第 1 部分



Java 基础程序设计

- Java 语言介绍
- 简单的 Java 程序
- Java 中的基本数据类型
- 运算符、表达式与语句
- 选择与循环结构
- 数组与方法的使用

第 1 章　Java 概述及开发环境搭建

通过本章的学习可以达到以下目标：

- ☑ 认识 Java 并了解其发展历史。
- ☑ 可以安装并配置 Java 开发环境。
- ☑ 掌握 Java 中 Path 及 classpath 属性的作用。
- ☑ 可以编写并运行一个简单的 Java 程序。
- ☑ 可以掌握 Java 的开发流程。

Java 是一门程序设计语言，其本身的发展经过了一系列的过程。本章将首先介绍 Java 的发展历程，让读者对 Java 有一个基本的认识，了解 Java 的主要作用，接着详细讲解 Java 开发环境的搭建，包括 Java 开发中一些基本的注意事项。**本章视频录像讲解时间为 35 分钟，源代码在光盘对应的章节下。**

1.1　认识 Java

1.1.1　什么是 Java

Java 是 Sun（全称为 Stanford University Network，1982 年成立，Sun 公司的 Logo 如图 1-1 所示）公司开发出来的一套编程语言，主设计者是 James Gosling（如图 1-2 所示），最早来源于一个叫 Green 的项目，这个项目原来的目的是为家用电子消费产品开发一个分布式代码系统，这样就可以通过网络对家用电器进行控制。开始 Sun 的工程师们准备采用 C++，但由于 C++过于复杂，安全性差，最后决定基于 C++开发一种新语言 Oak（Java 的前身，1991 年出现），Oak 是一种用于网络的、精巧而安全的语言，Sun 公司曾以此投标过一个交互式电视项目，但结果被 SGI 打败。于是当时的 Oak 几乎“无家可归”，恰巧这时 Mark Ardreesen 开发的 Mosaic 和 Netscape 启发了 Oak 项目组成员，Sun 的工程师们用 Java 编制了 HotJava 浏览器，得到了 Sun 公司首席执行官 Scott McNealy 的支持，触发了 Java 进军 Internet。



图 1-1　Sun 公司 Logo



图 1-2　James Gosling

Java 语言发展到今天经历了一系列的过程，Java 在 1995 年 5 月 23 日推出了 JDK 1.0 版本，此版本标志着 Java 正式进军 Internet 的开始，在 1998 年对之前的 JDK 进行了升级并推出了 JDK 1.2 的开发包，该版本加入了大量的轻量级组件包，从此之后 Java 被正式命名为 Java 2。

Java 语言发展到今天经历了以下 3 个发展方向。

- J2SE：Java 2 Platform Standard Edition。包含构成 Java 语言核心的类，如数据库连接、接口定义、输入/输出和网络编程。
- J2ME：Java 2 Platform Micro Edition。包含 J2SE 中一部分类，用于消费类电子产品的软件开发，如呼机、智能卡、手机、PDA 和机顶盒。
- J2EE：Java 2 Platform Enterprise Edition。Enterprise Edition（企业版）包含 J2SE 中的所有类，并且还包含用于开发企业级应用的类，如 EJB、Servlet、JSP、XML 和事务控制，也是现在 Java 应用的主要方向。

虽然 Java 语言的发展方向有 3 个，但是这 3 门技术中最核心的部分是 J2SE，而 J2ME 和 J2EE 是在 J2SE 基础之上发展起来的，3 种技术的关系如图 1-3 所示。另外要提醒读者的是，在 2005 年 Java 十周年大会之后这 3 门技术又重新更名：

- J2SE 更名为 JAVA SE。
- J2ME 更名为 JAVA ME。
- J2EE 更名为 JAVA EE。

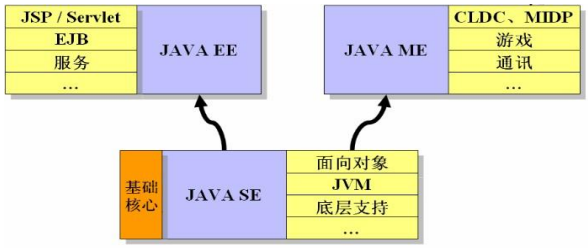


图 1-3　3 种技术的关系

1.1.2　Java 语言的特点

Java 语言的许多有效的特性吸引着程序员们，下面介绍最主要的几个。

1．简洁有效

Java 语言是一种相当简洁的“面向对象”程序设计语言，它省略了 C++语言中所有的难以理解、容易混淆的特性，如头文件、指针、结构、单元、运算符重载和虚拟基础类等，更加严谨、简洁。

2．可移植性

对于一个程序员而言，写出来的程序如果不需要修改就能够同时在 Windows、MacOS、UNIX 等平台上运行，简直就是美梦成真，而 Java 语言就让这个原本遥不可及的事已经越来越近了。使用 Java 语言编写的程序，只要做较少的修改，甚至有时根本不需修改即可在不同平台上运行。

3. 面向对象

可以这么说，“面向对象”是软件工程学的一次革命，大大提升了人类的软件开发能力，是一个伟大的进步，是软件发展的一个重大的里程碑。

在过去的 30 年间，“面向对象”有了长足的发展，充分体现了其自身的价值，到现在已经形成了一个包含“面向对象的系统分析”、“面向对象的系统设计”和“面向对象的程序设计”的完整体系。所以作为一种现代编程语言，是不能够偏离“面向对象”这一方向的，Java 语言也不例外。

4. 解释型

Java 语言是一种解释型语言，相对于 C/C++语言来说，用 Java 语言写出来的程序效率低，执行速度慢。但它可以通过在不同平台上运行 Java 解释器，对 Java 代码进行解释，来实现“一次编写，到处运行”的目标。为了达到目标，牺牲效率还是值得的，而且，现在的计算机技术日新月异，运算速度也越来越快，用户不会感到太慢。

5. 适合分布式计算

Java 语言具有强大的、易于使用的联网能力，非常适合开发分布式计算的程序。Java 应用程序可以像访问本地文件系统那样通过 URL 访问远程对象。

使用 Java 语言编写 Socket 通信程序比使用任何其他语言都简单。而且它还十分适用于公共网关接口（CGI）脚本的开发，另外还可以使用 Java 小应用程序（Applet）、Java 服务器页面（Java Server Page，JSP）、Servlet 等手段来构建更丰富的网页。

6. 拥有较好的性能

由于 Java 是一种解释型语言，所以它的执行效率相对就会慢一些，但由于 Java 语言采用了下面两种手段，使其拥有较好的性能。

- Java 语言源程序编写完成后，先使用 Java 伪编译器进行伪编译，将其转换为中间码（也称为字节码）再解释。
- 提供了一种“准实时”（Just-in-Time，JIT）编译器，当需要更快的速度时，可以使用 JIT 编译器将字节码转换成机器码，然后将其缓冲下来，这样速度就会更快。

7. 健壮、防患于未然的特征

Java 语言在伪编译时，做了许多早期潜在问题的检查，并且在运行时又做了一些相应的检查，可以说是一种最严格的“编译器”。

它的这种“防患于未然”的手段将许多程序中的错误都扼杀在“摇篮”之中，经常有许多在其他语言中必须通过运行才会暴露出来的错误，在编译阶段就被发现了。

另外，在 Java 语言中还具备了许多保证程序稳定、健壮的特性，有效地减少了错误，也使 Java 应用程序更加健壮。

8. 具有多线程处理能力

线程是一种轻量级进程，是现代程序设计中必不可少的一种特性。多线程处理能力使

程序具有更好的交互性、实时性。

Java 在多线程处理方面性能超群，具有让设计者惊喜的强大功能，而且在 Java 语言中进行多线程处理也很简单。

9．具有较高的安全性

Java 语言在设计时在安全性方面考虑得很仔细，做了许多探究，使 Java 语言成为目前最安全的一种程序设计语言。

尽管 Sun 公司曾经许诺过：“通过 Java 可以轻松构建出防病毒、防黑客的系统”，但就在 JDK（Java Development Kit）1.0 发布不久后，美国 Princeton（普林斯顿）大学的一组安全专家发现了 Java 1.0 安全特性中的第一例错误。从此，Java 安全方面的问题开始被关注。不过至今所发现的安全隐患都很微不足道，而且 Java 开发组还宣称，他们对系统安全方面的 Bugs 非常重视，会对这些被发现的 Bugs 立即进行修复。另外，由于 Sun 公司开放了 Java 解释器的细节，所以有助于通过各界力量共同发现、防范、制止这些安全隐患。

10．是一种动态语言

Java 是一种动态的语言，这表现在以下两个方面：

- 在 Java 语言中，可以简单、直观地查询运行时的信息。
- 可以将新代码加入到一个正在运行的程序中。

11．是一种中性结构

Java 编译器生成的是一种中性的对象文件格式，也就是说，Java 编译器通过伪编译后，将生成一个与任何计算机体系统无关的“中性”的字节码。

这种中性结构其实并不是 Java 首创的，在 Java 出现之前 UCSD Pascal 系统就已在一种商业产品中做到了这一点，另外，在 UCSD Pascal 之前也有这种方式的先例，如在 Niklaus Wirth 实现的 Pascal 语言中就采用了这种结构降低一些性能，以换取更好的可移植性和通用性的方法。

Java 的这种字节码经过了许多精心的设计，使其能够很好地兼容于当今大多数流行的计算机系统，在任何机器上都易于解释，易于动态翻译成为机器代码。

1.1.3　Java 程序的运行机制和 Java 虚拟机

计算机高级语言类型主要有编译型和解释型两种，Java 是两种类型的集合，在 Java 中处理代码的过程如图 1-4 所示。

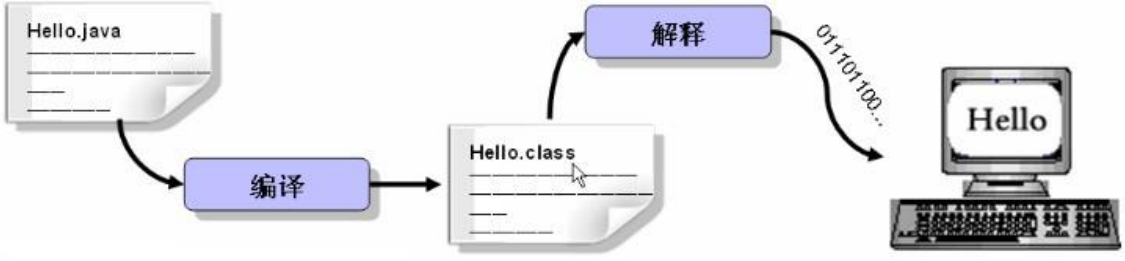


图 1-4　Java 程序的运行机制

从图 1-4 中可以看出，在 Java 中源文件名称的后缀为.java，之后通过编译使*.java 的文件生成一个*.class 文件，在计算机上执行*.class 文件，但是此时执行*.class 的计算机并不是一个物理上可以看到的计算机，而是 Java 自己设计的一个计算机——JVM，Java 也是通过 JVM 进行可移植性操作的。

在 Java 中所有的程序都是在 JVM（Java Virtual Machine）上运行的。JVM 是在一台计算机上由软件或硬件模拟的计算机。Java 虚拟机（JVM）读取并处理经编译过的、与平台无关的字节码 class 文件。Java 解释器负责将 Java 虚拟机的代码在特定的平台上运行。JVM 的基本原理如图 1-5 所示。

从图 1-5 中可以发现，所有的*.class 文件都是在 JVM 上运行的，即*.class 文件只需要认识 JVM，由 JVM 再去适应各个操作系统。如果不同的操作系统安装上符合其类型的 JVM，那么以后程序无论到哪个操作系统上都是可以正确执行的。

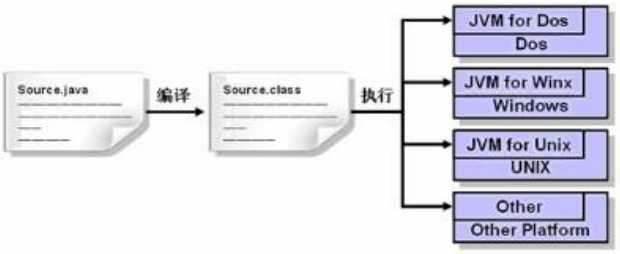


图 1-5　JVM 基本原理

提示：关于 Java 可移植性的简单理解。

有些读者可能很难理解以上的解释，其实这个过程就类似于下面的一种情景：

现在有一个中国富商，他同时要和美国、韩国、俄罗斯、日本、法国、德国等几个国家洽谈生意，可是他不懂这些国家的语言，所以他针对于每个国家请了一个翻译，他说的话就只对翻译说，不同的翻译会将他说的话翻译给不同国家的客户，这样富商只需要说一句话给翻译，那么就可以同几个国家的客户沟通了。

1.2　Java 开发环境搭建

Java 的开发环境主要使用 JDK，本书中使用的 JDK 版本是 JDK 1.6 版本。读者可以直接从 Sun 公司的官方网站（如图 1-6 所示）中下载此版本，网址为 www.sun.com。



图 1-6　Sun 公司网站首页

1.2.1　JDK 的安装与配置

安装 JDK 分为以下两个步骤：

（1）首先要准备好 JDK 的安装文件 jdk-6u11-windows-i586-p.exe。

（2）配置环境变量 Path。

在安装 JDK 时会让用户选择 JDK 和 JRE 的安装目录，本次安装目录使用的是默认的安装目录，如图 1-7～图 1-10 所示。

JDK 安装完成之后，即可看到如图 1-10 所示的安装文件夹，其中的 bin 文件夹是将来要使用的各种 Java 命令，但是这些命令本身并不在 Windows 环境之中，所以如果要想使用这些命令，则首先必须在 Windows 中注册此命令。



图 1-7　选择 JDK 的安装目录

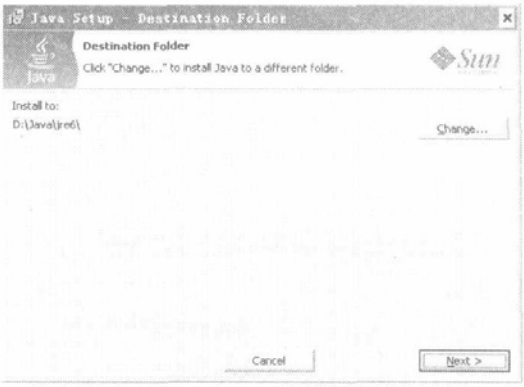


图 1-8　选择 JRE 的安装目录

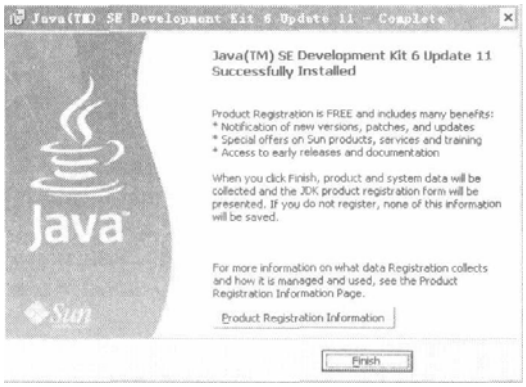


图 1-9　JDK 和 JRE 安装完成



图 1-10　JDK 安装后目录

设置 Path 的流程为：选择【我的电脑】→【属性】命令，打开【系统属性】对话框，如图 1-11 所示，选择【高级】选项卡，单击【环境变量】按钮，打开【环境变量】对话框，在其中可以编辑 Path 信息，如图 1-12 所示。

在编辑 Path 时，可以直接设置 Java 的安装路径。即在【环境变量】对话框中单击【系统变量】栏中的【编辑】按钮，弹出【编辑系统变量】对话框，在【变量值】文本框的最后加上路径，在加之前使用“;”将之前的路径分隔开，如加入内容“; D:\Java\jdk1.6.0_11\bin”，如图 1-13 所示。

设置完后单击【确定】按钮即可保存设置，然后启动 cmd 命令行方式，输入 javac，如

果出现如图 1-14 所示的内容，则表示 JDK 配置成功。

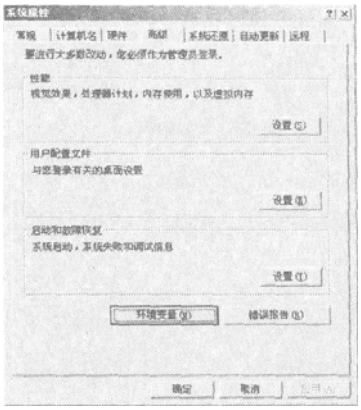


图 1-11 【系统属性】对话框

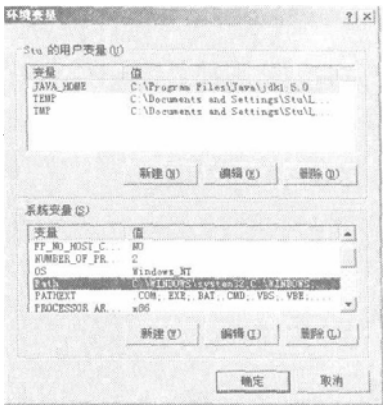


图 1-12 【环境变量】对话框

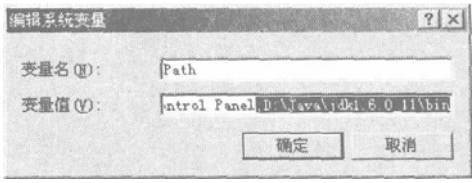


图 1-13 设置 path 路径

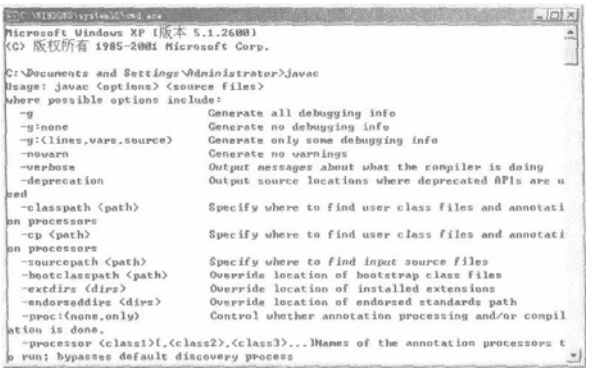


图 1-14 JDK 配置成功演示

注意 1：Path 修改之后要重新启动命令行方式。

有些读者在进行环境设置时可能会出现此命令不是系统内部命令的提示，有可能造成这种问题的原因是，在配置环境属性之前命令行方式已经启动，之后再配置的 Path 路径，则此环境肯定是无法立即生效的，此时可以重新启动命令行方式，即可把新的设置读取进来。

注意 2：javac 命令的作用。

javac.exe 是 Java 本身提供的编译命令，主要目的是用来将*.java 文件编译成*.class 文件，此命令本身不属于 Windows，所以在使用时需要单独配置，此命令的具体用法将在后面介绍。

1.2.2 编写第 1 个 Java 程序

Java 程序分为两种类型，一种是 Application 程序，另外一种是 Applet 程序，其中有 main 方法的程序主要都是 Application 程序，本书也主要使用 Application 程序进行讲解，Applet 程序主要应用在网页编程上，现在已经基本上不再使用，所以本书不再作任何介绍。

本书还是以输出“Hello World!!”字符串为第 1 个程序，代码如下所示。

范例： Hello.java

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

```
        System.out.println("Hello World!!");
    }
}
```

将上面的程序保存为 Hello.java，然后按照以下步骤进行操作：

（1）在命令行方式下，进入到程序所在的目录，执行 javac Hello.java 命令，对程序进行编译，编译完成之后可以发现在目录之中多了一个 Hello.class 的文件，此文件就是最终要使用的文件。

（2）程序编译之后，输入 java Hello，执行程序，即可得到程序的输出结果。

具体的操作过程可以参考图 1-15 完成。

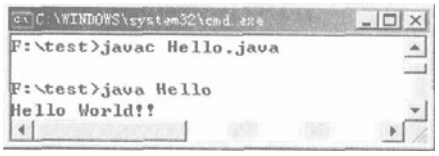


图 1-15　运行 Hello.java 程序

程序输出结果：

```
Hello World!!
```

程序说明：

在所有的 Java Application 中，所有程序都是从 public static void main(String args[])开始运行的，刚接触的读者可能会觉得有些难记，在后面的章节中将会详细讲解 main 方法的各个组成部分。

此时读者如果对上面的程序不明白也没有关系，只要将程序在电脑中输入，然后按照步骤编译、执行即可，在这里只是让读者对 Java Application 程序有一个初步印象，因为以后所有的内容讲解的都将围绕 Java Application 程序进行。

提示：注意程序中的大小写。

读者在编写以上程序时一定要注意字母的大小写问题，因为在 Java 中是严格区分大小写的。另外，为了方便代码的运行，建议在此处按照本书所提供的代码样式进行编写。

1.2.3　classpath 属性的作用

在 Java 中可以使用 set classpath 命令指定 Java 类的执行路径。下面通过一个例子来了解 classpath 的作用，假设这里的 Hello.class 类位于 f:\test 目录下。

在 D 盘的命令行窗口执行下面的指令：

```
set classpath=f:\test
```

然后在 D 盘根目录下执行 java Hello 命令，如图 1-16 所示。

由上面的输出结果可以发现，虽然在 D 盘中并没有 Hello.class 文件，但是也可以用 java Hello 执行 Hello.class 文件，之所以会有这种结果，就是因为在操作中使用了 set classpath 命令将类的查找路径指向了 f:\test 目录，所以在运行时，会从 f:\test 目录查找所需要的类。

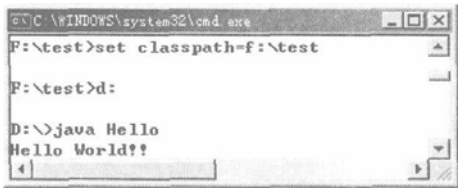


图 1-16　classpath 设置

提示：classpath 与 JVM 的关系。

classpath 主要指的是类的运行路径，实际上在读者执行 Java 命令时，对于本地的操作系统来说就意味着启动了一个 JVM，那么 JVM 在运行时需要通过 classpath 加载所需要的类，而默认情况下 classpath 是指向当前目录（当前命令行窗口所在的目录）的，所以会从此目录下直接查找。

可能有些读者在按照上述的方法操作时发现并不好用，这里要告诉读者的是，在设置 classpath 时，最好将 classpath 指向当前目录，即所有的 class 文件都从当前文件夹中开始查找。

提问：能否通过 classpath 在不同的文件夹中编译 Java 程序？

现在要编译一个*.java 的程序，能不能通过设置 classpath 在不同的目录中进行呢？例如，在 E 盘下建立一个 Hello.java，然后在 C 下编译。

回答：classpath 在编译时不起作用。

classpath 属性只在 Java 运行时起作用，而在 Java 编译时，如果命令行所在的目录不是*.java 所在的目录，则应该输入完整的路径名称，即“javac e:\Hello.java”。

1.3 本章要点

1．Java 实现可移植性靠的是 JVM，JVM 就是一台虚拟的计算机，只要在不同的操作系统上植入不同版本的 JVM，Java 程序就可以在各个平台上移植，做到“一次编写，处处运行”。

2．Java 中程序的执行步骤为：

① 使用 javac 将一个*.java 文件编译成*.class 文件。

② 使用 Java 可以执行一个*.class 文件。

3．每次使用 Java 命令执行一个 class 时，都会启动 JVM，JVM 通过 classpath 给出的路径加载所需要的类文件，可以通过 set classpath 设置类的加载路径。

4．Java 程序主要分为两种，即 Java Application 和 Java Applet 程序，Java Applet 主要是在网页中嵌入的 Java 程序，基本上已经不再使用了，而 Application 是指有 main 方法的程序，本书主要讲解 Application 程序。

1.4 习　　题

1．在屏幕上输出“我喜欢学习 Java”的信息。

2．在屏幕上打印出以下图形：

```
**************************************
*********    Java 程序设计    *********
**************************************
```

第 2 章　简单的 Java 程序

通过本章的学习可以达到以下目标：

- ☑ 可以进一步理解 Java 程序的基本组成。
- ☑ 可以使用注释对程序代码进行说明。
- ☑ 掌握 Java 中标识符的命名规则。
- ☑ 了解 Java 中的关键字。
- ☑ 可以使用 Java 定义变量或声明常量。

第 1 章已经介绍了 Java 的发展历程及主要特点，并带着读者编写了第 1 个程序，大部分刚刚开始学习的读者可能对这些程序并不是十分理解，那么本章将先通过一个简单的程序对之前的内容进行完整的说明，然后将为读者介绍 Java 中的注释、标识符、变量、关键字等与程序开发相关的知识。**本章视频录像讲解时间为 20 分钟，源代码在光盘对应的章节下。**

2.1　一个简单的 Java 范例

下面给出一个简单的 Java 程序范例，有兴趣的读者可以和第 1 章的范例进行对比，观察 Java 程序的基本结构及相同点。

范例：定义一个简单类

```
public class TestJava {
    public static void main(String[] args) {            // Java操作的一个简单
                                                            范例，输出和乘方
        int num = 10;                                    // 定义整型变量
        num = 30;                                        // 修改变量内容
        System.out.println("num的内容是：" + num);        // 输出内容
        System.out.println("num * num = " + num * num);// 输出乘方
    }
}
```

程序运行结果：

```
num的内容是：30
num * num = 900
```

程序说明：

（1）程序的第 2 行使用“//”声明的部分是 Java 的注释部分，注释有助于程序的阅读，它在编译时是不会被编译的，里面可以写任意的内容。

（2）public class 是 Java 中的关键字，表示的是定义一个类，在 Java 中所有的操作都是由类组成的，关于类的概念以后会有详细的解释，TestJava 是程序中类的名称。因为主方法编写在此类之中，所以本书将该类称为主类。

注意：关于使用 public class 和 class 声明类的区别。

在 Java 中声明一个类的方式有两种，即 public class 类名称和 class 类名称。

（1）使用“public class 类名称”声明一个类时，类名称必须与文件名称一致，否则程序将无法编译，如图 2-1 所示。

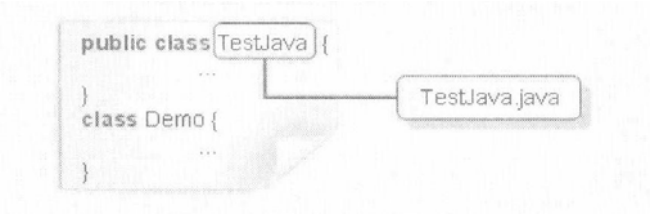


图 2-1　Java 命名

（2）使用“class 类名称”声明一个类时，类名称可以与文件名称不一致，但是在执行时一定要执行生成后的*.class。

范例：有如下程序

```
class Demo {                                          // 定义Demo类
    public static void main(String[] args) {          // 主方法定义在Demo类
        System.out.println("Hello World!!!") ;         // 输出信息
    }
}
```

虽然文件名称是 TestOther.java，但是由于主方法是在 Demo 类中，所以执行时直接执行 java Demo 命令即可，找的是生成后的*.class 文件。

（3）在一个 Java 文件中可以有多个 class 类的定义，但是只能有一个 public class 定义。

（4）读者可以发现，在定义类名称时，开头的首字母为大写，实际上这属于 Java 的命名规范，只要是类的定义，则类名称中每个单词的首字母必须大写。

（3）public static void main(String args[])是程序的主方法，即所有的程序都会以此方法作为起点并运行下来。

（4）“int num=10”意思是定义一个整型变量 num，int 表示变量的类型是整型，变量的内容是可以修改的，所以当程序执行完“num=30”之后，num 的值就是 30。

（5）System.out.println()起输出的作用，是直接将“()”中的内容进行输出，如果有多个值时，中间可以使用“+”连接。

（6）System.out 是指标准输出，通常与计算机的接口设备有关，如打印机、显示器等。其后所续的文字 println 是由 print 与 line 所组成的，意思是将后面括号中的内容打印在标准输出设备显示器上。因此，第 8 行的语句执行完后会换行，也就是把光标移到下一行的开头继续输出。读者可以把 System.out.println()改成 System.out.print()，看一下换行与不换行的区别。

（7）在 Java 中所有的程序都是由一个个代码段组成的，代码段使用“{}”声明，可以

嵌套。

2.2　Java 程序的注释

在任何编程语言之中，都存在注释，注释的主要功能是让其他用户可以方便地阅读每段程序，提高程序的可读性，还可以通过注释屏蔽掉一些暂时不用的语句，等需要时直接取消此语句的注释即可，在 Java 中根据功能的不同，注释主要分为单行注释、多行注释、文档注释 3 种。下面分别进行介绍。

- 单行注释，就是在注释内容前面加双斜线（//），Java 编译器会忽略掉这部分信息。如下面语句：

```
int num ;      // 定义一个整数
```

- 多行注释，就是在注释内容前面以单斜线加一个星形标记（/*）开头，并在注释内容末尾以一个星形标记加单斜线（*/）结束。当注释内容超过一行时一般使用这种方法，如下面语句：

```
/*
    int c = 10 ;
    int x = 5 ;
*/
```

- 文档注释，是以单斜线加两个星形标记（/**）开头，并以一个星形标记加单斜线（*/）结束。用这种方法注释的内容会被解释成程序的正式文档，并能包含在如 javadoc 之类工具生成的文档中，用以说明该程序的层次结构及方法。

提示：可以使用注释帮助用户调试代码。

用户在进行代码开发时，如果发现某段代码可能暂时不需要，最好使用注释的方式将其注释起来，这样就不会进行编译了。

2.3　Java 中的标识符

Java 中的包、类、方法、参数和变量的名字可由任意顺序的大小写字母、数字、下划线（_）和美元符号（$）组成，但标识符不能以数字开头，也不能是 Java 中的保留关键字。

如 yourname、yourname_lxh、li_yourname、$yourname 是合法的标识符，class、67.9、Hello LiXingHua 是非法的标识符。

提示：标识符编写的简单建议。

一些刚接触编程语言的读者可能会觉得记住上面的规则很麻烦，在这里提醒读者，标识符最好用字母开头，而且尽量不要包含其他符号。

2.4 Java 中的关键字

和其他语言一样，Java 中也有许多关键字（也叫保留字），如 public、static 等，这些关键字不能当作标识符使用。表 2-1 列出了 Java 中的关键字，这些关键字并不需要读者去强记，因为在程序开发中一旦使用了这些关键字做标识符时，编辑器会自动提示错误。

表 2-1 Java 中的关键字

abstract	assert	boolean	break	byte	case	catch
char	class	continue	const	default	do	double
else	extends	enum	final	finally	float	for
goto	if	implements	import	instanceof	int	interface
long	native	new	package	private	protected	public
return	short	static	synchronized	super	strictfp	this
throw	throws	transient	try	void	volatile	while

对于以上的关键字，读者要特别注意的有如下 3 点：

（1）虽然 goto、const 在 Java 中并没有任何意义，却也是保留字，它们与其他的关键字一样，在程序中不能用来作为自定义的标识符。

（2）true、false、null 虽然不是关键字，但是却作为一个单独标识类型，也不能直接使用。

（3）assert 和 enum 关键字是 Java 新增的内容，assert 是在 JDK 1.4 中增加进来的，而 enum 是在 JDK 1.5 后增加进来的，本书对这两个关键字都有详细的介绍。

提示：不要去强记 Java 中的关键字。

对于刚学习语言的读者可能会觉得如果要全部记住以上的关键字是一件比较麻烦的事，这里要告诉读者，以上的内容随着知识的熟练度会慢慢记住的，不用强记，回顾一下之前的内容，会发现已经见过了 public、class、void、static、int 等关键字，所以对于一门编程语言的学习应多加练习才是最好的掌握方法。

2.5 变量和常量

变量是利用声明的方式将内存中的某个内存块保留下来以供程序使用。可以声明的数据类型为整型、字符型、浮点型或是其他数据类型，作为变量的保存之用。变量在程序语言中“扮演”了最基本的“角色”。变量可以用来存放数据，而使用变量之前则必须先声明它的数据类型。

举例来说，想在程序中声明一个可以存放整数的变量，这个变量的名称为 num。

在程序中可以写出如下语句：

```
int num;                                    //声明num为整数变量
```

int 为 Java 的关键字，代表整数（Integer）的声明。若要同时声明多个整型的变量，可以像上面的语句一样分别声明它们，也可以把它们都写在同一个语句中，每个变量之间以逗号分开，如下面的写法：

```
int num,num1,num2 ;                    // 同时声明num,num1,num2为整数变量
```

除了整数类型之外，Java 还提供了多种其他的数据类型。Java 的变量类型可以是整型（int）、长整型（long）、短整型（short）、浮点型（float）、双精度浮点型（double）、字符型（char）和布尔型（boolean）。关于这些数据类型，本书在第 3 章中将有详细的介绍。

常量就是一个固定的数值，是不可改变的，例如，数字 1、2 就是整型的常量。

2.6　本 章 要 点

1．Java 程序的开始点是由主方法开始的。

2．Java 语言的注释方式有下面 3 种。

（1）单行注释：//。

（2）多行注释：/*…*/。

（3）文档注释：/** …*/。

3．Java 中的变量名称可以由英文字母、数字、下划线（_）和美元符号（$）组成，但标识符不能以数字开头，也不能是 Java 中的保留关键字。此外，Java 的变量有大小写之分。

4．Java 的基本组成是类，使用 public class 和 class 都可以声明一个类，但是前者类名称必须与文件名称一致，后者文件名称可以与类名称不一致，但是执行时必须执行生成后的*.class 文件。

5．goto 和 const 是未使用到的两个关键字。

6．assert 和 enum 是 JDK 新版本中增加的关键字。

7．System.out.print()是在标准输出设备——显示器上进行输出操作，后面可以使用 println 和 print 两种方法输出，前者是在输出之后加入换行，后者没有换行。

2.7　习　　题

请将下面的程序补充完整：

```
public class Test           {
    public static void main(String args[]){
        System.out.println();      ;   // 打印3*3
    }
}
```

第 3 章　Java 基础程序设计

通过本章的学习可以达到以下目标：

- ☑ 掌握 Java 中的数据类型划分方法。
- ☑ 掌握 8 种基本数据类型的使用方法。
- ☑ 掌握数据类型的转换方式。
- ☑ 掌握 Java 中的位运算。
- ☑ 掌握各个运算符、表达式的使用。
- ☑ 掌握判断、循环语句的使用方法，并可以编写简单的 Java 程序。
- ☑ 掌握 break 及 continue 关键字的作用。

本章将介绍 Java 语言的基本知识点，还将介绍 Java 中的各种数据类型及类型间的相互转换、Java 中各种运算操作等，同时也将向读者介绍 Java 中的各种循环、控制语句等基本程序语法。**本章视频录像讲解时间为 2 小时 09 分钟，源代码在光盘对应的章节下。**

3.1　数据类型划分

数据类型在程序语言的构成要素中占有相当重要的地位。Java 的数据类型可分为基本数据类型与引用数据类型。

原始数据类型也称为基本数据类型，它们包括了最基本的 boolean、byte、char、short、int、long、float 与 double 等类型。另一种数据类型为引用数据类型，它是以一种特殊的方式指向变量的实体，这种机制类似于 C/C++的指针。这类变量在声明时不会分配内存，必须另外进行开辟内存空间的操作，如字符串与数组均属于这种数据类型。

在 Java 中规定了 8 种基本数据类型变量来存储整数、浮点数、字符和布尔值，如图 3-1 所示。

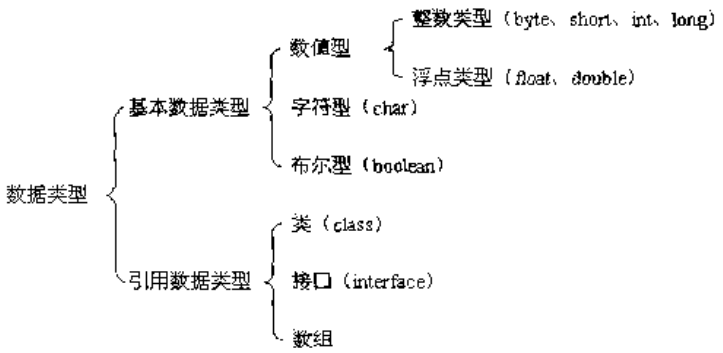


图 3-1　Java 的变量类型

考虑到学习层次问题，在这里只是先介绍基本数据类型，引用数据类型会在后面的章节中介绍。

3.2　基本数据类型

到目前为止，相信读者已经对 Java 有了一些初步的认识，如果想在程序中使用一个变量，就必须先声明，此时编译程序会在未使用的内存空间中寻找一块足够能保存这个变量的空间以供这个变量使用。Java 的基本数据类型如表 3-1 所示。

表 3-1　Java 基本数据类型

序　　号	数 据 类 型	大小/位	可表示的数据范围
1	long（长整数）	64	-9223372036854775808～9223372036854775807
2	int（整数）	32	-2147483648～2147483647
3	short（短整数）	16	-32768～32767
4	byte（位）	8	-128～127
5	char（字符）	2	0～255
6	float（单精度）	32	-3.4E38（-3.4×10^{38}）～3.4E38（3.4×10^{38}）
7	double（双精度）	64	-1.7E308（-1.7×10^{308}）～1.7E308（1.7×10^{308}）

3.2.1　整数类型

当数据不带有小数或分数时，可以声明为整数变量，如 3、-147 等即为整数。在 Java 中，整数数据类型可以分为 long、int、short 及 byte 4 种，long 为 64 位，也就是 8 个字节（bytes），可表示范围为-9223372036854775808～9223372036854775807；int 为 32 位，也就是 4 个字节，表示范围为-2147483648～2147483647；若数据值的范围为-32768～32767 时，可以声明为 short（短整数）类型；若数据值更小，在-128～127 之间时，可以声明为 byte 类型，以节省内存空间。例如，声明一个短整型变量 sum 时，可以在程序中作出如下声明：

```
short sum ;                                    // 声明sum为短整型
```

经过声明之后，Java 即会在可使用的内存空间中寻找一个占有两个字节的块供 sum 变量使用，同时，这个变量的范围只能在-32768～32767 之间。

在 Java 中对于一个整型常量，其类型默认的就是 int 型，所以读者在声明常量时不要超过 int 数据类型的范围。

范例：以下代码验证了如果数据过长则有可能出现的问题

```
class DataDemo01 {
    public static void main(String[] args) {
        int num = 99999999999999999999 ;// 定义整型变量，错误，超出长度范围
    }
}
```

以上代码为 int 变量初始化时值超过了其本身的范围的情况，所以在编译程序时出现了以下的错误提示：

```
DataDemo01.java:3: integer number too large: 99999999999999999999
            int num = 99999999999999999999 ;          // 定义整型变量，错
                                                         误，超出长度范围

1 error
```

3.2.2 数据的溢出

当整数的数据大小超出了可以表示的范围，而程序中又没有做数值范围的检查时，这个整型变量所输出的值将发生紊乱，且不是预期的运行结果。在下面的程序范例中声明了一个整型的数，并把它赋值为整型所可以表示范围的最大值，然后将它分别加 1 及加 2。

范例：将整型的最大值加 1 和加 2

```
class DataDemo02 {
    public static void main(String[] args) {
        int max = Integer.MAX_VALUE;                    // 得到整型的最大值
        System.out.println("整型的最大值: " + max);        // 输出最大值
        System.out.println("整型最大值 + 1:" + (max + 1));// 最大值加1
        System.out.println("整型最大值 + 2:" + (max + 2));// 最大值加2
    }
}
```

程序运行结果：

```
整型的最大值：2147483647
整型最大值 + 1:-2147483648
整型最大值 + 2:-2147483647
```

当最大值加上 1 时，结果变成表示范围中最小的值；当最大值加上 2 时，结果变成表示范围中次小的值，这就是数据类型的溢出。读者可以发现，上面例子会出现一个循环，若是想避免这种情况的发生，在程序中就必须加上数值范围的检查功能，或者使用较大的表示范围的数据类型，如长整型。

当声明了一整数 x，其表示的范围为-2147483648～2147483647 之间，当 x 的值设为最大值 2147483647，仍在整数的范围内，但是当 x 加 1 或加 2 时，整数 x 的值反而变成-2147483648 和-2147483647，成为可表示范围的最小及次小值。

上面的情形就像计数器的内容到最大值时会自动归零一样。而在整数中最小值为-2147483648，所以当整数 x 的值最大时，加上 1 就会变成最小值-2147483648，也就是产生了溢出。可以参考图 3-2 来了解数据类型的溢出问题。

为了避免 int 类型的溢出，可以在该表达式中的任一常量后加上大写的 L，或在变量前面加上 long，做强制类型的转换。在下面的程序中加上防止溢出的处理，为了让读者方便比较，特意保留一个整数的溢出语句。

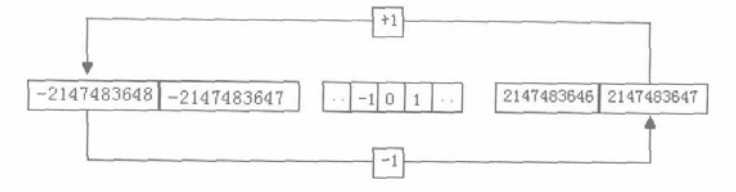


图 3-2　数据类型的溢出

范例：使用强制类型转换，防止数据溢出

```
public class DataDemo03 {
  public static void main(String[] args) {
    int max = Integer.MAX_VALUE;                              // 得到整型的最大值
    System.out.println("整型的最大值：" + max);                 // 输出最大值
    System.out.println("整型最大值 + 1:" + (max + 1));          // 最大值加1
    System.out.println("整型最大值 + 2:" + (max + 2L));         // 最大值加2，变为
                                                                 long型
    System.out.println("整型最大值 + 2:" + ((long)max + 2)); // 强制转换为long型
  }
}
```

程序运行结果：

```
整型的最大值：2147483647
整型最大值+1：-2147483648
整型最大值+2：2147483649
整型最大值+2：2147483649
```

由上面的程序可知，处理 int 类型的溢出，可以利用强制类型转换方式。但是对于 long 类型的溢出，就没有处理办法了，此时就需要在程序中加上变量值的界限检查，在运行时才不会发生错误。

提问：Integer 是什么？

Integer 的作用是什么？在有的书中也经常可以看到这样的名称？既然可以求出整型的最大值，那能否求出整型的最小值呢？除了这些之外，Integer 还有什么其他的作用？

回答：Integer 是以后要讲解的包装类。

Integer 在 Java 中属于包装类，对于包装类的概念，本书将在 Java 常用类库章节中会有所介绍，有兴趣的读者也可以自己先提前进行翻看，对于 Integer 可以使用 MAX_VALUE 取得最大值，也可以通过 MIN_VALUE 取得其最小值，Integer 的最大作用在于字符串与整型的转换上，还具有自动拆箱和装箱的功能，这点在以后都会有详细介绍，但在这里提醒读者的是，MAX_VALUE 必须大写。

3.2.3　字符类型

字符类型在内存中占有两个字节，可以用来保存英文字母等字符。计算机处理字符类

型时，是把这些字符当成不同的整数来看待，因此，严格来说，字符类型也算是整数类型的一种。

在计算机的世界里，所有的文字、数值都只是一连串的 0 与 1，这些 0 与 1 对于设计者来说实在是难以理解，于是就产生了各种方式的编码，它们通过指定一个数值来代表某个字符，如常用的字符码系统 ASCII。

虽然各类的编码系统合起来有数百种之多，却没有一种是包含足够的字符、标点符号及常用的专业技术符号。这些编码系统之间可能还会有相互冲突的情形发生，也就是说，不同的编码系统可能会使用相同的数值来表示不同的字符，在数据跨平台时就会发生错误。

Unicode 就是为了避免上述情况的发生而产生的，它为每个字符制订了一个唯一的数值，因此，在任何的语言、平台、程序中都可以安心地使用。Java 所使用的就是 Unicode 字符码系统。

例如，Unicode 中的小写 a 是以 97 来表示的，在下面的程序中可以看到，声明字符类型的变量 ch1、ch2，分别将变量 ch1 的值设为 97，ch2 的值设为字符 a，再输出字符变量 ch1 及 ch2 的内容。

范例： 测试字符和整型之间的相互转换

```
public class DataDemo04 {
    public static void main(String[] args) {
        char ch1 = 'a';                          // 定义字符
        char ch2 = 97;                           // 定义字符，整型转字符
        System.out.println("ch1 = " + ch1);      // 打印输出
        System.out.println("ch2 = " + ch2);      // 打印输出
    }
}
```

程序运行结果：

```
ch1 = a
ch2 = a
```

给字符变量赋值可以使用数值和字符，它们都可以使程序正确地运行。要注意的是，字符要用一对单引号（''）括起。

例如，想在程序中输出一个包括双引号的字符串时，可把字符变量赋值为转义字符，再将它输出，也就是说，在程序中声明一个字符类型变量 ch，然后把 ch 设置为“\""，再进行输出的操作，或者，也可以直接在要输出的字符串中加入特殊的转义字符。如表 3-2 所示为常用的转义字符。

表 3-2　常用的转义字符

序　号	转义字符	描　述	序　号	转义字符	描　述
1	\f	换页	5	\r	归位
2	\\	反斜线	6	\"	双引号
3	\b	倒退一格	7	\t	制表符 Tab
4	\'	单引号	8	\n	换行

以下面的程序为例，将 ch1 赋值为“\””、ch2 赋值为“\\”，并将字符变量 ch 输出在显示器上，同时，在打印的字符串中直接加入转义字符。

范例：转义字符的应用

```
public class DataDemo05 {
    public static void main(String[] args) {
        char ch1 = '\"';                                    // 定义转义字符
        char ch2 = '\\';                                    // 定义转义字符
        System.out.println("ch1 = " + ch1);                 // 打印输出
        System.out.println("ch2 = " + ch2);                 // 打印输出
        System.out.println("\"Hello World\"") ;              // 直接输出转义字符
    }
}
```

程序运行结果：

```
ch1 = "
ch2 = \
"Hello World"
```

3.2.4　浮点数类型与双精度浮点数类型

在日常生活中经常会使用到小数类型的数值，如身高、体重等，整数将不能满足程序设计者的要求。在数学中，这些带有小数点的数值称为实数，在 Java 中，这种数据类型称为浮点数类型（float），其长度为 32 个字节，有效范围为$-3.4E^{1038}$～$3.4E^{1038}$。当浮点数的表示范围不够大时，还有一种双精度（double）浮点数可供使用。双精度浮点数类型的长度为 64 个字节，有效范围为$-1.7E^{10308}$～$1.7E^{10308}$。

浮点数的表示方式除了指数的形式外，还可用带有小数点的一般形式来表示。例如，想声明一个 double 类型的变量 num 与一个 float 类型的变量 sum，并同时给 sum 赋初值 3.0，可以在程序中作出如下声明及设置：

```
double num ;                        // 声明num为双精度浮点型变量
float sum = 3.0f ;                  // 声明sum为浮点型变量，其初值为3.0
```

声明之后，Java 即会在可使用的内存空间中寻找一个占有 8 个字节的块供 num 变量使用，其范围在$-1.7E^{10308}$～$1.7E^{10308}$之间，寻找另一个占有 4 个字节的块供 sum 变量使用，而这个变量的范围只能在$-3.4E^{1038}$～$3.4E^{1038}$之间。在此例中，sum 的初值为 3.0。

下面为声明与设置 float 与 double 类型的例子：

```
double num1 = -6.3e64 ;             // 声明num1为double，其值为-6.3*10^64
double num2 = -5.34E16 ;            // e也可以用大写的E来取代
float num3 = 7.32f ;                // 声明num3为float，并设初值为7.32f
float num4 = 2.456E67 ;             // 错误，因为2.456*10^67已超过float可表示的范围
```

需要注意的是，使用浮点型数值时，默认的类型是 double，在数值后面可加上 D 或是 d，作为 double 类型的标识。在 Java 中，D 或 d 是可有可无的。在数据后面加上 F 或 f，则

作为 float 类型的识别。若没有加上，Java 就会将该数据视为 double 类型，而在编译时就会发生错误，错误提示会告诉设计者可能会失去精确度。

在下面的程序中，将声明一个 float 类型的变量 num，并赋值为 3.0，将 num*num 的运算结构输出到显示器上。

范例：浮点型数据计算

```
public class DataDemo06 {
    public static void main(String[] args) {
        float num = 3.0f;                                    // 定义float型变量
        System.out.println("两个小数相乘：" + num * num);// 计算两数相乘
    }
}
```

程序运行结果：

```
两个小数相乘：9.0
```

3.2.5 布尔类型

布尔（boolean）类型的变量只有 true（真）和 false（假）两种。也就是说，当将一个变量定义成布尔类型时，它的值只能是 true 或 false，除此之外，没有其他的值可以赋值给这个变量。例如，声明名称为 flag 变量的布尔类型，并设置为 true 值，可以使用下面的语句：

```
boolean flag = true ;        // 声明布尔变量flag，并赋值为true
```

经过声明之后，布尔变量的初值即为 true，当然如果在程序中需要更改 status 的值时，可以随时更改。将上述的内容写成了程序 DateDemo07，读者可以先熟悉一下布尔变量的使用。

范例：布尔类型的使用

```
public class DataDemo07 {
    public static void main(String[] args) {
        boolean flag = true;                          // 定义布尔型变量
        System.out.println("flag = " + flag);        // 打印输出
    }
}
```

程序运行结果：

```
flag = true
```

布尔值通常用来控制程序的流程，读者可能会觉得有些抽象，本书会在后面的章节中介绍布尔值在程序流程中所起的作用。

3.2.6 基本数据类型的默认值

在 Java 中，若在变量的声明时没有给变量赋初值，则会给该变量赋默认值，表 3-3 列

出了各种类型的默认值。

表 3-3　基本数据类型的默认值

序　号	数 据 类 型	默　认　值
1	byte	（byte）0
2	short	（short）0
3	int	0
4	long	0L
5	float	0.0f
6	double	0.0d
7	char	\u0000（空，''）
8	boolean	false

在某些情形下，Java 会给予这些没有赋初始值的变量一个确切的默认值，这没有任何意义，是没有必要的，但应该保证程序执行时不会有这种未定义值的变量存在。虽然这种方式给程序编写者带来了很多便利，但是过于依赖系统给变量赋初值，就不容易检测到是否已经给予变量应有的值了，这是需要注意的问题。

3.3　数据类型的转换

Java 的数据类型在定义时就已经确定了，因此不能随意转换成其他的数据类型，但 Java 允许用户有限度地做类型转换处理。数据类型的转换方式可分为“自动类型转换”及“强制类型转换”两种。

3.3.1　数据类型的自动转换

在程序中已经定义好了数据类型的变量，若要用另一种数据类型表示时，Java 会在下列的条件皆成立时，自动做数据类型的转换：

（1）转换前的数据类型与转换后的类型兼容。

（2）转换后的数据类型的表示范围比转换前的类型大。

例如，将 short 类型的变量 a 转换为 int 类型，由于 short 与 int 皆为整数类型，符合上述条件（1）；而 int 的表示范围比 short 大，符合条件（2）。因此 Java 会自动将原为 short 类型的变量 a 转换为 int 类型。

值得注意的是，类型的转换只限该行语句，并不会影响原先所定义的变量的类型，而且通过自动类型的转换可以保证数据的精确度，它不会因为转换而损失数据内容。这种类型的转换方式也称为扩大转换。

前面曾经提到过，若是整数的类型为 short 或 byte，为了避免溢出，Java 会将表达式中的 short 和 byte 类型自动转换成 int 类型，即可保证其运算结果的正确性，这也是 Java 所提供的“扩大转换”功能。

以“扩大转换”来看可能比较容易理解字符与整数是可使用自动类型转换的，整数与浮点数亦是兼容的，但是由于 boolean 类型只能存放 true 或 flase，与整数及字符是不兼容的，因此不可能做类型的转换。下面介绍当两个数中有一个为浮点数时，其运算的结果会有什么样的变化。

范例：数据类型的转换

```
public class DataDemo08 {
    public static void main(String[] args) {
        int x = 30;                                          // 定义整型变量
        float y = 22.19f;                                    // 定义浮点型变量
        System.out.println("x/y = " + (x/y));                // 除法操作
        System.out.println("10/3.5 = " + (10/3.5));          // 直接使用常量进行除法
        System.out.println("10/3 = " + (10/3));              // 直接使用常量进行除法
    }
}
```

程序运行结果：

```
x/y = 1.3519603
10/3.5 = 2.857142857142857
10/3 = 3
```

从程序的输出结果可以发现，int 类型与 float 类型进行计算之后，输出的结果会变成 float 类型，一个整型常量和一个浮点型常量进行计算之后，结果也会变为一个浮点数据，而如果是两个 int 类型的常量进行计算，最终结果还是 int 类型，而其小数部分将会被忽略。

也就是说，假设有一个整数和双精度浮点数作运算时，Java 会把整数转换成双精度浮点数后再作运算，运算结果也会变成双精度浮点数。关于表达式的数据类型转换，在后面的章节中会有更详细的介绍。

提示：任何类型的数据都向 String 转型。

一些读者从其他的 Java 教材之中经常会发现，有一种表示字符串的数据类型 String，从其定义上可以发现单词首字母大写了，所以此为一个类，属于引用数据类型，但是此类属于系统的类，而且使用上有一些注意事项，对于此种类型后面会有更详细的介绍，在此处读者所需要知道的只有以下两点：

（1）String 可以像普通变量那样直接通过赋值的方式进行声明。字符串是使用“""”括起来的。两个字符串之间可以使用“+”进行连接。

（2）任何数据类型碰到 String 类型的变量或常量之后都向 String 类型转换。

范例：定义字符串变量

```
public class DataDemo09 {
    public static void main(String[] args) {
        String str = "lixinghua ";                           // 定义字符串变量
        int x = 30;                                          // 定义整型变量
        str = str + x;                                       // 改变字符串变量内容
```

```
        System.out.println("str = " + str);          // 打印输出
    }
}
```

程序运行结果：

```
str = lixinghua 30
```

范例：字符串常量操作的问题

```
public class DataDemo10 {
    public static void main(String[] args) {
        int i = 1;                                   // 定义整型变量
        int j = 2;                                   // 定义整型变量
        System.out.println("1 + 2 = " + i + j);      // 加法计算
    }
}
```

程序运行结果：

```
1 + 2 = 12
```

从以上的输出结果中可以发现，程序的主要目的是要计算 i+j 的值，但是由于碰到了字符串常量，所以所有的数据类型都会变为字符串常量，也就是说此时的“+”实际上表示的是字符串连接的含义，对于以上的程序，如果要得到正确的结果，则必须修改为：

```
System.out.println("1 + 1 = " + (i + j));
```

加上括号之后，就表示输出时先计算两个数字相加的结果。

3.3.2　数据类型的强制转换

当两个整数进行运算时，其运算的结果也会是整数。例如，当做整数除法 8/3 的运算时，其结果为整数 2，并不是实际的 2.33333…，因此，在 Java 中若是想要得到计算的结果是浮点数时，就必须将数据类型做强制性的转换，转换的语法如下：

【格式 3-1　数据类型的强制性转换语法】

```
（要转换的数据类型）变量名称;
```

因为这种强制类型的转换是直接编写在程序代码中的，所以也称为显式转换。以下程序说明了在 Java 中整数与浮点数是如何转换的。

范例：数据类型的强制转换

```
public class DataDemo11 {
    public static void main(String[] args) {
        float f = 30.3f;                             // 定义浮点型变量
        int x = (int) f;                             // 强制转换为int型
        System.out.println("x = " + x);              // 输出转型之后的值
```

```
        System.out.println("10 / 3 = " + ((float) 10 /  3));// 常量计算使用
                                                          强制类型转换
    }
}
```

程序运行结果：

```
x = 30
10 / 3 = 3.3333333
```

在程序中，首先将一个浮点型的变量 f 的内容赋给了 int 型变量 x，因为 int 数据类型的长度小于 float 类型的长度，所以此处需要进行强制转换。程序的最后使用了两个整型常量进行计算，但是因为将其中的一个整型变量变为了 float 类型，所以在计算时会强制把其他的整型也变为 float 类型，所以计算的结果是包含小数的。

只要在变量前面加上要转换的数据类型，运行时就会自动将此行语句中的变量做类型转换的处理，但这并不影响原来所定义的数据类型。

此外，若是将一个超出该变量可表示范围的值赋值给这个变量时，这种转换称为缩小转换。由于在转换的过程中可能会丢失数据的精确度，Java 并不会自动做这些类型的转换，此时就必须要做强制性的转换。

3.4 运算符、表达式与语句

程序是由许多语句组成的，而语句的基本单位是表达式与运算符。像之前讲解过的“+”和“/”实际上都是运算符。下面将为读者介绍 Java 运算符的用法、表达式与运算符之间的关系以及表达式中各种变量的数据类型的转换等。

3.4.1 运算符

Java 中的语句有很多种形式，表达式就是其中一种形式。表达式是由操作数与运算符所组成的，操作数可以是常量、变量，也可以是方法，而运算符就是数学中的运算符号，如“+”、“-”、“*”、“/”、“%”等。在表达式（z+100）中，z 与 100 都是操作数，而“+”就是运算符，如图 3-3 所示。

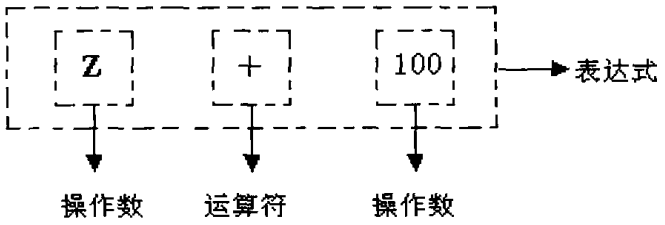


图 3-3 表达式是由操作数与运算符组成

Java 提供了许多的运算符，这些运算符除了可以处理一般的数学运算外，还可以做逻辑运算、地址运算等。根据其所使用的类的不同，运算符可分为赋值运算符、算术运算符、

关系运算符、逻辑运算符、条件运算符和括号运算符等。

1. 赋值运算符号

想为各种不同数据类型的变量赋值时，就必须使用赋值运算符（=），表 3-4 中所列出的赋值运算符虽然只有一个，但它却是 Java 语言中必不可缺的。

表 3-4　赋值运算符

赋值运算符号	描　述
=	赋值

等号（=）在 Java 中并不是“等于”的意思，而是“赋值”的意思。表达式的赋值范例如图 3-4 所示。

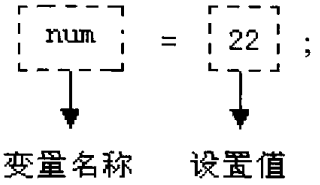


图 3-4　表达式的赋值范例

上面的语句是将整数 22 赋值给 num 这个变量。再来看看下面这个语句：

```
num = num - 3          // 将num-3的值赋给变量num存放
```

从未学习过任何编程语言的读者，可能会不习惯这种思考方式。等号（=）是“赋值”的语句，把 num-3 的值赋给 num 存放，因为之前已经把 num 的值设为 22，所以执行这个语句时，Java 会先处理等号后面的部分 num-3（值为 19），再赋值给等号前面的变量 num，执行后，存放在变量 num 的值就变成了 19。将上面的语句编写成下面的程序。

范例：验证以上概念

```
public class OperatorDemo01 {
    public static void main(String[] args) {
        int num = 22 ;                                  // 定义整型变量
        System.out.println("第一次输出：num = " + num) ;
        num = num - 3 ;                                 // 修改变量内容
        System.out.println("第二次输出：num = " + num) ;
    }
}
```

程序运行结果：

```
第一次输出：num = 22
第二次输出：num = 19
```

当然，在程序中也可以将等号后面的值赋值给其他的变量，如：

```
int sum = num1+num2 ;        // num1与num2相加之后的值再赋给变量sum存放
```

num1 与 num2 的值经过运算后仍然保持不变，sum 会因为“赋值”的操作而更改内容。

2. 一元运算符

对于大部分的表达式而言，运算符的前后都会有操作数。但是有一种运算符较特别，它只需要一个操作数，称为一元运算符。下面的语句就是由一元运算符与一个操作数组成的：

```
+3 ;                // 表示正3
b = -a ;            // 表示负a的值赋给变量b存放
!a ;                // a的NOT运算，若a为0，则!a为1，若a不为0，则!a为0
```

表 3-5 列出了一元运算符的成员。

表 3-5 一元运算符

序　号	一元运算符	描　述
1	+	正号
2	–	负号
3	!	NOT，否

下面的程序演示了上述 3 个一元运算符的使用。

范例：一元运算符的使用

```
public class OperatorDemo02 {
    public static void main(String[] args) {
        boolean b = false;                                          // 定义boolean
                                                                       型数据
        int x = 10 ;                                                // 定义一个正数
        int y = -30 ;                                               // 定义一个负数
        System.out.println("b = " + b + " , !b = " + !b);          // 取反
        System.out.println("x = " + x + " , -x = " + -x);          // 使用负号
        System.out.println("y = " + y + " , +y = " + +y);          // 使用正号
    }
}
```

程序运行结果：

```
b = false , !b = true
x = 10 , -x = -10
y = -30 , +y = -30
```

3. 算术运算符

算术运算符在数学中经常会使用到，表 3-6 列出了其操作符号。

表 3-6 算术运算符

序　号	算术运算符	描　述
1	+	加法
2	–	减法
3	*	乘法

续表

序　　号	算术运算符	描　　述
4	/	除法
5	%	取模（取余数）

范例：算术运算符的使用

```
public class OperatorDemo03 {
    public static void main(String[] args) {
        int i = 10;                                                    // 定义整型变量
        int j = 3;                                                     // 定义浮点型变量
        System.out.println(i + " + " + j + " = " + (i + j));           // 加法操作
        System.out.println(i + " - " + j + " = " + (i - j));           // 减法操作
        System.out.println(i + " * " + j + " = " + (i * j));           // 乘法操作
        System.out.println(i + " / " + j + " = " + (i / j));           // 除法操作
        System.out.println(i + " % " + j + " = " + (i % j));           // 取模操作
    }
}
```

程序运行结果：

```
10 + 3 = 13
10 - 3 = 7
10 * 3 = 30
10 / 3 = 3
10 % 3 = 1
```

4. 关系运算符

设计者常会在 if 语句中使用到关系运算符，if 语句的格式如下：

【格式 3-2　if 语句的格式】

```
if (布尔表达式){
    语句 ;
}
```

如果括号中的布尔表达式成立，就会执行{}中的语句；若是布尔表达式不成立，则后面的语句就不会被执行，如下面的程序片段：

```
if (x>10)
    System.out.println("Welcome To MLDN") ;
```

当 x 的值大于 0，就是判断条件成立时，会执行输出字符串“Welcome To MLDN”的操作；相反，当 x 的值为 0 或是小于 0 时，if 语句的判断条件不成立，就不会执行上述操作。表 3-7 列出了关系运算符，这些运算符在数学上也是经常使用的。

表 3-7 关系运算符

序　号	关系运算符	描　述
1	>	大于
2	<	小于
3	>=	大于等于
4	<=	小于等于
5	==	等于
6	!=	不等于

在 Java 中，关系运算符的表示方式和在数学中很类似，但是由于赋值运算符为“=”，为了避免混淆，当使用关系运算符“等于”（==）时，就必须用两个等号表示；而关系运算符“不等于”的形式有些特别，用“!=”代表，这是因为在键盘上想要取得数学上的不等于符号“≠”较为困难，所以就用“!=”表示不等于符号。

范例：关系运算符的使用

```
public class OperatorDemo04 {
    public static void main(String args[]) {
        System.out.println("3 > 1 = " + (3 > 1));       // 使用大于号
        System.out.println("3 < 1 = " + (3 < 1));       // 使用小于号
        System.out.println("3 >= 1 = " + (3 >= 1));     // 使用大于等于号
        System.out.println("3 <= 1 = " + (3 <= 1));     // 使用小于等于号
        System.out.println("3 == 1 = " + (3 == 1));     // 使用等于号
        System.out.println("3 != 1 = " + (3 == 1));     // 使用不等于号
    }
}
```

程序运行结果：

```
3 > 1 = true
3 < 1 = false
3 >= 1 = true
3 <= 1 = false
3 == 1 = false
3 != 1 = false
```

从以上程序可以发现，当使用关系运算符去判断一个表达式的成立与否时，若是判断式成立，会产生一个布尔值 true，若是判断式不成立，则会产生布尔值 false。下面的程序用来判断 if 语句括号中的条件是否成立，若成立则执行 if 后面的语句。

范例：在 if 语句中使用关系运算符

```
public class OperatorDemo05 {
    public static void main(String[] args) {
        if (5 > 2) {                                    // 判断5是否大于2
            System.out.println("条件成立：5大于2");
        }
```

```
        if (true)                                    // 判断是否为真
            System.out.println("直接写的true");
        if ((3 + 6) == (3 - 6)) {                    // 判断计算结果内容是否相等
            System.out.println("这是不可能成立的！");
        }
    }
}
```

程序运行结果：

```
条件成立：5大于2
直接写的true
```

从上面的程序可以发现，如果在 if 语句之中条件满足了，则会执行 if 语句之中的内容，另外，如果一个 if 语句之中只有一条语句，那么就可以不写“{}”，如下面的程序，但是为了程序便于调试，本书中并不建议读者采用这种简写的方式。

```
if (true)
        System.out.println("直接写的true");
```

5. 自增与自减运算符

自增与自减运算符也是各个语言中最常见到的语句，在 Java 中仍然将它们保留了下来，是因为它们具有相当大的便利性。表 3-8 列出了自增与自减运算符。

表 3-8　自增与自减运算符

序　　号	自增与自减运算符	描　　述
1	++	自增，变量值加 1
2	--	自减，变量值减 1

用自增与自减运算符可使程序更加简洁。例如，声明一个 int 类型的变量 a，在程序运行中想让它加 1，语句如下：

```
a = a+1 ;          // a加1后再赋值给a存放
```

将 a 的值加 1 后再赋值给 a 存放。也可以利用自增运算符“++”写出更简洁的语句，而语句的意义是相同的：

```
a++ ;          // a加1后再赋值给a存放，a++为简洁写法
```

在程序中还可以看到另外一种自增运算符“++”的用法，就是将自增运算符“++”写在变量的前面，如++a，这和 a++所代表的意义是不一样的。a++会先执行整个语句后再将 a 的值加 1，而++b 则先把 b 的值加 1 后，再执行整个语句。下面的程序为将 a 与 b 的值皆设为 3，将 a++及++b 输出来，可以轻易地比较出两者的不同。

范例：验证自增和自减

```
public class OperatorDemo06 {
    public static void main(String[] args) {
        int a = 3, b = 3 ;                                  // 定义整型变量a和b
```

```
        int x = 6, y = 6 ;                                    // 定义整型变量x和y
        System.out.println("a = " + a);                       // 输出变量a的值
        System.out.println("\t a++ = " + (a++) + " , a= " + a);// 先执行后
                                                                自增
        System.out.println("b = " + b);                       // 输出变量b的值
        System.out.println("\t ++b = " + (++b) + " , b= " + b);// 先自增后
                                                                执行
        System.out.println("x = " + x);                       // 输出变量x的值
        System.out.println("\t x-- = " + (x--) + " , x= " + x);// 先执行后
                                                                自减
        System.out.println("y = " + y);                       // 输出变量y的值
        System.out.println("\t --y = " + (--y) + " , y= " + y);// 先自减后
                                                                执行
    }
}
```

程序运行结果：

```
a = 3
     a++ = 3 , a= 4
b = 3
     ++b = 4 , b= 4
x = 6
     x-- = 6 , x= 5
y = 6
     --y = 5 , y= 5
```

6. 逻辑运算符

使用逻辑运算符可以连接多个逻辑运算。表 3-9 列出了常用的逻辑运算符。

表 3-9 逻辑运算符

序　号	逻辑运算符	描　述
1	&	AND，与
2	&&	短路与
3	|	OR，或
4	||	短路或

当使用逻辑运算符“&&”时，运算符前后的两个操作数的返回值皆为真，运算的结果才会为真；使用逻辑运算符“||”时，运算符前后的两个操作数的返回值只要有一个为真，运算的结果就会为真。如下面的语句：

```
a>0 && b> 0              // 两个操作数皆为真，运算结果才为真
a>0 || b>0               // 两个操作数只要一个为真，运算结果就为真
```

在 a>0 而且 b>0 时，表达式的返回值为 true，即表示这两个条件必须同时成立才行；在

a>0 或者 b>0 时，表达式的返回值即为 true，这两个条件仅需要一个成立即可，读者可以参考表 3-10 中所列出的结果。

表 3-10　AND 及 OR 结果表

序　号	条件 1	条件 2	结果	
			&&（与）	||（或）
1	true	true	true	true
2	true	false	false	true
3	false	true	false	true
4	false	false	false	false

范例：验证逻辑运算符

```
public class OperatorDemo07 {
    public static void main(String[] args) {
        boolean a = true;                                    // 定义布尔变量，默认为true
        boolean b = false;                                   // 定义布尔变量，默认为false
        System.out.println("a || b = " + (a || b));         // 执行或操作
        System.out.println("a | b = " + (a | b));           // 执行或操作
        System.out.println("a && b = " + (a && b));         // 执行与操作
        System.out.println("a & b = " + (a & b));           // 执行与操作
    }
}
```

程序运行结果：

```
a || b = true
a | b = true
a && b = false
a & b = false
```

从程序结果可以发现，与和或的结果都符合表 3-10 给出的结果。另外，如果一个 if 语句中需要同时判断多个条件，则也可以使用以上的逻辑运算符，具体程序如下所示。

范例：判断多个条件

```
public class OperatorDemo08 {
    public static void main(String[] args) {
        int score = 50;                                      // 定义变量score
        if ((score < 0) || (score > 100)){                   // 判断成绩是否有错
            System.out.println("输入的成绩有错误！");
        }
        if ((score < 60) && (score > 49)){                   // 判断成绩结果
            System.out.println("成绩不及格，准备补考吧！");
        }
```

```
    }
}
```

程序运行结果：

```
成绩不及格，准备补考吧！
```

以上程序中的判断语句同时使用了多个判断条件，多个条件之间使用逻辑运算符进行连接。

提问：为什么会有两种与和或的操作？

与操作有“&&”和“&”两种，或操作有“||”和“|”两种，这两种在使用上有哪些区别？针对不同的情况该具体使用哪种呢？

回答：与分为短路与和非短路与，或分为短路或和非短路或。

从表 3-10 中读者可以发现这样的规律：

（1）对于与操作来说，如果第一个条件为假，则后面条件不管是真是假，最终的结果都是假。

（2）对于或操作来说，如果第一个条件为真，则后面条件不管是真是假，最终的结果都是真。

那么也就是说对于与和或来说，第一个条件就是一个关键性的条件，所以在 Java 中设置了短路与和短路或，所以对于短路与和与、短路或和或就有以下两点区别：

（1）对于与来说，要求所有的条件都判断，而如果使用短路与，第一个条件又为 false，则后面的条件将不再判断。

（2）对于或来说，要求所有的条件都判断，而如果使用短路或，第一个条件又为 true，则后面的条件将不再判断。

下面通过两段代码来验证以上两点，在讲解具体代码之前，请读者先来观察一下测试代码（一）的错误。

测试代码（一）：观察被除数为 0 的情况

```
public class OperatorDemo09 {
    public static void main(String[] args) {
        int x = 10 / 0;                          // 定义变量
        System.out.println("x = " + x);          // 错误，被除数为0
    }
}
```

程序运行之后会出现以下错误提示：

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

造成以上问题的根本在于被除数为 0，那么下面就在以上代码的基础上为读者进一步讲解短路与和短路或的区别。

测试代码（二）：验证“&”的作用

```
public class OperatorDemo10 {
    public static void main(String[] args) {
        if (10 != 10 & 10 / 0 == 0) {        // 非短路与
            System.out.println("条件满足");
```

```
        }
    }
}
```

在程序运行后，又出现了以上的错误提示：

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

造成以上的根本原因在于“与”操作要把所有的条件进行判断，所以在计算 10/0 时就出现了错误，而如果修改为短路与呢？见下面的代码。

测试代码（三）：验证“&&”的作用

```
public class OperatorDemo11 {
    public static void main(String[] args) {
        if (10 != 10 && 10 / 0 == 0) {      // 短路与
            System.out.println("条件满足");
        }
    }
}
```

程序运行后，可以发现不再出现以上的错误提示，就是因为第一个条件不满足，那么后面的条件就不再判断了，这就是短路与的作用。

测试代码（四）：验证“|”的作用

```
public class OperatorDemo12 {
    public static void main(String[] args) {
        if (10 == 10 | 10 / 0 == 0) {       // 非短路或
            System.out.println("条件满足");
        }
    }
}
```

程序运行后，出现了与之前一样的错误提示，如果现在改为“短路或”呢？见下面的代码。

测试代码（五）：验证“||”的作用

```
public class OperatorDemo13 {
    public static void main(String[] args) {
        if (10 == 10 || 10 / 0 == 0) {      // 短路或
            System.out.println("条件满足");
        }
    }
}
```

程序运行结果：

```
条件满足
```

从运行结果可以发现，程序不再出现错误，就是因为使用了短路或之后，如果第一个条件为 true，则后面的条件将不再判断。

7. 括号运算符

除了前面讲解的运算符外，括号()也是 Java 的运算符，如表 3-11 所示。

表 3-11 括号运算符

括号运算符	描　述
()	提高括号中表达式的优先级

括号运算符用来处理表达式的优先级。如下面的代码：

```
3 + 5 + 4 * 6 - 7                    // 未加括号的表达式
```

相信根据读者现在所学过的数学知识，这道题应该很容易解开。用加、减、乘、除的优先级（*、/的优先级大于+、-）来计算结果，这个式子的答案为 25。但是如果想先计算 3+5+4 及 6-7 之后再将两数相乘时，就必须将 3+5+4 及 6-7 分别加上括号，而成为下面的式子：

```
( 3 + 5 + 4 ) * ( 6 - 7 )            // 加上括号的表达式
```

经过括号运算符()的运作后，计算结果为-12，所以括号运算符()可以使括号内表达式的处理顺序优先。

范例：使用括号改变运算顺序

```
public class OperatorDemo14 {
    public static void main(String[] args) {
        int result1 = 3 + 5 + 4 * 6 - 7 ;                    // 保存计算结果
        int result2 = ((3 + 5 + 4) * (6 - 7)) ;              // 保存计算结果
        System.out.println("3 + 5 + 4 * 6 - 7 = " + result1);
        System.out.println("3 + 5 + 4 * 6 - 7 = " + result2);
    }
}
```

程序运行结果：

```
3 + 5 + 4 * 6 - 7 = 25
3 + 5 + 4 * 6 - 7 = -12
```

8. 位运算符

Java 除了具备高级语言的特点之外，也支持位运算操作。位运算操作就是指进行二进制位的运算，在 Java 中支持的位运算符如表 3-12 所示。

表 3-12 位运算符

序　号	位 运 算 符	描　述
1	&	按位与
2	|	按位或
3	^	异或（相同为 0，不同为 1）
4	~	取反
5	<<	左移位
6	>>	右移位
7	>>>	无符号右移位

在 Java 中所有的数据都是以二进制数据的形式进行运算的，即如果是一个 int 型变量，要采用位运算时则必须将其变为二进制数据。每一位二进制进行与、或、异或操作的结果如表 3-13 所示。

表 3-13　位运算的结果表

序　号	二进制数 1	二进制数 2	与操作（&）	或操作（|）	异或操作（^）
1	0	0	0	0	0
2	0	1	0	1	1
3	1	0	0	1	1
4	1	1	1	1	0

下面通过一段代码来观察位运算符，假设有 3 和 6 两个数字进行与、或、异或的操作，具体代码如下所示。

范例：位运算

```
public class OperatorDemo15 {
    public static void main(String[] args) {
        int x = 3 ;                    // 3的二进制数据：00000000 00000000
                                                          00000000 00000011
        int y = 6 ;                    // 6的二进制数据：00000000 00000000
                                                          00000000 00000110
        System.out.println(x & y);     // 与操作的结果：  00000000 00000000
                                                          00000000 00000010
        System.out.println(x | y);     // 或操作的结果：  00000000 00000000
                                                          00000000 00000111
        System.out.println(x ^ y);     // 异或操作结果：  00000000 00000000
                                                          00000000 00000101
    }
}
```

程序运行结果：

```
2
7
5
```

以上程序分别进行了与、或、异或的操作，下面为读者讲解具体的操作流程。3 对应的二进制数据为 101，但是在 Java 中整型数据的长度为 32 位，所以前面要补上 29 个 0，所以结果为 00000000 00000000 00000000 00000011，6 对应的二进制数据为 00000000 00000000 00000000 00000110，计算的过程如图 3-5 所示。

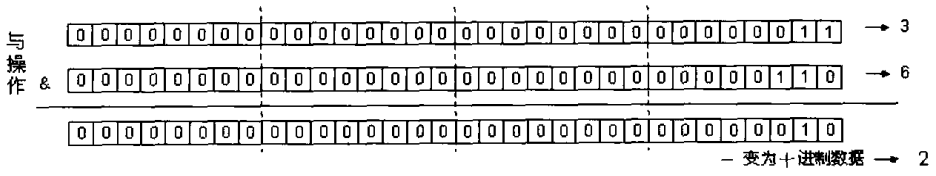


图 3-5　与、或、异或的操作

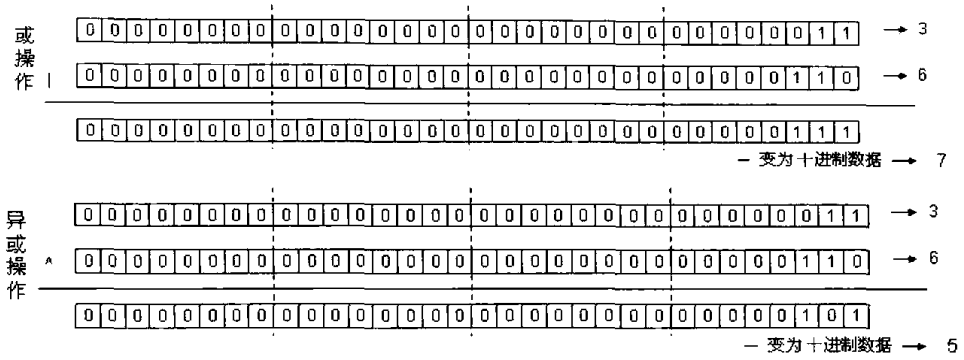


图 3-5　与、或、异或的操作（续）

在计算机的数据表示中只定义了正数的表示形式，并没有定义负数的表示形式，所以，负数一般都用补码的形式表示，正数的原码、反码、补码都相同，负数的反码是除符号位为 1 外，其他位全取反；补码就是“反码+1”。

范例：求出负数的反码

```
public class OperatorDemo16 {
    public static void main(String[] args) {
        int x = -3 ;    // -3的二进制数据：11111111 11111111 11111111 11111101
        System.out.println(x + "的反码是：" + ~x);
    }
}
```

程序运行结果：

```
-3的反码是：2
```

程序的运行结果为 2，因为在计算机中，负数都使用补码的形式计算，补码的计算是“反码+1”，然后再对-3 进行反码，操作过程如图 3-6 所示。

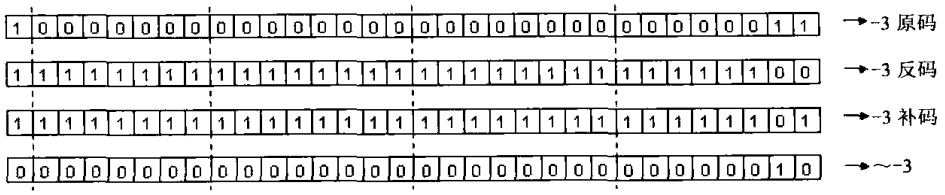


图 3-6　反码的计算

在 Java 中也提供了左移“<<”及右移“>>”两种操作。左移操作是将运算数的二进制码整体左移指定位数，左移之后的空位使用 0 来填充，如下面代码所示。

范例：左移操作

```
public class OperatorDemo17 {
    public static void main(String[] args) {
        int x = 3 ;     // 3的二进制数据：00000000 00000000 00000000 00000011
        System.out.println(x + "左移2位之后的内容：" + (x << 2));
    }
}
```

程序运行结果：

```
3左移2位之后的内容：12
```

程序的执行过程如图 3-7 所示。

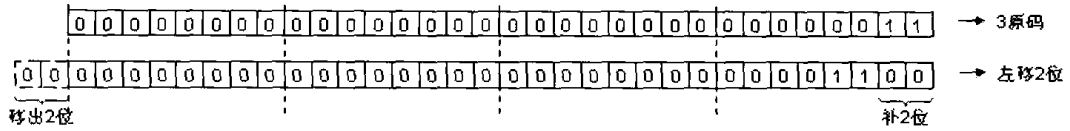


图 3-7　左移两位

右移操作“>>”是将运算数的二进制码整体右移，右移之后空出来的位置以符号位填充。如果是正数则使用 0 填充，如果是负数则使用 1 填充。

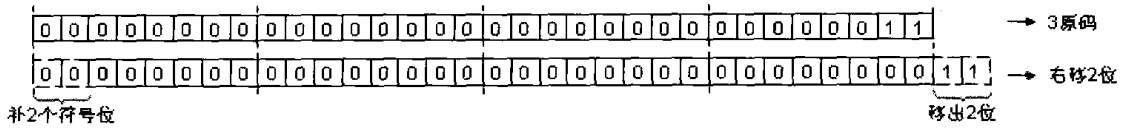
范例：右移操作

```
public class OperatorDemo18 {
    public static void main(String[] args) {
        int x = 3 ;      // 3的二进制数据：00000000 00000000 00000000 00000011
        int y = -3 ;     // -3的二进制数据：11111111 11111111 11111111 11111101
        System.out.println(x + "右移2位之后的内容：" + (x >> 2));
        System.out.println(y + "右移2位之后的内容：" + (y >> 2));
    }
}
```

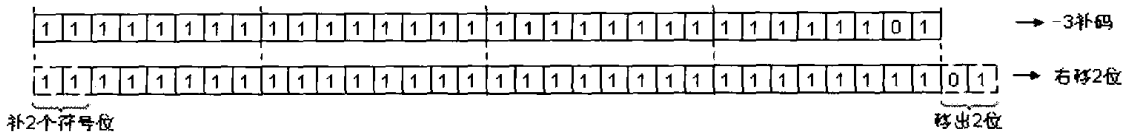
程序运行结果：

```
3右移2位之后的内容：0
-3右移2位之后的内容：-1
```

程序的执行过程如图 3-8 所示。



（a）正数的右移操作



（b）负数的右移操作

图 3-8　右移操作

从图 3-8（b）中可以看出因为其是负数，所以要采用补码的形式保存，所以最终的计算结果为-1。

以上的右移操作属于带符号位的右移操作，在 Java 中也提供了无符号的右移操作符“>>>”，使用此操作将以 0 填充空出来的位。

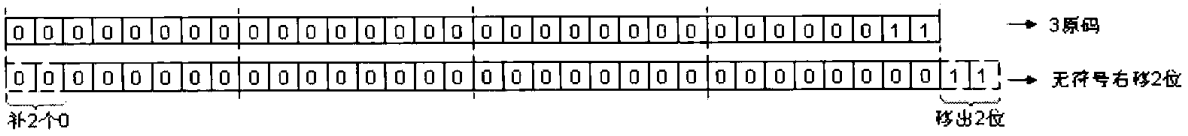
范例：无符号右移

```
public class OperatorDemo19 {
    public static void main(String[] args) {
        int x = 3 ;      // 3的二进制数据：00000000 00000000 00000000 00000011
        int y = -3 ;     // -3的二进制数据：11111111 11111111 11111111 11111101
        System.out.println(x + "右移2位之后的内容：" + (x >>> 2));
        System.out.println(y + "右移2位之后的内容：" + (y >>> 2));
    }
}
```

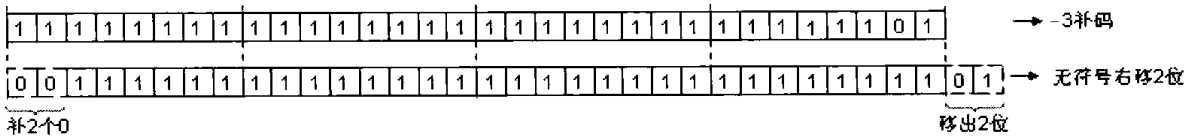
程序运行结果：

```
3右移2位之后的内容：0
-3右移2位之后的内容：1073741823
```

程序的执行过程如图 3-9 所示。



（a）正数的无符号右移



（b）负数的无符号右移

图 3-9　无符号右移操作

对于位操作只适用于 byte、short、int、char、long 类型，而且位操作之后原始的操作内容并不会发生任何的改变。

9．运算符的优先级

表 3-14 列出了各个运算符的优先级顺序，数字越小表示优先级越高。

表 3-14　运算符的优先级

优　先　级	运　算　符	类	结　合　性
1	()	括号运算符	由左至右
1	[]	方括号运算符	由左至右
2	!、+（正号）、-（负号）	一元运算符	由右至左
2	~	位逻辑运算符	由右至左
2	++、--	自增与自减运算符	由右至左
3	*、/、%	算术运算符	由左至右
4	+、-	算术运算符	由左至右
5	<<、>>	位左移、右移运算符	由左至右

续表

优　先　级	运　算　符	类	结　合　性
6	>、>=、<、<=	关系运算符	由左至右
7	==、!=	关系运算符	由左至右
8	&（位运算符 AND）	位逻辑运算符	由左至右
9	^（位运算符号 XOR）	位逻辑运算符	由左至右
10	|（位运算符号 OR）	位逻辑运算符	由左至右
11	&&	逻辑运算符	由左至右
12	||	逻辑运算符	由左至右
13	?:	三目运算符	由右至左
14	=	赋值运算符	由右至左

表 3-14 的最后一栏是运算符的结合性。结合性可以让程序设计者了解到运算符与操作数之间的关系及其相对位置。例如，当使用同一优先级的运算符时，结合性将非常重要，它决定谁会先被处理。如下面的例子：

```
a = b +d/5 * 4 ;
```

这个表达式中含有不同优先级的运算符，其中“/”与“*”的优先级高于“+”，而“+”又高于“=”，但“/”与“*”的优先级是相同的，究竟 d 该先除以 5 再乘以 4 呢，还是 5 乘以 4 后 d 再除以这个结果呢？结合性的定义就解决了这方面的困扰，算术运算符的结合性为“由左至右”，就是在相同优先级的运算符中，先由运算符左边的操作数开始处理，再处理右边的操作数。上面的式子中，由于“/”与“*”的优先级相同，因此 d 会先除以 5 再乘以 4 得到的结果加上 b 后，将整个值赋给 a 存放。

提示：多使用括号改变优先级。

部分刚刚入门的读者，一看到表 3-14 中的内容会比较头疼，因为一下子要记住这么多的内容似乎是一件很费劲的事，这里要告诉读者的是，对于以上的运算符优先级并没有必要完全记下来，必要时通过括号改变其优先级即可。

3.4.2　简洁表达式

表达式是由常量、变量或是其他操作数与运算符所组合而成的语句，如下面例子，均是表达式正确的使用方法：

```
-49                          // 表达式由一元运算符“-”与常量49组成
sum + 2                      // 表达式由变量sum、算术运算符与常量2组成
a+b－c/(d*3－9)              // 表达式由变量、常量与运算符所组成
```

此外，Java 还有一些将算术运算符和赋值运算符结合成为新的运算符，表 3-15 列出了这些运算符。

表 3-15 新的运算符

序 号	运 算 符	范 例 用 法	说 明	描 述
1	+=	a += b	a + b 的值存放到 a 中	a = a + b
2	-=	a -= b	a - b 的值存放到 a 中	a = a - b
3	*=	a *= b	a * b 的值存放到 a 中	a = a * b
4	/=	a /= b	a / b 的值存放到 a 中	a = a / b
5	%=	a %= b	a % b 的值存放到 a 中	a = a % b

以下几个表达式都是简洁的写法：

```
a++                          // 相当于a=a + 1
a-= 5                        // 相当于a=a - 5
b%= c                        // 相当于b=b % c
a/= b--                      // 相当于计算a=a / b 之后，再计算b--
```

上面的写法可以减少程序的行数，提高执行效率。

范例：验证简洁表达式

```
public class SimpleExpressDemo01 {
    public static void main(String[] args) {
        int a = 5, b = 8;                                    // 定义两个整型变量
        System.out.println("改变之前的数是： a = " + a + " , b = " + b);
        a += b;                                              // 等价于： a = a + b
        System.out.println("改变之后的数是： a = " + a + " , b = " + b);
    }
}
```

程序运行结果：

```
改变之前的数是： a = 5 , b = 8
改变之后的数是： a = 13 , b = 8
```

除了前面所提到的算术运算符和赋值运算符的结合可以存在于简洁的表达式中，自增、自减运算符也同样可以应用在简洁的表达式中。表 3-16 列出了一些简洁写法的运算符及其范例说明。

表 3-16 简洁表达式的范例

序号	运算符	范例	执行前		说 明	执行后	
			a	b		a	b
1	+=	a += b	12	3	a + b 的值存放到 a 中（同 a = a + b）	15	3
2	-=	a -= b	12	3	a - b 的值存放到 a 中（同 a = a - b）	9	3
3	*=	a *= b	12	3	a * b 的值存放到 a 中（同 a = a * b）	36	3
4	/=	a /= b	12	3	a / b 的值存放到 a 中（同 a = a / b）	4	3
5	%=	a %= b	12	3	a % b 的值存放到 a 中（同 a = a % b）	0	3
6	b++	a *= b++	12	3	a * b 的值存放到 a 后，b 加 1（同 a = a * b; b++）	36	4

续表

序号	运算符	范例	执行前		说　明	执行后	
			a	b		a	b
7	++b	a *= ++b	12	3	b 加 1 后，再将 a*b 的值存放到 a（同 b++; a=a*b）	48	4
8	b--	a *= b--	12	3	a * b 的值存放到 a 后，b 减 1（同 a=a*b; b--）	36	2
9	--b	a *= --b	12	3	b 减 1 后，再将 a*b 的值存放到 a（同 b--; a= a*b）	24	2

下面通过范例来说明这些简洁的表达式在程序中该如何应用。输入两个数，经过运算之后，来看这两个变量所存放的值有什么变化。

范例：验证简洁表达式

```
public class SimpleExpressDemo02 {
    public static void main(String[] args) {
        int a = 10, b = 6;                                  // 定义两个整型变量
        System.out.println("改变之前的数:  a = " + a + " , b = " + b);
        a -= b++;                                           // 执行计算操作
        System.out.println("改变之后的数:  a = " + a + " , b = " + b);
    }
}
```

程序运行结果：

```
改变之前的数:  a = 10 , b = 6
改变之后的数:  a = 4 , b = 7
```

3.5　选择与循环语句

到目前为止，本书所编写的程序都是简单的程序语句。如果想处理二选一或者多选一的操作，那么判断就是很好的选择，如果想处理重复的工作时，“循环”就是一个很好的选择，它可以运行相同的程序片段，还可以使程序结构化。本章将讲解选择与循环结构语句，还将讲解如何利用这些不同的结构编写出有趣的程序，让程序的编写更灵活，操控更方便。

3.5.1　程序的结构

一般来说程序的结构有顺序结构、选择结构和循环结构 3 种，这 3 种不同的结构有一个共同点，就是它们都只有一个入口，也只有一个出口。这些单一入、出口可以让程序易读、好维护，也可以减少调试的时间。下面以流程图的方式让读者了解这 3 种结构的不同。

1. 顺序结构

本书前面所讲的例子采用的都是顺序结构，程序至上而下逐行执行，一条语句执行完之后继续执行下一条语句，一直到程序的末尾。这种结构如图 3-10 所示。

顺序结构是在程序设计中最常使用到的结构，在程序中扮演了非常重要的角色，因为大部分的程序基本上都是依照由上而下的流程来设计的。

2. 选择结构

选择结构是根据条件的成立与否决定要执行哪些语句的一种结构，其流程图如图 3-11 所示。

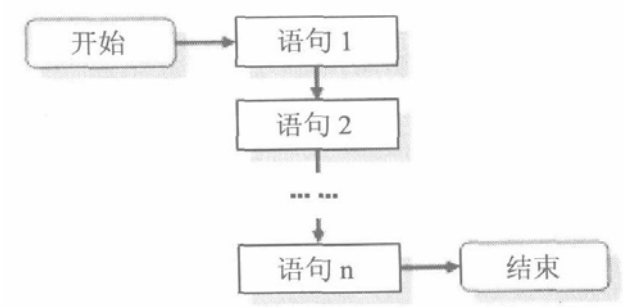


图 3-10　程序的顺序结构流程图

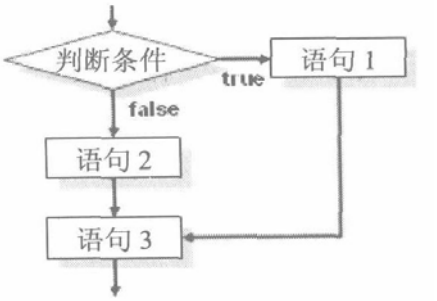


图 3-11　程序的选择结构流程图

这种结构可以依据判断条件的结构来决定要执行的语句。当判断条件的值为真时，就运行“语句 1”；当判断条件的值为假，则执行“语句 2”。不论执行哪一个语句，最后都会再回到“语句 3”继续执行。

范例：验证选择结构

```
public class IfDemo {
    public static void main(String[] args) {
        int x = 3;                                          // 定义整型变量x
        int y = 10;                                         // 定义整型变量y
        System.out.println("===== 比较开始 =====");          // 输出信息
        if (x > y) {                                        // 判断x是否比y大
            System.out.println("x比y大！");                  // 输出信息
        }
        if (x < y) {                                        // 判断x是否比y小
            System.out.println("x比y小！");                  // 输出信息
        }
        System.out.println("===== 比较完成 =====");          // 输出信息
    }
}
```

程序运行结果：

```
===== 比较开始 =====
x比y小！
===== 比较完成 =====
```

上面的程序完成了选择结果，当条件满足时会执行 if 语句之中的内容，读者可以自行修改程序中 x 和 y 的内容，以观察程序的运行结果。

3．循环结构

循环结构是根据判断条件的成立与否决定程序段落的执行次数，而这个程序段落就称为循环主体。循环结构的流程图如图 3-12 所示。

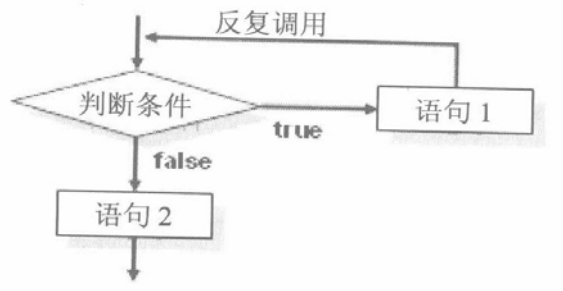


图 3-12　循环结构的流程图

3.5.2　选择结构

选择结构包括 if、if…else 及 switch 语句，语句中加上了选择结构之后，就像是“十字路口”，根据不同的选择，程序的运行会有不同的结果。

1．if 语句

前面简单地介绍了 if 的用法，要根据判断的结构来执行不同的语句时，使用 if 语句就是一个很好的选择，它会准确地检测判断条件成立与否，再决定是否要执行后面的语句。

【格式 3-3　if 语句的格式】

```
if (判断条件){
    语句1;
        ...
    语句2;
}
```

若在 if 语句主体中要处理的语句只有一个，则可省略左、右大括号。当判断条件的值不为假时，就会逐一执行大括号里面所包含的语句，if 语句的流程图如图 3-13 所示。

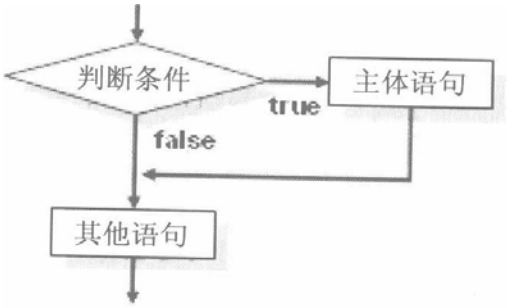


图 3-13　if 语句的流程图

选择结构除了 if 语句之外，还有 if…else 语句。在 if 语句中如果判断条件成立，即可执行语句主体内的语句，但若要在判断条件不成立时可以执行其他的语句，使用 if…else 语句

就可以节省判断的时间。

2. if…else 语句

当程序中存在含有分支的判断语句时，就可以用 if…else 语句处理。当判断条件成立时，即执行 if 语句主体；判断条件不成立时，则会执行 else 后面的语句主体。if…else 语句的格式如下：

【格式 3-4　if…else 语句的格式】

```
if (判断条件){
    语句主体1;
}else{
    语句主体2;
}
```

若在 if 语句或 else 语句主体中要处理的语句只有一个，可以将左、右大括号省去。if…else 语句的流程图如图 3-14 所示。

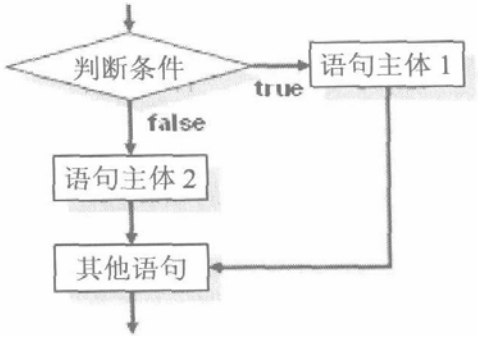


图 3-14　if…else 语句的流程图

范例：通过 if…else 语句判断一个数字是奇数还是偶数

```
public class IfElseDemo {
    public static void main(String[] args) {
        int x = 3;                                  // 定义整型变量x
        if (x % 2 == 1) {                           // 判断余数是否为1
            System.out.println("x是奇数！"); // 如果余数为1表示奇数
        } else {
            System.out.println("x是偶数！"); // 如果余数为0表示偶数
        }
    }
}
```

程序运行结果：

```
x是奇数！
```

以上程序通过取模的方式判断是奇数还是偶数，如果一个数字除以 2 余数为 0，则肯定是偶数，相反一定是奇数。

3．三目运算符

有一种运算符可以代替 if…else 语句，即三目运算符，如表 3-17 所示。

表 3-17　三目运算符

三目运算符	意　义
?:	根据条件的成立与否来决定结果为“:”前或“:”后的表达式

使用三目运算符时，操作数有 3 个，其格式如下：

【格式 3-5　if…else 语句的格式】

```
变量 = 条件判断? 表达式1：表达式2
```

将上面的格式以 if 语句解释，就是当条件成立时执行表达式 1，否则执行表达式 2，通常会将这两个表达式之一的运算结果指定给某个变量，也就相当于下面的 if…else 语句。

【格式 3-6　?:与 if…else 语句的相对关系】

```
if (条件判断)
   变量 x = 表达式1;
else
   变量 x = 表达式2;
```

范例：使用三目运算符求出两个数字中的最大值

```
public class MaxDemo {
    public static void main(String[] args) {
        int max = 0;                                    // 定义变量保存最大值
        int x = 3;                                      // 定义整型变量x
        int y = 10;                                     // 定义整型变量y
        max = x > y ? x : y;                            // 通过三目运算符求最大值
        System.out.println("最大值为：" + max);          // 输出求出的最大值
    }
}
```

程序运行结果：

```
最大值为：10
```

读者可以发现，使用条件运算符编写程序时较为简洁，它用一个语句就可以替代一长串的 if…else 语句，所以条件运算符的执行速度也较高。

4．if…else if…else 语句

如果需要在 if…else 中判断多个条件时，就需要 if…else if…else 语句了，其格式如下：

【格式 3-7　if…else if…else 语句】

```
if (条件判断1){
    语句主体1;
}else if (条件判断2){
```

```
    语句主体 2;
}
…  // 多个else if()语句
else{
    语句主体3;
}
```

这种方式用在含有多个判断条件的程序中，如下面的范例。

范例：使用 if…else if…else 语句

```
public class MoreIfElseDemo {
    public static void main(String[] args) {
        int x = 3;                                          // 定义整型变量x
        if (x == 1) {                                       // 判断x的值是否为1
            System.out.println("x的值是1！");                // 输出信息
        } else if (x == 2) {                                // 判断x的值是否为2
            System.out.println("x的值是2！");                // 输出信息
        } else if (x == 3) {                                // 判断x的值是否为3
            System.out.println("x的值是3！");                // 输出信息
        } else {                                            // 其他值
            System.out.println("x的值不是1、2、3中的一个！");
        }
    }
}
```

程序运行结果：

```
x的值是1！
```

5. switch 语句

switch 语句可以将多选一的情况简化，使程序简洁易懂。本节将介绍如何使用 switch 语句及 break 语句。此外，还要讨论在 switch 语句中如果不使用 break 语句会出现的问题。下面将介绍 switch 语句该如何使用。

要在许多的选择条件中找到并执行其中一个符合判断条件的语句时，除了可以使用 if…else 不断地判断外，还可以使用另一种更方便的方式，即多重选择——switch 语句。使用嵌套 if…else 语句最常发生的状况就是容易将 if 与 else 配对混淆，从而造成阅读及运行上的错误。使用 switch 语句则可以避免这种错误的发生。switch 语句的格式如下：

【格式 3-8　switch 语句】

```
switch (表达式){
    case 选择值1:    语句主体 1;
                     break;
    case 选择值2:    语句主体 2;
                     break;
        ......
```

```
    case 选择值n:     语句主体 n;
                      break;
    default:          语句主体;
}
```

要特别注意的是，在 switch 语句中选择值只能是字符或常量。

提示：switch 支持的新数据类型。

在 JDK 1.5 之后，switch 也支持枚举类的判断，这一点在本书的第 3 部分将为读者介绍。

switch 语句执行的流程如下：

（1）switch 语句先计算括号中表达式的结果，结果是数字、字符或是枚举。

（2）根据表达式的值检测是否符合 case 后面的选择值，若是所有 case 的选择值皆不符合，则执行 default 所包含的语句，执行完毕即离开 switch 语句。

（3）如果某个 case 的选择值符合表达式的结果，就会执行该 case 所包含的语句，一直遇到 break 语句后才离开 switch 语句。

（4）若是没有在 case 语句结尾处加上 break 语句，则会一直执行到 switch 语句的尾端才离开 switch 语句。break 语句在下面的章节中会介绍到，读者只要先记住 break 是跳出语句即可。

（5）若是没有定义 default 该执行的语句，则什么也不会执行，直接离开 switch 语句。

根据上面的描述，可以绘制出如图 3-15 所示的 switch 语句的流程图。

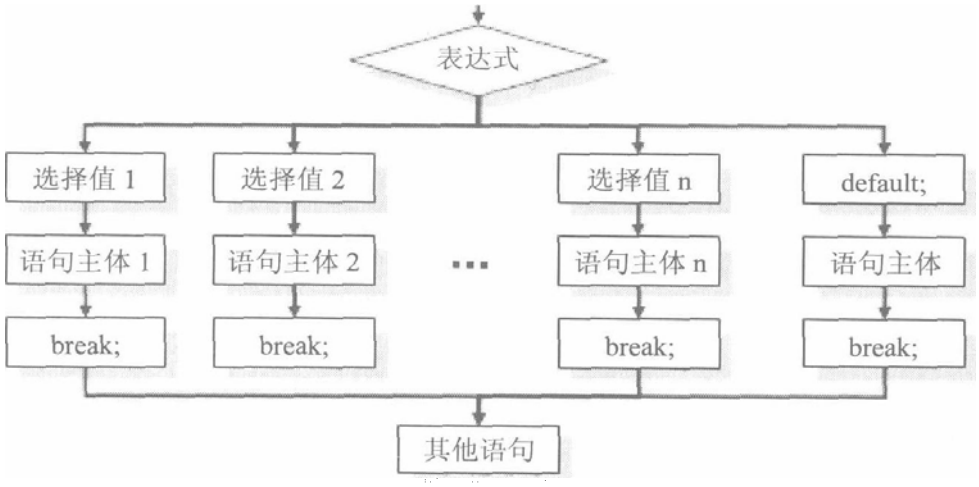


图 3-15　switch 语句的流程图

范例：以下程序验证了 switch 语句的作用

```
public class SwitchDemo01 {
    public static void main(String[] args) {
        int x = 3;                              // 声明整型变量x
        int y = 6;                              // 声明整型变量y
        char oper = '+';                        // 声明字符变量
        switch (oper) {                         // 将字符作为switch的判断条件
```

```
            case '+': {                                    // 判断字符内容是否是“+”
                System.out.println("x + y = " + (x + y));
                break;                                     // 退出switch
            }
            case '-': {                                    // 判断字符内容是否是“-”
                System.out.println("x - y = " + (x - y));
                break;                                     // 退出switch
            }
            case '*': {                                    // 判断字符内容是否是“*”
                System.out.println("x * y = " + (x * y));
                break;                                     // 退出switch
            }
            case '/': {                                    // 判断字符内容是否是“/”
                System.out.println("x / y = " + (x / y));
                break;                                     // 退出switch
            }
            default: {                                     // 其他字符
                System.out.println("未知的操作！");
                break ;                                    // 退出switch
            }
        }
    }
}
```

程序运行结果：

```
x + y = 9
```

读者可以自行将 oper 中的操作修改为“+”、“-”、“*”、“/”等，如果设置的是一个未知的操作，程序将提示“未知的操作！”

提示：break 语句的作用。

从以上程序可以发现，在每一个 case 语句之后都加上了一个 break 语句，如果不加入此语句，则 switch 语句会从第一个满足条件的 case 开始依次执行操作，如下面的测试代码：

```
public class SwitchDemo02 {
    public static void main(String[] args) {
        int x = 3;                                         // 声明整型变量x
        int y = 6;                                         // 声明整型变量y
        char oper = '+';                                   // 声明字符变量
        switch (oper) {                                    // 将字符作为switch的判断条件
            case '+': {                                    // 判断字符内容是否是“+”
                System.out.println("x + y = " + (x + y));
            }
            case '-': {                                    // 判断字符内容是否是“-”
```

```
                System.out.println("x - y = " + (x - y));
            }
            case '*': {                          // 判断字符内容是否是“*”
                System.out.println("x * y = " + (x * y));
            }
            case '/': {                          // 判断字符内容是否是“/”
                System.out.println("x / y = " + (x / y));
            }
            default: {                           // 其他字符
                System.out.println("未知的操作！");
            }
        }
    }
}
```

程序运行结果：

```
x + y = 9
x - y = -3
x * y = 18
x / y = 0
未知的操作！
```

从运行结果可以发现，程序在第一个条件满足之后，由于没有设置相应的 break 语句，所以从第一个满足条件开始就依次向下继续执行。

3.5.3　循环结构

1．while 循环

while 是循环语句，也是条件判断语句。当事先不知道循环该执行多少次时，就要用到 while 循环。while 循环的格式如下：

【格式 3-9　while 循环语句】

```
while (循环条件判断){
语句1 ;
语句2 ;
...
语句n ;
循环条件更改 ;
}
```

当 while 循环主体有且只有一个语句时，可以将大括号省去。在 while 循环语句中只有一个判断条件，它可以是任何表达式。当判断条件的值为真，循环就会执行一次，再重复测试判断条件，执行循环主体，直到判断条件的值为假，才会跳离 while 循环。下面列出了

while 循环执行的流程：

（1）第 1 次进入 while 循环前，必须先为循环控制变量（或表达式）赋起始值。

（2）根据判断条件的内容决定是否要继续执行循环，如果条件判断值为真（true），继续执行循环主体；若条件判断值为假（false），则跳出循环执行其他语句。

（3）执行完循环主体内的语句后，重新为循环控制变量（或表达式）赋值（增加或减少），由于 while 循环不会自动更改循环控制变量（或表达式）的内容，所以在 while 循环中为循环控制变量赋值的工作要由设计者自己来做，完成后再回到步骤（2）重新判断是否继续执行循环。

根据上述的程序流程，可以绘制出如图 3-16 所示的 while 循环流程图。



图 3-16　while 循环流程图

范例：使用 while 循环进行累加操作

```
public class WhileDemo {
    public static void main(String[] args) {
        int x = 1;                                          // 定义整型变量x
        int sum = 0;                                        // 定义整型变量，保存
                                                               累加结果
        while (x <= 10) {                                   // 判断循环条件
            sum += x;                                       // 执行累加操作
            x++;                                            // 修改循环条件
        }
        System.out.println("1-->10累加结果为：" + sum);     // 输出结果
    }
}
```

程序运行结果：

```
1-->10累加结果为：55
```

从以上程序可以发现在 while 语句执行之前先进行循环条件的判断（x<=10），在 while 语句之后对循环条件进行修改（x++），如果程序中没有修改循环条件，那么程序就将出现

“死循环”的情况。

2．do…while 循环

do…while 循环也是用于未知循环执行次数的情况，而 while 循环及 do…while 循环最大的不同就是进入 while 循环前，while 语句会先测试判断条件的真假，再决定是否执行循环主体，而 do…while 循环则是每次都是先执行一次循环主体，然后再测试判断条件的真假，所以无论循环成立的条件是什么，使用 do…while 循环时，至少都会执行一次循环主体。do…while 循环的格式如下：

【格式 3-10　do…while 循环语句】

```
do{
    语句1 ;
    语句2 ;
    ...
    语句n ;
    循环条件改变 ;
}while (循环条件判断);
```

当循环主体只有一个语句时，可以将左、右大括号省去。第一次进入 do…while 循环语句时，不管判断条件（它可以是任何表达式）是否符合执行循环的条件，都会直接执行循环主体。循环主体执行完毕才开始测试判断条件的值。如果判断条件的值为真，则再次执行循环主体，如此重复测试判断条件，执行循环主体，直到判断条件的值为假，才会跳离 do…while 循环。下面列出了 do…while 循环执行的流程：

（1）进入 do…while 循环前，要先为循环控制变量（或表达式）赋起始值。

（2）直接执行循环主体，循环主体执行完毕，才开始根据判断条件的内容决定是否继续执行循环，条件判断值为真（true）时，继续执行循环主体；条件判断值为假（false）时，则跳出循环，执行其他语句。

（3）执行完循环主体内的语句后，重新为循环控制变量（或表达式）赋值（增加或减少），由于 do…while 循环和 while 循环一样，不会自动更改循环控制变量（或表达式）的内容，所以在 do…while 循环中赋值循环控制变量的工作要由自己来做，然后再回到步骤（2）重新判断是否继续执行循环。

根据上述的描述，可以绘制出如图 3-17 所示的 do…while 循环流程图。

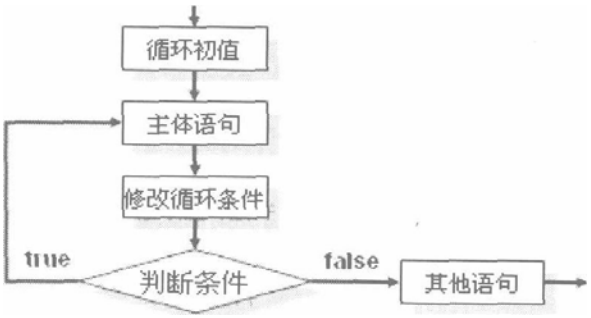


图 3-17　do…while 循环流程图

范例：使用 do…while 循环完成累加操作

```
public class DoWhileDemo {
    public static void main(String[] args) {
        int x = 1;                                          // 定义整型变量x
        int sum = 0;                                        // 定义整型变量，保存
                                                               累加结果
        do {                                                // do…while至少先执
                                                               行一次
            sum += x;                                       // 执行累加操作
            x++;                                            // 修改循环条件
        } while (x <= 10);                                  // 判断循环
        System.out.println("1-->10累加结果为：" + sum);  // 输出累加结果
    }
}
```

程序运行结果：

```
1-->10累加结果为：55
```

从程序的运行结果可以发现 while 和 do…while 的操作结果是一样的。但 do…while 与 while 循环不同的是 do…while 操作中就算条件不满足，也会至少执行一次。而 while 如果条件不满足，则一次也不会被执行。

3．for 循环

对于 while 和 do…while 两种循环来讲，操作时并不一定要明确地知道循环的次数，而如果开发者已经明确地知道了循环次数，那么就可以使用另外一种循环语句——for 循环。

【格式 3-11　for 循环语句】

```
for (赋值初值；判断条件；赋值增减量){
    语句1;
    ...
    语句n;
}
```

若是在循环主体中要处理的语句只有一个，可以将大括号省去。下面列出了 for 循环的流程：

（1）第一次进入 for 循环时，要为循环控制变量赋起始值。

（2）根据判断条件的内容检查是否要继续执行循环，当判断条件值为真（true）时，继续执行循环主体内的语句；判断条件值为假（false）时，则会跳出循环，执行其他语句。

（3）执行完循环主体内的语句后，循环控制变量会根据增减量的要求更改循环控制变量的值，然后再回到步骤（2）重新判断是否继续执行循环。

根据以上描述，可以绘制出如图 3-18 所示的 for 循环流程图。

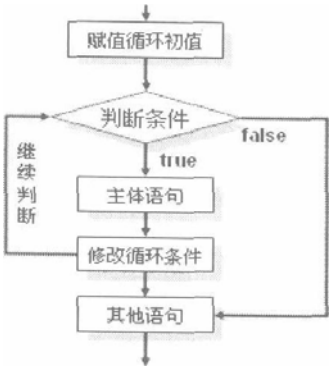


图 3-18　for 循环流程图

范例：使用 for 循环完成累加操作

```
public class ForDemo {
    public static void main(String[] args) {
        int sum = 0;                                    // 定义变量保存累加结果
        for (int x = 1; x <= 10; x++) {                 // 使用for循环
            sum += x;                                   // 执行累加操作
        }
        System.out.println("1-->10累加结果为：" + sum);// 输出累加结果
    }
}
```

程序运行结果：

```
1-->10累加结果为：55
```

4．循环的嵌套

多个循环语句是可以嵌套操作的，如果现在要打印一个九九乘法表，则肯定要使用两层循环完成。

范例：打印九九乘法表

```
public class ForNestedDemo {
    public static void main(String[] args) {
        for (int i = 1; i <= 9; i++) {                  // 第一层循环
            for (int j = 1; j <= i; j++) {              // 第二层循环
                System.out.print(i + "*" + j + "=" + (i * j) + "\t");
            }
            System.out.print("\n");                     // 换行
        }
    }
}
```

程序运行结果：

```
1*1=1
2*1=2   2*2=4
```

```
3*1=3   3*2=6   3*3=9
4*1=4   4*2=8   4*3=12  4*4=16
5*1=5   5*2=10  5*3=15  5*4=20  5*5=25
6*1=6   6*2=12  6*3=18  6*4=24  6*5=30  6*6=36
7*1=7   7*2=14  7*3=21  7*4=28  7*5=35  7*6=42  7*7=49
8*1=8   8*2=16  8*3=24  8*4=32  8*5=40  8*6=48  8*7=56  8*8=64
9*1=9   9*2=18  9*3=27  9*4=36  9*5=45  9*6=54  9*7=63  9*8=72  9*9=81
```

程序说明：

（1）i 为外层循环的循环控制变量，j 为内层循环的循环控制变量。

（2）当 i 为 1 时，符合外层 for 循环的判断条件（i<=9），进入另一个内层 for 循环主体，由于是第一次进入内层循环，所以 j 的初值为 1，符合内层 for 循环的判断条件（j<=i），进入循环主体，输出 i*j 的值(1*1=1)，如果最后 j 的值仍符合内层 for 循环的判断条件(j<=i)，则再次执行计算与输出的工作，直到 j 的值大于 i 时，离开内层 for 循环，回到外层循环。此时，i 会加 1 成为 2，符合外层 for 循环的判断条件，继续执行内层 for 循环主体，直到 i 的值大于 9 时离开嵌套循环。

3.5.4 循环的中断

在 Java 语言中，可以使用如 break、continue 等中断语句。站在结构化程序设计的角度上，并不鼓励用户使用中断语句，因为这些中断语句会增加调试及阅读的困难。因此建议读者尽量不要使用中断语句。在本节中将为读者介绍 break 及 continue 语句。

1. break 语句

break 语句可以强迫程序中断循环，当程序执行到 break 语句时，即会离开循环，继续执行循环外的下一个语句，如果 break 语句出现在嵌套循环中的内层循环，则 break 语句只会跳出当前层的循环。以下面的 for 循环为例，在循环主体中有 break 语句时，当程序执行到 break 时，会离开循环主体，而继续执行循环外层的语句。

【格式 3-12　break 语句格式】

```
for (初值赋值; 判断条件; 设增减量){
    语句1;
    语句2;
      ...
      break;
      ...      // 若执行break语句，则此块内的语句将不会被执行
    语句n;
}
...
```



下面的程序为利用 for 循环输出循环变量 i 的值，当 i 除以 3 所取的余数为 0 时，使用 break 语句的跳离循环，并于程序结束前输出循环变量 i 的最终值。

范例：使用 break

```
public class BreakDemo {
    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {          // 使用for循环
            if (i == 3) {                       // 如果i的值为3，则退出整个循环
                break;                          // 退出整个循环
            }
            System.out.println("i = " + i);// 打印信息
        }
    }
}
```

程序运行结果：

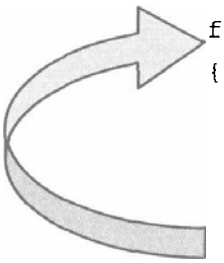
```
i = 0
i = 1
i = 2
```

从程序的运行结果中可以发现，当 i 的值为 3 时，判断语句满足，则执行 break 语句退出整个循环。

2. continue 语句

continue 语句可以强迫程序跳到循环的起始处，当程序运行到 continue 语句时，会停止运行剩余的循环主体，而是回到循环的开始处继续运行。在下面的 for 循环中，在循环主体中有 continue 语句，当程序执行到 continue 时，就会回到循环的起点，继续执行循环主体的部分语句。

【格式 3-13　continue 语句格式】



```
for (初值赋值；判断条件；设增减量)
{
        语句1;
        语句2;
          ...
        continue
                ...  // 若执行continue语句，则此处将不会被执行
        语句n;
}
          ...
```

范例：将前面 for 循环的程序修改为 continue 循环，观察运行结果

```
public class ContinueDemo {
    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {          // 使用for循环
            if (i == 3) {                       // 如果i的值为3，则退出整个循环
                continue;                       // 退出一次循环
```

```
                }
                System.out.println("i = " + i);// 打印信息
            }
        }
    }
```

程序运行结果：

```
i = 0
i = 1
i = 2
i = 4
i = 5
...
```

从程序的运行结果中可以发现，当 i 的值为 3 时，程序并没有向下执行输出语句，而是退回到了循环判断处继续向下执行，所以 continue 只是中断了一次的循环操作。

提示：局部变量。

另外，需要提醒读者的是，在循环语句中定义的变量属于局部变量，所谓的局部变量是指此变量只能在循环语句中使用，而在循环语句之外则无法使用。

范例： 如以下代码所示

```
for(int i=1;i<=10;i++){
    sum = sum + i ;
    System.out.println("i = "+i+", sum = "+sum);
}    // 循环之外的语句将不能再使用变量i
```

变量 i 的有效范围

3.6 本章要点

1．Java 的数据类型可分为基本数据类型和引用数据类型两种。

2．Unicode 为每个字符制定了一个唯一的数值，在任何的语言、平台、程序都可以安心地使用。

3．布尔（boolean）类型的变量只有 true（真）和 false（假）两个值。

4．数据类型的转换可分为“自动类型转换”与“强制类型转换”两种。

5．算术运算符的成员有加法运算符、减法运算符、乘法运算符、除法运算符和余数运算符。

6．if 语句可依据判断的结果来决定程序的流程。

7．自增与自减运算符有相当大的便利性，利用它们可提高程序的简洁程度。

8．括号()是用来处理表达式的优先级的，也是 Java 的运算符。

9．当表达式中有类型不匹配时，有下列的处理方法：

（1）占用较少字节的数据类型会转换成占用较多字节的数据类型。

（2）有 short 和 int 类型时，则用 int 类型。

（3）字符类型会转换成 short 类型。

（4）int 类型转换成 float 类型。

（5）若一个操作数的类型为 double，则其他的操作数也会转换成 double 类型。

（6）布尔类型不能转换至其他的类型。

10．程序的结构包含顺序结构、选择结构和循环结构。

11．需要重复执行某项功能时，循环就是最好的选择。可以根据程序的需求与习惯，选择使用 Java 所提供的 for、while 及 do…while 循环。

12．break 语句可以让强制程序脱离循环。当程序运行到 break 语句时，即会离开循环，继续执行循环外的下一个语句，如果 break 语句出现在嵌套循环中的内层循环，则 break 语句只会离开当前层循环。

13．continue 语句可以强制程序跳到循环的起始处，当程序运行到 continue 语句时，即会停止运行剩余的循环主体，转到循环的开始处继续运行。

14．选择结构包括 if、if…else 及 switch 语句，语句中加上了选择结构后，就像是“十字路口”，根据不同的选择，程序的运行会有不同的方向与结果。

15．在循环中也可以声明变量，但所声明的变量只是局部变量，只要跳出循环，这个变量便不能再使用。

3.7　习　　题

1．打印出所有的“水仙花数”，所谓“水仙花数”是指一个 3 位数，其各位数字立方和等于该数本身。例如，153 是一个“水仙花数”，因为 153=（1 的三次方+5 的三次方+3 的三次方）。153、370、371、407

2．通过代码完成两个整数内容的交换。

3．给定 3 个数字，求出这 3 个数字中的最大值，并将最大值输出。

4．判断某数能否被 3、7、5 同时整除。

5．编写程序，分别利用 while 循环、do…while 循环和 for 循环求出 100～200 的累加和。

6．编写 Java 程序，求 13-23+33-43+…+973-983+993-1003 的值。-1000

7．编写一个程序，实现两个数字的交换。

8．编写一个程序求 3 个数中的最大值。

9．编写一个程序，实现 1～100 的累加。

10．求 1～1000 之间可以同时被 3、5、7 整除的数字。

11．编程求 1!+2!+3!+…20!的值。268040729

12．使用 for 循环打印下面的图形：

```
    *
   * *
  * * *
 * * * *
* * * * *
```

第 4 章　数组与方法

通过本章的学习可以达到以下目标：

- ☑ 掌握数组的定义及使用方法。
- ☑ 掌握数组的引用传递。
- ☑ 掌握方法及方法的重载。
- ☑ 可以使用方法接收和返回一个数组。
- ☑ 了解 Java 对数组的操作支持。

若想定义多个重复类型的变量，那么使用数组是一个很好的方法。对于前面几章中的程序，都是用到什么代码就编写什么代码，而对于有些代码要反复调用的情况，可以将代码声明成一个方法，以供程序反复调用。本章将为读者介绍 Java 中数组的声明及使用、方法的参数传递、方法重载以及引用传递的基本概念。**本章视频录像讲解时间为 1 小时 48 分钟，源代码在光盘对应的章节下。**

4.1　数组的定义及使用

数组是一组相关数据的集合，一个数组实际上就是一连串的变量，数组可以分为一维数组、二维数组和多维数组。

4.1.1　一维数组

一维数组可以存放上千万个数据，并且这些数据的类型是完全相同的。

要使用 Java 的数组，必须经过声明数组和分配内存给数组两个步骤，这两个步骤的语法结构如下：

【格式 4-1　一维数组的声明与分配内存】

```
数据类型 数组名[ ] = null ;                    // 声明一维数组
数组名=new 数据类型[长度] ;                    // 分配内存给数组
```

对于数组的声明方式也用下面这种形式：

```
数据类型[ ] 数组名= null ;                     // 声明一维数组
```

这两种语法本身没有任何的区别，在本书中将使用第一种方式，此种数组因为数组名称之后只有一个“[]”，所以也称为一维数组。

数组的声明格式中，“数据类型”是声明数组元素的数据类型，常见的类型有整型、浮点型与字符型等。“数组名”是用来统一这组相同数据类型的元素的名称，其命名规则和变量相同，建议读者使用有意义的名称为数组命名。数组声明后实际上是在栈内存中保

存了此数组的名称，接下来便要在堆内存中配置数组所需的内存。其中，“长度”是告诉编译器所声明的数组要存放多少个元素，而 new 则是命令编译器根据括号里的长度在堆内存中开辟一块堆内存供该数组使用。下面是声明一维数组并分配内存给该数组的一个范例：

```
int score[] = null ;     // 声明整型数组score
score = new int[3];      // 为整型数组score分配内存空间，其元素个数为3
```

在上例中的第 1 行中，当声明一个整型数组 score 时，score 可视为数组类型的变量，此时，这个变量并没有包含任何内容，编译器仅会在栈内存中分配一块内存给它，用来保存指向数组实体的地址的名称，如图 4-1 所示。

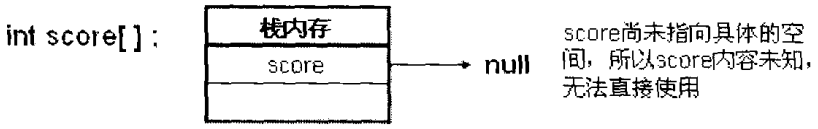


图 4-1　声明一个整型数组

提问：数组声明的疑问。

在数组的声明上为什么要写上一个 null 呢？如“int score[] = null;”。

回答：null 表示引用数据类型的默认值。

在数据类型划分上曾经讲解过，数组属于引用数据类型，那么对于引用数据类型来说，其默认值是 null，表示暂时还没有任何指向的内存空间。在 JDK 1.5 之后可以不用给数据默认值，但是为了开发方便，本书建议读者采用赋给变量默认值的方式。

声明之后，接着要做堆内存分配的操作，也就是上例中第 2 行语句。这一行会开辟 3 个可供保存整数的内存空间，并把此内存空间的参考地址赋给 score 变量。其内存分配的流程图如图 4-2 所示。

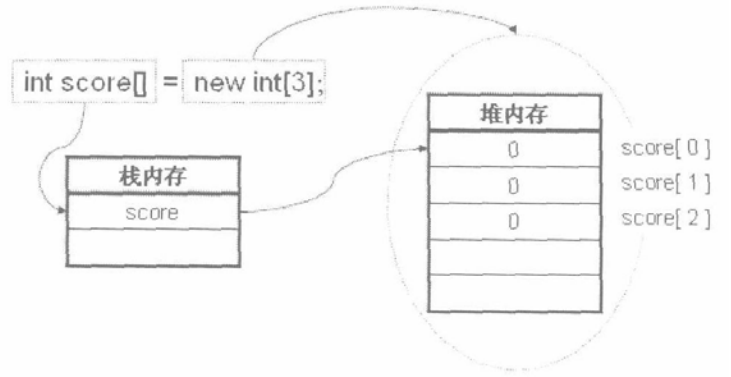


图 4-2　内存分配流程图

从图 4-2 中可以发现，一个数组开辟了堆内存之后，将在堆内存中保存数据，并将堆内存的操作地址给了数组名称 score。因为数组是引用数据类型，所以数组变量 score 所保存的并非是数组的实体，而是数组堆内存的参考地址。

提示：堆栈内存的解释。

数组操作中，在栈内存中保存的永远是数组的名称，只开辟了栈内存空间的数组是永远无法使用的，必须有指向的堆内存才可以使用，要想开辟新的堆内存则必须使用 new 关键字，然后只是将此堆内存的使用权交给了对应的栈内存空间，而且一个堆内存空间可以同时被多个栈内存空间指向，即一个人可以有多个名字，人就相当于堆内存，名字就相当于栈内存，如图 4-3 所示。

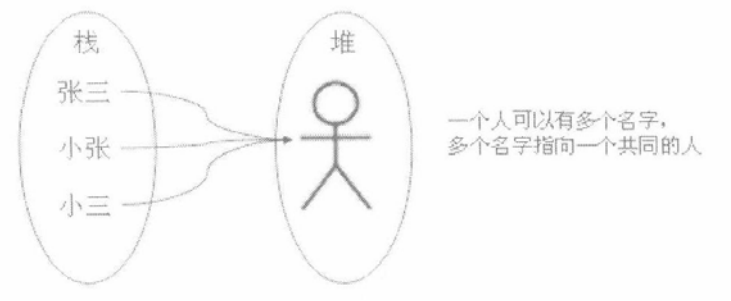


图 4-3　人和名字的关系

除了用格式 4-1 的方法来声明并分配内存给数组外，也可以用较为简洁的方式把两行缩成一行来编写，其格式如下：

【格式 4-2　声明数组的同时分配内存】

```
数据类型　数组名[] = new 数据类型[个数]
```

上述的格式会在声明的同时分配一块内存空间供该数组使用。下面的范例是声明整型数组 score，并开辟可以保存 10 个整数的内存给 score 变量。

```
int score[] = new int[10] ;
// 声明一个元素个数为10的整型数组score，同时开辟一块内存空间供其使用
```

在 Java 中，由于整数数据类型所占用的空间为 4 个字节，而整型数组 score 可保存的元素有 10 个，所以上例中占用的内存共有 4*10=40 个字节。图 4-4 是数组 score 的保存方式。

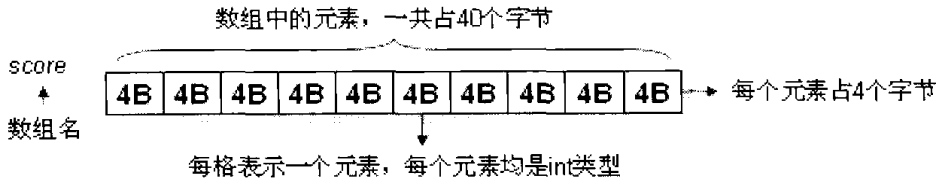


图 4-4　数组 score 的保存方式

4.1.2　数组中元素的表示方法

若要访问数组中的元素，可以利用索引来完成。Java 的数组索引编号由 0 开始，以 score[10]整型数组为例，score[0]代表第 1 个元素，score[1]代表第 2 个元素，score[9]为数组中第 10 个元素（也就是最后一个元素）。图 4-5 为 score 数组中元素的表示法及排列方式。

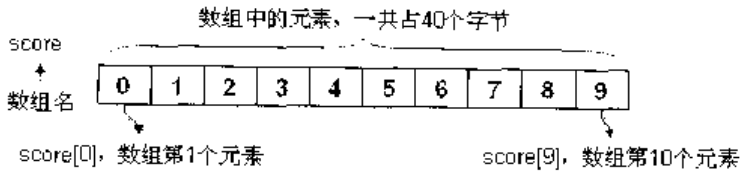


图 4-5　数组中元素的排列

范例：数组的声明及输出

```
public class ArrayDemo01 {
    public static void main(String[] args) {
        int score[] = null;                                    // 声明数组，但未开辟堆
                                                                  内存空间
        score = new int[3];                                    // 为数组开辟堆内存空间
        System.out.println("score[0] = " + score[0]);// 分别输出每个元素
        System.out.println("score[1] = " + score[1]);// 分别输出每个元素
        System.out.println("score[2] = " + score[2]);// 分别输出每个元素
        for (int x = 0; x < 3; x++) {                          // 使用循环依次输出数组
                                                                  中的全部内容
            System.out.println("score["+x+"] = " + score[x]) ;
        }
    }
}
```

程序运行结果：

```
score[0] = 0
score[1] = 0
score[2] = 0
score[0] = 0
score[1] = 0
score[2] = 0
```

从程序中可以发现，对于数组的访问采用“数组名称[下标]”的方式，之前一共开辟了 3 个空间大小的数组，所以下标的取值为 0～2，假设程序中取出的内容超过了这个下标，如 score[3]，则程序运行时会出现以下的错误提示：

```
java.lang.ArrayIndexOutOfBoundsException: 3
```

提示的内容为数组索引超出绑定异常，这一点在使用数组中是经常出现的问题，读者在编写程序时应该引起注意。此外，可以发现以上数组中的内容都为 0，这是因为声明的是整型数组，而此时又没有为整型数组中的内容赋值，所以现在都是默认值，整型的默认值为 0。下面的范例将为数组中的元素进行赋值并进行输出。

范例：为数组中的元素赋值并进行输出

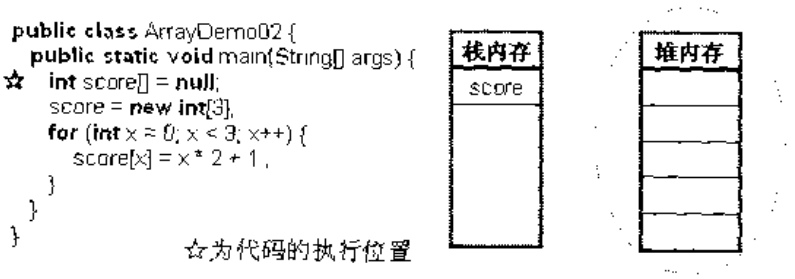
```
public class ArrayDemo02 {
    public static void main(String[] args) {
        int score[] = null;                                    // 声明数组，但未开辟堆内存
```

```
        score = new int[3];                    // 为数组开辟堆内存空间，大小为3
        for (int x = 0; x < 3; x++) {          // 为数组中的每个元素赋值
            score[x] = x * 2 + 1 ;             // 为每一个元素赋值
        }
        for (int x = 0; x < 3; x++) {          // 使用循环依次输出数组中的全部内容
            System.out.println("score[" + x + "] = " + score[x]);
        }
    }
}
```

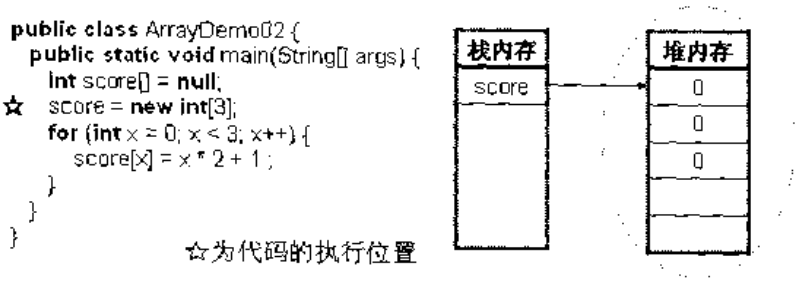
程序运行结果：

```
score[0] = 1
score[1] = 3
score[2] = 5
```

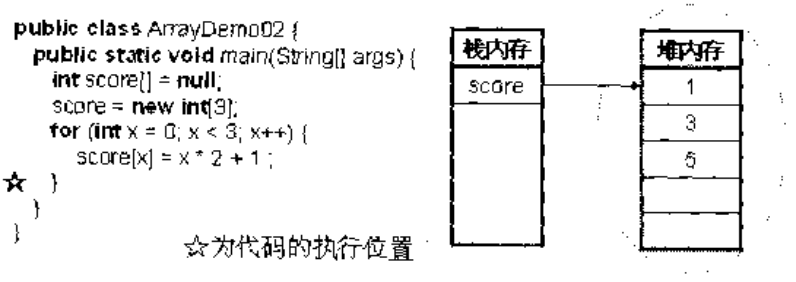
以上程序的作用是将奇数赋值给数组中的每个元素。以上程序的内存操作流程如图 4-6 所示的图形。



（a）数组声明时在栈内存中开辟



（b）为数组开辟相应的堆内存空间



（c）为数组中的每个元素赋值

图 4-6　内存操作流程

要特别注意的是，在 Java 中取得数组的长度（也就是数组元素的长度）可以利用“数组名称.length”，如下面的格式：

【格式 4-3　数组长度的取得】

```
数组名称.length　→　返回一个int型数据
```

范例：取得一个数组的长度

```
public class ArrayDemo03 {
    public static void main(String[] args) {
        int score[] = new int[3];                          // 声明并实例化数组
        System.out.println("数组长度为" + score.length); // 求出数组长度
    }
}
```

程序运行结果：

```
数组长度为3
```

4.1.3　数组的静态初始化

数组的内容分为动态初始化和静态初始化两种，之前所讲解的代码是采用先声明数组之后为数组中的每个内容赋值的方式完成的。也可以通过数组静态初始化，在数组声明时就指定其具体内容。

如果想直接在声明时就给数组赋初值，可以利用大括号完成。只要在数组的声明格式后面再加上初值的赋值即可，如下面的格式：

【格式 4-4　数组赋初值】

```
数据类型 数组名[] = {初值0 , 初值1 , … , 初值n}
```

在大括号内的初值会依序指定给数组的第 1、…、n+1 个元素。此外，在声明时，并不需要将数组元素的个数列出，编译器根据所给出的初值个数来判断数组的长度。如下面的数组声明及赋初值范例：

```
int score[] = {91,92,93,94,95,96};                    // 数组声明并赋初值
```

在上面的语句中声明了一个整型数组 score，虽然没有特别指明数组的长度，但是由于大括号中的初值有 10 个，编译器会分别依序指定给各元素存放，score[0]为 91，…，score[5]为 96。

范例：数组的静态初始化

```
public class ArrayDemo04 {
    public static void main(String[] args) {
        int score[] = {91,92,93,94,95,96};             // 使用静态初始化声明数组
        for (int x = 0; x < score.length; x++) {       // 循环输出
            System.out.println("score["+x+"] = " + score[x]) ;
        }
    }
}
```

程序运行结果：

```
score[0] = 91
score[1] = 92
score[2] = 93
score[3] = 94
score[4] = 95
score[5] = 96
```

4.1.4 数组应用范例

前面已经为读者介绍了一维数组的使用，下面通过一个范例来讲解数组的一些基本应用。

范例：求出数组中的最大和最小值

```
public class ArrayDemo05 {
    public static void main(String[] args) {
        int score[] = {67,89,87,69,90,100,75,90};  // 静态初始化数组
        int max = 0 ;                              // 定义变量保存最大值
        int min = 0 ;                              // 定义变量保存最小值
        max = min = score[0] ;                     // 把第1个元素的内容赋值给
                                                      max和min
        for (int x = 0; x < score.length; x++) {   // 循环求出最大和最小
            if(score[x]>max){                      // 依次判断后续元素是否比
                                                      max大
                max = score[x] ;                   // 如果大，则修改max内容
            }
            if(score[x]<min){                      // 依次判断后续的元素是否
                                                      比min小
                min = score[x] ;                   // 如果小，则修改min内容
            }
        }
        System.out.println("最高成绩：" + max);      // 输出最大值
        System.out.println("最低成绩：" + min);      // 输出最小值
    }
}
```

程序运行结果：

```
最高成绩：100
最低成绩：67
```

将变量 min 与 max 初值设成数组的第 1 个元素后，再逐一与数组中的各元素相比。比 min 小，就将该元素的值指定给 min 存放，使 min 的内容保持最小；同样，当该元素比 max 大时，就将该元素的值指定给 max 存放，使 max 的内容保持最大。for 循环执行完，也就表示数组中所有的元素都已经比较完毕，此时，变量 min 与 max 的内容就是最小值与最大值，

此过程如图 4-7 所示。

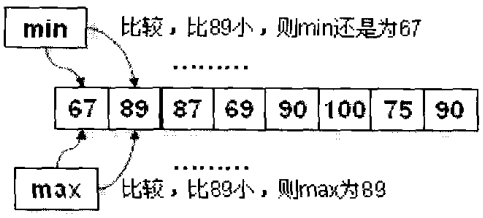


图 4-7　循环的比较过程

在数组的操作中，数组的排序也是一种比较常见的操作，以下范例为一个数组的排序操作。

范例：对整型数组按照由小到大的顺序进行排列

```
public class ArrayDemo06 {
    public static void main(String[] args) {
        int score[] = { 67, 89, 87, 69, 90, 100, 75, 90 };      // 声明数组
        for (int i = 1; i < score.length; i++) {                // 循环判断
            for (int j = 0; j < score.length; j++) {
                if (score[i] < score[j]) {                      // 交换位置
                    int temp = score[i];
                    score[i] = score[j];
                    score[j] = temp;
                }
            }
        }
        for (int i = 0; i < score.length; i++) {                // 数组输出
            System.out.print(score[i] + "\t");
        }
    }
}
```

程序运行结果：

```
67  69  75  87  89  90  90  100
```

以上程序采用了冒泡算法进行排序。即把数组中的每一个元素进行比较，如果第 i 个元素大于第 i+1 个元素，那么就要把两个数字进行交换，这样进行反复的比较就可以将一个数组按照由小到大的顺序进行排序。

范例：修改之前代码，显示每次的排序结果

```
public class ArrayDemo07 {
    public static void main(String[] args) {
        int score[] = { 67, 89, 87, 69, 90, 100, 75, 90 };      // 声明数组
        for (int i = 1; i < score.length; i++) {                // 循环判断
            for (int j = 0; j < score.length; j++) {
                if (score[i] < score[j]) {                      // 交换位置
```

```
                    int temp = score[i];
                    score[i] = score[j];
                    score[j] = temp;
                }
            }
            System.out.print("第"+i+"次排序的结果：\t") ;
            for (int j = 0; j < score.length; j++) {                    // 循环输出
                System.out.print(score[j] + "\t");
            }
            System.out.println("") ;                                    // 换行
        }
        System.out.print("最终的排序结果为：\t") ;
        for (int i = 0; i < score.length; i++) {
            System.out.print(score[i] + "\t");
        }
    }
}
```

程序运行结果：

```
第1次排序的结果：  67  100 87  69  89  90  75  90
第2次排序的结果：  67  87  100 69  89  90  75  90
第3次排序的结果：  67  69  87  100 89  90  75  90
第4次排序的结果：  67  69  87  89  100 90  75  90
第5次排序的结果：  67  69  87  89  90  100 75  90
第6次排序的结果：  67  69  75  87  89  90  100 90
第7次排序的结果：  67  69  75  87  89  90  90  100
最终的排序结果为：  67  69  75  87  89  90  90  100
```

读者可以根据以上结果观察每次循环后的数组内容顺序的改变。

4.1.5 二维数组

如果说可以把一维数组当成几何中的线性图形，那么二维数组就相当于是一个表格，像 Excel 中的表格那样，如图 4-8 所示。



图 4-8　一个二维数组就相当于是一个表格

二维数组声明的方式和一维数组类似，内存的分配也要使用 new 关键字。其声明与分配内存的格式如下：

【格式 4-5　二维数组的声明格式】

```
数据类型　数组名[][] ;
数组名 = new 数据类型[行的个数][列的个数] ;
```

与一维数组不同的是，二维数组在分配内存时，必须告诉编译器二维数组行与列的个数。因此，在格式 4-5 中，“行的个数”是告诉编译器所声明的数组有多少行，“列的个数”则是说明该数组有多少列，如下面的范例：

```
int score[][] ;                    // 声明整型数组score
score = new int[4][3] ;            // 配置一块内存空间，供4行3列的整型数组score使用
```

同样地，可以用较为简洁的方式来声明数组，其格式如下：

【格式 4-6　二维数组的声明格式】

```
数据类型 数组名[][] = new 数据类型[行的个数][列的个数];
```

若用上述的写法，则是在声明的同时就开辟了一块内存空间，以供该数组使用。编写的例子如下：

```
int score[][] = new int[4][3] ; // 声明整型数组score，同时为其开辟一块内存空间
```

上面的语句中，整型数据 score 可保存的元素有 4*3 = 12 个，而在 Java 中，int 数据类型所占用的空间为 4 个字节，因此，该整型数组占用的内存共为 4*12 = 48 个字节，如图 4-9 所示。

行＼列	0	1	2
0	0	0	0
1	0	0	0
2	0	0	0
3	0	0	0

图 4-9　二维数组存储

范例：二维数组的定义及使用

```
public class ArrayDemo08 {
    public static void main(String[] args) {
        int score[][] = new int[4][3];                  // 声明并实例化二维数组
        score[0][1] = 30 ;                              // 为数组中的部分内容赋值
        score[1][0] = 31 ;                              // 为数组中的部分内容赋值
        score[2][2] = 32 ;                              // 为数组中的部分内容赋值
        score[3][1] = 33 ;                              // 为数组中的部分内容赋值
        score[1][1] = 30 ;                              // 为数组中的部分内容赋值
        for (int i = 0; i < score.length; i++) {        // 外层循环行
            for(int j=0;j<score[i].length;j++){         // 内层循环列
                System.out.print(score[i][j] + "\t");
            }
            System.out.println("") ;                    // 换行
        }
    }
}
```

程序运行结果：

```
0    30   0
31   30   0
0    0    32
0    33   0
```

从程序中读者应该可以找到一个规律，即一维数组如果要全部输出，则需要使用一层循环，而二维数组要想全部输出，则应使用两层循环，同理，对于 N 维数组，则要使用 N 层循环。

二维数组也可以利用大括号进行静态初始化，只要在数组的声明格式后面再加上所赋的初值即可，如下面的格式：

【格式 4-7　二维数组静态初始化格式】

```
数据类型 数组名[][] = {
                {第0行初值},
                {第1行初值},
                ...
                {第n行初值},
};
```

要特别注意的是，用户不需要定义数组的长度，因此，在数组名后面的中括号中不必填入任何内容。此外，在大括号内还有几组大括号，每组大括号内的初值会依序指定给数组的第 0、1、…、n 行元素，如图 4-10 所示。

```
int score[][] = {
{ 67, 61 }, { 78, 89, 83 },
{ 99, 100, 98, 66, 95 } };
```

行＼列	0	1	2	3	4
0	67	61			
1	78	89	83		
2	99	100	98	66	95

图 4-10　静态初始化二维数组

下面是关于二维数组 score 声明及赋初值的范例。

范例：使用静态初始化声明一个二维数组

```
public class ArrayDemo09 {
    public static void main(String[] args) {
        // 静态初始化一个二维数组，每行的数组元素个数不一样
        int score[][] = { { 67, 61 }, { 78, 89, 83 }, { 99, 100, 98, 66, 95 } };
        for (int i = 0; i < score.length; i++) {          // 外层循环输出行
            for (int j = 0; j < score[i].length; j++) {// 内存循环输出列
                System.out.print(score[i][j] + "\t");  // 输出每一个元素
            }
            System.out.println("");                     // 换行
        }
    }
}
```

程序运行结果：

```
67  61
78  89  83
99  100 98  66  95
```

4.1.6　多维数组

经过前面一维、二维数组的练习后不难发现，想要提高数组的维数，只要在声明数组时将索引与中括号再加一组即可，所以三维数组的声明为 int score[][][]，而四维数组为 int score[][][][]…，依此类推。

使用多维数组时，输入、输出的方式和一维、二维数组相同，但是每多一维，嵌套循环的层数就必须多一层，所以维数越高的数组其复杂度也就越高。下面的程序为三维数组在声明数组时即赋初值，再将其元素值输出并计算总和。

范例：定义和使用三维数组

```
public class ArrayDemo10 {
    public static void main(String[] args) {
        // 定义一个三维数组，使用静态初始化的方式
        int score[][][] = { { { 5, 1 }, { 6, 7 } }, { { 9, 4 }, { 8, 3 } } };
        for (int i = 0; i < score.length; i++) {                    // 第1层循环
            for (int j = 0; j < score[i].length; j++) {             // 第2层循环
                for (int k = 0; k < score[i][j].length; k++) { // 第3层循环
                    System.out.println("score["+i+"]["+j+"]["+k+"]="
                            + score[i][j][k]);                      // 输出每一
                                                                       个元素
                }
            }
        }
    }
}
```

程序运行结果：

```
score[0][0][0] = 5
score[0][0][1] = 1
score[0][1][0] = 6
score[0][1][1] = 7
score[1][0][0] = 9
score[1][0][1] = 4
score[1][1][0] = 8
score[1][1][1] = 3
```

读者可以清楚地发现，因为定义的是三维数组，所以在输出时使用了 3 层循环，如果是四维数组输出的话就肯定需要使用 4 层循环，N 维数组就要使用 N 层循环，但是一般不

建议使用多维的数组进行操作。

4.2 方法的声明及使用

4.2.1 方法的定义

方法就是一段可重复调用的代码段，例如，有某段长度约 100 行的代码，要在多个地方使用此段代码，如果在各个地方都重复编写此部分代码，则肯定会比较麻烦，而且此部分代码如果进行修改，也比较困难，所以此时可以将此部分代码定义成一个方法，以供程序反复调用。

提示：方法与函数的关系。

在有些书中把方法称为函数，这两者本身并没有什么区别，是同样的概念，只是称呼方式不一样。

方法的定义在 Java 中可以使用多种方式，格式 4-8 定义的方法可以直接使用主方法（main()）调用，是因为在方法声明处加上了 public static 关键字，关键字的作用将在后面的章节中详细解释。方法暂时使用如下的语句进行定义：

【格式 4-8 方法的定义格式】

```
public static 返回值类型 方法名称（类型 参数1,类型 参数2,...）{
    程序语句;
    [return 表达式];
}
```

方法的主体

要特别注意的是，如果不需要传递参数到方法中，只要将括号写出，不必填入任何内容。此外，如果方法没有返回值，则在返回值类型处要明确写出 void，此时，在方法中的 return 语句可以省略。方法执行完后无论是否存在返回值都将返回到方法的调用处并向下继续执行。

范例：定义一个方法，在主方法中进行调用

```
public class MethodDemo01 {
    public static void main(String[] args) {
        printInfo() ;                                   // 调用printInfo()方法
        printInfo() ;                                   // 调用printInfo()方法
        printInfo() ;                                   // 调用printInfo()方法
        System.out.println("Hello World!") ;
    }
    // 此处由于此方法是由main方法直接调用的，所以一定要加上public static
    public static void printInfo() {                    // 此处方法没有返回值
        char c[] = {'H','e','l','l','o',
                ',','L','X','H'};                       // 定义一个字符数组
```

```
        for (int x = 0; x < c.length; x++) {    // 循环输出
            System.out.print(c[x]) ;
        }
        System.out.println("") ;                // 换行
    }
}
```

程序运行结果：

```
Hello,LXH
Hello,LXH
Hello,LXH
Hello World!
```

从程序中可以发现，因为 printInfo()方法本身不需要任何的返回值声明，所以使用了 void 关键字进行声明，表示此方法不需要任何的返回值，所以不需要编写 return 语句。

如图 4-11 所示为上述范例的执行流程图。

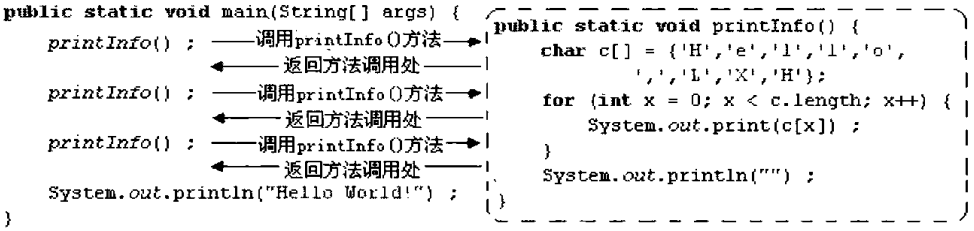


图 4-11　printInfo()方法的执行流程图

从图 4-11 中可以清楚地看出，当调用 printInfo()方法时，程序就会跳转到 printInfo()方法中执行，当 printInfo()方法全部执行完之后就会返回调用处向下继续执行。

注意：方法命名规范要求。

在定义类时，全部单词的首字母必须大写，那么在定义方法时也有命名规范要求，即第一个单词的首字母小写，之后每个单词的首字母大写，如 printInfo()方法。希望读者在日后的开发中一定要养成好的习惯。

前面已经为读者介绍了没有返回值的方法，下面来看一个有返回值的方法，此方法的功能就是计算两个数字相加后的结果，代码如下。

范例：有返回值的方法

```
public class MethodDemo02 {
    public static void main(String[] args) {
        int one = addOne(10, 20);           // 调用整数的加法操作
        float two = addTwo(10.3f, 13.3f);   // 调用浮点数的加法操作
        System.out.println("addOne的计算结果：" + one) ;
        System.out.println("addTwo的计算结果：" + two) ;
    }
    // 定义方法，完成两个整数的加法操作，方法返回一个int型数据
```

```
    public static int addOne(int x, int y) {
        int temp = 0;                          // temp为局部变量，只在此方法中有效
        temp = x + y;                          // 执行加法计算
        return temp;                           // 返回计算结果
    }
    // 定义方法，完成两个浮点数的加法操作，方法返回一个float型数据
    public static float addTwo(float x, float y) {
        float temp = 0;                        // temp为局部变量，只在此方法中有效
        temp = x + y;                          // 执行加法计算
        return temp;                           // 返回计算结果
    }
}
```

程序运行结果：

```
addOne的计算结果：30
addTwo的计算结果：23.6
```

另外，要提醒读者的是，在方法中可以定义多个变量，这些变量只在方法的内部起作用，所以也可以把这些变量称为局部变量。

4.2.2 方法的重载

方法的重载就是方法名称相同，但参数的类型和参数的个数不同。通过传递参数的个数及类型的不同可以完成不同功能的方法调用。如下面代码。

范例：验证方法的重载

```
public class MethodDemo03 {
    public static void main(String[] args) {
        int one = add(10, 20);                 // 调用有两个参数的整型加法
        int two = add(10, 20, 30);             // 调用有3个参数的整型加法
        float three = add(10.3f, 13.3f);       // 调用有两个参数的浮点型加法
        System.out.println("add(int x, int y)的计算结果：" + one) ;
        System.out.println("add(int x, int y, int z)的计算结果：" + two);
        System.out.println("add(float x, float y)的计算结果：" + three);
    }
    public static int add(int x, int y) {     // 定义add方法,完成两个整数相加
        int temp = 0;                          // 定义局部变量
        temp = x + y;                          // 执行加法计算
        return temp;                           // 返回计算结果
    }
    public static int add(int x, int y, int z){ // 定义add方法，完成3个整数相加
        int temp = 0;                          // 定义局部变量
        temp = x + y + z;                      // 执行加法操作
        return temp;                           // 返回计算结果
```

```
    }
    public static float add(float x, float y) {// 定义add方法，完成两个浮点数相加
        float temp = 0;                        // 定义局部变量
        temp = x + y;                          // 执行加法操作
        return temp;                           // 返回计算结果
    }
}
```

程序运行结果：

```
add(int x, int y)的计算结果：30
add(int x, int y, int z)的计算结果：60
add(float x, float y)的计算结果：23.6
```

从程序中可以发现 add()方法被重载了 3 次，而且每次重载时的参数类型或个数都有不同，所以在调用时，会根据参数的类型和个数自动进行区分，如图 4-12 所示。

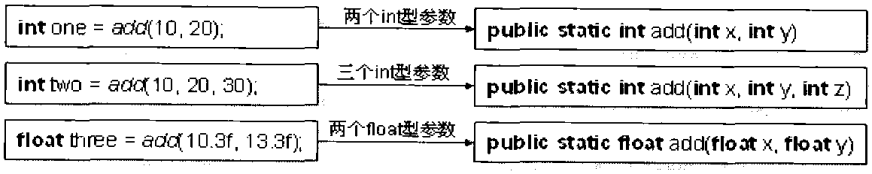


图 4-12　方法重载的调用

提示：System.out.println()方法也属于重载方法。

读者从前面的内容中可以知道对于屏幕打印语句 System.out.print()方法来说可以打印任何的数据，如：

```
System.out.println(3);                  // 输出整数
System.out.println(33.3);               // 输出浮点数
System.out.println('3');                // 输出字符
System.out.println(true);               // 输出布尔型
System.out.println(3 + 3);              // 输出计算结果
```

发现程序可以打印数字、小数、字符、布尔类型等数据，也就是说 print()方法被重载了。

提示：重载的注意事项。

另外还要提醒读者的是，方法的重载一定只是在参数上的类型或个数不同，而下面的代码不叫方法重载。

范例：错误的方法重载

```
public static float add(int x, int y) {// 返回float型，但参数类型及个数一致
    float temp = 0;
    temp = x + y;
    return temp;
}
public static int add(int x, int y) { // 返回int型，但参数类型及个数一致
    int temp = 0;
```

```
        temp = x + y;
        return temp;
    }
```

从上面的程序中可以发现，方法的接收参数类型和个数完全一样，但只是方法的返回值类型不一样，所以上面的代码程序是不可能编译通过的，所以不是方法重载。

4.2.3 使用 return 结束一个方法

在 Java 的方法定义中，可以使用 return 语句直接结束方法，如下所示。

范例：使用 return 结束方法

```
public class MethodDemo05 {
    public static void main(String[] args) {
        System.out.println("1、调用fun()方法之前。") ;
        fun(10) ;              // 调用fun()方法
        System.out.println("2、调用fun()方法之后。") ;
    }
    public static void fun(int x) {
        System.out.println("3、进入fun()方法。") ;
        if(x==10){
            return ;           // 结束方法，返回被调用处
        }
        System.out.println("4、正常执行完fun()方法。") ;
    }
}
```

程序运行结果：

```
1、调用fun()方法之前。
3、进入fun()方法。
2、调用fun()方法之后。
```

从程序运行结果发现，虽然在 return 中没有返回任何的内容，但是一旦执行到了 return 语句之后，方法将不再执行，而返回到被调用处继续向下执行。

4.2.4 方法的递归调用

递归调用是一种特殊的调用形式，是方法自己调用自己，如图 4-13 所示。

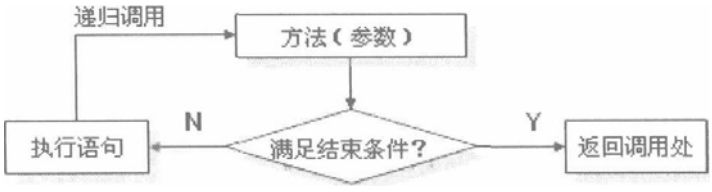


图 4-13 方法的递归调用

例如，要完成一个数字的累加操作，除了可以使用之前的循环方式外，还可以使用递归调用。

范例：递归调用

```
public class MethodDemo06 {
  public static void main(String args[]) {
    System.out.println("计算结果：" + sum(100)); // 调用操作
  }
  public static int sum(int num) {                    // 定义方法用于求和操作
    if (num == 1) {                                   // 判断是否是加到了最后一个数
      return 1;
    } else {
      return num + sum(num - 1);                      // 递归调用
    }
  }
}
```

程序运行结果：

```
计算结果：5050
```

上面程序完成了递归方法的调用，就相当于 100+sum(99)+sum(98)+…+sum(1)，从程序的 sum()方法中可以发现，递归调用时必须有一个明确的结束条件，然后不断改变传入的数据，才可以实现递归调用。

注意：尽量避免使用递归调用。

递归调用在操作时如果处理不好，则有可能出现内存的溢出，所以对于这种方法调用形式使用时要谨慎。

4.3　数组的引用传递

4.3.1　传递及返回数组

前面的操作传递和返回的都是基本数据类型，方法中也可用来传递和返回数组。如果要向方法中传递一个数组，则方法的接收参数必须是符合其类型的数组。而且数组属于引用数据类型，所以在把数组传递进方法之后，如果方法对数组本身做了任何修改，修改结果也将保存下来。

范例：向方法中传递数组

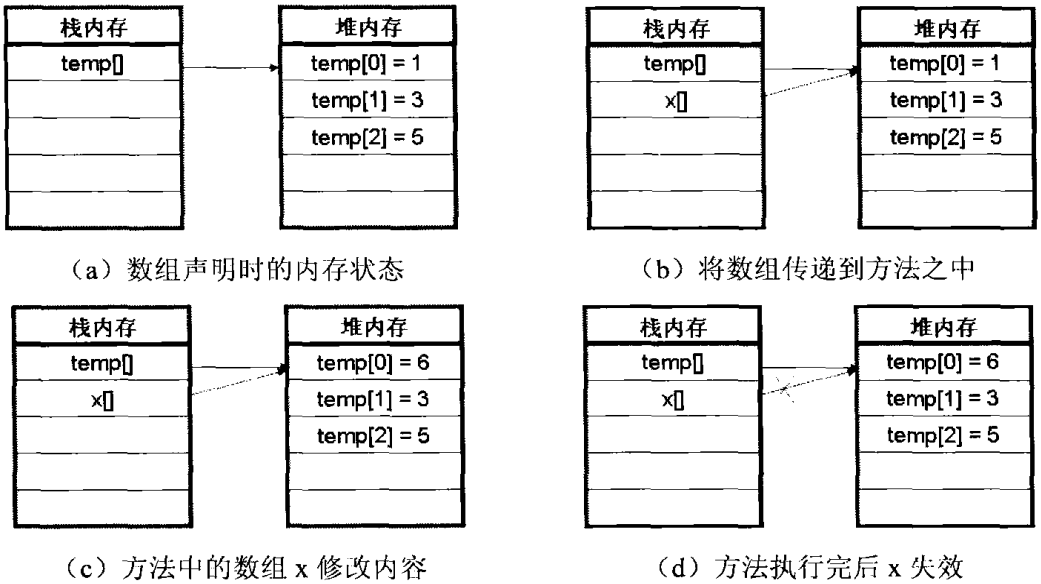
```
public class ArrayRefDemo01 {
    public static void main(String[] args) {
        int temp[] = { 1, 3, 5 };                     // 使用静态初始化定义数组
        fun(temp);                                    // 传递数组引用
        for (int i = 0; i < temp.length; i++) {// 循环输出
```

```
            System.out.print(temp[i] + "、");
        }
    }
    public static void fun(int x[]) {           // 接收整型数组引用
        x[0] = 6;                               // 修改第1个元素的内容
    }
}
```

程序运行结果：

```
6、3、5、
```

在程序中将一个整型数组 temp 传递到了方法之中，然后在 fun()方法中将此整型数组的第 1 个元素的内容修改为 6，因为数组是引用数据类型，所以，即使方法本身没有任何的返回值，修改后的结果也会被保存下来，向方法中传递数组的过程如图 4-14 所示。



（a）数组声明时的内存状态　（b）将数组传递到方法之中

（c）方法中的数组 x 修改内容　（d）方法执行完后 x 失效

图 4-14　向方法中传递数组的过程

从图 4-14 中可以发现，一开始声明的 temp 数组的内容是“1、3、5”，但是将此数组传递到了方法中，使用了数组 x 接收，也就是说此时 temp 实际上是将堆内存空间的使用权传递给了方法，为数组的具体内容起了一个别名 x，然后在方法中通过 x 修改数组中的内容，方法执行完毕之后，数组 x 因为是局部变量所以就失效了，但是对于数组内容的改变却保留了下来，这就是数组引用传递的过程。关于引用数据类型的引用传递，本书在面向对象部分还会有更加完整的介绍。

既然方法可以接收一个数组，那么方法也就可以返回一个数组，只需要在返回值类型声明处明确地写出返回的数组类型即可。

范例：使用方法返回一个数组

```
public class ArrayRefDemo02 {
    public static void main(String[] args) {
        int temp[] = fun();                     // 通过方法实例化数组
```

```
        print(temp);                                    // 向print()方法中传递数组
    }
    public static void print(int x[]) {                 // 接收数组
        for (int i = 0; i < x.length; i++) {            // 循环输出
            System.out.print(x[i] + "、");
        }
    }
    public static int[] fun() {                         // 此方法返回一个数组引用
        int ss[] = { 1, 3, 5, 7, 9 };                   // 定义一个数组
        return ss;                                      // 返回数组
    }
}
```

程序运行结果：

```
1、3、5、7、9、
```

4.3.2 范例——数组排序

范例：将数组排序程序修改成一个方法的调用形式

```
public class ArrayRefDemo03 {
    public static void main(String[] args) {
        int score[] = { 67, 89, 87, 69, 90, 100, 75, 90 }; // 定义整型数组
        int age[] = { 31, 30, 18, 17, 8, 9, 1, 39 };       // 定义整型数组
        sort(score);                                       // 数组排序
        print(score);                                      // 数组打印
        System.out.println("\n---------------------------------------");
        sort(age);                                         // 数组排序
        print(age);                                        // 数组打印
    }
    public static void sort(int temp[]) {                  // 数组排序
        for (int i = 1; i < temp.length; i++) {            // 使用冒泡算法
                                                           //   排序
            for (int j = 0; j < temp.length; j++) {
                if (temp[i] < temp[j]) {
                    int x = temp[i];                       // 交换位置操作
                    temp[i] = temp[j];
                    temp[j] = x;
                }
            }
        }
    }
    public static void print(int temp[]) {                 // 输出数组内容
```

```
            for (int i = 0; i < temp.length; i++) {
                System.out.print(temp[i] + "\t");
            }
        }
    }
```

程序运行结果：

```
67  69  75  87  89  90  90  100
---------------------------------------
1   8   9   17  18  30  31  39
```

以上的程序将排序和输出的功能分别定义成了一个方法，然后直接调用这两个方法，就可以完成排序或输出的功能，在调用 sort()或 print 方法时将数组的引用传递过去，这样就可以直接对主方法中定义的 score 和 age 数组进行排序或输出，如图 4-15 所示。

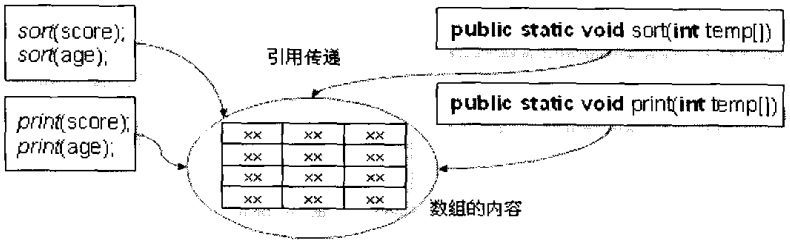


图 4-15　引用传递

当然，对于排序操作，在 Java 本身也是有类库支持的，读者可以直接使用“java.util.Arrays.sort(数组名称)”对数组进行排序。

范例：使用 Java 类库完成数组的排序操作

```
public class ArrayRefDemo04 {
    public static void main(String[] args) {
        int score[] = { 67, 89, 87, 69, 90, 100, 75, 90 }; // 定义整型数组
        int age[] = { 31, 30, 18, 17, 8, 9, 1, 39 };       // 定义整型数组
        java.util.Arrays.sort(score);                      // 使用Java提供
                                                              的排序操作
        print(score);                                      // 输出数组
        System.out.println("\n---------------------------------------");
        java.util.Arrays.sort(age);                        // 使用Java提供
                                                              的排序操作
        print(age);
    }
    public static void print(int temp[]) {                 // 数组输出
        for (int i = 0; i < temp.length; i++) {
            System.out.print(temp[i] + "\t");
        }
    }
}
```

程序运行结果：

```
67  69  75  87  89  90  90  100
-------------------------------------
1   8   9   17  18  30  31  39
```

从上面的程序可以发现，通过一条语句即可完成数组的排序功能，上面的调用语法暂时不理解也没有关系，读者只需要照原样写进去进行练习即可。此方法可以直接对各个基本数据类型的数组进行排序，包括浮点型、字符型等。

4.3.3　范例——数组复制

如果给定两个数组，将其中一个数组指定位置的内容复制给另外一个数组，可以使用方法来完成，在方法中接收 5 个参数，分别为“源数组名称”、“源数组开始点”、“目标数组名称”、“目标数组开始点”、“复制长度”，具体代码如下。

范例：数组复制操作

```
public class ArrayCopyDemo01 {
    public static void main(String args[]) {
        int i1[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };                    // 源数组
        int i2[] = { 11, 22, 33, 44, 55, 66, 77, 88, 99 }; // 目标数组
        copy(i1, 3, i2, 1, 3);                                        // 调用复制方法
        print(i2);                                                    // 输出数组
    }
    // 参数含义：源数组名称、源数组开始点、目标数组名称、目标数组开始点、复制长度
    public static void copy(int s[], int s1, int o[], int s2, int len) {
        for (int i = 0; i < len; i++) {
            o[s2 + i] = s[s1 + i];                                    // 修改目标数组
                                                                          内容
        }
    }
    public static void print(int temp[]) {                            // 数组输出
        for (int i = 0; i < temp.length; i++) {
            System.out.print(temp[i] + "\t");
        }
    }
}
```

程序运行结果：

```
11  4   5   6   55  66  77  88  99
```

从程序的运行结果可以看出，已经完成了数组的复制操作。对于此种代码在 Java 中也同样是存在类库支持的，直接使用 System.arraycopy()方法即可，此方法中也要接收参数，参数的接收顺序及意义与上面范例中的 copy 相同。

范例：使用 Java 类库中的方法完成数组复制操作

```
public class ArrayCopyDemo02 {
    public static void main(String args[]) {
        int i1[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };           // 源数组
        int i2[] = { 11, 22, 33, 44, 55, 66, 77, 88, 99 }; // 目标数组
        System.arraycopy(i1, 3, i2, 1, 3);                  // Java对数组复
                                                               制的支持
        print(i2);
    }
    public static void print(int temp[]) {                  // 输出数组
        for (int i = 0; i < temp.length; i++) {
            System.out.print(temp[i] + "\t");
        }
    }
}
```

程序运行结果：

```
11  4   5   6   55  66  77  88  99
```

从以上两个范例中可以发现，在 Java 中为开发者提供了各种各样的支持，随着读者开发经验的积累，了解的类库也会越来越多，开发的速度也就会越快。

提示：Java 中的类库。

在 Java 中提供了大量的类库，这些类库可以直接从 java doc 中查找到，但是在此处读者暂时没有必要研究 doc 文档，最好在学习完第 2 部分的内容后再进行深入研究。

4.4 Java 新特性对数组的支持

4.4.1 Java 新特性——可变参数

在调用一个方法时，必须根据方法的定义传递指定的参数，但是在 JDK 1.5（JAVA SE 5.0）之后产生了新的概念——可变参数，即方法中可以接收的参数不再是固定的，而是随着需要传递的，可变参数的定义格式如下：

【格式 4-9　可变参数的定义格式】

```
返回值类型 方法名称(类型...参数名称){}
```

向方法中传递可变参数之后，其中的参数是以数组的形式保存下来的。

范例：使用可变参数定义方法

```
public class NewDemo01 {
    public static void main(String[] args) {
        System.out.print("不传递参数（fun()）：");
```

```
        fun();                                          // 不传递参数
        System.out.print("\n传递1个参数（fun(1)）：");
        fun(1);                                         // 传递一个参数
        System.out.print("\n传递5个参数（fun(1, 2, 3, 4, 5)）：");
        fun(1, 2, 3, 4, 5);                             // 传递5个参数
    }
    public static void fun(int... arg) {    // 可变参数，可以接收任意多个参数
        for (int i = 0; i < arg.length; i++) {
            System.out.print(arg[i] + "、");
        }
    }
}
```

程序运行结果：

```
不传递参数（fun()）：
传递1个参数（fun(1)）：1、
传递5个参数（fun(1, 2, 3, 4, 5)）：1、2、3、4、5、
```

4.4.2　Java 新特性——foreach 输出

数组的输出一般都会使用 for 循环，但在 JDK 1.5 后为了方便数组的输出，提供了一种 foreach 语法，此语法的使用格式如下：

```
for(数据类型 变量名称 : 数组名称){
    ...
}
```

范例：使用 foreach 语法输出数组内容

```
public class NewDemo02 {
    public static void main(String[] args) {
        System.out.print("不传递参数（fun()）：");
        fun();                                          // 不传递参数
        System.out.print("\n传递1个参数（fun(1)）：");
        fun(1);                                         // 传递1个参数
        System.out.print("\n传递5个参数（fun(1, 2, 3, 4, 5)）：");
        fun(1, 2, 3, 4, 5);                             // 传递5个参数
    }
    public static void fun(int... arg) {    // 可变参数，可以接收任意多个参数
        for(int x : arg){                               // 使用foreach输出
            System.out.print(x + "、");
        }
    }
}
```

程序运行结果：

```
不传递参数（fun()）：
传递1个参数（fun(1)）：1、
传递5个参数（fun(1, 2, 3, 4, 5)）：1、2、3、4、5、
```

此程序的运行结果与前面的可变参数是一样的，在 Java 中有很多地方都可以使用 foreach 输出，随着本书的深入，读者可以了解更多可变参数和 foreach 的用法。

提示：尽量使用最标准的输出操作。

在 Java 中虽然提供了 foreach 语法，但是从实际的应用来看，还是使用最原始的输出操作最为合适，所以本书并不建议读者使用 foreach 输出。

4.5 本章要点

1．数组是由一组相同类型的变量所组成的数据类型，它们是以一个共同的名称来表示的。数组按存放元素的复杂程度分为一维、二维及多维数组。

2．使用 Java 中的数组必须经过声明数组和开辟内存给该数组两个步骤。声明数组时会在栈内存开辟空间，只开辟栈空间的数组是无法使用的，必须有指向的堆内存空间才能够使用，可以使用 new 关键字开辟堆内存空间，同时指定开辟的空间大小。

3．在 Java 中要取得数组的长度（也就是数组元素的个数），可以利用.length 来完成。

4．数组访问时要使用下标，如果下标的访问超过了数组的范围，则会出现数组越界异常。

5．Java 允许二维数组中每行的元素个数均不相同。

6．方法是一段可重复调用的代码段，在本章中因为方法可以由主方法直接调用，所以要加入 public static 关键字修饰。

7．方法的重载为方法名称相同，参数的类型或个数不同。

8．数组的传递属于引用数据类型的传递，传递的是堆内存地址的使用权，一个数组可以有多个名称指向同一个堆内存空间，每一个名称都可以修改堆内存中的内容。

9．Java 新特性中提供了可变参数，这样在传递参数时就可以不受参数的个数限制，全部的参数将以数组的形式保存下来。

10．foreach 是 Java 中的新特性，主要作用是方便地输出数组中的内容。

4.6 习　　题

1．编写程序求 1!+2!+…+30!的和并显示，要求使用方法完成。

2．定义一个由整数组成的数组，要求求出其中的奇数个数和偶数个数。

3．现在有如下的一个数组：

```
int oldArr[]={1,3,4,5,0,0,6,6,0,5,4,7,6,7,0,5} ;
```

要求将以上数组中值为 0 的项去掉，将不为 0 的值存入一个新的数组，生成的新数组为：

```
int newArr[]={1,3,4,5,6,6,5,4,7,6,7,5} ;
```

4．定义一个整型数组，求出数组元素的和、数组元素的最大值和最小值，并输出所求的结果。

5．给出 10 个整数（int 型），然后任意查询一个数字是否存在该 10 个数字内。

6．定义一个包含 10 个元素的数组，对其进行赋值，使每个元素的值等于其下标，然后输出；最后将这个数组倒置（即首尾交换）后输出。

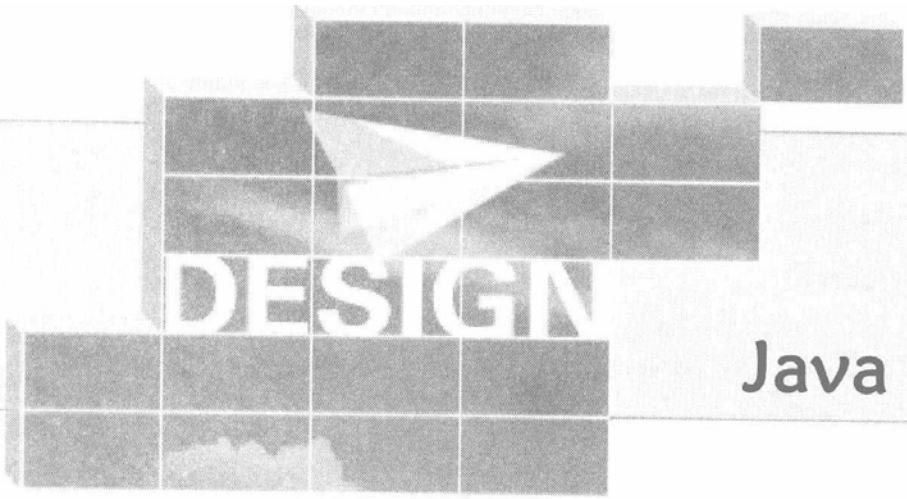
7．给出 10 个老师的打分，对 10 个老师的打分找到最高分。

8．有 30 个 0～9 之间的数字，分别统计 0～9 这 10 个数字分别出现了多少次。

9．定义一个整型数组，保存 10 个数据，利用程序完成将最大值保存在数组中第 1 个元素的操作。

10．在排序好的数组中添加一个数字，将添加后的数字插入到数组合适的位置。

第 2 部分



Java 面向对象程序设计

- 面向对象的概念
- 面向对象的封装性、继承性、多态性
- 抽象类与接口的应用
- 异常的捕获及处理
- 包及访问控制权限

第 5 章　面向对象（基础篇）

通过本章的学习可以达到以下目标：

- ☑ 了解面向对象的三大特征。
- ☑ 掌握类与对象的关系、定义及使用。
- ☑ 掌握对象的创建格式，可以创建多个对象。
- ☑ 掌握对象的内存分配过程。
- ☑ 掌握封装性的概念及使用。
- ☑ 掌握构造方法的定义格式、调用时机、构造方法的重载、构造方法的私有化意义。
- ☑ 掌握匿名对象的概念及应用。
- ☑ 可以通过所学到的知识完成简单类的开发。
- ☑ 掌握 String 类的特点及其主要的操作方法。
- ☑ 掌握 this 关键字的作用，并可以使用 this 关键字完成对象的比较操作。
- ☑ 掌握 static 关键字，并可以阐述出 static 属性及 static 方法的特点。
- ☑ 掌握主方法的组成及运行时的参数传递。
- ☑ 掌握 Java 中的普通代码块、构造块、静态块 3 种代码块的使用方法。
- ☑ 掌握内部类的特点及应用，以及内部类调用方法中参数的要求。

前面所学习的知识都只属于 Java 的基本程序设计范畴，属于结构化的程序开发，但是使用结构化方法开发的软件，其稳定性、可修改性和可重用性都比较差，这是因为结构化方法的本质是功能分解，是围绕实现处理功能的“过程”来构造系统的。然而在软件开发中大部分的用户需求是会随时改变的，所以对于使用结构化开发方式的设计是灾难性的。为了解决软件技术的多变性，并且可以很好地适应用户的变化，产生了面向对象技术，在本章将向读者介绍 Java 中的核心组成——类（class）。此外，为便于读者的理解，在本章中还加入了大量的程序分析，让读者清楚地知道每一个主要知识点的应用。**本章视频录像讲解时间为 6 小时 34 分钟，源代码在光盘对应的章节下。**

5.1　面向对象的基本概念

早期的程序经历了“面向问题”和“面向过程”的阶段，随着计算机的发展以及解决问题的复杂性的提高，以往的程序设计方法已经不能够适应现代的软件技术要求，于是，从 20 世纪 70 年代开始，相继出现了多种面向对象的程序设计语言，像 C++、Smalltalk、Java 等都是比较熟知的符合面向对象设计的语言。现在，面向对象的概念和应用已经超越了程序设计和软件开发，扩展到很宽的范围，如数据库系统、交互式界面、应用结构、应用平台、分布式系统、网络管理结构、CAD 技术、人工智能等领域。

提示：实例了解“面向过程”和“面向对象”的概念。

下面通过一个实例讲解“面向过程”和“面向对象”的关系。

例如，现在有两位师傅“面向过程”和“面向对象”要设计一个首饰盒：

（1）“面向过程”师傅：用户提出哪些要求，师傅就针对用户的要求进行制作，直接制作出一个完整的整体，本身也不准备好做首饰盒所需要的工具，而是需要什么再单独拿什么。

（2）“面向对象”师傅：针对用户提出的要求进行分析，并将分析的结果设计成一张完整的图纸，与需求的用户确认，然后将一切的准备工作全部处理完之后再分块制作，最后将各个小的部分组装在一起。

从以上两个师傅的做法可以发现，“面向对象”师傅要比“面向过程”师傅更能适应用户的变化，而一旦用户有变化之后，“面向过程”师傅基本上要推倒重做，而“面向对象”师傅却可以适应变化。

对于面向对象的程序设计有封装性、继承性、多态性 3 个主要特性。下面简单介绍这 3 种特性，在本书后面的内容中会对这 3 个方面进行完整的阐述。

1．封装性

封装是面向对象的方法所应遵循的一个重要原则，它有两个含义，一是指把对象的属性和行为看成一个密不可分的整体，将这两者“封装”在一个不可分割的独立单位（即对象）中；另一层含义指“信息隐蔽”，把不需要让外界知道的信息隐藏起来，有些对象的属性及行为允许外界用户知道或使用，但不允许更改，而另一些属性或行为，则不允许外界知道，或只允许使用对象的功能，而尽可能隐蔽对象的功能实现细节。

封装机制在程序设计中表现为，把描述对象属性的变量及实现对象功能的方法合在一起，定义为一个程序单位，并保证外界不能任意更改其内部的属性值，也不能任意调动其内部的功能方法。

封装机制的另一个特点是，为封装在一个整体内的变量及方法规定不同级别的“可见性”或访问权限。

2．继承性

继承是面向对象方法中的重要概念，并且是提高软件开发效率的重要手段。

首先拥有反映事物一般特性的类，然后在其基础上派生出反映特殊事物的类。如已有汽车的类，该类中描述了汽车的普遍属性和行为，进一步再产生轿车的类，轿车的类是继承于汽车类，轿车类不但拥有汽车类的全部属性和行为，还增加轿车特有的属性和行为。

在 Java 程序设计中，已有的类可以是 Java 开发环境所提供的一批最基本的程序——类库，用户开发的程序类是继承这些已有的类。这样，现在类所描述过的属性及行为，即已定义的变量和方法，在继承产生的类中完全可以使用。被继承的类称为父类或超类，而经继承产生的类称为子类或派生类。根据继承机制，派生类继承超类的所有成员，并相应地增加了自己的一些新的成员。

面向对象程序设计中的继承机制大大增强了程序代码的可复用性，提高了软件的开发效率，降低了程序产生错误的可能性，也为程序的修改扩充提供了便利。

若一个子类只允许继承一个父类，则称为单继承；若允许继承多个父类，则称为多继承。目前许多面向对象程序设计语言不支持多继承。而 Java 语言通过接口（interface）的方式来弥补由于 Java 不支持多继承而带来的子类不能享用多个父类的成员的缺点。

3. 多态性

多态是面向对象程序设计的又一个重要特征。多态是允许程序中出现重名现象。Java 语言中含有方法重载与对象多态两种形式的多态。

- 方法重载：在一个类中，允许多个方法使用同一个名字，但方法的参数不同，完成的功能也不同。
- 对象多态：子类对象可以与父类对象进行相互转换，而且根据其使用的子类的不同，完成的功能也不同。

多态的特性使程序的抽象程度和简捷程度更高，有助于程序设计人员对程序的分组协同开发。

5.2 类 与 对 象

5.2.1 类与对象的关系

在面向对象中，类和对象是最基本、最重要的组成单元。类实际上是表示一个客观世界某类群体的一些基本特征抽象。对象就是表示一个个具体的东西。例如，在现实生活中，人就可以表示为一个类，因为人本身属于一种广义的概念，并不是一个具体的。而某一个具体的人，就可以称为对象，可以通过各种信息完整地描述这个具体的人，如这个人的姓名、年龄、性别等信息，这些信息在面向对象的概念中就称为属性；当然人是可以吃饭、睡觉的，这些人的行为在类中就称为方法。也就是说如果要使用一个类，就一定有产生对象，每个对象之间是靠各个属性的不同来进行区分的，而每个对象所具备的操作就是类中规定好的方法。类与对象的关系如图 5-1 所示。

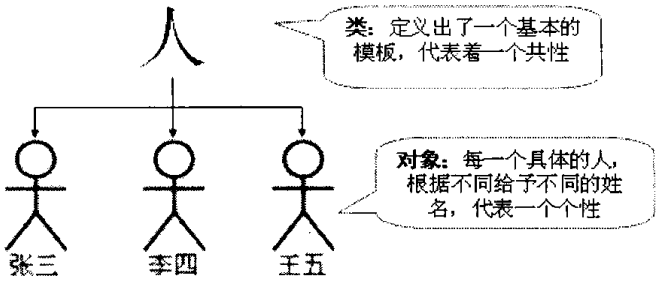


图 5-1 类与对象的关系

提示：类与对象的另一种解释。

关于类与对象，初学者在理解上是存在一定难度的，下面为各位读者作一个简单的比喻，读者应该都很清楚，如果要想生产出汽车，则首先一定要设计出一个汽车的设计图纸（如图 5-2 所示），然后按照此图纸规定的结构生产汽车。这样生产出的汽车结构和功能都是

一样的，但是每辆车的具体内容，如各个汽车颜色、是否有天窗等都会存在一些差异。在这个实例中，汽车设计图纸实际上就是规定出了汽车应该有的基本组成，包括外型、内部结构、发动机等信息的定义，那么这个图纸就可以称为一个类，显然只有图纸是无法使用的，而通过这个模型产生出的一辆辆的具体汽车是可以被用户使用的，所以就可以称其为对象。

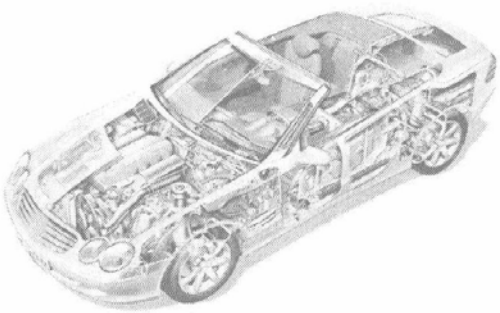


图 5-2　汽车设计图纸

5.2.2　类的定义

从类的概念中可以了解，类是由属性和方法组成的。属性中定义类一个个的具体信息，实际上一个属性就是一个变量，而方法是一些操作的行为，但是在程序设计中，定义类也是要按照具体的语法要求完成的，类的定义语法如下：

【格式 5-1　类的定义】

```
class 类名称{
    数据类型  属性 ;
    ...                                    } 声明成员变量（属性）

    public 返回值的数据类型 方法名称（参数1，参数2...）{
        程序语句 ;
        [return 表达式;]                    } 定义方法的内容
    }
}
```

下面根据以上的格式定义一个 Person 类。

提示：属性也可以称为变量。

在类中的属性实际上也就是相当于一个个变量，有时也称这些为 Field（成员）。

范例：定义 Person 类

```
class Person {
    String name;                    // 声明姓名属性
    int age;                        // 声明年龄属性
    public void tell() {            // 取得信息的方法
        System.out.println("姓名：" + name + "，年龄：" + age);
```

```
    }
}
```

在上面程序中，Person 类定义了 name 和 age 两个属性，分别表示人的姓名和年龄，然后定义了一个 tell 方法，此方法的功能就是打印这两个属性的内容。

提示：类中定义方法的补充说明。

读者可以发现，此处的方法与之前讲解的方法定义有些区别，并没有加上 static 关键字，这是因为此时定义的方法将要由对象调用，而不像之前那样与主方法定义在一个类中且由主方法直接调用。

类定义完成之后，也可以通过如图 5-3 所示的方法表示出类的定义。

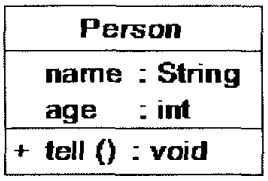


图 5-3 Person 的类图表示

图 5-3 所示的图形分为以下 3 个层次：

- 第 1 层表示类的名称，类的名称要求开头首字母大写。
- 第 2 层表示属性的定义，按照“访问权限 属性名称:属性类型”的格式定义，在本类中因为声明属性处没有写任何的访问权限，所以前面暂时不加任何的符号。
- 第 3 层表示类中方法的定义，按照“访问权限 方法名称():方法返回值”的格式定义，在本类中方法的声明处加上了 public（此为访问权限，表示任何地方都可以访问），所以使用“+”表示。另外，如果方法中有传递的参数，则方法定义格式为“访问权限 方法名称(参数名称:参数类型,参数名称:参数类型,...):方法返回值”。

在实际的开发中，以上的图形出现较多，本书会在后面章节介绍各种常见图形的画法。

5.2.3 对象的创建及使用

在上面已经创建好了一个 Person 类，要想使用一个类则必须有对象。下面给出了对象的创建格式：

【格式 5-2 对象的创建】

```
类名 对象名称 = null ;          // 声明对象
对象名称 = new 类名() ;          // 实例化对象
```

用以上格式产生对象分为声明对象和实例化对象两步。

当然也可以直接通过以下方式一步完成：

```
类名 对象名称 = new 类名() ;
```

细心的读者可以发现，以上格式与之前数组定义的格式相似，因为类和数组都属于引用数据类型，只要是引用数据类型的使用格式都可以使用如上的定义样式。

了解了上述的格式之后，下面就来看一下创建对象的具体范例。

```
class Person {
    String name;                                    // 声明姓名属性
    int age;                                        // 声明年龄属性
```

```
    public void tell() {                          // 取得信息的方法
        System.out.println("姓名：" + name + "，年龄：" + age);
    }
}
public class ClassDemo02 {
    public static void main(String args[]){
        Person per = new Person() ;               // 创建并实例化对象
    }
}
```

以上程序在主方法中实例化了一个 Person 对象，对象名称为 per。与前面的数组开辟空间一样，对象的实例化也是要划分堆、栈空间的，具体的内存分配如图 5-4 所示。

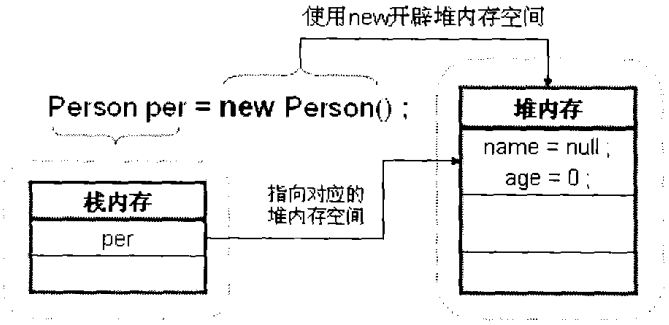


图 5-4　对象的实例化过程

从图 5-4 中可以发现，所有的对象名称都在栈内存中保存，而对象的具体内容则保存在对应的堆内存之中，必须使用 new 关键字才能开辟堆内存空间，此时，因为 per 对象刚刚被实例化完，所以对象中的属性内容都是默认值，字符串的默认值为 null，整数的默认值为 0。

提示：栈中存放的是堆空间的地址。

在本书中对于引用关系，为了让读者理解起来更加容易，所以在讲解时将对象名称保存在栈内存之中，实际上更准确的说法是在栈内存中实际上保存的是堆内存空间的访问地址。

如果要使用对象访问类中的某个属性或方法可以使用如下的语法实现：

【格式 5-3　访问对象中的属性或方法】

访问属性：对象名称.属性名

访问方法：对象名称.方法名()

范例：为对象的属性设置内容，同时调用 tell 方法把内容输出

```
class Person {
    String name;                                  // 声明姓名属性
    int age;                                      // 声明年龄属性
    public void tell() {                          // 取得信息的方法
        System.out.println("姓名：" + name + "，年龄：" + age);
    }
}
```

```
public class ClassDemo03 {
    public static void main(String args[]){
        Person per = null ;
        per.name = "张三" ;                    // 为name属性赋值
        per.age = 30 ;                         // 为age属性赋值
        per.tell() ;                           // 调用类中的方法
    }
}
```

程序运行结果：

```
姓名：张三，年龄：30
```

上面程序为属性赋值，并通过类中提供的方法把内容直接输出，属性赋值完之后的内存如图 5-5 所示。

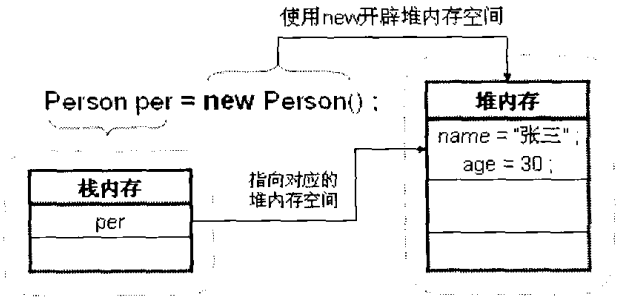


图 5-5　为对象中的属性赋值后的内存

注意：对象使用前必须实例化。

必须引起读者注意的是，如果一个对象要被使用，则对象必须被实例化（本书后面的部分将直接把已经实例化好的对象称为实例化对象），如果一个对象没有被实例化而直接调用了对象中的属性或方法，如下代码所示：

```
Person per = null ;
per.name = "张三" ;
per.age = 30 ;
per.tell() ;
```

则程序运行时会出现以下异常：

```
Exception in thread "main" java.lang.NullPointerException
```

此异常是开发中最常见的异常，也会始终伴随着每位开发人员。使用了未实例化的对象则肯定会出现此异常。

5.2.4　创建多个对象

前面已经介绍过一个对象的创建方法，那么按照此格式可以同时创建多个对象，每个对象会分别占据自己的堆、栈空间。

范例：创建两个对象

```
class Person {
    String name;                                    // 声明姓名属性
    int age;                                        // 声明年龄属性
    public void tell() {                            // 取得信息的方法
        System.out.println("姓名：" + name + "，年龄：" + age);
    }
}
public class ClassDemo04 {
    public static void main(String args[]) {
        Person per1 = null;                         // 声明per1对象
        Person per2 = null;                         // 声明per2对象
        per1 = new Person();                        // 实例化per1对象
        per2 = new Person();                        // 实例化per2对象
        per1.name = "张三";                         // 设置per1对象的name属性内容
        per1.age = 30;                              // 设置per1对象的age属性内容
        per2.name = "李四";                         // 设置per2对象的name属性内容
        per2.age = 33;                              // 设置per2对象的age属性内容
        System.out.print("per1对象中的内容 --> ") ;
        per1.tell();                                // per1调用方法
        System.out.print("per2对象中的内容 --> ") ;
        per2.tell();                                // per2调用方法
    }
}
```

程序运行结果：

```
per1对象中的内容 --> 姓名：张三，年龄：30
per2对象中的内容 --> 姓名：李四，年龄：33
```

由程序的运行结果可以发现，程序分别实例化了两个 Person 对象，那么也就意味着 per1 和 per2 对象分别指向各自的堆内存空间，如图 5-6 所示。

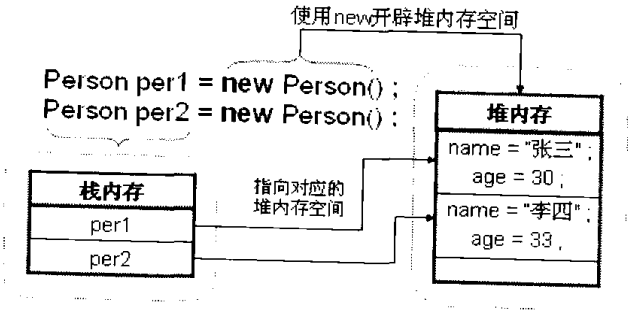


图 5-6　实例化两个对象

类属于引用数据类型，而且从数组的使用上也可以发现，引用数据类型就是指一段堆内存空间可以同时被多个栈内存指向。下面来看一个引用传递的简单范例。

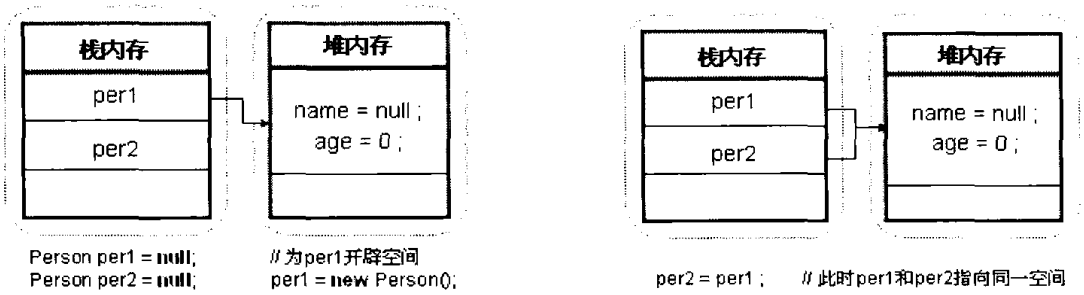
范例：对象引用传递

```
class Person {
    String name;                                    // 声明姓名属性
    int age;                                        // 声明年龄属性
    public void tell() {                            // 取得信息的方法
        System.out.println("姓名：" + name + "，年龄：" + age);
    }
}
public class ClassDemo05 {
    public static void main(String args[]) {
        Person per1 = null;                         // 声明per1对象
        Person per2 = null;                         // 声明per2对象
        per1 = new Person();                        // 只实例化per1一个对象
        per2 = per1 ;                               // 把per1的堆内存空间使用权给per2
        per1.name = "张三";                          // 设置per1对象的name属性内容
        per1.age = 30;                              // 设置per1对象的age属性内容
        // 设置per2对象的内容，实际上就是设置per1对象的内容
        per2.age = 33;
        System.out.print("per1对象中的内容 --> ") ;
        per1.tell();                                // 调用类中的方法
        System.out.print("per2对象中的内容 --> ") ;
        per2.tell();
    }
}
```

程序运行结果：

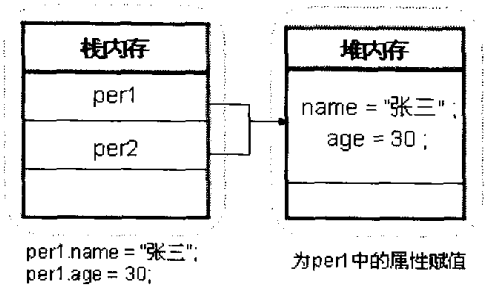
```
per1对象中的内容 --> 姓名：张三，年龄：33
per2对象中的内容 --> 姓名：张三，年龄：33
```

从程序运行结果可以发现，两个对象的输出内容是一样的，实际上所谓的引用传递，就是将一个堆内存空间的使用权给多个栈内存空间，每个栈内存空间都可以修改堆内存的内容，此程序的内存图如图 5-7 所示。

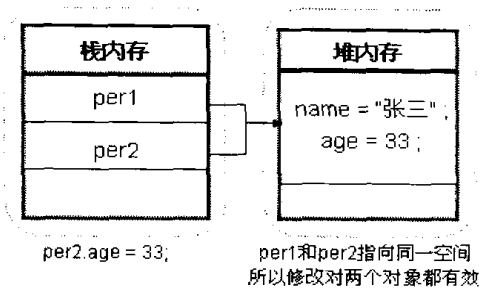


（a）声明对象并为 per1 开辟空间　　（b）per1 和 per2 指向同一个堆内存

图 5-7　对象引用传递的内存分配图



（c）为 per1 中的属性赋值



（d）通过 per2 修改 per1 中的 age 属性

图 5-7　对象引用传递的内存分配图（续）

从图 5-7 中可以发现内存空间的堆、栈指向的变化，程序最后由 per2 将 age 的值修改为 33，所以最终结果 per1 中的 age 属性值也就是 33。

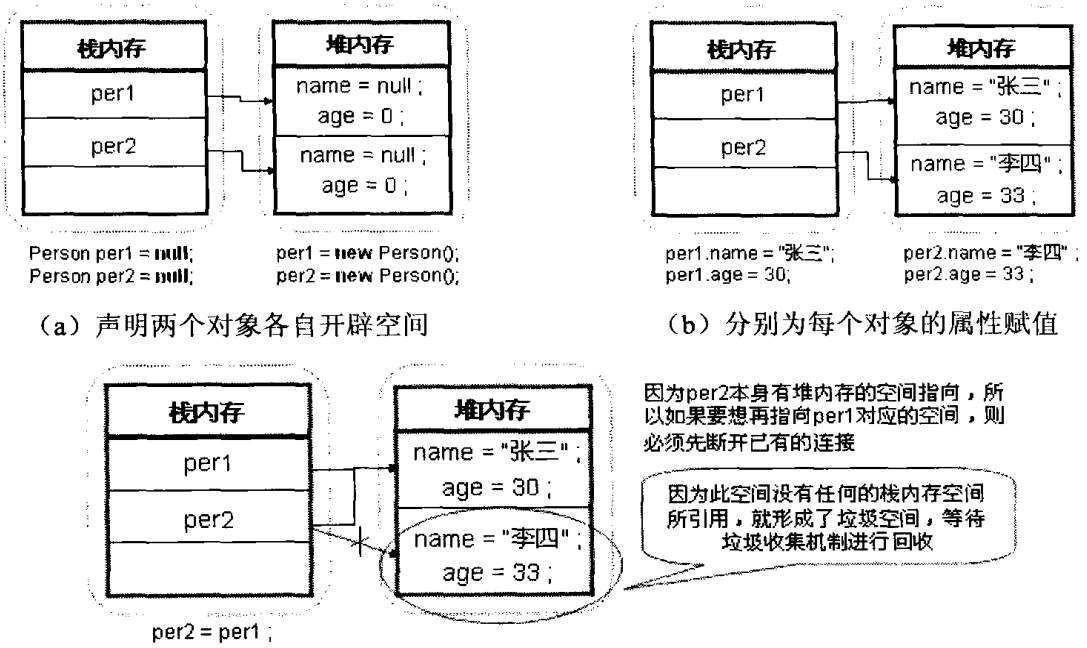
范例：对象引用传递

```
class Person {
    String name;                                    // 声明姓名属性
    int age;                                        // 声明年龄属性
    public void tell() {                            // 取得信息的方法
        System.out.println("姓名: " + name + ", 年龄: " + age);
    }
}
public class ClassDemo06 {
    public static void main(String args[]) {
        Person per1 = null;                         // 声明一个per1对象
        Person per2 = null;                         // 声明一个per2对象
        per1 = new Person();                        // 实例化per1对象
        per2 = new Person() ;                       // 实例化per2对象
        per1.name = "张三";                          // 设置per1的name属性内容
        per1.age = 30;                              // 设置per1的age属性内容
        per2.name = "李四" ;                         // 设置per2的name属性内容
        per2.age = 33 ;                             // 设置per2的age属性内容
        per2 = per1 ;                               // 将per1的引用传递给per2
        System.out.print("per1对象中的内容 --> ") ;
        per1.tell();                                // 调用类中的方法
        System.out.print("per2对象中的内容 --> ") ;
        per2.tell();
    }
}
```

程序运行结果：

```
per1对象中的内容 --> 姓名: 张三, 年龄: 30
per2对象中的内容 --> 姓名: 张三, 年龄: 30
```

从程序的输出结果应该不难明白本范例内存发生的变化，具体如图 5-8 所示。



（a）声明两个对象各自开辟空间　　（b）分别为每个对象的属性赋值

（c）断开已有的堆内存指向，重新指向新的堆内存空间

图 5-8　垃圾对象的产生

提示：关于垃圾空间的释放。

Java 本身提供垃圾收集机制（Garbage Collection，GC），会不定期地释放不用的内存空间，只要对象不使用了，就会等待 GC 释放空间。

从上面程序中必须明确的一点，即一个栈内存空间只能指向一个堆内存空间，如果要想再指向其他的堆内存空间，则必须先断开已有的指向才能分配新的指向。

5.3　封　装　性

封装性是面向对象的第一大特性，所谓的封装性就是指对外部不可见，那么为什么要有封装性呢？首先观察以下的程序。

范例：观察以下程序的问题

```
class Person {
    String name;                                    // 声明姓名属性
    int age;                                        // 声明年龄属性
    public void tell() {                            // 取得信息的方法
        System.out.println("姓名：" + name + "，年龄：" + age);
    }
}
public class EncDemo01 {
    public static void main(String args[]) {
```

```
        Person per = new Person();                    // 声明并实例化对象
        per.name = "张三";                            // 为name属性赋值
        per.age = -30;                                // 为age属性赋值
        per.tell();                                   // 调用方法
    }
}
```

程序运行结果：

```
姓名：张三，年龄：-30
```

由上面的程序可以发现，在程序中将年龄（age）赋值为-30 岁，这明显是一个不合法的数据，所以最终程序在调用 tell()方法时才会打印出这种错误的信息。这就好比要加工一件产品一样，本身加工的原料就有问题，那么最终加工出来的产品也一定是一个不合格的产品。而导致这种错误的原因就是因为程序在原料的入口处并没有加以检验，而加工的原料原本就是变质的，这样加工出来的产品也必然是一个不合要求的产品。

读者可以发现，之前所列举的程序都是用对象直接访问类中的属性，这在面向对象法则中是不允许的。所以为了避免程序中这种错误的发生，在一般的开发中往往要将类中的属性封装（private），封装的格式如下：

【格式 5-4　使用封装性】

为属性封装： private 属性类型 属性名称 ;

为方法封装： **private** 方法返回值 方法名称(参数列表){}

范例：为程序加上封装属性

```
class Person {
    private String name;                              // 声明姓名属性
    private int age;                                  // 声明年龄属性
    public void tell() {                              // 取得信息的方法
        System.out.println("姓名：" + name + "，年龄：" + age);
    }
}
public class EncDemo02 {
    public static void main(String args[]) {
        Person per = new Person();
        per.name = "张三";                            // 错误，无法访问封装属性
        per.age = -30;                                // 错误，无法访问封装属性
        per.tell();
    }
}
```

程序运行结果：

```
EncDemo02.java:11: name has private access in Person
            per.name = "张三";                        // 错误，无法访问封装属性
```

```
                    ^
EncDemo02.java:12: age has private access in Person
                per.age = -30;                          // 错误，无法访问封装属性
                    ^
2 errors
```

本程序与上面的范例除了在声明属性上有些区别之外，并没有其他的区别，而就是这一个小小的关键字 private，使程序连编译都无法通过，而所提示的错误为“属性（name、age）为私有的”，所以不能由对象直接进行访问，这样就可以保证对象无法直接去访问类中的属性，但是这样一来该如何访问此属性呢？为了解决属性必须封装且又必须访问的矛盾，在 Java 开发中对于私有属性的访问有了以下的明确定义：“只要是被封装的属性，则必须通过 setter 和 getter 方法设置和取得”。

提示：进一步深入。

对于私有属性要使用 setter 和 getter 方法设置和访问实际上还有一个原因，就是在 Java 中存在反射机制，在各个程序中，反射机制都会利用 setter 和 getter 设置和取得属性内容，关于反射机制可以参考第 3 部分的内容。

范例：为前面类中的私有属性加上 setter 和 getter 方法

```
class Person {
    private String name;                        // 声明姓名属性
    private int age;                            // 声明年龄属性
    public void tell() {                        // 取得信息的方法
        System.out.println("姓名：" + getName() + "，年龄：" + getAge());
    }
    public String getName() {                   // 取得姓名
        return name;
    }
    public void setName(String n) {             // 设置姓名
        name = n;
    }
    public int getAge() {                       // 取得年龄
        return age;
    }
    public void setAge(int a) {                 // 设置年龄
        age = a;
    }
}
public class EncDemo03 {
    public static void main(String args[]) {
        Person per = new Person();              // 声明并实例化对象
        per.setName("张三");                    // 调用setter设置姓名
```

```
        per.setAge(-30);                    // 调用setter设置年龄
        per.tell();                         // 输出信息
    }
}
```

程序运行结果：

```
姓名：张三，年龄：-30
```

可以先暂时不考虑上面程序的运行结果，先观察程序的结构，在程序中可以发现通过setter 和 getter 方法可以设置和取得属性，而在主方法调用时，也是调用了 setter()方法进行内容的赋值，也就是说如果想对设置进去的值进行检查，则只需要在 setter 方法处检查即可。

范例：在 setter 方法处加上检测代码

```
class Person {
    private String name;                    // 声明姓名属性
    private int age;                        // 声明年龄属性
    public void tell() {                    // 取得信息的方法
        System.out.println("姓名：" + getName() + "，年龄：" + getAge());
    }
    public String getName() {               // 取得姓名
        return name;
    }
    public void setName(String n) {         // 设置姓名
        name = n;
    }
    public int getAge() {                   // 取得年龄
        return age;
    }
    public void setAge(int a) {             // 设置年龄
        if (a >= 0 && a < 150) {            // 在此处加上验证代码
            age = a;
        }
    }
}
public class EncDemo04 {
    public static void main(String args[]) {
        Person per = new Person();          // 声明并实例化对象
        per.setName("张三");                // 调用setter设置姓名
        per.setAge(-30);                    // 调用setter设置年龄
        per.tell();                         // 输出信息
    }
}
```

程序运行结果：

姓名：张三，年龄：0

从程序运行结果可以发现，因为程序中在 setter 方法处加入了验证代码，所以如果设置的年龄数值不正确，则不会把值赋给 age 属性，所以程序运行结果处出现的年龄为 0。

提示：关于 private 的补充说明。

（1）在以后的开发中读者一定要明确，类中全部属性都必须封装，封装之后的属性必须通过 setter 和 getter 进行访问。

（2）面向对象的封装性本身并不是单单指 private 关键字，为了让读者可以更快地理解封装性，本章只是暂时将封装性的概念进行简单的讲解，读者必须记住的是，用 private 声明的属性或方法只能在其类的内部被调用，而不能在类的外部被调用。

（3）读者可以发现类中已经有很多的方法，正常情况下，类中的方法直接写上方法名称就可以完成本类中的方法调用，如果在此时非要强调是本类中的方法，也可以在调用时按“this.方法名称()”的形式编写：

```
public void tell() {
  System.out.println("姓名：" + this.getName() + "，年龄：" + this.getAge());
}
```

在代码中是否使用 this 明确地表示当前类中的方法并没有严格的要求，但是这里建议读者在编写代码时最好采用“this.方法名称()”的形式，这样会比较标准一些，在查错时也会更加方便。

程序中的属性进行封装后，在使用类图表示封装属性时就必须按照如下的风格：

-属性名称:数据类型

则上面程序的类图如图 5-9 所示。

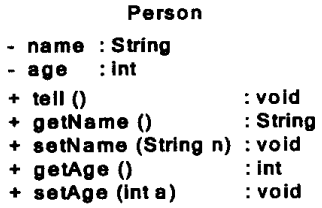


图 5-9　类图表示

提示：“-”表示 private。

在图 5-9 中，可以发现 name 属性前加上了“-”，实际上表示的是 private。

5.4　构 造 方 法

从前面所讲解的代码可以发现，实例化一个类的对象后，如果要为这个对象中的属性赋值，则必须用 setter 方法，那么有没有一种简单的方法，可以在对象实例化时就直接把对

象的值赋给属性呢？此时，就可以通过构造方法完成这样的操作，在面向对象程序中构造方法的主要作用是为类中的属性初始化。

可以回顾对象的产生格式，例如，现在要产生一个类的对象，则必须使用以下格式的代码：

```
类名称 对象名称 = new 类名称()
```

可以发现，在程序中只要有“()”就表示调用了方法，那么这个方法实际上就是表示要调用构造方法，构造方法可视为一种特殊的方法，它的定义方式与普通方法类似，其语法如下。

【格式 5-5　构造方法定义格式】

```
class 类名称{
    访问权限 类名称（类型1 参数1，类型2 参数2，…）{
        程序语句 ;
        ...    // 构造方法没有返回值
    }
}
```

在构造方法的声明中读者一定要牢记以下几点：

- 构造方法的名称必须与类名称一致。
- 构造方法的声明处不能有任何返回值类型的声明。
- 不能在构造方法中使用 return 返回一个值。

提示：关于访问权限。

在这里需要说明的是，对于访问权限，现在实际上本书只使用了 3 种，即 default（默认，什么都不写）、private、public，这 3 种都可以使用，为了以后的代码开发方便，本段内容暂时使用 public 访问权限，具体的区别在后面章节会为读者说明。

范例：声明一个构造方法

```
class Person {
    public Person(){                          // 声明构造方法
        System.out.println("一个新的Person对象产生。") ;
    }
}
public class ConsDemo01 {
    public static void main(String args[]) {
        System.out.println("声明对象：Person per = null ;") ;
        Person per = null ;                   // 声明对象时不调用构造
        System.out.println("实例化对象：per = new Person() ;") ;
        per = new Person();                   // 实例化对象时调用构造
    }
}
```

程序运行结果：

```
声明对象：Person per = null ;
实例化对象：per = new Person() ;
一个新的Person对象产生。
```

程序实例化对象的过程与之前是一样的，在类中定义一个 Person 类的构造方法，但是从程序的运行结果可以发现，当调用 new 关键字实例化对象时才会调用构造方法。

有些读者会觉得很奇怪，前面的类中并没有写构造方法，但是也可以调用，这是为什么呢？首先必须说明的是，只要是类就必须存在构造方法，在 Java 中如果一个类中没有明确地声明一个构造方法，则在编译时会直接生成一个无参数的、什么都不做的构造方法，也就是说，如果以上的 Person 类中没有明确地声明构造方法，实际上编译之后的类就会为用户自动加上以下形式的构造方法：

```
class Person {
    public Person(){}
}
```

正因为如此，所以前面的程序中即使没有构造方法，对象也是可以进行对象的实例化操作的。对于构造方法类图没有特别明确的要求，以上的程序直接使用以下的类图形式表示即可，如图 5-10 所示。

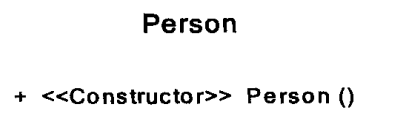


图 5-10　构造方法的类图表示

下面范例为通过构造方法为类中的属性赋值。

范例：通过构造方法为属性赋值

```
class Person {
    private String name;                          // 声明姓名属性
    private int age;                              // 声明年龄属性
    public Person(String name,int age){           // 定义构造方法为属性初始化
        this.setName(name) ;                      // 为name属性赋值
        this.setAge(age) ;                        // 为age属性赋值
    }
    public void tell() {                          // 取得信息的方法
        System.out.println("姓名：" + getName() + "，年龄：" + getAge());
    }
    public String getName() {                     // 取得姓名
        return name;
    }
    public void setName(String n) {               // 设置姓名
        name = n;
    }
```

```
    public int getAge() {                          // 取得年龄
        return age;
    }
    public void setAge(int a) {                    // 设置年龄
        if (a >= 0 && a < 150) {                   // 在此处加上验证代码
            age = a;
        }
    }
}
public class ConsDemo02 {
    public static void main(String args[]) {
        Person per = new Person("张三",30); // 调用构造方法，传递两个参数
        per.tell();                                // 输出信息
    }
}
```

程序运行结果：

```
姓名：张三，年龄：30
```

以上程序就是直接通过构造方法进行赋值，可以发现，这样赋值比对象实例化之后再单独调用 setter 方法更方便。

注意：关于自动生成构造方法的说明。

要提醒读者的是，在一个类中如果已经明确地声明了一个构造方法，那么程序在编译时将不会再生成默认的构造方法，即一个类中应保证至少有一个构造方法。

与普通方法一样，构造方法也是可以重载的，只要每个构造方法的参数类型或参数个数不同，即可实现重载。

范例：构造方法重载

```
class Person {
    private String name;                           // 声明姓名属性
    private int age;                               // 声明年龄属性
    public Person(){}                              // 定义无参构造
    public Person(String name){                    // 定义构造，为name属性赋值
        this.setName(name) ;
    }
    public Person(String name,int age){            // 定义构造方法为属性初始化
        this.setName(name) ;                       // 为name属性赋值
        this.setAge(age) ;                         // 为age属性赋值
    }
    public void tell() {                           // 取得信息的方法
        System.out.println("姓名：" + getName() + "，年龄：" + getAge());
    }
```

```
    public String getName() {                    // 取得姓名
        return name;
    }
    public void setName(String n) {              // 设置姓名
        name = n;
    }
    public int getAge() {                        // 取得年龄
        return age;
    }
    public void setAge(int a) {                  // 设置年龄
        if (a >= 0 && a < 150) {                 // 在此处加上验证代码
            age = a;
        }
    }
}
public class ConsDemo03 {
    public static void main(String args[]) {
        Person per = new Person("张三");         // 调用有一个参数的构造
        per.tell();                              // 输出信息
    }
}
```

程序运行结果：

```
姓名：张三，年龄：0
```

以上类的构造方法被重载了 3 次，在主方法调用的是只有一个参数的构造方法（只设置姓名），因为没有设置年龄，所以年龄为默认值 0。

5.5 匿 名 对 象

匿名对象就是没有明确给出名字的对象。一般匿名对象只使用一次，而且匿名对象只在堆内存中开辟空间，而不存在栈内存的引用。

范例：匿名对象的使用

```
class Person {
    private String name;                         // 声明姓名属性
    private int age;                             // 声明年龄属性
    public Person(String name,int age){          // 定义构造方法，为属性初始化
        this.setName(name) ;                     // 为name属性赋值
        this.setAge(age) ;                       // 为age属性赋值
    }
    public void tell() {                         // 取得信息的方法
        System.out.println("姓名：" + getName() + "，年龄：" + getAge());
```

```
    }
    public String getName() {                    // 取得姓名
        return name;
    }
    public void setName(String n) {              // 设置姓名
        name = n;
    }
    public int getAge() {                        // 取得年龄
        return age;
    }
    public void setAge(int a) {                  // 设置年龄
        if (a >= 0 && a < 150) {                 // 在此处加上验证代码
            age = a;
        }
    }
}
public class NonameDemo01 {
    public static void main(String args[]) {
        new Person("张三", 30).tell();           // 匿名对象
    }
}
```

程序运行结果：

```
姓名：张三，年龄：30
```

在以上程序的主方法中可以发现，直接使用了“new Person("张三", 30)”语句，这实际上就是一个匿名对象，与之前声明的对象不同，此处没有任何栈内存引用它，所以此对象使用一次之后就等待被垃圾收集机制回收。

提示：匿名对象的作用。

匿名对象在实际开发中基本上都是作为其他类实例化对象的参数传递的，在后面的 Java 应用部分的很多地方都可以发现其用法。而且细心的读者可以发现，匿名对象实际上就是一个堆内存空间，对象不管是匿名还是非匿名的，都必须在开辟堆内存空间之后才可以使用。

5.6　实例讲解——类设计分析

学习了以上的知识之后，下面来分析一道程序，以巩固之前所学知识。在具体题目讲解之前先给出一些分析的思路：

（1）根据要求写出类所包含的属性。

（2）所有的属性都必须进行封装（private）。

（3）封装之后的属性通过 setter 和 getter 设置和取得。

（4）如果需要可以加入若干构造方法。

（5）再根据其他要求添加相应的方法。

（6）类中的所有方法都不要直接输出，而是交给被调用处输出。

题目：

定义并测试一个名为 Student 的类，包括的属性有“学号”、“姓名”以及 3 门课程“数学”、“英语”和“计算机”的成绩，包括的方法有计算 3 门课程的“总分”、“平均分”、“最高分”及“最低分”。

1．本类中的属性及类型，如表 5-1 所示

表 5-1　Student 类中的属性及类型

序　号	属　性	属 性 类 型	属 性 名 称
1	学号	String	stuno
2	姓名	String	name
3	数学成绩	float	math
4	英语成绩	float	english
5	计算机成绩	float	computer

2．定义出需要的方法（普通方法、构造方法）

在本例中设计两个构造方法，一个是无参的构造方法，另外一个构造方法可以为 5 个属性进行赋值，如表 5-2 所示。

表 5-2　需要的方法

序　号	方 法 名 称	返回值类型	作　用
1	public void setStuno(String s)	void	设置学生编号
2	public void setName(String n)	void	设置学生姓名
3	public void setMath(float m)	void	设置数学成绩
4	public void setEnglish(float e)	void	设置英语成绩
5	public void setComputer(float c)	void	设置计算机成绩
6	public String getStuno()	String	取得学生编号
7	public String getName()	String	取得学生姓名
8	public float getMath()	float	取得数学成绩
9	public float getEnglish()	float	取得英语成绩
10	public float getComputer()	float	取得计算机成绩
11	public float sum()	float	计算成绩总和
12	public float avg()	float	计算平均成绩
13	public float max()	float	求出最高成绩
14	public float min()	float	求出最低成绩
15	public Student(){}	-	无参构造方法
16	public Student(String stuno,String name, float math,float english,float computer)	-	在对象实例化时直接将学号、姓名、数学成绩、英语成绩、计算机成绩设置进去

根据以上设置，本范例的类图如图 5-11 所示。

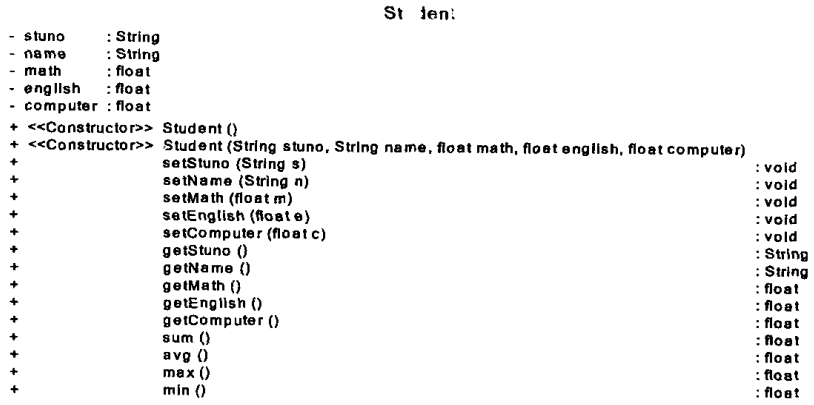


图 5-11　程序分析类图

按照以上类图，编写具体的代码如下。

范例：实现代码

```
class Student {                                          // 定义学生类
    private String stuno;                                // 学生编号
    private String name;                                 // 学生姓名
    private float math;                                  // 数学成绩
    private float english;                               // 英语成绩
    private float computer;                              // 计算机成绩
    public Student() {                                   // 定义无参构造
    }
    // 定义有5个参数的构造方法，为类中的属性初始化
    public Student(String stuno, String name, float math, float english,
            float computer) {
        this.setStuno(stuno);                            // 设置编号
        this.setName(name);                              // 设置姓名
        this.setMath(math);                              // 设置数学成绩
        this.setEnglish(english);                        // 设置英语成绩
        this.setComputer(computer);                      // 设置计算机成绩
    }
    public void setStuno(String s) {                     // 设置编号
        stuno = s;
    }
    public void setName(String n) {                      // 设置姓名
        name = n;
    }
    public void setMath(float m) {                       // 设置数学成绩
        math = m;
    }
    public void setEnglish(float e) {                    // 设置英语成绩
        english = e;
```

```
    }
    public void setComputer(float c) {                    // 设置计算机成绩
        computer = c;
    }
    public String getStuno() {                            // 取得编号
        return stuno;
    }
    public String getName() {                             // 取得姓名
        return name;
    }
    public float getMath() {                              // 取得数学成绩
        return math;
    }
    public float getEnglish() {                           // 取得英语成绩
        return english;
    }
    public float getComputer() {                          // 取得计算机成绩
        return computer;
    }
    public float sum() {                                  // 计算总分
        return math + english + computer;
    }
    public float avg() {                                  // 计算平均分
        return this.sum() / 3;                            // 总分除以3
    }
    public float max() {                                  // 最高成绩
        float max = math;
        max = max > computer ? max : computer;            // 使用三目运算符
        max = max > english ? max : english;              // 使用三目运算符
        return max;
    }
    public float min() {                                  // 最低成绩
        float min = math;
        min = min < computer ? min : computer;            // 使用三目运算符
        min = min < english ? min : english;              // 使用三目运算符
        return min;
    }
};
```

编写测试类，测试以上代码：

```
public class ExampleDemo01 {
    public static void main(String args[]){
        Student stu = null ;                              // 声明对象
```

```
        // 实例化Student对象，并通过构造方法赋值
        stu = new Student("MLDN-33", "李兴华", 95.0f, 89.0f, 96.0f);
        System.out.println("学生编号：" + stu.getStuno());
        System.out.println("学生姓名：" + stu.getName());
        System.out.println("数学成绩：" + stu.getMath());
        System.out.println("英语成绩：" + stu.getEnglish());
        System.out.println("计算机成绩：" + stu.getComputer());
        System.out.println("最高分：" + stu.max());
        System.out.println("最低分：" + stu.min());
    }
}
```

程序运行结果：

```
学生编号：MLDN-33
学生姓名：李兴华
数学成绩：95.0
英语成绩：89.0
计算机成绩：96.0
最高分：96.0
最低分：89.0
```

以上程序只是为读者简单地介绍了类的基本分析思路，实际问题肯定会比本道例题要复杂得多，此时，就需要读者耐心分析，只有掌握好面向对象中的各个概念，才可以对程序代码进行更加合理的分析与设计。

提示：先从最基本的类开始分析。

读者可以试着按照以上程序的分析思路分析一下身边的事物，例如，电脑、手机等，以加深对类的认识。

5.7 String

本书一直使用 String 声明一个字符串，相信读者对这个类型应该不是很陌生，但是从前面的要求来看，读者应该发现，String 声明时单词的首字母大写，所以 String 本身是一个类本类，但是此类在使用时却有很多的要求，而且此类在 Java 中也算是一个比较特殊的类，那么本节将为读者讲解 String 类的作用以及使用上的一些限制。

5.7.1 实例化 String 对象

对于 String 可以采用直接赋值的方式进行操作，如下面的代码。

范例：直接为 String 赋值

```
public class StringDemo01 {
    public static void main(String[] args) {
```

```
        String name = "LiXingHua";                    // 实例化String对象
        System.out.println("姓名: " + name);          // 输出字符串的内容
    }
}
```

程序运行结果：

```
姓名: LiXingHua
```

在 String 的使用上还有另外一种形式的实例化方法，就是直接调用 String 类中的构造方法，在 String 类存在以下的构造方法：

```
public String(String original)
```

所以上面的范例，也可以通过如下的代码进行编写。

范例：使用第 2 种 String 的实例化方式

```
public class StringDemo02 {
    public static void main(String[] args) {
        String name = new String("LiXingHua");          // 实例化String对象
        System.out.println("姓名: " + name);            // 输出字符串的内容
    }
}
```

程序运行结果：

```
姓名: LiXingHua
```

可以发现上面两段代码的使用效果类似，两种方法都可以使用，具体的区别在后面将有所讲解。

5.7.2 String 的内容比较

对于基本数据类型可以通过“==”进行内容的比较，如下面代码。

范例：使用“==”进行内容的比较

```
public class StringDemo03 {
    public static void main(String[] args) {
        int x = 30;                                     // 声明一个整型变量
        int y = 30;                                     // 声明一个整型变量
        System.out.println("两个数字的比较结果: " + (x == y));
    }
}
```

程序运行结果：

```
两个数字的比较结果: true
```

可以发现，两个数字的比较结果相等，下面为按照以上的程序思路比较两个字符串的代码。

范例：使用“==”比较字符串的内容

```
public class StringDemo04 {
  public static void main(String[] args) {
    String str1 = "hello";                                          // 直接赋值
    String str2 = new String("hello");                              // 通过new赋值
    String str3 = str2;                                             // 传递引用
    System.out.println("str1 == str2 --> " + (str1 == str2)); // --> false
    System.out.println("str1 == str3 --> " + (str1 == str3)); // --> false
    System.out.println("str2 == str3 --> " + (str2 == str3));// --> true
  }
}
```

程序运行结果：

```
str1 == str2 --> false
str1 == str3 --> false
str2 == str3 --> true
```

从程序运行结果中可以发现，虽然以上程序中 String 的内容都一样，但是比较的结果却是有的相等，有的不相等，这是为什么呢？下面通过图 5-12 为读者说明。

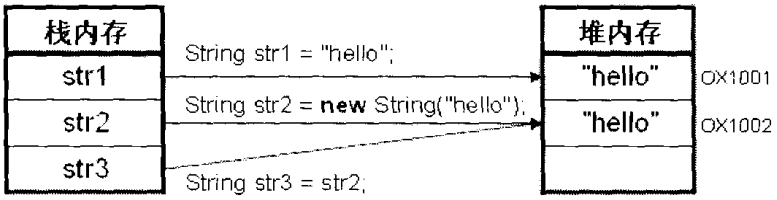


图 5-12　String 对象的声明

从图 5-12 中可以清楚地发现，每个 String 对象的内容实际上都是保存在堆内存之中的，而且堆中的内容是相等的。但是对 str1 和 str2 来说，其内容分别保存在了不同的空间，所以即使内容相等，但是地址的值也是不相等的，“==”是用来进行数值比较的，所以 str1 和 str2 比较不相等。那么 str2 和 str3 呢？从程序中可以发现 str2 和 str3 指向了同一个空间，是同一个地址，所以最终结果 str2 和 str3 的地址值是相等的，同理，str1 和 str3 的地址值也不相等，所以返回了 false。从上面的结果中可以清楚地发现“==”是用来进行地址值比较的。

那么既然无法使用“==”进行判断，那该如何去判断两个字符串的内容是否相等呢？此时，即可利用 String 中专门提供的方法（String 是一个类，则会存在各种方法）public boolean equals(String str)。

范例：使用 equals 方法对 String 的内容进行比较

```
public class StringDemo05 {
    public static void main(String[] args) {
        String str1 = "hello";                                  // 直接赋值
        String str2 = new String("hello");                      // 通过new赋值
        String str3 = str2;                                     // 传递引用
```

```
        System.out.println("str1 equals str2 --> " + (str1.equals(str2)));
        // --> true
        System.out.println("str1 equals str3 --> " + (str1.equals(str3)));
        // --> true
        System.out.println("str2 equals str3 --> " + (str2.equals(str3)));
        // --> true
    }
}
```

程序运行结果：

```
str1 equals str2 --> true
str1 equals str3 --> true
str2 equals str3 --> true
```

因为 equals()方法的作用是将内容进行比较，所以此处的返回结果都为 true。

5.7.3 String 两种实例化方式的区别

String 有两种实例化方式，一种是通过直接赋值的方式，另外一种是使用标准的 new 调用构造方法完成实例化，那么两种方式有什么区别？该使用哪种更好呢？

如果要想对以上问题进行解释，则首先必须明白一个重要概念，即一个字符串就是一个 String 类的匿名对象，匿名对象就是已经开辟了堆内存空间的并可以直接使用的对象。

范例：验证一个字符串就是 String 的匿名对象

```
public class StringDemo06 {
    public static void main(String[] args) {
        System.out.println("\"hello\" equals \"hello\" --> "
                + ("hello".equals("hello")));
    }
}
```

程序运行结果：

```
"hello" equals "hello" --> true
```

从程序的运行结果可以发现，一个字符串确实可以调用 String 类中的方法，也就证明了一个字符串就是一个 String 类的匿名对象。

所以，对于以下的代码：

```
String str1 = "hello";
```

实际上就是把一个在堆中开辟好的堆内存空间的使用权给了 str1 对象，实际上使用这种方式还有另外一个好处，就是如果一个字符串已经被一个名称所引用，则以后再有相同的字符串声明时，就不会再重新开辟空间。如下代码所示，观察代码的输出结果。

范例：采用直接赋值的方式声明多个 String 对象，并且内容相同，观察地址比较

```
public class StringDemo07 {
  public static void main(String[] args) {
```

```
        String str1 = "hello";      // 声明3个字符串变量，每个变量的内容都是一样的
        String str2 = "hello";      // 声明3个字符串变量，每个变量的内容都是一样的
        String str3 = "hello";      // 声明3个字符串变量，每个变量的内容都是一样的
        System.out.println("str1 == str2 --> " + (str1 == str2));  // --> true
        System.out.println("str1 == str3 --> " + (str1 == str3));  // --> true
        System.out.println("str2 == str3 --> " + (str2 == str3));  // --> true
    }
}
```

程序运行结果：

```
str1 == str2 --> true
str1 == str3 --> true
str2 == str3 --> true
```

从程序的运行结果可以发现，所有的字符串都使用了“==”进行比较，所得到的结果都为 true，那么也就是说这 3 个字符串指向的堆内存地址空间都是同一个，这也就是 String 使用直接赋值的方式之后，只要是以后声明的字符串内容相同，则不会再开辟新的内存空间，如图 5-13 所示。

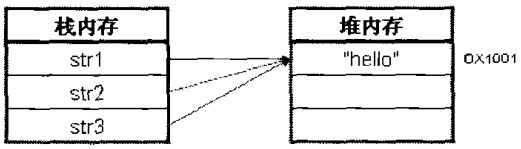


图 5-13　使用直接赋值的方式为 String 实例化

提示：在 Java 中会提供一个字符串池来保存全部的内容。

对于 String 的以上操作，在 Java 中称为共享设计，这种设计思路是，在 Java 中形成一个对象池，在这个对象池中保存多个对象，新实例化的对象如果已经在池中定义了，则不再重新定义，而从池中直接取出继续使用。String 就是因为采用了这样的设计，所以当内容重复时，会将对象指向已存在的实例空间。

下面为使用 new String()的方式实例化 String 对象的代码。

范例：使用 new String()的方式实例化 String 对象

```
public class StringDemo08 {
    public static void main(String[] args) {
        String str = new String("hello");  // 通过关键字new实例化
    }
}
```

一个字符串就是一个 String 类的匿名对象，而如果使用 new 关键字，不管如何都会再开辟一个新的空间，但是此时，此空间的内容还是 hello，所以上面的代码实际上是开辟了两个内存空间，如图 5-14 所示。

通过以上的两种实现方式比较可以知道哪种方式更合适，对于字符串的操作就采用直接赋值的方式完成，而不要采用构造方法传递字符串的方式完成，当然，在 String 类中也

存在一些其他的构造方法，在后面读者可以看到。

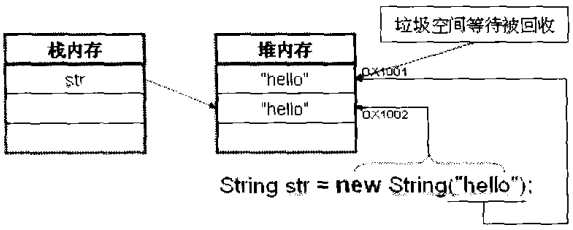


图 5-14　new String 方式实例化对象

5.7.4　字符串的内容不可改变

在使用 String 类进行操作时还有一个特性是特别重要的，那就是字符串的内容一旦声明则不可改变。下面通过一段代码进行讲解。

范例：修改字符串的内容

```
public class StringDemo09 {
    public static void main(String[] args) {
        String str = "hello";              // 声明字符串
        str = str + " world!";             // 修改字符串
        System.out.println("str = " + str);
    }
}
```

程序运行结果：

```
str = hello world!
```

从程序的运行结果发现，String 对象的内容可以修改，但是内容真的是修改了吗？下面通过内存的分配图说明字符串不可更改的含义，如图 5-15 所示。

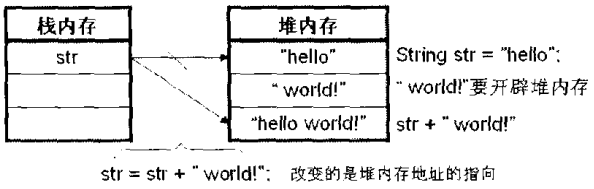


图 5-15　字符串内容的修改

从图 5-15 中可以清楚地发现，一个 String 对象内容的改变实际上是通过内存地址的“断开-连接”变化完成的，而本身字符串中的内容并没有任何的变化。

注意：String 在开发中的不正确应用。

读者日后在程序的开发中一定要明确记住字符串的内容不可改变这一重要特征，所以对于以下的程序代码，在开发中一定要尽可能回避。

范例：需要注意的代码

```
public class StringDemo10 {
    public static void main(String[] args) {
```

```
        String str1 = "LiXingHua";          // 声明字符串对象
        for (int i = 0; i < 100; i++) {     // 循环修改内容
            str1 += i;                      // 字符串引用不断改变
        }
        System.out.println(str1);
    }
}
```

以上代码通过循环修改了 100 次 str1 的内容，就意味着字符串的指向要“断开-连接”100 次，此程序代码的性能是很低的。当然，对于以上的需求在 Java 中也有相应的解决方式，可以使用 StringBuffer 类完成，StringBuffer 类的内容将在 Java 常用类库一章中为读者介绍。

5.7.5　String 类中常用方法

在 String 类中提供了大量的操作方法，表 5-3 中列出了一些比较常用的方法。

表 5-3　String 类常用操作方法

序号	方 法 定 义	类型	描　　述
1	public String(char[] value)	构造	直接将一个字符数组变为一个字符串
2	public String(char[] value,int offset,int count)	构造	将一个指定范围的字符数组变为字符串
3	public String(byte[] bytes)	构造	将一个 byte 数组全部变为字符串
4	public String(byte[] bytes,int offset,int length)	构造	将指定范围的 byte 数组变为字符串
5	public char[] toCharArray()	普通	将一个字符串变为字符数组
6	public char charAt(int index)	普通	从一个字符串中取出指定位置的字符
7	public byte[] getBytes()	普通	将一个字符串变为 byte 数组
8	public int length()	普通	取得字符串长度
9	public int indexOf(String str)	普通	从头开始查找指定的字符串位置
10	public int indexOf(String str,int fromIndex)	普通	从指定位置开始查找指定的字符串位置
11	public String trim()	普通	清除左、右两端的空格
12	public String substring(int beginIndex)	普通	从指定位置开始，一直取到尾进行字符串的截取
13	public String substring(int begin,int end)	普通	指定截取字符串的开始点和结束点
14	public String[] split(String regex)	普通	按照指定的字符串对字符串进行拆分（*）
15	public String toUpperCase()	普通	将一个字符串全部变为大写字母
16	public String toLowerCase()	普通	将一个字符串全部变为小写字母
17	public boolean startsWith(String prefix)	普通	判断是否以指定的字符串开头
18	public boolean endsWith(String suffix)	普通	判断是否以指定的字符串结尾
19	public boolean equals (String str)	普通	判断两个字符串内容是否相等
20	public boolean equalsIgnoreCase(String str)	普通	不区分大小写比较两个字符串是否相等
21	public String replaceAll(String regex,String replacement)	普通	字符串替换（*）

关于 String 类的更多方法，读者可以参考 JDK 开发文档，其中有更详细的解释。

提示：String 部分方法的使用注意。

split 和 replaceAll 两个方法是 String 对正则表达式的支持，在使用时可以使用正则表达式进行复杂的匹配。正则表达式的内容在常用类库章节将为读者介绍。

下面为以上方法的具体应用范例。

1. 字符串与字符数组的转换

字符串可以使用 toCharArray()方法变成一个字符数组，也可以使用 String 类的构造方法把一个字符数组变为一个字符串。

范例：验证以上的方法

```
public class StringAPIDemo01 {
    public static void main(String[] args) {
        String str1 = "hello";                          // 定义字符串
        char c[] = str1.toCharArray();                  // 将字符串变为字符数组
        for (int i = 0; i < c.length; i++) {            // 循环输出
            System.out.print(c[i] + "\t");
        }
        System.out.println("");
        String str2 = new String(c);                    // 将全部字符数组变为String
        String str3 = new String(c, 0, 3);              // 将部分字符数组变为String
        System.out.println(str2);                       // 输出字符串
        System.out.println(str3);                       // 输出字符串
    }
}
```

程序运行结果：

```
h    e    l    l    o
hello
hel
```

程序在一开始将一个字符串变为一个字符数组，字符串的长度就是转换之后字符数组的行数，也可以把一个字符数组变为字符串，可以将字符数组全部转换，也可以部分转换。

2. 从字符串中取出指定位置的字符

可以直接使用 String 类中的 charAt()方法取出字符串指定位置的字符。

范例：验证以上方法

```
public class StringAPIDemo02 {
    public static void main(String[] args) {
        String str1 = "hello" ;                         // 声明String对象
        System.out.println(str1.charAt(3)) ;            // 取出字符串中第4个字符
```

```
    }
}
```

程序运行结果：

```
l
```

3．把一个字符串变成一个 byte 数组，也可以把一个 byte 数组变成一个字符串

字符串可以通过 getBytes()方法将 String 变为一个 byte 数组，然后可以通过 String 的构造方法将一个字节数组重新变为字符串。

范例：验证以上代码

```
public class StringAPIDemo03 {
    public static void main(String[] args) {
        String str1 = "hello";                    // 定义字符串
        byte b[] = str1.getBytes();               // 将字符串变为byte数组
        System.out.println(new String(b));        // 将全部byte数组变为字符串
        System.out.println(new String(b, 1, 3));// 将部分byte数组变为字符串
    }
}
```

程序运行结果：

```
hello
ell
```

在以后的 IO 操作中经常会遇到字符串与 byte 数组或 char 数组之间的转换操作，可以发现，byte 数组与 char 数组的转换代码风格是相似的。

4．取得一个字符串的长度

在 String 中使用 length()方法取得字符串的长度。

范例：验证以上方法

```
public class StringAPIDemo04 {
    public static void main(String[] args) {
        String str1 = "hello LiXingHua";          // 定义字符串变量
        System.out.println("\"" + str1 + "\" 的长度为:" + str1.length());
    }
}
```

程序运行结果：

```
"hello LiXingHua" 的长度为:15
```

以上代码调用了 length()方法取得了字符串的长度。

注意：length 与 length()区别。

许多初学者经常对 length 和 length()两者的关系搞不清楚，在数组操作中，使用 length 取得数组的长度，但是操作的最后没有“()”，而字符串调用 length 是一个方法，只要是方法后面都有“()”。

5. 查找一个指定的字符串是否存在

在 String 中使用 indexOf()方法，可以返回指定的字符串的位置，如果不存在则返回-1。

范例：验证以上方法

```
public class StringAPIDemo05 {
    public static void main(String[] args) {
        String str1 = "abcdefgcgh";                    // 声明字符串
        System.out.println(str1.indexOf("c"));         // 查到返回位置
        System.out.println(str1.indexOf("c", 3));      // 查到返回位置，从第4个开
                                                          始查找
        System.out.println(str1.indexOf("x"));         // 没有查到返回-1
    }
}
```

程序运行结果：

```
2
7
-1
```

从程序的运行结果可以发现，如果查找到了指定的字符串，则会返回此字符串的位置，如果没有查找到，则返回-1。

6. 去掉左右空格

在实际的系统开发中，用户输入的数据中可能含有大量的空格，使用 trim()方法即可去掉字符串的左、右空格。

范例：验证以上方法

```
public class StringAPIDemo06 {
    public static void main(String[] args) {
        String str1 = "     hello          ";        // 定义字符串
        System.out.println(str1.trim());             // 去掉左右空格后输出
    }
}
```

程序运行结果：

```
Hello
```

从程序的运行结果可以发现，字符串中左、右两边的空格都被清除掉了。

7. 字符串截取

在 String 中提供了两个 substring()方法，一个是从指定位置截取到字符串结尾，另一个是截取指定范围的内容。

范例：验证以上方法

```
public class StringAPIDemo07 {
```

```
    public static void main(String[] args) {
        String str1 = "hello world";                    // 定义字符串
        System.out.println(str1.substring(6));          // 从第7个位置开始截取
        System.out.println(str1.substring(0, 5));       // 截取0～5个位置的内容
    }
}
```

程序运行结果：

```
world
hello
```

8. 按照指定的字符串拆分字符串

在 String 中通过 split()方法可以进行字符串的拆分操作，拆分的数据将以字符串数组的形式返回。

范例：验证以上方法

```
public class StringAPIDemo08 {
    public static void main(String[] args) {
        String str1 = "hello world";                    // 定义字符串
        String s[] = str1.split(" ");                   // 按空格进行字符串的拆分
        for (int i = 0; i < s.length; i++) {            // 循环输出
            System.out.println(s[i]);
        }
    }
}
```

程序运行结果：

```
hello
world
```

9. 字符串的大小写转换

在用户输入信息时，有时需要统一输入数据的大小写，此时就可以使用 toUpperCase()和 toLowerCase()两个方法完成大小写的转换操作。

范例：验证以上方法

```
public class StringAPIDemo09 {
    public static void main(String[] args) {
        System.out.println("将\"hello world\"转成大写：+"hello world".
        toUpperCase());
        System.out.println("将\"HELLO WORLD\"转成小写：+"HELLO WORLD".
        toLowerCase());
    }
}
```

程序运行结果：

```
将"hello world"转成大写：HELLO WORLD
将"HELLO WORLD"转成小写：hello world
```

10. 判断是否以指定的字符串开头或结尾

在 String 中使用 startsWith()方法可以判断字符串是否以指定的内容开头，使用 endsWith()方法可以判断字符串是否以指定的内容结尾。

范例：验证以上方法

```
public class StringAPIDemo10 {
    public static void main(String[] args) {
        String str1 = "**HELLO";                    // 定义字符串
        String str2 = "HELLO**";                    // 定义字符串
        if (str1.startsWith("**")) {                // 判断是否以“**”开头
            System.out.println("（**HELLO）以 ** 开头");
        }
        if (str2.endsWith("**")) {                  // 判断是否以“**”结尾
            System.out.println("（HELLO**）以 ** 结尾");
        }
    }
}
```

程序运行结果：

```
（**HELLO）以 ** 开头
（HELLO**）以 ** 结尾
```

11. 不区分大小写进行字符串比较

在 String 中可以通过 equals()方法进行字符串内容的比较，但是这种比较方法是区分大小写的比较，如果要完成不区分大小写的比较则可以使用 equalsIgnoreCase()方法。

范例：验证以上方法

```
public class StringAPIDemo11 {
    public static void main(String[] args) {
      String str1 = "HELLO";                        // 定义字符串
      String str2 = "hello";                        // 定义字符串
      System.out.println("\"HELLO\" equals \"hello\"" + str1.equals(str2));
      System.out.println("\"HELLO\" equalsIgnoreCase \"hello\""
            + str1.equalsIgnoreCase(str2)); // 不区分大小写比较字符串内容
    }
}
```

程序运行结果：

```
"HELLO" equals "hello" false
"HELLO" equalsIgnoreCase "hello" true
```

从程序的运行结果可以发现，equals 方法在比较时是区分大小写的。

12. 将一个指定的字符串替换成其他的字符串

使用 String 的 replaceAll()方法可以将字符串的指定内容进行替换。

范例：验证以上方法

```
public class StringAPIDemo12 {
    public static void main(String[] args) {
        String str = "hello";                          // 声明字符串
        String newStr = str.replaceAll("l", "x");  // 现在将所有的l替换成x
        System.out.println("替换之后的结果：" + newStr);
    }
}
```

程序运行结果：

```
替换之后的结果：hexxo
```

5.8　引用传递及基本应用

5.8.1　引用传递

前面已经介绍了引用传递的基本概念，所谓的引用传递就是指将堆内存空间的使用权交给多个栈内存空间，下面通过 3 个范例让读者加深对前面所学引用知识的理解。

1. 引用传递范例一

范例：对象引用传递

```
class Demo {
    int temp = 30;                                  // 此处为了访问方便，属性暂不封装
}
public class TestJava0538 {
    public static void main(String[] args) {
        Demo d1 = new Demo();
        d1.temp = 50;
        System.out.println("fun()方法调用之前：" + d1.temp);
        fun(d1);
        System.out.println("fun()方法调用之后：" + d1.temp);
    }
    public static void fun(Demo d2) {          // 此处的方法由主方法直接调用
        d2.temp = 1000;
    }
}
```

程序运行结果：

```
fun()方法调用之前：50
fun()方法调用之后：1000
```

从以上的运行结果可以发现，在 fun 方法之中接收了 Demo 类对象 d1，并将 temp 属性的内容进行了修改，因为是引用传递，所以最终 temp 的值是 1000，此程序可以通过图 5-16 表示。

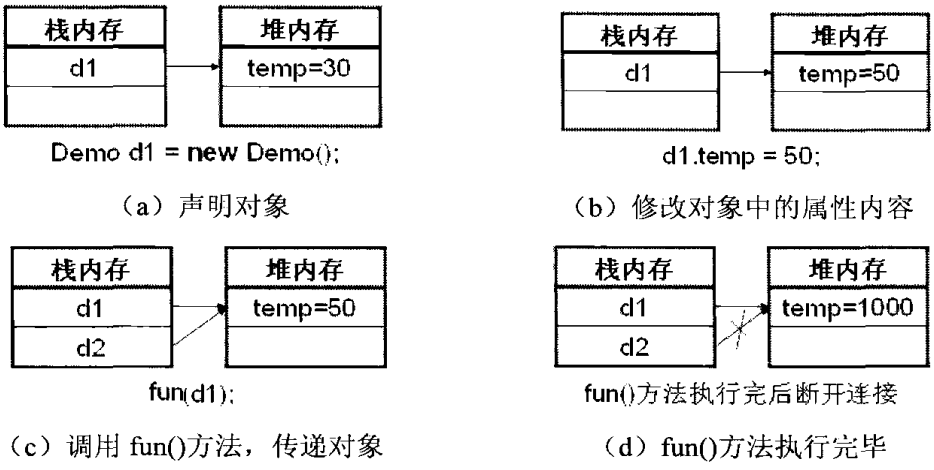


图 5-16　引用传递范例的内存分析图

提示：初学者要多画图以加深理解。

对于引用传递的操作，初学者应该多拿纸笔将内存的分配图形完整地描绘出来，这样可以加深对引用传递操作步骤的理解，为后续的学习打下基础。

以上的代码操作实际上与前面方法中传递数组的操作过程一致，了解了以上的范例之后，再来看下面的程序，观察结果。

2．引用传递范例二

范例：引用传递

```
public class RefDemo02 {
    public static void main(String[] args) {
        String str1 = "hello" ;                    // 实例化字符串对象
        System.out.println("fun()方法调用之前：" + str1);
        fun(str1);                                 // 调用fun()方法
        System.out.println("fun()方法调用之后：" + str1);
    }
    public static void fun(String str2) {   // 此处的方法由主方法直接调用
        str2 = "MLDN" ;                            // 修改字符串内容
    }
}
```

程序运行结果：

```
fun()方法调用之前：hello
fun()方法调用之后：hello
```

从程序的运行结果中发现，虽然此时传递的是一个 String 类型的对象，但是结果并没有像之前一样发生改变，因为字符串的内容一旦声明是不可改变的，改变的只是其内存地址的指向。本程序的内存操作如图 5-17 所示。

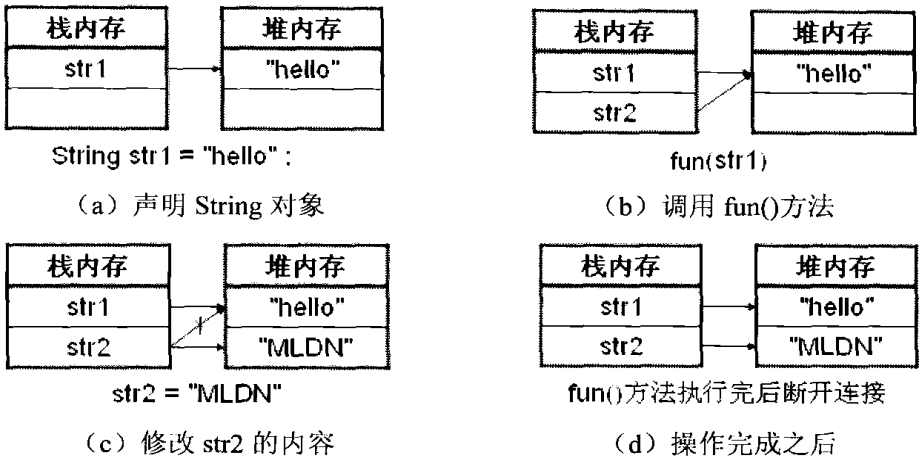


图 5-17　引用传递范例的内存分析图

以上的操作代码并不难理解，因为每一个字符串对象都表示一个匿名对象，这样在 fun() 方法操作中，如果为 str2 重新设置内容，就相当于改变了 str2 的引用，而 str1 本身的内容并不会受任何影响。

3．引用传递范例三

清楚了前面的两个范例之后，下面再来看第 3 个范例，读者可以先自行分析程序的运行结果。

范例：引用传递

```
class Demo {
    String temp = "hello";                    // 此处为了访问方便，属性暂不封装
}
public class RefDemo03 {
    public static void main(String[] args) {
        Demo d1 = new Demo();                 // 实例化对象
        d1.temp = "world";                    // 修改对象中的temp属性
        System.out.println("fun()方法调用之前：" + d1.temp);
        fun(d1);                              // 调用fun()方法
        System.out.println("fun()方法调用之后：" + d1.temp);
    }
    public static void fun(Demo d2) {         // 此处的方法由主方法直接调用
        d2.temp = "MLDN";                     // 修改属性的内容
```

```
    }
}
```

程序运行结果：

```
fun()方法调用之前：world
fun()方法调用之后：MLDN
```

本程序运行后，发现在 fun()方法中将属性的内容修改了，内存操作如图 5-18 所示。

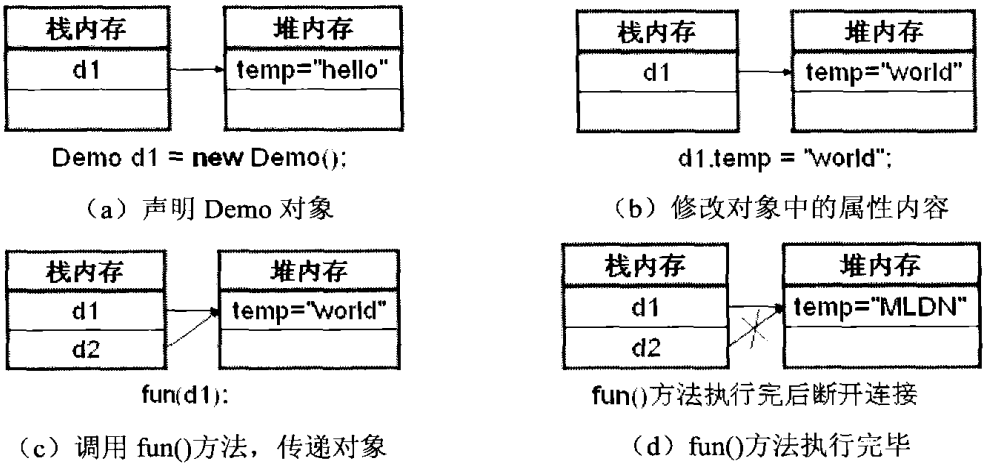


图 5-18　引用传递范例的内存分析图

从图 5-18 中可以清楚地发现，本程序的分析方法与第 1 个范例完全一样，因为 String 是作为一个 Demo 类的属性存在的，而在操作时更改的只是 Demo 类中属性的内容。

5.8.2　接收本类的引用

以上为引用传递的基本形式，实际上在对象引用传递上也可以在一个类中接收自己本类对象的实例，而且接收完之后，可以方便地通过此对象直接进行本类中封装属性的访问，如下面的代码。

范例：接收自己本类的对象

```
class Demo {                                    // 定义Demo类
    private int temp = 30;                      // 声明temp属性并封装
    public void fun(Demo d2) {                  // 接收本类的引用
        d2.temp = 50;                           // 直接通过对象调用本类的私有属性
    }
    public int getTemp() {                      // getter方法
        return temp;
    }
    public void setTemp(int t) {                // setter方法
        temp = t;
    }
}
```

```
public class RefDemo04 {
    public static void main(String[] args) {
        Demo d1 = new Demo();                    // 实例化Demo对象
        d1.setTemp(50);                          // 修改temp内容
        d1.fun(d1);                              // 此处把Demo的对象传回到自己的类中
        System.out.println("temp = " + d1.getTemp());
    }
}
```

程序运行结果：

```
temp = 50
```

有些读者会认为这样做有些不是很合理，因为私有属性不应该被对象访问，但是另一方面因为是在类的内部，所以可以访问其中的私有属性，因此只要符合引用传递的语法，则可以向任意地方传递。

提示：在对象比较时才会使用。

此种引用方式的传递在关于对象比较操作时才会使用，其他时候基本上都很少使用。

5.8.3 范例——一对一关系

实际上使用引用传递还可以表示出生活中的以下一种场景，例如，一个人有一本书，一本书属于一个人。因而可以得出这样的结论：人应该是一个具体的类，书也应该是一个具体的类，在人的类中应该存在一个属性表示书，在书的类中也应该存在一个属性表示人。

范例：代码实现

```
class Person {                                   // 定义Person类
    private String name;                         // 姓名
    private int age;                             // 年龄
    private Book book;                           // 一个人有一本书
    public Person(String name, int age) {        // 通过构造设置内容
        this.setName(name);
        this.setAge(age);
    }
    public String getName() {                    // 返回姓名
        return name;
    }
    public void setName(String n) {              // 设置姓名
        name = n;
    }
    public int getAge() {                        // 返回年龄
        return age;
    }
    public void setAge(int a) {                  // 设置年龄
```

```
        age = a;
    }
    public Book getBook() {                          // 得到本人的书
        return book;
    }
    public void setBook(Book b) {                    // 设置本人的书
        book = b;
    }
}
class Book {
    private String title;                            // 标题
    private float price;                             // 价格
    private Person person;                           // 一本书属于一个人
    public Book(String title, float price) {         // 通过构造设置属性内容
        this.setTitle(title);
        this.setPrice(price);
    }
    public String getTitle() {                       // 得到标题
        return title;
    }
    public void setTitle(String t) {                 // 设置标题
        title = t;
    }
    public float getPrice() {                        // 得到价格
        return price;
    }
    public void setPrice(float p) {                  // 设置价格
        price = p;
    }
    public Person getPerson() {                      // 得到书的所有人
        return person;
    }
    public void setPerson(Person person) {           // 设置书的所有人
        this.person = person;
    }
}
public class RefDemo05 {
    public static void main(String[] args) {
        Person per = new Person("张三", 30);          // 实例化Person对象
        Book bk = new Book("JAVA SE核心开发", 90.0f); // 实例化Book对象
        per.setBook(bk);                             // 设置两个对象间的关
                                                     系，一个人有一本书
```

```
        bk.setPerson(per);                                    // 设置两个对象间的关系，一本书
                                                                 属于一个人
        System.out.println("从人找到书 --> 姓名：" + per.getName()+"；年龄："
                +per.getAge()+"；书名："+per.getBook().getTitle()+"；价格："
                + per.getBook().getPrice());    // 可以通过人找到书
        System.out.println("从书找到人 --> 书名：" + per.getBook().getTitle()
        + "；价格："
                + per.getBook().getPrice() + "；姓名：" + per.getName() + "；
                年龄："
                + per.getAge());                              // 也可以通过书找到所有人
    }
}
```

程序运行结果：

```
从人找到书 --> 姓名：张三；年龄：30；书名：JAVA SE核心开发；价格：90.0
从书找到人 --> 书名：JAVA SE核心开发；价格：90.0；姓名：张三；年龄：30
```

从程序的运行结果中可以发现，人和书有关联关系，如图 5-19 所示。

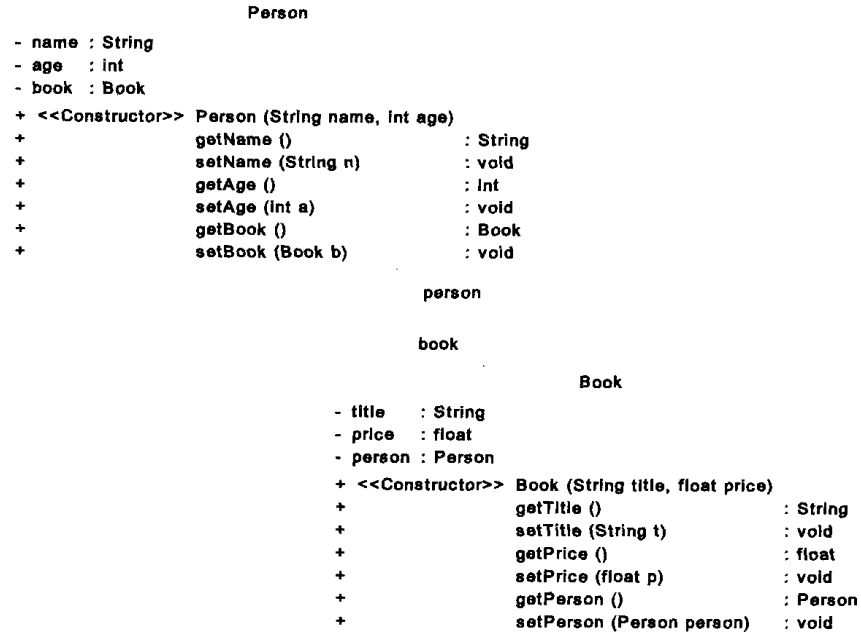


图 5-19　类图关系

提示：类的关系应用。

在一些复杂的系统中，为了更好地阐述类之间的关系，往往会在实体层进行设置，所谓的实体层就是一个个单独实体，例如，Book 类或 Person 类都是实体层的类。

5.8.4　范例——进一步深入一对一关系

以上已经完成了一个基本的关系，但是现在有了一个新的要求：一个人有一个孩子，

每个孩子还会有一本书，该如何分析呢？难道再单独建立一个孩子的类吗？很明显这样做是很不可取的，因为一个孩子也是一个人，所以此时，只需要简单地修改 Person 类，在类中增加一个自己的引用即可。

范例：对程序进行扩展

```
class Person {                                          // 定义Person类
    private String name;                                // 姓名
    private int age;                                    // 年龄
    private Book book;                                  // 一个人有一本书
    private Person child;                               // 一个人有一个孩子
    public Person(String name, int age) {               // 通过构造设置属性内容
        this.setName(name);
        this.setAge(age);
    }
    public String getName() {                           // 得到姓名
        return name;
    }
    public void setName(String n) {                     // 设置姓名
        name = n;
    }
    public int getAge() {                               // 得到年龄
        return age;
    }
    public void setAge(int a) {                         // 设置年龄
        age = a;
    }
    public Book getBook() {                             // 一个人有一个孩子
        return book;
    }
    public void setBook(Book b) {                       // 设置所拥有的书
        book = b;
    }
    public Person getChild() {                          // 得到孩子
        return child;
    }
    public void setChild(Person child) {                // 设置孩子
        this.child = child;
    }
}
class Book {
    ...                                                 // 此类与之前定义一样
}
public class RefDemo06 {
```

```
    public static void main(String[] args) {
        Person per = new Person("张三", 30);     // 实例化Person对象
        Person cld = new Person("张草", 10);     // 定义一个孩子
        Book bk = new Book("JAVA SE核心开发", 90.0f);    // 实例化Book对象
        Book b = new Book("一千零一夜", 30.3f);  // 定义孩子有的书
        per.setBook(bk);                    // 设置对象间的关系，一个人有一本书
        bk.setPerson(per);                  // 设置对象间的关系，一本书属于一个人
        cld.setBook(b) ;                    // 设置对象间的关系，一个孩子有一本书
        b.setPerson(cld) ;                  // 设置对象间的关系，一本书属于一个孩子
        per.setChild(cld) ;                 // 设置对象间的关系，一个人有一个孩子
        System.out.println("从人找到书 --> 姓名：" + per.getName()+"；年龄："
              + per.getAge()+"；书名："+per.getBook().getTitle()+"；价格："
              + per.getBook().getPrice());       // 可以通过人找到书
        System.out.println("从书找到人 --> 书名：" + per.getBook().getTitle()
        + "；价格："
              + per.getBook().getPrice()+"；姓名："+per.getName()+"；年龄："
              + per.getAge());                   // 也可以通过书找到其所有人
        // 通过人找到孩子，并找到孩子所拥有的书
        System.out.println(per.getName() + "的孩子 --> 姓名："
              + per.getChild().getName() + "；年龄：" + per.getChild().getAge()
              + "；书名：" + per.getChild().getBook().getTitle()+"；价格："
              + per.getChild().getBook().getPrice());
    }
}
```

程序运行结果：

```
从人找到书 --> 姓名：张三；年龄：30；书名：JAVA SE核心开发；价格：90.0
从书找到人 --> 书名：JAVA SE核心开发；价格：90.0；姓名：张三；年龄：30
张三的孩子 --> 姓名：张草；年龄：10；书名：一千零一夜；价格：30.3
```

5.9 this 关键字

在 Java 中 this 关键字可能是最难理解的，因为其语法较为灵活。从之前的代码中应该发现，可以使用 this 强调本类中的方法，除此之外 this 还有以下作用：

- 表示类中的属性。
- 可以使用 this 调用本类的构造方法。
- this 表示当前对象。

5.9.1 使用 this 调用本类中的属性

在程序中可以使用 this 调用本类属性。

范例：现在有以下的一个类

```
class Person {
    private String name;
    private int age;
    public Person(String n,int a){
        name = n ;
        age = a ;
    }
    public String getInfo() {
        return "姓名：" + name + "，年龄：" + age;
    }
}
```

在本类中构造方法的目的很明确，就是为类中的属性赋值，但是从构造方法的传递两个参数名称上很难看出 n 表示的意义或是 a 表示的意义，所以为了可以清楚地表示参数的意义，将以上的类修改为如下形式：

```
class Person {
    private String name;
    private int age;
    public Person(String name,int age){
        name = name ;
        age = age ;
    }
    public String getInfo() {
        return "姓名：" + name + "，年龄：" + age;
    }
}
```

此时，从参数的名称上可以很清楚地知道，要传递的第 1 个参数是姓名，第 2 个参数是年龄，但是这样一来，又一个新的问题出现了：

```
name = name ;
age = age ;
```

类的本意是要将参数传递的 name 值赋给类中的 name 属性，把 age 的值赋给 age 属性，但是以上程序真的可以达到这样的效果吗？下面来验证一下：

```
class Person {
    private String name;                          // 姓名
    private int age;                              // 年龄
    public Person(String name,int age){           // 通过构造赋值
        name = name ;                             // 此处是哪个name?
        age = age ;                               // 此处是哪个age?
    }
```

```
    public String getInfo() {                    // 取得信息
        return "姓名：" + name + "，年龄：" + age;
    }
}
public class ThisDemo01 {
    public static void main(String[] args) {
        Person per1 = new Person("张三",33) ;    // 调用构造实例化对象
        System.out.println(per1.getInfo()) ;     // 取得信息
    }
}
```

程序运行结果：

```
姓名：null，年龄：0
```

从程序的运行结果中可以发现，姓名为 null，年龄为 0，也就是说并没有把从构造方法中传递进去的参数值赋给属性，那么也就证明，现在的构造方法并不能成功地把传递进去的值赋值给类中的属性。也就是说，在赋值时属性并不是明确地被指出，所以此时就可以利用 this 关键字，代码修改如下：

```
class Person {
    private String name;                         // 姓名
    private int age;                             // 年龄
    public Person(String name,int age){          // 通过构造赋值
        this.name = name ;                       // 明确表示为类中的name属性赋值
        this.age = age ;                         // 明确表示为类中的age属性赋值
    }
    public String getInfo() {                    // 取得信息
        return "姓名：" + name + "，年龄：" + age;
    }
}
public class ThisDemo02 {
    public static void main(String[] args) {
        Person per1 = new Person("张三",33) ;// 调用构造实例化对象
        System.out.println(per1.getInfo()) ;// 取得信息
    }
}
```

程序运行结果：

```
姓名：张三，年龄：33
```

从程序的运行结果来看，程序达到了所需要的目的，而在构造方法之中，因为已经明确地标识出了类中的两个属性“this.name”和“this.age”，所以在进行赋值操作时也不会产生歧义。

提问：下面的代码操作的是哪个属性？

对于上面代码可知哪个是属性，哪个是参数，但是对于下面的代码：

```
name = name ;
age = age ;
```

其中的 name 和 age 分别是什么呢？

回答：都是方法中的参数。

实际上上面的“name=name”、“age=age”中的两个 name 和两个 age 都是构造方法中的参数。例如：

假设现在一个用户要拿笔写字，摆在此用户面前的有一支笔，离用户 10 米远的地方还有一支笔，那么如果您是这个用户，您会选择哪支笔？没错，肯定会使用身边的这支笔，因为离得近。

实际上对于程序上也是一样的，在程序的构造方法中已经存在了 name 和 age 属性，那么在构造方法中如果要使用 name 或 age 属性，则肯定要找近的，所以上面的 name 和 age 使用的都是构造方法中的参数。

所以，建议读者以后在类中访问属性时都加上 this 关键字。

5.9.2 使用 this 调用构造方法

如果一个类中有多个构造方法，也可以利用 this 关键字互相调用。

假设现在要求不管类中有多少个构造方法，只要对象一被实例化，就必须打印一行“一个新的对象被实例化”信息出来，很明显，此时如果在各个构造方法中编写此输出语句肯定不合适，所以可以利用 this 关键字完成，如以下代码。

范例：使用 this 调用本类的构造方法

```
class Person {
    private String name;                    // 姓名
    private int age;                        // 年龄
    public Person() {                       // 无参构造
        System.out.println("一个新的Person对象被实例化。");
    }
    public Person(String name, int age) {
        this();                             // 在此处调用Person类中的无参构造方法
        this.name = name;                   // 为name属性赋值
        this.age = age;                     // 为age属性赋值
    }
    public String getInfo() {               // 取得信息
        return "姓名：" + name + "，年龄：" + age;
    }
}
public class ThisDemo03 {
```

```
    public static void main(String[] args) {
        Person per1 = new Person("张三", 33);    // 调用有参构造
        System.out.println(per1.getInfo());     // 输出信息
    }
}
```

程序运行结果：

```
一个新的Person对象被实例化。
姓名：张三，年龄：33
```

本程序中提供了两个构造方法，其中有两个参数的构造方法中使用 this()的形式调用本类中的无参构造方法，所以即使是通过有两个参数的构造方法实例化，最终结果还是会把对象实例化的信息打印出来。

如果把 this()调用无参构造方法的位置任意调换，那么就可以在任何时候都可以调用构造方法了吗？实际上这样理解是错误的。构造方法是在实例化对象时被自动调用的，也就是说在类中的所有方法中，只有构造方法是被优先调用的，所以使用 this 调用构造方法必须也只能放在构造方法的第 1 行，下面的程序就是一个错误的程序。

范例：错误的代码调用

```
class Person {
    private String name;          // 姓名
    private int age;              // 年龄
    public Person() {             // 无参构造
        System.out.println("一个新的Person对象被实例化。");
    }
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
        this();                   // 错误的调用只能放在构造方法的首行
    }
    public String getInfo() {
        this();                   // 错误的调用只能放在构造方法的首行
        return "姓名：" + name + "，年龄：" + age;
    }
}
```

程序编译时会出现以下的错误：

```
ThisDemo04.java:10: call to this must be first statement in constructor
                this();                          // 错误的调用只能放在构造方法的首行
                    ^
ThisDemo04.java:13: call to this must be first statement in constructor
                this();                          // 错误的调用只能放在构造方法的首行
                    ^
2 errors
```

以上的错误提示就是 this()只能放在构造方法的首行。

另外，要提醒读者的是，this 调用构造方法时一定要留一个构造方法作为出口，即程序中至少存在一个构造方法不使用 this 调用其他构造方法。

范例：错误的代码

```
class Person {
    private String name;
    private int age;
    public Person() {
        this("LXH",30) ;                    // 此处为错误的代码
        System.out.println("一个新的Person对象被实例化。");
    }
    public Person(String name){
        this() ;
    }
    public Person(String name, int age) {
        this(name) ;
        this.age = age;
    }
    public String getInfo() {
        return "姓名：" + name + "，年龄：" + age;
    }
}
```

程序编译时出现以下错误提示：

```
ThisDemo05.java:5: recursive constructor invocation
      public Person() {
             ^
1 error
```

以上的错误提示为递归调用了构造方法，所以在构造方法间互相调用时一定要留一个出口，一般都将无参构造方法作为出口，即在无参构造方法中最好不要再去调用其他构造方法。

5.9.3 this 表示当前对象

前面已经讲解了何时使用 this 调用属性和何时使用 this 调用方法，实际上在这些特性之外，this 最重要的特点就是表示当前对象，在 Java 中当前对象就是指当前正在调用类中方法的对象。

提示：关于当前对象的解释。

现在假设有张三、李四、王五 3 个人在谈话，如果现在说话的人是张三，则“当前正在说话的人”就是张三，如果现在说话的人是李四，则“当前正在说话的人”就是李四，依此类推，所以只要是“当前正在说话的人”就表示是当前对象。

范例：观察 this 表示当前对象

```
class Person {
    public String getInfo() {
        System.out.println("Person类 --> " + this); // 直接打印this
        return null ;                               // 此处返回null，为的是让
                                                       语法不出错
    }
}
public class ThisDemo06 {
    public static void main(String[] args) {
        Person per1 = new Person() ;
        Person per2 = new Person() ;
        System.out.println("MAIN方法 --> " + per1); // 直接打印对象
        per1.getInfo() ;
        System.out.println("-------------------------") ;
        System.out.println("MAIN方法 --> " + per2); // 直接打印对象
        per2.getInfo() ;
    }
}
```

程序运行结果：

```
MAIN方法 --> org.lxh.demo05.Person@757aef
Person类 --> org.lxh.demo05.Person@757aef
-------------------------
MAIN方法 --> org.lxh.demo05.Person@d9f9c3
Person类 --> org.lxh.demo05.Person@d9f9c3
```

从程序的运行结果来看，直接打印对象和调用 getInfo()方法打印的结果是一样的，而且在 getInfo()方法中永远是一个 this 关键字，也就是说哪个对象调用了类中的方法，则 this 就表示哪个对象。那么这样的一个特性到底有什么用处呢？下面通过一个实例来分析 this 表示当前对象的用法。

现在假设有以下的类：

```
class Person {
    private String name;                          // 姓名
    private int age;                              // 年龄
    public Person(String name, int age) {         // 通过构造为属性赋值
        this.setName(name);                       // 设置姓名
        this.setAge(age);                         // 设置年龄
    }
    public String getName() {                     // 取得姓名
        return name;
    }
```

```
    public void setName(String n) {                    // 设置姓名
        name = n;
    }
    public int getAge() {                              // 取得年龄
        return age;
    }
    public void setAge(int a) {                        // 设置年龄
        age = a;
    }
}
```

在此类中有 name 和 age 两个属性，现在要求产生两个 Person 对象，并且判断这两个对象是否相等。那么这个时候就会产生以下两个问题：

- 如何进行对象的比较？
- 在哪里进行对象的比较？

对于第 1 个问题，实际上从前面讲解 String 的代码中就应该可以了解，要想进行对象的比较，必须比较其内容，但是 Person 中现在并不能使用 equals 方法，所以此时就需要对每一个属性进行依次的判断，如果所有属性的内容都相等，那么就可以证明这两个对象相等，这一点可以通过如下的代码进行验证。

范例：对象比较的第 1 种方式

```
public class ThisDemo07 {
    public static void main(String[] args) {
        Person per1 = new Person("张三",30) ;   // 声明两个对象，内容完全相等
        Person per2 = new Person("张三",30) ;   // 声明两个对象，内容完全相等
        // 直接在主方法中依次取得各个属性进行比较
        if (per1.getName().equals(per2.getName())
                && per1.getAge() == per2.getAge()) {
            System.out.println("两个对象相等！");
        } else {
            System.out.println("两个对象不相等！");
        }
    }
}
```

程序运行结果：

```
两个对象相等！
```

以上程序确实完成了两个对象的比较功能，但是一个新的问题又来了：把所有的判断方法放在主方法中合适吗？实际上这也就是之前所提出的第 2 个问题“在哪里进行对象的比较？”，可以做这样的一个比喻，假如现在有两个人，如果一个人要判断另外一个人和自己是否相等，则肯定由这个人发出比较的请求，即在人这个类中就应该存在与其他人比较的方法。所以，对象比较的方法应该放在 Person 类之中，而不应该在 main()方法中进行

编写，修改后的代码如下。

范例：修改后的对象比较操作

```
class Person {
    private String name;                          // 姓名
    private int age;                              // 年龄
    public Person(String name, int age) {         // 通过构造为属性赋值
        this.setName(name);                       // 设置姓名
        this.setAge(age);                         // 设置年龄
    }
    public boolean compare(Person per){
        // 调用此方法时里面存在两个对象：当前对象、传入的对象
        Person p1 = this ;                        // 表示当前调用方法的对象，为per1
        Person p2 = per ;                         // 传递到方法中的对象，为per2
        if(p1==p2){                               // 首先比较两个地址是否相等
            return true ;
        }
        // 分别判断每一个属性是否相等
        if(p1.name.equals(p2.name)&&p1.age==p2.age){
            return true ;                         // 两个对象相等
        }else{
            return false ;                        // 两个对象不相等
        }
    }
    public String getName() {                     // 取得姓名
        return name;
    }
    public void setName(String n) {               // 设置姓名
        name = n;
    }
    public int getAge() {                         // 取得年龄
        return age;
    }
    public void setAge(int a) {                   // 设置年龄
        age = a;
    }
}
public class ThisDemo08 {
    public static void main(String[] args) {
        Person per1 = new Person("张三",30) ;// 声明两个对象，内容完全相等
        Person per2 = new Person("张三",30) ;// 声明两个对象，内容完全相等
        if(per1.compare(per2)){                   // 进行对象的比较
            System.out.println("两个对象相等！");
```

```
            }else{
                System.out.println("两个对象不相等！") ;
            }
        }
    }
```

以上程序 Person 类中定义了一个 compare 方法，此方法的主要功能就是专门完成两个对象的比较操作，在比较时，首先进行地址的比较，如果两个对象的地址一样，则肯定是同一个对象，而如果地址不相等，则将一个个属性进行依次的比较。

注意：对象比较操作。

对象比较操作在开发中是一个重要的概念，在后面的章节中还将进行讲解，但其基本的操作形式是不变的，所以一定要重点掌握。

5.10 static 关键字

从前面讲解的概念中读者应该清楚，如果使用一个类分别开辟栈内存及堆内存，在堆内存中要保存对象中的属性，每个对象有自己的属性，如果现在有些属性希望被所有对象共享，则就必须将其声明为 static 属性。如果一个类中的方法想由类调用，则可以声明为 static 方法。

5.10.1 使用 static 声明属性

如果在程序中使用 static 声明属性，则此属性称为全局属性（有些也称为静态属性），那么声明成全局属性到底有什么用呢？先来观察以下的代码。

范例：观察以下代码

```
class Person {
    String name;                                    // 定义name属性，此处暂不封装
    int age;                                        // 定义age属性，此处暂不封装
    String country = "A城";                         // 定义城市属性，有默认值
    public Person(String name, int age) {           // 通过构造方法设置name和age
        this.name = name;                           // 为name赋值
        this.age = age;                             // 为age赋值
    }
    public void info() {                            // 直接打印信息
        System.out.println("姓名: "+this.name+", 年龄: "+this.age+", 城市: "
                + country);
    }
};
public class StaticDemo01 {
    public static void main(String args[]) {
```

```
        Person p1 = new Person("张三", 30);      // 实例化对象
        Person p2 = new Person("李四", 31);      // 实例化对象
        Person p3 = new Person("王五", 32);      // 实例化对象
        p1.info();                               // 输出信息
        p2.info();                               // 输出信息
        p3.info();                               // 输出信息
    }
};
```

程序运行结果：

```
姓名：张三，年龄：30，城市：A城
姓名：李四，年龄：31，城市：A城
姓名：王五，年龄：32，城市：A城
```

以上代码为了读者观察方便，暂时没有使用 private 关键字进行封装。以上的程序是一个很简单的应用，程序的运行结果也不难理解。但是在此代码之中也有一些不妥之处。

实际上，读者可以考虑这样一种情况：如果现在假设此城市不叫“A 城”，而改为了“B 城”，而且此类已经产生了 5000 个对象，那么意味着，如果要修改这些对象的城市信息，则要把这 5000 个对象中的城市属性同时修改。要修改 5000 遍。这样肯定是不行的，那么该怎么解决呢？最好的做法是一次性修改之后，所有对象的城市信息都可以修改成功，所以，此时就可以把城市的属性使用 static 关键字进行声明。

范例：使用 static 声明城市属性

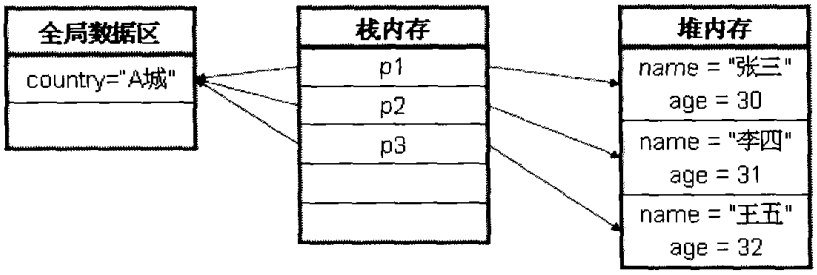
```
class Person {
    String name;                                  // 定义name属性，此处暂不封装
    int age;                                      // 定义age属性，此处暂不封装
    static String country = "A城";                // 使用static定义城市属性，有默认值
    public Person(String name, int age) {         // 通过构造方法设置name和age
        this.name = name;                         // 为name赋值
        this.age = age;                           // 为age赋值
    }
    public void info() {                          // 直接打印信息
        System.out.println("姓名："+this.name+"，年龄："+this.age+"，城市："
                + country);
    }
};
public class StaticDemo02 {
    public static void main(String args[]) {
        Person p1 = new Person("张三", 30); // 实例化对象
        Person p2 = new Person("李四", 31); // 实例化对象
        Person p3 = new Person("王五", 32); // 实例化对象
        System.out.println("------------ 修改之前 ----------------");
        p1.info();                                // 输出信息
```

```
            p2.info();                                    // 输出信息
            p3.info();                                    // 输出信息
            System.out.println("------------ 修改之后 -----------------");
            p1.country = "B城";                           // 修改static属性
            p1.info();                                    // 输出信息
            p2.info();                                    // 输出信息
            p3.info();                                    // 输出信息
        }
    };
```

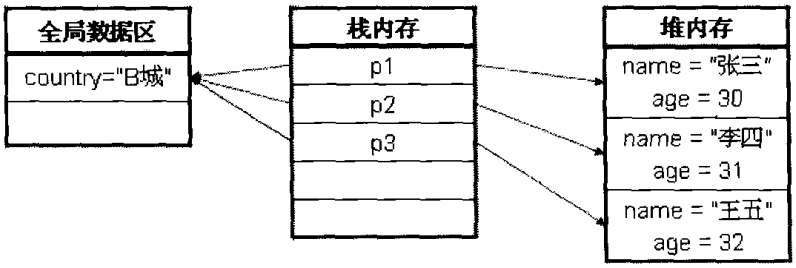
程序运行结果：

```
------------ 修改之前 -----------------
姓名：张三，年龄：30，城市：A城
姓名：李四，年龄：31，城市：A城
姓名：王五，年龄：32，城市：A城
------------ 修改之后 -----------------
姓名：张三，年龄：30，城市：B城
姓名：李四，年龄：31，城市：B城
姓名：王五，年龄：32，城市：B城
```

在程序中，为了读者观察方便，分别加入了“修改之前”和“修改之后”信息提示，从以上代码中可以发现，只修改了一个对象的城市属性，则全部对象的城市属性内容都发生了变化，说明使用 static 声明的属性是所有对象共享的，上面程序的内存分布图如图 5-20 所示。



（a）country 属性修改之前



（b）country 属性修改之后

图 5-20　static 属性保存的内存分配图

提示：Java 中常用的内存区域。

在 Java 中主要存在 4 块内存空间，这些内存空间的名称及作用如下。

（1）栈内存空间：保存所有的对象名称（更准确地说是保存了引用的堆内存空间的地址）。

（2）堆内存空间：保存每个对象的具体属性内容。

（3）全局数据区：保存 static 类型的属性。

（4）全局代码区：保存所有的方法定义。

以上程序已经完成了基本的修改功能，进一步思考：一个类中的公共属性现在由一个对象修改，这样操作合适吗？很明显不合适，类的公共属性应该由类进行修改是最合适的，因为用户不知道一个类到底有多少个对象产生。所以以上代码在访问 static 属性时最好可以由类名称直接调用，那么有时也就把使用 static 声明的属性称为类属性。

【格式 5-6　类属性调用】

```
类名称.static属性
```

所以，以上代码在访问 country 属性时最好使用如下代码。

范例：使用类名称访问 static 属性

```
Person.country = "B城";                                    // 使用类名称修改static属性内容
```

5.10.2　使用 static 声明方法

static 既可以在声明属性时使用，也可以用其来声明方法，用它声明的方法有时也被称为“类方法”，请看下面的范例。

范例：使用 static 声明方法

```
class Person {
    private String name;                                   // 定义name属性
    private int age;                                       // 定义age属性
    private static String country = "A城";                 // 定义static属性
    public static void setCountry(String c) {              // 定义static方法
        country = c;                                       // 修改static属性
    }
    public Person(String name, int age) {                  // 构造方法
        this.name = name;
        this.age = age;
    }
    public void info() {                                   // 信息输出
        System.out.println("姓名："+this.name+"，年龄："+this.age+"，城市："
                + country);
    }
    public static String getCountry() {                    // 取得static属性
        return country;
    }
};
```

```
public class StaticDemo04 {
    public static void main(String args[]) {
        Person p1 = new Person("张三", 30);      // 实例化对象
        Person p2 = new Person("李四", 31);      // 实例化对象
        Person p3 = new Person("王五", 32);      // 实例化对象
        System.out.println("------------ 修改之前 ----------------");
        p1.info();                               // 输出信息
        p2.info();                               // 输出信息
        p3.info();                               // 输出信息
        System.out.println("------------ 修改之后 ----------------");
        Person.setCountry("B城");                // 使用类名称调用static方法
        p1.info();                               // 输出信息
        p2.info();                               // 输出信息
        p3.info();                               // 输出信息
    }
};
```

程序运行结果：

```
------------ 修改之前 ----------------
姓名：张三，年龄：30，城市：A城
姓名：李四，年龄：31，城市：A城
姓名：王五，年龄：32，城市：A城
------------ 修改之后 ----------------
姓名：张三，年龄：30，城市：B城
姓名：李四，年龄：31，城市：B城
姓名：王五，年龄：32，城市：B城
```

在以上的程序中，Person 类将所有的属性都进行了封装，所以要想设置属性就必须使用 setter()方法，但是这里的 setter 方法是使用 static 声明的，所以可以直接使用类名称调用。

另外，在此处需要说明的是，非 static 声明的方法可以去调用 static 声明的属性或方法。但是 static 声明的方法是不能调用非 static 类型声明的属性或方法的。

范例：错误的代码，使用 static 方法调用非 static 方法及属性

```
class Person {
    private static String country = "A城";      // 定义静态属性
    private String name = "HELLO" ;
    public static void sFun(String c) {
        System.out.println("name = " + name) ; // 错误，不能调用非static属性
        fun() ;                                 // 错误，不能调用非static方法
    }
    public void fun() {
        System.out.println("World") ;
    }
};
```

程序编译时出现以下错误：

```
StaticDemo05.java:5: non-static variable name cannot be referenced from a
static context
        System.out.println("name = " + name) ;  // 错误，不能调用非static属性
                           ^
StaticDemo05.java:6: non-static method fun() cannot be referenced from a
staticcontext
        fun() ;         // 错误，不能调用非static方法
        ^
2 errors
```

从程序编译时所出现的错误信息可知，static 是不能调用任何非 static 内容的，这一点实际上也不难理解，因为在程序中所有的属性和方法必须在对象开辟堆内存之后才可以调用，而 static 类型的方法在对象未被实例化时就可以被类名调用。

5.10.3 static 的相关应用

下面为读者讲解两个 static 的相关应用实例，这两个应用将作为后续程序讲解的基础。

从前面的讲解可以知道 static 属性是所有对象共享的，那么就可以使用 static 属性统计一个类到底产生了多少个实例化对象。

范例：统计一个类产生了多少个实例化对象

```
class Demo {
    private static int count = 0;          // 所有对象共享此属性
    public Demo() {
        count++;                           // 只要有对象产生就应该自增
        System.out.println("产生了" + count + "个对象！");
    }
};
public class StaticDemo06 {
    public static void main(String args[]) {
        new Demo();                        // 增加新对象
        new Demo();                        // 增加新对象
        new Demo();                        // 增加新对象
    }
};
```

程序运行结果：

```
产生了1个对象！
产生了2个对象！
产生了3个对象！
```

从之前的定义中读者应该清楚，只要一有实例化对象产生，则一定会调用其中的构造方法，所以在构造方法中将 static 属性进行自增，这样就能够计算出一个类中到底有多少个

实例化对象。

可以使用 static 为对象进行自动的编名操作，此操作与上面代码类似。

范例：为对象自动进行编名

```
class Demo {
    private String name;                    // 保存名字
    private static int count = 0;           // 所有对象共享此属性
    public Demo() {
        count++;                            // 只要有对象产生就应该自增
        this.name = "DEMO-" + count;        // 自动进行编名操作
    }
    public Demo(String name) {              // 直接定义名字
        this.name = name ;
    }
    public String getName() {
        return this.name;
    }
};
public class StaticDemo07 {
    public static void main(String args[]) {
        System.out.println(new Demo().getName());
        System.out.println(new Demo("LXH").getName());
        System.out.println(new Demo().getName());
        System.out.println(new Demo("MLDN").getName());
        System.out.println(new Demo().getName());
    }
};
```

程序运行结果：

```
DEMO-1
LXH
DEMO-2
MLDN
DEMO-3
```

以上代码的运行结果并不是很难理解，通过 static 属性可以为类中的属性进行自动的编名操作，这一点的应用在以后的多线程讲解中读者会有所发现。

5.10.4 理解 main 方法

读者可以发现，之前使用的 main 方法的定义中，一直有 static 关键字的存在，那么主方法每个参数的含义是什么呢？下面为每个参数的含义。

- public：表示此方法可以被外部调用。

- static：表示此方法可以由类名称直接调用。
- void：主方法是程序的起点，所以不需要任何的返回值。
- main：系统规定好默认调用的方法名称，执行时默认找到 main 方法名称。
- String args[]：表示的是运行时的参数。参数传递的形式为“Java 类名称 参数 1 参数 2 参数 3…”。

范例：验证参数传递，输入的必须是 3 个参数，否则程序退出

```
public class StaticDemo08 {
    public static void main(String args[]) {
        if (args.length != 3) {                          // 所有的输入参数都保存在
                                                            args数组之中
            System.out.println("输入参数的个数不足3个，程序退出~");
                                                         // 参数不足，则程序退出
            System.exit(1);                              // 直接退出本程序
        }
        for (int x = 0; x < args.length; x++) {          // 输入参数正确，则将全部的
                                                            内容打印出来
            System.out.println("第" + (x + 1) + "个参数：" + args[x]);
        }
    }
};
```

程序执行命令：

```
java StaticDemo08 one two three
```

程序运行结果：

```
第1个参数：one
第2个参数：two
第3个参数：three
```

在程序中如果运行时没有输入参数，则会提示“输入参数的个数不足 3 个，程序退出～”。程序中的 System.exit(1)表示系统退出，只要在 exit()方法中设置一个非零的数字，则系统执行到此语句之后将退出程序。

如果读者在输入参数时希望参数中间加有空格，如“Hello World”、“Hello MLDN”等类的信息，则在输入参数时直接加上“""”即可进行整体的输入，程序如下所示。

程序执行命令：

```
java StaticDemo08 "Hello World!!!" "Hello MLDN" "Li Xing Hua"
```

程序运行结果：

```
第1个参数：Hello World!!!
第2个参数：Hello MLDN
第3个参数：Li Xing Hua
```

因为使用了“""”括起来，所以即使中间出现了空格，也按照一个参数进行输入。

提示：使用 static 定义方法的补充。

在讲解数组与方法时，本书给过读者一个方法的基本格式，而且一直强调，如果一个方法要由主方法直接调用，则必须按以下格式声明：“public static 方法的返回值类型 方法名称(参数列表){}”。

相信读者应该对此还有印象，那么此定义读者现在应该可以理解了，这是因为主方法是静态方法，而静态方法是不能调用非静态方法的，所以之前的方法声明处才必须加上 static 关键字。

5.11 代 码 块

代码块本身并不是很难理解的概念，实际上之前也一直在使用。所谓的代码块是指使用“{}”括起来的一段代码，根据位置不同，代码块可以分为普通代码块、构造块、静态代码块、同步代码块 4 种，其中，同步代码块本书将在多线程部分进行讲解，下面先来观察其他 3 种代码块。

5.11.1 普通代码块

普通代码块就是指直接在方法或是语句中定义的代码块，如下面代码。

范例：在方法中定义普通代码块

```
public class CodeDemo01 {
    public static void main(String args[]) {
        {                           // 定义一个普通代码块
            int x = 30 ;            // 定义局部变量
            System.out.println("普通代码块 --> x = " + x);
        }
        int x = 100 ;               // 与局部变量名称相同
        System.out.println("代码块之外 --> x = " + x);
    }
};
```

程序运行结果：

```
普通代码块 --> x = 30
代码块之外 --> x = 100
```

5.11.2 构造块

构造代码块是直接写在类中的代码块。

范例：定义构造块

```
class Demo {
    {                           // 定义构造块
```

```
        System.out.println("1、构造块。") ;
    }
    public Demo(){          // 定义构造方法
        System.out.println("2、构造方法。");
    }
}
public class CodeDemo02 {
    public static void main(String args[]) {
        new Demo() ;        // 实例化对象
        new Demo() ;        // 实例化对象
        new Demo() ;        // 实例化对象
    }
};
```

程序运行结果：

```
1、构造块。
2、构造方法。
1、构造块。
2、构造方法。
1、构造块。
2、构造方法。
```

本程序在主方法中产生了 3 个实例化对象，从输出的结果可以发现，构造块优先于构造方法执行，而且每次实例化对象时都会执行构造块中的代码，会执行多次。

5.11.3　静态代码块

静态代码块是使用 static 关键字声明的代码块。

范例：静态代码块

```
class Demo {
    {                           // 定义构造块
        System.out.println("1、构造块。") ;
    }
    static{                     // 定义静态代码块
        System.out.println("0、静态代码块") ;
    }
    public Demo(){              // 定义构造方法
        System.out.println("2、构造方法。");
    }
}
public class CodeDemo03 {
    static{                     // 在主方法所在类中定义静态块
```

```
        System.out.println("在主方法所在类中定义的代码块。") ;
    }
    public static void main(String args[]) {
        new Demo() ;          // 实例化对象
        new Demo() ;          // 实例化对象
        new Demo() ;          // 实例化对象
    }
};
```

程序运行结果：

```
在主方法所在类中定义的代码块。
0、静态代码块
1、构造块。
2、构造方法。
1、构造块。
2、构造方法。
1、构造块。
2、构造方法。
```

从程序的运行结果中可以发现，静态代码块优先于主方法执行，而在类中定义的静态代码块会优先于构造块执行，而且不管有多少个对象产生，静态代码块只执行一次。

提示：静态代码块的妙用。

从之前的代码中可以发现，静态代码块优先于主方法执行，那么就可以直接使用静态代码块而不使用主方法向屏幕上打印“Hello World!”了，观察以下代码：

代码一：使用静态块代替主方法

```
public class CodeDemo04 {
    static{
        System.out.println("Hello World!!!") ; // 直接在静态块中打印信息
    }
};
```

程序运行结果：

```
Hello World!!!
Exception in thread "main" java.lang.NoSuchMethodError: main
```

从程序的运行结果中可以发现，已经成功地打印出了“Hello World!!!”，但是到最后又出现了找不到主方法的异常，那么该如何解决呢？实际上最好的做法是，在程序输出完“HelloWorld！！！”之后直接让程序退出即可。

代码二：修改之前的代码

```
public class CodeDemo04 {
    static{
        System.out.println("Hello World!!!") ; // 直接在静态块中打印信息
        System.exit(1) ;                      // 直接退出，就可以避免程序寻找主方法
```

```
    }
};
```

以上程序即可完成所需要的功能，但是此程序本身并不具备任何的意义。

5.12　构造方法私有化

5.12.1　问题的引出

类的封装性不只体现在对属性的封装上，实际上方法也是可以被封装的，当然，在方法封装中也包含了对构造方法的封装。如以下代码，就是对构造方法进行了封装。

```
class Singleton {
    private Singleton() {              // 此处将构造方法进行封装
    }
    public void print(){               // 打印信息
        System.out.println("Hello World!!!");
    }
}
```

从之前讲解过的知识中可以清楚地知道，一个类要想使用，则必须有实例化对象的产生，而且现在要是想调用 Singleton 类中的 print()方法，则一定要首先产生 Singleton 的实例化对象，但是由于此时构造方法被私有化了，所以如果按照如下的程序编写，则肯定会出现错误。

范例：错误的代码，直接实例化 Singleton 类对象

```
public class SingleDemo02 {
    public static void main(String args[]) {
        Singleton s1 = null ;          // 可以声明对象
        s1 = new Singleton() ;         // 错误，无法实例化
        s1.print();
    }
};
```

程序编译时会出现以下错误：

```
SingleDemo02.java:11: Singleton() has private access in Singleton
            s1 = new Singleton() ;          // 错误，无法实例化
                 ^
1 error
```

从以上的错误提示中读者应该可以发现，程序是在使用 new 关键字实例化对象时出现了错误，而对于声明对象则没有任何的错误。那么该如何解决以上的问题呢？

封装是指一切都外部不可见，那么就意味着在外部根本就无法调用被封装的构造方法，既然外部不能调用，那么在内部呢？

范例：在内部产生 Singleton 的对象

```
class Singleton {
    Singleton instance = new Singleton() ;          // 在内部产生本类的实例化对象
    private Singleton() {                           // 此处将构造方法进行封装
    }
    public void print(){                            // 信息输出
        System.out.println("Hello World!!!");
    }
}
```

以上的程序编译后，并不会产生任何的问题，也就是说，此时的重点在于如何能够将内部的 instance 对象传递到类的外部去，这样外部就可以通过 instance 来实例化 Singleton 的对象。那么对象到底该如何取呢？下面对之前学习过的知识作进一步的分析。

5.12.2 问题的解决

学习 static 时曾经讲过，static 类型的属性可以由类名称直接调用，所以此时可以将 instance 属性声明为 static 类型，这样就可以通过类名称直接调用。

范例：将 instance 声明为 static 类型

```
class Singleton {
    static Singleton instance = new Singleton() ;// 在内部产生本类的实例化对象
    private Singleton() {                        // 此处将构造方法进行封装
    }
    public void print(){                         // 输出信息
        System.out.println("Hello World!!!");
    }
}
public class SingleDemo04 {
    public static void main(String args[]) {
        Singleton s1 = null ;
        s1 = Singleton.instance ;                // 访问类中的静态属性
        s1.print();
    }
};
```

程序运行结果：

```
Hello World!!!
```

由运行结果可以发现，程序成功地取得了 Singleton 的实例化对象并调用了其中的 print() 方法。但是这样做本身也存在着问题，因为类中的属性必须封装，所以此处应该将 instance 属性进行封装，而封装之后必须通过方法取得，但因为 instance 属性属于静态属性，所以此处必须声明一个静态方法，这样就可以被类名称直接调用。

范例：声明静态方法，取得 Singleton 类的实例

```
class Singleton {
    // 在内部产生本类的实例化对象，将属性封装
    private static Singleton instance = new Singleton();
    private Singleton() {                          // 此处将构造方法进行封装
    }
    public static Singleton getInstance(){ // 通过静态方法取得Singleton类的实例
        return instance;
    }
    public void print() {                          // 取得信息
        System.out.println("Hello World!!!");
    }
}
public class SingleDemo05 {
    public static void main(String args[]) {
        Singleton s1 = Singleton.getInstance();     // 访问类中的静态方法，取得对象实例
        Singleton s2 = Singleton.getInstance();     // 访问类中的静态方法，取得对象实例
        Singleton s3 = Singleton.getInstance();     // 访问类中的静态方法，取得对象实例
        s1.print();                                 // 输出信息
        s2.print();                                 // 输出信息
        s3.print();                                 // 输出信息
    }
};
```

程序运行结果：

```
Hello World!!!
Hello World!!!
Hello World!!!
```

5.12.3　程序的意义

从以上程序中可以发现虽然声明了 3 个 Singleton 对象，但是实际上所有的对象都只使用 instance 引用，也就是说，此时不管外面如何使用，最终结果也只是有一个实例化对象存在，如图 5-21 所示。

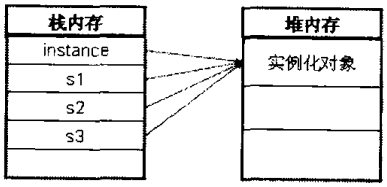


图 5-21　不管有多少个对象，实际上都只有一个实例

在设计模式中将这样的设计称为单例设计模式，即无论程序怎样运行，Singleton 类永远只会有一个实例化对象存在。

提示：单态的应用。

实际上这样的应用读者应该早就能有所了解了，读者应该都很清楚在 Windows 中有一个回收站程序，除了桌面上的回收站之外，每个硬盘上都有一个回收站，实际上每个硬盘的回收站和桌面上的回收站都是同一个，那么也就是说在整个操作系统上只有一个回收站实例，各个地方只是引用此实例而已。

在此读者只需要记住，只要将构造方法私有化，就可以控制实例化对象的产生。在后面讲解类库时，会讲解单例模式在 Java 中的一些应用。

5.13 对 象 数 组

对象数组的概念本身并不复杂，所谓的对象数组，就是指包含了一组相关的对象的数组，但是在对象数组的使用中读者一定要清楚一点：数组一定要先开辟空间，但是因为其是引用数据类型，所以数组中的每一个对象都是 null 值，则在使用时数组中的每一个对象必须分别进行实例化操作。

【格式 5-7　对象数组的声明】

```
类 对象数组名称[] = new 类[数组长度] ;
```

范例：声明一个对象数组

```
class Person {
    private String name;                        // 姓名属性
    public Person(String name) {                // 通过构造方法设置内容
        this.name = name;                       // 为姓名赋值
    }
    public String getName() {                   // 取得姓名
        return this.name;
    }
}
public class ObjectArrayDemo01 {
    public static void main(String args[]) {
        Person per[] = new Person[3];           // 声明一个对象数组，里面有3个对象
        // 对象数组初始化之前，每一个元素都是默认值
        System.out.println("================ 数组声明 ================");
        for (int x = 0; x < per.length; x++) {
            System.out.print(per[x] + "、");// 循环输出
        }
        // 分别为数组中的每个元素初始化，每一个都是对象，都需要单独实例化
        per[0] = new Person("张三");            // 实例化第1个元素
```

```
            per[1] = new Person("李四");                    // 实例化第2个元素
            per[2] = new Person("王五");                    // 实例化第3个元素
            System.out.println("\n================ 对象实例化
================");
            for (int x = 0; x < per.length; x++) {
                System.out.print(per[x].getName() + "、");
            }
        }
    };
```

程序运行结果：

```
================ 数组声明 ================
null、null、null、
================ 对象实例化 ================
张三、李四、王五、
```

与数组初始化的方式一样，对象数组也分为静态初始化和动态初始化。以上操作属于数组的动态初始化，静态初始化代码如下。

范例：对象数组的静态初始化

```
class Person {
    private String name;                                    // 姓名属性
    public Person(String name) {                            // 通过构造方法设置内容
        this.name = name;                                   // 为姓名赋值
    }
    public String getName() {                               // 取得姓名
        return this.name;
    }
}
public class ObjectArrayDemo02 {
    public static void main(String args[]) {
        // 声明一个对象数组，里面有3个对象，使用静态初始化方式
        Person per[] = { new Person("张三"), new Person("李四"), new Person("
        王五") };
        System.out.println("================ 数组输出 ================");
        for (int x = 0; x < per.length; x++) { // 循环输出
            System.out.print(per[x].getName() + "、");
        }
    }
};
```

程序运行结果：

```
================ 数组输出 ================
张三、李四、王五、
```

程序中在声明对象数组时采用了静态初始化的方式，然后采用循环的方式依次输出对象数组中的每一个元素。

提示：主方法中的 String args[]就是对象数组。

在主方法中，可以使用 String args[]接收初始化参数，实际上这里的 String 本身就是一个类，所以在主方法中的参数本身就是以对象数组的形式出现的。

5.14 内 部 类

5.14.1 内部类的基本定义

在类内部可定义成员变量与方法，而且在类内部也可以定义另一个类。如果在类 Outer 的内部再定义一个类 Inner，此时类 Inner 就称为内部类，而类 Outer 则称为外部类。

内部类可声明成 public 或 private。当内部类声明成 public 或 private 时，对其访问的限制与成员变量和成员方法完全相同。下面列出了内部类的定义格式：

【格式 5-8 内部类的声明格式】

```
标识符 class 外部类的名称{
    // 外部类的成员
    标识符 class 内部类的名称{          } 内部类
        // 内部类的成员
    }
}                                                  } 外部类
```

下面就根据以上的格式定义一个内部类。

范例：定义内部类

```
class Outer {                                          // 定义外部类
    private String info = "hello world!!!";            // 定义外部类的私有属性
    class Inner {                                      // 定义内部类
        public void print() {                          // 内部类的方法
            System.out.println(info);                  // 直接访问外部类私有属性
        }
    };
    public void fun() {                                // 定义方法
        new Inner().print();                           // 通过内部类对象调用方法
    }
};
public class InnerClassDemo01 {
    public static void main(String args[]) {
        new Outer().fun();                             // 调用外部类的fun()方法
    }
}
```

程序运行结果：

```
hello world!!!
```

从以上的程序中可以清楚地发现，Inner 类作为 Outer 类的内部类存在，并且在外部类的 fun()方法之中直接实例化内部类的对象并调用方法 print()，但是从以上代码中可以明显地发现，内部类的存在实际上已经破坏了一个类的基本结构，因为类是由属性及方法组成的，所以这是内部类的一个缺点，那么内部类有哪些优点呢？如果现在把内部类拿到外面来就能发现内部类的优点，如下面的代码。

范例：将内部类拿到外部

```
class Outer {
    private String info = "hello world!!!";
    public String getInfo() {                          // 取得私有属性
        return this.info;
    }
    public void fun() {
        new Inner(this).print();                       // 将当前对象传递到Inner中
    }
};
class Inner {
    private Outer out = null;                          // 声明外部类对象
    public Inner(Outer out) {                          // 接收外部类对象实例
        this.out = out;
    }
    public void print() {                              // 打印外部类的私有属性
        System.out.println(out.getInfo());
    }
};
public class InnerClassDemo02 {
    public static void main(String args[]) {
        new Outer().fun();                             // 调用Outer类的fun()
    }
};
```

程序运行结果：

```
hello world!!!
```

以上程序完成了与内部类同样的功能，但是代码明显比之前的更加复杂，所以内部类的唯一好处就是可以方便地访问外部类中的私有属性。

5.14.2　使用 static 定义内部类

使用 static 可以声明属性或方法，而使用 static 也可以声明内部类，用 static 声明的内部类变成了外部类，但是用 static 声明的内部类不能访问非 static 的外部类属性。

范例：使用 static 声明内部类

```
class Outer {
    private static String info = "hello world!!!";
    static class Inner {                          // 使用static定义内部类为外部类
        public void print() {
            System.out.println(info);
        }
    };
};
public class InnerClassDemo03 {
    public static void main(String args[]) {
        new Outer.Inner().print();                // 访问内部类
    }
}
```

程序运行结果：

```
hello world!!!
```

以上程序中将 info 属性定义成了 static 类型，这样程序中就可以通过 static 声明的内部类直接访问此 static 属性，当然，如果此时 info 属性不是 static 类型，则编译时将出现以下错误：

```
InnerClassDemo03.java:7: non-static variable info cannot be referenced from
a static context
                        System.out.println(info);
                                           ^
1 error
```

5.14.3 在外部访问内部类

一个内部类除了可以通过外部类访问，也可以直接在其他类中进行调用，调用的基本格式为：

【格式 5-9　在外部访问内部类】

```
外部类.内部类 内部类对象 = 外部类实例.new 内部类() ;
```

以上的操作格式中，首先要找到外部类的实例化对象之后，才可以通过外部类的实例化对象去实例化内部类的对象。

提示：观察内部类的 class 文件。

内部类定义之后，生成的 class 文件是以 Outer$Inner.class 的形式存在的，在 Java 中只要在文件中存在$，则在程序中应将其替换为“.”。

范例：在外部访问内部类

```
class Outer {                                     // 定义外部类
    private String info = "hello world!!!";
```

```
    class Inner {                                   // 定义内部类
        public void print() {
            System.out.println(info);               // 访问外部类的私有属性
        }
    };
};
public class InnerClassDemo04 {
    public static void main(String args[]) {
        Outer out = new Outer() ;                   // 实例化外部类对象
        Outer.Inner in = out.new Inner() ;          // 实例化内部类对象
        in.print() ;                                // 调用内部类方法
    }
}
```

程序运行结果：

```
hello world!!!
```

5.14.4　在方法中定义内部类

也可以在方法中定义一个内部类，但是在方法中定义的内部类不能直接访问方法中的参数，如果方法中的参数要想被内部类所访问，则参数前必须加上 final 关键字。

范例：在方法中定义内部类

```
class Outer{                                        // 定义外部类
    private String info = "hello world!!!" ;
    public void fun(final int temp){                // 参数要被访问必须用final声明
        class Inner{                                // 在方法中定义内部类
            public void print(){
                System.out.println("类中的属性：" +info) ;
                System.out.println("方法中的参数：" +temp) ;
            }
        };
        new Inner().print() ;
    }
};
public class InnerClassDemo05 {
    public static void main(String args[]) {
        new Outer().fun(30) ;                       // 调用外部类方法
    }
}
```

程序运行结果：

```
类中的属性：hello world!!!
方法中的参数：30
```

5.15 实 例 讲 解

5.15.1 系统登录

在各个应用系统中，读者应该可以看到用户登录程序的影子，下面就模拟一个简单的用户登录程序，为方便读者的理解，本书使用初始化参数的方式输入用户名和密码。在本程序中假设用户名为 lixinghua，密码为 mldn。

程序分析：

既然使用初始化参数的方式输入用户名和密码，所以在程序运行之前首先必须判断输入的参数个数是否合法，如果不合法，则必须提示用户的程序执行错误，并退出程序，如果用户已经正确地输入了参数，则可以进行用户名及密码的验证。如果信息正确则显示“欢迎 xxx 光临！”，否则显示“错误的用户名及密码”。

实现一：基本功能实现

```
public class LoginDemo01 {
    public static void main(String args[]) {
        if (args.length != 2) {                                    // 判断参数个数
            System.out.println("输入的参数不正确，系统退出！"); // 给一个正确的格式
            System.out.println("格式：java LoginDemo01  用户名 密码");
            System.exit(1);                                        // 系统退出
        }
        String name = args[0];                                     // 取出用户名
        String password = args[1];                                 // 取出密码
        if (name.equals("lixinghua")
                && password.equals("mldn")) {                      // 验证
            System.out.println("欢迎" + name + "光临！");
        } else {
            System.out.println("错误的用户名和密码！");
        }
    }
}
```

程序运行结果（在命令行中输入参数）：

（1）没有输入参数

```
输入的参数不正确，系统退出！
格式：java LoginDemo01 用户名 密码
```

（2）输入的用户名及密码错误

```
错误的用户名及密码！
```

（3）输入正确的用户名及密码

```
欢迎lixinghua光临！
```

以上程序已经完成了一个最基本的功能，但是读者现在思考一下，本程序中是否存在问题呢？以上程序并不是一个很好的程序，程序在主方法中编写了大量的代码，而主方法就好比是一个客户端，现在要求客户端自己比较、自己输出显然不合理，最简单的做法就是：客户端只要得到最终的结果集合，至于中间是如何判断的，它根本就没有必要去关心，就好比进楼门要刷卡一样，用户只需要返回刷卡成功或刷卡失败的信息即可，而不用关心里面是如何去运算的。所以此处最好的做法是，单独做一些类，并使用这些类封装具体的判断过程。如图5-22所示为刷卡流程。

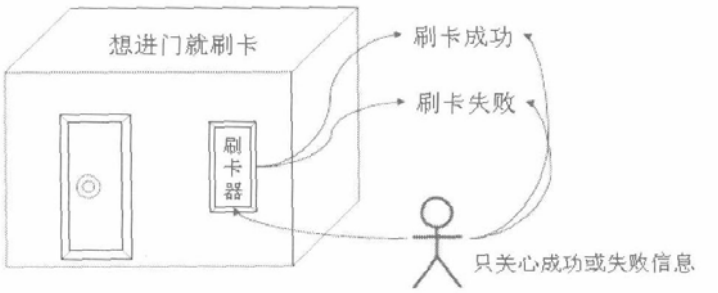


图5-22　刷卡流程

范例：修改以上的程序，将信息的判断过程形成若干类

```
class Check {
    public boolean validate(String name,
            String password) {                    // 执行具体的验证操作
        if (name.equals("lixinghua")
                && password.equals("mldn")) {     // 验证
            return true;                          // 登录信息正确，返回true
        } else {
            return false;                         // 登录信息失败，返回false
        }
    }
}
class Operate {                                   // 本类只是调用具体的验证操作
    private String info[];                        // 定义一个数组属性，用于接收全部输入参数

    public Operate(String info[]) {
        this.info = info;                         // 通过构造方法取得全部的输入参数

    }
    public String login() {
        Check check = new Check();                // 实例化Check对象，用于检查信息
        this.isExit();                            // 判断输入的参数格式，用来决定程序是否退出

        String name = this.info[0];               // 取出用户名
        String password = this.info[1];           // 取出密码
```

```
        String str = null;                          // 声明一个变量，用于接收信息
        if (check.validate(name, password)) {       // 登录验证
            str = "欢迎" + name + "光临！";
        } else {
            str = "错误的用户名和密码！";
        }
        return str;                                 // 返回信息给用户
    }
    public void isExit() {
        if (this.info.length != 2) {                // 判断参数个数
            System.out.println("输入的参数不正确，系统退出！"); // 给一个正确的格式
            System.out.println("格式：java LoginDemo02 用户名 密码");
            System.exit(1);                         // 系统退出
        }
    }
}
public class LoginDemo02 {
    public static void main(String args[]) {
        Operate oper = new Operate(args) ;          // 实例化操作类对象
        System.out.println(oper.login()) ;          // 取得验证信息
    }
}
```

程序的运行结果与之前相似，从本程序中可以发现以下几点：

- 主方法处代码较少，因为是客户端，所以要方便客户使用。
- Check 类的主要功能就是验证操作，只需要传入用户名和密码即可完成验证。
- Operate 类的主要功能就是封装 Check 类的操作，并把 Check 的信息返回给调用处。

以上程序的调用关系如图 5-23 所示。

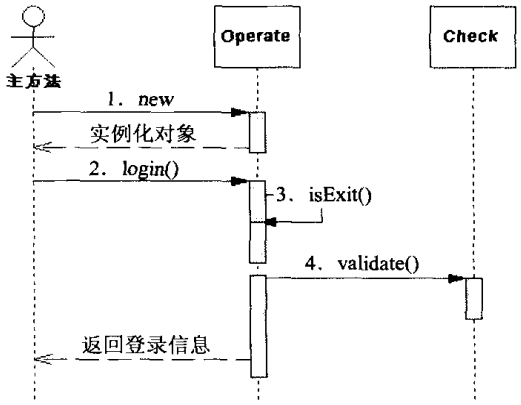


图 5-23　登录验证程序时序图

提问：怎么去开发一个程序呢？

该如何去开发一个程序呢？很多的知识都已经掌握了，可是拿到一个题目之后还是无从下手。

回答：基础扎实、循环分析。

首先，这是很多初学者都会遇到的问题，因为代码的开发是一个基础熟练过程的积累，所有的基础知识运用得越熟练，代码的开发速度越快，代码开发多了，思路也就有了。而且拿到一个问题之后，如果觉得分析类有困难，可以先暂时不分析，先把基本的功能做完，做完之后对一些输入的数据进行验证，再把主方法中的代码尽可能减少，然后再去考虑代码的可重用性，几次下来之后代码就会更好一些。

必须提醒读者的是，类的开发是一个渐进的过程，当发现代码中重复地进行复制/粘贴时，就可以考虑将之形成一个类进行调用。

5.15.2　单向链表实现（1）

链表是在数据结构中经常见到的一种形式，实际上在 Java 中也可以通过引用传递的方式进行实现，本节为读者简单介绍在 Java 中实现链表的基本形式的方法。

在讲解链表之前，首先简单介绍链表的基本概念。所谓的链表就好像火车车厢一样，从火车头开始，每一节车厢之后都连着后一节车厢，如图 5-24 所示。

从图 5-24 中可以发现，每一节车厢就相当于一个节点，每一个节点除了要保存自己的内容外，还要保存下一个节点的引用，那么这样的节点该如何通过程序表示呢？节点类设计如图 5-25 所示。

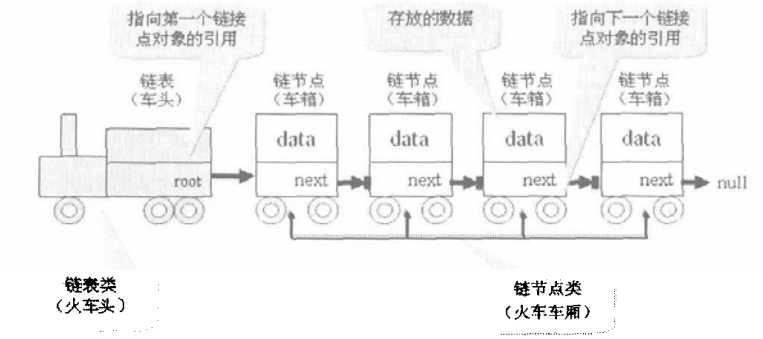


图 5-24　单向链表（只能从前向后找）

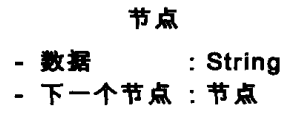


图 5-25　节点类设计

从图 5-25 类的设计中可以清楚地发现，要想清楚地表示出一个节点之后还有另外一个节点，可以在一个节点的内部存放下一个节点的引用。

范例：节点类

```
class Node {
    private String data;                          // 保存节点内容
    private Node next;                            // 保存下一个节点
    public Node(String data) {                    // 构造方法设置节点内容
        this.data = data;
    }
    public String getData() {                     // 得到节点内容
        return this.data;
```

```
    }
    public void setNext(Node next) {                    // 设置下一个节点
        this.next = next;
    }
    public Node getNext() {                             // 取得下一个节点
        return this.next;
    }
};
```

从以上代码的类中可以清楚地发现，每一个 Node 节点都有下一个 Node 对象的引用，那么具体的引用该如何设计呢？具体代码如下。

范例： 设置节点

```
public class LinkDemo01 {
    public static void main(String args[]) {
        Node root = new Node("火车头");          // 定义根节点
        Node n1 = new Node("车厢-A");            // 定义第1个车厢
        Node n2 = new Node("车厢-B");            // 定义第2个车厢
        Node n3 = new Node("车厢-C");            // 定义第3个车厢
        root.setNext(n1);                        // 火车头的下一个节点是第1个车厢
        n1.setNext(n2);                          // 第1个车厢的下一个节点是第2个车厢
        n2.setNext(n3);                          // 第2个车厢的下一个节点是第3个车厢
    }
}
```

从以上代码中可以清楚地发现，每一个节点都可以设置下一个节点，但是到第 3 个节点之后因为其不再有后续节点，所以不再单独设置关系。

节点全部设置完成之后，该怎么输出全部的节点呢？此时的节点很多，所以肯定要用循环的方式输出，基本思路是：判断一个节点之后是否还有后续节点，如果存在后续节点，则输出；如果不存在则不输出，所以输出可以使用方法的递归调用完成（递归调用就是自己调用自己），具体代码如下。

范例： 简单链表的设置及输出

```
public class LinkDemo01 {
    public static void main(String args[]) {
        Node root = new Node("火车头");          // 定义根节点
        Node n1 = new Node("车厢-A");            // 定义第1个车厢
        Node n2 = new Node("车厢-B");            // 定义第2个车厢
        Node n3 = new Node("车厢-C");            // 定义第3个车厢
        root.setNext(n1);                        // 火车头的下一个节点是第1个车厢
        n1.setNext(n2);                          // 第1个车厢的下一个节点是第2个车厢
        n2.setNext(n3);                          // 第2个车厢的下一个节点是第3个车厢
        printNode(root) ;                        // 从根节点输出
    }
```

```
    public static void printNode(Node node){
        System.out.print(node.getData()+"\t") ;// 输出数据
        if(node.getNext()!=null){                    // 判断节点是否为空
            printNode(node.getNext()) ;              // 继续向下打印
        }
    }
}
```

以上程序中的 printNode()方法就是采用了递归的调用形式，有时也可以把这种输出形式称为迭代输出。

5.15.3　单向链表实现（2）

5.15.2 节实现了一个简单单向链表，如果程序要按照上面的方式操作肯定会很麻烦，因为要由用户手工去处理各个节点的关系，这样肯定是不行的，所以最好将节点的操作进行封装，这样用户使用起来就会比较方便。在本节将为读者介绍如何对节点的操作进行封装。假设现在的节点操作有增加数据、查找数据、删除数据 3 种。如果要删除节点，则直接修改上一个节点的引用即可，如图 5-26 所示。

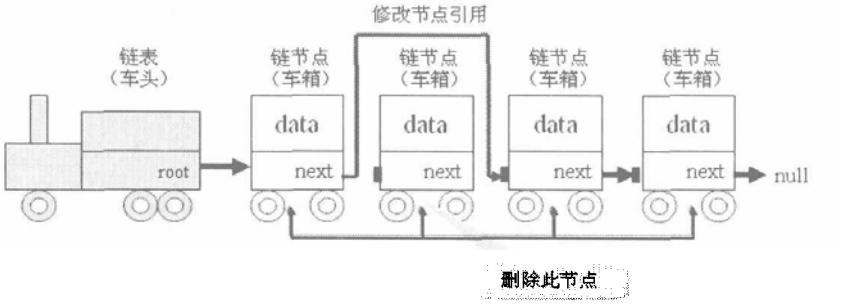


图 5-26　删除节点

本节为了操作代码方便，使用内部类的形式完成。

范例：链表操作

```
class Link{
    class Node{                                  // 把节点类定义成内部类
        private String data ;
        private Node next ;
        public Node(String data){
            this.data = data ;
        }
        public void add(Node newNode){           // 增加一个add操作
            if(this.next==null){                 // 判断下一个节点是否为空
                this.next = newNode ;            // 如果下一个节点为空，则把新节点设置
                                                    在next位置上
            }else{                               // 如果不为空，则表示还有下一个节点
                this.next.add(newNode) ;         // 在下一个位置处增加
```

```
            }
        }
        public void print(){
            System.out.print(this.data + "\t") ;
            if(this.next!=null){                    // 如果下一个节点不为空，则继续打印
                this.next.print() ;                 // 让下一个节点输出
            }
        }
        public boolean search(String data){// 内部定义搜索方法
            if(data.equals(this.data)){             // 判断当前节点的名字是否与查找的一致
                return true ;                       // 如果一致，则返回true
            }else{                                  // 继续判断下一个
                if(this.next!=null){                // 下一个节点存在，则继续查找
                    return this.next.search(data) ;// 返回下一个的查询结果
                }else{
                    return false ;                          // 节点不存在，返回false
                }
            }
        }
        public void delete(Node previous,String data){ // 删除节点
            if(data.equals(this.data)){                    // 找到了匹配节点
                previous.next = this.next ;                // 空出当前节点
            }else{
                if(this.next!=null){
                    this.next.delete(this,data) ;          // 继续向下找
                }
            }
        }
    };
    private Node root ;                             // 表示根节点
    public void addNode(String data){               // 增加节点的方法
        Node newNode = new Node(data) ;             // 建立一个新的节点
        if(this.root==null){                        // 没有根节点
            this.root = newNode ;                   // 将第一个节点设置成根节点
        }else{
            this.root.add(newNode) ;                // 添加到合适的位置
        }
    }
    public void printNode(){                        // 打印全部的节点
        if(this.root!=null){                        // 判断是否存在根节点
            this.root.print() ;
        }
```

```
        }
        public boolean contains(String name){    // 判断元素是否存在
            return this.root.search(name) ;      // 调用Node类中的search()方法
        }
        public void deleteNode(String data){     // 删除节点
            if(this.contains(data)){             // 如果节点存在，则执行删除操作
                if(this.root.data.equals(data)){// 判断根节点是否满足要求
                    this.root = this.root.next ;// 将根节点之后的内容设置成根节点
                }else{
                    this.root.next.delete(root,data) ; // 删除节点
                }
            }
        }
    };
    public class LinkDemo02 {
        public static void main(String args[]) {
            Link l = new Link();
            l.addNode("A");                                  // 增加节点
            l.addNode("B");                                  // 增加节点
            l.addNode("C");                                  // 增加节点
            l.addNode("D");                                  // 增加节点
            l.addNode("E");                                  // 增加节点
            System.out.println("==================== 删除之前
==================");
            l.printNode();                                   // 输出节点
            l.deleteNode("C");                               // 删除节点
            l.deleteNode("D");                               // 删除节点
            System.out.println();
            System.out.println("==================== 删除之后
==================");
            l.printNode();
            System.out.println();
            System.out.println("查询节点：" + l.contains("A"));
        }
    }
```

程序运行结果：

```
==================== 删除之前 ==================
A   B   C   D   E
==================== 删除之后 ==================
A   B   E
查询节点：true
```

以上程序对要操作的节点类进行了包装，以后用户直接调用包装后的类，即可方便地

执行节点的增加、删除、查找操作。

提示：理解操作原理。

以上程序只要理解其操作原理即可，在实际的使用中，Java 已经为用户提供了大量的数据结构实现类，这些类都将在类集框架中为读者介绍。

5.16 本章要点

1．面向对象的三大特征为封装、继承、多态。

2．类与对象的关系：类是对象的模板，对象是类的实例，类只能通过对象才可以使用。

3．类的组成为属性和方法。

4．对象的产生格式：类名称 对象名称 = new 类名称()。

5．如果一个对象没有被实例化而直接使用，则使用时会出现空指向异常。

6．类属于引用数据类型，进行引用传递时，传递的是堆内存的使用权。

7．类的封装性：通过 private 关键字进行修饰，被封装的属性不能被外部直接调用，而只能通过 setter 或 getter 方法完成。只要是属性，则必须全部封装。

8．构造方法可以为类中的属性初始化，构造方法与类名称相同，无返回值类型声明，如果在类中没有明确地定义出构造方法，则会自动生成一个无参的什么都不做的构造方法，在一个类中的构造方法可以重载，但是每个类都至少有一个构造方法。

9．String 类在 Java 中较为特殊，String 可以使用直接赋值的方式，也可以通过构造方法进行实例化，前者只产生一个实例化对象，而且此实例化对象可以重用，后者将产生两个实例化对象，其中一个是垃圾空间，在 String 中比较内容时使用 equals 方法，而“==”比较的只是两个字符串的地址值。字符串的内容一旦声明则不可改变。

10．在 Java 中使用 this 关键字可以表示当前的对象，通过“this.属性”可以调用本类中的属性，通过“this.方法()”可以调用本类中的其他方法，也可以通过 this()的形式调用本类中的构造方法，但是调用时要求要放在构造方法的首行。

11．使用 static 声明的属性和方法可以由类名称直接调用，static 属性是所有对象共享的，所有对象都可以对其进行操作。

12．如果需要限制类对象的产生，则可以将构造方法私有化。

13．对象数组的使用要分为声明数组和为数组开辟空间两步。开辟空间后数组中的每个元素的内容都是 null。

14．内部类是在一个类的内部定义另外的一个类，使用内部类可以方便地访问外部类的私有操作。在方法中声明的内部类要想访问方法的参数，则参数前必须加上 final 关键字。

5.17 习 题

1．编写并测试一个代表地址的 Address 类，地址信息由国家、省份、城市、街道、邮

编组成，并可以返回完整的地址信息。

2．定义并测试一个代表员工的 Employee 类。员工属性包括“编号”、“姓名”、“基本薪水”、“薪水增长额”，还包括计算薪水增长额及计算增长后的工资总额的操作方法。

3．编写程序，统计出字符串“want you to know one thing”中字母 n 和字母 o 的出现次数。

4．设计一个 Dog 类，有名字、颜色、年龄等属性，定义构造方法来初始化类的这些属性，定义方法输出 Dog 信息，编写应用程序使用 Dog 类。

5．设计一个表示用户的 User 类，类中的变量有用户名、口令和记录用户个数的变量，定义类的 3 个构造方法（无参、为用户名赋值、为用户名和口令赋值）、获取和设置口令的方法和返回类信息的方法。

6．字符串操作：

（1）从字符串“Java 技术学习班 20070326”中提取开班日期。

（2）将“MLDN JAVA”字符串中的“Java”替换为“J2EE”。

（3）取出“Java 技术学习班 20070326”中的第 8 个字符。

（4）清除“Java 技术学习班 20070326”中的所有 0。

（5）清除“Java 技术学习班 20070326 MLDN 老师”中的所有空格。

（6）从任意给定的身份证号码中提取此人的出生日期。

7．编写一个公司员工类。

（1）数据成员：员工号、姓名、薪水、部门。

（2）方法：

① 利用构造方法完成设置信息。

- 单参，只传递员工号，则员工姓名：无名氏，薪水：0，部门：未定。
- 双参，传递员工号，姓名，则员工薪水为 1000，部门：后勤。
- 4 参，传递员工号、姓名、部门、薪水。
- 无参，则均为空值。

② 显示信息

8．构造一个银行账户类，类的构成包括如下内容：

（1）数据成员用户的账户名称、用户的账户余额（private 数据类型）。

（2）方法包括开户（设置账户名称及余额），利用构造方法完成。

（3）查询余额。

9．声明一个图书类，其数据成员为书名、编号（利用静态变量实现自动编号）、书价，并拥有静态数据成员册数、记录图书的总册数，在构造方法中利用此静态变量为对象的编号赋值，在主方法中定义对象数组，并求出总册数。

第 6 章　面向对象（高级篇）

通过本章的学习可以达到以下目标：

- ☑ 掌握继承的基本概念及实现。
- ☑ 掌握继承实现的各种限制。
- ☑ 掌握子类对象的实例化过程。
- ☑ 掌握方法覆写的概念及实现。
- ☑ 掌握 super 关键字的作用。
- ☑ 掌握抽象类与接口的基本概念及实际应用。
- ☑ 掌握对象多态性的作用。
- ☑ 掌握常见的设计模式。
- ☑ 掌握 Object 类的作用及其主要方法的作用。
- ☑ 掌握包装类的作用以及自动装箱和拆箱的操作。
- ☑ 掌握匿名内部类的使用。

在前面的章节中已经介绍了类的基本使用方法，对于面向对象的程序而言，它的精华在于类的继承，继承可以在现有类的基础之上进行功能的扩充。通过这种方式能快速地开发出新的类，而不需编写相同的程序代码，这也是程序代码再利用的概念。本节将介绍继承的概念及其实际的应用。**本章视频录像讲解时间为 4 小时 20 分钟，源代码在光盘对应的章节下。**

6.1　继承的基本概念

在讲解继承的基本概念之前，读者可以先想一想这样一个问题：现在假设有一个 Person 类，其中包括 name 与 age 两个属性，而另外一个 Student 类，需要包含 name、age、school 3 个属性，如图 6-1 所示，具体代码如下。

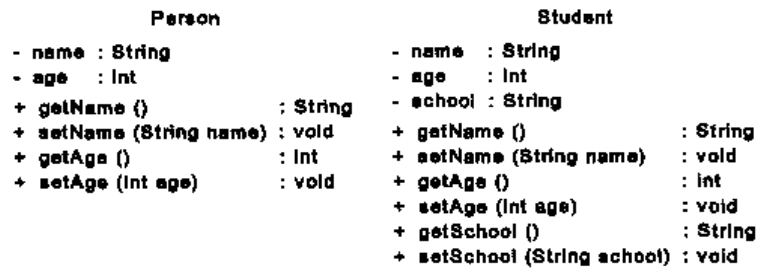


图 6-1　Person 和 Student 类

范例： Person 类

```
class Person{                                    // 定义Person类
```

```
    private String name ;                          // 定义name属性
    private int age ;                              // 定义age属性
    public String getName() {                      // 取得name属性
        return name;
    }
    public void setName(String name) {             // 设置name属性
        this.name = name;
    }
    public int getAge() {                          // 取得age属性
        return age;
    }
    public void setAge(int age) {                  // 设置age属性
        this.age = age;
    }
}
```

范例： Student 类

```
class Student{                                     // 定义Student类
    private String name ;                          // 定义name属性
    private int age ;                              // 定义age属性
    private String school ;                        // 定义school属性
    public String getName() {                      // 取得name属性
        return name;
    }
    public void setName(String name) {             // 设置name属性
        this.name = name;
    }
    public int getAge() {                          // 取得age属性
        return age;
    }
    public void setAge(int age) {                  // 设置age属性
        this.age = age;
    }
    public String getSchool() {                    // 取得school属性
        return school;
    }
    public void setSchool(String school) {         // 设置school属性
        this.school = school;
    }
}
```

从上面代码中可以发现 Person 类中已经存在有 name 和 age 两个属性，所以在 Student 类中不需要再重新声明这两个属性，这时就需要考虑是不是可以将 Person 类中的内容继续

保留到 Student 类中，也就引出了接下来所要介绍的类的继承概念。

在 Java 中如果要想实现继承，可以直接使用以下语法来表示：

【格式 6-1　类的继承格式】

```
class 父类{}                          // 定义父类
class 子类 extends 父类{}             // 使用extends关键字实现继承
```

下面按照以上格式进行继承的操作，具体代码如下。

范例：继承的实现

```
class Person {                                    // 定义Person类
    private String name;                          // 定义name属性
    private int age;                              // 定义age属性
    public String getName() {                     // 取得name属性
        return name;
    }
    public void setName(String name) {            // 设置name属性
        this.name = name;
    }
    public int getAge() {                         // 取得age属性
        return age;
    }
    public void setAge(int age) {                 // 设置age属性
        this.age = age;
    }
}
class Student extends Person {                    // Student是Person的子类
    // 此处任何代码都不编写
}
public class ExtDemo02 {
    public static void main(String args[]) {
        Student stu = new Student();
        stu.setName("张三");                      // 此时访问的方法是父类中的，子类中并没有定义
        stu.setAge(30);                           // 此时访问的方法是父类中的，子类中并没有定义
        System.out.println("姓名:" + stu.getName() + ",年龄:" + stu.getAge());
    }
}
```

程序运行结果：

```
姓名：张三，年龄：30
```

从以上程序中可以发现，在 Student 类中并没有定义任何的操作，但是通过继承的功能可以直接把父类中的操作拿到子类中使用。当然，子类也可以定义自己的属性或方法，如下面代码所示。

范例：通过子类扩展父类的功能

```
class Person {                                  // 定义Person类
    private String name;                        // 定义name属性
    private int age;                            // 定义age属性
    public String getName() {                   // 取得name属性
        return name;
    }
    public void setName(String name) {          // 设置name属性
        this.name = name;
    }
    public int getAge() {                       // 取得age属性
        return age;
    }
    public void setAge(int age) {               // 设置age属性
        this.age = age;
    }
}
class Student extends Person {                  // Student是Person的子类
    private String school;                      // 新定义的属性school
    public String getSchool() {                 // 取得school属性
        return school;
    }
    public void setSchool(String school) {      // 设置school属性
        this.school = school;
    }
}
public class ExtDemo03 {
    public static void main(String args[]) {
        Student stu = new Student();
        stu.setName("张三");                    // 此时访问的方法是父类中的，子类中并没有定义
        stu.setAge(30);                         // 此时访问的方法是父类中的，子类中并没有定义
        stu.setSchool("清华大学");              // 此时的方法是在子类中定义的
        System.out.println("姓名：" + stu.getName() + "，年龄：" + stu.getAge()
                + "，学校：" + stu.getSchool());
    }
}
```

程序运行结果：

```
姓名：张三，年龄：30，学校：清华大学
```

以上程序中的 Student 类扩充了 Person 类，增加了学校的属性及对应的 setter 和 getter 方法，也就是说此时 Student 类中已经存在了 3 个属性及 3 组 setter、getter 方法，如图 6-2 所示。

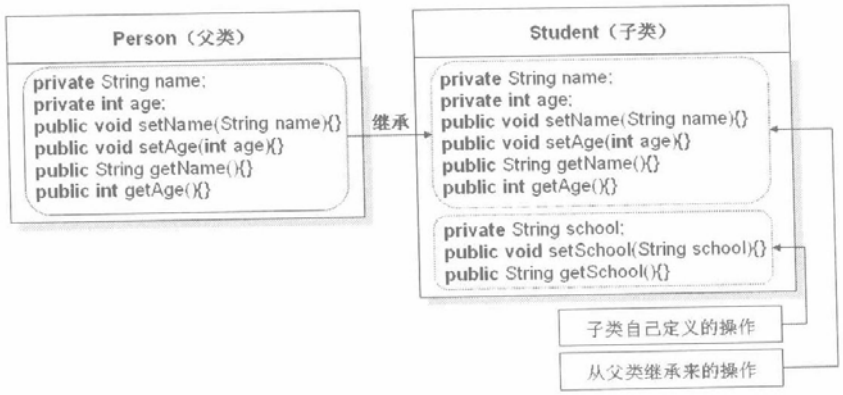


图 6-2　Person 与 Student 类的继承图

注意：只允许多层继承，而不能多重继承。

在 Java 中只允许单继承，不能使用多重继承，即一个子类只能继承一个父类。但是允许进行多层继承，即一个子类可以有一个父类，一个父类还可以有一个父类。分别如图 6-3 和图 6-4 所示。

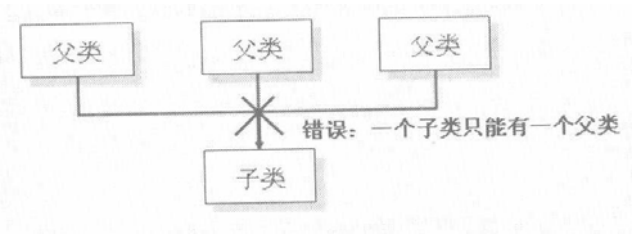


图 6-3　多重继承

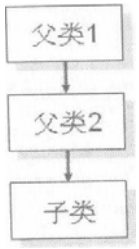


图 6-4　多层继承

范例： 错误的继承代码

```
class A{}
class B{}
// 同时继承两个类，代码错误
class C extends A,B{}
```

由上面的代码可以发现 C 类同时继承了 A 类与 B 类，也就是说 C 类同时继承了两个父类，这是不允许的。

范例： 多层继承

```
class A {}
class B extends A {}
class C extends B {}
```

由上面的代码可以发现 B 类继承了 A 类，而 C 类又继承了 B 类，也就是说 B 类是 A 类的子类，而 C 类则是 A 类的孙子类。

同样，在类图的表示中对于继承也是有明确要求的，可以使用如图 6-5 所示的图形表示类的继承关系。

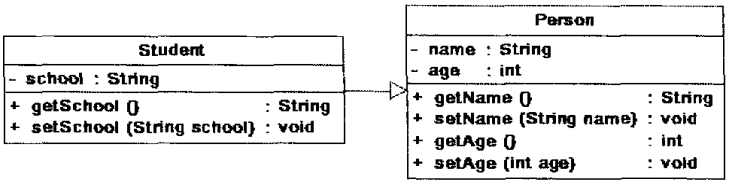
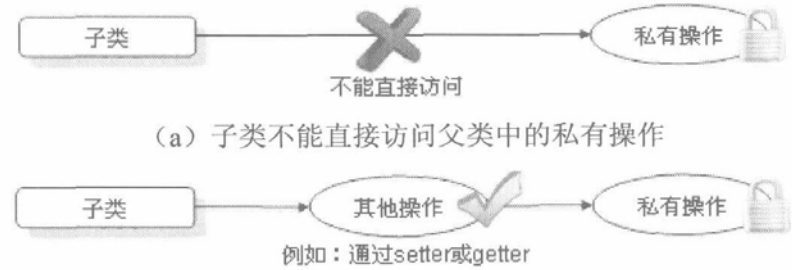


图 6-5　继承的类图表示

提示：继承的子类有时也称为派生类。

使用 extends 关键字可以实现继承的关系，但是这个关键字的本身含义是“扩展”，更准确地说是一个类扩展已有类的功能，在其他的书中也会经常把子类称为派生类。

在使用继承时应注意的是：子类是不能直接访问父类中的私有成员的，子类可以调用父类中的非私有方法，但是不能直接调用父类中的私有成员，如图 6-6 所示。



（a）子类不能直接访问父类中的私有操作

（b）子类可以通过其他操作间接访问父类中的私有操作

图 6-6　子类访问的限制

范例：子类访问限制

```
class Student extends Person {                                    // Student是
                                                                     Person的子类

    public void fun(){
        System.out.println("父类中的name属性：" + name) ;          // 错误，无法访问
        System.out.println("父类中的age属性：" + age) ;           // 错误，无法访问
    }
}
```

编译时错误：

```
ExtDemo04.java:19: name has private access in Person
            System.out.println("父类中的name属性：" + name);// 错误，无法访问
                                                    ^
ExtDemo04.java:20: age has private access in Person
            System.out.println("父类中的age属性：" + age) ;// 错误，无法访问
                                                    ^
2 errors
```

以上代码中，子类直接访问了父类中的私有属性，所以在编译时编译器会通知用户 name 和 age 属性无法访问，如果此时不直接访问 name 或 age 属性，而是直接通过 setter 或 getter 方法调用，则可以访问。

范例：通过 getter 方法间接取得属性内容

```
class Student extends Person {                                    // Student是
                                                                    Person的子类
    public void fun(){
        System.out.println("父类中的name属性：" + getName()) ;// 正确，间接访问
        System.out.println("父类中的age属性：" + getAge()) ; // 正确，间接访问
    }
}
```

6.2 继承的进一步研究

掌握了继承的基本概念及实现之后，下面将针对继承操作中的一些注意点做进一步的研究。

6.2.1 子类对象的实例化过程

在继承的操作中，对于子类对象的实例化也是有要求的，即子类对象在实例化之前必须首先调用父类中的构造方法后再调用子类自己的构造方法。子类对象实例化过程如图 6-7 所示。

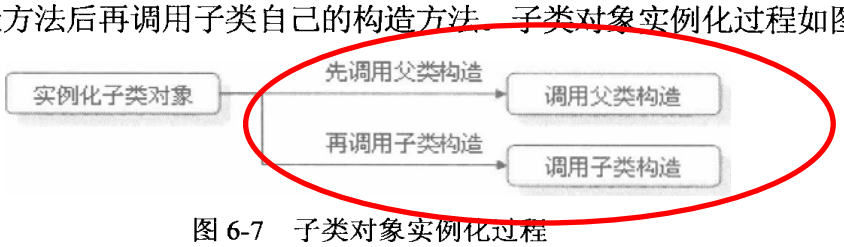


图 6-7 子类对象实例化过程

提示：子类对象的实例化过程与生活很类似。

在实际生活中肯定是要先有父母之后才能有孩子，孩子不可能凭空“蹦”出来，对于程序也是一样的，之所以会调用父类中的构造方法，就是要用父类的构造方法为父类中的属性初始化，就表示先有父类实例，然后才能产生子类实例。

范例：观察子类对象的实例化过程

```
class Person {                                          // 定义父类Person
    private String name;                                // 定义name属性
    private int age;                                    // 定义age属性
    public Person(){
        System.out.println("父类Person中的构造。") ; // 父类的构造方法
    }
    public String getName() {                           // 取得name属性
        return name;
    }
    public void setName(String name) {                  // 设置name属性
        this.name = name;
```

```
    }
    public int getAge() {                                    // 取得age属性
        return age;
    }
    public void setAge(int age) {                            // 设置age属性
        this.age = age;
    }
}
class Student extends Person {                               // Student是Person的子
                                                                类，扩展父类的功能
    private String school;                                   // 新定义的属性school
    public Student(){
        System.out.println("子类Student中的构造。") ;// 子类的构造方法
    }
    public String getSchool() {                              // 取得school属性
        return school;
    }
    public void setSchool(String school) {                   // 设置school属性
        this.school = school;
    }
}
public class InstanceDemo {
    public static void main(String args[]) {
        Student stu = new Student();
        stu.setName("张三");                                  // 此时访问的方法是父类的，
                                                                子类中并没有定义
        stu.setAge(30);                                      // 此时访问的方法是父类的，
                                                                子类中并没有定义
        stu.setSchool("清华大学");                            // 此方法是子类定义的
        System.out.println("姓名：" + stu.getName() + "，年龄：" + stu.getAge()
                + "，学校：" + stu.getSchool());
    }
}
```

程序运行结果：

```
父类Person中的构造。
子类Student中的构造。
姓名：张三，年龄：30，学校：清华大学
```

从程序的运行结果中可以清楚地发现，子类对象在实例化前会先默认调用父类中的构造方法。就好像没有父亲就没有孩子，所以在实例化子类对象前需要先将父类中的属性进行初始化。当然，对于以上的代码实际在子类的构造方法中隐含了一个super()的语法，代码如下所示：

```
class Student extends Person {                          // Student是Person的子
                                                           类，扩展父类的功能
    private String school;                              // 新定义的属性school
    public Student(){
        super() ;                                       // 加与不加此语句效果相同
        System.out.println("子类Student中的构造。") ;// 子类的构造方法
    }
    public String getSchool() {                         // 取得school属性
        return school;
    }
    public void setSchool(String school) {              // 设置school属性
        this.school = school;
    }
}
```

以上程序的运行结果与之前一样。super 表示超级的意思，在一些书中也喜欢把父类称为超类，而上面的语法就是表示子类可以直接使用 super()调用父类中的无参构造。

6.2.2 方法的覆写

在继承的关系中也存在着方法覆写的概念，所谓的方法覆写就是指子类定义了与父类中同名的方法，但是在方法覆写时必须考虑到权限，即被子类覆写的方法不能拥有比父类方法更加严格的访问权限。

提示：已学习过的 3 种访问权限。

关于访问权限，实际上前几章已经介绍了 3 种访问权限，即 private、default、public，这 3 种访问权限的具体作用本书后面会有介绍，读者现在只需要记住大小关系即可，即 private < default < public。

所以，如果在父类中使用 public 定义的方法，则子类的访问权限必须是 public，否则程序会无法编译。

范例：方法的覆写

```
class Person{
    void print(){                                   // 定义一个默认访问权限的方法
        System.out.println("Person --> void print(){}") ;
    }
}
class Student extends Person{                       // 定义一个子类继承Person类
    public void print(){                            // 覆写父类中的方法，扩大了权限
        System.out.println("Student --> void print(){}") ;
    }
}
```

```
public class OverrideDemo01 {
    public static void main(String args[]){
        new Student().print() ;                    // 此处执行的是被覆写过的方法
    }
}
```

程序运行结果：

```
Student --> void print(){}
```

从以上程序可以发现，Student 子类定义了与 Person 父类中同名的方法，但是在子类中此方法的访问权限被扩大了，符合覆写的概念，当方法被覆写之后，子类对象调用的将是被覆写后的方法。

另外，要提醒读者的是，如果现在被子类覆写的方法权限缩小，则在编译时将出现错误提示。

范例：错误的方法覆写

```
class Person{
    public void print(){                    // 定义一个默认访问权限的方法
        System.out.println("Person --> void print(){}") ;
    }
}
class Student extends Person{               // 定义一个子类继承Person类
    void print(){                           // 覆写父类中的方法，但缩小了权限，错误
        System.out.println("Student --> void print(){}") ;
    }
}
```

程序编译错误：

```
OverrideDemo02.java:7: print() in Student cannot override print() in Person;
attempting to assign weaker access privileges; was public
        void print(){           // 覆写父类中的方法，但缩小了权限，错误
             ^
1 error
```

以上的错误提示指的就是正在指定更小的访问权限，所以编译无法通过。

如果现在要在子类的方法中访问父类的方法，则使用 super 关键字即可，代码如下所示。

范例：调用父类中被子类覆写过的方法

```
class Person{
    void print(){                                   // 定义一个默认访问权限的方法
        System.out.println("Person --> void print(){}") ;
    }
}
class Student extends Person{                       // 定义一个子类继承Person类
    public void print(){                            // 覆写父类中的方法，扩大了权限
```

```
        super.print() ;                          // 调用父类中的print()方法
        System.out.println("Student --> void print(){}") ;
    }
}
public class OverrideDemo03 {
    public static void main(String args[]){
        new Student().print() ;                  // 此处执行的是被覆写过的方法
    }
}
```

程序运行结果：

```
Person --> void print(){}
Student --> void print(){}
```

从程序中可以清楚地发现，在子类中直接通过“super.方法()”形式即可访问父类中的相关方法。

提问：方法覆写时从 private 变为 default 算是方法覆写吗？

如果现在将父类的一个方法定义成 private 访问权限，在子类中将此方法声明为 default 访问权限，那么这样还叫做覆写吗？

回答：不算是方法覆写。

在解释这个问题之前，先看一下下面的程序代码。

范例：方法覆写

```
class Person{
    private void print(){                    // 方法用private访问权限
        System.out.println("Person --> void print(){}") ;
    }
    public void fun(){                       // 定义一个fun方法
        this.print();                        // 调用print()
    }
}
class Student extends Person{                // 定义一个子类继承Person类
    void print(){                            // 覆写父类中的方法，扩大权限
        System.out.println("Student --> void print(){}") ;
    }
}
public class OverrideDemo04 {
    public static void main(String args[]){
        new Student().fun() ;
    }
}
```

程序运行结果：

```
Person --> void print(){}
```

从程序运行结果可以清楚地发现，现在调用的方法是父类中的方法，也就是说此时子类并没有覆写父类中的方法，而是在子类中重新定义了一个新的方法，所以此时方法没有被覆写。

实际上与方法覆写概念相同的还有另外一个称为属性的覆盖，这一点在开发中使用较少，下面为读者简单介绍属性的覆盖操作。

范例：属性的覆盖

```
class Person{
    public String info = "MLDN" ;                          // 定义一个公共属性
}
class Student extends Person{                              // 定义一个子类继承
                                                              Person类
    public String info = "LXH" ;                           // 此属性与父类中的
                                                              属性名称一致
    public void print(){
        System.out.println("父类中的属性：" + super.info);// 访问父类中的info
                                                              属性
        System.out.println("子类中的属性：" + this.info); // 访问本类中的info
                                                              属性
    }
}
public class OverrideDemo05 {
    public static void main(String args[]){
        new Student().print() ;                            // 调用print()方法
    }
}
```

程序运行结果：

```
父类中的属性：MLDN
子类中的属性：LXH
```

以上程序只作为参考使用，读者只需要了解：如果子类和父类声明了相同名称的属性，则在子类中直接访问时一定是采用“就近访问原则”，即先找到本类中的属性，如果此时要调用父类中的属性，直接使用“super.属性”格式即可。

实际上读者已经发现，方法的重载与覆写是非常类似的，表 6-1 列出了两者的区别。

表 6-1　方法的重载与覆写的区别

序　号	区 别 点	重　载	覆　写
1	单词	Overloading	Overriding
2	定义	方法名称相同，参数的类型或个数不同	方法名称、参数的类型、返回值类型全部相同
3		对权限没有要求	被覆写的方法不能拥有更严格的权限
4	范围	发生在一个类中	发生在继承类中

6.2.3 super 关键字的作用

在上面的程序中一直都出现了 super 关键字，使用 super 关键字可以从子类中调用父类中的构造方法、普通方法和属性。之前已经为读者演示过了调用普通方法和属性的基本操作，下面将使用 super 调用父类中指定构造方法的操作，与 this 调用构造方法的要求一样，语句必须放在子类构造方法的首行。

范例：使用 super 调用父类中的指定构造方法

```
class Person {                                          // 定义父类Person
    private String name;                                // 定义name属性
    private int age;                                    // 定义age属性
    public Person(String name,int age){                 // 通过构造方法设置name和age
        this.setName(name) ;                            // 设置name属性内容
        this.setAge(age) ;                              // 设置age属性内容
    }
    public String getName() {                           // 取得name属性
        return name;
    }
    public void setName(String name) {                  // 设置name属性
        this.name = name;
    }
    public int getAge() {                               // 取得age属性
        return age;
    }
    public void setAge(int age) {                       // 设置age属性
        this.age = age;
    }
    public String getInfo(){                            // 信息输出
        return "姓名：" + this.getName() + "；年龄：" + this.getAge() ;
    }
}
class Student extends Person {                          // Student是Person的子类，扩展父类的功能
    private String school;                              // 新定义的属性school
    public Student(String name,int age,String school){
        super(name,age) ;                               // 指定调用父类中的构造方法
        this.setSchool(school) ;
    }
    public String getSchool() {                         // 取得school属性
        return school;
    }
    public void setSchool(String school) {              // 设置school属性
```

```
        this.school = school;
    }
    public String getInfo(){                          // 覆写父类中的方法
        return super.getInfo() +
                "；学校：" + this.getSchool() ；  // 扩充父类中的方法
    }
}
public class SuperDemo01 {
    public static void main(String args[]) {
        Student stu = new Student("张三",30,"清华大学");
        System.out.println(stu.getInfo());        // 打印信息，调用覆写过的方法
    }
}
```

程序运行结果：

```
姓名：张三；年龄：30；学校：清华大学
```

在以上程序的子类中使用 super()的形式调用了父类中有两个参数的构造方法，然后在子类中又覆写了父类中的 getInfo()方法，所以输出的内容是被子类覆写过的内容。

从以上代码中读者应该可以发现，super 与 this 的作用非常相似，均可以调用构造、普通方法和属性，那么两者之间到底有哪些区别呢？下面通过表 6-2 进行说明。

表 6-2　this 与 super 的区别

序　号	区 别 点	this	super
1	属性访问	访问本类中的属性，如果本类中没有此属性，则从父类中继续查找	访问父类中的属性
2	方法	访问本类中的方法，如果本类中没有此方法，则从父类中继续查找	直接访问父类中的方法
3	调用构造	调用本类构造，必须放在构造方法的首行	调用父类构造，必须放在子类构造方法的首行
4	特殊	表示当前对象	无此概念

表 6-2 中详细地列出了两者的区别，但是在这里还有一个问题需要注意，既然 this 和 super 都可以调用构造方法，那么两者是不可以同时出现的，因为两者调用构造时都必须放在构造方法的首行，另外，还需注意的是，无论子类如何操作，最终必须要先调用父类中的构造方法。

6.3　范例——继承的应用

定义一个整型数组类，要求包含构造方法，增加数据及输出数据成员方法，并利用数组实现动态内存分配。在此基础上定义出以下子类。

（1）排序类：实现排序。

（2）反转类：实现数据反向存放。

本程序要求数组实现动态的内存分配，也就是说其中数组的大小是由程序外部决定的，则在本类的构造方法中应该为类中的数组进行初始化操作，之后每次增加数据时都应该判断数组的内容是否是满的，如果不是满的，则可以向其中添加；如果是满的，则不能添加，另外，要添加数据时，肯定需要有一个指向可以插入的下标，用于记录插入的位置，如图 6-8 所示。

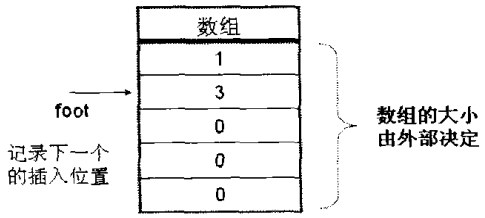


图 6-8　数组的动态分配

范例：实现反转及排序类

```
class Array {
    private int temp[];                          // 定义一个整型数组，此数组大小由外部决定
    private int foot;                            // 定义数组添加的下标
    public Array(int len) {                      // 数组的大小由外部决定
        if (len > 0) {                           // 判断传入的长度是否大于0
            this.temp = new int[len];            // 根据传入的大小开辟空间
        } else {
            this.temp = new int[1];              // 最小维持一个空间
        }
    }
    public boolean add(int i) {
        if (this.foot < this.temp.length) {              // 判断数组是否已经满了
            this.temp[foot] = i;                         // 没有存满则继续添加
            foot++;                                      // 修改下标
            return true;                                 // 添加成功
        } else {                                         // 数组已经存满，不能添加
            return false;                                // 添加失败
        }
    }
    public int[] getArray() {                            // 得到全部的数组
        return this.temp;
    }
};
```

范例：测试反转类

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

```
        ReverseArray a = null ;                          // 声明反转类对象
        a = new ReverseArray(5);                         // 实例化反转类对象
        System.out.print(a.add(23) + "\t");              // 添加内容
        System.out.print(a.add(21) + "\t");              // 添加内容
        System.out.print(a.add(2) + "\t");               // 添加内容
        System.out.print(a.add(42) + "\t");              // 添加内容
        System.out.print(a.add(5) + "\t");               // 添加内容
        System.out.print(a.add(6) + "\n");               // 添加内容
        print(a.getArray());                             // 输出内容
    }
    public static void print(int i[]) {                  // 循环输出数组中的内容
        for (int x = 0; x < i.length; x++) {
            System.out.print(i[x] + "、");
        }
    }
}
```

程序运行结果：

```
true    true    true    true    true    false
5、42、2、21、23、
```

范例：测试排序类

```
public class ArrayDemo {
    public static void main(String args[]) {
        SortArray a = null ;                             // 声明排序类对象
        a = new SortArray(5);                            // 实例化排序类对象
        System.out.print(a.add(23) + "\t");              // 添加内容
        System.out.print(a.add(21) + "\t");              // 添加内容
        System.out.print(a.add(2) + "\t");               // 添加内容
        System.out.print(a.add(42) + "\t");              // 添加内容
        System.out.print(a.add(5) + "\t");               // 添加内容
        System.out.print(a.add(6) + "\n");               // 添加内容
        print(a.getArray());                             // 输出内容
    }
    public static void print(int i[]) {                  // 循环输出数组中的内容
        for (int x = 0; x < i.length; x++) {
            System.out.print(i[x] + "、");
        }
    }
}
```

程序运行结果：

```
true    true    true    true    true    false
2、5、21、23、42、
```

从以上程序可以发现，虽然数组类产生了两个子类，但是各个子类最终取得全部内容的方法还是一样的，也就是说子类根据自己的需要覆写了父类中的方法。读者此时也就应该发现，实际上在继承的关系中父类的设计是最重要的。

6.4 final 关键字

final 在 Java 中表示的意思是最终，也可以称为完结器。可以使用 final 关键字声明类、属性、方法，在声明时需要注意以下几点：

- 使用 final 声明的类不能有子类。
- 使用 final 声明的方法不能被子类所覆写。
- 使用 final 声明的变量即成为常量，常量不可以修改。

范例：使用 final 修饰的类不能有子类

```
final class A {                              // 使用final定义类，不能被继承
}
class B extends A {                          // 错误，不能继承使用final声明的类
}
```

程序编译时出现以下错误：

```
FinalDemo01.java:3: cannot inherit from final A
class B extends A {
              ^
1 error
```

范例：使用 final 修饰的方法不能被子类覆写

```
class A {
    public final void print(){               // 使用final声明的方法不能被覆写
        System.out.println("Hello") ;
    }
}
class B extends A {
    public final void print(){               // 错误，不能覆写用final声明的方法
        System.out.println("MLDN") ;
    }
}
```

程序编译时出现以下错误：

```
FinalDemo02.java:7: print() in B cannot override print() in A; overridden
method is final
        public final void print(){
                          ^
1 error
```

范例：被 final 修饰的变量即成为常量，常量是不能被修改的

```
class A {
    private final String INFO = "LXH" ;      // 使用final声明的变量就是常量
    public final void print(){
        INFO = "HELLO" ;                     // 错误，常量不可修改
    }
}
```

程序编译时出现以下错误：

```
FinalDemo03.java:4: cannot assign a value to final variable INFO
            INFO = "HELLO" ;                        // 常量不可修改
            ^
1 error
```

使用 final 定义的常量本身不能修改，所以当修改时程序就会出现错误。

注意：final 变量的命名规则。

在使用 final 声明变量时，要求全部的字母大写，如 INFO，这点在开发中是非常重要的。

如果一个程序中的变量使用 public static final 声明，则此变量将称为全局常量，如下面的代码：

```
public static final String INFO = "LXH" ;
```

6.5　抽象类的基本概念

前面对类的继承进行了初步的讲解，通过继承可以从原有的类派生出新的类。原有的类称为基类或父类，而新的类则称为派生类或子类。通过继承机制，派生出的新类不仅可以保留原有类的功能，而且还可以拥有更多的功能。

除了上述的机制之外，在 Java 中也可以创建一种类专门用来当作父类，这种类称为“抽象类”。抽象类的作用类似“模板”，其目的是要设计者依据它的格式来修改并创建新的类。但是并不能直接由抽象类创建对象，只能通过抽象类派生出新的类，再由它来创建对象。但是在抽象类的使用中同样存在单继承的局限，即一个子类只能继承一个抽象类。

抽象类的定义及使用规则如下：

- 包含一个抽象方法的类必须是抽象类。
- 抽象类和抽象方法都要使用 abstract 关键字声明。
- 抽象方法只需声明而不需要实现。
- 抽象类必须被子继承，子类（如果不是抽象类）必须覆写抽象类中的全部抽象方法。

【格式 6-2　抽象类的定义格式】

```
abstract class 抽象类名称{
    属性 ;
```

```
    访问权限 返回值类型 方法名称(参数){                    // 普通方法
        [return 返回值] ;
    }
    访问权限 abstract 返回值类型 方法名称(参数) ;          // 抽象方法
    // 在抽象方法中是没有方法体的
}
```

从以上格式中可以发现，抽象类的定义比普通类多了一些抽象方法，其他地方与普通类的组成基本上都是一样的。

范例：定义一个抽象类

```
abstract class A{
    public static final String FLAG = "CHINA" ;
    private String name = "李兴华" ;
    public String getName() {                          // 设置姓名
        return name;
    }
    public void setName(String name) {                 // 取得姓名
        this.name = name;
    }
    public abstract void print() ;                     // 定义抽象方法
}
```

在以上的 A 类中，因为定义了 print()的抽象方法，所以此类的声明为 abstract class。

定义完成一个抽象类后，就可以编写一个子类继承此抽象类。此时需注意的是，子类必须覆写抽象类中的全部抽象方法。具体代码如下。

范例：继承抽象类

```
abstract class A{
    public static final String FLAG = "CHINA" ;
    private String name = "李兴华" ;
    public String getName() {                          // 设置姓名
        return name;
    }
    public void setName(String name) {                 // 取得姓名
        this.name = name;
    }
    public abstract void print() ;                     // 定义抽象方法
}
class B extends A{                                     // 继承抽象类，覆写全部抽象方法
    public void print(){
        System.out.println("FLAG = "+FLAG) ;
        System.out.println("姓名 = " + super.getName()) ;
    }
```

```
}
public class AbstractDemo02{
    public static void main(String args[]){
        B b = new B() ;                                    // 实例化子类对象
        b.print() ;                                        // 调用被子类覆写过的方法
    }
}
```

程序运行结果：

```
FLAG = CHINA
姓名 = 李兴华
```

以上程序完成了抽象类的基本操作，子类覆写了抽象方法，然后在主方法中通过子类的实例化对象就可以调用被子类覆写过的方法。以上程序的类图如图 6-9 所示。

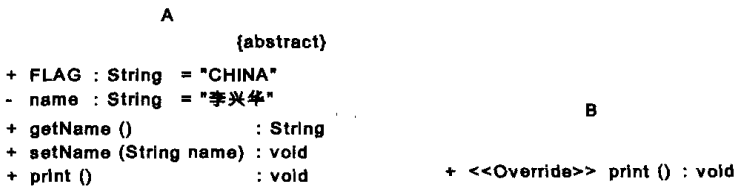


图 6-9　抽象类的图形表示

在图 6-9 中，抽象类使用了{abstract}的方式表示，而在有些时候，抽象类的图形也可以通过类名加斜体的方式表示，如图 6-10 所示。

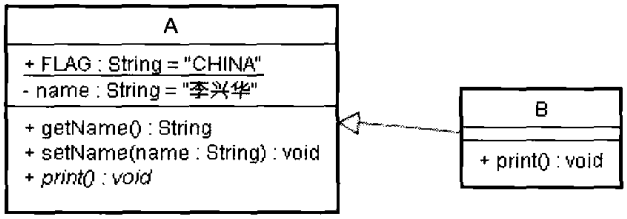


图 6-10　抽象类的另一种表现形式

了解抽象类的概念后，下面再来思考以下的两个问题。

- 一个抽象类可以使用 final 关键字声明吗？
- 一个抽象类中可以定义构造方法吗？

对于第 1 个问题，实际上很容易回答，从之前讲解的概念可以知道，一个类如果使用了 final 关键字声明，则此类不能被子类继承，而抽象类又必须被子类覆写，所以很明显，第 1 个问题的答案是一个抽象类不能使用 final 关键字声明。

注意：抽象方法不要使用 private 声明。

在使用 abstract 关键字修饰抽象方法时不能使用 private 修饰，因为抽象方法必须被子类覆写，而如果使用了 private 声明，则子类是无法覆写的。

第 2 个问题可能会难以回答，实际上在一个抽象类中是允许存在构造方法的，因为抽

象类依然使用的是类的继承关系，而且抽象类中也存在各个属性，所以子类在实例化之前必须先要对父类进行实例化。

范例：在抽象类中定义构造方法

```
abstract class A{
    public A(){                                    // 在抽象类中定义构造方法
        System.out.println("A、抽象类中的构造方法。") ;
    }
}
class B extends A{                                 // 定义子类
    public B(){
        System.out.println("B、子类中的构造方法。") ;
    }
}
public class AbstractDemo03{
    public static void main(String args[]){
        B b = new B() ;                            // 实例化子类对象
    }
}
```

程序运行结果：

```
A、抽象类中的构造方法。
B、子类中的构造方法。
```

从以上程序中可以发现，抽象类中定义了抽象方法，但是定义的抽象方法并不能被外部直接调用，在子类对象实例化前也同样会默认调用父类中的无参构造，也就是说此时的子类实际上也隐含了一个 super 关键字调用构造方法的语句：

```
class B extends A{                                 // 定义子类
    public B(){
        super() ;                                  // 隐含了此语句
        System.out.println("B、子类中的构造方法。") ;
    }
}
```

既然子类可以通过 super 调用抽象类中的构造方法，那么也可以直接在子类指定调用父类中的指定参数的构造方法。

范例：调用抽象类中指定参数的构造方法

```
abstract class Person{                             // 定义抽象类Person
    private String name ;                          // 定义name属性
    private int age ;                              // 定义age属性
    public Person(String name,int age){            // 在抽象类中定义构造方法
        this.setName(name) ;                       // 为name赋值
        this.setAge(age) ;                         // 为age赋值
```

```
    }
    public String getName() {                    // 取得name属性
        return name;
    }
    public void setName(String name) {           // 设置name属性
        this.name = name;
    }
    public int getAge() {                        // 取得age属性
        return age;
    }
    public void setAge(int age) {                // 设置age属性
        this.age = age;
    }
    public abstract String getInfo() ;           // 取得信息，抽象方法
}
class Student extends Person{                    // 定义子类
    private String school ;                      // 定义school属性
    public Student(String name,int age,String school){
        super(name,age) ;                        // 调用父类中有两个参数的构造方法
        this.setSchool(school) ;                 // 为school赋值
    }
    public String getSchool() {                  // 取得school属性
        return school;
    }
    public void setSchool(String school) {       // 设置school属性
        this.school = school;
    }
    public String getInfo(){                     // 覆写抽象类中的抽象方法
        return  "姓名：" + super.getName() +
                "；年龄：" + super.getAge() +
                "；学校：" + this.getSchool() ;
    }
}
public class AbstractDemo04{
    public static void main(String args[]){
        Student stu = new Student("张三",30,"清华大学") ;
        System.out.println(stu.getInfo()) ;
    }
}
```

程序运行结果：

姓名：张三；年龄：30；学校：清华大学

以上程序在实例化 Student 类时先调用了抽象类中的构造方法，将姓名和年龄进行设置，然后再通过覆写的 getInfo()方法输出信息。

提示：抽象类与普通类。

通过代码读者可以清楚地发现，实际上抽象类就是比普通类多定义了一个抽象方法，除了不能直接进行对象的实例化操作之外并没有任何的不同。

6.6 接口的基本概念

接口是 Java 中最重要的概念之一，它可以被理解为一种特殊的类，是由全局常量和公共的抽象方法所组成。

【格式 6-3 接口的定义格式】

```
interface 接口名称{
    全局常量 ;
    抽象方法 ;
}
```

需要注意的是，在接口中的抽象方法必须定义为 public 访问权限，这是绝对不可改变的。

提示：在接口中如果不写 public，则也是 public 访问权限。

在很多的 Java 程序中，经常看到编写接口方法时省略了 public，那么就会有很多的读者认为它的访问权限是 default，实际上这是错误的，不管写与不写，接口中的方法永远是 public。

范例：接口的定义

```
interface A{
    public static final String AUTHOR = "李兴华" ;  // 定义全局常量
    public abstract void print() ;                  // 定义抽象方法
    public abstract String getInfo() ;              // 定义抽象方法
}
```

以上程序定义了一个接口，这里要提醒读者的是，在接口的基本概念中已经明确地声明了接口是由全局常量和抽象方法组成的，所以此处的接口定义可以简化成如下的形式：

```
interface A{
    String AUTHOR = "李兴华" ;   // 等价于：public static final String AUTHOR
                                    = "李兴华" ;
    void print() ;               // 等价于：public abstract void print() ;
    String getInfo() ;           // 等价于：public abstract String getInfo() ;
}
```

在程序中接口可以通过图 6-11 所示的图形表示。

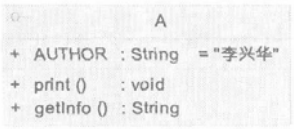


图 6-11　接口图形表示

与抽象类一样，接口若要使用也必须通过子类，子类通过 implements 关键字实现接口。

【格式 6-4　实现接口】

```
class 子类 implements 接口A,接口B,...{
}
```

从格式 6-4 中可以清楚地发现，一个子类可以同时实现多个接口，那么这实际上就摆脱了 Java 的单继承局限。

范例：实现接口

```
interface A{                                    // 定义接口A
    public String AUTHOR = "李兴华" ;           // 定义全局常量
    public void print() ;                       // 定义抽象方法
    public String getInfo() ;                   // 定义抽象方法
}
interface B{                                    // 定义接口B
    public void say() ;                         // 定义抽象方法
}
class X implements A,B{                         // 子类同时实现两个接口
    public void say() {                         // 覆写B接口中的抽象方法
        System.out.println("Hello World!!!");
    }
    public String getInfo() {                   // 覆写A接口中的抽象方法
        return "HELLO";
    }
    public void print() {                       // 覆写A接口中的抽象方法
        System.out.println("作者：" + AUTHOR);
    }
}
public class InterfaceDemo03{
    public static void main(String args[]){
        X x = new X() ;                         // 实例化子类对象
        x.say() ;                               // 调用被覆写过的方法
        x.print() ;                             // 调用被覆写过的方法
    }
}
```

程序运行结果：

```
Hello World!!!
作者：李兴华
```

以上的程序中，一个子类同时实现了两个接口，这样在子类中就必须同时覆写两个接口中的全部抽象方法。以上程序的类图如图 6-12 所示。

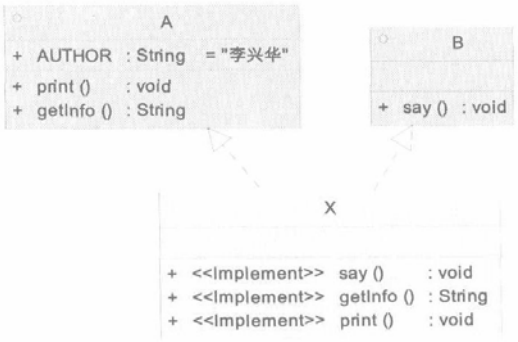


图 6-12　接口实现的类图

如果在开发中一个子类既要实现接口又要继承抽象类，则可以按照格式 6-5 进行定义。

【格式 6-5　继承抽象类实现接口】

```
class 子类 extends 抽象类 implements 接口A,接口B,...{
}
```

范例：子类同时继承抽象类和实现接口

```
interface A{                                    // 定义接口A
    public String AUTHOR = "李兴华" ;           // 定义全局常量
    public void print() ;                       // 定义抽象方法
    public String getInfo() ;                   // 定义抽象方法
}
abstract class B{                               // 定义抽象类 B
    public abstract void say() ;
}
class X extends B implements A{                 // 子类同时实现接口
    public void say() {                         // 覆写抽象类B中的抽象方法
        System.out.println("Hello World!!!");
    }
    public String getInfo() {                   // 覆写接口A中的抽象方法
        return "HELLO";
    }
    public void print() {                       // 覆写接口A中的抽象方法
        System.out.println("作者：" + AUTHOR);
    }
}
public class InterfaceDemo04 {
    public static void main(String args[]){
        X x = new X() ;                         // 实例化子类对象
        x.say() ;                               // 调用被覆写过的方法
        x.print() ;                             // 调用被覆写过的方法
```

```
    }
}
```

程序运行结果：

```
Hello World!!!
作者：李兴华
```

在以上程序中因为抽象类和接口本身都有抽象方法，所以实现类中必须覆写 3 个抽象方法。另外，在 Java 中是允许一个抽象类实现多个接口的。

范例：抽象类实现接口

```
interface A{                                    // 定义接口A
    public String AUTHOR = "李兴华" ;           // 定义全局常量
    public void print() ;                       // 定义抽象方法
    public String getInfo() ;                   // 定义抽象方法
}
abstract class B implements A{                  // 定义抽象类，实现接口
    public abstract void say() ;                // 此时抽象类中存在3个抽象方法
}
class X extends B{                              // 子类继承抽象类
    public void say() {                         // 覆写抽象类B中的抽象方法
        System.out.println("Hello World!!!");
    }
    public String getInfo() {                   // 覆写抽象类B中的抽象方法
        return "HELLO";
    }
    public void print() {                       // 覆写抽象类B中的抽象方法
        System.out.println("作者：" + AUTHOR);
    }
}
public class InterfaceDemo05 {
    public static void main(String args[]){
        X x = new X() ;                         // 实例化子类对象
        x.say() ;                               // 调用被覆写过的方法
        x.print() ;                             // 调用被覆写过的方法
    }
}
```

程序运行结果：

```
Hello World!!!
作者：李兴华
```

在以上程序中，因为抽象类实现了接口，所以在抽象类中就包含了 3 个抽象方法，这样抽象类的子类就必须同时覆写 3 个抽象方法，在 Java 中一个接口是不允许继承抽象类的，但是允许一个接口继承多个接口，如格式 6-6 所示。

【格式 6-6　接口的继承】

```
interface 子接口 extends 父接口A,父接口B,...{
}
```

范例：接口的多继承

```
interface A{                                    // 定义接口A
    public String AUTHOR = "李兴华" ;           // 定义全局常量
    public void printA() ;                      // 定义抽象方法
}
interface B{                                    // 定义接口B
    public void printB() ;                      // 定义抽象方法
}
interface C extends A,B{                        // 定义接口C，同时继承接口A、B
    public void printC() ;                      // 定义抽象方法
}
class X implements C{                           // 子类实现接口C
    public void printA() {                      // 覆写接口A中的printA()方法
        System.out.println("A、Hello World") ;
    }
    public void printB() {                      // 覆写接口B中的printB()方法
        System.out.println("B、Hello MLDN") ;
    }
    public void printC() {                      // 覆写接口C中的printC()方法
        System.out.println("C、Hello LXH") ;
    }
}
public class InterfaceDemo06 {
    public static void main(String args[]){
        X x = new X() ;                         // 实例化子类对象
        x.printA() ;                            // 调用方法
        x.printB() ;                            // 调用方法
        x.printC() ;                            // 调用方法
    }
}
```

程序运行结果：

```
A、Hello World
B、Hello MLDN
C、Hello LXH
```

从上述程序中可以发现，接口 C 继承了接口 A 和接口 B，这样在接口 C 中就同时存在 3 个抽象方法，子类 X 实现接口 C 后就必须同时覆写 3 个抽象方法。

6.7　对象的多态性

多态性在面向对象中是一个最重要的概念，在 Java 中面向对象主要有以下两种主要体现：

（1）方法的重载与覆写。

（2）对象的多态性。

对于方法的重载与覆写，本书之前已经作过详细介绍，下面将介绍对象的多态性。

对象的多态性主要分为以下两种类型。

（1）向上转型：子类对象→父类对象。

（2）向下转型：父类对象→子类对象。

对于向上转型，程序会自动完成，而对于向下转型时，必须明确地指明要转型的子类类型，如格式 6-7 所示。

【格式 6-7　对象转型】

对象向上转型：父类 父类对象 = 子类实例 ；

对象向下转型：子类 子类对象 = （子类）父类实例 ；

下面介绍如何进行对象的向上转型操作，具体代码如下。

范例：对象的向上转型

```
class A{                                    // 定义类A
    public void fun1(){                     // 定义fun1()方法
        System.out.println("A --> public void fun1(){}") ;
    }
    public void fun2(){                     // 定义fun2()方法
        this.fun1() ;
    }
};
class B extends A{                          // 子类通过extends继承父类
    public void fun1(){                     // 覆写父类中的fun1()方法
        System.out.println("B --> public void fun1(){}") ;
    }
    public void fun3(){                     // 子类自己定义的方法
        System.out.println("B --> public void fun3(){}") ;
    }
};
public class PolDemo01 {
    public static void main(String[] args) {
        B b = new B() ;                     // 定义子类实例化对象
        A a = b ;                           // 发生了向上转型的关系，子类 --> 父类
        a.fun1();                           // 此方法被子类覆写过
    }
}
```

程序运行结果：

```
B --> public void fun1(){}
```

以上程序就是一个对象向上转型的关系，从程序的运行结果中可以发现，此时虽然是使用父类对象调用了 fun1 方法，但实际上调用的方法是被子类覆写过的方法，也就是说，如果对象发生了向上转型关系后，所调用的方法一定是被子类覆写过的方法。但是在此时一定要注意，此时的对象 a 是无法调用 B 类中的 fun3()方法的，因为此方法只在子类定义，而没有在父类中定义，如果要想调用子类自己的方法，则肯定要使用子类实例，所以此时可以将对象进行向下转型。

范例：对象的向下转型

```
class A{                                    // 定义类A
    public void fun1(){                     // 定义fun1()方法
        System.out.println("A --> public void fun1(){}") ;
    }
    public void fun2(){                     // 定义fun2()方法
        this.fun1() ;
    }
};
class B extends A{                          // 子类通过extends继承父类
    public void fun1(){                     // 覆写父类中的fun1()方法
        System.out.println("B --> public void fun1(){}") ;
    }
    public void fun3(){                     // 子类自己定义的方法
        System.out.println("B --> public void fun3(){}") ;
    }
};
public class PolDemo02 {
    public static void main(String[] args) {
        A a = new B() ;                     // 发生了向上转型的关系，子类 --> 父类
        B b = (B)a ;                        // 此时发生了向下转型关系
        b.fun1() ;                          // 调用方法被覆写的方法
        b.fun2() ;                          // 调用父类的方法
        b.fun3() ;                          // 调用子类自己定义的方法
    }
}
```

程序运行结果：

```
B --> public void fun1(){}
B --> public void fun1(){}
B --> public void fun3(){}
```

从以上程序中可以发现，如果要想调用子类自己的方法，则一定只能用子类声明对象，另外，在子类中调用了父类中的 fun2()方法，fun2()方法要调用 fun1()方法，但此时 fun1()

方法已经被子类所覆写，所以，此时调用的方法是被子类覆写过的方法。

注意：对象向下转型的要求。

在以上程序中已经介绍了对象的向上和向下转型的基本概念，但是必须提醒读者的是，在进行对象的向下转型前，必须首先发生对象向上转型，否则将出现对象转换异常，如以下代码所示。

范例：错误的转型

```
class A{                                    // 定义类A
    public void fun1(){                     // 定义fun1()方法
        System.out.println("A --> public void fun1(){}") ;
    }
    public void fun2(){                     // 定义fun2()方法
        this.fun1() ;
    }
};
class B extends A{                          // 子类通过extends继承父类
    public void fun1(){                     // 覆写父类中的fun1()方法
        System.out.println("B --> public void fun1(){}") ;
    }
    public void fun3(){                     // 子类自己定义的方法
        System.out.println("B --> public void fun3(){}") ;
    }
};
public class PolDemo03 {
    public static void main(String[] args) {
        A a = new A() ;                     // 此时声明的是父类对象
        B b = (B)a ;                        // 此时发生了向下转型关系
        b.fun1() ;                          // 调用方法被覆写的方法
        b.fun2() ;                          // 调用父类的方法
        b.fun3() ;                          // 调用子类自己定义的方法
    }
}
```

程序运行时错误：

```
Exception in thread "main" java.lang.ClassCastException: A
    at PolDemo03.main(PolDemo03.java:21)
```

由以上程序可以发现，此时的 A 类对象是由 A 类本身进行实例化的，然后将 A 类的实例化对象强制转换为子类对象，这样写在语法上是没有任何错误的，但是在运行时出现了异常，这是为什么呢？为什么父类不可以向子类转换呢？其实这点并不难理解，读者可以想一下在现实生活中的例子，假如您今天刚买完一些生活用品，回家时在路上碰见一个孩子，这个孩子忽然对您说：“我是您的儿子，您把你买的东西给我吧！”，这时您肯定不会把您

的东西给这个孩子，因为您不确定他跟您是否有关系，怎么能给呢？那么在这个程序中也是同样的道理，父类用其本身类实例化自己的对象，但它并不知道谁是自己的子类，那肯定在转换时会出现错误，那么这个错误该如何纠正呢？只需要将两个对象建立好关系即可，在声明父类对象时先发生向上转型关系“A a = new B();”，这时相当于是由子类去实例化父类对象，也就是说这时父类知道有这么一个子类，也就相当于父亲知道了自己有这么一个孩子，所以下面再进行转换时就不会再有问题。

了解了对象多态性后，那么这个概念到底有哪些用处呢？下面要求设计一个方法，要求此方法可以接收 A 类的任意子类对象，并调用方法，此时，如果不使用对象多态性，则肯定会使用以下形式的代码：

范例：不使用对象多态性实现功能

```
class A{
    public void fun1(){                    // 定义fun1()方法
        System.out.println("A --> public void fun1(){}") ;
    }
    public void fun2(){                    // 定义fun2()方法
        this.fun1() ;
    }
};
class B extends A{                         // 子类通过extends继承父类
    public void fun1(){                    // 覆写父类中的fun1()方法
        System.out.println("B --> public void fun1(){}") ;
    }
    public void fun3(){                    // 子类自己定义的方法
        System.out.println("B --> public void fun3(){}") ;
    }
};
class C extends A{                         // 子类通过extends继承父类
    public void fun1(){                    // 覆写父类中的fun1()方法
        System.out.println("C --> public void fun1(){}") ;
    }
    public void fun5(){                    // 子类自己定义的方法
        System.out.println("C --> public void fun3(){}") ;
    }
};
public class PolDemo04 {
    public static void main(String[] args) {
        fun(new B()) ;                     // 传递B类实例
        fun(new C()) ;                     // 传递C类实例
    }
    public static void fun(B b){           // 接收子类B的实例
```

```
        b.fun1() ;                          // 统一调用覆写父类的fun1()方法
    }
    public static void fun(C c){            // 接收子类C的实例
        c.fun1() ;
    }
}
```

程序运行结果：

```
B --> public void fun1(){}
C --> public void fun1(){}
```

以上程序虽然实现了基本的要求，但是读者应该可以发现：如果按照以上的方式完成程序，则当产生了一个 A 类的子类时，fun()方法就要重载一次，这样如果功能扩充，则必须要修改类本身，那么如果使用对象多态性完成呢？具体代码如下所示。

范例：使用对象多态性实现此功能

```
class A{
    public void fun1(){                 // 定义fun1()方法
        System.out.println("A --> public void fun1(){}") ;
    }
    public void fun2(){                 // 定义fun2()方法
        this.fun1() ;
    }
};
class B extends A{                      // 子类通过extends继承父类
    public void fun1(){                 // 覆写父类中的fun1()方法
        System.out.println("B --> public void fun1(){}") ;
    }
    public void fun3(){                 // 子类自己定义的方法
        System.out.println("B --> public void fun3(){}") ;
    }
};
class C extends A{                      // 子类通过extends继承父类
    public void fun1(){                 // 覆写父类中的fun1()方法
        System.out.println("C --> public void fun1(){}") ;
    }
    public void fun5(){                 // 子类自己定义的方法
        System.out.println("C --> public void fun3(){}") ;
    }
};
public class PolDemo05 {
    public static void main(String[] args) {
        fun(new B()) ;                  // 传递B类实例，产生向上转型
        fun(new C()) ;                  // 传递C类实例，产生向上转型
```

```
    }
    public static void fun(A a){    // 接收父类对象
        a.fun1() ;
    }
}
```

程序运行结果：

```
B --> public void fun1(){}
C --> public void fun1(){}
```

从以上程序可以发现，此时由于在 fun()方法中使用了对象的多态性，所以可以接收任何的子类对象，这样无论子类如何增加，fun()方法都不用做任何的改变，因为一旦发生对象的向上转型关系后，调用的方法一定是被子类覆写过的方法。

6.8 instanceof 关键字

在Java中可以使用instanceof关键字判断一个对象到底是哪个类的实例，格式如下所示。

对象 instanceof 类 → 返回boolean类型

范例：验证 instanceof 关键字的作用

```
class A {
    public void fun1() {            // 定义fun1()方法
        System.out.println("A --> public void fun1(){}");
    }
    public void fun2() {            // 定义fun2()方法
        this.fun1();
    }
};
class B extends A {                 // 子类通过extends继承父类
    public void fun1() {            // 覆写父类中的fun1()方法
        System.out.println("B --> public void fun1(){}");
    }
    public void fun3() {            // 子类自己定义的方法
        System.out.println("B --> public void fun3(){}");
    }
};
public class InstanceofDemo01 {
    public static void main(String[] args) {
        A a1 = new B();             // 通过向上转型实例化A类对象
        System.out.println("A a1 = new B(): " + (a1 instanceof A));
        System.out.println("A a1 = new B(): " + (a1 instanceof B));
        A a2 = new A();             // 通过A类的构造实例化本类对象
```

```
        System.out.println("A a2 = new A(): " + (a2 instanceof A));
        System.out.println("A a2 = new A(): " + (a2 instanceof B));
    }
}
```

程序运行结果：

```
A a1 = new B(): true
A a1 = new B(): true
A a2 = new A(): true
A a2 = new A(): false
```

从以上程序中可以发现，由于通过子类实例化的对象同时是子类和父类的实例，所以可以直接进行向上或向下转型，但如果直接使用了父类实例化本类对象，则一定就不再是子类实例了，所以是不能转换的。

在进行对象的向下转型关系前最好先进行判断后再进行相应的转换操作，这样可以避免类型转换异常的出现。

范例：在向下转型前进行验证

```
class A{
    public void fun1(){                    // 定义fun1()方法
        System.out.println("A --> public void fun1(){}") ;
    }
    public void fun2(){                    // 定义fun2()方法
        this.fun1() ;
    }
};
class B extends A{                         // 子类通过extends继承父类
    public void fun1(){                    // 覆写父类中的fun1()方法
        System.out.println("B --> public void fun1(){}") ;
    }
    public void fun3(){                    // 子类自己定义的方法
        System.out.println("B --> public void fun3(){}") ;
    }
};
class C extends A{                         // 子类通过extends继承父类
    public void fun1(){                    // 覆写父类中的fun1()方法
        System.out.println("C --> public void fun1(){}") ;
    }
    public void fun5(){                    // 子类自己定义的方法
        System.out.println("C --> public void fun3(){}") ;
    }
};
public class InstanceofDemo02 {
    public static void main(String[] args) {
```

```
        fun(new B()) ;                  // 传递B类实例，产生向上转型
        fun(new C()) ;                  // 传递C类实例，产生向上转型
    }
    public static void fun(A a){        // 此方法可以分别调用各自子类单独定义的方法
        a.fun1() ;
        if(a instanceof B){             // 判断是否是B类实例
            B b = (B)a ;                // 进行向下转型
            b.fun3() ;                  // 调用子类自己定义的方法
        }
        if(a instanceof C){             // 判断是否是C类实例
            C c = (C)a ;                // 进行向下转型
            c.fun5() ;                  // 调用子类自己定义的方法
        }
    }
}
```

程序运行结果：

```
B --> public void fun1(){}
B --> public void fun3(){}
C --> public void fun1(){}
C --> public void fun3(){}
```

在上面的 fun()方法中因为要调用各个子类自己的方法，所以必须进行对象的向下转型，但是为了保证程序在运行时不出现类转换异常，所以在发生向下转型前要使用 instanceof 关键字判断是哪个子类的实例，以保证程序的运行正确。

提问：这里的父类设计得是否很合理？

如果按照以上的方式编写代码，那么如果再增加子类，则必须要修改 fun()方法，这样的类设计是不是很合理呢？

回答：在类设计时永远不要去继承一个已经实现好的类。

会产生这个疑问本身是非常好的，那么读者现在思考一下，如果此时在程序中将父类设计得足够合理，那是否还会产生这样的问题呢？所以，在类的设计中重点在于父类的设计，而且在类的设计中，永远不要去继承一个已经实现好的类，只能继承抽象类或实现接口，因为一旦发生对象的向上转型关系后，所调用的方法一定是被子类所覆写过的方法。

6.9 抽象类与接口的应用

6.9.1 为抽象类与接口实例化

在 Java 中可以通过对象的多态性为抽象类和接口实例化，这样再使用抽象类和接口时

即可调用本子类中所覆写过的方法。

范例：为抽象类实例化

```
abstract class A{                          // 定义抽象类A
    public abstract void print() ;         // 定义抽象方法print()
};
class B extends A{                         // 子类继承抽象类
    public void print(){
        System.out.println("Hello World!!!") ;
    }
};
public class AbstractCaseDemo01 {
    public static void main(String[] args) {
        A a = new B() ;                    // 通过子类为抽象类实例化
        a.print() ;                        // 调用的方法是被子类覆写过的方法
    }
}
```

程序运行结果：

```
Hello World!!!
```

用同样的方法可以为接口进行实例化。

范例：为接口实例化

```
interface A{                               // 定义接口A
    public abstract void print() ;         // 定义抽象方法print()
};
class B implements A{                      // 子类实现接口
    public void print(){
        System.out.println("Hello World!!!") ;
    }
};
public class InterfaceCaseDemo01 {
    public static void main(String[] args) {
        A a = new B() ;                    // 通过子类为接口实例化
        a.print() ;                        // 调用的方法是被子类覆写过的方法
    }
}
```

程序运行结果：

```
Hello World!!!
```

从以上的两组代码中可以清楚地发现，程序通过对象的多态性为抽象类及接口进行实例化，这样所调用的方法就是被子类所覆写过的方法。

6.9.2 抽象类的实际应用——模板设计

既然可以为抽象类实例化，那么抽象类到底应如何使用呢？来看下面的这样一种场景：假设人分为学生和工人，学生和工人都可以说话，但是学生和工人说话的内容是不一样的，也就是说，说话这个功能应该是一个具体功能，而说话的内容就要由学生或工人来决定了，所以此时就可以使用抽象类实现这种场景，如图 6-13 所示。

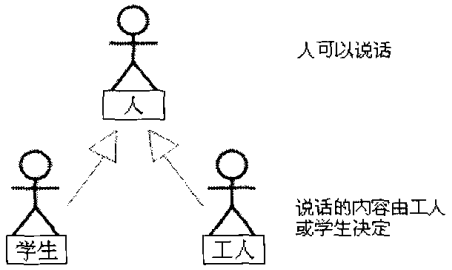


图 6-13 抽象类应用

范例：抽象类的实际应用

```
abstract class Person {                          // 定义抽象类Person
    private String name;                         // 定义name属性
    private int age;                             // 定义age属性
    public Person(String name, int age) {        // 为属性初始化
        this.name = name;                        // 为name属性赋值
        this.age = age;                          // 为age属性赋值
    }
    public String getName() {                    // 取得name属性内容
        return name;
    }
    public int getAge() {                        // 取得age属性内容
        return age;
    }
    public void say() {                          // 说话是具体功能，要定义成普通方法
        System.out.println(this.getContent());
    }
    public abstract String getContent();         // 说话的内容由子类决定
}
class Student extends Person {                   // 定义Student类继承Person类
    private float score;
    public Student(String name, int age, float score) {
        super(name, age);                        // 调用父类的构造方法
        this.score = score;
    }
    public String getContent() {                 // 覆写父类中的抽象方法
        return  "学生信息 --> 姓名：" + super.getName() +
                "；年龄：" + super.getAge()
                + "；成绩：" + this.score;
    }
}
class Worker extends Person {                    // 定义Worker类继承Person类
```

```
    private float salary;
    public Worker(String name, int age, float salary) {
        super(name, age);                          // 调用父类的构造方法
        this.salary = salary;
    }
    public String getContent() {                   // 覆写父类中的抽象方法
        return  "工人信息 --> 姓名：" + super.getName() +
                "；年龄：" + super.getAge()
                + "；工资：" + this.salary;
    }
}
public class AbstractCaseDemo02 {
    public static void main(String[] args) {
        Person per1 = null ;                       // 声明Person对象
        Person per2 = null ;                       // 声明Person对象
        per1 = new Student("张三",20,99.0f) ;      // 学生是一个人
        per2 = new Worker("李四",30,3000.0f) ;     // 工人是一个人
        per1.say() ;                               // 学生说学生的内容
        per2.say() ;                               // 工人说工人的内容
    }
}
```

程序运行结果：

```
学生信息 --> 姓名：张三；年龄：20；成绩：99.0
工人信息 --> 姓名：李四；年龄：30；工资：3000.0
```

从程序的运行结果中可以发现，在 Person 类中就相当于定义了一个模板，在主方法中调用时，调用的就是普通方法，而子类只需要实现父类中的抽象方法，就可以取得一个具体的信息。

提示：现实生活中的模板。

对于以上的操作代码，若读者不是很理解，则可以查看以下的说明，小时候有些读者因为违反校纪可能会填写过如下的违纪卡：

违纪卡			
姓名：		班级：	
日期：		事由：	

以上的一张表如果是空白的，则没有任何意义，但是每一位违纪者都知道自己该在哪个位置上填写对应的内容，填写完整后此卡片就有意义了，下面就是一个违纪者填写的表格：

违纪卡			
姓名：	李兴华	班级：	MLDN JAVA 学习班
日期：	2008 年 11 月 11 日 星期二	事由：	上课睡觉

如果一个违纪者将以上的违纪卡填写完后，相关的管理者就可以从这张卡片中取得自己需要的信息，实际上提供的是一个模板。

6.9.3 接口的实际应用——制定标准

接口是 Java 解决多继承局限的一种手段，而且从前面的内容读者也已经清楚可以通过对象多态性为接口进行实例化，但是接口在实际中更多的作用是用来制订标准。例如，U 盘和打印机都可以插在计算机上使用，这是因为它们都实现了 USB 的接口，对于计算机来说，只要是符合了 USB 接口标准的设备就都可以插进来，如图 6-14 所示。

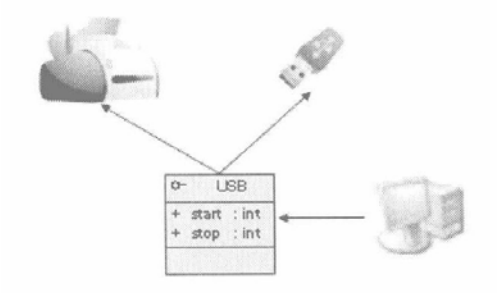


图 6-14 USB 设备

从图 6-12 中可以清楚地看到，若打印机和 U 盘均实现 USB 接口，则都可以插入计算机，以上的要求可以变为如下程序。

范例：制订 USB 标准

```
interface USB{                                          // 定义USB接口
    public void start() ;                               // USB设备开始工作
    public void stop() ;                                // USB设备结束工作
}
class Computer{
    public static void plugin(USB usb){                 // 只要是USB的设备就都可以插入
        usb.start() ;                                   // 让USB设备开始工作
        System.out.println("======= USB 设备工作 =======") ;
        usb.stop() ;                                    // 让USB设备停止工作
    }
};
class Flash implements USB{                             // U盘
    public void start(){                                // 覆写start()方法
        System.out.println("U盘开始工作。") ;
    }
    public void stop(){                                 // 覆写stop()方法
        System.out.println("U盘停止工作。") ;
    }
};
class Print implements USB{                             // 打印机
```

```
    public void start(){                          // 覆写start()方法
        System.out.println("打印机开始工作。") ;
    }
    public void stop(){                           // 覆写stop()方法
        System.out.println("打印机停止工作。") ;
    }
};
public class InterfaceCaseDemo02 {
    public static void main(String[] args) {
        Computer.plugin(new Flash());             // 插入U盘
        Computer.plugin(new Print()) ;            // 插入打印机
    }
}
```

程序运行结果：

```
U盘开始工作。
======= USB 设备工作 =======
U盘停止工作。
打印机开始工作。
======= USB 设备工作 =======
打印机停止工作。
```

从以上程序可以清楚地发现，接口就是规定出了一个标准，计算机认的只是接口，而对于具体的设备计算机本身并不关心。

6.9.4　设计模式——工厂设计

工厂设计是 Java 开发中使用得最多的一种设计模式，那么什么叫工厂设计？工厂设计有哪些作用呢？在说明问题前，请读者先观察以下的程序。

范例：观察程序中的问题

```
interface Fruit{                                  // 定义一个水果的接口
    public void eat() ;                           // 吃水果的方法
}
class Apple implements Fruit{                     // 定义子类Apple
    public void eat(){
        System.out.println("** 吃苹果。") ;
    }
};
class Orange implements Fruit{                    // 定义子类Orange
    public void eat(){                            // 覆写eat()方法
        System.out.println("** 吃橘子。") ;
    }
};
```

```
public class InterfaceCaseDemo03 {
    public static void main(String args[]){
        Fruit f = new Apple() ;                          // 实例化接口
        f.eat() ;                                        // 调用方法
    }
}
```

程序运行结果：

```
** 吃苹果。
```

以上程序相信读者都可以看明白，子类为接口实例化后，调用被子类覆写过的方法，但是以上的操作中是否存在问题呢？前面曾经为读者讲解过这样的一个注意事项：主方法实际上就相当于是一个客户端，如果此时需要更换一个子类，则必须要修改主方法，那么这实际上就存在了问题，但对于这样的问题该如何解决呢？本书之前介绍过的 JVM 工作原理为所有的程序只认 JVM，每个 JVM 会根据所在的操作系统不同自动进行设置，也就是说：程序→JVM→操作系统，实际上本程序也可以按照此种方式解决，在接口与具体子类之间可以加入一个过渡端，通过此过渡端取得接口实例，如图 6-15 所示。

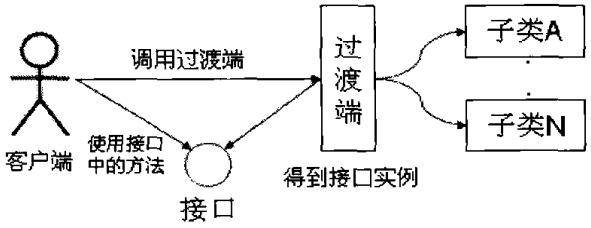


图 6-15　问题解决

从图 6-15 中可以清楚地发现，程序在接口和子类之间加入了一个过渡端，通过此过渡端取得接口的实例化对象，一般都会称这个过渡端为工厂类。

范例：工厂设计模式

```
interface Fruit{                                         // 定义一个水果的接口
    public void eat() ;                                  // 吃水果的方法
}
class Apple implements Fruit{                            // 定义子类Apple
    public void eat(){                                   // 覆写eat()方法
        System.out.println("** 吃苹果。") ;
    }
};
class Orange implements Fruit{                           // 定义子类Orange
    public void eat(){                                   // 覆写eat()方法
        System.out.println("** 吃橘子。") ;
    }
};
class Factory{                                           // 定义工厂类
```

```
    public static Fruit getInstance(String className){
        Fruit f = null ;                        // 定义接口对象
        if("apple".equals(className)){          // 判断是哪个子类的标记
            f = new Apple() ;                   // 通过Apple类实例化接口
        }
        if("orange".equals(className)){         // 判断是哪个子类的标记
            f = new Orange() ;                  // 通过Orange类实例化接口
        }
        return f ;
    }
}
public class InterfaceCaseDemo04 {
    public static void main(String args[]){
        Fruit f = null ;                        // 定义接口对象
        f = Factory.getInstance("apple") ;      // 通过工厂取得实例
        f.eat() ;                               // 调用方法
    }
}
```

程序运行结果：

```
** 吃苹果。
```

此时，代码是固定了一个 apple 的字符串，如果使用初始化参数的方式，则可以任意选择要使用的子类标记，如将主方法修改如下：

```
public class InterfaceCaseDemo05 {
    public static void main(String args[]){
        Fruit f = null ;                        // 定义接口对象
        f = Factory.getInstance(args[0]) ;      // 通过工厂取得实例
        if(f!=null){                            // 判断对象是否为空
            f.eat() ;                           // 如果输错了标记，则肯定返回的是一
                                                   个null
        }
    }
}
```

这样在运行时直接输入以下的命令即可：

```
取得苹果类的实例：java InterfaceCaseDemo05 apple
取得橘子类的实例：java InterfaceCaseDemo05 orange
```

程序的运行结果与之前一样，但是取得实例的过程却不太一样，因为接口对象的实例是通过工厂取得的，这样以后如果再有子类扩充，直接修改工厂类客户端就可以根据标记得到相应的实例，灵活性较高。以上程序的执行流程可以用图 6-16 表示。

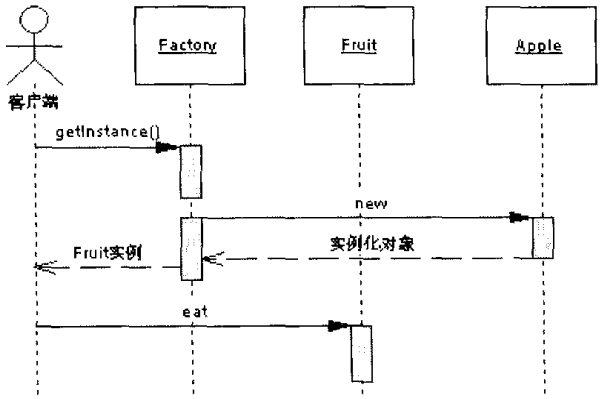


图 6-16　工厂类的执行流程

提问：为什么在进行字符串判断时要把字符串常量写在前面？

在工厂类中有下面的一段代码：

```
if("orange".equals(className)){
    f = new Orange();
}
```

为什么不写成如下的代码形式呢？

```
if(className.equals("orange")){
    f = new Orange() ;
}
```

回答：这样做可以避免空指向异常。

以上的两种形式实际上都没有任何问题，使用任何一种形式都可以完成功能，但是如果使用第 2 种形式，则在运行中有可能出现空指向异常，因为传入的 className 的值有可能为 null。下面比较以下的两段代码。

范例： 第 1 段代码

```
public class EqualDemo01 {
    public static void main(String args[]){
        String str = null ;                          // 字符串
        System.out.println(str.equals("MLDN")) ;     // 错误
    }
}
```

程序运行结果（出错）：

```
Exception in thread "main" java.lang.NullPointerException
```

因为 str 的内容为 null，所以调用任何类中的属性或方法时肯定出现空指向异常。

范例： 第 2 段代码

```
public class EqualDemo02{
    public static void main(String args[]){
        String str = null ;                          // 字符串
```

```
            System.out.println("MLDN".equals(str)) ;// 不会出错
        }
}
```

程序运行结果：

```
False
```

因为字符串本身就是一个 String 的匿名对象，所以永远是一个实例化对象，同样，两个值交换顺序之后就不会出现空指向异常。在以后的开发中建议读者也使用第 2 段代码的形式编写代码。

6.9.5　设计模式——代理设计

代理设计也是在 Java 开发中使用较多的一种设计模式，所谓的代理设计就是指由一个代理主题来操作真实主题，真实主题执行具体的业务操作，而代理主题负责其他相关业务的处理，就好像在生活中经常使用的代理上网一样，客户通过网络代理连接网络，由代理服务器完成用户权限和访问限制等与上网操作相关的操作，如图 6-17 所示。

不管是代理操作还是真实的操作，其共同的目的就是上网，所以用户关心的只是如何上网，至于具体是如何操作的用户并不关心，所以可以得出如图 6-18 所示的分析结果。

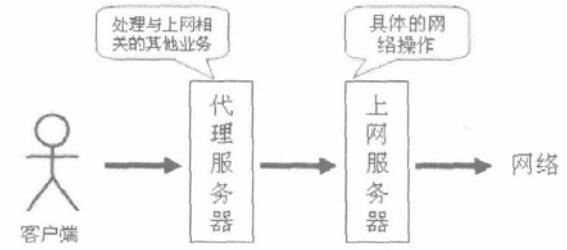


图 6-17　代理上网

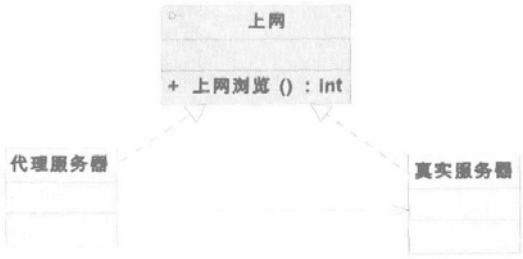


图 6-18　分析结果

从图 6-18 可以发现，只需要定义一个上网的接口，代理主题和真实主题都同时实现此接口，然后再由代理操作真实主题即可，以上的要求可以形成如下代码。

范例：代理操作

```
interface Network{                                  // 定义Network接口
    public void browse() ;                          // 定义浏览的抽象方法
}
class Real implements Network{                      // 真实的上网操作
    public void browse(){                           // 覆写抽象方法
        System.out.println("上网浏览信息") ;
    }
}
class Proxy implements Network{                     // 代理上网
    private Network network ;
    public Proxy(Network network){                  // 设置代理的真实操作
```

```
        this.network = network ;          // 设置代理的子类
    }
    public void check(){                  // 与具体上网相关的操作
        System.out.println("检查用户是否合法");
    }
    public void browse(){
        this.check() ;                    // 可以同时调用多个与具体业务相关的操作
        this.network.browse() ;           // 调用真实上网操作
    }
}
public class ProxyDemo {
    public static void main(String args[]){
        Network net = null ;              // 定义接口对象
        net = new Proxy(new Real()) ;     // 实例化代理，同时传入代理的真实操作
        net.browse() ;                    // 客户只关心上网浏览一个功能
    }
}
```

程序运行结果：

```
检查用户是否合法
上网浏览信息
```

以上程序的执行流程如图 6-19 所示。

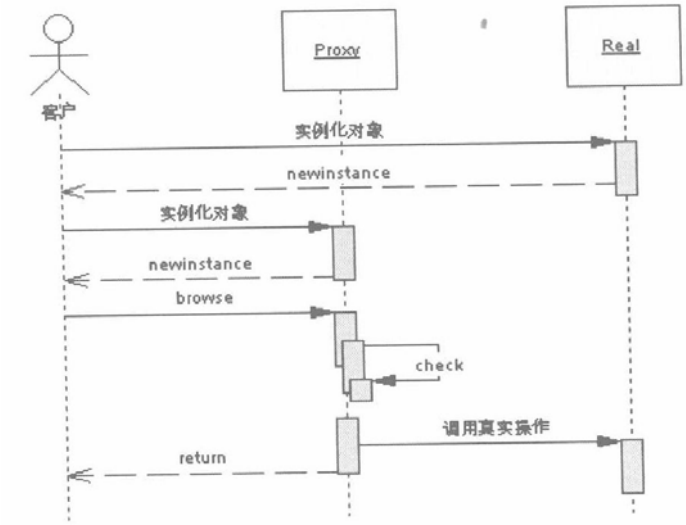


图 6-19　代理操作

从图 6-19 可以看出，真实主题完成的只是上网的最基本功能，而代理主题要做比真实主题更多的业务相关的操作。

6.9.6　设计模式——适配器设计

对于 Java 程序来说，如果一个类要实现一个接口，则必须要覆写此接口中的全部抽象

方法，那么如果此时一个接口中定义的抽象方法过多，但是在子类中又用不到这么多抽象方法，则肯定很麻烦，所以此时就需要一个中间的过渡，但是此过渡类又不希望被直接使用，所以将此过渡类定义成抽象类最合适，即一个接口首先被一个抽象类先实现（此抽象类通常称为适配器类），并在此抽象类中实现若干方法（方法体为空），则以后的子类直接继承此抽象类，就可以有选择地覆写所需要的方法，如图 6-20 所示。

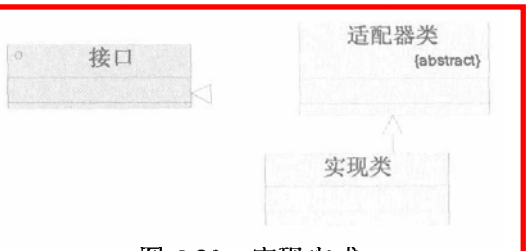


图 6-20　实现方式

范例：适配器设计实现

```
interface Window{                              // 定义Window接口，表示窗口操作
    public void open() ;                       // 窗口打开
    public void close() ;                      // 窗口关闭
    public void activated() ;                  // 窗口活动
    public void iconified() ;                  // 窗口最小化
    public void deiconified() ;                // 窗口恢复大小
}
// 定义抽象类实现接口，在此类中覆写方法，但是所有的方法体为空
abstract class WindowAdapter implements Window {
    public void activated() {}                 // 覆写activated()方法，方法体为空
    public void close() {}                     // 覆写close()方法，方法体为空
    public void deiconified() {}               // 覆写deiconified()方法，方法体为空
    public void iconified() {}                 // 覆写iconified()方法，方法体为空
    public void open() {}                      // 覆写open()方法，方法体为空
}
// 子类直接继承WindowAdapter类，有选择地实现需要的方法
class WindowImpl extends WindowAdapter{
    public void open(){                        // 真正实现open()方法
        System.out.println("窗口打开。") ;
    }
    public void close(){                       // 真正实现close()方法
        System.out.println("窗口关闭。") ;
    }
}
public class AdapterDemo {
    public static void main(String[] args) {
        Window win = new WindowImpl() ;// 实现接口对象
        win.open() ;                           // 调用open()方法
```

```
            win.close() ;                    // 调用close()方法
        }
    }
```

以上代码中因为采用了适配器这个中间环节，所以子类就不用必须实现接口中的全部方法，而是有选择地实现所需要的方法。

提示：在图形界面编程的事件处理中经常使用此设计模式。

在以后学习图形界面部分时，读者将看到大量的事件监听接口，如果全部实现方法则肯定不方便，所以在 Java 中将提供大量的适配器类供用户使用，读者从本程序掌握适配器的基本实现原理即可。

6.9.7 内部类的扩展

在面向对象的基础部分曾经为读者讲解过内部类的概念，实际上在一个抽象类中也可以定义多个接口或抽象类，在一个接口中也可以定义多个抽象类或接口。

范例：在一个抽象类中包含接口

```
abstract class A{                            // 定义抽象A
    public abstract void printA() ;          // 定义抽象方法
    interface B{                             // 定义内部接口
        public void printB() ;               // 定义抽象方法
    }
};
class X extends A{                           // 继承抽象类
    public void printA(){                    // 实现抽象方法
        System.out.println("HELLO --> A") ;
    }
    class Y implements B{                    // 定义内部类实现内部接口
        public void printB(){                // 实现内部接口的抽象方法
            System.out.println("HELLO --> B") ;
        }
    };
};
public class InnerExtDemo01 {
    public static void main(String args[]){
        A.B b = new X().new Y() ;            // 实例化内部接口对象
        b.printB() ;                         // 调用内部接口的抽象方法
    }
}
```

程序运行结果：

```
HELLO --> B
```

以上程序中在抽象类 A 中定义了一个内部接口 B，然后在抽象类的子类中也声明了一个内部类并且实现此内部接口，主方法中按照内部类被外部所访问的格式进行对象的实例化操作。

范例：在一个接口中包含抽象类

```
interface A{                                        // 定义接口A
    public abstract void printA() ;                 // 定义抽象方法
    abstract class B{                               // 定义内部抽象类
        public abstract void printB() ;             // 定义抽象方法
    }
};
class X implements A{
    public void printA(){                           // 实现接口的抽象方法
        System.out.println("HELLO --> A") ;
    }
    class Y extends B{                              // 在接口实现类中定义内部类
        public void printB(){                       // 实现内部抽象类的抽象方法
            System.out.println("HELLO --> B") ;
        }
    };
};
public class InnerExtDemo02 {
    public static void main(String args[]){
        A.B b = new X().new Y() ;                   // 实例化内部接口对象
        b.printB() ;                                // 调用内部抽象类的方法
    }
}
```

程序运行结果：

```
HELLO --> B
```

以上的程序组成结果与上一个程序类似，唯一不同的是在一个接口中定义了一个抽象类。

注意：在抽象类中可以定义多个内部抽象类，在接口中可以定义多个内部接口。

除了在抽象类中定义接口及在接口中定义抽象类外，对于抽象类来说也可以在内部定义多个抽象类，而一个接口也可以在内部定义多个接口。

6.9.8　抽象类与接口之间的关系

抽象类和接口在系统设计上都是用得最多的，表 6-3 中列出了两者的主要区别。

表 6-3 抽象类与接口的关系

序号	区别点	抽象类	接口
1	定义	包含一个抽象方法的类	抽象方法和全局常量的集合
2	组成	构造方法、抽象方法、普通方法、常量、变量	常量、抽象方法
3	使用	子类继承抽象类（extends）	子类实现接口（implements）
4	关系	抽象类可以实现多个接口	接口不能继承抽象类，但允许继承多个接口
5	常见设计模式	模板设计	工厂设计、代理设计
6	对象	都通过对象的多态性产生实例化对象	
7	局限	抽象类有单继承的局限	接口没有此局限
8	实际	作为一个模板	作为一个标准或表示一种能力
9	选择	如果抽象类和接口都可以使用，则优先使用接口，避免单继承的局限	
10	特殊	一个抽象类中可以包含多个接口，一个接口中可以包含多个抽象类	

在类的设计中，一定要明确记住一个原则，一个类不要去继承一个已经实现好的类，只能继承抽象类或实现接口，如果接口和抽象类都可以使用，那么优先使用接口，避免单继承局限。

注意：牢记以上的概念。

抽象类和接口在 Java 中是最重要的概念，这些概念不仅在开发中使用，在面试时也经常会被问到，所以读者一定要牢记以上的各个区别。

6.10 实例分析：宠物商店

设计一个“宠物商店”，在宠物商店中可以有多种（由用户决定数量）宠物，试表示出此种关系，并要求可以根据宠物的关键字查找到相应的宠物信息。所需要的宠物信息自行设计。

具体分析如下：

（1）本要求中提示宠物的信息可以自行设计，所以此时简单设计出名字、颜色、年龄 3 个属性。

（2）宠物的类别很多，如猫、狗等都属于宠物，所以宠物应该是一个标准。

（3）在宠物商店中，只要是符合了此宠物标准的就都应该可以放进宠物商店之中。

（4）宠物商店中要保存多种宠物，则肯定应该是一个宠物的对象数组，如果宠物的个数由用户决定，则应该在创建宠物商店时，就已经分配好宠物的个数。

根据以上分析可以得出如图 6-21 所示的图形。

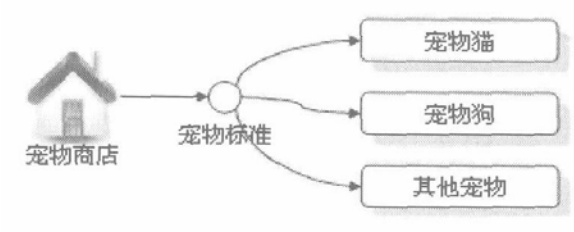


图 6-21　分析图

从图 6-21 中可以清楚地发现，宠物商店不管具体的宠物是哪一个，只要是宠物就可以放进去，所以此宠物的标准应该使用接口进行定义，每个具体的宠物都实现此接口，宠物商店与接口有关，可以得出如图 6-22 所示的类图关系。

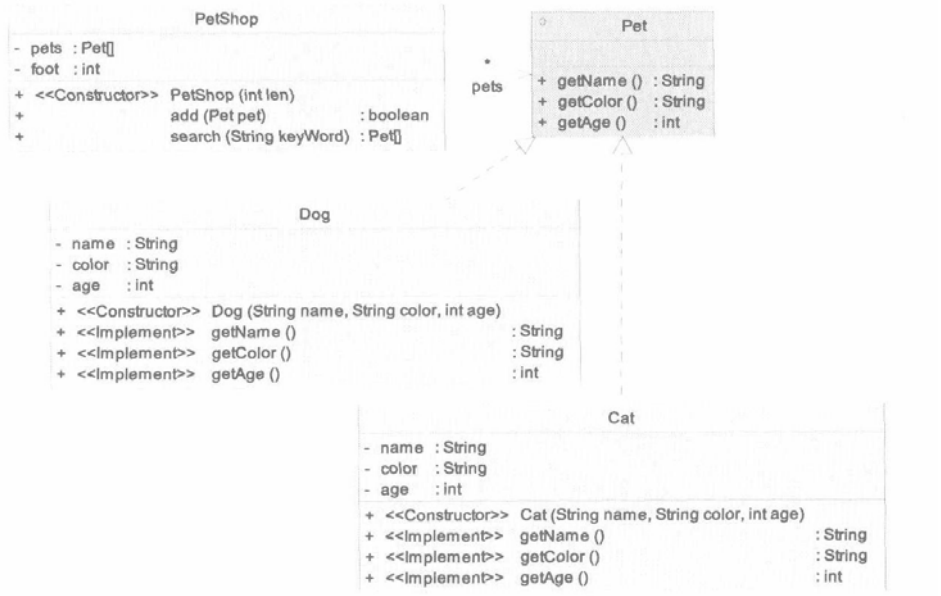


图 6-22　类图

从图 6-22 中可知，制订出了宠物的标准后，程序可以任意扩充具体的宠物，因为宠物商店只与宠物标准有关。

代码：宠物接口——Pet.java

```
interface Pet {
    public String getName();                    // 得到宠物的名字
    public String getColor();                   // 得到宠物的颜色
    public int getAge();                        // 得到宠物的年龄
}
```

然后根据此接口定义出具体的子类。

代码：宠物猫——Cat.java

```
class Cat implements Pet {
    private String name;                        // 宠物名字
    private String color;                       // 宠物颜色
    private int age;                            // 宠物年龄
    public Cat(String name,
```

```
            String color, int age) {            // 通过构造设置属性
        this.setName(name) ;
        this.setColor(color) ;
        this.setAge(age) ;
    }
    public String getName() {
        return this.name;
    }
    public String getColor() {
        return this.color;
    }
    public int getAge() {
        return this.age;
    }
    public void setName(String name) {
        this.name = name;
    }
    public void setColor(String color) {
        this.color = color;
    }
    public void setAge(int age) {
        this.age = age;
    }
};
```

代码：宠物狗——Dog.java

```
class Dog implements Pet {
    private String name;            // 宠物名字
    private String color;           // 宠物颜色
    private int age;                // 宠物年龄
    public Dog(String name,
            String color, int age) {// 通过构造设置属性
        this.name = name;
        this.color = color;
        this.age = age;
    }
    public String getName() {
        return this.name;
    }
    public String getColor() {
        return this.color;
    }
    public int getAge() {
```

```
        return this.age;
    }
    public void setName(String name) {
        this.name = name;
    }
    public void setColor(String color) {
        this.color = color;
    }
    public void setAge(int age) {
        this.age = age;
    }
};
```

此处只定义出了两种宠物，如果有更多的宠物，则只要实现 Pet 接口即可。下面开始定义宠物商店的操作类，在宠物商店中应该包含一个宠物接口的对象数组。

代码：宠物商店——PetShop.java

```
class PetShop {
    private Pet[] pets;                          // 保存多个属性
    private int foot;                            // 数据的保存位置
    public PetShop(int len) {                    // 构造方法开辟宠物数组的大小
        if(len>0){                               // 判断长度是否大于0
            this.pets = new Pet[len];            // 为对象数组开辟空间
        }else{
            this.pets = new Pet[1] ;             // 至少开辟一个空间
        }
    }
    public boolean add(Pet pet) {                // 增加宠物
        if (this.foot < this.pets.length) {// 判断宠物商店里的宠物是否已经满了
            this.pets[this.foot] = pet;          // 增加宠物
            this.foot++;                         // 修改保存位置
            return true;                         // 增加成功
        } else {
            return false;                        // 增加失败
        }
    }
    public Pet[] search(String keyWord){         // 关键字查找
        Pet p[] = null;                          // 此为查找之后的结果，此处的大小不
                                                    是固定的
        int count = 0;                           // 记录下多少个宠物符合查询结果
        // 确认开辟的空间大小，看有多少个宠物符合查询条件
        for (int i = 0; i < this.pets.length; i++) {
            if (this.pets[i] != null) {      //判断对象数组中的内容是否为空
```

```
                if (this.pets[i].getName().indexOf(keyWord) != -1
                        ||this.pets[i].getColor().indexOf(keyWord) != -1) {
                    count++;                    // 统计符合条件的宠物个数
                }
            }
        }
        p = new Pet[count];                     // 根据已经确定的记录数开辟对象数组
        int f = 0;                              // 设置增加的位置标记
        for (int i = 0; i < this.pets.length; i++) {
            if (this.pets[i] != null) {
                if (this.pets[i].getName().indexOf(keyWord) != -1
                        || this.pets[i].getColor().indexOf(keyWord)!= -1) {
                    p[f] = this.pets[i];    // 将符合查询条件的宠物信息保存
                    f++;
                }
            }
        }
        return p;
    }
};
```

以上的查询代码可能会比较复杂一些，下面简单为读者介绍以上代码的思路。

（1）在此方法设计时秉着一个原则，即所有的内容交给被调用处输出，类中不直接输出内容，所以本方法必须有一个返回值，因为查找出来的结果肯定是多个，所以返回值的类型定义为 Pet[]，表示返回一组查询到的宠物，所以方法名称设计如下：

```
public Pet[] search(String keyWord)
```

（2）一个宠物商店中会有多个宠物，符合查询条件的宠物只有几个，所以要想返回一个对象数组，则必须要确定好此数组需要开辟的空间的大小，操作代码如下：

```
        int count = 0;                          // 记录下多少个宠物符合查询结果
        // 确认开辟的空间大小，看有多少个宠物符合查询条件
        for (int i = 0; i < this.pets.length; i++) {
            if (this.pets[i] != null) {// 判断对象数组中的内容是否为空
                if (this.pets[i].getName().indexOf(keyWord) != -1
                        || this.pets[i].getColor().indexOf(keyWord)!= -1) {
                    count++;                    // 统计符合条件的宠物个数
                }
            }
        }
        p = new Pet[count];                     // 根据已经确定的记录数开辟对象数组
```

在以上代码中，查询使用的是 String 类中的 indexOf()方法，如果此方法的返回值不为-1，则表示已经找到了查询内容，因为在接口的定义中已经明确地定义了得到信息的操作，所以直接使用接口对象即可。

（3）为返回的对象数组开辟空间后，就要把每一个符合条件的对象向数组中依次加入，所以还需要再进行一次循环，操作代码如下：

```
int f = 0;                                    // 设置增加的位置标记
for (int i = 0; i < this.pets.length; i++) {
    if (this.pets[i] != null) {
        if (this.pets[i].getName().indexOf(keyWord) != -1
                || this.pets[i].getColor().indexOf(keyWord)!=-1) {
            p[f] = this.pets[i];// 将符合查询条件的宠物信息保存
            f++;
        }
    }
}
```

这样经过以上 3 步之后，就可以把全部符合查询条件的内容查找出来，并放在返回的对象数组之中。

下面对以上程序进行测试。

代码：测试宠物商店——PetShopDemo.java

```
public class PetShopDemo {
    public static void main(String args[]) {
        PetShop ps = new PetShop(5);                    // 5个宠物
        ps.add(new Cat("白猫", "白色的", 2));           // 增加宠物，成功
        ps.add(new Cat("黑猫", "黑色的", 3));           // 增加宠物，成功
        ps.add(new Cat("花猫", "花色的", 3));           // 增加宠物，成功
        ps.add(new Dog("拉布拉多", "黄色的", 3));       // 增加宠物，成功
        ps.add(new Dog("金毛", "金色的", 3));           // 增加宠物，成功
        ps.add(new Dog("黄狗", "黑色的", 3));           // 增加宠物，失败
        print(ps.search("黑"));
    }
    public static void print(Pet p[]) {                 // 输出操作
        for (int i = 0; i < p.length; i++) {            // 循环输出
            if (p[i] != null) {
                System.out.println(p[i].getName() +
                        "," + p[i].getColor() + ","
                        + p[i].getAge());
            }
        }
    }
}
```

程序运行结果：

```
黑猫，黑色的，3
```

在以上的程序中，宠物商店里只能存放 5 种宠物，所以再加入第 6 种宠物时就无法再

增加了，调用查询方法后返回的是一组宠物信息，所以直接定义一个 print()方法进行内容的输出即可。

6.11 Object 类

6.11.1 基本作用

在 Java 中所有的类都有一个公共的父类 Object，一个类只要没有明显地继承一个类，则肯定是 Object 类的子类。如下两行代码表示的含义都是一样的：

```
class Person extends Object{}
class Person{}
```

那么既然继承了 Object 类，Object 类有哪些作用呢？实际上在 Object 类中提供了很多的方法，这些方法经常被开发者所使用，Object 类中的主要方法如表 6-4 所示。

表 6-4 Object 类中的主要方法

序 号	方 法 名 称	类 型	描 述
1	public Object()	构造	构造方法
2	public boolean equals(Object obj)	普通	对象比较
3	public int hashCode()	普通	取得 Hash 码
4	public String toString()	普通	对象打印时调用

以上只是列出了 Object 类中比较常用的几个方法，其他的方法在本书的后续章节中会为读者介绍。那么以上的这些方法有哪些用处呢？实际上对于一个设计良好的类来说，最好覆写 Object 类中的 equals、hashCode、toString 3 个方法。考虑到学习知识的层次性，本章暂时只对 equals 和 toString 方法的应用进行讲解，而 hashCode()方法在 Java 类集应用中再为读者介绍。

6.11.2 主要方法

在讲解 toString()之前，读者先来看以下的一段代码。

范例：观察程序输出结果

```
class Demo {                                    // 定义Demo类，实际上继承Object类
}
public class ObjectDemo01 {
    public static void main(String args[]) {
        Demo d = new Demo();                    // 实例化Demo对象
        System.out.println("不加toString()输出：" + d);
        System.out.println("加上toString()输出：" + d.toString());
    }
}
```

程序运行结果：

```
不加toString()输出：Demo@c17164
加上toString()输出：Demo@c17164
```

以上的程序是随机输出了一些地址信息，从程序的运行结果可以清楚地发现，加与不加 toString()最终的输出结果是一样的，也就是说对象输出时一定会调用 Object 类中的 toString()方法打印内容。所以利用此特性就可以通过 toString()取得一些对象的信息，如下面代码。

范例：应用 toString()方法取得对象内容

```
class Person {                                          // 定义Person类
    private String name ;                               // 定义name属性
    private int age ;                                   // 定义age属性
    public Person(String name,int age){                 // 通过构造设置属性内容
        this.name = name ;
        this.age = age ;
    }
    public String toString(){                           // 此处覆写toString()方法
        return "姓名：" + this.name + "；年龄：" + this.age ;
    }
}
public class ObjectDemo02 {
    public static void main(String args[]) {
        Person per = new Person("李兴华",30) ; // 实例化Person对象
        System.out.println("对象信息：" + per) ;// 打印对象调用toString()方法
    }
}
```

程序运行结果：

```
对象信息：姓名：李兴华；年龄：30
```

程序的 Person 类中覆写了 Object 类中的 toString()方法，这样直接输出对象时调用的是被子类覆写过的 toString()方法。

equals()方法的功能就是对象的比较，实际上 String 也是 Object 类的子类，所以在 String 中已经实现了此方法，如果现在一个类需要实现对象的比较操作，则直接在类中覆写此方法即可。

提示：Object 类提供的 equals()方法默认是比较地址的。

Object 类中的 equals()方法实际上也是可以使用的，但是其默认使用的是按地址进行比较，并不能对内容进行比较。

范例：对象的比较操作

```
class Person {                                  // 定义Person类
    private String name ;                       // 定义name属性
```

```
    private int age ;                          // 定义age属性
    public Person(String name,int age){// 通过构造设置属性内容
        this.name = name ;                     // 为name属性赋值
        this.age = age ;                       // 为age属性赋值
    }
    public boolean equals(Object obj){ // 覆写Object类中的equals()方法
        if(this==obj){                         // 如果两个对象的地址相等，则肯定是同一个
                                                  对象
            return true ;
        }
        if(!(obj instanceof Person)){  // 判断传进来的对象是否是Person的实例
            return false ;                     // 如果不是，则直接返回false
        }
        Person per = (Person)obj ;             // 将传进来的对象向下转型
        if(per.name.equals(this.name)
                &&per.age==this.age){  // 逐个属性比较，看是否相等
            return true ;                      // 对象内容相等
        }else{
            return false ;                     // 对象内容不等
        }
    }
    public String toString(){                  // 此处覆写toString()方法
        return "姓名：" + this.name + "；年龄：" + this.age ;
    }
}
public class ObjectDemo03 {
    public static void main(String args[]) {
        Person per1 = new Person("李兴华",30) ;
        Person per2 = new Person("李兴华",30) ;
        System.out.println(per1.equals(per2)?"是同一个人！":"不是同一个人！") ;
        System.out.println(per1.equals("hello")?"是同一个人！" : "不是同一个
        人！") ;
    }
}
```

程序运行结果：

```
是同一个人！
不是同一个人！
```

在 Person 类中覆写了 equals()方法，首先判断传递进来的对象与当前对象的地址是否相等，如果相等，则肯定是同一个对象，因为在 equals()方法处传递的参数是 Object 类型，所以任何对象都可以接收，这样在对象进行向下转型前就必须进行判断，判断传进来的对象是否是 Person 的实例，如果不是，则直接返回 false；如果是，则将各个属性依次进行判断。

在编写测试方法时，为了测试程序的合理性，传入了一个字符串类型的对象，从结果中可以清楚地看到，因为字符串不是 Person 类型，所以直接返回了 false。

6.11.3　接收任意引用类型的对象

既然 Object 类是所有对象的父类，则所有的对象都可以向 Object 进行转换，在这其中也包含了数组和接口类型，即一切的引用数据类型都可以使用 Object 进行接收。

范例：使用 Object 接收接口实例

```
interface A{                                    // 定义接口A
    public String getInfo() ;                   // 定义抽象方法
}
class B implements A{                           // 子类实现接口
    public String getInfo(){                    // 覆写接口中的抽象方法
        return "Hello World!!!" ;
    }
}
public class ObjectDemo04 {
    public static void main(String args[]) {
        A a = new B() ;                         // 为接口实例化
        Object obj = a ;                        // 对象向上转型
        A x = (A)obj ;                          // 对象向下转型
        System.out.println(x.getInfo()) ;
    }
}
```

程序运行结果：

```
Hello World!!!
```

通过以上代码可以发现，虽然接口不能继承一个类，但是依然是 Object 类的子类，因为接口本身是引用数据类型，所以可以进行向上转型操作。

同理，也可以使用 Object 接收一个数组，因为数组本身也是引用数据类型。

范例：使用 Object 接收数组

```
public class ObjectDemo05 {
    public static void main(String args[]) {
        int temp[] = {1,3,5,7,9} ;              // 定义数组
        Object obj = temp ;                     // 使用Object接收数组
        print(obj) ;                            // 传递数组引用
    }
    public static void print(Object o){         // 接收一个对象
        if(o instanceof int[]){                 // 判断对象的类型
            int x[] = (int[])o ;                // 向下转型
            for(int i=0;i<x.length;i++){        // 循环输出
```

```
                System.out.print(x[i] + "\t") ;
            }
        }
    }
}
```

程序运行结果：

```
1    3    5    7    9
```

以上程序使用 Object 接收一个整型数组，因为数组本身属于引用数据类型，所以可以使用 Object 接收数组内容，在输出时通过 instanceof 判断类型是否是一个整型数组，然后进行输出操作。

提示：Object 类在实际开发中的作用。

因为 Object 类可以接收任意的引用数据类型，所以在很多的类库设计上都采用 Object 作为方法的参数，这样操作起来会比较方便。

6.12 包 装 类

6.12.1 包装类介绍

在 Java 的设计中提倡一种思想，即一切皆对象，那么这样就出现了一个矛盾，从数据类型的划分中可以知道 Java 中的数据类型分为基本数据类型和引用数据类型，但是基本数据类型怎么能够称为对象呢？此时，就需要将基本数据类型进行包装，将 8 种基本类型变为一个类的形式，那么这也就是包装类的作用。包装类与基本数据类型的关系如表 6-5 所示。

表 6-5 包装类

序 号	基本数据类型	包 装 类
1	int	Integer
2	char	Character
3	short	Short
4	long	Long
5	float	Float
6	double	Double
7	boolean	Boolean
8	byte	Byte

在表 6-5 所列的类中，除了 Integer 和 Character 定义的名称与基本类型定义的名称相差较大外，其他的 6 种类型的名称都是很好掌握的，而且读者可以从 JDK 文档中发现包装类中的继承关系：

（1）Integer、Byte、Float、Double、Short、Long 都属于 Number 类的子类，Number

类本身提供了一系列的返回以上 6 种基本数据类型的操作。

（2）Character 属于 Object 的直接子类。

（3）Boolean 属于 Object 的直接子类。

Number 类是一个抽象类，主要是将数字包装类中的内容变为基本数据类型，Number 类中定义的方法如表 6-6 所示。

表 6-6　Number 类定义

序　号	方　法	类　型	描　述
1	public byte byteValue()	普通	以 byte 形式返回指定的数值
2	public abstract double doubleValue()	普通	以 double 形式返回指定的数值
3	public abstract float floatValue()	普通	以 float 形式返回指定的数值
4	public abstract int intValue()	普通	以 int 形式返回指定的数值
5	public abstract long longValue()	普通	以 long 形式返回指定的数值
6	public short shortValue()	普通	以 short 形式返回指定的数值

本章重点讲解的是 Integer、Float 类型，其他类型的具体操作读者可自行查阅 JDK 文档。

6.12.2　装箱与拆箱

在了解包装类的概念后，下面介绍包装类的装箱与拆箱的概念，其实这两个概念本身并不难理解，将一个基本数据类型变为包装类，这样的过程称为装箱操作，而将一个包装类变为基本数据类型的过程称为拆箱操作。

范例：装箱及拆箱操作

```
public class WrapperDemo01 {
    public static void main(String[] args) {
        int x = 30 ;                        // 声明一个基本数据类型
        Integer i = new Integer(x) ;        // 装箱：将基本数据类型变为包装类
        int temp = i.intValue() ;           // 拆箱：将一个包装类变为基本数据类型
    }
}
```

以上代码演示了一个将基本数据类型进行装箱以及将一个包装类拆箱的操作，为了加深读者的理解，下面再看一个将浮点数装箱及拆箱的例子。

范例：浮点数装箱及拆箱

```
public class WrapperDemo02 {
    public static void main(String[] args) {
        float f = 30.3f ;                   // 声明一个基本数据类型
        Float x = new Float(f) ;            // 装箱：将基本数据类型变为包装类
        float y = x.floatValue() ;          // 拆箱：将一个包装类变为基本数据类型
    }
}
```

以上程序的功能与之前的类似，只是换了一个数据类型而已。而且读者也可以发现，对于拆箱操作中使用的方法就是 Number 类中定义的方法。

以上程序都要求进行手工装箱及拆箱操作，这是在 JDK 1.5 之前的做法，在 JDK 1.5 之后提供了自动的装箱及拆箱操作。

范例：自动装箱及拆箱操作

```
public class WrapperDemo03 {
    public static void main(String[] args) {
        Integer i = 30 ;                    // 自动装箱成Integer
        Float f = 30.3f ;                   // 自动装箱成Float
        int x = i ;                         // 自动拆箱为int
        float y = f ;                       // 自动拆箱为float
    }
}
```

可以发现，以上代码完成了自动的装箱及拆箱操作，不用再像之前那样进行手工操作，为开发者提供了更多的方便。

6.12.3 包装类应用

前面已经为读者详细地介绍了包装类的基本概念，那么包装类在实际中用得最多的还是字符串变为基本数据类型的操作，例如，将一个全由数字组成的字符串变为一个 int 或 float 类型的数据。在 Integer 和 Float 类中分别提供了以下两种方法：

（1）Integer 类（字符串转 int 型）

```
public static int parseInt(String s) throws NumberFormatException
```

（2）Float 类（字符串转 float 型）

```
public static float parseFloat(String s) throws NumberFormatException
```

但是在使用以上的两种操作时，一定要注意字符串必须由数字组成。

提示：方法声明的 throws 说明。

通过上述两种方法可以发现，在这两类操作语句中均存在 throws 语句，这表示的是异常处理语句，在之后的异常的捕获及处理章节将为读者介绍，此处可以暂不理解此语句作用。

范例：字符串变为基本数据类型

```
public class WrapperDemo04 {
    public static void main(String[] args) {
        String str1 = "30" ;                    // 由数字组成的字符串
        String str2 = "30.3" ;                  // 由数字组成的字符串
        int x = Integer.parseInt(str1) ;        // 将字符串变为int型
        float f = Float.parseFloat(str2) ;      // 将字符串变为float型
        System.out.println("整数乘方：" + x + " * " + x + " = " + (x * x));
        System.out.println("小数乘方：" + f + " * " + f + " = " + (f * f));
```

```
    }
}
```

程序运行结果：

```
整数乘方：30 * 30 = 900
小数乘方：30.3 * 30.3 = 918.08997
```

在这里要提醒读者的是，对于以上的转型操作，字符串中的数据必须由数字组成，否则转换时将会出现程序错误。

6.13 匿名内部类

在 Java 中除了内部类之外，还有一种匿名内部类。匿名内部类就是指没有一个具体名称的类，且是在接口和抽象类的应用上发展起来的，那么匿名内部类有哪些作用呢？例如，现在有如下的代码。

范例：一个简单的操作

```
interface A {                                    // 定义接口A
    public void printInfo();                     // 定义抽象方法
}
class B implements A {                           // 定义接口实现类
    public void printInfo() {                    // 覆写抽象方法
        System.out.println("Hello World!!!");
    }
}
class X {                                        // 定义X类
    public void fun1() {                         // 定义fun1()方法
        this.fun2(new B()) ;                     // 传递子类实例
    }
    public void fun2(A a) {                      // 接收接口实例
        a.printInfo() ;                          // 调用接口方法
    }
}
public class NoInnerClassDemo01 {
    public static void main(String args[]) {
        new X().fun1() ;                         // 实例化X类对象并调用fun1()方法
    }
}
```

程序运行结果：

```
Hello World!!!
```

相信读者对于本程序都应该很容易读懂，但是现在如果接口的实现类只使用一次，那

么还有必要单独定义一个子类 B 吗？很明显是没有必要的，所以此时就可以使用匿名内部类完成，代码修改如下。

范例：匿名内部类

```
interface A {                                    // 定义接口A
    public void printInfo();                     // 定义抽象方法
}
class X {                                        // 定义X类
    public void fun1() {                         // 定义fun1()方法
        this.fun2(new A() {                      // 匿名内部类
            public void printInfo() {            // 实现接口中的抽象方法
                System.out.println("Hello World!!!");
            }
        });
    }
    public void fun2(A a) {                      // 接收接口实例
        a.printInfo();                           // 调用接口方法
    }
}
public class NoInnerClassDemo02 {
    public static void main(String args[]) {
        new X().fun1();                          // 实例化X类对象并调用fun1()方法
    }
}
```

程序运行结果：

```
Hello World!!!
```

以上程序定义了一个匿名内部类，对于以上代码可能有一些读者看的不是很清楚，下面分步为读者解释此代码。

（1）直接实例化接口对象，代码如下：

```
new A() {}
```

此时，是直接为接口实例化，从前面的概念应该知道，接口本身是不能直接进行实例化的，所以在接口实例化后要有一个大括号，在其中编写具体的实现方法。

（2）实现抽象方法，代码如下：

```
new A() {                                        // 匿名内部类
    public void printInfo() {                    // 实现接口中的抽象方法
        System.out.println("Hello World!!!");
    }
}
```

整个代码编写完后，实际上此时就是一个接口的实例化对象了，其中的抽象方法也就实现了。

6.14　本 章 要 点

1．继承可以扩充已有类的功能，通过 extends 关键字实现，可将父类的成员（包含数据成员与方法）继承到子类。

2．Java 在执行子类的构造方法前会先调用父类中无参的构造方法，其目的是为了对继承自父类的成员做初始化的操作。

3．父类有多个构造方法时，如要调用特定的构造方法，则可在子类的构造方法中通过 super()关键字来实现。

4．this()用于在同一类内调用其他的构造方法，而 super()则用于从子类的构造方法中调用其父类的构造方法。

5．使用 this 调用属性或方法时会先从本类中查找，如果本类中没有查找到，再从父类中查找，而使用 super 则会直接从父类中查找需要的属性或方法。

6．this()与 super()其相似之处在于：（1）当构造方法有重载时，两者均会根据所给予的参数的类型与个数正确地执行相对应的构造方法。（2）两者均必须编写在构造方法内的第 1 行，也正是这个原因，this()与 super()无法同时存在在同一个构造方法内。

7．重载（overloading）是指在相同类内定义名称相同但参数个数或类型不同的方法，因此，Java 可依据参数的个数或类型调用相应的方法。

8．覆写（overriding）是在子类当中定义名称、参数个数与类型均与父类相同的方法，用以覆写父类中的方法。

9．如果父类的方法不希望被子类覆写，可在父类的方法前加上 final 关键字，这样该方法便不会被覆写。

10．final 的另一个功能是把它加在数据成员变量前面，这样该变量就变成了一个常量，因此便无法在程序代码中再做修改了。使用 public static final 可以声明一个全局常量。

11．所有的类均继承自 Object 类。一个好的类应该覆写 Object 类中的 toString()、equals()、hashCode() 3 个方法，所有的对象都可以向 Object 类进行向上转型。

12．Java 可以创建抽象类，专门用来当作父类。抽象类的作用类似于“模板”，其目的是依据其格式来修改并创建新的类。

13．抽象类的方法可分为两种，一种是一般的方法，另一种是以 abstract 关键字开头的抽象方法。抽象方法并没有定义方法体，而是要保留给由抽象类派生出的新类来定义。

14．抽象类不能直接用来产生对象，必须通过对象的多态性进行实例化操作。

15．接口是方法和全局常量的集合，接口必须被子类实现，一个接口可以同时继承多个接口，一个子类可以同时实现多个接口。

16．Java 并不允许类的多重继承，但是允许实现多个接口。

17．接口与一般类一样，均可通过扩展的技术来派生出新的接口。原来的接口称为基本接口或父接口；派生出的接口称为派生接口或子接口。通过这种机制，派生接口不仅可以保留父接口的成员，同时也可以加入新的成员以满足实际的需要。

18．Java 对象的多态性分为向上转型（自动）和向下转型（强制）。

19．通过 instanceof 关键字可以判断对象属于哪个类。

20．匿名内部类的作用是可利用内部类创建不具有名称的对象，并利用它访问类中的成员。

6.15 习 题

1．定义一个 ClassName 接口，接口中只有一个抽象方法 getClassName()；设计一个类 Company，该类实现接口 ClassName 中的方法 getClassName()，功能是获取该类的类名称；编写应用程序使用 Company 类。

2．考虑一个表示图形的类，写出类中的属性及方法。

3．建立一个人类（Person）和学生类（Student），功能要求如下：

（1）Person 中包含 4 个保护型的数据成员 name、addr、sex、age，分别为字符串型、字符串型、字符型及整型，表示姓名、地址、性别和年龄。用一个 4 参构造方法、一个两参构造方法、一个无参构造方法、一个输出方法显示 4 种属性。

（2）Student 类继承 Person 类，并增加输出成员 math、english 存放数学和英语成绩。用一个 6 参构造方法、一个两参构造方法、一个无参构造方法和重写输出方法用于显示 6 种属性。

4．定义员工类，具有姓名、年龄、性别属性，并具有构造方法和显示数据方法。定义管理层类，继承员工类，并有自己的属性职务和年薪。定义职员类，继承员工类，并有自己的属性所属部门和月薪。

5．定义类 Shape，用来表示一般二维图形。Shape 具有抽象方法 area 和 perimeter，分别用来计算形状的面积和周长。试定义一些二维形状类（如矩形、三角形、圆形、椭圆形等），这些类均为 Shape 类的子类。

6．使用面向对象的概念表示出下面的生活场景：

小明去超市买东西，所有买到的东西都放在了购物车之中，最后到收银台一起结账。

第 7 章　异常的捕获及处理

通过本章的学习可以达到以下目标：

- ☑ 了解异常的产生原理。
- ☑ 掌握异常处理语句的基本格式。
- ☑ 掌握 throw 和 throws 关键字的作用。
- ☑ 可以自定义异常。
- ☑ 了解 Exception 与 RuntimeException 的区别。
- ☑ 了解断言的作用。

在 Java 中程序的错误主要是语法错误和语义错误。一个程序即使在编译时没有错误信息产生，在运行时也有可能出现各种各样的错误导致程序退出，那么这些错误在 Java 中统一称为异常，在 Java 中对异常的处理提供了非常方便的操作。本章将介绍异常的基本概念以及相关的处理方式。**本章视频录像讲解时间为 1 小时 18 分钟，源代码在光盘对应的章节下。**

7.1　异常的基本概念

异常是导致程序中断运行的一种指令流，如果不对异常进行正确的处理，则可能导致程序的中断执行，造成不必要的损失，所以在程序的设计中必须要考虑各种异常的发生，并正确地做好相应的处理，这样才能保证程序正常地执行。在 Java 中一切的异常都秉着面向对象的设计思想，所有的异常都以类和对象的形式存在，除了 Java 中已经提供的各种异常类外，用户也可以根据需要定义自己的异常类。

提示：任何程序都可能存在问题。

在程序实际的应用中，可能存在大量的未知问题，所以在程序的开发中对于错误的处理是极其重要的。任何程序都很难做到百分百完美，所以程序开发中一定要对各种问题进行处理，而 Java 提供的异常处理机制可以帮用户更好地解决这方面的问题。

7.1.1　为什么需要异常处理

在没有异常处理的语言中就必须使用判断语句，配合所想到的错误状况来捕捉程序中所有可能发生的错误。但为了捕捉这些错误，编写出来的程序代码经常有大量的判断语句，有时这样也未必能捕捉到所有的错误，而且这样做势必导致程序运行效率的降低。

Java 的异常处理机制恰好改进了这一点。它具有易于使用、可自行定义异常类、处理抛出的异常同时又不会降低程序运行的速度等优点。因而在 Java 程序设计时，应充分地利

用 Java 的异常处理机制，以增进程序的稳定性及效率。

范例：认识异常

```
public class ExceptionDemo01 {
    public static void main(String args[]){
        System.out.println("********* 计算开始 **********") ;
        int i = 10 ;                                    // 定义整型变量
        int j = 0 ;                                     // 定义整型变量
        int temp = i / j ;                              // 此处会产生异常
        System.out.println("两个数字相除结果：" + temp) ;
        System.out.println("********* 计算结束 **********") ;
    }
}
```

程序运行结果：

```
********* 计算开始 **********
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at ExceptionDemo01.main(ExceptionDemo01.java:6)
```

在以上程序中，因为被除数为 0，所以程序中出现了异常，从运行结果可以发现，如果不对异常进行处理，则一旦出现了异常后，程序就立刻退出，所以后面的两条语句并没有打印输出。此程序的执行流程如图 7-1 所示。

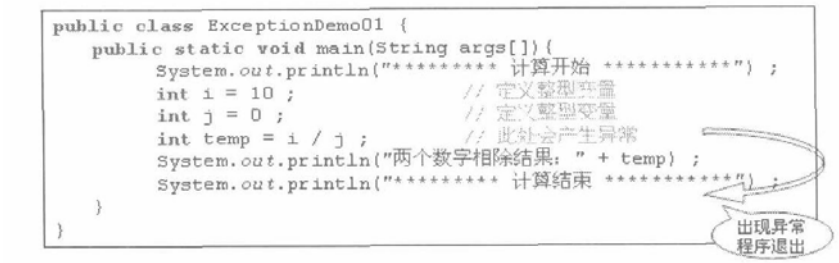


图 7-1　程序的流程

提示：关于被除数不能为 0 的说明。

在计算机的发展中有两大计算机“杀手”，一个是断电，另外一个是被除数为 0。因为被除数为 0 在数学上的解是无穷大，对于计算机来说，如果是无穷大，则意味着内存将全部被占满。

如果想保证以上的程序即使出现异常之后也可以正确执行，则必须进行异常的处理。

7.1.2　在程序中使用异常处理

在 Java 中异常处理语句的格式如下。

【格式 7-1　异常处理格式】

```
try{
    // 有可能出现异常的语句
```

```
}catch(异常类 异常对象){
    // 编写异常的处理语句
}[ catch(异常类 异常对象){
    // 编写异常的处理语句
} catch(异常类 异常对象){
    // 编写异常的处理语句
} ... ]
[finally{
    一定会运行到的程序代码;
}]
```

在格式 7-1 中已经明确地写出在 try 语句之中捕获可能出现的异常代码。如果在 try 中产生了异常，则程序会自动跳转到 catch 语句中找到匹配的异常类型进行相应的处理。最后不管程序是否会产生异常，则肯定都会执行到 finally 语句，finally 语句就作为异常的统一出口。需要提醒读者的是，finally 块是可以省略的。如果省略了 finally 块，则在 catch()块运行结束后，程序会跳到 try-catch 块之后继续执行。异常的处理流程如图 7-2 所示。

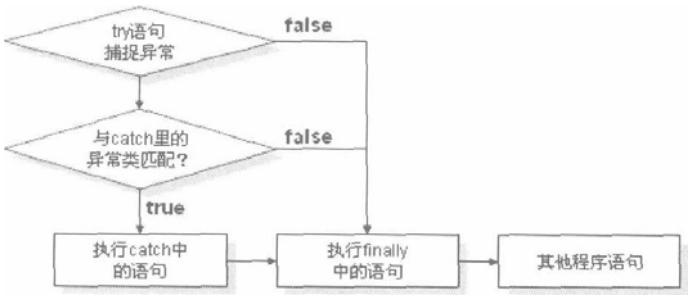


图 7-2　异常处理流程

范例：对异常进行捕捉

```
public class ExceptionDemo02 {
    public static void main(String args[]){
        System.out.println("********* 计算开始 **********") ;
        int i = 10 ;                                          // 定义整型变量
        int j = 0 ;                                           // 定义整型变量
        try{
            int temp = i / j ;                                // 此处会产生异常
            System.out.println("两个数字相除结果：" + temp) ;// 此代码不再执行
            System.out.println("----------------------");// 此代码不再执行
        }catch(ArithmeticException e){                        // 捕捉算术异常
            System.out.println("出现异常了：" + e) ;          // 出现异常执行异
                                                              常处理语句
        }
        System.out.println("********* 计算结束 **********") ;
    }
}
```

程序运行结果：

```
********* 计算开始 ***********
出现异常了：java.lang.ArithmeticException: / by zero
********* 计算结束 ***********
```

从程序运行结果可以清楚地发现，因为程序中加入了异常处理代码，所以当有异常发生后，整个程序也并不会因为异常的产生而中断执行。此程序的执行流程如图 7-3 所示。

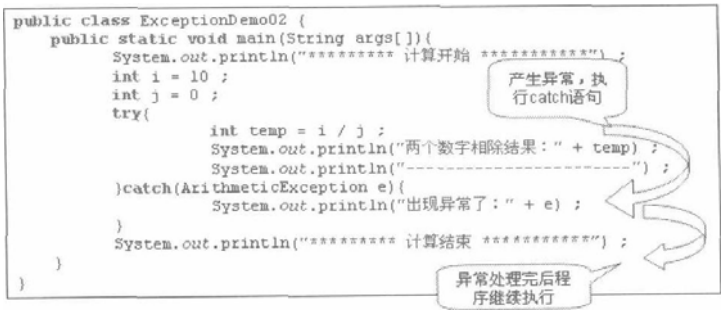


图 7-3　程序流程

从图 7-3 中可以清楚地看到，所有的异常都在 catch 中处理了，catch 处理完毕之后，程序正常结束。从格式 7-1 中可以发现，实际上在异常处理的最后有一个 finally 关键字，可以使用它作为异常的统一出口。

范例：验证 finally 关键字

```
public class ExceptionDemo03 {
    public static void main(String args[]){
        System.out.println("********* 计算开始 ***********") ;
        int i = 10 ;                                          // 定义整型变量
        int j = 0 ;                                           // 定义整型变量
        try{
            int temp = i / j ;                                // 此处会产生异常
            System.out.println("两个数字相除结果：" + temp) ; // 此代码不再执行
            System.out.println("----------") ;                // 此代码不再执行
        }catch(ArithmeticException e){                        // 捕捉算术异常
            System.out.println("出现异常了：" + e) ;           // 出现异常执行
                                                              异常处理语句
        }finally{                                             // 异常的统一出口
            System.out.println("不管是否出现异常，都执行此代码");
        }
        System.out.println("********* 计算结束 ***********") ;
    }
}
```

程序运行结果：

```
********* 计算开始 ***********
出现异常了：java.lang.ArithmeticException: / by zero
```

```
不管是否出现异常，都执行此代码
********** 计算结束 ************
```

当然在程序的开发中不会只存在一个异常，肯定会同时存在多个异常，此时，就需要使用多个 catch 语句进行处理。

范例：使用初始化参数输入两个数字，并进行除法操作

```
public class ExceptionDemo04 {
    public static void main(String args[]){
        System.out.println("********** 计算开始 ************") ;
        int i = 0 ;                                          // 定义整型变量
        int j = 0 ;                                          // 定义整型变量
        try{
            String str1 = args[0] ;                          // 接收第1个参数
            String str2 = args[1] ;                          // 接收第2个参数
            i = Integer.parseInt(str1) ;                     // 将第1个参数由字
                                                                符串变为整型
            j = Integer.parseInt(str2) ;                     // 将第2个参数由字
                                                                符串变为整型
            int temp = i / j ;                               // 此处会产生异常
            System.out.println("两个数字相除结果：" + temp) ;// 此代码不再执行
            System.out.println("----------------------") ;// 此代码不再执行
        }catch(ArithmeticException e){                       // 捕捉算术异常
            System.out.println("出现异常了：" + e) ;          // 出现异常执行异
                                                                常处理语句
        }
        System.out.println("********** 计算结束 ************") ;
    }
}
```

此时，因为要使用初始化输入参数，所以在程序执行时，输入以下的命令：

```
java ExceptionDemo04 10 2
```

程序运行结果：

```
********** 计算开始 ************
两个数字相除结果：5
----------------------
********** 计算结束 ************
```

第 1 个参数的值是 10，第 2 个参数的值是 2，所以两数相除的结果为 5。

提示：字符串转整型操作。

在以上程序中有这样几段代码：

```
String str1 = args[0] ;                                  // 接收第1个参数
String str2 = args[1] ;                                  // 接收第2个参数
```

```
i = Integer.parseInt(str1) ;                    // 将第1个参数由字符串变为整型
j = Integer.parseInt(str2) ;                    // 将第2个参数由字符串变为整型
```

其中 Integer.parseInt(字符串)是将输入的字符串变为 int 类型的数据，在第 6 章的包装类章节中已经为读者详细介绍过。

以上程序属于正确的运行，所以没有任何的问题，但是如果此时有以下的几种情况，则运行肯定会出现问题：

（1）没有输入参数或输入的参数不够，程序运行会出现以下的错误提示：

```
********* 计算开始 ***********
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
```

（2）运行时参数输入的不是数字，例如，使用以下的命令运行：

```
java ExceptionDemo04 a b
```

则程序运行时会出现以下的错误提示：

```
********* 计算开始 ***********
Exception in thread "main" java.lang.NumberFormatException:For input string:"a"
```

（3）输入的被除数是 0，此错误已经被捕捉了。

以上的程序实际上产生了 3 个比较明显的异常。

① 数组超出绑定异常：ArrayIndexOutOfBoundsException。

② 数字格式化异常：NumberFormatException。

③ 算术异常：ArithmeticException。

此时如果要想保持程序的执行正确，就必须同时对 3 个异常进行处理，所以此时的 catch 语句应该有 3 个，以分别处理不同的异常。

范例：捕捉多个异常

```
public class ExceptionDemo05 {
    public static void main(String args[]){
        System.out.println("********* 计算开始 ***********") ;
        int i = 0 ;                                          // 定义整型变量
        int j = 0 ;                                          // 定义整型变量
        try{
            String str1 = args[0] ;                          // 接收第1个参数
            String str2 = args[1] ;                          // 接收第2个参数
            i = Integer.parseInt(str1) ;                     // 将第1个参数由字
                                                                符串变为整型
            j = Integer.parseInt(str2) ;                     // 将第2个参数由字
                                                                符串变为整型
            int temp = i / j ;                               // 此处会产生异常
            System.out.println("两个数字相除结果：" + temp) ;// 此代码不再执行
            System.out.println("----------------------") ;// 此代码不再执行
        }catch(ArithmeticException e){                       // 捕捉算术异常
```

```
            System.out.println("算术异常：" + e) ;          // 处理算术异常
        }catch(NumberFormatException e){
            System.out.println("数字转换异常：" + e) ;      // 处理数字转换异常
        }catch(ArrayIndexOutOfBoundsException e){
            System.out.println("数组越界异常：" + e) ;      // 处理数组越界
        }
        System.out.println("********* 计算结束 **********") ;
    }
}
```

以上程序中使用了 3 个 catch 语句以分别处理 3 个不同的异常，但是如果每个程序的异常都使用这种方式处理则会很麻烦，因为在程序的开发过程中很难知道到底会有多少个异常。如果要解决这个问题，则首先要从异常的整体结构来看。

提示：Java 的异常处理格式也在不断改变。

在正常的异常处理中，异常处理的语句格式有 try…catch 和 try…catch…finally 两种。

以上是最标准的做法，但是随着 Java 的发展，对于异常的处理语句也可以采用 try-finally 的形式编写。

可以不写 catch 语法，但是这样的操作没有多大意义，有兴趣的读者可以自行研究。

7.1.3　异常类的继承结构

在整个 Java 的异常结构中，实际上有两个最常用的类，分别为 Exception 和 Error，这两个类全都是 Throwable 的子类，如图 7-4 所示。

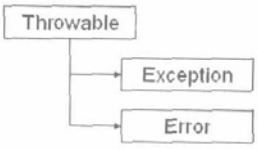


图 7-4　异常结构

- Exception：一般表示的是程序中出现的问题，可以直接使用 try…catch 处理。
- Error：一般指的是 JVM 错误，程序中无法处理。

一般情况下，开发者习惯于将 Exception 和 Error 统称为异常，而算术异常、数字格式化异常等都属于 Exception 的子类。

提示：异常信息的输出方式。

在 catch 语句输出异常时，除了可以直接使用“System.out.println(异常对象)”的方式打印异常信息外，有时也会直接使用 Exception 类中的 printStackTrace()方法输出异常信息，代码如下所示：

```
e.printStackTrace() ;
```

使用这种方式打印的异常信息是最完整的，所以在本书后面的有些章节也会使用上面的语法形式输出异常信息。

7.1.4 Java 的异常处理机制

在整个 Java 的异常处理中，实际上也是按照面向对象的方式进行处理，处理的步骤如下：

（1）一旦产生异常，则首先会产生一个异常类的实例化对象。

（2）在 try 语句中对此异常对象进行捕捉。

（3）产生的异常对象与 catch 语句中的各个异常类型进行匹配，如果匹配成功，则执行 catch 语句中的代码。

以上过程如图 7-5 所示。

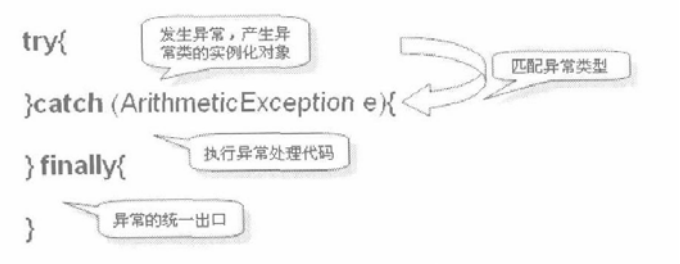


图 7-5 异常的处理步骤

从之前学习过的对象多态性可以清楚地知道，所有的子类实例可以全部使用父类接收，那么就可以利用向上转型的概念，让所有的异常对象都使用 Exception 接收。

范例：使用 Exception 处理其他异常

```
public class ExceptionDemo06 {
    public static void main(String args[]){
        System.out.println("********** 计算开始 ***********") ;
        int i = 0 ;                                          // 定义整型变量
        int j = 0 ;                                          // 定义整型变量
        try{
            String str1 = args[0] ;                          // 接收第1个参数
            String str2 = args[1] ;                          // 接收第2个参数
            i = Integer.parseInt(str1) ;                     // 将第1个参数由字
                                                                符串变为整型
            j = Integer.parseInt(str2) ;                     // 将第2个参数由字
                                                                符串变为整型
            int temp = i / j ;                               // 此处会产生异常
            System.out.println("两个数字相除结果：" + temp) ;// 此代码不再执行
            System.out.println("----------------------") ;// 此代码不再执行
        }catch(ArithmeticException e){                       // 捕捉算术异常
            System.out.println("算术异常：" + e) ;            // 处理算术异常
        }catch(NumberFormatException e){
            System.out.println("数字转换异常：" + e) ;        // 处理数字转换异常
        }catch(ArrayIndexOutOfBoundsException e){
            System.out.println("数组越界异常：" + e) ;        // 处理数组越界
```

```
        }catch(Exception e){
            System.out.println("其他异常：" + e) ;                    // 处理其他异常
        }
        System.out.println("********* 计算结束 **********") ;
    }
}
```

以上程序在最后直接使用 Exception 进行其他异常的捕获，那么本程序出现的全部异常就都可以处理了。但是要注意的是，在 Java 中所有捕获范围小的异常必须放在捕获大的异常之前，否则程序在编译时就会出现错误提示。

范例：错误的异常处理

```
public class ExceptionDemo07 {
    public static void main(String args[]){
        System.out.println("********* 计算开始 **********") ;
        int i = 0 ;                                                  // 定义整型变量
        int j = 0 ;                                                  // 定义整型变量
        try{
            String str1 = args[0] ;                                  // 接收第1个参数
            String str2 = args[1] ;                                  // 接收第2个参数
            i = Integer.parseInt(str1) ;                             // 将第1个参数由字
                                                                        符串变为整型
            j = Integer.parseInt(str2) ;                             // 将第2个参数由字
                                                                        符串变为整型
            int temp = i / j ;                                       // 此处会产生异常
            System.out.println("两个数字相除结果：" + temp) ;// 此代码不再执行
            System.out.println("----------------------") ;// 此代码不再执行
        }catch(Exception e){                                         // 异常捕获范围大
            System.out.println("其他异常：" + e) ;                    // 处理其他异常
        }catch(ArithmeticException e){                               // 错误，无法再捕
                                                                        捉算术异常
            System.out.println("算术异常：" + e) ;                    // 处理算术异常
        }
        System.out.println("********* 计算结束 **********") ;
    }
}
```

程序编译时错误：

```
ExceptionDemo07.java:16: exception java.lang.ArithmeticException has
already been caught
            }catch(ArithmeticException e){          // 错误，无法再捕捉算术异常
             ^
1 error
```

以上错误提示为“算术异常”已经被捕捉了，因为 Exception 捕捉的范围最大，所以以后的全部异常都是不可能处理的，所以一般在开发时，不管出现任何异常时都可以直接使用 Exception 进行处理，这样会比较方便。

范例：使用 Exception 处理异常

```
public class ExceptionDemo08 {
    public static void main(String args[]){
        System.out.println("********** 计算开始 ***********") ;
        int i = 0 ;                                          // 定义整型变量
        int j = 0 ;                                          // 定义整型变量
        try{
            String str1 = args[0] ;                          // 接收第1个参数
            String str2 = args[1] ;                          // 接收第2个参数
            i = Integer.parseInt(str1) ;                     // 将第1个参数由字
                                                                符串变为整型
            j = Integer.parseInt(str2) ;                     // 将第2个参数由字
                                                                符串变为整型
            int temp = i / j ;                               // 此处会产生异常
            System.out.println("两个数字相除结果：" + temp) ;// 此代码不再执行
            System.out.println("----------------------") ;// 此代码不再执行
        }catch(Exception e){                                 // 异常捕获范围大
            System.out.println("其他异常：" + e) ;           // 处理异常
        }
        System.out.println("********** 计算结束 ***********") ;
    }
}
```

以上代码直接使用 Exception 进行异常的处理，所以任何的异常都可以非常方便地进行处理。

提问：可不可以直接使用 Throwable？

既然可以使用 Exception 方便地捕获所有异常，那么以后在程序中直接使用 Throwable 的类不是更好吗？

回答：不建议这样使用，最大只能捕获 Exception。

首先使用 Throwable 捕获异常，这在代码中没有任何的问题，因为 Throwable 捕获的范围是最大的。但一般开发中是不会直接使用 Throwable 进行捕获的，对于 Throwable 来说有 Exception、Error 两个子类，Error 类本身不需要程序处理，而程序中需要处理的只是 Exception，所以没必要使用 Throwable。

另外，要提醒读者的是，对于一个程序来说，如果有多个异常最好分别进行捕获，而不要直接使用 Exception 捕获全部异常。

7.2　throws 与 throw 关键字

7.2.1　throws 关键字

在定义一个方法时可以使用 throws 关键字声明，使用 throws 声明的方法表示此方法不处理异常，而交给方法的调用处进行处理，throws 使用格式如下。

【格式 7-2　throws 使用格式】

```
public 返回值类型 方法名称(参数列表…) throws 异常类{}
```

范例：使用 throws 关键字

```
class Math{
    public int div(int i,int j) throws Exception{        // 方法可以不处理异常
        int temp = i / j ;
        return temp ;
    }
}
```

因为除法操作有可能出现异常，也有可能没有异常，所以在上面代码中的 div()方法中使用了 throws 关键字，表示不管是否会有异常，在调用此方法处都必须进行异常处理，代码如下。

范例：处理异常

```
class Math{
    public int div(int i,int j) throws Exception{  // 本方法中可以不处理异常
        int temp = i / j ;                         // 此处有可能产生异常
        return temp ;                              // 返回计算结果
    }
}
public class ThrowsDemo01 {
    public static void main(String args[]){
        Math m = new Math() ;                      // 实例化Math对象
        try {                          // 因为有throws，不管是否有异常，都必须处理
            System.out.println("除法操作：" + m.div(10, 2));
        } catch (Exception e) {
            e.printStackTrace();                   // 打印异常
        }
    }
}
```

程序运行结果：

```
除法操作：5
```

在以上代码中，不管是否有问题，都要使用 try…catch 块进行异常的捕获与处理。既然

throws 是在方法处定义的，那么主方法也可以使用 throws 关键字，但是主方法是程序的起点，所以此时主方法再向上抛异常，则只能将异常抛给 JVM 进行处理。这就好比一个公司，部门事情处理不了，要上报给经理，经理再解决不了要上报给董事长，到董事长了，也就到头了，那么 Java 的“头”就是 JVM。

范例：在主方法处使用 throws 关键字

```
class Math{
    public int div(int i,int j) throws Exception{  // 本方法中可以不处理异常
        int temp = i / j ;                          // 此处有可能产生异常
        return temp ;                               // 返回计算结果
    }
}
public class ThrowsDemo02 {
    // 本方法中的所有异常都可以不使用try…catch处理的
    public static void main(String args[]) throws Exception{
        Math m = new Math() ;                       // 实例化Math对象
        System.out.println("除法操作：" + m.div(10, 2));
    }
}
```

程序运行结果：

```
除法操作：5
```

以上代码在主方法处使用了 throws 关键字，所以在程序主方法中就可以不再使用 try…catch 语句进行异常的捕获及处理。

7.2.2 throw 关键字

与 throws 不同的是，可以直接使用 throw 抛出一个异常，抛出时直接抛出异常类的实例化对象即可。

范例：抛出异常

```
public class ThrowDemo01 {
    public static void main(String args[]){
        try{
            throw new Exception("自己抛出的异常！") ; // 抛出异常的实例化对象
        }catch(Exception e){                          // 捕获异常
            System.out.println(e);
        }
    }
}
```

程序运行结果：

```
java.lang.Exception: 自己抛出的异常！
```

以上代码并不难以理解，因为异常产生时肯定会由系统产生一个异常类的实例化对象，只是此时异常类的实例化对象是手工产生的。

注意：throw 不会单独使用。

throw 关键字的使用完全符合异常的处理机制，但是，一般来讲用户都在避免异常的产生，所以不会手工抛出一个新的异常类的实例，而往往会抛出程序中已经产生的异常类实例，这一点在 7.2.3 节中可以清楚地发现。

7.2.3　范例——throw 与 throws 的应用

学习了异常的处理格式、throw、throws 关键字之后，读者可能会有这样的疑问，这些技术到底该如何应用呢？实际上，在开发中，像 try…catch…finally、throw、throws 联合使用的情况是比较多的，例如，现在要设计一个相除的方法，但是要求必须打印“计算开始”信息、“计算结束”信息，如果有异常则肯定交给被调用处处理，面对这样的要求就必须把所有的技术点都使用上，具体代码如下。

范例：综合应用

```
class Math {
    public int div(int i, int j) throws Exception {// 方法可以不处理异常
        System.out.println("***** 计算开始 *****");
        int temp = 0;                                    // 声明整型变量
        try {
            temp = i / j;                                // 如果产生异常，则执行
                                                            catch
        } catch (Exception e) {                          // 捕获异常
            throw e;                                     // 把异常交给被调用处
        } finally {                                      // 不管是否产生异常都执行
                                                            此代码
            System.out.println("***** 计算结束 *****");
        }
        return temp;
    }
}
public class ThrowDemo02 {
    public static void main(String args[]) {
        Math m = new Math();                             // 实例化Math对象
        try {
            System.out.println("除法操作：" + m.div(10, 0));
        } catch (Exception e) {                          // 进行异常的捕获
            System.out.println("异常产生：" + e);
        }
    }
}
```

程序运行结果：

```
***** 计算开始 *****
***** 计算结束 *****
异常产生：java.lang.ArithmeticException: / by zero
```

从程序中可以发现，通过合理的搭配程序完成了需要的功能，不管在 Math 类中的 div() 方法是否会产生异常，都会执行“计算结束”的语句，如果有异常产生则将异常交给调用处进行处理。本程序的具体执行流程如图 7-6 所示。

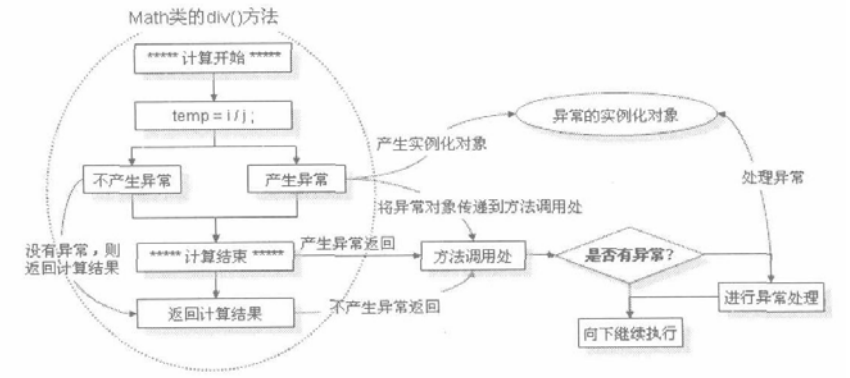


图 7-6　程序执行流程

以上的代码结构也是在日后的开发中使用得最多的形式，希望读者可以反复体会。

注意：finally 语句块的编写要求。

finally 作为异常的统一出口，所以在此语句块的编写中尽可能不要出现像 throw 或 return 这样的语句，这样可以避免不必要的问题出现。

7.3　Exception 类与 RuntimeException 类

在 Java 面试中经常会询问 Exception 类与 RuntimeException 类的区别，如果要想理解这两个类的区别，请看以下的一段代码。

范例：字符串变为整型

```
public class RuntimeExceptionDemo01 {
    public static void main(String args[]) {
        String str = "123" ;                    // 定义一个由数字组成的字符串
        int temp = Integer.parseInt(str) ;      // 将字符串变为int类型
        System.out.println(temp * temp);        // 计算乘方
    }
}
```

从以上将字符串变为整型的代码来看，Integer 因为开头首字母大写，所以肯定是一个类，而 parseInt()方法可以直接由类名称调用，所以此方法肯定是一个静态方法，此方法定义如下：

```
public static int parseInt(String s) throws NumberFormatException ;
```

应该要使用try catch 捕获异常的，但是可以不用

以上方法在声明时使用了 throws 关键字，但是在方法调用时并没有使用 try…catch 进行处理，这是为什么呢？下面观察 NumberFormatException 类的继承关系，如图 7-7 所示。

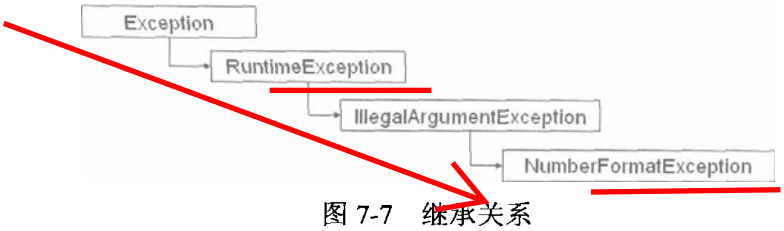


图 7-7　继承关系

从图 7-7 中可以发现，NumberFormatException 属于 RuntimeException 的子类，那么这时就可以清楚地知道以下概念：

- Exception 在程序中必须使用 try…catch 进行处理。
- RuntimeException 可以不使用 try…catch 进行处理，但是如果有异常产生，则异常将由 JVM 进行处理。

提示：对于 RuntimeException 的子类最好也使用异常处理机制。

虽然 RuntimeException 的异常可以不用 try…catch 进行处理，但是如果一旦出现异常，则肯定会导致程序中断执行，所以，为了保证程序在出错后依然可以执行，所以在开发代码时最好使用 try…catch 的异常处理机制进行处理。

7.4　自定义异常类

在 Java 中已经提供了大量的异常类，但是这些异常类有时也很难满足开发者的要求，所以用户可以根据自己的需要定义自己的异常类。定义异常类只需要继承 Exception 类即可。

范例：自定义异常类

```
class MyException extends Exception{                    // 自定义异常类，继承
                                                           Exception类
    public MyException(String msg){                     // 构造方法接收异常信息
        super(msg) ;                                    // 调用父类中的构造方法
    }
}
public class DefaultException {
    public static void main(String args[]) {
        try{
            throw new MyException("自定义异常。") ;      // 抛出异常
        }catch(Exception e){                            // 异常处理
            System.out.println(e) ;
        }
    }
}
```

程序运行结果：

```
MyException: 自定义异常。
```

7.5 断　　言

在 JDK 1.4 之后，Java 中增加了断言的功能。断言就是肯定某一个结果的返回值是正确的，如果最终此结果的返回值是错误的，则通过断言检查肯定会提示错误信息。断言的定义格式如下。

【格式 7-3　断言】

```
assert boolean表达式 ;
assert boolean表达式 : 详细的信息
```

如果以上 boolean 表达式的结果为 true，则什么错误信息都不会提示；如果为 false，则会提示错误信息；如果没有声明详细信息的描述，则系统会使用默认的错误信息提示方式。

范例：以下程序使用了断言

```
public class Test {
    public static void main(String args[]){
        int x[] = {1,2,3} ;          // 定义一个数组，长度为3
        assert x.length == 0 ;       // 此处断言数组长度为0，肯定是错误的
    }
}
```

以上程序中，数组 x 的长度是不可能为 0 的，所以，此处的断言结果是错误的，但此时运行程序并不会得到任何的结果，这是因为 Java 在设计此关键字时，考虑到了系统的应用，为了防止某些用户使用 assert 作为关键字，所以在程序正常运行时断言并不会起任何的作用，如果要想让断言起作用，则在使用 Java 运行时应该加入以下参数：

```
-enableassertions      → 可以简写为“-ea”
```

下面使用以上参数运行程序，运行程序格式如下：

```
编译程序：javac Test.java
验证程序：java -ea Test
```

程序运行时出现以下错误：

```
Exception in thread "main" java.lang.AssertionError
        at Test.main(Test.java:4)
```

以上是断言错误，因为数组 x 的长度不可能是 0，但是，此时的信息是系统默认的错误信息，如果要想显示自己的错误信息，则可以使用另外一种断言声明格式。

范例：Test 代码修改如下

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

```
        int x[] = {1,2,3} ;                          // 定义一个数组，长度为3
        assert x.length == 0 : "数组长度不为0" ; // 此处断言数组长度为0，肯定是
                                                     错误的
    }
}
```

再次验证断言，出现以下信息：

```
Exception in thread "main" java.lang.AssertionError: 数组长度不为0
        at Test.main(Test.java:4)
```

以上就是将自定义的错误信息进行输出，当然，如果此时断言的结果是正确的，则验证肯定不会有任何的问题。

注意：断言的使用。

（1）虽然断言返回的是 boolean 值，但是并不能将其作为条件判断语句。

（2）断言虽然有检查运行结果的功能，但是一般在开发中并不提倡使用断言。

7.6　本 章 要 点

1．异常是导致程序中断运行的一种指令流，当异常发生时，如果没有进行良好的处理，则程序将会中断执行。

2．异常可以使用 try…catch 进行处理，也可以使用 try…catch…finally 进行处理，在 try 语句中捕捉异常，然后在 catch 中处理异常，finally 作为异常的统一出口，不管是否发生异常都要执行此段代码。

3．异常的最大父类是 Throwable，其分为两个子类，分别为 Exception、Error。Exception 表示程序处理的异常，而 Error 表示 JVM 错误，一般不由程序开发人员处理。

4．发生异常后，JVM 会自动产生一个异常类的实例化对象，并匹配相应的 catch 语句中的异常类型，也可以利用对象的向上转型关系直接捕获 Exception。

5．throws 用在方法声明处，表示本方法不处理异常。

6．throw 表示在方法中手工抛出一个异常。

7．自定义异常类时，只需要继承 Exception 类即可。

8．断言是 JDK 1.4 后提供的新功能，可以用来检测程序的执行结果，但开发中并不提倡使用断言进行检测。

7.7　习　　题

1．编写应用程序，从命令行输入两个小数参数，求它们的商。要求程序中捕获 NumberFormatException 异常和 ArithmeticException 异常。

2．在第 6 章讲解的宠物商店程序中加入异常的处理操作。

第 8 章　包及访问控制权限

通过本章的学习可以达到以下目标：

- ☑ 掌握包的定义及使用。
- ☑ 掌握 Java 新特性——静态导入。
- ☑ 了解 Java 中的常用系统包。
- ☑ 掌握 jar 命令的使用。
- ☑ 掌握 Java 中的 4 种访问权限。
- ☑ 掌握 Java 语言的命名规范。

在 Java 中，可以将一个大型项目中的类分别独立出来，分门别类地存到文件里，再将这些文件一起编译执行，这样的程序代码将更易于维护，如图 8-1 所示。在将类分割开后对于类的使用也就有了相应的访问权限。本章将介绍如何使用包及访问控制权限。**本章视频录像讲解时间为 55 分钟，源代码在光盘对应的章节下。**

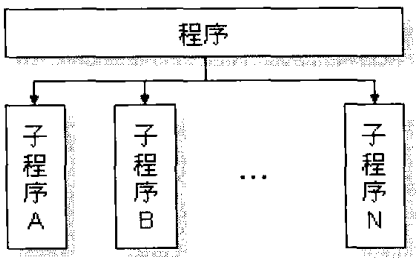


图 8-1　大型程序的拆分

8.1　包的概念及使用

8.1.1　包的基本概念

可以试想这样的一种情景，现在如果有多个开发人员共同开发同一个项目，则肯定会出现类名称相同的情况。这样一来就会比较麻烦，如图 8-2 所示。所以，可以利用 package 关键字来解决此问题。

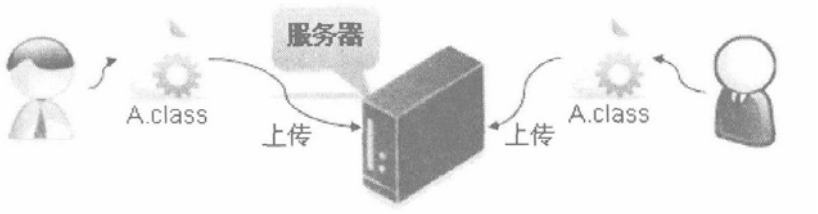


图 8-2　多人开发的问题

提示：多人开发。

在实际的开发中，所有的开发者都会将程序提交到一个统一的服务器上进行保存，实际上如果要对程序进行管理只使用包是不够的，还要对程序的更新、上传进行统一的控制。在实际开发中通常会配置一个版本控制工具帮助管理代码，读者可以自行查阅本书附录中讲解 CVS 的部分，以了解此类工具的使用方法。

package 是在使用多个类或接口时，为了避免名称重复而采用的一种措施，直接在程序中加入 package 关键字即可，格式如下。

【格式 8-1　包的定义】

```
package 包名称.子包名称 ;
```

下面使用以上格式为一个程序打包。

范例：为程序打包

```
package org.lxh.demo08;                                    // 定义一个包
class Demo{                                                // 定义Demo类
    public String getInfo(){                               // 返回信息
        return "Hello World!!!" ;
    }
}
public class PackageDemo01 {
    public static void main(String[] args) {
        System.out.println(new Demo().getInfo()) ; // 实例化本包对象
    }
}
```

以上程序的主要功能就是在屏幕上打印一个“Hello World!!!”字符串，但是因为程序是在包中的，所以此时完整的类名称应该为“包.类名称”，即“org.lxh.demo08.PackageDemo01”，在编译时就不能像之前那样编译，必须加上若干参数才行，编译语法如下：

```
javac -d . PackageDemo01.java
```

在以上的编译命令中加入了以下的两个参数。

- -d：表示生成目录，生成的目录以 package 的定义为准。
- .：表示在当前所在的文件夹中生成。

编译完成之后可以发现，程序会自动为程序进行打包操作，如图 8-3 所示。

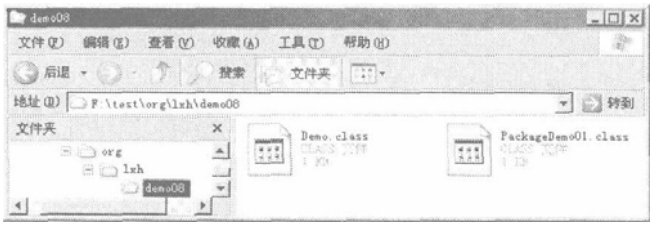


图 8-3　生成后的包

从图 8-3 中可以清楚地发现，以上命令执行完后，会自动生成相应的文件夹（包），之

后再执行类时就必须输入完整的“包.类名称”，执行语法如下：

```
java org.lxh.demo08.PackageDemo01
```

程序运行结果：

```
Hello World!!!
```

程序的运行结果与之前并没有什么不同，所以加入包只是让类的管理更加方便。

8.1.2 import 语句

8.1.1 节中所讲解的程序，两个类是存放在同一个包中的，因此代码与之前没有什么根本的不同，但是如果几个类存放在不同的包中，则在使用类时就必须通过 import 语句导入，import 的语法格式如下。

【格式 8-2　类的导入】

```
import 包名称.子包名称.类名称 ;          → 手工导入所需要的类
import 包名称.子包名称.* ;               → 由JVM自动加载所需要的类
```

以上两种格式本身并没有太大的区别，只是一般都比较习惯于使用后者，下面举例说明 import 命令的用法。此范例与 8.1.1 节的程序类似，只是将两个类分别放在不同的包中，如图 8-4 所示。

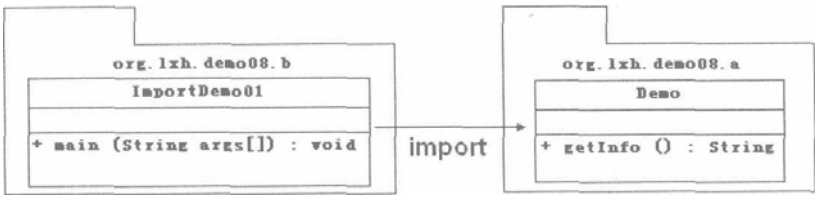


图 8-4　使用 import 导入包

范例：定义 org.lxh.demo08.a.Demo 类

```
package org.lxh.demo08.a;
class Demo{
    public String getInfo(){
        return "Hello World!!!" ;
    }
}
```

范例：在另外一个类中引用 Demo 类

```
package org.lxh.demo08.b;
import org.lxh.demo08.a.Demo ;                    // 导入不同包中的Demo类
public class ImportDemo01 {
    public static void main(String[] args) {
        System.out.println(new Demo().getInfo()) ;
    }
}
```

以上程序中的 ImportDemo01 类中因为要使用 Demo 类，所以在 package 语句后，通过 import 导入了所需要的类，所以在编译时应该先编译 Demo.java，再编译 ImportDemo01.java，编译语句如下。

- 编译 Demo.java：javac -d . Demo.java。
- 编译 ImportDemo01.java：javac -d . ImportDemo01.java。

但是在编译 ImportDemo01.java 时，编译器却提示出现了以下的错误：

```
ImportDemo01.java:2: org.lxh.demo08.a.Demo is not public in org.lxh.demo08.a;
cannot be accessed from outside package
import org.lxh.demo08.a.Demo ;                    // 导入不同包中的Demo类
                    ^
ImportDemo01.java:5: cannot find symbol
symbol  : class Demo
location: class org.lxh.demo08.b.ImportDemo01
            System.out.println(new Demo().getInfo()) ;
                               ^
2 errors
```

错误提示的意思是：Demo 类不是 public 定义的，所以在 ImportDemo01 中无法进行访问，也就是说如果一个类要被外包访问，则此类一定要定义成 public class。

提示：关于 public class 与 class 声明类的完整补充。

在本书开始曾经为读者讲解过，如果一个类声明为 public class，则文件名称必须与类名称一致，而且在一个类文件中只能有一个 public class，而如果使用 class 声明一个类，则文件名称可以与类名称不一致，但是执行时必须执行生成的 class 文件名称。除了这些之外，public class 和 class 还在包的访问上有所限制，如果一个类只在本包中访问，不需要被外包访问，则直接声明成 class 即可；而如果一个类需要被外包访问，则必须声明为 public class。

范例：修改 Demo 类的定义

```
package org.lxh.demo08.a;
public class Demo{
    public String getInfo(){
        return "Hello World!!!" ;
    }
}
```

再次编译 Demo 和 ImportDemo01 类就不会有任何的问题了，程序运行结果如下：

```
Hello World!!!
```

可以发现程序正常的编译通过，没有任何问题。如果在一个类中导入一个包中多个类时，一个个地导入会比较麻烦，可以使用导入“*”的方式由 JVM 根据需要自己加载所需的类。

范例：自动加载所需的类

```
package org.lxh.demo08.b;
import org.lxh.demo08.a.* ;                    // 导入不同包中的类
public class ImportDemo01 {
    public static void main(String[] args) {
        System.out.println(new Demo().getInfo()) ;
    }
}
```

以上程序的运行效果与之前的程序没有任何的区别。

③提问：分别导入与使用“*”导入哪种性能更高？

以上程序的两种导入方式如下。

- 第 1 种：“**import** org.lxh.demo08.a.Demo;”。
- 第 2 种：“**import** org.lxh.demo08.a.*;”。

从程序性能上来讲是不是第 1 种要比第 2 种性能更高呢？

回答：两种的性能是一样的。

这两种导入方式本身不存在任何的性能问题，因为使用“*”时程序也是自动加载所需要的类，而不需要的类根本是不会被加载进来的。

另外，还需要提醒读者注意的是，如果在一个程序中同时导入了两个包的同名类，在使用时就必须明确地写出完整的“包.类名称”，代码如下所示。

范例：定义另一个 Demo 类

```
package org.lxh.demo08.c;
public class Demo{
    public String getContent(){
        return "MLDN LXH" ;
    }
}
```

在程序中定义了一个新的包 org.lxh.demo08.c，类名称为 Demo，与 org.lxh.demo08.a 包中的 Demo 类重名，下面的程序同时导入了以上的两个包。

范例：同时导入不同包的相同类

```
package org.lxh.demo08.d;
import org.lxh.demo08.a.* ;                    // 包中存在Demo类
import org.lxh.demo08.c.* ;                    // 包中存在Demo类
public class ImportDemo02{
    public static void main(String[] args) {
        Demo d = new Demo() ;
        System.out.println(d.getInfo()) ;
    }
}
```

程序编译时出现以下错误：

```
    ImportDemo02.java:6: reference to Demo is ambiguous, both class
org.lxh.demo08.c.Demo in org.lxh.demo08.c and class org.lxh.demo08.a.Demo in
org.lxh.demo08.a match
                Demo d = new Demo() ;
                ^
    ImportDemo02.java:6: reference to Demo is ambiguous, both class
org.lxh.demo08.c.Demo in org.lxh.demo08.c and class org.lxh.demo08.a.Demo in
org.lxh.demo08.a match
                Demo d = new Demo() ;
                         ^
    2 errors
```

以上错误提示的意思是，导入的 Demo 类存在两个，所以程序在使用时出现了歧义，此时最好的做法是在声明 Demo 对象时明确地写出完整的“包.类名称”，具体代码如下。

范例：修改 ImportDemo02.java

```
package org.lxh.demo08.d;
import org.lxh.demo08.a.* ;                          // 包中存在Demo类
import org.lxh.demo08.c.* ;                          // 包中存在Demo类
public class ImportDemo02{
    public static void main(String[] args) {
        org.lxh.demo08.a.Demo d = new org.lxh.demo08.a.Demo() ;
        System.out.println(d.getInfo()) ;
    }
}
```

程序运行结果：

```
Hello World!!!
```

8.1.3　系统常见包

在 JDK 中为了方便用户开发程序，提供了大量的系统功能包，表 8-1 列出了一些常用的系统开发包。

表 8-1　Java 常用系统包

序　号	包 名 称	作　用
1	java.lang	此包为基本的包，String 都保存在此包之中，在 JDK 1.0 中如果想编写程序，则必须手工导入此包，但是随后的 JDK 解决了此问题，所以此包现在为自动导入
2	java.lang.reflect	此包为反射机制的包，是 java.lang 的子包，在 Java 反射机制中将会为读者介绍
3	java.util	此包为工具包，一些常用的类库、日期操作等都在此包中，如果把此包掌握精通，则各种设计思路都好理解

续表

序　号	包 名 称	作　用
4	java.text	提供了一些文本的处理类库
5	java.sql	数据库操作包，提供了各种数据库操作的类和接口
6	java.net	完成网络编程
7	java.io	输入、输出处理
8	java.awt	包含了构成抽象窗口工具集（abstract window toolkits）的多个类，这些类被用来构建和管理应用程序的图形用户界面（GUI）
9	javax.swing	此包用于建立图形用户界面，此包中的组件相对于 java.awt 包而言是轻量级组件

以上的大部分包在本书的后续章节会逐步为读者介绍，在此处只需要了解即可。

提示：关于包的补充说明。

java.util 包在开发中使用较多，因为其支持大量的工具类操作，像本书中的常用类库、Java 类集都是围绕此包展开讲解的。

java.lang.reflect 这个包一般在面试时有可能会问到，此包属于比较高级的开发包。

8.1.4 Java 新特性——静态导入

在 JDK 1.5 之后提供了静态导入功能。如果一个类中的方法全部是使用 static 声明的静态方法，则在导入时就可以直接使用 import static 的方式导入，导入的格式如下。

【格式 8-3　静态导入】

```
import static 包.类.* ;
```

下面介绍如何使用静态导入。

范例：定义一个类，全部由静态方法组成

```
package org.lxh.demo08.e;
public class Operate {                          // 里面的方法全部是static类型
    public static int add(int i, int j) {      // 加法操作
        return i + j;
    }
    public static int sub(int i, int j) {      // 减法操作
        return i - j;
    }
    public static int mul(int i, int j) {      // 乘法操作
        return i * j;
    }
    public static int div(int i, int j) {      // 除法操作
        return i / j;
    }
}
```

以上代码的类中的全部方法为 static 类型，所以在导入时就可以使用静态导入的方式。

范例：使用静态导入

```
package org.lxh.demo08.f;
import static org.lxh.demo08.e.Operate.*;                    // 静态导入
public class StaticImportDemo {
    public static void main(String[] args) {
        System.out.println("3 + 3 = " + add(3, 3));          // 直接调用静态方法
        System.out.println("3 - 2 = " + sub(3, 2));          // 直接调用静态方法
        System.out.println("3 * 3 = " + mul(3, 3));          // 直接调用静态方法
        System.out.println("3 / 3 = " + div(3, 3));          // 直接调用静态方法
    }
}
```

程序运行结果：

```
3 + 3 = 6
3 - 2 = 1
3 * 3 = 9
3 / 3 = 1
```

以上程序中使用了静态导入，所以 Operate 类中的所有静态方法可以直接在 StaticImportDemo 类中使用，而不再需要使用“类.静态方法()”的形式调用。

8.2　jar 命令的使用

当开发者为客户开发出了一套 Java 类之后，肯定要把这些类交给用户使用，但是如果所有的类直接通过*.class 的格式给用户，会比较麻烦，所以一般情况下会将这些*.class 文件压缩成一个文件交付给客户使用，那么这样的文件就称为 jar 文件（Java Archive File）。如果要想生成 jar 文件，直接使用 JDK 中 bin 目录里的 jar.exe 就可以将所有的类文件进行压缩，此命令是随 JDK 一起安装的，直接在命令行中输入 jar，即可看到此命令的提示界面，如图 8-5 所示。

当用户得到一个 jar 文件后，即可通过设置 classpath 的方式在系统中注册好此 jar 文件，以供程序使用。

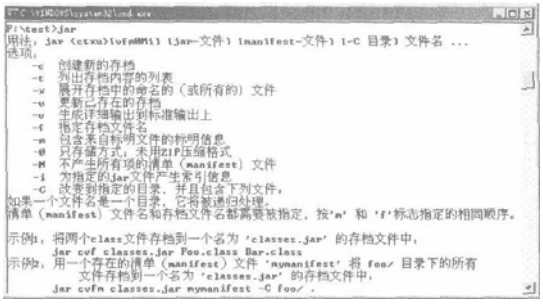


图 8-5　jar 命令

jar 命令中的主要参数如下。

- C：创建新的文档。
- V：生成详细的输出信息。
- F：指定存档的文件名。

下面使用 jar 命令进行文件的打包操作，为了方便读者理解，此处只为一个单独的类进行打包。

范例：定义一个简单的类

```
package org.lxh.demo08.demo;
public class Hello {
    public String getInfo() {
        return "Hello World!!!";
    }
}
```

然后对此程序进行编译，将生成如图 8-6 所示的目录。

将生成后的 org 的文件夹进行打包，此时会输出如图 8-7 所示的信息。

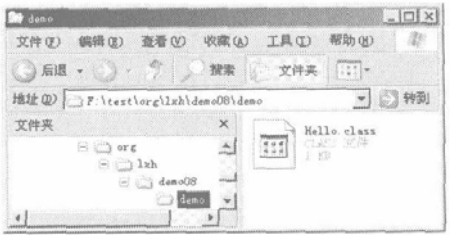


图 8-6　生成的目录

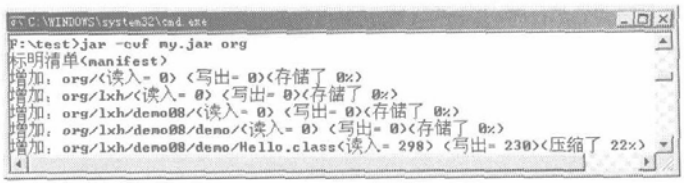


图 8-7　执行 jar 命令的输出信息

此时会在硬盘上生成一个 my.jar 的压缩文件，可以直接使用 WINRAR 打开此文件。如果想使用此文件，则必须设置 classpath，设置命令如下：

```
SET CLASSPATH=.;f:\test\my.jar
```

以上设置了两个 classpath 目录，一个是从当前所在的文件夹中查找，另外一个就是刚压缩好的*.jar 文件。

下面编写测试类，测试此 jar 文件是否可用。测试之前将生成的 Hello.class 文件连同包一起删除掉。

范例：测试程序

```
package org.lxh.demo08;
import org.lxh.demo08.demo.Hello;
public class ImportJarDemo {
    public static void main(String[] args) {
        Hello hello = new Hello() ;
        System.out.println(hello.getInfo()) ;
    }
}
```

程序运行结果：

```
Hello World!!!
```

如果现在想查看一个 jar 文件中的全部内容，可以直接输入“jar-tvf jar 文件的名称”，如图 8-8 所示。

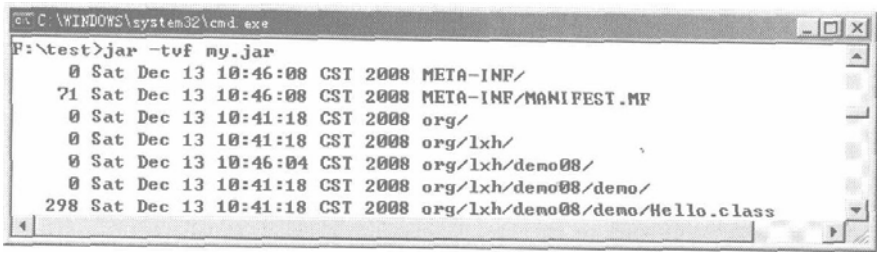


图 8-8　查看 jar 文件的全部内容

解压缩之后可以发现在一个 jar 包中存在一个 META-INF 文件夹，在此文件夹中存在一个 MANIFEST.MF 文件，此文件就是 jar 文件的清单文件。

如果需要把一个 jar 文件解压缩，则直接输入“jar-xf jar 文件名称”即可。

8.3　访问控制权限

有了包的概念之后，就可以为读者详细介绍 Java 中访问权限的概念，在 Java 中一共存在 4 种访问控制权限，即 private、default（默认）、protected 和 public。

1．private 访问权限

private 属于私有访问权限，前面已经介绍过 private 访问权限，可以用在属性的定义、方法的声明上，一旦使用了 private 关键字声明，则只能在本类中进行访问。

2．default（默认）访问权限

如果一个类中的属性或方法没有使用任何的访问权限声明，则就是默认的访问权限，默认的访问权限可以被本包中的其他类所访问，但是不能被其他包的类所访问。

3．protected 访问权限

protected 属于受保护的访问权限。一个类中的成员如果使用了 protected 访问权限，则只能被本包及不同包的子类访问。

4．public 访问权限

public 属于公共访问权限。如果一个类中的成员使用了 public 访问权限，就可以在所有类中被访问，不管是否在同一个包中。

最后使用表 8-2 来总结上述的访问控制权限。

表 8-2 访问控制权限

范　围	private	default	protected	public
同一类	✓	✓	✓	✓
同一包中的类		✓	✓	✓
不同包的子类			✓	✓
其他包中的类				✓

对于 private、default、public 3 种控制权限，前面已经为读者介绍过了，下面讲解 protected 访问权限的作用。

范例： protected 访问权限

在 org.lxh.demo08.g 包中定义一个 HelloDemo 类，其中包含一个 protected 访问权限。

```
package org.lxh.demo08.g;
public class HelloDemo {
    protected String name = "LXH" ;                    // 只能被本包及不同包的子类访问
}
```

在 org.lxh.demo08.h 包中的子类访问此类中的 name 属性。

```
package org.lxh.demo08.h;
import org.lxh.demo08.g.HelloDemo;
class SubHelloDemo extends HelloDemo{                  // 定义HelloDemo子类
    public void print(){
        System.out.println("访问受保护属性：" + super.name) ;// 可以访问protected
                                                              权限
    }
}
public class ProtectedDemo01 {
    public static void main(String[] args) {
        SubHelloDemo sub = new SubHelloDemo() ;// 实例化子类对象
        sub.print() ;
    }
}
```

程序运行结果：

```
访问受保护属性：LXH
```

以上程序中在不同包的子类里访问了 protected 属性，而如果现在由不同包的类直接访问 HelloDemo 类中的 protected 属性，则会出现下面的编译错误。

范例： 在其他包的类中访问 protected 属性

```
package org.lxh.demo08.h;
import org.lxh.demo08.g.HelloDemo;
public class ProtectedDemo02 {
    public static void main(String[] args) {
```

```
        HelloDemo sub = new HelloDemo() ;
        System.out.println(sub.name)  ;          // 错误，不同包的类无法访问
                                                    protected属性
    }
}
```

编译时出现以下错误：

```
ProtectedDemo02.java:7: name has protected access in
org.lxh.demo08.g.HelloDemo
            System.out.println(sub.name) ;       // 错误，不同包的类无法访问
                                                    protected属性
                                    ^
1 error
```

8.4　Java 命名规范

读者通过本书或者 JDK 的文档可以发现，声明类、方法、属性等都是有一定的规范的，此规范如下。

（1）类：所有单词的首字母大写，如 TestJava。

（2）方法：第 1 个单词的首字母小写，之后每个单词的首字母大写，如 getInfo()。

（3）属性：第 1 个单词的首字母小写，之后每个单词的首字母大写，如 studentName。

（4）包：所有单词的字母小写，如 org.lxh.demo。

（5）常量：所有单词的字母大写，如 FLAG。

8.5　本 章 要 点

1．Java 中使用包可以实现多人协作的开发模式。应避免类名称重复的麻烦。

2．在 Java 中使用 package 关键字来将一个类放入一个包中。

3．在 Java 中使用 import 语句可以导入一个已有的包。

4．如果在一个程序中导入了不同包的同名类，在使用时一定要明确地写出“包.类名称”。

5．Java 中的访问控制权限分为 4 种，即 private、default、protected、public。

6．使用 jar 命令可以将一个包打成一个 jar 文件，以供用户使用。

8.6　习　　题

将第 6 章的宠物商店代码中各个不同功能的类放在不同的包中定义，一定要严格遵循命名规范的要求。

回顾：

当产生了一个类之后，为了保证类中的内容不被外部直接看到，则使用 private 关键字

但是，如果现在两个有关系的类要进行属性互相访问的话就比较麻烦，之前只能使用 getter/setter 取得和设置，所以为了减少私有属性的访问的麻烦，使用了内部类，但是内部类本身会破坏程序的结构。

为了让一个类的内容可以继续方便的时候，使用了继承的概念，但是在继承中 private 属性也是无法被子类看到的，所以此时，为了方便子类的操作，可以将属性使用 protected 进行封装，这样一来外部也无法直接看到（不同包）。

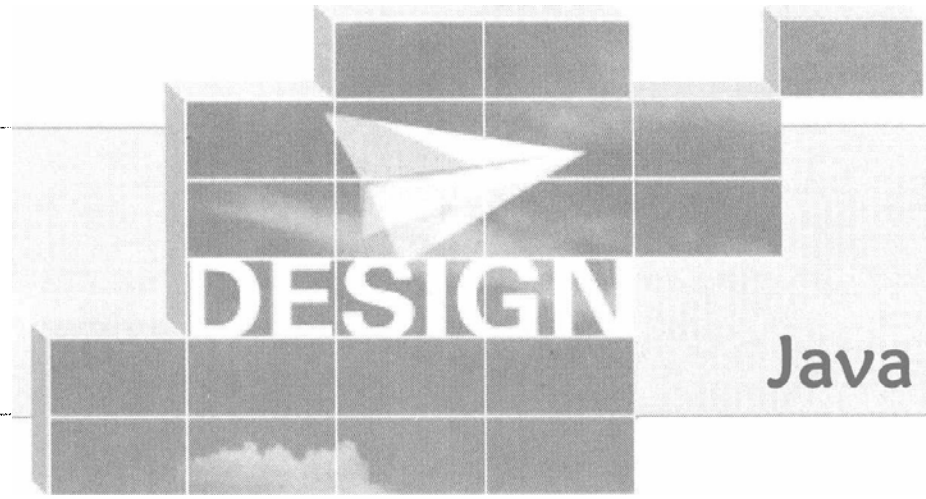
之后有了继承之后，既然有了父子的关系，所以就可以使用向上或向下的转型操作，以完成多态性，但是在开发中类与类之间的直接继承并不多见，而往往继承抽象类或实现接口，当若干个操作间需要解耦合的时候就可以使用接口完成。

既然有内部类，则如果一个接口或抽象类的子类只使用一次，则可以将其定义成匿名内部类。

开发中没有包的类是绝对不存在的。

封装 → 继承 → 多态。

第 3 部分



Java 应用程序设计

- 多线程
- Java 常用类库
- IO 操作
- Java 类集框架
- Java 图形界面
- Java 数据库编程
- Java 新特性：泛型、枚举、Annotation
- Java 反射机制
- Java 网络程序设计
- Java 新 IO

第9章 多 线 程

通过本章的学习可以达到以下目标：

- ☑ 了解进程与线程的区别。
- ☑ 掌握 Java 多线程的两种实现方式及区别。
- ☑ 了解线程的状态变化。
- ☑ 了解多线程的主要操作方法。
- ☑ 了解同步及死锁的概念。
- ☑ 了解线程的生命周期。

Java 是少数的几种支持多线程的语言之一。大多数的程序语言只能运行单独一个程序块，无法同时运行不同的多个程序块。Java 的多线程可以弥补这个缺憾，它可以让不同的程序块一起运行，这样可让程序运行更为顺畅，同时也可达到多任务处理的目的。**本章视频录像讲解时间为 2 小时 36 分钟，源代码在光盘对应的章节下。**

9.1 进程与线程

进程是程序的一次动态执行过程，它经历了从代码加载、执行到执行完毕的一个完整过程，这个过程也是进程本身从产生、发展到最终消亡的过程。多进程操作系统能同时运行多个进程（程序），由于 CPU 具备分时机制，所以每个进程都能循环获得自己的 CPU 时间片。由于 CPU 执行速度非常快，使得所有程序好像是在“同时”运行一样。

多线程是实现并发机制的一种有效手段。进程和线程一样，都是实现并发的一个基本单位。线程是比进程更小的执行单位，线程是在进程的基础之上进行的进一步划分。所谓多线程是指一个进程在执行过程中可以产生多个线程，这些线程可以同时存在、同时运行，一个进程可能包含了多个同时执行的线程。进程与线程的区别如图 9-1 所示。

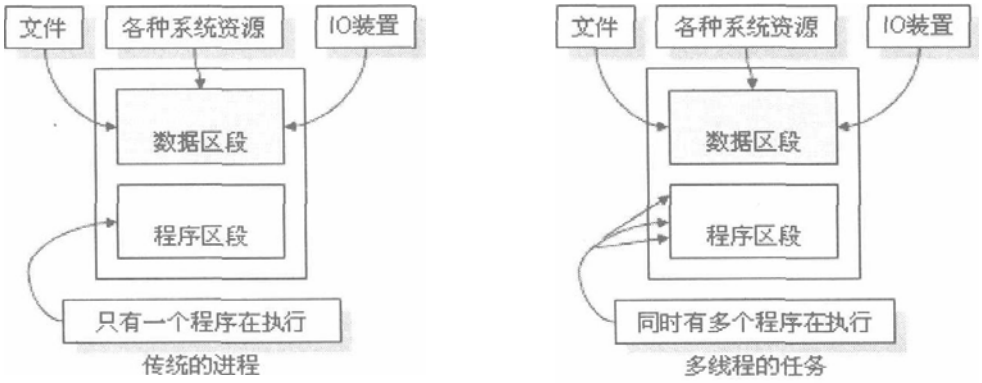


图 9-1 进程与线程的区别

提示：通过 Word 的使用了解进程与线程的区别。

读者应该都有过使用 Word 的经验，在 Word 中如果出现了单词的拼写错误，则 Word 会在出错的单词下划出红线。那么实际上每次启动一个 Word 对于操作系统而言就相当于启动了一个系统的进程，而在这个进程之上又有许多其他程序在运行（如拼写检查），那么这些程序就是一个个的线程。如果 Word 关闭了，则这些拼写检查的线程也将会消失，但是此时即使拼写检查的线程消失了，也并不一定会让 Word 的进程消失。

在传统的程序语言中，运行的顺序总是必须按照程序的流程运行，遇到 if…else 语句就加以判断，遇到 for、while 等循环就会多绕几个圈，最后程序还是按着一定的程序运行，且一次只能运行一个程序块。

Java 的多线程打破了这种传统的束缚。所谓的线程（Thread）是指程序的运行流程，多线程机制则是指可以同时运行多个程序块，使程序运行的效率变得更高，也可克服传统程序语言所无法解决的问题。例如，有些包含循环的线程可能要使用一段时间来运算，此时便可让另一个线程来做其他的处理。

9.2　Java 中线程的实现

在 Java 中要想实现多线程代码有两种手段，一种是继承 Thread 类，另一种就是实现 Runnable 接口。下面分别来介绍这两种方式的使用。

9.2.1　继承 Thread 类

Thread 类是在 java.lang 包中定义的，一个类只要继承了 Thread 类，此类就称为多线程操作类。在 Thread 子类中，必须明确地覆写 Thread 类中的 run()方法，此方法为线程的主体。线程类的定义如下。

【格式 9-1　多线程的定义语法】

```
class 类名称 extends Thread{ // 继承Thread类
    属性… ;                  // 类中定义属性
    方法… ;                  // 类中定义方法
    public void run(){       // 覆写Thread类中的run()方法，此方法是线程的主体
        线程主体;
    }
}
```

下面使用以上格式进行多线程的实现。

范例：继承 Thread 类实现多线程

```
package org.lxh.demo09.threaddemo;
class MyThread extends Thread {                    // 继承Thread类
    private String name;                           // 在类中定义一个属性
```

```
    public MyThread(String name) {                    // 通过构造方法设置属性内容
        this.name = name;                             // 为name属性赋值
    }
    public void run() {                               // 覆写Thread类中的run()方法
        for (int i = 0; i < 10; i++) {                // 循环10次输出
            System.out.println(name + "运行, i = " + i);
        }
    }
};
public class ThreadDemo01 {
    public static void main(String args[]) {
        MyThread mt1 = new MyThread("线程A ");  // 实例化对象
        MyThread mt2 = new MyThread("线程B ");  // 实例化对象
        mt1.run();                                    // 调用线程主体
        mt2.run();                                    // 调用线程主体
    }
};
```

程序运行结果：

```
线程A 运行, i = 0
线程A 运行, i = 1
线程A 运行, i = 2
线程A 运行, i = 3
线程A 运行, i = 4
线程A 运行, i = 5
线程A 运行, i = 6
线程A 运行, i = 7
线程A 运行, i = 8
线程A 运行, i = 9
线程B 运行, i = 0
线程B 运行, i = 1
线程B 运行, i = 2
线程B 运行, i = 3
线程B 运行, i = 4
线程B 运行, i = 5
线程B 运行, i = 6
线程B 运行, i = 7
线程B 运行, i = 8
线程B 运行, i = 9
```

编译并运行程序之后，发现以上的程序是先执行完 mt1 对象之后再执行 mt2 对象，并没有像之前解释的那样是交错运行的，也就是说，此时线程实际上并没有被启动，还是属于顺序式的执行方式，那么该如何启动线程呢？如果要正确地启动线程，是不能直接调用

run()方法的，而应该是调用从 Thread 类中继承而来的 start()方法，具体代码如下。

范例：启动线程

```
class MyThread extends Thread {                    // 继承Thread类
    private String name;                           // 在类中定义一个属性
    public MyThread(String name) {                 // 通过构造方法设置属性内容
        this.name = name;                          // 为name属性赋值
    }
    public void run() {                            // 覆写Thread类中的run()方法
        for (int i = 0; i < 10; i++) {             // 循环输出10次
            System.out.println(name + "运行, i = " + i);
        }
    }
};
public class ThreadDemo02 {
    public static void main(String args[]) {
        MyThread mt1 = new MyThread("线程A ");     // 实例化对象
        MyThread mt2 = new MyThread("线程B ");     // 实例化对象
        mt1.start();                               // 启动多线程
        mt2.start();                               // 启动多线程
    }
};
```

程序运行结果（可能的一种结果）：

```
线程A 运行, i = 0
线程B 运行, i = 0
线程B 运行, i = 1
线程A 运行, i = 1
线程A 运行, i = 2
线程B 运行, i = 2
线程B 运行, i = 3
线程A 运行, i = 3
线程B 运行, i = 4
线程A 运行, i = 4
线程A 运行, i = 5
线程B 运行, i = 5
线程B 运行, i = 6
线程A 运行, i = 6
线程A 运行, i = 7
线程B 运行, i = 7
线程A 运行, i = 8
线程B 运行, i = 8
线程A 运行, i = 9
线程B 运行, i = 9
```

从程序的运行结果中可以发现，两个线程现在是交错运行的，哪个线程对象抢到了 CPU 资源，哪个线程就可以运行，所以程序每次的运行结果肯定是不一样的，在线程启动时虽然调用的是 start()方法，但实际上调用的却是 run()方法的主体。

提问：为什么启动线程不能直接使用 run()方法？

在启动多线程时为什么必须通过 start()方法启动，而不能直接调用 run()方法呢？

回答：线程的运行需要本机操作系统支持。

首先来看一下 start()方法在 Thread 类中的定义。

代码： start()方法部分定义

```
public synchronized void start() {
    if (threadStatus != 0)
        throw new IllegalThreadStateException();
    ...
    start0();
    ...
}
private native void start0();
```

从以上代码中可以发现，在一个类中的 start()方法调用时可能会抛出“IllegalThreadStateException”异常，一般在重复调用 start()方法时会抛出这个异常。而且实际上此处真正调用的是 start0()方法，此方法在声明处使用了 native 关键字声明，此关键字表示调用本机的操作系统函数，因为多线程的实现需要依靠底层操作系统支持。

如果一个类通过继承 Thread 类来实现，那么只能调用一次 start()方法，如果调用多次，则将会抛出“IllegalThreadStateException”异常。

范例： 重复调用 start()方法

```
class MyThread extends Thread {                    // 继承Thread类
    private String name;                           // 在类中定义一个属性
    public MyThread(String name) {                 // 通过构造方法设置属性内容
        this.name = name;                          // 为name属性赋值
    }
    public void run() {                            // 覆写Thread类中的run()方法
        for (int i = 0; i < 10; i++) {             // 循环输出10次
            System.out.println(name + "运行, i = " + i);
        }
    }
};
public class ThreadDemo03 {
    public static void main(String args[]) {
        MyThread mt1 = new MyThread("线程A ");     // 实例化对象
        mt1.start();                               // 启动多线程
```

```
            mt1.start();                                    // 错误，第2次调用start()方法
        }
    };
```

程序运行结果：

```
线程A 运行，i = 0
线程A 运行，i = 1
Exception in thread "main" java.lang.IllegalThreadStateException
    at java.lang.Thread.start(Unknown Source)
    at org.lxh.demo09.threaddemo.ThreadDemo03.main(ThreadDemo03.java:26)
```

以上的错误本身并不难理解，只要参照 Thread 类的定义即可。当然，如果一个类只能继承 Thread 类才能实现多线程，则必定会受到单继承局限的影响。所以一般来说，要想实现多线程还可以通过实现 Runnable 接口完成。

9.2.2　实现 Runnable 接口

在 Java 中也可以通过实现 Runnable 接口的方式实现多线程，Runnable 接口中只定义了一个抽象方法：

```
public void run() ;
```

使用 Runnable 接口实现多线程的格式如下。

【格式 9-2　通过 Runnable 接口实现多线程】

```
class 类名称 implements Runnable{      // 实现Runnable接口
    属性… ;                            // 类中定义属性
    方法… ;                            // 类中定义方法
    public void run(){                 // 覆写Runnable接口中的run()方法
        线程主体 ;
    }
}
```

下面就使用此格式进行多线程的实现。

范例：实现 Runnable 接口

```
class MyThread implements Runnable {          // 实现Runnable接口
    private String name;                      // 在类中定义一个属性
    public MyThread(String name) {            // 通过构造方法设置属性内容
        this.name = name;
    }
    public void run() {                       // 覆写Runnable接口中的run()方法
        for (int i = 0; i < 10; i++) {
            System.out.println(name + "运行，i = " + i);
        }
    }
};
```

以上代码通过实现 Runnable 接口实现多线程，但是这样一来就会有新的问题产生，从之前代码中可以清楚地知道，要想启动一个多线程必须要使用 start()方法完成，如果继承了 Thread 类，则可以直接从 Thread 类中使用 start()方法，但是现在实现的是 Runnable 接口，那么该如何启动多线程呢？实际上，此时还是要依靠 Thread 类完成启动，在 Thread 类中提供了 public Thread(Runnable target)和 public Thread(Runnable target,String name)两个构造方法。

这两个构造方法都可以接收 Runnable 的子类实例对象，所以就可以依靠此点启动多线程，具体代码如下。

范例：使用 Thread 类启动多线程

```
class MyThread implements Runnable {                    // 实现Runnable接口
    private String name;                                 // 在类中定义一个属性
    public MyThread(String name) {                       // 通过构造方法设置属性内容
        this.name = name;                                // 为name属性赋值
    }
    public void run() {                                  // 覆写Runnable接口中的run()
                                                            方法
        for (int i = 0; i < 10; i++) {                   // 循环输出10次
            System.out.println(name + "运行，i = " + i);
        }
    }
};
public class RunnableDemo01 {
    public static void main(String args[]) {
        MyThread my1 = new MyThread("线程A ");           // 实例化Runnable子类对象
        MyThread my2 = new MyThread("线程B ");           // 实例化Runnable子类对象
        Thread t1 = new Thread(my1) ;                    // 实例化Thread类对象
        Thread t2 = new Thread(my2) ;                    // 实例化Thread类对象
        t1.start() ;                                     // 启动线程
        t2.start() ;                                     // 启动线程
    }
};
```

程序运行结果：

```
线程A 运行，i = 0
线程B 运行，i = 0
线程A 运行，i = 1
线程B 运行，i = 1
线程A 运行，i = 2
线程B 运行，i = 2
线程A 运行，i = 3
线程B 运行，i = 3
```

```
线程A 运行，i = 4
线程B 运行，i = 4
线程A 运行，i = 5
线程B 运行，i = 5
线程A 运行，i = 6
线程B 运行，i = 6
线程A 运行，i = 7
线程B 运行，i = 7
线程A 运行，i = 8
线程B 运行，i = 8
线程A 运行，i = 9
线程B 运行，i = 9
```

从以上两种实现可以发现，无论使用哪种方式，最终都必须依靠 Thread 类才能启动多线程。

9.2.3　Thread 类和 Runnable 接口

通过 Thread 类和 Runnable 接口都可以实现多线程，那么两者有哪些联系和区别呢？下面观察 Thread 类的定义。

```
public class Thread extends Object implements Runnable
```

从 Thread 类的定义可以清楚地发现，Thread 类也是 Runnable 接口的子类，但在 Thread 类中并没有完全地实现 Runnable 接口中的 run()方法，下面是 Thread 类的部分定义。

代码： Thread 类的部分定义

```
private Runnable target;
public Thread(Runnable target, String name) {
    init(null, target, name, 0);
}
private void init(ThreadGroup g, Runnable target, String name,long stackSize){
    ...
    this.target = target;
    ...
}
public void run() {
    if (target != null) {
        target.run();
    }
}
```

从定义中可以发现，在 Thread 类中的 run()方法调用的是 Runnable 接口中的 run()方法，也就是说此方法是由 Runnable 子类完成的，所以如果要通过继承 Thread 类实现多线程，则必须覆写 run()方法。

讲到这里，会有部分读者觉得以上的代码操作形式好像比较熟悉，Thread 和 Runnable 的子类都同时实现了 Runnable 接口，之后将 Runnable 的子类实例放到了 Thread 类之中，如图 9-2 所示，这种操作模式和讲解过的代理设计类似。

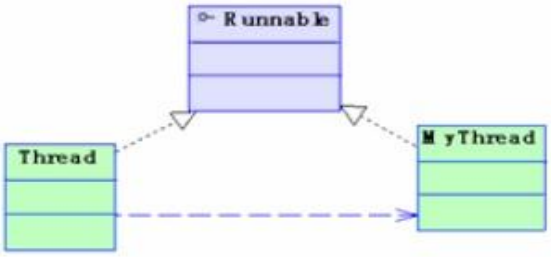


图 9-2　Thread 类、Runnable 接口、Runnable 接口实现类

实际上 Thread 类和 Runnable 接口之间在使用上也是有区别的，如果一个类继承 Thread 类，则不适合于多个线程共享资源，而实现了 Runnable 接口，就可以方便地实现资源的共享。

范例：继承 Thread 类不能资源共享

```
class MyThread extends Thread{                          // 继承Thread类
    private int ticket = 5 ;                            // 一共5张票
    public void run(){                                  // 覆写run()方法
        for(int i=0;i<100;i++){                         // 超出票数的循环
            if(ticket>0){                               // 判断是否有剩余票
                System.out.println("卖票：ticket = " + ticket--) ;
            }
        }
    }
};
public class ThreadDemo04 {
    public static void main(String args[]) {
        MyThread mt1 = new MyThread() ;                 // 定义线程对象
        MyThread mt2 = new MyThread() ;                 // 定义线程对象
        MyThread mt3 = new MyThread() ;                 // 定义线程对象
        mt1.start() ;                                   // 启动第1个线程
        mt2.start() ;                                   // 启动第2个线程
        mt3.start() ;                                   // 启动第3个线程
    }
};
```

程序运行结果：

```
卖票：ticket = 5
卖票：ticket = 4
卖票：ticket = 3
卖票：ticket = 2
卖票：ticket = 1
卖票：ticket = 5
```

```
卖票：ticket = 4
卖票：ticket = 3
卖票：ticket = 2
卖票：ticket = 1
卖票：ticket = 5
卖票：ticket = 4
卖票：ticket = 3
卖票：ticket = 2
卖票：ticket = 1
```

以上程序通过 Thread 类实现多线程，程序中启动了 3 个线程，但是 3 个线程却分别卖了各自的 5 张票，并没有达到资源共享的目的。

范例：实现 Runnable 接口可以资源共享

```
class MyThread implements Runnable{                          // 实现Runnable接口
    private int ticket = 5 ;                                 // 一共5张票
    public void run(){                                       // 覆写run()方法
        for(int i=0;i<100;i++){                              // 超出票数的循环
            if(ticket>0){                                    // 判断是否有剩余票
                System.out.println("卖票：ticket = " + ticket--) ;
            }
        }
    }
};
public class RunnableDemo02 {
    public static void main(String args[]) {
        MyThread my = new MyThread() ;
        new Thread(my).start() ;                             // 启动3个线程
        new Thread(my).start() ;                             // 启动3个线程
        new Thread(my).start() ;                             // 启动3个线程
    }
};
```

程序运行结果：

```
卖票：ticket = 5
卖票：ticket = 4
卖票：ticket = 3
卖票：ticket = 2
卖票：ticket = 1
```

从程序的运行结果中可以清楚地发现，虽然启动了 3 个线程，但是 3 个线程一共才卖了 5 张票，即 ticket 属性被所有的线程对象共享。

可见，实现 Runnable 接口相对于继承 Thread 类来说，有如下显著的优势：

（1）适合多个相同程序代码的线程去处理同一资源的情况。

（2）可以避免由于 Java 的单继承特性带来的局限。

（3）增强了程序的健壮性，代码能够被多个线程共享，代码与数据是独立的。

所以，在开发中建议读者使用 Runnable 接口实现多线程，本书后面的部分也都采用 Runnable 接口的方式实现多线程操作。

9.3 线程的状态

要想实现多线程，必须在主线程中创建新的线程对象。任何线程一般具有 5 种状态，即创建、就绪、运行、阻塞、终止。线程状态的转移与方法之间的关系可用图 9-3 来表示。

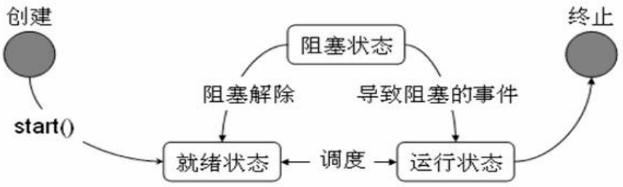


图 9-3 线程转换状态

1. 创建状态

在程序中用构造方法创建了一个线程对象后，新的线程对象便处于新建状态，此时，它已经有了相应的内存空间和其他资源，但还处于不可运行状态。新建一个线程对象可采用 Thread 类的构造方法来实现，例如“Thread thread=new Thread();”。

2. 就绪状态

新建线程对象后，调用该线程的 start()方法就可以启动线程。当线程启动时，线程进入就绪状态。此时，线程将进入线程队列排队，等待 CPU 服务，这表明它已经具备了运行条件。

3. 运行状态

当就绪状态的线程被调用并获得处理器资源时，线程就进入了运行状态。此时，自动调用该线程对象的 run()方法。run()方法定义了该线程的操作和功能。

4. 堵塞状态

一个正在执行的线程在某些特殊情况下，如被人为挂起或需要执行耗时的输入/输出操作时，将让出 CPU 并暂时中止自己的执行，进入堵塞状态。在可执行状态下，如果调用 sleep()、suspend()、wait()等方法，线程都将进入堵塞状态。堵塞时，线程不能进入排队队列，只有当引起堵塞的原因被消除后，线程才可以转入就绪状态。

5. 死亡状态

线程调用 stop()方法时或 run()方法执行结束后，即处于死亡状态。处于死亡状态的线程不具有继续运行的能力。

9.4　线程操作的相关方法

从前面的讲解中可以发现，在 Java 实现多线程的程序中，虽然 Thread 类实现了 Runnable 接口，但是操作线程的主要方法并不在 Runnable 接口中，而是在 Thread 类中，表 9-1 列出了 Thread 类中的主要方法。

表 9-1　Thread 类中的主要方法

序号	方 法 名 称	类型	描　　述
1	public Thread(Runnable target)	构造	接收 Runnable 接口子类对象，实例化 Thread 对象
2	public Thread(Runnable target,String name)	构造	接收 Runnable 接口子类对象，实例化 Thread 对象，并设置线程名称
3	public Thread(String name)	构造	实例化 Thread 对象，并设置线程名称
4	public static Thread currentThread()	普通	返回目前正在执行的线程
5	public final String getName()	普通	返回线程的名称
6	public final int getPriority()	普通	返回线程的优先级
7	public boolean isInterrupted()	普通	判断目前线程是否被中断，如果是，返回 true，否则返回 false
8	public final boolean isAlive()	普通	判断线程是否在活动，如果是，返回 true，否则返回 false
9	public final void join() throws Interrupted Exception	普通	等待线程死亡
10	public final synchronized void join(long millis) throws InterruptedException	普通	等待 millis 毫秒后，线程死亡
11	public void run()	普通	执行线程
12	public final void setName(String name)	普通	设定线程名称
13	public final void setPriority(int newPriority)	普通	设定线程的优先值
14	public static void sleep(long millis) throws InterruptedException	普通	使目前正在执行的线程休眠 millis 毫秒
15	public void start()	普通	开始执行线程
16	public String toString()	普通	返回代表线程的字符串
17	public static void yield()	普通	将目前正在执行的线程暂停，允许其他线程执行
18	public final void setDaemon(boolean on)	普通	将一个线程设置成后台运行

下面为读者介绍一些常用的线程操作方法。

9.4.1　取得和设置线程名称

在 Thread 类中可以通过 getName()方法取得线程的名称，还可以通过 setName()方法设置线程的名称。

线程的名称一般在启动线程前设置，但也允许为已经运行的线程设置名称。允许两个 Thread 对象有相同的名称，但应该尽量避免这种情况的发生。

提示：如果没有设置名称，系统会为其自动分配名称。

在线程操作中，如果没有为一个线程指定一个名称，则系统在使用时会为线程分配一个名称，名称的格式为 Thread-Xx。

范例：取得和设置线程的名称

```
class MyThread implements Runnable{                    // 实现Runnable接口
    public void run(){                                 // 覆写接口中的run()方法
        for(int i=0;i<3;i++){                          // 循环输出3次
            System.out.println(Thread.currentThread().getName()
                    + "运行， i = " + i) ;             // 取得当前线程的名称
        }
    }
};
public class ThreadNameDemo {
    public static void main(String args[]) {
        MyThread my = new MyThread() ;                 // 定义Runnable子类对象
        new Thread(my).start() ;                       // 系统自动设置线程名称
        new Thread(my,"线程-A").start() ;              // 手工设置线程名称
        new Thread(my,"线程-B").start() ;              // 手工设置线程名称
        new Thread(my).start() ;                       // 系统自动设置线程名称
        new Thread(my).start() ;                       // 系统自动设置线程名称
    }
};
```

程序运行结果：

```
Thread-0运行， i = 0
线程-A运行， i = 0
线程-B运行， i = 0
Thread-1运行， i = 0
Thread-2运行， i = 0
Thread-2运行， i = 1
Thread-1运行， i = 1
线程-B运行， i = 1
线程-A运行， i = 1
Thread-0运行， i = 1
Thread-0运行， i = 2
线程-A运行， i = 2
线程-B运行， i = 2
Thread-1运行， i = 2
Thread-2运行， i = 2
```

从程序的运行结果中可以发现，没有设置线程名称的 3 个线程对象的名称都是很有规

律的，分别为 Thread-0、Thread-1 和 Thread-2，从之前讲解的 static 关键字可以知道，在 Thread 类中必然存在一个 static 类型的属性，用于为线程自动命名。

了解以上代码之后，下面再来观察以下的代码。

范例：观察程序的输出

```
class MyThread implements Runnable{                    // 实现Runnable接口
    public void run(){                                 // 覆写接口中的run()方法
        for(int i=0;i<3;i++){                          // 循环输出3次
            System.out.println(Thread.currentThread().getName()
                    + "运行，  i = " + i) ;            // 取得当前线程的名称
        }
    }
};
public class CurrentThreadDemo {
    public static void main(String args[]) {
        MyThread my = new MyThread() ;                 // 定义Runnable子类对象
        new Thread(my,"线程").start() ;                // 启动线程
        my.run() ;                                     // 直接调用run方法
    }
};
```

程序运行结果：

```
main运行，  i = 0
线程运行，  i = 0
线程运行，  i = 1
main运行，  i = 1
线程运行，  i = 2
main运行，  i = 2
```

在以上程序中，主方法直接通过 Runnable 接口的子类对象调用其中的 run()方法，另外一个是通过线程对象调用 start()方法启动的，从结果中发现，主方法实际上也是一个线程。另外要提醒读者的是，在 Java 中所有的线程都是同时启动的，哪个线程先抢占到了 CPU 资源，哪个就先运行。

提问：Java 程序每次运行至少启动几个线程？

因为 Java 是多线程的编程语言，所以 Java 程序运行时也是以线程的方式运行的，那么主方法也就是一个线程（main），但是对于一个 Java 程序来说，一个 Java 程序运行至少会启动几个线程？

回答：至少启动两个线程。

从之前学习的知识中可以知道，每当使用 Java 命令执行一个类时，实际上都会启动一个 JVM，每一个 JVM 实际上就是在操作系统中启动了一个进程，Java 本身具备了垃圾的收集机制。所以在 Java 运行时至少会启动两个线程，一个是 main 线程，另外一个是垃圾收集线程。

9.4.2 判断线程是否启动

通过前面的讲解可知，通过 Thread 类中的 start()方法通知 CPU 这个线程已经准备好启动，然后就等待分配 CPU 资源，运行此线程。在 Java 中可以使用 isAlive()方法来测试线程是否已经启动而且仍然在启动。

范例：判断线程是否启动

```
class MyThread implements Runnable {                          // 实现Runnable接口
    public void run() {                                       // 覆写run()方法
        for (int i = 0; i < 3; i++) {                         // 循环输出3次
            System.out.println(Thread.currentThread().getName()
                               + "运行 --> " + i);            // 取得当前线程名称
        }
    }
};
public class ThreadAliveDemo {
    public static void main(String args[]) {
        MyThread mt = new MyThread();                         // 实例化对象
        Thread t = new Thread(mt, "线程");                    // 实例化Thread对象
        System.out.println("线程开始执行之前 --> " + t.isAlive());// 判断是否
                                                                  启动
        t.start();                                            // 启动线程
        System.out.println("线程开始执行之后 --> " + t.isAlive());// 判断是否
                                                                  启动
        for (int i = 0; i < 3; i++) {                         // 循环输出3次
            System.out.println(" main 运行 --> " + i);        // 输出
        }
        System.out.println("代码执行之后 --> " + t.isAlive());// 后面的输出结
                                                                  果不确定
    }
};
```

程序运行结果：

```
线程开始执行之前 --> false
线程开始执行之后 --> true
 main 运行 --> 0
 main 运行 --> 1
 main 运行 --> 2
代码执行之后 --> true
线程运行 --> 0
线程运行 --> 1
线程运行 --> 2
```

以上输出结果是不确定的，有可能到最后线程已经不存活了，但也有可能继续存活，这就要看哪个线程先执行完。

注意：主线程有可能比其他线程先执行完。

因为线程操作的不确定性，所以主线程有可能最先执行完，那么此时其他线程不会受到任何影响，并不会随着主线程的结束而结束。

9.4.3　线程的强制运行

在线程操作中，可以使用 join()方法让一个线程强制运行，线程强制运行期间，其他线程无法运行，必须等待此线程完成之后才可以继续执行。

范例：线程的强制运行

```
class MyThread implements Runnable {                    // 实现Runnable接口
    public void run() {                                 // 覆写run()方法
        for (int i = 0; i < 50; i++) {                  // 循环50次
            System.out.
                println(Thread.currentThread().getName()
                        + "运行 --> " + i);             // 输出线程名称
        }
    }
};
public class ThreadJoinDemo {
    public static void main(String args[]) {
        MyThread mt = new MyThread();                   // 实例化对象
        Thread t = new Thread(mt, "线程");              // 实例化Thread对象
        t.start();                                      // 线程启动
        for (int i = 0; i < 50; i++) {                  // 循环50次
            if (i > 10) {                               // 判断变量内容
                try {
                    t.join();                           // 线程t进行强制运行
                } catch (Exception e) {}                // 需要进行异常处理
            }
            System.out.println("Main 线程运行 --> " + i);
        }
    }
};
```

程序运行的部分结果：

```
Main 线程运行 --> 8
Main 线程运行 --> 9
Main 线程运行 --> 10
线程运行 --> 0
```

```
线程运行 --> 1
线程运行 --> 2
.........
```

因为程序的运行结果过长，所以此处只给出部分的结果，读者自行实验后可以发现，以后所有的输出将都由一个线程完成，主线程必须等待这个线程完成之后才会继续执行。

9.4.4 线程的休眠

在程序中允许一个线程进行暂时的休眠，直接使用 Thread.sleep()方法即可实现休眠。

范例：线程的休眠

```
class MyThread implements Runnable {                    // 实现Runnable接口
    public void run() {                                 // 覆写run()方法
        for (int i = 0; i < 5; i++) {                   // 循环5次
            try {
                Thread.sleep(500);                      // 线程休眠
            } catch (Exception e) {}                    // 需要异常处理
            System.out.println(Thread.currentThread().getName()
                    + "运行,  i = " + i);               // 输出线程名称
        }
    }
};
public class ThreadSleepDemo {
    public static void main(String args[]) {
        MyThread mt = new MyThread();                   // 实例化对象
        new Thread(mt, "线程").start();                 // 启动线程
    }
};
```

程序运行结果：

```
线程运行,  i = 0
线程运行,  i = 1
线程运行,  i = 2
线程运行,  i = 3
线程运行,  i = 4
```

以上程序在执行时，每次的输出都会间隔 500ms，达到了延迟操作的效果。

9.4.5 中断线程

当一个线程运行时，另外一个线程可以直接通过 interrupt()方法中断其运行状态。

范例：线程的中断

```
class MyThread implements Runnable {                    // 实现Runnable接口
    public void run() {                                 // 覆写run()方法
```

```
            System.out.println("1、进入run方法");
            try {
                Thread.sleep(10000);                    // 休眠10s
                System.out.println("2、已经完成休眠");
            } catch (Exception e) {
                System.out.println("3、休眠被终止");
                return;                                 // 让程序返回被调用处
            }
            System.out.println("4、run方法正常结束");
        }
    };
    public class ThreadInterruptDemo {
        public static void main(String args[]) {
            MyThread mt = new MyThread();               // 实例化子类对象
            Thread t = new Thread(mt, "线程");          // 实例化线程对象
            t.start();
            try {
                Thread.sleep(2000);                     // 稍微停2s再继续中断
            } catch (Exception e) {
            }
            t.interrupt();                              // 中断线程执行
        }
    };
```

程序运行结果：

```
1、进入run方法
3、休眠被终止
```

从以上程序可以看出，一个线程启动之后进入了休眠状态，原本是要休眠 10s 之后再继续执行，但是主方法在线程启动之后的两秒后就将其中断，休眠一旦中断之后将执行 catch 中的代码。

9.4.6　后台线程

在 Java 程序中，只要前台有一个线程在运行，则整个 Java 进程都不会消失，所以此时可以设置一个后台线程，这样即使 Java 进程结束了，此后台线程依然会继续执行。要想实现这样的操作，直接使用 setDaemon()方法即可。

范例：后台线程的设置

```
class MyThread implements Runnable {                    // 实现Runnable接口
    public void run() {                                 // 覆写run()方法
        while (true) {                                  // 无限制循环
            System.out.println(Thread.currentThread()
```

```
                    .getName() + "在运行。");     // 输出线程名称
            }
        }
    };
    public class ThreadDaemonDemo {
        public static void main(String args[]) {
            MyThread mt = new MyThread();              // 实例化线程对象
            Thread t = new Thread(mt, "线程");         // 实例化Thread类对象
            t.setDaemon(true) ;                        // 此线程在后台运行
            t.start();                                 // 启动线程
        }
    };
```

在线程类 MyThread 中，尽管 run()方法中是死循环的方式，但是程序依然可以执行完，因为方法中的死循环已经设置成后台运行了。

9.4.7 线程的优先级

在 Java 的线程操作中，所有的线程在运行前都会保持在就绪状态，那么此时，哪个线程的优先级高，哪个线程就有可能会先被执行，如图 9-4 所示。



图 9-4 线程优先级

在 Java 的线程中使用 setPriority()方法可以设置一个线程的优先级，在 Java 的线程中一共有 3 种优先级，如表 9-2 所示。

表 9-2 线程优先级

序 号	定 义	描 述	表示的常量
1	public static final int MIN_PRIORITY	最低优先级	1
2	public static final int NORM_PRIORITY	中等优先级，是线程的默认优先级	5
3	public static final int MAX_PRIORITY	最高优先级	10

下面通过一段代码来演示 3 种不同优先级的线程执行结果。

范例：测试线程优先级

```
class MyThread implements Runnable {                    // 实现Runnable接口
    public void run() {                                 // 覆写run()方法
        for (int i = 0; i < 5; i++) {                   // 循环5次
            try {
```

```
                Thread.sleep(500);                          // 线程休眠
            } catch (Exception e) {}                        // 需要异常处理
            System.out.println(Thread.currentThread().getName()
                    + "运行,  i = " + i);                   // 输出线程名称
        }
    }
};
public class ThreadPriorityDemo {
    public static void main(String[] args) {
        Thread t1 = new Thread(new MyThread(),"线程A");// 实例化线程对象
        Thread t2 = new Thread(new MyThread(),"线程B");// 实例化线程对象
        Thread t3 = new Thread(new MyThread(),"线程C");// 实例化线程对象
        t1.setPriority(Thread.MIN_PRIORITY) ;           // 设置线程优先级为最低
        t2.setPriority(Thread.MAX_PRIORITY) ;           // 设置线程优先级为最高
        t3.setPriority(Thread.NORM_PRIORITY) ;          // 设置线程优先级为中等
        t1.start() ;                                    // 启动线程
        t2.start() ;                                    // 启动线程
        t3.start() ;                                    // 启动线程
    }
}
```

程序运行结果：

```
线程B运行,  i = 0          → 第2个线程（t2）的优先级为：MAX_PRIORITY
线程C运行,  i = 0          → 第3个线程（t3）的优先级为：NORM_PRIORITY
线程A运行,  i = 0          → 第1个线程（t1）的优先级为：MIN_PRIORITY
线程B运行,  i = 1
线程C运行,  i = 1
线程A运行,  i = 1
线程B运行,  i = 2
线程C运行,  i = 2
线程A运行,  i = 2
线程B运行,  i = 3
线程C运行,  i = 3
线程A运行,  i = 3
线程B运行,  i = 4
线程C运行,  i = 4
线程A运行,  i = 4
```

从程序的运行结果中可以观察到，线程将根据其优先级的大小来决定哪个线程会先运行，但是读者一定要注意的是，并非线程的优先级越高就一定会先执行，哪个线程先执行将由 CPU 的调度决定。

提示：主方法的优先级是 NORM_PRIORITY。值为5

在之前曾经讲解过，主方法本身也是一个线程，那么此线程的优先级是什么呢？

范例：取得主方法的优先级

```
public class MainPriorityDemo {
    public static void main(String[] args) {
        System.out.println("主方法的优先级：" +
                Thread.currentThread().getPriority()); // 取得主方法的优先级
    }
}
```

程序运行结果：

```
主方法的优先级：5
```

程序中取得线程优先级的结果是 5，通过表 9-2 可知，数字 5 对应的是“NORM_PRIORITY”优先级，所以主线程的优先级是中等级别。

9.4.8 线程的礼让

在线程操作中，也可以使用 yield()方法将一个线程的操作暂时让给其他线程执行。

范例：线程的礼让

```
class MyThread implements Runnable {                    // 实现Runnable接口
    public void run() {                                 // 覆写run()方法
        for (int i = 0; i < 5; i++) {                   // 不断输出
            System.out.
                println(Thread.currentThread().getName()
                        + "运行 --> " + i);             // 输出线程名称
            if (i == 3) {
                System.out.print("线程礼让：");
                Thread.currentThread().yield() ;        // 线程礼让
            }
        }
    }
};
public class ThreadYieldDemo {
    public static void main(String args[]) {
        MyThread my = new MyThread() ;                  // 实例化MyThread对象
        Thread t1 = new Thread(my, "线程A") ;           // 定义线程对象
        Thread t2 = new Thread(my, "线程B");            // 定义线程对象
        t1.start() ;                                    // 启动多线程
        t2.start() ;                                    // 启动多线程
    }
};
```

程序运行结果：

```
线程A运行 --> 0
线程A运行 --> 1
线程A运行 --> 2
线程A运行 --> 3
线程礼让：线程B运行 --> 0
线程B运行 --> 1
线程B运行 --> 2
线程B运行 --> 3
线程礼让：线程A运行 --> 4
线程B运行 --> 4
```

从程序的运行结果中可以发现，每当线程满足条件（i == 3），就会将本线程暂停，而让其他线程先执行。

9.5　线程操作范例

设计一个线程操作类，可以产生 3 个线程对象，并分别设置 3 个线程的休眠时间，具体如下所示：

- 线程 A，休眠 10 秒。
- 线程 B，休眠 20 秒。
- 线程 C，休眠 30 秒。

从之前的学习应该可以知道，线程的实现有两种方式，一种是继承 Thread 类，另一种是实现 Runnable 接口。而且在类中应该存在保存线程名称和休眠时间两个属性。

9.5.1　实现一——继承 Thread 类

如果直接继承 Thread 类，则可以直接从 Thread 类中将线程名称的属性及操作方法继承下来，所以，此时只需要直接再设置一个休眠的时间属性即可。

范例：使用 Thread 类实现

```
class MyThread extends Thread {
    private int time;                                    // 保存线程的休眠时间
    public MyThread(String name, int time) {
        super(name);                                     // 设置线程名称
        this.time = time;                                // 设置休眠的时间
    }
    public void run() {                                  // 覆写run()方法
        try {
            Thread.sleep(this.time);                     // 指定休眠的时间
        } catch (InterruptedException e) {
```

```
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName() + "线程，休眠"
                + this.time + "毫秒。");                    // 输出信息
    }
}
public class ExecDemo01 {
    public static void main(String[] args) {
        MyThread mt1 = new MyThread("线程A", 10000);    // 定义线程对象，指定
                                                            休眠时间
        MyThread mt2 = new MyThread("线程B", 20000);    // 定义线程对象，指定
                                                            休眠时间
        MyThread mt3 = new MyThread("线程C", 30000);    // 定义线程对象，指定
                                                            休眠时间
        mt1.start();                                    // 启动线程
        mt2.start();                                    // 启动线程
        mt3.start();                                    // 启动线程
    }
}
```

程序运行结果：

```
线程A线程，休眠10000毫秒。
线程B线程，休眠20000毫秒。
线程C线程，休眠30000毫秒。
```

以上程序利用了 Thread 类的特点，所以不用单独设置 name 属性。

9.5.2 实现二——实现 Runnable 接口

如果使用 Runnable 接口实现多线程，则不像继承 Thread 类那样可以直接使用 Thread 类中的 name 属性，需要在类中单独定义一个 name 属性以保存名称。

范例：通过 Runnable 接口实现

```
class MyThread implements Runnable {                    // 实现Runnable接口
    private String name ;                               // 保存线程名称
    private int time;                                   // 保存线程的休眠时间
    public MyThread(String name, int time) {
        this.name = name;                               // 设置线程名称
        this.time = time;                               // 设置休眠的时间
    }
    public void run() {                                 // 覆写run()方法
        try {
            Thread.sleep(this.time);                    // 休眠指定的时间
        } catch (InterruptedException e) {
```

```
            e.printStackTrace();
        }
        System.out.println(this.name + "线程，休眠"
                + this.time + "毫秒。");                    // 输出信息
    }
}
public class ExecDemo02 {
    public static void main(String[] args) {
        MyThread mt1 = new MyThread("线程A", 10000);    // 定义线程对象，指定
                                                           休眠时间
        MyThread mt2 = new MyThread("线程B", 20000);    // 定义线程对象，指定
                                                           休眠时间
        MyThread mt3 = new MyThread("线程C", 30000);    // 定义线程对象，指定
                                                           休眠时间
        new Thread(mt1).start();                        // 启动线程
        new Thread(mt2).start();                        // 启动线程
        new Thread(mt3).start();                        // 启动线程
    }
}
```

程序运行结果：

```
线程A线程，休眠10000毫秒。
线程B线程，休眠20000毫秒。
线程C线程，休眠30000毫秒。
```

9.6 同步与死锁

一个多线程的程序如果是通过 Runnable 接口实现的，则意味着类中的属性将被多个线程共享，那么这样一来就会造成一种问题，如果这多个线程要操作同一资源时就有可能出现资源的同步问题。例如，前面的卖票程序，如果多个线程同时操作时就有可能出现卖出票为负数的问题。

9.6.1 问题的引出

现在通过 Runnable 接口实现多线程，并产生 3 个线程对象，同时卖 5 张票。

范例：观察的问题

```
class MyThread implements Runnable{                     // 实现Runnable接口
    private int ticket = 5 ;                            // 一共5张票
    public void run(){                                  // 覆写run()方法
        for(int i=0;i<100;i++){                         // 超出票数的循环
            if(ticket>0){                               // 判断是否有剩余票
```

```
                try {
                    Thread.sleep(300) ;                    // 加入延迟
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("卖票: ticket = " + ticket--) ;
            }
        }
    }
};
public class SyncDemo01 {
    public static void main(String[] args) {
        MyThread mt = new MyThread() ;                     // 定义线程对象
        Thread t1 = new Thread(mt) ;                       // 定义Thread对象
        Thread t2 = new Thread(mt) ;                       // 定义Thread对象
        Thread t3 = new Thread(mt) ;                       // 定义Thread对象
        t1.start() ;                                       // 启动线程
        t2.start() ;                                       // 启动线程
        t3.start() ;                                       // 启动线程
    }
}
```

程序运行结果：

```
...
卖票: ticket = 1
卖票: ticket = 0
卖票: ticket = -1
```

从程序的运行结果中可以发现，程序中加入了延迟操作，所以在运行的最后出现了负数的情况，那么为什么现在会产生这样的问题呢？

从上面的操作代码中可以发现对于票数的操作步骤如下：

（1）判断票数是否大于 0，大于 0 则表示还有票可以卖

（2）如果票数大于 0，则将票卖出。

但是，在上面的操作代码中，在步骤（1）和步骤（2）之间加入了延迟操作，那么一个线程就有可能在还没有对票数进行减操作之前，其他线程就已经将票数减少了，这样一来就会出现票数为负的情况，如图 9-5 所示。

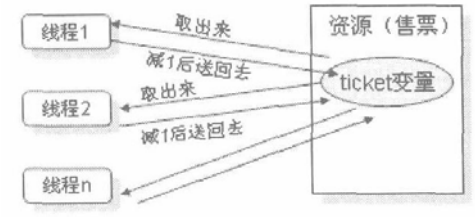
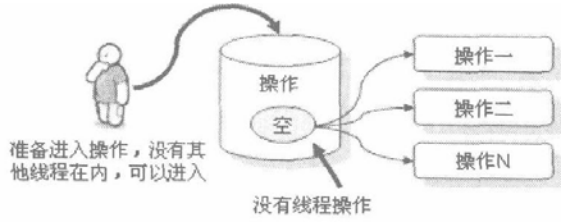
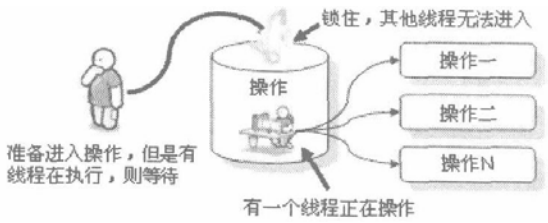


图 9-5　程序操作图

如果想解决这样的问题，就必须使用同步。所谓同步就是指多个操作在同一个时间段内只能有一个线程进行，其他线程要等待此线程完成之后才可以继续执行，如图 9-6 所示。



（a）操作中没有线程对象可以进入



（b）操作中有线程对象被锁住，其他线程不能进入

图 9-6　同步操作

9.6.2　使用同步解决问题

解决资源共享的同步操作，可以使用同步代码块和同步方法两种方式完成。

1．同步代码块

在前几章中曾经为读者介绍过代码块的概念，所谓代码块就是指使用“{}”括起来的一段代码，根据其位置和声明的不同，可以分为普通代码块、构造块、静态块 3 种，如果在代码块上加上 synchronized 关键字，则此代码块就称为同步代码块。同步代码块的格式如下。

【格式 9-3　同步代码块】

```
synchronized(同步对象){
    需要同步的代码 ;
}
```

从格式 9-3 中可以发现，在使用同步代码块时必须指定一个需要同步的对象，但一般都将当前对象（this）设置成同步对象。

范例：使用同步代码块解决的同步问题

```
class MyThread implements Runnable{                    // 实现Runnable接口
    private int ticket = 5 ;                           // 一共5张票
    public void run(){                                 // 覆写run()方法
        for(int i=0;i<100;i++){                        // 超出票数的循环
            synchronized (this) {                      // 设置需要同步的操作
                if(ticket>0){                          // 判断是否有剩余票
                    try {
                        Thread.sleep(300) ;            // 加入延迟
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    System.out.println("卖票：ticket = " + ticket--) ;
                }
```

```
            }
        }
    }
};
public class SyncDemo02 {
    public static void main(String[] args) {
        MyThread mt = new MyThread() ;                    // 定义线程对象
        Thread t1 = new Thread(mt) ;                      // 定义Thread对象
        Thread t2 = new Thread(mt) ;                      // 定义Thread对象
        Thread t3 = new Thread(mt) ;                      // 定义Thread对象
        t1.start() ;                                      // 启动线程
        t2.start() ;                                      // 启动线程
        t3.start() ;                                      // 启动线程
    }
}
```

程序运行结果：

```
卖票：ticket = 5
卖票：ticket = 4
卖票：ticket = 3
卖票：ticket = 2
卖票：ticket = 1
```

从程序运行中可以发现，以上代码将取值和修改值的操作进行了同步，所以不会再出现卖出票为负数的情况。

2. 同步方法

除了可以将需要的代码设置成同步代码块外，也可以使用 synchronized 关键字将一个方法声明成同步方法。

【格式 9-4　同步方法】

```
synchronized 方法返回值 方法名称(参数列表){
}
```

范例：使用同步方法解决以上问题

```
class MyThread implements Runnable{                       // 实现Runnable接口
    private int ticket = 5 ;                              // 一共5张票
    public void run(){                                    // 覆写run()方法
        for(int i=0;i<100;i++){                           // 超出票数的循环
            this.sale();                                  // 调用同步方法
        }
    }
    public synchronized void sale() {                     // 声明同步方法
        if(ticket>0){                                     // 判断是否有剩余票
```

```
                try {
                    Thread.sleep(300) ;                          // 加入延迟
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("卖票: ticket = " + ticket--) ;
            }
        }
    };
    public class SyncDemo03 {
        public static void main(String[] args) {
            MyThread mt = new MyThread() ;                       // 定义线程对象
            Thread t1 = new Thread(mt) ;                         // 定义Thread对象
            Thread t2 = new Thread(mt) ;                         // 定义Thread对象
            Thread t3 = new Thread(mt) ;                         // 定义Thread对象
            t1.start() ;                                         // 启动线程
            t2.start() ;                                         // 启动线程
            t3.start() ;                                         // 启动线程
        }
    }
```

程序运行结果：

```
卖票: ticket = 5
卖票: ticket = 4
卖票: ticket = 3
卖票: ticket = 2
卖票: ticket = 1
```

从程序的运行结果中可以发现，此代码完成了与之前同步代码块同样的功能。

提示：方法定义的完整格式。

学习完 synchronized 关键字后，即可给出 Java 中方法定义的完整格式：

```
访问权限{public|default|protected|private} [final] [static] [synchronized]
返回值类型|void 方法名称(参数类型 参数名称,…) [throws Exception1,Exception2]{
    [return [返回值|返回调用处]] ;
}
```

9.6.3 死锁

同步可以保证资源共享操作的正确性，但是过多同步也会产生问题，如图 9-7 所示。例如，现在张三想要李四的画，李四想要张三的书，张三对李四说了：“把你的画给我，我就给你书”，李四也对张三说了：“把你的书给我，我就给你画”，此时，张三在等着李

四的答复，而李四也在等着张三的答复，那么这样下去最终结果就是，张三得不到李四的画，李四也得不到张三的书，这实际上就是死锁的概念。

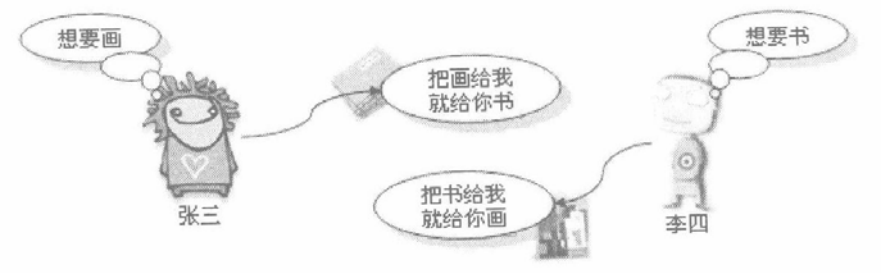


图 9-7　同步产生的问题

所谓死锁就是指两个线程都在等待彼此先完成，造成了程程序的停滞，一般程序的死锁都是在程序运行时出现的。下面通过一个简单的范例来观察出现死锁的情况。

范例：死锁

```
class Zhangsan {                                          // 定义表示张三的类
    public void say() {                                   // 定义say()方法
        System.out.println("张三对李四说："你给我画，我就把书给你"。");
    }
    public void get() {                                   // 定义得到东西的方法
        System.out.println("张三得到画了。");
    }
}
class Lisi {                                              // 定义表示李四的类
    public void say() {                                   // 定义say()方法
        System.out.println("李四对张三说："你给我书，我就把画给你"。");
    }
    public void get() {                                   // 定义得到东西的方法
        System.out.println("李四得到书了。");
    }
}
public class ThreadDeadLock implements Runnable {        // 定义多线程类
    private static Zhangsan zs = new Zhangsan();          // 实例化static型对象，数据共享
    private static Lisi ls = new Lisi();                  // 实例化static型对象，数据共享
    private boolean flag = false;                         // 声明标记，用于判断哪个对象先执行
    public void run() {                                   // 覆写run()方法
        if (flag) {                                       // 判断标志位，Zhangsan先执行
            synchronized (zs) {                           // 通过第1个对象
                zs.say();                                 // 调用方法
```

```
                try {
                    Thread.sleep(500);                    // 加入延迟
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                synchronized (ls) {                       // 同步第2个对象
                    zs.get();                             // 调用方法
                }
            }
        } else {                                          // Lisi先执行
            synchronized (ls) {                           // 同步第2个对象
                ls.say();                                 // 调用方法
                try {
                    Thread.sleep(500);                    // 加入延迟
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                synchronized (zs) {                       // 同步第1个对象
                    ls.get();                             // 调用方法
                }
            }
        }
    }
    public static void main(String[] args) {
        ThreadDeadLock t1 = new ThreadDeadLock();         // 实例化线程对象
        ThreadDeadLock t2 = new ThreadDeadLock();         // 实例化线程对象
        t1.flag = true;                                   // 设置标记
        t2.flag = false;                                  // 设置标记
        Thread thA = new Thread(t1);                      // 实例化Thread类对象
        Thread thB = new Thread(t2);                      // 实例化Thread类对象
        thA.start();                                      // 启动线程
        thB.start();                                      // 启动线程
    }
}
```

程序运行结果：

张三对李四说："你给我画，我就把书给你"。
李四对张三说："你给我书，我就把画给你"。
[以下代码不再执行，程序进入死锁状态]

从程序的运行结果中可以发现，两个线程都在彼此等待着对方的执行完成，这样，程序就无法向下继续执行，从而造成了死锁的现象。

注意：关于同步与死锁。

在这里读者一定要记住，多个线程共享同一资源时需要进行同步，以保证资源操作的完整性，但是过多的同步就有可能产生死锁。

9.7 线程操作案例——生产者及消费者

在线程操作中有一个经典的案例程序，即生产者和消费者问题，生产者不断生产，消费者不断取走生产者生产的产品，如图 9-8 所示。



图 9-8 生产者及消费者问题

在图 9-8 中非常清楚地表示出，生产者生产出信息后将其放到一个区域之中，消费者从此区域中取出数据，但是在本程序中因为牵涉到线程运行的不确定性，所以会存在以下两点问题：

（1）假设生产者线程刚向数据存储空间添加了信息的名称，还没有加入该信息的内容，程序就切换到了消费者线程，消费者线程将把信息的名称和上一个信息的内容联系到一起。

（2）生产者放了若干次的数据，消费者才开始取数据，或者是，消费者取完一个数据后，还没等到生产者放入新的数据，又重复取出已取过的数据。

9.7.1 程序的基本实现

因为现在程序中生产者不断生产的是信息，而消费者不断取出的也是信息，所以定义一个保存信息的类 Info.java。

范例： Info.java

```
class Info {                                        // 定义信息类
    private String name = "李兴华";                  // 信息名称，指定默认值
    private String content = "JAVA讲师";             // 信息内容，指定默认值
    public String getName() {                       // 取得信息名称
        return name;                                // 返回信息名称
    }
    public void setName(String name) {              // 设置信息名称
        this.name = name;                           // 设置name属性内容
    }
    public String getContent() {                    // 取得信息内容
        return content;                             // 返回信息内容
    }
```

```
    public void setContent(String content) {        // 设置信息内容
        this.content = content;                     // 设置content属性内容
    }
}
```

Info 类的组成非常简单，只包含了用于保存信息名称的 name 属性和用于保存信息内容的 content 属性，因为生产者和消费者要操作同一个空间的内容，所以生产者和消费者分别实现 Runnable 接口，并接收 Info 类的应用。

范例：生产者

```
class Producer implements Runnable {               // 定义生产者线程
    private Info info = null;                       // 保存Info引用
    public Producer(Info info) {                    // 通过构造方法设置info属
                                                       性内容
        this.info = info;                           // 为info属性初始化
    }
    public void run() {                             // 覆写run()方法
        boolean flag = false;                       // 定义标记位
        for (int i = 0; i < 50; i++) {              // 循环50次
            if (flag) {                             // 如果为true，则设置第1
                                                       个信息
                this.info.setName("李兴华");        // 设置信息名称
                try {
                    Thread.sleep(90);               // 加入延迟
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                this.info.setContent("JAVA讲师");   // 设置信息内容
                flag = false;                       // 修改标记位
            } else {                                // 如果为false，则设置第2
                                                       个信息
                this.info.setName("mldn");          // 设置信息名称
                try {
                    Thread.sleep(90);               // 加入延迟
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                this.info.setContent("www.mldnjava.cn"); // 设置信息内容
                flag = true;                        // 修改标记位
            }
        }
    }
}
```

在生产者类的构造方法中传入了 Info 类的实例化对象，然后在 run()方法中循环 50 次以产生信息的具体内容，此外，为了让读者更容易发现问题，本程序中在设置信息名称和内容的地方加入了延迟操作（Thread.sleep()）。

范例：消费者

```
class Consumer implements Runnable {                    // 定义消费者线程
    private Info info = null;                            // 保存Info引用
    public Consumer(Info info) {                         // 通过构造方法设置info属性内容
        this.info = info;                                // 为info属性初始化
    }
    public void run() {                                  // 覆写run()方法
        for (int i = 0; i < 50; i++) {                   // 循环50次
            try {
                Thread.sleep(100);                       // 加入延迟
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(this.info.getName() + " --> "
                    + this.info.getContent());           // 取出信息
        }
    }
}
```

在消费者线程类中也同样接收了一个 Info 对象的引用，并采用循环的方式取出 50 次信息并输出。

范例：测试程序

```
public class ThreadCaseDemo01 {
    public static void main(String[] args) {
        Info i = new Info();                             // 实例化Info对象
        Producer pro = new Producer(i);                  // 实例化生产者，传递Info引用
        Consumer con = new Consumer(i);                  // 实例化消费者，传递Info引用
        new Thread(pro).start();                         // 启动生产者线程
        new Thread(con).start();                         // 启动消费者线程
    }
}
```

程序运行结果（部分）：

```
mldn --> JAVA讲师
李兴华 --> www.mldnjava.cn
李兴华 --> www.mldnjava.cn
mldn --> JAVA讲师
李兴华 --> www.mldnjava.cn
mldn --> JAVA讲师
```

```
mldn --> JAVA讲师
李兴华 --> www.mldnjava.cn
李兴华 --> JAVA讲师
李兴华 --> JAVA讲师
```

因为输出较多，所以以上只列出了程序的部分运行结果，但是从这些运行结果中读者应该已经可以发现，之前提到的两点问题这里已经全部出现了，下面先来解决第 1 个问题，第 1 个问题肯定要使用同步的方式解决，即在一个线程设置完全部内容之后，另外一个线程才能继续操作。

9.7.2　问题解决 1——加入同步

如果要为操作加入同步，则可以通过定义同步方法的方式完成，即将设置名称和姓名定义成一个同步方法，代码如下所示。

范例：修改 Info 类

```
class Info {                                          // 定义信息类
    private String name = "李兴华";                    // 信息名称，指定默认值
    private String content = "JAVA讲师";               // 信息内容，指定默认值
    public synchronized void set(String name,
            String content) {                         // 设置信息名称及内容
        this.setName(name);                           // 设置信息名称
        try {
            Thread.sleep(300);                        // 加入延迟
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        this.setContent(content) ;                    // 设置信息内容
    }
    public synchronized void get() {                  // 取得信息内容
        try {
            Thread.sleep(300);                        // 加入延迟
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(this.getName()
                + " --> " + this.getContent());       // 输出信息
    }
    // setter及getter方法
}
```

以上类中定义了一个 set()和 get()方法，并且都使用 synchronized 关键字进行声明，因为现在不希望直接调用 getter 及 setter 方法，所以修改生产者和消费者类代码如下。

范例：修改生产者

```
class Producer implements Runnable {                    // 定义生产者线程
    private Info info = null;                            // 保存Info引用
    public Producer(Info info) {                         // 通过构造方法设置info属
                                                            性内容
        this.info = info;                                // 为info属性初始化
    }
    public void run() {                                  // 覆写run()方法
        boolean flag = false;                            // 定义标记位
        for (int i = 0; i < 50; i++) {                   // 循环50次
            if (flag) {                                  // 如果为true，则设置第1
                                                            个信息
                this.info.set("李兴华", "JAVA讲师");      // 设置信息
                flag = false;                            // 修改标记位
            } else {                                     // 如果为false，则设置第2
                                                            个信息
                this.info.set("mldn", "www.mldnjava.cn"); // 设置信息
                flag = true;                             // 修改标记位
            }
        }
    }
}
```

范例：修改消费者

```
class Consumer implements Runnable {                     // 定义消费者线程
    private Info info = null;                            // 保存Info引用
    public Consumer(Info info) {                         // 通过构造方法设置info属
                                                            性内容
        this.info = info;                                // 为info属性初始化
    }
    public void run() {                                  // 覆写run()方法
        for (int i = 0; i < 50; i++) {                   // 循环50次
            try {
                Thread.sleep(100);                       // 加入延迟
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            this.info.get();                             // 取出信息
        }
    }
}
```

程序运行结果（部分）：

```
李兴华 --> JAVA讲师
mldn --> www.mldnjava.cn
mldn --> www.mldnjava.cn
mldn --> www.mldnjava.cn
mldn --> www.mldnjava.cn
mldn --> www.mldnjava.cn
李兴华 --> JAVA讲师
mldn --> www.mldnjava.cn
```

从程序的运行结果中可以发现，信息错乱的问题已经解决了，但是依然存在重复读取的问题，既然有重复读取，则肯定会有重复设置的问题，那么对于这样的问题该如何解决呢？此时，就需要使用 Object 类。

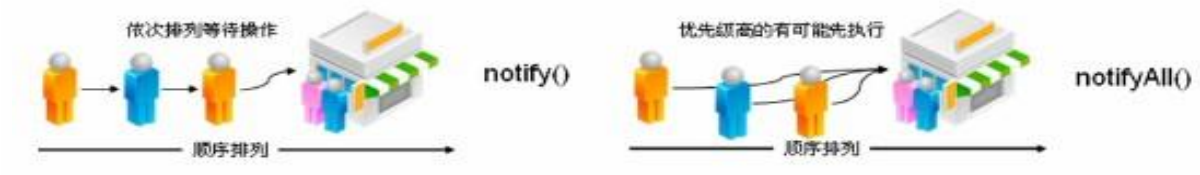
9.7.3　Object 类对线程的支持——等待与唤醒

Object 类是所有类的父类，在此类中有以下几种方法是对线程操作有所支持的，如表 9-3 所示。

表 9-3　Object 类对线程的支持

序　号	方　法	类　型	描　述
1	public final void wait() throws InterruptedException	普通	线程等待
2	public final void wait(long timeout) throws InterruptedException	普通	线程等待，并指定等待的最长时间，以毫秒为单位
3	public final void wait(long timeout,int nanos) throws InterruptedException	普通	线程等待，并指定等待的最长毫秒及纳秒
4	public final void notify()	普通	唤醒第 1 个等待的线程
5	public final void notifyAll()	普通	唤醒全部等待的线程

从表 9-3 中可以发现，可以将一个线程设置为等待状态，但是对于唤醒的操作却有两个，分别为 notify()和 notifyAll()。一般来说，所有等待的线程会按照顺序进行排列，如果现在使用了 notify()方法，则会唤醒第 1 个等待的线程执行，而如果使用了 notifyAll()方法，则会唤醒所有的等待线程，哪个线程的优先级高，哪个线程就有可能先执行，如图 9-9 所示。

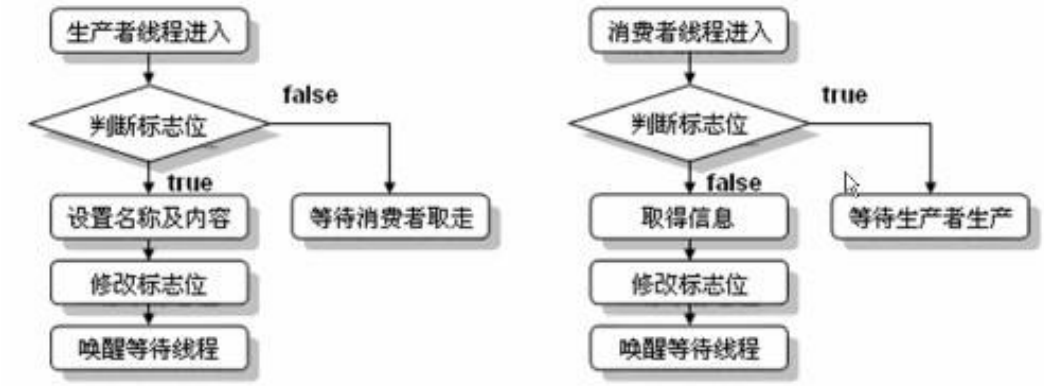


（a）notify 采用顺序操作　　（b）notifyAll 采用非顺序操作

图 9-9　notify()及 notifyAll()的区别

9.7.4 问题解决 2——加入等待与唤醒

如果想让生产者不重复生产，消费者不重复取走，则可以增加一个标志位，假设标志位为 boolean 型变量，如果标志位的内容为 true，则表示可以生产，但是不能取走，此时线程执行到了消费者线程则应该等待；如果标志位的内容为 false，则表示可以取走，但是不能生产，如果生产者线程运行，则应该等待，操作流程如图 9-10 所示。



（a）生产者操作　　（b）消费者操作

图 9-10　操作流程

要完成以上功能，直接修改 Info 类即可。在 Info 类中加入标志位，并通过判断标志位完成等待与唤醒的操作，代码如下。

范例：修改 Info 类

```
class Info {                                          // 定义信息类
    private String name = "李兴华";                    // 信息名称，指定默认值
    private String content = "JAVA讲师";               // 信息内容，指定默认值
    private boolean flag = false ;                    // 设置标志位
    public synchronized void set(String name,
            String content) {                         // 设置信息名称及内容
        if(!flag){                                    // 标志位为false，不可以生产
            try {
                super.wait() ;                        // 等待消费者取走
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        this.setName(name);                           // 设置信息名称
        try {
            Thread.sleep(300);                        // 加入延迟
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
```

```
            this.setContent(content) ;                    // 设置信息内容
            flag = false ;                                // 修改标志位，表示可以取走
            super.notify() ;                              // 唤醒等待线程
        }
        public synchronized void get() {                  // 取得信息内容
            if(flag){                                     // 标志位为true，不可以取走
                try {
                    super.wait() ;                        // 等待生产者生产
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            try {
                Thread.sleep(300);                        // 加入延迟
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(this.getName()
                    + " --> " + this.getContent());// 输出信息
            flag = true ;                                 // 修改标志位为true，表示可以生产
        super.notify() ;                                  // 唤醒等待线程
        }
        // setter()及getter()方法
    }
```

程序运行结果（部分）：

```
李兴华 --> JAVA讲师
mldn --> www.mldnjava.cn
李兴华 --> JAVA讲师
mldn --> www.mldnjava.cn
李兴华 --> JAVA讲师
mldn --> www.mldnjava.cn
李兴华 --> JAVA讲师
mldn --> www.mldnjava.cn
```

从程序的运行结果中可以清楚地发现，生产者每生产一个就要等待消费者取走，消费者每取走一个就要等待生产者生产，这样就避免了重复生产和重复取走的问题。

9.8 线程的生命周期

在 Java 中一个线程对象有自己的生命周期，如果要控制好线程的生命周期，则首先应认识其生命周期，如图 9-11 所示。

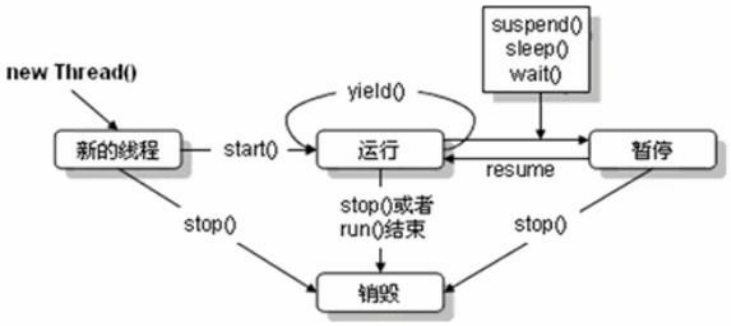


图 9-11　线程的生命周期

从图 9-11 中可以发现，大部分的线程生命周期的方法基本上都已经学过，其中的 3 个新方法介绍如下。

- suspend()方法：暂时挂起线程。
- resume()方法：恢复挂起的线程。
- stop()方法：停止线程。

但是对于线程中 suspend()、resume()、stop() 3 种方法并不推荐使用，因为这 3 种方法在操作时会产生死锁的问题。

注意：suspend()、resume()、stop()方法使用了@Deprecated 声明。

有兴趣的读者可以打开 Thread 类的源代码，可以发现 suspend()、resume()、stop()方法的声明上都加入了一条“@Deprecated”的注释，这属于 Annotation 的语法，表示此操作不建议使用。所以一旦使用了这些方法之后将出现警告信息。

既然以上 3 种方法不推荐使用，那么该如何停止一个线程的执行呢？在多线程的开发中可以通过设置标志位的方式停止一个线程的运行，代码如下。

范例：停止线程运行

```
class MyThread implements Runnable {
    private boolean flag = true;                          // 定义标志位属性
    public void run() {                                   // 覆写run()方法
        int i = 0;
        while (this.flag) {                               // 循环输出
            while (true) {
                System.out.println(Thread.currentThread().getName() + "运
                行，i = "
                        + (i++));                         // 输出当前线程名称
            }
        }
    }
    public void stop() {                                  // 编写停止方法
        this.flag = false;                                // 修改标志位
    }
}
```

```
public class StopDemo {
    public static void main(String[] args) {
        MyThread my = new MyThread();              // 实例化Runnable接口对象
        Thread t = new Thread(my, "线程");          // 建立线程对象
        t.start() ;                                // 启动线程
        my.stop() ;                                // 线程停止，修改标志位
    }
}
```

以上程序一旦调用 stop()方法就会将 MyThread 类中的 flag 变量设置为 false，这样 run()方法就会停止运行，这种停止方式是开发中比较常用的。

9.9　本 章 要 点

1．线程（Thread）是指程序的运行流程。多线程机制可以同时运行多个程序块，使程序运行的效率更高，也解决了传统程序设计语言所无法解决的问题。

2．如果要在类中激活线程，必须先做好下面两项准备：

（1）此类必须是扩展自 Thread 类，使自己成为它的子类。

（2）线程的处理必须编写在 run()方法内。

3．run()方法是定义在 Thread 类中的一种方法，因此把线程的程序代码编写在 run()方法内所做的就是覆写的操作。

4．Runnable 接口中声明了抽象的 run()方法，因此必须在实现 Runnable 接口的类中明确定义 run()方法。

5．在每一个线程创建和消亡之前，均会处于创建、就绪、运行、阻塞、终止状态之一。

6．暂停状态的线程可由下列情况产生：

（1）该线程调用对象的 wait()时。

（2）该线程本身调用 sleep()时。

（3）该线程和另一个线程 join()在一起时。

7．被冻结因素消失的原因有以下两点：

（1）如果线程是由调用对象的 wait()方法冻结，则该对象的 notify()方法被调用时可解除冻结。

（2）线程进入休眠（sleep）状态，但指定的休眠时间到了。

8．当线程的 run()方法运行结束，或是由线程调用其 stop()方法时，线程进入消亡状态。

9．Thread 类中的 sleep()方法可用来控制线程的休眠状态，休眠的时间要视 sleep()中的参数而定。

10．要强制某一线程运行，可用 join()方法。

11．join()方法会抛出 InterruptedException 的异常，所以编写时必须把 join()方法编写在 try…catch 块内。

12. 当多个线程对象操纵同一共享资源时，要使用 synchronized 关键字来进行资源的同步处理。

9.10 习　　题

1. 设计 4 个线程对象，两个线程执行减操作，两个线程执行加操作。

2. 设计一个生产电脑和搬运电脑类，要求生产出一台电脑就搬走一台电脑，如果没有新的电脑生产出来，则搬运工要等待新电脑产出；如果生产出的电脑没有搬走，则要等待电脑搬走之后再生产，并统计出生产的电脑数量。

第 10 章　泛　　型

通过本章的学习可以达到以下目标：

- ☑ 掌握泛型的基本原理及其应用。
- ☑ 掌握泛型通配符的使用。
- ☑ 指定泛型操作中的上限及下限。
- ☑ 在接口上应用泛型。
- ☑ 掌握泛型方法及泛型数组的使用。

在讲解本章之前必须说明的是，本章只是针对于 JDK 1.5 本身的泛型特性进行讲解，如果需要更加合理的应用泛型，就需要结合类集框架及反射机制，因为在 JDK 1.5 之后在类集和反射机制中已经大量应用了泛型操作。

JDK 1.5 中的最显著的变化之一就是添加对泛型类型的支持。所谓的泛型（Generics）就是指在对象建立时不指定类中属性的具体类型，而由外部在声明及实例化对象时指定类型。**本章视频录像讲解时间为 1 小时 52 分钟，源代码在光盘对应的章节下。**

10.1　为什么要使用泛型

在讲解泛型的概念前，先来看以下一个应用实例。

现在要求设计一个可以表示出坐标点的类，坐标由 X 和 Y 组成，坐标的表示方法有以下 3 种。

- ➘ 整数表示：x = 10、y = 20。
- ➘ 小数表示：x = 10.5、y = 20.6。
- ➘ 字符串表示：x = "东经 180 度"、y = "北纬 210 度"。

一看到这样的要求，读者首先就要考虑到，必须建立一个表示坐标点的类 Point，此类中有两个属性分别用来表示 x 坐标和 y 坐标，但是 x 和 y 中所保存的数据类型会有 3 种（int、float、String），而要想使用一个类型可以同时接收 3 种类型数据，则只能使用 Object，因为 Object 类可以接收任何类型的数据，都会自动发生向上转型操作，这样 3 种数据类型将按以下方式进行转换。

- ➘ 数字（int）→自动装箱成 Integer→向上转型使用 Object 接收。
- ➘ 小数（float）→自动装箱成 Float→向上转型使用 Object 接收。
- ➘ 字符串（String）→向上转型使用 Object 接收。

实现思路如图 10-1 所示。

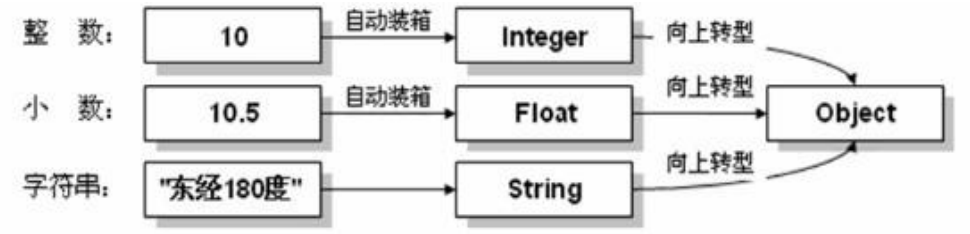


图 10-1 实现思路

按照此思路，此类设计代码如下。

范例：设计 Point 类

```
class Point {
    private Object x;           // 表示x坐标
    private Object y;           // 表示y坐标
    public void setX(Object x) {
        this.x = x;
    }
    public void setY(Object y) {
        this.y = y;
    }
    public Object getX() {
        return this.x;
    }
    public Object getY() {
        return this.y;
    }
};
```

以上程序在定义 Point 属性时使用了 Object 类型，则输入的数据可以是任意的类型。下面分别进行不同数据类型的应用。

范例：现在使用整数表示坐标

```
public class GenericsDemo01 {
    public static void main(String[] args) {
        Point p = new Point() ;
        p.setX(10) ;                // 利用自动装箱操作：int --> Integer -->
                                       Object
        p.setY(20) ;                // 利用自动装箱操作：int --> Integer -->
                                       Object
        int x = (Integer)p.getX() ;// 取出数据时先变为Integer，之后自动拆箱
        int y = (Integer)p.getY() ;// 取出数据时先变为Integer，之后自动拆箱
        System.out.println("整数表示，X坐标为：" + x) ;
        System.out.println("整数表示，Y坐标为：" + y) ;
    }
}
```

程序运行结果：

```
整数表示，X坐标为：10
整数表示，Y坐标为：20
```

以上程序中设置的 x 和 y 坐标的内容是数字，所以会自动发生装箱操作，并将 Integer 的对象利用向上转型的关系变为 Object 类型。下面继续使用小数表示坐标的情况。

范例：现在使用小数表示坐标

```
public class GenericsDemo02 {
    public static void main(String[] args) {
        Point p = new Point() ;
        p.setX(10.5f) ;              // 利用自动装箱操作：float --> Float -->
                                        Object
        p.setY(20.6f) ;              // 利用自动装箱操作：float --> Float -->
                                        Object
        float x = (Float)p.getX() ;// 取出数据时先变为Float，之后自动拆箱
        float y = (Float)p.getY() ;// 取出数据时先变为Float，之后自动拆箱
        System.out.println("小数表示，X坐标为：" + x) ;
        System.out.println("小数表示，Y坐标为：" + y) ;
    }
}
```

程序运行结果：

```
小数表示，X坐标为：10.5
小数表示，Y坐标为：20.6
```

以上程序将一个 float 的坐标设置到 X 和 Y 之中，然后自动进行装箱操作，并将对象进行向上转型，变为 Object 类型。下面继续使用字符串表示坐标的情况。

范例：现在使用字符串表示坐标

```
public class GenericsDemo03 {
    public static void main(String[] args) {
        Point p = new Point() ;
        p.setX("东经180度") ;                // String --> Object
        p.setY("北纬210度") ;                // String --> Object
        String x = (String)p.getX() ;       // 取出数据
        String y = (String)p.getY() ;       // 取出数据
        System.out.println("字符串表示，X坐标为：" + x) ;
        System.out.println("字符串表示，Y坐标为：" + y) ;
    }
}
```

程序运行结果：

```
字符串表示，X坐标为：东经180度
字符串表示，Y坐标为：北纬210度
```

以上程序直接使用字符串作为坐标的内容，所有的 String 类型自动向 Object 进行转换。

以上 3 个程序已经实现了所要求的功能，那么以上的实现是否存在问题呢？仔细考虑一下，Object 类可以接收任意的子类对象，那么也就是说可以把 X 坐标设置成数字，Y 坐标设置成字符串，代码如下所示：

```
public class GenericsDemo04 {
    public static void main(String[] args) {
        Point p = new Point() ;
        p.setX(10) ;                        // 利用自动装箱操作：int --> Integer
                                               --> Object
        p.setY("北纬210度") ;                 // String --> Object
        int x = (Integer)p.getX() ;         // 取出数据
        int y = (Integer)p.getY() ;         // 取出数据  →  此处出现了类转换错误
        System.out.println("整数表示，X坐标为：" + x) ;
        System.out.println("整数表示，Y坐标为：" + y) ;
    }
}
```

以上程序的语法并没有问题，程序可以正常地编译成功，但是程序运行时会发生以下错误：

```
Exception in thread "main" java.lang.ClassCastException: java.lang.String
    at org.lxh.genericsdemo.GenericsDemo04.main(GenericsDemo04.java:29)
```

程序出现了类转换异常，因为设置的 String 类型无法向 Integer 类型转换，之所以造成这样的问题是因为 Point 类中的属性使用 Object 类型进行接收，造成了类型安全问题。而要想解决以上问题就可以使用泛型技术。

10.2 泛 型 应 用

10.2.1 泛型的基本应用

泛型可以解决数据类型的安全性问题，其主要原理是在类声明时通过一个标识表示类中某个属性的类型或者是某个方法的返回值及参数类型。这样在类声明或实例化时只要指定好需要的类型即可。如格式 10-1 和格式 10-2 所示。泛型的指定如图 10-2 所示。

【格式 10-1　泛型类定义】

```
[访问权限] class 类名称<泛型类型标识1,泛型类型标识2,…,泛型类型标识3>{
    [访问权限] 泛型类型标识 变量名称 ;
    [访问权限] 泛型类型标识 方法名称(){} ;
    [访问权限] 返回值类型声明 方法名称(泛型类型标识 变量名称){} ;
}
```

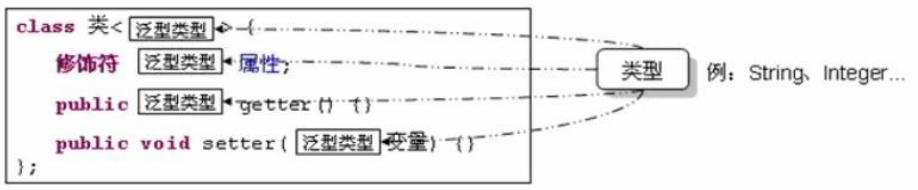


图 10-2　泛型的指定

【格式 10-2　泛型对象定义】

```
类名称<具体类> 对象名称 = new 类名称<具体类>() ;
```

下面使用以上两种格式定义类和声明对象。

范例：声明泛型

```
class Point<T> {                        // 此处可以是任意的标识符号，T是type的简称
    private T var;                      // 此变量的类型由外部决定
    public T getVar() {                 // 返回值的类型由外部指定
        return var;
    }
    public void setVar(T var) {         // 设置的类型由外部指定
        this.var = var;
    }
};
```

上面代码中的 Point 类在声明时使用了“<T>”的形式，T 表示此类型是由外部调用本类时指定的，这里使用任意的字母都可以，如“<A>”、“<B>”，之所以使用“<T>”是因为 T 是 type 的缩写，表示类型，这样比较好理解。之后在类中定义的 var 属性的类型也是 T，这就表示 var 这个属性的类型也是由外部来决定的，不是固定的。同理，setter 方法中的参数类型以及 getter 方法中的返回值类型也由外部设置。

范例：使用 Point 类将 var 的类型设置成整数

```
public class GenericsDemo05 {
    public static void main(String[] args) {
        Point<Integer> p = new Point<Integer>() ;  // 里面的var类型为
                                                   Integer类型
        p.setVar(30) ;                             // 设置数字，自动装箱
        System.out.println(p.getVar() * 2);        // 计算结果，按数字取出
    }
}
```

程序运行结果：

```
60
```

以上程序将 Point 类中的 var 属性设置成 Integer 类型，所以在声明及实例化对象时使用 Point<Integer>。这样实际上上面的 setter 和 getter 方法就变成了以下的格式：

```
    public Integer getVar() {
        return var;
```

```
    }
    public void setVar(Integer var) {
        this.var = var;
    }
```

上面代码只是对设置泛型之后的一种说明，但是可以发现一切操作都是按照数字的方式进行操作的。

提问：关于整型的设置问题。

为什么上面代码中设置泛型时不直接将 var 类型设置成 int 类型，而使用 Point<int>的形式呢？

回答：只能使用包装类。

在泛型的指定中是无法指定基本数据类型的，必须设置成一个类，这样在设置一个数字时就必须使用包装类，而在 JDK 1.5 之后提供了自动装箱的操作，操作时也不会太复杂。

下面按照上面思路再将 Point 类中的 var 属性设置成 String 类型。

范例：使用以上的 Point 类，将 var 设置成 String 类型

```
public class GenericsDemo06 {
    public static void main(String[] args) {
        Point<String> p = new Point<String>() ; // 里面的var类型为String类型
        p.setVar("李兴华") ;                    // 设置字符串
        System.out.println(p.getVar().length());// 取得字符串长度
    }
}
```

程序运行结果：

```
3
```

上面代码中将 var 类型设置成字符串，所以程序的最后可以直接通过 String 类中的 length()方法求出字符串的长度。

如果设置的内容与泛型所指定的类型不一致，则在编译时会出错。

范例：设置的内容与泛型类型不一致

```
public class GenericsDemo07 {
    public static void main(String[] args) {
        Point<Integer> p = new Point<Integer>() ;  // 里面的var类型为
                                                   Integer类型
        p.setVar("李兴华") ;                       // 设置字符串，错误
    }
}
```

编译时出现以下错误：

```
GenericsDemo07.java:13: setVar(java.lang.Integer) in Point<java.lang.Integer>
cannot be applied to (java.lang.String)
```

```
        p.setVar("李兴华") ;          // 设置字符串，错误
          ^
1 error
```

此时泛型设置的是 Integer 类型，所以如果将其设置字符串内容时就出现了以上的错误，而且读者可以直接从错误信息中发现 setVar()方法中只能设置 Integer 类型的数据。

注意：方法操作类型的替换。

加入泛型之后，如果设置的泛型类型是 java.lang.Integer，实际上对于 setVar()方法的定义就相当于“public void setVar(Integer var){}”。

其中能接收的只能是整型的数字，如果是一个整数常量，则会采用自动装箱的机制完成。

10.2.2　使用泛型修改代码

10.2.1 节讲解的坐标类因为要保证 X 和 Y 坐标的数据类型一致，所以最好应用泛型，按照泛型 Point 类将代码修改如下：

```
class Point<T> {                    // 指定泛型类型
    private T x;                    // 表示x坐标，具体类型由外部指定
    private T y;                    // 表示y坐标，具体类型由外部指定
    public void setX(T x) {
        this.x = x;
    }
    public void setY(T y) {
        this.y = y;
    }
    public T getX() {
        return this.x;
    }
    public T getY() {
        return this.y;
    }
};
```

以上 Point 类中的 X 和 Y 坐标的具体类型由外部直接指定，下面直接将其指定为 Integer 类型：

```
public class GenericsPoint {
    public static void main(String[] args) {
        Point<Integer> p = new Point<Integer>() ;  // 定义Point对象，指定
                                                   Integer为泛型类型
        p.setX(10) ;                               // 设置整数，自动装箱
        p.setY(20) ;                               // 设置整数，自动装箱
        int x = p.getX() ;                         // 自动拆箱
```

```
        int y = p.getY() ;                                    // 自动拆箱
        System.out.println("整数表示，X坐标：" + x) ; // 输出信息
        System.out.println("整数表示，Y坐标：" + y) ; // 输出信息
    }
}
```

程序运行结果：

```
整数表示，X坐标：10
整数表示，Y坐标：20
```

如果此时 Y 坐标指定的类型不是 Integer，则编译时将出现如下错误：

```
public class GenericsPoint {
    public static void main(String[] args) {
        Point<Integer> p = new Point<Integer>() ;  // 定义Point对象，指定
                                                      Integer为泛型类型
        p.setX(10) ;                                  // 设置整数，自动装箱
        p.setY("北纬210度") ;                          // 错误，不能设置字符串
        int x = p.getX() ;                            // 自动拆箱
        int y = p.getY() ;                            // 自动拆箱
        System.out.println("整数表示，X坐标：" + x) ; // 输出信息
        System.out.println("整数表示，Y坐标：" + y) ; // 输出信息
    }
}
```

编译时出现错误：

```
GenericsPoint.java:21: setY(java.lang.Integer) in Point<java.lang.Integer>
cannot be applied to (java.lang.String)
                p.setY("北纬210度") ;                  // 设置整数，自动装箱
                 ^
1 error
```

因为将 Integer 设置成泛型类型后，setY()方法能接收的类型就是 Integer，所以将其内容设置成 String 时会因为传递的类型不统一而出现错误提示。

提示：加入泛型后将使程序的操作更加安全。

加入泛型的最大好处实际上就是避免了类转换异常（ClassCastException）的发生，这样将使程序的操作更加安全。

10.2.3 泛型应用中的构造方法

构造方法可以为类中的属性初始化，那么如果类中的属性通过泛型指定，而又需要通过构造设置属性内容时，构造方法的定义与之前并无不同，不需要像声明类那样指定泛型，具体格式如下所示。

【格式 10-3　构造方法上应用泛型】

```
[访问权限] 构造方法 ([<泛型类型> 参数名称]){
}
```

范例：泛型类的构造方法定义

```
class Point<T> {                                    // 此处可以是任意的标识符号，T是type的简称
    private T var;                                  // 此变量的类型由外部决定
    public Point(T var) {
        this.var = var;
    }
    public T getVar() {                             // 返回值的类型由外部指定
        return var;
    }
    public void setVar(T var) {                     // 设置的类型由外部指定
        this.var = var;
    }
};
public class GenericsDemo08 {
    public static void main(String[] args) {
        Point<String> p = null ;                    // 定义泛型类对象
        p = new Point<String>("李兴华") ;           // 里面的var类型为String类型
        System.out.println("内容：" + p.getVar());// 取得字符串
    }
}
```

程序运行结果：

```
内容：李兴华
```

上面程序的构造方法与前面的程序是一样的，也就是说即便有泛型声明，也只是在类的定义上声明，而与构造方法的定义无关。

10.2.4　指定多个泛型类型

如果一个类中有多个属性需要使用不同的泛型声明，则可以在声明类时指定多个泛型类型。

范例：设置多个泛型类型

```
class Notepad<K,V> {                                // 此处指定两个泛型类型
    private K key;                                  // 此变量的类型由外部决定
    private V value;                                // 此变量的类型由外部决定
    public K getKey() {                             // 取得key
        return key;
    }
```

```
    public void setKey(K key) {                          // 设置key
        this.key = key;
    }
    public V getValue() {                                // 取得value
        return value;
    }
    public void setValue(V value) {                      // 设置value
        this.value = value;
    }
};
public class GenericsDemo09 {
    public static void main(String[] args) {
        Notepad<String,Integer> t = null ;               // 定义两个泛型类型的对象
        t = new Notepad<String,Integer>() ;              // 里面的key为String,
                                                            value为Integer
        t.setKey("李兴华") ;                              // 设置第1个内容
        t.setValue(30) ;                                 // 设置第2个内容，自动装箱
        System.out.print("姓名：" + t.getKey());          // 取得信息
        System.out.print(", 年龄：" + t.getValue());     // 取得信息
    }
}
```

程序运行结果：

```
姓名：李兴华，年龄：30
```

上面程序中的 key 属性为 String 类型，value 属性为 Integer，这两个属性的类型由外部指定。

10.3 泛型的安全警告

在泛型应用中最好在声明类对象时指定好其内部的数据类型，如 Info<String>，如果不指定类型，这样用户在使用这样的类时，就会出现不安全的警告信息。

范例：不指定泛型类型

```
class Info<T> {                                  // 此处可以是任意的标识符号，T是type的简称
    private T var;                               // 此变量的类型由外部决定
    public T getVar() {                          // 返回值的类型由外部指定
        return var;
    }
    public void setVar(T var) {                  // 设置的类型由外部指定
        this.var = var;
    }
    public String toString(){                    // 覆写Object类中的toString()方法
        return this.var.toString() ;
```

```
    }
};
public class GenericsDemo10 {
    public static void main(String[] args) {
        Info i = new Info() ;                          // 警告，没有指定泛型类型
        i.setVar("李兴华") ;                            // 设置字符串
        System.out.println("内容: " + i.getVar());     // 取得字符串
    }
}
```

编译时出现以下警告信息：

```
Note: GenericsDemo10.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

因为程序在使用类时没有指定其泛型类型，所以出现了警告信息，表示的信息是使用了不安全的操作，因为没有指定 var 属性的具体类型。

程序运行结果：

```
内容：李兴华
```

以上程序虽然在编译时出现了警告，但是并不影响程序的运行，这是因为在泛型的操作中为了方便用户的使用，就算没有指定泛型程序也可以正常使用，而所有的类型统一使用 Object 进行接收，所以以上程序的 var 属性实际上是 Object 类型的，也就是在定义时将泛型擦除了，以上程序类似于以下的定义。

范例：使用 Object 声明泛型类型

```
public class GenericsDemo11 {
    public static void main(String[] args) {
        Info<Object> i = new Info<Object>() ;          // 指定Object为泛型类型
        i.setVar("李兴华") ;                            // 设置字符串
        System.out.println("内容: " + i.getVar());     // 取得字符串
    }
}
```

程序运行结果：

```
内容：李兴华
```

以上程序本身没有任何的意义，因为就算不设置泛型也是 Object，但这样做的唯一好处是，在编译时警告信息消失了。不指定泛型的程序流程如图 10-3 所示。

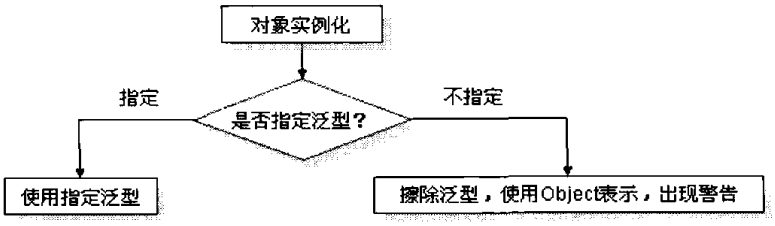


图 10-3　不指定泛型

10.4 通 配 符

以上程序中在操作时都设置了一个固定的类型，在泛型操作中也可以通过通配符接收任意指定泛型类型的对象。

10.4.1 匹配任意类型的通配符

在开发中对象的引用传递是最常见的，但是如果在泛型类的操作中，在进行引用传递时泛型类型必须匹配才可以传递，否则是无法传递的，如下面的代码。

范例：使用泛型声明后的对象引用传递问题

```
public class GenericsDemo12 {
    public static void main(String[] args) {
        Info<String> i = new Info<String>() ;        // 指定String为泛型类型
        i.setVar("李兴华") ;                          // 设置字符串
        fun(i) ;                                     // 错误，无法传递
    }
    public static void fun(Info<Object> temp){       // 此处可以接收Object泛型
                                                        类型的Info对象
        System.out.println("内容: " + temp) ;
    }
}
```

程序编译时错误：

```
GenericsDemo12.java:17: fun(Info<java.lang.Object>) in GenericsDemo12
cannot beapplied to (Info<java.lang.String>)
                fun(i) ;                // 错误，无法传递
                ^
1 error
```

以上程序中尽管 String 是 Object 类的子类，但是在进行引用传递时也同样无法进行操作，如果此时想让程序正确执行，可以将 fun()方法中定义的 Info<Object>修改为 Info，即不指定泛型。

范例：修改以上的程序使之编译通过

```
public class GenericsDemo13 {
    public static void main(String[] args) {
        Info<String> i = new Info<String>() ;  // 指定Object为泛型类型
        i.setVar("李兴华") ;                    // 设置字符串
        fun(i) ;                               // 错误，无法传递
    }
    public static void fun(Info temp){         // 此处可以接收Info的对象
        System.out.println("内容: " + temp) ;
```

```
    }
}
```

程序运行结果：

```
内容：李兴华
```

以上程序编译时不会出现任何的语法错误，程序可以正常地使用，但是在编写 fun()方法时 Info 中并没有指定任何的泛型类型，这样做有些不妥当，所以为了解决这个问题，Java 中引入了通配符“?”，表示可以接收此类型的任意泛型对象，以上代码修改如下。

范例：使用通配符“?”

```
public class GenericsDemo14 {
    public static void main(String[] args) {
        Info<String> i = new Info<String>() ;          // 指定Object为泛型类型
        i.setVar("李兴华") ;                            // 设置字符串
        fun(i) ;                                       // 错误，无法传递
    }
    public static void fun(Info<?> temp){              // 此处可以接收Info的对象
        System.out.println("内容：" + temp) ;
    }
}
```

以上程序在 fun()方法中使用 Info<?>的代码形式，表示可以使用任意的泛型类型对象，这样做的话 fun()方法定义得就比较合理了，但是在使用以上语法时也有一个注意点，即如果使用“?”接收泛型对象时，则不能设置被泛型指定的内容，代码如下所示。

范例：错误的泛型设置

```
public class GenericsDemo15 {
    public static void main(String[] args) {
        Info<?> i = new Info<String>() ;               // 使用“?”接收泛型对象
        i.setVar("李兴华") ;                            // 错误，无法设置
    }
}
```

程序编译时错误：

```
GenericsDemo15.java:16: setVar(capture of ?) in Info<capture of ?> cannot
be applied to (java.lang.String)
                i.setVar("李兴华") ;        // 错误，无法设置
                 ^
1 error
```

以上程序将一个字符串设置给泛型所声明的属性，因为使用 Info<?>的形式，所以无法将内容设置给 var 属性，但此时可以设置 null 值。

范例：设置 null 值

```
public class GenericsDemo16 {
    public static void main(String[] args) {
```

```
        Info<?> i = new Info<String>() ;                    // 使用“?”接收泛型对象
        i.setVar(null) ;                                    // null，可以设置
    }
}
```

10.4.2 受限泛型

在引用传递中，在泛型操作中也可以设置一个泛型对象的范围上限和范围下限。范围上限使用 extends 关键字声明，表示参数化的类型可能是所指定的类型或者是此类型的子类，而范围下限使用 super 进行声明，表示参数化的类型可能是所指定的类型，或者是此类型的父类型，或是 Object 类，具体格式如下。

【格式 10-4　设置上限】

声明对象：类名称<? extends 类> 对象名称
定义类：[访问权限] 类名称<泛型标识 extends 类>{}

【格式 10-5　设置下限】

声明对象：类名称<? super 类> 对象名称
定义类：[访问权限] 类名称<泛型标识 extends 类>{}

下面分别讲解两种格式的应用。

1. 泛型的上限

现在假设一个方法中能接收的泛型对象只能是数字（Byte、Short、Long、Integer、Float、Double）类型，此时在定义方法参数接收对象时，就必须指定泛型的上限。因为所有的数字包装类都是 Number 类型的子类，所以代码编写如下。

范例：设置方法只能接收泛型为 Number 或 Number 类型的子类

```
public class GenericsDemo17 {
    public static void main(String[] args) {
        Info<Integer> i1 = new Info<Integer>() ;    // 声明Integer的泛型对象
        Info<Float> i2 = new Info<Float>() ;        // 声明Integer的泛型对象
        i1.setVar(30) ;                             // 设置整数，自动装箱
        i2.setVar(30.1f) ;                          // 设置小数，自动装箱
        fun(i1) ;                                   // 是数字，可以传递
        fun(i2) ;                                   // 是数字，可以传递
    }
    // 接收Info对象，范围上限设置为Number，所以只能接收数字类型
    public static void fun(Info<? extends Number> temp){
        System.out.print(temp + "、");
    }
}
```

程序运行结果：

```
30、30.1、
```

以上程序在 fun()方法中只能接收数字类型的 Info 类的泛型对象，如果此时传递的是一个 String 类的泛型对象，则编译时将出现错误。

范例：错误的泛型传递

```
public class GenericsDemo18 {
    public static void main(String[] args) {
        Info<String> i1 = new Info<String>() ; // 声明String的泛型对象
        i1.setVar("hello") ;                   // 设置字符串
        fun(i1) ;                              // 错误，不是数字，不可以传递
    }
    public static void fun(Info<? extends Number> temp){
        System.out.print(temp + "、");
    }
}
```

程序编译时错误：

```
GenericsDemo18.java:17: fun(Info<? extends java.lang.Number>) in
GenericsDemo18cannot be applied to (Info<java.lang.String>)
                fun(i1) ;              // 错误，不是数字，不可以传递
                ^
1 error
```

也可以直接在类的声明处指定泛型的上限范围，代码如下所示。

范例：在类定义时指定泛型的上限

```
class Info<T extends Number> {          // 此处泛型只能是数字类型
    private T var;                      // 此变量的类型由外部决定
    public T getVar() {                 // 返回值的类型由外部指定
        return var;
    }
    public void setVar(T var) {         // 设置的类型由外部指定
        this.var = var;
    }
    public String toString(){           // 覆写Object类中的toString()方法
        return this.var.toString() ;
    }
};
```

以上代码 Info 类中泛型的范围就是所有的数字，如果此时声明的泛型对象是 Number 的子类，则肯定不会有任何问题；如果声明的不是数字类型，则肯定会出错。

范例：正确的声明

```
public class GenericsDemo19 {
    public static void main(String[] args) {
        Info<Integer> i1 = new Info<Integer>() ;    // 声明Integer的泛型对象
        System.out.println("内容：" + i1);
```

```
    }
}
```

范例：错误的声明

```
public class GenericsDemo20 {
    public static void main(String[] args) {
        Info<String> i1 = new Info<String>() ;      // 错误，声明的是String的
                                                       泛型对象
        System.out.println("内容: " + i1);
    }
}
```

2. 泛型的下限

当使用的泛型只能在本类及其父类类型上应用时，就必须使用泛型的范围下限进行配置。

范例：认识下限

```
class Info<T> {                          // 此处泛型只能是数字类型
    private T var;                       // 此变量的类型由外部决定
    public T getVar() {                  // 返回值的类型由外部指定
        return var;
    }
    public void setVar(T var) {          // 设置的类型由外部指定
        this.var = var;
    }
    public String toString(){            // 覆写Object类中的toString()方法
        return this.var.toString() ;
    }
};
public class GenericsDemo21 {
    public static void main(String[] args) {
        Info<Object> i1 = new Info<Object>() ; // 满足下限范围
        Info<String> i2 = new Info<String>() ; // 满足下限范围
        i1.setVar(new Object()) ;              // 设置Object对象
        i2.setVar("李兴华") ;                  // 设置字符串
        fun(i1) ;
        fun(i2) ;
    }
    public static void fun(Info<? super String> temp){ // 只能接收String或
Object类型的泛型
        System.out.println("内容: " + temp) ;
    }
}
```

程序运行结果：

```
内容：java.lang.Object@757aef
内容：李兴华
```

在以上代码的fun()方法中，Info进行了下限的配置，所以只能接收泛型是String及Object类型的引用。所以，一旦此时传递了其他泛型类型的对象，编译时将出现错误。

范例：错误的应用

```
public class GenericsDemo22 {
    public static void main(String[] args) {
        Info<Integer> i1 = new Info<Integer>() ;    // 满足下限范围
        i1.setVar(30) ;                             // 自动装箱
        fun(i1) ;                                   // 错误，不满足下限要求
    }
    public static void fun(Info<? super String> temp){ // 只能接收String或
                                                        Object类型的泛型
        System.out.println("内容：" + temp) ;
    }
}
```

编译时出现以下错误：

```
GenericsDemo23.java:17: fun(Info<? super java.lang.String>) in GenericsDemo23 cannot be applied to (Info<java.lang.Integer>)
                fun(i1) ;                   // 错误，不满足下限要求
                ^
1 error
```

因为指定了泛型下限，所以在将泛型设置成 Integer 类型时就出现了编译错误。

10.5　泛型与子类继承的限制

一个类的子类可以通过对象多态性为其父类实例化，但是在泛型操作中，子类的泛型类型是无法使用父类的泛型类型接收的，例如，Info <String>不能使用 Info<Object>接收。

范例：观察如下的程序代码

```
public class GenericsDemo23 {
    public static void main(String args[]){
        Info<String> i1 = new Info<String>() ; // 泛型类型为String
        Info<Object> i2 = null ;               // 泛型类型为Object
        i2 = i1 ;                   // 两个Info对象进行转换，Info<String> -->
                                       Info<Object>
    }
}
```

程序编译时错误：

```
GenericsDemo23.java:17: incompatible types
found   : Info<java.lang.String>
required: Info<java.lang.Object>
                i2 = i1 ;
                     ^
1 error
```

以上错误提示的含义是，不匹配的类型，即 Info<String>无法转换为 Info<Object>。虽然 String 是 Object 类的子类，但是在泛型操作中此概念无效，此时只能使用“?”接收。

提问：为什么这里不能使用向上转型？

String 是 Object 的子类，通过对象的多态性 Object 类可以接收任意引用类型的对象，为什么到了泛型中却无法使用了？

回答：如果将子类泛型变为父类泛型，则表示扩大了子类的内容。

以上代码有可能是读者在学习泛型中最难理解的部分，因为和之前的概念相反。为方便读者理解，下面举一个生活中的场景为读者说明：以商场购物为例，现在假设把以上的两个对象 Info<Object>和 Info<String>分别当作商场的全部商品和个人已购买的商品信息。一个人所购买的肯定是商场中很少的一部分商品，而如果现在使用“Info<Object> = Info<String>”，就相当于在个人已购买的商品加入了商场的全部商品，相当于个人把整个商场的商品全部买走了，这基本上是不可能的，所以程序无法编译通过，如图 10-4 所示。



图 10-4　泛型中无法向上转型

10.6　泛 型 接 口

10.6.1　定义泛型接口

在 JDK 1.5 之后，不仅可以声明泛型类，也可以声明泛型接口，声明泛型接口和声明泛型类的语法类似，也是在接口名称后面加上<T>，格式如下所示。

【格式 10-6　泛型接口】

```
[访问权限] interface 接口名称<泛型标识>{
}
```

下面利用以上格式定义一个泛型接口。

范例：定义泛型接口

```
interface Info<T>{                          // 在接口上定义泛型
    public T getVar() ;
}
```

10.6.2　泛型接口的两种实现方式

泛型接口定义完成之后，就要定义此接口的子类，定义泛型接口的子类有两种方式，一种是直接在子类后声明泛型，另一种是直接在子类实现的接口中明确地给出泛型类型。

范例：定义子类方式 1——在子类的定义上声明泛型类型

```
interface Info<T>{                          // 在接口上定义泛型
    public T getVar() ;
}
class InfoImpl<T> implements Info<T>{       // 定义泛型接口的子类
    private T var ;                         // 定义属性
    public InfoImpl(T var){                 // 通过构造方法设置属性内容
        this.setVar(var) ;
    }
    public void setVar(T var){
        this.var = var ;
    }
    public T getVar(){
        return this.var ;
    }
}
```

以上程序泛型接口的子类声明了与接口中同样的泛型标识，使用以上子类的方式与前面的程序类似，代码如下。

范例：使用泛型接口的子类

```
public class GenericsDemo24 {
    public static void main(String args[]){
        Info<String> i = null ;                 // 定义接口对象
        i = new InfoImpl<String>("李兴华") ;    // 通过子类实例化此对象
        System.out.println("内容：" + i.getVar()) ;
    }
}
```

程序运行结果：

```
内容：李兴华
```

以上程序并不难理解，虽然指定了泛型，但是依然可以使用对象的多态性通过一个子类为接口实例化。

范例：定义子类方式 2——直接在接口中指定具体类型

```
interface Info<T>{                          // 在接口上定义泛型
    public T getVar() ;
}
class InfoImpl implements Info<String>{// 定义泛型接口的子类，指定类型为String
    private String var ;                    // 定义属性
    public InfoImpl(String var){            // 通过构造方法设置属性内容
        this.setVar(var) ;
    }
    public void setVar(String var){
        this.var = var ;
    }
    public String getVar(){                 // 接口上已指明类型，所以此处必须是String
        return this.var ;
    }
}
```

以上程序泛型接口的子类在实现接口时，直接在实现的接口处指定了具体的泛型类型 String，这样在覆写 Info 接口中的 getVar()方法时直接指明类型为 String 即可。

范例：使用泛型接口的子类

```
public class GenericsDemo25 {
    public static void main(String args[]){
        Info<String> i = null ;          // 定义接口对象，指定泛型
        i = new InfoImpl("李兴华") ;     // 通过子类实例化此对象，不用指定泛型
        System.out.println("内容：" + i.getVar()) ;
    }
}
```

程序运行结果：

```
内容：李兴华
```

以上程序在实例化子类对象的使用时不用再指定泛型，因为在声明子类时已经明确地指定了具体类型。

提示：泛型接口在 Java 类库中大量的使用。

在 JDK 1.5 之后实际上很多的接口都加入了泛型的支持，这样在使用这些接口时为了防止安全警告信息的出现，都要指定具体的泛型类型。

10.7 泛 型 方 法

前面学习的所有泛型操作都是将整个类进行泛型化，但同样也可以在类中定义泛型化的方法。泛型方法的定义与其所在的类是否是泛型类是没有任何关系的，所在的类可以是

泛型类，也可以不是泛型类。

10.7.1　定义泛型方法

在泛型方法中可以定义泛型参数，此时，参数的类型就是传入数据的类型，可以使用如下的格式定义泛型方法。

【格式 10-7　泛型方法的简单定义】

[访问权限] <泛型标识> 泛型标识 方法名称([泛型标识 参数名称])

范例：定义一个泛型方法

```
class Demo{
    public <T> T fun(T t){                    // 可以接收任意类型的数据
        return t ;
    }
}
public class GenericsDemo26 {
    public static void main(String args[]){
        Demo d = new Demo() ;
        String str = d.fun("李兴华") ;        // 传递字符串
        int i = d.fun(30) ;                   // 传递数字，自动装箱
        System.out.println(str) ;             // 输出内容
        System.out.println(i) ;               // 输出内容
    }
}
```

程序运行结果：

```
李兴华
30
```

以上程序中的 fun()方法是将接收的参数直接返回，而且因为在方法接收参数中使用了泛型操作，所以此方法可以接收任意类型的数据，而且此方法的返回值类型将由泛型指定。

10.7.2　通过泛型方法返回泛型类实例

如果可以通过泛型方法返回一个泛型类的实例化对象，则必须在方法的返回类型声明处明确地指定泛型标识。

范例：通过方法返回泛型类实例

```
class Info<T extends Number> {                // 此处泛型只能是数字类型
    private T var;                            // 此变量的类型由外部决定
    public T getVar() {                       // 返回值的类型由外部指定
        return var;
    }
```

```
    public void setVar(T var) {          // 设置的类型由外部指定
        this.var = var;
    }
    public String toString(){            // 覆写Object类中的toString()方法
        return this.var.toString() ;
    }
};
public class GenericsDemo27 {
    public static void main(String args[]){
        Info<Integer> i = fun(30) ;      // 传递整数到fun()方法
        System.out.println(i.getVar()) ;
    }
    public static <T extends Number> Info<T> fun(T param){
        Info<T> temp = new Info<T>() ;   // 根据传入的数据类型实例化Info对象
        temp.setVar(param) ;             // 将传递的内容设置到Info类中的var属性
                                            之中
        return temp ;                    // 返回实例化对象
    }
}
```

程序运行结果：

```
30
```

上面代码中的 fun()方法在 static 关键字处定义了“<T **extends** Number>”的意思是，方法中传入或返回的泛型类型由调用方法时所设置的参数类型决定。

10.7.3 使用泛型统一传入的参数类型

如果现在一个方法要求传入的泛型对象的泛型类型一致，也可以通过泛型方法指定。

范例：统一输入对象的泛型类型

```
class Info<T> {
    private T var;                       // 此变量的类型由外部决定
    public T getVar() {                  // 返回值的类型由外部指定
        return var;
    }
    public void setVar(T var) {          // 设置的类型由外部指定
        this.var = var;
    }
    public String toString(){            // 覆写Object类中的toString()方法
        return this.var.toString() ;
    }
};
public class GenericsDemo28 {
```

```
    public static void main(String args[]) {
        Info<String> i1 = new Info<String>();  // 设置String为泛型类型
        Info<String> i2 = new Info<String>();  // 设置String为泛型类型
        i1.setVar("HELLO") ;                   // 设置内容
        i2.setVar("李兴华") ;                  // 设置内容
        add(i1,i2) ;
    }
    public static <T> void add(Info<T> i1, Info<T> i2) {
        System.out.println(i1.getVar() + " " + i2.getVar());
    }
}
```

程序运行结果：

```
HELLO 李兴华
```

上面程序中 add 方法中的两个 Info 对象的泛型类型必须一致。如果设置的类型不一致，则在编译时将出现错误。

范例：错误的程序

```
public class GenericsDemo29 {
    public static void main(String args[]) {
        Info<Integer> i1 = new Info<Integer>();// 设置Integer为泛型类型
        Info<String> i2 = new Info<String>();  // 设置String为泛型类型
        i1.setVar(30) ;                        // 设置内容
        i2.setVar("李兴华") ;                  // 设置内容
        add(i1,i2) ;
    }
    public static <T> void add(Info<T> i1, Info<T> i2) {
        System.out.println(i1.getVar() + " " + i2.getVar());
    }
}
```

程序编译时错误：

```
GenericsDemo29.java:19: <T>add(Info<T>,Info<T>) in GenericsDemo30 cannot be
applied to (Info<java.lang.Integer>,Info<java.lang.String>)
                add(i1,i2) ;
                ^
1 error
```

使用此种方法可以为程序操作的安全性提供保障。

10.8　泛 型 数 组

使用泛型方法时，也可以传递或返回一个泛型数组，代码如下所示。

范例：接收和返回泛型数组

```
public class GenericsDemo30 {
    public static void main(String args[]) {
        Integer i[] = fun1(1,2,3,4,5,6) ;          // 返回泛型数组
        fun2(i) ;                                  // 输出数组内容
    }
    public static <T> T[] fun1(T...arg){           // 接收可变参数，返回泛型数组
        return arg ;                               // 返回泛型数组
    }
    public static <T> void fun2(T param[]) {       // 接收泛型数组
        System.out.print("接收泛型数组：") ;
        for(T t : param){
            System.out.print(t + "、") ;
        }
        System.out.println() ;
    }
}
```

程序运行结果：

```
接收泛型数组：1、2、3、4、5、6、
```

以上程序从 fun1()方法返回一个泛型数组，在 fun1()方法接收参数时使用了可变参数传递方式，然后将 fun1()返回的泛型数组内容交给 fun2()方法进行输出。

10.9 泛型的嵌套设置

也可以在一个类的泛型中指定另外一个类的泛型，代码如下所示。

范例：定义两个泛型类

```
class Info<T, V> {                    // 指定两个泛型类型
    private T var;                    // 第1个泛型属性
    private V value;                  // 第2个泛型属性
    public Info(T var, V value) {     // 通过构造方法设置
        this.setVar(var);
        this.setValue(value);
    }
    public T getVar() {
        return var;
    }
    public void setVar(T var) {
        this.var = var;
    }
```

```
    public V getValue() {
        return value;
    }
    public void setValue(V value) {
        this.value = value;
    }
}
class Demo<S> {
    private S info;
    public Demo(S info) {
        this.setInfo(info);
    }
    public S getInfo() {
        return info;
    }
    public void setInfo(S info) {
        this.info = info;
    }
}
```

以上程序的 Info 类需要指定两个泛型类型，而 Demo 类也需要指定一个泛型类型，在操作时可以将 Info 设置成 Demo 的泛型类型。

范例：设置嵌套泛型

```
public class GenericsDemo31 {
    public static void main(String args[]) {
        Demo<Info<String, Integer>> d = null;  // 将Info作为Demo的泛型类型
        Info<String, Integer> i = null ;       // Info要指定两个泛型类型
        i = new Info<String, Integer>("李兴华", 30);
        d = new Demo<Info<String, Integer>>(i);// 在Demo类中设置Info类对象
        System.out.println("内容一：" + d.getInfo().getVar());
        System.out.println("内容二：" + d.getInfo().getValue());
    }
}
```

程序运行结果：

```
内容一：李兴华
内容二：30
```

10.10　范例——泛型应用

用户在设计类时往往会使用类的关联关系，例如，一个人中可以定义一个信息的属性，但是一个人可能有各种各样的信息（如联系方式、基本信息等），所以此信息属性的类型

就可以通过泛型进行声明，然后只要设计相应的信息类即可，如图 10-5 所示。

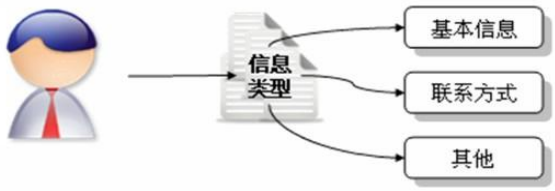


图 10-5　人的信息表示类型

但是需要注意的是，既然需要的是一个信息的类，所以在设计此类时最好做一个信息的标识接口，只要传递的是此接口的子类就可以设置成泛型类型。

范例：定义标识接口——信息

```
interface Info{                                    // 定义一个标识接口，此接口没有定义任何方法
}
```

范例：定义第 1 个表示信息的类——联系方式，此类实现 Info 接口

```
class Contact implements Info{                                    // 实现Info接口
    private String address ;                                      // 联系地址
    private String telphone ;                                     // 联系方式
    private String zipcode ;                                      // 邮政编码
    public Contact(String address,String telphone,String zipcode){
        this.setAddress(address) ;
        this.setTelphone(telphone) ;
        this.setZipcode(zipcode) ;
    }
    public String getAddress() {                                  // 取得联系地址
        return address;
    }
    public void setAddress(String address) {                      // 设置联系地址
        this.address = address;
    }
    public String getTelphone() {                                 // 取得联系电话
        return telphone;
    }
    public void setTelphone(String telphone) {                    // 设置联系电话
        this.telphone = telphone;
    }
    public String getZipcode() {                                  // 取得邮政编码
        return zipcode;
    }
    public void setZipcode(String zipcode) {                      // 设置邮政编码
        this.zipcode = zipcode;
    }
```

```
    public String toString(){                          // 覆写Object类中的toString()
                                                          方法
        return  "联系方式：" + "\n" +
                "\t|- 联系电话：" + this.telphone + "\n" +
                "\t|- 联系地址：" + this.address + "\n" +
                "\t|- 邮政编码：" + this.zipcode ;// 返回对象的信息
    }
}
```

范例：定义第 1 个表示信息的类——个人基本信息，此类实现 Info 接口

```
class Introduction implements Info{                    // 实现Info接口
    private String name ;                              // 定义name属性
    private String sex ;                               // 定义sex属性
    private int age ;                                  // 定义age属性
    public Introduction(String name,String sex,int age){
        this.setName(name) ;
        this.setSex(sex) ;
        this.setAge(age) ;
    }
    public String getName() {                          // 取得姓名
        return name;
    }
    public void setName(String name) {                 // 设置姓名
        this.name = name;
    }
    public String getSex() {                           // 取得性别
        return sex;
    }
    public void setSex(String sex) {                   // 设置性别
        this.sex = sex;
    }
    public int getAge() {                              // 取得年龄
        return age;
    }
    public void setAge(int age) {                      // 设置年龄
        this.age = age;
    }
    public String toString(){                          // 覆写Object类中的toString()
                                                          方法
        return  "基本信息：" + "\n" +
                "\t|- 姓名：" + this.name + "\n" +
                "\t|- 性别：" + this.sex + "\n" +
                "\t|- 年龄：" + this.age ;             // 返回对象信息
```

```
    }
}
```

范例：定义 Person 类，Person 类中 info 属性的类型使用泛型

```
class Person<T extends Info> {          // 此处指定了上限，必须是Info接口的子类
    private T info;                     // 此变量的类型由外部决定
    public Person(T info){              // 通过构造方法设置信息属性内容
        this.setInfo(info) ;
    }
    public T getInfo() {                // 取得info属性
        return info;
    }
    public void setInfo(T info) {       // 设置info属性
        this.info = info;
    }
    public String toString(){           // 覆写Object类中的toString()方法
        return this.info.toString() ;
    }
};
```

上面的 Person 程序中 info 属性并没有指定具体类型，此类型将由程序外部决定，但必须是 Info 接口的子类，而在 Person 类中覆写的 toString()方法也只是返回 info 属性中 toString()方法的内容。下面介绍应该如何应用上面的程序。

范例：将 Contact 类型设置成泛型类型

```
public class GenericsDemo32 {
    public static void main(String[] args) {
        Person<Contact> per = null ;    // 声明Person对象，同时指定Contact类型
        // 实例化Person对象，同时设置info的信息
        per = new Person<Contact>(new Contact("北京市","01051283346",
        "100088")) ;
        System.out.println(per);
    }
}
```

程序运行结果：

```
联系方式：
    |- 联系电话：01051283346
    |- 联系地址：北京市
    |- 邮政编码：100088
```

范例：将 Introduction 类型设置成泛型类型

```
public class GenericsDemo33 {
    public static void main(String[] args) {
```

```
        Person<Introduction> per = null ;        // 声明Person对象，同时指定
                                                    Introduction类型
        // 实例化Person对象，同时设置info的信息
        per = new Person<Introduction>(new Introduction("李兴华","男",30)) ;
        System.out.println(per);
    }
}
```

程序运行结果：

```
基本信息：
    |- 姓名：李兴华
    |- 性别：男
    |- 年龄：30
```

10.11　本 章 要 点

1. 泛型可以使程序的操作更加安全，可以避免发生类转换异常。
2. 在程序中如果使用类时没有指定泛型，则泛型将被擦除掉，将使用 Object 接收参数。
3. 可以使用通配符“?”接收全部的泛型类型对象。
4. 通过<? extends 类>可以设置泛型的上限，通过<? super 类>可以设置泛型的下限。
5. 泛型方法可以定义在泛型类中，也可以定义在普通类中。
6. 泛型可以在接口中定义，实现泛型接口的子类要指明具体的泛型类型。
7. 泛型可以嵌套使用。
8. 在程序中定义没有方法的接口，这样的接口一般称为标识接口。

10.12　习　　题

按照要求定义一个操作类：要求完成一个数组操作类，其中可以加入任意类型的数据，数组具体的操作类型由程序外部决定，并且可以实现查询功能。

希望：读者能把泛型的知识反复的熟练一下

第 11 章　Java 常用类库

通过本章的学习可以达到以下目标：

- ☑ 掌握 String 与 StringBuffer 的区别，可以熟练使用 StringBuffer 中的各种方法进行相关操作。
- ☑ 能够自己编写一个得到日期的操作类，并将这些日期进行格式化操作。
- ☑ 掌握比较器及其基本原理，并可以通过比较器进行对象数组的比较操作。
- ☑ 掌握对象克隆技术及其实现要求。
- ☑ 能够灵活地应用正则表达式对字符串的组成进行判断。
- ☑ 了解并使用 Java 中提供的观察者操作机制。
- ☑ 认识 Java 中的 Random、Locale、Math 等常用类，并可以使用 Local 进行国际化程序的编写。
- ☑ 描述出 Object、System 类对垃圾收集的支持。
- ☑ 使用 NumberFormat、DecimalFormat、BigInteger、BigDecimal 进行数字的操作。
- ☑ 了解 Java 定时器任务调度的实现。

在 Java 中提供了很多的操作类库，为读者开发程序提供了方便，之前讲解数组时的排序操作方法及 String 类等，都是 Java 提供给用户的类库。本章将介绍在实际开发时 Java 中出现的各种常见类库。**本章视频录像讲解时间为 5 小时 38 分钟，源代码在光盘对应的章节下。**

11.1　StringBuffer 类

11.1.1　认识 StringBuffer 类

前面已经详细地讲解了 String 类的各种概念，其中有一条特性是 String 的内容一旦声明则不可改变，如果要改变，则改变的肯定是 String 的引用地址，那么如果一个字符串要被经常改变，则就必须使用 StringBuffer 类。从前面的学习中读者已经清楚地知道，在 String 类中可以通过“+”进行字符串的连接，但在 StringBuffer 中却只能使用 append 方法进行字符串的连接，如图 11-1 所示。

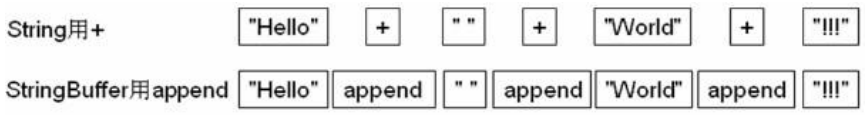


图 11-1　String 与 StringBuffer 连接字符串的不同操作

表 11-1 中列出了 StringBuffer 中的一些常用方法，其他方法读者可以自行查询 JDK 文档。

表 11-1　StringBuffer 的常用方法

序号	方 法 定 义	类型	描　　述
1	public StringBuffer()	构造	StringBuffer 的构造方法
2	public StringBuffer append(char c)	方法	在 StringBuffer 中提供了大量的追加操作（与 String 中使用“+”类似），可以向 StringBuffer 中追加内容，此方法可以添加任何的数据类型
3	public StringBuffer append(String str)	方法	
4	public StringBuffer append(StringBuffer sb)	方法	
5	public int indexOf(String str)	方法	查找指定字符串是否存在
6	public int indexOf(String str,int fromIndex)	方法	从指定位置开始查找指定字符串是否存在
7	public StringBuffer insert(int offset,String str)	方法	在指定位置处加上指定字符串
8	public StringBuffer reverse()	方法	将内容反转保存
9	public StringBuffer replace(int start,int end, String str)	方法	指定内容替换
10	public int length()	方法	求出内容长度
11	public StringBuffer delete(int start,int end)	方法	删除指定范围的字符串
12	public String substring(int start)	方法	字符串截取，指定开始点
13	public String substring(int start,int end)	方法	截取指定范围的字符串
14	public String toString()	方法	Object 类继承的方法，用于将内容变为 String 类型

下面来一一验证这些方法。

提示：StringBuffer 支持的方法大部分与 String 类似。

因为 StringBuffer 在开发中可以提升代码的性能，所以使用较多，这样为了保证用户操作的适应性，在 StringBuffer 类中定义的大部分方法名称都与 String 是一样的。

1．实例操作一：字符串连接操作

在程序中使用 append()方法可以进行字符串的连接，而且此方法返回了一个 StringBuffer 类的实例，这样就可以采用代码链的形式一直调用 append()方法，代码如下所示。

范例：通过 append 连接各种类型的数据

```
package org.lxh.demo11.stringbufferdemo;
public class StringBufferDemo01 {
    public static void main(String[] args) {
        StringBuffer buf = new StringBuffer();          // 声明StringBuffer对象
        buf.append("Hello ");                           // 向StringBuffer中添加
                                                           内容
        buf.append("World").append("!!!");              // 可以连续调用append方法
        buf.append("\n");                               // 添加一个转义字符表示换行
        buf.append("数字 = ").append(1).append("\n");    // 可以添加数字
```

```
        buf.append("字符 = ").append('C').append("\n"); // 可以添加字符
        buf.append("布尔 = "). append(true);             // 可以添加布尔类型
        System.out.println(buf);                         // 内容输出
    }
}
```

程序运行结果：

```
Hello World!!!
数字 = 1
字符 = C
布尔 = true
```

以上代码中的“buf.append("数字 = ").append(1).append("\n")”实际上就是一种代码链的操作形式。

范例：验证 StringBuffer 的内容是可以修改的

```
package org.lxh.demo11.stringbufferdemo;
public class StringBufferDemo02 {
    public static void main(String[] args) {
        StringBuffer buf = new StringBuffer(); // 声明StringBuffer对象
        buf.append("Hello ");                  // 向StringBuffer中添加内容
        fun(buf) ;                             // 传递StringBuffer引用
        System.out.println(buf);               // 将修改后的结果输出
    }
    public static void fun(StringBuffer s){    // 接收StringBuffer引用
        s.append("MLDN ").append("LiXingHua") ;// 修改StringBuffer内容
    }
}
```

程序运行结果：

```
Hello MLDN LiXingHua
```

从程序的运行结果中可以发现，将 StringBuffer 对象的内容传递到了 fun()方法后，对 StringBuffer 的内容进行修改，而且操作完毕后修改的内容将被保留下来，所以与 String 比较 StringBuffer 的内容是可以修改的。

2. 实例操作二：在任意位置处为 StringBuffer 添加内容

可以直接使用 insert()方法在指定的位置上为 StringBuffer 添加内容，代码如下。

范例：在任意位置处为 StringBuffer 添加内容

```
package org.lxh.demo11.stringbufferdemo;
public class StringBufferDemo03 {
    public static void main(String[] args) {
        StringBuffer buf = new StringBuffer(); // 声明StringBuffer对象
        buf.append("World!!");                 // 向StringBuffer中添加内容
```

```
        buf.insert(0, "Hello ") ;                    // 在所有内容之前添加
        System.out.println(buf);                     // 将修改后的结果输出
        buf.insert(buf.length(),"MLDN ~") ;          // 在最后添加
        System.out.println(buf);                     // 将内容输出
    }
}
```

程序运行结果：

```
Hello World!!
Hello World!!MLDN ~
```

3．实例操作三：字符串反转操作

在 StringBuffer 中专门提供了字符串反转的操作方法，所谓的字符串反转就是指将一个是“Hello”的字符串转为“olleH”。

范例：字符串反转操作

```
package org.lxh.demo11.stringbufferdemo;
public class StringBufferDemo04 {
    public static void main(String[] args) {
        StringBuffer buf = new StringBuffer(); // 声明StringBuffer对象
        buf.append("World!!");                 // 向StringBuffer中添加内容
        buf.insert(0, "Hello ") ;              // 在所有内容之前添加
        String str = buf.reverse().toString() ;// 将内容反转后变为String类型
        System.out.println(str);               // 将内容输出
    }
}
```

程序运行结果：

```
!!dlroW olleH
```

以上程序通过 append()和 insert()方法向 StringBuffer 加入数据后，使用 reverse()方法将所有的内容以逆序的方式输出。

提示：字符串反转操作是一种较为常见的操作。

在字符串的操作中，反转是一种较为常见的操作，最早的字符串反转实际上是通过入栈及出栈功能完成的，关于栈（Stack）在 Java 类集中将为读者介绍。

4．实例操作四：替换指定范围的内容

在 StringBuffer 类中也存在 replace()方法，使用此方法可以对指定范围的内容进行替换。

范例：替换指定范围的内容

```
package org.lxh.demo11.stringbufferdemo;
public class StringBufferDemo05 {
    public static void main(String[] args) {
```

```
        StringBuffer buf = new StringBuffer();          // 声明StringBuffer
                                                           对象
        buf.append("Hello ").append("World!!");         // 向StringBuffer中
                                                           添加内容
        buf.replace(6, 11, "LiXingHua");                // 将World替换为
                                                           LiXingHua
        System.out.println("内容替换之后的结果：" + buf); // 删除替换之后的内容
    }
}
```

程序运行结果：

```
内容替换之后的结果：Hello LiXingHua!!
```

在 String 中如果要进行替换，则使用的是 replaceAll()方法，而在 StringBuffer 中使用的是 replace()方法，这一点读者在使用时需要注意。

5. 实例操作五：字符串截取

通过 substring()方法可以直接从 StringBuffer 的指定范围中截取出内容。

范例：截取指定范围的字符串内容

```
package org.lxh.demo11.stringbufferdemo;
public class StringBufferDemo06 {
    public static void main(String[] args) {
        StringBuffer buf = new StringBuffer();      // 声明StringBuffer对象
        buf.append("Hello ").append("World!!");     // 向StringBuffer中添加
                                                       内容
        buf.replace(6, 11, "LiXingHua");            // 将World替换为
                                                       LiXingHua
        String str = buf.substring(6, 15);          // 指定截取范围，并把内容转
                                                       化为String类型
        System.out.println("截取之后的内容：" + str); // 将截取后的结果输出
    }
}
```

程序运行结果：

```
截取之后的内容：LiXingHua
```

6. 实例操作六：删除指定范围的字符串

因为 StringBuffer 本身的内容是可更改的，所以也可以通过 delete()方法删除指定范围的内容。

范例：从 StringBuffer 中删除指定范围的字符串

```
package org.lxh.demo11.stringbufferdemo;
public class StringBufferDemo07 {
```

```
    public static void main(String[] args) {
        StringBuffer buf = new StringBuffer();      // 声明StringBuffer对象
        buf.append("Hello ").append("World!!");     // 向StringBuffer中添加
                                                       内容
        buf.replace(6, 11, "LiXingHua");            // 将World替换为
                                                       LiXingHua
        String str = buf.delete(6, 15).toString(); // 删除指定范围的字符串
        System.out.println("删除之后的内容：" + str); // 输出删除之后的剩余内容
    }
}
```

程序运行结果：

```
删除之后的内容：Hello !!
```

7. 实例操作七：查找指定的内容是否存在

通过 indexOf()方法可以查找指定的内容，如果查找到了，则返回内容的位置，如果没有查找到则返回-1。

范例：查找指定的内容是否存在

```
package org.lxh.demo11.stringbufferdemo;
public class StringBufferDemo08 {
    public static void main(String[] args) {
        StringBuffer buf = new StringBuffer();  // 声明StringBuffer对象
        buf.append("Hello ").append("World!!"); // 向StringBuffer中添加内容
        if(buf.indexOf("Hello")==-1){           // 没有查找到则返回-1
            System.out.println("没有查找到指定的内容") ;
        }else{                                  // 不为-1表示查找到内容
            System.out.println("可以查找到指定的内容") ;
        }
    }
}
```

程序运行结果：

```
可以查找到指定的内容
```

从以上应用中可以发现，大部分形式与 String 类相似。

11.1.2　StringBuffer 类的应用

11.1.1 节为读者演示了 StringBuffer 类的基本用法，StringBuffer 类中的许多用法都与 String 相似，那么 StringBuffer 类到底在哪里使用较多呢？

在讲解 String 类时，曾有以下的代码：

```
package org.lxh.demo11.stringbufferdemo;
public class StringBufferDemo09 {
```

```
    public static void main(String[] args) {
        String str1 = "LiXingHua";
        for (int i = 0; i < 100; i++) {
            str1 += i;                    // 不断修改String的内存引用，性能低
        }
        System.out.println(str1);
    }
}
```

当时曾经提过，以上代码虽然最后字符串的结果改变了，但实际上是通过不断修改对象的引用来实现的，所以性能很差，也就是说要想解决此类问题就必须靠 StringBuffer，因为 StringBuffer 的内容是可以修改的，代码修改如下。

范例：使用 StringBuffer 完成要求

```
package org.lxh.demo11.stringbufferdemo;
public class StringBufferDemo10 {
    public static void main(String[] args) {
        StringBuffer buf = new StringBuffer() ;
        buf.append("LiXingHua");
        for (int i = 0; i < 100; i++) {
            buf.append(i) ;      // StringBuffer可以修改，性能高
        }
        System.out.println(buf);
    }
}
```

以上程序实现的功能与本节中第一个程序是一样的，但是由于使用了 StringBuffer，所以代码的性能有了很大的提高，也就是说对于频繁修改字符串内容的地方，最好使用 StringBuffer 类完成。

11.2 Runtime 类

11.2.1 认识 Runtime 类

在 Java 中 Runtime 类表示运行时操作类，是一个封装了 JVM 进程的类，每一个 JVM 都对应着一个 Runtime 类的实例，此实例由 JVM 运行时为其实例化。所以在 JDK 文档中读者不会发现任何有关 Runtime 类中构造方法的定义，这是因为 Runtime 类本身的构造方法是私有化的（单例设计），如果想取得一个 Runtime 实例，则只能通过以下方式：

```
Runtime run = Runtime.getRuntime() ;
```

也就是说在 Runtime 类中提供了一个静态的 getRuntime()方法，此类可以取得 Runtime 类的实例，那么取得了 Runtime 类的实例有什么用处呢？既然 Runtime 表示的是每一个 JVM

实例，所以就可以通过 Runtime 取得一些系统的信息，方法如表 11-2 所示。

表 11-2　Runtime 类中的方法

序　号	方 法 定 义	类　型	描　述
1	public static Runtime getRuntime()	普通	取得 Runtime 类的实例
2	public long freeMemory()	普通	返回 Java 虚拟机中的空闲内存量
3	public long maxMemory()	普通	返回 JVM 的最大内存量
4	public void gc()	普通	运行垃圾回收器，释放空间
5	public Process exec(String command) throws IOException	普通	执行本机命令

下面通过一些代码来验证以上的功能。

11.2.2　得到 JVM 的内存空间信息

使用 Runtime 类可以取得 JVM 中的内存空间，包括最大内存空间、空闲内存空间等，通过这些信息可以清楚地知道 JVM 的内存使用情况，操作代码如下。

范例：观察 JVM 的内存空间

```
package org.lxh.demo11.runtimedemo;
public class RuntimeDemo01 {
    public static void main(String[] args) {
        Runtime run = Runtime.getRuntime();// 通过Runtime类的静态方法为其进行
                                           实例化操作
        System.out.println("JVM最大内存量：" +
                run.maxMemory());          // 观察最大内存量，根据机器环境会有
                                           所不同
        System.out.println("JVM空闲内存量："
                + run.freeMemory());       // 取得程序运行之前的内存空闲量
        String str = "Hello" + "World" + "!!!"
                + "\t" + "Welcome " + "To"
                + "MLDN" + "~";
        System.out.println(str);
        for (int x = 0; x < 1000; x++) {   // 循环修改String，产生多个垃圾，会
                                           占用内存
            str += x;
        }
        System.out.println("操作String之后的，JVM空闲内存量："
                + run.freeMemory());       // 观察有多个垃圾空间产生之后的内存
                                           空闲量
        run.gc();                          // 进行垃圾收集，释放空间
        System.out.println("垃圾回收之后的，JVM空闲内存量："
                + run.freeMemory());       // 垃圾收集之后的内存空闲量
```

```
        }
    }
```

程序运行结果：

```
JVM最大内存量：66650112
JVM空闲内存量：1838792
HelloWorld!!!   Welcome ToMLDN~
操作String之后的，JVM空闲内存量：1628928
垃圾回收之后的，JVM空闲内存量：1895528
```

以上程序通过 for 循环修改了 String 中的内容，这样的操作必然会产生大量的垃圾，占用系统的内存区域，所以计算后可以发现 JVM 的内存量有所减少，但是当执行 gc()方法进行垃圾收集后，可用的空间就变大了。

11.2.3 Runtime 类与 Process 类

除了观察内存使用量外，也可以直接使用 Runtime 类运行本机的可执行程序，例如，以下程序可以调用本机的记事本程序，记事本程序的执行命令是“notepad.exe”。

范例：调用本机可执行程序

```
package org.lxh.demo11.runtimedemo;
public class RuntimeDemo02 {
    public static void main(String[] args) {
        Runtime run = Runtime.getRuntime();// 通过Runtime类的静态方法为其进行
                                           实例化操作
        try {
            run.exec("notepad.exe") ;          // 调用本机程序，必须进行异常处理
        } catch (Exception e) {
            e.printStackTrace();               // 打印异常信息
        }
    }
}
```

程序执行之后一个记事本程序会直接弹出来，当然也可以对此程序进行进一步的控制，例如，可以在 5 秒之后让记事本关闭，那么此时就需要观察 exec 方法的返回值类，此方法的返回值为 Process，表示一个操作系统的进程类，实际上如果一个记事本程序启动后，肯定会在操作系统中增加一个进程，此过程可以直接通过 Windows 的任务管理器查看，如图 11-2 所示。



图 11-2 任务管理器

也就是说直接控制 Process 可以进行系统进程的控制，如果现在要想通过程序让此进程消失，则可以直接使用 Process 类中的以下方法。

```
public void destroy()
```

范例：让记事本进程运行 5 秒后消失

```
package org.lxh.demo11.runtimedemo;
public class RuntimeDemo03 {
    public static void main(String[] args) {
        Runtime run = Runtime.getRuntime();// 通过Runtime类的静态方法为其进行
                                              实例化操作
        Process pro = null;                // 声明一个Process对象，接收启动的
                                              进程
        try {
            pro = run.exec("notepad.exe"); // 调用本机程序，必须进行异常处理
        } catch (Exception e) {
            e.printStackTrace();           // 打印异常信息
        }
        try {
            Thread.sleep(5000);            // 让此线程存活5秒
        } catch (Exception e) {
            e.printStackTrace();           // 打印异常信息
        }
        pro.destroy();                     // 结束此进程
    }
}
```

以上程序中记事本启动 5 秒之后此进程会自动关闭。

11.3　国际化程序

国际化操作是在开发中较为常见的一种要求，那么什么叫国际化操作呢？实际上国际化的操作就是指一个程序可以同时适应多门语言，即如果现在程序的使用者是中国人，则会以中文为显示文字，如果现在程序的使用者是英国人，则会以英语为显示的文字，也就是说可以通过国际化操作让一个程序适应各个国家的语言要求。

当然，要想实现国际化程序只靠 Locale 类是不够的，还需要属性文件和 ResourceBundle 类的支持，所谓的属性文件是指后缀为.properties 的文件，文件中的内容保存结构为“key=value”形式（关于属性文件的具体操作可以参照 Java 类集部分）。因为国际化的程序只是显示语言的不同，那么就可以根据不同的国家定义不同的属性文件，属性文件中保存真正要使用的文字信息，要访问这些属性文件，可以使用 ResourceBundle 类来完成。

11.3.1　国际化程序的实现思路

例如，现在有一个程序要求可以同时适应法语、英语、中文的显示，那么此时就必须使用国际化，实现的思路如图 11-3 所示。

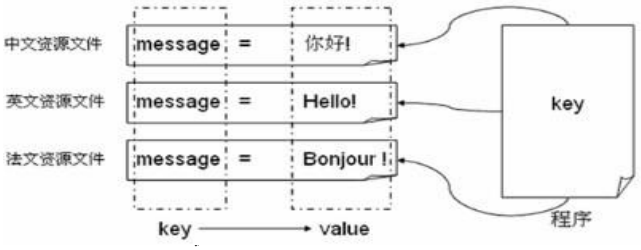


图 11-3　国际化程序实现思路

从图 11-3 中可以非常清楚地发现，可以根据各个不同的国家配置不同的资源文件（资源文件有时也称为属性文件，因为其后缀为.properties），所有的资源文件以“key→value”的形式出现（类似 message=你好！），在程序执行中只是根据 key 找到 value 并将 value 的内容进行显示。也就是说，只要 key 不变，value 的内容可以任意更换。

如果要实现 Java 程序的国际化操作必须通过以下 3 个类完成。

- java.util.Locale：用于表示一个国家语言类。
- java.util.ResourceBundle：用于访问资源文件。
- java.text.MessageFormat：格式化资源文件的占位字符串。

这 3 个类的具体操作流程为通过 Locale 类所指定的区域码，然后 ResourceBundle 根据 Locale 类所指定的区域码找到相应的资源文件，如果资源文件中存在动态文本，则使用 MessageFormat 进行格式化。

提示：资源文件。

资源文件有时也称为属性文件，可以直接使用 Java 类集中提供的 Properties 类进行操作，这一点在随后的章节中会为读者介绍。读者现在需要关心的只是文件的内容及表现形式。

11.3.2　Locale 类

要想实现国际化首先就要认识 Locale 类，因为此类是实现国际化的一个重要类。表 11-3 列出了 Locale 类的构造方法。

表 11-3　Locale 类构造方法

序　号	方 法 定 义	类　型	描　述
1	public Locale(String language)	构造	根据语言代码构造一个语言环境
2	public Locale(String language,String country)	构造	根据语言和国家构造一个语言环境

实际上对于各个国家都有对应的 ISO 编码，例如，中国的编码为 zh-CN、英语-美国的编码为 en-US、法语的编码为 fr-FR。

提示：取得各个国家的 ISO 编码。

对于各个国家的编码，读者实际上没有必要去记住，只需要知道几个常用的就可以了，如果想知道全部的国家编码可以直接搜索 ISO 国家编码。如果觉得麻烦也可以直接在 IE 浏览器中查看各个国家的编码，因为 IE 浏览器可以适应多个国家的语言显示要求，操作步骤为，选择【工具】→【Internet 选项】命令，在打开的对话框中选择【常规】选项卡，单击【语

言】按钮，在打开的对话框中单击【添加】按钮，弹出如图 11-4 所示的对话框。



图 11-4　查找国家编码

11.3.3　ResourceBundle 类

ResourceBundle 类主要作用是读取属性文件，读取属性文件时可以直接指定属性文件的名称（指定名称时不需要文件的后缀），也可以根据 Locale 所指定的区域码来选取指定的资源文件，ResourceBundle 类的常用方法如表 11-4 所示。

表 11-4　ResourceBundle 类的常用方法

序号	方　　法	类型	描　　述
1	public static final ResourceBundle getBundle (String baseName)	普通	取得 ResourceBundle 的实例，并指定要操作的资源文件名称
2	public static final ResourceBundle getBundle (String baseName,Locale locale)	普通	取得 ResourceBundle 的实例，并指定要操作的资源文件名称和区域码
3	public final String getString(String key)	普通	根据 key 从资源文件中取出对应的 value

如果现在要使用 ResourceBunlde 对象，则肯定直接通过 ResourceBundle 类中的静态方法 getBundle()取得。下面通过一个范例简单介绍 ResourceBundle 类的使用。

范例：通过 ResourceBundle 取得资源文件中的内容

（1）定义资源文件：Message.properties

```
info = HELLO
```

（2）从资源文件中取得内容

```
import java.util.ResourceBundle;
public class InterDemo01 {
    public static void main(String[] args) {
        ResourceBundle rb = ResourceBundle.getBundle("Message") ;// 找到资源文件
        System.out.println("内容：" + rb.getString("info")) ;// 从资源文件中取得内容
    }
}
```

程序运行结果：

```
内容：HELLO
```

从以上程序中可以发现，程序通过资源文件中的 key 取得了对应的 value。

11.3.4 Java 国际化程序实现

11.3.3 节中的代码演示了 ResourceBundle 类的作用，下面就在程序中结合 Locale 类一起完成一个国际化的程序开发。

下面完成一个简单的国际化操作，可以根据 Locale 所选择的国家不同，输出不同国家的“你好！”。

- 中文：你好！
- 英语：Hello!
- 法语：Bonjour!

首先根据不同的国家代码建立不同的属性文件，建立的属性文件与生成的*.class 保存在同一个文件夹中。因为现在程序要使用的有 3 种语言，所以要定义 3 种属性文件，在属性文件定义时必须按照“名称_国家代码”的形式命名，即所有相关的属性文件的名称全部一样，只有国家代码不一样，代码如下所示。

（1）中文的属性文件：Message_zh_CN.properties

```
info = \u4f60\u597d\uff01
```

实际上以上的信息就是中文的“你好！”，在属性文件的操作中是不能直接写入中文的，就算是写入了中文，读取出来的也必然是乱码，因此必须将相应的中文变为 Unicode 编码才可以，要成功地将一个中文编码变为 Unicode 编码，可以直接在运行处执行“native2ascii.exe”命令，输入相应的中文之后会自动将其进行编码。

（2）英语的属性文件：Message_en_US.properties

```
info = Hello!
```

（3）法语的属性文件：Message_fr_FR.properties

```
Info = Bonjour!
```

通过 Locale 类和 ResourceBundle 类读取属性文件的内容，代码如下。

```
import java.util.Locale;
import java.util.ResourceBundle;
public class InterDemo02 {
    public static void main(String[] args) {
        Locale zhLoc = new Locale("zh", "CN"); // 表示中国地区
        Locale enLoc = new Locale("en", "US"); // 表示美国地区
        Locale frLoc = new Locale("fr", "FR"); // 表示法国地区
        // 找到中文的属性文件
        ResourceBundle zhrb = ResourceBundle.getBundle("Message",zhLoc) ;
        // 找到英语的属性文件
        ResourceBundle enrb = ResourceBundle.getBundle("Message",enLoc) ;
        // 找到法语的属性文件
```

```
            ResourceBundle frrb = ResourceBundle.getBundle("Message",frLoc) ;
            // 依次读取各个属性文件的内容，通过键值读取，此时的键值名称为“info”
            System.out.println("中文：" + zhrb.getString("info")) ;
            System.out.println("英语：" + enrb.getString("info")) ;
            System.out.println("法语：" + frrb.getString("info")) ;
        }
    }
```

程序运行结果：

```
中文：你好！
英语：Hello!
法语：Bonjour!
```

从以上程序中可以清楚地发现，根据 Locale 所指定的国家不同，读取的资源文件也不同，这样也就实现了国际化程序。

11.3.5　处理动态文本

在以上程序中，所有资源内容都是固定的，但是输出的消息中如果包含了一些动态文本，则必须使用占位符清楚地表示出动态文本的位置，占位符使用“{编号}”的格式出现。使用占位符之后，程序可以直接通过 MessageFormat 对信息进行格式化，为占位符动态设置文本的内容。

MessageFormat 是 Format 类的子类，Format 类主要实现格式化操作，除了 MessageFormat 子类外，Format 还有 NumberFormat、DateFormat 两个子类，这两个子类的使用在本章的后面部分均有讲解，读者先来了解 MessageFormat 类的作用。

提示：关于 Format 子类的说明。

在进行国际化操作时，不光只有文字需要处理，实际上数字的显示、日期的显示都要符合各个区域的要求，读者可以通过“区域和语言选项”对话框观察到这一点，如图 11-5 所示。

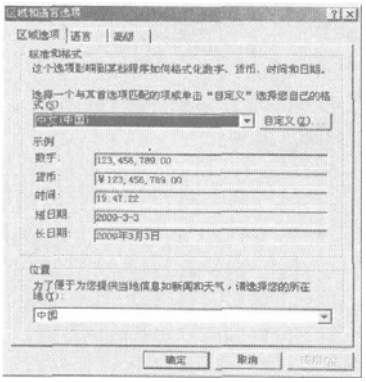


图 11-5　控制面板中的区域和语言选项

在这里要同时改变的有数字、货币、时间等，所以 Format 类提供了 3 个子类，即 MessageFormat、DateFormat、NumberFormat。

例如，现在要输出的信息（以中文为例）是“你好，xxx！”，其中，xxx 的内容是由程序动态设置的，所以，此时修改之前的 3 个属性文件，让其动态地接收程序的 3 个文本。

（1）中文的属性文件：Message_zh_CN.properties

```
info = \u4f60\u597d\uff0c{0}\uff01
```

以上信息就是中文的“你好，{0}！”。

（2）英语的属性文件：Message_en_US.properties

```
info = Hello,{0}!
```

（3）法语的属性文件：Message_fr_FR.properties

```
info = Bonjour,{0}!
```

在以上 3 个属性文件中，都加入了“{0}”，表示一个占位符，如果有更多的占位符，则直接在后面继续加上“{1}”、“{2}”即可。

然后继续使用之前的 Locale 类和 ResourceBundle 类读取资源文件的内容，但是读取之后的文件因为要处理占位符的内容，所以要使用 MessageFormat 类进行处理，主要使用以下方法：

```
public static String format(String pattern,Object…arguments)
```

以上方法中第 1 个参数表示要匹配的字符串，第 2 个参数“Object…arguments”表示输入参数可以任意多个，没有个数的限制。

范例：使用 MessageFormat 格式化动态文本

```
import java.text.MessageFormat;
import java.util.Locale;
import java.util.ResourceBundle;
public class InterDemo03 {
    public static void main(String[] args) {
        Locale zhLoc = new Locale("zh", "CN"); // 表示中国地区
        Locale enLoc = new Locale("en", "US"); // 表示美国地区
        Locale frLoc = new Locale("fr", "FR"); // 表示法国地区
        // 找到中文的属性文件
        ResourceBundle zhrb =ResourceBundle.getBundle("Message",zhLoc) ;
        // 找到英语的属性文件
        ResourceBundle enrb =ResourceBundle.getBundle("Message",enLoc) ;
        // 找到法语的属性文件
        ResourceBundle frrb =ResourceBundle.getBundle("Message",frLoc) ;
        // 依次读取各个属性文件的内容，通过键值读取，此时的键值名称为“info”
        String str1 = zhrb.getString("info") ;
        String str2 = enrb.getString("info") ;
        String str3 = frrb.getString("info") ;
        System.out.println("中文：" + MessageFormat.format(str1,"李兴华")) ;
        System.out.println("英语：" +MessageFormat.format(str2,"LiXingHua")) ;
```

```
            System.out.println("法语:" + MessageFormat.format(str3,"LiXingHua"));
        }
    }
```

程序运行结果：

```
中文：你好，李兴华!
英语：Hello,LiXingHua!
法语：Bonjour,LiXingHua!
```

以上程序代码通过 MessageFormat.format()方法设置动态文本的内容，如果有更多的占位符则可按照图 11-6 所示设置多个参数。

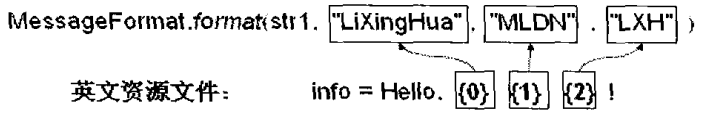


图 11-6　为多个占位符设置多个动态文本

提示：Java 新特性——可变参数传递中可以接收多个对象。

在方法传递参数时可以使用如下的形式：

返回值类型 方法名称(Object…args)

表示方法可以接收任意多个参数，然后按照数组的方式输出即可，代码如下所示。

范例：测试参数传递

```
public class InterDemo04 {
    public static void main(String[] args) {
        System.out.print("第一次运行：");
        fun("LXH","Li","李兴华") ;                    // 传入3个参数
        System.out.print("\n第二次运行：");
        fun("MLDN") ;                                 // 传入1个参数
    }
    public static void fun(Object...args){ // 可以输入任意多个数据，使用数组
                                                      表示
        for(int i=0;i<args.length;i++){
            System.out.print(args[i] + "、");
        }
    }
}
```

程序运行结果：

```
第一次运行：LXH、Li、李兴华、
第二次运行：MLDN、
```

从程序的运行结果中可以发现，无论在方法调用时输入多少个参数，方法都可以接收，接收参数之后可以直接以数组的形式对参数进行输出。

在其他的资料中读者也可能看到使用数组的方式向可变参数传递内容，代码如下所示。

范例： 使用数组传递参数

```
package org.lxh.demo11.interdemo;
public class InterDemo05 {
    public static void main(String[] args) {
        System.out.print("第一次运行：");
        Object[] arg1 = {"LXH","Li","李兴华"} ; // 对象数组，保存所有具体内容
        fun(arg1) ;                              // 传入3个参数
        System.out.print("\n第二次运行：");
        Object[] arg2 = {"MLDN"} ;
        fun(arg2) ;                              // 传入1个参数
        System.out.print("\n第三次运行：");
        Object[] arg3 = {} ;
        fun(arg3) ;                              // 没有参数传入
    }
    public static void fun(Object...args){       // 可变参数
        for(int i=0;i<args.length;i++){
            System.out.print(args[i] + "、");
        }
    }
}
```

程序运行结果：

```
第一次运行：LXH、Li、李兴华、
第二次运行：MLDN、
第三次运行：
```

从运行结果来看，两种方式是一样的，没有任何的区别，读者可以根据个人喜好来选择，并没有强制性的规定。

11.3.6 使用类代替资源文件

可以使用属性文件保存所有的资源信息，当然，在 Java 中也可以使用类来保存所有的资源信息，但是在开发中此种做法并不多见，主要还是以属性文件的应用为主。

与之前的资源文件一样，如果使用类保存信息，则也必须按照 key-value 的形式出现，而且类的命名必须与属性文件一致。而且此类必须继承 ListResourceBundle 类，继承之后要覆写此类中的 getContent()方法。下面以中文资源文件为例进行讲解。

范例： 建立中文的资源类

```
import java.util.ListResourceBundle;
public class Message_zh_CN extends ListResourceBundle{
    private final Object data[][] = {
            {"info","你好，{0}"}
    } ;
```

```
    public Object[][] getContents() {   // 覆写方法
        return data;
    }
}
```

范例：在国际化中使用以上定义的类

```
import java.text.MessageFormat;
import java.util.Locale;
import java.util.ResourceBundle;
public class InterDemo06 {
    public static void main(String[] args) {
        Locale zhLoc = new Locale("zh", "CN"); // 表示中国地区
        // 找到中文的属性文件
        ResourceBundle zhrb = ResourceBundle.getBundle("org.lxh.demo11.
        Message",zhLoc) ;
        // 依次读取各个属性文件的内容，通过键值读取，此时的键值名称为“info”
        String str1 = zhrb.getString("info") ;
        System.out.println("中文：" + MessageFormat.format(str1,"李兴华")) ;
    }
}
```

程序运行结果：

```
中文：你好，李兴华！
```

可以发现，此程序使用一个代替了原本的资源文件内容，但是，读者一定要注意的是，在资源类中的属性一定是一个二维数组。

提问：如果现在多种资源文件一起出现，该如何访问？

之前讲解的程序中出现了 Message.properties、Message_zh_CN.properties、Message_zh_CN.class，如果在一个项目中同时存在这 3 个类型的文件，那最终使用的是哪一个？

回答：只会使用一个，按照优先级使用。顺序为 Message_zh_CN.class、Message_zh_ CN.properties、Message.properties。

但是从实际开发的角度来看，使用一个类文件来代替资源文件的方式是很少见的，所以，重点要掌握资源文件的使用。

11.4　System 类

11.4.1　认识 System 类

System 类可能是读者最经常看见的类，之前的系统输出语句“System.out.println()”相信读者都应该不会陌生，那么 System 类到底是什么呢？实际上 System 类是一些与系统相关

属性和方法的集合，而且在 System 类中所有的属性都是静态的，要想引用这些属性和方法，直接使用 System 类调用即可。表 11-5 中列出了 System 类的一些常用方法。

表 11-5 System 类的常用方法

序号	方 法 定 义	类型	描 述
1	public static void exit(int status)	普通	系统退出，如果 status 为非 0 就表示退出
2	public static void gc()	普通	运行垃圾收集机制，调用的是 Runtime 类中的 gc 方法
3	public static long currentTimeMillis()	普通	返回以毫秒为单位的当前时间
4	public static void arraycopy(Object src,int srcPos,Object dest,int destPos,int length)	普通	数组复制操作
5	public static Properties getProperties()	普通	取得当前系统的全部属性
6	public static String getProperty(String key)	普通	根据键值取得属性的具体内容

系统退出的方法和数组复制方法前面已经介绍过。下面介绍其他方法的使用。

提示：System 类中的方法都是静态的。

System 类中的方法都是使用 static 定义的，所以在使用时直接使用类名称就可以调用，如 System.gc()。

范例：计算一个程序的执行时间

```
package org.lxh.demo11.systemdemo;
public class SystemDemo01 {
    public static void main(String[] args) {
        long startTime = System.currentTimeMillis();// 取得开始计算之前的时间
        int sum = 0;                                // 声明变量存放累加后的结果
        for (int i = 0; i < 30000000; i++) {        // 执行累加操作
            sum += i;
        }
        long endTime = System.currentTimeMillis();
        // 结束时间减去开始时间
        System.out.println("计算所花费的时间：" + (endTime - startTime)+"毫秒");
    }
}
```

程序运行结果：

```
计算所花费的时间：120毫秒
```

也可以直接通过 System 类取得本机的全部环境属性。

范例：取得本机的全部环境属性

```
package org.lxh.demo11.systemdemo;
public class SystemDemo02 {
```

```
    public static void main(String[] args) {
        System.getProperties().list(System.out) ;        // 列出系统的全部属性
    }
}
```

程序运行结果（部分）：

```
-- listing properties --
java.runtime.name=Java(TM) 2 Runtime Environment, Stand...
sun.boot.library.path=D:\Java\jre1.5.0_01\bin
java.vm.version=1.5.0_01-b08
path.separator=;
java.vm.name=Java HotSpot(TM) Client VM
file.encoding.pkg=sun.io
user.country=CN
user.dir=F:\test
java.runtime.version=1.5.0_01-b08
java.awt.graphicsenv=sun.awt.Win32GraphicsEnvironment
java.endorsed.dirs=D:\Java\jre1.5.0_01\lib\endorsed
os.arch=x86
java.io.tmpdir=C:\DOCUME~1\李兴华\LOCALS~1\Temp\
line.separator=
java.vm.specification.vendor=Sun Microsystems Inc.
user.variant=
os.name=Windows XP
sun.jnu.encoding=GBK
java.library.path=D:\Java\jre1.5.0_01\bin;.;C:\WINDOWS\...
java.specification.name=Java Platform API Specification
java.class.version=49.0
sun.management.compiler=HotSpot Client Compiler
os.version=5.1
user.home=C:\Documents and Settings\李兴华
user.timezone=
java.awt.printerjob=sun.awt.windows.WPrinterJob
file.encoding=GBK
user.name=李兴华
java.class.path=F:\test
java.vm.specification.version=1.0
sun.arch.data.model=32
java.home=D:\Java\jre1.5.0_01
user.language=zh
java.version=1.5.0_01
java.ext.dirs=D:\Java\jre1.5.0_01\lib\ext
file.separator=\
...
```

对于以上程序读者现在只需要了解即可，至于为什么要使用 System.out 以及之前的全部属性信息是怎样列出的。本书将在 IO 操作和类集部分为读者进行详细的解释。

提示：部分属性说明。

以上程序中列出了系统中与 Java 相关的各个属性，在属性中有以下两点需要关注。

- 文件默认编码：file.encoding=GBK。
- 文件分割符：file.separator=\。

以上程序列出了当前系统中的一些与 Java 有关的属性，当然也可以只列出指定的属性。

范例：列出指定属性

```
package org.lxh.demo11.systemdemo;
public class SystemDemo03 {
    public static void main(String[] args) {
        System.out.println("系统版本为:" + System.getProperty("os.name")
                + System.getProperty("os.version")
                + System.getProperty("os.arch"));  // 获取当前系统版本
        System.out.println("系统用户为:" + System.getProperty("user.name"));
        System.out.println("当前用户目录:" + System.getProperty("user.home"));
        System.out.println("当前用户工作目录:" + System.getProperty("user.
        dir"));
    }
}
```

程序运行结果：

```
系统版本为:Windows XP5.1x86
系统用户为：李兴华
当前用户目录:C:\Documents and Settings\李兴华
当前用户工作目录:F:\test
```

11.4.2 垃圾对象的回收

Java 中有垃圾的自动收集机制，可以不定期地释放 Java 中的垃圾空间，而且在前面讲解 Runtime 类时也已经了解了如何进行垃圾空间的释放。在 System 类中也有一个 gc()方法，此方法也可以进行垃圾的收集，而且此方法实际上是对 Runtime 类中的 gc()方法的封装，功能与其类似。

但在此处讲解的是如何对一个对象进行回收，一个对象如果不再被任何栈内存所引用，那么此对象就可以称为垃圾对象，等待被回收。实际上，等待的时间是不确定的，所以可以直接调用 System.gc()方法进行垃圾的回收。

提示：开发中的垃圾收集基本上都是由系统自动完成的。

在实际的开发中，垃圾内存的释放基本上都是由系统自动完成的，除非特殊的情况，一般都很少直接去调用 gc()方法。

但是如果在一个对象被回收之前要进行某些操作该怎么办呢？实际上在 Object 类中有一个 finalize()方法，此方法定义如下：

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

一个子类只需要覆写此方法即可在释放对象前进行某些操作。

范例：观察对象释放

```
package org.lxh.demo11.systemdemo;
class Person{
    private String name ;
    private int age ;
    public Person(String name,int age){
        this.name = name ;
        this.age = age ;
    }
    public String toString(){
        return "姓名：" + this.name + "，年龄：" + this.age ;
    }
    public void finalize() throws Throwable {          // 对象释放空间时默认
                                                          调用此方法
        System.out.println("对象被释放 --> " + this) ;  // 直接打印对象调用
                                                          toString
    }
}
public class SystemDemo04 {
    public static void main(String[] args) {
        Person per = new Person("张三",30) ;
        per = null ;                                   // 断开引用，释放空间
        System.gc() ;                                  // 强制性释放空间
    }
}
```

程序运行结果：

```
对象被释放 --> 姓名：张三，年龄：30
```

在以上程序中强制调用了释放空间的方法，而且在对象被释放前调用了 finalize()方法。如果在 finalize()方法中出现了异常，则程序并不会受其影响，会继续执行。

提示：finalize()方法抛出的是 Throwable 异常。

在 finalize()方法上可以发现抛出的异常并不是常见的 Exception，而是使用了 Throwable 进行抛出的异常，所以在调用此方法时不一定只会在程序运行中产生错误，也有可能产生 JVM 错误。

11.4.3 对象的生命周期

下面介绍对象的生命周期，如图 11-7 所示。

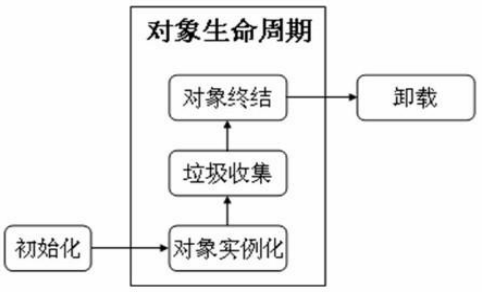


图 11-7 对象的生命周期

一个类加载后进行初始化，然后就可以进行对象的实例化，对象实例化时会调用构造方法完成，当一个对象不再使用时就要等待被垃圾收集，然后对象终结，最终被程序卸载。

提示：对象的生命周期与人的生命周期是一样的。

对象的生命周期实际上与人的生命周期是一样的，当一个人在母体中孕育生命时，实际上就是初始化的操作，是由 JVM 自动进行的，但是此时并不能立刻使用；当这个人出生时实际上就是对象的实例化操作；人出生之后可以进行很多的社会活动，那么这就相当于是使用对象调用了一系列的操作方法；当一个人工作了一辈子之后就要退休了，要把这个职位让给其他的人，实际上就属于垃圾收集的工作，释放空间给其他对象使用；当人死亡之后，实际上就是对象的终结；但是人死亡之后并不会关心后续的一些事情，至于这个人的葬礼等一系列的事情也是由其他人去做的，那么这就是卸载，也将由 JVM 进行自动处理。

11.5 日期操作类

在程序的开发中经常会遇到日期类型的操作，在 Java 中对于日期的操作也提供了良好的支持，主要使用 java.util 包中的 Date、Calendar 以及 java.text 包中的 SimpleDateFormat。下面介绍其具体的用法。

11.5.1 Date 类

Date 类是一个较为简单的操作类，在使用中直接使用 java.util.Date 类的构造方法并进行输出就可以得到一个完整的日期，构造方法定义如下：

```
public Date()
```

范例：得到当前系统日期

```
package org.lxh.demo11.datedemo;
import java.util.Date;
public class DateDemo01 {
```

```
    public static void main(String[] args) {
        Date date = new Date() ;                    // 实例化Date类对象
        System.out.println("当前日期为: "+date);      // 输出日期
    }
}
```

程序运行结果：

```
当前日期为: Sun Oct 19 09:43:32 CST 2008
```

从程序的运行结果看，已经得到了系统的当前日期，但是对于此日期可以发现格式并不是很符合平常看到的格式，而且现在的时间也不能准确地精确到毫秒，要想按照用户自己的格式显示时间可以使用 Calendar 类完成操作。

提示：Date 类使用较多。

虽然以上程序中，使用 Date 显示的日期格式并不符合用户的要求，但是从实际的角度来看 Date 类的使用频率也是很高的，因为只要使用格式化的方式就可以直接将一个日期格式按照用户所需要的方式进行显示，这一点在随后的部分可以看到。

11.5.2 Calendar 类

Calendar 类可以将取得的时间精确到毫秒。但是，这个类本身是一个抽象类，从之前学习到的知识可以知道，如果要想使用一个抽象类，则必须依靠对象的多态性，通过子类进行父类的实例化操作，Calendar 的子类是 GregorianCalendar 类。在 Calendar 中提供了表 11-6 所示的部分常量，分别表示日期的各个数字。

表 11-6　Calendar 类中的常量

序　号	常　量	类　型	描　述
1	public static final int YEAR	int	取得年
2	public static final int MONTH	int	取得月
3	public static final int DAY_OF_MONTH	int	取得日
4	public static final int HOUR_OF_DAY	int	取得小时，24 小时制
5	public static final int MINUTE	int	取得分
6	public static final int SECOND	int	取得秒
7	public static final int MILLISECOND	int	取得毫秒

除了以上提供的全局常量外，Calendar 还提供了一些常用方法，如表 11-7 所示。

表 11-7　Calendar 类中提供的方法

序　号	方　法	类　型	描　述
1	public static Calendar getInstance()	普通	根据默认的时区实例化对象
2	public boolean after(Object when)	普通	判断一个日期是否在指定日期之后
3	public boolean before(Object when)	普通	判断一个日期是否在指定日期之前
4	public int get(int field)	普通	返回给定日历字段的值

下面通过以上方法取得一个系统的当前日期。

范例：取得系统的当前日期

```
package org.lxh.demo11.datedemo;
import java.util.Calendar;
import java.util.GregorianCalendar;
public class DateDemo02 {
    public static void main(String[] args) {
        Calendar calendar = null;                    // 声明一个Calendar对象
        calendar = new GregorianCalendar();          // 通过子类为其实例化
        System.out.println("年：" + calendar.get(Calendar.YEAR));  // 年
        System.out.println("月：" + (calendar.get(Calendar.MONTH) + 1));
                                                     // 月
        System.out.println("日：" + calendar.get(Calendar.DAY_OF_MONTH));
                                                     // 日
        System.out.println("时：" + calendar.get(Calendar.HOUR_OF_DAY));
                                                     // 时
        System.out.println("分：" + calendar.get(Calendar.MINUTE));
                                                     // 分
        System.out.println("秒：" + calendar.get(Calendar.SECOND));
                                                     // 秒
        System.out.println("毫秒：" + calendar.get(Calendar.MILLISECOND));
                                                     // 秒
    }
}
```

程序运行结果：

```
年：2008
月：10
日：19
时：10
分：8
秒：3
毫秒：280
```

以上程序通过 GregorianCalendar 子类实例化 Calendar 类，然后通过 Calendar 类中的各种常量及方法取得系统的当前时间。

但是，如果按照以上的方式取得，代码会比较复杂，所以在 Java 中又提供了其他日期的类。

11.5.3 DateFormat 类

在 java.util.Date 类中实际上取得的时间是一个非常正确的时间。但是因为其显示的格式

不理想，所以无法符合中国人的习惯要求，那么实际上此时就可以为此类进行格式化操作，变为符合于中国人习惯的日期格式。

另外，要提醒读者的是 DateFormat 类与 MessageFormat 类都属于 Format 类的子类，专门用于格式化数据使用。DateFormat 类的定义如下：

```
public abstract class DateFormat
extends Format
```

从定义上看 DateFormat 类是一个抽象类，所以肯定无法直接实例化，但是在此抽象类中提供了一个静态方法，可以直接取得本类的实例，此类的常用方法如表 11-8 所示。

表 11-8　DateFormat 类的常用方法

序　号	方　法	类　型	描　述
1	public static final DateFormat getDateInstance()	普通	得到默认的对象
2	public static final DateFormat getDateInstance(int style, Locale aLocale)	普通	根据 Locale 得到对象
3	public static final DateFormat getDateTimeInstance()	普通	得到日期时间对象
4	public static final DateFormat getDateTimeInstance(int dateStyle,int timeStyle,Locale aLocale)	普通	根据 Locale 得到日期时间对象

以上 4 个方法可以构造 DateFormat 类的对象，但是发现以上方法中需要传递若干个参数，这些参数表示日期地域或日期的显示形式。下面通过几段代码来了解 DateFormat 类的操作。

范例：观察 DateFormat 中的默认操作

```
package org.lxh.demo11.datedemo;
import java.text.DateFormat;
import java.util.Date;
public class DateDemo03 {
    public static void main(String[] args) {
        DateFormat df1 = null;                          // 声明DateFormat类对象
        DateFormat df2 = null;                          // 声明DateFormat类对象
        df1 = DateFormat.getDateInstance();             // 取得日期
        df2 = DateFormat.getDateTimeInstance();         // 取得日期时间
        System.out.println("DATE: " + df1.format(new Date()));// 格式化日期
        System.out.println("DATETIME: " + df2.format(new Date()));
    }
}
```

程序运行结果：

```
DATE: 2008-12-2
DATETIME: 2008-12-2 16:25:11
```

从程序的运行结果中发现，第 2 个 DATETIME 显示了时间，但还不是比较合理的中文显示格式。如果想取得更加合理的时间，则必须在构造 DateFormat 对象时传递若干个参数。

范例：指定显示的风格

```
package org.lxh.demo11.datedemo;
import java.text.DateFormat;
import java.util.Date;
import java.util.Locale;
public class DateDemo04 {
    public static void main(String[] args) {
        DateFormat df1 = null;                          // 声明DateFormat对象
        DateFormat df2 = null;                          // 声明DateFormat对象
        df1 = DateFormat.getDateInstance(DateFormat.YEAR_FIELD, new Locale(
                "zh", "CN"));                           // 取得日期，并设置日期显示风格
        // 取得日期时间，设置日期的显示格式、时间的显示格式
        df2 = DateFormat.getDateTimeInstance(DateFormat.YEAR_FIELD,
                DateFormat.ERA_FIELD, new Locale("zh", "CN"));
        System.out.println("DATE: " + df1.format(new Date()));
        System.out.println("DATETIME: " + df2.format(new Date()));
    }
}
```

程序运行结果：

```
DATE: 2008年12月2日
DATETIME: 2008年12月2日 下午04时30分29秒 CST
```

可以发现，以上的日期时间已经被格式化，格式是其默认的时间显示格式，但是如果现在要想得到用户自己需要的日期显示格式，则必须通过 DateFormat 的子类 SimpleDateFormat 类完成。

11.5.4 SimpleDateFormat 类

在项目开发中，会经常将一个日期格式转换为另外一种日期格式，例如，原日期为 2008-10-19 10:11:30.345，转换后日期为 2008 年 10 月 19 日 10 时 11 分 30 秒 345 毫秒。从这两个日期可以发现，日期的数字完全一样，只是日期格式有所不同，要想实现转换就必须使用 java.text 包中的 SimpleDateFormat 类完成。

首先必须先定义出一个完整的日期转化模板，在模板中通过特定的日期标记可以将一个日期格式中的日期数字提取出来，日期格式化模板标记如表 11-9 所示。

表 11-9 日期格式化模板标记

序 号	标 记	描 述
1	y	年，年份是 4 位数字，所以需要使用 yyyy 表示
2	M	年中的月份，月份是两位数字，所以需要使用 MM 表示
3	d	月中的天数，天数是两位数字，所以需要使用 dd 表示
4	H	一天中的小时数（24 小时），小时是两位数字，使用 HH 表示

续表

序　号	标　记	描　述
5	m	小时中的分钟数，分钟是两位数字，使用 mm 表示
6	s	分钟中的秒数，秒是两位数字，使用 ss 表示
7	S	毫秒数，毫秒数是 3 位数字，使用 SSS 表示

此外，还需要 SimpleDateFormat 类中的方法才可以完成，SimpleDateFormat 类的常用方法如表 11-10 所示。

表 11-10　SimpleDateFormat 类中的常用方法

序　号	方　法	类　型	描　述
1	public SimpleDateFormat(String pattern)	构造	通过一个指定的模板构造对象
2	public Date parse(String source) throws ParseException	普通	将一个包含日期的字符串变为 Date 类型
3	public final String format(Date date)	普通	将一个 Date 类型按照指定格式变为 String 类型

范例：格式化日期操作

```
package org.lxh.demo11.datedemo;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
public class DateDemo05 {
    public static void main(String[] args){
        String strDate = "2008-10-19 10:11:30.345";  // 定义日期时间的字符串
        // 准备第1个模板，从字符串中提取日期数字
        String pat1 = "yyyy-MM-dd HH:mm:ss.SSS";
        // 准备第2个模板，将提取后的日期数字变为指定的格式
        String pat2 = "yyyy年MM月dd日HH时mm分ss秒SSS毫秒";
        SimpleDateFormat sdf1 = new SimpleDateFormat(pat1);// 实例化模板对象
        SimpleDateFormat sdf2 = new SimpleDateFormat(pat2);// 实例化模板对象
        Date d = null;
        try {
            d = sdf1.parse(strDate);          // 将给定字符串中的日期提取出来
        } catch (ParseException e) {          // 如果提供的字符串格式有错误，则进
                                              行异常处理
            e.printStackTrace();
        }
        System.out.println(sdf2.format(d));// 将日期变为新的格式
    }
}
```

程序运行结果：

```
2008年10月19日10时11分30秒345毫秒
```

以上程序中，首先使用第 1 个模板将字符串中表示的日期数字取出，然后再使用第 2 个模板将这些日期数字重新转化为新的格式表示。

提示：SimpleDateFormat 类经常用于将 String 变为 Date 型数据。

在实际的开发中，用户所输入的各个数据都是以 String 的方式进行接收的，所以此时为了可以正确地将 String 变为 Date 型数据，可以依靠 SimpleDateFormat 类完成。

范例： 将 String 型数据变为 Date 型数据

```
package org.lxh.demo11.datedemo;
import java.text.SimpleDateFormat;
import java.util.Date;
public class StringDateDemo {
    public static void main(String[] args) throws Exception {  // 所有异常
                                                                  抛出
        String strDate = "2008-10-19 10:11:30.345";  // 定义日期时间的字符串
        String pat = "yyyy-MM-dd HH:mm:ss.SSS";      // 定义模板
        SimpleDateFormat sdf = new SimpleDateFormat(pat); // 实例化模板对象
        Date d = sdf.parse(strDate);                 // 将给定字符串中的日期
                                                        提取出来
        System.out.println(d) ;                      // 输出Date对象
    }
}
```

程序运行结果：

```
Sun Oct 19 10:11:30 CST 2008
```

以上程序已经实现了 String 向 Date 的转型操作，这样的操作在讲解 Java 数据库编程时将会经常使用。

11.5.5 实例操作——取得完整日期

1. 实现一：基于 Calendar 类

以上已经为读者清楚地介绍了如何取得一个系统的日期以及对日期格式的转化操作，但是如果所有的程序都按 Calendar 程序那样取得日期，则代码肯定会重复很多次，所以可以考虑单独去设计一个类，此类可以取得当前系统的日期时间。但是在使用 Calendar 类得到时间时有一个问题，例如，现在的月份是 9 月，正确的日期格式应该是 09，但是因为得到日期的方法返回的是 int 类型，所以最终的返回结果是 9，也就是说要想达到正确的显示效果，则必须在 9 前补一个 0。

范例： 设计一个取得日期的类，此类可以取得系统的当前时间和时间戳

```
package org.lxh.demo11.datedemo;
import java.util.Calendar;
import java.util.GregorianCalendar;
```

```
class DateTime {
    private Calendar calendar = null;          // 定义一个Calendar对象，可以
                                                  取得时间
    public DateTime() {
        this.calendar = new GregorianCalendar();// 通过Calendar类的子类实例化
    }
    public String getDate() {   // 得到完整的日期，格式为：yyyy-MM-dd HH:mm:ss.
                                   SSS
        // 考虑到程序要频繁修改字符串，所以使用StringBuffer提升性能
        StringBuffer buf = new StringBuffer();
        // 依次取得时间
        buf.append(calendar.get(Calendar.YEAR)).append("-");
        buf.append(this.addZero(calendar.get(Calendar.MONTH) + 1, 2));
        buf.append("-");
        buf.append(this.addZero(calendar.get(Calendar.DAY_OF_MONTH), 2));
        buf.append(" ");
        buf.append(this.addZero(calendar.get(Calendar.HOUR_OF_DAY), 2));
        buf.append(":");
        buf.append(this.addZero(calendar.get(Calendar.MINUTE), 2));
        buf.append(":");
        buf.append(this.addZero(calendar.get(Calendar.SECOND), 2));
        buf.append(".");
        buf.append(this.addZero(calendar.get(Calendar.MILLISECOND), 3));
        return buf.toString();
    }
    // 得到完整的日期，格式为：yyyy年MM月dd日HH时mm分ss秒SSS毫秒
    public String getDateComplete() {
        StringBuffer buf = new StringBuffer();
        buf.append(calendar.get(Calendar.YEAR)).append("年");
        buf.append(this.addZero(calendar.get(Calendar.MONTH) + 1, 2));
        buf.append("月");
        buf.append(this.addZero(calendar.get(Calendar.DAY_OF_MONTH), 2));
        buf.append("日 ");
        buf.append(this.addZero(calendar.get(Calendar.HOUR_OF_DAY), 2));
        buf.append("时");
        buf.append(this.addZero(calendar.get(Calendar.MINUTE), 2));
        buf.append("分");
        buf.append(this.addZero(calendar.get(Calendar.SECOND), 2));
        buf.append("秒");
        buf.append(this.addZero(calendar.get(Calendar.MILLISECOND), 3));
        buf.append("毫秒");
        return buf.toString();
```

```
    }
    // 考虑到日期中有前导0，所以在此处加上了补零的方法
    private String addZero(int num, int len) {
        StringBuffer s = new StringBuffer();
        s.append(num);
        while (s.length() < len) {          // 如果长度不足，则继续补0
            s.insert(0, "0");               // 在第1个位置处补0
        }
        return s.toString();
    }
    public String getTimeStamp() {          // 得到时间戳：yyyyMMddHHmmssSSS
        StringBuffer buf = new StringBuffer();
        buf.append(calendar.get(Calendar.YEAR));
        buf.append(this.addZero(calendar.get(Calendar.MONTH) + 1, 2));
        buf.append(this.addZero(calendar.get(Calendar.DAY_OF_MONTH), 2));
        buf.append(this.addZero(calendar.get(Calendar.HOUR_OF_DAY), 2));
        buf.append(this.addZero(calendar.get(Calendar.MINUTE), 2));
        buf.append(this.addZero(calendar.get(Calendar.SECOND), 2));
        buf.append(this.addZero(calendar.get(Calendar.MILLISECOND), 3));
        return buf.toString();
    }
};
public class DateDemo06 {
    public static void main(String[] args) {
        DateTime dt = new DateTime();       // 实例化DateTime对象
        System.out.println("系统日期：" + dt.getDate());
        System.out.println("中文日期：：" + dt.getDateComplete());
        System.out.println("时间戳：" + dt.getTimeStamp());
    }
}
```

程序运行结果：

```
系统日期：2008-10-19 10:42:50.562
中文日期：：2008年10月19日 10时42分50秒562毫秒
时间戳：20081019104250562
```

以上程序已经完成了设计的要求，但程序代码太长、太复杂，那么有没有更好的方式呢？实际上是有的，SimpleDateFormat 类中可以对一个 Date 类型进行格式修改的方法，直接使用此方法就可以将一个当前日期的 Date 类型变为指定的日期格式。

2．实现二：基于 SimpleDateFormat 类

使用 SimpleDateFormat 类可以方便地把一个日期变为指定格式，所以直接使用此类操作是最简单、最合适的。

范例：修改之前的程序

```
package org.lxh.demo11.datedemo;
import java.text.SimpleDateFormat;
import java.util.Date;
class DateTime{
    // 声明日期格式化操作对象，直接对new Date()进行实例化
    private SimpleDateFormat sdf = null ;
    // 得到完整的日期，格式为：yyyy-MM-dd HH:mm:ss.SSS
    public String getDate(){
        this.sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS");
        return this.sdf.format(new Date()) ;
    }
    // 得到完整的日期，格式为：yyyy年MM月dd日HH时mm分ss秒SSS毫秒
    public String getDateComplete(){
        this.sdf = new SimpleDateFormat("yyyy年MM月dd日 HH时mm分ss秒SSS毫秒");
        return this.sdf.format(new Date()) ;
    }
    public String getTimeStamp(){   // 得到时间戳：yyyyMMddHHmmssSSS
        this.sdf = new SimpleDateFormat("yyyyMMddHHmmssSSS");
        return this.sdf.format(new Date()) ;
    }
};
public class DateDemo07 {
    public static void main(String[] args) {
        DateTime dt = new DateTime(); // 实例化DateTime对象
        System.out.println("系统日期：" + dt.getDate());
        System.out.println("中文日期：：" + dt.getDateComplete());
        System.out.println("时间戳：" + dt.getTimeStamp());
    }
}
```

程序运行结果：

```
系统日期：2008-10-19 10:47:48.610
中文日期：：2008年10月19日 10时47分48秒610毫秒
时间戳：20081019104748610
```

以上程序完成了与 11.5.1 节程序相同的功能，而且代码更为简单，所以读者日后在开发代码时一定要多多设计，灵活地运用各种方法，使程序的开发更为便捷。

11.6　Math 类

Math 类是数学操作类，提供了一系列的数学操作方法，包括求绝对值、三角函数等，在 Math 类中提供的一切方法都是静态方法，所以直接由类名称调用即可，下面简单介绍

Math 类的基本操作。

范例： Math 类的基本操作

```
package org.lxh.demo11.mathdemo;
public class MathDemo01 {
    public static void main(String[] args) {
        // Math类中的方法都是静态方法，直接使用“类名称.方法名称()”的形式调用即可
        System.out.println("求平方根：" + Math.sqrt(9.0));
        System.out.println("求两数的最大值：" + Math.max(10,30));
        System.out.println("求两数的最小值：" + Math.min(10,30));
        System.out.println("2的3次方：" + Math.pow(2,3));
        System.out.println("四舍五入：" + Math.round(33.6));
    }
}
```

程序运行结果：

```
求平方根：3.0
求两数的最大值：30
求两数的最小值：10
2的3次方：8.0
四舍五入：34
```

在上面的操作中，Math 类中 round()方法的主要作用是进行四舍五入操作，但是此方法在操作时将小数点后面的全部数字都忽略掉，如果想精确到小数点后的准确位数，则必须使用 BigDecimal 类完成。

11.7 Random 类

Random 类是随机数产生类，可以指定一个随机数的范围，然后任意产生在此范围中的数字。Random 类中的常用方法如表 11-11 所示。

表 11-11 Random 类中的常用方法

序　号	方　法	类　型	描　述
1	public boolean nextBoolean()	普通	随机生成 boolean 值
2	public double nextDouble()	普通	随机生成 double 值
3	public float nextFloat()	普通	随机生成 float 值
4	public int nextInt()	普通	随机生成 int 值
5	public int nextInt(int n)	普通	随机生成给定最大值的 int 值
6	public long nextLong()	普通	随机生成 long 值

范例： 生成 10 个随机数字，且数字不大于 100

```
package org.lxh.demo11.randomdemo;
import java.util.Random;
```

```
public class RandomDemo01 {
    public static void main(String args[]){
        Random r = new Random() ;                    // 定义Random对象
        for(int i=0;i<10;i++){                       // 输出10个随机数
            System.out.print(r.nextInt(100)+"\t") ;
        }
    }
}
```

程序运行结果（可能的结果）：

```
16  95  18  71  57  86  37  67  80  73
```

以上程序中使用 Random 类，并通过 for 循环生成了 10 个不大于 100 的随机数。

提示：Random 类在实际生活中的应用。

读者应该买过 36 选 7 的彩票，如果要实现自动选号，就可以通过 Random 完成，有兴趣的读者不妨自己编写一个这样的程序。

11.8　NumberFormat 类

11.8.1　NumberFormat 类的基本使用

NumberFormat 表示数字的格式化类，即可以按照本地的风格习惯进行数字的显示。此类的定义如下：

```
public abstract class NumberFormat extends Format
```

可以发现，NumberFormat 是一个抽象类，和 MessageFormat 类一样，都是 Format 的子类，本类在使用时可以直接使用 NumberFormat 类中提供的静态方法为其实例化，NumberFormat 类的常用方法如表 11-12 所示。

表 11-12　NumberFormat 类的常用方法

序号	方　法	类型	描　述
1	public static Locale[] getAvailableLocales()	普通	返回所有语言环境的数组
2	public static final NumberFormat getInstance()	普通	返回当前默认语言环境的数字格式
3	public static NumberFormat getInstance(Locale inLocale)	普通	返回指定语言环境的数字格式
4	public static final NumberFormat getCurrencyInstance()	普通	返回当前默认环境的货币格式
5	public static NumberFormat getCurrencyInstance(Locale inLocale)	普通	返回指定语言环境的数字格式

下面将使用 NumberFormat 类进行数字的格式化操作。

范例：使用当前语言环境格式化数字

```
package org.lxh.demo11.numberdemo;
import java.text.NumberFormat;
public class NumberFormatDemo01 {
    public static void main(String[] args) {
        NumberFormat nf = null;
        nf = NumberFormat.getInstance();                    // 得到默认的数字格式显示
        System.out.println("格式化之后的数字：" + nf.format(1000000));
        System.out.println("格式化之后的数字：" + nf.format(1000.345));
    }
}
```

程序运行结果：

```
格式化之后的数字：1,000,000
格式化之后的数字：1,000.345
```

因为现在的操作系统是中文语言环境，所以以上数字显示成了中国的数字格式化形式，当然在 NumberFormat 中还有一个比较常用的子类——DecimalFormat。

11.8.2 DecimalFormat 类

DecimalFormat 类也是 Format 的一个子类，主要作用是格式化数字。当然，在格式化数字时要比直接使用 NumberFormat 更加方便，因为可以直接指定按用户自定义的方式进行格式化操作，与 SimpleDateFormat 类似，如果要进行自定义格式化操作，则必须指定格式化操作的模板，此模板如表 11-13 所示。

表 11-13 DecimalFormat 格式化模板

序号	标 记	位 置	描 述
1	0	数字	代表阿拉伯数字，每一个 0 表示一位阿拉伯数字，如果该位不存在则显示 0
2	#	数字	代表阿拉伯数字，每一个#表示一位阿拉伯数字，如果该位不存在则不显示
3	.	数字	小数点分隔符或货币的小数分隔符
4	-	数字	代表负号
5	,	数字	分组分隔符
6	E	数字	分隔科学计数法中的尾数和指数
7	;	子模式边界	分隔正数和负数子模式
8	%	前缀或后缀	数字乘以 100 并显示为百分数
9	\u2030	前缀或后缀	乘以 1000 并显示为千分数
10	¤ \u00A4	前缀或后缀	货币记号，由货币号替换。如果两个同时出现，则用国际货币符号替换；如果出现在某个模式中，则使用货币小数分隔符，而不使用小数分隔符
11	'	前缀或后缀	用于在前缀或后缀中为特殊字符加引号，例如 "'#'#" 将 123 格式化为 "#123"。要创建单引号本身，则连续使用两个单引号，例如"# o"clock"

下面通过实例代码说明以上格式化模板的使用。

范例：格式化数字

```
package org.lxh.demo11.numberdemo;
import java.text.DecimalFormat;
class FormatDemo{
    public void format1(String pattern,double value){
        DecimalFormat df = null ;                    // 声明一个DecimalFormat对象
        df = new DecimalFormat(pattern) ;            // 实例化对象
        String str = df.format(value) ;              // 格式化数字
        System.out.println("使用" + pattern
                + "格式化数字"+value+": " + str);
    }
}
public class NumberFormatDemo02 {
    public static void main(String[] args) {
        FormatDemo demo = new FormatDemo() ;
        demo.format1("###,###.###", 111222.34567);
        demo.format1("000,000.000", 11222.34567);
        demo.format1("###,###.###¥", 111222.34567);
        demo.format1("000,000.000¥", 11222.34567);
        demo.format1("##.###%", 0.345678);           // 使用百分数形式
        demo.format1("00.###%", 0.0345678);          // 使用百分数形式
        demo.format1("###.###\u2030", 0.345678);     // 使用千分数形式
    }
}
```

程序运行结果：

```
使用###,###.###格式化数字111222.34567: 111,222.346
使用000,000.000格式化数字11222.34567: 011,222.346
使用###,###.###¥格式化数字111222.34567: 111,222.346¥
使用000,000.000¥格式化数字11222.34567: 011,222.346¥
使用##.###%格式化数字0.345678: 34.568%
使用00.###%格式化数字0.0345678: 03.457%
使用###.###‰格式化数字0.345678: 345.678‰
```

11.9　BigInteger 类

当一个数字非常大时，则肯定无法使用基本类型接收，所以，最早碰到大数字时往往会使用 String 类进行接收，然后再采用拆分的方式进行计算，但操作非常麻烦，所以在 Java 中为了解决这样的难题提供了 BigInteger 类。BigInteger 表示是大整数类，定义在 java.math

包中，如果在操作时一个整型数据已经超过了整数的最大类型长度 long，数据无法装入，此时可以使用 BigInteger 类进行操作。

在 BigInteger 类中封装了各个常用的基本运算，表 11-14 列出了此类的常用方法。

表 11-14 BigInteger 类的常用方法

序　号	方　法	类　型	描　述
1	public BigInteger(String val)	构造	将一个字符串变为 BigInteger 类型的数据
2	public BigInteger add(BigInteger val)	普通	加法
3	public BigInteger subtract(BigInteger val)	普通	减法
4	public BigInteger multiply(BigInteger val)	普通	乘法
5	public BigInteger divide(BigInteger val)	普通	除法
6	public BigInteger max(BigInteger val)	普通	返回两个大数字中的最大值
7	public BigInteger min(BigInteger val)	普通	返回两个大数字中的最小值
8	public BigInteger[] divideAndRemainder (BigInteger val)	普通	除法操作，数组的第 1 个元素为除法的商，第 2 个元素为除法的余数

表 11-4 列出了 BigInteger 中的常用方法，因为其方法较多，读者可以自行查阅 JDK 文档。

范例：验证 BigInteger

```
package org.lxh.demo11.numberdemo;
import java.math.BigInteger;
public class BigIntegerDemo01 {
    public static void main(String[] args) {
        BigInteger bi1 = new BigInteger("123456789") ;// 定义BigInteger对象
        BigInteger bi2 = new BigInteger("987654321") ;// 定义BigInteger对象
        System.out.println("加法操作：" + bi2.add(bi1));      // 加法操作
        System.out.println("减法操作：" + bi2.subtract(bi1));// 减法操作
        System.out.println("乘法操作：" + bi2.multiply(bi1));// 乘法操作
        System.out.println("除法操作：" + bi2.divide(bi1));   // 除法操作
        System.out.println("最大数：" + bi2.max(bi1));        // 求出最大数
        System.out.println("最小数：" + bi2.min(bi1));        // 求出最小数
        BigInteger result[] = bi2.divideAndRemainder(bi1) ;// 除法操作
        System.out.println("商是：" + result[0] + "；余数是：" + result[1]);
    }
}
```

程序运行结果：

```
加法操作：1111111110
减法操作：864197532
乘法操作：121932631112635269
除法操作：8
最大数：987654321
```

```
最小数：123456789
商是：8；余数是：9
```

11.10　BigDecimal 类

对于不需要任何准确计算精度的数字可以直接使用 float 或 double，但是如果需要精确计算的结果，则必须使用 BigDecimal 类，而且使用 BigDecimal 类也可以进行大数的操作。BigDecimal 类的常用方法如表 11-15 所示。

表 11-15　BigDecimal 类的常用方法

序号	方　法	类型	描　述
1	public BigDecimal(double val)	构造	将 double 表示形式转换为 BigDecimal
2	public BigDecimal(int val)	构造	将 int 表示形式转换为 BigDecimal
3	public BigDecimal(String val)	构造	将字符串表示形式转换为 BigDecimal
4	public BigDecimal add(BigDecimal augend)	普通	加法
5	public BigDecimal subtract(BigDecimal subtrahend)	普通	减法
6	public BigDecimal multiply(BigDecimal multiplicand)	普通	乘法
7	public BigDecimal divide(BigDecimal divisor)	普通	除法

范例：进行四舍五入的四则运算

```
package org.lxh.demo11.numberdemo;
import java.math.BigDecimal;
class MyMath {
    public static double add(double d1, double d2) {        // 进行加法运算
        BigDecimal b1 = new BigDecimal(d1);
        BigDecimal b2 = new BigDecimal(d2);
        return b1.add(b2).doubleValue();
    }
    public static double sub(double d1, double d2) {        // 进行减法运算
        BigDecimal b1 = new BigDecimal(d1);
        BigDecimal b2 = new BigDecimal(d2);
        return b1.subtract(b2).doubleValue();
    }
    public static double mul(double d1, double d2) {        // 进行乘法运算
        BigDecimal b1 = new BigDecimal(d1);
        BigDecimal b2 = new BigDecimal(d2);
        return b1.multiply(b2).doubleValue();
    }
```

```
    public static double div(double d1, double d2,int len) {// 进行除法运算
        BigDecimal b1 = new BigDecimal(d1);
        BigDecimal b2 = new BigDecimal(d2);
        return b1.divide(b2,len,BigDecimal.ROUND_HALF_UP).doubleValue();
    }
    public static double round(double d, int len) {          // 进行四舍五入
                                                                操作
        BigDecimal b1 = new BigDecimal(d);
        BigDecimal b2 = new BigDecimal(1);
        // 任何一个数字除以1都是原数字
        // ROUND_HALF_UP是BigDecimal的一个常量，表示进行四舍五入的操作
        return b1.divide(b2, len,BigDecimal.ROUND_HALF_UP).doubleValue();
    }
}
public class BigDecimalDemo01 {
    public static void main(String[] args) {
        System.out.println("加法运算：" + MyMath.round(MyMath.add(10.345,
        3.333), 1));
        System.out.println("乘法运算：" + MyMath.round(MyMath.mul(10.345,
        3.333), 3));
        System.out.println("除法运算：" + MyMath.div(10.345, 3.333, 3));
        System.out.println("减法运算：" + MyMath.round(MyMath.sub(10.345,
        3.333), 3));
    }
}
```

程序运行结果：

```
加法运算：13.7
乘法运算：34.48
除法运算：3.104
减法运算：7.012
```

以上程序中最重要的就是 round()方法，此处的四舍五入操作实际上是用 divide()方法实现的，因为只有此方法才可以指定小数点之后的位数，而且任何一个数字除以 1 都是原数字。

11.11 对象克隆技术

在 Java 中支持对象的克隆操作，直接使用 Object 类中的 clone()方法即可，方法定义如下：

```
protected Object clone() throws CloneNotSupportedException
```

以上方法是受保护的类型，所以在子类中必须覆写此方法，而且覆写之后应该扩大访问权限，这样才能被外部调用，但是具体的克隆方法的实现还是在 Object 中，所以在覆写的方法中只需要调用 Object 类中的 clone()方法即可完成操作，而且在对象所在的类中必须

实现 Cloneable 接口才可以完成对象的克隆操作。

但是如果直接查询 JDK 文档会发现 Cloneable 接口中并没有任何的方法定义，所以此接口在设计上称为是一种标识接口，表示对象可以被克隆。

范例：对象的克隆操作

```
package org.lxh.demo11.clonedemo;
class Person implements Cloneable { // 必须实现Cloneable接口
    private String name = null;
    public Person(String name) {
        this.name = name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getName() {
        return this.name;
    }
    // 需要子类覆写clone方法
    public Object clone() throws CloneNotSupportedException {
        return super.clone();        // 具体的克隆操作由父类完成
    }
    public String toString() {
        return "姓名：" + this.getName();
    }
};
public class CloneDemo01 {
    public static void main(String[] args) throws Exception {
        Person p1 = new Person("张三");
        Person p2 = (Person) p1.clone();
        p2.setName("李四");
        System.out.println("原始对象：" + p1);
        System.out.println("克隆之后的对象：" + p2);
    }
}
```

程序运行结果：

```
原始对象：姓名：张三
克隆之后的对象：姓名：李四
```

11.12　Arrays 类

Arrays 类是数组的操作类，定义在 java.util 包中，主要功能是实现数组元素的查找、数

组内容的填充、排序等，其常用方法如表 11-16 所示。

表 11-16 Arrays 类的常用方法

序号	方 法	类型	描 述
1	public static boolean equals(int[] a,int[] a2)	普通	判断两个数组是否相等，此方法被重载多次，可以判断各种数据类型的数组
2	public static void fill(int[] a,int val)	普通	将指定内容填充到数组之中，此方法被重载多次，可以填充各种数据类型的数组
3	public static void sort(int[] a)	普通	数组排序，此方法被重载多次，可以对各种类型的数组进行排序
4	public static int binarySearch(int[] a,int key)	普通	对排序后的数组进行检索，此方法被重载多次，可以对各种类型的数组进行搜索
5	public static String toString(int[] a)	普通	输出数组信息，此方法被重载多次，可以输出各种数据类型的数组

下面以整型数组为例讲解 Arrays 类的常用方法。

范例：操作 Arrays 类

```
package org.lxh.demo11.arraysdemo;
import java.util.* ;
public class ArraysDemo {
    public static void main(String args[]) {
        int temp[] = { 3, 5, 7, 9, 1, 2, 6, 8 };    // 声明一个整型数组
        Arrays.sort(temp);                           // 数组排序
        System.out.print("排序后的数组") ;
        System.out.println(Arrays.toString(temp)) ;// 以字符串输出数组
        int point = Arrays.binarySearch(temp, 3) ; // 检索数据位置
        System.out.println("元素'3'的位置在：" + point) ;
        Arrays.fill(temp, 3) ;                       // 填充数组
        System.out.print("数组填充：") ;
        System.out.println(Arrays.toString(temp)) ;// 以字符串输出数组
    }
}
```

程序运行结果：

```
排序后的数组[1, 2, 3, 5, 6, 7, 8, 9]
元素'3'的位置在：2
数组填充：[3, 3, 3, 3, 3, 3, 3, 3]
```

以上程序先使用静态初始化的方式声明了一个一维数组，然后利用 Arrays 类的 sort()方法进行排序，并通过二分查找法查找指定的内容是否存在，重新将数组的内容填充后，又利用 toString()方法将全部的内容变为 String 的形式并输出。

11.13　Comparable 接口

11.13.1　比较器的基本应用

在讲解数组时，曾经讲过可以直接使用 java.util.Arrays 类进行数组的排序操作，而且 Arrays 类中的 sort 方法被重载多次，可以对任意类型的数组排序，排列时会根据数值的大小进行排序。同样此类也可以对 Object 数组进行排序，但是要使用此种方法排序也是有要求的，即对象所在的类必须实现 Comparable 接口，此接口就是用于指定对象排序规则的。Comparable 接口的定义如下：

```
public interface Comparable<T>{
    public int compareTo(T o) ;
}
```

从以上定义中可以发现，在 Comparable 接口中也使用了 Java 的泛型技术。其中只有一个 compareTo 方法，此方法返回一个 int 类型的数据，但是此 int 的值只能是以下 3 种：

- 1：表示大于。
- -1：表示小于。
- 0：表示相等。

下面来看一下具体的应用。例如，现在要求设计一个学生类，此类包含姓名、年龄、成绩，并产生一个对象数组，要求按成绩由高到低排序，如果成绩相等，则按年龄由低到高排序。如果直接编写排序操作，则会比较麻烦，所以，此时来观察如何使用 Arrays 类中的 sort()方法进行排序操作。

范例：使用比较器进行排序操作

```
package org.lxh.demo11.comparabledemo;
class Student implements Comparable<Student> { // 指定类型为Student
    private String name;
    private int age;
    private float score;
    public Student(String name, int age, float score) {
        this.name = name;
        this.age = age;
        this.score = score;
    }
    public String toString() {
        return name + "\t\t" + age + "\t\t" + score;
    }
    public int compareTo(Student stu) {          // 覆写compareTo()方法，实现
                                                    排序规则的应用
        if (this.score > stu.score) {
```

```
            return -1;
        } else if (this.score < stu.score) {
            return 1;
        } else {
            if (this.age > stu.age) {
                return 1;
            } else if (this.age < stu.age) {
                return -1;
            } else {
                return 0;
            }
        }
    }
};
public class ComparableDemo01 {
    public static void main(String[] args) {
        Student stu[] = { new Student("张三", 20, 90.0f),
                new Student("李四", 22, 90.0f), new Student("王五", 20, 99.0f),
                new Student("赵六", 20, 70.0f), new Student("孙七", 22, 100.
                0f) };
        java.util.Arrays.sort(stu);                    // 进行排序操作
        for (int i = 0; i < stu.length; i++) { // 输出数组中的内容
            System.out.println(stu[i]);
        }
    }
}
```

程序运行结果：

```
孙七    22    100.0
王五    20    99.0
张三    20    90.0
李四    22    90.0
赵六    20    70.0
```

由程序运行结果可以发现，程序完成了要求的排序规则，对对象数组进行了排序操作，但是要特别提醒读者注意的是，如果在此时 Student 类中没有实现 Comparable 接口，则在执行时会出现以下异常：

```
Exception in thread "main" java.lang.ClassCastException:
org.lxh.demo11. comparabledemo.Student cannot be cast to java.lang.Comparable
    at java.util.Arrays.mergeSort(Unknown Source)
    at java.util.Arrays.sort(Unknown Source)
    at org.lxh.demo11.comparabledemo.ComparableDemo01.main(ComparableDemo01.
java:35)
```

此异常是类型转换异常，因为在排序时，所有的对象都将向 Comparable 进行转换，所以，一旦没有实现此接口就会出现以上错误。

11.13.2　分析比较器的排序原理

实际上 11.13.1 节所讲解的排序过程也就是数据结构中的二叉树排序方法，通过二叉树进行排序，然后利用中序遍历的方式把内容依次读取出来。

二叉树排序的基本原理就是：将第 1 个内容作为根节点保存，如果后面的值比根节点的值小，则放在根节点的左子树，如果后面的值比根节点的值大，则放在根节点的右子树。

按照这样的思路，如果给出了数字 8、3、10、9、1、5，则保存的树结构如图 11-8 所示。

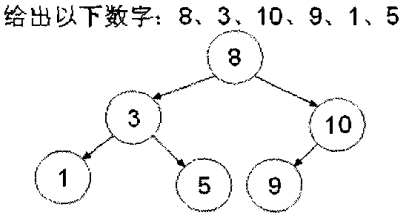


图 11-8　二叉树排序原理

然后根据中序遍历的原理（左子树→根节点→右子树的方式）即可将数字排序读取出来，排序后的结果为 1、3、5、8、9、10。

下面按照以上思路编写一个简单的二叉树排序操作，此处为了简化代码，直接使用 Integer 类，因为 Integer 类本身已经实现了 Comparable 接口。在讲解此操作之前，读者先看下面的范例。

范例：Integer 为 Comparable 接口实例化

```
package org.lxh.demo11.comparabledemo;
public class ComparableDemo02 {
    public static void main(String[] args) {
        Comparable com = null;                    // 声明一个Comparable接口对象
        com = 30;                                 // 通过Integer类为Comparable实例化
        System.out.println("内容为：" +com); // 实际上调用的是toString()方法
    }
}
```

程序运行结果：

```
内容为：30
```

从以上代码中可以清楚地发现，Comparable 接口通过 Integer 对象实例化，然后直接输出 Comparable 接口对象时实际上调用的是 Integer 类中的 toString()方法，此方法已经被 Integer 类覆写了。那么下面的代码就将直接使用 Comparable 接口完成，输出时为了方便，也直接将 Comparable 接口打印。

范例：基于 Comparable 接口实现的二叉树操作

```
package org.lxh.demo11.comparabledemo;
class BinaryTree {
    class Node {                                          // 声明一个节点类
        private Comparable data;                          // 保存具体的内容
        private Node left;                                // 保存左子树
        private Node right;                               // 保存右子树
        public void addNode(Node newNode) {
            // 要确定是放在左子树还是右子树
            if (newNode.data.compareTo(this.data) < 0) {
                if (this.left == null) {                  // 放在左子树
                    this.left = newNode;
                } else {
                    this.left.addNode(newNode);
                }
            }
            if (newNode.data.compareTo(this.data) >= 0) {
                if (this.right == null) {                 // 放在右子树
                    this.right = newNode;
                } else {
                    this.right.addNode(newNode);
                }
            }
        }
        public void printNode() {                         // 输出时采用中序遍历
            if (this.left != null) {                      // 先输出左子树
                this.left.printNode();
            }
            System.out.print(this.data + "\t");           // 再输出根节点
            if (this.right != null) {                     // 最后输出右子树
                this.right.printNode();
            }
        }
    };
    private Node root;                                    // 根元素
    public void add(Comparable data) {
        Node newNode = new Node();                        // 每传入一个新的内容就声
                                                          明一个根节点
        newNode.data = data;
        if (root == null) {
            root = newNode;                               // 如果是第1个元素，设置成
                                                          根元素
```

```
            } else {
                root.addNode(newNode);              // 确定节点是放在左子树还是右子树
            }
        }
        public void print() {                       // 输出节点
            this.root.printNode();
        }
    };
    public class ComparableDemo03 {
        public static void main(String args[]) {
            BinaryTree bt = new BinaryTree();
            bt.add(8);
            bt.add(3);
            bt.add(3);                              // 加入重复元素
            bt.add(10);
            bt.add(9);
            bt.add(1);
            bt.add(5);
            bt.add(5);                              // 加入重复元素
            System.out.println("排序之后的结果：");
            bt.print();
        }
    }
```

程序运行结果：

```
排序之后的结果：
1   3   3   5   5   8   9   10
```

以上程序的实现实际上与链表操作类似，只是多增加了一个节点的引用操作。对于这种排序的原理读者只需要了解其基本概念即可，因为在 Java 中已经对各个常见的数据结构算法提供了很好的支持，这一点在 Java 类集部分将为读者详细讲解。

11.14　另一种比较器 Comparator

如果一个类已经开发完成，但是在此类建立的初期并没有实现 Comparable 接口，此时肯定是无法进行对象排序操作的，所以为了解决这样的问题，Java 又定义了另一个比较器的操作接口——Comparator。此接口定义在 java.util 包中，接口定义如下：

```
public interface Comparator<T>{
    public int compare(To1,To2) ;
    boolean equals(Object obj) ;
}
```

可以发现，在此接口中也存在一个 compareTo()方法，但是与之前不同的是，此方法要接收两个对象，其返回值依然是 0、-1、1。

但是，此接口与之前不同的是，需要单独指定好一个比较器的比较规则类才可以完成数组排序。下面定义一个学生类，其中有姓名和年龄属性，并按照年龄排序。

范例：定义学生类

```
package org.lxh.demo11.comparatordemo;
public class Student {
    private String name;
    private int age;
    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public boolean equals(Object obj) {          // 覆写equals方法
        if (this == obj) {
            return true;
        }
        if (!(obj instanceof Student)) {
            return false;
        }
        Student stu = (Student) obj;
        if (stu.name.equals(this.name) && stu.age == this.age) {
            return true;
        } else {
            return false;
        }
    }
    public String toString() {
        return name + "\t\t" + age;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
```

```
        }
    }
```

范例：定义比较规则

```
package org.lxh.demo11.comparatordemo;
import java.util.Comparator;
public class StudentComparator implements Comparator<Student> {// 实现比较器接口
    public int compare(Student s1, Student s2) {    // 为学生类定义比较规则
        if (s1.equals(s2)) {                        // 如果相等，则返回0
            return 0;
        } else if (s1.getAge() < s2.getAge()) {     // 按年龄比较
            return 1;
        } else {
            return -1;
        }
    }
}
```

范例：为对象数组排序

```
package org.lxh.demo11.comparatordemo;
public class ComparatorDemo {
    public static void main(String[] args) {
        Student stu[] = { new Student("张三", 20),
                new Student("李四", 22), new Student("王五", 20),
                new Student("赵六", 20), new Student("孙七", 22) };
        java.util.Arrays.sort(stu,new StudentComparator()) ;// 排序，指定排序规则
        for (int i = 0; i < stu.length; i++) {              // 输出数组中的内容

            System.out.println(stu[i]);
        }
    }
}
```

程序运行结果：

```
李四    22
孙七    22
张三    20
王五    20
赵六    20
```

从程序的运行结果中可以发现，Comparator 和 Comparable 两个接口都可以实现相同的排序功能，但是与 Comparable 接口相比，Comparator 接口明显是一种补救的做法。所以，

本书建议读者还是使用 Comparable 接口进行排序操作比较方便。

11.15 观察者设计模式

11.15.1 什么叫观察者

在解释观察者设计模式的概念之前，读者先来看一个简单的例子：现在很多的购房者都在关注着房子的价格变化，每当房子价格变化时，所有的购房者都可以观察得到，实际上以上的购房者都属于观察者，他们都在关注着房子的价格。实际上这就叫观察者设计模式，如图 11-9 所示。

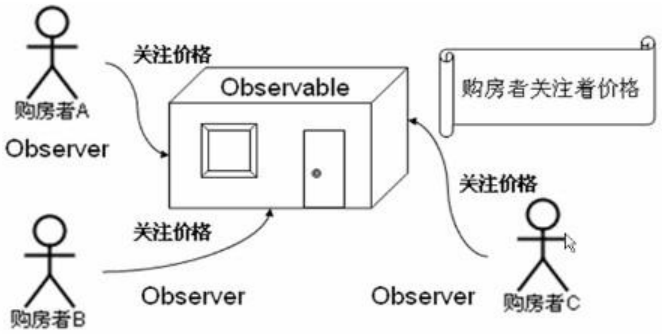


图 11-9 所有的购房者关注着价格

在 Java 中可以直接依靠 Observable 类和 Observer 接口轻松地实现以上功能。

11.15.2 观察者模式实现

在 java.util 包中提供了 Observable 类和 Observer 接口，使用它们即可完成观察者模式。需要被观察的类必须继承 Observable 类，Observable 类的常用方法如表 11-17 所示。

表 11-17 Observable 类的常用方法

序 号	方 法	类 型	描 述
1	public void addObserver(Observer o)	普通	添加一个观察者
2	public void deleteObserver(Observer o)	普通	删除一个观察者
3	protected void setChanged()	普通	被观察者状态发生改变
4	public void notifyObservers(Object arg)	普通	通知所有观察者状态改变

然后每一个观察者类都需要实现 Observer 接口，Observer 接口定义如下：

```
public interface Observer{
    void update(Observable o,Object arg) ;
}
```

在此接口中只定义了一个 update 方法，第 1 个参数表示被观察者实例，第 2 个参数表示修改的内容。

范例：观察者模式的实现

```
package org.lxh.demo11.obserdemo;
import java.util.Observable;
import java.util.Observer;
class House extends Observable {
    private float price;
    public House(float price) {
        this.price = price;
    }
    public float getPrice() {
        return price;
    }
    public void setPrice(float price) {
        super.setChanged();                    // 设置变化点
        super.notifyObservers(price);          // 通知所有观察者价格改变
        this.price = price;
    }
    public String toString() {
        return "房子价格为：" + this.price;
    }
}
class HousePriceObserver implements Observer {
    private String name;
    public HousePriceObserver(String name) {
        this.name = name;
    }
    public void update(Observable obj, Object arg) {
        if (arg instanceof Float) {            // 判断参数类型
            System.out.print(this.name + "观察到价格更改为：");
            System.out.println(((Float) arg).floatValue());
        }
    }
}
public class ObserDemo01{
    public static void main(String[] args) {
        House h = new House(1000000);
        HousePriceObserver hpo1 = new HousePriceObserver("购房者A");
        HousePriceObserver hpo2 = new HousePriceObserver("购房者B");
        HousePriceObserver hpo3 = new HousePriceObserver("购房者C");
        h.addObserver(hpo1);                   // 加入观察者
        h.addObserver(hpo2);                   // 加入观察者
        h.addObserver(hpo3);                   // 加入观察者
```

```
            System.out.println(h);                    // 输出房子价格
            h.setPrice(666666);                       // 修改房子价格
            System.out.println(h);                    // 输出房子价格
        }
    }
```

程序运行结果：

```
房子价格为：1000000.0
购房者C观察到价格更改为：666666.0
购房者B观察到价格更改为：666666.0
购房者A观察到价格更改为：666666.0
房子价格为：666666.0
```

从程序的运行结果中可以发现，多个观察者都在关注着价格的变化，只要价格一有变化，则所有观察者会立刻有所行动。

11.16 正则表达式

11.16.1 认识正则表达式

使用正则表达式可以方便地对数据进行匹配，还可以执行更加复杂的字符串验证、拆分、替换功能。例如，现在要求判断一个字符串是否由数字组成，则可以有以下两种做法。

范例：不使用正则验证

```
package org.lxh.demo11.regexdemo;
public class RegexDemo01 {
    public static void main(String[] args) {
        String str = "1234567890";                    // 此字符串由数字组成
        boolean flag = true;
        char c[] = str.toCharArray();                 // 将字符串变为字符数组
        for (int i = 0; i < c.length; i++) {// 依次判断
            if (c[i] < '0' || c[i] > '9') {  // 判断每一个字符是否是0～9
                flag = false;                         // 如果不是则退出循环，并做下标记
                break ;
            }
        }
        if (flag) {                                   // 判断输出
            System.out.println("是由数字组成！");
        } else {
            System.out.println("不是由数字组成！");
        }
    }
}
```

程序运行结果：

```
是由数字组成！
```

上面代码的基本思路是：先将一个字符串拆分成一个字符数组，然后对数组中的每个元素进行验证，如果发现字符的范围不是在 0～9 之间，则表示不是数字，则设置一个标志位，并退出循环。

范例：使用正则表达式验证

```
package org.lxh.demo11.regexdemo;
import java.util.regex.Pattern;
public class RegexDemo02 {
    public static void main(String[] args) {
        String str = "1234567890";                                    // 此字符串由数字组成
        if (Pattern.compile("[0-9]+").matcher(str).matches()) {// 使用正则验证
            System.out.println("是由数字组成！");
        } else {
            System.out.println("不是由数字组成！");
        }
    }
}
```

程序运行结果：

```
是由数字组成！
```

以上代码完成了和第 1 个范例同样的功能，但是代码的长度要比第 1 个程序短很多。实际上以上程序就是使用正则表达式进行验证的，而中间的“[0-9]+”就是正则表达式的匹配字符，表示的含义是：由 1 个以上的数字组成。

实际上具体的正则表达式操作类是需要通过 Pattern 和 Matcher 两个类完成操作的。

提示：正则表达式是在 JDK 1.4 之后加入的。

正在表达式是在 JDK 1.4 之后引入到 Java 之中的，在最早的 Java 开发中，如果要使用正则表达式，则需要单独安装 Apache 提供的正则开发包才可以使用。

11.16.2　Pattern 类和 Matcher 类

如果要在程序中应用正则表达式则必须依靠 Pattern 类与 Matcher 类，这两个类都在 java.util.regex 包中定义。Pattern 类的主要作用是进行正则规范（如 11.16.1 节程序中的“[0-9]”就属于正则规范）的编写，而 Matcher 类主要是执行规范，验证一个字符串是否符合其规范。

常用的正则规范的定义如表 11-18～表 11-20 所示。

表 11-18　常用正则规范

序号	规　范	描　述	序号	规　范	描　述
1	\\	表示反斜线（\）字符	9	\w	表示字母、数字、下划线
2	\t	表示制表符	10	\W	表示非字母、数字、下划线
3	\n	表示换行	11	\s	表示所有空白字符（换行、空格等）
4	[abc]	字符 a、b 或 c	12	\S	表示所有非空白字符
5	[^abc]	表示除了 a、b、c 之外的任意字符	13	^	行的开头
6	[a-zA-Z0-9]	表示由字母、数字组成	14	$	行的结尾
7	\d	表示数字	15	.	匹配除换行符之外的任意字符
8	\D	表示非数字			

表 11-19　数量表示（X 表示一组规范）

序号	规　范	描　述	序号	规　范	描　述
1	X	必须出现一次	5	X{n}	必须出现 n 次
2	X?	可以出现 0 次或 1 次	6	X{n,}	必须出现 n 次以上
3	X*	可以出现 0 次、1 次或多次	7	X{n,m}	必须出现 n～m 次
4	X+	可以出现 1 次或多次			

表 11-20　逻辑运算符（X、Y 表示一组规范）

序号	规　范	描　述	序号	规　范	描　述
1	XY	X 规范后跟着 Y 规范	3	(X)	作为一个捕获组规范
2	X|Y	X 规范或 Y 规范			

在 Pattern 类中直接使用表 11-18～表 11-20 中的正则规则即可完成相应的操作，Pattern 类的常用方法如表 11-21 所示。

表 11-21　Pattern 类的常用方法

序号	方　法	类型	描　述
1	public static Pattern compile(String regex)	普通	指定正则表达式规则
2	public Matcher matcher(CharSequence input)	普通	返回 Matcher 类实例
3	public String[] split(CharSequence input)	普通	字符串拆分

在 Pattern 类中如果要取得 Pattern 类实例，则必须调用 compile()方法。

如果要验证一个字符串是否符合规范，则可以使用 Matcher 类，Matcher 类的常用方法如表 11-22 所示。

表 11-22　Matcher 类的常用方法

序号	方　法	类型	描　述
1	public boolean matches()	普通	执行验证
2	public String replaceAll(String replacement)	普通	字符串替换

下面直接使用 Pattern 类和 Matcher 类完成一个简单的验证过程。

- 日期格式要求：yyyy-mm-dd。
- 正则表达式如图 11-10 所示。

日期：	1983	-	07	-	27
格式：	四位数字		两位数字		两位数字
正则：	\d{4}	-	\d{2}	-	\d{2}

图 11-10　正则表达式

范例：验证一个字符串是否是合法的日期格式

```
package org.lxh.demo11.regexdemo;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class RegexDemo03 {
    public static void main(String[] args) {
        String str = "1983-07-27";
        String pat = "\\d{4}-\\d{2}-\\d{2}" ;   // 定义验证规则
        Pattern p = Pattern.compile(pat) ;      // 实例化Pattern类
        Matcher m = p.matcher(str) ;            // 验证字符串内容是否合法
        if (m.matches()) {                      // 使用正则验证
            System.out.println("日期格式合法！");
        } else {
            System.out.println("日期格式不合法！！");
        }
    }
}
```

在程序中“\”字符是需要进行转义的，两个“\”实际上表示的是一个“\”，所以实际上“\\d”表示的是“\d”。

当然，也可以直接使用 Pattern 类对一个字符串进行拆分操作。

范例：按照字符串的数字将字符串拆分

```
package org.lxh.demo11.regexdemo;
import java.util.regex.Pattern;
public class RegexDemo04 {
    public static void main(String[] args) {
        String str = "A1B22C333D4444E55555F";
        String pat = "\\d+" ;                   // 定义验证规则
        Pattern p = Pattern.compile(pat) ;      // 实例化Pattern类
        String s[] = p.split(str);              // 进行字符串拆分
        for(int x=0;x<s.length;x++){
            System.out.print(s[x]+"\t") ;       // 输出
        }
```

```
    }
}
```

程序运行结果：

```
A   B   C   D   E   F
```

在字符串中因为数字的长度不一样，所以这里使用的“\\d+”表示 1 位或多位数字都可以拆分。

了解了匹配和拆分操作，下面再来看 Matcher 类中的替换操作。

范例：将全部的数字替换成“_”

```
package org.lxh.demo11.regexdemo;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class RegexDemo05 {
    public static void main(String[] args) {
        String str = "A1B22C333D4444E55555F";
        String pat = "\\d+" ;                    // 定义验证规则
        Pattern p = Pattern.compile(pat) ;       // 实例化Pattern类
        Matcher m = p.matcher(str) ;             // 实例化Matcher类
        String newString = m.replaceAll("_") ; // 替换数字
        System.out.println(newString) ;
    }
}
```

程序运行结果：

```
A_B_C_D_E_F
```

以上代码为读者演示了正则表达式的操作流程，有些读者可能会觉得字符串拆分和替换与 String 类中的方法功能类似，实际上，在 JDK 1.4 之后 String 已经可以很好地支持正则表达式的应用了。

11.16.3 String 类对正则表达式的支持

在 String 类中有 3 个方法支持正则操作，如表 11-23 所示。

表 11-23 String 类的正则支持

序　号	方　法	类　型	描　述
1	public boolean matches(String regex)	普通	字符串匹配
2	public String replaceAll(String regex,String replacement)	普通	字符串替换
3	public String[] split(String regex)	普通	字符串拆分

下面为读者演示表 11-23 所示的 3 个方法，为了方便读者理解，本范例直接对之前的代码进行修改。

范例：使用 String 修改之前操作

```
package org.lxh.demo11.regexdemo;
public class RegexDemo06 {
    public static void main(String[] args) {
        String str1 = "A1B22C333D4444E55555F".replaceAll("\\d+", "_");
        boolean temp = "1983-07-27".matches("\\d{4}-\\d{2}-\\d{2}");
        String s[] = "A1B22C333D4444E55555F".split("\\d+");
        System.out.println("字符串替换操作：" + str1);
        System.out.println("字符串替换验证：" + temp);
        System.out.print("字符串拆分：") ;
        for (int x = 0; x < s.length; x++) {
            System.out.print(s[x] + "\t");
        }
    }
}
```

程序运行结果：

```
字符串替换操作：A_B_C_D_E_F
字符串替换验证：true
字符串拆分：A B    C    D    E    F
```

可以发现，以上程序的运行结果和之前是一样的，所以在此处建议读者以后最好直接使用 String 类中的方法，这样会比较方便。

提示：对于一些敏感字符操作时需要转义。

在正则操作中，如果出现了一些正则表达式中的字符，则需要对这些字符进行转义，例如，现在有字符串“LXH:98|MLDN:90|LI:100”，要求将其拆分成以下形式：

- LXH　98
- MLDN　90
- LI　100

如果要完成这样的操作，则肯定应该先使用“|”拆分，之后再使用“:”拆分，如果直接使用“|”拆分会发现根本就无法正确地执行。

范例：使用“|”拆分

```
package org.lxh.demo11.regexdemo;
public class RegexDemo07{
    public static void main(String args[]){
        String info = "LXH:98|MLDN:90|LI:100" ;          // 定义一个字符串
        String s[] = info.split("|") ;                   // 使用“|”拆分
        System.out.print("字符串的拆分：") ;
        for(int x=0;x<s.length;x++){                     // 循环输出
            System.out.print(s[x] + "、");
        }
```

```
    }
};
```

程序运行结果：

```
字符串的拆分：、L、X、H、:、9、8、|、M、L、D、N、:、9、0、|、L、I、:、1、0、0、
```

从运行结果中可以发现，字符串并没有按照指定的样式进行拆分，这是因为“|”在正则中表示或的概念，所以在使用时必须对其进行转义，写成“\\|”形式，其中“\\”表示一个“\”。

范例：对正则进行转义

```
package org.lxh.demo11.regexdemo;
public class RegexDemo08{
    public static void main(String args[]){
        String info = "LXH:98|MLDN:90|LI:100" ;      // 定义一个字符串
        String s[] = info.split("\\|") ;             // 使用“|”拆分，需要转义
        System.out.println("字符串的拆分：") ;
        for(int x=0;x<s.length;x++){
            String s2[] = s[x].split(":") ;          // 使用“:”拆分
            System.out.println("\t|- " + s2[0] + "\t" + s2[1]) ;
        }
    }
};
```

程序运行结果：

```
字符串的拆分：
    |- LXH   98
    |- MLDN  90
    |- LI    100
```

注意：本节在开发中十分有用，希望读者注意一下

在正则操作中，如果发现某些字符无法直接使用，则最好将其进行转义处理。

11.17 定 时 调 度（在WEB上运用更多）

11.17.1 Timer 类

Timer 类是一种线程设施，可以用来实现在某一个时间或某一段时间后安排某一个任务执行一次或定期重复执行。该功能要与 TimerTask 配合使用。TimerTask 类用来实现由 Timer 安排的一次或重复执行的某一个任务。

每一个 Timer 对象对应的是一个线程，因此计时器所执行的任务应该迅速完成，否则可能会延迟后续任务的执行，而这些后续的任务就有可能堆在一起，等到该任务完成后才能快速连续执行。Timer 类中的常用方法如表 11-24 所示。

表 11-24　Timer 类中的常用方法

序号	方　法	类型	描　述
1	public Timer()	构造	用来创建一个计时器并启动该计时器
2	public void cancel()	普通	用来终止该计时器，并放弃所有已安排的任务，对当前正在执行的任务没有影响
3	public int purge()	普通	将所有已经取消的任务移除，一般用来释放内存空间
4	public void schedule(TimerTask task, Date time)	普通	安排一个任务在指定的时间执行，如果已经超过该时间，则立即执行
5	public void schedule(TimerTask task,Date firstTime, long period)	普通	安排一个任务在指定的时间执行，然后以固定的频率（单位：毫秒）重复执行
6	public void schedule(TimerTask task,long delay)	普通	安排一个任务在一段时间（单位：毫秒）后执行
7	public void schedule(TimerTask task,long delay, long period)	普通	安排一个任务在一段时间（单位：毫秒）后执行，然后以固定的频率（单位：毫秒）重复执行
8	public void scheduleAtFixedRate(TimerTask task, Date firstTime, long period)	普通	安排一个任务在指定的时间执行，然后以近似固定的频率（单位：毫秒）重复执行
9	Public void scheduleAtFixedRate(TimerTask task,long delay,long period)	普通	安排一个任务在一段时间（单位：毫秒）后执行，然后以近似固定的频率（单位：毫秒）重复执行

在这里需要说明的是，schedule()与 scheduleAtFixedRate()方法的区别在于重复执行任务时对于时间间隔出现延迟的情况处理：

- schedule()方法的执行时间间隔永远是固定的，如果之前出现了延迟的情况，之后也会继续按照设定好的间隔时间来执行。
- scheduleAtFixedRate()方法可以根据出现的延迟时间自动调整下一次间隔的执行时间。

11.17.2　TimerTask 类

要执行具体的任务，则必须使用 TimerTask 类。TimerTask 类是一个抽象类，如果要使用该类，需要自己建立一个类来继承此类，并实现其中的抽象方法。TimerTask 中的常用方法如表 11-25 所示。

表 11-25　TimerTask 类中的常用方法

序号	方　法	类　型	描　述
1	public void cancel()	普通	用来终止此任务，如果该任务只执行一次且还没有执行，则永远不会再执行，如果为重复执行任务，则之后不会再执行（如果任务正在执行，则执行完后不会再执行）

续表

序号	方　法	类　型	描　述
2	public void run()	普通	该任务所要执行的具体操作，该方法为引入的接口 Runnable 中的方法，子类需要覆写此方法
3	public long scheduledExecutionTime()	普通	返回最近一次要执行该任务的时间（如果正在执行，则返回此任务的执行安排时间），一般在 run()方法中调用，用来判断当前是否有足够的时间来执行完成该任务

学习完了基本知识后，下面介绍如何使用以上的两个类完成定时操作。

11.17.3 范例——定时操作

下面通过一个简单的范例来演示 Timer 和 TimerTask 类的使用。程序的主要功能是定时打印系统的当前时间。

范例：建立 TimerTask 的子类

```
package org.lxh.demo11.task;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.TimerTask;
public class MyTask extends TimerTask { // 任务调度类要继承TimerTask
    public void run() {
        SimpleDateFormat sdf = null;
        sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss:SSS");
        System.out.println("当前系统时间为：" + sdf.format(new Date()));
    }
}
```

范例：建立测试类，进行任务调度

```
package org.lxh.demo11.task;
import java.util.Timer;
public class TestTask {
    public static void main(String[] args) {
        Timer t = new Timer();              // 建立Timer类对象
        MyTask mytask = new MyTask();       // 定义任务
        t.schedule(mytask, 1000, 2000);// 设置任务的执行，1秒后开始，每2秒重复
    }
}
```

程序运行结果：

```
当前系统时间为：2008-12-07 13:15:03:267
当前系统时间为：2008-12-07 13:15:04:810
当前系统时间为：2008-12-07 13:15:06:812
当前系统时间为：2008-12-07 13:15:08:815
```

```
当前系统时间为：2008-12-07 13:15:10:818
当前系统时间为：2008-12-07 13:15:12:821
当前系统时间为：2008-12-07 13:15:14:824
当前系统时间为：2008-12-07 13:15:16:827
```

11.18　本 章 要 点

1．在一个字符串内容需要频繁修改时，使用 StringBuffer 可以提升操作性能，因为 StringBuffer 内容可以改变，而 String 内容不可以改变。

2．StringBuffer 类中提供了大量的字符串操作方法，如增加、替换、插入等。

3．Runtime 表示运行时在一个 JVM 中只存在一个 Runtime，所以如果要想取得 Runtime 类的对象，直接使用 Runtime 类中提供的静态方法 getRuntime()即可。

4．国际化程序实现的基本原理为：所有的语言信息以 key→value 的形式保存在资源文件之中，程序通过 key 找到相应的 value，根据其所设置国家的 Locale 对象不同，找到的资源文件也不同。要想实现国际化必须依靠 Locale、ResourceBundle 两个类共同完成。

5．System 类是系统类，可以取得系统的相关信息，使用 System.gc()方法可以强制性地进行垃圾的收集操作，调用此方法实际上就是调用了 Runtime 类中的 gc()方法。

6．Format 类为格式化操作类，主要的 3 个子类是 MessageFormat、NumberFormat、DateFormat。

7．使用 Date 类可以方便地取得时间，但取得的时间格式不符合地域的风格，所以可以使用 SimpleDateFormat 类进行日期的格式化操作。

8．处理大数字可以使用 BigInteger 和 BigDecimal 类，当需要精确小数点操作位数时应使用 BigDecimal 类。

9．通过 Random 类可以取得指定范围的随机数字。

10．如果一个类的对象要被克隆，则此对象所在的类必须实现 Cloneable 接口。

11．要对一组对象进行排序，则必须使用比较器，比较器接口 Comparable 中定义了 compareTo()的比较方法，用来设置比较规则。

12．正则表达式是开发中最常使用的一种验证方法，String 类中的 replaceAll()、split()、matches()方法都对正则有所支持。

13．可以使用 Time 和 TimeTask 类完成系统的定时操作。

11.19　习　　题

1．定义一个 StringBuffer 类对象，然后通过 append()方法向对象中添加 26 个小写字母，要求每次只添加一次，共添加 26 次，然后按照逆序的方式输出，并且可以删除前 5 个字符。

2．利用 Random 类产生 5 个 1～30 之间（包括 1 和 20）的随机整数。

3．输入一个 Email 地址，然后使用正则表达式验证该 Email 地址是否正确。

4．编写程序，用 0～1 之间的随机数来模拟扔硬币试验，统计扔 1000 次后出现正、反面的次数并输出。

5．编写正则表达式，判断给定的是否是一个合法的 IP 地址。

6．给定下面的 HTML 代码：

```
<font face="Arial,Serif" size="+2" color="red">
```



要求对内容进行拆分，拆分之后的结果是：

```
face Arial,Serif
size +2
color red
```

7．编写程序，实现国际化应用，从命令行输入国家的代号，例如，1 表示中国，2 表示美国，然后根据输入代号的不同调用不同的资源文件显示信息。

8．按照“姓名：年龄：成绩|姓名：年龄：成绩”的格式定义字符串“张三:21:98|李四:22:89|王五 20:70”，要求将每组值分别保存在 Student 对象之中，并对这些对象进行排序，排序的原则为：按照成绩由高到低排序，如果成绩相等，则按照年龄由低到高排序。

第 12 章　Java IO

通过本章的学习可以达到以下目标：

- ☑ 掌握 java.io 包中类的继承关系。
- ☑ 可以使用 File 类进行文件的操作。
- ☑ 可以使用字节流或字符流操作文件内容并区分出字节流与字符流的区别。
- ☑ 掌握内存操作输入/输出流的使用。
- ☑ 了解线程通讯流——管道流的作用。
- ☑ 掌握 System 类对 IO 的支持 System.out、System.err、System.in。
- ☑ 可以使用打印流方便地打印输出的内容，并可以使用 Java 新特性格式化输出。
- ☑ 可以使用 BufferedReader 类读取缓冲区中的内容。
- ☑ 了解 Java 提供的输入工具类 Scanner 类的使用。
- ☑ 掌握数据操作流 DataInputStream 和 DataOutputStream 类的使用。
- ☑ 可以使用 SequenceInputStream 合并两个文件的内容。
- ☑ 可以使用压缩流完成 ZIP 文件格式的压缩。
- ☑ 了解回退流（PushbackInputStream）类的作用。
- ☑ 了解字符的主要编码类型及乱码产生原因。
- ☑ 掌握对象序列化的作用以及 Serializable 接口、Externalizable 接口、transient 关键字的使用。

Java IO 操作主要指的是使用 Java 进行输入、输出操作，Java 中的所有操作类都存放在 java.io 包中，在使用时需要导入此包。

在整个 java.io 包中最重要的就是 5 个类和一个接口，5 个类指的是 File、OutputStream、InputStream、Writer、Reader；一个接口指的是 Serializable。掌握了这些 IO 操作的核心就可以掌握了。**本章视频录像讲解时间为 7 小时 52 分钟，源代码在光盘对应的章节下。**

12.1　操作文件的类——File

12.1.1　File 类的基本介绍

在整个 io 包中，唯一与文件本身有关的类就是 File 类。使用 File 类可以进行创建或删除文件等常用操作。要使用 File 类，则首先要观察 File 类的构造方法，此类的常用构造方法如下：

```
public File(String pathname)  → 实例化File类时，必须设置好路径。
```

可以发现，如果要使用一个 File 类，则必须向 File 类的构造方法中传递一个文件路径，

例如，现在要操作 D 盘下的 test.txt 文件，则路径必须写成“d:\\test.txt”，其中“\\”表示一个“\”，而要操作文件，还需要使用 File 类中定义的若干方法，File 类中的主要方法如表 12-1 所示。

表 12-1 File 类中的主要方法和常量

序号	方法或常量	类型	描 述
1	public static final String pathSeparator	常量	表示路径的分隔符（windows 是：“;”）
2	public static final String separator	常量	表示路径的分隔符（windows 是：“\”）
3	public File(String pathname)	构造	创建 File 类对象，传入完整路径
4	public boolean createNewFile() throws IOException	普通	创建新文件
5	public boolean delete()	普通	删除文件
6	public boolean exists()	普通	判断文件是否存在
7	public boolean isDirectory()	普通	判断给定的路径是否是一个目录
8	public long length()	普通	返回文件的大小
9	public String[] list()	普通	列出指定目录的全部内容，只是列出了名称
10	public File[] listFiles()	普通	列出指定目录的全部内容，会列出路径
11	public boolean mkdir()	普通	创建一个目录
12	public boolean renameTo(File dest)	普通	为已有的文件重新命名

以上操作方法是在开发中较为常见的，下面通过若干实例进行讲解。

提问：为什么 File 类中的常量定义的命名规则不符合标准命名规则？

在之前讲解 Java 命名规则时曾讲过，一个常量的全部标识符不是应该大写吗？为什么在定义 pathSeparator 或 separator 时没有定义。

回答：Java 发展的历史原因造成的。

这里确实不符合 Java 的标准命名规则，主要原因是因为 Java 的发展经过了一段相当长的时间，而命名规范也是逐步形成的，File 类因为出现较早，所以当时并没有对命名规范有严格的要求，这些都属于 Java 的历史遗留问题。

12.1.2 使用 File 类操作文件

1. 实例操作一：创建一个新文件

File 类的对象实例化完成之后，就可以使用 createNewFile 创建一个新文件，但是此方法使用了 throws 关键字，所以在使用中，必须使用 try…catch 进行异常的处理。例如，现在要在 D 盘上创建一个 test.txt 的文件。

范例：创建新文件

```
package org.lxh.demo12.filedemo;
import java.io.File;
```

```
import java.io.IOException;
public class FileDemo01 {
    public static void main(String args[]) {
        File f = new File("d:\\test.txt") ;     // 必须给出完整路径
        try {
            f.createNewFile() ;                 // 根据给定的路径创建新文件
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

运行程序之后可以发现，在 D 盘中已经创建好了 test.txt 文件。如果在不同的操作系统中，则路径的分隔符表示是不一样的，例如：

- Windows 中使用反斜杆表示目录的分隔符“\”。
- Linux 中使用正斜杠表示目录的分隔符“/”。

那么，既然 Java 程序本身具有可移植性的特点，则在编写路径时最好可以根据程序所在的操作系统自动使用符合本地操作系统要求的分隔符，这样才能达到可移植性的目的。要实现这样的功能，则就需要观察 File 类中提供的两个常量。

范例：观察 File 类中提供的两个常量

```
package org.lxh.demo12.filedemo;
import java.io.File;
public class FileDemo02 {
    public static void main(String args[]) {
        System.out.println("pathSeparator: " + File.pathSeparator);
                                                        // 调用静态常量
        System.out.println("separator: " + File.separator);// 调用静态常量
    }
}
```

程序运行结果：

```
pathSeparator: ;
separator: \
```

所以，对于之前创建文件的操作来讲，最好的做法是使用以上的常量表示路径，代码修改如下。

范例：修改创建文件的代码

```
package org.lxh.demo12.filedemo;
import java.io.File;
import java.io.IOException;
public class FileDemo03 {
    public static void main(String args[]) {
```

```
        String path = "d:" + File.separator + "test.txt";// 拼凑出可以适应操
                                                          作系统的路径
        File f = new File(path);                    // 必须给出路径
        try {
            f.createNewFile();                      // 根据给定的路径创建新文件
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

程序的运行结果与前面的程序一样，但是此时的程序可以在任意的操作系统中使用。

提示：在操作文件时一定要使用 File.separator 表示分隔符。

在程序的开发中，往往会使用 Windows 开发环境，因为在 Windows 操作系统中支持的开发工具较多，使用方便，而在程序发布时往往是直接在 Linux 或其他操作系统上部署，所以这时如果不使用 File.separator，则程序运行就有可能存在问题，这一点读者在日后的开发中一定要有所警惕。

2. 实例操作二：删除一个指定的文件

File 类中也支持删除文件的操作，如果要删除一个文件，则可以使用 File 类中的 delete() 方法。

范例：删除文件

```
package org.lxh.demo12.filedemo;
import java.io.File;
public class FileDemo04 {
    public static void main(String args[]) {
        File f = new File("d:" + File.separator + "test.txt");// 必须给出路径
        f.delete(); // 删除文件
    }
}
```

此时，文件已经删除，但是以上做法也存在问题，即在删除文件前应该保证文件存在，所以以上程序在使用时最好先判断文件是否存在，如果存在，则执行删除操作。判断一个文件是否存在可以直接使用 File 类提供的 exists()方法，此方法返回 boolean 类型。

范例：在删除文件中增加判断

```
package org.lxh.demo12.filedemo;
import java.io.File;
public class FileDemo05 {
    public static void main(String args[]) {
        File f = new File("d:"+File.separator+"test.txt") ;// 必须给出路径
```

```
            if(f.exists()){                                    // 判断文件是否存在
                f.delete() ;                                   // 如果存在，则删除文件
            }
        }
    }
```

以上代码就比较合理了，保证不会删除一个不存在的文件。

3．实例操作三：综合创建和删除文件的操作

现在给定一个文件的路径，如果此文件存在，则将其删除，如果文件不存在则创建一个新的文件。执行流程如图 12-1 所示。

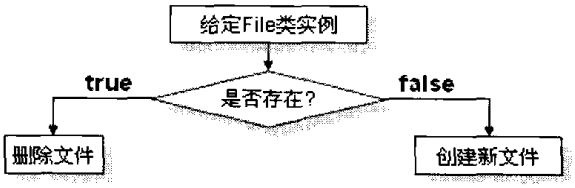


图 12-1　程序执行流程

要想实现这样的功能，则可以把前面的 3 个方法联合起来使用。

范例：实现以上的功能要求

```
package org.lxh.demo12.filedemo;
import java.io.File;
import java.io.IOException;
public class FileDemo06 {
    public static void main(String args[]) {
        File f = new File("d:" + File.separator + "test.txt"); // 文件路径
        if (f.exists()) {                    // 判断文件是否存在
            f.delete();                      // 删除文件
        } else {
            try {
                f.createNewFile();  // 创建文件
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

以上程序已经完成了所要求的功能，但是细心的读者可能会发现以上程序的问题，在每次程序执行完毕之后，文件并不会立刻创建或删除，会有一些延迟，这是因为所有的操作都需要通过 JVM 完成所造成的，如图 12-2 所示。所以读者在进行文件操作时，一定要考虑到延迟的影响。

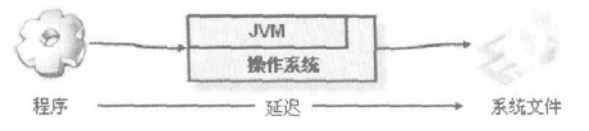


图 12-2　File 操作过程

提示：文件后缀可有可无。

在 Windows 中各个文件都存在后缀，例如，本例中的 text.txt，后缀是.txt，表示是一个文本后缀。但是一个文件的后缀本身并没有任何的意义，即不管有没有后缀并不影响文件本身的内容。而在 Windows 中为了实现程序使用的便捷化管理，所以应将文件的后缀进行比较合理的应用。

4．实例操作四：创建一个文件夹

除了可以创建文件外，也可以使用 File 类指定一个文件夹，直接使用 mkdir()方法就可以完成。

范例：创建文件夹

```
package org.lxh.demo12.filedemo;
import java.io.File;
public class FileDemo07 {
    public static void main(String args[]) {
        File f = new File("d:" + File.separator + "mldn"); // 给出路径
        f.mkdir() ;                                        // 创建文件夹
    }
}
```

5．实例操作五：列出指定目录的全部文件

如果现在给出了一个目录，则可以直接列出目录中的内容。在 File 类中定义了如下两个列出文件夹内容的方法。

- public String[] list()：列出全部名称，返回一个字符串数组。
- public File[] listFiles()：列出完整的路径，返回一个 File 对象数组。

范例：使用 list()方法列出一个目录中的全部内容

```
package org.lxh.demo12.filedemo;
import java.io.File;
public class FileDemo08 {
    public static void main(String args[]) {
        File f = new File("d:" + File.separator);  // 必须给出路径
        String str[] = f.list();                   // 列出给定目录中的内容
        for (int i = 0; i < str.length; i++) {
            System.out.println(str[i]);
        }
    }
}
```

程序运行结果：

```
eclipse
Java
Microsoft Visual Studio 8
MyEclipse 5.5.1 GA
MySQL
oracle
Sybase
Tomcat 5.5
WinRAR
... ... ...
```

从输出结果可以发现，以上列出的只是指定目录中的文件名称，并不是一个文件的完整路径，所以如果想列出每一个文件的完整路径，就必须使用另外一个列出方法——listFiles()。

范例：使用 listFiles()方法列出一个目录中的全部内容

```
package org.lxh.demo12.filedemo;
import java.io.File;
public class FileDemo09 {
    public static void main(String args[]) {
        File f = new File("d:" + File.separator);  // 必须给出路径
        File files[] = f.listFiles();              // 列出全部的文件
        for (int i = 0; i < files.length; i++) {
            System.out.println(files[i]) ;
        }
    }
}
```

程序运行结果：

```
d:\eclipse
d:\Java
d:\Microsoft Visual Studio 8
d:\MyEclipse 5.5.1 GA
d:\MySQL
d:\oracle
d:\Sybase
d:\Tomcat 5.5
d:\WinRAR
... ... ...
```

以上直接打印的是 File 类对象，可以把一个完整的路径取出。两者比较之后可以发现，使用 listFiles()方法列出目录中的内容更加方便。

6. 实例操作六：判断一个给定的路径是否是目录

可以直接使用 isDirectory()方法判断给定的一个路径是否是目录。

范例：判断给定路径是否是目录

```
package org.lxh.demo12.filedemo;
import java.io.File;
public class FileDemo10 {
    public static void main(String args[]) {
        File f = new File("d:" + File.separator);  // 路径
        if (f.isDirectory()) {                     // 判断是否是目录
            System.out.println(f.getPath() + "路径是目录。");
        } else {
            System.out.println(f.getPath() + "路径不是目录。");
        }
    }
}
```

程序运行结果：

```
d:\路径是目录。
```

以上为读者列举了一些 File 类中的常用方法，下面再看一个实例，以加深对 File 类的理解。

12.1.3 范例——列出指定目录的全部内容

给定一个目录，要求列出此目录下的全部内容，因为给定目录可能存在子文件夹，此时要求也可以把所有的子文件夹的子文件列出来。

要先判断给定的路径是否是目录，然后再使用 listFiles()列出一个目录中的全部内容，一个文件夹中可能包含其他的文件或子文件夹，子文件夹中也可能会包含其他的子文件夹，所以此处只能采用递归的调用方式完成。操作的流程如图 12-3 所示。



图 12-3 列出目录中的全部内容

范例：列出指定目录的全部内容

```
package org.lxh.iodemo.filedemo;
import java.io.File;
```

```
public class FileDemo11 {
    public static void main(String args[]) {
        File my = new File("d:" + File.separator); // 操作路径
        print(my) ;
    }
    public static void print(File file) {          // 递归调用此方法
        if (file != null) {                        // 增加一个检查机制
            if (file.isDirectory()) {              // 判断是否是目录
                File f[] = file.listFiles() ;      // 如果是目录，则列出全部的
                                                   内容
                if (f != null) {                   // 有可能无法列出目录中的
                                                   文件
                    for (int i = 0; i < f.length; i++) {
                        print(f[i]);               // 继续列出内容
                    }
                }
            }else{
                System.out.println(file);          // 如果不是目录，则直接打印
                                                   路径信息
            }
        }
    }
}
```

以上程序采用递归的调用形式不断地判断传进来的路径是否是目录，如果是目录，则继续列出子文件夹，如果不是，则直接打印路径名称。

12.2 RandomAccessFile 类

File 类只是针对文件本身进行操作，而如果要对文件内容进行操作，则可以使用 RandomAccessFile 类，此类属于随机读取类，可以随机地读取一个文件中指定位置的数据，例如，现在假设在文件中保存了以下 3 个数据：

- zhangsan，30。
- lisi，31。
- wangwu，32。

那么如果使用 RandomAccessFile 类读取“lisi”信息时，就可以将“zhangsan”的信息跳过，相当于在文件中设置了一个指针，根据此指针的位置进行读取。但是如果想实现这样的功能，则每个数据的长度应该保持一致，所以在设置姓名时应统一设置为 8 位，数字为 4 位，如图 12-4 所示。

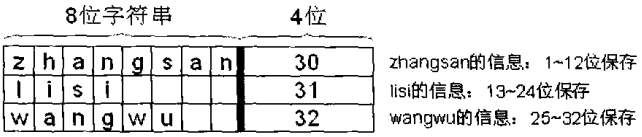


图 12-4　数据的保存

要实现功能，则必须依靠 RandomAccess 中的几种设置模式，然后在构造方法中传递此模式。此类的常用操作方法如表 12-2 所示。

表 12-2　RandomAccessFile 类的常用操作方法

序号	方　法	类型	描　述
1	public RandomAccessFile(File file,String mode) throws FileNotFoundException	构造	接收 File 类的对象，指定操作路径，但是在设置时需要设置模式，r 为只读；w 为只写；rw 为读写
2	public RandomAccessFile(String name,String mode) throws FileNotFoundException	构造	不再使用 File 类对象表示文件，而是直接输入了一个固定的文件路径
3	public void close() throws IOException	普通	关闭操作
4	public int read(byte[] b) throws IOException	普通	将内容读取到一个 byte 数组中
5	public final byte readByte() throws IOException	普通	读取一个字节
6	public final int readInt() throws IOException	普通	从文件中读取整型数据
7	public void seek(long pos) throws IOException	普通	设置读指针的位置
8	public final void writeBytes(String s) throws IOException	普通	将一个字符串写入到文件中，按字节的方式处理
9	public final void writeInt(int v) throws IOException	普通	将一个 int 型数据写入文件，长度为 4 位
10	public int skipBytes(int n) throws IOException	普通	指针跳过多少个字节

需要注意的是，如果使用 rw 方式声明 RandomAccessFile 对象时，要写入的文件不存在，系统将自动进行创建。

12.2.1　使用 RandomAccessFile 类写入数据

下面写入 12.2 节的 3 个数据，为了保证可以进行随机读取，所以写入的名字都是 8 个字节，写入的数字是固定的 4 个字节。

范例：写文件

```
package org.lxh.demo12.randomaccessdemo;
import java.io.File;
import java.io.RandomAccessFile;
public class RandomAccessFileDemo01 {
    // 直接抛出异常，程序中可以不用再分别处理
    public static void main(String[] args) throws Exception {
        File f = new File("d:" + File.separator + "test.txt"); // 指定要操
                                                               作的文件
```

```
        RandomAccessFile rdf = null ;          // 声明一个RandomAccessFile类对象
        rdf = new RandomAccessFile(f,"rw");// 以读写方式打开文件，会自动创建新
                                              文件
        String name = null ;
        int age = 0 ;
        name = "zhangsan" ;                    // 字符串长度为8
        age = 30 ;                             // 数字长度为4
        rdf.writeBytes(name);                  // 将姓名写入文件之中
        rdf.writeInt(age) ;                    // 将年龄写入文件之中
        name = "lisi    " ;                    // 字符串长度为8
        age = 31 ;                             // 数字长度为4
        rdf.writeBytes(name);                  // 将姓名写入文件中
        rdf.writeInt(age) ;                    // 将年龄写入文件中
        name = "wangwu  " ;                    // 字符串长度为8
        age = 32 ;                             // 数字长度为4
        rdf.writeBytes(name);                  // 将姓名写入文件中
        rdf.writeInt(age) ;                    // 将年龄写入文件中
        rdf.close() ;                          // 关闭文件
    }
}
```

写完后可以直接通过 RandomAccess 的方式进行随机读取。

12.2.2 使用 RandomAccessFile 类读取数据

读取时直接使用 r 的模式即可，以只读的方式打开文件。

读取时所有的字符串只能按照 byte 数组的方式读取出来，而且所有的长度是 8 位。

范例：随机读取

```
package org.lxh.demo12.randomaccessdemo;
import java.io.File;
import java.io.RandomAccessFile;
public class RandomAccessFileDemo02 {
    // 直接抛出异常，程序中可以不用再分别处理
    public static void main(String[] args) throws Exception {
        File f = new File("d:" + File.separator + "test.txt");// 指定要操作
                                                                 的文件
        RandomAccessFile rdf = null ;          // 声明一个RandomAccessFile
                                                  类对象
        rdf = new RandomAccessFile(f,"r") ;    // 以读方式打开文件，会自动创建
                                                  新文件
        String name = null ;
        int age = 0 ;
```

```
            byte b[] = new byte[8] ;                    // 准备空间读取姓名
            rdf.skipBytes(12) ;
            for (int i = 0; i < b.length; i++) {
                b[i] = rdf.readByte();                  // 循环读取出前8个内容
            }
            name = new String(b) ;                      // 将读取出来的byte数组变为String
            age = rdf.readInt() ;                       // 读取数字
            System.out.println("第二个人的信息 --> 姓名：" + name + "；年龄：" + age) ;
            rdf.seek(0) ;                               // 指针回到文件的开头
            b = new byte[8] ;                           // 准备空间读取姓名
            for (int i = 0; i < b.length; i++) {
                b[i] = rdf.readByte();                  // 循环读取出前8个内容
            }
            name = new String(b) ;                      // 将读取出来的byte数组变为String
            age = rdf.readInt() ;                       // 读取数字
            System.out.println("第一个人的信息 --> 姓名：" + name + "；年龄：" + age) ;
            rdf.skipBytes(12) ;                         // 跳过第1个人的信息
            b = new byte[8] ;                           // 准备空间读取姓名
            for (int i = 0; i < b.length; i++) {
                b[i] = rdf.readByte();                  // 循环读取出前8个内容
            }
            name = new String(b) ;                      // 将读取出来的byte数组变为String
            age = rdf.readInt() ;                       // 读取数字
            System.out.println("第三个人的信息 --> 姓名：" + name + "；年龄：" + age) ;
            rdf.close() ;                               // 关闭文件
        }
    }
```

程序运行结果：

```
第二个人的信息 --> 姓名：lisi    ；年龄：31
第一个人的信息 --> 姓名：zhangsan；年龄：30
第三个人的信息 --> 姓名：wangwu  ；年龄：32
```

可以发现，程序中可以随机跳过 12 位读取信息，也可以回到开始点重新读取。

随机读写流可以实现对文件内容的操作，但是一般情况下操作文件内容往往会使用字节或字符流。

12.3 字节流与字符流基本操作

在程序中所有的数据都是以流的方式进行传输或保存的，程序需要数据时要使用输入流读取数据，而当程序需要将一些数据保存起来时，就要使用输出流，可以通过图 12-5 表示出输入及输出的关系。

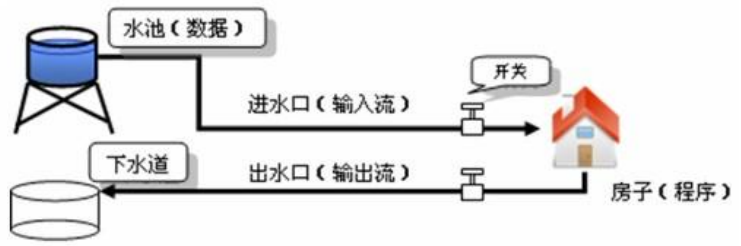


图 12-5　输入、输出的关系

在 java.io 包中流的操作主要有字节流、字符流两大类，两类都有输入和输出操作。在字节流中输出数据主要使用 OutputStream 类完成，输入使用的是 InputStream 类。在字符流中输出主要是使用 Writer 类完成，输入主要是使用 Reader 类完成。

在 Java 中 IO 操作也是有相应步骤的，以文件的操作为例，主要的操作流程如下：

（1）使用 File 类打开一个文件。

（2）通过字节流或字符流的子类指定输出的位置。

（3）进行读/写操作。

（4）关闭输入/输出。

下面分别介绍如何使用字节流或字符流保存或读取文件。

12.3.1　字节流

字节流主要操作 byte 类型数据，以 byte 数组为准，主要操作类是 OutputStream 类和 InputStream 类。

1．字节输出流：OutputStream

OutputStream 是整个 IO 包中字节输出流的最大父类，此类的定义如下：

```
public abstract class OutputStream
extends Object
implements Closeable, Flushable
```

从以上定义中可以发现，OutputStream 类是一个抽象类，如果要使用此类，则首先必须通过子类实例化对象。如果现在要操作的是一个文件，则可以使用 FileOutputStream 类，通过向上转型后，可以为 OutputStream 实例化，在 OutputStream 类中的主要操作方法如表 12-3 所示。

表 12-3　OutputStream 类的常用方法

序号	方　　法	类型	描　　述
1	public void close() throws IOException	普通	关闭输出流
2	public void flush() throws IOException	普通	刷新缓冲区
3	public void write(byte[] b) throws IOException	普通	将一个 byte 数组写入数据流
4	public void write(byte[] b,int off,int len) throws IOException	普通	将一个指定范围的 byte 数组写入数据流
5	public abstract void write(int b) throws IOException	普通	将一个字节数据写入数据流

此时使用 FileOutputStream 子类，此类的构造方法如下：

```
public FileOutputStream(File file) throws FileNotFoundException
```

操作时必须接收 File 类的实例，指明要输出的文件路径。

提示：关于 Closeable 和 Flushable 接口的说明。

在 OutputStream 类的定义中可以发现此类实现了 Closeable 和 Flushable 两个接口，那么这两个接口的定义如下。

Closeable 接口：

```
public interface Closeable{
    void close() throws IOException
}
```

Flushable 接口：

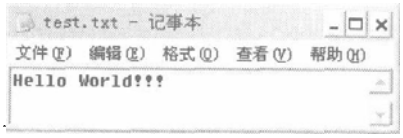
```
public interface Flushable{
    void flush() throws IOException
}
```

这两个接口的作用从其定义方法中可以发现，Closeable 表示可关闭，Flushable 表示可刷新，而且在 OutputStream 类中已经有了这两个方法的实现，所以操作时用户一般不会关心这两个接口，而直接使用 OutputStream 类即可。

范例：向文件中写入字符串

```
package org.lxh.demo12.byteiodemo;
import java.io.File;
import java.io.FileOutputStream;
import java.io.OutputStream;
public class OutputStreamDemo01 {
    public static void main(String[] args) throws Exception {  // 异常抛出，不处理
        // 第1步：使用File类找到一个文件
        File f = new File("d:" + File.separator + "test.txt");  // 声明File对象
        // 第2步：通过子类实例化父类对象
        OutputStream out = null;                // 准备好一个输出的对象
        out = new FileOutputStream(f);          // 通过对象多态性，进行实例化
        // 第3步：进行写操作
        String str = "Hello World!!!";          // 准备一个字符串
        byte b[] = str.getBytes();              // 只能输出byte数组，所以将字符串变为byte数组
        out.write(b);                           // 将内容输出，保存文件
        // 第4步：关闭输出流
        out.close();                            // 关闭输出流
    }
}
```

程序运行结果：



可以发现内容已经成功地写入到文件中，以上程序在实例化、写、关闭时都有异常发生，为了方便起见，直接在主方法上使用 thorws 关键字抛出异常，可以减少 try…catch 语句。

提示：文件不存在则会自动创建。

在以上操作的 test.txt 文件，在操作之前本身是不存在的，但是操作之后程序会为用户自动创建新的文件，并将内容写入到文件之中。

以上操作是直接将一个字符串变为 byte 数组，然后将 byte 数组直接写入到文件中，当然也可以通过循环把每一个字节一个个地写入到文件之中 。

范例：使用 write(int t)的方式写入文件内容

```
package org.lxh.demo12.byteiodemo;
import java.io.File;
import java.io.FileOutputStream;
import java.io.OutputStream;
public class OutputStreamDemo02 {
    public static void main(String[] args) throws Exception { // 异常抛出，不处理
        // 第1步：使用File类找到一个文件
        File f = new File("d:" + File.separator + "test.txt");// 声明File对象
        // 第2步：通过子类实例化父类对象
        OutputStream out = null;                    // 准备好一个输出的对象
        out = new FileOutputStream(f);              // 通过对象多态性，进行实例化
        // 第3步：进行写操作
        String str = "Hello World!!!";              // 准备一个字符串
        byte b[] = str.getBytes();                  // 只能输出byte数组，所以将字符串变为byte数组
        for (int i = 0; i < b.length; i++) {
            out.write(b[i]);                        // 将内容输出
        }
        // 第4步：关闭输出流
        out.close();                                // 关闭输出流
    }
}
```

上面程序是将 byte 数组中的内容一个个地写入到文件之中，实现的功能与上一个程序是一致的。以上两种做法并没有什么不同，两者可以任意使用。

2. 追加新内容

之前的所有操作中，如果重新执行程序，则肯定会覆盖文件中的已有内容，那么此时可以通过 FileOutputStream 向文件中追加内容，FileOutputStream 的另外一个构造方法如下：

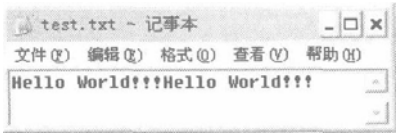
```
public FileOutputStream(File file,boolean append) throws FileNotFoundException
```

在构造方法中，如果将 append 的值设置为 true，则表示在文件的末尾追加内容。

范例：修改之前的程序，追加文件内容

```
package org.lxh.demo12.byteiodemo;
import java.io.File;
import java.io.FileOutputStream;
import java.io.OutputStream;
public class OutputStreamDemo03 {
    public static void main(String[] args) throws Exception { // 异常抛出，不处理
        // 第1步：使用File类找到一个文件
        File f = new File("d:" + File.separator + "test.txt");// 声明File对象
        // 第2步：通过子类实例化父类对象
        OutputStream out = null;                    // 准备好一个输出的对象
        out = new FileOutputStream(f,true);// 此处表示在文件末尾追加内容
        // 第3步：进行写操作
        String str = "Hello World!!!";              // 准备一个字符串
        byte b[] = str.getBytes();                  // 只能输出byte数组，所以将字符串变为byte数组
        for (int i = 0; i < b.length; i++) {
            out.write(b[i]);                        // 将内容输出
        }
        // 第4步：关闭输出流
        out.close();                                // 关闭输出流
    }
}
```

程序运行结果：



可以发现，每次执行后，内容会自动追加到文件的末尾。

❸提问：如何增加换行？

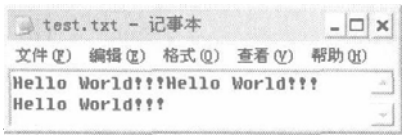
上面程序确实在文件之后追加内容，可是内容是紧跟在原有内容之后的，如何在文件中增加换行，使文件内容显示更加清晰呢？

回答：使用\r\n 增加换行。

如果要换行，则直接在字符串要换行处加入一个“\r\n”。

```
// 第3步：进行写操作
String str = "\r\n Hello World!!!"; // 准备一个字符串
```

程序运行结果：



从结果中发现，新的内容是在换行之后追加的。

3. 字节输入流 InputStream

既然程序可以向文件中写入内容，则可以通过 InputStream 从文件中把内容读取进来。InputStream 类的定义如下：

```
public abstract class InputStream
extends Object
implements Closeable
```

与 OutputStream 类一样，InputStream 本身也是一个抽象类，必须依靠其子类，如果现在从文件中读取，子类肯定是 FileInputStream。InputStream 类中的主要方法如表 12-4 所示。

表 12-4　InputStream 类的常用方法

序号	方　法	类型	描　述
1	public int available() throws IOException	普通	可以取得输入文件的大小
2	public void close() throws IOException	普通	关闭输入流
3	public abstract int read() throws IOException	普通	读取内容，以数字的方式读取
4	public int read(byte[] b) throws IOException	普通	将内容读到 byte 数组中，同时返回读入的个数

FileInputStream 类的构造方法如下：

```
public FileInputStream(File file) throws FileNotFoundException
```

范例：从文件中读取内容

```
package org.lxh.demo12.byteiodemo;
import java.io.File;
import java.io.FileInputStream;
import java.io.InputStream;
public class InputStramDemo01 {
    public static void main(String[] args) throws Exception { // 异常抛出，不处理
        // 第1步：使用File类找到一个文件
```

```
        File f = new File("d:" + File.separator + "test.txt"); // 声明File
                                                                对象
        // 第2步：通过子类实例化父类对象
        InputStream input = null;              // 准备好一个输入的对象
        input = new FileInputStream(f);        // 通过对象多态性进行实例化
        // 第3步：进行读操作
        byte b[] = new byte[1024];             // 所有的内容读到此数组中
        input.read(b);                         // 把内容取出，内容读到byte数组中
        // 第4步：关闭输入流
        input.close();                         // 关闭输入流
        System.out.println("内容为：" + new String(b)); // 把byte数组变为字符
                                                       串输出
    }
}
```

程序运行结果：

```
内容为：Hello World!!!_
（后面会有大量的空格，此处省略其显示）
```

内容已经被读取进来，但是发现后面有很多个空格，这是因为开辟的 byte 数组大小为 1024，而实际的内容只有 14 个字节，也就是说存在 1010 个空白的空间，在将 byte 数组变为字符串时也将这 1010 个无用的空间转为字符串，这样的操作肯定是不合理的。如果要想解决以上的问题，则要观察 read 方法，在此方法上有一个返回值，此返回值表示向数组中写入了多少个数据。

范例：修正以上的错误

```
package org.lxh.demo12.byteiodemo;
import java.io.File;
import java.io.FileInputStream;
import java.io.InputStream;
public class InputStreamDemo02 {
    public static void main(String[] args) throws Exception { // 异常抛出，
                                                              不处理
        // 第1步：使用File类找到一个文件
        File f = new File("d:" + File.separator + "test.txt");// 声明File
                                                              对象
        // 第2步：通过子类实例化父类对象
        InputStream input = null;              // 准备好一个输入的对象
        input = new FileInputStream(f);        // 通过对象多态性进行实例化
        // 第3步：进行读操作
        byte b[] = new byte[1024];             // 所有的内容读到此数组中
        int len = input.read(b);               // 将内容读出
        // 第4步：关闭输入流
        input.close();                         // 关闭输入流
```

```
        System.out.println("读入数据的长度：" + len);
        System.out.println("内容为：" + new String(b, 0, len));// 把byte数组变
                                                              为字符串输出
    }
}
```

程序运行结果：

```
读入数据的长度：14
内容为：Hello World!!!
```

此时，再次运行程序，发现没有多余的空格产生，这是因为程序在最后只是将 byte 数组指定范围中的内容变成了字符串。

注意：注意输入文件的路径。

在使用 FileInputStream 读取时如果指定的路径不存在，则程序运行会出现异常。

以上问题是否有其他的方式解决，因为虽然最后指定了 byte 数组的范围，但是程序依然开辟了很多的无用空间，这样肯定会造成资源的浪费，那么此时能否根据文件的数据量来选择开辟空间的大小呢？要想完成这样的操作，则要从 File 类中着手，因为在 File 类中存在一个 length()的方法，此方法可以取得文件的大小。

范例：开辟指定大小的 byte 数组

```
package org.lxh.demo12.byteiodemo;
import java.io.File;
import java.io.FileInputStream;
import java.io.InputStream;
public class InputStreamDemo03 {
    public static void main(String[] args) throws Exception { // 异常抛出，
                                                              不处理
        // 第1步：使用File类找到一个文件
        File f = new File("d:" + File.separator + "test.txt");// 声明File
                                                              对象
        // 第2步：通过子类实例化父类对象
        InputStream input = null;                    // 准备好一个输入的对象
        input = new FileInputStream(f);              // 通过对象多态性进行实例化
        // 第3步：进行读操作
        byte b[] = new byte[(int)f.length()];        // 所有的内容读到此数组中，数组
                                                        大小由文件决定
        input.read(b);                               // 将内容读出
        // 第4步：关闭输入流
        input.close();                               // 关闭输入流
        System.out.println("内容为：" + new String(b)); // 把byte数组变为字符
                                                        串输出
    }
}
```

程序运行结果：

```
内容为：Hello World!!!
```

除以上方式外，也可以通过循环从文件中一个个地把内容读取进来，直接使用 read() 方法即可。

范例：使用 read()通过循环读取

```
package org.lxh.demo12.byteiodemo;
import java.io.File;
import java.io.FileInputStream;
import java.io.InputStream;
public class InputStreamDemo04 {
    public static void main(String[] args) throws Exception { // 异常抛出，不处理
        // 第1步：使用File类找到一个文件
        File f = new File("d:" + File.separator + "test.txt"); //声明File对象
        // 第2步：通过子类实例化父类对象
        InputStream input = null;                    // 准备好一个输入的对象
        input = new FileInputStream(f);              // 通过对象多态性进行实例化
        // 第3步：进行读操作
        byte b[] = new byte[(int) f.length()]; // 所有的内容读到此数组中
        for (int i = 0; i < b.length; i++) {
            b[i] = (byte)input.read();               // 将内容读出
        }
        // 第4步：关闭输入流
        input.close();                               // 关闭输入流
        System.out.println("内容为：" + new String(b)); // 把byte数组变为字符串输出
    }
}
```

程序运行结果：

```
内容为：Hello World!!!
```

但是，以上程序是在明确知道了具体数组大小的前提下开展的，如果此时不知道要输入的内容有多大，则只能通过判断是否读到文件末尾的方式来读取文件。

范例：另一种方式的读取

```
package org.lxh.demo12.byteiodemo;
import java.io.File;
import java.io.FileInputStream;
import java.io.InputStream;
public class InputStreamDemo05 {
```

```
    public static void main(String[] args) throws Exception { // 异常抛出，不处理
        // 第1步：使用File类找到一个文件
        File f = new File("d:" + File.separator + "test.txt"); // 声明File对象
        // 第2步：通过子类实例化父类对象
        InputStream input = null;                    // 准备好一个输入的对象
        input = new FileInputStream(f);              // 通过对象多态性进行实例化
        // 第3步：进行读操作
        int len = 0;                                 // 用于记录读取的数据个数
        byte b[] = new byte[1024];                   // 所有的内容读到此数组中
        int temp = 0;                                // 接收读取的每一个内容
        while ((temp = input.read()) != -1) {
            // 将每次的读取内容给temp变量，如果temp的值不是-1，则表示文件没有读完
            b[len] = (byte) temp;
            len++;
        }
        // 第4步：关闭输入流
        input.close();                               // 关闭输入流
        System.out.println("内容为：" + new String(b, 0, len));
    }
}
```

程序运行结果：

```
内容为：Hello World!!!
```

以上几种读取字节流的方式，读者最好都掌握，因为随着开发的需要，都有可能使用。

提示：文件读到末尾了，则返回的内容为-1。

以上程序代码中要判断 temp 接收到的内容是否是-1，正常情况下是不会返回-1 的，只有当输入流的内容已经读到底，才会返回这个数字，通过此数字可以判断输入流中是否还有其他内容。

12.3.2　字符流

在程序中一个字符等于两个字节，那么 Java 提供了 Reader 和 Writer 两个专门操作字符流的类。

1. 字符输出流 Writer

Writer 本身是一个字符流的输出类，此类的定义如下：

```
public abstract class Writer
extends Object
implements Appendable, Closeable, Flushable
```

此类本身也是一个抽象类，如果要使用此类，则肯定要使用其子类，此时如果是向文件中写入内容，应该使用 FileWriter 的子类。Wirter 类的常用方法如表 12-5 所示。

表 12-5 Writer 类的常用方法

序　号	方　法	类　型	描　述
1	public abstract void close() throws IOException	普通	关闭输出流
2	public void write(String str) throws IOException	普通	将字符串输出
3	public void write(char[] cbuf) throws IOException	普通	将字符数组输出
4	public abstract void flush() throws IOException	普通	强制性清空缓存

FileWriter 类的构造方法定义如下：

```
public FileWriter(File file) throws IOException
```

提示：关于 Appendable 接口的说明。

在 Writer 类中除了实现 Closeable 和 Flushable 两个接口之外，还实现了一个 Appendable 接口，此接口定义如下：

```
public interface Appendable{
    Appendable append(CharSequence csq) throws IOException ;
    Appendable append(CharSequence csq,int start,int end) throws
IOException ;
    Appendable append(char c) throws IOException
}
```

此接口表示的是内容可以被追加，接收的参数是 CharSequence，实际上 String 类就实现了此接口，所以可以直接通过此接口的方法向输出流中追加内容。

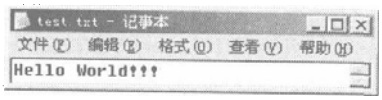
范例：向文件中写入数据

```
package org.lxh.demo12.chariodemo;
import java.io.File;
import java.io.FileWriter;
import java.io.Writer;
public class WriterDemo01 {
    public static void main(String[] args) throws Exception { // 异常抛出，
                                                              不处理
        // 第1步：使用File类找到一个文件
        File f = new File("d:" + File.separator + "test.txt");// 声明File
                                                              对象
        // 第2步：通过子类实例化父类对象
        Writer out = null;                          // 准备好一个输出的对象
        out = new FileWriter(f);                    // 通过对象多态性，进行实例化
        // 第3步：进行写操作
        String str = "Hello World!!!";              // 准备一个字符串
        out.write(str);                             // 将内容输出
```

```
        // 第4步：关闭输出流
        out.close();                                    // 关闭输出流
    }
}
```

整个程序与 OutputStream 的操作流程并没有什么太大的区别，唯一的好处是，可以直接输出字符串，而不用将字符串变为 byte 数组之后再输出。

程序运行结果：



2. 使用 FileWriter 追加文件的内容

在使用字符流操作时，也可以实现文件的追加功能，直接使用 FileWriter 类中的以下构造即可实现追加：

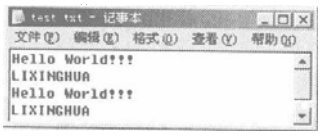
```
public FileWriter(File file,boolean append) throws IOException
```

将 append 的值设置为 true，表示追加。

范例：追加文件内容

```
package org.lxh.demo12.chariodemo;
import java.io.File;
import java.io.FileWriter;
import java.io.Writer;
public class WriterDemo02 {
    public static void main(String[] args) throws Exception {  // 异常抛出，不处理
        // 第1步：使用File类找到一个文件
        File f = new File("d:" + File.separator + "test.txt"); // 声明File对象
        // 第2步：通过子类实例化父类对象
        Writer out = null;                                  // 准备好一个输出的对象
        out = new FileWriter(f,true);                       // 通过对象多态性进行实例化
        // 第3步：进行写操作
        String str = "\r\nLIXINGHUA\r\nHello World!!!" ; // 准备一个字符串
        out.write(str); // 将内容输出
        // 第4步：关闭输出流
        out.close();                                        // 关闭输出流
    }
}
```

程序运行结果：



3. 字符输入流 Reader

Reader 是使用字符的方式从文件中取出数据，Reader 类的定义如下：

```
public abstract class Reader
extends Object
implements Readable, Closeable
```

Reader 本身也是抽象类，如果现在要从文件中读取内容，则可以直接使用 FileReader 子类。Reader 类的常用方法如表 12-6 所示。

表 12-6 Reader 类的常用方法

序号	方　法	类型	描　述
1	public abstract void close() throws IOException	普通	关闭输出流
2	public int read() throws IOException	普通	读取单个字符
3	public int read(char[] cbuf) throws IOException	普通	将内容读到字符数组中，返回读入的长度

FileReader 的构造方法定义如下：

```
public FileReader(File file) throws FileNotFoundException
```

范例：从文件中读取内容

```
package org.lxh.demo12.chariodemo;
import java.io.File;
import java.io.FileReader;
import java.io.Reader;
public class ReaderDemo01 {
    public static void main(String[] args) throws Exception {  // 异常抛出，不处理
        // 第1步：使用File类找到一个文件
        File f = new File("d:" + File.separator + "test.txt");// 声明File对象
        // 第2步：通过子类实例化父类对象
        Reader reader = null;                          // 准备好一个输入的对象
        reader = new FileReader(f);                    // 通过对象多态性进行实例化
        // 第3步：进行读操作
        char c[] = new char[1024];                     // 所有的内容读到此数组中
        int len = reader.read(c);                      // 将内容输出
        // 第4步：关闭输入流
        reader.close();                                // 关闭输入流
        System.out.println("内容为：" + new String(c, 0, len));
                                                       // 把char数组变为字符串输出
    }
}
```

程序运行结果：

```
内容为：Hello World!!!
LIXINGHUA
Hello World!!!
LIXINGHUA
Hello World!!!
```

如果此时不知道数据的长度，也可以像之前操作字节流那样，使用循环的方式进行内容的读取。

范例：使用循环的方式读取内容

```
package org.lxh.demo12.chariodemo;
import java.io.File;
import java.io.FileReader;
import java.io.Reader;
public class ReaderDemo02 {
    public static void main(String[] args) throws Exception {  // 异常抛出，不处理
        // 第1步：使用File类找到一个文件
        File f = new File("d:" + File.separator + "test.txt");// 声明File对象
        // 第2步：通过子类实例化父类对象
        Reader reader = null;                          // 准备好一个输入的对象
        reader = new FileReader(f);                    // 通过对象多态性进行实例化
        // 第3步：进行读操作
        int len = 0;                                   // 用于记录读取的数据个数
        char c[] = new char[1024];                     // 所有的内容读到此数组中
        int temp = 0;                                  // 接收读取的每一个内容
        while ((temp = reader.read()) != -1) {
            // 将每次的读取内容给temp变量，如果temp的值不是-1，则表示文件没有读完
            c[len] = (char) temp;
            len++;
        }
        // 第4步：关闭输入流
        reader.close();                                // 关闭输入流
        System.out.println("内容为：" + new String(c, 0, len));
    }
}
```

程序运行结果：

```
内容为：Hello World!!!
LIXINGHUA
Hello World!!!
```

```
LIXINGHUA
Hello World!!!
```

12.3.3 字节流与字符流的区别

字节流与和字符流的使用非常相似，两者除了操作代码上的不同之外，是否还有其他的不同呢？

实际上字节流在操作时本身不会用到缓冲区（内存），是文件本身直接操作的，而字符流在操作时使用了缓冲区，通过缓冲区再操作文件，如图 12-6 所示。

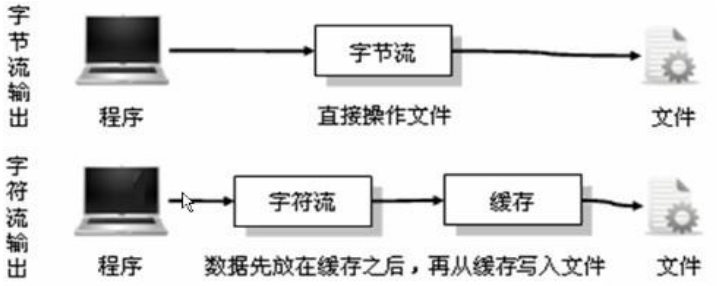


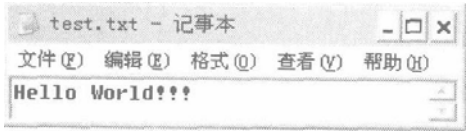
图 12-6 字节流与字符流操作的区别

下面以两个写文件的操作为主进行比较，但是在操作时字节流和字符流的操作完成之后都不关闭输出流。

范例：使用字节流不关闭执行

```
package org.lxh.demo12.byteiodemo;
import java.io.File;
import java.io.FileOutputStream;
import java.io.OutputStream;
public class OutputStreamDemo05 {
    public static void main(String[] args) throws Exception {  // 异常抛出，不处理
        // 第1步：使用File类找到一个文件
        File f = new File("d:" + File.separator + "test.txt");  // 声明File对象
        // 第2步：通过子类实例化父类对象
        OutputStream out = null;                    // 准备好一个输出的对象
        out = new FileOutputStream(f);              // 通过对象多态性进行实例化
        // 第3步：进行写操作
        String str = "Hello World!!!";              // 准备一个字符串
        byte b[] = str.getBytes();                  // 字符串转byte数组
        out.write(b);                               // 将内容输出
        // 第4步：关闭输出流
        // out.close();                             // 此时没有关闭
    }
}
```

程序运行结果：

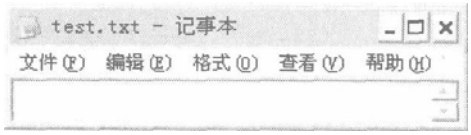


此时没有关闭字节流操作，但是文件中也依然存在了输出的内容，证明字节流是直接操作文件本身的。而下面继续使用字符流完成，再观察效果。

范例：使用字符流不关闭执行

```
package org.lxh.demo12.chariodemo;
import java.io.File;
import java.io.FileWriter;
import java.io.Writer;
public class WriterDemo03 {
    public static void main(String[] args) throws Exception { // 异常抛出，不处理
        // 第1步：使用File类找到一个文件
        File f = new File("d:" + File.separator + "test.txt");// 声明File对象
        // 第2步：通过子类实例化父类对象
        Writer out = null;                          // 准备好一个输出的对象
        out = new FileWriter(f);                    // 通过对象多态性进行实例化
        // 第3步：进行写操作
        String str = "Hello World!!!";              // 准备一个字符串
        out.write(str);                             // 将内容输出
        // 第4步：关闭输出流
        // out.close();                             // 此时没有关闭
    }
}
```

程序运行结果：



程序运行后会发现文件中没有任何内容，这是因为字符流操作时使用了缓冲区，而在关闭字符流时会强制性地将缓冲区中的内容进行输出，但是如果程序没有关闭，则缓冲区中的内容是无法输出的，所以得出结论：字符流使用了缓冲区，而字节流没有使用缓冲区。

提问：什么叫缓冲区？

在很多地方都碰到缓冲区这个名词，那么到底什么是缓冲区？又有什么作用呢？

回答：缓冲区可以简单地理解为一段内存区域。

可以简单地把缓冲区理解为一段特殊的内存。

某些情况下，如果一个程序频繁地操作一个资源（如文件或数据库），则性能会很低，此时为了提升性能，就可以将一部分数据暂时读入到内存的一块区域之中，以后直接从此区域中读取数据即可，因为读取内存速度会比较快，这样可以提升程序的性能。

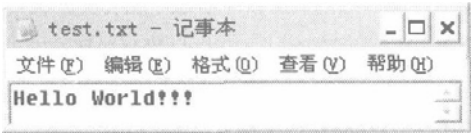
在字符流的操作中，所有的字符都是在内存中形成的，在输出前会将所有的内容暂时保存在内存之中，所以使用了缓冲区暂存数据。

如果想在不关闭时也可以将字符流的内容全部输出，则可以使用 Writer 类中的 flush() 方法完成。

范例：强制性清空缓冲区

```
package org.lxh.demo12.chariodemo;
import java.io.File;
import java.io.FileWriter;
import java.io.Writer;
public class WriterDemo04 {
    public static void main(String[] args) throws Exception { // 异常抛出，不处理
        // 第1步：使用File类找到一个文件
        File f = new File("d:" + File.separator + "test.txt");// 声明File对象
        // 第2步：通过子类实例化父类对象
        Writer out = null;                          // 准备好一个输出的对象
        out = new FileWriter(f);                    // 通过对象多态性进行实例化
        // 第3步：进行写操作
        String str = "Hello World!!!";              // 准备一个字符串
        out.write(str);                             // 将内容输出
        out.flush();                                // 强制性清空缓冲区中的内容
        // 第4步：关闭输出流
        // out.close();                             // 此时没有关闭
    }
}
```

程序运行结果：



此时，文件中已经存在了内容，更进一步证明内容是保存在缓冲区的。这一点在读者日后的开发中要特别引起注意。

提问：使用字节流好还是字符流好？

学习完字节流和字符流的基本操作后，已经大概地明白了操作流程的各个区别，那么在开发中是使用字节流好还是字符流好呢？

回答：使用字节流更好。

在回答之前，先为读者讲解这样的一个概念，所有的文件在硬盘或在传输时都是以字节的方式进行的，包括图片等都是按字节的方式存储的，而字符是只有在内存中才会形成，所以在开发中，字节流使用较为广泛。

12.3.4　范例——文件复制

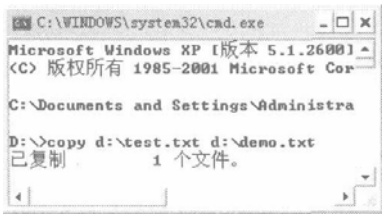
1．实现要求

在 DOS 命令中存在一个文件的复制命令（copy），例如，现在要将 D 盘中的 test.txt 文件复制到 D 盘中的 demo.txt 文件，则只要在命令行中输入 copy 即可完成。

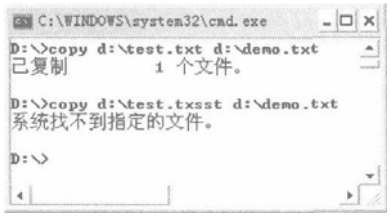
copy 命令的语法格式如下：

```
copy 源文件 目标文件
```

运行效果如下：



如果输入的源文件不存在，则会有以下提示：



下面使用 Java 完成以上功能，程序运行时可以按照如下格式进行：

```
java Copy 源文件 目标文件
```

2．程序实现

从运行格式中可以发现，要想输入源文件或目标文件的路径，可以通过命令行参数完成，但是此时就必须对输入的参数进行验证，如果输入的参数个数不是两个，或者输入的源文件路径不存在，则程序都应该给出错误信息并退出。

此时又面临一个选择，即是使用字节流完成操作还是使用字符流完成操作呢？

因为要复制的文件不一定都是文本文件，也有可能包含图片或声音等，所以如果此时

使用字符流的话肯定不能很好地完成操作，所以必须使用字节流完成，使用 OutputStream 和 InputStream 类。

而且要完成这样的复制程序可以用以下两种方式操作：

- 将源文件中的内容全部读取到内存中，并一次性写入到目标文件中。
- 不将源文件中的内容全部读取进来，而是采用边读边写的方式。

很明显采用第 2 种方式更合理，因为将源文件的内容一次性读取进来的话，如果文件内容过多，则整个内存是无法装下的，程序肯定会出现异常；而如果采用边读边写的方式，则肯定要比全部读进来性能高很多，下面按照此思路完成程序。

范例：实现复制功能

```
package org.lxh.demo12.byteiodemo;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
public class Copy {
    public static void main(String args[]) {
        if (args.length != 2) {                    // 判断是否是两个参数
            System.out.println("输入的参数不正确。");
            System.out.println("例：java Copy 源文件路径 目标文件路径");
            System.exit(1);                        // 系统退出
        }
        File f1 = new File(args[0]);               // 源文件的File对象
        File f2 = new File(args[1]);               // 目标文件的File对象
        if (!f1.exists()) {
            System.out.println("源文件不存在！");
            System.exit(1);
        }
        InputStream input = null;                  // 准备好输入流对象，读取源文件
        OutputStream out = null;                   // 准备好输出流对象，写入目标文件
        try {
            input = new FileInputStream(f1);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        try {
            out = new FileOutputStream(f2);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
```

```
            }
            if (input != null && out != null) {                    // 判断输入或输出是否
                                                                      准备好
                int temp = 0;
                try {
                    while ((temp = input.read()) != -1) {          // 开始复制
                        out.write(temp);                           // 边读边写
                    }
                    System.out.println("复制完成! ");
                } catch (IOException e) {
                    e.printStackTrace();
                    System.out.println("复制失败! ");
                }
                try {
                    input.close();
                    out.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
```

程序运行结果：

（1）没有输入源文件或目标文件

输入的参数不正确。
例：java Copy 源文件路径 目标文件路径

（2）输入的源文件路径不正确

源文件不存在!

（3）正确的执行：java Copy d:\test.txt d:\demo.txt

复制完成!

12.4　转换流——OutputStreamWriter 类与 InputStreamReader 类

整个 IO 包实际上分为字节流和字符流，但是除了这两个流之外，还存在一组字节流-字符流的转换类。

- OutputStreamWriter：是 Writer 的子类，将输出的字符流变为字节流，即将一个字符流的输出对象变为字节流输出对象。

- InputStreamReader：是 Reader 的子类，将输入的字节流变为字符流，即将一个字节流的输入对象变为字符流的输入对象。

如果以文件操作为例，则内存中的字符数据需要通过 OutputStreamWriter 变为字节流才能保存在文件中，读取时需要将读入的字节流通过 InputStreamReader 变为字符流，转换步骤如图 12-7 所示。

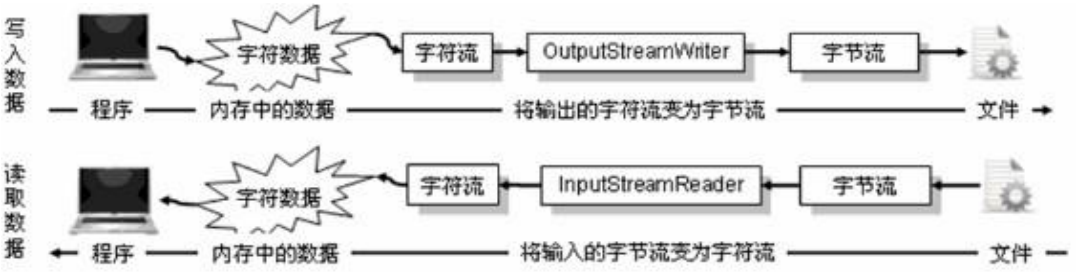


图 12-7　转换步骤

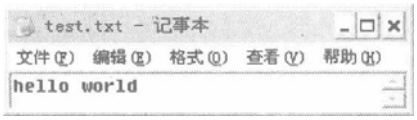
从图 12-7 中可以清楚地发现，不管如何操作，最终全部是以字节的形式保存在文件中。OutputStreamWriter 的构造方法如下：

```
public OutputStreamWriter(OutputStream out)
```

范例：将字节输出流变为字符输出流

```
package org.lxh.iodemo.changeiodemo;
import java.io.File;
import java.io.FileOutputStream;
import java.io.OutputStreamWriter;
import java.io.Writer;
public class OutputStreamWriterDemo01 {
    public static void main(String[] args) throws Exception { // 所有的异常抛出
        File f = new File("d:" + File.separator + "test.txt");
        Writer out = null;
        out = new OutputStreamWriter(new FileOutputStream(f));// 字节流变为字符流
        out.write("hello world");   // 使用字符流输出
        out.close();
    }
}
```

程序运行结果：



范例：将字节输入流变为字符输入流

```
package org.lxh.iodemo.changeiodemo;
import java.io.File;
```

```
import java.io.FileInputStream;
import java.io.InputStreamReader;
import java.io.Reader;
public class InputStreamReaderDemo01 {
public static void main(String[] args) throws Exception { // 所有的异常抛出
        File f = new File("d:" + File.separator + "test.txt");
        Reader reader = null;
        reader = new InputStreamReader(new FileInputStream(f));// 将字节流变为
                                                                字符流
        char c[] = new char[1024];
        int len = reader.read(c);
        reader.close();
        System.out.println(new String(c, 0, len));
    }
}
```

以上两个操作都是将字节流的操作类以字符流的形式进行输出和输入。

提示：FileWriter 和 FileReader 的说明。

从 JDK 文档中可以知道 FileOutputStream 是 OutputStream 的直接子类，FileInputStream 也是 InputStream 的直接子类，但是在字符流文件中的两个操作类却有一些特殊，FileWriter 并不直接是 Writer 的子类，而是 OutputStreamWriter 的子类，而 FileReader 也不直接是 Reader 的子类，是 InputStreamReader 的子类，那么从这两个类的继承关系就可以清楚地发现，不管是使用字节流还是字符流实际上最终都是以字节的形式操作输入/输出流的。

12.5　内存操作流

前面所讲解的程序中输出和输入都是从文件中来的，当然，也可以将输出的位置设置在内存上。此时就要使用 ByteArrayInputStream、ByteArrayOutputStream 来完成输入和输出功能。

ByteArrayInputStream 主要完成将内容写入到内存中，而 ByteArrayOutputStream 的功能主要是将内存中的数据输出，如图 12-8 所示。

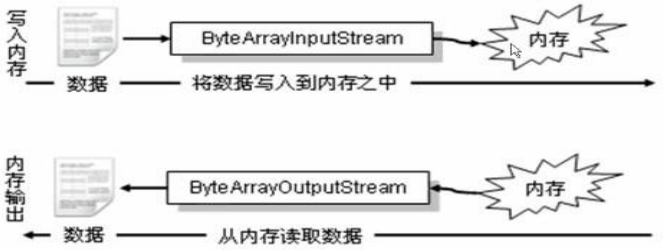


图 12-8　内存操作流

ByteArrayInputStream 类的主要方法如表 12-7 所示。

表 12-7　ByteArrayInputStream 类的主要方法

序　号	方　法	类　型	描　述
1	public ByteArrayInputStream(byte[] buf)	构造	将全部的内容写入内存中
2	public ByteArrayInputStream(byte[] buf,int offset, int length)	构造	将指定范围的内容写入到内存中

ByteArrayInputStream 主要是使用构造方法将全部的内容读取到内存中，如果要想把内容从内存中取出，则可以使用 ByteArrayOutputStream 类，此类的主要方法如表 12-8 所示。

表 12-8　ByteArrayOutputStream 类的主要方法

序　号	方　法	类　型	描　述
1	public ByteArrayOutputStream()	构造	创建对象
2	public void write(int b)	普通	将内容从内存中输出

下面通过以上的若干方法完成内存的 IO 操作。

范例：使用内存操作流完成一个大写字母转换为小写字母的程序

```
package org.lxh.demo12.bytearraydemo;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
public class ByteArrayDemo01 {
    public static void main(String[] args) {
        String str = "HELLOWORLD";                  // 定义串都是大写字母的一个字符
        ByteArrayInputStream bis = null;            // 声明一个内存的输入流
        ByteArrayOutputStream bos = null;           // 声明一个内存的输出流
        bis = new ByteArrayInputStream(str.getBytes());// 向内存中输出内容
        bos = new ByteArrayOutputStream(); // 准备从ByteArrayInputStream中
                                                    读数据
        int temp = 0;
        while ((temp = bis.read()) != -1) {
            char c = (char) temp;                   // 将读取的数字变为字符
            bos.write(Character.toLowerCase(c));    // 将字符变为小写
        }
        String newStr = bos.toString();             // 取出内容
        try {
            bis.close() ;
            bos.close() ;
        } catch (IOException e) {
            e.printStackTrace();
        }
        System.out.println(newStr);
    }
}
```

程序运行结果：

```
helloworld
```

可以发现，字符串已经由大写变为了小写字母，全部的操作都是在内存中完成的。

提示：内存操作流的使用。

内存操作流一般在生成一些临时信息时才会使用，而这些临时信息如果要保存在文件中，则代码执行完后肯定还要删除这个临时文件，那么此时使用内存操作流是最合适的。

12.6 管 道 流

管道流的主要作用是可以进行两个线程间的通信，如图 12-9 所示，分为管道输出流（PipedOutputStream）和管道输入流（PipedInputStream）。如果要进行管道输出，则必须把输出流连在输入流上，在 PipedOutputStream 类上有如下方法用于连接管道。

```
public void connect(PipedInputStream snk) throws IOException
```

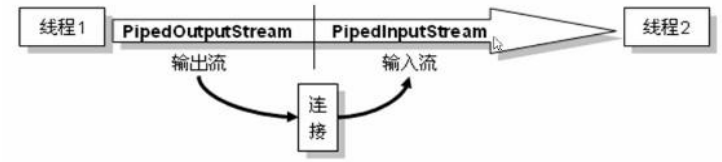


图 12-9 管道流操作

范例：验证管道流

```
package org.lxh.demo12.pipeddemo;
import java.io.IOException;
import java.io.PipedInputStream;
import java.io.PipedOutputStream;
class Send implements Runnable {                    // 实现Runnable接口
    private PipedOutputStream pos = null;  // 管道输出流
    public Send() {
        this.pos = new PipedOutputStream();// 实例化输出流
    }
    public void run() {
        String str = "Hello World!!!";
        try {
            this.pos.write(str.getBytes());// 输出信息
        } catch (IOException e) {
            e.printStackTrace();
        }
        try {
            this.pos.close();                        // 关闭输出流
        } catch (IOException e) {
            e.printStackTrace();
```

```
        }
    }
    public PipedOutputStream getPos() {     // 通过线程类得到输出流
        return pos;
    }
}
class Receive implements Runnable {         // 实现Runnable接口
    private PipedInputStream pis = null;
    public Receive() {
        this.pis = new PipedInputStream(); // 实例化输入流
    }
    public void run() {
        byte b[] = new byte[1024];
        int len = 0;
        try {
            len = this.pis.read(b);          // 接收数据
        } catch (IOException e) {
            e.printStackTrace();
        }
        try {
            this.pis.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
        System.out.println("接收的内容为: " + new String(b, 0, len));
    }
    public PipedInputStream getPis() {
        return pis;
    }
}
public class PipedDemo {
    public static void main(String[] args) {
        Send s = new Send();
        Receive r = new Receive();
        try {
            s.getPos().connect(r.getPis()); // 连接管道
        } catch (IOException e) {
            e.printStackTrace();
        }
        new Thread(s).start();               // 启动线程
        new Thread(r).start();               // 启动线程
    }
}
```

程序运行结果：

```
接收的内容为：Hello World!!!
```

以上程序定义了两个线程对象，在发送的线程类中定义了管道输出流，在接收的线程类中定义了管道的输入流，在操作时只需要使用 PipedOutputStream 类中提供的 connection() 方法就可以将两个线程管道连接在一起，线程启动后会自动进行管道的输入、输出操作。

12.7　打　印　流

12.7.1　打印流的基本操作

在整个 IO 包中，打印流是输出信息最方便的类，主要包含字节打印流（PrintStream）和字符打印流（PrintWriter）。打印流提供了非常方便的打印功能，可以打印任何的数据类型，如小数、整数、字符串等。本章主要使用字节打印流（PrintStream）进行讲解。

PrintStream 是 OutputStream 的子类，PrintStream 类的常用方法如表 12-9 所示。

表 12-9　PrintStream 类的常用方法

序号	方　　法	类型	描　　述
1	public PrintStream(File file) throws FileNotFoundException	构造	通过一个 File 对象实例化 PrintStream 类
2	public PrintStream(OutputStream out)	构造	接收 OutputStream 对象，实例化 PrintStream 类
3	public PrintStream printf(Locale l,String format,Object... args)	普通	根据指定的 Locale 进行格式化输出
4	public PrintStream printf(String format, Object... args)	普通	根据本地环境格式化输出
5	public void print(boolean b)	普通	此方法被重载很多次，输出任意数据
6	public void println(boolean b)	普通	此方法被重载很多次，输出任意数据后换行

在 PrintStream 类中定义的构造方法可以清楚地发现，有一个构造方法可以直接接收 OutputStream 类的实例，这是因为与 OutputStream 类相比，PrintStream 类可以更加方便地输出数据，这就好像将 OutputStream 类重新包装了一下，使之输出更加方便，如图 12-10 所示。



图 12-10　打印流操作

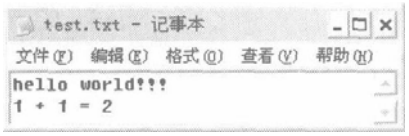
从图 12-10 中可以发现，把一个输出流的实例传递到打印流后，可以更加方便地输出内容，也就是说，是打印流把输出流重新装饰了一下，就像送别人礼物，需要把礼物包装一

下才会更加好看，所以，这样的设计称为装饰设计模式。

范例：使用 PrintStream 输出

```
package org.lxh.demo12.printdemo;
import java.io.File;
import java.io.FileOutputStream;
import java.io.PrintStream;
public class PrintDemo01 {
    public static void main(String[] args) throws Exception {// 所有异常抛出
        PrintStream ps = null;
        // 此时通过FileOutputStream实例化，意味着所有的输出是向文件中打印
        ps = new PrintStream(new FileOutputStream(new File("d:"
                + File.separator + "test.txt")));
        ps.print("hello ");
        ps.println("world!!!");
        ps.print("1 + 1 = " + 2);
        ps.close() ;
    }
}
```

程序运行结果：



以上程序代码在输出内容时与使用 OutputStream 直接输出相比，明显方便了许多。

12.7.2 使用打印流进行格式化

在 JDK 1.5 之后，Java 又对 PrintStream 类进行了扩充，增加了格式化的输出方式，直接使用 printf()方法可以完成操作。但是在进行格式化输出时需要指定其输出的数据类型，数据类型的格式化表示如表 12-10 所示。

表 12-10 格式化输出

序　号	字　符	描　述
1	%s	表示内容为字符串
2	%d	表示内容为整数
3	%f	表示内容为小数
4	%c	表示内容为字符

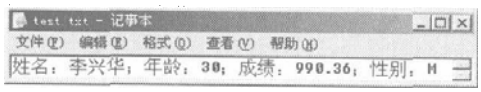
下面使用表 12-10 中的内容进行格式化输出操作。

范例：格式化输出

```
package org.lxh.demo12.printdemo;
import java.io.File;
```

```
import java.io.FileOutputStream;
import java.io.PrintStream;
public class PrintDemo02 {
    public static void main(String[] args) throws Exception { // 所有异常抛出
        PrintStream ps = null;
        // 此时通过FileOutputStream实例化，意味着所有的输出是向文件中打印
        ps = new PrintStream(new FileOutputStream(new File("d:"
                + File.separator + "test.txt")));
        String name = "李兴华" ;                    // 定义字符串
        int age = 30 ;                              // 定义整数
        float score = 990.356f ;                    // 定义小数
        char sex = 'M' ;                            // 定义字符
        // 格式化输出，字符串使用%s、整数使用%d、小数使用%f、字符使用%c
        ps.printf("姓名：%s；年龄：%d；成绩：%f；性别：%c", name, age, score, sex);
        ps.close() ;
    }
}
```

程序运行结果：





以上程序在执行时会将相应的内容替换成具体的内容，如图 12-11 所示。

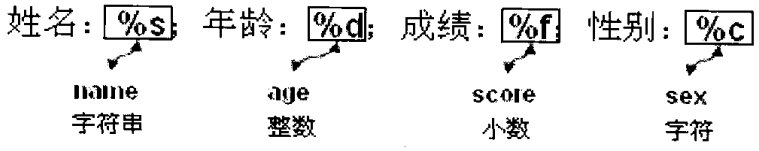


图 12-11　输出格式化

以上程序的%d、%f 和%c 可以全部使用“%s”代替，代码如下所示：

```
ps.printf("姓名：%s；年龄：%s；成绩：%s；性别：%s", name,age,score,sex) ;
```

运行结果如下：



12.8　System 类对 IO 的支持

System 表示系统类，此类在讲解 Java 常用类库时已经介绍过，实际上在 Java 中 System 类也对 IO 给予了一定的支持，在 System 类中定义了如表 12-11 所示的 3 个常量，这 3 个常量在 IO 操作中有着非常大的作用。

表 12-11　System 类的常量

序　号	System 类的常量	描　述
1	public static final PrintStream out	对应系统标准输出，一般是显示器
2	public static final PrintStream err	错误信息输出
3	public static final InputStream in	对应着标准输入，一般是键盘

表 12-11 中的 3 个常量是经常使用的，下面进行详细介绍。

提示：System 类中的 3 个常量也不符合命名规则。

System 类中提供的 in、out、err 3 个常量如果按照 Java 的命名要求来讲，全部字母是应该大写的，但是此处使用的是小写，这些都是 Java 历史发展的产物。

12.8.1　System.out

System.out 是 PrintStream 的对象，在 PrintStream 中定义了一系列的 print()和 println()方法，所以前面使用的 System.out.print()或 System.out.println()语句调用的实际上就是 PrintStream 类的方法。

既然此对象表示的是向显示器上输出，而 PrintStream 又是 OutputStream 的子类。所以可以直接利用此对象向屏幕上输出信息。

范例：使用 OutputStream 向屏幕上输出

```
package org.lxh.demo12.systemdemo;
import java.io.IOException;
import java.io.OutputStream;
public class SystemDemo01 {
    public static void main(String[] args) {
        OutputStream out = System.out ;                    // 此时的输出流是向屏
                                                              幕上输出
        try {
            out.write("hello world!!!".getBytes()) ;       // 向屏幕上输出
        } catch (IOException e) {
            e.printStackTrace();
        }
        try {
            out.close() ;                                  // 关闭输出流
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

程序运行结果：

```
hello world!!!
```

以上信息是直接使用 OutputStream 向屏幕上进行输出的，也就是说，OutputStream 的哪个子类为其实例化，就具备了向哪里输出的能力。如果是使用了 FileOutputStream 则表示向文件输出，如果使用了 System.out 则表示向显示器输出，这里完全显示出了 Java 的多态性的好处，即根据子类的不同完成的功能也不同。

12.8.2 System.err

System.err 表示的是错误信息输出，如果程序出现错误，则可以直接使用 System.err 进行输出，代码如下所示。

范例：错误信息输出

```
package org.lxh.demo12.systemdemo;
public class SystemDemo02 {
    public static void main(String[] args) {
        String str = "hello" ;                          // 声明一个非数字的字符串
        try {
            System.out.println(Integer.parseInt(str)) ;
        } catch (Exception e) {
            System.err.println(e);
        }
    }
}
```

程序运行结果：

```
java.lang.NumberFormatException: For input string: "hello"
```

以上程序要把字符串“hello”变为整型数据，这样肯定会引发 NumberFormatException 异常信息，所以，当捕捉到异常后，直接在 catch 中使用 System.err 进行信息的输出。

提问：System.err 的功能似乎与 System.out 的功能是一样的。

以上操作代码中，如果将 catch 中的 System.err 换成 System.out，输出结果怎么是完全一样的呢？

修改 catch 中的代码：

```
System.out.println(e);      // 此处使用 System.out
```

程序运行结果：

```
java.lang.NumberFormatException: For input string: "hello"
```

既然输出结果是一样的，为什么还要分为两个，该使用哪一个更好呢？

回答：只能从概念上解释 System.err 和 System.out。

从表 12-11 中可以发现，System.out 和 System.err 都是 PrintStream 的实例化对象，而且通过实例代码可以发现，两者都可以输出错误信息，但是一般来讲 System.out 是将信息显示给用户看，是正常的信息显示，而 System.err 的信息正好相反，是不希望用户看到的，会直接在后台打印，是专门显示错误的，如图 12-12 所示。

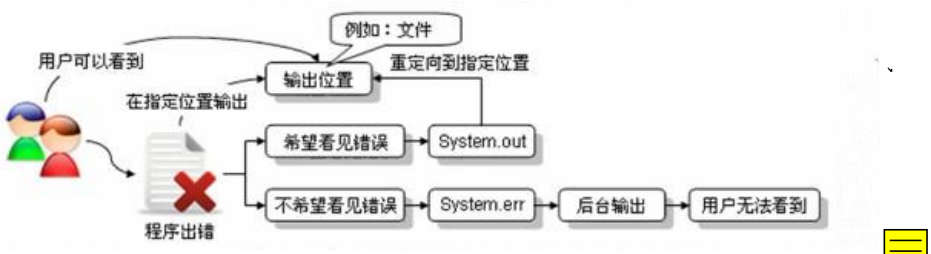


图 12-12　System.out 和 System.err 的区别

一般来讲，如果要输出错误信息，最好不要使用 System.out，而直接使用 System.err，这一点只能从其概念上划分。如果读者现在是使用 Eclipse 开发工具开发，则可以发现，使用 System.err 打印的异常信息是红色的，而使用 System.out 打印的异常信息是普通颜色的，当然，这只是在开发工具层次上的支持。

图 12-12 所示的“重定向”操作将在后面的章节中为读者介绍。

12.8.3　System.in

System.in 实际上是一个键盘的输入流，其本身是 InputStream 类型的对象。那么此时就可以利用 System.in 完成从键盘读取数据的功能。

范例：从键盘上读取数据

```
package org.lxh.demo12.systemdemo;
import java.io.InputStream;
public class SystemDemo04 {
    public static void main(String[] args) throws Exception {  // 所有异常
                                                                抛出
        InputStream input = System.in;                          // 从键盘接收数据
        byte b[] = new byte[1024];                              // 开辟空间，接收数据
        System.out.print("请输入内容：");                        // 信息提示
        int len = input.read(b);                                // 接收数据
        System.out.println("输入的内容为：" + new String(b, 0, len));
        input.close();                                          // 关闭输入流
    }
}
```

程序运行结果：

```
请输入内容：李兴华，北京MLDN软件实训中心
输入的内容为：李兴华，北京MLDN软件实训中心
```

可以发现，以上的代码已经实现了从键盘中输入数据的功能，但这个程序是否可以正常使用呢？

实际上在本程序存在以下两个问题：

- 指定了输入数据的长度，如果现在输入的数据超出了其长度范围，则只能输入部分数据。
- 如果指定的 byte 数组长度是奇数，则还有可能出现中文乱码。

为了验证以上两个问题，现在将程序中开辟的 byte 数组减小长度，同时将长度设置为奇数：

```
byte b[] = new byte[5];
```

运行修改后的程序，同时输入同样的数据就可以发现问题，运行结果如下：

```
请输入内容：李兴华，北京MLDN软件实训中心
输入的内容为：李兴?
```

以上运行结果出现了“李兴？”，这是因为一个中文等于两个字节，所以最后一个字因为只剩下了一个字节，则只能装下半个，这样就不知道是什么了，所以显示为“?”，此过程如图 12-13 所示。

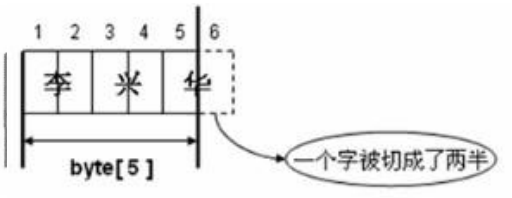


图 12-13　内容存放

既然怕指定的数组长度不够，那么如果此时不指定 byte 数组长度呢？是否可以完成输入的要求呢？下面通过一段代码来观察不指定 byte 数组长度接收数据的情况。

范例：不指定大小

```
package org.lxh.demo12.systemdemo;
import java.io.InputStream;
public class SystemDemo05 {
    public static void main(String[] args) throws Exception{// 所有异常抛出
        InputStream input = System.in;                  // 通过键盘接收数据
        StringBuffer buf = new StringBuffer(); // 声明StringBuffer用于接收
                                                        数据
        System.out.print("请输入内容：");
        int temp = 0;
        while ((temp = input.read()) != -1) {   // 循环接收
            char c = (char) temp;                       // 将数据变为字符
            if (c == '\n') {
                break;                                  // 退出循环，按Enter键表示输入
                                                        完成
            }
```

```
            buf.append(c);                                    // 追加数据
        }
        System.out.println("输入的内容为：" + buf);
        input.close();                                        // 关闭输入流
    }
}
```

此时输入的全部是字母，运行结果如下：

```
请输入内容：www.mldn.cn,lixinghua
输入的内容为：www.mldn.cn,lixinghua
```

此时输入的是中文，运行结果如下：

```
请输入内容：李兴华，北京MLDN软件实训中心
输入的内容为：????????¡À¡À??MLDN?¨ª??????????
```

以上程序中如果输入的是英文字母，没有任何的问题，而如果输入的是中文，则同样会产生乱码，这是因为数据是以一个个字节的方式读进来的，一个汉字是分两次读取的，所以造成了乱码。

指定大小会出现空间限制，不指定大小则输入中文时又会产生乱码，那么怎样的输入数据形式才是最合理的呢？

最好的输入方式是将全部输入的数据暂时放到一块内存中，然后一次性从内存中读取出数据，如图 12-14 所示，这样所有数据只读了一次，则不会造成乱码，而且也不会受长度的限制。如果要完成这样的操作则可以使用第 12.8.4 节中的 BufferedReader 类完成。

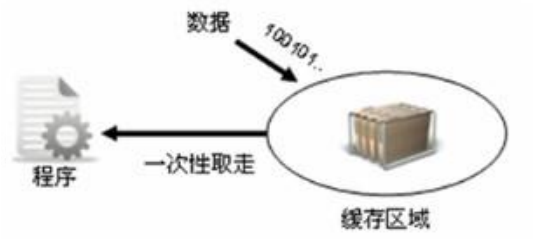


图 12-14　数据读取

12.8.4　输入/输出重定向

从前面的操作中读者已经了解了 System.out、System.err、System.in 3 个常量的作用，但是通过 System 类也可以改变 System.in 的输入流来源以及 System.out 和 System.err 两个输出流的输出位置，这些操作方法如表 12-12 所示。

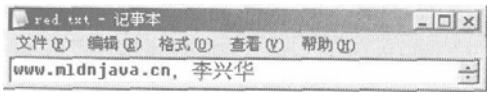
表 12-12　System 类提供的重定向方法

序　号	方　法	类　型	描　述
1	public static void setOut(PrintStream out)	普通	重定向“标准”输出流
2	public static void setErr(PrintStream err)	普通	重定向“标准”错误输出流
3	public,static void setIn(InputStream in)	普通	重定向“标准”输入流

范例：为 System.out 输出重定向

```
package org.lxh.demo12.systemdemo;
import java.io.File;
import java.io.FileOutputStream;
import java.io.PrintStream;
public class SystemDemo06 {
    public static void main(String[] args) throws Exception {// 所有异常抛出
        System.setOut(new PrintStream(new FileOutputStream("d:"
                + File.separator + "red.txt"))); // System.out输出重定向
        System.out.print("www.mldnjava.cn") ;     // 输出时，不再向屏幕上输出
        System.out.println(", 李兴华") ;          // 而是向指定的重定向位置输出
    }
}
```

程序运行结果：



以上程序运行后，所有使用 System.out 输出的信息不会再在屏幕上显示，而是直接将信息保存到“d:\red.txt”文件中。

实际上，此时可以使用此概念完成图 12-12 所示的情况，把需要用户看到的错误信息记录在文件中。

范例：为用户保存错误信息

```
package org.lxh.demo12.systemdemo;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.PrintStream;
public class SystemDemo07 {
    public static void main(String[] args) {
        String str = "hello";                          // 声明一个非数字的字符串
        try {
            System.out.println(Integer.parseInt(str));
        } catch (Exception e) {
            try {
                System.setOut(new PrintStream(new FileOutputStream("d:"
                        + File.separator + "err.log")));// 输出重定位
            } catch (FileNotFoundException e1) {
                e1.printStackTrace();
            }
            System.out.println(e) ;                    // 输出错误，保存到文件中
        }
```

```
        }
    }
```

程序运行结果：



以上代码将产生 NumberFormatException，因为程序中使用了 setOut 方法将其输出重定向到了“d:\err.log”文件，所以所有的错误信息将直接在 err.log 文件中保存。

除了可以为 System.out 重定向输出位置，也可以为 System.err 重定向输出位置，代码如下所示。

范例：为 System.err 输出重定向

```
package org.lxh.demo12.systemdemo;
import java.io.ByteArrayOutputStream;
import java.io.PrintStream;
public class SystemDemo08 {
    public static void main(String[] args) throws Exception {// 所有异常抛出
        ByteArrayOutputStream bos = null ;          // 定义内存输出流
        bos = new ByteArrayOutputStream() ;         // 实例化内存输出流
        System.setErr(new PrintStream(bos));        // System.err输出重定向
        System.err.print("www.mldnjava.cn") ;       // 错误输出时，不再向屏幕上输出
        System.err.println(", 李兴华") ;            // 而是向内存的位置输出
        System.out.println(bos);                    // 打印错误信息
    }
}
```

程序运行结果：

```
www.mldnjava.cn, 李兴华
```

以上操作将 System.err 的输出定位到内存输出流中，所以此时使用 System.err 完成的各个输出操作的输出结果都保存在内存中。

在使用以上两种操作时一定要记住一点：setOut()方法只负责 System.out 的输出重定向，而 setErr()方法只负责 System.err 的输出重定向，两者不可混用。

提示：不要修改 System.err 的输出重定向。

虽然在 System 类中提供了 setErr()这个错误输出的重定向方法，但是一般情况下，读者不要使用这个方法修改 System.err 的重定向，因为从概念上讲 System.err 的错误信息是不希望用户看到的。

以上两个操作完成了对输出的重定向，但是从表 12-12 中可以发现，不只可以对输出进行重定向，对输入也可以。现在假设在 D 盘上有一个 demo.txt 文件，将 System.in 的输入设置到从文件中读取，这样当使用 System.in 时就会从文件流中读取信息，而不是从键盘读取。demo.txt 文件的内容如图 12-15 所示。

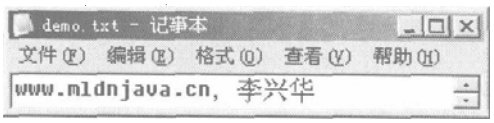


图 12-15　demo.txt 文件的内容

范例：设置 System.in 的输入重定向

```
package org.lxh.demo12.systemdemo;
import java.io.File;
import java.io.FileInputStream;
import java.io.InputStream;
public class SystemDemo09 {
    public static void main(String[] args) throws Exception {// 所有异常抛出
        System.setIn(new FileInputStream("d:"
                + File.separator + "demo.txt"));          // 设置输入重定向
        InputStream input = System.in;                    // 从文件中接收数据
        byte b[] = new byte[1024];                        // 开辟空间，接收数据
        int len = input.read(b);                          // 接收数据
        System.out.println("输入的内容为：" + new String(b, 0, len));
        input.close();                                    // 关闭输入流
    }
}
```

程序运行结果：

```
输入的内容为：www.mldnjava.cn，李兴华
```

以上代码修改了 System.in 的输入位置，而将其输入重定向到从文件中读取，所以读取时会将文件中的内容读取进来。

12.9　BufferedReader 类

BufferedReader 类用于从缓冲区中读取内容，所有的输入字节数据都将放在缓冲区中，常用的方法如表 12-13 所示。

表 12-13　BufferedReader 类的常用方法

序号	方　法	类型	描　述
1	public BufferedReader(Reader in)	构造	接收一个 Reader 类的实例
2	public String readLine() throws IOException	普通	一次性从缓冲区中将内容全部读取进来

BufferedReader 中定义的构造方法只能接收字符输入流的实例，所以必须使用字符输入流和字节输入流的转换类 InputStreamReader 将字节输入流 System.in 变为字符流，如图 12-16 所示。

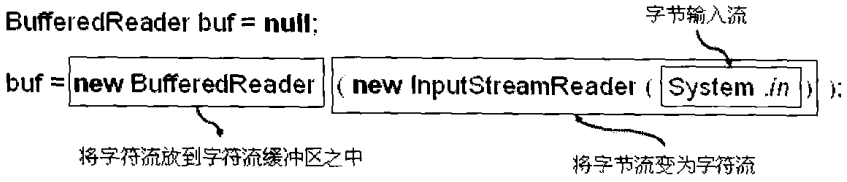


图 12-16　BufferedReader 实例化

提示：代码的说明。

BufferedReader 只能接收字符流的缓冲区，因为每一个中文要占两个字节，所以需要将 System.in 这个字节的输入流变为字符的输入流。

下面使用以上方式完成输入数据的基本操作。

12.9.1　键盘输入数据的标准格式

将 System.in 变为字符流放入到 BufferedReader 后，可以通过 readLine()方法等待用户输入信息。具体的操作代码如下。

范例：从键盘输入数据

```
package org.lxh.demo12.buffdemo;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
public class BufferedReaderDemo01 {
    public static void main(String[] args) {
        BufferedReader buf = null;                    // 声明BufferedReader对象
        buf = new BufferedReader(new InputStreamReader(System.in));
                                                      // 实例化BufferedReader
        String str = null;                            // 接收输入内容
        System.out.print("请输入内容：");
        try {
            str = buf.readLine();                     // 读取输入内容
        } catch (IOException e) {                     // 要进行异常处理
            e.printStackTrace();
        }
        System.out.println("输入的内容为：" + str);
    }
}
```

程序运行结果：

```
请输入内容：李兴华，北京MLDN软件实训中心
输入的内容为：李兴华，北京MLDN软件实训中心
```

可以发现，程序非但没有了长度的限制，也可以正确地接收中文了，所以以上代码就是键盘输入数据的标准格式。

下面通过一些实例来加强对操作的理解。

12.9.2　相关操作实例

1. 实例操作一：加法操作

现在要求从键盘输入两个数字，然后完成两个整数的加法操作。因为从键盘接收过来的内容全部是采用字符串的形式存放的，所以直接将字符串通过包装类 Integer 将字符串变为基本数据类型。

（1）完成最基本的功能

范例：输入两个数字，并让两个数字相加

```
package org.lxh.demo12.execdemo;
import java.io.BufferedReader;
import java.io.InputStreamReader;
public class ExecDemo01 {
    public static void main(String[] args) throws Exception {
        int i = 0;
        int j = 0;
        BufferedReader buf = null;                    // 接收键盘的输入数据
        buf = new BufferedReader(new InputStreamReader(System.in));
        String str = null;                            // 准备接收数据
        System.out.print("请输入第一个数字：");
        str = buf.readLine();
        i = Integer.parseInt(str);                    // 将字符串变为int型
        System.out.print("请输入第二个数字：");
        str = buf.readLine();
        j = Integer.parseInt(str);                    // 将字符串变为int型
        System.out.println(i + " + " + j + " = " + (i + j));
    }
}
```

程序运行结果：

```
请输入第一个数字：3
请输入第二个数字：5
3 + 5 = 8
```

以上程序已经实现了题目所要求的内容，但是还存在以下几个问题：

- 如果输入的字符串不是数字，则肯定无法转换，会出现数字格式化异常，所以在转换时应该使用正则进行验证，如果验证成功了，则表示可以进行转换；而如果验证失败了，表示无法进行转换，则要等待用户重新输入数字才可以。
- 只能输入整数。
- 代码重复，只要输入数据，则肯定使用 BufferedReader，重复出现 readLine()调用。

（2）对类进行合理的划分

对于输入数据，最常见的可能是整数、小数、日期、字符串，所以此时最好将其设计一个专门的输入数据类，完成输入数据的功能。

范例：完成一个专门处理输入数据的类，只能得到整数和字符串

```
package org.lxh.demo12.execdemo;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
public class InputData {
    private BufferedReader buf = null;
    public InputData() {                                    // 在类的构造方法中实例化
                                                               BufferedReader对象
        this.buf = new BufferedReader(new InputStreamReader(System.in));
    }
    public String getString(String info) {                  // 从此方法中得到字符串的信息
        String temp = null;
        System.out.print(info);                             // 打印提示信息
        try {
            temp = this.buf.readLine();
        } catch (IOException e) {
            e.printStackTrace();
        }
        return temp;
    }
    public int getInt(String info,String err) { // 得到一个整数的输入数据
        int temp = 0;
        String str = null;
        boolean flag = true;                                // 定义一个循环的处理标志
        while (flag) {
            str = this.getString(info);
            if (str.matches("^\\d+$")) {                    // 判断输入的是否是数字
                temp = Integer.parseInt(str);               // 将字符串变为数字
                flag = false;                               // 更改标志位之后，将退出循环
            } else {
                System.out.println(err);                    // 出现错误，则打印传递进的错误
                                                               信息
            }
        }
        return temp;
    }
}
```

范例：直接使用以上类即可输入数字

```
package org.lxh.demo12.execdemo;
public class ExecDemo02 {
    public static void main(String[] args) throws Exception {
        int i = 0;
        int j = 0;
        InputData input = new InputData() ;
        i = input.getInt("请输入第一个数字：","输入的数据必须是数字，请重新输入！") ;
        j = input.getInt("请输入第二个数字：","输入的数据必须是数字，请重新输入！") ;
        System.out.println(i + " + " + j + " = " + (i + j));
    }
}
```

程序运行结果：

```
请输入第一个数字：hello
输入的数据必须是数字，请重新输入！
请输入第一个数字：3
请输入第二个数字：5
3 + 5 = 8
```

使用以上程序将输入数据的操作定义成了一个类，以后只要是想得到输入数据，则直接从此类中得到即可。

（3）对输入数据类进一步扩充

在开发中最常见的输入数据类型就是整数、小数、字符串、日期，下面进一步扩充 InputData 类，输入各种类型的数据。

范例：扩充 InputData

```
package org.lxh.demo12.execdemo;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
public class InputData {
    private BufferedReader buf = null;
    public InputData() {                                    // 在类的构造方法中实例化
                                                               BufferedReader对象
        this.buf = new BufferedReader(new InputStreamReader(System.in));
    }
    public String getString(String info) {                  // 从此方法中得到字符串的信息
        String temp = null;
        System.out.print(info);                             // 打印提示信息
```

```
        try {
            temp = this.buf.readLine();
        } catch (IOException e) {
            e.printStackTrace();
        }
        return temp;
    }
    public int getInt(String info,String err) { // 得到一个整数的输入数据
        int temp = 0;
        String str = null;
        boolean flag = true;                    // 定义一个循环的处理标志
        while (flag) {
            str = this.getString(info);
            if (str.matches("^\\d+$")) {        // 判断输入的是否是数字
                temp = Integer.parseInt(str);   // 将字符串变为数字
                flag = false;                   // 更改标志位之后，将退出循环
            } else {
                System.out.println(err);        // 出现错误，则打印传递进的错误
                                                   信息
            }
        }
        return temp;
    }
    public float getFloat(String info,String err) { // 得到一个小数的输入数据
        float temp = 0;
        String str = null;
        boolean flag = true;                    // 定义一个循环的处理标志
        while (flag) {
            str = this.getString(info);
            if (str.matches("^\\d+.?\\d+$")) { // 判断是否是小数
                temp = Float.parseFloat(str);
                flag = false;                   // 更改标志位之后，将退出循环
            } else {
                System.out.println(err);
            }
        }
        return temp;
    }
    public Date getDate(String info,String err) {// 得到一个日期的数据
        Date d = null;
        String str = null;
        boolean flag = true;                    // 定义一个循环的处理标志
        while (flag) {
```

```
            str = this.getString(info);
            if (str.matches("^\\d{4}-\\d{2}-\\d{2}$")) {
                SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
                try {
                    d = sdf.parse(str);          // 将字符串变为Date型数据
                } catch (ParseException e) {
                    e.printStackTrace();
                }
                flag = false;                    // 更改标志位之后，将退出循环
            } else {
                System.out.println(err);
            }
        }
        return d;
    }
}
```

以上程序可以实现整数、小数、字符串、日期类型数据的输入。在得到日期类型时使用了 SimpleDateFormat 类，并指定了日期的转换模板，将一个字符串变为了一个 Date 类型的数据，这一点在开发中较为常用。

提示：注意实际问题的解决方法。

对于实际的开发来讲，很难一次性开发出完整的类设计，所以读者在编写时一定要首先完成基本功能的实现，然后再对实现功能的代码结构进行优化，这样就可以设计一个比较合理的类。

2. 实例操作二：菜单显示

菜单显示在系统中是经常出现的，下面使用 IO 操作完成一个简单的菜单程序，显示效果如下：

```
==== xxx系统 =====
[1]、增加数据
[2]、删除数据
[3]、修改数据
[4]、查看数据
[0]、系统退出

请选择：
```

如果用户输入的编号不正确，则要给出错误提示，并等待用户重新选择。

可以使用 switch 完成以上功能，以上程序本身需要接收输入数据，而且需要显示，在以后还可能在程序中加入具体的操作。为了应对这种情况，中间最好加入一个操作类，即菜单类调用操作类，而具体的实际操作由操作类完成。本程序的输入数据程序依然使用之

前的 InputData 完成。

范例：完成操作类

```
package org.lxh.demo12.execdemo;
public class Operate {
    public static void add() {      // 增加操作
        System.out.println("**选择的是增加操作");
    }
    public static void delete() {   // 删除操作
        System.out.println("**选择的是删除操作");
    }
    public static void update() {   // 修改操作
        System.out.println("**选择的是更新操作");
    }
    public static void find() {     // 查看操作
        System.out.println("**选择的是查看操作");
    }
}
```

操作类的代码比较简单，因为程序本身的功能要求只是实现菜单，如果要完成具体的操作，直接修改此类即可。

范例：菜单显示类，接收选择的数据，同时使用 switch 判断是哪个操作

```
package org.lxh.demo12.execdemo;
public class Menu {
    public Menu() {
        while (true) {
            this.show();                        // 无限制调用菜单的显示
        }
    }
    public void show() {
        System.out.println("==== Xxx系统 =====");
        System.out.println("  [1]、增加数据");
        System.out.println("  [2]、删除数据");
        System.out.println("  [3]、修改数据");
        System.out.println("  [4]、查看数据");
        System.out.println("  [0]、系统退出\n");
        InputData input = new InputData();
        int i = input.getInt("请选择：","请输入正确的选项！");
        switch (i) {
            case 1: {
                Operate.add();                  // 调用操作类的增加操作
                break;
            }
```

```
                case 2: {
                    Operate.delete();              // 调用操作类的删除操作
                    break;
                }
                case 3: {
                    Operate.update();              // 调用操作类的更新操作
                    break;
                }
                case 4: {
                    Operate.find();                // 调用操作类的查看操作
                    break;
                }
                case 0: {
                    System.exit(1);                // 系统退出
                    break;
                }
                default: {
                    System.out.println("请选择正确的操作！");
                }
            }
        }
    }
```

以上的菜单类因为菜单的内容要不断地显示，所以使用循环打印的方式，每一次操作完成后都会重新显示出所有的菜单内容以供用户选择。

范例：编写主方法验证以上的菜单

```
package org.lxh.demo12.execdemo;
public class ExecDemo03 {
    public static void main(String[] args) throws Exception {
        new Menu();                    // 显示菜单
    }
}
```

程序运行结果：

```
==== Xxx系统 =====
  [1]、增加数据
  [2]、删除数据
  [3]、修改数据
  [4]、查看数据
  [0]、系统退出

请选择：x
请输入正确的选项！
```

```
请选择：1
**选择的是增加操作
==== Xxx系统 =====
  [1]、增加数据
  [2]、删除数据
  [3]、修改数据
  [4]、查看数据
  [0]、系统退出

请选择：9
请选择正确的操作！
==== Xxx系统 =====
  [1]、增加数据
  [2]、删除数据
  [3]、修改数据
  [4]、查看数据
  [0]、系统退出

请选择：
```



12.10 Scanner 类

12.10.1 Scanner 类简介

在 JDK 1.5 之后 Java 提供了专门的输入数据类，此类不只可以完成输入数据操作，也可以方便地对输入数据进行验证。此类存放在 java.util 包中，其常用方法如表 12-14 所示。

表 12-14 Scanner 类的常用方法

序号	方　法	类型	描　述
1	public Scanner(File source) throws FileNotFoundException	构造	从文件中接收内容
2	public Scanner(InputStream source)	构造	从指定的字节输入流中接收内容
3	public boolean hasNext(Pattern pattern)	普通	判断输入的数据是否符合指定的正则标准
4	public boolean hasNextInt()	普通	判断输入的是否是整数
5	public boolean hasNextFloat()	普通	判断输入的是否是小数
6	public String next()	普通	接收内容
7	public String next(Pattern pattern)	普通	接收内容，进行正则验证
8	public int nextInt()	普通	接收数字
9	public float nextFloat()	普通	接收小数
10	public Scanner useDelimiter(String pattern)	普通	设置读取的分隔符

下面通过以上方法完成信息的输入功能。

提示：Scanner 类可以接收任意的输入流。

在 Scanner 类中提供了一个可以接收 InputStream 类型的构造方法，这就表示只要是字节输入流的子类都可以通过 Scanner 类进行方便的读取。

12.10.2　使用 Scanner 类输入数据

1. 实例操作一：实现基本的数据输入

最简单的数据输入直接使用 Scanner 类的 next()方法即可。

范例：输入数据

```
package org.lxh.demo12.scannerdemo;
import java.util.Scanner;
public class ScannerDemo01 {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in); // 从键盘接收数据
        System.out.print("输入数据：");
        String str = scan.next();
        System.out.println("输入的数据为："+str);
    }
}
```

程序运行结果：

```
输入数据：helloworld
输入的数据为：helloworld
```

但是，如果在以上程序中输入了带有空格的内容，则只能取出空格之前的数据，代码如下所示：

```
输入数据：hello world !!!
输入的数据为：hello
```

从运行结果中可以发现，空格后的数据没有了，造成这样的结果是因为 Scanner 将空格当作了一个分隔符，所以为了保证程序的正确，可以将分隔符号修改为“\n（回车）”。

范例：修改输入数据的分隔符

```
package org.lxh.demo12.scannerdemo;
import java.util.Scanner;
public class ScannerDemo02 {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in); // 从键盘接收数据
        scan.useDelimiter("\n") ;              // 修改输入数据的分隔符
        System.out.print("输入数据：");
        String str = scan.next();
```

```
            System.out.println("输入的数据为："+str);
        }
    }
```

程序运行结果：

```
输入数据：hello world
输入的数据为：hello world
```

以上代码完成了字符串内容的输入，如果要输入 int 或 float 类型的数据，在 Scanner 类中也有支持，但是在输入之前最好先使用 hasNextXxx()方法进行验证，代码如下所示。

范例：输入 int、float

```
package org.lxh.demo12.scannerdemo;
import java.util.Scanner;
public class ScannerDemo03 {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in); // 从键盘接收数据
        int i = 0 ;
        float f = 0.0f ;
        System.out.print("输入整数：");
        if(scan.hasNextInt()){                          // 判断输入的是否是整数
            i = scan.nextInt() ;                        // 接收整数
            System.out.println("整数数据：" + i) ;
        }else{                                          // 输入错误的信息
            System.out.println("输入的不是整数！") ;
        }
        System.out.print("输入小数：");
        if(scan.hasNextFloat()){                        // 判断输入的是否是小数
            f = scan.nextFloat() ;                      // 接收小数
            System.out.println("小数数据：" + f) ;
        }else{                                          // 输入错误的信息
            System.out.println("输入的不是小数！") ;
        }
    }
}
```

程序运行结果：

```
输入整数：3
整数数据：3
输入小数：5
小数数据：5.0
```

2. 实例操作二：实现日期格式的数据输入

在 Scanner 类中没有提供专门的日期格式输入操作，所以，如果想得到一个日期类型的数据，则必须自己编写正则验证，并手工转换。以下代码演示了具体的操作。

范例：得到日期

```
package org.lxh.demo12.scannerdemo;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Scanner;
public class ScannerDemo04 {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);          // 从键盘接收数据
        System.out.print("输入日期（yyyy-MM-dd）：");
        String str = null;
        Date date = null ;
        if (scan.hasNext("^\\d{4}-\\d{2}-\\d{2}$")) { // 判断输入格式是否是
                                                          日期
            str = scan.next("^\\d{4}-\\d{2}-\\d{2}$");// 接收日期格式的字符串
            try {                                     // 转换成日期
                date = new SimpleDateFormat("yyyy-MM-dd").parse(str) ;
            } catch (ParseException e) {
                e.printStackTrace();
            }
        }else{
            System.out.println("输入的日期格式错误！");
        }
        System.out.println(date);
    }
}
```

程序运行结果：

```
输入日期（yyyy-MM-dd）：2008-08-13
Wed Aug 13 00:00:00 CST 2008
```

以上程序使用 hasNext()对输入的数据进行正则验证，如果合法，则转换成 Date 类型。

3．实例操作三：从文件中得到数据

如果要从文件中取得数据，则直接将 File 类的实例传入到 Scanner 的构造方法中即可。例如，现在要显示“d:\test.txt”中的内容，则可以采用以下的代码，此文件的内容如图 12-17 所示。

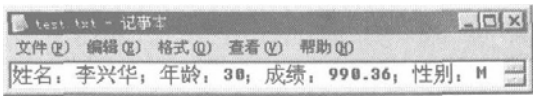


图 12-17　test.txt 文件的内容

范例：读取 test.txt 文件

```
package org.lxh.demo12.scannerdemo;
import java.io.File;
import java.io.FileNotFoundException;
```

```
import java.util.Scanner;
public class ScannerDemo05 {
    public static void main(String[] args) {
        File f = new File("D:" + File.separator + "test.txt"); // 指定操作
                                                                文件
        Scanner scan = null;
        try {
            scan = new Scanner(f);                    // 从文件接收数据
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        StringBuffer str = new StringBuffer();        // 用于接收数据
        while (scan.hasNext()) {                      // 判断是否还有内容
            str.append(scan.next()).append("\n");     // 取出内容
        }
        System.out.println(str);
    }
}
```

程序运行结果：

```
姓名：李兴华；年龄：30；成绩：990.356；性别：M
```

从 Scanner 类的操作中可以发现，Scanner 类有默认的分隔符，这样如果在文件中存在换行，则表示一次输入结束，所以本程序采用循环的方式读取，并在每次读完一行之后加入换行符，因为读取时内容需要反复修改，所以使用 StringBuffer 类以提升操作性能。

12.11 数据操作流

在 IO 包中，提供了两个与平台无关的数据操作流，分别为数据输出流（DataOutputStream）和数据输入流（DataInputStream），通常数据输出流会按照一定的格式将数据输出，再通过数据输入流按照一定的格式将数据读入，这样可以方便地对数据进行处理。

例如，有表 12-15 所示的一组表示订单的数据。

表 12-15 订单数据

商品名称	商品价格	商品数量
衬衣	98.3	3
手套	30.3	2
围巾	50.5	1

如果要将以上数据保存到文件中，就可以使用数据输出流将内容保存到文件，然后再使用数据输入流从文件中读取进来。

12.11.1　DataOutputStream 类

DataOutputStream 是 OutputStream 的子类，此类的定义如下：

```
public class DataOutputStream extends FilterOutputStream implements DataOutput
```

此类继承自 FilterOutputStream 类（FilterOutputStream 是 OutputStream 的子类），同时实现了 DataOutput 接口，在 DataOutput 接口定义了一系列的写入各种数据的方法。

提示：DataOutput 接口的作用。

DataOutput 是数据的输出接口，其中定义了各种数据的输出操作方法，例如，在 DataOutputStream 类中的各种 writeXxx()方法就是此接口定义的，但是在数据输出时一般都会直接使用 DataOutputStream，只有在对象序列化时才有可能直接操作到此接口，这一点将在讲解 Externalizable 接口时为读者介绍。

DataOutputStream 类的常用方法如表 12-16 所示。

表 12-16　DataOutputStream 类的常用方法

序号	方　法	类型	描　述
1	public DataOutputStream(OutputStream out)	构造	实例化对象
2	public final void writeInt(int v) throws IOException	普通	将一个 int 值以 4-byte 值形式写入基础输出流中
3	public final void writeDouble(double v) throws IOException	普通	写入一个 double 类型，该值以 8-byte 值形式写入基础输出流中
4	public final void writeChars(String s) throws IOException	普通	将一个字符串写入到输出流中
5	public final void writeChar(int v) throws IOException	普通	将一个字符写入到输出流中

范例：将订单数据写入到文件 order.txt 中

```
package org.lxh.demo12.datademo;
import java.io.DataOutputStream;
import java.io.File;
import java.io.FileOutputStream;
public class DataOutputStreamDemo {
    public static void main(String[] args) throws Exception {
        DataOutputStream dos = null ;                          // 声明数据输出流对象
        File f = new File("D:" + File.separator + "order.txt");// 指定文件的保存路径
        dos = new DataOutputStream(new FileOutputStream(f)) ; // 实例化数据输出流对象
```

```
        String names[] = { "衬衣", "手套", "围巾" };          // 商品名称
    float prices[] = { 98.3f, 30.3f, 50.5f };              // 商品价格
        int nums[] = { 3, 2, 1 };                          // 商品数量
        for (int i = 0; i < names.length; i++) {           // 循环写入
            dos.writeChars(names[i]) ;                     // 写入字符串
            dos.writeChar('\t') ;                          // 加入分隔符
            dos.writeFloat(prices[i]) ;                    // 写入小数
            dos.writeChar('\t') ;                          // 加入分隔符
            dos.writeInt(nums[i]) ;                        // 写入整数
            dos.writeChar('\n') ;                          // 换行
        }
        dos.close() ;                                      // 关闭输出流
    }
}
```

以上程序的结果中每条数据之间使用“\n”分隔，每条数据中的每个内容之间使用“\t”分隔，如图 12-18 所示。写入后就可以利用 DataInputStream 将内容读取进来。

商品	分隔符	价格	分隔符	数量	分隔符
衬衣	\t	98.3f	\t	3	\n
手套	\t	30.3f	\t	2	\n
围巾	\t	50.5f	\t	1	\n

图 12-18　数据的存储

12.11.2　DataInputStream 类

DataInputStream 是 InputStream 的子类，专门负责读取使用 DataOutputStream 输出的数据，此类的定义如下：

```
public class DataInputStream extends FilterInputStream implements DataInput
```

此类继承自 FilterInputStream 类（FilterInputStream 是 InputStream 的子类），同时实现 DataInput 接口，在 DataInput 接口中定义了一系列读入各种数据的方法。

提示：DataInput 接口的作用。

DataInput 接口是读取数据的操作接口，与 DataOutput 接口提供的各种 writerXxx()方法对应，在此接口中定义了一系列的 readXxx()方法，这些方法在 DataInputStream 类中都有实现。一般在操作时不会直接使用到此接口，而主要使用 DataInputStream 类完成读取功能，只有在对象序列化时才有可能直接利用此接口读取数据，这一点在讲解 Externalizable 接口时再为读者介绍。

DataInputStream 类的常用方法如表 12-17 所示。

表 12-17　DataInputStream 类的常用方法

序　号	方　法	类　型	描　述
1	public DataInputStream(InputStream in)	构造	实例化对象
2	public final int readInt() throws IOException	普通	从输入流中读取整数
3	public final float readFloat() throws IOException	普通	从输入流中读取小数
4	public final char readChar() throws IOException	普通	从输入流中读取一个字符

范例：从 order.txt 中读取数据

```
package org.lxh.demo12.datademo;
import java.io.DataInputStream;
import java.io.File;
import java.io.FileInputStream;
public class DataInputStreamDemo {
    public static void main(String[] args) throws Exception {
        DataInputStream dis = null ;                    // 声明数据输入流对象
        File f = new File("D:" + File.separator + "order.txt");// 要读取的文件
        dis = new DataInputStream(new FileInputStream(f)) ;  // 实例化数据输
                                                              入流对象
        String name = null ;                            // 接收名称
        float price = 0.0f ;                            // 接收价格
        int num = 0 ;                                   // 接收数量
        char temp[] = null ;                            // 接收字符串数据
        char c = 0 ;                                    // 声明字符变量
        int len = 0 ;                                   // 接收读取数据
        try{
            while(true){                                // 循环读取
                temp = new char[200] ;                  // 开辟空间
                len = 0 ;
                while((c=dis.readChar())!='\t'){        // 读取字符
                    temp[len] = c;                      // 接收内容
                    len++ ;                             // 读取长度加1
                }
                name = new String(temp,0,len) ;         // 将字符数组变为String
                price = dis.readFloat() ;               // 读取float
                dis.readChar() ;                        // 读出\t
                num = dis.readInt() ;                   // 读取int
                dis.readChar() ;                        // 读出\n
                System.out.printf("名称：%s；价格：%5.2f；数量：%d\n", name,
                price, num);
            }
        }catch(Exception e){                            // 如果读到底，则会出现异常
```

```
            }
            dis.close() ;                                    // 关闭输出流
        }
    }
```

程序运行结果：

```
名称：衬衣；价格：98.30；数量：3
名称：手套；价格：30.30；数量：2
名称：围巾；价格：50.50；数量：1
```

在使用数据输入流读取时，因为每条记录之间使用“\t”作为分隔，每行记录之间使用“\n”作为分隔，所以要分别使用 readChar()读取这两个分隔符，才能将数据正确地还原。

12.12 合 并 流

合并流的主要功能是将两个文件的内容合并成一个文件，如图 12-19 所示。



图 12-19 合并流

如果要实现合并流，则必须使用 SequenceInputStream 类，此类的常用方法如表 12-18 所示。

表 12-18 SequenceInputStream 类的常用方法

序 号	方 法	类 型	描 述
1	public SequenceInputStream(InputStream s1,InputStream s2)	构造	使用两个输入流对象实例化本类对象
2	public int available() throws IOException	普通	返回文件大小

范例：合并两个文件

```
package org.lxh.demo12.seqdemo;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.SequenceInputStream;
```

```
public class SequenceDemo {
    public static void main(String args[]) throws Exception {
        InputStream is1 = null;                          // 输入流1
        InputStream is2 = null;                          // 输入流2
        OutputStream os = null;                          // 输出流
        SequenceInputStream sis = null;                  // 合并流
        is1 = new FileInputStream("d:" + File.separator + "a.txt");
        is2 = new FileInputStream("d:" + File.separator + "b.txt");
        os = new FileOutputStream("d:" + File.separator + "ab.txt");
        sis = new SequenceInputStream(is1, is2);         // 实例化合并流
        int temp = 0;
        while ((temp = sis.read()) != -1) {              // 循环输出
            os.write(temp);                              // 保存内容
        }
        sis.close();                                     // 关闭合并流
        is1.close();                                     // 关闭输入流1
        is2.close();                                     // 关闭输入流2
        os.close();                                      // 关闭输出流
    }
}
```

以上程序在实例化 SequenceInputStream 类时指定了两个输入流，所以 SequenceInputStream 类在进行读取时实际上是从两个输入流中一起读取内容的。

12.13 压 缩 流

在日常的使用中经常会使用到 WinRAR 或 WinZIP 等压缩文件，通过这些软件可以把一个很大的文件进行压缩以方便传输，如图 12-20 所示。在 Java 中为了减少传输时的数据量也提供了专门的压缩流，可以将文件或文件夹压缩成 ZIP、JAR、GZIP 等文件形式。

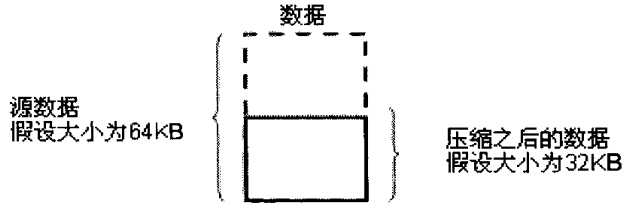


图 12-20 数据压缩

12.13.1 ZIP 压缩输入/输出流简介

ZIP 是一种较为常见的压缩形式，在 Java 中要实现 ZIP 的压缩需要导入 java.util.zip 包，可以使用此包中的 ZipFile、ZipOutputStream、ZipInputStream 和 ZipEntry 几个类完成操作。

提示：JAR 及 GZIP 文件格式的压缩输入、输出流。

在 Java IO 中，不仅可以实现 ZIP 压缩格式的输入、输出，也可以实现 JAR 及 GZIP 文件格式的压缩。

JAR 压缩的支持类保存在 java.util.jar 包中，常用类有如下几个。

- JAR 压缩输出流：JarOutputStream。
- JAR 压缩输入流：JarInputStream。
- JAR 文件：JARFile。
- JAR 实体：JAREntry。

GZIP 是用于 UNIX 系统的文件压缩，在 Linux 中经常会使用到*.gz 的文件，就是 GZIP 格式，GZIP 压缩的支持类保存在 java.util.zip 包中，常用类有如下两个。

- GZIP 压缩输出流：GZIPOutputStream。
- GZIP 压缩输入流：GZIPInputStream。

在每一个压缩文件中都会存在多个子文件，那么每一个子文件在 Java 中就使用 ZipEntry 表示，如图 12-21 所示，ZipEntry 类的常用方法如表 12-19 所示。



图 12-21　压缩文件

表 12-19　ZipEntry 类的常用方法

序　号	方　法	类　型	描　述
1	public ZipEntry(String name)	构造	创建对象并指定要创建的 ZipEntry 名称
2	public boolean isDirectory()	普通	判断此 ZipEntry 是否是目录

下面通过一些实例来了解如何压缩及解压缩 ZIP 文件。

注意：压缩的输入/输出类定义在 java.util.zip 包中。

压缩的输入/输出流也属于 InputStream 或 OutputStream 的子类，但是却没有定义在 java.io 包中，而是以一种工具类的形式提供的，在操作时还需要使用 java.io 包的支持。

12.13.2　ZipOutputStream 类

如果要完成一个文件或文件夹的压缩，则要使用 ZipOutputStream 类。ZipOutputStream 是 OutputStream 的子类，常用操作方法如表 12-20 所示。

表 12-20　ZipOutputStream 类的常用方法

序号	方　法	类型	描　述
1	public ZipOutputStream(OutputStream out)	构造	创建新的 ZIP 输出流
2	public void putNextEntry(ZipEntry e) throws IOException	普通	设置每一个 ZipEntry 对象
3	public void setComment(String comment)	普通	设置 ZIP 文件的注释

现在假设在 D 盘中存在一个 mldn.txt 文件，文件内容如图 12-22 所示，要将其压缩成 mldn.zip 文件，具体代码如下。

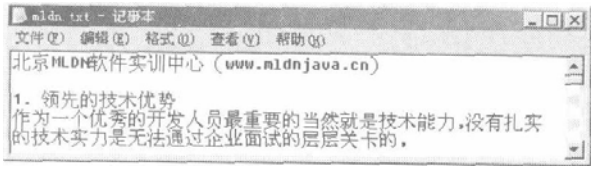


图 12-22　mldn.txt 文件的内容

范例：压缩 mldn.zip 文件

```
package org.lxh.demo12.zipdemo;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.util.zip.ZipEntry;
import java.util.zip.ZipOutputStream;
public class ZipOutputStreamDemo01 {
    public static void main(String[] args) throws Exception {
                                                    // 所有异常抛出
        File file = new File("d:" + File.separator + "mldn.txt");
                                                    // 定义要压缩的文件
        File zipFile = new File("d:" + File.separator + "mldn.zip");
                                                    // 定义压缩文件名称
        InputStream input = new FileInputStream(file);
                                                    // 定义输入文件流
        ZipOutputStream zipOut = null;              // 定义压缩输出流
        // 实例化压缩输出流对象，并指定压缩文件的输出路径
        zipOut = new ZipOutputStream(new FileOutputStream(zipFile));
        // 每一个被压缩的文件都用ZipEntry表示，需要为每一个压缩后的文件设置名称
        zipOut.putNextEntry(new ZipEntry(file.getName()));
                                                    // 创建ZipEntry
        zipOut.setComment("www.mldnjava.cn");       // 设置注释
        int temp = 0;                               // 接收输入的数据
        while ((temp = input.read()) != -1) {       // 读取内容
            zipOut.write(temp);                     // 压缩输出内容
```

```
            }
            input.close();                                    // 关闭输入流
            zipOut.close();                                   // 关闭压缩输出流
        }
    }
```

以上程序将 mldn.txt 作为源文件，然后使用 ZipOutputStream 将所有的压缩数据输出到 mldn.zip 文件中，在压缩时同样采用了边读边写的方式完成：

```
            while ((temp = input.read()) != -1) {             // 读取内容
                zipOut.write(temp);                           // 压缩输出内容
            }
```

程序运行后，会在 D 盘上创建一个 mldn.zip 的压缩文件，文件打开如图 12-23 所示。

上面是对一个文件进行压缩，但是在日常的开发中，往往需要对一个文件夹进行压缩，例如，现在在 D 盘中存在一个 mldn 的文件夹，如图 12-24 所示。

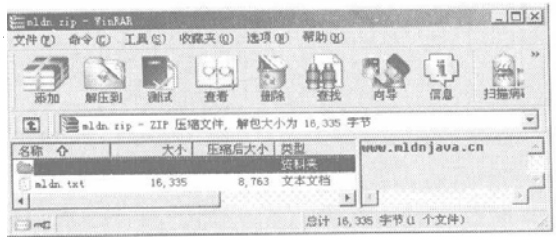


图 12-23　创建后的压缩文件

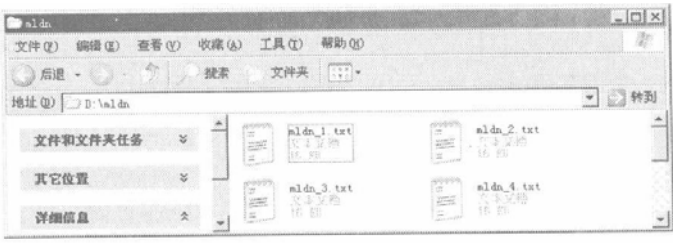


图 12-24　要压缩的文件夹

从使用各种压缩软件的经验来看，如果现在要进行压缩，则在压缩后的文件中应该存在一个 mldn 文件夹。在文件夹中应该存放着各个压缩文件。所以，在实现时就应该列出文件夹中的全部内容，并把每一个内容设置成 ZipEntry 对象，保存到压缩文件中，执行流程如图 12-25 所示。

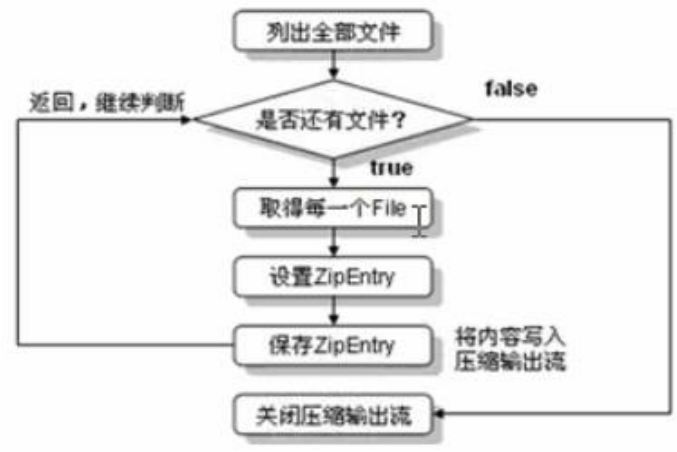


图 12-25　执行流程

范例：压缩一个文件夹

```
package org.lxh.demo12.zipdemo;
import java.io.File;
```

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.util.zip.ZipEntry;
import java.util.zip.ZipOutputStream;
public class ZipOutputStreamDemo02 {
    public static void main(String[] args) throws Exception {
                                                        // 所有异常抛出
        File file = new File("d:" + File.separator + "mldn");
                                                        // 要压缩的文件夹
        File zipFile = new File("d:" + File.separator + "mldndir.zip");
                                                        // 压缩文件的名称
        InputStream input = null;                       // 定义文件输入流
        ZipOutputStream zipOut = null;                  // 定义压缩输出流
        zipOut = new ZipOutputStream(new FileOutputStream(zipFile));
                                                        // 实例化压缩输出流
        zipOut.setComment("www.mldnjava.cn");           // 设置注释
        if (file.isDirectory()) {                       // 判断是否是目录
            File lists[] = file.listFiles();            // 列出全部文件
            for (int i = 0; i < lists.length; i++) {
                input = new FileInputStream(lists[i]); // 设置文件输入流
                // 每一个被压缩的文件都用ZipEntry表示，需要为每一个压缩后的文件设置
                   名称
                zipOut.putNextEntry(new ZipEntry(file.getName()
                        + File.separator + lists[i].getName()));
                                                        // 创建ZipEntry
                int temp = 0;                           // 接收输入的数据
                while ((temp = input.read()) != -1) {   // 读取内容
                    zipOut.write(temp);                 // 压缩输出内容
                }
                input.close();                          // 关闭输入流
            }
        }
        zipOut.close();                                 // 关闭压缩输出流
    }
}
```

以上代码将 mldn 文件夹的内容压缩成 mldndir.zip 文件。程序首先判断给定的路径是否是文件夹，如果是文件夹，则将此文件夹中的内容使用 listFiles()方法全部列出，此方法返回 File 的对象数组，然后将此 File 对象数组中的每个文件进行压缩，每次压缩时都要设置一个新的 ZipEntry 对象。

程序执行完毕后，在 D 盘中会生成一个 mldndir.zip 的文件，文件打开如图 12-26 所示。

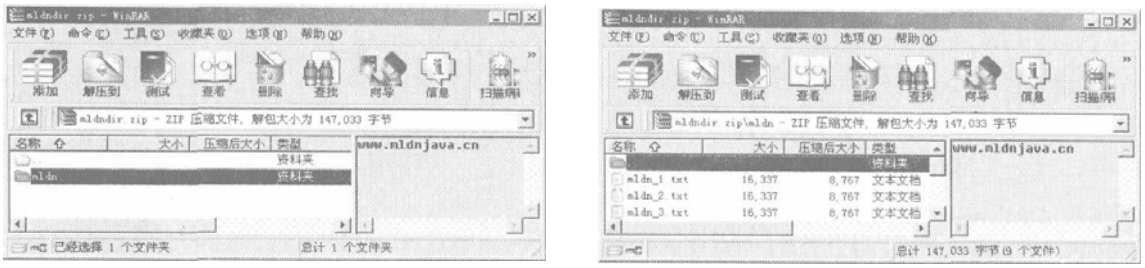


图 12-26　mldndir.zip 文件内容

12.13.3　ZipFile 类

在 Java 中，每一个压缩文件都可以使用 ZipFile 表示，还可以使用 ZipFile 根据压缩后的文件名称找到每一个压缩文件中的 ZipEntry 并将其进行解压缩操作，ZipFile 类的常用方法如表 12-21 所示。

表 12-21　ZipFile 类的常用方法

序　号	方　法	类　型	描　述
1	public ZipFile(File file) throws ZipException, IOException	构造	根据 File 类实例化 ZipFile 对象
2	public ZipEntry getEntry(String name)	普通	根据名称找到其对应的 ZipEntry
3	public InputStream getInputStream(ZipEntry entry) throws IOException	普通	根据 ZipEntry 取得 InputStream 实例
4	public String getName()	普通	得到压缩文件的路径名称

ZipFile 类实例化时需要 File 指定的路径，下面介绍 ZipFile 类的基本使用。

范例：实例化 ZipFile 类对象

```
package org.lxh.demo12.zipdemo;
import java.io.File;
import java.util.zip.ZipFile;
public class ZipFileDemo01 {
    public static void main(String[] args) throws Exception {
        File file = new File("d:" + File.separator + "mldn.zip");
                                                        // 找到压缩文件
        ZipFile zipFile = new ZipFile(file);            // 实例化ZipFile对象
        System.out.println("压缩文件的名称为：" + zipFile.getName());
                                                        // 得到压缩文件的名称
    }
}
```

程序运行结果：

```
压缩文件的名称为：d:\mldn.zip
```

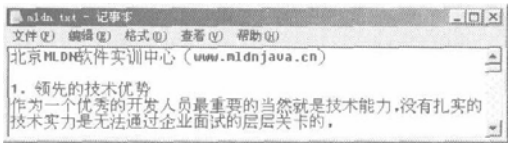
以上程序只是实例化 ZipFile 对象，并通过 getName()方法取得了压缩文件的名称。下面介绍如何利用此类进行文件的解压缩操作，以前面的 mldn.zip 文件为例进行解压缩（注

意，此压缩文件中只存在一个 ZipEntry，即 mldn.txt，如图 12-22 所示）。

范例：解压缩文件

```
package org.lxh.demo12.zipdemo;
import java.io.File;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.OutputStream;
import java.util.zip.ZipEntry;
import java.util.zip.ZipFile;
public class ZipFileDemo02 {
    public static void main(String[] args) throws Exception {
        File file = new File("d:" + File.separator + "mldn.zip");
                                                    // 找到压缩文件
        File outputFile = new File("d:" + File.separator
                + "mldn_unzip.txt");                // 定义解压缩的文件名称
        ZipFile zipFile = new ZipFile(file);        // 实例化ZipFile对象
        ZipEntry entry = zipFile.getEntry("mldn.txt");
                                                    // 得到一个压缩实体
        InputStream input = zipFile.getInputStream(entry);
                                                    // 取得ZipEntry输入流
        OutputStream out = new FileOutputStream(outputFile);
                                                    // 实例化输出流对象
        int temp = 0;                               // 保存接收数据
        while ((temp = input.read()) != -1) {       // 读取内容
            out.write(temp);                        // 输出内容
        }
        input.close();                              // 关闭输入流
        out.close();                                // 关闭输出流
    }
}
```

程序运行结果（还原压缩内容）：



以上程序是将 D 盘中 mldn.zip 中的文件解压缩到 mldn_unzip.txt 文件中。程序首先通过 getEntry()方法根据名称取得一个压缩的 ZipEntry，然后通过 InputStream 取得此 ZipEntry 的输入流，并通过循环方式将全部内容通过输出流输出。

但是，以上程序只适合于压缩文件中存在一个 ZipEntry 的情况，如果一个压缩文件中存在文件夹或者多个 ZipEntry 就无法使用了。如果要操作更加复杂的压缩文件，就必须结合 ZipInputStream 类完成。

12.13.4 ZipInputStream 类

ZipInputStream 是 InputStream 的子类，通过此类可以方便地读取 ZIP 格式的压缩文件，此类的常用方法如表 12-22 所示。

表 12-22 ZipInputStream 类的常用方法

序　号	方　法	类　型	描　述
1	public ZipInputStream(InputStream in)	构造	实例化 ZipInputStream 对象
2	public ZipEntry getNextEntry() throws IOException	普通	取得下一个 ZipEntry

使用 ZipInputStream 可以像 ZipFile 一样取得 ZIP 压缩文件中的每一个 ZipEntry。

范例：取得 mldn.zip 中的一个 ZipEntry

```
package org.lxh.demo12.zipdemo;
import java.io.File;
import java.io.FileInputStream;
import java.util.zip.ZipEntry;
import java.util.zip.ZipInputStream;
public class ZipInputStreamDemo01 {
    public static void main(String[] args) throws Exception {
                                                        // 所有异常抛出
        File zipFile = new File("d:" + File.separator + "mldn.zip");
                                                        // 定义压缩文件名称
        ZipInputStream input = null;                    // 定义压缩输入流
        input = new ZipInputStream(new FileInputStream(zipFile));
                                                        // 实例化压缩输入流
        ZipEntry entry = input.getNextEntry();          // 得到一个压缩实体
        System.out.println("压缩实体名称：" + entry.getName()) ;
                                                        // 输出实体名称
        input.close();                                  // 关闭压缩输入流
    }
}
```

程序运行结果：

```
压缩实体名称：mldn.txt
```

从以上代码中发现，通过 ZipInputStream 类中的 getNextEntry()方法可以依次取得每一个 ZipEntry，那么将此类与 ZipFile 结合就可以对压缩的文件夹进行解压缩操作。但是需要注意的是，在 mldndir.zip 文件中本身是包含压缩的文件夹的，所以在进行解压缩前，应该先根据 ZIP 文件中的文件夹的名称在硬盘上创建好一个对应的文件夹，然后才能把文件解压缩进去，而且在操作时对于每一个解压缩的文件都必须先创建（File 类的 createNewFile()方法可以创建新文件）后再将内容输出。

范例：解压缩 mldndir.zip 文件

```
package org.lxh.demo12.zipdemo;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.OutputStream;
import java.util.zip.ZipEntry;
import java.util.zip.ZipFile;
import java.util.zip.ZipInputStream;
public class ZipInputStreamDemo02 {
    public static void main(String[] args) throws Exception {
        File file = new File("d:" +
                File.separator + "mldndir.zip");          // 找到压缩文件
        File outFile = null;                              // 定义输出的文件对象
        ZipFile zipFile = new ZipFile(file);              // 实例化ZipFile对象
        ZipInputStream zipInput = new ZipInputStream(
                new FileInputStream(file));               // 实例化ZIP输入流
        ZipEntry entry = null;      // 定义一个ZipEntry对象，用于接收压缩文件中
                                       的每一个实体
        InputStream input = null;   // 定义输入流，用于读取每一个ZipEntry
        OutputStream out = null;    // 定义输出流，用于输出每一个实体内容
        while ((entry = zipInput.getNextEntry()) != null) { // 得到每一个
                                                               ZipEntry
            System.out.println("解压缩" + entry.getName() + "文件。");
            outFile = new File("d:" + File.separator + entry.getName());
                                                          // 实例化输出文件
            if (!outFile.getParentFile().exists()) {      // 判断文件夹是否存在
                outFile.getParentFile().mkdir();          // 创建文件夹
            }
            if (!outFile.exists()) {                      // 判断文件是否存在
                outFile.createNewFile();                  // 不存在则创建新文件
            }
            input = zipFile.getInputStream(entry);        // 得到压缩实体的输
                                                             入流
            out = new FileOutputStream(outFile);          // 实例化输出流对象
            int temp = 0;
            while ((temp = input.read()) != -1) {         // 读取内容
                out.write(temp);                          // 输出内容
            }
            input.close();                                // 关闭输入流
            out.close();                                  // 关闭输出流
```

```
            }
        }
    }
```

程序运行结果：

```
解压缩mldn\mldn_1.txt文件。
解压缩mldn\mldn_2.txt文件。
解压缩mldn\mldn_3.txt文件。
解压缩mldn\mldn_4.txt文件。
解压缩mldn\mldn_5.txt文件。
解压缩mldn\mldn_6.txt文件。
解压缩mldn\mldn_7.txt文件。
解压缩mldn\mldn_8.txt文件。
解压缩mldn\mldn_9.txt文件。
```

以上程序首先使用 ZipInputStream 读取 ZIP 格式的压缩文件，然后通过 getNextEntry()方法依次读取出其中每一个 ZipEntry 对象的名称，并通过 ZipFile 类取得每一个 ZipEntry 的输入流对象，在进行文件输出前，判断其输出文件夹及文件是否存在，如果不存在则创建。解压缩之后的效果如图 12-27 所示。

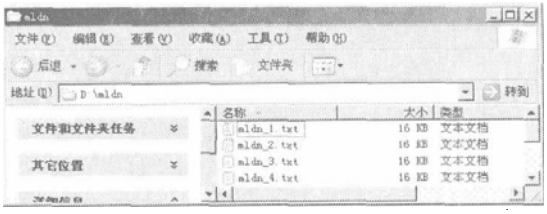


图 12-27　解压缩后的文件夹

12.14　回　退　流

在 Java IO 中所有的数据都是采用顺序的读取方式，即对于一个输入流来说，都是采用从头到尾的顺序读取的。如果在输入流中某个不需要的内容被读取进来，则只能通过程序将这些不需要的内容处理掉。为了解决这样的读取问题，在 Java 中提供了一种回退输入流（PushbackInputStream、PushbackReader），可以把读取进来的某些数据重新退回到输入流的缓冲区中。回退流操作机制如图 12-28 所示。

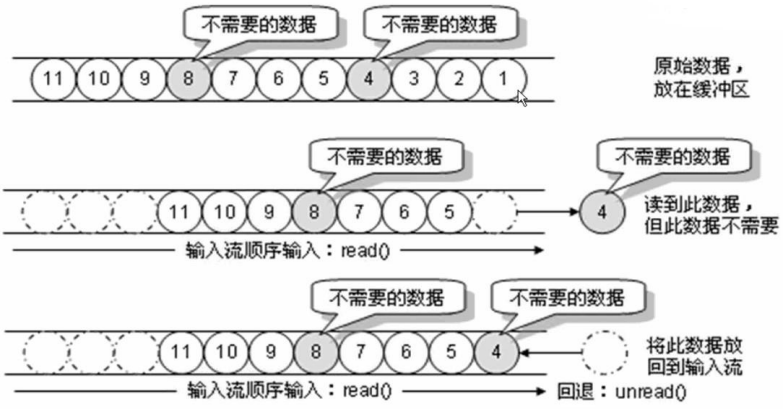


图 12-28　回退流操作机制

从图 12-28 可以发现，在回退流之中，对于不需要的数据可以使用 unread()方法将内容重新送回到输入流的缓冲区中。下面以 PushbackInputStream 为例进行讲解，PushbackInputStream 类的常用方法如表 12-23 所示。

表 12-23　PushbackInputStream 类的常用方法

序号	方　法	类型	描　述
1	public PushbackInputStream(InputStream in)	构造	将输入流放入到回退流中
2	public int read() throws IOException	普通	读取数据
3	public int read(byte[] b,int off,int len) throws IOException	普通	读取指定范围的数据
4	public void unread(int b) throws IOException	普通	回退一个数据到缓冲区前面
5	public void unread(byte[] b) throws IOException	普通	回退一组数据到缓冲区前面
6	public void unread(byte[] b,int off,int len) throws IOException	普通	回退指定范围的一组数据到缓冲区前面

表 12-24 中的 3 个 unread()方法与 InputStream（PushbackInputStream 是 InputStream 的子类）类中的 3 个 read()方法相对应，所以回退完全是针对于输入流进行操作的，如表 12-24 所示。

表 12-24　回退流与输入流的对应

序号	InputStream		PushbackInputStream	
1	读取一个	public abstract int read() throws IOException	回退一个	public void unread(int b)throws IOException
2	读取一组	public int read(byte[] b) throws IOException	回退一组	public void unread(byte[] b) throws IOException
3	读取部分	public int read(byte[] b,int off,int len) throws IOException	回退部分	public void unread(byte[] b,int off, int len) throws IOException

下面以一个简单的程序为例进行回退流的讲解，现在内存中有一个“www.mldnjava.cn”字符串，只要输入的内容是“.”则执行回退操作，即不读取“.”。

范例：操作回退流

```
package org.lxh.demo12.pushbackdemo;
import java.io.ByteArrayInputStream;
import java.io.PushbackInputStream;
public class PushInputStreamDemo {
    public static void main(String args[]) throws Exception {
                                                          // 所有异常抛出
        String str = "www.mldnjava.cn" ;                  // 定义字符串
        PushbackInputStream push = null ;                 // 定义回退流对象
        ByteArrayInputStream bai = null ;                 // 定义内存输入流
        bai = new ByteArrayInputStream(str.getBytes()) ;// 实例化内存输入流
        push = new PushbackInputStream(bai) ;             // 实例化回退流对象
```

```
            System.out.print("读取之后的数据为：") ;
            int temp = 0 ;                                          // 接收数据
            while ((temp = push.read()) != -1) {                    // 读取内容
                if (temp == '.') {
                    push.unread(temp) ;                             // 回退到缓冲区前面
                    temp = push.read() ;                            // 空出此数据
                    System.out.print("（退回"+(char)temp+"）");
                }else{
                    System.out.print((char)temp);                   // 输出内容
                }
            }
        }
    }
```

程序运行结果：

```
读取之后的数据为：www（退回.）mldnjava（退回.）cn
```

本程序中为了让读者看清楚哪些内容是被回退的，所以将被回退的部分打印输出。

12.15 字 符 编 码

12.15.1 Java 常见编码简介

在计算机世界里，任何的文字都是以指定的编码方式存在的，在 Java 程序的开发中最常见的是 ISO8859-1、GBK/GB2312、unicode、UTF 编码。

- ISO8859-1：属于单字节编码，最多只能表示 0～255 的字符范围，主要在英文上应用。
- GBK/GB2312：中文的国标编码，专门用来表示汉字，是双字节编码，如果在此编码中出现中文，则使用 ISO8859-1 编码，GBK 可以表示简体中文和繁体中文，而 GB2312 只能表示简体中文，GBK 兼容 GB2312。
- unicode：Java 中使用此编码方式，是最标准的一种编码，使用十六进制表示编码。但此编码不兼容 ISO8859-1 编码。
- UTF：由于 unicode 不支持 ISO8859-1 编码，而且容易占用更多的空间，而且对于英文字母也需要使用两个字节编码，这样使用 unicode 不便于传输和存储，因此产生了 UTF 编码。UTF 编码兼容了 ISO8859-1 编码，同时也可以用来表示所有的语言字符，不过 UTF 编码是不定长编码，每一个字符的长度为 1～6 个字节不等。一般在中文网页中使用此编码，可以节省空间。

在程序中如果处理不好字符的编码，则就有可能出现乱码问题。如果现在本机的默认编码是 GBK，但在程序中使用了 ISO8859-1 编码，则会出现字符的乱码问题。就像两个人交谈，一个人说的是中文，另外一个人说的是其他语言，如果语言不同，则肯定无法沟通，如图 12-29 所示。



图 12-29　乱码产生

所以，如果要避免产生乱码，则程序的编码与本地的默认编码保持一致即可，而如果要知道本地的默认编码，则可以使用 System 类完成。

12.15.2　得到本机的编码显示

在前面讲解常用类库时曾经为读者介绍过，使用 System 类可以取得与系统有关的信息，所以直接使用此类即可找到系统的默认编码，使用如下方法：

```
public static Properties getProperty()
```

范例：使用上述方法得到 JVM 的默认编码

```
package org.lxh.demo12.charsetdemo;
public class CharSetDemo01 {
    public static void main(String[] args) {
        System.out.println("系统默认编码:" +
                System.getProperty("file.encoding"));  // 获取当前系统编码
    }
}
```

程序运行结果：

```
系统默认编码:GBK
```

可以发现，现在操作系统的默认编码是 GBK，所以，如果此时使用了 ISO8859-1 编码，则肯定出现乱码。

12.15.3　乱码产生

下面通过一个范例讲解乱码的产生。现在本地的默认编码是 GBK，下面通过 ISO8859-1 编码对文字进行编码转换。如果要实现编码的转换可以使用 String 类中的 getBytes(String charset)方法，此方法可以设置指定的编码，该方法的格式如下：

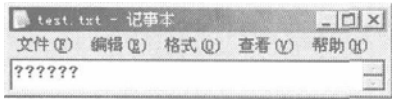
```
public byte[] getBytes(String charset) ;
```

范例：使用 ISO8859-1 编码

```
package org.lxh.demo12.charsetdemo;
import java.io.File;
import java.io.FileOutputStream;
import java.io.OutputStream;
public class CharSetDemo02 {
```

```
    public static void main(String[] args) throws Exception {
        File f = new File("D:" + File.separator + "test.txt");
        OutputStream out = new FileOutputStream(f);     // 实例化输出流
        byte b[] = "中国，你好！".getBytes("ISO8859-1"); // 指定ISO8859-1编码
        out.write(b) ;                                  // 保存转码之后的数据
        out.close();                                    // 关闭输出流
    }
}
```

程序运行结果：



可以发现，因为编码不一致，所以在保存时出现了乱码。在 Java 的开发中，乱码是一个比较常见的问题，乱码的产生就有一个原因，即输出内容的编码（例如：程序指定）与接收内容的编码（如本机环境默认）不一致。

本章目标

- 掌握对象序列化的作用
- 掌握Serializable接口的作用
- 可以使用ObjectOutputStream进行对象序列化操作
- 可以使用ObjectInputStream进行对象的反序列化操作
- 掌握Externalizable接口的作用及与Serializable接口的实现区别
- 掌握transient关键字的作用
- 可以序列化一组对象

12.16 对象序列化

12.16.1 基本概念与 Serializable 接口

对象序列化就是把一个对象变为二进制的数据流的一种方法，如图 12-30 所示。通过对象序列化可以方便地实现对象的传输或存储。

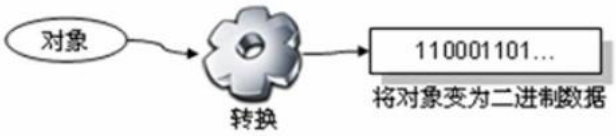


图 12-30 对象转换为二进制数据

如果一个类的对象想被序列化，则对象所在的类必须实现 java.io.Serializable 接口。此接口的定义如下：

```
public interface Serializable{}
```

可以发现在此接口中并没有定义任何的方法，所以此接口是一个标识接口。表示一个类具备了被序列化的能力。

范例：定义可序列化的类

```
package org.lxh.demo12.serdemo;
import java.io.Serializable;
public class Person implements Serializable {  // 此类的对象可以被序列化
    private String name;                        // 声明name属性
    private int age;                            // 声明age属性
    public Person(String name, int age) {       // 通过构造方法设置属性内容
```

```
        this.name = name;
        this.age = age;
    }
    public String toString() {                    // 覆写toString()方法
        return "姓名：" + this.name + "；年龄：" + this.age;
    }
}
```

以上的 Person 类已经实现了序列化接口，所以此类的对象是可以经过二进制数据流进行传输的。而如果要完成对象的输入或输出，还必须依靠对象输出流（ObjectOutputStream）和对象输入流（ObjectInputStream）。

使用对象输出流输出序列化对象的步骤有时也称为序列化，而使用对象输入流读入对象的过程有时也称为反序列化，如图 12-31 所示。



图 12-31　对象的序列化和反序列化过程

提示：对象序列化和对象反序列化操作时的版本兼容性问题。

在对象进行序列化或反序列化操作时，要考虑 JDK 版本的问题。如果序列化的 JDK 版本和反序列化的 JDK 版本不统一则就有可能造成异常，所以在序列化操作中引入了一个 serialVersionUID 的常量，可以通过此常量来验证版本的一致性。在进行反序列化时，JVM 会把传来的字节流中的 serialVersionUID 与本地相应实体（类）的 serialVersionUID 进行比较，如果相同就认为是一致的，可以进行反序列化，否则就会出现序列化版本不一致的异常。

当实现 java.io.Serializable 接口的实体（类）没有显式地定义一个名为 serialVersionUID、类型为 long 的变量时，Java 序列化机制在编译时会自动生成一个此版本的 serialVersionUID。当然，如果不希望通过编译来自动生成，也可以直接显式地定义一个名为 serialVersionUID、类型为 long 的变量，只要不修改这个变量值的序列化实体，都可以相互进行串行化和反串行化。

本程序中直接在 Person 中加入以下的常量即可：

```
private static final long serialVersionUID = 1L;
```

serialVersionUID 的具体内容由用户指定。

12.16.2　对象输出流 ObjectOutputStream

一个对象如果要进行输出，则必须使用 ObjectOutputStream 类，此类的定义如下：

```
public class ObjectOutputStream
extends OutputStream
implements ObjectOutput, ObjectStreamConstants
```

ObjectOutputStream 类属于 OutputStream 的子类，此类的常用方法如表 12-25 所示。

表 12-25　ObjectOutputStream 常用方法

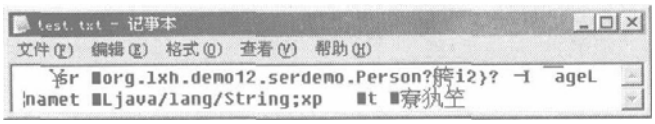
序号	方　　法	类型	描　　述
1	public ObjectOutputStream(OutputStream out) throws IOException	构造	传入输出的对象
2	public final void writeObject(Object obj) throws IOException	普通	输出对象

此类的使用形式与 PrintStream 非常相似，在实例化时也需要传入一个 OutputStream 的子类对象，然后根据传入的 OutputStream 子类的对象不同，输出的位置也不同。

范例：将 Person 类的对象保存在文件中

```
package org.lxh.demo12.serdemo;
import java.io.File;
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import java.io.OutputStream;
public class SerDemo01 {
    public static void main(String[] args) throws Exception {
        File f = new File("D:" + File.separator + "test.txt");
        ObjectOutputStream oos = null;
        OutputStream out = new FileOutputStream(f);          // 文件输出流
        oos = new ObjectOutputStream(out);                   // 为对象输出流实例化
        oos.writeObject(new Person("张三", 30));             // 保存对象到文件
        oos.close();                                         // 关闭输出
    }
}
```

程序运行结果：



通过以上代码可将内容保存到文件中，保存的内容全是二进制数据，但是保存的文件本身不可以直接修改，因为会破坏其保存格式。

❸提问：到底序列化了哪些内容？

一个对象被序列化后，到底哪些内容被保存了下来，是属性还是方法？

回答：只有属性被序列化。

在面向对象基础部分时曾经讲解过，每个对象都具备相同的方法，但是每个对象的属性不一定相同，也就是说，对象保存的只有属性信息，那么在序列化操作时也同样是这个道理，只有属性被序列化。

12.16.3 对象输入流 ObjectInputStream

使用 ObjectInputStream 可以直接把被序列化好的对象反序列化。ObjectInputStream 的定义如下：

```
public class ObjectInputStream
extends InputStream
implements ObjectInput, ObjectStreamConstants
```

ObjectInputStream 类也是 InputStream 的子类，与 PrintStream 类的使用类似。此类同样需要接收 InputStream 类的实例才可以实例化。主要操作方法如表 12-26 所示。

表 12-26 ObjectInputStream 的主要操作方法

序号	方 法	类型	描 述
1	public ObjectInputStream(InputStream in) throws IOException	构造	构造输入对象
2	public final Object readObject() throws IOException, ClassNotFoundException	普通	从指定位置读取对象

下面使用对象输入流将 12.16.2 节保存在文件中的对象读取出来，此过程也称为反序列化。

范例：从文件中将 Person 对象反序列化（读取）

```
package org.lxh.demo12.serdemo;
import java.io.File;
import java.io.FileInputStream;
import java.io.InputStream;
import java.io.ObjectInputStream;
public class SerDemo02 {
    public static void main(String[] args) throws Exception {
        File f = new File("D:" + File.separator + "test.txt");
        ObjectInputStream ois = null;
        InputStream input = new FileInputStream(f);      // 文件输入流
        ois = new ObjectInputStream(input);              // 为对象输出流实例化
        Object obj = ois.readObject();                   // 读取对象
        ois.close();                                     // 关闭输出
        System.out.println(obj);
    }
}
```

程序的运行结果：

```
姓名：张三；年龄：30
```

从程序的运行结果中可以清楚地发现，实现了 Serializable 接口类，对象中的所有属性都被序列化，如果用户想根据自己的需要选择被序列化的属性，则可以使用另外一种序列化接口——Externalizable 接口。

提问：可不可以让所有的类都实现 Serializble 接口？

一个类如果实现了 Serializable 接口后可以直接序列化，而且此接口中没有任何的方法，也不会让实现此接口的类增加不必要的操作，那么所有的类都实现此接口不是更好吗？这样也可以增加类的一个功能。

回答：不可以，这样在以后的版本升级中会存在问题。

在目前已知的 JDK 版本中，java.io.Serializable 接口中都没有定义任何的方法，所以如果所有的类都实现此接口在语法上并没有任何的问题，但是，如果在以后的 JDK 版本中修改了此接口而且又增加了许多方法呢？那么以往的系统中所有类就都被修改，这样肯定会很麻烦，所以最好只在需要被序列化对象的类实现 Serializable 接口。

12.16.4 Externalizable 接口

被 Serializable 接口声明的类的对象的内容都将被序列化，如果现在用户希望自己指定序列化的内容，则可以让一个类实现 Externalizable 接口，此接口定义如下：

```
public interface Externalizable extends Serializable {
    public void writeExternal(ObjectOutput out) throws IOException ;
    public void readExternal(ObjectInput in) throws IOException,
    ClassNot FoundException ;
}
```

Externalizable 接口是 Serializable 接口的子接口，在此接口中定义了两个方法，这两个方法的作用如下。

- writeExternal(ObjectOutput out)：在此方法中指定要保存的属性信息，对象序列化时调用。
- readExternal(ObjectInput in)：在此方法中读取被保存的信息，对象反序列化时调用。

这两个方法的参数类型是 ObjectOutput 和 ObjectInput，两个接口的定义如下。

- ObjectOutput 接口定义：

```
public interface ObjectOutput extends DataOutput
```

- ObjectInput 接口定义：

```
public interface ObjectInput extends DataInput
```

可以发现以上两个接口分别继承 DataOutput 和 DataInput，这样在这两个方法中就可以像 DataOutputStream 和 DataInputStream 那样直接输出和读取各种类型的数据。

如果一个类要使用 Externalizable 实现序列化时，在此类中必须存在一个无参构造方法，因为在反序列化时会默认调用无参构造实例化对象，如果没有此无参构造，则运行时将会出现异常，这一点的实现机制与 Serializable 接口是不同的。

范例：修改 Person 类并实现 Externalizable 接口

```
package org.lxh.demo12.serdemo;
import java.io.Externalizable;
```

```
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;
public class Person implements Externalizable {// 此类的对象可以被序列化
    private String name;                          // 声明name属性
    private int age;                              // 声明age属性
    public Person(){}                             // 必须定义无参构造
    public Person(String name, int age) {         // 通过构造方法设置属性内容
        this.name = name;
        this.age = age;
    }
    public String toString() {                    // 覆写toString()方法
        return "姓名: " + this.name + "; 年龄: " + this.age;
    }
    // 覆写此方法，根据需要读取内容，反序列化时使用
    public void readExternal(ObjectInput in) throws IOException,
            ClassNotFoundException {
        this.name = (String)in.readObject() ;     // 读取姓名属性
        this.age = in.readInt() ;                 // 读取年龄
    }
    // 覆写此方法，根据需要可以保存属性或具体内容，序列化时使用
public void writeExternal(ObjectOutput out) throws IOException {
        out.writeObject(this.name) ;              // 保存姓名属性
        out.writeInt(this.age) ;                  // 保存年龄属性
    }
}
```

以上程序中的 Person 类实现了 Externalizable 接口，这样用户就可以在类中有选择地保存需要的属性或者其他的具体数据。在本程序中，为了与之前的程序统一，将全部属性保存下来。

范例：序列化和反序列化 Person 对象

```
package org.lxh.demo12.serdemo;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.OutputStream;
public class SerDemo03 {
    public static void main(String[] args) throws Exception {
        ser();                                    // 序列化
```

```
        dser();                                                    // 反序列化
    }
    public static void ser() throws Exception {                    // 序列化操作
        File f = new File("D:" + File.separator + "test.txt");
        ObjectOutputStream oos = null;
        OutputStream out = new FileOutputStream(f);                // 文件输出流
        oos = new ObjectOutputStream(out);                         // 为对象输出流实例化
        oos.writeObject(new Person("张三", 30));                   // 保存对象到文件
        oos.close();                                               // 关闭输出
    }
    public static void dser() throws Exception {                   // 反序列化操作
        File f = new File("D:" + File.separator + "test.txt");
        ObjectInputStream ois = null;
        InputStream input = new FileInputStream(f);                // 文件输出流
        ois = new ObjectInputStream(input);                        // 为对象输出流实例化
        Object obj = ois.readObject();                             // 读取对象
        ois.close();                                               // 关闭输出
        System.out.println(obj);
    }
}
```

从以上代码中可以发现，使用 Externalizable 接口实现序列化明显要比使用 Serializable 接口实现序列化麻烦得多，除此之外，两者的实现还有不同，如表 12-27 所示。

表 12-27　Externalizable 接口与 Serializable 接口实现序列化的区别

序　号	区　别	Serializable	Externalizable
1	实现复杂度	实现简单，Java 对其有内建支持	实现复杂，由开发人员自己完成
2	执行效率	所有对象由 Java 统一保存，性能较低	开发人员决定哪个对象保存，可能造成速度提升
3	保存信息	保存时占用空间大	部分存储，可能造成空间减少

在一般的开发中，因为 Serializable 接口的使用较为方便，所以出现较多。

12.16.5　transient 关键字

Serializable 接口实现的操作实际上是将一个对象中的全部属性进行序列化，当然也可以使用 Externalizable 接口以实现部分属性的序列化，但这样操作比较麻烦。

当使用 Serializable 接口实现序列化操作时，如果一个对象中的某个属性不希望被序列化，则可以使用 transient 关键字进行声明，代码如下所示。

范例： Person 中的 name 属性不希望被序列化

```
package org.lxh.demo12.serdemo;
import java.io.Serializable;
public class Person implements Serializable {  // 此类的对象可以被序列化
```

```
    private transient String name;                    // 此属性将不被序列化
    private int age;                                  // 此属性将被序列化
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String toString() {                        // 覆盖toString()，输出信息
        return "姓名：" + this.name + "；年龄：" + this.age;
    }
}
```

范例：重新保存再读取对象

```
package org.lxh.demo12.serdemo;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.OutputStream;
public class SerDemo04 {
    public static void main(String[] args) throws Exception {
        ser();                                                    // 序列化
        dser();                                                   // 反序列化
    }
    public static void ser() throws Exception {                   // 序列化操作
        File f = new File("D:" + File.separator + "test.txt");
        ObjectOutputStream oos = null;
        OutputStream out = new FileOutputStream(f);               // 文件输出流
        oos = new ObjectOutputStream(out);                        // 为对象输出流实例化
        oos.writeObject(new Person("张三", 30));                  // 保存对象到文件
        oos.close();                                              // 关闭输出
    }
    public static void dser() throws Exception {                  // 反序列化操作
        File f = new File("D:" + File.separator + "test.txt");
        ObjectInputStream ois = null;
        InputStream input = new FileInputStream(f);               // 文件输出流
        ois = new ObjectInputStream(input);                       // 为对象输出流实例化
        Object obj = ois.readObject();                            // 读取对象
        ois.close();                                              // 关闭输出
        System.out.println(obj);
    }
}
```

程序运行结果：

```
姓名：null；年龄：30
```

以上程序中的姓名为 null，表示内容没有被序列化保存下来。这样在对象反序列化后，输出时姓名就是默认值 null。

12.16.6 序列化一组对象

对象输出时只提供了一个对象的输出操作（writeObject(Object obj)），并没有提供多个对象的输出，所以如果现在要同时序列化多个对象，就可以使用对象数组进行操作，因为数组属于引用数据类型，所以可以直接使用 Object 类型进行接收，如图 12-32 所示。

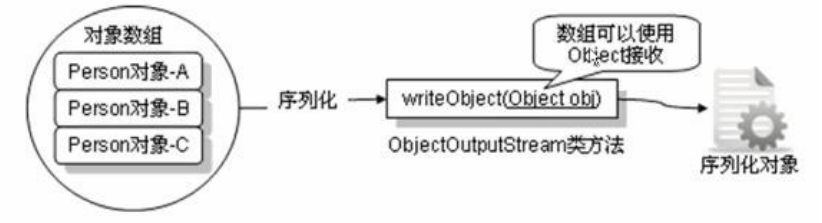


图 12-32 序列化一组对象

范例：序列化多个 Person 对象

```
package org.lxh.demo12.serdemo;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.OutputStream;
public class SerDemo05 {
    public static void main(String[] args) throws Exception {
        Person per[] = { new Person("张三", 30), new Person("李四", 31),
                new Person("王五", 32) };        // 定义对象数组
        ser(per);                                // 序列化对象数组
        Object o[] = dser();                     // 读取被序列化的对象数组
        for (int i = 0; i < o.length; i++) {
            Person p = (Person) o[i];
            System.out.println(p);
        }
    }
    public static void ser(Object obj[]) throws Exception {
        File f = new File("D:" + File.separator + "test.txt");
        ObjectOutputStream oos = null;
        OutputStream out = new FileOutputStream(f);     // 文件输出流
```

```
            oos = new ObjectOutputStream(out);                    // 为对象输出流实例化
            oos.writeObject(obj);                                 // 保存对象数组到文件
            oos.close();                                          // 关闭输出
        }
        public static Object[] dser() throws Exception {
            File f = new File("D:" + File.separator + "test.txt");
            ObjectInputStream ois = null;
            InputStream input = new FileInputStream(f);           // 文件输入流
            ois = new ObjectInputStream(input);                   // 为对象输出流实例化
            Object obj[] = (Object[]) ois.readObject();           // 读取对象数组
            ois.close();                                          // 关闭输出
            return obj;
        }
    }
```

程序运行结果：

```
姓名：张三；年龄：30
姓名：李四；年龄：31
姓名：王五；年龄：32
```

以上程序中使用对象数组可以保存多个对象，但是数组本身存在长度的限制，为了解决数组中的长度问题，所以使用动态对象数组（类集）完成，具体内容在第 13 章中将为读者介绍。

12.17　实例操作——单人信息管理程序

将前面的菜单程序进行扩充，要求增加时可以增加一个人的完整信息，人的信息包括姓名和年龄。保存后也可以修改、删除、查询此信息。

可以使用对象序列化保存。此时程序可以使用前面讲解过的 InputData、Person、Operate、Menu 几个类。需要增加文件操作类，专门负责保存和读取文件的内容，并修改 Operate 类，为其增加具体的操作，此程序的操作如图 12-33 所示。

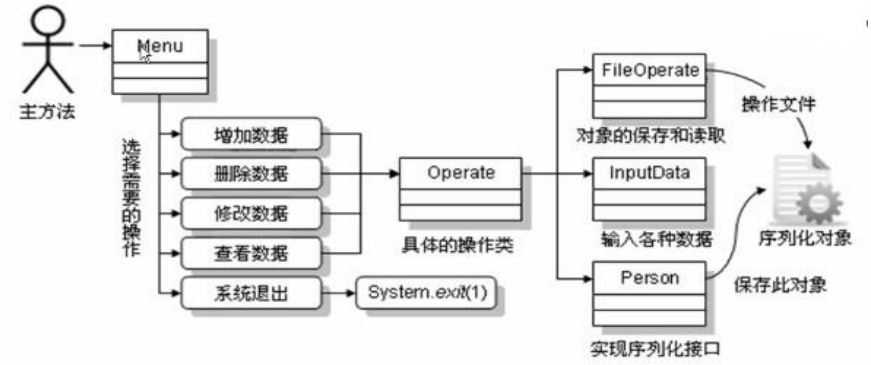


图 12-33　单人信息管理程序的操作

范例：增加文件操作类

```
package org.lxh.demo12.execdemo01;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
public class FileOperate {                          // 此类专门用于保存和读取对象
    private File file = null;                       // 定义一个文件对象
    public FileOperate(String pathName) {           // 通过构造方法传递文件路径
        this.file = new File(pathName);             // 实例化File类对象
    }
    public boolean save(Object obj) throws Exception { // 可以保存对象
        ObjectOutputStream oos = null;              // 对象输出流
        boolean flag = false;                       // 定义操作标志位
        try {
            oos = new ObjectOutputStream(
                    new FileOutputStream(this.file));  // 实例化对象输出流
            oos.writeObject(obj);                   // 保存对象
            flag = true;                            // 修改标志位
        } catch (Exception e) {
            throw e;                                // 有异常抛出
        } finally {
            if (oos != null) {                      // 判断对象输出流对象
                                                       是否被实例化
                oos.close();                        // 不管是否有异常都
                                                       关闭
            }
        }
        return flag;
    }
    public Object load() throws Exception {         // 读取对象
        Object obj = null;                          // 接收保存的对象
        ObjectInputStream ois = null;               // 声明对象输入流
        try {
            ois = new ObjectInputStream(
                    new FileInputStream(this.file));   // 实例化对象输入流
            obj = ois.readObject();                 // 读取对象
        } catch (Exception e) {
            throw e;                                // 有异常抛出
        } finally {
```

```
            if (ois != null) {                          // 判断输入流对象是否被实例化
                ois.close();                            // 关闭输入流
            }
        }
        return obj;
    }
}
```

以上程序中的类的功能就是向程序中写入对象和读取对象，在操作时只需要传入一个路径即可。

范例：修改 Person 类，增加 setter 和 getter

```
package org.lxh.demo12.execdemo01;
import java.io.Serializable;
public class Person implements Serializable {          // 此类的对象可以被序列化
    private String name;                                // 声明姓名属性
    private int age;                                    // 声明年龄属性
    public Person(String name, int age) {               // 通过构造设置属性内容
        this.name = name;
        this.age = age;
    }
    public String toString() {                          // 覆写toString()方法
        return "姓名：" + this.name + "；年龄：" + this.age;
    }
    public String getName() {                           // 取得姓名
        return name;
    }
    public void setName(String name) {                  // 设置姓名
        this.name = name;
    }
    public int getAge() {                               // 取得年龄
        return age;
    }
    public void setAge(int age) {                       // 设置年龄
        this.age = age;
    }
}
```

需要在增加、修改、删除、显示的地方编写具体的代码，但是在修改时应该先查询出来，显示已有的内容。

范例：修改操作类

```
package org.lxh.demo12.execdemo01;
public class Operate {
    public static void add() {                          // 增加数据操作
```

```
        InputData input = new InputData();          // 实例化输入数据对象
        FileOperate fo = new FileOperate("d:\\test.per");
        String name = input.getString("请输入姓名：");
        int age = input.getInt("请输入年龄：","年龄必须是数字！");
        Person per = new Person(name, age);         // 实例化Person对象
        try {
            fo.save(per);                           // 保存对象
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.out.println("信息增加成功！");
    }
    public static void delete() {                   // 删除数据操作
        FileOperate fo = new FileOperate("d:\\test.per");
        try {
            fo.save(null);                          // 清除对象
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.out.println("信息删除成功！");
    }
    public static void update() {                   // 修改数据操作
        InputData input = new InputData();          // 实例化输入数据对象
        FileOperate fo = new FileOperate("d:\\test.per");
        Person per = null;
        try {
            per = (Person) fo.load();               // 读取数据
        } catch (Exception e1) {
            e1.printStackTrace();
        }
        String name = input.getString("请输入新的姓名（原姓名：" + per.getName()
        + "）：");
        int age = input.getInt("请输入新的年龄（原年龄：" + per.getAge() + "）：
        ","年龄必须是数字！");
        per = new Person(name, age);                // 重新实例化对象
        try {
            fo.save(per);                           // 重新保存对象
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.out.println("信息更新成功！");
    }
```

```
    public static void find() {                          // 查看数据操作
        FileOperate fo = new FileOperate("d:\\test.per");
        Person per = null;
        try {
            per = (Person) fo.load();                    // 读取对象
        } catch (Exception e1) {
            e1.printStackTrace();
        }
        System.out.println(per);
    }
}
```

此时运行程序后，即可对单人信息进行增加、修改、删除、浏览的操作。

12.18　本 章 要 点

1. 本章主要讲解了如图 12-34 中所列的操作类，在图中列出了各类间的继承关系。

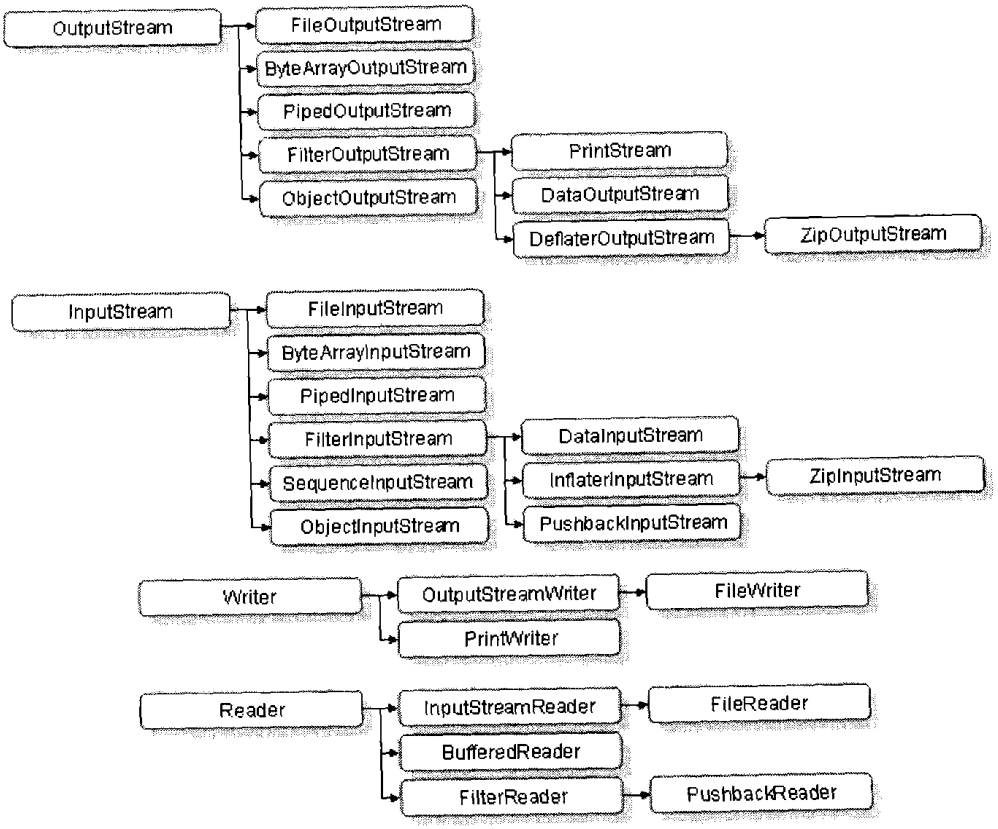


图 12-34　本章所使用到的 IO 操作类

2. 在 Java 中使用 File 类表示文件本身，可以直接使用此类完成文件的各种操作，如创建、删除等。

3．RandomAccessFile 类可以从指定的位置开始读取信息，但是要求文件中各个数据的保存长度必须固定。

4．输入/输出流主要分为字节流（OutputStream、InputStream）和字符流（Writer、Reader）两种，但是在传输中以字节流操作较多，字符流在操作时使用到缓冲区，而字节流没有使用到缓冲区。

5．字节或字符流都是以抽象类的形式定义的，根据其使用的子类不同，输入或输出的位置也不同。

6．在 IO 包中可以使用 OutputStreamWriter 和 InputStreamReader 完成字符流与字节流之间的转换操作。

7．使用 ByteArrayInputStream 和 ByteArrayOutputStream 类可以对内存进行输入/输出操作。

8．在线程之间进行输入/输出通信，主要使用 PipedOutputStream 和 PipedInputStream 类。

9．在 IO 中输出时最好使用打印流（PrintStream、PrintWriter），这样可以方便地输出各种类型的数据。

10．System 类提供了 3 个支持 IO 操作的常量即 out、err、in。

- System.out：对应显示器的标准输出。
- System.err：对应错误打印，一般此信息不希望被用户看到。
- System.in：对应标准的键盘输入。

在程序操作中，根据 setOut()方法可以修改 System.out 的输出位置，可以使用 setErr()方法修改 System.err 的输出位置，也可以使用 setIn()方法修改 System.in 的输入位置。

11．BufferedReader 可以直接从缓冲区中读取数据。

12．使用 Scanner 类可以方便地进行输入流操作。

13．数据操作流提供了与平台无关的数据操作，主要使用 DataOutputStream 和 DataInputStream 类。

14．使用合并流（SequenceInputStream）可以将两个文件的内容进行合并。

15．如果数据量过大，则可以使用压缩流压缩数据，在 Java 中支持 ZIP、JAR 和 GZIP 3 种压缩格式。

16．使用回退流可以将不需要的数据回退到数据缓冲区中以待重新读取。

17．造成字符乱码的根本原因就在于程序编码与本地编码的不统一。

18．对象序列化可以将内存中的对象转化为二进制数据，但对象所在的类必须实现 Serializable 接口，一个类中的属性如果使用 transient 关键字声明，则此属性的内容将不会被序列化。

19．对象的输入/输出主要使用 ObjectInputStream 和 ObjectOutputStream 两个类完成。

12.19 习　　题

1．编写 Java 程序，输入 3 个整数，并求出 3 个整数的最大值和最小值。

2．从键盘输入文件的内容和要保存的文件名称，然后根据输入的名称创建文件，并将内容保存到文件中。

3．从键盘传入多个字符串到程序中，并将它们按逆序输出在屏幕上。

4．从键盘输入以下的数据：“TOM：89 | JERRY：90 | TONY:95”，数据格式为“姓名：成绩 | 姓名：成绩 | 姓名：成绩”，对输入的内容按年龄进行排序，并将排序结果按照成绩由高到低排序。

5．将第 4 题中的内容进行扩展，可以将全部输入的信息保存在文件中，还可以添加信息，并可以显示全部的数据。

6．编写程序，程序运行后，根据屏幕提示输入一个数字字符串，输入后统计有多少个偶数数字和奇数数字。

7．完成系统登录程序，从命令行输入用户名和密码，如果没有输入用户名和密码，则提示输入用户名和密码；如果输入了用户名但是没有输入密码，则提示用户输入密码，然后判断用户名是否是 mldn，密码是否是 hello，如果正确，则提示登录成功；如果错误，显示登录失败的信息，用户再次输入用户名和密码，连续 3 次输入错误后系统退出。

8．完成文件复制操作，在程序运行后提示输入源文件路径，然后再输入目标文件路径。

9．编写程序，程序运行时输入目录名称，并把该目录下的所有文件名后缀修改为.txt。

10．编写一个投票程序，具体如下。

（1）功能描述

有一个班采用民主投票方法推选班长，班长候选人共 4 位，每个人姓名及代号分别为“张三　1；李四　2；王五　3；赵六　4”。程序操作员将每张选票上所填的代号（1、2、3 或 4）循环输入电脑，输入数字 0 结束输入，然后将所有候选人的得票情况显示出来，并显示最终当选者的信息。

（2）具体要求

① 要求用面向对象方法，编写学生类 Student，将候选人姓名、代号和票数保存到类 Student 中，并实现相应的 getXXX 和 setXXX 方法。

② 输入数据前，显示出各位候选人的代号及姓名（提示，建立一个候选人类型数组）。

③ 循环执行接收键盘输入的班长候选人代号，直到输入的数字为 0，结束选票的输入工作。

④ 在接收每次输入的选票后要求验证该选票是否有效，即如果输入的数不是 0、1、2、3、4 这 5 个数字之一，或者输入的是一串字母，应显示出错误提示信息“此选票无效，请输入正确的候选人代号！”，并继续等待输入。

⑤ 输入结束后显示所有候选人的得票情况，如参考样例所示。

⑥ 输出最终当选者的相关信息，如参考样例所示。

（3）参考样例

```
1：张三【0票】
2：李四【0票】
3：王五【0票】
4：赵六【0票】
```

```
请输入班长候选人代号（数字0结束）：1
请输入班长候选人代号（数字0结束）：1
请输入班长候选人代号（数字0结束）：1
请输入班长候选人代号（数字0结束）：2
请输入班长候选人代号（数字0结束）：3
请输入班长候选人代号（数字0结束）：4
请输入班长候选人代号（数字0结束）：5
此选票无效，请输入正确的候选人代号！
请输入班长候选人代号（数字0结束）：hello
此选票无效，请输入正确的候选人代号！
请输入班长候选人代号（数字0结束）：0
1：张三【4票】
2：李四【1票】
3：王五【1票】
4：赵六【1票】
投票最终结果：张三同学，最后以4票当选班长！
```

Java SE 中最重要的四部分：

Java 面向对象、Java类集、Java IO、JDBC

第 13 章　Java 类集

通过本章的学习可以达到以下目标：

- ☑ 了解 Java 设置类集的主要目的。
- ☑ 掌握 Collection 接口的作用及主要操作方法。
- ☑ 掌握 Collection 子接口 List、Set 的区别及常用子类的使用。
- ☑ 掌握 Map 接口的作用及与 Collection 接口的区别，并掌握 Map 接口的常用子类。
- ☑ 掌握 SortedSet、SortedMap 接口的排序原理。
- ☑ 掌握集合的 3 种常用输出方式：Iterator、Enumeration 和 foreach。
- ☑ 掌握 Properties 类的使用。
- ☑ 掌握集合的实际操作范例。
- ☑ 了解类集工具类 Collections 的作用。

类集是 Java 中的一个重要特性，在实际开发中占有很重要的地位，如果要写出一个好的程序，则一定要将类集的作用和各个组成部分的特点掌握清楚。本章将对 Java 类集进行完整的介绍，针对于一些较为常用的操作也将进行深入的讲解。

在 JDK 1.5 之后，为了使类集操作更加安全，对类集框架进行了修改，加入了泛型的操作。**本章视频录像讲解时间为 3 小时 50 分钟，源代码在光盘对应的章节下。**

提示：使用时不使用泛型将出现警告。

在使用各个类集接口时，如果没有指定泛型，则肯定会出现警告信息。此时，泛型将被擦除而全部使用 Object 接收。

13.1　认 识 类 集

13.1.1　基本概念

在讲解类集概念之前，先来思考这样一个问题，如果现在要保存一组对象，按照之前的做法则只能使用对象数组，但是使用对象数组操作本身有一个限制，就是数组有长度的限制；而通过一些数据结构的操作，如链表，则可以完成动态对象数组的操作，但是这些如果全部由开发人员来做，则肯定也是比较麻烦的。

类集框架恰好解决了上面的难题，所谓的类集就是一个动态的对象数组，是对一些实现好的数据结构进行了包装，这样在使用时就会非常方便，而且最重要的是类集框架本身不受对象数组长度的限制。

类集框架被设计成拥有以下几个特性：

（1）这种框架是高性能的，对基本类集（动态数组、链接表、树和散列表）的实现是高效率的。所以一般很少需要人工去对这些“数据引擎”编写代码。

（2）框架必须允许不同类型的类集以相同的方式和高度互操作方式工作。

（3）类集必须是容易扩展和修改的。为了实现这一目标，类集框架被设计成包含了一组标准接口。

提示：关于相同的方式及高度的解释。

在类集的操作中因为是使用类的形式实现的动态对象数组，所以对于任何对象所有的操作形式都是一样的。例如，只要是想向集合中增加内容，则一定使用 add()方法。高度一般指的是类集中的元素类型都是统一的，即一个集合中要么全都是 A 类对象，要么全都是 B 类对象。

13.1.2 类集框架主要接口

在整个 Java 类集中最常使用的类集接口是：Collection、List、Set、Map、Iterator、ListIterator、Enumeration、SortedSet、SortedMap、Queue、Map.Entry，这些接口的具体特点如表 13-1 所示。

表 13-1 类集框架接口的具体特点

序　号	接　口	描　述
1	Collection	是存放一组单值的最大接口，所谓的单值是指集合中的每个元素都是一个对象。一般很少直接使用此接口直接操作
2	List	是 Collection 接口的子接口，也是最常用的接口。此接口对 Collection 接口进行了大量的扩充，里面的内容是允许重复的
3	Set	是 Collection 接口的子类，没有对 Collection 接口进行扩充，里面不允许存放重复内容
4	Map	是存放一对值的最大接口，即接口中的每个元素都是一对，以 key→value 的形式保存
5	Iterator	集合的输出接口，用于输出集合中的内容，只能进行从前到后的单向输出
6	ListIterator	是 Iterator 的子接口，可以进行双向输出
7	Enumeration	是最早的输出接口，用于输出指定集合中的内容
8	SortedSet	单值的排序接口，实现此接口的集合类，里面的内容可以使用比较器排序
9	SortedMap	存放一对值的排序接口，实现此接口的集合类，里面的内容按照 key 排序，使用比较器排序
10	Queue	队列接口，此接口的子类可以实现队列操作
11	Map.Entry	Map.Entry 的内部接口，每个 Map.Entry 对象都保存着一对 key→value 的内容，每个 Map 接口中都保存有多个 Map.Entry 接口实例

这些接口中本身是存在继承关系的，其中部分接口的继承关系如图 13-1 所示。

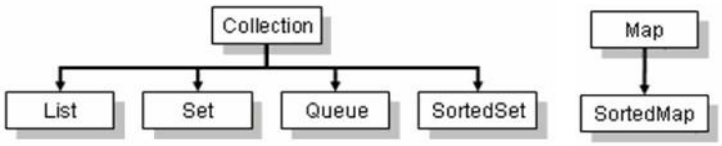


图 13-1　接口的继承关系

下面将介绍这些接口的作用及其常用子类。

提示：SortedXx 定义的接口都属于排序接口。

在 Java 类集中凡是以 Sorted 开头的全部都属于排序的接口，如 SortedSet、SortedMap。

13.2　Collection 接口

13.2.1　Collection 接口的定义

Collection 接口的定义如下：

```
public interface Collection<E> extends Iterable<E>
```

从接口的定义中可以发现，此接口使用了泛型的定义，在操作时必须指定具体的操作类型。这样可以保证类集操作的安全性，避免发生 ClassCastException 异常。

提示：JDK 1.5 之后类集才增加了泛型的支持。

在 JDK 1.5 之前的类集框架中可以存放任意的对象到集合中，这样一来在操作时就可能出现因为类型不统一而造成的 ClassCastException 异常。所以在 JDK 1.5 之后为了保证类集中所有元素的类型一致，将类集框架进行了升级，加入了泛型，这样就可以保证一个集合中的全部元素的类型是统一的。

Collection 接口是单值存放的最大父接口，可以向其中保存多个单值（单个的对象）数据。此接口定义了表 13-2 所示的方法定义。

表 13-2　Collection 接口的方法定义

序号	方　　法	类型	描　　述
1	public boolean add(E o)	普通	向集合中插入对象
2	public boolean addAll(Collection<? extends E> c)	普通	将一个集合的内容插入进来
3	public void clear()	普通	清除此集合中的所有元素
4	public boolean contains(Object o)	普通	判断某一个对象是否在集合中存在
5	public boolean containsAll(Collection<?> c)	普通	判断一组对象是否在集合中存在
6	public boolean equals(Object o)	普通	对象比较
7	public int hashCode()	普通	哈希码
8	public boolean isEmpty()	普通	集合是否为空
9	public Iterator<E> iterator()	普通	为 Iterator 接口实例化
10	public boolean remove(Object o)	普通	删除指定对象

续表

序号	方　法	类型	描　述
11	public boolean removeAll(Collection<?> c)	普通	删除一组对象
12	public boolean retainAll(Collection<?> c)	普通	保存指定内容
13	public int size()	普通	求出集合的大小
14	public Object[] toArray()	普通	将一个集合变为对象数组
15	public <T> T[] toArray(T[] a)	普通	指定好返回的对象数组类型

在一般的开发中，往往很少直接使用 Collection 接口进行开发，基本上都是使用其子接口。子接口主要有 List、Set、Queue 和 SortedSet。

提示：关于 Collection 接口很少直接使用的说明。

在 Java 最早的程序开发中提倡使用 Collection，这一点在 EJB 2.x（Enterprise JavaBean，专门用作分布式程序开发的一套标准规范）中体现得非常明显。但是随着 Java 的发展，为了让程序的开发及使用更加明确，例如，此集合中的内容是否可以重复、是否可以排序等，所以提倡直接使用 Collection 的子接口，这一点在 SUN 的开源项目——宠物商店中特别明显。

13.2.2 Collection 子接口的定义

Collection 接口虽然是集合的最大接口，但是如果直接使用 Collection 接口进行操作，则表示的操作意义不明确，所以在 Java 开发中不提倡直接使用 Collection 接口，主要的子接口介绍如下。

- List：可以存放重复的内容。
- Set：不能存放重复的内容，所有的重复内容靠 hashCode()和 equals()两个方法区分。
- Queue：队列接口。
- SortedSet：可以对集合中的数据进行排序。

13.3 List 接口

13.3.1 List 接口的定义

List 是 Collection 的子接口，其中可以保存各个重复的内容。此接口的定义如下：

```
public interface List<E> extends Collection<E>
```

但是与 Collection 不同的是，在 List 接口中大量地扩充了 Collection 接口，拥有了比 Collection 接口中更多的方法定义，其中有些方法还比较常用。表 13-3 列出了 List 中对 Collection 接口的扩展方法。

表 13-3　List 接口的扩展方法

序号	方　　法	类型	描　　述
1	public void add(int index, E element)	普通	在指定位置增加元素
2	public boolean addAll(int index, Collection<? extends E> c)	普通	在指定位置增加一组元素
3	E get(int index)	普通	返回指定位置的元素
4	public int indexOf(Object o)	普通	查找指定元素的位置
5	public int lastIndexOf(Object o)	普通	从后向前查找指定元素的位置
6	public ListIterator<E> listIterator()	普通	为 ListIterator 接口实例化
7	public E remove(int index)	普通	按指定的位置删除元素
8	public List<E> subList(int fromIndex, int toIndex)	普通	取出集合中的子集合
9	public E set(int index, E element)	普通	替换指定位置的元素

List 接口比 Collection 接口扩充了更多的方法，而且这些方法操作起来很方便。但是如果要想使用此接口，则需要通过其子类进行实例化。

13.3.2　List 接口的常用子类

1．新的子类：ArrayList

ArrayList 是 List 子类，可以直接通过对象的多态性为 List 接口实例化。此类的定义如下：

```
public class ArrayList<E> extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

从定义中可以发现 ArrayList 类继承了 AbstractList 类。AbstractList 类的定义如下：

```
public abstract class AbstractList<E>
extends AbstractCollection<E> implements List<E>
```

此类实现了 List 接口，所以可以直接使用 ArrayList 为 List 接口实例化。下面通过一些实例操作为读者讲解 List 接口中主要方法的使用。

（1）实例操作一：向集合中增加元素

要想完成此类操作，可以直接使用 Collection 接口中定义的两个方法。

- 增加一个元素：public boolean add(E o)。
- 增加一组元素：public boolean addAll(Collection<? extends E> c)。

也可以使用 List 扩充的 add()方法在指定位置处增加元素。

在指定位置处添加元素：public void add(int index,E element)。

范例：验证增加数据的操作

```
package org.lxh.demo13.listdemo;
import java.util.ArrayList;
import java.util.Collection;
import java.util.List;
```

```
public class ArrayListDemo01 {
    public static void main(String[] args) {
        List<String> allList = null ;                    // 定义List对象
        Collection<String> allCollection = null ;        // 定义Collection对象
        allList = new ArrayList<String>();               // 实例化List对象，只能是
                                                            String类型
        allCollection = new ArrayList<String>();         // 实例化Collection，只能
                                                            是String类型
        allList.add("Hello");                            // 从Collection继承的方法
        allList.add(0, "World");                         // 此方法为List扩充的方法
        System.out.println(allList);                     // 输出集合中的内容
        allCollection.add("LXH");                        // 增加数据
        allCollection.add("www.mldn.cn");                // 增加数据
        allList.addAll(allCollection);                   // 从Collection继承的方
                                                            法，增加一组对象
        allList.addAll(0, allCollection);                // 此方法是List自定义的，
                                                            增加一组对象
        System.out.println(allList);                     // 输出对象，调用
                                                            toString()方法
    }
}
```

程序运行结果：

```
[World, Hello]
[LXH, www.mldn.cn, World, Hello, MLDN, www.mldn.cn]
```

从程序的运行结果中可以发现，使用 List 中的 add(int index,E element)方法可以在集合中的指定位置增加元素，而其他的两个 add()方法只是在集合的最后进行内容的追加。

（2）实例操作二：删除元素

在类集中提供了专门的删除元素的方法，Collection 和 List 接口都分别定义了删除数据的方法。

- Collection 定义的方法
  - 每次删除一个对象：public boolean remove(Object o)。
  - 每次删除一组对象：public boolean removeAll(Collection<?> c)。
- List 扩展的方法
  - 删除指定位置的元素：public E remove(int index)。

范例：删除对象

```
package org.lxh.demo13.listdemo;
import java.util.ArrayList;
import java.util.List;
public class ArrayListDemo02 {
    public static void main(String[] args) {
```

```
        List<String> allList = null ;          // 声明List对象
        allList = new ArrayList<String>(); // 实例化List对象，只能是String
                                               类型
        allList.add("Hello");                  // 增加元素
        allList.add(0, "World");               // 此方法为List扩展的增加方法
        allList.add("MLDN");                   // 增加元素
        allList.add("www.mldn.cn");            // 增加元素
        allList.remove(0);                     // 删除指定位置的元素
        allList.remove("Hello");               // 删除指定内容的元素
        System.out.println(allList);           // 输出对象，调用toString()方法
    }
}
```

程序运行结果：

```
[MLDN, www.mldn.cn]
```

在集合中增加完数据后，可以通过下标或对象的方式直接对集合中的元素进行删除。

提示：关于使用 remove(Object o)方法删除对象的说明。

在集合中可以插入任意类型的对象，在本程序中是以 String 类的对象为例，所以在使用 remove(Object o)方法删除时可以直接删除；而对于自定义的类如果要通过此种方式删除，则必须在类中覆写 Object 类的 equals()及 hashCode()方法。这两个方法的使用在随后的章节中将为读者介绍。

（3）实例操作三：输出 List 中的内容

在 Collection 接口中定义了取得全部数据长度的方法 size()，而在 List 接口中存在取得集合中指定位置元素的操作 get(int index)，使用这两个方法即可输出集合中的全部内容。

范例：输出全部元素

```
package org.lxh.demo13.listdemo;
import java.util.ArrayList;
import java.util.List;
public class ArrayListDemo03 {
    public static void main(String[] args) {
        List<String> allList = null ;                    // 定义List接口对象
        allList = new ArrayList<String>();               // 实例化List对象，只
                                                            能是String类型
        allList.add("Hello");                            // 增加元素
        allList.add("Hello");                            // 增加元素
        allList.add(0, "World");                         // 增加元素
        allList.add("MLDN");                             // 增加元素
        allList.add("www.mldn.cn");                      // 增加元素
        System.out.print("由前向后输出：");                // 信息输出
        for (int i = 0; i < allList.size(); i++) {       // 循环输出集合内容
```

```
            System.out.print(allList.get(i) + "、");    // 通过下标取得集合中
                                                          的元素
        }
        System.out.print("\n由后向前输出：");
        for (int i = allList.size() - 1; i >= 0; i--) {// 循环输出集合内容
            System.out.print(allList.get(i) + "、");    // 通过下标取得集合中
                                                          的元素
        }
    }
}
```

程序运行结果：

```
由前向后输出：World、Hello、Hello、MLDN、www.mldn.cn、
由后向前输出：www.mldn.cn、MLDN、Hello、Hello、World、
```

从程序的运行结果中可以看出，在 List 集合中数据增加的顺序就是输出后的顺序，本身顺序不会发生改变。

（4）实例操作四：将集合变为对象数组

在 Collection 中定义了 toArray()方法，此方法可以将集合变为对象数组，但是由于在类集声明时已经通过泛型指定了集合中的元素类型，所以在接收时要使用泛型指定的类型。

范例：将集合变为对象数组

```
package org.lxh.demo13.listdemo;
import java.util.ArrayList;
import java.util.List;
public class ArrayListDemo04 {
    public static void main(String[] args) {
        List<String> allList = null ;                  // 声明List对象
        allList = new ArrayList<String>();             // 实例化List对象，只能是
                                                          String类型
        allList.add("Hello");                          // 增加元素
        allList.add(0, "World");                       // 增加元素
        allList.add("MLDN");                           // 增加元素
        allList.add("www.mldn.cn");                    // 增加元素
        String str[] = allList.toArray(new String[] {}); // 指定的泛型类型
        System.out.print("指定数组类型：");              // 信息输出
        for (int i = 0; i < str.length; i++) {         // 输出字符串数组中的内容
            System.out.print(str[i] + "、");           // 输出每一个元素
        }
        System.out.print("\n返回对象数组：");           // 信息输出
        Object obj[] = allList.toArray();              // 直接返回对象数组
        for (int i = 0; i < obj.length; i++) {         // 循环输出对象数组内容
            String temp = (String) obj[i];             // 每一个对象都是String类
                                                          型实例
```

```
            System.out.print(temp + "、");          // 输出每一个元素
        }
    }
}
```

程序运行结果：

```
指定数组类型：World、Hello、MLDN、www.mldn.cn、
返回对象数组：World、Hello、MLDN、www.mldn.cn、
```

（5）实例操作五：集合的其他相关操作

在 List 中还存在截取集合、查找元素位置、判断元素是否存在、集合是否为空等操作。下面直接测试以上的操作。

范例：测试其他操作

```
package org.lxh.demo13.listdemo;
import java.util.ArrayList;
import java.util.List;
public class ArrayListDemo05 {
    public static void main(String[] args) {
        List<String> allList = null ;                    // 声明List对象
        allList = new ArrayList<String>();               // 实例化List对象，只能是
                                                            String类型
        System.out.println("集合操作前是否为空？" + allList.isEmpty());
        allList.add("Hello");                            // 增加元素
        allList.add(0, "World");                         // 增加元素
        allList.add("MLDN");                             // 增加元素
        allList.add("www.mldn.cn");                      // 增加元素
        System.out.println(allList.contains("Hello") ?
                "\"Hello\"字符串存在！": "\"Hello\"字符串不存在！");
        List<String> allSub = allList.subList(2, 3);// 取出里面的部分集合
        System.out.print("集合截取：");
        for (int i = 0; i < allSub.size(); i++) {  // 截取部分集合
            System.out.print(allList.get(i) + "、");
        }
        System.out.println("") ;
        System.out.println("MLDN字符串的位置：" + allList.indexOf("MLDN"));
        System.out.println("集合操作后是否为空？" + allList.isEmpty());
    }
}
```

程序运行结果：

```
集合操作前是否为空？true
"Hello"字符串存在！
集合截取：World、
```

```
MLDN字符串的位置：2
集合操作后是否为空？false
```

List 集合在刚刚实例化之后因为还有为其增加内容，所以在使用 isEmpty()方法时返回的结果是 true，表示集合是空的，之后向集合中增加了 4 个元素，所以程序的最后再使用此方法判断时就返回 false，表示集合中已经存在内容。在程序中使用 contains()方法判断集合中是否存在指定的元素，如果存在，则输出存在的信息；反之，输出不存在的信息。在集合中也可以使用 subList()方法取出指定的子集合。

2．挽救的子类：Vector

在 List 接口中还有一个子类 Vector，Vector 类属于一个挽救的子类，从整个 Java 的集合发展历史来看，Vector 算是一个元老级的类，在 JDK 1.0 时就已经存在此类。到了 Java 2（JDK 1.2）之后重点强调了集合框架的概念，所以先后定义了很多的新接口（如 List 等），但是考虑到一大部分用户已经习惯了使用 Vector 类，所以 Java 的设计者就让 Vector 类多实现了一个 List 接口，这才将其保留下来。但是因为其是 List 子类，所以 Vector 类的使用与之前的并没有太大的区别。

Vector 类的定义如下：

```
public class Vector<E> extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

从 Vector 类的定义中可以发现，此类与 ArrayList 类一样也继承自 AbstractList 类。

范例：Vector 子类

```
package org.lxh.demo13.listdemo;
import java.util.List;
import java.util.Vector;
public class VectorDemo01 {
    public static void main(String[] args) {
        List<String> allList = null ;                    // 声明List对象
        allList = new Vector<String>();                  // 实例化List对象，只
                                                            能是String类型
        allList.add("Hello");                            // 增加元素
        allList.add(0, "World");                         // 增加元素
        allList.add("MLDN");                             // 增加元素
        allList.add("www.mldn.cn");                      // 增加元素
        for (int i = 0; i < allList.size(); i++) {       // 循环输出
            System.out.print(allList.get(i) + "、");     // 通过get()取出每一
                                                            个元素
        }
    }
}
```

程序运行结果：

```
World、Hello、MLDN、www.mldn.cn、
```

如果直接使用的是 List 接口进行操作，则在运行结果上与 ArrayList 本身没有任何的区别，但是因为 Vector 类出现较早，所以也定义了许多在 List 接口中没有定义的方法，这些方法的功能与 List 类似。例如，Vector 类中的 addElement(E o)方法，是最早的向集合中增加元素的操作，但是在 JDK 1.2 之后此方法的功能与 List 接口中 add(E o)方法是一致的。

范例：使用旧的方法

```
package org.lxh.demo13.listdemo;
import java.util.Vector;
public class VectorDemo02 {
    public static void main(String[] args) {
        Vector<String> allList = new Vector<String>();    // 实例化Vector对象
        allList.addElement("Hello");                      // 此方法为Vector自己定义
        allList.addElement("MLDN");                       // 此方法为Vector自己定义
        allList.addElement("www.mldn.cn");                // 此方法为Vector自己定义
        for (int i = 0; i < allList.size(); i++) {        // 循环输出
            System.out.print(allList.get(i) + "、");      // 取得集合中的每一个元素
        }
    }
}
```

1、 掌握了以上的操作方法，实际上 List 接口的作用就明白了，而且之后的全部类集操作中也基本上都是以这些方法为操作的标准，只是各个的特性不一样，例如：List 中允许有重复元素。

2、 ArrayList 和 Vector 类的区别

程序运行结果：

```
Hello、MLDN、www.mldn.cn、
```

3．子类的差异：ArrayList 与 Vector 的区别

在之前的程序中，ArrayList 和 Vector 在操作结果上没有什么特别大的区别，那么到底该使用哪一个呢？读者可以通过表 13-4 进行总结。

表 13-4　ArrayList 与 Vector 类的主要区别

序号	比较点	ArrayList	Vector
1	推出时间	JDK 1.2 之后推出的，属于新的操作类	JDK 1.0 时推出，属于旧的操作类
2	性能	采用异步处理方式，性能更高	采用同步处理方式，性能较低
3	线程安全	属于非线程安全的操作类	属于线程安全的操作类
4	输出	只能使用 Iterator、foreach 输出	可以使用 Iterator、foreach、Enumeration 输出

以上对两个类进行了简单的对比，但是从实际的应用开发来看，ArrayList 类使用较多，读者应重点掌握。

4．LinkedList 子类与 Queue 接口

LinkedList 表示的是一个链表的操作类，即 Java 中已经为开发者提供好了一个链表程

序，开发者直接使用即可，无须再重新开发。LinkedList 类的定义如下：

```
public class LinkedList<E> extends AbstractSequentialList<E>
implements List<E>, Queue<E>, Cloneable, Serializable
```

从此类的定义中可以发现，此类虽然实现了 List 接口，但也同时实现了 Queue 接口。Queue 表示的是队列操作接口，采用 FIFO（先进先出）的方式操作，就好像一对人排队那样，队列中有队头和队尾，队头永远指向新加入的对象，如图 13-2 所示。



图 13-2　一对人

提示：先进先出（First Input Fist Output，FIFO）。

先进先出指的是在一个集合中，会按照顺序增加内容，在输出时先进入的数据会首先输出。与之对应的还有另外一种先进后出，指的是先进入的数据最后取出，而后进入的数据最先取出，在随后的栈（Stack）中会为读者讲解。

Queue 接口是 Collection 的子接口，其定义如下：

```
public interface Queue<E> extends Collection<E>
```

Queue 接口也可以增加元素并输出。此接口的方法定义如表 13-5 所示。

表 13-5　Queue 接口定义的方法

序号	方　　法	类型	描　　述
1	public E element()	普通	找到链表的表头
2	public boolean offer(E o)	普通	将指定元素增加到链表的结尾
3	public E peek()	普通	找到但并不删除链表的头
4	public E poll()	普通	找到并删除此链表的头
5	public E remove ()	普通	检索并移除表头

在 LinkedList 类中除了实现表 13-5 所示的各种方法外，还提供了如表 13-6 所示的几个链表操作方法。

表 13-6　LinkedList 中操作链表的部分方法

序号	方　　法	类型	描　　述
1	public void addFirst(E o)	普通	在链表开头增加元素
2	public void addLast(E o)	普通	在链表结尾增加元素
3	public boolean offer(E o)	普通	将指定元素增加到链表的结尾
4	public E removeFirst()	普通	删除链表的第一个元素
5	public E removeLast()	普通	删除链表的最后一个元素

下面编写几个程序，让读者了解此类的操作。

（1）实例操作一：在链表的开头和结尾增加数据

为了达到操作链表的目的，必须直接使用 LinkedList 类。因为 List 接口中没有表 13-5 中所定义的方法。

范例：为链表增加数据

```
package org.lxh.demo13.listdemo;
import java.util.LinkedList;
public class LinkedListDemo01 {
    public static void main(String[] args) {
        LinkedList<String> link = new LinkedList<String>();
        link.add("A");                                  // 向链表中增加数据
        link.add("B");                                  // 向链表中增加数据
        link.add("C");                                  // 向链表中增加数据
        System.out.println("初始化链表：" + link);      // 输出链表内容，调用
                                                           toString()
        link.addFirst("X");                             // 在链表的表头增加内容
        link.addLast("Y");                              // 在链表的表尾增加内容
        System.out.println("增加头和尾之后的链表：" + link);// 输出链表内容，调
                                                                用toString()
    }
}
```

程序运行结果：

```
初始化链表：[A, B, C]
增加头和尾之后的链表：[X, A, B, C, Y]
```

（2）实例操作二：找到链表头

在 LinkedList 中存在很多种找到链表头的操作，其中最常用的介绍如下。

- 找到表头：public E element()。
- 找到不删除表头：public E peek()。
- 找到并删除表头：public E poll()。

范例：找到表头

```
package org.lxh.demo13.listdemo;
import java.util.LinkedList;
public class LinkedListDemo02 {
    public static void main(String[] args) {
        LinkedList<String> link = new LinkedList<String>();
        link.add("A");                                  // 向链表中增加数据
        link.add("B");                                  // 向链表中增加数据
        link.add("C");                                  // 向链表中增加数据
        System.out.println("1-1、element()方法找到表头：" + link.element());
        System.out.println("1-2、找完之后的链表内容：" + link);
```

```
            System.out.println("2-1、peek()方法找到表头：" + link.peek());
            System.out.println("2-2、找完之后的链表内容：" + link);
            System.out.println("3-1、poll()方法找到表头：" + link.poll());
            System.out.println("3-2、找完之后的链表内容：" + link);
        }
    }
```

程序运行结果：

```
1-1、element()方法找到表头：A
1-2、找完之后的链表内容：[A, B, C]
2-1、peek()方法找到表头：A
2-2、找完之后的链表内容：[A, B, C]
3-1、poll()方法找到表头：A
3-2、找完之后的链表内容：[B, C]
```

（3）实例操作三：以先进先出的方式取出全部的数据

在 LinkedList 类中存在 poll()方法，通过循环此操作，就可以把内容全部取出（以先进先出（FIFO）的方式）。

范例：以 FIFO 方式取出内容

```
package org.lxh.demo13.listdemo;
import java.util.LinkedList;
public class LinkedListDemo03 {
    public static void main(String[] args) {
        LinkedList<String> link = new LinkedList<String>();
        link.add("A");                                    // 向链表中增加数据
        link.add("B");                                    // 向链表中增加数据
        link.add("C");                                    // 向链表中增加数据
        System.out.print("以FIFO的方式输出：");             // 信息输出
        for (int i = 0; i < link.size() + 1; i++) {// 循环输出
            System.out.print(link.poll() + "、");         // 取出表头
        }
    }
}
```

程序运行结果：

```
以FIFO的方式输出：A、B、C
```

13.4 Set 接口

13.4.1 Set 接口的定义

Set 接口也是 Collection 接口的子接口，但是与 Collection 或 List 接口不同的是，Set 接

口中不能加入重复的元素。Set 接口的定义如下：

```
public interface Set<E> extends Collection<E>
```

从定义上可以发现，Set 接口与 List 接口的定义并没有太大的区别。但是 Set 接口的主要方法与 Collection 是一致的，也就是说 Set 接口并没有对 Collection 接口进行扩充，只是比 Collection 接口的要求更加严格了，不能增加重复元素。

Set 接口的实例无法像 List 接口那样可以进行双向输出，因为此接口没有提供像 List 接口定义的 get(int index)方法。

13.4.2　Set 接口的常用子类

1．散列的存放：HashSet

HashSet 是 Set 接口的一个子类，主要的特点是：里面不能存放重复元素，而且采用散列的存储方式，所以没有顺序。

范例：验证 HashSet 类

```
package org.lxh.demo13.setdemo;
import java.util.HashSet;
import java.util.Set;
public class HashSetDemo01 {
    public static void main(String[] args) {
        Set<String> allSet = new HashSet<String>();
        allSet.add("A") ;                    // 增加元素
        allSet.add("B") ;                    // 增加元素
        allSet.add("C") ;                    // 增加元素
        allSet.add("C") ;                    // 重复元素，不能加入
        allSet.add("C") ;                    // 重复元素，不能加入
        allSet.add("D") ;                    // 增加元素
        allSet.add("E") ;                    // 增加元素
        System.out.println(allSet) ;         // 输出集合对象，调用toString()
    }
}
```

程序运行结果：

```
[D, A, C, B, E]
```

从程序的运行结果中可以清楚地看出，对于重复元素只会增加一次，而且程序运行时向集合中加入元素的顺序并不是集合中的保存顺序，证明 HashSet 类中的元素是无序排列的。

2．有序的存放：TreeSet

如果想对输入的数据进行有序排列，则要使用 TreeSet 子类。TreeSet 类的定义如下：

```
public class TreeSet<E> extends AbstractSet<E>
implements SortedSet<E>, Cloneable, Serializable
```

TreeSet 类也是继承了 AbstractSet 类，此类的定义如下：

```
public abstract class AbstractSet<E>
extends AbstractCollection<E>
implements Set<E>
```

范例：验证 TreeSet 类

```
package org.lxh.demo13.setdemo;
import java.util.Set;
import java.util.TreeSet;
public class TreeSetDemo01 {
    public static void main(String[] args) {
        Set<String> allSet = new TreeSet<String>();
        allSet.add("C") ;                          // 加入元素
        allSet.add("C") ;                          // 重复元素，不能加入
        allSet.add("C") ;                          // 重复元素，不能加入
        allSet.add("D") ;                          // 加入元素
        allSet.add("B") ;                          // 加入元素
        allSet.add("A") ;                          // 加入元素
        allSet.add("E") ;                          // 加入元素
        System.out.println(allSet) ;               // 输出集合，调用toString()
    }
}
```

程序运行结果：

```
[A, B, C, D, E]
```

程序在向集合中插入数据时是没有顺序的，但是输出之后数据是有序的，所以 TreeSet 是可以排序的子类。

3. 关于 TreeSet 的排序说明

既然 TreeSet 本身是可以排序的，那么现在定义一个自己的类，是否也可以进行排序的操作呢？

范例：自定义类排序

```
package org.lxh.demo13.setdemo;
import java.util.Set;
import java.util.TreeSet;
class Person {                                     // 定义Person类
    private String name;                           // 定义name属性
    private int age;                               // 定义age属性
    public Person(String name, int age) {          // 通过构造方法为属性赋值
        this.name = name;                          // 为name属性赋值
        this.age = age;                            // 为age属性赋值
    }
```

```
    public String toString() {                          // 覆写toString()方法
        return "姓名：" + this.name + "；年龄：" + this.age;
    }
}
public class TreeSetDemo02 {
    public static void main(String[] args) {
        Set<Person> allSet = new TreeSet<Person>();// 实例化Set集合，类型为
                                                       Person
        allSet.add(new Person("张三", 30));          // 加入元素
        allSet.add(new Person("李四", 31));          // 加入元素
        allSet.add(new Person("王五", 32));          // 加入元素
        allSet.add(new Person("王五", 32));          // 重复元素，不能加入
        allSet.add(new Person("王五", 32));          // 重复元素，不能加入
        allSet.add(new Person("赵六", 33));          // 加入元素
        allSet.add(new Person("孙七", 33));          // 年龄重复
        System.out.println(allSet);                  // 输出集合内容
    }
}
```

程序运行时出现以下错误：

```
Exception in thread "main" java.lang.ClassCastException: org.lxh.demo13.
setdemo.Person cannot be cast to java.lang.Comparable
    at java.util.TreeMap.put(Unknown Source)
    at java.util.TreeSet.add(Unknown Source)
    at org.lxh.demo13.setdemo.TreeSetDemo02.main(TreeSetDemo02.java:19)
```

以上程序代码出现了类转换异常，会出现这样的问题，是因为 TreeSet 中的元素是有序存放，所以对于一个对象必须指定好其排序规则，且 TreeSet 中的每个对象所在的类都必须实现 Comparable 接口才可以正常使用。

范例：指定排序规则

```
package org.lxh.demo13.setdemo;
import java.util.Set;
import java.util.TreeSet;
class Person implements Comparable<Person> {    // 定义Person类，实现比较器
    private String name;                        // 定义name属性
    private int age;                            // 定义age属性
    public Person(String name, int age) {       // 通过构造方法为属性赋值
        this.name = name;                       // 为name属性赋值
        this.age = age;                         // 为age属性赋值
    }
    public String toString() {                  // 覆写toString()方法
        return "姓名：" + this.name + "；年龄：" + this.age;
    }
```

```
    public int compareTo(Person per) {                    // 覆写compareTo()方法，
                                                             指定排序规则
        if (this.age > per.age) {                         // 通过年龄排序
            return 1;
        } else if (this.age < per.age) {
            return -1;
        }else{
            return 0 ;
        }
    }
}
public class TreeSetDemo03 {
    public static void main(String[] args) {
        Set<Person> allSet = new TreeSet<Person>();// 实例化Set接口对象
        allSet.add(new Person("张三", 30));           // 加入元素
        allSet.add(new Person("李四", 31));           // 加入元素
        allSet.add(new Person("王五", 32));           // 加入元素
        allSet.add(new Person("王五", 32));           // 重复元素，不能加入
        allSet.add(new Person("王五", 32));           // 重复元素，不能加入
        allSet.add(new Person("赵六", 33));           // 加入元素
        allSet.add(new Person("孙七", 33));           // 年龄重复
        System.out.println(allSet);                   // 输出集合，调用
                                                         toString()
    }
}
```

程序运行结果：

[姓名：张三；年龄：30，姓名：李四；年龄：31，姓名：王五；年龄：32，姓名：赵六；年龄：33]

从程序的运行结果中可以发现，重复的“王五”对象只有一个了；“赵六”和“孙七”的数据中姓名并不重复，只是年龄重复了，但是“孙七”却没有加入到集合中。这是由于采用了比较器造成的，因为比较器操作时如果某个属性没有进行比较的指定，则也会认为是同一个对象，所以此时应该在 Person 类的 compareTo 方法中增加按姓名比较。

范例：修改 Person 的比较器

```
package org.lxh.demo13.setdemo;
import java.util.Set;
import java.util.TreeSet;
class Person implements Comparable<Person> {    // 定义Person类，实现比较器
    private String name;                        // 定义name属性
    private int age;                            // 定义age属性
    public Person(String name, int age) {       // 通过构造方法为属性赋值
        this.name = name;                       // 为name属性赋值
```

```
            this.age = age;                                    // 为age属性赋值
        }
        public String toString() {                             // 覆写toString()方法
            return "姓名：" + this.name + "；年龄：" + this.age;
        }
        public int compareTo(Person per) {                     // 覆写compareTo()方法，
                                                                  指定排序规则
            if (this.age > per.age) {                          // 通过年龄排序
                return 1;
            } else if (this.age < per.age) {
                return -1;
            }else{
                return this.name.compareTo(per.name) ; // 增加字符串的比较
            }
        }
    }
    public class TreeSetDemo04 {
        public static void main(String[] args) {
            Set<Person> allSet = new TreeSet<Person>();// 实例化Set接口对象
            allSet.add(new Person("张三", 30));            // 加入元素
            allSet.add(new Person("李四", 31));            // 加入元素
            allSet.add(new Person("王五", 32));            // 加入元素
            allSet.add(new Person("王五", 32));            // 重复元素，不能加入
            allSet.add(new Person("王五", 32));            // 重复元素，不能加入
            allSet.add(new Person("赵六", 33));            // 加入元素
            allSet.add(new Person("孙七", 33));            // 年龄重复
            System.out.println(allSet);                    // 输出集合，调用
                                                              toString()
        }
    }
```

程序运行结果：

[姓名：张三；年龄：30，姓名：李四；年龄：31，姓名：王五；年龄：32，姓名：孙七；年龄：33，姓名：赵六；年龄：33]

以上的程序运行结果中出现了“孙七”，而且去掉了重复的内容，但此时的重复内容去掉，并不是真正意义上的去掉重复元素。因为此时靠的是 Comparable 完成的，而如果换成 HashSet 则也会出现重复的内容，所以要想真正地去掉重复元素，则必须深入研究 Object 类。

4．关于重复元素的说明

在讲解之前，先来观察以下的代码，并观察其输出。

范例：加入重复对象

```
package org.lxh.demo13.setdemo;
import java.util.HashSet;
import java.util.Set;
class Person {                                          // 定义Person类
    private String name;                                // 定义name属性
    private int age;                                    // 定义age属性
    public Person(String name, int age) {               // 通过构造方法为属性赋值
        this.name = name;                               // 为name属性赋值
        this.age = age;                                 // 为age属性赋值
    }
    public String toString() {                          // 覆写toString()方法
        return "姓名：" + this.name + "；年龄：" + this.age;
    }
}
public class RepeatDemo01 {
    public static void main(String[] args) {
        Set<Person> allSet = new HashSet<Person>();// 实例化Set接口对象
        allSet.add(new Person("张三", 30));             // 加入元素
        allSet.add(new Person("李四", 31));             // 加入元素
        allSet.add(new Person("王五", 32));             // 加入元素
        allSet.add(new Person("王五", 32));             // 重复元素，不能加入
        allSet.add(new Person("王五", 32));             // 重复元素，不能加入
        allSet.add(new Person("赵六", 33));             // 加入元素
        allSet.add(new Person("孙七", 33));             // 年龄重复
        System.out.println(allSet);                     // 输出集合，调用
                                                           toString()
    }
}
```

程序运行结果：

[姓名：王五；年龄：32，姓名：李四；年龄：31，姓名：王五；年龄：32，姓名：赵六；年龄：33，姓名：张三；年龄：30，姓名：孙七；年龄：33，姓名：王五；年龄：32]

从程序的运行结果中可以发现，“王五”的内容重复了，也就是说，此时的程序并没有像 Set 接口规定的那样是不允许有重复元素的，而如果此时想要去掉重复元素，则必须首先进行对象是否重复的判断，而要想进行这样的判断则一个类中就必须覆写 Object 类中的 equals()方法，才能完成对象是否相等的判断，但是只覆写 equals()方法是不够的，还需要覆写 hashCode()方法，此方法表示一个哈希编码，可以简单地理解为表示一个对象的编码。一般的哈希码是通过公式进行计算的，可以将类中的全部属性进行适当的计算，以求出一个不会重复的哈希码。

范例：去掉重复元素

```
package org.lxh.demo13.setdemo;
import java.util.HashSet;
```

```
import java.util.Set;
class Person {                                          // 定义Person类
    private String name;                                // 定义name属性
    private int age;                                    // 定义age属性
    public Person(String name, int age) {               // 通过构造方法为属性赋值
        this.name = name;                               // 为name属性赋值
        this.age = age;                                 // 为age属性赋值
    }
    public boolean equals(Object obj) {                 // 覆写equals()方法
        if (this == obj) {                              // 地址相等
            return true;                                // 是同一个对象
        }
        if (!(obj instanceof Person)) {                 // 传递进来的不是本类的对象
            return false;                               // 不是同一个对象
        }
        Person p = (Person) obj;                        // 进行向下转型
        if (this.name.equals(p.name) && this.age == p.age) {
            return true ;                               // 属性依次比较
        }else{                                          // 全部属性相等，是同一对象
            return false ;                              // 属性不相等，不是同一对象
        }
    }
    public int hashCode(){                              // 覆写hashCode()方法
        return this.name.hashCode() * this.age ;        // 指定编码公式
    }
    public String toString() {                          // 覆写toString()方法
        return "姓名: " + this.name + "; 年龄: " + this.age;
    }
}
public class RepeatDemo02 {
    public static void main(String[] args) {
        Set<Person> allSet = new HashSet<Person>();// 实例化Set接口对象
        allSet.add(new Person("张三", 30));         // 加入元素
        allSet.add(new Person("李四", 31));         // 加入元素
        allSet.add(new Person("王五", 32));         // 加入元素
        allSet.add(new Person("王五", 32));         // 重复元素，不能加入
        allSet.add(new Person("王五", 32));         // 重复元素，不能加入
        allSet.add(new Person("赵六", 33));         // 加入元素
        allSet.add(new Person("孙七", 33));         // 年龄重复
        System.out.println(allSet);                 // 输出集合，调用
                                                        toString()
    }
}
```

程序运行结果：

[姓名：王五；年龄：32，姓名：李四；年龄：31，姓名：张三；年龄：30，姓名：孙七；年龄：33，姓名：赵六；年龄：33]

从最后输出的结果中可以发现，集合中的重复内容消失了，就是因为 equals 和 hashCode 共同作用的结果。

提示：Object 类总结。

经过以上程序代码的学习，Object 类中的全部方法就介绍完了，相信读者应该清楚地知道了各个方法的作用。在实际的开发中经常会碰到区分同一对象的问题，所以，一个完整的类最好覆写 Object 类的 hashCode()、equals()、toString() 3 个方法。

13.5 SortedSet 接口

从 TreeSet 类的定义中可以发现，TreeSet 中实现了 SortedSet 接口，此接口主要用于排序操作，即实现此接口的子类都属于排序的子类。SortedSet 接口定义如下：

```
public interface SortedSet<E> extends Set<E>
```

发现此接口也继承了 Set 接口。此接口中定义了如表 13-7 所示的方法。

表 13-7 SortedSet 接口中定义的方法

序号	方　法	类型	描　述
1	public Comparator<? super E> comparator()	普通	返回与排序有关联的比较器
2	public E first()	普通	返回集合中的第一个元素
3	public SortedSet<E> headSet(E toElement)	普通	返回从开始到指定元素的集合
4	public E last()	普通	返回最后一个元素
5	public SortedSet<E> subSet(E fromElement,E toElement)	普通	返回指定对象间的元素
6	public SortedSet<E> tailSet(E fromElement)	普通	从指定元素到最后

范例：验证 SortedSet 接口

```
package org.lxh.demo13.setdemo;
import java.util.SortedSet;
import java.util.TreeSet;
public class TreeSetDemo05 {
    public static void main(String[] args) {
        SortedSet<String> allSet = new TreeSet<String>();  // 为SortedSet
                                                              实例化
        allSet.add("A");                                   // 增加元素
        allSet.add("B");                                   // 增加元素
        allSet.add("C");                                   // 增加元素
        allSet.add("C");                                   // 重复元素，不能
                                                              加入
```

```
            allSet.add("C");                                    // 重复元素，不能加入
            allSet.add("D");                                    // 增加元素
            allSet.add("E");                                    // 增加元素
            System.out.println("第一个元素：" + allSet.first());
            System.out.println("最后一个元素：" + allSet.last());
            System.out.println("headSet元素：" + allSet.headSet("C"));
            System.out.println("tailSet元素：" + allSet.tailSet("C"));
            System.out.println("subSet元素：" + allSet.subSet("B","D"));
        }
    }
```

程序运行结果：

```
第一个元素：A
最后一个元素：E
headSet元素：[A, B]
tailSet元素：[C, D, E]
subSet元素：[B, C]
```

13.6　集合的输出

从之前讲解的集合操作中读者应该可以发现，如果要输出 Collection、Set 集合中的内容，可以将其转换为对象数组输出，而使用 List 则可以直接通过 get()方法输出，但是这些都不是集合的标准输出方式。在类集中提供了以下 4 种常见的输出方式。

- Iterator：迭代输出，是使用最多的输出方式。
- ListIterator：是 Iterator 的子接口，专门用于输出 List 中的内容。
- Enumeration：是一个旧的接口，功能与 Iterator 类似。
- foreach：JDK 1.5 之后提供的新功能，可以输出数组或集合。

注意：4 种输出操作以 Iterator 为操作的标准。

以上虽然提供了 4 种输出的操作，但是从实际的使用上来看，Iterator 接口是最常使用的输出形式，所以读者一定要重点掌握此接口的使用。

13.6.1　迭代输出：Iterator

1. Iterator 接口简介

在讲解 Iterator 接口之前，先向读者说一下个人的经验总结：“在使用集合输出时必须形成一个思路：‘只要是碰到了集合输出的操作，就一定使用 Iterator 接口’，因为这是最标准的做法”。

Iterator 是专门的迭代输出接口，所谓的迭代输出就是将元素一个个进行判断，判断其

是否有内容，如果有内容则把内容取出，如图 13-3 所示。

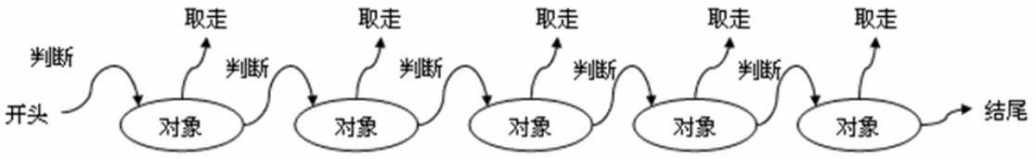


图 13-3　迭代输出原理

了解了其基本原理后，下面来看一下 Iterator 接口的定义：

```
public interface Iterator<E>
```

Iterator 接口在使用时也需要指定泛型，当然在此处指定的泛型类型最好与集合中的泛型类型一致。此接口中定义了 3 个方法，如表 13-8 所示。

表 13-8　Iterator 接口中的常用方法

序号	方　　法	类型	描　　述
1	public boolean hasNext()	普通	判断是否有下一个值
2	public E next()	普通	取出当前元素
3	public void remove()	普通	移除当前元素

下面通过一些实例为读者讲解以上方法的作用，以及使用以上方法操作时的注意点。

2．Iterator 接口的相关操作

（1）实例操作一：输出 Collection 中的全部内容

Iterator 是一个接口，可以直接使用 Collection 接口中定义的 iterator()方法为其实例化。既然 Collection 接口中存在了此方法，则 List 和 Set 接口中也一定存在此方法，所以也同样可以使用 Iterator 接口输出。

范例：进行输出

```
package org.lxh.demo13.iteratordemo;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
public class IteratorDemo01 {
    public static void main(String[] args) {
        List<String> all = new ArrayList<String>(); // 实例化List接口
        all.add("hello");                           // 增加元素
        all.add("_");                               // 增加元素
        all.add("world");                           // 增加元素
        Iterator<String> iter = all.iterator();     // 直接实例化Iterator
                                                       接口
        while (iter.hasNext()) {                    // 依次判断
            System.out.print(iter.next() + "、");   // 输出内容
        }
```

```
    }
}
```

程序运行结果：

```
hello'_'world、
```

以上的输出代码是 Iterator 的标准操作形式，将集合中的内容一个个地循环输出。此种输出也是必须掌握的形式。

（2）实例操作二：使用 Iterator 删除指定内容

在 Iterator 接口中除了可以输出内容之外，还可以删除当前的内容，直接使用 remove() 方法即可。

范例：删除元素

```
package org.lxh.demo13.iteratordemo;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
public class IteratorDemo02 {
    public static void main(String[] args) {
        List<String> all = new ArrayList<String>();     // 实例化List接口
        all.add("hello");                                // 增加元素
        all.add("_");                                    // 增加元素
        all.add("world");                                // 增加元素
        Iterator<String> iter = all.iterator();          // 直接实例化
                                                            Iterator接口
        while (iter.hasNext()) {                         // 依次输出
            String str = iter.next();                    // 取出内容
            if ("_".equals(str)) {                       // 判断内容是否是“_”
                iter.remove();                           // 删除当前元素
            } else {
                System.out.print(str + "、");            // 输出元素内容
            }
        }
        System.out.println("\n删除之后的集合：" + all);   // 输出集合内容，调用
                                                            toString()方法
    }
}
```

程序运行结果：

```
hello、world、
删除之后的集合：[hello, world]
```

（3）实例操作三：迭代输出时删除元素的注意点

正常情况下，一个集合要把内容交给 Iterator 输出，但是集合操作中也存在一个 remove()

方法，如果在使用 Iterator 输出时集合自己调用了删除方法，则会出现运行时的错误。

范例：不正确的删除方法

```
package org.lxh.demo13.iteratordemo;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
public class IteratorDemo03 {
    public static void main(String[] args) {
        List<String> all = new ArrayList<String>();      // 实例化List接口
        all.add("hello");                                 // 增加元素
        all.add("_");                                     // 增加元素
        all.add("world");                                 // 增加元素
        Iterator<String> iter = all.iterator();           // 直接实例化
                                                             Iterator接口
        while (iter.hasNext()) {
            String str = iter.next();                     // 取出内容
            if ("_".equals(str)) {                        // 判断内容是否是“_”
                all.remove(str);                          // 删除当前元素，使用
                                                             集合删除
            } else {
                System.out.print(str + "、");             // 输出元素内容
            }
        }
        System.out.println("\n删除之后的集合：" + all);   // 输出集合内容，调用
                                                             toString()方法
    }
}
```

程序运行结果：

```
hello、
删除之后的集合：[hello, world]
```

从程序的运行结果中可以发现，内容确实被删除了，但是迭代输出在内容删除之后就终止了。因为集合本身的内容被破坏掉，所以迭代将出现错误，会停止输出。

13.6.2 双向迭代输出：ListIterator

1. ListIterator 接口简介

Iterator 接口的主要功能是由前向后单向输出，而此时如果想实现由后向前或是由前向后的双向输出，则必须使用 Iterator 的子接口——ListIterator。

ListIterator 接口定义如下：

```
public interface ListIterator<E> extends Iterator<E>
```

此接口定义了比 Iterator 接口中更多的方法，方法定义如表 13-9 所示。

表 13-9　ListIterator 接口中的常用方法

序号	方　　法	类型	描　　述
1	public boolean hasNext()	普通	判断是否有下一个值
2	public E next()	普通	取出当前元素
3	public void remove()	普通	移除当前元素
4	public void add(E o)	普通	将指定元素增加集合
5	public boolean hasPrevious()	普通	判断是否有上一个元素
6	public E previous()	普通	取出当前元素
7	public int nextIndex()	普通	返回下一个元素的索引号
8	public int previousIndex()	普通	返回上一个元素的索引号
9	public void set(E o)	普通	替换元素

与 Iterator 接口不同的是，ListIterator 接口只能通过 List 接口实例化，即只能输出 List 接口中的内容。在 List 接口中定义了可以为 ListIterator 接口的实例化方法。

```
public ListIterator<E> listIterator()
```

2. ListIterator 接口的相关操作

（1）实例操作一：进行双向迭代

使用 ListIterator 接口中的 hasPrevious()方法由后向前判断，并使用 previous()方法取出前一个元素。

范例：进行双向迭代

```
package org.lxh.demo13.iteratordemo;
import java.util.ArrayList;
import java.util.List;
import java.util.ListIterator;
public class ListIteratorDemo01 {
    public static void main(String[] args) {
        List<String> all = new ArrayList<String>();    // 实例化List接口
        all.add("hello");                               // 增加元素
        all.add("_");                                   // 增加元素
        all.add("world");                               // 增加元素
        ListIterator<String> iter = all.listIterator();// 实例化ListIterator
                                                           接口
        System.out.print("由前向后输出：");              // 输出信息
        while (iter.hasNext()) {                        // 由前向后输出
            String str = iter.next();                   // 取出内容
            System.out.print(str + "、");               // 输出内容
        }
        System.out.print("\n由后向前输出：");            // 输出信息
```

```
            while (iter.hasPrevious()) {                    // 由前向后输出
                String str = iter.previous();               // 取出内容
                System.out.print(str + "、");                // 输出内容
            }
        }
    }
```

程序运行结果：

```
由前向后输出：hello、_、world、
由后向前输出：world、_、hello、
```

以上程序实现了双向的迭代输出，但是此种输出方式只有 List 接口才可以做到。

注意：由后向前输出时必须先由前向后输出。

在使用 ListIterator 接口进行双向输出时，如果想完成由后向前的输出，则一定要先进行由前向后的输出。

（2）实例操作二：增加及替换元素

使用 add()或 set()方法可以增加或替换集合中的元素，但是这样的操作在开发中不建议使用。

范例：增加及替换集合中的元素

```
package org.lxh.demo13.iteratordemo;
import java.util.ArrayList;
import java.util.List;
import java.util.ListIterator;
public class ListIteratorDemo02 {
    public static void main(String[] args) {
        List<String> all = new ArrayList<String>();    // 实例化List接口
        all.add("hello");                               // 增加元素
        all.add("_");                                   // 增加元素
        all.add("world");                               // 增加元素
        ListIterator<String> iter = all.listIterator();// 实例化ListIterator
                                                           接口
        System.out.print("由前向后输出：");              // 信息输出
        while (iter.hasNext()) {                        // 由前向后输出
            String str = iter.next();                   // 取出内容
            System.out.print(str + "、");               // 输出内容
            iter.set("LI-" + str);                      // 替换元素
        }
        System.out.print("\n由后向前输出：");            // 信息输出
        iter.add("LXH");                                // 增加元素
        while (iter.hasPrevious()) {                    // 由前向后输出
            String str = iter.previous();               // 取出内容
```

```
            System.out.print(str + "、");                    // 输出内容
        }
    }
}
```

程序运行结果：

```
由前向后输出：hello、_、world、
由后向前输出：LXH、LI-world、LI-_、LI-hello、
```

在 ListIterator 接口中使用 set()方法修改了每个元素的内容，而且也可以使用 ListIterator 接口中的 add()方法向集合中增加元素。

13.6.3　Java 新支持：foreach

之前已经向读者介绍过 foreach 的基本用法，使用 foreach 除了可以完成数组的输出，对于集合也同样支持。Foreach 的输出格式如 13-1 所示。

【格式 13-1　foreach 输出格式】

```
for(类 对象 : 集合){
    // 集合操作
}
```

范例：使用 foreach 输出

```
package org.lxh.demo13.iteratordemo;
import java.util.ArrayList;
import java.util.List;
public class ForeachDemo01 {
    public static void main(String[] args) {
        List<String> all = new ArrayList<String>();     // 实例化List接口
        all.add("hello");                                // 增加元素
        all.add("_");                                    // 增加元素
        all.add("world");                                // 增加元素
        for (String str : all) {                         // 使用foreach输出
            System.out.print(str + "、");                // 输出内容
        }
    }
}
```

程序运行结果：

```
hello、_、world、
```

虽然 foreach 输出的功能强大，而且操作的代码也比较简单，但是从实际的开发上来讲，本书还是建议读者使用 Iterator 接口完成输出功能。

13.6.4 废弃的接口：Enumeration

Enumeration 接口是 JDK 1.0 时就推出的，是最早的迭代输出接口，最早使用 Vector 时就是使用 Enumeration 接口进行输出的。Enumeration 接口的定义如下：

```
public interface Enumeration<E>
```

虽然 Enumeration 是一个旧的类，但是在 JDK 1.5 之后为 Enumeration 类进行了扩充，增加了泛型的操作应用。主要方法如表 13-10 所示。

表 13-10 Enumeration 接口中的常用方法

序号	方　法	类型	描　述
1	public boolean hasMoreElements()	普通	判断是否有下一个值
2	public E nextElement()	普通	取出当前元素

以上方法的功能与 Iterator 类似，只是 Iterator 中存在删除数据的方法，而此接口并不存在删除操作，而且可以发现，这里方法名称的定义要比 Iterator 中的方法名称更长。

要想使用此接口可以通过 Vector 类，Vector 类定义了以下的方法可以为 Enumeration 实例化。

```
public Enumeration<E> elements()
```

范例：使用 Enumeration 输出

```
package org.lxh.demo13.iteratordemo;
import java.util.Enumeration;
import java.util.Vector;
public class EnumerationDemo01 {
    public static void main(String[] args) {
        Vector<String> all = new Vector<String>();      // 只能使用Vector
        all.add("hello");                               // 增加元素
        all.add("_");                                   // 增加元素
        all.add("world");                               // 增加元素
        Enumeration<String> enu = all.elements();       // 实例化
                                                           Enumeration
        while (enu.hasMoreElements()) {                 // 循环输出
            System.out.print(enu.nextElement() + "、");// 输出元素
        }
    }
}
```

程序运行结果：

```
hello、_、world、
```

以上程序是使用 Vector 类的 elements()方法取得一个 Enumeration 接口的实例，之后使用 hasMoreElements()方法迭代判断并通过 nextElement()方法取出每一个元素，这一点与

Iterator 的操作非常类似。

提问：为什么还要继续使用 Enumeration？

Enumeration 和 Iterator 接口的功能非常类似，而且 Enumeration 接口中方法的名称也比 Iterator 接口中的方法名称长很多，那为什么还要继续使用 Enumeration 呢？

回答：在旧的操作中依然会使用 Enumeration 接口。

实际上 Java 的发展经历了很长的时间，一些比较古老的系统或是类库的方法中（例如，本系列的下一步 Web 开发中就存在这样的操作方法）还在使用 Enumeration 接口，所以掌握其操作也是很有必要的。

13.7　Map 接口

13.7.1　Map 接口简介

之前所讲解的 Collection、Set、List 接口都属于单值的操作，即每次只能操作一个对象，而 Map 与它们不同的是，每次操作的是一对对象，即二元偶对象，Map 中的每个元素都使用 key→value 的形式存储在集合中。Map 接口的定义如下：

```
public interface Map<K,V>
```

在 Map 上也应用了泛型，必须同时设置好 key 或 value 的类型，在 Map 中每一对 key→value 都表示一个值。同样，在 Map 接口中也定义了大量的方法，这些方法如表 13-11 所示。

表 13-11　Map 接口中的方法

序号	方　　法	类型	描　　述
1	public void clear()	普通	清空 Map 集合
2	public boolean containsKey(Object key)	普通	判断指定的 key 是否存在
3	public boolean containsValue(Object value)	普通	判断指定的 value 是否存在
4	public Set<Map.Entry<K,V>> entrySet()	普通	将 Map 对象变为 Set 集合
5	public boolean equals(Object o)	普通	对象比较
6	public V get(Object key)	普通	根据 key 取得 value
7	public int hashCode()	普通	返回哈希码
8	public boolean isEmpty()	普通	判断集合是否为空
9	public Set<K> keySet()	普通	取得所有的 key
10	public V put(K key, V value)	普通	向集合中加入元素
11	public void putAll(Map<? extends K,? extends V> t)	普通	将一个 Map 集合中的内容加入到另一个 Map
12	public V remove(Object key)	普通	根据 key 删除 value
13	public int size()	普通	取出集合的长度
14	public Collection<V> values()	普通	取出全部的 value

13.7.2 Map.Entry 接口简介

Map.Entry 是 Map 内部定义的一个接口，专门用来保存 key→value 的内容。Map.Entry 的定义如下：

```
public static interface Map.Entry<K,V>
```

Map.Entry 是使用 static 关键字声明的内部接口，此接口可以由外部通过“外部类.内部类”的形式直接调用。在本接口中提供了如表 13-12 所示的方法。

表 13-12 Map.Entry 接口的常用方法

序号	方　法	类型	描　述
1	public boolean equals(Object o)	普通	对象比较
2	public K getKey()	普通	取得 key
3	public V getValue()	普通	取得 value
4	public int hashCode()	普通	返回哈希码
5	public V setValue(V value)	普通	设置 value 的值

从之前的内容可以知道，在 Map 的操作中，所有的内容都是通过 key→value 的形式保存数据的，那么对于集合来讲，实际上是将 key→value 的数据保存在了 Map.Entry 的实例之后，再在 Map 集合中插入的是一个 Map.Entry 的实例化对象，如图 13-4 所示。

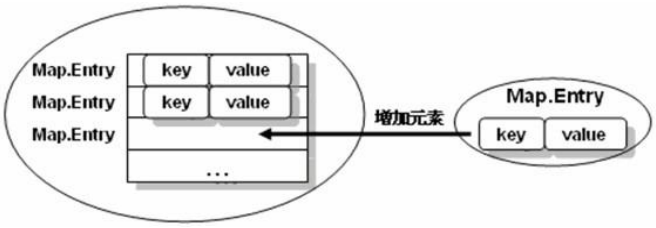


图 13-4 Map 与 Map.Entry

提示：Map.Entry 在集合输出时会使用到。

在一般的 Map 操作中（例如，增加或取出数据等操作）不用去管 Map.Entry 接口，但是在将 Map 中的数据全部输出时就必须使用 Map.Entry 接口，此内容随后会为读者介绍。

13.7.3 Map 接口的常用子类

与之前的 Collection 类似，如果要想使用 Map 接口也必须依靠其子类实例化。Map 接口中常用的子类介绍如下。

- HashMap：无序存放的，是新的操作类，key 不允许重复。
- Hashtable：无序存放的，是旧的操作类，key 不允许重复。
- TreeMap：可以排序的 Map 集合，按集合中的 key 排序，key 不允许重复。
- WeakHashMap：弱引用的 Map 集合，当集合中的某些内容不再使用时清除掉无用的数据，使用 gc 进行回收。

➘ IdentityHashMap：key 可以重复的 Map 集合。

1．新的子类：HashMap

HashMap 本身是 Map 的子类，直接使用此类为 Map 接口实例化即可。
HashMap 类的定义如下：

```
public class HashMap<K,V>
extends AbstractMap<K,V>
implements Map<K,V>, Cloneable, Serializable
```

HashMap 是 AbstractMap 类的子类。AbstractMap 类的定义如下：

```
public abstract class AbstractMap<K,V>
extends Object
implements Map<K,V>
```

2．相关操作实例

因为所有的操作都是以 Map 接口为操作的标准，且 HashMap 子类较为常用，所以下面的程序使用 HashMap 来为读者讲解 Map 接口中的主要操作方法。

（1）实例操作一：向集合中增加和取出内容

在 Map 接口中使用 put(Object key,Object value)方法可以向集合中增加内容，之后可以通过 get(E key)方法根据 key 找出其对应的 value。

范例：增加和取得内容

```
package org.lxh.demo13.mapdemo;
import java.util.HashMap;
import java.util.Map;
public class HashMapDemo01 {
    public static void main(String[] args) {
        Map<String, String> map = null;                          // 声明Map对象
        map = new HashMap<String, String>();                     // key和value是
                                                                    String类
        map.put("mldn", "www.mldn.cn");                          // 增加内容
        map.put("zhinangtuan", "www.zhinangtuan.net.cn");        // 增加内容
        map.put("mldnjava", "www.mldnjava.com.cn");              // 增加内容
        String val = map.get("mldn");                            // 根据key求出
                                                                    value
        System.out.println("取出的内容是:" + val);                 // 输出Map，调用
                                                                    toString()
    }
}
```

程序运行结果：

```
www.mldn.cn
```

在程序中声明Map接口对象时指定好了key和value的泛型类型为String，之后通过put()方法向Map集合中增加内容，最后通过get()方法取出一个key对应的value内容。

（2）实例操作二：判断指定的key或value是否存在

如果要判断某一个指定的key或value是否存在，可以使用Map接口中提供的containsKey(Object key)和containsValue(Object value)两个方法，前者是判断Map集合是否存在指定的key，后者是判断Map集合是否存在指定的value。

范例：判断指定内容是否存在

```
package org.lxh.demo13.mapdemo;
import java.util.HashMap;
import java.util.Map;
public class HashMapDemo02 {
    public static void main(String[] args) {
        Map<String, String> map = null;                          // 声明Map对象
        map = new HashMap<String, String>();                     // key和value是
                                                                    String类
        map.put("mldn", "www.mldn.cn");                          // 增加内容
        map.put("zhinangtuan", "www.zhinangtuan.net.cn");        // 增加内容
        map.put("mldnjava", "www.mldnjava.com.cn");              // 增加内容
        if (map.containsKey("mldn")) {                           // 查找指定的key
                                                                    是否存在
            System.out.println("搜索的key存在！");
        } else {
            System.out.println("搜索的key不存在！");
        }
        if(map.containsValue("www.mldn.cn")){                    // 查找指定的
                                                                    value是否存在
            System.out.println("搜索的value存在！");
        }else{
            System.out.println("搜索的value不存在!");
        }
    }
}
```

程序运行结果：

```
搜索的key存在!
搜索的value存在!
```

（3）实例操作三：输出全部的key

在Map中提供了一个叫做keySet()的方法，可以将一个Map中的全部key变为一个Set集合，一旦有了Set实例，就可以直接使用Iterator输出。但是在进行操作时一定要注意的是，接收的Set集合中指定的泛型要和Map中key的泛型类型保持一致。

范例：输出全部的 key

```
package org.lxh.demo13.mapdemo;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
public class HashMapDemo03 {
    public static void main(String[] args) {
        Map<String, String> map = null;                          // 声明Map对象
        map = new HashMap<String, String>();                     // key和value是
                                                                    String类
        map.put("mldn", "www.mldn.cn");                          // 增加内容
        map.put("zhinangtuan", "www.zhinangtuan.net.cn");        // 增加内容
        map.put("mldnjava", "www.mldnjava.com.cn");              // 增加内容
        Set<String> keys = map.keySet();                         // 得到全部的key
        Iterator<String> iter = keys.iterator();                 // 实例化
                                                                    Iterator
        System.out.print("全部的key: ");                         // 输出信息
        while (iter.hasNext()) {                                 // 迭代输出全部
                                                                    的key
            String str = iter.next();                            // 取出集合的key
            System.out.print(str + "、");                        // 输出内容
        }
    }
}
```

程序运行结果：

```
全部的key: mldn、zhinangtuan、mldnjava、
```

（4）实例操作四：输出全部的 value

如果要输出全部的 value，则使用 values()方法，此方法的返回类型是 Collection。在进行操作时也同样需要注意泛型的类型。

范例：输出全部的 value

```
package org.lxh.demo13.mapdemo;
import java.util.Collection;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
public class HashMapDemo04 {
    public static void main(String[] args) {
        Map<String, String> map = null;                  // 声明Map对象
        map = new HashMap<String, String>();             // key和value是String类
```

```
        map.put("mldn", "www.mldn.cn");                          // 增加内容
        map.put("zhinangtuan", "www.zhinangtuan.net.cn");        // 增加内容
        map.put("mldnjava", "www.mldnjava.com.cn");              // 增加内容
        Collection<String> values = map.values();                // 得到全部的
                                                                    value
        Iterator<String> iter = values.iterator();               // 实例化
                                                                    Iterator
        System.out.print("全部的value: ");                        // 输出信息
        while (iter.hasNext()) {                                 // 迭代输出
            String str = iter.next();                            // 取出value
            System.out.print(str + "、");                        // 输出内容
        }
    }
}
```

程序运行结果：

```
全部的value: www.mldn.cn、www.sina.com.cn、www.zhinangtuan.net.cn、
```

3. 旧的子类：Hashtable

Hashtable 也是 Map 中的一个子类，与 Vector 类的推出时间一样，都属于旧的操作类，其使用上也和之前没有什么太大的区别。

范例： Hashtable 操作

```
package org.lxh.demo13.mapdemo;
import java.util.Collection;
import java.util.Hashtable;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
public class HashtableDemo01 {
    public static void main(String[] args) {
        Map<String, String> map = null;                          // 声明Map对象
        map = new Hashtable<String, String>();                   // key和value是
                                                                    String类
        map.put("mldn", "www.mldn.cn");                          // 增加内容
        map.put("zhinangtuan", "www.zhinangtuan.net.cn");        // 增加内容
        map.put("mldnjava", "www.mldnjava.com.cn");              // 增加内容
        Set<String> keys = map.keySet();                         // 得到全部的key
        Iterator<String> iter1 = keys.iterator();                // 实例化
                                                                    Iterator
        System.out.print("全部的key: ");                         // 输出信息
        while (iter1.hasNext()) {                                // 迭代输出全部
                                                                    的key
```

```
            String str = iter1.next();                  // 取出内容
            System.out.print(str + "、");                // 输出内容
        }
        Collection<String> values = map.values();       // 得到全部的value
        Iterator<String> iter2 = values.iterator();// 实例化Iterator
        System.out.print("\n全部的value：");              // 输出信息
        while (iter2.hasNext()) {                       // 迭代输出全部的value
            String str = iter2.next();                  // 取出内容
            System.out.print(str + "、");                // 输出内容
        }
    }
}
```

程序运行结果：

```
全部的key：mldn、mldnjava、zhinangtuan、
全部的value：www.mldn.cn、www.mldnjava.com.cn、www.zhinangtuan.net.cn、
```

4．HashMap 与 Hashtable 的区别

在之前的程序中，HashMap 和 Hashtable 在操作上没有什么特别大的区别，那么到底该使用哪一个呢？这两个类的比较如表 13-13 所示。

表 13-13　HashMap 与 Hashtable 的比较

序号	比较点	HashMap	Hashtable
1	推出时间	JDK 1.2 之后推出的，属于新的操作类	JDK 1.0 时推出，属于旧的操作类
2	性能	采用异步处理方式，性能更高	采用同步处理方式，性能较低
3	线程安全	属于非线程安全的操作类	属于线程安全的操作类

以上对两个类进行了简单的对比，但是从实际的开发应用来看，HashMap 类使用较多，读者应该重点掌握。

5．排序的子类：TreeMap

之前的两个类在存放数据时并没有对其进行排序，细心的读者可以发现之前输出全部 key 时是无序的，而 TreeMap 的主要功能是按 key 排序。

范例：观察 TreeMap 排序

```
package org.lxh.demo13.mapdemo;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
import java.util.TreeMap;
public class TreeMapDemo01 {
    public static void main(String[] args) {
        Map<String, String> map = null;
```

```
            map = new TreeMap<String, String>();                    // 实例化Map对象
            map.put("A、mldn", "www.mldn.cn");                      // 增加内容
            map.put("C、zhinangtuan", "www.zhinangtuan.net.cn");// 增加内容
            map.put("B、mldnjava", "www.mldnjava.cn");              // 增加内容
            Set<String> keys = map.keySet();                        // 得到全部的key
            Iterator<String> iter = keys.iterator();                // 实例化Iterator
            while (iter.hasNext()) {                                // 迭代输出
                String str = iter.next();                           // 取出key
                System.out.println(str + " --> " + map.get(str));// 取出key对应
                                                                        的内容
            }
        }
    }
```

程序运行结果：

```
A、mldn --> www.mldn.cn
B、mldnjava --> www.mldnjava.cn
C、zhinangtuan --> www.zhinangtuan.net.cn
```

从程序的运行结果中可以发现，最终保存在 Map 中的数据是经过排序的数据，按其 key 排序。

注意：使用自定义类作为 key 时类需要实现 Comparable 接口。

TreeMap 可以按照 key 排序，之前的代码使用的是 String 类作为 key，因为 String 类本身已经实现了 Comparable 接口，所以程序执行时不会有任何的问题；而如果使用一个自定义的类作为 key，则此类必须实现 Comparable 接口，否则将出现类转换异常。

6．弱引用类：WeakHashMap

之前所讲解的 Map 子类中的数据都是使用强引用保存的，即里面的内容不管是否使用都始终在集合中保留，如果希望集合自动清理暂时不用的数据就使用 WeakHashMap 类。这样，当进行垃圾收集时会释放掉集合中的垃圾信息，WeakHashMap 的定义如下：

```
public class WeakHashMap<K,V>
extends AbstractMap<K,V>
implements Map<K,V>
```

范例：观察弱引用的 Map 集合

```
package org.lxh.demo13.mapdemo;
import java.util.Map;
import java.util.WeakHashMap;
public class WeakHashMapDemo01 {
    public static void main(String[] args) {
        Map<String, String> map = null;
        map = new WeakHashMap<String, String>();        // 实例化Map对象
```

```
        map.put(new String("mldn"), new String("www.mldn.cn"));
        map.put(new String("zhinangtuan"), new String("www.zhinangtuan.
        net.cn"));
        map.put(new String("mldnjava"), new String("www.mldnjava.cn"));

        System.gc();                                    // 进行垃圾收集
        map.put(new String("lxh"), new String("lixinghua")) ;
        System.out.println("内容: " + map);             // 一般只会剩下一个内容
    }
}
```

程序运行结果：

```
内容: {lxh=lixinghua}
```

从程序的运行结果中可以发现，因为使用了弱引用集合，所以对于 Map 集合中暂时不用的数据会被清除掉。

java.lang.ref

提示：对象的引用强度说明。

从 JDK1.2 版本开始，Java 把对象的引用分为 4 种级别，从而使程序能更加灵活地控制对象的生命周期。这 4 种级别由高到低依次为：强引用、软引用、弱引用和虚引用，下面简单介绍这 4 种引用的区别。

- 强引用：当内存不足时，JVM 宁可出现 OutOfMemeryError 错误而使程序停止，也不会回收此对象来释放空间。
- 软引用：当内存不足时，会回收这些对象的内存，用来实现内存敏感的高速缓存。
- 弱引用：无论内存是否紧张，被垃圾回收器发现立即回收。
- 虚引用：和没有任何引用一样。

13.7.4　Map 接口的使用注意事项

1. 注意事项一：不能直接使用迭代输出 Map 中的全部内容

对于 Map 接口来说，其本身是不能直接使用迭代（如 Iterator、foreach）进行输出的，因为 Map 中的每个位置存放的是一对值（key→value），而 Iterator 中每次只能找到一个值。所以，如果非要使用迭代进行输出，则必须按照以下步骤完成（以 Iterator 输出方法为例）。

（1）将 Map 的实例通过 entrySet()方法变为 Set 接口对象。
（2）通过 Set 接口实例为 Iterator 实例化。
（3）通过 Iterator 迭代输出，每个内容都是 Map.Entry 的对象。
（4）通过 Map.Entry 进行 key→value 的分离。

在前面提过，Map 中的每对数据都是通过 Map.Entry 保存的，所以如果最终要进行输出也应该使用 Map.Entry 完成。下面通过 Iterator 和 foreach 来为读者演示 Map 的输出。

提示：Map 一般很少直接输出，只是作为查询使用。

Map 集合的内容在开发中基本上作为查询的应用较多，全部输出的操作较少。而 Collection 接口在开发中的主要作用就是用来传递内容及输出的。

Map 输出方式一：Iterator 输出 Map

范例：使用 Iterator 输出 Map 实例

```
package org.lxh.demo13.iteratordemo;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
public class IteratorDemo04 {
    public static void main(String[] args) {
        Map<String, String> map = null;                          // 声明Map对象，指定泛型类型
        map = new HashMap<String, String>();                     // 实例化Map对象
        map.put("mldn", "www.mldn.cn");                          // 增加内容
        map.put("zhinangtuan", "www.zhinangtuan.net.cn");        // 增加内容
        map.put("mldnjava", "www.mldnjava.cn");                  // 增加内容
        Set<Map.Entry<String, String>> allSet = null;            // 声明一个Set集合，指定泛型
        allSet = map.entrySet();                                 // 将Map接口实例变为Set接口实例
        Iterator<Map.Entry<String, String>> iter = null;         // 声明Iterator对象
        iter = allSet.iterator();                                // 实例化Iterator对象
        while (iter.hasNext()) {
            Map.Entry<String,String> me = iter.next();           // 找到Map.Entry实例
            System.out.println(me.getKey()
                    + " --> " + me.getValue());                  // 输出key和value
        }
    }
}
```

程序运行结果：

```
mldn --> www.mldn.cn
zhinangtuan --> www.zhinangtuan.net.cn
mldnjava --> www.mldnjava.cn
```

以上的操作流程是 Map 输出最标准的操作流程，读者一定要牢牢掌握，这一点在以后的开发中非常重要。

Map 输出方式二：foreach 输出 Map

既然可以使用 Iterator 接口进行输出，那么也就可以使用 foreach 进行输出，输出时还是要将 Map 集合变为 Set 集合，Set 集合中的每一个元素都是 Map.Entry 对象。

范例：使用 foreach 输出内容

```
package org.lxh.demo13.iteratordemo;
import java.util.HashMap;
import java.util.Map;
public class ForeachDemo02 {
    public static void main(String[] args) {
        Map<String, String> map = null;          // 声明Map对象，指定泛型类型
        map = new HashMap<String, String>();     // 实例化Map对象
        map.put("mldn", "www.mldn.cn");          // 增加内容
        map.put("zhinangtuan", "www.zhinangtuan.net.cn");    // 增加内容
        map.put("mldnjava", "www.mldnjava.cn");              // 增加内容
        for (Map.Entry<String, String> me : map.entrySet()) {// 输出Set集合
            System.out.println(me.getKey()
                    + " --> " + me.getValue());              // 输出key和
                                                                value
        }
    }
}
```

程序运行结果：

```
mldn --> www.mldn.cn
zhinangtuan --> www.zhinangtuan.net.cn
mldnjava --> www.mldnjava.cn
```

以上程序利用 foreach 语法将集合中的每个元素用 Map.Entry 类型的对象进行接收，之后再进行 key 与 value 的分离。

2. 注意事项二：直接使用非系统类作为 key

既然在 Map 接口中所有的内容都是以 key→valued 的形式出现的，而且在声明 Map 对象时，也是使用了泛型声明类型，那么现在就应该可以使用一个“字符串”表示一个 Person 对象。

范例：String→Person 映射

```
package org.lxh.demo13.mapdemo;
import java.util.HashMap;
import java.util.Map;
class Person {                                          // 定义Person类
    private String name;                                // 定义name属性
    private int age;                                    // 定义age属性
    public Person(String name, int age) {               // 通过构造方法为属性赋值
```

```
        this.name = name;                                    // 为name属性赋值
        this.age = age;                                      // 为age属性赋值
    }
    public String toString() {                               // 覆写toString()方法
        return "姓名：" + this.name + "；年龄：" + this.age;
    }
}
public class HashMapDemo05 {
    public static void main(String[] args) {
        Map<String, Person> map = null;                      // 声明Map对象，指定泛型
                                                                类型
        map = new HashMap<String, Person>();                 // 实例化Map对象
        map.put("zhangsan", new Person("张三", 30));// 增加内容
        System.out.println(map.get("zhangsan"));             // 查找内容
    }
}
```

程序运行结果：

```
姓名：张三；年龄：30
```

以上程序在操作时是以一个字符串匿名对象的方式查找的，从结果来看，是能够查找到其对应内容的。那么，如果现在两个类型换一下位置呢？即使用一个 Person 对象来表示一个字符串。

范例： Person→String 映射

```
package org.lxh.demo13.mapdemo;
import java.util.HashMap;
import java.util.Map;
class Person {                                           // 定义Person类
    private String name;                                 // 定义name属性
    private int age;                                     // 定义age属性
    public Person(String name, int age) {                // 通过构造方法为属性赋值
        this.name = name;                                // 为name属性赋值
        this.age = age;                                  // 为age属性赋值
    }
    public String toString() {                           // 覆写toString()方法
        return "姓名：" + this.name + "；年龄：" + this.age;
    }
}
public class HashMapDemo06 {
    public static void main(String[] args) {
        Map<Person, String> map = null;                  // 声明Map对象，指定泛型类型
        map = new HashMap<Person, String>();             // 实例化Map对象
```

```
        map.put(new Person("张三", 30), "zhangsan");          // 增加内容
        System.out.println(map.get(new Person("张三", 30)));// 查找内容
    }
}
```

程序运行结果：

```
null
```

以上程序的结构和之前差不多，只是使用了一个 Person 的匿名对象作为 key，将字符串的匿名对象作为了 value，唯一不同的只是改变了之前的 key 和 value 的位置，但是此时却无法查找到指定的内容，那为什么一开始使用 String 可以，而现在使用 Person 对象不可以。为了研究出这个问题，继续观察以下的程序代码。

范例：声明 Person 对象→String 映射

```
package org.lxh.demo13.mapdemo;
import java.util.HashMap;
import java.util.Map;
class Person {                                          // 定义Person类
    private String name;                                // 定义name属性
    private int age;                                    // 定义age属性
    public Person(String name, int age) {               // 通过构造方法为属性赋值
        this.name = name;                               // 为name属性赋值
        this.age = age;                                 // 为age属性赋值
    }
    public String toString() {                          // 覆写toString()方法
        return "姓名：" + this.name + "；年龄：" + this.age;
    }
}
public class HashMapDemo07 {
    public static void main(String[] args) {
        Map<Person, String> map = null;                 // 声明Map对象，指定泛型类型
        map = new HashMap<Person, String>();            // 实例化Map对象
        Person per = new Person("张三", 30) ;           // 实例化Person对象
        map.put(per, "zhangsan");                       // 增加内容
        System.out.println(map.get(per));               // 查找内容
    }
}
```

程序运行结果：

```
zhangsan
```

可以发现，根据 key 能够找到对应的 value，但是为什么此时可以找到，而之前不可以找到呢？细心的读者可能已经发现，之前操作时都是以 Person 的匿名对象完成的，但是现在进行查找操作时使用的并不是匿名对象，这样程序中设置和取得时都是使用了 Person 的

实例化对象，地址没有变化，所以可以找到。

但是所有的程序不可能这样去完成查找功能，用户在真正操作时是不可能明确地知道其中的引用地址，都应该像 String 那样可以通过匿名对象找到对应的 value，要想实现这样的功能，就得靠 Object 类中 hashCode()和 equals()的帮助，以方便区分是否是同一个对象。

范例：覆写 equals()和 hashCode()方法

```
package org.lxh.demo13.mapdemo;
import java.util.HashMap;
import java.util.Map;
class Person {                                          // 定义Person类
    private String name;                                // 定义name属性
    private int age;                                    // 定义age属性
    public Person(String name, int age) {               // 通过构造方法为属性赋值
        this.name = name;                               // 为name属性赋值
        this.age = age;                                 // 为age属性赋值
    }
    public boolean equals(Object obj) {                 // 覆写equals()方法
        if (this == obj) {                              // 判断地址是否相等
            return true;                                // 返回true表示同一对象
        }
        if (!(obj instanceof Person)) {                 // 传递进来的不是本类的对象
            return false;                               // 返回false表示不是同一对象
        }
        Person p = (Person) obj;                        // 进行向下转型
        if (this.name.equals(p.name) && this.age == p.age) {
            return true ;                               // 属性依次比较，相等返回true
        }else{
            return false ;                              // 属性内容不相等，返回false
        }
    }
    public int hashCode(){                              // 覆写hashCode()方法
        return this.name.hashCode() * this.age ;// 计算公式
    }
    public String toString() {                          // 覆写toString()方法
        return "姓名: " + this.name + "; 年龄: " + this.age; // 返回信息
    }
}
public class HashMapDemo08 {
    public static void main(String[] args) {
        Map<Person, String> map = null;                 // 实例化Map对象，指定泛型类型
        map = new HashMap<Person, String>();            // 实例化Map对象
        Person per = new Person("张三", 30) ;           // 实例化Person对象
```

```
            map.put(per, "zhangsan");                    // 增加内容
            System.out.println(map.get(per));            // 查找内容
        }
    }
```

程序运行结果：

```
zhangsan
```

从程序的运行结果中可以发现，如果要使用一个自定义的对象表示 Map 中的 key，则对象所在的类中一定要覆写 equals()和 hashCode()方法；否则无法找到对应的 value。

13.7.5 key 可以重复的 Map 集合：IdentityHashMap

之前所讲解的所有 Map 操作中 key 的值是不能重复的，例如，HashMap 操作时 key 是不能重复的，如果重复则肯定会覆盖之前的内容，如下代码所示。

范例： Map 中的 key 不允许重复，重复就是覆盖

```
package org.lxh.demo13.mapdemo;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
class Person {                                   // 定义Person类
    private String name;                         // 定义name属性
    private int age;                             // 定义age属性
    public Person(String name, int age) {        // 通过构造方法为属性赋值
        this.name = name;                        // 为name属性赋值
        this.age = age;                          // 为age属性赋值
    }
    public boolean equals(Object obj) {          // 覆写equals()方法
        if (this == obj) {                       // 判断地址是否相等
            return true;                         // 返回true表示同一对象
        }
        if (!(obj instanceof Person)) {          // 传递进来的不是本类的对象
            return false;                        // 返回false表示不是同一对象
        }
        Person p = (Person) obj;                 // 进行向下转型
        if (this.name.equals(p.name) && this.age == p.age) {
            return true ;                        // 属性依次比较，相等返回true
        }else{
            return false ;                       // 属性内容不相等，返回false
        }
    }
    public int hashCode(){                       // 覆写hashCode()方法
```

```
        return this.name.hashCode() * this.age ;    // 计算公式
    }
    public String toString() {                      // 覆写toString()方法
        return "姓名：" + this.name + "；年龄：" + this.age; // 返回信息
    }
}
public class IdentityHashMapDemo01 {
    public static void main(String[] args) {
        Map<Person, String> map = null;                 // 声明Map对象，指定
                                                           泛型类型
        map = new HashMap<Person, String>();            // 实例化Map对象
        map.put(new Person("张三", 30), "zhangsan_1");  // 增加内容
        map.put(new Person("张三", 30), "zhangsan_2");  // 增加内容，key重复
        map.put(new Person("李四", 31), "lisi");        // 增加内容
        Set<Map.Entry<Person, String>> allSet = null;   // 声明一个Set集合
        allSet = map.entrySet();                        // 将Map接口实例变为
                                                           Set接口实例
        Iterator<Map.Entry<Person, String>> iter = null;   // 声明Iterator
                                                              对象
        iter = allSet.iterator();                       // 实例化Iterator
                                                           对象
        while (iter.hasNext()) {                        // 迭代输出
            Map.Entry<Person, String> me = iter.next();// 每个对象都是Map.
                                                          Entry实例
            System.out.println(me.getKey()
                    + " --> " + me.getValue());         // 输出key和value
        }
    }
}
```

程序运行结果：

```
姓名：李四；年龄：31 --> lisi
姓名：张三；年龄：30 --> zhangsan_2
```

从程序的运行结果中可以发现，第二个内容覆盖了第一个内容，所以此时可以使用 Identity HashMap。使用此类时只要地址不相等（key1！=key2），就表示不是重复的 key，可以添加到集合中。

范例：使用 IdentityHashMap 修改程序

```
package org.lxh.demo13.mapdemo;
import java.util.IdentityHashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
```

```
class Person {
    // 此类与之前定义一样，此处不再列出
}
public class IdentityHashMapDemo02 {
    public static void main(String[] args) {
        Map<Person, String> map = null;                    // 声明Map对象，指定
                                                              泛型类型
        map = new IdentityHashMap<Person, String>();       // 实例化Map对象
        map.put(new Person("张三", 30), "zhangsan_1");     // 增加内容
        map.put(new Person("张三", 30), "zhangsan_2");     // 增加内容，key重复
        map.put(new Person("李四", 31), "lisi");           // 增加内容
        Set<Map.Entry<Person, String>> allSet = null;      // 声明一个Set集合
        allSet = map.entrySet();                           // 将Map接口实例变为
                                                              Set接口实例
        Iterator<Map.Entry<Person, String>> iter = null;// 声明Iterator对象
        iter = allSet.iterator();                          // 实例化Iterator
                                                              对象
        while (iter.hasNext()) {                           // 迭代输出
            Map.Entry<Person, String> me = iter.next();// 每个对象都是Map.
                                                              Entry实例
            System.out.println(me.getKey()
                    + " --> " + me.getValue());            // 输出key和value
        }
    }
}
```

程序运行结果：

```
姓名：张三；年龄：30 --> zhangsan_2
姓名：张三；年龄：30 --> zhangsan_1
姓名：李四；年龄：31 --> lisi
```

从程序的运行结果中可以发现，现在的 key 允许重复，只要两个对象的地址不相等即可。

13.8　SortedMap 接口

SortedMap 接口是排序接口，只要是实现了此接口的子类，都属于排序的子类，TreeMap 也是此接口的一个子类。SortedMap 接口的定义如下：

```
public interface SortedMap<K,V>
extends Map<K,V>
```

之前讲解的 TreeMap 就是此接口的实现类，所以 TreeMap 可以完成排序的功能。在此接口上定义了一些 Map 中没有的方法，表 13-14 列出了部分的方法。

表 13-14 SortedMap 接口扩展的方法

序号	方 法	类型	描 述
1	public Comparator<? super K> comparator()	普通	返回比较器对象
2	public K firstKey()	普通	返回第一个元素的 key
3	public SortedMap<K,V> headMap(K toKey)	普通	返回小于等于指定 key 的部分集合
4	public K lastKey()	普通	返回最后一个元素的 key
5	public SortedMap<K,V> subMap(K fromKey,K toKey)	普通	返回指定 key 范围的集合
6	public SortedMap<K,V> tailMap(K fromKey)	普通	返回大于指定 key 的部分集合

下面通过一个范例让读者对以上的操作有所了解。

范例：SortedMap 演示

```
package org.lxh.demo13.mapdemo;
import java.util.Map;
import java.util.SortedMap;
import java.util.TreeMap;
public class SortedMapDemo {
    public static void main(String args[]){
        SortedMap<String, String> map = null;        // 声明SortedMap对象
        map = new TreeMap<String, String>();         // 实例化SortedMap对象
        map.put("D、jiangker", "http://www.jiangker.com/") ;    // 增加内容
        map.put("A、mldn", "www.mldn.cn");                      // 增加内容
        map.put("C、zhinangtuan", "www.zhinangtuan.net.cn");    // 增加内容
        map.put("B、mldnjava", "www.mldnjava.cn");              // 增加内容
        System.out.print("第一个元素的内容的key: " + map.firstKey()) ;
        System.out.println("; 对应的值: " + map.get(map.firstKey())) ;
        System.out.print("最后一个元素的内容的key: " + map.lastKey()) ;
        System.out.println("; 对应的值: " + map.get(map.lastKey())) ;
        System.out.println("返回小于指定范围的集合") ;
        for(Map.Entry<String,String> me : map.headMap("B、mldnjava").
        entrySet()){
            System.out.println("\t|- " + me.getKey() + " --> " + me.
            getValue());
        }
        System.out.println("返回大于指定范围的集合") ;
        for(Map.Entry<String,String> me : map.tailMap("B、mldnjava").
        entrySet()){
            System.out.println("\t|- " + me.getKey() + " --> " + me.
            getValue());
        }
        System.out.println("部分集合") ;
```

```
        for(Map.Entry<String,String> me : map.subMap("A、mldnjava","C、
zhinangtuan").entrySet()){
            System.out.println("\t|- " + me.getKey() + " --> " + me.
            GetValue());
        }
    }
}
```

程序运行结果：

```
第一个元素的内容的key：A、mldn；对应的值：www.mldn.cn
最后一个元素的内容的key：D、jiangker；对应的值：http://www.jiangker.com/
返回小于指定范围的集合
    |- A、mldn-->www.mldn.cn
返回大于指定范围的集合
    |- B、mldnjava-->www.mldnjava.cn
    |- C、zhinangtuan-->www.zhinangtuan.net.cn
    |- D、jiangker-->http://www.jiangker.com/
部分的集合
    |- B、mldnjava-->www.mldnjava.cn
```

以上规定了很多 Map 接口中没有的方法，但是如果要操作以上的方法，则对象所在的类必须实现 Comparable 接口。

13.9　集合工具类：Collections

13.9.1　Collections 简介

在集合的应用开发中，集合的若干接口和若干个子类是最常使用的，但是在 JDK 中提供了一种集合操作的工具类——Collections，可以直接通过此类方便地操作集合。

Collections 类的定义：

```
public class Collections extends Object
```



从定义上看，和集合没有什么太大的关系。Collections 类的常用方法及类型如表 13-15 所示。

表 13-15　Collections 类的常用方法及类型

序号	方　法	类型	描　述
1	public static final List EMPTY_LIST	常量	返回一个空的 List 集合
2	public static final Set EMPTY_SET	常量	返回空的 Set 集合
3	public static final Map EMPTY_MAP	常量	返回空的 Map 集合
4	public static <T> boolean addAll(Collection<? super T> c, T... a)	普通	为集合添加内容

续表

序号	方　法	类型	描　述
5	public static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll)	普通	找到最大的内容，按比较器排序
6	public static <T extends Object & Comparable<? super T>> T min(Collection<? extends T> coll)	普通	找到集合中的最小内容，按比较器排序
7	public static <T> boolean replaceAll(List<T> list, T oldVal,T newVal)	普通	用新的内容替换集合的指定内容
8	public static void reverse(List<?> list)	普通	集合反转
9	public static <T> int binarySearch(List<? extends Comparable<? super T>> list,T key)	普通	查找集合中的指定内容
10	public static final <T> List<T> emptyList()	普通	返回一个空的 List 集合
11	public static final <K,V> Map<K,V> emptyMap()	普通	返回一个空的 Map 集合
12	public static final <T> Set<T> emptySet()	普通	返回一个空的 Set 集合
13	public static <T extends Comparable<? super T>> void sort(List<T> list)	普通	集合排序操作，根据 Comparable 接口进行排序
14	public static void swap(List<?> list,int i,int j)	普通	交换指定位置的元素

13.9.2 Collections 操作实例

1. 实例操作一：返回不可变的集合

Collections 类中可以返回空的 List、Set、Map 集合，但是通过这种方式返回的对象是无法进行增加数据的，因为在这些操作中并没有实现 add()方法，如下所示。

范例：返回空的集合对象

```
package org.lxh.demo13.collectionsdemo;
import java.util.Collections;
import java.util.List;
import java.util.Set;
public class CollectionsDemo01 {
    public static void main(String[] args) {
        List<String> allList = Collections.emptyList() ;    // 返回不可变的空List集合
        Set<String> allSet = Collections.emptySet() ;        // 返回不可变的空List集合
        allList.add("Hello") ;                               // 错误，无法加入
    }
}
```

程序运行结果：

```
Exception in thread "main" java.lang.UnsupportedOperationException
    at java.util.AbstractList.add(Unknown Source)
```

```
        at java.util.AbstractList.add(Unknown Source)
        at
org.lxh.demo13.collectionsdemo.CollectionsDemo01.main(CollectionsDemo01.java
:11)
```

以上提示表示不支持的方法异常，因为没有找到 add()方法的实现。

2．实例操作二：为集合增加内容

使用 addAll()方法可以为一个集合增加内容。此方法可以接收可变参数，所以可以传递任意多的参数作为集合的内容。

范例：增加内容

```
package org.lxh.demo13.collectionsdemo;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;
import java.util.List;
public class CollectionsDemo02 {
    public static void main(String[] args) {
        List<String> all = new ArrayList<String>();          // 实例化List
        Collections.addAll(all, "MLDN", "LXH", "mldnjava"); // 增加内容
        Iterator<String> iter = all.iterator() ;             // 实例化
                                                                Iterator对象
        while (iter.hasNext()) {                             // 迭代输出
            System.out.print(iter.next() + "、");            // 输出内容
        }
    }
}
```

程序运行结果：

```
MLDN、LXH、mldnjava、
```

3．实例操作三：反转集合中的内容

直接使用 Collections 工具类中的 reverse()方法即可将集合类中的内容反转保存。

范例：反转内容

```
package org.lxh.demo13.collectionsdemo;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;
import java.util.List;
public class CollectionsDemo03 {
    public static void main(String[] args) {
        List<String> all = new ArrayList<String>();          // 实例化List
```

```
        Collections.addAll(all, "MLDN", "LXH", "mldnjava"); // 增加内容
        Collections.reverse(all) ;                          // 内容反转保存
        Iterator<String> iter = all.iterator() ;            // 实例化
                                                               Iterator对象
        while (iter.hasNext()) {                            // 迭代输出
            System.out.print(iter.next() + "、");           // 输出内容
        }
    }
}
```

程序运行结果：

```
mldnjava、LXH、MLDN、
```

4. 实例操作四：检索内容

直接通过 Collections 类中的 binarySearch()方法即可完成内容的检索，检索之后会返回内容的位置。

范例：检索内容

```
package org.lxh.demo13.collectionsdemo;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
public class CollectionsDemo04 {
    public static void main(String[] args) {
        List<String> all = new ArrayList<String>();          // 实例化List
        Collections.addAll(all, "MLDN", "LXH", "mldnjava");// 增加内容
        int point = Collections.binarySearch(all, "LXH") ; // 检索内容
        System.out.println("检索结果：" + point);            // 输出位置
        point = Collections.binarySearch(all, "LI") ;       // 检索内容，内容
                                                               不存在
        System.out.println("检索结果：" + point);            // 输出位置
    }
}
```

程序运行结果：

```
检索结果：1
检索结果：-1
```

5. 实例操作五：替换集合中的内容

Collections 类中也提供了 replaceAll()方法，可以替换一个集合中的指定内容。

范例：替换集合指定范围的内容

```
package org.lxh.demo13.collectionsdemo;
import java.util.ArrayList;
```

```
import java.util.Collections;
import java.util.Iterator;
import java.util.List;
public class CollectionsDemo05 {
    public static void main(String[] args) {
        List<String> all = new ArrayList<String>();          // 实例化List
        Collections.addAll(all, "MLDN", "LXH", "mldnjava");// 增加内容
        if(Collections.replaceAll(all, "LXH", "李兴华")){   // 替换内容
            System.out.println("内容替换成功！");             // 输出信息
        }
        System.out.print("替换之后的结果：") ;                // 输出信息
        Iterator<String> iter = all.iterator() ;            // 实例化
                                                               Iterator对象
        while (iter.hasNext()) {                            // 迭代输出
            System.out.print(iter.next() + "、");           // 输出内容
        }
    }
}
```

程序运行结果：

```
内容替换成功！
替换之后的结果：MLDN、李兴华、mldnjava、
```

6. 实例操作六：集合排序

可以通过 Collections 类中的 sort()方法对一个集合进行排序操作，但是要求集合中每个对象所在的类必须实现 Comparable 接口。

范例：集合排序

```
package org.lxh.demo13.collectionsdemo;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;
import java.util.List;
public class CollectionsDemo06 {
    public static void main(String[] args) {
        List<String> all = new ArrayList<String>();   // 实例化List
        Collections.addAll(all, "1、MLDN", "2、LXH", "X、mldnjava");
                                                      // 增加内容
        Collections.addAll(all, "B、www.mldn.cn");    // 增加内容
        Collections.addAll(all, "A、www.mldnjava.cn");// 增加内容
        System.out.print("排序之前的集合：");           // 输出信息
        Iterator<String> iter = all.iterator() ;      // 实例化Iterator对象
        while (iter.hasNext()) {                      // 迭代输出
```

```
            System.out.print(iter.next() + "、");   // 输出内容
        }
        Collections.sort(all) ;                     // 集合排序
        System.out.print("\n排序之后的集合：") ;      // 输出信息
        iter = all.iterator() ;                     // 实例化Iterator对象
        while (iter.hasNext()) {                    // 迭代输出
            System.out.print(iter.next() + "、");   // 输出内容
        }
    }
}
```

程序运行结果：

```
排序之前的集合：1、MLDN、2、LXH、X、mldnjava、B、www.mldn.cn、A、www.mldnjava.cn、
排序之后的集合：1、MLDN、2、LXH、A、www.mldnjava.cn、B、www.mldn.cn、X、mldnjava、
```

这里必须提醒读者的是，以上操作直接使用了 String 类完成，String 类本身已经实现好了 Comparable 接口。如果是一个自定义的类，则一定要实现 Comparable 接口，否则会出现类型转换异常。

7. 实例操作七：交换指定位置的内容

直接使用 swap()方法可以把集合中两个位置的内容进行交换。

范例：交换指定位置的元素

```
package org.lxh.demo13.collectionsdemo;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;
import java.util.List;
public class CollectionsDemo07 {
    public static void main(String[] args) {
        List<String> all = new ArrayList<String>();// 实例化List
        Collections.addAll(all, "1、MLDN", "2、LXH", "3、mldnjava");
                                                    // 增加内容
        System.out.print("交换之前的集合：") ;        // 输出信息
        Iterator<String> iter = all.iterator() ;    // 实例化Iterator对象
        while (iter.hasNext()) {                    // 迭代输出
            System.out.print(iter.next() + "、");   // 输出内容
        }
        Collections.swap(all,0,2) ;                 // 交换指定位置的内容
        System.out.print("\n交换之后的集合：") ;      // 输出信息
        iter = all.iterator() ;                     // 实例化Iterator对象
        while (iter.hasNext()) {                    // 迭代输出
            System.out.print(iter.next() + "、");   // 输出内容
        }
```

```
        }
    }
```

程序运行结果：

```
交换之前的集合：1、MLDN、2、LXH、3、mldnjava、
交换之后的集合：3、mldnjava、2、LXH、1、MLDN、
```

从程序的运行结果中可以发现，位置 0 和位置 2 的元素交换了。

13.10　其他集合类

13.10.1　Stack 类

栈是采用先进后出的数据存储方式，每一个栈都包含一个栈顶，每次出栈是将栈顶的数据取出，如图 13-5 和图 13-6 所示。

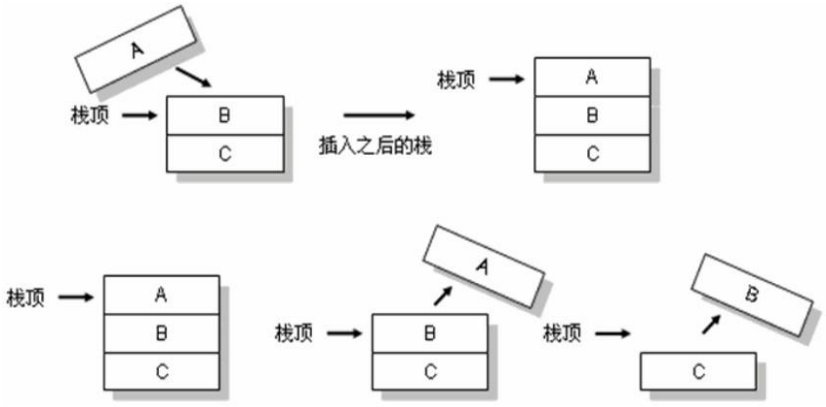


图 13-6　出栈操作

提示：栈的应用。

经常上网的读者应该清楚地知道，在浏览器中存在一个后退的按钮，每次后退都是后退到上一步的操作，那么实际上这就是一个栈的应用，采用的是一个先进后出的操作。

在 Java 中使用 Stack 类进行栈的操作，Stack 类是 Vector 的子类。Stack 类的定义如下：

```
public class Stack<E> extends Vector<E>
```

Stack 类的常用操作方法如表 13-16 所示。

表 13-16　Stack 类的常用方法

序号	方　　法	类型	描　　述
1	public boolean empty()	常量	测试栈是否为空
2	public E peek()	常量	查看栈顶，但不删除
3	public E pop()	常量	出栈，同时删除
4	public E push(E item)	普通	入栈
5	public int search(Object o)	普通	在栈中查找

范例：完成入栈及出栈程序

```
package org.lxh.demo13.stackdemo;
import java.util.Stack;
public class StackDemo {
    public static void main(String args[]) {
        Stack<String> s = new Stack<String>(); // 实例化Stack对象
        s.push("A");                           // 入栈
        s.push("B");                           // 入栈
        s.push("C");                           // 入栈
        System.out.print(s.pop() + "、") ;      // 出栈
        System.out.print(s.pop() + "、") ;      // 出栈
        System.out.println(s.pop() + "、") ;    // 出栈
        System.out.print(s.pop() + "、") ;      // 错误，出栈，出现异常，栈为空
    }
}
```

程序运行效果：

```
C、B、A、
Exception in thread "main" java.util.EmptyStackException
    at java.util.Stack.peek(Unknown Source)
    at java.util.Stack.pop(Unknown Source)
    at org.lxh.demo13.stackdemo.StackDemo.main(StackDemo.java:13)
```

从程序的运行结果来看，先进去的内容最后才取出，而且如果栈已经为空，则无法再弹出，会出现空栈异常。

13.10.2 属性类：Properties

1. Properties 类简介

在 Java 中属性操作类是一个较为重要的类。而要想明白属性操作类的作用，就必须先清楚什么叫属性文件，实际上在之前讲解国际化操作时就使用了属性文件（Message.properties），在一个属性文件中保存了多个属性，每一个属性就是直接用字符串表示出来的“key=value 对”，而如果要想轻松地操作这些属性文件中的属性，可以通过 Properties 类方便地完成。

提示：关于属性文件

对于属性文件其实在 Windows 操作系统的很多地方都可以见到。例如，Windows 的启动引导文件 boot.ini 就是使用属性文件的方式保存的，如下所示：

```
[boot loader]
timeout=5
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
```

可以发现其中都是通过“kye=value”形式保存的，那么这样的文件就是属性文件。

Properties 类本身是 Hashtable 类的子类，既然是其子类，则肯定也是按照 key 和 value 的形式存放数据的。Properties 类的定义如下：

```
public class Properties
extends Hashtable<Object,Object>
```

Properties 类中的很多方法都有实际用处，主要方法如表 13-17 所示。

表 13-17　Properties 类的主要方法

序号	方　　法	类型	描　　述
1	public Properties()	构造	构造一个空的属性类
2	public Properties(Properties defaults)	常量	构造一个指定属性内容的属性类
3	public String getProperty(String key)	常量	根据属性的 key 取得属性的 value，如果没有 key 则返回 null
4	public String getProperty(String key, String defaultValue)	普通	根据属性的 key 取得属性的 value，如果没有 key 则返回 defaultValue
5	public Object setProperty(String key, String value)	普通	设置属性
6	public void list(PrintStream out)	普通	属性打印
7	public void load(InputStream inStream) throws IOException	普通	从输入流中取出全部的属性内容
8	public void loadFromXML(InputStream in) throws IOException,InvalidPropertiesFormatException	普通	从 XML 文件格式中读取内容
9	public void store(OutputStream out,String comments) throws IOException	普通	将属性内容通过输出流输出，同时声明属性的注释
10	public void storeToXML(OutputStream os,String comment) throws IOException	普通	以 XML 文件格式输出属性，默认编码
11	public void storeToXML(OutputStream os,String comment,String encoding) throws IOException	普通	以 XML 文件格式输出属性，用户指定默认编码

虽然 Properties 类是 Hashtable 的子类，也可以像 Map 那样使用 put()方法保存任意类型的数据，但是一般属性都是由字符串组成的，所以在使用本类时本书只关心 Properties 类本身的方法，而从 Hashtable 接口继承下来的方法，本书将不作任何介绍。下面以一些实际操作向读者讲解 Properties 类中各种方法的使用。

提示：XML 文件格式说明。

XML（eXtensible Markup Language，可扩展的标记性语言）是在现在开发中使用最广泛的一门语言，所有的属性内容可以通过 storeToXML()和 loadFormXML()两个方法以 XML 文件格式进行保存和读取，但是使用 XML 格式保存时将按照指定的文档格式进行存放，如果格式出错，则将无法读取。

2．Properties 操作实例

（1）实例操作一：设置和取得属性

可以使用 setProperty()和 getProperty()方法设置和取得属性，操作时要以 String 为操作

类型。

范例：设置和取得属性

```
package org.lxh.demo13.propertiesdemo;
import java.util.Properties;
public class PropertiesDemo01 {
    public static void main(String[] args) {
        Properties pro = new Properties();                    // 创建Properties对象
        pro.setProperty("BJ", "BeiJing");                     // 设置内容
        pro.setProperty("TJ", "TianJin");                     // 设置内容
        pro.setProperty("NJ", "NanJing");                     // 设置内容
        System.out.println("1、BJ属性存在：" + pro.getProperty("BJ"));
        System.out.println("2、SC属性不存在：" + pro.getProperty("SC"));
        System.out.println("3、SC属性不存在，同时设置显示的默认值："
                + pro.getProperty("SC", "没有发现"));
    }
}
```

程序运行结果：

```
BJ属性存在：BeiJing
SC属性不存在：null
SC属性存在，同时设置默认值：没有发现
```

（2）实例操作二：将属性保存在普通文件中

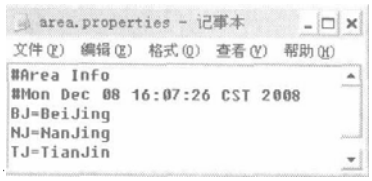
正常属性类操作完成之后，可以将其内容保存在文件中，那么直接使用 sote()方法即可，同时指定 OutputStream 类型，指明输出的位置。属性文件的后缀是任意的，但是最好按照标准，将属性文件的后缀统一设置成“*.properties”。

范例：保存属性到普通的属性文件中

```
package org.lxh.demo13.propertiesdemo;
import java.io.File;
import java.io.FileOutputStream;
import java.util.Properties;
public class PropertiesDemo02 {
    public static void main(String[] args) {
        Properties pro = new Properties();                    // 创建Properties对象
        pro.setProperty("BJ", "BeiJing");                     // 设置内容
        pro.setProperty("TJ", "TianJin");                     // 设置内容
        pro.setProperty("NJ", "NanJing");                     // 设置内容
        // 设置属性文件的保存路径
        File file = new File("D:" + File.separator + "area.properties");
        try {
            // 保存属性到普通文件中，并设置注释内容
            pro.store(new FileOutputStream(file), "Area Info");
```

```
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
```

程序运行结果：



可以发现在属性类中所保存的全部内容直接输出到了属性文件中。

（3）实例操作三：从普通文件中读取属性内容

既然可以保存，就可以通过 load()方法，从输入流中将所保存的所有属性内容读取出来。

范例：从属性文件中读取内容

```
package org.lxh.demo13.propertiesdemo;
import java.io.File;
import java.io.FileInputStream;
import java.util.Properties;
public class PropertiesDemo03 {
    public static void main(String[] args) {
        Properties pro = new Properties();          // 创建Properties对象
        // 设置属性文件的操作路径
        File file = new File("D:" + File.separator + "area.properties");
        try {
            pro.load(new FileInputStream(file));    // 读取属性文件
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.out.println("BJ属性值存在，内容是：" + pro.getProperty("BJ"));
    }
}
```

程序运行结果：

```
BJ属性值存在，内容是：BeiJing
```

（4）实例操作四：将属性保存在 XML 文件中

在 Properties 类中也可以把全部内容以 XML 格式的内容通过输出流输出，如果要把属性保存在 XML 文件中，则文件的后缀最好为“*.xml”。

范例：将属性保存在 XML 文件中

```
package org.lxh.demo13.propertiesdemo;
import java.io.File;
```

```
import java.io.FileOutputStream;
import java.util.Properties;
public class PropertiesDemo04 {
    public static void main(String[] args) {
        Properties pro = new Properties();          // 创建Properties对象
        pro.setProperty("BJ", "BeiJing");           // 设置内容
        pro.setProperty("TJ", "TianJin");           // 设置内容
        pro.setProperty("NJ", "NanJing");           // 设置内容
        // 设置属性文件的保存路径
        File file = new File("D:" + File.separator + "area.xml");
        try {
            pro.storeToXML(new FileOutputStream(file),
                "Area Info");                       // 保存属性到XML文件
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

程序运行结果：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
<comment>Area Info</comment>
<entry key="BJ">BeiJing</entry>
<entry key="NJ">NanJing</entry>
<entry key="TJ">TianJin</entry>
</properties>
```

（5）实例操作五：从 XML 文件中读取属性

以XML文件格式输出全部属性之后，必须要使用loadFromXML()方法将内容读取进来。

范例：读取文件内容

```
package org.lxh.demo13.propertiesdemo;
import java.io.File;
import java.io.FileInputStream;
import java.util.Properties;
public class PropertiesDemo05 {
    public static void main(String[] args) {
        Properties pro = new Properties();          // 创建Properties对象
        File file = new File("D:" + File.separator + "area.xml");
        try {
            pro.loadFromXML(new FileInputStream(file));// 读取xml文件
        } catch (Exception e) {
```

```
            e.printStackTrace();
        }
        System.out.println("BJ属性值存在，内容是：" + pro.getProperty("BJ"));
    }
}
```

程序运行结果：

```
BJ属性值存在，内容是：BeiJing
```

13.11　范　　例

下面将讲解两个基于类集应用的范例，这两个范例将作为日后 JAVA EE 的开发基础。

13.11.1　范例——一对多关系

使用类集可以表示出以下关系：一个学校可以包含多个学生，一个学生属于一个学校，那么这就是一个典型的一对多关系，此时就可以通过类集进行关系的表示。

范例：定义学生类

```
package org.lxh.demo13.execdemo01;
public class Student {                              // 定义Student类
    private String name ;                           // 定义name属性
    private int age ;                               // 定义age属性
    private School school ;                         // 一个学生属于一个学校
    public Student(String name,int age){            // 通过构造设置内容
        this.setName(name) ;                        // 设置name属性内容
        this.setAge(age) ;                          // 设置age属性内容
    }
    public String getName() {                       // 返回name属性内容
        return name;
    }
    public void setName(String name) {              // 设置name属性内容
        this.name = name;
    }
    public int getAge() {                           // 返回age属性内容
        return age;
    }
    public void setAge(int age) {                   // 设置age属性内容
        this.age = age;
    }
    public School getSchool() {                     // 返回所在学校
        return school;
    }
```

```
    public void setSchool(School school) {      // 设置所在学校
        this.school = school;
    }
    public String toString(){                   // 覆写toString()方法
        return "学生姓名：" + this.name + "；年龄：" + this.age ;
    }
}
```

在以上的 Student 类中包含了一个 School 属性，表示一个学生属于一个学校。在程序运行时，只需要传入 School 类的引用就可以完成这样的关系。

范例：定义学校类

```
package org.lxh.demo13.execdemo01;
import java.util.ArrayList;
import java.util.List;
public class School {                                   // 定义School类
    private String name ;                               // 定义name属性
    private List<Student> allStudents ;                 // 一个学校有多个学生
    public School(){                                    // 无参构造
        this.allStudents = new ArrayList<Student>() ;// 实例化List集合
    }
    public School(String name){                         // 通过构造设置name属性
        this() ;                                        // 调用无参构造
        this.setName(name) ;                            // 设置name属性内容
    }
    public String getName() {                           // 取得name属性内容
        return name;
    }
    public void setName(String name) {                  // 设置name属性内容
        this.name = name;
    }
    public List<Student> getAllStudents() {             // 取得全部的学生
        return allStudents;
    }
    public String toString(){                           // 覆写toString()方法
        return "学校名称：" + this.name ;
    }
}
```

在定义学校类时定义了一个 List 类型的属性，并指定其泛型类型是 Student 类型，这样一来就表示在一个 School 对象中会包含多个 Student 类型的引用。

范例：测试程序，并设置关系

```
package org.lxh.demo13.execdemo01;
import java.util.Iterator;
```

```
public class TestDemo {
    public static void main(String[] args) {
        School sch = new School("清华大学");                    // 实例化学校对象
        Student s1 = new Student("张三",21);                    // 实例化学生对象
        Student s2 = new Student("李四",22);                    // 实例化学生对象
        Student s3 = new Student("王五",23);                    // 实例化学生对象
        sch.getAllStudents().add(s1);                           // 在学校中加入学生
        sch.getAllStudents().add(s2);                           // 在学校中加入学生
        sch.getAllStudents().add(s3);                           // 在学校中加入学生
        s1.setSchool(sch);                                      // 一个学生属于一个学校
        s2.setSchool(sch);                                      // 一个学生属于一个学校
        s3.setSchool(sch);                                      // 一个学生属于一个学校
        System.out.println(sch);                                // 输出学校信息
        // 实例化Iterator对象，用于输出全部的学生信息
        Iterator<Student> iter = sch.getAllStudents().iterator();
        while (iter.hasNext()) {                                // 迭代输出
            System.out.println("\t|- " + iter.next());// 输出信息
        }
    }
}
```

程序运行结果：

```
学校名称：清华大学
    |- 学生姓名：张三；年龄：21
    |- 学生姓名：李四；年龄：22
    |- 学生姓名：王五；年龄：23
```

程序首先分别实例化了 School 及 Student 类的对象，之后通过两个类中的属性保存彼此的引用关系，从而形成了一个学校有多个学生，一个学生属于一个学校的一对多关系。

13.11.2　范例——多对多关系

使用类集不仅可以表示出一对一的关系，也可以表示出多对多的关系。例如，一个学生可以选多门课程，一门课程可以有多个学生参加，那么这就是一个典型的多对多关系。

要完成本程序，首先应该定义两个类：学生信息类 Student、课程信息类 Course。在一个学生类中存在一个集合，保存全部的课程；而在课程类中也要存在一个集合，保存全部的学生。

范例：定义学生类

```
package org.lxh.demo13.execdemo02;
import java.util.ArrayList;
import java.util.List;
public class Student {                                  // 定义学生类
```

```
    private String name ;                                    // 定义name属性
    private int age ;                                        // 定义age属性
    private List<Course> allCourses ;                        // 定义集合保存全部课程
    public Student(){                                        // 无参构造
        this.allCourses = new ArrayList<Course>() ;// 实例化List集合
    }
    public Student(String name,int age){                     // 通过构造设置属性内容
        this() ;                                             // 调用无参构造
        this.setName(name) ;                                 // 设置name属性内容
        this.setAge(age) ;                                   // 设置age属性内容
    }
    public String getName() {                                // 取得name属性
        return name;
    }
    public void setName(String name) {                       // 设置name属性
        this.name = name;
    }
    public int getAge() {                                    // 取得age属性
        return age;
    }
    public void setAge(int age) {                            // 设置age属性
        this.age = age;
    }
    public List<Course> getAllCourses() {                    // 取得全部课程
        return allCourses;
    }
    public String toString(){                                // 覆写toString()方法
        return "学生姓名：" + this.name + "；年龄：" + this.age ;
    }
}
```

在学生类中存在一个 allCourses 的 List 集合，这样在程序运行时，一个学生类中可以保存多个 Course 对象。

范例：定义课程类

```
package org.lxh.demo13.execdemo02;
import java.util.ArrayList;
import java.util.List;
public class Course {                                        // 定义Course类
    private String name ;                                    // 定义name属性
    private int credit ;                                     // 定义credit属性，表示学分
    private List<Student> allStudents ;                      // 定义集合保存多个学生
    public Course(){                                         // 无参构造方法
```

```
        this.allStudents = new ArrayList<Student>() ;  // 实例化List集合
    }
    public Course(String name,int credit){             // 设置name和credit
                                                          属性
        this() ;                                       // 调用无参构造
        this.setName(name) ;                           // 设置name属性
        this.setCredit(credit) ;                       // 设置credit属性
    }
    public String getName() {                          // 取得name属性
        return name;
    }
    public void setName(String name) {                 // 设置name属性
        this.name = name;
    }
    public int getCredit() {                           // 取得credit属性
        return credit;
    }
    public void setCredit(int credit) {                // 设置credit属性
        this.credit = credit;
    }
    public List<Student> getAllStudents() {            // 得到全部学生
        return allStudents;
    }
    public void setAllStudents(List<Student> allStudents) {// 设置全部学生
        this.allStudents = allStudents;
    }
    public String toString(){                          // 覆写toString()
                                                          方法
        return "课程名称：" + this.name + "；课程学分" + this.credit ;
    }
}
```

课程类与学生类一样，都定义了一个 List 集合，用于保存多个学生信息。

范例：测试程序

```
package org.lxh.demo13.execdemo02;
import java.util.Iterator;
public class TestMore {
    public static void main(String args[]) {
        Course c1 = new Course("英语", 3);             // 实例化课程对象
        Course c2 = new Course("计算机", 5);           // 实例化课程对象
        Student s1 = new Student("张三", 20);          // 实例化学生对象
        Student s2 = new Student("李四", 21);          // 实例化学生对象
        Student s3 = new Student("王五", 22);          // 实例化学生对象
```

```
        Student s4 = new Student("赵六", 23);          // 实例化学生对象
        Student s5 = new Student("孙七", 24);          // 实例化学生对象
        Student s6 = new Student("钱八", 25);          // 实例化学生对象
        // 第一门课程有3个人参加，向课程中增加3个学生信息，同时向学生中增加课程信息
        c1.getAllStudents().add(s1);                   // 向课程增加学生信息
        c1.getAllStudents().add(s2);                   // 向课程增加学生信息
        c1.getAllStudents().add(s6);                   // 向课程增加学生信息
        s1.getAllCourses().add(c1);                    // 向学生中增加课程信息
        s2.getAllCourses().add(c1);                    // 向学生中增加课程信息
        s6.getAllCourses().add(c1);                    // 向学生中增加课程信息
        // 第二门课程有6个人参加，向课程中增加6个学生信息，同时向学生中增加课程信息
        c2.getAllStudents().add(s1);                   // 向课程增加学生信息
        c2.getAllStudents().add(s2);                   // 向课程增加学生信息
        c2.getAllStudents().add(s3);                   // 向课程增加学生信息
        c2.getAllStudents().add(s4);                   // 向课程增加学生信息
        c2.getAllStudents().add(s5);                   // 向课程增加学生信息
        c2.getAllStudents().add(s6);                   // 向课程增加学生信息
        s1.getAllCourses().add(c2);                    // 向学生中增加课程信息
        s2.getAllCourses().add(c2);                    // 向学生中增加课程信息
        s3.getAllCourses().add(c2);                    // 向学生中增加课程信息
        s4.getAllCourses().add(c2);                    // 向学生中增加课程信息
        s5.getAllCourses().add(c2);                    // 向学生中增加课程信息
        s6.getAllCourses().add(c2);                    // 向学生中增加课程信息
        // 输出一门课程的信息，观察一门课程有多少个学生参加
        System.out.println(c1);                        // 输出第一门课程信息
        Iterator<Student> iter1 = c1.getAllStudents().iterator();
        while (iter1.hasNext()) {                      // 迭代输出
            Student s = iter1.next();                  // 取出学生对象
            System.out.println("\t|- " + s);           // 输出学生信息
        }
        // 输出一个学生参加的课程信息，观察有多少门课程
        System.out.println(s6);                        // 输出学生信息
        Iterator<Course> iter2 = s6.getAllCourses().iterator();
        while (iter2.hasNext()) {                      // 迭代输出
            Course c = iter2.next();                   // 取得所参加的课程
            System.out.println("\t|- " + c);           // 输出课程信息
        }
    }
}
```

程序运行结果：

```
课程名称：英语；课程学分3
    |- 学生姓名：张三；年龄：20
```

```
    |- 学生姓名：李四；年龄：21
    |- 学生姓名：钱八；年龄：25
学生姓名：钱八；年龄：25
    |- 课程名称：英语；课程学分3
    |- 课程名称：计算机；课程学分5
```

从程序来看，设置关系的地方较为复杂，因为现在的程序采用的是双向的处理关系，所以学生在选择一个课程时，除了课程中要添加学生以外，在学生中也要添加课程信息。在输出课程信息时，可以通过课程对象中的集合找到参加此课程的全部学生信息，也可以通过学生对象找到全部参加的课程信息。

13.12　本 章 要 点

1．类集的目的是用来创建动态的对象数组操作。

2．Collection 接口是类集中的最大单值操作的父接口，但是一般开发中不会直接使用此接口，而常使用 List 或 Set 接口。

3．List 接口扩展了 Collection 接口，里面的内容是允许重复的。

4．List 接口的常用子类是 ArrayList 和 Vector，在开发中 ArrayList 性能较高，属于异步处理，而 Vector 性能较低，属于同步处理。

5．Set 接口与 Collection 接口的定义一致，里面的内容是不允许重复的，依靠 Object 类中的 equals()和 hashCode()方法来区分是否是同一个对象。

6．Set 接口的常用子类是 HashSet 和 TreeSet，前者是散列存放，没有顺序；后者是顺序存放，使用 Comparable 进行排序操作。

7．集合的输出要使用 Iterator 接口完成，Iterator 属于迭代输出接口。

8．在 JDK 1.5 之后集合也可以使用 foreach 的方式输出。

9．Enumeration 属于最早的迭代输出接口，现在基本上很少使用，在类集中 Vector 类可以使用 Enumeration 接口进行内容的输出。

10．List 集合的操作可以使用 ListIterator 接口进行双向的输出操作。

11．Map 接口可以存放一对内容，所有的内容以 key→value 的形式保存，每一对 key→value 都是一个 Map.Entry 对象的实例。

12．Map 中的常用子类是 HashMap、TreeMap、Hashtable。HashMap 属于异步处理，性能较高；TreeMap 属于排序类，按照 Comparable 指定的顺序进行 key 的排序；Hashtable 属于同步处理，性能较低。

13．类集中提供了 Collections 工具类完成类集的相关操作。

14．Stack 类可以完成先进后出的操作。

15．Properties 类属于属性操作类，使用属性操作类可以直接操作属性文件，属性文件可以按普通文件或者是 XML 的文件格式进行保存。

16．使用类集可以方便地表示出一对多及多对多的关系。

13.13 习　　题

1．编写学生类，该类定义了 3 个属性：学号、姓名、成绩。可以通过构造方法设置 3 个属性的内容，并覆写 Object 类中的 toString()方法，在 List 集合中加入 5 个学生对象，并将内容输出，之后使用比较器将对象的内容进行排序并显示在屏幕上。

2．完成一个学生管理程序，使用学号作为键添加 5 个学生对象，并可以将全部的信息保存在文件中，可以实现对学生信息的学号查找、输出全部学生信息的功能。

3．自己动手编写一个双向链表。

4．编写程序通过栈的方式将任意输入的字符串内容进行逆序输出。

5．将本章中讲解的一对多关系加入菜单及文件的操作，可以实现一个学校-学生的管理程序。

第 14 章　枚　　举

通过本章的学习可以达到以下目标：

- ☑ 了解枚举可以解决哪些技术问题。
- ☑ 掌握枚举的定义方式。
- ☑ 清楚地掌握 enum 关键字与 Enum 类的关系。
- ☑ 掌握 JDK 1.5 之后的枚举操作类：EnumSet、EnumMap。
- ☑ 让枚举类实现一个接口或在枚举类中定义抽象方法。

枚举是在 JDK 1.5 之后另一个重要的特性，在枚举中可以限制一个类的对象产生范围，加入了枚举之后 Java 又对之前的类集进行了扩充，产生了一些新的枚举支持类——EnumSet、EnumMap，为枚举的使用提供方便，本章将对枚举的一些特点进行讲解。**本章视频录像讲解时间为 55 分钟，源代码在光盘对应的章节下。**

14.1　枚举类型简介

在 JDK 1.5 之前，Java 有两种方式定义新类型：类和接口。对于大部分面向对象编程来说，这两种方法看起来似乎足够了。但是在一些特殊情况下，这些方法就不适合。例如，想定义一个 Color 类，它只能有 Red、Green、Blue 3 种值，其他的任何值都是非法的，那么 JDK 1.5 之前虽然可以构造这样的代码，但是要做很多的工作，就可能带来各种不安全的问题。而 JDK 1.5 之后引入的枚举类型（Enum）就能避免这些问题。

14.2　使用简单程序完成枚举的功能

枚举是在 JDK 1.5 之后引入的，如果在 JDK 1.5 之前完成枚举的功能，则必须依靠类似于以下的程序代码完成。

范例：使用简单类完成枚举操作

```
package org.lxh.demo14.colordemo;
class Color {
    public static final Color RED = new Color("红色"); // 定义第一个对象
    public static final Color GREEN = new Color("绿色");// 定义第二个对象
    public static final Color BLUE = new Color("蓝色");// 定义第三个对象
    private String name;
    private Color(String name) {                        // 构造方法私有化，同
                                                           时设置颜色的名称
        this.setName(name);                             // 为颜色的名字赋值
```

```
    }
    public String getName() {                          // 取得颜色名称
        return this.name;
    }
    public void setName(String name) {                 // 设置颜色名称
        this.name = name;
    }
    public static Color getInstance(int i) {           // 得到一个颜色，只能从固定
                                                          的几个颜色中取得
        switch (i) {
            case 1: {                                  // 返回红色对象
                return RED;
            }
            case 2: {                                  // 返回绿色对象
                return GREEN;
            }
            case 3: {                                  // 返回蓝色对象
                return BLUE;
            }
            default: {
                return null;                           // 错误的值
            }
        }
    }
}
public class ColorDemo01 {
    public static void main(String[] args) {
        Color c1 = Color.RED ;                         // 取得红色
        System.out.println(c1.getName());              // 输出名字
        Color c2 = Color.getInstance(3);               // 根据编号取得名字
        System.out.println(c2.getName()) ;             // 输出名字
    }
}
```

程序运行结果：

```
红色
蓝色
```

以上程序将 Color 类中的构造方法私有，之后在类中准备了若干个实例化对象，以后如果要取得 Color 类的实例，则只能从 RED、GREEN、BLUE 3 个对象中取得，这样就有效地限制了对象的取得范围。

以上使用的 Color 对象指定的范围，是通过一个个常量对每个对象进行编号的。也就是说，一个个的对象就相当于用常量表示了，所以，按照这个思路也可以直接使用一个接口

规定出一组常量的范围。

例如：使用接口表示一组范围

```
package org.lxh.demo14.colordemo;
public interface Color{
    public static final int RED = 1 ;          // 表示红色
    public static final int GREEN = 2 ;        // 表示绿色
    public static final int BLUE = 3 ;         // 表示蓝色
}
```

以上表示出的各个颜色是通过接口指定好的常量范围，与之前相比更加简单。但是这样做也会存在另外一个问题，如果现在使用如下的代码：

```
package org.lxh.demo14.colordemo;
public class ColorDemo02 {
    public static void main(String[] args) {
        System.out.println(Color.RED + Color.GREEN) ;  // 颜色相加
    }
}
```

程序运行结果：

```
3
```

两个颜色的常量相加之后形成了“3”，这样的结果看起来会令人很困惑，操作很不明确。

另外，使用以上这种方式，用户如果想知道到底有多少个颜色可以使用，则实现的代码是非常复杂的。在 JDK 1.5 之后就专门解决了这样的问题。

14.3　定义一个枚举类型

在 JDK 1.5 之后，引入了一个新的关键字类型——enum，可以直接定义枚举类型，格式如 14-1 所示。

【格式 14-1　声明枚举类型】

```
[public] enum 枚举类型名称{
    枚举对象1,枚举对象2,…,枚举对象n ;
}
```

范例：定义一个 Color 的枚举类型

```
package org.lxh.demo14.enumdemo;
public enum Color {                            // 定义枚举类型
    RED, GREEN, BLUE;                          // 定义枚举的3个类型
}
```

以上的 Color 定义出来的枚举类型，其中包含 RED、GREEN、BLUE 3 个数据。可以使用“枚举.variable”的形式取出枚举中的指定内容。

范例：取出一个枚举内容

```
package org.lxh.demo14.enumdemo;
import org.lxh.demo14.Color;
public class GetEnumContent {
    public static void main(String[] args) {
        Color c = Color.BLUE ;                    // 取出蓝色
        System.out.println(c) ;                   // 输出信息
    }
}
```

程序运行结果：

```
BLUE
```

枚举类型的数据也可以使用“枚举.values()”的形式，将全部的枚举类型变为对象数组的形式，之后直接使用 foreach 进行输出。

范例：使用 foreach 输出枚举内容

```
package org.lxh.demo14.enumdemo;
import org.lxh.demo14.Color;
public class PrintEnum {
    public static void main(String[] args) {
        for(Color c:Color.values()){    // 枚举.values()表示得到全部枚举的内容
            System.out.println(c) ;     // 输出枚举内容
        }
    }
}
```

程序运行结果：

```
RED
GREEN
BLUE
```

枚举中的每个类型也可以使用 switch 进行判断，但在 switch 语句中使用枚举类型时，一定不能在每一个枚举类型值的前面加上枚举类型的类名（如 Color.BLUE），否则编译器就会报错。

范例：使用 switch 进行判断

```
package org.lxh.demo14.enumdemo;
import org.lxh.demo14.Color;
public class SwitchPrintDemo {
    public static void main(String[] args) {
        for(Color c:Color.values()){    // 枚举.values()表示得到全部枚举的内容
            print(c) ;
        }
    }
```

```
    public static void print(Color color){
        switch(color){                                    // 判断每个颜色
            case RED:{                                    // 直接判断枚举内容
                System.out.println("红颜色") ;
                break ;
            }
            case GREEN:{                                  // 直接判断枚举内容
                System.out.println("绿颜色") ;
                break ;
            }
            case BLUE:{                                   // 直接判断枚举内容
                System.out.println("蓝颜色") ;
                break ;
            }
            default:{                                     // 未知内容
                System.out.println("未知颜色") ;
                break ;
            }
        }
    }
}
```

程序运行结果：

```
红颜色
绿颜色
蓝颜色
```

上面的 Color.values()方法表示取得枚举中的全部内容，返回的是一个对象数组，这是枚举本身支持的一个方法，除了这个方法之外到底还有哪些方法可以供开发者使用呢？要了解这些，就必须深入分析枚举类——Enum。

14.4　Enum

从前面的讲解中已经清楚地知道，使用 enum 关键字可以定义一个枚举，实际上此关键字表示的是 java.lang.Enum 类型，即使用 enum 声明的枚举类型就相当于定义一个类，而此类则默认继承 java.lang.Enum 类。java.lang.Enum 类的定义如下：

```
public abstract class Enum<E extends Enum<E>>
extends Object
implements Comparable<E>, java.io.Serializable
```

从 Enum 类的定义中可以清楚地发现，此类实现了 Comparable 和 Serializable 两个接口，证明枚举类型可以使用比较器和序列化操作。

此类中包含的方法如表 14-1 所示。

表 14-1 枚举类的主要操作方法

序号	方 法	类型	描 述
1	protected Enum(String name,int ordinal)	构造	接收枚举的名称和枚举的常量创建枚举对象
2	protected final Object clone() throws CloneNotSupportedException	普通	克隆枚举对象
3	public final int compareTo(E o)	普通	对象比较
4	public final boolean equals(Object other)	普通	比较两个枚举对象
5	public final int hashCode()	普通	返回枚举常量的哈希码
6	public final String name()	普通	返回此枚举的名称
7	public final int ordinal()	普通	返回枚举常量的序数
8	public static <T extends Enum<T>> T valueOf (Class<T> enumType,String name)	普通	返回带指定名称的指定枚举类型的枚举常量

下面逐步观察这些方法的使用。

14.4.1 取得枚举的信息

在枚举类建立完成之后，实际上都会为其调用枚举类中的构造方法，为其赋值。在 Enum 类的构造方法中的第一个参数 name 就是定义的枚举的名称，第二个参数 ordinal 则会从 0 开始依次进行编号。之后可以使用 Enum 类中提供的 name()和 ordinal()方法取得名称和编号。

范例：验证 name()和 ordinal()方法

```
package org.lxh.demo14.enumapidemo;
import org.lxh.demo14.Color;
public class GetEnumInfo {
    public static void main(String[] args) {
        for(Color c:Color.values()){    // 枚举.values()表示得到全部枚举的内容
            System.out.println(c.ordinal() + " --> " + c.name()) ;
        }
    }
}
```

程序运行结果：

```
0 --> RED
1 --> GREEN
2 --> BLUE
```

从结果中发现 Enum 类自动为枚举中的每个元素进行编号，而且下标从 0 开始。

14.4.2　为每一个枚举对象属性赋值

1．通过构造方法为属性赋值

每个枚举类中都有其指定好的若干对象，当然，每一个枚举对象中也可以包含多个属性。而这些属性也可以通过构造方法为其赋值。

范例：通过构造方法为枚举属性赋值

```
package org.lxh.demo14.enumapidemo;
enum Color {
    RED("红色"), GREEN("绿色"), BLUE("蓝色");          // 定义枚举的3个类型
    private Color(String name){                        // 定义构造方法
        this.setName(name) ;                           // 设置名字
    }
    private String name ;                              // 定义name属性
    public String getName() {                          // 取得name属性
        return name;
    }
    public void setName(String name) {                 // 设置name属性
        this.name = name;
    }
}
public class ConstructorEnum {
    public static void main(String[] args) {
        for(Color c:Color.values()){                   // 枚举.values()表示得到
                                                       //   全部枚举的内容
            System.out.println(c.ordinal()
                    + " --> " + c.name()
                    + "(" + c.getName() + ")") ;       // 输出
        }
    }
}
```

程序运行结果：

```
0 --> RED(红色)
1 --> GREEN(绿色)
2 --> BLUE(蓝色)
```

以上程序代码在定义的 Color 枚举类中设置了一个 name 属性，并且通过构造方法设置 name 属性的内容。因为 Color 中已经明确地写出了有一个参数的构造方法，所以在声明枚举内容时就必须调用这个构造方法，这样在定义枚举内容时就必须使用如下的语句形式：

```
RED("红色"), GREEN("绿色"), BLUE("蓝色");
```

2. 通过 setter 方法为属性赋值

以上是通过构造方法的方式为属性赋值的，当然也可以通过调用 setter()的方式为指定的属性赋值。但是这样一来就必须明确每一个枚举类的对象，如果操作的对象是 RED，则名字应该为“红色”；如果操作的对象是 GREEN，则名字应该为“绿色”等。

范例：使用 setName()设置内容

```
package org.lxh.demo14.enumapidemo;
enum Color {
    RED, GREEN, BLUE;                                    // 定义枚举的3个类型
    private String name ;                                // 定义name属性
    public String getName() {
        return name;
    }
    public void setName(String name) {
        switch(this){
            case RED:{                                   // 判断是否是红色
                if("红色".equals(name)){
                    this.name = name ;                   // 设置名称
                }else{
                    System.out.println("设置内容错误。") ;
                }
                break ;
            }
            case GREEN:{                                 // 判断是否是绿色
                if("绿色".equals(name)){                 // 设置名称
                    this.name = name ;
                }else{
                    System.out.println("设置内容错误。") ;
                }
                break ;
            }
            case BLUE:{                                  // 判断是否是蓝色
                if("蓝色".equals(name)){                 // 设置名称
                    this.name = name ;
                }else{
                    System.out.println("设置内容错误。") ;
                }
                break ;
            }
        }
    }
}
```

```
public class SetEnum {
    public static void main(String[] args) {
        Color c = Color.BLUE ;
        c.setName("兰色") ;                          // 设置一个错误的名字
        c.setName("蓝色") ;                          // 设置一个正确的名字
        System.out.println(c.getName());
    }
}
```

程序运行结果：

```
设置内容错误。
蓝色
```

以上代码中首先通过枚举类取得了里面的一个对象。之后开始为其设置内容，一开始设置了一个不符合要求的内容，所以会出现“设置内容错误”的提示；而如果设置的内容正确，则可以直接将其赋值给 name 属性。

如果不想通过“枚举类.对象”的形式取得每一个枚举类的对象，也可以使用 Enum 类定义的“枚举类.valueof()”方法的形式进行调用。

范例：使用 valueof()方法找到一个枚举对象

```
public class ValueOfDemo {
    public static void main(String[] args) {
        Color c = Enum.valueOf(Color.class, "BLUE") ;
        c.setName("兰色") ;                          // 设置一个错误的名字
        c.setName("蓝色") ;                          // 设置一个正确的名字
        System.out.println(c.getName());
    }
}
```

程序运行结果：

```
设置内容错误。
蓝色
```

以上操作代码使用了 valueof()，但是在设置第一个参数时使用了“枚举.class”的形式，此为 Java 的反射机制。在随后的章节中会为读者详细解释，此处读者只需要按照此方式使用即可。

14.4.3　使用比较器

在 Enum 类的定义中已经实现好了 Comparable 接口，所以枚举类的内容本身是可以进行排序的，下面通过 TreeSet 演示枚举的排序操作。

在类集部分曾经介绍过 TreeSet 可以直接进行排序，排序的规则是通过 Comparable 接口完成的。

范例：验证枚举比较器

```
package org.lxh.demo14.enumapidemo;
import java.util.Iterator;
import java.util.Set;
import java.util.TreeSet;
enum Color {
    RED, GREEN, BLUE;                                    // 定义枚举的3个类型
}
public class ComparableEnum {
    public static void main(String[] args) {
        Set<Color> t = new TreeSet<Color>() ;            // 只能加入Color类型
        t.add(Color.GREEN) ;                             // 增加绿色的枚举对象
        t.add(Color.BLUE) ;                              // 增加蓝色的枚举对象
        t.add(Color.RED) ;                               // 增加红色的枚举对象
        Iterator<Color> iter = t.iterator() ;            // 使用迭代输出
        while(iter.hasNext()){
            System.out.print(iter.next() + "、");
        }
    }
}
```

程序运行结果：

```
RED、GREEN、BLUE、
```

以上代码中数据加入的顺序是 GREEN、BLUE、RED，但是输出之后的顺序是 RED、GREEN、BLUE，证明已经被排序了，是使用 Enum 类中的 ordinal 属性排序的。

14.5 类集对枚举的支持——EnumMap、EnumSet

在 JDK 1.5 的 java.util 程序包中提供了两个新的集合操作类：EnumMap 和 EnumSet，这两个类与枚举类型的结合应用可使以前非常繁琐的程序变得简单方便。EnumSet 类提供了 java.util.Set 接口的一个特殊实现；EnumMap 类提供了 java.util.Map 接口的一个特殊实现，该类中的键（key）是一个枚举类型。

14.5.1 EnumMap

EnumMap 是 Map 接口的子类，所以本身还是以 Map 的形式进行操作，即 key→value。如果要使用 EnumMap，则首先要创建 EnumMap 的对象，在创建此对象时必须指定要操作的枚举类型，构造方法如下：

```
public EnumMap(Class<K> keyType)
```

范例：验证 EnumMap

```
package org.lxh.demo14.enumcol;
import java.util.EnumMap;
import java.util.Map;
enum Color {
    RED, GREEN, BLUE;                                    // 定义枚举的3个类型
}
public class EnumMapDemo {
    public static void main(String[] args) {
        Map<Color,String> desc = null ;                  // 定义一个Map对象
        desc = new EnumMap<Color,String>(Color.class) ;// 实例化EnumMap
        desc.put(Color.RED, "红色") ;                    // 加入一个内容
        desc.put(Color.GREEN, "绿色") ;                  // 加入一个内容
        desc.put(Color.BLUE, "蓝色") ;                   // 加入一个内容
        System.out.println("===== 输出全部的内容 =====");
        for(Color c:Color.values()){                     // 取得全部的枚举
            System.out.println(c.name() + " --> " + desc.get(c));
        }
        System.out.println("===== 输出全部的键值 =====");
        for(Color c:desc.keySet()){                      // 取得全部的key
            System.out.print(c.name() + "、");
        }
        System.out.println() ;                           // 换行
        System.out.println("===== 输出全部的内容 =====");
        for(String s:desc.values()){                     // 取得全部的value
            System.out.print(s + "、");
        }
    }
}
```

程序运行结果：

```
===== 输出全部的内容 =====
RED --> 红色
GREEN --> 绿色
BLUE --> 蓝色
===== 输出全部的键值 =====
RED、GREEN、BLUE、
===== 输出全部的内容 =====
红色、绿色、蓝色、
```

14.5.2　EnumSet

EnumSet 是 Set 接口的子类，所以里面的内容是无法重复的。使用 EnumSet 时不能直接

使用关键字 new 为其进行实例化，所以在此类中提供了很多的静态方法，如表 14-2 所示。

表 14-2 EnumSet 类的常用方法

序号	方 法	类型	描 述
1	public static <E extends Enum<E>> EnumSet <E> allOf(Class<E> elementType)	普通	将枚举中的全部内容设置到 EnumSet 中
2	public static <E extends Enum<E>> EnumSet <E> of(E first,E... rest)	普通	创建一个包含枚举指定内容的 EnumSet 对象
3	public static <E extends Enum<E>> EnumSet <E> copyOf(Collection<E> c)	普通	创建一个从指定 Collection 中指定的 EnumSet 对象
4	public static <E extends Enum<E>> EnumSet <E> complementOf(EnumSet<E> s)	普通	创建一个其元素类型与指定枚举 set 相同的枚举 set，最初包含指定集合中所不包含的此类型的所有元素
5	public static <E extends Enum<E>> EnumSet <E> noneOf(Class<E> elementType)	普通	创建一个可以接收指定类的空集合

范例：验证 EnumSet——将全部的集合设置到 EnumSet 集合中

```
package org.lxh.demo14.enumcol;
import java.util.EnumSet;
enum Color {
    RED, GREEN, BLUE;                                    // 定义枚举的3个类型
}
public class EnumSetDemo01 {
    public static void main(String[] args) {
        EnumSet<Color> es = null ;                       // 声明一个EnumSet对象
        System.out.println("===== EnumSet.allOf(Color.class) =====");
        es = EnumSet.allOf(Color.class) ;                // 将枚举的全部类型设置到
                                                            EnumSet对象中
        print(es) ;
    }
    public static void print(EnumSet<Color> temp){ // 专门的输出操作
        for(Color c:temp){                               // 循环输出EnumSet中的内容
            System.out.print(c + "、");
        }
        System.out.println() ;
    }
}
```

程序运行结果：

```
===== EnumSet.allOf(Color.class) =====
RED、GREEN、BLUE、
```

以上程序使用 EnumSet 提供的 static 方法 allOf()，将一个枚举中的全部内容设置到

EnumSet 集合中。

范例：验证 EnumSet——只设置一个枚举的类型到集合中

```
package org.lxh.demo14.enumcol;
import java.util.EnumSet;
enum Color {
    RED, GREEN, BLUE;                                    // 定义枚举的3个类型
}
public class EnumSetDemo02 {
    public static void main(String[] args) {
        EnumSet<Color> es = null ;                       // 声明一个EnumSet对象
        System.out.println("===== EnumSet.of(Color.BLUE) =====");
        es = EnumSet.of(Color.BLUE) ;                    // 设置一个枚举的内容
        print(es) ;
    }
    public static void print(EnumSet<Color> temp){ // 专门的输出操作
        for(Color c:temp){                               // 循环输出EnumSet中的内容
            System.out.print(c + "、");
        }
        System.out.println() ;
    }
}
```

程序运行结果：

```
===== EnumSet.of(Color.BLUE) =====
BLUE、
```

以上程序使用 EnumSet 提供的 static 方法 of()，将一个枚举中的一个内容设置到 EnumSet 集合中。

范例：验证 EnumSet——创建只能放入指定枚举类型的集合

```
package org.lxh.demo14.enumcol;
import java.util.EnumSet;
enum Color {
    RED, GREEN, BLUE;                                    // 定义枚举的3个类型
}
public class EnumSetDemo03 {
    public static void main(String[] args) {
        EnumSet<Color> es = null ;                       // 声明一个EnumSet对象
        System.out.println("===== EnumSet.noneOf(Color.class) =====");
        es = EnumSet.noneOf(Color.class) ;               // 创建一个可以加入Color
                                                            类型的对象
        es.add(Color.RED) ;                              // 增加内容
        es.add(Color.GREEN) ;                            // 增加内容
```

```
        print(es) ;
    }
    public static void print(EnumSet<Color> temp){ // 专门的输出操作
        for(Color c:temp){                          // 循环输出EnumSet中的内容
            System.out.print(c + "、");
        }
        System.out.println() ;
    }
}
```

程序运行结果：

```
===== EnumSet.noneOf(Color.class) =====
RED、GREEN、
```

以上程序使用 EnumSet 提供的 static 方法 noneOf()，将集合设置成只能增加 Color 类型的集合，但并不设置任何的内容到集合中。

范例：验证 EnumSet——创建不包含指定元素的集合

```
package org.lxh.demo14.enumcol;
import java.util.EnumSet;
enum Color {
    RED, GREEN, BLUE;                               // 定义枚举的3个类型
}
public class EnumSetDemo04 {
    public static void main(String[] args) {
        EnumSet<Color> esOld = null ;               // 声明一个EnumSet对象
        EnumSet<Color> esNew = null ;               // 声明一个EnumSet对象
        esOld = EnumSet.noneOf(Color.class) ;       // 创建一个可以加入Color
                                                       类型的对象
        esOld.add(Color.RED) ;                      // 增加内容
        esOld.add(Color.GREEN) ;                    // 增加内容
        System.out.println("===== EnumSet.complementOf(es) =====");
        esNew = EnumSet.complementOf(esOld) ;       // 创建一个不包含指定元素
                                                       的集合
        print(esNew) ;
    }
    public static void print(EnumSet<Color> temp){ // 专门的输出操作
        for(Color c:temp){                          // 循环输出EnumSet中的内容
            System.out.print(c + "、");
        }
        System.out.println() ;
    }
}
```

程序运行结果：

```
===== EnumSet.complementOf(es) =====
BLUE、
```

以上程序使用 EnumSet 提供的 static 方法 noneOf()建立了一个集合 esOld，又加入了两个内容，之后以 esOld 的集合为参考建立了一个新的集合，新的集合中将不包含 esOld 中已有的内容。

范例：验证 EnumSet——复制已有的内容

```
package org.lxh.demo14.enumcol;
import java.util.EnumSet;
enum Color {
    RED, GREEN, BLUE;                                    // 定义枚举的3个类型
}
public class EnumSetDemo05 {
    public static void main(String[] args) {
        EnumSet<Color> esOld = null ;                    // 声明一个EnumSet对象
        EnumSet<Color> esNew = null ;                    // 声明一个EnumSet对象
        esOld = EnumSet.noneOf(Color.class) ;            // 创建一个可以加入Color
                                                            类型的对象
        esOld.add(Color.RED) ;                           // 增加内容
        esOld.add(Color.GREEN) ;                         // 增加内容
        System.out.println("===== EnumSet.copyOf(es) =====");
        esNew = EnumSet.copyOf(esOld) ;                  // 从已有的集合中复制出内容
        print(esNew) ;
    }
    public static void print(EnumSet<Color> temp){ // 专门的输出操作
        for(Color c:temp){                               // 循环输出EnumSet中的内容
            System.out.print(c + "、");
        }
        System.out.println() ;
    }
}
```

程序运行结果：

```
===== EnumSet.copyOf(es) =====
RED、GREEN、
```

以上程序通过 EnumSet 中的 copyOf()方法复制了现有的集合内容。

14.6　让枚举类实现一个接口

枚举类也可以实现一个接口，但是因为接口中会存在抽象方法，所以枚举类中的每个

对象都必须分别实现此抽象方法，如下所示。

范例：让枚举类实现一个接口

```
package org.lxh.demo14.otherdemo;
interface Print {                                          // 定义Print接口
    public String getColor();                              // 定义抽象方法
}
enum Color implements Print {                              // 枚举类实现接口
    RED {                                                  // 枚举对象实现抽象方法
        public String getColor() {
            return "红色";
        }
    },
    GREEN {                                                // 枚举对象实现抽象方法
        public String getColor() {
            return "绿色";
        }
    },
    BLUE {                                                 // 枚举对象实现抽象方法
        public String getColor() {
            return "蓝色";
        }
    };
}
public class InterfaceEnumDemo {
    public static void main(String[] args) {
        for(Color c:Color.values()){                       // foreach输出
            System.out.print(c.getColor() + "、") ;// 输出
        }
    }
}
```

程序运行结果：

```
红色、绿色、蓝色、
```

以上程序在接口中定义了一个 getColor()方法，枚举类在实现此接口之后，就必须对枚举类中的每个对象分别实现接口中的 getColor()方法。

14.7 在枚举类中定义抽象方法

枚举类除了可以实现接口外，还可以在枚举类中定义抽象方法，这样每个枚举的对象只要分别实现了此抽象方法即可。

范例：在枚举类中定义抽象方法

```
package org.lxh.demo14.otherdemo;
enum Color {                                          // 枚举类实现接口
    RED {                                             // 枚举对象实现抽象方法
        public String getColor() {
            return "红色";
        }
    },
    GREEN {                                           // 枚举对象实现抽象方法
        public String getColor() {
            return "绿色";
        }
    },
    BLUE {                                            // 枚举对象实现抽象方法
        public String getColor() {
            return "蓝色";
        }
    };
    public abstract String getColor();                // 定义抽象方法
}
public class AbstractMethodEnum {
    public static void main(String[] args) {
        for(Color c:Color.values()){                  // foreach输出
            System.out.print(c.getColor() + "、") ;// 输出
        }
    }
}
```

程序运行结果：

```
红色、绿色、蓝色、
```

14.8　本 章 要 点

1．在程序中可以使用一个枚举来指定对象的取值范围。

2．在 Java 中使用 enum 关键字定义一个枚举类，每一个枚举类都是继承 Enum 类。

3．在枚举中可以通过 values()方法取得枚举中的全部内容。

4．在枚举类中可以定义构造方法，但在设置枚举范围时必须显式地调用构造方法。

5．所有的枚举类都可以直接使用 Comparable 进行排序，因为 Enum 类实现了 Comparable 接口。

6．Java 类集中提供枚举的支持类是 EnumMap、EnumSet。

7．一个枚举类可以实现一个接口或者直接定义一个抽象方法，但是每个枚举对象都必须分别实现全部的抽象方法。

14.9 习　　题

1．定义一个品牌电脑的枚举类，其中只有固定的几个电脑品牌。

2．定义一个 Person 类，其中包含姓名、年龄、生日、性别的属性，其中性别只能是“男”或“女”。

第 15 章　Java 反射机制

通过本章的学习可以达到以下目标：

- ☑ 了解反射的基本原理。
- ☑ 掌握 Class 类的使用。
- ☑ 使用 Class 类并结合其他类取得一个类的完整结构。
- ☑ 通过反射机制动态地调用类中的指定方法，并能向这些方法中传递参数。

在 Java 中较为重要的就是反射机制，那么什么是反射机制呢？举个简单的例子来说，正常情况下如果已经有一个类，则肯定可以通过类创建对象；那么如果现在要求通过一个对象找到一个类的名称，此时就需要用到反射机制。如果要完成反射操作，则首先应该认识的就是 Class 类。

在反射操作的学习中，读者一定要把握住一个核心的概念："一切的操作都将使用 Object 完成，类、数组的引用都可以使用 Object 进行接收"，只有把握了这个概念才能更清楚地掌握反射机制的作用。**本章视频录像讲解时间为 2 小时 24 分钟，源代码在光盘对应的章节下。**

15.1　认识 Class 类

在正常情况下，需要先有一个类的完整路径引入之后才可以按照固定的格式产生实例化对象，但是在 Java 中也允许通过一个实例化对象找到一个类的完整信息，那么这就是 Class 类的功能。在讲解 Class 类之前，先来看以下一段代码。

范例：通过一个对象得到完整的"包.类"名称

```
package org.lxh.demo15.getclassdemo;
class X {                                                    // 声明X类
}
public class GetClassDemo01 {
    public static void main(String args[]) {
        X x = new X();                                       // 实例化X类的对象
        System.out.println(x.getClass().getName());          // 得到对象所在类
    }
};
```

程序运行结果：

```
org.lxh.demo15.getclassdemo.X
```

从程序的运行结果中可以发现，通过一个对象得到了对象所在的完整的"包.类"名称，那么有些读者会觉得奇怪，getClass()方法是哪里定义的呢？从之前学习到的知识读者知道，

任何一个类如果没有明确地声明继承自哪个父类时，则默认继承 Object 类，所以 getClass() 方法是 Object 类中的，此方法的定义如下：

```
public final Class getClass()
```

以上方法返回值的类型是一个 Class 类，实际上此类是 Java 反射的源头。所谓反射从程序的运行结果来看也很好理解，即可以通过对象反射求出类的名称，如图 15-1 所示。



图 15-1　反射过程

提示：所有类的对象实际上都是 Class 类的实例。

在 Java 中 Object 类是一切类的父类，那么所有类的对象实际上也就都是 java.lang.Class 类的实例，所以所有的对象都可以转变为 java.lang.Class 类型表示。

在定义 Class 类时也使用了泛型声明，所以为了让程序不出现警告信息，可以指定好其操作的泛型类型，下面介绍 Class 类的作用。

Class 本身表示一个类的本身，通过 Class 可以完整地得到一个类中的完整结构，包括此类中的方法定义、属性定义等。表 15-1 定义了 Class 类的一些常用操作。

表 15-1　Class 类的常用方法

序号	方　法	类型	描　述
1	public static Class<?> forName(String className) throws ClassNotFoundException	普通	传入完整的“包.类”名称实例化 Class 对象
2	public Constructor[] getConstructors() throws Security Exception	普通	得到一个类中的全部构造方法
3	public Field[] getDeclaredFields() throws Security Exception	普通	得到本类中单独定义的全部属性
4	public Field[] getFields() throws SecurityException	普通	取得本类继承而来的全部属性
5	public Method[] getMethods() throws SecurityException	普通	得到一个类中的全部方法
6	public Method getMethod(String name,Class... parameterTypes)throws NoSuchMethodException,SecurityException	普通	返回一个 Method 对象，并设置一个方法中的所有参数类型
7	public Class[] getInterfaces()	普通	得到一个类中所实现的全部接口
8	public String getName()	普通	得到一个类完整的“包.类”名称
9	public Package getPackage()	普通	得到一个类的包
10	public Class getSuperclass()	普通	得到一个类的父类
11	public Object newInstance() throws InstantiationException, IllegalAccessException	普通	根据 Class 定义的类实例化对象
12	public Class<?> getComponentType()	普通	返回表示数组类型的 Class
13	public boolean isArray()	普通	判断此 Class 是否是一个数组

在 Class 类中本身没有定义任何的构造方法，所以如果要使用则首先必须通过 forName() 方法实例化对象。除此之外，也可以使用“类.class”或“对象.getClass()”方法实例化。

范例：实例化 Class 类对象

```
package org.lxh.demo15.getclassdemo;
class X {                                                    // 定义X类
}
public class GetClassDemo02 {
    public static void main(String args[]) {
        Class<?> c1 = null;                                  // 指定泛型
        Class<?> c2 = null;                                  // 指定泛型
        Class<?> c3 = null;                                  // 指定泛型
        try {
            c1 = Class.forName(
                    "org.lxh.demo15.getclassdemo.X");        // 最常用的形式
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        c2 = new X().getClass();                             // 通过Object类中的
                                                                方法实例
        c3 = X.class;                                        // 通过类.class实例化
        System.out.println("类名称：" + c1.getName()) ;      // 得到类的名称
        System.out.println("类名称：" + c2.getName()) ;      // 得到类的名称
        System.out.println("类名称：" + c3.getName()) ;      // 得到类的名称
    }
}
```

程序运行结果：

```
类名称：org.lxh.demo15.getclassdemo.X
类名称：org.lxh.demo15.getclassdemo.X
类名称：org.lxh.demo15.getclassdemo.X
```

从程序的运行结果中可以发现，3 种实例化 Class 对象的方式是一样的，但是使用 forName()的静态方法实例化 Class 对象的方式是较为常用的一种方式，读者应该重点掌握。

提示：使用 forName()方法更加具备灵活性。

从以上程序的 3 种使用方法上可以发现，除 forName()方法外，其他的两种：“对象.getClass()”、“类.class”都需要导入一个明确的类，如果一个类操作不明确时，使用起来可能会受到一些限制。但是 forName()传入时只需要以字符串的方式传入即可，这样就让程序具备了更大的灵活性，所以此种方法是最为常见的一种方式，在本章后续部分将结合设计模式一起讲解其使用。

15.2 Class 类的使用

了解了 Class 类的实例化过程，那么到底该如何去使用 Class 类呢？实际上 Class 类在开发中最常见的用法就是实例化对象的操作，即可以通过一个给定的字符串（此字符串包含了完整的“包.类”的路径）来实例化一个类的对象。

15.2.1 通过无参构造实例化对象

如果要想通过 Class 类本身实例化其他类的对象，则可以使用 newInstance()方法，但是必须要保证被实例化的类中存在一个无参构造方法，代码如下所示。

范例：通过 Class 类实例化对象

```
package org.lxh.demo15.instancedemo;
class Person {                                    // 定义Person类
    private String name;                          // 定义name属性
    private int age;                              // 定义age属性
    public String getName() {                     // 取得name属性
        return name;
    }
    public void setName(String name) {            // 设置name属性
        this.name = name;
    }
    public int getAge() {                         // 取得age属性
        return age;
    }
    public void setAge(int age) {                 // 设置age属性
        this.age = age;
    }
    public String toString() {                    // 覆写toString()方法
        return "姓名：" + this.name + "，年龄：" + this.age;
    }
}
public class InstanceDemo01 {
    public static void main(String[] args) {
        Class<?> c = null;                        // 指定泛型
        try {
            c = Class.forName(
                    "org.lxh.demo15.instancedemo.Person");
                                                  // 传入要实例化类的完整包.类名称
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
```

```
        }
        Person per = null;                          // 声明Person对象
        try {
            per = (Person) c.newInstance();         // 实例化Person对象
        } catch (Exception e) {
            e.printStackTrace();
        }
        per.setName("李兴华");                       // 设置姓名
        per.setAge(30);                             // 设置年龄
        System.out.println(per);                    // 内容输出，调用toString()
    }
}
```

程序运行结果：

```
姓名：李兴华，年龄：30
```

从程序的运行结果可以发现，通过 Class.forName()方法实例化 Class 对象之后，直接调用 newInstance()方法就可以根据传入的完整“包.类”名称的方式进行对象的实例化操作，完全取代了之前使用关键字 new 的操作方式。

但是在使用以上操作时读者一定要记住一点，被实例化对象的类中必须存在无参构造方法，如果不存在，则肯定是无法实例化的。以下代码为读者演示了一个错误的实例化操作。

范例：错误的代码

```
package org.lxh.demo15.instancedemo;
class Person {                                      // 定义Person类
    private String name;                            // 定义name属性
    private int age;                                // 定义age属性
    public Person(String name,int age){             // 通过构造设置属性内容
        this.setName(name) ;                        // 设置name属性内容
        this.setAge(age) ;                          // 设置age属性内容
    }
    public String getName() {                       // 取得name属性
        return name;
    }
    public void setName(String name) {              // 设置name属性
        this.name = name;
    }
    public int getAge() {                           // 取得age属性
        return age;
    }
    public void setAge(int age) {                   // 设置age属性
        this.age = age;
```

```
    }
    public String toString() {                    // 覆写toString()方法
        return "姓名：" + this.name + "，年龄：" + this.age;
    }
}
public class InstanceDemo02 {
    public static void main(String[] args) {
        Class<?> c = null;
        try {
            c = Class.forName(
                    "org.lxh.demo15.instancedemo.Person");
                                                  // 传入要实例化类的包.类名称
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        Person per = null;
        try {
            per = (Person) c.newInstance();       // 实例化Person对象
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

程序运行时出现以下错误：

```
java.lang.InstantiationException: org.lxh.demo15.instancedemo.Person
    at java.lang.Class.newInstance0(Unknown Source)
    at java.lang.Class.newInstance(Unknown Source)
    at org.lxh.demo15.instancedemo.InstanceDemo02.main(InstanceDemo02.
java:37)
```

从以上代码中可以发现，因为类中并没有存在无参构造方法，所以是根本无法直接使用 newInstance()方法实例化的。在此，笔者建议读者在使用 Class 类实例化对象时一定要在类中编写无参构造方法。

提示：各种高级应用中都提倡类中存在无参构造方法。

在实际的 Java 程序开发中，反射是最为重要的操作原理，在现在的开发设计中大量地应用了反射处理机制，如 Struts、Spring 框架等；在大部分的操作中基本上都是操作无参构造方法，所以希望读者在日后使用反射开发时类中一定要保留无参构造方法。

15.2.2 调用有参构造实例化对象

当然，对于以上程序也并非没有解决的方法，可以通过其他的方式进行实例化操作，

只是在操作时需要明确地调用类中的构造方法，并将参数传递进去之后才可以进行实例化操作。操作步骤如下：

（1）通过 Class 类中的 getConstructors()取得本类中的全部构造方法。

（2）向构造方法中传递一个对象数组进去，里面包含了构造方法中所需的各个参数。

（3）之后通过 Constructor 实例化对象。

在这里使用了 Constructor 类，表示的是构造方法。此类的常用方法如表 15-2 所示。

表 15-2　Constructor 常用方法

序号	方　法	类型	描　述
1	public int getModifiers()	普通	得到构造方法的修饰符
2	public String getName()	普通	得到构造方法的名称
3	public Class<?>[] getParameterTypes()	普通	得到构造方法中参数的类型
4	public String toString()	普通	返回此构造方法的信息
5	public T newInstance(Object... initargs) throws InstantiationException,IllegalAccessException,IllegalArgumentException, InvocationTargetException	普通	向构造方法中传递参数，实例化对象

下面先使用 newInstance()方法解决实例化对象的问题，其他方法在之后会为读者介绍。

范例：调用类中的有参构造

```
package org.lxh.demo15.instancedemo;
import java.lang.reflect.Constructor;                    // 导入反射包
class Person {                                           // 定义Person类
    private String name;                                 // 定义name属性
    private int age;                                     // 定义age属性
    public Person(String name,int age){                  // 通过构造设置属性内容
        this.setName(name) ;                             // 设置name属性内容
        this.setAge(age) ;                               // 设置age属性内容
    }
    public String getName() {                            // 取得name属性
        return name;
    }
    public void setName(String name) {                   // 设置name属性
        this.name = name;
    }
    public int getAge() {                                // 取得age属性
        return age;
    }
    public void setAge(int age) {                        // 设置age属性
        this.age = age;
    }
    public String toString() {                  // 覆写toString()方法
```

```
            return "姓名: " + this.name + ", 年龄: " + this.age;
        }
    }
    public class InstanceDemo03 {
        public static void main(String[] args) {
            Class<?> c = null;
            try {
                c = Class.forName(
                        "org.lxh.demo15.instancedemo.Person");// 声明Class对象
            } catch (ClassNotFoundException e) {
                e.printStackTrace();
            }
            Person per = null;                          // 声明Person对象
            Constructor<?> cons[] = null;               // 声明一个表示构造方法的数组
            cons = c.getConstructors();                 // 通过反射取得全部构造
            try {
                // 向构造方法中传递参数，此方法使用可变参数接收，并实例化对象
                per = (Person) cons[0].newInstance("李兴华", 30);
            } catch (Exception e) {                     // 因为只有一个构造，所以数组下标为0
                e.printStackTrace();
            }
            System.out.println(per);                    // 输出对象
        }
    }
```

程序运行结果：

```
姓名：李兴华，年龄：30
```

以上程序通过 Class 类取得了一个类中的全部构造方法，并以对象数组的形式返回，因为显式地调用了类中的构造方法，且在 Person 类中只有一个构造方法，所以直接取出对象数组中的第一个元素即可（下标为 0 就表示调用第一个构造方法）。在声明对象数组时，必须考虑到构造方法中参数的类型顺序，所以第一个参数的类型为 String，第二个参数的类型为 Integer（在使用时可以自动拆箱），如图 15-2 所示。

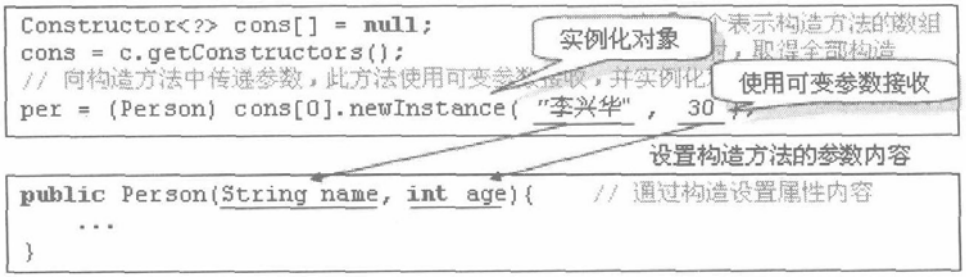


图 15-2　调用构造实例化对象

以上代码虽然已经完成了对象的实例化过程，但是读者可以发现以上代码比较复杂，

所以建议读者最好在类中留一个无参构造方法。

15.3　反射的应用——取得类的结构

在实际开发中，以上程序就是反射应用最多的地方。当然，反射机制所提供的功能远不止如此，还可以通过反射得到一个类的完整结构，那么这就要使用到 java.lang.reflect 包中的以下几个类。

- Constructor：表示类中的构造方法。
- Field：表示类中的属性。
- Method：表示类中的方法。

这 3 个类都是 AccessibleObject 类的子类，如图 15-3 所示。

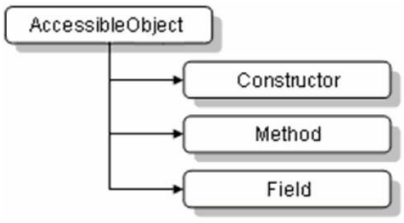


图 15-3　类的继承关系

下面通过以上几个类和 Class 类共同完成类的反射操作。

范例：Person.java

```
package org.lxh.demo15;
interface China {                                          // 定义China接口
    public static final String NATIONAL = "China";        // 定义全局常量
    public static final String AUTHOR = "李兴华";          // 定义全局常量
    public void sayChina();                                // 定义无参的抽象方法
    public String sayHello(String name, int age);          // 定义有参的抽象方法
}
public class Person implements China {                     // 定义Person类实现接口
    private String name;                                   // 定义name属性
    private int age;                                       // 定义age属性
    public Person() {                                      // 声明无参构造
    }
    public Person(String name) {                           // 声明有一个参数的构造方法
        this.name = name ;                                 // 为name属性赋值
    }
    public Person(String name, int age) {                  // 声明有两个参数的构造方法
        this(name);                                        // 调用有一个参数的构造
        this.setAge(age);                                  // 为age属性赋值
    }
    public String getName() {                              // 取得name属性内容
```

```
        return name;
    }
    public void setName(String name) {                    // 设置name属性内容
        this.name = name;
    }
    public int getAge() {                                 // 取得age属性内容
        return age;
    }
    public void setAge(int age) {                         // 设置age属性内容
        this.age = age;
    }
    public void sayChina() {                              // 覆写方法输出信息
        System.out.println("作者: " + AUTHOR + ", 国籍: " + NATIONAL);
    }
    public String sayHello(String name, int age) {        // 覆写方法，返回信息
        return name + ", 你好! 我今年" + age + "岁了! ";
    }
}
```

15.3.1 取得所实现的全部接口

要取得一个类所实现的全部接口，则必须使用 Class 类中的 getInterfaces ()方法。此方法定义如下：

```
public Class[] getInterfaces()
```

getInterfaces()方法返回一个 Class 类的对象数组，之后直接利用 Class 类中的 getName()方法输出即可。

范例：取得 Person 类实现的全部接口

```
package org.lxh.demo15.classinfodemo;
public class GetInterfaceDemo {
    public static void main(String[] args) {
        Class<?> c1 = null ;                                  // 声明Class对象
        try {
            c1 = Class.forName("org.lxh.demo15.Person") ;     // 实例化Class
                                                                对象
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        Class<?> c[] = c1.getInterfaces() ;                   // 取得实现的全
                                                                部接口
        for(int i=0;i<c.length;i++){
```

```
                System.out.println("实现的接口名称：" + c[i].getName());
                                                        // 输出接口名称
            }
        }
    }
```

程序运行结果：

```
实现的接口名称：org.lxh.demo15.China
```

因为接口是类的特殊形式，而且一个类可以实现多个接口，所以此时以 Class 数组的形式将全部的接口对象返回，并利用循环的方式将内容依次输出。

15.3.2　取得父类

一个类可以实现多个接口，但是只能继承一个父类，所以如果要取得一个类的父类，可以直接使用 Class 类中的 getSuperclass()方法。此方法定义如下：

```
public Class<? super T> getSuperclass()
```

getSuperclass()方法返回的是 Class 实例，和之前得到接口一样，可以通过 getName()方法取得名称。

范例：取得 Person 的父类

```
package org.lxh.demo15.classinfodemo;
public class GetSuperClassDemo {
    public static void main(String[] args) {
        Class<?> c1 = null ;                                    // 声明Class对象
        try {
            c1 = Class.forName("org.lxh.demo15.Person");// 实例化Class对象
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        Class<?> c2 = c1.getSuperclass();                       // 取得父类
        System.out.println("父类名称：" + c2.getName()); // 输出信息
    }
}
```

程序运行结果：

```
父类名称：java.lang.Object
```

Person 类在编写时没有明确地继承一个父类，所以默认继承 Object 类。

15.3.3　取得全部构造方法

要取得一个类中的全部构造方法，则必须使用 Class 类中的 getConstructors()方法。在表 15-2 中已经为读者介绍过 Constructor 类的常用方法，下面使用这些方法为读者列出一个

类中全部的构造方法。

范例：取得 Person 类中的全部构造方法

```
package org.lxh.demo15.classinfodemo;
import java.lang.reflect.Constructor;                    // 导入反射操作包
public class GetConstructorDemo01 {
    public static void main(String[] args) {
        Class<?> c1 = null ;                             // 声明Class对象
        try {
            c1 = Class.forName("org.lxh.demo15.Person");// 实例化Class对象
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        Constructor<?> con[] = c1.getConstructors();     // 得到全部构造方法
        for (int i = 0; i < con.length; i++) {           // 循环输出
            System.out.println("构造方法：" + con[i]);    // 直接打印输出
        }
    }
}
```

程序运行结果：

```
构造方法：public org.lxh.demo15.Person()
构造方法：public org.lxh.demo15.Person(java.lang.String,int)
构造方法：public org.lxh.demo15.Person(java.lang.String)
```

以上确实取得了全部的构造方法，取得时是直接通过输出 Constructor 对象得到的完整信息，是比较全的信息。当然，用户也可以自己手工拼凑出信息，在表 15-2 中定义了取得一个构造方法中名称、访问权限和参数类型的操作。下面使用这 3 个方法取得一个类的全部构造方法。

范例：取得一个类的全部构造方法

```
package org.lxh.demo15.classinfodemo;
import java.lang.reflect.Constructor;
public class GetConstructorDemo02 {
    public static void main(String[] args) {
        Class<?> c1 = null ;                             // 声明Class对象
        try {
            c1 = Class.forName("org.lxh.demo15.Person");// 实例化Class对象
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        Constructor<?> con[] = c1.getConstructors();     // 得到全部构造
        for (int i = 0; i < con.length; i++) {           // 循环输出
            Class<?> p[] = con[i].getParameterTypes();// 列出构造中的参数类型
```

```
                System.out.print("构造方法：") ;
                System.out.print(con[i].getModifiers() + " ") ;     // 取出权限
                System.out.print(con[i].getName());                 // 输出构造
                                                                       方法名称
                System.out.print("(");                              // 输出“(”
                for (int j = 0; j < p.length; j++) {                // 循环输出
                    System.out.print(p[j].getName() + " arg" + i); // 打印参数
                                                                       类型
                    if (j < p.length - 1) {                         // 判断是否
                                                                       要输出“,”
                        System.out.print(",");                      // 输出“,”
                    }
                }
                System.out.println("){}");                          // 输出“){}”
            }
        }
    }
```

程序运行结果：

```
构造方法：1 org.lxh.demo15.Person(){}
构造方法：1 org.lxh.demo15.Person(java.lang.String arg1,int arg1){}
构造方法：1 org.lxh.demo15.Person(java.lang.String arg2){}
```

从程序的运行结果中可以发现，已经取得了构造方法的方法名称及参数类型，但是在取得权限时却发现返回的是一个数字而不是 public。这是因为在整个 Java 中对于方法的修饰符是使用一定的数字表示出来的，而如果要把这个数字还原成用户可以看懂的关键字，则必须依靠 Modifier 类完成，此类定义在 java.lang.reflect 包中。直接使用 Modifier 类的以下方法即可还原修饰符：

```
public static String toString(int mod)
```

范例：使用 Modifier 还原修饰符

```
package org.lxh.demo15.classinfodemo;
import java.lang.reflect.Constructor;
import java.lang.reflect.Modifier;
public class GetConstructorDemo03 {
    public static void main(String[] args) {
        Class<?> c1 = null ;                                    // 声明Class对象
        try {
            c1 = Class.forName("org.lxh.demo15.Person");// 实例化Class对象
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        Constructor<?> con[] = c1.getConstructors();    // 得到全部构造
```

```
        for (int i = 0; i < con.length; i++) {                    // 循环输出
            Class<?> p[] = con[i].getParameterTypes();            // 列出构造中的
                                                                     参数类型
            System.out.print("构造方法: ") ;
            int mo = con[i].getModifiers() ;                      // 取出权限
            System.out.print(Modifier.toString(mo) + " ") ;// 还原权限
            System.out.print(con[i].getName());                   // 输出构造方法
                                                                     名称
            System.out.print("(");                                // 输出“(”
            for (int j = 0; j < p.length; j++) {                  // 循环输出
                System.out.print(p[j].getName() + " arg" + i);
                                                                  // 打印参数类型
                if (j < p.length - 1) {                           // 判断是否要输
                                                                     出“,”
                    System.out.print(",");                        // 输出“,”
                }
            }
            System.out.println("){}");                           // 输出“){}”
        }
    }
}
```

程序运行结果：

```
构造方法: public org.lxh.demo15.Person(){}
构造方法: public org.lxh.demo15.Person(java.lang.String arg1,int arg1){}
构造方法: public org.lxh.demo15.Person(java.lang.String arg2){}
```

从程序的运行结果中可以发现，使用 Modifier 将取出来的修饰符数字还原成了用户可以看得懂的权限修饰符。

15.3.4 取得全部方法

要取得一个类中的全部方法，可以使用 Class 类中的 getMethods()方法，此方法返回一个 Method 类的对象数组。而如果要想进一步取得方法的具体信息，例如，方法的参数、抛出的异常声明等，则就必须依靠 Method 类，此类中的常用方法如表 15-3 所示。

表 15-3 Method 类中的常用方法

序号	方 法	类型	描 述
1	public int getModifiers()	普通	取得本方法的访问修饰符
2	public String getName()	普通	取得方法的名称
3	public Class<?>[] getParameterTypes()	普通	得到方法的全部参数类型
4	public Class<?> getReturnType()	普通	得到方法的返回值类型

续表

序号	方　法	类型	描　述
5	public Class<?>[] getExceptionTypes()	普通	得到一个方法的全部抛出异常
6	public Object invoke(Object obj,Object... args) throws IllegalAccessException,IllegalArgument Exception,InvocationTargetException	普通	通过反射调用类中的方法，此方法在后面将为读者介绍

下面利用以上方法取得一个类中的全部方法定义。

范例：取得一个类的全部方法定义

```
package org.lxh.demo15.classinfodemo;
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;
public class GetMethodDemo {
    public static void main(String[] args) {
        Class<?> c1 = null ;                                    // 声明Class对象
        try {
            c1 = Class.forName("org.lxh.demo15.Person");// 实例化Class对象
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        Method m[] = c1.getMethods();                           // 取得全部的方法
        for (int i = 0; i < m.length; i++) {                    // 循环输出
            Class<?> r = m[i].getReturnType();                  // 得到方法的返回值
                                                                   类型
            Class<?> p[] = m[i].getParameterTypes();            // 得到全部的参数类型
            int xx = m[i].getModifiers();                       // 得到方法的修饰符
            System.out.print(Modifier.toString(xx) + " ");// 还原修饰符
            System.out.print(r.getName() + " ");                // 得到方法名称
            System.out.print(m[i].getName());                   // 取得方法名称
            System.out.print("(");                              // 输出“(”
            for (int x = 0; x < p.length; x++) {                // 循环输出
                System.out.print(p[x].getName() + " " + "arg" + x);
                                                                // 输出参数
                if (x < p.length - 1) {                         // 判断是否输出“,”
                    System.out.print(",");                      // 输出“,”
                }
            }
            Class<?> ex[] = m[i].getExceptionTypes() ; // 得到全部的异常抛出
            if(ex.length>0){                                    // 判断是否有异常
                System.out.print(") throws ");                  // 输出“)throws”
            }else{
                System.out.print(") ") ;                        // 输出“)”
```

```
                }
                for (int j = 0; j < ex.length; j++) {          // 循环输出
                    System.out.print(ex[j].getName()) ;        // 输出异常信息
                    if (j < ex.length - 1) {                   // 判断是否输出“,”
                        System.out.print(",") ;                // 输出“,”
                    }
                }
                System.out.println();                          // 换行
            }
        }
    }
```

程序运行结果：

```
    public void setAge(int arg0)
    public int getAge()
    public void sayChina()
    public java.lang.String sayHello(java.lang.String arg0,int arg1)
    public java.lang.String getName()
    public void setName(java.lang.String arg0)
    public final void wait() throws java.lang.InterruptedException
    public final void wait(long arg0,int arg1) throws java.lang.Interrupted
Exception
    public final native void wait(long arg0) throws java.lang.Interrupted
Exception
    public native int hashCode()
    public final native java.lang.Class getClass()
    public boolean equals(java.lang.Object arg0)
    public java.lang.String toString()
    public final native void notify()
    public final native void notifyAll()
```

从程序的运行结果可以发现，程序不仅将 Person 类的方法输出，也把从 Object 类中继承而来的方法同样进行了输出。

提示：开发工具是利用反射的原理。

在使用 IDE 进行程序开发时，基本上都是带随笔提示功能的，即使用一个“.”就可以找到一个类的全部方法，实际上这个功能就是利用此种方式完成的。

15.3.5 取得全部属性

在反射操作中也同样可以取得一个类中的全部属性，但是在取得属性时有以下两种不同的操作。

- 得到实现的接口或父类中的公共属性：public Field[] getFields() throws SecurityException。

➘ 得到本类中的全部属性：public Field[] getDeclaredFields() throws SecurityException。

以上方法返回的都是 Field 的数组，每一个 Field 对象表示类中的一个属性，而要想取得属性的进一步信息，就需要使用表 15-4 所示的方法。

表 15-4　Field 类的常用方法

序号	方　法	类型	描　述
1	public Object get(Object obj) throws IllegalArgumentException,IllegalAccessException	普通	得到一个对象中属性的具体内容
2	public void set(Object obj,Object value) throws IllegalArgumentException,IllegalAccessException	普通	设置指定对象中属性的具体内容
3	public int getModifiers()	普通	得到属性的修饰符
4	public String getName()	普通	返回此属性的名称
5	public boolean isAccessible()	普通	判断此属性是否可被外部访问
6	public void setAccessible(boolean flag) throws SecurityException	普通	设置一个属性是否可被外部访问
7	public static void setAccessible(AccessibleObject[] array,boolean flag) throws SecurityException	普通	设置一组属性是否可被外部访问
8	public String toString()	普通	返回此 Field 类的信息

以上是 Field 中的一些主要方法，其中设置和取得属性的操作在后面将为读者介绍，下面介绍如何取得一个类中的全部属性信息。

范例：取得 Person 类中的属性

```
package org.lxh.demo15.classinfodemo;
import java.lang.reflect.Field;
import java.lang.reflect.Modifier;
public class GetFieldDemo {
    public static void main(String[] args) {
        Class<?> c1 = null ;                                    // 声明Class对象
        try {
            c1 = Class.forName("org.lxh.demo15.Person");// 实例化Class对象
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        {                                                       // 普通代码块
            Field f[] = c1.getDeclaredFields();                 // 取得本类属性
            for (int i = 0; i < f.length; i++) {                // 循环输出
                Class<?> r = f[i].getType();                    // 取得属性的类型
                int mo = f[i].getModifiers() ;                  // 得到修饰符数字
                String priv = Modifier.toString(mo);            // 取得属性的修饰符
                System.out.print("本类属性：");
                System.out.print(priv + " ");                   // 输出修饰符
                System.out.print(r.getName() + " ");            // 输出属性类型
```

```
                System.out.print(f[i].getName());          // 输出属性名称
                System.out.println(" ;");                  // 换行
            }
        }
        System.out.println("-----------------------------------");
        {                                                  // 普通代码块
            Field f[] = c1.getFields();                    // 取得父类公共属性
            for (int i = 0; i < f.length; i++) {           // 循环输出
                Class<?> r = f[i].getType();               // 取得属性的类型
                int mo = f[i].getModifiers() ;             // 取得修饰符数字
                String priv = Modifier.toString(mo);       // 取得属性修饰符
                System.out.print("公共属性: ");
                System.out.print(priv + " ");              // 输出修饰符
                System.out.print(r.getName() + " ");       // 输出参数类型
                System.out.print(f[i].getName());          // 输出属性名称
                System.out.println(" ;");                  // 换行
            }
        }
    }
}
```

程序运行结果：

```
本类属性: private java.lang.String name ;
本类属性: private int age ;
-----------------------------------
公共属性: public static final java.lang.String NATIONAL ;
公共属性: public static final java.lang.String AUTHOR ;
```

15.4 Java 反射机制的深入应用

反射除了可以取得一个类的完整结构外，还可以调用类中的指定方法或指定属性，并且可以通过反射完成对数组的操作。

15.4.1 通过反射调用类中的方法

如果要使用反射调用类中的方法可以通过 Method 类完成，操作步骤如下：

（1）通过 Class 类的 getMethod(String name,Class...parameterTypes) 方法取得一个 Method 的对象，并设置此方法操作时所需要的参数类型。

（2）之后才可以使用 invoke 进行调用，并向方法中传递要设置的参数。

下面通过一个具体的范例来为读者演示操作，此操作主要完成调用 Person 类中 sayChina()方法的功能。

范例：调用 Person 类中的 sayChina()方法

```
package org.lxh.demo15.invokedemo;
import java.lang.reflect.Method;
public class InvokeSayChinaDemo {
    public static void main(String[] args) {
        Class<?> c1 = null ;                                    // 声明Class对象
        try {
            c1 = Class.forName("org.lxh.demo15.Person");// 实例化Class对象
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        try {
            Method met = c1.getMethod("sayChina") ;     // 此方法没有参数
            met.invoke(c1.newInstance()) ;              // 调用方法，必须传递
                                                        对象实例
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

程序运行结果：

```
作者：李兴华，国籍：China
```

以上程序中通过 Class 类的 getMethod()方法根据一个类中的方法名称取得 Method 对象，并通过 invoke 调用指定的方法。但是在使用 invoke()方法时必须传入一个类的实例化对象，因为在 sayChina()方法上没有任何的参数，所以此处没有设置参数类型和参数内容，本程序的操作如图 15-4 所示。

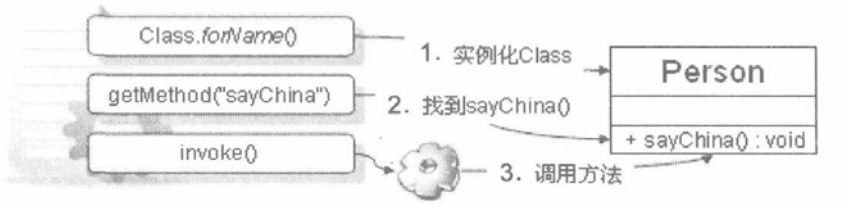


图 15-4　程序操作

下面再为读者演示一个向方法中传递参数的实例，以调用 Person 类中的 sayHello(String name,int age)方法为例，此方法需要传递两个参数。

范例：调用 Person 类中的 sayHello()方法

```
package org.lxh.demo15.invokedemo;
import java.lang.reflect.Method;
public class InvokeSayHelloDemo {
    public static void main(String[] args) {
```

```
        Class<?> c1 = null ;                                    // 声明Class对象
        try {
            c1 = Class.forName("org.lxh.demo15.Person");// 实例化Class对象
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        try {
            Method met = c1.getMethod("sayHello",
                    String.class,int.class) ;                   // 此方法需要两个参数
            String rv = null ;                                  // 接收方法的返回值
            // 调用方法，必须传递对象实例，同时传递两个参数值
            rv = (String)met.invoke(c1.newInstance(),"李兴华",30) ;
            System.out.println(rv);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

程序运行结果：

```
李兴华，你好！我今年30岁了！
```

以上程序中，因为 sayHello()方法本身要接收两个参数，所以在使用 getMethod()方法调用时除了要指定调用的方法名称，也同样指定了参数的类型，因为 sayHello()方法调用完之后存在返回值，而且返回值的类型是 String，所以使用了一个字符串接收返回的内容。

15.4.2 调用 setter 及 getter 方法

从面向对象部分开始就一直对读者强调：“类中的属性必须封装，封装之后的属性要通过 setter 及 getter”方法设置和取得，那么在使用反射的调用方法操作中，最重要的是调用类中的 setter 及 getter 方法，这一点在 Java 的开发中随处可见，下面为读者演示如何完成这样的功能。直接调用 Person 类中的 setter 及 getter 方法。

范例：调用 setter 及 getter 方法

```
package org.lxh.demo15.invokedemo;
import java.lang.reflect.Method;
public class InvokeSetGetDemo {
    public static void main(String[] args) {
        Class<?> c1 = null;                                     // 声明Class对象
        Object obj = null;                                      // 声明一个对象
        try {
            c1 = Class.forName("org.lxh.demo15.Person");// 实例化Class对象
        } catch (ClassNotFoundException e) {
```

```
            e.printStackTrace();
        }
        try {
            obj = c1.newInstance();                          // 实例化操作对象
        } catch (InstantiationException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
        setter(obj, "name", "李兴华", String.class);          // 调用setter方法
        setter(obj, "age", 30, int.class);                   // 调用setter方法
        System.out.print("姓名：");
        getter(obj, "name");                                 // 调用getter方法
        System.out.print("年龄：");
        getter(obj, "age");                                  // 调用getter方法
    }
    /**
     * @param obj 操作的对象
     * @param att 操作的属性
     * @param value 设置的值
     * @param type 参数的类型
     */
    public static void setter(Object obj, String att, Object value,
            Class<?> type) {                                 // 调用setter方法
        try {
            Method met = obj.getClass().
                getMethod("set" + initStr(att), type);       // 设置方法参数类型
            met.invoke(obj, value);                          // 调用方法
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    public static void getter(Object obj, String att) {// 调用getter方法
        try {
            Method met = obj.getClass().
                getMethod("get" + initStr(att));             // 此方法不需要参数
            System.out.println(met.invoke(obj));             // 接收方法的返回值
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    public static String initStr(String old) {               // 单词首字母大写
```

```
            String str = old.substring(0, 1).toUpperCase() + old.substring(1);
            return str;
        }
    }
```

程序运行结果：

```
姓名：李兴华
年龄：30
```

以上程序完成了调用类中 setter 及 getter 方法的功能，一些读者可能觉得以上程序有些难以理解，下面分步将本程序的实现思路为读者进行说明。

（1）设置方法名称，在设置方法名称时，本程序直接使用的是属性名称，例如 name 或 age。但是实际上方法名称是 setName()、getName()、setAge()、getAge()，所有属性名称的首字母需要大写，所以为了解决这样的问题，单独设置了一个方法 initStr()，通过此方法将字符串中的首字母大写。首字母大写之后再增加 set 及 get 字符串以找到对应的方法。

（2）调用 setter()方法时，传入了实例化对象、要操作的属性名称（在方法中会将其首字母大写）、要设置的参数内容以及具体的参数类型，这样做是为了满足 getMethod()和 invoke()方法的使用要求。

（3）在调用 getter()方法时，也同样传入了同一个实例化对象，因为其本身不需要任何的接收参数，所以只传入了属性名称（在方法中会将其首字母大写），并在此方法中将内容打印输出。

以上程序是在反射调用方法时的重要应用，读者一定要掌握其原理。因为在以后的开发中，很多系统会为开发者实现好以上的功能。

提示：以上的程序模拟了 JSP 中的指令。

本处只是对相关的原理进行模拟实现，读者在此处只需要了解其基本的操作原理即可。

15.4.3 通过反射操作属性

在反射操作中虽然可以使用 Method 调用类中的 setter 及 getter 方法设置和取得属性，但是这样操作毕竟很麻烦，所以在反射机制中也可以直接通过 Field 类操作类中的属性，通过 Field 类提供的 set()和 get()方法就可以完成设置和取得属性内容的操作。但是在操作前首先需要注意的是，在类中的所有属性已经都设置成私有的访问权限，所以在使用 set()或 get()方法时首先要使用 Field 类中的 setAccessible(true)方法将需要操作的属性设置成可以被外部访问。

范例：直接操作类中的属性

```
package org.lxh.demo15.invokedemo;
import java.lang.reflect.Field;
public class InvokeFieldDemo {
    public static void main(String[] args) throws Exception {
        Class<?> c1 = null;                          // 声明Class对象
```

```
        Object obj = null;                                  // 声明一个对象
        c1 = Class.forName("org.lxh.demo15.Person");// 实例化Class对象
        obj = c1.newInstance() ;                            // 实例化对象
        Field nameField = null ;                            // 表示name属性
        Field ageField = null ;                             // 表示age属性
        nameField = c1.getDeclaredField("name") ;           // 取得name属性
        ageField = c1.getDeclaredField("age") ;             // 取得age属性
        nameField.setAccessible(true) ;                     // 将name属性设置成可被外
                                                               部访问
        nameField.set(obj, "李兴华") ;                      // 设置name属性内容
        ageField.setAccessible(true) ;                      // 将age属性设置成可被外
                                                               部访问
        ageField.set(obj, 30) ;                             // 设置age属性内容
        System.out.println("姓名:" + nameField.get(obj));   // 通过get取得属
                                                               性内容
        System.out.println("年龄:" + ageField.get(obj));    // 通过get取得属
                                                               性内容
    }
}
```

程序运行结果：

```
姓名:李兴华
年龄:30
```

从代码的操作形式上观察，可以非常清楚地发现，明显比之前使用 setter 或 getter 方法操作属性的代码更加简单、方便。

注意：使用反射操作属性时最好通过 setter 及 getter 方法。

以上程序是属于扩大类属性的访问权限后直接操作属性，所以在 Person 类中并不需要编写 setter 和 getter 方法，但是在开发中调用属性时都要使用 setter 及 getter 方法，这一点在之前讲解 private 关键字时已经为读者详细解释过。所以，读者以后在开发时一定注意，不要直接操作属性，而是要通过 setter 及 getter 方法调用。

15.4.4　通过反射操作数组

反射机制不仅只能用在类上，还可以应用在任意的引用数据类型的数据上，当然，这本身就包含了数组，即可以使用反射操作数组。可以通过 Class 类的以下方法取得一个数组的 Class 对象。

```
public Class<?> getComponentType()
```

在反射操作包 java.lang.reflect 中使用 Array 类表示一个数组，可以通过此类取得数组长度，取得数组内容的操作。Array 类的常用方法如表 15-5 所示。

表 15-5 Array 类的常用方法

序号	方 法	类型	描 述
1	public static Object get(Object array,int index) throws IllegalArgumentException,ArrayIndexOutOfBoundsException	普通	根据下标取得数组内容
2	public static Object newInstance(Class<?> componentType,int length) throws NegativeArraySizeException	普通	根据已有的数组类型开辟新的数组对象
3	public static void set(Object array,int index,Object value) throws IllegalArgumentException,ArrayIndexOutOfBoundsException	普通	修改指定位置的内容

下面通过两个范例让读者了解以上方法的使用。

范例：取得数组信息并修改数组内容

```
package org.lxh.demo15.arraydemo;
import java.lang.reflect.Array;
public class ClassArrayDemo {
    public static void main(String[] args) throws Exception {
        int temp[] = {1,2,3} ;                                    // 声明一个整型数组
        Class<?> c = temp.getClass().getComponentType();// 取得数组的Class对象
        System.out.println("类型：" + c.getName());          // 得到数组类型名称
        System.out.println("长度：" + Array.getLength(temp));// 得到数组长度
        System.out.println("第一个内容：" + Array.get(temp,0));// 得到第一个
                                                                  内容
        Array.set(temp, 0, 6) ;                                   // 修改第一个内容
        System.out.println("第一个内容：" + Array.get(temp,0)) ;// 得到第一个
                                                                  内容
    }
}
```

程序运行结果：

```
类型：int
长度：3
第一个内容：1
第一个内容：6
```

以上程序中通过 Array 类取得了数组的相关信息，并通过 Array 类中的 set()方法修改了数组中的元素内容。

在应用中还可以通过 Array 类根据已有的数组类型来开辟新的数组对象，下面就使用 Array 完成一个可以修改已有数组大小的功能。

范例：修改数组的大小

```
package org.lxh.demo15.arraydemo;
import java.lang.reflect.Array;
public class ChangeArrayDemo {
```

```
    public static void main(String[] args) throws Exception {
        int temp[] = {1,2,3} ;                                  // 声明一个整型数组
        int newTemp[] = (int[])arrayInc(temp,5) ;               // 扩大数组长度
        print(newTemp) ;                                        // 打印数组信息
        System.out.println("\n------------------------------") ;
        String t[] = { "lxh", "mldn", "mldnjava" };             // 声明一个字符串数组
        String nt[] = (String[])arrayInc(t,8) ;                 // 扩大数组长度
        print(nt) ;                                             // 打印数组信息
    }
    public static Object arrayInc(Object obj,int len){ // 修改数组大小
        Class<?> c = obj.getClass() ;                           // 通过数组得到Class
                                                                   对象
        Class<?> arr = c.getComponentType() ;                   // 得到数组的Class
                                                                   对象
        Object newO = Array.newInstance(arr, len) ;             // 重新开辟新的数组
                                                                   大小
        int co = Array.getLength(obj) ;                         // 取得数组长度
        System.arraycopy(obj,0,newO,0,co) ;                     // 复制数组内容
        return newO ;
    }
    public static void print(Object obj){                       // 输出数组
        Class<?> c = obj.getClass() ;                           // 通过数组得到Class
                                                                   对象
        if(!c.isArray()){                                       // 判断是否是数组
            return ;                                            // 不是则返回
        }
        Class<?> arr = c.getComponentType() ;                   // 取得数组的Class
        System.out.println(arr.getName()
                + "数组的长度是:" + Array.getLength(obj));// 输出数组信息
        for (int i = 0; i < Array.getLength(obj); i++){ // 循环输出
            System.out.print(Array.get(obj, i) + "、");// 通过Array输出
        }
    }
}
```

程序运行结果：

```
int数组的长度是:5
1、2、3、0、0、
--------------------------------
java.lang.String数组的长度是:8
lxh、mldn、mldnjava、null、null、null、null、null、
```

15.5 动 态 代 理

之前曾为读者讲解过代理机制的操作，但是所讲解的代理设计属于静态代理，因为每一个代理类只能为一个接口服务，这样一来程序开发中必然会产生过多的代理。最好的做法是可以通过一个代理类完成全部的代理功能，那么此时就必须使用动态代理完成。

在 Java 中要想实现动态代理机制，则需要 java.lang.reflect.InvocationHandler 接口和 java.lang.reflect.Proxy 类的支持。

InvocationHandler 接口的定义如下：

```
public interface InvocationHandler{
    public Object invoke(Object proxy,Method method,Object[] args) throws
Throwable
}
```

在此接口中只定义了一个 invoke()方法，此方法中有 3 个参数，其参数的意义如下。

- Object proxy：被代理的对象。
- Method method：要调用的方法。
- Object args[]：方法调用时所需要的参数。

Proxy 类是专门完成代理的操作类，可以通过此类为一个或多个接口动态地生成实现类。Proxy 类提供了如下的操作方法：

```
public static Object newProxyInstance(ClassLoader loader,Class<?>[]
interfaces,
        InvocationHandler h) throws IllegalArgumentException
```

通过 newProxyInstance()方法可以动态地生成实现类，在此方法中的参数意义如下。

- ClassLoader loader：类加载器。
- Class<?>[] interfaces：得到全部的接口。
- InvocationHandler h：得到 InvocationHandler 接口的子类实例。

提示：类加载器。

在 Proxy 类的 newProxyInstance()方法中需要一个 ClassLoader 类的实例，ClassLoader 实际上对应的是类加载器，在 Java 中主要有以下 3 种类加载器。

- Bootstrap ClassLoader：此加载器采用 C++编写，一般开发中是看不到的。
- Extension ClassLoader：用来进行扩展类的加载，一般对应的是 jre\lib\ext 目录中的类。
- AppClassLoader：加载 classpath 指定的类，是最常使用的一种加载器。

范例：取得类加载器

```
package org.lxh.demo15.classloaderdemo;
class Person {}                                          // 定义Person类
public class ClassLoaderDemo {
    public static void main(String[] args) {
```

```
        Person stu = new Person() ;                    // 实例化子类对象
        System.out.println("类加载器: "
                + stu.getClass().getClassLoader().getClass().getName());
    }
}
```

程序运行结果:

```
类加载器: sun.misc.Launcher$AppClassLoader
```

从程序的运行结果中可以发现，默认的 ClassLoader 就是 AppClassLoader。在开发中用户也可以通过继承 ClassLoader 类来实现自己的类加载器，但这样做的意义不大，所以本书对此部分不作深入介绍。

如果要完成动态代理，首先定义一个 InvocationHandler 接口的子类，以完成代理的具体操作。

范例：定义 MyInvocationHandler 的类

```
package org.lxh.demo15.dynaproxydemo;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
public class MyInvocationHandler implements InvocationHandler {
    private Object obj;                                    // 真实主题
    public Object bind(Object obj) {                       // 绑定真实操作主题
        this.obj = obj;
        return Proxy.newProxyInstance(obj.getClass().getClassLoader(),
                obj.getClass().getInterfaces(), this); // 取得代理对象
    }
    public Object invoke(Object proxy, Method method, Object[] args)
            throws Throwable {                             // 动态调用方法
        Object temp = method.invoke(this.obj, args);       // 调用方法，传入真实
                                                           主题和参数
        return temp;                                       // 返回方法的返回信息
    }
}
```

在 MyInvocationHandler 类的 bind()方法中接受被代理对象的真实主题实现，之后覆写 InvocationHandler 接口中的 invoke()方法，完成具体的方法调用。

范例：定义接口

```
package org.lxh.demo15.dynaproxydemo;
public interface Subject{                                  // 定义Subject接口
    public String say(String name,int age) ;               // 定义抽象方法say
}
```

范例：定义真实主题实现类

```
package org.lxh.demo15.dynaproxydemo;
public class RealSubject implements Subject{           // 真实实现类
    public String say(String name, int age) {          // 覆写say()方法
        return "姓名：" + name + "，年龄：" + age;       // 返回信息
    }
}
```

以上定义了接口及真实主题类，这样在操作时直接将真实主题类的对象传入到 MyInvocationHandler 类的 bind()方法中即可。

范例：测试动态代理

```
package org.lxh.demo15.dynaproxydemo;
public class DynaProxyDemo {
    public static void main(String[] args) {
        MyInvocationHandler handler = new MyInvocationHandler();
                                                       // 实例化代理操作类
        Subject sub = (Subject) handler.bind(new RealSubject());
                                                       // 绑定对象
        String info = sub.say("李兴华", 30);           // 通过动态代理调用方法
        System.out.println(info);                      // 信息输出
    }
}
```

程序运行结果：

```
姓名：李兴华，年龄：30
```

从程序的运行结果中可以发现，完成的功能与之前的静态代理操作没什么不同，所以在一般的开发中很少会使用到这种动态代理机制，但是在编写一些底层代码或使用一些框架（如 Spring Framework）时这种动态代理模式就比较常用了。

15.6 类的生命周期

在一个类编译完成之后，下一步就要开始使用类，如果要使用一个类，肯定离不开 JVM。在程序执行中 JVM 通过装载、链接、初始化 3 个步骤完成，类的装载就是通过类加载器把.class 二进制文件装入 JVM 的方法区，并在堆区创建描述该类的 java.lang.Class 对象，用来封装数据。需要提醒读者的是，同一个类只会被 JVM 加载一次。链接就是把二进制数据组装成可以运行的状态。

链接分为校验、准备和解析 3 个步骤。校验用来确认此二进制文件是否适合当前的 JVM（版本）；准备就是为静态成员分配内存空间，并设置默认值；解析指的是转换常量池的代码引用为直接引用的过程，直到所有的符号引用都可被运行程序使用（建立完整的对应关系）。完成之后，类型即可初始化，初始化之后类的对象就可以正常地使用，直到一个

对象不再使用之后，将被垃圾回收，释放空间。当没有任何引用指向 Class 对象时将会被卸载，结束类的生命周期，以上所讲解的生命周期如图 15-5 所示。

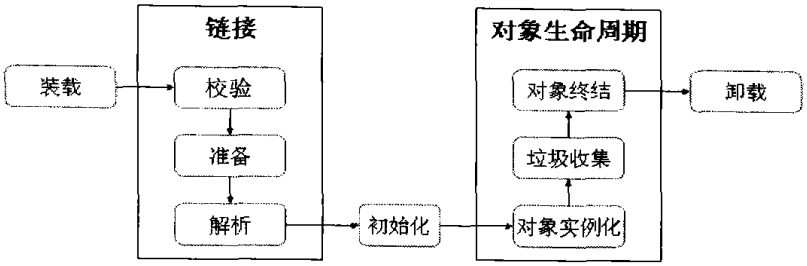


图 15-5　类及对象的生命周期

15.7　工厂设计模式

15.7.1　将反射应用在工厂模式上

工厂设计模式在实际的开发中使用得非常多，之前读者已经学习过简单的工厂模式，通过简单的工厂设计模式可以达到类的解耦合目的，但是之前的工厂设计模式依然存在问题，那就是在增加一个子类时都需要修改工厂类，这样肯定会非常麻烦。学习完反射机制之后，实际上，此时就可以通过反射机制来改善工厂类，让其在增加子类时可以不用做任何的修改，就可以达到功能的扩充，如下所示。

范例：使用反射完成工厂设计

```
package org.lxh.demo15.factorydemo01;
interface Fruit {                                          // 水果接口
    public void eat();                                     // 吃水果
}
class Apple implements Fruit {                             // 定义苹果
    public void eat() {                                    // 覆写抽象方法
        System.out.println("** 吃苹果。");
    }
}
class Orange implements Fruit {                            // 定义橘子
    public void eat() {                                    // 覆写抽象方法
        System.out.println("** 吃橘子");
    }
}
class Factory {
    public static Fruit getInstance(String className) {    // 取得接口实例
        Fruit fruit = null;                                // 定义接口对象
        try {
            fruit = (Fruit) Class.forName(className)
```

```
                .newInstance();                                    // 实例化对象
            } catch (Exception e) {
                e.printStackTrace();
            }
            return fruit;
        }
    }
    public class FactoryDemo01 {
        public static void main(String[] args) {
            // 通过工厂类取得接口实例，传入完整的包.类名称
            Fruit f =
Factory.getInstance("org.lxh.demo15.factorydemo01.Apple");
            if (f != null) {                                       // 判断是否取得接口实例
                f.eat();                                           // 调用方法
            }
        }
    }
```

程序运行结果：

```
** 吃苹果。
```

以上的工厂操作类中使用了反射操作取得 Fruit 实例，这样无论增加多少个子类，工厂类都不用做任何的修改。

15.7.2 结合属性文件的工厂模式

以上操作代码虽然可以通过反射取得接口的实例，但是在操作时需要传入完整的包.类名称，而且用户也无法知道一个接口有多少个可以使用的子类，所以此时可以通过属性文件的形式配置所要的子类信息。

范例：属性文件 fruit.properties

```
apple = org.lxh.demo15.factorydemo02.Apple
orange = org.lxh.demo15.factorydemo02.Orange
```

在属性文件中使用 apple 和 orange 表示完整的包.类名称，这样在使用时直接通过属性名称即可。

范例：属性操作类

```
class Init{                                                        // 定义初始化操作类
    public static Properties getPro(){
        Properties pro = new Properties() ;                        // 实例化属性类
        File f = new File("d:\\fruit.properties") ;// 找到属性文件
        try{
            if(f.exists()){                                        // 属性文件中已存在
                pro.load(new FileInputStream(f)) ; // 读取属性
```

```
            }else{                          // 建立一个新的属性文件，同时设置好默认内容
                pro.setProperty("apple","org.lxh.demo15.factorydemo02.
                Apple") ;
                pro.setProperty("orange","org.lxh.demo15.factorydemo02.
                Orange") ;
                pro.store(new FileOutputStream(f),"FRUIT CLASS") ;
            }
        }catch(Exception e){
            e.printStackTrace() ;
        }
        return pro ;
    }
};
```

此类中的主要功能是取得属性文件中的配置信息，如果属性文件不存在，则创建一个新的，并设置默认值。

范例：测试程序

```
public class FactoryDemo02 {
    public static void main(String[] args) {
        Properties pro = Init.getPro() ;                    // 初始化属性类
        // 通过工厂类取得接口实例，通过属性的key传入完整的包.类名称
        Fruit f = Factory.getInstance(pro.getProperty("apple"));
        if (f != null) {                                    // 判断是否取得接口实例
            f.eat();                                        // 调用方法
        }
    }
}
```

在通过工厂类取得接口实例时，直接输入属性的 key 就可以找到其完整的包.类名称，以达到对象的实例化功能。

在本程序中可以发现，程序很好地实现了代码与配置的分离。通过配置文件配置要使用的类，之后通过程序读取配置文件，以完成具体的功能，如图 15-6 所示。当然，这些程序完成的前提是基于接口，所以接口在实际的开发中用处是最大的。

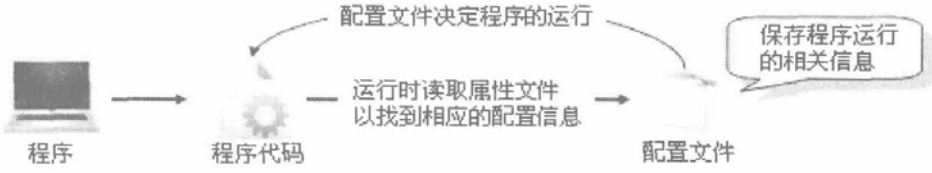


图 15-6　代码与文件相分离

以上程序是读者日后学习高级程序开发的一个基础，通过本程序要更好地理解配置文件在程序开发中的作用，这样就能编写出更好的程序代码。

15.8 本章要点

1．Class 类是反射机制操作的源头。

2．Class 类的对象有 3 种实例化方式：

（1）通过 Object 类中的 getClass()方法。

（2）通过“类.class”的形式。

（3）通过 Class.forName()方法，此种方式最为常用。

3．可以通过 Class 类中的 newInstance()方法进行对象的实例化操作，但是要求类中必须存在无参构造方法，如果类中没有无参构造，则必须使用 Constructor 类完成对象的实例化操作。

4．可以通过反射取得一个类所继承的父类、实现的接口、类中的全部构造方法、全部普通方法及全部属性。

5．使用反射机制可以通过 Method 调用类中的方法，也可以直接操作类中的属性。

6．动态代理可以解决开发中代理类过多的问题，提供统一的代理功能实现。

7．在程序的开发中使用反射机制并结合属性文件，可以达到程序代码与配置文件相分离的目的。

15.9 习　　题

定义一个学生类，其中包含姓名、年龄、成绩的属性，之后由键盘输入学生的内容，并将内容保存在文件中，所有的操作要求全部使用反射机制完成，即不能使用通过关键字 new 创建学生类对象的操作。

第 16 章　Annotation

通过本章的学习可以达到以下目标：

- ☑ 了解 Annotation 的作用。
- ☑ 掌握 JDK 1.5 中内建的 3 种 Annotation 的使用。
- ☑ 掌握自定义 Annotation 的语法及其应用。
- ☑ 掌握@Retention、@Target、@Documented、@Inherited 注释。

Annotation 是 Java 中的重要特性，此技术已经在 EJB 3.0 中大量地被采用。本章将 Java 中提供的 3 种内建 Annotation 及自定义的 Annotation 进行讲解，并在最后结合 Java 的反射机制让读者理解 Annotation 的实际作用。**本章视频录像讲解时间为 1 小时 19 分钟，源代码在光盘对应的章节下。**

16.1　Annotation 简介

J2SE 5.0 提供了很多新的特性。其中一个很重要的特性就是对元数据（Metadata）的支持。在 J2SE 5.0 中，这种元数据称为注释（Annotation）。通过使用注释，程序开发人员可以在不改变原有逻辑的情况下，在源文件嵌入一些补充的信息。

Annotation 可以用来修饰类、属性、方法，而且 Annotation 不影响程序运行，无论是否使用 Annotation 代码都可以正常执行。

java.lang.annotation.Annotation 是 Annotation 的接口，只要是 Annotation 都必须实现此接口。此接口定义如下：

```
public interface Annotation{
    public Class<? extends Annotation> annotationType() ;
                                                    // 返回此annotation 的注释类型
    public boolean equals(Object obj) ;
    public int hashCode() ;
    String toString() ;
}
```

本章以后所介绍的系统内建的 Annotation 和用户自定义的 Annotation 都是默认实现了此接口。

16.2　系统内建的 Annotation

在 JDK 1.5 之后，系统已经建立了如下 3 个内建的 Annotation 类型，用户直接使用即可。

- @Override：覆写的 Annotation。
- @Deprecated：不赞成使用的 Annotation。
- @SuppressWarnings：压制安全警告的 Annotation。

提示：3 个 Annotation 在 java.lang 包中定义。

以上 3 个内建的 Annotation 全部是在 java.lang 包中定义的。因为此包在使用时是自动导入的，所以可以直接使用以上 3 个 Annotation。

以上 3 个 Annotation 都是 java.lang.annotation.Annotation 接口的子类。在 Java 中都有各自的定义，如表 16-1 所示。

表 16-1　3 种内定的 Annotation

序号	Annotation	Java 中的声明
1	@Override	@Target(value=METHOD) @Retention(value=SOURCE) public @interface Override
2	@Deprecated	@Documented @Retention(value=RUNTIME) public @interface Deprecated
3	@SuppressWarnings	@Target(value={TYPE,FIELD,METHOD,PARAMETER,CONSTRUCTOR, LOCAL_VARIABLE}) @Retention(value=SOURCE) public @interface SuppressWarnings

以上的 Target 和 Retention 在本章后面会有介绍，下面分别观察以上 3 种 Annotation 的使用。

16.2.1　@Override

@Override 主要是在方法覆写时使用，用于保证方法覆写的正确性。下面介绍此注释的基本使用。

范例：观察@Override 注释的作用

```
package org.lxh.demo16.systemannotation;
class Person{                                          // 定义Person类
    public String getInfo(){                           // 定义getInfo()方法
        return "这是一个Person类。" ;                    // 返回信息
    }
}
class Student extends Person{                          // 子类继承父类
    @Override                                          // 此处明确地指出方法覆写操作
    public String getInfo() {                          // 覆写父类的方法
        return "这是一个Student类。";
```

```
    }
}
public class OverrideAnnotationDemo01 {
    public static void main(String[] args) {
        Person per = new Student() ;                    // 通过子类实例化父类对象
        System.out.println(per.getInfo()) ;             // 输出信息
    }
}
```

程序运行结果：

```
这是一个Student类。
```

以上程序中的子类 Student 继承 Person 类，之后覆写了 Person 类中的 getInfo()方法，程序的运行结果与之前一样，唯一不同的只是在覆写的 getInfo()方法前加上了@Override 注释，这样做的目的主要是防止用户在覆写方法时将方法定义出错。下面演示了一个方法覆写错误的程序，读者可以通过程序的编译自行观察其错误提示。

范例：错误的覆写

```
package org.lxh.demo16.systemannotation;
class Person{                                           // 定义Person类
    public String getInfo(){                            // 定义getInfo()方法
        return "这是一个Person类。" ;                    // 返回信息
    }
}
class Student extends Person{                           // 子类继承父类
    @Override                                           // 此处明确地指出方法覆写操作
    public String getinfo() {                           // 此处将方法名称写错
        return "这是一个Student类。";
    }
}
```

上面的子类 Student 中原本是要覆写 Person 类中的方法，但是由于在编写方法名称时出现了大小写的问题，所以以上程序在编译时，将出现以下错误：

```
OverrideAnnotationDemo02.java:8: method does not override or implement a
method from a supertype
        @Override                   // 此处明确地指出方法覆写操作
        ^
1 error
```

编译时，会在出错的地方给出指示，因为 getInfo()和 getinfo()是两个方法，所以使用了@Override 注释，可以确保方法被正确覆写。

注意：@Override 使用限制。

@Override 在使用时只能在方法上应用，而其他元素，如类、属性等上是不能使用此 Annotation 的。

16.2.2 @Deprecated

@Deprecated 注释的主要功能是用来声明一个不建议使用的方法。如果在程序中使用了此方法，则在编译时将出现警告信息。

范例：使用@Deprecated 声明一个不建议使用的方法

```
package org.lxh.demo16.systemannotation;
class Demo{                                          // 定义Demo类
    @Deprecated                                      // 声明不建议使用的操作
    public String getInfo(){                         // 此方法不建议用户使用
        return "这是一个Person类。" ;                  // 返回信息
    }
}
public class DeprecatedAnnotationDemo01 {
    public static void main(String[] args) {
        Demo d = new Demo() ;                        // 实例化Demo对象
        System.out.println(d.getInfo()) ;            // 编译时，将出现警告信息
    }
}
```

以上的 Demo 类中的 getInfo()方法上使用了@Deprecated 注释声明，这就表示此方法不建议用户继续使用，所以在编译时将出现以下的警告信息。

```
Note: DeprecatedAnnotationDemo01.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
```

虽然以上有警告信息，但是程序依然可以正常执行。因为该注释只是表示方法不建议使用，但并不是不能使用。

使用@Deprecated 除了可以在方法上声明外，在类中也可以进行声明，如下所示。

范例：在类声明中使用@Deprecated 注释

```
package org.lxh.demo16.systemannotation;
@Deprecated                                          // 定义不建议的操作
class Demo{                                          // 此类不建议用户使用
    public String getInfo(){                         // 取得信息
        return "这是一个Person类。" ;                  // 返回信息
    }
}
public class DeprecatedAnnotationDemo02 {
    public static void main(String[] args) {
        Demo d = new Demo() ;                        // 编译时，将出现警告信息
        System.out.println(d.getInfo()) ;            // 输出信息
    }
}
```

编译时警告：

```
Note: DeprecatedAnnotationDemo02.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
```

以上程序在编译时同样会出现警告信息，因为 Demo 类不建议使用。

提示：Thread 类中的@Deprecated 声明。

在 Thread 类中曾经为读者介绍过 suspend()、stop()、resume() 3 个方法，不建议使用，实际上这 3 个方法就是使用了@Deprecated 注释声明的。读者可以通过 JDK 安装目录下的 src.zip 查看 Thread 类的定义。

16.2.3 @SuppressWarnings

@SuppressWarnings 注释的主要功能是用来压制警告，例如，之前讲解泛型操作时，如果在一个类声明时没有指明泛型，则肯定在编译时产生，那么此时就可以使用@SuppressWarnings 压制住这种警告，如下面代码所示。

范例：压制一个警告

```
package org.lxh.demo16.systemannotation;
class Demo<T>{                                          // 定义Demo类，使用泛型
    private T var ;                                     // 定义泛型变量
    public T getVar() {                                 // 取得泛型变量的内容
        return var;
    }
    public void setVar(T var) {                         // 设置泛型变量
        this.var = var;
    }
}
public class SuppressWarningsAnnotationDemo01 {
    @SuppressWarnings("unchecked")                      // 压制主方法的警告信息
    public static void main(String[] args) {
        Demo d = new Demo() ;                           // 编译时，将出现警告信息，
                                                        但此时警告被压制
        d.setVar("李兴华") ;
        System.out.println("内容: "+d.getVar()) ;       // 输出
    }
}
```

以上程序在声明 Demo 对象时，并没有指定具体的泛型类型，那么如果按照之前的方式使用，则在编译时肯定会出现警告信息。但是由于使用了@SuppressWarnings 注释，所以此警告信息将不会出现。在@SuppressWarnings 注释中的 unchecked，表示的是不检查，当然，如果现在需要压制更多的警告信息，则可以在后面继续增加字符串，只是在增加时，要按照数组的格式增加。

范例：压制多个警告

```
package org.lxh.demo16.systemannotation;
@Deprecated                                          // 以下操作不建议使用
class Demo<T>{                                       // 定义Demo类，使用泛型
    private T var ;                                  // 定义泛型变量
    public T getVar() {                              // 取得泛型变量的内容
        return var;
    }
    public void setVar(T var) {                      // 设置泛型变量
        this.var = var;
    }
}
public class SuppressWarningsAnnotationDemo02 {
    @SuppressWarnings({ "unchecked", "deprecation"})// 此时压制两条警告
    public static void main(String[] args) {
        Demo d = new Demo() ;                        // 编译时的警告信息将被压制
        d.setVar("李兴华") ;
        System.out.println("内容: "+d.getVar()) ;    // 输出
    }
}
```

以上程序同时存在了泛型和不建议使用方法两种警告信息，但是由于使用了@SuppressWarning 注释，所以此时在程序编译时将不会出现任何的警告信息。

以上在压制多个警告时，使用了 unchecked 和 deprecation 两种关键字，是以字符串数组的形式设置进行声明的。@SuppressWarnings 中的关键字如表 16-2 所示。

表 16-2　@SuppressWarnings 中的关键字

序号	关 键 字	描　　述
1	deprecation	使用了不赞成使用的类或方法时的警告
2	unchecked	执行了未检查的转换时警告，例如，泛型操作中没有指定泛型类型
3	fallthrough	当使用 switch 操作时 case 后未加入 break 操作，而导致程序继续执行其他 case 语句时出现的警告
4	path	当设置了一个错误的类路径、源文件路径时出现的警告
5	serial	当在可序列化的类上缺少 serialVersionUID 定义时的警告
6	finally	任何 finally 子句不能正常完成时的警告
7	all	关于以上所有情况的警告

另外，这里还要提醒读者的是，在设置注释信息时，是以 key→value 的形式出现的，所以以上的@SuppressWarnings 也可以直接使用“value={"unchecked","deprecation"}”的方式设置，如下所示。

范例：另一种形式的@SuppressWarnings

```
package org.lxh.demo16.systemannotation;
@Deprecated                                          // 以下操作不建议使用
```

```
class Demo<T>{                                          // 定义Demo类，使用泛型
    private T var ;                                     // 定义泛型变量
    public T getVar() {                                 // 取得泛型变量的内容
        return var;
    }
    public void setVar(T var) {                         // 设置泛型变量
        this.var = var;
    }
}
public class SuppressWarningsAnnotationDemo03 {
    @SuppressWarnings(value = { "unchecked", "deprecation" }) // 此时压制
                                                              两条警告
    public static void main(String[] args) {
        Demo d = new Demo() ;                           // 编译时的警告信息被压制
        d.setVar("李兴华") ;
        System.out.println("内容: "+d.getVar()) ;
    }
}
```

16.3　自定义 Annotation

之前已经介绍了 3 种 Java 中自建的 Annotation，只要通过固定的格式调用即可。这里提醒读者的是，要想充分地理解 Annotation 的作用必须结合之前讲解过的枚举、反射等机制才可以，所以，以下只介绍了如何定义 Annotation。具体的使用在本章的最后将为读者讲解。

16.3.1　Annotation 的定义格式

1. 定义简单的 Annotation

在定义 Annotation 时，可以采用格式 16-1 的形式进行定义。

【格式 16-1　Annotation 定义】

```
[public] @interface Annotation名称{
    数据类型 变量名称() ;
}
```

要定义 Annotation 必须使用@interface 的方式进行定义，但是从格式 16-1 中可以发现，在定义 Annotation 时也可以定义各种变量，但是变量定义之后必须使用“()”。

提示：使用@interface 就相当于继承了 Annotation 接口。

在程序中只要使用了@interface 声明 Annotation，那么此 Annotation 实际上就相当于继承了 java.lang.annotation.Annotation 接口。

范例：自定义一个 Annotation

```
package org.lxh.demo16.defaultannotation;
public @interface MyDefaultAnnotationNoneParam{        // 定义Annotation
}
```

以上就定义完了一个 Annotation，定义的名称为 MyDefaultAnnotationNoneParam，其中没有定义任何的变量。那么此时该如何使用呢？要想使用此 Annotation，需要定义其他的类，并使用“@Annotation 名称”的形式。

范例：使用 Annotation

```
package org.lxh.demo16.defaultannotation;
@MyDefaultAnnotationNoneParam                    // 使用自定义的Annotation
class Demo {
}
```

以上的 Demo 类的声明处就使用了之前定义好的 MyDefaultAnnotationNoneParam 注释。

2. 向 Annotation 中设置内容

回顾一下之前在讲解@SuppressWarnings 注释时不是可以向 Annotation 中传递吗？所以此时也可以向自定义的 Annotation 中传递参数，在 Annotation 中必须使用变量接收参数。

范例：定义 MyDefaultAnnotationSingleParam，可以接收一个变量

```
package org.lxh.demo16.defaultannotation;
public @interface MyDefaultAnnotationSingleParam{      // 定义Annotation
    public String value() ;                            // 接收设置的内容
}
```

在以上的 Annotation 中定义了一个 value，以后在使用此 Annotation 时，可以将内容设置给 value。

范例：向 Annotation 中设置内容

```
package org.lxh.demo16.defaultannotation;
@MyDefaultAnnotationSingleParam("李兴华")        // 使用自定义的Annotation
class Demo {
}
```

以上程序在使用 Annotation 时，向其中设置了一个内容，使用 MyDefaultAnnotationSingleParam 中定义的 value 属性进行接收。在使用 MyDefaultAnnotationSingleParam 时也可以直接指定接收的属性名称，如下所示。

范例：指定接收的参数的变量

```
package org.lxh.demo16.defaultannotation;
@MyDefaultAnnotationSingleParam(value = "李兴华") // 使用自定义的Annotation
class Demo {
}
```

以上形式一般只有向一个 Annotation 中指定多个属性时才会使用，如下所示。

范例：在 Annotation 中设置多个属性

```
package org.lxh.demo16.defaultannotation;
public @interface MyDefaultAnnotationMoreParam{          // 定义Annotation
    public String key() ;                                // 接收设置的内容
    public String value() ;                              // 接收设置的内容
}
```

以上的 Annotation 中可以设置两个内容：key 和 value。所以在使用 Annotation 时就可以明确地指定出变量的名称。

范例：分别设置内容

```
package org.lxh.demo16.defaultannotation;
@MyDefaultAnnotationMoreParam(key = "MLDN", value = "李兴华")// 使用自定义的
                                                              Annotation
class Demo {
}
```

但在现在定义的 Annotation 中，每次只能向 Annotation 中的属性传递一个内容，如果要为一个属性设置多个内容，则必须可以将一个属性定义成一个数组。

范例：建立 Annotation 并设置数组属性

```
package org.lxh.demo16.defaultannotation;
public @interface MyDefaultAnnotationArrayParam{    // 定义Annotation
    public String[] value() ;                       // 接收设置的内容是一个字
                                                       符串数组
}
```

范例：调用 Annotation，设置一个数组

```
package org.lxh.demo16.defaultannotation;
@MyDefaultAnnotationArrayParam(value = { "MLDN", "李兴华" })// 使用自定义的
                                                              Annotation
class Demo {
}
```

在以上操作中，所有内容都是需要用户在调用 Annotation 时手工设置的。当然，为了方便用户，有时也可以通过设置默认值的方式减少用户设置的内容。

3. 默认值

如果在一个定义好的 Annotation 中已经定义了若干个属性，但是在使用 Annotation 时并没有指定其具体的内容，则在编译时也会出现错误。

范例：有如下的 Annotation

```
package org.lxh.demo16.defaultannotation;
public @interface MyDefaultAnnotationValue{          // 定义Annotation
    public String key() ;                            // 接收设置的内容
```

```
    public String value() ;                          // 接收设置的内容
}
```

范例：错误地使用 Annotation

```
package org.lxh.demo16.defaultannotation;
@MyDefaultAnnotationValue                            // 使用自定义的Annotation
class Demo {
}
```

编译时出现以下错误：

```
UseDefaultAnnotationValue01.java:2: annotation org.lxh.demo16.
defaultannotation.MyDefaultAnnotationValue is missing value
@MyDefaultAnnotationValue                    // 使用自定义的Annotation
^
1 error
```

也就是说，只要在定义 Annotation 时设置了属性，就必须在使用时设置其内容。当然，有时为了方便用户使用，也可以在定义 Annotation 属性时指定其默认值，此时定义格式如格式 16-2 所示。

【格式 16-2　Annotation 定义变量的默认值】

```
[public] @interface Annotation名称{
    数据类型 变量名称() default 默认值;
}
```

下面使用以上格式定义一个存在默认值的 Annotation。

```
package org.lxh.demo16.defaultannotation;
public @interface MyDefaultAnnotationValue {          // 定义Annotation
    public String key() default "MLDN";               // 接收设置的内容
    public String value() default "李兴华";           // 接收设置的内容
}
```

这样，以后在使用 MyDefaultAnnotationValue 时，如果没有设置内容，则会将默认值赋给属性，而如果已经明确地给出了内容，则将给出的内容赋给属性。

4．使用枚举限制设置的内容

在 Annotation 中也可以通过枚举来限制 Annotation 的取值范围，例如，下面定义了一个 MyName 的 Enum 类型。

范例：定义一个 MyName 的枚举

```
package org.lxh.demo16;
public enum MyName {                                  // 定义枚举，其中包括3个内容
    MLDN, LXH, Li ;
}
```

再定义 MyDefaultAnnotationEnum，此 Annotation 的取值必须是 MyName 中的枚举类型。

范例：定义 MyDefaultAnnotationEnum

```
package org.lxh.demo16.defaultannotation;
public @interface MyDefaultAnnotationEnum {     // 定义Annotation
    public MyName name() default MyName.LXH ;   // 只能设置枚举中的取值，默认值
                                                   为Enum中的内容
}
```

这样，以后在使用 MyDefaultAnnotation 时，所有的取值就必须从 MyName 这个枚举中取得。

范例：使用 MyDefaultAnnotation

```
package org.lxh.demo16.defaultannotation;
@MyDefaultAnnotationEnum(name = MyName.Li)      // 使用自定义的Annotation
class Demo {
}
```

使用枚举限制 Annotation 取值范围，这一点在以后的部分也会经常看到。

16.3.2　Retention 和 RetentionPolicy

在 Annotation 中，可以使用 Retention 定义一个 Annotation 的保存范围。此 Annotation 的定义如下：

```
@Documented
@Retention(value=RUNTIME)
@Target(value=ANNOTATION_TYPE)
public @interface Retention{
    RetentionPolicy value() ;
}
```

在以上的 Retention 定义中存在一个 RetentionPolicy 的变量，此变量用于指定 Annotation 的保存范围。RetentionPolicy 包含如表 16-3 所示的 3 个范围。

表 16-3　RetentionPolicy 的 3 个范围

序号	范　围	描　述
1	SOURCE	此 Annotation 类型的信息只会保留在程序源文件中（*.java），编译之后不会保存在编译好的类文件（*.class）中
2	CLASS	此 Annotation 类型将保留在程序源文件（*.java）和编译之后的类文件（*.class）中。在使用此类时，这些 Annotation 信息将不会被加载到虚拟机（JVM）中，如果一个 Annotation 声明时没有指定范围，则默认是此范围
3	RUNTIME	此 Annotation 类型的信息保留在源文件（*.java）、类文件（*.class）中，在执行时也会加载到 JVM 中

下面使用以上 Retention 指定一个在 RUNTIME 范围有效的 Annotation。读者可以回顾一下之前讲解的 3 个内建的 Annotation 的定义：

- Override 定义采用的是@Retention(value=SOURCE)，只能在源文件中出现。
- Deprecated 定义采用的是@Retention(value=RUNTIME)，可以在执行时出现。
- SuppressWarnings 定义采用的是@Retention(value=SOURCE)，只能在源文件中出现。

范例：定义在 RUNTIME 范围有效的 Annotation

```
package org.lxh.demo16.retentiondemo;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
@Retention(value=RetentionPolicy.RUNTIME)  // 此Annotation在执行时起作用
public @interface MyDefaultRetentionAnnotation {
    public String name() default "李兴华" ; // 只能设置枚举中的取值
}
```

以上定义的 Annotation 在程序执行时起作用，这是一种比较常见的使用方式，而如果此时将其设置成其他范围，则以后在 Annotation 的应用中肯定是无法访问到的。要想让一个 Annotation 起作用，则必须结合 Java 中的反射机制。

16.4 通过反射取得 Annotation

如果要让一个 Annotation 起作用，则必须结合反射机制。在 Class 类中存在以下几种与 Annotation 操作有关的方法，如表 16-4 所示。

表 16-4 与 Annotation 有关的操作

序号	方 法	类型	描 述
1	public <A extends Annotation> A getAnnotation(Class<A> annotationClass)	普通	如果在一个元素中存在注释，则取得全部注释
2	public Annotation[] getAnnotations()	普通	返回此元素上的所有注释
3	public Annotation[] getDeclaredAnnotations()	普通	返回直接存放在此元素上的所有注释
4	public boolean isAnnotation()	普通	判断元素是否表示一个注释
5	public boolean isAnnotationPresent(Class<? extends Annotation> annotationClass)	普通	判断一个元素上是否存在注释

下面通过一些实例来观察以上方法的使用。

16.4.1 范例——取得全部的 Annotation

为了方便操作，以下程序首先定义了一个类，并在类中的方法上使用了系统内建好的 3 个 Annotation。

范例：定义一个简单类

```
package org.lxh.demo16.reflectannotation;
public class SimpleBeanOne {
```

```
    @SuppressWarnings("unchecked")    // 使用@SuppressWarnings的Annotation
    @Deprecated                       // 使用@Deprecated的Annotation
    @Override                         // 使用@Override的Annotation
    public String toString() {        // 覆写toString()方法
        return "Hello LiXingHua!!!" ;// 返回信息
    }
}
```

以上程序覆写的 toString()方法中使用了 3 个内建的 Annotation，读者无须讨论此做法是否有意义，因为本部分只是利用以上的类经过反射取得 toString()方法上定义的全部 Annotation。

要想取得 SimpleBeanOne 中 toString()方法的全部 Annotation，则必须首先通过反射找到 toString()方法。

范例：取得全部的 Annotation

```
package org.lxh.demo16.reflectannotation;
import java.lang.annotation.Annotation;
import java.lang.reflect.Method;
public class ReflectDemo01 {
    public static void main(String args[]) throws Exception {
                                                        // 所有异常抛出
        Class<?> c = null ;                             // 取得Class实例
        c = Class.forName("org.lxh.demo16.reflectannotation.SimpleBeanOne") ;
        Method toM = c.getMethod("toString");           // 取得toString()方法
        Annotation an[] = toM.getAnnotations();         // 取得全部的Annotation
        for (Annotation a : an) {                       // foreach输出
            System.out.println(a);                      // 直接输出信息
        }
    }
}
```

程序运行结果：

```
@java.lang.Deprecated()
```

SimpleBeanOne 的 toString()方法上虽然使用了 3 个 Annotation 注释，但是最后真正得到的只能是一个，这是因为只有 Deprecated 使用了 RUNTIME 的方式声明，所以只有它可以取得。

16.4.2　范例——取得指定的 Annotation 中的内容

以上程序是取得了一个元素所声明的全部 RUNTIME 的 Annotation，但有时需要取得的是某个指定的 Annotation，所以此时在取得之前就必须进行明确的判断，使用 isAnnotationPresent()方法进行判断。为了方便读者理解，本部分使用以下自定义的 Annotation 完成。

范例：定义一个 RUNTIME 的 Annotation

```
package org.lxh.demo16.reflectannotation;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
@Retention(value=RetentionPolicy.RUNTIME)  // 此Annotation在类执行时依然有效
public @interface MyDefaultAnnotationReflect {
    public String key() default "LXH" ;     // 定义一个key变量，默认值为“LXH”
    public String value() default "李兴华" ;// 定义一个value变量，默认值为
                                            “李兴华”
}
```

之后编写一个类，在调用时明确地传入 key 和 value 的内容。

范例：在类中使用 Annotation

```
package org.lxh.demo16.reflectannotation;
public class SimpleBeanTwo {
    @SuppressWarnings("unchecked")    // 使用@SuppressWarnings的Annotation
    @Deprecated                       // 使用@Deprecated的Annotation
    @Override                         // 使用@Override的Annotation
    // 使用自定义的Annotation并设置两个属性内容
    @MyDefaultAnnotationReflect(key = "MLDN", value = "www.mldnjava.cn")
    public String ~~toString~~() {        // 覆写toString()方法
        return "Hello LiXingHua!!!" ;// 返回信息
    }
}
```

以上类为了说明问题，同时使用了 4 个 Annotation，其中 3 个为 JDK 内建的。

范例：取得指定的 Annotation

```
package org.lxh.demo16.reflectannotation;
import java.lang.reflect.Method;
public class ReflectDemo02 {
    public static void main(String args[]) throws Exception {
        Class<?> c = null ;                          // 取得Class实例
        c = Class.forName("org.lxh.demo16.reflectannotation.SimpleBeanTwo") ;
        Method toM = c.getMethod("toString");        // 取得toString()方法
        if (toM.isAnnotationPresent(MyDefaultAnnotationReflect.class)) {
            MyDefaultAnnotationReflect mda = null; // 声明自定义的
                                                      Annotation对象
            // 取得自定义的Annotation
            mda = toM.getAnnotation(MyDefaultAnnotationReflect.class);
            String key = mda.key();                  // 得到Annotation中指定
                                                      变量的内容
```

```
                String value = mda.value();                    // 得到Annotation中
                                                                  指定变量的内容
                System.out.println("key = " + key);             // 输出Annotation中
                                                                  的key
                System.out.println("value = " + value);         // 输出Annotation中
                                                                  的value
            }
        }
    }
```

程序运行结果：

```
key = MLDN
value = www.mldnjava.cn
```

通过以上程序读者可以发现，只要适当地应用反射机制，就可以将 Annotation 中指定的内容设置到对应的操作中，这些操作在 EJB 3.0 中经常使用。

16.5　@Target 注释

如果一个 Annotation 没有明确地指明定义的位置，则可以在任意的位置使用，例如之前所讲解的全部的 Annotation 因为没有指定应用位置，所以可以在任意位置上进行定义，如下所示：

```
package org.lxh.demo16.targetdemo;
import org.lxh.demo16.reflectannotation.MyDefaultAnnotationReflect;
@MyDefaultAnnotationReflect(key = "MLDN", value = "www.mldnjava.cn")
public class SimpleBean {
    // 使用自定义的Annotation并设置两个属性内容
    @MyDefaultAnnotationReflect(key = "MLDN", value = "www.mldnjava.cn")
    public String toString() {                    // 覆写toString()方法
        return "Hello LiXingHua!!!" ;             // 返回信息
    }
}
```

以上定义的 MyDefaultAnnotationReflect 可以在方法上声明，也可以在类上声明；而如果希望一个自定义的 Annotation 只能在指定的位置上出现，例如，只能在类上或只能在方法中声明，则必须使用@Target 注释。

@Target 明确地指出了一个 Annotation 的使用位置，此注释的定义如下：

```
@Documented
@Retention(value=RUNTIME)
@Target(value=ANNOTATION_TYPE)
public @interface Target{
    public abstract ElementType[] value
}
```

在 Target 的注释中存在一个 ElementType[]枚举类型的变量，这个变量主要指定 Annotation 的使用限制。此枚举类定义如表 16-5 所示的几种保存范围。

表 16-5 ElementType 的保存范围

序号	范 围	描 述
1	public static final ElementType ANNOTATION_TYPE	只能用在注释的声明上
2	public static final ElementType CONSTRUCTOR	只能用在构造方法的声明上
3	public static final ElementType FIELD	只能用在字段的声明（包括枚举常量）上
4	public static final ElementType LOCAL_VARIABLE	只能用在局部变量的声明上
5	public static final ElementType METHOD	只能用在方法的声明上
6	public static final ElementType PACKAGE	只能用在包的声明上
7	public static final ElementType PARAMETER	只能用在参数的声明上
8	public static final ElementType TYPE	只能用在类、接口、枚举类型上

下面定义一个 Annotation，此 Annotation 只能用在类上，不能用在方法中。

范例：定义 Annotation

```
package org.lxh.demo16.targetdemo;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
@Target(ElementType.TYPE)                          // 此Annotation只能用在类上
@Retention(value=RetentionPolicy.RUNTIME)// 此Annotation在类执行时依然有效
public @interface MyTargetAnnotation {
    public String key() default "LXH" ; // 定义一个key变量，默认值为“LXH”
    public String value() default "李兴华";// 定义一个value变量，默认值为“李兴华”
}
```

下面定义一个类，此类在方法上使用此 Annotation。因为在 Target 上已经明确地指明是在类上使用，所以，代码编译时将出现错误。

范例：错误的 Annotation 应用

```
package org.lxh.demo16.targetdemo;
@MyTargetAnnotation(key = "MLDN", value = "www.mldnjava.cn")
public class SimpleBean {
    // 使用自定义的Annotation并设置两个属性内容
    @MyTargetAnnotation(key = "MLDN", value = "www.mldnjava.cn")
    public String toString() {                    // 覆写toString()方法
        return "Hello LiXingHua!!!" ;             // 返回信息
    }
}
```

编译时出现以下错误：

```
SimpleBean.java:7: annotation type not applicable to this kind of declaration
```

```
        @MyTargetAnnotation(key = "MLDN", value = "www.mldnjava.cn")
        ^
1 error
```

以上因为 MyTargetAnnotation 只能应用在类的声明上，所以在方法上使用时将出现错误。另外，从@Target 的定义上可以发现其中可以接收一个 ElementType 的一个数组，说明可以同时设置多个 Annotation 的出现位置。

范例：定义一个 Annotation 可以出现在类、方法上

```
package org.lxh.demo16.targetdemo;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
@Target({ElementType.TYPE,ElementType.METHOD})          // 此Annotation只能用在类及方法上
@Retention(value=RetentionPolicy.RUNTIME)               // 此Annotation在类执行时依然有效
public @interface MyTargetAnnotation {
    public String key() default "LXH" ;                 // 定义一个key变量，默认值为“LXH”
    public String value() default "李兴华" ;             // 定义一个value变量，默认值为“李兴华”
}
```

16.6 @Documented 注释

任何一个自定义的 Annotation 实际上都是通过@Documented 进行注释的，在生成 javadoc 时可以通过@Documented 将一些文档的说明信息写入。@Documented 的使用格式如格式 16-3 所示。

【格式 16-3 使用@Documented 注释】

```
@Documented
[public] @interface Annotation名称{
    数据类型 变量名称() ;
}
```

范例：使用完整格式定义 Annotation

```
package org.lxh.demo16.documenteddemo;
import java.lang.annotation.Documented;
@Documented
public @interface MyDocumentedAnnotation {
    public String key() ;
```

```
    public String value() ;
}
```

使用@Documented 注释后，此 Annotation 在生成 java doc 时就可以加入类或方法的一些说明信息，这样可以便于用户了解类或方法的作用。

范例：对定义的方法进行 javadoc 注释

```
package org.lxh.demo16.documenteddemo;
@MyDocumentedAnnotation(key = "LXH", value = "LiXingHua")
public class SimpleBeanDocumented {
    /**
     * 此方法在对象输出时调用，返回对象的信息
     */
    @MyDocumentedAnnotation(key = "MLDN", value = "www.mldnjava.cn")
    public String toString() {
        return "Hello LiXingHua!!!";
    }
}
```

在命令行方式中执行以下语句：

```
javadoc -d doc SimpleBeanDocumented.java
```

运行结果如图 16-1 所示。

以上的命令表示在 doc 目录中生成 SimpleBean 的 java doc 文档。生成之后的文件夹内容如图 16-2 所示。



图 16-1　执行命令后的结果

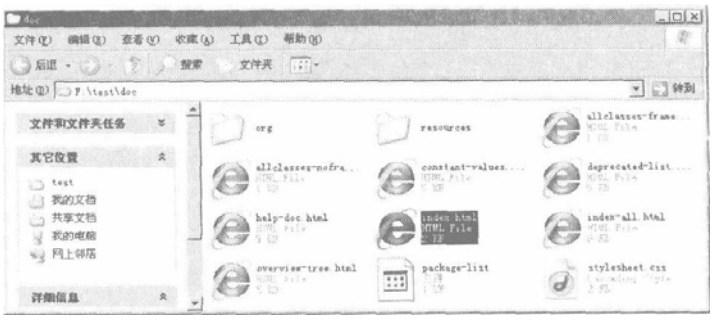


图 16-2　生成之后的文件夹内容

直接打开其中的 index.html 就可以查看之前编写好的 java doc，如图 16-3 所示。



图 16-3　打开 index.html 后的页面

16.7 @Inherited 注释

@Inherited 用于标注一个父类的注释是否可以被子类所继承，如果一个 Annotation 需要被其子类所继承，则在声明时直接使用@Inherited 注释即可。

Inherited 的 Annotation 定义如下：

```
@Documented
@Retention(value=RUNTIME)
@Target(value=ANNOTATION_TYPE)
public @interface Inherited{
}
```

下面通过代码来观察此 Inherited 的作用。

范例： 定义一个 Annotation

```
package org.lxh.demo16.inheriteddemo;
import java.lang.annotation.Documented;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
@Documented
@Inherited                                          // 此Annotation可以被继承
@Retention(value=RetentionPolicy.RUNTIME)           // 此Annotation在类执行时依然有效
public @interface MyInheritedAnnotation {
    public String name() ;                          // 定义name变量接收内容
}
```

定义一个 Person 类，并在此类中使用此 Annotation。

范例： 定义 Person 类

```
package org.lxh.demo16.inheriteddemo;
@MyInheritedAnnotation(name = "李兴华")             // 使用自定义的Annotation
public class Person {                               // 定义Person类
}
```

下面定义一个 Person 的子类。

范例： 定义 Person 子类——Student

```
package org.lxh.demo16.inheriteddemo;
public class Student extends Person {                    // 继承Person类
}
```

之后为了验证之前的 Annotation 是否被继承，可以直接通过反射机制取出全部的 Annotation。

范例：通过反射取出全部的 Annotation

```
package org.lxh.demo16.inheriteddemo;
import java.lang.annotation.Annotation;
public class ReflectInheritedDemo {
    public static void main(String args[]) throws Exception {
        Class<?> c = null ;                          // 取得Class实例
        c = Class.forName("org.lxh.demo16.inheriteddemo.Student") ;
        Annotation ann[] = c.getAnnotations();
        for (Annotation a : ann) {                   // 输出全部的Annotation
            System.out.println(a);
        }
        if (c.isAnnotationPresent(MyInheritedAnnotation.class)) {
            MyInheritedAnnotation mda = null;        // 声明自定义的Annotation
                                                        对象
            // 取得自定义的Annotation，此Annotation是从父类继承而来
            mda = c.getAnnotation(MyInheritedAnnotation.class);
            String name = mda.name();                // 得到自定义的Annotation
                                                        指定变量内容
            System.out.println("name = " + name);// 输出name变量的内容
        }
    }
}
```

程序运行结果：

```
@org.lxh.demo16.inheriteddemo.MyInheritedAnnotation(name=李兴华)
name = 李兴华
```

以上程序虽然是通过子类进行反射操作的，但是依然可以取得父类中定义的 Annotation，证明此 Annotation 已经被继承下来，而如果在声明 Annotation 时没有加上@Inherited 注释，则此 Annotation 是无法被继承的。

16.8 本章要点

1．Annotation 是 JDK 1.5 之后新增的功能，主要是使用注释的形式进行程序的开发。

2．在系统中提供了 3 个内建的 Annotation：@Override、@Deprecated、@SuppressWarnings。

3．在 Java 中使用@interface 定义一个 Annotation。

4．在 Annotation 中可以使用变量或数组接收设置内容，也可以通过枚举指定内容的取值范围。

5．在 Annotation 中使用 Retention 和 RetentionPolicy 指定 Annotation 的存在范围。

6．如果要让一个 Annotation 有意义，则必须结合反射机制一起使用。

7．@Target 注释可以指定一个 Annotation 的使用范围。

8．@Documented 注释可以在生成 java doc 时编写文档信息。

9．如果一个 Annotation 希望被使用类的子类所继承，则要使用@Inherited 注释。

16.9 习 题

定义一个可以用来接收用户登录信息的 Annotation。其中用户名和密码要求通过 Annotation 设置到验证的方法中，如下所示：

```
@LoginInfo(name="用户名",password="密码")
public boolean login(String name,String password){}
```

之后编写程序由键盘输入用户的登录信息，并通过 login()方法判断输入的用户名和密码是否正确。

第 17 章　Java 数据库编程

通过本章的学习可以达到以下目标：

- ☑ 了解 JDBC 的概念以及 4 种驱动分类。
- ☑ 了解 MySQL 的基本操作以及 SQL 语法的基本操作。
- ☑ 使用 JDBC 进行 MySQL 数据库的开发。
- ☑ 使用 DriverManager、Connection、PreparedStatement、ResultSet 对数据库进行增、删、改、查操作。
- ☑ 区分 Statement 和 PreparedStatement 的使用。
- ☑ 使用 JDBC 处理大数据。
- ☑ 掌握事务的概念以及 JDBC 对事务的支持。
- ☑ 掌握 JDBC 连接 Oracle 数据库的操作步骤。
- ☑ 了解 JDBC 中的元数据操作。

在现代的程序开发中，大量的开发都是基于数据库的，使用数据库可以方便地实现数据的存储及查找，本章将讲解 Java 的数据库操作技术——JDBC。**本章视频录像讲解时间为 4 小时 15 分钟，源代码在光盘对应的章节下。**

17.1　JDBC 概述

17.1.1　JDBC 简介

JDBC（Java Database Connectivity，Java 数据库连接），提供了一种与平台无关的用于执行 SQL 语句的标准 Java API，可以方便地实现多种关系型数据库的统一操作，它由一组用 Java 语言编写的类和接口组成。

在实际开发中可以直接使用 JDBC 进行各个数据库的连接与操作，而且可以方便地向数据库中发送各种 SQL 命令。在 JDBC 中提供的是一套标准的接口，这样，各个支持 Java 的数据库生产商只要按照此接口提供相应的实现，就都可以使用 JDBC 进行操作，极大地体现了 Java 的可移植性的设计思想。

17.1.2　JDBC 驱动分类

JDBC 本身提供的是一套数据库操作标准，而这些标准又需要各个数据库厂商实现，所以针对每一个数据库厂商都会提供一个 JDBC 的驱动程序，目前比较常见的 JDBC 驱动程序可分为以下 4 类：

➘ JDBC-ODBC 桥驱动（如图 17-1 所示）

JDBC-ODBC 是 Sun 公司提供的一个标准的 JDBC 操作，直接利用微软的 ODBC 进行数据库的连接操作，但是这种操作性能较低，所以通常情况下不推荐使用这种方式进行操作。

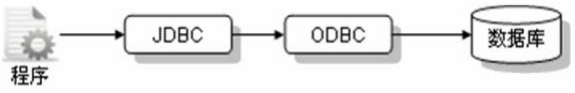


图 17-1　JDBC-ODBC 桥驱动

提示：ODBC。

ODBC（Open Database Connectivity，开放数据库连接）是微软公司提供的一套数据库操作的编程接口，Sun 公司的 JDBC 实现实际上也是模仿了 ODBC 的设计。

➘ JDBC 本地驱动（如图 17-2 所示）

直接使用各个数据库生产商提供的 JDBC 驱动程序，但是因为其只能应用在特定的数据库上，会丧失程序的可移植性，不过这样操作的性能较高。



图 17-2　JDBC 本地驱动

➘ JDBC 网络驱动

这种驱动程序将 JDBC 转换为与 DBMS 无关的网络协议，之后又被某个服务器转换为一种 DBMS 协议。这种网络服务器中间件能够将它的纯 Java 客户机连接到多种不同的数据库上，所用的具体协议取决于提供者。通常，这是最为灵活的 JDBC 驱动程序。

➘ 本地协议纯 JDBC 驱动

这种类型的驱动程序将 JDBC 调用直接转换为 DBMS 所使用的网络协议。这将允许从客户机机器上直接调用 DBMS 服务器，是 Intranet 访问的一个很实用的解决方法。

17.1.3　JDBC 的主要操作类及接口

JDBC 的核心是为用户提供 Java API 类库，让用户能够创建数据库连接、执行 SQL 语句、检索结果集、访问数据库元数据等。Java 程序开发人员可以利用这些类库来开发数据库应用程序。JDBC API 中主要包括了表 17-1 所示的类和接口。

表 17-1　JDBC 的主要操作类及接口

序号	类 及 接 口	描　　述
1	java.sql.DriverManager	用于管理 JDBC 驱动程序
2	java.sql.Connection	用于建立与特定数据库的连接，一个连接就是一个会话，建立连接后便可以执行 SQL 语句和获得检索结果
3	java.sql.Statement	一个 Statement 对象用于执行静态 SQL 语句，并获得语句执行后产生的结果
4	java.sql.PreparedStatement	创建一个可以编译的 SQL 语句对象，该对象可以被多次运行，以提高执行的效率，该接口是 Statement 的子接口

数据库的操作过程：
- 打开数据库的服务、
- 连接数据库，连接的时候一般都要输入用户名或密码
- 操作数据库：创建表、查询表、更新记录
- 关闭

DriverManager：是一个最常用的类，使用此类可以取得一个数据库的连接。
Connection：每一个 Connection 的实例化对象都表示一个数据库连接
数据库的操作：Statement、PreparedStatement
数据库的查询：ResultSet
调用存储过程：CallableStatement

续表

序号	类 及 接 口	描　　述
5	java.sql.ResultSet	用于创建表示 SQL 语句检索结果的结果集，用户通过结果集完成对数据库的访问
6	java.sql.Date	该类是标准 java.util.Date 的一个子集，用于表示与 SQL DATE 相同的日期类型，该日期不包括时间
7	java.sql.Timestamp	标准 java.util.Date 类的扩展，用于表示 SQL 时间戳，并增加了一个能表示纳秒的时间域
8	java.sql.CallableStatement	用于执行 SQL 存储过程
9	java.sql.DatabaseMetaData	与 java.sql.ResultSetMetaData 一同用于访问数据库的元信息
10	java.sql.Driver	定义一个数据库驱动程序的接口
11	java.sql.DataTruncation	在 JDBC 遇到数据截断的异常时，报告一个警告（读数据时）或产生一个异常（写数据时）
12	java.sql.DriverPropertyInfo	高级程序设计人员通过 DriverPropertyInfo 与 Driver 进行交流，可使用 getDriverPropertyInfo 获取或提供驱动程序的信息
13	java.sql.Time	该类是标准 java.util.Date 的一个子集，用于表示时、分、秒
14	java.sql.SQLException	对数据库访问时产生的错误的描述信息
15	java.sql.SQLWarning	对数据库访问时产生的警告的描述信息
16	java.sql.Types	定义了表示 SQL 类型的常量

在 JDBC 的基本操作中最常用的类和接口就是 DriverManager、Connection、Statement、Result、PreparedStatement。

17.2　MySQL 数据库

本章操作中将使用 MySQL 数据库进行操作，下面简单介绍 MySQL 的安装及使用。

17.2.1　MySQL 简介

MySQL 是一个小型关系型数据库管理系统，开发者为瑞典 MySQL AB 公司。在 2008 年 1 月 16 日被 Sun 公司收购。目前 MySQL 被广泛地应用在 Internet 上的中小型网站中。由于其体积小、速度快、总体拥有成本低，尤其是开放源码这一特点，许多中小型网站为了降低网站总体拥有成本而选择了 MySQL 作为网站数据库。MySQL 的官方网站的网址是 www.mysql.com，如图 17-3 所示。本书使用的 MySQL 版本是 5.0。



图 17-3　MySQL 官方网站

17.2.2　MySQL 安装及配置

下载 MySQL 后就可以直接执行安装操作，安装完成之后会直接提示配置的命令窗口，如图 17-4 所示。

在数据库配置时选择默认配置即可，端口设置时默认值是 3306，如图 17-5 所示。

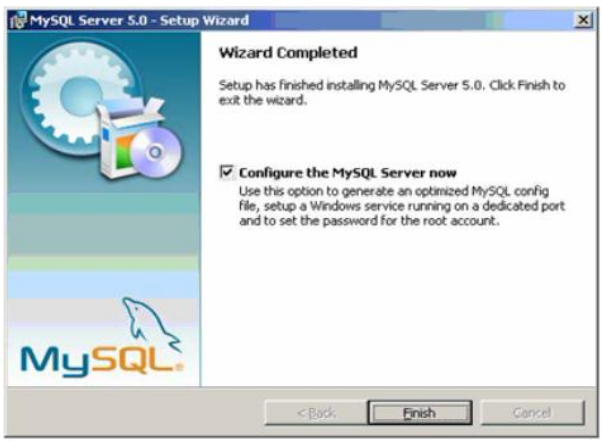


图 17-4　配置数据库提示

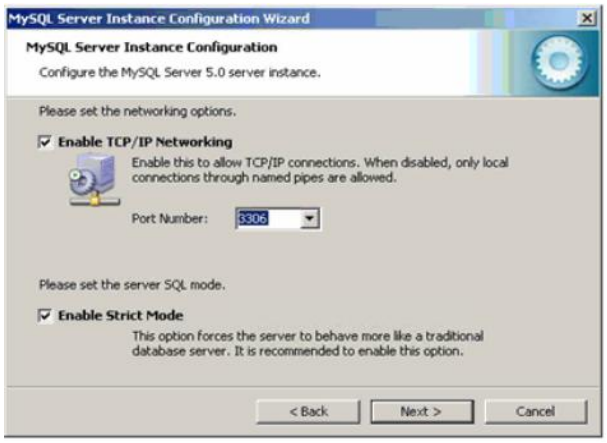


图 17-5　配置端口

之后继续单击【Next】按钮，会进入到数据库的编码设置框，在此处选择手工设置编码，因为现在是在英文环境下，所以将编码设置成 gbk，如图 17-6 所示。

编码完成之后，会出现将 MySQL 设置成系统服务的提示框，如图 17-7 所示。这样直接从 Windows 服务中找到 MySQL，就可以控制 MySQL 服务的启动和关闭，如图 17-8 所示。在此处还询问是否要在 Windows 的 path 路径中注册 MySQL 的常用命令，如果此时没有注册这些常用命令，则无法直接使用 MySQL 提供的各种命令。

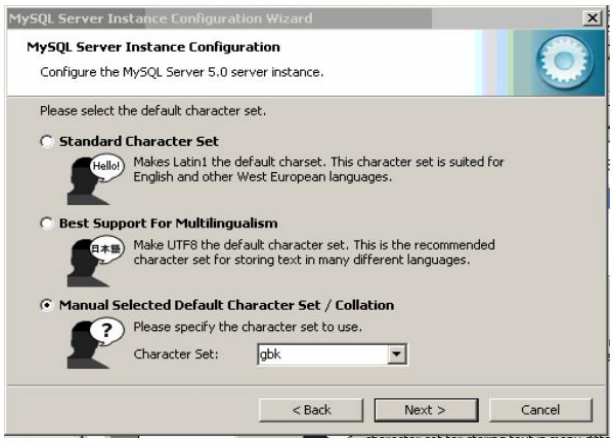


图 17-6　手工设置编码为 gbk

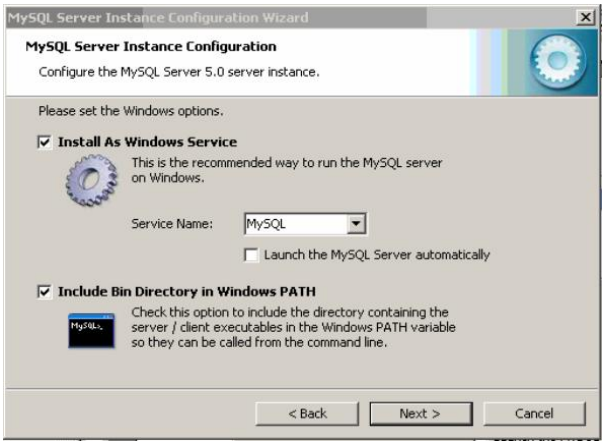


图 17-7　将 MySQL 作为服务

下一步操作之后，会出现输入密码的设置框，设置的是管理员 root 的密码，为了方便起见，所有的密码设置成 mysqladmin，如图 17-9 所示。

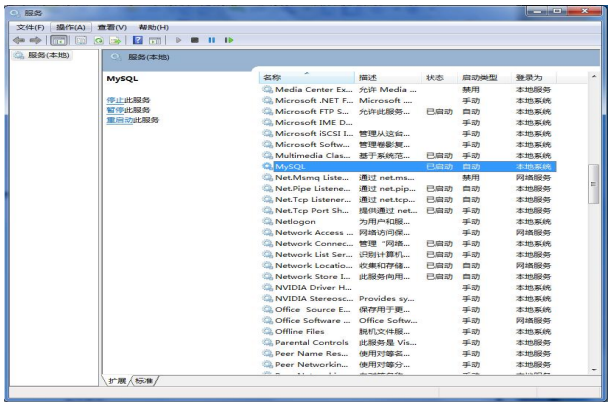


图 17-8　在服务中注册 MySQL 服务



图 17-9　设置密码

此时 MySQL 配置完成，以后可以直接通过命令行方式操作 MySQL 数据库。

提示：修改数据库编码。

如果在数据库配置时没有选择好编码，则以后在插入数据时有可能出现乱码，此时可以直接打开 MySQL 安装目录，找到其中的 my.ini 文件，将全部的 default-character-set 内容修改，如下所示：

```
default-character-set=gbk
```

修改完之后，将 MySQL 数据库的服务重新启动即可。

17.2.3　MySQL 常用命令

MySQL 安装完后就可以在命令行方式下进行数据库的连接操作，打开 mysql 的系统服务，之后按照 17-1 的命令格式输入，就可以连接到 mysql 数据库，如图 17-10 所示。MySQL 中的超级管理员名称为 root。

【格式 17-1　连接 mysql 数据库】

```
mysql -u用户名 -p密码
```

如果想知道在 mysql 中有哪些命令，则可以输入“?”查看，如图 17-11 所示。

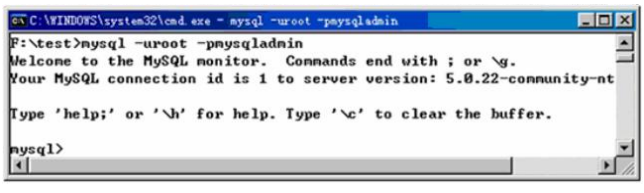


图 17-10　mysql 连接之后

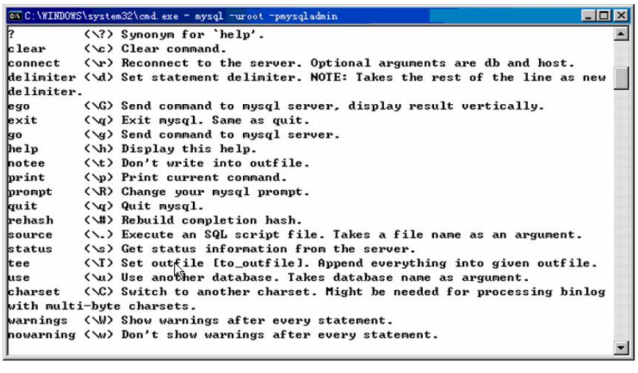


图 17-11　浏览 mysql 命令

在 mysql 中可以方便地创建和删除数据库（如图 17-12 所示），使用 17-2 所示的格式即可。

【格式 17-2　创建数据库】

创建数据库：CREATE DATABASE 数据库名称 ；

删除数据库：DROP DATABASE 数据库名称 ；

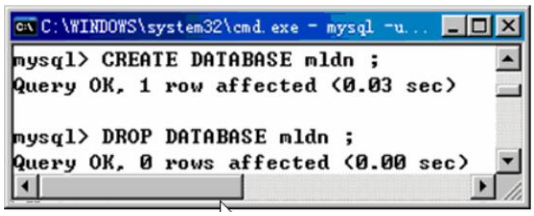


图 17-12　创建和删除数据库

如果要使用一个数据库（如图 17-13 所示），则可以通过 17-3 所示的格式。

【格式 17-3　使用数据库】

USE 数据库名称 ；

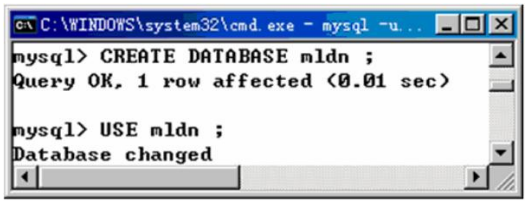


图 17-13　使用数据库

连接到数据库后，就可以执行建表语句，使用 17-4 所示的语句格式，可以创建数据库表，如图 17-14 所示。

【格式 17-4　创建数据库表】

```
CREATE TABLE 表名称(
    字段名称1字段类型 [DEFAULT 默认值] [约束] ,
    字段名称2字段类型 [DEFAULT 默认值] [约束] ,
    ...,
    字段名称n字段类型 [DEFAULT 默认值] [约束]
) ;
```

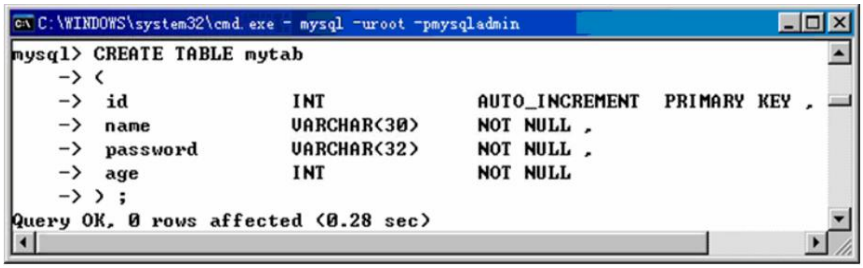


图 17-14　创建数据库表

以上表中的 id 和 age 字段是整数类型，name 和 password 都是使用字符串表示，name 字段可以保存的长度为 30，password 可以保存的长度为 32。

以上建立 id 字段时使用了一个 AUTO_INCREMENT，表示此列的内容可以自动增长。表建立完如果不需要了，也可以按照 17-5 所示的语法格式删除一张表（如图 17-15 所示）。

【格式 17-5　删除数据库表】

```
DROP TABLE 表名称 ;
```



图 17-15　删除表

如果要查看一个表结构（如图 17-16 所示），则可以直接使用 desc 命令，格式如 17-6 所示。

【格式 17-6　查看表结构】

```
DESC 表名称 ;
```

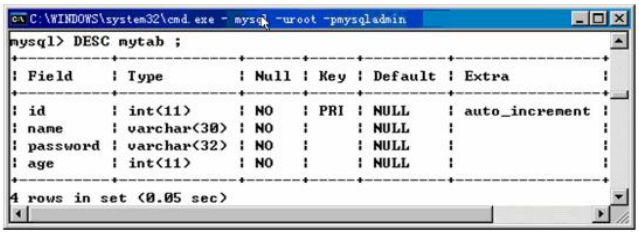


图 17-16　查看表结构

在 mysql 中可以通过 show 命令查看全部的数据库以及一个数据库下的全部表，如图 17-17 所示，格式如 17-7 所示。

【格式 17-7　查看数据库信息】

查看全部数据库：SHOW DATABASES ;

查看一个数据库的全部表：SHOW TABLES ;

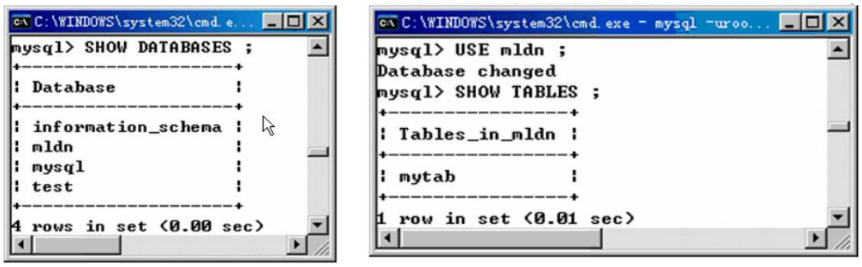


图 17-17　查看数据库信息

注意：不建议初学者使用 mysql 前台工具。

有些初学者认为以上的命令过多无法记住，便纷纷去网上搜索各种 mysql 的前台，但是这一点在本书中绝对禁止。因为这样一来会让初学者变得“懒惰”而不去熟练基本的语法，所以在本书及视频讲解中都将使用最基本的命令完成全部的操作。

17.3　SQL 语法基础

SQL（Structured Query Language，结构查询语言）是一个功能强大的数据库语言。SQL

通常用于与数据库的通信。ANSI（美国国家标准学会）声称，SQL 是关系数据库管理系统的标准语言。

SQL 功能强大，概括起来可以分成以下几组。

- DML（Data Manipulation Language，数据操作语言）：用于检索或者修改数据。
- DDL（Data Definition Language，数据定义语言）：用于定义数据的结构，如创建、修改或者删除数据库对象。
- DCL（Data Control Language，数据控制语言）：用于定义数据库用户的权限。

提示：本书不是一本专门讲解数据库的书籍。

SQL 语法的相关知识点很多，在本书讲解时因为是围绕程序进行讲解的，所以对于 SQL 语法只是讲解了本书所使用到的部分。而如果读者想连接更多的关于 SQL 语法的知识，可以直接登录 www.mldnjava.cn 网址进行教学视频的下载。

在 MySQL 数据库中良好地支持了标准的 SQL 语法，本章主要使用的是 DDL，下面分别对数据库的增加、修改、删除、查询语句进行介绍。为了方便操作，本部分使用表 17-2 完成全部操作。

表 17-2　user 表（用户表）

user		
id	int	<pk>
name	varchar(30)	
password	VARCHAR(32)	
age	INT	
sex	VARCHAR(2)	
birthday	DATE	

序号	字段名称	类型	描述
1	id	整型	用户编号，自动增长，主键
2	name	字符串	用户的真实姓名，不能为空
3	password	字符串	用户的密码，不能为空
4	age	整型	用户年龄，不能为空
5	sex	字符串	用户性别，默认值是男
6	birthday	日期	用户生日

数据库创建脚本：

```
DROP TABLE IF EXISTS user ;          -- 删除数据库表
CREATE TABLE user                    -- 创建新的数据库表
(
    id              INT              AUTO_INCREMENT  PRIMARY KEY ,
    name            VARCHAR(30)      NOT NULL ,
    password        VARCHAR(32)      NOT NULL ,
    age             INT              NOT NULL ,
    sex             VARCHAR(2)       DEFAULT '男' ,
    birthday        DATE
) ;
```

以上数据库脚本在执行时先判断数据库中是否存在表，如果存在则先删除之后再进行创建。创建时人员编号使用 AUTO_INCREMENT 完成自动增长列的操作。

17.3.1 MySQL 中的数据类型

任何数据库中都定义了很多的表示数据库中的字段类型，在 MySQL 中也同样有此定义，如表 17-3 所示。

表 17-3 MySQL 中的数据类型

序号	数 据 类 型	长 度	描 述
1	TINYINT(M)、BIT、BOOL、BOOLEAN	1	如果为无符号数，可以存储从 0～255 的数；否则可以存储从-128～127 的数
2	SMALLINT(M)	2	如果为无符号数，可以存储从 0～65535 的数；否则可以存储从-32768～32767 的数
3	MEDIUMINT(M)	3	如果为无符号数，可以存储从 0～16777215 的数；否则可以存储从-8388608～8388607 的数
4	INT(M)、INTEGER(M)	4	如果为无符号数，可以存储从 0～4294967295 的数，否则可以存储从-2147483648～2147483647 的数
5	BIGINT(M)	8	如果为无符号数，可以存储从 0～18446744073709551615 的数，否则可以存储从-9223372036854775808～9223372036854775807 的数
6	FLOAT(precision)	4 或 8	这里的 precision 是可以直达 53 的整数。如果 precision<=24 则转换为 FLOAT，如果 precision>24 并且 precision<=53 则转换为 DOUBLE
7	FLOAT(M,D)	4	单精度浮点数
8	DOUBLE(M,D)、DOUBLE PRECISION、REAL	8	双精度浮点数
9	DECIMAL(M,D)、DEC、NUMERIC、FIXED	M+1 或 M+2	未打包的浮点数
10	DATE	3	以 YYYY-MM-DD 的格式显示
11	DATETIME HH:MM:SS	8	以 YYYY-MM-DD 的格式显示
12	TIMESTAMP	4	以 YYYY-MM-DD 的格式显示
13	TIME	3	以 HH:MM:SS 的格式显示
14	YEAR	1	以 YYYY 的格式显示
15	CHAR(M)	M	定长字符串
16	VARCHAR(M)	最大 M	变长字符串。M<=255
17	TINYBLOB、TINYTEXT	最大 255	TINYBLOB 为大小写敏感，而 TINYTEXT 不是大小写敏感的
18	BLOB、TEXT	最大 64KB	BLOB 为大小写敏感的，而 TEXT 不是大小写敏感的
19	MEDIUMBLOB、MEDIUMTEXT	最大 16MB	MEDIUMBLOB 为大小写敏感的，而 MEDIUMTEXT 不是大小写敏感的

续表

序号	数 据 类 型	长　　度	描　　述
20	LONGBLOB、LONGTEXT	最大 4GB	LONGBLOB 为大小写敏感的，而 LONGTEXT 不是大小写敏感的
21	ENUM(VALUE1,...)	1 或 2	最大可达 65535 个不同的值
22	SET(VALUE1,...)	可达 8	最大可达 64 个不同的值

在以上定义的数据类型中，最常使用的是 INT、FLOAT、VARCHAR(M)、TEXT、DATE、DATETIME、BLOB，这些应用在操作 JDBC 时都会为读者讲解清楚的。

17.3.2　增加数据

向表中增加数据，可以使用如 17-8 所示的格式。

【格式 17-8　增加数据】

```
INSERT INTO 表名称[(字段1,字段2,字段3,...,字段n)] VALUES (值1,值2,值3,...值n) ;
```

但是需要向读者说明的是，在增加数据时，如果是字符串，则一定要使用“'”括起来；如果是数字，则不需要“'”；如果是日期，则按照标准的日期格式进行插入。

提示：MySQL 中的日期格式。

MySQL 数据库中的日期使用 yyyy-mm-dd 的格式保存，所以在插入数据时，必须按照此种格式插入，例如 2008-08-27。

范例：向 user 表中插入数据库（如图 17-18 所示）

```
INSERT INTO user(name,password,age,sex,birthday) VALUES ('李兴华','www.mldn.cn',30,'男','2008-08-27') ;
INSERT INTO user(name,password,age,sex,birthday) VALUES ('李康','www.mldnjava.cn',23,'女','2005-08-27') ;
```

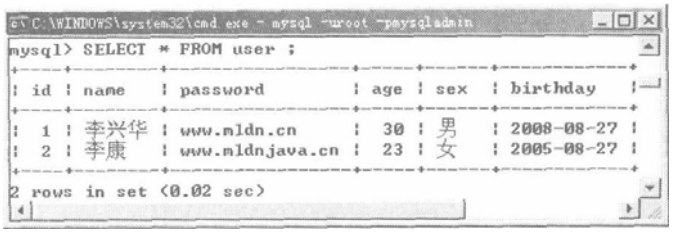


图 17-18　增加数据之后的 user 表内容

在 user 表中 id 字段属于自动增长列，从运行结果来看，会为每一条数据自动进行编号。

17.3.3　删除数据

删除表中的数据，可以使用如 17-9 所示的格式。

【格式 17-9　删除数据】

```
DELETE FROM 表名称 [删除条件] ;
```

在进行删除数据时，最好指定删除的条件，如果没有指定，则表示删除一张表中的全部数据。一般这个条件都使用 id 表示。

范例：删除第一条记录（如图 17-19 所示）

```
DELETE FROM user WHERE id=1 ;
```

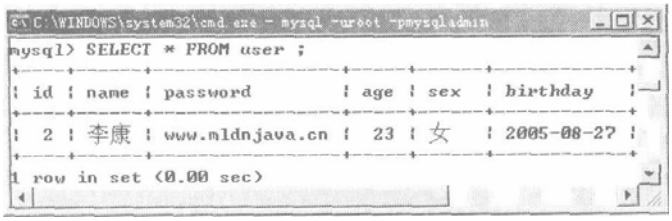


图 17-19　删除数据之后的 user 表内容

17.3.4　更新数据

当需要修改数据表中的某些记录时，就可以使用 UPDATE 语句，语句格式如 17-10 所示。

【格式 17-10　更新数据】

```
UPDATE 表名称 SET 字段1=值1,....字段n=值1 [WHERE 更新条件] ;
```

与删除语句一样，修改时也需要指定修改条件，否则数据表的全部记录都将被修改。一般的条件都使用 id 表示。

范例：修改数据（如图 17-20 所示）

```
UPDATE user SET name='MLDN',age=5,birthday='2000-08-27' WHERE id=2 ;
```

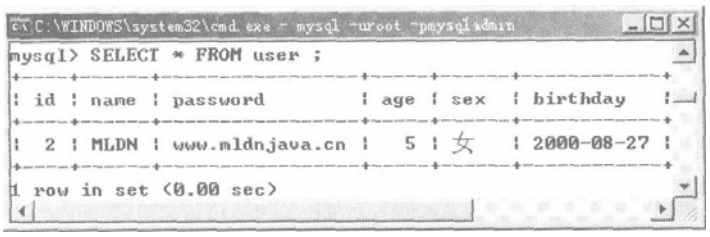


图 17-20　更新数据之后的 user 表内容

17.3.5　查询数据

在 SQL 语法中，如果要查看表中的记录内容，可以通过查询语句完成，用户发出查询指令时会将全部的查询结果返回给用户。查询语句的格式如 17-11 所示。

【格式 17-11　简单查询语句】

```
SELECT {*|column alias}
FROM 表名称 别名
[WHERE condition(s)] ;          → 设置查询条件
```

范例：查询全部数据（如图 17-21 所示）

```
SELECT * FROM user ;
```

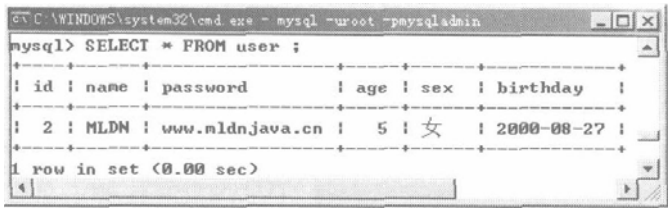


图 17-21　查询全部数据

范例：查询 user 表中的姓名和生日（如图 17-22 所示）

```
SELECT name,birthday FROM user ;
```

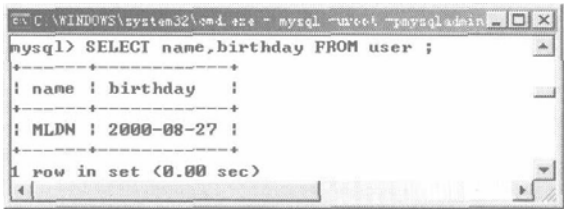


图 17-22　查询 user 表中的姓名和生日

范例：查询姓名或密码中包含字母 m 的用户（如图 17-23 所示）

```
SELECT * FROM user WHERE name LIKE '%m%' OR password LIKE '%m%' ;
```

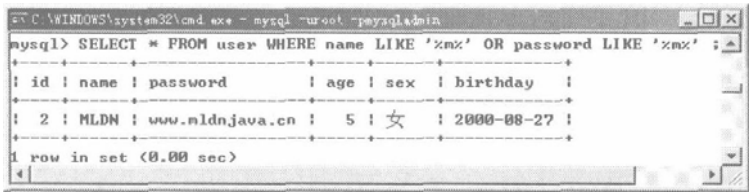


图 17-23　模糊查询

上述查询语句中使用了 LIKE 语句，用于进行数据的模糊查询，其中%表示匹配任意的数据。

以上的查询是将全部的数据都取了出来，如果要想取出数据库表中的一部分数据，则在查询的最后加上一个 LIMIT 语句即可。此语句格式如 17-12 所示。

【格式 17-12　LIMIT 语句】

```
SELECT 字段 FROM 表 WHERE 条件 LIMIT 开始行,取出的数据个数 ;
```

范例：取出第 1～5 行的记录（如图 17-24 所示）

提示：读者此时可以向数据库中多增加些数据，这样观察起来会比较清晰。

```
SELECT * FROM user WHERE name LIKE '%m%' OR password LIKE '%m%' LIMIT 0,5 ;
```

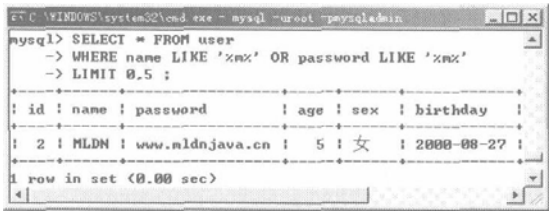


图 17-24　取出第 1～5 行数据

因为在数据库表中只有一条数据，所以此时只显示了一条数据，这种 SQL 语句在 Java 的分页开发中使用非常广泛。

提示：常见的数据库。

在现在的开发中一般比较常见的数据库有 Oracle、DB2、MySQL 等，这些数据库在使用时都支持标准的 SQL 语法操作，所以各个数据库只要了解其基本命令就可以很快地上手开发。

17.4 JDBC 操作步骤

数据库安装并配置完成后，即可按图 17-25 所示步骤进行数据库的操作。

（1）加载数据库驱动程序：各个数据库都会提供 JDBC 的驱动程序开发包，直接把 JDBC 操作所需要的开发包（一般为*.jar 或*.zip）配置到 classpath 路径即可。

（2）连接数据库：根据各个数据库的不同，连接的地址也不同，此连接地址将由数据库厂商提供。一般在使用 JDBC 连接数据库时都要求用户输入数据库连接的用户名和密码，本章使用的是 mysql 数据库，所以用户名为 root，密码为 mysqladmin，用户在取得连接之后才可以对数据库进行查询或更新的操作。

（3）使用语句进行数据库操作：数据库操作分为更新和查询两种，除了可以使用标准的 SQL 语句外，对于各个数据库也可以使用其自己提供的各种命令。

（4）关闭数据库连接：数据库操作完毕之后需要关闭连接以释放资源。



图 17-25 JDBC 操作步骤

17.5 连接数据库

17.5.1 配置 MySQL 数据库的驱动程序

如果要使用 MySQL 数据库进行开发，首先必须将 MySQL 数据库的驱动程序配置到 classpath 中，直接修改本机的环境 classpath 属性即可。现在假定 MySQL 的数据库驱动程序保存在 F:\test\mysql-connector-java-5.0.4-bin.jar 路径中，如图 17-26 所示。

提示：MySQL 驱动程序下载。

MySQL 的驱动程序可以直接从 MySQL 的官方网站 http://dev.mysql.com/downloads/connector/ 下载到。



图 17-26　设置 MySQL 驱动程序

提示：没有 classpath 属性用户可以自己建立。

如果在环境变量中没有发现 CLASSPATH 属性，则可以直接建立一个新的用户变量，但是建立时一定要注意变量名称 CLASSPATH 必须大写，在设置变量值时要加入一个“.”表示可以从当前所在的文件夹查找所要的类，之后的每一个新的内容使用“;”与之前的内容进行分割。

17.5.2　加载驱动程序

加载数据库驱动程序是 JDBC 操作的第一步，在之前已经将数据库的驱动程序直接配置到了 classpath 中，所以，此时可以直接进行加载。不同数据库的驱动程序路径是不一样的，MySQL 中的数据库驱动程序路径是 org.gjt.mm.mysql.Driver。

提示：不用强记数据库的驱动程序路径。

各个数据库的驱动程序路径实际上就是驱动包提供的类名称，采用“包.类”名称的方式，以 MySQL 为例，使用 winrar 打开 mysql-connector-java-5.0.4-bin.jar 即可发现，如图 17-27 所示。

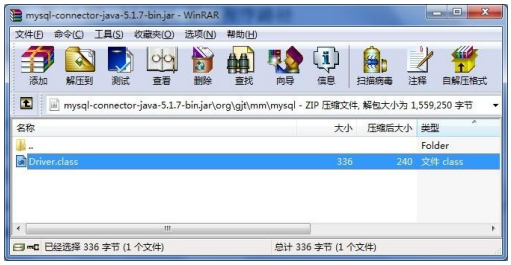


图 17-27　MySQL 驱动程序路径

得到驱动程序路径后，即可利用 Class 类进行驱动程序的加载，如下所示。

范例：加载驱动程序

```
package org.lxh.demo17. connectdemo;
public class ConnectionDemo01 {
    // 定义MySQL的数据库驱动程序
    public static final String DBDRIVER = "org.gjt.mm.mysql.Driver" ;
    public static void main(String[] args) {
        try {
            Class.forName(DBDRIVER) ;                          // 加载驱动程序
        } catch (ClassNotFoundException e) {
```

```
            e.printStackTrace();
        }
    }
}
```

如果以上程序可以正常执行，则证明数据库驱动程序已经配置成功。

提示：配置出错时的解决！

如果配置时出现了以下的错误提示：

```
java.lang.ClassNotFoundException: org.gjt.mm.mysql.Driver
```

则肯定是 CLASSPATH 设置有问题，也有可能是命令行方式没有重新启动造成的。

17.5.3 连接及关闭数据库

如果数据库驱动程序可以正常加载，下面就可以使用 DriverManager 类连接数据库。DriverManager 类中的常用方法如表 17-4 所示。

表 17-4 DriverManager 类的常用方法

序号	方　法	类型	描　述
1	public static Connection getConnection(String url) throws SQLException	普通	通过连接地址连接数据库
2	public static Connection getConnection(String url, String user,String password) throws SQLException	普通	通过连接地址连接数据库，同时输入用户名和密码

在 DriverManager 中，提供的主要操作就是得到一个数据库的连接，getConnection()方法就是取得连接对象，此方法返回的类型是 Connection 对象，不管使用哪种方式连接，都必须提供一个数据库的连接地址，MySQL 数据库的连接地址形式如格式 17-13 所示。如果在连接数据库时需要用户名和密码，则还需要将用户名和密码设置上。

【格式 17-13 MySQL 数据库的连接地址格式】

```
jdbc:mysql://IP地址:端口号/数据库名称
```

现在在本机上存在一个 mldn 的数据库，所以此时的地址为 jdbc:mysql://localhost:3306/mldn。

提示：数据库连接地址的形式。

JDBC 虽然提供了与平台无关的数据库操作，但是各个数据库的连接地址是有差异的，JDBC 的连接地址实际上由以下 3 个部分组成。

- jdbc 协议：JDBC URL 中的协议总是 jdbc。
- 子协议：驱动程序名或数据库连接机制（这种机制可由一个或多个驱动程序支持）的名称，如 mysql。
- 子名称：一种标识数据库的方法。必须遵循“//主机名：端口/子协议”的标准 URL 命名约定，如//localhost:3306/mldn。

了解了 DriverManager 类之后，再来看 Connection 接口，因为以后所有数据库的操作都从此接口开始。Connection 接口中的常用方法如表 17-5 所示。

表 17-5　Connection 接口的常用方法

序号	方　　法	类型	描　　述
1	Statement createStatement() throws SQLException	普通	创建一个 Statement 对象
2	Statement createStatement(int resultSetType,int resultSetConcurrency) throws SQLException	普通	创建一个 Statement 对象，该对象将生成具有给定类型和并发性的 ResultSet 对象
3	PreparedStatement prepareStatement(String sql) throws SQLException	普通	创建一个 PreparedStatement 类型的对象
4	PreparedStatement prepareStatement(String sql, int resultSetType,int resultSetConcurrency) throws SQLException	普通	创建一个 PreparedStatement 对象，该对象将生成具有给定类型和并发性的 ResultSet 对象
5	CallableStatement prepareCall(String sql) throws SQLException	普通	创建一个 CallableStatement 对象，此对象专门用于调用数据库的存储过程
6	CallableStatement prepareCall(String sql,int resultSetType,int resultSetConcurrency) throws SQLException	普通	创建一个 CallableStatement 对象，该对象将生成具有给定类型和并发性的 ResultSet 对象
7	DatabaseMetaData getMetaData() throws SQLException	普通	得到数据库的元数据
8	void setAutoCommit(boolean autoCommit) throws SQLException	普通	设置数据库的自动提交，与事务有关
9	boolean getAutoCommit() throws SQLException	普通	判断数据库是否可以自动提交，与事务有关
10	Savepoint setSavepoint() throws SQLException	普通	设置数据库的恢复点，与事务有关
11	Savepoint setSavepoint(String name) throws SQLException	普通	为数据库的恢复点指定一个名字，与事务有关
12	void rollback() throws SQLException	普通	数据库操作回滚，与事务有关
13	void rollback(Savepoint savepoint) throws SQLException	普通	数据库回滚到指定的保存点，与事务有关
14	void commit() throws SQLException	普通	提交操作，与事务有关
15	boolean isClosed() throws SQLException	普通	判断连接是否已关闭
16	void close() throws SQLException	普通	关闭数据库
17	DatabaseMetaData getMetaData() throws SQLException	普通	得到数据库的元数据对象

以上列举的方法在 JDBC 操作中都非常有用，本章将分部介绍方法的使用。下面先来介绍如何连接数据库。

范例：连接数据库

```
package org.lxh.demo17.connectdemo;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
```

```
public class ConnectionDemo02 {
    // 定义MySQL的数据库驱动程序
    public static final String DBDRIVER = "org.gjt.mm.mysql.Driver" ;
    // 定义MySQL数据库的连接地址
    public static final String DBURL = "jdbc:mysql://localhost:3306/mldn" ;
    // MySQL数据库的连接用户名
    public static final String DBUSER = "root" ;
    // MySQL数据库的连接密码
    public static final String DBPASS = "mysqladmin" ;
    public static void main(String[] args) {
        Connection conn = null ;                          // 数据库连接
        try {
            Class.forName(DBDRIVER) ;                     // 加载驱动程序
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        try {
            // 连接MySQL数据库时，要写上连接的用户名和密码
            conn = DriverManager.getConnection(DBURL, DBUSER, DBPASS);
        } catch (SQLException e) {
            e.printStackTrace();
        }
        System.out.println(conn) ;
        try {
            conn.close() ;                                // 数据库关闭
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

程序运行结果：

```
com.mysql.jdbc.Connection@119cca4
```

程序的运行结果不为空，说明此时数据库已经连接成功。

注意：数据库打开之后必须关闭。

在程序操作中，数据库的资源是非常有限的，这就要求开发者在操作完数据库之后必须将其关闭。如果没有这样做，在程序运行中就会产生无法连接到数据库的异常。

17.6 执行数据库的更新操作

数据库连接后，下面即可进行数据库的具体操作，如果要对数据库进行操作，则肯定

要使用 Statement 接口完成，此接口可以使用 Connection 接口中提供的 createStatement()方法实例化。Statement 接口中定义了如表 17-6 所示的常用方法。

表 17-6　Statement 接口的常用方法

序号	方　　法	类型	描　　述
1	int executeUpdate(String sql) throws SQLException	普通	执行数据库更新的 SQL 语句，如 INSERT、UPDATE、DELETE 等语句，返回更新的记录数
2	ResultSet executeQuery(String sql) throws SQLException	普通	执行数据库查询操作，返回一个结果集对象
3	void addBatch(String sql) throws SQLException	普通	增加一个待执行的 SQL 语句
4	int[] executeBatch() throws SQLException	普通	批量执行 SQL 语句
5	void close() throws SQLException	普通	关闭 Statement 操作
6	boolean execute(String sql) throws SQLException	普通	执行 SQL 语句

下面使用 Statement 接口分别完成数据库的插入、修改、删除操作。

17.6.1　实例操作——执行数据库插入操作

下面直接向 user 表中增加一条新的记录，编写一条完整的 SQL 语句，并通过 Statement 执行。

范例：插入数据

```
package org.lxh.demo17.statementdemo;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
public class InsertDemo01 {
    // 定义MySQL的数据库驱动程序
    public static final String DBDRIVER = "org.gjt.mm.mysql.Driver" ;
    // 定义MySQL数据库的连接地址
    public static final String DBURL = "jdbc:mysql://localhost:3306/mldn" ;
    // MySQL数据库的连接用户名
    public static final String DBUSER = "root" ;
    // MySQL数据库的连接密码
    public static final String DBPASS = "mysqladmin" ;
    public static void main(String[] args) throws Exception {
        Connection conn = null ;                    // 数据库连接
        Statement stmt = null ;                     // 数据库操作
        String sql = "INSERT INTO user(name,password,age,sex,birthday) "
                + " VALUES ('李兴华','www.mldn.cn',30,'男','2008-08-27')";
        Class.forName(DBDRIVER) ;                   // 加载驱动程序
        // 连接MySQL数据库时，要写上连接的用户名和密码
```

```
            conn = DriverManager.getConnection(DBURL, DBUSER, DBPASS);
            stmt = conn.createStatement() ;              // 实例化Statement对象
            stmt.executeUpdate(sql) ;                    // 执行数据库更新操作
            stmt.close() ;                               // 操作关闭
            conn.close() ;                               // 数据库关闭
        }
    }
```

本程序为了读者观看方便，将所有的异常直接在主方法抛出以减少程序中的 try…catch 代码，本程序执行完毕后，查询 mysql 数据库的 user 表，得到如图 17-28 所示的信息。

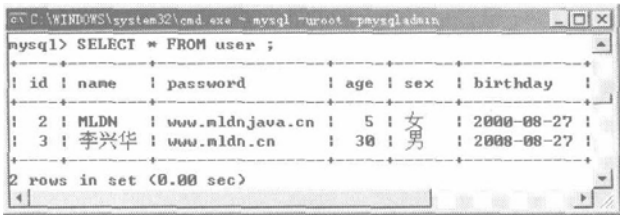


图 17-28　执行插入操作之后的 user 表内容

从以上程序中可以发现，执行 SQL 语句时，执行的是一条标准的 SQL 语句，肯定可以正常执行，但是程序的 SQL 语句中的数据是固定的。如果将全部的内容换成变量，则可以按照以下的代码编写：

```
Statement stmt = null ;                          // 数据库操作
String name = "李康";                            // 姓名
String password = "www.mldnjava.cn";             // 密码
int age = 23;                                    // 年龄
String sex = "女";                               // 性别
String birthday = "2003-08-27";                  // 生日
String sql = "INSERT INTO user(name,password,age,sex,birthday) "
        + " VALUES ('" + name + "','" + password + "'," + age + ",'"
        + sex + "','" + birthday + "')";// 拼凑出一个完整的SQL语句
```

以上在编写 SQL 语句时，有些读者可能难以理解，会被许多的“"”或“'”搞迷糊，耐心分析一下就会发现，此处操作时只是多了几个字符串连接而已。现在的 SQL 语句同样可以执行，只是采用了拼凑的形式。但是如果使用 Statement 进行开发，则肯定会采用以上的拼凑 SQL 语句形式。

❓提问：为什么要写两个关闭？

以上代码中分别关闭了 Statement 和 Connection，在开发中只关闭一个行不行？

回答：可以只关闭一次连接。

在数据库操作中都存在关闭方法，连接有关闭、操作有关闭，一般来说连接只要一关闭，则其他的所有操作都会关闭。但是在开发 JDBC 代码中一般习惯分别按照顺序关闭，即先打开的后关闭，所以在此处先关闭 Statement 操作再关闭 Connection 操作。

17.6.2　实例操作二——执行数据库修改

要想执行数据库修改操作，只需要将 SQL 语句修改为 UPDATE 即可。

范例：数据库修改操作

```
package org.lxh.demo17.statementdemo;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
public class UpdateDemo {
    // 定义MySQL的数据库驱动程序
    public static final String DBDRIVER = "org.gjt.mm.mysql.Driver" ;
    // 定义MySQL数据库的连接地址
    public static final String DBURL = "jdbc:mysql://localhost:3306/mldn" ;
    // MySQL数据库的连接用户名
    public static final String DBUSER = "root" ;
    // MySQL数据库的连接密码
    public static final String DBPASS = "mysqladmin" ;
    public static void main(String[] args) throws Exception {
        Connection conn = null ;                    // 数据库连接
        Statement stmt = null ;                     // 数据库操作
        int id = 3;                                 // id
        String name = "李华";                       // 姓名
        String password = "mldn";                   // 密码
        int age = 25;                               // 年龄
        String sex = "男";                          // 性别
        String birthday = "2001-09-13";             // 生日
        String sql = "UPDATE user SET name='" + name + "',password='"
                + password + "',age=" + age + ",sex='" + sex + "',birthday='"
                + birthday + "' WHERE id=" + id ;   // 拼凑出一个完整的SQL语句
        Class.forName(DBDRIVER) ;                   // 加载驱动程序
        // 连接MySQL数据库时，要写上连接的用户名和密码
        conn = DriverManager.getConnection(DBURL, DBUSER, DBPASS);
        stmt = conn.createStatement() ;             // 实例化Statement对象
        stmt.executeUpdate(sql) ;                   // 执行数据库更新操作
        stmt.close() ;                              // 操作关闭
        conn.close() ;                              // 数据库关闭
    }
}
```

执行以上操作后，再次查询数据库，发现数据库中的数据已经被修改，如图 17-29 所示。

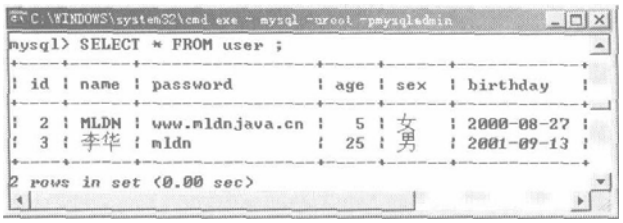


图 17-29　执行修改操作之后的数据表内容

17.6.3　实例操作三——执行数据库删除操作

与之前一样，直接执行 DELETE 的 SQL 语句即可完成记录的删除操作。

范例：按 id 删除一条记录

```
package org.lxh.demo17.statementdemo;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
public class DeleteDemo {
    // 定义MySQL的数据库驱动程序
    public static final String DBDRIVER = "org.gjt.mm.mysql.Driver" ;
    // 定义MySQL数据库的连接地址
    public static final String DBURL = "jdbc:mysql://localhost:3306/mldn" ;
    // MySQL数据库的连接用户名
    public static final String DBUSER = "root" ;
    // MySQL数据库的连接密码
    public static final String DBPASS = "mysqladmin" ;
    public static void main(String[] args) throws Exception {
        Connection conn = null ;                        // 数据库连接
        Statement stmt = null ;                         // 数据库操作
        int id = 3;                                     // id
        String sql = "DELETE FROM user WHERE id=" + id; // 拼凑出一个完整的
                                                            SQL语句
        Class.forName(DBDRIVER) ;                       // 加载驱动程序
        // 连接MySQL数据库时，要写上连接的用户名和密码
        conn = DriverManager.getConnection(DBURL, DBUSER, DBPASS);
        stmt = conn.createStatement() ;                 // 实例化Statement对象
        stmt.executeUpdate(sql) ;                       // 执行数据库更新操作
        stmt.close() ;                                  // 操作关闭
        conn.close() ;                                  // 数据库关闭
    }
}
```

查询数据库后，可以发现编号为 3 的用户已经被删除，如图 17-30 所示。

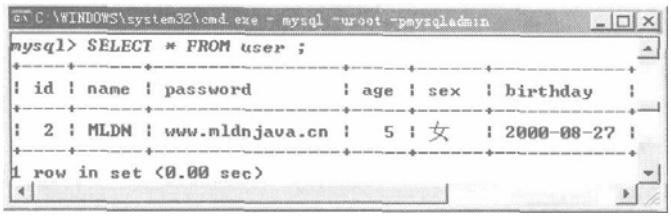


图 17-30　执行删除之后的数据表内容

17.7　ResultSet 接口

使用 SQL 中的 SELECT 语句可以查询出数据库的全部结果，在 JDBC 的操作中数据库的所有查询记录将使用 ResultSet 进行接收，并使用 ResultSet 显示内容，如图 17-31 所示。

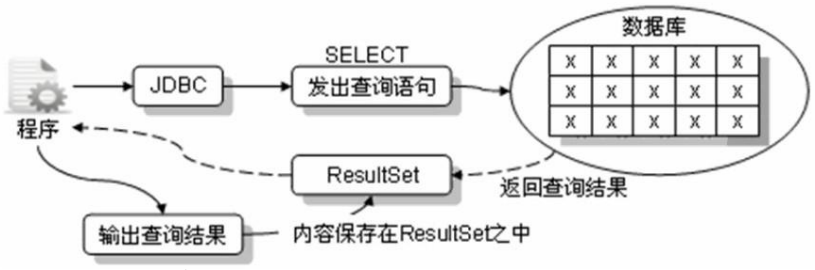


图 17-31　查询过程

注意：开发中要限制查询数量。

在 JDBC 的查询操作中，因为是将数据库表中的全部查询结果保存在了 ResultSet 对象中，实际上也就是保存在了内存中，所以如果查询出来的数据总量过大，则系统将会出现问题。这一点在系统的开发中尤为重要。

之前所讲解的全部操作都属于数据库的更新操作，直接使用 Statement 接口中定义的 executeUpdate()方法即可完成操作。如果现在进行数据库查询，则需要使用 Statement 接口定义的 executeQuery()方法，此方法返回值类型就是一个 ResultSet 的对象，此对象中存放了所有的查询结果。ResultSet 接口的常用操作方法如表 17-7 所示。

表 17-7　ResultSet 接口的常用操作方法

序号	方　　法	类型	描　　述
1	boolean next() throws SQLException	普通	将指针移到下一行
2	int getInt(int columnIndex) throws SQLException	普通	以整数形式按列的编号取得指定列的内容
3	int getInt(String columnName) throws SQLException	普通	以整数形式取得指定列的内容
4	float getFloat(int columnIndex) throws SQLException	普通	以浮点数的形式按列的编号取得指定列的内容
5	float getFloat(String columnName) throws SQLException	普通	以浮点数的形式取得指定列的内容
6	String getString(int columnIndex) throws SQLException	普通	以字符串的形式按列的编号取得指定列的内容

续表

序号	方　　法	类型	描　　述
7	String getString(String columnName) throws SQLException	普通	以字符串的形式取得指定列的内容
8	Date getDate(int columnIndex) throws SQLException	普通	以 Date 的形式按列的编号取得指定列的内容
9	Date getDate(String columnName) throws SQLException	普通	以 Date 的形式取得指定列的内容

下面使用以上方法将之前在 user 表中保存的数据取出来，并在屏幕上显示，假设 user 表中现在存在如图 17-32 所示的记录。

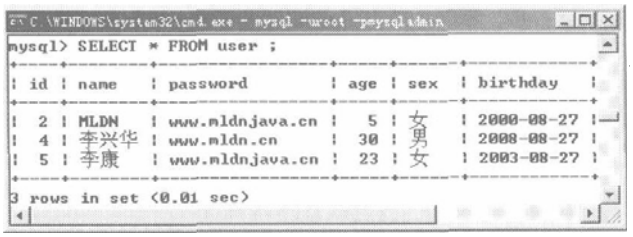


图 17-32　查询之前的 user 表数据

范例：从 user 表中查询数据

```
package org.lxh.demo17.resultsetdemo;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
public class ResultSetDemo01 {
    // 定义MySQL的数据库驱动程序
    public static final String DBDRIVER = "org.gjt.mm.mysql.Driver" ;
    // 定义MySQL数据库的连接地址
    public static final String DBURL = "jdbc:mysql://localhost:3306/mldn" ;
    // MySQL数据库的连接用户名
    public static final String DBUSER = "root" ;
    // MySQL数据库的连接密码
    public static final String DBPASS = "mysqladmin" ;
    public static void main(String[] args) throws Exception {
        Connection conn = null ;                        // 数据库连接
        Statement stmt = null ;                         // 数据库操作
        ResultSet rs = null ;                           // 保存查询结果
        String sql = "SELECT id,name,password,age,sex,birthday FROM user" ;
        Class.forName(DBDRIVER) ;                       // 加载驱动程序
        // 连接MySQL数据库时，要写上连接的用户名和密码
        conn = DriverManager.getConnection(DBURL, DBUSER, DBPASS);
        stmt = conn.createStatement() ;                 // 实例化Statement对象
        rs = stmt.executeQuery(sql) ;                   // 实例化ResultSet对象
```

```
            while(rs.next()){                                    // 指针向下移动
                int id = rs.getInt("id") ;                       // 取得id内容
                String name = rs.getString("name") ;             // 取得name内容
                String pass = rs.getString("password") ;         // 取得password内容
                int age = rs.getInt("age") ;                     // 取得age内容
                String sex = rs.getString("sex") ;               // 取得sex内容
                java.util.Date d = rs.getDate("birthday") ;// 取得birthday内容
                System.out.print("编号：" + id + "；") ;          // 输出编号
                System.out.print("姓名：" + name + "；") ;        // 输出姓名
                System.out.print("密码：" + pass + "；") ;        // 输出密码
                System.out.print("年龄：" + age + "；") ;         // 输出年龄
                System.out.print("性别：" + sex + "；") ;         // 输出性别
                System.out.println("生日：" + d) ;                // 输出生日
                System.out.println("--------------------------"); // 换行
            }
            rs.close() ;                                         // 关闭结果集
            stmt.close() ;                                       // 操作关闭
            conn.close() ;                                       // 数据库关闭
        }
    }
```

程序运行结果：

```
编号：2；姓名：MLDN；密码：www.mldnjava.cn；年龄：5；性别：女；生日：2000-08-27
--------------------------
编号：4；姓名：李兴华；密码：www.mldn.cn；年龄：30；性别：男；生日：2008-08-27
--------------------------
编号：5；姓名：李康；密码：www.mldnjava.cn；年龄：23；性别：女；生日：2003-08-27
--------------------------
```

在执行查询语句时，是将数据库中的查询结果返回到内存中，所以 rs.next()的作用是将返回的结果依次判断，如果有结果，则使用 getXxx()语句的形式将内容取出。

提示：ResultSet 中的所有数据都可以通过 getString()方法取得。

String 可以接收表中任意类型列的内容，所以在以上程序中全部都使用 getString()接收是没有任何问题的。

如果感觉以上代码在取出数据时输入列的名称比较麻烦，则可以按取值的顺序采用编号的形式将内容取出。

```
            while(rs.next()){                                    // 指针向下移动
                int id = rs.getInt(1) ;                          // 取得id内容
                String name = rs.getString(2) ;                  // 取得name内容
                String pass = rs.getString(3) ;                  // 取得password内容
                int age = rs.getInt(4) ;                         // 取得age内容
                String sex = rs.getString(5) ;                   // 取得sex内容
```

```
            java.util.Date d = rs.getDate(6) ;          // 取得birthday内容
            ...
        }
```

以上两个操作的结果是完全一样的，都是将数据表中的数据取出来，但是很明显第二种操作更简单，所以一般建议使用第二种方式进行操作。

提问：查询时能不能使用“*”？

在以上程序中如果直接使用“SELECT * FROM user”的方式不是更简单吗？为什么还要明确地写出查询列？

回答：开发中必须明确地写出查询列。

直接写成按“*”形式，查询出所有列本身没有任何的问题，但是这样一来从查询语句上就很难知道所要的具体列是什么，在使用 ResultSet 取内容时会比较麻烦，所以开发中建议不要使用“*”的方式查询。

17.8 PreparedStatement 接口

17.8.1 PreparedStatement 简介

PreparedStatement 是 Statement 的子接口，属于预处理操作，与直接使用 Statement 不同的是，PreparedStatement 在操作时，是先在数据表中准备好了一条 SQL 语句，但是此 SQL 语句的具体内容暂时不设置，而是之后再进行设置。以插入数据为例，使用 PreparedStatement 插入数据时，数据表中的指针首先指向最后一条数据之后，但里面的内容是不知道的，而是等待用户分别设置，如图 17-33 所示。

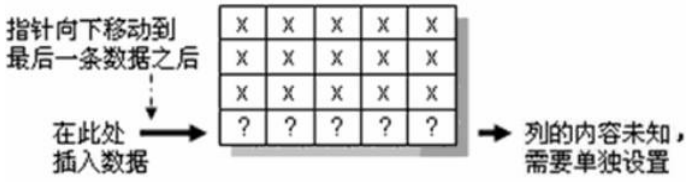


图 17-33 使用 PreparedStatement 插入数据的操作

提示：关于预处理的解释。

预处理的操作实际上与在实际生活中占座的道理是一样，A 帮 B 占座，但是此时 B 没有来，但不管 B 是否来，A 都会把这个座位先占着，等待 B 的到来。

由于 PreparedStatement 对象已预编译过，所以其执行速度要高于 Statement 对象。因此，对于需要多次执行的 SQL 语句经常使用 PreparedStatement 对象操作，以提高效率。

在 PreparedStatement 中执行的 SQL 语句与之前并没有什么不同，但是对于具体的内容是采用“?”的占位符形式出现的，设置时要按照“?”的顺序设置具体的内容。

PreparedStatement 除了继承 Statement 的所有操作之外，自己还增加了许多新的操作。

下面先介绍 PreparedStatement 的基本操作方法，如表 17-8 所示。

表 17-8　PreparedStatement 的基本操作方法

序号	方　法	类型	描　述
1	int executeUpdate() throws SQLException	普通	执行设置的预处理 SQL 语句
2	ResultSet executeQuery() throws SQLException	普通	执行数据库查询操作，返回 ResultSet
3	void setInt(int parameterIndex,int x) throws SQLException	普通	指定要设置的索引编号，并设置整数内容
4	void setFloat(int parameterIndex,float x) throws SQLException	普通	指定要设置的索引编号，并设置浮点数内容
5	void setString(int parameterIndex,String x) throws SQLException	普通	指定要设置的索引编号，并设置字符串内容
6	void setDate(int parameterIndex,Date x) throws SQLException	普通	指定要设置的索引编号，并设置 java.sql.Date 类型的日期内容

以上方法中特别要提醒读者注意的是 setDate()方法，此方法可以设置日期内容，但是使用时，后面的 Date 类型变量是 java.sql.Date，而不是 java.util.Date，所以如果要将一个 java.util.Date 类型的内容变为 java.sql.Date 类型的内容应该使用如下的语句形式：

```
String birthday = "2007-08-27";                    // 生日
java.util.Date temp = null ;                       // 声明一个Date对象
// 通过SimpleDateFormat类将一个字符串变为java.util.Date类型
temp = new SimpleDateFormat("yyyy-MM-dd").parse(birthday) ;
// 通过java.util.Date取出具体的日期数，并将其变为java.sql.Date类型
java.sql.Date bir = new java.sql.Date(temp.getTime()) ;
```

以上代码使用了 SimpleDateFormat 类将一个字符串变为 java.util.Date 类型，但是因为此类型无法直接在 PreparedStatement 对象上使用，所以调用 java.util.Date 类的 getTime()方法，将其内容变为 java.sql.Date 类型的数据。

17.8.2　使用 PreparedStatement 执行数据库操作

下面使用 PreparedStatement 完成数据的增加和查询操作。

范例：使用 PreparedStatement 完成数据插入操作

```
package org.lxh.demo17.preparedstatementdemo;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.text.SimpleDateFormat;
public class PreparedStatementDemo01 {
    // 定义MySQL的数据库驱动程序
    public static final String DBDRIVER = "org.gjt.mm.mysql.Driver" ;
    // 定义MySQL数据库的连接地址
```

```
    public static final String DBURL = "jdbc:mysql://localhost:3306/mldn" ;
    // MySQL数据库的连接用户名
    public static final String DBUSER = "root" ;
    // MySQL数据库的连接密码
    public static final String DBPASS = "mysqladmin" ;
    public static void main(String[] args) throws Exception {
        Connection conn = null ;                    // 数据库连接
        PreparedStatement pstmt = null ;            // 数据库操作
        String name = "李兴华";                      // 姓名
        String password = "www.mldnjava.cn";        // 密码
        int age = 30;                               // 年龄
        String sex = "男";                          // 性别
        String birthday = "2007-08-27";             // 生日
        java.util.Date temp = null ;                // 声明一个Date对象
        // 通过SimpleDateFormat类将一个字符串变为java.util.Date类型
        temp = new SimpleDateFormat("yyyy-MM-dd").parse(birthday) ;
        // 通过java.util.Date取出具体的日期数，并将其变为java.sql.Date类型
        java.sql.Date bir = new java.sql.Date(temp.getTime()) ;
        String sql = "INSERT INTO user(name,password,age,sex,birthday) "
                + " VALUES (?,?,?,?,?)";            // 编写预处理SQL
        Class.forName(DBDRIVER) ;                   // 加载驱动程序
        // 连接MySQL数据库时，要写上连接的用户名和密码
        conn = DriverManager.getConnection(DBURL, DBUSER, DBPASS);
        pstmt = conn.prepareStatement(sql) ;        // 实例化PreparedStatement
        pstmt.setString(1, name) ;                  // 设置第一个“?”的内容
        pstmt.setString(2,password) ;               // 设置第二个“?”的内容
        pstmt.setInt(3,age) ;                       // 设置第三个“?”的内容
        pstmt.setString(4,sex) ;                    // 设置第四个“?”的内容
        pstmt.setDate(5,bir) ;                      // 设置第五个“?”的内容
        pstmt.executeUpdate() ;                     // 执行数据库更新操作，不需要SQL
        pstmt.close() ;                             // 操作关闭
        conn.close() ;                              // 数据库关闭
    }
}
```

执行完后可以发现，此时已经完成了数据的增加操作，如图 17-34 所示。

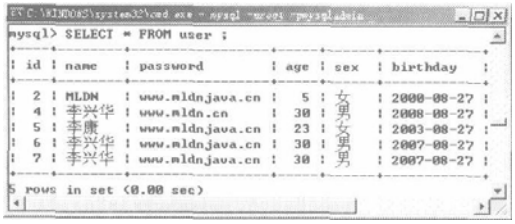


图 17-34　使用预处理增加数据

从程序中可以发现，预处理就是使用“?”进行占位的，如下所示：

```
String sql = "INSERT INTO user(name,password,age,sex,birthday) "
        + " VALUES (?,?,?,?,?)";                    // 编写预处理SQL
```

每一个“?”都对应着一个具体的字段，在设置时，按照“?”的顺序设置即可，第一个“?”对应的是 name 的内容，第二个“?”对应的是 password 内容，依此类推。

完成插入操作后，下面介绍如何进行查询操作，为了说明问题，本部分将使用模糊查询的方式编写程序代码。

范例：模糊查询

```
package org.lxh.demo17.preparedstatementdemo;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
public class PreparedStatementDemo02 {
    // 定义MySQL的数据库驱动程序
    public static final String DBDRIVER = "org.gjt.mm.mysql.Driver" ;
    // 定义MySQL数据库的连接地址
    public static final String DBURL = "jdbc:mysql://localhost:3306/mldn" ;
    // MySQL数据库的连接用户名
    public static final String DBUSER = "root" ;
    // MySQL数据库的连接密码
    public static final String DBPASS = "mysqladmin" ;
    public static void main(String[] args) throws Exception {
        Connection conn = null ;                        // 数据库连接
        PreparedStatement pstmt = null ;                // 数据库操作
        String keyWord = "李" ;                         // 设置查询关键字
        ResultSet rs = null ;                           // 保存查询结果
        String sql = "SELECT id,name,password,age,sex,birthday "+
            " FROM user WHERE name LIKE ? OR password LIKE ? OR sex LIKE ?" ;
        Class.forName(DBDRIVER) ;                       // 加载驱动程序
        // 连接MySQL数据库时，要写上连接的用户名和密码
        conn = DriverManager.getConnection(DBURL, DBUSER, DBPASS);
        pstmt = conn.prepareStatement(sql) ;            // 实例化对象
        pstmt.setString(1, "%" + keyWord + "%");        // 设置第一个“?”，要使用
                                                        模糊查询
        pstmt.setString(2, "%" + keyWord + "%");        // 设置第二个“?”，要使用
                                                        模糊查询
        pstmt.setString(3, "%" + keyWord + "%");        // 设置第三个“?”，要使用
                                                        模糊查询
        rs = pstmt.executeQuery() ;                     // 实例化ResultSet对象
        while(rs.next()){                               // 指针向下移动
```

```
            int id = rs.getInt(1) ;                    // 取得id内容
            String name = rs.getString(2) ;            // 取得name内容
            String pass = rs.getString(3) ;            // 取得password内容
            int age = rs.getInt(4) ;                   // 取得age内容
            String sex = rs.getString(5) ;             // 取得sex内容
            java.util.Date d = rs.getDate(6) ;         // 取得birthday内容
            System.out.print("编号: " + id + "; ") ;
            System.out.print("姓名: " + name + "; ") ;
            System.out.print("密码: " + pass + "; ") ;
            System.out.print("年龄: " + age + "; ") ;
            System.out.print("性别: " + sex + "; ") ;
            System.out.println("生日: " + d) ;
            System.out.println("---------------------------");
        }
        rs.close() ;                                   // 关闭结果集
        pstmt.close() ;                                // 操作关闭
        conn.close() ;                                 // 数据库关闭
    }
}
```

程序运行结果：

```
编号: 4; 姓名: 李兴华; 密码: www.mldn.cn; 年龄: 30; 性别: 男; 生日: 2008-08-27
---------------------------
编号: 5; 姓名: 李康; 密码: www.mldnjava.cn; 年龄: 23; 性别: 女; 生日: 2003-08-27
---------------------------
编号: 6; 姓名: 李兴华; 密码: www.mldnjava.cn; 年龄: 30; 性别: 男; 生日: 2007-08-27
---------------------------
编号: 7; 姓名: 李兴华; 密码: www.mldnjava.cn; 年龄: 30; 性别: 男; 生日: 2007-08-27
---------------------------
```

以上程序进行的是模糊查询，在模糊查询中使用“%”表示通配符，但是这个通配符是要在设置具体查询内容（setXxx()方法调用）时才使用。如果此时只是查询全部，则不用再设置任何的内容，如下所示：

```
String sql = "SELECT id,name,password,age,sex,birthday "+
    " FROM user" ;                                 // 此处不需要设置任何内容
...
pstmt = conn.prepareStatement(sql) ;               // 实例化对象
rs = pstmt.executeQuery() ;                        // 实例化ResultSet对象
```

以上只列出了部分的片段代码，因为现在是查询全部的数据，所以在查询时不需要设置任何的参数，所以此处不需要调用 PreparedStatement 中的 setXxx()方法进行内容的设置。

提问：到底是使用 Statement 还是 PreparedStatement 呢？

这两个对象的操作目的都是一样的，那么在开发中到底使用哪一个更好？

回答：开发中要使用 PreparedStatement。

在开发中是很少使用 Statement 对象进行操作的，因为 Statement 执行的是一个完整的 SQL 语句，这样在程序中往往要使用拼凑的 SQL 语句完成。而且如果此时由用户自己输入数据，往往会出现输入非法字符而造成程序出错，也可能引起系统的安全漏洞，所以开发中不建议使用 Statement 完成，而都是使用 PreparedStatement 完成操作的。

17.9　处理大数据对象

大对象处理主要指的是 CLOB 和 BLOB 两种类型的字段。在 CLOB 中可以存储海量文字，例如，存储一部《三国演义》或者是《红楼梦》等；在 BLOB 中可以存储二进制数据，如图片、电影等。如果在程序中要想处理这样的大对象操作，则必须使用 PreparedStatement 完成，所有的内容要通过 IO 流的方式从大文本字段中保存和读取。大对象字段的 IO 操作如图 17-35 所示。

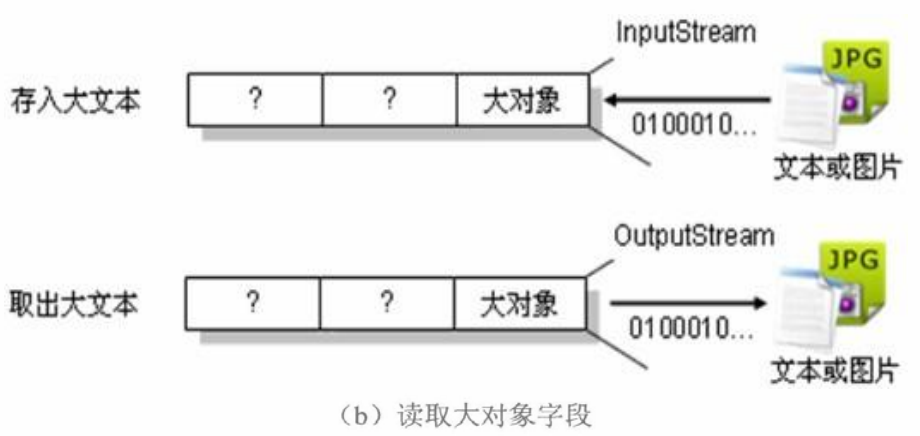


图 17-35　大对象字段的 IO 操作

PreparedStatement 提供了如表 17-9 所示的方法，专门用于写入大对象数据。

表 17-9　写入大对象数据

序号	方　　法	类型	描　　述
1	void setAsciiStream(int parameterIndex,InputStream x, int length) throws SQLException	普通	将指定的输入流写入数据库的文本字段
2	void setBinaryStream(int parameterIndex,InputStream x, int length) throws SQLException	普通	将二进制的输入流数据写入到二进制字段中

大对象设置到数据库中后，在查询时就需要使用 ResultSet 将其读取进来。在 ResultSet 中提供了如表 17-10 所示的几个方法，可以读取出大对象数据。

表 17-10　读取大对象数据

序号	方　　法	类型	描　　述
1	InputStream getAsciiStream(int columnIndex) throws SQLException	普通	根据列的编号返回大对象的文本输入流
2	InputStream getAsciiStream(String columnName) throws SQLException	普通	根据列的名称返回大对象的文本输入流
3	Clob getClob(int i) throws SQLException	普通	根据列的编号返回 Clob 数据
4	Clob getClob(String colName) throws SQLException	普通	根据列名称返回 Clob 数据
5	InputStream getBinaryStream(int columnIndex) throws SQLException	普通	根据列的编号返回二进制数据
6	InputStream getBinaryStream(String columnName) throws SQLException	普通	根据列的名称返回二进制数据
7	Blob getBlob(int i) throws SQLException	普通	根据列的编号返回 Blob 数据
8	Blob getBlob(String colName) throws SQLException	普通	根据列名称返回 Blob 数据

以上方法实际上分为两组，Clob 和 AsciiStream 一般都返回大文本，而 Blob 和 BinaryStream 一般都返回二进制数据。下面分别来看这些方法的操作，而 Clob 和 Blob 也将在操作时为读者介绍。

17.9.1　处理 CLOB 数据

CLOB 表示大文本数据，在 MySQL 中提供了 LONGTEXT 表示大文本数据，此字段的最大保存数据量为 4GB。例如，有如下的数据库创建脚本：

```
DROP TABLE userclob ;
CREATE TABLE userclob
(
    id          INT             AUTO_INCREMENT  PRIMARY KEY ,
    name        VARCHAR(30)     NOT NULL ,
    note        LONGTEXT
) ;
```

下面向此表中插入数据，其中对于 note 的内容，使用如图 17-36 所示的文本表示，此文本的大小是 16KB，保存在 d 盘中，文件名称是 mldn.txt。

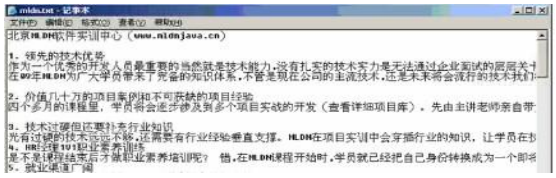


图 17-36　mldn.txt 文本的内容

范例：写入大文本数据

```
package org.lxh.demo17.lobdemo;
import java.io.File;
```

```
import java.io.FileInputStream;
import java.io.InputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
public class ClobDemo01 {
    // 定义MySQL的数据库驱动程序
    public static final String DBDRIVER = "org.gjt.mm.mysql.Driver" ;
    // 定义MySQL数据库的连接地址
    public static final String DBURL = "jdbc:mysql://localhost:3306/mldn" ;
    // MySQL数据库的连接用户名
    public static final String DBUSER = "root" ;
    // MySQL数据库的连接密码
    public static final String DBPASS = "mysqladmin" ;
    public static void main(String[] args) throws Exception {
        Connection conn = null ;                    // 数据库连接
        PreparedStatement pstmt = null ;            // 数据库操作
        String name = "李兴华";                      // 姓名
        String sql = "INSERT INTO userclob(name,note) VALUES (?,?) " ;
        Class.forName(DBDRIVER) ;                   // 加载驱动程序
        // 连接MySQL数据库时，要写上连接的用户名和密码
        conn = DriverManager.getConnection(DBURL, DBUSER, DBPASS);
        pstmt = conn.prepareStatement(sql) ;    // 实例化PreparedStatement
        // 声明一个File对象，用于找到要操作的大文本文件
        File f = new File("d:" + File.separator + "mldn.txt");
        InputStream input = null ;                  // 通过输入流读取内容
        input = new FileInputStream(f) ;            // 通过输入流读取文件
        pstmt.setString(1, name) ;                  // 设置第一个“?”的内容
        pstmt.setAsciiStream(2, input, (int)f.length()) ;  // 设置输入流
        pstmt.executeUpdate() ;                     // 执行数据库更新操作
        pstmt.close() ;                             // 操作关闭
        conn.close() ;                              // 数据库关闭
    }
}
```

因为内容保存在文件中，所以使用 FileInputStream 类将文本文件读取进来，之后直接通过 PreparedStatement 对象将其写入到对应的大文本字段中。

写入完成后，下面直接使用 ResultSet 将其读取进来，因为写入时是按照输入流的方式写入的，所以此时也需要按照输入流的方式读取进来。

范例：读取大文本字段

```
package org.lxh.demo17.lobdemo;
import java.io.InputStream;
```

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.Scanner;
public class ClobDemo02 {
    // 定义MySQL的数据库驱动程序
    public static final String DBDRIVER = "org.gjt.mm.mysql.Driver" ;
    // 定义MySQL数据库的连接地址
    public static final String DBURL = "jdbc:mysql://localhost:3306/mldn" ;
    // MySQL数据库的连接用户名
    public static final String DBUSER = "root" ;
    // MySQL数据库的连接密码
    public static final String DBPASS = "mysqladmin" ;
    public static void main(String[] args) throws Exception {
        Connection conn = null ;                    // 数据库连接
        PreparedStatement pstmt = null ;            // 数据库操作
        ResultSet rs = null ;                       // 保存结果集
        int id = 1 ;                                // id
        String sql = "SELECT name,note FROM userclob WHERE id=?" ;
        Class.forName(DBDRIVER) ;                   // 加载驱动程序
        // 连接MySQL数据库时，要写上连接的用户名和密码
        conn = DriverManager.getConnection(DBURL, DBUSER, DBPASS);
        pstmt = conn.prepareStatement(sql) ;        // 实例化PreparedStatement
        pstmt.setInt(1, id) ;                       // 设置查询的id
        rs = pstmt.executeQuery() ;                 // 查询
        if(rs.next()){                              // 判断是否有内容
            String name = rs.getString(1) ;         // 取出name列的内容
            StringBuffer note = new StringBuffer() ;
            System.out.println("姓名：" + name);
            InputStream input = rs.getAsciiStream(2) ;      // 接收全部的大
                                                               文本数据
            Scanner scan = new Scanner(input) ;// 接收数据
            scan.useDelimiter("\r\n") ;             // 将文件换行作为分割符
            while(scan.hasNext()){
                note.append(scan.next()).append("\n") ;     // 不断读取内容
            }
            System.out.println("内容：" + note) ;// 输出内容
            input.close() ;
        }
        pstmt.close() ;                             // 操作关闭
        conn.close() ;                              // 数据库关闭
```

```
        }
    }
```

程序运行结果（部分）：

```
姓名：李兴华
内容：北京MLDN软件实训中心（www.mldnjava.cn）

1．领先的技术优势
作为一个优秀的开发人员最重要的当然就是技术能力，没有扎实的技术实力是无法通过企业面试的层层关卡的，
在09年MLDN为广大学员带来了完备的知识体系，不管是现在公司的主流技术，还是未来将会流行的技术我们将一网打尽，为大家带来为期四个多月的Java饕餮盛宴。

2．价值几十万的项目案例和不可或缺的项目经验
...
```

以上程序中，为了操作方便，使用了 Scanner 类接收全部的输入内容。由于在文本保存时存在换行，所以使用“\r\n”作为分割符，之后通过循环的方式不断地把内容取出并将内容保存在 StringBuffer 对象中。

以上的做法是将大文本数据内容直接通过 ResultSet 读取进来，当然也可以使用 ResultSet 中提供的 getClob()方法，将全部的内容变为 Clob 对象的内容。直接使用 Clob 可以方便地取得大文本的数据，也可以对这些文本数据进行一些简单的操作，如截取指定长度的文本等。Clob 的常用方法如表 17-11 所示。

表 17-11　Clob 类的常用方法

序号	方　法	类型	描　述
1	InputStream getAsciiStream() throws SQLException	普通	返回输入流对象
2	String getSubString(long pos,int length) throws SQLException	普通	从 Clob 中得到指定范围的字符串
3	long length() throws SQLException	普通	返回 Clob 中的字符数量
4	void truncate(long len) throws SQLException	普通	截取指定长度的大文本

范例：使用 Clob 读取内容

```
package org.lxh.demo17.lobdemo;
import java.sql.Clob;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
public class ClobDemo03 {
    // 定义MySQL的数据库驱动程序
    public static final String DBDRIVER = "org.gjt.mm.mysql.Driver" ;
    // 定义MySQL数据库的连接地址
```

```
    public static final String DBURL = "jdbc:mysql://localhost:3306/mldn" ;
    // MySQL数据库的连接用户名
    public static final String DBUSER = "root" ;
    // MySQL数据库的连接密码
    public static final String DBPASS = "mysqladmin" ;
    public static void main(String[] args) throws Exception {
        Connection conn = null ;                    // 数据库连接
        PreparedStatement pstmt = null ;            // 数据库操作
        ResultSet rs = null ;                       // 保存结果集
        int id = 1 ;                                // id
        String sql = "SELECT name,note FROM userclob WHERE id=?" ;
        Class.forName(DBDRIVER) ;                   // 加载驱动程序
        // 连接MySQL数据库时，要写上连接的用户名和密码
        conn = DriverManager.getConnection(DBURL, DBUSER, DBPASS);
        pstmt = conn.prepareStatement(sql) ;        // 实例化PreparedStatement
        pstmt.setInt(1, id) ;                       // 设置查询的id
        rs = pstmt.executeQuery() ;                 // 查询
        if(rs.next()){                              // 判断是否有内容
            String name = rs.getString(1) ;         // 取出name列的内容
            Clob c = rs.getClob(2) ;                // 取出大文本数据
            String note = c.getSubString(1, (int)c.length()) ;
            System.out.println("姓名：" + name);
            System.out.println("内容：" + note) ;
            c.truncate(100) ;                       // 读取100个长度的内容
            System.out.println("部分读取内容：" + c.getSubString(1, (int)c.
            length())) ;
        }
        pstmt.close() ;                             // 操作关闭
        conn.close() ;                              // 数据库关闭
    }
}
```

程序运行结果（部分）：

```
姓名：李兴华
内容：北京MLDN软件实训中心（www.mldnjava.cn）

1．领先的技术优势
...以下部分内容省略
部分读取内容：北京MLDN软件实训中心（www.mldnjava.cn）

1．领先的技术优势
作为一个优秀的开发人员最重要的当然就是技术能力，没有扎实的技术实力是无法通过企业面试的层层关卡的，
在09年
```

以上程序同样完成了读取大对象的操作，但是从代码来看，直接使用 Clob 操作要比使用 InputStream 方便很多。

17.9.2　处理 BLOB 数据

BLOB 的操作与 CLOB 相似，只是 BLOB 专门用于存放二进制数据，如图片、电影等，下面通过 BLOB 进行图片的保存与读取。在 MySQL 中使用 LONGBLOB 声明，最高可以保存 4GB 大小的内容。例如，有如下的数据库创建脚本：

```
DROP TABLE userblob ;
CREATE TABLE userblob
(
    id          INT             AUTO_INCREMENT  PRIMARY KEY ,
    name        VARCHAR(30)     NOT NULL ,
    photo       LONGBLOB
) ;
```

下面就向表中插入数据，对于 photo 字段的内容使用如图 17-37 所示的图片表示，此图片保存在 d 盘中。



图 17-37　mldn.gif 图片

范例：将图片写入到数据表中

```
package org.lxh.demo17.lobdemo;
import java.io.File;
import java.io.FileInputStream;
import java.io.InputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
public class BlobDemo01 {
    // 定义MySQL的数据库驱动程序
    public static final String DBDRIVER = "org.gjt.mm.mysql.Driver" ;
    // 定义MySQL数据库的连接地址
    public static final String DBURL = "jdbc:mysql://localhost:3306/mldn" ;
    // MySQL数据库的连接用户名
    public static final String DBUSER = "root" ;
    // MySQL数据库的连接密码
    public static final String DBPASS = "mysqladmin" ;
```

```
    public static void main(String[] args) throws Exception {
        Connection conn = null ;                    // 数据库连接
        PreparedStatement pstmt = null ;            // 数据库操作
        String name = "李兴华";                      // 姓名
        String sql = "INSERT INTO userblob(name,photo) VALUES (?,?) " ;
        Class.forName(DBDRIVER) ;                   // 加载驱动程序
        // 连接MySQL数据库时，要写上连接的用户名和密码
        conn = DriverManager.getConnection(DBURL, DBUSER, DBPASS);
        pstmt = conn.prepareStatement(sql) ;        // 实例化PreparedStatement
        // 声明一个File对象，用于找到要操作的大文本文件
        File f = new File("d:" + File.separator + "mldn.gif");
        InputStream input = null ;                  // 通过输入流读取内容
        input = new FileInputStream(f) ;            // 通过输入流读取文件
        pstmt.setString(1, name) ;                  // 设置第一个“?”的内容
        pstmt.setBinaryStream(2, input, (int) f.length()); // 设置输入流
        pstmt.executeUpdate() ;                     // 执行数据库更新操作
        pstmt.close() ;                             // 操作关闭
        conn.close() ;                              // 数据库关闭
    }
}
```

程序执行完毕后，图片的信息就以二进制数据的形式保存在数据表中，如果直接向数据库中发出查询指令，则只能显示一些二进制代码，而图片无法显示。

下面直接使用 ResultSet 中的方法直接从数据库中读出所要的图片。因为是在命令行方式下运行程序的，所以读取出来的图片无法直接显示，此时，图片的保存路径为 d:\loadmldn.gif。

范例：读取内容，并将图片信息保存

```
package org.lxh.demo17.lobdemo;
import java.io.File;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
public class BlobDemo02 {
    // 定义MySQL的数据库驱动程序
    public static final String DBDRIVER = "org.gjt.mm.mysql.Driver" ;
    // 定义MySQL数据库的连接地址
    public static final String DBURL = "jdbc:mysql://localhost:3306/mldn" ;
    // MySQL数据库的连接用户名
    public static final String DBUSER = "root" ;
```

```
    // MySQL数据库的连接密码
    public static final String DBPASS = "mysqladmin" ;
    public static void main(String[] args) throws Exception {
        Connection conn = null ;                    // 数据库连接
        PreparedStatement pstmt = null ;            // 数据库操作
        ResultSet rs = null ;                       // 保存结果集
        int id = 1 ;                                // id
        String sql = "SELECT name,photo FROM userblob WHERE id=?" ;
        Class.forName(DBDRIVER) ;                   // 加载驱动程序
        // 连接MySQL数据库时，要写上连接的用户名和密码
        conn = DriverManager.getConnection(DBURL, DBUSER, DBPASS);
        pstmt = conn.prepareStatement(sql) ;        // 实例化PreparedStatement
        pstmt.setInt(1, id) ;                       // 设置查询的id
        rs = pstmt.executeQuery() ;                 // 查询
        if(rs.next()){                              // 判断是否有内容
            String name = rs.getString(1) ;         // 取出name列的内容
            System.out.println("姓名: " + name);
            InputStream input = rs.getBinaryStream(2) ;     // 接收全部的大
                                                            文本数据
            FileOutputStream out = null ;
            out = new FileOutputStream(new File("d:" +
                File.separator + "loadmldn.gif"));
            int temp = 0 ;
            while((temp=input.read())!=-1){         // 边读边写
                out.write(temp) ;
            }
            input.close() ;
            out.close() ;
        }
        pstmt.close() ;                             // 操作关闭
        conn.close() ;                              // 数据库关闭
    }
}
```

程序正常执行完毕后，可以直接在 f 盘下发现 loadmldn.gif 图片。与大文本操作类似，在 JDBC 中也提供了专门的 Blob 类型，用于表示二进制数据。Blob 类的常用方法如表 17-12 所示。

表 17-12　Blob 类的常用方法

序号	方　法	类型	描　述
1	InputStream getBinaryStream() throws SQLException	普通	取得 Blob 的输入流
2	byte[] getBytes(long pos,int length) throws SQLException	普通	以字节数组的形式返回 Blob 数据的内容
3	long length() throws SQLException	普通	取出数据的长度

下面直接使用 Blob 类完成读取 Blob 字段的操作，本操作的功能与之前一样，还是将内容保存在硬盘上。

范例：使用 Blob 读取内容

```
package org.lxh.demo17.lobdemo;
import java.io.File;
import java.io.FileOutputStream;
import java.sql.Blob;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
public class BlobDemo03 {
    // 定义MySQL的数据库驱动程序
    public static final String DBDRIVER = "org.gjt.mm.mysql.Driver" ;
    // 定义MySQL数据库的连接地址
    public static final String DBURL = "jdbc:mysql://localhost:3306/mldn" ;
    // MySQL数据库的连接用户名
    public static final String DBUSER = "root" ;
    // MySQL数据库的连接密码
    public static final String DBPASS = "mysqladmin" ;
    public static void main(String[] args) throws Exception {
        Connection conn = null ;                    // 数据库连接
        PreparedStatement pstmt = null ;            // 数据库操作
        ResultSet rs = null ;                       // 保存结果集
        int id = 1 ;                                // id
        String sql = "SELECT name,photo FROM userblob WHERE id=?" ;
        Class.forName(DBDRIVER) ;                   // 加载驱动程序
        // 连接MySQL数据库时，要写上连接的用户名和密码
        conn = DriverManager.getConnection(DBURL, DBUSER, DBPASS);
        pstmt = conn.prepareStatement(sql) ;        // 实例化PreparedStatement
        pstmt.setInt(1, id) ;                       // 设置查询的id
        rs = pstmt.executeQuery() ;                 // 查询
        if(rs.next()){                              // 判断是否有内容
            String name = rs.getString(1) ;         // 取出name列的内容
            System.out.println("姓名: " + name);
            Blob b = rs.getBlob(2) ;                // 读取Blob数据
            FileOutputStream out = null ;
            out = new FileOutputStream(new File("d:" +
                    File.separator + "loadmldn.gif"));
            out.write(b.getBytes(1, (int)b.length())) ;    // 保存内容
            out.close() ;
        }
```

```
            pstmt.close() ;                                    // 操作关闭
            conn.close() ;                                     // 数据库关闭
        }
    }
```

在这里要提醒读者的是，虽然在 JDBC 中提供了大对象的操作支持，但是如果内容太大（如电影），则不建议使用以上方式进行保存。可以通过做映射路径的方式保存具体的信息，即在数据库中使用一个 VARCHAR 的普通字段保存一个文件的文件名称，且文件直接保存在硬盘上。

17.10　CallableStatement 接口

CallableStatement 主要是调用数据库中的存储过程，CallableStatement 也是 Statement 接口的子接口。在使用 CallableStatement 时可以接收过程的返回值，此接口的常用方法如表 17-13 所示。

表 17-13　CallableStatement 接口的常用方法

序号	方　法	类型	描　述
1	int getInt(int parameterIndex) throws SQLException	普通	根据编号取出过程的返回值
2	float getFloat(int parameterIndex) throws SQLException	普通	根据编号取出过程的返回值
3	void setInt(String parameterName,int x) throws SQLException	普通	设置指定编号的内容
4	void setFloat(String parameterName,float x) throws SQLException	普通	设置指定编号的内容
5	void registerOutParameter(int parameterIndex,int sqlType) throws SQLException	普通	设置返回值的类型，需要使用 Types 类

在 JDBC 中，如果要设置过程的返回值类型，可以使用 Types 完成。在 Types 中定义了很多的常量，如果现在返回值类型为 int，则使用 Types.INTEGER。

下面具体看一下 JDBC 操作过程的演示，首先在 MySQL 中建立一个叫 myproc 的过程，如图 17-38 所示。

```
DELIMITER //                          -- 改变分割符
DROP PROCEDURE myproc //              -- 删除过程
CREATE PROCEDURE myproc(IN p1 int,INOUT p2 int,OUT p3 int)
BEGIN
    SELECT p1,p2,p3 ;                 -- 输出p1、p2、p3的内容
    SET p1=10 ;                       -- 设置p1的内容为10
    SET p2=20 ;                       -- 设置p2的内容为20
    SET p3=30 ;                       -- 设置p3的内容为30
END
//
```

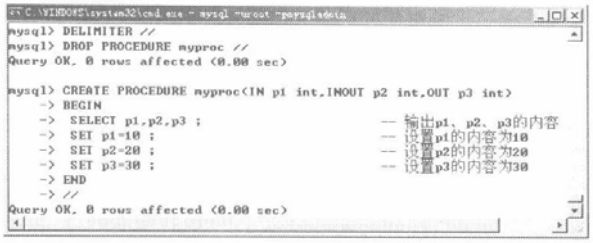


图 17-38　建立 mysql 存储过程

在 myproc 的过程中定义了 3 个变量，分别使用了 IN、INOUT、OUT 3 种类型声明，这 3 种类型的意义如下。

- IN 类型（默认的设置）：表示只是将值传递进来。
- INOUT 类型：表示把值传递到过程中，可以保留过程对此值的修改值。
- OUT 类型：可以不用传递内容进来，过程中对此值的操作可以返回。

下面对以上过程进行测试，定义两个变量 x1、x2。调用建立好的 myproc 过程如图 17-39 所示。

```
DELIMITER ;                          -- 改变分割符
SET @x1=70 ;                         -- 定义变量x1的内容为70
SET @x2=80 ;                         -- 定义变量x2的内容为80
CALL myproc(@x1,@x2,@x3) ;           -- 调用过程
```

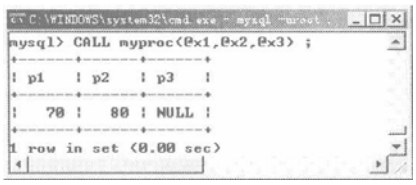


图 17-39　调用建立好的 myproc 过程

在以上调用过程时，将原本的 3 个变量内容进行输出，过程完成之后再次查询 3 个变量的内容，如图 17-40 所示。

```
SELECT @x1,@x2,@x3 ;
```

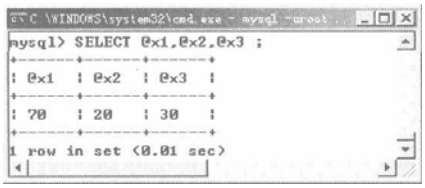


图 17-40　查询过程执行完之后的变量内容

从查询结果中可以发现，IN 操作只是将值传递进去，而本身没有任何修改；而 INOUT 操作，既可以传递值也可以被过程修改；OUT 操作中即使没有传递具体内容，但也可以将过程中修改后的内容取出。

范例：调用 myproc 存储过程

```
package org.lxh.demo17.procdemo;
import java.sql.CallableStatement;
```

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Types;
public class ProcDemo {
    // 定义MySQL的数据库驱动程序
    public static final String DBDRIVER = "org.gjt.mm.mysql.Driver" ;
    // 定义MySQL数据库的连接地址
    public static final String DBURL = "jdbc:mysql://localhost:3306/mldn" ;
    // MySQL数据库的连接用户名
    public static final String DBUSER = "root" ;
    // MySQL数据库的连接密码
    public static final String DBPASS = "mysqladmin" ;
    public static void main(String[] args) throws Exception {
        Connection conn = null ;                          // 数据库连接
        CallableStatement cstmt = null ;                  // 数据库操作
        String sql = "{CALL myproc(?,?,?)}" ;             // 调用过程
        Class.forName(DBDRIVER) ;                         // 加载驱动程序
        // 连接MySQL数据库时，要写上连接的用户名和密码
        conn = DriverManager.getConnection(DBURL, DBUSER, DBPASS);
        cstmt = conn.prepareCall(sql) ;                   // 实例化对象
        cstmt.setInt(1, 70) ;                             // 设置第一个参数是70
        cstmt.setInt(2, 80) ;                             // 设置第二个参数是80
        cstmt.registerOutParameter(2, Types.INTEGER) ; // 设置返回值类型
        cstmt.registerOutParameter(3, Types.INTEGER) ; // 设置返回值类型
        cstmt.execute() ;                                 // 执行存储过程
        System.out.println("INOUT的返回值：" + cstmt.getInt(2));
        System.out.println("OUT的返回值：" + cstmt.getInt(3));
        cstmt.close() ;                                   // 操作关闭
        conn.close() ;                                    // 数据库关闭
    }
}
```

程序运行结果：

```
INOUT的返回值：20
OUT的返回值：30
```

同直接调用数据库的过程一样，INOUT 和 OUT 都将过程中修改后的内容带了出来。

17.11　JDBC 2.0 操作

之前所讲解的大部分操作都是属于最基本的 JDBC 操作，在 JDBC 2.0 之后为了方便操作者进行数据库的开发提供了许多更加方便的操作，包括可滚动的结果集和使用结果集直

接更新数据库。如果要使用这样的特性则必须依靠 ResultSet，看一下 JDBC 2.0 对 ResultSet 的新支持，如表 17-14 所示。

表 17-14 JDBC 2.0 对 ResultSet 的新支持

序号	方法及常量	类型	描 述
1	static final int TYPE_FORWARD_ONLY	常量	表示指针只能向前移动的 ResultSet，是默认值
2	static final int TYPE_SCROLL_SENSITIVE	常量	表示 ResultSet 可以滚动，可以更新内容
3	static final int TYPE_SCROLL_INSENSITIVE	常量	表示 ResultSet 可以滚动，但是不能更新内容
4	static final int CONCUR_READ_ONLY	常量	按只读的方式打开数据库
5	static final int CONCUR_UPDATABLE	常量	表示 ResultSet 可以更新
6	boolean absolute(int row) throws SQLException	普通	将结果集移到指定行
7	void afterLast() throws SQLException	普通	将结果集移动到末尾之后
8	void beforeFirst() throws SQLException	普通	将结果集移动到首行之前
9	boolean first() throws SQLException	普通	将结果集移动到第一行
10	boolean last() throws SQLException	普通	将结果集移动到最后一行
11	boolean previous() throws SQLException	普通	将结果集向上移动
12	void updateString(int columnIndex,String x) throws SQLException	普通	指定更新列的内容，此方法被重载多次，支持各种数据类型
13	void updateString(String columnName,String x) throws SQLException	普通	指定更新列的内容，此方法被重载多次，支持各种数据类型
14	void moveToInsertRow() throws SQLException	普通	将指针移动到插入行
15	void updateRow() throws SQLException	普通	更新行数据信息
16	void cancelRowUpdates() throws SQLException	普通	取消更新数据，在 updateRow()调用之前有效
17	void insertRow() throws SQLException	普通	插入行数据
18	void deleteRow() throws SQLException	普通	删除行数据

以上列出了 ResultSet 与 JDBC 2.0 有关的操作，当然，JDBC 2.0 不止包括这些，在其中还有一个最重要的操作就是可以进行批处理操作。

提示：关于 ResultSet 常量的补充说明。

ResultSet 中提供了很多常量，但是其整体划分为两类。

- 设置 ResultSet 类型：TYPE_XXX 设置的都是类型，类型主要表示其是否可以滚动以及是否可以修改数据表的内容。
- 设置并发性：CONCUR_XXX 设置的都是并发性，并发性主要表示结果集是否是只读还是可以进行数据库更新操作等。

17.11.1 可滚动的结果集

在之前所讲解的 ResultSet 操作中，返回的结果只能进行由前向后的顺序输出，如果现在想取结果集中任意位置的数据，则必须使用可滚动的结果集。可滚动的结果集操作本身并不难理解，只是在创建数据库操作对象时加入若干参数即可，如下所示：

```
            conn = DriverManager.getConnection(DBURL, DBUSER, DBPASS);
            pstmt = conn.prepareStatement(sql,
ResultSet.TYPE_SCROLL_SENSITIVE,
                    ResultSet.CONCUR_READ_ONLY);          // 实例化对象
            rs = pstmt.executeQuery() ;                   // 实例化ResultSet对象
```

以上在使用 Connection 创建 PreparedStatement 操作时，除了设置操作的 SQL 语句外，还设置了两个 ResultSet 定义的常量，第一个常量表示结果集可滚动，第二个常量表示以只读的形式打开结果集。

现在假定数据库中只有如图 17-41 所示的 3 条数据。

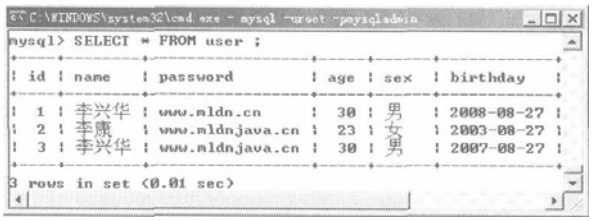


图 17-41　数据库中的数据

范例：让结果集滚动起来

```
package org.lxh.demo17.jdbc20;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
public class JDBC20ReadDemo {
    // 定义MySQL的数据库驱动程序
    public static final String DBDRIVER = "org.gjt.mm.mysql.Driver" ;
    // 定义MySQL数据库的连接地址
    public static final String DBURL = "jdbc:mysql://localhost:3306/mldn" ;
    // MySQL数据库的连接用户名
    public static final String DBUSER = "root" ;
    // MySQL数据库的连接密码
    public static final String DBPASS = "mysqladmin" ;
    public static void main(String[] args) throws Exception {
        Connection conn = null ;                          // 数据库连接
        PreparedStatement pstmt = null ;                  // 数据库操作
        ResultSet rs = null ;                             // 保存查询结果
        String sql = "SELECT id,name,password,age,sex,birthday "+
            " FROM user" ;                                // 此处不需要设置任何内容
        Class.forName(DBDRIVER) ;                         // 加载驱动程序
        // 连接MySQL数据库时，要写上连接的用户名和密码
        conn = DriverManager.getConnection(DBURL, DBUSER, DBPASS);
        pstmt = conn.prepareStatement(sql, ResultSet.TYPE_SCROLL_SENSITIVE,
```

```
                ResultSet.CONCUR_READ_ONLY);          // 实例化对象
        rs = pstmt.executeQuery() ;                   // 实例化ResultSet对象
        System.out.println("第2条数据：") ;
        rs.absolute(1) ;
        print(rs,1) ;
        System.out.println("第1条数据：") ;
        rs.beforeFirst() ;
        print(rs,1) ;
        System.out.println("第3条数据：") ;
        rs.afterLast() ;
        print(rs,-1) ;
        rs.close() ;                                  // 关闭结果集
        pstmt.close() ;                               // 操作关闭
        conn.close() ;                                // 数据库关闭
    }
    public static void print(ResultSet rs,int re) throws Exception{
        if(re>0){
            rs.next() ;                               // 由前向后输出
        }else{
            rs.previous() ;                           // 由后向前输出
        }
        int id = rs.getInt(1) ;                       // 取得id内容
        String name = rs.getString(2) ;               // 取得name内容
        String pass = rs.getString(3) ;               // 取得password内容
        int age = rs.getInt(4) ;                      // 取得age内容
        String sex = rs.getString(5) ;                // 取得sex内容
        java.util.Date d = rs.getDate(6) ;            // 取得birthday内容
        System.out.print("编号：" + id + "；") ;
        System.out.print("姓名：" + name + "；") ;
        System.out.print("密码：" + pass + "；") ;
        System.out.print("年龄：" + age + "；") ;
        System.out.print("性别：" + sex + "；") ;
        System.out.println("生日：" + d) ;
        System.out.println("--------------------------");
    }
}
```

程序运行结果：

```
第2条数据：
编号：2；姓名：李康；密码：www.mldnjava.cn；年龄：23；性别：女；生日：2003-08-27
--------------------------
第1条数据：
编号：1；姓名：李兴华；密码：www.mldn.cn；年龄：30；性别：男；生日：2008-08-27
```

```
-------------------------
第3条数据：
编号：3；姓名：李兴华；密码：www.mldnjava.cn；年龄：30；性别：男；生日：2007-08-27
-------------------------
```

以上查询结果因为设置了其可以滚动，所以可以通过方法输出指定的行，也可以跳转到结果集的开头和末尾。

17.11.2　使用结果集插入数据

在 JDBC 2.0 中，如果要进行数据库的更新操作，则在创建 PreparedStatement 对象时必须指定结果集可以更新数据库，如下所示：

```
pstmt = conn.prepareStatement(sql, ResultSet.TYPE_SCROLL_SENSITIVE,
        ResultSet.CONCUR_UPDATABLE);                    // 实例化对象
```

范例：直接在 user 表中增加数据

```
package org.lxh.demo17.jdbc20;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
public class JDBC20InsertDemo {
    // 定义MySQL的数据库驱动程序
    public static final String DBDRIVER = "org.gjt.mm.mysql.Driver" ;
    // 定义MySQL数据库的连接地址
    public static final String DBURL = "jdbc:mysql://localhost:3306/mldn" ;
    // MySQL数据库的连接用户名
    public static final String DBUSER = "root" ;
    // MySQL数据库的连接密码
    public static final String DBPASS = "mysqladmin" ;
    public static void main(String[] args) throws Exception {
        Connection conn = null ;                        // 数据库连接
        PreparedStatement pstmt = null ;                // 数据库操作
        ResultSet rs = null ;                           // 保存查询结果
        String sql = "SELECT id,name,password,age,sex,birthday "+
            " FROM user" ;                              // 此处不需要设置任何内容
        Class.forName(DBDRIVER) ;                       // 加载驱动程序
        // 连接MySQL数据库时，要写上连接的用户名和密码
        conn = DriverManager.getConnection(DBURL, DBUSER, DBPASS);
        pstmt = conn.prepareStatement(sql, ResultSet.TYPE_SCROLL_SENSITIVE,
                ResultSet.CONCUR_UPDATABLE);            // 实例化对象
        rs = pstmt.executeQuery() ;                     // 实例化ResultSet对象
        rs.moveToInsertRow() ;                          // 移动到可以插入的数据行
```

```
            rs.updateString("name", "李华") ;                    // 设置要插入的姓名
            rs.updateString("password", "lixinghua") ;           // 设置要插入的密码
            rs.updateInt("age", 33) ;                            // 设置要插入的年龄
            rs.updateString("sex", "女") ;                       // 设置要插入的性别
            rs.updateDate("birthday", new java.sql.Date(new java.util.Date()
                    .getTime()));                                // 将今天的日期设置为生日
            rs.insertRow() ;                                     // 插入数据
            rs.close() ;                                         // 关闭结果集
            pstmt.close() ;                                      // 操作关闭
            conn.close() ;                                       // 数据库关闭
        }
    }
```

以上是直接使用结果集进行的数据库插入操作，其功能与之前是一样的。使用结果集插入之后的 user 表如图 17-42 所示。

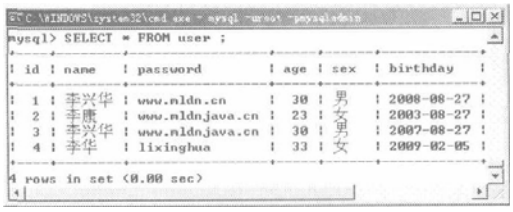


图 17-42　使用结果集插入之后的 user 表

17.11.3　使用结果集更新数据

使用 ResultSet 完成更新操作也是比较容易的，但是一般在更新时，都会对执行行进行更新，所以以下代码将使用限定查询，查询 id 编号为 3 的用户信息，并更新其信息。

范例：使用结果集更新

```
package org.lxh.demo17.jdbc20;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
public class JDBC20UpdateDemo {
    // 定义MySQL的数据库驱动程序
    public static final String DBDRIVER = "org.gjt.mm.mysql.Driver" ;
    // 定义MySQL数据库的连接地址
    public static final String DBURL = "jdbc:mysql://localhost:3306/mldn" ;
    // MySQL数据库的连接用户名
    public static final String DBUSER = "root" ;
    // MySQL数据库的连接密码
    public static final String DBPASS = "mysqladmin" ;
    public static void main(String[] args) throws Exception {
```

```
        Connection conn = null ;                          // 数据库连接
        PreparedStatement pstmt = null ;                  // 数据库操作
        ResultSet rs = null ;                             // 保存查询结果
        String sql = "SELECT id,name,password,age,sex,birthday "+
            " FROM user WHERE id=?" ;                     // 此处使用预处理操作
        Class.forName(DBDRIVER) ;                         // 加载驱动程序
        // 连接MySQL数据库时，要写上连接的用户名和密码
        conn = DriverManager.getConnection(DBURL, DBUSER, DBPASS);
        pstmt = conn.prepareStatement(sql, ResultSet.TYPE_SCROLL_SENSITIVE,
                ResultSet.CONCUR_UPDATABLE);              // 实例化对象
        pstmt.setInt(1, 3) ;                              // 更新3号用户
        rs = pstmt.executeQuery() ;                       // 实例化ResultSet对象
        rs.last() ;                                       // 移动到最后一行
        rs.updateString("name", "李藏") ;                 // 设置要插入的姓名
        rs.updateString("password", "lizang") ;           // 设置要插入的密码
        rs.updateInt("age", 23) ;                         // 设置要插入的年龄
        rs.updateString("sex", "女") ;                    // 设置要插入的性别
        rs.updateDate("birthday", new java.sql.Date(new java.util.Date()
                .getTime()));                             // 将今天的日期设置为生日
        rs.updateRow() ;                                  // 更新数据
        rs.close() ;                                      // 关闭结果集
        pstmt.close() ;                                   // 操作关闭
        conn.close() ;                                    // 数据库关闭
    }
}
```

与插入不同的是，此处使用的是 updateRow()进行了数据库更新操作。而且在修改数据时必须将 ResultSet 的指针指向要更改的行，因为此时只查询了一条语句，所以直接使用 last()方法即可。ResultSet 更新之后的 user 表如图 17-43 所示。



图 17-43　ResultSet 更新之后的 user 表

提示：在数据库更新之前可以使用 cancelRowUpdates()方法取消更新操作。

在执行 updateRow()方法前，如果发现数据库更新有错误，则可以使用 cancelRowUpdates()方法取消之前的更新，这样就算再执行 updateRow()方法，也不会去更新数据库，如下所示：

```
        rs.cancelRowUpdates() ;                           // 取消更新
        rs.updateRow() ;                                  // 更新数据
```

一定要记住，cancelRowUpdates()方法要在 updateRow()方法执行之前调用，否则是无效的。

17.11.4 使用结果集删除数据

使用结果集删除数据与之前的操作完全一样，唯一不同的是，此时使用的是 deleteRow() 方法。

范例： 删除指定编号的数据

```
package org.lxh.demo17.jdbc20;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
public class JDBC20DeleteDemo {
    // 定义MySQL的数据库驱动程序
    public static final String DBDRIVER = "org.gjt.mm.mysql.Driver" ;
    // 定义MySQL数据库的连接地址
    public static final String DBURL = "jdbc:mysql://localhost:3306/mldn" ;
    // MySQL数据库的连接用户名
    public static final String DBUSER = "root" ;
    // MySQL数据库的连接密码
    public static final String DBPASS = "mysqladmin" ;
    public static void main(String[] args) throws Exception {
        Connection conn = null ;                        // 数据库连接
        PreparedStatement pstmt = null ;                // 数据库操作
        ResultSet rs = null ;                           // 保存查询结果
        String sql = "SELECT id,name,password,age,sex,birthday "+
            " FROM user WHERE id=?" ;                   // 此处使用预处理操作
        Class.forName(DBDRIVER) ;                       // 加载驱动程序
        // 连接MySQL数据库时，要写上连接的用户名和密码
        conn = DriverManager.getConnection(DBURL, DBUSER, DBPASS);
        pstmt = conn.prepareStatement(sql, ResultSet.TYPE_SCROLL_SENSITIVE,
                ResultSet.CONCUR_UPDATABLE);            // 实例化对象
        pstmt.setInt(1, 2) ;                            // 删除2号用户
        rs = pstmt.executeQuery() ;                     // 实例化ResultSet对象
        rs.last() ;                                     // 移动到最后一行
        rs.deleteRow() ;                                // 删除数据
        rs.close() ;                                    // 关闭结果集
        pstmt.close() ;                                 // 操作关闭
        conn.close() ;                                  // 数据库关闭
    }
}
```

以上操作代码完成了数据的删除操作，程序首先根据 id 查找到指定的一行记录，再通

过 last()方法跳转到最后一行记录，之后执行 deleteRow()方法，即将当前行的数据删除。删除之后的 user 表内容如图 17-44 所示。

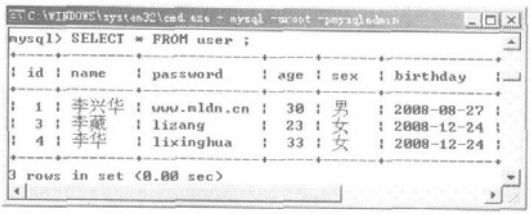


图 17-44　使用 ResultSet 删除之后的 user 表

17.11.5　批处理

在 JDBC 2.0 中，最重要的概念是批处理操作。使用批处理可以一次性插入多条 SQL 语句，如果要完成批处理操作，则要使用 addBatch()加入要执行的一条 SQL 命令以及 executeBatch()执行全部命令两个方法完成。

范例：批量插入数据

```
package org.lxh.demo17.jdbc20;
import java.sql.Connection;
import java.sql.Date;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
public class JDBC20BatchDemo {
    // 定义MySQL的数据库驱动程序
    public static final String DBDRIVER = "org.gjt.mm.mysql.Driver" ;
    // 定义MySQL数据库的连接地址
    public static final String DBURL = "jdbc:mysql://localhost:3306/mldn" ;
    // MySQL数据库的连接用户名
    public static final String DBUSER = "root" ;
    // MySQL数据库的连接密码
    public static final String DBPASS = "mysqladmin" ;
    public static void main(String[] args) throws Exception {
        Connection conn = null ;                        // 数据库连接
        PreparedStatement pstmt = null ;                // 数据库操作
        String sql = "INSERT INTO user(name,password,age,sex,birthday) "
            + " VALUES (?,?,?,?,?)";                    // 此处使用预处理操作
        Class.forName(DBDRIVER) ;                       // 加载驱动程序
        // 连接MySQL数据库时，要写上连接的用户名和密码
        conn = DriverManager.getConnection(DBURL, DBUSER, DBPASS);
        pstmt = conn.prepareStatement(sql);             // 实例化对象
        for (int i = 0; i < 10; i++) {
            pstmt.setString(1, "李兴华 - " + i);
            pstmt.setString(2, "MLDN - " + i);
```

```
            pstmt.setInt(3, 20 + i);
            pstmt.setString(4, "男");
            pstmt.setDate(5, new Date(new java.util.Date().getTime()));
            pstmt.addBatch() ;                          // 加入批处理等待执行
        }
        int temp[] = pstmt.executeBatch() ;             // 批量执行
        System.out.println("更新了" + temp.length + "条数据。");
        pstmt.close() ;                                 // 操作关闭
        conn.close() ;                                  // 数据库关闭
    }
}
```

程序运行结果：

```
更新了10条数据。
```

以上完成了批处理的操作，数据库一次性提交了 10 条新记录到 user 表中，如图 17-45 所示。

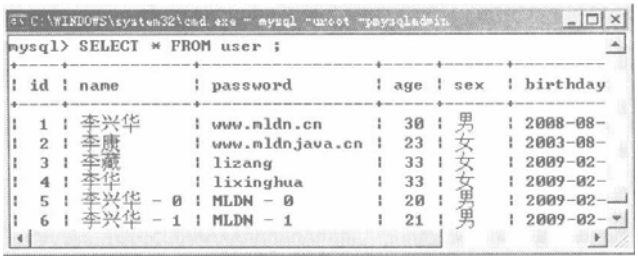


图 17-45　执行批处理

17.12　事 务 处 理

事务处理在数据库开发中有着非常重要的作用，所谓事务就是所有的操作要么一起成功，要么一起失败，事务本身具有原子性（Atomicity）、一致性（Consistency）、隔离性或独立性（Isolation）、持久性（Durability）4 个特征，这 4 个特征也被称为 ACID 特征。

- 原子性：原子性是事务最小的单元，是不可再分割的单元，相当于一个个小的数据库操作，这些操作必须同时完成，如果有一个失败了，则一切的操作将全部失败。如图 17-46 所示，A 转账和 B 结账分别是两个不可再分的操作，但是如果 A 的转账失败，则 B 的操作也肯定无法成功。
- 一致性：指的是在数据库操作的前后是完全一致的，保证数据的有效性，如果事务正常操作则系统会维持有效性，如果事务出现了错误，则回到最原始状态，也要维持其有效性，这样保证事务开始时和结束时系统处于一致状态。如图 17-46 所示，如果 A 和 B 转账成功，则保持其一致性；如果现在 A 和 B 的转账失败，则保持操作之前的一致性，即 A 的钱不会减少，B 的钱不会增加。

- 隔离性：多个事务可以同时进行且彼此之间无法访问，只有当事务完成最终操作时，才可以看到结果。
- 持久性：当一个系统崩溃时，一个事务依然可以坚持提交，当一个事务完成后，操作的结果保存在磁盘中，永远不会被回滚。如图 17-46 所示，所有资金数都保存在磁盘中，所以，即使系统发生了错误，用户的资金也不会减少。

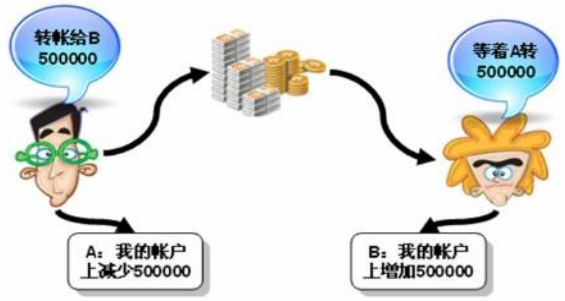


图 17-46　转账操作

17.12.1　MySQL 对事务的支持

在 MySQL 中提供了如表 17-15 所示的几个命令，可以进行事务的处理。

表 17-15　MySQL 中对事务的支持命令

序号	命　　令	描　　述
1	SET AUTOCOMMIT=0	取消自动提交处理，开启事务处理
2	SET AUTOCOMMIT=1	打开自动提交处理，关闭事务处理
3	START TRANSACTION	启动事务
4	BEGIN	启动事务，相当于执行 START TRANSACTION
5	COMMIT	提交事务
6	ROLLBACK	回滚全部操作
7	SAVEPOINT 事务保存点名称	设置事务保存点
8	ROLLBACK TO SAVEPOINT 保存点名称	回滚操作到保存点

以上所有操作都是针对于一个 session 的，在数据库操作中把每一个连接到此数据库上的用户都称为一个 session，如图 17-47 所示。

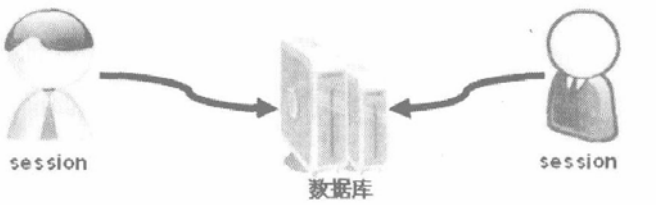


图 17-47　每一个连接到数据库的用户就是一个 session

在 MySQL 中，如果要应用事务处理，则应该按照以下顺序输入命令。

- 取消自动提交，执行：SET AUTOCOMMIT=0。这样所有的更新指令并不会立刻发送到数据表中，而只存在于当前的 session。

- 开启事务，执行：START TRANSACTION 或 BEGIN。
- 编写数据库更新语句，如增加、修改、删除，可以在编写的更新语句之间记录事务的保存点，使用 SAVEPOINT 指令。
- 提交事务，如果确信数据库的修改没有任何的错误，则使用 COMMIT 提交事务。在提交之前对数据库所做的全部操作都将保存在 session 中。
- 事务回滚，如果发现执行的 SQL 语句有错误，则使用 ROLLBACK 命令全部撤销，或者使用 ROLLBACK TO SAVEPOINT 记录点，让其回滚到指定的位置。

当一个事务进行时，其他的 session 是无法看到此事务的操作状态的。即此 session 对数据库所做的一切修改，如果没有提交事务，则其他 session 是无法看到此 session 操作结果的。

17.12.2 执行 JDBC 的事务处理

在讲解 JDBC 进行事务处理之前，先来看一种情况，现在要求在数据库中执行 5 条 SQL 语句，这些 SQL 语句本身需要保持一致，即要么同时成功，要么同时失败，此时，先不使用事务处理。

提示：重新建立 user 表。

为了方便读者观察运行效果，下面的操作每次执行前都将删除已有的 user 表，并重新建立 user 表，这样自动编号将从 1 开始。

范例：没有使用事务处理

```
package org.lxh.demo17.trandemo;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
public class TranDemo01 {
    // 定义MySQL的数据库驱动程序
    public static final String DBDRIVER = "org.gjt.mm.mysql.Driver" ;
    // 定义MySQL数据库的连接地址
    public static final String DBURL = "jdbc:mysql://localhost:3306/mldn" ;
    // MySQL数据库的连接用户名
    public static final String DBUSER = "root" ;
    // MySQL数据库的连接密码
    public static final String DBPASS = "mysqladmin" ;
    public static void main(String[] args) throws Exception {
        Connection conn = null ;                                    // 数据库连接
        Statement stmt = null ;                                     // 数据库操作
        Class.forName(DBDRIVER) ;                                   // 加载驱动程序
        // 连接MySQL数据库时，要写上连接的用户名和密码
        conn = DriverManager.getConnection(DBURL, DBUSER, DBPASS);
        stmt = conn.createStatement();                              // 实例化对象
```

```
            // 加入5条SQL处理语句
            stmt.addBatch("INSERT INTO user(name,password,age,sex,birthday)"
                    + " VALUES ('LXH-1','hello-1',11,'男','1975-03-05')");
            stmt.addBatch("INSERT INTO user(name,password,age,sex,birthday)"
                    + " VALUES ('LXH-2','hello-2',12,'女','1976-01-07')");
            // 第3个加入的预处理语句估计写错，将名字写成“LXH-'3'”，多了一个“'”
            stmt.addBatch("INSERT INTO user(name,password,age,sex,birthday)"
                    + " VALUES ('LXH-'3','hello-3',13,'男','1976-06-01')");
            stmt.addBatch("INSERT INTO user(name,password,age,sex,birthday)"
                    + " VALUES ('LXH-4','hello-4',14,'女','1977-08-01')");
            stmt.addBatch("INSERT INTO user(name,password,age,sex,birthday)"
                    + " VALUES ('LXH-5','hello-5',15,'男','1978-10-01')");
            int temp[] = stmt.executeBatch() ;                    // 批量执行
            System.out.println("更新了" + temp.length + "条数据。");
            stmt.close() ;                                        // 操作关闭
            conn.close() ;                                        // 数据库关闭
        }
    }
```

执行时出现以下错误信息：

```
Exception in thread "main" java.sql.BatchUpdateException: You have an error
in your SQL syntax; check the manual that corresponds to your MySQL server version
for the right syntax to use near '3','hello-3',13,'??','1976-06-01')' at line 1
        at com.mysql.jdbc.Statement.executeBatch(Statement.java:919)
        at org.lxh.demo17.trandemo.TranDemo01.main(TranDemo01.java:35)
```

此时再观察数据库中的信息，查询全部数据。没有使用事务处理，只插入了部分的数据如图 17-48 所示。

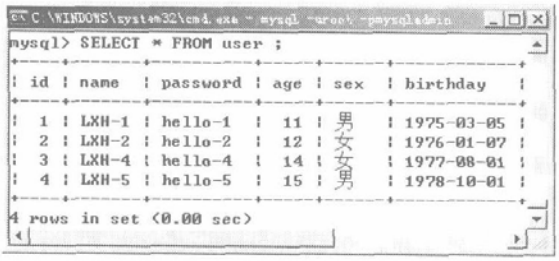


图 17-48　没有使用事务处理，只插入了部分的数据

从以上的运行结果发现，程序只将出错之前的 SQL 语句正确执行了，但这些语句是一个整体，是应该集体成功之后才有效的。很明显这样的运行结果是无法符合用户要求的，而在使用了事务处理之后，此问题即可解决。在 JDBC 中，如果要想进行事务处理，也需要按照指定的步骤完成。

- 取消 Connection 中设置的自动提交方式“conn.setAutoCommit(**false**) ;”。
- 如果批处理操作成功，则执行提交事务“conn.commit() ;”。

- 如果操作失败，则肯定会引发异常，在异常处理中让事务回滚“conn.rollback();”。
- 如果需要，可以设置 Savepoint“Savepoint sp = conn.setSavepoint();”。

范例：事务基本操作

```
package org.lxh.demo17.trandemo;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
public class TranDemo02 {
    // 定义MySQL的数据库驱动程序
    public static final String DBDRIVER = "org.gjt.mm.mysql.Driver" ;
    // 定义MySQL数据库的连接地址
    public static final String DBURL = "jdbc:mysql://localhost:3306/mldn" ;
    // MySQL数据库的连接用户名
    public static final String DBUSER = "root" ;
    // MySQL数据库的连接密码
    public static final String DBPASS = "mysqladmin" ;
    public static void main(String[] args) throws Exception {
        Connection conn = null ;                                    // 数据库连接
        Statement stmt = null ;                                     // 数据库操作
        Class.forName(DBDRIVER) ;                                   // 加载驱动程序
        // 连接MySQL数据库时，要写上连接的用户名和密码
        conn = DriverManager.getConnection(DBURL, DBUSER, DBPASS);
        conn.setAutoCommit(false) ;                                 // 取消自动提交
        stmt = conn.createStatement();                              // 实例化对象
        // 加入5条SQL处理语句
        stmt.addBatch("INSERT INTO user(name,password,age,sex,birthday)"
                + " VALUES ('LXH-1','hello-1',11,'男','1975-03-05')");
        stmt.addBatch("INSERT INTO user(name,password,age,sex,birthday)"
                + " VALUES ('LXH-2','hello-2',12,'女','1976-01-07')");
        // 第3个加入的预处理语句估计写错，将名字写成“LXH-'3'”，多了一个“'”
        stmt.addBatch("INSERT INTO user(name,password,age,sex,birthday)"
                + " VALUES ('LXH-'3','hello-3',13,'男','1976-06-01')");
        stmt.addBatch("INSERT INTO user(name,password,age,sex,birthday)"
                + " VALUES ('LXH-4','hello-4',14,'女','1977-08-01')");
        stmt.addBatch("INSERT INTO user(name,password,age,sex,birthday)"
                + " VALUES ('LXH-5','hello-5',15,'男','1978-10-01')");
        try{
            int temp[] = stmt.executeBatch() ;                      // 批量执行
            System.out.println("更新了" + temp.length + "条数据。");
            conn.commit() ;                                         // 提交事务
        } catch (Exception e) {
            try {
```

```
                conn.rollback();                                    // 事务回滚
            } catch (Exception ex) {
            }
        }
        stmt.close() ;                                              // 操作关闭
        conn.close() ;                                              // 数据库关闭
    }
}
```

以上程序中加入了事务的处理操作，这样当更新出错后，数据库会进行回滚，则所有的更新操作全部失效，如图 17-49 所示。

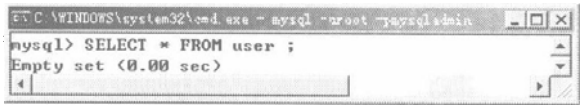


图 17-49　加入事务操作之后的 user 表

当然，也可以加入若干的 Savepoint 作为事务的保存点。

范例：加入 Savepoint

```
package org.lxh.demo17.trandemo;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Savepoint;
import java.sql.Statement;
public class TranDemo03 {
    // 定义MySQL的数据库驱动程序
    public static final String DBDRIVER = "org.gjt.mm.mysql.Driver" ;
    // 定义MySQL数据库的连接地址
    public static final String DBURL = "jdbc:mysql://localhost:3306/mldn" ;
    // MySQL数据库的连接用户名
    public static final String DBUSER = "root" ;
    // MySQL数据库的连接密码
    public static final String DBPASS = "mysqladmin" ;
    public static void main(String[] args) throws Exception {
        Connection conn = null ;                                    // 数据库连接
        Statement stmt = null ;                                     // 数据库操作
        Class.forName(DBDRIVER) ;                                   // 加载驱动程序
        // 连接MySQL数据库时，要写上连接的用户名和密码
        conn = DriverManager.getConnection(DBURL, DBUSER, DBPASS);
        conn.setAutoCommit(false) ;                                 // 取消自动提交
        stmt = conn.createStatement();                              // 实例化对象
        // 加入5条SQL处理语句
        stmt.executeUpdate("INSERT INTO user(name,password,age,sex,birthday)"
                + " VALUES ('LXH-1','hello-1',11,'男','1975-03-05')");
```

```
            stmt.executeUpdate("INSERT INTO user(name,password,age,sex,birthday)"
                    + " VALUES ('LXH-2','hello-2',12,'女','1976-01-07')");
            Savepoint sp = conn.setSavepoint() ;                    // 设置保存点
            stmt.executeUpdate("INSERT INTO user(name,password,age,sex,birthday)"
                    + " VALUES ('LXH-4','hello-4',14,'女','1977-08-01')");
            stmt.executeUpdate("INSERT INTO user(name,password,age,sex,birthday)"
                    + " VALUES ('LXH-5','hello-5',15,'男','1978-10-01')");
            try{
                conn.rollback(sp) ;                                 // 回滚到保存点
                conn.commit() ;                                     // 提交事务
            } catch (Exception e) {
            }
            stmt.close() ;                                          // 操作关闭
            conn.close() ;                                          // 数据库关闭
        }
    }
```

以上程序在执行 SQL 语句时设置了一个保存点，在提交之前，将操作回滚到保存点上。这样也就相当于此时只能真正向数据库中插入两条 SQL 语句，如图 17-50 所示。

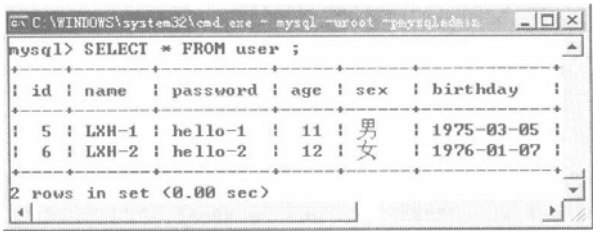


图 17-50　只有两条 SQL 语句提交

以上程序中虽然设置了一个保存点，但是在操作保存点时要以一个对象的形式操作，而且本身也没有太大的意义，以上的程序只需要了解其基本操作形式即可。

17.13　使用元数据分析数据库

在 JDBC 中除了可以支持数据库的更新和查询操作外，还可以使用其本身提供的元数据类对数据库的组成进行分析。在 JDBC 中提供了 DatabaseMetaData 和 ResultSetMetaData 接口来分析数据库的元数据。

17.13.1　DatabaseMetaData

DatabaseMetaData 可以得到数据库的一些基本信息，包括数据库的名称、版本，以及得到表的信息，这些方法如表 17-16 所示。

表 17-16　DatabaseMetaData 基本方法

序号	方　法	类型	描　述
1	String getDatabaseProductName() throws SQLException	普通	得到数据库的名称
2	int getDriverMajorVersion()	普通	得到数据库的主版本号
3	int getDriverMinorVersion()	普通	得到数据库的次版本号
4	ResultSet getPrimaryKeys(String catalog,String schema,String table) throws SQLException	普通	得到表的主键信息，每一个主键都有以下的描述： ● TABLE_CAT String：表类别（可为 null） ● TABLE_SCHEM String：表模式（可为 null） ● TABLE_NAME String：表名称 ● COLUMN_NAME String：列名称 ● KEY_SEQ short：主键中的序列号 ● PK_NAME String：主键的名称（可为 null）

范例：使用 DatabaseMetaData 取得数据库的元信息

```
package org.lxh.demo17.metademo;
import java.sql.Connection;
import java.sql.DatabaseMetaData;
import java.sql.DriverManager;
import java.sql.ResultSet;
public class DatabaseMetaDataDemo {
    // 定义MySQL的数据库驱动程序
    public static final String DBDRIVER = "org.gjt.mm.mysql.Driver" ;
    // 定义MySQL数据库的连接地址
    public static final String DBURL = "jdbc:mysql://localhost:3306/mldn" ;
    // MySQL数据库的连接用户名
    public static final String DBUSER = "root" ;
    // MySQL数据库的连接密码
    public static final String DBPASS = "mysqladmin" ;
    public static void main(String[] args) throws Exception {
        Connection conn = null ;                          // 数据库连接
        DatabaseMetaData dmd = null ;                     // 数据库元数据
        ResultSet rs = null ;                             // 结果集对象
        Class.forName(DBDRIVER) ;                         // 加载驱动程序
        // 连接MySQL数据库时，要写上连接的用户名和密码
        conn = DriverManager.getConnection(DBURL, DBUSER, DBPASS);
        dmd = conn.getMetaData() ;                        // 实例化元数据
        System.out.println("数据库名称：" + dmd.getDatabaseProductName());
        System.out.println("数据库版本：" + dmd.getDriverMajorVersion() + "."
```

```
                + dmd.getDatabaseMinorVersion());
        rs = dmd.getPrimaryKeys(null, null, "user") ; // 得到表的主键
        while (rs.next()) {
            System.out.println("表类别: " + rs.getString(1));
            System.out.println("表模式: " + rs.getString(2));
            System.out.println("表名称: " + rs.getString(3));
            System.out.println("列名称: " + rs.getString(4));
            System.out.println("主键序列号: " + rs.getString(5));
            System.out.println("主键名称: " + rs.getString(6));
        }
        conn.close() ;                                    // 数据库关闭
    }
}
```

程序运行结果：

```
数据库名称：MySQL
数据库版本：5.0
表类别：mldn
表模式：null
表名称：user
列名称：id
主键序列号：1
主键名称：PRIMARY
```

17.13.2 ResultSetMetaData

使用 ResultSetMetaData 可获取关于 ResultSet 对象中列的类型和属性信息的对象，ResultSetMetaData 存储了 ResultSet 的 MetaData，可以通过以下方法取得一些 ResultSet 的信息，如表 17-17 所示。

表 17-17 ResultSetMetaData 的方法

序号	方 法	类型	描 述
1	int getColumnCount() throws SQLException	普通	返回一个查询结果中的列数
2	boolean isAutoIncrement(int column) throws SQLException	普通	判断指定列是否是自动编号
3	String getColumnName(int column) throws SQLException	普通	返回列的名称

要想取得一个 ResultSetMetaData 的对象，则可以使用 PreparedStatement 接口提供的以下方法：

```
ResultSetMetaData getMetaData() throws SQLException
```

范例：取得 ResultSet 的 MetaData

```
package org.lxh.demo17.metademo;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSetMetaData;
public class ResultSetMetaDataDemo {
    // 定义MySQL的数据库驱动程序
    public static final String DBDRIVER = "org.gjt.mm.mysql.Driver" ;
    // 定义MySQL数据库的连接地址
    public static final String DBURL = "jdbc:mysql://localhost:3306/mldn" ;
    // MySQL数据库的连接用户名
    public static final String DBUSER = "root" ;
    // MySQL数据库的连接密码
    public static final String DBPASS = "mysqladmin" ;
    public static void main(String[] args) throws Exception {
        Connection conn = null ;                          // 数据库连接
        ResultSetMetaData rsmd = null ;                   // 结果集元数据
        PreparedStatement pstmt = null ;                  // 数据库操作对象
        Class.forName(DBDRIVER) ;                         // 加载驱动程序
        // 连接MySQL数据库时，要写上连接的用户名和密码
        conn = DriverManager.getConnection(DBURL, DBUSER, DBPASS);
        String sql = "SELECT id,name,password,age,sex,birthday FROM user" ;
        pstmt = conn.prepareStatement(sql) ;              // 实例化对象
        rsmd = pstmt.getMetaData() ;                      // 得到结果集元数据
        System.out.println(" 一共返回" + rsmd.getColumnCount() + "条数据。");
        if(rsmd.isAutoIncrement(1)){
            System.out.println(rsmd.getColumnName(1) + "列是自动增长的。");
        }
        conn.close() ;                                    // 数据库关闭
    }
}
```

程序运行结果：

```
一共返回6条数据。
id列是自动增长的。
```

17.14　使用 JDBC 连接 Oracle 数据库

之前的 JDBC 操作是以 MySQL 数据库为基础讲解的。当然，也可以使用 JDBC 连接 Oracle 数据库，此时就必须使用 Oracle 的 JDBC 驱动程序，此驱动程序是随着 Oracle 一起

安装到目录下的，保存路径是 D:\oracle\product\10.1.0\db_1\jdbc\lib\classes12.jar。本章所使用的 Oracle 版本是 10g。

不管使用哪种数据库都需要数据库的驱动程序路径、连接地址、用户名、密码等信息，此相关信息如下所示。

- 驱动程序：oracle.jdbc.driver.OracleDriver。
- 连接地址：jdbc:oracle:thin:@ip 地址:端口:数据库名称。

此时数据库安装在本地，名称为“mldn”，所以此时数据库的连接地址为 jdbc:oracle:thin:@localhost:1521:MLDN。

- 用户名：scott。
- 密码：tiger。

在 Oracle 中不像 MySQL 那样可以直接使用 AUTO_INCREMENT 完成自动增长列，在 Oracle 中要使用序列完成自动增长列的操作，例如，现在有如下的数据表：

```
DROP TABLE person ;
DROP SEQUENCE myseq ;
CREATE SEQUENCE myseq ;
CREATE TABLE person
(
    id          INT             PRIMARY KEY NOT NULL ,
    name        VARCHAR(50)     NOT NULL ,
    age         INT ,
    birthday    DATE
) ;
```

范例：向 person 表中插入数据

```
package org.lxh.demo17.oracledemo;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
public class JDBCOracle {
    // Oracle数据库驱动程序
    public static final String DBDRIVER = "oracle.jdbc.driver.OracleDriver";
    // Oracle数据库的连接地址
    public static final String DBURL = "jdbc:oracle:thin:@localhost:1521:MLDN";
    // Oracle数据库的连接用户名
    public static final String DBUSER = "scott";
    // Oracle数据库的连接密码
    public static final String DBPASSWORD = "tiger";
    public static void main(String[] args) throws Exception {
        Connection conn = null;                          // 连接数据库接口
        PreparedStatement pstmt = null;                  // 操作数据库
        Class.forName(DBDRIVER);                         // 加载驱动程序
```

```
            // 连接数据库，连接时要设置连接地址、用户名、密码
            conn = DriverManager.getConnection(DBURL, DBUSER, DBPASSWORD);
            // 在编写SQL语句时，指明序列的内容
            String sql = "INSERT INTO person(id,name,age,birthday) VALUES (myseq.
            nextVal,?,?,?) ";
            pstmt = conn.prepareStatement(sql);
            pstmt.setString(1, "李兴华");
            pstmt.setInt(2, 30);
            pstmt.setDate(3, new java.sql.Date(new java.util.Date().getTime()));
            pstmt.executeUpdate();                              // 执行SQL语句
            pstmt.close();
            conn.close();
        }
    }
```

从程序代码来看，JDBC 操作 Oracle 和 MySQL 本身并没有太大的区别，所以 JDBC 提供的是一个标准的操作。但是各个数据库本身也是有差异的，例如，MySQL 中自动增长使用的 AUTO_INCREMENT 不需要用户调用，而在 Oracle 中就必须使用手工调用 SEQUENCE 的形式完成。这些数据库间的不同特点，读者在日后开发时要特别注意。

提问：在开发中到底是使用 MySQL 还是 Oracle 呢？

现在学习过了两种数据库的 JDBC 操作，那么在开发中到底是应该使用 Oracle 好还是使用 MySQL 好？

回答：针对需要选择数据库。

在实际开发中，数据库的选择有时是由需求方直接选定的，有时是由开发者自行选定的。但是因为 MySQL 是免费的，所以对于一些小型数据量的操作而言使用 MySQL 就足够了；而如果数据量较大，而且需求方又提供了购买 Oracle 数据库费用，则可以使用 Oracle。对个人学习而言，使用 MySQL 就足够了。

17.15　本 章 要 点

1．DBC 提供了一套与平台无关的标准数据库操作接口和类，只要是支持 Java 的数据库厂商，所提供的数据库都可以使用 JDBC 操作。

2．JDBC 的操作步骤如下所示。

（1）加载驱动程序：驱动程序由各个数据库生产商提供。

（2）连接数据库：连接时要提供连接路径、用户名、密码。

（3）实例化操作：通过连接对象实例化 Statement 或 PreparedStatement 对象。

（4）操作数据库：使用 Statement 或 PreparedStatement 操作，如果是查询，则全部的查询结果使用 ResultSet 进行接收。

3．JDBC 提供了大对象的操作类，操作大对象时使用 IO 流的方式写入，读取时也可以使用 Clob、Blob 方便操作。

4．可以使用 CallableStatement 调用数据库中的存储过程。

5．可以在创建 Statement 或 PreparedStatement 时，指定 ResultSet 的操作类型和并发性，这样就可以使用 ResultSet 进行滚动以及使用 ResultSet 直接更新数据库操作。

6．JDBC 2.0 中提供的重要特性就是批处理操作，此操作可以让多条 SQL 语句一次性执行完毕。

7．MySQL 提供了事务的支持命令，在 JDBC 中也同样可以进行事务操作，JDBC 事务操作的步骤如下所示。

（1）取消自动提交。

（2）使用手工提交方式。

（3）如果出现了操作错误，则一切操作回滚。

8．在 JDBC 中可以使用 DatabaseMetaData 和 ResultSetMetaData 分析数据库。

9．各个数据库生产商都会提供各自的数据库驱动程序，在使用 JDBC 连接不同数据库时都需要将驱动程序配置到 classpath 中。

17.16 习　　题

1．建立一个用户表（数据表和表中的数据列由用户自行创建），使用键盘输入列的信息，并将信息保存在数据库中。

2．编写一个程序，可以通过此程序完成一个表的创建操作，输入表名称、各个列的名称及类型，输入完成后直接通过 JDBC 创建指定的表。

3．使用键盘输入流，接收一段大文本数据，通过输入的路径输入保存的图片路径，并将这些数据保存在数据表中，数据表由用户自行创建。

4．将第 6 章的宠物商店程序修改为使用数据库保存全部的宠物信息，并可以实现关键字查找。

5．建立一张雇员表（雇员编号、姓名、工作、雇佣日期、基本工资、部门名称），在命令窗口下将表中的全部数据列出。

明确的注意点：

在实际的 JAVA 开发中已经很少在去编写图形界面，所以对于这些操作，读者只需要大概了解即可。JAVA 的重点一定要放在类的设计上以及：IO、类集、面向对象、JDBC

第 18 章　图 形 界 面

通过本章的学习可以达到以下目标：

- ☑ 了解 AWT 与 Swing 的关系。
- ☑ 掌握组件、容器、布局管理器的概念。
- ☑ 了解 JFrame、JPanel、JSplitPane、JTabbedPane、JScrollPane、JDesktopPane 等常见容器。
- ☑ 了解 JLable 组件及 JButton 组件，并可以通过设置显示文字风格及显示图像。
- ☑ 了解事件处理作用及实现机制。
- ☑ 了解文本框组件、密码框组件、文本域组件的使用。
- ☑ 了解单选按钮、复选框、列表框、下拉列表框等常见组件的使用及事件处理。
- ☑ 了解菜单组件及文件选择组件的使用。
- ☑ 了解表格的建立，并可以使用 TableModel 构建一个表格。

在一个系统中，一个良好的人机界面无外乎是最重要的，Windows 以其良好的人机操作界面在操作系统中占有绝对的统治地位，作用可见一般。庆幸的是在 Java 中也可以完成这样的操作界面，图形用户界面（Graphical User Interface，GUI）编程主要有以下几个特征：

- ➘ 图形界面对象及其框架（图形界面对象之间的包含关系）。
- ➘ 图形界面对象的布局（图形界面对象之间的位置关系）。
- ➘ 图形界面对象上的事件响应（图形界面对象上的动作）。

在 Java 的图形界面开发中有两种可使用的技术：AWT、Swing。但是在 AWT 中大量地使用了 Windows 的系统函数，不是使用 Java 开发的；而 Swing 是由 Java 来实现的用户界面类，可以在任意的系统平台上工作，但是在 Swing 中仍然大量使用了 AWT 的概念，为了让读者更加清楚 Swing 的组成，下面先来介绍 AWT 技术。**本章视频录像讲解时间为 6 小时 15 分钟，源代码在光盘对应的章节下。**

18.1　AWT 简介

AWT（Abstract Windowing Toolkit），抽象窗口工具包，是 Sun 公司在发布 JDK 1.0 时一个重要的组成部分，是 Java 提供的用来建立和设置 Java 的图形用户界面的基本工具。AWT 中的所有工具类都保存在 java.awt 包中，此包中的所有操作类可用来建立与平台无关的图形用户界面（GUI）的类，这些类又被称为组件（Components）。

在整个 AWT 包中提供的所有工具类主要分为以下 3 种。

- ➘ 组件：Component。
- ➘ 容器：Container。

➘ 布局管理器：LayoutManager。

在 java.awt 包中所提供的组件类非常多，主要的几个类如图 18-1 所示。

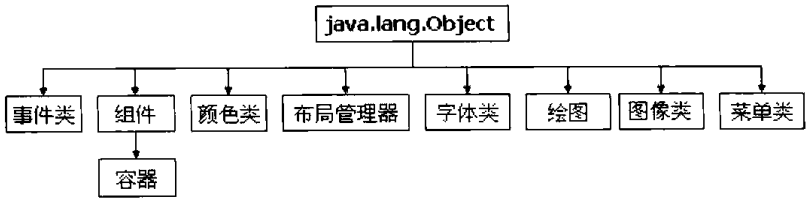


图 18-1 AWT 包的整体结构

18.1.1 组件

在图形界面中，用户经常会看到一个个的按钮、标签、菜单等，这些实际上就是一个个的组件。这些组件都会在一个窗体上显示，如图 18-2 所示。



图 18-2 认识组件

在整个 AWT 包中，所有的组件类（如按钮、文本框等）都是从 Component 和 MenuComponent 扩展而来的，这些类会继承这两个类的公共操作，继承关系分别如图 18-3 和图 18-4 所示。

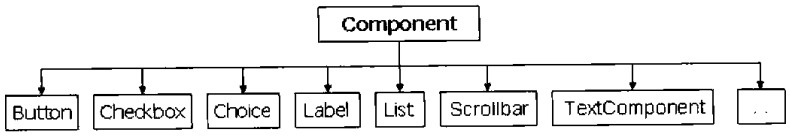


图 18-3 Component 继承关系

提示：Swing 中的组件。

在 Swing 中的所有组件类实际上也都是 Component 的子类，与 Component 不同的是，所有的组件前都加上了一个“JXx”的形式，如 JButton、JLabel 等。

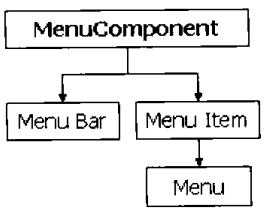


图 18-4 MenuComponent 继承关系

18.1.2　容器

所有的 AWT 组件都应该放到容器中，并可以设置其位置、大小等，所有的容器都是 Component 的子类，在 AWT 中包含如图 18-5 所示的几种容器。

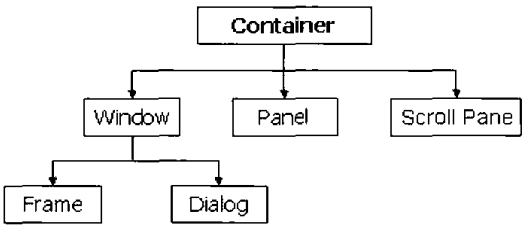


图 18-5　AWT 容器

18.1.3　布局管理器

使用布局管理器可以使容器中的组件按照指定的位置进行摆放，另外一个好处是，即使容器改变了大小，布局管理器也可以准确地把组件放到指定的位置，这样就可以有效地保证版面不会混乱。在 AWT 中所有的布局管理器都是 LayoutManager 的子类。布局管理类的继承关系如图 18-6 所示。

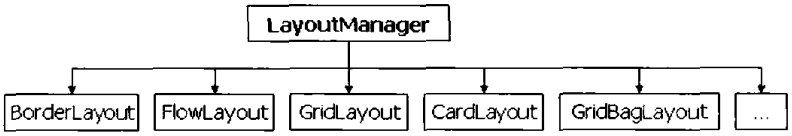


图 18-6　布局管理器的继承关系

18.2　Swing 简介

AWT 大量地引入了 Windows 函数，所以经常被称为重量级组件。在 Java 2 中提供了轻量级的图形界面组件——Swing，Swing 使用 Java 语言实现，是以 AWT 平台为基础构建起来的新组件，直接使用 Swing 可以更加轻松地构建用户界面。

在 Java 中所有的 Swing 都保存在 javax.swing 包中，从包的名称中（javax）可以清楚地发现此包是一个扩展包，所有的组件是从 JComponent 扩展出来的。此类实际上是 java.awt.Component 的子类，如图 18-7 所示。

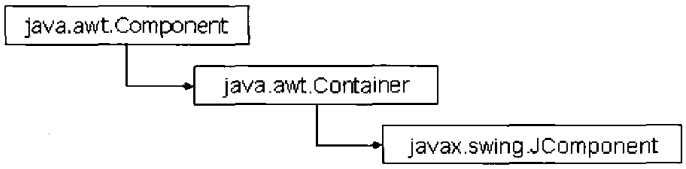


图 18-7　JComponent 的继承关系

JComponent 类几乎是所有 Swing 组件的公共超类。就像 Component 类是所有的 AWT

组件的父类一样，所以 JComponent 的所有子类也都继承了本类的全部公共操作，继承关系如图 18-8 所示。

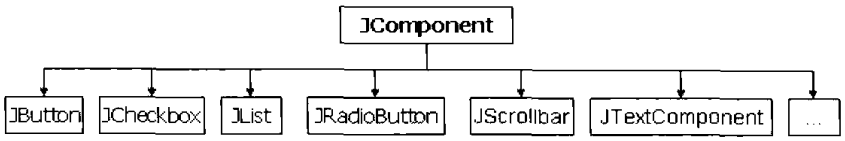


图 18-8　JComponent 的常用子类

从图 18-8 中各个组件类的定义来看，所有的 Swing 组件只是比 AWT 组件前多增加了一个字母“J”而已。

在 Swing 中依然存在容器的概念，所有的容器类都是继承自 AWT 组件包。例如，在 Swing 中容器使用 JFrame、JWindow、JPanel 等，这些分别是 Frame、Window、Panel 的子类。

在 Swing 中依然可以使用 AWT 中所提供的各个布局管理器，为组件进行统一的布局管理。要想使用布局管理器，则首先应该了解 Swing 中的容器及基本组件。

18.3　基本容器：JFrame

如果要使用 Swing 创建一个窗口，则直接使用 JFrame 类即可，此类是 Component 的子类，常用的操作方法如表 18-1 所示。

表 18-1　JFrame 类的常用操作方法

序号	方　法	类型	描　述
1	public JFrame() throws HeadlessException	构造	创建一个普通的窗体对象
2	public JFrame(String title) throws HeadlessException	构造	创建一个窗体对象，并指定标题
3	public void setSize(int width, int height)	普通	设置窗体大小
4	public void setSize(Dimension d)	普通	通过 Dimension 设置窗体大小
5	public void setBackground(Color c)	普通	设置窗体的背景颜色
6	public void setLocation(int x,int y)	普通	设置组件的显示位置
7	public void setLocation(Point p)	普通	通过 Point 来设置组件的显示位置
8	public void setVisible(boolean b)	普通	显示或隐藏组件
9	public Component add(Component comp)	普通	向容器中增加组件
10	public void setLayout(LayoutManager mgr)	普通	设置布局管理器，如果设置为 null 表示不使用
11	public void pack()	普通	调整窗口大小，以适合其子组件的首选大小和布局
12	public Container getContentPane()	普通	返回此窗体的容器对象

下面使用以上方法创建一个新的窗体，如下所示。

范例：创建一个新的窗体

```
package org.lxh.demo18.jframedemo;
import java.awt.Color;
```

```
import javax.swing.JFrame;
public class JFrameDemo01 {
    public static void main(String args[]) {
        JFrame f = new JFrame("第一个Swing窗体") ;    // 实例化窗体对象
        f.setSize(230, 80) ;                          // 设置窗体大小
        f.setBackground(Color.WHITE) ;                // 设置窗体的背景颜色
        f.setLocation(300, 200) ;                     // 设置窗体的显示位置
        f.setVisible(true) ;                          // 让组件显示
    }
}
```

程序运行结果：



以上程序运行之后，会直接显示出一个窗体，可以发现此窗体的标题就是在实例化 JFrame 时设置的标题，底色为白色，通过 setSize()方法设置了其显示的大小。在代码中最重要的就是 setVisible(true)，如果没有写这句话，则窗体不会显示。

在设置背景颜色时使用了 java.awt.Color 类完成，此类存在了大量的颜色常量，直接使用这些常量就可以改变颜色。

提示：此时的程序无法退出。

虽然现在的窗体上有关闭按钮，但是遗憾的是此按钮无法让程序退出。因为在 Swing 编程中，要想让窗口的关闭按钮起作用，则必须编写相关的事件进行处理，此处可以暂时使用 Ctrl+C 组合键退出程序。

在 JFrame 中也可以使用 Dimension 类设置窗体的大小，此类封装的是组件显示宽度和高度。Dimension 类的常用操作方法如表 18-2 所示。

表 18-2　Dimension 类的常用方法

序号	方　　法	类型	描　　述
1	public Dimension()	构造	创建一个 Dimension 实例
2	public void setSize(double width,double height)	普通	设置显示的宽和高
3	public void setSize(int width,int height)	普通	设置显示的宽和高
4	public double getWidth()	普通	返回组件的宽
5	public double getHeight()	普通	返回组件的高

除了设置组件大小可以使用 Dimension 外，对于组件的显示位置也存在一个 Point 类，此类封装的是显示位置的 X、Y 坐标。Point 类的常用操作方法如表 18-3 所示。

表 18-3 Point 类的常用方法

序号	方　法	类型	描　述
1	public Point()	构造	在坐标原点创建对象
2	public Point(int x,int y)	构造	在指定的坐标点创建对象
3	public void setLocation(double x,double y)	普通	设置 X、Y 坐标
4	public void setLocation(int x,int y)	普通	设置 X、Y 坐标
5	public void move(int x,int y)	普通	将此组件移动到指定坐标位置，此方法与 setLocation(int,int)相同
6	public void translate(int dx,int dy)	普通	平移(x,y)位置的点，沿 x 轴平移 dx，沿 y 轴平移 dy，移动后得到点(x+dx,y+dy)

范例：使用 Dimension 和 Point 类设置组件大小和显示位置

```
package org.lxh.demo18.jframedemo;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Point;
import javax.swing.JFrame;
public class JFrameDemo02 {
    public static void main(String args[]) {
        JFrame f = new JFrame("第一个Swing窗体") ;    // 实例化窗体对象
        Dimension d = new Dimension() ;              // 实例化Dimension对象
        d.setSize(230, 80) ;                         // 设置大小
        f.setSize(d) ;                               // 设置组件大小
        f.setBackground(Color.WHITE) ;               // 设置窗体的背景颜色
        Point p = new Point(300,200) ;               // 设置显示的坐标点
        f.setLocation(p) ;                           // 设置窗体的显示位置
        f.setVisible(true) ;                         // 让组件显示
    }
}
```

本程序的运行结果与之前是完全一样的。以上两个程序，为读者简单地演示了 Swing 编程的基本形式，而且一个 JFrame 就表示一个容器，在容器上可以安装多个组件。

提问：为什么在实例化 Frame 对象时不用异常处理？

在 Frame 类的构造方法上使用了 throws 关键字抛出异常，为什么在此时的程序中却没有使用 try…catch 进行处理呢？

回答：因为 HeadlessException 是 RuntimeException 的子类。

在之前讲解异常处理部分时曾经为读者讲解过，在 Java 的异常处理中只要是 RuntimeException 类型的异常，则程序不使用 try…catch 处理也是可以的。

大概了解了窗体的概念之后，下面来看两个基础组件的使用，这样可以让读者更加轻

松地理解容器和组件间的关系。

18.4　标签组件：JLabel

18.4.1　认识 JLabel

JLabel 组件表示的是一个标签，本身是用于显示信息的，一般情况下是不能直接更改其显示内容的。创建完的 Label 对象可以通过 Container 类中的 add()方法加入到容器中，JLabel 类的常用方法和常量如表 18-4 所示。

表 18-4　JLabel 类的常用方法和常量

序号	方法及常量	类型	描　　述
1	public static final int LEFT	常量	标签文本左对齐
2	public static final int CENTER	常量	标签文本居中对齐
3	public static final int RIGHT	常量	标签文本右对齐
4	public JLabel() throws HeadlessException	构造	创建一个 JLabel 对象
5	public JLabel(String text) throws HeadlessException	构造	创建一个标签并指定文本内容，默认为左对齐
6	public Label(String text,int alignment) throws HeadlessException	构造	创建一个标签并指定文本内容以及对齐方式，可以使用 JLabel.LEFT、JLabel.RIGHT、JLabel.CENTER 3 个值
7	public JLabel(String text,Icon icon,int horizontalAlignment)	构造	创建具有指定文本、图像和水平对齐方式的 JLabel 对象
8	public JLabel(Icon image,int horizontalAlignment)	构造	创建具有指定图像和水平对齐方式的 JLabel 实例
9	public void setText(String text)	普通	设置标签的文本
10	public String getText()	普通	取得标签的文本
11	public void setAlignment(int alignment)	普通	设置标签的对齐方式
12	public void setIcon(Icon icon)	普通	设置指定的图像

范例：使用一个标签

```
package org.lxh.demo18.jlabeldemo;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Point;
import javax.swing.JFrame;
import javax.swing.JLabel;
public class JLabelDemo01 {
    public static void main(String args[]) {
        JFrame frame = new JFrame("Welcome To MLDN") ; // 实例化窗体对象
```

```
        JLabel lab = new JLabel("MLDN", JLabel.CENTER);// 实例化对象，使用居
                                                        中对齐
        frame.add(lab) ;                                // 向容器中加入组件
        Dimension dim = new Dimension() ;               // 实例化Dimension
                                                        对象
        dim.setSize(200, 70) ;                          // 设置大小
        frame.setSize(dim) ;                            // 设置组件大小
        frame.setBackground(Color.WHITE) ;              // 设置窗体的背景颜色
        Point point = new Point(300,200) ;              // 设置显示的坐标点
        frame.setLocation(point) ;                      // 设置窗体的显示位置
        frame.setVisible(true) ;                        // 让组件显示
    }
}
```

程序运行结果：



以上程序中，实例化了一个 JLabel 对象，并指定内容以及对齐方式，之后将此组件加入到了容器中，并进行显示。

如果非要在以上代码中体现容器的概念，则可以直接通过 JFrame 中的 getContentPane() 取得窗体的 Container 对象，并在此对象中加入组件，如下所示：

```
Container cont = frame.getContentPane() ;       // 取得窗体的容器
cont.add(lab) ;                                 // 在容器中加入组件
```

18.4.2 更改 JLabel 的文字样式

以上的标签内容只是使用了默认的字体及颜色显示，如果现在要更改使用的字体，则可以直接使用 Component 类中定义的以下方法：

```
public void setFont(Font f)
```

在设置时使用了 java.awt.Font 类来表示字体，Font 类的常用操作方法及常量如表 18-5 所示。

表 18-5 Font 类的常用操作方法及常量

序号	方法及常量	类型	描　述
1	public static final int BOLD	常量	文字显示为粗体
2	public static final int ITALIC	常量	文字显示风格为斜体
3	public static final int PLAIN	常量	文字显示风格为普通样式
4	public Font(String name,int style,int size)	构造	实例化对象，指定显示风格及大小
5	public String getFontName()	普通	得到字体的名称

范例：设置标签的显示字体、大小、背景颜色

```
package org.lxh.demo18.jlabeldemo;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.Point;
import javax.swing.JFrame;
import javax.swing.JLabel;
public class JLabelDemo02 {
    public static void main(String args[]) {
        JFrame frame = new JFrame("Welcome To MLDN") ; // 实例化窗体对象
        JLabel lab = new JLabel("MLDN", JLabel.CENTER);// 实例化对象，使用居
                                                       中对齐
        Font fnt = new Font("Serief", Font.ITALIC + Font.BOLD, 28);
        lab.setFont(fnt) ;                             // 设置标签的显示字体
        lab.setForeground(Color.RED) ;                 // 设置标签的文字颜色
        frame.add(lab) ;                               // 向容器中加入组件
        Dimension dim = new Dimension() ;              // 实例化Dimension
                                                       对象
        dim.setSize(200, 70) ;                         // 设置大小
        frame.setSize(dim) ;                           // 设置组件大小
        frame.setBackground(Color.WHITE) ;             // 设置窗体的背景颜色
        Point point = new Point(300,200) ;             // 设置显示的坐标点
        frame.setLocation(point) ;                     // 设置窗体的显示位置
        frame.setVisible(true) ;                       // 让组件显示
    }
}
```

程序运行结果：



以上标签显示后，文字设置成了红色，而且通过 Font 指定了其字体风格以及大小，在本次操作中使用的是粗体+斜体的文字显示风格。

提问：如何取得本机的全部字体？

现在的程序中使用了一种字体样式，如果想取得本机的全部字体该如何编写程序呢？

回答：使用 GraphicsEnvironment 类完成。

java.awt.GraphicsEnvironment 可以返回本机的全部绘图设备和全部字体的集合。本类是抽象类，要想实例化对象，直接使用类中的 getLocalGraphicsEnvironment()方法即可，之后通过 getAvailableFontFamilyNames()方法就可以取得全部的可用字体，此程序代码如下：

范例： 取得本机中的全部可用字体

```
import java.awt.GraphicsEnvironment;
public class GetAllFonts {
    public static void main(String[] args) {
        // 取得本地的绘图设备和字体的集合
        GraphicsEnvironment eq = GraphicsEnvironment
                .getLocalGraphicsEnvironment();
        // 取得全部可用的字体
        String[] fontNames = eq.getAvailableFontFamilyNames();
        // 循环输出全部的内容
        for (int x = 0; x < fontNames.length; x++) {
            // 输出每一个字体名称
            System.out.println(fontNames[x]);
        }
    }
}
```

程序运行结果（部分）：

```
宋体
新宋体
楷体_GB2312
黑体
...
```

18.4.3 在 JLabel 中设置图片

如果现在想将一个图像设置到 JLabel 中也是可以的，直接使用 Icon 接口以及 ImageIcon 子类即可，在 ImageIcon 中可以使用如表 18-6 所示的构造方法，将图像的数据以 byte 数组的形式设置上去。

表 18-6 ImageIcon 类的构造方法

序号	方　法	类型	描　述
1	public ImageIcon(byte[] imageData)	构造	将保存图片信息的 byte 数组设置到 ImageIcon 中
2	public ImageIcon(String filename)	构造	通过文件名称创建 ImageIcon 对象
3	public ImageIcon(String filename,String description)	构造	设置图片路径以及图片的简单描述

下面先来看第一个构造方法的使用，如果要从一个图像文件中取得字节数组，则就必须使用 InputStream 类完成操作。此时图像的路径为 f:\mldn.gif。

范例： 在标签上设置图像

```
package org.lxh.demo18.jlabeldemo;
import java.awt.Color;
```

```
import java.awt.Font;
import java.io.File;
import java.io.FileInputStream;
import java.io.InputStream;
import javax.swing.Icon;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;
public class JLabelDemo03 {
    public static void main(String args[]) {
        JFrame frame = new JFrame("Welcome To MLDN") ; // 实例化窗体对象
        String picPath = "f:" + File.separator + "mldn.gif" ;
        File file = new File(picPath);                 // 实例化File对象
        InputStream input = null ;                     // 输入流对象
        byte b[] = new byte[(int)file.length()];       // 根据图片大小开辟
                                                          byte数组
        try {
            input = new FileInputStream(file) ;        // 实例化输入流对象
            input.read(b) ;                            // 读取文件信息
            input.close() ;                            // 关闭输入流
        } catch (Exception e) {
            e.printStackTrace();
        }
        Icon icon = new ImageIcon(b) ;                 // 实例化Icon对象
        JLabel lab = null ;
        lab = new JLabel("MLDN", icon, JLabel.CENTER); // 实例化对象，使用居
                                                          中对齐
        Font fnt = new Font("Serief", Font.ITALIC + Font.BOLD, 28);
        lab.setFont(fnt) ;                             // 设置标签的显示字体
        lab.setBackground(Color.YELLOW) ;              // 设置标签的背景颜色
        lab.setForeground(Color.RED) ;                 // 设置标签的文字颜色
        frame.add(lab) ;                               // 向容器中加入组件
        frame.setSize(300, 160) ;                      // 设置组件大小
        frame.setBackground(Color.WHITE) ;             // 设置窗体的背景颜色
        frame.setLocation(300,200) ;                   // 设置窗体的显示位置
        frame.setVisible(true) ;                       // 让组件显示
    }
}
```

程序运行结果：



如果使用一个 IO 流进行图片读取来设置 JLabel 显示图形太麻烦，则可以直接将一个图片的路径传递到 ImageIcon 的实例化中，并使用 JLabel 类中可以设置 ImageIcon 的构造方法即可，如下所示。

范例：另一种 ImageIcon 的使用

```
package org.lxh.demo18.jlabeldemo;
import java.awt.Color;
import java.io.File;
import javax.swing.Icon;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;
public class JLabelDemo04 {
    public static void main(String args[]) {
        JFrame frame = new JFrame("Welcome To MLDN") ; // 实例化窗体对象
        String picPath = "f:" + File.separator + "mldn.gif" ;
        Icon icon = new ImageIcon(picPath) ;           // 实例化Icon对象
        JLabel lab = null ;
        lab = new JLabel(icon, JLabel.CENTER);         // 实例化对象，使用居
                                                          中对齐
        lab.setBackground(Color.YELLOW) ;              // 设置标签的背景颜色
        lab.setForeground(Color.RED) ;                 // 设置标签的文字颜色
        frame.add(lab) ;                               // 向容器中加入组件
        frame.setSize(300, 160) ;                      // 设置组件大小
        frame.setBackground(Color.WHITE) ;             // 设置窗体的背景颜色
        frame.setLocation(300,200) ;                   // 设置窗体的显示位置
        frame.setVisible(true) ;                       // 让组件显示
    }
}
```

程序运行结果：



以上两段代码完成的功能是一样的，如果此时图像是从一个不确定输入流（如从数据库中读取 BLOB 字段）中而来，则要使用第一种方法；如果图片是从文件中而来，那么就使用第二种方法会很方便。

18.5 按钮组件：JButton

JButton 组件表示一个普通的按钮，使用此类可以直接在窗体中增加一个按钮。JButton

类的常用方法如表 18-7 所示。

表 18-7　JButton 类的常用方法

序号	方　　法	类型	描　　述
1	public JButton() throws HeadlessException	构造	创建一个 Button 对象
2	public JButton(String label) throws HeadlessException	构造	创建一个 Button 对象，同时指定其显示内容
3	public JButton(Icon icon)	构造	创建一个带图片的按钮
4	public JButton(String text,Icon icon)	构造	创建一个带图片和文字的按钮
5	public void setLabel(String label)	普通	设置 Button 的显示内容
6	public String getLabel()	普通	得到 Button 的显示内容
7	public void setBounds(int x,int y,int width,int height)	普通	设置组件的大小及显示方式
8	public void setMnemonic(int mnemonic)	普通	设置按钮的快捷键

范例：建立一个普通的按钮

```
package org.lxh.demo18.jbuttondemo;
import java.awt.Color;
import java.awt.Font;
import javax.swing.JButton;
import javax.swing.JFrame;
public class JButtonDemo01 {
    public static void main(String args[]) {
        JFrame frame = new JFrame("Welcome To MLDN") ; // 实例化窗体对象
        JButton but = new JButton("按我") ;             // 定义按钮对象
        Font fnt = new Font("Serief", Font.BOLD, 28);  // 设置按钮的显示字体
        but.setFont(fnt) ;                             // 设置按钮的显示字体
        frame.add(but) ;                               // 向容器中加入组件
        frame.setSize(200, 70) ;                       // 设置窗体大小
        frame.setLocation(300,200) ;                   // 设置窗体的显示位置
        frame.setVisible(true) ;                       // 让组件显示
    }
}
```

程序运行结果：



从程序代码来看，一个按钮的显示是非常容易的，而且使用起来与 JLabel 没有任何区别。JButton 也可以为一个按钮设置一张显示图片，与 JLabel 类似，直接在创建按钮对象时设置即可，图像文件的保存路径为 f:\mldnjava.gif。

范例：在按钮上显示图片

```
package org.lxh.demo18.jbuttondemo;
import java.io.File;
```

```
import javax.swing.Icon;
import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JFrame;
public class JButtonDemo02 {
    public static void main(String args[]) {
        JFrame frame = new JFrame("Welcome To MLDN") ; // 实例化窗体对象
        String picPath = "f:" + File.separator + "mldnjava.gif" ;
        Icon icon = new ImageIcon(picPath , "MLDN") ;  // 实例化Icon对象
        JButton but = new JButton(icon) ;              // 定义按钮对象
        frame.add(but) ;                               // 向容器中加入组件
        frame.setSize(206, 78) ;                       // 设置窗体大小
        frame.setLocation(300,200) ;                   // 设置窗体的显示位置
        frame.setVisible(true) ;                       // 让组件显示
    }
}
```

程序运行结果：



18.6 布局管理器

了解了基本的窗体及两个组件之后，可以发现，如果不对窗体进行版面设置，则一个组件会直接充满整个窗体，所以在 Java 中专门提供了布局管理器来管理组件，通过布局管理器可以使用不同方式的排列组件。每当需要重新调整屏幕大小或重新绘制屏幕上任一项时，都要用到布局管理器。在 Swing 中使用的所有布局管理器都可以实现 LayoutManager 接口，在 Swing 中主要使用以下 5 种常见的布局管理器：FlowLayout、BorderLayout、GridLayout、CardLayout、绝对定位。

18.6.1 FlowLayout

FlowLayout 属于流式布局管理器，使用此种布局方式会使所有的组件像流水一样依次进行排列，FlowLayout 类的常用方法及常量如表 18-8 所示。

表 18-8 FlowLayout 类的常用方法及常量

序号	方法及常量	类型	描　述
1	public static final int CENTER	常量	居中对齐
2	public static final int LEADING	常量	与容器的开始端对齐方式一样
3	public static final int LEFT	常量	左对齐

续表

序号	方法及常量	类型	描　　述
4	public static final int RIGHT	常量	右对齐
5	public static final int TRAILING	常量	与容器的结束端对齐方式一样
6	public FlowLayout()	构造	构造一个新的 FlowLayout，居中对齐，默认的水平和垂直间距是 5 个单位
7	public FlowLayout(int align)	构造	构造一个 FlowLayout，并指定对齐方式
8	public FlowLayout(int align,int hgap,int vgap)	构造	指定对齐方式、水平、垂直间距

范例：设置 FlowLayout

```
package org.lxh.demo18.layoutdemo;
import java.awt.FlowLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
public class FlowLayoutDemo01 {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Welcome To MLDN") ; // 实例化窗体对象
        // 设置窗体中的布局管理器为FlowLayout，所有的组件居中对齐，水平和垂直间距为3
        frame.setLayout(new FlowLayout(FlowLayout.CENTER, 3, 3));
        JButton but = null;
        for (int i = 0; i < 9; i++) {
            but = new JButton("按钮 - "+i) ;
            frame.add(but) ;                                    // 加入按钮
        }
        frame.setSize(280,123) ;                                // 设置窗体大小
        frame.setVisible(true) ;                                // 设置窗体可见
    }
}
```

程序运行结果：



从程序的运行结果中可以发现，所有的组件按照顺序依次向下排列，每个组件之间的顺序是 3。之前讲解的 JPanel 就是采用此种布局方式，所以显示组件时顺序地向 JPanel 中加入。

18.6.2　BorderLayout

BorderLayout 将一个窗体的版面划分成东、西、南、北、中 5 个区域，可以直接将需要的组件放到这 5 个区域中。BorderLayout 的常用方法及常量如表 18-9 所示。

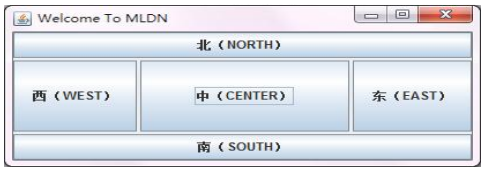
表 18-9 BorderLayout 类的常用方法及常量

序号	方法及常量	类型	描　述
1	public static final String EAST	常量	将组件设置在东区域
2	public static final String WEST	常量	将组件设置在西区域
3	public static final String SOUTH	常量	将组件设置在南区域
4	public static final String NORTH	常量	将组件设置在北区域
5	public static final String CENTER	常量	将组件设置在中区域
6	public BorderLayout()	构造	构造没有间距的布局器
7	public BorderLayout(int hgap,int vgap)	构造	构造有水平和垂直间距的布局器

范例：设置 BorderLayout

```
package org.lxh.demo18.layoutdemo;
import java.awt.BorderLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
public class BorderLayoutDemo01 {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Welcome To MLDN") ; // 实例化窗体对象
        // 设置窗体中的布局管理器为BorderLayout，所有的组件水平和垂直间距为3
        frame.setLayout(new BorderLayout( 3, 3));
        frame.add(new JButton("东（EAST）"), BorderLayout.EAST);
        frame.add(new JButton("西（WEST）"), BorderLayout.WEST);
        frame.add(new JButton("南（SOUTH）"), BorderLayout.SOUTH);
        frame.add(new JButton("北（NORTH）"), BorderLayout.NORTH);
        frame.add(new JButton("中（CENTER）"), BorderLayout.CENTER);
        frame.pack() ;                                    // 根据组件自动调整窗
                                                             体大小
        frame.setVisible(true) ;                          // 设置窗体可见
    }
}
```

程序运行结果：



18.6.3 GridLayout

GridLayout 布局管理器是以表格的形式进行管理的，在使用此布局管理器时必须设置显示的行数和列数，常用的方法如表 18-10 所示。

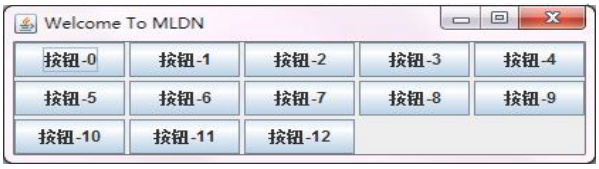
表 18-10　GridLayout 类的常用方法

序号	方　　法	类型	描　　述
1	public GridLayout(int rows,int cols)	构造	构造一个指定行和列的布局管理器
2	public GridLayout(int rows,int cols,int hgap,int vgap)	构造	构造时指定行和列、水平和垂直间距

范例：设置 GridLayout

```
package org.lxh.demo18.layoutdemo;
import java.awt.GridLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
public class GridLayoutDemo01 {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Welcome To MLDN") ; // 实例化窗体对象
        // 设置窗体中的布局管理器为GridLayout，按3×5进行排列，水平和垂直间距为3
        frame.setLayout(new GridLayout(3, 5, 3, 3));
        JButton but = null;
        for (int i = 0; i < 13; i++) {
            but = new JButton("按钮 - "+i) ;
            frame.add(but) ;                                    // 加入按钮
        }
        frame.pack() ;                                          // 根据组件自动调整窗
                                                                   体大小
        frame.setVisible(true) ;                                // 设置窗体可见
    }
}
```

程序运行结果：



提示：在计算器的操作上可以使用 GridLayout 布局管理器。

在 Windows 的计算器程序中可以发现有多个按钮，那么此时实际上就可以通过 GridLayout 布局管理摆放全部的按钮。



18.6.4 CardLayout

CardLayout 就是将一组组件彼此重叠地进行布局，就像一张张卡片一样，这样每次只会展现一个界面。CardLayout 类的常用方法如表 18-11 所示。

表 18-11 CardLayout 类的常用方法

序号	方 法	类型	描 述
1	public CardLayout()	构造	构造 CardLayout 对象，各组件间距为 0
2	public CardLayout(int hgap,int vgap)	构造	构造 CardLayout 对象，指定组件间距
3	public void next(Container parent)	普通	翻转到下一张卡片
4	public void previous(Container parent)	普通	翻转到上一张卡片
5	public void first(Container parent)	普通	翻转到第一张卡片
6	public void last(Container parent)	普通	翻转到最后一张卡片
7	public void show(Container parent,String name)	普通	显示具有指定组件名称的卡片

范例：设置 CardLayout

```
package org.lxh.demo18.layoutdemo;
import java.awt.CardLayout;
import java.awt.Container;
import javax.swing.JFrame;
import javax.swing.JLabel;
public class CardLayoutDemo01 {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Welcome To MLDN") ; // 实例化窗体对象
        Container cont = frame.getContentPane() ;      // 取得窗体容器
        CardLayout card = new CardLayout() ;           // 定义布局管理器
        frame.setLayout(card);                         // 设置布局管理器
        cont.add(new JLabel("标签-A",JLabel.CENTER),"first") ;
        cont.add(new JLabel("标签-B",JLabel.CENTER),"second") ;
        cont.add(new JLabel("标签-C",JLabel.CENTER),"third") ;
        cont.add(new JLabel("标签-D",JLabel.CENTER),"fourth") ;
        cont.add(new JLabel("标签-E",JLabel.CENTER),"fifth") ;
        frame.pack() ;                                 // 根据组件自动调整窗
                                                          体大小
        frame.setVisible(true) ;                       // 设置窗体可见
        card.show(cont, "third") ;                     // 显示第3张卡片
        for (int i = 0; i < 5; i++) {                  // 循环显示每张卡片
            try {
                Thread.sleep(3000) ;                   // 加入显示延迟
            } catch (InterruptedException e) {
            }
            card.next(cont) ;                          // 从容器中取出组件
```

```
            }
        }
    }
```

程序运行结果：



以上内容在显示时首先会显示第 3 张卡片，之后循环显示每一张卡片。

18.6.5　绝对定位

如果不想在窗体中指定布局管理器，也可以通过设置绝对坐标的方式完成。在 Component 中提供了 setBounds()方法，可以定位一个组件的坐标，使用 X、Y 的坐标表示方式。此方法定义如下：

```
public void setBounds(int x,int y,int width,int height)
```

在每一个组件中都存在以上方法，通过它可以设置每一个组件的具体位置，如图 18-9 所示。

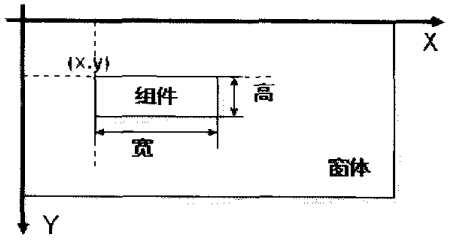


图 18-9　设置组件的显示坐标及大小

范例：使用绝对定位排版

```
package org.lxh.demo18.layoutdemo;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
public class AbsoluteLayoutDemo01 {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Welcome To MLDN") ; // 实例化窗体对象
        frame.setLayout(null);                          // 使用绝对定位
        JLabel title = new JLabel("www.mldnjava.cn") ; // 建立标签对象
        JButton enter = new JButton("进入") ;           // 建立按钮对象
        JButton help = new JButton("帮助") ;            // 建立按钮对象
        frame.setSize(200, 90);                         // 设置窗体大小
        title.setBounds(45, 5, 150, 20) ;               // 设置组件位置及大小
        enter.setBounds(10, 30, 80, 20);                // 设置组件位置及大小
        help.setBounds(100, 30, 80, 20);                // 设置组件位置及大小
```

```
            frame.add(title) ;                                    // 加入组件
            frame.add(enter) ;                                    // 加入组件
            frame.add(help) ;                                     // 加入组件
            frame.setVisible(true) ;                              // 设置窗体可见
        }
    }
```

程序运行结果：



使用绝对定位有一点好处就是，不管窗体如何改变大小，组件的位置是固定不动的；而之前的各种排版方式，当窗体改变时，组件大小也会跟着改变。

18.7 其他容器

在 Swing 开发中，窗体部分除了可以使用 JFrame 表示之外，还有其他几种常见的窗体：JPanel、JSplitPane、JTabbedPane、JScrollPane、JDesktopPane、JInternalFrame 等，下面分别进行介绍。

18.7.1 JPanel

JPanel 也是一种经常使用到的容器之一，可以使用 JPanel 完成各种复杂的界面显示。在 JPanel 中可以加入任意的组件，之后直接将 JPanel 容器加入到 JFrame 容器中即可显示。JPanel 常用的方法如表 18-12 所示。

表 18-12 JPanel 类的常用方法

序号	方 法	类型	描 述
1	public JPanel()	构造	创建一个默认的 JPanel 对象，使用流布局管理
2	public JPanel(LayoutManager layout)	构造	创建一个指定布局管理器的 JPanel 对象

范例：JPanel 的基本使用

```
package org.lxh.demo18.jpaneldemo;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
public class JPanelDemo01 {
    public static void main(String args[]) {
        JFrame frame = new JFrame("Welcome To MLDN") ; // 实例化窗体对象
        JPanel pan = new JPanel() ;                    // 实例化JPanel对象
```

```
            pan.add(new JLabel("标签-A")) ;          // 加入标签组件
            pan.add(new JLabel("标签-B")) ;          // 加入标签组件
            pan.add(new JLabel("标签-C")) ;          // 加入标签组件
            pan.add(new JButton("按钮-X")) ;         // 加入按钮组件
            pan.add(new JButton("按钮-Y")) ;         // 加入按钮组件
            pan.add(new JButton("按钮-Z")) ;         // 加入按钮组件
            frame.add(pan) ;                         // 将JPanel加入到窗体
            frame.pack() ;                           // 根据组件自动调整窗体大小
            frame.setLocation(300,200) ;             // 设置窗体的显示位置
            frame.setVisible(true) ;                 // 让组件显示
        }
    }
```

程序运行结果：



从程序的运行结果中可以发现，所有的组件是采用顺序的形式加入到 JPanel 中，最后再将 JPanel 加入到 JFrame 中，它们之间的关系如图 18-10 所示。

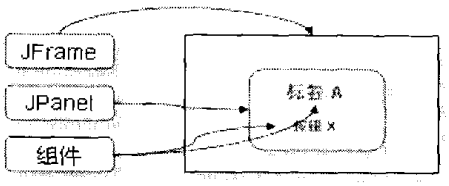


图 18-10　JFrame、JPanel、组件的关系

提示：可以将多个组件加入到 JPanel 中实现复杂的排列。

使用 JPanel 并结合之前的布局管理器可以更加方便地管理组件，可以使用布局管理器对多个 JPanel 进行排列，而每个 JPanel 中也可以分别使用不同的布局管理器管理组件，这样就可以对组件进行复杂的排列。

18.7.2　JSplitPane

JSplitPane 的主要功能是分割面板，可以将一个窗体分为两个子窗体，可以是水平排列也可以是垂直排列，如图 18-11 所示。

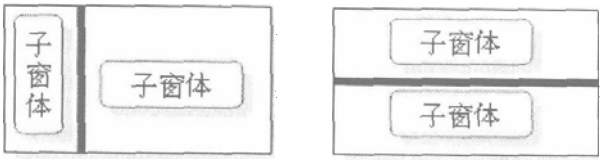


图 18-11　按行或列进行拆分

JSplitPane 的主要方法及常量如表 18-13 所示。

表 18-13 JSplitPane 类的主要方法及常量

序号	方法及常量	类型	描 述
1	public static final int HORIZONTAL_SPLIT	常量	表示水平分割
2	public static final int VERTICAL_SPLIT	常量	表示垂直分割
3	public JSplitPane(int newOrientation)	构造	创建对象，并指明分割方式
4	Public JSplitPane(int newOrientation,boolean newContinuousLayout, Component newLeftComponent, Component newRightComponent)	构造	创建对象、指明分割方式、分割条改变是否重绘图像以及两端的显示组件
5	public void setDividerLocation(double proportionalLocation)	普通	按百分比设置分割条的位置
6	public void setOneTouchExpandable(boolean newValue)	普通	设置是否提供快速展开/折叠的功能
7	public void setDividerSize(int newSize)	普通	设置分割条大小

下面使用分割面板的方式完成一个上下、左右的分割程序，程序的实现思路如图 18-12 所示。

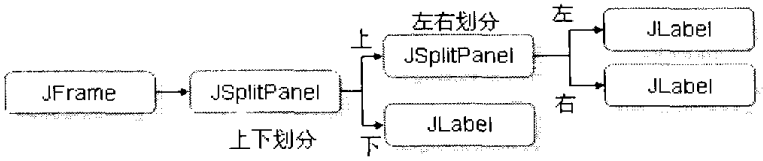


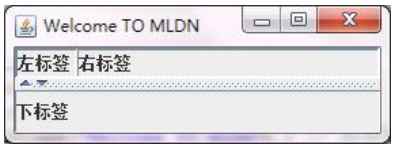
图 18-12 程序的实现思路

范例：实现复杂分割

```
package org.lxh.demo18.jsplitdemo;
import java.awt.Container;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JSplitPane;
public class JSplitPaneDemo01 {
    public static void main(String args[]) {
        JFrame frame = new JFrame("Welcome To MLDN") ;// 实例化窗体对象
        Container cont = frame.getContentPane() ;      // 得到窗体容器
        JSplitPane lfsplit = null ;                    // 左右分割的
                                                          SplitPanel
        JSplitPane tbsplit = null ;                    // 上下分割的
                                                          SplitPanel
        lfsplit = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT, new JLabel("
        左标签"),
                new JLabel("右标签"));                 // 设置水平分割
        lfsplit.setDividerSize(3) ;                    // 设置左右的分割线大小
        tbsplit = new JSplitPane(JSplitPane.VERTICAL_SPLIT, lfsplit,
                new JLabel("下标签"));          // 设置垂直分割
        tbsplit.setDividerSize(10) ;            // 设置上下的分割线大小
```

```
            tbsplit.setOneTouchExpandable(true) ;   // 快速展开/折叠分隔条
            cont.add(tbsplit) ;                     // 将JSplitPane加入到窗体
            frame.setSize(230, 80) ;                // 设置窗体大小
            frame.setLocation(300,200) ;            // 设置窗体的显示位置
            frame.setVisible(true) ;                // 让组件显示
        }
    }
```

程序运行结果：



从程序的运行结果中可以发现，左右的分割面板的分割线长度比较小，而上下的分割面板的分割线长度大一些，并且设置了快速展开/折叠功能。

18.7.3　JTabbedPane

JTabbedPane 是在一个面板上设置多个选项卡供用户选择，如图 18-13 所示。



图 18-13　【系统属性】对话框中的选项卡

如果需要查看哪方面的信息，则直接选择选项卡即可浏览。通过这种方式用户可以方便地管理自己的相关信息。JTabbedPane 类的常用方法及常量表 18-14 所示。

表 18-14　JTabbedPane 类的常用方法及常量

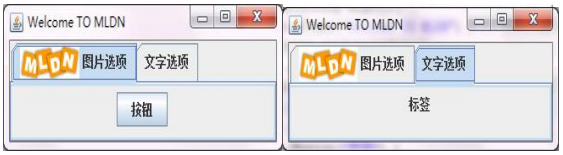
序号	方法及常量	类型	描　　述
1	static final int TOP	常量	表示指向框顶部位置
2	static final int BOTTOM	常量	表示指向框底部位置
3	static final int LEFT	常量	表示指向框左部位置
4	static final int RIGHT	常量	表示指向框右部位置
5	public JTabbedPane(int tabPlacement)	构造	创建对象，并指定选项卡布局
6	public void addTab(String title,Component component)	普通	添加一个有标题而没有图标的组件
7	public void addTab(String title,Icon icon,Component component)	普通	添加一个有标题、有图标的组件
8	public void addTab(String title,Icon icon,Component component,String tip)	普通	添加一个有标题、有图标、有提示信息的组件

范例：设置选项卡

```
package org.lxh.demo18.jtabbeddemo;
import java.awt.Container;
import java.io.File;
import javax.swing.ImageIcon;
```

```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTabbedPane;
public class JTabbedPaneDemo01 {
    public static void main(String args[]) {
        JFrame frame = new JFrame("Welcome To MLDN") ; // 实例化窗体对象
        Container cont = frame.getContentPane() ;      // 得到窗体容器
        JTabbedPane tab = null ;
        tab = new JTabbedPane(JTabbedPane.TOP) ;       // 设置标签在顶部显示
        JPanel pan1 = new JPanel() ;                   // 设置面板
        JPanel pan2 = new JPanel() ;                   // 设置面板
        JButton but = new JButton("按钮") ;            // 定义一个按钮
        JLabel lab = new JLabel("标签") ;              // 定义一个标签
        pan1.add(but) ;                                // 第一个面板加入一个
                                                          按钮
        pan2.add(lab) ;                                // 第二个面板加入一个
                                                          标签
        String picPath = "f:" + File.separator + "logo.gif" ;
        tab.addTab("图片选项", new ImageIcon(picPath), pan1, "图像");
        tab.addTab("文字选项", pan2) ;                 // 加入两个组件
        cont.add(tab) ;                                // 加入到容器中
        frame.setSize(230, 120) ;                      // 设置窗体大小
        frame.setLocation(300,200) ;                   // 设置窗体的显示位置
        frame.setVisible(true) ;                       // 让组件显示
    }
}
```

程序运行结果：



以上程序中建立了两个选项卡，并且将选项卡放在了窗体顶部。为了让读者清楚地知道其具体应用，所以每一个选项卡都加入了一个 JPanel 面板，层次关系如图 18-14 所示。

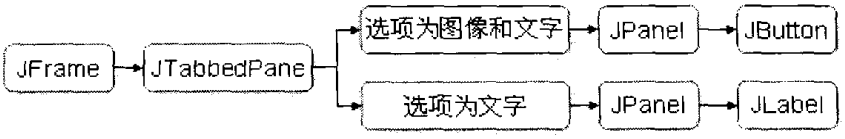


图 18-14　层次关系

18.7.4　JScrollPane

在一般的图形界面中如果显示的区域不够大，往往会出现滚动条以方便用户浏览，在 Swing 中 JScrollPane 的主要功能就是为显示的内容加入水平滚动条。JScrollPane 主要由 JViewport 和 JScrollBar 两部分组成，前者主要是显示一个矩形的区域让用户浏览，而后者主要是形成水平或垂直的滚动条，它们的关系如图 18-15 所示。但在开发中用户一般情况下不经常使用这两个类，因为在 JScrollPane 中已经为开发者提供了足够的功能。在表 18-15 中列出了 JScrollPane 常用的方法及常量。

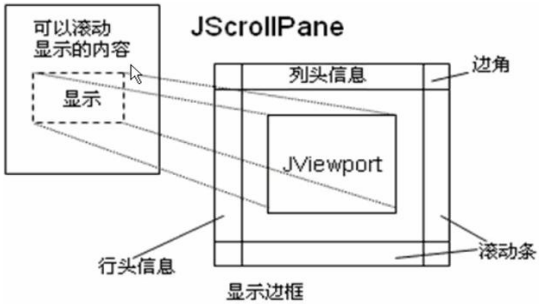


图 18-15　JScrollPane 容器

表 18-15　JScrollPane 的常用方法及常量

序号	方法及常量	类型	描述
1	static final int HORIZONTAL_SCROLLBAR_ALWAYS	常量	始终显示水平滚动条
2	static final int HORIZONTAL_SCROLLBAR_NEVER	常量	任何情况下都不显示水平滚动条
3	static final int HORIZONTAL_SCROLLBAR_AS_NEEDED	常量	根据自身需要显示水平滚动条
4	static final int VERTICAL_SCROLLBAR_ALWAYS	常量	始终显示垂直滚动条
5	static final int VERTICAL_SCROLLBAR_NEVER	常量	任何情况下都不显示垂直滚动条
6	static final int VERTICAL_SCROLLBAR_AS_NEEDED	常量	根据自身需要显示垂直滚动条
7	public JScrollPane(Component view)	构造	将指定的组件加入滚动条，根据大小显示水平或垂直滚动条
8	public JScrollPane(Component view,int vsbPolicy,int hsbPolicy)	构造	将指定的组件加入滚动条，根据需要设置是否显示垂直或水平滚动条
9	public void setHorizontalScrollBarPolicy(int policy)	普通	设置水平滚动条的显示策略
10	public void setVerticalScrollBarPolicy(int policy)	普通	设置垂直滚动条的显示策略

范例：为图片设置滚动条

```
package org.lxh.demo18.jscrolldemo;
import java.awt.Container;
import java.io.File;
import javax.swing.Icon;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
```

```
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
public class JScrollPaneDemo01 {
    public static void main(String args[]) {
        JFrame frame = new JFrame("Welcome To MLDN") ; // 实例化窗体对象
        Container cont = frame.getContentPane() ;      // 得到窗体容器
        String picPath = "f:" + File.separator + "mldn.gif" ;
        Icon icon = new ImageIcon(picPath) ;           // 实例化Icon对象
        JPanel pan = new JPanel() ;                    // 定义一个面板
        JLabel lab = new JLabel(icon) ;                // 定义一个标签，显示
                                                       图片
        pan.add(lab) ;                                 // 将标签加到面板中
        JScrollPane scr1 = null ;                      // 声明滚动面板
        // 垂直滚动条始终会显示，水平滚动条根据需要显示
        scr1 = new JScrollPane(pan, JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
                JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED);
        cont.add(scr1) ;                               // 加入到容器中
        frame.setSize(230, 120) ;                      // 设置窗体大小
        frame.setLocation(300,200) ;                   // 设置窗体的显示位置
        frame.setVisible(true) ;                       // 让组件显示
    }
}
```

程序运行结果：



以上图片因为大小已经超过了容器本身，所以只显示了部分内容。水平滚动条是根据需要来显示的，而垂直滚动条将始终显示，用户可以自由改变窗体大小观察滚动条的显示情况。

18.7.5 JDesktopPane 与 JInternalFrame

在 Swing 中也可以完成内部窗体的显示，即在一个窗体中可以出现多个子窗体，每一个子窗体都无法超出父窗体的区域。

JDesktopPane 规定了一个父窗体的基本形式，而 JInternalFrame 规定了各个子窗体，JInternalFrame 需要加入到 JDesktopPane 中。这两个类的常用方法分别如表 18-16 和表 18-17 所示。

表 18-16　JDesktopPane 类的常用方法

序号	方　　法	类型	描　　述
1	public JDesktopPane()	构造	创建一个 JDesktopPane()对象
2	public void setSclectedFrame(JInternalFrame f)	普通	设置此 JDesktopPane 中当前活动的 JInternalFrame

表 18-17　JInternalFrame 类的常用方法

序号	方　　法	类型	描　　述
1	public JInternalFrame(String title)	构造	创建不可调整大小、不可关闭、不可最大化、不可图标化、具有指定标题的 JInternalFrame
2	public JInternalFrame(String title,boolean resizable)	构造	创建不可关闭、不可最大化、不可图标化，以及具有指定标题和可调整大小的 JInternalFrame
3	public JInternalFrame(String title,boolean resizable, boolean closable,boolean maximizable,boolean iconifiable)	构造	创建可调整、可关闭、可最大化、可图标化的 JInternalFrame

范例：设置内部窗体

```
package org.lxh.demo18.jinterdemo;
import java.awt.BorderLayout;
import java.awt.Container;
import java.io.File;
import javax.swing.Icon;
import javax.swing.ImageIcon;
import javax.swing.JDesktopPane;
import javax.swing.JFrame;
import javax.swing.JInternalFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
public class JInternalFrameDemo01 {
    public static void main(String args[]) {
        JFrame frame = new JFrame("Welcome To MLDN");// 实例化窗体对象
        frame.setLayout(new BorderLayout());         // 设置布局管理器
        Container cont = frame.getContentPane() ;    // 得到窗体容器
        JDesktopPane desk = new JDesktopPane() ;     // 实例化JdesktopPane
                                                        容器
        cont.add(desk, BorderLayout.CENTER);         // 将Desktop加入到窗体
        cont.add(new JLabel("内部窗体"), BorderLayout.SOUTH);
        JInternalFrame jif = null ;                  // 声明一个内部窗体对象
```

```
            String picPath = "f:" + File.separator + "mldn.gif" ;
            Icon icon = new ImageIcon(picPath) ;              // 实例化Icon对象
            JPanel pan = null ;
            for(int i=0;i<3;i++){
                // 实例化一个指定标题、可以改变大小、可关闭、可最大化、可图标化的内部窗体
                jif = new JInternalFrame("MLDN-" + i, true, true, true, true);
                pan = new JPanel() ;                          // 定义一个面板
                pan.add(new JLabel(icon)) ;                   // 加入一个标签
                jif.setLocation(35 - i * 10, 35 - i * 10); // 设置显示位置
                jif.add(pan) ;                                // 将面板加入到内部
                                                                 窗体
                jif.pack() ;
                jif.setVisible(true) ;                        // 设置内部窗体可见
                desk.add(jif) ;                               // 将内部窗体加入到
                                                                 Desktop中
            }
            frame.setSize(300, 270) ;                         // 设置窗体大小
            frame.setLocation(300,200) ;                      // 设置窗体的显示位置
            frame.setVisible(true) ;                          // 让组件显示
        }
    }
```

程序运行结果：





以上代码通过循环产生了 3 个内部窗体，并依次设置好其在窗体中的显示位置，无论现在怎么拖拽都无法将这些内部窗体移动到窗体之外。

18.8 不弹起的按钮组件：JToggleButton

JButton 提供了一个按钮的基本实现，但是此按钮每次单击之后都会恢复自动弹起。如果现在希望单击一个按钮后不再自动弹起，而是在第二次单击此按钮时弹起，则就必须使用 JToggleButton 类。此类的常用方法如表 18-18 所示。

表 18-18　JToggleButton 类的常用方法

序号	方　　法	类型	描　　述
1	public JToggleButton()	构造	创建一个最基本的按钮
2	public JToggleButton(Icon icon)	构造	为按钮设置图片
3	public JToggleButton(String text)	构造	为按钮设置文字
4	public JToggleButton(String text,Icon icon)	构造	为按钮设置文字和图片
5	public JToggleButton(String text,boolean selected)	构造	为按钮设置文字和选择状态

范例：使用 JToggleButton 建立按钮

```
package org.lxh.demo18.jtogglebuttondemo;
import java.awt.GridLayout;
import javax.swing.JFrame;
import javax.swing.JToggleButton;
public class JToggleButtonDemo01 {
    public static void main(String args[]) {
        JFrame frame = new JFrame("Welcome To MLDN") ; // 实例化窗体对象
        JToggleButton but1 = new JToggleButton("已选中", true);
                                                        // 定义按钮对象
        JToggleButton but2 = new JToggleButton("未选中"); // 定义按钮对象
        JToggleButton but3 = new JToggleButton("按我") ;// 定义按钮对象
        frame.setLayout(new GridLayout(3, 1));          // 设置排版样式
        frame.add(but1) ;                               // 向容器中加入组件
        frame.add(but2) ;                               // 向容器中加入组件
        frame.add(but3) ;                               // 向容器中加入组件
        frame.setSize(200, 100) ;                       // 设置窗体大小
        frame.setLocation(300,200) ;                    // 设置窗体的显示位置
        frame.setVisible(true) ;                        // 让组件显示
    }
}
```

程序运行结果：



程序中的 3 个按钮，but1 因为已经设置为已选中状态，所以程序一运行就是按下的状态，之后又设置了两个没有被默认选中的按钮，当每次单击之后按钮不会自动弹起。

18.9　文本组件：JTextComponent

各个软件系统中都存在文本框，以方便用户输入数据，在 Swing 中也提供了同样的文

本框组件，但是文本输入组件在 Swing 中也分为以下几类。

- 单行文本框：JTextField。
- 密码文本框：JPasswordField。
- 多行文本框：JTextArea。

在开发中 JTextComponent 的常用方法如表 18-19 所示。

表 18-19 JTextComponent 的常用方法

序号	方 法	类型	描 述
1	public String getText()	普通	返回文本框的所有内容
2	public String getSelectedText()	普通	返回文本框中选定的内容
3	public int getSelectionStart()	普通	返回文本框选定内容的开始点
4	public int getSelectionEnd()	普通	返回文本框选定内容的结束点
5	public void selectAll()	普通	选择此文本框的所有内容
6	public void setText(String t)	普通	设置此文本框的内容
7	public void select(int selectionStart,int selectionEnd)	普通	将指定开始点和结束点之间的内容选定
8	public void setEditable(boolean b)	普通	设置此文本框是否可编辑

18.9.1 单行文本输入组件：JTextField

如果要实现一个单行的输入文本，可以使用 JTextField 组件。此类除了可以使用 JTextComponent 类的方法外，还可以使用如表 18-20 所示的几个方法。

表 18-20 JTextField 的常用方法

序号	方 法	类型	描 述
1	public JTextField()	构造	构造一个默认的文本框
2	public JTextField(String text)	构造	构造一个指定文本内容的文本框
3	public void setColumns(int columns)	普通	设置显示长度

范例：使用 JTextField 定义普通文本框

```
package org.lxh.demo18.jtextdemo;
import java.awt.GridLayout;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;
public class JTextDemo01 {
    public static void main(String args[]) {
        JFrame frame = new JFrame("Welcome To MLDN");// 实例化窗体对象
        JTextField name = new JTextField(30) ;       // 定义文本框，并指定长度
        JTextField noed = new JTextField("MLDN", 10);// 定义文本框，指定内容和
                                                     长度
```

```
        JLabel nameLab = new JLabel("输入用户姓名：") ;   // 定义标签
        JLabel noedLab = new JLabel("不可编辑文本：") ;   // 定义标签
        noed.setEnabled(false) ;                          // 此文本框不可编辑
        name.setColumns(30) ;                             // 设置长度，但此时不起作用
        noed.setColumns(10) ;                             // 设置长度，但此时不起作用
        frame.setLayout(new GridLayout(2, 2));            // 设置容器的布局管理器
        frame.add(nameLab) ;                              // 向容器中增加组件
        frame.add(name) ;                                 // 向容器中增加组件
        frame.add(noedLab) ;                              // 向容器中增加组件
        frame.add(noed) ;                                 // 向容器中增加组件
        frame.setSize(300, 100) ;                         // 设置窗体大小
        frame.setLocation(300,200) ;                      // 设置窗体的显示位置
        frame.setVisible(true) ;                          // 让组件显示
    }
}
```

程序运行结果：



以上程序使用了 GridLayout 的排版格式，第一个文本框是可编辑的，第二个文本框是不可编辑的。细心的读者可能已经发现，在以上的程序中，虽然使用 setColumns()方法设置显示的行数，但是在显示上并没有任何的改变，主要原因是由于 GridLayout 在使用时会忽略这些设置值，让每一个格子都具有相同的大小。如果要解决这样的问题，可以取消布局管理器，而使用绝对定位的方式进行设置。

范例：使用绝对定位显示

```
package org.lxh.demo18.jtextdemo;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;
public class JTextDemo02 {
    public static void main(String args[]) {
        JFrame frame = new JFrame("Welcome To MLDN");// 实例化窗体对象
        JTextField name = new JTextField(30) ;       // 定义文本框，并指定长度
        JTextField noed = new JTextField("MLDN", 10);// 定义文本框，指定内容和
                                                     长度
        JLabel nameLab = new JLabel("输入用户姓名：") ;// 定义标签
        JLabel noedLab = new JLabel("不可编辑文本：") ;// 定义标签
        noed.setEnabled(false) ;                     // 此文本框不可编辑
        nameLab.setBounds(10,10,100,20) ;            // 设置组件位置及大小
        noedLab.setBounds(10,40,100,20) ;            // 设置组件位置及大小
```

```
        name.setBounds(110, 10, 80, 20) ;                    // 设置组件位置及大小
        noed.setBounds(110, 40, 50, 20) ;                    // 设置组件位置及大小
        frame.setLayout(null);                               // 使用绝对定位
        frame.add(nameLab) ;                                 // 向容器中增加组件
        frame.add(name) ;                                    // 向容器中增加组件
        frame.add(noedLab) ;                                 // 向容器中增加组件
        frame.add(noed) ;                                    // 向容器中增加组件
        frame.setSize(300, 100) ;                            // 设置窗体大小
        frame.setLocation(300,200) ;                         // 设置窗体的显示位置
        frame.setVisible(true) ;                             // 让组件显示
    }
}
```

程序运行结果：



以上使用了绝对定位的方法，对组件进行了显示设置，这样组件的大小、位置就可以由用户根据需要自由定义。

18.9.2 密文输入组件：JPasswordField

JTextField 是使用明文方式进行数据显示的，如果现在需要将回显的内容设置成其他字符，则可以使用 JPasswordField 类。此类的常用方法如表 18-21 所示。

表 18-21 JPasswordField 类的常用方法

序号	方　法	类型	描　述
1	public JPasswordField()	构造	构造默认的 JPasswordField 对象
2	public JPasswordField(String text)	构造	构造指定内容的 JPasswordField 对象
3	public char getEchoChar()	构造	设置回显的字符，默认为“*”
4	public char getEchoChar()	构造	得到回显的字符
5	public char[] getPassword()	构造	得到此文本框的所有内容

范例：设置密文显示框

```
package org.lxh.demo18.jtextdemo;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPasswordField;
public class JPasswordDemo01 {
    public static void main(String args[]) {
        JFrame frame = new JFrame("Welcome To MLDN") ; // 实例化窗体对象
        JPasswordField jpf1 = new JPasswordField() ;   // 定义密文框
```

```
        JPasswordField jpf2 = new JPasswordField();     // 定义密文框
        jpf2.setEchoChar('#') ;                         // 设置回显字符为“#”
        JLabel lab1 = new JLabel("默认的回显：") ;       // 定义标签
        JLabel lab2 = new JLabel("回显设置"#"：") ;      // 定义标签
        lab1.setBounds(10,10,100,20) ;                  // 设置组件位置及大小
        lab2.setBounds(10,40,100,20) ;                  // 设置组件位置及大小
        jpf1.setBounds(110, 10, 80, 20) ;               // 设置组件位置及大小
        jpf2.setBounds(110, 40, 50, 20) ;               // 设置组件位置及大小
        frame.setLayout(null);                          // 使用绝对定位
        frame.add(lab1) ;                               // 向容器中增加组件
        frame.add(jpf1) ;                               // 向容器中增加组件
        frame.add(lab2) ;                               // 向容器中增加组件
        frame.add(jpf2) ;                               // 向容器中增加组件
        frame.setSize(300, 100) ;                       // 设置窗体大小
        frame.setLocation(300,200) ;                    // 设置窗体的显示位置
        frame.setVisible(true) ;                        // 让组件显示
    }
}
```

程序运行结果：



第一个密文框使用的是默认的回显字符，而第二个密文框是将“#”设置成了回显字符。

18.9.3　多行文本输入组件：JTextArea

如果要输入多行文本，则可以使用 JTextArea 实现多行文本的输入。此类扩展了 JTextComponent 类，常用方法如表 18-22 所示。

表 18-22　JTextArea 类的常用方法

序号	方　　法	类型	描　　述
1	public JTextArea()	构造	构造文本域，行数和列数为 0
2	public JTextArea(int rows,int columns)	构造	构造文本域，指定文本域的行数和列数
3	public JTextArea(String text,int rows,int columns)	构造	指定构造文本域的内容、行数和列数
4	public void append(String str)	普通	在文本域中追加内容
5	public void replaceRange(String str,int start,int end)	普通	替换文本域中指定范围的内容
6	public void insert(String str,int pos)	普通	在指定位置插入文本
7	public void setLineWrap(boolean wrap)	普通	设置换行策略

范例：使用 JtextArea

```
package org.lxh.demo18.jtextdemo;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextArea;
public class JTextAreaDemo01 {
    public static void main(String args[]) {
        JFrame frame = new JFrame("Welcome To MLDN") ; // 实例化窗体对象
        JTextArea jta = new JTextArea(3,10) ;          // 构造一个文本域
        JLabel lab = new JLabel("多行文本域：") ;       // 定义标签
        lab.setBounds(10,10,120,20) ;                  // 设置组件位置及大小
        jta.setBounds(130, 10, 150, 100) ;             // 设置组件位置及大小
        frame.setLayout(null);                         // 使用绝对定位
        frame.add(lab) ;                               // 向容器中增加组件
        frame.add(jta) ;                               // 向容器中增加组件
        frame.setSize(300, 150) ;                      // 设置窗体大小
        frame.setLocation(300,200) ;                   // 设置窗体的显示位置
        frame.setVisible(true) ;                       // 让组件显示
    }
}
```

程序运行结果：



以上只是简单地构建了一个文本域。但是如果一个文本域设置得过大，则肯定会使用滚动条显示，此时就需要将文本域设置在带滚动条的面板中，使用 JScrollPane 即可。

范例：为文本域加入滚动条

```
package org.lxh.demo18.jtextdemo;
import java.awt.GridLayout;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
public class JTextAreaDemo02 {
    public static void main(String args[]) {
        JFrame frame = new JFrame("Welcome To MLDN");// 实例化窗体对象
        JTextArea jta = new JTextArea(3,20) ;        // 构造一个文本域
        jta.setLineWrap(true) ;                      // 如果内容过长，自动换行
        // 在文本域上加入滚动条，水平和垂直滚动条始终出现
```

```
        JScrollPane scr = new JScrollPane(jta,
                JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
                JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
        JLabel lab = new JLabel("多行文本域：") ;                    // 定义标签
        frame.setLayout(new GridLayout(2,1));                       // 设置布局管理器
        frame.add(lab) ;                                            // 向容器中增加组件
        frame.add(scr) ;                                            // 向容器中增加组件
        frame.setSize(300, 150) ;                                   // 设置窗体大小
        frame.setLocation(300,200) ;                                // 设置窗体的显示位置
        frame.setVisible(true) ;                                    // 让组件显示
    }
}
```

程序运行结果：



从程序的运行结果中可以清楚地发现，如果一个文本域中的内容过多，则可以自动进行换行显示。

18.10　事 件 处 理

一个图形界面制作完成了，但是在程序开发中这只是完成了起步的工作。因为要想让每一个组件都发挥其自己的作用，就必须对所有的组件进行事件处理。

18.10.1　事件和监听器

要想清楚事件处理，则首先应该知道事件的定义是什么，事件就是表示一个对象的发生状态变化。例如，每当一个按钮按下时，实际上按钮的状态就发生了改变，那么此时就会产生一个事件，而如果要想处理此事件，就需要事件的监听者不断地监听事件的变化，并根据这些事件进行相应的处理。

在 Swing 编程中，依然使用了最早 AWT 的事件处理方式，所有的事件类（基本上任意的一个组件都有对应的事件）都是 EventObject 类的子类，如图 18-16 所示。

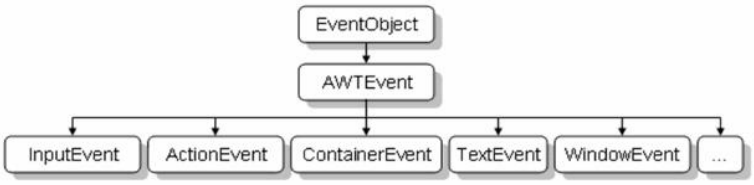


图 18-16　事件类的继承关系

EventObject 类的定义格式如下所示：

```
public class EventObject extends Object implements Serializable{
    public EventObject(Object source){ // 构造一个发生事件的对象
    }
    public Object getSource(){          // 返回一个事件对象
    }
    public String toString(){           // 得到信息
    }
}
```

在这个类中定义了 3 个方法，其中可以通过 getSource()取得发生此事件的源对象。

了解了事件源后，下面再来看一下事件的监听器，如果没有能够接收和处理事件的对象，图形界面程序所生成的一切事件都是无用的，这些事件接收对象被称为事件监听器。所有的事件监听器都是以监听接口的形式出现的，处理时只需要实现此接口即可。

整个事件的处理流程如图 18-17 所示。

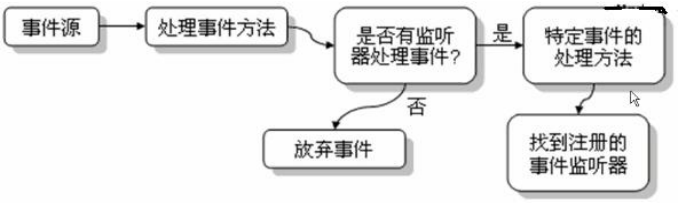


图 18-17　Java 事件处理流程

了解了这些概念之后，下面通过几个事件来进一步说明事件的处理流程。大部分图形界面的事件处理类或接口都保存在 java.awt.event 包中。

18.10.2　窗体事件

WindowListener 是专门处理窗体的事件监听接口，一个窗体的所有变化，如窗口的打开、关闭等都可以使用这个接口进行监听。此接口定义的方法如表 18-23 所示。

表 18-23　WindowListener 接口的方法

序号	方　法	类型	描　述
1	void windowActivated(WindowEvent e)	普通	将窗口变为活动窗口时触发
2	void windowDeactivated(WindowEvent e)	普通	将窗口变为不活动窗口时触发
3	void windowClosed(WindowEvent e)	普通	当窗口被关闭时触发
4	void windowClosing(WindowEvent e)	普通	当窗口正在关闭时触发
5	void windowIconified(WindowEvent e)	普通	窗口最小化时触发
6	void windowDeiconified(WindowEvent e)	普通	窗口从最小化恢复到正常状态时触发
7	void windowOpened(WindowEvent e)	普通	窗口打开时触发

范例：实现 WindowListener

```
package org.lxh.demo18.windoweventdemo;
import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;
public class MyWindowEventHandle
                implements WindowListener {          // 实现窗口监听
    public void windowActivated(WindowEvent arg0) {// 设置为活动窗口时触发
        System.out.println("windowActivated --> 窗口被选中。");
    }
    public void windowClosed(WindowEvent arg0) {    // 窗口被关闭时触发
        System.out.println("windowClosed --> 窗口被关闭。");
    }
    public void windowClosing(WindowEvent arg0) {  // 窗口关闭时触发，按下关闭
                                                      按钮
        System.out.println("windowClosing --> 窗口关闭。");
        System.exit(1) ;                             // 系统退出
    }
    public void windowDeactivated(WindowEvent arg0){// 设置为非活动窗口时触发
        System.out.println("windowDeactivated --> 取消窗口选中。");
    }
    public void windowDeiconified(WindowEvent arg0){// 窗口从最小化还原时触发
        System.out.println("windowDeiconified --> 窗口从最小化恢复。") ;
    }
    public void windowIconified(WindowEvent arg0){ // 窗口最小化时触发
        System.out.println("windowIconified --> 窗口最小化。") ;
    }
    public void windowOpened(WindowEvent arg0) {    // 窗口打开时触发
        System.out.println("windowOpened --> 窗口被打开。") ;
    }
}
```

单单只有一个监听器是不够的，还需要在组件使用时注册监听，这样才可以处理，直接使用窗体的 addWindowListener（监听对象）方法即可注册事件监听。

范例：在窗体上注册事件监听器

```
package org.lxh.demo18.windoweventdemo;
import java.awt.Color;
import javax.swing.JFrame;
public class MyWindowEventJFrame01 {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Welcome To MLDN") ; // 实例化窗体对象
        // 将此窗体加入到一个窗口事件监听器中，这样监听器就可以根据事件进行处理
        frame.addWindowListener(new MyWindowEventHandle());
```

```
        frame.setSize(300, 160) ;                          // 设置组件大小
        frame.setBackground(Color.WHITE) ;                 // 设置窗体的背景颜色
        frame.setLocation(300,200) ;                       // 设置窗体的显示位置
        frame.setVisible(true) ;                           // 让组件显示
    }
}
```

程序运行结果：

```
windowActivated --> 窗口被选中。
windowOpened --> 窗口被打开。
windowIconified --> 窗口最小化。
windowDeactivated --> 取消窗口选中。
windowDeiconified --> 窗口从最小化恢复。
windowActivated --> 窗口被选中。
windowClosing --> 窗口关闭。
```

程序运行之后会显示一个窗体，此时对窗体进行相应的状态改变，则在后台会打印以上的一些信息。在关闭监听中编写了 System.exit(1)语句，这样关闭按钮就真正起作用，可以让程序正常结束退出。

18.10.3 监听适配器

大致了解了事件处理的基本流程之后，读者可能会有这样的一个问题：“如果现在只需要对关闭窗口的事件进行监听，其他的操作根本就不关心，那么还有必要覆写那么多的方法吗？能不能只根据个人的需要来进行覆写呢？”

如果要解决这样的问题，直接使用监听接口肯定不合适，因为一个类如果实现接口，则必须覆写其中的全部抽象方法。而在之前讲解面向对象时曾经为读者讲解过一种称为适配器的设计模式，在实现类和接口之间增加一个过渡的抽象类，子类继承抽象类就可以根据自己的需要进行方法的覆写，所以在整个事件处理中提供了很多的 Adapter（适配器）类，方便用户进行事件处理的实现。以 WindowAdapter 为例，用户只要继承了此类，就可以根据自己的需要覆写方法，如果现在只需要关心窗口关闭方法，则只在子类中覆写 windowClosing()方法即可。

范例：通过 WindowAdapter 实现监听

```
package org.lxh.demo18.windoweventdemo;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
public class MyWindowAdapterEventHandle
                    extends WindowAdapter {          // 实现适配器类
    // 此时可以根据自己的需要覆写方法，此类只覆写windowClosing()方法
    public void windowClosing(WindowEvent arg0){     // 窗口关闭时触发，按下关闭
                                                        按钮
        System.out.println("windowClosing --> 窗口关闭。");
```

```
            System.exit(1) ;                                    // 系统退出
        }
    }
```

而在窗体操作的代码中，直接使用以上的监听器类即可，代码如下所示：

```
        JFrame frame = new JFrame("Welcome To MLDN") ; // 实例化窗体对象
        // 此时直接使用WindowAdapter的子类完成监听的处理
        frame.addWindowListener(new MyWindowAdapterEventHandle());
```

这样，当单击【关闭】按钮时会自动进行监听处理，让系统退出。但是这样一来也会有一个新的问题产生，如果此监听处理只需要操作一次，那还有必要将其设置成为一个单独的类吗？很明显，将其设置成一个单独的类有些多余，那么此时就可以利用之前的匿名内部类来完成监听操作。

范例：使用匿名内部类

```
package org.lxh.demo18.windoweventdemo;
import java.awt.Color;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JFrame;
public class MyWindowEventJFrame03 {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Welcome To MLDN") ; // 实例化窗体对象
        // 此时直接使用WindowAdapter的子类完成监听的处理
        frame.addWindowListener(new WindowAdapter()     {
            // 覆写窗口关闭的方法
            public void windowClosing(WindowEvent arg0) {
                System.exit(1) ;                        // 系统退出
            }
        });
        frame.setSize(300, 160) ;                       // 设置组件大小
        frame.setBackground(Color.WHITE) ;              // 设置窗体的背景颜色
        frame.setLocation(300,200) ;                    // 设置窗体的显示位置
        frame.setVisible(true) ;                        // 让组件显示
    }
}
```

从以上代码可以发现，如果现在使用适配器操作类，则直接编写匿名内部类就可以减少监听类的定义，这在开发中也是较为常见的一种做法。

18.10.4　动作事件及监听处理

要想让一个按钮变得有意义，就必须使用事件处理。在 Swing 的事件处理中，可以使用 ActionListener 接口处理按钮的动作事件。ActionListener 接口只定义了一个方法，如

表 18-24 所示。

表 18-24 ActionListener 接口方法

序号	方 法	类型	描 述
1	void actionPerformed(ActionEvent e)	普通	发生操作时调用

下面是使用以上监听接口监听按钮的单击事件。

范例：使用 ActionListener 监听

```
package org.lxh.demo18.actioneventdemo;
import java.awt.Font;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;
class ActionHandle{
    private JFrame frame = new JFrame("Welcome To MLDN") ; // 声明一个窗体
                                                            对象
    private JButton but = new JButton("显示") ;     // 声明一个按钮
    private JLabel lab = new JLabel() ;             // 声明一个标签
    private JTextField text = new JTextField(10) ; // 定义一个文本域
    private JPanel pan = new JPanel() ;             // 定义一个面板
    public ActionHandle(){
        Font fnt = new Font("Serief", Font.ITALIC + Font.BOLD, 28);
        lab.setFont(fnt) ;                          // 设置标签的显示字体
        lab.setText("等待用户输入信息！") ;          // 设置默认显示文字
        but.addActionListener(new ActionListener(){// 采用匿名内部类
            public void actionPerformed(ActionEvent arg0) {
                if (arg0.getSource() == but) {      // 判断触发源是否是按钮
                    lab.setText(text.getText()) ;   // 将文本文字设置到标签
                }
            }
        }) ;                                        // 此处加入动作监听
        frame.addWindowListener(new WindowAdapter(){// 加入窗口监听
            // 覆写窗口关闭的方法
            public void windowClosing(WindowEvent arg0) {
                System.exit(1) ;                    // 系统退出
```

```
            }
        });
        frame.setLayout(new GridLayout(2, 1));          // 定义窗体布局，2行1列
        pan.setLayout(new GridLayout(1,2)) ;            // 定义面板布局，1行2列
        pan.add(text) ;                                 // 向面板中增加文本域
        pan.add(but) ;                                  // 向面板中增加按钮
        frame.add(pan) ;                                // 将面板加入到窗体
        frame.add(lab) ;                                // 将标签加入到窗体
        frame.pack() ;                                  // 根据组件自动调整大小
        frame.setVisible(true) ;                        // 显示窗体
    }
}
public class MyActionEventDemo01 {
    public static void main(String[] args) {
        new ActionHandle() ;
    }
}
```

程序运行结果：



程序运行后会出现等待用户输入信息的提示框，用户输入信息选择显示后，会在标签中显示用户输入的内容。

以上程序的事件操作也使用了匿名内部类的方式完成，因为动作事件有可能有多种事件源调用，所以为了保险起见，在代码操作之前增加了以下验证：

```
if (arg0.getSource() == but) {                  // 判断触发源是否是按钮
}
```

直接使用了地址比较方式，如果操作的事件源是由按钮触发的，则执行语句。当然，此时代码是在一个类中完成的，如果现在换成了两个类，则可以通过 instanceof 关键字来判断操作的数据源类型，如下所示：

```
if (arg0.getSource() instanceof JButton) { // 判断触发源是否是按钮
}
```

了解了动作事件之后，下面实际上就可以使用此事件完成一个简单的用户登录操作。如果用户输入的用户名为 lixinghua，密码为 mldn，则认为是合法用户，提示登录成功的信息；反之，则提示登录失败的信息，操作代码如下所示。

范例：用户登录操作

➘ 编写登录验证类

```
class LoginCheck {                                  // 编写登录验证类
    private String name ;                           // 用户名
```

```
    private String password ;                                    // 密码
    public LoginCheck(String name,String password){              // 构造方法赋值
        this.name = name ;                                       // 为name赋值
        this.password = password ;                               // 为password赋值
    }
    public boolean validate(){                                   // 登录验证方法
        if ("lixinghua".equals(name) && "mldn".equals(password)) {
            return true ;                                        // 登录成功
        } else {
            return false ;                                       // 登录失败
        }
    }
}
```

以上的类只是根据输入进来的用户名和密码进行验证，在本类中不涉及任何的显示操作，只是完成逻辑功能。使用时通过构造方法对 name 和 password 属性的内容进行赋值，之后调用 validate()方法完成验证的操作。

提问：为什么字符串比较时，把字符串放在前面？

以上程序在编写验证用户名和密码时为什么使用如下的语句形式：

```
if ("mldn".equals(name) && "lxh".equals(password))
```

为什么通过字符串调用 equals 方法，直接使用 name 或 password 调用不行吗？

回答：为避免空指向异常。

首先 String 是一个引用数据类型，如果引用数据类型没有开辟空间，则内容为 null，null 调用方法则肯定会出现空指向异常。而在之前讲解 String 时曾经讲解过，一个字符串就是 String 的匿名对象，那么匿名对象就永远不可能为空，这样即便 name 或 password 没有正确的赋值，则也可以避免空指向异常。此种做法是在开发中较为常见的一种做法，读者要习惯于这种写法。

编写图形界面：本程序为了简便起见，将使用绝对定位的方式进行版面的布局

```
package org.lxh.demo18.actioneventdemo;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPasswordField;
import javax.swing.JTextField;
```

```
class ActionHandle{
    private JFrame frame = new JFrame("Welcome To MLDN") ; // 声明一个窗体
                                                           对象
    private JButton submit = new JButton("登录") ;       // 声明一个按钮
    private JButton reset = new JButton("重置") ;        // 声明一个按钮
    private JLabel nameLab = new JLabel("用户名: ") ;    // 声明一个标签
    private JLabel passLab = new JLabel("密　　码: ") ; // 声明一个标签
    private JLabel infoLab = new JLabel("用户登录系统") ;// 声明一个标签
    private JTextField nameText = new JTextField() ;     // 定义一个文本域
    private JPasswordField passText = new JPasswordField() ;// 定义一个文本域
    public ActionHandle(){
        Font fnt = new Font("Serief", Font.BOLD, 12);  // 定义显示字体
        infoLab.setFont(fnt) ;                         // 设置标签的显示字体
        submit.addActionListener(new ActionListener(){ // 采用匿名内部类
            public void actionPerformed(ActionEvent arg0) {
                if (arg0.getSource() == submit) {      // 判断触发源是否是提
                                                          交按钮
                    String tname = nameText.getText() ;// 得到输入的用户名
                    // 得到输入的密码，此时通过getPassword()方法返回的是字符数组
                    String tpass = new String(passText.getPassword());
                    // 实例化LoginCheck对象，传入输入的用户名和密码
                    LoginCheck log = new LoginCheck(tname,tpass) ;
                    if (log.validate()) {              // 对用户名和密码进行
                                                          验证
                        infoLab.setText("登录成功，欢迎光临！");
                    } else {
                        infoLab.setText("登录失败，错误的用户名或密码！");
                    }
                }
                if (arg0.getSource() == reset) {       // 判断触发源是否是重
                                                          置按钮
                    nameText.setText("") ;             // 清空文本框内容
                    passText.setText("") ;             // 清空密码框内容
                    infoLab.setText("用户登录系统！");  // 恢复标签显示
                }
            }
        }) ;                                           // 此处加入动作监听
        frame.addWindowListener(new WindowAdapter(){   // 加入窗口监听
            // 覆写窗口关闭的方法
            public void windowClosing(WindowEvent arg0) {
                System.exit(1) ;                       // 系统退出
            }
```

```
        });
        frame.setLayout(null);                          // 使用绝对定位
        nameLab.setBounds(5, 5, 60, 20) ;               // 设置标签的位置及大小
        passLab.setBounds(5, 30, 60, 20) ;              // 设置标签的位置及大小
        infoLab.setBounds(5, 65, 220, 30) ;             // 设置标签的位置及大小
        nameText.setBounds(65, 5, 100, 20) ;            // 设置文本域的位置及大小
        passText.setBounds(65, 30, 100, 20) ;           // 设置密码域的位置及大小
        submit.setBounds(165, 5, 60, 20);               // 设置按钮的位置及大小
        reset.setBounds(165, 30, 60, 20);               // 设置按钮的位置及大小
        frame.add(nameLab) ;                            // 向窗体加入标签
        frame.add(passLab) ;                            // 向窗体加入标签
        frame.add(infoLab) ;                            // 向窗体加入标签
        frame.add(nameText) ;                           // 向窗体加入文本框
        frame.add(passText) ;                           // 向窗体加入密码框
        frame.add(submit) ;                             // 向窗体加入按钮
        frame.add(reset) ;                              // 向窗体加入按钮
        frame.setSize(280, 130);                        // 设置窗体大小
        frame.setVisible(true) ;                        // 显示窗体
    }
}
public class MyActionEventDemo03 {
    public static void main(String[] args) {
        new ActionHandle() ;
    }
}
```

程序运行结果：



以上程序在窗体上使用了一个文本框和一个密码框接收输入的用户名和密码，当单击按钮时会触发 ActionEvent 事件，并判断发生此事件的是哪个按钮。如果是【登录】按钮触发的则从文本框中取出内容并通过 LoginCheck 类进行用户名和密码的验证；如果是【重置】按钮则将用户名和密码框的内容清除。

提示：注意程序的分层。

本书从面向对象开始就一直强调对程序的结构进行合理的划分，在本程序中有专门负责业务处理的 LoginCheck 类，也有完成界面显示的 MyActionEventDemo03 类，很好地达到了显示和业务的分离。当然，更加合理的做法是中间要加入一个控制端，即形成“视图-控制-模型”的三层结构。

18.10.5　键盘事件及监听处理

在 Swing 的事件处理中也可以对键盘的操作进行监听，直接使用 KeyListener 接口即可。此接口定义了如表 18-25 所示的方法。

表 18-25　KeyListener 接口方法

序号	方　法	类型	描　述
1	void keyTyped(KeyEvent e)	普通	输入某个键时调用
2	void keyPressed(KeyEvent e)	普通	键盘按下时调用
3	void keyReleased(KeyEvent e)	普通	键盘松开时调用

如果要取得键盘输入的内容，则可以通过 KeyEvent 取得。此类的常用方法如表 18-26 所示。

表 18-26　KeyEvent 事件的常用方法

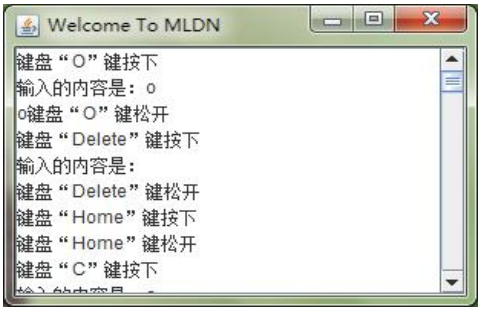
序号	方　法	类型	描　述
1	public char getKeyChar()	普通	返回输入的字符，只针对于 keyTyped 有意义
2	public int getKeyCode()	普通	返回输入字符的键码
3	public static String getKeyText(int keyCode)	普通	返回此键的信息，如“HOME”、“F1”或“A”等

范例：实现键盘监听

```
package org.lxh.demo18.keyeventdemo;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
// 此类定义时直接继承了JFrame类、实现KeyListener接口
class MyKeyHandle extends JFrame implements KeyListener {
    private JTextArea text = new JTextArea() ;
    public MyKeyHandle() {
        super.setTitle("Welcome To MLDN");
        JScrollPane scr = new JScrollPane(text);          // 加入滚动条
        scr.setBounds(5,5, 300, 200) ;
        super.add(scr) ;                                  // 向窗体加入组件
        text.addKeyListener(this) ;                       // 加入key监听
        super.setSize(310,210) ;                          // 设置窗体大小
        super.setVisible(true) ;                          // 显示窗体
```

```
            super.addWindowListener(new WindowAdapter(){          // 加入事件监听
                // 覆写窗口关闭方法
                public void windowClosing(WindowEvent arg0) {
                    System.exit(1) ;                              // 系统退出
                }
            }) ;
        }
        public void keyPressed(KeyEvent e) {                      // 键盘按下
            text.append("键盘"" + KeyEvent.getKeyText(e.getKeyCode()) + ""键按
            下\n");
        }
        public void keyReleased(KeyEvent e) {                     // 键盘松开
            text.append("键盘"" + KeyEvent.getKeyText(e.getKeyCode()) + ""键松
            开\n");
        }
        public void keyTyped(KeyEvent e) {                        // 输入内容
            text.append("输入的内容是: " + e.getKeyChar() + "\n");
        }
    }
    public class MyKeyEventDemo01 {
        public static void main(String[] args) {
            new MyKeyHandle() ;
        }
    }
```

程序运行结果：



以上程序中针对每个键盘的操作都会进行监听，而且在取得键盘信息时最好使用 KeyEvent 类提供的静态方法 getKeyText()完成。

提示：this 表示当前对象。

以上程序中，MyKeyHandle 实现了 KeyListener 监听接口，所以此类也就是监听操作类，这样当 JTextArea 增加事件时直接使用了 this 关键字，如下所示：

```
text.addKeyListener(this) ;                                       // 加入 key 监听
```

this 表示当前对象，此时将 this 加入到监听器中，就表示将一个监听处理类加入到监听器中。

在键监听中，也可以使用 KeyAdapter 适配器完成键盘事件的监听，如下所示。

范例：使用 KeyAdapter

```
package org.lxh.demo18.keyeventdemo;
import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
class MyKeyHandle extends JFrame{                          // 此类定义时直接继承了JFrame类
    private JTextArea text = new JTextArea() ;
    public MyKeyHandle() {
        super.setTitle("Welcome To MLDN");
        JScrollPane scr = new JScrollPane(text);           // 加入滚动条
        scr.setBounds(5,5, 300, 200) ;
        super.add(scr) ;                                   // 向窗体加入组件
        text.addKeyListener(new KeyAdapter(){
            public void keyTyped(KeyEvent e) {             // 输入内容
                text.append("输入的内容是: " + e.getKeyChar() + "\n");
            }
        }) ;                                               // 加入key监听
        super.setSize(310,210) ;                           // 设置窗体大小
        super.setVisible(true) ;                           // 显示窗体
        super.addWindowListener(new WindowAdapter(){       // 加入事件监听
            public void windowClosing(WindowEvent arg0) {  // 覆写窗口关闭方法
                System.exit(1) ;                           // 系统退出
            }
        }) ;
    }
}
public class MyKeyEventDemo02 {
    public static void main(String[] args) {
        new MyKeyHandle() ;
    }
}
```

以上代码完成了与之前一样的功能，此时只是针对输入的按键进行了监听。

18.10.6 鼠标事件及监听处理

如果想对一个鼠标的操作进行监听，如鼠标按下、松开等，则可以使用 MouseListener 接口。此接口定义了如表 18-27 所示的方法。

表 18-27 MouseListener 接口的方法

序号	方 法	类型	描 述
1	void mouseClicked(MouseEvent e)	普通	鼠标单击时调用（按下并释放）
2	void mousePressed(MouseEvent e)	普通	鼠标按下时调用
3	void mouseReleased(MouseEvent e)	普通	鼠标松开时调用
4	void mouseEntered(MouseEvent e)	普通	鼠标进入到组件时调用
5	void mouseExited(MouseEvent e)	普通	鼠标离开组件时调用

在每个事件触发后都会产生 MouseEvent 事件，此事件可以得到鼠标的相关操作。MouseEvent 类的常用方法及常量如表 18-28 所示。

表 18-28 MouseEvent 事件的常用方法及常量

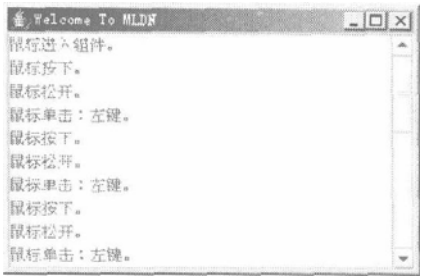
序号	方法及常量	类型	描 述
1	public static final int BUTTON1	常量	表示鼠标左键的常量
2	public static final int BUTTON2	常量	表示鼠标滚轴的常量
3	public static final int BUTTON3	常量	表示鼠标右键的常量
4	public int getButton()	普通	以数字形式返回按下的鼠标键
5	public int getClickCount()	普通	返回鼠标的单击次数
6	public static String get MouseModifiers Text(int modifiers)	普通	以字符串形式返回鼠标按下的键信息
7	public int getX()	普通	返回鼠标操作的 X 坐标
8	public int getY()	普通	返回鼠标操作的 Y 坐标

范例：实现鼠标监听

```
package org.lxh.demo18.mouseeventdemo;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
// 此类定义时直接继承了JFrame类、实现MouseListener接口
class MyMouseHandle extends JFrame implements MouseListener {
    private JTextArea text = new JTextArea() ;                    // 定义文本域
    public MyMouseHandle() {
```

```
        super.setTitle("Welcome To MLDN");                    // 设置标题
        JScrollPane scr = new JScrollPane(text);              // 加入滚动条
        scr.setBounds(5,5, 300, 200) ;                        // 设置组件大小
        super.add(scr) ;                                      // 向窗体加入组件
        text.addMouseListener(this) ;                         // 加入mouse监听
        super.setSize(310,210) ;                              // 设置窗体大小
        super.setVisible(true) ;                              // 显示窗体
        super.addWindowListener(new WindowAdapter(){          // 加入事件监听
            public void windowClosing(WindowEvent arg0){// 覆写窗口关闭方法
                System.exit(1) ;                              // 系统退出
            }
        }) ;
    }
    public void mouseClicked(MouseEvent e) {                  // 鼠标单击事件
        int c = e.getButton() ;                               // 得到按下的鼠标键
        String mouseInfo = null ;                             // 接收信息
        if (c == MouseEvent.BUTTON1) {                        // 判断是鼠标左键按下
            mouseInfo = "左键";
        } else if (c == MouseEvent.BUTTON3) {                 // 判断是鼠标右键按下
            mouseInfo = "右键";
        } else {                                              // 判断是鼠标滚轴按下
            mouseInfo = "滚轴";
        }
        text.append("鼠标单击: " + mouseInfo + "。\n");
    }
    public void mouseEntered(MouseEvent e) {                  // 鼠标进入组件
        text.append("鼠标进入组件。\n") ;                      // 追加文本内容
    }
    public void mouseExited(MouseEvent e) {                   // 鼠标退出组件
        text.append("鼠标离开组件。\n") ;                      // 追加文本内容
    }
    public void mousePressed(MouseEvent e) {                  // 鼠标按下
        text.append("鼠标按下。\n");                          // 追加文本内容
    }
    public void mouseReleased(MouseEvent e) {                 // 鼠标松开
        text.append("鼠标松开。\n");                          // 追加文本内容
    }
}
public class MyMouseEventDemo01 {
    public static void main(String[] args) {
        new MyMouseHandle() ;
    }
}
```

程序运行结果：



此时，只要是进入或离开此组件，都会有鼠标事件触发，还可以通过单击事件取得鼠标哪个键被按下。

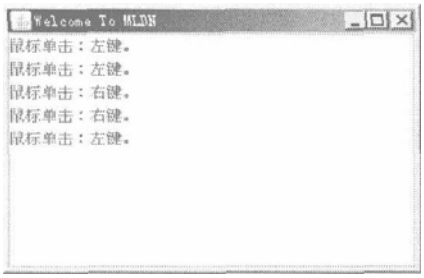
以上实现了 MouseListener 接口，这样在子类中就必须覆写 5 个抽象方法。为了简化起见，也可以直接使用 MouseAdapter 完成对鼠标指定事件的监听。

范例：通过 MouseAdapter 实现对指定鼠标操作监听

```
package org.lxh.demo18.mouseeventdemo;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
class MyMouseHandle extends JFrame {                    // 此类定义时直接继承了
                                                           JFrame类
    private JTextArea text = new JTextArea() ;          // 创建文本区保存内容
    public MyMouseHandle() {
        super.setTitle("Welcome To MLDN");              // 设置窗体标题
        JScrollPane scr = new JScrollPane(text);        // 加入滚动条
        scr.setBounds(5,5, 300, 200) ;                  // 设置组件大小
        super.add(scr) ;                                // 向窗体加入组件
        text.addMouseListener(new MouseAdapter(){
            public void mouseClicked(MouseEvent e){// 鼠标单击事件
                int c = e.getButton() ;                 // 得到按下的鼠标键
                String mouseInfo = null ;
                if (c == MouseEvent.BUTTON1) {          // 判断按下的是鼠标左键
                    mouseInfo = "左键";
                } else if (c == MouseEvent.BUTTON3){// 判断按下的是鼠标右键
                    mouseInfo = "右键";
                } else {                                // 判断按下的是鼠标滚轴
                    mouseInfo = "滚轴";
                }
                text.append("鼠标单击：" + mouseInfo + "。\n");
            }
```

```
            }) ;                                                    // 加入mouse监听
            super.setSize(310,210) ;                                // 设置窗体大小
            super.setVisible(true) ;                                // 显示窗体
            super.addWindowListener(new WindowAdapter(){            // 加入事件监听
                public void windowClosing(WindowEvent arg0){// 覆写窗口关闭方法
                    System.exit(1) ;                                // 系统退出
                }
            }) ;
        }
    }
    public class MyMouseEventDemo02 {
        public static void main(String[] args) {
            new MyMouseHandle() ;
        }
    }
```

程序运行结果：



以上程序中因为使用了 MouseAdapter 类，所以在实现程序时只需要覆写需要的方法即可，在本程序中只覆写了 mouseClicked()方法，所以只能处理鼠标单击事件。

18.10.7　鼠标拖拽事件及监听处理

在一般的图形界面中经常可以看到鼠标拖拽操作的情况，在 Swing 的事件处理中可以使用 MouseMotionListener 接口完成鼠标的拖拽操作。此接口定义了如表 18-29 所示的方法。

表 18-29　MouseMotionListener 接口的方法

序号	方　法	类型	描　述
1	void mouseDragged(MouseEvent e)	普通	在组件上按下并拖动时调用
2	void mouseMoved(MouseEvent e)	普通	鼠标移动到组件时调用

范例：观察鼠标拖拽操作

```
package org.lxh.demo18.mousemotioneventdemo;
import java.awt.event.MouseEvent;
import java.awt.event.MouseMotionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
```

```
import javax.swing.JFrame;
class MyMouseMotionHandle extends JFrame {                    // 此类定义时直接继
                                                                 承了JFrame类
    public MyMouseMotionHandle() {
        super.setTitle("Welcome To MLDN");
        super.addMouseMotionListener(new MouseMotionListener(){
            public void mouseDragged(MouseEvent e) {          // 鼠标拖拽时调用
                System.out.println("鼠标拖拽到: X = " +
                        e.getX() + ", Y = " + e.getY());
            }
            public void mouseMoved(MouseEvent arg0) {         // 鼠标移动到组件时
                                                                 调用
                System.out.println("鼠标移动到窗体。") ;
            }
        }) ;                                                  // 加入mouse监听
        super.setSize(310,210) ;                              // 设置窗体大小
        super.setVisible(true) ;                              // 显示窗体
        super.addWindowListener(new WindowAdapter(){          // 加入事件监听
            public void windowClosing(WindowEvent arg0){      // 覆写窗口关闭方法
                System.exit(1) ;                              // 系统退出
            }
        }) ;
    }
}
public class MyMouseMotionEventDemo01 {
    public static void main(String[] args) {
        new MyMouseMotionHandle() ;
    }
}
```

程序运行结果：

```
鼠标移动到窗体。
鼠标拖拽到: X = 138, Y = 97
鼠标拖拽到: X = 139, Y = 97
鼠标拖拽到: X = 140, Y = 97
鼠标移动到窗体。
...
```

程序运行后，只要鼠标一向窗体移动就会触发 mouseMoved()事件；只要是在窗体上拖拽，则会触发 mouseDragged 事件。

当然在鼠标拖拽操作中也同样存在 MouseMotionAdapter 类，如果要使用此类，直接将加入监听操作的以下代码修改即可。

```
        super.addMouseMotionListener(new MouseMotionAdapter(){
            public void mouseDragged(MouseEvent e) {    // 鼠标拖拽时调用
                System.out.println("鼠标拖拽到: X = " +
                        e.getX() + ", Y = " + e.getY());
            }
        }) ;
```

18.11　单选按钮：JRadioButton

18.11.1　认识 JRadioButton

单选按钮就是在给出的多个显示信息中指定选择一个，在 Swing 中可以使用 JRadioButton 完成一组单选按钮的操作。JRadioButton 类的常用方法如表 18-30 所示。

表 18-30　JRadioButton 类的常用方法

序号	方　法	类型	描　述
1	public JRadioButton(Icon icon)	构造	建立一个单选按钮，并指定图片
2	public JRadioButton(Icon icon, boolean selected)	构造	建立一个单选按钮，并指定图片和其是否选定
3	public JRadioButton(String text)	构造	建立一个单选按钮，并指定其文字，默认为不选定
4	public JRadioButton(String text, boolean selected)	构造	建立一个单选按钮，并指定文字和其是否选定
5	public JRadioButton(String text,Icon icon,boolean selected)	构造	建立一个单选按钮，并指定图片、文字和其是否选定
6	public void setSelected(boolean b)	普通	设置是否选中
7	public boolean isSelected()	普通	返回是否被选中
8	public void setText(String text)	普通	设置显示文本
9	public void setIcon(Icon defaultIcon)	普通	设置图片

范例：显示单选按钮

```
package org.lxh.demo18.jradiodemo;
import java.awt.Container;
import java.awt.GridLayout;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.BorderFactory;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JRadioButton;
class MyRadio{
```

```
    private JFrame frame = new JFrame("Welcome To MLDN") ; // 定义窗体
    private Container cont = frame.getContentPane() ;      // 得到窗体容器
    private JRadioButton jrb1 = new JRadioButton("MLDN") ; // 定义一个单选
                                                           按钮
    private JRadioButton jrb2 = new JRadioButton("MLDNJAVA");// 定义一个单选
                                                           按钮
    private JRadioButton jrb3 = new JRadioButton("智囊团");// 定义一个单选
                                                           按钮
    private JPanel pan = new JPanel() ;                    // 定义一个面板
    public MyRadio() {
        // 定义一个面板的边框显示条
        pan.setBorder(BorderFactory.createTitledBorder("请选择最喜爱的网站"));
        pan.setLayout(new GridLayout(1, 3));               // 定义排版，1行
                                                           3列
        pan.add(this.jrb1) ;                               // 加入组件
        pan.add(this.jrb2) ;                               // 加入组件
        pan.add(this.jrb3) ;                               // 加入组件
        cont.add(pan) ;                                    // 加入面板
        this.frame.setSize(330,80) ;                       // 定义窗体大小
        this.frame.setVisible(true) ;                      // 显示窗体
        this.frame.addWindowListener(new WindowAdapter(){  // 加入事件监听
            public void windowClosing(WindowEvent arg0) {  // 覆写窗口关闭
                                                           方法
                System.exit(1) ;                           // 系统退出
            }
        }) ;
    }
}
public class JRadioButtonDemo01 {
    public static void main(String args[]) {
        new MyRadio() ;
    }
}
```

程序运行结果：



以上正确地显示了一个面板，而且因为设置面板时使用了 setBorder()方法，所以在面板上定义了一个标题的边框。

但是此时的程序依然存在一个问题，就是单选按钮只是样式上像了，并不能实现单选的功能。如果现在选中每一个单选按钮，则会出现如下的运行结果：



从运行结果可以发现，此时已经形成了“多选”的功能。之所以会出现这样的问题，主要是由于并没有将所有的单选按钮加入到一个组中。使用 ButtonGroup 就可以将所有的单选按钮加入到一个组中，使用方式如下：

```
ButtonGroup group = new ButtonGroup() ;// 定义一个按钮组
group.add(this.jrb1) ;                 // 将所有单选按钮加入到一个组中
group.add(this.jrb2) ;                 // 将所有单选按钮加入到一个组中
group.add(this.jrb3) ;                 // 将所有单选按钮加入到一个组中
```

此时，再次运行程序，将不会出现多选的情况，每次只会有一个单选按钮选中，如下所示。



18.11.2　JRadioButton 事件处理

在单选按钮操作中，可以使用 ItemListener 接口进行事件的监听。此接口定义了如表 18-31 所示的方法。

表 18-31　ItemListener 接口的方法

序号	方　法	类型	描　述
1	void itemStateChanged(ItemEvent e)	普通	当用户取消或选定某个选项时调用

此方法中存在 ItemEvent 事件，此事件的常用方法及常量如表 18-32 所示。

表 18-32　ItemEvent 类的常用方法及常量

序号	方法及常量	类型	描　述
1	public static final int SELECTED	常量	选项被选中
2	public static final int DESELECTED	常量	选项未被选中
3	public Object getItem()	普通	返回受事件影响的选项
4	public int getStateChange()	普通	返回选定状态的类型（已选择或已取消）

范例：单选按钮事件操作——性别选择

现在按钮的状态使用两张图片来表示，如果已经选中，则显示 right.png 图片；如果没有被选中，则使用 wrong.png 图片。

```
package org.lxh.demo18.jradiodemo;
import java.awt.Container;
import java.awt.GridLayout;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
```

```
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.BorderFactory;
import javax.swing.ButtonGroup;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JRadioButton;
class MyRadio implements ItemListener{
    private String right = "f:" + File.separator + "right.png";// 定义图片
                                                                路径
    private String wrong = "f:" + File.separator + "wrong.png";// 定义图片
                                                                路径
    private JFrame frame = new JFrame("Welcome To MLDN") ; // 定义窗体
    private Container cont = frame.getContentPane() ;      // 得到窗体容器
    private JRadioButton jrb1 = new JRadioButton("男",new ImageIcon(right),
    true) ;
    private JRadioButton jrb2 = new JRadioButton("女",new ImageIcon(wrong),
    false) ;
    private JPanel pan = new JPanel() ;                    // 定义一个面板
    public MyRadio() {
        // 定义一个面板的边框显示条
        pan.setBorder(BorderFactory.createTitledBorder("选择性别"));
        pan.setLayout(new GridLayout(1, 3));               // 定义排版，1行3列
        ButtonGroup group = new ButtonGroup() ;            // 定义一个按钮组
        group.add(this.jrb1) ;                             // 将所有单选按钮加入
                                                              到一个组
        group.add(this.jrb2) ;                             // 将所有单选按钮加入
                                                              到一个组
        pan.add(this.jrb1) ;                               // 加入组件
        pan.add(this.jrb2) ;                               // 加入组件
        jrb1.addItemListener(this) ;                       // 加入事件监听
        jrb2.addItemListener(this) ;                       // 加入事件监听
        cont.add(pan) ;                                    // 加入面板
        this.frame.setSize(200,100) ;                      // 定义窗体大小
        this.frame.setVisible(true) ;                      // 显示窗体
        this.frame.addWindowListener(new WindowAdapter(){// 加入事件监听
            public void windowClosing(WindowEvent arg0) {// 覆写窗口关闭方法
                System.exit(1) ;                           // 系统退出
            }
        }) ;
    }
```

```
        public void itemStateChanged(ItemEvent e) {
            if (e.getSource() == jrb1) {                    // 判断是哪个按钮选中
                jrb1.setIcon(new ImageIcon(right));         // 改变选项图片
                jrb2.setIcon(new ImageIcon(wrong));         // 改变选项图片
            } else {
                jrb1.setIcon(new ImageIcon(wrong));         // 改变选项图片
                jrb2.setIcon(new ImageIcon(right));         // 改变选项图片
            }
        }
    }
    public class JRadioButtonDemo03 {
        public static void main(String args[]) {
            new MyRadio() ;
        }
    }
```

程序运行结果：



在以上程序中，使用 ImageIcon 设置了两个单选按钮的图片，每次选项改变后都会触发 itemStateChanged 事件，之后修改每个选项显示图片。

18.12　复选框：JCheckBox

18.12.1　认识 JCheckBox

程序可以通过 JRadioButton 实现单选按钮的功能，那么如果要实现复选框的功能，则必须使用 JCheckBox 完成。此类的常用方法如表 18-33 所示。

表 18-33　JCheckBox 类的常用方法

序号	方　　法	类型	描　　述
1	public JCheckBox(Icon icon)	构造	创建一个带图标的对象，但不选定
2	public JCheckBox(Icon icon,boolean selected)	构造	创建一个带图标的对象，并指定其是否选定
3	public JCheckBox(String text)	构造	创建一个带文本的对象，但不选定
4	public JCheckBox(String text,boolean selected)	构造	创建一个带文本的对象，并指定其是否选定
5	public JCheckBox(String text,Icon icon,boolean selected)	构造	创建一个带文本和图标的对象，并指定是否被选定

JCheckBox 的其他使用方法和 JRadioButton 类的操作类似，下面介绍如何构建一个复选框操作。

范例：定义复选框

```
package org.lxh.demo18.jcheckdemo;
import java.awt.Container;
import java.awt.GridLayout;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.BorderFactory;
import javax.swing.JCheckBox;
import javax.swing.JFrame;
import javax.swing.JPanel;
class MyCheckBox {
    private JFrame frame = new JFrame("Welcome To MLDN") ; // 定义窗体
    private Container cont = frame.getContentPane() ;      // 得到窗体容器
    private JCheckBox jcb1 = new JCheckBox("MLDN") ;       // 定义一个复选框
    private JCheckBox jcb2 = new JCheckBox("MLDNJAVA") ;   // 定义一个复选框
    private JCheckBox jcb3 = new JCheckBox("智囊团") ;      // 定义一个复选框
    private JPanel pan = new JPanel() ;                    // 定义一个面板
    public MyCheckBox() {
        // 定义一个面板的边框显示条
        pan.setBorder(BorderFactory.createTitledBorder("请选择最喜爱的网站"));
        pan.setLayout(new GridLayout(1, 3));               // 定义排版，1行
                                                           3列
        pan.add(this.jcb1) ;                               // 加入组件
        pan.add(this.jcb2) ;                               // 加入组件
        pan.add(this.jcb3) ;                               // 加入组件
        cont.add(pan) ;                                    // 加入面板
        this.frame.setSize(330,80) ;                       // 定义窗体大小
        this.frame.setVisible(true) ;                      // 显示窗体
        this.frame.addWindowListener(new WindowAdapter(){  // 加入事件监听
            public void windowClosing(WindowEvent arg0) {  // 覆写窗口关闭
                                                           方法
                System.exit(1) ;                           // 系统退出
            }
        }) ;
    }
}
public class JCheckBoxDemo01 {
    public static void main(String args[]) {
        new MyCheckBox() ;
    }
}
```

程序运行结果：



读者可以发现，与单选按钮不同的是，此时选择的地方变成了“□”型，这与各个系统中常见的复选框功能是一样的。

18.12.2　JCheckBox 事件处理

JCheckBox 和 JRadioButton 的事件处理监听接口是一样的，都是使用 ItemListener 接口。下面来观察此接口在 JCheckBox 上的应用。

范例：复选框事件操作

```
package org.lxh.demo18.jcheckdemo;
import java.awt.Container;
import java.awt.GridLayout;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.io.File;
import javax.swing.BorderFactory;
import javax.swing.ImageIcon;
import javax.swing.JCheckBox;
import javax.swing.JFrame;
import javax.swing.JPanel;
class MyItemListener implements ItemListener{
    private String right = "f:" + File.separator + "right.png";// 定义图片
                                                                路径
    private String wrong = "f:" + File.separator + "wrong.png";// 定义图片
                                                                路径
    public void itemStateChanged(ItemEvent e) {
        JCheckBox jcb = (JCheckBox) e.getItem();
        if (jcb.isSelected()) {                                 // 判断是否
                                                                   被选中
            jcb.setIcon(new ImageIcon(right)) ;                 // 修改图片
        } else {
            jcb.setIcon(new ImageIcon(wrong)) ;                 // 修改图片
        }
    }
}
class MyCheckBox{
```

```
    private String wrong = "f:" + File.separator + "wrong.png";// 定义图片路径
    private JFrame frame = new JFrame("Welcome To MLDN") ;  // 定义窗体
    private Container cont = frame.getContentPane() ;       // 得到窗体容器
    // 定义3个带文本、图片的复选框
    private JCheckBox jcb1 = new JCheckBox("MLDN", new ImageIcon(wrong));
    private JCheckBox jcb2 = new JCheckBox("MLDNJAVA", new ImageIcon(wrong));
    private JCheckBox jcb3 = new JCheckBox("智囊团", new ImageIcon(wrong));
    private JPanel pan = new JPanel() ;                     // 定义一个面板
    public MyCheckBox() {
        // 定义一个面板的边框显示条
        pan.setBorder(BorderFactory.createTitledBorder("请选择最喜爱的网站"));
        pan.setLayout(new GridLayout(1, 3));                // 定义排版，1行
                                                               3列
        pan.add(this.jcb1) ;                                // 加入组件
        pan.add(this.jcb2) ;                                // 加入组件
        pan.add(this.jcb3) ;                                // 加入组件
        jcb1.addItemListener(new MyItemListener()) ;        // 加入监听
        jcb2.addItemListener(new MyItemListener()) ;        // 加入监听
        jcb3.addItemListener(new MyItemListener()) ;        // 加入监听
        cont.add(pan) ;                                     // 加入面板
        this.frame.setSize(330,100) ;                       // 定义窗体大小
        this.frame.setVisible(true) ;                       // 显示窗体
        this.frame.addWindowListener(new WindowAdapter(){   // 加入事件监听
            public void windowClosing(WindowEvent arg0) {   // 覆写窗口关闭
                                                               方法
                System.exit(1) ;                            // 系统退出
            }
        }) ;
    }
}
public class JCheckBoxDemo02 {
    public static void main(String args[]) {
        new MyCheckBox() ;
    }
}
```

程序运行结果：



此程序的功能与 JRadioButton 的事件监听操作非常相似，当选择好选项之后根据选定的状态设置其显示的图片。

18.13　列表框：JList

18.13.1　认识 JList

列表框可以同时将多个选项信息以列表的方式展现给用户，使用 JList 可以构建一个列表框。JList 的常用方法如表 18-34 所示。

表 18-34　JList 的常用方法

序号	方　法	类型	描　述
1	public JList(ListModel dataModel)	构造	根据 ListModel 构造 JList
2	public JList(Object[] listData)	构造	根据对象数组构造 JList
3	public JList(Vector<?> listData)	构造	根据一个 Vector 构造 JList
4	public void setSelectionMode(int selectionMode)	普通	设置选择模式，是多选还是单选
5	public ListModel getModel()	普通	返回列表框的列表模型
6	public int[] getSelectedIndices()	普通	返回所选择的全部数组

对于列表框是多选还是单选可以通过 ListSelectionModel 接口完成，此接口定义了如表 18-35 所示的常量。

表 18-35　ListSelectionModel 定义的常量

序号	常　量	类型	描　述
1	static final int MULTIPLE_INTERVAL_SELECTION	常量	一次选择一个或多个连续的索引范围
2	static final int SINGLE_INTERVAL_SELECTION	常量	一次选择一个连续范围的值
3	static final int SINGLE_SELECTION	常量	一次选择一个值

范例：使用 JList 实现列表框的功能

```
package org.lxh.demo18.jlistdemo;
import java.awt.Container;
import java.awt.GridLayout;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.util.Vector;
import javax.swing.BorderFactory;
import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.ListSelectionModel;
class MyList{
    private JFrame frame = new JFrame("Welcome To MLDN") ; // 定义窗体
    private Container cont = frame.getContentPane() ;      // 得到窗体容器
    private JList list1 = null ;                           // 定义列表框
```

```
    private JList list2 = null ;                                    // 定义列表框
    public MyList() {
        this.frame.setLayout(new GridLayout(1, 3));                 // 定义排版，1行2列
        String nations[] = { "中国", "巴西", "美国", "韩国", "意大利", "法国" };
        Vector<String> v = new Vector<String>() ;                   // 定义一个Vector集合
        v.add("MLDN") ;                                             // 加入元素
        v.add("MLDNJAVA") ;                                         // 加入元素
        v.add("智囊团") ;                                           // 加入元素
        this.list1 = new JList(nations) ;                           // 实例化JList
        this.list2 = new JList(v) ;                                 // 实例化JList
        // 定义一个列表框的边框显示条
        list1.setBorder(BorderFactory.createTitledBorder("你喜欢去哪个国家
        旅游？"));
        list2.setBorder(BorderFactory.createTitledBorder("你喜欢的网站？"));
        // 第一个列表框每次可以选择多个选项
        list1.setSelectionMode(ListSelectionModel.MULTIPLE_INTERVAL_
        SELECTION);
        // 第二个列表框每次只能选择一个选项
        list2.setSelectionMode(ListSelectionModel.SINGLE_SELECTION) ;
        cont.add(this.list1) ;                                      // 加入面板
        cont.add(this.list2) ;                                      // 加入面板
        this.frame.setSize(330,180) ;                               // 定义窗体大小
        this.frame.setVisible(true) ;                               // 显示窗体
        this.frame.addWindowListener(new WindowAdapter(){// 加入事件监听
            public void windowClosing(WindowEvent arg0){ // 覆写窗口关闭方法
                System.exit(1) ;                                    // 系统退出
            }
        }) ;
    }
}
public class JListDemo01 {
    public static void main(String args[]) {
        new MyList() ;
    }
}
```

程序运行结果：



以上程序中第一个 JList 通过字符串数组设置列表的内容，并通过 setSelectionMode()方法设置一次性可以选择多个选项；第二个 JList 通过 Vector 设置列表的内容，并通过 setSelectionMode()方法将其设置成每次只能选择一个选项。

提示：在 JList 中要使用 JScrollPane 才能出现滚动条。

在 JList 显示时如果内容过多不会将其加入滚动条显示，要想进行滚动条的显示，则必须将 JList 加入到一个 JScrollPane 中才可以。

18.13.2　使用 ListModel 构造 JList

在 JList 的构造方法中有一个使用 ListModel 构造 JList 对象的操作，ListModel 是一个专门用于创建 JList 列表内容操作的接口，在此接口中定义了如表 18-36 所示的方法。

表 18-36　ListModel 定义的方法

序号	方　法	类型	描　述
1	void addListDataListener(ListDataListener l)	普通	加入数据改变事件的监听
2	void removeListDataListener(ListDataListener l)	普通	加入数据删除时的监听
3	Object getElementAt(int index)	普通	返回指定索引处的内容
4	int getSize()	普通	返回列表长度

以上虽然定义了 4 个方法，但是在实际的开发中比较常用的只有 getElementAt()和 getSize()两个方法，所以一般在使用 ListModel 时都很少让子类直接实现此接口，而是通过继承 AbstractListModel 完成。下面使用 AbstractListModel 完成一个列表框的显示操作。

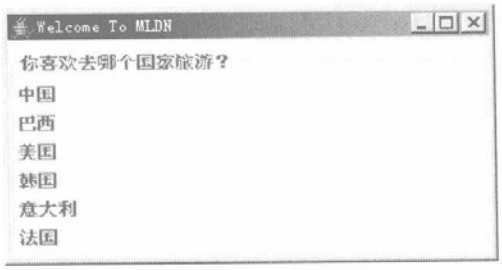
范例：利用 AbstractListModel 创建 JList

```
package org.lxh.demo18.jlistdemo;
import java.awt.Container;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.AbstractListModel;
import javax.swing.BorderFactory;
import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.ListSelectionModel;
class MyListModel extends AbstractListModel{                    // 定义列表内容
    private String nations[] = { "中国", "巴西", "美国", "韩国", "意大利",
    "法国" };
    public Object getElementAt(int ind) {
        if (ind < this.nations.length) {                        // 返回内容
            return this.nations[ind];
        } else {
            return null;
        }
```

```
    }
    public int getSize() {
        return this.nations.length;
    }
}
class MyList {
    private JFrame frame = new JFrame("Welcome To MLDN") ; // 定义窗体
    private Container cont = frame.getContentPane() ;      // 得到窗体容器
    private JList list1 = null ;                           // 定义列表框
    public MyList() {
        this.list1 = new JList(new MyListModel()) ;        // 实例化JList
        // 定义一个列表框的边框显示条
        list1.setBorder(BorderFactory.createTitledBorder("你喜欢去哪个国家
        旅游？"));
        // 第一个列表框每次可以选择多个选项

list1.setSelectionMode(ListSelectionModel.MULTIPLE_INTERVAL_SELECTION) ;
        cont.add(this.list1) ;                                 // 加入面板
        this.frame.setSize(330,180) ;                          // 定义窗体大小
        this.frame.setVisible(true) ;                          // 显示窗体
        this.frame.addWindowListener(new WindowAdapter(){      // 加入事件监听
            public void windowClosing(WindowEvent arg0) {      // 覆写窗口关闭
                                                               方法
                System.exit(1) ;                               // 系统退出
            }
        }) ;
    }
}
public class JListDemo02 {
    public static void main(String args[]) {
        new MyList() ;
    }
}
```

程序运行结果：



本程序实现了与之前相同的结果，使用一个类保存列表内容，这样在操作时比较方便。

18.13.3　JList 事件处理

在 JList 中可以使用 ListSelectionListener 的监听接口实现对 JList 中的选项进行监听，此接口定义了如表 18-37 所示的方法。

表 18-37　ListSelectionListener 接口定义的方法

序号	方　　法	类型	描　　述
1	void valueChanged(ListSelectionEvent e)	普通	当值发生改变时调用

valueChanged()方法会产生 ListSelectionEvent 事件，此事件中的常用方法如表 18-38 所示。

表 18-38　ListSelectionEvent 事件中的常用方法

序号	方　　法	类型	描　　述
1	public int getFirstIndex()	普通	返回选择的第一个选项的索引值
2	public int getLastIndex()	普通	返回选择的最后一个选项的索引值

范例：对 JList 进行监听

```
package org.lxh.demo18.jlistdemo;
import java.awt.Container;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.AbstractListModel;
import javax.swing.BorderFactory;
import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.JScrollPane;
import javax.swing.event.ListSelectionEvent;
import javax.swing.event.ListSelectionListener;
class MyListModel extends AbstractListModel{                    // 定义列表内容
    private String nations[] = { "中国", "巴西", "美国", "韩国", "意大利",
    "法国" };
    public Object getElementAt(int ind) {
        if (ind < this.nations.length) {                        // 返回内容
            return this.nations[ind];
        } else {
            return null;
        }
    }
    public int getSize() {
        return this.nations.length;
    }
}
```

```
class MyList implements ListSelectionListener{                    // 实现监听接口
    private JFrame frame = new JFrame("Welcome To MLDN") ; // 定义窗体
    private Container cont = frame.getContentPane() ;              // 得到窗体容器
    private JList list1 = null ;                                   // 定义列表框
    public MyList() {
        this.list1 = new JList(new MyListModel()) ;                // 实例化JList
        // 定义一个列表框的边框显示条
        list1.setBorder(BorderFactory.createTitledBorder("你喜欢去哪个国家
        旅游？"));
        list1.addListSelectionListener(this) ;                     // 加入事件监听
        cont.add(new JScrollPane(this.list1)) ;                    // 加入滚动条
        this.frame.setSize(330,180) ;                              // 定义窗体大小
        this.frame.setVisible(true) ;                              // 显示窗体
        this.frame.addWindowListener(new WindowAdapter(){          // 加入事件监听
            public void windowClosing(WindowEvent arg0) {          // 覆写窗口关闭
                                                                   方法
                System.exit(1) ;                                   // 系统退出
            }
        }) ;
    }
    public void valueChanged(ListSelectionEvent e) {
        int temp[] = list1.getSelectedIndices() ;                  // 取得全部选定
                                                                   索引
        System.out.print("选定的内容：") ;
        for (int i = 0; i < temp.length; i++) {
            System.out.print(list1.getModel().getElementAt(i) + "、") ;
        }
        System.out.println() ;                                     // 换行
    }
}
public class JListDemo03 {
    public static void main(String args[]) {
        new MyList() ;
    }
}
```

程序运行结果：




```
选定的内容：巴西、
选定的内容：巴西、
选定的内容：巴西、韩国、
选定的内容：巴西、韩国、
选定的内容：巴西、韩国、法国、
选定的内容：巴西、韩国、法国、
```

18.14　下拉列表框：JComboBox

18.14.1　认识 JComboBox

JList 是以列表框的形式进行显示的，当然，在实际的软件使用中，读者应该经常会看到如图 18-18 所示的一种列表框，这种列表框就称为下拉列表框。在 Swing 中使用 JComboBox 类即可完成这样的功能。

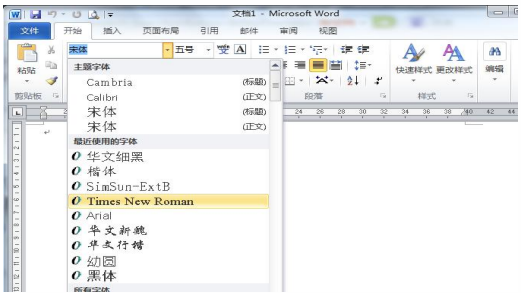


图 18-18　下拉列表框

JComboBox 类的常用方法如表 18-39 所示。

表 18-39　JComboBox 类的常用方法

序号	方　　法	类型	描　　述
1	public JComboBox(ComboBoxModel aModel)	构造	利用 ComboBox 构造一个 JComboBox 对象
2	public JComboBox(Object[] items)	构造	利用对象数组构造一个 JComboBox 对象
3	public JComboBox(Vector<?> items)	构造	利用 Vector 构造一个 JComboBox 对象
4	public Object getItemAt(int index)	普通	返回指定索引处的列表项
5	public int getItemCount()	普通	返回列表中的项数
6	public void addItem(Object anObject)	普通	为列表增加内容
7	public void setEditable(boolean aFlag)	普通	设置此下拉列表框是否可编辑
8	public void setMaximumRowCount(int count)	普通	设置下拉列表框显示的最大行数
9	public void setSelectedIndex(int anIndex)	普通	设置默认选项的索引号
10	public ComboBoxEditor getEditor()	普通	返回 JComboBox 的内容编辑器
11	public void configureEditor(ComboBoxEditor anEditor,Object anItem)	普通	初始化编辑器

范例：建立下拉列表框

```
package org.lxh.demo18.jcombodemo;
import java.awt.Container;
import java.awt.GridLayout;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.util.Vector;
```

```
import javax.swing.BorderFactory;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
class MyComboBox{
    private JFrame frame = new JFrame("Welcome To MLDN");// 定义窗体
    private Container cont = frame.getContentPane() ;  // 得到窗体容器
    private JComboBox jcb1 = null ;                    // 定义下拉列表框
    private JComboBox jcb2 = null ;                    // 定义下拉列表框
    public MyComboBox() {
        this.frame.setLayout(new GridLayout(2,2));     // 定义排版
        String nations[] = { "中国", "巴西", "美国", "韩国", "意大利", "法国" };
        Vector<String> v = new Vector<String>() ;      // 定义一个Vector集合
        v.add("MLDN") ;                                // 加入元素
        v.add("MLDNJAVA") ;                            // 加入元素
        v.add("智囊团") ;                              // 加入元素
        this.jcb1 = new JComboBox(nations) ;           // 实例化JComboBox
        this.jcb2 = new JComboBox(v) ;                 // 实例化JComboBox
        // 定义一个列表框的边框显示条
        jcb1.setBorder(BorderFactory.createTitledBorder("你喜欢去哪个国家旅
        游？"));
        jcb2.setBorder(BorderFactory.createTitledBorder("你喜欢的网站？"));
        jcb1.setMaximumRowCount(3) ;                   // 最多显示3个选项
        jcb2.setMaximumRowCount(3) ;                   // 最多显示3个选项
        cont.add(this.jcb1) ;                          // 加入组件
        cont.add(this.jcb2) ;                          // 加入组件
        cont.add(new JLabel("下拉列表框")) ;           // 加入组件
        this.frame.setSize(300,150) ;                  // 定义窗体大小
        this.frame.setVisible(true) ;                  // 显示窗体
        this.frame.addWindowListener(new WindowAdapter(){// 加入事件监听
            public void windowClosing(WindowEvent arg0){// 覆写窗口关闭方法
                System.exit(1) ;                       // 系统退出
            }
        }) ;
    }
}
public class JComboBoxDemo01 {
    public static void main(String args[]) {
        new MyComboBox() ;
    }
}
```

程序运行结果：



以上程序中使用 JComboBox 建立了两个下拉列表框，每个下拉列表框可以出现选项的个数为 3 个。

提示：在 JComboBox 中没有必要设置选择的选项数。

在 JList 中可以通过 setSelectionMode()方法设置选择的选项数，但是在 JComboBox 中因为每次只能选择一项，所以没有必要再进行此种设置。

18.14.2　使用 ComboBoxModel 构造 JComboBox

与 JList 一样，在 JComboBox 中也可以使用 ComboBoxModel 接口来构造 JComboBox 中的列表内容，ComboBoxModel 接口是 ListModel 接口的子类，除了继承 ListModel 中的所有操作方法之外，还增加了如表 18-40 所示的方法。

表 18-40　ComboBoxModel 接口增加的方法

序号	方　法	类型	描　述
1	void setSelectedItem(Object anItem)	普通	设置选项内容
2	Object getSelectedItem()	普通	返回选择项

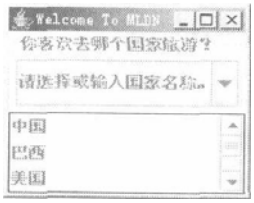
范例：使用 ComboBoxModel 和 AbstractListModel 构造可编辑的 JcomboBox

```
package org.lxh.demo18.jcombodemo;
import java.awt.Container;
import java.awt.GridLayout;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.AbstractListModel;
import javax.swing.BorderFactory;
import javax.swing.ComboBoxEditor;
import javax.swing.ComboBoxModel;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
class MyComboBoxModel extends AbstractListModel implements ComboBoxModel{
    String nations[] = { "中国", "巴西", "美国", "韩国", "意大利", "法国" };
    String item = null ;                                    // 保存选项内容
    public Object getSelectedItem() {                       // 返回选项
```

```
        return this.item;
    }
    public void setSelectedItem(Object arg) {
        this.item = (String) arg;                          // 设置选项内容
    }
    public Object getElementAt(int ind) {                  // 根据编号返回
                                                              选项
        return this.nations[ind];
    }
    public int getSize() {                                 // 取得选项长度
        return this.nations.length ;
    }
}
class MyComboBox{
    private String defaultMsg = "请选择或输入国家名称。" ;     // 定义默认值
    private JFrame frame = new JFrame("Welcome To MLDN") ; // 定义窗体
    private Container cont = frame.getContentPane() ;      // 得到窗体容器
    private JComboBox jcb1 = null ;                        // 定义下拉列表框
    public MyComboBox() {
        this.frame.setLayout(new GridLayout(2,1));         // 定义排版
        this.jcb1 = new JComboBox(new MyComboBoxModel()) ; // 实例化
                                                              JComboBox
        // 定义一个列表框的边框显示条
        jcb1.setBorder(BorderFactory.createTitledBorder("你喜欢去哪个国家旅
        游？"));
        jcb1.setMaximumRowCount(3) ;                       // 最多显示3个
                                                              选项
        jcb1.setEditable(true) ;                           // 设置可编辑
        ComboBoxEditor editor = null ;                     // 编辑器对象
        editor = jcb1.getEditor() ;                        // 实例化编辑器
        jcb1.configureEditor(editor, defaultMsg) ;         // 配置编辑器及
                                                              默认值
        cont.add(this.jcb1) ;                              // 加入组件
        cont.add(new JLabel("下拉列表框")) ;                // 加入组件
        this.frame.setSize(180,150) ;                      // 定义窗体大小
        this.frame.setVisible(true) ;                      // 显示窗体
        this.frame.addWindowListener(new WindowAdapter(){  // 加入事件监听
            public void windowClosing(WindowEvent arg0) {  // 覆写窗口关闭
                                                              方法
                System.exit(1) ;                           // 系统退出
            }
        }) ;
```

```
    }
}
public class JComboBoxDemo02 {
    public static void main(String args[]) {
        new MyComboBox() ;
    }
}
```

程序运行结果：



在本程序中使用 ComboBoxModel 构造了 JComBox 对象，在程序中将 JComboBox 设置为可编辑，即用户可以直接通过文本选择选项的内容。

18.14.3　JComboBox 事件处理

JComboBox 依然可以使用之前学习过的 ItemListener 作为事件的监听接口。另外，因为其可以直接在文本框中输入内容，所以在处理时也要处理文本的行为事件，即还要使用 ActionListener 接口。

范例：JComboBox 事件处理

```
package org.lxh.demo18.jcombodemo;
import java.awt.Container;
import java.awt.Font;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.BorderFactory;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
class MyComboBox implements ItemListener, ActionListener {
    private JFrame frame = new JFrame("Welcome To MLDN") ; // 定义窗体
    private Container cont = frame.getContentPane() ;      // 得到窗体容器
    private JComboBox jcb1 = null ;                        // 定义下拉列表框
    private JLabel label = new JLabel("www.MLDNJAVA.cn") ; // 定义一个标签
```

```
    private String fontSize[] = { "10", "12", "14", "16", "18", "20", "22",
            "24", "26", "48", "72" };                          // 定义文字大小
    public MyComboBox() {
        this.frame.setLayout(new GridLayout(2,1));             // 定义排版
        this.jcb1 = new JComboBox(this.fontSize) ;             // 实例化JComboBox
        // 定义一个列表框的边框显示条
        jcb1.setBorder(BorderFactory.createTitledBorder("请选择显示文字大
        小。"));
        jcb1.addItemListener(this) ;                           // 加入选项监听
        jcb1.addActionListener(this) ;                         // 加入动作监听
        jcb1.setMaximumRowCount(3) ;                           // 最多显示3个选项
        jcb1.setEditable(true) ;                               // 设置可编辑文本
        jcb1.configureEditor(jcb1.getEditor(), "12") ; // 定义默认值
        this.changeFontSize(12) ;                              // 设置默认字体
        cont.add(this.jcb1) ;                                  // 加入组件
        cont.add(label) ;                                      // 加入组件
        this.frame.setSize(300,150) ;                          // 定义窗体大小
        this.frame.setVisible(true) ;                          // 显示窗体
        this.frame.addWindowListener(new WindowAdapter(){// 加入事件监听
            public void windowClosing(WindowEvent arg0) {// 覆写窗口关闭方法
                System.exit(1) ;                               // 系统退出
            }
        }) ;
    }
    public void itemStateChanged(ItemEvent e) {            // 选项改变时触发
        if(e.getStateChange()==ItemEvent.SELECTED){        // 判断是否选定
            String itemSize = (String) e.getItem();        // 得到选项
            try{
                this.changeFontSize(Integer.parseInt(itemSize));// 改变字体
            }catch(Exception ex){}
        }
    }
    public void actionPerformed(ActionEvent e) {           // 输入信息时触发
        String itemSize = (String) this.jcb1.getSelectedItem();// 得到选项
        int size = 12 ;                                    // 定义默认值
        try{
            size = Integer.parseInt(itemSize) ;            // 字符串转整数
        }catch(Exception ex){
            this.jcb1.getEditor().setItem("输入的不是数字。") ;// 显示错误信息
        }
        this.changeFontSize(size) ;                        // 改变字体大小
        if (!this.isExists(itemSize)) {                    // 判断是否存在
```

```
                this.jcb1.addItem(jcb1.getSelectedItem());      // 不存在，加入下
                                                                   拉选项
            }
        }
        public boolean isExists(String item) {                  // 判断指定的内
                                                                   容是否存在
            boolean flag = false ;                              // 定义一个标志
                                                                   变量
            for (int i = 0; i < this.jcb1.getItemCount(); i++){// 循环判断
                if (item.equals(this.jcb1.getItemAt(i))){       // 判断是否存在
                    flag = true ;                               // 存在修改标志
                                                                   变量
                }
            }
            return flag ;
        }
        public void changeFontSize(int size) {                  // 改变文字大小
            Font fnt = new Font("Serief", Font.BOLD, size);     // 定义Font对象
            this.label.setFont(fnt) ;                           // 设置文字大小
        }
    }
    public class JComboBoxDemo03 {
        public static void main(String args[]) {
            new MyComboBox() ;
        }
    }
```

程序运行结果：



程序运行后，不管是通过下拉列表框选择，还是通过文本框直接输入，都可以改变标签中的文字大小。如果输入的不是数字，则会将错误信息提示给用户。

18.15　菜 单 组 件

18.15.1　JMenu 与 JMenuBar

对于菜单读者应该都不会陌生，在 Windows 中经常会看到如图 18-19 所示的菜单。

若在 Java 中实现此类菜单，则可以使用 JMenu 组件，当然，如果要使用 JMenu 则首先

要了解 JMenuBar 组件。JMenuBar 组件的功能是用来摆放 JMenu 组件，当建立完许多的 JMenu 组件之后，需要通过 JMenuBar 组件来将 JMenu 组件加入到窗口中。JMenuBar 的常用方法如表 18-41 所示，JMenu 的常用方法如表 18-42 所示。



图 18-19　菜单

表 18-41　JMenuBar 的常用方法

序号	方　法	类型	描　述
1	public JMenuBar()	构造	创建新的 JMenuBar 对象
2	public JMenu add(JMenu c)	普通	将指定的 JMenu 加入到 JMenuBar 中
3	public JMenu getMenu(int index)	普通	返回指定位置的菜单
4	public int getMenuCount()	普通	返回菜单栏上的菜单数

表 18-42　JMenu 的常用方法

序号	方　法	类型	描　述
1	public JMenu(String s)	构造	创建新的 JMenu，并指定菜单名称
2	public JMenuItem add(JMenuItem menuItem)	普通	增加新的菜单项
3	public void addSeparator()	普通	加入分隔线

下面使用 JMenu 和 JMenuBar 构建一个简单的菜单。

范例：使用 JMenu 和 JMenuBar 构建简单的菜单

```
package org.lxh.demo18.menudemo;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.io.File;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
public class JMenuDemo01 {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Welcome To MLDN");          // 定义窗体
        JTextArea text = new JTextArea();                      // 定义文本域
        text.setEditable(true);                                // 定义文本组件
                                                               可编辑
        frame.getContentPane().add(new JScrollPane(text)); // 在面板中加入文
                                                               本框及滚动条
        JMenu menuFile = new JMenu("文件");                    // 定义JMenu组件
        menuFile.setIcon(new ImageIcon("d:" + File.separator + "icons"
```

```
                + File.separator + "file.gif"));                    // 定义显示图标
        JMenuBar menuBar = new JMenuBar();                          // 定义JMenuBar
        menuBar.add(menuFile);                                      // 加入JMenu
        frame.addWindowListener(new WindowAdapter() {               // 加入事件监听
            public void windowClosing(WindowEvent arg0) {           // 覆写窗口关闭
                                                                        方法
                System.exit(1);                                     // 系统退出
            }});
        frame.setJMenuBar(menuBar);                                 // 在窗体中加入
                                                                        JMenuBar组件
        frame.setVisible(true);                                     // 显示组件
        frame.setSize(300, 100);                                    // 定义窗体大小
        frame.setLocation(300, 200);                                // 设置显示位置
    }
}
```

程序运行结果：



在程序中首先建立了一个 JMenu，之后将 JMenu 的对象加入到 JMenuBar 中，再通过 JFrame 中的 setJMenuBar()方法将此菜单栏加入到窗体上。

18.15.2　JMenuItem

之前只是构建了一个菜单，但是此菜单中没有任何的菜单项，如果想为菜单加入菜单项，则可以使用 JMenuItem 组件。

JMenuItem 继承 AbstractButton 类，因此 JMenuItem 也可以理解为一种特殊的 JButton 组件，当选择某个菜单项时也会触发 ActionEvent 事件。JMenuItem 类的常用方法如表 18-43 所示。

表 18-43　JMenuItem 类的常用方法

序号	方　　法	类型	描　　述
1	public JMenuItem(Icon icon)	构造	创建带有图标的 JMenuItem
2	public JMenuItem(String text)	构造	创建带有指定文本的 JMenuItem
3	public JMenuItem(String text,Icon icon)	构造	创建带有指定文本和图标的 JMenuItem
4	public JMenuItem(String text, int mnemonic)	构造	创建带有指定文本的 JMenuItem，并指定助记符
5	public void setMnemonic(int mnemonic)	普通	指定菜单项的助记符
6	public void setAccelerator(KeyStroke keyStroke)	普通	设置快捷键的组合键

范例：构建菜单项

```
package org.lxh.demo18.menudemo;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.io.File;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.KeyStroke;
public class JMenuDemo02 {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Welcome To MLDN");  // 定义窗体
        JTextArea text = new JTextArea();              // 定义文本域
        text.setEditable(true);                        // 定义文本组件可编辑
        frame.getContentPane().add(new JScrollPane(text)); // 在面板中加入文本
                                                           框及滚动条
        JMenu menuFile = new JMenu("文件");            // 定义JMenu组件
        menuFile.setIcon(new ImageIcon("d:" + File.separator + "icons"
                + File.separator + "file.gif"));       // 定义显示图标
        JMenuBar menuBar = new JMenuBar();             // 定义JMenuBar
        JMenuItem newItem = new JMenuItem("新建",
                new ImageIcon("d:" + File.separator + "icons"
                        + File.separator + "new.gif"));// 创建JMenuItem
        JMenuItem openItem = new JMenuItem("打开",
                new ImageIcon("d:" + File.separator + "icons"
                        + File.separator + "open.gif"));// 创建JMenuItem
        JMenuItem closeItem = new JMenuItem("关闭",
                new ImageIcon("d:" + File.separator + "icons"
                        + File.separator + "close.gif"));// 创建JMenuItem
        JMenuItem exitItem = new JMenuItem("退出",
                new ImageIcon("d:" + File.separator + "icons"
                        + File.separator + "exit.gif"));// 创建JMenuItem
        newItem.setMnemonic('N') ;                     // 设置快捷键N
        openItem.setMnemonic('O') ;                    // 设置快捷键O
        closeItem.setMnemonic('C') ;                   // 设置快捷键C
        exitItem.setMnemonic('E') ;                    // 设置快捷键E
        newItem.setAccelerator(KeyStroke.getKeyStroke('N',
                java.awt.Event.CTRL_MASK));            // Ctrl + N
```

```
        openItem.setAccelerator(KeyStroke.getKeyStroke('O',
                java.awt.Event.CTRL_MASK));                    // Ctrl + O
        closeItem.setAccelerator(KeyStroke.getKeyStroke('C',
                java.awt.Event.ALT_MASK));                     // Alt + C
        exitItem.setAccelerator(KeyStroke.getKeyStroke('E',
                java.awt.Event.ALT_MASK));                     // Alt + E
        menuFile.add(newItem) ;                                // 加入菜单项
        menuFile.add(openItem) ;                               // 加入菜单项
        menuFile.add(closeItem) ;                              // 加入菜单项
        menuFile.addSeparator() ;                              // 加入分隔线
        menuFile.add(exitItem) ;                               // 加入菜单项
        menuBar.add(menuFile);                                 // 加入JMenu
        frame.addWindowListener(new WindowAdapter() {          // 加入事件监听
            public void windowClosing(WindowEvent arg0) {      // 覆写窗口关闭
                                                                  方法
                System.exit(1);                                // 系统退出
            }});
        frame.setJMenuBar(menuBar);                            // 在窗体中加入
                                                                  JMenuBar组件
        frame.setVisible(true);                                // 显示组件
        frame.setSize(300, 180);                               // 定义窗体大小
        frame.setLocation(300, 200);                           // 设置显示位置
    }
}
```

程序运行结果：



以上程序在一个菜单中加入了 4 个菜单项，并使用 addSeparator()方法增加了一条分割线，之后程序为了每个菜单的操作方便，又分别为每一个菜单项设置了快捷键。

18.15.3　事件处理

JMenuItem 与 JButton 都是 AbstractButton 类的子类，所以事件处理机制是完全一样的，选择一个菜单项实际上与单击一个按钮的效果是完全一样的。

范例：加入事件监听处理

```
package org.lxh.demo18.menudemo;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
```

```
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.io.File;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.KeyStroke;
public class JMenuDemo03 {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Welcome To MLDN");          // 定义窗体
        final JTextArea text = new JTextArea();                // 定义文本域，可
                                                               以被内部类访问
        text.setEditable(true);                                // 定义文本组件
                                                               可编辑
        frame.getContentPane().add(new JScrollPane(text)); // 在面板中加入文
                                                               本框及滚动条
        JMenu menuFile = new JMenu("文件");                    // 定义JMenu组件
        menuFile.setIcon(new ImageIcon("d:" + File.separator + "icons"
                + File.separator + "file.gif"));               // 定义显示图标
        JMenuBar menuBar = new JMenuBar();                     // 定义JMenuBar
        JMenuItem newItem = new JMenuItem("新建",
                new ImageIcon("d:" + File.separator + "icons"
                        + File.separator + "new.gif"));// 创建JMenuItem
        JMenuItem openItem = new JMenuItem("打开",
                new ImageIcon("d:" + File.separator + "icons"
                        + File.separator + "open.gif"));// 创建JMenuItem
        JMenuItem closeItem = new JMenuItem("关闭",
                new ImageIcon("d:" + File.separator + "icons"
                        + File.separator + "close.gif"));// 创建JMenuItem
        JMenuItem exitItem = new JMenuItem("退出",
                new ImageIcon("d:" + File.separator + "icons"
                        + File.separator + "exit.gif"));// 创建JMenuItem
        newItem.setMnemonic('N') ;                             // 设置快捷键N
        openItem.setMnemonic('O') ;                            // 设置快捷键O
        closeItem.setMnemonic('C') ;                           // 设置快捷键C
        exitItem.setMnemonic('E') ;                            // 设置快捷键E
        newItem.setAccelerator(KeyStroke.getKeyStroke('N',
                java.awt.Event.CTRL_MASK));                    // Ctrl + N
```

```
        openItem.setAccelerator(KeyStroke.getKeyStroke('O',
                java.awt.Event.CTRL_MASK));          // Ctrl + O
        closeItem.setAccelerator(KeyStroke.getKeyStroke('C',
                java.awt.Event.ALT_MASK));           // Alt + C
        exitItem.setAccelerator(KeyStroke.getKeyStroke('E',
                java.awt.Event.ALT_MASK));           // Alt + E
        menuFile.add(newItem) ;                      // 加入菜单项
        menuFile.add(openItem) ;                     // 加入菜单项
        menuFile.add(closeItem) ;                    // 加入菜单项
        menuFile.addSeparator() ;                    // 加入分隔线
        menuFile.add(exitItem) ;                     // 加入菜单项
        menuBar.add(menuFile);                       // 加入JMenu
        newItem.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e){
                text.append("=== 选择了“新建”菜单项\n") ;
            }}) ;                                    // 为菜单项加入事件监听
        openItem.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e){
                text.append("=== 选择了“打开”菜单项\n") ;
            }}) ;                                    // 为菜单项加入事件监听
        closeItem.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e){
                text.append("=== 选择了“关闭”菜单项\n") ;
            }}) ;                                    // 为菜单项加入事件监听
        exitItem.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e){
                text.append("=== 选择了“退出”菜单项\n") ;
            }}) ;                                    // 为菜单项加入事件监听
        frame.addWindowListener(new WindowAdapter() {      // 加入事件监听
            public void windowClosing(WindowEvent arg0) {  // 覆写窗口关闭
                                                              方法
                System.exit(1);                      // 系统退出
            }
        });
        frame.setJMenuBar(menuBar);                  // 在窗体中加入JmenuBar
                                                        组件
        frame.setVisible(true);                      // 显示组件
        frame.setSize(300, 180);                     // 定义窗体大小
        frame.setLocation(300, 200);                 // 设置显示位置
    }
}
```

程序运行结果：



此时是通过鼠标选择菜单项还是使用快捷键其操作结果都会触发事件操作，这点与 JButton 是完全一样的。

18.16 文件选择框：JFileChooser

在使用记事本或者 Word 时，可以通过文件选择框选择要打开或保存的文件，在 Swing 中可以使用 JFileChooser 组件实现此功能。JFileChooser 类的常用方法如表 18-44 所示。

表 18-44 JFileChooser 类的常用方法

序号	方 法	类型	描 述
1	public JFileChooser()	构造	指向用户默认的目录，在 Windows 中是“我的文档”，在 UNIX 上是用户的主目录
2	public JFileChooser(String currentDirectoryPath)	构造	指定文件选择的目录
3	public File getSelectedFile()	普通	返回选择的文件
4	public int showSaveDialog(Component parent) throws HeadlessException	普通	打开保存对话框，返回保存状态
5	public int showOpenDialog(Component parent) throws HeadlessException	普通	打开文件选择对话框，返回打开状态
6	public void setDialogTitle(String dialogTitle)	普通	设置文件选择框的标题
7	public void setApproveButtonText(String approveButtonText)	普通	设置文件选择的接收按钮内容

对于打开文件选择框及文件保存框有 3 种状态返回，这 3 种状态分别对应着 JFileChooser 类中定义的 3 个常量，如表 18-45 所示。

表 18-45 文件选择框的 3 种返回状态

序号	常 量	描 述
1	public static final int APPROVE_OPTION	单击【确定】按钮后返回该值
2	public static final int CANCEL_OPTION	单击【取消】按钮后返回该值
3	public static final int ERROR_OPTION	发生错误后返回该值

下面使用 JFileChooser 完成文件的读取和保存操作，读取时通过文件选择框选择要显示的文件，并将内容显示在文本区中；如果需要保存文件，则通过文件选择框选择文件的保存路径。

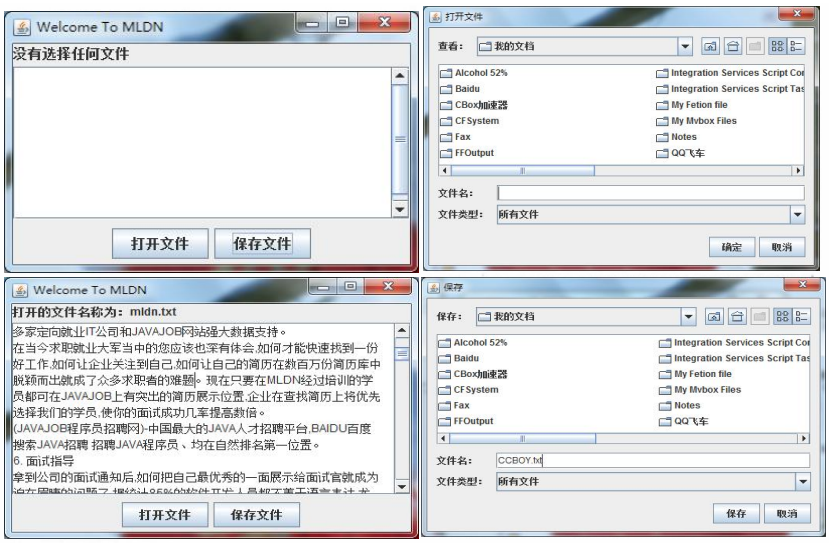
范例：打开和保存文件

```
package org.lxh.demo18.jfilechooserdemo;
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.PrintStream;
import java.util.Scanner;
import javax.swing.JButton;
import javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
class Note implements ActionListener {
    private JTextArea area = new JTextArea(8, 10);;          // 定义文本区
    private JFrame frame = new JFrame("Welcome To MLDN");  // 实例化窗体对象
    private JButton open = new JButton("打开文件");          // 实例化按钮
    private JButton save = new JButton("储存文件");          // 实例化按钮
    private JLabel label = new JLabel("现在没有打开的文件"); // 定义标签，显示
                                                              文件信息
    private JPanel butPan = new JPanel();                    // 定义面板
    public Note() {
        this.butPan.add(open);
        this.butPan.add(save);
        // 设置窗体中的布局管理器为BorderLayout，所有的组件水平和垂直间距为3
        frame.setLayout(new BorderLayout(3, 3));
        frame.add(this.label, BorderLayout.NORTH);           // 组件排列在上方
        frame.add(this.butPan, BorderLayout.SOUTH);          // 组件排列在下方
        frame.add(new JScrollPane(this.area),
                BorderLayout.CENTER);                        // 组件排列在中间
        this.frame.setSize(330, 180);                        // 定义窗体大小
        this.frame.setVisible(true);                         // 显示窗体
        this.frame.addWindowListener(new WindowAdapter() {  // 加入事件监听
            public void windowClosing(WindowEvent arg0) {   // 覆写窗口关闭
                                                              方法
```

```
            System.exit(1);                                    // 系统退出
        }});
    this.open.addActionListener(this);                         // 为按钮加入监听
    this.save.addActionListener(this);                         // 为按钮加入监听
    frame.setSize(300, 200);                                   // 根据组件自动调整窗
                                                                  体大小
    frame.setVisible(true);                                    // 设置窗体可见
}
public void actionPerformed(ActionEvent e) {                   // 按钮事件监听
    File file = null;                                          // 接收打开的文件
    int result = 0;                                            // 接收操作结果
    JFileChooser fileChooser = new JFileChooser();             // 实例化文件选择框
    if (e.getSource() == this.open) {                          // 判断是否是open按钮
        this.area.setText("");                                 // 清空文本区中的原有
                                                                  内容
        fileChooser.setApproveButtonText("确定");              // 定义“确定”按钮
                                                                  信息
        fileChooser.setDialogTitle("打开文件");                // 设置文件选择框标题
        result = fileChooser.showOpenDialog(this.frame);// 显示打开对
                                                                  话框
        if (result == JFileChooser.APPROVE_OPTION) {// 表示选择了确定按钮
            file = fileChooser.getSelectedFile();              // 得到选择的File对象
            this.label.setText("打开的文件名称为：" + file.getName());
                                                               // 修改标签文字
        } else if (result == JFileChooser.CANCEL_OPTION) {
            this.label.setText("没有选择任何文件");            // 修改标签显示文字
        } else {// result == JFileCHooser.ERROR_OPTION
            this.label.setText("操作出现错误");                // 修改标签显示文字
        }
        if (file != null) {                                    // 如果选择的文件不
                                                                  为空
            try {
                Scanner scan = new Scanner(
                        new FileInputStream(file));// 设置输入流
                scan.useDelimiter("\n");                       // 设置换行为分割符
                while (scan.hasNext()) {                       // 循环读取
                    this.area.append(scan.next());             // 读取内容到文本区
                    this.area.append("\n");                    // 设置换行
                }
                scan.close();                                  // 关闭
            } catch (Exception ex) {
```

```
                    this.label.setText("文件读取出错");
                }
            }
        }
        if (e.getSource() == this.save) {                    // 判断是否是save按钮
            result = fileChooser.showSaveDialog(this.frame);// 显示保存文件框
            if (result == JFileChooser.APPROVE_OPTION) {     // 判断选择的是否是保存
                file = fileChooser.getSelectedFile();        // 选择要保存的文件
                this.label.setText("选择的存储文件名称为：" + file.getName());
                                                             // 修改标签文字
            } else if (result == JFileChooser.CANCEL_OPTION) {// 判断是否是取消
                this.label.setText("没有选择任何文件");       // 修改标签显示文字
            } else {// result == JFileCHooser.ERROR_OPTION
                this.label.setText("操作出现错误");           // 修改标签显示文字
            }
            if (file != null) {                              // 判断文件是否为空
                try {
                    PrintStream out = new PrintStream(
                            new FileOutputStream(file));     // 实例化打印流对象
                    out.print(this.area.getText());          // 输出全部内容
                    out.close();                             // 关闭输出流
                } catch (Exception ex) {
                    this.label.setText("文件保存失败");       // 设置错误信息
                }
            }
        }
    }
}
public class JFileChooserDemo {
    public static void main(String[] args) {
        new Note();
    }
}
```

程序运行结果：



程序在执行时，首先通过【打开文件】按钮触发了一个事件，之后在此事件中使用 JFileChooser 打开了一个文件的选择框，并使用 Scanner 类将内容全部显示到文本区中。当单击【储存文件】按钮时会打开保存的文件选择框，可以输入文件名称，之后会通过打印流将全部的内容输出到文件中。

18.17 表　　格

18.17.1 JTable

表格组件提供了以行和列的形式显示数据的视图。在程序开发中是一个非常重要的组件，尤其在需要将一堆数据有条理地展现给用户时，表格设计更能显示出它的重要性。

在 Swing 中可以通过 JTable 组件非常轻松地构造出所需要的表格，并且也提供了一些方法来管理这些表格的内容。JTable 类的常用方法如表 18-46 所示。

表 18-46　JTable 类的常用方法

序号	方　　法	类型	描　　述
1	public JTable(Object[][] rowData,Object[] columnNames)	构造	创建一个 JTable 对象，设置显示数据和表格的标题
2	public JTable(Vector rowData,Vector columnNames)	构造	创建一个 JTable 对象，通过 Vector 设置显示数据和显示的标题
3	public JTable(TableModel dm)	构造	使用 TableModel 创建表格
4	public TableColumnModel getColumnModel()	普通	返回 TableColumnModel 对象

范例：建立表格

```
package org.lxh.demo18.jtabledemo;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
```

```
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTable;
public class JTableDemo01 {
    public static void main(String args[]) {
        JFrame frame = new JFrame("Welcome To MLDN");    // 实例化窗体对象
        String[] titles = { "姓名", "年龄", "性别",
                "数学成绩","英语成绩","总分","是否及格" };   // 定义数组表示表格
                                                            标题
        Object[][] userInfo = { { "李兴华", 30, "男", 89, 97, 186, true },
                { "李康", 23, "女", 90, 93, 183, false } };// 定义二维对象数组
                                                            表示每行的数据
        JTable table = new JTable(userInfo, titles);     // 实例化JTable组件
        JScrollPane scr = new JScrollPane(table);        // 加入滚动条
        frame.add(scr);                                  // 将表格加入到窗体
        frame.setSize(370, 90);                          // 定义窗体大小
        frame.setVisible(true);                          // 显示窗体
        frame.addWindowListener(new WindowAdapter() {    // 加入事件监听
            public void windowClosing(WindowEvent arg0){// 覆写窗口关闭方法
                System.exit(1);                          // 系统退出
            }});
    }
}
```

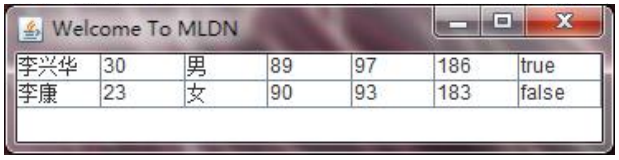
程序运行结果：



在以上程序中使用一个二维数组表示表格中的数据，表格中的数据按照标题栏设置的顺序进行排列，之后将表格加入到一个滚动框中，再将滚动框加入到面板上。

注意：JTable 使用时要加入到 JScrollPane 中。

在使用 JTable 时，如果不将一个 JTable 加入到 JScrollPane 中，则表格的标题将无法显示。



18.17.2　使用 TableModel 构建表格

使用 JTable 构建的表格相对比较单一，如果想制作一些操作界面更加友好的表格，例如在表格上加入一些单选按钮或下拉列表框，则就要借助于 TableModel 接口。在此接口中

定义了许多与表格操作相关的方法，常用的方法如表 18-47 所示。

表 18-47 TableModel 接口的常用方法

序号	方 法	类型	描 述
1	public Class<?> getColumnClass(int columnIndex)	普通	得到表格中每一列的数据类型
2	public int getColumnCount()	普通	返回表格中的列数
3	public String getColumnName(int columnIndex)	普通	返回表格中列的名字
4	public int getRowCount()	普通	返回表格中的行数
5	public Object getValueAt(int rowIndex,int columnIndex)	普通	根据行和列取得指定位置的元素
6	public boolean isCellEditable(int rowIndex,int columnIndex)	普通	返回单元格是否可编辑
7	public void setValueAt(Object aValue,int rowIndex,int columnIndex)	普通	设置表格内容

在一般的开发中很少直接实现以上的接口，而都使用其接口的子类 AbstractTableModel 和 DefaultTableModel。但是不管使用哪个子类实现，首先必须了解的是 TableColumnModel 接口，此接口定义了许多与表格的行或列有关的方法，例如，增加行（列）、删除行（列），设置与取得表格元素的信息等操作。通常用户不会直接实现 TableColumnModel 接口，而是会通过 JTable 中的 getColumnModel()方法取得 TableColumnModel 的实例。

TableColumnModel 接口中也同样定义了大量的方法，其常用方法如表 18-48 所示。

表 18-48 TableColumnModel 接口的常用方法

序号	方 法	类型	描 述
1	public void addColumn(TableColumn aColumn)	普通	增加列
2	public void removeColumn(TableColumn column)	普通	删除指定列
3	public TableColumn getColumn(int columnIndex)	普通	根据索引取得指定列
4	public int getColumnCount()	普通	取得全部列数
5	public Enumeration<TableColumn> getColumns()	普通	返回全部列

在此接口的操作中读者可以发现，增加或者删除都是以 TableColumn 类的形式出现的，此类表示每一列的数据，这一点可以在随后的开发中见到。

下面就先使用 AbstractTableModel 类为读者建立一个表格，建立一个 DefaultTable 类直接继承 AbstractTableModel 类，之后在里面覆写相应的方法即可。

范例：使用 AbstractTableModel 构建表格

```
package org.lxh.demo18.jtabledemo;
import java.awt.BorderLayout;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.DefaultCellEditor;
import javax.swing.JComboBox;
import javax.swing.JFrame;
```

```
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.table.AbstractTableModel;
class DefaultTable extends AbstractTableModel {
    private String[] titles = { "姓名", "年龄", "性别", "数学成绩",
            "英语成绩", "总分", "是否及格" };                    // 定义数组表示表格标题
    private Object[][] userInfo = { { "李兴华", 30, "男", 89, 97, 186, true },
            { "李康", 23, "女", 90, 93, 183, false } };// 定义二维对象数组表示
                                                                数据
    public int getColumnCount() {                               // 返回列的个数
        return this.titles.length;
    }
    public int getRowCount() {                                  // 返回表格的行数
        return this.userInfo.length;
    }
    public Object getValueAt(int row, int col) {                // 返回指定行和列的数据
        return this.userInfo[row][col];
    }
    // 得到列的名字，如果不覆写此方法，则以后无法显示列的名称
    public String getColumnName(int col) {
        return this.titles[col];                                // 根据下标返回指定列的
                                                                名字
    }
    /*
     * 返回列的类型，如果不覆写此方法，则无法按格式显示数据，格式如下：
     * 1、boolean类型：以CheckBox方式显示
     * 2、数值类型：以JLabel显示，文字向右排列
     */
    public Class<?> getColumnClass(int col) {
        return this.getValueAt(0, col).getClass();              // 取得指定列的类型
    }
    // 如果不覆写此方法，则表格无法编辑
    public boolean isCellEditable(int row, int col) {
        return true;                                            // 所有表格单元均可编辑
    }
    public void setValueAt(Object newValue, int row, int col) {// 修改表格
                                                                数据
        this.userInfo[row][col] = newValue;                     // 修改数据
    }
}
class TableColumnModelDemo {
```

```
    private JFrame frame = new JFrame("Welcome To MLDN");// 实例化窗体对象
    private JTable table = null;                          // 声明JTable对象
    private DefaultTable defaultTable = new DefaultTable();// 实例化
    private JComboBox sexList = new JComboBox();          // 定义下拉列表框
    public TableColumnModelDemo() {                       // 定义构造方法，完成
                                                          显示
        this.table = new JTable(this.defaultTable);       // 实例化JTable对象
        this.sexList.addItem("男");                       // 增加下拉选项
        this.sexList.addItem("女");                       // 增加下拉选项
        this.table.getColumnModel().getColumn(2).setCellEditor(
                new DefaultCellEditor(this.sexList));     // 将下拉列表选项加入
                                                          到表格
        JScrollPane scr = new JScrollPane(this.table);    // 加入滚动条
        JPanel toolBar = new JPanel();                    // 定义面板
        this.frame.add(toolBar, BorderLayout.NORTH);      // 将面板加入到窗体
        this.frame.add(scr, BorderLayout.CENTER);         // 将表格加入到窗体
        this.frame.setSize(370, 160);                     // 定义窗体大小
        this.frame.setVisible(true);                      // 显示窗体
        this.frame.addWindowListener(new WindowAdapter() {
                                                          // 加入事件监听
            public void windowClosing(WindowEvent arg0) {
                                                          // 覆写窗口关闭方法
                System.exit(1);                           // 系统退出
            }});
    }
}
public class JTableDemo02 {
    public static void main(String args[]) {
        new TableColumnModelDemo();
    }
}
```

程序运行结果：



从以上程序运行结果来看，表格中的数据都是按照指定格式进行显示的，而且因为在 DefaultTable 类中覆写了 isCellEditable()方法，所以表格的每一项都是可编辑的。对于性别的选项使用了一个下拉列表框的形式进行显示，而且在“是否及格”列上，是使用一个 CheckBox 来代替了原有的 true 及 false。

使用 AbstractTableModel 类可以方便地构建表格，而使用其子类 DefaultTableModel 则可以对表格进行动态的操作，例如，增加行（列）、删除行（列）等。在 DefaultTableModel 类中提供了表 18-49 所示的常用操作方法。

表 18-49　DefaultTableModel 类的常用方法

序号	方　法	类型	描　述
1	public void addColumn(Object columnName)	普通	增加列，并指定列名称
2	public void addRow(Object[] rowData)	普通	增加行
3	public int getRowCount()	普通	返回表格的行数
4	public int getColumnCount()	普通	返回表格的列数
5	public void setRowCount(int rowCount)	普通	设置表格中的行数
6	public void setColumnCount(int columnCount)	普通	设置表格中的列数
7	public void removeRow(int row)	普通	删除表格中的指定行

范例：动态操作表格

```
package org.lxh.demo18.jtabledemo;
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.table.DefaultTableModel;
import javax.swing.table.TableColumn;
import javax.swing.table.TableColumnModel;
class ChangeTable implements ActionListener {                    // 实现事件监听
    private JFrame frame = new JFrame("Welcome To MLDN");// 实例化窗体对象
    private JTable table = null;                                  // 声明JTable对象
    private DefaultTableModel tableModel = null;                  // 声明Default
                                                                     TableModel对象
    private String[] titles = { "姓名", "年龄", "性别", "数学成绩",
            "英语成绩", "总分", "是否及格" };                        // 定义数组表示表格
                                                                     标题
    private Object[][] userInfo = { { "李兴华", 30, "男", 89, 97, 186, true },
            { "李康", 23, "女", 90, 93, 183, false } }; // 定义二维对象数组表
                                                                     示数据
    private JButton addRowBtn = new JButton("增加行");  // 定义按钮
    private JButton removeRowBtn = new JButton("删除行");// 定义按钮
```

```
    private JButton addColBtn = new JButton("增加列");  // 定义按钮
    private JButton removeColBtn = new JButton("删除列"); // 定义按钮
    public ChangeTable() {                              // 定义构造方法，完成
                                                           显示
        // 实例化DefaultTableModel对象，显示内容
        this.tableModel = new DefaultTableModel(this.userInfo, this.titles);
        this.table = new JTable(this.tableModel);       // 根据TableModel显
                                                           示内容
        JScrollPane scr = new JScrollPane(this.table); // 加入滚动条
        JPanel toolBar = new JPanel();                  // 定义面板
        toolBar.add(this.addRowBtn);                    // 向面板增加按钮
        toolBar.add(this.removeRowBtn);                 // 向面板增加按钮
        toolBar.add(this.addColBtn);                    // 向面板增加按钮
        toolBar.add(this.removeColBtn);                 // 向面板增加按钮
        this.frame.add(toolBar, BorderLayout.NORTH);    // 将面板加入到窗体
        this.frame.add(scr, BorderLayout.CENTER);       // 将表格加入到窗体
        this.frame.setSize(370, 160);                   // 定义窗体大小
        this.frame.setVisible(true);                    // 显示窗体
        this.addRowBtn.addActionListener(this);         // 为按钮增加监听
        this.removeRowBtn.addActionListener(this);      // 为按钮增加监听
        this.addColBtn.addActionListener(this);         // 为按钮增加监听
        this.removeColBtn.addActionListener(this);      // 为按钮增加监听
        this.frame.addWindowListener(new WindowAdapter(){// 加入事件监听
            public void windowClosing(WindowEvent arg0){// 覆写窗口关闭方法
                System.exit(1);                         // 系统退出
            }});
    }
    public void actionPerformed(ActionEvent e) {        // 实现事件操作
        if (e.getSource() == this.addColBtn) {          // 判断触发事件按钮
            this.tableModel.addColumn("新增列");        // 增加列
        }
        if (e.getSource() == this.addRowBtn) {          // 判断触发事件按钮
            this.tableModel.addRow(new Object[] {});    // 增加行
        }
        if (e.getSource() == this.removeColBtn) {       // 判断触发事件按钮
            int colCount = this.tableModel.getColumnCount();// 得到最后一列
            if (colCount >= 0) {                        // 判断是否还有列
                // 取得TableColumnModel实例，以方便删除列
                TableColumnModel columnMode = this.table.getColumnModel();
                // 取得删除列的对象，根据列的编号取得TableColumn对象
                TableColumn tableColumn = columnMode.getColumn(colCount);
                columnMode.removeColumn(tableColumn);   // 删除列
```

```
                    // 修改表格的列数，如果不修改则在下次再删除列时将出现数组越界异常
                    this.tableModel.setColumnCount(colCount);
                }
            }
            if (e.getSource() == this.removeRowBtn) {       // 判断触发事件按钮
                int rowCount = this.tableModel.getRowCount() - 1;
                                                            // 得到最后一行
                if (rowCount >= 0) {                        // 判断是否还有行
                    this.tableModel.removeRow(rowCount);    // 删除行
                    this.tableModel.setRowCount(rowCount); // 设置行数
                }
            }
        }
    }
    public class JTableDemo03 {
        public static void main(String args[]) {
            new ChangeTable();
        }
    }
```

程序运行结果：



在本程序中，使用 DefaultTableModel 可以方便地实现增加行（列）、删除行（列）的各种操作。在删除行前先取得最后一行的编号，之后通过 removeRow()方法根据行号删除掉指定的行，但是在行删除之后一定要使用 setRowCount()方法重新设置行数，这样在下次操作时就不会出现数组越界的异常信息。

18.18　本 章 要 点

1．Swing 是在 AWT 基础上的一种扩展应用，提供了一套轻量级的操作组件。

2．在 Swing 中所有的组件都是以字母 J 开头的，所有的组件都是继承自 Component 类。

3．JFrame 是 Swing 提供的一个标准容器，可以向该容器上加入各个组件。

4．JLabel 是一个标签组件，用于显示基本信息。

5．JButton 是一个按钮组件，如果要构建不弹起的按钮则要使用 JToggleButton。

6．在图形界面中主要提供了 FlowLayout、BorderLayout、GridLayout、CardLayout、绝对定位 5 种布局管理器。

7．可以使用 JTextField 完成一个文本框，使用 JPasswordField 完成一个密码框，使用 JTextArea 完成一个文本域。

8．事件发生时会产生事件源并找到相应的监听程序进行处理，在图形界面中提供了多个 Listener 接口进行事件的处理。

9．在使用 JRadioButton 完成单选操作时，需要将多个 JRadioButton 加入到 ButtonGroup 中，才能实现单选的功能。

10．JComboBox 可以完成下拉列表框的操作，使用 JList 可以完成列表框的显示。

11．要完成菜单显示，需要将多个 JMenu 加入到 JMenuBar 中，菜单中的菜单项使用 JMenuItem 处理，每一个 JMenuItem 都表示一个特殊的按钮。

12．JFileChooser 可以完成文件选择的对话框。

13．如果要显示一组数据，则可以使用表格的形式进行显示。

18.19 习　　题

1．编写程序，在文本框中输入一个浮点型华式温度，根据下面的公式将其转化成摄氏温度值并输出：

摄氏温度=（华氏-32）*5/9

2．改善登录程序，使之可以完成基于数据库的登录操作。数据库表由用户自行设计。

3．定义一张数据库表，其中包含编号（自动增长）、姓名、年龄、生日、学校，并将全部的信息通过 JTable 显示在图形界面中。

4．编写程序，在文本框中输入英文字母，再根据“大写字母按钮”或“小写字母按钮”将输入的内容进行转换，并将转换后的结果显示在标签上。

5．模仿 Windows 的记事本（notepad.exe）程序完成以下要求：

（1）文件的打开和保存。

（2）设置显示字体的大小。

6．模仿 Windows 计算器程序完成一个可以用于执行+、-、*、/的操作。

第 19 章　Java 网络编程

通过本章的学习可以达到以下目标：

- ☑ 了解 IP 地址与 InetAddress 类的关系。
- ☑ 了解如何使用 URL 定位网络资源。
- ☑ 了解编码和解码的操作。
- ☑ 了解 ServerSocket 类与 Socket 类的关系以及客户端与服务器端的通信模式。
- ☑ 了解如何将多线程机制应用在服务器开发上。
- ☑ 了解 UDP 程序与 TCP 程序的实现区别。

网络可以使不同物理位置上的计算机达到资源共享和通信的目的，在 Java 中也提供了专门的网络开发程序包——java.net，以方便开发者进行网络程序的开发。

Java 的网络编程提供了两种通信协议：TCP（传输控制协议）和 UDP（数据报协议）。本章将就这两种实现进行讲解，在讲解中也将融合之前讲解的 IO 及多线程技术进行网络程序的开发。**本章视频录像讲解时间为 1 小时 08 分钟，源代码在光盘对应的章节下。**

提示：关于 TCP 及 UDP 协议。

TCP 和 UDP 都属于传输层协议，TCP（Transmission Control Protocol）是可靠的传输协议，传输前会采用“三方握手”的方式建立连接，以保证传输的可靠性；而 UDP（User Datagram Protocol）协议是不可靠的传输协议，即发送出去的数据不一定接收得到，网上的聊天工具一般采用此种协议。

19.1　IP（Internet Protocol）与 InetAddress

19.1.1　IP 地址（Internet Protocol）简介

互联网上的每一台计算机都有一个唯一表示自己的标记，这个标记就是 IP 地址。在 Windows 操作系统中，用户可以通过图 19-1 所示的设置界面方便地设置每一台电脑的 IP 地址。

IP 地址使用 32 位长度二进制数据表示，一般在实际中看到的大部分 IP 地址都是以十进制的数据形式表示的，如 192.168.1.3。

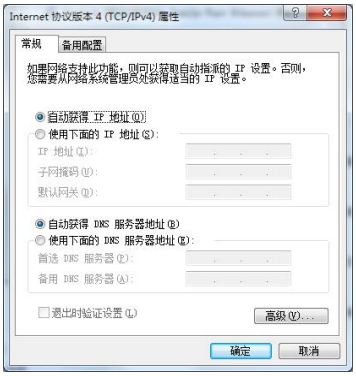


图 19-1　IP 地址设置

【格式 19-1　IP 地址格式】

IP地址=网络地址+主机地址

- 网络号：用于识别主机所在的网络。
- 主机号：用于识别该网络中的主机。

IP 地址分为 5 类，A 类保留给政府机构，B 类分配给中等规模的公司，C 类分配给任何需要的人，D 类用于组播，E 类用于实验，各类可容纳的地址数目不同。这 5 类地址的范围如表 19-1 所示。

表 19-1　IP 地址的范围

序　号	地 址 分 类	地 址 范 围
1	A 类地址	1.0.0.1～126.255.255.254
2	B 类地址	128.0.0.1～191.255.255.254
3	C 类地址	192.0.0.1～223.255.255.254
4	D 类地址	224.0.0.1～239.255.255.254
5	E 类地址	240.0.0.1～255.255.255.254

在以上的地址分类中可以发现没有 127.X.X.X 的表示，因为其是保留地址，用作循环测试，在开发中经常使用 127.0.0.1 表示本机的 IP 地址，关于这点读者在随后的开发中可以见到。

提示：IP 地址有 IPv4 和 IPv6 两类。

IPv4（Internet Protocol version 4）是互联网协议的第 4 个版本，也是使用最广泛的版本。但是 IPv4 已经无法满足当今互联网上的主机数量，所以在此基础上又产生了新的版本 IPv6，使用 IPv6 可以比 IPv4 容纳更多的主机。

19.1.2　InetAddress

InetAddress 类主要表示 IP 地址，这个类有两个子类：Inet4Address、Inet6Address，一个用于表示 IPv4，另一个用于表示 IPv6。InetAddress 类的常用方法如表 19-2 所示。

表 19-2　InetAddress 类的常用方法

序号	方　法	类型	描　述
1	public static InetAddress getByName(String host) throws UnknownHostException	普通	通过主机名称得到 InetAddress 对象
2	public static InetAddress getLocalHost() throws UnknownHostException	普通	通过本机得到 InetAddress 对象
3	public String getHostName()	普通	得到 IP 地址
4	public boolean isReachable(int timeout) throws IOException	普通	判断地址是否可以到达，同时指定超时时间

范例：测试 InetAddress 类

```
package org.lxh.demo19.inetaddressdemo;
import java.net.InetAddress;
```

```
public class InetAddressDemo {
    public static void main(String[] args) throws Exception {  // 所有异常抛出
        InetAddress locAdd = null;                              // 声明InetAddress对象
        InetAddress remAdd = null;                              // 声明InetAddress对象
        locAdd = InetAddress.getLocalHost();                    // 得到本地InetAddress对象
        remAdd = InetAddress.getByName("www.mldnjava.cn");      // 取得远程InetAddress
        System.out.println("本机IP地址："
                + locAdd.getHostAddress());                     // 得到本地IP地址
        System.out.println("MLDNJAVA的IP地址："
                + remAdd.getHostAddress());                     // 得到远程IP地址
        System.out.println("本机是否可达："
                + locAdd.isReachable(5000));
    }
}
```

程序运行结果：

```
本机IP地址：169.254.109.88
MLDNJAVA的IP地址：119.161.130.55
本机是否可达：true
```

19.2　URL 与 URLConnection

19.2.1　URL

URL（Uniform Resource Locator）统一资源定位符，可以直接使用此类找到互联网上的资源（如一个简单的网页）。URL 类的常用方法如表 19-3 所示。

表 19-3　URL 类的常用方法

序号	方　法	类型	描　述
1	public URL(String spec) throws MalformedURLException	构造	根据指定的地址实例化 URL 对象
2	public URL(String protocol,String host,int port,String file) throws MalformedURLException	构造	实例化 URL 对象，并指定协议、主机、端口名称、资源文件
3	public URLConnection openConnection() throws IOException	普通	取得一个 URLConnection 对象
4	public final InputStream openStream() throws IOException	普通	取得输入流

下面通过一个范例来观察如何使用 URL，本范例将根据指定的 URL 对资源使用 InputStream 进行读取，读取的 URL 地址是 http://www.mldnjava.cn/curriculum.htm。

范例：使用 URL 读取内容

```
package org.lxh.demo19.urldemo;
import java.io.InputStream;
import java.net.URL;
import java.util.Scanner;
public class URLDemo {
    public static void main(String[] args) throws Exception {
                                                    // 所有异常抛出
        URL url = new URL("http", "www.mldnjava.cn",
                80, "/curriculum.htm");             // 指定操作的URL
        InputStream input = url.openStream();       // 打开输入流，读取URL内容
        Scanner scan = new Scanner(input);          // 实例化Scanner对象
        scan.useDelimiter("\n");                    // 设置读取分隔符
        while (scan.hasNext()) {                    // 不断读取内容
            System.out.println(scan.next());        // 输出内容
        }
    }
}
```

以上程序运行时，使用 URL 找到了指定主机上的 curriculum.htm 页面资源，并使用 Scanner 将页面中的内容下载下来直接显示在屏幕上。

提示：显示出来的内容是 HTML 代码。

在以上程序中下载下来的内容全部都是页面的 HTML 代码，HTML（HyperText Mark-up Language）即超文本标记语言或超文本链接标示语言，是目前网络上应用最为广泛的语言，也是构成网页文档的主要语言。

19.2.2 URLConnection

URLConnection 是封装访问远程网络资源一般方法的类，通过它可以建立与远程服务器的连接，检查远程资源的一些属性。此类的常用方法如表 19-4 所示。

表 19-4 URLConnection 类的常用方法

序号	方　法	类型	描　述
1	public int getContentLength()	普通	取得内容的长度
2	public String getContentType()	普通	取得内容的类型
3	public InputStream getInputStream() throws IOException	普通	取得连接的输入流

URLConnection 对象可以通过 URL 类的 openConnection()方法取得，下面通过 URLConnection 对象取得一个 URL 的基本信息。

范例：取得 URL 的基本信息

```
package org.lxh.demo19.urldemo;
import java.net.URL;
```

```
import java.net.URLConnection;
public class URLConnectionDemo {
    public static void main(String[] args) throws Exception{// 所有异常抛出
        URL url = new URL("http://www.mldnjava.cn");          // 指定操作的URL
        URLConnection urlCon = url.openConnection() ;         // 建立连接
        System.out.println("内容大小：" + urlCon.getContentLength());
                                                              // 取得内容大小
        System.out.println("内容类型：" + urlCon.getContentType());
                                                              // 取得内容类型
    }
}
```

程序运行结果：

```
内容大小：21496
内容类型：text/html
```

19.3　URLEncoder 与 URLDecoder

在使用 URL 访问时，经常会看到在地址后会有很多其他的附带信息，例如，在百度网上搜索“mldn 李兴华”时在地址栏后就会附带有很多其他的信息，如图 19-2 所示。

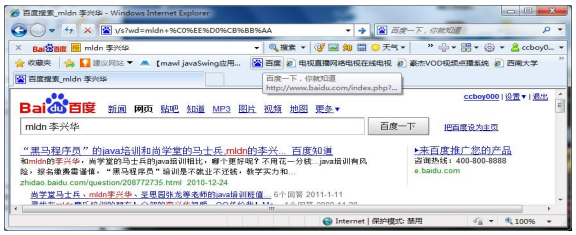


图 19-2　地址的附带信息

从这些地址的附带信息上可以发现英文单词可以正常显示，但是对于中文，则会将其进行一系列的编码操作，在 Java 中如果要完成这样的编码和解码操作就必须使用 URLEncoder 和 URLDecoder 两个类。URLEncoder 可以为传递的内容进行编码，而 URLDecoder 可以为传递的内容进行解码。这两个类的常用方法分别如表 19-5 和表 19-6 所示。

表 19-5　URLEncoder 类的常用方法

序号	方　　法	类型	描　　述
1	public static String encode(String s,String enc) throws UnsupportedEncodingException	普通	使用指定的编码机制将字符串转换为 application/x-www-form-urlencoded 格式

表 19-6　URLDecoder 类的常用方法

序号	方　　法	类型	描　　述
1	public static String decode(String s,String enc) throws UnsupportedEncodingException	普通	使用指定的编码机制对 application/x-www-form-urlencoded 字符串解码

范例：编码及解码操作

```
package org.lxh.demo19.codedemo;
import java.net.URLDecoder;
import java.net.URLEncoder;
public class CodeDemo {
    public static void main(String[] args) throws Exception {
                                                                    // 所有异常抛出
        String keyWord = "mldn 李兴华";                              // 要编码的内容
        String encod = URLEncoder.encode(keyWord, "UTF-8");// 对内容进行编码
        System.out.println("编码之后的内容：" + encod);
        String decod = URLDecoder.decode(encod, "UTF-8");  // 对内容进行解码
        System.out.println("解码之后的内容：" + decod);
    }
}
```

程序运行结果：

```
编码之后的内容：mldn+%E6%9D%8E%E5%85%B4%E5%8D%8E
解码之后的内容：mldn 李兴华
```

以上程序将内容通过 URLEncoder 编码成 UTF-8 的形式，之后再通过 URLDecoder 按照 UTF-8 进行解码。

19.4 TCP 程序设计

在 Java 中使用 Socket（即套接字）完成 TCP 程序的开发，使用此类可以方便地建立可靠的、双向的、持续的、点对点的通信连接。

在 Socket 的程序开发中，服务器端使用 ServerSocket 等待客户端的连接，对于 Java 的网络程序来讲，每一个客户端都使用一个 Socket 对象表示，如图 19-3 所示。

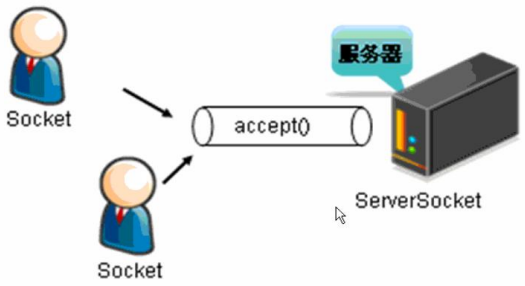


图 19-3　Socket 与 ServerSocket

19.4.1 ServerSocket 类与 Socket 类

ServerSocket 类主要用在服务器端程序的开发上，用于接收客户端的连接请求。ServerSocket 类的常用方法如表 19-7 所示。

表 19-7　ServerSocket 类的常用方法

序号	方　法	类型	描　述
1	public ServerSocket(int port) throws IOException	构造	创建 ServerSocket 实例，并指定监听端口
2	public Socket accept() throws IOException	普通	等待客户端连接，此方法连接之前一直阻塞
3	public InetAddress getInetAddress()	普通	返回服务器的 IP 地址
4	public boolean isClosed()	普通	返回 ServerSocket 的关闭状态
5	public void close() throws IOException	普通	关闭 ServerSocket

在服务器端每次运行时都要使用 accept()方法等待客户端连接，此方法执行之后服务器端将进入阻塞状态，直到客户端连接之后程序才可以向下继续执行。此方法的返回值类型是 Socket，每一个 Socket 都表示一个客户端对象。Socket 类的常用方法如表 19-8 所示。

表 19-8　Socket 类的常用方法

序号	方　法	类型	描　述
1	public Socket(String host,int port) throws UnknownHostException,IOException	构造	构造 Socket 对象，同时指定要连接服务器的主机名称及连接端口
2	public InputStream getInputStream() throws IOException	普通	返回此套接字的输入流
3	public OutputStream getOutputStream() throws IOException	普通	返回此套接字的输出流
4	public void close() throws IOException	普通	关闭此 Socket
5	public boolean isClosed()	普通	判断此套接字是否被关闭

在客户端，程序可以通过 Socket 类的 getInputStream()方法取得服务器的输出信息，在服务器端可以通过 getOutputStream()方法取得客户端的输出信息，如图 19-4 所示。

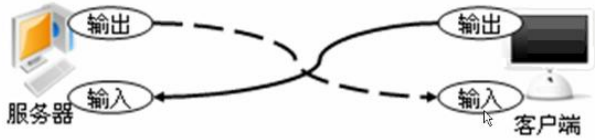


图 19-4　服务器与客户端的通信

从图 19-4 可以看出，在网络程序中要使用输入及输出流的形式完成信息的传递，所以在开发时需要导入 java.io 包。

19.4.2　第一个 TCP 程序

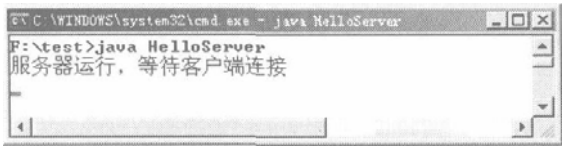
下面通过 ServerSocket 类及 Socket 类完成一个服务器的程序开发，此服务器向客户端输出“hello world!”的字符串信息。

范例：建立服务器程序

```
import java.net.*;
import java.io.*;
public class HelloServer {
    public static void main(String args[]) throws Exception {  // 所有异常
                                                                抛出
```

```
        ServerSocket server = null;          // 声明ServerSocket对象
        Socket client = null;                // 一个Socket对象表示一个客户端
        PrintStream out = null;              // 声明打印流对象，以向客户端输出
        server = new ServerSocket(8888);     // 此时服务器在8888端口上等待客户
                                                端的访问
        System.out.println("服务器运行，等待客户端连接");
        client = server.accept();            // 程序阻塞，等待客户端连接
        String str = "hello world!!!";       // 要向客户端输出的信息
        out = new PrintStream(client.getOutputStream()); // 实例化打印流对象，
                                                            输出信息
        out.println(str);                    // 输出信息
        out.close();                         // 关闭打印流
        client.close();                      // 关闭客户端连接
        server.close();                      // 关闭服务器连接
    }
};
```

程序运行结果：



从程序的运行结果中可以发现，服务器程序一执行到 accept()方法后，程序进入到阻塞状态，此阻塞状态会在客户端连接之后改变。

提示：可以使用 telnet 命令进行验证。

服务器端程序建立完成之后，因为其是以 TCP 为通信协议的，所以可以直接使用 telnet 命令进行连接测试，输入“open localhost 8888”即可取得服务器的输出信息。

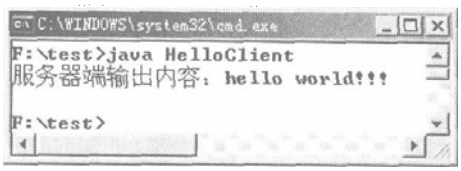
服务器程序编写完成之后，下面介绍如何通过客户端访问服务器，直接使用 Socket 类指定连接的地址及端口号，并通过输入流读取服务器的输出信息。

范例：编写客户端程序

```
import java.net.*;
import java.io.*;
public class HelloClient {
    public static void main(String artsg[]) throws Exception {
        Socket client = null;                          // 声明Socket对象
        client = new Socket("localhost", 8888);        // 指定连接的主机和端口
        BufferedReader buf = null;                     // 声明BufferedReader对
                                                          象，接收信息
        buf = new BufferedReader(
                new InputStreamReader(
                        client.getInputStream()));     // 取得客户端的输入流
```

```
            String str = buf.readLine();                    // 读取信息
            System.out.println("服务器端输出内容：" + str);
            client.close();                                 // 关闭Socket
            buf.close();                                    // 关闭输入流
        }
    };
```

程序运行结果（重新打开一个命令行窗口）：



此时客户端从服务器端将信息读取进来，读取完毕后，因为服务器端此时只能处理一次连接请求，所以也将关闭。

19.4.3　案例：Echo 程序

Echo 程序是一个网络编程通信交互的一个经典案例，称为回应程序，即客户端输入哪些内容，服务器端会在这些内容前加上“ECHO：”并将信息发回给客户端，下面就完成这样的一个程序。

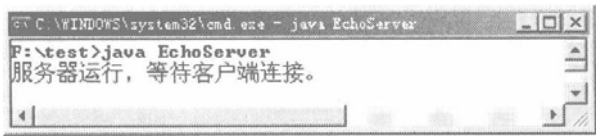
之前的程序代码，服务器端每次执行完毕后服务器都会退出，这是因为服务器端只能接收一个客户端的连接，主要是由于 accept()方法只能使用一次。本程序中将通过循环的方式使用 accept()，这样，每一个客户端执行完毕后，服务器端都可以重新等待用户连接。

范例：EchoServer

```
import java.net.*;
import java.io.*;
public class EchoServer {
    public static void main(String args[]) throws Exception {  // 所有异常
                                                                抛出
        ServerSocket server = null;                 // 定义ServerSocket对象
        Socket client = null;                       // 定义Socket对象，表示客户端
        PrintStream out = null;                     // 定义输出流
        BufferedReader buf = null;                  // 用于接收客户端发送来的信息
        server = new ServerSocket(8888);            // 此服务器在8888端口上进行监听
        boolean f = true;                           // 定义一个标记位
        while (f) {                                 // 无限制接收客户端连接
            System.out.println("服务器运行，等待客户端连接。") ;
            client = server.accept();               // 接收客户端连接
            buf = new BufferedReader(
                    new InputStreamReader(client
                    .getInputStream()));            // 得到客户端的输入信息
```

```
            out = new PrintStream(
                    client.getOutputStream());          // 实例化客户端的输出流
            boolean flag = true;                        // 标志位，表示一个客户端是
                                                           否操作完毕
            while (flag) {                              // 客户端循环操作
                String str = buf.readLine();            // 在此处不断地接收信息
                if (str == null || "".equals(str)) {// 判断输入的信息是否为空
                    flag = false;                       // 客户端操作结束
                } else {
                    if ("bye".equals(str)) {            // 如果输入信息为bye表示
                                                           结束
                        flag = false;                   // 客户端操作结束
                    } else {
                        out.println("ECHO:" + str);// 向客户端回显信息
                    }
                }
            }
            out.close();                                // 关闭输出流
            client.close();                             // 关闭客户端
        }
        server.close();                                 // 关闭服务器端
    }
};
```

程序运行结果：



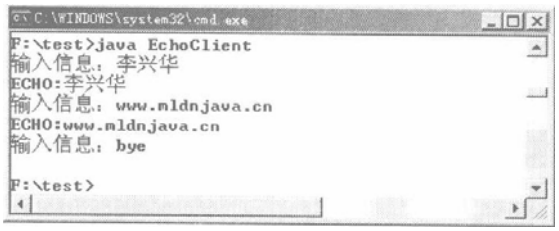
服务器运行之后，和之前一样，要等待客户端的连接，下面来编写客户端程序。

范例：EchoClient

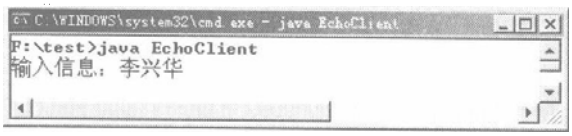
```
import java.net.*;
import java.io.*;
public class EchoClient {
    public static void main(String artsg[]) throws Exception {// 所有异常
                                                                  抛出
        Socket client = null;                       // 声明Socket对象
        client = new Socket("localhost", 8888);// 指定连接主机及端口
        BufferedReader buf = null;                  // 接收服务器端发送回来的信息
        PrintStream out = null;                     // 输出流，向服务器端发送信息
        BufferedReader input = null;                // 声明BufferedReader对象
        input = new BufferedReader(
                new InputStreamReader(System.in)); // 从键盘接收数据
```

```
            out = new PrintStream(
                    client.getOutputStream());          // 向服务器端输出信息
            buf = new BufferedReader(
                    new InputStreamReader(
                            client.getInputStream()));  // 接收服务器端输入信息
            boolean flag = true;                        // 定义标志位
            while (flag) {                              // 不断发送和接收数据
                System.out.print("输入信息：") ;
                String str = input.readLine();          // 从键盘接收数据
                out.println(str);                       // 向服务器端输入信息
                if ("bye".equals(str)) {                // 如果输入信息为bye表示
                                                           结束
                    flag = false;                       // 客户端操作结束
                }else{
                    String echo = buf.readLine() ;      // 接收ECHO信息
                    System.out.println(echo);           // 输出ECHO信息
                }
            }
            client.close();                             // 关闭Socket
            buf.close();                                // 关闭输入流
        }
    };
```

程序运行结果（重新打开一个命令行窗口）：



从程序的运行结果中可以发现，所有的输入信息最终都会通过回显的方式发回给客户端，并且前面加上了“ECHO”的信息。另外在本程序中，当一个客户端结束之后，服务器端并不会退出，会等待下一个用户连接，继续执行。但在本程序中同时存在一个严重的问题，即现在的服务器端每次只能有一个用户连接，属于单线程的处理机制，如果现在有其他用户连接会出现以下的问题：



因为已经有了一个程序连接到服务器端，所以此时其他客户端是无法连接的，要等待服务器出现空闲才可以连接。为了保证服务器可以同时连接多个客户端，可以加入多线程机制，即每一个客户端连接之后都启动一个线程，这样一个服务器就可以同时支持多个客户端连接，如图 19-5 所示。

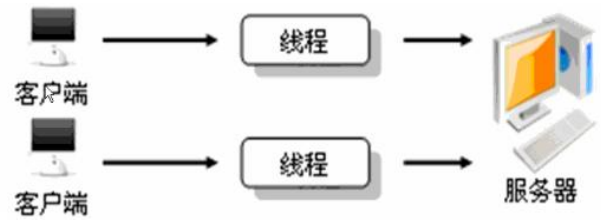


图 19-5　加入多线程

19.4.4　在服务器上应用多线程

对于服务器端来说，如果要加入多线程机制，则应该在每个用户连接之后启动一个新的线程。所以下面先建立一个 EchoThread 类，此类专门用于处理多线程操作，此时的多线程使用 Runnable 接口实现。

范例：EchoThread

```
import java.io.*;
import java.net.*;
public class EchoThread implements Runnable {          // 实现Runnable接口
    private Socket client = null;                      // 接收客户端
    public EchoThread(Socket client) {                 // 通过构造方法设置Socket
        this.client = client;
    }
    public void run() {                                // 覆写run()方法
        PrintStream out = null;                        // 定义输出流
        BufferedReader buf = null;                     // 用于接收客户端发送来的
                                                       信息
        try {
            buf = new BufferedReader(
                    new InputStreamReader(client
                    .getInputStream()));               // 得到客户端的输入信息
            out = new PrintStream(
                    client.getOutputStream());         // 实例化客户端的输出流
            boolean flag = true;                       // 标志位，表示一个客户端是
                                                       否操作完毕
            while (flag) {                             // 客户端循环操作
                String str = buf.readLine();           // 在此处不断地接收信息
                if (str == null || "".equals(str)){// 判断输入的信息是否为空
                    flag = false;                      // 客户端操作结束
                } else {
                    if ("bye".equals(str)) {           // 如果输入信息为bye表示
                                                       结束
                        flag = false;                  // 客户端操作结束
                    } else {
                        out.println("ECHO:" + str);// 向客户端回显信息
```

```
                    }
                }
            }
            out.close();                                    // 关闭输出流
            client.close();                                 // 关闭客户端
        } catch (Exception e) {
        }
    }
};
```

从本线程类中可以发现，上述程序的主要功能是接收每一个客户端的 Socket，并通过循环的方式接收客户端的输入信息及向客户端输出信息。下面编写 EchoThreadServer 类，并使用以上的 EchoThread 类。

范例： EchoThreadServer

```
import java.net.ServerSocket;
import java.net.Socket;
public class EchoThreadServer {
    public static void main(String args[]) throws Exception {  // 所有异常
                                                                抛出
        ServerSocket server = null;                 // 定义ServerSocket对象
        Socket client = null;                       // 定义Socket对象，表示客户端
        server = new ServerSocket(8888);            // 此服务器在8888端口上进行监听
        boolean f = true;                           // 定义一个标记位
        while (f) {                                 // 无限制接收客户端连接
            System.out.println("服务器运行，等待客户端连接。") ;
            client = server.accept();               // 接收客户端连接
            new Thread(new EchoThread(client)).start() ;// 实例化并启动一个线
                                                        程对象
        }
        server.close();                             // 关闭服务器端
    }
};
```

在服务器端，每一个连接到服务器的客户端 Socket 都会以一个线程的方式运行，这样无论有多少个客户端连接都可以同时完成操作。

19.5　UDP 程序设计

19.5.1　UDP 简介

TCP 的所有操作都必须建立可靠的连接，这样肯定会浪费大量的系统性能。为了减少这种开销，在网络中又提供了另外一种传输协议——UDP，是不可靠的连接，这种协议在

各种聊天工具中被广泛地应用。

使用 UDP 发送的信息，对方不一定会接收到。所有的信息使用数据报的形式发送出去，所以这就要求客户端要始终等待服务器发送的消息才能进行接收，在 Java 中使用 DatagramSocket 类和 DatagramPacket 类完成 UDP 程序的开发。

提示：关于 UDP 开发中服务器和客户端的解释。

使用 UDP 开发的网络程序类似于平常使用的手机，手机实际上相当于一个客户端，如果手机要想正常地接收到信息，则手机肯定要先开机才行。

19.5.2 UDP 程序实现

在UDP开发中使用DatagramPacket类包装一条要发送的信息，之后使用DatagramSocket类用于完成信息的发送操作，这两个类的常用方法分别如表 19-9 和表 19-10 所示。

表 19-9 DatagramPacket 类的常用方法

序号	方　法	类型	描　述
1	public DatagramPacket(byte[] buf,int length)	构造	实例化 DatagramPacket 对象时指定接收数据的长度
2	public DatagramPacket(byte[] buf,int length, InetAddress address,int port)	构造	实例化 DatagramPacket 对象时指定发送的数据、数据的长度、目标地址及端口
3	public byte[] getData()	普通	返回接收的数据
4	public int getLength()	普通	返回要发送或接收数据的长度

表 19-10 DatagramSocket 类的常用方法

序号	方　法	类型	描　述
1	public DatagramSocket(int port) throws SocketException	构造	创建 DatagramPacket 对象，并指定监听的端口
2	public void send(DatagramPacket p) throws IOException	普通	发送数据报
3	public void receive(DatagramPacket p) throws IOException	普通	接收数据报

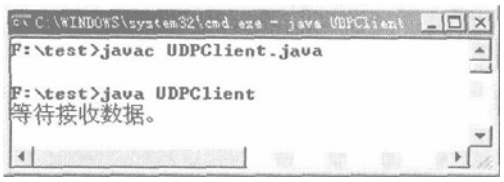
要想实现 UDP 程序，则首先应该从客户端编写，在客户端指定要接收数据的端口和取得数据。

范例： UDP 客户端——UDPClient

```
import java.net.DatagramPacket;
import java.net.DatagramSocket;
public class UDPClient {
    public static void main(String[] args) throws Exception {
        DatagramSocket ds = null;                    // 声明DatagramSocket对象
        byte[] buf = new byte[1024];                 // 定义接收数据的字节数组
        DatagramPacket dp = null;                    // 声明DatagramPacket对象
        ds = new DatagramSocket(9000);               // 此客户端在9000端口监听
        dp = new DatagramPacket(buf, 1024);          // 指定接收数据的长度为1024
```

```
            System.out.println("等待接收数据。") ;          // 输出信息
            ds.receive(dp);                                // 接收数据
            String str = new String(dp.getData(), 0, dp.getLength()) + " from "
                    + dp.getAddress().getHostAddress()
                    + " : " + dp.getPort();                // 接收数据
            System.out.println(str);                       // 输出数据
            ds.close();                                    // 关闭
        }
    }
```

程序运行结果：

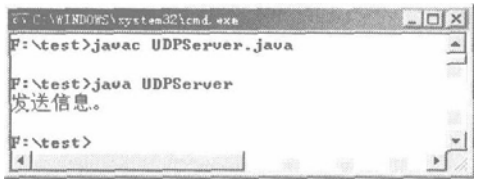


程序运行之后，客户端程序已经打开了监听端口，等待服务器端向客户端发送信息。

范例：编写 UDP 发送的服务器端程序——UDPServer

```
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
public class UDPServer {
    public static void main(String[] args) throws Exception {
        DatagramSocket ds = null;                  // 声明DatagramSocket对象
        DatagramPacket dp = null;                  // 声明DatagramPacket对象
        ds = new DatagramSocket(3000);             // 服务器端在3000端口监听
        String str = "hello world ";               // 准备好要发送的信息
        // 实例化DatagramPacket对象，指定数据内容、数据长度、要发送的目标地址、发送
           端口
        dp = new DatagramPacket(str.getBytes(), str.length(), InetAddress
                .getByName("localhost"), 9000);// 此处向客户端所在的9000端口
                                                  发送信息
        System.out.println("发送信息。") ;          // 信息输出
        ds.send(dp);                               // 发送数据报
        ds.close();                                // 关闭
    }
}
```

程序运行结果：



服务器端运行完成后，客户端可以接收服务器端发送过来的信息，如图 19-6 所示。

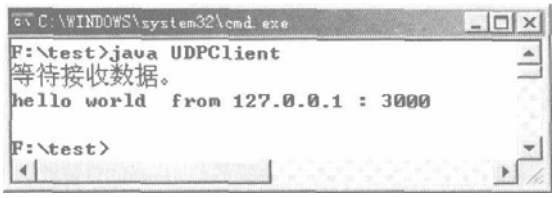


图 19-6 UDP 客户端运行

19.6 本章要点

1．InetAddress 表示 IP 地址的操作类，可以通过此类指定要连接的主机名称。

2．在开发中如果要取得一个网络资源可以使用 URL 类进行定位，之后使用 IO 流的方式进行读取。

3．使用 URLEncoder 可以为信息编码，使用 URLDecoder 可以为编码的内容进行解码操作。

4．ServerSocket 主要用在 TCP 协议的服务器程序开发上，使用 accept()方法等待客户端连接，每一个连接的客户端都使用一个 Socket 表示。

5．服务器端加入多线程机制之后，就可以同时为多个用户提供服务。

6．UDP 属于不可靠的连接协议，采用数据报的形式，对于服务器发送的信息，客户端不一定能接收到。

19.7 习 题

1．将 Echo 程序进行扩充，通过图形界面编写服务器和客户端，实现通过按钮控制服务器的启动与关闭，并使用界面发送和接收服务器的返回信息。

2．使用网络程序完成一个多用户的用户注册操作，所有的用户只要连接到服务器上，就可以对数据库进行增加数据的操作。

3．使用 TCP 程序完成以下功能：

在服务器端的数据库表中建立部门表（部门编号、部门名称、部门位置），当客户端连接到服务器之后，服务器将这些信息全部返回到客户端进行输出。

第 20 章　Java 新 IO

通过本章的学习可以达到以下目标：

- ☑ 了解新 IO 操作与传统 IO 操作的区别。
- ☑ 掌握新 IO 操作中缓冲区的操作原理。
- ☑ 了解子缓冲区、只读缓冲区、直接缓冲区的区别及使用。
- ☑ 了解新 IO 中通道的概念，并可以使用通道进行双向操作。
- ☑ 掌握文件锁的使用。
- ☑ 掌握编码器的作用。
- ☑ 了解 Selector 实现非阻塞服务器的操作。

之前为读者讲解过，如果一个程序现在需要等待用户输入数据，则可以通过 System.in 来完成。但是这样一来，在使用时就会出现一个问题：如果用户没有输入信息，则肯定会一直等待用户输入，大量的系统资源都会被白白浪费。所以，为了提升传统 IO 的操作性能，在 JDK 1.4 之后引入了新 IO 处理机制——NIO。**本章视频录像讲解时间为 1 小时 33 分钟，源代码在光盘对应的章节下。**

20.1　Java 新 IO 简介

NIO 提供了一个全新的底层 I/O 模型。与最初的 java.io 包中面向流（stream-oriented）的概念不同，NIO 中采用了面向块的概念（block-oriented）。这意味着在尽可能的情况下，I/O 操作以大的数据块为单位进行，而不是一次一个字节或字符进行。采用这样的操作方式 Java 的 I/O 性能已经有了很大的提高，当然，这样做也牺牲了 Java 操作的简单性。

NIO 中提供与平台无关的非阻塞 I/O（nonblocking I/O）。与面向线程的、阻塞式 I/O 方式相比，多道通信、非阻塞 I/O 技术可以使应用程序更有效地处理大量连接的情况。

提示：IO 的阻塞操作。

之前讲解过 IO 操作中接收键盘数据的操作时只要执行到 readLine()方法，程序就要停止等待用户输入数据；在网络编程中讲解过在服务器端使用 ServerSocket 类的 accept()方法时，服务器一直处于等待操作，要等待客户端连接。这两类操作都属于阻塞操作，因为都会让程序暂停执行。

新 IO 并没有在原来的 IO 基础上开发，而是采用了全新的类和接口，除了原有的功能之外还提供了以下新的特性：

- ➘ 多路选择的非封锁式 I/O 设施。
- ➘ 支持文件锁和内存映射。

- 支持正则表达式的模式匹配设施。
- 字符集编码器和译码器。

在 Java 新 IO 中使用 Buffer 和 Channel 支持以上的操作，下面分别介绍这两个概念。

20.2 缓冲区与 Buffer

在基本 IO 操作中所有的操作都是直接以流的形式完成的；而在 NIO 中所有的操作都要使用到缓冲区处理，且所有的读写操作都是通过缓冲区完成的。缓冲区（Buffer）是一个线性的、有序的数据集，只能容纳某种特定的数据类型。

20.2.1 Buffer 的基本操作

java.nio.Buffer 本身是一个抽象类，常用方法如表 20-1 所示。

表 20-1 Buffer 类的常用方法

序号	方 法	类型	描 述
1	public final int capacity()	普通	返回此缓冲区的容量
2	public final int limit()	普通	返回此缓冲区的限制
3	public final Buffer limit(int newLimit)	普通	设置缓冲区的限制
4	public final int position()	普通	返回缓冲区的操作位置
5	public final Buffer position(int newPosition)	普通	设置缓冲区的操作位置
6	public final Buffer clear()	普通	清空缓冲区
7	public final Buffer flip()	普通	重设缓冲区，在写入之前调用，改变缓冲的指针
8	public final Buffer reset()	普通	恢复缓冲区中的标记位置
9	public final boolean hasRemaining()	普通	判断在当前位置和限制之间是否有内容

在新 IO 中针对每一种基本数据类型都有一种对应的缓冲区操作类，如表 20-2 所示。

表 20-2 各种数据类型的缓冲区类

序号	缓 冲 区 类	描 述
1	java.nio.ByteBuffer	存储字节的 Buffer
2	java.nio.CharBuffer	存储字符的 Buffer
3	java.nio.ShortBuffer	存储短整型的 Buffer
4	java.nio.IntBuffer	存储整型的 Buffer
5	java.nio.LongBuffer	存储长整型的 Buffer
6	java.nio.FloatBuffer	存储单精度浮点型的 Buffer
7	java.nio.DoubleBuffer	存储双精度浮点型的 Buffer

在以上的 7 种数据缓冲操作类中，都提供了如表 20-3 所示的几组常用方法。

表 20-3　各数据类型缓冲区类提供的常用方法

序号	方　　法	类型	描　　述
1	public static 缓冲区类型 allocate(int capacity)	普通	分配缓冲区空间
2	public 基本数据类型 get()	普通	取得当前位置的内容
3	public 基本数据类型 get(int index)	普通	取得指定位置的内容
4	public 缓冲区类型 put(基本数据类型 x)	普通	写入指定基本数据类型的数据
5	public final 缓冲区类型 put(数据类型[] src)	普通	写入一组指定的基本数据类型的数据
6	public final 缓冲区类型 put(数据类型[] src,int offset,int length)	普通	写入一组指定的基本数据类型的数据
7	public 缓冲区类型 slice()	普通	创建子缓冲区，其中一部分与原缓冲区共享数据
8	public 缓冲区类型 asReadOnlyBuffer()	普通	将缓冲区设置为只读缓冲区

以上方法针对于各个缓冲区类型都有效，下面介绍基本的操作流程，以 IntBuffer 类的操作为例。

范例：演示缓冲区的操作流程，同时观察 position、limit 和 capacity

```
package org.lxh.demo20.bufferdemo;
import java.nio.IntBuffer;
public class IntBufferDemo01 {
    public static void main(String[] args) {
        IntBuffer buf = IntBuffer.allocate(10);     // 开辟10个大小的缓冲区
        System.out.print("1、写入数据之前的position、limit和capacity: ") ;
        System.out.println("position = " + buf.position() + ", limit = "
                + buf.limit() + ", capacity = " + buf.capacity());
        int temp[] = { 5, 7, 9 };                   // 定义整型数组
        buf.put(3);                                 // 向缓冲区写入数据
        buf.put(temp);                              // 向缓冲区写入一组数据
        System.out.print("2、写入数据之后的position、limit和capacity: ") ;
        System.out.println("position = " + buf.position() + ", limit = "
                + buf.limit() + ", capacity = " + buf.capacity());
        buf.flip();                                 // 重设缓冲区
        System.out.print("3、准备输出数据时的position、limit和capacity: ") ;
        System.out.println("position = " + buf.position() + ", limit = "
                + buf.limit() + ", capacity = " + buf.capacity());
        System.out.print("缓冲区中的内容: ") ;
        while (buf.hasRemaining()) {                // 只要缓冲区有内容则输出
            int x = buf.get();                      // 取出当前内容
            System.out.print(x + "、");             // 输出内容
        }
    }
}
```

程序运行结果：

```
1、写入数据之前的position和limit: position = 0, limit = 10, capacity = 10
2、写入数据之后的position和limit: position = 4, limit = 10, capacity = 10
3、准备输出数据时的position和limit: position = 0, limit = 4, capacity = 10
缓冲区中的内容：3、5、7、9、
```

以上程序首先开辟了 10 个长度的缓冲区空间，之后向缓冲区中写入了 4 个元素，在每次写入之后 position 都会有变化，而当调用 flip()方法时，position 和 limit 将同时产生变化，这实际上就是面向块的操作。程序的最后使用 hasRemaining()方法循环判断缓冲区中是否有内容，有内容则依次取出。

提示：写入的数据不能超出规定的缓冲区大小。

如果只开辟了 3 个缓冲区，但是却写入了 5 个数据，则操作中就会出现以下错误提示：

```
Exception in thread "main" java.nio.BufferOverflowException
    at java.nio.HeapIntBuffer.put(Unknown Source)
```

20.2.2 深入缓冲区操作

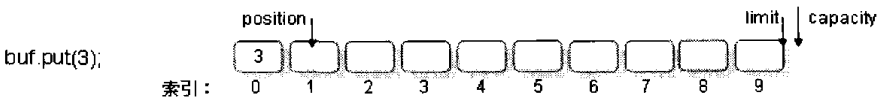
在 Buffer 中存在一系列的状态变量，这些状态变量随着写入或读取都有可能被改变。在缓冲区中可以使用 3 个值表示缓冲区的状态。

- position：表示下一个缓冲区读取或写入的操作指针，当向缓冲区中写入数据时此指针就会改变，指针永远放到写入的最后一个元素之后。例如，如果写入了 4 个位置的数据，则 position 会指向第 5 个位置。
- limit：表示还有多少数据需要存储或读取，position<=limit。
- capacity：表示缓冲区的最大容量，limit<=capacity。此值在分配缓冲区时被设置，一般不会更改。

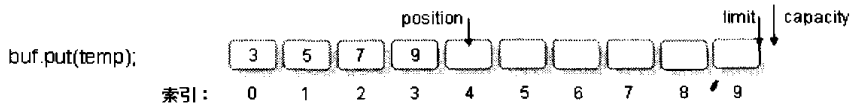
在之前的例子中，读者应该知道随着数据向缓冲区中的写入或是重设，对应的 position 和 limit 也会改变。下面将逐步分析这些操作的步骤，如图 20-1 所示。



（a）开辟缓冲区

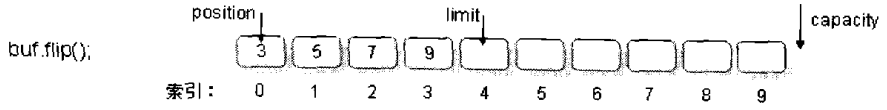


（b）向缓冲区中增加一个数据



（c）向缓冲区中增加一组数据

图 20-1　缓冲区操作细节



（d）执行 flip()方法，limit 设置为 position ，position 设置为 0

图 20-1　缓冲区操作细节（续）

20.2.3　创建子缓冲区

可以使用各个缓冲区类的 slice()方法从一个缓冲区中创建一个新的子缓冲区，子缓冲区与原缓冲区中的部分数据可以共享。下面通过范例来观察其使用。

范例：观察子缓冲区的创建及数据共享

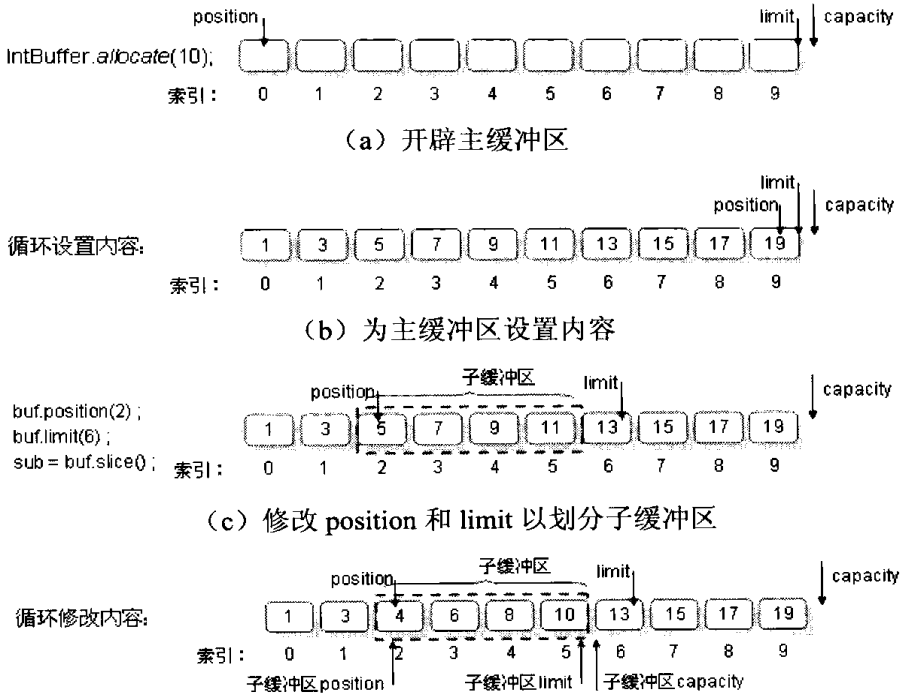
```
package org.lxh.demo20.bufferdemo;
import java.nio.IntBuffer;
public class IntBufferDemo02 {
    public static void main(String[] args) {
        IntBuffer buf = IntBuffer.allocate(10);     // 开辟10个大小的缓冲区
        IntBuffer sub = null ;                      // 定义缓冲区对象
        for (int i = 0; i < 10; i++) {
            buf.put(2 * i + 1);                     // 加入10个奇数
        }
        buf.position(2) ;                           // 主缓冲区指针设置在第3个
                                                       元素上
        buf.limit(6) ;                              // 主缓冲区limit为6
        sub = buf.slice() ;                         // 开辟子缓冲区
        for (int i = 0; i < sub.capacity(); i++) {
            int temp = sub.get(i) ;                 // 根据下标取得元素
            sub.put(temp - 1);                      // 修改子缓冲区内容
        }
        buf.flip() ;                                // 重设缓冲区
        buf.limit(buf.capacity()) ;                 // 设置limit
        System.out.print("主缓冲区中的内容：") ;
        while (buf.hasRemaining()) {                // 只要缓冲区有内容则输出
            int x = buf.get();                      // 取出当前内容
            System.out.print(x + "、");             // 输出内容
        }
    }
}
```

程序运行结果：

```
主缓冲区中的内容：1、3、4、6、8、10、13、15、17、19、
```

以上程序中，子缓冲区的内容改变之后主缓冲区的内容也跟着一起变化。本程序的缓

冲操作如图 20-2 所示。



（a）开辟主缓冲区

（b）为主缓冲区设置内容

（c）修改 position 和 limit 以划分子缓冲区

（d）修改子缓冲区中的内容，因为是共享数据，所以主缓冲区的内容也一起改变

图 20-2　子缓冲区操作图

20.2.4　创建只读缓冲区

如果现在需要使用到缓冲区中的内容，但又不希望其内容被修改，则可以通过 asReadOnlyBuffer()方法创建一个只读缓冲区，但是创建完毕后，此缓冲区不能变为可写状态。

范例：创建只读缓冲区

```
package org.lxh.demo20.bufferdemo;
import java.nio.IntBuffer;
public class IntBufferDemo03 {
    public static void main(String[] args) {
        IntBuffer buf = IntBuffer.allocate(10);      // 开辟10个大小的缓冲区
        IntBuffer read = null ;                      // 定义缓冲区对象
        for (int i = 0; i < 10; i++) {
            buf.put(2 * i + 1);                      // 加入10个奇数
        }
        read = buf.asReadOnlyBuffer() ;              // 创建只读缓冲区
        read.flip() ;                                // 重设缓冲区
        System.out.print("缓冲区中的内容：") ;
        while (read.hasRemaining()) {                // 只要缓冲区有内容则输出
            int x = read.get();                      // 取出当前内容
```

```
            System.out.print(x + "、");                    // 输出内容
        }
        System.out.println();                              // 换行
        read.put(30) ;                                     // 错误，不可写
    }
}
```

程序运行结果：

```
缓冲区中的内容：1、3、5、7、9、11、13、15、17、19、
Exception in thread "main" java.nio.ReadOnlyBufferException
    at java.nio.HeapIntBufferR.put(Unknown Source)
```

从程序的运行结果中可以发现，从之前的缓冲区中创建了一个新的缓冲区，新创建的缓冲区可以输出里面的内容，但是一旦修改内容就会出现异常。

20.2.5　创建直接缓冲区

在缓冲区操作类中，只有 ByteBuffer 可以创建直接缓冲区，这样 Java 虚拟机将尽最大努力直接对其执行本机的 IO 操作。也就是说，在每次调用基础操作系统的一个本机 I/O 操作之前（或之后），虚拟机都会尽量避免将缓冲区的内容复制到中间缓冲区中（或从中间缓冲区中复制内容）。

创建直接缓冲区时直接使用 ByteBuffer 类定义如下方法即可：

```
public static ByteBuffer allocateDirect(int capacity)
```

范例：创建直接缓冲区

```
package org.lxh.demo20.bufferdemo;
import java.nio.ByteBuffer;
public class ByteBufferDemo01 {
    public static void main(String[] args) {
        ByteBuffer buf = null ;                            // 声明ByteBuffer对象
        buf = ByteBuffer.allocateDirect(10) ;              // 开辟直接缓冲区
        byte temp[] = { 1, 3, 5, 7, 9 };                   // 定义byte数组
        buf.put(temp);                                     // 向缓冲区写入一组数据
        buf.flip();                                        // 重设缓冲区
        System.out.print("缓冲区中的内容：") ;
        while (buf.hasRemaining()) {                       // 只要缓冲区有内容则输出
            int x = buf.get();                             // 取出当前内容
            System.out.print(x + "、");                    // 输出内容
        }
    }
}
```

程序运行结果：

```
缓冲区中的内容：1、3、5、7、9、
```

20.3 通　　道

通道（Channel）可以用来读取和写入数据，通道类似于之前的输入/输出流，但是程序不会直接操作通道，所有的内容都是先读到或写入到缓冲区中，再通过缓冲区中取得或写入的。

通道与传统的流操作不同，传统的流操作分为输入或输出流，而通道本身是双向操作的，既可以完成输入也可以完成输出，如图 20-3 所示。

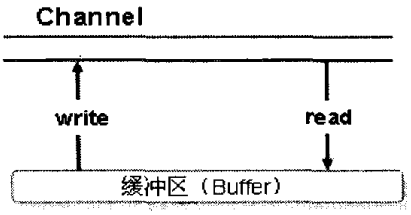


图 20-3　通道的操作流程

Channel 本身是一个接口，此接口定义了如表 20-4 所示的方法。

表 20-4　Channel 接口定义的方法

序号	方　　法	类型	描　　述
1	void close() throws IOException	普通	关闭通道
2	boolean isOpen()	普通	判断此通道是否是打开的

20.3.1 FileChannel

FileChannel 是 Channel 的子类，可以进行文件的读/写操作。此类的常用方法如表 20-5 所示。

表 20-5　FileChannel 类的常用方法

序号	方　　法	类型	描　　述
1	public int read(ByteBuffer dst) throws IOException	普通	将内容读入到缓冲区中
2	public int write(ByteBuffer src) throws IOException	普通	将内容从缓冲区写入到通道
3	public final void close() throws IOException	普通	关闭通道
4	public abstract MappedByteBuffer map(FileChannel.MapMode mode,long position,long size) throws IOException	普通	将通道的文件区域映射到内存中，同时指定映射模式、文件中的映射文件以及要映射的区域大小

如果要使用 FileChannel，则可以依靠 FileInputStream 或 FileOutputStream 类中的 getChannel()方法取得输入或输出的通道。下面为读者演示一个使用通道写入文本的操作。

范例：使用输出通道输出内容

```
package org.lxh.demo20.channeldemo;
import java.io.File;
import java.io.FileOutputStream;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
public class FileChannelDemo01 {
    public static void main(String[] args) throws Exception {
        String info[] = { "MLDN", "MLDNJAVA", "www.mldn.cn", "www.mldnjava.cn",
                "李兴华", "lixinghua" };                    // 待输出的数据
        File file = new File("d:" + File.separator + "out.txt");
        FileOutputStream output = null;                    // 文件输出流
        output = new FileOutputStream(file);               // 实例化输出流
        FileChannel fout = null ;                          // 声明输出的通道对象
        fout = output.getChannel() ;                       // 得到输出的文件通道
        ByteBuffer buf = ByteBuffer.allocate(1024);// 开辟缓冲
        for (int i = 0; i < info.length; i++) {            // 循环将内容写入缓冲
            buf.put(info[i].getBytes()) ;                  // 向缓冲中写入数据
        }
        buf.flip() ;                                       // 重设缓冲区，准备输出
        fout.write(buf) ;                                  // 输出
        fout.close() ;                                     // 关闭输出通道
        output.close() ;                                   // 关闭输出流
    }
}
```

程序运行结果：



以上程序是使用输出通道将内容全部放到缓冲中，一次性写入到文件中的，实际上 FileChannel 是双向操作的，同时可以完成输出和输入数据的功能。下面介绍如何应用其进行读写文件的操作。

范例：使用通道进行读写操作

```
package org.lxh.demo20.channeldemo;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
public class FileChannelDemo02 {
    public static void main(String[] args) throws Exception {
```

```
        File file1 = new File("d:" + File.separator + "note.txt");
        File file2 = new File("d:" + File.separator + "outnote.txt");
        FileInputStream input = null;                          // 文件输入流
        FileOutputStream output = null;                        // 文件输出流
        input = new FileInputStream(file1);                    // 实例化输入流
        output = new FileOutputStream(file2);                  // 实例化输出流
        FileChannel fin = null ;                               // 声明输入的通道对象
        FileChannel fout = null ;                              // 声明输出的通道对象
        fin = input.getChannel() ;                             // 得到输入的文件通道
        fout = output.getChannel() ;                           // 得到输出的文件通道
        ByteBuffer buf = ByteBuffer.allocate(1024) ;           // 开辟缓冲
        int temp = 0 ;                                         // 声明变量接收内容
        while ((temp = fin.read(buf)) != -1) {                 // 如果没读到底
            buf.flip() ;                                       // 重设缓冲区
            fout.write(buf) ;                                  // 输出缓冲区
            buf.clear() ;                                      // 清空缓冲区
        }
        fin.close() ;                                          // 关闭输入通道
        fout.close() ;                                         // 关闭输出通道
        input.close() ;                                        // 关闭输入流
        output.close() ;                                       // 关闭输出流
    }
}
```

20.3.2 内存映射

内存映射可以把文件映射到内存中，这样文件内的数据就可以用内存读/写指令来访问，而不是用 InputStream 或 OutputStream 这样的 I/O 操作类，采用此种方式读取文件的速度是最快的。

提示：Java 中访问文件内容的 4 种方法。

- RandomAccessFile，随机读取数据，此种访问速度较慢。
- FileInputStream，文件输入流，使用此种方式速度较慢。
- 缓冲读取（例 BufferedReader），使用此种方式访问速度较快。
- 内存映射（MappedByteBuffer），使用此种方式读取速度最快。

要想将文件映射到内存中，可以使用 FileChannel 类提供的 map()方法，此方法定义如下：

```
public abstract MappedByteBuffer map(FileChannel.MapMode mode,long position,
long size) throws IOException
```

map()方法在使用时要指定映射模式，在内存映射中提供了 3 种模式，此 3 种模式分别

由 FileChannel 类中的 3 个常量表示，如表 20-6 所示。

表 20-6　FileChannel 类的 3 种内存映射模式

序号	常　　量	类型	描　　述
1	public static final FileChannel.MapMode READ_ONLY	常量	只读映射模式
2	public static final FileChannel.MapMode READ_WRITE	常量	读取/写入映射模式
3	public static final FileChannel.MapMode PRIVATE	常量	专用（写入时复制）映射模式

下面通过一个读取文件的操作来观察如何使用 MappedByteBuffer 读取硬盘上的文件，假设在 d 盘中存在一个 mldn.txt 的文件，文件内容如图 20-4 所示。

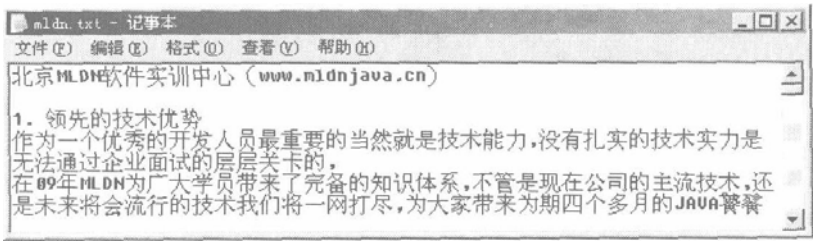


图 20-4　d:\mldn.txt 文件的内容

范例：内存映射

```
package org.lxh.demo20.channeldemo;
import java.io.File;
import java.io.FileInputStream;
import java.nio.MappedByteBuffer;
import java.nio.channels.FileChannel;
public class FileChannelDemo03 {
    public static void main(String[] args) throws Exception {
        File file = new File("d:" + File.separator + "mldn.txt");
        FileInputStream input = null;                          // 文件输入流
        input = new FileInputStream(file);                     // 实例化输入流
        FileChannel fin = null ;                               // 声明输入的通道对象
        fin = input.getChannel() ;                             // 得到输入文件通道
        MappedByteBuffer mbb = null ;                          // 声明文件的内存映射
        mbb = fin.map(FileChannel.MapMode.READ_ONLY,
                0, file.length()) ;                            // 将文件映射到内存中
        byte data[] = new byte[(int) file.length()];           // 开辟字节数组，接收
                                                               数据
        int foot = 0 ;                                         // 定义下标
        while(mbb.hasRemaining()){                             // 判断是否有数据
            data[foot++] = mbb.get() ;                         // 取出数据
        }
        System.out.println(new String(data)) ;                 // 显示输入的数据
        fin.close() ;                                          // 关闭输入通道
```

```
            input.close() ;                                    // 关闭输入流
        }
    }
```

程序运行结果（部分）：

```
北京MLDN软件实训中心（www.mldnjava.cn）

1．领先的技术优势
作为一个优秀的开发人员最重要的当然就是技术能力，没有扎实的技术实力是无法通过企业面试的层层关卡的，
在09年MLDN为广大学员带来了完备的知识体系，不管是现在公司的主流技术，还是未来将会流行的技术我们将一网打尽
...
```

尽管创建内存映射文件非常简单，但是如果使用 MappedByteBuffer 写入数据就可能非常危险。因为仅仅是改变数组中的单个元素的内容这样的简单操作，就有可能直接修改磁盘上的具体文件，因为修改数据与数据重新保存到磁盘是一样的。

20.4 文件锁：FileLock

在 Java 新 IO 中提供了文件锁的功能，这样当一个线程将文件锁定之后，其他线程是无法操作此文件的。要想进行文件的锁定操作，则要使用 FileLock 类完成，此类的对象需要依靠 FileChannel 进行实例化操作。

FileChannel 类中提供了如表 20-7 所示的几个方法取得 FileLock 类的实例化对象。

表 20-7 FileChannel 类中提供的实例化 FileLock 对象的方法

序号	方　法	类型	描　述
1	public final FileLock lock() throws IOException	普通	获得此通道文件的独占锁定
2	public abstract FileLock lock(long position,long size, boolean shared) throws IOException	普通	获得此通道文件给定区域的锁定，并指定锁定位置、锁定大小、是共享锁定（true）或是独占锁定（false）
3	public final FileLock tryLock() throws IOException	普通	试图获取此通道的独占锁定
4	public abstract FileLock tryLock(long position,long size,boolean shared) throws IOException	普通	试图获取此通道指定区域的锁定，并指定锁定位置、锁定大小，属于共享锁定（true）或是独占锁定（false）

表 20-7 所示的文件锁定方式有两种。

- 共享锁：允许多个线程进行文件的读取操作。
- 独占锁：只允许一个线程进行文件的读/写操作。

文件锁定之后需要依靠 FileLock 类进行解锁，此类的常用方法如表 20-8 所示。

表 20-8　FileLock 类的常用方法

序号	方　法	类型	描　述
1	public final boolean isShared()	普通	判断锁定是否为共享锁定
2	public final FileChannel channel()	普通	返回此锁定的 FileChannel
3	public abstract void release() throws IOException	普通	释放锁定（解锁）
4	public final long size()	普通	返回锁定区域的大小

范例：将 d:\\mldn.txt 文件锁定

```
package org.lxh.demo20.filelockdemo;
import java.io.File;
import java.io.FileOutputStream;
import java.nio.channels.FileChannel;
import java.nio.channels.FileLock;
public class FileLockDemo {
    public static void main(String[] args) throws Exception {
        File file = new File("d:" + File.separator + "mldn.txt");
        FileOutputStream output = null;                    // 文件输入流
        output = new FileOutputStream(file , true);// 实例化输出流
        FileChannel fout = null ;                          // 声明输出的通道对象
        fout = output.getChannel() ;                       // 得到输入文件通道
        FileLock lock = fout.tryLock() ;                   // 试图获得此通道的文件锁
        if (lock != null) {                                // 判断是否锁定
            System.out.println(file.getName() + "文件锁定300秒。");
            Thread.sleep(300000);                          // 将文件锁定300秒
            lock.release();                                // 释放文件锁
            System.out.println(file.getName() + "文件解除锁定。");
        }
        fout.close() ;                                     // 关闭输出通道
        output.close() ;                                   // 关闭输出流
    }
}
```

以上程序在运行时将文件进行独占锁定，这样其他线程在锁定的 300 秒内是无法对此文件进行读写操作的。

20.5　字符集：Charset

在 Java 语言中所有的信息都是以 UNICODE 进行编码的，但是在计算机的世界里并不只单单存在一种编码，而是多个，而且如果对编码处理不当，就有可能产生乱码。在 Java 的新 IO 包中提供了 Charset 类来负责处理编码的问题，该类还包含了创建编码器（CharsetEncoder）和创建解码器（CharsetDecoder）的操作。

提示：编码器和解码器。

编码和解码实际上是从最早的电报发展起来的，所有的内容如果需要使用电报传送，则必须变为相应的编码，之后再通过指定的编码进行解码的操作。在新 IO 中为了保证程序可以适应各种不同的编码，所以提供了编码器和解码器，通过解码器程序可以方便地读取各个平台上不同编码的数据，之后再通过编码器将程序的内容以正确的编码进行输出。

Charset 类的常用操作方法如表 20-9 所示。

表 20-9 Charset 类的常用方法

序号	方 法	类型	描 述
1	public static SortedMap<String,Charset> availableCharsets()	普通	取得 Charset 的全部字符集
2	public static Charset forName(String charsetName)	普通	返回指定编码方式的 Charset 对象
3	public abstract CharsetEncoder newEncoder()	普通	创建编码器
4	public abstract CharsetDecoder newDecoder()	普通	创建解码器

CharsetEncoder 类的常用方法如表 20-10 所示。

表 20-10 CharsetEncoder 类的常用方法

序号	方 法	类型	描 述
1	public final ByteBuffer encode(CharBuffer in) throws CharacterCodingException	普通	将单个输入字符进行编码

CharsetDecoder 类的常用方法如表 20-11 所示。

表 20-11 CharsetDecoder 类的常用方法

序号	方 法	类型	描 述
1	public final CharBuffer decode(ByteBuffer in) throws CharacterCodingException	普通	将编码后的内容进行解码

要想对内容进行编码，则首先应该掌握 Charset 类中所支持的全部编码，可以通过下面的程序取得 Charset 类支持的全部编码。

范例：取得 Charset 类的全部编码

```
package org.lxh.demo20.charsetdemo;
import java.nio.charset.Charset;
import java.util.Iterator;
import java.util.Map;
import java.util.SortedMap;
public class GetAllCharsetDemo {
    public static void main(String[] args) {
        SortedMap<String, Charset> all = null;      // 声明SortedMap集合
        all = Charset.availableCharsets();          // 取得全部编码
        Iterator<Map.Entry<String,Charset>> iter = null;// 声明Iterator对象
```

```
            iter = all.entrySet().iterator();                    // 实例化Iterator
            while (iter.hasNext()) {                             // 迭代输出
                Map.Entry<String, Charset> me = iter.next();// 取出每一个Map.Entry
                System.out.println(me.getKey() +
                        " ----> " + me.getValue());              // 输出信息
            }
        }
    }
```

程序运行结果（部分）：

```
Big5 ----> Big5
Big5-HKSCS ----> Big5-HKSCS
EUC-JP ----> EUC-JP
EUC-KR ----> EUC-KR
GB18030 ----> GB18030
GB2312 ----> GB2312
GBK ----> GBK
...
```

运行以上程序之后将返回全部支持的字符集，Map 集合中的 key 保存的是每种编码的别名，在实际使用时可以使用 forName()方法根据编码的别名实例化 CharSet 对象。

下面的范例演示了对 d:\mldn.txt 中的内容使用 ISO-8859-1 编码的过程，本范例将使用 CharsetEncoder 和 CharsetDecoder 类完成操作。

范例：编码-解码操作

```
package org.lxh.demo20.charsetdemo;
import java.nio.ByteBuffer;
import java.nio.CharBuffer;
import java.nio.charset.Charset;
import java.nio.charset.CharsetDecoder;
import java.nio.charset.CharsetEncoder;
public class CharsetEnDeDemo {
    public static void main(String[] args) throws Exception {
        Charset latin1 = Charset.forName("ISO-8859-1");// 以ISO-8859-1编码
        CharsetEncoder encoder = latin1.newEncoder() ; // 实例化编码对象
        CharsetDecoder decoder = latin1.newDecoder() ; // 实例化解码对象
        // 通过CharBuffer类中的wrap()方法，将一个字符串变为CharBuffer类型
        CharBuffer cb = CharBuffer.wrap("北京MLDN软件实训中心！");
        ByteBuffer buf = encoder.encode(cb);                 // 进行编码操作
        System.out.println(decoder.decode(buf));             // 进行解码操作
    }
}
```

程序运行后将以 ISO-8859-1 的方式显示中文，这样肯定是无法显示的，会造成乱码问

题。一般 CharsetEncoder 和 CharsetDecoder 都经常使用在文件的读写上，以实现文件编码的转换功能。

20.6 Selector

在新 IO 中 Selector 是一个极其重要的概念，在原来使用 IO 和 Socket 构造网络服务时，所有的网络服务将使用阻塞的方式进行客户端的连接，而如果使用了新 IO 则可以构造一个非阻塞的网络服务。Selector 类的常用方法如表 20-12 所示。

表 20-12 Selector 类的常用方法

序号	方 法	类型	描 述
1	public static Selector open() throws IOException	普通	打开一个选择器
2	public abstract int select() throws IOException	普通	选择一组键，通道已经为 IO 做好准备
3	public abstract Set<SelectionKey> selectedKeys()	普通	返回此选择器已选择的 key

在进行非阻塞网络开发时需要使用 SelectableChannel 类向 Select 类注册，而且在新 IO 中实现网络程序需要依靠 ServerSocketChannel 类与 SocketChannel 类，这两个类都是 SelectableChannel 的子类，SelectableChannel 提供了注册 Selector 的方法和阻塞模式。

ServerSocketChannel 类的常用方法如表 20-13 所示。

表 20-13 ServerSocketChannel 类的常用方法

序号	方 法	类型	描 述
1	public abstract SelectableChannel configureBlocking (boolean block) throws IOException	普通	调整此通道的阻塞模式，如果为 true 将被设置为阻塞模式，如果为 false 将被设置为非阻塞模式
2	public final SelectionKey register(Selector sel,int ops) throws ClosedChannelException	普通	向指定的选择器注册通道并设置 Selector 域，返回一个选择键
3	public static ServerSocketChannel open() throws IOException	普通	打开服务器的套接字通道
4	public abstract ServerSocket socket()	普通	返回与此通道关联的服务器套接字

在使用 register()方法时需要指定一个选择器（Selector 对象）以及 Select 域，Selector 对象可以通过 Selector 中的 open()方法取得，而 Selector 域则在 SelectionKey 类中定义，如表 20-14 所示。

表 20-14 4 种 Selector 域

序号	方 法	类型	描 述
1	public static final int OP_ACCEPT	常量	相当于 ServerSocket 中的 accpet()操作
2	public static final int OP_CONNECT	常量	连接操作
3	public static final int OP_READ	常量	读操作
4	public static final int OP_WRITE	常量	写操作

通过如下代码可以建立一个非阻塞的服务器端：

```
int ports = 8888;                                    // 定义连接的端口号
Selector selector = Selector.open();                 // 打开一个选择器
ServerSocketChannel initSer = null ;                 // 声明ServerSocketChannel
initSer = ServerSocketChannel.open();                // 打开服务器套接字通道
initSer.configureBlocking(false);                    // 服务器配置为非阻塞
ServerSocket initSock = initSer.socket();            // 检索与此通道关联的服务器套接字
InetSocketAddress address = null ;                   // 表示监听地址
address = new InetSocketAddress(ports);              // 实例化绑定地址
initSock.bind(address);                              // 绑定地址
initSer.register(selector,SelectionKey.OP_ACCEPT); // 注册选择器，相当于使用
                                                        accept()方法接收
System.out.println("服务器运行，在" + ports + "端口监听。");
```

上述程序代码中，首先通过 Selector.open()方法打开一个选择器，之后通过 ServerSocketChannel 类中的 open()方法打开一个服务器套接字通道。程序中最重要的就是 configureBlocking()方法，将服务器设置为非阻塞状态，之后通过 ServerSocketChannel 的 socket()方法返回一个 ServerSocket 对象，并在 8888 端口绑定服务器的监听端口。程序的最后使用 register()方法将选择器注册为 accept 方式，等待客户端连接。

如果要使用服务器向客户端发送信息，则需要通过 SelectionKey 类中提供的方法判断服务器的操作状态。而要想取得客户端的连接也需要使用 SelectionKey 类，此类的常用方法如表 20-15 所示。

表 20-15　SelectionKey 类的常用方法

序号	方　法	类型	描　述
1	public abstract SelectableChannel channel()	普通	返回创建此 key 的通道
2	public final boolean isAcceptable()	普通	判断此通道是否可以接收新的连接
3	public final boolean isConnectable()	普通	判断此通道是否完成套接字的连接操作
4	public final boolean isReadable()	普通	判断此通道是否可以进行读取操作
5	public final boolean isWriteable()	普通	判断此通道是否可以进行写操作

因为 SelectionKey 中提供了 4 种操作状态，所以 4 种状态也对应了 4 个 isXxx()方法，取得 SelectionKey 的操作方法如下所示：

```
int keysAdd = 0;                                     // 接收一组SelectionKey
while ((keysAdd = selector.select()) > 0) {          // 选择一组键，相应的通道已
                                                        为IO准备就绪
    Set<SelectionKey> selectedKeys = selector.selectedKeys();// 取出全部生成
                                                                的key
    Iterator<SelectionKey> iter = selectedKeys.iterator(); // 实例化
                                                              Iterator
    while (iter.hasNext()) {                         // 迭代全部的key
```

```
            SelectionKey key = (SelectionKey)iter.next();// 取出每一个SelectionKey
                if (key.isAcceptable()) {                    // 判断客户端是否已经连接上
                    // 执行服务器的输出操作
                }
            }
            selectedKeys.clear();                            // 清除全部的key
        }
    }
```

下面使用 Selector 创建一个非阻塞的服务器，此服务器向客户端返回当前的系统时间。

范例：取得时间的服务器

```
package org.lxh.demo20.selectordemo;
import java.net.InetSocketAddress;
import java.net.ServerSocket;
import java.nio.ByteBuffer;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.util.Date;
import java.util.Iterator;
import java.util.Set;
public class DateServer {
    public static void main(String args[]) throws Exception {// 所有异常
                                                                抛出
        int ports[] = { 8000, 8001, 8002, 8003, 8005, 8006 };// 定义一组连
                                                              接端口号
        Selector selector = Selector.open();          // 打开一个选择器
        for (int i = 0; i < ports.length; i++) {      // 构造服务器的启动信息
            ServerSocketChannel initSer = null ;
                                            // 声明ServerSocketChannel
            initSer = ServerSocketChannel.open();     // 打开服务器套接字通道
            initSer.configureBlocking(false);         // 服务器配置为非阻塞
            ServerSocket initSock = initSer.socket();  // 检索与此通道关联
                                                         的服务器套接字
            InetSocketAddress address = null ;        // 表示监听地址
            address = new InetSocketAddress(ports[i]); // 实例化绑定地址
            initSock.bind(address);                     // 绑定地址
            // 注册选择器，相当于使用accept()方法接收
            initSer.register(selector,SelectionKey.OP_ACCEPT);
            System.out.println("服务器运行，在" + ports[i] + "端口监听。");
        }
```

```
            int keysAdd = 0;                                    // 接收一组SelectionKey
            while ((keysAdd = selector.select()) > 0) {// 选择一组键，相应的通道
                                                                已为IO准备就绪
                Set<SelectionKey> selectedKeys = selector.selectedKeys();
                                                    // 取出全部生成的key
                Iterator<SelectionKey> iter = selectedKeys.iterator();
                                                    // 实例化Iterator
                while (iter.hasNext()) {            // 迭代全部的key
                    SelectionKey key = (SelectionKey) iter.next();
                                                    // 取出每一个SelectionKey
                    if (key.isAcceptable()) {       // 判断客户端是否已经连接上
                        ServerSocketChannel server = (ServerSocketChannel) key
                                .channel();         // 取得Channel
                        SocketChannel client = server.accept();// 接收新连接
                        client.configureBlocking(false);        // 设置成非阻塞
                                                                状态
                        ByteBuffer outBuf = ByteBuffer.allocateDirect(1024);
                                                    // 开辟缓冲区
                        outBuf.put(("当前时间为：" + new Date()).getBytes());
                                                    // 向缓冲区设置内容
                        outBuf.flip();              // 重置缓冲区
                        client.write(outBuf);       // 输出信息
                        client.close();             // 关闭输出流
                    }
                }
                selectedKeys.clear();               // 清除全部的key
            }
        }
    }
```

以上程序运行之后，程序将在 8000、8001、8002、8003、8005、8006 6 个端口进行服务器的监听，等待客户端连接，客户端连接后将返回系统的当前时间。

客户端可以直接使用普通的 Socket 创建或者使用 telnet 命令进行连接。

提示：服务器运行后并不会退出。

在使用 Selector 实现的服务器操作代码中，程序执行完后并不会像传统的 Socket 那样立刻关闭服务器，而是会继续等待下一次的连接。

20.7　本 章 要 点

1．使用 Java 新 IO 可以提升传统 IO 的操作性能。

2．在新 IO 中所有的读、写操作都是通过缓冲区完成，缓冲区中只能容纳特定的数据类型。

3．在缓冲区中使用 position、limit、capacity 表示缓冲区的操作状态。

4．通道提供了双向的读、写操作。

5．使用内存映射可以提升输入流的性能。

6．如果一个线程操作一个文件时不希望其他线程进行访问，则可以通过 FileLock 锁定一个文件。

7．在 Java 新 IO 中可以使用 CharsetEncoder 和 CharsetDecoder 完成编码的转换操作。

第 21 章　Eclipse 开发工具

通过本章的学习可以达到以下目标：

- ☑ 了解 Eclipse 开发工具的作用及下载。
- ☑ 掌握 Eclipse 中 JDT 工具的使用。
- ☑ 掌握 JUnit 测试工具的使用。
- ☑ 掌握 CVS 服务器端及 Eclipse 客户端的配置。

本章视频录像讲解时间为 1 小时 35 分钟，源代码在光盘对应的章节下。

21.1　Eclipse 简介

Eclipse 是一个开放源代码的、基于 Java 的可扩展开发平台。就其本身而言，它只是一个框架和一组服务，用于通过插件组件构建开发环境。

Eclipse 最初是由 IBM 公司开发的替代商业软件 Visual Age for Java 的下一代 IDE 开发环境，2001 年 11 月贡献给开源社区，现在它由非营利软件供应商联盟 Eclipse 基金会（Eclipse Foundation）管理。

提示：Eclipse 提供的是一个平台，主要靠插件进行收费。

Eclipse 本身虽然提供了一个开发的平台，但是此平台上只支持 Java 的开发，所以如果要想开发 Java 的其他程序则必须使用单独的插件才能让开发更加快捷。

读者可以直接从 www.eclipse.org 网站上下载到 Eclipse 的开发工具，如图 21-1 所示。本章使用的 Eclipse 版本是 3.4.1。

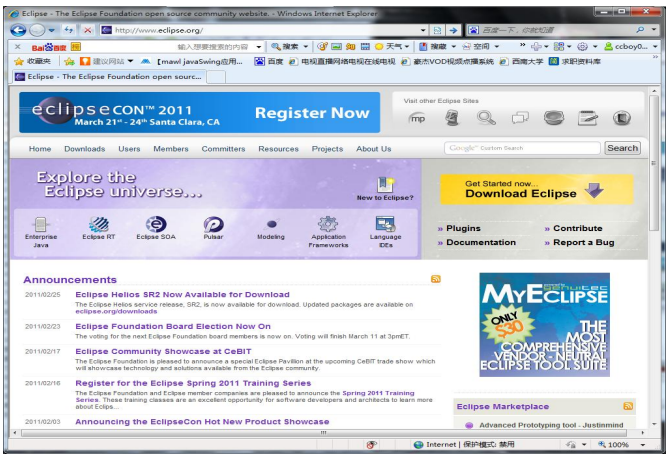


图 21-1　Eclipse 的官方网站

下载下来的 Eclipse 主要包含以下几种开发支持。

- JDT（Java Development Tools）：专门开发 JAVA SE 程序的平台，提供调试、运行、随笔提示等常见功能。
- JUnit：单元测试软件，可以直接对开发的类进行测试。
- CVS 客户端：版本控制软件的连接客户端，使用时需要进行服务器端的配置。
- 插件开发：可以开发 Eclipse 使用的各种插件，丰富开发工具的功能。

21.2 使用 Eclipse

Eclipse 开发包下载完成后，直接进行解压缩即可。解压缩之后的文件目录如图 21-2 所示。

直接运行 eclipse.exe 即可启动 Eclipse 开发工具，启动后的界面如图 21-3 所示。

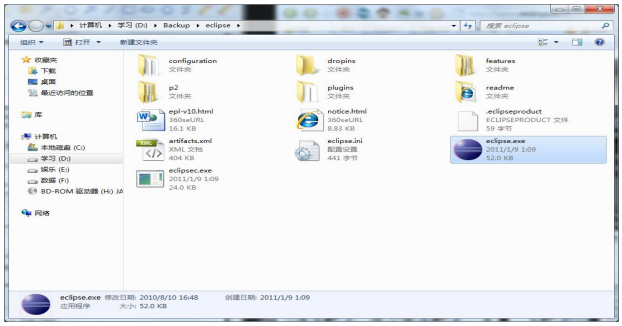


图 21-2 Eclipse 工具解压缩之后的目录

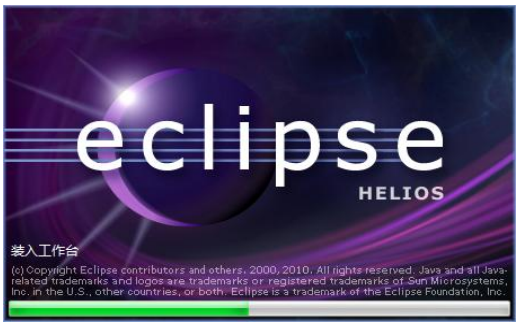


图 21-3 启动 Eclipse

启动之后会询问用户建立工作区的路径，下面将 d:\eclipsedemo 作为工作区，如图 21-4 所示。

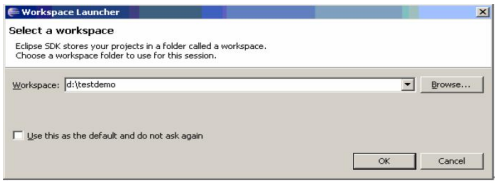


图 21-4 选择工作区

工具启动之后即可进入 Eclipse 的工作区界面，如图 21-5 所示。

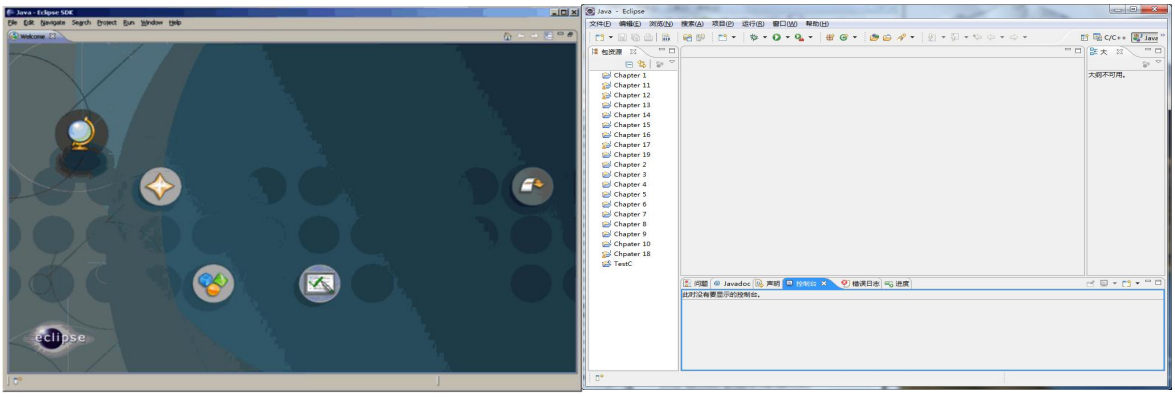


图 21-5 工作区界面

21.3　开发 Java 程序

在 Eclipse 中直接使用 JDT 即可完成 Java 程序的开发，在之前建立好的工作区中建立 Java 项目，选择【File】→【New】→【Java Project】命令，进入如图 21-6 所示的界面。

项目建立完成后，在工作窗口的左边会出现项目的结构树，如图 21-7 所示。

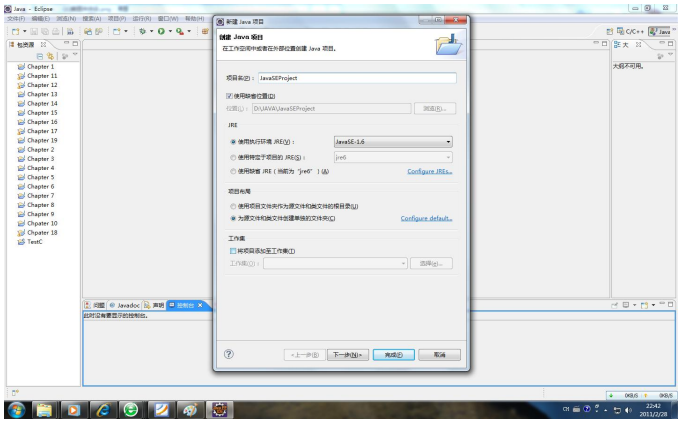


图 21-6　新建 Java 项目

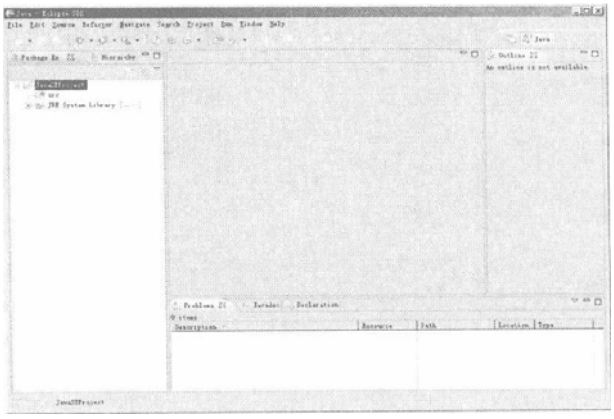


图 21-7　项目建立完成

此时可以在 src 上单击鼠标右键，建立新的 class，类名称为 HelloDemo，建立时选择建立主方法，如图 21-8 所示。建立完成后的工作窗口如图 21-9 所示。

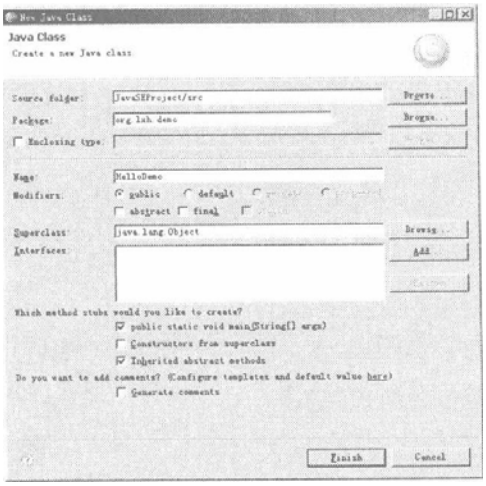


图 21-8　建立新的类

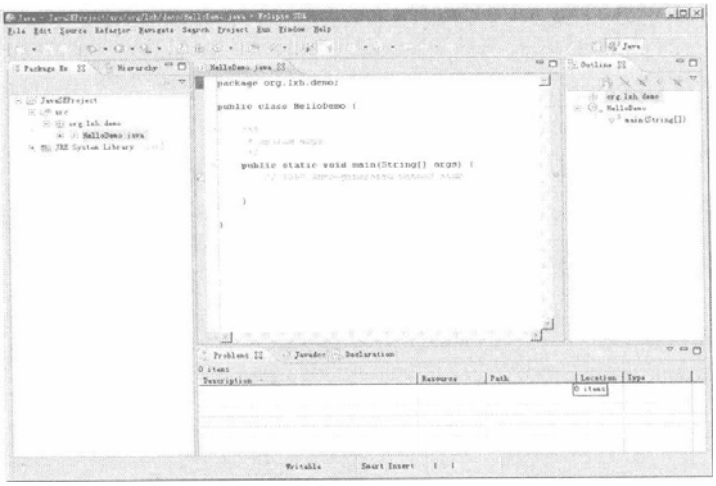


图 21-9　类建立后的工作窗口

范例：编辑 HelloDemo.java

```
package org.lxh.demo;
public class HelloDemo {
    public static void main(String[] args) {
        System.out.println("Hello MLDN!!!");    // 输出信息
    }
}
```

类建立完成后，每次保存都会自动生成 HelloDemo.class 文件，这样用户就不用再手工进行编译了。如果想运行此程序，直接在左边工具栏选择好要执行的类，并在类上单击鼠标右键，在弹出的快捷菜单中选择【Run As】→【Java Application】命令即可。

程序运行后，将弹出【Console】窗口，并在窗口上显示运行效果，如图 21-10 所示。

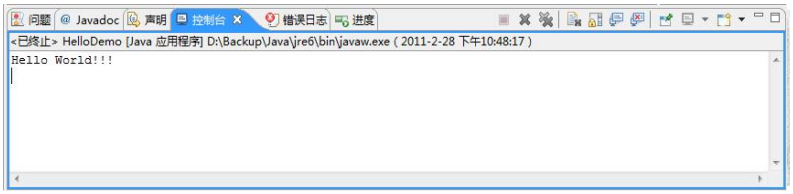


图 21-10　程序的运行效果

在 Eclipse 中也可以对程序进行断点调试的操作，例如，现在有如下的类。

范例：数学计算

```
package org.lxh.debugdemo;
class Math {
    public int div(int i, int j) throws Exception {     // 所有异常抛出
        int temp = 0;                                    // 定义局部变量
        temp = i / j;                                    // 执行除法计算
        return temp;                                     // 返回计算结果
    }
}
public class DebugDemo {
    public static void main(String[] args) {
        Math m = new Math();                             // 实例化Math类对象
        int x = 10;                                      // 定义整型变量，在此
                                                            处加入断点
        int y = 2;                                       // 定义整型变量
        int result = 0;                                  // 保存计算结果
        try {
            result = m.div(x, y);                        // 执行除法计算
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.out.println("计算结果，result = " + result);
    }
}
```

之后在“int x = 10;”的左边条上，选择 Toggle BreakPoint（如图 21-11 所示），这样程序以 debug 调试时执行到此处代码就会进入到调试视图。

选择所要运行的类，单击鼠标右键，在弹出的快捷菜单中选择【Debug As】命令，以调试方式启动。调试方式启动后，会询问是否进入调试窗口，单击【yes】按钮进入到调试窗口，如图 21-12 所示。

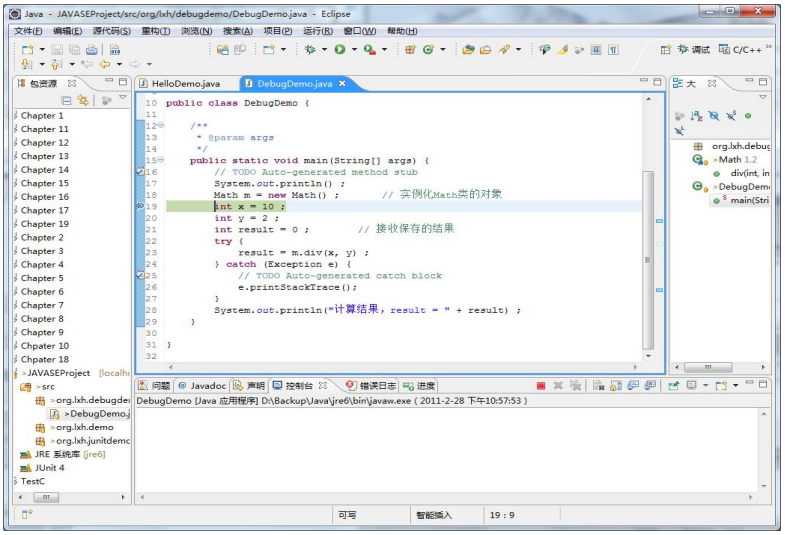


图 21-11　加入断点

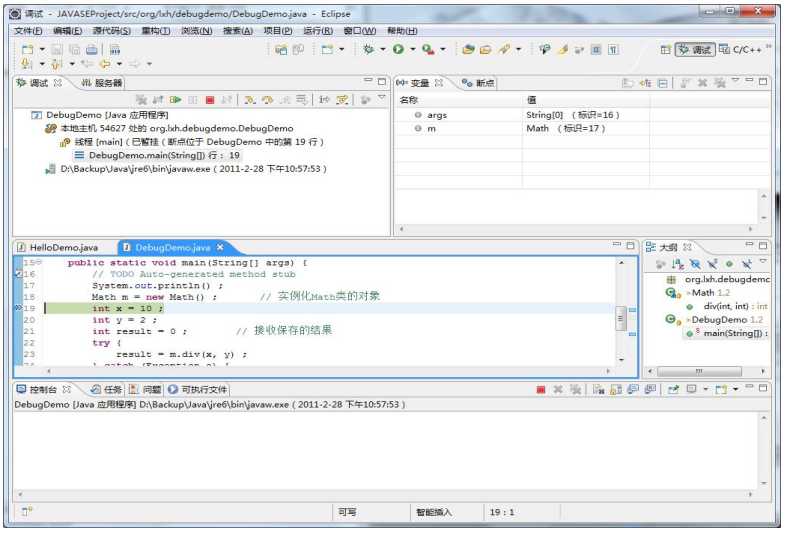


图 21-12　进入到调试窗口

在调试窗口中可以发现，在加入断点的地方程序停了下来，之后用户可以以下两种方式进行调试。

- 单步跳入（Step Info）：进入代码内部观察，对应的快捷键是 F5。
- 单步跳过（Step Over）：只观察代码的运行结果，对应的快捷键是 F6。
- 执行完毕（Resume）：整个代码向后自动执行完毕，对应的快捷键是 F8。

21.4　JUnit 测试工具

JUnit 是由 Erich Gamma 和 Kent Beck 编写的一个回归测试框架（regression testing framework）。JUnit 测试是程序员测试，即白盒测试，因为程序员知道被测试的软件如何（How）完成功能和完成什么样（What）的功能。JUnit 是一套框架，所有需要测试的类直接继承 TestCase 类，即可用 JUnit 进行自动测试。

JUnit 是一个开发源代码的 Java 测试框架，用于编写和运行可重复的测试。在 Eclipse 中已经集成好了此项工具。

例如，在 org.lxh.junitdemo 包中建立了一个 MyMath 的类，代码如下所示。

范例：MyMath 类

```
package org.lxh.junitdemo;
public class MyMath {
    public int div(int i, int j) throws Exception {     // 所有异常抛出
        int temp = 0;                                    // 定义局部变量
        temp = i / j;                                    // 执行除法计算
        return temp;                                     // 返回计算结果
    }
}
```

以上类的功能比较简单，只是完成了一个简单的除法操作，这时即可使用 JUnit 来测试此段代码的结果是否正确。

选择【新建】→【Java】→【junit】命令，建立测试用例或测试站点，如图 21-13 所示。一般来说一个测试用例对应着一个测试程序，一个测试站点对应着一套测试用例，如图 21-14 所示。

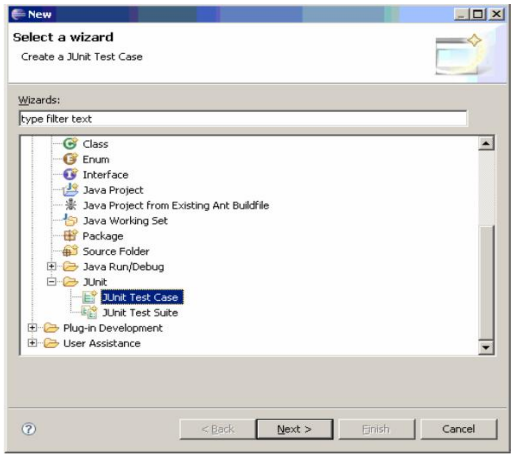


图 21-13　建立 JUnit 测试用例

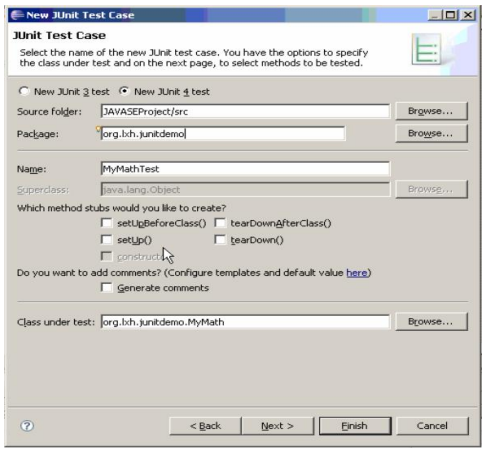


图 21-14　选择测试的程序

之后选择要测试的方法，现在要测试的是 MyMath 类中的 div()方法，如图 21-15 所示。

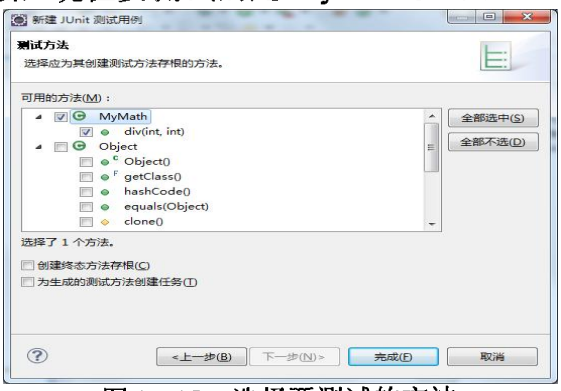


图 21-15　选择要测试的方法

选择完成后，Eclipse 会提示用户是否把 JUnit 的开发包加入到项目中，单击【OK】按钮，如图 21-16 所示。

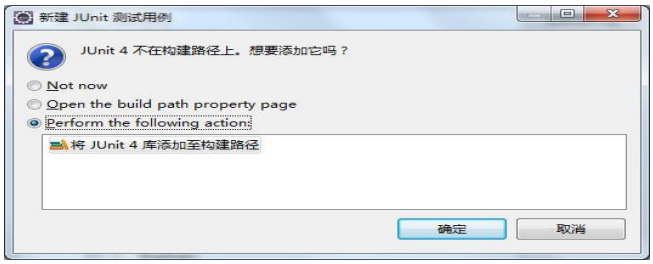


图 21-16　提示是否增加 JUnit 的包

全部完成后，会建立一个 MyMathTest 的类，此类代码如下所示：

```
package org.lxh.junitdemo;
import static org.junit.Assert.*;
import org.junit.Test;
public class MyMathTest {
    @Test
    public void testDiv() {
        fail("Not yet implemented");
    }
}
```

之后在 testDiv()方法中编写要测试的方法，代码如下所示。

范例：修改 MyMathTest 类

```
package org.lxh.junitdemo;
import junit.framework.Assert;
import org.junit.Test;
public class MyMathTest {
    @Test
    public void testDiv() {
        try {
            Assert.assertEquals(new MyMath().div(10,2), 5) ;
                                                    // 判断10/2的值是否是5
            Assert.assertEquals(new MyMath().div(10,3), 3,1) ;
                                                    // 判断10/2的值是否是5
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

类建立完成后，可以直接使用 JUnit 进行代码的测试，运行时以 JUnit 的方式运行，运行的结果可能如图 21-17 或图 21-18 所示。

如果运行正确，则会显示一个 Green Bar，表示测试通过；而如果运行错误（计算的结

果不正确）则会显示 Red Bar。

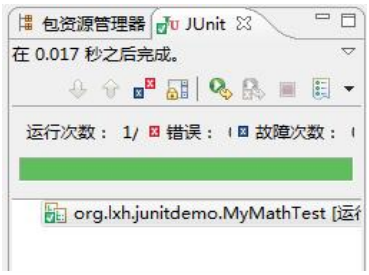


图 21-17　正确的运行



图 21-18　错误的运行

21.5　CVS 客户端的配置

CVS（Concurrent Version System）版本控制系统是一种 GNU 软件包，主要用于在多人开发环境下的源码的维护。实际上 CVS 可以维护任意文档的开发和使用，例如共享文件的编辑修改，而不仅局限于程序设计。CVS 维护的文件类型可以是文本类型也可以是二进制类型。CVS 用 Copy-Modify-Merge（复制、修改、合并）变化表支持对文件的同时访问和修改。它明确地将源文件的存储和用户的工作空间独立开来，并使其并行操作。CVS 基于客户端/服务器的行为使其可容纳多个用户，构成网络也很方便。这一特性使得 CVS 成为位于不同地点的人同时处理数据文件（特别是程序的源代码）时的首选。

21.5.1　CVS 服务器端配置

本章使用 CVSNT 作为 CVS 的服务器端，读者可以直接从 http://www.cvsnt.org 网站上下载，如图 21-19 所示。

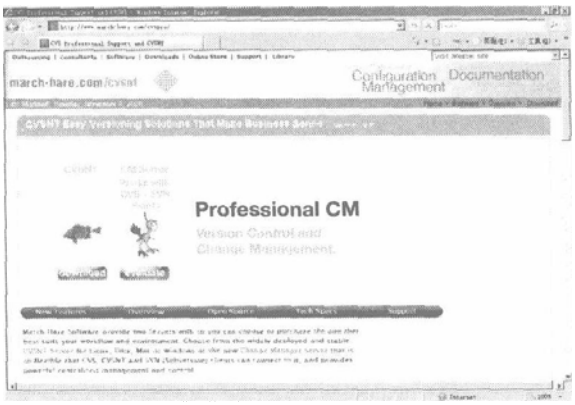


图 21-19　CVSNT 下载

提示：CVS 现在已经被 SVN 所取代。

在当今项目的代码管理中已经很少使用 CVS，应用最广泛的版本控制工具是 SVN(subversion)，它被称为是 CVS 的接班人。

本章所使用到的 CVS 服务器的版本是 cvsnt-server-2.5.04，下载之后直接进行安装（如

图 21-20 所示）即可。

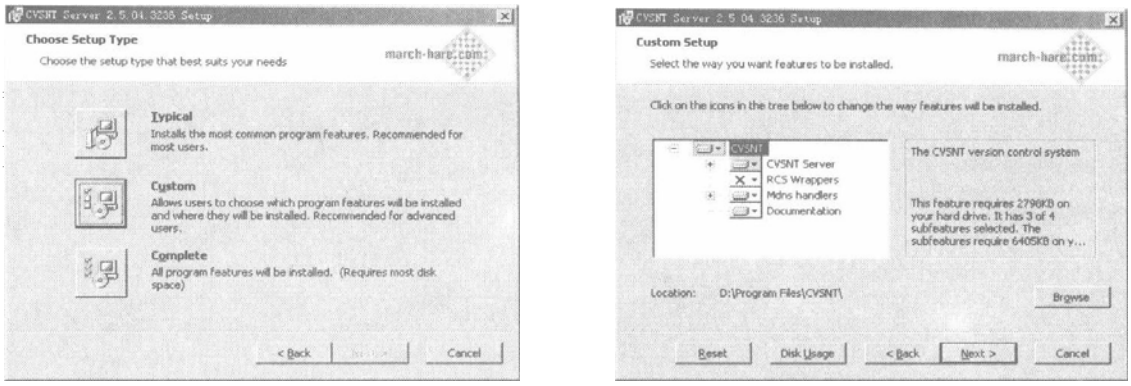


图 21-20　CVSNT 的安装

安装完成之后，在 d 盘上建立一个 uploadpro 文件夹，将所有的开发文件提交到此文件夹中。

启动 CVSNT 控制面板，选择【Repository configuration】选项卡，单击【Add】按钮，将刚才建立的目录配置上，如图 21-21 所示。

按如图 21-22 所示设置添加目录。

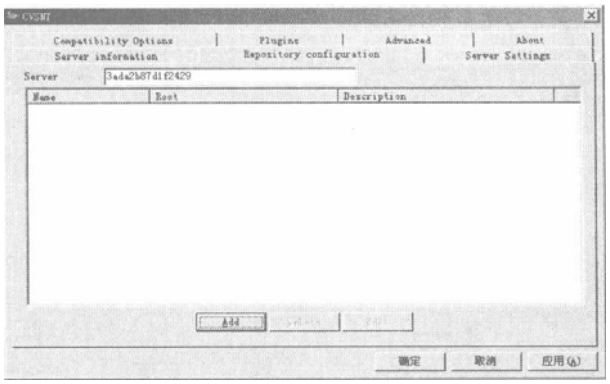


图 21-21　单击【Add】按钮

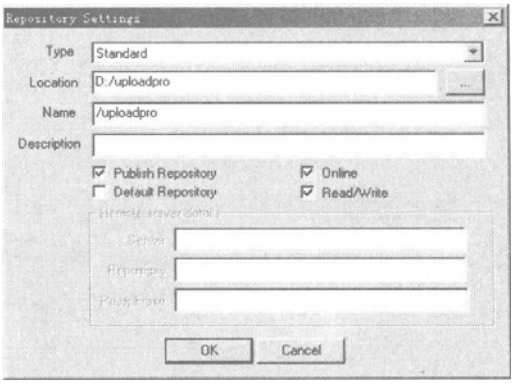


图 21-22　添加目录

如图 21-23 所示，目录添加完成后，选择【Compatibility Options】选项卡，选中 CVSNT Clients 组中的【Respond as cvs 1.11.2 to version r】和【Hide extended log/status infor】复选框，如图 21-24 所示。

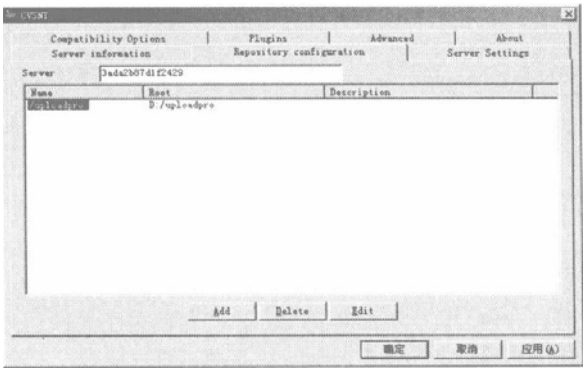


图 21-23　添加完成

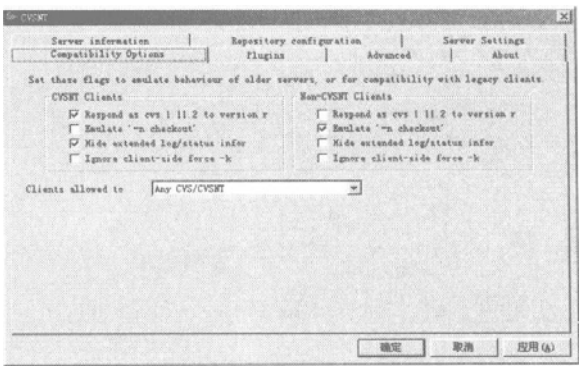


图 21-24　配置 Compatibility Options

此时 CVS 的服务器端已经配置完成，在 Eclipse 中直接将项目提交到 CVS 上即可。现在将 Java SE Project 项目进行共享，选择要共享的项目，单击鼠标右键，在弹出的快捷菜单中选择【Team】→【Share Project】命令，之后配置好 CVS 服务器的相关信息，如图 21-25 所示，之后选择下一步即可将项目共享，共享完项目之后的目录如图 21-26 所示。

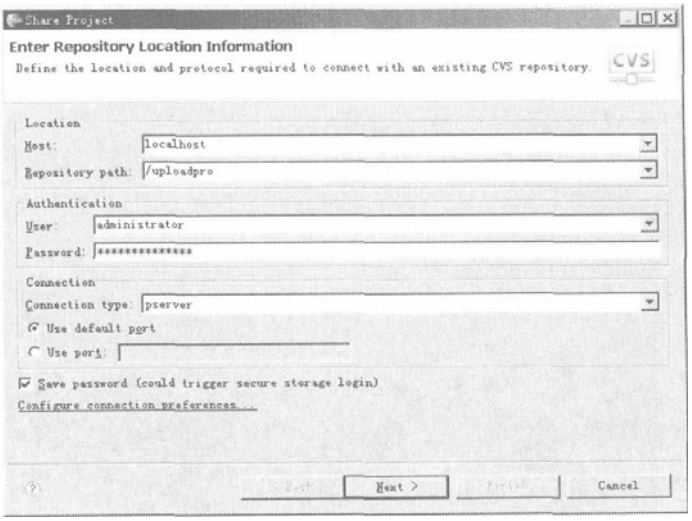


图 21-25　配置 CVS 服务器

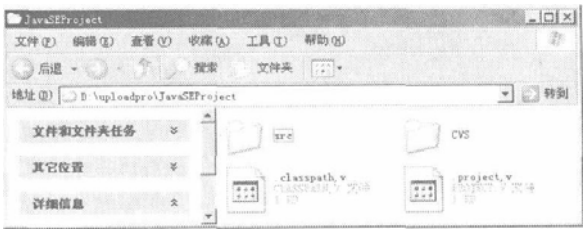


图 21-26　项目共享之后的目录

21.5.2　在 Eclipse 中连接 CVS

为了验证 CVS 是否可以正常使用，可以先删除 Eclipse 中的项目，然后再从 CVS 服务器上直接将此项目下载下来。

选择【File】→【Import】命令，在弹出的窗口中选择【CVS】→【Project from CVS】选项，如图 21-27 所示。单击【Next】按钮后，在弹出的窗口中选择要使用的 CVS 服务器，如图 21-28 所示。

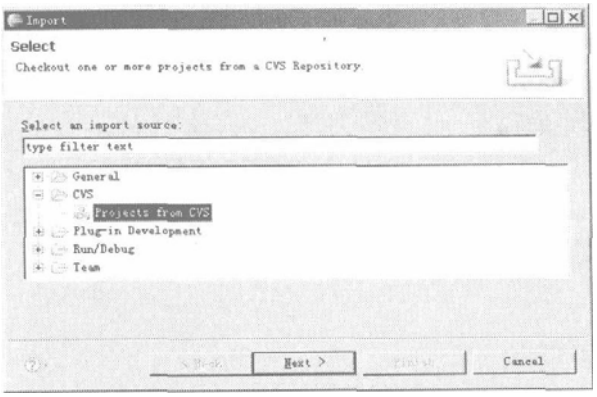


图 21-27　从 CVS 导入项目

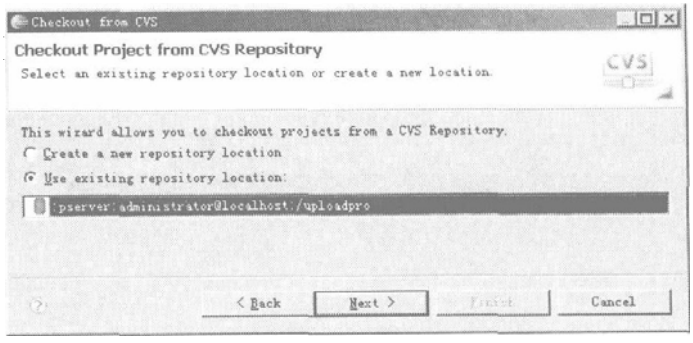


图 21-28　选择要使用的 CVS 服务器

进入到服务器后，可以直接从服务器上选择已经保存的项目，如图 21-29 所示。

此时即可将保存在服务器上的项目下载下来。用户可以直接开发，每次开发时如果修改了一个代码的内容，则需要将代码直接通过【Team】选项提交到服务器上。

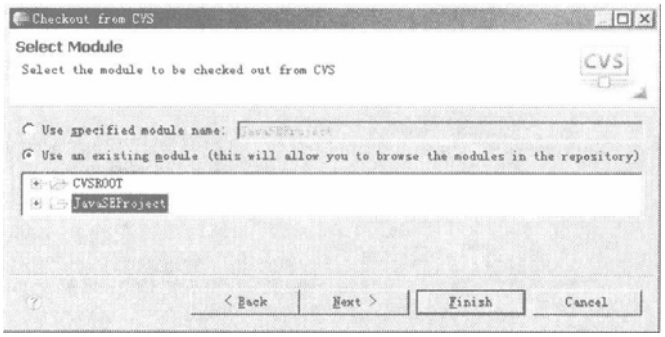


图 21-29　选择要导入的项目

21.6　本 章 要 点

1．Eclipse 是一个开源的开发工具，最早由 IBM 开发。

2．Eclipse 本身提供了 JDT 开发工具，可以使用此工具直接开发 Java 程序，用户在每次编写完成之后，会自动将其编译成相应的 class 文件。

3．JUnit 是一套测试开发包，是专门的白盒测试工具。

4．CVS 是版本控制工具，可以在多人开发时提供程序的版本控制功能，在 Eclipse 中已经提供了 CVS 的客户端，直接配置 CVS 服务器端即可。

参 考 文 献

1．刘宝林．Java 程序设计与案例．北京：高等教育出版社，2004

2．飞思科技产品研发中心．Java TCP/IP 应用开发详解．北京：电子工业出版社，2002

3．李钟尉，马文强，陈丹丹．Java 从入门到精通．北京：清华大学出版社，2008

4．朱喜福．Java 程序设计（第 2 版）．北京：人民邮电出版社，2007

5．迟丽华．Java 程序设计教程．北京：清华大学出版社，2004

6．朱喜福，马涛，魏绍谦．Java 程序设计——示例与习题解析．北京：人民邮电出版社，2004

7．Michael Main．数据结构 Java 语言描述．北京：中国电力出版社，2005

8．洪维恩．Java2 面向对象程序设计．北京：中国铁道出版社，2002