第 1 部分　Java 基础程序设计

- Java 语言介绍
- 简单的 Java 程序
- Java 中的变量与数据类型
- 运算符、表达式与语句
- 循环与选择结构
- 数组与方法的使用

第 1 章　认识 Java

1.1　Java 的历史

Java 来自于 Sun 公司的一个叫 Green 的项目，其原先的目的是为家用电子消费产品开发一个分布式代码系统，这样就可以把 E-mail 发给电冰箱、电视机等家用电器，对它们进行控制，和它们进行信息交流。开始他们准备采用 C++，但 C++太复杂，安全性差，最后基于 C++开发一种新语言 Oak（Java 的前身）。Oak 是一种用于网络的精巧而安全的语言，Sun 公司曾以此投标一个交互式电视项目，但结果被 SGI 打败。于是 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，他们用 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。

Java 技术是由美国 Sun 公司倡导和推出的，它包括 Java 语言和 Java Media APIS、Security APIs、Management APIs、Java Applet、Java RMI、JavaBean、JavaOS、Java Servlet、JavaServer Page 以及 JDBC 等。现把 Java 技术的发展历程简述如下：

- 1990 年，Sun 公司 James Gosling 领导的小组设计了一种平台独立的语言 Oak，主要用于为各种家用电器编写程序。
- 1995 年 1 月，Oak 被改名为 Java；1995 年 5 月 23 日，Sun 公司在 Sun World ‘95 上正式发布 Java 和 HotJava 浏览器。
- 1995 年 8 月至 12 月，Netscape 公司、Oracle 公司、Borland 公司、SGI 公司、Adobe 公司、IBM 公司、AT&T 公司、Intel 公司获得 Java 许可证。
- 1996 年 1 月，Sun 公司宣布成立新的业务部门——JavaSoft 部，以开发、销售并支持基于 Java 技术的产品，由 Alan Baratz 任总裁。同时推出 Java 开发工具包 JDK（Java Development Kit）1.0，为开发人员提供用来编制 Java 应用软件所需的工具。
- 1996 年 2 月，Sun 公司发布 Java 芯片系列，包括 PicoJava、MicroJava 和 UltraJava，并推出 Java 数据库连接 JDBC（Java Database Connectivity）。
- 1996 年 3 月，Sun 公司推出 Java WorkShop。

- 1996 年 4 月，Microsoft 公司、SCO 公司、苹果电脑公司（Apple）、NEC 公司等获得 Java 许可证。Sun 公司宣布允许苹果电脑、HP、日立、IBM、Microsoft、Novell、SGI、SCO、Tamdem 等公司将 Java 平台嵌入到其操作系统中。
- 1996 年 5 月，HP 公司、Sybase 公司获得 Java 许可证。北方电讯公司宣布把 Java 技术和 Java 微处理器应用到其下一代电话机中的计划。5 月 29 日，Sun 公司在旧金山举行第一届 JavaOne 世界 Java 开发者大会，业界人士踊跃参加。Sun 公司在大会上推出一系列 Java 平台新技术。
- 1996 年 8 月，JavaWorkShop 成为 Sun 公司通过互联网提供的第一个产品。
- 1996 年 9 月，Addison-Wesley 和 Sun 公司推出 Java 虚拟机规范和 Java 类库。
- 1996 年 10 月，德州仪器等公司获得 Java 许可证。Sun 公司提前完成 JavaBean 规范并发布，同时发布第一个 Java JIT（Just-In-Time）编译器，并打算在 Java WorkShop 和 Solaris 操作系统中加入 JIT。10 月 29 日，Sun 公司发布 Java 企业计算技术，包括 JavaStation 网络计算机、65 家公司发布的 85 个 Java 产品及应用、7 个新的 Java 培训课程及 Java 咨询服务、基于 Java 的 Solstice 互联网邮件软件、新的 Java 开发者支持服务、HotJava Views 演示、Java Tutor、Java Card API 等。Sun 公司宣布完成 Java Card API 规范，这是智能卡使用的第一个开放 API。Java Card 规范将把 Java 能力赋予全世界亿万张智能卡。
- 1996 年 11 月，IBM 公司获得 JavaOS 和 HotJava 许可证。Novell 公司获得 Java WorkShop 许可证。Sun 公司和 IBM 公司宣布双方就提供 Java 化的商业解决方案达成一项广泛协议，IBM 公司同意建立第一个 Java 检验中心。
- 1996 年 12 月，Xerox 等公司获得 Java 或 JavaOS 许可证。Sun 公司发布 JDK1.1、Java 商贸工具包、JavaBean 开发包及一系列 Java APIs；推出一个新的 JavaServer 产品系列，其中包括 Java Web Server、Java NC Server 和 JavaServer Toolkit。Sun 公司发布 100%纯 Java 计划，得到百家公司的支持。
- 1997 年 1 月，SAS 等公司获得 Java 许可证。Sun 公司交付完善的 JavaBean 开发包，这是在确定其规范后不到 8 个月内完成的。
- 1997 年 2 月，Sun 公司和 ARM 公司宣布同意使 JavaOS 运行在 ARM 公司的 RISC 处理器架构上。Informix 公司宣布在其 Universal Server 和其他数据库产品上支持 JDK1.1。Netscape 公司宣布其 Netscape Communicator 支持所有 Java

化的应用软件和核心 API。

- 1997 年 3 月，HP 公司获得 Java WorkShop 许可证，用于其 HP-UX 操作系统。西门子、AG 公司等获得 Java 许可证。日立半导体公司、Informix 公司等获得 JavaOS 许可证。Novell 公司获得 Java Studio 许可证。Sun 公司发售的 JavaOS 1.0 操作系统，这是一种在微处理器上运行 Java 环境的最小、最快的方法，可提供给 JavaOS 许可证持有者使用。Sun 公司发售 HotJava Browser 1.0，这是一种 Java 浏览器，可以方便地按需编制专用的信息应用软件，如客户自助台和打上公司牌号的网络应用软件。
- 1996 年 6 月，Sun 公司发布 JSP1.0，同时推出 JDK1.3 和 Java Web Server 2.0。
- 1999 年 11 月，Sun 公司发布 JSP1.1，同时推出 JSWDK1.0.1 和 Java Servlet 2.2。
- 2000 年 9 月，Sun 公司发布 JSP1.2 和 Java Servlet 2.3 API。

1.2 Java 的现状

Java 是 Sun 公司推出的新一代面向对象程序设计语言，特别适于 Internet 应用程序开发，它的平台无关性直接威胁到 Wintel 的垄断地位，这表现在以下几个方面：

- 计算机产业的许多大公司购买了 Java 许可证，包括 IBM、Apple、DEC、Adobe、SiliconGraphics、HP、Oracle、TOSHIBA 以及 Microsoft。这一点说明，Java 已得到了业界的认可。
- 众多的软件开发商开始支持 Java 软件产品。例如 Inprise 公司的 JBuilder、Sun 公司自己做的 Java 开发环境 JDK 与 JRE。Sysbase 公司和 Oracle 公司均已支持 HTML 和 Java。
- Intranet 正在成为企业信息系统最佳的解决方案，而其中 Java 将发挥不可替代的作用。Intranet 的目的是将 Internet 用于企业内部的信息类型，它的优点是便宜、易于使用和管理。用户不管使用何种类型的机器和操作系统，界面是统一的 Internet 浏览器，而数据库、Web 页面、Applet、Servlet、JSP 则存储在 Web 服务器上，无论是开发人员还是管理人员，或是用户都可以受益于该解决方案。

1.3 Java 的特点

1.3.1 Java 语言的优点

Java 语言是一种优秀的编程语言。它最大的优点就是与平台无关，在 Windows 9x、Windows NT、Solaris、Linux、MacOS 以及其它平台上，都可以使用相同的代码。“一次编写，到处运行”的特点，使其在互联网上被广泛采用。

由于 Java 语言的设计者们十分熟悉 C++语言，所以在设计时很好地借鉴了 C++语言。可以说，Java 语言是一种比 C++语言“还面向对象”的一种编程语言。Java 语言的语法结构与 C++语言的语法结构十分相似，这使得 C++程序员学习 Java 语言更加容易。

当然，如果仅仅是对 C++改头换面，那么就不会有今天的 Java 热了。Java 语言提供的一些有用的新特性，使得使用 Java 语言比 C++语言更容易写出“无错代码”。

这些新特性包括：

1、 提供了对内存的自动管理，程序员无需在程序中进行分配、释放内存，那些可怕的内存分配错误不会再打扰设计者了；

2、 去除了 C++语言中的令人费解、容易出错的“指针”，用其它方法来进行弥补；

3、 避免了赋值语句（如 a = 3）与逻辑运算语句（如 a = = 3）的混淆；

4、 取消了多重继承这一复杂的概念。

Java 语言的规范是公开的，可以在 http://www.sun.com 上找到它，阅读 Java 语言的规范是提高技术水平的好方法。

1.3.2 Java 语言的关键特性

Java 语言有许多有效的特性，吸引着程序员们，最主要的有以下几个：

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。Java 语言省略了 C++语言中所有的难以理解、容易混淆的特性，例如头文件、指针、结构、单元、运算符重载、虚拟基础类等。它更加严谨、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，简直就是美梦成真的好事！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就可以在不同平台上运行了。

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了“面向对象的系统分析”、“面向对象的系统设计”、“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。但它正是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”的宏伟目标的。为了达到目标，牺牲效率还是值得的，况且，现在的计算机技术日新月异，运算速度也越来越快，用户是不会感到太慢的。

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 通信程序十分简单，使用它比使用任何其它语言都简

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，简称 JSP）、Servlet 等等手段来构建更丰富的网页。

6.拥有较好的性能

正如前面所述，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 语言采用了两种手段，使得其性能还是不错的。

A、 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；

B、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

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扼杀在摇蓝之中。经常有许多在其它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，这样使得 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。多线程处理能力使得程序能够具有更好的交互性、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很简单。

9.具有较高的安全性

由于 Java 语言在设计时，在安全性方面考虑很仔细，做了许多探究，使得 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“世界上没有绝对的安全”这一真理是不会因为某人的许诺而失灵验的。

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。

而且由于 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：

A、 在 Java 语言中，可以简单、直观地查询运行时的信息；

B、 可以将新代码加入到一个正在运行的程序中去。

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，在 Niklaus Wirth 实现的 Pascal 语言中就采用了这种降低一些性能，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，使得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。

1.4 Java 虚拟机(JVM)

Java 虚拟机(JVM)是可运行 Java 代码的假想计算机。只要根据 JVM 规范描述将解释器移植到特定的计算机上，就能保证经过编译的任何 Java 代码能够在该系统上运行。如图 1-1 所示：

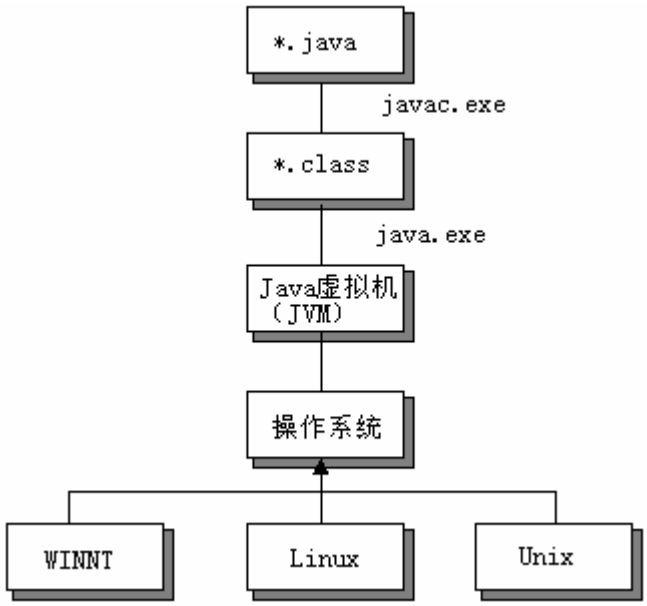


图 1-1　Java 虚拟机

从图 1-1 中不难明白 JAVA 可以实现可移植性的原因，只要在操作系统上（WINNT、Linux、Unix）植入 JVM（Java 虚拟机），JAVA 程序就具有可移植性，也符合 SUN 公司提出的口号“Write Once, Run Anywhere”（“一次编写，处处运行”）。

目前，Java 技术的架构包括以下三个方面：

- J2EE (Java 2 Platform Enterprise Edition) 企业版，是以企业为环境而开发应用程序的解决方案。
- J2SE (Java 2 Platform Stand Edition) 标准版，是桌面开发和低端商务应用的解决方案。
- J2ME (Java 2 Platform Micro Edition) 小型版，是致力于消费产品和嵌入式设备的最佳解决方案。

1.5　JDK 的安装及环境变量的配置

要开发 Java 程序首先必须要配置好环境变量，而 Java 的运行环境的配置比较麻烦，相信有些读者也会有这种体会，下面来看一下 JDK 的安装过程。在这里 JDK 选用的是 J2SDK1.4.2 版本。

安装分为两个步骤：

1、 首先要准备好 JDK 的安装文件：j2sdk-1_4_0_03-windows-i586 ;

2、 配置环境变量 path。

先来看一下步骤一的安装过程：

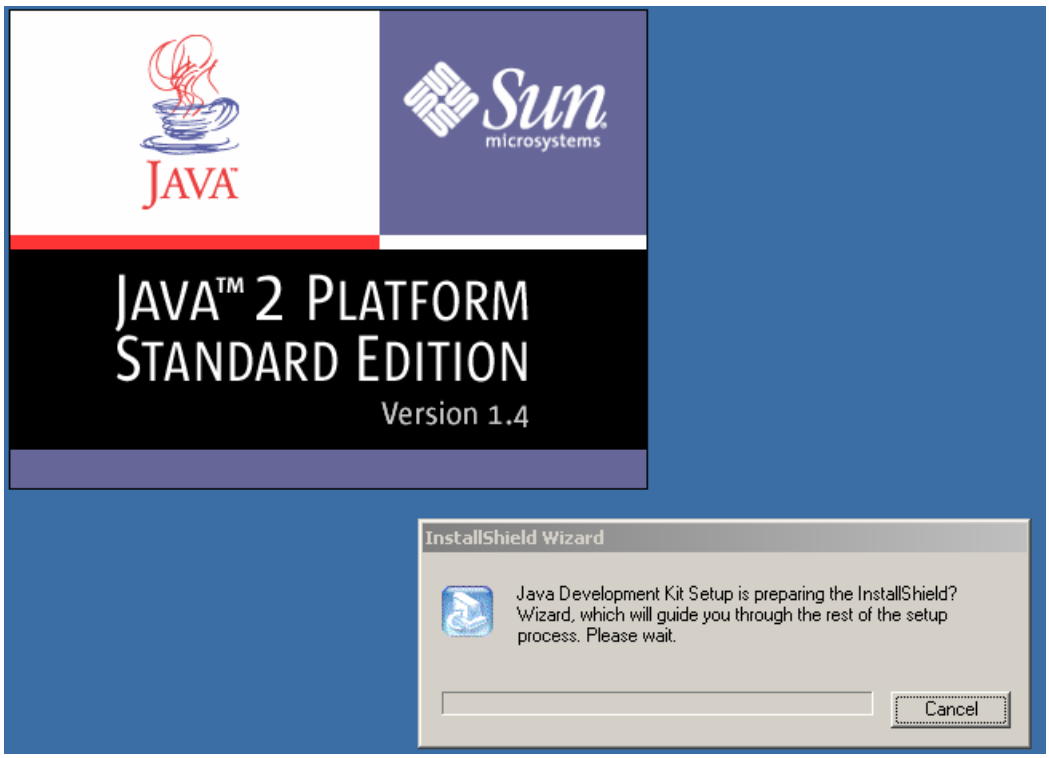


图 1-2　　启动 JDK 安装程序

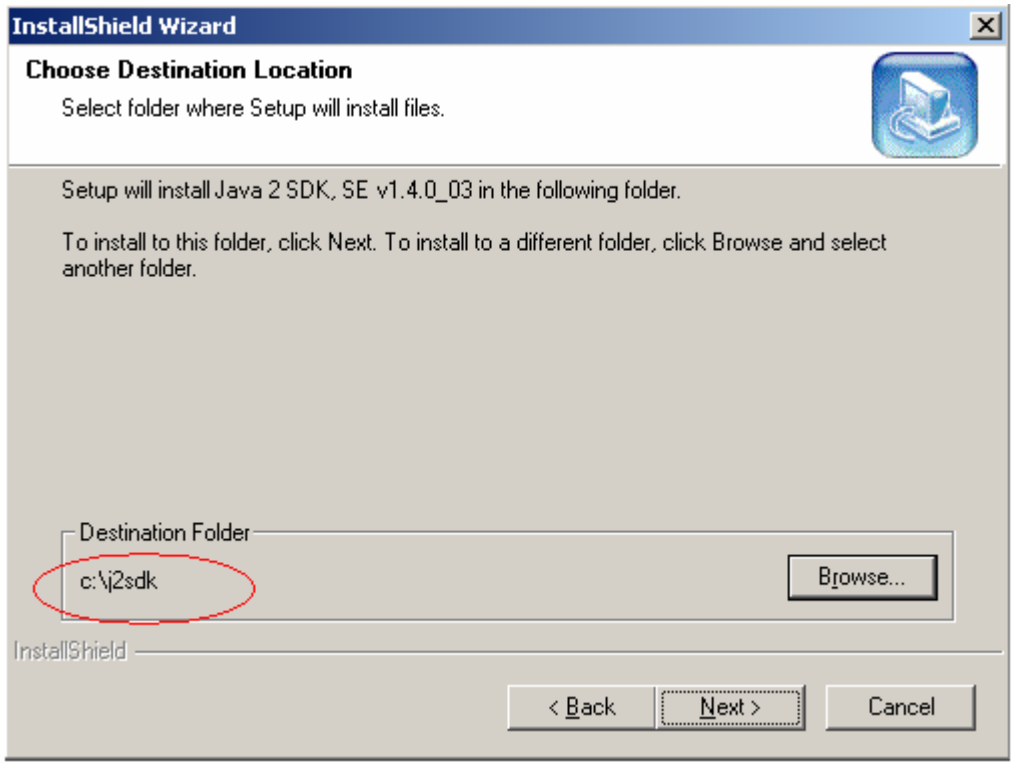


图 1-3　将安装路径设置为 c:\j2sdk

之后走默认的安装设置即可。

从图 1-1 可以看出，在编译 Java 程序时需要用到 Javac 这个命令，执行 Java 程序需要 java 这个命令，而这两个命令并不是 windows 自带的命令，所以使用它们的时候需要配置好环境变量，这样就可以在任何的目录下使用这两个命令了。

那么该如何设置环境变量呢？

在我的电脑上点击右键——>选择属性——>选择高级——>环境变量——>path。

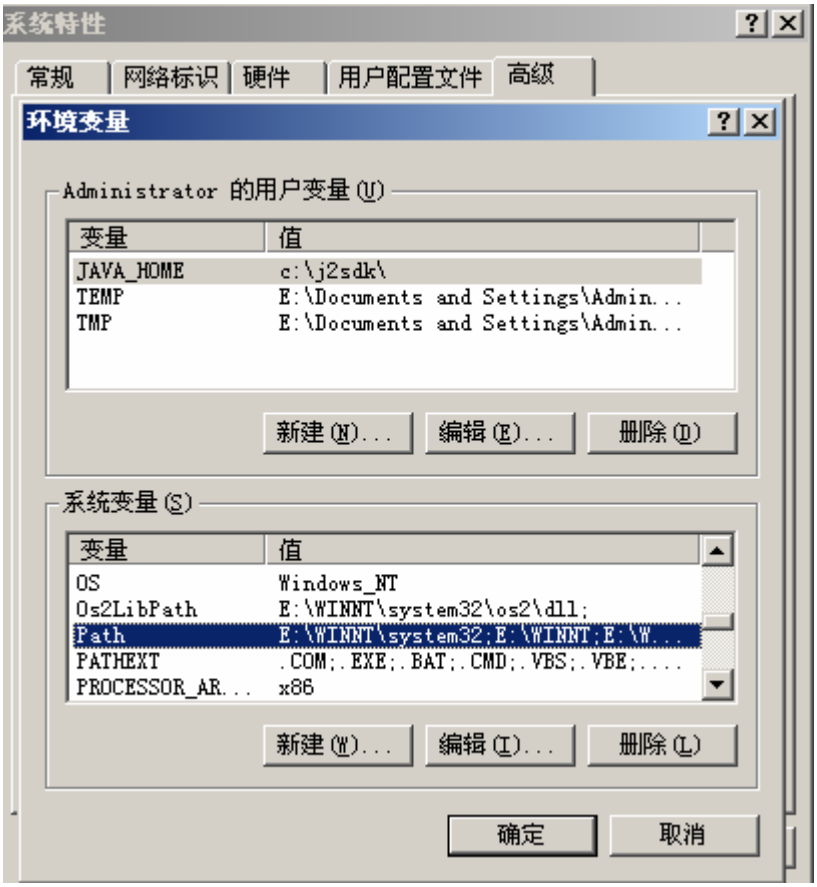


图 1-4　系统环境变量的配置

在 path 后面加上 c:\j2sdk\bin;

c:\j2sdk 是安装 JDK 的路径，如果记不清楚了，可回去看一下图 1-3。

注意：

在这里使用的是 windows 2000 操作系统，至于其他的操作系统，如 windows98 在设置环境变量的时与 windows2000 的配置有许多不同，如果读者感兴趣，可以去查阅其他的资料。

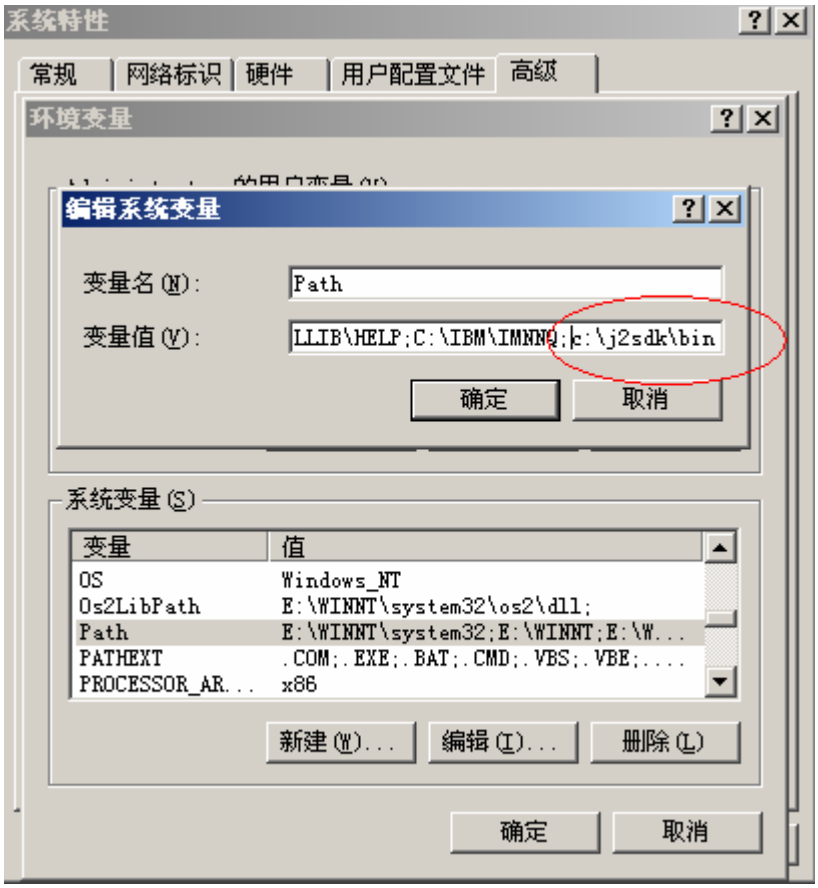


图 1-5　添加环境变量

这样就可以在任何目录下使用 javac 和 java 这两个命令了。

1.6　编写第一个 Java 程序

说了这么多，现在就自己来动手编写一个 Java 的程序来亲自感受一下 Java 语言的基本形式。需要说明的是，JAVA 程序分为两种形式：一种是网页上使用的 Applet 程序（Java 小程序），另一种是 Appliction 程序（即：Java 应用程序），本书主要讲解的是 Java Application 程序。

范例：Hello.java

```
01  public class Hello
02  {
03      // 是程序的起点，所有程序由此开始运行
04      public static void main(String args[])
05      {
06          // 此语句表示向屏幕上打印"Hello World !"字符串
07          System.out.println("Hello World !");
08      }
09  }
```

将上面的程序保存为 HelloCareers.java 文件，并在命令行中输入 javac Hello.java，没有错误后输入 java Hello。

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，在后面的章节中会详细给读者讲解 main 方法的各个组成部分。

上面的程序如果暂时不明白也没有关系，读者只要将程序一点一点都敲下来，之后按照步骤编译、执行，就可以了，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。

1.7 classpath 的指定

在 java 中可以使用 set classpath 命令指定 java 类的执行路径。下面通过一个实验来了解 classpath 的作用，假设这里的 Hello.class 类位于 c:\盘下。

在 D 盘下的命令行窗口执行下面的指令：

set classpath=c:

之后在 D 盘根目录下执行：java Hello，如下图所示：

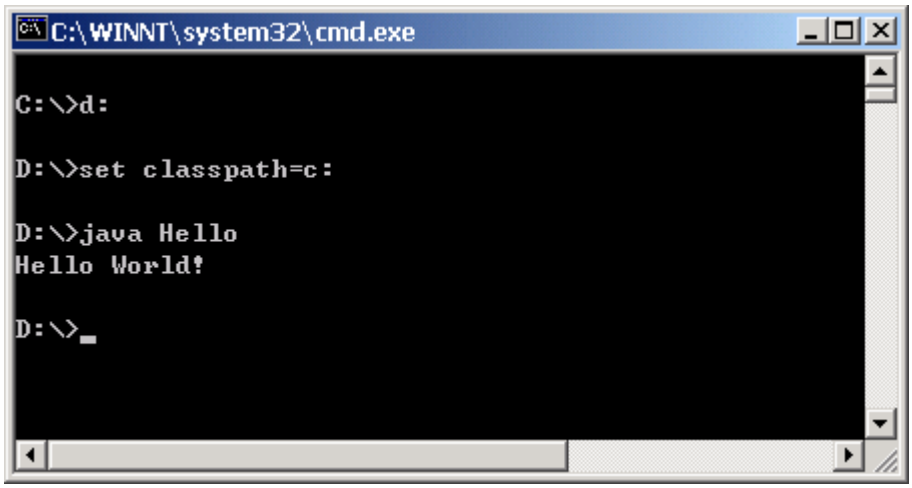


图 1-6

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，但是却也可以用 java Hello 执行 Hello.class 文件，之所以会有这种结果，就是因为在 D 盘中使用 set classpath 命令，将类的查找路径指向了 C 盘，所以在运行时，会从 C 盘下开始查找。

小提示:

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:

set classpath=.

• 本章摘要：

1、 Java 程序比较特殊，它必须先经过编译，然后再利用解释的方式来运行。

2、 Byte-codes 最大的好处是——可越平台运行，可让“一次编写，处处运行”成为可能。

3、 使用 classpath 可以指定 class 的运行路径。

第 2 章　简单的 Java 程序

从本章开始，就要正式学习 Java 语言的程序设计，除了认识程序的架构外，本章还介绍了修饰符、关键字以及一些基本的数据类型。通过简单的范例，让读者了解到检测与提高程序可读性的方法，以培养读者正确的程序编写观念与习惯。

2.1　一个简单的例子

首先来看一个简单的 Java 程序。在介绍程序的内容之前，先简单回顾一下第一章讲解的例子，之后再来看下面这个程序，试试看是否看得出它是在做哪些事情！

范例：TestJava2_1.java

```
01  // TestJava2_1.java，java 的简单范例
02  public class TestJava2_1
03  {
04      public static void main(String args[])
05      {
06          int num ;       // 声明一个整型变量 num
07          num = 3 ;       // 将整型变量赋值为 3
08          // 输出字符串，这里用"+" 号连接变量
09          System.out.println("这是数字 "+num);
10          System.out.println("我有 "+num+" 本书！");
11      }
12  }
```

输出结果：

```
这是数字 3
我有 3 本书!
```

如果现在看不懂上面的这个程序也没有关系，先将它敲进 Java 编辑器里，将它存盘、编辑、运行，就可以看到上面的输出结果。

从上面的输出结果中可以看出 System.out.println()的作用，就是输出括号内所包含的文字，至于 public、class、static、void 这些关键字的意思，将在以后的章节中再做更深入一层的探讨。

程序说明：

1、 第 1 行为程序的注释，Java 语言的注释是以“/ /”标志开始的，注释有助于对程序的阅读与检测，被注释的内容在编译时不会被执行。

2、 第 2 行 public class TestJava2_1 中的 public 与 class 是 Java 的关键字，class 为“类”的意思，后面接上类名称，在本程序中取名为 TestJava2_1。public 则是用来表示该类为公有，也就是在整个程序里都可以访问到它。

需要注意的是，如果将一个类声明成 public，则也要将文件名称取成和这个类一样的名称，如图 2-1 所示。本例中的文件名为 TestJava2_1.java，而 public 之后所接的类名称也为 TestJava2_1。也就是说，在一个 Java 文件里，最多只能有一个 public 类，否则.java 的文件便无法命名。

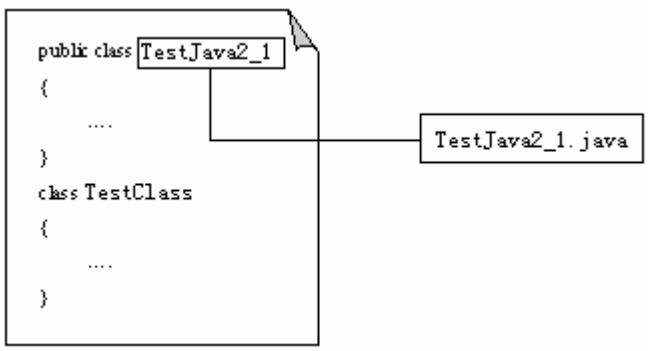


图 2-1　如果将类声明成 public，则也要将文件名称取成和这个类一样的名称

3、 第 4 行 public static void main(String args[])为程序运行的起点。第 4~10 行的功能类似于一般程序语言中的函数（function），但在 Java 中称之为 method（方法）。因此 C 语言里的 main()函数（主函数），在 Java 中则被称为 main() method（主方法）。

4、 main() method 的主体（body）从第 5 行的左大括号“{”到第 11 行的右大括号“}”

为止。每一个独立的 Java 程序一定要有 main() method 才能运行，因为它是程序开始运行的起点。

5、 第 6 行“int num;”的目的是声明 num 为一个整数类型的变量。在使用变量之前必须先声明其类型。

6、 第 7 行“num=3;”为一赋值语句，即把整数 2 赋给存放整数的变量 num。

7、 第 8 行的语句为：

System.out.println("这是数字 "+num);

程序运行时会在显示器上输出引号（””）内所包含的内容。包括“这是数字”和整数变量 num 所存放的值两部分内容。

8、 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。

9、 第 10 行的右大括号则告诉编译器 main() method 到这儿结束。

10、第 11 行的右大括号则告诉编译器 class TestJava2_1 到这儿结束。

这里只是简单的介绍了一下 TestJava2_1 这个程序，相信读者已经对 Java 语言有了初步的了解。TestJava2_1 程序虽然很短，却是一个相当完整的 Java 程序！在后面的章节中，将会对 Java 语言的细节部分，做详细的讨论。

2.2 简单的 Java 程序解析

本节将仔细探讨 Java 语言的一些基本规则及用法。

2.2.1 类（class）

Java 程序是由类（class）所组成，至于类的概念在以后会有详细讲解，读者只要先记住所有的 Java 程序都是由类组成的就可以了。下面的程序片段即为定义类的典型范例：

```
public class Test    // 定义 public 类 Test
{
    …
}
```

上面的程序定义了一个新的 public 类 Test，这个类的原始程序的文件名称应取名为 Test.java。类 Test 的范围由一对大括号所包含。public 是 Java 的关键字，指的是对于类的访问方式为公有。

需要读者注意的是，由于 Java 程序是由类所组成，因此在完整的 Java 程序里，至少需要有一个类。此外，本书曾在前面提到过在 Java 程序中，其原始程序的文件名不能随意命名，必须和 public 类名称一样，因此在一个独立的原始程序里，只能有一个 public 类，却可以有许多 non-public 类。

此外，若是在一个 Java 程序中没有一个类是 public，那么该 Java 程序的文件名就可以随意命名了。

2.2.2 大括号、段及主体

将类名称定出之后，就可以开始编写类的内容。左大括号“{”为类的主体开始标记，而整个类的主体至右大括号“}”结束。每个命令语句结束时，必须以分号“;”做结尾。当某个命令的语句不只一行时，必须以一对大括号“{}”将这些语句包括起来，形成一个程序段（segment）或是块（block）。

再以一个简单的程序为例来说明什么是段与主体。若是暂时看不懂 TestJava2_2

这个程序，也不用担心，慢慢的都会讲到该程序中所用到的命令。在下面的程序中，可以看到 main() method 的主体以左右大括号包围起来；for 循环中的语句不只一行，所以使用左右大括号将属于 for 循环的段内容包围起来；整个程序语句的内容又被第 3 与第 13 行的左右大括号包围，这个块属于 public 类 TestJava2_2 所有。此外，应该注意到每个语句结束时，都是以分号作为结尾。

范例：TestJava2_2.java

```
01      // TestJava2_2，简单的 Java 程序
02      public class TestJava2_2
03      {
04          public static void main(String args[])
05          {
06              int x ;
07              for(x=1;x<3;x++)
08              {
09                  System.out.println(x+" * "+x);
10                  System.out.println(" = "+x*x);
11              }
12          }
13      }
```

for循环所属于的段　main() method的主体　public 类TestJava2_2主体

输出结果：

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

2.2.3 程序运行的起始点 —— main() method

Java 程序是由一个或一个以上的类组合而成，程序起始的主体也是被包含在类之中。这个起始的地方称为 main()，用左右大括号将属于 main()段内容包围起来，称之

为 method（方法）。

main()方法为程序的主方法，在一个 Java 程序中有且只能有一个 main()方法，它是程序运行的开端，通常看到的 main() method 如下面的语句片段所示：

```
public static void main(String args[])        // main() method，主程序开始
{
    …
}
```

如前一节所述，main() method 之前必须加上 public static void 这三个标识符。public 代表 main()公有的 method；static 表示 main()在没有创建类对象的情况下，仍然可以被运行；void 则表示 main()方法没有返回值。Main 后的括号（）中的参数 String args[]表示运行该程序时所需要的参数，这是固定的用法，如果现在不了解也没有关系，在以后的章节中会一一介绍。

2.2.4 Java 程序的注释

为程序添加注释可以用来解释程序的某些语句的作用和功能，提高程序的可读性。也可以使用注释在原程序中插入设计者的个人信息。此外，还可以用程序注释来暂时屏蔽某些程序语句，让编译器暂时不要处理这部分语句，等到需要处理的时候，只需把注释标记取消就可以了，Java 里的注释根据不同的用途分为三种类型：

- **单行注释**
- **多行注释**
- **文档注释**

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。如下所示：

```
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，如下所示：

```
/*
    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，并能包含进如 javadoc 之类的工具生成的文档里，用以说明该程序的层次结构及其方法。

2.2.5 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，不能是 Java 中的保留关键字。

- 下面是合法的标识符：

  yourname

  your_name

  _yourname

  $yourname

- 下面是非法的标识符：

  class

  67.9

  Hello Careers

！小提示：

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，所以在这里提醒读者，标识符最好永远用字母开头，而且尽量不要包含其他的符号。

2.2.6 Java 的关键字

和其他语言一样，Java 中也有许多保留关键字，如 public、static 等，这些保留关键字不能当作标识符使用。下面列出了 Java 中的保留关键字，这些关键字并不需要读者去强记，因为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。

Java 中的保留关键字：

abstract	boolean	break	byte	case	catch
char	class	continue	default	do	double
else	extend	false	final	finally	float
for	if	implement	import	instanceof	int
interface	long	native	new	null	package
private	protected	public	return	short	static
synchronized	super	this	throw	throws	transient
true	try	void	volatile	while	

要特别注意的是，虽然 goto、const 在 Java 中并没有任何意义，却也是保留字，与其它的关键字一样，在程序里不能用来做为自定义的标识符。

2.2.7 变量

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，而使用变量之前必须先声明它所预保存的数据类型。接下来，来看看在 Java 中变量的使用规则。

2.2.7.1 变量的声明

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，

在程序中即可写出如下所示的语句：

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

2.2.7.2　变量的数据类型

除了整数类型之外，Java 还提供了多种其它的数据类型。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）等，除了这些数据类型外，还有字符型（char）或字符串型（String）。关于这些数据类型，本书在第三章中有详细的介绍。

2.2.7.3　变量名称

读者可以依据个人的喜好来决定变量的名称，这些变量的名称不能使用到 Java 的关键字。通常会以变量所代表的意义来取名（如 num 代表数字）。当然也可以使用 a、b、c 等简单的英文字母代表变量，但是当程序很大时，需要的变量数量会很多，这些简单名称所代表的意义就比较容易忘记，必然会增加阅读及调试程序的困难度。

2.2.7.4　变量名称的限制

同 2-2-5 所述标识符的名称限制。

2.2.8 变量的设置

给所声明的变量赋予一个属于它的值，用等号运算符（=）来实现。具体可使用如下所示的三种方法进行设置：

方法 1 —— 在声明变量的时设置

举例来说，在程序中声明一个整数的变量 num，并直接把这个变量赋值为 2 ，可以在程序中写出如下的语句：

```
int num = 2 ;        // 声明变量，并直接设置
```

方法 2 —— 声明后再设置

一般来说也可以在声明后再给变量赋值。举例来说，在程序中声明整数的变量 num1、num2 及字符变量 ch，并且给它们分别赋值，在程序中即可写出如下面的语句：

```
int num1,num2 ;  // 声明变量
char c ;
num1 = 2 ;           // 赋值给变量
num2 = 3 ;
ch = 'z' ;
```

方法 3 —— 在程序中的任何位置声明并设置

以声明一个整数的变量 num 为例，可以等到要使用这个变量时，再给它赋值：

```
int num ;      // 声明变量
…
num = 2 ;      // 用到变量时，再赋值
```

2.2.9 println()

读者会发现从第 1 章开始，所有的例题中出现了不少次的“System.out.println()”，

在本节中，就先了解一下 println()。至于详细的使用方法，在第三章会讲到。

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。其后所连接的 println，是由 print 与 line 所组成的，意义是将后面括号中的内容打印在标准输出设备——显示器上。左、右括号之间的内容，即是欲打印到显示器中的参数，参数可以是字符、字符串、数值、常量或是表达式，参数与参数之间以括号作为间隔。

当参数为字符串时以一对双引号（“”）包围；变量则直接将其名称做为参数；表达式作为参数时，要用括号将表达式包围起来。举例来说，想在屏幕上输出“我有 20 本书!”，其中 20 以变量 num 代替，程序的编写如下：

范例：TestJava2_3.java

```
01  // 下面这段程序采用声明变量并赋值的方式，之后在屏幕上打印输出
02  public class TestJava2_3
03  {
04      public static void main(String args[])
05      {
06          int num = 2 ;        // 声明变量并直接赋值为 2
07          System.out.println("我有 "+num+" 本书! ");
08      }
09  }
```

输出结果：

```
我有 20 本书!
```

在 TestJava2_3 程序中，println()中的变量共有三个，以加号连接这些将被打印的数据。在此，加号是“合并”的意思，并非作为算术运算符号的用途。

2.3 程序的检测

现在相信读者大概可以依葫芦画瓢似地写出几个类似的程序了，接下来，要对读者做一些小检测！看看读者是否能够准确地找出下面的程序中存在错误：

范例：TestJava2_4.java

```
01  // 下面程序的错误属于语法错误，在编译的时候会自动检测到
02  public class TestJava2_4
03  {
04      public static void main(String args[])
05      {
06          int num1 = 2 ;          // 声明整数变量 num1，并赋值为 2
07          int num2 = 3 ;             声明整数变量 num2，并赋值为 3
08
09          System.out.println("我有 "+num1" 本书！");
10          System.out.println("你有 "+num2+"本书！")
11      )
12  }
```

2.3.1 语法错误

程序 TestJava2_4 在语法上犯了几个错误，若是通过编译器编译，便可以把这些错误找出来。首先，可以看到第 4 行，main() method 的主体以左大括号开始，应以右大括号结束。所有括号的出现都是成双成对的，因此第 11 行 main() method 主体结束时应以右大括号做结尾，而 Careers2_4 中却以右括号“)”结束。

注释的符号为“//”，但是在第 7 行的注释中，没有加上“//”。在第 9 行，字符串的连接中少了一个“+”号，最后，还可以看到在第 10 行的语句结束时，少了分号作为结束。

上述的三个错误均属于语法错误。当编译程序发现程序语法有错误时，会把这些错误的位置指出，并告诉设计者错误的类型，即可以根据编译程序所给予的信息加以更正。将程序更改后重新编译，若还是有错误，再依照上述的方法重复测试，这些错误就将会被一一改正，直到没有错误为止。上面的程序经过检测、调试之后运行的结果如下：

输出结果

我有 2 本书!

你有 3 本书!

2.3.2 语义错误

当程序本身的语法都没有错误，但是运行后的结果却不符合设计者的要求，此时可能犯了语义错误，也就是程序逻辑上的错误。读者会发现，想要找出语义错误会比找语法错误更难，以下面的程序进行简单的说明：

范例：TestJava2_5.java

```
01  // 下面这段程序原本是要计算一共有多少本书，但是由于错把加号写成了减号，
    // 所以造成了输出结果不正确属于语义错误
02  public class TestJava2_5
03  {
04      public static void main(String args[])
05      {
06          int num1 = 4 ;      //声明一整型变量 num1
07          int num2 = 5 ;      //声明一整型变量 num2
08
09          System.out.println("我有 "+num1+" 本书! ");
10          System.out.println("你有 "+num2+" 本书! ");
11          // 输出 num1-num2 的值 s
12          System.out.println("我们一共有 "+(num1-num2)+ " 本书! ");
13      }
14  }
```

输出结果：

我有 4 本书!

你有 5 本书!

我们一共有 –1 本书!

可以发现，在程序编译过程中并没有发现错误，但是运行后的结果却是不正确的，这种错误就是语义错误，就是在第 12 行中，因失误将“num1+num2”写成了“num1-num2”，虽然语法是正确的，但是却不符合程序的要求，只要将错误更正后，程序的运行结果就是想要的了。

2.4 提高程序的可读性

能够写出一个程序的确很让人兴奋，但如果这个程序除了本人之外，其他人都很难读懂，那这就不算是一个好的程序，所以每个程序设计者在设计程序的时候，也要学习程序设计的规范格式，除了前面所说的加上注释之外，还应当保持适当的缩进，可以看见上面的范例程序都是按缩进的方法编写的，是不是觉得看起来很清晰、明白？读者可以比较下面两个程序，相信看完之后，就会明白程序中使用缩进的好处了！

范例：TestJava2_6.java

```
01  // 以下这段程序是有缩进的样例，可以发现这样的程序看起来比较清楚
02  public class Careers2_6
03  {
04      public static void main(String args[])
05      {
06          int x ;
07
08          for(x=1;x<=3;x++)
09          {
10              System.out.print("x = "+x+", ");
11              System.out.println("x * x = "+(x*x));
12          }
13      }
14  }
```

下面是没有缩进的例子：

范例：TestJava2_7.java

```
01  // 下面的程序于前面程序的输出结果是一样的，但不同的是，
    //这个程序没有采用任何缩进，所以看起来很累
02  public class TestJava2_7{
03  public static void main(String args[]){
04  int x ; for(x=1;x<=3;x++){
05  System.out.print("x = "+x+", ");
06  System.out.println("x * x = "+(x*x));}}}
```

TestJava2_7 这个例子虽然简短，而且语法也没有错误，但是因为编写风格的关系，阅读起来肯定没有 TestJava2_6 这个程序好读，所以建议读者尽量使用缩进，养成良好的编程习惯。

范例 TestJava2_6 和 TestJava2_7 运行后的输出结果如下：

```
x = 1 , x * x = 1
x = 2 , x * x = 4
x = 3 , x * x = 9
```

• 本章摘要：

1、 Java 语言的注释方式有三种：

（1）、“//”记号开始，至该行结束；

（2）、“/*”与“*/”这两个符号之间的文字；

（3）、文档注释。

2、 如果将一个类声明成 public，则它的文件名称必须取成这个类的名称才能顺利编译。

3、 main()在 Java 里是一个相当特殊的 method，它一定要声明成 public，使得在类的其它地方皆可调用到它，且 main() method 没有返回值，所以在它之前要加上 void 关键字。

4、 System.out 是指标准输出，其后所连接的 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。

5、 由于 Java 程序是由类所组成，所以在完整的 Java 程序里，必须且至少有一个类。

6、 Java 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。

7、 变量的设置有以下三种方法：在声明的时候设置、声明后再设置、在程序中的任何位置声明并设置。

8、 提高程序可读性的方法有：

（1）在程序中加上批注；

（2）为变量取个有意义的名称；

（3）保持每一行只有一个语句；

（4）适当的缩进。

第 3 章　Java 基本程序设计

3.1　变量与数据类型

变量是利用声明的方式，将内存中的某个块保留下来以供程序使用。可以声明为块记载的数据类型为整型、字符型、浮点型或是其他数据类型，作为变量的保存之用。本章将就变量及各种数据类型做一个基础性地介绍。

数据类型在程序语言的构成要素里，占有相当重要的地位。Java 的数据类型可分为原始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。这类的变量在声明时是不会分配内存的，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。

3.1.1　变量与常量

下面先来看一个简单的实例，好让读者了解 Java 里变量与常量之间的关系，下面的程序里声明了两种 Java 经常使用到的变量，分别为整型变量 num 与字符变量 ch。为它们赋值后，再把它们的值分别显示在显示器上：

范例：TestJava3_1.java

```
01  // 下面的程序声明了两个变量，一个是整型，一个是字符型
02  public class TestJava 3_1
03  {
04      public static void main(String args[])
05      {
06          int num = 3 ;                    // 声明一整型变量 num，赋值为 3
07          char ch = 'z';                   // 声明一字符变量 ch，赋值为 z
```

```
08              System.out.println(num+ "是整数！ ");       // 输出 num 的值
09              System.out.println(ch + "是字符！ ");       // 输出 ch 的值
10          }
11      }
```

输出结果：

3 是整数!

z 是字符!

在 TestJava 3_1 中，声明了两种不同类型的变量 num 与 ch，并分别将常量 3 与字符“z”赋值给这两个变量，最后再将它们显示在显示器上。

声明一个变量时，编译程序会在内存里开辟一块足以容纳此变量的内存空间给它。不管变量的值如何改变，都永远使用相同的内存空间。因此，善用变量将会是一种节省内存的方式。

常量是不同于变量的一种类型，它的值是固定的，例如整数常量、字符串常量。通常给变量赋值时，会将常量赋值给它，在程序 TestJava 3_1 中，第 6 行 num 是整型变量，而 3 则是常量。此行的作用是声明 num 为整型变量，并把常量 3 这个值赋给它。相同的，第 7 行声明了一个字符变量 ch，并将字符常量'z'赋给它。当然，在程序进行的过程中，可以为变量重新赋值，也可以使用已经声明过的变量。

3.1.2 Java 的变量类型

在 Java 中规定了八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：

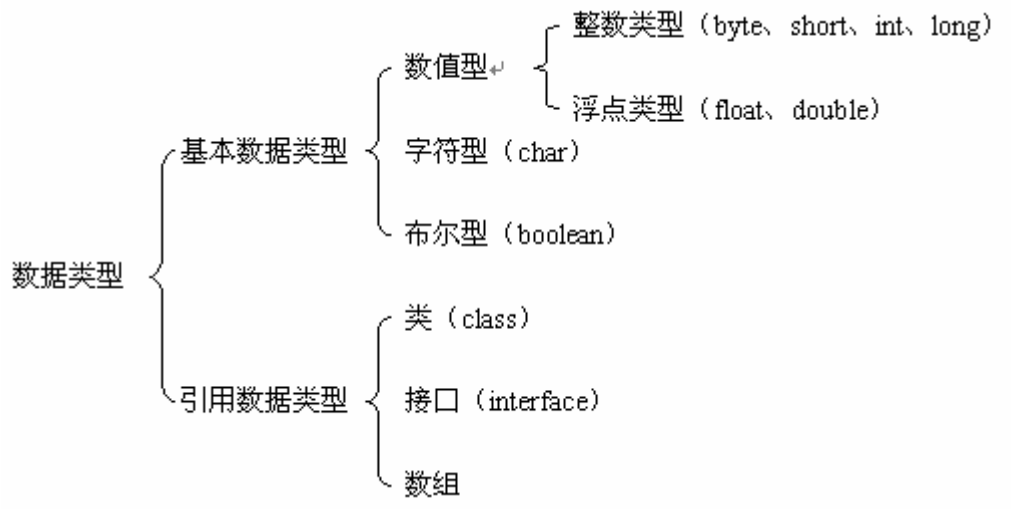


图 3-1　Java 的变量类型

在这里只是先介绍基本数据类型，引用数据类型会在以后的章节中介绍。

3.1.3　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 的基本数据类型

数据类型	字节	表示范围
long（长整数）	8	-9223372036854775808 ~ 9223372036854775807
int（整数）	4	-2147483648 ~ 2147483647
short（短整数）	2	-32768 ~ 32767
byte（位）	1	-128 ~ 127
char（）	1	0 ~ 255
boolean（）	1	布尔值只能使用 true 或 false
float（）	4	-3.4E38（-3.4×10^{38}） ~ 3.4E38（3.4×10^{38}）
double（）	8	-1.7E308（-1.7×10^{308}）~ 1.7E308（1.7×10^{308}）

3.1.3.1 整数类型

当数据不带有小数或分数时，即可以声明为整数变量，如 3，-147 等即为整数。Java 中，整数数据类型可以分为 long、int、short 及 byte 四种：long 为 64 位，也就是 8 个字节（bytes），可表示范围为-9223372036854775808 到 9223372036854775807；int 为 32 位，也就是 4 个字节，表示范围为-2147483648 到 2147483647；若是数据值的范围在-32768 到 32767 之间时，可以声明为 short（短整数）类型；若是数据值更小，在-128 到 127 之间时，可以声明为 byte 类型以节省内存空间。举例来说，想声明一个短整型变量 sum 时，可以在程序中做出如下的声明：

short sum ;　　// 声明 snum 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 2 个字节的块供 sum 变量使用，同时这个变量的范围只能在-32768 到 32767 之间。

3.1.3.1.1 常量的数据类型

有趣的是，Java 把整数常量的数据类型均视为 int 型，因此，如果在程序中使用了超过 2147483647 这个大小的常量，编译时将发生错误，如下面的范例：

范例：TestJava3_2.java

```
01  // 下面这段程序说明了值的可取范围的问题
02  public class TestJava3_2
03  {
04      public static void main(String args[])
05      {
06          long num = 329852547553;          // 声明一长整型变量
07          System.out.println("num = "+num);
08      }
09  }
```

如果编译上面的程序代码，将会得到下列的错误信息：

TestJava3_2.java:5: integer number too large: 329852547553
long num = 329852547553;

这是因为把整数常量看成是 int 类型，但 329852547553 这个整数已超出了 int 类型所能表示的范围，因此虽然把 num 的类型设为 long，但编译时仍然会发生错误。要解决这个问题，只要在整数常量后面加上一个大写的“L”即可，此举代表该常量是 long 类型的整数常量。所以只要把第 6 行的语句改成：

06 long num = 329852547553L ;

即可成功地编译运行。

3.1.3.1.2 数据类型的最大值与最小值

Java 提供了 long、int、short 及 byte 四种整数类型的最大值、最小值的代码，以方便设计者使用。最大值的代码是 MAX_VALUE，最小值是 MIN_VALUE。如果要取用某个类型的最大值或最小值，只要在这些代码之前，加上它们所属的类的全名即可。举例来说，如果程序代码里需要用到长整数的最大值，如图 3-2 所示的语法表示。

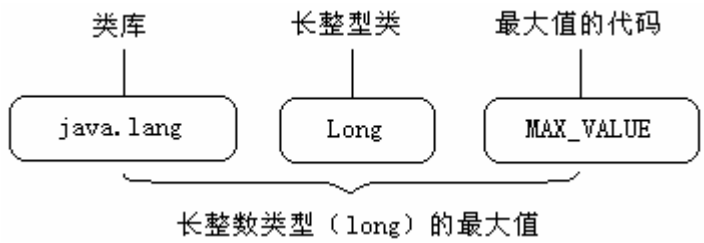


图 3-2 代码的表示法

由上面的语法可知，如果要使用某个类型的代码，则必须先指定该类型所在的类库以及该类型所属的类，但因 java.lang 这个类库属于常用类库，所以默认的 Java 程序会将它加载，因此在实际的应用上设计者可以将它省略。

Java 所提供的整数的最大值与最小值的标识符及常量值，可在表 3-2 中查阅。若是读者现在不懂什么叫类库也没有关系，现在只要会使用它就行了。

表 3-2 整数常量的特殊值代码

	Long	**int**
使用类全名	java.lang.Long	java.lang.Integer
最大值代码	**MAX_VALUE**	**MAX_VALUE**
最大值常量	9223372036854775807	2147483647
最小值代码	**MIN_VALUE**	**MIN_VALUE**
最小值常量	-9223372036854775808	-2147483648

	short	**byte**
使用类全名	java.lang.Short	java.lang.Byte
最大值代码	**MAX_VALUE**	**MAX_VALUE**
最大值常量	32767	127
最小值代码	**MIN_VALUE**	**MIN_VALUE**
最小值常量	-32768	-128

下面程序是输出 Java 定义的四种整数类型的常量的最大和最小值，可以将程序与上表做对照、比较。

范例：TestJava3_3.java

```
01  // 下面这段程序调用了表 3-2 中的方法，可以得到数据类型的最大值和最小值
02  public class TestJava3_3
03  {
04      public static void main(String args[])
05      {
06          long long_max = java.lang.Long.MAX_VALUE ;//得到长整型的最大值
07          int int_max = java.lang.Integer.MAX_VALUE ; // 得到整型的最大值
08          short short_max = Short.MAX_VALUE ;        // 得到短整型的最大值
09          byte byte_max = Byte.MAX_VALUE ;           // 得到 Byte 型的最大值
10
11          System.out.println("LONG 的最大值：  "+long_max);
```

```
12              System.out.println("INT 的最大值："+int_max);
13              System.out.println("SHORT 的最大值："+short_max);
14              System.out.println("BYTE 的最大值："+byte_max);
15          }
16      }
```

输出结果：

LONG 的最大值： 9223372036854775807

INT 的最大值：2147483647

SHORT 的最大值：32767

BYTE 的最大值：127

程序 TestJava3_3 列出了各种整数类型的最大值，通过它的运行，读者可以了解到 Java 对于整数的最大值、最小值的规定。读者可以自己改写程序，输出一下最小值。

3.1.3.1.3 溢出的发生

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：TestJava3_4.java

```
01  // 整数值如果超出了自己所可以表示范围的最大值，会出现溢出
02  public class TestJava3_4
03  {
04      public static void main(String args[])
05      {
06          int x = java.lang.Integer.MAX_VALUE ;        // 得到整型的最大值
07
08          System.out.println("x = "+x);
```

```
09              System.out.println("x+1 = "+(x+1));
10              System.out.println("x+2 = "+(x+2));
11          }
12  }
```

输出结果：

```
x = 2147483647
x+1 = -2147483648
x+2 = -2147483647
```

当最大值加上 1 时，结果反而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，这个情形会出现一个循环，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 i，其表示的范围为-2147483648 ~ 2147483647 之间，当 i 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。

上述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 3-3 来了解数据类型的溢出问题。

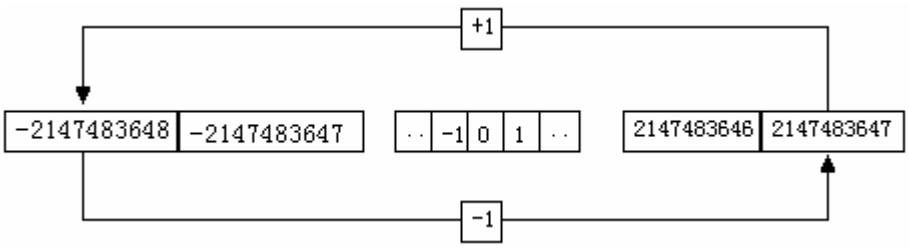


图 3-3　数据类型的溢出

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”，或是在变量前面加上 long，作为强制类型的转换。以 TestJava3_5 为例，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，特地保留一个整数的溢出语句。

范例：TestJava3_5.java

```
01  // 下面的这段程序当整型发生溢出之后，用强制类型进行转换
02  public class TestJava3_5
03  {
04      public static void main(String args[])
05      {
06          int x = java.lang.Integer.MAX_VALUE ;
06
08          System.out.println("x = "+x);
09          System.out.println("x + 1 = "+(x+1));
10          System.out.println("x + 2 = "+(x+2L));
11          System.out.println("x + 3 = "+((long)x+3));
12      }
13  }
```

输出结果：

x = 2147483647

x + 1 = -2147483648

x + 2 = 2147483649

x + 3 = 2147483650

程序说明：

1、 第 6 行声明 int 类型的整数变量 x，并赋值为整数最大值，即 2147483647。

2、 第 8 行输出 x 的值，即：2147483647。

3、 第 9 行输出 x+1 的值，此时溢出发生，运行结果变成-2147483648。

4、 第 10 行输出 x+2 的值，为了避免溢出发生，在表达式的常量部分 2 后加上 L，执行结果变成 2147483649。

5、 第 11 行输出 x+3 的值，为了避免溢出发生，在表达式的整数部分 x 之前加上 long，执行结果变成 2147483650。

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。

3.1.3.2 字符类型

字符类型在内存中占有 2 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，严格说来，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，在数据跨平台的时候就会发生错误。

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。

举例来说，Unicode 中的小写 a 是以 97 来表示，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：TestJava3_6.java

```
01  // 字符类型也可以直接赋给数值，下面的这段程序就是采用这种赋值方式
02  public class TestJava3_6
03  {
04      public static void main(String args[])
05      {
06          char ch1 = 97 ;
07          char ch2 = 'a' ;
08
09          System.out.println("ch1 = "+ch1);
```

```
10              System.out.println("ch2 = "+ch2);
11          }
12      }
```

输出结果：

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

给字符变量在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。

举例来说，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，再将它输出来，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。表 3-3 为常用的转义字符：

表 3-3　常用的转义字符

转义字符	所代表的意义	转义字符	所代表的意义
\f	换页	\\	反斜线
\b	倒退一格	\'	单引号
\r	归位	\"	双引号
\t	跳格	\n	换行

以下面的程序为例，将 ch 赋值为“\"”（要以单引号（'）包围），并将字符变量 ch 输出在显示器上，同时在打印的字符串里直接加入转义字符，读者可自行比较一下两种方式的差异。

范例：TestJava3_7.java

```
01  // 下面这道程序表明了转义字符的使用方法
02  public class TestJava3_7
03  {
04      public static void main(String args[])
05      {
06          char ch =' \" ';
```

```
07
08              System.out.println(ch+"测试转义字符！"+ch);
09              System.out.println("\"hello world！\"");
10          }
11  }
```

输出结果：

"测试转义字符！"

"hello world！"

不管是用变量存放转义字符，或是直接使用转义字符的方式来输出字符串，程序都可以顺利运行。由于使用变量会占用内存资源，似乎显得有些不妥；当然也可以不必声明字符变量就可以输出转义字符，但如果在程序中加上太多的转移字符，以至于造成混淆而不易阅读时，利用声明字符变量的方式就是一个很好的选择。

3.1.3.3 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、体重等需要精确的数值时，整数就不能满足程序设计者的要求了。在数学中，这些带有小数点的数值称为实数，在Java里，这种数据类型称为浮点数类型（float），其长度为 4 个字节，有效范围为 -3.4×10^{38} 到 3.4×10^{38}。当浮点数的表示范围不够大的时候，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 8 个字节，有效范围为 -1.7×10^{308} 到 1.7×10^{308}。

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。举例来说，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，可以在程序中做出如下的声明及设置：

```
double num ;            // 声明 sum 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-1.7×10^{308} 到 1.7×10^{308}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-3.4×10^{38} 到 3.4×10^{38}之间。在此例中，sum的初值为 3.0。

下列为声明与设置 float 与 double 类型的变量时应注意的事项：

double num1 = -6.3e64 ;　　// 声明 num1 为 double，其值为-6.3×1064

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 2.456×1067 已超过 float 可表示的范围

值得一提的是，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 或是 f，则作为 float 类型的识别。若是没有加上，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。

下面举一个简单的例子，在下面的程序里，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：TestJava3_8.java

```
01  // 下面这道程序说明了浮点数类型的使用方法
02  public class TestJava3_8
03  {
04      public static void main(String args[])
05      {
06          float num = 3.0f ;
07          System.out.println(num+" *"+num+" = "+(num*num));
08      }
09  }
```

输出结果：

```
3.0 * 3.0 = 9.0
```

Java 也提供了浮点数类型的最大值与最小值的代码，其所使用的类全名与所代表的值的范围，可以在表 3-4 中查阅：

表 3-4　浮点数常量的特殊值

	float	**double**
使用类全名	java.lang.Float	java.lang.Double
最大值	**MAX_VALUE**	**MAX_VALUE**
最大值常量	3.4028235E38	107976931348623157E308
最小值	**MIN_VALUE**	**MIN_VALUE**
最小值常量	1.4E-45	4.9E-324

相同的，在类全名中，可以省去类库 java.lang，直接取用类名称即可。下面的程序是输出 float 与 double 两种浮点数类型的最大与最小值，读者可以将下面程序的输出结果与上表一一进行比较。

范例：TestJava3_9.java

```
01  // 下面这道程序用于取得单精度和双精度浮点数类型的最大、最小值
02  public class TestJava3_9
03  {
04      public static void main(String args[])
05      {
06          System.out.println("float_max = "+java.lang.Float.MAX_VALUE);
07          System.out.println("float_min = "+java.lang.Float.MIN_VALUE);
08          System.out.println("double_max = "+java.lang.Double.MAX_VALUE);
09          System.out.println("double_min = "+java.lang.Double.MIN_VALUE);
10      }
11  }
```

输出结果：

```
float_max = 3.4028235E38
float_min = 1.4E-45
double_max = 1.7976931348623157E308
double_min = 4.9E-324
```

3.1.3.4 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。举例来说，想声明名称为 status 变量为的布尔类型，并设置为 true 值，可以使用下面的语句：

```
boolean status = true ;        // 声明布尔变量 status，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，即可以随时更改。将上述的内容写成了程序 TestJava3_10，读者可以先熟悉一下布尔变量的使用：

范例：TestJava3_10.java

```
01  // 下面的程序声明了一个布尔值类型的变量
02  public class TestJava3_10
03  {
04        public static void main(String args[])
05      {
06  // 声明一布尔型的变量 status，布尔型只有两个值一个是 true 一个是 false
08              boolean status = true ;
09              System.out.println("status = "+status);
10        }
11  }
```

输出结果：

```
status = 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，本书会陆续在后面的章节中介绍布尔值在程序流程中所起的作用。

3.1.3.5 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 3-5 列出了各种类型的默认值。

表 3-5 基本数据类型的默认值

数据类型	默认值
byte	（byte）0
short	（short）0
int	0
long	0L
float	0.0f
double	0.0d
char	\u0000（空）
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，但这没有任何意义，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，这是个需要注意的问题。

3.1.4 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它的数据类型，但 Java 容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。

3.1.4.1 自动类型转换

在程序中已经定义好了数据类型的变量，若是想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。

举例来说，若是想将 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，亦符合条件 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，因此是不可能做类型的转换。接下来看看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：TestJava3_11.java

```
01  // 下面这段程序声明了两个变量，一个是整型，一个是浮点型
02  public class TestJava3_11
03  {
04      public static void main(String args[])
05      {
06          int a = 156 ;
07          float b = 24.1f ;                    // 声明一浮点型变量 f，并赋值
08
09          System.out.println("a = "+a+" , b = "+b);
```

```
10              System.out.println("a / b = "+(a/b));     // 这里整型会自动转化为浮点型
11          }
12      }
```

输出结果：

a = 156 , b = 24.0

a / b = 6.5

从运行的结果可以看出，当两个数中有一个为浮点数时，其运算的结果会直接转换为浮点数。当表达式中变量的类型不同时，Java 会自动以较小的表示范围转换成较大的表示范围后，再作运算。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。

3.1.4.2 强制类型转换

当两个整数进行运算时，其运算的结果也会是整数。举例来说，当做整数除法 8/3 的运算，其结果为整数 2，并不是实际的 2.6666…，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 】

```
（欲转换的数据类型）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，所以也称为显性转换。下面的程序说明了在 Java 里，整数与浮点数是如何转换的。

范例：TestJava3_12

```
01  // 下面范例中说明了自动转换和强制转换这两种转换的使用方法
02  public class TestJava3_12
03  {
04      public static void main(String args[])
05      {
```

```
06              int a = 55 ;
07              int b= 9 ;
08              float g,h ;
09
10              System.out.println("a = "+a+" , b = "+b);
11              g = a/b ;
12              System.out.println("a / b ="+g+"\n");
13              System.out.println("a = "+a+" , b = "+b);
14              h = (float)a/b ;                    //在这里将数据类型进行强制类型转换
15              System.out.println("a /b = "+h);
16          }
17  }
```

输出结果：

a = 55 , b = 9

a / b = 6.0

a = 55 , b = 9

a /b = 6.111111

当两个整数相除时，小数点以后的数字会被截断，使得运算的结果保持为整数。但由于这并不是预期的计算结果，而想要得到运算的结果为浮点数，就必须将两个整数中的其中一个（或是两个）强制转换类型为浮点数，下面的三种写法都正确：

（1） (float)a/b // 将整数 a 强制转换成浮点数，再与整数 b 相除

（2） a/(float)b // 将整数 b 强制转换成浮点数，再以整数 a 除之

（3） (float)a/(/float)b // 将整数 a 与 b 同时强制转换成浮点数，再相除

只要在变量前面加上欲转换的数据类型，运行时就会自动将此行语句里的变量做类型转换的处理，但这并不影响原先所定义的数据类型。

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。

3.2　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。本章将介绍 Java 运算符的用法、表达式与运算符之间的关系，以及表达式里各种变量的数据类型的转换等。学完本章，希望读者能对 Java 语句的运作过程有更深一层的认识。

3.2.1　表达式与运算符

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：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。以下面的表达式（z+100）为例，“z”与“100”都是操作数，而“+”就是运算符。

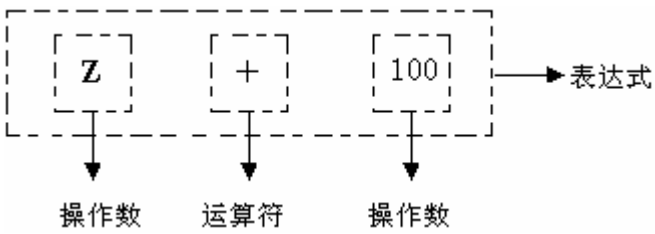


图 3-4　表达式是由操作数与运算符所组成

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、括号运算符等。

3.2.1.1　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 3-6 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 3-6　赋值运算符

赋值运算符号	意义
=	赋值

等号（=）在 Java 中并不是“等于”的意思，而是“赋值”的意思。还记得在前几章中，为变量赋值的语句吗？

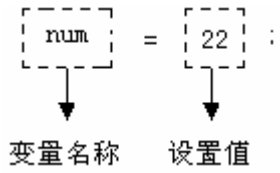


图 3-5　表达式的赋值范例

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 的值运算之后再赋值给变量 num 存放

从未学习过 C 或 C++的读者，可能会不习惯这种思考方式。若是把等号（=）当成“等于”，这种语句在数学上根本说不通，但是把它看成“赋值”时，这个语句就很容易理解了，把 num-3 的值运算之后再赋值给 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 了。将上面的语句编写成下面这个程序：

范例：TestJava3_13.java

```
01  // 在程序中赋值采用“=”
02  public class TestJava4_1
03  {
04      public static void main(String args[])
05      {
06          int num = 22 ;                  // 声明整数变量 num，并赋值为 22
```

```
07
08              System.out.println("第一次赋值后，num = "+num);    // 输出 num 的值
09
10              num = num –3 ;            // 将变量 num 的值减三之后再赋给 num 变量
11              System.out.println("改变之后的值，num = "+num);//输出计算后 num 的值
12          }
13      }
```

输出结果：

第一次赋值后，num = 22

改变之后的值，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：

in 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。

3.2.1.2　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所组成的。

```
+3 ;        // 表示正 3
~a ;        // 表示取 a 的补码
b = -a ;    // 表示负 a 的值赋值给变量 b 存放
!a ;        // a 的 NOT 运算，若 a 为零，则!a 为 1，若 a 不为零，则!a 为零
```

表 3-7 列出了一元运算符的成员：

表 3-7　一元运算符

一元运算符	意义
+	正号
–	负号
!	NOT，否
~	取补码

下面的程序声明了 byte 类型的变量 a 及 boolean 类型的变量 b，可以看到两个变量分别进行了“~”与“!”运算之后的结果。

范例：TestJava3_14.java

```
01  // 下面这段程序说明了一元运算符的使用
02  public class TestJava3_14
03  {
04      public static void main(String args[])
05      {
06          byte a = java.lang.Byte.MAX_VALUE ; // 声明并将其类型最大值赋给 a
07          boolean b = false ;
08
09          System.out.println("a = "+a+" , ~a = "+(~a));
10          System.out.println("b = "+b+" , !b = "+(!b));
11      }
12  }
```

输出结果：

```
a = 127 , ~a = –128
b = false , !b = true
```

程序说明：

1、 第 6 行声明了 byte 变量 a，并赋值为该类型的最大值，即 a 的值为 127。程序第 7 行，声明 boolean 变量 b，赋值为 false。

2、 第 9 行输出 a 与~a 的运算结果。

3、 第 10 行输出 b 与！b 的运算结果。b 的值为 flase，因此进行“!”运算后，b 的值就变成了 true。

3.2.1.3 算术运算符

算术运算符在数学上面经常会使用到，表 3-8 列出了它的成员：

表 3-8 算术运算符

算术运算符	意义
+	加法
–	减法
*	乘法
/	除法
%	余数

3.2.1.3.1 加法运算符“+”

将加法运算符“+”的前后两个操作数相加。如下面的语句：

```
System.out.println("3 + 8 = "+(3+8));          // 直接输出表达式的值
```

3.2.1.3.2 减法运算符“–”

将减法运算符“–”前面的操作数减去后面的操作数，如下面的语句：

```
num = num –3 ;          // 将 num-3 运算之后赋值给 num 存放
a = b – c ;             // 将 b–c 运算之后赋值给 a 存放
120 - 10 ;              // 运算 120 – 10 的值
```

3.2.1.3.3 乘法运算符“*”

将乘法运算符“*”的前后两个操作数相乘，如下面的语句：

```
b = b * 5 ;                    // 将 b*5 运算之后赋值给 b 存放
a = a *a ;                     // 将 a * a 运算之后赋值给 a 存放
19 * 2 ;                       // 运算 19 * 2 的值
```

3.2.1.3.4 除法运算符“/”

将除法运算符“/”前面的操作数除以后面的操作数，如下面的语句：

```
a = b / 5 ;            // 将 b / 5 运算之后的值赋给 a 存放
c = c / d ;            // 将 c / d 运算之后的值赋给 c 存放
15 / 5 ;               // 运算 14 / 7 的值
```

使用除法运算符时要特别注意一点，就是数据类型的问题。以上面的例子来说当 a、b、c、d 的类型皆为整数，若是运算的结果不能整除时，输出的结果与实际的值会有差异，这是因为整数类型的变量无法保存小数点后面的数据，因此在声明数据类型及输出时要特别小心。以下面的程序为例，在程序里给两个整型变量 a、b 赋值，并将 a / b 的运算结果输出：

范例：TestJava3_15.java

```
01  // 下面这段程序说明了除法运算符的使用方法
02  public class TestJava3_15
03  {
04      public static void main(String[] args)
05      {
06          int a = 13 ;
07          int b = 4 ;
08
09          System.out.println("a = "+a+" , b = "+b);
10          System.out.println("a / b = "+(a/b));
11          System.out.println("a / b = "+((float)a/b));      // 进行强制类型转换
```

```
12          }
13    }
```

输出结果：

a = 13 , b = 4

a / b = 3

a / b = 3.25

程序说明：

1、 第 10 行与 11 行，程序分别做出不同的输出：第 10 行中，因为 a，b 皆为整数类型，输出结果也会是整数类型，程序运行结果与实际的值不同。

2、 第 11 行中，为了保证程序运行结果与实际的值相同，所以使用了强制性的类型转换，即将整数类型（int）转换成浮点数类型（float），程序运行的结果才不会有问题。

3.2.1.3.5 余数运算符“%”

将余数运算符“%”前面的操作数除以后面的操作数，取其所得到的余数。下面的语句是余数运算符的使用范例：

```
num = num % 3 ;          // 将 num%3 运算之后赋值给 num 存放
a = b % c ;              // 将 b%c 运算之后赋值给 a 存放
100 % 7 ;                // 运算 100%7 的值
```

以下面的程序为例，声明两个整型变量 a、b，并分别赋值为 5 和 3，再将 a%b 的运算结果输出。

范例：TestJava3_16.java

```
01  // 在 JAVA 中用%进行取模操作
02  public class TestJava3_16
03  {
```

```
04        public static void main(String[] args)
05        {
06              int a = 5 ;
07              int b = 3 ;
08
09              System.out.println(a+" % "+b+" = "+(a%b));
10              System.out.println(b+" % "+a+" = "+(b%a));
11        }
12  }
```

输出结果：

5 % 3 = 2

3 % 5 = 3

3.2.1.4 关系运算符与 if 语句

设计者常常会在 if 语句中使用到关系运算符，所以有必要先来认识 if 语句的用法。if 语句的格式如下：

【 格式 3-2 if 语句的格式 】

```
if (判断条件)
    语句 ;
```

如果括号中的判断条件成立，就会执行后面的语句；若是判断条件不成立，则后面的语句就不会被执行，如下面的程序片段：

```
if (x>0)
    System.out.println("I like Java ！ ");
```

当 x 的值大于 0，就是判断条件成立时，会执行输出字符串“I like Java！”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

了。表 3-9 列出了关系运算符的成员，这些运算符在数学上也是经常使用的。

表 3-9　关系运算符

关系运算符	意义
>	大于
<	小于
>=	大于等于
<=	小于等于
==	等于
!=	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 2 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。

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响应值 true，若是判断式不成立则会产生响应值 false。以下面的程序为例，判断 if 语句括号中的条件是否成立，若是成立则执行 if 后面的语句。

范例：TestJava3_17.java

```
01  // 下面这段程序说明了关系运算符的使用方法，关系运算符返回值为布尔值
02  public class TestJava3_17
03  {
04      public static void main(String[] args)
05      {
06          if(5>2)
07              System.out.println("返回值："+(5>2));
08
09          if(true)
10              System.out.println("Hello Java !");
```

```
11
12              if((3+6)==(3-6))
13                  System.out.println("I like Java !");
14          }
15  }
```

输出结果：

返回值：true

Hello Java !

程序说明：

1、 在第 6 行中，由于 5>2 的条件成立，所以执行第 7 行的语句：输出返回值 true。

2、 在第 9 行中，若是 if 语句的参数为 true，判断亦成立，所以接着执行第 10 行的语句：输出字符串 Hello TestJava!。

3、 第 12 行，3+6 并不等于 3-6，if 的判断条件不成立，所以第 13 行语句不被执行。

3.2.1.5 递增与递减运算符

递增与递减运算符在 C / C++中就已经存在了，Java 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 3-10 列出了递增与递减运算符的成员。

表 3-10 递增与递减运算符

递增与递减运算符	意义
++	递增，变量值加 1
––	递减，变量值减 1

善用递增与递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。也可以利用递增运算符“++”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递增运算符“++”的用法，就是递增运算符“++”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。以下面的程序为例，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：TestJava3_18.java

```
01  // 下面这段程序说明了“++”的两种用法的使用
02  public class TestJava3_18
03  {
04      public static void main(String args[])
05      {
06          int a = 3 , b = 3 ;
07
08          System.out.print("a = "+a);                          // 输出 a
09          System.out.println(" , a++ = "+(a++)+" , a= "+a);    // 输出 a++和 a
10          System.out.print("b = "+b);                          // 输出 b
11          System.out.println(" , ++b = "+(++b)+" , b= "+b);    // 输出++b 和 b
12      }
13  }
```

输出结果：

a = 3 , a++ = **3** , a= 4

b = 3 , ++b = 4 , b= 4

程序说明：

1、 在第 9 行中，输出 a++及运算后的 a 的值，所以执行完 a++后，a 的值才会加 1，变成 4。

2、 程序的第 11 行中，输出++b 运算后 b 的值，所以执行++b 前，b 的值即先加 1，变成 4。

同样的，递减运算符“--”的使用方式和递增运算符“++”是相同的，递增

运算符“++”用来将变量值加 1，而递减运算符“--”则是用来将变量值减 1。此外，递增与递减运算符只能将变量加 1 或减 1，若是想要将变量加减非 1 的数时，还是得用原来的“a = a+2”的方法。

3.2.1.6 逻辑运算符

在 if 语句中也可以看到逻辑运算符，表 3-11 列出了它的成员。

表 3-11 逻辑运算符

逻辑运算符	意义
&&	AND，与
||	OR，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：

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

在第 1 个例子中， 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；在第 2 个例子中，只要 a>0 或者 b>0，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 3-12 中所列出的结果：

表 3-12 AND 及 OR 结果表

条件 1	条件 2	结果	
		&&（与）	||（或）
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

由表 3-12 可以看出，在条件中，只要有一个条件为假（false），相与的结果就是假（false）；相反的如果有一个条件为真（true），那么相或的结果就为真（true）。

下面的这个程序是判断 a 的值是否在 0~100 之间，如果不在即表示成绩输入错误；若是 a 的值在 50~60 之间，则需要补考。

范例：TestJava3_19.java

```
01  // 下面这段程序是对与操作进行的说明，返回值为布尔类型
02  public class TestJava3_19
03  {
04      public static void main(String[] args)
05      {
06          int a = 56 ;
07
08          if((a<0)||(a>100))
09              System.out.println("输入的数据有错误！");
10          if((a<60)&&(a>49))
11              System.out.println("准备补考吧！");
12      }
13  }
```

输出结果：

准备补考吧！

程序说明：

1、 当程序执行到第 8 行时，if 会根据括号中 a 的值作判断，a<0 或是 a>100 时，条件判断成立，即会执行第 9 行的语句：输出字符串“输入的数据有错误!”。由于学生成绩是介于 0~100 分之间，因此当 a 的值不在这个范围时，就会视为是输入错误。

2、 不管第 9 行是否有执行，都会接着执行第 10 行的程序。if 再根据括号中 a 的值做判断，a<60 且 a>49 时，条件判断成立，表示该成绩需要进行补考，即会执行第 11 行的语句，输出“准备补考吧!”。

3.2.1.7　括号运算符

除了前面所述的内容外，括号（）也是 Java 的运算符，如表 3-13 所示：

表 3-13　括号运算符

括号运算符	意义
（）	提高括号中表达式的优先级

括号运算符（）是用来处理表达式的优先级的。以一个简单的加减乘除式子为例：

```
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。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。

3.2.2　运算符的优先级

表 3-14 列出了各个运算符的优先级的排列，数字越小的表示优先级越高。

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	++、−−	递增与递减运算符	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
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	?:	条件运算符	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。什么是结合性呢？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。举例来说，当使用同一优先级的运算符时，结合性就非常重要了，它决定谁会先被处理。读者可以看看下面的例子：

a = b +d　/ 5 * 4 ;

这个表达式中含有不同优先级的运算符，其中是“/”与“*”的优先级高于“+”，而“+”又高于“=”，但是读者会发现，“/”与“*”的优先级是相同的，到底 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 得到的结果如上 b 后，将整个值赋给 a 存放。

3.2.3 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 还有一些相当简洁的写法，是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　简洁的表达式

运算符	范例用法	说明	意义
+=	a += b	a + b 的值存放到 a 中	a = a + b
-=	a -= b	a - b 的值存放到 a 中	a = a - b
*=	a *= b	a * b 的值存放到 a 中	a = a * b
/=	a /= b	a / b 的值存放到 a 中	a = a / b
%=	a %= b	a % b 的值存放到 a 中	a = a % b

下面的几个表达式，皆是简洁的写法：

```
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--
```

这种独特的写法虽然看起来有些怪异，但是它却可以减少程序的行数，提高运行的速度！看下面这道范例：

范例：TestJava3_20.java

```
01  // 下面是关于简洁写法的一段程序
02  public class TestJava3_20
03  {
04      public static void main(String[] args)
05      {
06          int a = 5 , b = 8 ;
07
08          System.out.println("改变之前的数是：  a = "+a+" , b = "+b);
09          a +=b ;
10          System.out.println("改变之后的数是：  a = "+a+" , b = "+b);
11      }
12  }
```

输出结果：

改变之前的数是： a = 5 , b = 8

改变之后的数是： a = 13 , b = 8

程序说明：

1、 第 6 行分别把变量 a、b 赋值为 5 及 8。

2、 第 8 行在运算之前先输出变量 a、b 的值，a 为 5，b 为 8。

3、 第 9 行计算 a+=b，这个语句也就相当于 a = a +b，将 a+b 的值存放到 a 中。计算 5+8 的结果后赋值给 a 存放。

4、 程序第 10 行，再输出运算之后变量 a、b 的值。所以 a 的值变成 13，而 b 仍为 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，递增、递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

运算符	范例	执行前		说明	执行后	
		a	**b**		**a**	**b**
+=	a += b	12	4	a + b 的值存放到 a 中（同 a = a + b）	16	4
-=	a -= b	12	4	a - b 的值存放到 a 中（同 a = a – b）	8	4
*=	a *= b	12	4	a * b 的值存放到 a 中（同 a = a * b）	48	4
/=	a /= b	12	4	a / b 的值存放到 a 中（同 a = a / b）	3	4
%=	a %= b	12	4	a % b 的值存放到 a 中（同 a = a % b）	0	4
B++	a *= b++	12	4	a * b 的值存放到 a 后，b 加 1 （同 a = a * b；b++）	48	5
++b	a *= ++b	12	4	b 加 1 后，再将 a*b 的值存放到 a （同 b++；a=a*b）	60	5
b--	a *= b--	12	4	a * b 的值存放到 a 后，b 减 1 （同 a=a*b；b--）	48	3
--b	a *= --b	12	4	b 减 1 后，再将 a*b 的值存放到 a （同 b--；a=a*b）	36	3

举一个实例来说明这些简洁的表达式在程序中该如何应用。以 TestJava3_21 为例，输入两个数，经过运算之后，来看看这两个变量所存放的值有什么变化。

范例：TestJava3_21.java

```
01  // 下面的程序说明了简洁表达式的使用方法，但这种方式现在已不提倡使用了。
02  public class TestJava3_21
03  {
04      public static void main(String[] args)
05      {
06          int a = 10 , b = 6 ;
07
08          System.out.println("改变之前的数：  a = "+a+" , b = "+b);
09          a -= b++ ;          // 先计算 a-b 的值，将结果设给 a 之后，再将 b 值加 1
10          System.out.println("改变之后的数：  a = "+a+" , b = "+b);
11      }
12  }
```

输出结果：

改变之前的数： a = 10 , b = 6

改变之后的数： a = 4 , b = 7

程序说明：

1、 第 8 行输出运算前变量 a、b 的值。在程序中 a、b 的赋值为 10、6，因此输出的结果 a 为 10，b 为 6。

2、 第 9 行计算 a -= b++，也就是执行下面这两个语句：

a = a- b ; **//（a = 10 - 6 = 4，所以 a = 4）**

b++ ; **//（b = b+1 = 6+1 = 7 ，所以 b = 7）**

3、 程序第 10 行，将经过运算之后的结果输出，即可得到 a 为 4，b 为 7 的答案

3.2.4 表达式的类型转换

当 int 类型遇上了 float 类型，到底谁是“赢家”呢？在前面曾提到过数据类型的转换，在这里，要再一次详细讨论表达式的类型转换。

Java 是一个很有弹性的程序设计语言，当上述的情况发生时，只要坚持“以不流失数据为前提”的大原则，即可做不同的类型转换，使不同类型的数据、表达式都能继续存储。依照大原则，当 Java 发现程序的表达式中有类型不相符的情况时，会依据下列的规则来处理类型的转换。

1、 占用字节较少的类型转换成占用字节较多的类型。

2、 字符类型会转换成 int 类型。

3、 int 类型会转换成 float 类型。

4、 表达式中若某个操作数的类型为 double，则另一个操作数字也会转换成 double 类型。

5、 布尔类型不能转换成其它类型。

范例：TestJava3_22.java

```
01  // 下面的程序说明了表达式类型的自动转换问题
02  public class TestJava3_22
03  {
04      public static void main(String[] args)
05      {
06          char ch = 'a' ;
07          short a = -2 ;
08          int b = 3 ;
09          float f = 5.3f ;
10          double d = 6.28 ;
11
12          System.out.print("(ch / a) - (d / f) - (a + b) = ");
13          System.out.println((ch / a) - (d / f) - (a + b));
14      }
15  }
```

输出结果：

```
(ch / a) - (d / f) - (a + b) = -50.18490561773532
```

先别急着看结果，在程序运行之前可先思考一下，这个复杂的表达式（ch / a）- (d / f) – (b + a)最后的输出类型是什么？它又是如何将不同的数据类型转换成相同的呢？读者可以参考图 3-6 的分析过程。

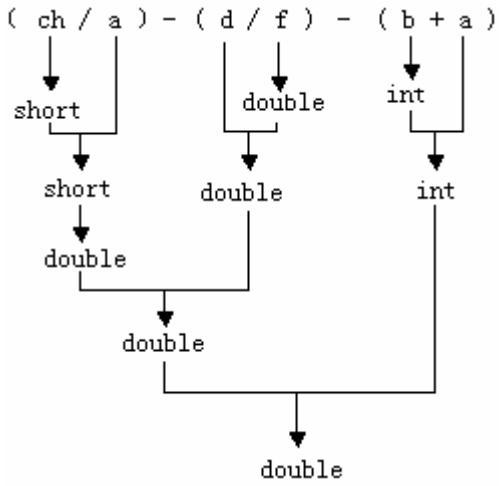


图 3-6　数据类型的转换过程

3.3 循环与选择性语句

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。在本章中就要认识选择与循环结构语句，学习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。

3.3.1 程序的结构设计

一般来说程序的结构包含有下面三种：

1、 顺序结构

2、 选择结构

3、 循环结构

这三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。程序中使用了上面这些结构到底有什么好处呢？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。现在以流程图的方式来让读者了解这三种结构的不同。

3.3.1.1 顺序结构

本书前面所讲的那些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 3-7 所示：

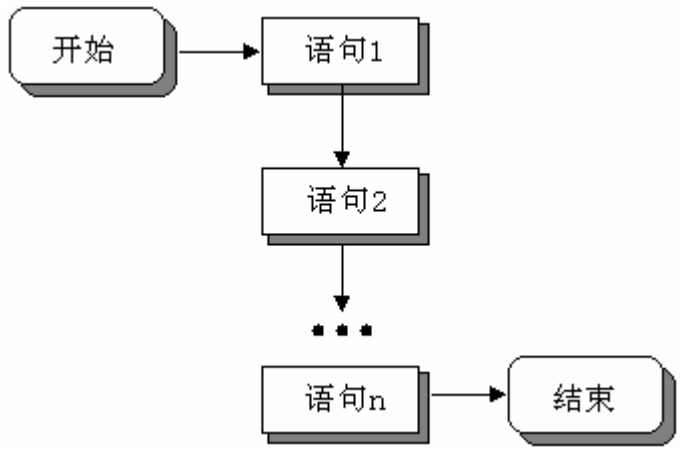


图 3-7　顺序结构的基本流程

顺序结构在程序设计中是最常使用到的结构，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这种由上而下的流程来设计。

3.3.1.2　选择结构

选择结构是根据条件的成立与否，再决定要执行哪些语句的结构，其流程图如图 3-8 所示。

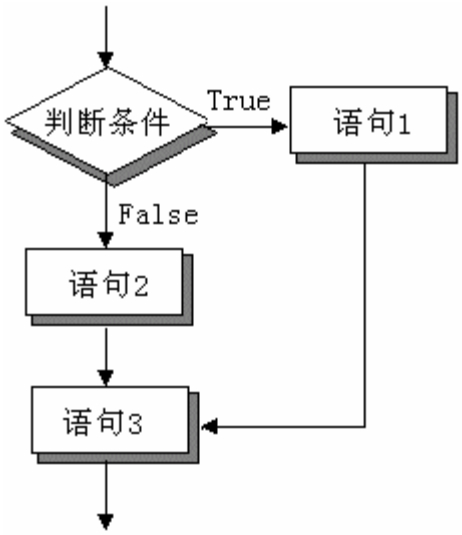


图 3-8　选择结构的基本流程

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。举例来说，想在下面的程序中声明两个整数 a 及 b，并赋其初值，如果 a 大于 b，在显示器中输出 a-b 的计算结果。无论 a 是否大于 b，最后均输出 a*b 的值。

范例：TestJava3_23.java

```
01  // 下面的程序说明了 if 语句的操作，只有当条件满足时才会被执行
02  public class TestJava3_23
03  {
04      public static void main(String[] args)
05      {
06          int a = 6 , b = 5 ;
07
08          System.out.println("a = "+a+" , b = "+b);
09          if(a>b)
10              System.out.println("a - b = "+(a-b));
11          System.out.println("a * b = "+(a*b));
12      }
13  }
```

输出结果：

```
a = 6 , b = 5
a - b = 1
a * b = 30
```

读者可以试着更改程序第 6 行中变量 a、b 的初值，将 a 的值设置得比 b 值小，可较容易观察程序运行的流程。如果输入的 a 值小于或等于 b 值，则会跳至执行第 11 行。

3.3.1.3 循环结构

循环结构则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下图 3-9 所示。

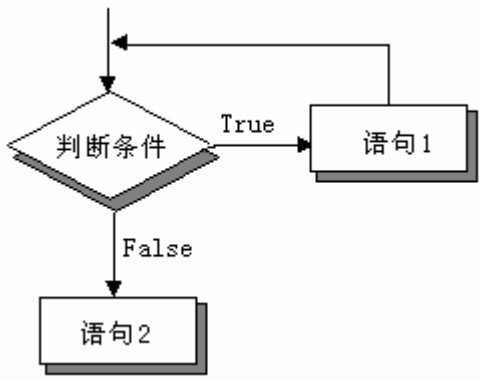


图 3-9　循环结构的基本流程

3.3.2 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。现在先来看看 if 语句。

3.3.2.1 if 语句

在前面简单地介绍了 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。

if 语句的格式如下所示：

【 格式 3-3 if 语句的格式 】

```
if (判断条件)
{
    语句 1 ；
    语句 2 ；
        …
    语句 3 ；
}
```

若是在 if 语句主体中要处理的语句只有 1 个，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 3-10 所示。

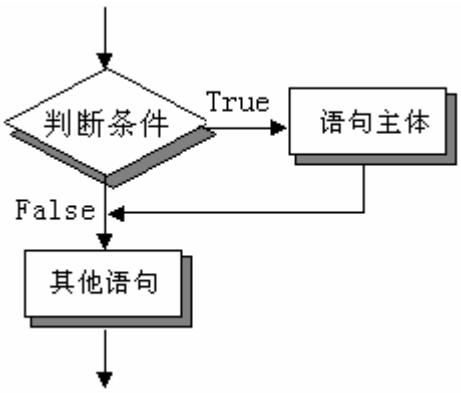


图 3-10　if 语句的流程图

选择结构中除了 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。

3.3.2.2 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，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;
}
```

若是在 if 语句或 else 语句主体中要处理的语句只有一个，可以将左、右大括号去除。if…else 语句的流程图如图 3-17 所示。

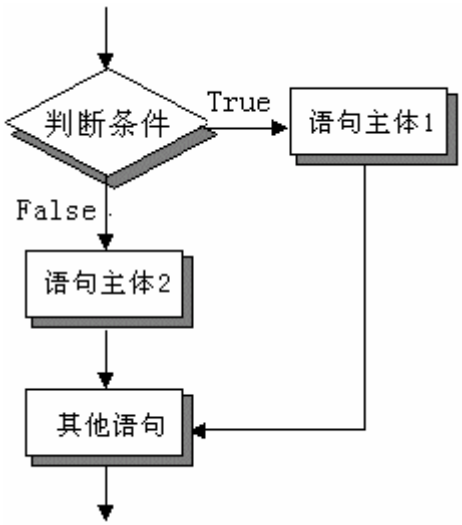


图 3-17 if..else 语句的基本流程

下面举一个简单的例子：声明一个整型变量 a，并给其赋初值 5，在程序中判断 a 是奇数还是偶数，再将判断的结果输出。

范例：TestJava3_24.java

```
01  // 以下程序说明了 if...else 的使用方法
02  public class TestJava3_24
03  {
04      public static void main(String[] args)
05      {
06          int a = 5 ;
07
08          if(a%2 = = 1)
09              System.out.println(a+" 是奇数！");
10          else
11              System.out.println(a+" 是偶数！");
12      }
13  }
```

输出结果：

5 是奇数!

程序说明：

1、 第 8~11 行为 if..else 语句。在第 8 行中，if 的判断条件为 a%2 = = 1，当 a 除以 2 取余数，若得到的结果为 1，表示 a 为奇数，若 a 除以 2 取余数得到的结果为 0，则 a 为偶数。

2、 当 a 除以 2 取余数的结果为 1 时，即执行第 9 行的语句，输出“a 为奇数!”；否则执行第 11 行，输出“a 为偶数!”。

3、 读者可以自行将变量 a 的初值更改，再重复执行程序。

从上面的程序中发现，程序的缩进在这种选择结构中起着非常重要的作用，它可以使设计者编写的程序结构层次清晰，在维护上也就比较简单。所以本书建议读者以后在编写程序时要养成缩进的好习惯。

3.3.2.3 条件运算符

还有一种运算符可以代替 if…else 语句，即条件运算符，如表 3-17 所示：

表 3-17 条件运算符

条件运算符	意义
?:	根据条件的成立与否，来决定结果为“:”前或“:”后的表达式

使用条件运算符时，操作数有 3 个，其格式如下：

【 格式 3-5 if…else 语句的格式 】

```
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 ；
```

接下来，可以试着练习用条件运算符来编写程序，在下面的程序中声明变量 a、b，并为其赋初值，再利用条件运算符判断其最大值。

范例：TestJava3_25.java

```
01  // 以下程序说明了条件运算符的使用方法
02  public class TestJava3_25
03  {
04      public static void main(String[] args)
```

```
05      {
06          int a = 5 , b = 13 , max ;
07
08          max = (a>b)?a:b ;
09
10          System.out.println("a = "+a+" , b = "+b);
11          System.out.println("最大的数是：  "+max);
12      }
13  }
```

输出结果：

```
a = 5 , b = 13
最大的数是：  13
```

程序说明：

1、 第 6 行声明变量并为其赋初值。a、b 为要比较大小的两个整数值；max 存放比较大小后的最大的那个值。

2、 第 8 行（max = (a>b)?a:b）赋值当 a>b 时，max = a，否则 max = b。

3、 第 10 行输出 a、b 的值。程序第 11 行，输出最大值。

4、 可以自行将 a、b 的值更改，再运行此程序。

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。

3.3.2.4 if..else if..else 语句

如果需要在 if..else 里判断多个条件时，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，请看下面的范例：

范例：TestJava3_26.java

```
01  // 以下程序说明了多分支条件语句 if..else if ...else 的使用
02  public class TestJava3_26
03  {
04        public static void main(String[] args)
05        {

06              int x = 1 ;
07
08              if(x= =1)
09                    System.out.println("x = = 1");
10              else if(x= =2)
11                    System.out.println("x = = 2");
12              else if(x= =3)
13                    System.out.println("x = = 3");
14              else
15                    System.out.println("x > 3");
16        }
```

```
17  }
```

输出结果：

x = = 1

可以看出 if … else if ..else 比单纯的 if..else 语句可以含有更多的条件判断语句，可是读者想一想如果有很多条件都要判断的话，这样写会不会是一件很头疼的事情，下面将为读者介绍的多重选择语句就可以为读者解决这一件头疼的问题。

3.3.3 多重选择语句——switch 语句

switch 语句可以将多选一的情况简化，而使程序简洁易懂，在本节中，将要介绍如何使用 switch 语句以及它的好伙伴——break 语句；此外，也要讨论在 switch 语句中如果不使用 break 语句会出现的问题。首先，先来了解 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 不断地判断之外，也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 配对混淆而造成阅读及运行上的错误。使用 swtich 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 语句里的选择值只能是字符或是常量。接下来看看 switch 语句执行的流程。

1、 switch 语句先计算括号中表达式的结果。

2、 根据表达式的值检测是否符合执行 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 语句的尾端才会离开 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 是跳出语句就可以了。

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 3-18 所示的 switch 语句流程图：

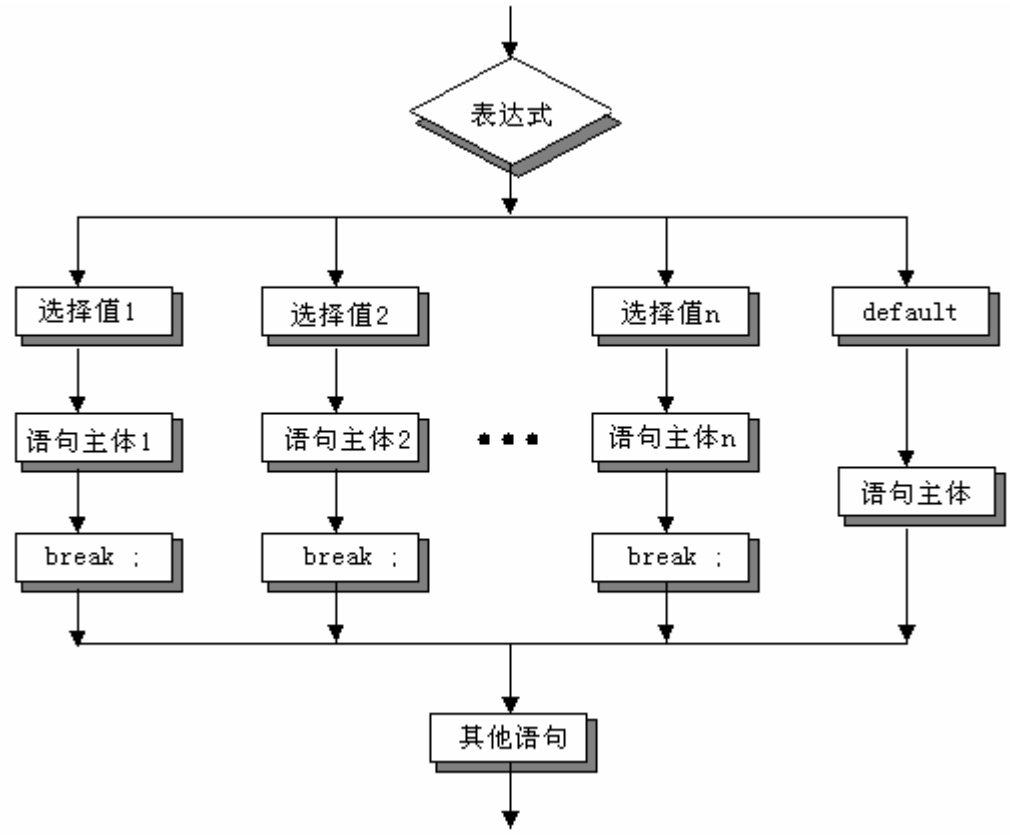


图 3-18　switch 语句的基本流程

下面的程序是一个简单的赋值表达式，利用 switch 语句处理此表达式中的运算符，再输出运算后的结果。

范例：TestJava3_27.java

```
01  // 以下程序说明了多分支条件语句的使用
02  public class TestJava3_27
03  {
04      public static void main(String[] args)
05      {
06          int a = 100 , b = 7 ;
07          char oper ='/' ;
08
09          switch(oper)        // 用 switch 实现多分支语句
10          {

11              case '+':
12                  System.out.println(a+" + "+b+" = "+(a+b));
13                  break ;
14              case '-':
15                  System.out.println(a+" - "+b+" = "+(a-b));
16                  break ;
17              case '*':
18                  System.out.println(a+" * "+b+" = "+(a*b));
19                  break ;
20              case '/':
21                  System.out.println(a+" / "+b+" = "+((float)a/b));
22                  break ;
23              default:
24                  System.out.println("未知的操作！ ");
25          }
26      }
27  }
```

输出结果：

```
100 / 7 = 14.285714
```

程序说明：

1、 第 7 行，利用变量存放一个运算符号，如 3+2、5*7 等。

2、 第 9~25 行为 switch 语句。当 oper 为字符+、-、*、/、%时，输出运算的结果后离开 switch 语句；若是所输入的运算符皆不在这些范围时，即执行 default 所包含的:语句输出“未知的操作!”，再离开 switch。

3、 选择值为字符时，必须用单引号将字符包围起来。

程序运行的结果会因为没有加上 break 语句而出现错误，所以程序设计者在使用 switch 语句的时候，要特别注意是否需要加上 break 语句。

3.3.4 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的时，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 (判断条件)
{
    语句 1 ；
    语句 2 ；
    …
    语句 n ；
}
```

当 while 循环主体有且只有一个语句时，可以将大括号除去。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、 第一次进入 while 循环前，就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；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 3-19 所示的 while 循环流程图：

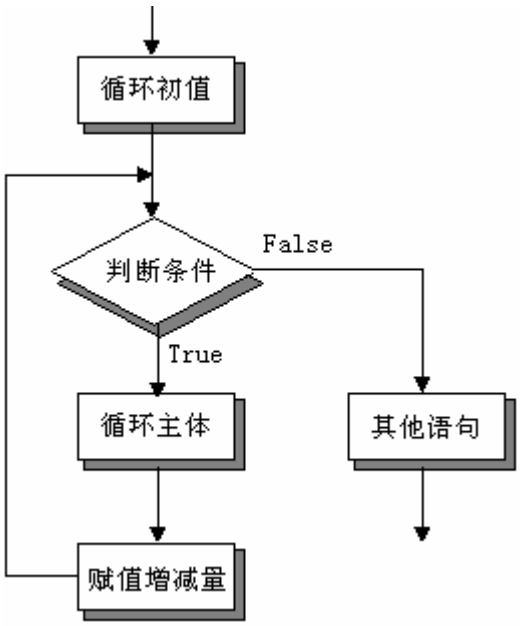


图 3-19　while 循环的基本流程

下面这道范例是循环计算 1 累加至 10：

范例：TestJava3_28.java

```
01  // 以下程序说明了 while 循环的使用方法
02  public class TestJava3_28
03  {
04      public static void main(String[] args)
05      {
06          int i = 1 ,sum = 0 ;
07
08          while(i<=10)
```

```
09              {
10                  sum += i ;        // 累加计算
11                  i++ ;
12              }
13              System.out.println("1 + 2 + ...+ 10 = "+sum);       // 输出结果
14          }
15  }
```

输出结果：

```
1 + 2 + ...+ 10 = 55
```

程序说明：

1、 在第 6 行中，将循环控制变量 i 的值赋值为 1。

2、 第 8 行进入 while 循环的判断条件为 i<=10，第一次进入循环时，由于 i 的值为 1，所以判断条件的值为真，即进入循环主体。

3、 第 9~12 行为循环主体，sum+i 后再指定给 sum 存放，i 的值加 1，再回到循环起始处，继续判断 i 的值是否仍在所限定的范围内，直到 i 大于 10 即跳出循环，表示累加的操作已经完成，最后再将 sum 的值输出即可。

3.3.5 do…while 循环

do…while 循环也是用于未知循环执行次数的时候，而 while 循环及 do…while 循环最大不同就是进入 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“先做再说”，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 ；
}while (判断条件);
```

当循环主体只有一个语句时，可以将左、右大括号去除。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...whle 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，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 3-20 所示的 do..while 循环流程图：

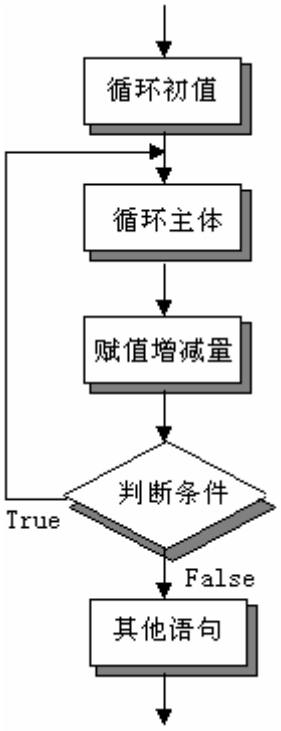


图 3-20　do…whle 循环的基本流程

把 TestJava3_28.java（1+2+…+10）的程序稍加修改，用 do…while 循环设计一个能累加至 n 的程序，并且能够限制 n 的范围（n 要大于 0），就是下面的范例 TestJava3_29.java：

范例：TestJava3_29.java

```
01  // 以下程序说明了 do...while 循环的使用
02  public class TestJava3_29
03  {
04      public static void main(String[] args)
05      {
06          int i = 1 ,sum = 0 ;
07          // do.while 是先执行一次，再进行判断。即，循环体至少会被执行一次
08          do
09          {
10              sum += i ;                              // 累加计算
```

```
11                  i++ ;
12              }while(i<=10);
13              System.out.println("1 + 2 + ...+ 10 = "+sum);       // 输出结果
14          }
15  }
```

输出结果：

1 + 2 + ...+ 10 = 55

首先，声明程序中要使用的变量 i（循环记数及累加操作数）及 sum（累加的总和），并将 sum 设初值为 0；由于要计算 1+2+…+10，因此在第一次进入循环的时候，将 i 的值设为 1，接着判断 i 是否小于等于 10，如果 i 小于等于 10，则计算 sum+i 的值后再指定给 sum 存放。i 的值已经不满足循环条件时，i 即会跳出循环，表示累加的操作已经完成，再输出 sum 的值，程序即结束运行。

程序说明：

1、 第 08~12 行利用 do…while 循环计算 1~10 的数累加

2、 第 13 行，输出 1~10 的数的累加结果：1 + 2 + ...+ 10 = 55

do..while 循环不管条件是什么，都是先做再说，因此循环的主体最少会被执行一次。在日常生活中，如果能够多加注意，并不难找到 do…while 循环的影子！举例来说，在利用提款机提款前，会先进入输入密码的画面，让使用者输入三次密码，如果皆输入错误，即会将银行卡吞掉，其程序的流程就是利用 do…while 循环设计而成的。

3.3.6 for 循环

当很明确地知道循环要执行的次数时，就可以使用 for 循环，其语句格式如下：

【 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 1 个，可以将大括号去除。下面列出了 for 循环的流程。

1、 第一次进入 for 循环时，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，再回到步骤 2 重新判断是否继续执行循环。

根据上述描述，可以绘制出如图 3-21 所示的 for 循环流程图。

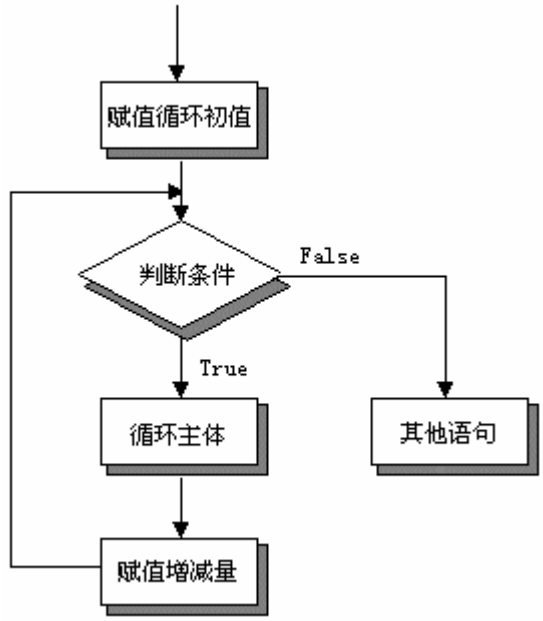


图 3-21　for 循环的基本流程

范例 TestJava3_30 可以使读者熟悉 for 循环的使用，它是利用 for 循环来完成由 1 至 10 的数的累加运算。

范例：TestJava3_30.java

```
01  // 以下程序说明了 for 循环的使用方法
02  public class TestJava3_30
03  {
04      public static void main(String[] args)
05      {
06          int i , sum = 0 ;
07          // for 循环的使用，用来计算数字累加之和
08          for(i=1;i<=10;i++)
09              sum += i ;          // 计算 sum = sum+i
10          System.out.println("1 + 2 + ... + 10 = "+sum);
11      }
12  }
```

输出结果：

1 + 2 + ... + 10 = 55

程序说明：

1、 在第 06 行声明两个变量 sum 和 i，I 用于循环的记数控制。

2、 08~09 行，做 1~10 之间的循环累加，执行的结果如 TestJava3_28，如果读者不明白的话，可以和 TestJava3_28 的程序说明比较一下，相信就可以明白 for 的用法了。

3.3.7 循环嵌套

当循环语句中又出现循环语句时，就称为嵌套循环。如嵌套 for 循环、嵌套 while 循环等，当然读者也可以使用混合嵌套循环，也就是循环中又有其他不同种类的循环。以打印九九乘法表为例，练习嵌套循环的用法。

范例：TestJava3_31.java

```
01  // 以下程序说明了 for 循环的嵌套使用方法
02  public class TestJava3_31
03  {
04      public static void main(String[] args)
05      {
06          int i , j ;
07          // 用两层 for 循环输出乘法表
08          for(i=1;i<=9;i++)
09          {
10              for(j=1;j<=9;j++)
11                  System.out.print(i+"*"+j+"="+(i*j)+"\t");
12              System.out.print("\n");
13          }
14      }
15  }
```

输出结果：

1*1=1 1*2=2 1*3=3 1*4=4 1*5=5 1*6=6 1*7=7 1*8=8 1*9=9
2*1=2 2*2=4 2*3=6 2*4=8 2*5=10 2*6=12 2*7=14 2*8=16 2*9=18
3*1=3 3*2=6 3*3=9 3*4=12 3*5=15 3*6=18 3*7=21 3*8=24 3*9=27
4*1=4 4*2=8 4*3=12 4*4=16 4*5=20 4*6=24 4*7=28 4*8=32 4*9=36
5*1=5 5*2=10 5*3=15 5*4=20 5*5=25 5*6=30 5*7=35 5*8=40 5*9=45
6*1=6 6*2=12 6*3=18 6*4=24 6*5=30 6*6=36 6*7=42 6*8=48 6*9=54
7*1=7 7*2=14 7*3=21 7*4=28 7*5=35 7*6=42 7*7=49 7*8=56 7*9=63
8*1=8 8*2=16 8*3=24 8*4=32 8*5=40 8*6=48 8*7=56 8*8=64 8*9=72
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<=**9**），进入循环主体，输出 i*j 的值（1*1=1），j 再加 1 等于 2，仍符合内层 for 循环的判断条件（j<=**9**），再次执行计算与输出的工作，直到 j 的值大于 9 即离开内层 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 时即离开嵌套循环。

3、 整个程序到底执行了几次循环呢？可以看到，当 i 为 1 时，内层循环会执行 9 次（j 为 1~9）,当 i 为 2 时，内层循环也会执行 9 次（j 为 1~9），以此类推的结果，这个程序会执行 81 次循环，而显示器上也正好输出 81 个式子。

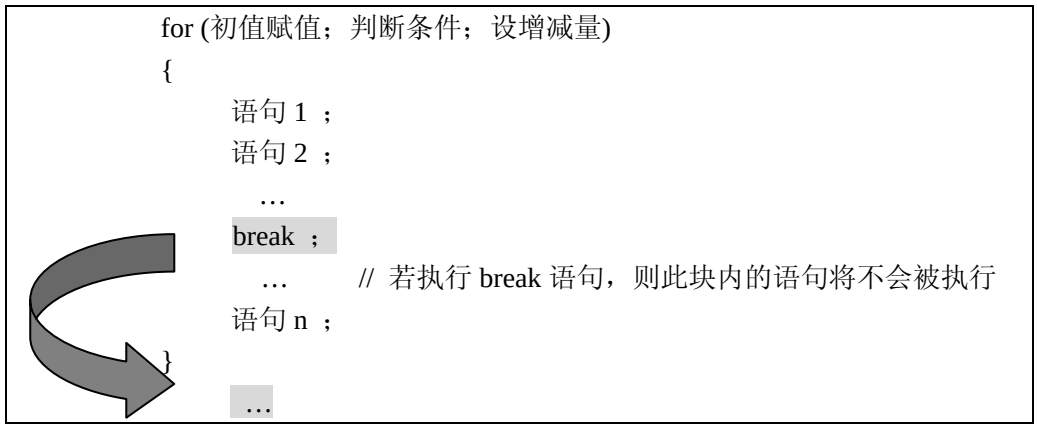
3.3.8 循环的跳离

在 Java 语言中，有一些跳离的语句，如 break、continue 等语句，站在结构化程序设计的角度上，并不鼓励用户使用，因为这些跳离语句会增加调试及阅读上的困难。因此建议读者：除非在某些不得以的情况下之外，否则尽量不要去使用它们。在本节中，将为读者介绍 break 及 continue 语句。

3.3.8.1 break 语句

break 语句可以强迫程序跳离循环，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 语句只会跳离当前层的循环。以下图的 for 循环为例，在循环主体中有 break 语句时，当程序执行到 break，即会离开循环主体，而继续执行循环外层的语句。

【 格式 3-12 break 语句格式】



以下面的程序为例，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，即使用 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：TestJava3_32.java

```
01  // 下面的程序是介绍 break 的使用方法
02  public class TestJava3_32
03  {
04      public static void main(String[] args)
05      {
06          int i ;
07
08          for(i=1;i<=10;i++)
09          {
10              if(i%3 == 0)
```

```
11                      break ;                 // 跳出整个循环体
12                  System.out.println("i = "+i);
13              }
14              System.out.println("循环中断：i = "+i);
15          }
16  }
```

输出结果：

i = 1

i = 2

循环中断：i = 3

程序说明：

1、 第 9~13 行为循环主体，i 为循环的控制变量。

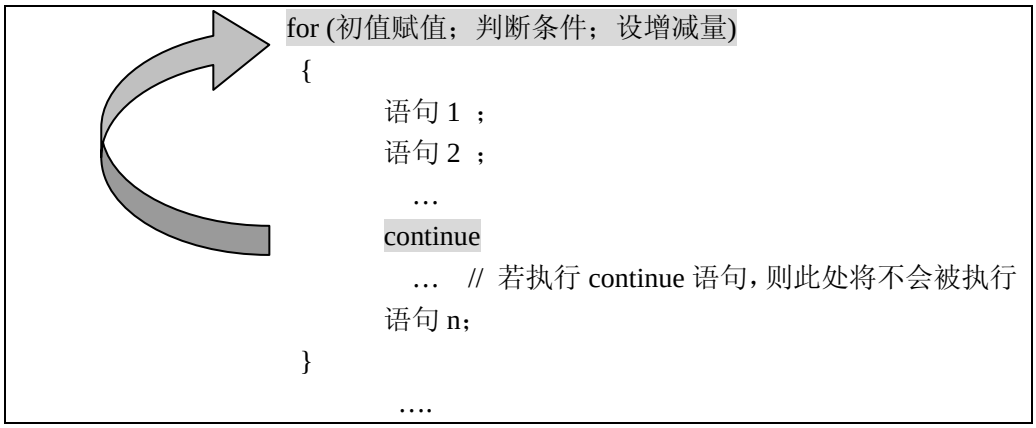
2、 当 i%3 为 0 时，符合 if 的条件判断，即执行第 11 行的 break 语句，跳离整个 for 循环。此例中，当 i 的值为 3 时，3%3 的余数为 0，符合 if 的条件判断，离开 for 循环，执行第 14 行：输出循环结束时循环控制变量 i 的值 3。

通常设计者都会设定一个条件，当条件成立时，不再继续执行循环主体。所以在循环中出现 break 语句时，if 语句通常也会同时出现。此外，读者可以回想一下前面提到过的 switch 语句中是不是也有一个 break，如果记不清楚的读者可以翻看一下本章前面的内容。

3.3.8.2 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，即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。以下图的 for 循环为例，在循环主体中有 continue 语句，当程序执行到 continue，即会回到循环的起点，继续执行循环主体的部分语句。

【 格式 3-13 continue 语句格式】



将程序 TestJava3_32 中的 break 语句改成 continue 语句就形成了程序 TestJava3_33.java。读者可以观察一下这两种跳离语句的不同。break 语句是跳离当前层循环，而 continue 语句是回到循环的起点。程序如下所示：

范例：TestJava3_33.java

```
01  // 下面的程序是介绍 continue 的使用方法
02  public class TestJava3_33
03  {
04      public static void main(String[] args)
05      {
06          int i ;
07
08          for(i=1;i<=10;i++)
09          {
10              if(i%3==0)
11                  continue ;              // 跳出一次循环
12              System.out.println("i = "+i);
13          }
14          System.out.println("循环中断：i = "+i);
15      }
16  }
```

输出结果：

i = 1

i = 2

i = 4

i = 5

i = 7

i = 8

i = 10

循环中断：i = 11

程序说明：

1、 第 9~13 行为循环主体，i 为循环控制变量。

2、 当 i%3 为 0 时，符合 if 的条件判断，即执行第 11 行的 continue 语句，跳离目前的 for 循环（不再执行循环体内的其他的语句），而是回到循环开始处继续判断是否执行循环。此例中，当 i 的值为 3、6、9 时，取余数为 0，符合 if 判断条件，离开当前层的 for 循环，再回到循环开始处继续判断是否执行循环。

3、 当 i 的值为 11 时，不符合循环执行的条件，此时执行程序第 14 行：输出循环结束时循环控制变量 i 的值 11。

当判断条件成立时，break 语句与 continue 语句会有不同的执行方式。Break 语句不管情况如何，先离开循环再说；而 continue 语句则不再执行此次循环的剩余语句，直接回到循环的起始处。

3.3.9 局部变量

Java 可以在程序的任何地方声明变量，当然也可以在循环里声明。有趣的是，在循环里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。下面以一个范例来说明局部变量的使用方法。

范例：TestJava3_34.java

```
01  // 以下程序说明了 for 循环的使用方法
02  public class TestJava3_34
03  {
04      public static void main(String[] args)
05      {
06          int sum = 0 ;
07          // 下面是 for 循环的使用，计算 1~5 数字累加之和
08          for(int i=1;i<=5;i++)
09          {
10              sum = sum + i ;
11              System.out.println("i = "+i+", sum = "+sum);
12          }
13      }
14  }
```

变量 i 的有效范围

输出结果：

```
i = 1, sum = 1
i = 2, sum = 3
i = 3, sum = 6
i = 4, sum = 10
i = 5, sum = 15
```

在 TestJava3_34 中，把变量 i 声明在 for 循环里，因此变量 i 在此就是局部变量，它的有效范围仅在 for 循环内（8~12 行），只要一离开这个循环，变量 i 便无法使用。相对的，变量 sum 是声明在 main() 方法的开始处，因此它的有效范围从第 6 行开始到第 12 行结束，当然，for 循环内也是属于变量 sum 的有效范围。

• 本章摘要：

1、 Java 的数据类型可分为下列两种：基本数据类型和引用数据类型。

2、 Java 提供 long、int、short 及 byte 四种整数类型的最大值、最小值的代码。最大值的代码是 MAX_VALUE，最小值是 MIN_VALUE。如果使用某个类型的最大值或最小值，只要在这些代码之前，加上它们所属的类的全名即可。

3、 Unicode，它为每个字符制订了一个唯一的数值，如此在任何的语言、平台、程序都可以安心地使用。

4、 布尔（boolean）类型的变量，只有 true（真）和 false（假）两个值。

5、 数据类型的转换可分为下列两种：“自动类型转换”与“强制类型转换”。

6、 表达式是由操作数与运算符所组成的。

7、 一元运算符只需要一个操作数。如“+3”、“~a”、“-a”与“!a”等均是由一元运算符与一个操作数所组成的。

8、 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、余数运算符。

9、 if 语句可依据判断的结果来决定程序的流程。

10、递增与递减运算符有着相当大的便利性，善用它们可提高程序的简洁程度，其成员请参照表 4-5。

11、括号（）是用来处理表达式的优先级的，也是 Java 的运算符。

12、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，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，则其它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。

13、程序的结构包含：（1）顺序结构、（2）选择结构、（3）循环结构。

14、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 循环来完成。

15、break 语句可以让强制程序逃离循环。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 语句只会逃离当前层循环。

16、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，而到循环的开始处继续运行。

17、选择结构包括了 if、if-else 及 switch 语句，语句中加上了选择的结构之后，就像是十字路口，根据不同的选择，程序的运行会有不同的方向与结果。

18、在循环里也可以声明变量，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。

第四章、数组与方法

若想要存放一连串相关的数据，使用数组是个相当好用的选择。此外，如果某个程序片段经常反复出现，那么将它定义成一个方法可以有效地简化程序代码。本章的中心是在介绍数组的基本用法与方法的应用，学完本章，将会对数组与方法的使用有更深一层的认识。

数组是由一组相同类型的变量所组成的数据类型，它们以一个共同的名称表示，数组中的个别元素则以标注来表示其存放的位置。数组依照存放元素的复杂程度分为一维数组、二维和多维数组，本章先从一维数组谈起。

4.1 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。

4.1.1 一维数组的声明与内存的分配

要使用 Java 的数组，必须经过两个步骤：（1）声明数组、（2）分配内存给该数组。这两个步骤的语法如下：

【 格式 4-1 一维数组的声明与分配内存】

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

数组的声明格式里，“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，其命名规则和变量的相同，建议读者使用有意义的名称为数组命名。数组声明后，接下来便是要配置数组所需的内存，其中“个数”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个数，在内存中开辟一块内存供该数组使用。下面是关于一维数组的声明并分配内存给该数组的一个范例：

```
int score[] ;                // 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 4
```

在上例中的第一行，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，编译器仅会分配一块内存给它，用来保存指向数组实体的地址，如图 4-1 所示。

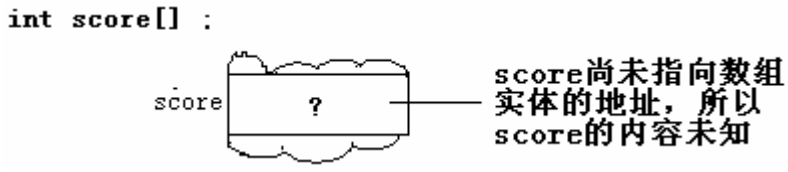


图 4-1　声明整型数组

声明之后，接着要做内存分配的操作，也就是上例中第二行语句。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。其内存分配的流程如图 4-2 所示。

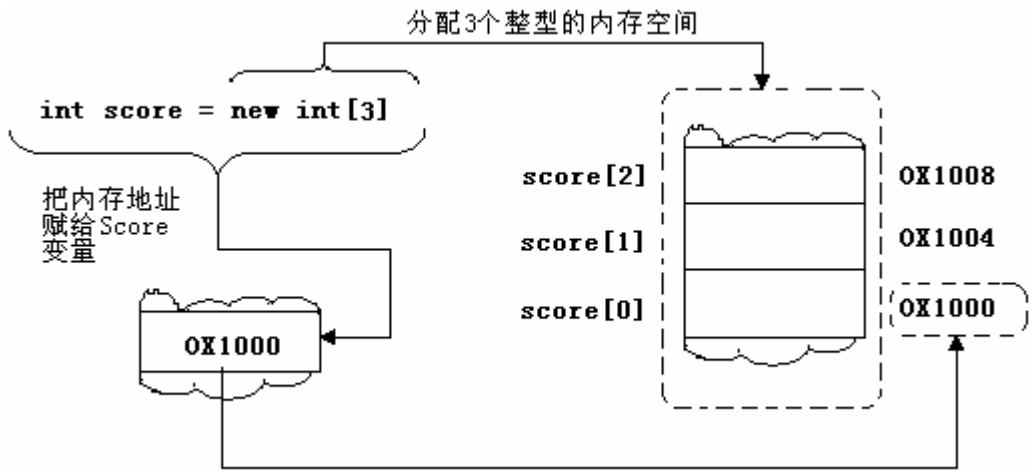


图 4-2　内存分配

上图中的内存参考地址 Ox1000 是假赋值，此值会因环境的不同而异。如第 3 章所述，数组是属于非基本数据类型，因此数组变量 score 所保存的并非是数组的实体，而是数组实体的参考地址。

除了用格式 4-1 的这两行来声明并分配内存给数组之外，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，即分配一块内存空间，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 11 个整数的内存给 score 变量。

```
int score[] = new int[11] ;
// 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 bytes，而整型数组 score 可保存的元素有 **11 个**，所以上例中占用的内存共有 4 * 11 = 44 个字节。图 4-3 是将数组 score 用图形来表示，读者可以比较容易理解数组的保存方式。

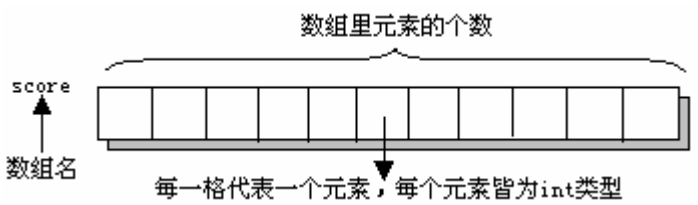


图 4-3　数组的保存方式

4.1.2　数组中元素的表示方法

想要使用数组里的元素，可以利用索引来完成。Java 的数组索引编号由 0 开始，以上一节中的 score 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 4-4 为 score 数组中元素的表示法及

排列方式：

图 4-4　数组中元素的排列

接下来，看一个范例。下面的程序里，声明了一个一维数组，其长度为 3，利用 for 循环输出数组的内容后，再输出数组的元素个数。

范例：TestJava4_1.java

```
01  // 下面这段程序说明了一维数组的使用方法
02  public class TestJava4_1
03  {
04      public static void main(String args[])
05      {
06          int i;
07          int a[];              // 声明一个整型数组 a
08          a=new int[3];         // 开辟内存空间供整型数组 a 使用，其元素个数为 3
09
10          for(i=0;i<3;i++)          // 输出数组的内容
11              System.out.print("a["+i+"] = "+a[i]+",\t");
12
13          System.out.println("\n 数组长度是：  "+a.length);     // 输出数组长度
14      }
15  }
```

输出结果：

```
a[0] = 0,  a[1] = 0,  a[2] = 0,
数组长度是：  3
```

程序说明：

1、 第 7 行声明整型数组 a；第 8 行开辟了一块内存空间，以供整型数组 a 使用，其元素个数为 3。

2、 第 10~11 行，利用 for 循环输出数组的内容。由于程序中并未给予数组元素赋值，因此输出的结果都是 0。

3、 第 13 行输出数组的长度。此例中数组的长度是 3，即代表数组元素的个数有 3 个。

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

【 格式 4-3 数组长度的取得】

数组名.length

也就是说，若是要取得 TestJava4_1 中所声明的数组 a 的元素个数，只要在数组 a 的名称后面加上“.length”即可，如下面的程序片段：

```
a.length ;              // 取得数组的长度
```

4.1.3 数组初值的赋值

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，在声明的时候，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 day[] = {32,23,45,22,13,45,78,96,43,32};　　// 数组声明并赋初值

在上面的语句中，声明了一个整型数组 day，虽然没有特别指明数组的长度，但是由于大括号里的初值有 10 个，编译器会分别依序指定给各元素存放，day[0]为 32，day[1]为 23，…，day[9]为 32。

范例：TestJava4_2.java

```
01  // 一维数组的赋值，这里采用静态方式赋值
02  public class TestJava4_2
03  {
04      public static void main(String args[])
05      {
06          int i;
07          int a[]={5,6,8};              // 声明一个整数数组 a 并赋初值
08
09          for(i=0;i<a.length;i++)       // 输出数组的内容
10              System.out.print("a["+i+"]="+a[i]+",\t");
11
12          System.out.println("\n 数组长度是："+a.length);
13      }
14  }
```

输出结果：

```
a[0]=5,    a[1]=6,    a[2]=8,
数组长度是：3
```

除了在声明时就赋初值之外，也可以在程序中为某个特定的数组元素赋值。可以将程序 TestJava4_2 的第 7 行更改成下面的程序片段：

```
int a [] = new int[] ;
a[0] = 5 ;
a[1] = 6 ;
a[2] = 8 ;
```

4.1.4 简单的范例：找出数组元素中的最大值与最小值

由前几节的范例可知，数组的索引就好象饭店房间的编号一样，想要找到某个房间时，就得先找到房间编号！接下来再举一个例子，说明如何将数组里的最大值及最小值列出。

范例：TestJava4_3.java

```
01  // 这个程序主要是求得数组中的最大值和最小值
02  public class TestJava4_3
03  {
04      public static void main(String args[])
05      {
06          int i,min,max;
07          int A[]={74,48,30,17,62};              // 声明整数数组 A,并赋初值
08
09          min=max=A[0];
10          System.out.print("数组 A 的元素包括：  ");
11          for(i=0;i<A.length;i++)
12          {
13              System.out.print(A[i]+"   ");
14              if(A[i]>max)                       // 判断最大值
15                  max=A[i];
16              if(A[i]<min)                       // 判断最小值
17                  min=A[i];
18          }
19          System.out.println("\n 数组的最大值是："+max);    // 输出最大值
20          System.out.println("数组的最小值是："+min);       // 输出最小值
21      }
22  }
```

输出结果：

数组 A 的元素包括： 74 48 30 17 62

数组的最大值是：74

数组的最小值是：17

程序说明：

1、 第 6 行声明整数变量 i 做为循环控制变量及数组的索引：另外也声明存放最小值的变量 min 与最大值的变量 max。

2、 第 7 行声明整型数组 A，其数组元素有 5 个，其值分别为 74、48、30、17、62。

3、 第 9 行将 min 与 max 的初值设为数组的第一个元素。

4、 第 10~18 行逐一输出数组里的内容，并判断数组里的最大值与最小值。

5、 第 19~20 行输出比较后的最大值与最小值。

将变量 min 与 max 初值设成数组的第一个元素后，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；同样的，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。

4.1.5 与数组操作有关的 API 方法

在 Java 语言中提供了许多的 API 方法，供开发人员使用，下面介绍两种常用的数组操作方法，一个是数组的拷贝操作，另一个是数组的排序操作。其它的操作请读者自行查阅 JDK 帮助文档。

范例：TestJava4_4.java

```
01  // 以下这段程序说明数组的拷贝操作
02  public class TestJava4_4
03  {
04      public static void main(String[] args)
```

```
05        {
06            int a1[] = {1,2,3,4,5} ;  //声明两个整型数组 a1、a2，并进行静态初始化
07            int a2[] = {9,8,7,6,5,4,3} ;
08            System.arraycopy(a1,0,a2,0,3);          // 执行数组拷贝的操作
09            System.out.print("a1 数组中的内容：");
10            for(int i=0;i<a1.length;i++)            // 输出 a1 数组中的内容
11                System.out.print(a1[i]+"   ");
12            System.out.println();
13
14            System.out.print("a2 数组中的内容：");
15            for(int i=0;i<a2.length;i++)            //输出 a2 数组中的内容
16                System.out.print(a2[i] +"   ");
17            System.out.println("\n 数组拷贝完成！");
18        }
19    }
```

输出结果：

a1 数组中的内容：1 2 3 4 5

a2 数组中的内容：1 2 3 6 5 4 3

数组拷贝完成!

System.arrayCopy(source,0,dest,0,x)：语句的意思就是：复制源数组从下标 0 开始的 x 个元素到目标数组，从目标数组的下标 0 所对应的位置开始存取。

范例：TestJava4_5.java

```
01  // 以下程序是数组的排序操作，在这里使用了 sort 方法对数组进行排序
02  import java.util.*;
03  public class TestJava4_5
04  {
05      public static void main(String[] args)
06      {
07          int a[] = {4,32,45,32,65,32,2} ;
08
09          System.out.print("数组排序前的顺序：");
10          for(int i=0;i<a.length;i++)
```

```
11                  System.out.print(a[i]+"    ");
12              Arrays.sort(a);             // 数组的排序方法
13              System.out.print("\n 数组排序后的顺序：");
14              for(int i=0;i<a.length;i++)
15                  System.out.print(a[i]+"    ");
16          }
17  }
```

输出结果：

数组排序前的顺序：4　32　45　32　65　32　2

数组排序后的顺序：2　4　32　32　32　45　65

程序第 12 行的 Arrays.sort(数组名)为数组排序的操作，但这个方法在 java.util 这个包里面，所以在用到的时候需要先将它导入，至于包的概念，本书以后章节会讲到。

4.2　二维数组

虽然一维数组可以处理一般简单的数据，但是在实际的应用上仍显不足，所以 Java 也提供了二维数组以及多维数组供程序设计人员使用。学会了如何使用一维数组后，再来看看二维数组的使用方法。

4.2.1　二维数组的声明与分配内存

二维数组声明的方式和一维数组类似，内存的分配也一样是用 new 这个关键字。其声明与分配内存的格式如下所示：

【 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
```

同样的，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。编写的范例如下：

```
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。

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

【 格式 4-7 二维数组赋初值的格式】

```
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里不必填入任何的内容。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。如下面的关于数组 num 声明及赋初值的范例：

```
int num[][] = {
                {23,45,21,45},          // 二维数组的初值赋值
                {45,23,46,23}
              };
```

在上面的语句中，声明了一个整型数组 num，数组有 2 行 4 列共 8 个元素，大括号里的几组初值会分别依序指定给各行里的元素存放，num[0][0]为 23，num[0][1]为 45，…，num[1][3]为 23。

4.2.1.1 每行的元素个数不同的二维数组

值得一提的是 Java 允许二维数组中每行的元素个数均不相同，这点与一般的程序语言是不同的。例如，下面的语句是声明整型数组 num 并赋初值，而初值的赋值指明了 num 具有三行元素，其中第一行有 4 个元素，第二行有 3 个元素，第三行则有 5 个元素：

```
int num[][] = {
                {42,54,34,67},
                {33,34,56},
                {12,34,56,78,90}
              };
```

4.2.1.2 取得二维数组的行数与特定行的元素的个数

在二维数组中，若是想取得整个数组的行数，或者是某行元素的个数时，可利用“.length”来获取，其语法如下：

【 格式 4-8 取得二维数组的行数与特定行的元素的个数】

```
数组名.length                  // 取得数组的行数
数组名[行的索引].length     // 取得特定行元素的个数
```

也就是说，如要取得二维数组的行数，只要用数组名加上“.length”即可；如要取得数组中特定行的元素的个数，则须在数组名后面加上该行的索引值，再加上“.length”，如下面的程序片段：

```
num.length;          // 计算数组 num 的行数，其值为 3
num[0].length        // 计算数组 num 的第 1 行元素的个数，其值为 4
num[2].length        // 计算数组 num 的第 3 行元素的个数，其值为 5
```

4.2.2 二维数组元素的引用及访问

二维数组元素的输入与输出方式与一维数组相同，看下面这个范例：

范例：TestJava4_6.java

```
01  // 二维数组的使用说明，这里采用静态赋值的方式
02  public class TestJava4_6
03  {
04      public static void main(String args[])
05      {
06          int i,j,sum=0;
07          int num[][]={{30,35,26,32},{33,34,30,29}};      // 声明数组并设置初值
08
09          for(i=0;i<num.length;i++)                  // 输出销售量并计算总销售量
10          {
11              System.out.print("第 "+(i+1)+" 个人的成绩为：");
12              for(j=0;j<num[i].length;j++)
13              {
14                  System.out.print(num[i][j]+" ");
15                  sum+=num[i][j];
16              }
17                System.out.println();
```

```
18              }
19              System.out.println("\n 总成绩是 "+sum+" 分！");
20          }
21      }
```

输出结果：

```
第 1 个人的成绩为：30 35 26 32
第 2 个人的成绩为：33 34 30 29
总成绩是 249 分！
```

程序说明：

1、 第 6 行声明整数变量 i、j 做为外层与内层循环控制变量及数组的索引，i 控制行的元素，j 控制列的元素；而 sum 则使用来存放所有数组元素值的和，也就是总成绩。

2、 第 7 行声明一整型数组 num，并为数组元素赋初值，该整型数组共有 8 个元素。

3、 第 9~18 行输出数组里各元素的内容，并进行成绩汇总。

4、 第 19 行输出 sum 的结果即为总销售量。

4.3 多维数组

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

使用多维数组时，输入、输出的方式和一、二维相同，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。以三维数组为例，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：TestJava4_7.java

```
01  // 下面程序说明了三维数组的使用方法，要输出数组的内容需要采用三重循环
02  public class TestJava4_7
```

```
{
    public static void main(String args[])
    {
        int i,j,k,sum=0;
        int A[][][]={{{5,1},{6,7}},{{9,4},{8,3}}};      // 声明数组并设置初值
        // 三维数组的输出需要采用三层 for 循环方式输出
        for(i=0;i<A.length;i++)                          // 输出数组内容并计算总和
            for(j=0;j<A[i].length;j++)
                for(k=0;k<A[j].length;k++)
                {
                    System.out.print("A["+i+"]["+j+"]["+k+"]=");
                    System.out.println(A[i][j][k]);
                    sum+=A[i][j][k];
                 }
              System.out.println("sum="+sum);
    }
}
```

输出结果：

```
A[0][0][0]=5
A[0][0][1]=1
A[0][1][0]=6
A[0][1][1]=7
A[1][0][0]=9
A[1][0][1]=4
A[1][1][0]=8
A[1][1][1]=3
sum=43
```

由于使用的是三维数组，所以嵌套循环有三层。

4.4 方法

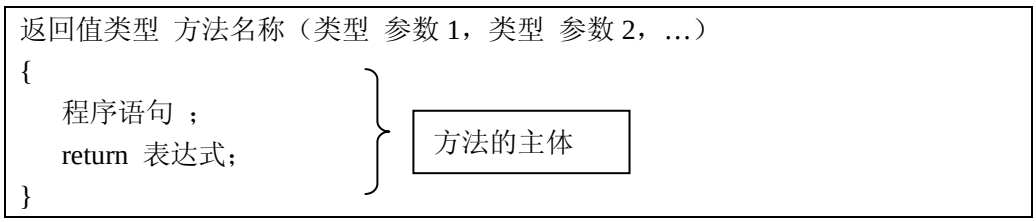
方法可以简化程序的结构，也可以节省编写相同程序代码的时间，达到程序模块化的目的。

其实读者对方法应该不陌生，在每一个类里出现的 main()即是一个方法。使用方

法来编写程序代码有相当多的好处，它可简化程序代码、精简重复的程序流程，并把具有特定功能的程序代码独立出来，使程序的维护成本降低。

方法可用如下的语法来定义：

【 格式 4-9 声明方法，并定义其内容】



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 return 语句可以省略。

4.4.1 方法操作的简单范例

TestJava4_8 是一个简单的方法操作范例，它在显示器上先输出 19 个星号“*”，换行之后再输出“I Like Java!”这一字符串，最后再输出 19 个星号。

范例：TestJava4_8.java

```
01  // 以下程序主要说明如何去声明并使用一个方法
02  public class TestJava4_8
03  {
04      public static void main(String args[])
05      {
06          star();                              // 调用 star() 方法
07          System.out.println("I Like Java !");
08          star();                              //调用 star() 方法
09      }
10
11      public static void star()                // star() 方法
12      {
13          for(int i=0;i<19;i++)
```

```
14                  System.out.print("*");          // 输出 19 个星号
15              System.out.print("\n");              // 换行
16          }
17  }
```

输出结果：

```
********************
I Like Java !
********************
```

TestJava4_8 中声明了两个方法，分别为 main()方法与 star()方法。因为 main()方法是程序进入的起点，所以把调用 star()的程序代码编写在 main()里。在 main()的第 6 行调用 start() 方法，此时程序的运行流程便会进到 11~16 行的 star()方法里执行。执行完毕后，程序返回 main()方法，继续运行第 7 行，输出“I Like Java ！”字符串。

接着第 8 行又调用 sart()方法，程序再度进到第 11~16 行的 star()方法里运行。运行完后，返回 main()方法里，因 main()方法接下来已经没有程序代码可供执行，于是结束程序 TestJava4_8。

从本程序中，可以很清楚地看出，当调用方法时，程序会跳到被调用的方法里去运行，结束后则返回原调用处继续运行。在 TestJava4_8 中，调用与运行 star()方法的流程如图 4-5 所示：

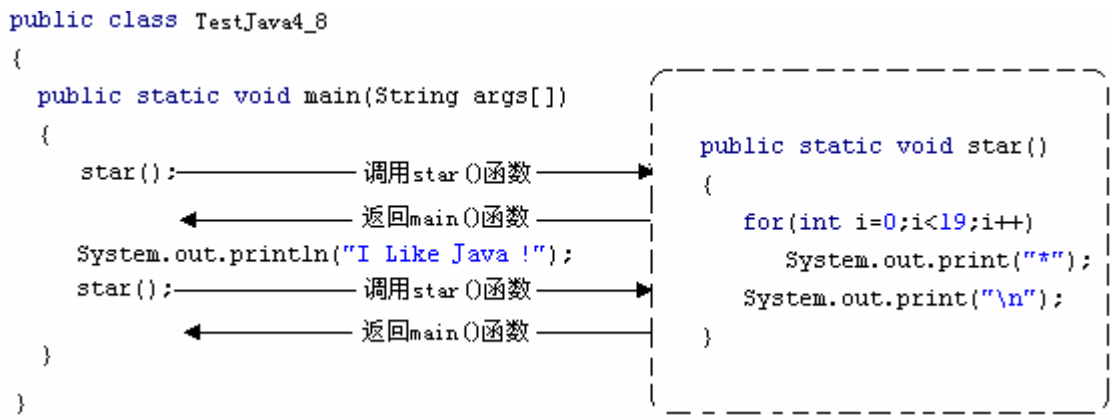


图 4-5　调用与运行 star()方法的流程

不知道读者是否注意到，在程序 TestJava4_8 中 star()方法并没有任何返回值，所以 star()方法前面加上了一个 void 关键字。此外，因为 star()没有传递任何的参数，所以 star()方法的括号内保留空白即可。

至于在 star() 方法之前要加上 static 关键字，这是因为 main()方法本身也声明成 static，而在 static 方法内只能访问到 static 成员变量（包括数据成员和方法成员）之故，因 star()方法被 main()方法所调用，自然也要把 star()声明成 static 才行。此时如果还不了解 static 的真正用意也没有关系，本书将在以后的章节对 static 关键字做详尽的介绍。

4.4.2 方法的参数与返回值

如果方法有返回值，则在声明方法之前就必须指定返回值的数据类型。相同的，如果有参数要传递到方法内，则在方法的括号内必须填上该参数及其类型。TestJava4_9 是用来说明方法的使用的另一个范例，它可以接收一个整数参数 n，输出 2*n 个星号后，返回整数 2*n。

范例：TestJava4_9

```
01  // 以下程序是关于方法的返回类型是整型的范例
02  public class TestJava4_9
03  {
04      public static void main(String args[])
05      {
06          int num;
07          num=star(7);              // 输入 7 给 star()，并以 num 接收返回的数值
08          System.out.println(num+" stars printed");
09      }
10
11      public static int star(int n)          // star() method
12      {
13          for(int i=1;i<=2*n;i++)
14              System.out.print("*");     // 输出 2*n 个星号
15          System.out.print("\n");         // 换行
16          return 2*n;                    // 返回整数 2*n
17      }
```

```
18  }
```

输出结果：

```
**************
14 stars printed
```

在 TestJava4_9 中，因 star()传递整数值，所以第 11 行的声明要在 star() 方法之前加上 int 关键字，此外，因要传入一个整数给 star()，所以 star()的括号内也要注明参数的名称与数据类型：

如果要传递一个参数，只要在方法的括号内填上所要传入的参数名称与类型即可。TestJava4_10 是一个关于计算长方形对角线长度的范例，其中 show_length()方法可接收长方形的宽与高，计算后返回对角线的长度。

范例：TestJava4_10.java

```
01  // 以下的程序说明了方法的使用
02  public class TestJava4_10
03  {
04      public static void main(String args[])
05      {
06          double num;
07          num=show_length(22,19);    // 输入 22 与 19 两个参数到 show_length()里
08          System.out.println("对角线长度 = "+num);
09      }
10
11      public static double show_length(int m, int n)
12      {
13          return Math.sqrt(m*m+n*n);        // 返回对角线长度
14      }
15  }
```

输出结果：

```
对角线长度 = 29.068883707497267
```

TestJava4_10 的第 7 行调用 show_length(22,19)，把整数 22 和 19 传入 show_length()

方法中。第 13 行则利用 Math 类里的 sqrt()方法计算对角线长度。而 sqrt(n)的作用是将参数 n 开根号。因 sqrt()的返回值是 double 类型，因此 show_length()返回值也是 double 类型。

4.4.3 方法的重载

方法的重载就是在同一个类中允许同时存在一个以上的同名方法，只要它们的参数个数或类型不同即可。在这种情况下，该方法就叫被重载了，这个过程称为方法的重载。

范例：TestJava4_11.java

```
01  // 以下程序说明了方法的重载操作
02  public class TestJava4_11
03  {
04      public static void main(String[] args)
05      {
06          int int_sum ;
07          double double_sum ;
08          int_sum = add(3,5) ;          // 调用有两个参数的 add 方法
09          System.out.println("int_sum = add(3,5)的值是："+int_sum);
10          int_sum = add(3,5,6) ;        // 调用有三个参数的 add 方法
11          System.out.println("int_sum = add(3,5,6)的值是："+int_sum);
12          double_sum = add(3.2,6.5);  // 传入的数值为 doule 类型
13          System.out.println("double_sum = add(3.2,6.5)的值是："+double_sum);
14      }
15      public static int add(int x,int y)
16      {
17          return x+y ;
18      }
19      public static int add(int x,int y,int z)
20      {
21          return x+y+z ;
22      }
23      public static double add(double x,double y)
```

```
24          {
25                  return x+y ;
26          }
27    }
```

输出结果：

```
int_sum = add(3,5)的值是：8
int_sum = add(3,5,6)的值是：14
double_sum = add(3.2,6.5)的值是：9.7
```

可以发现上题中的 add 被重载了三次，但每个重载了的方法所能接受参数的个数和类型不同，相信读者现在应该可以明白方法重载的概念了。

4.4.4 将数组传递到方法里

方法不只可以用来传递一般的变量，也可用来传递数组。本节将讲述在 Java 里是如何传递数组以及如何处理方法的返回值是一维数组的问题。

4.4.4.1 传递一维数组

要传递一维数组到访法里，只要指明传入的参数是一个数组即可。TestJava4_12 是传递一维数组到 largest()方法的一个范例，当 largest()接收到此数组时，便会把数组的最大值输出。

范例：TestJava4_12.java

```
01    // 一维数组作为参数来传递，这里的一维数组采用静态方式赋值
02    public class TestJava4_12
03    {
04          public static void main(String args[])
05          {
06                int score[]={7,3,8,19,6,22};          // 声明一个一维数组 score
07                largest(score);                    // 将一维数组 score 传入 largest() 方法中
```

```
08          }
09
10
11          public static void largest(int arr[])
12          {
13                  int tmp=arr[0];
14                  for(int i=0;i<arr.length;i++)
15                          if(tmp<arr[i])
16                                  tmp=arr[i];
17                  System.out.println("最大的数  = "+tmp);
18          }
19  }
```

输出结果：

```
最大的数  = 22
```

TestJava4_12 的第 11~18 行声明 largest()方法，并将一维数组作为该方法的参数。第 13~17 行找出数组的最大值，并将它输出来。注意如果要传递数组到方法里，只要在方法内填上数组的名称即可，如本题的第 7 行所示。

4.4.4.2 传递二维数组

二维数组的传递与一维数组相当类似，只要在方法里声明传入的参数是一个二维数组即可。程序 TestJava4_13 是有关传递二维数组的一个范例，把二维数组 A 传递到 print_mat()方法里，并在 print_mat()方法里把该数组值输出。

范例：TestJava4_13.java

```
01  // 以下程序说明了如何将一个二维数组作为参数传递到方法中
02  public class TestJava4_13
03  {
04          public static void main(String args[])
05          {
```

```
06
07              int A[][]={{51,38,22,12,34},{72,64,19,31}};   // 定义一个二维数组 A
08              print_mat(A);
09          }
10
11          public static void print_mat(int arr[][])   // 接收整数类型的二维数组
12          {
13              for(int i=0;i<arr.length;i++)
14              {
15                  for(int j=0;j<arr[i].length;j++)
16                      System.out.print(arr[i][j]+" ");            // 输出数组值
17                  System.out.print("\n");     // 换行
18              }
19          }
20      }
```

输出结果：

51 38 22 12 34

72 64 19 31

TestJava4_13 的第 11~18 行声明了 print_mat()方法，它可接收二维数组，并利用两个 for 循环把数组的值输出来。注意可以利用.length 取出数组的行数或列数，如程序的第 13 与 15 行所示。

4.4.4.3 返回数组的方法

如果方法返回整数，则必须在声明时在方法的前面加上 int 关键字。相反的如果返回的是一维的整型数组，则必须在方法的前面加上 int[]。若是返回二维的整型数组，则加上 int[][]，以此类推。

TestJava4_14.java 是返回二维数组的一个范例。将一个二维数组传入 add10()方法中，在 add10()方法内将每一个元素加 10 之后返回它，最后在 main()里输出此数组。

范例：TestJava4_14.java

```
01  // 以下的程序说明了方法中返回一个二维数组的实现过程
02  public class TestJava4_14
03  {
04      public static void main(String args[])
05      {
06          int A[][]={{51,38,82,12,34},{72,64,19,31}};  // 定义二维数组
07          int B[][]=new int[2][5];
08          B=add10(A);                  // 调用 add10()，并把返回的值设给数组 B
09          for(int i=0;i<B.length;i++)          // 输出数组的内容
10          {
11              for(int j=0;j<B[i].length;j++)
12                  System.out.print(B[i][j]+" ");
13              System.out.print("\n");
14          }
15      }
16
17      public static int[][] add10(int arr[][])
18      {
19          for(int i=0;i<arr.length;i++)
20              for(int j=0;j<arr[i].length;j++)
21                  arr[i][j]+=10;              // 将数组元素加 10
22          return arr;                         // 返回二维数组
23      }
24  }
```

输出结果：

```
61 48 92 22 44
82 74 29 41
```

虽然 TestJava4_14 的程序代码有点长，但是还是很好理解的。第 17 行赋值 add10() 是可接收二维数组，且返回类型是二维的整型数组。第 21 行是完成了在循环内将数组元素值加 10 的操作，而运算之后的结果再由第 22 行的 return 语句返回。

• 本章摘要：

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，必须经过两个步骤：（1）声明数组、（2）开辟内存给该数组。

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

4、 如果想在声明时就给数组赋初值，只要在数组的声明格式后面加上初值的赋值即可。

5、 Java 允许二维数组中每行的元素个数均不相同。

6、 在二维数组中，若是想取得整个数组的行数，或是某行元素的个数时，也可以利用.length 来获取。

7、 方法的重载：在同一个类中允许同时存在一个以上的同名方法，只要它们的参数个数或类型不同即可。在这种情况下，该方法就叫被重载了，这个过程称为方法的重载。

第 2 部分　Java 面向对象程序设计

- 面向对象概念
- 类与对象
- 类的封装性、继承性、多态性
- Java 异常处理机制
- 包的使用

第 5 章　类的基本形式

到目前为止，前面所学习到的 Java 语法都属于 Java 语言的最基本功能，其中包括了数据类型和程序控制语句、循环语句等。但随着计算机的发展，面向对象的概念也随之孕育而生。类（class）是面向对象程序设计最重要的概念之一，要深入了解 Java 程序语言，一定要了解面向对象程序设计的观念，从本章将开始学习 Java 程序中类的设计！

5.1　面向对象程序设计的基本概念

早期的程序设计经历了“面向问题”、“面向过程”的阶段，随着计算机技术的发展，以及所要解决问题的复杂性的提高，以往的程序设计方法已经不能适应这种发展的需求。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（如图 5-1 所示），并逐渐产生了面向对象的程序设计方法。面向对象的程序设计涉及到对象、封装、类、继承及多态等几个基本概念。

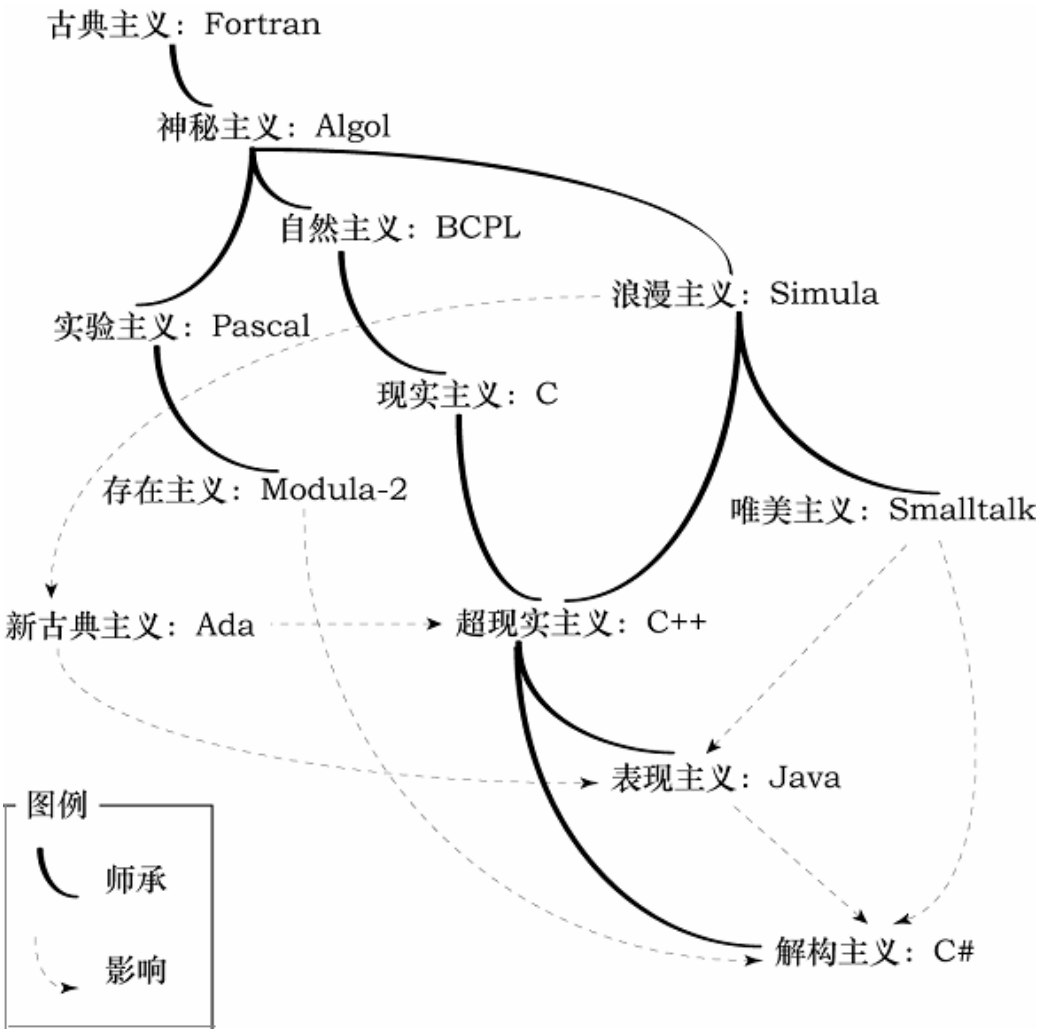


图 5-1　计算机语言的发展过程

5.1.1　对象

何谓面向对象是什么意思呢？

面向对象程序设计是将人们认识世界过程中普遍采用的思维方法应用到程序设计中。对象是现实世界中存在的事物，它们是有形的，如某个人、某种物品；也可以是无形的，如某项计划、某次商业交易。对象是构成现实世界的一个独立单位，人们

对世界的认识，是从分析对象的特征入手的。

对象的特征分为静态特征和动态特征两种。静态的特征指对象的外观、性质、属性等；动态的特征指对象具有的功能、行为等。客观事物是错综复杂的，但人们总是从某一目的出发，运用抽象分析的能力，从众多的特征中抽取最具代表性、最能反映对象本质的若干特征加以详细研究。

人们将对象的静态特征抽象为属性，用数据来描述，在 Java 语言中称之为变量；人们将对象的动态特征抽象为行为，用一组代码来表示，完成对数据的操作，在 Java 语言中称之为方法，。一个对象由一组属性和一组对属性进行操作的方法构成。

5.1.3 类

将具有相同属性及相同行为的一组对象称为类。广义地讲，具有共同性质的事物的集合就称为类。

在面向对象程序设计中，类是一个独立的单位，它有一个类名，其内部包括成员变量，用于描述对象的属性；还包括类的成员方法，用于描述对象的行为。在 Java 程序设计中，类被认为是一种抽象数据类型，这种数据类型，不但包括数据，还包括方法。这大大地扩充了数据类型的概念。

类是一个抽象的概念，要利用类的方式来解决问题，必须用类创建一个实例化的类对象，然后通过类对象去访问类的成员变量，去调用类的成员方法来实现程序的功能。这如同“汽车”本身是一个抽象的概念，只有使用了一辆具体的汽车，才能感受到汽车的功能。

一个类可创建多个类对象，它们具有相同的属性模式，但可以具有不同的属性值。Java 程序为每一个类对象都开辟了内存空间，以便保存各自的属性值。

面向对象的程序设计有三个主要特征，如下：

- 封装性
- 继承性
- 多态性

5.1.2 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，则不允许外界知晓；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了不同级别的“可见性”或访问权限。

5.1.4 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。如已有的汽车的类，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，派生类继承了超类的所有成员，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，称为单继承；若允许继承多个父类，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憾。

5.1.5 类的多态性

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成员覆盖两种形式的多态。

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。

成员覆盖：子类与父类允许具有相同的变量名称，但数据类型不同，允许具有相同的方法名称，但完成的功能不同。

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

面向对象的编程思想力图使在计算机语言中对事物的描述与现实世界中该事物的本来面目尽可能地一致，类（class）和对象（object）就是面向对象方法的核心概念。类是对某一类事物的描述，是抽象的、概念上的定义；对象是实际存在的该类事物的个体，因而也称实例（Instance）。如图 5-2 就是一个说明类与对象的典型范例：

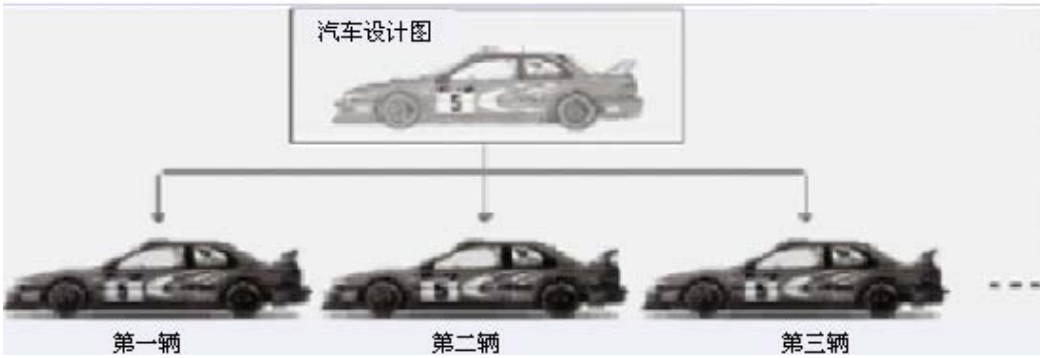


图 5-2　类与对象的实例化说明

上图中，汽车设计图就是“类”，由这个图纸设计出来的若干的汽车就是按照该类产生的“对象”。可见，类描述了对象的属性和对象的行为，类是对象的模板。对象是类的实例，是一个实实在在的个体，一个类可以对应多个对象。可见，如果将对象比作汽车，那么类就是汽车的设计图纸，所以面向对象程序设计的重点是类的设计，而不是对象的设计。

同一个类按同种方法产生出来的多个对象，其开始的状态都是一样的，但是修改其中一个对象的时候，其他的对象是不会受到影响的，比如修改第一辆汽车的时候，其他的汽车是不会受到影响的

5.2.2　类的声明

在使用类之前，必须先定义它，然后才可利用所定义的类来声明变量，并创建对象。类定义的语法如下：

【 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）

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

下面给读者举一个 Person 类的例子，来让读者清楚认识类的组成。

范例：Person.java

```
01  class Person
02  {
03      String name ;
04      int age ;
05      void talk()
06      {
07          System.out.println("我是："+name+"，今年："+age+"岁");
08      }
09  }
```

程序说明：

1、 程序首先用 class 声明了一个名为 Person 的类，这里 Person 是类的名称。

2、 在第 3、4 行，先声明了两个属性 name 和 age，name 为 String（字符串类型）型、age 为 int（整型）型。

3、 在第 5~8 行，声明了一个 talk()方法，此方法用于向屏幕打印信息。

为了更好的说明类的关系，请参见图 5-3。

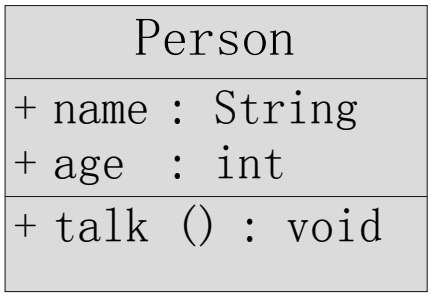


图 5-3　Person 类图

！小提示:

读者可以发现在本例中，声明类 Person 时，类名中单词的首字母是大写的，这是规定的一种符合标准的写法，在本书以后的范例中都将采用这种写法。

5.2.3　创建新的对象

在上面的范例中，已经创建好了一个 Person 的类，相信类的基本形式读者应该已经很清楚了，但是在实际中单单有类是不够的，类提供的只是一个摸板，必须依照它创建出对象之后才可以使用。下面定义了由类产生对象的基本形式：

【 格式 5-2 对象的产生】

```
类名 对象名 =  new 类名();
```

了解了上述的概念之后，便可动手编写程序了。创建属于某类的对象，需要通过下面两个步骤来实现：

1、 声明指向"由类所创建的对象"的变量

2、 利用 new 创建新的对象，并指派给先前所创建的变量。

举例来说，如果要创建 Person 类的对象，可用下列的语句来实现：

```
Person p  ;                  // 先声明一个 Person 类的对象 p
p = new Person() ;           // 用 new 关键字实例化 Person 的对象 p
```

当然也可以用下面这种形式来声明变量：

Person p = new Person() ;　// 声明 Person 对象 p 并直接实例化此对象

！小提示：

对象只有在实例化之后才能被使用，而实例化对象的关键字就是 new。

关于对象实例化的过程，请参见图 5-4：

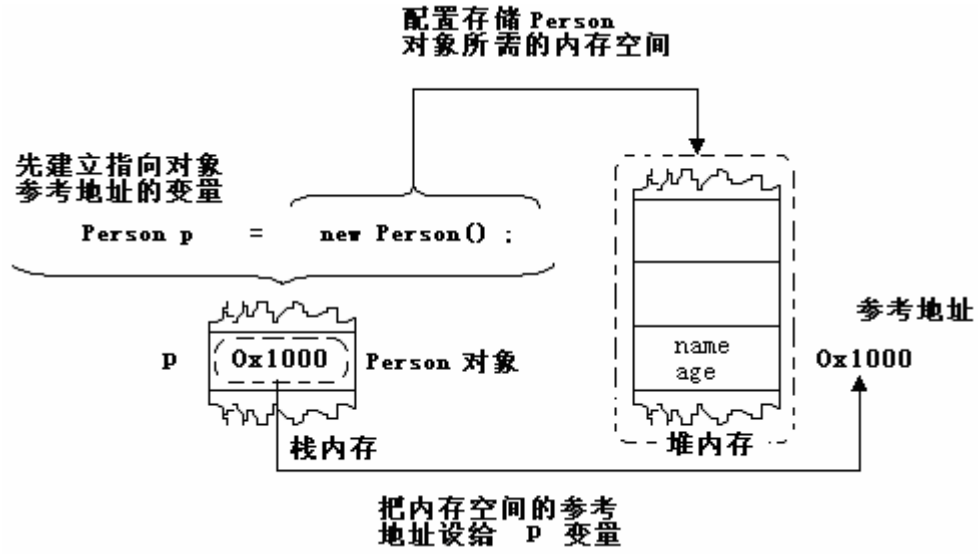


图 5-4　Person 类对象的实例化过程

由图中可以看出，当语句执行到 Person p 的时候，只是在栈内存中声明了一个 Person 的对象 p，但是这个时候 p 并没有在堆内存中开辟空间，所以这个时候的 p 是一个未实例化的对象，用 new 关键字实际上就是开辟堆内存，把堆内存的引用赋给了 p，这个时候的 p 才称为一实例化对象。

如果要访问对象里的某个成员变量或方法时，可以通过下面语法来实现：

【 格式 5-3 访问对象中某个变量或方法】

访问属性：对象名称. 属性名 **访问方法：**对象名称. 方法名()

例如：如果想访问 Person 类中的 name 和 age 属性，可以用如下方法来访问：

```
p.name ;        // 访问 Person 类中的 name 属性
p.age ;         // 访问 Person 类中的 age 属性
```

因此：若想将 Person 类的对象 p 中的属性 name 赋值为"张三"，年龄赋值为 25，则可以采用下面的写法：

```
p.name = "张三" ;
p.age = 25 ;
```

如果想调用 Person 中的 talk()方法，可以采用下面这种写法：

```
p.talk() ;      // 调用 Person 类中的 talk()方法
```

请读者看下面的完整的程序：

范例：Person.java

```
01  class Person
02  {
03      String name ;
04      int age ;
05      void talk()
06      {
07          System.out.println("我是："+name+"，今年："+age+"岁");
08      }
09  }
```

范例：TestPersonDemo.java

```
01  // 下面这个范例说明了使用 Person 类的对象调用类中的属性与方法的过程
02  class TestPersonDemo
03  {
04      public static void main(String[] args)
05      {
06          Person p = new Person() ;
07          p.name = "张三" ;
08          p.age = 25 ;
09          p.talk();
10      }
11  }
```

输出结果：

我是：张三，今年：25 岁

程序说明：

1、 第 6 行声明了一个 Person 类的实例对象 p，并直接实例化此对象

2、 第 7、8 行给 p 对象中的属性赋值

3、 第 9 行调用 talk()方法，在屏幕上输出信息

可以参照上述程序代码与图 5-5 的内容，即可了解到 Java 是如何对对象成员进行访问操作的。

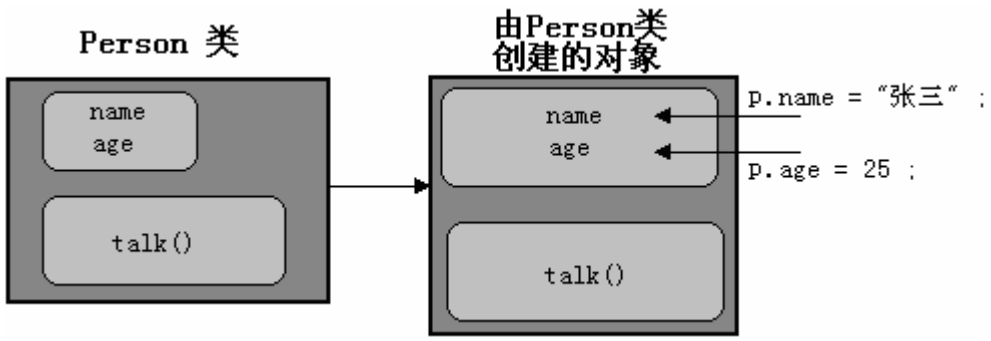


图 5-5　对 Person 对象 p 的访问操作过程

5.2.4 创建多个新对象

在上面的 TestPerson.java 程序中，只建立了一个 Person 的对象 p，如果需要创建多个对象的话，则可以依照格式 5-2 产生多个对象，如下范例所示：

范例：TestPersonDemo1.java

```
01  class Person
02  {
03      String name ;
04      int age ;
05      void talk()
06      {
07          System.out.println("我是："+name+"，今年："+age+"岁");
08      }
09  }
10
11  public class TestPersonDemo1
12  {
13      public static void main(String[] args)
14      {
15          // 声明并实例化一 Person 对象 p1
16          Person p1 = new Person() ;
17          // 声明并实例化一 Person 对象 p2
18          Person p2 = new Person() ;
19
20          // 给 p1 的属性赋值
21          p1.name = "张三" ;
22          p1.age = 25 ;
23          // 给 p2 的属性赋值
24          p2.name = "李四" ;
25          p2.age = 30 ;
26
27          // 分别用 p1、p2 调用 talk()方法
28          p1.talk() ;
29          p2.talk() ;
```

```
30          }
31    }
```

输出结果：

```
我是：张三，今年：25 岁
我是：李四，今年：30 岁
```

程序说明：

1、 1~9 行声明了一个新的类 Person，类中有 name、age 两个属性，还有一个 talk() 方法用于输出信息。

2、 15~18 声明了 Person 的两个实例对象 p1、p2。

3、 21、22 行给 p1 对象的属性赋值。

4、 24、25 行给 p2 对象的属性赋值。

5、 28、29 行分别用 p1、p2 调用 Person 类中的 talk()方法，用于在屏幕上打印信息。

5、 在程序中声明了两个对象 p1 和 p2，之后为 p1 与 p2 分别赋值，可以发现 p1 与 p2 赋的值互不影响，此关系可由图 5-6 表示出来。

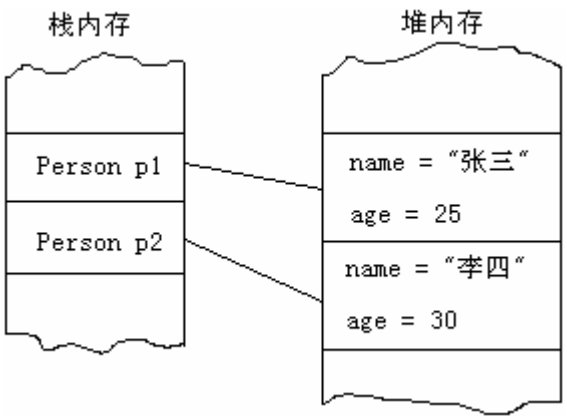


图 5-6　Person 中 p1 与 p2 的内存分配图

可以发现 p1 与 p2 各自占有一块内存空间，p1、p2 中各有自己的属性值，所以 p1、p2 不会互相影响。

5.3 类的封装性

本书开始讨论类的封装性，在本章的一开始已经介绍过面向对象的三大特性了，那么现在就来看一看面向对象第一大特性 —— 封装性。那什么是封装性呢？读者可以先看看下面的程序，看看会产生什么问题：

范例：TestPersonDemo2.java

```
01  class Person
02  {
03      String name ;
04      int age ;
05      void talk()
06      {
07          System.out.println("我是："+name+"，今年："+age+"岁");
08      }
09  }
10
11  public class TestPersonDemo2
12  {
13      public static void main(String[] args)
14      {
15          // 声明并实例化一 Person 对象 p
16          Person p = new Person() ;
17          // 给 p 中的属性赋值
18          p.name = "张三" ;
19          // 在这里将对象 p 的年龄属性赋值为-25 岁
20          p.age = -25 ;
21          // 调用 Person 类中的 talk()方法
22          p.talk() ;
23      }
24  }
```

输出结果：

```
我是：张三，今年：-25 岁
```

程序说明：

1、 1~9 行声明了一个新的类 Person，类中有 name、age 两个属性，还有一个 talk()方法用于输出信息。

2、 程序第 16 行，声明并实例化一 Person 的对象 p。

3、 18~22 行，分别为 p 对象中的属性赋值，并调用 talk()方法。

由上面的程序可以发现，在程序的第 20 行，将年龄（age）赋值为-25 岁，这明显是一个不合法的数据，最终程序在调用 talk()方法的时候才会打印出了这种错误的信息。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出，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情况的发生，在一般的开发中往往要将类中的属性封装（private），对范例 TestPersonDemo2.java 做了相应的修改后形成下面的程序 TestPersonDemo3-1.java，如下所示：

范例：TestPersonDemo3-1.java

```
01  class Person
02  {
03      private String name ;
04      private int age ;
05      void talk()
06      {
07          System.out.println("我是："+name+"，今年："+age+"岁");
08      }
09  }
10
11  public class TestPersonDemo3-1
12  {
13      public static void main(String[] args)
14      {
15          // 声明并实例化一 Person 对象 p
```

```
16              Person p = new Person() ;
17              // 给 p 中的属性赋值
18              p.name = "张三" ;
19              // 在这里将 p 对象中的年龄赋值为-25 岁
20              p.age = -25 ;
21              // 调用 Person 类中的 talk()方法
22              p.talk() ;
23          }
24      }
```

编译结果：

```
TestPersonDemo3.java:18: name has private access in Person
                        p.name = "张三" ;
                         ^
TestPersonDemo3.java:20: age has private access in Person
                        p.age = -25 ;
                         ^
2 errors
```

程序说明：

1、 1~9 行声明了一个新的类 Person，类中有 name、age 两个属性，还有一个 talk()方法用于输出信息，与前面不同的是这里的属性在声明时前面都加上了 private 关键字。

2、 程序第 16 行，声明并实例化一 Person 类的对象 p。

3、 18~22 行，分别为 p 对象中的属性赋值，并调用 talk()方法。

可以发现本程序与上面的范例除了在声明属性上有些区别之外，并没有其它的区别，而就是这一个小小的关键字，却可以发现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，但是如果非要给对象赋值的话，而这一矛盾该如何解决呢？程序设计人员一般在类的设计时，会对属性增加一些方法，如：setXxx()、getXxx()这样的公有方法来解决这一矛盾。请看下面的范例：

范例：TestPersonDemo3-2.java

```
01  class Person
02  {
03      private String name ;
04      private int age ;
05      void talk()
06      {
07          System.out.println("我是："+name+"，今年："+age+"岁");
08      }

09      public void setName(String str)
10      {
11          name = str ;
12      }
13      public void setAge(int a)
14      {
15          if(a>0)
16              age = a ;
17      }
18      public String getName()
19      {
20          return name ;
21      }
22      public int getAge()
23      {
24          return age ;
25      }
26  }
27
28  public class TestPersonDemo3-2
29  {
30      public static void main(String[] args)
31      {
32          // 声明并实例化一 Person 对象 p
33          Person p = new Person() ;
34          // 给 p 中的属性赋值
35          p.setName("张三") ;
```

```
36              // 在这里将 p 对象中的年龄赋值为-25 岁
37              p.setAge(-25) ;
38              // 调用 Person 类中的 talk()方法
39              p.talk() ;
40          }
41      }
```

输出结果：

我是：张三，今年：0 岁

程序说明：

1、 在程序 9~25 行，加入了一些 setXxx()、getXxx()方法，主要用来设置和取得类中的私有属性。

2、 在程序 35 行调用了 Person 类中的 setName()方法，并赋值为“张三”，在程序 37 行调用了 setAge()方法，同时传进了一个-25 的不合理年龄。

3、 程序 13~17 行设置年龄的时候在程序中加了些判断语句，如果传入的数值大于 0，则将值赋给 age 属性。

可以发现在本程序中，传进了一个-25 的不合理的数值，这样在设置 Person 中属性的时候因为不满足条件而不能被设置成功，所以 age 的值依然为自己的默认值 —— 0。这样在输出的时候可以发现，那些错误的数据并没有被赋到属性上去，而只输出了默认值。

由此可以发现，用 private 可以将属性封装起来，当然 private 也可以封装方法，封装的形式请参考格式 5-4。

【 格式 5-4 封装类中的属性或方法】

封装属性：private 属性类型 属性名 **封装方法：**private 方法返回类型 方法名称（参数）

注意：

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，读者可以先暂时简单的理解为，在类的外部不能用对象去调用 private 声明的属性或方法。

下面这道程序修改自上面的程序（**TestPersonDemo3-2.java**），在这里，将 talk() 方法封装了起来。

范例：TestPersonDemo4.java

```
01  class Person
02  {
03      private String name ;
04      private int age ;
05      private void talk()
06      {
07          System.out.println("我是："+name+"，今年："+age+"岁");
08      }
09      public void setName(String str)
10      {
11          name = str ;
12      }
13      public void setAge(int a)
14      {
15          if(a>0)
16              age = a ;
17      }
18      public String getName()
19      {
20          return name ;
21      }
22      public int getAge()
23      {
24          return age ;
25      }
26  }
```

```
27
28  public class TestPersonDemo4
29  {
30      public static void main(String[] args)
31      {
32          // 声明并实例化一 Person 对象 p
33          Person p = new Person() ;
34          // 给 p 中的属性赋值
35          p.setName("张三") ;
36          // 在这里将 p 对象中的年龄赋值为-25 岁
37          p.setAge(-25) ;
38          // 调用 Person 类中的 talk()方法
39          p.talk() ;
40      }
41  }
```

编译结果：

```
TestPersonDemo4.java:39: talk() has private access in Person
                    p.talk() ;
                     ^
1 error
```

程序说明：

1、 在程序 9~25 行，加入了一些 setXxx()、getXxx()方法，主要用来设置和取得类中的私有属性。

2、 在程序 35 行调用了类 Person 中的 setName()方法，并赋值为"张三"，在程序 37 行调用了 setAge()方法，同时传进了一个不合理的年龄-25。

3、 程序 13~17 行设置年龄的时候在程序中加了些判断语句，如果传入的数值大于 0，则将值赋给 age 属性。

4、 在程序第 5 行，将 talk()方法用 private 来声明。

可以发现 private 也是同样可以用来声明方法的，这样这个方法就只能在类的内部被访问了。

！小提示：

读者可能会问，到底什么时候需要封装，什么时候不用封装。在这里可以告诉读者，关于封装与否并没有一个明确的规定，不过从程序设计角度来说，一般说来设计较好的程序的类中的属性都是需要封装的。此时，要设置或取得属性值，则只能用 setXxx()、getXxx()方法，这是一个明确且标准的规定。

5.4 在类内部调用方法

通过上面的几个范例，读者应该可以清楚，在一个 java 程序中是可以通过对象去调用类中的方法的，当然类的内部也能互相调用各自的方法，比如下面的程序，在下面的程序中，修改了以前的程序代码，新增加了一个公有的 say()方法，并用这个方法去调用私有的 talk()方法。

范例：TestPersonDemo5.java

```
01  class Person
02  {
03      private String name ;
04      private int age ;
05      private void talk()
06      {
07          System.out.println("我是："+name+"，今年："+age+"岁");
08      }
09      public void say()
10      {
11          talk();
12      }
13      public void setName(String str)
14      {
15          name = str ;
16      }
17      public void setAge(int a)
18      {
```

```
19              if(a>0)
20                  age = a ;
21          }
22          public String getName()
23          {
24              return name ;
25          }
26          public int getAge()
27          {
28              return age ;
29          }
30      }
31
32      public class TestPersonDemo5
33      {
34          public static void main(String[] args)
35          {
36              // 声明并实例化一 Person 对象 p
37              Person p = new Person() ;
38              // 给 p 中的属性赋值
39              p.setName("张三") ;
40              // 在这里将 p 对象中的年龄属性赋值为-25 岁
41              p.setAge(30) ;
42              // 调用 Person 类中的 say()方法
43              p.say() ;
44          }
45      }
```

输出结果：

我是：张三，今年：30 岁

程序说明：

1、 程序 9~12 行声明一公有方法 say()，此方法用于调用类内部的私有方法 talk()。

2、 在程序第 43 行调用 Person 类中的 say()方法，其实也就是调用了 Person 类中的 talk()方法。

注意：

这个时候 say()方法调用 talk()方法，如果非要强调对象本身的话，也可以写成如下形式:

```
this.talk() ;
```

读者也许会觉得这样写有些多余，当然 this 的使用方法很多，在以后的章节中会有更加完整的介绍，读者可自行修改上面的程序试验一下，看看结果是不是与原来的相同。

5.5 引用数据类型的传递

在本书的开始就已经提到过，java 中使用引用来取代 C++中的指针，那么什么是引用？java 又是怎样通过引用来取代 C++中指针的呢？请读者先看一下下面程序的代码。

范例：TestRefDemo1.java

```
01  class Person
02  {
03      String name ;
04      int age ;
05  }
06  public class TestRefDemo1
07  {
08      public static void main(String[] args)
09      {
10          // 声明一对象 p1，此对象的值为 null，表示未实例化
11          Person p1 = null ;
12          // 声明一对象 p2，此对象的值为 null，表示未实例
13          Person p2 = null ;
14          // 实例化 p1 对象
15          p1 = new Person() ;
16          // 为 p1 对象中的属性赋值
```

```
17              p1.name = "张三" ;
18              p1.age = 25 ;
19              // 将 p1 的引用赋给 p2
20              p2 = p1 ;
21              // 输出 p2 对象中的属性
22              System.out.println("姓名："+p2.name);
23              System.out.println("年龄："+p2.age);
24              p1 = null ;
25          }
26  }
```

输出结果：

姓名：张三

年龄：25

程序说明：

1、 1~5 行声明一 Person 类，有 name 与 age 两个属性

2、 程序 11、13 行，分别声明两个 Person 的对象 p1 和 p2，但这两个对象在声明时都同时赋值为 null，表示此对象未实例化。

3、 程序第 15 行为对对象 p1 进行实例化。

4、 17、18 行分别为 p1 对象中的属性赋值。

5、 20 行，将 p1 的引用赋给 p2，此时相当于 p1 与 p2 都同时指向同一块堆内存。

6、 第 22、23 行分别调用 p2.name 和 p2.age 输出 p2 对象中的属性。

7、 第 24 行把 p1 对象赋值为 null，表示此对象不再引用任何内存空间。

8、 程序执行到第 24 行时，实际上 p1 断开了对其之前实例化对象的引用，而 p2 则继续指向 p1 原先的引用。

由程序中可以发现，在程序中并为用 new 关键字为对象 p2 实例化，而到最后依然可以用 p2.name 与 p2.age 方式输出属性的内容，而且内容与 p1 对象中的内容相似，也就是说在这道程序之中 p2 是通过 p1 对象实例化的，或者说 p1 将其自身的引用传递给了 p2。读者可以参考图 5-7：

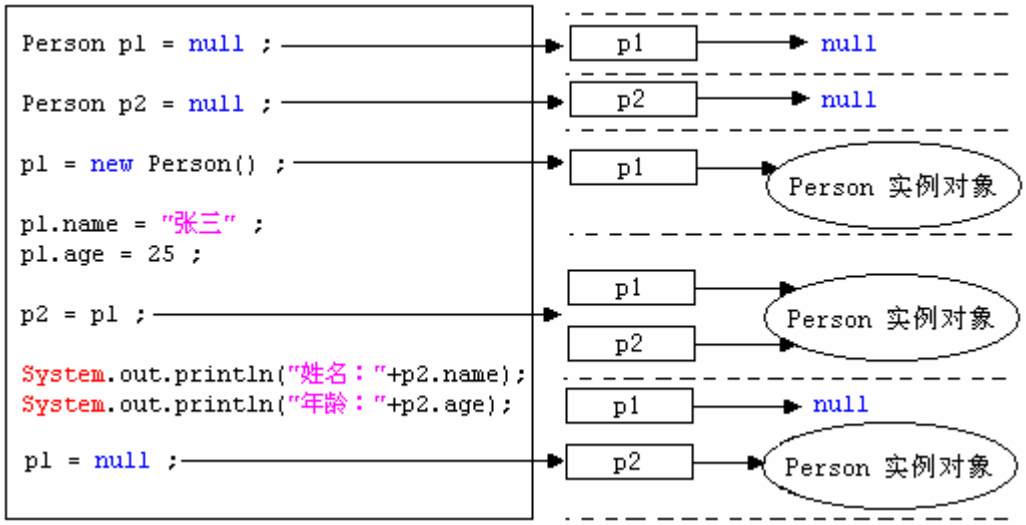


图 5-7　引用数据类型的传递

注意：

如果在程序最后又加了一段代码，令 p2=null，则之前由 p1 创建的实例化对象不再有任何对象使用它，则此对象称为垃圾对象，如图 5-8 所示：

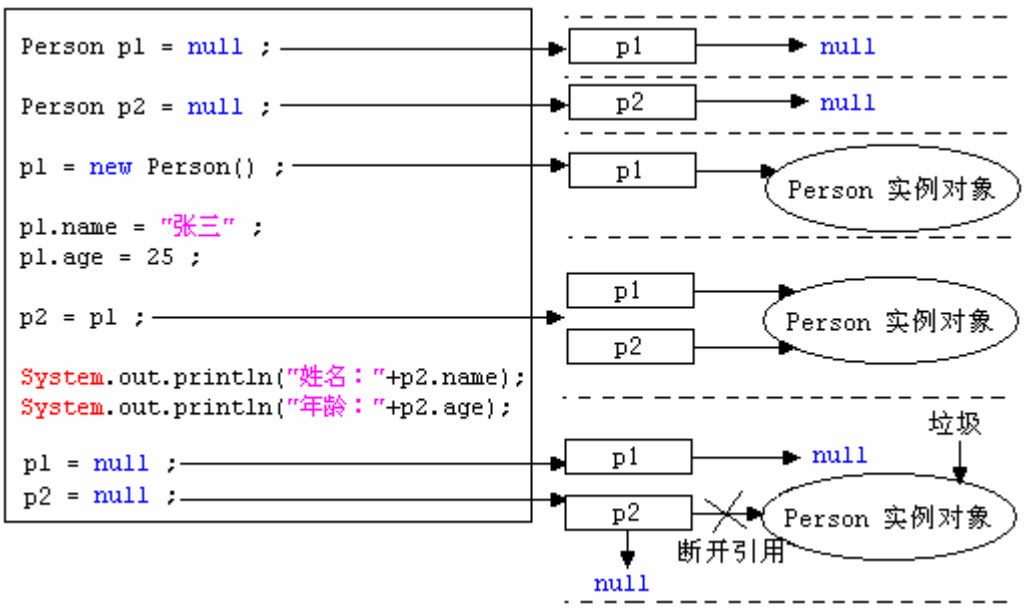


图 5-8　垃圾对象的产生

所谓垃圾对象，就是指程序中不再使用的对象引用，关于垃圾收集的概念，在本章的最后会有介绍。

范例：TestRefDemo2.java

```
01  class Change
02  {
03      int x = 0 ;
04  }
05
06  public class TestRefDemo2
07  {
08      public static void main(String[] args)
09      {
10          Change c = new Change() ;
11          c.x = 20 ;
12          fun(c) ;
13          System.out.println("x = "+c.x);
14      }
15      public static void fun(Change c1)
16      {
17          c1.x = 25 ;
18      }
19  }
```

输出结果：

```
x = 25
```

程序说明：

1、 第 1~4 行，声明一名为 Change 的类，里面有一个属性 x。

2、 程序第 10 行实例化了一个 Change 类的对象 c。

3、 程序第 11 行将对象 c 中的 x 属性赋值为 20。

4、 程序第 12 行调用 fun()方法，将对象 c 传递到方法之中。

5、 15~18 行声明这个 fun 方法，接收参数类型为 Change 类型。

6、 第 17 行将对象 c1 中的 x 属性赋值为 25。

可以发现程序最后的输出结果为“x = 25”，而程序只有在 fun()方法中，才将 x 的值赋为 25，为什么方法完成之后值还依然被保留下来了呢？读者可以发现在程序第 15 行，fun()方法接收的参数是 Change c1，也就是说 fun()方法接收的是一个对象的引用。所以在 fun 方法中所做的操作，是会影响原先的参数，可以参考图 5-9，来了解整个过程：

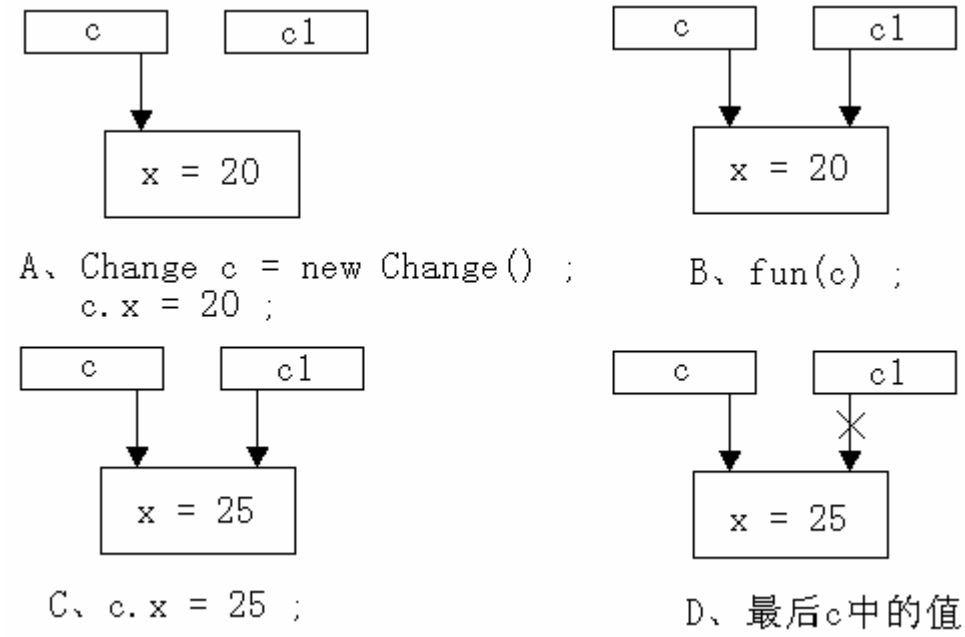


图 5-9　操作过程

从图中可以发现，最开始声明的对象 c 将其内部的 x 赋值为 20（图 A），之后调用 fun()方法，将对象赋值给 c1，c1 再继续修改 x 的值，此时 c1 与 c 同时指向同一内存空间，所以 c1 操作了 x 也就相当于 c 操作了 x，所以 fun()方法执行完毕之后，x 的值为 25。

5.6　匿名对象

“匿名对象”，顾名思义，就是没有明确的声明的对象。读者也可以简单的理解

为只使用一次的对象，即没有任何一个具体的对象名称引用它。请看下面的范例：

范例：TestNoName.java

```
01  class Person
02  {
03      private String name = "张三" ;
04      private int age = 25 ;
05      public String talk()
06      {
07          return "我是："+name+"，今年："+age+"岁" ;
08      }
09  }
10
11  public class TestNoName
12  {
13      public static void main(String[] args)
14      {
15          System.out.println(new Person().talk());
16      }
17  }
```

输出结果：

```
我是：张三，今年：25 岁
```

程序说明：

1、 程序 1~9 行声明了一 Person 类，里面有 name 与 age 两个私有属性，并分别赋了初值。

2、 在程序第 15 行，声明了一 Person 匿名对象，调用 Person 类中的 talk()方法。

读者由程序中可以发现用 new Person()声明的对象并没有赋给任何一个 Person 类对象的引用，所以此对象只使用了一次，之后就会被 Java 的垃圾收集器回收。

5.7 构造方法

在 Java 程序里，构造方法所完成的主要工作是帮助新创建的对象赋初值。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，其语法如下所示：

【 格式 5-5 构造方法的定义方式】

```
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
        }
}
```

在使用构造方法的时候请注意以下几点：

1、 它具有与类名相同的名称

2、 它没有返回值

如上所述，构造方法除了没有返回值，且名称必须与类的名称相同之外，它的调用时机也与一般的方法不同。一般的方法是在需要时才调用，而构造方法则是在创建对象时，便自动调用，并执行构造方法的内容。因此，构造方法无需在程序中直接调用，而是在对象产生时自动执行。

基于上述构造方法的特性，可利用它来对对象的数据成员做初始化的赋值。所谓初始化就是为对象的赋初值。

下面的程序 TestConstruct.java 说明了构造方法的使用。

范例：TestConstruct.java

```
01  class Person
02  {
03      public Person()          // Person 类的构造方法
```

```
04        {
05            System.out.println("public Person()") ;
06        }
07    }
08
09    public class TestConstruct
10    {
11        public static void main(String[] args)
12        {
13            Person p = new Person() ;
14        }
15    }
```

输出结果：

public Person()

程序说明：

1、 程序 1~7 行声明了一 Person 类，此类中只有一 Person 的构造方法。

2、 程序 3~6 行声明了一 Person 类的构造方法，此方法只含有一个输出语句。

3、 程序第 13 行实例化一个 Person 类的对象 p，此时会自动调用 Person 中的无参构造方法，即在屏幕上打印信息。

从此程序中读者不难发现，在类中声明的构造方法，会在实例化对象时自动调用。

读者可能会问，在之前的程序中用同样的方法来产生对象，但是在类中并没有声明任何构造方法，而程序不也一样正常运行了吗？

实际上，读者在执行 javac 编译 java 程序的时候，如果在程序中没有明确声明一构造方法的话，系统会自动为类加入一个无参的且什么都不做的构造方法。类似于下面代码：

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

所以，之前所使用的程序虽然没有明确的声明构造方法，也是可以正常运行的。

5.7.1 构造方法的重载

在 Java 里，不仅普通方法可以重载，构造方法也可以重载。只要构造方法的参数个数不同，或是类型不同，便可定义多个名称相同的构造方法。这种做法在 java 中是常见的，请看下面的程序。

范例：TestConstruct1.java

```
01  class Person
02  {
03      private String name ;
04      private int age ;
05      public Person(String n,int a)
06      {
07          name = n ;
08          age = a ;
09          System.out.println("public Person(String n,int a)") ;
10      }
11      public String talk()
12      {
13          return "我是："+name+"，今年："+age+"岁" ;
14      }
15  }
16
17  public class TestConstruct1
18  {
19      public static void main(String[] args)
20      {
21          Person p = new Person("张三",25) ;
22          System.out.println(p.talk()) ;
23      }
24  }
```

输出结果：

public Person(String n,int a)

我是：张三，今年：25 岁

程序说明：

1、 程序 1~15 行声明一名为 Person 的类，里面有 name 与 age 两个私有属性，和一个 talk()方法。

2、 程序第 5~10 行，在 Person 类中声明一各含有两个参数的构造方法，此构造方法用于将传入的值赋给 Person 类的属性。

3、 程序第 21 行调用 Person 类中有的含有两个参数的构造方法（**new Person("张三",25)** ），同时将姓名和年龄传到类里，分别为各个属性赋值。

4、 程序第 22 行调用 Person 类中的 talk()方法，打印信息。

从本程序可以发现，构造方法的基本作用就是为类中的属性初始化的，在程序产生类的实例对象时，将需要的参数由构造方法传入，之后再由构造方法为其内部的属性进行初始化。这是在一般开发中经常使用的技巧，但是这里有一个问题是读者应该注意的，请看下面的程序：

范例：TestConstruct2.java

```
01  class Person
02  {
03      private String name ;
04      private int age ;
05      public Person(String n,int a)
06      {
07          name = n ;
08          age = a ;
09          System.out.println("public Person(String n,int a)") ;
10      }
11      public String talk()
12      {
13          return "我是："+name+"，今年："+age+"岁" ;
14      }
```

```
15  }
16
17  public class TestConstruct2
18  {
19      public static void main(String[] args)
20      {
21          Person p = new Person() ;
22          System.out.println(p.talk()) ;
23      }
24  }
```

编译结果：

```
TestConstruct2.java:21: Person(java.lang.String,int) in Person cannot be applied to ()
                Person p = new Person() ;
                           ^
1 error
```

可以发现，在编译程序第 21 行（Person p = new Person()）时发生了错误，这个错误说找不到 Person 类的无参构造方法。记得在前面曾经提过，如果程序中没有声明构造方法，程序就会自动声明一个无参的且什么都不做的构造方法，可是现在却发生了找不到无参构造方法的问题，这是为什么呢？读者可以发现第 5~10 行已经声明了一个含有两个参数的构造方法。在 java 程序中只要明确的声明了构造方法，则默认的构造方法将不会被自动生成。而要解决这一问题，只需要简单的修改一下 Person 类就可以了，可以在 Person 类中明确地声明一无参的且什么都不做的构造方法。

范例：TestConstruct2.java

```
01  class Person
02  {
03      private String name ;
04      private int age ;
05      public Person()
06      {}
07      public Person(String n,int a)
08      {
09          name = n ;
```

```
10              age = a ;
11              System.out.println("public Person(String n,int a)") ;
12          }
13          public String talk()
14          {
15              return "我是："+name+"，今年："+age+"岁" ;
16          }
17      }
18
19      public class TestConstruct2
20      {
21          public static void main(String[] args)
22          {
23              Person p = new Person() ;
24              System.out.println(p.talk()) ;
25          }
26      }
```

可以看见，在程序的第 5、6 行声明了一无参的且什么都不做的构造方法，此时再编译程序的话，就可以正常编译而不会出现错误了。

5.8 对象的比较

有两种方式可用于对象间的比较，它们是“= =”运算符与 equals()方法，“= =”操作符用于比较两个对象的内存地址值是否相等，equals()方法用于比较两个对象的内容是否一致。

范例：TestEquals.java

```
01  public class TestEquals
02  {
03      public static void main(String[] args)
04      {
05          String str1 = new String("java") ;
06          String str2 = new String("java") ;
07          String str3 = str2 ;
```

```
08              if(str1==str2)
09              {
10                  System.out.println("str1 == str2");
11              }
12              else
13              {
14                  System.out.println("str1 != str2") ;
15              }
16              if(str2==str3)
17              {
18                  System.out.println("str2 == str3");
19              }
20              else
21              {
22                  System.out.println("str2 != str3") ;
23              }
24          }
25  }
```

输出结果：

str1 != str2

str2 == str3

由程序的输出结果可以发现，str1 不等于 str2，有些读者可能会问，str1 与 str2 的内容完全一样，为什么会不等于呢？读者可以发现在程序的第 5 和第 6 行分别实例化了 String 类的两个对象，此时，这两个对象指向不同的内存空间，所以它们的内存地址是不一样的。这个时候程序中是用的“==”比较，比较的是内存地址值，所以输出 str1!=str2。程序第 7 行，将 str2 的引用赋给 str3，这个时候就相当于 str3 也指向了 str2 的引用，此时，这两个对象指向的是同一内存地址，所以比较值的结果是 str2==str3。

读者可能会问，那该如何去比较里面的内容呢？这就需要采用另外一种方式——“equals”，请看下面的程序，下面的程序 TestEquals1.java 修改自程序 TestEquals.java，如下所示：

范例：TestEquals1.java

```
01  public class TestEquals1
02  {
03      public static void main(String[] args)
04      {
05          String str1 = new String("java") ;
06          String str2 = new String("java") ;
07          String str3 = str2 ;
08          if(str1.equals(str2))
09          {
10              System.out.println("str1 equals str2");
11          }
12          else
13          {
14              System.out.println("str1 not equals str2") ;
15          }
16          if(str2.equals(str3))
17          {
18              System.out.println("str2 equals str3");
19          }
20          else
21          {
22              System.out.println("str2 note equals str3") ;
23          }
24      }
25  }
```

输出结果：

```
str1 equals str2
str2 equals str3
```

这个时候可以发现，在程序中将比较的方式换成了 equals，而且调用 equals()方法的是 String 类的对象，所以可以知道 equals 是 String 类中的方法。在这里读者一定要记住：**“==”是比较内存地址值的，“equals”是比较内容的。**

！小提示：

有些读者可能会问，下面两种 String 对象的声明方式到底有什么不同？

```
String str1 = new String("java");
String str2 = "java";
```

下面先来看一个范例：

```
01  public class StringDemo
02  {
03      public static void main(String[] args)
04      {
05          String str1 = "java" ;
06          String str2 = new String("java") ;
07          String str3 = "java" ;
08          System.out.println("str1 == str2 ? --- > "+(str1==str2)) ;
09          System.out.println("str1 == str3 ? --- > "+(str1==str3)) ;
10          System.out.println("str3 == str2 ? --- > "+(str3==str2)) ;
11      }
12  }
```

输出结果：

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

由程序输出结果可以发现，str1 与 str3 相等，这是为什么呢？还记得上面刚提到过“==”是用来比较内存地址值的。现在 str1 与 str3 相等，则证明 str1 与 str3 是指向同一个内存空间的。可以用图 5-10 来说明：

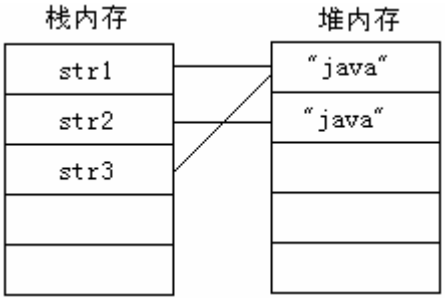


图 5-10　String 对象的声明与使用

由图中可以看出“java”这个字符串在内存中开辟的一个空间，而 str1 与 str3 又同时指向同一内存空间，所以即使 str1 与 str3 虽然是分两次声明的，但最终却都指向了同一内存空间。而 str2 是用 new 关键字来开辟的空间，所以单独占有自己的一个内存空间。

另外，还要提醒读者注意的是，String 对象的内容一旦声明则不能轻易改变。如果想改变一个 String 对象的值，则第一步要做的是先将原有的 String 引用断开，之后再开辟新的内存空间，而且如果用 new 关键字开辟 String 对象的内存空间的话，则实际上就开辟了两个内存空间，如图 5-11 所示:

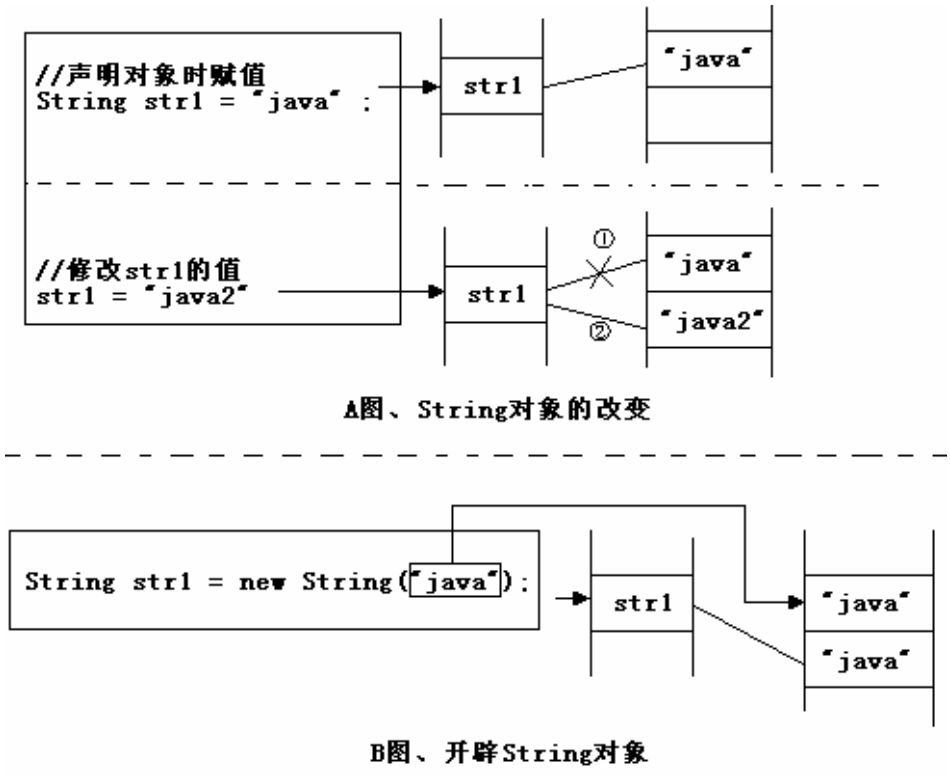


图 5-11　String 对象内容的改变

由图 5-11（A）中不难发现，String 对象一旦声明则不能轻易改变，如果要改变则需要先断开原有的对象引用，再开辟新的对象，之后再指向新的对象空间。

用图 5-11(B)的方法也可以实现改变 String 对象的声明的操作，可以发现，用 new String(“java”)方式实例化 String 对象时，实际上是开辟了两个内存空间，所以一般在开发上都采用直接赋值的方式，即：String str1 = "java" 。

5.9　this 关键字的使用

在整个 java 的面向对象程序设计中，this 是一个比较难理解的关键字，读者应该还记得在前面 5-4 节调用类内部方法时，曾提到过，this 强调对象本身，那么什么是对象本身呢？其实在这里读者只要遵循一个原则就可以，就是 this 表示当前对象，而所谓

的当前对象就是指调用类中方法或属性的那个对象。先来看一下下面的程序片段：

范例：Person.java

```
01  class Person
02  {
03      private String name ;
04      private int age ;
05      public Person(String name,int age)
06      {
07          name = name ;
08          age = age ;
09      }
10  }
```

看上面的程序可以发现会有些迷惑，程序的本意是通过构造方法为 name 和 age 进行初始化的，但是在构造方法中声明的两个参数的名称也同样是 name 和 age，这就造成了一种不清楚的关系，到底第 7 行的形参 name 是赋给了类中的属性 name，还是类中的属性 name 赋给了形参中的 name 呢？为了避免这种混淆的出现，可以采用 this 这种方式，请看修改后的代码：

范例：Person-1.java

```
01  class Person-1
02  {
03      private String name ;
04      private int age ;
05      public Person(String name,int age)
06      {
07          this.name = name ;
08          this.age = age ;
09      }
10  }
```

Person-1.java 这段代码与 **Person.java** 的不同之处在于在第 7、8 行分别加上了 this 关键字。还记得之前说过的 this 表示当前对象吗？那么此时的 this.name 和 this.age 就

分别代表类中的 name 与 age 属性，这个时候再完成赋值操作的话，就可以清楚的知道谁赋值给谁了。完整程序代码如下：

范例：TestJavaThis.java

```
01  class Person
02  {
03      private String name ;
04      private int age ;
05      public Person(String name,int age)
06      {
07          this.name = name ;
08          this.age = age ;
09      }
10      public String talk()
11      {
12          return "我是："+name+"，今年："+age+"岁" ;
13      }
14  }
15  public class TestJavaThis
16  {
17      public static void main(String[] args)
18      {
19          Person p = new Person("张三",25) ;
20          System.out.println(p.talk()) ;
21      }
22  }
```

输出结果：

我是：张三，今年：25 岁

程序说明：

1、 程序第 1~14 行声明了一个名为 Person 的类。

2、 第 5~9 行声明 Person 类的一个构造方法，此构造方法的作用是为类中的属性赋初值。

为了更好的说明 this 是表示当前对象的，下面再举一个范例：判断两个对象是否相等，在这里假设只要姓名、年龄都相同的就为同一个人，否则不是同一个人。

范例：TestCompare.java

```
01  class Person
02  {
03      String name ;
04      int age ;
05      Person(String name,int age)
06      {
07          this.name = name ;
08          this.age = age ;
09      }
10      boolean compare(Person p)
11      {
12          if(this.name.equals(p.name)&&this.age==p.age)
13          {
14              return true ;
15          }
16          else
17          {
18              return false ;
19          }
20      }
21  }
22
23  public class TestCompare
24  {
25      public static void main(String[] args)
26   {
27      Person p1 = new Person("张三",30);
28      Person p2 = new Person("张三",30);
29      System.out.println(p1.compare(p2)?"相等,是同一人!":"不相等,不是同一人!");
30   }
31  }
```

输出结果：

相等,是同一人!

程序说明：

1、 程序 1~21 行声明了一个名为 Person 的类，里面有一个构造方法与一个比较方法。

2、 程序 10~20 行在 Person 类中声明了一个 compare 方法，此方法接收 Person 实例对象的引用。

3、 程序第 12 行比较姓名和年龄是否同时相等。

4、 程序第 29 行，由 p1 调用 compare()方法，将 p2 传入到 compare 方法之中，所以在程序中第 12 行的 this.name，就代表 p1.name，this.age，就代表 p1.age，而传入的参数 p2 则被 compare()方法中的参数 p 表示。

由此不难理解 this 是表示当前对象这一重要概念，所以程序最后输出了“相等，是同一人!”的正确判断信息。

5.9.1 用 this 调用构造方法

如果在程序中想用某一构造方法调用另一构造方法，可以用 this 来实现，具体的调用形式如下：

```
this() ;
```

范例：TestJavaThis1.java

```
01  class Person
02  {
03      String name ;
04      int age ;
05      public Person()
06      {
07          System.out.println("1. public Person()");
08      }
```

```
09          public Person(String name,int age)
10          {
11              // 调用本类中无参构造方法
12              this() ;
13              this.name = name ;
14              this.age = age ;
15              System.out.println("2. public Person(String name,int age)");
16          }
17      }
18      public class TestJavaThis1
19      {
20          public static void main(String[] args)
22          {
23              new Person("张三",25) ;
24          }
25      }
```

输出结果：

1. public Person()

2. public Person(String name,int age)

程序说明：

1、 程序 1~17 行，声明一个名为 Person 的类，类中声明了一个无参、一个有参这样两个构造方法。

2、 程序第 12 行使用 this()调用本类中的无参构造方法。

3、 程序第 23 行声明一 Person 类的匿名对象，调用了有参的构造方法。

由程序 TestJavaThis1.java 可以发现，在第 23 行虽然调用了 Person 中有两个参数的构造方法，但是由于程序第 12 行，使用了 this()调用本类中的无参构造方法，所以程序先去执行 Person 中无参构造方法，之后再去继续执行其他的构造方法。

注意：

有的读者经常会有这样的疑问。如果我把 this()调用无参构造方法的位置任意调换，那不就可以在任何时候都可以调用构造方法了么？实际上这样理解是错误的。构造方法是在实例化一对象时被自动调用的，也就是说在类中的所有方法里，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一行，下面的程序就是一个错误的程序:

```
01  class Person
02  {
03      String name ;
04      int age ;
05      public Person()
06      {
07          System.out.println("1. public Person()");
08      }
09      public Person(String name,int age)
10      {
11          this.name = name ;
12          this.age = age ;
13          // 用 this 调用构造方法，此时不是放在构造方法的首行
14          this() ;
15          System.out.println("2. public Person(String name,int age)");
16      }
17      public String talk()
18      {
19          // 在这里也用一个 this
20          this() ;
21          System.out.println("我是："+name+"，今年："+age+"岁");
22      }
23  }
24  public class TestJavaThis1
25  {
26      public static void main(String[] args)
27      {
28          new Person("张三",25) ;
```

```
29        }
}
```

在此程序中，将 this 调用构造方法的位置放在了任意行，所以程序编译时会有如下警告:

TestJavaThis1.java:14: call to this must be first statement in constructor

this() ;

1 error

由此可见，this()调用构造方法必须放在首行。

5.10 static 关键字的使用

5.10.1 静态变量

在程序中如果用 static 声明变量的话，则此变量称为静态变量。那么什么是静态变量？使用静态变量又有什么好处呢？读者先来看一下下面的范例：

范例：TestStaticDemo1.java

```
01  class Person
02  {
03      String name ;
04      String city ;
05      int age ;
06      public Person(String name, String city, int age)
07      {
08          this.name = name ;
09          this.city = city ;
10          this.age = age ;
11      }
12      public String talk()
13      {
14          return   "我是："+this.name+"，今年："+this.age+"岁，来自："+this.city;
```

```
15          }
16      }
17      public class TestStaticDemo1
18      {
19          public static void main(String[] args)
20          {
21              Person p1 = new Person("张三",25,"中国") ;
22              Person p2 = new Person("李四",30,"中国") ;
23              Person p3 = new Person("王五",35,"中国") ;
24              System.out.println(p1.talk()) ;
25              System.out.println(p2.talk()) ;
26              System.out.println(p3.talk()) ;
27          }
28      }
```

输出结果：

我是：张三，今年：25 岁，来自：中国

我是：李四，今年：30 岁，来自：中国

我是：王五，今年：35 岁，来自：中国

程序说明：

1、 程序 1~16 行声明一个名为 Person 的类，含有三个属性：name、age、city。

2、 程序 6~11 行声明 Person 类的一个构造方法，此构造方法分别为各属性赋值。

3、 程序 12~15 行声明一 talk()方法，此方法用于返回用户信息。

4、 程序 21~23 行分别实例化三个 Person 对象。

5、 程序 24~26 行分别调用类中的 talk()方法，输出用户信息。

由上面的程序可以发现，所有的 Person 对象都有一个 city 属性，而且所有的属性也全部相同，如图 5-11 所示：

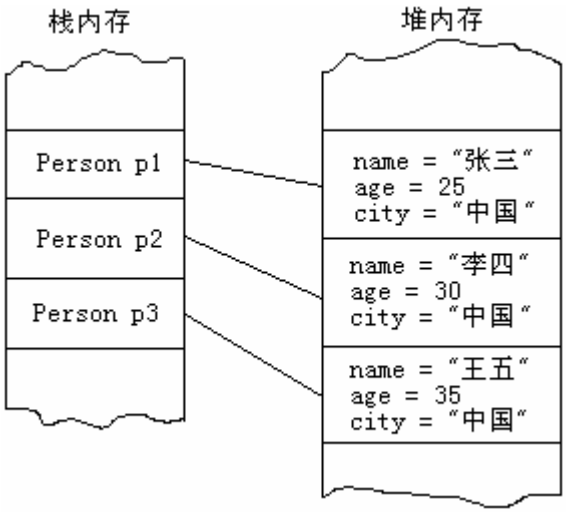


图 5-12

读者可以试着想一想，现在假设程序产生了 50 个 Person 对象，如果想修改所有人的 city 属性的话，是不是就要调用 50 遍 city 属性，进行重新修改，显然太麻烦了。所以在 java 中提供了 static 关键字，用它来修饰类的属性后，则此属性就是公共属性了。将 TestStaticDemo1.java 程序稍作修改就形成了范例 TestStaticDemo2.java，如下所示：

范例：TestStaticDemo2.java

```
01  class Person
02  {
03      String name ;
04      static String city = "中国";
05      int age ;
06      public Person(String name,int age)
07      {
08          this.name = name ;
09          this.age = age ;
10      }
11      public String talk()
12      {
13          return  "我是："+this.name+"，今年："+this.age+"岁，来自："+city;
```

```
14          }
15    }
16    public class TestStaticDemo2
17    {
18          public static void main(String[] args)
19          {
20                Person p1 = new Person("张三",25) ;
21                Person p2 = new Person("李四",30) ;
22                Person p3 = new Person("王五",35) ;
23                System.out.println("修改之前信息："+p1.talk()) ;
24                System.out.println("修改之前信息："+p2.talk()) ;
25                System.out.println("修改之前信息："+p3.talk()) ;
26                System.out.println("    ************** 修改之后信息 ***************");
27                // 修改后的信息
28                p1.city = "美国" ;
29                System.out.println("修改之后信息："+p1.talk()) ;
30                System.out.println("修改之后信息："+p2.talk()) ;
31                System.out.println("修改之后信息："+p3.talk()) ;
32          }
33    }
```

输出结果：

```
修改之前信息:我是：张三，今年：25 岁，来自：中国
修改之前信息:我是：李四，今年：30 岁，来自：中国
修改之前信息:我是：王五，今年：35 岁，来自：中国
   ************** 修改之后信息 ***************
修改之后信息:我是：张三，今年：25 岁，来自：美国
修改之后信息:我是：李四，今年：30 岁，来自：美国
修改之后信息:我是：王五，今年：35 岁，来自：美国
```

程序说明：

1、 程序 1~15 行声明一个名为 Person 的类，含有三个属性：name、age、city。其中 city 为 static 类型。

2、 程序 6~10 行声明 Person 类的一个构造方法，此构造方法作用是分别为 name、age 属性赋值。

3、 程序 11~14 行声明一 talk()方法，此方法用于返回用户信息。

4、 程序 20~22 行分别实例化三个 Person 对象。

5、 程序 23~25 行分别调用类中的 talk()方法，输出用户信息。

6、 程序 28 行修改了 p1 中的 city 属性。

从上面的程序中可以发现，程序只在第 28 行修改了 city 属性，而且只修改了一个对象的 city 属性，但再次输出时，发现全部的对象的 city 值都发生了一样的变化，这就说明了用 static 声明的属性是所有对象共享的。

如下图 5-13 所示：

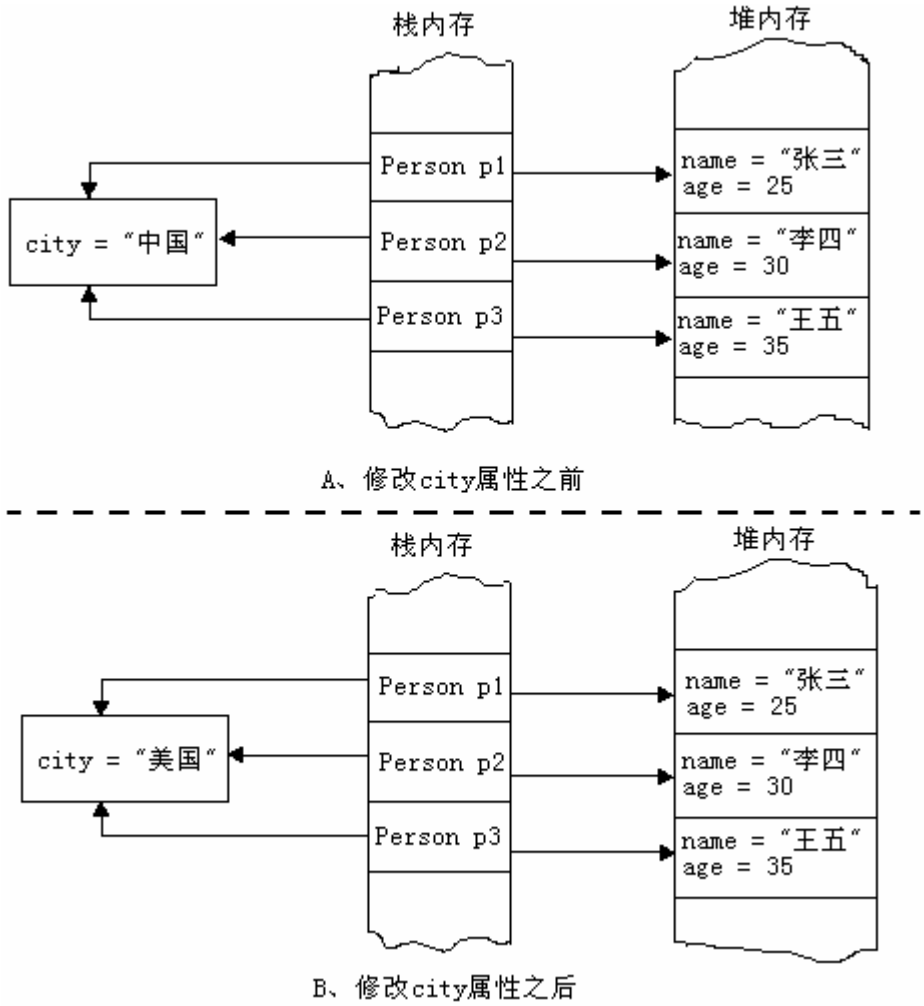


图 5-13　static 变量的使用

在图中可以发现，所有的对象都指向同一个 city 属性，只要当中有一个对象修改了 city 属性的内容，则所有对象都会被同时修改。

另外，读者也需要注意一点，用 static 方式声明的属性，也可以用类名直接访问，拿上面的程序来说，如果想修改 city 属性中的内容，可以用如下方式：

Person.city = "美国" ;

所以有些书上也把用 static 类型声明的变量，称之为“类变量” 。

小提示:

既然 static 类型的变量是所有对象共享的内存空间，也就是说无论最终有多少个对象产生，也都只有一个 static 类型的属性，那可不可以用它来计算类到底产生了多少个实例对象呢？读者可以想一想，只要一个类产生一个新的实例对象，都会去调用构造方法，所以可以在构造方法中加入一些记数操作，如下面的程序:

```
class Person
{
    static int count = 0 ;     // 声明一 static 类型的整型变量
    public Person()
    {

        count++ ;              // 增加了一个对象
        System.out.println("产生了："+count+"个对象！");
    }
}

public class TestStaticDemo3
{
    public static void main(String[] args)
    {
        new Person() ;
        new Person() ;
    }
}
```

5.10.2 静态方法

static 既可以在声明变量时使用，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：TestStaticDemo4.java

```
01  class Person
02  {
03      String name ;
04      private static String city = "中国";
05      int age ;
06      public Person(String name,int age)
07      {
08          this.name = name ;
09          this.age = age ;
10      }
11      public String talk()
12      {
13          return   "我是："+this.name+"，今年："+this.age+"岁，来自："+city;
14      }
15      public static void setCity(String c)
16      {
17          city = c ;
18      }
19  }
20  public class TestStaticDemo4
21  {
22      public static void main(String[] args)
23      {
24          Person p1 = new Person("张三",25) ;
25          Person p2 = new Person("李四",30) ;
26          Person p3 = new Person("王五",35) ;
27          System.out.println("修改之前信息："+p1.talk()) ;
28          System.out.println("修改之前信息："+p2.talk()) ;
29          System.out.println("修改之前信息："+p3.talk()) ;
30          System.out.println("     **********修改之后信息**********");
31          // 修改后的信息
32          Person.setCity("美国") ;
33          System.out.println("修改之后信息："+p1.talk()) ;
34          System.out.println("修改之后信息："+p2.talk()) ;
35          System.out.println("修改之后信息："+p3.talk()) ;
36      }
```

```
37  }
```

输出结果：

```
修改之前信息：我是：张三，今年：25 岁，来自：中国
修改之前信息：我是：李四，今年：30 岁，来自：中国
修改之前信息：我是：王五，今年：35 岁，来自：中国
    **********修改之后信息**********
修改之后信息：我是：张三，今年：25 岁，来自：美国
修改之后信息：我是：李四，今年：30 岁，来自：美国
修改之后信息：我是：王五，今年：35 岁，来自：美国
```

程序说明：

1、 程序 1~19 行声明一个名为 Person 的类，类中含有一个 static 类型的变量 city，并对此对象进行了封装。

2、 15~18 行声明一个 static 类型的方法，此方法也可以用类名直接调用，用于修改 city 属性的内容。

3、 程序第 32 行由 Person 调用 setCity()方法，对 city 的内容进行修改。

注意：

在使用 static 类型声明的方法时需要注意的是:如果在类中声明了一 static 类型的属性，则此属性既可以在非 static 类型的方法中使用，也可以在 static 类型的方法中使用。但用 static 类型的属性调用非 static 类型的属性时，则会出现错误。

```
public class PersonStatic
{
    String name = "张三" ;
    static String city = "中国";
    int age ;
    public PersonStatic(String name,int age)
    {
        this.name = name ;
        this.age = age ;
```

```
    }
    public static void print()
    {
        System.out.println(name);
    }
    public String talk()
    {
        return  "我是："+this.name+"，今年："+this.age+"岁，来自："+city;
    }
}
```

编译结果：

```
PersonStatic.java:13: non-static variable name cannot be referenced from a static context
                System.out.println(name);
                                   ^
1 error
```

可以发现，程序中 print()方法是 static 类型的，而 name 属性是非 static 类型的，所以 print 在调用非 static 类型的 name 属性时就出现了错误。

5.10.3 理解 main()方法

在前面的章节中，已经多次看到，如果一个类要被 Java 解释器直接装载运行，这个类中必须有 main()方法。有了前面所有的知识，读者现在可以理解 main()方法的含义了。

由于 Java 虚拟机需要调用类的 main()方法，所以该方法的访问权限必须是 public，又因为 Java 虚拟机在执行 main()方法时不必创建对象，所以该方法必须是 static 的，该方法接收一个 String 类型的数组参数，该数组中保存执行 Java 命令时传递给所运行的类的参数。

向 java 中传递参数可用如下的命令：

java 类名称 参数 1 参数 2 参数 3

通过运行程序 TestMain.java 来了解如何向类中传递参数以及程序又是如何取得这些参数的，如下所示：

范例：TestMain.java

```
01  public class TestMain
02  {
03      /*
04          public：表示公共方法
05          static：表示此方法为一静态方法，可以由类名直接调用
06          void：表示此方法无返回值
07          main：系统定义的方法名称
08          String args[]：接收运行时参数
09      */
10      public static void main(String[] args)
11      {
12          // 取得输入参数的长度
13          int j = args.length ;
14          if(j!=2)
15          {
16              System.out.println("输入参数个数有错误！") ;
17              // 退出程序
18              System.exit(1) ;
19          }
20          for (int i=0;i<args.length ;i++ )
21          {
22              System.out.println(args[i]) ;
23          }
24      }
25  }
```

运行程序：

java TestMain first second

输出结果：

first

second

程序说明：

1、 程序第 14 行判断输入参数的个数是否为两个参数，如果不是，则退出程序。

2、 所有的接收的参数都被存放在 args[]字符串数组之中，用 for 循环输出全部内容。

5.10.4 静态代码块

一个类可以使用不包含在任何方法体中的静态代码块，当类被载入时，静态代码块被执行，且只执行一次，静态代码块经常用来进行类属性的初始化。如下面的程序代码所示：

范例：TestStaticDemo5.java

```
01  class Person
02  {
03      public Person()
04      {
05          System.out.println("1.public Person()");
06      }
07      // 此段代码会首先被执行
08      static
09      {
10          System.out.println("2.Person 类的静态代码块被调用！");
11      }
12  }
13  public class TestStaticDemo5 {
14      // 运行本程序时，静态代码块会被自动执行
15      static
16      {
17          System.out.println("3.TestStaticDemo5 类的静态代码块被调用！");
18      }
19      public static void main(String[] args) {
```

```
20              System.out.println("4.程序开始执行！");
21              // 产生两个实例化对象
22              new Person() ;
23              new Person() ;
24          }
25  }
```

输出结果：

3.TestStaticDemo5 类的静态代码块被调用！

4.程序开始执行！

2.Person 类的静态代码块被调用！

1.public Person()

1.public Person()

程序说明：

1、 程序 1~12 行声明一个名为 Person 的类。

2、 程序 8~11 行声明一静态代码块，此代码块放在 Person 类之中。

3、 程序 15~18 行在类 TestStaticDemo5 中也声明一静态代码块。

4、 22、23 行分别产生了两个 Person 类的匿名对象。

从程序运行结果中可以发现，放在 TestStaticDemo5 类中的静态代码块首先被调用，这是因为程序首先执行 TestStaticDemo5 类，所以此程序的静态代码块会首先被执行。程序在 22、23 行产生了两个匿名对象，可以发现 Person 类中的静态代码块只执行了一次，而且静态代码块优先于静态方法，由此可以发现，静态代码块可以为静态属性初始化。

5.11 构造方法的私有

方法依实际需要，可分为 public 与 private。同样的，构造方法也有 public 与 private 之分。到目前为止，所使用的构造方法均属于 public，它可以在程序的任何地方被调

用，所以新创建的对象也都可以自动调用它。如果构造方法被设为 private，则无法在该构造方法所在的类以外的地方被调用。请看下面的程序：

范例：TestSingleDemo1.java

```
01  public class TestSingleDemo1
02  {
03      private TestSingleDemo1()
04      {
05          System.out.println("private TestSingleDemo1 .") ;
06      }
07      public static void main(String[] args)
08      {
09          new TestSingleDemo1() ;
10      }
11  }
```

输出结果：

```
private TestSingleDemo1 .
```

由上面程序可以发现，程序第 3 行在声明构造方法的时候将之声明为 private 类型，则此构造方法只能在本类内被调用。同时读者可以发现，而在本程序中的 main 方法也放在 TestSingleDemo1 类的内部，所以在本类中可以自己产生实例化对象。

看过上面的程序，这么做似乎没有什么意义，因为一个类最终都会由外部去调用，如果这么做的话，岂不是所有构造方法被私有化了的类都需要这么去调用了吗？那程序岂不是会有很多的 main()方法了吗？举这个例子主要是让读者清楚：构造方法虽然被私有了，但并不一定是说此类不能产生实例化对象，只是产生这个实例化对象的位置有所变化，即只能在本类中产生实例化对象。请看下面的范例：

范例：TestSingleDemo2

```
01  class Person
02  {
03      String name ;
04  // 在本类声明一 Person 对象 p，注意此对象用 final 标记，表示不能再重新实例化
```

```
05          private static final Person p = new Person() ;
06          private Person()
07          {
08                  name = "张三" ;
09          }
10          public static Person getP()
11          {
12                  return p ;
13          }
14  }
15
16  public class TestSingleDemo2
17  {
18          public static void main(String[] args)
19          {
20                  // 声明一 Person 类的对象
21                  Person p = null ;
22                  p = Person.getP() ;
23                  System.out.println(p.name) ;
24          }
25  }
```

输出结果：

张三

程序说明：

1、 程序 6~9 行将 Person 类的构造方法封装起来，外部无法通过其构造方法产生实例化对象。

2、 程序第 5 行声明一个 Person 类的实例化对象，此对象是在 Person 类内部实例化，所以可以调用私有构造方法。另外，此对象被标识为 static 类型，表示为一静态属性，同时此对象被私有化，另外在声明 Person 对象的时候加上了一个 final 关键字，此关键字表示 Person 的对象 p 不能被重新实例化。（关于 final 的介绍请参见第八章接口部分的内容）

3、 程序第 21 行声明一个 Person 类的对象 p，但未实例化。

4、 程序第 22 行调用 Person 类中的 getP()方法，此方法返回 Person 类的实例化对象。

从上面程序中可以发现，无论在 Person 类的外部声明多少个对象，最终得到的都是同一个引用，因为此类只能产生一个实例对象，这种做法在设计模式中称为单态模式。而所谓设计模式也就是在大量的实践中总结和理论化之后优选的代码结构、编程风格以及解决问题的思考方式。有兴趣的读者可以研究一下。

5.12 对象数组的使用

在前面的章节中已介绍过如何以数组来保存基本数据类型的变量。相同的，对象也可以用数组来存放，通过下面两个步骤来实现：

1、 声明类类型的数组变量，并用 new 分配内存空间给数组。

2、 用 new 产生新的对象，并分配内存空间给它

例如，要创建三个 Person 类类型的数组元素，可用下列的语法：

```
Person p[] ;                    // 声明 Person 类类型的数组变量
p = new Person[3] ;             // 用 new 分配内存空间
```

创建好数组元素之后，便可把数组元素指向由 Person 类所产生的对象：

```
p[0] = new Person () ;
p[1] = new Person () ;          } 用 new 产生新的对象，并分配内存空间给它
p[2] = new Person () ;
```

此时，p[0]、p[1]、p[2]是属于 Person 类类型的变量，它们分别指向新建对象的内存参考地址。当然也可以写成如下形式：

```
Person p[] = new Person[3];     // 创建对象数组元素，并分配内存空间
```

当然，也可以利用 for 循环来完成对象数组内的初始化操作，此方式属于动态初始化：

```
for(int i=0;i<p.length;i++)
{
    p[i] = new Person() ;
}
```

或者采用静态方式初始化对象数组：

```
Person p[] = {new Person(),new Person(),new Person()} ;
```

范例：TestObjectArray.java

```
01  class Person
02  {
03      String name ;
04      int age ;
05      public Person()
06      {
07      }
08      public Person(String name,int age)
09      {
10          this.name = name ;
11          this.age = age ;
12      }
13      public String talk()
14      {
15          return "我是："+this.name+"，今年："+this.age+"岁" ;
16      }
17  }
18  public class TestObjectArray
19  {
20      public static void main(String[] args)
21      {
22          Person p[] = {
23            new Person("张三",25),new Person("李四",30),new Person("王五",35)
24                      } ;
25          for(int i=0;i<p.length;i++)
26          {
27              System.out.println(p[i].talk()) ;
```

```
28              }
29          }
30  }
```

输出结果：

我是：张三，今年：25 岁

我是：李四，今年：30 岁

我是：王五，今年：35 岁

程序说明：

1、 程序第 22~24 行用静态声明方式声明了三个对象的 Person 类的对象数组。

2、 程序第 25~28 行用 for 循环输出所有对象，并分别调用 talk()方法打印信息。

5.13 内部类

现在已经知道，在类内部可定义成员变量与方法，有趣的是，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 5-6 定义内部类】

```
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                  ┐             │ 外部类
    {                                          │ 内部类      │
        // 内部类的成员                        ┘             │
    }                                                        │
}                                                            ┘
```

范例：InnerClassDemo.java

```
01  class Outer
02  {
03      int score = 95;
04      void inst()
05      {
06          Inner in = new Inner();
07          in.display();
08      }
09      class Inner
10      {
11          void display()
12          {
13              System.out.println("成绩: score = " + score);
14          }
15      }
16  }
17  public class InnerClassDemo
18  {
19      public static void main(String[] args)
20      {
21          Outer outer = new Outer();
22          outer.inst();
23      }
24  }
```

输出结果：

```
成绩: score = 95
```

程序说明：

1、 程序 9~15 行，在 Outer 类的内部声明一 Inner 类，此类之有一个 display()方法，用于打印外部类中的 score 属性。

2、 程序 4~8 行声明一 inst()方法，此方法用于实例化内部类的对象 in。

由上面的程序可以发现，内部类 Inner 可以直接调用外部类 Outer 中的 score 属性，

现在如果把内部类拿到外面来单独声明，则在使用外部类中的 score 属性时，则需要先产生 Outer 类的对象，再由对象去调用 Outer 类的 score 属性。

可以发现由于使用了内部类操作，所以程序在调用 score 属性的时候减少了创建对象的操作，从而省去了一部分的内存开销，但是内部类在声明时，会破坏程序的结构，在开发中往往不建议读者去使用。

由上面的程序可以发现，外部类声明的属性可以被内部类所访问，那内部类声明的属性可以被外部类所访问么？请看下面的范例：

范例：InnerClassDemo1.java

```
01  class Outer
02  {
03      int score = 95;
04      void inst()
05      {
06          Inner in = new Inner();
07          in.display();
08      }
09      public class Inner
10      {
11          void display()
12          {
13              // 在内部类中声明一 name 属性
14              String name = "张三" ;
15              System.out.println("成绩: score = " + score);
16          }
17      }
18      public void print()
19      {
20          // 在此调用内部类的 name 属性
21          System.out.println("姓名： "+name) ;
22      }
23  }
24  public class InnerClassDemo1
25  {
26      public static void main(String[] args)
```

```
27          {
28                  Outer outer = new Outer();
29                  outer.inst();
30          }
31  }
```

编译结果：

InnerClassDemo1.java:21: cannot resolve symbol

symbol : variable name

location: class Outer

System.out.println("姓名： "+name) ;

^

1 error

从程序编译结果中可以发现，外部类是无法找到内部类中所声明的属性。而内部类则可以访问外部类的属性。

前面已经学过了 static 的用法，用 static 可以声明属性或方法，而用 static 也可以声明内部类，用 static 声明的内部类则变成外部类，但是用 static 声明的内部类不能访问非 static 的外部类属性。

范例：InnerClassDemo2.java

```
01  class Outer
02  {
03          int score = 95;
04          void inst()
05          {
06                  Inner in = new Inner();
07                  in.display();
08          }
09          // 这里用 static 声明内部类
10          static class Inner
11          {
12                  void display()
```

```
13              {
14                  System.out.println("成绩: score = " + score);
15              }
16          }
17      }
18      public class InnerClassDemo2
19      {
20          public static void main(String[] args)
21          {
22              Outer outer = new Outer();
23              outer.inst();
24          }
25      }
```

编译结果：

InnerClassDemo2.java:14: non-static variable score cannot be referenced from a static context

System.out.println("成绩: score = " + score);

^

1 error

由编译结果可以发现，由于内部类 Inner 声明为 static 类型，所以无法访问外部类中的非 static 类型属性 score。

5.13.1 在类外部引用内部类

内部类也可以通过创建对象从外部类之外被调用，只要将内部类声明为 public 即可，请看下面的范例：

范例：InnerClassDemo3.java

```
01  class Outer
02  {
03      int score = 95;
04      void inst()
```

```
05          {
06                  Inner in = new Inner();
07                  in.display();
08          }
09          public class Inner
10          {
11                  void display()
12                  {
13                          System.out.println("成绩: score = " + score);
14                  }
15          }
16  }
17  public class InnerClassDemo3
18  {
19          public static void main(String[] args)
20          {
21                  Outer outer = new Outer();
22                  Outer.Inner inner = outer.new Inner();
23                  inner.display() ;
24          }
25  }
```

输出结果：

成绩: score = 95

程序说明：

1、 程序 9~15 行用 public 声明一内部类，此内部类可被外部类访问。

2、 程序第 22 行用外部类的对象去实例化一内部类的对象。

5.13.2 在方法中定义内部类

内部类不仅可以在类中定义，也可以在方法中定义内部类。

范例：InnerClassDemo4.java

```
01  class Outer
02  {
03      int score = 95;
04      void inst()
05      {
06          class Inner
07          {
08              void display()
09              {
10                  System.out.println("成绩: score = " + score);
11              }
12          }
13          Inner in = new Inner();
14          in.display();
15      }
16  }
17  public class InnerClassDemo4
18  {
19      public static void main(String[] args)
20      {
21          Outer outer = new Outer();
22          outer.inst() ;
23      }
24  }
```

输出结果：

成绩: score = 95

程序说明：

1、 程序 4~15 行在 Outer 类中声明一 inst()方法，在此方法中声明一 Inner 的内部类，同时产生了 Inner 的内部类实例化对象，调用其内部的方法。

2、 程序第 21 行，产生一 Outer 类的实例化对象，22 行调用 Outer 类中的 inst()方法。

在方法中定义的内部类只能访问方法中的 final 类型的局部变量，因为用 final 定义的局部变量相当于是一个常量，它的生命周期超出方法运行的生命周期。如下面范例所示：

范例：InnerClassDemo5

```
01  class Outer
02  {
03      int score = 95;
04      void inst(final int s)
05      {
06          int temp = 20 ;
07          class Inner
08          {
09              void display()
10              {
11                  System.out.println("成绩: score = " + (score+s+temp));
12              }
13          }
14          Inner in = new Inner();
15          in.display();
16      }
17  }
18  public class InnerClassDemo5
19  {
20      public static void main(String[] args)
21      {
22          Outer outer = new Outer();
23          outer.inst(5) ;
24      }
25  }
```

编译结果：

InnerClassDemo5.java:11: local variable temp is accessed from within inner class; needs to be declared final

System.out.println(" 成 绩 : score = " + (score+s+temp));

^

1 error

由编译结果可以发现，内部类可以访问用 final 标记的变量 s，却无法访问在方法内声明的变量 temp，所以只要将第 6 行的变量声明时，加上一个 final 修饰即可：

final int temp = 20 ;

完整程序如下：

范例：InnerClassDemo5.java

```
01  class Outer
02  {
03      int score = 95;
04      void inst(final int s)
05      {
06          final int temp = 20 ;
07          class Inner
08          {
09              void display()
10              {
11                  System.out.println("成绩: score = " + (score+s+temp));
12              }
13          }
14          Inner in = new Inner();
15          in.display();
16      }
17  }
18  public class InnerClassDemo5
19  {
20      public static void main(String[] args)
21      {
22          Outer outer = new Outer();
23          outer.inst(5) ;
24      }
25  }
```

输出结果：

成绩: score = 120

程序说明：

1、 程序第 4~16 行声明一 inst()方法，并在此方法参数处声明一 final 类型的变量。

2、 程序第 6 行声明一 final 类型的变量 temp，此变量用 final 声明，表示此变量可以由 Inner 类调用。

3、 程序第 22 行声明并实例化一 Outer 类型的对象 outer。

4、 outer 对象调用本类中的 inst()方法。

5.14 Java 文档注释

在本书的第一部分已经提到过，Java 支持三种形式的注释。前两种是// 和/*… */。第三种方式被称为文档注释。它以“/**”开始，以“*/”标志结束。文档注释提供将程序信息嵌入到程序中的功能。开发者可以使用 javadoc 工具将信息取出，然后转换为 HTML 文件。文档注释提供了编写程序文档的便利方式。

5.14.1 javadoc 标记

javadoc程序识别下列标记：

Tag标记意义

@author	确定类的作者
@deprecated	指示反对使用这个类或成员
{@docRoot}	指定当前文档的根目录路径(Java2的1.3版新增)
@exception	确定一个方法引发的异常
{@link}	插入对另一个主题的内部链接
@param	为方法的参数提供文档
@return	为方法的返回值提供文档
@see	指定对另一个主题的链接
@serial	为默认的可序列化字段提供文档
@serialData	为writeObject()或者writeExternal()方法编写的数据提供文档
@serialField	为ObjectStreamField组件提供文档
@since	当引入一个特定改变时，声明发布版本
@throws	与@exception相同
@version	指定类的版本

正如上面看到的那样，所有的文档标记都以“(@)”标志开始。在一个文档注释中，也可以使用其它的标准HTML标记。然而，一些标记（如标题）是不能使用的，因为它们会破坏由javadoc生成的HTML文件外观。

可以使用文档注释为类、接口、域、构造方法和方法提供文档。在所有这些情况中，文档注释必须紧接在被注释的项目之前。为变量做注释可以使用@see, @since, @serial,@serialField, 和@deprecated文档标记。为类做注释可以使用@see, @author, @since,@deprecated, 和@version 文档标记。为方法做注释可以用@see, @return, @param, @since,@deprecated, @throws, @serialData, 和@exception文档标记。而{@link} 或{@docRoot}标记则可以用在任何地方。下面详细列出了每个标记的使用方法：

@ author

标记@author 指定一个类的作者，它的语法如下：

@author description

其中， description 通常是编写这个类的作者名字。标记@author 只能用在类的文档中。在执行javadoc时，需要指定-author 选项，才可将@author 域包括在HTML文档

中。

@deprecated

@deprecated 标记指示不赞成使用一个类或是一个成员。建议使用@see 标记指示程序员其他的正确的选择。其语法如下：

```
@deprecated description
```

其中description 是描述所反对的信息。由@deprecated 标记指定的信息由编译器识别，包括在生成的.class文件中，因此在编译Java源文件时，程序员可以得到这个信息。

@deprecated 标记可以用于变量，方法和为类做注释的文档中。

{@docRoot}

{@docRoot}指定当前文档的根目录路径。

@exception

@exception 标记描述一个方法的异常，其语法如下：

```
@exception exception-name explanation
```

其中，异常的完整名称由exception-name指定，explanation 是描述异常是如何产生的字符串。@exception 只用于为方法做注释的文档。

{@link}

{@link} 标记提供一个附加信息的联机超链接。其语法如下：

```
{@link name text}
```

其中，name 是加入超链接的类或方法的名字， text 是显示的字符串。

@param

@param 标记注释一个方法的参数。其语法如下所示：

```
@param parameter-name explanation
```

其中parameter-name指定方法的参数名。这个参数的含义由explanation描述。

@param 标记只用在为方法做注释的文档中。

@return

@return 标记描述一个方法的返回值。其语法如下：

```
@return explanation
```

其中，explanation 描述方法返回值的类型和含义。@return 标记只用为方法做注释的文档中。

@see

@see 标记提供附加信息的引用。最常见的使用形式如下所示：

```
@see anchor
@see pkg.class#member text
```

在第一种格式中，anchor 是一个指向绝对或相对URL的超链接。第二种格式，pkg.class#member 指示项目的名字，text是项目的文本显示。文本参数是可选的，如果不用，显示由pkg.class#membe 指定的项目。成员名，也是可选的。因此，除了指向特定方法或者字段的引用之外，还可以指定一个引用指向一个包，一个类或是一个接口。名字可以是完全限定的，也可以是部分限定的。但是，成员名（如果存在的话）之前的点必须被替换成一个散列字符。

@serial

@serial 标记为默认的可序列化字段定义注释文档。其语法如下：

```
@serial description
```

其中， description 是字段的注释。

@serialData

@serialData 标记为writeObject( )或者writeExternal( )方法编写的数据提供文档。其语法如下：

```
@serialData description
```

其中，description 是数据的注释。

@serialField

@serialField 标记为ObjectStreamField组件提供注释，其语法如下：

```
@serialField name type description
```

其中，name是域名，type是域的类型，description是域的注释。

@since

@since标记声明由一个特定发布版本引入的类或成员。其语法如下：

```
@since release
```

其中，release是指示这个特性可用的版本或发布的字符串。@since 标记可以用在为变量、方法和类的做注释文档中。

@throws

@throws 标记与@exception标记的含义相同。

@version

@version标记指示类的版本。其语法如下：

```
@version info
```

其中，info 是包含版本信息的字符串，典型情况下是如2.2这样的版本号。@version标记只用在为类的做注释文档中。在执行javadoc时，指定-version 选项，可将@version域包含在HTML文档中。

5.14.2 文档注释的一般形式

在用/**开头后，第一行，或者头几行是类、变量或方法的主要描述。其后，可以包括一个或多个不同的@标记。每个@标记必须在一个新行的开头，或者跟随一个星号(*)之后。同类型的多个标记应该组合在一起。例如，如果有三个@see标记，最好是一个挨着一个。

下面是一个类的文档注释的例子：

```
/**
* This class draws a bar chart.
* @author Herbert Schildt
* @version 3.2
*/
```

5.14.3 javadoc 的输出

javadoc程序将Java程序的源文件作为输入，输出几个包含该程序文档的HTML文件。每个类的信息都在其自己的HTML文件中。同时，javadoc还输出一个索引和一个层次结构树。

Javadoc还可生成其他HTML文件。不同版本的javadoc工作方式也有所不同，应该仔细阅读Java开发系统的说明书以了解所使用的版本的细节处理的规定。

看下面的例子，下面的例子说明了 JavaDoc 的使用方法：

范例：PersonJavaDoc.java

```
/**
 *Title: PersonJavaDoc<br>
 *Description: 通过 PersonJavaDoc 类来说明 Java 中的文档注释<br>
 *Copyright:(c) 2004 www.sun.com.cn<br>
 *Company : sun java </br>
 *@author lixinghua
 *@version 1.00
 */

public class PersonJavaDoc
{
    private String name = "Careers";
    private String sex = "male";
    private int age = 30;
    /**
     *这是 PersonJavaDoc 对象无参数的构造方法
```

```
 */

public PersonJavaDoc()
{   }
/**
 *这是 PersonJavaDoc 类的有参构造方法
 *@param name PersonJavaDoc 的名字
 *@param sex PersonJavaDoc 的性别
 *@param age PersonJavaDoc 的年龄
 */
public PersonJavaDoc(String name,String sex,int age)
{
    this.name = name ;
    this.sex = sex;
    this.age = age;
}
/**
 *这是设置 name 的值的方法，将参数 name 的值赋给变量 this.name
 *@param name PersonJavaDoc 的名字
 */
public void setName(String name)
{
    this.name = name;
}
/**
 *这是取得 name 的值的方法
 *@返回 name 的值
 */
public String getName()
{
    return name;
}
/**
 *这是取得 age 的值的方法
 *@返回 age 的值
 */
public int getAge()
{
```

```
        return age;
    }
    /**
     *这是设置 sex 的值的方法，将参数 sex 的值赋给变量 this.sex
     *@param sex PersonJavaDoc 的性别
     */
    public void setSex(String sex)
    {
        this.sex = sex ;
    }
    /**
     *这是取得 sex 的值的方法
     *@返回 sex 的值
     */
    public String getSex()
    {
        return sex;
    }
    /**
     *这是设置 age 的值的方法
     *@param age PersonJavaDoc 的年龄
     */
    public void setAge(int age)
    {
        if(age<0)
        this.age = 0;
        else
        this.age = age;
    }
    public void shout()
    {
        System.out.println("我是 "+name+"，我性别 "+sex+"，今年 "+age+" 岁！
");
    }
}
```

之后用 javadoc 命令进行编译，请看下面的命令：

```
javadoc -d PersonJavaDoc -version -author PersonJavaDoc.java
```

-d：表示生成目录，目录名称为 PersonJavaDoc

-version：表示要求 javadoc 程序在说明文件中加入版本信息。

-author：表示要求 javadoc 程序在说明文件中加入作者信息。

最终可以发现在硬盘上新生成了一个 PersonJavaDoc 的文件夹，如图 5-14 所示：

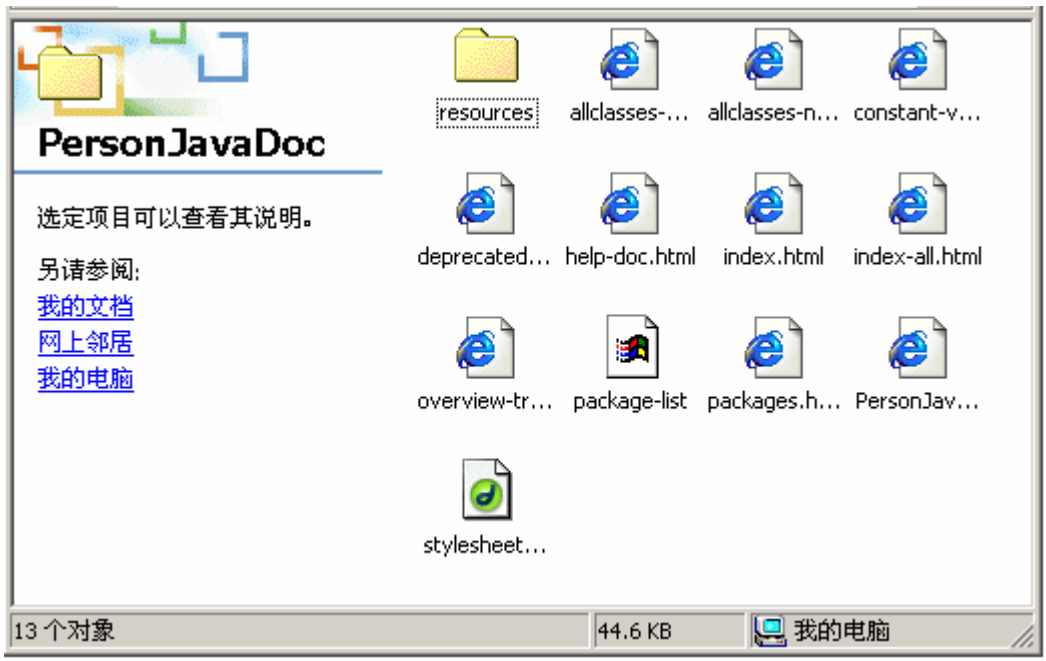


图 5-14　PersonJavaDoc 文件夹的内容

打开文件夹里的 index.html 文件可以看见如下所示的界面：

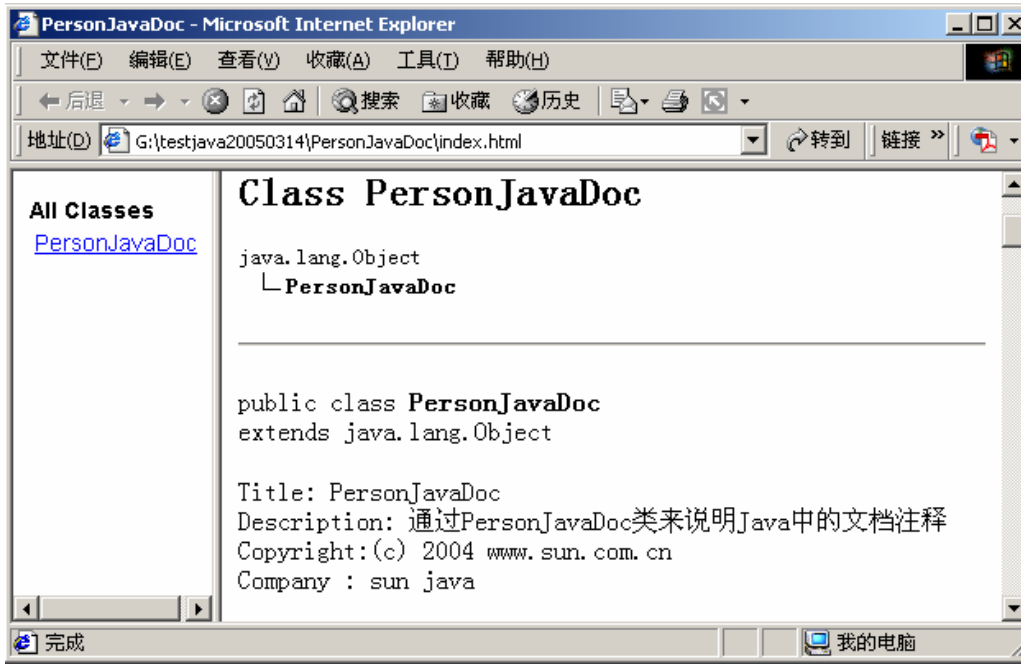


图 5-15　index.html

• 本章摘要：

1、 “类”是把事物的数据与相关的功能封装在一起，形成的一种特殊结构，用以表达对真实世界的一种抽象概念。

2、 Java 把数据成员称为 field（属性），把方法成员称为 method（方法）。

3、 由类所创建的对象称为 instance，译为“实例”。

4、 创建属于某类的对象，可通过下面两个步骤来达成：（1）、声明指向“由类所创建的对象”的变量。（2）、利用 new 创建新的对象，并指派给步骤一中所创建的变量。

5、 要访问到对象里的某个属性（field）时，可通过“对象名称.属性”语法来实现，如果要调用封装在类里的方法，则可通过“对象名称.method”语法来实现。

6、 有些方法不必传递任何数据给调用端程序，因此是没有返回值的。若方法本身没有返回值，则必须在方法定义语句前面加上关键字 void。

7、 私有成员（private member）可限定类中的属性，被限制成私有的属性仅能供同一类内的方法所访问。

8、 类外部可访问到类内部的公有成员（public member）。

9、 “封装”（encapsulation）：是把属性和方法包装在一个类内以限定成员的访问，以起到保护数据的作用。

10、构造方法可视为一种特殊的方法，它的主要作用是为所创建的对象赋初值。

11、构造方法的名称必须与其所属的类的类名称相同，且不能有返回值。

12、从某一构造方法内调用另一构造方法，是通过 this() 这个关键字来完成的。

13、构造方法有 public 与 private 之分。声明为 public 的构造方法可以在程序的任何地方被调用，所以新创建的对象都可以自动调用它。而被声明为 private 的则无法在该构造方法所在的类以外的其它地方被调用。

14、如果构造方法省略不写，Java 则会自动调用默认的构造方法（默认的构造方法没有任何参数）。

15、“基本类型的变量”是指用 int、double 等关键字所声明的变量，而由类声明而得的变量，称之为“类类型的变量”，它是属于“非基本类型的变量”的一种。

16、对象也可以用数组来存放，但必须有下面两个步骤：（1）、声明类类型的数组变量，并用 new 分配内存空间给数组。（2）、用 new 产生新的对象，并分配内存空

间给它。

17、如果在类 Outer 的内部再定义一个类 Inner，此时类 Inner 称为内部类（inner class），而类 Outer 则称为外部类（outer class）。

第六章　类的继承

在前面的两章里已经了解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可以以既有的类为基础，进而派生出新的类。通过这种方式，便能快速地开发出新的类，而不需编写相同的程序代码，这也就是程序代码再利用的概念。本节将介绍继承的概念以及其实际的应用。

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，里面有 name 与 age 两个属性，而另外一个 Student 类，需要有 name、age、school 三个属性，如图 6-1 所示，从这里可以发现 Person 中已经存在有 name 和 age 两个属性，所以不希望在 Student 类中再重新声明这两个属性，这个时候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，也就是引出了接下来所要介绍的类的继承概念。

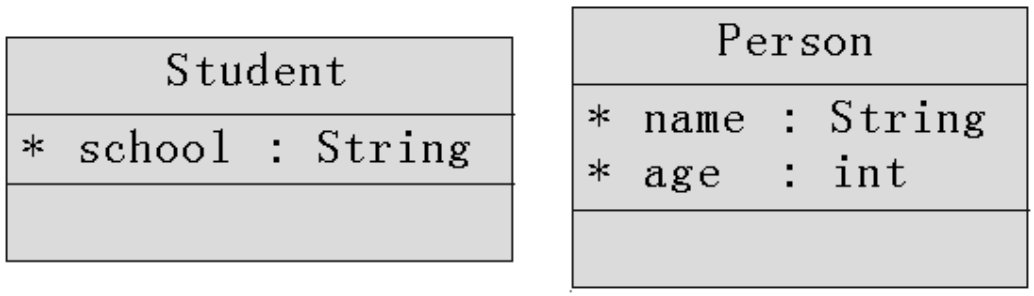


图 6-1　Student 类与 Person 类

在这里希望 Student 类能够将 Person 类的内容继承下来后继续使用，可用图 6-2 表示：

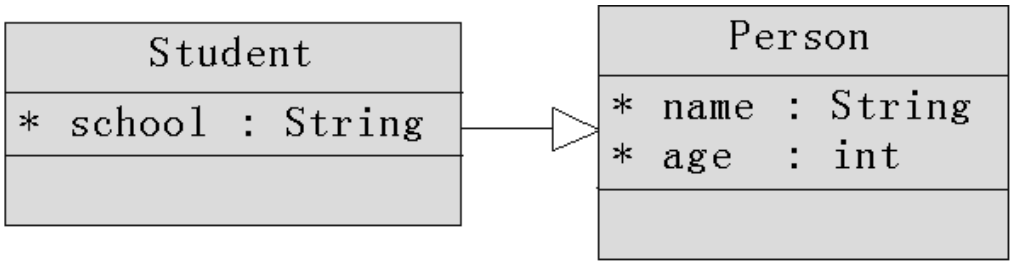


图 6-2　Person 与 Student 的继承关系

Java 类的继承，可用下面的语法来表示：

【 格式 6-1 类的继承格式】

```
class 父类                          // 定义父类
{
}
class 子类 extends 父类     // 用 extends 关键字实现类的继承
{
}
```

范例：TestPersonStudentDemo.java

```
01  class Person
02  {
03      String name ;
04      int age ;
05  }
06  class Student extends Person
07  {
08      String school ;
09  }
10
11  public class TestPersonStudentDemo
12  {
13      public static void main(String[] args)
14      {
15          Student s = new Student() ;
16          // 访问 Person 类中的 name 属性
```

```
17              s.name = "张三" ;
18              // 访问 Person 类中的 age 属性
19              s.age = 25 ;
20              // 访问 Student 类中的 school 属性
21              s.school = "北京" ;
22              System.out.println("姓名："+s.name+"，年龄："+s.age+"，学校："+s.school) ;
23          }
24      }
```

输出结果：

姓名：张三，年龄：25，学校：北京

程序说明：

1、 程序 1~5 行声明一个名为 Person 的类，里面有 name 与 age 两个属性。

2、 程序第 6~9 行声明一个名为 Student 的类，并继承自 Person 类。

3、 程序第 15 行声明并实例化一 Student 类的对象

4、 程序第 17、19、21 行分别用 Student 类的对象调用程序中的 name、age、school 属性。

由上面的程序可以发现，在 Student 类中虽然并未定义 name 与 age 属性，但在程序外部却依然可以调用 name 或 age，这是因为 Student 类直接继承自 Person 类，也就是说 Student 类直接继承了 Person 类中的属性，所以 Student 类的对象才可以访问到父类中的成员。以上所述，可用图 6-3 表示：

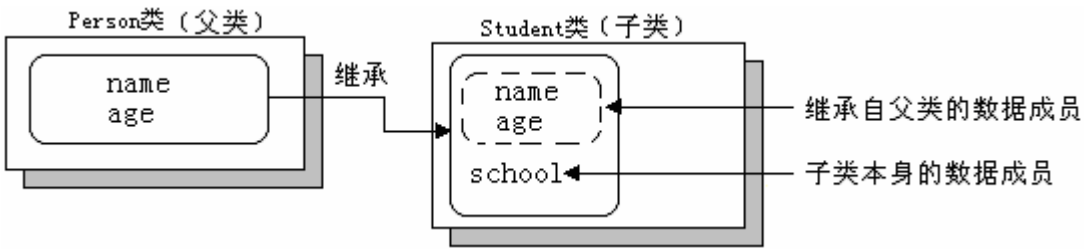


图 6-3　Person 与 Student 类的继承图

注意：

在 java 中只允许单继承，而不允许多重继承，也就是说一个子类只能有一个父类，但是 java 中却允许多层继承。

多重继承:

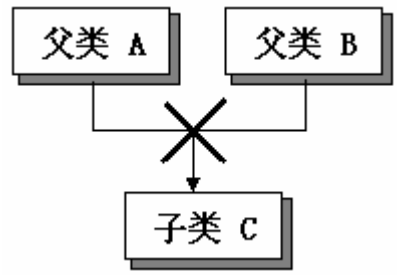


图 6-4　多重继承

```
class A
{}
class B
{}
class C extends A,B
{}
```

由上面可以发现类 C 同时继承了类 A 与类 B，也就是说 C 类同时继承了两个父类，这在 JAVA 中是不允许的。

多层继承:

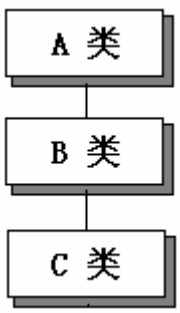


图 6-5　多层继承

```
class A
{}
```

```
class B extends A
{}
class C extends B
{}
```

由上面可以发现类 B 继承了类 A，而类 C 又继承了类 B，也就是说类 B 是类 A 的子类，而类 C 则是类 A 的孙子类。

6.1.1 子类对象的实例化过程

既然子类可以继承直接父类中的方法与属性，那父类中的构造方法呢？请看下面的范例：

范例：TestPersonStudentDemo1.java

```
01  class Person
02  {
03      String name ;
04      int age ;
05      // 父类的构造方法
06      public Person()
07      {
08          System.out.println("1.public Person(){}") ;
09      }
10  }
11  class Student extends Person
12  {
13      String school ;
14      // 子类的构造方法
15      public Student()
16      {
17          System.out.println("2.public Student(){}");
18      }
19  }
20

21  public class TestPersonStudentDemo1
22  {
```

```
23          public static void main(String[] args)
24          {
25                  Student s = new Student() ;
26          }
27  }
```

输出结果：

1.public Person(){}

2.public Student(){}

程序说明：

1、 程序 1~10 行，声明一 Person 类，此类中有一无参构造方法。

2、 程序 11~19 行，声明一 Student 类，此类继承自 Person 类，此类中也有一无参构造方法。

3、 程序 25 行，声明并实例化一 Student 类的对象 s。

从程序输出结果中可以发现，虽然程序第 25 行实例化的是子类的对象，但是程序却先去调用父类中的无参构造方法，之后再调用了子类本身的构造方法。所以由此可以得出结论，子类对象在实例化时会默认先去调用父类中的无参构造方法，之后再调用本类中的相应构造方法。

实际上在本范例中，实际上在子类构造方法的第一行默认隐含了一个“super()”语句，上面的程序如果改写成下面的形式，也是可以的：

```
class Student extends Person
{
    String school ;
    // 子类的构造方法
    public Student()
    {
        super() ;        <-------- 实际上在程序的这里隐含了这样一条语句
        System.out.println("2.public Student(){}");

    }
}
```

小提示:

在子类继承父类的时候经常会有下面的问题发生，请看下面的范例:

范例：TestPersonStudentDemo2.java

```
01  class Person
02  {
03      String name ;
04      int age ;
05      // 父类的构造方法
06      public Person(String name,int age)
07      {
08          this.name = name ;
09          this.age = age ;
10      }
11  }
12  class Student extends Person
13  {
14      String school ;
15      // 子类的构造方法
16      public Student()
17      {
18      }
19  }
20
21  public class TestPersonStudentDemo2
22  {
23      public static void main(String[] args)
24      {
25          Student s = new Student() ;
26      }
27  }
```

编译结果:

TestPersonStudentDemo2.java:17: Person(java.lang.String,int) in

Person cannot

be applied to ()

{}

^

1 error

由编译结果可以发现，系统提供的出错信息是因为无法找到 Person 类，所以造成了编译错误，这是为什么呢？读者可以发现，在类 Person 中提供了一个有两个参数的构造方法，而并没有明确的写出无参构造方法，在前面本书已提到过，如果程序中指定了构造方法，则默认构造方法不会再生成，本例就是这个道理。由于第 25 行实例化子类对象时找不到父类中无参构造方法，所以程序出现了错误，而只要在 Person 类中增加一个什么都不做的构造方法，就这一问题就可以解决了。对范例 TestPersonStudentDemo2.java 作相应的修改就形成了范例 TestPersonStudentDemo3.java，如下所示：

范例：TestPersonStudentDemo3.java

```
01  class Person
02  {
03      String name ;
04      int age ;
05      // 增加一个什么都不做的无参构造方法
06      public Person(){}
07      // 父类的构造方法
08      public Person(String name,int age)
09      {
10          this.name = name ;
11          this.age = age ;
12      }
13  }
14  class Student extends Person
15  {

16      String school ;
17      // 子类的构造方法
18      public Student()
```

```
19        {}
20    }
21
22    public class TestPersonStudentDemo3
23    {
24        public static void main(String[] args)
25        {
26            Student s = new Student() ;
27        }
28    }
```

读者可以发现，在程序的第 6 行声明了一 Person 类的无参的且什么都不做的构造方法，所以程序在编译就可以正常通过了。

6.1.2 super 关键字的使用

在上面的程序中曾经提到过 super 的使用，那 super 到底是什么呢？从 TestPersonStudentDemo1 中读者应该可以发现，super 关键字出现在子类中，而且是去调用了父类中的构造方法，所以可以得出结论：super 主要的功能是完成子类调用父类中的内容，也就是调用父类中的属性或方法。将程序 TestPersonStudentDemo3.java 作相应的修改就形成了范例 TestPersonStudentDemo4.java，如下所示：

范例：TestPersonStudentDemo4.java

```
01  class Person
02  {
03      String name ;
04      int age ;
05      // 父类的构造方法
06      public Person(String name,int age)
07      {
08          this.name = name ;
09          this.age = age ;

10      }
11  }
12  class Student extends Person
```

```
13  {
14      String school ;
15      // 子类的构造方法
16      public Student()
17      {
18          // 在这里用 super 调用父类中的构造方法
19          super("张三",25);
20      }
21  }
22
23  public class TestPersonStudentDemo4
24  {
25      public static void main(String[] args)
26      {
27          Student s = new Student() ;
28          // 为 Student 类中的 school 赋值
29          s.school = "北京" ;
30          System.out.println("姓名："+s.name+"，年龄："+s.age+"，学校："+s.school) ;
31      }
32  }
```

输出结果：

姓名：张三，年龄：25，学校：北京

程序说明：

1、 程序第 1~11 行声明一名为 Person 的类，里面有 name 与 age 两个属性，并声明了一个含有两个参数的构造方法。

2、 程序第 12~21 行，声明一个名为 Student 的类，此类继承自 Person 类，在 16~20 行声明一子类的构造方法，在此方法中用 super("张三",25)调用父类中有两个参数的构造方法。

3、 程序在第 27 行声明并实例化一 Student 类的对象 s。29 行为 Student 对象 s 中的 school 赋值为“北京”。

读者可以发现，本例与范例 TestPersonStudentDemo3.java 的程序基本上是一样的，唯一的不同是在子类的构造方法中明确的指明调用的是父类中有两个参数的构造方法，所以程序在编译时不再去找父类中无参的构造方法。

注意：

用 super 调用父类中的构造方法，只能放在程序的第一行。

super 关键字不仅可以调用父类中的构造方法，也可以调用父类中的属性或方法，如格式 6-2 所示：

【 格式 6-2 super 调用属性或方法】

```
super.父类中的属性 ；
super.父类中的方法() ;
```

范例：TestPersonStudentDemo5.java

```
01  class Person
02  {
03      String name ;
04      int age ;
05      // 父类的构造方法
06      public Person()
07      {
08      }
09      public String talk()
10      {
11          return "我是："+this.name+"，今年："+this.age+"岁" ;
12      }
13  }
14  class Student extends Person
15  {

16      String school ;
17      // 子类的构造方法
18      public Student(String name,int age,String school)
```

```
19          {
20              // 在这里用 super 调用父类中的属性
21              super.name = name ;
22              super.age = age ;
23
24              // 调用父类中的 talk()方法
25              System.out.print(super.talk());
26
27              // 调用本类中的 school 属性
28              this.school = school ;
29          }
30      }
31
32      public class TestPersonStudentDemo5
33      {
34          public static void main(String[] args)
35          {
36              Student s = new Student("张三",25,"北京") ;
37              System.out.println("，学校："+s.school) ;
38          }
39      }
```

输出结果：

姓名：张三，年龄：25，学校：北京

程序说明：

1、 程序第 1~13 行声明一个名为 Person 的类，并声明 name 与 age 两个属性、一个返回 String 类型的 talk()方法，以及一个无参构造方法。

2、 程序第 14~30 行声明一个名为 Student 的类，此类直接继承自 Person 类。

3、 程序第 21、22 行，通过 super.属性的方式调用父类中的 name 与 age 属性，并分别赋值。

4、 程序第 25 行调用父类中的 talk()方法并打印信息。

从上面的程序中可以发现，子类 Student 可以通过 super 调用父类中的属性或方法，但是细心的读者在本例中可以发现，如果程序第 21、22、25 行换成 this 调用也是可以

的，那为什么还要用 super 呢？读者看完下面的内容就可以清楚的知道什么时候该这样使用了。

6.1.3 限制子类的访问

有些时候，父类并不希望子类可以访问自己的类中的全部的属性或方法，所以需要将一些属性与方法隐藏起来，不让子类去使用，所以在声明属性或方法时往往加上“private”关键字，表示私有。

范例：TestPersonStudentDemo6.java

```
01  class Person
02  {
03      // 在这里将属性封装
04      private String name ;
05      private int age ;
06  }
07  class Student extends Person
08  {
09      // 在这里访问父类中被封装的属性
10      public void setVar()
11      {
12          name = "张三" ;
13          age = 25 ;
14      }
15  }
16
17  class TestPersonStudentDemo6
18  {
19      public static void main(String[] args)
20      {
21          new Student().setVar() ;

22      }
23  }
```

编译结果：

```
TestPersonStudentDemo6.java:12: name has private access in Person
                name = "张三" ;
                ^
TestPersonStudentDemo6.java:13: age has private access in Person
                age = 25 ;
                ^
2 errors
```

由编译器的错误提示可以发现，name 与 age 属性在子类中无法进行访问。

小提示:

上面所示范例，读者可能会有这样的印象：只要父类中的属性被“private”声明的话，那么子类就再也无法访问到它了。实际上并不是这样的，在父类中加入了 private 关键字修饰，其目的只是相当于对子类隐藏了此属性，子类无法去显式的调用这些属性，但是却可以隐式地去调用，请看下面的范例，下面的范例修改自范例 TestPersonStudentDemo6.java

范例：TestPersonStudentDemo7.java

```
class Person
{
    // 在这里将属性封装
    private String name ;
    private int age ;
    // 添加了两个 setXxx()方法
    public void setName(String name)
    {
        this.name = name ;
    }
    public void setAge(int age)
    {

        this.age = age ;
    }
    public String talk()
```

```
    {
        return "我是："+this.name+"，今年："+this.age+"岁！"；
    }
}
class Student extends Person
{
    // 在这里访问父类中被封装的属性
    public void setVar()
    {
        super.setName("张三") ;
        super.setAge(25) ;
    }
}

class TestPersonStudentDemo7
{
    public static void main(String[] args)
    {
        Student s = new Student() ;
        s.setVar() ;
        System.out.println(s.talk()) ;
    }
}
```

输出结果：

```
我是：张三，今年：25 岁！
```

从上面程序中可以发现，虽然在子类中并没有 name 与 age 两个属性，但是子类对象依然可以去调用父类中的这两个属性，并打印输出，所以可以得出结论——子类在继承父类时，会继承父类中的全部的属性与方法。

6.1.4 复写

“复写”的概念与“重载”相似，它们均是 Java“多态”的技术之一，所谓“重载”，即是方法名称相同，但却可在不同的场合做不同的事。当一个子类继承一父类，而子类中的方法与父类中的方法的名称，参数个数、类型都完全一致时，就称子类中的这个方法复写了父类中的方法。同理，如果子类中重复定义了父类中已有的属性，则称此子类中的属性复写了父类中的属性。

【 格式 6-3 方法的复写】

```
class Super
{
    访问权限 方法返回值类型 方法 1（参数 1）
    {}
}
class Sub extends Super
{
    访问权限 方法返回值类型 方法 1（参数 1）——►  复写父类中的方法
    {}
}
```

范例：TestOverDemo1.java

```
01  class Person
02  {
03          String name ;
04          int age ;
05          public String talk()
06          {
07                  return "我是："+this.name+"，今年："+this.age+"岁";
08          }
09  }
10
11  class Student extends Person
12  {
```

```
13          String school ;
14          public Student(String name,int age,String school)
15          {
16              // 分别为属性赋值
17              this.name = name ;
18              this.age = age ;
19              this.school = school ;
20          }
21          // 此处复写 Person 中的 talk()方法
22          public String talk()
23          {
24              return "我在"+this.school+"上学" ;
25          }
26      }
27
28      class TestOverDemo1
29      {
30          public static void main(String[] args)
31          {
32              Student s = new Student("张三",25,"北京") ;
33              // 此时调用的是子类中的 talk()方法
34              System.out.println(s.talk()) ;
35          }
36      }
```

输出结果：

我在北京上学

程序说明：

1、 程序 1~9 行声明一个名为 Person 的类，里面声明了 name 与 age 两个属性，和一个 talk()方法。

2、 程序 11~26 行声明一个名为 Student 的类，此类继承自 Person 类，也就继承了 name 与 age 属性，同时声明了一个与父类中同名的 talk()方法，也可以说此时 Student 类中的 talk()方法复写了 Person 类中的 talk()方法。

3、 程序第 32 行实例化一子类对象，并同时调用子类构造方法为属性赋初值。

4、 程序第 34 行用子类对象调用 talk()方法，但此时调用的是子类中的 talk()方法。

由输出结果可以发现，在子类中复写了父类中的 talk()方法，所以子类对象在调用 talk()方法时，实际上调用的是子类中被复写好了的方法。另外可以发现，子类的 talk()方法与父类的 talk()方法，在声明权限时，都声明为 public，也就是说这两个方法的访问权限都是一样的。

注意：

子类复写父类中的方法时，被子类复写的方法不能拥有比父类中更严格的访问权限（关于访问权限的概念，后面会有更详细介绍），即：

```
class Person
{
    public String talk ()
    {}
}
class Student extends Person
{
    // 此处会因为权限出现错误
    String talk ()
    {}
}
```

在子类中 talk()方法处并没有声明权限，如果不声明则权限为 default，但父类中的 talk 方法有 public，而 public 权限要高于 default 权限，所以此时子类的方法比父类中拥有更严格的访问权限，所以会出现错误。

由 TestOverDemo1.java 程序中可以发现，在程序 34 行调用 talk()方法实际上调用的只是子类的方法，那如果现在需要调用父类中的方法该如何实现呢？请看下面的范例，下面的范例修改自 TestOverDemo1.java。

范例：TestOverDemo2.java

```
01  class Person
02  {
03      String name ;
04      int age ;
05      public String talk()
06      {
07          return "我是："+this.name+"，今年："+this.age+"岁";
08      }
09  }
10
11  class Student extends Person
12  {
13      String school ;
14      public Student(String name,int age,String school)
15      {
16          // 分别为属性赋值
17          this.name = name ;
18          this.age = age ;
19          this.school = school ;
20      }
21      // 此处复写 Person 类中的 talk()方法
22      public String talk()
23      {
24          return super.talk()+"，我在"+this.school+"上学"；
25      }
26  }
27
28  class TestOverDemo2
29  {
30      public static void main(String[] args)
31      {
32          Student s = new Student("张三",25,"北京") ;
33          //此时调用的是子类中的 talk()方法
34          System.out.println(s.talk()) ;
35      }
36  }
```

输出结果：

我是：张三，今年：25 岁，我在北京上学

程序说明：

1、 程序 1~9 行声明一 Person 类，里面声明了 name 与 age 两个属性，和一个 talk()方法。

2、 程序 11~26 行声明一 Student 类，此类继承自 Person，也就继承了 name 与 age 属性，同时声明了一个与父类中同名的 talk()方法，也可以说此时 Student 类中的 talk()方法复写了 Person 类中的 talk()方法，但在 24 行通过 super.talk()方式，调用父类中的 talk()方法。

3、 程序第 32 行实例化一子类对象，并同时调用子类构造方法为属性赋初值。

4、 程序第 34 行用子类对象调用 talk()方法，但此时调用的是子类中的 talk()方法。

由上面程序可以发现，在子类可以通过 super.方法()的方式调用父类中被子类复写的方法。

小提示：

关于 this 与 super 关键字的使用，对于一些初学者来说可能有些混淆，表 6-1 对 this 与 super 的差别进行比较，如下所示：

表 6-1　this 与 super 的比较

关键字	说明
this	1、表示当前对象
	2、调用本类中的方法或属性
	3、调用本类中的构造方法时，放在程序首行
super	1、子类调用父类的方法或属性
	2、调用父类中构造方法时，放在程序首行

读者从上表中不难发现，用 super 或 this 调用构造方法时都需要放在首行，所以，super 与 this 调用构造方法的操作是不能同时出现的。

6.2 抽象类

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。通过这种机制，派生出的新的类不仅可以保留原有的类的功能，而且还可以拥有更多的功能。

除了上述的机制之外，Java 也可以创建一种类专门用来当作父类，这种类称为“抽象类”。抽象类的作用有点类似“模版”，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。

抽象类定义规则

- 抽象类和抽象方法都必须用 abstract 关键字来修饰。
- 抽象类不能被实例化，也就是不能用 new 关键字去产生对象。
- 抽象方法只需声明，而不需实现。
- 含有抽象方法的类必须被声明为抽象类，抽象类的子类必须复写所有的抽象方法后才能被实例化，否则这个子类还是个抽象类。

【 格式 6-4 抽象类的定义格式】

```
abstract class 类名称   // 定义抽象类
{
    声明数据成员 ;

    访问权限 返回值的数据类型 方法名称（参数…）
    {                                                    } 定义一般方法
        …
    }
    abstract 返回值的数据类型 方法名称（参数…）;
      // 定义抽象方法，在抽象方法里，没有定义方法体
}
```

注意：

在抽象类定义的语法中，方法的定义可分为两种：一种是一般的方法，它和先前介绍过的方法没有什么两样；另一种是“抽象方法”，它是以 abstract 关键字为开头的方法，此方法只声明了返回值的数据类型、方法名称与所需的参数，但没有定义方法体。

范例：TestAbstractDemo1.java

```
01  abstract class Person
02  {
03      String name ;
04      int age ;
05      String occupation ;
06      // 声明一抽象方法 talk()
07      public abstract String talk() ;
08  }
09  // Student 类继承自 Person 类
10  class Student extends Person
11  {
12      public Student(String name,int age,String occupation)
13      {
14          this.name = name ;
15          this.age = age ;
16          this.occupation = occupation ;
17      }
18      // 复写 talk()方法
19      public String talk()
20      {
21          return "学生——>姓名："+this.name+"，年龄："+this.age+"，职业：
22                  "+this.occupation+"！" ;
23      }
24  }
25  // Worker 类继承自 Person 类
26  class Worker extends Person
27  {
28      public Worker(String name,int age,String occupation)
```

```
29          {
30                  this.name = name ;
31              this.age = age ;
32              this.occupation = occupation ;
33          }
34          // 复写 talk()方法
35          public String talk()
36          {
37              return "工人——>姓名："+this.name+"，年龄："+this.age+"，职业：
38              "+this.occupation+"！" ;
39          }
40      }
41      class TestAbstractDemo1
42      {
43              public static void main(String[] args)
44              {
45              Student s = new Student("张三",20,"学生") ;
46              Worker w = new Worker("李四",30,"工人") ;
47              System.out.println(s.talk()) ;
48              System.out.println(w.talk()) ;
49          }
50      }
```

输出结果：

学生——>姓名：张三，年龄：20，职业：学生！

工人——>姓名：李四，年龄：30，职业：工人！

程序说明：

1、 程序 1~8 行声明一名为 Person 的抽象类，在 Person 中声明了三个属性，和一个抽象方法——talk()。

2、 程序 9~24 行声明一 Student 类，此类继承自 Person 类，而且此类不为抽象类，所以需要复写 Person 类中的抽象方法——talk()。

3、 程序 25~40 行声明一 Worker 类，此类继承自 Person 类，而且此类不为抽象类，所以需要复写 Person 类中的抽象方法——talk()。

4、 程序第 45、46 行分别实例化 Student 类与 Worker 类的对象，并调用各自构造方法初始化类属性。

5、 程序 47、48 行分别调用各自类中被复写的 talk()方法。

可以发现两个子类 Student、Worker 都分别按各自的要求复写了 talk()方法。上面的程序可由图 6-6 表示，如下所示：

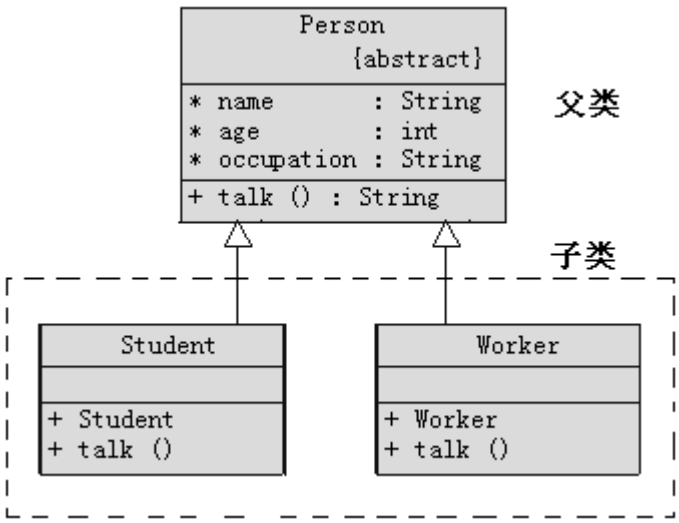


图 6-6 抽象类的继承关系

小提示:

与一般类相同，在抽象类中，也可以拥有构造方法，但是这些构造方法必须在子类中被调用。

```
abstract class Person
{
    String name ;
    int age ;
    String occupation ;
    public Person(String name,int age,String occupation)
    {
        this.name = name ;
```

```
            this.age = age ;
            this.occupation = occupation ;
        }
        public abstract String talk() ;
    }

    class Student extends Person
    {
        public Student(String name,int age,String occupation)
        {
            // 在这里必须明确调用抽象类中的构造方法
            super(name,age,occupation);
        }
        public String talk() {
            return "学生——>姓名："+this.name+"，年龄："+this.age+"，职业："
"+this.occupation+"！";
        }
    }
    class   TestAbstractDemo2
    {
        public static void main(String[] args)
        {
            Student s = new Student("张三",20,"学生") ;
            System.out.println(s.talk()) ;
        }
    }
```

输出结果：

学生——>姓名：张三，年龄：20，职业：学生！

从上面的程序中可以发现，抽象类也可以像普通类一样，有构造方法、一般方法、属性，更重要的是还可以有一些抽象方法，留给子类去实现，而且在抽象类中声明构造方法后，在子类中**必须**明确调用。

6.3 Object 类

Java 中有一个比较特殊的类，就是 Object 类，它是所有类的父类，如果一个类没有使用 extends 关键字明确标识继承另外一个类，那么这个类就默认继承 Object 类。因此，Object 类是 Java 类层中的最高层类，是所有类的超类。换句话说，Java 中任何一个类都是它的子类。由于所有的类都是由 Object 类衍生出来的，所以 Oject 类中的方法适用于所有类。

```
public class Person    // 当没有指定父类时，会默认 Object 类为其父类
{
    …
}
```

上面的程序等价于：

```
public class Person extends Object
{
   …
}
```

读者查一下 JDK 的帮助文档，可以发现在 Object 类的方法中有一个 toString()方法。

此方法是在打印对象时被调用的，下面有两个范例，一个是没复写了 toString()方法，另一个是复写了 toString()方法，读者可比较两者的区别。

范例：TestToStringDemo1.java

```
01  class Person extends Object
02  {
03      String name = "张三";
04      int age = 25 ;
05  }
06  class TestToStringDemo1
07  {
```

```
08          public static void main(String[] args)
09          {
10                  Person p = new Person() ;
11                  System.out.println(p);
12          }
13  }
```

输出结果：

```
Person@1ea2dfe
```

程序说明：

1、 程序 1~5 行声明一名为 Person 的类，并明确指出继承自 Object 类。

2、 程序第 10 行声明并实例化一 Person 类的对象 p，11 行打印对象。

从上面的程序中可以发现，在打印对象 p 的时候实际上打印出的是一些无序的字符串，这样的字符串很少有人能看懂什么意思，之后可以再观察下面的范例，下面的范例复写了 Object 类中的 toString()方法。

范例：TestToStringDemo2.java

```
01  class Person extends Object
02  {
03          String name = "张三";
04          int age = 25 ;
05          // 复写 Object 类中的 toString()方法
06          public String toString()
07          {
08                  return "我是："+this.name+"，今年："+this.age+"岁";
09          }
10  }
11  class TestToStringDemo2
12  {
13          public static void main(String[] args)
14          {
15                  Person p = new Person() ;
```

```
16              System.out.println(p);
17      }
18  }
```

输出结果：

我是：张三，今年：25 岁

与 TestToStringDemo1.java 程序相比，程序 TestToStringDemo2.java 程序在 Person 类中明确复写了 toString()方法，这样在打印对象 p 的时候，实际上是去调用了 toString()方法，只是并没有明显的指明调用 toString()方法而已，此时第 16 行相当于：

System.out.println(p.toString());

6.4 final 关键字

在 Java 中声明类、属性和方法时，可使用关键字 final 来修饰。

1、 final 标记的类不能被继承。

2、 final 标记的方法不能被子类复写。

3、 final 标记的变量（成员变量或局部变量）即为常量，只能赋值一次。

范例：TestFinalDemo1.java

```
01  class TestFinalDemo1
02  {
03      public static void main(String[] args)
04      {
05          final int i = 10 ;
06          // 修改用 final 修饰的变量 i
07          i++ ;
08      }
09  }
```

编译结果：

```
TestFinalDemo1.java:6: cannot assign a value to final variable i
                        i++ ;
                        ^
1 error
```

范例：TestFinalDemo2.java

```
01  final class Person
02  {
03  }
04  class Student extends Person
05  {
06  }
```

编译结果：

```
TestFinalDemo2.java:4: cannot inherit from final Person
class Student extends Person
                      ^
1 error
```

范例：TestFinalDemo3.java

```
01  class Person
02  {
03      // 此方法声明为 final 不能被子类复写
04      final public String talk()
05      {
06          return "Person：talk()" ;
07      }
08  }
09
10  class Student extends Person
11  {
12      public String talk()
13      {
14          return "Student：talk()" ;
15      }
```

```
16  }
```

编译结果：

```
TestFinalDemo3.java:12: talk() in Student cannot override talk() in Person; overridden method is final
        public String talk()
                      ^
1 error
```

6.5 接口（interface）

接口（interface）是 Java 所提供的另一种重要技术，它的结构和抽象类非常相似，也具有数据成员与抽象方法，但它与抽象类又有以下两点不同：

1、 接口里的数据成员必须初始化，且数据成员均为常量。

2、 接口里的方法必须全部声明为 abstract，也就是说，接口不能像抽象类一样保有一般的方法，而必须全部是“抽象方法”。

接口定义的语法如下：

【 格式 6-5 接口的定义格式】

```
interface 接口名称       // 定义抽象类
{
      final 数据类型 成员名称 = 常量 ;          // 数据成员必须赋初值

      abstract 返回值的数据类型 方法名称（参数…）;
      // 抽象方法，注意在抽象方法里，没有定义方法主体。
}
```

接口与一般类一样，本身也具有数据成员与方法，但数据成员一定要赋初值，且此值将不能再更改，方法也必须是“抽象方法”。也正因为方法必须是抽象方法，而没有一般的方法，所以格式 6-5 中，抽象方法声明的关键字 abstract 是可以省略的。相

同的情况也发生在数据成员身上，因数据成员必须赋初值，且此值不能再被更改，所以声明数据成员的关键字 final 也可省略。事实上只要记得：（1）、接口里的“抽象方法”只要做声明即可，而不用定义其处理的方式；（2）、数据成员必须赋初值，这样就可以了。

在 java 中接口是用于实现多继承的一种机制，也是 java 设计中最重要的一个环节，每一个由接口实现的类必须在类内部复写接口中的抽象方法，且可自由地使用接口中的常量。

既然接口里只有抽象方法，它只要声明而不用定义处理方式，于是自然可以联想到接口也没有办法像一般类一样，再用它来创建对象。利用接口打造新的类的过程，称之为接口的实现（implementation）。

格式 6-6 为接口实现的语法：

【 格式 6-6 接口的实现】

```
class 类名称 implements 接口 A,接口 B     // 接口的实现
{
   …
}
```

下面的范例修改自范例 TestAbstractDemo1. java：

范例：TestInterfaceDemo1.java

```
01  interface Person
02  {
03        String name = "张三";
04        int age = 25 ;
05        String occupation = "学生" ;
06        // 声明一抽象方法 talk()
07        public abstract String talk() ;
08  }
09  //   Student 类继承自 Person 类
10  class Student implements Person
11  {
12        // 复写 talk()方法
```

```
13          public String talk()
14          {
15                  return "学生——>姓名："+this.name+"，年龄："+this.age+"，职业：
16                  "+this.occupation+"！" ;
17           }
18  }
19  class TestInterfaceDemo1
20  {
21          public static void main(String[] args)
22          {
23                  Student s = new Student() ;
24                  System.out.println(s.talk()) ;
25          }
26  }
```

输出结果：

学生——>姓名：张三，年龄：25，职业：学生！

程序说明：

1、 程序 1~8 行，声明一 Person 接口，并在里面声明了三个常量：name、age、occupation，并分别赋值。

2、 程序 10~18 行声明一 Student 类，此类实现 Person 接口，并复写 Person 中的 talk()方法。

3、 程序第 23 行实例化一 Student 的对象 s，并在第 24 行调用 talk()方法，打印信息。

有些读者可能会觉得这样做与抽象类并没有什么不同，在这里需要再次提醒读者的是，接口是 java 实现多继承的一种机制，一个类只能继承一个父类，但如果需要一个类继承多个抽象方法的话，就明显无法实现，所以就出现了接口的概念。一个类只可以继承一个父类，但却可以实现多个接口。

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，同时也可加入新的成员以满足实际的需要。

同样的，接口的扩展（或继承）也是通过关键字 extends 来实现的。有趣的是，一个接口可以继承多个接口，这点与类的继承有所不同。格式 6-7 是接口扩展的语法：

【 格式 6-7 接口的扩展】

```
interface 子接口名称 extends 父接口 1，父接口 2，…
{
    … …
}
```

范例：TestInterfaceDemo2.java

```
01  interface A
02  {
03      int i = 10 ;
04      public void sayI() ;
05  }
06  interface E
07  {
08      int x = 40 ;
09      public void sayE() ;
10  }
11  // B 同时继承了 A、E 两个接口
12  interface B extends A,E
13  {
14      int j = 20 ;
15      public void sayJ() ;
16  }
17
18  // C 继承实现 B 接口，也就意味着要实现 A、B、E 三个接口的抽象方法
19  class C implements B
20  {
21      public void sayI()
22      {
23          System.out.println("i = "+i);
24      }
25      public void sayJ()
```

```
26          {
27                  System.out.println("j = "+j) ;
28          }
29          public void sayE()
30          {
31                  System.out.println("e = "+x);
32          }
33  }
34  class TestInterfaceDemo2
35  {
36          public static void main(String[] args)
37          {
38                  C c = new C() ;
39                  c.sayI() ;
40                  c.sayJ() ;
41          }
42  }
```

输出结果：

i = 10

j = 20

程序说明：

1、 程序 1~5 行声明一接口 A，并声明一常量 i 和一抽象方法 sayI()。

2、 程序 6~10 行声明一接口 E，并声明一常量 x 和一抽象方法 sayE()。

3、 程序 12~16 行声明一接口 B，此接口同时继承 A、E 接口，同时声明一常量 j 和一抽象方法 sayJ()。

4、 程序 19~33 行声明一类 C，此类实现了 B 接口，所以此类中要复写接口 A、B、E 中的全部抽象方法。

从此程序中读者可以发现与类的继承不同的是，一个接口可以同时继承多个接口，也就是同时继承了多个接口的抽象方法与常量。

6.6 对象多态性

在前面已经将面向对象的封装性、继承性详细地解释给读者了，下面就来看一下面向对象中最后一个，也是最重要的一个特性——多态性。

那什么叫多态性呢？读者应该还清楚在之前曾解释过重载的概念，重载的最终效果就是调用同一个方法名称，却可以根据传入参数的不同而得到不同的处理结果，这其实就是多态性的一种体现。

下面用一道范例为读者简单介绍一下多态多态的概念，请看下面的范例：

范例：TestJavaDemo1.java

```
01  class Person
02  {
03      public void fun1()
04      {
05          System.out.println("1.Person{fun1()}") ;
06      }
07      public void fun2()
08      {
09          System.out.println("2.Person{fun2()}") ;
10      }
11  }
12
13  // Student 类扩展自 Person 类，也就继承了 Person 类中的 fun1()、fun2()方法
14  class Student extends Person
15  {
16      // 在这里复写了 Person 类中的 fun1()方法
17      public void fun1()
18      {
19          System.out.println("3.Student{fun1()}") ;
20      }
21      public void fun3()
22      {
23          System.out.println("4.Studen{fun3()}") ;
```

```
24          }
25      }
26
27      class TestJavaDemo1
28      {
29          public static void main(String[] args)
30          {
31              // 此处，父类对象由子类实例化
32              Person p = new Student() ;
33              // 调用 fun1()方法，观察此处调用的是哪个类里的 fun1()方法
34              p.fun1() ;
35              p.fun2() ;
36          }
37      }
```

输出结果：

3.Student{fun1()}

2.Person{fun2()}

程序说明：

1、 程序 1~11 行声明一 Person 类，此类中有 fun1()、fun2()两个方法。

2、 程序 14~25 行声明一 Student 类，此类继承自 Person 类，并复写了 fun1()方法。

3、 程序第 32 行声明一 Person 类（父类）的对象，之后由子类对象去实例化此对象。

4、 第 34 行由父类对象调用 fun1()方法。

从程序的输出结果中可以发现，p 是父类的对象，但调用 fun1()方法的时候并没有调用其本身的 fun1()方法，而是调用了子类中被复写了的 fun1()方法。之所以会产生这样的结果，最根本的原因就是因为父类对象并非由其本身的类实例化，而是通过子类实例化，这就是所谓的对象的多态性，即子类实例化对象可以转换为父类实例化对象。

在这里要着重讲解两个概念，希望读者加以重视：

1、 向上转型：

在上面范例 TestJavaDemo1.java 中，父类对象通过子类对象去实例化，实际

上就是对象的向上转型。向上转型是不需要进行强制类型转换的，但是向上转型会丢失精度。

2、 向下转型：

与向上转型对应的一个概念就是“向下转型”，所谓向下转型，也就是说父类的对象可以转换为子类对象，但是需要注意的是，这时则必须要进行强制的类型转换。

读者可能觉得上面的两个概念有些难以理解，下面举个例子来帮助读者理解：

有个小孩在马路上看见了一辆跑车，他指着跑车说那是汽车。相信读者都会认为这句话没有错，跑车的确是符合汽车的标准，所以把跑车说成汽车并没有错误，只是不准确而已。不管是小轿车也好，货车也好，其实都是汽车，这在现实生活中是说得通的，在这里读者可以将这些小轿、货车都想象成汽车的子类，它们都是扩展了汽车的功能，都具备了汽车的功能，所以它们都可以叫做汽车，那么这种概念就称为向上转型。而相反，假如说把所有的汽车都当成跑车，那结果肯定是不正确的了，因为汽车有很多种，必须明确的指明是哪辆跑车才可以，需要加一些限制，这个时候就必须明确的指明是哪辆车，所以需要进行强制的说明。

上面的解释可以概括成两句话：

一、向上转型可以自动完成；

二、向下转型必须进行强制类型转换。

小提示：

另外，要提醒读者注意的是，并非全部的父类对象都可以强制转换为子类对象，请看下面的范例：

范例：TestJavaDemo2.java

```
01  class Person
02  {
```

```
03          public void fun1()
04          {
05              System.out.println("1.Person{fun1()}") ;
06          }
07          public void fun2()
08          {
09              System.out.println("2.Person{fun2()}") ;
10          }
11      }
12
13  // Student 类继承 Person 类，也就继承了 Person 类的 fun1()、fun2()方法
14      class Student extends Person
15      {
16          // 在这里复写了 Person 类中的 fun1()方法
17          public void fun1()
18          {
19              System.out.println("3.Student{fun1()}") ;
20          }
21          public void fun3()
22          {
23              System.out.println("4.Studen{fun3()}") ;
24          }
25      }
26
27      class TestJavaDemo2
28      {
29          public static void main(String[] args)
30          {
31              // 此处，父类对象由自身实例化
32              Person p = new Person() ;
33              // 将 p 对象向下转型
34              Student s = (Student)p ;
35              p.fun1() ;
36              p.fun2() ;
37          }
38      }
```

运行结果：

Exception in thread "main" java.lang.ClassCastException

at TestJavaDemo2.main(TestJavaDemo2.java:34)

由程序可以发现,程序 32 行 Person 对象 p 是由 Person 类本身实例化的，在第 34 行将 Person 对象 p 强制转换为子类对象，这样写在语法上是没有任何错误的，但是在运行时可以发现出了异常，这是为什么呢？为什么父类不可以向子类转换了呢？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，假如你今天刚买完一些生活用品，回家的时候在路上碰见一个孩子，这个孩子忽然对你说：“我是你的儿子，你把你的东西给我吧!”，这个时候你肯定不会把你的东西给这个孩子，因为你不确定他跟你是否有关系，怎么能给呢？那么在这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，那肯定在转换的时候会出现错误，那么这个错误该如何纠正呢？只需要将第 32 行的代码修改成如下形式即可：

Person p = new Student() ;

这个时候相当于是由子类去实例化父类对象，也就是说这个时候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的时候就不会再有问题了。

6.6.1 instanceof 关键字的使用

可以用 instanceof 判断一个类是否实现了某个接口，也可以用它来判断一个实例对象是否属于一个类。instanceof 的语法格式为：

对象 instanceof 类(或接口)

它的返回值是布尔型的，或真(true)、或假(false)。

将上面的范例的程序进行简单的修改就形成了范例 TestJavaDemo3.java，如下所示：

范例：TestJavaDemo3.java

```
01  class Person
02  {
03      public void fun1()
04      {
05          System.out.println("1.Person{fun1()}") ;
06      }
07      public void fun2()
08      {
09          System.out.println("2.Person{fun2()}") ;
10      }
11  }
12
13  // Student 类继承自 Person 类，也就继承了 Person 类中的 fun1()、fun2()方法
14  class Student extends Person
15  {
16      // 在这里复写了 Person 类中的 fun1()方法
17      public void fun1()
18      {
19          System.out.println("3.Student{fun1()}") ;
20      }
21      public void fun3()
22      {
23          System.out.println("4.Studen{fun3()}") ;
24      }
25  }
26  class TestJavaDemo3
27  {
28      public static void main(String[] args)
29      {
30          // 声明一父类对象并通过子类对象对其进行实例化
31          Person p = new Student() ;
32          // 判断对象 p 是否是 Student 类的实例
33          if(p instanceof Student)
34          {
35              // 将 Person 类的对象 p 转型为 Student 类型
36              Student s = (Student)p ;
```

```
37                      s.fun1() ;
38              }
39              else
40              {
41                      p.fun2() ;
42              }
43      }
44  }
```

输出结果：

3.Student{fun1()}

程序说明：

1、 程序第 31 行声明一父类对象 p，并通过其子类实例化此对象。

2、 程序第 33 行用 instanceof 关键字判断 p 对象是否是 Student 的实例，此范例中，因为 p 是通过 Student 类实例化的，所以此条件会满足，第 36 行将 p 对象强制转换为 Student 类的对象，并调用 fun1()方法，调用此方法时，实际上调用的是被子类复写了的 fun1()方法。

6.6.2 复写 Object 类中的 equals 方法

前面已经介绍过 Object 是所有类的父类，其中的 toString()方法是需要被复写的，如果读者去查 JDK 手册，会发现在 Object 类中有一个 equals 方法，此方法用于比较对象是否相等，而且此方法必须被复写，为什么要复写它呢？请看下面的范例，下面的范例是一个没有复写 equals()方法的范例。

范例：TestOverEquals.java

```
01  class Person
02  {
03          private String name ;
04          private int age ;
```

```
05        public Person(String name,int age)
06        {
07              this.name = name ;
08              this.age = age ;
09        }
10  }
11  class TestOverEquals1
12  {
13        public static void main(String[] args)
14        {
15              Person p1 = new Person("张三",25);
16              Person p2 = new Person("张三",25);
17              // 判断 p1 和 p2 的内容是否相等
18              System.out.println(p1.equals(p2)?"是同一个人！ ":"不是同一个人");
19        }
20  }
```

输出结果：

不是同一个人

程序说明：

1、 程序第 1~10 行声明一 Person 类，并声明一构造方法为类的属性初始化。

2、 程序第 15、16 行声明两个 Person 对象 p1、p2，其内容相等。

3、 程序第 18 行是比较两对象的内容是否相等。

从上面的程序中可以发现，两对象的内容完全相等，但为什么比较的结果是不相等呢？因为 p1 与 p2 的内容分别在不同的内存空间，指向了不同的内存地址，所以在用 equals 比较时，实际上是调用了 Object 类中的 equals 方法，但可以发现此方法并不好用，所以在开发中往往需要复写 equals 方法，请看下面的范例：

范例：TestOverEquals2.java

```
01  class Person
02  {
03        private String name ;
```

```
04        private int age ;
05        public Person(String name,int age)
06        {
07            this.name = name ;
08            this.age = age ;
09        }
10        // 复写父类（Object 类）中的 equals 方法
11        public boolean equals(Object o)
12        {
13            boolean temp = true ;
14            // 声明一 p1 对象，此对象实际上就是当前调用 equals 方法的对象
15            Person p1 = this ;
16            // 判断 Object 类对象是否是 Person 的实例
17            if(o instanceof Person)
18            {
19                // 如果是 Person 类实例，则进行向下转型
20                Person p2 = (Person)o ;
21                // 调用 String 类中的 equals 方法
22                if(!(p1.name.equals(p2.name)&&p1.age==p2.age))
23                {
24                    temp = false ;
25                }
26            }
27            else
28            {
29                // 如果不是 Person 类实例，则直接返回 false
30                temp = false ;
31            }
32            return temp ;
33        }
34    }
35    class TestOverEquals2
36    {
37        public static void main(String[] args)
38        {
39            Person t_p1 = new Person("张三",25);
40            Person t_p2 = new Person("张三",25);
```

```
41  // 判断 p1 和 p2 的内容是否相等，如果相等，则表示是一个人，反之，则不是
42              System.out.println(t_p1.equals(t_p2)?"是同一个人！":"不是同一个人");
43      }
44  }
```

输出结果：

是同一个人！

程序说明：

1、 程序 1~34 行声明一 Person 类，并在类中复写了 Object 类的 equals 方法。

2、 程序第 15 行声明一 Person 对象 p1，并用 this 实例化。此时，this 就相当于当前调用此方法的对象，也就是第 42 行的 t_p1 对象。

3、 第 17 行，判断传进去的实例对象 o 是否属于 Person 类的实例化对象，如果是，则进行转型，否则返回 false。

4、 程序第 22 行分别比较两个对象的内容是否相等，如果不相等，则返回 false 。

5、 第 42 行，通过 t_p1 调用 equals 方法，并将 t_p2 对象的实例传到 equals 方法之中，比较两对象是否相等。

6.6.3 接口对象的实例化

前面已经讲解了接口的概念，相信读者应该已经清楚了，接口是无法直接实例化的，因为接口中没有构造方法，但是却可以根据对象多态性的概念，通过接口的子类对其进行实例化，请看下面的范例：

范例：TestInterfaceObject

```
01  interface Person
02  {
03      public void fun1() ;
04  }
05  class Student implements Person
06  {
```

```
07          public void fun1()
08          {
09                  System.out.println("Student fun1()");
10          }
11  }
12  class TestInterfaceObject
13  {
14          public static void main(String[] args)
15          {
16                  Person p = new Student() ;
17                  p.fun1() ;
18          }
19  }
```

输出结果：

Student fun1()

程序说明：

1、 程序第 1~4 行声明一 Person 接口，此接口中只有一个抽象方法 fun1()。

2、 程序第 5~11 行声明一 Student 类，此类实现 Person 接口，并复写 fun1()方法。

3、 程序第 16 行声明一 Person 接口的对象 p，并通过其子类 Student 类去实例化此对象。

4、 程序第 17 行调用 fun1()方法，此时调用的是子类中复写了的 fun1()方法。

从上面的程序中可以发现，接口是可以被实例化的，但是不能被直接实例化，只能通过其子类进行实例化，而在这里将 Person 声明为抽象类道理也是一样的，这要留给读者自己去完成。

那么这样去实例化到底有什么好处呢？举一个例子来说明，读者应该知道现在已经有了很多的 USB 设备，无论是移动硬盘，还是 MP3，这些设备都具备一个重要的特点，就是都拥有一个 USB 接口，而电脑上也有相应的插槽，所以这些设备才能正常的使用。以此为例编写程序如下所示：

范例：TestInterface.java

```
01  interface Usb
02  {
03      public void start() ;
04      public void stop() ;
05  }
06  class MoveDisk implements Usb
07  {
08      public void start()
09      {
10          System.out.println("MoveDisk start...") ;
11      }
12      public void stop()
13      {
14          System.out.println("MoveDisk stop...") ;
15      }
16  }
17  class Mp3 implements Usb
18  {
19      public void start()
20      {
21          System.out.println("Mp3 start...") ;
22      }
23      public void stop()
24      {
25          System.out.println("Mp3 stop...") ;
26      }
27  }
28
29  class Computer
30  {
31      public void work(Usb u)
32      {
33          u.start() ;
34          u.stop() ;
35      }
36  }
```

```
37
38  class TestInterface
39  {
40          public static void main(String[] args)
41          {
42                  new Computer().work(new MoveDisk()) ;
43                  new Computer().work(new Mp3()) ;
44          }
45  }
```

输出结果：

```
MoveDisk start...
MoveDisk stop...
Mp3 start...
Mp3 stop...
```

程序说明：

1、 程序 1~5 行声明一 Usb 接口，此接口中有两个抽象方法：start()、stop()。

2、 程序 6~16 行声明一 MoveDisk 类，此类实现 Usb 接口，并复写了里面的两个抽象方法。

3、 程序 17~27 行声明一 Mp3 类，此类实现 Usb 接口，并复写了里面的两个抽象方法。

4、 程序 29~36 行声明一 Computer 类，在类中有一 work()方法，此方法接收 Usb 对象的实例，并调用 start、stop 方法。

5、 程序 42、43 行分别用 Computer 类的实例化对象调用 work()方法，并根据传进对象的不同而产生不同的结果。

从上面的程序中可以发现，使用接口实际上就是定义出了一个统一的标准，在后面的章节中，还会介绍接口的其它使用方法。

6.7 匿名内部类

在前面的章节中已经为读者介绍了内部类的概念，而之前所介绍的内部类只是单一的类，并没有继承或实现任何类或接口，当然也是可以继承一个抽象类或实现一个接口。

下面先来看一个比较简单的范例。

范例：TestNonameInner1.java

```
01  interface A
02  {
03      public void fun1() ;
04  }
05  class B
06  {
07      int i = 10 ;
08      class C implements A
09      {
10          public void fun1()
11          {
12              System.out.println(i) ;
13          }
14      }
15      public void get(A a)
16      {
17          a.fun1() ;
18      }
19      public void test()
20      {
21          this.get(new C()) ;
22      }
23  }
24  class TestNonameInner1
25  {
26      public static void main(String[] args)
```

```
27          {
28                  B b = new B() ;
29                  b.test() ;
30          }
31  }
```

输出结果：

```
10
```

程序说明：

1、 程序 1~4 行声明一 A 接口，并声明一 fun1()抽象方法。

2、 程序 5~23 行声明一类 B，此类中有一个内部类 C，和一个变量 i。

3、 程序 8~14 行在类 B 中声明一内部类 C，此类实现 A 接口，并复写了 fun1 方法。

4、 程序 15~18 行声明一 get()方法，此方法用于 A 接口对象的实例化，并调用 fun1()方法。

5、 程序 19~22 行声明一 test 方法，此方法用于调用 get()方法。

6、 程序第 28 行声明一 B 类的实例化对象 b，并在第 29 行调用 test()方法。

上面的程序有些读者可能会认为与以前的内部类没有太大的区别，也是在类内部声明了一个类，与原先不同的是多实现了一个接口罢了。的确，这里只是简单的声明了一个内部类，目的并不是要再重新介绍一遍内部类的概念，而是为了引出下面的概念——匿名内部类。

范例：TestNonameInner2.java

```
01  interface A
02  {
03          public void fun1() ;
04  }
05  class B
06  {
07          int i = 10 ;
08          public void get(A a)
09          {
```

```
10              a.fun1() ;
11          }
12          public void test()
13          {
14              this.get(new A()
15                  {
16                      public void fun1()
17                      {
18                          System.out.println(i) ;
19                      }
20                  }
21              ) ;
22          }
23      }
24      class TestNonameInner2
25      {
26          public static void main(String[] args)
27          {
28              B b = new B() ;
29              b.test() ;
30          }
31      }
```

输出结果：

```
10
```

由此程序可以发现，在程序中并没有明确的声明出实现接口 A 的类，但是在程序第 14~21 行，可以发现，程序实现了接口 A 中的 fun1()方法，并把整个的一个实现类，传递到了方法 get 中，这就是所谓的匿名内部类。它不用声明实质上的类，就可以使用，上面 14~21 行有些读者会觉得难以理解，下面将这段程序拆分后再进行说明：

```
get(new A()) ;
```

这段代码读者应该可以看明白，它的主要作用是传了一个 A 的匿名对象，之后，实现 A 接口里的 fun1 方法：

```
get(new A()
    {
```

```
        }

    ) ;

    get(new A()
        {
            public void fun1()
                {
                    System.out.println(i) ;
                }
        }
    ) ;
```

上面的程序就是匿名内部类，有些读者可能会觉得难以理解，但是这样的程序见多了、用多了也就可以慢慢理解了。

• 本章摘要：

1、通过 extends 关键字，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之前，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、 父类有数个构造方法时，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()这个关键字来完成。

4、 this()是在同一类内调用其它的构造方法，而 super()则是从子类的构造方法调用其父类的构造方法。

5、 this()除了可用来调用同一类内的其它构造方法之外，如果同一类内“实例变量”与局部（local）变量的名称相同时，也可利用它来调用同一类内的“实例变量”。

6、 this()与 super()其相似之处：（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一行，也正是这个原因，this()与 super()无法同时存在同一个构造方法内。

7、 “重载”（overloading），它是指在相同类内，定义名称相同，但参数个数或类型不同的方法，因此 Java 便可依据参数的个数或类型调用相应的方法。

8、 “复写”（overriding），它是在子类当中，定义名称、参数个数与类型均与父类相同的方法，用以复写父类里的方法。

9、 如果父类的方法不希望子类的方法来复写它，可在父类的方法之前加上“final”关键字，如此该方法便不会被复写。

10、final 的另一个功用是把它加在数据成员变量前面，如此该变量就变成了一个常量（constant），如此便无法在程序代码中再做修改了。

11、所有的类均继承自 Object 类。

12、复写 Object 类中的 equals() method 可用来比较两个类的对象是否相等。

13、Java 可以创建抽象类，专门用来当做父类。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。

14、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。

15、利用父类的变量数组来访问子类的内容的较好的做法是：

（1） 先创建父类的变量数组；

（2） 利用数组元素创建子类的对象，并以它来访问子类的内容。

16、抽象类不能直接用来产生对象。

17、接口的结构和抽象类非常相似，它也具有数据成员与抽象 method，但它与抽象类有两点不同：（1）、接口的数据成员必须初始化。（2）、接口里的方法必须全部都声明成 abstract。

18、利用接口的特性来打造一个新的类，称为接口的实现（implementation）。

19、Java 并不允许多重继承。

20、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；派生出的接口成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。

21、Java 对象的多态性分为：向上转型（自动）、向下转型（强制）。

22、通过 instanceof 关键字，可以判断对象属于那个类。

23、匿名内部类（anonymous inner class）的好处是可利用内部类创建不具有名称的对象，并利用它访问到类里的成员。

第七章　异常处理

即使在编译时没有错误信息产生，但在程序运行时，经常会出现一些运行时的错误，这种错误对 Java 而言是一种异常。有了异常就要有相应的处理方式。本章将介绍异常的基本概念以及相关的处理方式。

7.1　异常的基本概念

异常也称为例外，是在程序运行过程中发生的、会打断程序正常执行的事件，下面是几种常见的异常；

1、 算术异常（ArithmeticException）。

2、 没有给对象开辟内存空间时会出现空指针异常（NullPointerException）。

3、 找不到文件异常（FileNotFoundException）。

所以在程序设计时，必须考虑到可能发生的异常事件，并做出相应的处理。这样才能保证程序可以正常运行。

Java 的异常处理机制也秉承着面向对象的基本思想。在 Java 中，所有的异常都是以类的类型存在，除了内置的异常类之外，Java 也可以自定义的异常类。此外，Java 的异常处理机制也允许自定义抛出异常。关于这些概念，将在后面介绍。

7.1.1　为何需要异常处理？

在没有异常处理的语言中，就必须使用 if 或 switch 等语句，配合所想得到的错误状况来捕捉程序里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很多的 if 语句，有时候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。

7.1.2 简单的异常范例

Java 本身已有相当好的机制来处理异常的发生。本节先来看看 Java 是如何处理异常的。TestException7_1 是一个错误的程序，它在访问数组时，下标值已超过了数组下标所容许的最大值，因此会有异常发生：

范例：TestException7_1.java

```
01  public class TestException7_1
02  {
03      public static void main(String args[])
04      {
05          int arr[]=new int[5];                    // 容许 5 个元素
06          arr[10]=7;                               // 下标值超出所容许的范围
07          System.out.println("end of main() method !!");
08      }
09  }
```

在编译的时候程序不会发生任何错误，但是在执行到第 6 行时，会产生下列的错误信息：

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
at TestException7_1.main(TestException7_1.java:6)

错误的原因在于数组的下标值超出了最大允许的范围。Java 发现这个错误之后，便由系统抛出“ArrayIndexOutOfBoundsException”这个种类的异常，用来表示错误的原因，并停止运行程序。如果没有编写相应的处理异常的程序代码，则 Java 的默认异常处理机制会先抛出异常、然后停止程序运行。

7.1.3 异常的处理

TestException7_1 的异常发生后，Java 便把这个异常抛了出来，可是抛出来之后没有程序代码去捕捉它，所以程序到第 6 行便结束，因此根本不会执行到第 7 行。如果加上捕捉异常的程序代码，则可针对不同的异常做妥善的处理。这种处理的方式称为异常处理。

异常处理是由 try、catch 与 finally 三个关键字所组成的程序块，其语法如下：

【 格式 7-1 异常处理的语法】

```
try
{
    要检查的程序语句 ；                    } try 语句块
      …
}
catch(异常类  对象名称)
{                                          } catch 语句块
    异常发生时的处理语句 ；
}
finally
{                                          } finally 语句块
    一定会运行到的程序代码 ；
}
```

格式 7_1 的语法是依据下列的顺序来处理异常：

1、 try 程序块若是有异常发生时，程序的运行便中断，并抛出“异常类所产生的对象”。
2、 抛出的对象如果属于 catch()括号内欲捕获的异常类，则 catch 会捕捉此异常，然后进到 catch 的块里继续运行。
3、 无论 try 程序块是否有捕捉到异常，或者捕捉到的异常是否与 catch()括号里的异常相同，最后一定会运行 finally 块里的程序代码。

finally 的程序代码块运行结束后，程序再回到 try-catch-finally 块之后继续执行。

由上述的过程可知，异常捕捉的过程中做了两个判断：第一个是 try 程序块是否有异常产生，第二个是产生的异常是否和 catch()括号内欲捕捉的异常相同。

值得一提的是，finally 块是可以省略的。如果省略了 finally 块不写，则在 catch() 块运行结束后，程序跳到 try-cath 块之后继续执行。

根据这些基本概念与运行的步骤，可以绘制出如图 7-1 所示的流程图：

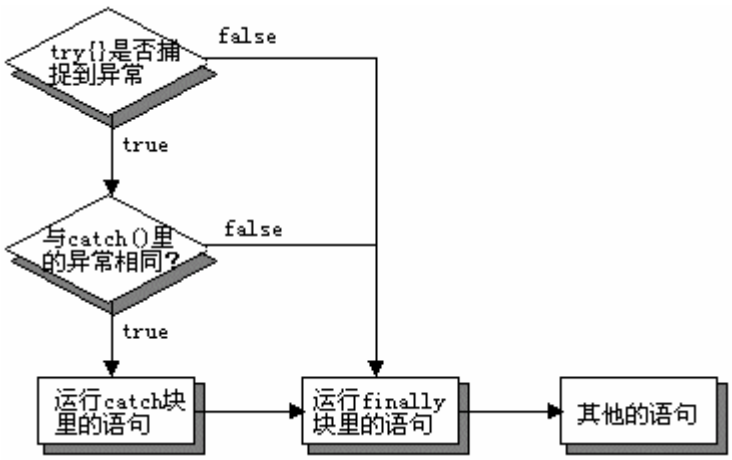


图 7-1　异常处理的流程图

在格式 7-1 中，“异常类”指的是由程序抛出的对象所属的类，例如 TestException7_1 中出现的“ArrayIndexOutOfBoundsException”就是属于异常类的一种。至于有哪些异常类以及它们之间的继承关系，稍后本书将会做更进一步的探讨。下面的程序代码加入了 try 与 catch，使得程序本身具有捕捉异常与处理异常的能力。

范例：TestException7_2.java

```
01  public class TestException7_2
02  {
03      public static void main(String args[])
04      {
05          try                    // 检查这个程序块的代码
06          {
```

```
07                  int arr[]=new int[5];
08                  arr[10]=7;              // 在这里会出现异常
09              }
10              catch(ArrayIndexOutOfBoundsException e)
11              {
12                  System.out.println("数组超出绑定范围！");
13              }
14              finally                     // 这个块的程序代码一定会执行
15              {
16                  System.out.println("这里一定会被执行！");
17              }
18              System.out.println("main()方法结束！");
19          }
20      }
```

输出结果：

数组超出绑定范围!

这里一定会被执行!

main()方法结束!

程序说明：

1、 程序第 7 行声明一 arr 的整型数组，并开辟了 5 个数据空间。

2、 程序第 8 行为数组中的第 10 个元素赋值，此时已经超出了该数组本身的范围，所以会出现异常。发生异常之后，程序语句转到 catch 语句中去处理，程序通过 finally 代码块统一结束。

上面程序的第 5~9 行的 try 块是用来检查是否会有异常发生。若有异常发生，且抛出的异常是属于 ArrayIndexOutOfBoundsException 类，则会运行第 10~13 行的代码块。因为第 8 行所抛出的异常正是 ArrayIndexOutOfBoundsException 类，因此第 12 行会输出“数组超出绑定范围!”字符串。由本例可看出，通过异常的机制，即使程序运行时发生问题，只要能捕捉到异常，程序便能顺利地运行到最后，且还能适时的加入对错误信息的提示。

程序 TestException7_2 里的第 10 行，如果程序捕捉到了异常，则在 catch 括号内的异常类 ArrayIndexOutOfBoundsException 之后生成一个对象 e，利用此对象可以得到异常的相关信息，下例说明了类对象 e 的应用：

范例：TestException7_3.java

```
01  public class TestException7_3
02  {
03      public static void main(String args[])
04      {
05          try
06          {
07              int arr[]=new int[5];
08              arr[10]=7;
09          }
10          catch(ArrayIndexOutOfBoundsException e){
11              System.out.println("数组超出绑定范围！");
12              System.out.println("异常："+e);     // 显示异常对象 e 的内容
13          }
14          System.out.println("main()方法结束！");
15      }
16  }
```

输出结果：

```
数组超出绑定范围!
异常：java.lang.ArrayIndexOutOfBoundsException: 10
main()方法结束!
```

例题 TestException7_3 省略了 finally 块，但程序依然可以运行。在第 10 行中，把 catch()括号内的内容想象成是方法的参数，而 e 就是 ArrayIndexOutOfBoundsException 类的对象。对象 e 接收到由异常类所产生的对象之后，就进到第 11 行，输出“数组超出绑定范围!”这一字符串，而第 12 行则是输出异常所属的种类，也就是 java.lang.ArrayIndexOutOfBoundsException。而 java.lang 正是 ArrayIndexOutOfBoundsException 类所属的包。

7.1.4 异常处理机制的回顾

当异常发生时，通常可以用两种方法来处理，一种是交由 Java 默认的异常处理机制做处理。但这种处理方式，Java 通常只能输出异常信息，接着便终止程序的运行。如 TestException7_1 的异常发生后，Java 默认的异常处理机制会显示出：

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
at TestException7_1.main(TestException7_1.java:6)

接着结束 TestException7_1 的运行。

另一种是处理方式是自行编写的 try-catch-finally 块来捕捉异常，如 TestException7_2 与 TestException7_3。自行编写程序代码来捕捉异常最大的好处是：可以灵活操控程序的流程，且可做出最适当的处理。图 7-2 绘出了异常处理机制的选择流程。

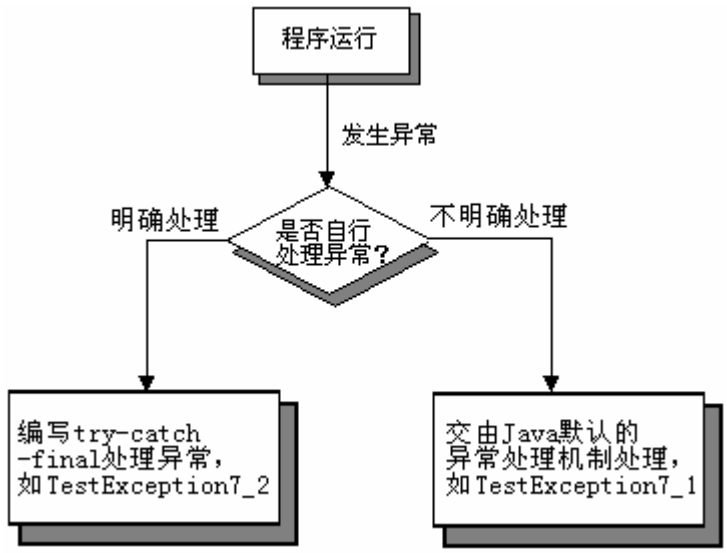


图 7-2　异常处理的方法

7.2 异常类的继承架构

异常可分为两大类：java.lang.Exception 类与 java.lang.Error 类。这两个类均继承自 java.lang.Throwable 类。图 7-3 为 Throwable 类的继承关系图。

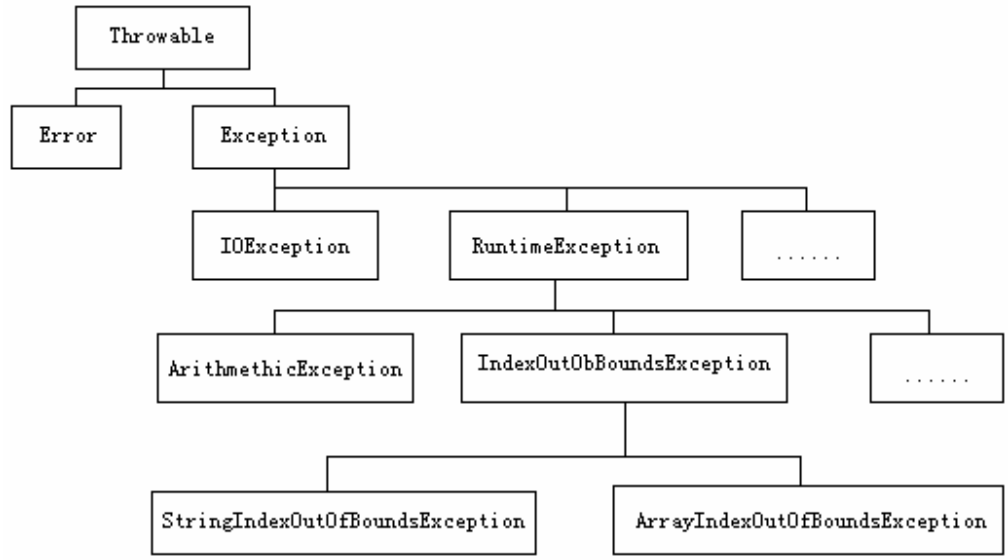


图 7-3　Throwable 类的继承关系图

习惯上将 Error 与 Exception 类统称为异常类，但这两者本质上还是有不同的。Error 类专门用来处理严重影响程序运行的错误，可是通常程序设计者不会设计程序代码去捕捉这种错误，其原因在于即使捕捉到它，也无法给予适当的处理，如 JAVA 虚拟机出错就属于一种 Error。

不同于 Error 类，Exception 类包含了一般性的异常，这些异常通常在捕捉到之后便可做妥善的处理，以确保程序继续运行，如 TestException7_2 里所捕捉到的 ArrayIndexOutOfBoundsException 就是属于这种异常。

从异常类的继承架构图中可以看出：Exception 类扩展出数个子类，其中 IOException、RunntimeException 是较常用的两种。RunntimeException 即使不编写异常处理的程序代码，依然可以编译成功，而这种异常必须是在程序运行时才有可能发生，例如：数组的索引值超出了范围。与 RunntimeException 不同的是，IOException

一定要编写异常处理的程序代码才行，它通常用来处理与输入/输出相关的操作，如文件的访问、网络的连接等。

当异常发生时，发生异常的语句代码会抛出一个异常类的实例化对象，之后此对象与 catch 语句中的类的类型进行匹配，然后在相应的 catch 中进行处理。

7.3 抛出异常

前两节介绍了 try_catch_finally 程序块的编写方法，本节将介绍如何抛出（throw）异常，以及如何由 try-catch 来接收所抛出的异常。抛出异常有下列两种方式：

1、 程序中抛出异常

2、 指定方法抛出异常

以下两小节将介绍如何在程序中抛出异常以及如何指定方法抛出异常。

7.3.1 在程序中抛出异常

在程序中抛出异常时，一定要用到 throw 这个关键字，其语法如下：

【 格式 7-2 抛出异常的语法】

```
throw 异常类实例对象 ;
```

从格式 7-2 中可以发现在 throw 后面抛出的是一个异常类的实例对象，下面来看一个实例：

范例：TestException7_4.java

```
01  public class TestException7_4
02  {
03      public static void main(String args[])
04      {
05          int a=4,b=0;
06          try
```

```
07              {
08                  if(b==0)
09                      throw new ArithmeticException("一个算术异常");   // 抛出异常
10                  else
11                      System.out.println(a+"/"+b+"="+a/b);// 若抛出异常，则执行此行
12              }
13              catch(ArithmeticException e)
14              {
15                  System.out.println("抛出异常为："+e);
16              }
17          }
18  }
```

输出结果：

抛出异常为：java.lang.ArithmeticException: 一个算术异常

程序说明：

1、 程序 TestException7_4 是要计算 a/b 的值。因 b 是除数，不能为 0。若 b 为 0，则系统会抛出 ArithmeticException 异常，代表除到 0 这个数。

2、 在 try 块里，利用第 8 行来判断除数 b 是否为 0。如果 b=0，则运行第 9 行的 throw 语句，抛出 ArithmeticException 异常。如果 b 不为 0，则输出 a/b 的值。在此例中强制把 b 设为 0，因此 try 块的第 9 行会抛出异常，并由第 13 行的 catch()捕捉到异常。

3、 抛出异常时，throw 关键字所抛出的是异常类的实例对象，因此第 9 行的 throw 语句必须使用 new 关键字来产生对象。

7.3.2 指定方法抛出异常

如果方法内的程序代码可能会发生异常，且方法内又没有使用任何的代码块来捕捉这些异常时，则必须在声明方法时一并指明所有可能发生的异常，以便让调用此方法的程序得以做好准备来捕捉异常。也就是说，如果方法会抛出异常，则可将处理此

异常的 try-catch-finally 块写在调用此方法的程序代码内。

如果要由方法抛出异常，则方法必须以下面的语法来声明：

【 格式 7-3 由方法抛出异常】

方法名称（参数…） throws 异常类 1，异常类 2，…

范例 TestException7_5 是指定由方法来抛出异常的，注意此处把 main()方法与 add()方法编写在同一个类内，如下所示：

范例：TestException7_5.java

```
01  class Test
02  {
03      // throws 在指定方法中不处理异常，在调用此方法的地方处理
04      void add(int a,int b) throws Exception
05      {
06          int c;
07          c=a/b;
08          System.out.println(a+"/"+b+"="+c);
09      }
10  }
11  public class TestException7_5
12  {
13      public static void main(String args[])
14      {
15          Test t = new Test() ;
16          t.add(4,0);
17      }
18  }
```

编译结果：

TestException7_5.java:16: unreported exception java.lang.Exception; must be caught
or declared to be thrown
t.add(4,0);
^
1 error

程序说明：

1、 程序 1~10 行声明一 Test 类，此类中有一 add(int a,int b)方法，但在此方法后用 throws 关键字抛出一 Exception 异常。

2、 程序第 15 行实例化一 Test 对象 t，在第 16 行调用 Test 类中的 add()方法。

从上面的编译结果中可以发现，如果在类的方法中用 throws 抛出一个异常，则在调用它的地方就必须明确地用 try-catch 来捕捉。

小提示：

在 TestExeption7_5 程序之中，如果在 main()方法后再用 throws Exception 声明的话，那么程序也是依然可以编译通过的。也就是说在调用用 throws 抛出异常的方法时，可以将此异常在方法中再向上传递，而 main()方法是整个程序的起点，所以如果在 main()方法处如果再用 throws 抛出异常，则此异常就将交由 JVM 进行处理了。

7.4 编写自己的异常类

为了各种异常，Java 可通过继承的方式编写自己的异常类。因为所有可处理的异常类均继承自 Exception 类，所以自定义异常类也必须继承这个类。自己编写异常类的语法如下：

【 格式 7-4 编写自定义异常类】

```
class 异常名称 extends Exception
{
   … …
}
```

读者可以在自定义异常类里编写方法来处理相关的事件，甚至可以不编写任何语句也可正常地工作，这是因为父类 Exception 已提供相当丰富的方法，通过继承，子类均可使用它们。

接下来以一个范例来说明如何定义自己的异常类以及如何使用它们。

范例：TestException7_6.java

```
01  class DefaultException extends Exception
02  {
03      public DefaultException(String msg)
04      {
05          // 调用 Exception 类的构造方法，存入异常信息
06          super(msg) ;
07      }
08  }
09  public class TestException7_6
10  {
11      public static void main(String[] args)
12      {
13          try
14          {
15              // 在这里用 throw 直接抛出一个 DefaultException 类的实例对象
16              throw new DefaultException("自定义异常！ ") ;
17          }
18          catch(Exception e)
19          {
20              System.out.println(e) ;
21          }
22      }
23  }
```

输出结果：

DefaultException: 自定义异常！

程序说明：

1、 程序 1~8 行声明一 DefaultException 类，此类继承自 Exception 类，所以此类为自定义异常类。

2、 程序第 6 行调用 super 关键字，调用父类（Exception）的有一个参数的构造方法，传入的为异常信息。

Exception 构造方法：

public Exception(String message)

3、 程序第 16 行用 throw 抛出一 DefaultException 异常类的实例化对象。

在 JDK 中提供的大量 API 方法之中含有大量的异常类，但这些类在实际开发中往往并不能完全的满足设计者对程序异常处理的需要，在这个时候就需要用户自己去定义所需的异常类了，用一个类清楚的写出所需要处理的异常。

• 本章摘要：

1、 程序中没有处理异常代码时，Java 的默认异常处理机制会做下面的操作：

（1）、 抛出异常。

（2）、 停止程序运行。

2、 异常处理是由 try、catch 与 finally 三个关键字所组成的程序块，其语法请参考格式 7-1。

3、 try 程序块中若有异常发生时，程序的运行便会中断，抛出“由系统类所产生的对象”，并依下列的步骤来运行：

（1）、 抛出的对象如果属于 catch()括号内所要捕捉的异常类，catch 会捕捉此异常，然后进到 catch 程序块里继续执行。

（2）、 无论 try 程序块是否捕捉到异常，也不管捕捉到的异常是否与 catch()括号里的异常相同，最后都会运行 finally 块里的程序代码。

（3）、 finally 中的代码是异常的统一出口，无论是否发生异常都会执行此段代码。

4、 当异常发生时，有两种处理方式：

（1）、 交由 Java 默认的异常处理机制去处理。

（2）、 自行编写 try-catch-finally 块来捕捉异常。

5、 异常可分为两大类：java.lang.Exception 与 java.lang.Error 类。

6、 RunntimeException 可以不编写异常处理的程序代码，依然可以编译成功，它是在程序运行时才有可能发生的；而其它的 Exception 一定要编写异常处理的程序代码才能使程序通过编译。

7、 catch()括号内，只接收由 Throwable 类的子类所产生的对象，其它的类均不接收。

8、 抛出异常有下列两种方式：

（1）、 在程序中抛出异常。

（2）、 指定方法抛出异常。

9、 程序中抛出异常时，要用到 throw 这个关键字。

10、如果方法会抛出异常（使用 throws），则可将处理此异常的 try-catch-finally 块写在调用此方法的程序代码中。

第八章 包及访问权限

在 Java 里，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，如此的程序代码将更易于维护。同时在将类分割开之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。

8.1 包的概念及使用

8.1.1 包（package）的基本概念

当一个大型程序由数个不同的组别或人员开发共同开发时，用到相同的类名称是很有可能的事。如果这种情况发生，还要确保程序可以正确运行，就必须通过 package 关键字来帮忙了。

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。那么具体应该怎么使用呢？在类或接口的最上面一行加上 package 的声明就可以了。

【 格式 8-1 package 的声明】

```
package package 名称 ;
```

经过 package 的声明之后，在同一文件内的接口或类都被纳入相同的 package 中。程序 TestPackage1 是使用 package 的一个范例，如下所示：

范例：TestPackage1.java

```
01  package demo.java ;
02  class Person
03  {
04      public String talk()
05      {
```

```
06              return "Person —— >> talk()" ;
07          }
08  }
09
10  class TestPackage1
11  {
12          public static void main(String[] args)
13          {
14              System.out.println(new Person().talk()) ;
15          }
16  }
```

在 TestPackage1.java 中，除了第 1 行的加的 package demo.java 声明语句之外，其余的程序都是读者见过的。由于第 1 行声明了一个 demo.java 的包，所以就相当于将 Person 类、TestPackage1 类放入了 demo.java 文件夹之下。现在来看一下上面的程序是如何编译的：

```
javac -d . TestPackage1.java
```

“-d”：表示生成目录

“.”：表示在当前目录下生成

这样就会在当前目录下生成一个 demo 的文件夹，在 demo 文件夹下又会生成一个 java 文件夹，在此文件夹下会有编译好的 Person.class 和 TestPackage1.class，编译好之后用下面的语法来执行它：

```
java demo.java.TestPackage1
```

输出结果：

```
Person —— >> talk()
```

8.1.2 import 语句的使用

到目前为止，所介绍的类都是属于同一个 package 的，因此在程序代码的编写上并不需要做修改。但如果几个类分别属于不同的 package 时，在某个类要访问到其它类的成员时，则必须做下列的修改：

若某个类需要被访问时，则必须把这个类公开出来，也就是说，此类必须声明成 public。

若要访问不同 package 内某个 public 类的成员时，在程序代码内必须明确地指明“被访问 package 的名称.类名称”。

【 格式 8-2 package 的导入】

```
import package 名称.类名称 ；
```

通过 import 命令，可将某个 package 内的整个类导入，因此后续的程序代码便不用再写上被访问 package 的名称了。

举一个范例来说明 import 命令的用法。此范例与 TestPackage1 类似，只是将两个类分别放在不同的包中：

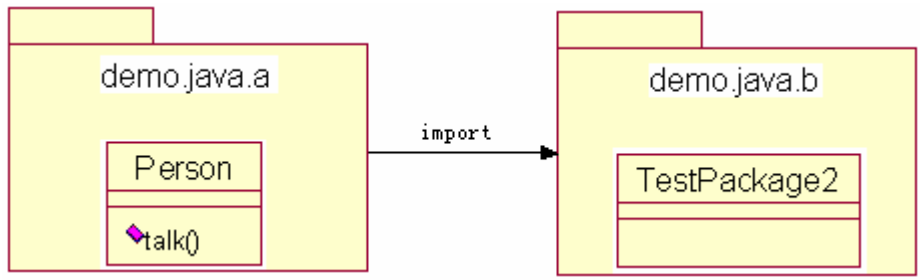


图 8-1　包的导入

范例：Person.java

```
01  package demo.java.a ;
02
03  public class Person
04  {
05      public String talk()
```

```
06          {
07                  return "Person —— >> talk()" ;
08          }
09  }
```

程序说明：

程序第 1 行声明一个 demo.java.a 的包，将 Person 类放入此包之中。

范例：TestPackage2.java

```
01  package demo.java.b ;
02  import demo.java.a.Person ;
03
04  class TestPackage2
05  {
06          public static void main(String[] args)
07          {
08                  System.out.println(new Person().talk()) ;
09          }
10  }
```

输出结果：

```
Person —— >> talk()
```

程序说明：

1、 程序第 1 行声明一个 demo.java.b 包，将 TestPackage2 类放入此包之中。

2、 程序第 2 行使用 import 语句，将 demo.java.a 包中的 Person 类导入到此包之中。

注意：

可以将第 2 行的 import demo.java.a.Person 改成 import demo.java.a.*，表示导入包中的所有类，另外需要告诉读者的是，在 java 中有这样的规定：导入全部类或是导入指定的类，对于程序的性能是没有影响的，所以在开发中可以直接写导入全部类会比较方便。

另外，TestPackage2.java 程序也可以写成如下形式：

范例：TestPackage3.java

```
01  package demo.java.b ;
02
03  class TestPackage3
04  {
05      public static void main(String[] args)
06      {
07          System.out.println(new demo.java.a.Person().talk()) ;
08      }
09  }
```

可以发现在 TestPackage3.java 程序中并没有使用 import 语句，但是在程序第 7 行使用 Person 类的时候使用了“包名.类名”的方式使用了 Person 类，所以在程序中也可以使用此方法来使用非本类所在的包中的类。

8.1.3 JDK 中常见的包

SUN 公司在 JDK 中为程序开发者提供了各种实用类，这些类按功能不同分别被放入了不同的包中，供开发者使用，下面简要介绍其中最常用的几个包：

1、 java.lang — 包含一些 Java 语言的核心类，如 String、Math、Integer、System 和 Thread，提供常用功能。在 java.lang 包中还有一个子包：java.lang.reflect 用于实现 java 类的反射机制。

2、 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)。

3、 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。

4、 java.applet — 包含 applet 运行所需的一些类。

5、 java.net — 包含执行与网络相关的操作的类。

6、 java.io — 包含能提供多种输入/输出功能的类。

7、 java.util — 包含一些实用工具类，如定义系统特性、与日期日历相关的函数。

注意：java1.2 以后的版本中，java.lang 这个包会自动被导入，对于其中的类，不需要使用 import 语句来做导入了，如前面经常使用的 System 类。

8.2 类成员的访问控制权限

有了包的概念之后，下面就可以开始为读者讲解 JAVA 语言之中的访问控制权限的概念了。在 JAVA 中有四种访问控制权限，分别为：private、default、protected、public。

1、 private 访问控制符

在前面已经介绍了 private 访问控制符的作用，如果一个成员方法或成员变量名前使用了 private 访问控制符，那么这个成员只能在这个类的内部使用。

注意：

不能在方法体内声明的变量前加 private 修饰符。

2、 默认访问控制符

如果一个成员方法或成员变量名前没有使用任何访问控制符，就称这个成员所拥有的是默认的（default）访问控制符。默认的访问控制成员可以被这个包中的其它类访问。如果一个子类与其父类位于不同的包中，子类也不能访问父类中的默认访问控制成员。

3、 protected 访问控制符

如果一个成员方法或成员变量名前使用了 protected 访问控制符，那么这个成员既可以被同一个包中的其它类访问，也可以被不同包中的子类访问。

4、 public 访问控制符

如果一个成员方法或成员变量名前使用了 public 访问控制符，那么这个成员可以被所有的类访问，不管访问类与被访问类是否在同一个包中。

最后，用表 8.1 来总结上述访问控制符的权限。

	private	default	protected	public
同一类	✓	✓	✓	✓
同一包中的类		✓	✓	✓
子类			✓	✓
其他包中的类				✓

对于 private、default、public 三种控制权限，前面已经为读者介绍过了，下面的范例为读者讲解 protected 关键字的使用方法：

范例：Person.java

```
01  package demo.java.a ;
02
03  public class Person
04  {
05      protected String name ;
06      public String talk()
07      {
08          return "Person —— >> talk()" ;
09      }
10  }
```

程序说明：

程序第 5 行声明一 String 类型的属性 name，该变量用 protected 关键字声明，所以此属性只能在本类及其子类中使用。

范例：Student.java

```
01  package demo.java.b ;
02  import demo.java.a.* ;
03
04  public class Student extends Person
05  {

06      public Student(String name)
07      {
```

```
08              this.name = name ;
09          }
10          public String talk()
11          {
12              return "Person ——— >> talk() ，  "+this.name ;
13          }
14      }
```

程序说明：

1、 程序第 2 行，导入 Person 类。

2、 程序第 4 行 Student 类继承自 Person 类

3、 程序第 8 行 Student 类访问了 Person 类中的 name 属性。

范例：TestPackage4.java

```
01  package demo.java.c ;
02  import demo.java.b.* ;
03
04  class TestPackage4
05  {
06      public static void main(String[] args)
07      {
08          Student s = new Student("javafans") ;
09          s.name = "javafans" ;
10          System.out.println(s.talk()) ;
11      }
12  }
```

编译结果：

```
TestPackage4.java:9: name has protected access in demo.java.a.Person
                    s.name = "javafans" ;
                     ^
1 error
```

可以发现，在程序第 9 行，通过对象调用受保护的属性，所以程序在编译时 JDK 会报错。

8.2 Java 的命名习惯

读者通过查看 JDK 文档，可以发现，在 JDK 中类的声明、方法的声明、常量的声明都是有一定规律的，规律如下：

- 包名中的字母一律小写，如：demo.java。
- 类名、接口名应当使用名词，每个单词的首字母大写，如：TestPerson。
- 方法名，第一个单词小写，后面每个单词的首字母大写，如：talkMySelf。
- 常量名中的每个字母一律大写，如：COUNTRY。

8.3 Jar 文件的使用

开发者用的 JDK 中的包和类主要在 JDK 的安装目录下的 jre\lib\rt.jar 文件中，由于 Java 虚拟机会自动找到这个 jar 包，所以在开发中使用这个 jar 包的类时，无需再用 classpath 来指向它们的位置了。

jar 文件就是 Java Archive File，而它的应用是与 Java 息息相关的。jar 文件就是一种压缩文件，与常见的 ZIP 压缩文件格式兼容，习惯上称之为 jar 包。如果开发者开发了许多类，当需要把这些类提供给用户使用时，通常都会将这些类压缩到一个 jar 文件中，以 jar 包的方式提供给用户使用。只要别人的 classpath 环境变量的设置中包含这个 jar 文件，Java 虚拟机就能自动在内存中解压这个 jar 文件，把这个 jar 文件当作一个目录，在这个 jar 文件中去寻找所需要的类及包名所对应的目录结构。

jar 命令是随 JDK 一起安装的，存放在 JDK 安装目录下的 bin 目录中（比如在本书中是 c:\j2sdk1.4.0\bin 目录下），Windows 下的文件名为 jar.exe，Linux 下的文件名为 jar。jar 命令是 Java 中提供的一个非常有用的命令，可以用来对大量的类(.class 文件)进行压缩，然后存为.jar 文件。那么通过 jar 命令所生成的.jar 压缩文件又有什么优点

呢？一方面，可以方便管理大量的类文件，另一方面，进行了压缩也减少了文件所占的空间。对于用户来说，除了安装 JDK 之外什么也不需要做，因为 SUN 公司已经帮

开发者做好了其它的一切工作。

可以在命令行窗口下运行 jar.exe 程序，就可以看下图示：

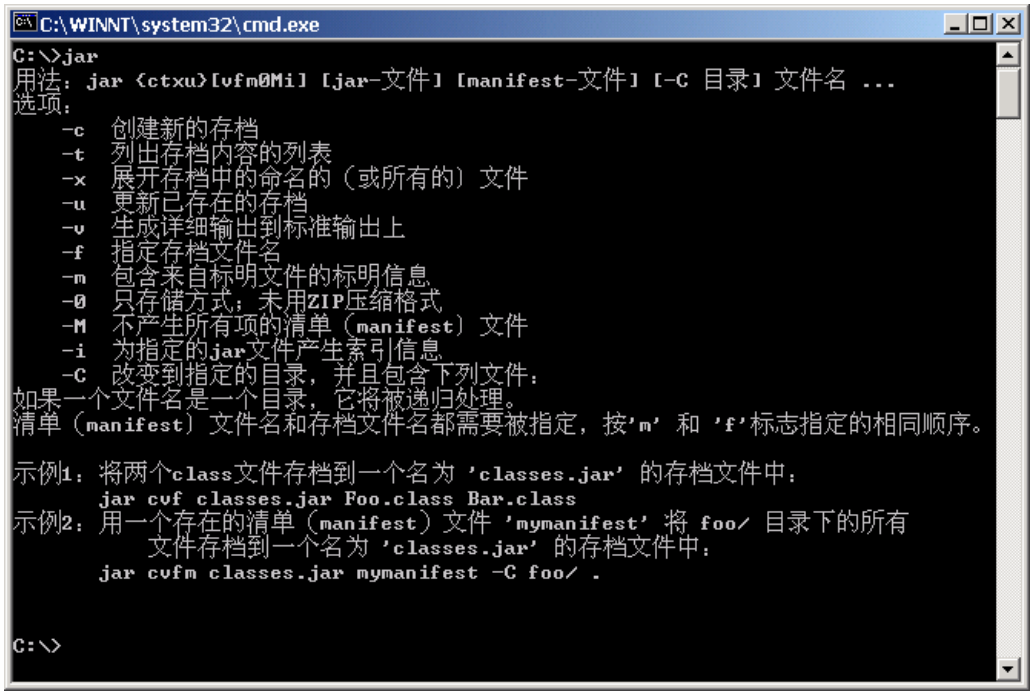


图 8_2

从 JDK 的提示中可以发现，只要在命令行中用以下命令就可以将一个包打成一个 jar 文件：

jar –cvf create.jar demo

- -c：创建新的存档
- -v：生成详细输出到标准输出上
- -f：指定存档文件名
- create.jar：是生成 jar 文件的名称
- demo：要打成 jar 文件的包

• 本章摘要：

1、 java 中使用包可以实现多人协作的开发模式。

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、 java 中的访问控制权限分为四种：private、default、protected、public。

5、 使用 jar 命令可以将一个包打成一个 jar 文件，供用户使用。

第 3 部分　　Java 程序应用

- JAVA 多线程机制
- IO 操作
- 网络程序设计
- Java 常用 API 使用

第 9 章　多线程

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循序运行单独一个程序块，但无法同时运行不同的多个程序块。Java 的“多线程”恰可弥补这个缺憾，它可以让不同的程序块一起运行，如此一来可让程序运行更为顺畅，同时也可达到多任务处理的目的。

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，使得所有程序好象是在“同时”运行一样。

线程是比进程更小的执行单位，线程是进程内部单一的一个顺序控制流。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，形成多条执行线索。一个进程可能包含了多个同时执行的线程。

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。线程和进程的主要差别体现在以下两个方面：

（1）、同样作为基本的执行单元，线程是划分得比进程更小的执行单位。

（2）、每个进程都有一段专用的内存区域。与此相反，线程却共享内存单元（包括代码和数据），通过共享的内存单元来实现数据交换、实时通信与必要的同步操作。

多线程的应用范围很广。在一般情况下，程序的某些部分同特定的事件或资源联系在一起，同时又不想为它而暂停程序其它部分的执行，这种情况下，就可以考虑创建一个线程，令它与那个事件或资源关联到一起，并让它独立于主程序运行。通过使用线程，可以避免用户在运行程序和得到结果之间的停顿，还可以让一些任务（如打印任务）在后台运行，而用户则在前台继续完成一些其它的工作。总之，利用多线程

技术，可以使编程人员方便地开发出能同时处理多个任务的功能强大的应用程序。

9.2 认识线程

在传统的程序语言里，运行的顺序总是必须顺着程序的流程来走，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走，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”的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比较长的一段时间来运算，此时便可让另一个线程来做其它的处理。

本节将用一个简单的程序来说明单一线程与多线程的不同。ThreadDemo9_1 是单一线程的范例，其程序代码编写方法与前几节的程序代码并没有什么不同。

范例：ThreadDemo9_1.java

```
01  public class ThreadDemo9_1
02  {
03      public static void main(String args[])
04      {
05          new TestThread().run();
06          // 循环输出
07          for(int i=0;i<10;i++)
08          {
09              System.out.println("main 线程在运行");
10          }
11      }
12  }
13  class TestThread
14  {
15      public void run()
16      {
17          for(int i=0;i<10;i++)
```

```
18              {
19                  System.out.println("TestThread  在运行");
20              }
21          }
22      }
```

输出结果：

```
TestThread  在运行
TestThread  在运行
TestThread  在运行
TestThread  在运行
TestThread  在运行
TestThread  在运行
TestThread  在运行
TestThread  在运行
TestThread  在运行
TestThread  在运行
main  线程在运行
main  线程在运行
main  线程在运行
main  线程在运行
main  线程在运行
main  线程在运行
main  线程在运行
main  线程在运行
main  线程在运行
main  线程在运行
```

程序说明：

1、 在 15~21 行里定义了 run()方法，用循环输出 10 个连续的字符串。

2、 第 5 行创建 TestThread 对象之后调用 run()方法，输出“TestThread 在运行”，最后执行 main 方法中的循环，输出“main 线程在运行”。

从本例中可看出，要想运行 main 方法中的循环，必须要等 TestThread 类中的 run()方法执行完之后才可以运行，这便是单一线程的缺陷，在 Java 里，是否可以同时运行

9 与 19 行的语句，使得“main 线程在运行”和“TestThread 在运行”交错输出呢？答案是肯定的，其方法是——在 Java 里激活多个线程。

那么，该如何激活线程呢？

如果在类里要激活线程，必须先做好下面两个准备：

（1）、线程必须扩展自 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。

9.2.1 通过继承 Thread 类实现多线程

Thread 存放在 java.lang 类库里，但并不需加载 java.lang 类库，因为它会自动加载。此外，run()方法是定义在 Thread 类里的一个方法，因此把线程的程序代码编写在 run()方法内，事实上所做的就是覆盖的操作。因此要使一个类可激活线程，必须按照下面的语法来编写：

【 格式 9-1 多线程的定义语法】

```
class 类名称 extends Thread     // 从 Thread 类扩展出子类
{
    属性
    方法…
    修饰符 run(){                // 复写 Thread 类里的 run()方法
       以线程处理的程序;
    }
}
```

接下来用按照上述的语法来重新编写 ThreadDemo9_1，使它可以同时激活多个线程：

范例：修改后的 ThreadDemo9_1.java

```
01  public class ThreadDemo9_1
02  {
03      public static void main(String args[])
04      {
05          new TestThread().start();
06          // 循环输出
07          for(int i=0;i<10;i++)
08          {
09              System.out.println("main  线程在运行");
10          }
11      }
12  }
13  class TestThread extends Thread
14  {
15      public void run()
16      {
17          for(int i=0;i<10;i++)
18          {
19              System.out.println("TestThread  在运行");
20          }
21      }
22  }
```

调用 Thread 类中的 start()方法，实际上是调用 run()方法

TestThread 类继承了 Thread 类，此类实现了多线程

输出结果：

```
main  线程在运行
TestThread  在运行
main  线程在运行
TestThread  在运行
main  线程在运行
TestThread  在运行
main  线程在运行
TestThread  在运行
main  线程在运行
```

```
TestThread 在运行
main 线程在运行
TestThread 在运行
main 线程在运行
TestThread 在运行
main 线程在运行
TestThread 在运行
main 线程在运行
TestThread 在运行
main 线程在运行
TestThread 在运行
```

从运行结果中可以发现，两行输出是交替进行的，也就是说程序是采用多线程机制运行的。与之前的程序相比，修改后的程序的第 13 行 TestThread 类继承了 Thread 类，第 5 行调用的不再是 run()方法，而是 start()方法。所以，要启动线程必须调用 Thread 类之中的 start()方法，而调用了 start()方法，也就是调用了 run()方法。

9.2.2 通过实现 Runnable 接口实现多线程

从前面的章节中读者应该已经清楚了，JAVA 程序只允许单一继承，即一个子类只能有一个父类，所以在 Java 中如果一个类继承了某一个类，同时又想采用多线程技术的时，就不能用 Thread 类产生线程，因为 Java 不允许多继承，这时就要用 Runnable 接口来创建线程了。

【 格式 9-2 多线程的定义语法】

```
class 类名称 implements Runnable     // 实现 Runnable 接口
{
    属性
    方法…
    修饰符 run(){                    // 复写 Thread 类里的 run()方法
       以线程处理的程序;
    }
}
```

范例：ThreadDemo9_2.java

```
01  public class ThreadDemo9_2
02  {
03      public static void main(String args[])
04      {
05          TestThread t = new TestThread() ;
06          new Thread(t).start();
07          // 循环输出
08          for(int i=0;i<10;i++)
09          {
10              System.out.println("main 线程在运行");
11          }
12      }
13  }
14  class TestThread implements Runnable
15  {
16      public void run()
17      {
18          for(int i=0;i<10;i++)
19          {
20              System.out.println("TestThread 在运行");
21          }
22      }
23  }
```

产生 Runnable 接口的子类实例化对象

通过 Thread 类的 start()方法，启动多线程程序

TestThread 通过实现 Runnable 接口，实现多线程

输出结果：

```
main 线程在运行
TestThread 在运行
main 线程在运行
TestThread 在运行
main 线程在运行
TestThread 在运行
main 线程在运行
TestThread 在运行
main 线程在运行
TestThread 在运行
main 线程在运行
TestThread 在运行
main 线程在运行
TestThread 在运行
main 线程在运行
TestThread 在运行
main 线程在运行
TestThread 在运行
main 线程在运行
TestThread 在运行
```

程序说明：

1、 程序第 14 行 TestThread 类实现了 Runnable 接口，同时复写了 Runnable 接口之中的 run()方法，也就是说此类为一多线程实现类。

2、 程序第 5 行实例化一 TestThread 类的对象。

3、 程序第 6 行通过 TestThread 类（Runnable 接口的子类）去实例化一 Thread 类的对象，之后调用 start()方法启动多线程。

从输出结果中可以发现，无论继承了 Thread 类还是实现了 Runnable 接口运行的结果都是一样的。有些读者可能会不理解了，为什么实现了 Runnable 接口还需要调用 Thread 类中的 start()方法才能启动多线程呢？读者通过查找 JDK 文档就可以发现，在 Runnable 接口中只有一个 run()方法，如图 9-1 所示：

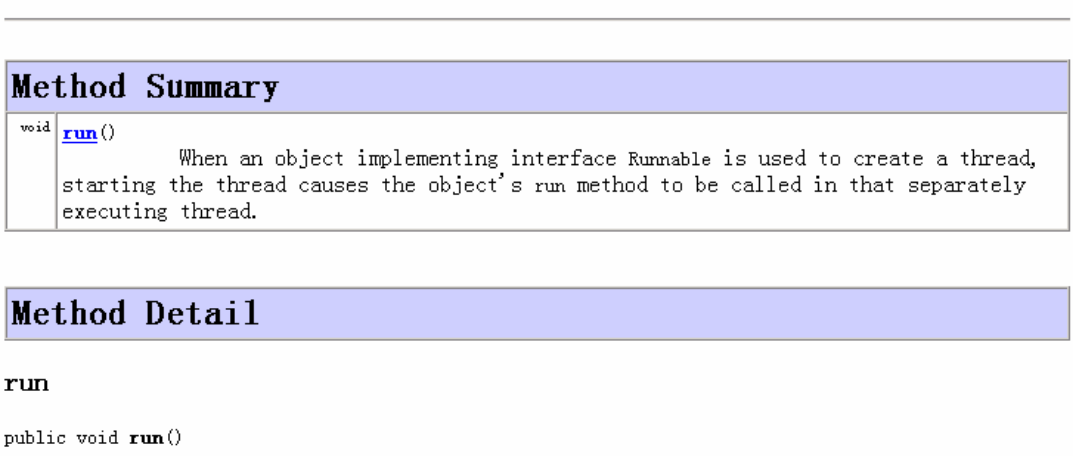


图 9-1　Runnable 接口中的方法列表

从上图中可以发现，在 Runnable 接口中并没有 start()方法，所以一个类实现了 Runnable 接口也必须用 Thread 类中的 start()方法来启动多线程。这点可以通过查找 JDK 文档中的 Thread 类发现，在 Thread 类之中，有这样一个构造方法：

public Thread(Runnable target)

由此构造方法可以发现，可以将一个 Runnable 接口的实例化对象作为参数去实例化 Thread 类对象。在实际的开发中，希望读者尽可能去使用 Runnable 接口去实现多线程机制。

9.2.3　两种多线程实现机制的比较

从前两节中可以发现，不管实现了 Runnable 接口还是继承了 Thread 类其结果都是一样的，那么这两者之间有什么关系呢？

读者可以通过查看 JDK 文档发现二者之间的联系，如下图 9-2 所示：

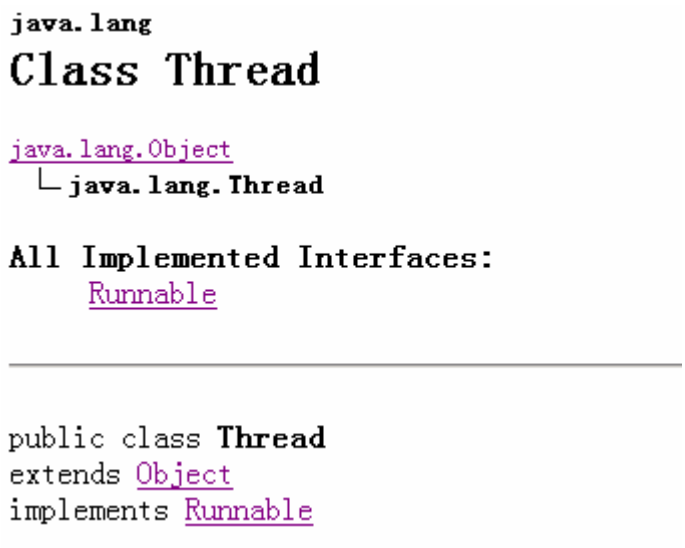


图 9-2　Thread 类与 Runnable 接口的关系

从图 9-2 中可以发现，Thread 类实现了 Runnable 接口，也就是说 Thread 类也是 Runnable 接口的一个子类。

那么两者之间除了这些联系之外还有什么区别呢？下面通过编写一个应用程序，来进行比较分析。程序 ThreadDemo9_3.java 是一个模拟铁路售票系统的范例，实现四个售票点发售某日某次列车的车票 20 张，一个售票点用一个线程来表示。

首先用继承 Thread 类来实现这个程序。

范例：ThreadDemo9_3.java

```
01  public class ThreadDemo9_3
02  {
03      public static void main(String [] args)
04      {
05          TestThread t=new TestThread();
06          // 一个线程对象只能启动一次
07          t.start();
```

```
08              t.start();
09              t.start();
10              t.start();
11          }
12      }
13      class TestThread extends Thread
14      {
15          private int tickets=20;
16          public void run()
17          {
18              while(true)
19              {
20                  if(tickets>0)
21              System.out.println(Thread.currentThread().getName()+"出售票"+tickets--);
22              }
23          }
24      }
```

输出结果：

```
Exception in thread "main" java.lang.IllegalThreadStateException
        at java.lang.Thread.start(Native Method)
        at ThreadDemo9_3.main(ThreadDemo9_3.java:8)
Thread-0 出售票 20
Thread-0 出售票 19
Thread-0 出售票 18
Thread-0 出售票 17
Thread-0 出售票 16
Thread-0 出售票 15
Thread-0 出售票 14
Thread-0 出售票 13
Thread-0 出售票 12
Thread-0 出售票 11
Thread-0 出售票 10
Thread-0 出售票 9
Thread-0 出售票 8
Thread-0 出售票 7
Thread-0 出售票 6
```

```
Thread-0 出售票 5
Thread-0 出售票 4
Thread-0 出售票 3
Thread-0 出售票 2
Thread-0 出售票 1
```

从上面的程序中可以发现，第 5 行创建了一个 TestThread 类的实例化对象，之后调用了四次 start()方法，但在运行结果可以发现，程序运行时出现了异常，之后却只有一个线程在运行。这也就说明了一个类继承 Thread 类之后，这个类的对象无论调用多少次 start()方法，结果都只有一个线程在运行。

另外，读者在第 21 行可以发现这样一条语句“Thread.currentThread().getName()”，此语句表示取得当前运行的线程名称，此方法会在后面进行讲解。

下面修改 ThreadDemo9_3 程序，这里让 main()方法中产生四个线程，修改后的程序如下：

范例：修改后的 ThreadDemo9_3.java

```
01  public class ThreadDemo9_3
02  {
03      public static void main(String [] args)
04      {
05          // 启动了四个线程，分别执行各自的操作
06          new TestThread().start();
07          new TestThread().start();
08          new TestThread().start();
09          new TestThread().start();
10      }
11  }
12  class TestThread extends Thread
13  {
14      private int tickets=20;
15      public void run()
16      {
17          while(true)
18        {
19              if(tickets>0)
```

```
20                 System.out.println(Thread.currentThread().getName()+"出售票"+tickets--);
21             }
22         }
23   }
```

输出结果：

```
Thread-3 出售票 5
Thread-2 出售票 4
Thread-3 出售票 4
Thread-2 出售票 3
Thread-3 出售票 3
Thread-1 出售票 2
Thread-2 出售票 2
Thread-1 出售票 1
Thread-2 出售票 1
Thread-3 出售票 2
Thread-3 出售票 1
Thread-0 出售票 12
Thread-0 出售票 11
Thread-0 出售票 10
Thread-0 出售票 9
Thread-0 出售票 8
Thread-0 出售票 7
Thread-0 出售票 6
Thread-0 出售票 5
Thread-0 出售票 4
Thread-0 出售票 3
Thread-0 出售票 2
Thread-0 出售票 1
```

由于程序的输出结果过长，所以只截取了后面一部分，但从这部分输出结果中可以发现，这里启动了四个线程对象，但这四个线程对象，各自占有各自的资源，所以可以得出结论，用 Thread 类实际上无法达到资源共享的目的。

那么实现 Runnable 接口会如何呢？下面这个范例也修改自 ThreadDemo9_3，读者可以观察一下输出结果。

范例：ThreadDemo9_4.java

```
01  public class ThreadDemo9_4
02  {
03      public static void main(String [] args)
04      {
05          TestThread t = new TestThread() ;
06          // 启动了四个线程，并实现了资源共享的目的
07          new Thread(t).start();
08          new Thread(t).start();
09          new Thread(t).start();
10          new Thread(t).start();
11      }
12  }
13  class TestThread implements Runnable
14  {
15      private int tickets=20;
16      public void run()
17      {
18          while(true)
19          {
20              if(tickets>0)
21          System.out.println(Thread.currentThread().getName()+"出售票"+tickets--);
22          }
23      }
24  }
```

输出结果：

```
Thread-1 出售票 20
Thread-1 出售票 19
Thread-1 出售票 18
Thread-1 出售票 17
Thread-1 出售票 16
Thread-1 出售票 15
Thread-1 出售票 14
Thread-1 出售票 13
Thread-2 出售票 12
Thread-3 出售票 11
Thread-4 出售票 10
```

Thread-2 出售票 9
Thread-3 出售票 8
Thread-4 出售票 7
Thread-2 出售票 6
Thread-3 出售票 5
Thread-4 出售票 4
Thread-2 出售票 3
Thread-3 出售票 2
Thread-4 出售票 1

从上面的程序中可以发现，在第 7 行到第 10 行启动了四个线程，但是从程序的输出结果来看，尽管启动了四个线程对象，但是结果都是操纵了同一个资源，实现了资源共享的目的。

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，把虚拟 CPU（线程）同程序的代码、数据有效分离，较好地体现了面向对象的设计思想。

（2）、 可以避免由于 Java 的单继承特性带来的局限。开发中经常碰到这样一种情况，即：当要将已经继承了某一个类的子类放入多线程中，由于一个类不能同时有两个父类，所以不能用继承 Thread 类的方式，那么就只能采用实现 Runnable 接口的方式了。

（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。当多个线程的执行代码来自同一个类的实例时，即称它们共享相同的代码。多个线程可以操作相同的数据，与它们的代码无关。当共享访问相同的对象时，即共享相同的数据。当线程被构造时，需要的代码和数据通过一个对象作为构造函数实参传递进去，这个对象就是一个实现了 Runnable 接口的类的实例。

事实上，几乎所有多线程应用都可用第二种方式，即实现 Runnable 接口。

9.3　线程的状态

每个 Java 程序都有一个缺省的主线程，对于 Java 应用程序，主线程是 main()方法执行的线索；对于 Applet 程序，主线程是指挥浏览器加载并执行 Java Applet 程序的线索。要想实现多线程，必须在主线程中创建新的线程对象。任何线程一般具有五种状态，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：

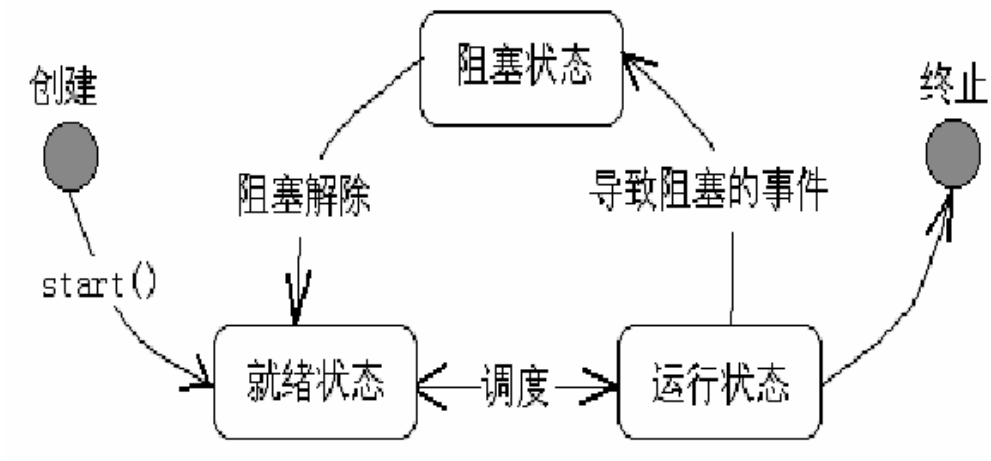


图 9-3　线程的状态转换

1、 新建状态

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它资源，但还处于不可运行状态。新建一个线程对象可采用线程构造方法来实现。

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，线程即处于死亡状态。处于死亡状态的线程不具有继续运行的能力。

9.4 线程操作的一些方法

从前面的讲解中可以发现，在 JAVA 实现多线程的程序里，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 接口之中，而是在 Thread 类之中，表 9-1 中，列出了 Thread 类中的主要方法。

表 9-1　Thread 类中的主要方法

方法名称	方法说明
public static int activeCount()	返回线程组中目前活动的线程的数目
public static native Thread　currentThread()	返回目前正在执行的线程
public void destroy()	销毁线程
public static int enumerate(Thread tarray[])	将当前和子线程组中的活动线程拷贝至指定的线程数组
public final String getName()	返回线程的名称
public final int getPriority()	返回线程的优先级
public final ThreadGroup getThreadGroup()	返回线程的线程组
public static boolean interrupted()	判断目前线程是否被中断，如果是，返回 true，否则返回 false
public final native boolean isAlive()	判断线程是否在活动，如果是，返回 true，否则返回 false
public boolean isInterrupted()	判断目前线程是否被中断，如果是，返回 true，否则返回 false
public final void join() throws InterruptedException	等待线程死亡
public final synchronized void join(long millis) throws InterruptedException	等待 millis 毫秒后，线程死亡
public final synchronized void join(long millis, int nanos) throws InterruptedException	等待 millis 毫秒加上 nanos 微秒后，线程死亡
public void run()	执行线程
public final void　setName()	设定线程名称
public final void　setPriority(int newPriority)	设定线程的优先值
public static native void　sleep(long millis) throws InterruptedException	使目前正在执行的线程休眠 millis 毫秒
public static void sleep(long millis,int nanos) throws InterruptedException	使目前正在执行的线程休眠 millis 毫秒加上 nanos 微秒
public native synchronized void start()	开始执行线程
public String toString()	返回代表线程的字符串
public static native void yield()	将目前正在执行的线程暂停，允许其它线程执行。

下面为读者介绍一些常用的方法。

9.4.1 取得和设置线程的名称

在 Thread 类之中，可以通过 getName()方法取得线程的名称，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 对象有相同的名字，但为了清晰，应该尽量避免这种情况的发生。

另外，如果程序并没有为线程指定名称，则系统会自动的为线程分配一个名称，如下范例所示：

范例：GetNameThreadDemo.java

```
01  public class GetNameThreadDemo extends Thread
02  {
03      public void run()
04      {
05          for(int i=0;i<10;i++)
06              printMsg();
07      }
08      public void printMsg()
09      {
10          // 获得运行此代码的线程的引用
11          Thread t = Thread.currentThread();
12          String name = t.getName();
13          System.out.println("name = "+name);
14      }
15      public static void main(String[] args)
16      {
17          GetNameThreadDemo t1 = new GetNameThreadDemo();
18          t1.start();
19          for(int i=0;i<10;i++)
20          {
```

```
21                      t1.printMsg();
22                  }
23              }
24      }
```

输出结果：

```
name = Thread-0
name = main
name = Thread-0
name = main
name = Thread-0
name = main
name = Thread-0
name = main
name = Thread-0
name = main
name = main
name = Thread-0
name = Thread-0
name = main
name = Thread-0
name = main
name = Thread-0
name = main
name = Thread-0
name = main
```

程序说明：

1、 程序第 1 行声明一 GetNameThreadDemo 类，此类继承自 Thread 类，之后 3~7 行复写 Thread 类之中的 run()方法。

2、 程序 8~14 行声明一 printMsg()方法，此方法用于取得当前线程的信息。在第 11 行，通过 Thread 类之中的 currentThread()方法，返回一 Thread 类的实例化对象，在表 9-1 中可以知道，此方法返回当前正在运行的线程，即：返回正在调用此方法的线程。第 12 行通过调用 Thread 类之中的 getName()方法，返回当前运行的线程的名称。

3、 第 6 行和第 21 行分别调用了 printMsg()方法，但第 6 行是从多线程的 run()方法中

调用，而第 21 行是从 main()方法中调用。

小提示：

有些读者可能会不理解了，为什么程序中输出的运行线程的名称中会有一个 main 呢？这是因为 main()方法也是一个线程，实际上在命令行中运行 java 命令时，就启动了一个 JVM 的进程，默认情况下此进程会产生两个线程：一个是 main()方法线程，另外一个就是垃圾回收（GC）线程。

下面再来看一下，如何在线程中设置线程的名称。

范例：SetNameThreadDemo.java

```
01  public class SetNameThreadDemo extends Thread
02  {
03      public void run()
04      {
05          for(int i=0;i<10;i++)
06          {
07              printMsg();
08          }
09      }
10      public void printMsg()
11      {
12          // 获得运行此代码的线程的引用
13          Thread t = Thread.currentThread();
14          String name = t.getName();
15          System.out.println("name = "+name);
16      }
17      public static void main(String args[])
18      {
19          SetNameThreadDemo tt = new SetNameThreadDemo();
20          // 在这里设置线程的名称
21          tt.setName("test thread");
22          tt.start();
23          for(int i=0;i<10;i++)
```

```
24              {
25                  tt.printMsg();
26              }

27          }
28      }
```

输出结果：

```
name = main
name = test thread
name = main
name = test thread
name = main
name = test thread
name = main
name = test thread
name = main
name = test thread
name = main
name = test thread
name = test thread
name = main
name = test thread
name = main
name = test thread
name = main
name = test thread
name = main
```

程序说明：

本程序与上面程序类似，唯一不同之处在于程序第 21 行调用了 Thread 类之中的 setName()方法，用于设置线程的名称，所以从运行结果中可以观察到，运行的线程中有一个名称为“test thread”线程。

9.4.2 判断线程是否启动

通过前面的讲解读者已经清楚了，通过 Thread 类之中的 start()方法通知线程规划器这个新线程已准备就绪，而且应当在规划器的最早方便时间调用它的 run()方法。

在程序中也可以通过 isAlive()方法来测试线程是否已经启动而且仍然在启动。

范例：StartThreadDemo.java

```
01  public class StartThreadDemo extends Thread
02  {
03      public void run()
04      {
05          for(int i=0;i<10;i++)
06          {
07              printMsg();
08          }
09      }
10      public void printMsg()
11      {
12          // 获得运行此代码的线程的引用
13          Thread t = Thread.currentThread();
14          String name = t.getName();
15          System.out.println("name = "+name);
16      }
17      public static void main(String[] args) {
18          StartThreadDemo t = new StartThreadDemo();
19          t.setName("test Thread");
20          System.out.println("调用 start()方法之前 , t.isAlive() = "+t.isAlive());
21          t.start();
22          System.out.println("刚调用 start()方法时 , t.isAlive() = "+t.isAlive());
23          for(int i=0;i<10;i++)
24          {
25              t.printMsg();
26          }
27          // 下面于句的数出结果是不固定的，有时输出 false，有时输出 true
28          System.out.println("main()方法结束时 , t.isAlive() = "+t.isAlive());
```

```
29          }
30    }
```

输出结果：

```
调用 start()方法之前 , t.isAlive() = false
刚调用 start()方法时 , t.isAlive() = true
name = main
name = test Thread
name = main
name = test Thread
name = main
name = test Thread
name = main
name = test Thread
name = main
name = test Thread
name = test Thread
name = main
name = test Thread
name = main
name = test Thread
name = main
name = test Thread
name = main
name = test Thread
name = main
main()方法结束时 , t.isAlive() = false
```

程序说明：

1、 程序第 20 行在线程运行之前调用 isAlive()方法，判断线程是否启动，但在此处并没有启动，所以返回“false”，表示线程未启动。

2、 程序第 22 行在启动线程之后调用 isAlive()方法，此时线程已经启动，所以返回“true”。

3、 程序第 28 行，在 main()方法快结束时调用 isAlive()方法，此时的状态不再固定，有可能是 true 有可能是 false。

9.4.3 后台线程与 setDaemon()方法

对 Java 程序来说，只要还有一个前台线程在运行，这个进程就不会结束，如果一个进程中只有后台线程在运行，这个进程就会结束。前台线程是相对后台线程而言的，前面所介绍的线程都是前台线程。那么什么样的线程是后台线程呢？如果某个线程对象在启动（调用 start()方法）之前调用了 setDaemon(true)方法，这个线程就变成了后台线程。下面来看一下进程中只有后台线程在运行的情况，如下所示：

范例：ThreadDaemon.java

```
01  public class ThreadDaemon
02  {
03      public static void main(String args[])
04      {
05          ThreadTest t = new ThreadTest() ;
06          Thread tt = new Thread(t) ;
07          tt.setDaemon(true) ;      // 设置后台运行
08          tt.start();
09      }
10  }
11
12  class ThreadTest implements Runnable
13  {
14          public void run()
15          {
16              while(true)
17            {
18              System.out.println(Thread.currentThread().getName()+"is running.");
19            }
```

```
20          }
21    }
```

从上面的程序和运行结果（图 9-4）上，可以看到：虽然创建了一个无限循环的线程，但因为它是后台线程，整个进程在主线程结束时就随之终止运行了。这验证了进程中只有后台线程运行时，进程就会结束的说法。

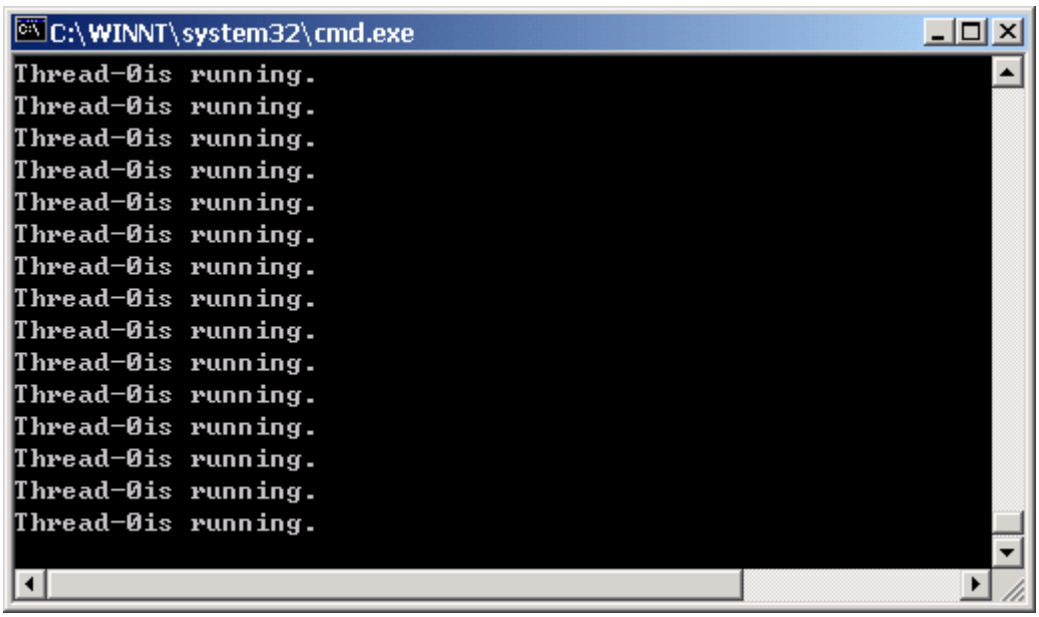


图 9-4　后台线程运行结果

9.4.4　线程的强制运行

在讲解此概念之前，请先看下面的范例：

范例：ThreadJoin.java

```
01    public class ThreadJoin
02    {
03          public static void main(String[] args)
04          {
```

```
05              ThreadTest t=new ThreadTest();
06              Thread pp=new Thread(t);
07              pp.start();
08              int i=0;
09              for(int x=0;x<10;x++)

10              {
11                  if(i==5)
12                  {
13                      try
14                      {
15                          pp.join();      // 强制运行完一线程后，再运行后面的线程
16                      }
17                      catch(Exception e)      // 会抛出 InterruptedException
18                      {
19                          System.out.println(e.getMessage());
20                      }
21                  }
22                  System.out.println("main Thread "+i++);
23              }
24          }
25      }
26      class ThreadTest implements Runnable
27      {
28          public void run()
29          {
30              String str=new String();
31              int i=0;
32              for(int x=0;x<10;x++)
33              {
34                  System.out.println(Thread.currentThread().getName()+" ---->> "+i++);
35              }
36          }
37      }
```

输出结果：

```
main Thread 0
main Thread 1
```

```
main Thread 2
main Thread 3
main Thread 4
Thread-0 ---->> 0
Thread-0 ---->> 1

Thread-0 ---->> 2
Thread-0 ---->> 3
Thread-0 ---->> 4
Thread-0 ---->> 5
Thread-0 ---->> 6
Thread-0 ---->> 7
Thread-0 ---->> 8
Thread-0 ---->> 9
main Thread 5
main Thread 6
main Thread 7
main Thread 8
main Thread 9
```

程序说明：

1、 在程序启动了两个线程，一个是 main()线程，一个是 pp 线程。

2、 程序第 15 行，调用 pp 线程对象的 join()方法，在程序的输出结果中可以发现，调用 join()方法之后，只有 pp 的线程对象在运行，也就是说，join()方法用来强制某一线程运行。

由上可见，pp 线程中的代码被并入到了 main 线程中，也就是 pp 线程中的代码不执行完，main 线程中的代码就只能一直等待。查看 JDK 文档可以发现，除了有无参数的 join 方法外，还有两个带参数的 join 方法，分别是 join(long millis)和 join(long millis,int nanos)，它们的作用是指定合并时间，前者精确到毫秒，后者精确到纳秒，意思是两个线程合并指定的时间后，又开始分离，回到合并前的状态。读者可以把上面的程序中的 join 方法修改成为有参数的，再看看程序运行的结果。

9.4.5 线程的休眠

在 Thread 类之中可以发现有一个名为 sleep(long millis)的静态方法，此方法用于线程的休眠。

范例：TwoThreadSleep.java

```
01  public class TwoThreadSleep extends Thread {
02      public void run() {
03          loop();
04      }
05      public void loop() {
06          String name = Thread.currentThread().getName();
07          System.out.println(name+" ---->> 刚进入 loop 方法");
08          for ( int i = 0; i < 10; i++ )
09          {
10              try {
11                  Thread.sleep(2000);
12              } catch ( InterruptedException x ) {}
13              System.out.println("name=" + name);
14          }
15          System.out.println(name+" ---->> 离开 loop 方法") ;
16      }
17      public static void main(String[] args)
18      {
19          TwoThreadSleep tt = new TwoThreadSleep();
20          tt.setName("my worker thread");
21          tt.start();
22          try {
23              Thread.sleep(700);
24          } catch ( InterruptedException x ) {}
25          tt.loop();
26      }
27  }
```

输出结果：

```
my worker thread ---->> 刚进入 loop 方法
```

```
main ---->> 刚进入 loop 方法
name=my worker thread
name=main
name=my worker thread
name=main

name=my worker thread
name=main
name=my worker thread
name=main
name=my worker thread
name=main
name=my worker thread
name=main
name=my worker thread
name=main
name=my worker thread
name=main
name=my worker thread
name=main
name=my worker thread
my worker thread ---->> 离开 loop 方法
name=main
main ---->> 离开 loop 方法
```

由程序可以发现，程序第 11 和 23 行，分别使用了 sleep()方法，所以运行此程序时，会发现运行的速度明显降低了很多，这是因为每次运行时，都需要先休眠一会儿。由于使用 sleep()方法会抛出一个 InterruptedException，所以在程序中需要用 try..catch()捕获。

9.4.6 线程的中断

当一个线程运行时，另一个线程可以调用对应的 Thread 对象的 interrupt()方法来中断它。

范例：SleepInterrupt.java

```
01  public class SleepInterrupt implements Runnable
02  {
03      public void run()
04      {
05          try {
06              System.out.println("在 run()方法中 - 这个线程休眠 20 秒");
07              Thread.sleep(20000);
08              System.out.println("在 run()方法中 - 继续运行");
09          }
10          catch (InterruptedException x) {
11              System.out.println("在 run()方法中 - 中断线程");
12              return;
13          }
14          System.out.println("在 run()方法中 - 休眠之后继续完成");
15          System.out.println("在 run()方法中 - 正常退出");
16      }
17      public static void main(String[] args)
18      {
19          SleepInterrupt si = new SleepInterrupt();
20          Thread t = new Thread(si);
21          t.start();
22          // 在此休眠是为确保线程能运行一会
23          try {
24              Thread.sleep(2000);
25          }
26          catch (InterruptedException x) {}
27          System.out.println("在 main()方法中 - 中断其它线程");
28          t.interrupt();
29          System.out.println("在 main()方法中 - 退出");
30      }
31  }
```

输出结果：

```
在 run()方法中 - 这个线程休眠 20 秒
在 main()方法中 - 中断其它线程
在 run()方法中 - 中断线程
```

在 main()方法中 - 退出

程序说明：

1、 SleepInterrupt 类实现了 Runnable 接口，同时复写 run()方法，在 run()方法之中，将线程休眠 20 秒。

2、 程序 21 行调用 start()方法，此方法用于启动线程。

3、 程序 23~26 行调用 sleep()方法，将线程休眠 2 秒，这样做，是为了保证 run()方法中的内容能够多执行一会儿。

4、 程序第 28 行，因为 Thread 对象 t 在 main()方法之中，所以由 main()线程调用 interrupt()方法，将另外一个线程中断。

也可以用 Thread 对象调用 isInterrupted()方法来检查每个线程的中断状态。

范例：InterruptCheck.java

```
01  public class InterruptCheck
02  {
03      public static void main(String[] args)
04      {
05          Thread t = Thread.currentThread();
06          System.out.println("A：t.isInterrupted() = " + t.isInterrupted());
07          t.interrupt();
08          System.out.println("B：t.isInterrupted() = " + t.isInterrupted());
09          System.out.println("C：t.isInterrupted() = " + t.isInterrupted());
10          try {
11              Thread.sleep(2000);
12              System.out.println("线程没有被中断！");
13          } catch ( InterruptedException x ) {
14          System.out.println("线程被中断！");
15          }
16          // 因为 sleep 抛出了异常，所以它清除了中断标志
17          System.out.println("D：t.isInterrupted() = " + t.isInterrupted());
18      }
```

```
19  }
```

输出结果：

A：t.isInterrupted() = false

B：t.isInterrupted() = true

C：t.isInterrupted() = true

线程被中断!

D：t.isInterrupted() = false

程序说明：

1、 程序第 5 行通过 Thread 类中的 currentThread()方法，取得当前运行的线程，因为此代码是在 main()方法之中运行，所以当前的线程就为 main()线程。

2、 程序第 6 行，因为没有调用中断方法，所以此时线程未中断，但在第 7 行调用了中断方法，所以之后的线程状态都为中断。

3、 程序第 11 行，让线程开始休眠，但此时线程已经被中断，所以这个时候会抛出中断异常，抛出中断异常之后，会清除中断标记，所以最后在判断是否中断的时候，会返回线程未中断。

9.5 多线程的同步

9.5.1 同步问题的引出

在 9-2-3 节中讲解过的卖票程序中，极有可能碰到一种意外，就是同一张票号被打印两次或多次，也可能出现打印出的票号为 0 或是负数。这个意外出现的原因出现在下面这部分代码中：

```
if(tickets>0)
    System.out.println(Thread.currentThread().getName()+"  出  售  票
```

"+tickets--);

假设 tickets 的值为 1 的时候，线程 1 刚执行完 if(tickets>0)这行代码，正准备执行下面的代码，就在这时，操作系统将 CPU 切换到了线程 2 上执行，此时 tickets 的值

仍为 1，线程 2 执行完上面两行代码，tickets 的值变为 0 后，CPU 又切回到了线程 1 上执行，线程 1 不会再执行 if(tickets>0)这行代码，因为先前已经比较过了，并且比较的结果为真，线程 1 将直接往下执行这行代码：

System.out.println(Thread.currentThread().getName()+"出售票"+tickets--);

但此刻 tickets 的值已变为 0，屏幕打印出的将是 0。

要想立即见到这种意外，可用在程序中调用 Thread.sleep()静态方法来刻意造成线程间的这种切换，Thread.sleep()方法迫使线程执行到该处后暂停执行，让出 CPU 给别的线程，在指定的时间（这里是毫秒）后，CPU 回到刚才暂停的线程上执行。修改完的 TestThread 代码如下：

范例：ThreadDemo9_5.java

```
01  public class ThreadDemo9_5
02  {
03      public static void main(String [] args)
04      {
05          TestThread t = new TestThread() ;
06          // 启动了四个线程，实现了资源共享的目的
07          new Thread(t).start();
08          new Thread(t).start();
09          new Thread(t).start();
10          new Thread(t).start();
11      }
12  }
13  class TestThread implements Runnable
14  {
15      private int tickets=20;
16      public void run()
17      {
```

```
18              while(true)
19              {
20                  if(tickets>0)
21                  {

22                      try{
23                          Thread.sleep(100);
24                      }
25                      catch(Exception e){}
26              System.out.println(Thread.currentThread().getName()+"出售票"+tickets--);
27                  }
28              }
29          }
30  }
```

在上面的程序代码中，故意实现线程执行完 if(tickets>0)语句后，执行 Thread.sleep(100)，以让出 CPU 给别的线程。

编译运行上面的程序，屏幕上打出的最后几行结果如下所示：

Thread-1 出售票 3
Thread-2 出售票 2
Thread-3 出售票 1
Thread-0 出售票 0
Thread-1 出售票-1
Thread-2 出售票-2

从运行结果中可以发现，票号被打印出来了负数，这就说明了有同一张票被卖了 4 次的意外发生。

这种意外问题的解决，就是本节要讲解的“线程安全”问题。造成这种意外的根本原因就是因为资源数据访问不同步引起的。那么该如何去解决这个问题呢？解决这种问题的关键是下面要引入的同步的概念。

9.5.2 同步代码块

如何避免上面的这种意外出现呢？如何保证开发出的程序是线程安全的呢？这就是下面要为读者讲解的如何实现线程间的同步问题。要解决上面的问题，必须保证下面这段代码的原子性：

```
if(tickets>0)
{
    System.out.println(Thread.currentThread().getName()+"出售票"+tickets--);
}
```

即当一个线程运行到 if(tickets>0)后，CPU 不去执行其它线程中的、可能影响当前线程中的下一句代码的执行结果的代码块，必须等到下一句执行完后才能去执行其它线程中的有关代码块。这段代码就好比一座独木桥，任何时刻，都只能有一个人在桥上行走，程序中不能有多个线程同时在这两句代码之间执行，这就是线程同步。

【 格式 9-3 同步代码块定义语法】

```
…
synchronized(对象)
{
    需要同步的代码 ;
}
…
```

现在修改一下 ThreadDemo9_5 程序中的 TestThread 类，使程序具有同步性，修改后的代码如下：

```
class TestThread implements Runnable
{
    private int tickets=20;
    public void run()
    {
```

```
        while(true)
        {
            synchronized(this)
            {
                if(tickets>0)
                {
                    try{
                        Thread.sleep(100);
                    }
                    catch(Exception e){}
                System.out.println(Thread.currentThread().getName()+"出售票"+tickets--);
                }
            }
        }
    }
}
```

在上面的代码中，程序将这些需要具有原子性的代码，放入 synchronized 语句内，形成了同步代码块。在同一时刻只能有一个线程可以进入同步代码块内运行，只有当该线程离开同步代码块后，其它线程才能进入同步代码块内运行。

9.5.3 同步方法

除了可以对代码块进行同步外，也可以对函数实现同步，只要在需要同步的函数定义前加上 synchronized 关键字即可。

【 格式 9-4 同步方法定义语法】

```
访问控制符 synchronized 返回值类型 方法名称(参数)
{
    …. ;
}
```

根据上述格式，修改 TestThread 类。

```
class TestThread implements Runnable
```

```
{
    private int tickets=20;
    public void run()
    {
        while(true)
        {
            sale() ;
        }
    }
    public synchronized void sale()
    {
        if(tickets>0)
        {
            try{
                Thread.sleep(100);
            }
            catch(Exception e){}
        System.out.println(Thread.currentThread().getName()+"出售票"+tickets--);
        }
    }
}
```

可见，编译运行后的结果同上面同步代码块方式的运行结果完全一样，也就是说在方法定义前使用 synchronized 关键字也能够很好地实现线程间的同步。

在同一类中，使用 synchronized 关键字定义的若干方法，可以在多个线程之间同步，当有一个线程进入了有 synchronized 修饰的方法时，其它线程就不能进入同一个对象使用 synchronized 来修饰的所有方法，直到第一个线程执行完它所进入的 synchronized 修饰的方法为止。

9.5.3 死锁

一旦有多个进程，且它们都要争用对多个锁的独占访问，那么就有可能发生死锁。如果有一组进程或线程，其中每个都在等待一个只有其它进程或线程才可以执行的操作，那么就称它们被死锁了。

最常见的死锁形式是当线程 1 持有对象 A 上的锁，而且正在等待对象 B 上的锁；而线程 2 持有对象 B 上的锁，却正在等待对象 A 上的锁。这两个线程永远都不会获得第二个锁，或是释放第一个锁，所以它们只会永远等待下去。这就好比两个

人在吃饭，甲拿到了一根筷子和一把刀子，乙拿到了一把叉子和一根筷子，他们都无法吃到饭。

于是，发生了下面的事件：

甲：“你先给我筷子，我再给你刀子!”

乙：“你先给我刀子，我才给你筷子”

……

结果可想而知，谁也没吃到饭。

要避免死锁，应该确保在获取多个锁时，在所有的线程中都以相同的顺序获取锁。

在下面的例子中，程序创建了两个类 A 和 B，它们分别具有方法 funA()和 funB()，在调用对方的方法前，funA()和 funB()都睡眠一会儿。主类 DeadLockDemo 创建 A 和 B 实例，然后，产生第二个线程以构成死锁条件。funA()和 funB()使用 sleep()方法来强制死锁条件出现。而在真实程序中死锁是较难发现的：

范例：DeadLockDemo.java

```
01  class A
02  {
03      synchronized void funA(B b)
04      {
05          String name=Thread.currentThread().getName();
06          System.out.println(name+ "  进入  A.foo ");
07          try
08          {
09              Thread.sleep(1000);
10          }
11          catch(Exception e)
12          {
13              System.out.println(e.getMessage());
14          }
15          System.out.println(name+ "  调用  B 类中的 last()方法");
16          b.last();
```

```
17        }
18        synchronized void last()
19        {

20            System.out.println("A 类中的 last()方法");
21        }
22    }
23    class B
24    {
25        synchronized void funB(A a)
26        {
27            String name=Thread.currentThread().getName();
28            System.out.println(name + "  进入  B  类中的");
29            try
30            {
31                Thread.sleep(1000);
32            }
33            catch(Exception e)
34            {
35                System.out.println(e.getMessage());
36            }
37            System.out.println(name + "  调用  A 类中的 last()方法");
38            a.last();
39        }
40        synchronized void last()
41        {
42            System.out.println("B 类中的 last()方法");
43        }
44    }
45    class DeadLockDemo implements Runnable
46    {
47        A a=new A();
48        B b=new B();
49        DeadLockDemo()
50        {
51            // 设置当前线程的名称
52            Thread.currentThread().setName("Main -->> Thread");
53            new Thread(this).start();
```

```
54              a.funA(b);
55              System.out.println("main 线程运行完毕");
56          }

57          public void run()
58          {
59              Thread.currentThread().setName("Test -->> Thread");
60              b.funB(a);
61              System.out.println("其他线程运行完毕");
62          }
63          public static void main(String[] args)
64          {
65              new DeadLockDemo();
66          }
67      }
```

运行结果如图 9-5 所示：

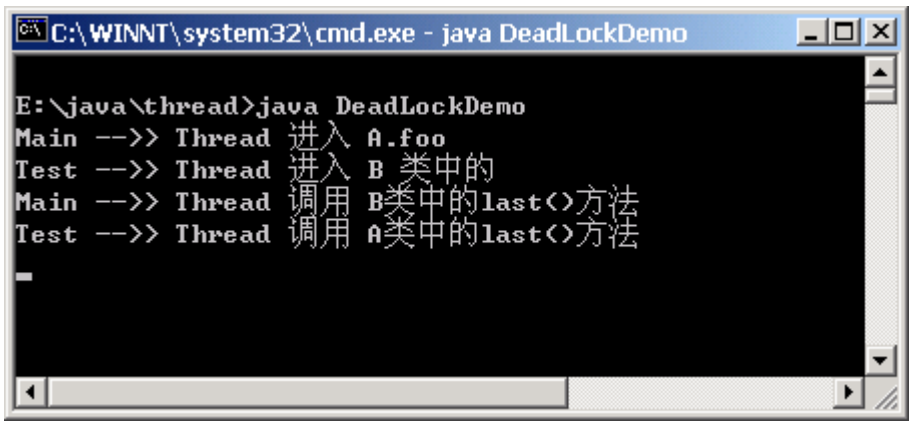


图 9-5　DeadLockDemo 的运行结果

从运行结果可以发现，Test -->> Thread 进入了 b 的监视器，然后又在等待 a 的监视器。同时 Main -->> Thread 进入了 a 的监视器并等待 b 的监视器。这个程序永远不会完成。

9.6 线程间通讯

9.6.1 问题的引出

下面通过这样的一个应用来讲解线程间的通信。把一个数据存储空间划分为两部分：一部分用于存储人的姓名，另一部分用于存储人的性别。这里的应用包含两个线程：一个线程向数据存储空间添加数据（生产者）；另一个线程从数据存储空间中取出数据（消费者）。这个程序有两种意外需要读者考虑：

第一种意外：假设生产者线程刚向数据存储空间中添加了一个人的姓名，还没有加入这个人的性别，CPU 就切换到了消费者线程，消费者线程将把这个人的姓名和上一个人的性别联系到了一起。

这个过程可用图 9-6 表示：

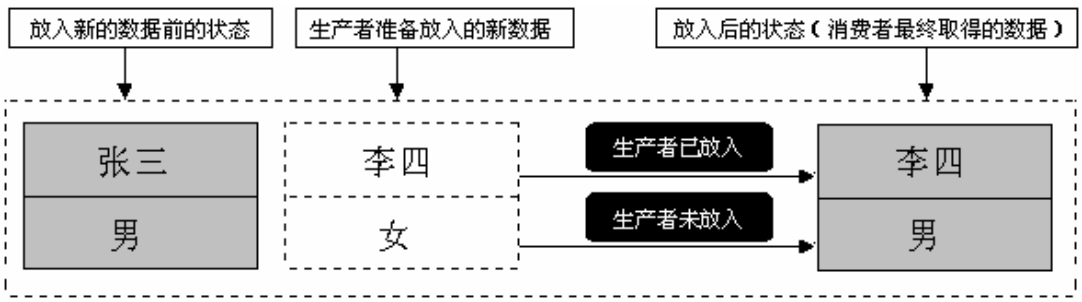


图 9-6

第二种意外：生产者放了若干次数据，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。

9.6.2 问题如何解决

下面先来构思这个程序，程序中的生产者线程和消费者线程运行的是不同的程序代码，因此这里需要编写两个包含有 run 方法的类来完成这两个线程，一个是生产者类 Producer，一个是消费者类 Consumer。

```
class Producer implements Runnable
{
   public void run()
   {
      while(true)
      {
            // 编写往数据存储空间中放入数据的代码
      }
   }
}
class Consumer implements Runnable
{
   public void run()
   {
      while(true)
      {
            // 编写从数据存储空间中读取数据的代码
      }
   }
}
```

当程序写到这里，还需要定义一个新的数据结构来作为数据存储空间。

```
class P
{
      String name;
      String sex;
```

```
}
```

Producer 和 Consumer 中的 run 函数都需要操作类 P 的同一个对象实例，接下来，对 Producer 和 Consumer 这两个类作如下修改，顺便写出程序的主调用类 ThreadCommunation。

范例：ThreadCommunation.java

```
01  class Producer implements Runnable
02  {
03      P q=null;
04      public Producer(P q)
05      {
06          this.q=q;
07      }
08      public void run()
09      {
10          int i=0;
11          while(true)
12          {
13              if(i==0)
14              {
15                  q.name="张三";
16                  q.sex="男";
17              }
18              else
19              {
20                  q.name="李四";
21                  q.sex="女";
22              }
23              i=(i+1)%2;
24          }
25      }
26  }
27  class P
28  {
```

```
29          String name="李四";
30          String sex="女";
31      }

32      class Consumer implements Runnable
33      {
34          P q=null;
35          public Consumer(P q)
36          {
37              this.q=q;
38          }
39          public void run()
40          {
41              while(true)
42              {
43                  System.out.println(q.name + " ---->" + q.sex);
44              }
45          }
46      }
47      public class ThreadCommunation
48      {
49          public static void main(String [] args)
50          {
51              P q=new P();
52              new Thread(new Producer(q)).start();
53              new Thread(new Consumer(q)).start();
54          }
55      }
```

输出结果，如图 9-7 所示：

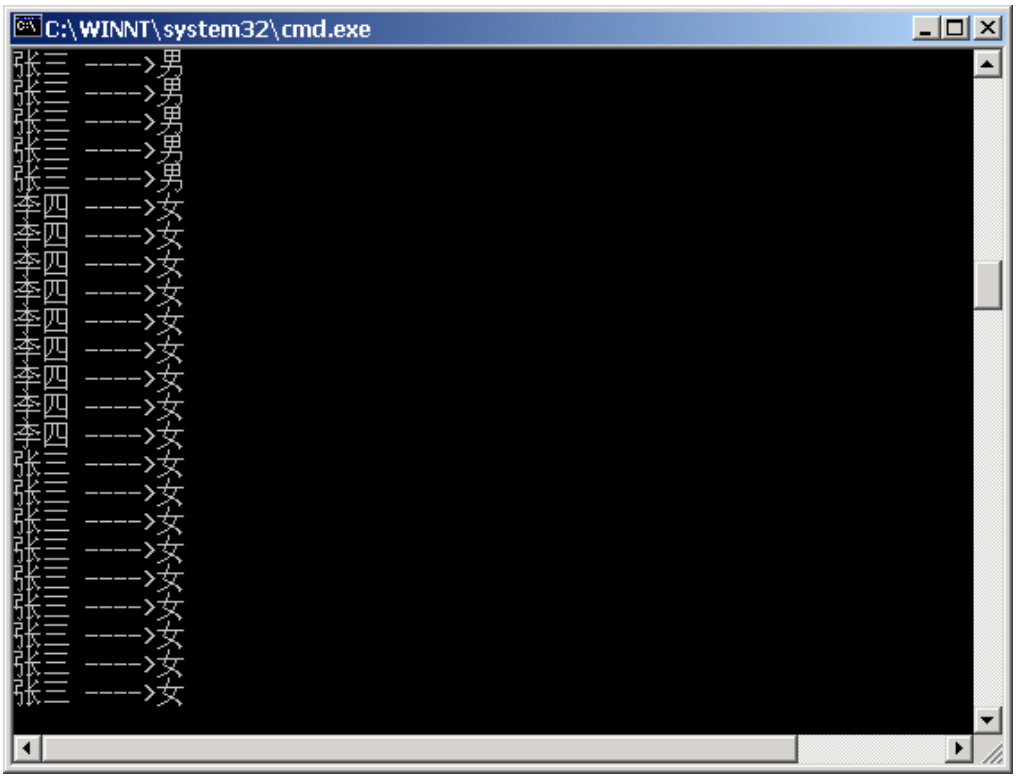


图 9-7　ThreadCommunation 输出结果

从输出结果可以发现，原本李四是女、张三是男，现在却打印出了张三是女的奇怪现象，这是什么原因呢？从程序可以发现，Producer 类和 Consumer 类都是操纵了同一个 P 类，有可能 Producer 类还未操纵完 P 类，Consumer 类就已经将 P 类中的内容取走了，这就是资源不同步的与原因，可以在 P 类中增加两个同步方法：set()和 get()，具体代码如下：

范例：ThreadCommunation.java

```
01  class Producer implements Runnable
02  {
03      P q=null;
```

```
04          public Producer(P q)
05          {
06                  this.q=q;

07          }
08          public void run()
09          {
10                  int i=0;
11                  while(true)
12                  {
13                          if(i==0)
14                          {
15                                  q.set("张三","男");
16                          }
17                          else
18                          {
19                                  q.set("李四","女");
20                          }
21                          i=(i+1)%2;
22                  }
23          }
24  }
25  class P
26  {
27          private String name="李四";
28          private String sex="女";
29          public synchronized void set(String name,String sex)
30          {
31                  this.name = name ;
32                  this.sex =sex ;
33          }
34          public synchronized void get()
35          {
36                  System.out.println(this.name + " ----->" + this.sex ) ;
37          }
38  }
39  class Consumer implements Runnable
40  {
```

```
41          P q=null;
42          public Consumer(P q)
43          {

44                  this.q=q;
45          }
46          public void run()
47          {
48                  while(true)
49                  {
50                          q.get();
51                  }
52          }
52  }
53  public class ThreadCommunation
54  {
55          public static void main(String [] args)
56          {
57                  P q=new P();
58                  new Thread(new Producer(q)).start();
59                  new Thread(new Consumer(q)).start();
60          }
61  }
```

输出结果，如图 9-8 所示：

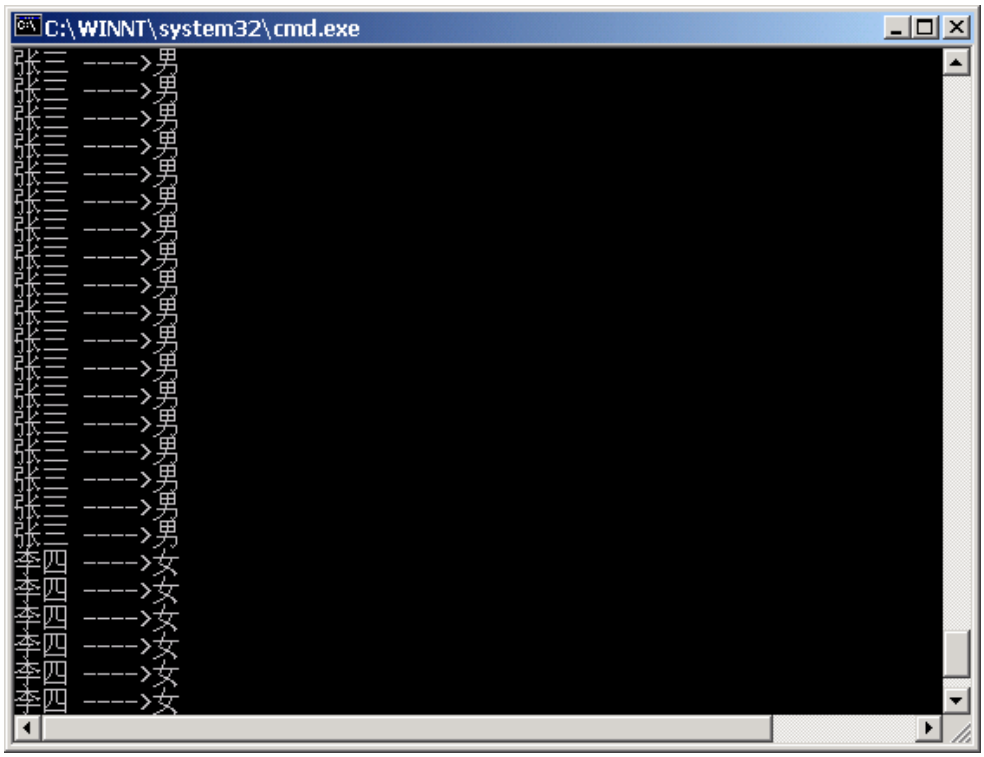


图 9-8

可以发现程序的输出结果是正确的，但是这里又有一个新的问题产生了，从程序的执行结果来看，Consumer 线程对 Producer 线程放入的一次数据连续读取了多次，并不符合实际的要求。实际要求的结果是，Producer 放一次数据，Consumer 就取一次；反之，Producer 也必须等到 Consumer 取完后才能放入新的数据，而这一问题的解决就需要使用下面所要讲到的线程间的通信。Java 是通过 Object 类的 wait、notify、notifyAll 这几个方法来实现线程间的通信的，又因为所有的类都是从 Object 继承的，所以任何类都可以直接使用这些方法。下面是这三个方法的简要说明：

wait：告诉当前线程放弃监视器并进入睡眠状态，直到其它线程进入同一监视器并调用 notify 为止。

notify：唤醒同一对象监视器中调用 wait 的第一个线程。类似排队买票，一个人买完之后，后面的人可以继续买。

notifyAll：唤醒同一对象监视器中调用 wait 的所有线程，具有最高优先级的线程首先被唤醒并执行。

如果想让上面的程序符合预先的设计需求，必须在类 P 中定义一个新的成员变量 bFull 来表示数据存储空间的状态，当 Consumer 线程取走数据后，bFull 值为 false，当 Producer 线程放入数据后，bFull 值为 true。只有 bFull 为 true 时，Consumer 线程才能取走数据，否则就必须等待 Producer 线程放入新的数据后的通知；反之，只有 bFull 为 false，Producer 线程才能放入新的数据，否则就必须等待 Consumer 线程取走数据后的通知。修改后的 P 类的程序代码如下：

```
01  class P
02  {
03      private String name="李四";
04      private String sex="女";
05      boolean bFull = false ;
06      public synchronized void set(String name,String sex)
07      {
08          if(bFull)
09          {
10              try
11              {
12                  wait() ;   // 后来的线程要等待
13              }
14              catch(InterruptedException e)
15              {}
16          }
17          this.name = name ;
18          try
19          {
20              Thread.sleep(10);
21          }
22          catch(Exception e)
23          {
24              System.out.println(e.getMessage());
25          }
26          this.sex = sex ;
27          bFull = true ;
```

```
28              notify();                       // 唤醒最先到达的线程
29          }
30          public synchronized void get()
31          {
32              if(!bFull)
33              {
34                  try
35                  {
36                      wait() ;
37                  }
38                  catch(InterruptedException e)
39                  {}
40              }
41              System.out.println(name+" ---->"+sex);
42              bFull = false ;
43              notify();
44          }
45      }
```

输出结果，如图 9-9 所示：

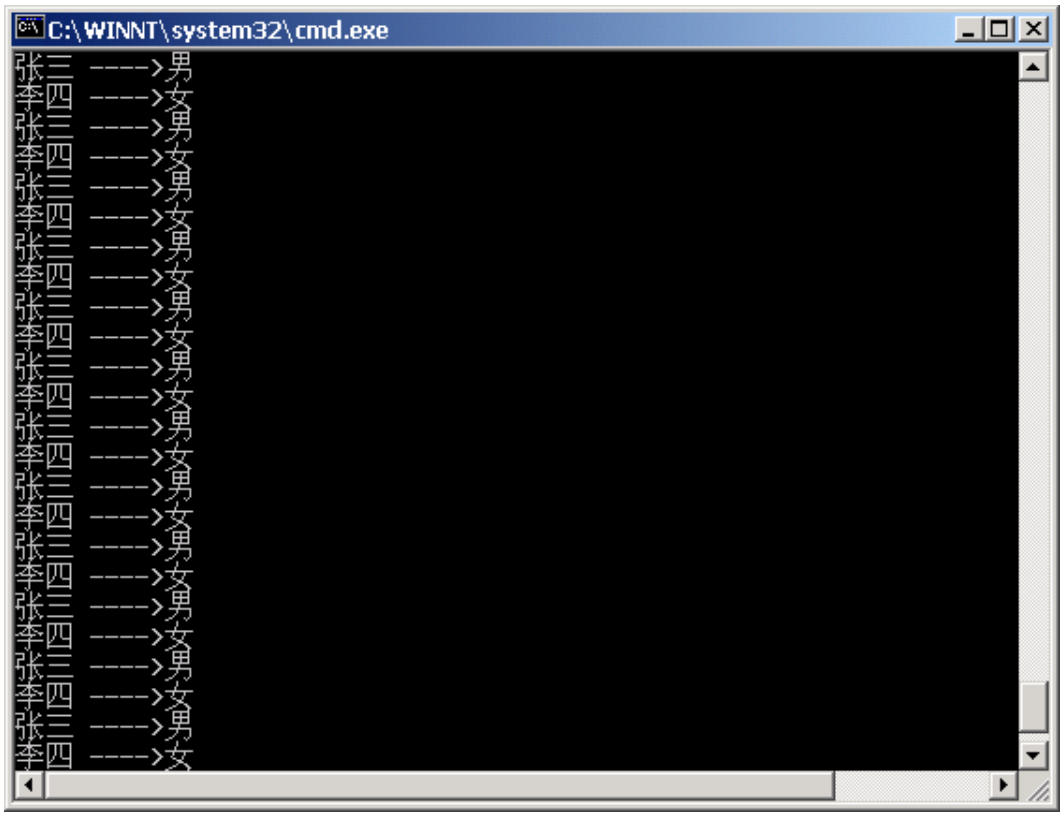


图 9-9

上面的程序满足了设计的需求，解决了线程间通信的问题。

wait、notify、notifyAll 这三个方法只能在 synchronized 方法中调用，即无论线程调用一个对象的 wait 还是 notify 方法，该线程必须先得到该对象的锁标记，这样，notify 只能唤醒同一对象监视器中调用 wait 的线程，使用多个对象监视器，就可以分别有多个 wait、notify 的情况，同组里的 wait 只能被同组的 notify 唤醒。

一个线程的等待和唤醒过程可以用图 9-10 表示：

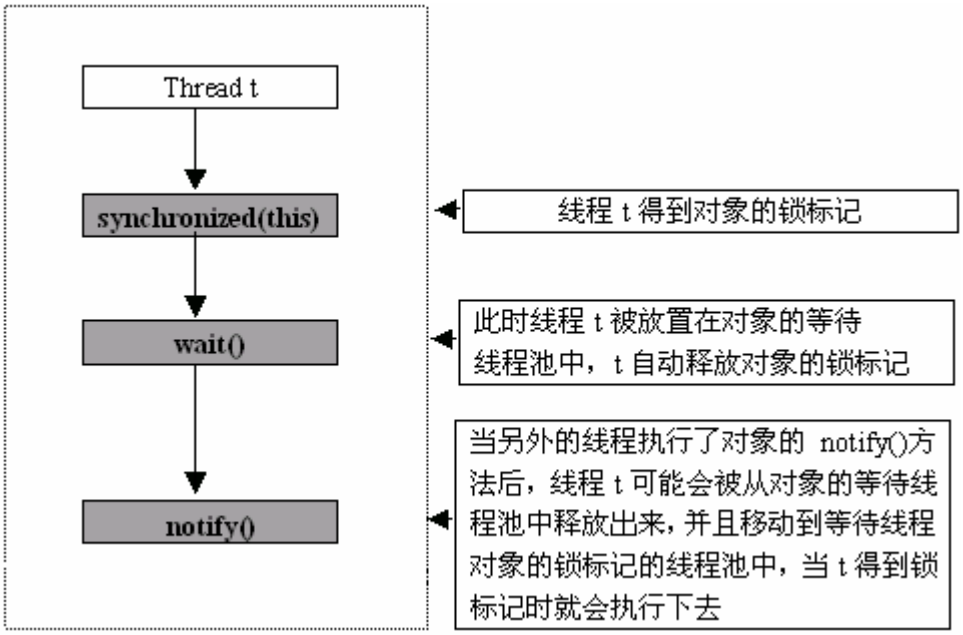


图 9-10

9.7 线程生命周期的控制

任何事务都有一个生命周期，线程也不例外。那么在一个程序中，怎样控制一个线程的生命并让它更有效地工作呢？要想控制线程的生命，先得了解线程产生和消亡的整个过程。请读者结合前面讲的内容，仔细看一看图 9-11：

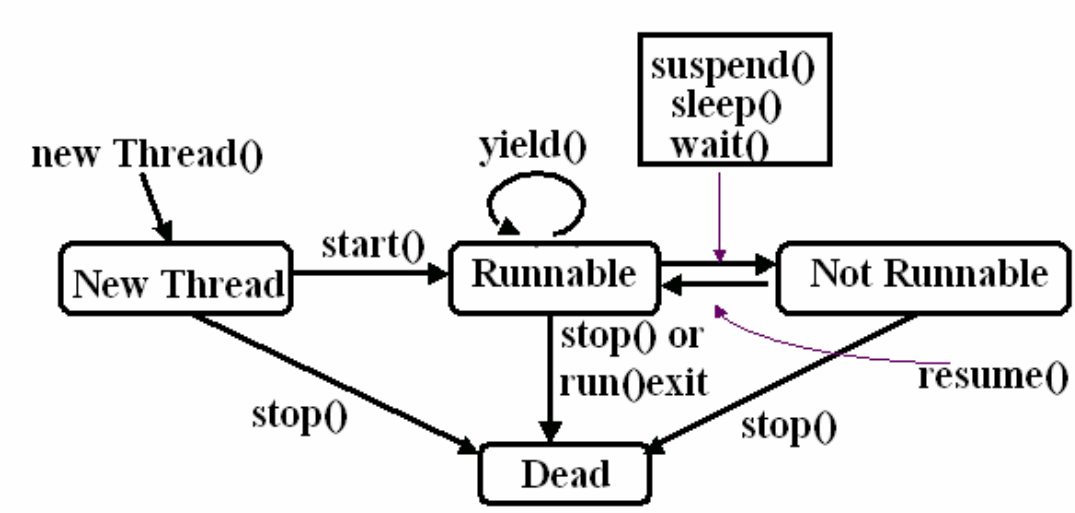


图 9-11　线程的生命周期

了解了线程的生命周期，就不难想出能够控制线程生命的办法了吧？其实，控制线程生命周期的方法有很多种，如：suspend 方法、resume 方法和 stop 方法。但这三个方法都不推荐使用，其中，不推荐使用 suspend 和 resume 的原因是：

(1)、会导致死锁的发生。

(2)、它允许一个线程（甲）通过直接控制另外一个线程（乙）的代码来直接控制那个线程（乙）。

虽然 stop 能够避免死锁的发生，但是也有其它的不足：如果一个线程正在操作共享数据段，操作过程没有完成就被“stop”了的话，将会导致数据的不完整性。因此 stop 方法也不提倡使用了！

既然这三个方法都不推荐使用，那么到底该用什么方法呢？请看下面的代码：

范例：ThreadLife.java

```
01  public class ThreadLife
02  {
03      public static void main(String [] args)
04      {
05          TestThread t=new TestThread();
06          new Thread(t).start();
```

```
07              for(int i=0;i<10; i++)
08              {
09                  if(i == 5)
10                      t.stopMe();
11                  System.out.println("Main  线程在运行");
12              }
13          }
14      }
15      class TestThread implements Runnable
16      {
17          private boolean bFlag = true;
18          public void stopMe()
19          {
20              bFlag = false;
21          }
22          public void run()
23          {
24              while(bFlag)
25              {
26                  System.out.println(Thread.currentThread().getName()+"  在运行");
27              }
28          }
29      }
```

输出结果：

```
Main  线程在运行
Thread-0  在运行
Main  线程在运行
Thread-0  在运行
Main  线程在运行
Thread-0  在运行
Main  线程在运行
Thread-0  在运行
Main  线程在运行
Thread-0  在运行
Thread-0  在运行
Main  线程在运行
```

```
Main 线程在运行
Main 线程在运行
Main 线程在运行
Main 线程在运行
```

上面的程序中定义了一个计数器 i，用来控制 main 线程的循环打印次数，在 i 的值从 0 到 4 的这段时间内，两个线程是交替运行的，但当计数器 i 的取值变为 5 的时候，程序调用了 TestThread 类的 stopMe 方法，而在 stopMe 方法中，将 bFlag 变量赋值为 false，也就是终止了 while 循环，run 方法结束，Thread-0 线程随之结束。main 线程在计数器 i 等于 5 的时候，调用了 TestThread 类的 stopMe 方法后，CPU 不一定会马上切换到 Thread-0 线程上，也就是说 Thread-0 线程不一定会马上终止，main 线程的计数器 i 可能还会继续累加后，Thread-0 线程才真正结束。

综上所述，通过控制 run 方法中循环条件的方式来结束一个线程的方法是推荐读者使用的，这也是实际中用的最多的方法。

• 本章摘要：

1、 线程（thread）是指程序的运行流程。“多线程”的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、 如果在类里要激活线程，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 类里的一个方法，因此把线程的程序代码编写在 run()方法内，所做的就是覆盖的操作。

4、 Runnable 接口里声明了抽象的 run()方法，因此必须在实现 Runnable 接口的类里明确定义 run()方法。

5、 每一个线程，在其创建和消亡之前，均会处于下列五种状态之一：创建、就绪、运行、阻塞、终止。

6、 暂停状态的线程可由下列的情况所产生：

（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：

（1）如果线程是由调用对象的 wait()方法所冻结，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，或是由线程调用它的 stop()方法时，则线程进入消亡状态。

9、 Thread 类里的 sleep()方法可用来控制线程的休眠状态，休眠的时间要视 sleep()里的参数而定。

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。

第 10 章　文件（IO）操作

大多数的应用程序都需要与外部设备进行数据交换，最常见的外部设备包含磁盘和网络。IO 就是指应用程序对这些设备的数据输入与输出，在程序中，键盘被用作文件输入，显示器被用作文件输出。JAVA 语言定义了许多类专门负责各种方式的输入输出，这些类都被放在 java.io 包中。

10.1　File 类

File 类是 IO 包中唯一代表磁盘文件本身的对象，File 类定义了一些与平台无关的方法来操纵文件，通过调用 File 类提供的各种方法，能够完成创建、删除文件，重命名文件，判断文件的读写权限及是否存在，设置和查询文件的最近修改时间等操作。

Java 能正确处理 UNIX 和 Windows/DOS 约定路径分隔符。如果在 Windows 版本的 Java 下用斜线（/），路径处理依然正确。记住，如果使用 Windows/DOS 使用反斜线（\）的约定，就需要在字符串内使用它的转义序列（\\）。Java 约定是用 UNIX 和 URL 风格的斜线来作路径分隔符。

下面的构造方法可以用来生成 File 对象：

File(String directoryPath)

这里，directoryPath 是文件的路径名。

File 定义了很多获取 File 对象标准属性的方法。例如：getName( )用于返回文件名，getParent( )返回父目录名，exists( )方法在文件存在的情况下返回 true，反之返回 false。然而 File 类是不对称的，意思是虽然存在可以验证一个简单文件对象属性的很多方法，但是没有相应的方法来改变这些属性。下面的例子说明了 File 的几个方法：

范例：FileDemo.java

```
01  import java.io.*;
02  public class FileDemo
03  {
```

```
04      public static void main(String[] args)
05      {
06          File f=new File("c:\\1.txt");
07          if(f.exists())
08              f.delete();
09          else
10              try
11              {
12                  f.createNewFile();
13              }
14              catch(Exception e)
15              {
16                  System.out.println(e.getMessage());
17              }
18          // getName()方法，取得文件名
19          System.out.println("文件名："+f.getName());
20          // getPath()方法，取得文件路径
21          System.out.println("文件路径："+f.getPath());
22          // getAbsolutePath()方法，得到绝对路径名
23          System.out.println("绝对路径："+f.getAbsolutePath());
24          // getParent()方法，得到父文件夹名
25          System.out.println("父文件夹名称："+f.getParent());
26          // exists()，判断文件是否存在
27          System.out.println(f.exists()?"文件存在":"文件不存在");
28          // canWrite()，判断文件是否可写
29          System.out.println(f.canWrite()?"文件可写":"文件不可写");
30          // canRead()，判断文件是否可读
31          System.out.println(f.canRead()?"文件可读":"文件不可读");
32          /// isDirectory()，判断是否是目录
33          System.out.println(f.isDirectory()?"是":"不是"+"目录");
34          // isFile()，判断是否是文件
35          System.out.println(f.isFile()?"是文件":"不是文件");
36          // isAbsolute()，是否是绝对路径名称
37          System.out.println(f.isAbsolute()?"是绝对路径":"不是绝对路径");
38          // lastModified()，文件最后的修改时间
39          System.out.println("文件最后修改时间："+f.lastModified());
40          // length()，文件的长度
```

```
41                 System.out.println("文件大小："+f.length()+" Bytes");
42         }
43   }
```

输出结果：

```
文件名：1.txt
文件路径：c:\1.txt
绝对路径：c:\1.txt
父文件夹名称：c:\
文件存在
文件可写
文件可读
不是目录
是文件
是绝对路径
文件最后修改时间：1122277108000
文件大小：0 Bytes
```

在 File 类中还有许多的方法，读者没有必要去死记这些用法，只要记住在需要的时候去查 Java 的 API 手册就可以了。

10-2 RandomAccessFile 类

RandomAccessFile 类可以说是 Java 语言中功能最为丰富的文件访问类，它提供了众多的文件访问方法。RandomAccessFile 类支持“随机访问”方式，可以跳转到文件的任意位置处读写数据。在要访问一个文件的时候，不想把文件从头读到尾，而是希望像访问一个数据库一样地访问一个文本文件，这时，使用 RandomAccessFile 类就是最佳选择。

RandomAccessFile 对象类有个位置指示器，指向当前读写处的位置，当读写 n 个字节后，文件指示器将指向这 n 个字节后的下一个字节处。刚打开文件时，文件指示器指向文件的开头处，可以移动文件指示器到新的位置，随后的读写操作将从新的位置开始。RandomAccessFile 在数据等长记录格式文件的随机（相对顺序而言）读取时

有很大的优势，但该类仅限于操作文件，不能访问其它的 IO 设备，如网络、内存映像等。

有关 RandomAccessFile 类中的成员方法及使用说明请参阅 JDK 文档。下面是一个使用 RandomAccessFile 的例子，往文件中写入三名员工的信息，然后按照第二名员工，第一名员工，第三名员工的先后顺序读出。RandomAccessFile 可以以只读或读写方式打开文件，具体使用哪种方式取决于用户创建 RandomAccessFile 类对象的构造方法：

```
new RandomAccessFile(f,"rw");          // 读写方式
new RandomAccessFile(f,"r");           // 只读方式
```

注意：

当程序需要以读写的方式打开一个文件时，如果这个文件不存在，程序会自动创建此文件。

这里还需要设计一个类来封装员工信息。一个员工信息就是文件中的一条记录，而且必须保证每条记录在文件中的大小相同，也就是每个员工的姓名字段在文件中的长度是一样的，这样才能够准确定位每条记录在文件中的具体位置。假设 name 中有 8 个字符，少于 8 个则补空格(这里用"\u0000")，多于 8 个则去掉后面多余的部分。由于年龄是整型数，不管这个数有多大，只要它不超过整型数的范围，在内存中都是占 4 个字节大小。

范例：RandomFileDemo.java

```
01  import java.io.*;
02  public class RandomFileDemo
03  {
04      public static void main(String [] args) throws Exception
05      {
06          Employee e1 = new Employee("zhangsan",23);
07          Employee e2 = new Employee("lisi",24);
08          Employee e3 = new Employee("wangwu",25);
09          RandomAccessFile ra=new RandomAccessFile("c:\\employee.txt","rw");
10          ra.write(e1.name.getBytes());
```

```
11              ra.writeInt(e1.age);
12              ra.write(e2.name.getBytes());

13              ra.writeInt(e2.age);
14              ra.write(e3.name.getBytes());
15              ra.writeInt(e3.age);
16              ra.close();
17              RandomAccessFile raf=new RandomAccessFile("c:\\employee.txt","r");
18              int len=8;
19              raf.skipBytes(12);  // 跳过第一个员工的信息,其姓名 8 字节,年龄 4 字节
20              System.out.println("第二个员工信息:");
21              String str="";
22              for(int i=0;i<len;i++)
23                  str=str+(char)raf.readByte();
24              System.out.println("name:"+str);
25              System.out.println("age:"+raf.readInt());
26              System.out.println("第一个员工的信息:");
27              raf.seek(0);            // 将文件指针移动到文件开始位置
28              str="";
29              for(int i=0;i<len;i++)
30                  str=str+(char)raf.readByte();
31              System.out.println("name:"+str);
32              System.out.println("age:"+raf.readInt());
33              System.out.println("第三个员工的信息:");
34              raf.skipBytes(12);  // 跳过第二个员工信息
35              str="" ;
36              for(int i=0;i<len;i++)
37                  str=str+(char)raf.readByte();
38              System.out.println("name:"+str.trim());
39              System.out.println("age:"+raf.readInt());
40              raf.close();
41          }
42  }
43  class Employee
44  {
45          String name;
46          int age;
47          final static int LEN=8;
```

```
48          public Employee(String name,int age)
49          {

50                  if(name.length()>LEN)
51                  {
52                          name=name.substring(0,8);
53                  }
54                  else
55                  {
56                          while(name.length()<LEN)
57                                  name=name+"\u0000";
58                  }
59                  this.name=name;
60                  this.age=age;
61          }
62  }
```

输出结果：

第二个员工信息:

name:lisi

age:24

第一个员工的信息:

name:zhangsan

age:23

第三个员工的信息:

name:wangwu

age:25

上面的这个程序完成了所要实现的功能，显示出了 RandomAccessFile 类的作用。String.substring(int beginIndex,int endIndex)方法可以用于取出一个字符串中的部分子字符串，但要注意的一个细节是：子字符串中的第一个字符对应的是原字符串中的脚标为 beginIndex 处的字符，但最后的字符对应的是原字符串中的脚标为 endIndex-1 处的字符，而不是 endIndex 处的字符。

10-3 流类

Java 的流式输入/输出建立在四个抽象类的基础上：InputStream, OutputStream, Reader 和 Writer。它们用来创建具体流式子类。尽管程序通过具体子类执行输入/输出操作，但顶层的类定义了所有流类的基本通用功能。

InputStream 和 OutputStream 设计成字节流类。Reader 和 Writer 为字符流设计。字节流类和字符流类形成分离的层次结构。一般说来，处理字符或字符串时应使用字符流类，处理字节或二进制对象时应用字节流类。

一般在操作文件流时，不管是字节流还是字符流都可以按照以下的方式进行：

1、 使用 File 类找到一个文件

2、 通过 File 类的对象去实例化字节流或字符流的子类

3、 进行字节（字符）的读、写操作

4、 关闭文件流

下面分别讲述字节流和字符流类。

10.3.1 字节流

字节流类为处理字节式输入/输出提供了丰富的环境。一个字节流可以和其它任何类型的对象并用，包括二进制数据。这样的多功能性使得字节流对很多类型的程序都很重要。因为字节流类以 InputStream 和 OutputStream 为顶层，就从讨论这两个类开始。

10.3.1.1 InputStream（输入字节流）

InputStream 是一个定义了 Java 流式字节输入模式的抽象类。该类的所有方法在

出错条件下都会引发一个 IOException 异常。表 10-1 显示了 InputStream 的方法：

表10-1 InputStream 定义的方法

方法	描述
int available( )	返回当前可读的输入字节数
void close( )	关闭输入流。关闭之后若再读取则会产生IOException异常
void mark(int numBytes)	在输入流的当前点放置一个标记。该流在读取N个Bytes字节前都保持有效
boolean markSupported( )	如果调用的流支持mark( )/reset( )就返回true
int read( )	如果下一个字节可读则返回一个整型，遇到文件尾时返回-1
int read(byte buffer[ ])	试图读取buffer.length个字节到buffer中，并返回实际成功读取的字节数。遇到文件尾时返回-1
int read(byte buffer[ ], int offset,int numBytes)	试图读取buffer中从buffer[offset]开始的numBytes个字节，返回实际读取的字节数。遇到文件结尾时返回-1
void reset( )	重新设置输入指针到先前设置的标志处
long skip(long numBytes)	忽略numBytes个输入字节，返回实际忽略的字节数

10.3.1.2 OutputStream（输出字节流）

OutputStream是定义了流式字节输出模式的抽象类。该类的所有方法返回一个void值并且在出错情况下引发一个IOException异常。表10-2显示了OutputStream的方法。

表10-2 OutputStream 定义的方法

方法	描述
void close( )	关闭输出流。关闭后的写操作会产生IOException异常
void flush( )	定制输出状态以使每个缓冲器都被清除，也就是刷新输出缓冲区
void write(int b)	向输出流写入单个字节。注意参数是一个整型数，它允许设计者不必把参数转换成字节型就可以调用write（）方法
void write(byte buffer[ ])	向一个输出流写一个完整的字节数组
void write(byte buffer[ ], int offset,int numBytes)	写数组buffer以buffer[offset]为起点的numBytes个字节区域内的内容

注意：

上两个表中的多数方法由 InputStream 和 OutputStream 的子类来实现，但 mark( )和 reset( )方法除外。注意下面讨论的每个子类中这些方法的使用和不用的情况。

10.3.1.3 FileInputStream（文件输入流）

FileInputStream 类创建一个能从文件读取字节的InputStream 类，它的两个常用的构造方法如下：

FileInputStream(String filepath)

FileInputStream(File fileObj)

这两个构造方法都能引发FileNotFoundException异常。这里，filepath 是文件的绝对路径，fileObj是描述该文件的File对象。

下面的例子创建了两个使用同样磁盘文件且各含一个上面所描述的构造方法的FileInputStreams类：

```
InputStream f0 = new FileInputStream("c:\\test.txt") ;
File f = new File("c:\\test.txt");
InputStream f1 = new FileInputStream(f);
```

尽管第一个构造方法可能更常用到，而第二个构造方法则允许在把文件赋给输入流之前用File方法更进一步检查文件。当一个FileInputStream被创建时，它可以被公开读取。

10.3.1.4 FileOutputStream（文件输出流）

FileOutputStream 创建了一个可以向文件写入字节的类 OutputStream，它常用的

构造方法如下：

FileOutputStream(String filePath)

FileOutputStream(File fileObj)

FileOutputStream(String filePath, boolean append)

它们可以引发 IOException 或 SecurityException 异常。这里 filePath 是文件的绝对路径，fileObj 是描述该文件的 File 对象。如果 append 为 true，文件则以设置搜索路径模式打开。FileOutputStream 的创建不依赖于文件是否存在。在创建对象时，FileOutputStream 会在打开输出文件之前就创建它。这种情况下如果试图打开一个只读文件，会引发一个 IOException 异常。

在下面的例子中，用 FileOutputStream 类向文件中写入一字符串，并用 FileInputStream 读出写入的内容。

范例：StreamDemo.java

```
01  import java.io.*;
02  public class StreamDemo
03  {
04      public static void main(String args[])
05      {
06          File f = new File("c:\\temp.txt") ;
07          OutputStream out = null ;
08          try
09          {
10              out = new FileOutputStream(f) ;
11          }
12          catch (FileNotFoundException e)
13          {
14              e.printStackTrace();
15          }
16          // 将字符串转成字节数组
17          byte b[] = "Hello World!!!".getBytes() ;
18          try
19          {
20              // 将 byte 数组写入到文件之中
```

```
21              out.write(b) ;
22          }
23          catch (IOException e1)
24          {
25              e1.printStackTrace();
26          }
27          try
28          {
29              out.close() ;
30          }
31          catch (IOException e2)
32          {
33              e2.printStackTrace();
34          }
35
36          // 以下为读文件操作
37          InputStream in = null ;
38          try
39          {
40              in = new FileInputStream(f) ;
41          }
42          catch (FileNotFoundException e3)
43          {
44              e3.printStackTrace();
45          }
46          // 开辟一个空间用于接收文件读进来的数据
47          byte b1[] = new byte[1024] ;
48          int i = 0 ;
49          try
50          {
51          // 将 b1 的引用传递到 read()方法之中，同时此方法返回读入数据的个数
52              i = in.read(b1) ;
53          }
54          catch (IOException e4)
55          {
56              e4.printStackTrace();
57          }
58          try
```

```
59              {
60                  in.close() ;
61              }
62              catch (IOException e5)
63              {
64                  e5.printStackTrace();
65              }
66              //将 byte 数组转换为字符串输出
67              System.out.println(new String(b1,0,i)) ;
68          }
69      }
```

输出结果：

Hello World!!!

程序说明：

此程序分为两个部分，一部分是向文件中写入内容（第 8 行~第 34 行），另一部分是从文件中读取内容（第 36 行~第 65 行）：

1、 程序第 6 行通过一个 File 类找到 C 盘下的一个 temp.txt 文件。

2、 向文件写入内容：

（1）、 第 8 行~第 15 行通过 File 类的对象去实例化 OutputStream 的对象，此时是通过其子类 FileOutputStream 实例化的 OutputStream 对象，属于对象的向上转型。

（2）、 因为字节流主要以操作 byte 数组为主，所以程序第 17 行通过 String 类中的 getBytes()方法，将字符串转换成一 byte 数组。

（3）、 第 18 行~第 26 行调用 OutputStream 类中的 write()方法将 byte 数组中的内容写入到文件中。

（4）、 第 27 行~第 34 行调用 OutputStream 类中的 close()方法，关闭数据流操作。

3、 从文件中读入内容：

（1）、 第 37 行~第 45 行通过 File 类的对象去实例化 InputStream 的对象，此时是通过其子类 FileInputStream 实例化的 InputStream 对象，属于对象的向上转型。

（2）、 因为字节流主要以操作 byte 数组为主，所以程序第 47 行声明一 1024 大小的 byte 数组，此数组用于存放读入的数据。

（3）、 第 49 行~第 57 行调用 InputStream 类中的 read()方法将文件中的内容读入到 byte 数组中，同时返回读入数据的个数。

（4）、 第 58 行~第 65 行调用 InputStream 类中的 close()方法，关闭数据流操作。

（5）、 第 67 行将 byte 数组转成字符串输出。

从上面这道范例中读者可以发现，大部分的方法操作时都进行了异常处理，这是因为所使用的方法处都用 throws 关键字抛出了异常，所以这里需要进行异常捕捉，不清楚的读者可以查找 JDK 文档，相信就可以明白了。

10.3.2 字符流

尽管字节流提供了处理任何类型输入/输出操作的足够的功能，但它们不能直接操作 Unicode 字符。既然 Java 的一个主要目标是支持“一次编写，处处运行”，包含直接的字符输入 / 输出的支持是必要的。本节将讨论几个字符输入 / 输出类。如前所述，字符流层次结构的顶层是 Reader 和 Writer 抽象类，将从它们开始介绍。

10.3.2.1 Reader

Reader是定义Java的流式字符输入模式的抽象类。该类的所有方法在出错情况下都将引发IOException 异常。

表10-3给出了Reader类中的方法。

表10-3　Reader 定义的方法

方法	描述
abstract void close( )	关闭输入源。进一步的读取将会产生IOException异常
void mark(int numChars)	在输入流的当前位置设立一个标志。该输入流在numChars个字符被读取之前有效
boolean markSupported( )	该流支持mark( )/reset( )则返回true
int read( )	如果调用的输入流的下一个字符可读则返回一个整型。遇到文件尾时返回-1
int read(char buffer[ ])	试图读取buffer中的buffer.length个字符，返回实际成功读取的字符数。遇到文件尾返回-1
abstract int read(char buffer[ ],int offset,int numChars)	试图读取buffer中从buffer[offset]开始的numChars个字符，返回实际成功读取的字符数。遇到文件尾返回-1
boolean ready( )	如果下一个输入请求不等待则返回true，否则返回false
long skip(long numChars)	跳过numChars个输入字符，返回跳过的字符设置输入指针到先前设立的标志处数

10.3.2.2　Writer

Writer 是定义流式字符输出的抽象类。所有该类的方法都返回一个void 值并在出错条件下引发IOException 异常。表10-4给出了Writer类中方法。

表10-4　Writer 定义的方法

方法	描述
abstract void close( )	关闭输出流。关闭后的写操作会产生IOException异常
abstract void flush( )	定制输出状态以使每个缓冲器都被清除。也就是刷新输出缓冲
void write(int ch)	向输出流写入单个字符。注意参数是一个整型，它允许设计者不必把参数转换成字符型就可以调用write（）方法
void write(char buffer[ ])	向一个输出流写一个完整的字符数组
abstract void write(char buffer[ ],int offset, int numChars)	向调用的输出流写入数组buffer 以buffer[offset] 为起点的N个Chars区域内的内容
void write(String str)	向调用的输出流写str
void write(String str, int offset,int numChars)	写数组str中以制定的offset为起点的长度为numChars个字符区域内的内容

10.3.2.3 FileReader

FileReader 类创建了一个可以读取文件内容的 Reader 类。它最常用的构造方法显示如下：

FileReader(String filePath)

FileReader(File fileObj)

每一个都能引发一个 FileNotFoundException 异常。这里，filePath 是一个文件的完整路径，fileObj 是描述该文件的 File 对象。

10.3.2.4 FileWriter

FileWriter 创建一个可以写文件的 Writer 类。它最常用的构造方法如下：

FileWriter(String filePath)

FileWriter(String filePath, boolean append)

FileWriter(File fileObj)

它们可以引发 IOException 或 SecurityException 异常。这里，filePath 是文件的绝对路径，fileObj 是描述该文件的 File 对象。如果 append 为 true，输出是附加到文件尾的。FileWriter 类的创建不依赖于文件存在与否。在创建文件之前，FileWriter 将在创建对象时打开它来作为输出。如果试图打开一个只读文件，将引发一个 IOException 异常。

下面的例子是将上面的例题进行改写，先来看一下代码：

范例：CharDemo.java

```
01  import java.io.*;
02  public class CharDemo
03  {
04      public static void main(String args[])
05      {
06          File f = new File("c:\\temp.txt") ;
07          Writer out = null ;
08          try
09          {
10              out = new FileWriter(f) ;
11          }
12          catch (IOException e)
13          {
14              e.printStackTrace();
15          }
16          // 声明一个 String 类型对象
17          String str = "Hello World!!!" ;
18          try
19          {
20              // 将 str 内容写入到文件之中
21              out.write(str) ;
22          }
```

```
23          catch (IOException e1)
24          {
25              e1.printStackTrace();
26          }
27          try
28          {
29              out.close() ;
30          }
31          catch (IOException e2)
32          {
33              e2.printStackTrace();
34          }
35
36          // 以下为读文件操作
37          Reader in = null ;
38          try
39          {
40              in = new FileReader(f) ;
41          }
42          catch (FileNotFoundException e3)
43          {
44              e3.printStackTrace();
45          }
46          // 开辟一个空间用于接收文件读进来的数据
47          char c1[] = new char[1024] ;
48          int i = 0 ;
49          try
50          {
51          // 将 c1 的引用传递到 read()方法之中，同时此方法返回读入数据的个数
52              i = in.read(c1) ;
53          }
54          catch (IOException e4)
55          {
56              e4.printStackTrace();
57          }
58          try
59          {
```

```
60                  in.close() ;
61              }
62              catch (IOException e5)
63              {
64                  e5.printStackTrace();
65              }
66              //将字符数组转换为字符串输出
67              System.out.println(new String(c1,0,i)) ;
68          }
69      }
```

输出结果：

Hello World!!!

程序说明：

此程序与上面范例类似，也同样分为两部分，一部分是向文件中写入内容（第 8 行~第 34 行），另一部分是从文件中读取内容（第 36 行~第 65 行）：

1、 程序第 6 行通过一个 File 类找到 C 盘下的一个 temp.txt 文件。

2、 向文件写入内容：

（1）、 第 7 行~第 15 行通过 File 类的对象去实例化 Writer 的对象，此时是通过其子类 FileWriter 实例化的 Writer 对象，属于对象的向上转型。

（2）、 因为字符流主要以操作字符为主，所以程序第 17 行声明一 String 类的对象 str。

（3）、 第 18 行~第 26 行调用 Writer 类中的 write()方法将字符串中的内容写入到文件中。

（4）、 第 27 行~第 34 行调用 Writer 类中的 close()方法，关闭数据流操作。

3、 从文件中读入内容：

（1）、 第 37 行~第 45 行通过 File 类的对象去实例化 Reader 的对象，此时是通过其子类 FileReader 实例化的 Reader 对象，属于对象的向上转型。

（2）、 因为字节流主要以操作 char 数组为主，所以程序第 47 行声明一 1024 大小的 char 数组，此数组用于存放读入的数据。

（3）、 第 49 行~第 57 行调用 Reader 类中的 read()方法将文件中的内容读入到 char

数组中，同时返回读入数据的个数。

（4）、 第 58 行~第 65 行调用 Reader 类中的 close()方法，关闭数据流操作。

（5）、 第 67 行将 char 数组转成字符串输出。

小提示：

读者可以将范例 CharDemo 中第 27 行到第 34 行注释掉，也就是说在向文件写入内容之后不关闭文件，这时再打开文件，可以发现文件中没有任何内容，这是为什么呢？从 JDK 文档之中查找 FileWriter 类，如图 10-1 所示：

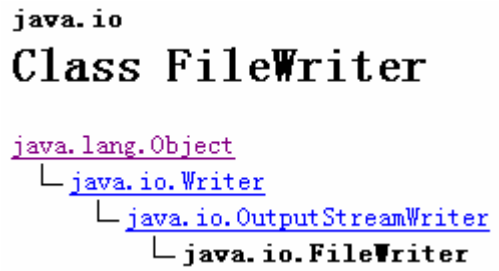


图 10-1 FileWriter 类

从图中可以发现，FileWriter 类并不是直接继承自 Writer 类，而是继承了 Writer 的子类（OutputStreamWriter）此类为字节流和字符流的转换类，本书后面会有介绍，也就说真正从文件中读取进来的数据还是字节，只是在内存中将字节转换成了字符，所以得出一个结论，字符流用到了缓冲区，而字节流没有用到缓冲区。另外也可以用 Writer 类中的 flush()方法，强制清空缓冲区。

10.3.3 管道流

管道流主要作用是可以连接两个线程间的通信。管道流也分为字节流（PipedInputStream、PipedOutputStream）与字符流（PipedReader、PipedWriter）两种类型，本节主要讲解 PipedInputStream 与 PipedOutputStream。

一个 PipedInputStream 对象必须和一个 PipedOutputStream 对象进行连接而产生一个通信管道，PipedOutputStream 可以向管道中写入数据，PipedInputStream 可以从管

道中读取 PipedOutputStream 写入的数据。如图 10-2 所示，这两个类主要用来完成线程之间的通信，一个线程的 PipedInputStream 对象能够从另外一个线程的 PipedOutputStream 对象中读取数据。请看下面的范例。

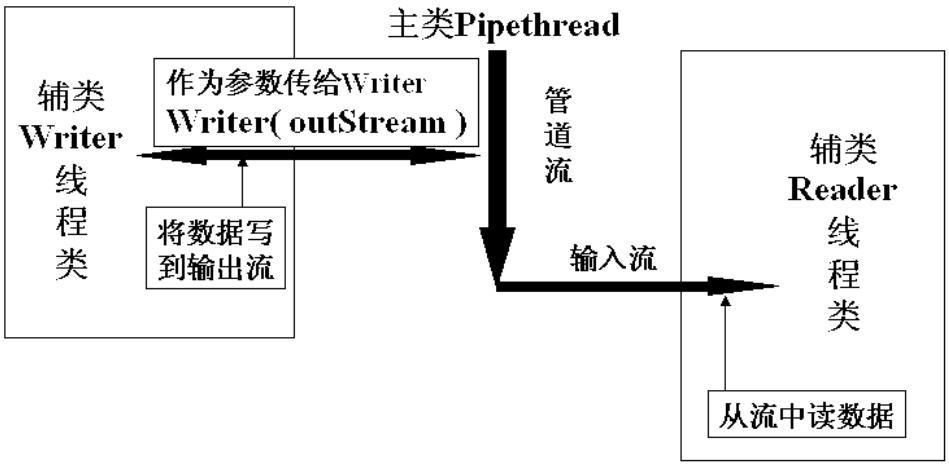


图 10-2　　管道流

范例：PipeStreamDemo.java

```
01  import java.io.*;
02  public class PipeStreamDemo
03  {
04      public static void main(String args[])
05      {
06          try
07          {
08              Sender sender = new Sender();       // 产生两个线程对象
09              Receiver receiver = new Receiver();
10              PipedOutputStream out = sender.getOutputStream();   // 写入
11              PipedInputStream in = receiver.getInputStream();       // 读出
12              out.connect(in);       // 将输出发送到输入
13              sender.start();        // 启动线程
14              receiver.start();
15          }
16          catch(IOException e)
17          {
```

```
18                  System.out.println(e.getMessage());
19              }
20          }
21  }
22  class Sender extends Thread
23  {
24          private PipedOutputStream out=new PipedOutputStream();
25          public PipedOutputStream getOutputStream()
26          {
27              return out;
28          }
29          public void run()
30          {
31              String s=new String("Receiver，你好!");
32              try
33              {
34                  out.write(s.getBytes());        // 写入（发送）
35                  out.close();
36              }
37              catch(IOException e)
38              {
39                  System.out.println(e.getMessage());
40              }
41          }
42  }
43  class Receiver extends Thread
44  {
45          private PipedInputStream in=new PipedInputStream();
46          public PipedInputStream getInputStream()
47          {
48              return in;
49          }
50          public void run()
51          {
52              String s=null;
53              byte [] buf = new byte[1024];
54              try
```

```
55              {
56                  int len =in.read(buf);              // 读出数据
57                  s = new String(buf,0,len);
58                  System.out.println("收到了以下信息："+s);
59                  in.close();
60              }
61              catch(IOException e)
62              {
63                  System.out.println(e.getMessage());
64              }
65          }
66  }
```

输出结果：

收到了以下信息：Receiver，你好!

程序说明：

1、 程序第 22~42 行声明一 Sender 类，此类继承了 Thread 类，所以此类复写了 Runnable 接口之中的 run()方法。程序第 24 行声明一 PipedOutputStream 对象 out，此对象用于发送信息。

2、 程序第 43~66 行声明一 Receiver 类，此类继承了 Thread 类，所以此类复写了 Runnable 接口之中的 run()方法。程序第 45 行声明了一 PipedInputStream 对象 in，此对象用于接收其它线程发来的信息。

3、 程序第 8 行和第 9 行分别声明了 Sender 和 Receiver 的实例化对象，之后返回各自的管道输出流及管道输入流对象，通过管道输出流的 connect 方法，将两个管道连接在一起。之后分别启动线程。

10.3.4 ByteArrayInputStream 与 ByteArrayOutputStream

ByteArrayInputStream 是输入流的一种实现，它有两个构造函数，每个构造函数都需要一个字节数组来作为其数据源：

ByteArrayInputStream(byte[] buf)
ByteArrayInputStream(byte[] buf,int offse , int length)
ByteArrayOutputStream()
BuyteArrayoutputStream(int)

如果程序在运行过程中要产生一些临时文件，可以采用虚拟文件方式实现，JDK 中提供了 ByteArrayInputStream 和 ByteArrayOutputStream 两个类可实现类似于内存虚拟文件的功能。

范例：ByteArrayDemo.java

```
01  import java.io.* ;
02  public class ByteArrayDemo {
03      public static void main(String[] args) throws Exception
04      {
05          String tmp = "abcdefghijklmnopqrstuvwxyz";
06          byte[] src = tmp.getBytes();              // src 为转换前的内存块
07          ByteArrayInputStream input = new ByteArrayInputStream(src);
08          ByteArrayOutputStream output = new ByteArrayOutputStream();
09          new ByteArrayDemo().transform(input, output);
10          byte[] result = output.toByteArray();    // result 为转换后的内存块
11          System.out.println(new String(result));
12      }
13      public void transform(InputStream in, OutputStream out)
14      {
15          int c = 0;
16          try
17          {
18              while ((c = in.read()) != -1)         // read 在读到流的结尾处返回-1
19              {
20                  int C = (int) Character.toUpperCase((char) c);

21                  out.write(C);
```

```
22                  }
23              }
24              catch (Exception e)
25              {
26                  e.printStackTrace();
27              }
28          }
29  }
```

10.3.5 System.in 和 System.out

为了支持标准输入输出设备，Java 定义了两个特殊的流对象：System.in 和 System.out。System.in 对应键盘，是 InputStream 类型的，程序使用 System.in 可以读取从键盘上输入的数据；System.out 对应显示器，是 PrintStream 类型的，PrintStream 是 OutputStream 的一个子类，程序使用 System.out 可以将数据输出到显示器上。键盘可以被当作一个特殊的输入流，显示器可以被当作一个特殊的输出流。

10.3.6 打印流

PrintStream 类提供了一系列的 print 和 println 方法，可以实现将基本数据类型的格式转化成字符串输出。在前面的程序中大量用到“System.out.println”语句中的 System.out 就是 PrintStream 类的一个实例对象。PrintStream 有下面几个构造方法：

PrintStream(OutputStream out)

PrintStream(OutputStream out,boolean auotflush)

PrintStream(OutputStream out,boolean auotflush, String encoding)

其中 autoflush 控制在 Java 中遇到换行符(\n)时是否自动清空缓冲区，encoding 是指定编码方式。关于编码方式，本章在后面有详细的介绍。

println 方法与 print 方法的区别是：前者会在打印完的内容后再多打印一个换行符(\n)，所以 println()等于 print("\n")。

Java 的 PrintStream 对象具有多个重载的 print 和 println 方法，它们可输出各种类型（包括 Object）的数据。对于基本数据类型的数据，print 和 println 方法会先将它们转换成字符串的形式然后再输出，而不是输出原始的字节内容，如：整数 221 的打印结果是字符‘2’、‘2’、‘1’所组合成的一个字符串，而不是整数 221 在内存中的原始字节数据。对于一个非基本数据类型的对象，print 和 println 方法会先调用对象的 toString 方法，然后再输出 toString 方法所返回的字符串。

IO 包中提供了一个与 PrintStream 对应的 PrintWriter 类，PrintWriter 类的有下列几个构造方法：

PrintWriter(OutputStream)

PrintWriter(OutputStream, boolean)

PrintWriter(Writer)

PrintWriter(Writer, boolean)

PrintWriter 即使遇到换行符(\n)也不会自动清空缓冲区，只在设置了 autoflush 模式下使用了 println 方法后才自动清空缓冲区。PrintWriter 相对 PrintStream 最有利的一个地方就是 println 方法的行为，在 Windows 的文本换行是"\r\n"，而 Linux 下的文本换行是"\n"，如果希望程序能够生成平台相关的文本换行，而不是在各种平台下都用"\n"作为文本换行，那么就应该使用 PrintWriter 的 println 方法时，PrintWriter 的 println 方法能根据不同的操作系统而生成相应的换行符。

下面的范例通过 PrintWriter 类向屏幕上打印信息。

范例：SystemPrintDemo.java

```
01  import java.io.*;
02  public class SystemPrintDemo
03  {
04      public static void main(String args[])
05      {
06          PrintWriter out = null;
07          // 通过 System.out 为 PrintWriter 实例化
08          out = new PrintWriter(System.out);
09          // 向屏幕上输出

10          out. print ("Hello World!");
```

```
11              out.close();
12          }
13  }
```

输出结果：

Hello World!

程序说明：

程序第 8 行通过 System.out 实例化 PrintWriter，此时，PrintWriter 类的实例化对象 out，就具备了向屏幕输出信息的能力，所以在第 10 行调用 print()方法时，就会将内容打印到屏幕上。

下面的范例通过 PrintWriter 向文件中打印信息。

范例：FilePrint.java

```
01  import java.io.*;
02  public class FilePrint
03  {
04      public static void main(String args[])
05      {
06          PrintWriter out = null ;
07          File f = new File("c:\\temp.txt") ;
08          try
09          {
10              out = new PrintWriter(new FileWriter(f)) ;
11          }
12          catch (IOException e)
13          {
14              e.printStackTrace();
15          }
16          // 由 FileWriter 实例化，则向文件中输出
17          out. print ("Hello World!"+"\r\n");
18          out.close() ;

19      }
```

```
20  }
```

输出结果：

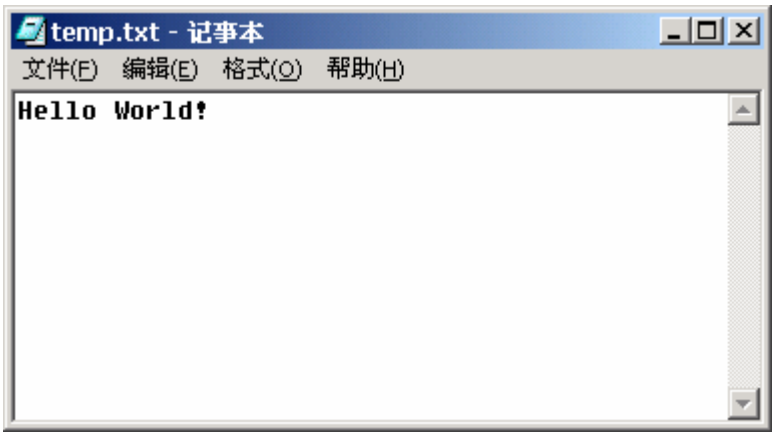


图 10-3　FilePrint 输出结果

程序说明：

程序第 10 行通过 FileWriter 类实例化 PrintWriter，此时，PrintWriter 类的实例化对象 out，就具备了向文件输出信息的能力，所以在第 17 行调用 print()方法时，就会将内容输出到文件之中。

10.3.7　DataInputStream 与 DataOutputStream

DataInputStream 与 DataOutputStream 提供了与平台无关的数据操作，通常会先通过 DataOutputStream 按照一定的格式输出，再通过 DataInputStream 按照一定格式读入。由于可以得到 java 的各种基本类型甚至字符串，这样对得到的数据便可以方便地进行处理，这在通过协议传输的信息的网络上是非常适用的。

如下面范例：

用户的定单用如下格式储存为 order.txt 文件：

价格	数量	描述
18.99	10	T 恤衫
9.22	10	杯子

实现机制如下：

（1）、写入端构造一个 DataOutputStream，并按照一定格式写入数据：

```
// 将数据写入某一种载体
DataOutputStream out = new DataOutputStream(new FileOutputStream("order.txt"));
// 价格
double[] prices = { 18.99, 9.22, 14.22, 5.22, 4.21 };
// 数目
int[] units = { 10, 10, 20, 39, 40 };
// 产品名称
String[] descs = { "T 恤衫", "杯子", "洋娃娃", "大头针", "钥匙链" };
// 向数据过滤流写入主要类型
for (int i = 0; i < prices.length; i++)
{
    // 写入价格，使用 tab 隔开数据
    out.writeDouble(prices[i]);
    out.writeChar('\t');
    // 写入数目
    out.writeInt(units[i]);
    out.writeChar('\t');
    // 写入产品名称，行尾加入换行符
    out.writeChars(descs[i]);
    out.writeChar('\n');
}
out.close();
```

（2）、计价程序读入并在标准输出中输出：

```
// 将数据读出
DataInputStream in = new DataInputStream(new FileInputStream("order.txt"));

double price;
int unit;
```

```
StringBuffer desc;
double total = 0.0;
try
{
    // 当文本被全部读出以后会抛出 EOFException 例外，中断循环
    while (true)
    {
        // 读出价格
        price = in.readDouble();
        // 跳过 tab
        in.readChar();
        // 读出数目
        unit = in.readInt();
        // 跳过 tab
        in.readChar();
        char chr;
        // 读出产品名称
        desc = new StringBuffer();
        while ((chr = in.readChar()) != '\n')
        {
            desc.append(chr);
        }
          System.out.println("定单信息：  " + "产品名称："+desc+"，\t 数量：
                                "+unit+"，\t 价格："+price);
        total = total + unit * price;
    }
}
catch (EOFException e)
{
    System.out.println("\n 总共需要：" + total+"元");
}
in.close();
```

范例：DataInputStreamDemo.java

```
01  import java.io.* ;
```

```
02  public class DataStreamDemo
03  {
04      public static void main(String[] args) throws IOException
05      {
06          // 将数据写入某一种载体
07  DataOutputStream out = new DataOutputStream(new FileOutputStream("order.txt"));
08          // 价格
09          double[] prices = { 18.99, 9.22, 14.22, 5.22, 4.21 };
10          // 数目
11          int[] units = { 10, 10, 20, 39, 40 };
12          // 产品名称
13          String[] descs = { "T 恤衫", "杯子", "洋娃娃", "大头针", "钥匙链" };
14
15          // 向数据过滤流写入主要类型
16          for (int i = 0; i < prices.length; i++)
17          {
18              // 写入价格，使用 tab 隔开数据
19              out.writeDouble(prices[i]);
20              out.writeChar('\t');
21              // 写入数目
22              out.writeInt(units[i]);
23              out.writeChar('\t');
24              // 写入产品名称，行尾加入换行符
25              out.writeChars(descs[i]);
26              out.writeChar('\n');
27          }
28          out.close();
29
30          // 将数据读出
31      DataInputStream in = new DataInputStream(new FileInputStream("order.txt"));
32
33          double price;
34          int unit;
35          StringBuffer desc;
36          double total = 0.0;
37
38          try
```

```
39              {
40                  // 当文本被全部读出以后会抛出 EOFException，中断循环
41                  while (true)
42                  {
43                      // 读出价格
44                      price = in.readDouble();
45                      // 跳过 tab
46                      in.readChar();
47                      // 读出数目
48                      unit = in.readInt();
49                      // 跳过 tab
50                      in.readChar();
51                      char chr;
52                      // 读出产品名称
53                      desc = new StringBuffer();
54
55                      while ((chr = in.readChar()) != '\n')
56                      {
57                          desc.append(chr);
58                      }
59                  System.out.println("定单信息：  " + "产品名称："+desc+"，\t 数量：
                    "
60                                      +unit+"，\t 价格："+price);
61                      total = total + unit * price;
62                  }
63              }
64              catch (EOFException e)
65              {
66                  System.out.println("\n 总共需要：" + total+"元");
67              }
68              in.close();
69          }
70  }
```

输出结果：

定单信息： 产品名称：T 恤衫， 数量：10， 价格：18.99
定单信息： 产品名称：杯子， 数量：10， 价格：9.22
定单信息： 产品名称：洋娃娃， 数量：20， 价格：14.22
定单信息： 产品名称：大头针， 数量：39， 价格：5.22
定单信息： 产品名称：钥匙链， 数量：40， 价格：4.21

总共需要：938.4799999999999 元

10.3.8 合并流

采用 SequenceInputStream 类，可以实现两个文件的合并操作。如图 10-4 所示：

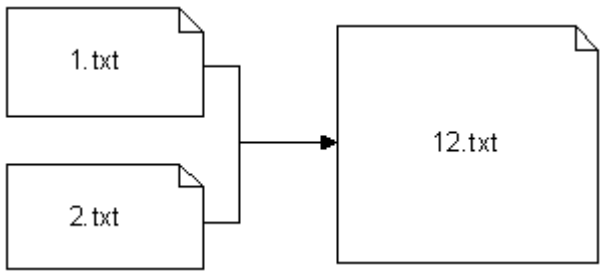


图 10-4 合并流操作

范例：SequenceDemo.java

```
01  import java.io.*;
02  public class SequenceDemo
03  {
04      public static void main(String[] args) throws IOException
05      {
06          // 声明两个文件读入流
07          FileInputStream in1 = null, in2 = null;
08          // 声明一个序列流
09          SequenceInputStream s = null;

10          FileOutputStream out = null;
```

```
11
12              try
13              {
14                  // 构造两个被读入的文件
15                  File inputFile1 = new File("c:\\1.txt");
16                  File inputFile2 = new File("c:\\2.txt");
17                  // 构造一个输出文件
18                  File outputFile = new File("c:\\12.txt");
19
20                  in1 = new FileInputStream(inputFile1);
21                  in2 = new FileInputStream(inputFile2);
22
23                  // 将两输入流合为一个输入流
24                  s = new SequenceInputStream(in1, in2);
25                  out = new FileOutputStream(outputFile);
26
27                  int c;
28                  while ((c = s.read()) != -1)
29                      out.write(c);
30
31                  in1.close();
32                  in2.close();
33                  s.close();
34                  out.close();
35                  System.out.println("ok...");
36              }
37              catch (IOException e)
38              {
39                  e.printStackTrace();
40              }
41              finally
42              {
43                  if (in1 != null)
44                      try {
45                          in1.close();
46                      } catch (IOException e) {

47                      }
```

```
48                  if (in2 != null)
49                      try {
50                          in2.close();
51                      } catch (IOException e) {
52                      }
53                  if (s != null)
54                      try {
55                          s.close();
56                      } catch (IOException e) {
57                      }
58                  if (out != null)
59                      try {
60                          out.close();
61                      } catch (IOException e) {
62                      }
63              }
64          }
65      }
```

输出结果：

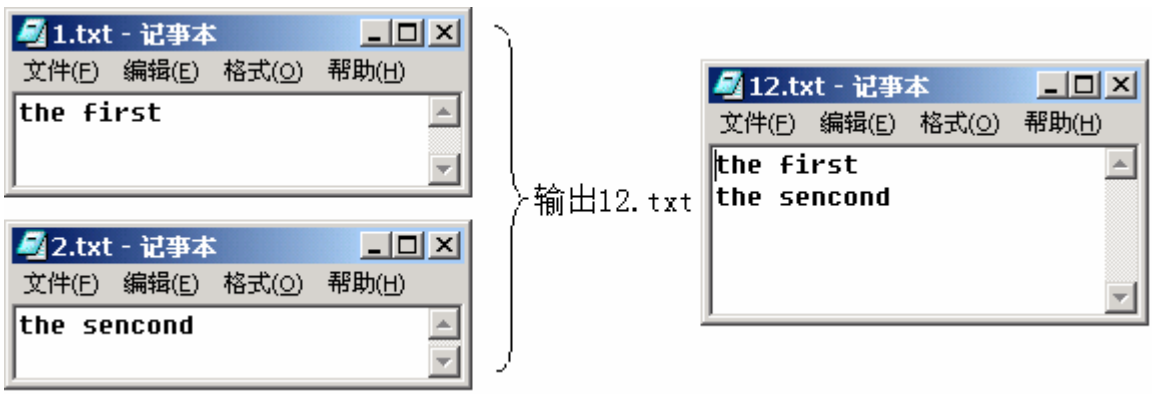


图 10-5　SequenceDemo 输出结果

10.3.9 字节流与字符流的转换

前面已经讲过，Java 支持字节流和字符流，但有时需要字节流和字符流之间的转换。

InputStreamReader 和 OutputStreamWriter，这两个类是字节流和字符流之间转换的类，InputStreamReader 可以将一个字节流中的字节解码成字符，OuputStreamWriter 将写入的字符编码成字节后写入一个字节流。

InputStreamReader 有两个主要的构造函数：

InputStreamReader(InputStream in)

// 用默认字符集创建一个 InputStreamReader 对象

InputStreamReader(InputStream in,String CharsetName)

// 接受已指定字符集名的字符串，并用该字符集创建对象

OutputStreamWriter 也有对应的两个主要的构造函数：

OutputStreamWriter(OutputStream in)

// 用默认字符集创建一个 OutputStreamWriter 对象

OutputStreamWriter(OutputStream in,String CharsetNarme)

// 接受已指定字符集名的字符串，并用该字符集创建 OutputStreamWriter 对象

为了达到最高的效率，避免频繁地进行字符与字节间的相互转换，最好不要直接使用这两个类来进行读写，应尽量使用 BufferedWriter 类包装 OutputStreamWriter 类，用 BufferedReader 类包装 InputStreamReader 类。例如：

BufferedWriter out=new BufferedWriter(newOutputStreamWriter(System.out));
BufferedReader in=new BufferedReader(new InputStreamReader(System.in));

接着从一个实际的应用中来了解 InputStreamReader 的作用，怎样用一种简单的方式一下子就读取到键盘上输入的一整行字符呢？只要用下面的两行程序代码就可以解决这个问题：

BufferedReader in=new BufferedReader(new InputStreamReader(System.in));

String strLine = in.readLine();

可见，构建 BufferedReader 对象时，必须传递一个 Reader 类型的对象作为参数，而键盘对应的 System.in 是一个 InputStream 类型的对象，所以这里需要用到一个 InputStreamReader 的转换类，将 System.in 转换成字符流之后，放入到字符流缓冲区之中，之后从缓冲区中每次读入一行数据。

【 格式 10-1 由键盘输入数据基本形式 】

```
import java.io.*;
public class class_name       // 类名
{
    public static void main(String args[]) throws IOException
    {
        BufferedReader buf;    // 声明 buf 为 BufferedReader 类的变量
        String str;            // 声明 str 为 String 类型的变量
        ... ...

        buf=new BufferedReader(new InputStreamReader(System.in));
        str=buf.readLine();    // 读入字符串至 buf
        ... ...
    }
}
```

范例：BufferDemo.java

```
01  public class BufferDemo
02  {
03      public static void main(String args[])
04      {
05          BufferedReader buf = null;
06          buf = new BufferedReader(new InputStreamReader(System.in));
07          String str = null;
08          while (true)

09          {
10              System.out.print("请输入数字：");
11              try
```

```
12                  {
13                      str = buf.readLine();
14                  } catch (IOException e)
15                  {
16                      e.printStackTrace();
17                  }
18                  int i = -1;
19                  try
20                  {
21                      i = Integer.parseInt(str);
22                      i++;
23                      System.out.println("输入的数字修改后为：" + i);
24                      break;
25                  }
26                  catch (Exception e)
27                  {
28                      System.out.println("输入的内容不正确，请重新输入！");
29                  }
30              }
31          }
32      }
```

输出结果：

请输入数字：22

输入的数字修改后为：23

程序说明：

1、 程序第 5 行、第 6 行，为 BufferedReader 对象实例化，因为现在需要从键盘输入数据，所以需要使用 System.in 进行实例化，但 System.in 是一 InputStream 类型，所以使用 InputStreamReader 类将字节流转换成字符流，之后将字符流放入到了 BufferedReader 之中。

2、 程序第 13 行通过 BufferedReader 类中的 readLine()方法，等待键盘的输入数据。

3、 程序第 21 行通过 Integer 类将输入的字符串转换成基本数据类型中的整型。

4、 第 22 行将输入的数字加一。第 23 行输出修改后的数据。

10.3.10　IO 包中的类层次关系图

10.3.10.1　字节输入流（InputStream）

InputStream 类的层次结构如图 10-6 所示：

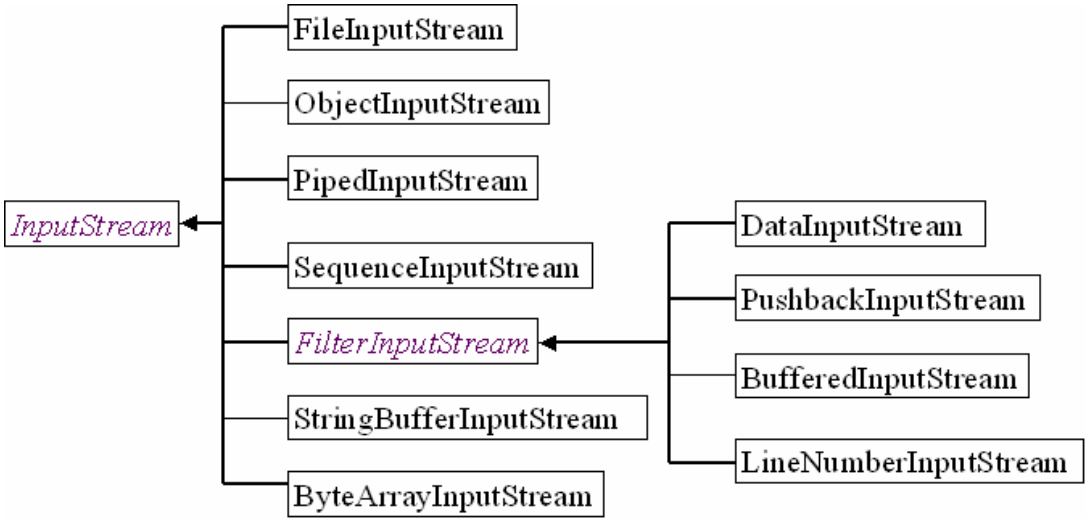


图 10-6　InputStream 类层次结构图

10.3.10.2　字节输出流

OutputStream 类的层次结构如图 10-7 所示：

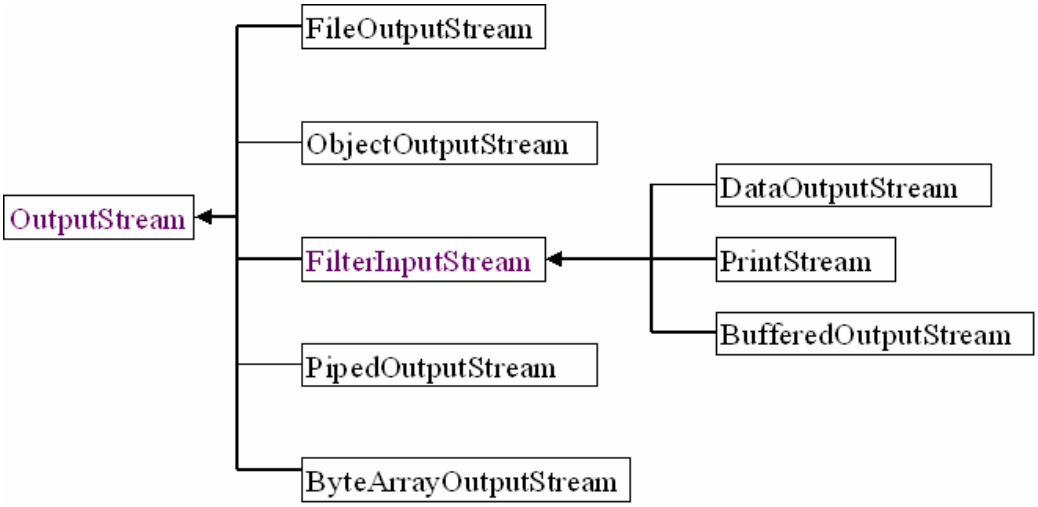


图 10-7　OutputStream 类层次结构图

10.3.10.3　字符输入流

Reader 类的层次结构如图 10-8 所示：

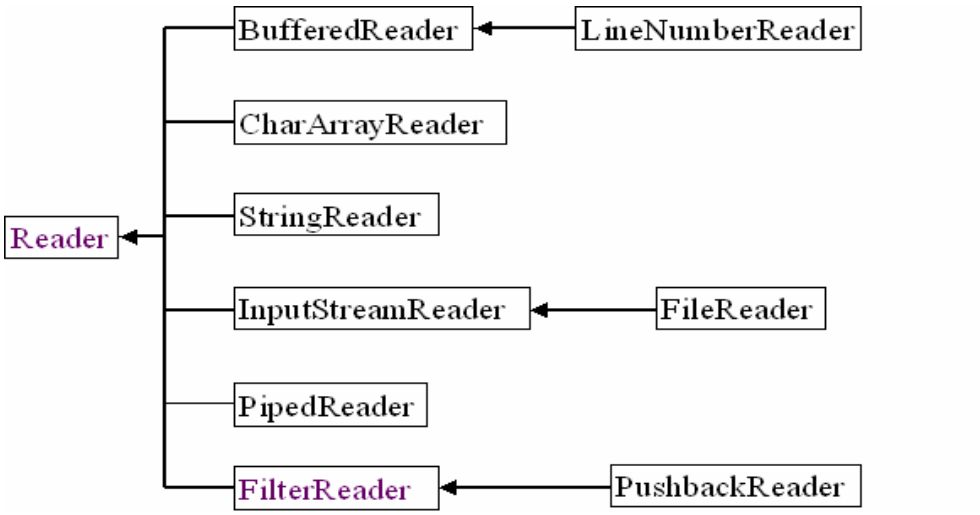


图 10-8　Reader 类层次结构图

10-3.10.4　字符输出流

Writer 类的层次结构如图 10-9 所示:

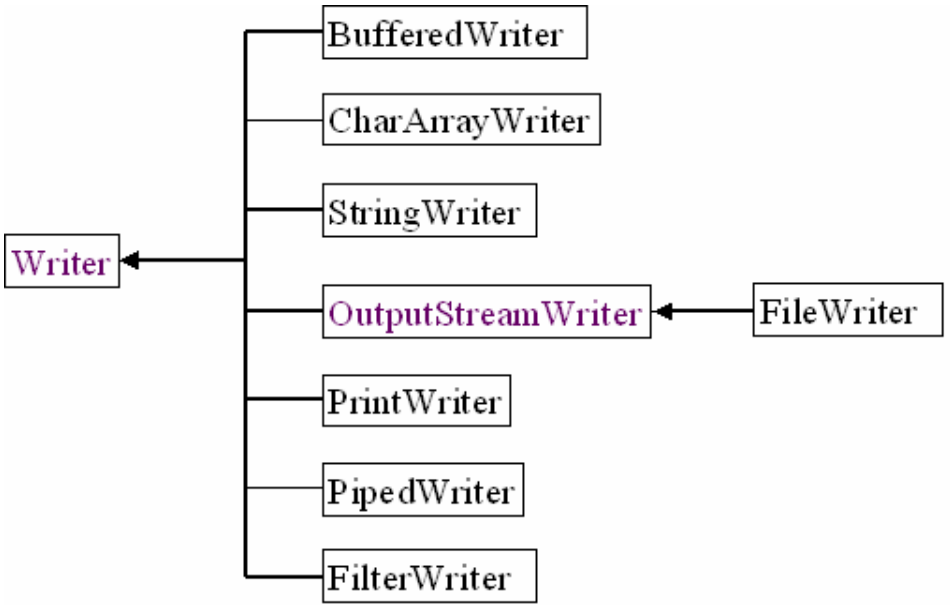


图 10-9　Writer 类层次结构图

10.4　字符编码

计算机里只有数字，在计算机软件里的一切都是用数字来表示，屏幕上显示的一个个字符也不例外。最初的计算机的使用是在美国，当时所用到的字符也就是现在键盘上的一些符号和少数几个特殊的符号，每一个字符都用一个数字来表示，一个字节所能表示的数字范围内足以容纳所有的这些字符，实际上表示这些字符的数字的字节最高位（bit）都为 0，也就是说这些数字都在 0 到 127 之间，如字符 a 对应数字 97，字符 b 对应数字 98 等，这种字符与数字对应的编码固定下来后，这套编码规则被称为 ASCII 码（美国标准信息交换码）。

随着计算机在其它国家的逐渐应用和普及，许多国家都把本地的字符集引入了计算机，大大扩展了计算机中字符的范围。一个字节所能表示的数字范围是不能容纳所

有的中文汉字的。中国大陆将每一个中文字符都用两个字节的数字来表示，原有的ASCII 码字符的编码保持不变，仍用一个字节表示，为了将一个中文字符与两个 ASCII 码字符相区别，中文字符的每个字节的最高位（bit）都为 1，中国大陆为每一个中文字符都指定了一个对应的数字，并作为标准的编码固定了下来，这套编码规则称为 GBK（国标码），后来又在 GBK 的基础上对更多的中文字符（包括繁体）进行了编码，新的编码系统就是 GB2312，而 GBK 则是 GB2312 的子集。使用中文的国家和地区很多，同样的一个字符，如“中国”的“中”字，在中国大陆的编码是十六进制的 D6D0，而在中国台湾的编码是十六进制的 A4A4，台湾地区对中文字符集的编码规则称为 BIG5（大五码）。

在一个国家的本地化系统中出现的一个字符，通过电子邮件传送到另外一个国家的本地化系统中，看到的就不是那个原始字符了，而是另外那个国家的一个字符或乱码，因为计算机里面并没有真正的字符，字符都是以数字的形式存在的，通过邮件传送一个字符，实际上传送的是这个字符对应的编码数字，同一个数字在不同的国家和地区代表的很可能是不同的符号，如十六进制的 D6D0 在中国大陆的本地化系统中显示为“中”这个符号，但在伊拉克的本地化系统就不知对应的是一个什么样的伊拉克字符了，反正人们看到的不是“中”这个符号。随着世界各国的交往越来越密切，全球一体化的趋势越来越明显，人们不可能完全忘记母语，都去使用英文在不同的国家和地区间交换越来越多的电子文档，特别是人们开发的应用软件都希望能走出国门、走向世界，可见，各个国家和地区都使用各自不同的本地化字符编码，已经给生活和工作带来了很多的不方便，严重制约了国家和地区间在计算机使用和技术方面的交流。

为了解决各个国家和地区使用自不同的本地化字符编码带来的不便，人们将全世界所有的符号进行了统一编码，称之为 Unicode 编码。所有字符不再区分国家和地区，都是人类共有的符号，如“中国”的“中”这个符号，在全世界的任何角落始终对应的都是一个十六进制的数字 4e2d，如果所有的计算机系统都使用这种编码方式，在中国大陆的本地化系统中显示的“中”这个符号，发送到伊拉克的本地化系统中，显示的仍然是“中”这个符号，至于那个伊拉克人能不能认识这个符号，就不是计算机所要解决的问题了。Unicode 编码的字符都占用两个字节的大小，也就是说全世界所有的字符个数不会超过 2 的 16 次方（65536），据推测一定是 Unicode 编码中没有包

诸如中国的藏文和满文这些少数民族的文字。

长期养成的保守习惯不可能一下子就改变过来，特别是不可能完全推翻那些已经存在的运行良好的系统。新开发的软件要做到瞻前顾后，既能够在存在的系统上运行，又便于以后的战略扩张和适应新的形式。Unicode 一统天下的局面暂时还难以形成，在相当长的一段时期内，人们看到的都是本地化字符编码与 Unicode 编码共存的景象。既然本地化字符编码与 Unicode 编码共存，那就少不了涉及两者之间的转化问题，在 Java 中的字符使用的都是 Unicode 编码，Java 技术在通过 Unicode 保证跨平台特性的前提下也支持了全扩展的本地平台字符集，而显示输出和键盘输入都是采用的本地编码。

通过下面的程序，来看一下字符乱码问题。在这里使用 String 类中的 get Bytes() 方法，为字符进行编码转换。

范例：EncodingDemo.java

```
01  public class EncodingDemo
02  {
03      public static void main(String args[]) throws Exception
04      {
05          // 在这里将字符串通过 getBytes()方法，编码成 GB2312
06          byte b[] = "大家一起来学 Java 语言".getBytes("GB2312") ;
07          OutputStream out = new FileOutputStream(new File("c:\\encoding.txt")) ;
08          out.write(b) ;
09          out.close() ;
10      }
11  }
```

输出结果：

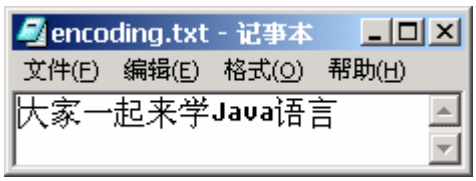


图 10-10 正常输出文字，无编码问题

上面的程序读者应该已经非常清楚了，但这里与之前稍有不同的是，在将字符串转换成 byte 数组的时候，用到了“GB2312”编码。

读到这里读者应该还是无法体会到字符编码问题，那么现在修改 EncodingDemo 程序，将字符编码转换成 ISO8859-1，但在执行此程序之前，先执行下面的程序：

范例：SetDemo.java

```
01  public class SetDemo
02  {
03      public static void main(String args[])
04      {
05          System.getProperties().put("file.encoding","GB2312") ;
06      }
07  }
```

执行此程序之后，再运行 EncodingDemo.java 程序，修改后的程序如下：

范例：EncodingDemo.java

```
01  public class EncodingDemo
02  {
03      public static void main(String args[]) throws Exception
04      {
05          // 在这里将字符串通过 getBytes()方法，编码成 ISO8859-1
06          byte b[] = "大家一起来学 Java 语言".getBytes("ISO8859-1") ;
07          OutputStream out = new FileOutputStream(new File("c:\\encoding.txt")) ;
08          out.write(b) ;
09          out.close() ;
10      }
11  }
```

输出结果：

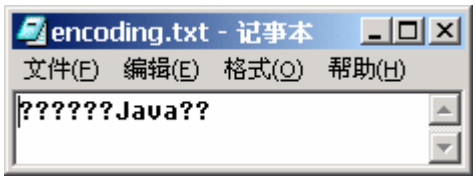


图 10-11 字符输出出现乱码

可以发现输出结果出现了乱码，这是为什么呢？这就是本节所要讨论的字符编码问题，之所以产生这样的问题，是因为在运行这段代码之前，先运行了 setDemo.java 程序，此程序主要是用来设置 JDK 环境的编码问题，所以乱码问题主要是由于 JDK 设置环境所引起的，为什么呢？读者可以运行下面的程序，观察其输出就可以发现问题。

范例：GetDemo.java

```
01  public class GetDemo
02  {
03      public static void main(String args[])
04      {
05          // 输出全部环境变量
06          System.getProperties().list(System.out);
07      }
08  }
```

输出结果：

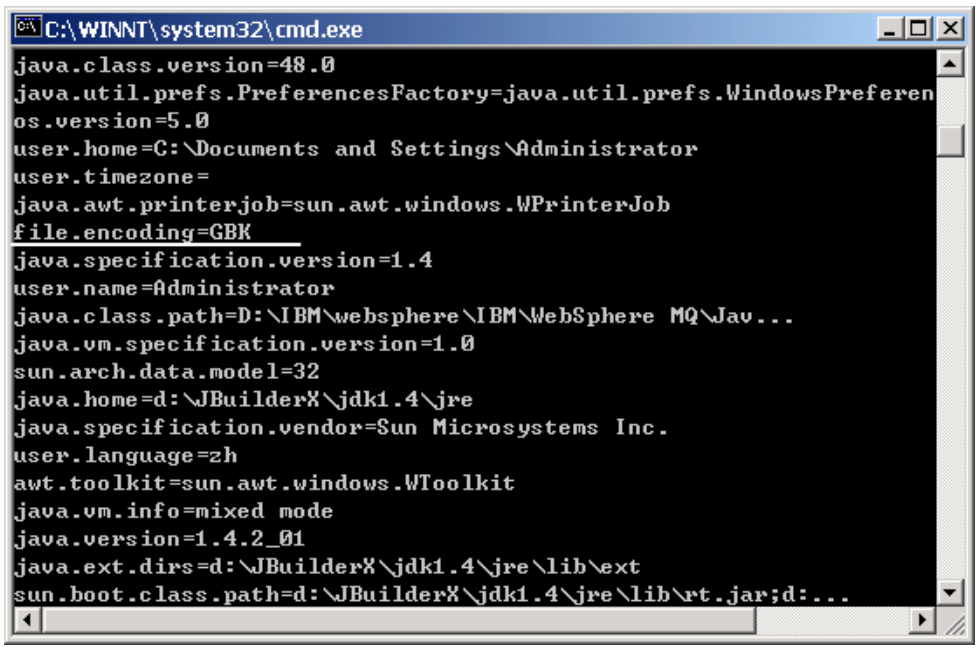


图 10-12 系统环境

读者可以发现，在环境变量之中有一个 file.encoding=GBK，清楚地表明了所使用的是 GBK 编码，而读者可以发现在修改过的 EncodingDemo.java 程序中的第 6 行：

byte b[] = "大家一起来学 Java 语言".getBytes("ISO8859-1") ;

将字符串编码换成了 ISO8859-1 编码，从前面介绍的基本知识已经知道 ISO8859-1 编码要大于 GBK 编码，所以才造成了字符的乱码问题。

10.5 对象序列化

所谓的对象序列化（在某些书中也叫串行化），是指将对象转换成二进制数据流的一种实现手段，通过将对象序列化，可以方便的实现对象的传输及保存。

在 Java 中提供了 ObjectlnputStream 与 ObjectOutputStream 这两个类用于序列化对象的操作。这两个类是用于存储和读取对象的输入输出流类，不难想象，只要把对象中的所有成员变量都存储起来，就等于保存了这个对象，之后从保存的对象之中再将对象读取进来就可以继续使用此对象。ObjectInputStream 与 ObjectOutputStream 类，可以帮开发者完成保存和读取对象成员变量取值的过程，但要求读写或存储的对象必须实现了 Serializable 接口，但 Serializable 接口中没有定义任何方法，仅仅被用作一种标记，以被编译器作特殊处理。如下范例所示：

范例：Person.java

```
01  import java.io.* ;
02  public class Person implements Serializable
03  {
04      private String name ;
05      private int age ;
06      public Person(String name,int age)
07      {
08          this.name = name ;
09          this.age = age ;
10      }
11      public String toString()
```

```
12        {
13              return " 姓名："+this.name+"，年龄："+this.age ;
14        }
15  };
```

第 2 行所中，类 Person 实现了 Serializable 接口，所以此类的对象可序列化。下面的范例使用 ObjectOutputStream 与 ObjectInputStream 将 Person 类的对象保存在文件之中。

范例：SerializableDemo.java

```
01  import java.io.*;
02  public class SerializableDemo
03  {
04        public static void main( String args[] ) throws Exception
05        {
06              File f = new File("SerializedPerson") ;
07              serialize(f);
08              deserialize(f);
09        }
10
11        // 以下方法为序列化对象方法
12        public static void serialize(File f) throws Exception
13        {
14              OutputStream outputFile = new FileOutputStream(f);
15              ObjectOutputStream cout = new ObjectOutputStream(outputFile);
16              cout.writeObject(new Person("张三",25));
17              cout.close();
18        }

19        // 以下方法为反序列化对象方法
20        public static void deserialize(File f) throws Exception
21        {
22              InputStream inputFile = new FileInputStream(f);
23              ObjectInputStream cin = new ObjectInputStream(inputFile);
24              Person p = (Person) cin.readObject();
25              System.out.println(p);
```

```
26          }
27    }
```

输出结果：

姓名：张三，年龄：25

程序说明：

1、 程序第 12 行~第 18 行声明一 serialize()方法，此方法用于将对象保存在文件之中。程序第 14 行、第 15 行为 ObjectOutputStream 对象实例化，此对象是通过 FileOutputStream 对象实例化，所以此类在保存 Person 对象时，向文件中输出。

2、 程序第 20 行~第 26 行，声明一 deserialize()方法，此方法用于从文件中读取已经保存的对象。程序第 22 行、第 23 行为 ObjectInputStream 对象实例化。第 24 行调用 ObjectInputStream 类中的 readObject()方法，从文件中读入内容，之后将读入的内容转型为 Person 类的实例。第 25 行直接打印 Person 对象实例，在打印对象时，默认调用 Person 类中的 toString()方法。

另外，要告诉读者的话，如果不希望类中的属性被序列化，可以在声明属性之前加上 transient 关键字。如下所示，下面的代码修改自前面所用到的 Person.java 程序，在声明属性时，前面多加了一个 transient 关键字。

```
04          private transient String name ;
05          private transient int age ;
```

再此运行 SerializableDemo.java 程序时，其输出结果为：

姓名：null，年龄：0

由输出结果可以发现，Person 类中的两个属性并没有被保存下来，输出时，是直接输出了其默认值。

• 本章摘要：

1、 Java 中要进行 IO 操作，需要导入 java.io 包。

2、 Java 中的 File 类是唯一操作磁盘文件的类。

3、 Java 中的数据操作主要分为两种：

（1）、 字节流（OutputStrea、InputStream）

（2）、 字符流（Writer、Reader）

这四个类都是抽象类，使用时，都必须依靠其子类实例化。

4、 Java 定义了两个特殊的流对象：System.in 和 System.out。System.in 对应键盘，是 InputStream 类型的，程序使用 System.in 可以读取从键盘上输入的数据;System.out 对应显示器，可以向显示器上输出内容。

5、 InputStreamReader 和 OutputStreamWriter，这两个类是字节流和字符流之间转换的类，InputStreamReader 可以将一个字节流中的字节解码成字符，OuputStreamWriter 将写入的字符编码成字节后写入一个字节流。

6、 一个类实现了 Serializable 接口之后，此类的对象可以被序列化，就表示可以保存在文件之中、或网络传输之中。如果不希望类中的某个属性被保存下来，可以用 transient 关键字声明属性。

第 11 章　Java Applet 程序

11.1　Applet 程序简介

Applet 程序是一个经过编译的 Java 程序，它既可以在 Appletviewer 下运行，也可以在支持 Java 的 Web 浏览器中运行。Applet 程序可以完成图形显示、声音演奏、接受用户输入、处理输入内容等工作。Applet 程序中必须有一个是 Applet 类的子类。

Applet 程序能跨平台、跨操作系统、跨网络运行，因此，它在 Internet 和 www 中得到广泛地使用。另外，由于 Applet 程序代码小，易于快速地下载和发送，并且，它具有不需要修改应用程序就可增加 Web 页新功能的特性，因此 Applet 程序倍受用户青睐。

Applet 程序不能单独运行，必须通过 HTML 调入后，方能执行以实现其功能。

下面先来看一个关于 Applet 的简单范例：

范例：HelloApplet.java

```
01  package test ;
02  import java.awt.Graphics;
03  import java.applet.Applet;
04  public class HelloApplet extends Applet
05  {
06      public void paint(Graphics g)
07      {
08          g.drawString("Hello World!!",5,30);     // 输出字符串
09      }
10  }
```

要运行 applet 程序只有 java 程序是不够的，还需要另外编写 html 文件，将 applet 程序嵌入到 html 文件之中。

范例：hello.htm

```
<HTML>
<HEAD>
<TITLE> Applet 程序 </TITLE>
<BODY>
    <APPLET CODE="test.HelloApplet" WIDTH="300" HEIGHT="100">
    </APPLET>
</BODY>
</HTML>
```

之后在命令行下执行 **appletviewer hello.htm** 文件

输出结果：

如图 11-1 所示：

图 11-1　applet 程序的运行

11.2　Applet 程序中使用的几个基本方法

Applet 类是浏览器类库中最为重要的类，同时也是所有 java 小应用程序的基本类。Apple 类中只有一种格式的构造方法 public Apple()，此种方法用来创建一个 Apple 类的实例。因此，在编写 Applet 程序时，首先必须引入 java.applet.Applet 包。

一个 Applet 应用程序从开始运行到结束时所经历的过程被称为 Applet 的生命周期。Applet 的生命周期涉及 init()、start()、stop()和 destroy()四种方法，这 4 种方法都

是 Applet 类的成员，可以继承这些方法，也可以重写这些方法，覆盖原来定义的这些方法。除此之外，为了在 Applet 程序中实现输出功能，每个 Applet 程序中还需要重载 paint()方法。

值得注意的是，在 Applet 类中没有提供 init()、start()、stop()、destroy()和 paint()方法的任何实现，且它们都是被浏览器或 Appletviewer 调用的，所以这几个方法要完成的功能应由编程人员自行编制。

1、 public void init()

init()方法是 Applet 运行的起点。当启动 Applet 程序时，系统首先调用此方法，以执行初始化任务。

2、 public void start()

start()方法是表明 Applet 程序开始执行的方法。当含有此 Applet 程序的 Web 页被再次访问时调用此方法。因此，如果每次访问 Web 页都需要执行一些操作的话，就需要在 Applet 程序中重载该方法。 在 Applet 程序中，系统总是先调用 init()方法，后调用 start()方法。

3、 public void stop()

stop()方法使 Applet 停止执行，当含有该 Applet 的 Web 页被其他页代替时也要调用该方法。

4、 public void destroy()

destroy()方法收回 Applet 程序的所有资源，即释放已分配给它的所有资源。在 Applet 程序中，系统总是先调用 stop()方法，后调用 destroy()方法。

5、 paint(Graphics g)

paint(Graphics g)方法可以使 Applet 程序在屏幕上显示某些信息，如文字、色彩、背景或图像等。参数 g 是 Graphics 类的一个对象实例，实际上可以把 g 理解为一个画笔。对象 g 中包含了许多绘制方法，如 drawstring()方法就是输出字符串。

repaint()方法的功能是，程序首先清除 paint()方法以前所画的内容，然后再调用

paint()方法。

范例：HelloAppletDemo.java

```
01  package test ;
02  import java.awt.Graphics;
03  import java.applet.Applet;
04  public class HelloAppletDemo extends Applet
05  {
06      String mystring="";
07      public void paint(Graphics g)
08      {
09          g.drawString(mystring,5,30);    // 输出字符串
10      }
11      public void init()
12      {
13          mystring=mystring+"正在初始化……";
14          repaint();
15      }
16      public void start()
17      {
18          mystring=mystring+"正在开始执行程序……";
19          repaint();
20      }
21      public void stop()
22      {
23          mystring=mystring+"正在停止执行程序……";
24          repaint();
25      }
26      public void destory()
27      {
28          mystring=mystring+"正在收回资源……";
29          repaint();
30      }
31  }
```

hello.htm

```
<HTML>
<HEAD>
<TITLE> Applet 程序 </TITLE>
<BODY>
    <APPLET CODE="test.HelloAppletDemo" WIDTH="300" HEIGHT="100">
    </APPLET>
</BODY>
</HTML>
```

输出结果：

运行结果如图 11-2 所示：

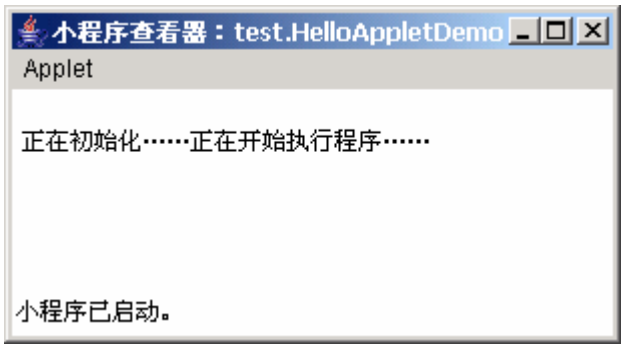


图 11-2　运行结果

由上面的输出可以得到 applet 程序运行的过程：在浏览器打开网页时，会调用 Applet 对象的 init()方法，接着是 start()方法，在离开此网页时，会调用 Applet 对象的 stop()方法，接着是 destroy()方法。

11.2 在 HTML 中嵌入 Applet 程序

11.2.1 HTML 代码的基本结构

在 WWW 中，每一显示单位称为一个网页，该网页是由 HTML 语言（HyperText Markup Language，超文本标记语言）编写而成的，HTML 是一种分层语言，各种标记均成对出现，用“< >”括起来，开始和结束标记的区别在于结束标记以“/”开头，标记字母忽略大小写。每个页面都必须包含相同的整体结构，它的结构如下：

```
<HTML>
<HEAD>  ……
<TITLE>  …… </TITLE>
</HEAD>
    <BODY>  ……</BODY>
</HTML>
```

其中：

（1）、 HTML 标记是最外层的标记，表示整个文档的开始和结束。

（2）、 HEAD 标记是第 2 层，用于把与文档有关的信息与文档主体分开，相当于文档的头部。

（3）、 TITLE 标记包含于 HEAD 内，向用户提示文档内包含的信息类型，并且为其页面提供一个描述性的标题。

（4）、 BODY 标记表示文档主体部分。

HTML 语言还包括许多其它标记，由于篇幅有限，这里只列举了几个，有兴趣的读者可查阅相关资料。

11.2.2 Applet 标记

<APPLET>标记的完整语法中，可以有若干个属性，其中必须的属性是 CODE，WIDTH 和 HEIGHT，其余均为可选项。<APPLET>标记的属性应该出现在<APPLET>

和</APPLET>之间。

下面是<APPLET>标记所具有的属性：

1、 CODEBASE = codebaseURL

可选属性，它指定 Java 字节代码的路径或 URL。如果未指定该属性，则将使用与.html 文档相同的目录。CODEBASE 的主要用途是告诉 Java 浏览器到哪里寻找在当前目录中没有显示的字节代码文件（.class）。

2、 ARCHIVE = archiveList

可选属性，它描述一个或多个包含有要“预加载”的类或其它资源的文档。archiveList 中的文档用“,”分隔。在 JAVA 2 中，具有相同 CODEBASE 的多个 Applet 程序共享一个 ClassLoader 实例。有些客户机代码使用它来实现 Applet 程序间通讯。从安全方面的因素考虑，Applet 程序的类加载器只能从所启动的那个 CODEBASE 中读取信息。这意味着 archiveList 中的文档位于和 codebaseURL 相同的目录中或其子目录中。

3、 CODE = AppletFile

必须属性，它提供包含 Applet 类的经编译后的 Applet 小程序。AppletFile 是一个已经编译后的 Java Applet 小程序，即扩展名为.class 的文件。在<APPLET>标记中，CODE 和 OBJECT 属性必须有一个存在。

4、 OBJECT = serialiaedApplet

可选属性，它给出包含 Applet 程序序列化表示的文件名。此时，Applet 程序中的 init()方法将不会被调用，但是其 start()方法将会被调用。由于该属性有一些严格规定的特性，所以不常使用。

5、 ALT=alternateText

可选属性，它指定在浏览器能识别<APPLET>标记但不能运行 Java Applet 程序时显示的正文内容。

6、 NAME=AppletInstanceName

可选属性，它用来为 Applet 程序指定一个符号名，该符号名在相同页的不同 Applet 程序之间通信时使用。

7、 WIDTH=pixels HEIGHT = pixels

两个必须属性，它们提供了 Applet 程序显示区域的初始宽度和高度（单位为像素），但不包括 Applet 程序中各种方法的任何显示窗口或对话框。

8、 ALIGN=alignment

可选属性，它指定 Applet 程序执行结果的对齐方式。该属性的可能值与 IMG 标记相同，即包括 LEFT，RIGHT，TOP，TEXTTOP，MIDDLE，ABSMIDDLE，BASELINE，BOTTOM 和 ABSBOTTOM。

9、 VSPACE= pixels HSPACE= pixels

两个可选属性，它们指定 Applet 程序执行结果的显示区上下（VSPACE）和两边（HSPACE）的像素数，对它们的处理方式与 IMG 标记的 VSPACE 和 HSPACE 属性相同。

10、<PARAM NAME=AppletAttribute 1 VALUE =value>

<PARAM NAME= AppletAttribute 2 VALUE = value>……

可选属性，它指定给 Applet 程序传递参数的名字和数据。Applet 程序中使用 getParameter()方法可以得到这些参数值。

HTML 中使用<APPLET>标记的属性的大部分是比较容易理解的，惟有传递参数的属性复杂一些，因为它实现了从 Web 页面到 Applet 程序的通信。

11.2.3 在 HTML 中传递 Applet 程序使用的参数

为了使 Applet 程序更具灵活性，需要在小程序中设置一些未知参数，以接受来自 Web 页面的信息。即在 HTML 中需要传递参数给 Applet 程序。在 HTML 中传递 Applet 程序使用的参数，可以使用<APPLET>标记的属性<PARAM>来实现，Applet 程序中可以使用 String getParameter(String name)方法得到 HTML 中<APPLET>标记的 PARAM 属性传递的参数值，该方法可以在任何地方被调用，但是建议在 init()方法中使用。需要注意的是，参数名是区分大小写的。

请看下面的范例：

范例：AcceptParam.java

```
01  package test ;
02  import java.awt.Graphics;
03  import java.applet.Applet;
04  public class AcceptParam extends Applet
05  {
06      String tempString,score;
07      public void init()
08      {
09          // 得到 web 页中的 str 参数的值
10          tempString = getParameter("str");
11          if(tempString.equals("及格"))      // 如果字符串等于"及格"
12              score = "60-70";
13          else if(tempString.equals("中"))        // 如果字符串等于"中"
14              score = "70-80";
15          else if(tempString.equals("良"))        // 如果字符串等于"良"
16              score = "80-90";
17          else if(tempString.equals("优"))        // 如果字符串等于"优"
18              score = "90-100";
19          else
20              score = "0-60" ;
21      }
22      public void paint(Graphics g)
23      {
```

```
24              g.drawString(score,10,25);          // 输出分数段
25          }
26      }
```

下面编写 HTML 文件，在 HTML 源文件中使用 PARAM 属性来指定 APPLET 程序欲使用的参数值。

范例：appletdemo.htm

```
<HTML>
<HEAD>在 HTML 中传递 Applet 使用的字符串参数</HEAD>
<HR>
<BODY>
    <APPLET CODE="test.AcceptParam" WIDTH=150 HEIGHT=30>
        <PARAM NAME="str" VALUE="差">
    </APPLET>
<BR>
    <APPLET CODE="test.AcceptParam" WIDTH=150 HEIGHT=30>
        <PARAM NAME="str" VALUE="及格">
    </APPLET>
<BR>
    <APPLET CODE="test.AcceptParam" WIDTH=150 HEIGHT=30>
        <PARAM NAME="str" VALUE="中">
    </APPLET>
<BR>
    <APPLET CODE="test.AcceptParam" WIDTH=150 HEIGHT=30>
        <PARAM NAME="str" VALUE="良">
    </APPLET>
<BR>
    <APPLET CODE="test.AcceptParam" WIDTH=150 HEIGHT=30>
        <PARAM NAME="str" VALUE="优">
    </APPLET>
</BODY>
</HTML>
```

输出结果：

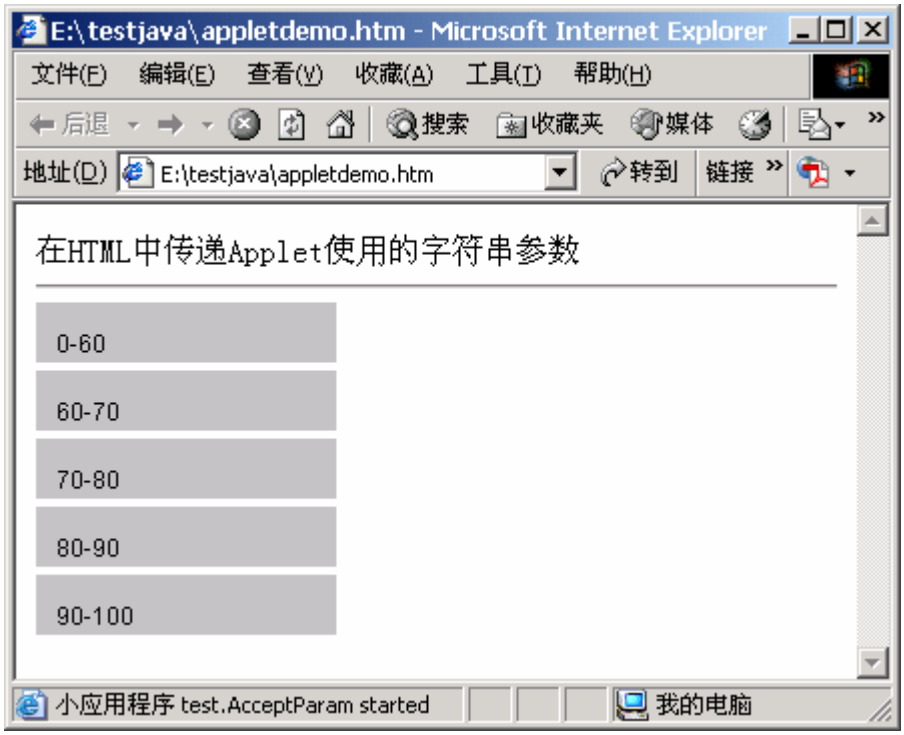


图 11-3

随着 java 技术的不断发展，Applet 程序的使用已经逐渐减少，在这里，读者只需要了解 Applet 程序的基本组成，及如何使用 Applet 程序就可以了。

• 本章摘要：

1、 要编写一 Applet 程序时，一个类必须继承自 Applet 类，之后要复写里面的 paint(Graphics g)方法。

2、 Applet 程序不能单独运行，必须嵌套在 HTML 中才可以使用。

第 12 章　Java 常用类库

12.1　API 概念

API (Application Programming Interface)就是应用程序编程接口。

假设现在要编写一个机器人程序，去控制一个机器人踢足球，程序需要向机器人发出向前跑、向后转、射门、拦截等命令，没有编过程序的人很难想象如何编写这样的程序。但对于有经验的人来说，就知道机器人厂商一定会提供一些控制这些机器人的 Java 类，该类中就有操纵机器人的各种动作的方法，只需要为每个机器人安排一个该类的实例对象，再调用这个对象的各种方法，机器人就会去执行各种动作。这个 Java 类就是机器人厂家提供的应用程序编程的接口，厂家就可以对这些 Java 类美其名曰：Xxx Robot API（也就是 Xxx 厂家的机器人 API）。好的机器人厂家不仅会提供 Java 程序用的 Robot API，也会提供 Windows 编程语言（如 VC++）用的 Robot API，以满足各类编程人员的需要。

在学习 Windows 编程时，经常听说 Windows API，其实也就是 Windows 操作系统提供的编写 Windows 程序的一些函数，如 CreateWindow 就是一个 API 函数，在应用程序中调用这个函数，操作系统就会按照该函数提供的参数信息产生一个相应的窗口。在 Java 中，经常提到的 API，就是 JDK 中提供的各种功能的 Java 类。

12.2　String 类和 StringBuffer 类

一个字符串就是一连串的字符，字符串的处理在许多程序中都用得到。Java 定义了 String 和 StringBuffer 两个类来封装对字符串的各种操作。它们都被放到了 java.lang 包中，不需要用 import java.lang 这个语句导入该包就可以直接使用它们。

String 类用于比较两个字符串、查找和抽取串中的字符或子串、字符串与其它类型之间的相互转换等。String 类对象的内容一旦被初始化就不能再改变。

StringBuffer 类用于内容可以改变的字符串，可以将其它各种类型的数据增加、插

入到字符串中，也可以转置字符串中原来的内容。一旦通过 StringBuffer 生成了最终想要的字符串，就应该使用 StringBuffer.toString 方法将其转换成 String 类，随后，就可以使用 String 类的各种方法操纵这个字符串了。

Java 为字符串提供了特别的连接操作符（+），可以把其它各种类型的数据转换成字符串，并前后连接成新的字符串。连接操作符（+）的功能是通过 StringBuffer 类和它的 append 方法实现的。例如：

```
String x = "a" + 4 + "c";
```

编译时等效于

```
String x=new StringBuffer().append("a").append(4).append("c").toString();
```

在实际开发中，如果需要频繁改变字符串的内容就需要考虑用 StringBuffer 类实现，因为其内容可以改变，所以执行性能会比 String 类更高。

12.3 基本数据类型的包装类

Java 对数据既提供基本数据的简单类型，也提供了相应的包装类（也叫包装类）。使用基本数据类型，可以改善系统的性能，也能够满足大多数应用需求。但基本数据类型不具有对象的特性，不能满足某些特殊的需求。从 JDK 中可以知道，Java 中的很多类的很多方法的参数类型都是 Object，即这些方法接收的参数都是对象，同时，又需要用这些方法来处理基本数据类型的数据，这时就要用到包装类。比如，用 Integer 类来包装整数。关于这种应用，在本章后面讲解集合类时会讲到。

读者从前面的章节中应该已经了解到 Java 中的基本数据类型共有八种，那么与之相对应的基本数据类型包装类也同样是有八种，表 12-1 列出了其对应关系：

表 12-1　基本数据类型的包装类与基本数据类型的对应关系

基本数据类型	基本数据类型包装类
int	Integer
char	Character
float	Float
double	Double
byte	Byte
long	Long
short	Short
boolean	Boolean

下面举一个具体的例子，告诉读者如何去使用这些包装类：

范例：IntegerDemo.java

```
01  class IntegerDemo
02  {
03      public static void main(String[] args)
04      {
05          String a = "123" ;
06          int i = Integer.parseInt(a) ;
07          i++;
08          System.out.println(i);
09      }
10  }
```

输出结果：

```
124
```

程序说明：

本程序使用 Integer 类中的 parseInt()方法，将一字符串转换成基本数据类型。

12.4 System 类与 Runtime 类

12.4.1 System 类

Java 不支持全局函数和变量，Java 设计者将一些系统相关的重要函数和变量收集到了一个统一的类中，这就是 System 类。System 类中的所有成员都是静态的，而要引用这些变量和方法时，直接使用 System 类名作前缀，在前面已经使用到了标准输入和输出的 in 和 out 变量。下面再介绍 System 类中几个方法，其它的方法读者还是参看 JDK 文档资料。

exit(int status)方法，提前终止虚拟机的运行。对于发生了异常情况而想终止虚拟机的运行，传递一个非零值作为参数。若在用户正常操作下，终止虚拟机的运行，传递零值作为参数。

CurrentTimeMillis 方法返回自 1970 年 1 月 1 日 0 点 0 分 0 秒起至今的以毫秒为单位的时间，这是一个 long 类型的大数值。在计算机内部，只有数值，没有真正的日期类型及其它各种类型，也就是说，平常用到的日期本质上就是一个数值，但通过这个数值，能够推算出其对应的具体日期时间。

getProperties 方法与 Java 的环境属性

getProperties 方法是获得当前虚拟机的环境属性。如果读者明白 Windows 的环境属性，如本书在第 1 章中讲到的 classpath 就是其中的两个环境变量，每一个属性都是变量与值以成对的形式出现的。

同样的道理，Java 作为一个虚拟的操作系统，它也有自己的环境属性，Properties 是 Hashtable 的子类，正好可以用于存储环境属性中的多个“变量/值”对格式的数据，getProperties 方法返回值是，包含了当前虚拟机的所有环境属性的 Properties 类型的对象。下面的例子打印出当前虚拟机的所有环境属性的变量和值。

范例：SystemInfo.java

```
01  import java.util.*;
02  public class SystemInfo
```

```
03  {
04      public static void main(String[] args)
05      {
06          Properties sp=System.getProperties();
07          Enumeration e=sp.propertyNames();
08          while(e.hasMoreElements())
09          {
10              String key=(String)e.nextElement();
11              System.out.println(key+" = "+sp.getProperty(key));
12          }
13      }
14  }
```

输出结果：

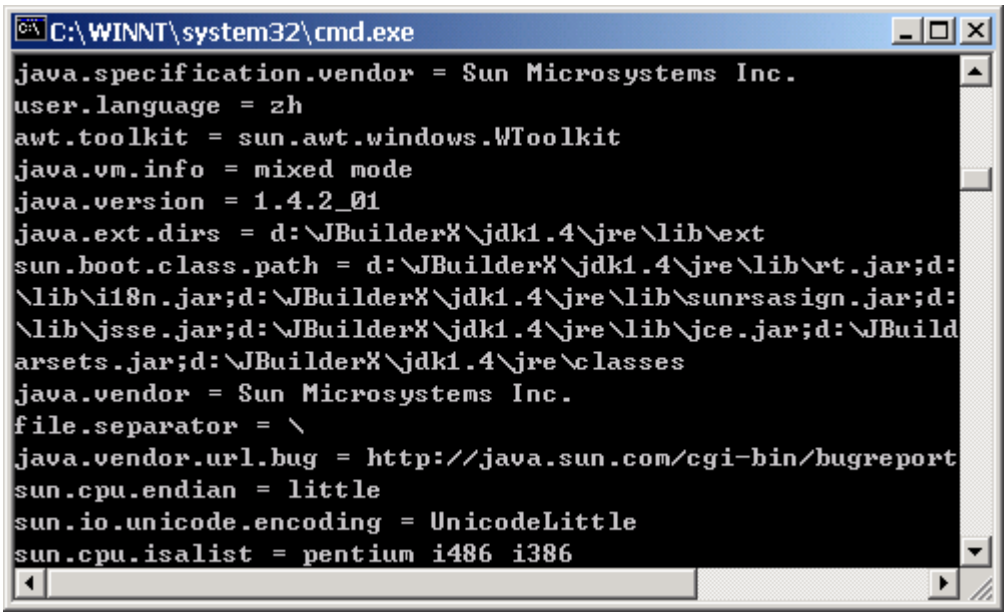


图 12-1　SystemInfo.java 程序的输出结果

在 Windows 中，增加一个新的环境属性是很容易的，但如何为 Java 虚拟机增加一个新的环境属性呢？在命令行窗口中直接运行 Java 命令，在显示的用法帮助中，会看到 Java 命令有一个-D<name>=<value>格式的选项可以设置新的系统环境属性。按下

面的格式运行：

java -DAuthor=lxh SystemInfo

运行后如图 12-2 所示：

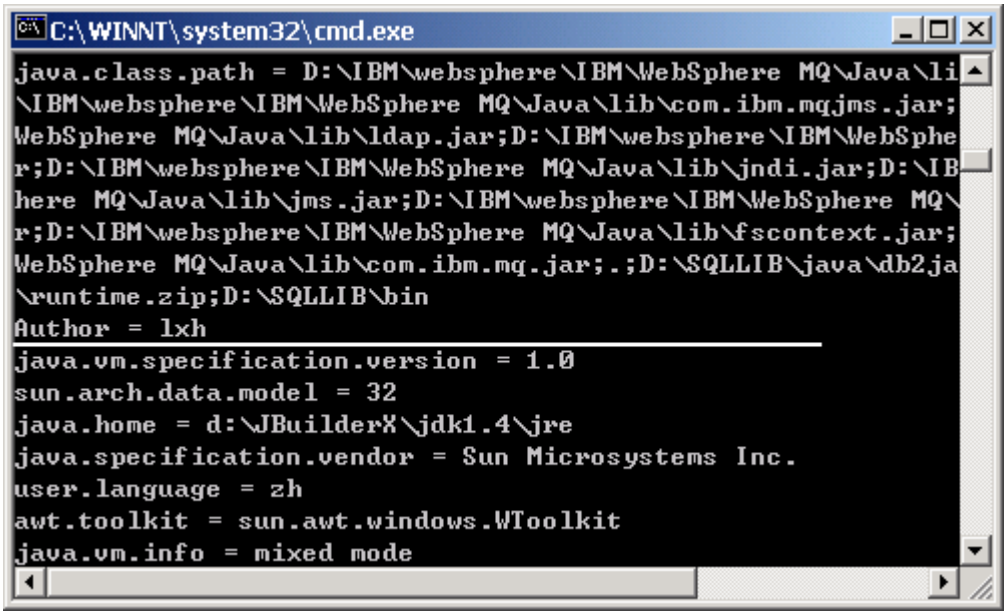


图 12-2

可以看到输出的结果中多了一行“Author = lxh”，即 Java 虚拟机中多了一个新的环境属性 Author。

注意：

-D 与 Author 之间没有空格。

讲解了 getProperties 方法，读者应该能够明白 setProperties 方法了。

12.4.2 Runtime 类

Runtime 类封装了 Java 命令本身的运行进程，其中的许多方法与 System 中的方法重复。不能直接创建 Runtime 实例，但可以通过静态方法 Runtime.getRuntime 获得正在运行的 Runtime 对象的引用。

Java 命令运行后，本身是多任务操作系统上的一个进程，在这个进程中启动一个新的进程，即执行其它程序时使用 exec 方法。exec 方法返回一个代表子进程的 Process 类对象，通过这个对象，Java 进程可以与子进程交互。

范例：RuntimeDemo.java

```
01  public class RuntimeDemo
02  {
03      public static void main(String[] args)
04      {
05          Runtime run = Runtime.getRuntime() ;
06          try
07          {
08              run.exec("notepad.exe") ;
09          }
10          catch (Exception e)
11          {
12              e.printStackTrace();
13          }
14      }
15  }
```

运行程序之后，可以发现程序已经为读者打开了记事本程序。所以通过 Runntime 类可以为开发者执行操作系统的可执行程序。

12.5 Date 与 Calendar、DateFormat 类

Date 类用于表示日期和时间，最简单的构造函数是 Date()，它以当前的日期和时间初始化一个 Date 对象。由于开始设计 Date 时没有考虑到国际化，所以后来又设计了两个新的类来解决 Date 类中的问题，一个是 Calendar 类，一个是 DateFormat 类。

Calendar 类是一个抽象基类，主要完成日期字段之间相互操作的功能，如 Calendar.add 方法可以实现在某一日期的基础上增加若干天（或年、月、小时、分、秒等日期字段）后的新日期，Calendar.get 方法可以取出日期对象中的年、月、日、小时、分、秒等日期字段的值，Calendar.set 方法修改日期对象中的年、月、日、小时、分、秒等日期字段的值。Calendar.getInstance 方法可以返回一个 Calendar 类型（更确切地说是它的某个子类）的对象实例，GregorianCalendar 类是 JDK 目前提供的一个惟一的 Calendar 子类，Calendar.getInstance 方法返回的就是预设了当前时间的 GregorianCalendar 类对象。

下面的例子计算出距当前日期时间 230 天后的日期时间，并用“xxxx 年 xx 月 xx 日 xx 小时：xx 分：xx 秒”的格式输出。

范例：CalendarDemo.java

```
01  import java.util.*;
02  public class CalendarDemo
03  {
04      public static void main(String[] args)
05      {
06          Calendar c1=Calendar.getInstance();
07          // 下面打印当前时间
08          System.out.println(c1.get(c1.YEAR)+"年"+(c1.get(c1.MONTH)+1)+
09              "月"+c1.get(c1.DAY_OF_MONTH)+"日"+c1.get(c1.HOUR)+
10                  ":"+c1.get(c1.MINUTE)+":"+c1.get(c1.SECOND));
11          // 增加天数为 230
12          c1.add(c1.DAY_OF_YEAR,230);
13
14          // 下面打印的是 230 天后的时间
15          System.out.println(c1.get(c1.YEAR)+"年"+(c1.get(c1.MONTH)+1)+
```

```
16                          "月"+c1.get(c1.DAY_OF_MONTH)+"日"+c1.get(c1.HOUR)+
17                              ":"+c1.get(c1.MINUTE)+":"+c1.get(c1.SECOND));
18          }
19  }
```

输出结果：

2005 年 7 月 29 日 5:56:57

2006 年 3 月 16 日 5:56:57

虽然 Calendar 类几乎完全替代了 Date 类，但在某些情况下，开发者仍有可能要用到 Date 类，譬如，程序中用的另外一个类的方法要求一个 Date 类型的参数。有时，要将用 Date 对象表示的日期以指定的格式输出或是将用特定格式显示的日期字符串转换成一个 Date 对象。而 Java.text.DateFormat 就是实现这种功能的抽象基类，java.text.SimpleDateFormat 类是 JDK 目前提供的一个的 DateFormat 子类，它是一个具体类，具有把 Date 对象格式化为本地字符串，或者通过语义分析把日期或时间字符串转换为 Date 对象的功能。

下面的范例将“2005-8-11 18:30:38”格式的日期字符串转成“2005 年 08 月 11 日 06 点 30 分 38 秒”的形式。

范例：DateFormatDemo.java

```
01  import java.text.* ;
02  import java.util.Date;
03  public class DateFormatDemo
04  {
05      public static void main(String[] args)
06      {
07      SimpleDateFormat sp1 = new SimpleDateFormat("yyyy-MM-dd hh:mm:ss") ;
08      SimpleDateFormat sp2 =
09              new SimpleDateFormat("yyyy 年 MM 月 dd 日  hh 时 mm 分 ss 秒") ;
10      try
11          {
12              Date d = sp1.parse("2005-8-11 18:30:38") ;
13              System.out.println(sp2.format(d)) ;
```

```
14                          }
15          catch (ParseException e)
16              {
17                  e.printStackTrace();
18              }
19          }
20  }
```

输出结果：

2005 年 08 月 11 日 06 点 30 分 38 秒

SimpleDateFormat 类就相当于一个模板，其中 yyyy 对应的是年，MM 对应的是月，dd 对应的是日，更详细的细节查阅 JDK 文档，关于这些参数，JDK 中写得非常清楚。

在上面程序中，定义了一个 SimpleDateFormat 类的对象 sp1 来接收和转换源格式字符串“2005-8-11 18:30:38”，随后又定义了该类的另一个对象 sp2 来接收 sdf1 转换成的 Date 类的对象，并按 sp2 所定义的格式转换成字符串。

在这个过程中，已经实现了利用 SimpleDateFormat 类来把一个字符串转换成 Date 类对象及把一个 Date 对象按用户指定的格式输出的两个功能。

12.6 Math 与 Random 类

Math 类包含了所有用于几何和三角的浮点运算函数，这些函数都是静态的，每个方法的使用都非常简单，读者一看 JDK 文档就能明白。

Random 类是一个随机数产生器，随机数是按照某种算法产生的，一旦用一个初值创建 Random 对象，就可以得到一系列的随机数，但如果用相同的初值创建 Random 对象，得到的随机数序列是相同的，也就是说，在程序中看到的“随机数”是固定的那些数，起不到“随机”的作用，针对这个问题，Java 设计者们在 Random 类的 Random() 构造方法中使用当前的时间来初始化 Random 对象，因为没有任何时刻的时间是相同的，所以就可以减少随机数序列相同的可能性。

下面的程序就是利用 Random 类产生 5 个 0~100 之间的随机整数。

范例：RandomDemo.java

```
01  import java.util.Random;
02  public class RandomDemo
03  {
04      public static void main(String[] args)
05      {
06          Random r = new Random() ;
07          for(int i=0;i<5;i++)
08          {
09              System.out.print(r.nextInt(100)+"\t") ;
10          }
11      }
12  }
```

输出结果：

```
82  83  62  39  96
```

12.7 类集框架

Java 的类集（Collection）框架使程序处理对象组的方法标准化。在 Java 2 出现之前，Java 提供了一些专门的类如 Dictionary，Vector，Stack 和 Properties 去存储和操作对象组。尽管这些类非常有用，但它们却缺少一个集中、统一的主题。因此使用 Vector 的方法就会与使用 Properties 的方法不同。以前的专门的方法也没有被设计成易于扩展和能适应新的环境的形式。而类集解决了这些（以及其它的一些）问题。类集框架被设计成拥有以下几个特性：首先，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。一般很少需要人工去对这些“数据引擎”编写代码（如果有的话）。第二点，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。第三点，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。为这些接口提供了几个标准的实现工具（例如 LinkedList，HashSet 和 TreeSet），通常就是这样使用的。如果读者愿意的话，也可以实现自己的类集。

为了方便起见，创建用于各种特殊目的的实现工具。一部分工具可以使自己的类

集实现更加容易。最后，增加了允许将标准数组融合到类集框架中的机制。

算法（Algorithms）是类集机制的另一个重要部分。算法操作类集，它在 Collections 类中被定义为静态方法。因此它们可以被所有的类集所利用。每一个类集的类不必实现它自己的方案，算法提供了一个处理类集的标准方法。

由类集框架创建的另一项是 Iterator 接口。一个迭代程序（iterator）提供了一个多用途的，标准化的方法，用于每次访问类集的一个元素。因此迭代程序提供了一种枚举类集内容（enumerating the contents of a collection）的方法。因为每一个类集都实现 Iterator，所以通过由 Iterator 定义的方法，任一类集类的元素都能被访问到。因此，稍作修改，循环通过集合的程序代码也可以被用来循环通过列表。

除了类集之外，框架定义了几个映射接口和类。映射（Maps）存储键值到值的对应。尽管映射在对象的正确使用上不是“类集”，但它们完全用类集集成。在类集框架的语言中，可以获得映射的类集“视图”（collection-view）。这个“视图”包含了从存储在类集中的映射得到的元素。因此，如果选择了一个映射，就可以将其当做一个类集来处理。

对于由 java.util 定义的原始类，类集机制被更新以便它们也能够集成到新的系统里。所以理解下面的说法是很重要的：尽管类集的增加改变了许多原始工具类的结构，但它却不会导致原始的工具类被抛弃。类集仅仅是提供了处理事情的一个更好的方法。

12.7.1 类集接口

类集框架定义了几个接口。本节对每一个接口都进行了概述。首先讨论类集接口是因为它们决定了collection类的基本特性。不同的是，具体类仅仅是提供了标准接口的不同实现。支持类集的接口总结在表12-1中：

表12-1　类集接口

接口	描述
Collection	能操作对象组，它位于类集层次结构的顶层
List	扩展Collection去处理序列（对象的列表）
Set	扩展Collection去处理集合，集合必须包含唯一元素
SortedSet	扩展Set去处理排序集合

除了类集接口之外，类集也使用Comparator，Iterator和ListIterator接口。关于这些接口将在本章后面做更深入的讲解。简单地说，Comparator接口定义了两个对象比较方法。

Iterator和ListIterator接口类集中的对象。

为了在它们的使用中提供最大的灵活性，类集接口允许对一些方法进行选择。可选择的方法使得使用者可以更改类集的内容。支持这些方法的类集被称为可修改的（modifiable）。不允许修改其内容的类集被称为不可修改的（unmodifiable）。而所有内置的类集都是可修改的。如果对一个不可修改的类集使用这些方法，将引发一个UnsupportedOperationException异常。

12.7.1.1　类集接口

Collection 接口是构造类集框架的基础。它声明所有类集都将拥有的核心方法。这些方法被总结在表 12-2 中。因为所有类集实现 Collection，所以熟悉它的方法对于清楚地理解框架是必要的。其中几种方法可能会引发一个 UnsupportedOperationException 异常。正如上面的解释，这些将发生在当类集不能被修改的时候。当一个对象与另一个对象不兼容，例如：当企图增加一个不兼容的对象到一个类集中时，将产生一个 ClassCastException 异常。

方法	描述
boolean add(Object obj)	将obj加入到调用类集中。如果obj被加入到类集中了，则返回true；如果obj已经是类集中的一个成员或类集不能被复制时，则返回false
boolean addAll(Collection c)	将c中的所有元素都加入到调用类集中，如果操作成功（元素被加入了），则返回true；否则返回false
void clear( )	从调用类集中删除所有元素
boolean contains(Object obj)	如果obj是调用类集的一个元素，则返回true，否则，返回false
boolean containsAll(Collection c)	如果调用类集包含了c中的所有元素，则返回true；否则，返回false
boolean equals(Object obj)	如果调用类集与obj相等，则返回true；否则返回false
int hashCode( )	返回调用类集的散列码
boolean isEmpty( )	如果调用类集是空的，则返回true；否则返回false
Iterator iterator( )	返回调用类集的迭代程序
Boolean remove(Object obj)	从调用类集中删除obj的一个实例。如果这个元素被删除了，则返回true；否则返回false
Boolean removeAll(Collection c)	从调用类集中删除c的所有元素。如果类集被改变了（也就是说元素被删除了），则返回true；否则返回false
Boolean retainAll(Collection c)	删除调用类集中除了包含在c中的元素之外的全部元素。如果类集被改变了（也就是说元素被删除了），则返回true，否则返回false
int size( )	返回调用类集中元素的个数
Object[ ] toArray( )	返回一个数组，该数组包含了所有存储在调用类集中的元素。数组元素是类集元素的拷贝
Object[ ] toArray(Object array[ ])	返回一个数组，该数组仅仅包含了那些类型与数组元素类型匹配的类集元素。数组元素是类集元素的拷贝。如果array的大小与匹配元素的个数相等，它们被返回到array。如果array的大小比匹配元素的个数小，将分配并返回一个所需大小的新数组，如果array的大小比匹配元素的个数大，在数组中，在类集元素之后的单元被置为null。如果任一类集元素的类型都不是array的子类型，则引发一个ArrayStoreException异常

表12-2　由Collection 定义的方法

调用 add( )方法可以将对象加入类集。注意 add( )带一个 Object 类型的参数。因为 Object 是所有类的超类，所以任何类型的对象可以被存储在一个类集中。然而原始类型可能不行。例如，一个类集不能直接存储 int，char，double 等类型的值。可以通过调用 addAll( )方法将一个类集的全部内容增加到另一个类集中。

可以通过调用 remove( )方法将一个对象删除。为了删除一组对象，可以调用 removeAll( )方法。调用 retainAll( )方法可以将除了一组指定的元素之外的所有元素删除。为了清空类集，可以调用 clear( )方法。

通过调用 contains( )方法，可以确定一个类集是否包含了一个指定的对象。为了确定一个类集是否包含了另一个类集的全部元素，可以调用 containsAll( )方法。当一个类集是空的时候，可以通过调用 isEmpty( )方法来予以确认。调用 size( )方法可以获得类集中当前元素的个数。

toArray( )方法返回一个数组，这个数组包含了存储在调用类集中的元素。这个方法比它初看上去的能力要更重要。经常使用类数组语法来处理类集的内容是有优势的。通过在类集和数组之间提供一条路径，可以充分利用这两者的优点。

调用 equals( )方法可以比较两个类集是否相等。“相等”的精确含义可以不同于从类集到类集。例如，可以执行 equals( )方法以便用于比较存储在类集中的元素的值，换句话说，equals( )方法能比较对象元素的引用。

一个更加重要的方法是 iterator( )，该方法对类集返回一个迭代程序。当使用一个类集框架时，迭代程序对于编程的成功与否是至关重要的。

12.7.1.2 List 接口

List 接口扩展了 Collection 并声明存储一系列元素的类集的特性。使用一个基于零的下标，元素可以通过它们在列表中的位置被插入和访问。一个列表可以包含复制元素。除了由 Collection 定义的方法之外，List 还定义了一些它自己的方法，这些方法总结在表 12-3 中。需要再次注意当类集不能被修改时，其中的几种方法引发 UnsupportedOperationException 异常。当一个对象与另一个不兼容，例如：当企图将一个不兼容的对象加入一个类集中时，将产生 ClassCastException 异常。

表12-3　由List 定义的方法

方法	描述
Void add(int index,Object obj)	将obj插入调用列表，插入位置的下标由index传递。任何已存在的，在插入点以及插入点之后的元素将前移。因此，没有元素被覆盖
Boolean addAll(int index,Collection c)	将c中的所有元素插入到调用列表中,插入点的下标由index传递。在插入点以及插入点之后的元素将前移。因此，没有元素被覆盖。如果调用列表改变了，则返回true；否则返回false
Object get(int index)	返回存储在调用类集内指定下标处的对象
int indexOf(Object obj)	返回调用列表中obj的第一个实例的下标。如果obj不是列表中的元素，则返回-1
int lastIndexOf(Object obj)	返回调用列表中obj的最后一个实例的下标。如果obj不是列表中的元素，则返回-1
ListIterator listIterator()	返回调用列表开始的迭代程序
ListIterator listIterator(int index)	返回调用列表在指定下标处开始的迭代程序
Object remove(int index)	删除调用列表中index位置的元素并返回删除的元素。删除后，列表被压缩。也就是说，被删除元素后面的元素的下标减一
Object set(int index,Object obj)	用obj对调用列表内由index指定的位置进行赋值
List subList(int start,int end)	返回一个列表，该列表包括了调用列表中从start到end－1的元素。返回列表中的元素也被调用对象引用

对于由 Collection 定义的 add( ) 和 addAll( )方法，List 增加了方法 add(int, Object)和 addAll(int, Collection)。这些方法在指定的下标处插入元素。由 Collection 定义的 add(Object)和 addAll(Collection)的语义也被 List 改变了，以便它们在列表的尾部增加元素。

为了获得在指定位置存储的对象，可以用对象的下标调用 get( )方法。为了给类表中的一个元素赋值，可以调用 set( )方法，指定被改变的对象的下标。调用 indexOf( )或 lastIndexOf( )可以得到一个对象的下标。通过调用 subList( )方法，可以获得列表的

一个指定了开始下标和结束下标的子列表。

12.7.1.3 集合接口

集合接口定义了一个集合。它扩展了 Collection 并说明了不允许复制元素的类集的特性。因此，如果试图将复制元素加到集合中时，add()方法将返回 false。它本身并没有定义任何附加的方法。

12.7.1.4 SortedSet 接口

SortedSet 接口扩展了 Set 并说明了按升序排列的集合的特性。除了那些由 Set 定义的方法之外，由 SortedSet 接口说明的方法列在表 12-4 中。当没有项包含在调用集合中时，其中的几种方法会引发 NoSuchElementException 异常。当对象与调用集合中的元素不兼容时，将引发 ClassCastException 异常。如果试图使用 null 对象，而集合不允许 null 时，会引发 NullPointerException 异常。

表12-4 由SortedSet 定义的方法

方法	描述
Comparator comparator( )	返回调用被排序集合的比较函数，如果对该集合使用自然顺序，则返回null
Object first( )	返回调用被排序集合的第一个元素
SortedSet headSet(Object end)	返回一个包含那些小于end的元素的SortedSet，那些元素包含在调用被排序集合中。返回被排序集合中的元素也被调用被排序集合所引用
Object last( )	返回调用被排序集合的最后一个元素
SortedSet subSet(Object start, Object end)	返回一个SortedSet，它包括了从start到end－1的元素。返回类集中的元素也被调用对象所引用
SortedSet tailSet(Object start)	返回一个SortedSet，它包含了那些包含在分类集合中的大于等于start的元素。返回集合中的元素也被调用对象所引用

SortedSet 定义了几种方法，使得对集合的处理更加方便。调用 first()方法，可以获得集合中的第一个对象。调用 last()方法，可以获得集合中的最后一个元素。调用 subSet()方法，可以获得排序集合的一个指定了第一个和最后一个对象的子集合。如果需要得到从集合的第一个元素开始的一个子集合，可以使用 headSet()方法。如果需要获得集合尾部的一个子集合，可以使用 tailSet()方法。

12.7.2 Collection 接口

现在，读者已经熟悉了类集接口，下面开始讨论实现它们的标准类。一些类提供了完整的可以被使用的工具。另一些类是抽象的，提供主框架工具，作为创建具体类集的起始点。没有一个Collection接口是同步的，在本章后面看到的那样，有可能获得同步版本。标准的Collection实现类总结表12-5中。

表12-5 Collection实现类

类	描述
AbstractCollection	实现大多数Collection接口
AbstractList	扩展AbstractCollection并实现大多数List接口
AbstractSequentialList	为了被类集使用而扩展AbstractList，该类集是连续而不是随机方式访问其元素
LinkedList	通过扩展AbstractSequentialList来实现链接表
ArrayList	通过扩展AbstractList来实现动态数组
AbstractSet	扩展AbstractCollection并实现大多数Set接口
HashSet	为了使用散列表而扩展AbstractSet
TreeSet	实现存储在树中的一个集合。扩展AbstractSet

注意：

除了 Collection 接口之外，还有几个从以前版本遗留下来的类，如 Vector，Stack 和 Hashtable 均被重新设计成支持类集的形式。这些内容将在本章后面讨论。下面讨论具体的 Collection 接口，举例说明它们的用法。

12.7.2.1 ArrayList 类

ArrayList 类扩展 AbstractList 并执行 List 接口。ArrayList 支持可随需要而增长的动态数组。在 Java 中，标准数组是定长的。在数组创建之后，它们不能被加长或缩短，这也就意味着开发者必须事先知道数组可以容纳多少元素。但是，一般情况下，只有在运行时才能知道需要多大的数组。为了解决这个问题，类集框架定义了 ArrayList。本质上，ArrayList 是对象引用的一个变长数组。也就是说，ArrayList 能够动态地增加或减小其大小。数组列表以一个原始大小被创建。当超过了它的大小，类集自动增大。当对象被删除后，数组就可以缩小。注意：动态数组也被从以前版本遗留下来的类 Vector 所支持。关于这一点，将在后面介绍。

ArrayList 有如下的构造方法：

ArrayList( )
ArrayList(Collection c)
ArrayList(int capacity)

其中第一个构造方法建立一个空的数组列表。第二个构造方法建立一个数组列表，该数组列表由类 c 中的元素初始化。第三个构造函数建立一个数组列表，该数组有指定的初始容量（capacity）。容量是用于存储元素的基本数组的大小。当元素被追加到数组列表上时，容量会自动增加。

下面的程序是 ArrayList 的一个简单应用。首先创建一个数组列表，接着添加 String 类型的对象（回想一个引用字符串被转化成一个字符串（String）对象的方法）。接着列表被显示出来。将其中的一些元素删除后，再一次显示列表。

范例：ArrayListDemo.java

```
01  import java.util.*;
02  public class ArrayListDemo
03  {
04      public static void main(String args[])
05      {
06          // 创建一个 ArrarList 对象
07          ArrayList al = new ArrayList();
```

```
08              System.out.println("a1 的初始化大小：" + al.size());
09              // 向 ArrayList 对象中添加新内容
10              al.add("C");          // 0 位置
11              al.add("A");          // 1 位置
12              al.add("E");          // 2 位置
13              al.add("B");          // 3 位置
14              al.add("D");          // 4 位置
15              al.add("F");          // 5 位置
16              // 把 A2 加在 ArrayList 对象的第 2 个位置
17              al.add(1, "A2");      // 加入之后的内容：C A2 A E B D F
18              System.out.println("a1 加入元素之后的大小：" + al.size());
19              // 显示 Arraylist 数据
20              System.out.println("a1 的内容: " + al);
21              // 从 ArrayList 中移除数据
22              al.remove("F");
23              al.remove(2);         // C A2 E B D
24              System.out.println("a1 删除元素之后的大小: " + al.size());
25              System.out.println("a1 的内容: " + al);
26          }
27      }
```

输出结果：

```
a1 的初始化大小：0
a1 加入元素之后的大小：7
a1 的内容: [C, A2, A, E, B, D, F]
a1 删除元素之后的大小: 5
a1 的内容: [C, A2, E, B, D]
```

注意 a1 开始时是空的，当添加元素后，它的大小增加了。当有元素被删除后，它的大小又会变小。在前面的例子中，使用由 toString( )方法提供的默认的转换显示类集的内容，toString( )方法是从 AbstractCollection 继承下来的。它对简短的程序来说是足够了，但很少使用这种方法去显示实际中的类集的内容。通常编程者会提供自己的输出程序。但在下面的几个例子中，仍将采用由 toString( )方法创建的默认输出。

尽管当对象被存储在 ArrayList 对象中时，其容量会自动增加。然而，也可以通过调用 ensureCapacity( )方法来人工地增加 ArrayList 的容量。如果事先知道将在当前能

够容纳的类集中存储许许多多的内容时，读者可能会想这样做。在开始时，通过一次性地增加它的容量，就能避免后面的再分配。因为再分配是很花时间的，避免不必要的处理可以提高性能。

范例：ArrayListToArray.java

```
01  import java.util.*;
02  public class ArrayListToArray
03  {
04      public static void main(String args[])
05      {
06          // 创建一 ArrayList 对象 al
07          ArrayList al = new ArrayList();
08          // 向 ArrayList 中加入对象
09          al.add(new Integer(1));
10          al.add(new Integer(2));
11          al.add(new Integer(3));
12          al.add(new Integer(4));
13          System.out.println("ArrayList 中的内容 ： " + al);
14          // 得到对象数组
15          Object ia[] = al.toArray();
16          int sum = 0;
17          // 计算数组内容
18          for (int i = 0; i < ia.length; i++)
19              sum += ((Integer) ia[i]).intValue();
20          System.out.println("数组累加结果是： " + sum);
21  }
22  }
```

输出结果：

```
ArrayList 中的内容 ： [1, 2, 3, 4]
数组累加结果是： 10
```

程序开始时创建一个整数的类集。由于不能将原始类型存储在类集中，因此类型 Integer 的对象被创建并被保存。接下来，toArray( )方法被调用，它获得了一个 Object

对象数组。这个数组的内容被置成整型（Integer），接下来对这些值进行求和操作。

12.7.2.2 LinkedList 类

LinkedList 类扩展了 AbstractSequentialList 类并实现 List 接口。它提供了一个链接列表的数据结构。它具有如下的两个构造方法，说明如下：

LinkedList( )

LinkedList(Collection c)

第一个构造方法建立一个空的链接列表。第二个构造方法建立一个链接列表，该链接列表由类 c 中的元素初始化。

除了它继承的方法之外，LinkedList 类本身还定义了一些有用的方法，这些方法主要用于操作和访问列表。使用 addFirst( )方法可以在列表头增加元素；使用 addLast( )方法可以在列表的尾部增加元素。它们的形式如下所示：

void addFirst(Object obj)

void addLast(Object obj)

这里，obj 是被增加的项。

调用 getFirst( )方法可以获得第一个元素。调用 getLast( )方法可以得到最后一个元素。它们的形式如下所示：

Object getFirst( )

Object getLast( )

为了删除第一个元素，可以使用 removeFirst( )方法；为了删除最后一个元素，可以调用 removeLast( )方法。它们的形式如下所示：

Object removeFirst( )

Object removeLast( )

下面的程序举例是对几个 LinkedList 支持的方法的说明：

范例：LinkedListDemo.java

```
01  import java.util.*;
02  public class LinkedListDemo
03  {
04      public static void main(String args[])
05      {
06          // 创建 LinkedList 对象
07          LinkedList ll = new LinkedList();
08          // 加入元素到 LinkedList 中
09          ll.add("F");
10          ll.add("B");
11          ll.add("D");
12          ll.add("E");
13          ll.add("C");
14          // 在链表的最后个位置加上数据
15          ll.addLast("Z");
16          // 在链表的第一个位置上加入数据
17          ll.addFirst("A");
18          // 在链表第二个元素的位置上加入数据
19          ll.add(1, "A2");
20          System.out.println("ll 最初的内容：" + ll);
21          // 从 linkedlist 中移除元素
22          ll.remove("F");
23          ll.remove(2);
24          System.out.println("从 ll 中移除内容之后：" + ll);
25          // 移除第一个和最后一个元素
26          ll.removeFirst();
27          ll.removeLast();
28          System.out.println("ll 移除第一个和最后一个元素之后的内容：" + ll);
29          // 取得并设置值
30          Object val = ll.get(2);
31          ll.set(2, (String) val + " Changed");
32          System.out.println("ll 被改变之后：" + ll);
33      }
34  }
```

输出结果：

ll 最初的内容：[A, A2, F, B, D, E, C, Z]

从 ll 中移除内容之后：[A, A2, D, E, C, Z]

ll 移除第一个和最后一个元素之后的内容：[A2, D, E, C]

ll 被改变之后：[A2, D, E Changed, C]

因为 LinkedList 实现 List 接口，调用 add(Object)将项目追加到列表的尾部，如同 addLast( )方法所做的那样。使用 add( )方法的 add(int, Object)形式，插入项目到指定的位置，如例子程序中调用 add（1，“A2”）的举例。

注意如何通过调用 get( )和 set( )方法而使得 ll 中的第三个元素发生了改变。为了获得一个元素的当前值，通过 get( )方法传递存储该元素的下标值。为了对这个下标位置赋一个新值，通过 set( )方法传递下标和对应的新值。

12.7.2.3 HashSet 类

HashSet扩展AbstractSet并且实现Set接口。它创建一个类集，该类集使用散列表进行存储。而散列表通过使用称之为散列法的机制来存储信息。

在散列（hashing）中，一个关键字的信息内容被用来确定唯一的一个值，称为散列码（hash code）。而散列码被用来当做与关键字相连的数据的存储下标。关键字到其散列码的转换是自动执行的——看不到散列码本身。程序代码也不能直接索引散列表。散列法的优点在于即使对于大的集合，它允许一些基本操作，如：add()，contains()，remove()和size( )方法的运行时间保持不变。

下面的构造方法定义为：

```
HashSet( )
HashSet(Collection c)
HashSet(int capacity)
HashSet(int capacity, float fillRatio)
```

第一种形式构造一个默认的散列集合。第二种形式用c中的元素初始化散列集合。

第三种形式用capacity初始化散列集合的容量。第四种形式用它的参数初始化散列集合的容量和填充比（也称为加载容量）。填充比必须介于0.0与1.0之间，它决定在散列集合向上调整大小之前，有多少能被充满。具体的说，就是当元素的个数大于散列集合容量乘以它的填充比时，散列集合被扩大。对于没有获得填充比的构造方法，默认使用0.75。

HashSet没有定义任何超类和接口提供的其它方法。

重要的是，注意散列集合并不能确定其元素的排列顺序，因为散列法的处理通常不让自己参与创建排序集合。如果需要排序存储，另一种类集——TreeSet 将是一个更好的选择。

范例：HashSetDemo.java

```
01  import java.util.*;
02  public class HashSetDemo
03  {
04      public static void main(String args[])
05      {
06          // 创建 HashSet 对象
07          HashSet hs = new HashSet();
08          // 加入元素到 HastSet 中
09          hs.add("B");
10          hs.add("A");
11          hs.add("D");
12          hs.add("E");
13          hs.add("C");
14          hs.add("F");
15          System.out.println(hs);
16      }
17  }
```

输出结果：

```
[D, A, F, C, B, E]
```

如上面解释的那样，元素并没有按顺序进行存储。

12.7.2.4 TreeSet 类

TreeSet 为使用树来进行存储的 Set 接口提供了一个工具，对象按升序存储。访问和检索是很快的。在存储了大量的需要进行快速检索的排序信息的情况下，TreeSet 是一个很好的选择。

下面的构造方法定义为：

TreeSet( )

TreeSet(Collection c)

TreeSet(Comparator comp)

TreeSet(SortedSet ss)

第一种形式构造一个空的树集合，该树集合将根据其元素的自然顺序按升序排序。第二种形式构造一个包含了 c 的元素的树集合。第三种形式构造一个空的树集合，它按照由 comp 指定的比较方法进行排序（比较方法将在本章后面介绍）。第四种形式构造一个包含了 ss 的元素的树集合。下面是一个 TreeSet 的使用范例：

范例：TreeSetDemo.java

```
01  import java.util.*;
02  public class TreeSetDemo
03  {
04      public static void main(String args[])
05      {
06          // 创建一 TreeSet 对象
07          TreeSet ts = new TreeSet();
08          // 加入元素到 TreeSet 中
09          ts.add("C");
10          ts.add("A");
11          ts.add("B");
12          ts.add("E");
13          ts.add("F");
14          ts.add("D");
15          System.out.println(ts);
```

```
16        }
17    }
```

输出结果：

[A, B, C, D, E, F]

正如上面解释的那样，因为 TreeSet 按树存储其元素，它们被按照排序次序自动排序。

12.7.3 通过迭代方法访问类集

通常开发者希望通过循环输出类集中的元素。例如，可能会希望显示每一个元素。到目前为止，处理这个问题的最简单方法是使用 iterator，iterator 是一个或者实现 Iterator 或者实现 ListIterator 接口的对象。Iterator 可以完成通过循环输出类集内容，从而获得或删除元素。ListIterator 扩展 Iterator，允许双向遍历列表，并可以修改单元。Iterator 接口说明的方法总结在表 12-6 中。ListIterator 接口说明的方法总结在表 12-7 中。

表12-6 Iterator 定义的方法

方法	描述
boolean hasNext( )	如果存在更多的元素，则返回true，否则返回false
Object next( )	返回下一个元素。如果没有下一个元素，则引发NoSuchElementException异常
void remove( )	删除当前元素，如果试图在调用next( )方法之后，调用remove( )方法，则引发IllegalStateException异常

表12-7 ListIterator 定义的方法

方法	描述
void add(Object obj)	将obj插入列表中的一个元素之前，该元素在下一次调用next( )方法时，被返回
boolean hasNext( )	如果存在下一个元素，则返回true；否则返回false
boolean hasPrevious( )	如果存在前一个元素，则返回true；否则返回false
Object next( )	返回下一个元素，如果不存在下一个元素，则引发一个NoSuchElementException异常
int nextIndex( )	返回下一个元素的下标，如果不存在下一个元素，则返回列表的大小
Object previous( )	返回前一个元素，如果前一个元素不存在，则引发一个NoSuchElementException异常
int previousIndex( )	返回前一个元素的下标，如果前一个元素不存在，则返回-1
void remove( )	从列表中删除当前元素。如果remove( )方法在next( )方法或previous( )方法调用之前被调用，则引发一个IllegalStateException异常
void set(Object obj)	将obj赋给当前元素。这是上一次调用next( )方法或previous( )方法最后返回的元素

12.7.3.1 使用迭代方法

在通过迭代方法访问类集之前，必须得到一个迭代方法。每一个 Collection 类都提供一个 iterator( )方法，该方法返回一个对类集的迭代方法。通过使用这个迭代方法对象，可以一次一个地访问类集中的每一个元素。通常，使用迭代方法通过循环输出类集的内容，步骤如下：

1、 通过调用类集的 iterator( )方法获得对类集的迭代方法。

2、 建立一个调用 hasNext( )方法的循环，只要 hasNext( )返回 true，就进行循环迭代。

3、 在循环内部，通过调用 next( )方法来得到每一个元素。

对于执行 List 的类集，也可以通过调用 ListIterator 来获得迭代方法。正如上面解释的那样，列表迭代方法提供了前向或后向访问类集的能力，并可以修改元素。否则，ListIterator 如同 Iterator 功能一样。程序 IteratorDemo.java 是一个实现上述描述的例子，说明了 Iterator 和 ListIterator 的使用方法。在范例中使用的是 ArrayList 类，但是原则上它是适用于任何类型的类集的。当然，ListIterator 只适用于那些实现 List 接口的类集。

范例：IteratorDemo.java

```
01  import java.util.* ;
02  public class IteratorDemo
03  {
04      public static void main(String args[])
05      {
06          // 创建一个 ArrayList 数组
07          ArrayList al = new ArrayList();
08          // 加入元素到 ArrayList 中
09          al.add("C");
10          al.add("A");
11          al.add("E");
12          al.add("B");
13          al.add("D");
14          al.add("F");
15          // 使用 iterator 显示 a1 中的内容
16          System.out.print("a1 中原始内容是：");
17          Iterator itr = al.iterator();
18          while (itr.hasNext())
19          {
20              Object element = itr.next();
21              System.out.print(element + " ");
22          }
23          System.out.println();
24          // 在 ListIterator 中修改内容
25          ListIterator litr = al.listIterator();
26          while (litr.hasNext())
```

```
27              {
28                  Object element = litr.next();
29                  // 用 set 方法修改其内容
30                  litr.set(element + "+");
31              }
32              System.out.print("a1 被修改之后的内容：");
33              itr = al.iterator();
34              while (itr.hasNext())
35              {
36                  Object element = itr.next();
37                  System.out.print(element + " ");
38              }
39              System.out.println();
40              // 下面是将列表中的内容反向输出
41              System.out.print("将列表反向输出：");
42              // hasPreviours 由后向前输出
43              while (litr.hasPrevious())
44              {
45                  Object element = litr.previous();
46                  System.out.print(element + " ");
47              }
48              System.out.println();
49          }
50      }
```

输出结果：

a1 中原始内容是：C A E B D F

a1 被修改之后的内容：C+ A+ E+ B+ D+ F+

将列表反向输出：F+ D+ B+ E+ A+ C+

值得注意的是列表是如何被反向显示的。在列表被修改之后，litr 指向列表的末端（记住，当到达列表末端时，litr.hasNext( )方法返回 false）。为了反向遍历列表，程序继续使用 litr，但这一次，程序将检测它是否有前一个元素，只要它有前一个元素，该元素就被获得并被显示出来。

12.7.4 处理映射

除了类集，Java 2 还在 java.util 中增加了映射。映射（map）是一个存储关键字和值的关联或者说是“关键字/值”对的对象，即给定一个关键字，可以得到它的值。关键字和值都是对象，关键字必须是唯一的，但值是可以被复制的。有的映射可以接收 null 关键字和 null 值，有的则不能。

12.7.4.1 映射接口

正因为映射接口定义了映射的特征和本质，所以从这里开始讨论映射。表12-8所列为支持映射的接口：

表12-8 支持映射的接口

接口	描述
Map	映射唯一关键字给值
Map.Entry	描述映射中的元素（“关键字/值”对）。是Map的一个内部类
SortedMap	扩展Map以便关键字按升序保持

下面对每个接口依次进行讨论。

Map 接口

Map接口映射唯一关键字到值。关键字（key）是以后用于检索值的对象。给定一个关键字和一个值，可以存储这个值到一个Map对象中。当这个值被存储以后，就可以使用它的关键字来检索它。由Map说明的方法总结在表12-9中。当调用的映射中没有项存在时，其中的几种方法会引发一个NoSuchElementException异常。而当对象与映射中的元素不兼容时，引发一个ClassCastException异常。如果试图使用映射不允

许使用的null对象时，则引发一个NullPointerException异常。当试图改变一个不允许修改的映射时，则引发一个UnsupportedOperationException异常。

表12-9 Map接口 定义的方法

方法	描述
void clear()	从调用映射中删除所有的“关键字/值”对
boolean containsKey(Object k)	如果调用映射中包含了作为关键字的k，则返回true；否则返回false
boolean containsValue(Object v)	如果映射中包含了作为值的v，则返回true；否则返回false
Set entrySet()	返回包含了映射中的项的集合（Set）。该集合包含了类型Map.Entry的对象。这个方法为调用映射提供了一个集合“视图”
Boolean equals(Object obj)	如果obj是一个Map并包含相同的输入，则返回true；否则返回false
Object get(Object k)	返回与关键字k相关联的值
int hashCode()	返回调用映射的散列码
boolean isEmpty()	如果调用映射是空的，则返回true；否则返回false
Set keySet()	返回一个包含调用映射中关键字的集合（Set）。这个方法为调用映射的关键字提供了一个集合“视图”
Object put(Object k,Object v)	将一个输入加入调用映射，覆盖原先与该关键字相关联的值。关键字和值分别为k和v。如果关键字已经不存在了，则返回null；否则，返回原先与关键字相关联的值
void putAll(Map m)	将所有来自m的输入加入调用映射
Object remove(Object k)	删除关键字等于k的输入
int size()	返回映射中“关键字/值”对的个数

Collection values()	返回一个包含了映射中的值的类集。这个方法为映射中的值提供了一个类集“视图”映射循环使用两个基本操作：get()和put()。使用put()方法可以将一个指定了关键字和其对应的值加入映射。为了得到值，可以通过将关键字作为参数来调用get()方法，调用返回该值。

正如前面谈到的，映射不是类集，但可以获得映射的类集“视图”。为了实现这种功能，可以使用entrySet( )方法，它返回一个包含了映射中元素的集合（Set）。为了得到关键字的类集“视图”，可以使用keySet( )方法。为了得到值的类集“视图”，可以使用values()方法。类集“视图”是将映射集成到类集框架内的手段。

SortedMap 接口

SortedMap接口扩展了Map，它确保了各项按关键字升序排序。由SortedMap说明的方法总结在表12-10中。当调用映射中没有的项时，其中的几种方法引发一个NoSuchElementException异常。当对象与映射中的元素不兼容时，则引发一个ClassCastException异常。当试图使用映射不允许使用的null对象时，则引发一个ullPointerException异常。

表12-10 SortedMap接口定义的方法

方法	描述
Comparator comparator()	返回调用排序映射的比较方法。如果调用映射使用的是自然顺序的话，则返回null
Object firstKey()	返回调用映射的第一个关键字
SortedMap headMap(Object end)	返回一个排序映射，该映射包含了那些关键字小于end的映射输入
Object lastKey()	返回调用映射的最后一个关键字
SortedMap subMap(Object start,Object end)	返回一个映射，该映射包含了那些关键字大于等于start同时小于end的输入

SortedMap tailMap(Object start)	返回一个映射，该映射包含了那些关键字大于等于start的输入排序映射允许对子映射（换句话说，就是映射的子集）进行高效的处理。使用headMap()，tailMap()或subMap()方法可以获得子映射。调用firstKey()方法可以获得集合的第一个关键字。而调用lastKey()方法可以获得集合的最后一个关键字。

Map.Entry 接口

Map.Entry接口可以操作映射的输入。回想由Map接口说明的entrySet( )方法，调用该方法返回一个包含映射输入的集合（Set）。这些集合元素的每一个都是一个Map.Entry对象。表12-11总结了由该接口说明的方法。

表12-11 Map.Entry 定义的方法

方法	描述
boolean equals(Object obj)	如果obj是一个关键字和值都与调用对象相等的Map.Entry，则返回true
Object getKey()	返回该映射项的关键字
Object getValue()	返回该映射项的值
int hashCode()	返回该映射项的散列值
Object setValue(Object v)	将这个映射输入的值赋给v。如果v不是映射的正确类型，则引发一个ClassCastException异常。如果v存在问题，则引发一个IllegalArgumentException异常。如果v是null而映射又不允许null关键字，则引发一个NullPointerException异常。如果映射不能被改变，则引发一个UnsupportedOperationException异常。

12.7.4.2 映射类

有几个类提供了映射接口的实现。可以被用做映射的类如表12-12所示：

表12-12 映射接口的实现类

类	描述

AbstractMap	实现大多数的Map接口
HashMap	将AbstractMap扩展到使用散列表
TreeMap	将AbstractMap扩展到使用树

注意AbstractMap对三个具体的映射实现来说，是一个超类。WeakHashMap实现一个使用“弱关键字”的映射，当该映射的关键字不再被使用时，它允许映射中的元素被放入回收站。关于这个类，在这里不做更深入的讨论。其它的类将在下面介绍。

HashMap 类

HashMap类使用散列表实现Map接口。这允许一些基本操作，如：get( )和put( )的运行时间保持恒定，即便对大型集合，也是这样的。下面的构造方法定义为：

```
HashMap( )
HashMap(Map m)
HashMap(int capacity)
HashMap(int capacity, float fillRatio)
```

第一种形式构造一个默认的散列映射。第二种形式用m的元素初始化散列映射。第三种形式将散列映射的容量初始化为capacity。第四种形式用它的参数同时初始化散列映射的容量和填充比。容量和填充比的含义与前面介绍的HashSet中的容量和填充比相同。

HashMap实现Map并扩展AbstractMap。它本身并没有增加任何新的方法。应该注意的是散列映射并不保证它的元素的顺序。因此，元素加入散列映射的顺序并不一定是它们被迭代方法读出的顺序。

下面的程序举例说明了 HashMap。它将名字映射到账目资产平衡表。注意集合“视图”是如何获得和被使用的。

范例：HashMapDemo.java

```
01  import java.util.* ;
02  public class HashMapDemo
03  {
04      public static void main(String args[])
05      {
```

```
06              // 创建 HashMap 对象
07              HashMap hm = new HashMap();
08              // 加入元素到 HashMap 中
09              hm.put("John Doe", new Double(3434.34));
10              hm.put("Tom Smith", new Double(123.22));
11              hm.put("Jane Baker", new Double(1378.00));
12              hm.put("Todd Hall", new Double(99.22));
13              hm.put("Ralph Smith", new Double(-19.08));
14              // 返回包含映射中项的集合
15              Set set = hm.entrySet();
16              // 用 Iterator 得到 HashMpa 中的项
17              Iterator i = set.iterator();
18              // 显示元素
19              while (i.hasNext())
20              {
21                  // Map.Entry 可以操作映射的输入
22                  Map.Entry me = (Map.Entry) i.next();
23                  System.out.print(me.getKey() + ": ");
24                  System.out.println(me.getValue());
25              }
26              System.out.println();
27              // 让 John Doe 中的值增加 1000
28              double balance = ((Double) hm.get("John Doe")).doubleValue();
29              // 用新的值替换掉旧的值
30              hm.put("John Doe", new Double(balance + 1000));
31              System.out.println("John Doe's 现在的资金：" + hm.get("John Doe"));
32          }
33      }
```

输出结果：

Todd Hall: 99.22

Ralph Smith: -19.08

Tom Smith: 123.22

John Doe: 3434.34

Jane Baker: 1378.0

John Doe's 现在的资金：4434.34

程序开始创建一个散列映射，然后将名字的映射增加到平衡表中。接下来，映射的内容通过使用由调用方法 entrySet( )而获得的集合“视图”而显示出来。关键字和值通过调用由 Map.Entry 定义的 getKey( )和 getValue( )方法而显示。注意存款是如何被制成 John Doe 的账目的。put( )方法自动用新值替换与指定关键字相关联的原先的值。因此，在 John Doe 的账目被更新后，散列映射将仍然仅仅保留一个“John Doe”账目。

TreeMap 类

TreeMap 类通过使用树实现 Map 接口。TreeMap 提供了按排序顺序存储“关键字/值”对的有效手段，同时允许快速检索。应该注意的是，不像散列映射，树映射保证它的元素按照关键字升序排序。

下面的 TreeMap 构造方法定义为：

TreeMap( )

TreeMap(Comparator comp)

TreeMap(Map m)

TreeMap(SortedMap sm)

第一种形式构造一个空树的映射，该映射使用其关键字的自然顺序来排序。第二种形式构造一个空的基于树的映射，该映射通过使用 Comparator comp 来排序（比较方法 Comparators 将在本章后面进行讨论）。第三种形式用从 m 的输入初始化树映射，该映射使用关键字的自然顺序来排序。第四种形式用从 sm 的输入来初始化一个树映射，该映射将按与 sm 相同的顺序来排序。

TreeMap 实现 SortedMap 并且扩展 AbstractMap。而它本身并没有另外定义其它方法。

下面的程序重新运行前面的范例，以便在其中使用 TreeMap：

范例：TreeMapDemo.java

```
01  import java.util.* ;
02  public class TreeMapDemo
03  {
```

```
04      public static void main(String args[])
05      {
06          // 创建 TreeMap 对象
07          TreeMap tm = new TreeMap();
08          // 加入元素到 TreeMap 中
09          tm.put(new Integer(10000 - 2000), "张三");
10          tm.put(new Integer(10000 - 1500), "李四");
11          tm.put(new Integer(10000 - 2500), "王五");
12          tm.put(new Integer(10000 - 5000), "赵六");
13          Collection col = tm.values();
14          Iterator i = col.iterator();
15          System.out.println("按工资由高到低顺序输出：");
16          while (i.hasNext())
17          {
18              System.out.println(i.next());
19          }
20      }
21  }
```

输出结果：

```
按工资由高到低顺序输出：
赵六
王五
张三
李四
```

注意对关键字进行了排序。然而，在这种情况下，是用名字而不是用姓进行了排序。可以通过在创建映射时，指定一个比较方法来改变这种排序。在下一节将介绍如何做。

12.7.4.3 比较方法

TreeSet 和 TreeMap 都按排序顺序存储元素。然而，精确定义采用何种“排序顺序”的是比较方法。通常在默认的情况下，这些类通过使用被 Java 称之为“自然顺序”的

顺序存储它们的元素，而这种顺序通常也是所需要的（A 在 B 的前面，1 在 2 的前面，等等）。如果需要用不同的方法对元素进行排序，可以在构造集合或映射时，指定一个 Comparator 对象。这样做为开发者提供了一种精确控制如何将元素储存到排序类集和映射中的功能。

Comparator 接口定义了两个方法：compare( )和 equals( )。这里给出的 compare( )方法按顺序比较了两个元素：

int compare(Object obj1, Object obj2)

obj1 和 obj2 是被比较的两个对象。当两个对象相等时，该方法返回 0；当 obj1 大于 obj2 时，返回一个正值；否则，返回一个负值。如果用于比较的对象的类型不兼容的话，该方法引发一个 ClassCastException 异常。通过覆盖 compare( )，可以改变对象排序的方式。例如，通过创建一个颠倒比较输出的比较方法，可以实现按逆向排序。

这里给出的 equals( )方法，测试一个对象是否与调用比较方法相等：

boolean equals(Object obj)

obj 是被用来进行相等测试的对象。如果 obj 和调用对象都是 Comparator 的对象并且使用相同的排序。该方法返回 true。否则返回 false。重载 equals( )方法是没有必要的，大多数简单的比较方法都不这样做。

下面是一个说明定制的比较方法能力的例子。该例子实现 compare( )方法以便它按正常顺序的逆向进行操作。因此，它使得一个树集合按逆向的顺序进行存储。

范例：ComparatorDemo.java

```
01  import java.util.*;
02  class MyComp implements Comparator
03  {
04      public int compare(Object o1, Object o2)
05      {
06          String aStr, bStr;
07          aStr = (String)o1;
08          bStr = (String)o2;
09          return bStr.compareTo(aStr);
10      }
11  }
```

```
12  public class ComparatorDemo
13  {
14      public static void main(String args[])
15      {
16          // 创建一个 TreeSet 对象
17          TreeSet ts = new TreeSet(new MyComp());
18          // 向 TreeSet 对象中加入内容
19          ts.add("C");
20          ts.add("A");
21          ts.add("B");
22          ts.add("E");
23          ts.add("F");
24          ts.add("D");
25          // 得到 Iterator 的实例化对象
26          Iterator i = ts.iterator();
27          // 显示全部内容
28          while (i.hasNext())
29          {
30              Object element = i.next();
31              System.out.print(element + " ");
32          }
33      }
34  }
```

输出结果：

F E D C B A

仔细观察实现 Comparator 并覆盖 compare( )方法的 MyComp 类（正如前面所解释的那样，覆盖 equals( )方法既不是必须的，也不是常用的）。在 compare( )方法内部，用 String 类中的 compareTo( )比较两个字符串。然而由 bStr——不是 aStr——调用 compareTo( )方法，这导致比较的结果被逆向。

下面是一个更实际的范例，是用 TreeMap 程序实现前面介绍的存储账目资产平衡表例子的程序。下面的程序按姓对账目进行排序。为了实现这种功能，程序使用了比较方法来比较每一个账目下姓的排序，得到的映射是按姓进行排序的。

范例：TreeMapDemo2.java

```
01  import java.util.* ;
02  class Employee implements Comparator
03  {
04      public int compare(Object a, Object b)
05      {
06          int k;
07          String aStr, bStr;
08          aStr = (String) a;
09          bStr = (String) b;
10          k = aStr.compareTo(bStr);
11          if(k==0)
12              return aStr.compareTo(bStr);
13          else
14              return k;
15      }
16  }
17
18  public class TreeMapDemo2
19  {
20      public static void main(String args[])
21      {
22          // 创建一个 TreeMap 对象
23          TreeMap tm = new TreeMap(new Employee());
24          // 向 Map 对象中插入元素
25          tm.put("Z、张三", new Double(3534.34));
26          tm.put("L、李四", new Double(126.22));
27          tm.put("W、王五", new Double(1578.40));
28          tm.put("Z、赵六", new Double(99.62));
29          tm.put("S、孙七", new Double(-29.08));
30          Set set = tm.entrySet();
31          Iterator itr = set.iterator();
32          while(itr.hasNext())
33          {
34              Map.Entry me = (Map.Entry)itr.next();
35              System.out.print(me.getKey() + ": ");
36              System.out.println(me.getValue());
37          }
```

```
38              System.out.println();
39              double balance = ((Double)tm.get("Z、张三")).doubleValue();
40              tm.put("Z、张三", new Double(balance + 2000));
41              System.out.println("张三最新的资金数为：  " + tm.get("Z、张三"));
42          }
43  }
```

输出结果：

L、李四: 126.22

S、孙七: -29.08

W、王五: 1578.4

Z、张三: 3534.34

Z、赵六: 99.62

张三最新的资金数为： 5534.34

12.7.5 从以前版本遗留下来的类和接口

java.util 的最初版本中不包括类集框架。取而代之，它定义了几个类和接口提供专门的方法用于存储对象。随着在 Java 2 中引入类集，有几种最初的类被重新设计成支持类集接口。因此它们与框架完全兼容。尽管实际上没有类被摈弃，但其中某些仍被认为是过时的。当然，在那些重复从以前版本遗留下来的类的功能的地方，通常都愿意用类集编写新的代码程序。一般地，对从以前版本遗留下来的类的支持是因为仍然存在大量使用它们的基本代码。包括现在仍在被 Java 2 的应用编程接口（API）使用的程序。

另一点，没有一个类集类是同步的。但是所有的从以前版本遗留下来的类都是同步的。这一区别在有些情况下是很重要的。当然，通过使用由 Collections 提供的算法也很容易实现类集同步。

由 java.util 定义的从以前版本遗留下来的类说明如下：

Dictionary　　Hashtable　　Properties　　Stack　　Vector

有一个枚举（Enumeration）接口是从以前版本遗留下来。在下面依次介绍 Enumeration 和每一种从以前版本遗留下来的类。

12.7.5.1 Enumeration 接口

Enumeration 接口定义了可以对一个对象的类集中的元素进行枚举（一次获得一个）的方法。这个接口尽管没有被摈弃，但已经被 Iterator 所替代。Enumeration 对新程序来说是过时的。然而它仍被几种从以前版本遗留下来的类（例如 Vector 和 Properties）所定义的方法使用，被几种其它的 API 类所使用以及被目前广泛使用的应用程序所使用。

Enumeration 指定下面的两个方法：

boolean hasMoreElements( )

Object nextElement( )

执行后，当仍有更多的元素可提取时，hasMoreElements( )方法一定返回 true。当所有元素都被枚举了，则返回 false。nextElement( )方法将枚举中的下一个对象做为一个类属 Object 的引用而返回。也就是每次调用 nextElement( )方法获得枚举中的下一个对象。调用例程必须将那个对象置为包含在枚举内的对象类型。

12.7.5.2 Vector

Vector 实现动态数组。这与 ArrayList 相似，但两者不同的是：Vector 是同步的，并且它包含了许多不属于类集框架的从以前版本遗留下来的方法。随着 Java 2 的公布，Vector 被重新设计来扩展 AbstractList 和实现 List 接口，因此现在它与类集是完全兼容的。

这里是 Vector 的构造方法：

Vector( )
Vector(int size)
Vector(int size, int incr)
Vector(Collection c)

第一种形式创建一个原始大小为 10 的默认矢量。第二种形式创建一个其原始容量由 size 指定的矢量。第三种形式创建一个其原始容量由 size 指定，并且它的增量由 incr 指定的矢量。增量指定了矢量每次允许向上改变大小的元素的个数。第四种形式创建一个包含了类集 c 中元素的矢量。这个构造方法是在 Java 2 中新增加的。

所有的矢量开始都有一个原始的容量。在这个原始容量达到以后，下一次再试图向矢量中存储对象时，矢量自动为那个对象分配空间同时为别的对象增加额外的空间。通过分配超过需要的内存，矢量减小了可能产生的分配的次数。这种次数的减少是很重要的，因为分配内存是很花时间的。在每次再分配中，分配的额外空间的总数由在创建矢量时指定的增量来确定。如果没有指定增量，在每个分配周期，矢量的大小增一倍。

Vector 定义了下面的保护数据成员：

```
int capacityIncrement;
int elementCount;
Object elementData[ ];
```

增量值被存储在 capacityIncrement 中。矢量中的当前元素的个数被存储在 elementCount 中。保存矢量的数组被存储在 elementData 中。

除了由 List 定义的类集方法之外，Vector 还定义了几个从以前版本遗留下来的方法，这些方法列在表 12-13 中：

表 12-13 由 Vector 定义的方法

方法	描述
final void addElement(Object element)	将由element指定的对象加入矢量
int capacity()	返回矢量的容量
Object clone()	返回调用矢量的一个拷贝
Boolean contains(Object element)	如果element被包含在矢量中，则返回true；如果不包含于其中，则返回false
void copyInto(Object array[])	将包含在调用矢量中的元素复制到由array指定的数组中
Object elementAt(int index)	返回由index指定位置的元素
Enumeration elements()	返回矢量中元素的一个枚举

Object firstElement()	返回矢量的第一个元素
int indexOf(Object element)	返回element首次出现的位置下标。如果对象不在矢量中，则返回-1
int indexOf(Object element,int start)	返回element在矢量中在start及其之后第一次出现的位置下标。如果该对象不属于矢量的这一部分，则返回-1
void insertElementAt(Object element,int index)	在矢量中，在由index指定的位置处加入element
boolean isEmpty()	如果矢量是空的，则返回true。如果它包含了一个或更多个元素，则返回false
Object lastElement()	返回矢量中的最后一个元素
int lastIndexOf(Object element)	返回element在矢量中最后一次出现的位置下标。如果对象不包含在矢量中，则返回-1
int lastIndexOf(Object element,int start)	返回element在矢量中，在start之前最后一次出现的位置下标。如果该对象不属于矢量的这一部分，则返回-1
void removeAllElements()	清空矢量，在这个方法执行以后，矢量的大小为0
boolean removeElement(Object element)	从矢量中删除element。对于指定的对象，矢量中如果有其多个实例，则其中第一个实例被删除。如果成功删除，则返回true；如果没有发现对象，则返回false
void removeElementAt(int index)	删除由index指定位置处的元素
void setElementAt(Object element,int index)	将由index指定的位置分配给element
void setSize(int size)	将矢量中元素的个数设为size。如果新的长度小于旧的长度，元素将丢失；如果新的长度大于的长度，则在其后增加null元素
int size()	返回矢量中当前元素的个数
String toString()	返回矢量的字符串等价形式
void trimToSize()	将矢量的容量设为与其当前拥有的元素的个数相等

因为 Vector 实现 List，所以可以像使用 ArrayList 的一个实例那样使用矢量。也可以使用它的从以前版本遗留下来的方法来操作它。例如，在后面实例化 Vector，可以通过调用 addElement( )方法而为其增加一个元素。调用 elementAt( )方法可以获得指定位置处的元素。

调用 firstElement( )方法可以得到矢量的第一个元素。调用 lastElement( )方法可以检索到矢量的最后一个元素。使用 indexOf( ) 和 lastIndexOf( ) 方法可以获得元素的下标。调用 removeElement( )或 removeElementAt( )方法可以删除元素。

下面的程序使用矢量存储不同类型的数值对象。程序说明了几种由 Vector 定义的从以前版本遗留下来的方法，同时它也说明了枚举（Enumeration）接口。

范例：VectorDemo.java

```
01  import java.util.* ;
02  public class VectorDemo
03  {
04      public static void main(String[] args)
05      {
06          Vector v = new Vector() ;
07          v.add("A") ;
08          v.add("B") ;
09          v.add("C") ;
10          v.add("D") ;
11          v.add("E") ;
12          v.add("F") ;
13          Enumeration e = v.elements() ;
14          while(e.hasMoreElements())
15          {
16              System.out.print(e.nextElement()+"\t") ;
17          }
18      }
19  }
```

输出结果：

```
A   B   C   D   E   F
```

随着 Java 2 的公布，Vector 增加了对迭代方法的支持。现在可以使用迭代方法来替代枚举去遍历对象（正如前面的程序所做的那样）。例如，下面的基于迭代方法的程序代码可以被替换到上面的程序中：

```
Iterator i = v.iterator() ;
```

```
while(i.hasNext())
{
    System.out.print(i.next()+"\t") ;
}
```

建议不要使编写枚举新的程序代码，所以通常可以使用迭代方法来对矢量的内容进行枚举。当然，业已存在的大量的老程序采用了枚举。不过幸运的是，枚举和迭代方法的工作方式几乎相同。

12.7.5.3 Stack

Stack 是 Vector 的一个子类，它实现标准的后进先出堆栈。Stack 仅仅定义了创建空堆栈的默认构造方法。Stack 包括了由 Vector 定义的所有方法，同时增加了几种它自己定义的方法，具体总结在表 12-14 中。

表12-14 由Stack 定义的方法

方法	描述
boolean empty()	如果堆栈是空的，则返回true，当堆栈包含有元素时，返回false
Object peek()	返回位于栈顶的元素，但是并不在堆栈中删除它
Object pop()	返回位于栈顶的元素，并在进程中删除它
Object push(Object element)	将element压入堆栈，同时也返回element
int search(Object element)	在堆栈中搜索element，如果发现了，则返回它相对于栈顶的偏移量。否则，返回-1调用push()方法可将一个对象压入栈顶。调用pop()方法可以删除和返回栈顶的元素。当调用堆栈是空的时，如果调用pop()方法，将引发一个EmptyStackException异常。调用peek()方法返回但不删除栈顶的对象。调用empty()方法，当堆栈中没有元素时，返回true。
search()	方法确定一个对象是否存在于堆栈，并且返回将其指向栈顶所需的弹出次数。

下面是一个创建堆栈的范例，在范例中，将几个整型（Integer）对象压入堆栈，然后再将它们弹出。

范例：StackDemo.java

```
01  import java.util.* ;
02  public class StackDemo
03  {
04      static void showpush(Stack st, int a)
05      {
06          st.push(new Integer(a));
07          System.out.println("入栈(" + a + ")");
08          System.out.println("Stack: " + st);
09      }
10
11      static void showpop(Stack st)
12      {
13          System.out.print("出栈  -> ");
14          Integer a = (Integer) st.pop();
15          System.out.println(a);
16          System.out.println("Stack: " + st);
17      }
18
19      public static void main(String args[])
20      {
21          Stack st = new Stack();
22          System.out.println("Stack: " + st);
23          showpush(st, 42);
24          showpush(st, 66);
25          showpush(st, 99);
26          showpop(st);
27          showpop(st);
28          showpop(st);
29          // 出栈时会有一个 EmptyStackException 的异常，需要进行异常处理
30          try {
31              showpop(st);
32          } catch (EmptyStackException e) {
33              System.out.println("出现异常：栈中内容为空");
```

```
34              }
35          }
36      }
```

下面是由该程序产生的输出。注意对于 EmptyStackException 的异常处理程序是如何被捕获的，以便于能够从容地处理堆栈的下溢。

输出结果：

```
Stack: []
入栈(42)
Stack: [42]
入栈(66)
Stack: [42, 66]
入栈(99)
Stack: [42, 66, 99]
出栈  -> 99
Stack: [42, 66]
出栈  -> 66
Stack: [42]
出栈  -> 42
Stack: []
出栈  ->  出现异常：栈中内容为空
```

12.7.5.4 Dictionary

字典（Dictionary）是一个表示“关键字/值”存储库的抽象类，同时它的操作也很像映射（Map）。给定一个关键字和值，可以将值存储到字典（Dictionary）对象中。一旦这个值被存储了，就能够用它的关键字来检索它。因此，与映射一样，字典可以被当做“关键字/值”对列表来考虑。尽管在 Java 2 中并没有摈弃字典（Dictionary），由于它被映射（Map）所取代，从而被认为是过时的。然而由于目前 Dictionary 被广泛地使用，因此这里仍对它进行详细的讨论。

由 Dictionary 定义的抽象方法在表 12-15 中列出。

表12-15 由Dictionary 定义的抽象方法

方法	描述
Enumeration elements()	返回对包含在字典中的值的枚举
Object get(Object key)	返回一个包含与key相连的值的对象。如果key不在字典中，则返回一个空对象
boolean isEmpty()	如果字典是空的，则返回true；如果字典中至少包含一个关键字，则返回false
Enumeration keys()	返回包含在字典中的关键字的枚举
Object put(Object key,Object value)	将一个关键字和它的值插入字典中。如果key已经不在字典中了,则返回null;如果key已经在字典中了，则返回与key相关联的前一个值
Object remove(Object key)	删除key和它的值。返回与key相关联的值。如果key不在字典中，则返回null
int size()	返回字典中的项数使用put()方法在字典中增加关键字和值。使用get()方法检索给定关键字的值。当分别使用keys()和elements()方法进行枚举（Enumeration）时，关键字和值可以分别逐个地返回。
size()	方法返回存储在字典中的“关键字/值”对的个数。当字典是空的时候，isEmpty()返回true。使用remove()方法可以删除“关键字/值”对。

注意：

Dictionary 类是过时的。应该执行 Map 接口去获得“关键字/值”存储的功能。

12.7.5.5 Hashtable

散列表（Hashtable）是原始java.util中的一部分同时也是Dictionary的一个具体实现。然而，Java 2重新设计了散列表（Hashtable）以便它也能实现映射（Map）接口。因此现在Hashtable也被集成到类集框架中。它与HashMap相似，但它是同步的。和HashMap一样，Hashtable将“关键字/值”对存储到散列表中。使用Hashtable时，指定一个对象作为关键字，同时指定与该关键字相关联的值。接着该关键字被散列，

而把得到的散列值作为存储在表中的值的下标。

散列表仅仅可以存储重载由Object定义的hashCode()和equals()方法的对象。hashCode()方法计算和返回对象的散列码。当然，equals()方法比较两个对象。幸运的是，许多Java内置的类已经实现了hashCode()方法。例如，大多数常见的Hashtable类型使用字符串（String）对象作为关键字。String实现hashCode()和equals()方法。

Hashtable的构造方法如下所示：

```
Hashtable( )
Hashtable(int size)
Hashtable(int size, float fillRatio)
Hashtable(Map m)
```

第一种形式是默认的构造方法。第二种形式创建一个散列表，该散列表具有由size指定的原始大小。第三种形式创建一个散列表，该散列表具有由size指定的原始大小和由fillRatio指定的填充比。填充比必须介于0.0和1.0之间，它决定了在散列表向上调整大小之前散列表的充满度。具体地说，当元素的个数大于散列表的容量乘以它的填充比时，散列表被扩展。

如果没有指定填充比，默认使用0.75。最后，第四种形式创建一个散列表，该散列表用m中的元素初始化。散列表的容量被设为m中元素的个数的两倍。默认的填充因子设为0.75。第四种构造方法是在Java 2中新增加的。

除了Hashtable目前实现的由Map接口定义的方法之外，Hashtable定义的从以前版本遗留下来的方法列在表12-16中。

表12-16 由Hashtable 定义的从以前版本遗留下来的方法

方法	描述
void clear()	复位并清空散列表
Object clone()	返回调用对象的复制
boolean contains(Object value)	如果一些值与存在于散列表中的value相等的话，则返回true；如果这个值不存在，则返回false

boolean containsKey(Object key)	如果一些关键字与存在于散列表中的key相等的话，则返回true；如果这个关键字不存在，则返回false
boolean containsValue(Object value)	如果一些值与散列表中存在的value相等的话，返回true；如果这个值没有找到，则返回false（是一种为了保持一致性而在Java 2中新增加的非Map方法）
Enumeration elements()	返回包含在散列表中的值的枚举
Object get(Object key)	返回包含与key相关联的值的对象。如果key不在散列表中，则返回一个空对象
boolean isEmpty()	如果散列表是空的，则返回true；如果散列表中至少包含一个关键字，则返回false
Enumeration keys()	返回包含在散列表中的关键字的枚举
Object put(Object key,Object value)	将关键字和值插入散列表中。如果key已经不在散列表中，返回null。如果key已经存在于散列表中，则返回与key相连的前一个值
void rehash()	增大散列表的大小并且对其关键字进行再散列。
Object remove(Object key)	删除key及其对应的值。返回与key相关联的值。如果key不在散列表中，则返回一个空对象
int size()	返回散列表中的项数
String toString()	返回散列表的等价字符串形式

范例：HashtableDemo.java

```
01  import java.util.* ;
02  public class HashtableDemo
03  {
04      public static void main(String[] args)
05      {
06          Hashtable numbers = new Hashtable();
07          numbers.put("one", new Integer(1));
08          numbers.put("two", new Integer(2));
09          numbers.put("three", new Integer(3));
10          Integer n = (Integer) numbers.get("two");
11          if (n != null)
12          {
13              System.out.println("two = " + n);
14          }
15      }
```

```
16  }
```

输出结果：

```
two = 2
```

12.7.5.5 Properties

属性（Properties）是Hashtable的一个子类。它用来保持值的列表，在其中关键字和值都是字符串（String）。Properties类被许多其它的Java类所使用。例如，当获得系统环境值时，System.getProperties( )返回对象的类型。

Properties定义了下面的实例变量：

```
Properties defaults;
```

这个变量包含了一个与属性（Properties）对象相关联的默认属性列表。Properties定义了如下的构造方法：

```
Properties( )
Properties(Properties propDefault)
```

第一种形式创建一个没有默认值的属性（Properties）对象。第二种形式创建一个将propDefault作为其默认值的对象。在这两种情况下，属性列表都是空的。

除了Properties从Hashtable中继承下来的方法之外，Properties自己定义的方法列在表12-16中。Properties也包含了一个不被赞成使用的方法：save( )。因为它不能正确地处理错误，所以被store( )方法所取代了。

表12-16 由Properties 定义的从以前版本遗留下来的方法

方法	描述
String getProperty(String key)	返回与key相关联的值。如果key既不在列表中，也不在默认属性列表中，则返回一个null对象

String getProperty(String key,String defaultProperty)	返回与key相关联的值。如果key既不在列表中，也不在默认属性列表中，则返回defaultProperty
void list(PrintStream streamOut)	将属性列表发送给与streamOut相链接的输出流
void list(PrintWriter streamOut)	将属性列表发送给与streamOut相链接的输出流
void load(InputStream streamIn) throws IOException	从与streamIn相链接的输入数据流输入一个属性列表Enumeration propertyNames()返回关键字的枚举，也包括那些在默认属性列表中找到的关键字
Object setProperty(String key,String value)	将value与key关联。返回与key关联的前一个值，如果不存在这样的关联，则返回null（为了保持一致性，在Java 2中新增加的）
void store(OutputStream streamOut,String description)	在写入由description指定的字符串之后，属性列表被写入与streamOut相链接的输出流（在Java 2中新增加的）

Properties 类的一个有用的功能是可以指定一个默认属性，如果没有值与特定的关键字相关联，则返回这个默认属性。例如，默认值可以与关键字一起在 getProperty( ) 方法中被指定——如 getProperty(“name”，“default value”)。如果“name”值没有找到，则返回“defaultvalue”。当构造一个 Properties 对象时，可以传递 Properties 的另一个实例做为新实例的默认值。在这种情况下，如果对一个给定的 Properties 对象调用 getProperty(“foo”)，而“foo”并不存在时，Java 在默认 Properties 对象中寻找“foo”。它允许默认属性的任意层嵌套。

下面的范例说明了 Properties 的使用。该程序创建一个属性列表，在其中关键字是各国的名称，值是这些国家的首都。注意试图寻找包括默认值的美国的首都时的情况。

范例：PropertiesDemo.java

```
01  import java.util.*;
02  public class PropertiesDemo
03  {
04      public static void main(String args[])
```

```
05  {
06              Properties capitals = new Properties();
07              Set states;
08              String str;
09              capitals.put("中国", "北京");
10              capitals.put("俄罗斯", "莫斯科");
11              capitals.put("日本", "东京");
12              capitals.put("法国", "巴黎");
13              capitals.put("英国", "伦敦");
14
15              // 返回包含映射中项的集合
16              states = capitals.keySet();
17              Iterator itr = states.iterator();
18              while (itr.hasNext())
19              {
20                    str = (String) itr.next();
21          System.out.println("国家：" + str + " ，首都： "+ capitals.getProperty(str) + ".");
22              }
23              System.out.println();
24              // 查找列表，如果没有则显示为“没发现”
25              str = capitals.getProperty("美国", "没有发现");
26              System.out.println("美国的首都：" + str + ".");
27          }
28  }
```

输出结果：

```
国家：法国 ，首都： 巴黎.
国家：俄罗斯 ，首都： 莫斯科.
国家：英国 ，首都： 伦敦.
国家：中国 ，首都： 北京.
国家：日本 ，首都： 东京.

美国的首都：没有发现.
```

由于美国不在列表中，所以使用了默认值。尽管当调用 getProperty( )方法时，使用默认值是十分有效的，正如上面的程序所展示的那样，但对大多数属性列表的应用

来说，有更好的方法去处理默认值。为了提高灵活性，当构造一个属性（Properties）对象时，指定一个默认的属性列表。如果在主列表中没有发现期望的关键字，将会搜索默认列表。

12.7.5.6 Properties 类中使用 store()和 load()方法

Properties 的一个最有用的方面是可以利用 store()和 load()方法方便地对包含在属性（Properties）对象中的信息进行存储或从盘中装入信息。在任何时候，都可以将一个属性（Properties）对象写入流或从流中将其读出。这使得属性列表特别方便实现简单的数据库。

范例：PropertiesFile.java

```
01  import java.io.* ;
02  import java.util.* ;
03  public class PropertiesFile
04  {
05      public static void main(String[] args)
06      {
07          Properties settings = new Properties();
08          try {
09              settings.load(new FileInputStream("c:\\count.txt"));
10          } catch (Exception e) {
11              settings.setProperty("count", new Integer(0).toString());
12          }
13          int c = Integer.parseInt(settings.getProperty("count")) + 1;
14          System.out.println("这是本程序第" + c + "次被使用");
15          settings.put("count", new Integer(c).toString());
16          try {
17      settings.store(new FileOutputStream("c:\\count.txt"), "PropertiesFile use it .");
18          } catch (Exception e) {
19              System.out.println(e.getMessage());
20          }
21      }
22  }
```

输出结果：

如图 12-3 所示：

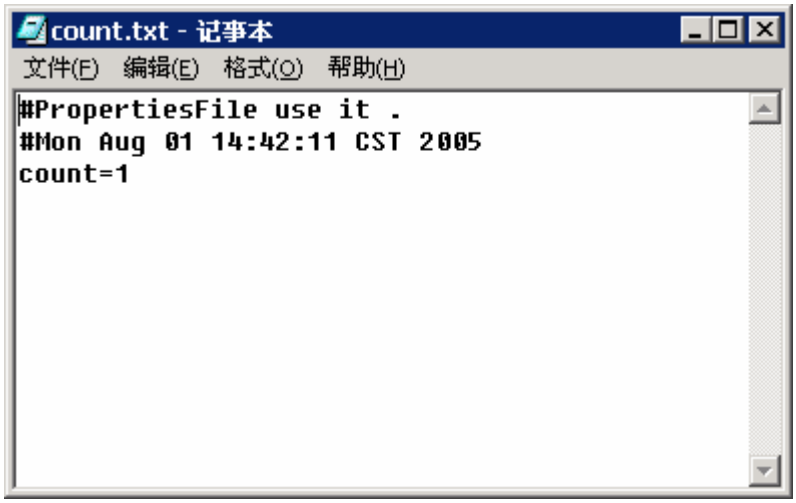


图 12-3　PropertiesFile 运行结果

程序每次启动时都去读取那个记录文件，直接取出文件中所记录的运行次数并加 1 后，又重新将新的运行次数存回文件。由于第一次运行时硬盘上还没有那个记录文件，程序去读取那个记录文件时会报出一个异常，就在处理异常的语句中将属性的值设置为 0，表示程序以前还没有被运行过。如果要用到 Properties 类的 store()方法进行存储，每个属性的关键字和值都必须是字符串类型的，所以上面的程序没有用从父类 HashTable 继承到的 put、get 方法进行属性的设置与读取，而是直接用了 Properties 类的 setProperty、getProperty 方法进行属性的设置与读取。

类集框架为程序员提供了一个功能强大的设计方案以解决编程过程中面临的大多数任务。下一次当开发者需要存储和检索信息时，可考虑使用类集。记住，类集不仅仅是专为那些“大型作业”，例如：联合数据库，邮件列表或产品清单系统等所专用的，它们对于一些小型作业也是很有效的。例如，TreeMap 可以给出一个很好的类集以保留一组文件的字典结构。TreeSet 在存储工程管理信息时是十分有用的。坦白地说，对于采用基于类集的解决方案而受益的问题种类只受限于开发者的想象力。

12.8 hashCode()方法

在前面已经学习过了 Object 类中的两个方法：equals()、toString()方法，实际上在改写 equals()方法时，也应该去考虑改写 Object 类中的 hashCode()方法，下面就先来看一下不改写 hashCode()方法时的情况：

范例：HashCodeDemo.java

```
01  import java.util.* ;
02  class Person
03  {
04      private String name ;
05      private int age ;
06      Person(String name,int age)
07      {
08          this.name = name ;
09          this.age = age ;
10      }
11      public String toString()
12      {
13          return "姓名："+this.name+"，年龄："+this.age ;
14      }
15  }
16  public class HashCodeDemo
17  {
18      public static void main(String args[])
19      {
20          HashMap hm = new HashMap() ;
21          hm.put(new Person("张三",20),"张三") ;
22          System.out.println(hm.get(new Person("张三",20))) ;
23      }
24  }
```

输出结果：

```
null
```

运行这个程序，读者应该觉得会输出“张三”，但它实际上返回了一个 null 值，其原因就是因为没有复写一个 hashCode()方法，读者可以将下面的代码，加入到程序之中：

```
// 注意：这里为了说明问题，只是返回了一个 true
public boolean equals(Object obj)
{
    return true ;
}
public int hashCode()
{
    return 20 ;
}
```

将上面代码加入到 HashCodeDemo.java 程序之中，运行之后可以发现，程序得到了预期的结果：“张三”，到这里，读者应该已经清楚了 hashCode()的含义，在用于存取散列表的时候使用，但是从上面修改后的程序也会有一个问题，就是所有的 Person 类的对象都拥有同一个散列码，这样在实际中是不可取的。而散列码的取得，也是根据实际的情况计算出的，换句话说，只要保证不同的对象有不同的散列码就可以。

12.9 对象克隆

“对象克隆（clone）”实际上是指将对象重新复制一份。在 Java 里提到 clone 技术，就不能不提 java.lang.Cloneable 接口和含有 clone 方法的 Object 类。所有具有 clone 功能的类都有一个特性，那就是它直接或间接地实现了 Cloneable 接口。否则，在尝试调用 clone()方法时，将会触发 CloneNotSupportedException 异常。

那么该如何实现对象的克隆呢？

1、 实现 Cloneable 接口

通过之前的介绍，读者应该知道一个类若要具备 clone 功能，就必须实现 Cloneable 接口。做到这一步，clone 功能已经基本实现了。Clone 功能对开发者来说，最主要的还是要能够使用它。那么如何才能使用 clone 功能呢？答案是覆盖 Object 类中的 clone()方法。

2、 覆盖 Object 类中的 clone()方法

为什么需要覆盖 Object 类中的 clone()方法？这里得再次从 jdk 源码说起。JDK

中 Object 类中 clone()方法的声明是：

protected native Object clone() throws CloneNotSupportedException;

是否注意到，这里 clone()方法修饰符是 protected，而不是 public。这种访问的不可见性使得用户对 clone()方法不可见。相信读者已明白为什么要覆盖 clone()方法了。而且，覆盖的方法的修饰符必须是 public，如果还保留为 protected，覆盖将变得没有实际意义。

范例：CloneDemo.java

```
01  class Employee implements Cloneable
02  {
03      private String name;
04      private int age;
05      public Employee(String name, int age)
06      {
07          this.name = name;
08          this.age = age;
09      }
10      public Object clone() throws CloneNotSupportedException
11      {
12          return super.clone();
13      }
14      public String toString()
15      {
16          return "姓名：" + this.name + "，年龄：" + this.age;
17      }
18      public int getAge()
19      {
20          return age;
21      }
22      public void setAge(int age)
23      {
24          this.age = age;
25      }
26      public String getName()
27      {
28          return name;
```

```
29          }
30          public void setName(String name)
31          {
32                  this.name = name;
33          }
34  }
35
36  public class CloneDemo
37  {
38          public static void main(String args[])
39          {
40                  Employee e1 = new Employee("张三", 21);
41                  Employee e2 = null;
42                  try {
43                          e2 = (Employee) e1.clone();
44                  } catch (CloneNotSupportedException e) {
45                          e.printStackTrace();
46                  }
47                  e2.setName("李四");
48                  e2.setAge(30);
49                  System.out.println("两个对象的内存地址比较：" + (e1 == e2));
50                  System.out.println(e1);
51                  System.out.println(e2);
52          }
53  }
```

输出结果：

两个对象的内存地址比较：false

姓名：张三，年龄：21

姓名：李四，年龄：30

• 本章摘要：

1、 String 类与 StringBuffer 类都是用于操作字符串的类，StringBuffer 类中的内容可以改变，而 String 类中的内容不可改变，StringBuffer 要比 String 类性能高。

2、 JAVA 中可用 ArrayList、Vector 实现动态对象数组的存取，唯一不同的是 ArrayList 是异步工作方式，而 Vector 是同步工作方式。

3、 JAVA 中可用 HashMap、Hashtable 实现二元偶元素的存储，且两个类都不直接支持 Iterator 输出，其中 HashMap 为异步工作方式、Hashtable 类为同步工作方式。

4、 JAVA 中可采用 Iterator 输出 ArrayList 或 Vector 类中的内容，其中 Vector 也支持 Enumeration 输出。

5、 一个类的对象要想实现克隆，必须实现 Cloneable 接口。

第十三章 Java 网络程序设计

13-1、Socket 介绍

Socket 是网络上运行的两个程序间双向通讯的一端，它既可以接受请求，也可以发送请求，利用它可以较为方便地编写网络上数据的传递。在 Java 中，有专门的 Socket 类来处理用户的请求和响应。利用 Socket 类的方法，就可以实现两台计算机之间的通讯。本章介绍在 Java 中如何利用 Socket 进行网络编程。

在 Java 中 Socket 可以理解为客户端或者服务器端的一个特殊的对象，这个对象有两个关键的方法，一个是 getInputStream()方法，另一个是 getOutputStream()方法。getInputStream()方法可以得到一个输入流，客户端的 Socket 对象上的 getInputStream()方法得到的输入流其实就是从服务器端发回的数据流。getOutputStream()方法得到一个输出流，客户端 Socket 对象上的 getOutputStream()方法返回的输出流就是将要发送到服务器端的数据流，（其实是一个缓冲区，暂时存储将要发送过去的数据）。

Sockets 有两种主要的操作方式：面向连接的和无连接的。面向连接的 sockets 操作就像一部电话，必须建立一个连接和一人呼叫。所有的事情在到达时的顺序与它们出发时的顺序时一样，无连接的 sockets 操作就像是一个邮件投递，没有什么保证，多个邮件可能在到达时的顺序与出发时的顺序不一样。

到底用哪种模式是由应用程序的需要决定的。如果可靠性更重要的话，用面向连接的操作会好一些。比如文件服务器需要数据的正确性和有序性，如果一些数据丢失了，系统的有效性将会失去。比如一些服务器间歇性地发送一些数据块，如果数据丢了的话，服务器并不想要再重新发过一次，因为当数据到达的时候，它可能已经过时了。确保数据的有序性和正确性需要额外的操作的内存消耗，额外的费用将会降低系统的回应速率。

无连接的操作使用数据报协议。一个数据报是一个独立的单元，它包含了所有的这次投递的信息。可以把它想象成一个信封，它有目的地址和要发送的内容。这个模式下的 socket 不需要连接一个目的的 socket，它只是简单地投出数据报.无连接的操作是快速的和高效的，但是数据安全性不佳。

面向连接的操作使用 TCP 协议。一个这个模式下的 socket 必须在发送数据之前与目的地的 socket 取得一个连接。一旦连接建立了，sockets 就可以使用一个流接口：打

开-读-写-关闭。所有的发送的信息都会在另一端以同样的顺序被接收。面向连接的操作比无连接的操作效率更低，但是数据的安全性更高。

13-2、Socket 程序

在 Java 中面向连接的类有两种形式，它们分别是客户端和服务器端。下面先讨论服务器端。

下面首先先建立服务器端程序，此服务器端程序只用于向客户端输出“hello world!”字符串。

范例：HelloServer.java

```
01  import java.io.*;
02  import java.net.*;
03  public class HelloServer
04  {
05      public static void main(String[] args) throws IOException
06      {
07          ServerSocket serversocket=null;
08          PrintWriter out=null;
09          try
10          {
11              // 在下面实例化了一个服务器端的 Socket 连接
12              serversocket=new ServerSocket(9999);
13          }
14          catch(IOException e)
15          {
16              System.err.println("Could not listen on port:9999.");
17              System.exit(1);
18          }
19          Socket clientsocket=null;
20          try
21          {
22              // accept()方法用来监听客户端的连接
23              clientsocket=serversocket.accept();
24          }
```

```
25              catch(IOException e)
26              {
27                  System.err.println("Accept failed.");
28                  System.exit(1);
29              }
30              out=new PrintWriter(clientsocket.getOutputStream(),true);
31              out.println("hello world!");
32              clientsocket.close();
33              serversocket.close();
34          }
35  }
```

运行结果：

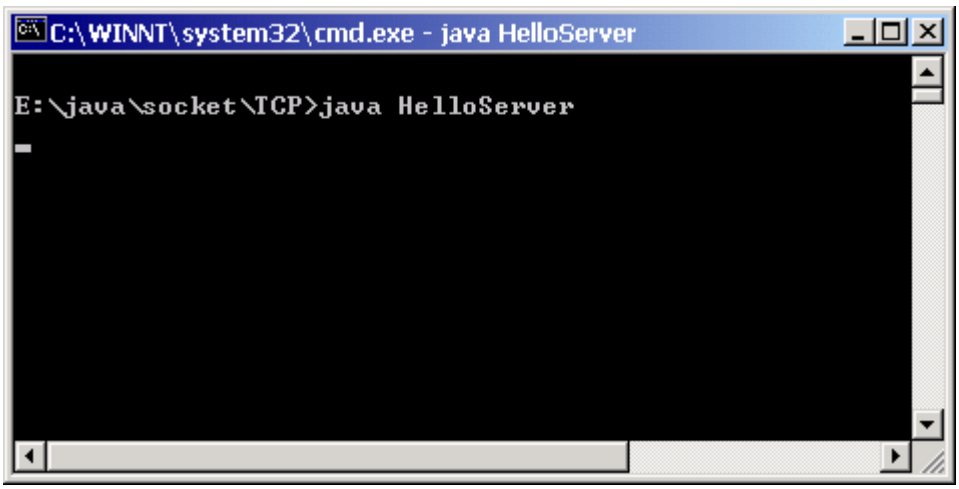


图 13-1　HelloServer 运行

程序说明：

1、 程序第 7 行声明了一个 ServerSocket 的对象。

2、 程序第 8 行声明了一个 PrintWriter 的对象，用于向可户端打印输出。

3、 程序第 9 行~第 18 行实例化 ServerSocket 对象，在 9999 端口进行监听。

4、 程序第 19 行声明一 Socket 对象 clientsocket，此对象用于接收客户端的 Socket 连接。

5、 程序第 20 行~29 行通过 ServerSocket 类中的 accept()方法，接收客户端的 Socket 请求，此方法返回一个客户端的 Socket 请求。

6、 程序第 30 行，通过客户端的 Socket 对象去实例化 PrintWriter 对象，此时 out 对象就具备了向客户端打印信息的能力。

7、 程序第 31 行调用 println()方法，将信息打印至客户端。

8、 第 32 行关闭客户端 Socket 连接。

9、 第 33 行关闭服务器端 Socket 连接。

由运行结果可以发现，执行 java HelloServer 之后，程序停在原处不动了，这就表示服务器在等待客户端的连接，下面为程序的客户端程序。

范例：HelloClient.java

```
01  import java.io.*;
02  import java.net.*;
03  public class HelloClient
04  {
05      public static void main(String[] args) throws IOException
06      {
07          Socket hellosocket=null;
08          BufferedReader in=null;
09          // 下面这段程序，用来将输入输出流与 socket 关联
10          try
11          {
12              hellosocket=new Socket("localhost",9999);
13      in=new BufferedReader(new InputStreamReader(hellosocket.getInputStream()));
14          }
15          catch(UnknownHostException e)
16          {
17              System.err.println("Don't know about host:localhost!");
18              System.exit(1);
19          }
20          catch(IOException e)
21          {
22              System.err.println("Couldn't get I/O for the connection.");
```

```
23                  System.exit(1);
24              }
25              System.out.println(in.readLine());
26              in.close();
27              hellosocket.close();
28          }
29  }
```

输出结果：

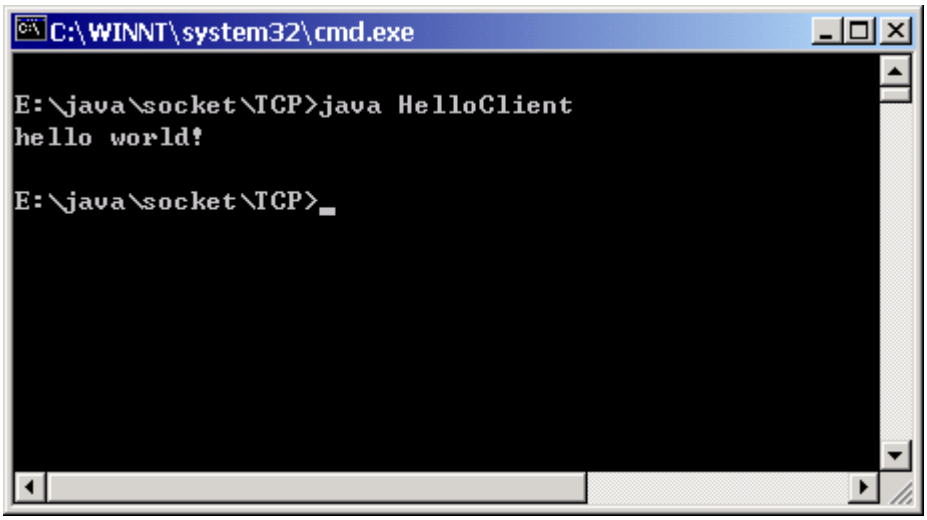


图 13-2　HelloClient 运行结果

程序说明：

1、 程序第 7 行声明一 Socket 的对象 hellosocket。

2、 程序第 8 行声明一 BufferedReader 的对象 in，此对象用于读取服务器端发送过来的信息。

3、 程序第 12 行实例化 hellosocket 对象，此连接在本机的 9999 端口上监听。

4、 程序第 13 行通过 hellosocket 对象实例化 BufferedReader 对象。

5、 程序第 25 行等待服务器端发送过来的信息并打印。

6、 程序第 26 行关闭 BufferedReader。

7、 程序第 27 行关闭 Socket 对象。

下面再来看一个 Socket 的经典范例 —— Echo 程序，读者可自行分析。

Echo 服务器端程序：

范例：EchoServer.java

```
01  import java.io.*;
02  import java.net.*;
03  public class EchoServer
04  {
05      public static void main(String[] args) throws IOException
06      {
07          ServerSocket serverSocket = null;
08          PrintWriter out = null;
09          BufferedReader in = null;
10          try
11          {
12              // 实例化监听端口
13              serverSocket = new ServerSocket(1111);
14          }
15          catch (IOException e)
16          {
17              System.err.println("Could not listen on port: 1111.");
18              System.exit(1);
19          }
20          Socket incoming = null;
21          while(true)
22          {
23              incoming = serverSocket.accept();
24              out = new PrintWriter(incoming.getOutputStream(), true);
25  // 先将字节流通过 InputStreamReader 转换为字符流，之后将字符流放入缓冲之中
26      in = new BufferedReader(new InputStreamReader(incoming.getInputStream()));
27              // 提示信息
28              out.println("Hello! . . . ");
29              out.println("Enter BYE to exit");
30              out.flush();
31              // 没有异常的情况不断循环
32              while(true)
```

```
33                  {
34                      // 只有当有用户输入的时候才返回数据
35                      String str = in.readLine();
36                      // 当用户连接断掉时会返回空值 null
37                      if(str == null)
38                      {
39                          // 退出循环
40                          break;
41                      }
42                      else
43                      {
44                          // 对用户输入字串加前缀 Echo:，将此信息打印到客户端
45                          out.println("Echo: "+str);
46                          out.flush();
47                          // 退出命令，equalsIgnoreCase()是不区分大小写的比较
48                          if(str.trim().equalsIgnoreCase("BYE"))
49                             break;
50                      }
51                  }
52                  // 收尾工作
53                  out.close();
54                  in.close();
55                  incoming.close();
56                  serverSocket.close();
57              }
58          }
59      }
```

Echo 客户端程序：

范例：EchoClient.java

```
01  import java.io.*;
02  import java.net.*;
03  // 客户端程序
04  public class EchoClient
05  {
06      public static void main(String[] args) throws IOException
```

```
07        {
08              Socket echoSocket = null;
09              PrintWriter out = null;
10              BufferedReader in = null;
11              try
12              {
13                    echoSocket = new Socket ( "localhost", 1111);
14                    out = new PrintWriter(echoSocket.getOutputStream(), true);
15        in = new BufferedReader(new InputStreamReader(echoSocket.getInputStream()));
16              }
17              catch (UnknownHostException e)
18              {
19                    System.err.println("Don't know about host: localhost.");
20                    System.exit(1);
21              }
22              System.out.println(in.readLine());
23              System.out.println(in.readLine());
24    BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in));
25              String userInput;
26    // 将客户端 Socket 输入流（既服务器端 Socket 的输出流）输出到标准输出上
27              while ((userInput = stdIn.readLine()) != null)
28              {
29                    out.println(userInput);
30                    System.out.println(in.readLine());
31              }
32              out.close();
33              in.close();
34              echoSocket.close();
35        }
36    }
```

输出结果：

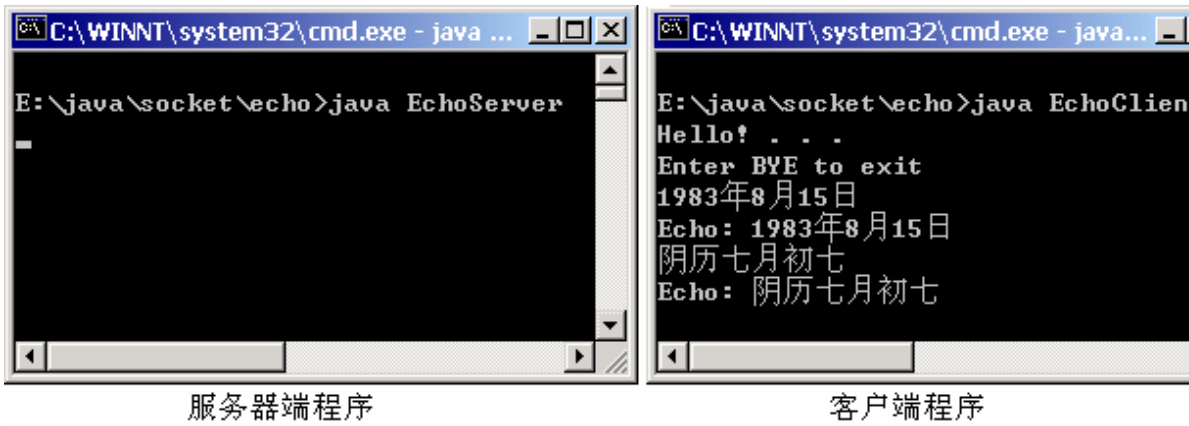


图 13-3　Echo 程序运行结果

运行上面的程序可以发现，无论客户端输入什么，服务器端都会对数据进行回应，输入"bye"之后程序会退出。

但读者可能会发现，此程序只能允许一个客户端进行操作，即：其它客户端程序无法再进行 Socket 连接，那该如何去解决这个问题呢？读者应该还记得之前讲解过的多线程的概念，只要在服务器端实现多线程，则服务器可以同时处理多个客户端请求，下面的代码是改进后的 EchoServer 程序。

范例：EchoMultiServerThread.java

```
01  import java.net.*;
02  import java.io.*;
03  public class EchoMultiServerThread extends Thread
04  {
05      private Socket socket = null;
06      public EchoMultiServerThread(Socket socket)
07      {
08          super("EchoMultiServerThread");
09          // 声明一个 socket 对象
10          this.socket = socket;
11      }
12      public void run()
13      {
```

```
14              try
15              {
16                  PrintWriter out = null;
17                  BufferedReader in = null;
18                  out = new PrintWriter(socket.getOutputStream(), true);
19  in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
20                  out.println("Hello! . . . ");
21                  out.println("Enter BYE to exit");
22                  out.flush();
23                  while(true)
24                  {
25                      String str = in.readLine();
26                      if(str == null)
27                      {
28                          break;
29                      }
30                      else
31                      {
32                          out.println("Echo: "+str);
33                          out.flush();
34                          if(str.trim().equalsIgnoreCase("BYE"))
35                              break;
36                      }
37                  }
38                  out.close();
39                  in.close();
40                  socket.close();
41              }
42              catch (IOException e)
43              {
44                  e.printStackTrace();
45              }
46          }
37  }
```

范例：EchoServerThread.java

```
01  import java.io.*;
02  import java.net.*;
03  // 多线程的服务器端程序
04  public class EchoServerThread
05  {
06      public static void main(String[] args) throws IOException
07      {
08          // 声明一个 serverSocket
09          ServerSocket serverSocket = null;
10          // 声明一个监听标识
11          boolean listening = true;
12          try
13          {
14              serverSocket = new ServerSocket(1111);
15          }
16          catch (IOException e)
17          {
18              System.err.println("Could not listen on port: 1111.");
19              System.exit(1);
20          }
21          // 如果处于监听态则开启一个线程
22          while(listening)
23          {
24              // 实例化一个服务端的 socket 与请求 socket 建立连接
25              new EchoMultiServerThread(serverSocket.accept()).start();
26          }
27          // 将 serverSocket 的关闭操作放在循环外，
28          // 只有当监听为 false 是，服务才关闭
29          serverSocket.close();
30      }
31  }
```

运行上面的服务器端程序之后，可以发现服务器可以同时处理多个客户端的 Socket 连接。

13-3、DatagramSocket 程序

因为使用流套接字的每个连接均要花费一定的时间，要减少这种开销，网络 API 提供了第二种套接字：自寻址套接字（datagram socket），自寻址使用 UDP 发送寻址信息（从客户程序到服务程序或从服务程序到客户程序），不同的是可以通过自寻址套接字发送多 IP 信息包，自寻址信息包含在自寻址包中，此外自寻址包又包含在 IP 包内，这就将寻址信息长度限制在 60000 字节内。图 13-4 显示了位于 IP 包内的自寻址包的自寻址信息。

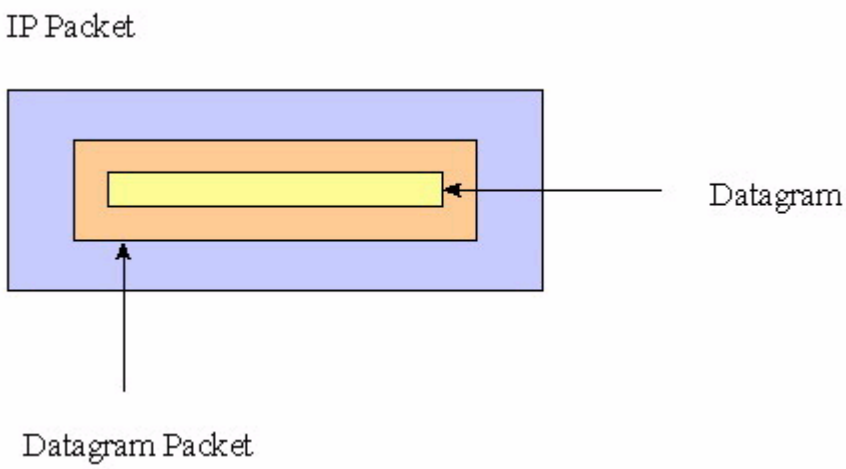


图 13-4

与 TCP 保证信息到达信息目的地的方式不同，UDP 提供了另外一种方法，如果自寻址信息包没有到达目的地，，那么 UDP 也不会请求发送者重新发送自寻址包，这是因为 UDP 在每一个自寻址包中包含了错误检测信息，在每个自寻址包到达目的地之后 UDP 只进行简单的错误检查，如果检测失败，UDP 将抛弃这个自寻址包，也不会从发送者那里重新请求替代者，这与通过邮局发送信件相似，发信人在发信之前不需要与收信人建立连接，同样也不能保证信件能到达收信人那里

自寻址套接字工作常用的类包括下面两个类：DatagramPacket 和 DatagramSocket。DatagramPacket 对象描绘了自寻址包的地址信息，DatagramSocket 表示客户程序和服务程序自寻址套接字，这两个类均位于 java.net 包内。

DatagramPacket 类

在使用自寻址包之前，需要首先熟悉 DatagramPacket 类，地址信息和自寻址包以字节数组的方式同时压缩入这个类创建的对象中。

DatagramPacket 有数个构造方法，即使这些构造方法的形式不同，但通常情况下他们都有两个共同的参数：byte [] buffer 和 int length，buffer 参数包含了一个对保存自寻址数据包信息的字节数组的引用，length 表示字节数组的长度。

最简单的构造方法是 DatagramPacket(byte [] buffer, int length)，这个构造方法确定了自寻址数据包数组和数组的长度，但没有任何自寻址数据包的地址和端口信息，这些信息可以通过调用方法 setAddress(InetAddress addr)和 setPort(int port)添加上。

DatagramSocket 类

DatagramSocket 类在客户端创建自寻址套接字与服务器端进行通信连接，并发送和接受自寻址套接字。虽然有多个构造方法可供选择，但发现创建客户端自寻址套接字最便利的选择是 DatagramSocket()方法，而服务器端则是 DatagramSocket(int port)方法，如果未能创建自寻址套接字或绑定自寻址套接字到本地端口，那么这两个方法都将抛出一个 SocketException 对象，一旦程序创建了 DatagramSocket 对象，那么程序分别调用 send(DatagramPacket dgp)和 receive(DatagramPacket dgp)来发送和接收自寻址数据包，

下面看一个简单范例：

范例：UdpReceive.java

```
01  import java.net.*;
02  import java.io.*;
03
04  public class UdpReceive
05  {
06      public static void main(String[] args)
07      {
08          DatagramSocket ds = null;
09          byte[] buf = new byte[1024];
10          DatagramPacket dp = null;
```

```
11              try {
12                  ds = new DatagramSocket(9000);
13              } catch (SocketException ex) {
14              }
15              // 创建 DatagramPacket 时，要求的数据格式是 byte 型数组
16              dp = new DatagramPacket(buf, 1024);
17              try {
18                  ds.receive(dp);
19              } catch (IOException ex1) {
20              }
21              /*
22               * 调用 public String(byte[] bytes,int offset,int length)构造方法，
23               * 将 byte 型的数组转换成字符串
24               */
25              String str = new String(dp.getData(), 0, dp.getLength()) + " from "
26                      + dp.getAddress().getHostAddress() + " : " + dp.getPort();
27              System.out.println(str);
28              ds.close();
29          }
30      }
```

范例：UdpSend.java

```
01  import java.net.*;
02  import java.io.*;
03  public class UdpSend
04  {
05      public static void main(String[] args)
06      {
07          // 要编写 UDP 网络程序，首先要用到 java.net.DatagramSocket 类
08          DatagramSocket ds = null;
09          DatagramPacket dp = null;
10          try {
11              ds = new DatagramSocket(3000);
12          }
13          catch (SocketException ex) {
14          }
15          String str = "hello world ";
```

```
16              try {
17                  dp = new DatagramPacket(str.getBytes(), str.length(), InetAddress
18                          .getByName("localhost"), 9000);
19          //调用 InetAddress.getByName()方法可以返回一个 InetAddress 类的实例对象
20              } catch (UnknownHostException ex1) {
21              }
22              try {
23                  ds.send(dp);
24              } catch (IOException ex2) {
25              }
26              ds.close();
27          }
28      }
```

输出结果：

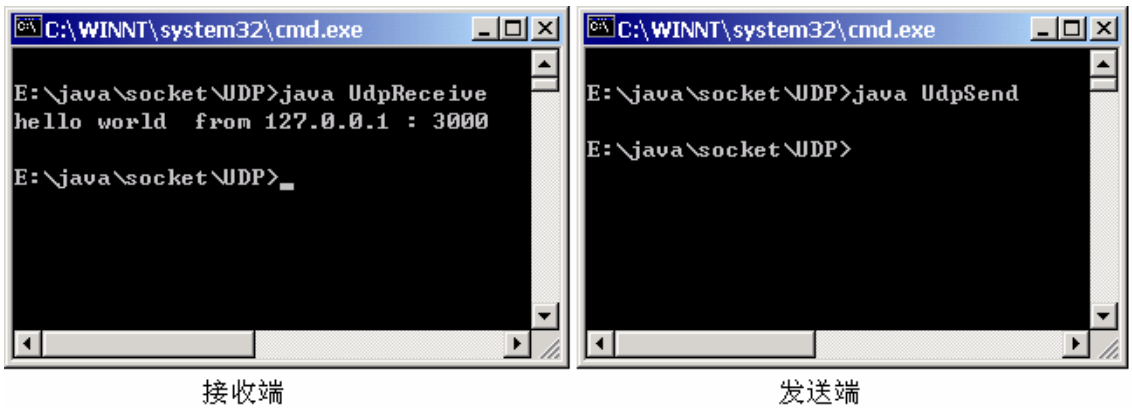


图 13-5　UDP 程序运行结果

UDP 数据的发送，道理类似于发送寻呼，发送者将数据发送出去就不管了，是不可靠的，有可能在发送的过程中发生数据丢失。就像寻呼机必须先处于开机接收状态才能接收寻呼一样的道理，这里要先运行 UDP 接收程序，再运行 UDP 发送程序，UDP 数据包的接收是过期作废的。因此，前面的接收程序要比发送程序早运行才行。

当 UDP 接收程序运行到 DatagramSocket.receive 方法接收数据时，如果还没有可以接收的数据，在正常情况下，receive 方法将阻塞，一直等到网络上有数据到来，receive 接收该数据并返回。

• 本章摘要：

1、 Java 提供了两种类型的网络程序实现：面向连接（TCP）、面向无连接（UDP）。

2、 TCP 程序的实现可用：Socket 类、ServerSocket 类。

3、 UDP 程序的实现可用：DatagramPacket 类、DatagramSocket 类。

附　　录

- Java 定义格式

附录　JAVA 定义格式

【 格式 3-1 数据类型的强制性转换语法 】

```
（欲转换的数据类型）变量名称;
```

【 格式 3-2 if 语句的格式 】

```
if (条件判断)
    语句 ;
```

【 格式 3-3 if 语句的格式 】

```
if (判断条件)
{
    语句 1 ;
    语句 2 ;
        …
    语句 3 ;
}
```

【 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;
}
```

【 格式 3-5 if…else 语句的格式 】

```
条件判断？表达式 1：表达式 2
```

【 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 ；
```

【 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 ；
}
```

【 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 ；
}
```

【 格式 3-9 while 循环语句】

```
while (判断条件)
{
     语句 1 ；
     语句 2 ；
     …
     语句 n ；
}
```

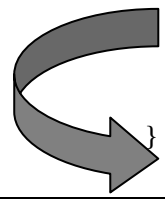
【 格式 3-10 do…while 循环语句】

```
do
{
      语句 1 ；
      语句 2 ；
        ….
      语句 n ；
}while (判断条件);
```

【 格式 3-11 for 循环语句】

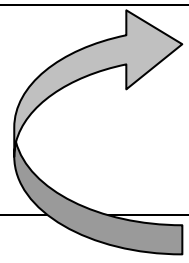
```
for (赋值处值；判断条件；赋值增减量)
{
      语句 1 ；
      ….
      语句 n ；
}
```

【 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，则此块内的语句不会被执行
      语句 n ；
}
      …
```

【 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，则此块内的语句不会被执行
                语句 n；
        }
                ….
```

【 格式 4-1 一维数组的声明与分配内存】

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

【 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]
```

【 格式 4-3 数组长度的取得】

```
数组名.length
```

【 格式 4-4 数组初值的赋值】

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

【 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] ;
```

【 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] ;
```

【 格式 4-7 二维数组初值的赋值格式】

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

【 格式 4-8 取得二维数组的行数与特定行的元素的个数】

```
数组名.length                  //取得数组的行数
数组名[行的索引].length     //取得特定行元素的个数
```

【 格式 4-9 声明方法，并定义其内容】

```
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

【 格式 5-1 类的定义】

```
class 类名称
{
    数据类型  属性 ；
    ….

    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 ；
    }
}
```

声明成员变量（属性）

定义方法的内容

【 格式 5-2 对象的产生】

```
类名 对象名 =  new 类名();
```

【 格式 5-3 访问对象中某个变量或方法】

访问对象：对象名称 . 属性名
访问方法：对象 . 方法名()

【 格式 5-4 封装类中的属性或方法】

封装属性：private 属性类型 属性名
封装方法：private 方法返回值 方法名称（参数）

【 格式 5-5 构造方法的定义方式】

```
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
        }
}
```

【 格式 5-6 定义内部类】

```
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
      {
           //内部类的成员
      }
}
```

外部类

内部类

【 格式 6-1 类的继承格式】

```
class 父类                        //定义类 CCircle
{
}
class 子类 extends 父类     //用 extends 关键字实现类的继承
{
}
```

【 格式 6-2 super 调用属性或方法】

```
super.父类中的属性 ；
super.父类中的方法() ;
```

【 格式 6-3 方法的复写】

```
class Super
{
```

```
    访问权限 方法返回值类型 方法 1（参数 1）
    {}
}
class Sub extends Super
{
    访问权限 方法返回值类型 方法 1（参数 1）——►  复写父类中的方法
    {}
}
```

【 格式 6-4 抽象类的定义格式】

```
abstract class 类名称   //定义抽象类
{
    声明数据成员 ;

    访问权限 返回值的数据类型 方法名称（参数...）
    {                                                    } 定义一般方法
        ...
    }
    abstract 返回值的数据类型 方法名称（参数...）;
      //定义抽象方法，在抽象方法里，没有定义处理的方式
}
```

【 格式 6-5 接口的定义格式】

```
interface 接口名称      //定义抽象类
{
     final 数据类型 成员名称 = 常量 ;        //数据成员必须赋值初值

     abstract 返回值的数据类型 方法名称（参数...）;
     //抽象方法，注意在抽象方法里，没有定义处理的方式。
}
```

【 格式 6-6 接口的实现】

```
class 类名称 implements 接口 A,接口 B     //接口的实现
{
   ...
}
```

【 格式 6-7 接口的扩展】

```
interface 子接口名称 extends 父接口 1，父接口 2，…
{
    ……
}
```

【 格式 7-1 异常处理的语法】

```
try                                          ┐
{                                            │
    要检查的程序语句 ;                        ├  try 语句块
      …                                      │
}                                            ┘
catch(异常类  对象名称)                       ┐
{                                            ├  catch 语句块
    异常发生时的处理语句 ;                    │
}                                            ┘
finally                                      ┐
{                                            ├  finally 语句块
    一定会运行到的程序代码 ;                  │
}                                            ┘
```

【 格式 7-2 抛出异常的语法】

```
throw 异常类实例对象 ;
```

【 格式 7-3 由方法抛出异常】

```
方法名称（参数…） throws 异常类 1，异常类 2，…
```

【 格式 7-4 编写自定义异常类的】

```
class 异常名称 extends Exception
{
   … …
}
```

【 格式 8-1 package 的声明】

```
package package 名称 ;
```

【 格式 8-2 package 的导入】

```
import package 名称.类名称 ;
```

【 格式 10-1 由键盘输入数据基本形式 】

```
import java.io.*;
public class class_name       //类名
{
    public static void main(String args[]) throws IOException
    {
        BufferedReader buf;    //声明 buf 为 BufferedReader 类的变量
        String str;            //声明 str 为 String 类型的变量
        ... ...

        buf=new BufferedReader(new InputStreamReader(System.in));    //产生
buf 对象
        str=buf.readLine();      //读入字符串至 buf
        ... ...
    }
}
```