连接表来记录它们的关联关系。

建立外键约束同样可以采用列级约束语法和表级约束语法。如果仅对单独的数据列建立外键约束，则使用列级约束语法即可；如果需要对多列组合创建外键约束，或者需要为外键约束指定名字，则必须使用表级约束语法。

采用列级约束语法建立外键约束直接使用 references 关键字，references 指定该列参照哪个主表，以及参照主表的哪一列。如下 SQL 语句所示：

```
# 为了保证从表参照的主表存在，通常应该先建主表
create table teacher_table
(
    # auto_increment：代表数据库的自动编号策略，通常用作数据表的逻辑主键
    teacher_id int auto_increment,
    teacher_name varchar(255),
    primary key(teacher_id)
);
```

```
create table student_table
(
    # 为本表建立主键约束
    student_id int auto_increment primary key,
    student_name varchar(255),
    # 指定 java_teacher 参照到 teacher_table 的 teacher_id 列
    java_teacher int references teacher_table(teacher_id)
);
```

值得指出的是，虽然 MySQL 支持使用列级约束语法来建立外键约束，但这种列级约束语法建立的外键约束不会生效，MySQL 提供这种列级约束语法仅仅是为了和标准 SQL 保持良好的兼容性。因此，如果要使 MySQL 中的外键约束生效，则应使用表级约束语法。

```
# 为了保证从表参照的主表存在，通常应该先建主表
create table teacher_table1
(
    # auto_increment：代表数据库的自动编号策略，通常用作数据表的逻辑主键
    teacher_id int auto_increment,
    teacher_name varchar(255),
    primary key(teacher_id)
);
create table student_table1
(
    # 为本表建立主键约束
    student_id int auto_increment primary key,
    student_name varchar(255),
    # 指定 java_teacher 参照到 teacher_table1 的 teacher_id 列
    java_teacher int,
    foreign key(java_teacher) references teacher_table1(teacher_id)
);
```

如果使用表级约束语法，则需要使用 foreign key 来指定本表的外键列，并使用 references 来指定参照哪个主表，以及参照到主表的哪个数据列。使用表级约束语法可以为外键约束指定约束名，如果创建外键约束时没有指定约束名，则 MySQL 会为该外键约束命名为 table_name_ibfk_*n*，其中 table_name 是从表的表名，而 *n* 是从 1 开始的整数。

如果需要显式指定外键约束的名字，则可使用 constraint 来指定名字。如下 SQL 语句所示：

```
# 为了保证从表参照的主表存在，通常应该先建主表
create table teacher_table2
(
    # auto_increment：代表数据库的自动编号策略，通常用作数据表的逻辑主键
    teacher_id int auto_increment,
    teacher_name varchar(255),
    primary key(teacher_id)
);
create table student_table2
(
    # 为本表建立主键约束
    student_id int auto_increment primary key,
    student_name varchar(255),
    java_teacher int,
    # 使用表级约束语法建立外键约束，指定外键约束的约束名为 student_teacher_fk
    constraint student_teacher_fk foreign key(java_teacher) references
        teacher_table2(teacher_id)
);
```

如果需要建立多列组合的外键约束，则必须使用表级约束语法，如下 SQL 语句所示：

```
# 为了保证从表参照的主表存在，通常应该先建主表
create table teacher_table3
(
    teacher_name varchar(255),
    teacher_pass varchar(255),
    # 以两列建立组合主键
    primary key(teacher_name , teacher_pass)
);
create table student_table3
```

```
(
    # 为本表建立主键约束
    student_id int auto_increment primary key,
    student_name varchar(255),
    java_teacher_name varchar(255),
    java_teacher_pass varchar(255),
    # 使用表级约束语法建立外键约束，指定两列的联合外键
    foreign key(java_teacher_name , java_teacher_pass)
        references teacher_table3(teacher_name , teacher_pass)
);
```

删除外键约束的语法很简单，在 alter table 后增加“drop foreign key 约束名”子句即可。如下代码所示：

```
# 删除 student_table3 表上名为 student_table3_ibfk_1 的外键约束
alter table student_table3
drop foreign key student_table3_ibfk_1;
```

增加外键约束通常使用 add foreign key 命令。如下 SQL 语句所示：

```
# 修改 student_table3 数据表，增加外键约束
alter table student_table3
add foreign key(java_teacher_name , java_teacher_pass)
    references teacher_table3(teacher_name , teacher_pass);
```

值得指出的是，外键约束不仅可以参照其他表，而且可以参照自身，这种参照自身的情况通常被称为自关联。例如，使用一个表保存某个公司的所有员工记录，员工之间有部门经理和普通员工之分，部门经理和普通员工之间存在一对多的关联关系，但他们都是保存在同一个数据表里的记录，这就是典型的自关联。下面的 SQL 语句用于建立自关联的外键约束。

```
# 使用表级约束语法建立外约束键，且直接参照自身
create table foreign_test
(
    foreign_id int auto_increment primary key,
    foreign_name varchar(255),
    # 使用该表的 refer_id 参照到本表的 foreign_id 列
    refer_id int,
    foreign key(refer_id) references foreign_test(foreign_id)
);
```

如果想定义当删除主表记录时，从表记录也会随之删除，则需要在建立外键约束后添加 on delete cascade 或添加 on delete set null，第一种是删除主表记录时，把参照该主表记录的从表记录全部级联删除；第二种是指定当删除主表记录时，把参照该主表记录的从表记录的外键设为 null。如下 SQL 语句所示：

```
# 为了保证从表参照的主表存在，通常应该先建主表
create table teacher_table4
(
    # auto_increment：代表数据库的自动编号策略，通常用作数据表的逻辑主键
    teacher_id int auto_increment,
    teacher_name varchar(255),
    primary key(teacher_id)
);
create table student_table4
(
    # 为本表建立主键约束
    student_id int auto_increment primary key,
    student_name varchar(255),
    java_teacher int,
    # 使用表级约束语法建立外键约束，定义级联删除
    foreign key(java_teacher) references teacher_table4(teacher_id)
        on delete cascade # 也可用 on delete set null
);
```

5. CHECK 约束

当前版本的MySQL支持建表时指定CHECK约束，但这个CHECK约束不会有任何作用。建立CHECK约束的语法很简单，只要在建表的列定义后增加 check（逻辑表达式）即可。如下 SQL 语句所示：

```
create table check_test
(
    emp_id int auto_increment,
    emp_name varchar(255),
    emp_salary decimal,
    primary key(emp_id),
    # 建立 CHECK 约束
    check(emp_salary>0)
);
```

虽然上面的 SQL 语句建立的 check_test 表中有 CHECK 约束，CHECK 约束要求 emp_salary 大于 0，但这个要求实际上并不会起作用。

提示：

MySQL 作为一个开源、免费的数据库系统，对有些功能的支持确实不太好。如果读者确实希望 MySQL 创建的数据表有 CHECK 约束，甚至有更复杂的完整性约束，则可借助于 MySQL 的触发器机制。本书不会介绍 MySQL 的触发器内容，读者可参考其他相关书籍。

13.2.6　索引

索引是存放在模式（schema）中的一个数据库对象，虽然索引总是从属于数据表，但它也和数据表一样属于数据库对象。创建索引的唯一作用就是加速对表的查询，索引通过使用快速路径访问方法来快速定位数据，从而减少了磁盘的 I/O。

索引作为数据库对象，在数据字典中独立存放，但不能独立存在，必须属于某个表。

提示：

MySQL 使用 information_schema 数据库里的 STATISTICS 表来保存该数据库实例中的所有索引信息，用户可通过查询该表来获取该数据库的索引信息。

创建索引有两种方式。

- 自动：当在表上定义主键约束、唯一约束和外键约束时，系统会为该数据列自动创建对应的索引。
- 手动：用户可以通过 create index...语句来创建索引。

删除索引也有两种方式。

- 自动：数据表被删除时，该表上的索引自动被删除。
- 手动：用户可以通过 drop index...语句来删除指定数据表上的指定索引。

索引的作用类似于书的目录，几乎没有一本书没有目录，因此几乎没有一个表没有索引。一个表中可以有多个索引列，每个索引都可用于加速该列的查询速度。

正如书的目录总是根据书的知识点来建立一样——因为读者经常要根据知识点来查阅一本书。类似的，通常为经常需要查询的数据列建立索引，可以在一列或者多列上创建索引。创建索引的语法格式如下：

```
create index index_name
on table_name (column[, column]...);
```

下面的索引将会提高对 employees 表基于 last_name 字段的查询速度。

```
create index emp_last_name_idx
on employees(last_name);
```

也可同时对多列建立索引，如下 SQL 语句所示：

```
# 下面语句为 employees 的 first_name 和 last_name 两列同时建立索引
create index emp_last_name_idx2
```

```
on employees(first_name, last_name);
```

MySQL 中删除索引需要指定表，采用如下语法格式：

```
drop index 索引名 on 表名
```

如下 SQL 语句删除了 employees 表上的 emp_last_name_idx2 索引：

```
drop index emp_last_name_idx2
on employees
```

有些数据库删除索引时无须指定表名，因为它们要求建立索引时每个索引都有唯一的名字，所以无须指定表名，例如 Oracle 就采用这种策略。但 MySQL 只要求同一个表内的索引不能同名，所以删除索引时必须指定表名。

索引的好处是可以加速查询。但索引也有如下两个坏处。

- 与书的目录类似，当数据表中的记录被添加、删除、修改时，数据库系统需要维护索引，因此有一定的系统开销。
- 存储索引信息需要一定的磁盘空间。

13.2.7 视图

视图看上去非常像一个数据表，但它不是数据表，因为它并不能存储数据。视图只是一个或多个数据表中数据的逻辑显示。使用视图有如下几个好处。

- 可以限制对数据的访问。
- 可以使复杂的查询变得简单。
- 提供了数据的独立性。
- 提供了对相同数据的不同显示。

因为视图只是数据表中数据的逻辑显示——也就是一个查询结果，所以创建视图就是建立视图名和查询语句的关联。创建视图的语法如下：

```
create or replace view 视图名
as
subquery
```

从上面的语法可以看出，创建、修改视图都可使用上面语法。上面语法的含义是，如果该视图不存在，则创建视图；如果指定视图名的视图已经存在，则使用新视图替换原有视图。后面的 subquery 就是一个查询语句，这个查询可以非常复杂。

注意：

通过建立视图的语法规则不难看出，所谓视图的本质，其实就是一条被命名的 SQL 查询语句。

一旦建立了视图以后，使用该视图与使用数据表就没有什么区别了，但通常只是查询视图数据，不会修改视图里的数据，因为视图本身没有存储数据。

如下 SQL 语句就创建了一个简单的视图：

```
create or replace view view_test
as
select teacher_name , teacher_pass from teacher_table;
```

通常不推荐直接改变视图的数据，因为视图并不存储数据，它只是相当于一条命名的查询语句而已。为了强制不允许改变视图的数据，MySQL 允许在创建视图时使用 with check option 子句，使用该子句创建的视图不允许修改，如下所示：

```
create or replace view view_test
as
select teacher_name from teacher_table
# 指定不允许修改该视图的数据
with check option;
```

大部分数据库都采用 with check option 来强制不允许修改视图的数据，但 Oracle 采用 with read only 来强制不允许修改视图的数据。

删除视图使用如下语句：

```
drop view 视图名
```

如下 SQL 语句删除了前面刚刚创建的视图：

```
drop view view_test;
```

13.2.8 DML 语句语法

与 DDL 操作数据库对象不同，DML 主要操作数据表里的数据，使用 DML 可以完成如下三个任务。

- 插入新数据。
- 修改已有数据。
- 删除不需要的数据。

DML 语句由 insert into、update 和 delete from 三个命令组成。

1. insert into 语句

insert into 用于向指定数据表中插入记录。对于标准的 SQL 语句而言，每次只能插入一条记录。insert into 语句的语法格式如下：

```
insert into table_name [(column [, column...])]
values(value [, value...]);
```

执行插入操作时，表名后可以用括号列出所有需要插入值的列名，而 values 后用括号列出对应需要插入的值。

如果省略了表名后面的括号及括号里的列名列表，默认将为所有列都插入值，则需要为每一列都指定一个值。如果既不想在表名后列出列名，又不想为所有列都指定值，则可以为那些无法确定值的列分配 null。下面的 SQL 语句示范了如何向数据表中插入记录。

只有在数据库中已经成功创建了数据表之后，才可以向数据表中插入记录。下面的 SQL 语句以前面介绍外键约束时所创建的 teacher_table2 和 student_table2 为例来介绍数据插入操作。

在表名后使用括号列出所有需要插入值的列：

```
insert into teacher_table2(teacher_name)
values('xyz');
```

如果不想在表后用括号列出所有列，则需要为所有列指定值；如果某列的值不能确定，则为该列分配一个 null 值。

```
insert into teacher_table2
# 使用 null 代替主键列的值
values(null , 'abc');
```

经过两条插入语句后，可以看到 teacher_table2 表中的数据如图 13.9 所示。

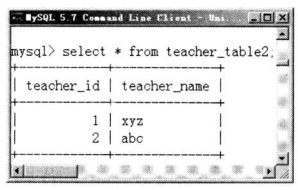


图 13.9 插入 2 条记录

从图 13.9 中看到 abc 记录的主键列的值是 2，而不是 SQL 语句插入的 null，因为该主键列是自增长的，系统会自动为该列分配值。

根据前面介绍的外键约束规则：外键列里的值必须是被参照列里已有的值，所以向从表中插入记录之前，通常应该先向主表中插入记录，

否则从表记录的外键列只能为 null。现在主表 teacher_table2 中已有了 2 条记录，现在可以向从表 student_table2 中插入记录了，如下 SQL 语句所示：

```
insert into student_table2
# 当向外键列里插值时，外键列的值必须是被参照列里已有的值
values(null , '张三' , 2);
```

注意：

外键约束保证被参照的记录必须存在，但并不保证必须有被参照记录，即外键列可以为 null。如果想保证每条从表记录必须存在对应的主表记录，则应使用非空、外键两个约束。

在一些特别的情况下，可以使用带子查询的插入语句，带子查询的插入语句可以一次插入多条记录，如下 SQL 语句所示：

```
insert into student_table2(student_name)
# 使用子查询的值来插入
select teacher_name from teacher_table2;
```

正如上面的 SQL 语句所示，带子查询的插入语句甚至不要求查询数据的源表和插入数据的目的表是同一个表，它只要求选择出来的数据列和插入目的表的数据列个数相等、数据类型匹配即可。

MySQL 甚至提供了一种扩展的语法，通过这种扩展的语法也可以一次插入多条记录。MySQL 允许在 values 后使用多个括号包含多条记录，表示多条记录的多个括号之间以英文逗号（,）隔开。如下 SQL 语句所示：

```
insert into teacher_table2
# 同时插入多个值
values(null , "Yeeku"),
(null , "Sharfly");
```

2. update 语句

update 语句用于修改数据表的记录，每次可以修改多条记录，通过使用 where 子句限定修改哪些记录。where 子句是一个条件表达式，该条件表达式类似于 Java 语言的 if，只有符合该条件的记录才会被修改。没有 where 子句则意味着 where 表达式的值总是 true，即该表的所有记录都会被修改。update 语句的语法格式如下：

```
update table_name
set column1= value1[, column2 = value2] …
[WHERE condition];
```

使用 update 语句不仅可以一次修改多条记录，也可以一次修改多列。修改多列都是通过在 set 关键字后使用 column1=value1,column2=value2…来实现的，修改多列的值之间以英文逗号（,）隔开。

下面的 SQL 语句将会把 teacher_table2 表中所有记录的 teacher_name 列的值都改为'孙悟空'。

```
update teacher_table2
set teacher_name = '孙悟空';
```

也可以通过添加 where 条件来指定只修改特定记录，如下 SQL 语句所示：

```
# 只修改 teacher_id 大于 1 的记录
update teacher_table2
set teacher_name = '猪八戒'
where teacher_id > 1;
```

3. delete from 语句

delete from 语句用于删除指定数据表的记录。使用 delete from 语句删除时不需要指定列名，因为总是整行地删除。

使用 delete from 语句可以一次删除多行，删除哪些行采用 where 子句限定，只删除满足 where 条件的记录。没有 where 子句限定将会把表里的全部记录删除。

delete from 语句的语法格式如下：

```
delete from table_name
[WHERE condition];
```

如下 SQL 语句将会把 student_table2 表中的记录全部删除：

```
delete from student_table2;
```

也可以使用 where 条件来限定只删除指定记录，如下 SQL 语句所示：

```
delete from teacher_table2
where teacher_id > 2;
```

注意：

当主表记录被从表记录参照时，主表记录不能被删除，只有先将从表中参照主表记录的所有记录全部删除后，才可删除主表记录。还有一种情况，定义外键约束时定义了主表记录和从表记录之间的级联删除 on delete cascade，或者使用 on delete set null 用于指定当主表记录被删除时，从表中参照该记录的从表记录把外键列的值设为 null。

13.2.9　单表查询

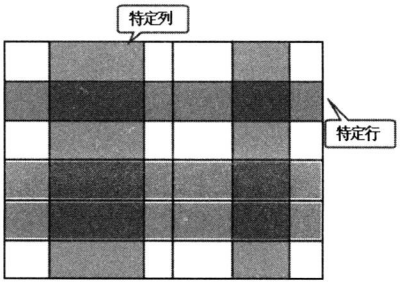


图 13.10　select 语句选择特定行、特定列的示意图

select 语句的功能就是查询数据。select 语句也是 SQL 语句中功能最丰富的语句，select 语句不仅可以执行单表查询，而且可以执行多表连接查询，还可以进行子查询，select 语句用于从一个或多个数据表中选出特定行、特定列的交集。select 语句最简单的功能如图 13.10 所示。

单表查询的 select 语句的语法格式如下：

```
select column1, column2 ...
from 数据源
[where condition]
```

上面语法格式中的数据源可以是表、视图等。从上面的语法格式中可以看出，select 后的列表用于确定选择哪些列，where 条件用于确定选择哪些行，只有满足 where 条件的记录才会被选择出来；如果没有 where 条件，则默认选出所有行。如果想选择出所有列，则可使用星号（*）代表所有列。

下面的 SQL 语句将会选择出 teacher_table 表中的所有行、所有列的数据。

```
select *
from teacher_table;
```

提示：

为了能看到查询的效果，必须准备数据表，并向数据表中插入一些数据，因此在运行本节的 select 语句之前，请先导入 codes\13\13.2\select_data.sql 文件中的 SQL 语句。

如果增加 where 条件，则只选择出符合 where 条件的记录。如下 SQL 语句将选择出 student_table 表中 java_teacher 值大于 3 的记录的 student_name 列的值。

```
select student_name
from student_table
where java_teacher > 3;
```

当使用 select 语句进行查询时，还可以在 select 语句中使用算术运算符（+、－、*、/），从而形成算术表达式。使用算术表达式的规则如下。

- 对数值型数据列、变量、常量可以使用算术运算符（+、－、*、/）创建表达式；
- 对日期型数据列、变量、常量可以使用部分算术运算符（+、－）创建表达式，两个日期之间可以进行减法运算，日期和数值之间可以进行加、减运算；
- 运算符不仅可以在列和常量、变量之间进行运算，也可以在两列之间进行运算。

不论从哪个角度来看，数据列都很像一个变量，只是这个变量的值具有指定的范围——逐行计算表中的每条记录时，数据列的值依次变化。因此能使用变量的地方，基本上都可以使用数据列。

下面的 select 语句中使用了算术运算符。

```
# 数据列实际上可当成一个变量
select teacher_id + 5
from teacher_table;
# 查询出 teacher_table 表中 teacher_id * 3 大于 4 的记录
select *
from teacher_table
where teacher_id * 3 > 4;
```

需要指出的是，select 后的不仅可以是数据列，也可以是表达式，还可以是变量、常量等。例如，如下语句也是正确的。

```
# 数据列实际上可当成一个变量
select 3*5, 20
from teacher_table;
```

SQL 语句中算术运算符的优先级与 Java 语言中的运算符优先级完全相同，乘法和除法的优先级高于加法和减法，同级运算的顺序是从左到右，表达式中使用括号可强行改变优先级的运算顺序。

MySQL 中没有提供字符串连接运算符，即无法使用加号（+）将字符串常量、字符串变量或字符串列连接起来。MySQL 使用 concat 函数来进行字符串连接运算。

如下 SQL 语句所示：

```
# 选择出 teacher_name 和'xx'字符串连接后的结果
select concat(teacher_name ,'xx')
from teacher_table;
```

对于 MySQL 而言，如果在算术表达式中使用 null，将会导致整个算术表达式的返回值为 null；如果在字符串连接运算中出现 null，将会导致连接后的结果也是 null。如下 SQL 语句将会返回 null。

```
select concat(teacher_name , null)
from teacher_table;
```

对某些数据库而言，如果让字符串和 null 进行连接运算，它会把 null 当成空字符串处理。

如果不希望直接使用列名作为列标题，则可以为数据列或表达式起一个别名，为数据列或表达式起别名时，别名紧跟数据列，中间以空格隔开，或者使用 as 关键字隔开。如下 SQL 语句所示：

```
select teacher_id + 5 as MY_ID
from teacher_table;
```

执行此条 SQL 语句的效果如图 13.11 所示。

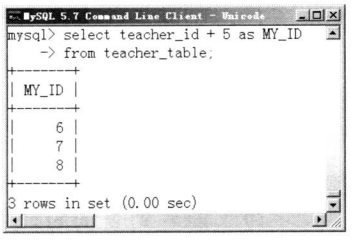


图 13.11 为数据列起别名

从图 13.11 中可以看出，为列起别名，可以改变列的标题头，用于表示计算结果的具体含义。如果列别名中使用特殊字符（例如空格），或者需要强制大小写敏感，都可以通过为别名添加双引号来实现。如下 SQL 语句所示：

```
# 可以为选出的列起别名，别名中包括单引号字符，所以把别名用双引号引起来
select teacher_id + 5  "MY'id"
from teacher_table;
```

如果需要选择多列，并为多列起别名，则列与列之间以逗号隔开，但列和列别名之间以空格隔开。如下 SQL 语句所示：

```
select teacher_id + 5 MY_ID , teacher_name 老师名
from teacher_table;
```

不仅可以为列或表达式起别名，也可以为表起别名，为表起别名的语法和为列或表达式起别名的语法完全一样，如下 SQL 语句所示：

```
select teacher_id + 5 MY_ID , teacher_name 老师名
# 为 teacher_table 起别名 t
from teacher_table t
```

前面已经提到，列名可以当成变量处理，所以运算符也可以在多列之间进行运算，如下 SQL 语句所示：

```
select teacher_id + 5 MY_ID , concat(teacher_name , teacher_id) teacher_name
from teacher_table
where teacher_id * 2 > 3;
```

甚至可以在 select、where 子句中都不出现列名，如下 SQL 语句所示：

```
select 5 + 4
from teacher_table
where 2 < 9;
```

这种情况比较特殊：where 语句后的条件表达式总是 true，所以会把 teacher_table 表中的每条记录都选择出来——但 SQL 语句没有选择任何列，仅仅选择了一个常量，所以 SQL 会把该常量当成一列，teacher_table 表中有多少条记录，该常量就出现多少次。运行上面的 SQL 语句，结果如图 13.12 所示。

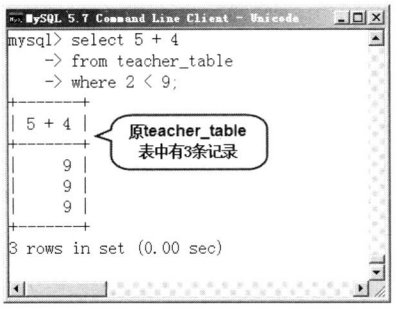


图 13.12　选择常量的结果

对于选择常量的情形，指定数据表可能没有太大的意义，所以 MySQL 提供了一种扩展语法，允许 select 语句后没有 from 子句，即可写成如下形式：

```
select 5 + 4;
```

上面这种语句并不是标准 SQL 语句。例如，Oracle 就提供了一个名为 dual 的虚表（最新的 MySQL 数据库也支持 dual 虚表），它没有任何意义，仅仅相当于 from 后的占位符。如果选择常量，则可使用如下语句：

```
select 5+4 from dual;
```

select 默认会把所有符合条件的记录全部选出来，即使两行记录完全一样。如果想去除重复行，则可以使用 distinct 关键字从查询结果中清除重复行。比较下面两条 SQL 语句的执行结果：

```
# 选出所有记录，包括重复行
select student_name,java_teacher
from student_table;

# 去除重复行
select distinct student_name,java_teacher
from student_table;
```

注意：

使用 distinct 去除重复行时，distinct 紧跟 select 关键字。它的作用是去除后面字段组合的重复值，而不管对应记录在数据库里是否重复。例如，(1 , 'a' , 'b')和(2 , 'a' , 'b')两条记录在数据库里是不重复的，但如果仅选择后面两列，则 distinct 会认为两条记录重复。

前面已经看到了 where 子句的作用——可以控制只选择指定的行。因为 where 子句里包含的是一个条件表达式，所以可以使用>、>=、<、<=、=和<>等基本的比较运算符。SQL 中的比较运算符不仅可以比较数值之间的大小，也可以比较字符串、日期之间的大小。

注意：

SQL 中判断两个值是否相等的比较运算符是单等号，判断不相等的运算符是<>；SQL 中的赋值运算符不是等号，而是冒号等号（:=）。

除此之外，SQL 还支持如表 13.3 所示的特殊的比较运算符。

表 13.3 特殊的比较运算符

运 算 符	含 义
expr1 between expr2 and expr3	要求 expr1 >= expr2 并且 expr2<=expr3
expr1 in(expr2 , expr3 , expr4 , ...)	要求 expr1 等于后面括号里任意一个表达式的值
like	字符串匹配，like 后的字符串支持通配符
is null	要求指定值等于 null

下面的 SQL 语句选出 student_id 大于等于 2，且小于等于 4 的所有记录。

```
select * from student_table
where student_id between 2 and 4;
```

使用 between val1 and val2 必须保证 val1 小于 val2，否则将选不出任何记录。除此之外，between val1 and val2 中的两个值不仅可以是常量，也可以是变量，或者是列名也行。如下 SQL 语句选出 java_teacher 小于等于 2，student_id 大于等于 2 的所有记录。

```
select * from student_table
where 2 between java_teacher and student_id;
```

使用 in 比较运算符时，必须在 in 后的括号里列出一个或多个值，它要求指定列必须与 in 括号里任意一个值相等。如下 SQL 语句所示：

```
# 选出 student_id 为 2 或 4 的所有记录
select * from student_table
where student_id in(2, 4);
```

与之类似的是，in 括号里的值既可以是常量，也可以是变量或者列名，如下 SQL 语句所示：

```
# 选出 student_id、java_teacher 列的值为 2 的所有记录
select * from student_table
where 2 in(student_id, java_teacher);
```

like 运算符主要用于进行模糊查询，例如，若要查询名字以“孙”开头的所有记录，这就需要用到模糊查询，在模糊查询中需要使用 like 关键字。SQL 语句中可以使用两个通配符：下画线（_）和百分号（%），其中下画线可以代表一个任意的字符，百分号可以代表任意多个字符。如下 SQL 语句将查询出所有学生中名字以“孙”开头的学生。

```
select * from student_table
where student_name like '孙%';
```

下面的 SQL 语句将查询出名字为两个字符的所有学生。

```
select * from student_table
# 下面使用两个下画线代表两个字符
where student_name like '__';
```

在某些特殊的情况下，查询的条件里需要使用下画线或百分号，不希望 SQL 把下画线和百分号当成通配符使用，这就需要使用转义字符，MySQL 使用反斜线（\）作为转义字符，如下 SQL 语句所示：

```
# 选出所有名字以下画线开头的学生
select * from student_table
where student_name like '\_%';
```

标准 SQL 语句并没有提供反斜线（\）的转义字符，而是使用 escape 关键字显式进行转义。例如，为了实现上面功能需要使用如下 SQL 语句：

```
#在标准的 SQL 中选出所有名字以下画线开头的学生
select * from student_table
where student_name like '\_%' escape '\';
```

is null 用于判断某些值是否为空，判断是否为空不要用=null 来判断，因为 SQL 中 null=null 返回 null。如下 SQL 语句将选择出 student_table 表中 student_name 为 null 的所有记录。

```
select * from student_table
where student_name is null;
```

如果 where 子句后有多个条件需要组合，SQL 提供了 and 和 or 逻辑运算符来组合两个条件，并提供了 not 来对逻辑表达式求否。如下 SQL 语句将选出学生名字为 2 个字符，且 student_id 大于 3 的所有记录。

```
select * from student_table
# 使用 and 来组合多个条件
where student_name like '__' and student_id > 3;
```

下面的 SQL 语句将选出 student_table 表中姓名不以下画线开头的所有记录。

```
select * from student_table
# 使用 not 对 where 条件取否
where not student_name like '\_%';
```

当使用比较运算符、逻辑运算符来连接表达式时，必须注意这些运算符的优先级。SQL 中比较运算符、逻辑运算符的优先级如表 13.4 所示。

表 13.4　SQL 中比较运算符、逻辑运算符的优先级

运　算　符	优先级（优先级小的优先）
所有的比较运算符	1
not	2
and	3
or	4

如果 SQL 代码需要改变优先级的默认顺序，则可以使用括号，括号的优先级比所有的运算符高。如下 SQL 语句使用括号来改变逻辑运算符的优先级。

```
select * from student_table
# 使用括号强制先计算 or 运算
where (student_id > 3 or student_name > '张')
    and java_teacher > 1;
```

执行查询后的查询结果默认按插入顺序排列；如果需要查询结果按某列值的大小进行排序，则可以使用 order by 子句。order by 子句的语法格式如下：

```
order by column_name1 [desc] , column_name2 …
```

进行排序时默认按升序排列，如果强制按降序排列，则需要在列后使用 desc 关键字（与之对应的是 asc 关键字，用不用该关键字的效果完全一样，因为默认是按升序排列）。

上面语法中设定排序列时可采用列名、列序号和列别名。如下 SQL 语句选出 student_table 表中的所有记录，选出后按 java_teacher 列的升序排列。

```
select * from student_table
order by java_teacher;
```

如果需要按多列排序，则每列的 asc、desc 必须单独设定。如果指定了多个排序列，则第一个排序列是首要排序列，只有当第一列中存在多个相同的值时，第二个排序列才会起作用。如下 SQL 语句先按 java_teacher 列的降序排列，当 java_teacher 列的值相同时按 student_name 列的升序排列。

```
select * from student_table
order by java_teacher desc , student_name;
```

13.2.10　数据库函数

正如前面看到的连接字符串使用的 concat 函数，每个数据库都会在标准的 SQL 基础上扩展一些函数，这些函数用于进行数据处理或复杂计算，它们通过对一组数据进行计算，得到最终需要的输出结果。函数一般都会有一个或者多个输入，这些输入被称为函数的参数，函数内部会对这些参数进行判断和计算，最终只有一个值作为返回值。函数可以出现在 SQL 语句的各个位置，比较常用的位置是 select 之后和 where 子句中。

根据函数对多行数据的处理方式，函数被分为单行函数和多行函数，单行函数对每行输入值单独计算，每行得到一个计算结果返回给用户；多行函数对多行输入值整体计算，最后只会得到一个结果。单

行函数和多行函数的示意图如图 13.13 所示。

SQL 中的函数和 Java 语言中的方法有点相似，但 SQL 中的函数是独立的程序单元，也就是说，调用函数时无须使用任何类、对象作为调用者，而是直接执行函数。执行函数的语法如下：

```
function_name(arg1, arg2 ...)
```

多行函数也称为聚集函数、分组函数，主要用于完成一些统计功能，在大部分数据库中基本相同。但不同数据库中的单行函数差别非常大，MySQL 中的单行函数具有如下特征。

- 单行函数的参数可以是变量、常量或数据列。单行函数可以接收多个参数，但只返回一个值。
- 单行函数会对每行单独起作用，每行（可能包含多个参数）返回一个结果。
- 使用单行函数可以改变参数的数据类型。单行函数支持嵌套使用，即内层函数的返回值是外层函数的参数。

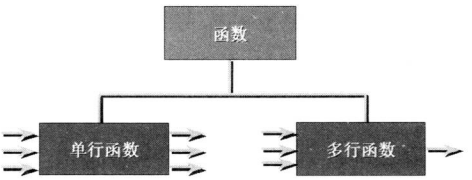


图 13.13　单行函数和多行函数的示意图

MySQL 的单行函数分类如图 13.14 所示。

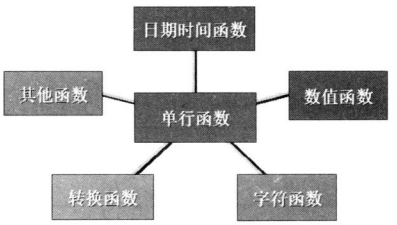


图 13.14　MySQL 的单行函数分类

MySQL 数据库的数据类型大致分为数值型、字符型和日期时间型，所以MySQL分别提供了对应的函数。转换函数主要负责完成类型转换，其他函数又大致分为如下几类。

- 位函数
- 流程控制函数
- 加密解密函数
- 信息函数

每个数据库都包含了大量的单行函数，这些函数的用法也存在一些差异，但有一点是相同的——每个数据库都会为一些常用的计算功能提供相应的函数，这些函数的函数名可能不同，用法可能有差异，但所有数据库提供的函数库所能完成的功能大致相似，读者可以参考各数据库系统的参考文档来学习这些函数的用法。下面通过一些例子来介绍 MySQL 单行函数的用法。

```
# 选出 teacher_table 表中 teacher_name 列的字符长度
select char_length(teacher_name)
from teacher_table;
# 计算 teacher_name 列的字符长度的 sin 值
select sin(char_length(teacher_name))
from teacher_table;
# 计算 1.57 的 sin 值，约等于 1
select sin(1.57);
# 为指定日期添加一定的时间
# 在这种用法下 interval 是关键字，需要一个数值，还需要一个单位
SELECT DATE_ADD('1998-01-02', interval 2 MONTH);
# 这种用法更简单
select ADDDATE('1998-01-02',3);
# 获取当前日期
select CURDATE();
# 获取当前时间
select curtime();
# 下面的 MD5 是 MD5 加密函数
select MD5('testing');
```

MySQL 提供了如下几个处理 null 的函数。

- ifnull(expr1,expr2)：如果 expr1 为 null，则返回 expr2，否则返回 expr1。
- nullif(expr1,expr2)：如果 erpr1 和 expr2 相等，则返回 null，否则返回 expr1。
- if(expr1,expr2,expr3)：有点类似于?:三目运算符，如果 expr1 为 true，不等于 0，且不等于 null，

则返回 expr2，否则返回 expr3。

- isnull(expr1)：判断 expr1 是否为 null，如果为 null 则返回 true，否则返回 false。

```
# 如果 student_name 列为 null，则返回'没有名字'
select ifnull(student_name,'没有名字')
from student_table;
# 如果 student_name 列等于'张三'，则返回 null
select nullif(student_name,'张三')
from student_table;
# 如果 student_name 列为 null，则返回'没有名字'，否则返回'有名字'
select if(isnull(student_name),'没有名字', '有名字')
from student_table;
```

MySQL 还提供了一个 case 函数，该函数是一个流程控制函数。case 函数有两个用法，case 函数第一个用法的语法格式如下：

```
case value
when compare_value1 then result1
when compare_value2 then result2
...
else result
end
```

case 函数用 value 和后面的 compare_value1、compare_value2、…依次进行比较，如果 value 和指定的 compare_value1 相等，则返回对应的 result1，否则返回 else 后的 result。例如如下 SQL 语句：

```
# 如果 java_teacher 为 1，则返回'Java 老师'，为 2 返回'Ruby 老师'，否则返回'其他老师'
select student_name , case java_teacher
when 1 then 'Java 老师'
when 2 then 'Ruby 老师'
else '其他老师'
end
from student_table;
```

case 函数第二个用法的语法格式如下：

中華工匠

```
case
when condition1 then result1
when condition2 then result2
...
else result
end
```

在第二个用法中，condition1、condition2 都是一个返回 boolean 值的条件表达式，因此这种用法更加灵活。例如如下 SQL 语句：

```
# id 小于 3 的为初级班，3~6 的为中级班，其他的为高级班
select student_name,case
when student_id<=3 then '初级班'
when student_id<=6 then '中级班'
else '高级班'
end
from student_table;
```

虽然此处介绍了一些 MySQL 常用函数的简单用法，但通常不推荐在 Java 程序中使用特定数据库的函数，因为这将导致程序代码与特定数据库耦合；如果需要把该程序移植到其他数据库系统上时，可能需要打开源程序，重新修改 SQL 语句。

13.2.11 分组和组函数

组函数也就是前面提到的多行函数，组函数将一组记录作为整体计算，每组记录返回一个结果，而不是每条记录返回一个结果。常用的组函数有如下 5 个。

- avg([distinct|all]expr)：计算多行 expr 的平均值，其中，expr 可以是变量、常量或数据列，但其数据类型必须是数值型。还可以在变量、列前使用 distinct 或 all 关键字，如果使用 distinct，则表明不计算重复值；all 用和不用的效果完全一样，表明需要计算重复值。

- count({ *|[distinct|all]expr})：计算多行 expr 的总条数，其中，expr 可以是变量、常量或数据列，其数据类型可以是任意类型；用星号（*）表示统计该表内的记录行数；distinct 表示不计算重复值。
- max(expr)：计算多行 expr 的最大值，其中 expr 可以是变量、常量或数据列，其数据类型可以是任意类型。
- min(expr)：计算多行 expr 的最小值，其中 expr 可以是变量、常量或数据列，其数据类型可以是任意类型。
- sum([distinct|all]expr)：计算多行 expr 的总和，其中，expr 可以是变量、常量或数据列，但其数据类型必须是数值型；distinct 表示不计算重复值。

```
# 计算 student_table 表中的记录条数
select count(*)
from student_table;
# 计算 java_teacher 列总共有多少个值
select count(distinct java_teacher)
from student_table;
# 统计所有 student_id 的总和
select sum(student_id)
from student_table;
# 计算的结果是 20 * 记录的行数
select sum(20)
from student_table;
# 选出 student_table 表中 student_id 最大的值
select max(student_id)
from student_table;
# 选出 teacher_table 表中 teacher_id 最小的值
select min(teacher_id)
from teacher_table;
# 因为 sum 里的 expr 是常量 34，所以每行的值都相同
# 使用 distinct 强制不计算重复值，所以下面计算结果为 34
select sum(distinct 34)
from student_table;
# 使用 count 统计记录行数时，null 不会被计算在内
select count(student_name)
from student_table;
```

对于可能出现 null 的列，可以使用 ifnull 函数来处理该列。

```
# 计算 java_teacher 列所有记录的平均值
select avg(ifnull(java_teacher , 0))
from student_table;
```

值得指出的是，distinct 和*不同时使用，如下 SQL 语句有错误。

```
select count(distinct *)
from student_table;
```

在默认情况下，组函数会把所有记录当成一组，为了对记录进行显式分组，可以在 select 语句后使用 group by 子句，group by 子句后通常跟一个或多个列名，表明查询结果根据一列或多列进行分组——当一列或多列组合的值完全相同时，系统会把这些记录当成一组。如下 SQL 语句所示：

```
# count(*)将会对每组得到一个结果
select count(*)
from student_table
# 将 java_teacher 列值相同的记录当成一组
group by java_teacher;
```

如果对多列进行分组，则要求多列的值完全相同才会被当成一组。如下 SQL 语句所示：

```
select count(*)
from student_table
# 当 java_teacher、student_name 两列的值完全相同时才会被当成一组
group by java_teacher , student_name;
```

对于很多数据库而言，分组计算时有严格的规则——如果查询列表中使用了组函数，或者 select 语句中使用了 group by 分组子句，则要求出现在 select 列表中的字段，要么使用组函数包起来，要么必须

出现在 group by 子句中。这条规则很容易理解，因为一旦使用了组函数或使用了 group by 子句，都将导致多条记录只有一条输出，系统无法确定输出多条记录中的哪一条记录。

对于 MySQL 来说，并没有上面的规则要求，如果某个数据列既没有出现在 group by 之后，也没有使用组函数包起来，则 MySQL 会输出该列的第一条记录的值。图 13.15 显示了 MySQL 的处理结果。

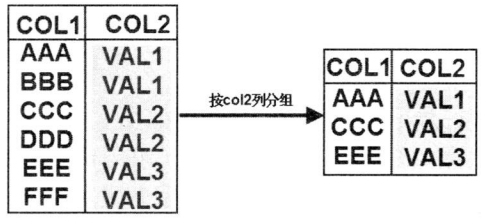


图 13.15　MySQL 处理不在 group by、组函数中的列

如果需要对分组进行过滤，则应该使用 having 子句，having 子句后面也是一个条件表达式，只有满足该条件表达式的分组才会被选出来。having 子句和 where 子句非常容易混淆，它们都有过滤功能，但它们有如下区别。

- 不能在 where 子句中过滤组，where 子句仅用于过滤行。过滤组必须使用 having 子句。
- 不能在 where 子句中使用组函数，having 子句才可使用组函数。

如下 SQL 语句所示：

```
select *
from student_table
group by java_teacher
# 对组进行过滤
having count(*) > 2;
```

13.2.12　多表连接查询

很多时候，需要选择的数据并不是来自一个表，而是来自多个数据表，这就需要使用多表连接查询。例如，对于上面的 student_table 和 teacher_table 两个数据表，如果希望查询出所有学生以及他的老师名字，这就需要从两个表中取数据。

多表连接查询有两种规范，较早的 SQL 92 规范支持如下几种多表连接查询。

- 等值连接。
- 非等值连接。
- 外连接。
- 广义笛卡儿积。

SQL 99 规范提供了可读性更好的多表连接语法，并提供了更多类型的连接查询。SQL 99 支持如下几种多表连接查询。

- 交叉连接。
- 自然连接。
- 使用 using 子句的连接。
- 使用 on 子句的连接。
- 全外连接或者左、右外连接。

1. SQL 92 的连接查询

SQL 92 的多表连接语法比较简洁，这种语法把多个数据表都放在 from 之后，多个表之间以逗号隔开；连接条件放在 where 之后，与查询条件之间用 and 逻辑运算符连接。如果连接条件要求两列值相等，则称为等值连接，否则称为非等值连接；如果没有任何连接条件，则称为广义笛卡儿积。SQL 92 中多表连接查询的语法格式如下：

```
select column1 , column2 ...
from table1, table2 ...
[where join_condition]
```

多表连接查询中可能出现两个或多个数据列具有相同的列名，则需要在这些同名列之间使用表名前缀或表别名前缀作为限制，避免系统混淆。

实际上，所有的列都可以增加表名前缀或表别名前缀。只是进行单表查询时，绝不可能出现同名列，所以系统不可能混淆，因此通常省略表名前缀。

如下 SQL 语句查询出所有学生的资料以及对应的老师姓名。

```
select s.* , teacher_name
# 指定多个数据表，并指定表别名
from student_table s , teacher_table t
# 使用 where 指定连接条件
where s.java_teacher = t.teacher_id;
```

执行上面查询语句，将看到如图 13.16 所示的结果。

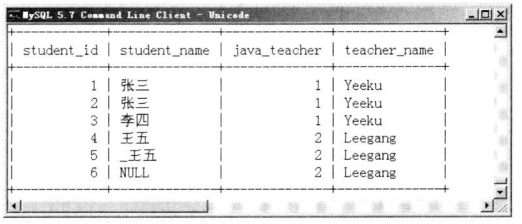


图 13.16 等值连接查询的结果

上面的查询结果正好满足要求，可以看到每个学生以及他对应的老师的名字。实际上，多表查询的过程可理解成一个嵌套循环，这个嵌套循环的伪码如下：

```
// 依次遍历 teacher_table 表中的每条记录
for t in teacher_table
{
    // 遍历 student_table 表中的每条记录
    for s in student_table
    {
        // 当满足连接条件时，输出两个表连接后的结果
        if (s.java_teacher = t.teacher_id)
            output s + t
    }
}
```

理解了上面的伪码之后，接下来即可很轻易地理解多表连接查询的运行机制。如果求广义笛卡儿积，则 where 子句后没有任何连接条件，相当于没有上面的 if 语句，广义笛卡儿积的结果会有 $n \times m$ 条记录。只要把 where 后的连接条件去掉，就可以得到广义笛卡儿积，如下 SQL 语句所示：

```
# 不使用连接条件，得到广义笛卡儿积
select s.* , teacher_name
# 指定多个数据表，并指定表别名
from student_table s , teacher_table t;
```

与此类似的是，非等值连接的执行结果可以使用上面的嵌套循环来计算，如下 SQL 语句所示：

```
select s.* , teacher_name
# 指定多个数据表，并指定表别名
from student_table s , teacher_table t
# 使用 where 指定连接条件，非等值连接
where s.java_teacher > t.teacher_id;
```

上面 SQL 语句的执行结果相当于 if 条件换成了 s.java_teacher > t.teacher_id。

如果还需要对记录进行过滤，则将过滤条件和连接条件使用 and 连接起来，如下 SQL 语句所示：

```
select s.* , teacher_name
# 指定多个数据表，并指定表别名
from student_table s , teacher_table t
# 使用 where 指定连接条件，并指定 student_name 列不能为 null
where s.java_teacher = t.teacher_id and student_name is not null;
```

虽然 MySQL 不支持 SQL 92 中的左外连接、右外连接，但本书还是有必要了解一下 SQL 92 中的左外连接和右外连接。SQL 92 中的外连接就是在连接条件的列名后增加括号包起来的外连接符（+或*，不同的数据库有一定的区别），当外连接符出现在左边时称为左外连接，出现在右边时则称为右外连接。如下 SQL 语句所示：

```
select s.* , teacher_name
from student_table s , teacher_table t
# 右外连接
where s.java_teacher = t.teacher_id(*);
```

外连接就是在外连接符所在的表中增加一个“万能行”，这行记录的所有数据都是 null，而且该行可以与另一个表中所有不满足条件的记录进行匹配，通过这种方式就可以把另一个表中的所有记录选出来，不管这些记录是否满足连接条件。

除此之外，还有一种自连接，正如前面介绍外键约束时提到的自关联，如果同一个表中的不同记录

之间存在主、外键约束关联，例如把员工、经理保存在同一个表里，则需要使用自连接查询。

自连接只是连接的一种用法，并不是一种连接类型，不管是 SQL 92 还是 SQL 99 都可以使用自连接查询。自连接的本质就是把一个表当成两个表来用。

下面的 SQL 语句建立了一个自关联的数据表，并向表中插入了 4 条数据。

```
create table emp_table
(
    emp_id int auto_increment primary key,
    emp_name varchar(255),
    manager_id int,
    foreign key(manager_id) references emp_table(emp_id)
);
insert into emp_table
values(null , '唐僧' , null),
(null , '孙悟空' , 1),
(null , '猪八戒' , 1),
(null , '沙僧' , 1);
```

如果需要查询该数据表中的所有员工名，以及每个员工对应的经理名，则必须使用自连接查询。所谓自连接就是把一个表当成两个表来用，这就需要为一个表起两个别名，而且查询中用的所有数据列都要加表别名前缀，因为两个表的数据列完全一样。下面的自连接查询可以查询出所有的员工名，以及对应的经理名。

```
select emp.emp_id,emp.emp_name 员工名,mgr.emp_name 经理名
from emp_table emp,emp_table mgr
where emp.manager_id = mgr.emp_id;
```

2. SQL 99 的连接查询

SQL 99 的连接查询与 SQL 92 的连接查询原理基本相似，不同的是 SQL 99 连接查询的可读性更强——查询用的多个数据表显式使用 xxx join 连接，而不是直接依次排列在 from 之后，from 后只需要放一个数据表；连接条件不再放在 where 之后，而是提供了专门的连接条件子句。

- **交叉连接（cross join）**：交叉连接效果就是 SQL 92 中的广义笛卡儿积，所以交叉连接无须任何连接条件，如下 SQL 语句所示：

```
select s.* , teacher_name
# SQL 99 多表连接查询的 from 后只有一个表名
from student_table s
# cross join 交叉连接，相当于广义笛卡儿积
cross join teacher_table t;
```

- **自然连接（natural join）**：自然连接表面上看起来也无须指定连接条件，但自然连接是有连接条件的，自然连接会以两个表中的同名列作为连接条件；如果两个表中没有同名列，则自然连接与交叉连接效果完全一样——因为没有连接条件。如下 SQL 语句所示：

```
select s.* , teacher_name
# SQL 99 多表连接查询的 from 后只有一个表名
from student_table s
# natural join 自然连接使用两个表中的同名列作为连接条件
natural join teacher_table t;
```

- **using 子句连接**：using 子句可以指定一列或多列，用于显式指定两个表中的同名列作为连接条件。假设两个表中有超过一列的同名列，如果使用 natural join，则会把所有的同名列当成连接条件；使用 using 子句，就可显式指定使用哪些同名列作为连接条件。如下 SQL 语句所示：

```
select s.* , teacher_name
# SQL 99 多表连接查询的 from 后只有一个表名
from student_table s
# join 连接另一个表
```

```
join teacher_table t
using(teacher_id);
```

运行上面语句将出现一个错误，因为student_table表中并不存在名为teacher_id的列。也就是说，如果使用using子句来指定连接条件，则两个表中必须有同名列，否则就会出现错误。

- **on子句连接**：这是最常用的连接方式，SQL 99语法的连接条件放在on子句中指定，而且每个on子句只指定一个连接条件。这意味着：如果需要进行*N*表连接，则需要有*N*-1个join...on对。如下SQL语句所示：

```
select s.* , teacher_name
# SQL 99多表连接查询的from后只有一个表名
from student_table s
# join连接另一个表
join teacher_table t
# 使用on来指定连接条件
on s.java_teacher = t.teacher_id;
```

使用 on子句的连接完全可以代替SQL 92中的等值连接、非等值连接，因为on子句的连接条件除等值条件之外，也可以是非等值条件。如下SQL语句就是SQL 99中的非等值连接。

```
select s.* , teacher_name
# SQL 99多表连接查询的from后只有一个表名
from student_table s
# join连接另一个表
join teacher_table t
# 使用on来指定连接条件：非等值连接
on s.java_teacher > t.teacher_id;
```

- **左、右、全外连接**：这三种外连接分别使用left [outer] join、right [outer] join和full [outer] join，这三种外连接的连接条件一样通过 on 子句来指定，既可以是等值连接条件，也可以是非等值连接条件。

下面使用右外连接，连接条件是非等值连接。

```
select s.* , teacher_name
# SQL 99多表连接查询的from后只有一个表名
from student_table s
# right join右外连接另一个表
right join teacher_table t
# 使用on来指定连接条件，使用非等值连接
on s.java_teacher < t.teacher_id;
```

下面使用左外连接，连接条件是非等值连接。

```
select s.* , teacher_name
# SQL 99多表连接查询的from后只有一个表名
from student_table s
# left join左外连接另一个表
left join teacher_table t
# 使用on来指定连接条件，使用非等值连接
on s.java_teacher > t.teacher_id;
```

运行上面两条外连接语句并查看它们的运行结果，不难发现SQL 99外连接与SQL 92外连接恰好相反，SQL99左外连接将会把左边表中所有不满足连接条件的记录全部列出；SQL 99右外连接将会把右边表中所有不满足连接条件的记录全部列出。

下面的SQL语句使用全外连接，连接条件是等值连接。

```
select s.* , teacher_name
# SQL 99多表连接查询的from后只有一个表名
from student_table s
# full join全外连接另一个表
full join teacher_table t
# 使用on来指定连接条件，使用等值连接
on s.java_teacher = t.teacher_id;
```

SQL 99的全外连接将会把两个表中所有不满足连接条件的记录全部列出。

13.2.13 子查询

子查询就是指在查询语句中嵌套另一个查询，子查询可以支持多层嵌套。对于一个普通的查询语句而言，子查询可以出现在两个位置。

- 出现在 from 语句后当成数据表，这种用法也被称为行内视图，因为该子查询的实质就是一个临时视图。
- 出现在 where 条件后作为过滤条件的值。

使用子查询时要注意如下几点。

- 子查询要用括号括起来。
- 把子查询当成数据表时（出现在 from 之后），可以为该子查询起别名，尤其是作为前缀来限定数据列时，必须给子查询起别名。
- 把子查询当成过滤条件时，将子查询放在比较运算符的右边，这样可以增强查询的可读性。
- 把子查询当成过滤条件时，单行子查询使用单行运算符，多行子查询使用多行运算符。

对于把子查询当成数据表是完全把子查询当做数据表来用，只是把之前的表名变成子查询（也可为子查询起别名），其他部分与普通查询没有任何区别。下面的 SQL 语句示范了把子查询当成数据表的用法。

```
select *
# 把子查询当成数据表
from (select * from student_table) t
where t.java_teacher > 1;
```

把子查询当成数据表的用法更准确地说是当成视图，可以把上面的 SQL 语句理解成在执行查询时创建了一个临时视图，该视图名为 t，所以这种临时创建的视图也被称为行内视图。理解了这种子查询的实质后，不难知道这种子查询可以完全代替查询语句中的数据表，包括在多表连接查询中使用这种子查询。

还有一种情形：把子查询当成 where 条件中的值，如果子查询返回单行、单列值，则被当成一个标量值使用，也就可以使用单行记录比较运算符。例如如下 SQL 语句：

```
select *
from student_table
where java_teacher >
# 返回单行、单列的子查询可以当成标量值使用
(select teacher_id
from teacher_table
 where teacher_name='Yeeku');
```

上面查询语句中的子查询（粗体字部分）将返回一个单行、单列值（该值就是 1），如果把上面查询语句的括号部分换成 1，那么这条语句就再简单不过了——实际上，这就是这种子查询的实质，单行、单列子查询的返回值被当成标量值处理。

如果子查询返回多个值，则需要使用 in、any 和 all 等关键字，in 可以单独使用，与前面介绍比较运算符时所讲的 in 完全一样，此时可以把子查询返回的多个值当成一个值列表。如下 SQL 语句所示：

```
select *
from student_table
where student_id in
(select teacher_id
 from teacher_table);
```

上面查询语句中的子查询（粗体字部分）将返回多个值，这多个值将被当成一个值列表，只要 student_id 与该值列表中的任意一个值相等，就可以选出这条记录。

any 和 all 可以与>、>=、<、<=、<>、=等运算符结合使用，与 any 结合使用分别表示大于、大于等于、小于、小于等于、不等于、等于其中任意一个值；与 all 结合使用分别表示大于、大于等于、小

于、小于等于、不等于、等于全部值。从上面介绍中可以看出，=any 的作用与 in 的作用相同。如下 SQL 语句使用=any 来代替上面的 in。

```
select *
from student_table
where student_id =
any(select teacher_id
 from teacher_table);
```

<ANY 只要小于值列表中的最大值即可，>ANY 只要大于值列表中的最小值即可。<All 要求小于值列表中的最小值，>All 要求大于值列表中的最大值。

下面的 SQL 语句选出 student_table 表中 student_id 大于 teacher_table 表中所有 teacher_id 的记录。

```
select *
from student_table
where student_id >
all(select teacher_id
 from teacher_table);
```

还有一种子查询可以返回多行、多列，此时 where 子句中应该有对应的数据列，并使用圆括号将多个数据列组合起来。如下 SQL 语句所示：

```
select *
from student_table
where (student_id,student_name)
=any(select teacher_id, teacher_name
 from teacher_table);
```

13.2.14 集合运算

select 语句查询的结果是一个包含多条数据的结果集，类似于数学里的集合，可以进行交（intersect）、并（union）和差（minus）运算，select 查询得到的结果集也可能需要进行这三种运算。

为了对两个结果集进行集合运算，这两个结果集必须满足如下条件。

- 两个结果集所包含的数据列的数量必须相等。
- 两个结果集所包含的数据列的数据类型也必须一一对应。

1. union 运算

union 运算的语法格式如下：

```
select 语句 union select 语句
```

下面的 SQL 语句查询出所有教师的信息和主键小于 4 的学生信息。

```
# 查询结果集包含两列，第一列为 int 类型，第二列为 varchar 类型
select * from teacher_table
union
# 这个结果集的数据列必须与前一个结果集的数据列一一对应
select student_id , student_name from student_table;
```

2. minus 运算

minus 运算的语法格式如下：

```
select 语句 minus select 语句
```

上面的语法格式十分简单，不过很遗憾，MySQL 并不支持使用 minus 运算符，因此只能借助于子查询来“曲线”实现上面的 minus 运算。

假如想从所有学生记录中“减去”与老师记录的 ID 相同、姓名相同的记录，则可进行如下的 minus 运算：

```
select student_id , student_name from student_table
minus
# 两个结果集的数据列的数量相等，数据类型一一对应，可以进行 minus 运算
select teacher_id , teacher_name from teacher_table;
```

不过，MySQL 并不支持这种运算。但可以通过如下子查询来实现上面运算。

```
select student_id , student_name from student_table
where(student_id , student_name)
not in
(select teacher_id , teacher_name from teacher_table);
```

3. intersect 运算

intersect 运算的语法格式如下：

```
select 语句 intersect select 语句
```

上面的语法格式十分简单，不过很遗憾，MySQL 并不支持使用 intersect 运算符，因此只能借助于多表连接查询来“曲线”实现上面的 intersect 运算。

假如想找出学生记录中与老师记录中的 ID 相同、姓名相同的记录，则可进行如下的 intersect 运算：

```
select student_id , student_name from student_table
intersect
# 两个结果集的数据列的数量相等，数据类型一一对应，可以进行 intersect 运算
select teacher_id , teacher_name from teacher_table;
```

不过，MySQL 并不支持这种运算。但可以通过如下多表连接查询来实现上面运算。

```
select student_id , student_name from student_table
join
teacher_table
on(student_id=teacher_id and student_name=teacher_name);
```

需要指出的是，如果进行 intersect 运算的两个 select 子句中都包括了 where 条件，那么将 intersect 运算改写成多表连接查询后还需要将两个 where 条件进行 and 运算。假如有如下 intersect 运算的 SQL 语句：

```
select student_id , student_name from student_table where student_id<4
intersect
# 两个结果集的数据列的数量相等，数据类型一一对应，可以进行 intersect 运算
select teacher_id , teacher_name from teacher_table where teacher_name like '李%';
```

上面语句改写如下：

```
select student_id , student_name from student_table
join
teacher_table
on(student_id=teacher_id and student_name=teacher_name)
where student_id<4 and teacher_name like '李%';
```

13.3 JDBC 的典型用法

掌握了标准的 SQL 命令语法之后，就可以开始使用 JDBC 开发数据库应用了。

13.3.1 JDBC 4.2 常用接口和类简介

Java 8 支持 JDBC 4.2 标准，JDBC 4.2 在原有 JDBC 标准上增加了一些新特性。下面介绍这些 JDBC API 时会提到 Java 8 新增的功能。

- DriverManager：用于管理 JDBC 驱动的服务类。程序中使用该类的主要功能是获取 Connection 对象，该类包含如下方法。
  - public static synchronized Connection getConnection(String url, String user, String pass) throws SQLException：该方法获得 url 对应数据库的连接。
- Connection：代表数据库连接对象，每个 Connection 代表一个物理连接会话。要想访问数据库，必须先获得数据库连接。该接口的常用方法如下。
  - Statement createStatement() throws SQLExcetpion：该方法返回一个 Statement 对象。

- PreparedStatement prepareStatement(String sql) throws SQLExcetpion：该方法返回预编译的Statement对象，即将SQL语句提交到数据库进行预编译。
- CallableStatement prepareCall(String sql) throws SQLExcetpion：该方法返回CallableStatement对象，该对象用于调用存储过程。

上面三个方法都返回用于执行SQL语句的Statement对象，PreparedStatement、CallableStatement是Statement的子类，只有获得了Statement之后才可执行SQL语句。

除此之外，Connection还有如下几个用于控制事务的方法。

- Savepoint setSavepoint()：创建一个保存点。
- Savepoint setSavepoint(String name)：以指定名字来创建一个保存点。
- void setTransactionIsolation(int level)：设置事务的隔离级别。
- void rollback()：回滚事务。
- void rollback(Savepoint savepoint)：将事务回滚到指定的保存点。
- void setAutoCommit(boolean autoCommit)：关闭自动提交，打开事务。
- void commit()：提交事务。

Java 7为Connection新增了setSchema(String schema)、getSchema()两个方法，这两个方法用于控制该Connection访问的数据库Schema。Java 7还为Connection新增了setNetworkTimeout(Executor executor, int milliseconds)、getNetworkTimeout()两个方法来控制数据库连接的超时行为。

- Statement：用于执行SQL语句的工具接口。该对象既可用于执行DDL、DCL语句，也可用于执行DML语句，还可用于执行SQL查询。当执行SQL查询时，返回查询到的结果集。它的常用方法如下。
  - ResultSet executeQuery(String sql)throws SQLException：该方法用于执行查询语句，并返回查询结果对应的ResultSet对象。该方法只能用于执行查询语句。
  - int executeUpdate(String sql)throws SQLExcetion：该方法用于执行DML语句，并返回受影响的行数；该方法也可用于执行DDL语句，执行DDL语句将返回0。
  - boolean execute(String sql)throws SQLException：该方法可执行任何SQL语句。如果执行后第一个结果为ResultSet对象，则返回true；如果执行后第一个结果为受影响的行数或没有任何结果，则返回false。

Java 7为Statement新增了closeOnCompletion()方法，如果Statement执行了该方法，则当所有依赖于该Statement的ResultSet关闭时，该Statement会自动关闭。Java 7还为Statement提供了一个isCloseOnCompletion()方法，该方法用于判断该Statement是否打开了“closeOnCompletion”。

Java 8为Statement新增了多个重载的executeLargeUpdate()方法，这些方法相当于增强版的executeUpdate()方法，返回值类型为long——也就是说，当DML语句影响的记录条数超过Integer.MAX_VALUE时，就应该使用executeLargeUpdate()方法。

提示：

考虑到目前应用程序所处理的数据量越来越大，使用executeLargeUpdate()方法具有更好的适应性。但遗憾的是，目前最新的MySQL驱动暂不支持该方法。

- PreparedStatement：预编译的Statement对象。PreparedStatement是Statement的子接口，它允许数据库预编译SQL语句（这些SQL语句通常带有参数），以后每次只改变SQL命令的参数，避免数据库每次都需要编译SQL语句，因此性能更好。相对于Statement而言，使用PreparedStatement执行SQL语句时，无须再传入SQL语句，只要为预编译的SQL语句传入参数值即可。所以它比Statement多了如下方法。
  - void setXxx(int parameterIndex,Xxx value)：该方法根据传入参数值的类型不同，需要使用不同的方法。传入的值根据索引传给SQL语句中指定位置的参数。

注意：

PreparedStatement 同样有 executeUpdate()、executeQuery()和 execute()三个方法，只是这三个方法无须接收 SQL 字符串，因为 PreparedStatement 对象已经预编译了 SQL 命令，只要为这些命令传入参数即可。Java 8 还为 PreparedStatement 增加了不带参数的 executeLargeUpdate()方法——执行 DML 语句影响的记录条数可能超过 Integer.MAX_VALUE 时，就应该使用 executeLargeUpdate()方法。

- **ResultSet**：结果集对象。该对象包含访问查询结果的方法，ResultSet 可以通过列索引或列名获得列数据。它包含了如下常用方法来移动记录指针。
  - void close()：释放 ResultSet 对象。
  - boolean absolute(int row)：将结果集的记录指针移动到第 row 行，如果 row 是负数，则移动到倒数第 row 行。如果移动后的记录指针指向一条有效记录，则该方法返回 true。
  - void beforeFirst()：将 ResultSet 的记录指针定位到首行之前，这是 ResultSet 结果集记录指针的初始状态——记录指针的起始位置位于第一行之前。
  - boolean first()：将 ResultSet 的记录指针定位到首行。如果移动后的记录指针指向一条有效记录，则该方法返回 true。
  - boolean previous()：将 ResultSet 的记录指针定位到上一行。如果移动后的记录指针指向一条有效记录，则该方法返回 true。
  - boolean next()：将 ResultSet 的记录指针定位到下一行，如果移动后的记录指针指向一条有效记录，则该方法返回 true。
  - boolean last()：将 ResultSet 的记录指针定位到最后一行，如果移动后的记录指针指向一条有效记录，则该方法返回 true。
  - void afterLast()：将 ResultSet 的记录指针定位到最后一行之后。

注意：

在 JDK 1.4 以前，采用默认方法创建的 Statement 所查询得到的 ResultSet 不支持 absolute()、previous()等移动记录指针的方法，它只支持 next()这个移动记录指针的方法，即 ResultSet 的记录指针只能向下移动，而且每次只能移动一格。从 Java 5.0 以后就避免了这个问题，程序采用默认方法创建的 Statement 所查询得到的 ResultSet 也支持 absolute()、previous()等方法。

当把记录指针移动到指定行之后，ResultSet 可通过 getXxx(int columnIndex)或 getXxx(String columnLabel)方法来获取当前行、指定列的值，前者根据列索引获取值，后者根据列名获取值。Java 7 新增了<T> T getObject(int columnIndex, Class<T> type)和<T> T getObject(String columnLabel, Class<T> type)两个泛型方法，它们可以获取任意类型的值。

13.3.2 JDBC 编程步骤

大致了解了 JDBC API 的相关接口和类之后，下面就可以进行 JDBC 编程了，JDBC 编程大致按如下步骤进行。

① 加载数据库驱动。通常使用 Class 类的 forName()静态方法来加载驱动。例如如下代码：

```
// 加载驱动
Class.forName(driverClass)
```

提示：

最新的 JDBC 驱动已经可以通过 SPI 自动注册驱动类了，在 JDBC 驱动 JAR 包的 META-INF\services 路径下会包含一个 java.sql.Driver 文件，该文件指定了 JDBC 驱动类。因此，如果使用这种最新的驱动 JAR 包，第 1 步其实可以省略。

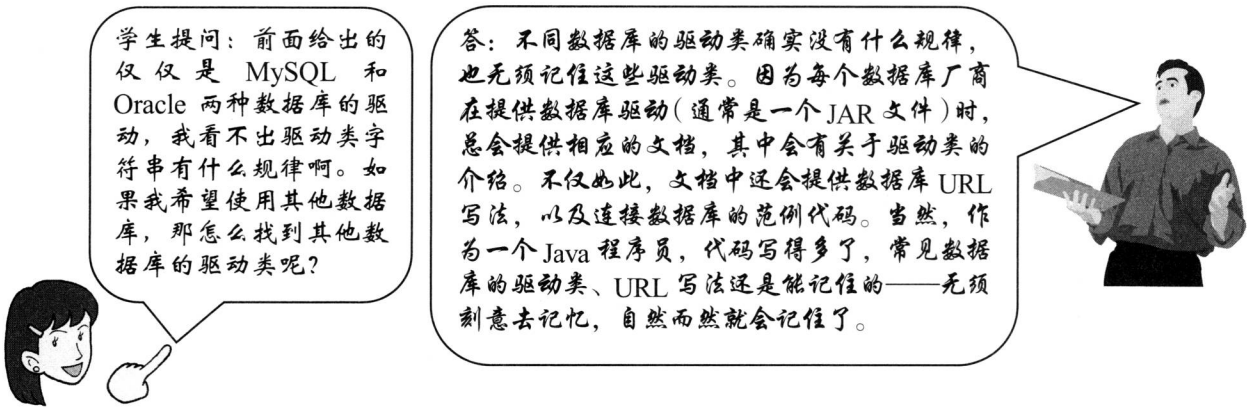
上面代码中的 driverClass 就是数据库驱动类所对应的字符串。例如，加载 MySQL 的驱动采用如下代码：

```
// 加载 MySQL 的驱动
Class.forName("com.mysql.jdbc.Driver");
```

而加载 Oracle 的驱动则采用如下代码：

```
// 加载 Oracle 的驱动
Class.forName("oracle.jdbc.driver.OracleDriver");
```

从上面代码中可以看出，加载驱动时并不是真正使用数据库的驱动类，只是使用数据库驱动类名的字符串而已。



② 通过 DriverManager 获取数据库连接。DriverManager 提供了如下方法：

```
// 获取数据库连接
DriverManager.getConnection(String url,String user,String pass);
```

当使用 DriverManager 获取数据库连接时，通常需要传入三个参数：数据库 URL、登录数据库的用户名和密码。这三个参数中用户名和密码通常由 DBA（数据库管理员）分配，而且该用户还应该具有相应的权限，才可以执行相应的 SQL 语句。

数据库 URL 通常遵循如下写法：

```
jdbc:subprotocol:other stuff
```

上面 URL 写法中的 jdbc 是固定的，而 subprotocol 指定连接到特定数据库的驱动，而后面的 other 和 stuff 也是不固定的——也没有较强的规律，不同数据库的 URL 写法可能存在较大差异。例如，MySQL 数据库的 URL 写法如下：

```
jdbc:mysql://hostname:port/databasename
```

Oracle 数据库的 URL 写法如下：

```
jdbc:oracle:thin:@hostname:port:databasename
```

提示：

如果想了解特定数据库的 URL 写法，请查阅该数据库 JDBC 驱动的文档。

③ 通过 Connection 对象创建 Statement 对象。Connection 创建 Statement 的方法有如下三个。

- createStatement()：创建基本的 Statement 对象。
- prepareStatement(String sql)：根据传入的 SQL 语句创建预编译的 Statement 对象。
- prepareCall(String sql)：根据传入的 SQL 语句创建 CallableStatement 对象。

④ 使用 Statement 执行 SQL 语句。所有的 Statement 都有如下三个方法来执行 SQL 语句。

- execute()：可以执行任何 SQL 语句，但比较麻烦。
- executeUpdate()：主要用于执行 DML 和 DDL 语句。执行 DML 语句返回受 SQL 语句影响的行

数，执行 DDL 语句返回 0。

- executeQuery()：只能执行查询语句，执行后返回代表查询结果的 ResultSet 对象。

⑤ 操作结果集。如果执行的 SQL 语句是查询语句，则执行结果将返回一个 ResultSet 对象，该对象里保存了 SQL 语句查询的结果。程序可以通过操作该 ResultSet 对象来取出查询结果。ResultSet 对象主要提供了如下两类方法。

- next()、previous()、first()、last()、beforeFirst()、afterLast()、absolute()等移动记录指针的方法。
- getXxx()方法获取记录指针指向行、特定列的值。该方法既可使用列索引作为参数，也可使用列名作为参数。使用列索引作为参数性能更好，使用列名作为参数可读性更好。

ResultSet 实质是一个查询结果集，在逻辑结构上非常类似于一个表。图 13.17 显示了 ResultSet 的逻辑结构，以及操作 ResultSet 结果集并获取值的方法示意图。

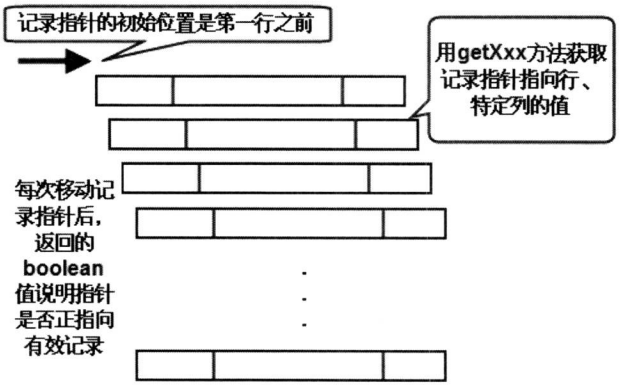


图 13.17　ResultSet 结果集示意图

⑥ 回收数据库资源，包括关闭 ResultSet、Statement 和 Connection 等资源。

下面程序简单示范了 JDBC 编程，并通过 ResultSet 获得结果集的过程。

程序清单：codes\13\13.3\ConnMySql.java

```
public class ConnMySql
{
    public static void main(String[] args) throws Exception
    {
        // 1.加载驱动，使用反射知识，现在记住这么写
        Class.forName("com.mysql.jdbc.Driver");
        try(
            // 2.使用 DriverManager 获取数据库连接
            // 其中返回的 Connection 就代表了 Java 程序和数据库的连接
            // 不同数据库的 URL 写法需要查驱动文档，用户名、密码由 DBA 分配
            Connection conn = DriverManager.getConnection(
                "jdbc:mysql://127.0.0.1:3306/select_test?useSSL=true"
                , "root" , "32147");
            // 3.使用 Connection 来创建一个 Statement 对象
            Statement stmt = conn.createStatement();
            // 4.执行 SQL 语句
            /*
            Statement 有三种执行 SQL 语句的方法：
            1. execute()可执行任何 SQL 语句—返回一个 boolean 值
              如果执行后第一个结果是 ResultSet，则返回 true，否则返回 false
            2. executeQuery()执行 select 语句 — 返回查询到的结果集
            3. executeUpdate()用于执行 DML 语句— 返回一个整数
              代表被 SQL 语句影响的记录条数
            */
            ResultSet rs = stmt.executeQuery("select s.* , teacher_name"
                + " from student_table s , teacher_table t"
                + " where t.teacher_id = s.java_teacher"))
        {
            // ResultSet 有一系列的 getXxx(列索引 | 列名)方法，用于获取记录指针
            // 指向行、特定列的值，不断地使用 next()将记录指针下移一行
```

```
            // 如果移动之后记录指针依然指向有效行，则 next()方法返回 true
            while(rs.next())
            {
                System.out.println(rs.getInt(1) + "\t"
                    + rs.getString(2) + "\t"
                    + rs.getString(3) + "\t"
                    + rs.getString(4));
            }
        }
    }
}
```

上面程序严格按 JDBC 访问数据库的步骤执行了一条多表连接查询语句，这条连接查询语句就是前面介绍 SQL 92 连接时所讲的连接查询语句。

提示：

目前最新的 MySQL 驱动推荐使用 SSL 连接（安全的网络连接，这样可以保证更好的数据安全性）。

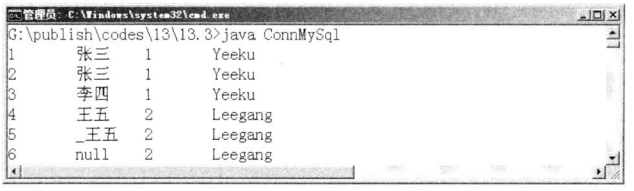


图 13.18　使用 JDBC 执行查询的结果

与前面介绍的步骤略有区别的是，本程序采用了自动关闭资源的 try 语句来关闭各种数据库资源，Java 7 改写了 Connection、Statement、ResultSet 等接口，它们都继承了 AutoCloseable 接口，因此它们都可以由 try 语句来关闭。

运行上面程序，会看到如图 13.18 所示的结果。

提示：

上面的运行结果也是基于前面所使用的 select_test 数据库，所以运行该程序之前应该先导入 codes\13\13.2 路径下的 select_data.sql 文件。除此之外，运行本程序时需要使用 MySQL 数据库驱动，该驱动 JAR 文件就是 codes\13\mysql-connector-java-5.1.44-bin.jar 文件，读者应该把该文件添加到系统的 CLASSPATH 环境变量里，或者直接使用 codes\13\13.3\路径下的 runConnMySql.cmd 来运行该程序，本章所有的程序都会提供这样一个对应的 cmd 批处理文件。

13.4　执行 SQL 语句的方式

前面介绍了 JDBC 执行查询等示例程序，实际上，JDBC 不仅可以执行查询，也可以执行 DDL、DML 等 SQL 语句，从而允许通过 JDBC 最大限度地控制数据库。

13.4.1　使用 Java 8 新增的 executeLargeUpdate 方法执行 DDL 和 DML 语句

Statement 提供了三个方法来执行 SQL 语句，前面已经介绍了使用 executeQuery()来执行查询语句，下面将介绍使用 executeLargeUpdate()（或 executeUpdate()）来执行 DDL 和 DML 语句。使用 Statement 执行 DDL 和 DML 语句的步骤与执行普通查询语句的步骤基本相似，区别在于执行了 DDL 语句后返回值为 0，执行了 DML 语句后返回值为受影响的记录条数。

下面程序示范了使用 executeUpdate()方法（此处暂未使用 executeLargeUpdate()方法是因为 MySQL 驱动暂不支持）创建数据表。该示例并没有直接把数据库连接信息写在程序里，而是使用一个 mysql.ini 文件（就是一个 properties 文件）来保存数据库连接信息，这是比较成熟的做法——当需要把应用程序从开发环境移植到生产环境时，无须修改源代码，只需要修改 mysql.ini 配置文件即可。

程序清单：codes\13\13.4\ExecuteDDL.java

```
public class ExecuteDDL
{
    private String driver;
    private String url;
    private String user;
    private String pass;
```

```
    public void initParam(String paramFile)
        throws Exception
    {
        // 使用 Properties 类来加载属性文件
        Properties props = new Properties();
        props.load(new FileInputStream(paramFile));
        driver = props.getProperty("driver");
        url = props.getProperty("url");
        user = props.getProperty("user");
        pass = props.getProperty("pass");
    }
    public void createTable(String sql)throws Exception
    {
        // 加载驱动
        Class.forName(driver);
        try(
        // 获取数据库连接
        Connection conn = DriverManager.getConnection(url , user , pass);
        // 使用 Connection 来创建一个 Statement 对象
        Statement stmt = conn.createStatement())
        {
            // 执行 DDL 语句，创建数据表
            stmt.executeUpdate(sql);
        }
    }
    public static void main(String[] args) throws Exception
    {
        ExecuteDDL ed = new ExecuteDDL();
        ed.initParam("mysql.ini");
        ed.createTable("create table jdbc_test "
            + "( jdbc_id int auto_increment primary key, "
            + "jdbc_name varchar(255), "
            + "jdbc_desc text);");
        System.out.println("-----建表成功-----");
    }
}
```

运行上面程序，执行成功后会看到 select_test 数据库中添加了一个 jdbc_test 数据表，这表明 JDBC 执行 DDL 语句成功。

使用 executeUpdate()执行 DML 语句与执行 DDL 语句基本相似，区别是 executeUpdate()执行 DDL 语句后返回 0，而执行 DML 语句后返回受影响的记录条数。下面程序将会执行一条 insert 语句，这条 insert 语句会向刚刚建立的 jdbc_test 数据表中插入几条记录。因为使用了带子查询的 insert 语句，所以可以一次插入多条语句。

程序清单：codes\13\13.4\ExecuteDML.java

```
public class ExecuteDML
{
    private String driver;
    private String url;
    private String user;
    private String pass;
    public void initParam(String paramFile)
        throws Exception
    {
        // 使用 Properties 类来加载属性文件
        Properties props = new Properties();
        props.load(new FileInputStream(paramFile));
        driver = props.getProperty("driver");
        url = props.getProperty("url");
        user = props.getProperty("user");
        pass = props.getProperty("pass");
    }
    public int insertData(String sql)throws Exception
    {
        // 加载驱动
        Class.forName(driver);
```

```
        try(
            // 获取数据库连接
            Connection conn = DriverManager.getConnection(url
                , user , pass);
            // 使用 Connection 来创建一个 Statement 对象
            Statement stmt = conn.createStatement())
        {
            // 执行 DML 语句，返回受影响的记录条数
            return stmt.executeUpdate(sql);
        }
    }
    public static void main(String[] args)throws Exception
    {
        ExecuteDML ed = new ExecuteDML();
        ed.initParam("mysql.ini");
        int result = ed.insertData("insert into jdbc_test(jdbc_name,jdbc_desc)"
            + "select s.student_name , t.teacher_name "
            + "from student_table s , teacher_table t "
            + "where s.java_teacher = t.teacher_id;");
        System.out.println("--系统中共有" + result + "条记录受影响--");
    }
}
```

运行上面程序，执行成功将会看到 jdbc_test 数据表中多了几条记录，而且在程序控制台会看到输出有几条记录受影响的信息。

13.4.2 使用 execute 方法执行 SQL 语句

Statement 的 execute()方法几乎可以执行任何 SQL 语句，但它执行 SQL 语句时比较麻烦，通常没有必要使用 execute()方法来执行 SQL 语句，使用 executeQuery()或 executeUpdate()方法更简单。但如果不清楚 SQL 语句的类型，则只能使用 execute()方法来执行该 SQL 语句了。

使用 execute()方法执行 SQL 语句的返回值只是 boolean 值，它表明执行该 SQL 语句是否返回了 ResultSet 对象。那么如何来获取执行 SQL 语句后得到的 ResultSet 对象呢？Statement 提供了如下两个方法来获取执行结果。

- getResultSet()：获取该 Statement 执行查询语句所返回的 ResultSet 对象。
- getUpdateCount()：获取该 Statement()执行 DML 语句所影响的记录行数。

下面程序示范了使用 Statement 的 execute()方法来执行任意的 SQL 语句，执行不同的 SQL 语句时产生不同的输出。

程序清单：codes\13\13.4\ExecuteSQL.java

```
public class ExecuteSQL
{
    private String driver;
    private String url;
    private String user;
    private String pass;
    public void initParam(String paramFile)throws Exception
    {
        // 使用 Properties 类来加载属性文件
        Properties props = new Properties();
        props.load(new FileInputStream(paramFile));
        driver = props.getProperty("driver");
        url = props.getProperty("url");
        user = props.getProperty("user");
        pass = props.getProperty("pass");
    }
    public void executeSql(String sql)throws Exception
    {
        // 加载驱动
        Class.forName(driver);
        try(
            // 获取数据库连接
            Connection conn = DriverManager.getConnection(url
                , user , pass);
```

```
            // 使用 Connection 来创建一个 Statement 对象
            Statement stmt = conn.createStatement())
        {
            // 执行 SQL 语句，返回 boolean 值表示是否包含 ResultSet
            boolean hasResultSet = stmt.execute(sql);
            //如果执行后有 ResultSet 结果集
            if (hasResultSet)
            {
                try(
                    // 获取结果集
                    ResultSet rs = stmt.getResultSet())
                {
                    // ResultSetMetaData 是用于分析结果集的元数据接口
                    ResultSetMetaData rsmd = rs.getMetaData();
                    int columnCount = rsmd.getColumnCount();
                    // 迭代输出 ResultSet 对象
                    while (rs.next())
                    {
                        // 依次输出每列的值
                        for (int i = 0 ; i < columnCount ; i++ )
                        {
                            System.out.print(rs.getString(i + 1) + "\t");
                        }
                        System.out.print("\n");
                    }
                }
            }
            else
            {
                System.out.println("该 SQL 语句影响的记录有"
                    + stmt.getUpdateCount() + "条");
            }
        }
    }
    public static void main(String[] args) throws Exception
    {
        ExecuteSQL es = new ExecuteSQL();
        es.initParam("mysql.ini");
        System.out.println("------执行删除表的 DDL 语句-----");
        es.executeSql("drop table if exists my_test");
        System.out.println("------执行建表的 DDL 语句-----");
        es.executeSql("create table my_test"
            + "(test_id int auto_increment primary key, "
            + "test_name varchar(255))");
        System.out.println("------执行插入数据的 DML 语句-----");
        es.executeSql("insert into my_test(test_name) "
            + "select student_name from student_table");
        System.out.println("------执行查询数据的查询语句-----");
        es.executeSql("select * from my_test");
    }
}
```

运行上面程序，会看到使用 Statement 的不同方法执行不同 SQL 语句的效果。执行 DDL 语句显示受影响的记录条数为 0；执行 DML 语句显示插入、修改或删除的记录条数；执行查询语句则可以输出查询结果。

提示：

上面程序获得 SQL 执行结果时没有根据各列的数据类型调用相应的 getXxx()方法，而是直接使用 getString()方法来取得值，这是可以的。ResultSet 的 getString()方法几乎可以获取除 Blob 之外的任意类型列的值，因为所有的数据类型都可以自动转换成字符串类型。

13.4.3 使用 PreparedStatement 执行 SQL 语句

如果经常需要反复执行一条结构相似的 SQL 语句，例如如下两条 SQL 语句：

```
insert into student_table values(null,'张三',1);
insert into student_table values(null,'李四',2);
```

对于这两条 SQL 语句而言，它们的结构基本相似，只是执行插入时插入的值不同而已。对于这种情况，可以使用带占位符（?）参数的 SQL 语句来代替它：

```
insert into student_table values(null,?,?);
```

但 Statement 执行 SQL 语句时不允许使用问号占位符参数，而且这个问号占位符参数必须获得值后才可以执行。为了满足这种功能，JDBC 提供了 PreparedStatement 接口，它是 Statement 接口的子接口，它可以预编译 SQL 语句，预编译后的 SQL 语句被存储在 PreparedStatement 对象中，然后可以使用该对象多次高效地执行该语句。简而言之，使用 PreparedStatement 比使用 Statement 的效率要高。

创建 PreparedStatement 对象使用 Connection 的 prepareStatement()方法，该方法需要传入一个 SQL 字符串，该 SQL 字符串可以包含占位符参数。如下代码所示：

```
// 创建一个 PreparedStatement 对象
pstmt = conn.prepareStatement("insert into student_table values(null,?,1)");
```

PreparedStatement 也提供了 execute()、executeUpdate()、executeQuery()三个方法来执行 SQL 语句，不过这三个方法无须参数，因为 PreparedStatement 已存储了预编译的 SQL 语句。

使用 PreparedStatement 预编译 SQL 语句时，该 SQL 语句可以带占位符参数，因此在执行 SQL 语句之前必须为这些参数传入参数值，PreparedStatement 提供了一系列的 setXxx(int index , Xxx value)方法来传入参数值。

提示：

如果程序很清楚 PreparedStatement 预编译 SQL 语句中各参数的类型，则使用相应的 setXxx()方法来传入参数即可；如果程序不清楚预编译 SQL 语句中各参数的类型，则可以使用 setObject()方法来传入参数，由 PreparedStatement 来负责类型转换。

下面程序示范了使用 Statement 和 PreparedStatement 分别插入 100 条记录的对比。使用 Statement 需要传入 100 条 SQL 语句，但使用 PreparedStatement 则只需要传入 1 条预编译的 SQL 语句，然后 100 次为该 PreparedStatement 的参数设值即可。

程序清单：codes\13\13.4\PreparedStatementTest.java

```
public class PreparedStatementTest
{
    private String driver;
    private String url;
    private String user;
    private String pass;
    public void initParam(String paramFile)throws Exception
    {
        // 使用 Properties 类来加载属性文件
        Properties props = new Properties();
        props.load(new FileInputStream(paramFile));
        driver = props.getProperty("driver");
        url = props.getProperty("url");
        user = props.getProperty("user");
        pass = props.getProperty("pass");
        // 加载驱动
        Class.forName(driver);
    }
    public void insertUseStatement()throws Exception
    {
        long start = System.currentTimeMillis();
        try(
            // 获取数据库连接
            Connection conn = DriverManager.getConnection(url
                , user , pass);
            // 使用 Connection 来创建一个 Statement 对象
            Statement stmt = conn.createStatement())
        {
```

```
            // 需要使用 100 条 SQL 语句来插入 100 条记录
            for (int i = 0; i < 100 ; i++ )
            {
                stmt.executeUpdate("insert into student_table values("
                    + " null ,'姓名" + i + "' , 1)");
            }
            System.out.println("使用 Statement 费时:"
                + (System.currentTimeMillis() - start));
        }
    }
    public void insertUsePrepare()throws Exception
    {
        long start = System.currentTimeMillis();
        try(
            // 获取数据库连接
            Connection conn = DriverManager.getConnection(url
                , user , pass);
            // 使用 Connection 来创建一个 PreparedStatement 对象
            PreparedStatement pstmt = conn.prepareStatement(
                "insert into student_table values(null,?,1)"))
        {
            // 100 次为 PreparedStatement 的参数设值，就可以插入 100 条记录
            for (int i = 0; i < 100 ; i++ )
            {
                pstmt.setString(1 , "姓名" + i);
                pstmt.executeUpdate();
            }
            System.out.println("使用 PreparedStatement 费时:"
                + (System.currentTimeMillis() - start));
        }
    }
    public static void main(String[] args) throws Exception
    {
        PreparedStatementTest pt = new PreparedStatementTest();
        pt.initParam("mysql.ini");
        pt.insertUseStatement();
        pt.insertUsePrepare();
    }
}
```

多次运行上面程序，可以发现使用 PreparedStatement 插入 100 条记录所用的时间比使用 Statement 插入 100 条记录所用的时间少，这表明 PreparedStatement 的执行效率比 Statement 的执行效率高。

除此之外，使用 PreparedStatement 还有一个优势——当 SQL 语句中要使用参数时，无须“拼接”SQL 字符串。而使用 Statement 则要“拼接”SQL 字符串，如上程序中粗体字代码所示，这是相当容易出现错误的——注意粗体字代码中的单引号，这是因为 SQL 语句中的字符串必须用单引号引起来。尤其是当 SQL 语句中有多个字符串参数时，“拼接”这条 SQL 语句时就更容易出错了。使用 PreparedStatement 则只需要使用问号占位符来代替这些参数即可，降低了编程复杂度。

使用 PreparedStatement 还有一个很好的作用——用于防止 SQL 注入。

提示：

SQL 注入是一个较常见的 Cracker 入侵方式，它利用 SQL 语句的漏洞来入侵。

下面以一个简单的登录窗口为例来介绍这种 SQL 注入的结果。下面登录窗口包含两个文本框，一个用于输入用户名，一个用于输入密码，系统根据用户输入与 jdbc_test 表里的记录进行匹配，如果找到相应记录则提示登录成功。

程序清单：codes\13\13.4\LoginFrame.java

```
public class LoginFrame
{
    private final String PROP_FILE = "mysql.ini";
    private String driver;
    // url 是数据库的服务地址
    private String url;
```

```
    private String user;
    private String pass;
    // 登录界面的 GUI 组件
    private JFrame jf = new JFrame("登录");
    private JTextField userField = new JTextField(20);
    private JTextField passField = new JTextField(20);
    private JButton loginButton = new JButton("登录");
    public void init()throws Exception
    {
        Properties connProp = new Properties();
        connProp.load(new FileInputStream(PROP_FILE));
        driver = connProp.getProperty("driver");
        url = connProp.getProperty("url");
        user = connProp.getProperty("user");
        pass = connProp.getProperty("pass");
        // 加载驱动
        Class.forName(driver);
        // 为登录按钮添加事件监听器
        loginButton.addActionListener(e -> {
            // 登录成功则显示“登录成功”
            if (validate(userField.getText(), passField.getText()))
            {
                JOptionPane.showMessageDialog(jf, "登录成功");
            }
            // 否则显示“登录失败”
            else
            {
                JOptionPane.showMessageDialog(jf, "登录失败");
            }
        });
        jf.add(userField , BorderLayout.NORTH);
        jf.add(passField);
        jf.add(loginButton , BorderLayout.SOUTH);
        jf.pack();
        jf.setVisible(true);
    }
    private boolean validate(String userName, String userPass)
    {
        // 执行查询的 SQL 语句
        String sql = "select * from jdbc_test "
            + "where jdbc_name='" + userName
            + "' and jdbc_desc='" + userPass + "'";
        System.out.println(sql);
        try(
            Connection conn = DriverManager.getConnection(url , user ,pass);
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery(sql))
        {
            // 如果查询的 ResultSet 里有超过一条的记录，则登录成功
            if (rs.next())
            {
                return true;
            }
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
        return false;
    }
    public static void main(String[] args) throws Exception
    {
        new LoginFrame().init();
    }
}
```

运行上面程序，如果用户正常输入其用户名、密码当然没有问题，输入正确可以正常登录，输入错误将提示输入失败。但如果这个用户是一个 Cracker，他可以按图 13.19 所示来输入。

图 13.19 所示的输入明显不正确，但当单击“登录”按钮后也会显示“登录成功”对话框。可以在程序运行的后台看到如下 SQL 语句：

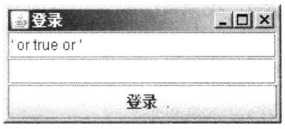


图 13.19　利用 SQL 注入

```
    # 利用 SQL 注入后生成的 SQL 语句
    select * from jdbc_test where jdbc_name='' or true or
'' and jdbc_desc=''
```

看到这条 SQL 语句，读者应该不难明白为什么这样输入也可以显示“正常登录”对话框了，因为 Cracker 直接输入了 true，而 SQL 把这个 true 当成了直接量。

提示：

JDBC 编程本身并没有提供图形界面功能，它仅仅提供了数据库访问支持。如果希望 JDBC 程序有较好的图形用户界面，则需要结合前面介绍的 AWT 或 Swing 编程才可以做到。在 Web 编程中，数据库访问也是非常重要的基础知识。

如果把上面的 validate()方法换成使用 PreparedStatement 来执行验证，而不是直接使用 Statement。程序如下：

```
private boolean validate(String userName, String userPass)
{
    try(
        Connection conn = DriverManager.getConnection(url
            , user ,pass);
        PreparedStatement pstmt = conn.prepareStatement(
            "select * from jdbc_test where jdbc_name=? and jdbc_desc=?"))
    {
        pstmt.setString(1, userName);
        pstmt.setString(2, userPass);
        try(
            ResultSet rs = pstmt.executeQuery())
        {
            // 如果查询的 ResultSet 里有超过一条的记录，则登录成功
            if (rs.next())
            {
                return true;
            }
        }
    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
    return false;
}
```

将上面的 validate()方法改为使用 PrepareStatement 来执行 SQL 语句之后，即使用户按图 13.19 所示输入，系统一样会显示“登录失败”对话框。

总体来看，使用 PreparedStatement 比使用 Statement 多了如下三个好处。

- PreparedStatement 预编译 SQL 语句，性能更好。
- PreparedStatement 无须“拼接”SQL 语句，编程更简单。
- PreparedStatement 可以防止 SQL 注入，安全性更好。

基于以上三点，通常推荐避免使用 Statement 来执行 SQL 语句，改为使用 PreparedStatement 执行 SQL 语句。

注意：

使用 PreparedStatement 执行带占位符参数的 SQL 语句时，SQL 语句中的占位符参数只能代替普通值，不要使用占位符参数代替表名、列名等数据库对象，更不要用占位符参数来代替 SQL 语句中的 insert、select 等关键字。

13.4.4 使用 CallableStatement 调用存储过程

下面的 SQL 语句可以在 MySQL 数据库中创建一个简单的存储过程。

```
delimiter //
create procedure add_pro(a int , b int, out sum int)
begin
set sum = a + b;
end;
//
```

上面的 SQL 语句将 MySQL 的语句结束符改为双斜线（//），这样就可以在创建存储过程中使用分号作为分隔符（MySQL 默认使用分号作为语句结束符）。上面程序创建了名为 add_pro 的存储过程，该存储过程包含三个参数：a、b 是传入参数，而 sum 使用 out 修饰，是传出参数。

提示：

关于存储过程的介绍请读者自行查阅相关书籍，本书仅介绍如何使用 JDBC 调用存储过程，并不会介绍创建存储过程的知识。

调用存储过程使用 CallableStatement，可以通过 Connection 的 prepareCall()方法来创建 CallableStatement 对象，创建该对象时需要传入调用存储过程的 SQL 语句。调用存储过程的 SQL 语句总是这种格式：{call 过程名(?,?,?...)}，其中的问号作为存储过程参数的占位符。例如，如下代码就创建了调用上面存储过程的 CallableStatement 对象。

```
// 使用 Connection 来创建一个 CallableStatement 对象
cstmt = conn.prepareCall("{call add_pro(?,?,?)}");
```

存储过程的参数既有传入参数，也有传出参数。所谓传入参数就是 Java 程序必须为这些参数传入值，可以通过 CallableStatement 的 setXxx()方法为传入参数设置值；所谓传出参数就是 Java 程序可以通过该参数获取存储过程里的值，CallableStatement 需要调用 registerOutParameter()方法来注册该参数。如下代码所示：

```
// 注册 CallableStatement 的第三个参数是 int 类型
cstmt.registerOutParameter(3, Types.INTEGER);
```

经过上面步骤之后，就可以调用 CallableStatement 的 execute()方法来执行存储过程了，执行结束后通过 CallableStatement 对象的 getXxx(int index)方法来获取指定传出参数的值。下面程序示范了如何来调用该存储过程。

程序清单：codes\13\13.4\CallableStatementTest.java

```
public class CallableStatementTest
{
    private String driver;
    private String url;
    private String user;
    private String pass;
    public void initParam(String paramFile)throws Exception
    {
        // 使用 Properties 类来加载属性文件
        Properties props = new Properties();
        props.load(new FileInputStream(paramFile));
        driver = props.getProperty("driver");
        url = props.getProperty("url");
        user = props.getProperty("user");
        pass = props.getProperty("pass");
    }
    public void callProcedure()throws Exception
    {
        // 加载驱动
        Class.forName(driver);
        try(
            // 获取数据库连接
            Connection conn = DriverManager.getConnection(url
```

```
                , user , pass);
            // 使用 Connection 来创建一个 CallableStatement 对象
            CallableStatement cstmt = conn.prepareCall(
                "{call add_pro(?,?,?)}"))
        {
            cstmt.setInt(1, 4);
            cstmt.setInt(2, 5);
            // 注册 CallableStatement 的第三个参数是 int 类型
            cstmt.registerOutParameter(3, Types.INTEGER);
            // 执行存储过程
            cstmt.execute();
            // 获取并输出存储过程传出参数的值
            System.out.println("执行结果是: " + cstmt.getInt(3));
        }
    }
    public static void main(String[] args) throws Exception
    {
        CallableStatementTest ct = new CallableStatementTest();
        ct.initParam("mysql.ini");
        ct.callProcedure();
    }
}
```

上面程序中的粗体字代码就是执行存储过程的关键代码，运行上面程序将会看到这个简单存储过程的执行结果，传入参数分别是 4、5，执行加法后传出总和 9。

13.5 管理结果集

JDBC 使用 ResultSet 来封装执行查询得到的查询结果，然后通过移动 ResultSet 的记录指针来取出结果集的内容。除此之外，JDBC 还允许通过 ResultSet 来更新记录，并提供了 ResultSetMetaData 来获得 ResultSet 对象的相关信息。

13.5.1 可滚动、可更新的结果集

前面提到，ResultSet 定位记录指针的方法有 absolute()、previous()等方法，但前面程序自始至终都只用了 next()方法来移动记录指针，实际上也可以使用 absolute()、previous()、last()等方法来移动记录指针。可以使用 absolute()、previous()、afterLast()等方法自由移动记录指针的 ResultSet 被称为可滚动的结果集。

提示：

在 JDK 1.4 以前，默认打开的 ResultSet 是不可滚动的，必须在创建 Statement 或 PreparedStatement 时传入额外的参数。从 Java 5.0 以后，默认打开的 ResultSet 就是可滚动的，无须传入额外的参数。

以默认方式打开的 ResultSet 是不可更新的，如果希望创建可更新的 ResultSet，则必须在创建 Statement 或 PreparedStatement 时传入额外的参数。Connection 在创建 Statement 或 PreparedStatement 时还可额外传入如下两个参数。

- resultSetType：控制 ResultSet 的类型，该参数可以取如下三个值。
  - ResultSet.TYPE_FORWARD_ONLY：该常量控制记录指针只能向前移动。这是 JDK 1.4 以前的默认值。
  - ResultSet.TYPE_SCROLL_INSENSITIVE：该常量控制记录指针可以自由移动（可滚动结果集），但底层数据的改变不会影响 ResultSet 的内容。
  - ResultSet.TYPE_SCROLL_SENSITIVE：该常量控制记录指针可以自由移动（可滚动结果集），而且底层数据的改变会影响 ResultSet 的内容。

- resultSetConcurrency：控制 ResultSet 的并发类型，该参数可以接收如下两个值。
  - ResultSet.CONCUR_READ_ONLY：该常量指示 ResultSet 是只读的并发模式（默认）。
  - ResultSet.CONCUR_UPDATABLE：该常量指示 ResultSet 是可更新的并发模式。

下面代码通过这两个参数创建了一个 PreparedStatement 对象，由该对象生成的 ResultSet 对象将是可滚动、可更新的结果集。

```
// 使用 Connection 创建一个 PreparedStatement 对象
// 传入控制结果集可滚动、可更新的参数
pstmt = conn.prepareStatement(sql , ResultSet.TYPE_SCROLL_INSENSITIVE
    , ResultSet.CONCUR_UPDATABLE);
```

需要指出的是，可更新的结果集还需要满足如下两个条件。

- 所有数据都应该来自一个表。
- 选出的数据集必须包含主键列。

通过该 PreparedStatement 创建的 ResultSet 就是可滚动、可更新的，程序可调用 ResultSet 的 updateXxx(int columnIndex , Xxx value)方法来修改记录指针所指记录、特定列的值，最后调用 ResultSet 的 updateRow()方法来提交修改。

Java 8 为 ResultSet 添加了 updateObject(String columnLabel, Object x, SQLType targetSqlType)和 updateObject(int columnIndex, Object x, SQLType targetSqlType)两个默认方法，这两个方法可以直接用 Object 来修改记录指针所指记录、特定列的值，其中 SQLType 用于指定该数据列的类型。但目前最新的 MySQL 驱动暂不支持该方法。

下面程序示范了这种创建可滚动、可更新的结果集的方法。

程序清单：codes\13\13.5\ResultSetTest.java

```
public class ResultSetTest
{
    private String driver;
    private String url;
    private String user;
    private String pass;
    public void initParam(String paramFile)throws Exception
    {
        // 使用 Properties 类来加载属性文件
        Properties props = new Properties();
        props.load(new FileInputStream(paramFile));
        driver = props.getProperty("driver");
        url = props.getProperty("url");
        user = props.getProperty("user");
        pass = props.getProperty("pass");
    }
    public void query(String sql)throws Exception
    {
        // 加载驱动
        Class.forName(driver);
        try(
            // 获取数据库连接
            Connection conn = DriverManager.getConnection(url, user , pass);
            // 使用 Connection 来创建一个 PreparedStatement 对象
            // 传入控制结果集可滚动、可更新的参数
            PreparedStatement pstmt = conn.prepareStatement(sql
                , ResultSet.TYPE_SCROLL_INSENSITIVE
                , ResultSet.CONCUR_UPDATABLE);
            ResultSet rs = pstmt.executeQuery())
        {
            rs.last();
```

```
            int rowCount = rs.getRow();
            for (int i = rowCount; i > 0 ; i-- )
            {
                rs.absolute(i);
                System.out.println(rs.getString(1) + "\t"
                    + rs.getString(2) + "\t" + rs.getString(3));
                // 修改记录指针所指记录、第 2 列的值
                rs.updateString(2 , "学生名" + i);
                // 提交修改
                rs.updateRow();
            }
        }
    }
    public static void main(String[] args) throws Exception
    {
        ResultSetTest rt = new ResultSetTest();
        rt.initParam("mysql.ini");
        rt.query("select * from student_table");
    }
}
```

上面程序中的粗体字代码示范了如何自由移动记录指针并更新记录指针所指的记录。运行上面程序，将会看到 student_table 表中的记录被倒过来输出了，因为是从最大记录行开始输出的。而且当程序运行结束后，student_table 表中所有记录的 student_name 列的值都被修改了。

注意：

如果要创建可更新的结果集，则使用查询语句查询的数据通常只能来自于一个数据表，而且查询结果集中的数据列必须包含主键列，否则将会引起更新失败。

13.5.2　处理 Blob 类型数据

Blob（Binary Long Object）是二进制长对象的意思，Blob 列通常用于存储大文件，典型的 Blob 内容是一张图片或一个声音文件，由于它们的特殊性，必须使用特殊的方式来存储。使用 Blob 列可以把图片、声音等文件的二进制数据保存在数据库里，并可以从数据库里恢复指定文件。

如果需要将图片插入数据库，显然不能直接通过普通的 SQL 语句来完成，因为有一个关键的问题——Blob 常量无法表示。所以将 Blob 数据插入数据库需要使用 PreparedStatement，该对象有一个方法：setBinaryStream(int parameterIndex, InputStream x)，该方法可以为指定参数传入二进制输入流，从而可以实现将 Blob 数据保存到数据库的功能。

当需要从 ResultSet 里取出 Blob 数据时，可以调用 ResultSet 的 getBlob(int columnIndex)方法，该方法将返回一个 Blob 对象，Blob 对象提供了 getBinaryStream()方法来获取该 Blob 数据的输入流，也可以使用 Blob 对象提供的 getBytes()方法直接取出该 Blob 对象封装的二进制数据。

为了把图片放入数据库，本程序先使用如下 SQL 语句来建立一个数据表。

```
create table img_table
(
    img_id int auto_increment primary key,
    img_name varchar(255),
    # 创建一个 mediumblob 类型的数据列，用于保存图片数据
    img_data mediumblob
);
```

提示：

上面 SQL 语句中的 img_data 列使用 mediumblob 类型，而不是 blob 类型。因为 MySQL 数据库里的 blob 类型最多只能存储 64KB 内容，这可能不够满足实际用途。所以使用 mediumblob 类型，该类型的数据列可以存储 16MB 内容。

下面程序可以实现图片“上传”——实际上就是将图片保存到数据库，并在右边的列表框中显示图片的名字，当用户双击列表框中的图片名时，左边窗口将显示该图片——实质就是根据选中的 ID 从数

据库里查找图片，并将其显示出来。

程序清单：codes\13\13.5\BlobTest.java

```
public class BlobTest
{
    JFrame jf = new JFrame("图片管理程序");
    private static Connection conn;
    private static PreparedStatement insert;
    private static PreparedStatement query;
    private static PreparedStatement queryAll;
    // 定义一个 DefaultListModel 对象
    private DefaultListModel<ImageHolder> imageModel
        = new DefaultListModel<>();
    private JList<ImageHolder> imageList = new JList<>(imageModel);
    private JTextField filePath = new JTextField(26);
    private JButton browserBn = new JButton("...");
    private JButton uploadBn = new JButton("上传");
    private JLabel imageLabel = new JLabel();
    // 以当前路径创建文件选择器
    JFileChooser chooser = new JFileChooser(".");
    // 创建文件过滤器
    ExtensionFileFilter filter = new ExtensionFileFilter();
    static
    {
        try
        {
            Properties props = new Properties();
            props.load(new FileInputStream("mysql.ini"));
            String driver = props.getProperty("driver");
            String url = props.getProperty("url");
            String user = props.getProperty("user");
            String pass = props.getProperty("pass");
            Class.forName(driver);
            // 获取数据库连接
            conn = DriverManager.getConnection(url , user , pass);
            // 创建执行插入的 PreparedStatement 对象
            // 该对象执行插入后可以返回自动生成的主键
            insert = conn.prepareStatement("insert into img_table"
                + " values(null,?,?)" , Statement.RETURN_GENERATED_KEYS);
            // 创建两个 PreparedStatement 对象，用于查询指定图片，查询所有图片
            query = conn.prepareStatement("select img_data from img_table"
                + " where img_id=?");
            queryAll = conn.prepareStatement("select img_id, "
                + " img_name from img_table");
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
    public void init()throws SQLException
    {
        // -------初始化文件选择器--------
        filter.addExtension("jpg");
        filter.addExtension("jpeg");
        filter.addExtension("gif");
        filter.addExtension("png");
        filter.setDescription("图片文件(*.jpg,*.jpeg,*.gif,*.png)");
        chooser.addChoosableFileFilter(filter);
        // 禁止“文件类型”下拉列表中显示“所有文件”选项
        chooser.setAcceptAllFileFilterUsed(false);
        // ---------初始化程序界面---------
        fillListModel();
        filePath.setEditable(false);
        // 只能单选
        imageList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
        JPanel jp = new JPanel();
        jp.add(filePath);
```

```
        jp.add(browserBn);
        browserBn.addActionListener(event -> {
            // 显示文件对话框
            int result = chooser.showDialog(jf , "浏览图片文件上传");
            // 如果用户选择了 APPROVE（赞同）按钮，即打开，保存等效按钮
            if(result == JFileChooser.APPROVE_OPTION)
            {
                filePath.setText(chooser.getSelectedFile().getPath());
            }
        });
        jp.add(uploadBn);
        uploadBn.addActionListener(avt -> {
            // 如果上传文件的文本框有内容
            if (filePath.getText().trim().length() > 0)
            {
                // 将指定文件保存到数据库
                upload(filePath.getText());
                // 清空文本框内容
                filePath.setText("");
            }
        });
        JPanel left = new JPanel();
        left.setLayout(new BorderLayout());
        left.add(new JScrollPane(imageLabel) , BorderLayout.CENTER);
        left.add(jp , BorderLayout.SOUTH);
        jf.add(left);
        imageList.setFixedCellWidth(160);
        jf.add(new JScrollPane(imageList) , BorderLayout.EAST);
        imageList.addMouseListener(new MouseAdapter()
        {
            public void mouseClicked(MouseEvent e)
            {
                // 如果鼠标双击
                if (e.getClickCount() >= 2)
                {
                    // 取出选中的 List 项
                    ImageHolder cur = (ImageHolder)imageList.
                    getSelectedValue();
                    try
                    {
                        // 显示选中项对应的 Image
                        showImage(cur.getId());
                    }
                    catch (SQLException sqle)
                    {
                        sqle.printStackTrace();
                    }
                }
            }
        });
        jf.setSize(620, 400);
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setVisible(true);
    }
    // ----------查找 img_table 填充 ListModel----------
    public void fillListModel()throws SQLException
    {

        try(
            // 执行查询
            ResultSet rs = queryAll.executeQuery())
        {
            // 先清除所有元素
            imageModel.clear();
            // 把查询的全部记录添加到 ListModel 中
            while (rs.next())
            {
                imageModel.addElement(new ImageHolder(rs.getInt(1)
                    ,rs.getString(2)));
```

```
            }
        }
    }
    // ---------将指定图片放入数据库---------
    public void upload(String fileName)
    {
        // 截取文件名
        String imageName = fileName.substring(fileName.lastIndexOf('\\')
            + 1 , fileName.lastIndexOf('.'));
        File f = new File(fileName);
        try(
            InputStream is = new FileInputStream(f))
        {
            // 设置图片名参数
            insert.setString(1, imageName);
            // 设置二进制流参数
            insert.setBinaryStream(2, is , (int)f.length());
            int affect = insert.executeUpdate();
            if (affect == 1)
            {
                // 重新更新 ListModel，将会让 JList 显示最新的图片列表
                fillListModel();
            }
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
    // ---------根据图片 ID 来显示图片----------
    public void showImage(int id)throws SQLException
    {
        // 设置参数
        query.setInt(1, id);
        try(
            // 执行查询
            ResultSet rs = query.executeQuery())
        {
            if (rs.next())
            {
                // 取出 Blob 列
                Blob imgBlob = rs.getBlob(1);
                // 取出 Blob 列里的数据
                ImageIcon icon=new ImageIcon(imgBlob.getBytes(1L
                    ,(int)imgBlob.length()));
                imageLabel.setIcon(icon);
            }
        }
    }
    public static void main(String[] args)throws SQLException
    {
        new BlobTest().init();
    }
}
// 创建 FileFilter 的子类，用以实现文件过滤功能
class ExtensionFileFilter extends FileFilter
{
    private String description = "";
    private ArrayList<String> extensions = new ArrayList<>();
    // 自定义方法，用于添加文件扩展名
    public void addExtension(String extension)
    {
        if (!extension.startsWith("."))
        {
            extension = "." + extension;
            extensions.add(extension.toLowerCase());
        }
    }
    // 用于设置该文件过滤器的描述文本
```

```
    public void setDescription(String aDescription)
    {
        description = aDescription;
    }
    // 继承 FileFilter 类必须实现的抽象方法，返回该文件过滤器的描述文本
    public String getDescription()
    {
        return description;
    }
    // 继承 FileFilter 类必须实现的抽象方法，判断该文件过滤器是否接受该文件
    public boolean accept(File f)
    {
        // 如果该文件是路径，接受该文件
        if (f.isDirectory()) return true;
        // 将文件名转为小写（全部转为小写后比较，用于忽略文件名大小写）
        String name = f.getName().toLowerCase();
        // 遍历所有可接受的扩展名，如果扩展名相同，该文件就可接受
        for (String extension : extensions)
        {
            if (name.endsWith(extension))
            {
                return true;
            }
        }
        return false;
    }
}
// 创建一个 ImageHolder 类，用于封装图片名、图片 ID
class ImageHolder
{
    // 封装图片的 ID
    private int id;
    // 封装图片的名字
    private String name;
    public ImageHolder(){}
    public ImageHolder(int id , String name)
    {
        this.id = id;
        this.name = name;
    }
    // id 的 setter 和 getter 方法
    public void setId(int id)
    {
        this.id = id;
    }
    public int getId()
    {
        return this.id;
    }
    // name 的 setter 和 getter 方法
    public void setName(String name)
    {
        this.name = name;
    }
    public String getName()
    {
        return this.name;
    }
    // 重写 toString()方法，返回图片名
    public String toString()
    {
        return name;
    }
}
```

上面程序中的第一段粗体字代码用于控制将一个图片文件保存到数据库，第二段粗体字代码用于控制将数据库里的图片数据显示出来。运行上面程序，并上传一些图片，会看到如图 13.20 所示的界面。

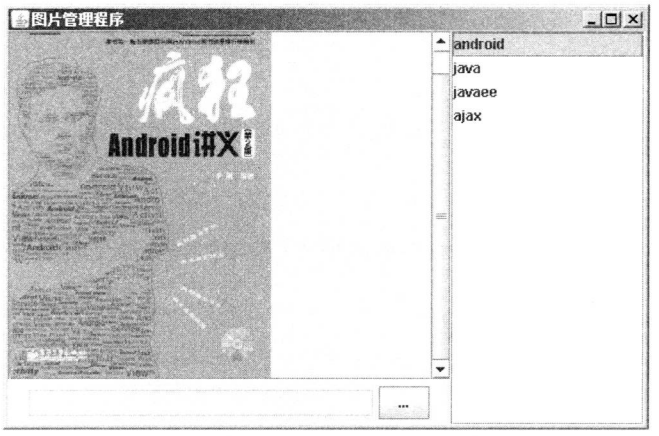


图 13.20　使用 Blob 保存图片

13.5.3　使用 ResultSetMetaData 分析结果集

当执行 SQL 查询后可以通过移动记录指针来遍历 ResultSet 的每条记录，但程序可能不清楚该 ResultSet 里包含哪些数据列，以及每个数据列的数据类型，那么可以通过 ResultSetMetaData 来获取关于 ResultSet 的描述信息。

提示：

MetaData 的意思是元数据，即描述其他数据的数据，因此 ResultSetMetaData 封装了描述 ResultSet 对象的数据；后面还要介绍的 DatabaseMetaData 则封装了描述 Database 的数据。

ResultSet 里包含一个 getMetaData()方法，该方法返回该 ResultSet 对应的 ResultSetMetaData 对象。一旦获得了 ResultSetMetaData 对象，就可通过 ResultSetMetaData 提供的大量方法来返回 ResultSet 的描述信息。常用的方法有如下三个。

- int getColumnCount()：返回该 ResultSet 的列数量。
- String getColumnName(int column)：返回指定索引的列名。
- int getColumnType(int column)：返回指定索引的列类型。

下面是一个简单的查询执行器，当用户在文本框内输入合法的查询语句并执行成功后，下面的表格将会显示查询结果。

程序清单：codes\13\13.5\QueryExecutor.java

```
public class QueryExecutor
{
    JFrame jf = new JFrame("查询执行器");
    private JScrollPane scrollPane;
    private JButton execBn = new JButton("查询");
    // 用于输入查询语句的文本框
    private JTextField sqlField = new JTextField(45);
    private static Connection conn;
    private static Statement stmt;
    // 采用静态初始化块来初始化 Connection、Statement 对象
    static
    {
        try
        {
            Properties props = new Properties();
            props.load(new FileInputStream("mysql.ini"));
            String drivers = props.getProperty("driver");
            String url = props.getProperty("url");
            String username = props.getProperty("user");
            String password = props.getProperty("pass");
```

```
            // 加载数据库驱动
            Class.forName(drivers);
            // 取得数据库连接
            conn = DriverManager.getConnection(url, username, password);
            stmt = conn.createStatement();
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
    // --------初始化界面的方法---------
    public void init()
    {
        JPanel top = new JPanel();
        top.add(new JLabel("输入查询语句："));
        top.add(sqlField);
        top.add(execBn);
        // 为执行按钮、单行文本框添加事件监听器
        execBn.addActionListener(new ExceListener());
        sqlField.addActionListener(new ExceListener());
        jf.add(top , BorderLayout.NORTH);
        jf.setSize(680, 480);
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setVisible(true);
    }
    // 定义监听器
    class ExceListener implements ActionListener
    {
        public void actionPerformed(ActionEvent evt)
        {
            // 删除原来的 JTable(JTable 使用 scrollPane 来包装)
            if (scrollPane != null)
            {
                jf.remove(scrollPane);
            }
            try(
                // 根据用户输入的 SQL 执行查询
                ResultSet rs = stmt.executeQuery(sqlField.getText()))
            {
                // 取出 ResultSet 的 MetaData
                ResultSetMetaData rsmd = rs.getMetaData();
                Vector<String> columnNames =  new Vector<>();
                Vector<Vector<String>> data = new Vector<>();
                // 把 ResultSet 的所有列名添加到 Vector 里
                for (int i = 0 ; i < rsmd.getColumnCount(); i++ )
                {
                    columnNames.add(rsmd.getColumnName(i + 1));
                }
                // 把 ResultSet 的所有记录添加到 Vector 里
                while (rs.next())
                {
                    Vector<String> v = new Vector<>();
                    for (int i = 0 ; i < rsmd.getColumnCount(); i++ )
                    {
                        v.add(rs.getString(i + 1));
                    }
                    data.add(v);
                }
                // 创建新的 JTable
                JTable table = new JTable(data , columnNames);
                scrollPane = new JScrollPane(table);
```

```
                // 添加新的 Table
                jf.add(scrollPane);
                // 更新主窗口
                jf.validate();
            }
            catch (Exception e)
            {
                e.printStackTrace();
            }
        }
    }
    public static void main(String[] args)
    {
        new QueryExecutor().init();
    }
}
```

上面程序中的粗体字代码就是根据 ResultSetMetaData 分析 ResultSet 的关键代码，使用 ResultSetMetaData 查询 ResultSet 包含多少列，并把所有数据列的列名添加到一个 Vector 里，然后把 ResultSet 里的所有数据添加到 Vector 里，并使用这两个 Vector 来创建新的 TableModel，再利用该 TableModel 生成一个新的 JTable，最后将该 JTable 显示出来。运行上面程序，会看到如图 13.21 所示的窗口。

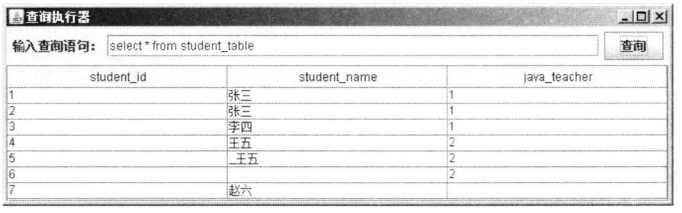


图 13.21　使用 ResultSetMetaData 分析 ResultSet

注意：

虽然 ResultSetMetaData 可以准确地分析出 ResultSet 里包含多少列，以及每列的列名、数据类型等，但使用 ResultSetMetaData 需要一定的系统开销，因此如果在编程过程中已经知道 ResultSet 里包含多少列，以及每列的列名、类型等信息，就没有必要使用 ResultSetMetaData 来分析该 ResultSet 对象了。

13.6　Javar 的 RowSet

RowSet 接口继承了 ResultSet 接口，RowSet 接口下包含 JdbcRowSet、CachedRowSet、FilteredRowSet、JoinRowSet 和 WebRowSet 常用子接口。除 JdbcRowSet 需要保持与数据库的连接之外，其余 4 个子接口都是离线的 RowSet，无须保持与数据库的连接。

与 ResultSet 相比，RowSet 默认是可滚动、可更新、可序列化的结果集，而且作为 JavaBean 使用，因此能方便地在网络上传输，用于同步两端的数据。对于离线 RowSet 而言，程序在创建 RowSet 时已把数据从底层数据库读取到了内存，因此可以充分利用计算机的内存，从而降低数据库服务器的负载，提高程序性能。

提示：

当年 C#提供了 DataSet，它可以把底层的数据读取到内存中进行离线操作，操作完成后再同步到底层数据源。Java 则提供了与此功能类似的 RowSet。

图 13.22 显示了 RowSet 规范的接口类图。

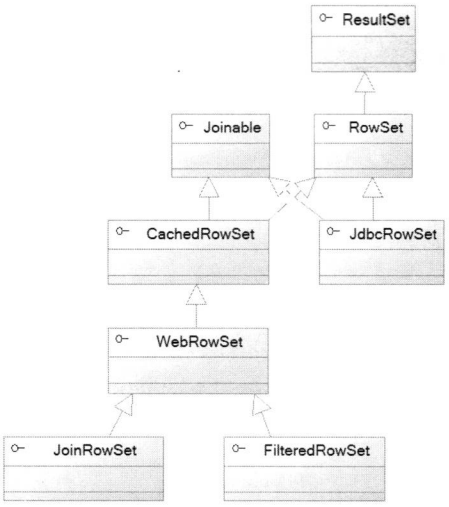


图 13.22 RowSet 规范的接口类图

在图 13.22 所示的各种接口中，CachedRowSet 及其子接口都代表了离线 RowSet，它们都不需要底层数据库连接。

13.6.1 Java 7 新增的 RowSetFactory 与 RowSet

Java 7 新增了 RowSetProvider 类和 RowSetFactory 接口，其中 RowSetProvider 负责创建 RowSetFactory，而 RowSetFactory 则提供了如下方法来创建 RowSet 实例。

- CachedRowSet createCachedRowSet()：创建一个默认的 CachedRowSet。
- FilteredRowSet createFilteredRowSet()：创建一个默认的 FilteredRowSet。
- JdbcRowSet createJdbcRowSet()：创建一个默认的 JdbcRowSet。
- JoinRowSet createJoinRowSet()：创建一个默认的 JoinRowSet。
- WebRowSet createWebRowSet()：创建一个默认的 WebRowSet。

通过使用 RowSetFactory，就可以把应用程序与 RowSet 实现类分离开，避免直接使用 JdbcRow SetImpl 等非公开的 API，也更有利于后期的升级、扩展。

通过 RowSetFactory 的几个工厂方法不难看出，使用 RowSetFactory 创建的 RowSet 其实并没有装填数据。

为了让 RowSet 能抓取到数据库的数据，需要为 RowSet 设置数据库的 URL、用户名、密码等连接信息。因此，RowSet 接口中定义了如下常用方法。

- setUrl(String url)：设置该 RowSet 要访问的数据库的 URL。
- setUsername(String name)：设置该 RowSet 要访问的数据库的用户名。
- setPassword(String password)：设置该 RowSet 要访问的数据库的密码。
- setCommand(String sql)：设置使用该 sql 语句的查询结果来装填该 RowSet。
- execute()：执行查询。

下面程序通过 RowSetFactory 示范了使用 JdbcRowSet 的可滚动、可修改特性。

程序清单：codes\13\13.6\RowSetFactoryTest.java

```
public class RowSetFactoryTest
{
    private String driver;
    private String url;
    private String user;
    private String pass;
    public void initParam(String paramFile)throws Exception
    {
        // 使用 Properties 类来加载属性文件
        Properties props = new Properties();
        props.load(new FileInputStream(paramFile));
```

```
        driver = props.getProperty("driver");
        url = props.getProperty("url");
        user = props.getProperty("user");
        pass = props.getProperty("pass");
    }
    public void update(String sql)throws Exception
    {
        // 加载驱动
        Class.forName(driver);
        // 使用 RowSetProvider 创建 RowSetFactory
        RowSetFactory factory = RowSetProvider.newFactory();
        try(
            // 使用 RowSetFactory 创建默认的 JdbcRowSet 实例
            JdbcRowSet jdbcRs = factory.createJdbcRowSet())
        {
            // 设置必要的连接信息
            jdbcRs.setUrl(url);
            jdbcRs.setUsername(user);
            jdbcRs.setPassword(pass);
            // 设置 SQL 查询语句
            jdbcRs.setCommand(sql);
            // 执行查询
            jdbcRs.execute();
            jdbcRs.afterLast();
            // 向前滚动结果集
            while (jdbcRs.previous())
            {
                System.out.println(jdbcRs.getString(1)
                    + "\t" + jdbcRs.getString(2)
                    + "\t" + jdbcRs.getString(3));
                if (jdbcRs.getInt("student_id") == 3)
                {
                    // 修改指定记录行
                    jdbcRs.updateString("student_name", "孙悟空");
                    jdbcRs.updateRow();
                }
            }
        }
    }
    public static void main(String[] args)throws Exception
    {
        RowSetFactoryTest jt = new RowSetFactoryTest();
        jt.initParam("mysql.ini");
        jt.update("select * from student_table");
    }
}
```

上面程序中的粗体字代码使用了 RowSetFactory 来创建 JdbcRowSet 对象，这就避免了与 JdbcRowSetImpl 实现类耦合。由于通过这种方式创建的 JdbcRowSet 还没有传入 Connection 参数，因此程序还需调用 setUrl()、setUsername()、setPassword()等方法来设置数据库连接信息。

编译、运行该程序，一切正常。JdbcRowSet 是一个可滚动、可修改的结果集，因此底层数据表中相应的记录也被修改了。

提示：

这个程序在 Java 9 上运行会提示“No suitable driver”异常，这可能和 Java 9 的版本升级有关。

13.6.2 离线 RowSet

在使用 ResultSet 的时代，程序查询得到 ResultSet 之后必须立即读取或处理它对应的记录，否则一旦 Connection 关闭，再去通过 ResultSet 读取记录就会引发异常。在这种模式下，JDBC 编程十分痛苦——假设应用程序架构被分为两层：数据访问层和视图显示层，当应用程序在数据访问层查询得到

ResultSet 之后，对 ResultSet 的处理有如下两种常见方式。

- 使用迭代访问 ResultSet 里的记录，并将这些记录转换成 Java Bean，再将多个 Java Bean 封装成一个 List 集合，也就是完成“ResultSet→Java Bean 集合”的转换。转换完成后可以关闭 Connection 等资源，然后将 Java Bean 集合传到视图显示层，视图显示层可以显示查询得到的数据。
- 直接将 ResultSet 传到视图显示层——这要求当视图显示层显示数据时，底层 Connection 必须一直处于打开状态，否则 ResultSet 无法读取记录。

第一种方式比较安全，但编程十分烦琐；第二种方式则需要 Connection 一直处于打开状态，这不仅不安全，而且对程序性能也有较大的影响。

通过使用离线 RowSet 可以十分“优雅”地处理上面的问题，离线 RowSet 会直接将底层数据读入内存中，封装成 RowSet 对象，而 RowSet 对象则完全可以当成 Java Bean 来使用。因此不仅安全，而且编程十分简单。CachedRowSet 是所有离线 RowSet 的父接口，因此下面以 CachedRowSet 为例进行介绍。看下面程序。

程序清单：codes\13\13.6\CachedRowSetTest.java

```
public class CachedRowSetTest
{
    private static String driver;
    private static String url;
    private static String user;
    private static String pass;
    public void initParam(String paramFile)throws Exception
    {
        // 使用 Properties 类来加载属性文件
        Properties props = new Properties();
        props.load(new FileInputStream(paramFile));
        driver = props.getProperty("driver");
        url = props.getProperty("url");
        user = props.getProperty("user");
        pass = props.getProperty("pass");
    }
    public CachedRowSet query(String sql)throws Exception
    {
        // 加载驱动
        Class.forName(driver);
        // 获取数据库连接
        Connection conn = DriverManager.getConnection(url , user , pass);
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery(sql);
        // 使用 RowSetProvider 创建 RowSetFactory
        RowSetFactory factory = RowSetProvider.newFactory();
        // 创建默认的 CachedRowSet 实例
        CachedRowSet cachedRs = factory.createCachedRowSet();
        // 使用 ResultSet 装填 RowSet
        cachedRs.populate(rs);    // ①
        // 关闭资源
        rs.close();
        stmt.close();
        conn.close();
        return cachedRs;
    }
    public static void main(String[] args)throws Exception
    {
        CachedRowSetTest ct = new CachedRowSetTest();
        ct.initParam("mysql.ini");
        CachedRowSet rs = ct.query("select * from student_table");
        rs.afterLast();
        // 向前滚动结果集
        while (rs.previous())
        {
            System.out.println(rs.getString(1)
                + "\t" + rs.getString(2)
                + "\t" + rs.getString(3));
```

```
            if (rs.getInt("student_id") == 3)
            {
                // 修改指定记录行
                rs.updateString("student_name", "孙悟空");
                rs.updateRow();
            }
        }
        // 重新获取数据库连接
        Connection conn = DriverManager.getConnection(url
            , user , pass);
        conn.setAutoCommit(false);
        // 把对 RowSet 所做的修改同步到底层数据库
        rs.acceptChanges(conn);
    }
}
```

上面程序中的①号粗体字代码调用了 RowSet 的 populate(ResultSet rs)方法来包装给定的 ResultSet，接下来的粗体字代码关闭了 ResultSet、Statement、Connection 等数据库资源。如果程序直接返回 ResultSet，那么这个 ResultSet 无法使用——因为底层的 Connection 已经关闭；但程序返回的是 CachedRowSet，它是一个离线 RowSet，因此程序依然可以读取、修改 RowSet 中的记录。

运行该程序，可以看到在 Connection 关闭的情况下，程序依然可以读取、修改 RowSet 里的记录。为了将程序对离线 RowSet 所做的修改同步到底层数据库，程序在调用 RowSet 的 acceptChanges()方法时必须传入 Connection。

提示：

上面程序没有使用自动关闭资源的 try 语句来关闭 Connection 等数据库资源，这只是为了让读者更明确地看到 Connection 已被关闭。在实际项目中还是推荐使用自动关闭资源的 try 语句。

13.6.3 离线 RowSet 的查询分页

由于 CachedRowSet 会将数据记录直接装载到内存中，因此如果 SQL 查询返回的记录过大，CachedRowSet 将会占用大量的内存，在某些极端的情况下，它甚至会直接导致内存溢出。

为了解决该问题，CachedRowSet 提供了分页功能。所谓分页功能就是一次只装载 ResultSet 里的某几条记录，这样就可以避免 CachedRowSet 占用内存过大的问题。

CachedRowSet 提供了如下方法来控制分页。

- populate(ResultSet rs, int startRow)：使用给定的 ResultSet 装填 RowSet，从 ResultSet 的第 startRow 条记录开始装填。
- setPageSize(int pageSize)：设置 CachedRowSet 每次返回多少条记录。
- previousPage()：在底层 ResultSet 可用的情况下，让 CachedRowSet 读取上一页记录。
- nextPage()：在底层 ResultSet 可用的情况下，让 CachedRowSet 读取下一页记录。

下面程序示范了 CachedRowSet 的分页支持。

程序清单：codes\13\13.6\CachedRowSetPage.java

```
public class CachedRowSetPage
{
    private String driver;
    private String url;
    private String user;
    private String pass;
    public void initParam(String paramFile)throws Exception
    {
        // 使用 Properties 类来加载属性文件
        Properties props = new Properties();
        props.load(new FileInputStream(paramFile));
        driver = props.getProperty("driver");
        url = props.getProperty("url");
        user = props.getProperty("user");
        pass = props.getProperty("pass");
```

```
    }
    public CachedRowSet query(String sql , int pageSize
        , int page)throws Exception
    {
        // 加载驱动
        Class.forName(driver);
        try(
            // 获取数据库连接
            Connection conn = DriverManager.getConnection(url , user , pass);
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery(sql))
        {
            // 使用 RowSetProvider 创建 RowSetFactory
            RowSetFactory factory = RowSetProvider.newFactory();
            // 创建默认的 CachedRowSet 实例
            CachedRowSet cachedRs = factory.createCachedRowSet();
            // 设置每页显示 pageSize 条记录
            cachedRs.setPageSize(pageSize);
            // 使用 ResultSet 装填 RowSet，设置从第几条记录开始
            cachedRs.populate(rs , (page - 1) * pageSize + 1);
            return cachedRs;
        }
    }
    public static void main(String[] args)throws Exception
    {
        CachedRowSetPage cp = new CachedRowSetPage();
        cp.initParam("mysql.ini");
        CachedRowSet rs = cp.query("select * from student_table" , 3 , 2);  // ①
        // 向后滚动结果集
        while (rs.next())
        {
            System.out.println(rs.getString(1)
                + "\t" + rs.getString(2)
                + "\t" + rs.getString(3));
        }
    }
}
```

上面两行粗体字代码就是使用 CachedRowSet 实现分页的关键代码。程序中①号代码显示要查询第 2 页的记录，每页显示 3 条记录。运行上面程序，可以看到程序只会显示从第 4 行到第 6 行的记录，这就实现了分页。

13.7 事务处理

对于任何数据库应用而言，事务都是非常重要的，事务是保证底层数据完整的重要手段，没有事务支持的数据库应用，那将非常脆弱。

13.7.1 事务的概念和 MySQL 事务支持

事务是由一步或几步数据库操作序列组成的逻辑执行单元，这系列操作要么全部执行，要么全部放弃执行。程序和事务是两个不同的概念。一般而言，一段程序中可能包含多个事务。

事务具备 4 个特性：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）和持续性（Durability）。这 4 个特性也简称为 ACID 性。

- 原子性（Atomicity）：事务是应用中最小的执行单位，就如原子是自然界的最小颗粒，具有不可再分的特征一样，事务是应用中不可再分的最小逻辑执行体。
- 一致性（Consistency）：事务执行的结果，必须使数据库从一个一致性状态，变到另一个一致性状态。当数据库只包含事务成功提交的结果时，数据库处于一致性状态。如果系统运行发生中断，某个事务尚未完成而被迫中断，而该未完成的事务对数据库所做的修改已被写入数据库，此时，数据库就处于一种不正确的状态。比如银行在两个账户之间转账：从 A 账户向 B 账户转入 1000 元，系统先减少 A 账户的 1000 元，然后再为 B 账户增加 1000 元。如果全部执行成功，

数据库处于一致性状态；如果仅执行完 A 账户金额的修改，而没有增加 B 账户的金额，则数据库就处于不一致性状态；因此，一致性是通过原子性来保证的。

- 隔离性（Isolation）：各个事务的执行互不干扰，任意一个事务的内部操作对其他并发的事务都是隔离的。也就是说，并发执行的事务之间不能看到对方的中间状态，并发执行的事务之间不能互相影响。
- 持续性（Durability）：持续性也称为持久性（Persistence），指事务一旦提交，对数据所做的任何改变都要记录到永久存储器中，通常就是保存进物理数据库。

数据库的事务由下列语句组成。

- 一组 DML 语句，经过这组 DML 语句修改后的数据将保持较好的一致性。
- 一条 DDL 语句。
- 一条 DCL 语句。

DDL 和 DCL 语句最多只能有一条，因为 DDL 和 DCL 语句都会导致事务立即提交。

当事务所包含的全部数据库操作都成功执行后，应该提交（commit）事务，使这些修改永久生效。事务提交有两种方式：显式提交和自动提交。

- 显式提交：使用 commit。
- 自动提交：执行 DDL 或 DCL 语句，或者程序正常退出。

当事务所包含的任意一个数据库操作执行失败后，应该回滚（rollback）事务，使该事务中所做的修改全部失效。事务回滚有两种方式：显式回滚和自动回滚。

- 显式回滚：使用 rollback。
- 自动回滚：系统错误或者强行退出。

MySQL 默认关闭事务（即打开自动提交），在默认情况下，用户在 MySQL 控制台输入一条 DML 语句，这条 DML 语句将会立即保存到数据库里。为了开启 MySQL 的事务支持，可以显式调用如下命令：

```
SET AUTOCOMMIT = {0 | 1} 0 为关闭自动提交，即开启事务
```

提示： 自动提交和开启事务恰好相反，如果开启自动提交就是关闭事务；关闭自动提交就是开启事务。

一旦在 MySQL 的命令行窗口中输入 set autocommit=0 开启了事务，该命令行窗口里的所有 DML 语句都不会立即生效，上一个事务结束后第一条 DML 语句将开始一个新的事务，而后续执行的所有 SQL 语句都处于该事务中，除非显式使用 commit 来提交事务，或者正常退出，或者运行 DDL、DCL 语句导致事务隐式提交。当然，也可以使用 rollback 回滚来结束事务，使用 rollback 结束事务将导致本次事务中 DML 语句所做的修改全部失效。

提示： 一个 MySQL 命令行窗口代表一次连接 Session，在该窗口里设置 set autocommit=0，相当于关闭了该连接 Session 的自动提交，对其他连接不会有任何影响，也就是对其他 MySQL 命令行窗口不会有任何影响。

除此之外，如果不想关闭整个命令行窗口的自动提交，而只是想临时性地开始事务，则可以使用 MySQL 提供的 start transaction 或 begin 两个命令，它们都表示临时性地开始一次事务，处于 start transaction 或 begin 后的 DML 语句不会立即生效，除非使用 commit 显式提交事务，或者执行 DDL、DCL 语者来隐式提交事务。如下 SQL 代码将不会对数据库有任何影响。

```
# 临时开始事务
begin;
# 向 student_table 表中插入 3 条记录
insert into student_table
values(null , 'xx' , 1);
insert into student_table
```

```
values(null , 'yy' , 1);
insert into student_table
values(null , 'zz' , 1);
# 查询 student_table 表的记录
select * from student_table;      # ①
# 回滚事务
rollback;
# 再次查询
select * from student_table;      # ②
```

执行上面 SQL 语句中的第①条查询语句将会看到刚刚插入的 3 条记录，如果打开 MySQL 的其他命令行窗口将看不到这 3 条记录——这正体现了事务的隔离性。接着程序 rollback 了事务中的全部修改，执行第②条查询语句时将看到数据库又恢复到事务开始前的状态。

提交，不管是显式提交还是隐式提交，都会结束当前事务；回滚，不管是显式回滚还是隐式回滚，都会结束当前事务。

除此之外，MySQL 还提供了 savepoint 来设置事务的中间点，通过使用 savepoint 设置事务的中间点可以让事务回滚到指定中间点，而不是回滚全部事务。如下 SQL 语句设置了一个中间点：

```
savepoint a;
```

一旦设置了中间点后，就可以使用 rollback 回滚到指定中间点，回滚到指定中间点的代码如下：

```
rollback to a;
```

注意：

普通的提交、回滚都会结束当前事务，但回滚到指定中间点因为依然处于事务之中，所以不会结束当前事务。

13.7.2 JDBC 的事务支持

JDBC 连接也提供了事务支持，JDBC 连接的事务支持由 Connection 提供，Connection 默认打开自动提交，即关闭事务，在这种情况下，每条 SQL 语句一旦执行，便会立即提交到数据库，永久生效，无法对其进行回滚操作。

可以调用 Connection 的 setAutoCommit()方法来关闭自动提交，开启事务，如下代码所示：

```
// 关闭自动提交，开启事务
conn.setAutoCommit(false);
```

程序中还可调用 Connection 提供的 getAutoCommit()方法来返回该连接的自动提交模式。

一旦事务开始之后，程序可以像平常一样创建 Statement 对象，创建了 Statement 对象之后，可以执行任意多条 DML 语句，如下代码所示：

```
stmt.executeUpdate(...);
stmt.executeUpdate(...);
stmt.executeUpdate(...);
```

上面这些 SQL 语句虽然被执行了，但这些 SQL 语句所做的修改不会生效，因为事务还没有结束。如果所有的 SQL 语句都执行成功，程序可以调用 Connection 的 commit()方法来提交事务，如下代码所示：

```
// 提交事务
conn.commit();
```

如果任意一条 SQL 语句执行失败，则应该用 Connection 的 rollback()方法来回滚事务，如下代码所示：

```
// 回滚事务
conn.rollback();
```

提示：

实际上，当 Connection 遇到一个未处理的 SQLException 异常时，系统将会非正常退出，事务也会自动回滚。但如果程序捕获了该异常，则需要在异常处理块中显式地回滚事务。

下面程序示范了当程序出现未处理的 SQLException 异常时，系统将自动回滚事务。

程序清单：codes\13\13.7\TransactionTest.java

```
public class TransactionTest
{
    private String driver;
    private String url;
    private String user;
    private String pass;
    public void initParam(String paramFile)throws Exception
    {
        // 使用 Properties 类来加载属性文件
        Properties props = new Properties();
        props.load(new FileInputStream(paramFile));
        driver = props.getProperty("driver");
        url = props.getProperty("url");
        user = props.getProperty("user");
        pass = props.getProperty("pass");
    }
    public void insertInTransaction(String[] sqls) throws Exception
    {
        // 加载驱动
        Class.forName(driver);
        try(
            Connection conn = DriverManager.getConnection(url , user , pass))
        {
            // 关闭自动提交，开启事务
            conn.setAutoCommit(false);
            try(
                // 使用 Connection 来创建一个 Statement 对象
                Statement stmt = conn.createStatement())
            {
                // 循环多次执行 SQL 语句
                for (String sql : sqls)
                {
                    stmt.executeUpdate(sql);
                }
            }
            // 提交事务
            conn.commit();
        }
    }
    public static void main(String[] args) throws Exception
    {
        TransactionTest tt = new TransactionTest();
        tt.initParam("mysql.ini");
        String[] sqls = new String[]{
            "insert into student_table values(null , 'aaa' ,1)",
            "insert into student_table values(null , 'bbb' ,1)",
            "insert into student_table values(null , 'ccc' ,1)",
            // 下面这条 SQL 语句将会违反外键约束
            // 因为 teacher_table 表中没有 ID 为 5 的记录。
            "insert into student_table values(null , 'ccc' ,5)" //①
        };
        tt.insertInTransaction(sqls);
    }
}
```

上面程序中的粗体字代码只是开启事务、提交事务的代码，并没有回滚事务的代码。但当程序执行到第 4 条 SQL 语句（①处代码）时，这条语句将会引起外键约束异常，该异常没有得到处理，引起程序非正常结束，所以事务自动回滚。

Connection 也提供了设置中间点的方法：setSavepoint()，Connection 提供了两个方法来设置中间点。

- Savepoint setSavepoint()：在当前事务中创建一个未命名的中间点，并返回代表该中间点的 Savepoint 对象。

- Savepoint setSavepoint(String name)：在当前事务中创建一个具有指定名称的中间点，并返回代表该中间点的 Savepoint 对象。

通常来说，设置中间点时没有太大的必要指定名称，因为 Connection 回滚到指定中间点时，并不是根据名字回滚的，而是根据中间点对象回滚的，Connection 提供了 rollback(Savepoint savepoint)方法回滚到指定中间点。

13.7.3 Java 8 增强的批量更新

JDBC 还提供了一个批量更新的功能，使用批量更新时，多条 SQL 语句将被作为一批操作被同时收集，并同时提交。

提示：

批量更新必须得到底层数据库的支持，可以通过调用 DatabaseMetaData 的 supportsBatchUpdates()方法来查看底层数据库是否支持批量更新。

使用批量更新也需要先创建一个 Statement 对象，然后利用该对象的 addBatch()方法将多条 SQL 语句同时收集起来，最后调用 Java 8 为 Statement 对象新增的 executeLargeBatch()（或原有的 executeBatch()）方法同时执行这些 SQL 语句。只要批量操作中任何一条 SQL 语句影响的记录条数可能超过 Integer.MAX_VALUE，就应该使用 executeLargeBatch()方法，而不是 executeBatch()方法。

如下代码片段示范了如何执行批量更新。

```
Statement stmt = conn.createStatement();
// 使用 Statement 同时收集多条 SQL 语句
stmt.addBatch(sql1);
stmt.addBatch(sql2);
stmt.addBatch(sql3);
...
// 同时执行所有的 SQL 语句
stmt.executeLargeBatch();
```

执行 executeLargeBatch()方法将返回一个 long[]数组，因为使用 Statement 执行 DDL、DML 语句都将返回一个 long 值，而执行多条 DDL、DML 语句将会返回多个 long 值，多个 long 值就组成了这个 long[]数组。如果在批量更新的 addBatch()方法中添加了 select 查询语句，程序将直接出现错误。

为了让批量操作可以正确地处理错误，必须把批量执行的操作视为单个事务，如果批量更新在执行过程中失败，则让事务回滚到批量操作开始之前的状态。为了达到这种效果，程序应该在开始批量操作之前先关闭自动提交，然后开始收集更新语句，当批量操作执行结束后，提交事务，并恢复之前的自动提交模式。如下代码示范了如何使用 JDBC 的批量更新。

```
public class BatchTest
{
    private String driver;
    private String url;
    private String user;
    private String pass;
    public void initParam(String paramFile)throws Exception
    {
        // 省略使用 Properties 类来加载属性文件
        ...
    }
    public void insertBatch(String[] sqls) throws Exception
    {
        // 加载驱动
        Class.forName(driver);
        try(
            Connection conn = DriverManager.getConnection(url , user , pass))
        {
            // 关闭自动提交，开启事务
            conn.setAutoCommit(false);
            // 保存当前的自动提交模式
            boolean autoCommit = conn.getAutoCommit();
            // 关闭自动提交
```

```
            conn.setAutoCommit(false);
            try(
                // 使用Connection来创建一个Statement对象
                Statement stmt = conn.createStatement())
            {
                // 循环多次执行SQL语句
                for (String sql : sqls)
                {
                    stmt.addBatch(sql);
                }
                // 同时提交所有的SQL语句
                stmt.executeLargeBatch();
                // 提交修改
                conn.commit();
                // 恢复原有的自动提交模式
                conn.setAutoCommit(autoCommit);
            }
            // 提交事务
            conn.commit();
        }
    }
    public static void main(String[] args) throws Exception
    {
        TransactionTest tt = new TransactionTest();
        tt.initParam("mysql.ini");
        String[] sqls = new String[]{
            "insert into student_table values(null , 'aaa' ,1)",
            "insert into student_table values(null , 'bbb' ,1)",
            "insert into student_table values(null , 'ccc' ,1)",
        };
        tt.insertInTransaction(sqls);
    }
}
```

13.8 分析数据库信息

大部分时候，只需要对指定数据表进行插入（C）、查询（R）、修改（U）、删除（D）等CRUD操作；但在某些时候，程序需要动态地获取数据库的相关信息，例如数据库里的数据表信息、列信息。除此之外，如果希望在程序中动态地利用底层数据库所提供的特殊功能，则都需要动态分析数据库相关信息。

13.8.1 使用DatabaseMetaData分析数据库信息

JDBC提供了DatabaseMetaData来封装数据库连接对应数据库的信息，通过Connection提供的getMetaData()方法就可以获取数据库对应的DatabaseMetaData对象。

DatabaseMetaData接口通常由驱动程序供应商提供实现，其目的是让用户了解底层数据库的相关信息。使用该接口的目的是发现如何处理底层数据库，尤其是对于试图与多个数据库一起使用的应用程序——因为应用程序需要在多个数据库之间切换，所以必须利用该接口来找出底层数据库的功能，例如，调用supportsCorrelatedSubqueries()方法查看是否可以使用关联子查询，或者调用supportsBatchUpdates()方法查看是否可以使用批量更新。

许多DatabaseMetaData方法以ResultSet对象的形式返回查询信息，然后使用ResultSet的常规方法（如getString()和getInt()）即可从这些ResultSet对象中获取数据。如果查询的信息不可用，则将返回一个空ResultSet对象。

DatabaseMetaData的很多方法都需要传入一个xxxPattern模式字符串，这里的xxxPattern不是正则表达式，而是SQL里的模式字符串，即用百分号（%）代表任意多个字符，使用下画线（_）代表一个字符。在通常情况下，如果把该模式字符串的参数值设置为null，即表明该参数不作为过滤条件。

下面程序通过DatabaseMetaData分析了当前Connection连接对应数据库的一些基本信息，包括当前数据库包含多少数据表，存储过程，student_table表的数据列、主键、外键等信息。

程序清单：codes\13\13.8\DatabaseMetaDataTest.java

```
public class DatabaseMetaDataTest
{
    private String driver;
    private String url;
    private String user;
    private String pass;
    public void initParam(String paramFile)throws Exception
    {
        // 使用 Properties 类来加载属性文件
        Properties props = new Properties();
        props.load(new FileInputStream(paramFile));
        driver = props.getProperty("driver");
        url = props.getProperty("url");
        user = props.getProperty("user");
        pass = props.getProperty("pass");
    }
    public void info() throws Exception
    {
        // 加载驱动
        Class.forName(driver);
        try(
            // 获取数据库连接
            Connection conn = DriverManager.getConnection(url
                , user , pass))
        {
            // 获取 DatabaseMetaData 对象
            DatabaseMetaData dbmd = conn.getMetaData();
            // 获取 MySQL 支持的所有表类型
            ResultSet rs = dbmd.getTableTypes();
            System.out.println("--MySQL 支持的表类型信息--");
            printResultSet(rs);
            // 获取当前数据库的全部数据表
            rs = dbmd.getTables(null,null, "%" , new String[]{"TABLE"});
            System.out.println("--当前数据库里的数据表信息--");
            printResultSet(rs);
            // 获取 student_table 表的主键
            rs = dbmd.getPrimaryKeys(null , null, "student_table");
            System.out.println("--student_table 表的主键信息--");
            printResultSet(rs);
            // 获取当前数据库的全部存储过程
            rs = dbmd.getProcedures(null , null, "%");
            System.out.println("--当前数据库里的存储过程信息--");
            printResultSet(rs);
            // 获取 teacher_table 表和 student_table 表之间的外键约束
            rs = dbmd.getCrossReference(null,null, "teacher_table"
                , null, null, "student_table");
            System.out.println("--teacher_table 表和 student_table 表之间"
                + "的外键约束--");
            printResultSet(rs);
            // 获取 student_table 表的全部数据列
            rs = dbmd.getColumns(null, null, "student_table", "%");
            System.out.println("--student_table 表的全部数据列--");
            printResultSet(rs);
        }
    }
    public void printResultSet(ResultSet rs)throws SQLException
    {
        ResultSetMetaData rsmd = rs.getMetaData();
        // 打印 ResultSet 的所有列标题
        for (int i = 0 ; i < rsmd.getColumnCount() ; i++ )
        {
            System.out.print(rsmd.getColumnName(i + 1) + "\t");
        }
        System.out.print("\n");
        // 打印 ResultSet 里的全部数据
        while (rs.next())
```

```
        {
            for (int i = 0; i < rsmd.getColumnCount() ; i++ )
            {
                System.out.print(rs.getString(i + 1) + "\t");
            }
            System.out.print("\n");
        }
        rs.close();
    }
    public static void main(String[] args)
        throws Exception
    {
        DatabaseMetaDataTest dt = new DatabaseMetaDataTest();
        dt.initParam("mysql.ini");
        dt.info();
    }
}
```

上面程序中的粗体字代码就是使用 DatabaseMetaData 分析数据库信息的示例代码。运行上面程序，将可以看到通过 DatabaseMetaData 分析数据库信息的结果。

13.8.2 使用系统表分析数据库信息

除可以使用 DatabaseMetaData 来分析底层数据库信息之外，如果已经确定应用程序所使用的数据库系统，则可以通过数据库的系统表来分析数据库信息。前面已经提到，系统表又称为数据字典，数据字典的数据通常由数据库系统负责维护，用户通常只能查询数据字典，而不能修改数据字典的内容。

提示：

几乎所有的数据库都会提供系统表供用户查询，用户可以通过查询系统表来获得数据库的相关信息。对于像 MySQL 和 SQL Server 这样的数据库，它们还提供一个系统数据库来存储这些系统表。系统表相当于视图，用户只能查看系统表的数据，不能直接修改系统表中的数据。

MySQL 数据库使用 information_schema 数据库来保存系统表，在该数据库里包含了大量系统表，常用系统表的简单介绍如下。

- tables：存放数据库里所有数据表的信息。
- schemata：存放数据库里所有数据库（与 MySQL 的 Schema 对应）的信息。
- views：存放数据库里所有视图的信息。
- columns：存放数据库里所有列的信息。
- triggers：存放数据库里所有触发器的信息。
- routines：存放数据库里所有存储过程和函数的信息。
- key_column_usage：存放数据库里所有具有约束的键信息。
- table_constraints：存放数据库里全部约束的表信息。
- statistics：存放数据库里全部索引的信息。

从这些系统表中取得的数据库信息会更加准确，例如，若要查询当前 MySQL 数据库中包含多少数据库及其详细信息，则可以查询 schemata 系统表；如果需要查询指定数据库中的全部数据表，则可以查询 tables 系统表；如果需要查询指定数据表的全部数据列，就可以查询 columns 系统表。图 13.23 显示了通过系统表查询所有的数据库、select_test 数据库的全部数据表、student_table 表的所有数据列的 SQL 语句及执行效果。

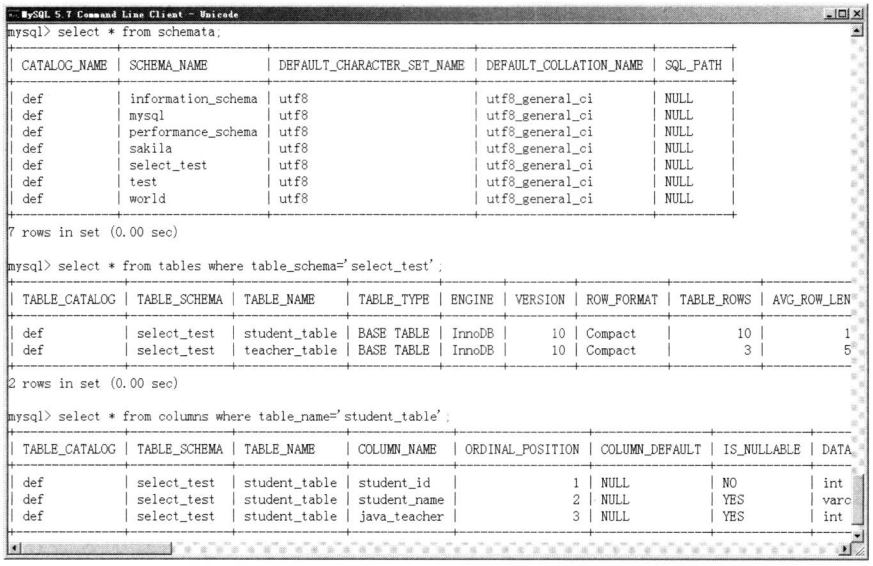


图 13.23　使用系统表分析数据库信息

13.8.3　选择合适的分析方式

本章后面的练习需要完成一个仿 SQLyog 应用程序，这个应用程序需要根据数据库、表、列等信息创建一棵树，这就需要利用 DatabaseMetaData 来分析数据库信息，或者利用 MySQL 系统表来分析数据库信息。

通常而言，如果使用 DatabaseMetaData 来分析数据库信息，则具有更好的跨数据库特性，应用程序可以做到数据库无关；但可能无法准确获得数据库的更多细节。

使用数据库系统表来分析数据库系统信息会更加准确，但使用系统表也有坏处——这种方式与底层数据库耦合严重，采用这种方式将会导致程序只能运行于特定的数据库之上。

通常来说，如果需要获得数据库信息，包括该数据库驱动提供了哪些功能，则应该利用 DatabaseMetaData 来了解该数据库支持哪些功能。完全可能出现这样一种情况：对于底层数据库支持的功能，但数据库驱动没有提供该功能，程序还是不能使用该功能。使用 DatabaseMetaData 则不会出现这种问题。

如果需要纯粹地分析数据库的静态对象，例如分析数据库系统里包含多少数据库、数据表、视图、索引等信息，则利用系统表会更加合适。

提示：

如果希望利用系统表时具有更好的通用性，程序可以通过 DatabaseMetaData 的 getDatabaseProductName()、getDatabaseProductVersion()方法来获取底层数据库的产品名、产品版本号，还可以通过 DatabaseMetaData 的 getDriverName()和 getDriverVersion()方法获取驱动程序名和驱动程序版本号。

13.9　使用连接池管理连接

数据库连接的建立及关闭是极耗费系统资源的操作，在多层结构的应用环境中，这种资源的耗费对系统性能影响尤为明显。通过前面介绍的方式（通过 DriverManager 获取连接）获得的数据库连接，一个数据库连接对象均对应一个物理数据库连接，每次操作都打开一个物理连接，使用完后立即关闭连接。频繁地打开、关闭连接将造成系统性能低下。

数据库连接池的解决方案是：当应用程序启动时，系统主动建立足够的数据库连接，并将这些连接组成一个连接池。每次应用程序请求数据库连接时，无须重新打开连接，而是从连接池中取出已有的连接使用，使用完后不再关闭数据库连接，而是直接将连接归还给连接池。通过使用连接池，将大大提高程序的运行效率。

对于共享资源的情况，有一个通用的设计模式：资源池（Resource Pool），用于解决资源的频繁请求、释放所造成的性能下降。为了解决数据库连接的频繁请求、释放，JDBC 2.0 规范引入了数据库连接池技术。数据库连接池是 Connection 对象的工厂。数据库连接池的常用参数如下。

- 数据库的初始连接数。
- 连接池的最大连接数。
- 连接池的最小连接数。
- 连接池每次增加的容量。

JDBC 的数据库连接池使用 javax.sql.DataSource 来表示，DataSource 只是一个接口，该接口通常由商用服务器（如 WebLogic、WebSphere）等提供实现，也有一些开源组织提供实现（如 DBCP 和 C3P0 等）。

提示：

DataSource 通常被称为数据源，它包含连接池和连接池管理两个部分，但习惯上也经常把 DataSource 称为连接池。

本节不打算介绍任何商用服务器的数据源实现，主要介绍 DBCP 和 C3P0 两种开源的数据源实现。

13.9.1 DBCP 数据源

DBCP 是 Apache 软件基金组织下的开源连接池实现，该连接池依赖该组织下的另一个开源系统：common-pool。如果需要使用该连接池实现，则应在系统中增加如下两个 jar 文件。

- commons-dbcp.jar：连接池的实现。
- commons-pool.jar：连接池实现的依赖库。

登录 http://commons.apache.org/站点即可下载 commons-pool.zip 和 commons-dbcp.zip 两个压缩文件，解压缩这两个文件即可得到上面提到的两个 JAR 文件。为了在程序中使用这两个 JAR 文件，应该把它们添加到系统的类加载路径中（比如添加到 CLASSPATH 环境变量中）。

Tomcat 的连接池正是采用该连接池实现的。数据库连接池既可以与应用服务器整合使用，也可以由应用程序独立使用。下面的代码片段示范了使用 DBCP 来获得数据库连接的方式。

```
// 创建数据源对象
BasicDataSource ds = new BasicDataSource();
// 设置连接池所需的驱动
ds.setDriverClassName("com.mysql.jdbc.Driver");
// 设置连接数据库的 URL
ds.setUrl("jdbc:mysql://localhost:3306/javaee");
// 设置连接数据库的用户名
ds.setUsername("root");
// 设置连接数据库的密码
ds.setPassword("pass");
// 设置连接池的初始连接数
ds.setInitialSize(5);
// 设置连接池最多可有多少个活动连接数
ds.setMaxActive(20);
// 设置连接池中最少有 2 个空闲的连接
ds.setMinIdle(2)
```

数据源和数据库连接不同，数据源无须创建多个，它是产生数据库连接的工厂，因此整个应用只需要一个数据源即可。也就是说，对于一个应用，上面代码只要执行一次即可。建议把上面程序中的 ds 设置成 static 成员变量，并且在应用开始时立即初始化数据源对象，程序中所有需要获取数据库连接的地方直接访问该 ds 对象，并获取数据库连接即可。通过 DataSource 获取数据库连接的代码示例如下：

```
// 通过数据源获取数据库连接
Connection conn = ds.getConnection();
```

当数据库访问结束后，程序还是像以前一样关闭数据库连接，如下代码所示：

```
// 释放数据库连接
conn.close();
```

但上面代码并没有关闭数据库的物理连接，它仅仅把数据库连接释放，归还给连接池，让其他客户端可以使用该连接。

13.9.2 C3P0 数据源

相比之下，C3P0 数据源性能更胜一筹，Hibernate 就推荐使用该连接池。C3P0 连接池不仅可以自动清理不再使用的 Connection，还可以自动清理 Statement 和 ResultSet。C3P0 连接池需要版本为 1.3 以上的 JRE，推荐使用 1.4 以上的 JRE。如果需要使用 C3P0 连接池，则应在系统中增加如下 JAR 文件。

- c3p0-0.9.1.2.jar：C3P0 连接池的实现。

登录 http://sourceforge.net/projects/c3p0/站点即可下载 C3P0 数据源的最新版本，下载后得到一个 c3p0-0.9.1.2.bin.zip 文件（版本号可能有区别），解压缩该文件，即可得到上面提到的 JAR 文件。

下面代码通过 C3P0 连接池获得数据库连接。

```
// 创建连接池实例
ComboPooledDataSource ds = new ComboPooledDataSource();
// 设置连接池连接数据库所需的驱动
ds.setDriverClass("com.mysql.jdbc.Driver");
// 设置连接数据库的 URL
ds.setJdbcUrl("jdbc:mysql://localhost:3306/javaee");
// 设置连接数据库的用户名
ds.setUser("root");
// 设置连接数据库的密码
ds.setPassword("32147");
// 设置连接池的最大连接数
ds.setMaxPoolSize(40);
// 设置连接池的最小连接数
ds.setMinPoolSize(2);
// 设置连接池的初始连接数
ds. setInitialPoolSize(10);
// 设置连接池的缓存 Statement 的最大数
ds.setMaxStatements(180);
```

在程序中创建 C3P0 连接池的方法与前面介绍的创建 DBCP 连接池的方法基本类似，此处不再解释。一旦获取了 C3P0 连接池之后，程序同样可以通过如下代码来获取数据库连接。

```
// 获得数据库连接
Connection conn = ds.getConnection();
```

13.10 本章小结

本章从标准的 SQL 语句讲起，简单介绍了关系数据库的基本理论及标准的 SQL 语句的相关语法，包括 DDL、DML、简单查询语句、多表连接查询和子查询语句。本章重点讲解了 JDBC 数据库访问的详细步骤，包括加载数据库驱动，获取数据库连接，执行 SQL 语句，处理执行结果等。

本章在介绍 JDBC 数据库访问时详细讲解了 Statement、PreparedStatement、CallableStatement 的区别和联系，并介绍了如何处理数据表的 Blob 列。本章还介绍了事务相关知识，包括如何在标准的 SQL 语句中进行事务控制和在 JDBC 编程中进行事务控制。本章最后介绍了如何利用 DatabaseMetaData、系统表来分析数据库信息，并讲解了数据源的原理和作用，示范了两个开源数据源实现的用法。

本章练习

1. 设计一个数据表用于保存图书信息，需要保存图书的书名、价格、作者、出版社、封面（图片）等信息。开发一个带界面的程序，用户可向该数据表中添加记录、删除记录，也可修改已有的图书记录，并可根据书名、价格、作者等条件查询图书。

2. 开发 C/S 结构的图书销售管理系统，要求实现两个模块：① 后台管理，包括管理种类、管理图书库存（可以上传图书封面图片）、出版社管理；② 销售前台，包括查询图书资料（根据种类、书名、出版社）、销售图书（会影响库存），并记录每条销售信息，统计每天、每月的销售情况。

3. 开发 MySQL 企业管理器，功能类似于 SQLyog。

CHAPTER

14

第14章 注解（Annotation）

本章要点

- 注解的概念和作用
- @Override 注解的功能和用法
- @Deprecated 注解的功能和用法
- @SuppressWarnings 注解的功能和用法
- @Retention 注解的功能和用法
- @Target 注解的功能和用法
- @Documented 注解的功能和用法
- @Inherited 注解的功能和用法
- 自定义注解
- 提取注解信息
- 重复注解
- 类型注解
- 使用 APT 工具

从 JDK 5 开始，Java 增加了对元数据（MetaData）的支持，也就是 Annotation（即注解，也被翻译为注释），这种注解与第 3 章所介绍的注释有一定的区别。本章所介绍的注解，其实是代码里的特殊标记，这些标记可以在编译、类加载、运行时被读取，并执行相应的处理。通过使用注解，程序开发人员可以在不改变原有逻辑的情况下，在源文件中嵌入一些补充的信息。代码分析工具、开发工具和部署工具可以通过这些补充信息进行验证或者进行部署。

注解提供了一种为程序元素设置元数据的方法，从某些方面来看，注解就像修饰符一样，可用于修饰包、类、构造器、方法、成员变量、参数、局部变量的声明，这些信息被存储在注解的“name=value”对中。

注意：

注解是一个接口，程序可以通过反射来获取指定程序元素的 java.lang.annotation.Annotation 对象，然后通过 java.lang.annotation.Annotation 对象来取得注解里的元数据。

注解能被用来为程序元素（类、方法、成员变量等）设置元数据。值得指出的是，注解不影响程序代码的执行，无论增加、删除注解，代码都始终如一地执行。如果希望让程序中的注解在运行时起一定的作用，只有通过某种配套的工具对注解中的信息进行访问和处理，访问和处理注解的工具统称 APT（Annotation Processing Tool）。

14.1 基本注解

注解必须使用工具来处理，工具负责提取注解里包含的元数据，工具还会根据这些元数据增加额外的功能。在系统学习新的注解语法之前，先看一下 Java 提供的 5 个基本注解的用法——使用注解时要在其前面增加@符号，并把该注解当成一个修饰符使用，用于修饰它支持的程序元素。

5 个基本的注解如下：

- @Override
- @Deprecated
- @SuppressWarnings
- @SafeVarargs
- @FunctionalInterface

上面 5 个基本注解中的@SafeVarargs 是 Java 7 新增的、@FunctionalInterface 是 Java 8 新增的。这 5 个基本的注解都定义在 java.lang 包下，读者可以通过查阅它们的 API 文档来了解关于它们的更多细节。

14.1.1 限定重写父类方法：@Override

@Override 就是用来指定方法覆载的，它可以强制一个子类必须覆盖父类的方法。如下程序中使用@Override 指定子类 Apple 的 info()方法必须重写父类方法。

程序清单：codes\14\14.1\Fruit.java

```
public class Fruit
{
    public void info()
    {
        System.out.println("水果的 info 方法...");
    }
}
class Apple extends Fruit
{
    // 使用@Override 指定下面方法必须重写父类方法
    @Override
    public void info()
    {
        System.out.println("苹果重写水果的 info 方法...");
    }
}
```

编译上面程序，可能丝毫看不出程.序中的@Override 有何作用，因为@Override 的作用是告诉编译器检查这个方法，保证父类要包含一个被该方法重写的方法，否则就会编译出错。@Override 主要是帮助程序员避免一些低级错误，例如把上面 Apple 类中的 info 方法不小心写成了 inf0，这样的“低级错误”可能会成为后期排错时的巨大障碍。

提示：

疯狂软件教育中心在讲解 Struts 2.x 框架过程中会告诉学员定义 Action 的方法：需要继承系统的 Action 基类，并重写 execute()方法，但由于 Struts Action 基类里包含的 execute()方法比较复杂，经常有学员出现重写 execute()方法时方法签名写错的错误——这种错误在编译、运行时都没有任何提示，只是运行时不出现所期望的结果，这种没有任何错误提示的错误才是最难调试的错误。如果在重写 execute()方法时使用了@Override 修饰，就可以轻松避免这个问题。

如果把 Apple 类中的 info 方法误写成 inf0，编译程序时将出现如下错误提示：

```
Fruit.java:23: 错误: 方法不会覆盖或实现超类型的方法
    @Override
    ^
1 个错误
```

注意：

@Override 只能修饰方法，不能修饰其他程序元素。

14.1.2 Java 9 增强的@Deprecated

@Deprecated 用于表示某个程序元素（类、方法等）已过时，当其他程序使用已过时的类、方法时，编译器将会给出警告。如下程序指定 Apple 类中的 info()方法已过时，其他程序中使用 Apple 类的 info()方法时编译器将会给出警告。

Java 9 为@Deprecated 注解增加了如下两个属性。

- forRemoval：该 boolean 类型的属性指定该 API 在将来是否会被删除。
- since：该 String 类型的属性指定该 API 从哪个版本被标记为过时。

程序清单：codes\14\14.1\DeprecatedTest.java

```
class Apple
{
    // 定义 info 方法已过时
    // since 属性指定从哪个版本开始，forRemoval 指定该 API 将来会被删除
    @Deprecated(since="9", forRemoval=true)    public void info()
    {
        System.out.println("Apple 的 info 方法");
    }
}
public class DeprecatedTest
{
    public static void main(String[] args)
    {
        // 下面使用 info()方法时将会被编译器警告
        new Apple().info();
    }
}
```

上面程序中的粗体字代码使用了 Apple 的 info()方法，而 Apple 类中定义 info()方法时使用了@Deprecated 修饰，表明该方法已过时，所以将会引起编译器警告。

注意：

@Deprecated 的作用与文档注释中的@deprecated 标记的作用基本相同，但它们的用法不同，前者是 JDK 5 才支持的注解，无须放在文档注释语法（/**...*/部分）中，而是直接用于修饰程序中的程序单元，如方法、类、接口等。

14.1.3 抑制编译器警告：@SuppressWarnings

@SuppressWarnings 指示被该注解修饰的程序元素（以及该程序元素中的所有子元素）取消显示指定的编译器警告。@SuppressWarnings 会一直作用于该程序元素的所有子元素，例如，使用@SuppressWarnings 修饰某个类取消显示某个编译器警告，同时又修饰该类里的某个方法取消显示另一个编译器警告，那么该方法将会同时取消显示这两个编译器警告。

在通常情况下，如果程序中使用没有泛型限制的集合将会引起编译器警告，为了避免这种编译器警告，可以使用@SuppressWarnings 修饰。下面程序取消了没有使用泛型的编译器警告。

程序清单：codes\14\14.1\SuppressWarningsTest.java

```
// 关闭整个类里的编译器警告
@SuppressWarnings(value="unchecked")
public class SuppressWarningsTest
{
    public static void main(String[] args)
    {
        List<String> myList = new ArrayList();     // ①
    }
}
```

程序中的粗体字代码使用@SuppressWarnings 来关闭 SuppressWarningsTest 类里的所有编译器警告，编译上面程序时将不会看到任何编译器警告。如果删除程序中的粗体字代码，将会在程序的①处看到编译器警告。

正如从程序中粗体字代码所看到的，当使用@SuppressWarnings 注解来关闭编译器警告时，一定要在括号里使用 name=value 的形式为该注解的成员变量设置值。关于如何为注解添加成员变量请看下一节介绍。

14.1.4 “堆污染”警告与 Java 9 增强的@SafeVarargs

前面介绍泛型擦除时，介绍了如下代码可能导致运行时异常。

```
List list = new ArrayList<Integer>();
list.add(20);      // 添加元素时引发 unchecked 异常
// 下面代码引起“未经检查的转换”的警告，编译、运行时完全正常
List<String> ls = list;     // ①
// 但只要访问 ls 里的元素，如下面代码就会引起运行时异常
System.out.println(ls.get(0));
```

Java 把引发这种错误的原因称为“堆污染”（Heap pollution），当把一个不带泛型的对象赋给一个带泛型的变量时，往往就会发生这种“堆污染”，如上①号粗体字代码所示。

对于形参个数可变的方法，该形参的类型又是泛型，这将更容易导致“堆污染”。例如如下工具类。

程序清单：codes\14\14.1\ErrorUtils.java

```
public class ErrorUtils
{
    public static void faultyMethod(List<String>... listStrArray)
    {
        // Java 语言不允许创建泛型数组，因此 listArray 只能被当成 List[]处理
        // 此时相当于把 List<String>赋给了 List，已经发生了“堆污染”
        List[] listArray = listStrArray;
        List<Integer> myList = new ArrayList<Integer>();
        myList.add(new Random().nextInt(100));
        // 把 listArray 的第一个元素赋为 myArray
        listArray[0] = myList;
        String s = listStrArray[0].get(0);
    }
}
```

上面程序中的粗体字代码已经发生了“堆污染”。由于该方法有个形参是 List<String>...类型，个数可变的形参相当于数组，但 Java 又不支持泛型数组，因此程序只能把 List<String>...当成 List[]处理，这

里就发生了“堆污染”。

在 Java 6 以及更早的版本中，Java 编译器认为 faultyMethod()方法完全没有问题，既不会提示错误，也没有提示警告。

等到使用该方法时，例如如下程序。

程序清单：codes\14\14.1\ErrorUtilsTest.java

```
public class ErrorUtilsTest
{
    public static void main(String[] args)
    {
        ErrorUtils.faultyMethod(Arrays.asList("Hello!")
            , Arrays.asList("World!"));        // ①
    }
}
```

编译该程序将会在①号代码处引发一个 unchecked 警告。这个 unchecked 警告出现得比较“突兀”：定义 faultyMethod()方法时没有任何警告，调用该方法时却引发了一个“警告”。

注意：

上面程序故意利用了“堆污染”，因此程序运行时也会在①号代码处引发 ClassCastException 异常。

从 Java 7 开始，Java 编译器将会进行更严格的检查，Java 编译器在编译 ErrorUtils 时就会发出一个如下所示的警告。

```
ErrorUtils.java:15: 警告: [unchecked] 参数化 vararg 类型 List<String>的堆可能已受污染
    public static void faultyMethod(List<String>... listStrArray)
                                                ^
1 个警告
```

由此可见，Java 7 会在定义该方法时就发出“堆污染”警告，这样保证开发者“更早”地注意到程序中可能存在的“漏洞”。

但在有些时候，开发者不希望看到这个警告，则可以使用如下三种方式来“抑制”这个警告。

- 使用@SafeVarargs 修饰引发该警告的方法或构造器。Java 9 增强了该注解，允许使用该注解修饰私有实例方法。
- 使用@SuppressWarnings("unchecked")修饰。
- 编译时使用-Xlint:varargs 选项。

很明显，第三种方式一般比较少用，通常可以选择第一种或第二种方式，尤其是使用@SafeVarargs 修饰引发该警告的方法或构造器，它是 Java 7 专门为抑制“堆污染”警告提供的。

如果程序使用@SafeVarargs 修饰 ErrorUtils 类中的 faultyMethod()方法，则编译上面两个程序时都不会发出任何警告。

14.1.5 Java 8 的函数式接口与@FunctionalInterface

前面已经提到，Java 8 规定：如果接口中只有一个抽象方法（可以包含多个默认方法或多个 static 方法），该接口就是函数式接口。@FunctionalInterface 就是用来指定某个接口必须是函数式接口。例如，如下程序使用@FunctionalInterface 修饰了函数式接口。

提示：

函数式接口就是为 Java 8 的 Lambda 表达式准备的，Java 8 允许使用 Lambda 表达式创建函数式接口的实例，因此 Java 8 专门增加了@FunctionalInterface。

程序清单：codes\14\14.1\FunInterface.java

```
@FunctionalInterface
public interface FunInterface
```

```
{
    static void foo()
    {
        System.out.println("foo 类方法");
    }
    default void bar()
    {
        System.out.println("bar 默认方法");
    }
    void test(); // 只定义一个抽象方法
}
```

编译上面程序，可能丝毫看不出程序中的@FunctionalInterface 有何作用，因为@FunctionalInterface 只是告诉编译器检查这个接口，保证该接口只能包含一个抽象方法，否则就会编译出错。@FunctionalInterface 主要是帮助程序员避免一些低级错误，例如，在上面的 FunInterface 接口中再增加一个抽象方法 abc()，编译程序时将出现如下错误提示：

```
FunInterface.java:13: 错误: 意外的 @FunctionalInterface 注释
@FunctionalInterface
^
  FunInterface 不是函数式接口
    在 接口 FunInterface 中找到多个非覆盖抽象方法
1 个错误
```

注意：

@FunInterface 只能修饰接口，不能修饰其他程序元素。

14.2 JDK 的元注解

JDK 除在 java.lang 下提供了 5 个基本的注解之外，还在 java.lang.annotation 包下提供了 6 个 Meta 注解（元注解），其中有 5 个元注解都用于修饰其他的注解定义。其中@Repeatable 专门用于定义 Java 8 新增的重复注解，本章后面会重点介绍相关内容。此处先介绍常用的 4 个元注解。

14.2.1 使用@Retention

@Retention 只能用于修饰注解定义，用于指定被修饰的注解可以保留多长时间，@Retention 包含一个 RetentionPolicy 类型的 value 成员变量，所以使用@Retention 时必须为该 value 成员变量指定值。

value 成员变量的值只能是如下三个。

- RetentionPolicy.CLASS：编译器将把注解记录在 class 文件中。当运行 Java 程序时，JVM 不可获取注解信息。这是默认值。
- RetentionPolicy.RUNTIME：编译器将把注解记录在 class 文件中。当运行 Java 程序时，JVM 也可获取注解信息，程序可以通过反射获取该注解信息。
- RetentionPolicy.SOURCE：注解只保留在源代码中，编译器直接丢弃这种注解。

如果需要通过反射获取注解信息，就需要使用 value 属性值为 RetentionPolicy.RUNTIME 的@Retention。使用@Retention 元注解可采用如下代码为 value 指定值。

```
// 定义下面的@Testable 注解保留到运行时
@Retention(value= RetentionPolicy.RUNTIME)
public @interface Testable{}
```

也可采用如下代码来为 value 指定值。

```
// 定义下面的@Testable 注解将被编译器直接丢弃
@Retention(RetentionPolicy.SOURCE)
public @interface Testable{}
```

上面代码中使用@Retention 元注解时，并未通过 value=RetentionPolicy.SOURCE 的方式来为该成员

变量指定值，这是因为当注解的成员变量名为 value 时，程序中可以直接在注解后的括号里指定该成员变量的值，无须使用 name=value 的形式。

提示：

如果使用注解时只需要为 value 成员变量指定值，则使用该注解时可以直接在该注解后的括号里指定 value 成员变量的值，无须使用“value=变量值”的形式。

14.2.2 使用@Target

@Target 也只能修饰注解定义，它用于指定被修饰的注解能用于修饰哪些程序单元。@Target 元注解也包含一个名为 value 的成员变量，该成员变量的值只能是如下几个。

- ElementType.ANNOTATION_TYPE：指定该策略的注解只能修饰注解。
- ElementType.CONSTRUCTOR：指定该策略的注解只能修饰构造器。
- ElementType.FIELD：指定该策略的注解只能修饰成员变量。
- ElementType.LOCAL_VARIABLE：指定该策略的注解只能修饰局部变量。
- ElementType.METHOD：指定该策略的注解只能修饰方法定义。
- ElementType.PACKAGE：指定该策略的注解只能修饰包定义。
- ElementType.PARAMETER：指定该策略的注解可以修饰参数。
- ElementType.TYPE：指定该策略的注解可以修饰类、接口（包括注解类型）或枚举定义。

与使用@Retention 类似的是，使用@Target 也可以直接在括号里指定 value 值，而无须使用 name=value 的形式。如下代码指定@ActionListenerFor 注解只能修饰成员变量。

```
@Target(ElementType.FIELD)
public @interface ActionListenerFor{}
```

如下代码片段指定@Testable 注解只能修饰方法。

```
@Target(ElementType.METHOD)
public @interface Testable { }
```

14.2.3 使用@Documented

@Documented 用于指定被该元注解修饰的注解类将被 javadoc 工具提取成文档，如果定义注解类时使用了@Documented 修饰，则所有使用该注解修饰的程序元素的 API 文档中将会包含该注解说明。

下面代码定义了一个 Testable 注解，程序使用@Documented 来修饰@Testable 注解定义，所以该注解将被 javadoc 工具所提取。

程序清单：codes\14\14.2\Testable.java

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
// 定义 Testable 注解将被 javadoc 工具提取
@Documented
public @interface Testable
{
}
```

上面代码中的粗体字代码指定了 javadoc 工具生成的 API 文档将提取@Testable 的使用信息。

下面代码定义了一个 MyTest 类，该类中的 info()方法使用了@Testable 修饰。

程序清单：codes\14\14.2\MyTest.java

```
public class MyTest
{
    // 使用@Testable 修饰 info()方法
    @Testable
    public void info()
    {
        System.out.println("info 方法...");
    }
}
```

使用 javadoc 工具为 Testable.java、MyTest.java 文件生成 API 文档后的效果如图 14.1 所示。

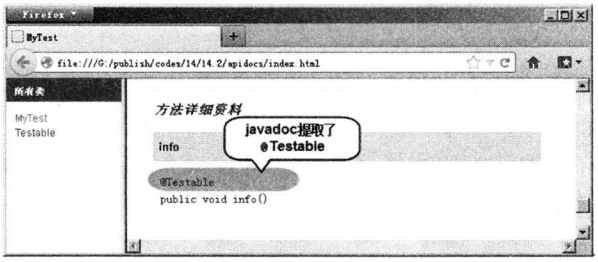


图 14.1 javadoc 提取了有@Documented 修饰的注解

如果把上面 Testable.java 程序中的粗体字代码删除或注释掉，再次使用 javadoc 工具生成的 API 文档如图 14.2 所示。

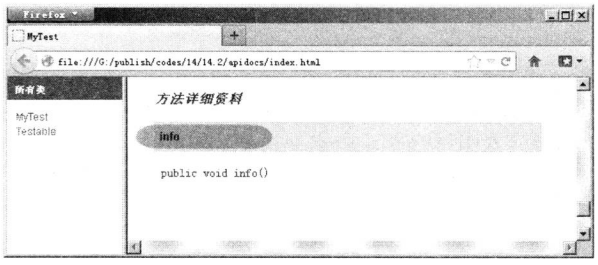


图 14.2 javadoc 不提取没有@Documented 修饰的注解

对比图 14.1 和 14.2 所示两份 API 文档中灰色区域覆盖的 info 方法说明，图 14.1 中的 info 方法说明里包含了@Testable 的信息，这就是使用@Documented 元 Annotatiom 的作用。

14.2.4 使用@Inherited

@Inherited 元注解指定被它修饰的注解将具有继承性——如果某个类使用了@Xxx 注解（定义该注解时使用了@Inherited 修饰）修饰，则其子类将自动被@Xxx 修饰。

下面使用@Inherited 元注解修饰@Inheritable 定义，则该注解将具有继承性。

程序清单：codes\14\14.2\Inheritable.java

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Inherited
public @interface Inheritable
{
}
```

上面程序中的粗体字代码表明@Inheritable 具有继承性，如果某个类使用了@Inheritable 修饰，则该类的子类将自动使用@Inheritable 修饰。

下面程序中定义了一个 Base 基类，该基类使用了@Inheritable 修饰，则 Base 类的子类将会默认使用@Inheritable 修饰。

程序清单：codes\14\14.2\InheritableTest.java

```
// 使用@Inheritable 修饰的 Base 类
@Inheritable
class Base
{
}
// InheritableTest 类只是继承了 Base 类
// 并未直接使用@Inheritable Annotiation 修饰
public class InheritableTest extends Base
{
    public static void main(String[] args)
    {
        // 打印 InheritableTest 类是否有@Inheritable 修饰
```

```
        System.out.println(InheritableTest.class
            .isAnnotationPresent(Inheritable.class));
    }
}
```

上面程序中的 Base 类使用了@Inheritable 修饰，而该注解具有继承性，所以其子类也将自动使用@Inheritable 修饰。运行上面程序，会看到输出：true。

如果将 InheritableTest.java 程序中的粗体字代码注释掉或者删除，将会导致@Inheritable 不具有继承性。运行上面程序，将看到输出：false。

14.3 自定义注解

前面已经介绍了如何使用 java.lang 包下的 4 个基本的注解，下面介绍如何自定义注解，并利用注解来完成一些实际的功能。

14.3.1 定义注解

定义新的注解类型使用@interface 关键字（在原有的 interface 关键字前增加@符号）定义一个新的注解类型与定义一个接口非常像，如下代码可定义一个简单的注解类型。

```
// 定义一个简单的注解类型
public @interface Test
{
}
```

定义了该注解之后，就可以在程序的任何地方使用该注解，使用注解的语法非常类似于 public、final 这样的修饰符，通常可用于修饰程序中的类、方法、变量、接口等定义。通常会把注解放在所有修饰符之前，而且由于使用注解时可能还需要为成员变量指定值，因而注解的长度可能较长，所以通常把注解另放一行，如下程序所示。

```
// 使用@Test 修饰类定义
@Test
public class MyClass
{
    ...
}
```

在默认情况下，注解可用于修饰任何程序元素，包括类、接口、方法等，如下程序使用@Test 来修饰方法。

```
public class MyClass
{
    // 使用@Test 注解修饰方法
    @Test
    public void info()
    {
        ...
    }
    ...
}
```

注解不仅可以是这种简单的注解，还可以带成员变量，成员变量在注解定义中以无形参的方法形式来声明，其方法名和返回值定义了该成员变量的名字和类型。如下代码可以定义一个有成员变量的注解。

```
public @interface MyTag
{
    // 定义带两个成员变量的注解
    // 注解中的成员变量以方法的形式来定义
    String name();
    int age();
}
```

可能有读者会看出，上面定义注解的代码与定义接口的语法非常像，只是 MyTag 使用@interface 关键字来定义，而接口使用 interface 来定义。

注意：

使用 @interface 定义的注解的确非常像定义了一个注解接口，这个注解接口继承了 java.lang.annotation.Annotation 接口，这一点可以通过反射看到 MyTag 接口里包含了 java.lang.annotation.Annotation 接口里的方法。

一旦在注解里定义了成员变量之后，使用该注解时就应该为它的成员变量指定值，如下代码所示。

```
public class Test
{
    // 使用带成员变量的注解时，需要为成员变量赋值
    @MyTag(name="xx", age=6)
    public void info()
    {
        ...
    }
    ...
}
```

也可以在定义注解的成员变量时为其指定初始值（默认值），指定成员变量的初始值可使用 default 关键字。如下代码定义了@MyTag 注解，该注解里包含了两个成员变量：name 和 age，这两个成员变量使用 default 指定了初始值。

```
public @interface MyTag
{
    // 定义了两个成员变量的注解
    // 使用 default 为两个成员变量指定初始值
    String name() default "yeeku";
    int age() default 32;
}
```

如果为注解的成员变量指定了默认值，使用该注解时则可以不为这些成员变量指定值，而是直接使用默认值。

```
public class Test
{
    // 使用带成员变量的注解
    // 因为它的成员变量有默认值，所以可以不为它的成员变量指定值
    @MyTag
    public void info()
    {
        ...
    }
    ...
}
```

当然也可以在使用 MyTag 注解时为成员变量指定值，如果为 MyTag 的成员变量指定了值，则默认值不会起作用。

根据注解是否可以包含成员变量，可以把注解分为如下两类。

- 标记注解：没有定义成员变量的注解类型被称为标记。这种注解仅利用自身的存在与否来提供信息，如前面介绍的@Override、@Test 等注解。
- 元数据注解：包含成员变量的注解，因为它们可以接受更多的元数据，所以也被称为元数据注解。

14.3.2 提取注解信息

使用注解修饰了类、方法、成员变量等成员之后，这些注解不会自己生效，必须由开发者提供相应的工具来提取并处理注解信息。

Java 使用 java.lang.annotation.Annotation 接口来代表程序元素前面的注解，该接口是所有注解的父接口。Java 5 在 java.lang.reflect 包下新增了 AnnotatedElement 接口，该接口代表程序中可以接受注解的程序元素。该接口主要有如下几个实现类。

- Class：类定义。
- Constructor：构造器定义。
- Field：类的成员变量定义。
- Method：类的方法定义。
- Package：类的包定义。

java.lang.reflect 包下主要包含一些实现反射功能的工具类，从 Java 5 开始，java.lang.reflect 包所提供的反射 API 增加了读取运行时注解的能力。只有当定义注解时使用了@Retention(RetentionPolicy.RUNTIME)修饰，该注解才会在运行时可见，JVM 才会在装载*.class 文件时读取保存在 class 文件中的注解信息。

AnnotatedElement 接口是所有程序元素（如 Class、Method、Constructor 等）的父接口，所以程序通过反射获取了某个类的 AnnotatedElement 对象（如 Class、Method、Constructor 等）之后，程序就可以调用该对象的如下几个方法来访问注解信息。

- <A extends Annotation> A getAnnotation(Class<A> annotationClass)：返回该程序元素上存在的、指定类型的注解，如果该类型的注解不存在，则返回 null。
- <A extends Annotation> A getDeclaredAnnotation(Class<A> annotationClass)：这是 Java 8 新增的方法，该方法尝试获取直接修饰该程序元素、指定类型的注解。如果该类型的注解不存在，则返回 null。
- Annotation[] getAnnotations()：返回该程序元素上存在的所有注解。
- Annotation[] getDeclaredAnnotations()：返回直接修饰该程序元素的所有注解。
- boolean isAnnotationPresent(Class<? extends Annotation> annotationClass)：判断该程序元素上是否存在指定类型的注解，如果存在则返回 true，否则返回 false。
- <A extends Annotation> A[] getAnnotationsByType(Class<A> annotationClass)：该方法的功能与前面介绍的 getAnnotation()方法基本相似。但由于 Java 8 增加了重复注解功能，因此需要使用该方法获取修饰该程序元素、指定类型的多个注解。
- <A extends Annotation> A[] getDeclaredAnnotationsByType(Class<A> annotationClass)：该方法的功能与前面介绍的 getDeclaredAnnotations()方法基本相似。但由于 Java 8 增加了重复注解功能，因此需要使用该方法获取直接修饰该程序元素、指定类型的多个注解。

注意：

为了获得程序中的程序元素（如 Class、Method 等），必须使用反射知识，如果读者需要获得关于反射的更详细内容，可以参考本书第 18 章的介绍。

下面程序片段用于获取 Test 类的 info 方法里的所有注解，并将这些注解打印出来。

```
// 获取 Test 类的 info 方法的所有注解
Annotation[] aArray = Class.forName("Test").getMethod("info").getAnnotations();
// 遍历所有注解
for (Annotation an : aArray )
{
    System.out.println(an);
}
```

如果需要获取某个注解里的元数据，则可以将注解强制类型转换成所需的注解类型，然后通过注解对象的抽象方法来访问这些元数据。如下代码片段所示。

```
// 获取 tt 对象的 info 方法所包含的所有注解
Annotation[] annotation = tt.getClass().getMethod("info").getAnnotations();
// 遍历每个注解对象
for (Annotation tag :annotation)
{
    // 如果 tag 注解是 MyTag1 类型
    if (tag instanceof MyTag1)
    {
```

```
        System.out.println("Tag is: " + tag);
        // 将 tag 强制类型转换为 MyTag1
        // 输出 tag 对象的 method1 和 method2 两个成员变量的值
        System.out.println("tag.name(): " + ((MyTag1)tag).method1());
        System.out.println("tag.age(): " + ((MyTag1)(tag)).method2());
    }
    // 如果 tag 注解是 MyTag2 类型
    if (tag instanceof MyTag2)
    {
        System.out.println("Tag is: " + tag);
        // 将 tag 强制类型转换为 MyTag2
        // 输出 tag 对象的 method1 和 method2 两个成员变量的值
        System.out.println("tag.name(): " + ((MyTag2)tag).method1());
        System.out.println("tag.age(): " + ((MyTag2)(tag)).method2());
    }
}
```

14.3.3 使用注解的示例

下面分别介绍两个使用注解的例子，第一个注解@Testable 没有任何成员变量，仅是一个标记注解，它的作用是标记哪些方法是可测试的。

程序清单：codes\14\14.3\01\Testable.java

```
// 使用@Retention 指定注解的保留到运行时
@Retention(RetentionPolicy.RUNTIME)
// 使用@Target 指定被修饰的注解可用于修饰方法
@Target(ElementType.METHOD)
// 定义一个标记注解，不包含任何成员变量，即不可传入元数据
public @interface Testable
{
}
```

上面程序定义了一个@Testable 注解，定义该注解时使用了@Retention 和@Target 两个 JDK 的元注解，其中@Retention 注解指定 Testable 注解可以保留到运行时（JVM 可以提取到该注解的信息），而@Target 注解指定@Testable 只能修饰方法。

提示：

上面的@Testable 用于标记哪些方法是可测试的，该注解可以作为 JUnit 测试框架的补充，在 JUnit 框架中它要求测试用例的测试方法必须以 test 开头。如果使用@Testable 注解，则可把任何方法标记为可测试的。

如下 MyTest 测试用例中定义了 8 个方法，这 8 个方法没有太大的区别，其中 4 个方法使用@Testable 注解来标记这些方法是可测试的。

程序清单：codes\14\14.3\01\MyTest.java

```
public class MyTest
{
    // 使用@Testable 注解指定该方法是可测试的
    @Testable
    public static void m1()
    {
    }
    public static void m2()
    {
    }
    // 使用@Testable 注解指定该方法是可测试的
    @Testable
    public static void m3()
    {
        throw new IllegalArgumentException("参数出错了！");
    }
    public static void m4()
    {
```

```
    }
    // 使用@Testable 注解指定该方法是可测试的
    @Testable
    public static void m5()
    {
    }
    public static void m6()
    {
    }
    // 使用@Testable 注解指定该方法是可测试的
    @Testable
    public static void m7()
    {
        throw new RuntimeException("程序业务出现异常！");
    }
    public static void m8()
    {
    }
}
```

正如前面提到的，仅仅使用注解来标记程序元素对程序是不会有任何影响的，这也是 Java 注解的一条重要原则。为了让程序中的这些注解起作用，接下来必须为这些注解提供一个注解处理工具。

下面的注解处理工具会分析目标类，如果目标类中的方法使用了@Testable 注解修饰，则通过反射来运行该测试方法。

程序清单：codes\14\14.3\01\ProcessorTest.java

```
public class ProcessorTest
{
    public static void process(String clazz)
        throws ClassNotFoundException
    {
        int passed = 0;
        int failed = 0;
        // 遍历 clazz 对应的类里的所有方法
        for (Method m : Class.forName(clazz).getMethods())
        {
            // 如果该方法使用了@Testable 修饰
            if (m.isAnnotationPresent(Testable.class))
            {
                try
                {
                    // 调用 m 方法
                    m.invoke(null);
                    // 测试成功，passed 计数器加 1
                    passed++;
                }
                catch (Exception ex)
                {
                    System.out.println("方法" + m + "运行失败，异常："
                        + ex.getCause());
                    // 测试出现异常，failed 计数器加 1
                    failed++;
                }
            }
        }
        // 统计测试结果
        System.out.println("共运行了:" + (passed + failed)
            + "个方法，其中：\n" + "失败了:" + failed + "个，\n"
            + "成功了:" + passed + "个！");
    }
}
```

ProcessorTest 类里只包含一个 process(String clazz)方法，该方法可接收一个字符串参数，该方法将会分析 clazz 参数所代表的类，并运行该类里使用@Testable 修饰的方法。

该程序的主类非常简单，提供主方法，使用 ProcessorTest 来分析目标类即可。

程序清单：codes\14\14.3\01\RunTests.java

```
public class RunTests
{
    public static void main(String[] args)
        throws Exception
    {
        // 处理 MyTest 类
        ProcessorTest.process("MyTest");
    }
}
```

运行上面程序，会看到如下运行结果：

```
方法 public static void MyTest.m3()运行失败，异常：java.lang.IllegalArgumentException: 参数出错了！
方法 public static void MyTest.m7()运行失败，异常：java.lang.RuntimeException: 程序业务出现异常！
共运行了:4 个方法，其中：
失败了:2 个，
成功了:2 个！
```

通过这个运行结果可以看出，程序中的@Testable 起作用了，MyTest 类里以@Testable 注解修饰的方法都被测试了。

注意：

通过上面例子读者不难看出，其实注解十分简单，它是对源代码增加的一些特殊标记，这些特殊标记可通过反射获取，当程序获取这些特殊标记后，程序可以做出相应的处理（当然也可以完全忽略这些注解）。

前面介绍的只是一个标记注解，程序通过判断该注解存在与否来决定是否运行指定方法。下面程序通过使用注解来简化事件编程，在传统的事件编程中总是需要通过 addActionListener()方法来为事件源绑定事件监听器，本示例程序中则通过@ActionListenerFor 来为程序中的按钮绑定事件监听器。

程序清单：codes\14\14.3\02\ActionListenerFor.java

```
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface ActionListenerFor
{
    // 定义一个成员变量，用于设置元数据
    // 该 listener 成员变量用于保存监听器实现类
    Class<? extends ActionListener> listener();
}
```

定义了这个@ActionListenerFor 之后，使用该注解时需要指定一个 listener 成员变量，该成员变量用于指定监听器的实现类。下面程序使用@ActionListenerFor 注解来为两个按钮绑定事件监听器。

程序清单：codes\14\14.3\02\AnnotationTest.java

```
public class AnnotationTest
{
    private JFrame mainWin = new JFrame("使用注解绑定事件监听器");
    // 使用@ActionListenerFor 注解为 ok 按钮绑定事件监听器
    @ActionListenerFor(listener=OkListener.class)
    private JButton ok = new JButton("确定");
    // 使用@ActionListenerFor 注解为 cancel 按钮绑定事件监听器
    @ActionListenerFor(listener=CancelListener.class)
    private JButton cancel = new JButton("取消");
    public void init()
    {
        // 初始化界面的方法
        JPanel jp = new JPanel();
        jp.add(ok);
        jp.add(cancel);
        mainWin.add(jp);
```

```
        ActionListenerInstaller.processAnnotations(this);    // ①
        mainWin.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        mainWin.pack();
        mainWin.setVisible(true);
    }
    public static void main(String[] args)
    {
        new AnnotationTest().init();
    }
}
// 定义 ok 按钮的事件监听器实现类
class OkListener implements ActionListener
{
    public void actionPerformed(ActionEvent evt)
    {
        JOptionPane.showMessageDialog(null , "单击了确认按钮");
    }
}
// 定义 cancel 按钮的事件监听器实现类
class CancelListener implements ActionListener
{
    public void actionPerformed(ActionEvent evt)
    {
        JOptionPane.showMessageDialog(null , "单击了取消按钮");
    }
}
```

上面程序中的粗体字代码定义了两个 JButton 按钮，并使用@ActionListenerFor 注解为这两个按钮绑定了事件监听器，使用@ActionListenerFor 注解时传入了 listener 元数据，该数据用于设定每个按钮的监听器实现类。

正如前面提到的，如果仅在程序中使用注解是不会起任何作用的，必须使用注解处理工具来处理程序中的注解。程序中①处代码使用了 ActionListenerInstaller 类来处理本程序中的注解，该处理器分析目标对象中的所有成员变量，如果该成员变量前使用了@ActionListenerFor 修饰，则取出该注解中的 listener 元数据，并根据该数据来绑定事件监听器。

程序清单：codes\14\14.3\02\ActionListenerInstaller.java

```
public class ActionListenerInstaller
{
    // 处理注解的方法，其中 obj 是包含注解的对象
    public static void processAnnotations(Object obj)
    {
        try
        {
            // 获取 obj 对象的类
            Class cl = obj.getClass();
            // 获取指定 obj 对象的所有成员变量，并遍历每个成员变量
            for (Field f : cl.getDeclaredFields())
            {
                // 将该成员变量设置成可自由访问
                f.setAccessible(true);
                // 获取该成员变量上 ActionListenerFor 类型的注解
                ActionListenerFor a = f.getAnnotation(ActionListenerFor.class);
                // 获取成员变量 f 的值
                Object fObj  = f.get(obj);
                // 如果 f 是 AbstractButton 的实例，且 a 不为 null
                if (a != null && fObj != null
                    && fObj instanceof AbstractButton)
                {
                    // 获取 a 注解里的 listener 元数据（它是一个监听器类）
                    Class<? extends ActionListener> listenerClazz = a.listener();
                    // 使用反射来创建 listener 类的对象
                    ActionListener al = listenerClazz.newInstance();
                    AbstractButton ab = (AbstractButton)fObj;
                    // 为 ab 按钮添加事件监听器
```

```
                    ab.addActionListener(al);
                }
            }
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

上面程序中的两行粗体字代码根据@ActionListenerFor 注解的元数据取得了监听器实现类，然后通过反射来创建监听器对象，接下来将监听器对象绑定到指定的按钮（按钮由被@ActionListenerFor 修饰的 Field 表示）。

运行上面的 AnnotationTest 程序，会看到如图 14.3 所示的窗口。

单击 “确定”按钮，将会弹出如图 14.4 所示的“单击了确认按钮”对话框，这表明使用该注解成功地为 ok、cancel 两个按钮绑定了事件监听器。

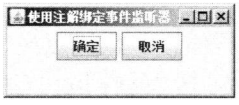


图 14.3 使用注解绑定事件监听器

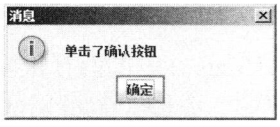


图 14.4 使用注解成功地绑定了事件监听器

14.3.4 Java 8 新增的重复注解

在 Java 8 以前，同一个程序元素前最多只能使用一个相同类型的注解；如果需要在同一个元素前使用多个相同类型的注解，则必须使用注解“容器”。例如在 Struts 2 开发中，有时需要在 Action 类上使用多个@Result 注解。在 Java 8 以前只能写成如下形式：

```
@Results({@Result(name="failure", location="failed.jsp"),
@Result(name="success", location="succ.jsp")})
public Acton FooAction{ ... }
```

上面代码中使用了两个@Result 注解，但由于传统 Java 语法不允许多次使用@Result 修饰同一个类，因此程序必须使用@Results 注解作为两个@Result 的容器——实质是，@Results 注解只包含一个名字为 value、类型为 Result[]的成员变量，程序指定的多个@Result 将作为@Results 的 value 属性（数组类型）的数组元素。

从 Java 8 开始，上面语法可以得到简化：Java 8 允许使用多个相同类型的注解来修饰同一个类，因此上面代码可能（之所以说可能，是因为重复注解还需要对原来的注解进行改造）可简化为如下形式：

```
@Result(name="failure", location="failed.jsp")
@Result(name="success", location="succ.jsp")
public Acton FooAction{ ... }
```

提示：

读者暂时无须理会 Struts 2 Action 的功能和用法，此处只是介绍如何在 Action 类前使用多个@Result 注解。在传统语法下，必须使用@Results 来包含多个@Result；在 Java 8 语法规范下，即可直接使用多个@Result 修饰 Action 类。

开发重复注解需要使用@Repeatable 修饰，下面通过示例来示范如何开发重复注解。首先定义一个 FKTag 注解。

程序清单：codes\14\14.3\FkTag.java

```
// 指定该注解信息会保留到运行时
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface FkTag
{
    // 为该注解定义 2 个成员变量
    String name() default "疯狂软件";
```

```
    int age();
}
```

上面定义了 FKTag 注解，该注解包含两个成员变量。但该注解默认不能作为重复注解使用，如果使用两个以上的该注解修饰同一个类，编译器会报错。

为了将该注解改造成重复注解，需要使用@Repeatable 修饰该注解，使用@Repeatable 时必须为 value 成员变量指定值，该成员变量的值应该是一个“容器”注解——该“容器”注解可包含多个@FkTag，因此还需要定义如下的“容器”注解。

程序清单：codes\14\14.3\FkTags.java

```
// 指定该注解信息会保留到运行时
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface FkTags
{
    // 定义 value 成员变量，该成员变量可接受多个@FkTag 注解
    FkTag[] value();
}
```

留意定义@FkTags 注解的两行粗体字代码，先看第二行粗体字代码，该代码定义了一个 FkTag[]类型的 value 成员变量，这意味着@FkTags 注解的 value 成员变量可接受多个@FkTag 注解，因此@FkTags 注解可作为@FkTag 的容器。

定义@FkTags 注解的第一行粗体字代码指定@FKTags 注解信息也可保留到运行时，这是必需的，因为：@FKTag 注解信息需要保留到运行时，如果@FkTags 注解只能保留到源代码级别(RetentionPolicy.SOURCE)或类文件(RetentionPolicy.CLASS)，将会导致@FkTags 的保留期小于@FkTag 的保留期，如果程序将多个@FkTag 注解放入@FkTags 中，若 JVM 丢弃了@FKTags 注解，自然也就丢弃了@FkTag 的信息——而我们希望@FkTag 注解可以保留到运行时，这就矛盾了。

注意：

“容器”注解的保留期必须比它所包含的注解的保留期更长，否则编译器会报错。

接下来程序可在定义@FkTag 注解时添加如下修饰代码：

```
@Repeatable(FkTags.class)
```

经过上面步骤，就成功地定义了一个重复注解：@FkTag。读者可能已经发现，实际上@FkTag 依然有“容器”注解，因此依然可用传统代码来使用该注解：

```
@FkTags({@FkTag(age=5),
    @FkTag(name="疯狂 Java" , age=9)})
```

又由于@FkTag 是重复注解，因此可直接使用两个@FkTag 注解，如下代码所示。

```
@FkTag(age=5)
@FkTag(name="疯狂 Java" , age=9)
```

实际上，第二种用法只是一个简化写法，系统依然将两个@FkTag 注解作为@FkTags 的 value 成员变量的数组元素。如下程序演示了重复注解的本质。

程序清单：codes\14\14.3\FkTagTest.java

```
@FkTag(age=5)
@FkTag(name="疯狂 Java" , age=9)
public class FkTagTest
{
    public static void main(String[] args)
    {
        Class<FkTagTest> clazz = FkTagTest.class;
        /* 使用 Java 8 新增的 getDeclaredAnnotationsByType()方法获取
            修饰 FkTagTest 类的多个@FkTag 注解 */
        FkTag[] tags = clazz.getDeclaredAnnotationsByType(FkTag.class);
        // 遍历修饰 FkTagTest 类的多个@FkTag 注解
```

```
        for(FkTag tag : tags)
        {
            System.out.println(tag.name() + "-->" + tag.age());
        }
        /* 使用传统的 getDeclaredAnnotation()方法获取
            修饰 FkTagTest 类的@FkTags 注解 */
        FkTags container = clazz.getDeclaredAnnotation(FkTags.class);
        System.out.println(container);
    }
}
```

上面程序中第一行粗体字代码获取修饰 FkTagTest 类的多个@FkTag 注解，此行代码使用的是 Java 8 新增的 getDeclaredAnnotationsByType()方法，该方法的功能与传统的 getDeclaredAnnotation()方法相同，只不过 getDeclaredAnnotationsByType()方法相当于功能增强版，它可以获取多个重复注解，而 getDeclaredAnnotation()方法则只能获取一个（在 Java 8 以前，不允许出现重复注解）。

上面程序中第二行粗体字代码尝试获取修饰 FkTagTest 类的@FkTags 注解，虽然上面源代码中并未显式使用@FkTags 注解，但由于程序使用了两个@FkTag 注解修饰该类，因此系统会自动将两个@FkTag 注解作为@FkTags 的 value 成员变量的数组元素处理。因此，第二行粗体字代码将可以成功地获取到@FkTags 注解。

编译、运行程序，可以看到如下输出：

```
疯狂软件-->5
疯狂 Java-->9
@FkTags(value=[@FkTag(name=疯狂软件, age=5), @FkTag(name=疯狂 Java, age=9)])
```

注意：

重复注解只是一种简化写法，这种简化写法是一种假象：多个重复注解其实会被作为“容器”注解的 value 成员变量的数组元素。例如上面的重复的@FkTag 注解其实会被作为@FkTags 注解的 value 成员变量的数组元素处理。

14.3.5 Java 8 新增的类型注解

Java 8 为 ElementType 枚举增加了 TYPE_PARAMETER、TYPE_USE 两个枚举值，这样就允许定义枚举时使用@Target(ElementType.TYPE_USE)修饰，这种注解被称为类型注解（Type Annotation），类型注解可用于修饰在任何地方出现的类型。

在 Java 8 以前，只能在定义各种程序元素（定义类、定义接口、定义方法、定义成员变量……）时使用注解。从 Java 8 开始，类型注解可以修饰在任何地方出现的类型。比如，允许在如下位置使用类型注解。

- 创建对象（用 new 关键字创建）。
- 类型转换。
- 使用 implements 实现接口。
- 使用 throws 声明抛出异常。

上面这些情形都会用到类型，因此都可以使用类型注解来修饰。

下面程序将会定义一个简单的类型注解，然后就可在任何用到类型的地方使用类型注解了，读者可通过该示例了解类型注解无处不在的神奇魔力。

程序清单：codes\14\14.3\TypeAnnotationTest.java

```
// 定义一个简单的类型注解，不带任何成员变量
@Target(ElementType.TYPE_USE)
@interface NotNull{}
// 定义类时使用类型注解
@NotNull
public class TypeAnnotationTest
    implements @NotNull /* implements 时使用类型注解 */ Serializable
{
    // 方法形参中使用类型注解
```

```
    public static void main(@NotNull String[] args)
        // throws 时使用类型注解
        throws @NotNull FileNotFoundException
    {
        Object obj = "fkjava.org";
        // 强制类型转换时使用类型注解
        String str = (@NotNull String)obj;
        // 创建对象时使用类型注解
        Object win = new @NotNull JFrame("疯狂软件");
    }
    // 泛型中使用类型注解
    public void foo(List<@NotNull String> info){}
}
```

上面的粗体字代码都是可正常使用类型注解的例子，从这个示例可以看到，Java 程序到处“写满”了类型注解，这种“无处不在”的类型注解可以让编译器执行更严格的代码检查，从而提高程序的健壮性。

需要指出的是，上面程序虽然大量使用了@NotNull 注解，但这些注解暂时不会起任何作用——因为并没有为这些注解提供处理工具。而且 Java 8 本身并没有提供对类型注解执行检查的框架，因此如果需要让这些类型注解发挥作用，开发者需要自己实现类型注解检查框架。

幸运的是，Java 8 提供了类型注解之后，第三方组织在发布他们的框架时，可能会随着框架一起发布类型注解检查工具，这样普通开发者即可直接使用第三方框架提供的类型注解，从而让编译器执行更严格的检查，保证代码更加健壮。

14.4 编译时处理注解

APT（Annotation Processing Tool）是一种注解处理工具，它对源代码文件进行检测，并找出源文件所包含的注解信息，然后针对注解信息进行额外的处理。

使用 APT 工具处理注解时可以根据源文件中的注解生成额外的源文件和其他的文件（文件的具体内容由注解处理器的编写者决定），APT 还会编译生成的源代码文件和原来的源文件，将它们一起生成 class 文件。

使用 APT 的主要目的是简化开发者的工作量，因为 APT 可以在编译程序源代码的同时生成一些附属文件（比如源文件、类文件、程序发布描述文件等），这些附属文件的内容也都与源代码相关。换句话说，使用 APT 可以代替传统的对代码信息和附属文件的维护工作。

了解过 Hibernate 早期版本的读者都知道：每写一个 Java 类文件，还必须额外地维护一个 Hibernate 映射文件（名为*.hbm.xml 的文件，也有一些工具可以自动生成）。下面将使用注解来简化这步操作。

提示：

不了解 Hibernate 的读者也无须担心，你只需要明白此处要做什么即可——通过注解可以在 Java 源文件中放置一些注解，然后使用 APT 工具就可以根据该注解生成另一份 XML 文件，这就是注解的作用。

Java 提供的 javac.exe 工具有一个-processor 选项，该选项可指定一个注解处理器，如果在编译 Java 源文件时通过该选项指定了注解处理器，那么这个注解处理器将会在编译时提取并处理 Java 源文件中的注解。

每个注解处理器都需要实现 javax.annotation.processing 包下的 Processor 接口。不过实现该接口必须实现它里面所有的方法，因此通常会采用继承 AbstractProcessor 的方式来实现注解处理器。一个注解处理器可以处理一种或者多种注解类型。

为了示范使用 APT 根据源文件中的注解来生成额外的文件，下面将定义 3 种注解类型，分别用于修饰持久化类、标识属性和普通成员属性。

程序清单：codes\14\14.4\Persistent.java

```
@Target(ElementType.TYPE)
```

```
@Retention(RetentionPolicy.SOURCE)
@Documented
public @interface Persistent
{
    String table();
}
```

这是一个非常简单的注解，它能修饰类、接口等类型声明，这个注解使用了@Retention 元注解指定它仅在 Java 源文件中保留，运行时不能通过反射来读取该注解信息。

下面是修饰标识属性的@Id 注解。

程序清单：codes\14\14.4\Id.java

```
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.SOURCE)
@Documented
public @interface Id
{
    String column();
    String type();
    String generator();
}
```

这个@Id 与前一个@Persistent 的结构基本相似，只是多了两个成员变量而已。下面还有一个用于修饰普通成员属性的注解。

程序清单：codes\14\14.4\Property.java

```
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.SOURCE)
@Documented
public @interface Property
{
    String column();
    String type();
}
```

定义了这三个注解之后，下面提供一个简单的 Java 类文件，这个 Java 类文件使用这三个注解来修饰。

程序清单：codes\14\14.4\Person.java

```
@Persistent(table="person_inf")
public class Person
{
    @Id(column="person_id",type="integer",generator="identity")
    private int id;
    @Property(column="person_name",type="string")
    private String name;
    @Property(column="person_age",type="integer")
    private int age;
    // 无参数的构造器
    public Person()
    {
    }
    // 初始化全部成员变量的构造器
    public Person(int id , String name , int age)
    {
        this.id = id;
        this.name = name;
        this.age = age;
    }
    // 下面省略所有成员变量的 setter 和 getter 方法
    ...
}
```

上面的 Person 类是一个非常普通的 Java 类，但这个普通的 Java 类中使用了@Persistent、@Id 和@Property 三个注解进行修饰。下面为这三个注解提供一个 APT 工具，该工具的功能是根据注解来生成一个 Hibernate 映射文件（不懂 Hibernate 也没有关系，读者只需要明白可以根据这些注解来生成另一份

XML 文件即可)。

程序清单：codes\14\14.4\HibernateAnnotationProcessor.java

```
@SupportedSourceVersion(SourceVersion.RELEASE_8)
// 指定可处理@Persistent、@Id、@Property 三个注解
@SupportedAnnotationTypes({"Persistent" , "Id" , "Property"})
public class HibernateAnnotationProcessor
    extends AbstractProcessor
{
    // 循环处理每个需要处理的程序对象
    public boolean process(Set<? extends TypeElement> annotations
        , RoundEnvironment roundEnv)
    {
        // 定义一个文件输出流，用于生成额外的文件
        PrintStream ps = null;
        try
        {
            // 遍历每个被@Persistent 修饰的 class 文件
            for (Element t : roundEnv.getElementsAnnotatedWith(Persistent.class))
            {
                // 获取正在处理的类名
                Name clazzName = t.getSimpleName();
                // 获取类定义前的@Persistent 注解
                Persistent per = t.getAnnotation(Persistent.class);
                // 创建文件输出流
                ps = new PrintStream(new FileOutputStream(clazzName
                    + ".hbm.xml"));
                // 执行输出
                ps.println("<?xml version=\"1.0\"?>");
                ps.println("<!DOCTYPE hibernate-mapping PUBLIC");
                ps.println("    \"-//Hibernate/Hibernate "
                    + "Mapping DTD 3.0//EN\"");
                ps.println("    \"http://www.hibernate.org/dtd/"
                    + "hibernate-mapping-3.0.dtd\">");
                ps.println("<hibernate-mapping>");
                ps.print("    <class name=\"" + t);
                // 输出 per 的 table()的值
                ps.println("\" table=\"" + per.table() + "\">");
                for (Element f : t.getEnclosedElements())
                {
                    // 只处理成员变量上的注解
                    if (f.getKind() == ElementKind.FIELD)   // ①
                    {
                        // 获取成员变量定义前的@Id 注解
                        Id id = f.getAnnotation(Id.class);      // ②
                        // 当@Id 注解存在时输出<id.../>元素
                        if(id != null)
                        {
                            ps.println("            <id name=\""
                                + f.getSimpleName()
                                + "\" column=\"" + id.column()
                                + "\" type=\"" + id.type()
                                + "\">");
                            ps.println("            <generator class=\""
                                + id.generator() + "\"/>");
                            ps.println("            </id>");
                        }
                        // 获取成员变量定义前的@Property 注解
                        Property p = f.getAnnotation(Property.class);  // ③
                        // 当@Property 注解存在时输出<property.../>元素
                        if (p != null)
                        {
                            ps.println("            <property name=\""
                                + f.getSimpleName()
                                + "\" column=\"" + p.column()
                                + "\" type=\"" + p.type()
                                + "\"/>");
```

```
                    }
                }
            }
            ps.println("    </class>");
            ps.println("</hibernate-mapping>");
        }
    }
    catch (Exception ex)
    {
        ex.printStackTrace();
    }
    finally
    {
        if (ps != null)
        {
            try
            {
                ps.close();
            }
            catch (Exception ex)
            {
                ex.printStackTrace();
            }
        }
    }
    return true;
  }
}
```

上面的注解处理器其实非常简单，与前面通过反射来获取注解信息不同的是，这个注解处理器使用 RoundEnvironment 来获取注解信息，RoundEnvironment 里包含了一个 getElementsAnnotatedWith()方法，可根据注解获取需要处理的程序单元，这个程序单元由 Element 代表。Element 里包含一个 getKind()方法，该方法返回 Element 所代表的程序单元，返回值可以是 ElementKind.CLASS（类）、ElementKind.FIELD（成员变量）……

除此之外，Element 还包含一个 getEnclosedElements()方法，该方法可用于获取该 Element 里定义的所有程序单元，包括成员变量、方法、构造器、内部类等。

接下来程序只处理成员变量前面的注解，因此程序先判断这个 Element 必须是 ElementKind.FIELD（如上程序中①号粗体字代码所示）。

再接下来程序调用了 Element 提供的 getAnnotation(Class clazz)方法来获取修饰该 Element 的注解，如上程序中②③号粗体字部分就是获取成员变量上注解对象的代码。获取到成员变量上的@Id、@Property 注解之后，接下来就根据它们提供的信息执行输出。

提示： 上面程序中大量使用了 IO 流来执行输出，关于 IO 流的知识请参考本书第 15 章的介绍。

提供了上面的注解处理器类之后，接下来就可使用带-processor 选项的 javac.exe 命令来编译 Person.java 了。例如如下命令：

```
rem 使用 HibernateAnnotationProcessor 作为 APT 处理 Person.java 中的注解
javac -processor HibernateAnnotationProcessor Person.java
```

提示： 上面命令被保存在 codes\14\14.4\run.cmd 文件中，读者可以直接双击该批处理文件来运行上面的命令。

通过上面的命令编译 Person.java 后，将可以看到在相同路径下生成了一个 Person.hbm.xml 文件，该文件就是根据 Person.java 里的注解生成的。该文件的内容如下：

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
```

```
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="Person" table="person_inf">
        <id name="id" column="person_id" type="integer">
        <generator class="identity"/>
        </id>
        <property name="name" column="person_name" type="string"/>
        <property name="age" column="person_age" type="integer"/>
    </class>
</hibernate-mapping>
```

对比上面 XML 文件中的粗体字部分与 Person.java 中的注解部分，它们是完全对应的，这即表明这份 XML 文件是根据 Person.java 中的注解生成的。从生成的这份 XML 文件可以看出，通过使用 APT 工具确实可以简化程序开发，程序员只需把一些关键信息通过注解写在程序中，然后使用 APT 工具就可生成额外的文件。

14.5 本章小结

本章主要介绍了 Java 的注解支持，通过使用注解可以为程序提供一些元数据，这些元数据可以在编译、运行时被读取，从而提供更多额外的处理信息。本章详细介绍了 JDK 提供的 5 个基本注解的用法，也详细讲解了 JDK 提供的 4 个用于修饰注解的元注解的用法。除此之外，本章也介绍了如何自定义并使用注解，最后还介绍了使用 APT 工具来处理注解。

本章练习

1. 定义一个简单的@Foo 注解，该注解只能修饰类、方法，该注解只在源代码阶段有效。
2. 定义一个@Bar 注解，并为该注解提供 name 和 price 两个属性，该注解只能修饰方法、成员变量。
3. 定义@Getter 和@Setter 注解，它们只能修饰成员变量。为这两个注解编写 APT 工具，APT 工具会为它们修饰的成员变量对应地添加 getter、setter 方法。

CHAPTER

15

第15章 输入/输出

本章要点

- 使用 File 类访问本地文件系统
- 使用文件过滤器
- 理解 IO 流的模型和处理方式
- 使用 IO 流执行输入、输出操作
- 使用转换流将字节流转换为字符流
- 推回流的功能和用法
- 重定向标准输入、输出
- 访问其他进程的输入、输出
- RandomAccessFile 的功能和用法
- 对象序列化机制和作用
- 通过实现 Serializable 接口实现序列化
- 实现定制的序列化
- 通过实现 Externalizable 接口实现序列化
- Java 新 IO 的概念和作用
- 使用 Buffer 和 Channel 完成输入、输出
- Charset 的功能和用法
- FileLock 的功能和用法
- NIO.2 的文件 IO 和文件系统
- 通过 NIO.2 监控文件变化
- 通过 NIO.2 访问、修改文件属性

IO（输入/输出）是比较乏味的事情，因为看不到明显的运行效果，但输入/输出是所有程序都必需的部分——使用输入机制，允许程序读取外部数据（包括来自磁盘、光盘等存储设备的数据）、用户输入数据；使用输出机制，允许程序记录运行状态，将程序数据输出到磁盘、光盘等存储设备中。

Java 的 IO 通过 java.io 包下的类和接口来支持，在 java.io 包下主要包括输入、输出两种 IO 流，每种输入、输出流又可分为字节流和字符流两大类。其中字节流以字节为单位来处理输入、输出操作，而字符流则以字符来处理输入、输出操作。除此之外，Java 的 IO 流使用了一种装饰器设计模式，它将 IO 流分成底层节点流和上层处理流，其中节点流用于和底层的物理存储节点直接关联——不同的物理节点获取节点流的方式可能存在一定的差异，但程序可以把不同的物理节点流包装成统一的处理流，从而允许程序使用统一的输入、输出代码来读取不同的物理存储节点的资源。

Java 7 在 java.nio 及其子包下提供了一系列全新的 API，这些 API 是对原有新 IO 的升级，因此也被称为 NIO 2，通过这些 NIO 2，程序可以更高效地进行输入、输出操作。本章也会介绍 Java 7 所提供的 NIO 2。

除此之外，本章还会介绍 Java 对象的序列化机制，使用序列化机制可以把内存中的 Java 对象转换成二进制字节流，这样就可以把 Java 对象存储到磁盘里，或者在网络上传输 Java 对象。这也是 Java 提供分布式编程的重要基础。

15.1 File 类

File 类是 java.io 包下代表与平台无关的文件和目录，也就是说，如果希望在程序中操作文件和目录，都可以通过 File 类来完成。值得指出的是，不管是文件还是目录都是使用 File 来操作的，File 能新建、删除、重命名文件和目录，File 不能访问文件内容本身。如果需要访问文件内容本身，则需要使用输入/输出流。

15.1.1 访问文件和目录

File 类可以使用文件路径字符串来创建 File 实例，该文件路径字符串既可以是绝对路径，也可以是相对路径。在默认情况下，系统总是依据用户的工作路径来解释相对路径，这个路径由系统属性“user.dir”指定，通常也就是运行 Java 虚拟机时所在的路径。

一旦创建了 File 对象后，就可以调用 File 对象的方法来访问，File 类提供了很多方法来操作文件和目录，下面列出一些比较常用的方法。

1. 访问文件名相关的方法

- String getName()：返回此 File 对象所表示的文件名或路径名（如果是路径，则返回最后一级子路径名）。
- String getPath()：返回此 File 对象所对应的路径名。
- File getAbsoluteFile()：返回此 File 对象的绝对路径。
- String getAbsolutePath()：返回此 File 对象所对应的绝对路径名。
- String getParent()：返回此 File 对象所对应目录（最后一级子目录）的父目录名。
- boolean renameTo(File newName)：重命名此 File 对象所对应的文件或目录，如果重命名成功，则返回 true；否则返回 false。

2. 文件检测相关的方法

- boolean exists()：判断 File 对象所对应的文件或目录是否存在。
- boolean canWrite()：判断 File 对象所对应的文件和目录是否可写。
- boolean canRead()：判断 File 对象所对应的文件和目录是否可读。
- boolean isFile()：判断 File 对象所对应的是否是文件，而不是目录。
- boolean isDirectory()：判断 File 对象所对应的是否是目录，而不是文件。
- boolean isAbsolute()：判断 File 对象所对应的文件或目录是否是绝对路径。该方法消除了不同平台的差异，可以直接判断 File 对象是否为绝对路径。在 UNIX/Linux/BSD 等系统上，如果路

径名开头是一条斜线（/），则表明该 File 对象对应一个绝对路径；在 Windows 等系统上，如果路径开头是盘符，则说明它是一个绝对路径。

3．获取常规文件信息

- long lastModified()：返回文件的最后修改时间。
- long length()：返回文件内容的长度。

4．文件操作相关的方法

- boolean createNewFile()：当此 File 对象所对应的文件不存在时，该方法将新建一个该 File 对象所指定的新文件，如果创建成功则返回 true；否则返回 false。
- boolean delete()：删除 File 对象所对应的文件或路径。
- static File createTempFile(String prefix, String suffix)：在默认的临时文件目录中创建一个临时的空文件，使用给定前缀、系统生成的随机数和给定后缀作为文件名。这是一个静态方法，可以直接通过 File 类来调用。prefix 参数必须至少是 3 字节长。建议前缀使用一个短的、有意义的字符串，比如 "hjb" 或 "mail"。suffix 参数可以为 null，在这种情况下，将使用默认的后缀“.tmp”。
- static File createTempFile(String prefix, String suffix, File directory) ：在 directory 所指定的目录中创建一个临时的空文件，使用给定前缀、系统生成的随机数和给定后缀作为文件名。这是一个静态方法，可以直接通过 File 类来调用。
- void deleteOnExit()：注册一个删除钩子，指定当 Java 虚拟机退出时，删除 File 对象所对应的文件和目录。

5．目录操作相关的方法

- boolean mkdir()：试图创建一个 File 对象所对应的目录，如果创建成功，则返回 true；否则返回 false。调用该方法时 File 对象必须对应一个路径，而不是一个文件。
- String[] list()：列出 File 对象的所有子文件名和路径名，返回 String 数组。
- File[] listFiles()：列出 File 对象的所有子文件和路径，返回 File 数组。
- static File[] listRoots()：列出系统所有的根路径。这是一个静态方法，可以直接通过 File 类来调用。

上面详细列出了 File 类的常用方法，下面程序以几个简单方法来测试一下 File 类的功能。

程序清单：codes\15\15.1\FileTest.java

```
public class FileTest
{
    public static void main(String[] args)
        throws IOException
    {
        // 以当前路径来创建一个 File 对象
        File file = new File(".");
        // 直接获取文件名，输出一点
        System.out.println(file.getName());
        // 获取相对路径的父路径可能出错，下面代码输出 null
        System.out.println(file.getParent());
        // 获取绝对路径
        System.out.println(file.getAbsoluteFile());
        // 获取上一级路径
        System.out.println(file.getAbsoluteFile().getParent());
        // 在当前路径下创建一个临时文件
        File tmpFile = File.createTempFile("aaa", ".txt", file);
        // 指定当 JVM 退出时删除该文件
        tmpFile.deleteOnExit();
        // 以系统当前时间作为新文件名来创建新文件
        File newFile = new File(System.currentTimeMillis() + "");
        System.out.println("newFile 对象是否存在：" + newFile.exists());
        // 以指定 newFile 对象来创建一个文件
```

```
        newFile.createNewFile();
        // 以 newFile 对象来创建一个目录，因为 newFile 已经存在
        // 所以下面方法返回 false，即无法创建该目录
        newFile.mkdir();
        // 使用 list()方法列出当前路径下的所有文件和路径
        String[] fileList = file.list();
        System.out.println("====当前路径下所有文件和路径如下====");
        for (String fileName : fileList)
        {
            System.out.println(fileName);
        }
        // listRoots()静态方法列出所有的磁盘根路径
        File[] roots = File.listRoots();
        System.out.println("====系统所有根路径如下====");
        for (File root : roots)
        {
            System.out.println(root);
        }
    }
}
```

运行上面程序，可以看到程序列出当前路径的所有文件和路径时，列出了程序创建的临时文件，但程序运行结束后，aaa.txt 临时文件并不存在，因为程序指定虚拟机退出时自动删除该文件。

上面程序还有一点需要注意，当使用相对路径的 File 对象来获取父路径时可能引起错误，因为该方法返回将 File 对象所对应的目录名、文件名里最后一个子目录名、子文件名删除后的结果，如上面程序中的粗体字代码所示。

注意：

Windows 的路径分隔符使用反斜线（\），而 Java 程序中的反斜线表示转义字符，所以如果需要在 Windows 的路径下包括反斜线，则应该使用两条反斜线，如 F:\\abc\\test.txt，或者直接使用斜线（/）也可以，Java 程序支持将斜线当成平台无关的路径分隔符。

15.1.2 文件过滤器

在 File 类的 list()方法中可以接收一个 FilenameFilter 参数，通过该参数可以只列出符合条件的文件。这里的 FilenameFilter 接口和 javax.swing.filechooser 包下的 FileFilter 抽象类的功能非常相似，可以把 FileFilter 当成 FilenameFilter 的实现类，但可能 Sun 在设计它们时产生了一些小小遗漏，所以没有让 FileFilter 实现 FilenameFilter 接口。

FilenameFilter 接口里包含了一个 accept(File dir, String name)方法，该方法将依次对指定 File 的所有子目录或者文件进行迭代，如果该方法返回 true，则 list()方法会列出该子目录或者文件。

程序清单：codes\15\15.1\FilenameFilterTest.java

```
public class FilenameFilterTest
{
    public static void main(String[] args)
    {
        File file = new File(".");
        // 使用 Lambda 表达式（目标类型为 FilenameFilter）实现文件过滤器
        // 如果文件名以.java 结尾，或者文件对应一个路径，则返回 true
        String[] nameList = file.list((dir, name) -> name.endsWith(".java")
            || new File(name).isDirectory());
        for(String name : nameList)
        {
            System.out.println(name);
        }
    }
}
```

上面程序中的粗体字代码部分实现了 accept()方法，实现 accept()方法就是指定自己的规则，指定哪些文件应该由 list()方法列出。

运行上面程序，将看到当前路径下所有的*.java 文件以及文件夹被列出。

提示：

FilenameFilter 接口内只有一个抽象方法，因此该接口也是一个函数式接口，可使用 Lambda 表达式创建实现该接口的对象。

15.2 理解 Java 的 IO 流

Java 的 IO 流是实现输入/输出的基础，它可以方便地实现数据的输入/输出操作，在 Java 中把不同的输入/输出源（键盘、文件、网络连接等）抽象表述为“流”（stream），通过流的方式允许 Java 程序使用相同的方式来访问不同的输入/输出源。stream 是从起源（source）到接收（sink）的有序数据。

Java 把所有传统的流类型（类或抽象类）都放在 java.io 包中，用以实现输入/输出功能。

提示：

因为 Java 提供了这种 IO 流的抽象，所以开发者可以使用一致的 IO 代码去读写不同的 IO 流节点。

15.2.1 流的分类

按照不同的分类方式，可以将流分为不同的类型，下面从不同的角度来对流进行分类，它们在概念上可能存在重叠的地方。

1. 输入流和输出流

按照流的流向来分，可以分为输入流和输出流。

- 输入流：只能从中读取数据，而不能向其写入数据。
- 输出流：只能向其写入数据，而不能从中读取数据。

此处的输入、输出涉及一个方向问题，对于如图 15.1 所示的数据流向，数据从内存到硬盘，通常称为输出流——也就是说，这里的输入、输出都是从程序运行所在内存的角度来划分的。

提示：

如果从硬盘的角度来考虑，如图 15.1 所示的数据流应该是输入流才对；但划分输入/输出流时是从程序运行所在内存的角度来考虑的，因此如图 15.1 所的流是输出流，而不是输入流。

对于如图 15.2 所示的数据流向，数据从服务器通过网络流向客户端，在这种情况下，Server 端的内存负责将数据输出到网络里，因此 Server 端的程序使用输出流；Client 端的内存负责从网络里读取数据，因此 Client 端的程序应该使用输入流。

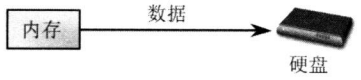


图 15.1 数据从内存到硬盘

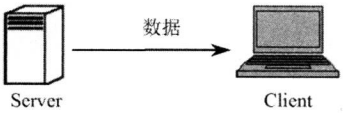


图 15.2 数据从服务器到客户端

Java 的输入流主要由 InputStream 和 Reader 作为基类，而输出流则主要由 OutputStream 和 Writer 作为基类。它们都是一些抽象基类，无法直接创建实例。

2. 字节流和字符流

字节流和字符流的用法几乎完全一样，区别在于字节流和字符流所操作的数据单元不同——字节流操作的数据单元是 8 位的字节，而字符流操作的数据单元是 16 位的字符。

字节流主要由 InputStream 和 OutputStream 作为基类，而字符流则主要由 Reader 和 Writer 作为基类。

3. 节点流和处理流

按照流的角色来分，可以分为节点流和处理流。

可以从/向一个特定的 IO 设备（如磁盘、网络）读/写数据的流，称为节点流，节点流也被称为低级流（Low Level Stream）。图 15.3 显示了节点流示意图。

从图 15.3 中可以看出，当使用节点流进行输入/输出时，程序直接连接到实际的数据源，和实际的输入/输出节点连接。

处理流则用于对一个已存在的流进行连接或封装，通过封装后的流来实现数据读/写功能。处理流也被称为高级流。图 15.4 显示了处理流示意图。

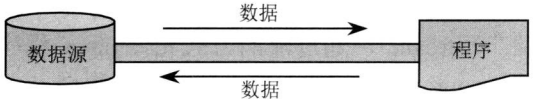


图 15.3　节点流示意图

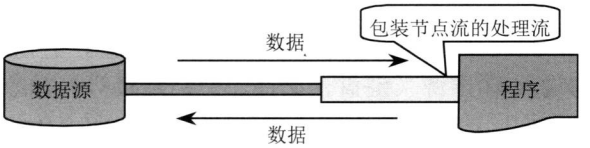


图 15.4　处理流示意图

从图 15.4 中可以看出，当使用处理流进行输入/输出时，程序并不会直接连接到实际的数据源，没有和实际的输入/输出节点连接。使用处理流的一个明显好处是，只要使用相同的处理流，程序就可以采用完全相同的输入/输出代码来访问不同的数据源，随着处理流所包装节点流的变化，程序实际所访问的数据源也相应地发生变化。

提示：

实际上，Java 使用处理流来包装节点流是一种典型的装饰器设计模式，通过使用处理流来包装不同的节点流，既可以消除不同节点流的实现差异，也可以提供更方便的方法来完成输入/输出功能。因此处理流也被称为包装流。

15.2.2　流的概念模型

Java 把所有设备里的有序数据抽象成流模型，简化了输入/输出处理，理解了流的概念模型也就了解了 Java IO。

Java 的 IO 流共涉及 40 多个类，这些类看上去芜杂而凌乱，但实际上非常规则，而且彼此之间存在非常紧密的联系。Java 的 IO 流的 40 多个类都是从如下 4 个抽象基类派生的。

- InputStream/Reader：所有输入流的基类，前者是字节输入流，后者是字符输入流。
- OutputStream/Writer：所有输出流的基类，前者是字节输出流，后者是字符输出流。

对于 InputStream 和 Reader 而言，它们把输入设备抽象成一个“水管”，这个水管里的每个“水滴”依次排列，如图 15.5 所示。

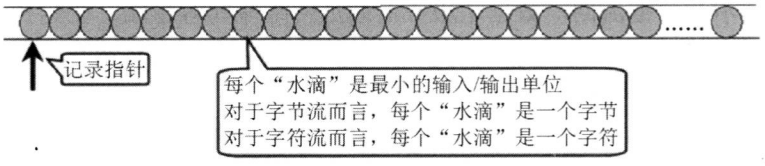


图 15.5　输入流模型图

从图 15.5 中可以看出，字节流和字符流的处理方式其实非常相似，只是它们处理的输入/输出单位不同而已。输入流使用隐式的记录指针来表示当前正准备从哪个“水滴”开始读取，每当程序从 InputStream 或 Reader 里取出一个或多个“水滴”后，记录指针自动向后移动；除此之外，InputStream 和 Reader 里都提供一些方法来控制记录指针的移动。

对于 OutputStream 和 Writer 而言，它们同样把输出设备抽象成一个“水管”，只是这个水管里没有任何水滴，如图 15.6 所示。

正如图 15.6 所示，当执行输出时，程序相当于依次把“水滴”放入到输出流的水管中，输出流同样采用隐式的记录指针来标识当前水滴即将放入的位置，每当程序向 OutputStream 或 Writer 里输出一个或多个水滴后，记录指针自动向后移动。

图 15.5 和图 15.6 显示了 Java IO 流的基本概念模型，除此之外，Java 的处理流模型则体现了 Java 输入/输出流设计的灵活性。处理流的功能主要体现在以下两个方面。

- 性能的提高：主要以增加缓冲的方式来提高输入/输出的效率。
- 操作的便捷：处理流可能提供了一系列便捷的方法来一次输入/输出大批量的内容，而不是输入/输出一个或多个“水滴”。

处理流可以“嫁接”在任何已存在的流的基础之上，这就允许 Java 应用程序采用相同的代码、透明的方式来访问不同的输入/输出设备的数据流。图 15.7 显示了处理流的模型。

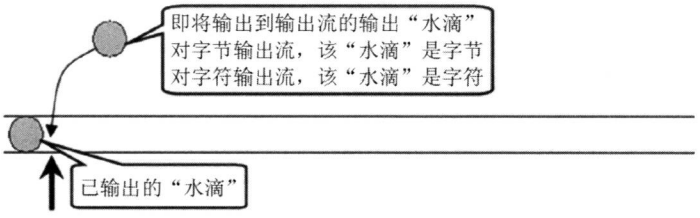


图 15.6 输出流模型图

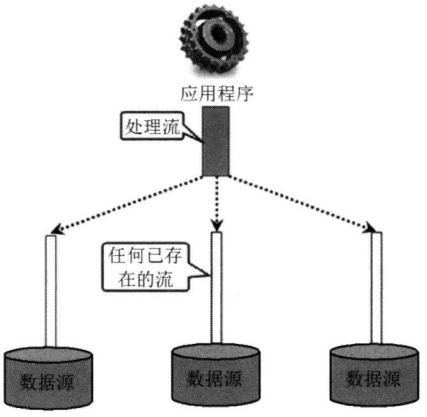


图 15.7 处理流模型图

通过使用处理流，Java 程序无须理会输入/输出节点是磁盘、网络还是其他的输入/输出设备，程序只要将这些节点流包装成处理流，就可以使用相同的输入/输出代码来读写不同的输入/输出设备的数据。

15.3 字节流和字符流

本书会把字节流和字符流放在一起讲解，因为它们的操作方式几乎完全一样，区别只是操作的数据单元不同而已——字节流操作的数据单元是字节，字符流操作的数据单元是字符。

15.3.1 InputStream 和 Reader

InputStream 和 Reader 是所有输入流的抽象基类，本身并不能创建实例来执行输入，但它们将成为所有输入流的模板，所以它们的方法是所有输入流都可使用的方法。

在 InputStream 里包含如下三个方法。

- int read()：从输入流中读取单个字节（相当于从图 15.5 所示的水管中取出一滴水），返回所读取的字节数据（字节数据可直接转换为 int 类型）。
- int read(byte[] b)：从输入流中最多读取 b.length 个字节的数据，并将其存储在字节数组 b 中，返回实际读取的字节数。
- int read(byte[] b, int off, int len)：从输入流中最多读取 len 个字节的数据，并将其存储在数组 b 中，放入数组 b 中时，并不是从数组起点开始，而是从 off 位置开始，返回实际读取的字节数。

在 Reader 里包含如下三个方法。

- int read()：从输入流中读取单个字符（相当于从图 15.5 所示的水管中取出一滴水），返回所读取的字符数据（字符数据可直接转换为 int 类型）。
- int read(char[] cbuf)：从输入流中最多读取 cbuf.length 个字符的数据，并将其存储在字符数组 cbuf 中，返回实际读取的字符数。
- int read(char[] cbuf, int off, int len)：从输入流中最多读取 len 个字符的数据，并将其存储在字符数组 cbuf 中，放入数组 cbuf 中时，并不是从数组起点开始，而是从 off 位置开始，返回实际读取的字符数。

对比 InputStream 和 Reader 所提供的方法，就不难发现这两个基类的功能基本是一样的。InputStream 和 Reader 都是将输入数据抽象成如图 15.5 所示的水管，所以程序既可以通过 read()方法每次读取一个

“水滴”，也可以通过 read(char[] cbuf)或 read(byte[] b)方法来读取多个“水滴”。当使用数组作为 read()方法的参数时，可以理解为使用一个“竹筒”到如图 15.5 所示的水管中取水，如图 15.8 所示。read(char[] cbuf)方法中的数组可理解成一个“竹筒”，程序每次调用输入流的 read(char[] cbuf)或 read(byte[] b)方法，就相当于用“竹筒”从输入流中取出一筒“水滴”，程序得到“竹筒”里的“水滴”后，转换成相应的数据即可；程序多次重复这个“取水”过程，直到最后。程序如何判断取水取到了最后呢？直到 read(char[] cbuf)或 read(byte[] b)方法返回–1，即表明到了输入流的结束点。

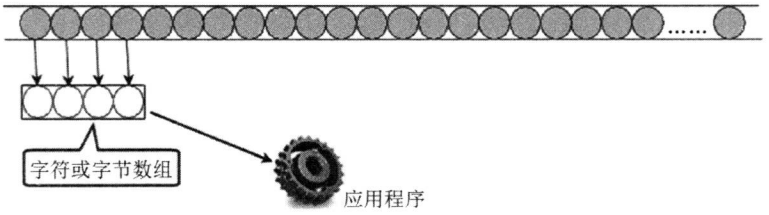


图 15.8 从输入流中读取数据

正如前面提到的，InputStream 和 Reader 都是抽象类，本身不能创建实例，但它们分别有一个用于读取文件的输入流：FileInputStream 和 FileReader，它们都是节点流——会直接和指定文件关联。下面程序示范了使用 FileInputStream 来读取自身的效果。

程序清单：codes\15\15.3\FileInputStreamTest.java

```
public class FileInputStreamTest
{
    public static void main(String[] args) throws IOException
    {
        // 创建字节输入流
        FileInputStream fis = new FileInputStream(
            "FileInputStreamTest.java");
        // 创建一个长度为 1024 的“竹筒”
        byte[] bbuf = new byte[1024];
        // 用于保存实际读取的字节数
        int hasRead = 0;
        // 使用循环来重复“取水”过程
        while ((hasRead = fis.read(bbuf)) > 0 )
        {
            // 取出“竹筒”中的水滴（字节），将字节数组转换成字符串输入
            System.out.print(new String(bbuf , 0 , hasRead ));
        }
        // 关闭文件输入流，放在 finally 块里更安全
        fis.close();
    }
}
```

上面程序中的粗体字代码是使用 FileInputStream 循环“取水”的过程，运行上面程序，将会输出上面程序的源代码。

注意：

上面程序创建了一个长度为 1024 的字节数组来读取该文件，实际上该 Java 源文件的长度还不到 1024 字节，也就是说，程序只需要执行一次 read()方法即可读取全部内容。但如果创建较小长度的字节数组，程序运行时在输出中文注释时就可能出现乱码——这是因为本文件保存时采用的是 GBK 编码方式，在这种方式下，每个中文字符占 2 字节，如果 read()方法读取时只读到了半个中文字符，这将导致乱码。

上面程序最后使用了 fis.close()来关闭该文件输入流，与 JDBC 编程一样，程序里打开的文件 IO 资源不属于内存里的资源，垃圾回收机制无法回收该资源，所以应该显式关闭文件 IO 资源。Java 7 改写了所有的 IO 资源类，它们都实现了 AutoCloseable 接口，因此都可通过自动关闭资源的 try 语句来关闭这些 IO 流。下面程序使用 FileReader 来读取文件本身。

程序清单：codes\15\15.3\FileReaderTest.java

```
public class FileReaderTest
{
    public static void main(String[] args) throws IOException
    {
        try(
            // 创建字符输入流
            FileReader fr = new FileReader("FileReaderTest.java"))
        {
            // 创建一个长度为 32 的“竹筒”
            char[] cbuf = new char[32];
            // 用于保存实际读取的字符数
            int hasRead = 0;
            // 使用循环来重复“取水”过程
            while ((hasRead = fr.read(cbuf)) > 0 )
            {
                // 取出“竹筒”中的水滴（字符），将字符数组转换成字符串输入！
                System.out.print(new String(cbuf , 0 , hasRead));
            }
        }
        catch (IOException ex)
        {
            ex.printStackTrace();
        }
    }
}
```

上面的 FileReaderTest. java 程序与前面的 FileInputStreamTest.java 并没有太大的不同，程序只是将字符数组的长度改为 32，这意味着程序需要多次调用 read()方法才可以完全读取输入流的全部数据。程序最后使用了自动关闭资源的 try 语句来关闭文件输入流，这样可以保证输入流一定会被关闭。

除此之外，InputStream 和 Reader 还支持如下几个方法来移动记录指针。

- void mark(int readAheadLimit)：在记录指针当前位置记录一个标记（mark）。
- boolean markSupported()：判断此输入流是否支持 mark()操作，即是否支持记录标记。
- void reset()：将此流的记录指针重新定位到上一次记录标记（mark）的位置。
- long skip(long n)：记录指针向前移动 *n* 个字节/字符。

15.3.2 OutputStream 和 Writer

OutputStream 和 Writer 也非常相似，它们采用如图 15.6 所示的模型来执行输出，两个流都提供了如下三个方法。

- void write(int c)：将指定的字节/字符输出到输出流中，其中 c 既可以代表字节，也可以代表字符。
- void write(byte[]/char[] buf)：将字节数组/字符数组中的数据输出到指定输出流中。
- void write(byte[]/char[] buf, int off, int len)：将字节数组/字符数组中从 off 位置开始，长度为 len 的字节/字符输出到输出流中。

因为字符流直接以字符作为操作单位，所以 Writer 可以用字符串来代替字符数组，即以 String 对象作为参数。Writer 里还包含如下两个方法。

- void write(String str)：将 str 字符串里包含的字符输出到指定输出流中。
- void write(String str, int off, int len)：将 str 字符串里从 off 位置开始，长度为 len 的字符输出到指定输出流中。

下面程序使用 FileInputStream 来执行输入，并使用 FileOutputStream 来执行输出，用以实现复制 FileOutputStreamTest.java 文件的功能。

程序清单：codes\15\15.3\FileOutputStreamTest.java

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

```
        try(
            // 创建字节输入流
            FileInputStream fis = new FileInputStream(
                "FileOutputStreamTest.java");
            // 创建字节输出流
            FileOutputStream fos = new FileOutputStream("newFile.txt"))
        {
            byte[] bbuf = new byte[32];
            int hasRead = 0;
            // 循环从输入流中取出数据
            while ((hasRead = fis.read(bbuf)) > 0 )
            {
                // 每读取一次，即写入文件输出流，读了多少，就写多少
                fos.write(bbuf , 0 , hasRead);
            }
        }
        catch (IOException ioe)
        {
            ioe.printStackTrace();
        }
    }
}
```

运行上面程序，将看到系统当前路径下多了一个文件：newFile.txt，该文件的内容和 FileOutputStreamTest.java 文件的内容完全相同。

注意：

使用 Java 的 IO 流执行输出时，不要忘记关闭输出流，关闭输出流除可以保证流的物理资源被回收之外，可能还可以将输出流缓冲区中的数据 flush 到物理节点里（因为在执行 close()方法之前，自动执行输出流的 flush()方法）。Java 的很多输出流默认都提供了缓冲功能，其实没有必要刻意去记忆哪些流有缓冲功能、哪些流没有，只要正常关闭所有的输出流即可保证程序正常。

如果希望直接输出字符串内容，则使用 Writer 会有更好的效果，如下程序所示。

程序清单：codes\15\15.3\FileWriterTest.java

```
public class FileWriterTest
{
    public static void main(String[] args)
    {
        try(
            FileWriter fw = new FileWriter("poem.txt"))
        {
            fw.write("锦瑟 - 李商隐\r\n");
            fw.write("锦瑟无端五十弦，一弦一柱思华年。\r\n");
            fw.write("庄生晓梦迷蝴蝶，望帝春心托杜鹃。\r\n");
            fw.write("沧海月明珠有泪，蓝田日暖玉生烟。\r\n");
            fw.write("此情可待成追忆，只是当时已惘然。\r\n");
        }
        catch (IOException ioe)
        {
            ioe.printStackTrace();
        }
    }
}
```

运行上面程序，将会在当前目录下输出一个 poem.txt 文件，文件内容就是程序中输出的内容。

注意：

上面程序在输出字符串内容时，字符串内容的最后是\r\n，这是 Windows 平台的换行符，通过这种方式就可以让输出内容换行；如果是 UNIX/Linux/BSD 等平台，则使用\n 就作为换行符。

15.4 输入/输出流体系

上一节介绍了输入/输出流的 4 个抽象基类，并介绍了 4 个访问文件的节点流的用法。通过上面示例程序不难发现，4 个基类使用起来有些烦琐。如果希望简化编程，这就需要借助于处理流了。

15.4.1 处理流的用法

图 15.7 显示了处理流的功能，它可以隐藏底层设备上节点流的差异，并对外提供更加方便的输入/输出方法，让程序员只需关心高级流的操作。

使用处理流时的典型思路是，使用处理流来包装节点流，程序通过处理流来执行输入/输出功能，让节点流与底层的 I/O 设备、文件交互。

实际识别处理流非常简单，只要流的构造器参数不是一个物理节点，而是已经存在的流，那么这种流就一定是处理流；而所有节点流都是直接以物理 IO 节点作为构造器参数的。

提示：

关于使用处理流的优势，归纳起来就是两点：①对开发人员来说，使用处理流进行输入/输出操作更简单；②使用处理流的执行效率更高。

下面程序使用 PrintStream 处理流来包装 OutputStream，使用处理流后的输出流在输出时将更加方便。

程序清单：codes\15\15.4\PrintStreamTest.java

```
public class PrintStreamTest
{
    public static void main(String[] args)
    {
        try(
            FileOutputStream fos = new FileOutputStream("test.txt");
            PrintStream ps = new PrintStream(fos))
        {
            // 使用 PrintStream 执行输出
            ps.println("普通字符串");
            // 直接使用 PrintStream 输出对象
            ps.println(new PrintStreamTest());
        }
        catch (IOException ioe)
        {
            ioe.printStackTrace();
        }
    }
}
```

上面程序中的两行粗体字代码先定义了一个节点输出流 FileOutputStream，然后程序使用 PrintStream 包装了该节点输出流，最后使用 PrintStream 输出字符串、输出对象……PrintStream 的输出功能非常强大，前面程序中一直使用的标准输出 System.out 的类型就是 PrintStream。

提示：

由于 PrintStream 类的输出功能非常强大，通常如果需要输出文本内容，都应该将输出流包装成 PrintStream 后进行输出。

从前面的代码可以看出，程序使用处理流非常简单，通常只需要在创建处理流时传入一个节点流作为构造器参数即可，这样创建的处理流就是包装了该节点流的处理流。

注意：

在使用处理流包装了底层节点流之后，关闭输入/输出流资源时，只要关闭最上层的处理流即可。关闭最上层的处理流时，系统会自动关闭被该处理流包装的节点流。

15.4.2 输入/输出流体系

Java 的输入/输出流体系提供了近 40 个类，这些类看上去杂乱而没有规律，但如果将其按功能进行分类，则不难发现其是非常规律的。表 15.1 显示了 Java 输入/输出流体系中常用的流分类。

从表 15.1 中可以看出，Java 的输入/输出流体系之所以如此复杂，主要是因为 Java 为了实现更好的设计，它把 IO 流按功能分成了许多类，而每类中又分别提供了字节流和字符流（当然有些流无法提供字节流，有些流无法提供字符流），字节流和字符流里又分别提供了输入流和输出流两大类，所以导致整个输入/输出流体系格外复杂。

表 15.1 Java 输入/输出流体系中常用的流分类

分 类	字节输入流	字节输出流	字符输入流	字符输出流
抽象基类	*InputStream*	*OutputStream*	*Reader*	*Writer*
访问文件	**FileInputStream**	**FileOutputStream**	**FileReader**	**FileWriter**
访问数组	**ByteArrayInputStream**	**ByteArrayOutputStream**	**CharArrayReader**	**CharArrayWriter**
访问管道	**PipedInputStream**	**PipedOutputStream**	**PipedReader**	**PipedWriter**
访问字符串			**StringReader**	**StringWriter**
缓冲流	BufferedInputStream	BufferedOutputStream	BufferedReader	BufferedWriter
转换流			InputStreamReader	OutputStreamWriter
对象流	ObjectInputStream	ObjectOutputStream		
抽象基类	*FilterInputStream*	*FilterOutputStream*	*FilterReader*	*FilterWriter*
打印流		PrintStream		PrintWriter
推回输入流	PushbackInputStream		PushbackReader	
特殊流	DataInputStream	DataOutputStream		

注：表 15.1 中的粗体字标出的类代表节点流，必须直接与指定的物理节点关联；斜体字标出的类代表抽象基类，无法直接创建实例。

通常来说，字节流的功能比字符流的功能强大，因为计算机里所有的数据都是二进制的，而字节流可以处理所有的二进制文件——但问题是，如果使用字节流来处理文本文件，则需要使用合适的方式把这些字节转换成字符，这就增加了编程的复杂度。所以通常有一个规则：如果进行输入/输出的内容是文本内容，则应该考虑使用字符流；如果进行输入/输出的内容是二进制内容，则应该考虑使用字节流。

提示：

计算机的文件常被分为文本文件和二进制文件两大类——所有能用记事本打开并看到其中字符内容的文件称为文本文件，反之则称为二进制文件。但实质是，计算机里的所有文件都是二进制文件，文本文件只是二进制文件的一种特例，当二进制文件里的内容恰好能被正常解析成字符时，则该二进制文件就变成了文本文件。更甚至于，即使是正常的文本文件，如果打开该文件时强制使用了“错误”的字符集，例如使用 EditPlus 打开刚刚生成的 poem.txt 文件时指定使用 UTF-8 字符集，如图 15.9 所示，则将看到打开的 poem.txt 文件内容变成了乱码。因此，如果希望看到正常的文本文件内容，则必须在打开文件时与保存文件时使用相同的字符集（Windows 下简体中文默认使用 GBK 字符集，而 Linux 下简体中文默认使用 UTF-8 字符集）。

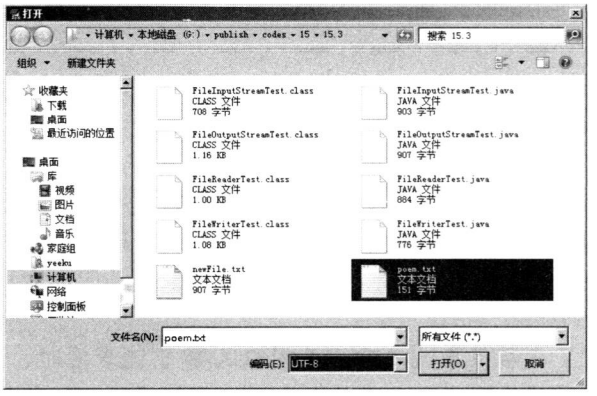


图 15.9 选择错误的字符集将导致文本文件变成“乱码”

表 15.1 仅仅总结了输入/输出流体系中位于 java.io 包下的流，还有一些诸如 AudioInputStream、CipherInputStream、DeflaterInputStream、ZipInputStream 等具有访问音频文件、加密/解密、压缩/解压等功能的字节流，它们具有特殊的功能，位于 JDK 的其他包下，本书不打算介绍这些特殊的 IO 流。

表 15.1 中还列出了一种以数组为物理节点的节点流，字节流以字节数组为节点，字符流以字符数组为节点；这种以数组为物理节点的节点流除在创建节点流对象时需要传入一个字节数组或者字符数组之外，用法上与文件节点流完全相似。与此类似的是，字符流还可以使用字符串作为物理节点，用于实现从字符串读取内容，或将内容写入字符串（用 StringBuffer 充当字符串）的功能。下面程序示范了使用字符串作为物理节点的字符输入/输出流的用法。

程序清单：codes\15\15.4\StringNodeTest.java

```
public class StringNodeTest
{
    public static void main(String[] args)
    {
        String src = "从明天起，做一个幸福的人\n"
            + "喂马，劈柴，周游世界\n"
            + "从明天起，关心粮食和蔬菜\n"
            + "我有一所房子，面朝大海，春暖花开\n"
            + "从明天起，和每一个亲人通信\n"
            + "告诉他们我的幸福\n";
        char[] buffer = new char[32];
        int hasRead = 0;
        try(
            StringReader sr = new StringReader(src))
        {
            // 采用循环读取的方式读取字符串
            while((hasRead = sr.read(buffer)) > 0)
            {
                System.out.print(new String(buffer ,0 , hasRead));
            }
        }
        catch (IOException ioe)
        {
            ioe.printStackTrace();
        }
        try(
            // 创建 StringWriter 时，实际上以一个 StringBuffer 作为输出节点
            // 下面指定的 20 就是 StringBuffer 的初始长度
            StringWriter sw = new StringWriter())
        {
            // 调用 StringWriter 的方法执行输出
            sw.write("有一个美丽的新世界，\n");
            sw.write("她在远方等我,\n");
            sw.write("那里有天真的孩子，\n");
            sw.write("还有姑娘的酒窝\n");
            System.out.println("----下面是 sw 字符串节点里的内容----");
            // 使用 toString()方法返回 StringWriter 字符串节点的内容
            System.out.println(sw.toString());
        }
        catch (IOException ex)
        {
            ex.printStackTrace();
        }
    }
}
```

上面程序与前面使用 FileReader 和 FileWriter 的程序基本相似，只是在创建 StringReader 和 StringWriter 对象时传入的是字符串节点，而不是文件节点。由于 String 是不可变的字符串对象，所以 StringWriter 使用 StringBuffer 作为输出节点。

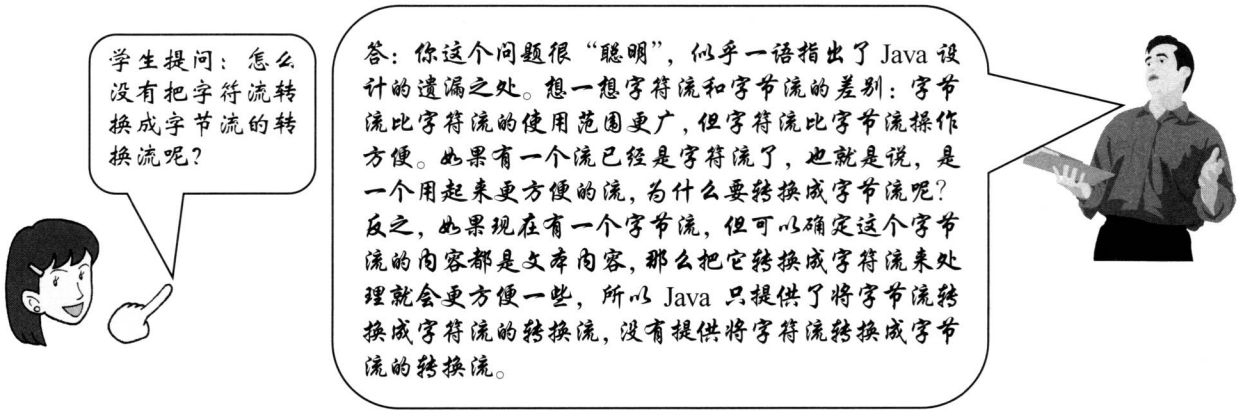
表 15.1 中列出了 4 个访问管道的流：PipedInputStream、PipedOutputStream、PipedReader、PipedWriter，它们都是用于实现进程之间通信功能的，分别是字节输入流、字节输出流、字符输入流和字符输出流。本书将在第 16 章介绍这 4 个流的用法。

表 15.1 中的 4 个缓冲流则增加了缓冲功能，增加缓冲功能可以提高输入、输出的效率，增加缓冲功能后需要使用 flush()才可以将缓冲区的内容写入实际的物理节点。

表 15.1 中的对象流主要用于实现对象的序列化，本章的 15.8 节将系统介绍对象序列化。

15.4.3 转换流

输入/输出流体系中还提供了两个转换流，这两个转换流用于实现将字节流转换成字符流，其中 InputStreamReader 将字节输入流转换成字符输入流，OutputStreamWriter 将字节输出流转换成字符输出流。



下面以获取键盘输入为例来介绍转换流的用法。Java 使用 System.in 代表标准输入，即键盘输入，但这个标准输入流是 InputStream 类的实例，使用不太方便，而且键盘输入内容都是文本内容，所以可以使用 InputStreamReader 将其转换成字符输入流，普通的 Reader 读取输入内容时依然不太方便，可以将普通的 Reader 再次包装成 BufferedReader，利用 BufferedReader 的 readLine()方法可以一次读取一行内容。如下程序所示。

程序清单：codes\15\15.4\KeyinTest.java

```
public class KeyinTest
{
    public static void main(String[] args)
    {
        try(
            // 将 Sytem.in 对象转换成 Reader 对象
            InputStreamReader reader = new InputStreamReader(System.in);
            // 将普通的 Reader 包装成 BufferedReader
            BufferedReader br = new BufferedReader(reader))
        {
            String line = null;
            // 采用循环方式来逐行地读取
            while ((line = br.readLine()) != null)
            {
                // 如果读取的字符串为"exit"，则程序退出
                if (line.equals("exit"))
                {
                    System.exit(1);
                }
                // 打印读取的内容
                System.out.println("输入内容为:" + line);
            }
        }
        catch (IOException ioe)
        {
            ioe.printStackTrace();
        }
    }
}
```

上面程序中的粗体字代码负责将 System.in 包装成 BufferedReader，BufferedReader 流具有缓冲功能，它可以一次读取一行文本——以换行符为标志，如果它没有读到换行符，则程序阻塞，等到读到换行符

为止。运行上面程序可以发现这个特征，在控制台执行输入时，只有按下回车键，程序才会打印出刚刚输入的内容。

提示：

由于 BufferedReader 具有一个 readLine()方法，可以非常方便地一次读入一行内容，所以经常把读取文本内容的输入流包装成 BufferedReader，用来方便地读取输入流的文本内容。

15.4.4 推回输入流

在输入/输出流体系中，有两个特殊的流与众不同，就是 PushbackInputStream 和 PushbackReader，它们都提供了如下三个方法。

- void unread(byte[]/char[] buf)：将一个字节/字符数组内容推回到推回缓冲区里，从而允许重复读取刚刚读取的内容。
- void unread(byte[]/char[] b, int off, int len)：将一个字节/字符数组里从 off 开始，长度为 len 字节/字符的内容推回到推回缓冲区里，从而允许重复读取刚刚读取的内容。
- void unread(int b)：将一个字节/字符推回到推回缓冲区里，从而允许重复读取刚刚读取的内容。

细心的读者可能已经发现了这三个方法与 InputStream 和 Reader 中的三个 read()方法一一对应，没错，这三个方法就是 PushbackInputStream 和 PushbackReader 的奥秘所在。

这两个推回输入流都带有一个推回缓冲区，当程序调用这两个推回输入流的 unread()方法时，系统将会把指定数组的内容推回到该缓冲区里，而推回输入流每次调用 read()方法时总是先从推回缓冲区读取，只有完全读取了推回缓冲区的内容后，但还没有装满 read()所需的数组时才会从原输入流中读取。图 15.10 显示了这种推回输入流的处理示意图。

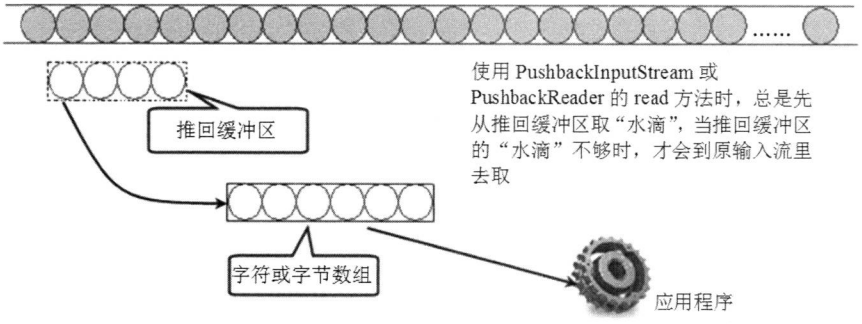


图 15.10 推回输入流的处理示意图

根据上面的介绍可以知道，当程序创建一个 PushbackInputStream 和 PushbackReader 时需要指定推回缓冲区的大小，默认的推回缓冲区的长度为 1。如果程序中推回到推回缓冲区的内容超出了推回缓冲区的大小，将会引发 Pushback buffer overflow 的 IOException 异常。

注意：

虽然图 15.10 中的推回缓冲区的长度看似比 read()方法的数组参数的长度小，但实际上，推回缓冲区的长度与 read()方法的数组参数的长度没有任何关系，完全可以更大。

下面程序试图找出程序中的"new PushbackReader"字符串，当找到该字符串后，程序只是打印出目标字符串之前的内容。

程序清单：codes\15\15.4\PushbackTest.java

```
public class PushbackTest
{
    public static void main(String[] args)
    {
        try(
            // 创建一个 PushbackReader 对象，指定推回缓冲区的长度为 64
```

```
            PushbackReader pr = new PushbackReader(new FileReader(
                "PushbackTest.java") , 64))
        {
            char[] buf = new char[32];
            // 用以保存上次读取的字符串内容
            String lastContent = "";
            int hasRead = 0;
            // 循环读取文件内容
            while ((hasRead = pr.read(buf)) > 0)
            {
                // 将读取的内容转换成字符串
                String content = new String(buf , 0 , hasRead);
                int targetIndex = 0;
                // 将上次读取的字符串和本次读取的字符串拼起来
                // 查看是否包含目标字符串，如果包含目标字符串
                if ((targetIndex = (lastContent + content)
                    .indexOf("new PushbackReader")) > 0)
                {
                    // 将本次内容和上次内容一起推回缓冲区
                    pr.unread((lastContent + content).toCharArray());
                    // 重新定义一个长度为 targetIndex 的 char 数组
                    if(targetIndex > 32)
                    {
                        buf = new char[targetIndex];
                    }
                    // 再次读取指定长度的内容（就是目标字符串之前的内容）
                    pr.read(buf , 0 , targetIndex);
                    // 打印读取的内容
                    System.out.print(new String(buf , 0 , targetIndex));
                    System.exit(0);
                }
                else
                {
                    // 打印上次读取的内容
                    System.out.print(lastContent);
                    // 将本次内容设为上次读取的内容
                    lastContent = content;
                }
            }
        }
        catch (IOException ioe)
        {
            ioe.printStackTrace();
        }
    }
}
```

上面程序中的粗体字代码实现了将指定内容推回到推回缓冲区，于是当程序再次调用 read()方法时，实际上只是读取了推回缓冲区的部分内容，从而实现了只打印目标字符串前面内容的功能。

15.5 重定向标准输入/输出

第 7 章介绍过，Java 的标准输入/输出分别通过 System.in 和 System.out 来代表，在默认情况下它们分别代表键盘和显示器，当程序通过 System.in 来获取输入时，实际上是从键盘读取输入；当程序试图通过 System.out 执行输出时，程序总是输出到屏幕。

在 System 类里提供了如下三个重定向标准输入/输出的方法。

- static void setErr(PrintStream err)：重定向“标准”错误输出流。
- static void setIn(InputStream in)：重定向“标准”输入流。
- static void setOut(PrintStream out)：重定向“标准”输出流。

下面程序通过重定向标准输出流，将 System.out 的输出重定向到文件输出，而不是在屏幕上输出。

程序清单：codes\15\15.5\RedirectOut.java

```
public class RedirectOut
{
    public static void main(String[] args)
    {
        try(
            // 一次性创建 PrintStream 输出流
            PrintStream ps = new PrintStream(new FileOutputStream("out.txt")))
        {
            // 将标准输出重定向到 ps 输出流
            System.setOut(ps);
            // 向标准输出输出一个字符串
            System.out.println("普通字符串");
            // 向标准输出输出一个对象
            System.out.println(new RedirectOut());
        }
        catch (IOException ex)
        {
            ex.printStackTrace();
        }
    }
}
```

上面程序中的粗体字代码创建了一个 PrintStream 输出流，并将系统的标准输出重定向到该 PrintStream 输出流。运行上面程序时将看不到任何输出——这意味着标准输出不再输出到屏幕，而是输出到 out.txt 文件，运行结束后，打开系统当前路径下的 out.txt 文件，即可看到文件里的内容，正好与程序中的输出一致。

下面程序重定向标准输入，从而可以将 System.in 重定向到指定文件，而不是键盘输入。

程序清单：codes\15\15.5\RedirectIn.java

```
public class RedirectIn
{
    public static void main(String[] args)
    {
        try(
            FileInputStream fis = new FileInputStream("RedirectIn.java"))
        {
            // 将标准输入重定向到 fis 输入流
            System.setIn(fis);
            // 使用 System.in 创建 Scanner 对象，用于获取标准输入
            Scanner sc = new Scanner(System.in);
            // 增加下面一行只把回车作为分隔符
            sc.useDelimiter("\n");
            // 判断是否还有下一个输入项
            while(sc.hasNext())
            {
                // 输出输入项
                System.out.println("键盘输入的内容是：" + sc.next());
            }
        }
        catch (IOException ex)
        {
            ex.printStackTrace();
        }
    }
}
```

上面程序中的粗体字代码创建了一个 FileInputStream 输入流，并使用 System 的 setIn()方法将系统标准输入重定向到该文件输入流。运行上面程序，程序不会等待用户输入，而是直接输出了 RedirectIn.java 文件的内容，这表明程序不再使用键盘作为标准输入，而是使用 RedirectIn.java 文件作为标准输入源。

15.6 Java 虚拟机读写其他进程的数据

在第 7 章已经介绍过，使用 Runtime 对象的 exec()方法可以运行平台上的其他程序，该方法产生一

个 Process 对象，Process 对象代表由该 Java 程序启动的子进程。Process 类提供了如下三个方法，用于让程序和其子进程进行通信。

- InputStream getErrorStream()：获取子进程的错误流。
- InputStream getInputStream()：获取子进程的输入流。
- OutputStream getOutputStream()：获取子进程的输出流。

注意：

此处的输入流、输出流非常容易混淆，如果试图让子进程读取程序中的数据，那么应该用输入流还是输出流？不是输入流，而是输出流。要站在 Java 程序的角度来看问题，子进程读取 Java 程序的数据，就是让 Java 程序把数据输出到子进程中（就像把数据输出到文件中一样，只是现在由子进程节点代替了文件节点），所以应该使用输出流。

下面程序示范了读取其他进程的输出信息。

程序清单：codes\15\15.6\ReadFromProcess.java

```
public class ReadFromProcess
{
    public static void main(String[] args)
        throws IOException
    {
        // 运行 javac 命令，返回运行该命令的子进程
        Process p = Runtime.getRuntime().exec("javac");
        try(
            // 以 p 进程的错误流创建 BufferedReader 对象
            // 这个错误流对本程序是输入流，对 p 进程则是输出流
            BufferedReader br = new BufferedReader(new
                InputStreamReader(p.getErrorStream())))
        {
            String buff = null;
            // 采取循环方式来读取 p 进程的错误输出
            while((buff = br.readLine()) != null)
            {
                System.out.println(buff);
            }
        }
    }
}
```

上面程序中的第一行粗体字代码使用 Runtime 启动了 javac 程序，获得了运行该程序对应的子进程；第二行粗体字代码以 p 进程的错误输入流创建了 BufferedReader，这个输入流的流向如图 15.11 所示。

如图 15.11 所示的数据流对 p 进程（javac 进程）而言，它是输出流；但对本程序（ReadFromProcess）而言，它是输入流——衡量输入、输出时总是站在运行本程序所在内存的角度，所以该数据流应该是输入流。运行上面程序，会看到如图 15.12 所示的运行窗口。

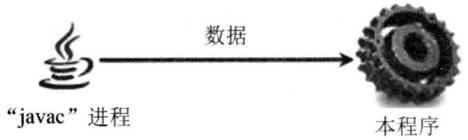


图 15.11　数据从 p 进程流向本程序所在的内存

图 15.12　Java 程序获得 javac 命令的错误输出

不仅如此，也可以通过 Process 的 getOutputStream()方法获得向进程输入数据的流（该流对 Java 程序是输出流，对子进程则是输入流），如下程序实现了在 Java 程序中启动 Java 虚拟机运行另一个 Java 程序，并向另一个 Java 程序中输入数据。

程序清单：codes\15\15.6\WriteToProcess.java

```
public class WriteToProcess
{
    public static void main(String[] args)
        throws IOException
    {
        // 运行 java ReadStandard 命令，返回运行该命令的子进程
        Process p = Runtime.getRuntime().exec("java ReadStandard");
        try(
            // 以 p 进程的输出流创建 PrintStream 对象
            // 这个输出流对本程序是输出流，对 p 进程则是输入流
            PrintStream ps = new PrintStream(p.getOutputStream()))
        {
            // 向 ReadStandard 程序写入内容，这些内容将被 ReadStandard 读取
            ps.println("普通字符串");
            ps.println(new WriteToProcess());
        }
    }
}
// 定义一个 ReadStandard 类，该类可以接收标准输入
// 并将标准输入写入 out.txt 文件
class ReadStandard
{
    public static void main(String[] args)
    {
        try(
            // 使用 System.in 创建 Scanner 对象，用于获取标准输入
            Scanner sc = new Scanner(System.in);
            PrintStream ps = new PrintStream(
            new FileOutputStream("out.txt")))
        {
            // 增加下面一行只把回车作为分隔符
            sc.useDelimiter("\n");
            // 判断是否还有下一个输入项
            while(sc.hasNext())
            {
                // 输出输入项
                ps.println("键盘输入的内容是：" + sc.next());
            }
        }
        catch(IOException ioe)
        {
            ioe.printStackTrace();
        }
    }
}
```

上面程序中的 ReadStandard 是一个使用 Scanner 获取标准输入的类，该类提供了 main()方法，可以被运行——但此处不打算直接运行该类，而是由 WriteToProcess 类来运行 ReadStandard 类。在程序的第一行粗体字代码中，程序使用 Runtime 的 exec()方法运行了 java ReadStandard 命令，该命令将运行 ReadStandard 类，并返回运行该程序的子进程；程序的第二行粗体字代码获得进程 p 的输出流——该输出流对进程 p 是输入流，只是对本程序是输出流，程序通过该输出流向进程 p（也就是 ReadStandard 程序）输出数据，这些数据将被 ReadStandard 类读到。

运行上面的 WriteToProcess 类，程序运行结束将看到产生了一个 out.txt 文件，该文件由 ReadStandard 类产生，该文件的内容由 WriteToProcess 类写入 ReadStandard 进程里，并由 ReadStandard 读取这些数据，并将这些数据保存到 out.txt 文件中。

15.7 RandomAccessFile

RandomAccessFile是Java输入/输出流体系中功能最丰富的文件内容访问类，它提供了众多的方法来访问文件内容，它既可以读取文件内容，也可以向文件输出数据。与普通的输入/输出流不同的是，RandomAccessFile支持“随机访问”的方式，程序可以直接跳转到文件的任意地方来读写数据。

由于RandomAccessFile可以自由访问文件的任意位置，所以如果只需要访问文件部分内容，而不是把文件从头读到尾，使用RandomAccessFile将是更好的选择。

与OutputStream、Writer等输出流不同的是，RandomAccessFile允许自由定位文件记录指针，RandomAccessFile可以不从开始的地方开始输出，因此RandomAccessFile可以向已存在的文件后追加内容。如果程序需要向已存在的文件后追加内容，则应该使用RandomAccessFile。

RandomAccessFile的方法虽然多，但它有一个最大的局限，就是只能读写文件，不能读写其他IO节点。

RandomAccessFile对象也包含了一个记录指针，用以标识当前读写处的位置，当程序新创建一个RandomAccessFile对象时，该对象的文件记录指针位于文件头（也就是0处），当读/写了n个字节后，文件记录指针将会向后移动n个字节。除此之外，RandomAccessFile可以自由移动该记录指针，既可以向前移动，也可以向后移动。RandomAccessFile包含了如下两个方法来操作文件记录指针。

- long getFilePointer()：返回文件记录指针的当前位置。
- void seek(long pos)：将文件记录指针定位到pos位置。

RandomAccessFile既可以读文件，也可以写，所以它既包含了完全类似于InputStream的三个read()方法，其用法和InputStream的三个read()方法完全一样；也包含了完全类似于OutputStream的三个write()方法，其用法和OutputStream的三个write()方法完全一样。除此之外，RandomAccessFile还包含了一系列的readXxx()和writeXxx()方法来完成输入、输出。

提示：

计算机里的“随机访问”是一个很奇怪的词，对于汉语而言，随机访问是具有不确定性的——具有一会儿访问这里，一会儿访问那里的意思，如果按这种方式来理解“随机访问”，那么就会对所谓的“随机访问”方式感到十分迷惑，这也是十多年前我刚接触RAM（Random Access Memory，即内存）感到万分迷惑的地方。实际上，“随机访问”是由Random Access两个单词翻译而来，而Random在英语里不仅有随机的意思，还有任意的意思——如果能这样理解Random，就可以更好地理解Random Access了——应该是任意访问，而不是随机访问，也就是说，RAM是可以自由访问任意存储点的存储器（与磁盘、磁带等需要寻道、倒带才可访问指定存储点等存储器相区分）；而RandomAccessFile的含义是可以自由访问文件的任意地方（与InputStream、Reader需要依次向后读取相区分），所以RandomAccessFile的含义决不是“随机访问”，而应该是“任意访问”。在后来的日子里，我无数次发现一些计算机专业术语翻译得如此让人深恶痛绝，于是造成了很多人觉得IT行业较难的后果；再后来，我决定尽量少看被翻译后的IT技术文章，要么看原版IT技术文章，要么就直接看国内的IT技术文章。

RandomAccessFile类有两个构造器，其实这两个构造器基本相同，只是指定文件的形式不同而已——一个使用String参数来指定文件名，一个使用File参数来指定文件本身。除此之外，创建RandomAccessFile对象时还需要指定一个mode参数，该参数指定RandomAccessFile的访问模式，该参数有如下4个值。

- "r"：以只读方式打开指定文件。如果试图对该RandomAccessFile执行写入方法，都将抛出IOException异常。
- "rw"：以读、写方式打开指定文件。如果该文件尚不存在，则尝试创建该文件。
- "rws"：以读、写方式打开指定文件。相对于"rw"模式，还要求对文件的内容或元数据的每个更新都同步写入到底层存储设备。

- "rwd"：以读、写方式打开指定文件。相对于"rw"模式，还要求对文件内容的每个更新都同步写入到底层存储设备。

下面程序使用了 RandomAccessFile 来访问指定的中间部分数据。

程序清单：codes\15\15.7\RandomAccessFileTest.java

```
public class RandomAccessFileTest
{
    public static void main(String[] args)
    {
        try(
            RandomAccessFile raf = new RandomAccessFile(
                "RandomAccessFileTest.java" , "r"))
        {
            // 获取 RandomAccessFile 对象文件指针的位置，初始位置是 0
            System.out.println("RandomAccessFile 的文件指针的初始位置："
                + raf.getFilePointer());
            // 移动 raf 的文件记录指针的位置
            raf.seek(300);
            byte[] bbuf = new byte[1024];
            // 用于保存实际读取的字节数
            int hasRead = 0;
            // 使用循环来重复“取水”过程
            while ((hasRead = raf.read(bbuf)) > 0 )
            {
                // 取出“竹筒”中的水滴（字节），将字节数组转换成字符串输入
                System.out.print(new String(bbuf , 0 , hasRead ));
            }
        }
        catch (IOException ex)
        {
            ex.printStackTrace();
        }
    }
}
```

上面程序中的第一行粗体代码创建了一个 RandomAccessFile 对象，该对象以只读方式打开了RandomAccessFileTest.java 文件，这意味着该RandomAccessFile对象只能读取文件内容，不能执行写入。

程序中第二行粗体字代码将文件记录指针定位到 300 处，也就是说，程序将从 300 字节处开始读、写，程序接下来的部分与使用 InputStream 读取并没有太大的区别。运行上面程序，将看到程序只读取后面部分的效果。

下面程序示范了如何向指定文件后追加内容，为了追加内容，程序应该先将记录指针移动到文件最后，然后开始向文件中输出内容。

程序清单：codes\15\15.7\AppendContent.java

```
public class AppendContent
{
    public static void main(String[] args)
    {
        try(
            // 以读、写方式打开一个 RandomAccessFile 对象
            RandomAccessFile raf = new RandomAccessFile("out.txt" , "rw"))
        {
            // 将记录指针移动到 out.txt 文件的最后
            raf.seek(raf.length());
            raf.write("追加的内容！\r\n".getBytes());
        }
        catch (IOException ex)
        {
            ex.printStackTrace();
        }
    }
}
```

上面程序中的第一行粗体字代码先以读、写方式创建了一个 RandomAccessFile 对象，第二行粗体

字代码将 RandomAccessFile 对象的记录指针移动到最后；接下来使用 RandomAccessFile 执行输出，与使用 OutputStream 或 Writer 执行输出并没有太大区别。

每运行上面程序一次，都可以看到 out.txt 文件中多一行“追加的内容!”字符串，程序在该字符串后使用“\r\n”是为了控制换行。

注意：

RandomAccessFile 依然不能向文件的指定位置插入内容，如果直接将文件记录指针移动到中间某位置后开始输出，则新输出的内容会覆盖文件中原有的内容。如果需要向指定位置插入内容，程序需要先把插入点后面的内容读入缓冲区，等把需要插入的数据写入文件后，再将缓冲区的内容追加到文件后面。

下面程序实现了向指定文件、指定位置插入内容的功能。

程序清单：codes\15\15.7\InsertContent.java

```
public class InsertContent
{
    public static void insert(String fileName , long pos
        , String insertContent) throws IOException
    {
        File tmp = File.createTempFile("tmp" , null);
        tmp.deleteOnExit();
        try(
            RandomAccessFile raf = new RandomAccessFile(fileName , "rw");
            // 使用临时文件来保存插入点后的数据
            FileOutputStream tmpOut = new FileOutputStream(tmp);
            FileInputStream tmpIn = new FileInputStream(tmp))
        {
            raf.seek(pos);
            // ------下面代码将插入点后的内容读入临时文件中保存------
            byte[] bbuf = new byte[64];
            // 用于保存实际读取的字节数
            int hasRead = 0;
            // 使用循环方式读取插入点后的数据
            while ((hasRead = raf.read(bbuf)) > 0 )
            {
                // 将读取的数据写入临时文件
                tmpOut.write(bbuf , 0 , hasRead);
            }
            // ----------下面代码用于插入内容----------
            // 把文件记录指针重新定位到 pos 位置
            raf.seek(pos);
            // 追加需要插入的内容
            raf.write(insertContent.getBytes());
            // 追加临时文件中的内容
            while ((hasRead = tmpIn.read(bbuf)) > 0 )
            {
                raf.write(bbuf , 0 , hasRead);
            }
        }
    }
    public static void main(String[] args)
        throws IOException
    {
        insert("InsertContent.java" , 45 , "插入的内容\r\n");
    }
}
```

上面程序中使用 File 的 createTempFile(String prefix, String suffix)方法创建了一个临时文件（该临时文件将在 JVM 退出时被删除），用以保存被插入文件的插入点后面的内容。程序先将文件中插入点后的内容读入临时文件中，然后重新定位到插入点，将需要插入的内容添加到文件后面，最后将临时文件的

内容添加到文件后面，通过这个过程就可以向指定文件、指定位置插入内容。

每次运行上面程序，都会看到向 InsertContent.java 中插入了一行字符串。

提示：

多线程断点的网络下载工具（如 FlashGet 等）就可通过 RandomAccessFile 类来实现，所有的下载工具在下载开始时都会建立两个文件：一个是与被下载文件大小相同的空文件，一个是记录文件指针的位置文件，下载工具用多条线程启动输入流来读取网络数据，并使用 RandomAccessFile 将从网络上读取的数据写入前面建立的空文件中，每写一些数据后，记录文件指针的文件就分别记下每个 RandomAccessFile 当前的文件指针位置——网络断开后，再次开始下载时，每个 RandomAccessFile 都根据记录文件指针的文件中记录的位置继续向下写数据。本书将会在介绍多线程和网络知识之后，更加详细地介绍如何开发类似于 FlashGet 的多线程断点传输工具。

15.8 Java 9 改进的对象序列化

对象序列化的目标是将对象保存到磁盘中，或允许在网络中直接传输对象。对象序列化机制允许把内存中的 Java 对象转换成平台无关的二进制流，从而允许把这种二进制流持久地保存在磁盘上，通过网络将这种二进制流传输到另一个网络节点。其他程序一旦获得了这种二进制流（无论是从磁盘中获取的，还是通过网络获取的），都可以将这种二进制流恢复成原来的 Java 对象。

15.8.1 序列化的含义和意义

序列化机制允许将实现序列化的 Java 对象转换成字节序列，这些字节序列可以保存在磁盘上，或通过网络传输，以备以后重新恢复成原来的对象。序列化机制使得对象可以脱离程序的运行而独立存在。

对象的序列化（Serialize）指将一个 Java 对象写入 IO 流中，与此对应的是，对象的反序列化（Deserialize）则指从 IO 流中恢复该 Java 对象。

Java 9 增强了对象序列化机制，它允许对读入的序列化数据进行过滤，这种过滤可在反序列化之前对数据执行校验，从而提高安全性和健壮性。

如果需要让某个对象支持序列化机制，则必须让它的类是可序列化的（serializable）。为了让某个类是可序列化的，该类必须实现如下两个接口之一。

- Serializable
- Externalizable

Java 的很多类已经实现了 Serializable，该接口是一个标记接口，实现该接口无须实现任何方法，它只是表明该类的实例是可序列化的。

所有可能在网络上传输的对象的类都应该是可序列化的，否则程序将会出现异常，比如 RMI（Remote Method Invoke，即远程方法调用，是 Java EE 的基础）过程中的参数和返回值；所有需要保存到磁盘里的对象的类都必须可序列化，比如 Web 应用中需要保存到 HttpSession 或 ServletContext 属性的 Java 对象。

因为序列化是 RMI 过程的参数和返回值都必须实现的机制，而 RMI 又是 Java EE 技术的基础——所有的分布式应用常常需要跨平台、跨网络，所以要求所有传递的参数、返回值必须实现序列化。因此序列化机制是 Java EE 平台的基础。通常建议：程序创建的每个 JavaBean 类都实现 Serializable。

15.8.2 使用对象流实现序列化

如果需要将某个对象保存到磁盘上或者通过网络传输，那么这个类应该实现 Serializable 接口或者 Externalizable 接口之一。关于这两个接口的区别和联系，后面将有更详细的介绍，读者先不去理会 Externalizable 接口。

使用 Serializable 来实现序列化非常简单，主要让目标类实现 Serializable 标记接口即可，无须实现任何方法。

一旦某个类实现了 Serializable 接口，该类的对象就是可序列化的，程序可以通过如下两个步骤来

序列化该对象。

① 创建一个 ObjectOutputStream，这个输出流是一个处理流，所以必须建立在其他节点流的基础之上。如下代码所示。

```
// 创建个 ObjectOutputStream 输出流
ObjectOutputStream oos = new ObjectOutputStream(
    new FileOutputStream("object.txt"));
```

② 调用 ObjectOutputStream 对象的 writeObject()方法输出可序列化对象，如下代码所示。

```
// 将一个 Person 对象输出到输出流中
oos.writeObject(per);
```

下面程序定义了一个 Person 类，这个 Person 类就是一个普通的 Java 类，只是实现了 Serializable 接口，该接口标识该类的对象是可序列化的。

程序清单：codes\15\15.8\Person.java

```
public class Person
    implements java.io.Serializable
{
    private String name;
    private int age;
    // 注意此处没有提供无参数的构造器
    public Person(String name , int age)
    {
        System.out.println("有参数的构造器");
        this.name = name;
        this.age = age;
    }
    // 省略 name 与 age 的 setter 和 getter 方法
    ...
}
```

下面程序使用 ObjectOutputStream 将一个 Person 对象写入磁盘文件。

程序清单：codes\15\15.8\WriteObject.java

```
public class WriteObject
{
    public static void main(String[] args)
    {
        try(
            // 创建一个 ObjectOutputStream 输出流
            ObjectOutputStream oos = new ObjectOutputStream(
                new FileOutputStream("object.txt")))
        {
            Person per = new Person("孙悟空", 500);
            // 将 per 对象写入输出流
            oos.writeObject(per);
        }
        catch (IOException ex)
        {
            ex.printStackTrace();
        }
    }
}
```

上面程序中的第一行粗体字代码创建了一个 ObjectOutputStream 输出流，这个 ObjectOutputStream 输出流建立在一个文件输出流的基础之上；程序第二行粗体字代码使用 writeObject()方法将一个 Person 对象写入输出流。运行上面程序，将会看到生成了一个 object.txt 文件，该文件的内容就是 Person 对象。

如果希望从二进制流中恢复 Java 对象，则需要使用反序列化。反序列化的步骤如下。

① 创建一个 ObjectInputStream 输入流，这个输入流是一个处理流，所以必须建立在其他节点流的基础之上。如下代码所示。

```
// 创建一个 ObjectInputStream 输入流
ObjectInputStream ois = new ObjectInputStream(
    new FileInputStream("object.txt"));
```

② 调用 ObjectInputStream 对象的 readObject()方法读取流中的对象，该方法返回一个 Object 类型的 Java 对象，如果程序知道该 Java 对象的类型，则可以将该对象强制类型转换成其真实的类型。如下代码所示。

```
// 从输入流中读取一个 Java 对象，并将其强制类型转换为 Person 类
Person p = (Person)ois.readObject();
```

下面程序示范了从刚刚生成的 object.txt 文件中读取 Person 对象的步骤。

程序清单：codes\15\15.8\ReadObject.java

```
public class ReadObject
{
    public static void main(String[] args)
    {
        try(
            // 创建一个 ObjectInputStream 输入流
            ObjectInputStream ois = new ObjectInputStream(
                new FileInputStream("object.txt")))
        {
            // 从输入流中读取一个 Java 对象，并将其强制类型转换为 Person 类
            Person p = (Person)ois.readObject();
            System.out.println("名字为: " + p.getName()
                + "\n 年龄为: " + p.getAge());
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
        }
    }
}
```

上面程序中第一行粗体字代码将一个文件输入流包装成 ObjectInputStream 输入流，第二行粗体字代码使用 readObject()读取了文件中的 Java 对象，这就完成了反序列化过程。

必须指出的是，反序列化读取的仅仅是 Java 对象的数据，而不是 Java 类，因此采用反序列化恢复 Java 对象时，必须提供该 Java 对象所属类的 class 文件，否则将会引发 ClassNotFoundException 异常。

还有一点需要指出：Person 类只有一个有参数的构造器，没有无参数的构造器，而且该构造器内有一个普通的打印语句。当反序列化读取 Java 对象时，并没有看到程序调用该构造器，这表明反序列化机制无须通过构造器来初始化 Java 对象。

提示：

在 ObjectInputStream 输入流中的 readObject()方法声明抛出了 ClassNotFoundException 异常，也就是说，当反序列化时找不到对应的 Java 类时将会引发该异常。

如果使用序列化机制向文件中写入了多个 Java 对象，使用反序列化机制恢复对象时必须按实际写入的顺序读取。

当一个可序列化类有多个父类时（包括直接父类和间接父类），这些父类要么有无参数的构造器，要么也是可序列化的——否则反序列化时将抛出 InvalidClassException 异常。如果父类是不可序列化的，只是带有无参数的构造器，则该父类中定义的成员变量值不会序列化到二进制流中。

➤➤ 15.8.3 对象引用的序列化

前面介绍的 Person 类的两个成员变量分别是 String 类型和 int 类型，如果某个类的成员变量的类型不是基本类型或 String 类型，而是另一个引用类型，那么这个引用类必须是可序列化的，否则拥有该类型成员变量的类也是不可序列化的。

如下 Teacher 类持有一个 Person 类的引用，只有 Person 类是可序列化的，Teacher 类才是可序列化的。如果 Person 类不可序列化，则无论 Teacher 类是否实现 Serilizable、Externalizable 接口，则 Teacher 类都是不可序列化的。

程序清单：codes\15\15.8\Teacher.java

```
public class Teacher
    implements java.io.Serializable
{
    private String name;
    private Person student;
    public Teacher(String name , Person student)
    {
        this.name = name;
        this.student = student;
    }
    // 此处省略了 name 与 student 的 setter 和 getter 方法
    ...
}
```

注意：

当程序序列化一个 Teacher 对象时，如果该 Teacher 对象持有一个 Person 对象的引用，为了在反序列化时可以正常恢复该 Teacher 对象，程序会顺带将该 Person 对象也进行序列化，所以 Person 类也必须是可序列化的，否则 Teacher 类将不可序列化。

现在假设有如下一种特殊情形：程序中有两个 Teacher 对象，它们的 student 实例变量都引用到同一个 Person 对象，而且该 Person 对象还有一个引用变量引用它。如下代码所示。

```
Person per = new Person("孙悟空", 500);
Teacher t1 = new Teacher("唐僧" , per);
Teacher t2 = new Teacher("菩提祖师" , per);
```

上面代码创建了两个 Teacher 对象和一个 Person 对象，这三个对象在内存中的存储示意图如图 15.13 所示。

这里产生了一个问题——如果先序列化 t1 对象，则系统将该 t1 对象所引用的 Person 对象一起序列化；如果程序再序列化 t2 对象，系统将一样会序列化该 t2 对象，并且将再次序列化该 t2 对象所引用的 Person 对象；如果程序再显式序列化 per 对象，系统将再次序列化该 Person 对象。这个过程似乎会向输出流中输出三个 Person 对象。

如果系统向输出流中写入了三个 Person 对象，那么后果是当程序从输入流中反序列化这些对象时，将会得到三个 Person 对象，从而引起 t1 和 t2 所引用的 Person 对象不是同一个对象，这显然与图 15.13 所示的效果不一致——这也就违背了 Java 序列化机制的初衷。

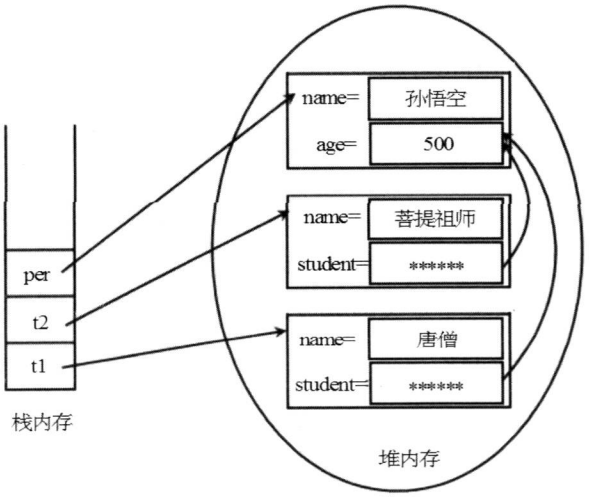


图 15.13　两个 Teacher 对象的 Student 实例变量都引用到同一个 Person 对象

所以，Java 序列化机制采用了一种特殊的序列化算法，其算法内容如下。

- 所有保存到磁盘中的对象都有一个序列化编号。

- 当程序试图序列化一个对象时，程序将先检查该对象是否已经被序列化过，只有该对象从未（在本次虚拟机中）被序列化过，系统才会将该对象转换成字节序列并输出。
- 如果某个对象已经序列化过，程序将只是直接输出一个序列化编号，而不是再次重新序列化该对象。

根据上面的序列化算法，可以得到一个结论——当第二次、第三次序列化 Person 对象时，程序不会再次将 Person 对象转换成字节序列并输出，而是仅仅输出一个序列化编号。假设有如下顺序的序列化代码：

```
oos.writeObject(t1);
oos.writeObject(t2);
oos.writeObject(per);
```

上面代码依次序列化了 t1、t2 和 per 对象，序列化后磁盘文件的存储示意图如图 15.14 所示。

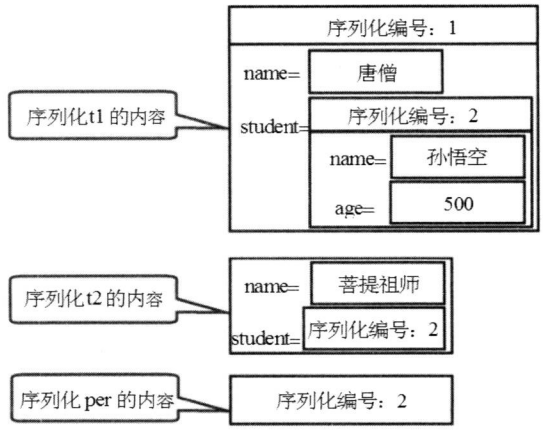


图 15.14　序列化机制示意图

通过图 15.14 可以很好地理解 Java 序列化的底层机制，通过该机制不难看出，当多次调用 writeObject()方法输出同一个对象时，只有第一次调用 writeObject()方法时才会将该对象转换成字节序列并输出。

下面程序序列化了两个 Teacher 对象，两个 Teacher 对象都持有一个引用到同一个 Person 对象的引用，而且程序两次调用 writeObject()方法输出同一个 Teacher 对象。

程序清单：codes\15\15.8\WriteTeacher.java

```
public class WriteTeacher
{
    public static void main(String[] args)
    {
        try(
            // 创建一个 ObjectOutputStream 输出流
            ObjectOutputStream oos = new ObjectOutputStream(
                new FileOutputStream("teacher.txt")))
        {
            Person per = new Person("孙悟空", 500);
            Teacher t1 = new Teacher("唐僧" , per);
            Teacher t2 = new Teacher("菩提祖师" , per);
            // 依次将 4 个对象写入输出流
            oos.writeObject(t1);
            oos.writeObject(t2);
            oos.writeObject(per);
            oos.writeObject(t2);
        }
        catch (IOException ex)
        {
            ex.printStackTrace();
        }
    }
}
```

上面程序中的粗体字代码 4 次调用了 writeObject()方法来输出对象，实际上只序列化了三个对象，而且序列的两个 Teacher 对象的 student 引用实际是同一个 Person 对象。下面程序读取序列化文件中的对象即可证明这一点。

程序清单：codes\15\15.8\ReadTeacher.java

```
public class ReadTeacher
{
    public static void main(String[] args)
    {
        try(
            // 创建一个 ObjectInputStream 输入流
```

```
            ObjectInputStream ois = new ObjectInputStream(
                new FileInputStream("teacher.txt")))
        {
            // 依次读取 ObjectInputStream 输入流中的 4 个对象
            Teacher t1 = (Teacher)ois.readObject();
            Teacher t2 = (Teacher)ois.readObject();
            Person p = (Person)ois.readObject();
            Teacher t3 = (Teacher)ois.readObject();
            // 输出 true
            System.out.println("t1 的 student 引用和 p 是否相同: "
                + (t1.getStudent() == p));
            // 输出 true
            System.out.println("t2 的 student 引用和 p 是否相同: "
                + (t2.getStudent() == p));
            // 输出 true
            System.out.println("t2 和 t3 是否是同一个对象: "
                + (t2 == t3));
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
        }
    }
}
```

上面程序中的粗体字代码依次读取了序列化文件中的 4 个 Java 对象，但通过后面比较判断，不难发现 t2 和 t3 是同一个 Java 对象，t1 的 student 引用的、t2 的 student 引用的和 p 引用变量引用的也是同一个 Java 对象——这证明了图 15.14 所示的序列化机制。

由于 Java 序列化机制使然：如果多次序列化同一个 Java 对象时，只有第一次序列化时才会把该 Java 对象转换成字节序列并输出，这样可能引起一个潜在的问题——当程序序列化一个可变对象时，只有第一次使用 writeObject()方法输出时才会将该对象转换成字节序列并输出，当程序再次调用 writeObject()方法时，程序只是输出前面的序列化编号，即使后面该对象的实例变量值已被改变，改变的实例变量值也不会被输出。如下程序所示。

程序清单：codes\15\15.8\SerializeMutable.java

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

        try(
            // 创建一个 ObjectOutputStream 输出流
            ObjectOutputStream oos = new ObjectOutputStream(
                new FileOutputStream("mutable.txt"));
            // 创建一个 ObjectInputStream 输入流
            ObjectInputStream ois = new ObjectInputStream(
                new FileInputStream("mutable.txt")))
        {
            Person per = new Person("孙悟空", 500);
            // 系统将 per 对象转换成字节序列并输出
            oos.writeObject(per);
            // 改变 per 对象的 name 实例变量的值
            per.setName("猪八戒");
            // 系统只是输出序列化编号，所以改变后的 name 不会被序列化
            oos.writeObject(per);
            Person p1 = (Person)ois.readObject();    // ①
            Person p2 = (Person)ois.readObject();    // ②
            // 下面输出 true，即反序列化后 p1 等于 p2
            System.out.println(p1 == p2);
            // 下面依然看到输出“孙悟空”，即改变后的实例变量没有被序列化
            System.out.println(p2.getName());
        }
        catch (Exception ex)
        {
```

```
            ex.printStackTrace();
        }
    }
}
```

程序中第一段粗体字代码先使用 writeObject()方法写入了一个 Person 对象，接着程序改变了 Person 对象的 name 实例变量值，然后程序再次输出 Person 对象，但这次的输出已经不会将 Person 对象转换成字节序列并输出了，而是仅仅输出了一个序列化编号。

程序中①②号粗体字代码两次调用 readObject()方法读取了序列化文件中的 Java 对象，比较两次读取的 Java 对象将完全相同，程序输出第二次读取的 Person 对象的 name 实例变量的值依然是“孙悟空”，表明改变后的 Person 对象并没有被写入——这与 Java 序列化机制相符。

注意：

当使用 Java 序列化机制序列化可变对象时一定要注意，只有第一次调用 wirteObject()方法来输出对象时才会将对象转换成字节序列，并写入到 ObjectOutputStream；在后面程序中即使该对象的实例变量发生了改变，再次调用 writeObject()方法输出该对象时，改变后的实例变量也不会被输出。

15.8.4 Java 9 增加的过滤功能

Java 9 为 ObjectInputStream 增加了 setObjectInputFilter()、getObjectInputFilter()两个方法，其中第一个方法用于为对象输入流设置过滤器。当程序通过 ObjectInputStream 反序列化对象时，过滤器的 checkInput()方法会被自动激发，用于检查序列化数据是否有效。

使用 checkInput()方法检查序列化数据时有 3 种返回值。

- Status.REJECTED：拒绝恢复。
- Status.ALLOWED：允许恢复。
- Status.UNDECIDED：未决定状态，程序继续执行检查。

ObjectInputStream 将会根据 ObjectInputFilter 的检查结果来决定是否执行反序列化，如果 checkInput()方法返回 Status.REJECTED，反序列化将会被阻止；如果 checkInput()方法返回 Status.ALLOWED，程序将可执行反序列化。

下面程序对前的 ReadObject.java 程序进行改进，该程序将会在反序列化之前对数据执行检查。

程序清单：codes\15\15.8\FilterTest.java

```
public class FilterTest
{
    public static void main(String[] args)
    {
        try(
            // 创建一个 ObjectInputStream 输入流
            ObjectInputStream ois = new ObjectInputStream(
                new FileInputStream("object.txt")))
        {
            ois.setObjectInputFilter((info) -> {
                System.out.println("===执行数据过滤===");
                ObjectInputFilter serialFilter =
                    ObjectInputFilter.Config.getSerialFilter();
                if (serialFilter != null) {
                    // 首先使用 ObjectInputFilter 执行默认的检查
                    ObjectInputFilter.Status status =
                        serialFilter.checkInput(info);
                    // 如果默认检查的结果不是 Status.UNDECIDED
                    if (status != ObjectInputFilter.Status.UNDECIDED) {
                        // 直接返回检查结果
                        return status;
                    }
                }
                // 如果要恢复的对象不是 1 个
                if(info.references() != 1)
```

```
                        {
                            // 不允许恢复对象
                            return ObjectInputFilter.Status.REJECTED;
                        }
                        if (info.serialClass() != null &&
                            // 如果恢复的不是 Person 类
                            info.serialClass() != Person.class)
                        {
                            // 不允许恢复对象
                            return ObjectInputFilter.Status.REJECTED;
                        }
                        return ObjectInputFilter.Status.UNDECIDED;
                    });
                // 从输入流中读取一个 Java 对象，并将其强制类型转换为 Person 类
                Person p = (Person)ois.readObject();
                System.out.println("名字为: " + p.getName()
                    + "\n 年龄为: " + p.getAge());
            }
            catch (Exception ex)
            {
                ex.printStackTrace();
            }
        }
    }
```

上面程序中的粗体字代码为 ObjectInputStream 设置了 ObjectInputFilter 过滤器（程序使用 Lambda 表达式创建过滤器），程序重写了 checkInput()方法。

重写 checkInput()方法时先使用默认的 ObjectInputFilter 执行检查，如果检查结果不是 Status.UNDECIDED，程序直接返回检查结果。接下来程序通过 FilterInfo 检验序列化数据，如果序列化数据中的对象不唯一（数据已被污染），程序拒绝执行反序列化；如果序列化数据中的对象不是 Person 对象（数据被污染），程序拒绝执行反序列化。通过这种检查，程序可以保证反序列化出来的是唯一的 Person 对象，这样就让反序列化更加安全、健壮。

15.8.5 自定义序列化

在一些特殊的场景下，如果一个类里包含的某些实例变量是敏感信息，例如银行账户信息等，这时不希望系统将该实例变量值进行序列化；或者某个实例变量的类型是不可序列化的，因此不希望对该实例变量进行递归序列化，以避免引发 java.io.NotSerializableException 异常。

提示： 当对某个对象进行序列化时，系统会自动把该对象的所有实例变量依次进行序列化，如果某个实例变量引用到另一个对象，则被引用的对象也会被序列化；如果被引用的对象的实例变量也引用了其他对象，则被引用的对象也会被序列化，这种情况被称为递归序列化。

通过在实例变量前面使用 transient 关键字修饰，可以指定 Java 序列化时无须理会该实例变量。如下 Person 类与前面的 Person 类几乎完全一样，只是它的 age 使用了 transient 关键字修饰。

程序清单：codes\15\15.8\transient\Person.java

```
public class Person
    implements java.io.Serializable
{
    private String name;
    private transient int age;
    // 注意此处没有提供无参数的构造器
    public Person(String name , int age)
    {
        System.out.println("有参数的构造器");
        this.name = name;
        this.age = age;
    }
    // 省略 name 与 age 的 setter 和 getter 方法
    ...
}
```

提示：

transient 关键字只能用于修饰实例变量，不可修饰 Java 程序中的其他成分。

下面程序先序列化一个 Person 对象，然后再反序列化该 Person 对象，得到反序列化的 Person 对象后程序输出该对象的 age 实例变量值。

程序清单：codes\15\15.8\transient\TransientTest.java

```
public class TransientTest
{
    public static void main(String[] args)
    {
        try(
            // 创建一个 ObjectOutputStream 输出流
            ObjectOutputStream oos = new ObjectOutputStream(
                new FileOutputStream("transient.txt"));
            // 创建一个 ObjectInputStream 输入流
            ObjectInputStream ois = new ObjectInputStream(
                new FileInputStream("transient.txt")))
        {
            Person per = new Person("孙悟空", 500);
            // 系统将 per 对象转换成字节序列并输出
            oos.writeObject(per);
            Person p = (Person)ois.readObject();
            System.out.println(p.getAge());
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
        }
    }
}
```

中華工匠

上面程序中的第一行粗体字代码创建了一个 Person 对象，并为它的 name、age 两个实例变量指定了值；第二行粗体字代码将该 Person 对象序列化后输出；第三行粗体字代码从序列化文件中读取该 Person 对象；第四行粗体字代码输出该 Person 对象的 age 实例变量值。由于本程序中的 Person 类的 age 实例变量使用 transient 关键字修饰，所以程序第四行粗体字代码将输出 0。

使用 transient 关键字修饰实例变量虽然简单、方便，但被 transient 修饰的实例变量将被完全隔离在序列化机制之外，这样导致在反序列化恢复 Java 对象时无法取得该实例变量值。Java 还提供了一种自定义序列化机制，通过这种自定义序列化机制可以让程序控制如何序列化各实例变量，甚至完全不序列化某些实例变量（与使用 transient 关键字的效果相同）。

在序列化和反序列化过程中需要特殊处理的类应该提供如下特殊签名的方法，这些特殊的方法用以实现自定义序列化。

- private void writeObject(java.io.ObjectOutputStream out)throws IOException
- private void readObject(java.io.ObjectInputStream in)throws IOException, ClassNotFoundExcception;
- private void readObjectNoData()throws ObjectStreamException;

writeObject()方法负责写入特定类的实例状态，以便相应的 readObject()方法可以恢复它。通过重写该方法，程序员可以完全获得对序列化机制的控制，可以自主决定哪些实例变量需要序列化，需要怎样序列化。在默认情况下，该方法会调用 out.defaultWriteObject 来保存 Java 对象的各实例变量，从而可以实现序列化 Java 对象状态的目的。

readObject()方法负责从流中读取并恢复对象实例变量，通过重写该方法，程序员可以完全获得对反序列化机制的控制，可以自主决定需要反序列化哪些实例变量，以及如何进行反序列化。在默认情况下，该方法会调用 in.defaultReadObject 来恢复 Java 对象的非瞬态实例变量。在通常情况下，readObject()方法与 writeObject()方法对应，如果 writeObject()方法中对 Java 对象的实例变量进行了一些处理，则应该在 readObject()方法中对其实例变量进行相应的反处理，以便正确恢复该对象。

当序列化流不完整时，readObjectNoData()方法可以用来正确地初始化反序列化的对象。例如，接收方使用的反序列化类的版本不同于发送方，或者接收方版本扩展的类不是发送方版本扩展的类，或者

序列化流被篡改时，系统都会调用 readObjectNoData()方法来初始化反序列化的对象。

下面的 Person 类提供了 writeObject()和 readObject()两个方法，其中 writeObject()方法在保存 Person 对象时将其 name 实例变量包装成 StringBuffer，并将其字符序列反转后写入；在 readObject()方法中处理 name 的策略与此对应——先将读取的数据强制类型转换成 StringBuffer，再将其反转后赋给 name 实例变量。

程序清单：codes\15\15.8\custom\Person.java

```
public class Person
    implements java.io.Serializable
{
    private String name;
    private int age;
    // 注意此处没有提供无参数的构造器
    public Person(String name , int age)
    {
        System.out.println("有参数的构造器");
        this.name = name;
        this.age = age;
    }
    // 省略 name 与 age 的 setter 和 getter 方法
    ...
    private void writeObject(java.io.ObjectOutputStream out)
        throws IOException
    {
        // 将 name 实例变量值反转后写入二进制流
        out.writeObject(new StringBuffer(name).reverse());
        out.writeInt(age);
    }
    private void readObject(java.io.ObjectInputStream in)
        throws IOException, ClassNotFoundException
    {
        // 将读取的字符串反转后赋给 name 实例变量
        this.name = ((StringBuffer)in.readObject()).reverse()
            .toString();
        this.age = in.readInt();
    }
}
```

上面程序中用粗体字标出的方法用以实现自定义序列化，对于这个 Person 类而言，序列化、反序列化 Person 实例并没有任何区别——区别在于序列化后的对象流，即使有 Cracker 截获到 Person 对象流，他看到的 name 也是加密后的 name 值，这样就提高了序列化的安全性。

注意：

writeObject()方法存储实例变量的顺序应该和 readObject()方法中恢复实例变量的顺序一致，否则将不能正常恢复该 Java 对象。

对 Person 对象进行序列化和反序列化的程序与前面程序没有任何区别，故此处不再赘述。

还有一种更彻底的自定义机制，它甚至可以在序列化对象时将该对象替换成其他对象。如果需要实现序列化某个对象时替换该对象，则应为序列化类提供如下特殊方法。

```
ANY-ACCESS-MODIFIER Object writeReplace() throws ObjectStreamException;
```

此 writeReplace()方法将由序列化机制调用，只要该方法存在。因为该方法可以拥有私有（private）、受保护的（protected）和包私有（package-private）等访问权限，所以其子类有可能获得该方法。例如，下面的 Person 类提供了 writeReplace()方法，这样可以在写入 Person 对象时将该对象替换成 ArrayList。

程序清单：codes\15\15.8\replace\Person.java

```
public class Person
    implements java.io.Serializable
{
    private String name;
    private int age;
```

```
    // 注意此处没有提供无参数的构造器
    public Person(String name , int age)
    {
        System.out.println("有参数的构造器");
        this.name = name;
        this.age = age;
    }
    // 省略 name 与 age 的 setter 和 getter 方法
    ...
    // 重写 writeReplace 方法，程序在序列化该对象之前，先调用该方法
    private Object writeReplace() throws ObjectStreamException
    {
        ArrayList<Object> list = new ArrayList<Object>();
        list.add(name);
        list.add(age);
        return list;
    }
}
```

Java 的序列化机制保证在序列化某个对象之前，先调用该对象的 writeReplace()方法，如果该方法返回另一个 Java 对象，则系统转为序列化另一个对象。如下程序表面上是序列化 Person 对象，但实际上序列化的是 ArrayList。

程序清单：codes\15\15.8\replace\ReplaceTest.java

```
public class ReplaceTest
{
    public static void main(String[] args)
    {
        try(
            // 创建一个 ObjectOutputStream 输出流
            ObjectOutputStream oos = new ObjectOutputStream(
                new FileOutputStream("replace.txt"));
            // 创建一个 ObjectInputStream 输入流
            ObjectInputStream ois = new ObjectInputStream(
                new FileInputStream("replace.txt")))
        {
            Person per = new Person("孙悟空", 500);
            // 系统将 per 对象转换成字节序列并输出
            oos.writeObject(per);
            // 反序列化读取得到的是 ArrayList
            ArrayList list = (ArrayList)ois.readObject();
            System.out.println(list);
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
        }
    }
}
```

上面程序中第一行粗体字代码使用 writeObject()写入了一个 Person 对象，但第二行粗体字代码使用 readObject()方法返回的实际上是一个 ArrayList 对象，这是因为 Person 类的 writeReplace()方法返回了一个 ArrayList 对象，所以序列化机制在序列化 Person 对象时，实际上是转为序列化 ArrayList 对象。

根据上面的介绍，可以知道系统在序列化某个对象之前，会先调用该对象的 writeReplace()和 writeObject()两个方法，系统总是先调用被序列化对象的 writeReplace()方法，如果该方法返回另一个对象，系统将再次调用另一个对象的 writeReplace()方法……直到该方法不再返回另一个对象为止，程序最后将调用该对象的 writeObject()方法来保存该对象的状态。

与 writeReplace()方法相对的是，序列化机制里还有一个特殊的方法，它可以实现保护性复制整个对象。这个方法就是：

```
ANY-ACCESS-MODIFIER Object readResolve() throws ObjectStreamException;
```

这个方法会紧接着 readObject()之后被调用，该方法的返回值将会代替原来反序列化的对象，而原来 readObject()反序列化的对象将会被立即丢弃。

readResolve()方法在序列化单例类、枚举类时尤其有用。当然，如果使用 Java 5 提供的 enum 来定义枚举类，则完全不用担心，程序没有任何问题。但如果应用中有早期遗留下来的枚举类，例如下面的 Orientation 类就是一个枚举类。

程序清单：codes\15\15.8\resolve\Orientation.java

```
public class Orientation
{
    public static final Orientation HORIZONTAL = new Orientation(1);
    public static final Orientation VERTICAL = new Orientation(2);
    private int value;
    private Orientation(int value)
    {
        this.value = value;
    }
}
```

在 Java 5 以前，这种代码是很常见的。Orientation 类的构造器私有，程序只有两个 Orientation 对象，分别通过 Orientation 的 HORIZONTAL 和 VERTICAL 两个常量来引用。但如果让该类实现 Serializable 接口，则会引发一个问题，如果将一个 Orientation.HORIZONTAL 值序列化后再读出，如下代码片段所示。

```
oos = new ObjectOutputStream(
    new FileOutputStream("transient.txt"));
// 写入 Orientation.HORIZONTAL 值
oos.writeObject(Orientation.HORIZONTAL);
// 创建一个 ObjectInputStream 输入流
ois = new ObjectInputStream(
    new FileInputStream("transient.txt"));
// 读取刚刚序列化的值
Orientation ori = (Orientation)ois.readObject();
```

如果立即拿 ori 和 Orientation.HORIZONTAL 值进行比较，将会发现返回 false。也就是说，ori 是一个新的 Orientation 对象，而不等于 Orientation 类中的任何枚举值——虽然 Orientation 的构造器是 private 的，但反序列化依然可以创建 Orientation 对象。

提示：

前面已经指出，反序列化机制在恢复 Java 对象时无须调用构造器来初始化 Java 对象。从这个意义上来看，序列化机制可以用来“克隆”对象。

在这种情况下，可以通过为 Orientation 类提供一个 readResolve()方法来解决该问题，readResolve()方法的返回值将会代替原来反序列化的对象，也就是让反序列化得到的 Orientation 对象被直接丢弃。下面是为 Orientation 类提供的 readResolve()方法（程序清单同上）。

```
// 为枚举类增加 readResolve()方法
private Object readResolve()throws ObjectStreamException
{
    if (value == 1)
    {
        return HORIZONTAL;
    }
    if (value == 2)
    {
        return VERTICAL;
    }
    return null;
}
```

通过重写 readResolve()方法可以保证反序列化得到的依然是 Orientation 的 HORIZONTAL 或 VERTICAL 两个枚举值之一。

提示：

所有的单例类、枚举类在实现序列化时都应该提供 readResolve()方法，这样才可以保证反序列化的对象依然正常。关于上面示例程序的完整代码，请参考随书光盘中 codes\15\15.8\resolve 路径下的程序。

与 writeReplace()方法类似的是，readResolve()方法也可以使用任意的访问控制符，因此父类的readResolve()方法可能被其子类继承。这样利用 readResolve()方法时就会存在一个明显的缺点，就是当父类已经实现了 readResolve()方法后，子类将变得无从下手。如果父类包含一个 protected 或 public 的readResolve()方法，而且子类也没有重写该方法，将会使得子类反序列化时得到一个父类的对象——这显然不是程序要的结果，而且也不容易发现这种错误。总是让子类重写 readResolve()方法无疑是一个负担，因此对于要被作为父类继承的类而言，实现 readResolve()方法可能有一些潜在的危险。

通常的建议是，对于 final 类重写 readResolve()方法不会有任何问题；否则，重写 readResolve()方法时应尽量使用 private 修饰该方法。

15.8.6 另一种自定义序列化机制

Java 还提供了另一种序列化机制，这种序列化方式完全由程序员决定存储和恢复对象数据。要实现该目标，Java 类必须实现 Externalizable 接口，该接口里定义了如下两个方法。

- void readExternal(ObjectInput in)：需要序列化的类实现 readExternal()方法来实现反序列化。该方法调用 DataInput（它是 ObjectInput 的父接口）的方法来恢复基本类型的实例变量值，调用 ObjectInput 的 readObject()方法来恢复引用类型的实例变量值。
- void writeExternal(ObjectOutput out)：需要序列化的类实现 writeExternal()方法来保存对象的状态。该方法调用 DataOutput（它是 ObjectOutput 的父接口）的方法来保存基本类型的实例变量值，调用 ObjectOutput 的 writeObject()方法来保存引用类型的实例变量值。

实际上，采用实现 Externalizable 接口方式的序列化与前面介绍的自定义序列化非常相似，只是 Externalizable 接口强制自定义序列化。下面的 Person 类实现了 Externalizable 接口，并且实现了该接口里提供的两个方法，用以实现自定义序列化。

程序清单：codes\15\15.8\externalizable\Person.java

```
public class Person
    implements java.io.Externalizable
{
    private String name;
    private int age;
    // 注意必须提供无参数的构造器，否则反序列化时会失败
    public Person(){}
    public Person(String name , int age)
    {
        System.out.println("有参数的构造器");
        this.name = name;
        this.age = age;
    }
    // 省略 name 与 age 的 setter 和 getter 方法
    ...
    public void writeExternal(java.io.ObjectOutput out)
        throws IOException
    {
        // 将 name 实例变量值反转后写入二进制流
        out.writeObject(new StringBuffer(name).reverse());
        out.writeInt(age);
    }
    public void readExternal(java.io.ObjectInput in)
        throws IOException, ClassNotFoundException
    {
        // 将读取的字符串反转后赋给 name 实例变量
        this.name = ((StringBuffer)in.readObject()).reverse().toString();
        this.age = in.readInt();
    }
}
```

上面程序中的 Person 类实现了 java.io.Externalizable 接口（如程序中第一行粗体字代码所示），该 Person 类还实现了 readExternal()、writeExternal()两个方法，这两个方法除方法签名和 readObject()、writeObject()两个方法的方法签名不同之外，其方法体完全一样。

如果程序需要序列化实现 Externalizable 接口的对象，一样调用 ObjectOutputStream 的 writeObject()

方法输出该对象即可；反序列化该对象，则调用 ObjectInputStream 的 readObject()方法，此处不再赘述。

需要指出的是，当使用 Externalizable 机制反序列化对象时，程序会先使用 public 的无参数构造器创建实例，然后才执行 readExternal()方法进行反序列化，因此实现 Externalizable 的序列化类必须提供 public 的无参数构造器。

关于两种序列化机制的对比如表 15.2 所示。

表 15.2 两种序列化机制的对比

实现 Serializable 接口	实现 Externalizable 接口
系统自动存储必要信息	程序员决定存储哪些信息
Java 内建支持，易于实现，只需实现该接口即可，无须任何代码支持	仅仅提供两个空方法，实现该接口必须为两个空方法提供实现
性能略差	性能略好

虽然实现 Externalizable 接口能带来一定的性能提升，但由于实现 Externalizable 接口导致了编程复杂度的增加，所以大部分时候都是采用实现 Serializable 接口方式来实现序列化。

关于对象序列化，还有如下几点需要注意。

- 对象的类名、实例变量（包括基本类型、数组、对其他对象的引用）都会被序列化；方法、类变量（即 static 修饰的成员变量）、transient 实例变量（也被称为瞬态实例变量）都不会被序列化。
- 实现 Serializable 接口的类如果需要让某个实例变量不被序列化，则可在该实例变量前加 transient 修饰符，而不是加 static 关键字。虽然 static 关键字也可达到这个效果，但 static 关键字不能这样用。
- 保证序列化对象的实例变量类型也是可序列化的，否则需要使用 transient 关键字来修饰该实例变量，要不然，该类是不可序列化的。
- 反序列化对象时必须有序列化对象的 class 文件。
- 当通过文件、网络来读取序列化后的对象时，必须按实际写入的顺序读取。

15.8.7 版本

根据前面的介绍可以知道，反序列化 Java 对象时必须提供该对象的 class 文件，现在的问题是，随着项目的升级，系统的 class 文件也会升级，Java 如何保证两个 class 文件的兼容性？

Java 序列化机制允许为序列化类提供一个 private static final 的 serialVersionUID 值，该类变量的值用于标识该 Java 类的序列化版本，也就是说，如果一个类升级后，只要它的 serialVersionUID 类变量值保持不变，序列化机制也会把它们当成同一个序列化版本。

分配 serialVersionUID 类变量的值非常简单，例如下面代码片段：

```
pubic class Test
{
    // 为该类指定一个 serialVersionUID 类变量值
    private static final long serialVersionUID = 512L;
    ...
}
```

为了在反序列化时确保序列化版本的兼容性，最好在每个要序列化的类中加入 private static final long serialVersionUID 这个类变量，具体数值自己定义。这样，即使在某个对象被序列化之后，它所对应的类被修改了，该对象也依然可以被正确地反序列化。

如果不显式定义 serialVersionUID 类变量的值，该类变量的值将由 JVM 根据类的相关信息计算，而修改后的类的计算结果与修改前的类的计算结果往往不同，从而造成对象的反序列化因为类版本不兼容而失败。

可以通过 JDK 安装路径的 bin 目录下的 serialver.exe 工具来获得该类的 serialVersionUID 类变量的值，如下命令所示。

```
serialver Person
```

运行该命令，输出结果如下：

```
Person: static final long serialVersionUID = -2595800114629327570L;
```

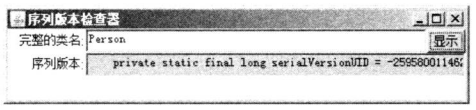


图 15.15 使用 serialver 查看类的 serialVersionUI 类变量

上面的3069227031912694124L就是系统为该Person类生成的 serialVersionUID 类变量的值。如果在运行 serialver 命令时指定-show 选项（不要跟类名参数），即可启动如图 15.15 所示的图形用户界面。

不显式指定 serialVersionUID 类变量的值的另一个坏处是，不利于程序在不同的 JVM 之间移植。因为不同的编译器对该类变量的计算策略可能不同，从而造成虽然类完全没有改变，但是因为 JVM 不同，也会出现序列化版本不兼容而无法正确反序列化的现象。

如果类的修改确实会导致该类反序列化失败，则应该为该类的 serialVersionUID 类变量重新分配值。那么对类的哪些修改可能导致该类实例的反序列化失败呢？下面分三种情况来具体讨论。

- 如果修改类时仅仅修改了方法，则反序列化不受任何影响，类定义无须修改 serialVersionUID 类变量的值。
- 如果修改类时仅仅修改了静态变量或瞬态实例变量，则反序列化不受任何影响，类定义无须修改 serialVersionUID 类变量的值。
- 如果修改类时修改了非瞬态的实例变量，则可能导致序列化版本不兼容。如果对象流中的对象和新类中包含同名的实例变量，而实例变量类型不同，则反序列化失败，类定义应该更新 serialVersionUID 类变量的值。如果对象流中的对象比新类中包含更多的实例变量，则多出的实例变量值被忽略，序列化版本可以兼容，类定义可以不更新 serialVersionUID 类变量的值；如果新类比对象流中的对象包含更多的实例变量，则序列化版本也可以兼容，类定义可以不更新 serialVersionUID 类变量的值；但反序列化得到的新对象中多出的实例变量值都是 null（引用类型实例变量）或 0（基本类型实例变量）。

15.9 NIO

前面介绍 BufferedReader 时提到它的一个特征——当 BufferedReader 读取输入流中的数据时，如果没有读到有效数据，程序将在此处阻塞该线程的执行（使用 InputStream 的 read()方法从流中读取数据时，如果数据源中没有数据，它也会阻塞该线程），也就是前面介绍的输入流、输出流都是阻塞式的输入、输出。不仅如此，传统的输入流、输出流都是通过字节的移动来处理的（即使不直接去处理字节流，但底层的实现还是依赖于字节处理），也就是说，面向流的输入/输出系统一次只能处理一个字节，因此面向流的输入/输出系统通常效率不高。

从 JDK 1.4 开始，Java 提供了一系列改进的输入/输出处理的新功能，这些功能被统称为新 IO（New IO，简称 NIO），新增了许多用于处理输入/输出的类，这些类都被放在 java.nio 包以及子包下，并且对原 java.io 包中的很多类都以 NIO 为基础进行了改写，新增了满足 NIO 的功能。

15.9.1 Java 新 IO 概述

新 IO 和传统的 IO 有相同的目的，都是用于进行输入/输出，但新 IO 使用了不同的方式来处理输入/输出，新 IO 采用内存映射文件的方式来处理输入/输出，新 IO 将文件或文件的一段区域映射到内存中，这样就可以像访问内存一样来访问文件了（这种方式模拟了操作系统上的虚拟内存的概念），通过这种方式来进行输入/输出比传统的输入/输出要快得多。

Java 中与新 IO 相关的包如下。

- java.nio 包：主要包含各种与 Buffer 相关的类。
- java.nio.channels 包：主要包含与 Channel 和 Selector 相关的类。
- java.nio.charset 包：主要包含与字符集相关的类。
- java.nio.channels.spi 包：主要包含与 Channel 相关的服务提供者编程接口。
- java.nio.charset.spi 包：包含与字符集相关的服务提供者编程接口。

Channel（通道）和 Buffer（缓冲）是新 IO 中的两个核心对象，Channel 是对传统的输入/输出系统的模拟，在新 IO 系统中所有的数据都需要通过通道传输；Channel 与传统的 InputStream、OutputStream 最大的区别在于它提供了一个 map()方法，通过该 map()方法可以直接将“一块数据”映射到内存中。

如果说传统的输入/输出系统是面向流的处理，则新 IO 则是面向块的处理。

Buffer 可以被理解成一个容器，它的本质是一个数组，发送到 Channel 中的所有对象都必须首先放到 Buffer 中，而从 Channel 中读取的数据也必须先放到 Buffer 中。此处的 Buffer 有点类似于前面介绍的“竹筒”，但该 Buffer 既可以像“竹筒”那样一次次去 Channel 中取水，也允许使用 Channel 直接将文件的某块数据映射成 Buffer。

除 Channel 和 Buffer 之外，新 IO 还提供了用于将 Unicode 字符串映射成字节序列以及逆映射操作的 Charset 类，也提供了用于支持非阻塞式输入/输出的 Selector 类。

15.9.2 使用 Buffer

从内部结构上来看，Buffer 就像一个数组，它可以保存多个类型相同的数据。Buffer 是一个抽象类，其最常用的子类是 ByteBuffer，它可以在底层字节数组上进行 get/set 操作。除 ByteBuffer 之外，对应于其他基本数据类型(boolean 除外)都有相应的 Buffer 类：CharBuffer、ShortBuffer、IntBuffer、LongBuffer、FloatBuffer、DoubleBuffer。

上面这些 Buffer 类，除 ByteBuffer 之外，它们都采用相同或相似的方法来管理数据，只是各自管理的数据类型不同而已。这些 Buffer 类都没有提供构造器，通过使用如下方法来得到一个 Buffer 对象。

- static XxxBuffer allocate(int capacity)：创建一个容量为 capacity 的 XxxBuffer 对象。

但实际使用较多的是 ByteBuffer 和 CharBuffer，其他 Buffer 子类则较少用到。其中 ByteBuffer 类还有一个子类：MappedByteBuffer，它用于表示 Channel 将磁盘文件的部分或全部内容映射到内存中后得到的结果，通常 MappedByteBuffer 对象由 Channel 的 map()方法返回。

在 Buffer 中有三个重要的概念：容量（capacity）、界限（limit）和位置（position）。

- 容量（capacity）：缓冲区的容量（capacity）表示该 Buffer 的最大数据容量，即最多可以存储多少数据。缓冲区的容量不可能为负值，创建后不能改变。
- 界限（limit）：第一个不应该被读出或者写入的缓冲区位置索引。也就是说，位于 limit 后的数据既不可被读，也不可被写。
- 位置（position）：用于指明下一个可以被读出的或者写入的缓冲区位置索引（类似于 IO 流中的记录指针）。当使用 Buffer 从 Channel 中读取数据时，position 的值恰好等于已经读到了多少数据。当刚刚新建一个 Buffer 对象时，其 position 为 0；如果从 Channel 中读取了 2 个数据到该 Buffer 中，则 position 为 2，指向 Buffer 中第 3 个（第 1 个位置的索引为 0）位置。

除此之外，Buffer 里还支持一个可选的标记（mark，类似于传统 IO 流中的 mark），Buffer 允许直接将 position 定位到该 mark 处。这些值满足如下关系：

```
0≤mark≤position≤limit≤capacity
```

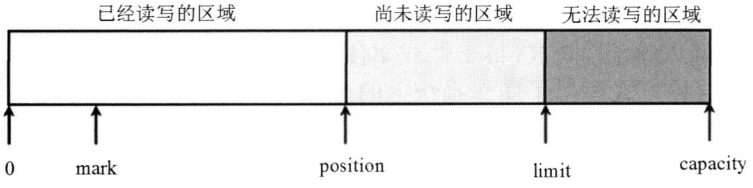


图 15.16 Buffer 读入数据后的示意图

图 15.16 显示了某个 Buffer 读入了一些数据后的示意图。

Buffer 的主要作用就是装入数据，然后输出数据（其作用类似于前面介绍的取水的“竹筒”），开始时 Buffer 的 position 为 0，limit 为 capacity，程序可通过 put()方法向 Buffer 中放入一些数据（或者从 Channel 中获取一些数据），每放入一些数据，Buffer 的 position 相应地向后移动一些位置。

当 Buffer 装入数据结束后，调用 Buffer 的 flip()方法，该方法将 limit 设置为 position 所在位置，并将 position 设为 0，这就使得 Buffer 的读写指针又移到了开始位置。也就是说，Buffer 调用 flip()方法之后，Buffer 为输出数据做好准备；当 Buffer 输出数据结束后，Buffer 调用 clear()方法，clear()方法不是清空 Buffer 的数据，它仅仅将 position 置为 0，将 limit 置为 capacity，这样为再次向 Buffer 中装入数据做好准备。

提示：
Buffer 中包含两个重要的方法，即 flip()和 clear()，flip()为从 Buffer 中取出数据做好准备，而 clear()为再次向 Buffer 中装入数据做好准备。

除此之外，Buffer 还包含如下一些常用的方法。

- int capacity()：返回 Buffer 的 capacity 大小。
- boolean hasRemaining()：判断当前位置（position）和界限（limit）之间是否还有元素可供处理。
- int limit()：返回 Buffer 的界限（limit）的位置。
- Buffer limit(int newLt)：重新设置界限（limit）的值，并返回一个具有新的 limit 的缓冲区对象。
- Buffer mark()：设置 Buffer 的 mark 位置，它只能在 0 和位置（position）之间做 mark。
- int position()：返回 Buffer 中的 position 值。
- Buffer position(int newPs)：设置 Buffer 的 position，并返回 position 被修改后的 Buffer 对象。
- int remaining()：返回当前位置和界限（limit）之间的元素个数。
- Buffer reset()：将位置（position）转到 mark 所在的位置。
- Buffer rewind()：将位置（position）设置成 0，取消设置的 mark。

除这些移动 position、limit、mark 的方法之外，Buffer 的所有子类还提供了两个重要的方法：put()和 get()方法，用于向 Buffer 中放入数据和从 Buffer 中取出数据。当使用 put()和 get()方法放入、取出数据时，Buffer 既支持对单个数据的访问，也支持对批量数据的访问（以数组作为参数）。

当使用 put()和 get()来访问 Buffer 中的数据时，分为相对和绝对两种。

- 相对（Relative）：从 Buffer 的当前 position 处开始读取或写入数据，然后将位置（position）的值按处理元素的个数增加。
- 绝对（Absolute）：直接根据索引向 Buffer 中读取或写入数据，使用绝对方式访问 Buffer 里的数据时，并不会影响位置（position）的值。

下面程序示范了 Buffer 的一些常规操作。

程序清单：codes\15\15.9\BufferTest.java

```
public class BufferTest
{
    public static void main(String[] args)
    {
        // 创建 Buffer
        CharBuffer buff = CharBuffer.allocate(8);    // ①
        System.out.println("capacity: "    + buff.capacity());
        System.out.println("limit: " + buff.limit());
        System.out.println("position: " + buff.position());
        // 放入元素
        buff.put('a');
        buff.put('b');
        buff.put('c');      // ②
        System.out.println("加入三个元素后，position = "
            + buff.position());
        // 调用 flip()方法
        buff.flip();      // ③
        System.out.println("执行 flip()后，limit = " + buff.limit());
        System.out.println("position = " + buff.position());
        // 取出第一个元素
        System.out.println("第一个元素(position=0): " + buff.get());   // ④
        System.out.println("取出一个元素后，position = "
            + buff.position());
        // 调用 clear()方法
        buff.clear();     // ⑤
        System.out.println("执行 clear()后，limit = " + buff.limit());
        System.out.println("执行 clear()后，position = "
            + buff.position());
        System.out.println("执行 clear()后，缓冲区内容并没有被清除："
            + "第三个元素为：" +  buff.get(2));    // ⑥
```

```
        System.out.println("执行绝对读取后，position = "
            + buff.position());
    }
}
```

在上面程序的①号代码处，通过 CharBuffer 的一个静态方法 allocate()创建了一个 capacity 为 8 的 CharBuffer，此时该 Buffer 的 limit 和 capacity 为 8，position 为 0，如图 15.17 所示。

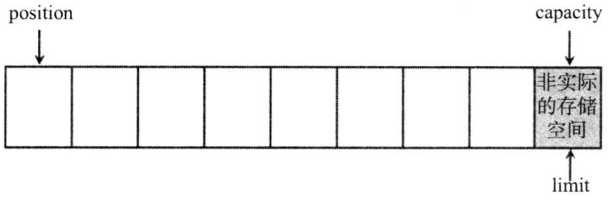


图 15.17 新分配的 CharBuffer 对象

接下来程序执行到②号代码处，程序向 CharBuffer 中放入 3 个数值，放入 3 个数值后的 CharBuffer 效果如图 15.18 所示。

程序执行到③号代码处，调用了 Buffer 的 flip()方法，该方法将把 limit 设为 position 处，把 position 设为 0，如图 15.19 所示。

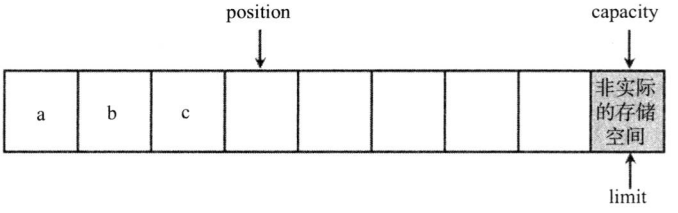


图 15.18 向 Buffer 中放入 3 个对象后的示意图

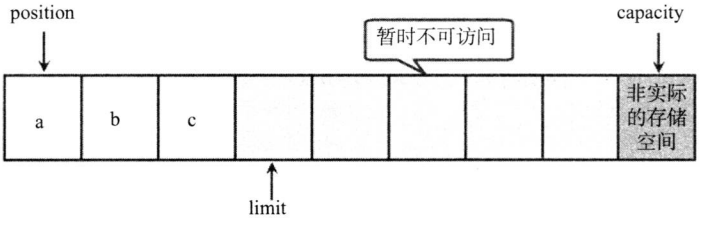


图 15.19 执行 Buffer 的 flip()方法后的示意图

从图 15.19 中可以看出，当 Buffer 调用了 flip()方法之后，limit 就移到了原来 position 所在位置，这样相当于把 Buffer 中没有数据的存储空间“封印”起来，从而避免读取 Buffer 数据时读到 null 值。

接下来程序在④号代码处取出一个元素，取出一个元素后 position 向后移动一位，也就是该 Buffer 的 position 等于 1。程序执行到⑤号代码处，Buffer 调用 clear()方法将 position 设为 0，将 limit 设为与 capacity 相等。执行 clear()方法后的 Buffer 示意图如图 15.20 所示。

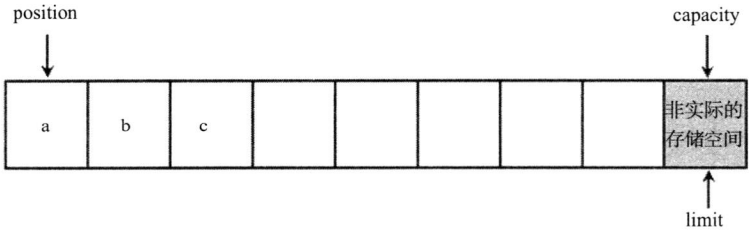


图 15.20 执行 clear()后的 Buffer 示意图

从图 15.20 中可以看出，对 Buffer 执行 clear()方法后，该 Buffer 对象里的数据依然存在，所以程序在⑥号代码处依然可以取出位置为 2 的值，也就是字符 c。因为⑥号代码采用的是根据索引来取值的方式，所以该方法不会影响 Buffer 的 position。

通过 allocate()方法创建的 Buffer 对象是普通 Buffer，ByteBuffer 还提供了一个 allocateDirect()方法来

创建直接 Buffer。直接 Buffer 的创建成本比普通 Buffer 的创建成本高，但直接 Buffer 的读取效率更高。

提示：

由于直接 Buffer 的创建成本很高，所以直接 Buffer 只适用于长生存期的 Buffer，而不适用于短生存期、一次用完就丢弃的 Buffer。而且只有 ByteBuffer 才提供了 allocateDirect()方法，所以只能在 ByteBuffer 级别上创建直接 Buffer。如果希望使用其他类型，则应该将该 Buffer 转换成其他类型的 Buffer。

直接 Buffer 在编程上的用法与普通 Buffer 并没有太大的区别，故此处不再赘述。

15.9.3 使用 Channel

Channel 类似于传统的流对象，但与传统的流对象有两个主要区别。

- Channel 可以直接将指定文件的部分或全部直接映射成 Buffer。
- 程序不能直接访问 Channel 中的数据，包括读取、写入都不行，Channel 只能与 Buffer 进行交互。也就是说，如果要从 Channel 中取得数据，必须先用 Buffer 从 Channel 中取出一些数据，然后让程序从 Buffer 中取出这些数据；如果要将程序中的数据写入 Channel，一样先让程序将数据放入 Buffer 中，程序再将 Buffer 里的数据写入 Channel 中。

Java 为 Channel 接口提供了 DatagramChannel、FileChannel、Pipe.SinkChannel、Pipe.SourceChannel、SelectableChannel、ServerSocketChannel、SocketChannel 等实现类，本节主要介绍 FileChannel 的用法。根据这些 Channel 的名字不难发现，新 IO 里的 Channel 是按功能来划分的，例如 Pipe.SinkChannel、Pipe.SourceChannel 是用于支持线程之间通信的管道 Channel；ServerSocketChannel、SocketChannel 是用于支持 TCP 网络通信的 Channel；而 DatagramChannel 则是用于支持 UDP 网络通信的 Channel。

提示：

本书将会在第 17 章介绍网络通信编程的详细内容，如果需要掌握 ServerSocketChannel、SocketChannel 等 Channel 的用法，可以参考本书第 17 章。

所有的 Channel 都不应该通过构造器来直接创建，而是通过传统的节点 InputStream、OutputStream 的 getChannel()方法来返回对应的 Channel，不同的节点流获得的 Channel 不一样。例如，FileInputStream、FileOutputStream 的 getChannel()返回的是 FileChannel，而 PipedInputStream 和 PipedOutputStream 的 getChannel()返回的是 Pipe.SinkChannel、Pipe.SourceChannel。

Channel 中最常用的三类方法是 map()、read()和 write()，其中 map()方法用于将 Channel 对应的部分或全部数据映射成 ByteBuffer；而 read()或 write()方法都有一系列重载形式，这些方法用于从 Buffer 中读取数据或向 Buffer 中写入数据。

map()方法的方法签名为：MappedByteBuffer map(FileChannel.MapMode mode, long position, long size)，第一个参数执行映射时的模式，分别有只读、读写等模式；而第二个、第三个参数用于控制将 Channel 的哪些数据映射成 ByteBuffer。

下面程序示范了直接将 FileChannel 的全部数据映射成 ByteBuffer 的效果。

程序清单：codes\15\15.9\FileChannelTest.java

```
public class FileChannelTest
{
    public static void main(String[] args)
    {
        File f = new File("FileChannelTest.java");
        try(
            // 创建 FileInputStream，以该文件输入流创建 FileChannel
            FileChannel inChannel = new FileInputStream(f).getChannel();
            // 以文件输出流创建 FileChannel，用以控制输出
            FileChannel outChannel = new FileOutputStream("a.txt")
                .getChannel())
        {
            // 将 FileChannel 里的全部数据映射成 ByteBuffer
            MappedByteBuffer buffer = inChannel.map(FileChannel
```

```
                .MapMode.READ_ONLY , 0 , f.length());  // ①
            // 使用GBK的字符集来创建解码器
            Charset charset = Charset.forName("GBK");
            // 直接将buffer里的数据全部输出
            outChannel.write(buffer);     // ②
            // 再次调用buffer的clear()方法，复原limit、position的位置
            buffer.clear();
            // 创建解码器（CharsetDecoder）对象
            CharsetDecoder decoder = charset.newDecoder();
            // 使用解码器将ByteBuffer转换成CharBuffer
            CharBuffer charBuffer =  decoder.decode(buffer);
            // CharBuffer的toString方法可以获取对应的字符串
            System.out.println(charBuffer);
        }
        catch (IOException ex)
        {
            ex.printStackTrace();
        }
    }
}
```

上面程序中的两行粗体字代码分别使用 FileInputStream、FileOutputStream 来获取 FileChannel，虽然 FileChannel 既可以读取也可以写入，但 FileInputStream 获取的 FileChannel 只能读，而 FileOutputStream 获取的 FileChannel 只能写。程序中①号代码处直接将指定 Channel 中的全部数据映射成 ByteBuffer，然后程序中②号代码处直接将整个 ByteBuffer 的全部数据写入一个输出 FileChannel 中，这就完成了文件的复制。

程序后面部分为了能将 FileChannelTest.java 文件里的内容打印出来，使用了 Charset 类和 CharsetDecoder 类将 ByteBuffer 转换成 CharBuffer。关于 Charset 和 CharsetDecoder 下一节将会有更详细的介绍。

不仅 InputStream、OutputStream 包含了 getChannel()方法，在 RandomAccessFile 中也包含了一个 getChannel()方法，RandomAccessFile 返回的 FileChannel()是只读的还是读写的，则取决于 RandomAccessFile 打开文件的模式。例如，下面程序将会对 a.txt 文件的内容进行复制，追加在该文件后面。

程序清单：codes\15\15.9\RandomFileChannelTest.java

```
public class RandomFileChannelTest
{
    public static void main(String[] args)
        throws IOException
    {
        File f = new File("a.txt");
        try(
            // 创建一个RandomAccessFile对象
            RandomAccessFile raf = new RandomAccessFile(f, "rw");
            // 获取RandomAccessFile对应的Channel
            FileChannel randomChannel = raf.getChannel())
        {
            // 将Channel中的所有数据映射成ByteBuffer
            ByteBuffer buffer = randomChannel.map(FileChannel
                .MapMode.READ_ONLY, 0 , f.length());
            // 把Channel的记录指针移动到最后
            randomChannel.position(f.length());
            // 将buffer中的所有数据输出
            randomChannel.write(buffer);
        }
    }
}
```

上面程序中的粗体字代码可以将 Channel 的记录指针移动到该 Channel 的最后，从而可以让程序将指定 ByteBuffer 的数据追加到该 Channel 的后面。每次运行上面程序，都会把 a.txt 文件的内容复制一份，并将全部内容追加到该文件的后面。

如果读者习惯了传统 IO 的“用竹筒多次重复取水”的过程，或者担心 Channel 对应的文件过大，

使用 map()方法一次将所有的文件内容映射到内存中引起性能下降，也可以使用 Channel 和 Buffer 传统的“用竹筒多次重复取水”的方式。如下程序所示。

程序清单：codes\15\15.9\ReadFile.java

```
public class ReadFile
{
    public static void main(String[] args)
        throws IOException
    {
        try(
            // 创建文件输入流
            FileInputStream fis = new FileInputStream("ReadFile.java");
            // 创建一个 FileChannel
            FileChannel fcin = fis.getChannel())
        {
            // 定义一个 ByteBuffer 对象，用于重复取水
            ByteBuffer bbuff = ByteBuffer.allocate(256);
            // 将 FileChannel 中的数据放入 ByteBuffer 中
            while(fcin.read(bbuff) != -1 )
            {
                // 锁定 Buffer 的空白区
                bbuff.flip();
                // 创建 Charset 对象
                Charset charset = Charset.forName("GBK");
                // 创建解码器（CharsetDecoder）对象
                CharsetDecoder decoder = charset.newDecoder();
                // 将 ByteBuffer 的内容转码
                CharBuffer cbuff = decoder.decode(bbuff);
                System.out.print(cbuff);
                // 将 Buffer 初始化，为下一次读取数据做准备
                bbuff.clear();
            }
        }
    }
}
```

上面代码虽然使用 FileChannel 和 Buffer 来读取文件，但处理方式和使用 InputStream、byte[]来读取文件的方式几乎一样，都是采用“用竹筒多次重复取水”的方式。但因为 Buffer 提供了 flip()和 clear()两个方法，所以程序处理起来比较方便，每次读取数据后调用 flip()方法将没有数据的区域“封印”起来，避免程序从 Buffer 中取出 null 值；数据取出后立即调用 clear()方法将 Buffer 的 position 设 0，为下一次读取数据做准备。

15.9.4 字符集和 Charset

前面已经提到：计算机里的文件、数据、图片文件只是一种表面现象，所有文件在底层都是二进制文件，即全部都是字节码。图片、音乐文件暂时先不说，对于文本文件而言，之所以可以看到一个个的字符，这完全是因为系统将底层的二进制序列转换成字符的缘故。在这个过程中涉及两个概念：编码（Encode）和解码（Decode），通常而言，把明文的字符序列转换成计算机理解的二进制序列（普通人看不懂）称为编码，把二进制序列转换成普通人能看懂的明文字符串称为解码，如图 15.21 所示。

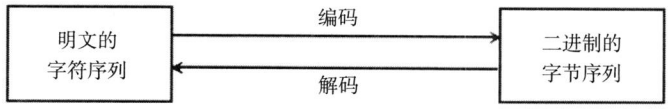


图 15.21 编码/解码示意图

提示：

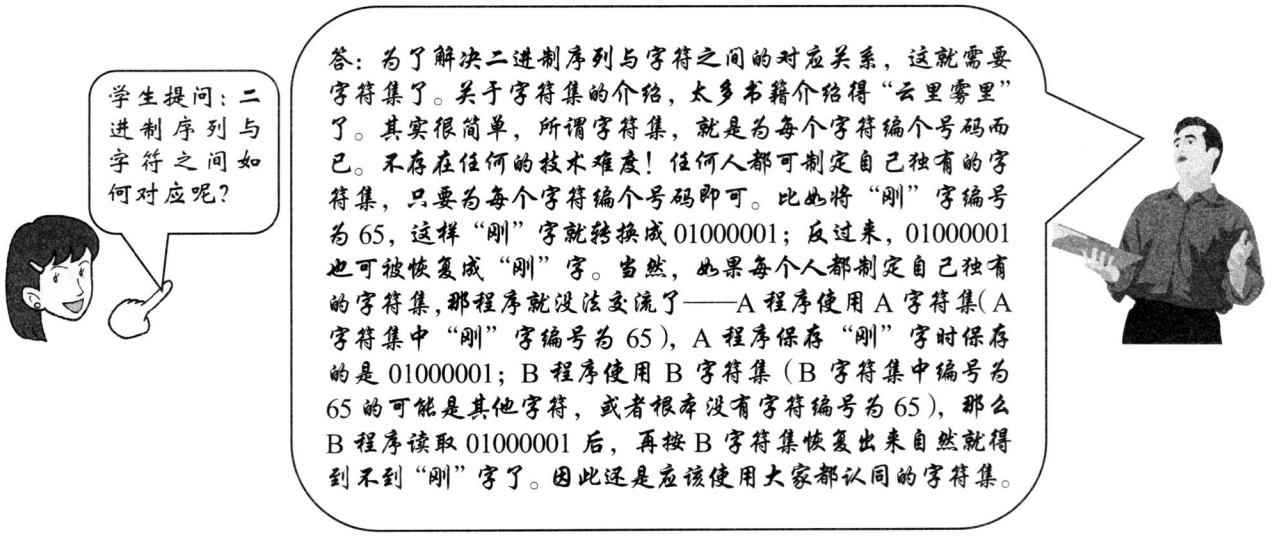
Encode 和 Decode 两个专业术语来自于早期的电报、情报等，当把明文的消息转换成普通人看不懂的电码（或密码）的过程就是 Encode，而将电码（或密码）翻译成明文的消息则被称为 Decode。后来计算机也采用了这两个概念，其作用已经发生了变化。

计算机底层是没有文本文件、图片文件之分的，它只是忠实地记录每个文件的二进制序列而已。当需要保存文本文件时，程序必须先把文件中的每个字符翻译成二进制序列；当需要读取文本文件时，程序必须把二进制序列转换为一个个的字符。

Java 默认使用 Unicode 字符集，但很多操作系统并不使用 Unicode 字符集，那么当从系统中读取数据到 Java 程序中时，就可能出现乱码等问题

JDK 1.4 提供了 Charset 来处理字节序列和字符序列（字符串）之间的转换关系，该类包含了用于创建解码器和编码器的方法，还提供了获取 Charset 所支持字符集的方法，Charset 类是不可变的。

Charset 类提供了一个 availableCharsets()静态方法来获取当前 JDK 所支持的所有字符集。所以程序可以使用如下程序来获取该 JDK 所支持的全部字符集。



程序清单：codes\15\15.9\CharsetTest.java

```
public class CharsetTest
{
    public static void main(String[] args)
    {
        // 获取 Java 支持的全部字符集
        SortedMap<String,Charset>  map = Charset.availableCharsets();
        for (String alias : map.keySet())
        {
            // 输出字符集的别名和对应的 Charset 对象
            System.out.println(alias + "----->"
                + map.get(alias));
        }
    }
}
```

上面程序中的粗体字代码获取了当前 Java 所支持的全部字符集，并使用遍历方式打印了所有字符集的别名（字符集的字符串名称）和 Charset 对象。从上面程序可以看出，每个字符集都有一个字符串名称，也被称为字符串别名。对于中国的程序员而言，下面几个字符串别名是常用的。

- GBK：简体中文字符集。
- BIG5：繁体中文字符集。
- ISO-8859-1：ISO 拉丁字母表 No.1，也叫做 ISO-LATIN-1。
- UTF-8：8 位 UCS 转换格式。
- UTF-16BE：16 位 UCS 转换格式，Big-endian（最低地址存放高位字节）字节顺序。
- UTF-16LE：16 位 UCS 转换格式，Little-endian（最高地址存放低位字节）字节顺序。
- UTF-16：16 位 UCS 转换格式，字节顺序由可选的字节顺序标记来标识。

提示：

可以使用 System 类的 getProperties()方法来访问本地系统的文件编码格式，文件编码格式的属性名为 file.encoding。例如，9.2 节介绍 System 类时得到的 a.txt 文件中包含 file.encoding=GBK 行，这表明编写本书用的操作系统使用 GBK 编码方式。

一旦知道了字符集的别名之后，程序就可以调用 Charset 的 forName()方法来创建对应的 Charset 对象，forName()方法的参数就是相应字符集的别名。例如如下代码：

```
Charset cs = Charset.forName("ISO-8859-1");
Charset csCn = Charset.forName("GBK");
```

获得了 Charset 对象之后，就可以通过该对象的 newDecoder()、newEncoder()这两个方法分别返回 CharsetDecoder 和 CharsetEncoder 对象，代表该 Charset 的解码器和编码器。调用 CharsetDecoder 的 decode()方法就可以将 ByteBuffer（字节序列）转换成 CharBuffer（字符序列），调用 CharsetEncoder 的 encode()方法就可以将 CharBuffer 或 String（字符序列）转换成 ByteBuffer（字节序列）。如下程序使用了 CharsetEncoder 和 CharsetDecoder 完成了 ByteBuffer 和 CharBuffer 之间的转换。

注意：

Java 7 新增了一个 StandardCharsets 类，该类里包含了 ISO_8859_1、UTF_8、UTF_16 等类变量，这些类变量代表了最常用的字符集对应的 Charset 对象。

程序清单：codes\15\15.9\CharsetTransform.java

```
public class CharsetTransform
{
    public static void main(String[] args)
        throws Exception
    {
        // 创建简体中文对应的 Charset
        Charset cn = Charset.forName("GBK");
        // 获取 cn 对象对应的编码器和解码器
        CharsetEncoder cnEncoder = cn.newEncoder();
        CharsetDecoder cnDecoder = cn.newDecoder();
        // 创建一个 CharBuffer 对象
        CharBuffer cbuff = CharBuffer.allocate(8);
        cbuff.put('孙');
        cbuff.put('悟');
        cbuff.put('空');
        cbuff.flip();
        // 将 CharBuffer 中的字符序列转换成字节序列
        ByteBuffer bbuff = cnEncoder.encode(cbuff);
        // 循环访问 ByteBuffer 中的每个字节
        for (int i = 0; i < bbuff.capacity() ; i++)
        {
            System.out.print(bbuff.get(i) + " ");
        }
        // 将 ByteBuffer 的数据解码成字符序列
        System.out.println("\n" + cnDecoder.decode(bbuff));
    }
}
```

上面程序中的两行粗体字代码分别实现了将 CharBuffer 转换成 ByteBuffer，将 ByteBuffer 转换成 CharBuffer 的功能。实际上，Charset 类也提供了如下三个方法。

- CharBuffer decode(ByteBuffer bb)：将 ByteBuffer 中的字节序列转换成字符序列的便捷方法。
- ByteBuffer encode(CharBuffer cb)：将 CharBuffer 中的字符序列转换成字节序列的便捷方法。
- ByteBuffer encode(String str)：将 String 中的字符序列转换成字节序列的便捷方法。

也就是说，获取了 Charset 对象后，如果仅仅需要进行简单的编码、解码操作，其实无须创建 CharsetEncoder 和 CharsetDecoder 对象，直接调用 Charset 的 encode()和 decode()方法进行编码、解码即可。

提示：

在 String 类里也提供了一个 getBytes(String charset)方法，该方法返回 byte[]，该方法也是使用指定的字符集将字符串转换成字节序列。

15.9.5 文件锁

文件锁在操作系统中是很平常的事情，如果多个运行的程序需要并发修改同一个文件时，程序之间需要某种机制来进行通信，使用文件锁可以有效地阻止多个进程并发修改同一个文件，所以现在的大部分操作系统都提供了文件锁的功能。

文件锁控制文件的全部或部分字节的访问，但文件锁在不同的操作系统中差别较大，所以早期的 JDK 版本并未提供文件锁的支持。从 JDK 1.4 的 NIO 开始，Java 开始提供文件锁的支持。

在 NIO 中，Java 提供了 FileLock 来支持文件锁定功能，在 FileChannel 中提供的 lock()/tryLock()方法可以获得文件锁 FileLock 对象，从而锁定文件。lock()和 tryLock()方法存在区别：当 lock()试图锁定某个文件时，如果无法得到文件锁，程序将一直阻塞；而 tryLock()是尝试锁定文件，它将直接返回而不是阻塞，如果获得了文件锁，该方法则返回该文件锁，否则将返回 null。

如果 FileChannel 只想锁定文件的部分内容，而不是锁定全部内容，则可以使用如下的 lock()或 tryLock()方法。

- lock(long position, long size, boolean shared)：对文件从 position 开始，长度为 size 的内容加锁，该方法是阻塞式的。
- tryLock(long position, long size, boolean shared)：非阻塞式的加锁方法。参数的作用与上一个方法类似。

当参数 shared 为 true 时，表明该锁是一个共享锁，它将允许多个进程来读取该文件，但阻止其他进程获得对该文件的排他锁。当 shared 为 false 时，表明该锁是一个排他锁，它将锁住对该文件的读写。程序可以通过调用 FileLock 的 isShared 来判断它获得的锁是否为共享锁。

注意：

直接使用 lock()或 tryLock()方法获取的文件锁是排他锁。

处理完文件后通过 FileLock 的 release()方法释放文件锁。下面程序示范了使用 FileLock 锁定文件的示例。

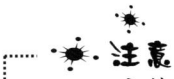
程序清单：codes\15\15.9\FileLockTest.java

```
public class FileLockTest
{
    public static void main(String[] args)
        throws Exception
    {

        try(
            // 使用 FileOutputStream 获取 FileChannel
            FileChannel channel = new FileOutputStream("a.txt")
                .getChannel())
        {
            // 使用非阻塞式方式对指定文件加锁
            FileLock lock = channel.tryLock();
            // 程序暂停 10s
            Thread.sleep(10000);
            // 释放锁
            lock.release();
        }
    }
}
```

上面程序中的第一行粗体字代码用于对指定文件加锁，接着程序调用 Thread.sleep(10000)暂停了 10 秒后才释放文件锁（如程序中第二行粗体字代码所示），因此在这 10 秒之内，其他程序无法对 a.txt 文

件进行修改。



注意：

文件锁虽然可以用于控制并发访问，但对于高并发访问的情形，还是推荐使用数据库来保存程序信息，而不是使用文件。

关于文件锁还需要指出如下几点。

- 在某些平台上，文件锁仅仅是建议性的，并不是强制性的。这意味着即使一个程序不能获得文件锁，它也可以对该文件进行读写。
- 在某些平台上，不能同步地锁定一个文件并把它映射到内存中。
- 文件锁是由 Java 虚拟机所持有的，如果两个 Java 程序使用同一个 Java 虚拟机运行，则它们不能对同一个文件进行加锁。
- 在某些平台上关闭 FileChannel 时，会释放 Java 虚拟机在该文件上的所有锁，因此应该避免对同一个被锁定的文件打开多个 FileChannel。

15.10 Java 7 的 NIO.2

Java 7 对原有的 NIO 进行了重大改进，改进主要包括如下两方面的内容。

- 提供了全面的文件 IO 和文件系统访问支持。
- 基于异步 Channel 的 IO。

第一个改进表现为 Java 7 新增的 java.nio.file 包及各个子包；第二个改进表现为 Java 7 在 java.nio.channels 包下增加了多个以 Asynchronous 开头的 Channel 接口和类。Java 7 把这种改进称为 NIO.2，本章先详细介绍 NIO 的第二个改进。

15.10.1 Path、Paths 和 Files 核心 API

早期的 Java 只提供了一个 File 类来访问文件系统，但 File 类的功能比较有限，它不能利用特定文件系统的特性，File 所提供的方法的性能也不高。而且，其大多数方法在出错时仅返回失败，并不会提供异常信息。

NIO.2 为了弥补这种不足，引入了一个 Path 接口，Path 接口代表一个平台无关的平台路径。除此之外，NIO.2 还提供了 Files、Paths 两个工具类，其中 Files 包含了大量静态的工具方法来操作文件；Paths 则包含了两个返回 Path 的静态工厂方法。

提示：

Files 和 Paths 两个工具类非常符合 Java 一贯的命名风格，比如前面介绍的操作数组的工具类为 Arrays，操作集合的工具类为 Collections，这种一致的命名风格可以让读者快速了解这些工具类的用途。

下面程序简单示范了 Path 接口的功能和用法。

程序清单：codes\15\15.10\PathTest.java

```
public class PathTest
{
    public static void main(String[] args)
        throws Exception
    {
        // 以当前路径来创建 Path 对象
        Path path = Paths.get(".");
        System.out.println("path 里包含的路径数量："
            + path.getNameCount());
        System.out.println("path 的根路径：" + path.getRoot());
        // 获取 path 对应的绝对路径
        Path absolutePath = path.toAbsolutePath();
        System.out.println(absolutePath);
```

```
        // 获取绝对路径的根路径
        System.out.println("absolutePath 的根路径: "
            + absolutePath.getRoot());
        // 获取绝对路径所包含的路径数量
        System.out.println("absolutePath 里包含的路径数量: "
            + absolutePath.getNameCount());
        System.out.println(absolutePath.getName(3));
        // 以多个 String 来构建 Path 对象
        Path path2 = Paths.get("g:" , "publish" , "codes");
        System.out.println(path2);
    }
}
```

从上面程序可以看出，Paths 提供了 get(String first, String... more)方法来获取 Path 对象，Paths 会将给定的多个字符串连缀成路径，比如 Paths.get("g:" , "publish" , "codes")就返回 g:\publish\codes 路径。

上面程序中的粗体字代码示范了 Path 接口的常用方法，读者可能对 getNameCount()方法感到有点困惑，此处简要说明一下：它会返回 Path 路径所包含的路径名的数量，例如 g:\publish\codes 调用该方法就会返回 3。

Files 是一个操作文件的工具类，它提供了大量便捷的工具方法，下面程序简单示范了 Files 类的用法。

程序清单：codes\15\15.10\FilesTest.java

```
public class FilesTest
{
    public static void main(String[] args)
        throws Exception
    {
        // 复制文件
        Files.copy(Paths.get("FilesTest.java")
            , new FileOutputStream("a.txt"));
        // 判断 FilesTest.java 文件是否为隐藏文件
        System.out.println("FilesTest.java 是否为隐藏文件: "
            + Files.isHidden(Paths.get("FilesTest.java")));
        // 一次性读取 FilesTest.java 文件的所有行
        List<String> lines = Files.readAllLines(Paths
            .get("FilesTest.java"), Charset.forName("gbk"));
        System.out.println(lines);
        // 判断指定文件的大小
        System.out.println("FilesTest.java 的大小为: "
            + Files.size(Paths.get("FilesTest.java")));
        List<String> poem = new ArrayList<>();
        poem.add("水晶潭底银鱼跃");
        poem.add("清徐风中碧竿横");
        // 直接将多个字符串内容写入指定文件中
        Files.write(Paths.get("pome.txt") , poem
            , Charset.forName("gbk"));
        // 使用 Java 8 新增的 Stream API 列出当前目录下所有文件和子目录
        Files.list(Paths.get(".")).forEach(path->System.out.println(path));  // ①
        // 使用 Java 8 新增的 Stream API 读取文件内容
        Files.lines(Paths.get("FilesTest.java") , Charset.forName("gbk"))
            .forEach(line -> System.out.println(line));     // ②
        FileStore cStore = Files.getFileStore(Paths.get("C:"));
        // 判断 C 盘的总空间、可用空间
        System.out.println("C:共有空间: " + cStore.getTotalSpace());
        System.out.println("C:可用空间: " + cStore.getUsableSpace());
    }
}
```

上面程序中的粗体字代码简单示范了 Files 工具类的用法。从上面程序不难看出，Files 类是一个高度封装的工具类，它提供了大量的工具方法来完成文件复制、读取文件内容、写入文件内容等功能——这些原本需要程序员通过 IO 操作才能完成的功能，现在 Files 类只要一个工具方法即可。

Java 8 进一步增强了 Files 工具类的功能，允许开发者使用 Stream API 来操作文件目录和文件内容，上面示例程序中①号代码使用 Stream API 列出了指定路径下的所有文件和目录；②号代码则使用了

Stream API 读取文件内容。

提示：

读者应该熟练掌握 Files 工具类的用法，它所包含的工具方法可以大大地简化文件 IO。

15.10.2 使用 FileVisitor 遍历文件和目录

在以前的 Java 版本中，如果程序要遍历指定目录下的所有文件和子目录，则只能使用递归进行遍历，但这种方式不仅复杂，而且灵活性也不高。

有了 Files 工具类的帮助，现在可以用更优雅的方式来遍历文件和子目录。Files 类提供了如下两个方法来遍历文件和子目录。

- walkFileTree(Path start, FileVisitor<? super Path> visitor)：遍历 start 路径下的所有文件和子目录。
- walkFileTree(Path start, Set<FileVisitOption> options, int maxDepth, FileVisitor<? super Path> visitor)：与上一个方法的功能类似。该方法最多遍历 maxDepth 深度的文件。

上面两个方法都需要 FileVisitor 参数，FileVisitor 代表一个文件访问器，walkFileTree()方法会自动遍历 start 路径下的所有文件和子目录，遍历文件和子目录都会“触发”FileVisitor 中相应的方法。FileVisitor 中定义了如下 4 个方法。

- FileVisitResult postVisitDirectory(T dir, IOException exc)：访问子目录之后触发该方法。
- FileVisitResult preVisitDirectory(T dir, BasicFileAttributes attrs)：访问子目录之前触发该方法。
- FileVisitResult visitFile(T file, BasicFileAttributes attrs)：访问 file 文件时触发该方法。
- FileVisitResult visitFileFailed(T file, IOException exc)：访问 file 文件失败时触发该方法。

上面 4 个方法都返回一个 FileVisitResult 对象，它是一个枚举类，代表了访问之后的后续行为。FileVisitResult 定义了如下几种后续行为。

- CONTINUE：代表“继续访问”的后续行为。
- SKIP_SIBLINGS：代表“继续访问”的后续行为，但不访问该文件或目录的兄弟文件或目录。
- SKIP_SUBTREE：代表“继续访问”的后续行为，但不访问该文件或目录的子目录树。
- TERMINATE：代表“中止访问”的后续行为。

实际编程时没必要为 FileVisitor 的 4 个方法都提供实现，可以通过继承 SimpleFileVisitor（FileVisitor 的实现类）来实现自己的“文件访问器”，这样就根据需要、选择性地重写指定方法了。

如下程序示范了使用 FileVisitor 来遍历文件和子目录。

程序清单：codes\15\15.10\FileVisitorTest.java

```
public class FileVisitorTest
{
    public static void main(String[] args)
        throws Exception
    {
        // 遍历 g:\publish\codes\15 目录下的所有文件和子目录
        Files.walkFileTree(Paths.get("g:", "publish" , "codes" , "15")
            , new SimpleFileVisitor<Path>()
        {
            // 访问文件时触发该方法
            @Override
            public FileVisitResult visitFile(Path file
                , BasicFileAttributes attrs) throws IOException
            {
                System.out.println("正在访问" + file + "文件");
                // 找到了 FileVisitorTest.java 文件
                if (file.endsWith("FileVisitorTest.java"))
                {
                    System.out.println("--已经找到目标文件--");
                    return FileVisitResult.TERMINATE;
                }
                return FileVisitResult.CONTINUE;
            }
            // 开始访问目录时触发该方法
            @Override
```

```
            public FileVisitResult preVisitDirectory(Path dir
                , BasicFileAttributes attrs) throws IOException
            {
                System.out.println("正在访问: " + dir + " 路径");
                return FileVisitResult.CONTINUE;
            }
        });
    }
}
```

上面程序中使用了Files工具类的walkFileTree()方法来遍历g:\publish\codes\15目录下的所有文件和子目录，如果找到的文件以“FileVisitorTest.java”结尾，则程序停止遍历——这就实现了对指定目录进行搜索，直到找到指定文件为止。

15.10.3 使用WatchService监控文件变化

在以前的Java版本中，如果程序需要监控文件的变化，则可以考虑启动一条后台线程，这条后台线程每隔一段时间去“遍历”一次指定目录的文件，如果发现此次遍历结果与上次遍历结果不同，则认为文件发生了变化。但这种方式不仅十分烦琐，而且性能也不好。

NIO.2的Path类提供了如下一个方法来监听文件系统的变化。

- register(WatchService watcher, WatchEvent.Kind<?>... events)：用watcher监听该path代表的目录下的文件变化。events参数指定要监听哪些类型的事件。

在这个方法中WatchService代表一个文件系统监听服务，它负责监听path代表的目录下的文件变化。一旦使用register()方法完成注册之后，接下来就可调用WatchService的如下三个方法来获取被监听目录的文件变化事件。

- WatchKey poll()：获取下一个WatchKey，如果没有WatchKey发生就立即返回null。
- WatchKey poll(long timeout, TimeUnit unit)：尝试等待 timeout时间去获取下一个WatchKey。
- WatchKey take()：获取下一个WatchKey，如果没有WatchKey发生就一直等待。

如果程序需要一直监控，则应该选择使用take()方法；如果程序只需要监控指定时间，则可考虑使用poll()方法。下面程序示范了使用WatchService来监控C:盘根路径下文件的变化。

程序清单：codes\15\15.10\WatchServiceTest.java

```
public class WatchServiceTest
{
    public static void main(String[] args)
        throws Exception
    {
        // 获取文件系统的WatchService对象
        WatchService watchService = FileSystems.getDefault()
            .newWatchService();
        // 为C:盘根路径注册监听
        Paths.get("C:/").register(watchService
            , StandardWatchEventKinds.ENTRY_CREATE
            , StandardWatchEventKinds.ENTRY_MODIFY
            , StandardWatchEventKinds.ENTRY_DELETE);
        while(true)
        {
            // 获取下一个文件变化事件
            WatchKey key = watchService.take();  // ①
            for (WatchEvent<?> event : key.pollEvents())
            {
                System.out.println(event.context() +" 文件发生了 "
                    + event.kind()+ "事件! ");
            }
            // 重设WatchKey
            boolean valid = key.reset();
            // 如果重设失败，退出监听
            if (!valid)
            {
```

```
                break;
            }
        }
    }
}
```

上面程序使用了一个死循环重复获取 C:盘根路径下文件的变化，程序在①号代码处试图获取下一个 WatchKey，如果没有发生就等待。因此 C:盘根路径下每次文件的变化都会被该程序监听到。

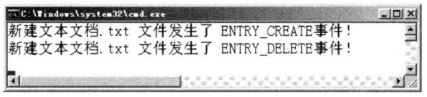


图 15.22 监控文件的变化

运行该程序，然后在 C:盘下新建一个文件，再删除该文件，将看到如图 15.22 所示的输出。

从图 15.22 不难看出，通过使用 WatchService 可以非常优雅地监控指定目录下文件的变化，至于文件发生变化后，程序应该进行哪些处理，这就取决于程序的业务需要了。

15.10.4 访问文件属性

早期的 Java 提供的 File 类可以访问一些简单的文件属性，比如文件大小、修改时间、文件是否隐藏、是文件还是目录等。如果程序需要获取或修改更多的文件属性，则必须利用运行所在平台的特定代码来实现，这是一件非常困难的事情。

Java 7 的 NIO.2 在 java.nio.file.attribute 包下提供了大量的工具类，通过这些工具类，开发者可以非常简单地读取、修改文件属性。这些工具类主要分为如下两类。

- XxxAttributeView：代表某种文件属性的“视图”。
- XxxAttributes：代表某种文件属性的“集合”，程序一般通过 XxxAttributeView 对象来获取 XxxAttributes。

在这些工具类中，FileAttributeView 是其他 XxxAttributeView 的父接口，下面简单介绍一下这些 XxxAttributeView。

AclFileAttributeView：通过 AclFileAttributeView，开发者可以为特定文件设置 ACL（Access Control List）及文件所有者属性。它的 getAcl()方法返回 List<AclEntry>对象，该返回值代表了该文件的权限集。通过 setAcl(List)方法可以修改该文件的 ACL。

BasicFileAttributeView：它可以获取或修改文件的基本属性，包括文件的最后修改时间、最后访问时间、创建时间、大小、是否为目录、是否为符号链接等。它的 readAttributes()方法返回一个 BasicFileAttributes 对象，对文件夹基本属性的修改是通过 BasicFileAttributes 对象完成的。

DosFileAttributeView：它主要用于获取或修改文件 DOS 相关属性，比如文件是否只读、是否隐藏、是否为系统文件、是否是存档文件等。它的 readAttributes()方法返回一个 DosFileAttributes 对象，对这些属性的修改其实是由 DosFileAttributes 对象来完成的。

FileOwnerAttributeView：它主要用于获取或修改文件的所有者。它的 getOwner()方法返回一个 UserPrincipal 对象来代表文件所有者；也可调用 setOwner(UserPrincipal owner)方法来改变文件的所有者。

PosixFileAttributeView：它主要用于获取或修改 POSIX（Portable Operating System Interface of INIX）属性，它的 readAttributes()方法返回一个 PosixFileAttributes 对象，该对象可用于获取或修改文件的所有者、组所有者、访问权限信息（就是 UNIX 的 chmod 命令负责干的事情）。这个 View 只在 UNIX、Linux 等系统上有用。

UserDefinedFileAttributeView：它可以让开发者为文件设置一些自定义属性。

下面程序示范了如何读取、修改文件的属性。

程序清单：codes\15\15.10\AttributeViewTest.java

```
public class AttributeViewTest
{
    public static void main(String[] args)
        throws Exception
    {
        // 获取将要操作的文件
        Path testPath = Paths.get("AttributeViewTest.java");
        // 获取访问基本属性的 BasicFileAttributeView
        BasicFileAttributeView basicView = Files.getFileAttributeView(
```

```
            testPath , BasicFileAttributeView.class);
        // 获取访问基本属性的 BasicFileAttributes
        BasicFileAttributes basicAttribs = basicView.readAttributes();
        // 访问文件的基本属性
        System.out.println("创建时间: " + new Date(basicAttribs
            .creationTime().toMillis()));
        System.out.println("最后访问时间: " + new Date(basicAttribs
            .lastAccessTime().toMillis()));
        System.out.println("最后修改时间: " + new Date(basicAttribs
            .lastModifiedTime().toMillis()));
        System.out.println("文件大小: " + basicAttribs.size());
        // 获取访问文件属主信息的 FileOwnerAttributeView
        FileOwnerAttributeView ownerView = Files.getFileAttributeView(
            testPath, FileOwnerAttributeView.class);
        // 获取该文件所属的用户
        System.out.println(ownerView.getOwner());
        // 获取系统中 guest 对应的用户
        UserPrincipal user = FileSystems.getDefault()
            .getUserPrincipalLookupService()
            .lookupPrincipalByName("guest");
        // 修改用户
        ownerView.setOwner(user);
        // 获取访问自定义属性的 FileOwnerAttributeView
        UserDefinedFileAttributeView userView = Files.getFileAttributeView(
            testPath, UserDefinedFileAttributeView.class);
        List<String> attrNames = userView.list();
        // 遍历所有的自定义属性
        for (String name : attrNames)
        {
            ByteBuffer buf = ByteBuffer.allocate(userView.size(name));
            userView.read(name, buf);
            buf.flip();
            String value = Charset.defaultCharset().decode(buf).toString();
            System.out.println(name + "--->" + value) ;
        }
        // 添加一个自定义属性
        userView.write("发行者", Charset.defaultCharset()
            .encode("疯狂 Java 联盟"));
        // 获取访问 DOS 属性的 DosFileAttributeView
        DosFileAttributeView dosView = Files.getFileAttributeView(testPath
            , DosFileAttributeView.class);
        // 将文件设置隐藏、只读
        dosView.setHidden(true);
        dosView.setReadOnly(true);
    }
}
```

上面程序中的 4 段粗体字代码分别访问了 4 种不同类型的文件属性，关于读取、修改文件属性的说明，程序中的代码已有详细说明，因此不再过多地解释。第二次运行该程序（记住第一次运行后 AttributeViewTest.java 文件变成隐藏、只读文件，因此第二次运行之前一定要先取消只读属性），将看到如图 15.23 所示的输出。

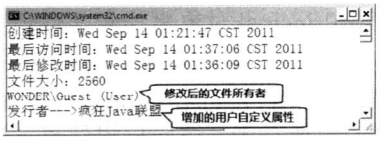


图 15.23　读取、修改文件属性

15.11　本章小结

本章主要介绍了 Java 输入/输出体系的相关知识。本章介绍了如何使用 File 来访问本地文件系统，以及 Java IO 流的三种分类方式。本章重点讲解了 IO 流的处理模型，以及如何使用 IO 流来读取物理存储节点中的数据，归纳了 Java 不同 IO 流的功能，并介绍了几种典型 IO 流的用法。本章也介绍了 RandomAccessFile 类的用法，通过 RandomAccessFile 允许程序自由地移动文件指针，任意访问文件的指定位置。

本章也介绍了 Java 对象序列化的相关知识，程序通过序列化把 Java 对象转换成二进制字节流，然

后就可以把二进制字节流写入网络或者永久存储器。本章还介绍了 Java 提供的新 IO 支持，使用新 IO 能以更高效的方式来进行输入、输出操作。本章最后介绍了 Java 7 提供的 NIO.2 的文件 IO 和文件系统访问支持，NIO.2 极大地增强了 Java IO 的功能。

➤➤ 本章练习

1. 定义一个工具类，该类要求用户运行该程序时输入一个路径。该工具类会将该路径下（及其子目录下）的所有文件列出来。

2. 定义一个工具类，该类要求用户运行该程序时输入一个路径。该工具类会将该路径下的文件、文件夹的数量统计出来。

3. 定义一个工具类，该工具类可实现 copy 功能（不允许使用 Files 类）。如果被 copy 的对象是文件，程序将指定文件复制到指定目录下；如果被 copy 的对象是目录，程序应该将该目录及其目录下的所有文件复制到指定目录下。

4. 编写仿 Windows 记事本的小程序。

5. 编写一个命令行工具，这个命令行工具就像 Windows 提供的 cmd 命令一样，可以执行各种常见的命令，如 dir、md、copy、move 等。

6. 完善第 12 章的仿 EditPlus 的编辑器，提供文件的打开、保存等功能。

CHAPTER

16

第16章 多线程

本章要点

- 线程的基础知识
- 理解线程和进程的区别与联系
- 三种创建线程的方式
- 线程的 run()方法和 start()方法的区别与联系
- 线程的生命周期
- 线程死亡的几种情况
- 控制线程的常用方法
- 线程同步的概念和必要性
- 使用 synchronized 控制线程同步
- 使用 Lock 对象控制线程同步
- 使用 Object 提供的方法实现线程通信
- 使用条件变量实现线程通信
- 使用阻塞队列实现线程通信
- 线程组的功能和用法
- 线程池的功能和用法
- Java 8 增强的 ForkJoinPool
- ThreadLocal 类的功能和用法
- 使用线程安全的集合类
- Java 9 新增的发布-订阅框架

前面大部分程序，都只是在做单线程的编程，前面所有程序（除第 11 章、第 12 章的程序之外，它们有内建的多线程支持）都只有一条顺序执行流——程序从 main()方法开始执行，依次向下执行每行代码，如果程序执行某行代码时遇到了阻塞，则程序将会停滞在该处。如果使用 IDE 工具的单步调试功能，就可以非常清楚地看出这一点。

但实际的情况是，单线程的程序往往功能非常有限，例如开发一个简单的服务器程序，这个服务器程序需要向不同的客户端提供服务时，不同的客户端之间应该互不干扰，否则会让客户端感觉非常沮丧。多线程听上去是非常专业的概念，其实非常简单——单线程的程序（前面介绍的绝大部分程序）只有一个顺序执行流，多线程的程序则可以包括多个顺序执行流，多个顺序流之间互不干扰。可以这样理解：单线程的程序如同只雇佣一个服务员的餐厅，他必须做完一件事情后才可以做下一件事情；多线程的程序则如同雇佣多个服务员的餐厅，他们可以同时做多件事情。

Java 语言提供了非常优秀的多线程支持，程序可以通过非常简单的方式来启动多线程。本章将会详细介绍 Java 多线程编程的相关方面，包括创建、启动线程、控制线程，以及多线程的同步操作，并会介绍如何利用 Java 内建支持的线程池来提高多线程性能。

16.1 线程概述

几乎所有的操作系统都支持同时运行多个任务，一个任务通常就是一个程序，每个运行中的程序就是一个进程。当一个程序运行时，内部可能包含了多个顺序执行流，每个顺序执行流就是一个线程。

16.1.1 线程和进程

几乎所有的操作系统都支持进程的概念，所有运行中的任务通常对应一个进程（Process）。当一个程序进入内存运行时，即变成一个进程。进程是处于运行过程中的程序，并且具有一定的独立功能，进程是系统进行资源分配和调度的一个独立单位。

一般而言，进程包含如下三个特征。

- 独立性：进程是系统中独立存在的实体，它可以拥有自己独立的资源，每一个进程都拥有自己私有的地址空间。在没有经过进程本身允许的情况下，一个用户进程不可以直接访问其他进程的地址空间。
- 动态性：进程与程序的区别在于，程序只是一个静态的指令集合，而进程是一个正在系统中活动的指令集合。在进程中加入了时间的概念。进程具有自己的生命周期和各种不同的状态，这些概念在程序中都是不具备的。
- 并发性：多个进程可以在单个处理器上并发执行，多个进程之间不会互相影响。

注意：

并发性（concurrency）和并行性（parallel）是两个概念，并行指在同一时刻，有多条指令在多个处理器上同时执行；并发指在同一时刻只能有一条指令执行，但多个进程指令被快速轮换执行，使得在宏观上具有多个进程同时执行的效果。

大部分操作系统都支持多进程并发运行，现代的操作系统几乎都支持同时运行多个任务。例如，程序员一边开着开发工具在写程序，一边开着参考手册备查，同时还使用电脑播放音乐……除此之外，每台电脑运行时还有大量底层的支撑性程序在运行……这些进程看上去像是在同时工作。

但事实的真相是，对于一个 CPU 而言，它在某个时间点只能执行一个程序，也就是说，只能运行一个进程，CPU 不断地在这些进程之间轮换执行。那为什么用户感觉不到任何中断现象呢？这是因为 CPU 的执行速度相对人的感觉来说实在是太快了（如果启动的程序足够多，用户依然可以感觉到程序的运行速度下降），所以虽然 CPU 在多个进程之间轮换执行，但用户感觉到好像有多个进程在同时执行。

现代的操作系统都支持多进程的并发，但在具体的实现细节上可能因为硬件和操作系统的不同而采用不同的策略。比较常用的方式有：共用式的多任务操作策略，例如 Windows 3.1 和 Mac OS 9；目前操

作系统大多采用效率更高的抢占式多任务操作策略，例如 Windows NT、Windows 2000 以及 UNIX/Linux 等操作系统。

多线程则扩展了多进程的概念，使得同一个进程可以同时并发处理多个任务。线程（Thread）也被称作轻量级进程（Lightweight Process），线程是进程的执行单元。就像进程在操作系统中的地位一样，线程在程序中是独立的、并发的执行流。当进程被初始化后，主线程就被创建了。对于绝大多数的应用程序来说，通常仅要求有一个主线程，但也可以在该进程内创建多条顺序执行流，这些顺序执行流就是线程，每个线程也是互相独立的。

线程是进程的组成部分，一个进程可以拥有多个线程，一个线程必须有一个父进程。线程可以拥有自己的堆栈、自己的程序计数器和自己的局部变量，但不拥有系统资源，它与父进程的其他线程共享该进程所拥有的全部资源。因为多个线程共享父进程里的全部资源，因此编程更加方便；但必须更加小心，因为需要确保线程不会妨碍同一进程里的其他线程。

线程可以完成一定的任务，可以与其他线程共享父进程中的共享变量及部分环境，相互之间协同来完成进程所要完成的任务。

线程是独立运行的，它并不知道进程中是否还有其他线程存在。线程的执行是抢占式的，也就是说，当前运行的线程在任何时候都可能被挂起，以便另外一个线程可以运行。

一个线程可以创建和撤销另一个线程，同一个进程中的多个线程之间可以并发执行。

从逻辑角度来看，多线程存在于一个应用程序中，让一个应用程序中可以有多个执行部分同时执行，但操作系统无须将多个线程看作多个独立的应用，对多线程实现调度和管理以及资源分配。线程的调度和管理由进程本身负责完成。

简而言之，一个程序运行后至少有一个进程，一个进程里可以包含多个线程，但至少要包含一个线程。

提示：

归纳起来可以这样说：操作系统可以同时执行多个任务，每个任务就是进程；进程可以同时执行多个任务，每个任务就是线程。

16.1.2 多线程的优势

线程在程序中是独立的、并发的执行流，与分隔的进程相比，进程中线程之间的隔离程度要小。它们共享内存、文件句柄和其他每个进程应有的状态。

因为线程的划分尺度小于进程，使得多线程程序的并发性高。进程在执行过程中拥有独立的内存单元，而多个线程共享内存，从而极大地提高了程序的运行效率。

线程比进程具有更高的性能，这是由于同一个进程中的线程都有共性——多个线程共享同一个进程虚拟空间。线程共享的环境包括：进程代码段、进程的公有数据等。利用这些共享的数据，线程很容易实现相互之间的通信。

当操作系统创建一个进程时，必须为该进程分配独立的内存空间，并分配大量的相关资源；但创建一个线程则简单得多，因此使用多线程来实现并发比使用多进程实现并发的性能要高得多。

总结起来，使用多线程编程具有如下几个优点。

- 进程之间不能共享内存，但线程之间共享内存非常容易。
- 系统创建进程时需要为该进程重新分配系统资源，但创建线程则代价小得多，因此使用多线程来实现多任务并发比多进程的效率高。
- Java 语言内置了多线程功能支持，而不是单纯地作为底层操作系统的调度方式，从而简化了 Java 的多线程编程。

在实际应用中，多线程是非常有用的，一个浏览器必须能同时下载多个图片；一个 Web 服务器必须能同时响应多个用户请求；Java 虚拟机本身就在后台提供了一个超级线程来进行垃圾回收；图形用户界面（GUI）应用也需要启动单独的线程从主机环境收集用户界面事件……总之，多线程在实际编程中的应用是非常广泛的。

16.2 线程的创建和启动

Java 使用 Thread 类代表线程，所有的线程对象都必须是 Thread 类或其子类的实例。每个线程的作用是完成一定的任务，实际上就是执行一段程序流（一段顺序执行的代码）。Java 使用线程执行体来代表这段程序流。

16.2.1 继承 Thread 类创建线程类

通过继承 Thread 类来创建并启动多线程的步骤如下。

① 定义 Thread 类的子类，并重写该类的 run()方法，该 run()方法的方法体就代表了线程需要完成的任务。因此把 run()方法称为线程执行体。

② 创建 Thread 子类的实例，即创建了线程对象。

③ 调用线程对象的 start()方法来启动该线程。

下面程序示范了通过继承 Thread 类来创建并启动多线程。

程序清单：codes\16\16.2\FirstThread.java

```
// 通过继承 Thread 类来创建线程类
public class FirstThread extends Thread
{
    private int i ;
    // 重写 run()方法，run()方法的方法体就是线程执行体
    public void run()
    {
        for ( ; i < 100 ; i++ )
        {
            // 当线程类继承 Thread 类时，直接使用 this 即可获取当前线程
            // Thread 对象的 getName()返回当前线程的名字
            // 因此可以直接调用 getName()方法返回当前线程的名字
            System.out.println(getName() +  " " + i);
        }
    }
    public static void main(String[] args)
    {
        for (int i = 0; i < 100;  i++)
        {
            // 调用 Thread 的 currentThread()方法获取当前线程
            System.out.println(Thread.currentThread().getName()
                +  " " + i);
            if (i == 20)
            {
                // 创建并启动第一个线程
                new FirstThread().start();
                // 创建并启动第二个线程
                new FirstThread().start();
            }
        }
    }
}
```

上面程序中的 FirstThread 类继承了 Thread 类，并实现了 run()方法，如程序中第一段粗体字代码所示，该 run()方法里的代码执行流就是该线程所需要完成的任务。程序的主方法中也包含了一个循环，当循环变量 i 等于 20 时创建并启动两个新线程。运行上面程序，会看到如图 16.1 所示的界面。

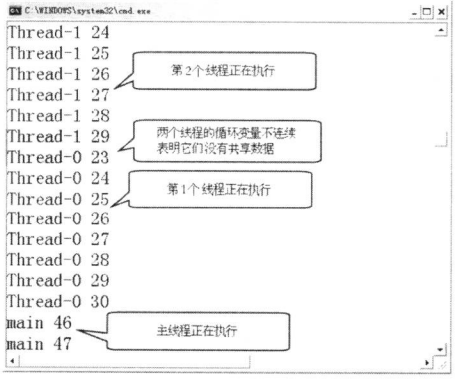


图 16.1 多线程运行的效果

虽然上面程序只显式地创建并启动了 2 个线程，但实际上程序有 3 个线程，即程序显式创建的 2 个子线程和主线程。前面已经提到，当 Java 程序开始运行后，程序至少会创建一个主线程，主线程的线程执

行体不是由 run()方法确定的，而是由 main()方法确定的——main()方法的方法体代表主线程的线程执行体。

注意：

进行多线程编程时不要忘记了 Java 程序运行时默认的主线程，main()方法的方法体就是主线程的线程执行体。

除此之外，上面程序还用到了线程的如下两个方法。

- Thread.currentThread()：currentThread()是 Thread 类的静态方法，该方法总是返回当前正在执行的线程对象。
- getName()：该方法是 Thread 类的实例方法，该方法返回调用该方法的线程名字。

提示：

程序可以通过 setName(String name)方法为线程设置名字，也可以通过 getName()方法返回指定线程的名字。在默认情况下，主线程的名字为 main，用户启动的多个线程的名字依次为 Thread-0、Thread-1、Thread-2、…、Thread-*n* 等。

从图 16.1 中的灰色覆盖区域可以看出，Thread-0 和 Thread-1 两个线程输出的 i 变量不连续——注意：i 变量是 FirstThread 的实例变量，而不是局部变量，但因为程序每次创建线程对象时都需要创建一个 FirstThread 对象，所以 Thread-0 和 Thread-1 不能共享该实例变量。

注意：

使用继承 Thread 类的方法来创建线程类时，多个线程之间无法共享线程类的实例变量。

16.2.2 实现 Runnable 接口创建线程类

实现 Runnable 接口来创建并启动多线程的步骤如下。

① 定义 Runnable 接口的实现类，并重写该接口的 run()方法，该 run()方法的方法体同样是该线程的线程执行体。

② 创建 Runnable 实现类的实例，并以此实例作为 Thread 的 target 来创建 Thread 对象，该 Thread 对象才是真正的线程对象。代码如下所示。

```
// 创建 Runnable 实现类的对象
SecondThread st = new SecondThread();
// 以 Runnable 实现类的对象作为 Thread 的 target 来创建 Thread 对象，即线程对象
new Thread(st);
```

也可以在创建 Thread 对象时为该 Thread 对象指定一个名字，代码如下所示。

```
// 创建 Thread 对象时指定 target 和新线程的名字
new Thread(st , "新线程 1");
```

提示：

Runnable 对象仅仅作为 Thread 对象的 target，Runnable 实现类里包含的 run()方法仅作为线程执行体。而实际的线程对象依然是 Thread 实例，只是该 Thread 线程负责执行其 target 的 run()方法。

③ 调用线程对象的 start()方法来启动该线程。

下面程序示范了通过实现 Runnable 接口来创建并启动多线程。

程序清单：codes\16\16.2\SecondThread.java

```
// 通过实现 Runnable 接口来创建线程类
public class SecondThread implements Runnable
{
    private int i ;
    // run()方法同样是线程执行体
```

```
    public void run()
    {
        for ( ; i < 100 ; i++ )
        {
            // 当线程类实现 Runnable 接口时
            // 如果想获取当前线程，只能用 Thread.currentThread()方法
            System.out.println(Thread.currentThread().getName()
                + "  " + i);
        }
    }
    public static void main(String[] args)
    {
        for (int i = 0; i < 100;  i++)
        {
            System.out.println(Thread.currentThread().getName()
                + "  " + i);
            if (i == 20)
            {
                SecondThread st = new SecondThread();     // ①
                // 通过 new Thread(target , name)方法创建新线程
                new Thread(st , "新线程 1").start();
                new Thread(st , "新线程 2").start();
            }
        }
    }
}
```

上面程序中的粗体字代码部分实现了 run()方法，也就是定义了该线程的线程执行体。对比 FirstThread 中的 run()方法体和 SecondThread 中的 run()方法体不难发现，通过继承 Thread 类来获得当前线程对象比较简单，直接使用 this 就可以了；但通过实现 Runnable 接口来获得当前线程对象，则必须使用 Thread.currentThread()方法。

提示：

Runnable 接口中只包含一个抽象方法，从 Java 8 开始，Runnable 接口使用了@FunctionalInterface 修饰。也就是说，Runnable 接口是函数式接口，可使用 Lambda 表达式创建 Runnable 对象。接下来介绍的 Callable 接口也是函数式接口。

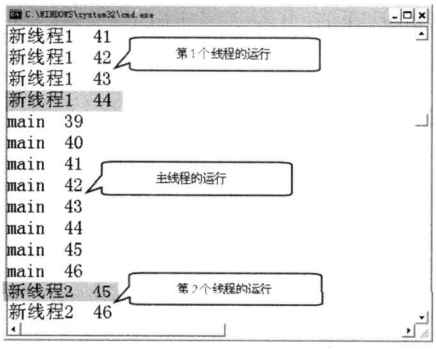


图 16.2 实现 Runnable 接口创建的多线程

除此之外，上面程序中的粗体字代码创建了两个 Thread 对象，并调用 start()方法来启动这两个线程。在 FirstThread 和 SecondThread 中创建线程对象的方式有所区别：前者直接创建的 Thread 子类即可代表线程对象；后者创建的 Runnable 对象只能作为线程对象的 target。

运行上面程序，会看到如图 16.2 所示的界面。

从图 16.2 中的两个灰色覆盖区域可以看出，两个子线程的 i 变量是连续的，也就是采用 Runnable 接口的方式创建的多个线程可以共享线程类的实例变量。这是因为在这种方式下，程序所创建的 Runnable 对象只是线程的 target，而多个线程可以共享同一个 target，所以多个线程可以共享同一个线程类（实际上应该是线程的 target 类）的实例变量。

16.2.3 使用 Callable 和 Future 创建线程

前面已经指出，通过实现 Runnable 接口创建多线程时，Thread 类的作用就是把 run()方法包装成线程执行体。那么是否可以直接把任意方法都包装成线程执行体呢？Java 目前不行！但 Java 的模仿者 C#可以（C#可以把任意方法包装成线程执行体，包括有返回值的方法）。

也许受此启发，从 Java 5 开始，Java 提供了 Callable 接口，该接口怎么看都像是 Runnable 接口的增强版，Callable 接口提供了一个 call()方法可以作为线程执行体，但 call()方法比 run()方法功能更强大。

- call()方法可以有返回值。

➢ call()方法可以声明抛出异常。

因此完全可以提供一个 Callable 对象作为 Thread 的 target，而该线程的线程执行体就是该 Callable 对象的 call()方法。问题是：Callable 接口是 Java 5 新增的接口，而且它不是 Runnable 接口的子接口，所以 Callable 对象不能直接作为 Thread 的 target。而且 call()方法还有一个返回值——call()方法并不是直接调用，它是作为线程执行体被调用的。那么如何获取 call()方法的返回值呢？

Java 5 提供了 Future 接口来代表 Callable 接口里 call()方法的返回值，并为 Future 接口提供了一个 FutureTask 实现类，该实现类实现了 Future 接口，并实现了 Runnable 接口——可以作为 Thread 类的 target。

在 Future 接口里定义了如下几个公共方法来控制它关联的 Callable 任务。

➢ boolean cancel(boolean mayInterruptIfRunning)：试图取消该 Future 里关联的 Callable 任务。

➢ V get()：返回 Callable 任务里 call()方法的返回值。调用该方法将导致程序阻塞，必须等到子线程结束后才会得到返回值。

➢ V get(long timeout, TimeUnit unit)：返回 Callable 任务里 call()方法的返回值。该方法让程序最多阻塞 timeout 和 unit 指定的时间，如果经过指定时间后 Callable 任务依然没有返回值，将会抛出 TimeoutException 异常。

➢ boolean isCancelled()：如果在 Callable 任务正常完成前被取消，则返回 true。

➢ boolean isDone()：如果 Callable 任务已完成，则返回 true。

注意：

Callable 接口有泛型限制，Callable 接口里的泛型形参类型与 call()方法返回值类型相同。而且 Callable 接口是函数式接口，因此可使用 Lambda 表达式创建 Callable 对象。

创建并启动有返回值的线程的步骤如下。

① 创建 Callable 接口的实现类，并实现 call()方法，该 call()方法将作为线程执行体，且该 call()方法有返回值，再创建 Callable 实现类的实例。从 Java 8 开始，可以直接使用 Lambda 表达式创建 Callable 对象。

② 使用 FutureTask 类来包装 Callable 对象，该 FutureTask 对象封装了该 Callable 对象的 call()方法的返回值。

③ 使用 FutureTask 对象作为 Thread 对象的 target 创建并启动新线程。

④ 调用 FutureTask 对象的 get()方法来获得子线程执行结束后的返回值。

下面程序通过实现 Callable 接口来实现线程类，并启动该线程。

程序清单：codes\16\16.2\ThirdThread.java

```
public class ThirdThread
{
    public static void main(String[] args)
    {
        // 创建 Callable 对象
        ThirdThread rt = new ThirdThread();
        // 先使用 Lambda 表达式创建 Callable<Integer>对象
        // 使用 FutureTask 来包装 Callable 对象
        FutureTask<Integer> task = new FutureTask<Integer>((Callable<Integer>)()->{
            int i = 0;
            for ( ; i < 100 ; i++ )
            {
                System.out.println(Thread.currentThread().getName()
                    + " 的循环变量 i 的值: " + i);
            }
            // call()方法可以有返回值
            return i;
        });
        for (int i = 0 ; i < 100 ; i++)
        {
            System.out.println(Thread.currentThread().getName()
                + " 的循环变量 i 的值: " + i);
            if (i == 20)
            {
```

```
                // 实质还是以 Callable 对象来创建并启动线程的
                new Thread(task , "有返回值的线程").start();
            }
        }
        try
        {
            // 获取线程返回值
            System.out.println("子线程的返回值：" + task.get());
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
        }
    }
}
```

上面程序中使用 Lambda 表达式直接创建了 Callable 对象，这样就无须先创建 Callable 实现类，再创建 Callable 对象了。实现 Callable 接口与实现 Runnable 接口并没有太大的差别，只是 Callable 的 call()方法允许声明抛出异常，而且允许带返回值。

上面程序中的粗体字代码是以 Callable 对象来启动线程的关键代码。程序先使用 Lambda 表达式创建一个 Callable 对象，然后将该实例包装成一个 FutureTask 对象。主线程中当循环变量 i 等于 20 时，程序启动以 FutureTask 对象为 target 的线程。程序最后调用 FutureTask 对象的 get()方法来返回 call()方法的返回值——该方法将导致主线程被阻塞，直到 call()方法结束并返回为止。

运行上面程序，将看到主线程和 call()方法所代表的线程交替执行的情形，程序最后还会输出 call()方法的返回值。

16.2.4 创建线程的三种方式对比

通过继承 Thread 类或实现 Runnable、Callable 接口都可以实现多线程，不过实现 Runnable 接口与实现 Callable 接口的方式基本相同，只是 Callable 接口里定义的方法有返回值，可以声明抛出异常而已。因此可以将实现 Runnable 接口和实现 Callable 接口归为一种方式。这种方式与继承 Thread 方式之间的主要差别如下。

采用实现 Runnable、Callable 接口的方式创建多线程的优缺点：

- 线程类只是实现了 Runnable 接口或 Callable 接口，还可以继承其他类。
- 在这种方式下，多个线程可以共享同一个 target 对象，所以非常适合多个相同线程来处理同一份资源的情况，从而可以将 CPU、代码和数据分开，形成清晰的模型，较好地体现了面向对象的思想。
- 劣势是，编程稍稍复杂，如果需要访问当前线程，则必须使用 Thread.currentThread()方法。

采用继承 Thread 类的方式创建多线程的优缺点：

- 劣势是，因为线程类已经继承了 Thread 类，所以不能再继承其他父类。
- 优势是，编写简单，如果需要访问当前线程，则无须使用 Thread.currentThread()方法，直接使用 this 即可获得当前线程。

鉴于上面分析，因此一般推荐采用实现 Runnable 接口、Callable 接口的方式来创建多线程。

16.3 线程的生命周期

当线程被创建并启动以后，它既不是一启动就进入了执行状态，也不是一直处于执行状态，在线程的生命周期中，它要经过新建（New）、就绪（Runnable）、运行（Running）、阻塞（Blocked）和死亡（Dead）5 种状态。尤其是当线程启动以后，它不可能一直“霸占”着 CPU 独自运行，所以 CPU 需要在多条线程之间切换，于是线程状态也会多次在运行、就绪之间切换。

16.3.1 新建和就绪状态

当程序使用 new 关键字创建了一个线程之后，该线程就处于新建状态，此时它和其他的 Java 对象

一样，仅仅由 Java 虚拟机为其分配内存，并初始化其成员变量的值。此时的线程对象没有表现出任何线程的动态特征，程序也不会执行线程的线程执行体。

当线程对象调用了 start()方法之后，该线程处于就绪状态，Java 虚拟机会为其创建方法调用栈和程序计数器，处于这个状态中的线程并没有开始运行，只是表示该线程可以运行了。至于该线程何时开始运行，取决于 JVM 里线程调度器的调度。

注意：

启动线程使用 start()方法，而不是 run()方法！永远不要调用线程对象的 run()方法！调用 start()方法来启动线程，系统会把该 run()方法当成线程执行体来处理；但如果直接调用线程对象的 run()方法，则 run()方法立即就会被执行，而且在 run()方法返回之前其他线程无法并发执行——也就是说，如果直接调用线程对象的 run()方法，系统把线程对象当成一个普通对象，而 run()方法也是一个普通方法，而不是线程执行体。

程序清单：codes\16\16.3\InvokeRun.java

```
public class InvokeRun extends Thread
{
    private int i ;
    // 重写 run()方法，run()方法的方法体就是线程执行体
    public void run()
    {
        for ( ; i < 100 ; i++ )
        {
            // 直接调用 run()方法时，Thread 的 this.getName()返回的是该对象的名字
            // 而不是当前线程的名字
            // 使用 Thread.currentThread().getName()总是获取当前线程的名字
            System.out.println(Thread.currentThread().getName()
                + " " + i);   // ①
        }
    }
    public static void main(String[] args)
    {
        for (int i = 0; i < 100;  i++)
        {
            // 调用 Thread 的 currentThread()方法获取当前线程
            System.out.println(Thread.currentThread().getName()
                + " " + i);
            if (i == 20)
            {
                // 直接调用线程对象的 run()方法
                // 系统会把线程对象当成普通对象，把 run()方法当成普通方法
                // 所以下面两行代码并不会启动两个线程，而是依次执行两个 run()方法
                new InvokeRun().run();
                new InvokeRun().run();
            }
        }
    }
}
```

上面程序创建线程对象后直接调用了线程对象的 run()方法（如粗体字代码所示），程序运行的结果是整个程序只有一个线程：主线程。还有一点需要指出，如果直接调用线程对象的 run()方法，则 run()方法里不能直接通过 getName()方法来获得当前执行线程的名字，而是需要使用 Thread.currentThread()方法先获得当前线程，再调用线程对象的 getName()方法来获得线程的名字。

通过上面程序不难看出，启动线程的正确方法是调用 Thread 对象的 start()方法，而不是直接调用 run()方法，否则就变成单线程程序了。

需要指出的是，调用了线程的 run()方法之后，该线程已经不再处于新建状态，不要再次调用线程对象的 start()方法。

注意：

只能对处于新建状态的线程调用 start()方法，否则将引发 IllegalThreadStateException 异常。

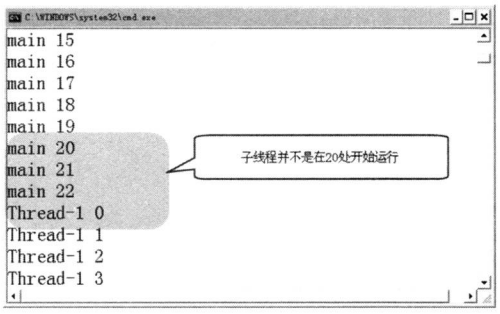


图 16.3 调用 start()方法后的线程并没有立即运行

调用线程对象的 start()方法之后，该线程立即进入就绪状态——就绪状态相当于“等待执行”，但该线程并未真正进入运行状态。这一点可以通过再次运行 16.2 节中的 FirstThread 或 SecondThread 来证明。再次运行该程序，会看到如图 16.3 所示的输出。

从图 16.3 中可以看出，主线程在 i 等于 20 时调用了子线程的 start()方法来启动当前线程，但当前线程并没有立即执行，而是等到 i 为 22 时才看到子线程开始执行（读者运行时不一定是 22 时切换，这种切换由底层平台控制，具有一定的随机性）。

提示：

如果希望调用子线程的 start()方法后子线程立即开始执行，程序可以使用 Thread.sleep(1)来让当前运行的线程（主线程）睡眠 1 毫秒——1 毫秒就够了，因为在这 1 毫秒内 CPU 不会空闲，它会去执行另一个处于就绪状态的线程，这样就可以让子线程立即开始执行。

16.3.2 运行和阻塞状态

如果处于就绪状态的线程获得了 CPU，开始执行 run()方法的线程执行体，则该线程处于运行状态，如果计算机只有一个 CPU，那么在任何时刻只有一个线程处于运行状态。当然，在一个多处理器的机器上，将会有多个线程并行（注意是并行：parallel）执行；当线程数大于处理器数时，依然会存在多个线程在同一个 CPU 上轮换的现象。

当一个线程开始运行后，它不可能一直处于运行状态（除非它的线程执行体足够短，瞬间就执行结束了），线程在运行过程中需要被中断，目的是使其他线程获得执行的机会，线程调度的细节取决于底层平台所采用的策略。对于采用抢占式策略的系统而言，系统会给每个可执行的线程一个小时间段来处理任务；当该时间段用完后，系统就会剥夺该线程所占用的资源，让其他线程获得执行的机会。在选择下一个线程时，系统会考虑线程的优先级。

所有现代的桌面和服务器操作系统都采用抢占式调度策略，但一些小型设备如手机则可能采用协作式调度策略，在这样的系统中，只有当一个线程调用了它的 sleep()或 yield()方法后才会放弃所占用的资源——也就是必须由该线程主动放弃所占用的资源。

当发生如下情况时，线程将会进入阻塞状态。

- 线程调用 sleep()方法主动放弃所占用的处理器资源。
- 线程调用了一个阻塞式 IO 方法，在该方法返回之前，该线程被阻塞。
- 线程试图获得一个同步监视器，但该同步监视器正被其他线程所持有。关于同步监视器的知识、后面将有更深入的介绍。
- 线程在等待某个通知（notify）。
- 程序调用了线程的 suspend()方法将该线程挂起。但这个方法容易导致死锁，所以应该尽量避免使用该方法。

当前正在执行的线程被阻塞之后，其他线程就可以获得执行的机会。被阻塞的线程会在合适的时候重新进入就绪状态，注意是就绪状态而不是运行状态。也就是说，被阻塞线程的阻塞解除后，必须重新等待线程调度器再次调度它。

针对上面几种情况，当发生如下特定的情况时可以解除上面的阻塞，让该线程重新进入就绪状态。

- 调用 sleep()方法的线程经过了指定时间。
- 线程调用的阻塞式 IO 方法已经返回。

- 线程成功地获得了试图取得的同步监视器。
- 线程正在等待某个通知时，其他线程发出了一个通知。
- 处于挂起状态的线程被调用了 resume()恢复方法。

图 16.4 显示了线程状态转换图。

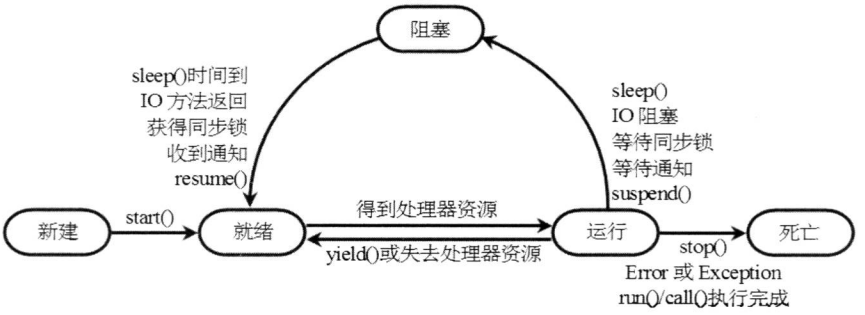


图 16.4 线程状态转换图

从图 16.4 中可以看出，线程从阻塞状态只能进入就绪状态，无法直接进入运行状态。而就绪和运行状态之间的转换通常不受程序控制，而是由系统线程调度所决定，当处于就绪状态的线程获得处理器资源时，该线程进入运行状态；当处于运行状态的线程失去处理器资源时，该线程进入就绪状态。但有一个方法例外，调用 yield()方法可以让运行状态的线程转入就绪状态。关于 yield()方法后面有更详细的介绍。

16.3.3 线程死亡

线程会以如下三种方式结束，结束后就处于死亡状态。

- run()或 call()方法执行完成，线程正常结束。
- 线程抛出一个未捕获的 Exception 或 Error。
- 直接调用该线程的 stop()方法来结束该线程——该方法容易导致死锁，通常不推荐使用。

注意：

当主线程结束时，其他线程不受任何影响，并不会随之结束。一旦子线程启动起来后，它就拥有和主线程相同的地位，它不会受主线程的影响。

为了测试某个线程是否已经死亡，可以调用线程对象的 isAlive()方法，当线程处于就绪、运行、阻塞三种状态时，该方法将返回 true；当线程处于新建、死亡两种状态时，该方法将返回 false。

注意：

不要试图对一个已经死亡的线程调用 start()方法使它重新启动，死亡就是死亡，该线程将不可再次作为线程执行。

下面程序尝试对处于死亡状态的线程再次调用 start()方法。

程序清单：codes\16\16.3\StartDead.java

```
public class StartDead extends Thread
{
    private int i ;
    // 重写 run()方法，run()方法的方法体就是线程执行体
    public void run()
    {
        for ( ; i < 100 ; i++ )
        {
            System.out.println(getName() + " " + i);
        }
    }
    public static void main(String[] args)
```

```
    {
        // 创建线程对象
        StartDead sd = new StartDead();
        for (int i = 0; i < 300;  i++)
        {
            // 调用 Thread 的 currentThread()方法获取当前线程
            System.out.println(Thread.currentThread().getName()
                + " " + i);
            if (i == 20)
            {
                // 启动线程
                sd.start();
                // 判断启动后线程的 isAlive()值，输出 true
                System.out.println(sd.isAlive());
            }
            // 当线程处于新建、死亡两种状态时，isAlive()方法返回 false
            // 当 i > 20 时，该线程肯定已经启动过了，如果 sd.isAlive()为假时
            // 那就是死亡状态了
            if (i > 20 && !sd.isAlive())
            {
                // 试图再次启动该线程
                sd.start();
            }
        }
    }
}
```

上面程序中的粗体字代码试图在线程已死亡的情况下再次调用 start()方法来启动该线程。运行上面程序，将引发 IllegalThreadStateException 异常，这表明处于死亡状态的线程无法再次运行了。

注意：

不要对处于死亡状态的线程调用 start()方法，程序只能对新建状态的线程调用 start()方法，对新建状态的线程两次调用 start()方法也是错误的。这都会引发 IllegalThreadStateException 异常。

16.4 控制线程

Java 的线程支持提供了一些便捷的工具方法，通过这些便捷的工具方法可以很好地控制线程的执行。

16.4.1 join 线程

Thread 提供了让一个线程等待另一个线程完成的方法——join()方法。当在某个程序执行流中调用其他线程的 join()方法时，调用线程将被阻塞，直到被 join()方法加入的 join 线程执行完为止。

join()方法通常由使用线程的程序调用，以将大问题划分成许多小问题，每个小问题分配一个线程。当所有的小问题都得到处理后，再调用主线程来进一步操作。

程序清单：codes\16\16.4\JoinThread.java

```
public class JoinThread extends Thread
{
    // 提供一个有参数的构造器，用于设置该线程的名字
    public JoinThread(String name)
    {
        super(name);
    }
    // 重写 run()方法，定义线程执行体
    public void run()
    {
        for (int i = 0; i < 100 ; i++ )
        {
            System.out.println(getName() + "  " + i);
        }
```

```
    }
    public static void main(String[] args)throws Exception
    {
        // 启动子线程
        new JoinThread("新线程").start();
        for (int i = 0; i < 100 ; i++ )
        {
            if (i == 20)
            {
                JoinThread jt = new JoinThread("被 Join 的线程");
                jt.start();
                // main 线程调用了 jt 线程的 join()方法，main 线程
                // 必须等 jt 执行结束才会向下执行
                jt.join();
            }
            System.out.println(Thread.currentThread().getName()
                + "  " + i);
        }
    }
}
```

上面程序中一共有 3 个线程，主方法开始时就启动了名为“新线程”的子线程，该子线程将会和 main 线程并发执行。当主线程的循环变量 i 等于 20 时，启动了名为“被 Join 的线程”的线程，该线程不会和 main 线程并发执行，main 线程必须等该线程执行结束后才可以向下执行。在名为“被 Join 的线程”的线程执行时，实际上只有 2 个子线程并发执行，而主线程处于等待状态。运行上面程序，会看到如图 16.5 所示的运行效果。

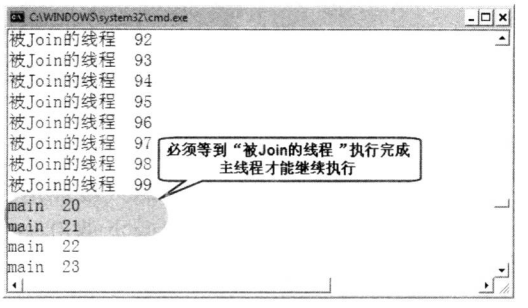


图 16.5　主线程等待 join 线程的效果

从图 16.5 中可以看出，主线程执行到 i == 20 时，程序启动并 join 了名为“被 Join 的线程”的线程，所以主线程将一直处于阻塞状态，直到名为“被 Join 的线程”的线程执行完成。

join()方法有如下三种重载形式。

- join()：等待被 join 的线程执行完成。
- join(long millis)：等待被 join 的线程的时间最长为 millis 毫秒。如果在 millis 毫秒内被 join 的线程还没有执行结束，则不再等待。
- join(long millis, int nanos)：等待被 join 的线程的时间最长为 millis 毫秒加 nanos 毫微秒。

提示：通常很少使用第三种形式，原因有两个：程序对时间的精度无须精确到毫微秒；计算机硬件、操作系统本身也无法精确到毫微秒。

16.4.2　后台线程

有一种线程，它是在后台运行的，它的任务是为其他的线程提供服务，这种线程被称为“后台线程（Daemon Thread）”，又称为“守护线程”或“精灵线程”。JVM 的垃圾回收线程就是典型的后台线程。

后台线程有个特征：如果所有的前台线程都死亡，后台线程会自动死亡。

调用 Thread 对象的 setDaemon(true)方法可将指定线程设置成后台线程。下面程序将执行线程设置成后台线程，可以看到当所有的前台线程死亡时，后台线程随之死亡。当整个虚拟机中只剩下后台线程时，程序就没有继续运行的必要了，所以虚拟机也就退出了。

程序清单：codes\16\16.4\DaemonThread.java

```
public class DaemonThread extends Thread
{
    // 定义后台线程的线程执行体与普通线程没有任何区别
    public void run()
    {
        for (int i = 0; i < 1000 ; i++ )
```

```
        {
            System.out.println(getName() + "  " + i);
        }
    }
    public static void main(String[] args)
    {
        DaemonThread t = new DaemonThread();
        // 将此线程设置成后台线程
        t.setDaemon(true);
        // 启动后台线程
        t.start();
        for (int i = 0 ; i < 10 ; i++ )
        {
            System.out.println(Thread.currentThread().getName()
                + "  " + i);
        }
        // -----程序执行到此处，前台线程（main 线程）结束------
        // 后台线程也应该随之结束
    }
}
```

上面程序中的粗体字代码先将 t 线程设置成后台线程，然后启动该线程，本来该线程应该执行到 i 等于 999 时才会结束，但运行程序时不难发现该后台线程无法运行到 999，因为当主线程也就是程序中唯一的前台线程运行结束后，JVM 会主动退出，因而后台线程也就被结束了。

Thread 类还提供了一个 isDaemon()方法，用于判断指定线程是否为后台线程。

从上面程序可以看出，主线程默认是前台线程，t 线程默认也是前台线程。并不是所有的线程默认都是前台线程，有些线程默认就是后台线程——前台线程创建的子线程默认是前台线程，后台线程创建的子线程默认是后台线程。

注意：

前台线程死亡后，JVM 会通知后台线程死亡，但从它接收指令到做出响应，需要一定时间。而且要将某个线程设置为后台线程，必须在该线程启动之前设置，也就是说，setDaemon(true)必须在 start()方法之前调用，否则会引发 IllegalThreadStateException 异常。

16.4.3 线程睡眠：sleep

如果需要让当前正在执行的线程暂停一段时间，并进入阻塞状态，则可以通过调用 Thread 类的静态 sleep()方法来实现。sleep()方法有两种重载形式。

- static void sleep(long millis)：让当前正在执行的线程暂停 millis 毫秒，并进入阻塞状态，该方法受到系统计时器和线程调度器的精度与准确度的影响。
- static void sleep(long millis, int nanos)：让当前正在执行的线程暂停 millis 毫秒加 nanos 毫微秒，并进入阻塞状态，该方法受到系统计时器和线程调度器的精度与准确度的影响。

与前面类似的是，程序很少调用第二种形式的 sleep()方法。

当当前线程调用 sleep()方法进入阻塞状态后，在其睡眠时间段内，该线程不会获得执行的机会，即使系统中没有其他可执行的线程，处于 sleep()中的线程也不会执行，因此 sleep()方法常用来暂停程序的执行。

下面程序调用 sleep()方法来暂停主线程的执行，因为该程序只有一个主线程，当主线程进入睡眠后，系统没有可执行的线程，所以可以看到程序在 sleep()方法处暂停。

程序清单：codes\16\16.4\SleepTest.java

```
public class SleepTest
{
    public static void main(String[] args)
        throws Exception
    {
        for (int i = 0; i < 10 ; i++ )
        {
```

```
            System.out.println("当前时间: " + new Date());
            // 调用 sleep()方法让当前线程暂停 1s
            Thread.sleep(1000);
        }
    }
}
```

上面程序中的粗体字代码将当前执行的线程暂停 1 秒，运行上面程序，看到程序依次输出 10 条字符串，输出 2 条字符串之间的时间间隔为 1 秒。

此外，Thread 还提供了一个与 sleep()方法有点相似的 yield()静态方法，它也可以让当前正在执行的线程暂停，但它不会阻塞该线程，它只是将该线程转入就绪状态。yield()只是让当前线程暂停一下，让系统的线程调度器重新调度一次，完全可能的情况是：当某个线程调用了 yield()方法暂停之后，线程调度器又将其调度出来重新执行。

实际上，当某个线程调用了 yield()方法暂停之后，只有优先级与当前线程相同，或者优先级比当前线程更高的处于就绪状态的线程才会获得执行的机会。

关于 sleep()方法和 yield()方法的区别如下。

- sleep()方法暂停当前线程后，会给其他线程执行机会，不会理会其他线程的优先级；但 yield()方法只会给优先级相同，或优先级更高的线程执行机会。
- sleep()方法会将线程转入阻塞状态，直到经过阻塞时间才会转入就绪状态；而 yield()不会将线程转入阻塞状态，它只是强制当前线程进入就绪状态。因此完全有可能某个线程被 yield()方法暂停之后，立即再次获得处理器资源被执行。
- sleep()方法声明抛出了 InterruptedException 异常，所以调用 sleep()方法时要么捕捉该异常，要么显式声明抛出该异常；而 yield()方法则没有声明抛出任何异常。
- sleep()方法比 yield()方法有更好的可移植性，通常不建议使用 yield()方法来控制并发线程的执行。

16.4.4 改变线程优先级

每个线程执行时都具有一定的优先级，优先级高的线程获得较多的执行机会，而优先级低的线程则获得较少的执行机会。

每个线程默认的优先级都与创建它的父线程的优先级相同，在默认情况下，main 线程具有普通优先级，由 main 线程创建的子线程也具有普通优先级。

Thread 类提供了 setPriority(int newPriority)、getPriority()方法来设置和返回指定线程的优先级，其中 setPriority()方法的参数可以是一个整数，范围是 1~10 之间，也可以使用 Thread 类的如下三个静态常量。

- MAX_PRIORITY：其值是 10。
- MIN_PRIORITY：其值是 1。
- NORM_PRIORITY：其值是 5。

下面程序使用了 setPriority()方法来改变主线程的优先级，并使用该方法改变了两个线程的优先级，从而可以看到高优先级的线程将会获得更多的执行机会。

程序清单：codes\16\16.4\PriorityTest.java

```
public class PriorityTest extends Thread
{
    // 定义一个有参数的构造器，用于创建线程时指定 name
    public PriorityTest(String name)
    {
        super(name);
    }
    public void run()
    {
        for (int i = 0 ; i < 50 ; i++ )
        {
            System.out.println(getName() +  ",其优先级是: "
                + getPriority() + ",循环变量的值为:" + i);
```

```
        }
    }
    public static void main(String[] args)
    {
        // 改变主线程的优先级
        Thread.currentThread().setPriority(6);
        for (int i = 0 ; i < 30 ; i++ )
        {
            if (i == 10)
            {
                PriorityTest low  = new PriorityTest("低级");
                low.start();
                System.out.println("创建之初的优先级:"
                    + low.getPriority());
                // 设置该线程为最低优先级
                low.setPriority(Thread.MIN_PRIORITY);
            }
            if (i == 20)
            {
                PriorityTest high = new PriorityTest("高级");
                high.start();
                System.out.println("创建之初的优先级:"
                    + high.getPriority());
                // 设置该线程为最高优先级
                high.setPriority(Thread.MAX_PRIORITY);
            }
        }
    }
}
```

上面程序中的第一行粗体字代码改变了主线程的优先级为6，这样由main线程所创建的子线程的优先级默认都是6，所以程序直接输出low、high两个线程的优先级时应该看到6。接着程序将low线程的优先级设为Priority.MIN_PRIORITY，将high线程的优先级设置为Priority.MAX_PRIORITY。

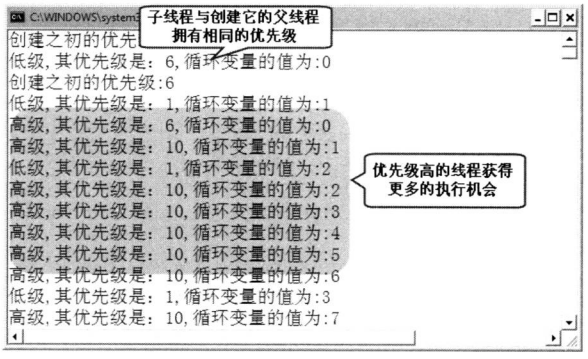


图16.6 改变线程优先级的效果

运行上面程序，会看到如图16.6所示的效果。

值得指出的是，虽然Java提供了10个优先级级别，但这些优先级级别需要操作系统的支持。遗憾的是，不同操作系统上的优先级并不相同，而且也不能很好地和Java的10个优先级对应，例如Windows 2000仅提供了7个优先级。因此应该尽量避免直接为线程指定优先级，而应该使用MAX_PRIORITY、MIN_PRIORITY和NORM_PRIORITY三个静态常量来设置优先级，这样才可以保证程序具有最好的可移植性。

16.5 线程同步

多线程编程是有趣的事情，它很容易突然出现“错误情况”，这是由系统的线程调度具有一定的随机性造成的，不过即使程序偶然出现问题，那也是由于编程不当引起的。当使用多个线程来访问同一个数据时，很容易“偶然”出现线程安全问题。

16.5.1 线程安全问题

关于线程安全问题，有一个经典的问题——银行取钱的问题。银行取钱的基本流程基本上可以分为如下几个步骤。

1. 用户输入账户、密码，系统判断用户的账户、密码是否匹配。
2. 用户输入取款金额。

③ 系统判断账户余额是否大于取款金额。

④ 如果余额大于取款金额，则取款成功；如果余额小于取款金额，则取款失败。

乍一看上去，这个流程确实就是日常生活中的取款流程，这个流程没有任何问题。但一旦将这个流程放在多线程并发的场景下，就有可能出现问题。注意此处说的是有可能，并不是说一定。也许你的程序运行了一百万次都没有出现问题，但没有出现问题并不等于没有问题！

按上面的流程去编写取款程序，并使用两个线程来模拟取钱操作，模拟两个人使用同一个账户并发取钱的问题。此处忽略检查账户和密码的操作，仅仅模拟后面三步操作。下面先定义一个账户类，该账户类封装了账户编号和余额两个实例变量。

程序清单：codes\16\16.5\Account.java

```
public class Account
{
    // 封装账户编号、账户余额的两个成员变量
    private String accountNo;
    private double balance;
    public Account(){}
    // 构造器
    public Account(String accountNo , double balance)
    {
        this.accountNo = accountNo;
        this.balance = balance;
    }
    // 此处省略了 accountNo 与 balance 的 setter 和 getter 方法
    ...
    // 下面两个方法根据 accountNo 来重写 hashCode()和 equals()方法
    public int hashCode()
    {
        return accountNo.hashCode();
    }
    public boolean equals(Object obj)
    {
        if(this == obj)
            return true;
        if (obj !=null
            && obj.getClass() == Account.class)
        {
            Account target = (Account)obj;
            return target.getAccountNo().equals(accountNo);
        }
        return false;
    }
}
```

接下来提供一个取钱的线程类，该线程类根据执行账户、取钱数量进行取钱操作，取钱的逻辑是当其余额不足时无法提取现金，当余额足够时系统吐出钞票，余额减少。

程序清单：codes\16\16.5\DrawThread.java

```
public class DrawThread extends Thread
{
    // 模拟用户账户
    private Account account;
    // 当前取钱线程所希望取的钱数
    private double drawAmount;
    public DrawThread(String name , Account account
        , double drawAmount)
    {
        super(name);
        this.account = account;
        this.drawAmount = drawAmount;
    }
    // 当多个线程修改同一个共享数据时，将涉及数据安全问题
    public void run()
    {
        // 账户余额大于取钱数目
```

```
        if (account.getBalance() >= drawAmount)
        {
            // 吐出钞票
            System.out.println(getName()
                + "取钱成功！吐出钞票:" + drawAmount);
            /*
            try
            {
                Thread.sleep(1);
            }
            catch (InterruptedException ex)
            {
                ex.printStackTrace();
            }
            */
            // 修改余额
            account.setBalance(account.getBalance() - drawAmount);
            System.out.println("\t 余额为: " + account.getBalance());
        }
        else
        {
            System.out.println(getName() + "取钱失败！余额不足！");
        }
    }
}
```

读者先不要管程序中那段被注释掉的粗体字代码，上面程序是一个非常简单的取钱逻辑，这个取钱逻辑与实际的取钱操作也很相似。程序的主程序非常简单，仅仅是创建一个账户，并启动两个线程从该账户中取钱。程序如下。

程序清单：codes\16\16.5\DrawTest.java

```
public class DrawTest
{
    public static void main(String[] args)
    {
        // 创建一个账户
        Account acct = new Account("1234567" , 1000);
        // 模拟两个线程对同一个账户取钱
        new DrawThread("甲" , acct , 800).start();
        new DrawThread("乙" , acct , 800).start();
    }
}
```

多次运行上面程序，很有可能都会看到如图 16.7 所示的错误结果。

如图 16.7 所示的运行结果并不是银行所期望的结果（不过有可能看到运行正确的效果），这正是多线程编程突然出现的“偶然”错误——因为线程调度的不确定性。假设系统线程调度器在粗体字代码处暂停，让另一个线程执行——为了强制暂停，只要取消上面程序中粗体字代码的注释即可。取消注释后再次编译 DrawThread.java，并再次运行 DrawTest 类，将总可以看到如图 16.7 所示的错误结果。

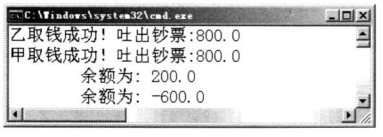


图 16.7 线程同步的问题

问题出现了：账户余额只有 1000 时取出了 1600，而且账户余额出现了负值，这不是银行希望的结果。虽然上面程序是人为地使用 Thread.sleep(1)来强制线程调度切换，但这种切换也是完全可能发生的——100000 次操作只要有 1 次出现了错误，那就是编程错误引起的。

16.5.2 同步代码块

之所以出现如图 16.7 所示的结果，是因为 run()方法的方法体不具有同步安全性——程序中有两个并发线程在修改 Account 对象；而且系统恰好在粗体字代码处执行线程切换，切换给另一个修改 Account 对象的线程，所以就出现了问题。

提示：

就像前面介绍的文件并发访问，当有两个进程并发修改同一个文件时就有可能造成异常。

为了解决这个问题，Java 的多线程支持引入了同步监视器来解决这个问题，使用同步监视器的通用方法就是同步代码块。同步代码块的语法格式如下：

```
synchronized(obj)
{
    ...
    // 此处的代码就是同步代码块
}
```

上面语法格式中 synchronized 后括号里的 obj 就是同步监视器，上面代码的含义是：线程开始执行同步代码块之前，必须先获得对同步监视器的锁定。

注意：

任何时刻只能有一个线程可以获得对同步监视器的锁定，当同步代码块执行完成后，该线程会释放对该同步监视器的锁定。

虽然 Java 程序允许使用任何对象作为同步监视器，但想一下同步监视器的目的：阻止两个线程对同一个共享资源进行并发访问，因此通常推荐使用可能被并发访问的共享资源充当同步监视器。对于上面的取钱模拟程序，应该考虑使用账户（account）作为同步监视器，把程序修改成如下形式。

程序清单：codes\16\16.5\synchronizedBlock\DrawThread.java

```
public class DrawThread extends Thread
{
    // 模拟用户账户
    private Account account;
    // 当前取钱线程所希望取的钱数
    private double drawAmount;
    public DrawThread(String name , Account account
        , double drawAmount)
    {
        super(name);
        this.account = account;
        this.drawAmount = drawAmount;
    }
    // 当多个线程修改同一个共享数据时，将涉及数据安全问题
    public void run()
    {
        // 使用 account 作为同步监视器，任何线程进入下面同步代码块之前
        // 必须先获得对 account 账户的锁定——其他线程无法获得锁，也就无法修改它
        // 这种做法符合："加锁 → 修改 → 释放锁" 的逻辑
        synchronized (account)
        {
            // 账户余额大于取钱数目
            if (account.getBalance() >= drawAmount)
            {
                // 吐出钞票
                System.out.println(getName()
                    + "取钱成功！吐出钞票:" + drawAmount);
                try
                {
                    Thread.sleep(1);
                }
                catch (InterruptedException ex)
                {
                    ex.printStackTrace();
                }
                // 修改余额
                account.setBalance(account.getBalance() - drawAmount);
                System.out.println("\t 余额为: " + account.getBalance());
            }
```

```
            else
            {
                System.out.println(getName() + "取钱失败！余额不足！");
            }
        }
        // 同步代码块结束，该线程释放同步锁
    }
}
```

上面程序使用 synchronized 将 run()方法里的方法体修改成同步代码块，该同步代码块的同步监视器是 account 对象，这样的做法符合“加锁→修改→释放锁”的逻辑，任何线程在修改指定资源之前，首先对该资源加锁，在加锁期间其他线程无法修改该资源，当该线程修改完成后，该线程释放对该资源的锁定。通过这种方式就可以保证并发线程在任一时刻只有一个线程可以进入修改共享资源的代码区（也被称为临界区），所以同一时刻最多只有一个线程处于临界区内，从而保证了线程的安全性。

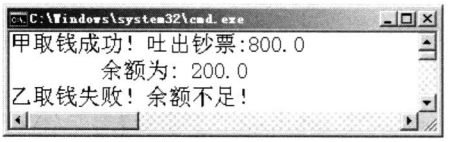


图 16.8 使用线程同步来保证线程安全

将 DrawThread 修改为上面所示的情形之后，多次运行该程序，总可以看到如图 16.8 所示的正确结果。

16.5.3 同步方法

与同步代码块对应，Java 的多线程安全支持还提供了同步方法，同步方法就是使用 synchronized 关键字来修饰某个方法，则该方法称为同步方法。对于 synchronized 修饰的实例方法（非 static 方法）而言，无须显式指定同步监视器，同步方法的同步监视器是 this，也就是调用该方法的对象。

通过使用同步方法可以非常方便地实现线程安全的类，线程安全的类具有如下特征。

- 该类的对象可以被多个线程安全地访问。
- 每个线程调用该对象的任意方法之后都将得到正确结果。
- 每个线程调用该对象的任意方法之后，该对象状态依然保持合理状态。

前面介绍了可变类和不可变类，其中不可变类总是线程安全的，因为它的对象状态不可改变；但可变对象需要额外的方法来保证其线程安全。例如上面的 Account 就是一个可变类，它的 accountNo 和 balance 两个成员变量都可以被改变，当两个线程同时修改 Account 对象的 balance 成员变量的值时，程序就出现了异常。下面将 Account 类对 balance 的访问设置成线程安全的，那么只要把修改 balance 的方法变成同步方法即可。程序如下所示。

程序清单：codes\16\16.5\synchronizedMethod\Account.java

```
public class Account
{
    // 封装账户编号、账户余额的两个成员变量
    private String accountNo;
    private double balance;
    public Account(){}
    // 构造器
    public Account(String accountNo , double balance)
    {
        this.accountNo = accountNo;
        this.balance = balance;
    }
    // 省略 accountNo 的 setter 和 getter 方法
    ...
    // 因为账户余额不允许随便修改，所以只为 balance 提供 getter 方法
    public double getBalance()
    {
        return this.balance;
    }
    // 提供一个线程安全的 draw()方法来完成取钱操作
    public synchronized void draw(double drawAmount)
    {
        // 账户余额大于取钱数目
        if (balance >= drawAmount)
        {
```

```
            // 吐出钞票
            System.out.println(Thread.currentThread().getName()
                + "取钱成功！吐出钞票:" + drawAmount);
            try
            {
                Thread.sleep(1);
            }
            catch (InterruptedException ex)
            {
                ex.printStackTrace();
            }
            // 修改余额
            balance -= drawAmount;
            System.out.println("\t 余额为: " + balance);
        }
        else
        {
            System.out.println(Thread.currentThread().getName()
                + "取钱失败！余额不足！");
        }
    }
    // 省略 hashCode()和 equals()方法
    ...
}
```

上面程序中增加了一个代表取钱的 draw()方法，并使用了 synchronized 关键字修饰该方法，把该方法变成同步方法，该同步方法的同步监视器是 this，因此对于同一个 Account 账户而言，任意时刻只能有一个线程获得对 Account 对象的锁定，然后进入 draw ()方法执行取钱操作——这样也可以保证多个线程并发取钱的线程安全。

因为 Account 类中已经提供了 draw()方法，而且取消了 setBalance()方法，DrawThread 线程类需要改写，该线程类的 run()方法只要调用 Account 对象的 draw()方法即可执行取钱操作。run()方法代码片段如下。

注意：

synchronized 关键字可以修饰方法，可以修饰代码块，但不能修饰构造器、成员变量等。

程序清单：codes\16\16.5\synchronizedMethod\DrawThread.java

```
public void run()
{
    // 直接调用 account 对象的 draw()方法来执行取钱操作
    // 同步方法的同步监视器是 this，this 代表调用 draw()方法的对象
    // 也就是说，线程进入 draw()方法之前，必须先对 account 对象加锁
    account.draw(drawAmount);
}
```

上面的 DrawThread 类无须自己实现取钱操作，而是直接调用 account 的 draw()方法来执行取钱操作。由于已经使用 synchronized 关键字修饰了 draw()方法，同步方法的同步监视器是 this，而 this 总代表调用该方法的对象——在上面示例中，调用 draw()方法的对象是 account，因此多个线程并发修改同一份 account 之前，必须先对 account 对象加锁。这也符合了“加锁 → 修改 → 释放锁”的逻辑。

提示：

在 Account 里定义 draw()方法，而不是直接在 run()方法中实现取钱逻辑，这种做法更符合面向对象规则。在面向对象里有一种流行的设计方式：Domain Driven Design（领域驱动设计，DDD），这种方式认为每个类都应该是完备的领域对象，例如 Account 代表用户账户，应该提供用户账户的相关方法；通过 draw()方法来执行取钱操作（实际上还应该提供 transfer()等方法来完成转账等操作），而不是直接将 setBalance()方法暴露出来任人操作，这样才可以更好地保证 Account 对象的完整性和一致性。

可变类的线程安全是以降低程序的运行效率作为代价的，为了减少线程安全所带来的负面影响，程

序可以采用如下策略。

- 不要对线程安全类的所有方法都进行同步，只对那些会改变竞争资源（竞争资源也就是共享资源）的方法进行同步。例如上面 Account 类中的 accountNo 实例变量就无须同步，所以程序只对 draw()方法进行了同步控制。
- 如果可变类有两种运行环境：单线程环境和多线程环境，则应该为该可变类提供两种版本，即线程不安全版本和线程安全版本。在单线程环境中使用线程不安全版本以保证性能，在多线程环境中使用线程安全版本。

提示：

JDK 所提供的 StringBuilder、StringBuffer 就是为了照顾单线程环境和多线程环境所提供的类，在单线程环境下应该使用 StringBuilder 来保证较好的性能；当需要保证多线程安全时，就应该使用 StringBuffer。

16.5.4 释放同步监视器的锁定

任何线程进入同步代码块、同步方法之前，必须先获得对同步监视器的锁定，那么何时会释放对同步监视器的锁定呢？程序无法显式释放对同步监视器的锁定，线程会在如下几种情况下释放对同步监视器的锁定。

- 当前线程的同步方法、同步代码块执行结束，当前线程即释放同步监视器。
- 当前线程在同步代码块、同步方法中遇到 break、return 终止了该代码块、该方法的继续执行，当前线程将会释放同步监视器。
- 当前线程在同步代码块、同步方法中出现了未处理的 Error 或 Exception，导致了该代码块、该方法异常结束时，当前线程将会释放同步监视器。
- 当前线程执行同步代码块或同步方法时，程序执行了同步监视器对象的 wait()方法，则当前线程暂停，并释放同步监视器。

在如下所示的情况下，线程不会释放同步监视器。

- 线程执行同步代码块或同步方法时，程序调用 Thread.sleep()、Thread.yield()方法来暂停当前线程的执行，当前线程不会释放同步监视器。
- 线程执行同步代码块时，其他线程调用了该线程的 suspend()方法将该线程挂起，该线程不会释放同步监视器。当然，程序应该尽量避免使用 suspend()和 resume()方法来控制线程。

16.5.5 同步锁（Lock）

从 Java 5 开始，Java 提供了一种功能更强大的线程同步机制——通过显式定义同步锁对象来实现同步，在这种机制下，同步锁由 Lock 对象充当。

Lock 提供了比 synchronized 方法和 synchronized 代码块更广泛的锁定操作，Lock 允许实现更灵活的结构，可以具有差别很大的属性，并且支持多个相关的 Condition 对象。

Lock 是控制多个线程对共享资源进行访问的工具。通常，锁提供了对共享资源的独占访问，每次只能有一个线程对 Lock 对象加锁，线程开始访问共享资源之前应先获得 Lock 对象。

某些锁可能允许对共享资源并发访问，如 ReadWriteLock（读写锁），Lock、ReadWriteLock 是 Java 5 提供的两个根接口，并为 Lock 提供了 ReentrantLock（可重入锁）实现类，为 ReadWriteLock 提供了 ReentrantReadWriteLock 实现类。

Java 8 新增了新型的 StampedLock 类，在大多数场景中它可以替代传统的 ReentrantReadWriteLock。ReentrantReadWriteLock 为读写操作提供了三种锁模式：Writing、ReadingOptimistic、Reading。

在实现线程安全的控制中，比较常用的是 ReentrantLock（可重入锁）。使用该 Lock 对象可以显式地加锁、释放锁，通常使用 ReentrantLock 的代码格式如下：

```
class X
{
    // 定义锁对象
    private final ReentrantLock lock = new ReentrantLock();
    // ...
```

```
    // 定义需要保证线程安全的方法
    public void m()
    {
        // 加锁
        lock.lock();
        try
        {
            // 需要保证线程安全的代码
            // ... method body
        }
        // 使用 finally 块来保证释放锁
        finally
        {
            lock.unlock();
        }
    }
}
```

使用 ReentrantLock 对象来进行同步，加锁和释放锁出现在不同的作用范围内时，通常建议使用 finally 块来确保在必要时释放锁。通过使用 ReentrantLock 对象，可以把 Account 类改为如下形式，它依然是线程安全的。

程序清单：codes\16\16.5\Lock\Account.java

```
public class Account
{
    // 定义锁对象
    private final ReentrantLock lock = new ReentrantLock();
    // 封装账户编号、账户余额的两个成员变量
    private String accountNo;
    private double balance;
    public Account(){}
    // 构造器
    public Account(String accountNo , double balance)
    {
        this.accountNo = accountNo;
        this.balance = balance;
    }
    // 省略 accountNo 的 setter 和 getter 方法
    ...
    // 因为账户余额不允许随便修改，所以只为 balance 提供 getter 方法
    public double getBalance()
    {
        return this.balance;
    }
    // 提供一个线程安全的 draw()方法来完成取钱操作
    public void draw(double drawAmount)
    {
        // 加锁
        lock.lock();
        try
        {
            // 账户余额大于取钱数目
            if (balance >= drawAmount)
            {
                // 吐出钞票
                System.out.println(Thread.currentThread().getName()
                    + "取钱成功！吐出钞票:" + drawAmount);
                try
                {
                    Thread.sleep(1);
                }
                catch (InterruptedException ex)
                {
                    ex.printStackTrace();
                }
                // 修改余额
                balance -= drawAmount;
                System.out.println("\t 余额为: " + balance);
```

```
            }
            else
            {
                System.out.println(Thread.currentThread().getName()
                    + "取钱失败！余额不足！");
            }
        }
        finally
        {
            // 修改完成，释放锁
            lock.unlock();
        }
    }
    // 省略 hashCode()和 equals()方法
    ...
}
```

上面程序中的第一行粗体字代码定义了一个 ReentrantLock 对象，程序中实现 draw()方法时，进入方法开始执行后立即请求对 ReentrantLock 对象进行加锁，当执行完 draw()方法的取钱逻辑之后，程序使用 finally 块来确保释放锁。

提示：

使用 Lock 与使用同步方法有点相似，只是使用 Lock 时显式使用 Lock 对象作为同步锁，而使用同步方法时系统隐式使用当前对象作为同步监视器，同样都符合“加锁→修改→释放锁”的操作模式，而且使用 Lock 对象时每个 Lock 对象对应一个 Account 对象，一样可以保证对于同一个 Account 对象，同一时刻只能有一个线程能进入临界区。

同步方法或同步代码块使用与竞争资源相关的、隐式的同步监视器，并且强制要求加锁和释放锁要出现在一个块结构中，而且当获取了多个锁时，它们必须以相反的顺序释放，且必须在与所有锁被获取时相同的范围内释放所有锁。

虽然同步方法和同步代码块的范围机制使得多线程安全编程非常方便，而且还可以避免很多涉及锁的常见编程错误，但有时也需要以更为灵活的方式使用锁。Lock 提供了同步方法和同步代码块所没有的其他功能，包括用于非块结构的 tryLock()方法，以及试图获取可中断锁的 lockInterruptibly()方法，还有获取超时失效锁的 tryLock(long, TimeUnit)方法。

ReentrantLock 锁具有可重入性，也就是说，一个线程可以对已被加锁的 ReentrantLock 锁再次加锁，ReentrantLock 对象会维持一个计数器来追踪 lock()方法的嵌套调用，线程在每次调用 lock()加锁后，必须显式调用 unlock()来释放锁，所以一段被锁保护的代码可以调用另一个被相同锁保护的方法。

16.5.6 死锁

当两个线程相互等待对方释放同步监视器时就会发生死锁，Java 虚拟机没有监测，也没有采取措施来处理死锁情况，所以多线程编程时应该采取措施避免死锁出现。一旦出现死锁，整个程序既不会发生任何异常，也不会给出任何提示，只是所有线程处于阻塞状态，无法继续。

死锁是很容易发生的，尤其在系统中出现多个同步监视器的情况下，如下程序将会出现死锁。

程序清单：codes\16\16.5\DeadLock.java

```
class A
{
    public synchronized void foo( B b )
    {
        System.out.println("当前线程名: " + Thread.currentThread().getName()
            + " 进入了 A 实例的 foo()方法" );     // ①
        try
        {
            Thread.sleep(200);
        }
        catch (InterruptedException ex)
        {
            ex.printStackTrace();
        }
```

```
        System.out.println("当前线程名: " + Thread.currentThread().getName()
            + " 企图调用 B 实例的 last()方法");    // ③
        b.last();
    }
    public synchronized void last()
    {
        System.out.println("进入了 A 类的 last()方法内部");
    }
}
class B
{
    public synchronized void bar( A a )
    {
        System.out.println("当前线程名: " + Thread.currentThread().getName()
            + " 进入了 B 实例的 bar()方法" );   // ②
        try
        {
            Thread.sleep(200);
        }
        catch (InterruptedException ex)
        {
            ex.printStackTrace();
        }
        System.out.println("当前线程名: " + Thread.currentThread().getName()
            + " 企图调用 A 实例的 last()方法");  // ④
        a.last();
    }
    public synchronized void last()
    {
        System.out.println("进入了 B 类的 last()方法内部");
    }
}
public class DeadLock implements Runnable
{
    A a = new A();
    B b = new B();
    public void init()
    {
        Thread.currentThread().setName("主线程");
        // 调用 a 对象的 foo()方法
        a.foo(b);
        System.out.println("进入了主线程之后");
    }
    public void run()
    {
        Thread.currentThread().setName("副线程");
        // 调用 b 对象的 bar()方法
        b.bar(a);
        System.out.println("进入了副线程之后");
    }
    public static void main(String[] args)
    {
        DeadLock dl = new DeadLock();
        // 以 dl 为 target 启动新线程
        new Thread(dl).start();
        // 调用 init()方法
        dl.init();
    }
}
```

运行上面程序，将会看到如图 16.9 所示的效果。

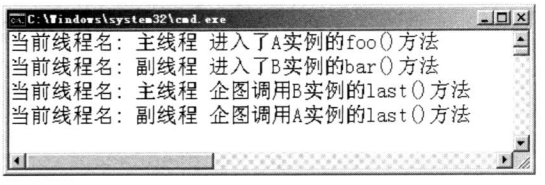


图 16.9　死锁效果

从图 16.9 中可以看出，程序既无法向下执行，也不会抛出任何异常，就一直“僵持”着。究其原因，是因为：上面程序中 A 对象和 B 对象的方法都是同步方法，也就是 A 对象和 B 对象都是同步锁。程序中两个线程执行，副线程的线程执行体是 DeadLock 类的 run()方法，主线程

的线程执行体是 DeadLock 的 main()方法（主线程调用了 init()方法）。其中 run()方法中让 B 对象调用 bar()方法，而 init()方法让 A 对象调用 foo()方法。图 16.9 显示 init()方法先执行，调用了 A 对象的 foo()方法，进入 foo()方法之前，该线程对 A 对象加锁——当程序执行到①号代码时，主线程暂停 200ms；CPU 切换到执行另一个线程，让 B 对象执行 bar()方法，所以看到副线程开始执行 B 实例的 bar()方法，进入 bar()方法之前，该线程对 B 对象加锁——当程序执行到②号代码时，副线程也暂停 200ms；接下来主线程会先醒过来，继续向下执行，直到③号代码处希望调用 B 对象的 last()方法——执行该方法之前必须先对 B 对象加锁，但此时副线程正保持着 B 对象的锁，所以主线程阻塞；接下来副线程应该也醒过来了，继续向下执行，直到④号代码处希望调用 A 对象的 last()方法——执行该方法之前必须先对 A 对象加锁，但此时主线程没有释放对 A 对象的锁——至此，就出现了主线程保持着 A 对象的锁，等待对 B 对象加锁，而副线程保持着 B 对象的锁，等待对 A 对象加锁，两个线程互相等待对方先释放，所以就出现了死锁。

注意：

由于 Thread 类的 suspend()方法也很容易导致死锁，所以 Java 不再推荐使用该方法来暂停线程的执行。

16.6 线程通信

当线程在系统内运行时，线程的调度具有一定的透明性，程序通常无法准确控制线程的轮换执行，但 Java 也提供了一些机制来保证线程协调运行。

16.6.1 传统的线程通信

假设现在系统中有两个线程，这两个线程分别代表存款者和取钱者——现在假设系统有一种特殊的要求，系统要求存款者和取钱者不断地重复存款、取钱的动作，而且要求每当存款者将钱存入指定账户后，取钱者就立即取出该笔钱。不允许存款者连续两次存钱，也不允许取钱者连续两次取钱。

为了实现这种功能，可以借助于 Object 类提供的 wait()、notify()和 notifyAll()三个方法，这三个方法并不属于 Thread 类，而是属于 Object 类。但这三个方法必须由同步监视器对象来调用，这可分成以下两种情况。

- 对于使用 synchronized 修饰的同步方法，因为该类的默认实例（this）就是同步监视器，所以可以在同步方法中直接调用这三个方法。
- 对于使用 synchronized 修饰的同步代码块，同步监视器是 synchronized 后括号里的对象，所以必须使用该对象调用这三个方法。

关于这三个方法的解释如下。

- wait()：导致当前线程等待，直到其他线程调用该同步监视器的 notify()方法或 notifyAll()方法来唤醒该线程。该 wait()方法有三种形式——无时间参数的 wait（一直等待，直到其他线程通知）、带毫秒参数的 wait()和带毫秒、毫微秒参数的 wait()（这两种方法都是等待指定时间后自动苏醒）。调用 wait()方法的当前线程会释放对该同步监视器的锁定。
- notify()：唤醒在此同步监视器上等待的单个线程。如果所有线程都在此同步监视器上等待，则会选择唤醒其中一个线程。选择是任意性的。只有当前线程放弃对该同步监视器的锁定后（使用 wait()方法），才可以执行被唤醒的线程。
- notifyAll()：唤醒在此同步监视器上等待的所有线程。只有当前线程放弃对该同步监视器的锁定后，才可以执行被唤醒的线程。

程序中可以通过一个旗标来标识账户中是否已有存款，当旗标为 false 时，表明账户中没有存款，存款者线程可以向下执行，当存款者把钱存入账户后，将旗标设为 true，并调用 notify()或 notifyAll()方法来唤醒其他线程；当存款者线程进入线程体后，如果旗标为 true 就调用 wait()方法让该线程等待。

当旗标为 true 时，表明账户中已经存入了存款，则取钱者线程可以向下执行，当取钱者把钱从账

户中取出后，将旗标设为 false，并调用 notify()或 notifyAll()方法来唤醒其他线程；当取钱者线程进入线程体后，如果旗标为 false 就调用 wait()方法让该线程等待。

本程序为 Account 类提供 draw()和 deposit()两个方法，分别对应该账户的取钱、存款等操作，因为这两个方法可能需要并发修改 Account 类的 balance 成员变量的值，所以这两个方法都使用 synchronized 修饰成同步方法。除此之外，这两个方法还使用了 wait()、notifyAll()来控制线程的协作。

程序清单：codes\16\16.6\synchronized\Account.java

```
public class Account
{
    // 封装账户编号、账户余额的两个成员变量
    private String accountNo;
    private double balance;
    // 标识账户中是否已有存款的旗标
    private boolean flag = false;
    public Account(){}
    // 构造器
    public Account(String accountNo , double balance)
    {
        this.accountNo = accountNo;
        this.balance = balance;
    }
    // 省略 accountNo 的 setter 和 getter 方法
    ...
    // 因为账户余额不允许随便修改，所以只为 balance 提供 getter 方法
    public double getBalance()
    {
        return this.balance;
    }
    public synchronized void draw(double drawAmount)
    {
        try
        {
            // 如果 flag 为假，表明账户中还没有人存钱进去，取钱方法阻塞
            if (!flag)
            {
                wait();
            }
            else
            {
                // 执行取钱操作
                System.out.println(Thread.currentThread().getName()
                    + " 取钱:" + drawAmount);
                balance -= drawAmount;
                System.out.println("账户余额为: " + balance);
                // 将标识账户是否已有存款的旗标设为 false
                flag = false;
                // 唤醒其他线程
                notifyAll();
            }
        }
        catch (InterruptedException ex)
        {
            ex.printStackTrace();
        }
    }
    public synchronized void deposit(double depositAmount)
    {
        try
        {
            // 如果 flag 为真，表明账户中已有人存钱进去，存钱方法阻塞
            if (flag)           // ①
            {
                wait();
            }
            else
            {
                // 执行存款操作
```

```
                System.out.println(Thread.currentThread().getName()
                    + " 存款:" + depositAmount);
                balance += depositAmount;
                System.out.println("账户余额为: " + balance);
                // 将表示账户是否已有存款的旗标设为true
                flag = true;
                // 唤醒其他线程
                notifyAll();
            }
        }
        catch (InterruptedException ex)
        {
            ex.printStackTrace();
        }
    }
    // 省略hashCode()和equals()方法
    ...
}
```

上面程序中的粗体字代码使用 wait()和 notifyAll()进行了控制，对存款者线程而言，当程序进入deposit()方法后，如果 flag 为 true，则表明账户中已有存款，程序调用 wait()方法阻塞；否则程序向下执行存款操作，当存款操作执行完成后，系统将 flag 设为 true，然后调用 notifyAll()来唤醒其他被阻塞的线程——如果系统中有存款者线程，存款者线程也会被唤醒，但该存款者线程执行到①号代码处时再次进入阻塞状态，只有执行 draw()方法的取钱者线程才可以向下执行。同理，取钱者线程的运行流程也是如此。

程序中的存款者线程循环 100 次重复存款，而取钱者线程则循环 100 次重复取钱，存款者线程和取钱者线程分别调用 Account 对象的 deposit()、draw()方法来实现。

程序清单：codes\16\16.6\synchronized\DrawThread.java

```
public class DrawThread extends Thread
{
    // 模拟用户账户
    private Account account;
    // 当前取钱线程所希望取的钱数
    private double drawAmount;
    public DrawThread(String name , Account account
        , double drawAmount)
    {
        super(name);
        this.account = account;
        this.drawAmount = drawAmount;
    }
    // 重复100次执行取钱操作
    public void run()
    {
        for (int i = 0 ; i < 100 ; i++ )
        {
            account.draw(drawAmount);
        }
    }
}
```

程序清单：codes\16\16.6\synchronized\DepositThread.java

```
public class DepositThread extends Thread
{
    // 模拟用户账户
    private Account account;
    // 当前存款线程所希望存的钱数
    private double depositAmount;
    public DepositThread(String name , Account account
        , double depositAmount)
    {
        super(name);
        this.account = account;
        this.depositAmount = depositAmount;
    }
```

```
    // 重复100次执行存款操作
    public void run()
    {
        for (int i = 0 ; i < 100 ; i++ )
        {
            account.deposit(depositAmount);
        }
    }
}
```

主程序可以启动任意多个存款线程和取钱线程，可以看到所有的取钱线程必须等存款线程存钱后才可以向下执行，而存款线程也必须等取钱线程取钱后才可以向下执行。主程序代码如下。

程序清单：codes\16\16.6\synchronized\DrawTest.java

```
public class DrawTest
{
    public static void main(String[] args)
    {
        // 创建一个账户
        Account acct = new Account("1234567" , 0);
        new DrawThread("取钱者" , acct , 800).start();
        new DepositThread("存款者甲" , acct , 800).start();
        new DepositThread("存款者乙" , acct , 800).start();
        new DepositThread("存款者丙" , acct , 800).start();
    }
}
```

运行该程序，可以看到存款者线程、取钱者线程交替执行的情形，每当存款者向账户中存入800元之后，取钱者线程立即从账户中取出这笔钱。存款完成后账户余额总是800元，取钱结束后账户余额总是0元。运行该程序，会看到如图16.10所示的结果。

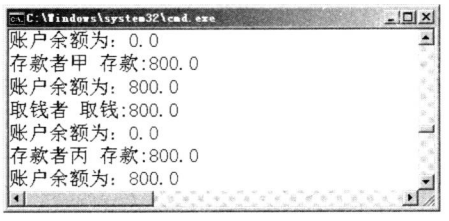


图16.10　线程协调运行的结果

从图16.10中可以看出，3个存款者线程随机地向账户中存款，只有1个取钱者线程执行取钱操作。只有当取钱者取钱后，存款者才可以存款；同理，只有等存款者存款后，取钱者线程才可以取钱。

图16.10显示程序最后被阻塞无法继续向下执行，这是因为3个存款者线程共有300次尝试存款操作，但1个取钱者线程只有100次尝试取钱操作，所以程序最后被阻塞！

注意：

如图16.10所示的阻塞并不是死锁，对于这种情况，取钱者线程已经执行结束，而存款者线程只是在等待其他线程来取钱而已，并不是等待其他线程释放同步监视器。不要把死锁和程序阻塞等同起来！

16.6.2　使用Condition控制线程通信

如果程序不使用synchronized关键字来保证同步，而是直接使用Lock对象来保证同步，则系统中不存在隐式的同步监视器，也就不能使用wait()、notify()、notifyAll()方法进行线程通信了。

当使用Lock对象来保证同步时，Java提供了一个Condition类来保持协调，使用Condition可以让那些已经得到Lock对象却无法继续执行的线程释放Lock对象，Condition对象也可以唤醒其他处于等待的线程。

Condition将同步监视器方法（wait()、notify()和notifyAll()）分解成截然不同的对象，以便通过将这些对象与Lock对象组合使用，为每个对象提供多个等待集（wait-set）。在这种情况下，Lock替代了同步方法或同步代码块，Condition替代了同步监视器的功能。

Condition实例被绑定在一个Lock对象上。要获得特定Lock实例的Condition实例，调用Lock对象的newCondition()方法即可。Condition类提供了如下三个方法。

- await()：类似于隐式同步监视器上的wait()方法，导致当前线程等待，直到其他线程调用该Condition的signal()方法或signalAll()方法来唤醒该线程。该await()方法有更多变体，如long awaitNanos(long nanosTimeout)、void awaitUninterruptibly()、awaitUntil(Date deadline)等，可以完成更丰富的等待操作。
- signal()：唤醒在此Lock对象上等待的单个线程。如果所有线程都在该Lock对象上等待，则会选择唤醒其中一个线程。选择是任意性的。只有当前线程放弃对该Lock对象的锁定后（使用await()方法），才可以执行被唤醒的线程。
- signalAll()：唤醒在此Lock对象上等待的所有线程。只有当前线程放弃对该Lock对象的锁定后，才可以执行被唤醒的线程。

下面程序中Account使用Lock对象来控制同步，并使用Condition对象来控制线程的协调运行。

程序清单：codes\16\16.6\condition\Account.java

```
public class Account
{
    // 显式定义 Lock 对象
    private final Lock lock = new ReentrantLock();
    // 获得指定 Lock 对象对应的 Condition
    private final Condition cond  = lock.newCondition();
    // 封装账户编号、账户余额的两个成员变量
    private String accountNo;
    private double balance;
    // 标识账户中是否已有存款的旗标
    private boolean flag = false;
    public Account(){}
    // 构造器
    public Account(String accountNo , double balance)
    {
        this.accountNo = accountNo;
        this.balance = balance;
    }
    // 省略 accountNo 的 setter 和 getter 方法
    ...
    // 因为账户余额不允许随便修改，所以只为 balance 提供 getter 方法
    public double getBalance()
    {
        return this.balance;
    }
    public void draw(double drawAmount)
    {
        // 加锁
        lock.lock();
        try
        {
            // 如果 flag 为假，表明账户中还没有人存钱进去，取钱方法阻塞
            if (!flag)
            {
                cond.await();
            }
            else
            {
                // 执行取钱操作
                System.out.println(Thread.currentThread().getName()
                    + " 取钱:" +  drawAmount);
                balance -= drawAmount;
                System.out.println("账户余额为: " + balance);
                // 将标识账户是否已有存款的旗标设为 false
                flag = false;
                // 唤醒其他线程
                cond.signalAll();
            }
        }
        catch (InterruptedException ex)
        {
            ex.printStackTrace();
```

```
        }
        // 使用 finally 块来释放锁
        finally
        {
            lock.unlock();
        }
    }
    public void deposit(double depositAmount)
    {
        lock.lock();
        try
        {
            // 如果 flag 为真，表明账户中已有人存钱进去，存钱方法阻塞
            if (flag)            // ①
            {
                cond.await();
            }
            else
            {
                // 执行存款操作
                System.out.println(Thread.currentThread().getName()
                    + " 存款:" +  depositAmount);
                balance += depositAmount;
                System.out.println("账户余额为: " + balance);
                // 将表示账户是否已有存款的旗标设为 true
                flag = true;
                // 唤醒其他线程
                cond.signalAll();
            }
        }
        catch (InterruptedException ex)
        {
            ex.printStackTrace();
        }
        // 使用 finally 块来释放锁
        finally
        {
            lock.unlock();
        }
    }
    // 此处省略了 hashCode()和 equals()方法
    ...
}
```

用该程序与 codes\16\16.6\synchronized 路径下的 Account.java 进行对比，不难发现这两个程序的逻辑基本相似，只是现在显式地使用 Lock 对象来充当同步监视器，则需要使用 Condition 对象来暂停、唤醒指定线程。

该示例程序的其他类与前一个示例程序的其他类完全一样，读者可以参考光盘 codes\16\16.6\condition 路径下的代码。运行该程序的效果与前一个示例程序的运行效果完全一样，此处不再赘述。

提示：

本书第 1 版还介绍了一种使用管道流进行线程通信的情形，但实际上由于两个线程属于同一个进程，它们可以非常方便地共享数据，因此很少需要使用管道流进行通信，故此处不再介绍那种烦琐的方式。

16.6.3 使用阻塞队列（BlockingQueue）控制线程通信

Java 5 提供了一个 BlockingQueue 接口，虽然 BlockingQueue 也是 Queue 的子接口，但它的主要用途并不是作为容器，而是作为线程同步的工具。BlockingQueue 具有一个特征：当生产者线程试图向 BlockingQueue 中放入元素时，如果该队列已满，则该线程被阻塞；当消费者线程试图从 BlockingQueue 中取出元素时，如果该队列已空，则该线程被阻塞。

程序的两个线程通过交替向 BlockingQueue 中放入元素、取出元素，即可很好地控制线程的通信。

BlockingQueue 提供如下两个支持阻塞的方法。

- put(E e)：尝试把 E 元素放入 BlockingQueue 中，如果该队列的元素已满，则阻塞该线程。
- take()：尝试从 BlockingQueue 的头部取出元素，如果该队列的元素已空，则阻塞该线程。

BlockingQueue 继承了 Queue 接口，当然也可使用 Queue 接口中的方法。这些方法归纳起来可分为如下三组。

- 在队列尾部插入元素。包括 add(E e)、offer(E e)和 put(E e)方法，当该队列已满时，这三个方法分别会抛出异常、返回 false、阻塞队列。
- 在队列头部删除并返回删除的元素。包括 remove()、poll()和 take()方法。当该队列已空时，这三个方法分别会抛出异常、返回 false、阻塞队列。
- 在队列头部取出但不删除元素。包括 element()和 peek()方法，当队列已空时，这两个方法分别抛出异常、返回 false。

BlockingQueue 包含的方法之间的对应关系如表 16.1 所示。

表 16.1 BlockingQueue 包含的方法之间的对应关系

	抛出异常	不同返回值	阻塞线程	指定超时时长
队尾插入元素	add(e)	offer(e)	put(e)	offer(e, time, unit)
队头删除元素	remove()	poll()	take()	poll(time ,unit)
获取、不删除元素	element()	peek()	无	无

BlockingQueue 与其实现类之间的类图如图 16.11 所示。

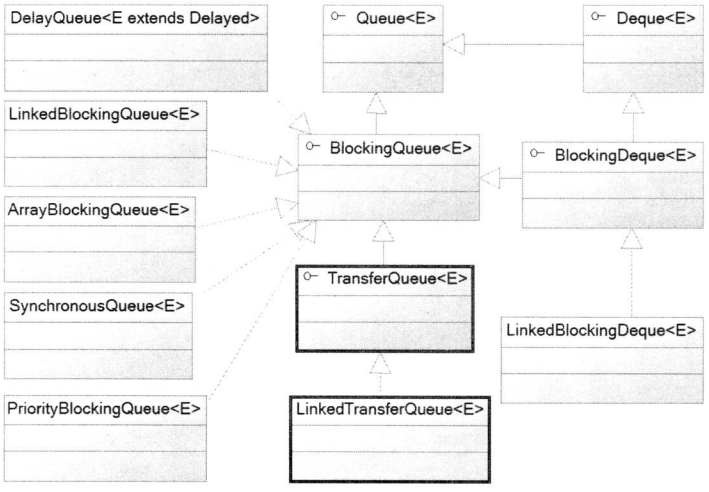


图 16.11 BlocKingQueue 与其实现类之间的类图

图 16.11 中以黑色方框框出的都是 Java 7 新增的阻塞队列。从图 16.11 可以看到，BlockingQueue 包含如下 5 个实现类。

- ArrayBlockingQueue：基于数组实现的 BlockingQueue 队列。
- LinkedBlockingQueue：基于链表实现的 BlockingQueue 队列。
- PriorityBlockingQueue：它并不是标准的阻塞队列。与前面介绍的 PriorityQueue 类似，该队列调用 remove()、poll()、take()等方法取出元素时，并不是取出队列中存在时间最长的元素，而是队列中最小的元素。PriorityBlockingQueue 判断元素的大小即可根据元素（实现 Comparable 接口）的本身大小来自然排序，也可使用 Comparator 进行定制排序。
- SynchronousQueue：同步队列。对该队列的存、取操作必须交替进行。
- DelayQueue：它是一个特殊的 BlockingQueue，底层基于 PriorityBlockingQueue 实现。不过，DelayQueue 要求集合元素都实现 Delay 接口（该接口里只有一个 long getDelay()方法），DelayQueue 根据集合元素的 getDalay()方法的返回值进行排序。

下面以 ArrayBlockingQueue 为例介绍阻塞队列的功能和用法。下面先用一个最简单的程序来测试 BlockingQueue 的 put()方法。

程序清单：codes\16\16.6\BlockingQueueTest.java

```
public class BlockingQueueTest
{
    public static void main(String[] args)
        throws Exception
    {
        // 定义一个长度为 2 的阻塞队列
        BlockingQueue<String> bq = new ArrayBlockingQueue<>(2);
        bq.put("Java"); // 与bq.add("Java")、bq.offer("Java")相同
        bq.put("Java"); // 与bq.add("Java")、bq.offer("Java")相同
        bq.put("Java"); // ① 阻塞线程
    }
}
```

上面程序先定义一个大小为 2 的 BlockingQueue，程序先向该队列中放入两个元素，此时队列还没有满，两个元素都可以放入，因此使用 put()、add()和 offer()方法效果完全一样。当程序试图放入第三个元素时，如果使用 put()方法尝试放入元素将会阻塞线程，如上面程序①号代码所示。如果使用 add()方法尝试放入元素将会引发异常；如果使用 offer()方法尝试放入元素则会返回 false，元素不会被放入。

与此类似的是，在 BlockingQueue 已空的情况下，程序使用 take()方法尝试取出元素将会阻塞线程；使用 remove()方法尝试取出元素将引发异常；使用 poll()方法尝试取出元素将返回 false，元素不会被删除。

掌握了 BlockingQueue 阻塞队列的特性之后，下面程序就可以利用 BlockingQueue 来实现线程通信了。

程序清单：codes\16\16.6\BlockingQueueTest2.java

```
class Producer extends Thread
{
    private BlockingQueue<String> bq;
    public Producer(BlockingQueue<String> bq)
    {
        this.bq = bq;
    }
    public void run()
    {
        String[] strArr = new String[]
        {
            "Java",
            "Struts",
            "Spring"
        };
        for (int i = 0 ; i < 999999999 ; i++ )
        {
            System.out.println(getName() + "生产者准备生产集合元素！");
            try
            {
                Thread.sleep(200);
                // 尝试放入元素，如果队列已满，则线程被阻塞
                bq.put(strArr[i % 3]);
            }
            catch (Exception ex){ex.printStackTrace();}
            System.out.println(getName() + "生产完成：" + bq);
        }
    }
}
class Consumer extends Thread
{
    private BlockingQueue<String> bq;
    public Consumer(BlockingQueue<String> bq)
    {
        this.bq = bq;
    }
    public void run()
    {
        while(true)
        {
            System.out.println(getName() + "消费者准备消费集合元素！");
            try
            {
```

```
                Thread.sleep(200);
                // 尝试取出元素，如果队列已空，则线程被阻塞
                bq.take();
            }
            catch (Exception ex){ex.printStackTrace();}
            System.out.println(getName() + "消费完成：" + bq);
        }
    }
}
public class BlockingQueueTest2
{
    public static void main(String[] args)
    {
        // 创建一个容量为 1 的 BlockingQueue
        BlockingQueue<String> bq = new ArrayBlockingQueue<>(1);
        // 启动 3 个生产者线程
        new Producer(bq).start();
        new Producer(bq).start();
        new Producer(bq).start();
        // 启动一个消费者线程
        new Consumer(bq).start();
    }
}
```

上面程序启动了 3 个生产者线程向 BlockingQueue 集合放入元素，启动了 1 个消费者线程从 BlockingQueue 集合取出元素。本程序的 BlockingQueue 集合容量为 1，因此 3 个生产者线程无法连续放入元素，必须等待消费者线程取出一个元素后，3 个生产者线程的其中之一才能放入一个元素。运行该程序，会看到如图 16.12 所示的结果。

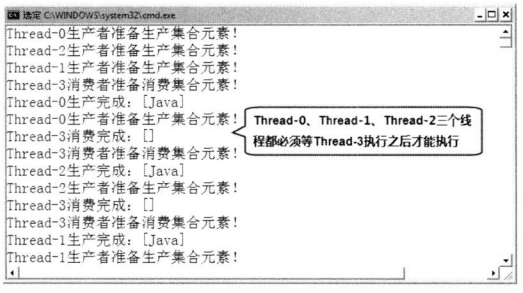


图 16.12 使用 BlockingQueue 控制线程通信

从图 16.12 可以看出，3 个生产者线程都想向 BlockingQueue 中放入元素，但只要其中一个线程向该队列中放入元素之后，其他生产者线程就必须等待，等待消费者线程取出 BlockingQueue 队列里的元素。

16.7 线程组和未处理的异常

Java 使用 ThreadGroup 来表示线程组，它可以对一批线程进行分类管理，Java 允许程序直接对线程组进行控制。对线程组的控制相当于同时控制这批线程。用户创建的所有线程都属于指定线程组，如果程序没有显式指定线程属于哪个线程组，则该线程属于默认线程组。在默认情况下，子线程和创建它的父线程处于同一个线程组内，例如 A 线程创建了 B 线程，并且没有指定 B 线程的线程组，则 B 线程属于 A 线程所在的线程组。

一旦某个线程加入了指定线程组之后，该线程将一直属于该线程组，直到该线程死亡，线程运行中途不能改变它所属的线程组。

Thread 类提供了如下几个构造器来设置新创建的线程属于哪个线程组。

- Thread(ThreadGroup group, Runnable target)：以 target 的 run()方法作为线程执行体创建新线程，属于 group 线程组。
- Thread(ThreadGroup group, Runnable target, String name)：以 target 的 run()方法作为线程执行体创建新线程，该线程属于 group 线程组，且线程名为 name。
- Thread(ThreadGroup group, String name)：创建新线程，新线程名为 name，属于 group 线程组。

因为中途不可改变线程所属的线程组，所以 Thread 类没有提供 setThreadGroup()方法来改变线程所属的线程组，但提供了一个 getThreadGroup()方法来返回该线程所属的线程组，getThreadGroup()方法的返回值是 ThreadGroup 对象，表示一个线程组。ThreadGroup 类提供了如下两个简单的构造器来创建实例。

- ThreadGroup(String name)：以指定的线程组名字来创建新的线程组。
- ThreadGroup(ThreadGroup parent, String name)：以指定的名字、指定的父线程组创建一个新线程组。

上面两个构造器在创建线程组实例时都必须为其指定一个名字，也就是说，线程组总会具有一个字符串类型的名字，该名字可通过调用 ThreadGroup 的 getName()方法来获取，但不允许改变线程组的名字。

ThreadGroup 类提供了如下几个常用的方法来操作整个线程组里的所有线程。

- int activeCount()：返回此线程组中活动线程的数目。
- interrupt()：中断此线程组中的所有线程。
- isDaemon()：判断该线程组是否是后台线程组。
- setDaemon(boolean daemon)：把该线程组设置成后台线程组。后台线程组具有一个特征——当后台线程组的最后一个线程执行结束或最后一个线程被销毁后，后台线程组将自动销毁。
- setMaxPriority(int pri)：设置线程组的最高优先级。

下面程序创建了几个线程，它们分别属于不同的线程组，程序还将一个线程组设置成后台线程组。

程序清单：codes\16\16.7\ThreadGroupTest.java

```
class MyThread extends Thread
{
    // 提供指定线程名的构造器
    public MyThread(String name)
    {
        super(name);
    }
    // 提供指定线程名、线程组的构造器
    public MyThread(ThreadGroup group , String name)
    {
        super(group, name);
    }
    public void run()
    {
        for (int i = 0; i < 20 ; i++ )
        {
            System.out.println(getName() + " 线程的 i 变量" + i);
        }
    }
}
public class ThreadGroupTest
{
    public static void main(String[] args)
    {
        // 获取主线程所在的线程组，这是所有线程默认的线程组
        ThreadGroup mainGroup = Thread.currentThread().getThreadGroup();
        System.out.println("主线程组的名字: "
            + mainGroup.getName());
        System.out.println("主线程组是否是后台线程组: "
            + mainGroup.isDaemon());
        new MyThread("主线程组的线程").start();
        ThreadGroup tg = new ThreadGroup("新线程组");
        tg.setDaemon(true);
        System.out.println("tg 线程组是否是后台线程组: "
            + tg.isDaemon());
        MyThread tt = new MyThread(tg , "tg 组的线程甲");
        tt.start();
        new MyThread(tg , "tg 组的线程乙").start();
    }
}
```

上面程序中的第一段粗体字代码用于获取主线程所属的线程组，并访问该线程组的相关属性；第二段粗体字代码创建了一个新线程组，并将该线程组设置为后台线程组。

ThreadGroup 内还定义了一个很有用的方法：void uncaughtException(Thread t, Throwable e)，该方法可以处理该线程组内的任意线程所抛出的未处理异常。

从 Java 5 开始，Java 加强了线程的异常处理，如果线程执行过程中抛出了一个未处理异常，JVM 在结束该线程之前会自动查找是否有对应的 Thread.UncaughtExceptionHandler 对象，如果找到该处理器对象，则会调用该对象的 uncaughtException(Thread t, Throwable e)方法来处理该异常。

Thread.UncaughtExceptionHandler 是 Thread 类的一个静态内部接口，该接口内只有一个方法：void uncaughtException(Thread t, Throwable e)，该方法中的 t 代表出现异常的线程，而 e 代表该线程抛出的异常。

Thread 类提供了如下两个方法来设置异常处理器。

- static setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)：为该线程类的所有线程实例设置默认的异常处理器。
- setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)：为指定的线程实例设置异常处理器。

ThreadGroup 类实现了 Thread.UncaughtExceptionHandler 接口，所以每个线程所属的线程组将会作为默认的异常处理器。当一个线程抛出未处理异常时，JVM 会首先查找该异常对应的异常处理器（setUncaughtExceptionHandler()方法设置的异常处理器），如果找到该异常处理器，则将调用该异常处理器处理该异常；否则，JVM 将会调用该线程所属的线程组对象的 uncaughtException()方法来处理该异常。线程组处理异常的默认流程如下。

① 如果该线程组有父线程组，则调用父线程组的 uncaughtException()方法来处理该异常。

② 如果该线程实例所属的线程类有默认的异常处理器（由 setDefaultUncaughtExceptionHandler()方法设置的异常处理器），那么就调用该异常处理器来处理该异常。

③ 如果该异常对象是 ThreadDeath 的对象，则不做任何处理；否则，将异常跟踪栈的信息打印到 System.err 错误输出流，并结束该线程。

下面程序为主线程设置了异常处理器，当主线程运行抛出未处理异常时，该异常处理器将会起作用。

程序清单：codes\16\16.7\ExHandler.java

```
// 定义自己的异常处理器
class MyExHandler implements Thread.UncaughtExceptionHandler
{
    // 实现 uncaughtException()方法，该方法将处理线程的未处理异常
    public void uncaughtException(Thread t, Throwable e)
    {
        System.out.println(t + " 线程出现了异常：" + e);
    }
}
public class ExHandler
{
    public static void main(String[] args)
    {
        // 设置主线程的异常处理器
        Thread.currentThread().setUncaughtExceptionHandler
            (new MyExHandler());
        int a = 5 / 0;        // ①
        System.out.println("程序正常结束！");
    }
}
```

上面程序的主方法中粗体字代码为主线程设置了异常处理器，而①号代码处将引发一个未处理异常，则该异常处理器会负责处理该异常。运行该程序，会看到如下输出：

```
Thread[main,5,main] 线程出现了异常：java.lang.ArithmeticException: / by zero
```

从上面程序的执行结果来看，虽然程序中粗体字代码指定了异常处理器对未捕获的异常进行处理，而且该异常处理器也确实起作用了，但程序依然不会正常结束。这说明异常处理器与通过 catch 捕获异

常是不同的——当使用 catch 捕获异常时，异常不会向上传播给上一级调用者；但使用异常处理器对异常进行处理之后，异常依然会传播给上一级调用者。

16.8 线程池

系统启动一个新线程的成本是比较高的，因为它涉及与操作系统交互。在这种情形下，使用线程池可以很好地提高性能，尤其是当程序中需要创建大量生存期很短暂的线程时，更应该考虑使用线程池。

与数据库连接池类似的是，线程池在系统启动时即创建大量空闲的线程，程序将一个 Runnable 对象或 Callable 对象传给线程池，线程池就会启动一个空闲的线程来执行它们的 run()或 call()方法，当 run()或 call()方法执行结束后，该线程并不会死亡，而是再次返回线程池中成为空闲状态，等待执行下一个 Runnable 对象的 run()或 call()方法。

提示：

关于池的概念，读者可以参考本书 13.9 节的介绍。

除此之外，使用线程池可以有效地控制系统中并发线程的数量，当系统中包含大量并发线程时，会导致系统性能剧烈下降，甚至导致 JVM 崩溃，而线程池的最大线程数参数可以控制系统中并发线程数不超过此数。

16.8.1 Java 8 改进的线程池

在 Java 5 以前，开发者必须手动实现自己的线程池；从 Java 5 开始，Java 内建支持线程池。Java 5 新增了一个 Executors 工厂类来产生线程池，该工厂类包含如下几个静态工厂方法来创建线程池。

- newCachedThreadPool()：创建一个具有缓存功能的线程池，系统根据需要创建线程，这些线程将会被缓存在线程池中。
- newFixedThreadPool(int nThreads)：创建一个可重用的、具有固定线程数的线程池。
- newSingleThreadExecutor()：创建一个只有单线程的线程池，它相当于调用 newFixedThread Pool()方法时传入参数为 1。
- newScheduledThreadPool(int corePoolSize)：创建具有指定线程数的线程池，它可以在指定延迟后执行线程任务。corePoolSize 指池中所保存的线程数，即使线程是空闲的也被保存在线程池内。
- newSingleThreadScheduledExecutor()：创建只有一个线程的线程池，它可以在指定延迟后执行线程任务。
- ExecutorService newWorkStealingPool(int parallelism)：创建持有足够的线程的线程池来支持给定的并行级别，该方法还会使用多个队列来减少竞争。
- ExecutorService newWorkStealingPool()：该方法是前一个方法的简化版本。如果当前机器有 4 个 CPU，则目标并行级别被设置为 4，也就是相当于为前一个方法传入 4 作为参数。

上面 7 个方法中的前三个方法返回一个 ExecutorService 对象，该对象代表一个线程池，它可以执行 Runnable 对象或 Callable 对象所代表的线程；而中间两个方法返回一个 ScheduledExecutorService 线程池，它是 ExecutorService 的子类，它可以在指定延迟后执行线程任务；最后两个方法则是 Java 8 新增的，这两个方法可充分利用多 CPU 并行的能力。这两个方法生成的 work stealing 池，都相当于后台线程池，如果所有的前台线程都死亡了，work stealing 池中的线程会自动死亡。

由于目前计算机硬件的发展日新月异，即使普通用户使用的电脑通常也都是多核 CPU，因此 Java 8 在线程支持上也增加了利用多 CPU 并行的能力，这样可以更好地发挥底层硬件的性能。

ExecutorService 代表尽快执行线程的线程池（只要线程池中有空闲线程，就立即执行线程任务），程序只要将一个 Runnable 对象或 Callable 对象（代表线程任务）提交给该线程池，该线程池就会尽快执行该任务。ExecutorService 里提供了如下三个方法。

- Future<?> submit(Runnable task)：将一个 Runnable 对象提交给指定的线程池，线程池将在有空闲线程时执行 Runnable 对象代表的任务。其中 Future 对象代表 Runnable 任务的返回值——但 run()方法没有返回值，所以 Future 对象将在 run()方法执行结束后返回 null。但可以调用 Future

的 isDone()、isCancelled()方法来获得 Runnable 对象的执行状态。

- <T> Future<T> submit(Runnable task, T result)：将一个 Runnable 对象提交给指定的线程池，线程池将在有空闲线程时执行 Runnable 对象代表的任务。其中 result 显式指定线程执行结束后的返回值，所以 Future 对象将在 run()方法执行结束后返回 result。
- <T> Future<T> submit(Callable<T> task)：将一个 Callable 对象提交给指定的线程池，线程池将在有空闲线程时执行 Callable 对象代表的任务。其中 Future 代表 Callable 对象里 call()方法的返回值。

ScheduledExecutorService 代表可在指定延迟后或周期性地执行线程任务的线程池，它提供了如下 4 个方法。

- ScheduledFuture<V> schedule(Callable<V> callable, long delay, TimeUnit unit)：指定 callable 任务将在 delay 延迟后执行。
- ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit)：指定 command 任务将在 delay 延迟后执行。
- ScheduledFuture<?> scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)：指定 command 任务将在 delay 延迟后执行，而且以设定频率重复执行。也就是说，在 initialDelay 后开始执行，依次在 initialDelay+period、initialDelay+2*period…处重复执行，依此类推。
- ScheduledFuture<?> scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit)：创建并执行一个在给定初始延迟后首次启用的定期操作，随后在每一次执行终止和下一次执行开始之间都存在给定的延迟。如果任务在任一次执行时遇到异常，就会取消后续执行；否则，只能通过程序来显式取消或终止该任务。

用完一个线程池后，应该调用该线程池的 shutdown()方法，该方法将启动线程池的关闭序列，调用 shutdown()方法后的线程池不再接收新任务，但会将以前所有已提交任务执行完成。当线程池中的所有任务都执行完成后，池中的所有线程都会死亡；另外也可以调用线程池的 shutdownNow()方法来关闭线程池，该方法试图停止所有正在执行的活动任务，暂停处理正在等待的任务，并返回等待执行的任务列表。

使用线程池来执行线程任务的步骤如下。

1. 调用 Executors 类的静态工厂方法创建一个 ExecutorService 对象，该对象代表一个线程池。
2. 创建 Runnable 实现类或 Callable 实现类的实例，作为线程执行任务。
3. 调用 ExecutorService 对象的 submit()方法来提交 Runnable 实例或 Callable 实例。
4. 当不想提交任何任务时，调用 ExecutorService 对象的 shutdown()方法来关闭线程池。

下面程序使用线程池来执行指定 Runnable 对象所代表的任务。

程序清单：codes\16\16.8\ThreadPoolTest.java

```
public class ThreadPoolTest
{
    public static void main(String[] args)
        throws Exception
    {
        // 创建一个具有固定线程数（6）的线程池
        ExecutorService pool = Executors.newFixedThreadPool(6);
        // 使用 Lambda 表达式创建 Runnable 对象
        Runnable target = () -> {
            for (int i = 0; i < 100 ; i++ )
            {
                System.out.println(Thread.currentThread().getName()
                    + "的 i 值为:" + i);
            }
        };
        // 向线程池中提交两个线程
        pool.submit(target);
        pool.submit(target);
        // 关闭线程池
        pool.shutdown();
    }
}
```

上面程序中创建 Runnable 实现类与最开始创建线程池并没有太大差别，创建了 Runnable 实现类之后程序没有直接创建线程、启动线程来执行该 Runnable 任务，而是通过线程池来执行该任务，使用线程池来执行 Runnable 任务的代码如程序中粗体字代码所示。运行上面程序，将看到两个线程交替执行的效果，如图 16.13 所示。

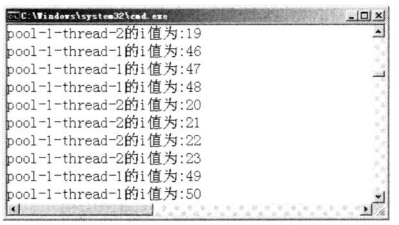


图 16.13　使用线程池并发执行两个任务

16.8.2　Java 8 增强的 ForkJoinPool

现在计算机大多已向多 CPU 方向发展，即使普通 PC，甚至小型智能设备（如手机）、多核处理器也已被广泛应用。在未来的日子里，处理器的核心数将会发展到更多。

虽然硬件上的多核 CPU 已经十分成熟，但很多应用程序并未为这种多核 CPU 做好准备，因此并不能很好地利用多核 CPU 的性能优势。

为了充分利用多 CPU、多核 CPU 的性能优势，计算机软件系统应该可以充分“挖掘”每个 CPU 的计算能力，绝不能让某个 CPU 处于“空闲”状态。为了充分利用多 CPU、多核 CPU 的优势，可以考虑把一个任务拆分成多个“小任务”，把多个“小任务”放到多个处理器核心上并行执行；当多个“小任务”执行完成之后，再将这些执行结果合并起来即可。

Java 7 提供了 ForkJoinPool 来支持将一个任务拆分成多个“小任务”并行计算，再把多个“小任务”的结果合并成总的计算结果。ForkJoinPool 是 ExecutorService 的实现类，因此是一种特殊的线程池。ForkJoinPool 提供了如下两个常用的构造器。

- ForkJoinPool(int parallelism)：创建一个包含 parallelism 个并行线程的 ForkJoinPool。
- ForkJoinPool()：以 Runtime.availableProcessors()方法的返回值作为 parallelism 参数来创建 ForkJoinPool。

Java 8 进一步扩展了 ForkJoinPool 的功能，Java 8 为 ForkJoinPool 增加了通用池功能。ForkJoinPool 类通过如下两个静态方法提供通用池功能。

- ForkJoinPool commonPool()：该方法返回一个通用池，通用池的运行状态不会受 shutdown()或 shutdownNow()方法的影响。当然，如果程序直接执行 System.exit(0);来终止虚拟机，通用池以及通用池中正在执行的任务都会被自动终止。
- int getCommonPoolParallelism()：该方法返回通用池的并行级别。

创建了 ForkJoinPool 实例之后，就可调用 ForkJoinPool 的 submit(ForkJoinTask task)或 invoke(ForkJoinTask task)方法来执行指定任务了。其中 ForkJoinTask 代表一个可以并行、合并的任务。ForkJoinTask 是一个抽象类，它还有两个抽象子类：RecursiveAction 和 RecursiveTask。其中 RecursiveTask 代表有返回值的任务，而 RecursiveAction 代表没有返回值的任务。

图 16.14 显示了 ForkJoinPool、ForkJoinTask 等类的类图。

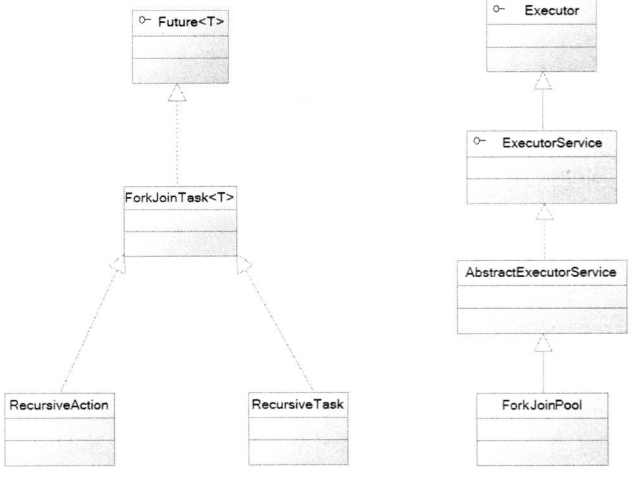


图 16.14　线程池工具类的类图

下面以执行没有返回值的“大任务”（简单地打印0~300的数值）为例，程序将一个“大任务”拆分成多个“小任务”，并将任务交给ForkJoinPool来执行。

程序清单：codes\16\16.8\ForkJoinPoolTest.java

```
// 继承RecursiveAction来实现“可分解”的任务
class PrintTask extends RecursiveAction
{
    // 每个“小任务”最多只打印50个数
    private static final int THRESHOLD = 50;
    private int start;
    private int end;
    // 打印从start到end的任务
    public PrintTask(int start, int end)
    {
        this.start = start;
        this.end = end;
    }
    @Override
    protected void compute()
    {
        // 当end与start之间的差小于THRESHOLD时，开始打印
        if(end - start < THRESHOLD)
        {
            for (int i = start ; i < end ; i++ )
            {
                System.out.println(Thread.currentThread().getName()
                    + "的i值: " + i);
            }
        }
        else
        {
            // 当end与start之间的差大于THRESHOLD，即要打印的数超过50个时
            // 将大任务分解成两个“小任务”
            int middle = (start + end) / 2;
            PrintTask left = new PrintTask(start, middle);
            PrintTask right = new PrintTask(middle, end);
            // 并行执行两个“小任务”
            left.fork();
            right.fork();
        }
    }
}
public class ForkJoinPoolTest
{
    public static void main(String[] args)
        throws Exception
    {
        ForkJoinPool pool = new ForkJoinPool();
        // 提交可分解的PrintTask任务
        pool.submit(new PrintTask(0 , 300));
        pool.awaitTermination(2, TimeUnit.SECONDS);
        // 关闭线程池
        pool.shutdown();
    }
}
```

上面程序中的粗体字代码实现了对指定打印任务的分解，分解后的任务分别调用 fork()方法开始并行执行。运行上面程序，可以看到如图16.15所示的结果。

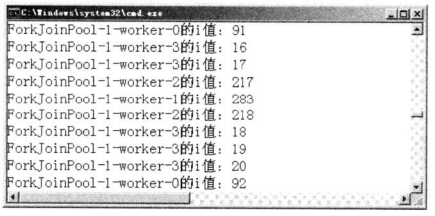


图16.15 使用ForkJoinPool的示例结果

从如图16.15所示的执行结果来看，ForkJoinPool启动了4个线程来执行这个打印任务——这是因为测试计算机的CPU是4核的。不仅如此，读者可以看到程序虽然打印了0~299这300个数字，但并不是连续打印的，这是因为程序将这个打印任务进行了分解，分解后的任务会并行执行，所以不会按顺序从0打印到299。

上面定义的任务是一个没有返回值的打印任务，如果大任务是有返回值的任务，则可以让任务继承RecursiveTask<T>，其中泛型参数T就代表了该任务的返回值类型。下面程序示范了使用Recursive Task对一个长度为100的数组的元素值进行累加。

程序清单：codes\16\16.8\Sum.java

```
// 继承RecursiveTask来实现“可分解”的任务
class CalTask extends RecursiveTask<Integer>
{
    // 每个“小任务”最多只累加20个数
    private static final int THRESHOLD = 20;
    private int arr[];
    private int start;
    private int end;
    // 累加从start到end的数组元素
    public CalTask(int[] arr , int start, int end)
    {
        this.arr = arr;
        this.start = start;
        this.end = end;
    }
    @Override
    protected Integer compute()
    {
        int sum = 0;
        // 当end与start之间的差小于THRESHOLD时，开始进行实际累加
        if(end - start < THRESHOLD)
        {
            for (int i = start ; i < end ; i++ )
            {
                sum += arr[i];
            }
            return sum;
        }
        else
        {
            // 当end与start之间的差大于THRESHOLD，即要累加的数超过20个时
            // 将大任务分解成两个“小任务”
            int middle = (start + end) /2;
            CalTask left = new CalTask(arr , start, middle);
            CalTask right = new CalTask(arr , middle, end);
            // 并行执行两个“小任务”
            left.fork();
            right.fork();
            // 把两个“小任务”累加的结果合并起来
            return left.join() + right.join();        // ①
        }
    }
}
public class Sum
{
    public static void main(String[] args)
        throws Exception
    {
        int[] arr = new int[100];
        Random rand = new Random();
        int total = 0;
        // 初始化100个数字元素
        for (int i = 0 , len = arr.length; i < len ; i++ )
        {
            int tmp = rand.nextInt(20);
            // 对数组元素赋值，并将数组元素的值添加到sum总和中
            total += (arr[i] = tmp);
        }
        System.out.println(total);
        // 创建一个通用池
        ForkJoinPool pool = ForkJoinPool.commonPool();
        // 提交可分解的CaltTask任务
```

```
        Future<Integer> future = pool.submit(new CalTask(arr , 0 , arr.length));
        System.out.println(future.get());
        // 关闭线程池
        pool.shutdown();
    }
}
```

上面程序与前一个程序基本相似，同样是将任务进行了分解，并调用分解后的任务的 fork()方法使它们并行执行。与前一个程序不同的是，现在任务是带返回值的，因此程序还在①号代码处将两个分解后的“小任务”的返回值进行了合并。

运行上面程序，将可以看到程序通过 CalTask 计算出来的总和，与初始化数组元素时统计出来的总和总是相等，这表明程序一切正常。

提示：

Java 的确是一门非常优秀的编程语言，在多 CPU、多核 CPU 时代来到时，Java 语言的多线程已经为多核 CPU 做好了准备。

16.9 线程相关类

Java 还为线程安全提供了一些工具类，如 ThreadLocal 类，它代表一个线程局部变量，通过把数据放在 ThreadLocal 中就可以让每个线程创建一个该变量的副本，从而避免并发访问的线程安全问题。除此之外，Java 5 还新增了大量的线程安全类。

16.9.1 ThreadLocal 类

早在 JDK 1.2 推出之时，Java 就为多线程编程提供了一个 ThreadLocal 类；从 Java 5.0 以后，Java 引入了泛型支持，Java 为该 ThreadLocal 类增加了泛型支持，即：ThreadLocal<T>。通过使用 ThreadLocal 类可以简化多线程编程时的并发访问，使用这个工具类可以很简捷地隔离多线程程序的竞争资源。

ThreadLocal，是 Thread Local Variable（线程局部变量）的意思，也许将它命名为 ThreadLocalVar 更加合适。线程局部变量（ThreadLocal）的功用其实非常简单，就是为每一个使用该变量的线程都提供一个变量值的副本，使每一个线程都可以独立地改变自己的副本，而不会和其他线程的副本冲突。从线程的角度看，就好像每一个线程都完全拥有该变量一样。

ThreadLocal 类的用法非常简单，它只提供了如下三个 public 方法。

- T get()：返回此线程局部变量中当前线程副本中的值。
- void remove()：删除此线程局部变量中当前线程的值。
- void set(T value)：设置此线程局部变量中当前线程副本中的值。

下面程序将向读者证明 ThreadLocal 的作用。

程序清单：codes\16\16.9\ThreadLocalTest.java

```
class Account
{
    /* 定义一个 ThreadLocal 类型的变量，该变量将是一个线程局部变量
    每个线程都会保留该变量的一个副本 */
    private ThreadLocal<String> name = new ThreadLocal<>();
    // 定义一个初始化 name 成员变量的构造器
    public Account(String str)
    {
        this.name.set(str);
        // 下面代码用于访问当前线程的 name 副本的值
        System.out.println("---" + this.name.get());
    }
    // name 的 setter 和 getter 方法
    public String getName()
    {
        return name.get();
    }
    public void setName(String str)
    {
        this.name.set(str);
    }
```

```
}
class MyTest extends Thread
{
    // 定义一个 Account 类型的成员变量
    private Account account;
    public MyTest(Account account, String name)
    {
        super(name);
        this.account = account;
    }
    public void run()
    {
        // 循环 10 次
        for (int i = 0 ; i < 10 ; i++)
        {
            // 当 i == 6 时输出将账户名替换成当前线程名
            if (i == 6)
            {
                account.setName(getName());
            }
            // 输出同一个账户的账户名和循环变量
            System.out.println(account.getName()
                + " 账户的 i 值: " + i);
        }
    }
}
public class ThreadLocalTest
{
    public static void main(String[] args)
    {
        // 启动两个线程，两个线程共享同一个 Account
        Account at = new Account("初始名");
        /*
        虽然两个线程共享同一个账户，即只有一个账户名
        但由于账户名是 ThreadLocal 类型的，所以每个线程
        都完全拥有各自的账户名副本，因此在 i == 6 之后，将看到两个
        线程访问同一个账户时出现不同的账户名
        */
        new MyTest(at , "线程甲").start();
        new MyTest(at , "线程乙").start ();
    }
}
```

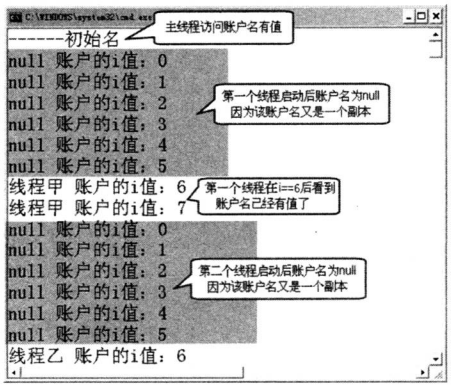


图 16.16　线程局部变量互不干扰的情形

上面 Account 类中的三行粗体字代码分别完成了创建 ThreadLocal 对象、从 ThreadLocal 中取出线程局部变量、修改线程局部变量的操作。由于程序中的账户名是一个 ThreadLocal 变量，所以虽然程序中只有一个 Account 对象，但两个子线程将会产生两个账户名（主线程也持有一个账户名的副本）。两个线程进行循环时都会在 i == 6 时将账户名改为与线程名相同，这样就可以看到两个线程拥有两个账户名的情形，如图 16.16 所示。

从上面程序可以看出，实际上账户名有三个副本，主线程一个，另外启动的两个线程各一个，它们的值互不干扰，每个线程完全拥有自己的 ThreadLocal 变量，这就是 ThreadLocal 的用途。

ThreadLocal 和其他所有的同步机制一样，都是为了解决多线程中对同一变量的访问冲突，在普通的同步机制中，是通过对象加锁来实现多个线程对同一变量的安全访问的。该变量是多个线程共享的，所以要使用这种同步机制，需要很细致地分析在什么时候对变量进行读写，什么时候需要锁定某个对象，什么时候释放该对象的锁等。在这种情况下，系统并没有将这份资源复制多份，只是采用了安全机制来控制对这份资源的访问而已。

ThreadLocal 从另一个角度来解决多线程的并发访问，ThreadLocal 将需要并发访问的资源复制多份，每个线程拥有一份资源，每个线程都拥有自己的资源副本，从而也就没有必要对该变量进行同步了。

ThreadLocal 提供了线程安全的共享对象，在编写多线程代码时，可以把不安全的整个变量封装进 ThreadLocal，或者把该对象与线程相关的状态使用 ThreadLocal 保存。

ThreadLocal 并不能替代同步机制，两者面向的问题领域不同。同步机制是为了同步多个线程对相同资源的并发访问，是多个线程之间进行通信的有效方式；而 ThreadLocal 是为了隔离多个线程的数据共享，从根本上避免多个线程之间对共享资源（变量）的竞争，也就不需要对多个线程进行同步了。

通常建议：如果多个线程之间需要共享资源，以达到线程之间的通信功能，就使用同步机制；如果仅仅需要隔离多个线程之间的共享冲突，则可以使用 ThreadLocal。

16.9.2 包装线程不安全的集合

前面介绍 Java 集合时所讲的 ArrayList、LinkedList、HashSet、TreeSet、HashMap、TreeMap 等都是线程不安全的，也就是说，当多个并发线程向这些集合中存、取元素时，就可能会破坏这些集合的数据完整性。

如果程序中有多个线程可能访问以上这些集合，就可以使用 Collections 提供的类方法把这些集合包装成线程安全的集合。Collections 提供了如下几个静态方法。

- <T> Collection<T> synchronizedCollection(Collection<T> c)：返回指定 collection 对应的线程安全的 collection。
- static <T> List<T> synchronizedList(List<T> list)：返回指定 List 对象对应的线程安全的 List 对象。
- static <K,V> Map<K,V> synchronizedMap(Map<K,V> m)：返回指定 Map 对象对应的线程安全的 Map 对象。
- static <T> Set<T> synchronizedSet(Set<T> s)：返回指定 Set 对象对应的线程安全的 Set 对象。
- static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> m)：返回指定 SortedMap 对象对应的线程安全的 SortedMap 对象。
- static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s)：返回指定 SortedSet 对象对应的线程安全的 SortedSet 对象。

例如需要在多线程中使用线程安全的 HashMap 对象，则可以采用如下代码：

```
// 使用 Collections 的 synchronizedMap 方法将一个普通的 HashMap 包装成线程安全的类
HashMap m = Collections.synchronizedMap(new HashMap());
```

注意：

如果需要把某个集合包装成线程安全的集合，则应该在创建之后立即包装，如上程序所示——当 HashMap 对象创建后立即被包装成线程安全的 HashMap 对象。

16.9.3 线程安全的集合类

实际上从 Java 5 开始，在 java.util.concurrent 包下提供了大量支持高效并发访问的集合接口和实现类，如图 16.17 所示。

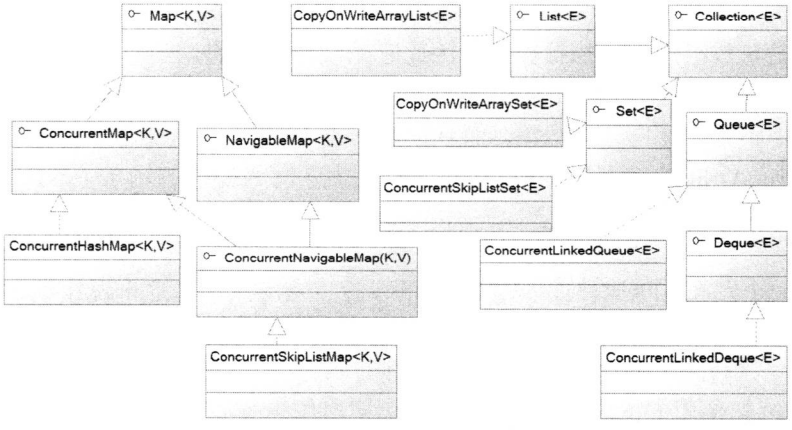


图 16.17 线程安全的集合类

从图 16.17 所示的类图可以看出，这些线程安全的集合类可分为如下两类。

- 以Concurrent开头的集合类，如ConcurrentHashMap、ConcurrentSkipListMap、ConcurrentSkip ListSet、ConcurrentLinkedQueue 和 ConcurrentLinkedDeque。
- 以 CopyOnWrite 开头的集合类，如 CopyOnWriteArrayList、CopyOnWriteArraySet。

其中以 Concurrent 开头的集合类代表了支持并发访问的集合，它们可以支持多个线程并发写入访问，这些写入线程的所有操作都是线程安全的，但读取操作不必锁定。以 Concurrent 开头的集合类采用了更复杂的算法来保证永远不会锁住整个集合，因此在并发写入时有较好的性能。

当多个线程共享访问一个公共集合时，ConcurrentLinkedQueue 是一个恰当的选择。ConcurrentLinkedQueue 不允许使用 null 元素。ConcurrentLinkedQueue 实现了多线程的高效访问，多个线程访问 ConcurrentLinkedQueue 集合时无须等待。

在默认情况下，ConcurrentHashMap 支持 16 个线程并发写入，当有超过 16 个线程并发向该 Map 中写入数据时，可能有一些线程需要等待。实际上，程序通过设置 concurrencyLevel 构造参数（默认值为 16）来支持更多的并发写入线程。

与前面介绍的 HashMap 和普通集合不同的是，因为 ConcurrentLinkedQueue 和 ConcurrentHashMap 支持多线程并发访问，所以当使用迭代器来遍历集合元素时，该迭代器可能不能反映出创建迭代器之后所做的修改，但程序不会抛出任何异常。

Java 8 扩展了 ConcurrentHashMap 的功能，Java 8 为该类新增了 30 多个新方法，这些方法可借助于 Stream 和 Lambda 表达式支持执行聚集操作。ConcurrentHashMap 新增的方法大致可分为如下三类。

- forEach 系列（forEach,forEachKey, forEachValue, forEachEntry）
- search 系列（search, searchKeys, searchValues, searchEntries）
- reduce 系列（reduce, reduceToDouble, reduceToLong, reduceKeys, reduceValues）

除此之外，ConcurrentHashMap 还新增了 mappingCount()、newKeySet()等方法，增强后的 ConcurrentHashMap 更适合作为缓存实现类使用。

注意：

使用 java.util 包下的 Collection 作为集合对象时，如果该集合对象创建迭代器后集合元素发生改变，则会引发 ConcurrentModificationException 异常。

由于 CopyOnWriteArraySet 的底层封装了 CopyOnWriteArrayList，因此它的实现机制完全类似于 CopyOnWriteArrayList 集合。

对于 CopyOnWriteArrayList 集合，正如它的名字所暗示的，它采用复制底层数组的方式来实现写操作。

当线程对 CopyOnWriteArrayList 集合执行读取操作时，线程将会直接读取集合本身，无须加锁与阻塞。当线程对 CopyOnWriteArrayList 集合执行写入操作时（包括调用 add()、remove()、set()等方法），该集合会在底层复制一份新的数组，接下来对新的数组执行写入操作。由于对 CopyOnWriteArrayList 集合的写入操作都是对数组的副本执行操作，因此它是线程安全的。

需要指出的是，由于 CopyOnWriteArrayList 执行写入操作时需要频繁地复制数组，性能比较差，但由于读操作与写操作不是操作同一个数组，而且读操作也不需要加锁，因此读操作就很快、很安全。由此可见，CopyOnWriteArrayList 适合用在读取操作远远大于写入操作的场景中，例如缓存等。

16.9.4 Java 9 新增的发布-订阅框架

Java 9 新增了一个发布-订阅框架，该框架是基于异步响应流的。这个发布-订阅框架可以非常方便地处理异步线程之间的流数据交换（比如两个线程之间需要交换数据）。而且这个发布-订阅框架不需要使用数据中心来缓冲数据，同时具有非常高效的性能。

这个发布-订阅框架使用 Flow 类的 4 个静态内部接口作为核心 API。

- Flow.Publisher：代表数据发布者、生产者。

- Flow.Subscriber：代表数据订阅者、消费者。
- Flow.Subscription：代表发布者和订阅者之间的链接纽带。订阅者既可通过调用该对象的 request()方法来获取数据项，也可通过调用对象的 cancel()方法来取消订阅。
- Flow.Processor：数据处理器，它可同时作为发布者和订阅者使用。

Flow.Publisher 发布者作为生产者，负责发布数据项，并注册订阅者。Flow.Publisher 接口定义了如下方法来注册订阅者。

- void subscribe(Flow.Subscriber<? super T> subscriber)：程序调用此方法注册订阅者时，会触发订阅者的 onSubscribe()方法，而 Flow.Subscription 对象作为参数传给该方法；如果注册失败，将会触发订阅者的 onError() 方法。

Flow.Subscriber 接口定义了如下方法。

- void onSubscribe(Flow.Subscription subscription)：订阅者注册时自动触发该方法。
- void onComplete()：当订阅结束时触发该方法。
- void onError(Throwable throwable)：当订阅失败时触发该方法。
- void onNext(T item)：订阅者从发布者处获取数据项时触发该方法，订阅者可通过该方法获取数据项。

为了处理一些通用发布者的场景，Java 9 为 Flow.Publisher 提供了一个 SubmissionPublisher 实现类，它可向当前订阅者异步提交非空的数据项，直到它被关闭。每个订阅者都能以相同的顺序接收到新提交的数据项。

程序创建 SubmissionPublisher 对象时，需要传入一个线程池作为底层支撑；该类也提供了一个无参数的构造器，该构造器使用 ForkJoinPool.commonPool()方法来提交发布者，以此实现发布者向订阅者提供数据项的异步特性。

下面程序示范了使用 SubmissionPublisher 作为发布者的用法。

```
public class PubSubTest
{
    public static void main(String[] args)
    {
        // 创建一个 SubmissionPublisher 作为发布者
        SubmissionPublisher<String> publisher = new SubmissionPublisher<>();
        // 创建订阅者
        MySubscriber<String> subscriber = new MySubscriber<>();
        // 注册订阅者
        publisher.subscribe(subscriber);
        // 发布几个数据项
        System.out.println("开发发布数据...");
        List.of("Java", "Kotlin", "Go", "Erlang", "Swift", "Lua")
            .forEach(im -> {
            // 提交数据
            publisher.submit(im);
            try
            {
                Thread.sleep(500);
            }
            catch (Exception ex){}
        });
        // 发布结束
        publisher.close();
        // 发布结束后，为了让发布者线程不会死亡，暂停线程
        synchronized("fkjava")
        {
            try
            {
                "fkjava".wait();
            }
            catch (Exception ex){}
        }
    }
```

```
}
// 创建订阅者
class MySubscriber<T> implements Subscriber<T>
{
    // 发布者与订阅者之间的纽带
    private Subscription subscription;
    @Override  // 订阅时触发该方法
    public void onSubscribe(Subscription subscription)
    {
        this.subscription = subscription;
        // 开始请求数据
        subscription.request(1);
    }
    @Override  // 接收到数据时触发该方法
    public void onNext(T item)
    {
        System.out.println("获取到数据: " + item);
        // 请求下一条数据
        subscription.request(1);
    }
    @Override // 订阅出错时触发该方法
    public void onError(Throwable t)
    {
        t.printStackTrace();
        synchronized("fkjava")
        {
            "fkjava".notifyAll();
        }
    }
    @Override  // 订阅结束时触发该方法
    public void onComplete()
    {
        System.out.println("订阅结束");
        synchronized("fkjava")
        {
            "fkjava".notifyAll();
        }
    }
}
```

上面程序中第一行粗体字代码用于创建 SubmissionPublisher 对象，该对象可作为发布者；第二行粗体字代码用于创建订阅者对象，该订阅者类是一个自定义类；第三行粗体字代码用于注册订阅者。

完成上面步骤之后，程序即可调用 SubmissionPublisher 对象的 submit()方法来发布数据项，发布者通过该方法发布数据项。

上面程序实现了一个自定义的订阅者，该订阅者实现了 Subscriber 接口的 4 个方法，重点就是实现 onNext()方法——当订阅者获取到数据时就会触发该方法，订阅者通过该方法接收数据。至于订阅者接收到数据项之后的处理，则取决于程序的业务需求。

运行该程序，可以看到订阅者逐项获得数据的过程。

16.10 本章小结

本章主要介绍了 Java 的多线程编程支持；简要介绍了线程的基本概念，并讲解了线程和进程之间的区别与联系。本章详细讲解了如何创建、启动多线程，并对比了三种创建多线程方式之间的优势和劣势，也详细介绍了线程的生命周期。本章通过示例程序示范了控制线程的几个方法，还详细讲解了线程同步的意义和必要性，并介绍了三种不同的线程同步方法：同步方法、同步代码块和显式使用 Lock 控制线程同步。本章也介绍了三种实现线程通信的方式：使用同步监视器的方法实现通信、显式使用 Condition 对象实现线程通信和使用阻塞对象实现线程通信。

此外，本章还介绍了线程组和线程池，由于线程属于创建成本较大的对象，因此程序应该考虑复用

线程，线程池是在实际开发中不错的选择。

本章最后介绍了线程相关的工具类，比如ThreadLocal、线程安全的集合类，以及如何使用Collections包装线程不安全的集合类。

本章练习

1. 写2个线程，其中一个线程打印1~52，另一个线程打印A~Z，打印顺序应该是12A34B56C…5152Z。该习题需要利用多线程通信的知识。

2. 假设车库有3个车位（可以用boolean[]数组来表示车库）可以停车，写一个程序模拟多个用户开车离开、停车入库的效果。注意：车位有车时不能停车。

CHAPTER

17

第17章 网络编程

本章要点

- 计算机网络基础
- IP 地址和端口
- 使用 InetAddress 包装 IP 地址
- 使用 URLEncoder 和 URLDecoder 工具类
- 使用 URLConnection 访问远程资源
- TCP 协议基础
- 使用 ServerSocket 和 Socket
- 使用 NIO 实现非阻塞式网络通信
- 使用 AIO 实现异步网络通信
- UDP 协议基础
- 使用 DatagramSocket 发送/接收数据报（DatagramPacket）
- 使用 MulticastSocket 实现多点广播
- 通过 Proxy 使用代理服务器
- 通过 ProxySelector 使用代理服务器

本章将主要介绍 Java 网络通信的支持，通过这些网络支持类，Java 程序可以非常方便地访问互联网上的 HTTP 服务、FTP 服务等，并可以直接取得互联网上的远程资源，还可以向远程资源发送 GET、POST 请求。

本章先简要介绍计算机网络的基础知识，包括 IP 地址和端口等概念，这些知识是网络编程的基础。本章会详细介绍 InetAddress、URLDecoder、URLEncoder、URL 和 URLConnection 等网络工具类，并会深入介绍通过 URLConnection 发送请求、访问远程资源等操作。

本章将重点介绍 Java 提供的 TCP 网络通信支持，包括如何利用 ServerSocket 建立 TCP 服务器，利用 Socket 建立 TCP 客户端。实际上 Java 的网络通信非常简单，服务器端通过 ServerSocket 建立监听，客户端通过 Socket 连接到指定服务器后，通信双方就可以通过 IO 流进行通信。本章将以采用逐步迭代的方式开发一个 C/S 结构多人网络聊天工具为例，向读者介绍基于 TCP 协议的网络编程。

本章还将重点介绍 Java 提供的 UDP 网络通信支持，主要介绍如何使用 DatagramSocket 来发送、接收数据报（DatagramPacket），并讲解如何使用 MulticastSocket 来实现多点广播通信。本章也将以开发局域网通信程序为例来介绍 MulticastSocket 和 DatagramSocket 的实际用法。

本章最后还会介绍利用 Proxy 和 ProxySelector 在 Java 程序中通过代理服务器访问远程资源。

17.1 网络编程的基础知识

时至今日，计算机网络缩短了人们之间的距离，把“地球村”变成现实，网络应用已经成为计算机领域最广泛的应用。

17.1.1 网络基础知识

所谓计算机网络，就是把分布在不同地理区域的计算机与专门的外部设备用通信线路互连成一个规模大、功能强的网络系统，从而使众多的计算机可以方便地互相传递信息，共享硬件、软件、数据信息等资源。

计算机网络是现代通信技术与计算机技术相结合的产物，计算机网络可以提供以下一些主要功能。

- 资源共享。
- 信息传输与集中处理。
- 均衡负荷与分布处理。
- 综合信息服务。

通过计算机网络可以向全社会提供各种经济信息、科研情报和咨询服务。其中，国际互联网 Internet 上的全球信息网（WWW，World Wide Web）服务就是一个最典型也是最成功的例子。实际上，今天的网络承载绝大部分大型企业的运转，一个大型的、全球性的企业或组织的日常工作流程都是建立在互联网基础之上的。

计算机网络的品种很多，根据各种不同的分类原则，可以得到各种不同类型的计算机网络。计算机网络通常是按照规模大小和延伸范围来分类的，常见的划分为：局域网（LAN）、城域网（MAN）、广域网（WAN）。Internet 可以视为世界上最大的广域网。

如果按照网络的拓扑结构来划分，可以分为星型网络、总线型网络、环型网络、树型网络、星型环型网络等；如果按照网络的传输介质来划分，可以分为双绞线网、同轴电缆网、光纤网和卫星网等。

计算机网络中实现通信必须有一些约定，这些约定被称为通信协议。通信协议负责对传输速率、传输代码、代码结构、传输控制步骤、出错控制等制定处理标准。为了让两个节点之间能进行对话，必须在它们之间建立通信工具，使彼此之间能进行信息交换。

通信协议通常由三部分组成：一是语义部分，用于决定双方对话的类型；二是语法部分，用于决定双方对话的格式；三是变换规则，用于决定通信双方的应答关系。

国际标准化组织 ISO 于 1978 年提出“开放系统互连参考模型”，即著名的 OSI（Open System Interconnection）。

开放系统互连参考模型力求将网络简化，并以模块化的方式来设计网络。

开放系统互连参考模型把计算机网络分成物理层、数据链路层、网络层、传输层、会话层、表示层、

应用层七层，受到计算机界和通信业的极大关注。通过十多年的发展和推进，OSI 模式已成为各种计算机网络结构的参考标准。

图 17.1 显示了 OSI 参考模型的推荐分层。

前面介绍过通信协议是网络通信的基础，IP 协议则是一种非常重要的通信协议。IP（Internet Protocol）协议又称互联网协议，是支持网间互联的数据报协议。它提供网间连接的完善功能，包括 IP 数据报规定互联网络范围内的地址格式。

经常与 IP 协议放在一起的还有 TCP（Transmission Control Protocol）协议，即传输控制协议，它规定一种可靠的数据信息传递服务。虽然 IP 和 TCP 这两个协议功能不尽相同，也可以分开单独使用，但它们是在同一个时期作为一个协议来设计的，并且在功能上也是互补的。因此实际使用中常常把这两个协议统称为 TCP/IP 协议，TCP/IP 协议最早出现在 UNIX 操作系统中，现在几乎所有的操作系统都支持 TCP/IP 协议，因此 TCP/IP 协议也是 Internet 中最常用的基础协议。

按 TCP/IP 协议模型，网络通常被分为一个四层模型，这个四层模型和前面的 OSI 七层模型有大致的对应关系，图 17.2 显示了 TCP/IP 分层模型和 OSI 分层模型之间的对应关系。

图 17.1 OSI 参考模型的推荐分层

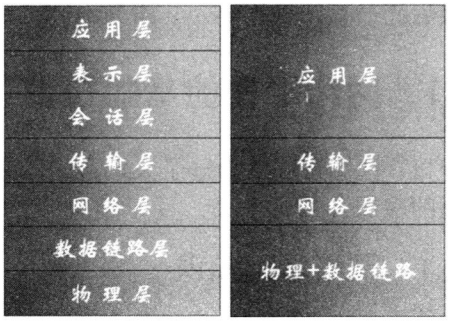


图 17.2 OSI 分层模型和 TCP/IP 分层模型的对应关系

17.1.2 IP 地址和端口号

IP 地址用于唯一地标识网络中的一个通信实体，这个通信实体既可以是一台主机，也可以是一台打印机，或者是路由器的某一个端口。而在基于 IP 协议网络中传输的数据包，都必须使用 IP 地址来进行标识。

就像写一封信，要标明收信人的通信地址和发信人的地址，而邮政工作人员则通过该地址来决定邮件的去向。类似的过程也发生在计算机网络里，每个被传输的数据包也要包括一个源 IP 地址和一个目的 IP 地址，当该数据包在网络中进行传输时，这两个地址要保持不变，以确保网络设备总能根据确定的 IP 地址，将数据包从源通信实体送往指定的目的通信实体。

IP 地址是数字型的，IP 地址是一个 32 位（32bit）整数，但通常为了便于记忆，通常把它分成 4 个 8 位的二进制数，每 8 位之间用圆点隔开，每个 8 位整数可以转换成一个 0~255 的十进制整数，因此日常看到的 IP 地址常常是这种形式：202.9.128.88。

NIC（Internet Network Information Center）统一负责全球 Internet IP 地址的规划、管理，而 Inter NIC、APNIC、RIPE 三大网络信息中心具体负责美国及其他地区的 IP 地址分配。其中 APNIC 负责亚太地区的 IP 管理，我国申请 IP 地址也要通过 APNIC，APNIC 的总部设在日本东京大学。

IP 地址被分成了 A、B、C、D、E 五类，每个类别的网络标识和主机标识各有规则。

- A 类：10.0.0.0~10.255.255.255
- B 类：172.16.0.0~172.31.255.255
- C 类：192.168.0.0~192.168.255.255

IP 地址用于唯一地标识网络上的一个通信实体，但一个通信实体可以有多个通信程序同时提供网络服务，此时还需要使用端口。

端口是一个 16 位的整数，用于表示数据交给哪个通信程序处理。因此，端口就是应用程序与外界交流的出入口，它是一种抽象的软件结构，包括一些数据结构和 I/O（基本输入/输出）缓冲区。

不同的应用程序处理不同端口上的数据，同一台机器上不能有两个程序使用同一个端口，端口号可

以从 0 到 65535，通常将它分为如下三类。

- 公认端口（Well Known Ports）：从 0 到 1023，它们紧密绑定（Binding）一些特定的服务。
- 注册端口（Registered Ports）：从 1024 到 49151，它们松散地绑定一些服务。应用程序通常应该使用这个范围内的端口。
- 动态和/或私有端口（Dynamic and/or Private Ports）：从 49152 到 65535，这些端口是应用程序使用的动态端口，应用程序一般不会主动使用这些端口。

如果把 IP 地址理解为某个人所在地方的地址（包括街道和门牌号），但仅有地址还是找不到这个人，还需要知道他所在的房号才可以找到这个人。因此如果把应用程序当作人，把计算机网络当作类似邮递员的角色，当一个程序需要发送数据时，需要指定目的地的 IP 地址和端口，如果指定了正确的 IP 地址和端口号，计算机网络就可以将数据送给该 IP 地址和端口所对应的程序。

17.2 Java 的基本网络支持

Java 为网络支持提供了 java.net 包，该包下的 URL 和 URLConnection 等类提供了以编程方式访问 Web 服务的功能，而 URLDecoder 和 URLEncoder 则提供了普通字符串和 application/x-www-form-urlencoded MIME 字符串相互转换的静态方法。

17.2.1 使用 InetAddress

Java 提供了 InetAddress 类来代表 IP 地址，InetAddress 下还有两个子类：Inet4Address、Inet6Address，它们分别代表 Internet Protocol version 4（IPv4）地址和 Internet Protocol version 6（IPv6）地址。

InetAddress 类没有提供构造器，而是提供了如下两个静态方法来获取 InetAddress 实例。

- getByName(String host)：根据主机获取对应的 InetAddress 对象。
- getByAddress(byte[] addr)：根据原始 IP 地址来获取对应的 InetAddress 对象。

InetAddress 还提供了如下三个方法来获取 InetAddress 实例对应的 IP 地址和主机名。

- String getCanonicalHostName()：获取此 IP 地址的全限定域名。
- String getHostAddress()：返回该 InetAddress 实例对应的 IP 地址字符串（以字符串形式）。
- String getHostName()：获取此 IP 地址的主机名。

除此之外，InetAddress 类还提供了一个 getLocalHost()方法来获取本机 IP 地址对应的 InetAddress 实例。

InetAddress 类还提供了一个 isReachable()方法，用于测试是否可以到达该地址。该方法将尽最大努力试图到达主机，但防火墙和服务器配置可能阻塞请求，使得它在访问某些特定的端口时处于不可达状态。如果可以获得权限，典型的实现将使用 ICMP ECHO REQUEST；否则它将试图在目标主机的端口 7（Echo）上建立 TCP 连接。下面程序测试了 InetAddress 类的简单用法。

程序清单：codes\17\17.2\InetAddressTest.java

```
public class InetAddressTest
{
    public static void main(String[] args)
        throws Exception
    {
        // 根据主机名来获取对应的 InetAddress 实例
        InetAddress ip = InetAddress.getByName("www.crazyit.org");
        // 判断是否可达
        System.out.println("crazyit 是否可达：" + ip.isReachable(2000));
        // 获取该 InetAddress 实例的 IP 字符串
        System.out.println(ip.getHostAddress());
        // 根据原始 IP 地址来获取对应的 InetAddress 实例
        InetAddress local = InetAddress.getByAddress(
            new byte[]{127,0,0,1});
        System.out.println("本机是否可达：" + local.isReachable(5000));
        // 获取该 InetAddress 实例对应的全限定域名
        System.out.println(local.getCanonicalHostName());
    }
}
```

上面程序简单地示范了 InetAddress 类的几个方法的用法，InetAddress 类本身并没有提供太多功能，它代表一个 IP 地址对象，是网络通信的基础，在后面介绍中将大量使用该类。

17.2.2 使用 URLDecoder 和 URLEncoder

URLDecoder 和 URLEncoder 用于完成普通字符串和 application/x-www-form-urlencoded MIME 字符串之间的相互转换。可能有读者觉得后一个字符串非常专业，以为又是什么特别高深的知识，其实不是。

在介绍 application/x-www-form-urlencoded MIME 字符串之前，先使用 www.google.com.hk 搜索关键字“疯狂 java”，将看到如图 17.3 所示的界面。

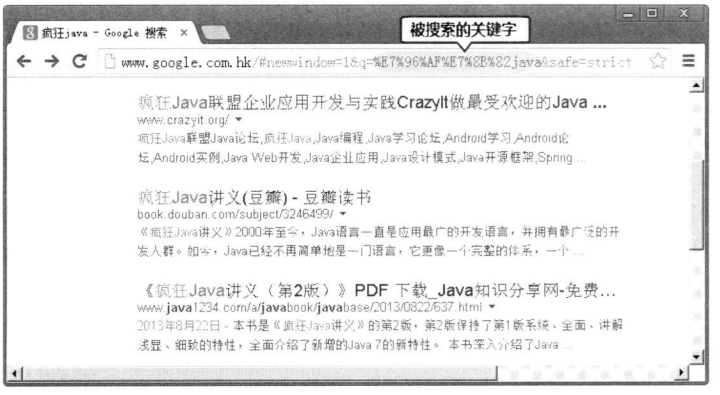


图 17.3　搜索关键字包含中文

从图 17.3 中可以看出，当关键字包含中文时，这些关键字就会变成如图 17.3 所示的“乱码”——实际上这不是乱码，这就是所谓的 application/x-www-form-urlencoded MIME 字符串。

当 URL 地址里包含非西欧字符的字符串时，系统会将这些非西欧字符串转换成如图 17.3 所示的特殊字符串。编程过程中可能涉及普通字符串和这种特殊字符串的相关转换，这就需要使用 URLDecoder 和 URLEncoder 类。

- URLDecoder 类包含一个 decode(String s,String enc)静态方法，它可以将看上去是乱码的特殊字符串转换成普通字符串。
- URLEncoder 类包含一个 encode(String s,String enc)静态方法，它可以将普通字符串转换成 application/x-www-form-urlencoded MIME 字符串。

下面程序示范了如何将图 17.3 所示地址栏中的“乱码”转换成普通字符串，并示范了如何将普通字符串转换成 application/x-www-form-urlencoded MIME 字符串。

程序清单：codes\17\17.2\URLDecoderTest.java

```
public class URLDecoderTest
{
    public static void main(String[] args)
        throws Exception
    {
        // 将 application/x-www-form-urlencoded 字符串
        // 转换成普通字符串
        // 其中的字符串直接从图 17.3 所示的窗口中复制过来
        String keyWord = URLDecoder.decode(
            "%E7%96%AF%E7%8B%82java", "utf-8");
        System.out.println(keyWord);
        // 将普通字符串转换成
        // application/x-www-form-urlencoded 字符串
        String urlStr = URLEncoder.encode(
            "疯狂 Android 讲义" , "GBK");
        System.out.println(urlStr);
    }
}
```

上面程序中的粗体字代码用于完成普通字符串和 application/x-www-form-urlencoded MIME 字符串之间的转换。运行上面程序，将看到如下输出：

```
疯狂 java
%B7%E8%BF%F1Android%BD%B2%D2%E5
```

提示：

仅包含西欧字符的普通字符串和 application/x-www-form-urlencoded MIME 字符串无须转换，而包含中文字符的普通字符串则需要转换，转换方法是每个中文字符占两个字节，每个字节可以转换成两个十六进制的数字，所以每个中文字符将转换成“%XX%XX”的形式。当然，采用不同的字符集时，每个中文字符对应的字节数并不完全相同，所以使用 URLEncoder 和 URLDecoder 进行转换时也需要指定字符集。

17.2.3 URL、URLConnection 和 URLPermission

URL（Uniform Resource Locator）对象代表统一资源定位器，它是指向互联网“资源”的指针。资源可以是简单的文件或目录，也可以是对更为复杂对象的引用，例如对数据库或搜索引擎的查询。在通常情况下，URL 可以由协议名、主机、端口和资源组成，即满足如下格式：

```
protocol://host:port/resourceName
```

例如如下的 URL 地址：

```
http://www.crazyit.org/index.php
```

提示：

JDK 中还提供了一个 URI（Uniform Resource Identifiers）类，其实例代表一个统一资源标识符，Java 的 URI 不能用于定位任何资源，它的唯一作用就是解析。与此对应的是，URL 则包含一个可打开到达该资源的输入流，可以将 URL 理解成 URI 的特例。

URL 类提供了多个构造器用于创建 URL 对象，一旦获得了 URL 对象之后，就可以调用如下方法来访问该 URL 对应的资源。

- String getFile()：获取该 URL 的资源名。
- String getHost()：获取该 URL 的主机名。
- String getPath()：获取该 URL 的路径部分。
- int getPort()：获取该 URL 的端口号。
- String getProtocol()：获取该 URL 的协议名称。
- String getQuery()：获取该 URL 的查询字符串部分。
- URLConnection openConnection()：返回一个 URLConnection 对象，它代表了与 URL 所引用的远程对象的连接。
- InputStream openStream()：打开与此URL的连接，并返回一个用于读取该URL资源的InputStream。

URL 对象中的前面几个方法都非常容易理解，而该对象提供的 openStream()方法可以读取该 URL 资源的 InputStream，通过该方法可以非常方便地读取远程资源——甚至实现多线程下载。如下程序实现了一个多线程下载工具类。

程序清单：codes\17\17.2\DownUtil.java

```
public class DownUtil
{
    // 定义下载资源的路径
    private String path;
    // 指定所下载的文件的保存位置
    private String targetFile;
    // 定义需要使用多少个线程下载资源
    private int threadNum;
    // 定义下载的线程对象
    private DownThread[] threads;
    // 定义下载的文件的总大小
    private int fileSize;

    public DownUtil(String path, String targetFile, int threadNum)
    {
        this.path = path;
```

```
        this.threadNum = threadNum;
        // 初始化 threads 数组
        threads = new DownThread[threadNum];
        this.targetFile = targetFile;
    }
    public void download() throws Exception
    {
        URL url = new URL(path);
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
        conn.setConnectTimeout(5 * 1000);
        conn.setRequestMethod("GET");
        conn.setRequestProperty(
            "Accept",
            "image/gif, image/jpeg, image/pjpeg, image/pjpeg, "
            + "application/x-shockwave-flash, application/xaml+xml, "
            + "application/vnd.ms-xpsdocument, application/x-ms-xbap, "
            + "application/x-ms-application, application/vnd.ms-excel, "
            + "application/vnd.ms-powerpoint, application/msword, */*");
        conn.setRequestProperty("Accept-Language", "zh-CN");
        conn.setRequestProperty("Charset", "UTF-8");
        conn.setRequestProperty("Connection", "Keep-Alive");
        // 得到文件大小
        fileSize = conn.getContentLength();
        conn.disconnect();
        int currentPartSize = fileSize / threadNum + 1;
        RandomAccessFile file = new RandomAccessFile(targetFile, "rw");
        // 设置本地文件的大小
        file.setLength(fileSize);
        file.close();
        for (int i = 0; i < threadNum; i++)
        {
            // 计算每个线程下载的开始位置
            int startPos = i * currentPartSize;
            // 每个线程使用一个 RandomAccessFile 进行下载
            RandomAccessFile currentPart = new RandomAccessFile(targetFile,
                "rw");
            // 定位该线程的下载位置
            currentPart.seek(startPos);
            // 创建下载线程
            threads[i] = new DownThread(startPos, currentPartSize,
                currentPart);
            // 启动下载线程
            threads[i].start();
        }
    }
    // 获取下载的完成百分比
    public double getCompleteRate()
    {
        // 统计多个线程已经下载的总大小
        int sumSize = 0;
        for (int i = 0; i < threadNum; i++)
        {
            sumSize += threads[i].length;
        }
        // 返回已经完成的百分比
        return sumSize * 1.0 / fileSize;
    }
    private class DownThread extends Thread
    {
        // 当前线程的下载位置
        private int startPos;
        // 定义当前线程负责下载的文件大小
        private int currentPartSize;
        // 当前线程需要下载的文件块
        private RandomAccessFile currentPart;
        // 定义该线程已下载的字节数
        public int length;
        public DownThread(int startPos, int currentPartSize,
            RandomAccessFile currentPart)
        {
```

```
            this.startPos = startPos;
            this.currentPartSize = currentPartSize;
            this.currentPart = currentPart;
        }
        public void run()
        {
            try
            {
                URL url = new URL(path);
                HttpURLConnection conn = (HttpURLConnection)url
                    .openConnection();
                conn.setConnectTimeout(5 * 1000);
                conn.setRequestMethod("GET");
                conn.setRequestProperty(
                    "Accept",
                    "image/gif, image/jpeg, image/pjpeg, image/pjpeg, "
                    + "application/x-shockwave-flash, application/xaml+xml, "
                    + "application/vnd.ms-xpsdocument, application/x-ms-xbap, "
                    + "application/x-ms-application, application/vnd.ms-excel, "
                    + "application/vnd.ms-powerpoint, application/msword, */*");
                conn.setRequestProperty("Accept-Language", "zh-CN");
                conn.setRequestProperty("Charset", "UTF-8");
                InputStream inStream = conn.getInputStream();
                // 跳过 startPos 个字节，表明该线程只下载自己负责的那部分文件
                inStream.skip(this.startPos);
                byte[] buffer = new byte[1024];
                int hasRead = 0;
                // 读取网络数据，并写入本地文件
                while (length < currentPartSize
                    && (hasRead = inStream.read(buffer)) != -1)
                {
                    currentPart.write(buffer, 0, hasRead);
                    // 累计该线程下载的总大小
                    length += hasRead;
                }
                currentPart.close();
                inStream.close();
            }
            catch (Exception e)
            {
                e.printStackTrace();
            }
        }
    }
}
```

上面程序中定义了DownThread线程类，该线程负责读取从start开始，到end结束的所有字节数据，并写入RandomAccessFile对象。这个DownThread线程类的run()方法就是一个简单的输入、输出实现。

程序中DownUtils类中的download()方法负责按如下步骤来实现多线程下载。

1. 创建URL对象。
2. 获取指定URL对象所指向资源的大小（通过getContentLength()方法获得），此处用到了URLConnection类，该类代表Java应用程序和URL之间的通信链接。后面还有关于URLConnection更详细的介绍。
3. 在本地磁盘上创建一个与网络资源具有相同大小的空文件。
4. 计算每个线程应该下载网络资源的哪个部分（从哪个字节开始，到哪个字节结束）。
5. 依次创建、启动多个线程来下载网络资源的指定部分。

提示：

上面程序已经实现了多线程下载的核心代码，如果要实现断点下载，则需要额外增加一个配置文件（读者可以发现，所有的断点下载工具都会在下载开始时生成两个文件：一个是与网络资源具有相同大小的空文件，一个是配置文件），该配置文件分别记录每个线程已经下载到哪个字节，当网络断开后再次开始下载时，每个线程根据配置文件里记录的位置向后下载即可。

有了上面的DownUtil工具类之后，接下来就可以在主程序中调用该工具类的down()方法执行下载，如下程序所示。

程序清单：codes\17\17.2\MultiThreadDown.java

```
public class MultiThreadDown
{
    public static void main(String[] args) throws Exception
    {
        // 初始化 DownUtil 对象
        final DownUtil downUtil = new DownUtil("http://www.crazyit.org/"
            + "attachments/month_1403/1403202355ff6cc9a4fbf6f14a.png"
            , "ios.png", 4);
        // 开始下载
        downUtil.download();
        new Thread(() -> {
            while(downUtil.getCompleteRate() < 1)
            {
                // 每隔 0.1 秒查询一次任务的完成进度
                // GUI 程序中可根据该进度来绘制进度条
                System.out.println("已完成: "
                    + downUtil.getCompleteRate());
                try
                {
                    Thread.sleep(1000);
                }
                catch (Exception ex){}
            }
        }).start();
    }
}
```

运行上面程序，即可看到程序从www.crazyit.org下载得到一份名为ios.png的图片文件。

上面程序还用到URLConnection和HttpURLConnection对象，其中前者表示应用程序和URL之间的通信连接，后者表示与URL之间的HTTP连接。程序可以通过URLConnection实例向该URL发送请求、读取URL引用的资源。

Java 8新增了一个URLPermission工具类，用于管理HttpURLConnection的权限问题，如果在HttpURLConnection安装了安全管理器，通过该对象打开连接时就需要先获得权限。

通常创建一个和URL的连接，并发送请求、读取此URL引用的资源需要如下几个步骤。

1. 通过调用URL对象的openConnection()方法来创建URLConnection对象。
2. 设置URLConnection的参数和普通请求属性。
3. 如果只是发送GET方式请求，则使用connect()方法建立和远程资源之间的实际连接即可；如果需要发送POST方式的请求，则需要获取URLConnection实例对应的输出流来发送请求参数。
4. 远程资源变为可用，程序可以访问远程资源的头字段或通过输入流读取远程资源的数据。

在建立和远程资源的实际连接之前，程序可以通过如下方法来设置请求头字段。

- setAllowUserInteraction()：设置该URLConnection的allowUserInteraction请求头字段的值。
- setDoInput()：设置该URLConnection的doInput请求头字段的值。
- setDoOutput()：设置该URLConnection的doOutput请求头字段的值。
- setIfModifiedSince()：设置该URLConnection的ifModifiedSince请求头字段的值。
- setUseCaches()：设置该URLConnection的useCaches请求头字段的值。

除此之外，还可以使用如下方法来设置或增加通用头字段。

- setRequestProperty(String key, String value)：设置该URLConnection的key请求头字段的值为value。如下代码所示：

```
conn.setRequestProperty("accept" , "*/*")
```

- addRequestProperty(String key, String value)：为该URLConnection的key请求头字段增加value值，该方法并不会覆盖原请求头字段的值，而是将新值追加到原请求头字段中。

当远程资源可用之后，程序可以使用以下方法来访问头字段和内容。

- Object getContent()：获取该 URLConnection 的内容。
- String getHeaderField(String name)：获取指定响应头字段的值。
- getInputStream()：返回该 URLConnection 对应的输入流，用于获取 URLConnection 响应的内容。
- getOutputStream()：返回该 URLConnection 对应的输出流，用于向 URLConnection 发送请求参数。

getHeaderField()方法用于根据响应头字段来返回对应的值。而某些头字段由于经常需要访问，所以 Java 提供了以下方法来访问特定响应头字段的值。

- getContentEncoding()：获取 content-encoding 响应头字段的值。
- getContentLength()：获取 content-length 响应头字段的值。
- getContentType()：获取 content-type 响应头字段的值。
- getDate()：获取 date 响应头字段的值。
- getExpiration()：获取 expires 响应头字段的值。
- getLastModified()：获取 last-modified 响应头字段的值。

注意：

如果既要使用输入流读取 URLConnection 响应的内容，又要使用输出流发送请求参数，则一定要先使用输出流，再使用输入流。

下面程序示范了如何向 Web 站点发送 GET 请求、POST 请求，并从 Web 站点取得响应。

程序清单：codes\17\17.2\GetPostTest.java

```
public class GetPostTest
{
    /**
     * 向指定 URL 发送 GET 方式的请求
     * @param url 发送请求的 URL
     * @param param 请求参数，格式满足 name1=value1&name2=value2 的形式
     * @return URL 代表远程资源的响应
     */
    public static String sendGet(String url , String param)
    {
        String result = "";
        String urlName = url + "?" + param;
        try
        {
            URL realUrl = new URL(urlName);
            // 打开和 URL 之间的连接
            URLConnection conn = realUrl.openConnection();
            // 设置通用的请求属性
            conn.setRequestProperty("accept", "*/*");
            conn.setRequestProperty("connection", "Keep-Alive");
            conn.setRequestProperty("user-agent"
                , "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1)");
            // 建立实际的连接
            conn.connect();
            // 获取所有的响应头字段
            Map<String, List<String>> map = conn.getHeaderFields();
            // 遍历所有的响应头字段
            for (String key : map.keySet())
            {
                System.out.println(key + "--->" + map.get(key));
            }
            try(
                // 定义 BufferedReader 输入流来读取 URL 的响应
                BufferedReader in = new BufferedReader(
                    new InputStreamReader(conn.getInputStream() , "utf-8")))
            {
                String line;
                while ((line = in.readLine())!= null)
                {
```

```
                result += "\n" + line;
            }
        }
    }
    catch(Exception e)
    {
        System.out.println("发送 GET 请求出现异常! " + e);
        e.printStackTrace();
    }
    return result;
}
/**
 * 向指定 URL 发送 POST 方式的请求
 * @param url 发送请求的 URL
 * @param param 请求参数，格式应该满足 name1=value1&name2=value2 的形式
 * @return URL 代表远程资源的响应
 */
public static String sendPost(String url , String param)
{
    String result = "";
    try
    {
        URL realUrl = new URL(url);
        // 打开和 URL 之间的连接
        URLConnection conn = realUrl.openConnection();
        // 设置通用的请求属性
        conn.setRequestProperty("accept", "*/*");
        conn.setRequestProperty("connection", "Keep-Alive");
        conn.setRequestProperty("user-agent",
        "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1)");
        // 发送 POST 请求必须设置如下两行
        conn.setDoOutput(true);
        conn.setDoInput(true);
        try(
          , // 获取 URLConnection 对象对应的输出流
            PrintWriter out = new PrintWriter(conn.getOutputStream()))
        {
            // 发送请求参数
            out.print(param);
            // flush 输出流的缓冲
            out.flush();
        }
        try(
            // 定义 BufferedReader 输入流来读取 URL 的响应
            BufferedReader in = new BufferedReader(new InputStreamReader
                (conn.getInputStream() , "utf-8")))
        {
            String line;
            while ((line = in.readLine())!= null)
            {
                result += "\n" + line;
            }
        }
    }
    catch(Exception e)
    {
        System.out.println("发送 POST 请求出现异常! " + e);
        e.printStackTrace();
    }
    return result;
}
// 提供主方法，测试发送 GET 请求和 POST 请求
public static void main(String args[])
{
    // 发送 GET 请求
    String s = GetPostTest.sendGet("http://localhost:8888/abc/a.jsp"
        , null);
    System.out.println(s);
```

```
        // 发送 POST 请求
        String s1 = GetPostTest.sendPost("http://localhost:8888/abc/login.jsp"
            , "name=crazyit.org&pass=leegang");
        System.out.println(s1);
    }
}
```

上面程序中发送 GET 请求时只需将请求参数放在 URL 字符串之后，以?隔开，程序直接调用 URLConnection 对象的 connect()方法即可，如 sendGet()方法中粗体字代码所示；如果程序要发送 POST 请求，则需要先设置 doIn 和 doOut 两个请求头字段的值，再使用 URLConnection 对应的输出流来发送请求参数，如 sendPost()方法中粗体字代码所示。

不管是发送 GET 请求，还是发送 POST 请求，程序获取 URLConnection 响应的方式完全一样——如果程序可以确定远程响应是字符流，则可以使用字符流来读取；如果程序无法确定远程响应是字符流，则使用字节流读取即可。

注意：

上面程序中发送请求的两个 URL 是部署在本机的 Web 应用（该应用位于 codes\17\17.2\abc 目录中），关于如何创建 Web 应用、编写 JSP 页面请参考疯狂 Java 体系的《轻量级 Java EE 企业应用实战》。由于程序可以使用这种方式向服务器发送请求——相当于提交 Web 应用中的登录表单页，这样就可以让程序不断地变换用户名、密码来提交登录请求，直到返回登录成功，这就是所谓的暴力破解。

17.3 基于 TCP 协议的网络编程

TCP/IP 通信协议是一种可靠的网络协议，它在通信的两端各建立一个 Socket，从而在通信的两端之间形成网络虚拟链路。一旦建立了虚拟的网络链路，两端的程序就可以通过虚拟链路进行通信。Java 对基于 TCP 协议的网络通信提供了良好的封装，Java 使用 Socket 对象来代表两端的通信端口，并通过 Socket 产生 IO 流来进行网络通信。

17.3.1 TCP 协议基础

IP 协议是 Internet 上使用的一个关键协议，它的全称是 Internet Protocol，即 Internet 协议，通常简称 IP 协议。通过使用 IP 协议，从而使 Internet 成为一个允许连接不同类型的计算机和不同操作系统的网络。

要使两台计算机彼此能进行通信，必须使两台计算机使用同一种“语言”，IP 协议只保证计算机能发送和接收分组数据。IP 协议负责将消息从一个主机传送到另一个主机，消息在传送的过程中被分割成一个个的小包。

尽管计算机通过安装 IP 软件，保证了计算机之间可以发送和接收数据，但 IP 协议还不能解决数据分组在传输过程中可能出现的问题。因此，若要解决可能出现的问题，连上 Internet 的计算机还需要安装 TCP 协议来提供可靠并且无差错的通信服务。

TCP 协议被称作一种端对端协议。这是因为它对两台计算机之间的连接起了重要作用——当一台计算机需要与另一台远程计算机连接时，TCP 协议会让它们建立一个连接：用于发送和接收数据的虚拟链路。

TCP 协议负责收集这些信息包，并将其按适当的次序放好传送，接收端收到后再将其正确地还原。TCP 协议保证了数据包在传送中准确无误。TCP 协议使用重发机制——当一个通信实体发送一个消息给另一个通信实体后，需要收到另一个通信实体的确认信息，如果没有收到另一个通信实体的确认信息，则会再次重发刚才发送的信息。

通过这种重发机制，TCP 协议向应用程序提供了可靠的通信连接，使它能够自动适应网上的各种变化。即使在 Internet 暂时出现堵塞的情况下，TCP 也能够保证通信的可靠性。

图 17.4 显示了 TCP 协议控制两个通信实体互相通信的示意图。

综上所述，虽然 IP 和 TCP 这两个协议的功能不尽相同，也可以分开单独使用，但它们是在同一时期作为一个协议来设计的，并且在功能上也是互补的。只有两者结合起来，才能保证 Internet 在复杂的环境下正常运行。凡是要连接到 Internet 的计算机，都必须同时安装和使用这两个协议，因此在实际中常把这两个协议统称为 TCP/IP 协议。

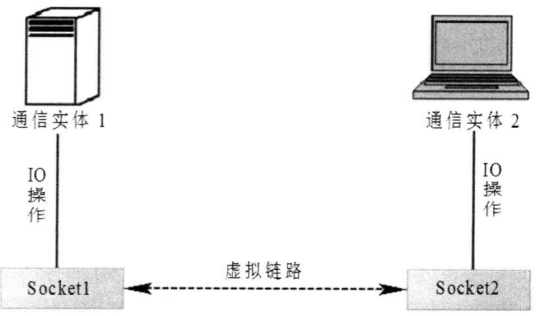


图 17.4　TCP 协议控制两个通信实体互相通信的示意图

17.3.2　使用 ServerSocket 创建 TCP 服务器端

看图 17.4，并没有看出 TCP 通信的两个通信实体之间有服务器端、客户端之分，这是因为此图是两个通信实体已经建立虚拟链路之后的示意图。在两个通信实体没有建立虚拟链路之前，必须有一个通信实体先做出“主动姿态”，主动接收来自其他通信实体的连接请求。

Java 中能接收其他通信实体连接请求的类是 ServerSocket，ServerSocket 对象用于监听来自客户端的 Socket 连接，如果没有连接，它将一直处于等待状态。ServerSocket 包含一个监听来自客户端连接请求的方法。

- Socket accept()：如果接收到一个客户端 Socket 的连接请求，该方法将返回一个与客户端 Socket 对应的 Socket（如图 17.4 所示，每个 TCP 连接有两个 Socket）；否则该方法将一直处于等待状态，线程也被阻塞。

为了创建 ServerSocket 对象，ServerSocket 类提供了如下几个构造器。

- ServerSocket(int port)：用指定的端口 port 来创建一个 ServerSocket。该端口应该有一个有效的端口整数值，即 0~65535。
- ServerSocket(int port,int backlog)：增加一个用来改变连接队列长度的参数 backlog。
- ServerSocket(int port,int backlog,InetAddress localAddr)：在机器存在多个 IP 地址的情况下，允许通过 localAddr 参数来指定将 ServerSocket 绑定到指定的 IP 地址。

当 ServerSocket 使用完毕后，应使用 ServerSocket 的 close()方法来关闭该 ServerSocket。在通常情况下，服务器不应该只接收一个客户端请求，而应该不断地接收来自客户端的所有请求，所以 Java 程序通常会通过循环不断地调用 ServerSocket 的 accept()方法。如下代码片段所示。

```
// 创建一个 ServerSocket，用于监听客户端 Socket 的连接请求
ServerSocket ss = new ServerSocket(30000);
// 采用循环不断地接收来自客户端的请求
while (true)
{
    // 每当接收到客户端 Socket 的请求时，服务器端也对应产生一个 Socket
    Socket s = ss.accept();
    // 下面就可以使用 Socket 进行通信了
    ...
}
```

提示：

上面程序中创建 ServerSocket 没有指定 IP 地址，则该 ServerSocket 将会绑定到本机默认的 IP 地址。程序中使用 30000 作为该 ServerSocket 的端口号，通常推荐使用 1024 以上的端口，主要是为了避免与其他应用程序的通用端口冲突。

17.3.3　使用 Socket 进行通信

客户端通常可以使用 Socket 的构造器来连接到指定服务器，Socket 通常可以使用如下两个构造器。

- Socket(InetAddress/String remoteAddress, int port)：创建连接到指定远程主机、远程端口的 Socket，该构造器没有指定本地地址、本地端口，默认使用本地主机的默认 IP 地址，默认使用系统动态分配的端口。
- Socket(InetAddress/String remoteAddress, int port, InetAddress localAddr, int localPort)：创建连接到

指定远程主机、远程端口的 Socket，并指定本地 IP 地址和本地端口，适用于本地主机有多个 IP 地址的情形。

上面两个构造器中指定远程主机时既可使用 InetAddress 来指定，也可直接使用 String 对象来指定，但程序通常使用 String 对象（如 192.168.2.23）来指定远程 IP 地址。当本地主机只有一个 IP 地址时，使用第一个方法更为简单。如下代码所示。

```
// 创建连接到本机、30000 端口的 Socket
Socket s = new Socket("127.0.0.1" , 30000);
// 下面就可以使用 Socket 进行通信了
...
```

当程序执行上面代码中的粗体字代码时，该代码将会连接到指定服务器，让服务器端的 ServerSocket 的 accept()方法向下执行，于是服务器端和客户端就产生一对互相连接的 Socket。

提示：

上面程序连接到“远程主机”的 IP 地址使用的是 127.0.0.1，这个 IP 地址是一个特殊的地址，它总是代表本机的 IP 地址。因为本书的示例程序的服务器端、客户端都是在本机运行的，所以 Socket 连接的远程主机的 IP 地址使用 127.0.0.1。

当客户端、服务器端产生了对应的 Socket 之后，就得到了如图 17.4 所示的通信示意图，程序无须再区分服务器端、客户端，而是通过各自的 Socket 进行通信。Socket 提供了如下两个方法来获取输入流和输出流。

- InputStream getInputStream()：返回该 Socket 对象对应的输入流，让程序通过该输入流从 Socket 中取出数据。
- OutputStream getOutputStream()：返回该 Socket 对象对应的输出流，让程序通过该输出流向 Socket 中输出数据。

看到这两个方法返回的 InputStream 和 OutputStream，读者应该可以明白 Java 在设计 IO 体系上的苦心了——不管底层的 IO 流是怎样的节点流：文件流也好，网络 Socket 产生的流也好，程序都可以将其包装成处理流，从而提供更多方便的处理。下面以一个最简单的网络通信程序为例来介绍基于 TCP 协议的网络通信。

下面的服务器端程序非常简单，它仅仅建立 ServerSocket 监听，并使用 Socket 获取输出流输出。

程序清单：codes\17\17.3\Server.java

```
public class Server
{
    public static void main(String[] args)
        throws IOException
    {
        // 创建一个 ServerSocket，用于监听客户端 Socket 的连接请求
        ServerSocket ss = new ServerSocket(30000);
        // 采用循环不断地接收来自客户端的请求
        while (true)
        {
            // 每当接收到客户端 Socket 的请求时，服务器端也对应产生一个 Socket
            Socket s = ss.accept();
            // 将 Socket 对应的输出流包装成 PrintStream
            PrintStream ps = new PrintStream(s.getOutputStream());
            // 进行普通 IO 操作
            ps.println("您好，您收到了服务器的新年祝福！");
            // 关闭输出流，关闭 Socket
            ps.close();
            s.close();
        }
    }
}
```

下面的客户端程序也非常简单，它仅仅使用 Socket 建立与指定 IP 地址、指定端口的连接，并使用 Socket 获取输入流读取数据。

程序清单：codes\17\17.3\Client.java

```
public class Client
{
    public static void main(String[] args)
        throws IOException
    {
        Socket socket = new Socket("127.0.0.1" , 30000);    // ①
        // 将 Socket 对应的输入流包装成 BufferedReader
        BufferedReader br = new BufferedReader(
        new InputStreamReader(socket.getInputStream()));
        // 进行普通 IO 操作
        String line = br.readLine();
        System.out.println("来自服务器的数据：" + line);
        // 关闭输入流，关闭 Socket
        br.close();
        socket.close();
    }
}
```

上面程序中①号粗体字代码是使用 ServerSocket 和 Socket 建立网络连接的代码，接下来的粗体字代码是通过 Socket 获取输入流、输出流进行通信的代码。通过程序不难看出，一旦使用 ServerSocket、Socket 建立网络连接之后，程序通过网络通信与普通 IO 并没有太大的区别。

先运行程序中的 Server 类，将看到服务器一直处于等待状态，因为服务器使用了死循环来接收来自客户端的请求；再运行 Client 类，将看到程序输出：“来自服务器的数据：您好，您收到了服务器的新年祝福!”，这表明客户端和服务器端通信成功。

注意：

上面程序为了突出通过 ServerSocket 和 Socket 建立连接，并通过底层 IO 流进行通信的主题，程序没有进行异常处理，也没有使用 finally 块来关闭资源。

在实际应用中，程序可能不想让执行网络连接、读取服务器数据的进程一直阻塞，而是希望当网络连接、读取操作超过合理时间之后，系统自动认为该操作失败，这个合理时间就是超时时长。Socket 对象提供了一个 setSoTimeout(int timeout)方法来设置超时时长。如下代码片段所示。

```
Socket s = new Socket("127.0.0.1" , 30000);
//设置 10 秒之后即认为超时
s.setSoTimeout(10000);
```

为 Socket 对象指定了超时时长之后，如果在使用 Socket 进行读、写操作完成之前超出了该时间限制，那么这些方法就会抛出 SocketTimeoutException 异常，程序可以对该异常进行捕获，并进行适当处理。如下代码所示。

```
try
{
    // 使用 Scanner 来读取网络输入流中的数据
    Scanner scan = new Scanner(s.getInputStream())
    // 读取一行字符
    String line = scan.nextLine()
    ...
}
// 捕获 SocketTimeoutException 异常
catch(SocketTimeoutException ex)
{
    // 对异常进行处理
    ...
}
```

假设程序需要为 Socket 连接服务器时指定超时时长，即经过指定时间后，如果该 Socket 还未连接到远程服务器，则系统认为该 Socket 连接超时。但 Socket 的所有构造器里都没有提供指定超时时长的参数，所以程序应该先创建一个无连接的 Socket，再调用 Socket 的 connect()方法来连接远程服务器，而 connect()方法就可以接收一个超时时长参数。如下代码所示。

```
// 创建一个无连接的 Socket
Socket s = new Socket();
// 让该 Socket 连接到远程服务器，如果经过 10 秒还没有连接上，则认为连接超时
s.connect(new InetSocketAddress (host, port) ,10000);
```

17.3.4 加入多线程

前面 Server 和 Client 只是进行了简单的通信操作：服务器端接收到客户端连接之后，服务器端向客户端输出一个字符串，而客户端也只是读取服务器端的字符串后就退出了。实际应用中的客户端则可能需要和服务器端保持长时间通信，即服务器端需要不断地读取客户端数据，并向客户端写入数据；客户端也需要不断地读取服务器端数据，并向服务器端写入数据。

在使用传统 BufferedReader 的 readLine()方法读取数据时，在该方法成功返回之前，线程被阻塞，程序无法继续执行。考虑到这个原因，服务器端应该为每个 Socket 单独启动一个线程，每个线程负责与一个客户端进行通信。

客户端读取服务器端数据的线程同样会被阻塞，所以系统应该单独启动一个线程，该线程专门负责读取服务器端数据。

现在考虑实现一个命令行界面的 C/S 聊天室应用，服务器端应该包含多个线程，每个 Socket 对应一个线程，该线程负责读取 Socket 对应输入流的数据（从客户端发送过来的数据），并将读到的数据向每个 Socket 输出流发送一次（将一个客户端发送的数据“广播”给其他客户端），因此需要在服务器端使用 List 来保存所有的 Socket。

下面是服务器端的实现代码，程序为服务器端提供了两个类，一个是创建 ServerSocket 监听的主类，一个是负责处理每个 Socket 通信的线程类。

程序清单：codes\17\17.3\MultiThread\server\MyServer.java

```
public class MyServer
{
    // 定义保存所有 Socket 的 ArrayList，并将其包装为线程安全的
    public static List<Socket> socketList
        = Collections.synchronizedList(new ArrayList<>());
    public static void main(String[] args)
        throws IOException
    {
        ServerSocket ss = new ServerSocket(30000);
        while(true)
        {
            // 此行代码会阻塞，将一直等待别人的连接
            Socket s = ss.accept();
            socketList.add(s);
            // 每当客户端连接后启动一个 ServerThread 线程为该客户端服务
            new Thread(new ServerThread(s)).start();
        }
    }
}
```

上面程序实现了服务器端只负责接收客户端 Socket 的连接请求，每当客户端 Socket 连接到该 ServerSocket 之后，程序将对应 Socket 加入 socketList 集合中保存，并为该 Socket 启动一个线程，该线程负责处理该 Socket 所有的通信任务，如程序中 4 行粗体字代码所示。服务器端线程类的代码如下。

程序清单：codes\17\17.3\MultiThread\server\ServerThread.java

```
// 负责处理每个线程通信的线程类
public class ServerThread implements Runnable
{
    // 定义当前线程所处理的 Socket
    Socket s = null;
    // 该线程所处理的 Socket 对应的输入流
    BufferedReader br = null;
    public ServerThread(Socket s)
    throws IOException
    {
        this.s = s;
        // 初始化该 Socket 对应的输入流
```

```
            br = new BufferedReader(new InputStreamReader(s.getInputStream()));
        }
        public void run()
        {
            try
            {
                String content = null;
                // 采用循环不断地从 Socket 中读取客户端发送过来的数据
                while ((content = readFromClient()) != null)
                {
                    // 遍历 socketList 中的每个 Socket
                    // 将读到的内容向每个 Socket 发送一次
                    for (Socket s : MyServer.socketList)
                    {
                        PrintStream ps = new PrintStream(s.getOutputStream());
                        ps.println(content);
                    }
                }
            }
            catch (IOException e)
            {
                e.printStackTrace();
            }
        }
        // 定义读取客户端数据的方法
        private String readFromClient()
        {
            try
            {
                return br.readLine();
            }
            // 如果捕获到异常，则表明该 Socket 对应的客户端已经关闭
            catch (IOException e)
            {
                // 删除该 Socket
                MyServer.socketList.remove(s);      // ①
            }
            return null;
        }
    }
```

上面的服务器端线程类不断地读取客户端数据，程序使用 readFromClient()方法来读取客户端数据，如果读取数据过程中捕获到 IOException 异常，则表明该 Socket 对应的客户端 Socket 出现了问题（到底什么问题不用深究，反正不正常），程序就将该 Socket 从 socketList 集合中删除，如 readFromClient()方法中①号代码所示。

当服务器端线程读到客户端数据之后，程序遍历 socketList 集合，并将该数据向 socketList 集合中的每个 Socket 发送一次——该服务器端线程把从 Socket 中读到的数据向 socketList 集合中的每个 Socket 转发一次，如 run()线程执行体中的粗体字代码所示。

每个客户端应该包含两个线程，一个负责读取用户的键盘输入，并将用户输入的数据写入 Socket 对应的输出流中；一个负责读取 Socket 对应输入流中的数据（从服务器端发送过来的数据），并将这些数据打印输出。其中负责读取用户键盘输入的线程由 MyClient 负责，也就是由程序的主线程负责。客户端主程序代码如下。

程序清单：codes\17\17.3\MultiThread\client\MyClient.java

```
public class MyClient
{
    public static void main(String[] args)throws Exception
    {
        Socket s = new Socket("127.0.0.1" , 30000);
        // 客户端启动 ClientThread 线程不断地读取来自服务器的数据
        new Thread(new ClientThread(s)).start();   // ①
        // 获取该 Socket 对应的输出流
        PrintStream ps = new PrintStream(s.getOutputStream());
        String line = null;
```

```
            // 不断地读取键盘输入
            BufferedReader br = new BufferedReader(
                new InputStreamReader(System.in));
            while ((line = br.readLine()) != null)
            {
                // 将用户的键盘输入内容写入 Socket 对应的输出流
                ps.println(line);
            }
        }
    }
```

上面程序中获取键盘输入的代码在第 15 章中已有详细解释，此处不再赘述。当该线程读到用户键盘输入的内容后，将用户键盘输入的内容写入该 Socket 对应的输出流。

除此之外，当主线程使用 Socket 连接到服务器之后，启动了 ClientThread 来处理该线程的 Socket 通信，如程序中①号代码所示。ClientThread 线程负责读取 Socket 输入流中的内容，并将这些内容在控制台打印出来。

程序清单：codes\17\17.3\MultiThread\client\ClientThread.java

```
public class ClientThread implements Runnable
{
    // 该线程负责处理的 Socket
    private Socket s;
    // 该线程所处理的 Socket 对应的输入流
    BufferedReader br = null;
    public ClientThread(Socket s)
        throws IOException
    {
        this.s = s;
        br = new BufferedReader(
            new InputStreamReader(s.getInputStream()));
    }
    public void run()
    {
        try
        {
            String content = null;
            // 不断地读取 Socket 输入流中的内容，并将这些内容打印输出
            while ((content = br.readLine()) != null)
            {
                System.out.println(content);
            }
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

上面线程的功能也非常简单，它只是不断地获取 Socket 输入流中的内容，当获取到 Socket 输入流中的内容后，直接将这些内容打印在控制台，如上面程序中粗体字代码所示。

先运行上面程序中的 MyServer 类，该类运行后只是作为服务器，看不到任何输出。再运行多个 MyClient——相当于启动多个聊天室客户端登录该服务器，然后可以在任何一个客户端通过键盘输入一些内容后按回车键，即可在所有客户端（包括自己）的控制台上收到刚刚输入的内容，这就粗略地实现了一个 C/S 结构聊天室的功能。

➤➤ 17.3.5 记录用户信息

上面程序虽然已经完成了粗略的通信功能，每个客户端可以看到其他客户端发送的信息，但无法知道是哪个客户端发送的信息，这是因为服务器端从未记录过用户信息，当客户端使用 Socket 连接到服务器端之后，程序只是使用 socketList 集合保存了服务器端对应生成的 Socket，并没有保存该 Socket 关联的客户信息。

下面程序将考虑使用 Map 来保存用户状态信息，因为本程序将考虑实现私聊功能，也就是说，一个客户端可以将信息发送给另一个指定客户端。实际上，所有客户端只与服务器端连接，客户端之间并没有互相连接，也就是说，当一个客户端信息发送到服务器端之后，服务器端必须可以判断该信息到底是向所有用户发送，还是向指定用户发送，并需要知道向哪个用户发送。这里需要解决如下两个问题。

- 客户端发送来的信息必须有特殊的标识——让服务器端可以判断是公聊信息，还是私聊信息。
- 如果是私聊信息，客户端会发送该消息的目的用户（私聊对象）给服务器端，服务器端如何将该信息发送给该私聊对象。

为了解决第一个问题，可以让客户端在发送不同信息之前，先对这些信息进行适当处理，比如在内容前后添加一些特殊字符——这种特殊字符被称为协议字符。本例提供了一个 CrazyitProtocol 接口，该接口专门用于定义协议字符。

程序清单：codes\17\17.3\Senior\server\CrazyitProtocol.java

```
public interface CrazyitProtocol
{
    // 定义协议字符串的长度
    int PROTOCOL_LEN = 2;
    // 下面是一些协议字符串，服务器端和客户端交换的信息都应该在前、后添加这种特殊字符串
    String MSG_ROUND = "§γ";
    String USER_ROUND = "∏Σ";
    String LOGIN_SUCCESS = "1";
    String NAME_REP = "-1";
    String PRIVATE_ROUND = "★【";
    String SPLIT_SIGN = "※";
}
```

实际上，由于服务器端和客户端都需要使用这些协议字符串，所以程序需要在客户端和服务器端同时保留该接口对应的 class 文件。

为了解决第二个问题，可以考虑使用一个 Map 来保存聊天室所有用户和对应 Socket 之间的映射关系——这样服务器端就可以根据用户名来找到对应的 Socket。但实际上本程序并未这么做，程序仅仅是用 Map 保存了聊天室所有用户名和对应输出流之间的映射关系，因为服务器端只要获取该用户名对应的输出流即可。服务器端提供了一个 HashMap 的子类，该类不允许 value 重复，并提供了根据 value 获取 key，根据 value 删除 key 等方法。

程序清单：codes\17\17.3\Senior\server\CrazyitMap.java

```
// 通过组合 HashMap 对象来实现 CrazyitMap，CrazyitMap 要求 value 也不可重复
public class CrazyitMap<K,V>
{
    // 创建一个线程安全的 HashMap
    public Map<K ,V> map = Collections.synchronizedMap(new HashMap<K,V>());
    // 根据 value 来删除指定项
    public synchronized void removeByValue(Object value)
    {
        for (Object key : map.keySet())
        {
            if (map.get(key) == value)
            {
                map.remove(key);
                break;
            }
        }
    }
    // 获取所有 value 组成的 Set 集合
    public synchronized Set<V> valueSet()
    {
        Set<V> result = new HashSet<V>();
        // 将 map 中的所有 value 添加到 result 集合中
        map.forEach((key , value) -> result.add(value));
        return result;
    }
    // 根据 value 查找 key
```

```
    public synchronized K getKeyByValue(V val)
    {
        // 遍历所有 key 组成的集合
        for (K key : map.keySet())
        {
            // 如果指定 key 对应的 value 与被搜索的 value 相同，则返回对应的 key
            if (map.get(key) == val || map.get(key).equals(val))
            {
                return key;
            }
        }
        return null;
    }
    // 实现 put()方法，该方法不允许 value 重复
    public synchronized V put(K key,V value)
    {
        // 遍历所有 value 组成的集合
        for (V val : valueSet() )
        {
            // 如果某个 value 与试图放入集合的 value 相同
            // 则抛出一个 RuntimeException 异常
            if (val.equals(value)
                && val.hashCode()== value.hashCode())
            {
                throw new RuntimeException("MyMap 实例中不允许有重复 value!");
            }
        }
        return map.put(key , value);
    }
}
```

严格来讲，CrazyitMap 已经不是一个标准的 Map 结构了，但程序需要这样一个数据结构来保存用户名和对应输出流之间的映射关系，这样既可以通过用户名找到对应的输出流，也可以根据输出流找到对应的用户名。

服务器端的主类一样只是建立 ServerSocket 来监听来自客户端 Socket 的连接请求，但该程序增加了一些异常处理，可能看上去比上一节的程序稍微复杂一点。

程序清单：codes\17\17.3\Senior\server\Server.java

```
public class Server
{
    private static final int SERVER_PORT = 30000;
    // 使用 CrazyitMap 对象来保存每个客户名字和对应输出流之间的对应关系
    public static CrazyitMap<String , PrintStream> clients
        = new CrazyitMap<>();
    public void init()
    {
        try(
            // 建立监听的 ServerSocket
            ServerSocket ss = new ServerSocket(SERVER_PORT))
        {
            // 采用死循环来不断地接收来自客户端的请求
            while(true)
            {
                Socket socket = ss.accept();
                new ServerThread(socket).start();
            }
        }
        // 如果抛出异常
        catch (IOException ex)
        {
            System.out.println("服务器启动失败，是否端口"
                + SERVER_PORT + "已被占用？");
        }
    }
    public static void main(String[] args)
    {
        Server server = new Server();
```

```
            server.init();
        }
    }
```

该程序的关键代码依然只有三行，如程序中粗体字代码所示。它们依然是完成建立 ServerSocket，监听客户端 Socket 连接请求，并为已连接的 Socket 启动单独的线程。

服务器端线程类比上一节的程序要复杂一点，因为该线程类要分别处理公聊、私聊两类聊天信息。除此之外，还需要处理用户名是否重复的问题。服务器端线程类的代码如下。

程序清单：codes\17\17.3\Senior\server\ServerThread.java

```
public class ServerThread extends Thread
{
    private Socket socket;
    BufferedReader br = null;
    PrintStream ps = null;
    // 定义一个构造器，用于接收一个 Socket 来创建 ServerThread 线程
    public ServerThread(Socket socket)
    {
        this.socket = socket;
    }
    public void run()
    {
        try
        {
            // 获取该 Socket 对应的输入流
            br = new BufferedReader(new InputStreamReader(socket
                .getInputStream()));
            // 获取该 Socket 对应的输出流
            ps = new PrintStream(socket.getOutputStream());
            String line = null;
            while((line = br.readLine())!= null)
            {
                // 如果读到的行以 CrazyitProtocol.USER_ROUND 开始，并以其结束
                // 则可以确定读到的是用户登录的用户名
                if (line.startsWith(CrazyitProtocol.USER_ROUND)
                    && line.endsWith(CrazyitProtocol.USER_ROUND))
                {
                    // 得到真实消息
                    String userName = getRealMsg(line);
                    // 如果用户名重复
                    if (Server.clients.map.containsKey(userName))
                    {
                        System.out.println("重复");
                        ps.println(CrazyitProtocol.NAME_REP);
                    }
                    else
                    {
                        System.out.println("成功");
                        ps.println(CrazyitProtocol.LOGIN_SUCCESS);
                        Server.clients.put(userName , ps);
                    }
                }
                // 如果读到的行以 CrazyitProtocol.PRIVATE_ROUND 开始，并以其结束
                // 则可以确定是私聊信息，私聊信息只向特定的输出流发送
                else if (line.startsWith(CrazyitProtocol.PRIVATE_ROUND)
                    && line.endsWith(CrazyitProtocol.PRIVATE_ROUND))
                {
                    // 得到真实消息
                    String userAndMsg = getRealMsg(line);
                    // 以 SPLIT_SIGN 分割字符串，前半是私聊用户，后半是聊天信息
                    String user = userAndMsg.split(CrazyitProtocol.SPLIT_ SIGN) [0];
                    String msg = userAndMsg.split(CrazyitProtocol.SPLIT_ SIGN) [1];
                    // 获取私聊用户对应的输出流，并发送私聊信息
                    Server.clients.map.get(user).println(Server.clients
                        .getKeyByValue(ps) + "悄悄地对你说：" + msg);
                }
                // 公聊要向每个 Socket 发送
                else
```

```
                {
                    // 得到真实消息
                    String msg = getRealMsg(line);
                    // 遍历 clients 中的每个输出流
                    for (PrintStream clientPs : Server.clients.valueSet())
                    {
                        clientPs.println(Server.clients.getKeyByValue(ps)
                            + "说: " + msg);
                    }
                }
            }
        }
        // 捕获到异常后，表明该 Socket 对应的客户端已经出现了问题
        // 所以程序将其对应的输出流从 Map 中删除
        catch (IOException e)
        {
            Server.clients.removeByValue(ps);
            System.out.println(Server.clients.map.size());
            // 关闭网络、IO 资源
            try
            {
                if (br != null)
                {
                    br.close();
                }
                if (ps != null)
                {
                    ps.close();
                }
                if (socket != null)
                {
                    socket.close();
                }
            }
            catch (IOException ex)
            {
                ex.printStackTrace();
            }
        }
    }
    // 将读到的内容去掉前后的协议字符，恢复成真实数据
    private String getRealMsg(String line)
    {
        return line.substring(CrazyitProtocol.PROTOCOL_LEN
            , line.length() - CrazyitProtocol.PROTOCOL_LEN);
    }
}
```

上面程序比前一节的程序除增加了异常处理之外，主要增加了对读取数据的判断，如程序中两行粗体字代码所示。程序读取到客户端发送过来的内容之后，会根据该内容前后的协议字符串对该内容进行相应的处理。

客户端主类增加了让用户输入用户名的代码，并且不允许用户名重复。除此之外，还可以根据用户的键盘输入来判断用户是否想发送私聊信息。客户端主类的代码如下。

程序清单：codes\17\17.3\Senior\client\Client.java

```
public class Client
{
    private static final int SERVER_PORT = 30000;
    private Socket socket;
    private PrintStream ps;
    private BufferedReader brServer;
    private BufferedReader keyIn;
    public void init()
    {
        try
        {
            // 初始化代表键盘的输入流
```

```
            keyIn = new BufferedReader(
                new InputStreamReader(System.in));
            // 连接到服务器端
            socket = new Socket("127.0.0.1", SERVER_PORT);
            // 获取该 Socket 对应的输入流和输出流
            ps = new PrintStream(socket.getOutputStream());
            brServer = new BufferedReader(
                new InputStreamReader(socket.getInputStream()));
            String tip = "";
            // 采用循环不断地弹出对话框要求输入用户名
            while(true)
            {
                String userName = JOptionPane.showInputDialog(tip
                    + "输入用户名");    // ①
                // 在用户输入的用户名前后增加协议字符串后发送
                ps.println(CrazyitProtocol.USER_ROUND + userName
                    + CrazyitProtocol.USER_ROUND);
                // 读取服务器端的响应
                String result = brServer.readLine();
                // 如果用户名重复，则开始下次循环
                if (result.equals(CrazyitProtocol.NAME_REP))
                {
                    tip = "用户名重复！请重新";
                    continue;
                }
                // 如果服务器端返回登录成功，则结束循环
                if (result.equals(CrazyitProtocol.LOGIN_SUCCESS))
                {
                    break;
                }
            }
        }
        // 捕获到异常，关闭网络资源，并退出该程序
        catch (UnknownHostException ex)
        {
            System.out.println("找不到远程服务器，请确定服务器已经启动！");
            closeRs();
            System.exit(1);
        }
        catch (IOException ex)
        {
            System.out.println("网络异常！请重新登录！");
            closeRs();
            System.exit(1);
        }
        // 以该 Socket 对应的输入流启动 ClientThread 线程
        new ClientThread(brServer).start();
    }
    // 定义一个读取键盘输出，并向网络发送的方法
    private void readAndSend()
    {
        try
        {
            // 不断地读取键盘输入
            String line = null;
            while((line = keyIn.readLine()) != null)
            {
                // 如果发送的信息中有冒号，且以//开头，则认为想发送私聊信息
                if (line.indexOf(":") > 0 && line.startsWith("//"))
                {
                    line = line.substring(2);
                    ps.println(CrazyitProtocol.PRIVATE_ROUND +
                    line.split(":")[0] + CrazyitProtocol.SPLIT_SIGN
                        + line.split(":")[1] + CrazyitProtocol.PRIVATE_ROUND);
                }
                else
                {
                    ps.println(CrazyitProtocol.MSG_ROUND + line
                        + CrazyitProtocol.MSG_ROUND);
                }
```

```
            }
        }
        // 捕获到异常，关闭网络资源，并退出该程序
        catch (IOException ex)
        {
            System.out.println("网络通信异常！请重新登录！");
            closeRs();
            System.exit(1);
        }
    }
    // 关闭 Socket、输入流、输出流的方法
    private void closeRs()
    {
        try
        {
            if (keyIn != null)
            {
                ps.close();
            }
            if (brServer != null)
            {
                ps.close();
            }
            if (ps != null)
            {
                ps.close();
            }
            if (socket != null)
            {
                keyIn.close();
            }
        }
        catch (IOException ex)
        {
            ex.printStackTrace();
        }
    }
    public static void main(String[] args)
    {
        Client client = new Client();
        client.init();
        client.readAndSend();
    }
}
```

上面程序使用 JOptionPane 弹出一个输入对话框让用户输入用户名，如程序 init()方法中的①号粗体字代码所示。然后程序立即将用户输入的用户名发送给服务器端，服务器端会返回该用户名是否重复的提示，程序又立即读取服务器端提示，并根据服务器端提示判断是否需要继续让用户输入用户名。

与前一节的客户端主类程序相比，该程序还增加了对用户输入信息的判断——程序判断用户输入的内容是否以斜线（/）开头，并包含冒号（:），如果满足该特征，系统认为该用户想发送私聊信息，就会将冒号（:）之前的部分当成私聊用户名，冒号（:）之后的部分当成聊天信息，如 readAndSend()方法中粗体字代码所示。

本程序中客户端线程类几乎没有太大的改变，仅仅添加了异常处理部分的代码。

程序清单：codes\17\17.3\Senior\client\ClientThread.java

```
public class ClientThread extends Thread
{
    // 该客户端线程负责处理的输入流
    BufferedReader br = null;
    // 使用一个网络输入流来创建客户端线程
    public ClientThread(BufferedReader br)
    {
        this.br = br;
    }
    public void run()
    {
```

```
            try
            {
                String line = null;
                // 不断地输入流中读取数据，并将这些数据打印输出
                while((line = br.readLine())!= null)
                {
                    System.out.println(line);
                    /*
                    本例仅打印了从服务器端读到的内容。实际上，此处的情况可以更复杂：如
                    果希望客户端能看到聊天室的用户列表，则可以让服务器端在每次有用户登
                    录、用户退出时，将所有的用户列表信息都向客户端发送一遍。为了区分服
                    务器端发送的是聊天信息，还是用户列表，服务器端也应该在要发送的信息
                    前、后都添加一定的协议字符串，客户端则根据协议字符串的不同而进行不
                    同的处理！
                    更复杂的情况：
                    如果两端进行游戏，则还有可能发送游戏信息，例如两端进行五子棋游戏，
                    则需要发送下棋坐标信息等，服务器端同样在这些下棋坐标信息前、后添加
                    协议字符串后再发送，客户端就可以根据该信息知道对手的下棋坐标。
                    */
                }
            }
            catch (IOException ex)
            {
                ex.printStackTrace();
            }
            // 使用 finally 块来关闭该线程对应的输入流
            finally
            {
                try
                {
                    if (br != null)
                    {
                        br.close();
                    }
                }
                catch (IOException ex)
                {
                    ex.printStackTrace();
                }
            }
        }
    }
```

虽然上面程序非常简单，但正如程序注释中所指出的，如果服务器端可以返回更多丰富类型的数据，则该线程类的处理将会更复杂，那么该程序可以扩展到非常强大。

先运行上面的 Server 类，启动服务器；再多次运行 Client 类启动多个客户端，并输入不同的用户名，登录服务器后的聊天界面如图 17.5 所示。

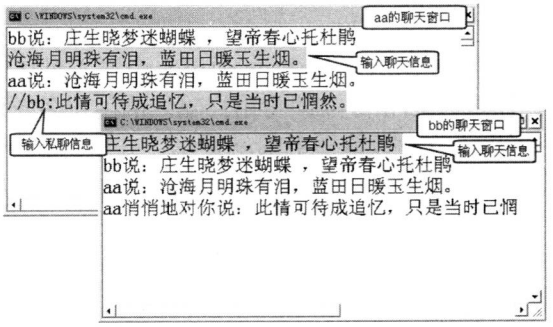


图 17.5　两个客户端的聊天界面

提示：

本程序没有提供 GUI 界面部分，直接使用 DOS 窗口进行聊天——因为增加 GUI 界面会让程序代码更多，从而引起读者的畏难心理。如果读者理解了本程序之后，相信读者一定乐意为该程序添加界面部分，因为整个程序的所有核心功能都已经实现了。不仅如此，读者完全可以在本程序的基础上扩展成一个仿 QQ 游戏大厅的网络程序——疯狂软件教育中心的很多学生都可以做到这一点。

17.3.6 半关闭的 Socket

前面介绍服务器端和客户端通信时，总是以行作为通信的最小数据单位，在每行内容的前后分别添加特殊的协议字符串，服务器端处理信息时也是逐行进行处理的。在另一些协议里，通信的数据单位可能是多行的，例如前面介绍的通过 URLConnection 来获取远程主机的数据，远程主机响应的内容就包含很多数据——在这种情况下，需要解决一个问题：Socket 的输出流如何表示输出数据已经结束？

在第 15 章介绍 IO 时提到，如果要表示输出已经结束，则可以通过关闭输出流来实现。但在网络通信中则不能通过关闭输出流来表示输出已经结束，因为当关闭输出流时，该输出流对应的 Socket 也将随之关闭，这样导致程序无法再从该 Socket 的输入流中读取数据了。

在这种情况下，Socket 提供了如下两个半关闭的方法，只关闭 Socket 的输入流或者输出流，用以表示输出数据已经发送完成。

- shutdownInput()：关闭该 Socket 的输入流，程序还可通过该 Socket 的输出流输出数据。
- shutdownOutput()：关闭该 Scoket 的输出流，程序还可通过该 Socket 的输入流读取数据。

当调用 shutdownInput()或 shutdownOutput()方法关闭 Socket 的输入流或输出流之后，该 Socket 处于“半关闭”状态，Socket 可通过 isInputShutdown()方法判断该 Socket 是否处于半读状态（read-half），通过 isOutputShutdown()方法判断该 Socket 是否处于半写状态（write-half）。

注意：

即使同一个 Socket 实例先后调用 shutdownInput()、shutdownOutput()方法，该 Socket 实例依然没有被关闭，只是该 Socket 既不能输出数据，也不能读取数据而已。

下面程序示范了半关闭方法的用法。在该程序中服务器端先向客户端发送多条数据，数据发送完成后，该 Socket 对象调用 shutdownOutput()方法来关闭输出流，表明数据发送结束——关闭输出流之后，依然可以从 Socket 中读取数据。

程序清单：codes\17\17.3\HalfClose\Server.java

```
public class Server
{
    public static void main(String[] args)
        throws Exception
    {
        ServerSocket ss = new ServerSocket(30000);
        Socket socket = ss.accept();
        PrintStream ps = new PrintStream(socket.getOutputStream());
        ps.println("服务器的第一行数据");
        ps.println("服务器的第二行数据");
        // 关闭 socket 的输出流，表明输出数据已经结束
        socket.shutdownOutput();
        // 下面语句将输出 false，表明 socket 还未关闭
        System.out.println(socket.isClosed());
        Scanner scan = new Scanner(socket.getInputStream());
        while (scan.hasNextLine())
        {
            System.out.println(scan.nextLine());
        }
        scan.close();
        socket.close();
        ss.close();
    }
}
```

上面程序中的第一行粗体字代码关闭了 Socket 的输出流之后，程序判断该 Socket 是否处于关闭状态，将可看到该代码输出 false。反之，如果将第一行粗体字代码换成 ps.close()——关闭输出流，将可看到第二行粗体字代码输出 true，这表明关闭输出流导致 Socket 也随之关闭。

本程序的客户端代码比较普通，只是先读取服务器端返回的数据，再向服务器端输出一些内容。客户端代码比较简单，故此处不再赘述，读者可参考 codes\17\17.3\HalfClose\Client.java 程序来查看该代码。

当调用 Socket 的 shutdownOutput()或 shutdownInput()方法关闭了输出流或输入流之后，该 Socket 无法再次打开输出流或输入流，因此这种做法通常不适合保持持久通信状态的交互式应用，只适用于一站式的通信协议，例如 HTTP 协议——客户端连接到服务器端后，开始发送请求数据，发送完成后无须再次发送数据，只需要读取服务器端响应数据即可，当读取响应完成后，该 Socket 连接也被关闭了。

17.3.7 使用 NIO 实现非阻塞 Socket 通信

从 JDK 1.4 开始，Java 提供了 NIO API 来开发高性能的网络服务器，前面介绍的网络通信程序是基于阻塞式 API 的——即当程序执行输入、输出操作后，在这些操作返回之前会一直阻塞该线程，所以服务器端必须为每个客户端都提供一个独立线程进行处理，当服务器端需要同时处理大量客户端时，这种做法会导致性能下降。使用 NIO API 则可以让服务器端使用一个或有限几个线程来同时处理连接到服务器端的所有客户端。

提示：
如果读者忘记了 NIO 里 Channel、Buffer、Charset 等 API 的概念和用法，可以再次阅读本书第 15 章关于新 IO 的内容。

Java 的 NIO 为非阻塞式 Socket 通信提供了如下几个特殊类。

- Selector：它是 SelectableChannel 对象的多路复用器，所有希望采用非阻塞方式进行通信的 Channel 都应该注册到 Selector 对象。可以通过调用此类的 open()静态方法来创建 Selector 实例，该方法将使用系统默认的 Selector 来返回新的 Selector。

Selector 可以同时监控多个 SelectableChannel 的 IO 状况，是非阻塞 IO 的核心。一个 Selector 实例有三个 SelectionKey 集合。

- 所有的 SelectionKey 集合：代表了注册在该 Selector 上的 Channel，这个集合可以通过 keys()方法返回。
- 被选择的 SelectionKey 集合：代表了所有可通过 select()方法获取的、需要进行 IO 处理的 Channel，这个集合可以通过 selectedKeys()返回。
- 被取消的 SelectionKey 集合：代表了所有被取消注册关系的 Channel，在下一次执行 select()方法时，这些 Channel 对应的 SelectionKey 会被彻底删除，程序通常无须直接访问该集合。

除此之外，Selector 还提供了一系列和 select()相关的方法，如下所示。

- int select()：监控所有注册的 Channel，当它们中间有需要处理的 IO 操作时，该方法返回，并将对应的 SelectionKey 加入被选择的 SelectionKey 集合中，该方法返回这些 Channel 的数量。
- int select(long timeout)：可以设置超时时长的 select()操作。
- int selectNow()：执行一个立即返回的 select()操作，相对于无参数的 select()方法而言，该方法不会阻塞线程。
- Selector wakeup()：使一个还未返回的 select()方法立刻返回。
- SelectableChannel：它代表可以支持非阻塞 IO 操作的 Channel 对象，它可被注册到 Selector 上，这种注册关系由 SelectionKey 实例表示。Selector 对象提供了一个 select()方法，该方法允许应用程序同时监控多个 IO Channel。

应用程序可调用 SelectableChannel 的 register()方法将其注册到指定 Selector 上，当该 Selector 上的某些 SelectableChannel 上有需要处理的 IO 操作时，程序可以调用 Selector 实例的 select()方法获取它们的数量，并可以通过 selectedKeys()方法返回它们对应的 SelectionKey 集合——通过该集合就可以获取所有需要进行 IO 处理的 SelectableChannel 集。

SelectableChannel 对象支持阻塞和非阻塞两种模式（所有的 Channel 默认都是阻塞模式），必须使用非阻塞模式才可以利用非阻塞 IO 操作。SelectableChannel 提供了如下两个方法来设置和返回该 Channel 的模式状态。

- SelectableChannel configureBlocking(boolean block)：设置是否采用阻塞模式。
- boolean isBlocking()：返回该 Channel 是否是阻塞模式。

不同的 SelectableChannel 所支持的操作不一样，例如 ServerSocketChannel 代表一个 ServerSocket，它就只支持 OP_ACCEPT 操作。SelectableChannel 提供了如下方法来返回它支持的所有操作。

- int validOps()：返回一个整数值，表示这个 Channel 所支持的 IO 操作。

提示：

在 SelectionKey 中，用静态常量定义了 4 种 IO 操作：OP_READ（1）、OP_WRITE（4）、OP_CONNECT（8）、OP_ACCEPT（16），这个值任意 2 个、3 个、4 个进行按位或的结果和相加的结果相等，而且它们任意 2 个、3 个、4 个相加的结果总是互不相同，所以系统可以根据 validOps()方法的返回值确定该 SelectableChannel 支持的操作。例如返回 5，即可知道它支持读（1）和写（4）。

除此之外，SelectableChannel 还提供了如下几个方法来获取它的注册状态。

- boolean isRegistered()：返回该 Channel 是否已注册在一个或多个 Selector 上。
- SelectionKey keyFor(Selector sel)：返回该 Channel 和 sel Selector 之间的注册关系，如果不存在注册关系，则返回 null。
- SelectionKey：该对象代表 SelectableChannel 和 Selector 之间的注册关系。
- ServerSocketChannel：支持非阻塞操作，对应于 java.net.ServerSocket 这个类，只支持 OP_ACCEPT 操作。该类也提供了 accept()方法，功能相当于 ServerSocket 提供的 accept()方法。
- SocketChannel：支持非阻塞操作，对应于 java.net.Socket 这个类，支持 OP_CONNECT、OP_READ 和 OP_WRITE 操作。这个类还实现了 ByteChannel 接口、ScatteringByteChannel 接口和 GatheringByteChannel 接口，所以可以直接通过 SocketChannel 来读写 ByteBuffer 对象。

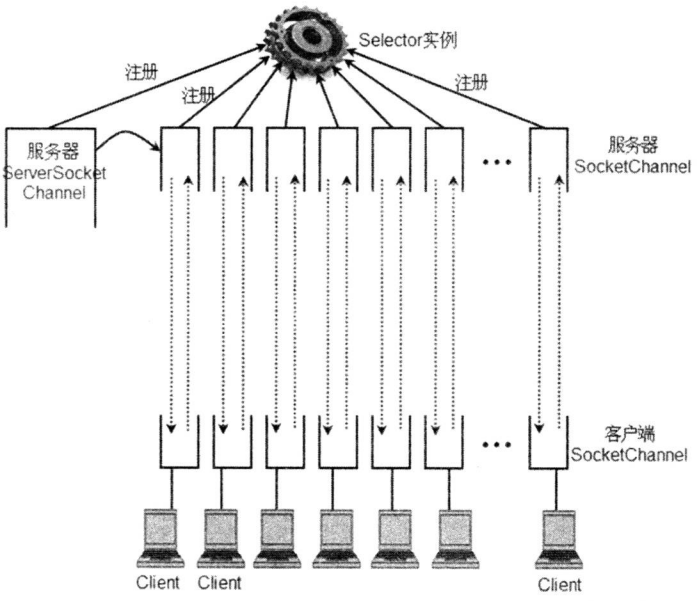


图 17.6　NIO 的非阻塞式服务器示意图

图 17.6 显示了 NIO 的非阻塞式服务器示意图。

从图 17.6 中可以看出，服务器上的所有 Channel（包括 ServerSocketChannel 和 SocketChannel）都需要向 Selector 注册，而该 Selector 则负责监视这些 Socket 的 IO 状态，当其中任意一个或多个 Channel 具有可用的 IO 操作时，该 Selector 的 select()方法将会返回大于 0 的整数，该整数值就表示该 Selector 上有多少个 Channel 具有可用的 IO 操作，并提供了 selectedKeys()方法来返回这些 Channel 对应的 SelectionKey 集合。正是通过 Selector，使得服务器端只需要不断地调用 Selector 实例的 select()方法，即可知道当前的所有 Channel 是否有需要处理的 IO 操作。

提示：

当 Selector 上注册的所有 Channel 都没有需要处理的 IO 操作时，select()方法将被阻塞，调用该方法的线程被阻塞。

本示例程序使用 NIO 实现了多人聊天室的功能，服务器端使用循环不断地获取 Selector 的 select()方法返回值，当该返回值大于 0 时就处理该 Selector 上被选择的 SelectionKey 所对应的 Channel。

服务器端需要使用 ServerSocket Channel 来监听客户端的连接请求，Java 对该类的设计比较难用：它不像 ServerSocket 可以直接指定监听某个端口；而且不能使用已有的 ServerSocket 的 getChannel()方法

来获取 ServerSocket Channel 实例。程序必须先调用它的 open()静态方法返回一个 ServerSocketChannel 实例，再使用它的 bind()方法指定它在某个端口监听。创建一个可用的 Server SocketChannel 需要采用如下代码片段：

```
// 通过 open 方法来打开一个未绑定的 ServerSocketChannel 实例
ServerSocketChannel server = ServerSocketChannel.open();
InetSocketAddress isa = new InetSocketAddress("127.0.0.1", 30000);
// 将该 ServerSocketChannel 绑定到指定 IP 地址
server.bind(isa);
```

提示：

在 Java 7 以前，ServerSocketChannel 的设计更糟糕——要让 ServerSocketChannel 监听指定端口，必须先调用它的 socket()方法获取它关联的 ServerSocket 对象，再调用 ServerSocket 的 bind()方法去监听指定端口。Java 7 为 ServerSocketChannel 新增了 bind()方法，因此稍微简单了一些。

如果需要使用非阻塞方式来处理该 ServerSocketChannel，还应该设置它的非阻塞模式，并将其注册到指定的 Selector。如下代码片段所示。

```
// 设置 ServerSocket 以非阻塞方式工作
server.configureBlocking(false);
// 将 server 注册到指定的 Selector 对象
server.register(selector, SelectionKey.OP_ACCEPT);
```

经过上面步骤后，该 ServerSocketChannel 可以接收客户端的连接请求，还需要调用 Selector 的 select()方法来监听所有 Channel 上的 IO 操作。

程序清单：codes\17\17.3\NoBlock\NServer.java

```
public class NServer
{
    // 用于检测所有 Channel 状态的 Selector
    private Selector selector = null;
    static final int PORT = 30000;
    // 定义实现编码、解码的字符集对象
    private Charset charset = Charset.forName("UTF-8");
    public void init()throws IOException
    {
        selector = Selector.open();
        // 通过 open 方法来打开一个未绑定的 ServerSocketChannel 实例
        ServerSocketChannel server = ServerSocketChannel.open();
        InetSocketAddress isa = new InetSocketAddress("127.0.0.1", PORT);
        // 将该 ServerSocketChannel 绑定到指定 IP 地址
        server.bind(isa);
        // 设置 ServerSocket 以非阻塞方式工作
        server.configureBlocking(false);
        // 将 server 注册到指定的 Selector 对象
        server.register(selector, SelectionKey.OP_ACCEPT);
        while (selector.select() > 0)
        {
            // 依次处理 selector 上的每个已选择的 SelectionKey
            for (SelectionKey sk : selector.selectedKeys())
            {
                // 从 selector 上的已选择 Key 集中删除正在处理的 SelectionKey
                selector.selectedKeys().remove(sk);      // ①
                // 如果 sk 对应的 Channel 包含客户端的连接请求
                if (sk.isAcceptable())        // ②
                {
                    // 调用 accept 方法接受连接，产生服务器端的 SocketChannel
                    SocketChannel sc = server.accept();
                    // 设置采用非阻塞模式
                    sc.configureBlocking(false);
                    //将该 SocketChannel 也注册到 selector
                    sc.register(selector, SelectionKey.OP_READ);
                    // 将 sk 对应的 Channel 设置成准备接收其他请求
                    sk.interestOps(SelectionKey.OP_ACCEPT);
```

```
                }
                // 如果 sk 对应的 Channel 有数据需要读取
                if (sk.isReadable())     // ③
                {
                    // 获取该 SelectionKey 对应的 Channel，该 Channel 中有可读的数据
                    SocketChannel sc = (SocketChannel)sk.channel();
                    // 定义准备执行读取数据的 ByteBuffer
                    ByteBuffer buff = ByteBuffer.allocate(1024);
                    String content = "";
                    // 开始读取数据
                    try
                    {
                        while(sc.read(buff) > 0)
                        {
                            buff.flip();
                            content += charset.decode(buff);
                        }
                        // 打印从该 sk 对应的 Channel 里读取到的数据
                        System.out.println("读取的数据：" + content);
                        // 将 sk 对应的 Channel 设置成准备下一次读取
                        sk.interestOps(SelectionKey.OP_READ);
                    }
                    // 如果捕获到该 sk 对应的 Channel 出现了异常，即表明该 Channel
                    // 对应的 Client 出现了问题，所以从 Selector 中取消 sk 的注册
                    catch (IOException ex)
                    {
                        // 从 Selector 中删除指定的 SelectionKey
                        sk.cancel();
                        if (sk.channel() != null)
                        {
                            sk.channel().close();
                        }
                    }
                    // 如果 content 的长度大于 0，即聊天信息不为空
                    if (content.length() > 0)
                    {
                        // 遍历该 selector 里注册的所有 SelectionKey
                        for (SelectionKey key : selector.keys())
                        {
                            // 获取该 key 对应的 Channel
                            Channel targetChannel = key.channel();
                            // 如果该 Channel 是 SocketChannel 对象
                            if (targetChannel instanceof SocketChannel)
                            {
                                // 将读到的内容写入该 Channel 中
                                SocketChannel dest = (SocketChannel)targetChannel;
                                dest.write(charset.encode(content));
                            }
                        }
                    }
                }
            }
        }
    }
    public static void main(String[] args)
        throws IOException
    {
        new NServer().init();
    }
}
```

中掌工匠

上面程序启动时即建立了一个可监听连接请求的 ServerSocketChannel，并将该 Channel 注册到指定的 Selector，接着程序直接采用循环不断地监控 Selector 对象的 select()方法返回值，当该返回值大于 0 时，处理该 Selector 上所有被选择的 SelectionKey。

开始处理指定的 SelectionKey 之后，立即从该 Selector 上被选择的 SelectionKey 集合中删除该 SelectionKey，如程序中①号代码所示。

服务器端的 Selector 仅需要监听两种操作：连接和读数据，所以程序中分别处理了这两种操作，如

程序中②和③代码所示——处理连接操作时，系统只需将连接完成后产生的 SocketChannel 注册到指定的 Selector 对象即可；处理读数据操作时，系统先从该 Socket 中读取数据，再将数据写入 Selector 上注册的所有 Channel 中。

本示例程序的客户端程序需要两个线程，一个线程负责读取用户的键盘输入，并将输入的内容写入 SocketChannel 中；另一个线程则不断地查询 Selector 对象的 select()方法的返回值，如果该方法的返回值大于 0，那就说明程序需要对相应的 Channel 执行 IO 处理。

提示：

使用 NIO 来实现服务器端时，无须使用 List 来保存服务器端所有的 SocketChannel，因为所有的 SocketChannel 都已注册到指定的 Selector 对象。除此之外，当客户端关闭时会导致服务器端对应的 Channel 也抛出异常，而且本程序只有一个线程，如果该异常得不到处理将会导致整个服务器端退出，所以程序捕获了这种异常，并在处理异常时从 Selector 中删除异常 Channel 的注册，如程序中粗体字代码所示。

程序清单：codes\17\17.3\NoBlock\NClient.java

```
public class NClient
{
    // 定义检测 SocketChannel 的 Selector 对象
    private Selector selector = null;
    static final int PORT = 30000;
    // 定义处理编码和解码的字符集
    private Charset charset = Charset.forName("UTF-8");
    // 客户端 SocketChannel
    private SocketChannel sc = null;
    public void init()throws IOException
    {
        selector = Selector.open();
        InetSocketAddress isa = new InetSocketAddress("127.0.0.1", PORT);
        // 调用 open 静态方法创建连接到指定主机的 SocketChannel
        sc = SocketChannel.open(isa);
        // 设置该 sc 以非阻塞方式工作
        sc.configureBlocking(false);
        // 将 SocketChannel 对象注册到指定的 Selector
        sc.register(selector, SelectionKey.OP_READ);
        // 启动读取服务器端数据的线程
        new ClientThread().start();
        // 创建键盘输入流
        Scanner scan = new Scanner(System.in);
        while (scan.hasNextLine())
        {
            // 读取键盘输入
            String line = scan.nextLine();
            // 将键盘输入的内容输出到 SocketChannel 中
            sc.write(charset.encode(line));
        }
    }
    // 定义读取服务器端数据的线程
    private class ClientThread extends Thread
    {
        public void run()
        {
            try
            {
                while (selector.select() > 0)    // ①
                {
                    // 遍历每个有可用 IO 操作的 Channel 对应的 SelectionKey
                    for (SelectionKey sk : selector.selectedKeys())
                    {
                        // 删除正在处理的 SelectionKey
                        selector.selectedKeys().remove(sk);
                        // 如果该 SelectionKey 对应的 Channel 中有可读的数据
                        if (sk.isReadable())
                        {
```

```
                        // 使用 NIO 读取 Channel 中的数据
                        SocketChannel sc = (SocketChannel)sk.channel();
                        ByteBuffer buff = ByteBuffer.allocate(1024);
                        String content = "";
                        while(sc.read(buff) > 0)
                        {
                            buff.flip();
                            content += charset.decode(buff);
                        }
                        // 打印输出读取的内容
                        System.out.println("聊天信息: " + content);
                        // 为下一次读取做准备
                        sk.interestOps(SelectionKey.OP_READ);
                    }
                }
            }
        }
        catch (IOException ex)
        {
            ex.printStackTrace();
        }
    }
}
public static void main(String[] args)
    throws IOException
{
    new NClient().init();
}
}
```

相比之下，客户端程序比服务器端程序要简单多了，客户端只有一个 SocketChannel，将该 SocketChannel 注册到指定的 Selector 后，程序启动另一个线程来监听该 Selector 即可。如果程序监听到该 Selector 的 select()方法返回值大于 0（如上面程序中①号粗体字代码所示），就表明该 Selector 上有需要进行 IO 处理的 Channel，接着程序取出该 Channel，并使用 NIO 读取该 Channel 中的数据，如上面程序中粗体字代码段所示。

17.3.8 使用 Java 7 的 AIO 实现非阻塞通信

Java 7 的 NIO.2 提供了异步 Channel 支持，这种异步 Channel 可以提供更高效的 IO，这种基于异步 Channel 的 IO 机制也被称为异步 IO（Asynchronous IO）。

提示：

如果按 POSIX 的标准来划分 IO，可以把 IO 分为两类：同步 IO 和异步 IO。对于 IO 操作可以分成两步：①程序发出 IO 请求；②完成实际的 IO 操作。前面两节所介绍的阻塞 IO、非阻塞 IO 都是针对第一步来划分的，如果发出 IO 请求会阻塞线程，就是阻塞 IO；如果发出 IO 请求没有阻塞线程，就是非阻塞 IO；但同步 IO 与异步 IO 的区别在第二步——如果实际的 IO 操作由操作系统完成，再将结果返回给应用程序，这就是异步 IO；如果实际的 IO 需要应用程序本身去执行，会阻塞线程，那就是同步 IO。前面介绍的传统 IO、基于 Channel 的非阻塞 IO 其实都是同步 IO。

NIO.2 提供了一系列以 Asynchronous 开头的 Channel 接口和类，图 17.7 显示了 AIO 的接口和实现类。

从图 17.7 可以看出，NIO.2 为 AIO 提供了两个接口和三个实现类，其中 AsynchronousSocketChannel、AsynchronousServerSocketChannel 是支持 TCP 通信的异步 Channel，这也是本节要重点介绍的两个实现类。

AsynchronousServerSocketChannel 是一个负责监听的 Channel，与 ServerSocketChannel 相似，创建可用的 AsynchronousServerSocketChannel 需要如下两步。

1. 调用它的 open()静态方法创建一个未监听端口的 AsynchronousServerSocketChannel。
2. 调用 AsynchronousServerSocketChannel 的 bind()方法指定该 Channel 在指定地址、指定端口监听。

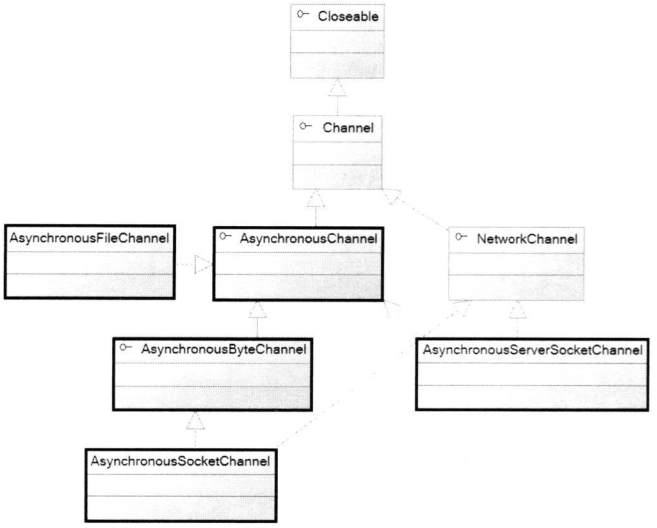


图 17.7　AIO 的接口和实现类

AsynchronousServerSocketChannel 的 open()方法有以下两个版本。

- open()：创建一个默认的 AsynchronousServerSocketChannel。
- open(AsynchronousChannelGroup group)：使用指定的 AsynchronousChannelGroup 来创建 AsynchronousServerSocketChannel。

上面方法中的 AsynchronousChannelGroup 是异步 Channel 的分组管理器，它可以实现资源共享。创建 AsynchronousChannelGroup 时需要传入一个 ExecutorService，也就是说，它会绑定一个线程池，该线程池负责两个任务：处理 IO 事件和触发 CompletionHandler。

提示：

AIO 的 AsynchronousServerSocketChannel、AsynchronousSocketChannel 都允许使用线程池进行管理，因此创建 AsynchronousSocketChannel 时也可以传入 AsynchronousChannelGroup 对象进行分组管理。

直接创建 AsynchronousServerSocketChannel 的代码片段如下：

```
// 以指定线程池来创建一个 AsynchronousServerSocketChannel
serverChannel = AsynchronousServerSocketChannel
    .open().bind(new InetSocketAddress(PORT));
```

使用 AsynchronousChannelGroup 创建 AsynchronousServerSocketChannel 的代码片段如下：

```
// 创建一个线程池
ExecutorService executor = Executors.newFixedThreadPool(80);
// 以指定线程池来创建一个 AsynchronousChannelGroup
AsynchronousChannelGroup channelGroup = AsynchronousChannelGroup
    .withThreadPool(executor);
// 以指定线程池来创建一个 AsynchronousServerSocketChannel
serverChannel = AsynchronousServerSocketChannel
    .open(channelGroup)
    .bind(new InetSocketAddress(PORT));
```

AsynchronousServerSocketChannel 创建成功之后，接下来可调用它的 accept()方法来接受来自客户端的连接，由于异步 IO 的实际 IO 操作是交给操作系统来完成的，因此程序并不清楚异步 IO 操作什么时候完成——也就是说，程序调用 AsynchronousServerSocketChannel 的 accept()方法之后，当前线程不会阻塞，而程序也不知道 accept()方法什么时候会接收到客户端的请求。为了解决这个异步问题，AIO 为 accept()方法提供了如下两个版本。

- Future<AsynchronousSocketChannel> accept()：接受客户端的请求。如果程序需要获得连接成功后返回的 AsynchronousSocketChannel，则应该调用该方法返回的 Future 对象的 get()方法——但

get()方法会阻塞线程，因此这种方式依然会阻塞当前线程。

- <A> void accept(A attachment, CompletionHandler<AsynchronousSocketChannel,? super A> handler)：接受来自客户端的请求，连接成功或连接失败都会触发 CompletionHandler 对象里相应的方法。其中 AsynchronousSocketChannel 就代表连接成功后返回的 AsynchronousSocketChannel。

CompletionHandler 是一个接口，该接口中定义了如下两个方法。

- completed(V result, A attachment)：当 IO 操作完成时触发该方法。该方法的第一个参数代表 IO 操作所返回的对象；第二个参数代表发起 IO 操作时传入的附加参数。
- failed(Throwable exc, A attachment)：当 IO 操作失败时触发该方法。该方法的第一个参数代表 IO 操作失败引发的异常或错误；第二个参数代表发起 IO 操作时传入的附加参数。

提示：

如果读者学习过疯狂 Java 体系的《疯狂 Ajax 讲义》，那么对 Ajax 技术应该有一定印象。Ajax 的关键在于异步请求：浏览器使用 JavaScript 发送异步请求——但异步请求的响应何时到来，程序无从知晓，因此程序会使用监听器来监听服务器端响应的到来。类似的，异步 Channel 发起 IO 操作后，IO 操作由操作系统执行，IO 操作何时完成，程序无从知晓，因此程序使用 CompletionHandler 对象来监听 IO 操作的完成。实际上，不仅 AsynchronousServerSocketChannel 的 accept()方法可以接受 CompletionHandler 监听器；AsynchronousSocketChannel 的 read()、write()方法都有两个版本，其中一个版本需要接受 CompletionHandler 监听器。

通过上面介绍不难看出，使用 AsynchronousServerSocketChannel 只要三步。

1. 调用 open()静态方法创建 AsynchronousServerSocketChannel。
2. 调用 AsynchronousServerSocketChannel 的 bind()方法让它在指定 IP 地址、端口监听。
3. 调用 AsynchronousServerSocketChannel 的 accept()方法接受连接请求。

下面使用最简单、最少的步骤来实现一个基于 AsynchronousServerSocketChannel 的服务器端。

程序清单：codes\17\17.3\SimpleAIO\SimpleAIOServer.java

```
public class SimpleAIOServer
{
    static final int PORT = 30000;
    public static void main(String[] args)
        throws Exception
    {
        try(
            // ①创建 AsynchronousServerSocketChannel 对象
            AsynchronousServerSocketChannel serverChannel =
                AsynchronousServerSocketChannel.open())
        {
            // ②指定在指定地址、端口监听
            serverChannel.bind(new InetSocketAddress(PORT));
            while (true)
            {
                // ③采用循环接受来自客户端的连接
                Future<AsynchronousSocketChannel> future
                    = serverChannel.accept();
                // 获取连接完成后返回的 AsynchronousSocketChannel
                AsynchronousSocketChannel socketChannel = future.get();
                // 执行输出
                socketChannel.write(ByteBuffer.wrap("欢迎你来到 AIO 的世界！"
                    .getBytes("UTF-8"))).get();
            }
        }
    }
}
```

上面程序中①②③号代码就代表了使用 AsynchronousServerSocketChannel 的三个基本步骤，由于该程序力求简单，因此程序并未使用 CompletionHandler 监听器。当程序接收到来自客户端的连接之后，

服务器端产生了一个与客户端对应的 AsynchronousSocketChannel，它就可以执行实际的 IO 操作了。

上面程序中粗体字代码是使用 AsynchronousSocketChannel 写入数据的代码，下面详细介绍该类的功能和用法。

AsynchronousSocketChannel 的用法也可分为三步。

① 调用 open()静态方法创建 AsynchronousSocketChannel。调用 open()方法时同样可指定一个 AsynchronousChannelGroup 作为分组管理器。

② 调用 AsynchronousSocketChannel 的 connect()方法连接到指定 IP 地址、指定端口的服务器。

③ 调用 AsynchronousSocketChannel 的 read()、write()方法进行读写。

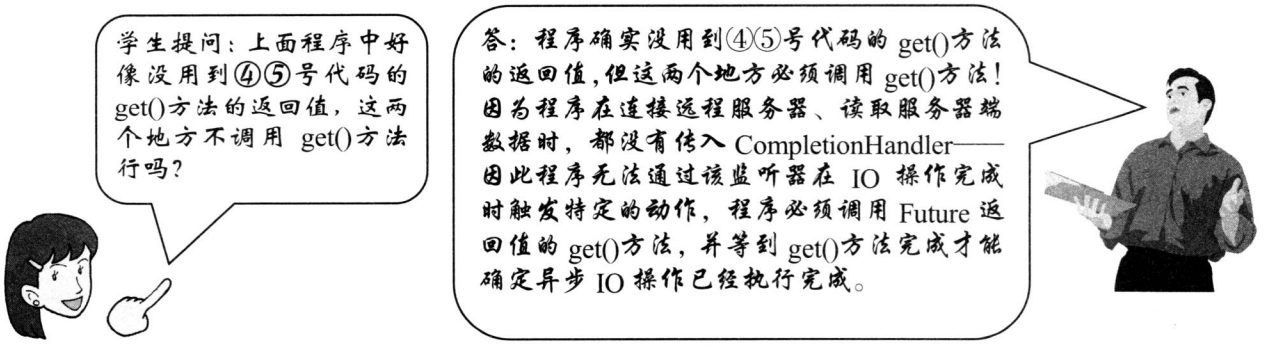
AsynchronousSocketChannel 的 connect()、read()、write()方法都有两个版本：一个返回 Future 对象的版本，一个需要传入 CompletionHandler 参数的版本。对于返回 Future 对象的版本，必须等到 Future 对象的 get()方法返回时 IO 操作才真正完成；对于需要传入 CompletionHandler 参数的版本，则可通过 CompletionHandler 在 IO 操作完成时触发相应的方法。

下面先用返回 Future 对象的 read()方法来读取服务器端响应数据。

程序清单：codes\17\17.3\SimpleAIO\SimpleAIOClient.java

```
public class SimpleAIOClient
{
    static final int PORT = 30000;
    public static void main(String[] args)
        throws Exception
    {
        // 用于读取数据的 ByteBuffer
        ByteBuffer buff = ByteBuffer.allocate(1024);
        Charset utf = Charset.forName("utf-8");
        try(
            // ①创建 AsynchronousSocketChannel 对象
            AsynchronousSocketChannel clientChannel
                = AsynchronousSocketChannel.open())
        {
            // ②连接远程服务器
            clientChannel.connect(new InetSocketAddress("127.0.0.1"
                , PORT)).get();    // ④
            buff.clear();
            // ③从 clientChannel 中读取数据
            clientChannel.read(buff).get();    // ⑤
            buff.flip();
            // 将 buff 中的内容转换为字符串
            String content = utf.decode(buff).toString();
            System.out.println("服务器信息: " + content);
        }
    }
}
```

上面程序中①②③号代码就代表了使用 AsynchronousSocketChannel 的三个基本步骤，当程序获得连接好的 AsynchronousSocketChannel 之后，就可通过它来执行实际的 IO 操作了。



先运行上面程序的服务器端，再运行客户端，将可以看到每个客户端都可以接收到来自于服务器端的欢迎信息。

上面基于 AIO 的应用程序十分简单，还没有充分利用 Java AIO 的优势，如果要充分挖掘 Java AIO 的优势，则应该考虑使用线程池来管理异步 Channel，并使用 CompletionHandler 来监听异步 IO 操作。

下面程序用于开发一个更完善的 AIO 多人聊天工具。服务器端程序代码如下。

程序清单：codes\17\17.3\AIO\AIOServer.java

```
public class AIOServer
{
    static final int PORT = 30000;
    final static String UTF_8 = "utf-8";
    static List<AsynchronousSocketChannel> channelList
        = new ArrayList<>();
    public void startListen() throws InterruptedException,
        Exception
    {
        // 创建一个线程池
        ExecutorService executor = Executors.newFixedThreadPool(20);
        // 以指定线程池来创建一个 AsynchronousChannelGroup
        AsynchronousChannelGroup channelGroup = AsynchronousChannelGroup
            .withThreadPool(executor);
        // 以指定线程池来创建一个 AsynchronousServerSocketChannel
        AsynchronousServerSocketChannel serverChannel
            = AsynchronousServerSocketChannel.open(channelGroup)
            // 指定监听本机的 PORT 端口
            .bind(new InetSocketAddress(PORT));
        // 使用 CompletionHandler 接收来自客户端的连接请求
        serverChannel.accept(null, new AcceptHandler(serverChannel)); // ①
    }
    public static void main(String[] args)
        throws Exception
    {
        AIOServer server = new AIOServer();
        server.startListen();
    }
}
// 实现自己的 CompletionHandler 类
class AcceptHandler implements
    CompletionHandler<AsynchronousSocketChannel, Object>
{
    private AsynchronousServerSocketChannel serverChannel;
    public AcceptHandler(AsynchronousServerSocketChannel sc)
    {
        this.serverChannel = sc;
    }
    // 定义一个 ByteBuffer 准备读取数据
    ByteBuffer buff = ByteBuffer.allocate(1024);
    // 当实际 IO 操作完成时触发该方法
    @Override
    public void completed(final AsynchronousSocketChannel sc
        , Object attachment)
    {
        // 记录新连接进来的 Channel
        AIOServer.channelList.add(sc);
        // 准备接收客户端的下一次连接
        serverChannel.accept(null , this);
        sc.read(buff , null
            , new CompletionHandler<Integer,Object>()  // ②
        {
            @Override
            public void completed(Integer result
                , Object attachment)
            {
                buff.flip();
                // 将 buff 中的内容转换为字符串
                String content = StandardCharsets.UTF_8
                    .decode(buff).toString();
                // 遍历每个 Channel，将收到的信息写入各 Channel 中
                for(AsynchronousSocketChannel c : AIOServer.channelList)
```

```
            {
                try
                {
                    c.write(ByteBuffer.wrap(content.getBytes(
                        AIOServer.UTF_8))).get();
                }
                catch (Exception ex)
                {
                    ex.printStackTrace();
                }
            }
            buff.clear();
            // 读取下一次数据
            sc.read(buff , null , this);
        }
        @Override
        public void failed(Throwable ex, Object attachment)
        {
            System.out.println("读取数据失败: " + ex);
            // 从该 Channel 中读取数据失败，就将该 Channel 删除
            AIOServer.channelList.remove(sc);
        }
    });
}
@Override
public void failed(Throwable ex, Object attachment)
{
    System.out.println("连接失败: " + ex);
}
}
```

上面程序与前一个服务器端程序的编程步骤大致相似，但这个程序使用了 CompletionHandler 监听来自客户端的连接，如程序中①号粗体字代码所示；当连接成功后，系统会自动触发该监听器的 completed() 方法——在该方法中，程序再次使用了 CompletionHandler 去读取来自客户端的数据，如程序中②号粗体字代码所示。这个程序一共用到了两个 CompletionHandler，这两个 Handler 类也是该程序的关键。

本程序的客户端提供一个简单的 GUI 界面，允许用户通过该 GUI 界面向服务器端发送信息，并显示其他用户的聊天信息。客户程序代码如下。

程序清单：codes\17\17.3\AIO\AIOClient.java

```
public class AIOClient
{
    final static String UTF_8 = "utf-8";
    final static int PORT = 30000;
    // 与服务器端通信的异步 Channel
    AsynchronousSocketChannel clientChannel;
    JFrame mainWin = new JFrame("多人聊天");
    JTextArea jta = new JTextArea(16 , 48);
    JTextField jtf = new JTextField(40);
    JButton sendBn = new JButton("发送");
    public void init()
    {
        mainWin.setLayout(new BorderLayout());
        jta.setEditable(false);
        mainWin.add(new JScrollPane(jta), BorderLayout.CENTER);
        JPanel jp = new JPanel();
        jp.add(jtf);
        jp.add(sendBn);
        // 发送消息的 Action，Action 是 ActionListener 的子接口
        Action sendAction = new AbstractAction()
        {
            public void actionPerformed(ActionEvent e)
            {
                String content = jtf.getText();
                if (content.trim().length() > 0)
                {
                    try
                    {
```

```
                // 将 content 内容写入 Channel 中
                clientChannel.write(ByteBuffer.wrap(content
                    .trim().getBytes(UTF_8))).get();     // ①
            }
            catch (Exception ex)
            {
                ex.printStackTrace();
            }
        }
        // 清空输入框
        jtf.setText("");
    }
};
sendBn.addActionListener(sendAction);
// 将"Ctrl+Enter"键和"send"关联
jtf.getInputMap().put(KeyStroke.getKeyStroke('\n'
    , java.awt.event.InputEvent.CTRL_DOWN_MASK) , "send");
// 将"send"和 sendAction 关联
jtf.getActionMap().put("send", sendAction);
mainWin.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
mainWin.add(jp , BorderLayout.SOUTH);
mainWin.pack();
mainWin.setVisible(true);
}
public void connect()
    throws Exception
{
    // 定义一个 ByteBuffer 准备读取数据
    final ByteBuffer buff = ByteBuffer.allocate(1024);
    // 创建一个线程池
    ExecutorService executor = Executors.newFixedThreadPool(80);
    // 以指定线程池来创建一个 AsynchronousChannelGroup
    AsynchronousChannelGroup channelGroup =
        AsynchronousChannelGroup.withThreadPool(executor);
    // 以 channelGroup 作为组管理器来创建 AsynchronousSocketChannel
    clientChannel = AsynchronousSocketChannel.open(channelGroup);
    // 让 AsynchronousSocketChannel 连接到指定 IP 地址、指定端口
    clientChannel.connect(new InetSocketAddress("127.0.0.1"
        , PORT)).get();
    jta.append("---与服务器连接成功---\n");
    buff.clear();
    clientChannel.read(buff, null
        , new CompletionHandler<Integer,Object>()   // ②
    {
        @Override
        public void completed(Integer result, Object attachment)
        {
            buff.flip();
            // 将 buff 中的内容转换为字符串
            String content = StandardCharsets.UTF_8
                .decode(buff).toString();
            // 显示从服务器端读取的数据
            jta.append("某人说: " + content + "\n");
            buff.clear();
            clientChannel.read(buff , null , this);
        }
        @Override
        public void failed(Throwable ex, Object attachment)
        {
            System.out.println("读取数据失败: " + ex);
        }
    });
}
public static void main(String[] args)
    throws Exception
{
    AIOClient client = new AIOClient();
    client.init();
```

```
            client.connect();
        }
    }
```

上面程序同样使用了 CompletionHandler 来读取服务器端数据，如程序中②号粗体字代码所示。上面程序使用了 Swing 的键盘驱动，因此当用户在 JTextField 组件中按下“Ctrl+Enter”键时即可向服务器端发送消息，向服务器端发送消息的代码如程序中①号粗体字代码所示。

17.4 基于 UDP 协议的网络编程

UDP 协议是一种不可靠的网络协议，它在通信实例的两端各建立一个 Socket，但这两个 Socket 之间并没有虚拟链路，这两个 Socket 只是发送、接收数据报的对象。Java 提供了 DatagramSocket 对象作为基于 UDP 协议的 Socket，使用 DatagramPacket 代表 DatagramSocket 发送、接收的数据报。

17.4.1 UDP 协议基础

UDP 协议是英文 User Datagram Protocol 的缩写，即用户数据报协议，主要用来支持那些需要在计算机之间传输数据的网络连接。UDP 协议从问世至今已经被使用了很多年，虽然 UDP 协议目前应用不如 TCP 协议广泛，但 UDP 协议依然是一个非常实用和可行的网络传输层协议。尤其是在一些实时性很强的应用场景中，比如网络游戏、视频会议等，UDP 协议的快速更具有独特的魅力。

UDP 协议是一种面向非连接的协议，面向非连接指的是在正式通信前不必与对方先建立连接，不管对方状态就直接发送。至于对方是否可以接收到这些数据内容，UDP 协议无法控制，因此说 UDP 协议是一种不可靠的协议。UDP 协议适用于一次只传送少量数据、对可靠性要求不高的应用环境。

与前面介绍的 TCP 协议一样，UDP 协议直接位于 IP 协议之上。实际上，IP 协议属于 OSI 参考模型的网络层协议，而 UDP 协议和 TCP 协议都属于传输层协议。

因为 UDP 协议是面向非连接的协议，没有建立连接的过程，因此它的通信效率很高；但也正因为如此，它的可靠性不如 TCP 协议。

UDP 协议的主要作用是完成网络数据流和数据报之间的转换——在信息的发送端，UDP 协议将网络数据流封装成数据报，然后将数据报发送出去；在信息的接收端，UDP 协议将数据报转换成实际数据内容。

提示：

可以认为 UDP 协议的 Socket 类似于码头，数据报则类似于集装箱；码头的作用就是负责发送、接收集装箱，而 DatagramSocket 的作用则是发送、接收数据报。因此对于基于 UDP 协议的通信双方而言，没有所谓的客户端和服务器端的概念。

UDP 协议和 TCP 协议简单对比如下。

- TCP 协议：可靠，传输大小无限制，但是需要连接建立时间，差错控制开销大。
- UDP 协议：不可靠，差错控制开销较小，传输大小限制在 64KB 以下，不需要建立连接。

17.4.2 使用 DatagramSocket 发送、接收数据

Java 使用 DatagramSocket 代表 UDP 协议的 Socket，DatagramSocket 本身只是码头，不维护状态，不能产生 IO 流，它的唯一作用就是接收和发送数据报，Java 使用 DatagramPacket 来代表数据报，DatagramSocket 接收和发送的数据都是通过 DatagramPacket 对象完成的。

先看一下 DatagramSocket 的构造器。

- DatagramSocket()：创建一个 DatagramSocket 实例，并将该对象绑定到本机默认 IP 地址、本机所有可用端口中随机选择的某个端口。
- DatagramSocket(int prot)：创建一个 DatagramSocket 实例，并将该对象绑定到本机默认 IP 地址、指定端口。
- DatagramSocket(int port, InetAddress laddr)：创建一个 DatagramSocket 实例，并将该对象绑定到

指定 IP 地址、指定端口。

通过上面三个构造器中的任意一个构造器即可创建一个 DatagramSocket 实例，通常在创建服务器时，创建指定端口的 DatagramSocket 实例——这样保证其他客户端可以将数据发送到该服务器。一旦得到了 DatagramSocket 实例之后，就可以通过如下两个方法来接收和发送数据。

- receive(DatagramPacket p)：从该 DatagramSocket 中接收数据报。
- send(DatagramPacket p)：以该 DatagramSocket 对象向外发送数据报。

从上面两个方法可以看出，使用 DatagramSocket 发送数据报时，DatagramSocket 并不知道将该数据报发送到哪里，而是由 DatagramPacket 自身决定数据报的目的地。就像码头并不知道每个集装箱的目的地，码头只是将这些集装箱发送出去，而集装箱本身包含了该集装箱的目的地。

下面看一下 DatagramPacket 的构造器。

- DatagramPacket(byte[] buf,int length)：以一个空数组来创建 DatagramPacket 对象，该对象的作用是接收 DatagramSocket 中的数据。
- DatagramPacket(byte[] buf, int length, InetAddress addr, int port)：以一个包含数据的数组来创建 DatagramPacket 对象，创建该 DatagramPacket 对象时还指定了 IP 地址和端口——这就决定了该数据报的目的地。
- DatagramPacket(byte[] buf, int offset, int length)：以一个空数组来创建 DatagramPacket 对象，并指定接收到的数据放入 buf 数组中时从 offset 开始，最多放 length 个字节。
- DatagramPacket(byte[] buf, int offset, int length, InetAddress address, int port)：创建一个用于发送的 DatagramPacket 对象，指定发送 buf 数组中从 offset 开始，总共 length 个字节。

提示：

当 Client/Server 程序使用 UDP 协议时，实际上并没有明显的服务器端和客户端，因为两方都需要先建立一个 DatagramSocket 对象，用来接收或发送数据报，然后使用 DatagramPacket 对象作为传输数据的载体。通常固定 IP 地址、固定端口的 DatagramSocket 对象所在的程序被称为服务器，因为该 DatagramSocket 可以主动接收客户端数据。

在接收数据之前，应该采用上面的第一个或第三个构造器生成一个 DatagramPacket 对象，给出接收数据的字节数组及其长度。然后调用 DatagramSocket 的 receive()方法等待数据报的到来，receive()将一直等待（该方法会阻塞调用该方法的线程），直到收到一个数据报为止。如下代码所示。

```
// 创建一个接收数据的 DatagramPacket 对象
DatagramPacket packet=new DatagramPacket(buf, 256);
// 接收数据报
socket.receive(packet);
```

在发送数据之前，调用第二个或第四个构造器创建 DatagramPacket 对象，此时的字节数组里存放了想发送的数据。除此之外，还要给出完整的目的地址，包括 IP 地址和端口号。发送数据是通过 DatagramSocket 的 send()方法实现的，send()方法根据数据报的目的地址来寻径以传送数据报。如下代码所示。

```
// 创建一个发送数据的 DatagramPacket 对象
DatagramPacket packet = new DatagramPacket(buf, length, address, port);
// 发送数据报
socket.send(packet);
```

提示：

使用 DatagramPacket 接收数据时，会感觉 DatagramPacket 设计得过于烦琐。开发者只关心该 DatagramPacket 能放多少数据，而 DatagramPacket 是否采用字节数组来存储数据完全不想关心。但 Java 要求创建接收数据用的 DatagramPacket 时，必须传入一个空的字节数组，该数组的长度决定了该 DatagramPacket 能放多少数据，这实际上暴露了 DatagramPacket 的实现细节。接着 DatagramPacket 又提供了一个 getData()方法，该方法又可以返回 Datagram Packet 对象里封装的字节数组，该方法更显得有些多余——如果程序需要获取 DatagramPacket 里封装的字节数组，直接访问传给 DatagramPacket 构造器的字节数组实参即可，无须调用该方法。

当服务器端（也可以是客户端）接收到一个 DatagramPacket 对象后，如果想向该数据报的发送者“反馈”一些信息，但由于 UDP 协议是面向非连接的，所以接收者并不知道每个数据报由谁发送过来，但程序可以调用 DatagramPacket 的如下三个方法来获取发送者的 IP 地址和端口。

- InetAddress getAddress()：当程序准备发送此数据报时，该方法返回此数据报的目标机器的 IP 地址；当程序刚接收到一个数据报时，该方法返回该数据报的发送主机的 IP 地址。
- int getPort()：当程序准备发送此数据报时，该方法返回此数据报的目标机器的端口；当程序刚接收到一个数据报时，该方法返回该数据报的发送主机的端口。
- SocketAddress getSocketAddress()：当程序准备发送此数据报时，该方法返回此数据报的目标 SocketAddress；当程序刚接收到一个数据报时，该方法返回该数据报的发送主机的 SocketAddress。

提示：

getSocketAddress()方法的返回值是一个 SocketAddress 对象，该对象实际上就是一个 IP 地址和一个端口号。也就是说，SocketAddress 对象封装了一个 InetAddress 对象和一个代表端口的整数，所以使用 SocketAddress 对象可以同时代表 IP 地址和端口。

下面程序使用 DatagramSocket 实现了 Server/Client 结构的网络通信。本程序的服务器端使用循环 1000 次来读取 DatagramSocket 中的数据报，每当读取到内容之后便向该数据报的发送者送回一条信息。服务器端程序代码如下。

程序清单：codes\17\17.4\UdpServer.java

```
public class UdpServer
{
    public static final int PORT = 30000;
    // 定义每个数据报的大小最大为 4KB
    private static final int DATA_LEN = 4096;
    // 定义接收网络数据的字节数组
    byte[] inBuff = new byte[DATA_LEN];
    // 以指定字节数组创建准备接收数据的 DatagramPacket 对象
    private DatagramPacket inPacket =
        new DatagramPacket(inBuff , inBuff.length);
    // 定义一个用于发送的 DatagramPacket 对象
    private DatagramPacket outPacket;
    // 定义一个字符串数组，服务器端发送该数组的元素
    String[] books = new String[]
    {
        "疯狂 Java 讲义",
        "轻量级 Java EE 企业应用实战",
        "疯狂 Android 讲义",
        "疯狂 Ajax 讲义"
    };
    public void init()throws IOException
    {
        try(
            // 创建 DatagramSocket 对象
            DatagramSocket socket = new DatagramSocket(PORT))
        {
            // 采用循环接收数据
            for (int i = 0; i < 1000 ; i++ )
            {
                // 读取 Socket 中的数据，读到的数据放入 inPacket 封装的数组里
                socket.receive(inPacket);
                // 判断 inPacket.getData()和 inBuff 是否是同一个数组
                System.out.println(inBuff == inPacket.getData());
                // 将接收到的内容转换成字符串后输出
                System.out.println(new String(inBuff
                    , 0 , inPacket.getLength()));
                // 从字符串数组中取出一个元素作为发送数据
                byte[] sendData = books[i % 4].getBytes();
                // 以指定的字节数组作为发送数据，以刚接收到的 DatagramPacket 的
                // 源 SocketAddress 作为目标 SocketAddress 创建 DatagramPacket
```

```
                outPacket = new DatagramPacket(sendData
                    , sendData.length , inPacket.getSocketAddress());
                // 发送数据
                socket.send(outPacket);
            }
        }
    }
    public static void main(String[] args)
        throws IOException
    {
        new UdpServer().init();
    }
}
```

上面程序中的粗体字代码就是使用 DatagramSocket 发送、接收 DatagramPacket 的关键代码，该程序可以接收 1000 个客户端发送过来的数据。

客户端程序代码也与此类似，客户端采用循环不断地读取用户键盘输入，每当读取到用户输入的内容后就将该内容封装成 DatagramPacket 数据报，再将该数据报发送出去；接着把 DatagramSocket 中的数据读入接收用的 DatagramPacket 中（实际上是读入该 DatagramPacket 所封装的字节数组中）。客户端程序代码如下。

程序清单：codes\17\17.4\UdpClient.java

```
public class UdpClient
{
    // 定义发送数据报的目的地
    public static final int DEST_PORT = 30000;
    public static final String DEST_IP = "127.0.0.1";
    // 定义每个数据报的大小最大为 4KB
    private static final int DATA_LEN = 4096;
    // 定义接收网络数据的字节数组
    byte[] inBuff = new byte[DATA_LEN];
    // 以指定的字节数组创建准备接收数据的 DatagramPacket 对象
    private DatagramPacket inPacket =
        new DatagramPacket(inBuff , inBuff.length);
    // 定义一个用于发送的 DatagramPacket 对象
    private DatagramPacket outPacket = null;
    public void init()throws IOException
    {
        try(
            // 创建一个客户端 DatagramSocket，使用随机端口
            DatagramSocket socket = new DatagramSocket())
        {
            // 初始化发送用的 DatagramSocket，它包含一个长度为 0 的字节数组
            outPacket = new DatagramPacket(new byte[0] , 0
                , InetAddress.getByName(DEST_IP) , DEST_PORT);
            // 创建键盘输入流
            Scanner scan = new Scanner(System.in);
            // 不断地读取键盘输入
            while(scan.hasNextLine())
            {
                // 将键盘输入的一行字符串转换成字节数组
                byte[] buff = scan.nextLine().getBytes();
                // 设置发送用的 DatagramPacket 中的字节数据
                outPacket.setData(buff);
                // 发送数据报
                socket.send(outPacket);
                // 读取 Socket 中的数据，读到的数据放在 inPacket 所封装的字节数组中
                socket.receive(inPacket);
                System.out.println(new String(inBuff , 0
                    , inPacket.getLength()));
            }
        }
    }
    public static void main(String[] args)
        throws IOException
    {
```

```
            new UdpClient().init();
        }
    }
```

上面程序中的粗体字代码同样也是使用 DatagramSocket 发送、接收 DatagramPacket 的关键代码，这些代码与服务器端代码基本相似。而客户端与服务器端的唯一区别在于：服务器端的 IP 地址、端口是固定的，所以客户端可以直接将该数据报发送给服务器端，而服务器端则需要根据接收到的数据报来决定“反馈”数据报的目的地。

读者可能会发现，使用 DatagramSocket 进行网络通信时，服务器端无须也无法保存每个客户端的状态，客户端把数据报发送到服务器端后，完全有可能立即退出。但不管客户端是否退出，服务器端都无法知道客户端的状态。

当使用 UDP 协议时，如果想让一个客户端发送的聊天信息被转发到其他所有的客户端则比较困难，可以考虑在服务器端使用 Set 集合来保存所有的客户端信息，每当接收到一个客户端的数据报之后，程序检查该数据报的源 SocketAddress 是否在 Set 集合中，如果不在就将该 SocketAddress 添加到该 Set 集合中。这样又涉及一个问题：可能有些客户端发送一个数据报之后永久性地退出了程序，但服务器端还将该客户端的 SocketAddress 保存在 Set 集合中……总之，这种方式需要处理的问题比较多，编程比较烦琐。幸好 Java 为 UDP 协议提供了 MulticastSocket 类，通过该类可以轻松地实现多点广播。

17.4.3 使用 MulticastSocket 实现多点广播

DatagramSocket 只允许数据报发送给指定的目标地址，而 MulticastSocket 可以将数据报以广播方式发送到多个客户端。

若要使用多点广播，则需要让一个数据报标有一组目标主机地址，当数据报发出后，整个组的所有主机都能收到该数据报。IP 多点广播（或多点发送）实现了将单一信息发送到多个接收者的广播，其思想是设置一组特殊网络地址作为多点广播地址，每一个多点广播地址都被看做一个组，当客户端需要发送、接收广播信息时，加入到该组即可。

IP 协议为多点广播提供了这批特殊的 IP 地址，这些 IP 地址的范围是 224.0.0.0 至 239.255.255.255。多点广播示意图如图 17.8 所示。

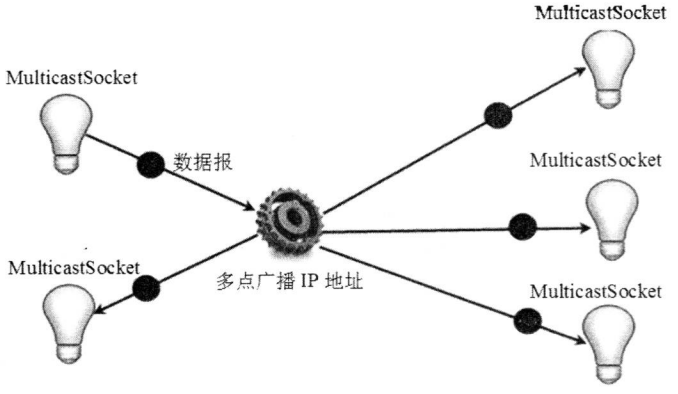


图 17.8 多点广播示意图

从图 17.8 中可以看出，MulticastSocket 类是实现多点广播的关键，当 MulticastSocket 把一个 DatagramPacket 发送到多点广播 IP 地址时，该数据报将被自动广播到加入该地址的所有 Multicast Socket。MulticastSocket 既可以将数据报发送到多点广播地址，也可以接收其他主机的广播信息。

MulticastSocket 有点像 DatagramSocket，事实上 MulticastSocket 是 DatagramSocket 的一个子类，也就是说，MulticastSocket 是特殊的 DatagramSocket。当要发送一个数据报时，可以使用随机端口创建 MulticastSocket，也可以在指定端口创建 MulticastSocket。MulticastSocket 提供了如下三个构造器。

- public MulticastSocket()：使用本机默认地址、随机端口来创建 MulticastSocket 对象。
- public MulticastSocket(int portNumber)：使用本机默认地址、指定端口来创建 MulticastSocket 对象。
- public MulticastSocket(SocketAddress bindaddr)：使用本机指定 IP 地址、指定端口来创建

MulticastSocket 对象。

创建 MulticastSocket 对象后，还需要将该 MulticastSocket 加入到指定的多点广播地址，MulticastSocket 使用 joinGroup()方法加入指定组；使用 leaveGroup()方法脱离一个组。

- joinGroup(InetAddress multicastAddr)：将该 MulticastSocket 加入指定的多点广播地址。
- leaveGroup(InetAddress multicastAddr)：让该 MulticastSocket 离开指定的多点广播地址。

在某些系统中，可能有多个网络接口。这可能会给多点广播带来问题，这时候程序需要在一个指定的网络接口上监听，通过调用 setInterface()方法可以强制 MulticastSocket 使用指定的网络接口；也可以使用 getInterface()方法查询 MulticastSocket 监听的网络接口。

提示：

如果创建仅用于发送数据报的 MulticastSocket 对象，则使用默认地址、随机端口即可。但如果创建接收用的 MulticastSocket 对象，则该 MulticastSocket 对象必须具有指定端口，否则发送方无法确定发送数据报的目标端口。

MulticastSocket 用于发送、接收数据报的方法与 DatagramSocket 完全一样。但 MulticastSocket 比 DatagramSocket 多了一个 setTimeToLive(int ttl)方法，该 ttl 参数用于设置数据报最多可以跨过多少个网络，当 ttl 的值为 0 时，指定数据报应停留在本地主机；当 ttl 的值为 1 时，指定数据报发送到本地局域网；当 ttl 的值为 32 时，意味着只能发送到本站点的网络上；当 ttl 的值为 64 时，意味着数据报应保留在本地区；当 ttl 的值为 128 时，意味着数据报应保留在本大洲；当 ttl 的值为 255 时，意味着数据报可发送到所有地方；在默认情况下，该 ttl 的值为 1。

从图 17.8 中可以看出，使用 MulticastSocket 进行多点广播时所有的通信实体都是平等的，它们都将自己的数据报发送到多点广播 IP 地址，并使用 MulticastSocket 接收其他人发送的广播数据报。下面程序使用 MulticastSocket 实现了一个基于广播的多人聊天室。程序只需要一个 MulticastSocket，两个线程，其中 MulticastSocket 既用于发送，也用于接收；一个线程负责接收用户键盘输入，并向 MulticastSocket 发送数据，另一个线程则负责从 MulticastSocket 中读取数据。

程序清单：codes\17\17.4\MulticastSocketTest.java

```
// 让该类实现 Runnable 接口，该类的实例可作为线程的 target
public class MulticastSocketTest implements Runnable
{
    // 使用常量作为本程序的多点广播 IP 地址
    private static final String BROADCAST_IP
        = "230.0.0.1";
    // 使用常量作为本程序的多点广播目的地端口
    public static final int BROADCAST_PORT = 30000;
    // 定义每个数据报的大小最大为 4KB
    private static final int DATA_LEN = 4096;
    // 定义本程序的 MulticastSocket 实例
    private MulticastSocket socket = null;
    private InetAddress broadcastAddress = null;
    private Scanner scan = null;
    // 定义接收网络数据的字节数组
    byte[] inBuff = new byte[DATA_LEN];
    // 以指定字节数组创建准备接收数据的 DatagramPacket 对象
    private DatagramPacket inPacket
        = new DatagramPacket(inBuff , inBuff.length);
    // 定义一个用于发送的 DatagramPacket 对象
    private DatagramPacket outPacket = null;
    public void init()throws IOException
    {
        try(
            // 创建键盘输入流
            Scanner scan = new Scanner(System.in))
        {
            // 创建用于发送、接收数据的 MulticastSocket 对象
            // 由于该 MulticastSocket 对象需要接收数据，所以有指定端口
            socket = new MulticastSocket(BROADCAST_PORT);
            broadcastAddress = InetAddress.getByName(BROADCAST_IP);
```

```
            // 将该 socket 加入指定的多点广播地址
            socket.joinGroup(broadcastAddress);
            // 设置本 MulticastSocket 发送的数据报会被回送到自身
            socket.setLoopbackMode(false);
            // 初始化发送用的 DatagramSocket，它包含一个长度为 0 的字节数组
            outPacket = new DatagramPacket(new byte[0]
                , 0 , broadcastAddress , BROADCAST_PORT);
            // 启动以本实例的 run()方法作为线程执行体的线程
            new Thread(this).start();
            // 不断地读取键盘输入
            while(scan.hasNextLine())
            {
                // 将键盘输入的一行字符串转换成字节数组
                byte[] buff = scan.nextLine().getBytes();
                // 设置发送用的 DatagramPacket 里的字节数据
                outPacket.setData(buff);
                // 发送数据报
                socket.send(outPacket);
            }
        }
        finally
        {
            socket.close();
        }
    }
    public void run()
    {
        try
        {
            while(true)
            {
                // 读取 Socket 中的数据，读到的数据放在 inPacket 所封装的字节数组里
                socket.receive(inPacket);
                // 打印输出从 socket 中读取的内容
                System.out.println("聊天信息: " + new String(inBuff
                    , 0 , inPacket.getLength()));
            }
        }
        // 捕获异常
        catch (IOException ex)
        {
            ex.printStackTrace();
            try
            {
                if (socket != null)
                {
                    // 让该 Socket 离开该多点 IP 广播地址
                    socket.leaveGroup(broadcastAddress);
                    // 关闭该 Socket 对象
                    socket.close();
                }
                System.exit(1);
            }
            catch (IOException e)
            {
                e.printStackTrace();
            }
        }
    }
    public static void main(String[] args)
        throws IOException
    {
        new MulticastSocketTest().init();
    }
}
```

上面程序中 init()方法里的第一行粗体字代码先创建了一个 MulticastSocket 对象，由于需要使用该对象接收数据报，所以为该 Socket 对象设置使用固定端口；第二行粗体字代码将该 Socket 对象添加到

指定的多点广播 IP 地址；第三行粗体字代码设置该 Socket 发送的数据报会被回送到自身（即该 Socket 可以接收到自己发送的数据报）。至于程序中使用 MulticastSocket 发送、接收数据报的代码，与使用 DatagramSocket 并没有区别，故此处不再赘述。

下面将结合 MulticastSocket 和 DatagramSocket 开发一个简单的局域网即时通信工具，局域网内每个用户启动该工具后，就可以看到该局域网内所有的在线用户，该用户也会被其他用户看到，即看到如图 17.9 所示的窗口。

在图 17.9 所示的用户列表中双击任意一个用户，即可启动一个如图 17.10 所示的交谈界面。

如果双击图 17.9 所示用户列表窗口中的“所有人”列表项，即可启动一个与图 17.10 相似的交谈界面，不同的是通过该窗口发送的消息将会被所有人看到。

该程序的实现思路是，每个用户都启动两个 Socket，即一个 MulticastSocket，一个 DatagramSocket。其中 MulticastSocket 会周期性地向 230.0.0.1 发送在线信息，且所有用户的 MulticastSocket 都会加入到 230.0.0.1 这个多点广播 IP 地址中，这样每个用户都可以收到其他用户广播的在线信息，如果系统经过一段时间没有收到某个用户广播的在线信息，则从用户列表中删除该用户。除此之外，该 MulticastSocket 还用于向所有用户发送广播信息。

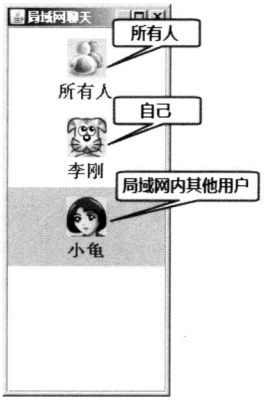


图 17.9　局域网聊天工具

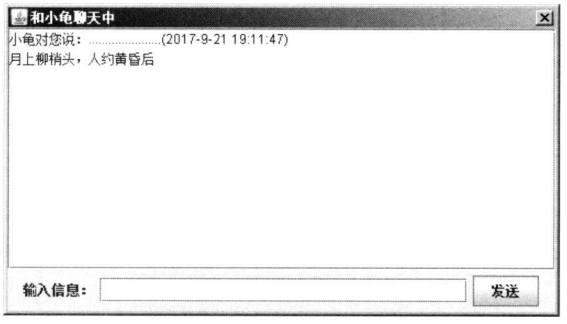


图 17.10　与特定用户交谈界面

DatagramSocket 主要用于发送私聊信息，当用户收到其他用户广播来的 DatagramPacket 时，即可获取该用户 MulticastSocket 对应的 SocketAddress，这个 SocketAddress 将作为发送私聊信息的重要依据——本程序让 MulticastSocket 在 30000 端口监听，而 DatagramSocket 在 30001 端口监听，这样程序就可以根据其他用户广播来的 DatagramPacket 得到他的 DatagramSocket 所在的地址。

本系统提供了一个 UserInfo 类，该类封装了用户名、图标、对应的 SocketAddress 以及该用户对应的交谈窗口、失去联系的次数等信息。该类的代码片段如下。

程序清单：codes\17\17.4\LanTalk\UserInfo.java

```
public class UserInfo
{
    // 该用户的图标
    private String icon;
    // 该用户的名字
    private String name;
    // 该用户的 MulitcastSocket 所在的 IP 地址和端口
    private SocketAddress address;
    // 该用户失去联系的次数
    private int lost;
    // 该用户对应的交谈窗口
    private ChatFrame chatFrame;
    public UserInfo(){}
    // 有参数的构造器
    public UserInfo(String icon , String name
        , SocketAddress address , int lost)
    {
        this.icon = icon;
        this.name = name;
```

```
        this.address = address;
        this.lost = lost;
    }
    // 省略所有成员变量的 setter 和 getter 方法
    ...
    // 使用 address 作为该用户的标识，所以根据 address
    // 重写 hashCode()和 equals()方法
    public int hashCode()
    {
        return address.hashCode();
    }
    public boolean equals(Object obj)
    {
        if (obj != null && obj.getClass() == UserInfo.class)
        {
            UserInfo target = (UserInfo)obj;
            if (address != null)
            {
                return address.equals(target.getAddress());
            }
        }
        return false;
    }
}
```

通过 UserInfo 类的封装，所有客户端只需要维护该 UserInfo 类的列表，程序就可以实现广播、发送私聊信息等功能。本程序底层通信的工具类则需要一个 MulticastSocket 和一个 DatagramSocket，该工具类的代码如下。

程序清单：codes\17\17.4\LanTalk\ComUtil.java

```
// 聊天交换信息的工具类
public class ComUtil
{
    // 定义本程序通信所使用的字符集
    public static final String CHARSET = "utf-8";
    // 使用常量作为本程序的多点广播 IP 地址
    private static final String BROADCAST_IP
        = "230.0.0.1";
    // 使用常量作为本程序的多点广播目的地端口
    // DatagramSocket 所用的端口为该端口号+1
    public static final int BROADCAST_PORT = 30000;
    // 定义每个数据报的大小最大为 4KB
    private static final int DATA_LEN = 4096;
    // 定义本程序的 MulticastSocket 实例
    private MulticastSocket socket = null;
    // 定义本程序私聊的 Socket 实例
    private DatagramSocket singleSocket = null;
    // 定义广播的 IP 地址
    private InetAddress broadcastAddress = null;
    // 定义接收网络数据的字节数组
    byte[] inBuff = new byte[DATA_LEN];
    // 以指定字节数组创建准备接收数据的 DatagramPacket 对象
    private DatagramPacket inPacket =
        new DatagramPacket(inBuff , inBuff.length);
    // 定义一个用于发送的 DatagramPacket 对象
    private DatagramPacket outPacket = null;
    // 聊天的主界面程序
    private LanTalk lanTalk;
    // 构造器，初始化资源
    public ComUtil(LanTalk lanTalk) throws Exception
    {
        this.lanTalk = lanTalk;
        // 创建用于发送、接收数据的 MulticastSocket 对象
        // 因为该 MulticastSocket 对象需要接收数据，所以有指定端口
        socket = new MulticastSocket(BROADCAST_PORT);
        // 创建私聊用的 DatagramSocket 对象
```

```
        singleSocket = new DatagramSocket(BROADCAST_PORT + 1);
        broadcastAddress = InetAddress.getByName(BROADCAST_IP);
        // 将该 socket 加入指定的多点广播地址
        socket.joinGroup(broadcastAddress);
        // 设置本 MulticastSocket 发送的数据报被回送到自身
        socket.setLoopbackMode(false);
        // 初始化发送用的 DatagramSocket，它包含一个长度为 0 的字节数组
        outPacket = new DatagramPacket(new byte[0]
            , 0 , broadcastAddress , BROADCAST_PORT);
        // 启动两个读取网络数据的线程
        new ReadBroad().start();
        Thread.sleep(1);
        new ReadSingle().start();
    }
    // 广播消息的工具方法
    public void broadCast(String msg)
    {
        try
        {
            // 将 msg 字符串转换成字节数组
            byte[] buff = msg.getBytes(CHARSET);
            // 设置发送用的 DatagramPacket 里的字节数据
            outPacket.setData(buff);
            // 发送数据报
            socket.send(outPacket);
        }
        // 捕获异常
        catch (IOException ex)
        {
            ex.printStackTrace();
            if (socket != null)
            {
                // 关闭该 Socket 对象
                socket.close();
            }
            JOptionPane.showMessageDialog(null
                , "发送信息异常，请确认 30000 端口空闲，且网络连接正常！"
                , "网络异常", JOptionPane.ERROR_MESSAGE);
            System.exit(1);
        }
    }
    // 定义向单独用户发送消息的方法
    public void sendSingle(String msg , SocketAddress dest)
    {
        try
        {
            // 将 msg 字符串转换成字节数组
            byte[] buff = msg.getBytes(CHARSET);
            DatagramPacket packet = new DatagramPacket(buff
                , buff.length , dest);
            singleSocket.send(packet);
        }
        // 捕获异常
        catch (IOException ex)
        {
            ex.printStackTrace();
            if (singleSocket != null)
            {
                // 关闭该 Socket 对象
                singleSocket.close();
            }
            JOptionPane.showMessageDialog(null
                , "发送信息异常，请确认 30001 端口空闲，且网络连接正常！"
                , "网络异常", JOptionPane.ERROR_MESSAGE);
            System.exit(1);
        }
    }
    // 不断地从 DatagramSocket 中读取数据的线程
```

```
class ReadSingle extends Thread
{
    // 定义接收网络数据的字节数组
    byte[] singleBuff = new byte[DATA_LEN];
    private DatagramPacket singlePacket =
        new DatagramPacket(singleBuff , singleBuff.length);
    public void run()
    {
        while (true)
        {
            try
            {
                // 读取 Socket 中的数据
                singleSocket.receive(singlePacket);
                // 处理读到的信息
                lanTalk.processMsg(singlePacket , true);
            }
            // 捕获异常
            catch (IOException ex)
            {
                ex.printStackTrace();
                if (singleSocket != null)
                {
                    // 关闭该 Socket 对象
                    singleSocket.close();
                }
                JOptionPane.showMessageDialog(null
                    , "接收信息异常，请确认 30001 端口空闲，且网络连接正常！"
                    , "网络异常", JOptionPane.ERROR_MESSAGE);
                System.exit(1);
            }
        }
    }
}
// 持续读取 MulticastSocket 的线程
class ReadBroad extends Thread
{
    public void run()
    {
        while (true)
        {
            try
            {
                // 读取 Socket 中的数据
                socket.receive(inPacket);
                // 打印输出从 Socket 中读取的内容
                String msg = new String(inBuff , 0
                    , inPacket.getLength() , CHARSET);
                // 读到的内容是在线信息
                if (msg.startsWith(YeekuProtocol.PRESENCE)
                    && msg.endsWith(YeekuProtocol.PRESENCE))
                {
                    String userMsg = msg.substring(2
                        , msg.length() - 2);
                    String[] userInfo = userMsg.split(YeekuProtocol
                        .SPLITTER);
                    UserInfo user = new UserInfo(userInfo[1]
                        , userInfo[0] , inPacket.getSocketAddress(), 0);
                    // 控制是否需要添加该用户的旗标
                    boolean addFlag = true;
                    ArrayList<Integer> delList = new ArrayList<>();
                    // 遍历系统中已有的所有用户，该循环必须循环完成
                    for (int i = 1 ; i < lanTalk.getUserNum() ; i++ )
                    {
                        UserInfo current = lanTalk.getUser(i);
                        // 将所有用户失去联系的次数加 1
                        current.setLost(current.getLost() + 1);
                        // 如果该信息由指定用户发送
```

```
                        if (current.equals(user))
                        {
                            current.setLost(0);
                            // 设置该用户无须添加
                            addFlag = false;
                        }
                        if (current.getLost() > 2)
                        {
                            delList.add(i);
                        }
                    }
                    // 删除 delList 中的所有索引对应的用户
                    for (int i = 0; i < delList.size() ; i++)
                    {
                        lanTalk.removeUser(delList.get(i));
                    }
                    if (addFlag)
                    {
                        // 添加新用户
                        lanTalk.addUser(user);
                    }
                }
                // 读到的内容是公聊信息
                else
                {
                    // 处理读到的信息
                    lanTalk.processMsg(inPacket , false);
                }
            }
            // 捕获异常
            catch (IOException ex)
            {
                ex.printStackTrace();
                if (socket != null)
                {
                    // 关闭该 Socket 对象
                    socket.close();
                }
                JOptionPane.showMessageDialog(null
                    , "接收信息异常，请确认 30000 端口空闲，且网络连接正常！"
                    , "网络异常", JOptionPane.ERROR_MESSAGE);
                System.exit(1);
            }
        }
    }
}
```

该类主要实现底层的网络通信功能，在该类中提供了一个 broadCast()方法，该方法使用 Multicast Socket 将指定字符串广播到所有客户端；还提供了 sendSingle()方法，该方法使用 DatagramSocket 将指定字符串发送到指定 SocketAddress，如程序中前两行粗体字代码所示。除此之外，该类还提供了两个内部线程类：ReadSingle 和 ReadBroad，这两个线程类采用循环不断地读取 DatagramSocket 和 Multicast Socket 中的数据，如果读到的信息是广播来的在线信息，则保持该用户在线；如果读到的是用户的聊天信息，则直接将该信息显示出来。

在该类中用到了本程序的一个主类：LanTalk，该类使用 DefaultListModel 来维护用户列表，该类里的每个列表项就是一个 UserInfo。该类还提供了一个 ImageCellRenderer，该类用于将列表项绘制出用户图标和用户名字。

程序清单：codes\17\17.4\LanTalk\LanTalk.java

```
public class LanTalk extends JFrame
{
    private DefaultListModel<UserInfo> listModel
        = new DefaultListModel<>();
    // 定义一个 JList 对象
    private JList<UserInfo> friendsList = new JList<>(listModel);
```

```
// 定义一个用于格式化日期的格式器
private DateFormat formatter = DateFormat.getDateTimeInstance();
public LanTalk()
{
    super("局域网聊天");
    // 设置该 JList 使用 ImageCellRenderer 作为单元格绘制器
    friendsList.setCellRenderer(new ImageCellRenderer());
    listModel.addElement(new UserInfo("all" , "所有人"
        , null , -2000));
    friendsList.addMouseListener(new ChangeMusicListener());
    add(new JScrollPane(friendsList));
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setBounds(2, 2, 160 , 600);
}
// 根据地址来查询用户
public UserInfo getUserBySocketAddress(SocketAddress address)
{
    for (int i = 1 ; i < getUserNum() ; i++)
    {
        UserInfo user = getUser(i);
        if (user.getAddress() != null
            && user.getAddress().equals(address))
        {
            return user;
        }
    }
    return null;
}
// ------下面四个方法是对 ListModel 的包装------
// 向用户列表中添加用户
public void addUser(UserInfo user)
{
    listModel.addElement(user);
}
// 从用户列表中删除用户
public void removeUser(int pos)
{
    listModel.removeElementAt(pos);
}
// 获取该聊天窗口的用户数量
public int getUserNum()
{
    return listModel.size();
}
// 获取指定位置的用户
public UserInfo getUser(int pos)
{
    return listModel.elementAt(pos);
}
// 实现 JList 上的鼠标双击事件监听器
class ChangeMusicListener extends MouseAdapter
{
    public void mouseClicked(MouseEvent e)
    {
        // 如果鼠标的击键次数大于 2
        if (e.getClickCount() >= 2)
        {
            // 取出鼠标双击时选中的列表项
            UserInfo user = (UserInfo)friendsList.getSelectedValue();
            // 如果该列表项对应用户的交谈窗口为 null
            if (user.getChatFrame() == null)
            {
                // 为该用户创建一个交谈窗口，并让该用户引用该窗口
                user.setChatFrame(new ChatFrame(null , user));
            }
            // 如果该用户的窗口没有显示，则让该用户的窗口显示出来
            if (!user.getChatFrame().isShowing())
            {
```

```
                    user.getChatFrame().setVisible(true);
                }
            }
        }
    }
    /**
     * 处理网络数据报，该方法将根据聊天信息得到聊天者
     * 并将信息显示在聊天对话框中
     * @param packet 需要处理的数据报
     * @param single 该信息是否为私聊信息
     */
    public void processMsg(DatagramPacket packet , boolean single)
    {
        // 获取发送该数据报的 SocketAddress
        InetSocketAddress srcAddress = (InetSocketAddress)
            packet.getSocketAddress();
        // 如果是私聊信息，则该 Packet 获取的是 DatagramSocket 的地址
        // 将端口号减 1 才是对应的 MulticastSocket 的地址
        if (single)
        {
            srcAddress = new InetSocketAddress(srcAddress.getHostName()
                , srcAddress.getPort() - 1);
        }
        UserInfo srcUser = getUserBySocketAddress(srcAddress);
        if (srcUser != null)
        {
            // 确定消息将要显示到哪个用户对应的窗口中
            UserInfo alertUser = single ? srcUser : getUser(0);
            // 如果该用户对应的窗口为空，则显示该窗口
            if (alertUser.getChatFrame() == null)
            {
                alertUser.setChatFrame(new ChatFrame(null , alertUser));
            }
            // 定义添加的提示信息
            String tipMsg = single ? "对您说：" : "对大家说：";
            try{
                // 显示提示信息
                alertUser.getChatFrame().addString(srcUser.getName()
                    + tipMsg + "......................("
                    + formatter.format(new Date()) + ")\n"
                    + new String(packet.getData() , 0 , packet.getLength()
                    , ComUtil.CHARSET)) + "\n");
            }
            catch (Exception ex) { ex.printStackTrace(); }
            if (!alertUser.getChatFrame().isShowing())
            {
                alertUser.getChatFrame().setVisible(true);
            }
        }
    }
    // 主方法，程序的入口
    public static void main(String[] args)
    {
        LanTalk lanTalk = new LanTalk();
        new LoginFrame(lanTalk , "请输入用户名、头像后登录");
    }
}
// 定义用于改变 JList 列表项外观的类
class ImageCellRenderer extends JPanel
    implements ListCellRenderer<UserInfo>
{
    private ImageIcon icon;
    private String name;
    // 定义绘制单元格时的背景色
    private Color background;
    // 定义绘制单元格时的前景色
    private Color foreground;
    @Override
```

```
    public Component getListCellRendererComponent(JList list
        , UserInfo userInfo , int index
        , boolean isSelected , boolean cellHasFocus)
    {
        // 设置图标
        icon = new ImageIcon("ico/" + userInfo.getIcon() + ".gif");
        name = userInfo.getName();
        // 设置背景色、前景色
        background = isSelected ? list.getSelectionBackground()
            : list.getBackground();
        foreground = isSelected ? list.getSelectionForeground()
            : list.getForeground();
        // 返回该 JPanel 对象作为单元格绘制器
        return this;
    }
    // 重写 paintComponent 方法，改变 JPanel 的外观
    public void paintComponent(Graphics g)
    {
        int imageWidth = icon.getImage().getWidth(null);
        int imageHeight = icon.getImage().getHeight(null);
        g.setColor(background);
        g.fillRect(0, 0, getWidth(), getHeight());
        g.setColor(foreground);
        // 绘制好友图标
        g.drawImage(icon.getImage() , getWidth() / 2 - imageWidth / 2
            , 10 , null);
        g.setFont(new Font("SansSerif" , Font.BOLD , 18));
        // 绘制好友用户名
        g.drawString(name, getWidth() / 2 - name.length() * 10
            , imageHeight + 30 );
    }
    // 通过该方法来设置该 ImageCellRenderer 的最佳大小
    public Dimension getPreferredSize()
    {
        return new Dimension(60, 80);
    }
}
```

上面类中提供的 addUser()和 removeUser()方法暴露给通信类 ComUtil 使用，用于向用户列表中添加、删除用户。除此之外，该类还提供了一个 processMsg()方法，该方法用于处理网络中读取的数据报，将数据报中的内容取出，并显示在特定的窗口中。

提示：

上面讲解的只是本程序的关键类，本程序还涉及 YeekuProtocol、ChatFrame、LoginFrame 等类，由于篇幅关系，此处不再给出这些类的源代码，读者可以参考 codes\17\17.4\LanTalk 路径下的源代码。

17.5 使用代理服务器

从 Java 5 开始，Java 在 java.net 包下提供了 Proxy 和 ProxySelector 两个类，其中 Proxy 代表一个代理服务器，可以在打开 URLConnection 连接时指定 Proxy，创建 Socket 连接时也可以指定 Proxy；而 ProxySelector 代表一个代理选择器，它提供了对代理服务器更加灵活的控制，它可以对 HTTP、HTTPS、FTP、SOCKS 等进行分别设置，而且还可以设置不需要通过代理服务器的主机和地址。通过使用 ProxySelector，可以实现像在 Internet Explorer、Firefox 等软件中设置代理服务器类似的效果。

提示：

代理服务器的功能就是代理用户去取得网络信息。当使用浏览器直接连接其他 Internet 站点取得网络信息时，通常需要先发送请求，然后等响应到来。代理服务器是介于浏览器和服务器之间的一台服务器，设置了代理服务器之后，浏览器不是直接向 Web 服务器发送请求，而是向代理服务器发送请求，浏览器请求被先送到代理服务器，由代理服务器向真正的 Web 服务器发送请求，并取回浏览器所需要的信息，再送回给浏览器。由于大部

分代理服务器都具有缓冲功能，它会不断地将新取得的数据存储到代理服务器的本地存储器上，如果浏览器所请求的数据在它本机的存储器上已经存在而且是最新的，那么它就无须从 Web 服务器取数据，而直接将本地存储器上的数据送回浏览器，这样能显著提高浏览速度。归纳起来，代理服务器主要提供如下两个功能。

- 突破自身 IP 限制，对外隐藏自身 IP 地址。突破 IP 限制包括访问国外受限站点，访问国内特定单位、团体的内部资源。
- 提高访问速度，代理服务器提供的缓冲功能可以避免每个用户都直接访问远程主机，从而提高客户端访问速度。

17.5.1 直接使用 Proxy 创建连接

Proxy 有一个构造器：Proxy(Proxy.Type type, SocketAddress sa)，用于创建表示代理服务器的 Proxy 对象。其中 sa 参数指定代理服务器的地址，type 表示该代理服务器的类型，该服务器类型有如下三种。

- Proxy.Type.DIRECT：表示直接连接，不使用代理。
- Proxy.Type.HTTP：表示支持高级协议代理，如 HTTP 或 FTP。
- Proxy.Type.SOCKS：表示 SOCKS（V4 或 V5）代理。

一旦创建了 Proxy 对象之后，程序就可以在使用 URLConnection 打开连接时，或者创建 Socket 连接时传入一个 Proxy 对象，作为本次连接所使用的代理服务器。

其中 URL 包含了一个 URLConnection openConnection(Proxy proxy)方法，该方法使用指定的代理服务器来打开连接；而 Socket 则提供了一个 Socket(Proxy proxy)构造器，该构造器使用指定的代理服务器创建一个没有连接的 Socket 对象。

下面以 URLConnection 为例来介绍如何在 URLConnection 中使用代理服务器。

程序清单：codes\17\17.5\ProxyTest.java

```
public class ProxyTest
{
    // 下面是代理服务器的地址和端口
    // 换成实际有效的代理服务器的地址和端口
    final String PROXY_ADDR = "129.82.12.188";
    final int PROXY_PORT = 3124;
    // 定义需要访问的网站地址
    String urlStr = "http://www.crazyit.org";
    public void init()
        throws IOException , MalformedURLException
    {
        URL url = new URL(urlStr);
        // 创建一个代理服务器对象
        Proxy proxy = new Proxy(Proxy.Type.HTTP
            , new InetSocketAddress(PROXY_ADDR , PROXY_PORT));
        // 使用指定的代理服务器打开连接
        URLConnection conn = url.openConnection(proxy);
        // 设置超时时长
        conn.setConnectTimeout(3000);
        try(
            // 通过代理服务器读取数据的 Scanner
            Scanner scan = new Scanner(conn.getInputStream());
            PrintStream ps = new PrintStream("index.htm"))
        {
            while (scan.hasNextLine())
            {
                String line = scan.nextLine();
                // 在控制台输出网页资源内容
                System.out.println(line);
                // 将网页资源内容输出到指定输出流
                ps.println(line);
            }
        }
    }
    public static void main(String[] args)
```

```
        throws IOException , MalformedURLException
    {
        new ProxyTest().init();
    }
}
```

上面程序中第一行粗体字代码创建了一个 Proxy 对象，第二行粗体字代码就是用 Proxy 对象来打开 URLConnection 连接。接下来程序使用 URLConnection 读取了一份网络资源，此时的 URLConnection 并不是直接连接到 www.crazyit.org，而是通过代理服务器去访问该网站。

17.5.2 使用 ProxySelector 自动选择代理服务器

前面介绍的直接使用 Proxy 对象可以在打开 URLConnection 或 Socket 时指定代理服务器，但使用这种方式每次打开连接时都需要显式地设置代理服务器，比较麻烦。如果希望每次打开连接时总是具有默认的代理服务器，则可以借助于 ProxySelector 来实现。

ProxySelector 代表一个代理选择器，它本身是一个抽象类，程序无法创建它的实例，开发者可以考虑继承 ProxySelector 来实现自己的代理选择器。实现 ProxySelector 的步骤非常简单，程序只要定义一个继承 ProxySelector 的类，并让该类实现如下两个抽象方法。

- List<Proxy> select(URI uri)：根据业务需要返回代理服务器列表，如果该方法返回的集合中只包含一个 Proxy，该 Proxy 将会作为默认的代理服务器。
- connectFailed(URI uri, SocketAddress sa, IOException ioe)：连接代理服务器失败时回调该方法。

提示：

系统默认的代理服务器选择器也重写了 connectFailed 方法，它重写该方法的处理策略是：当系统设置的代理服务器失败时，默认代理选择器将会采用直连的方式连接远程资源，所以当运行上面程序等待了足够长时间时，程序依然可以打印出该远程资源的所有内容。

实现了自己的 ProxySelector 类之后，调用 ProxySelector 的 setDefault(ProxySelector ps)静态方法来注册该代理选择器即可。

下面程序示范了如何让自定义的 ProxySelector 来自动选择代理服务器。

程序清单：codes\17\17.5\ProxySelectorTest.java

```
public class ProxySelectorTest
{
    // 下面是代理服务器的地址和端口
    // 随便一个代理服务器的地址和端口
    final String PROXY_ADDR = "139.82.12.188";
    final int PROXY_PORT = 3124;
    // 定义需要访问的网站地址
    String urlStr = "http://www.crazyit.org";
    public void init()
        throws IOException , MalformedURLException
    {
        // 注册默认的代理选择器
        ProxySelector.setDefault(new ProxySelector()
        {
            @Override
            public void connectFailed(URI uri
                , SocketAddress sa, IOException ioe)
            {
                System.out.println("无法连接到指定代理服务器！");
            }
            // 根据业务需要返回特定的对应的代理服务器
            @Override
            public List<Proxy> select(URI uri)
            {
                // 本程序总是返回某个固定的代理服务器
                List<Proxy> result = new ArrayList<>();
                result.add(new Proxy(Proxy.Type.HTTP
                    , new InetSocketAddress(PROXY_ADDR , PROXY_PORT)));
                return result;
```

```
            }
        });
        URL url = new URL(urlStr);
        // 没有指定代理服务器，直接打开连接
        URLConnection conn = url.openConnection();    // ①
        ...
    }
```

上面程序的关键是粗体字代码部分采用匿名内部类实现了一个 ProxySelector，这个 ProxySelector 的 select()方法总是返回一个固定的代理服务器，也就是说，程序默认总会使用该代理服务器。因此程序在①号代码处打开连接时虽然没有指定代理服务器，但实际上程序依然会使用代理服务器——如果用户设置一个无效的代理服务器，系统将会在连接失败时回调 ProxySelector 的 connectFailed()方法，这可以说明代理选择器起作用了。

除此之外，Java 为 ProxySelector 提供了一个实现类：sun.net.spi.DefaultProxySelector（这是一个未公开 API，应尽量避免直接使用该 API），系统已经将 DefaultProxySelector 注册成默认的代理选择器，因此程序可调用 ProxySelector.getDefault()方法来获取 DefaultProxySelector 实例。

DefaultProxySelector 继承了 ProxySelector，当然也实现了两个抽象方法，它的实现策略如下。

- connectFailed()：如果连接失败，DefaultProxySelector 将会尝试不使用代理服务器，直接连接远程资源。
- select()：DefaultProxySelector 会根据系统属性来决定使用哪个代理服务器。ProxySelector 会检测系统属性与 URL 之间的匹配，然后决定使用相应的属性值作为代理服务器。关于代理服务器常用的属性名有如下三个。
  - http.proxyHost：设置 HTTP 访问所使用的代理服务器的主机地址。该属性名的前缀可以改为 https、ftp 等，分别用于设置 HTTPS 访问和 FTP 访问所用的代理服务器的主机地址。
  - http.proxyPort：设置 HTTP 访问所使用的代理服务器的端口。该属性名的前缀可以改为 https、ftp 等，分别用于设置 HTTPS 访问和 FTP 访问所用的代理服务器的端口。
  - http.nonProxyHosts：设置 HTTP 访问中不需要使用代理服务器的主机，支持使用*通配符；支持指定多个地址，多个地址之间用竖线（|）分隔。

下面程序示范了通过改变系统属性来改变默认的代理服务器。

程序清单：codes\17\17.5\DefaultProxySelectorTest.java

```
public class DefaultProxySelectorTest
{
    // 定义需要访问的网站地址
    static String urlStr = "http://www.crazyit.org";
    public static void main(String[] args) throws Exception
    {
        // 获取系统的默认属性
        Properties props = System.getProperties();
        // 通过系统属性设置 HTTP 访问所用的代理服务器的主机地址、端口
        props.setProperty("http.proxyHost", "192.168.10.96");
        props.setProperty("http.proxyPort", "8080");
        // 通过系统属性设置 HTTP 访问无须使用代理服务器的主机
        // 可以使用*通配符，多个地址用|分隔
        props.setProperty("http.nonProxyHosts", "localhost|192.168.10.*");
        // 通过系统属性设置 HTTPS 访问所用的代理服务器的主机地址、端口
        props.setProperty("https.proxyHost", "192.168.10.96");
        props.setProperty("https.proxyPort", "443");
        /* DefaultProxySelector 不支持 https.nonProxyHosts 属性
         DefaultProxySelector 直接按 http.nonProxyHosts 的设置规则处理 */
        // 通过系统属性设置 FTP 访问所用的代理服务器的主机地址、端口
        props.setProperty("ftp.proxyHost", "192.168.10.96");
        props.setProperty("ftp.proxyPort", "2121");
        // 通过系统属性设置 FTP 访问无须使用代理服务器的主机
        props.setProperty("ftp.nonProxyHosts", "localhost|192.168.10.*");
        // 通过系统属性设置 SOCKS 代理服务器的主机地址、端口
        props.setProperty("socks.ProxyHost", "192.168.10.96");
        props.setProperty("socks.ProxyPort", "1080");
```

```
        // 获取系统默认的代理选择器
        ProxySelector selector = ProxySelector.getDefault();   // ①
        System.out.println("系统默认的代理选择器: " + selector);
        // 根据 URI 动态决定所使用的代理服务器
        System.out.println("系统为 ftp://www.crazyit.org 选择的代理服务器为: "
            +ProxySelector.getDefault().select(new URI("ftp://www.crazyit.org")));// ②
        URL url = new URL(urlStr);
        // 直接打开连接，默认的代理选择器会使用 http.proxyHost、http.proxyPort 系统属性
        // 设置的代理服务器
        // 如果无法连接代理服务器，则默认的代理选择器会尝试直接连接
        URLConnection conn = url.openConnection();   // ③
        // 设置超时时长
        conn.setConnectTimeout(3000);
        try(
            Scanner scan = new Scanner(conn.getInputStream() , "utf-8"))
        {
            // 读取远程主机的内容
            while(scan.hasNextLine())
            {
                System.out.println(scan.nextLine());
            }
        }
    }
}
```

上面程序中①号粗体字代码返回了系统默认注册的ProxySelector，并返回DefaultProxySelector实例。程序中三行粗体字代码设置 HTTP 访问的代理服务器属性，其中前两行代码设置代理服务器的地址和端口，第三行代码设置 HTTP 访问哪些主机时不需要使用代理服务器。上面程序中③号代码处直接打开一个 URLConnection，系统会在打开该 URLConnection 时使用代理服务器。程序在②号代码处让默认的ProxySelector 为 ftp://www.crazyit.org 选择代理服务器，它将使用 ftp.proxyHost 属性设置的代理服务器。

运行上面程序，由于 192.168.0.96 通常并不是有效的代理服务器（如果读者运行的机器恰好可以使用 192.168.10.96:8080 的代理服务器，则另当别论），因此程序将会等待几秒钟——无法连接到指定的代理服务器——默认的代理选择器的 connectFailed()方法被回调，该方法会尝试不使用代理服务器，直接连接远程资源。

17.6 本章小结

本章重点介绍了 Java 网络编程的相关知识。本章还简要介绍了计算机网络的相关知识，并介绍了IP 地址和端口的概念，这是进行网络编程的基础。本章还介绍了 Java 提供的 InetAddress、URLEncoder、URLDecoder、URLConnection 等工具类的使用，并通过一个多线程下载工具详细介绍了如何使用 URL Connection 访问远程资源。

本章详细介绍了 ServerSocket 和 Socket 两个类，程序可以通过这两个类实现 TCP 服务器、TCP 客户端。本章除介绍 Java 传统的网络编程知识外，也介绍了 Java NIO 提供的非阻塞网络通信，并详细介绍了 Java 7 提供的 AIO 网络通信。本章还介绍了 Java 提供的 UDP 通信支持类：DatagramSocket、DatagramPacket 和 MulticastSocket，并通过一个局域网通信工具示范了如何利用它们开发实际的应用。本章最后介绍了如何利用 Proxy 和 ProxySelector 在程序中使用代理服务器。

➤➤ 本章练习

1. 开发仿 FlashGet 的断点续传、多线程下载工具。
2. 开发基于 C/S 结构的游戏大厅。
3. 扩展 LanTalk 开发局域网内的即时通信、数据传输工具。

CHAPTER

18

第 18 章
类加载机制与反射

本章要点

- 类加载
- 类连接的过程
- 类初始化的过程
- 类加载器以及实现机制
- 继承 ClassLoader 实现自定义类加载器
- 使用 URLClassLoader
- 使用 Class 对象
- Java 8 新增的方法参数反射
- 动态创建 Java 对象
- 动态调用方法
- 访问并修改 Java 对象的属性值
- 使用反射操作数组
- 使用 Proxy 和 InvocationHandler 创建动态代理
- AOP 入门
- Class 类的泛型
- 通过反射获取泛型类型

本章将会深入介绍 Java 类的加载、连接和初始化知识，并重点介绍 Java 反射的相关内容。读者在阅读本章的类加载、连接及初始化知识时，可能会感觉这些知识比较底层，但掌握这些底层的运行原理会让读者对 Java 程序的运行有更好的把握。而且 Java 类加载器除了根类加载器之外，其他类加载器都是使用 Java 语言编写的，所以程序员完全可以开发自己的类加载器，通过使用自定义类加载器，可以完成一些特定的功能。

本章将重点介绍 java.lang.reflect 包下的接口和类，包括 Class、Method、Field、Constructor 和 Array 等，这些类分别代表类、方法、成员变量、构造器和数组，Java 程序可以使用这些类动态地获取某个对象、某个类的运行时信息，并可以动态地创建 Java 对象，动态地调用 Java 方法，访问并修改指定对象的成员变量值。本章还将介绍该包下的 Type 和 ParameterizedType 两个接口，其中 Type 是 Class 类所实现的接口，而 ParameterizedType 则代表一个带泛型参数的类型。

本章将介绍使用 Proxy 和 InvocationHandler 来创建 JDK 动态代理，并会通过 JDK 动态代理向读者介绍高层次解耦的方法，还会讲解 JDK 动态代理和 AOP（Aspect Orient Programming，面向切面编程）之间的内在关系。

18.1 类的加载、连接和初始化

系统可能在第一次使用某个类时加载该类，也可能采用预加载机制来加载某个类。本节将会详细介绍类加载、连接和初始化过程中的每个细节。

18.1.1 JVM 和类

当调用 java 命令运行某个 Java 程序时，该命令将会启动一个 Java 虚拟机进程，不管该 Java 程序有多么复杂，该程序启动了多少个线程，它们都处于该 Java 虚拟机进程里。正如前面介绍的，同一个 JVM 的所有线程、所有变量都处于同一个进程里，它们都使用该 JVM 进程的内存区。当系统出现以下几种情况时，JVM 进程将被终止。

- 程序运行到最后正常结束。
- 程序运行到使用 System.exit()或 Runtime.getRuntime().exit()代码处结束程序。
- 程序执行过程中遇到未捕获的异常或错误而结束。
- 程序所在平台强制结束了 JVM 进程。

从上面的介绍可以看出，当 Java 程序运行结束时，JVM 进程结束，该进程在内存中的状态将会丢失。下面以类的类变量来说明这个问题。下面程序先定义了一个包含类变量的类。

程序清单：codes\18\18.1\A.java

```
public class A
{
    // 定义该类的类变量
    public static int a = 6;
}
```

上面程序中的粗体字代码定义了一个类变量 a，接下来定义一个类创建 A 类的实例，并访问 A 对象的类变量 a。

程序清单：codes\18\18.1\ATest1.java

```
public class ATest1
{
    public static void main(String[] args)
    {
        // 创建 A 类的实例
        A a = new A();
        // 让 a 实例的类变量 a 的值自加
        a.a ++;
        System.out.println(a.a);
    }
}
```

下面程序也创建 A 对象，并访问其类变量 a 的值。

程序清单：codes\18\18.1\ATest2.java

```
public class ATest2
{
    public static void main(String[] args)
    {
        // 创建 A 类的实例
        A b = new A();
        // 输出 b 实例的类变量 a 的值
        System.out.println(b.a);
    }
}
```

在 ATest1.java 程序中创建了 A 类的实例，并让该实例的类变量 a 的值自加，程序输出该实例的类变量 a 的值将看到 7，相信读者对这个答案没有疑问。关键是运行第二个程序 ATest2 时，程序再次创建了 A 对象，并输出 A 对象类变量的 a 的值，此时 a 的值是多少呢？结果依然是 6，并不是 7。这是因为运行 ATest1 和 ATest2 是两次运行 JVM 进程，第一次运行 JVM 结束后，它对 A 类所做的修改将全部丢失——第二次运行 JVM 时将再次初始化 A 类。

提示：

在疯狂软件教育中心见过一些学员，他们在回答这个问题时会毫不犹豫地说 7。他们认为 A 类里的 a 成员变量是静态变量（即类变量），同一个类的所有实例的静态变量共享同一块内存区，因为第一次运行时改变了第一个 A 实例的 a 变量，所以第二次运行时第二个 A 实例的 a 变量也将受到影响。实际上他们忘记了两次运行 Java 程序处于两个不同的 JVM 进程中，两个 JVM 之间并不会共享数据。

18.1.2 类的加载

当程序主动使用某个类时，如果该类还未被加载到内存中，则系统会通过加载、连接、初始化三个步骤来对该类进行初始化。如果没有意外，JVM 将会连续完成这三个步骤，所以有时也把这三个步骤统称为类加载或类初始化。

类加载指的是将类的 class 文件读入内存，并为之创建一个 java.lang.Class 对象，也就是说，当程序中使用任何类时，系统都会为之建立一个 java.lang.Class 对象。

提示：

前面介绍面向对象时提到：类是某一类对象的抽象，类是概念层次的东西。但不知道读者有没有想过：类也是一种对象。就像平常说概念主要用于定义、描述其他事物，但概念本身也是一种事物，那么概念本身也需要被描述——这有点像一个哲学命题。但事实就是这样，每个类是一批具有相同特征的对象的抽象（或者说概念），而系统中所有的类实际上也是实例，它们都是 java.lang.Class 的实例。

类的加载由类加载器完成，类加载器通常由 JVM 提供，这些类加载器也是前面所有程序运行的基础，JVM 提供的这些类加载器通常被称为系统类加载器。除此之外，开发者可以通过继承 ClassLoader 基类来创建自己的类加载器。

通过使用不同的类加载器，可以从不同来源加载类的二进制数据，通常有如下几种来源。

- 从本地文件系统加载 class 文件，这是前面绝大部分示例程序的类加载方式。
- 从 JAR 包加载 class 文件，这种方式也是很常见的，前面介绍 JDBC 编程时用到的数据库驱动类就放在 JAR 文件中，JVM 可以从 JAR 文件中直接加载该 class 文件。
- 通过网络加载 class 文件。
- 把一个 Java 源文件动态编译，并执行加载。

类加载器通常无须等到“首次使用”该类时才加载该类，Java 虚拟机规范允许系统预先加载某些类。

18.1.3 类的连接

当类被加载之后，系统为之生成一个对应的 Class 对象，接着将会进入连接阶段，连接阶段负责把类的二进制数据合并到 JRE 中。类连接又可分为如下三个阶段。

（1）验证：验证阶段用于检验被加载的类是否有正确的内部结构，并和其他类协调一致。

（2）准备：类准备阶段则负责为类的类变量分配内存，并设置默认初始值。

（3）解析：将类的二进制数据中的符号引用替换成直接引用。

18.1.4 类的初始化

在类的初始化阶段，虚拟机负责对类进行初始化，主要就是对类变量进行初始化。在 Java 类中对类变量指定初始值有两种方式：① 声明类变量时指定初始值；② 使用静态初始化块为类变量指定初始值。例如下面代码片段。

```
public class Test
{
    // 声明变量 a 时指定初始值
    static int a = 5;
    static int b;
    static int c;
    static
    {
        // 使用静态初始化块为变量 b 指定初始值
        b = 6;
    }
    ...
}
```

对于上面代码，程序为类变量 a、b 都显式指定了初始值，所以这两个类变量的值分别为 5、6，但类变量 c 则没有指定初始值，它将采用默认初始值 0。

声明变量时指定初始值，静态初始化块都将被当成类的初始化语句，JVM 会按这些语句在程序中的排列顺序依次执行它们，例如下面的类。

程序清单：codes\18\18.1\Test.java

```
public class Test
{
    static
    {
        // 使用静态初始化块为变量 b 指定初始值
        b = 6;
        System.out.println("----------");
    }
    // 声明变量 a 时指定初始值
    static int a = 5;
    static int b = 9;          // ①
    static int c;
    public static void main(String[] args)
    {
        System.out.println(Test.b);
    }
}
```

上面代码先在静态初始化块中为 b 变量赋值，此时类变量 b 的值为 6；接着程序向下执行，执行到①号代码处，这行代码也属于该类的初始化语句，所以程序再次为类变量 b 赋值。也就是说，当 Test 类初始化结束后，该类的类变量 b 的值为 9。

JVM 初始化一个类包含如下几个步骤。

1. 假如这个类还没有被加载和连接，则程序先加载并连接该类。
2. 假如该类的直接父类还没有被初始化，则先初始化其直接父类。
3. 假如类中有初始化语句，则系统依次执行这些初始化语句。

当执行第 2 个步骤时，系统对直接父类的初始化步骤也遵循此步骤 1~3；如果该直接父类又有直接

父类，则系统再次重复这三个步骤来先初始化这个父类……依此类推，所以 JVM 最先初始化的总是 java.lang.Object 类。当程序主动使用任何一个类时，系统会保证该类以及所有父类（包括直接父类和间接父类）都会被初始化。关于这一点请参考 5.9.3 节的内容。

18.1.5 类初始化的时机

当 Java 程序首次通过下面 6 种方式来使用某个类或接口时，系统就会初始化该类或接口。

- 创建类的实例。为某个类创建实例的方式包括：使用 new 操作符来创建实例，通过反射来创建实例，通过反序列化的方式来创建实例。
- 调用某个类的类方法（静态方法）。
- 访问某个类或接口的类变量，或为该类变量赋值。
- 使用反射方式来强制创建某个类或接口对应的 java.lang.Class 对象。例如代码：Class.forName("Person")，如果系统还未初始化 Person 类，则这行代码将会导致该 Person 类被初始化，并返回 Person 类对应的 java.lang.Class 对象。关于 Class 的 forName 方法请参考 18.3 节。
- 初始化某个类的子类。当初始化某个类的子类时，该子类的所有父类都会被初始化。
- 直接使用 java.exe 命令来运行某个主类。当运行某个主类时，程序会先初始化该主类。

除此之外，下面的几种情形需要特别指出。

对于一个 final 型的类变量，如果该类变量的值在编译时就可以确定下来，那么这个类变量相当于“宏变量”。Java 编译器会在编译时直接把这个类变量出现的地方替换成它的值，因此即使程序使用该静态类变量，也不会导致该类的初始化。例如下面示例程序的结果。

程序清单：codes\18\18.1\CompileConstantTest.java

```
class MyTest
{
    static
    {
        System.out.println("静态初始化块...");
    }
    // 使用一个字符串直接量为 static final 的类变量赋值
    static final String compileConstant = "疯狂 Java 讲义";
}
public class CompileConstantTest
{
    public static void main(String[] args)
    {
        // 访问、输出 MyTest 中的 compileConstant 类变量
        System.out.println(MyTest.compileConstant);   // ①
    }
}
```

上面程序的 MyTest 类中有一个 compileConstant 的类变量，该类变量使用了 final 修饰，而且它的值可以在编译时确定下来，因此 compileConstant 会被当成“宏变量”处理。程序中所有使用 compileConstant 的地方都会在编译时被直接替换成它的值——也就是说，上面程序中①处的粗体字代码在编译时就会被替换成“疯狂 Java 讲义”，所以①行代码不会导致初始化 MyTest 类。

提示： 当某个类变量（也叫静态变量）使用了 final 修饰，而且它的值可以在编译时就确定下来，那么程序其他地方使用该类变量时，实际上并没有使用该类变量，而是相当于使用常量。

反之，如果 final 修饰的类变量的值不能在编译时确定下来，则必须等到运行时才可以确定该类变量的值，如果通过该类来访问它的类变量，则会导致该类被初始化。例如将上面程序中定义 compileConstant 的代码改为如下：

```
// 采用系统当前时间为 static final 类变量赋值
static final String compileConstant =
    System.currentTimeMillis() + "";
```

因为上面定义的 compileConstant 类变量的值必须在运行时才可以确定，所以①处的粗体字代码必须保留为对 MyTest 类的类变量的引用，这行代码就变成了使用 MyTest 的类变量，这将导致 MyTest 类被初始化。

当使用 ClassLoader 类的 loadClass()方法来加载某个类时，该方法只是加载该类，并不会执行该类的初始化。使用 Class 的 forName()静态方法才会导致强制初始化该类。例如如下代码。

程序清单：codes\18\18.1\ClassLoaderTest.java

```
class Tester
{
    static
    {
        System.out.println("Tester 类的静态初始化块...");
    }
}
public class ClassLoaderTest
{
    public static void main(String[] args)
        throws ClassNotFoundException
    {
        ClassLoader cl = ClassLoader.getSystemClassLoader();
        // 下面语句仅仅是加载 Tester 类
        cl.loadClass("Tester");
        System.out.println("系统加载 Tester 类");
        // 下面语句才会初始化 Tester 类
        Class.forName("Tester");
    }
}
```

上面程序中的两行粗体字代码都用到了 Tester 类，但第一行粗体字代码只是加载 Tester 类，并不会初始化 Tester 类。运行上面程序，会看到如下运行结果：

```
系统加载 Tester 类
Tester 类的静态初始化块...
```

从上面运行结果可以看出，必须等到执行 Class.forName("Tester")时才完成对 Tester 类的初始化。

18.2 类加载器

类加载器负责将.class 文件（可能在磁盘上，也可能在网络上）加载到内存中，并为之生成对应的 java.lang.Class 对象。尽管在 Java 开发中无须过分关心类加载机制，但所有的编程人员都应该了解其工作机制，明白如何做才能让其更好地满足我们的需要。

18.2.1 类加载机制

类加载器负责加载所有的类，系统为所有被载入内存中的类生成一个 java.lang.Class 实例。一旦一个类被载入 JVM 中，同一个类就不会被再次载入了。现在的问题是，怎么样才算“同一个类”？

正如一个对象有一个唯一的标识一样，一个载入 JVM 中的类也有一个唯一的标识。在 Java 中，一个类用其全限定类名（包括包名和类名）作为标识；但在 JVM 中，一个类用其全限定类名和其类加载器作为唯一标识。例如，如果在 pg 的包中有一个名为 Person 的类，被类加载器 ClassLoader 的实例 kl 负责加载，则该 Person 类对应的 Class 对象在 JVM 中表示为（Person、pg、kl）。这意味着两个类加载器加载的同名类：（Person、pg、kl）和（Person、pg、kl2）是不同的，它们所加载的类也是完全不同、互不兼容的。

当 JVM 启动时，会形成由三个类加载器组成的初始类加载器层次结构。

- Bootstrap ClassLoader：根类加载器。
- Extension ClassLoader：扩展类加载器。
- System ClassLoader：系统类加载器。

Bootstrap ClassLoader 被称为引导（也称为原始或根）类加载器，它负责加载 Java 的核心类。在 Sun 的 JVM 中，当执行 java.exe 命令时，使用-Xbootclasspath 或-D 选项指定 sun.boot.class.path 系统属性值

可以指定加载附加的类。

JVM 的类加载机制主要有如下三种。

- 全盘负责。所谓全盘负责，就是当一个类加载器负责加载某个 Class 时，该 Class 所依赖的和引用的其他 Class 也将由该类加载器负责载入，除非显式使用另外一个类加载器来载入。
- 父类委托。所谓父类委托，则是先让 parent（父）类加载器试图加载该 Class，只有在父类加载器无法加载该类时才尝试从自己的类路径中加载该类。
- 缓存机制。缓存机制将会保证所有加载过的 Class 都会被缓存，当程序中需要使用某个 Class 时，类加载器先从缓存区中搜寻该 Class，只有当缓存区中不存在该 Class 对象时，系统才会读取该类对应的二进制数据，并将其转换成 Class 对象，存入缓存区中。这就是为什么修改了 Class 后，必须重新启动 JVM，程序所做的修改才会生效的原因。

注意：

类加载器之间的父子关系并不是类继承上的父子关系，这里的父子关系是类加载器实例之间的关系。

除了可以使用 Java 提供的类加载器之外，开发者也可以实现自己的类加载器，自定义的类加载器通过继承 ClassLoader 来实现。JVM 中这 4 种类加载器的层次结构如图 18.1 所示。

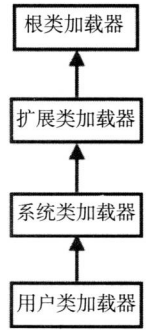


图 18.1　JVM 中 4 种类加载器的层次结构

下面程序示范了访问 JVM 的类加载器。

程序清单：codes\18\18.2\ClassLoaderPropTest.java

```
public class ClassLoaderPropTest
{
    public static void main(String[] args)
        throws IOException
    {
        // 获取系统类加载器
        ClassLoader systemLoader = ClassLoader.getSystemClassLoader();
        System.out.println("系统类加载器: " + systemLoader);
        /*
        获取系统类加载器的加载路径——通常由 CLASSPATH 环境变量指定
        如果操作系统没有指定 CLASSPATH 环境变量，则默认以当前路径作为
        系统类加载器的加载路径
        */
        Enumeration<URL> em1 = systemLoader.getResources("");
        while(em1.hasMoreElements())
        {
            System.out.println(em1.nextElement());
        }
        // 获取系统类加载器的父类加载器，得到扩展类加载器
        ClassLoader extensionLader = systemLoader.getParent();
        System.out.println("扩展类加载器: " + extensionLader);
        System.out.println("扩展类加载器的加载路径: "
            + System.getProperty("java.ext.dirs"));
        System.out.println("扩展类加载器的 parent: "
            + extensionLader.getParent());
    }
}
```

运行上面程序，会看到如下运行结果：

```
系统类加载器: jdk.internal.loader.ClassLoaders$AppClassLoader@726f3b58
ile:/G:/publish/codes/18/18.2/
扩展类加载器: jdk.internal.loader.ClassLoaders$PlatformClassLoader@e6ea0c6
扩展类加载器的加载路径: null
扩展类加载器的 parent: null
```

从上面运行结果可以看出，系统类加载器的加载路径是程序运行的当前路径，扩展类加载器的加载路径是 null（与 Java 8 有区别），但此处看到扩展类加载器的父加载器是 null，并不是根类加载器。这

是因为根类加载器并没有继承 ClassLoader 抽象类，所以扩展类加载器的 getParent()方法返回 null。但实际上，扩展类加载器的父类加载器是根类加载器，只是根类加载器并不是 Java 实现的。

从运行结果可以看出，系统类加载器是 AppClassLoader 的实例，扩展类加载器 PlatformClassLoader 的实例。实际上，这两个类都是 URLClassLoader 类的实例。

注意：

JVM 的根类加载器并不是 Java 实现的，而且由于程序通常无须访问根类加载器，因此访问扩展类加载器的父类加载器时返回 null。

类加载器加载 Class 大致要经过如下 8 个步骤。

① 检测此 Class 是否载入过（即在缓存区中是否有此 Class），如果有则直接进入第 8 步，否则接着执行第 2 步。

② 如果父类加载器不存在（如果没有父类加载器，则要么 parent 一定是根类加载器，要么本身就是根类加载器），则跳到第 4 步执行；如果父类加载器存在，则接着执行第 3 步。

③ 请求使用父类加载器去载入目标类，如果成功载入则跳到第 8 步，否则接着执行第 5 步。

④ 请求使用根类加载器来载入目标类，如果成功载入则跳到第 8 步，否则跳到第 7 步。

⑤ 当前类加载器尝试寻找 Class 文件（从与此 ClassLoader 相关的类路径中寻找），如果找到则执行第 6 步，如果找不到则跳到第 7 步。

⑥ 从文件中载入 Class，成功载入后跳到第 8 步。

⑦ 抛出 ClassNotFoundException 异常。

⑧ 返回对应的 java.lang.Class 对象。

其中，第5、6步允许重写 ClassLoader 的 findClass()方法来实现自己的载入策略，甚至重写 loadClass()方法来实现自己的载入过程。

18.2.2 创建并使用自定义的类加载器

JVM 中除根类加载器之外的所有类加载器都是 ClassLoader 子类的实例，开发者可以通过扩展 ClassLoader 的子类，并重写该 ClassLoader 所包含的方法来实现自定义的类加载器。查阅 API 文档中关于 ClassLoader 的方法不难发现，ClassLoader 中包含了大量的 protected 方法——这些方法都可被子类重写。

ClassLoader 类有如下两个关键方法。

- loadClass(String name, boolean resolve)：该方法为 ClassLoader 的入口点，根据指定名称来加载类，系统就是调用 ClassLoader 的该方法来获取指定类对应的 Class 对象。
- findClass(String name)：根据指定名称来查找类。

如果需要实现自定义的 ClassLoader，则可以通过重写以上两个方法来实现，通常推荐重写 findClass()方法，而不是重写 loadClass()方法。loadClass()方法的执行步骤如下。

① 用 findLoadedClass(String) 来检查是否已经加载类，如果已经加载则直接返回。

② 在父类加载器上调用 loadClass()方法。如果父类加载器为 null，则使用根类加载器来加载。

③ 调用 findClass(String)方法查找类。

从上面步骤中可以看出，重写 findClass()方法可以避免覆盖默认类加载器的父类委托、缓冲机制两种策略；如果重写 loadClass()方法，则实现逻辑更为复杂。

在 ClassLoader 里还有一个核心方法：Class defineClass(String name, byte[] b, int off, int len)，该方法负责将指定类的字节码文件（即 Class 文件，如 Hello.class）读入字节数组 byte[] b 内，并把它转换为 Class 对象，该字节码文件可以来源于文件、网络等。

defineClass()方法管理 JVM 的许多复杂的实现，它负责将字节码分析成运行时数据结构，并校验有效性等。不过不用担心，程序员无须重写该方法。实际上该方法是 final 的，即使想重写也没有机会。

除此之外，ClassLoader 里还包含如下一些普通方法。

- findSystemClass(String name)：从本地文件系统装入文件。它在本地文件系统中寻找类文件，如

果存在，就使用 defineClass()方法将原始字节转换成 Class 对象，以将该文件转换成类。

- static getSystemClassLoader()：这是一个静态方法，用于返回系统类加载器。
- getParent()：获取该类加载器的父类加载器。
- resolveClass(Class<?> c)：链接指定的类。类加载器可以使用此方法来链接类 c。读者无须理会关于此方法的太多细节。
- findLoadedClass(String name)：如果此 Java 虚拟机已加载了名为 name 的类，则直接返回该类对应的 Class 实例，否则返回 null。该方法是 Java 类加载缓存机制的体现。

下面程序开发了一个自定义的 ClassLoader，该 ClassLoader 通过重写 findClass()方法来实现自定义的类加载机制。这个 ClassLoader 可以在加载类之前先编译该类的源文件，从而实现运行 Java 之前先编译该程序的目标，这样即可通过该 ClassLoader 直接运行 Java 源文件。

程序清单：codes\18\18.2\CompileClassLoader.java

```
public class CompileClassLoader extends ClassLoader
{
    // 读取一个文件的内容
    private byte[] getBytes(String filename)
        throws IOException
    {
        File file = new File(filename);
        long len = file.length();
        byte[] raw = new byte[(int)len];
        try(
            FileInputStream fin = new FileInputStream(file))
        {
            // 一次读取 Class 文件的全部二进制数据
            int r = fin.read(raw);
            if(r != len)
            throw new IOException("无法读取全部文件："
                + r + " != " + len);
            return raw;
        }
    }
    // 定义编译指定 Java 文件的方法
    private boolean compile(String javaFile)
        throws IOException
    {
        System.out.println("CompileClassLoader:正在编译 "
            + javaFile + "...");
        // 调用系统的 javac 命令
        Process p = Runtime.getRuntime().exec("javac " + javaFile);
        try
        {
            // 其他线程都等待这个线程完成
            p.waitFor();
        }
        catch(InterruptedException ie)
        {
            System.out.println(ie);
        }
        // 获取 javac 线程的退出值
        int ret = p.exitValue();
        // 返回编译是否成功
        return ret == 0;
    }
    // 重写 ClassLoader 的 findClass 方法
    protected Class<?> findClass(String name)
        throws ClassNotFoundException
    {
        Class clazz = null;
        // 将包路径中的点（.）替换成斜线（/）
        String fileStub = name.replace("." , "/");
        String javaFilename = fileStub + ".java";
        String classFilename = fileStub + ".class";
```

```
        File javaFile = new File(javaFilename);
        File classFile = new File(classFilename);
        // 当指定 Java 源文件存在，且 Class 文件不存在，或者 Java 源文件
        // 的修改时间比 Class 文件的修改时间更晚时，重新编译
        if(javaFile.exists() && (!classFile.exists()
            || javaFile.lastModified() > classFile.lastModified()))
        {
            try
            {
                // 如果编译失败，或者该 Class 文件不存在
                if(!compile(javaFilename) || !classFile.exists())
                {
                    throw new ClassNotFoundException(
                        "ClassNotFoundExcetpion:" + javaFilename);
                }
            }
            catch (IOException ex)
            {
                ex.printStackTrace();
            }
        }
        // 如果 Class 文件存在，系统负责将该文件转换成 Class 对象
        if (classFile.exists())
        {
            try
            {
                // 将 Class 文件的二进制数据读入数组
                byte[] raw = getBytes(classFilename);
                // 调用 ClassLoader 的 defineClass 方法将二进制数据转换成 Class 对象
                clazz = defineClass(name,raw,0,raw.length);
            }
            catch(IOException ie)
            {
                ie.printStackTrace();
            }
        }
        // 如果 clazz 为 null，表明加载失败，则抛出异常
        if(clazz == null)
        {
            throw new ClassNotFoundException(name);
        }
        return clazz;
    }
    // 定义一个主方法
    public static void main(String[] args) throws Exception
    {
        // 如果运行该程序时没有参数，即没有目标类
        if (args.length < 1)
        {
            System.out.println("缺少目标类，请按如下格式运行 Java 源文件：");
            System.out.println("java CompileClassLoader ClassName");
        }
        // 第一个参数是需要运行的类
        String progClass = args[0];
        // 剩下的参数将作为运行目标类时的参数
        // 将这些参数复制到一个新数组中
        String[] progArgs = new String[args.length-1];
        System.arraycopy(args , 1 , progArgs
            , 0 , progArgs.length);
        CompileClassLoader ccl = new CompileClassLoader();
        // 加载需要运行的类
        Class<?> clazz = ccl.loadClass(progClass);
        // 获取需要运行的类的主方法
        Method main = clazz.getMethod("main" , (new String[0]).getClass());
        Object argsArray[] = {progArgs};
        main.invoke(null,argsArray);
    }
}
```

上面程序中的粗体字代码重写了 findClass()方法，通过重写该方法就可以实现自定义的类加载机制。在本类的 findClass()方法中先检查需要加载类的 Class 文件是否存在，如果不存在则先编译源文件，再调用 ClassLoader 的 defineClass()方法来加载这个 Class 文件，并生成相应的 Class 对象。

提示：

上面程序的 main()方法中的粗体字代码使用了反射来调用方法，关于使用反射调用方法的内容请参考本章 18.4 节的内容。

接下来可以随意提供一个简单的主类，该主类无须编译就可以使用上面的 CompileClass Loader 来运行它。

程序清单：codes\18\18.2\Hello.java

```
public class Hello
{
    public static void main(String[] args)
    {
        for (String arg : args)
        {
            System.out.println("运行 Hello 的参数：" + arg);
        }
    }
}
```

无须编译该 Hello.java，可以直接使用如下命令来运行该 Hello.java 程序。

```
java CompileClassLoader Hello 疯狂 Java 讲义
```

运行结果如下：

```
CompileClassLoader:正在编译 Hello.java...
运行 Hello 的参数：疯狂 Java 讲义
```

本示例程序提供的类加载器功能比较简单，仅仅提供了在运行之前先编译 Java 源文件的功能。实际上，使用自定义的类加载器，可以实现如下常见功能。

- 执行代码前自动验证数字签名。
- 根据用户提供的密码解密代码，从而可以实现代码混淆器来避免反编译*.class 文件。
- 根据用户需求来动态地加载类。
- 根据应用需求把其他数据以字节码的形式加载到应用中。

18.2.3 URLClassLoader 类

Java 为 ClassLoader 提供了一个 URLClassLoader 实现类，该类也是系统类加载器和扩展类加载器的父类（此处的父类，就是指类与类之间的继承关系）。URLClassLoader 功能比较强大，它既可以从本地文件系统获取二进制文件来加载类，也可以从远程主机获取二进制文件来加载类。

在应用程序中可以直接使用 URLClassLoader 加载类，URLClassLoader 类提供了如下两个构造器。

- URLClassLoader(URL[] urls)：使用默认的父类加载器创建一个 ClassLoader 对象，该对象将从 urls 所指定的系列路径来查询并加载类。
- URLClassLoader(URL[] urls, ClassLoader parent)：使用指定的父类加载器创建一个 ClassLoader 对象，其他功能与前一个构造器相同。

一旦得到了 URLClassLoader 对象之后，就可以调用该对象的 loadClass()方法来加载指定类。下面程序示范了如何直接从文件系统中加载 MySQL 驱动，并使用该驱动来获取数据库连接。通过这种方式来获取数据库连接，可以无须将 MySQL 驱动添加到 CLASSPATH 环境变量中。

程序清单：codes\18\18.2\URLClassLoaderTest.java

```
public class URLClassLoaderTest
{
    private static Connection conn;
    // 定义一个获取数据库连接的方法
```

```
    public static Connection getConn(String url ,
        String user , String pass) throws Exception
    {
        if (conn == null)
        {
            // 创建一个URL数组
            URL[] urls = {new URL(
                "file:mysql-connector-java-5.1.30-bin.jar")};
            // 以默认的ClassLoader作为父ClassLoader，创建URLClassLoader
            URLClassLoader myClassLoader = new URLClassLoader(urls);
            // 加载MySQL的JDBC驱动，并创建默认实例
            Driver driver = (Driver)myClassLoader.
                loadClass("com.mysql.jdbc.Driver").getConstructor().newInstance();
            // 创建一个设置JDBC连接属性的Properties对象
            Properties props = new Properties();
            // 至少需要为该对象传入user和password两个属性
            props.setProperty("user" , user);
            props.setProperty("password" , pass);
            // 调用Driver对象的connect方法来取得数据库连接
            conn = driver.connect(url , props);
        }
        return conn;
    }
    public static void main(String[] args)throws Exception
    {
        System.out.println(getConn("jdbc:mysql://localhost:3306/mysql"
            , "root" , "32147"));
    }
}
```

上面程序中的前两行粗体字代码创建了一个URLClassLoader对象，该对象使用默认的父类加载器，该类加载器的类加载路径是当前路径下的mysql-connector-java-5.1.30-bin.jar文件，将MySQL驱动复制到该路径下，这样保证该ClassLoader可以正常加载到com.mysql.jdbc.Driver类。

程序的第三行粗体字代码使用ClassLoader的loadClass()加载指定类，并调用Class对象的newInstance()方法创建了一个该类的默认实例——也就是得到com.mysql.jdbc.Driver类的对象，当然该对象的实现类实现了java.sql.Driver接口，所以程序将其强制类型转换为Driver。程序的最后一行粗体字代码通过Driver而不是DriverManager来获取数据库连接，关于Driver接口的用法读者可以自行查阅API文档。

正如前面所看到的，创建URLClassLoader时传入了一个URL数组参数，该ClassLoader就可以从这系列URL指定的资源中加载指定类，这里的URL可以以file:为前缀，表明从本地文件系统加载；可以以http:为前缀，表明从互联网通过HTTP访问来加载；也可以以ftp:为前缀，表明从互联网通过FTP访问来加载……功能非常强大。

18.3 通过反射查看类信息

Java程序中的许多对象在运行时都会出现两种类型：编译时类型和运行时类型，例如代码：Person p = new Student();，这行代码将会生成一个p变量，该变量的编译时类型为Person，运行时类型为Student；除此之外，还有更极端的情形，程序在运行时接收到外部传入的一个对象，该对象的编译时类型是Object，但程序又需要调用该对象运行时类型的方法。

为了解决这些问题，程序需要在运行时发现对象和类的真实信息。解决该问题有以下两种做法。

- 第一种做法是假设在编译时和运行时都完全知道类型的具体信息，在这种情况下，可以先使用instanceof运算符进行判断，再利用强制类型转换将其转换成其运行时类型的变量即可。关于这种方式请参考5.7节的内容。
- 第二种做法是编译时根本无法预知该对象和类可能属于哪些类，程序只依靠运行时信息来发现该对象和类的真实信息，这就必须使用反射。

18.3.1 获得 Class 对象

前面已经介绍过了，每个类被加载之后，系统就会为该类生成一个对应的 Class 对象，通过该 Class 对象就可以访问到 JVM 中的这个类。在 Java 程序中获得 Class 对象通常有如下三种方式。

- 使用 Class 类的 forName(String clazzName)静态方法。该方法需要传入字符串参数，该字符串参数的值是某个类的全限定类名（必须添加完整包名）。
- 调用某个类的 class 属性来获取该类对应的 Class 对象。例如，Person.class 将会返回 Person 类对应的 Class 对象。
- 调用某个对象的 getClass()方法。该方法是 java.lang.Object 类中的一个方法，所以所有的 Java 对象都可以调用该方法，该方法将会返回该对象所属类对应的 Class 对象。

对于第一种方式和第二种方式都是直接根据类来取得该类的 Class 对象，相比之下，第二种方式有如下两种优势。

- 代码更安全。程序在编译阶段就可以检查需要访问的 Class 对象是否存在。
- 程序性能更好。因为这种方式无须调用方法，所以性能更好。

也就是说，大部分时候都应该使用第二种方式来获取指定类的 Class 对象。但如果程序只能获得一个字符串，例如"java.lang.String"，若需要获取该字符串对应的 Class 对象，则只能使用第一种方式，使用 Class 的 forName(String clazzName)方法获取 Class 对象时，该方法可能抛出一个 ClassNotFoundException 异常。

一旦获得了某个类所对应的 Class 对象之后，程序就可以调用 Class 对象的方法来获得该对象和该类的真实信息了。

18.3.2 从 Class 中获取信息

Class 类提供了大量的实例方法来获取该 Class 对象所对应类的详细信息，Class 类大致包含如下方法，下面每个方法都可能包括多个重载的版本，读者应该查阅 API 文档来掌握它们。

下面 4 个方法用于获取 Class 对应类所包含的构造器。

- Connstructor<T> getConstructor(Class<?>... parameterTypes)：返回此 Class 对象对应类的、带指定形参列表的 public 构造器。
- Constructor<?>[] getConstructors()：返回此 Class 对象对应类的所有 public 构造器。
- Constructor<T> getDeclaredConstructor(Class<?>... parameterTypes)：返回此 Class 对象对应类的、带指定形参列表的构造器，与构造器的访问权限无关。
- Constructor<?>[] getDeclaredConstructors()：返回此 Class 对象对应类的所有构造器，与构造器的访问权限无关。

下面 4 个方法用于获取 Class 对应类所包含的方法。

- Method getMethod(String name, Class<?>... parameterTypes)：返回此 Class 对象对应类的、带指定形参列表的 public 方法。
- Method[] getMethods()：返回此 Class 对象所表示的类的所有 public 方法。
- Method getDeclaredMethod(String name, Class<?>... parameterTypes)：返回此 Class 对象对应类的、带指定形参列表的方法，与方法的访问权限无关。
- Method[] getDeclaredMethods()：返回此 Class 对象对应类的全部方法，与方法的访问权限无关。

如下 4 个方法用于访问 Class 对应类所包含的成员变量。

- Field getField(String name)：返回此 Class 对象对应类的、指定名称的 public 成员变量。
- Field[] getFields()：返回此 Class 对象对应类的所有 public 成员变量。
- Field getDeclaredField(String name)：返回此 Class 对象对应类的、指定名称的成员变量，与成员变量的访问权限无关。
- Field[] getDeclaredFields()：返回此 Class 对象对应类的全部成员变量，与成员变量的访问权限无关。

如下几个方法用于访问 Class 对应类上所包含的 Annotation。

- <A extends Annotation> A getAnnotation(Class<A> annotationClass)：尝试获取该 Class 对象对应类上存在的、指定类型的 Annotation；如果该类型的注解不存在，则返回 null。
- <A extends Annotation> A getDeclaredAnnotation(Class<A> annotationClass)：这是 Java 8 新增的方法，该方法尝试获取直接修饰该 Class 对象对应类的、指定类型的 Annotation；如果该类型的注解不存在，则返回 null。
- Annotation[] getAnnotations()：返回修饰该 Class 对象对应类上存在的所有 Annotation。
- Annotation[] getDeclaredAnnotations()：返回直接修饰该 Class 对应类的所有 Annotation。
- <A extends Annotation> A[] getAnnotationsByType(Class<A> annotationClass)：该方法的功能与前面介绍的 getAnnotation()方法基本相似。但由于 Java 8 增加了重复注解功能，因此需要使用该方法获取修饰该类的、指定类型的多个 Annotation。
- <A extends Annotation> A[] getDeclaredAnnotationsByType(Class<A> annotationClass)：该方法的功能与前面介绍的 getDeclaredAnnotations ()方法基本相似。但由于 Java 8 增加了重复注解功能，因此需要使用该方法获取直接修饰该类的、指定类型的多个 Annotation。

如下方法用于访问该 Class 对象对应类包含的内部类。

- Class<?>[] getDeclaredClasses()：返回该 Class 对象对应类里包含的全部内部类。

如下方法用于访问该 Class 对象对应类所在的外部类。

- Class<?> getDeclaringClass()：返回该 Class 对象对应类所在的外部类。

如下方法用于访问该 Class 对象对应类所实现的接口。

- Class<?>[] getInterfaces()：返回该 Class 对象对应类所实现的全部接口。

如下几个方法用于访问该 Class 对象对应类所继承的父类。

- Class<? super T> getSuperclass()：返回该 Class 对象对应类的超类的 Class 对象。

如下方法用于获取 Class 对象对应类的修饰符、所在包、类名等基本信息。

- int getModifiers()：返回此类或接口的所有修饰符。修饰符由 public、protected、private、final、static、abstract 等对应的常量组成，返回的整数应使用 Modifier 工具类的方法来解码，才可以获取真实的修饰符。
- Package getPackage()：获取此类的包。
- String getName()：以字符串形式返回此 Class 对象所表示的类的名称。
- String getSimpleName()：以字符串形式返回此 Class 对象所表示的类的简称。

除此之外，Class 对象还可调用如下几个判断方法来判断该类是否为接口、枚举、注解类型等。

- boolean isAnnotation()：返回此 Class 对象是否表示一个注解类型（由@interface 定义）。
- boolean isAnnotationPresent(Class<? extends Annotation> annotationClass)：判断此 Class 对象是否使用了 Annotation 修饰。
- boolean isAnonymousClass()：返回此 Class 对象是否是一个匿名类。
- boolean isArray()：返回此 Class 对象是否表示一个数组类。
- boolean isEnum()：返回此 Class 对象是否表示一个枚举（由 enum 关键字定义）。
- boolean isInterface()：返回此 Class 对象是否表示一个接口（使用 interface 定义）。
- boolean isInstance(Object obj)：判断 obj 是否是此 Class 对象的实例，该方法可以完全代替 instanceof 操作符。

上面的多个 getMethod()方法和 getConstructor()方法中，都需要传入多个类型为 Class<?>的参数，用于获取指定的方法或指定的构造器。关于这个参数的作用，假设某个类内包含如下三个 info 方法签名。

- public void info()
- public void info(String str)
- public void info(String str , Integer num)

这三个同名方法属于重载，它们的方法名相同，但参数列表不同。在 Java 语言中要确定一个方法光有方法名是不行的，如果仅仅只指定 info 方法——实际上可以是上面三个方法中的任意一个！如果需要确定一个方法，则应该由方法名和形参列表来确定，但形参名没有任何实际意义，所以只能由形参类型来确定。例如想指定第二个 info 方法，则必须指定方法名为 info，形参列表为 String.class——因此在程

序中获取该方法使用如下代码：

```
// 前一个参数指定方法名，后面的个数可变的 Class 参数指定形参类型列表
clazz.getMethod("info" , String.class)
```

如果需要获取第三个 info 方法，则使用如下代码：

```
// 前一个参数指定方法名，后面的个数可变的 Class 参数指定形参类型列表
clazz.getMethod("info" , String.class, Integer.class)
```

获取构造器时无须传入构造器名——同一个类的所有构造器的名字都是相同的，所以要确定一个构造器只要指定形参列表即可。

下面程序示范了如何通过该 Class 对象来获取对应类的详细信息。

程序清单：codes\18\18.3\ClassTest.java

```
// 定义可重复注解
@Repeatable(Annos.class)
@interface Anno {}
@Retention(value=RetentionPolicy.RUNTIME)
@interface Annos {
    Anno[] value();
}
// 使用 4 个注解修饰该类
@SuppressWarnings(value="unchecked")
@Deprecated
// 使用重复注解修饰该类
@Anno
@Anno
public class ClassTest
{
    // 为该类定义一个私有的构造器
    private ClassTest()
    {
    }
    // 定义一个有参数的构造器
    public ClassTest(String name)
    {
        System.out.println("执行有参数的构造器");
    }
    // 定义一个无参数的 info 方法
    public void info()
    {
        System.out.println("执行无参数的 info 方法");
    }
    // 定义一个有参数的 info 方法
    public void info(String str)
    {
        System.out.println("执行有参数的 info 方法"
            + ", 其 str 参数值: " + str);
    }
    // 定义一个测试用的内部类
    class Inner
    {
    }
    public static void main(String[] args)
        throws Exception
    {
        // 下面代码可以获取 ClassTest 对应的 Class
        Class<ClassTest> clazz = ClassTest.class;
        // 获取该 Class 对象所对应类的全部构造器
        Constructor[] ctors = clazz.getDeclaredConstructors();
        System.out.println("ClassTest 的全部构造器如下: ");
        for (Constructor c : ctors)
        {
            System.out.println(c);
        }
        // 获取该 Class 对象所对应类的全部 public 构造器
        Constructor[] publicCtors = clazz.getConstructors();
        System.out.println("ClassTest 的全部 public 构造器如下: ");
```

```
        for (Constructor c : publicCtors)
        {
            System.out.println(c);
        }
        // 获取该 Class 对象所对应类的全部 public 方法
        Method[] mtds = clazz.getMethods();
        System.out.println("ClassTest 的全部 public 方法如下：");
        for (Method md : mtds)
        {
            System.out.println(md);
        }
        // 获取该 Class 对象所对应类的指定方法
        System.out.println("ClassTest 里带一个字符串参数的 info 方法为："
            + clazz.getMethod("info" , String.class));
        // 获取该 Class 对象所对应类的全部注解
        Annotation[] anns = clazz.getAnnotations();
        System.out.println("ClassTest 的全部 Annotation 如下：");
        for (Annotation an : anns)
        {
            System.out.println(an);
        }
        System.out.println("该 Class 元素上的@SuppressWarnings 注解为："
            + Arrays.toString(clazz.getAnnotationsByType(SuppressWarnings.class)));
        System.out.println("该 Class 元素上的@Anno 注解为："
            + Arrays.toString(clazz.getAnnotationsByType(Anno.class)));
        // 获取该 Class 对象所对应类的全部内部类
        Class<?>[] inners = clazz.getDeclaredClasses();
        System.out.println("ClassTest 的全部内部类如下：");
        for (Class c : inners)
        {
            System.out.println(c);
        }
        // 使用 Class.forName()方法加载 ClassTest 的 Inner 内部类
        Class inClazz = Class.forName("ClassTest$Inner");
        // 通过 getDeclaringClass()访问该类所在的外部类
        System.out.println("inClazz 对应类的外部类为：" +
            inClazz.getDeclaringClass());
        System.out.println("ClassTest 的包为：" + clazz.getPackage());
        System.out.println("ClassTest 的父类为：" + clazz.getSuperclass());
    }
}
```

上面程序无须过多解释，程序获取了 ClassTest 类对应的 Class 对象后，通过调用该 Class 对象的不同方法来得到该 Class 对象的详细信息。运行该程序，会看到如图 18.2 所示的运行结果。

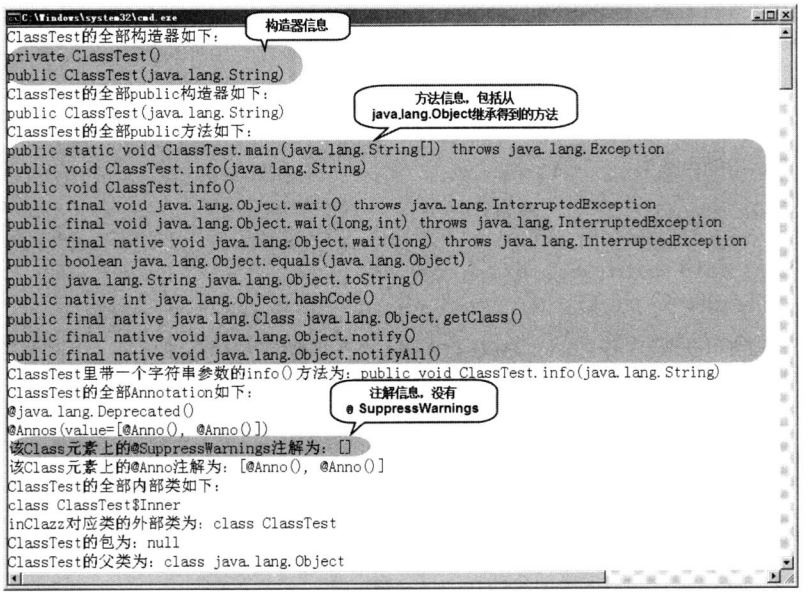


图 18.2　使用 Class 对象查看该类的内部信息

从图 18.2 所示的运行结果来看，Class 提供的功能非常丰富，它可以获取该类里包含的构造器、方法、内部类、注解等信息，也可以获取该类所包括的成员变量（Field）信息——通过 getFields()或 getField(String name)方法即可。

值得指出的是，虽然定义 ClassTest 类时使用了@SuppressWarnings 注解，但程序运行时无法分析出该类里包含的该注解，这是因为@SuppressWarnings 使用了@Retention(value=SOURCE)修饰，这表明@SuppressWarnings 只能保存在源代码级别上，而通过 ClassTest.class 获取该类的运行时 Class 对象，所以程序无法访问到@SuppressWarnings 注解。

注意：

对于只能在源代码上保留的注解，使用运行时获得的 Class 对象无法访问到该注解对象。

通过 Class 对象可以得到大量的 Method、Constructor、Field 等对象，这些对象分别代表该类所包括的方法、构造器和成员变量等，程序还可以通过这些对象来执行实际的功能，例如调用方法、创建实例。

18.3.3 Java 8 新增的方法参数反射

Java 8 在 java.lang.reflect 包下新增了一个 Executable 抽象基类，该对象代表可执行的类成员，该类派生了 Constructor、Method 两个子类。

Executable 基类提供了大量方法来获取修饰该方法或构造器的注解信息；还提供了 isVarArgs()方法用于判断该方法或构造器是否包含数量可变的形参，以及通过 getModifiers()方法来获取该方法或构造器的修饰符。除此之外，Executable 提供了如下两个方法来获取该方法或参数的形参个数及形参名。

- int getParameterCount()：获取该构造器或方法的形参个数。
- Parameter[] getParameters()：获取该构造器或方法的所有形参。

上面第二个方法返回了一个 Parameter[]数组，Parameter 也是 Java 8 新增的 API，每个 Parameter 对象代表方法或构造器的一个参数。Parameter 也提供了大量方法来获取声明该参数的泛型信息，还提供了如下常用方法来获取参数信息。

- getModifiers()：获取修饰该形参的修饰符。
- String getName()：获取形参名。
- Type getParameterizedType()：获取带泛型的形参类型。
- Class<?> getType()：获取形参类型。
- boolean isNamePresent()：该方法返回该类的 class 文件中是否包含了方法的形参名信息。
- boolean isVarArgs()：该方法用于判断该参数是否为个数可变的形参。

需要指出的是，使用 javac 命令编译 Java 源文件时，默认生成的 class 文件并不包含方法的形参名信息，因此调用 isNamePresent()方法将会返回 false，调用 getName()方法也不能得到该参数的形参名。如果希望 javac 命令编译 Java 源文件时可以保留形参信息，则需要为该命令指定-parameters 选项。

如下程序示范了 Java 8 的方法参数反射功能。

程序清单：codes\18\18.3\MethodParameterTest.java

```
class Test
{
    public void replace(String str, List<String> list){}
}
public class MethodParameterTest
{
    public static void main(String[] args)throws Exception
    {
        // 获取 String 的类
        Class<Test> clazz = Test.class;
        // 获取 String 类的带两个参数的 replace()方法
        Method replace = clazz.getMethod("replace"
            , String.class, List.class);
```

```
        // 获取指定方法的参数个数
        System.out.println("replace 方法参数个数: " + replace.getParameterCount());
        // 获取 replace 的所有参数信息
        Parameter[] parameters = replace.getParameters();
        int index = 1;
        // 遍历所有参数
        for (Parameter p : parameters)
        {
            if (p.isNamePresent())
            {
                System.out.println("---第" + index++ + "个参数信息---");
                System.out.println("参数名: " + p.getName());
                System.out.println("形参类型: " + p.getType());
                System.out.println("泛型类型: " + p.getParameterizedType());
            }
        }
    }
}
```

上面程序先定义了一个包含简单的Test类，该类中包含一个replace(String str, List<String> list)方法，程序中第一行粗体字代码获取了该方法，接下来程序中三行粗体字代码分别用于获取该方法的形参名、形参类型和泛型信息。

由于上面程序中三行粗体字代码位于p.isNamePresent()条件为true的执行体内，也就是只有当该类的class文件中包含形参名信息时，程序才会执行条件体内的三行粗体字代码。因此需要使用如下命令来编译该程序：

```
javac -parameters -d . MethodParameterTest.java
```

上面命令中-parameters选项用于控制javac命令保留方法形参名信息。

运行该程序，即可看到如下输出：

```
replace 方法参数个数: 2
---第 1 个参数信息---
参数名: str
形参类型: class java.lang.String
泛型类型: class java.lang.String
---第 2 个参数信息---
参数名: list
形参类型: interface java.util.List
泛型类型: java.util.List<java.lang.String>
```

18.4 使用反射生成并操作对象

Class对象可以获得该类里的方法（由Method对象表示）、构造器（由Constructor对象表示）、成员变量（由Field对象表示），这三个类都位于java.lang.reflect包下，并实现了java.lang.reflect.Member接口。程序可以通过Method对象来执行对应的方法，通过Constructor对象来调用对应的构造器创建实例，能通过Field对象直接访问并修改对象的成员变量值。

18.4.1 创建对象

通过反射来生成对象需要先使用Class对象获取指定的Constructor对象，再调用Constructor对象的newInstance()方法来创建该Class对象对应类的实例。通过这种方式可以选择使用指定的构造器来创建实例。

在很多Java EE框架中都需要根据配置文件信息来创建Java对象，从配置文件读取的只是某个类的字符串类名，程序需要根据该字符串来创建对应的实例，就必须使用反射。

下面程序就实现了一个简单的对象池，该对象池会根据配置文件读取key-value对，然后创建这些对象，并将这些对象放入一个HashMap中。

程序清单：codes\18\18.4\ObjectPoolFactory.java

```
public class ObjectPoolFactory
{
```

```
    // 定义一个对象池，前面是对象名，后面是实际对象
    private Map<String ,Object> objectPool = new HashMap<>();
    // 定义一个创建对象的方法
    // 该方法只要传入一个字符串类名，程序可以根据该类名生成 Java 对象
    private Object createObject(String clazzName)
        throws Exception
        , IllegalAccessException , ClassNotFoundException
    {
        // 根据字符串来获取对应的 Class 对象
        Class<?> clazz = Class.forName(clazzName);
        // 使用 clazz 对应类的默认构造器创建实例
        return clazz.getConstructor().newInstance();
    }
    // 该方法根据指定文件来初始化对象池
    // 它会根据配置文件来创建对象
    public void initPool(String fileName)
        throws InstantiationException
        , IllegalAccessException ,ClassNotFoundException
    {
        try(
            FileInputStream fis = new FileInputStream(fileName))
        {
            Properties props = new Properties();
            props.load(fis);
            for (String name : props.stringPropertyNames())
            {
                // 每取出一对 key-value 对，就根据 value 创建一个对象
                // 调用 createObject()创建对象，并将对象添加到对象池中
                objectPool.put(name ,
                    createObject(props.getProperty(name)));
            }
        }
        catch (Exception ex)
        {
            System.out.println("读取" + fileName + "异常");
        }
    }
    public Object getObject(String name)
    {
        // 从 objectPool 中取出指定 name 对应的对象
        return objectPool.get(name);
    }
    public static void main(String[] args)
        throws Exception
    {
        ObjectPoolFactory pf = new ObjectPoolFactory();
        pf.initPool("obj.txt");
        System.out.println(pf.getObject("a"));      // ①
        System.out.println(pf.getObject("b"));      // ②
    }
}
```

上面程序中 createObject()方法里的两行粗体字代码就是根据字符串来创建 Java 对象的关键代码，程序调用 Class 对象的 newInstance()方法即可创建一个 Java 对象。程序中的 initPool()方法会读取属性文件，对属性文件中每个 key-value 对创建一个 Java 对象，其中 value 是该 Java 对象的实现类，而 key 是该 Java 对象放入对象池中的名字。为该程序提供如下属性配置文件。

程序清单：codes\18\18.4\obj.txt

```
a=java.util.Date
b=javax.swing.JFrame
```

编译、运行上面的 ObjectPoolFactory 程序，执行到 main 方法中的①号代码处，将看到输出系统当前时间——这表明对象池中已经有了一个名为 a 的对象，该对象是一个 java.util.Date 对象。执行到②号代码处，将看到输出一个 JFrame 对象。

提示：

这种使用配置文件来配置对象，然后由程序根据配置文件来创建对象的方式非常有用，大名鼎鼎的 Spring 框架就采用这种方式大大简化了 Java EE 应用的开发。当然，Spring 采用的是 XML 配置文件——毕竟属性文件能配置的信息太有限了，而 XML 配置文件能配置的信息就丰富多了。

如果不想利用默认构造器来创建 Java 对象，而想利用指定的构造器来创建 Java 对象，则需要利用 Constructor 对象，每个 Constructor 对应一个构造器。为了利用指定的构造器来创建 Java 对象，需要如下三个步骤。

1. 获取该类的 Class 对象。
2. 利用 Class 对象的 getConstructor()方法来获取指定的构造器。
3. 调用 Constructor 的 newInstance()方法来创建 Java 对象。

下面程序利用反射来创建一个 JFrame 对象，而且使用指定的构造器。

程序清单：codes\18\18.4\CreateJFrame.java

```
public class CreateJFrame
{
    public static void main(String[] args)
        throws Exception
    {
        // 获取 JFrame 对应的 Class 对象
        Class<?> jframeClazz = Class.forName("javax.swing.JFrame");
        // 获取 JFrame 中带一个字符串参数的构造器
        Constructor ctor = jframeClazz
            .getConstructor(String.class);
        // 调用 Constructor 的 newInstance 方法创建对象
        Object obj = ctor.newInstance("测试窗口");
        // 输出 JFrame 对象
        System.out.println(obj);
    }
}
```

上面程序中第一行粗体字代码用于获取 JFrame 类的指定构造器，前面已经提到：如果要唯一地确定某类中的构造器，只要指定构造器的形参列表即可。第一行粗体字代码获取构造器时传入了一个 String 类型，即表明想获取只有一个字符串参数的构造器。

程序中第二行粗体字代码使用指定构造器的 newInstance()方法来创建一个 Java 对象，当调用 Constructor 对象的 newInstance()方法时通常需要传入参数，因为调用 Constructor 的 newInstance()方法实际上等于调用它对应的构造器，传给 newInstance()方法的参数将作为对应构造器的参数。

对于上面的 CreateFrame.java 中已知 java.swing.JFrame 类的情形，通常没有必要使用反射来创建该对象，毕竟通过反射创建对象时性能要稍低一些。实际上，只有当程序需要动态创建某个类的对象时才会考虑使用反射，通常在开发通用性比较广的框架、基础平台时可能会大量使用反射。

18.4.2 调用方法

当获得某个类对应的 Class 对象后，就可以通过该 Class 对象的 getMethods()方法或者 getMethod()方法来获取全部方法或指定方法——这两个方法的返回值是 Method 数组，或者 Method 对象。

每个 Method 对象对应一个方法，获得 Method 对象后，程序就可通过该 Method 来调用它对应的方法。在 Method 里包含一个 invoke()方法，该方法的签名如下。

- Object invoke(Object obj, Object... args)：该方法中的 obj 是执行该方法的主调，后面的 args 是执行该方法时传入该方法的实参。

下面程序对前面的对象池工厂进行加强，允许在配置文件中增加配置对象的成员变量的值，对象池工厂会读取为该对象配置的成员变量值，并利用该对象对应的 setter 方法设置成员变量的值。

程序清单：codes\18\18.4\ExtendedObjectPoolFactory.java

```
public class ExtendedObjectPoolFactory
```

```
{
    // 定义一个对象池，前面是对象名，后面是实际对象
    private Map<String ,Object> objectPool = new HashMap<>();
    private Properties config = new Properties();
    // 从指定属性文件中初始化 Properties 对象
    public void init(String fileName)
    {
        try(
            FileInputStream fis = new FileInputStream(fileName))
        {
            config.load(fis);
        }
        catch (IOException ex)
        {
            System.out.println("读取" + fileName + "异常");
        }
    }
    // 定义一个创建对象的方法
    // 该方法只要传入一个字符串类名，程序可以根据该类名生成 Java 对象
    private Object createObject(String clazzName)
        throws Exception
    {
        // 根据字符串来获取对应的 Class 对象
        Class<?> clazz =Class.forName(clazzName);
        // 使用 clazz 对应类的默认构造器创建实例
        return clazz.getConstructor().newInstance();
    }
    // 该方法根据指定文件来初始化对象池
    // 它会根据配置文件来创建对象
    public void initPool()throws Exception
    {
        for (String name : config.stringPropertyNames())
        {
            // 每取出一个 key-value 对，如果 key 中不包含百分号（%）
            // 这就表明是根据 value 来创建一个对象
            // 调用 createObject 创建对象，并将对象添加到对象池中
            if (!name.contains("%"))
            {
                objectPool.put(name ,
                    createObject(config.getProperty(name)));
            }
        }
    }
    // 该方法将会根据属性文件来调用指定对象的 setter 方法
    public void initProperty()throws InvocationTargetException
        ,IllegalAccessException,NoSuchMethodException
    {
        for (String name : config.stringPropertyNames())
        {
            // 每取出一对 key-value 对，如果 key 中包含百分号（%）
            // 即可认为该 key 用于控制调用对象的 setter 方法设置值
            // %前半为对象名字，后半控制 setter 方法名
            if (name.contains("%"))
            {
                // 将配置文件中的 key 按%分割
                String[] objAndProp = name.split("%");
                // 取出调用 setter 方法的参数值
                Object target = getObject(objAndProp[0]);
                // 获取 setter 方法名:set + "首字母大写" + 剩下部分
                String mtdName = "set" +
                objAndProp[1].substring(0 , 1).toUpperCase()
                    + objAndProp[1].substring(1);
                // 通过 target 的 getClass()获取它的实现类所对应的 Class 对象
                Class<?> targetClass = target.getClass();
                // 获取希望调用的 setter 方法
                Method mtd = targetClass.getMethod(mtdName , String.class);
                // 通过 Method 的 invoke 方法执行 setter 方法
```

```
                // 将 config.getProperty(name)的值作为调用 setter 方法的参数
                mtd.invoke(target , config.getProperty(name));
            }
        }
    }
    public Object getObject(String name)
    {
        // 从 objectPool 中取出指定 name 对应的对象
        return objectPool.get(name);
    }
    public static void main(String[] args)
        throws Exception
    {
        ExtendedObjectPoolFactory epf = new ExtendedObjectPoolFactory();
        epf.init("extObj.txt");
        epf.initPool();
        epf.initProperty();
        System.out.println(epf.getObject("a"));
    }
}
```

上面程序中 initProperty()方法里的第一行粗体字代码获取目标类中包含一个 String 参数的 setter 方法，第二行粗体字代码通过调用 Method 的 invoke()方法来执行该 setter 方法，该方法执行完成后，就相当于执行了目标对象的 setter 方法。为上面程序提供如下配置文件。

程序清单：codes\18\18.4\extObj.text

```
a=javax.swing.JFrame
b=javax.swing.JLabel
#set the title of a
a%title=Test Title
```

上面配置文件中的a%title行表明希望调用a对象的setTitle()方法，调用该方法的参数值为Test Title。编译、运行上面的 ExtendedObjectPoolFactory.java 程序，可以看到输出一个 JFrame 窗口，该窗口的标题为 Test Title。

提示：

Spring 框架就是通过这种方式将成员变量值以及依赖对象等都放在配置文件中进行管理的，从而实现了较好的解耦。这也是 Spring 框架的 IoC 的秘密。

当通过 Method 的 invoke()方法来调用对应的方法时，Java 会要求程序必须有调用该方法的权限。如果程序确实需要调用某个对象的 private 方法，则可以先调用 Method 对象的如下方法。

- setAccessible(boolean flag)：将 Method 对象的 accessible 设置为指定的布尔值。值为 true，指示该 Method 在使用时应该取消 Java 语言的访问权限检查；值为 false，则指示该 Method 在使用时要实施 Java 语言的访问权限检查。

注意：

实际上，setAccessible()方法并不属于 Method，而是属于它的父类 AccessibleObject。因此 Method、Constructor、Field 都可调用该方法，从而实现通过反射来调用 private 方法、private 构造器和 private 成员变量，下一节将会让读者看到这种示例。也就是说，它们可以通过调用该方法来取消访问权限检查，通过反射即可访问 private 成员。

18.4.3　访问成员变量值

通过 Class 对象的 getFields()或 getField()方法可以获取该类所包括的全部成员变量或指定成员变量。Field 提供了如下两组方法来读取或设置成员变量值。

- getXxx(Object obj)：获取 obj 对象的该成员变量的值。此处的 Xxx 对应 8 种基本类型，如果该成员变量的类型是引用类型，则取消 get 后面的 Xxx。
- setXxx(Object obj , Xxx val)：将 obj 对象的该成员变量设置成 val 值。此处的 Xxx 对应 8 种基本

类型，如果该成员变量的类型是引用类型，则取消 set 后面的 Xxx。

使用这两个方法可以随意地访问指定对象的所有成员变量，包括 private 修饰的成员变量。

程序清单：codes\18\18.4\FieldTest.java

```
class Person
{
    private String name;
    private int age;
    public String toString()
    {
        return "Person[name:" + name +
        " , age:" + age + " ]";
    }
}
public class FieldTest
{
    public static void main(String[] args)
        throws Exception
    {
        // 创建一个 Person 对象
        Person p = new Person();
        // 获取 Person 类对应的 Class 对象
        Class<Person> personClazz = Person.class;
        // 获取 Person 的名为 name 的成员变量
        // 使用 getDeclaredField()方法表明可获取各种访问控制符的成员变量
        Field nameField = personClazz.getDeclaredField("name");
        // 设置通过反射访问该成员变量时取消访问权限检查
        nameField.setAccessible(true);
        // 调用 set()方法为 p 对象的 name 成员变量设置值
        nameField.set(p , "Yeeku.H.Lee");
        // 获取 Person 类名为 age 的成员变量
        Field ageField = personClazz.getDeclaredField("age");
        // 设置通过反射访问该成员变量时取消访问权限检查
        ageField.setAccessible(true);
        // 调用 setInt()方法为 p 对象的 age 成员变量设置值
        ageField.setInt(p , 30);
        System.out.println(p);
    }
}
```

上面程序中先定义了一个 Person 类，该类里包含两个 private 成员变量：name 和 age，在通常情况下，这两个成员变量只能在 Person 类里访问。但本程序 FieldTest 的 main()方法中 6 行粗体字代码通过反射修改了 Person 对象的 name、age 两个成员变量的值。

第一行粗体字代码使用 getDeclaredField()方法获取了名为 name 的成员变量，注意此处不是使用 getField()方法，因为 getField()方法只能获取 public 访问控制的成员变量，而 getDeclaredField()方法则可以获取所有的成员变量；第二行粗体字代码则通过反射访问该成员变量时不受访问权限的控制；第三行粗体字代码修改了 Person 对象的 name 成员变量的值。修改 Person 对象的 age 成员变量的值的方式与此完全相同。

编译、运行上面程序，会看到如下输出：

```
Person [ name:Yeeku.H.Lee , age:30 ]
```

18.4.4 操作数组

在 java.lang.reflect 包下还提供了一个 Array 类，Array 对象可以代表所有的数组。程序可以通过使用 Array 来动态地创建数组，操作数组元素等。

Array 提供了如下几类方法。

- static Object newInstance(Class<?> componentType, int... length)：创建一个具有指定的元素类型、指定维度的新数组。
- static xxx getXxx(Object array, int index)：返回 array 数组中第 index 个元素。其中 xxx 是各种基

本数据类型，如果数组元素是引用类型，则该方法变为 get(Object array, int index)。

- static void setXxx(Object array, int index, xxx val)：将 array 数组中第 index 个元素的值设为 val。其中 xxx 是各种基本数据类型，如果数组元素是引用类型，则该方法变成 set(Object array, int index, Object val)。

下面程序示范了如何使用 Array 来生成数组，为指定数组元素赋值，并获取指定数组元素的方式。

程序清单：codes\18\18.4\ArrayTest1.java

```
public class ArrayTest1
{
    public static void main(String args[])
    {
        try
        {
            // 创建一个元素类型为 String，长度为 10 的数组
            Object arr = Array.newInstance(String.class, 10);
            // 依次为 arr 数组中 index 为 5、6 的元素赋值
            Array.set(arr, 5, "疯狂 Java 讲义");
            Array.set(arr, 6, "轻量级 Java EE 企业应用实战");
            // 依次取出 arr 数组中 index 为 5、6 的元素的值
            Object book1 = Array.get(arr , 5);
            Object book2 = Array.get(arr , 6);
            // 输出 arr 数组中 index 为 5、6 的元素
            System.out.println(book1);
            System.out.println(book2);
        }
        catch (Throwable e)
        {
            System.err.println(e);
        }
    }
}
```

上面程序中三行粗体字代码分别是通过 Array 创建数组，为数组元素设置值，访问数组元素的值的示例代码，程序通过使用 Array 就可以动态地创建并操作数组。

下面程序比上面程序稍微复杂一点，下面程序使用 Array 类创建了一个三维数组。

程序清单：codes\18\18.4\ArrayTest2.java

```
public class ArrayTest2
{
    public static void main(String args[])
    {
        /*
          创建一个三维数组
          根据前面介绍数组时讲的：三维数组也是一维数组
          是数组元素是二维数组的一维数组
          因此可以认为 arr 是长度为 3 的一维数组
        */
        Object arr = Array.newInstance(String.class, 3, 4, 10);
        // 获取 arr 数组中 index 为 2 的元素，该元素应该是二维数组
        Object arrObj = Array.get(arr, 2);
        // 使用 Array 为二维数组的数组元素赋值，二维数组的数组元素是一维数组
        // 所以传入 Array 的 set() 方法的第三个参数是一维数组
        Array.set(arrObj , 2 , new String[]
        {
            "疯狂 Java 讲义",
            "轻量级 Java EE 企业应用实战"
        });
        // 获取 arrObj 数组中 index 为 3 的元素，该元素应该是一维数组
        Object anArr  = Array.get(arrObj, 3);
        Array.set(anArr , 8  , "疯狂 Android 讲义");
        // 将 arr 强制类型转换为三维数组
        String[][][] cast = (String[][][])arr;
```

```
        // 获取 cast 三维数组中指定元素的值
        System.out.println(cast[2][3][8]);
        System.out.println(cast[2][2][0]);
        System.out.println(cast[2][2][1]);
    }
}
```

上面程序的第一行粗体字代码使用 Array 创建了一个三维数组，程序中较难理解的地方是第二段粗体字代码部分，使用 Array 为 arrObj 的指定元素赋值，相当于为二维数组的元素赋值。由于二维数组的元素是一维数组，所以程序传入的参数是一个一维数组对象。

运行上面程序，将看到 cast[2][3][8]、cast[2][2][0]、cast[2][2][1]元素都有值，这些值就是刚才程序通过反射传入的数组元素值。

18.5 使用反射生成 JDK 动态代理

在 Java 的 java.lang.reflect 包下提供了一个 Proxy 类和一个 InvocationHandler 接口，通过使用这个类和接口可以生成 JDK 动态代理类或动态代理对象。

18.5.1 使用 Proxy 和 InvocationHandler 创建动态代理

Proxy 提供了用于创建动态代理类和代理对象的静态方法，它也是所有动态代理类的父类。如果在程序中为一个或多个接口动态地生成实现类，就可以使用 Proxy 来创建动态代理类；如果需要为一个或多个接口动态地创建实例，也可以使用 Proxy 来创建动态代理实例。

Proxy 提供了如下两个方法来创建动态代理类和动态代理实例。

- static Class<?> getProxyClass(ClassLoader loader, Class<?>... interfaces)：创建一个动态代理类所对应的 Class 对象，该代理类将实现 interfaces 所指定的多个接口。第一个 ClassLoader 参数指定生成动态代理类的类加载器。
- static Object newProxyInstance(ClassLoader loader,Class<?>[] interfaces, InvocationHandler h)：直接创建一个动态代理对象，该代理对象的实现类实现了 interfaces 指定的系列接口，执行代理对象的每个方法时都会被替换执行 InvocationHandler 对象的 invoke 方法。

实际上，即使采用第一个方法生成动态代理类之后，如果程序需要通过该代理类来创建对象，依然需要传入一个 InvocationHandler 对象。也就是说，系统生成的每个代理对象都有一个与之关联的 InvocationHandler 对象。

提示：

计算机是很“蠢”的，当程序使用反射方式为指定接口生成系列动态代理对象时，这些动态代理对象的实现类实现了一个或多个接口。动态代理对象就需要实现一个或多个接口里定义的所有方法，但问题是：系统怎么知道如何实现这些方法？这个时候就轮到 InvocationHandler 对象登场了——当执行动态代理对象里的方法时，实际上会替换成调用 InvocationHandler 对象的 invoke 方法。

程序中可以采用先生成一个动态代理类，然后通过动态代理类来创建代理对象的方式生成一个动态代理对象。代码片段如下：

```
// 创建一个 InvocationHandler 对象
InvocationHandler handler = new MyInvocationHandler(...);
// 使用 Proxy 生成一个动态代理类 proxyClass
Class proxyClass = Proxy.getProxyClass(Foo.class.getClassLoader()
    , new Class[] { Foo.class });
// 获取 proxyClass 类中带一个 InvocationHandler 参数的构造器
Constructor ctor = proxyClass. getConstructor(new Class[]
    { InvocationHandler.class });
// 调用 ctor 的 newInstance 方法来创建动态实例
Foo f = (Foo)ctor.newInstance(new Object[]{handler});
```

上面代码也可以简化成如下代码：

```
// 创建一个 InvocationHandler 对象
InvocationHandler handler = new MyInvocationHandler(...);
// 使用 Proxy 直接生成一个动态代理对象
Foo f = (Foo)Proxy.newProxyInstance(Foo.class.getClassLoader()
    , new Class[]{Foo.class} , handler);
```

下面程序示范了使用 Proxy 和 InvocationHandler 来生成动态代理对象。

程序清单：codes\18\18.5\ProxyTest.java

```
interface Person
{
    void walk();
    void sayHello(String name);
}
class MyInvokationHandler implements InvocationHandler
{
    /*
    执行动态代理对象的所有方法时，都会被替换成执行如下的 invoke 方法
    其中：
    proxy：代表动态代理对象
    method：代表正在执行的方法
    args：代表调用目标方法时传入的实参
    */
    public Object invoke(Object proxy, Method method, Object[] args)
    {
        System.out.println("----正在执行的方法:" + method);
        if (args != null)
        {
            System.out.println("下面是执行该方法时传入的实参为：");
            for (Object val : args)
            {
                System.out.println(val);
            }
        }
        else
        {
            System.out.println("调用该方法没有实参！");
        }
        return null;
    }
}
public class ProxyTest
{
    public static void main(String[] args)
        throws Exception
    {
        // 创建一个 InvocationHandler 对象
        InvocationHandler handler = new MyInvokationHandler();
        // 使用指定的 InvocationHandler 来生成一个动态代理对象
        Person p = (Person)Proxy.newProxyInstance(Person.class.getClassLoader()
            , new Class[]{Person.class}, handler);
        // 调用动态代理对象的 walk()和 sayHello()方法
        p.walk();
        p.sayHello("孙悟空");
    }
}
```

上面程序首先提供了一个 Person 接口，该接口中包含了 walk()和 sayHello()两个抽象方法，接着定义了一个简单的 InvocationHandler 实现类，定义该实现类时需要重写 invoke()方法——调用代理对象的所有方法时都会被替换成调用该 invoke()方法。该 invoke()方法中的三个参数解释如下。

- proxy：代表动态代理对象。
- method：代表正在执行的方法。
- args：代表调用目标方法时传入的实参。

上面程序中第一行粗体字代码创建了一个 InvocationHandler 对象，第二行粗体字代码根据 InvocationHandler 对象创建了一个动态代理对象。运行上面程序，会看到如图 18.3 所示的运行效果。

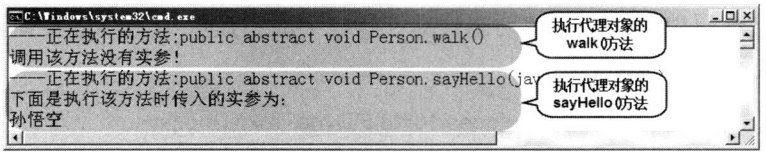


图 18.3　调用动态代理对象的方法效果

从图 18.3 可以看出，不管程序是执行代理对象的 walk()方法，还是执行代理对象的 sayHello()方法，实际上都是执行 InvocationHandler 对象的 invoke()方法。

看完了上面的示例程序，可能有读者会觉得这个程序没有太大的实用价值，难以理解 Java 动态代理的魅力。实际上，在普通编程过程中，确实无须使用动态代理，但在编写框架或底层基础代码时，动态代理的作用就非常大。

18.5.2　动态代理和 AOP

根据前面介绍的 Proxy 和 InvocationHandler，实在很难看出这种动态代理的优势。下面介绍一种更实用的动态代理机制。

开发实际应用的软件系统时，通常会存在相同代码段重复出现的情况，在这种情况下，对于许多刚开始从事软件开发的人而言，他们的做法是：选中那些代码，一路“复制“、“粘贴”，立即实现了系统功能，如果仅仅从软件功能上来看，他们确实已经完成了软件开发。

通过这种“复制”、“粘贴”方式开发出来的软件如图 18.4 所示。

采用图 18.4 所示结构实现的软件系统，在软件开发期间可能会觉得无所谓，但如果有一天需要修改程序的深色代码的实现，则意味着打开三份源代码进行修改。如果有 100 个地方甚至 1000 个地方使用了这段深色代码段，那么修改、维护这段代码的工作量将变成噩梦。

在这种情况下，大部分稍有经验的开发者都会将这段深色代码段定义成一个方法，然后让另外三段代码段直接调用该方法即可。在这种方式下，软件系统的结构如图 18.5 所示。

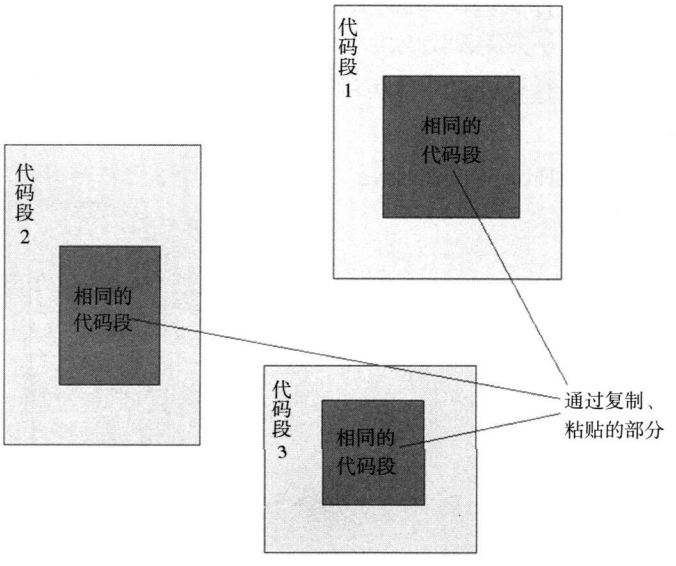


图 18.4　多个地方包含相同代码的软件

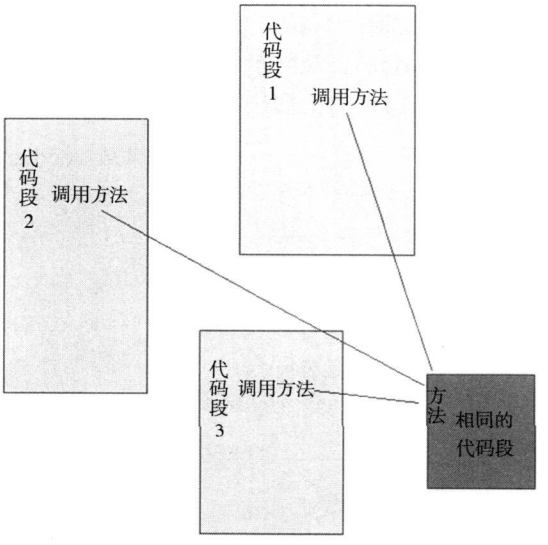


图 18.5　通过方法调用实现代码复用

对于如图 18.5 所示的软件系统，如果需要修改深色部分的代码，则只要修改一个地方即可，而调用该方法的代码段，不管有多少个地方调用了该方法，都完全无须任何修改，只要被调用方法被修改了，所有调用该方法的地方就会自然改变——通过这种方式，大大降低了软件后期维护的复杂度。

但采用这种方式来实现代码复用依然产生一个重要问题：代码段 1、代码段 2、代码段 3 和深色代码段分离开了，但代码段 1、代码段 2 和代码段 3 又和一个特定方法耦合了！最理想的效果是：代码块 1、代码块 2 和代码块 3 既可以执行深色代码部分，又无须在程序中以硬编码方式直接调用深色代码的方法，这时就可以通过动态代理来达到这种效果。

由于JDK动态代理只能为接口创建动态代理，所以下面先提供一个Dog接口，该接口代码非常简单，仅仅在该接口里定义了两个方法。

程序清单：codes\18\18.5\DynaProxy\Dog.java

```
public interface Dog
{
    // info()方法声明
    void info();
    // run()方法声明
    void run();
}
```

上面接口里只是简单地定义了两个方法，并未提供方法实现。如果直接使用Proxy为该接口创建动态代理对象，则动态代理对象的所有方法的执行效果又将完全一样。实际情况通常是，软件系统会为该Dog接口提供一个或多个实现类。此处先提供一个简单的实现类：GunDog。

程序清单：codes\18\18.5\DynaProxy\GunDog.java

```
public class GunDog implements Dog
{
    // 实现info()方法，仅仅打印一个字符串
    public void info()
    {
        System.out.println("我是一只猎狗");
    }
    // 实现run()方法，仅仅打印一个字符串
    public void run()
    {
        System.out.println("我奔跑迅速");
    }
}
```

上面代码没有丝毫的特别之处，该Dog的实现类仅仅为每个方法提供了一个简单实现。再看需要实现的功能：让代码段1、代码段2和代码段3既可以执行深色代码部分，又无须在程序中以硬编码方式直接调用深色代码的方法。此处假设info()、run()两个方法代表代码段1、代码段2，那么要求：程序执行info()、run()方法时能调用某个通用方法，但又不想以硬编码方式调用该方法。下面提供一个DogUtil类，该类里包含两个通用方法。

程序清单：codes\18\18.5\DynaProxy\DogUtil.java

```
public class DogUtil
{
    // 第一个拦截器方法
    public void method1()
    {
        System.out.println("=====模拟第一个通用方法=====");
    }
    // 第二个拦截器方法
    public void method2()
    {
        System.out.println("=====模拟通用方法二=====");
    }
}
```

借助于Proxy和InvocationHandler就可以实现——当程序调用info()方法和run()方法时，系统可以“自动”将method1()和method2()两个通用方法插入info()和run()方法中执行。

这个程序的关键在于下面的MyInvokationHandler类，该类是一个InvocationHandler实现类，该实现类的invoke()方法将会作为代理对象的方法实现。

程序清单：codes\18\18.5\DynaProxy\MyInvokationHandler.java

```
public class MyInvokationHandler implements InvocationHandler
{
    // 需要被代理的对象
    private Object target;
    public void setTarget(Object target)
```

```
    {
        this.target = target;
    }
    // 执行动态代理对象的所有方法时，都会被替换成执行如下的 invoke 方法
    public Object invoke(Object proxy, Method method, Object[] args)
        throws Exception
    {
        DogUtil du = new DogUtil();
        // 执行 DogUtil 对象中的 method1 方法
        du.method1();
        // 以 target 作为主调来执行 method 方法
        Object result = method.invoke(target , args);
        // 执行 DogUtil 对象中的 method2 方法
        du.method2();
        return result;
    }
}
```

上面程序实现 invoke()方法时包含了一行关键代码（以粗体字标出），这行代码通过反射以 target 作为主调来执行 method 方法，这就是回调了 target 对象的原有方法。在粗体字代码之前调用 DogUtil 对象的 method1()方法，在粗体字代码之后调用 DogUtil 对象的 method2()方法。

下面再为程序提供一个 MyProxyFactory 类，该对象专为指定的 target 生成动态代理实例。

程序清单：codes\18\18.5\DynaProxy\MyProxyFactory.java

```
public class MyProxyFactory
{
    // 为指定的 target 生成动态代理对象
    public static Object getProxy(Object target)
        throws Exception
        {
        // 创建一个 MyInvokationHandler 对象
        MyInvokationHandler handler =
        new MyInvokationHandler();
        // 为 MyInvokationHandler 设置 target 对象
        handler.setTarget(target);
        // 创建并返回一个动态代理
        return Proxy.newProxyInstance(target.getClass().getClassLoader()
            , target.getClass().getInterfaces() , handler);
    }
}
```

上面的动态代理工厂类提供了一个 getProxy()方法，该方法为 target 对象生成一个动态代理对象，这个动态代理对象与 target 实现了相同的接口，所以具有相同的 public 方法——从这个意义上来看，动态代理对象可以当成 target 对象使用。当程序调用动态代理对象的指定方法时，实际上将变为执行 MyInvokationHandler 对象的 invoke()方法。例如，调用动态代理对象的 info()方法，程序将开始执行 invoke()方法，其执行步骤如下。

1. 创建 DogUtil 实例。
2. 执行 DogUtil 实例的 method1()方法。
3. 使用反射以 target 作为调用者执行 info()方法。
4. 执行 DogUtil 实例的 method2()方法。

看到上面的执行过程，读者应该已经发现：当使用动态代理对象来代替 target 对象时，代理对象的方法就实现了前面的要求——程序执行 info()、run()方法时既能“插入”method1()、method2()通用方法，但 GunDog 的方法中又没有以硬编码方式调用 method1()和 method2()方法。

下面提供一个主程序来测试这种动态代理的效果。

程序清单：codes\18\18.5\DynaProxy\Test.java

```
public class Test
{
    public static void main(String[] args)
        throws Exception
    {
        // 创建一个原始的 GunDog 对象，作为 target
```

```
        Dog target = new GunDog();
        // 以指定的 target 来创建动态代理对象
        Dog dog = (Dog)MyProxyFactory.getProxy(target);
        dog.info();
        dog.run();
    }
}
```

上面程序中的 dog 对象实际上是动态代理对象，只是该动态代理对象也实现了 Dog 接口，所以也可以当成 Dog 对象使用。程序执行 dog 的 info()和 run()方法时，实际上会先执行 DogUtil 的 method1()方法，再执行 target 对象的 info()和 run()方法，最后执行 DogUtil 的 method2()方法。运行上面程序，会看到如图 18.6 所示的运行结果。

通过图 18.6 所示的运行结果来看，不难发现采用动态代理可以非常灵活地实现解耦。通常而言，使用 Proxy 生成一个动态代理时，往往并不会凭空产生一个动态代理，这样没有太大的实际意义。通常都是为指定的目标对象生成动态代理。

这种动态代理在 AOP（Aspect Orient Programming，面向切面编程）中被称为 AOP 代理，AOP 代理可代替目标对象，AOP 代理包含了目标对象的全部方法。但 AOP 代理中的方法与目标对象的方法存在差异：AOP 代理里的方法可以在执行目标方法之前、之后插入一些通用处理。

AOP 代理包含的方法与目标对象包含的方法示意图如图 18.7 所示。

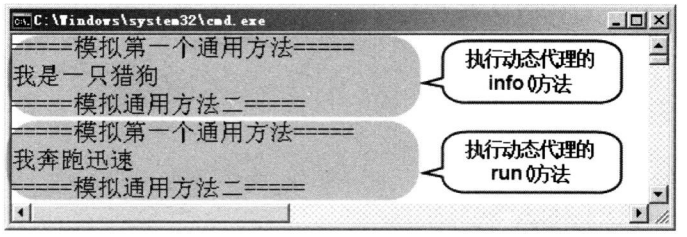


图 18.6　动态代理效果

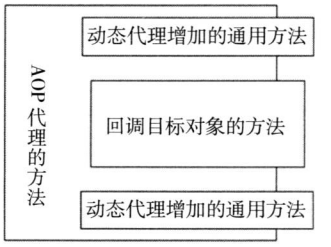


图 18.7　AOP 代理的方法与目标对象的方法示意图

提示：

关于 AOP 更详细的介绍，读者可以在有一定的 Java 编程经验后参考疯狂 Java 体系的《轻量级 Java EE 企业应用实战》，书中有关于 AOP 编程更详细、更深入的内容。

18.6　反射和泛型

从 JDK 5 以后，Java 的 Class 类增加了泛型功能，从而允许使用泛型来限制 Class 类，例如，String.class 的类型实际上是 Class<String>。如果 Class 对应的类暂时未知，则使用 Class<?>。通过在反射中使用泛型，可以避免使用反射生成的对象需要强制类型转换。

18.6.1　泛型和 Class 类

使用 Class<T>泛型可以避免强制类型转换。例如，下面提供一个简单的对象工厂，该对象工厂可以根据指定类来提供该类的实例。

程序清单：codes\18\18.6\CrazyitObjectFactory.java

```
public class CrazyitObjectFactory
{
    public static Object getInstance(String clsName)
    {
        try
        {
            // 创建指定类对应的 Class 对象
            Class cls = Class.forName(clsName);
            // 返回使用该 Class 对象创建的实例
            return cls.newInstance();
        }
        catch(Exception e)
```

```
        {
            e.printStackTrace();
            return null;
        }
    }
}
```

上面程序中两行粗体字代码根据指定的字符串类型创建了一个新对象，但这个对象的类型是 Object，因此当需要使用 CrazyitObjectFactory 的 getInstance()方法来创建对象时，将会看到如下代码：

```
// 获取实例后需要强制类型转换
Date d = (Date)Crazyit.getInstance("java.util.Date");
```

甚至出现如下代码：

```
JFrame f = (JFrame)Crazyit.getInstance("java.util.Date");
```

上面代码在编译时不会有任何问题，但运行时将抛出 ClassCastException 异常，因为程序试图将一个 Date 对象转换成 JFrame 对象。

如果将上面的 CrazyitObjectFactory 工厂类改写成使用泛型后的 Class，就可以避免这种情况。

程序清单：codes\18\18.6\CrazyitObjectFactory2.java

```
public class CrazyitObjectFactory2
{
    public static <T> T getInstance(Class<T> cls)
    {
        try
        {
            return cls.newInstance();
        }
        catch(Exception e)
        {
            e.printStackTrace();
            return null;
        }
    }
    public static void main(String[] args)
    {
        // 获取实例后无须类型转换
        Date d = CrazyitObjectFactory2.getInstance(Date.class);
        JFrame f = CrazyitObjectFactory2.getInstance(JFrame.class);
    }
}
```

在上面程序的 getInstance()方法中传入一个 Class<T>参数，这是一个泛型化的 Class 对象，调用该 Class 对象的 newInstance()方法将返回一个 T 对象，如程序中粗体字代码所示。接下来当使用 CrazyitObjectFactory2 工厂类的 getInstance()方法来产生对象时，无须使用强制类型转换，系统会执行更严格的检查，不会出现 ClassCastException 运行时异常。

前面介绍使用 Array 类来创建数组时，曾经看到如下代码：

```
// 使用 Array 的 newInstance 方法来创建一个数组
Object arr = Array.newInstance(String.class, 10);
```

对于上面的代码其实使用并不是非常方便，因为 newInstance()方法返回的确实是一个 String[]数组，而不是简单的 Object 对象。如果需要将 arr 对象当成 String[]数组使用，则必须使用强制类型转换——这是不安全的操作。

提示：

奇怪的是，Array 的 newInstance()方法签名为如下形式：

```
public static Object newInstance(Class<?> componentType, int... dimensions)
```

在这个方法签名中使用了 Class<?>泛型，但并没有真正利用这个泛型；如果将该方法签名改为如下形式：

```
public static <T> T[] newInstance(Class<T> componentType, int length)
```

这样就可以在调用该方法后无须强制类型转换了。不过，这个方法暂时只能创建一维数组，也就不能利用可变个数的参数优势了。

为了示范泛型的优势，可以对 Array 的 newInstance()方法进行包装。

程序清单：codes\18\18.6\CrazyitArray.java

```
public class CrazyitArray
{
    // 对 Array 的 newInstance 方法进行包装
    @SuppressWarnings("unchecked")
    public static <T> T[] newInstance(Class<T> componentType, int length)
    {
        return (T[])Array.newInstance(componentType , length); // ①
    }
    public static void main(String[] args)
    {
        // 使用 CrazyitArray 的 newInstance()创建一维数组
        String[] arr = CrazyitArray.newInstance(String.class , 10);
        // 使用 CrazyitArray 的 newInstance()创建二维数组
        // 在这种情况下，只要设置数组元素的类型是 int[]即可
        int[][] intArr = CrazyitArray.newInstance(int[].class , 5);
        arr[5] = "疯狂 Java 讲义";
        // intArr 是二维数组，初始化该数组的第二个数组元素
        // 二维数组的元素必须是一维数组
        intArr[1] = new int[]{23, 12};
        System.out.println(arr[5]);
        System.out.println(intArr[1][1]);
    }
}
```

上面程序中粗体字代码定义的 newInstance()方法对 Array 类提供的 newInstance()方法进行了包装，将方法签名改成了 public static <T> T[] newInstance(Class<T> componentType, int length)，这就保证程序通过该 newInstance()方法创建数组时的返回值就是数组对象，而不是 Object 对象，从而避免了强制类型转换。

提示：

程序在①行代码处将会有一个 unchecked 编译警告，所以程序使用了@SuppressWarnings 来抑制这个警告信息。

18.6.2 使用反射来获取泛型信息

通过指定类对应的 Class 对象，可以获得该类里包含的所有成员变量，不管该成员变量是使用 private 修饰，还是使用 public 修饰。获得了成员变量对应的 Field 对象后，就可以很容易地获得该成员变量的数据类型，即使用如下代码即可获得指定成员变量的类型。

```
// 获取成员变量 f 的类型
Class<?> a = f.getType();
```

但这种方式只对普通类型的成员变量有效。如果该成员变量的类型是有泛型类型的类型，如 Map<String , Integer>类型，则不能准确地得到该成员变量的泛型参数。

为了获得指定成员变量的泛型类型，应先使用如下方法来获取该成员变量的泛型类型。

```
// 获得成员变量 f 的泛型类型
Type gType = f.getGenericType();
```

然后将 Type 对象强制类型转换为 ParameterizedType 对象，ParameterizedType 代表被参数化的类型，也就是增加了泛型限制的类型。ParameterizedType 类提供了如下两个方法。

- getRawType()：返回没有泛型信息的原始类型。
- getActualTypeArguments()：返回泛型参数的类型。

下面是一个获取泛型类型的完整程序。

程序清单：codes\18\18.6\GenericTest.java

```
public class GenericTest
{
    private Map<String , Integer> score;
    public static void main(String[] args)
        throws Exception
    {
        Class<GenericTest> clazz = GenericTest.class;
        Field f = clazz.getDeclaredField("score");
        // 直接使用 getType()取出类型只对普通类型的成员变量有效
        Class<?> a = f.getType();
        // 下面将看到仅输出 java.util.Map
        System.out.println("score 的类型是:" + a);
        // 获得成员变量 f 的泛型类型
        Type gType = f.getGenericType();
        // 如果 gType 类型是 ParameterizedType 对象
        if(gType instanceof ParameterizedType)
        {
            // 强制类型转换
            ParameterizedType pType = (ParameterizedType)gType;
            // 获取原始类型
            Type rType = pType.getRawType();
            System.out.println("原始类型是: " + rType);
            // 取得泛型类型的泛型参数
            Type[] tArgs = pType.getActualTypeArguments();
            System.out.println("泛型信息是:");
            for (int i = 0; i < tArgs.length; i++)
            {
                System.out.println("第" + i + "个泛型类型是: " + tArgs[i]);
            }
        }
        else
        {
            System.out.println("获取泛型类型出错！");
        }
    }
}
```

上面程序中的粗体字代码就是取得泛型类型的关键代码。运行上面程序，将看到如下运行结果：

```
score 的类型是:interface java.util.Map
原始类型是: interface java.util.Map
泛型信息是:
第 0 个泛型类型是: class java.lang.String
第 1 个泛型类型是: class java.lang.Integer
```

从上面的运行结果可以看出，使用 getType()方法只能获取普通类型的成员变量的数据类型；对于增加了泛型的成员变量，应该使用 getGenericType()方法来取得其类型。

提示：

Type 也是 java.lang.reflect 包下的一个接口，该接口代表所有类型的公共高级接口，Class 是 Type 接口的实现类。Type 包括原始类型、参数化类型、数组类型、类型变量和基本类型等。

18.7 本章小结

本章详细介绍了 Java 反射的相关知识。本章内容对于普通的 Java 学习者而言，确实显得有点深入，并且会感觉不太实用。但随着知识的慢慢积累，当读者希望开发出更多基础的、适应性更广的、灵活性更强的代码时，就会想到使用反射知识了。本章从类的加载、初始化开始，深入介绍了 Java 类加载器的原理和机制。本章重点在于介绍 Class、Method、Field、Constructor、Type、ParameterizedType 等类和接口的用法，包括动态创建 Java 实例和动态调用 Java 对象的方法。本章介绍的两个对象工厂实际上

就是 Spring 框架的核心，希望读者用心揣摩。

本章也介绍了利用 Proxy 和 InvocationHandler 来创建 JDK 动态代理，并详细介绍了 JDK 动态代理和 AOP 之间的关系，这也是 Java 灵活性的重要方面，对于提高系统解耦也十分重要，希望读者能用心掌握。

➤➤ 本章练习

1．开发一个工具类，该工具类提供一个 eval()方法，实现 JavaScript 中 eval()函数的功能——可以动态运行一行或多行程序代码。例如 eval("System.out.println(\"aa\")")，将输出 aa。

2．开发一个对象工厂池，这个对象工厂池不仅可以管理对象的 String 类型成员变量的值，还可以管理容器中对象的其他类型成员变量的值，甚至可以将对象的成员变量设置成引用到容器中其他对象（这就是 Spring 所提出的控制反转，即 IoC）。

附录 A　Java 9 的模块化系统

面世 20 多年的 Java，已经发展成为一门影响深远的编程语言，无数平台、系统都采用 Java 语言编写。但 Java 也越来越庞大，逐渐发展成一头“臃肿”的大象：无论是运行一个大型的系统平台，还是运行一个小小的工具软件，JVM 总要加载整个 Java 运行时环境，其中位于 JDK 安装目录下 jre\lib 下的 rt.jar 就超过 60MB，而位于 JDK 安装目录下 lib 目录下的 tools.jar 也达到 17.3MB。即使程序只需要使用 Java 的部分核心功能，JVM 也需要完整地加载数百 MB 的 JRE 环境。

为了给 Java“瘦身”，让 Java 实现轻量化，Java 9 正式推出了模块化系统（项目代号为 Jigsaw），Java 正式被拆分成 *N* 个模块，并允许 Java 程序可以根据需要选择只加载程序必需的 Java 模块，这样就可以让 Java 以轻量化的方式来运行。

提示：

Java 7 已经提出了模块化的概念，但由于其过于复杂，Java 7、Java 8 一直未能真正推出。Java 模块化直到 Java 9 才真正成熟。

对于 Java 而言，模块化系统是一次真正的自我革新，这种革新使“古老而庞大”的 Java 语言重新焕发年轻的活力。

A.1　理解模块化系统

在 Java 9 之前，一个 Java 程序通常会以 *N* 个包的形式进行组织，每个包下可包含 *N* 个 Java 类型（类、接口、枚举和注解），这种程序组织结构本身就存在以下问题。

- 包只是充当命名空间的角色，包中的公共类型可以在所有其他包中访问；包并没有真正充当访问权限的界定边界。
- Java 程序运行时只能看到程序加载系列 JAR 包，无法真正确定不同 JAR 包中是否包含多个相同类型的不同副本，而 Java 程序运行默认加载类路径中遇到的第一个 JAR 包所包含的 Java 类型。
- Java 程序运行时经常由于缺失某个 JAR 包而导致 ClassNotFoundException 异常。有时候也会因为包含错误的 JAR 版本而导致运行时错误。

另外，庞大而臃肿的 JRE 库也是一个问题，无论所运行的 Java 软件多么小，系统总需要下载、启动整个 JRE，这样既增加了系统开销，又降低了程序运行性能。

Java 9 的模块化系统致力于解决以上问题，模块化系统从两方面进行规范。

- 模块化系统将整个 JDK、JRE 本身分解成多个相对独立的模块，这样应用程序可根据需要只加载必需的模块。
- 应用程序、框架、库本身可以被分解成相对独立的模块，模块与模块相对独立，而且模块可作为访问权限的界定边界。

每个模块都有如图 A.1 所示的结构。

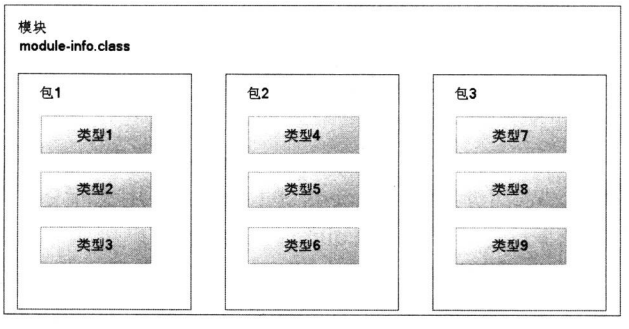


图 A.1　模块的结构

从图 A.1 可以看出，模块是一个比“包”更大的程序单元，一个模块可以包含多个包，

而每个包下又可包含 *N* 个 Java 类型（类、接口、枚举和注解）。

此外，模块也可作为访问权限的界定边界，模块可通过模块描述文件指定哪些包需要被暴露出来、哪些包需要被隐藏。对于被隐藏的包，即使它所包含的 Java 类型使用了 public 修饰，这些 Java 类型也依然不能被其他模块访问。

A.2 创建模块

Java 9 为 javac 命令添加了几个选项，允许它一次编译一个模块或多个模块。如果需要一次编译多个模块，则必须将每个模块的源代码保存在与模块名称相同的目录下。实际上，即使只有一个模块，最好也遵循此命名约定。

下面将会按此约定来示范开发一个模块。首先在当前工作目录下新建 3 个子目录。

- -src：用于保存源代码。
- -mods：用于保存所生成的字节码文件。
- -lib：用于保存模块生成的 JAR 包。

此处打算新建一个名为 org.cimodule 的模块，因此在 src 目录下新建一个名为 org.cimodule 的文件夹。

提示：

模块名看上去与包名有点相似，模块名同样应该每个字母都小写。模块并不保存在对应的文件结构下，只要保存在与模块同名的目录下即可。比如 org.cimodule 模块，只要将源文件保存在名为 org.cimodule 的目录下即可，不需要保存在 org\cimodule 目录下。

接下来在 src\org.cimodule 目录下新建一个模块描述文件——一个名为 module-info.java 的 Java 文件（编译后会生成 module-info.class 文件），该文件专门定义模块名、访问权限、模块依赖等相关信息。

提示：

这个模块描述文件其实就相当于一个 XML 配置文件，但 Java 9 没有打算采用其他文件格式，而是直接使用 Java 文件本身作为模块描述文件。

module-info.java 与前面介绍的 Java 源代码完全不同，它只是用于定义模块，不再定义任何 Java 类、接口、枚举或注解。module-info.java 文件的完整语法格式如下：

```
[open] module <modulename>
{
    <module-statement>;
    <module-statement>;
    ...
}
```

关于上面语法格式的解释如下：

- module 是一个关键字，表明此处正在定义模块。
- open 修饰符是可选的，它声明一个开放模块。开放模块导出所有的包，以便其他模块可通过反射来访问该模块中的类型。
- <modulename>用于指定模块的名称。
- <module-statement>是模块语句。在模块声明中可以包含 0~*N* 条模块语句。Java 9 共支持以下 5 种模块语句。
  - 导出语句（exports statement）：用于指定暴露该模块中哪些包。
  - 开放语句（opens statement）：用于指定开放该模块中哪些包。
  - 需要语句（requires statement）：用于声明该模块需要依赖的其他模块。
  - 使用语句（uses statement）：用于声明该模块可供使用的服务接口。
  - 提供语句（provides statement）：用于声明为服务接口提供实现。

注意：

导出（exports）和开放（opens）看上去似乎差不多，但实际上有区别：模块中被导出的包所包含的类型可以被其他模块自由访问（如果访问权限允许）；但模块中被开放的包所包含的类型只能被其他模块通过反射访问。一旦模块声明使用 open 修饰符，就意味着该模块内所有的包都是开放的，因此不再需要在模块内定义开放语句。

下面定义本模块的 module-info.java 文件。由于这是我们的第一个模块，因此该模块描述文件非常简单：只是定义模块名。

程序清单：codes\a01\moduleExample\src\org.cimodule\module-info.java

```
module org.cimodule
{
}
```

接下来程序为该模块定义两个 Java 源程序，并将这两个 Java 源程序单独放在不同的包下。第一个 User 类位于 org.crazyit.domain 包下，因此将该源程序保存在 org.cimodule\org\crazyit\domain 目录下，其中 org.cimodule 是模块名，org\crazyit\domain 是包名对应的文件结构。

程序清单：codes\a01\moduleExample\src\org.cimodule\org\crazyit\domain\User.java

```
public class User
{
    public String sayHi(String name)
    {
        System.out.println("--执行 User 的 sayHi 方法--");
        return name + "您好！" + new java.util.Date();
    }
}
```

再定义一个 Hello 类，该类位于 org.crazyit.main 包下，因此将该源程序保存在 org.cimodule\org\crazyit\main 目录下。

程序清单：codes\a01\moduleExample\src\org.cimodule\org\crazyit\main\Hello.java

```
public class Hello
{
    public void info()
    {
        System.out.println("Hello 的 info 方法");
    }
    public static void main(String[] args)
    {
        // 获取 Hello 类
        Class<Hello> cls = Hello.class;
        // 获取 Hello 类所在的模块
        Module mod = cls.getModule();
        // 输出模块名
        System.out.println(mod.getName());
        new Hello().info();
        // 创建 User 对象的实例，并调用它的方法
        System.out.println(new User().sayHi("孙悟空"));
    }
}
```

上面程序中粗体字代码可通过 Class 对象获取 Hello 类所在的模块。此外，上面两个 Java 类并没有太多特别的地方，故此处不做过多解释。

将该模块示例与 5.4 节所介绍的非模块项目进行对比，不难发现添加模块功能之后，项目的文件结构只需做以下两点改变。

- 增加一个与模块同名的目录，该模块所包含的包对应的文件结构应该保存在模块目录下；而 Java 源文件依然保存在各自包所对应的文件结构下。

➢ 在模块目录下增加一个 module-info.java 模块描述文件。

接下来即可使用 Java 9 增强的 javac 编译器来编译一个或多个模块。下面为 javac 使用的两个选项。

➢ --module-source-path：指定一个或多个模块的源路径。

➢ --module-version：指定模块的版本。

在 moduleExample 目录下执行如下命令：

```
javac -d mods --module-source-path src --module-version 1.0
src\org.cimodule\module-info.java src\org.cimodule\org\crazyit\main\Hello.java
```

上面命令同时编译两个 Java 文件，并指定将生成的模块字节码文件放在 mods 目录下。此外，--module-source-path 指定 javac 到 src 目录下搜索模块，因此在 src 目录下应保存与模块名同名的子文件夹（本例只包含一个 org.cimodule 模块，因此在 src 目录下包含了该文件夹）。

执行上面命令，将会在 mods 目录下根据 org.cimodule 模块建立一个同名的目录，并在该目录下为 Java 类的包生成对应的文件结构，将 class 文件放入该目录结构下。此时可以在 mods 目录下看到如下文件结构。

```
mods
    org.cimodule
     ├─module-info.class
     └─org
         └─crazyit
             ├─domain
             │    └─User.class
             └─main
                  └─Hello.class
```

上面 org.cimodule 目录代表了 org.cimodule 模块，module-info.class 就是模块描述文件。

为了运行模块中的 Java 类，Java 9 对 java 命令也进行了增强。java 命令多了如下用法。

```
java [options] -m <模块>[/<主类>] [args...]
```

或者

```
java [options] --module <模块>[/<主类>] [args...]
```

从上面命令可以看出，--module 和-m 的效果相同。

此外，Java 9 还为 java 命令增加了如下有关模块的选项。

➢ --module-path 或-p <模块路径>：用于指定模块的加载路径。

➢ --list-modules：列出模块。

➢ --d 或--describe-module <模块名称>：用于描述指定模块。

执行如下命令：

```
java --module-path mods --list-modules
```

上面 java 命令通过--module-path 选项告诉系统在 mods 目录下搜索模块；--list-modules 选项说明要列出当前模块。运行上面命令，可以看到如图 A.2 所示的输出。

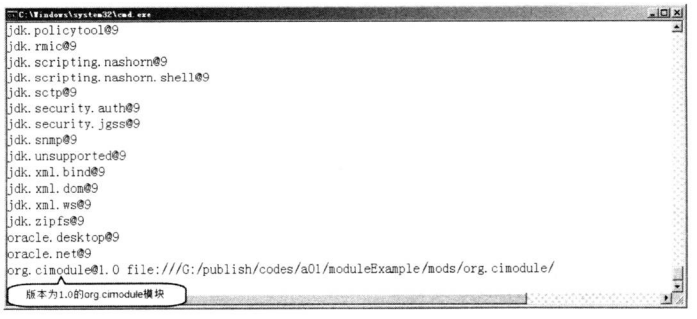


图 A.2 列出模块

从图 A.2 可以看出，系统前面列出了 JDK 9 内置的各种模块，最后一行列出了版本为 1.0 的 org.cimodule 模块，即刚刚开发的放在 mods 目录下的 org.cimodule 模块。

接下来可使用如下命令来运行程序。

```
java -p mods -m org.cimodule/org.crazyit.main.Hello
```

上面命令中的-p mods 指定 java 命令从 mods 目录下获取模块，-m 是 Java 9 增强的新用法，表明要运行模块中的 Java 类，该选项的值应该是<模块>[/<主类>]，如上命令所示。执行上面命令，可以看到如下输出：

```
org.cimodule
Hello 的 info 方法
--执行 User 的 sayHi 方法--
孙悟空您好! Wed Oct 25 18:34:38 CST 2017
```

上面第一行输出的就是 Hello 类所在的模块。

A.3 用 jar 命令打包模块

Java 9 同样增强了 jar 命令，开发者可通过 jar 命令打包模块。Java 9 为 jar 命令增加了如下与模块相关的选项。

- --module-version=VERSION：设置模块的版本。
- -p 或--module-path：设置模块的加载路径。

--module-version 选项用于设置模块的版本，此处设置的版本会覆盖 javac 命令中使用--module-version 选项指定的版本。例如，执行如下命令：

```
jar -c -v -f lib/org.cimodule-2.0.jar --module-version 2.0 -C mods/org.cimodule .
```

上面命令将会在 lib 目录下生成一个 org.cimodule-2.0.jar 文件，这就是一个模块化的 JAR 包，其中--module-version 2.0 指定该模块的版本是 2.0。

如果再次使用 java 的--list-modules 选项列出模块，将可以看到 org.cimodule 模块的版本变成了 2.0。

也可使用 java 命令直接运行打包后的模块 JAR 包，例如如下命令：

```
java -p lib -m org.cimodule/org.crazyit.main.Hello
```

从上面命令可以看出，运行 JAR 包中的模块与运行目录下的模块基本一样。原来的模块没有打包，直接将模块放在 mods 目录下，因此为 java 命令指定-p mods 选项；现在模块被打包成 JAR 包，放在 lib 目录下，因此为 java 命令指定-p lib 选项，其他的完全一样。

A.4 管理模块的依赖

模块之间的可访问性指的是两个模块之间的双向协议——模块导出指定的包供其他模块调用；反过来，模块也要明确指定需要依赖哪个模块。

模块中的所有未导出的包都是模块私有的，它们不能在模块之外被访问；反过来，模块要访问其他模块，必须明确指定依赖哪些模块，未明确指定依赖的模块不能访问。

模块导出使用 exports 语句，exports 语句的完整语法如下：

```
exports <package>;
exports <package> to <module1>, <module2>...;
```

第一种语句用于将 package 导出给任意模块调用；第二种语句用于将 package 只导出给一个或多个模块，这种导出语句被称为“限定（qualified）导出”。

模块还支持 opens 语句，opens 语句的完整语法与 exports 相似，同样支持如下两种用法：

```
opens <package>;
opens <package> to <module1>, <module2>...;
```

第一种语句用于将 package 开放给任意模块调用；第二种语句用于将 package 只开放给一个或多个模块，这种开放语句被称为“限定（qualified）开放”。

导出与开放的区别在于：被导出的包是彻底暴露的，只要访问权限允许，其他模块中的类就完全可

以自由访问被导出包中的类型（只要声明了依赖该模块）；被开放的包并不是彻底暴露的，其他模块中的类只能通过反射来访问被开放包中的类型（只要声明了依赖该模块）。

模块依赖使用 requires 语句，requires 语句的完整语法如下：

```
requires [transitive] [static] <module>;
```

requires 语句中的 static 修饰符表示该依赖模块在编译时是必需的，但在运行时则是可选的。比如在模块 P 中声明如下 requires 语句：

```
requires static Q;
```

它表明程序编译模块 P 时必须依赖模块 Q；但模块 P 在运行时，模块 Q 是可选的。

requires 语句中的 transitive 修饰符表明该依赖具有传递性。假如现在有三个模块：P、Q 和 R，模块 P 依赖模块 Q，而在模块 Q 中声明了如下语句：

```
requires transitive R;
```

这意味着模块 Q 对模块 R 的依赖具有传递性，既然模块 P 依赖模块 Q，那么模块 P 也依赖模块 R。

在 Java 9 模块化系统的表达中，以下三个术语的含义是相同的：需要（require）、读取（read）和依赖（depend）。对于 P、Q 两个模块，以下三个语句的含义是相同的：P 读取 Q，P 需要 Q，P 依赖 Q。

下面示例将会定义两个模块，其中第一个模块包含两个包，一个包被导出（exports），一个包只是被开放（opens）；另一个模块会声明依赖第一个模块，并通过合适方式来使用第一个模块的两个包中的类。

首先在当前工作目录（moduleDepend）下创建两个子目录。

- -src：用于保存源代码。
- -mods：用于保存所生成的字节码文件。

然后创建一个名为 org.cimodule 的模块，因此在 src 目录下新建一个名为 org.cimodule 的文件夹。接下来在 org.cimodule 目录下新建一个 module-info.java 文件，该文件内容如下。

程序清单：codes\a01\moduleDepend\src\org.cimodule\module-info.java

```
module org.cimodule
{
    exports org.crazyit.user;
    opens org.crazyit.shop;
}
```

上面第一行粗体字代码为该模块声明“导出”org.crazyit.user 包，这意味着该包中所有的类型都可以被其他模块自由访问（只要访问权限允许）；第二行粗体字代码为该模块声明“开放”org.crazyit.shop 包，这意味着该包中所有的类型都可以被其他模块通过反射访问（只要访问权限允许）。

也可将上面两行粗体字代码改为如下形式：

```
exports org.crazyit.user to org.fkmodule;
opens org.crazyit.shop to org.fkmodule;
```

上面两行代码分别代表“限定导出”和“限定开放”，其中第一行代码表明 org.crazyit.user 只“导出”给 org.fkmodule 模块，因此只有 org.fkmodule 模块中的类型可自由访问 org.crazyit.user 包中的类型，其他模块不行；第二行代码表明 org.crazyit.shop 只“开放”给 org.fkmodule 模块，因此只有 org.fkmodule 模块中的类型可通过反射访问 org.crazyit.shop 包中的类型，其他模块不行。

接下来在 org.cimodule 模块下添加两个类，即位于 org.crazyit.user 包下的 User 类和位于 org.crazyit.shop 包下的 Item 类。当然，这两个类的源文件也应该放在 src\org.cimodule 路径下对应的文件结构中。由于这两个类的代码比较简单，此处不再给出。

再创建一个名为 org.fkmodule 的模块，因此在 src 目录下新建一个名为 org.fkmodule 的文件夹。由于该模块中的 Main 类打算调用 org.cimodule 模块中的 User 类和 Item 类，因此 org.fkmodule 模块需要依赖 org.cimodule 模块。

在 org.cimodule 目录下新建一个 module-info.java 文件，通过该模块描述文件声明 org.fkmodule 模块

依赖 org.cimodule 模块。该模块描述文件的内容如下。

程序清单：codes\a01\moduleDepend\src\org.fkmodule\module-info.java

```
module org.fkmodule
{
    requires org.cimodule;
}
```

上面程序中粗体字代码声明了 org.fkmodule 模块需要依赖 org.cimodule 模块，这样 org.fkmodule 模块才可使用 org.cimodule 模块中的类型。

接下来在 org.fkmodule 模块下添加一个位于 org.crazyit.main 包下的 Main 类，该 Main 类的代码如下。

程序清单：codes\a01\moduleDepend\src\org.fkmodule\org\fkjava\main\Main.java

```
public class Main
{
    public static void main(String[] args)throws Exception
    {
        // org.crazyit.shop 包中的类只是声明为“导出（exports）”，
        // 因此可以自由访问 User 类
        User user = new User();
        System.out.println(user.addUser("yeeku"));
        // org.crazyit.shop 包中的类只是声明为“开放（opens）”，
        // 因此只能通过反射访问该包中的 Item 类
        Class<?> clazz = Class.forName("org.crazyit.shop.Item");
        Object im = clazz.getConstructor().newInstance();
        Method mtd = clazz.getMethod("showInfo");
        mtd.invoke(im);
    }
}
```

由于 org.fkmodule 模块声明了依赖 org.cimodule 模块，因此 org.fkmodule 模块中的类型可使用 org.cimodule 模块中的 User 类和 Item 类。但由于 User 类位于 org.crazyit.user 包下，而该包是被导出的，因此 Main 类可直接使用 User 类；而 Item 类位于 org.crazyit.shop 包下，而该包只是被“开放”，因此 Main 类只能通过反射来使用 Item 类。

执行如下命令编译两个模块：

```
javac -d mods --module-source-path src --module-version 1.0 src\org.cimodule\org\crazyit\
shop\Item.java src\org.fkmodule\org\fkjava\main\Main.java
```

执行上面的编译命令时，程序会自动编译 Main.java 和 Item.java 源文件。由于 Main.java 用到了 User 类，因此编译器也会自动编译 User.java 源文件。

编译之后生成如下文件结构：

```
mods
    org.cimodule
    ├─module-info.class
    └─org
        └─crazyit
            ├─user
            │    └─User.class
            └─shop
                 └─Item.class
    org.fkmodule
    ├─module-info.class
    └─org
        └─fkjava
            └─main
                 └─Main.class
```

从上面的文件结构可以看出，此处在mods目录下包含了两个模块：org.cimodule模块和org.fkmodule模块。

执行如下命令来运行Main类：

```
java -p mods --module org.fkmodule/org.fkjava.main.Main
```

执行上面命令，可以看到org.fkmodule模块中的Main类成功调用了org.cimodule模块中的User类和Item类。

根据上面介绍可以看出，Java 9的模块化系统丰富了访问权限的功能。在Java 8中，使用public修饰的类型，意味着它是真正公共的，可以被任意类型自由调用；但在Java 9中，使用public修饰的类型，并不一定是真正公共的。public类型可产生如下三种情形。

- public类型所在的包被导出或开放，该public类型可被任意模块访问。
- public类型所在的包被限定导出或开放给指定模块，该public类型只能被特定模块访问。
- public类型所在的包没有被导出或开放，该public类型只能在当前模块中被访问。

如果程序在Main类的import部分添加如下导包语句：

```
import java.sql.*;
```

再次使用上面的javac命令编译两个模块，将可以看到如下错误提示：

```
src\org.fkmodule\org\fkjava\main\Main.java:6: 错误: 程序包 java.sql 不可见
import java.sql.*;
         ^
  (程序包 java.sql 已在模块 java.sql 中声明,但模块 org.fkmodule 未读取它)
1 个错误
```

从上面的错误提示可以看出，一旦开始使用Java 9的模块化系统，我们所开发的模块就不会自动加载整个JRE，它只加载JRE的核心模块：java.base，这样该模块程序运行时就可以节省系统开销。

如果程序需要使用java.sql的功能，则需要在该模块的module-info.java文件中声明该模块需要依赖java.sql模块，这样该模块运行时会同时加载java.base模块和java.sql模块——依然不需要加载整个JRE，运行该程序的系统开销同样比较小。

提示：

所有Java模块在编译、运行时总会自动加载java.base模块。

在org.fkmodule模块的module-info.java文件中增加如下一行：

```
requires java.sql;
```

再次使用上面的javac命令来编译两个模块，将可以看到这两个模块都可以编译成功，这意味着当org.fkmodule编译和运行时都需要依赖java.sql模块。

某些时候，如果开发的模块确实需要使用整个Java SE的全部功能，则可直接声明依赖java.se模块。java.se模块是一个不包含任何Java类型的模块，它只是负责收集并重新导出其他模块的内容。例如，java.se模块的module-info.java文件的内容片段如下：

```
module java.se {
    requires transitive java.sql;
    requires transitive java.rmi;
    requires transitive java.desktop;
    requires transitive java.security.jgss;
    requires transitive java.security.sasl;
    requires transitive java.management;
    requires transitive java.logging;
    requires transitive java.xml;
    requires transitive java.scripting;
    requires transitive java.compiler;
    requires transitive java.naming;
    requires transitive java.instrument;
    requires transitive java.xml.crypto;
```

```
    requires transitive java.prefs;
    requires transitive java.sql.rowset;
    requires java.base;
    requires transitive java.datatransfer;
}
```

像 java.se 这样的模块被称为“聚合模块”。

实际上，开发者同样可以定义自己的“聚合模块”。假设项目中几个模块都依赖另外 9 个模块，此时就可将这 9 个模块创建成一个聚合模块，然后这些模块只要依赖这个“聚合模块”即可。

A.5 实现服务

从 Java SE 6 开始，Java 提供了一种服务机制，允许服务提供者和服务使用者之间完全解耦。简单来说，就是服务使用者只面向服务接口编程，但并不清楚服务提供者的实现类。

Java 9 的模块化系统则进一步简化了 Java 的服务机制。Java 9 允许将服务接口定义在一个模块中，并使用 uses 语句来声明该服务接口；然后针对该服务接口提供不同的服务实现类，这些服务实现类可分布在不同的模块中，服务实现模块则使用 provides 语句为服务接口指定实现类。

定义服务接口的模块与定义服务实现的模块是完全分离的，系统可根据需要任意添加、删除一个实现模块——代码无须任何修改，甚至配置文件都无须修改。而服务使用者则更彻底，它只需要面向服务接口编程，只依赖包含服务接口的模块，压根就不知道包含服务实现类的模块。图 A.3 显示了 Java 9 模块化系统的服务架构示意图。

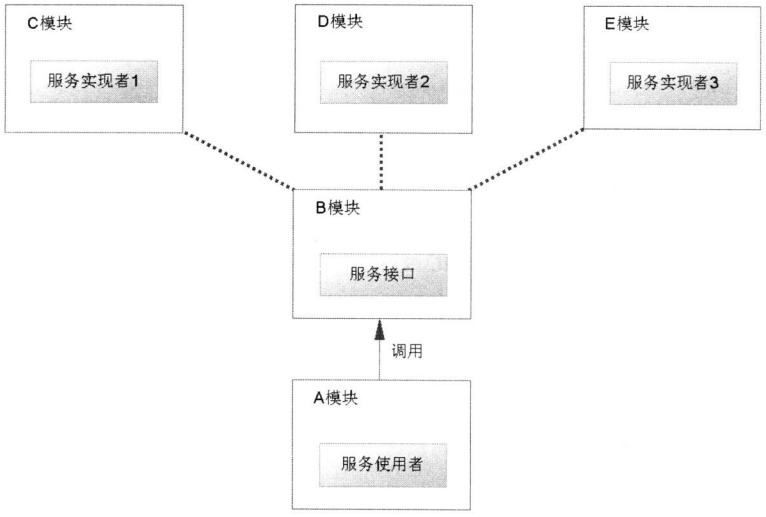


图 A.3 模块化系统的服务架构示意图

从图 A.3 可以看出，模块 A 是服务使用者，模块 B 包含服务接口（用 uses 语句声明可使用服务），因此模块 A 需要依赖、调用模块 B。模块 A 不需要依赖任何服务实现者所在的模块，也就是说，模块 A 与模块 C、模块 D、模块 E 之间没有任何关系。而且，模块 B 与模块 C、模块 D、模块 E 之间也没有任何关系，因此模块 C、模块 D、模块 E 完全可以随时添加或删除，这种操作对系统本身没有任何影响——只是服务接口增加或减少一个实现而已。

下面示例先定义一个包含 UserService 服务接口的 org.cimodule 模块——在 src 目录下创建 org.cimodule 文件夹，并在该文件夹下创建模块描述文件：module-info.java。该文件内容如下。

程序清单：codes\a01\moduleService\src\org.cimodule\module-info.java

```
module org.cimodule
{
    // 导出 org.crazyit.oa 包，以便其他包能使用该包下的服务接口
    exports org.crazyit.oa;
    // 声明该模块提供 UserService 服务接口
    uses org.crazyit.oa.UserService;
}
```

上面模块描述文件的第一行导出了 org.crazyit.oa 包，以便其他模块可使用该包下的 Java 类型。上面的粗体字代码使用 uses 语句声明该模块包含 UserService 服务接口。

接下来为 org.cimodule 模块定义 UserService 服务接口，该接口代码如下。

程序清单：codes\a01\moduleService\src\org.cimodule\org\crazyit\oa\UserService.java

```
public interface UserService
{
    Integer addUser(String name);
    String getImplName();
    static UserService newInstance(){
        // 通过 ServiceLoader 加载所有服务实现者
        return ServiceLoader.load(UserService.class)
            // 返回第一个服务实现者
            .findFirst()
            .orElseThrow(() -> new IllegalArgumentException(
                "找不到默认的服务实现者！"));
    }
    static UserService newInstance(String providerName){
        // 通过 ServiceLoader 加载所有服务实现者
        ServiceLoader<UserService> sl = ServiceLoader.load(UserService.class);
        // 遍历所有服务实现者
        for (UserService us : sl)
        {
            if (us.getImplName().equalsIgnoreCase(providerName))
            {
                return us;
            }
        }
        throw new IllegalArgumentException("无法找到名为'"
            + providerName + "'的服务实现者！");
    }
}
```

上面 UserService 接口定义了 addUser()和 getImplName()两个抽象方法，这两个抽象方法将会由该接口的实现类负责实现。此外，在该接口中还定义了两个 newInstance()方法，其中第一个 newInstance()方法总是返回默认的服务实例；第二个 newInstance()方法将根据名称来返回服务实例。

为了实现上面的 newInstance()方法，上面程序用到一个 ServiceLoader 工具类，该工具类在 Java 6 中就已经提供，但 Java 9 增强了该工具类的功能，ServiceLoader 会读取系统各模块的 module-info.java 描述文件，获取服务接口的所有服务实现。

例如上面程序中的如下语句：

```
ServiceLoader<UserService> sl = ServiceLoader.load(UserService.class);
```

上面代码将会读取系统各模块为 UserService 提供所有实现类，ServiceLoader 本身实现了 Iterable 接口，因此程序可用 foreach 循环来迭代 ServiceLoader 所包含的全部服务实现类。

从上面代码可以看出，UserService 接口此时并不清楚系统到底包含多少个服务实现类。只要系统通过模块来提供服务实现者，ServiceLoader 的 load()方法就会自动加载它们。

接下来程序会为 UserService 服务接口提供实现，服务接口的实现类必须遵守如下要求。

- 服务实现类包含无参的构造器，ServiceLoader 将通过该无参的构造器来创建服务实现者的实例。在这种情况下，服务实现类必须实现服务接口。
- 服务实现类不包含无参的构造器，但服务实现类包含一个无参的、public static 修饰的 provider 方法，ServiceLoader 将通过该方法来创建服务实现者的实例，因此该方法返回的对象必须实现服务接口。

下面使用 org.cimodule.basic 模块为 UserService 接口提供第一个实现类。在 src 目录下创建 org.cimodule.basic 文件夹，并在该文件夹下创建模块描述文件：module-info.java。该文件内容如下。

程序清单：codes\a01\moduleService\src\org.cimodule.basic\module-info.java

```
import org.crazyit.oa.UserService;
```

```
import org.crazyit.oa.impl.UserServiceImpl;
module org.cimodule.basic
{
    // 指定依赖服务接口所在的模块
    requires org.cimodule;
    // 为 UserService 服务接口提供 UserServiceImpl 实现类
    provides UserService with UserServiceImpl;
}
```

上面模块描述文件的第一行指定该模块依赖 org.cimodule 模块，这样该模块即可使用 org.cimodule 模块下的 UserService 接口（实现类肯定要实现 UserService 接口）。上面的粗体字代码使用 provides 语句声明为 UserService 服务接口使用 UserServiceImpl 实现类——当系统包含该模块时，前面介绍的 ServiceLoader 类就会读取 module-info.java 中的此行声明，为 UserService 服务加载一个服务实现者。

UserServiceImpl 类非常简单，该类由系统自动生成一个无参的构造器，并实现 UserService 接口中的两个抽象方法。UserServiceImpl 类的代码如下。

程序清单：codes\a01\moduleService\src\org.cimodule.basic\org\crazyit\oa\impl\UserServiceImpl.java

```
public class UserServiceImpl implements UserService
{
    static final String IMPL_NAME = "basic user service";
    public Integer addUser(String name)
    {
        System.out.println("普通的 UserService 实现添加用户：" + name);
        return 19;
    }
    @Override
    public String getImplName()
    {
        return IMPL_NAME;
    }
}
```

该 UserServiceImpl 提供了无参的构造器，ServiceLoader 将会使用该构造器创建服务实现者的实例，因此该 UserServiceImpl 必须实现 UserService 接口。

下面使用 org.cimodule.senior 模块为 UserService 接口提供第二个实现类。在 src 目录下创建 org.cimodule.senior 文件夹，并在该文件夹下创建模块描述文件：module-info.java。该文件内容如下。

程序清单：codes\a01\moduleService\src\org.cimodule.senior\module-info.java

```
import org.crazyit.oa.UserService;
import org.crazyit.oa.senior.UserServiceSenior;
module org.cimodule.senior
{
    // 指定依赖服务接口所在的模块
    requires org.cimodule;
    // 为 UserService 服务接口提供 UserServiceSenior 实现类
    provides UserService with UserServiceSenior;
}
```

与前面介绍的 org.cimodule.basic 模块类似，模块描述文件的第一行指定该模块依赖 org.cimodule 模块。上面的粗体字代码使用 provides 语句声明为 UserService 服务接口使用 UserServiceSenior 实现类——当系统包含该模块时，ServiceLoader 也会加载 UserServiceSenior 类作为 UserService 服务接口的实现。

该服务实现类 UserServiceSenior 不再提供无参的构造器，而是提供一个 public staic 修饰的、无参的 provider()方法。UserServiceSenior 类的代码如下。

程序清单：codes\a01\moduleService\src\org.cimodule.senior\org\crazyit\oa\senior\UserServiceSenior.java

```
public class UserServiceSenior implements UserService
{
    static final String IMPL_NAME = "senior user service";
    // 构造器私有
    private UserServiceSenior(){}
    // 通过 static、无参的 provider 方法来返回服务实现者对象
    public static UserService provider()
```

```
    {
        return new UserServiceSenior();
    }
    public Integer addUser(String name)
    {
        System.out.println("===高级的 UserService 实现添加用户: " + name);
        return 29;
    }
    @Override
    public String getImplName()
    {
        return IMPL_NAME;
    }
}
```

该 UserServiceSenior 并未提供无参的构造器，而是提供了 public static 修饰的、无参的 provider()方法，因此 ServiceLoader 将会使用该 provider()创建服务实现者的实例。由于 provider()方法直接返回 UserServiceSenior 的实例，因此 UserServiceSenior 也必须实现 UserService 接口。

下面使用 org.cimodule.best 模块为 UserService 接口提供第三个实现类。在 src 目录下创建 org.cimodule.best 文件夹，并在该文件夹下创建模块描述文件：module-info.java。该文件内容如下。

程序清单：codes\a01\moduleService\src\org.cimodule.best\module-info.java

```
import org.crazyit.oa.UserService;
import org.crazyit.oa.best.UserServiceBest;
module org.cimodule.best
{
    // 指定依赖服务接口所在的模块
    requires org.cimodule;
    // 为 UserService 服务接口提供 UserServiceBest 实现类
    provides UserService with UserServiceBest;
}
```

上面的模块描述文件与前面两个 module-info.java 文件大致相似，此处不再详细解释。

本服务实现类 UserServiceBest 也不提供无参的构造器，而是提供一个 public staic 修饰的、无参的 provider()方法。UserServiceBest 类的代码如下。

程序清单：codes\a01\moduleService\src\org.cimodule.best\org\crazyit\oa\best\UserServiceBest.java

```
public class UserServiceBest
{
    static final String IMPL_NAME = "best user service";
    // 通过 static、无参的 provider 方法来返回服务实现者对象
    public static UserService provider()
    {
        return new UserService(){
            public Integer addUser(String name)
            {
                System.out.println("======最好的 UserService 实现添加用户: " + name);
                return 47;
            }
            @Override
            public String getImplName()
            {
                return IMPL_NAME;
            }
        };
    }
}
```

该 UserServiceBest 并未提供无参的构造器，而是提供了 public static 修饰的、无参的 provider()方法，因此 ServiceLoader 将会使用该 provider()创建服务实现者的实例。

此处程序在 provider()方法中使用匿名内部类创建了 UserService 实现类的实例，这也是允许的。在这种情况下，UserServiceBest 不需要实现 UserService 接口。

通过上面介绍不难看出，org.cimodule 模块负责提供服务接口，该模块的 module-info.java 文件使用

uses org.crazyit.oa.UserService 声明该模块的服务接口；而 org.cimodule.basic、org.cimodule.senior、org.cimodule.best 模块负责为服务接口提供实现，因此这三个模块的 module-info.java 文件都使用 provides 语句为服务接口提供实现类——而 ServiceLoader 则读取 module-info.java 中的 provides 语句，根据它们为服务接口加载实现类。

服务客户端只需依赖服务接口所在的模块，与服务实现者所在的模块没有任何关联。下面使用 org.fkmodule 模块作为服务客户端。在 src 目录下新建一个 org.fkmodule 文件夹，并在该文件夹下新建一个 module-info.java 模块描述文件。该文件内容如下。

程序清单：codes\a01\moduleService\src\org.fkmodule\module-info.java

```
module org.fkmodule
{
    // 指定依赖服务接口所在的模块
    requires org.cimodule;
}
```

从上面的粗体字代码可以看出，服务客户端只需要依赖服务接口所在的模块。服务客户端代码可通过 UserService 的 newInstance()方法来获取服务实现者实例。下面是服务客户端的代码。

程序清单：codes\a01\moduleService\src\org.fkmodule\org\fkjava\oa\client\Client.java

```
public class Client
{
    public static void main(String[] args)
    {
        // 使用默认的服务提供者，具体使用哪个不确定
        UserService us1 = UserService.newInstance();
        System.out.println(us1.addUser("yeeku"));
        // 获取"basic user service"服务实现者
        UserService us2 = UserService.newInstance("basic user service");
        System.out.println(us2.addUser("yeeku"));
        // 获取"senior user service"服务实现者
        UserService us3 = UserService.newInstance("senior user service");
        System.out.println(us3.addUser("yeeku"));
        // 获取"best user service"服务实现者
        UserService us4 = UserService.newInstance("best user service");
        System.out.println(us4.addUser("yeeku"));
    }
}
```

上面的 4 行粗体字代码分别用于获取不同的服务实现者实例，其中第一行粗体字代码获取 UserService 服务接口的“第一个”被找到的服务实现者，因此第一行粗体字代码所返回的服务实现者是不确定的。

使用如下命令来编译所有模块：

```
javac -d mods --module-source-path src
src\org.cimodule.basic\org\crazyit\oa\impl\UserServiceImpl.java
src\org.cimodule.senior\org\crazyit\oa\senior\UserServiceSenior.java
src\org.cimodule.best\org\crazyit\oa\best\UserServiceBest.java
src\org.fkmodule\org\fkjava\oa\client\Client.java
```

该命令将会编译该系统的 5 个模块（服务接口一个模块、三个服务实现类各自一个模块、服务客户端一个模块），此时系统为 UserService 服务接口提供了三个服务实现类，正如上面 Client 程序的后三行粗体字代码所对应的服务实现类。

在保证 5 个模块都在的情况下，通过如下命令执行服务客户端模块的 Client 程序。

```
java -p mods -m org.fkmodule/org.fkjava.oa.client.Client
```

程序将会产生如下输出：

```
===高级的 UserService 实现添加用户：yeeku
29
普通的 UserService 实现添加用户：yeeku
19
```

```
===高级的 UserService 实现添加用户：yeeku
29
======最好的 UserService 实现添加用户：yeeku
47
```

从上面第一行输出可以看到：UserService 的 newInstance()方法默认返回的 UserServiceSenior 实现类；后面输出则清楚地显示了三个模块分别为 UserService 接口提供的服务实现类。

读者可以试着删除 org.cimodule.basic、org.cimodule.senior、org.cimodule.best 三个模块的其中一个或多个，再次运行上面的 Client 程序，将会看到程序并不会受到太大的影响——只是 UserService 少了一个或多个服务实现者而已，因此只要把对应的 newInstance()代码删除即可。

由此可见，通过模块管理服务接口和服务实现者非常方便，服务接口与服务实现者实现了彻底解耦，服务客户端也与服务实现者实现了彻底解耦，因此系统可以随时根据需要添加或删除服务实现者模块。