

# 第1章 Java 开发前奏

Java 是一种简单易用、完全面向对象、具有平台无关性、且安全可靠的主要面向 Internet 的开发工具。自从 1995 年正式问世以来，Java 的快速发展已经让整个 Web 世界发生了翻天覆地的变化。随着 Java Servlet 的推出，Java 在电子商务方面开始崭露头角，最新的 Java Server Page(JSP)技术的推出，更是让 Java 成为了基于 Web 应用程序的首选开发工具。

Java 是第一套允许使用者将应用程序通过 Internet 从远端服务器传输到本地机上并执行的一种语言；是一种应用程序提供者不需要知道使用者的计算机硬件与软件环境的语言。比尔·盖茨曾经说过：“Java 是最卓越的程序设计语言”。

Java 自问世以来，技术和应用发展非常快，在计算机、移动电话、家用电器等领域中无一没有 Java 技术的存在。在 Internet 上，几乎每个网页都包含 Java 程序或代码。由于 Java 的广泛应用，使它受到了史无前例的关注，Java 是一种 Internet 编程语言，Java 还曾是网络动画技术的代名词，Java 虚拟机更是让人耳目一新，Java 简单易学、跨平台而又不失强大功能。

相比其他语言，Java 技术平台具有鲜明的优越性。从最初建造 Java 平台开始，就考虑了安全性的关系，即其安全性是建立在 Java 平台的内核中的。其他的语言只是在软件开发时才由用户自行处理其安全问题，难免会有安全漏洞。其次，对于程序员来说，Java 语言比以往其他的任何一门语言都好用，原因在于：Java 有自动垃圾回收的功能，Java 增加了对象和变量的强制类型检查，Java 还取消了指针。

目前，Java 技术的架构包括三个方面：

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

J2EE 目前已经成为开发商创建电子商务应用的事实标准。

J2SE 是 Java 2 平台的标准版，它适用于桌面系统，提供 CORBA 标准的 ORB 技术，结合 Java 的 RMI 支持分布式互操作环境。

J2ME 提供了 HTTP 高级 Internet 协议，使移动电话能以 Client/Server 方式直接访问 Internet 的全部信息，不同的 Client 访问不同的文件，此外还能访问本地存储区，提供最高效率的无线交流。

Java 技术又有其广泛的内涵：

首先，它是一门编程语言，Java 能够创建所有其它传统语言能编写的应用程序，Java 能够编写独立的应用程序，运行在装有 Java 虚拟机的操作系统上。Java 编写的程序经常用在 WWW 环境中(比如 applet, servlet 等)，Applet 在浏览器中执行不需要操作系统 JVM 的支持。

其次，Java 还是一个开发环境，Java 技术包含一系列的工具：编译器、解释器、文档生成工具、打包工具等等。另外由 Java 2 SDK 提供的 JRE (Java Runtime Environment) 还包括了一个完整的 Java 的类集合，比如基本语言类，GUI 控件类等。

最后，Java 也是一个运行环境，我们可以从 Sun 公司提供的 Java2 SDK 中来搭建运行环境，而这一切都是免费的。

时至今日，Java 仍然一步步地朝着远大的梦想而迈进，显而易见的成果便是在国际互联网中的应用，如今常见的浏览器也都全面支持 Java Applet 以及衍生出来的 JavaScript 语言等，而市面上可见的操作系统，也利用 Java 的跨平台特性来开发，包括 Windows 系列、各类 UNIX, Linux, Mac 等操作系统，都可以看到 Java 的踪迹。

这一切都充分说明了 Java 在计算机语言大家庭中的地位。Java 对 Internet 编程的影响就如同 C 语言对系统编程的影响一样。

## 1.1 Java 虚拟机及 Java 的跨平台原理

Java 虚拟机 (JVM) 是可运行 Java 字节码的假想计算机，Java 的跨平台性是相对于其他编程语言而言的，我们这里就用 C 语言的编程与执行过程来同 Java 的编程与执行过程进行对比说明。

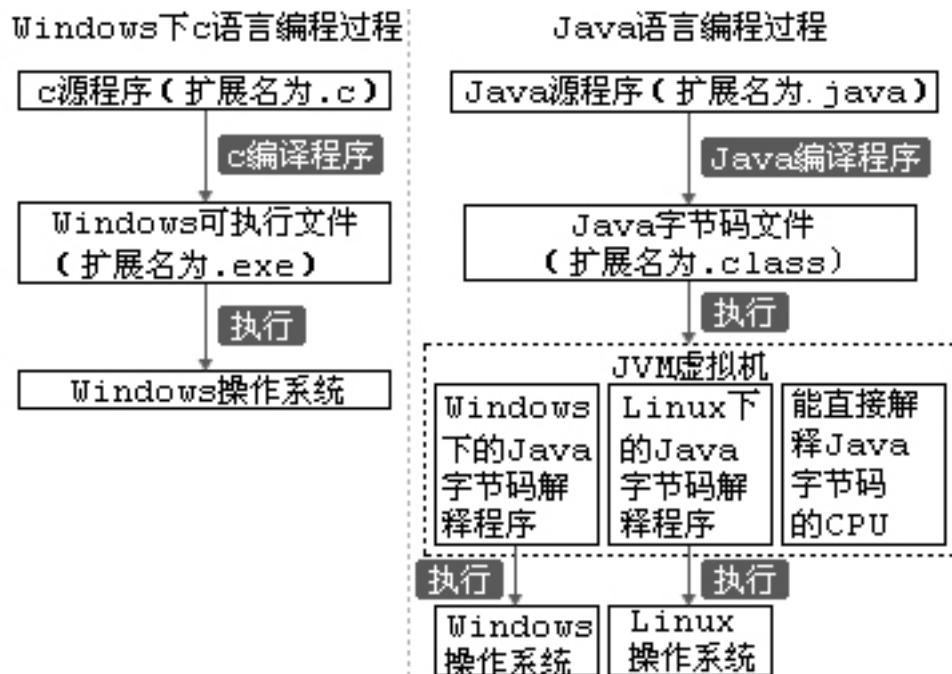


图 1.1

同 C 语言编程一样，我们首先也要编写 Java 源程序，然后由 Java 编译程序将 Java 源程序编译为 JVM 可执行的代码，即 Java 字节码。Java 源程序编译后生成的字节码文件就相当于 C 源程序编译后的 Windows 上的 exe 可执行文件，JVM 虚拟机的作用类似 Windows 操作系统。在 Windows 上运行的是 exe 文件，在 JVM 上运行的是 Java 字节码文件，也就是扩展名为 .class 的文件。

Windows 执行 exe 可执行文件的过程，就是从 exe 文件中取出一条条的计算机指令，交给 CPU 去解释执行。JVM 执行 Java 字节码文件的过程，也是 JVM 虚拟机从 Java 字节码文件中取出一条条的字节码指令交给“CPU”去执行。硬件与软件是可以相互转化的，我们可以用视霸卡硬件解码 VCD 数据，也可以用超级解霸软件解码 VCD 数据。执行 Java 字节码的“CPU”可以是硬件，也可以是某个系统上运行的一个软件，这个软件称为 Java 字节码解释程序（也就是 Java 虚拟机）。

可见，只要实现了特定平台下的解释器程序，Java 字节码就能通过解释器程序在该平台下运行，这是 Java 跨平台的根本。当前，并不是在所有的平台下都有相应的 Java 解释器程序，这也是 Java 并不是在所有的平台下都能运行的原因，它只能在已实现了 Java 解释器程序的平台下运行。

顺便来一句：Java 兼顾解释性与编译性语言的特点，.java 源文件转换成.class 字节码文件的过程是编译型的，.class 在操作系统上运行的过程则是解释型的，Java 虚拟机充当了解释器的作用。关于解释型和编译型的区别就不在这里详细叙述了，不能理解的读者可以自己去查阅相关资料，这并不影响我们学习 Java 的效果。

## 1.2 Java 的开发环境的搭建

Sun 公司提供了自己的一套 Java 开发环境，通常称之为 JDK(Java Development Kit)。Sun 公司提供了多种操作系统下的 JDK，随着时间的推移和技术的进步，JDK 的版本也在不断地升级，如 JDK1.2，JDK1.3，JDK1.4。各种操作系统下的 JDK 的各种版本在使用上基本相似，读者可以根据自己的环境，从 Sun 公司的网站 <http://java.sun.com> 上下载相应的 JDK 版本，一般情况下是越新越好。本书的讲解都是基于 Windows 平台下的 JDK1.4，Windows 下的 JDK 安装过程非常简单，这里就不再多说，这个工具包中的内容都放在 JDK 安装目录下（下面的讲解都假设我们的安装目录为 c:\j2sdk1.4.0），其中的 bin 子目录中包含了所有相关的可执行文件。如图 1.2 所示：

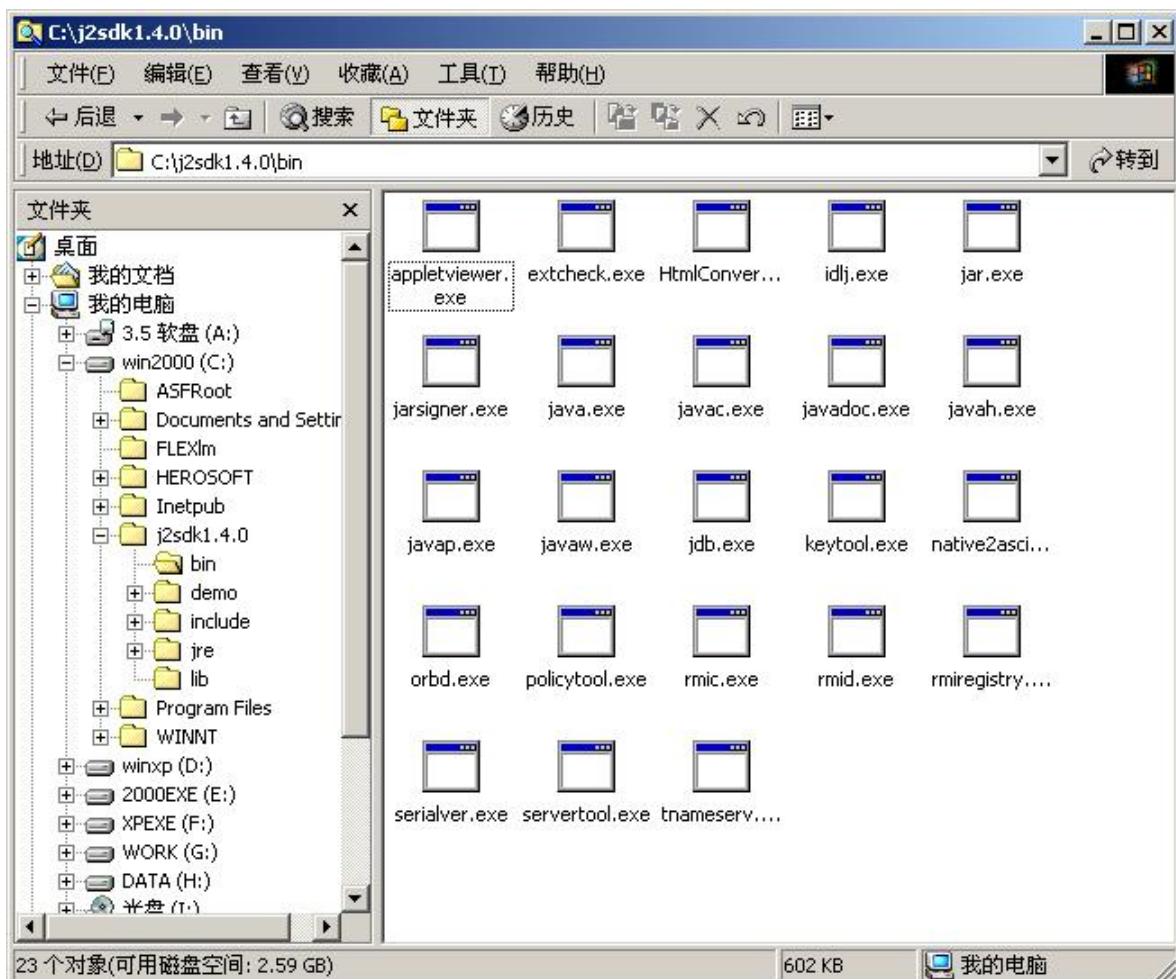


图 1.2

下面是 bin 目录中的常用命令的简要介绍（读者暂且不要关心 JDK 下的其他目录，在以后的章节中，我们会逐步涉及，到时候理解起来也要比现在容易得多）。

**javac.exe** 是 Java 源文件的编译工具，Java 源文件的扩展名为.java，如 Test.java，Java 源文件被编译后的 Java 字节码文件的扩展名为.class，如 Test.class。

**java.exe** 是 Java 字节码解释程序，负责解释执行 Java 字节码文件，就是一个 JVM。

在命令行窗口下，执行 Java 命令，如果屏幕上能够打出关于这个命令的用法介绍，如图 1.3 所示：

```
C:\>java
Usage: java [-options] class [args...]
              <to execute a class>
      or  java -jar [-options] jarfile [args...]
              <to execute a jar file>

where options include:
  -client          to select the "client" VM
  -server          to select the "server" VM
  -hotspot         is a synonym for the "client" VM [deprecated]
                  The default VM is client.

  -cp <classpath> <directories and zip/jar files separated by ;>
                  set search path for application classes and resources
  -D<name>=<value>
                  set a system property
  -verbose[:class|gc|jni]
                  enable verbose output
  -version
                  print product version and exit
```

图 1.3

这时你的 JDK 基本上就可以使用了。也许你会碰到两种错误：

第一种错误，说 Java 不是一个可运行的程序，如图 1.4 所示：

```
C:\>java
'java' 不是内部或外部命令，也不是可运行的程序
或批处理文件。
```

图 1.4

第二种可能碰到的错误是，当我们执行 Java 命令时，屏幕上不是打出关于这个命令的用法介绍，而是类似图 1.5 所示的错误信息：

```
C:\>C:\WINNT\System32\cmd.exe
C:\>java
Error: could not open 'C:\JBuilder8\jdk1.4\jre\lib\i386\jvm.cfg'

C:\>
```

图 1.5

下面的小节我们就来分析和解决这两种错误：

## 1.2.1 环境变量的介绍

太多的现代人都不会 DOS 了，根本就不明白系统环境变量和 path 环境变量的作用。通俗的说，系统环境变量就是在操作系统中定义的变量，可供操作系统上的所有应用程序使用。

## 1.2.2 如何查看系统环境变量

以 Windows2000 为例（由于 Windows2000 具有众多的新特性，这些特性在 Java 的开发过程中，能为我们提供许多方便快捷的功能，所以在这里建议读者使用 Windows2000 来做系统平台）。

首先右键单击桌面上的“我的电脑”，从下拉菜单中选择“属性”，在出现的属性面板中选择“高级”标签，如图 1.6 所示：

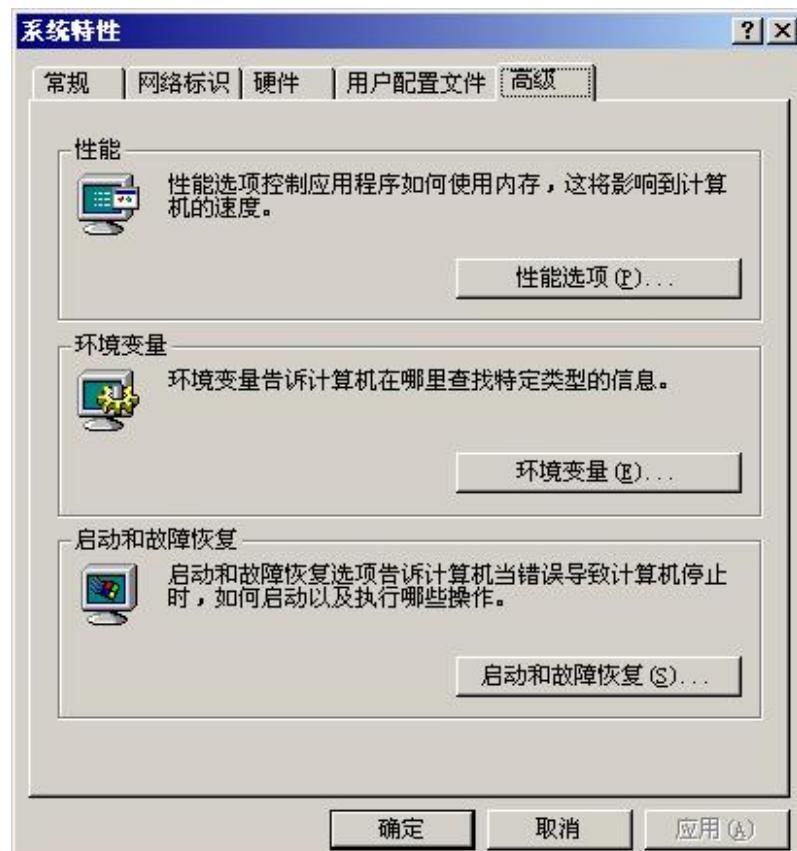


图 1.6

然后点击“环境变量”打开环境变量面板，在这里可以看到上下两个窗口，上面窗口名为“某用户的环境变量”（在这里是 Administrator 即管理员的用户变量），下面窗口名为“系统变量”，如图 1.7 所示：



图 1.7

可以在其中任意一个窗口里进行设置，区别在于上面窗口的设置用于个人环境变量，只有以该用户身份登录系统时才有效，而下面窗口中的设置则对所有用户都有效。

我们也可以启动一个命令行窗口（点击“开始”菜单，在“运行”中输入 cmd，然后按下“Enter”键即可。或者点击“开始”菜单，在“程序”中选择“附件”，然后运行“命令提示符”，这两种启动方式的效果是一样的），在这个命令行窗口中执行 set 命令，如下图 1.8 所示：

```
命令提示符
LOGONSERVER=\\PC-CHQ
NUMBER_OF_PROCESSORS=1
OS=Windows_NT
Os2LibPath=C:\\WINNT\\system32\\os2\\dll;
Path=c:\\winnt\\system32;c:\\winnt;c:\\winnt\\system32\\Wbem;c:\\j2sdk1.4
PATHEXT=.COM;.EXE;.BAT;.CMD;.UFS;.UBE;.JS;.JSE;.WSF;.WSH
PROCESSOR_ARCHITECTURE=x86
PROCESSOR_IDENTIFIER=x86 Family 15 Model 1 Stepping 3, GenuineInte
PROCESSOR_LEVEL=15
PROCESSOR_REVISION=0103
ProgramFiles=C:\\Program Files
PROMPT=$P$G
SystemDrive=C:
SystemRoot=C:\\WINNT
TEMP=C:\\DOCUMENTS\\ADMINISTRATOR\\LOCALS\\Temp
TMP=C:\\DOCUMENTS\\ADMINISTRATOR\\LOCALS\\Temp
```

图 1.8

如上图所示，在命令行窗口中设置的 path 变量的值是图 1.7 所示的用户变量和系统变量窗口中的 path 变量的值的总和。也就是说，用 set 命令看到的环境变量值是 Windows 环境变量窗口中用户环境变量和系统环境变量的值的总和。作者在教学的过程中，有时会看到另外一种情况：命令行窗口中用 set 命令看到的环境变量的值不是如先前 Windows 窗口中图 1.7 所示的用户变量和系统变量的值的总和，而是只有用户变量窗口中的值。至于为什么会有这种情况，只要不影响我们的使用，我们就不必再去管它了。

实际上，真正起作用的就是我们在命令行窗口中运行 set 命令所看到的所有变量和值，所以建议读者每次配置完成后，用 set 命令查看一下。你就能发现，用户环境变量修改之后，需要用户重新登录才会生效，相信聪明的读者应该能体会到其中的道理，也知道如何解决环境变量所引起的问题了。

事实上，有经验的程序员在遇到环境变量的问题时，都是通过这样的途径和手段解决问题的。我们接着就可以进行系统环境变量的配置了。

### 1.2.3 如何设置系统环境变量

还是以 path 环境变量为例进行讲解吧，path 环境变量的作用是设置供操作系统去寻找和执行应用程序的路径，也就是说，如果操作系统在当前目录下没有找到我们想要执行的程序和命令时，操作系统就会按照 path 环境变量指定的目录依次去查找，以最先找到的为准。path 环境变量可以存放多个路径，路径和路径之间用分号(;)隔开。在其他的操作系统下可能是用其他的符号分隔，比如在 Linux 下就是用冒号(:)。

我们可以在 Windows 系统环境变量窗口中设置系统变量。在图 1.7 窗口中，单击名为“path”的变量（如果没有你想设置的环境变量选项，在“用户变量”或“系统变量”中选择“新建”来添加），选择“编辑”。然后如下图所示：在打开的“编辑系统变量”窗口中的“变量值”输入框中加入你想设置的环境变量值。



图 1.9

对于 path，我们可以在原有值的基础上添加新的路径，因为我们想在任意路径下运行 java.exe, javac.exe 等程序，所以我们应当在 path 原有值的末尾加上分号(;)，然后再加上你的 Java 编译器所在的路径（这里是 c:\j2sdk1.4.0\bin），最后点击“确定”按钮，这样设置就完成了。接着，我们重新启动一个新的命令行窗口，执行 set 命令，查看我们刚才的设置结果。这种方法的优点是设置一次之后，系统会保存此设置，对以后在当前操作系统上运行的任何程序都有效，但不会影响先前已经运行起来的程序，特别是命令行窗口程序（如果想要使设置的值生效，只能关闭原来的命令行窗口，再重新启动一个新的命令行窗口程序）。

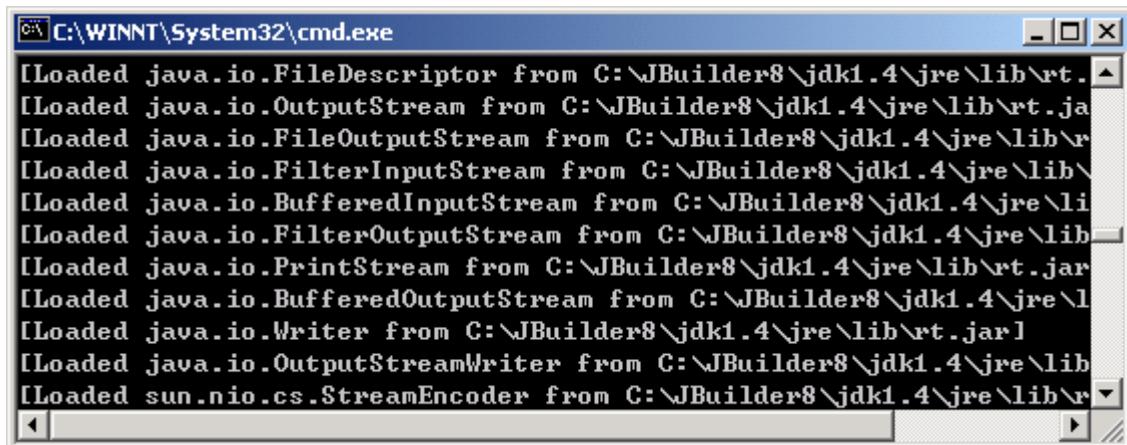
我们也可以直接在命令行窗口下进行设置，针对上面的情况，可以在命令行窗口中执行 set path=c:\j2sdk1.4.0\bin;%path%;，这样在这个命令行窗口中的任意路径下都可以执行 Java.exe 程序了。讲到这里肯定会有读者提出疑问了，%path%究竟起什么作用呢？将某个环境变量包含在一对百分号(%)之间，就表示该环境变量的当前取值。如图 1.8 中，“C:\WINNT\system32;C:\WINNT;C:\WINNT\System32\Wbem;c:\j2sdk1.4.0\bin;”这个长长的字符串就代表了%path%。务

必注意，在命令行窗口下的设置只对当前这个窗口有效，并不会影响到其他命令行窗口和程序。

解决了第一种错误，我们再来看看如图 1.5 所示的第二种错误。产生这种错误的原因可能是由于在设置 path 时，忽视了 path 环境变量中的目录的设置顺序，如我们使用 `set path=%path%;c:\j2sdk1.4.0\bin` 命令设置 path 环境变量，由于 `c:\winnt\system32` 这个路径排在 `c:\j2sdk1.4.0\bin` 前面，而在 `c:\winnt\system32` 下也有一个 `java.exe` 程序。这样，当我们运行 Java 命令时，执行的不是 `c:\j2sdk1.4.0\bin\java.exe`，而是 `c:\winnt\system32\java.exe`，恰恰 `c:\winnt\system32\java.exe` 又是一个有问题的程序。在 `c:\winnt\system32` 目录下，怎么会出现一个有问题的 `java.exe` 呢？造成这种错误的原因有多种，譬如，我们安装完 JBuilder8 这类 Java 开发工具软件时，JBuilder8 安装程序会将自己的 `java.exe` 拷贝到 `c:\winnt\system32` 目录下。`java.exe` 启动需要到原始安装目录中去装载许多相关文件，如果我们以后又删除或移动了 JBuilder8 的安装目录，就会出现这样的问题。解决这个问题，我们只要将我们想用的 `java.exe` 所在的目录放在 path 环境变量的前面，如：`set path=c:\j2sdk1.4.0\bin;%path%`。有时，我们的计算机上安装了多个 Java 开发工具，我们怎么知道在命令行窗口中执行的 `java` 命令属于哪个开发工具包中的呢？其实，我们只要在运行 `java.exe` 的时候加上 `verbose` 参数，格式如下：

```
java -verbose
```

java 虚拟机启动时，就会显示其详细的加载过程信息，如图 1.10 所示：



```
[Loaded java.io.FileDescriptor from C:\JBuilder8\jdk1.4\jre\lib\rt.jar]
[Loaded java.io.OutputStream from C:\JBuilder8\jdk1.4\jre\lib\rt.jar]
[Loaded java.io.FileOutputStream from C:\JBuilder8\jdk1.4\jre\lib\rt.jar]
[Loaded java.io.FilterInputStream from C:\JBuilder8\jdk1.4\jre\lib\rt.jar]
[Loaded java.io.BufferedInputStream from C:\JBuilder8\jdk1.4\jre\lib\rt.jar]
[Loaded java.io.FilterOutputStream from C:\JBuilder8\jdk1.4\jre\lib\rt.jar]
[Loaded java.io.PrintStream from C:\JBuilder8\jdk1.4\jre\lib\rt.jar]
[Loaded java.io.BufferedOutputStream from C:\JBuilder8\jdk1.4\jre\lib\rt.jar]
[Loaded java.io.Writer from C:\JBuilder8\jdk1.4\jre\lib\rt.jar]
[Loaded java.io.OutputStreamWriter from C:\JBuilder8\jdk1.4\jre\lib\rt.jar]
[Loaded sun.nio.cs.StreamEncoder from C:\JBuilder8\jdk1.4\jre\lib\rt.jar]
```

图 1.10

从上面显示的信息中，我们就能看出所运行的 Java 命令是属于哪个开发工具包。

作为初学者，你不一定马上会碰到上面的问题，但你一旦变成了 Java 老手，反而有可能会碰到这些问题，因为老手使用 Java 的频率更高了，遇到的环境更复杂了，碰到问题的概率也就大多了。尽管传染病的医生防范传染病的经验很丰富，但他接触传染源的机会也越多，反而比普通医生被传染的可能性大多了。这正是“常在河边走，哪能不湿鞋”。别着急，熟能生巧、勤能补拙，编的程序多了，遇到的错误就多了，排除错误的能力也多了，你的经验也就更丰富了。

现在读者基本有了一个可实验的环境，我们就可以来体验一下 Java 的编程过程了。

### 1.3 体验 Java 编程的过程

首先用记事本程序建立一个名为 `Test.java` 的源文件（在实际操作中，我们常常会用到一些更好的工具软件，例如 UltraEdit、EditPlus 等，它们有很多记事本程序不能比拟的优点。比如：支持用不同的颜色标记关键字，类名；自动显示行号，以便于我们更加方便的查找所需要的代码；能够自动缩进，减少了书写程序代码的工作量；能够同时编辑多个文件，方便在多个文件之间反复切换；还可以正常显示 Linux 格式的文本文件），文件内容如下：

程序清单：`Test.java`

```

class Test
{
    public static void main(String [] args)
    {
        System.out.println("My first Java program");
    }
}

```

在编译和运行这个程序之前，我们必须对这个程序的内容作简要介绍：

1). java 中的程序必须以类(class)的形式存在，一个类要能被解释器直接启动运行，这个类中必须有 main 函数，java 虚拟机运行时首先调用这个类中的 main 函数，main 函数的写法是固定的，必须是 public static void main(String [] args)，等到大家学到后面的章节，就明白这个函数的各组成部分的具体意义了，由于以后的每个例子几乎都要用这个函数，读者现在先硬记下来再说。

2). 如果我们要让程序在屏幕上打印出一串字符信息（包括一个字符），我们可以用 System.out.println(“填写你要打印的若干字符” )语句，或是 System.out.print(“填写你要打印的若干字符” )语句。前者会在打印完的内容后多打印一个换行符(\n)，你的窗口光标的位置会移动到打印行的下一行的开始处。而后者只打印你的字符串，不增加换行符，你窗口的光标停留在所打印出的字符串的最后一个字符后面。println()等于 print("\n")。

3). 如果在 class 之前没有使用 public 修饰符，源文件的名可以是一切合法的名称。而带有 public 修饰符的类名必须与源文件名相同，如上面程序第一行改为下面的形式，源文件名必须是 Test.java，但与源文件名相同的类却不一定带有 public 修饰符。

```
public class Test
```

在命令行窗口中，用 cd 命令进入 Test.java 源文件所在的目录，运行 javac Test.java。命令执行完后，我们能看到该目录下多了一个 Test.class 文件，这就是编译后的 Java 字节码文件。

经常有初学者问我一个如下面这样的错误，如图 1.11 所示：

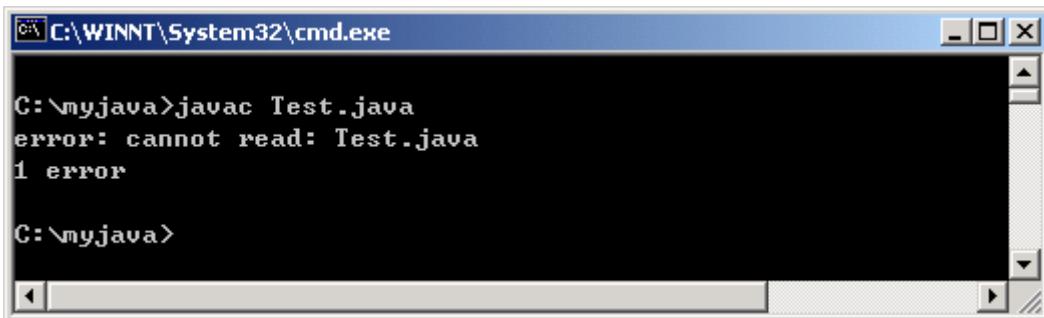


图 1.11

这是因为你可能在设置 Windows 的文件夹选项时，选中了“隐藏已知文件类型的扩展名”，如图 1.12 所示：

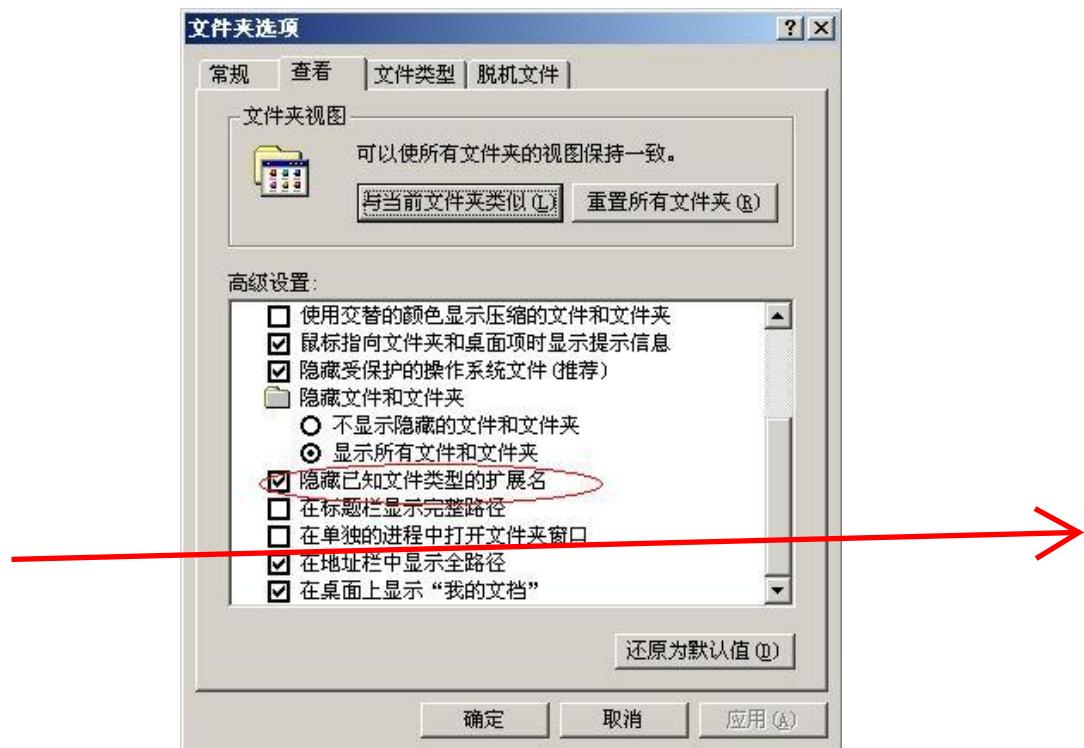


图 1.12

我们的源文件名实际上是 Test.java.txt，但因为系统隐藏了后面的 txt 扩展名，我们会误以为源文件名就是 Test.java。我曾经碰到过一个网站泡沫时代的“中级程序员”，因为一个类似的问题而花费了几天的时间也没有找到问题所在，后来我发现不少学员也被这个问题给绊倒了，所以拿出来说一下，建议读者取消“隐藏已知文件类型的扩展名”的设置。

我们接着运行 java Test 命令，屏幕上打出 My first Java program 这行文字，这样，我们编译运行第一个 Java 程序的过程就算完成了。

## M 脚下留心：

上面运行时用的是 `java Test`，而不是 `java Test.class`！运行时不要带上 `.class` 扩展名。如果我们的源文件名为 `Xxx.java`，文件中有多个类（class）定义，我们编译后的 `.class` 文件就有多个，能直接用 `java` 命令启动运行只有那个含有 `main` 函数的类，`java` 命令后跟的参数是类名，而不是文件名，尽管我们的文件名为 `Xxx.java`，但我们却不见得是以 `java Xxx` 的形式来运行我们的程序的，关键还是要看类名。

对 `java Test` 命令，从 Windows 的方面讲，Java 是一个命令，Test 是这个命令的参数。从 Java 方面来说，Java 命令是启动了一个虚拟机，Test 是这个虚拟机上能独立运行的一个 Java 类，而不是 `Test.class` 文件，关于类的知识，我们将在后面的章节进行详细讲解。

## 1.4 classpath 的设置

运气差的读者在执行 `java Test` 命令时，会碰到这样或那样的错误提示信息，运气好的读者也不要庆幸和掉以轻心，这对你来说，是迟早要遇到的问题。当 Java 虚拟机要装载某一个类时，它会去哪个目录下查找这个类呢？这里通过几个实验步骤来了解 `classpath` 的作用，假设我们的 `Test.class` 类位于 `c:\myjava` 目录中。

步骤 1，在命令行窗口下用 `cd` 命令进入一个除 `c:\myjava` 以外的目录，如 `c:\根目录`。运行 `java`

Test，结果屏幕上提示找不到 Test 这个类。

步骤 2，执行 set classpath=c:\myj ava，再运行 java Test，结果正常。

这个实验说明了 classpath 的作用和 path 环境变量的作用相似，Java 虚拟机按照 classpath 环境变量指定的目录顺序去查找这个类，以最先找到的为准。Java 虚拟机除了在 classpath 的环境变量指定的目录中查找要运行的类，会不会也在当前目录下查找呢？我们接着用几个实验来说明这个问题。

步骤 3，进入 c:\myj ava 目录（也就是 Test.class 所在的目录），执行：

```
set classpath=c:\mytmpdir
```

注意：这里指定的 c:\mytmpdir 可以用除了 c:\myj ava 之外的任意目录替代，再运行 java Test，屏幕上提示找不到 Test 这个类。这个结果似乎告诉我们，Java 虚拟机查找类的过程，同 Windows 查找可执行命令（.exe,.bat 或.cmd 文件以及.dll 动态连接库）的过程还是有点区别，不会在当前目录下查找，只找 classpath 指定的目录。熟悉 Linux 的读者，知道 Linux 命令也不会在当前目录下查找，只在 path 指定的目录中查找。

步骤 4，还是在 c:\myj ava 目录下，执行 set classpath=c:\mytmpdir;，注意最后多了个分号（;），或干脆执行 set classpath=，取消 classpath 环境变量的定义。再运行 java Test，结果又正常了。这又与我们刚才在第三步中得到的结论相矛盾了。

我们与其去反复思考为什么，还不如认为这是 Sun 公司提供的 JDK 的问题，像这些东西是没法死记硬背的，在不同的环境下可能会有不同的情况。读者只要明白了那些最根本的东西，再多动手实践，总结分析，就能够使这些问题迎刃而解。

步骤 5，既然 Java 总是查找 classpath 中所指定的路径，我们能否通过某种方式，让 Java 虚拟机在任何情况下都会去当前目录下查找要使用的类呢？我们只要在 classpath 环境变量中添加一个点（.），如“set classpath=c:\mydir.;”即可，这个点（.）就代表 Java 虚拟机运行时的当前工作目录。

作者不敢肯定 Sun 公司在以后的 JDK 版本中是否会注意并更正上面的那些小问题，给用户一个方便，但这对初学者来说，确实非常重要，要不然，连学习 Java 的第一步都无法通过，怎么可能还有信心继续学习下去呢？有好的产品，却不一定能够赢得天下，虽然 Java 是一门优秀的语言和技术，但从上面这些小的问题方面，我们不难想象 Sun 公司的业务不如 Microsoft 公司的原因，因为他们替一般用户想的不多。

## M 脚下留心：

在配置 Java 环境变量的时候经常会因为空格而导致错误，比如以下两种情况：

1. set classpath =c:\j2sdk1.4.0\bin;
2. set classpath= c:\j2sdk1.4.0\bin;

第一种错误：等号和 classpath 之间有空格。这样本来应该设置变量“classpath”的值，却被设置成了变量“classpath+空格”的值。

第二种错误：等号和路径名之间有空格。把需要设置的正确路径“c:\j2sdk1.4.0\bin;”替换成了“空格+c:\j2sdk1.4.0\bin;”的路径，导致了路径设置的错误。

还有一种情况：

```
set classpath= C:\Documents and Settings\Administrator\My Documents;
```

虽然在等号的两端都没有空格，但是在路径“Documents and Settings”中却出现了空格，这在 Windows 中是没有问题的，因为 Windows 允许有带空格的目录名，也确实有这个目录。但 Java 是不允许的，与 java 有关的环境变量对空格和中文是非常敏感的，比较忌讳，初学者往往会在这些方面被弄得莫名其妙，吃了不少苦头。

## & 多学两招:

### 1. 如何快速得到路径字符串

我们在设置 path 和 classpath 环境变量时，都要用到目录的完整路径，当目录的层次很深时，这个路径名将会非常之长，如果我们一个一个地键入其中的字母，会非常浪费时间，也很容易出错。如果能够使用 **ctrl+c**、**ctrl+v** 这样的拷贝、贴功能来完成这些任务就好了，作者平时用的两种方法可供读者借鉴。

第一种办法是，在你的 Windows 操纵系统的文件夹选项对话框中，选中“在地址栏中显示全路径”选项，如图 1.13 所示：

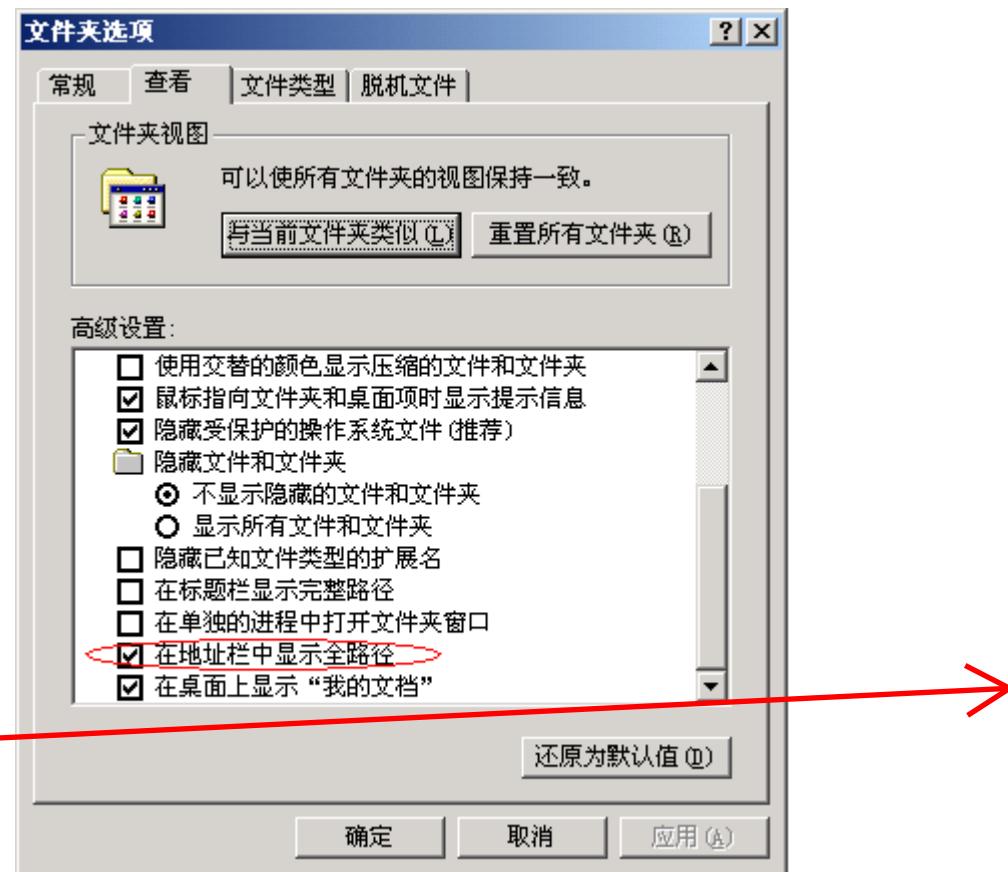


图 1.13

进行此项设置后，你在 Windows 资源管理器中进入的某个目录后，Windows 资源管理器的地址栏中会显示该目录的完整路径，如图 1.14 所示。

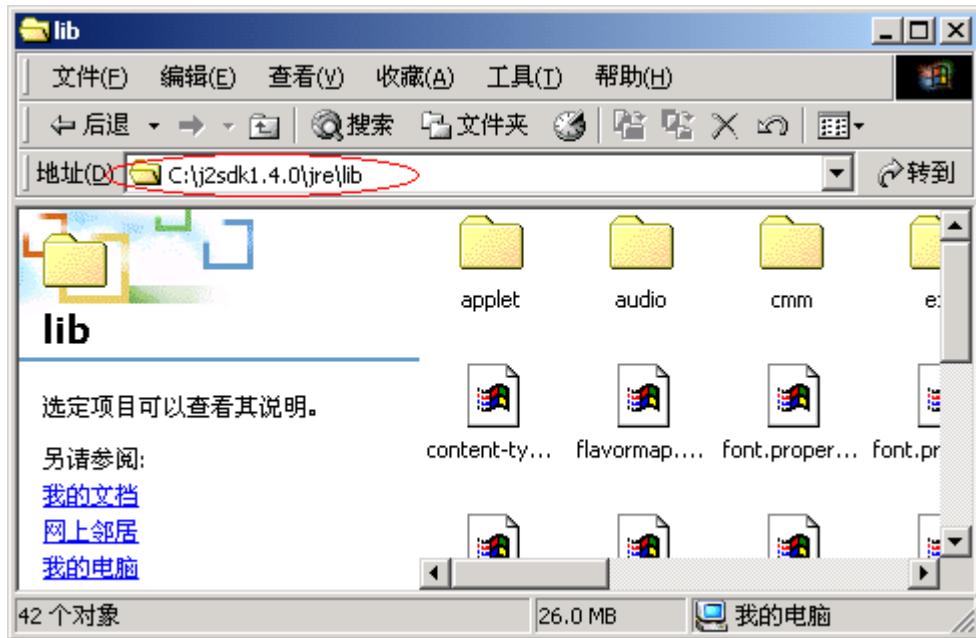


图 1.14

这样，你就可以在地址栏中用 **ctrl+c** 拷贝这个路径名了。

第二种办法是，在 Windows 资源管理器中选中想要的文件夹或文件，将它拖入运行对话框，如图 1.15 所示。

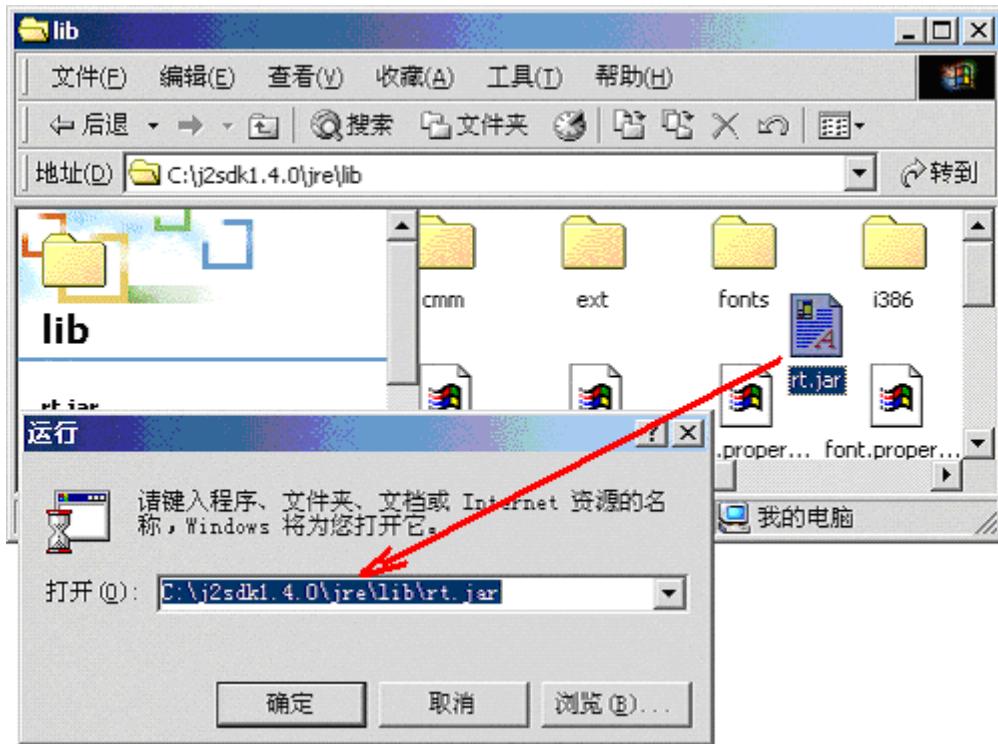


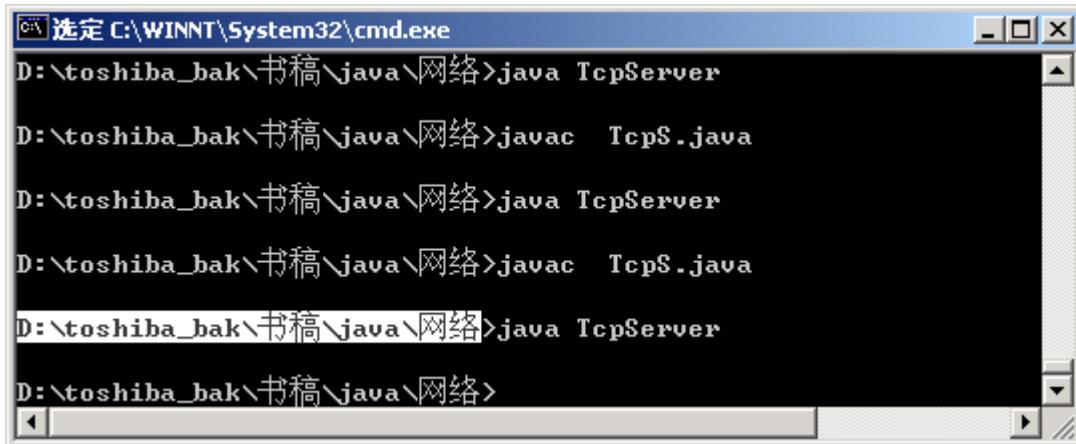
图 1.15

这样，我们就可以用 **ctrl+c** 拷贝运行对话框中的路径名了。作者最早帮助别人解决类似前面讲过的 Test.java 与 Test.java.txt 的问题，就是通过这种方式发现的。

## 2. 如何在命令行窗口中拷贝和粘贴

我们学会了拷贝路径的问题，但如果我们要在命令行中使用这个路径，能够使用

ctrl+v 进行粘贴吗？这是不行的，但我们可以命令行窗口中任意位置单击鼠标右键，先前拷贝到剪贴板上的内容便被粘贴在当前插入提示符指示的地方。如果我们想将命令行窗口中的文本拷贝到剪贴板，我们只要用鼠标选中要拷贝的区域，再单击鼠标右键即可。命令行窗口中拷贝和粘贴是 Windows 2000 中新加的功能（又是 Microsoft 从 Linux 那学来的哟！），所以作者建议读者使用 Windows 2000 来学习 Java。



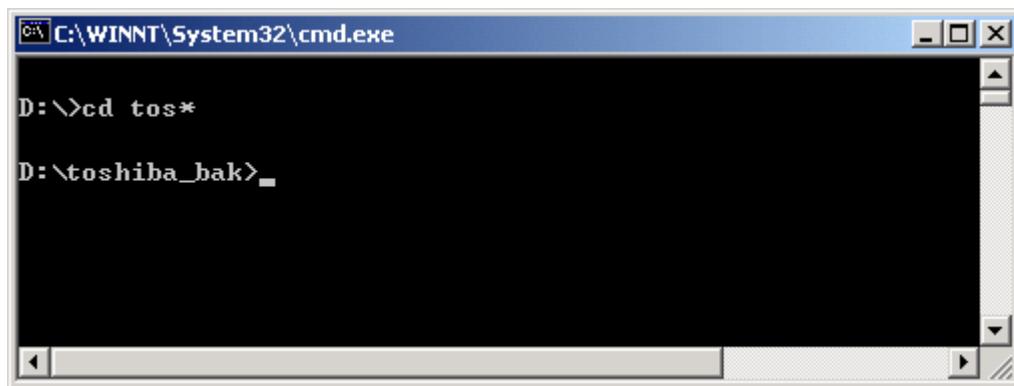
The screenshot shows a Windows Command Prompt window titled '选定 C:\WINNT\System32\cmd.exe'. The window displays several lines of command history:

```
D:\toshiba_bak\书稿\java\网络>java TcpServer  
D:\toshiba_bak\书稿\java\网络>javac TcpS.java  
D:\toshiba_bak\书稿\java\网络>java TcpServer  
D:\toshiba_bak\书稿\java\网络>javac TcpS.java  
D:\toshiba_bak\书稿\java\网络>java TcpServer  
D:\toshiba_bak\书稿\java\网络>
```

图 1.16

### 3. 如何在命令行中快速进入某个目录：

我们在命令行窗口中，经常要用到 cd 命令进入子目录或其他目录，如果目录的名称比较长，我们可以只输入目录名中的部分字符，其他字符用\*替代，如图 1.17 所示。



The screenshot shows a Windows Command Prompt window titled 'C:\WINNT\System32\cmd.exe'. The user has entered the command 'cd tos\*' to navigate to the 'toshiba\_bak' directory.

```
D:\>cd tos*  
D:\toshiba_bak>
```

图 1.17

可见，我们用 cd tos\* 就可以进入 toshiba\_bak 目录。

如果我们在命令行窗口中要进入的目录，已经出现在 Windows 资源管理器窗口中，我们可以在 Windows 资源管理器窗口中选中该目录，按住鼠标左按钮，直接将该目录拖到命令行窗口中，如图 1.18 所示：

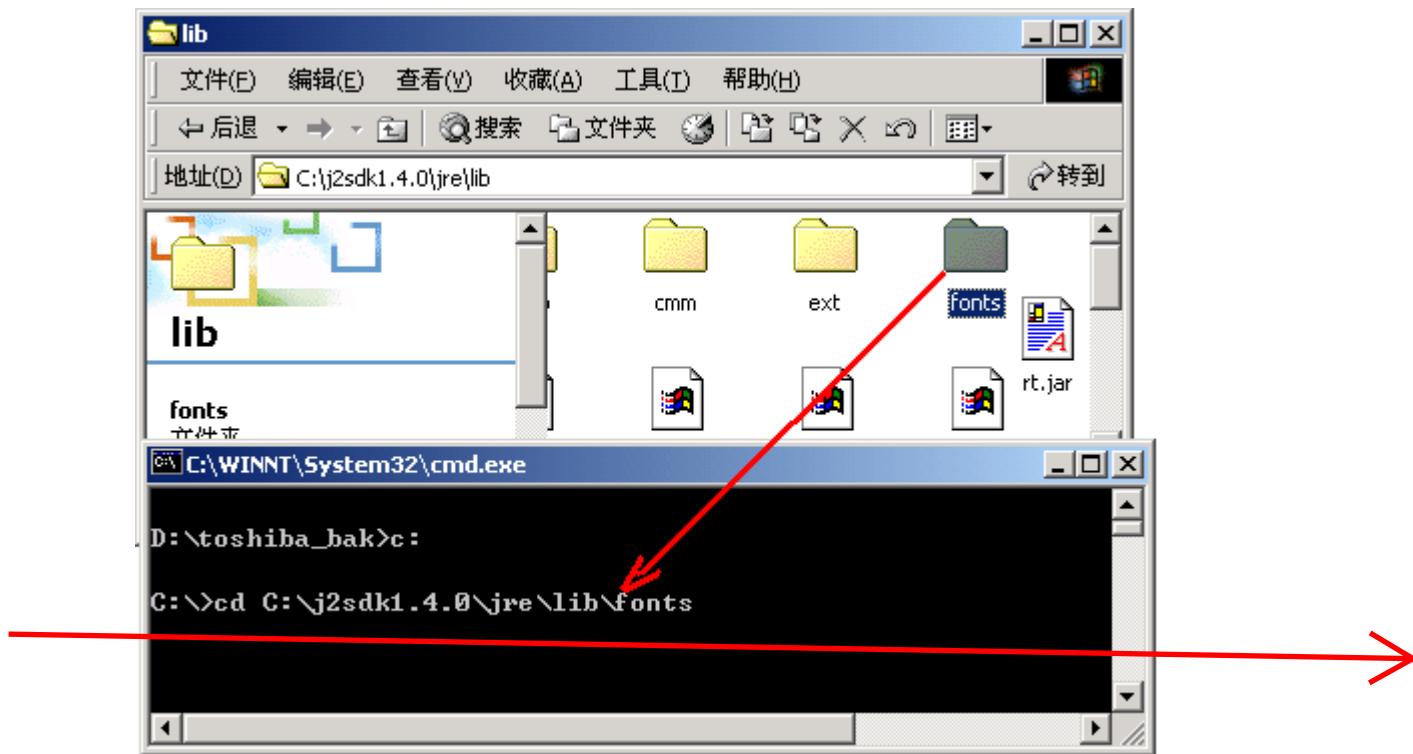


图 1.18

## 1.5 有效利用 Java 的文档帮助

Sun 公司为 JDK 工具包提供了一整套文档资料，我们习惯上称之为 JDK 文档。JDK 文档中提供了 Java 中的各种技术的详细资料，以及 JDK 中提供的各种类的帮助说明。JDK 文档是 Java 语言的完整说明，大多数书籍中的类的介绍都要参照它来完成，它是编程者们最经常查阅的资料。

我们可以从 Sun 公司的网站 <http://java.sun.com> 上下载到最新的 JDK 文档。JDK 文档通常有两种格式：HTML 格式和 CHM 格式（分别如图 1.19、1.20 所示）。其中 HTML 文档属于官方文档，由 Sun 公司定期发布。如果你想了解最新的 Java 知识，建议下载 HTML 格式的文档。目前的 CHM 格式的文档则由一些 Java 爱好者大公无私地奉献而制作出来的，虽然推出时间可能会略晚于 HTML 格式的文档，但是由于 CHM 文档具有独特的搜索功能，更是被许多编程者所钟爱。作者在平时的开发和编写此书时所用的就是 JDK1.4.0 的 CHM 文档。

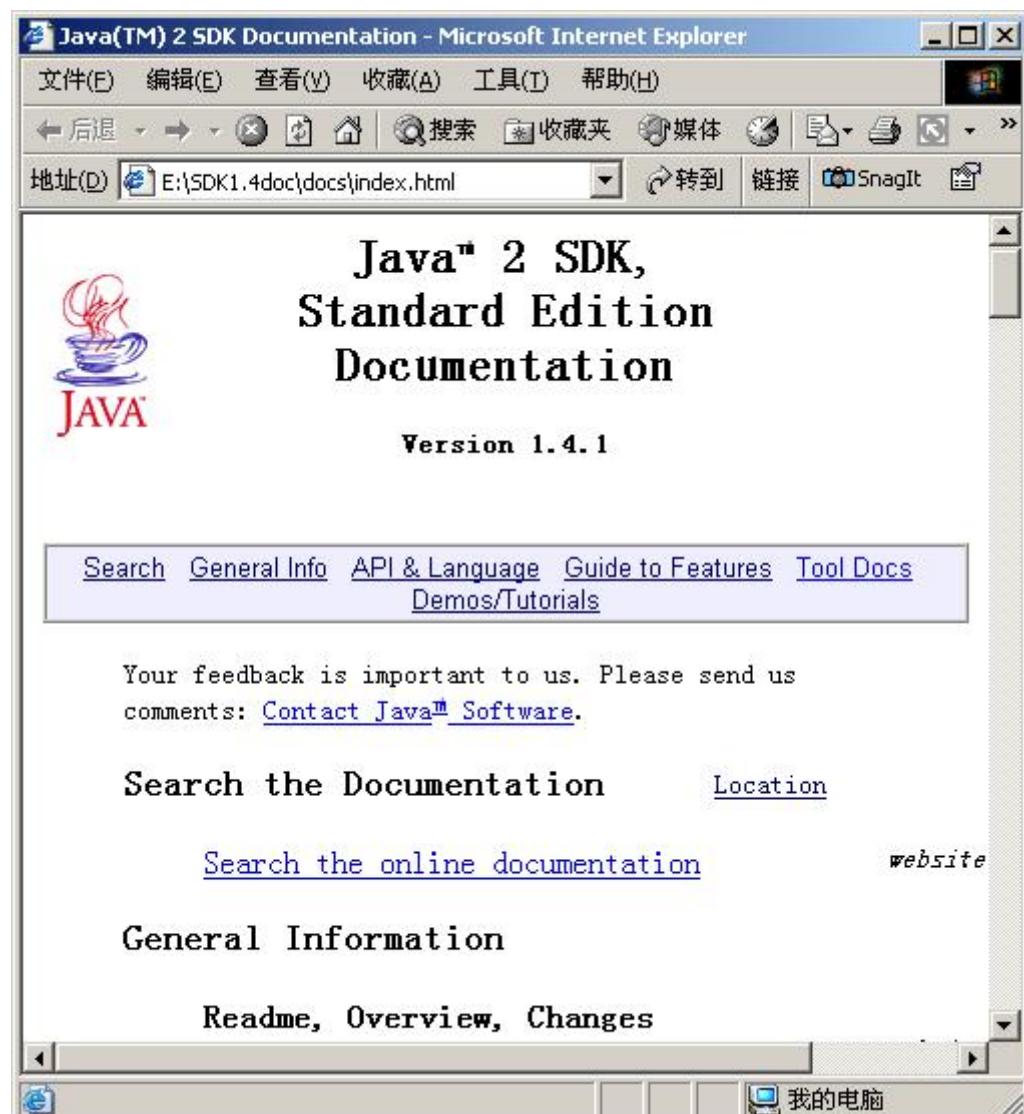


图 1.19

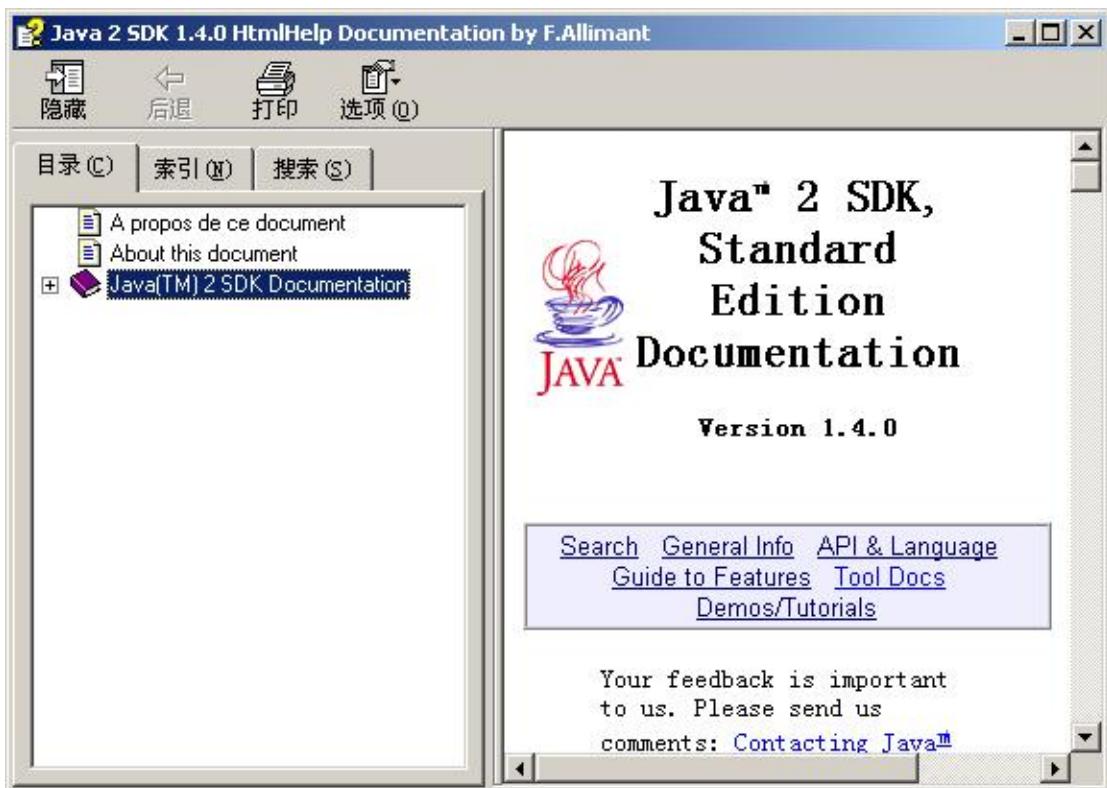


图 1.20

Java 文档的安装非常简单，我们下载到硬盘上的 HTML 文档通常都是一个 ZIP 压缩文件，我们只需要把它解压缩到一个目录里，然后进入目录运行 index.html 即可。对于 CHM 格式的文档，只需要运行解压缩后的 chm 文件就可以了（这里我们使用的是 jdk140.chm）。

我们有时想实现某种功能，但不知道该用什么方法，即使知道具体的方法名，但却不知道怎样使用这个方法，以及这个方法在文档中的具体位置。但我们知道所要完成的功能会涉及到的某些英文单词，我们想通过这些英文单词而查找到我们想要内容，遇到这种情况，CHM 文档的搜索功能就派上用场了。CHM 格式的文档提供了模糊搜索功能，通过搜索功能可以使我们快速找到自己所需要的知识点，Sun 公司推出的 HTML 文档目前还没有提供这种功能。

下面，我们来演示一下怎样使用 CHM 格式的文档。假设我们现在想得到窗口的颜色，我们很容易想到 getcolor 这个英文拼写。我们只需要输入 getcolor 前几个字母，搜索功能便会自动帮我们找到相关的方法，接着我们就能够顺藤摸瓜地找到所需的方法并还能了解到更详细的介绍。如图 1.21 所示。

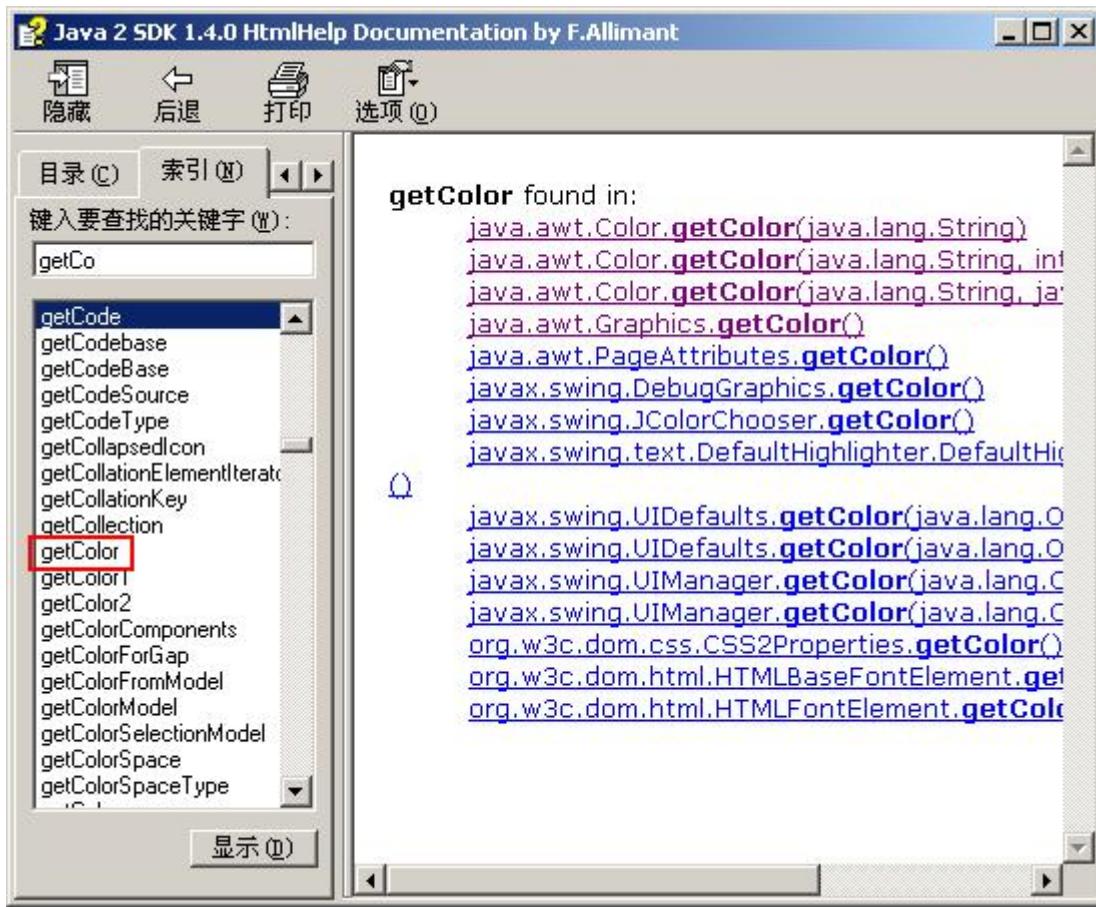


图 1.21

到此为止，读者基本上就可以开始 Java 编程之旅了。在这之前我们再来介绍一些有关 Java 其他方面的知识。

## 1.6 JVM（虚拟机）的运行过程

Java 编译程序将 Java 源程序翻译为 JVM 可执行代码——Java 字节码。这一编译过程同 C/C++ 的编译有些不同。当 C 编译器编译生成一个对象的代码时，该代码是为在某一特定硬件平台运行而产生的。因此，在编译过程中，编译程序通过查表将所有对符号的引用转换为特定的内存偏移量，以保证程序运行。Java 编译器却不将对变量和方法的引用编译为数值引用，也不确定程序执行过程中的内存布局，而是将这些符号引用信息保留在字节码中，由解释器在运行过程中创立内存布局，然后再通过查表来确定一个方法所在的地址。这样就有效的保证了 Java 的可移植性和安全性。

运行 JVM 字节码的工作是由解释器来完成的。解释执行过程分三部进行：代码的装入、代码的校验和代码的执行。装入代码的工作由“类装载器”（class loader）完成。类装载器负责装入运行一个程序需要的所有代码，这也包括程序代码中的类所继承的类和被其调用的类。当类装载器装入一个类时，该类被放在自己的名字空间中。除了通过符号引用自己名字空间以外的类，类之间没有其他办法可以影响其他类。在本台计算机上的所有类都在同一地址空间内，而所有从外部引进的类，都有一个自己独立的名字空间。这使得本地类通过共享相同的名字空间获得较高的运行效率，同时又保证它们与从外部引进的类不会相互影响。当装入了运行程序需要的所有类后，解释器便可确定整个可执行程序的内存布局。解释器为符号引用与特定的地址空间建立对应

关系及查询表。通过在这一阶段确定代码的内存布局，Java很好地解决了由超类改变而使子类崩溃的问题，同时也防止了代码对地址的非法访问。

随后，被装入的代码由字节码校验器进行检查。校验器可发现操作数栈溢出，非法数据类型转化等多种错误。通过校验后，代码便开始执行了。

Java字节码的执行有两种方式：

1. 即时编译方式：解释器先将字节码编译成机器码，然后再执行该机器码。
2. 解释执行方式：解释器通过每次解释并执行一小段代码来完成Java字节码程序的所有操作。

通常采用的是第二种方法。由于JVM规格描述具有足够的灵活性，这使得将字节码翻译为机器代码的工作具有较高的效率。对于那些对运行速度要求较高的应用程序，解释器可将Java字节码即时编译为机器码，从而很好地保证了Java代码的可移植性和高性能。

为了便于读者更加容易的理解，我们用下面的一张图来概括JVM（虚拟机）的运行过程。

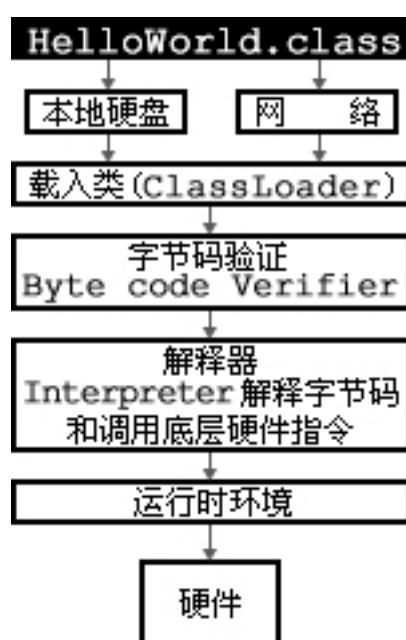


图 1.22

## 1.7 垃圾回收器

Java的一个重要特点就是具有一个垃圾回收器，并且能够自动回收垃圾，这也是Java相对于其他语言有优势的地方。

Java类的实例对象和数组所需的存储空间是在堆上分配的，解释器具体承担为类实例分配空间的工作。解释器在为一个实例对象分配完存储空间后，便开始记录对该实例对象所占用的内存区域的使用。一旦对象使用完毕，便将其回收到垃圾箱中。

在Java语言中，除了new语句外没有其他方法为一个对象申请和释放内存。对内存进行释放和回收的工作是由Java运行系统承担的，这允许Java运行系统的设计师自己决定碎片回收的方法。在Sun公司开发的Java解释器和Hot Java环境中，碎片回收用后台线程的方式来执行，这不但为运行系统提供了良好的性能，而且使程序设计人员摆脱了自己控制内存使用的风险。Java的自动垃圾回收功能解决了两个最常见的应用程序错误：内存泄露和无效内存的引用。

初始化的重要性是不言而喻的，程序员们都能体会到。可是清理垃圾数据的重要性就常常被忽视，当程序的某个部件完成使用后，程序员往往都弃置不顾，这是很危险的，这些垃圾会占据系统资源，一直到系统资源（尤其是内存）被耗尽。Java 提供了一种叫做垃圾回收的机制来避免程序员忽略垃圾的处理，Java 自动帮我们完成垃圾回收的工作，而不用程序员再去考虑。

在 Java 程序运行过程中，一个垃圾回收器会不定时地被唤起检查是否有不再被使用的对象，并释放它们占用的内存空间。垃圾回收器的启用不由程序员控制，也无规律可循，并不会一产生了垃圾，它就被唤起，甚至有可能到程序终止，它都没有启动的机会。因此这并不是一个很可靠的机制，这或许不是件坏事，因为垃圾回收器会给系统资源带来额外负担。它被起用的几率越小，带来额外负担的几率也就越小，当然如果它永远都不被启动，也就永远不必付出额外的代价了。

不同的 Java 虚拟机会采用不同的回收策略，一般有两种比较常用，一种叫做复制式回收策略。这种策略的执行模式是先将正在运行中的程序暂停，然后把正在被使用的所有对象从他们所在的堆内存里复制到另一块堆内存，那些不再被使用的对象所占据的内存空间就被释放掉。

这种机制需要两块堆内存用于将内存中的内容搬运复制，这就需要维护所需内存数量的两倍的内存空间，更麻烦的是即使程序只产生了少量垃圾甚至没有垃圾，回收器仍然会把堆内存里的内容复制到另一块堆内存中，这就使得这种策略效率低下，为解决这种问题，另一种叫做“自省式”的策略被采用。

自省式回收器会检测所有正在使用的对象，并为它们标注，完成这项工作后再将所有不再被使用的对象所占据的内存空间一次释放。可想而知，这种方式的速度仍然很慢，不过如果程序只产生少量垃圾甚至不产生垃圾时，这种策略就极具优势了。

这两种方式颇具互补性，因此在一些 JVM 里两种方式被有机地结合运用，在实际应用中，JVM 会监督这两种模式的运作效率，如果程序中的对象长期被使用，JVM 就转换至“自省式”回收模式，而当产生大量垃圾或对象所占内存不连续情况严重时，又会转换至“复制式”模式，如此循环，实现两种机制的交互。

## 1.8 反编译工具的介绍

### 1.8.1 JAD

本章的主要目的是让读者熟悉 Java 程序的开发过程，掌握相关工具命令的使用，对 Java 有一个初步的认识，我们就不妨再为大家介绍一个 Java 反编译工具——JAD。

如果有时读者想研究一下别人的 Java 程序，手头只有.class 文件，却没有源程序，该怎么办呢？从原理上讲，.class 中的字节码是可以反编译成.java 源文件的，JAD 就是一个可以实现反编译的小工具。关于 JAD 软件的下载，读者可以自己到网上查找一下。

我们假设 JAD 安装在 c:\jad 目录下，首先将先前编译生成的 Test.class 文件复制到此目录下，接着来进行操作。

在命令行窗口环境中进入 jad 目录，然后运行：

```
jad -s java Test.class
```

JAD 工具便将 Test.class 转换为 Test.java。就这么简单，反编译成功了！读者可以看到这个 Test.java 内容跟我们先前自己编写的那个 Test.java 源文件的内容是一样的。

若需要 JAD 更为详细的使用帮助信息，请直接运行 jad.exe 查看。

如果 Test.class 文件和 jad 程序不在同一个目录，我们该如何操作呢？不要忘了先前讲过的 path 环境变量哟！

### 1.8.2 FrontEnd

前面我们介绍了 JAD 工具的一些基本用法，在这里我们顺便再介绍一下 FrontEnd 工具的用法。FrontEnd 是专为 JAD 做的一个图形化操作界面，它的反编译引擎就是 jad.exe，弥补了 JAD 只能在命令行窗口下运行的不足，是对 JAD 在 Windows 操作系统下的扩充。

FrontEnd.exe 必须于 jad.exe 位于同一目录，如图 1.23 所示。

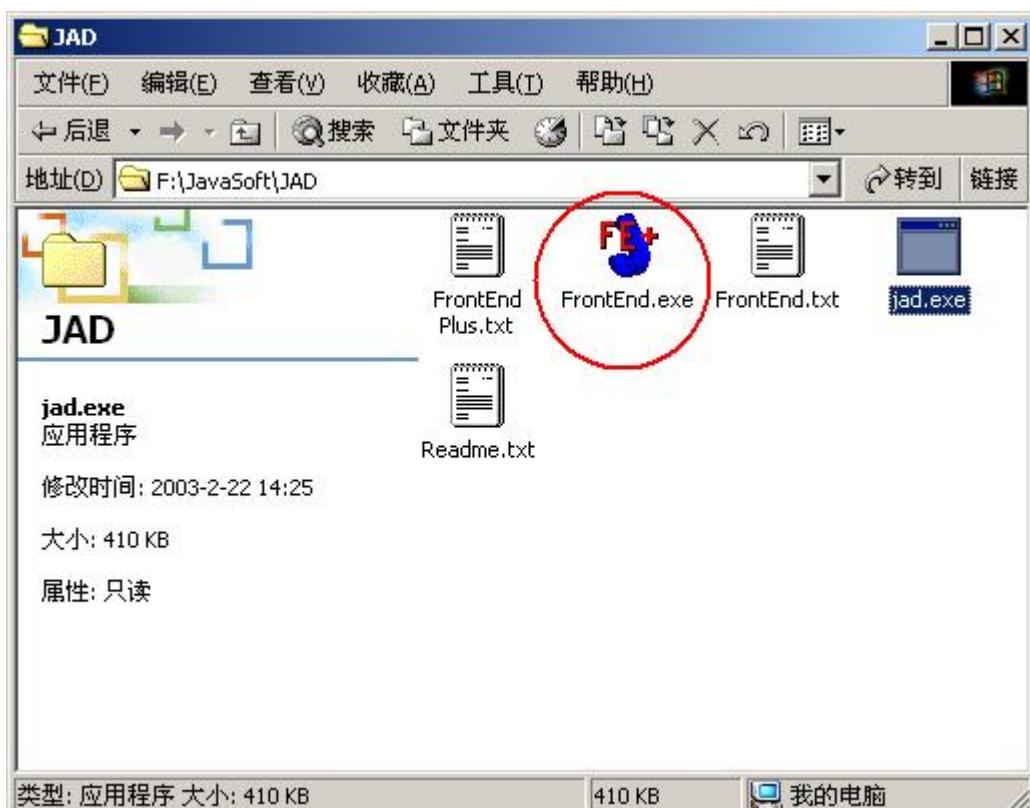


图 1.23

如果我们要使用 FrontEnd 来反编译一个.class 文件，我们只需要运行 FrontEnd，然后在 FrontEnd 的运行界面中选择 “File” --> “DeCompile”，如图 1.24 所示。

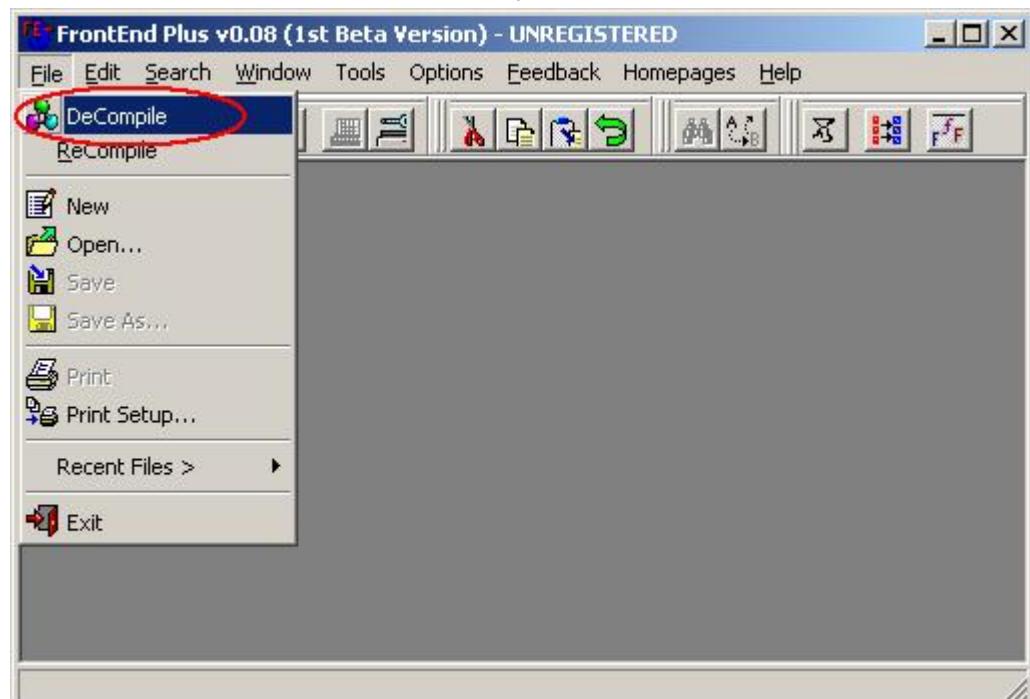


图 1.24

接着选定我们所需要反编译的.class 文件(例如这里我们使用 Send.class), 如图 1.25 所示。

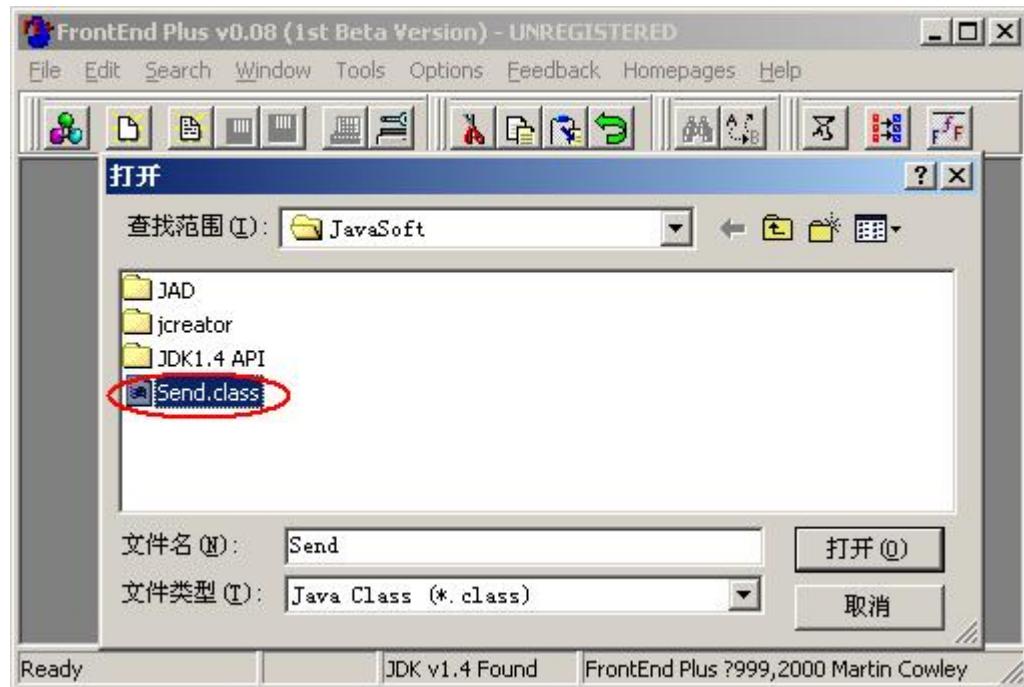


图 1.25

选择好我们需要反编译的.class 文件之后, 点击“打开”, 便能够看到经过反编译之后的源文件内容了, 如图 1.26 所示。其实我们从 FrontEnd 的标题栏中就能看到, Send.class 文件已经成功的被反编译成 Send.java 文件了。到此为止, 反编译成功结束。

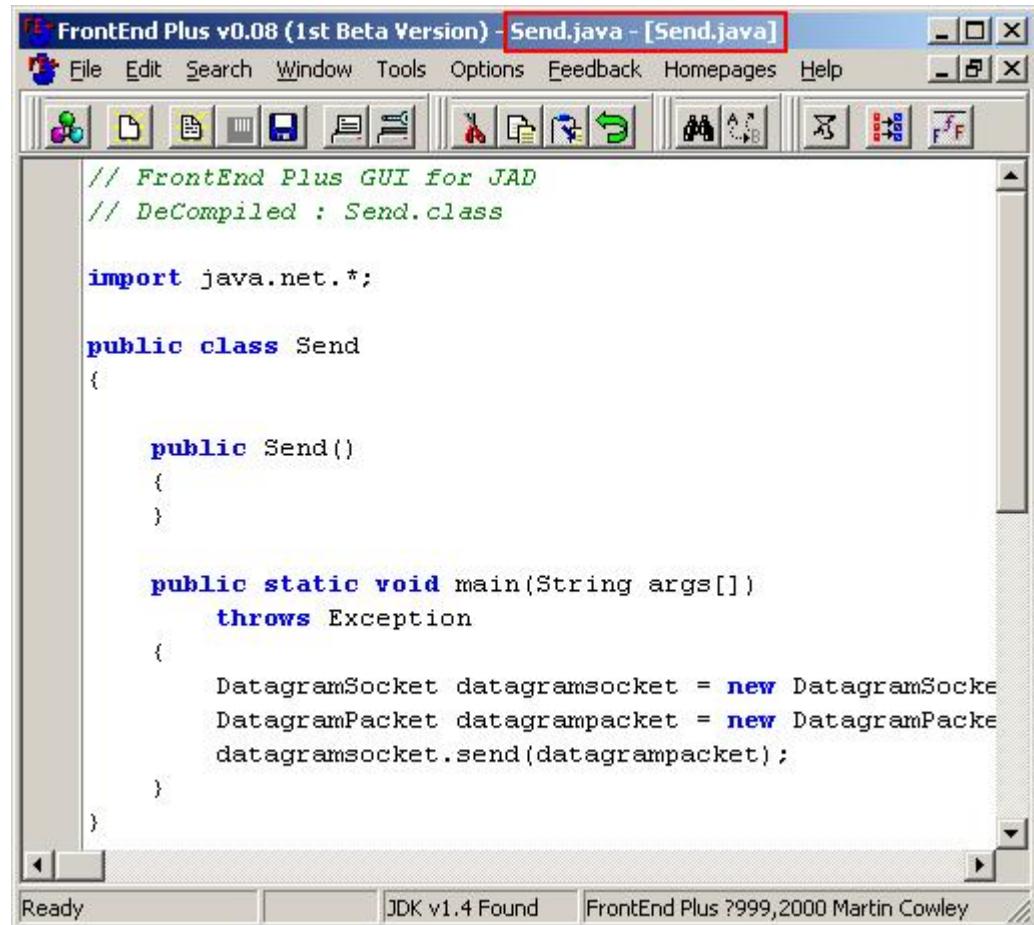


图 1.26

|                                |    |
|--------------------------------|----|
| 第 1 章 Java 开发前奏 .....          | 1  |
| 1.1 Java 虚拟机及 Java 的跨平台原理..... | 2  |
| 1.2 Java 的开发环境的搭建.....         | 3  |
| 1.2.1 环境变量的介绍.....             | 5  |
| 1.2.2 如何查看系统环境变量.....          | 5  |
| 1.2.3 如何设置系统环境变量.....          | 7  |
| 1.3 体验 Java 编程的过程.....         | 8  |
| 脚下留心：运行 Java 程序的注意事项           |    |
| 1.4 classpath 的设置 .....        | 10 |
| 脚下留心：配置 Java 环境变量时应忌讳的问题       |    |
| 多学两招： 1.如何快速得到路径字符串            |    |
| 2.如何在命令行窗口中拷贝和粘帖               |    |
| 3.如何在命令行窗口中快速进入某个目录            |    |
| 1.5 有效利用 Java 的文档帮助.....       | 15 |
| 1.6 JVM (虚拟机) 的运行过程.....       | 18 |
| 1.7 垃圾回收器.....                 | 19 |
| 1.8 反编译工具的介绍.....              | 20 |

|       |               |    |
|-------|---------------|----|
| 1.8.1 | JAD .....     | 20 |
| 1.8.2 | FrontEnd..... | 20 |

# 第 2 章 Java 编程基础

## 2.1 Java 基本语法格式

### 2.1.1 Java 代码的落脚点

Java 中的所有程序代码都必须存在于一个类中，用 class 关键字定义类，在 class 前面可以有一些修饰符。格式如下：

```
修饰符 class 类名  
{  
    程序代码  
}
```

### 2.1.2 Java 是严格区分大小写的

例如，我们不能将 class 写成 Class。

### 2.1.3 Java 是一种自由格式的语言

在 Java 中，所有的程序代码分为结构定义语句和功能执行语句，一条语句可以写在若干行上，功能执行语句的最后必须用分号(;)结束，不必要对齐或缩进一致。可以按自己的意愿任意编排，只要每个词之间用空格、制表符、换行符或大括号、小括号这样的分隔符隔开就行。例如第一章的例子程序改成下面这种编排方式也是可以的：

```
class Test{public static void main(String [  
] args){System.out.println("My first java program");}}
```

用哪种代码书写的格式因个人爱好而定，但出于可读性的考虑不建议使用这种格式。

**M** 脚下留心：

1. Java 程序中一句连续的字符串不能分开在两行中写，以上程序如果写成下面这种方式是会编译出错的：

```
class Test{public static void main(String [  
] args){System.out.println("My first java  
program");}}
```

2. 功能执行语句的最后必须用分号(;)结束，但中国的初学者常将这个英文的(;)误写成中文的(；)自己却找不出错误的原因来，对于这样的情况，编译器通常会报告“illegal character”（非法字符）这样的错误信息。

### 2.1.4 Java 程序的注释

为程序添加注释可以提高程序的可读性，它是写在程序里的信息，用来说明某段程序的作用和功能。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

第一种是单行注释，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。如下例：

```
int c = 10; // 定义一个整型
```

第二种是多行注释，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：

```
/* int c = 10; // 定义一个整型  
   int x=5;      */
```

第三种注释方法是文档注释，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，并能被包含在诸如 javadoc 之类的工具程序提取的文档里，用以说明该程序的层次结构及其方法。关于这种注释的详细用法，我们会在后面的章节中讲解。

## M 脚下留心：

/\*.....\*/ 中可以嵌套 “//” 注释，但不能嵌套 “ /\*\*/ ”，如：下面的注释是非法的：

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

## F 不得不说：

我们要从开始就养成良好的编程风格，软件编码规范中说：“可读性第一，效率第二”。在程序中必须包含适量的注释，以提高程序的可读性和易于维护性，程序注释一般占程序代码总量的 20%-50%。

### 2.1.5 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，不能是关键字。

下面是合法的标识符：

```
identifier,  
username  
user_name  
_userName  
$username
```

下面是非法的标识符：

```
class  
98.3  
Hello World
```

正确的路有一条，错误的路千万条，何苦要去记住有哪些错误的路呢？永远用字母开头，尽量不要包含其他的符号就行了。

## 2.1.6 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`, `break` 等，这些保留关键字不能被当作标识符使用。其实大家不用死记硬背到底有哪些关键字，知道有这回事就足够了，万一不小心把某个关键字用作标识符了，编译器就能告诉我们这个错误。下面是 Java 的关键字列表，大家就留个初步的印象吧！

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

注意：Java 没有 `sizeof`、`goto`、`const` 这些关键字，但不能用 `goto`、`const` 作为变量名。

## 2.1.7 Java 中的常量

常量就是程序里持续不变的值（有的书中称其为字面量或实字），Java 中的常量包含整型常量，浮点数常量，布尔常量等，下面我们来看一下它们是如何表示的：

### 整型常量

整型常量可以分为十进制，十六进制和八进制来表示：

十进制：

0 1 2 3 4 5 6 7 8 9

注意：以十进制表示时，第一位不能是 0（数字 0 除外）。

十六进制：

0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F

注意：以十六进制表示时，需以 `0x` 或 `0X` 开头，如：

`0x8a` `0Xff` `0X9A` `0x12`

八进制：

0 1 2 3 4 5 6 7

注意：八进制必须以 0 开头。如：

`0123` `045` `098` `046`

长整型：

长整型必须以 L 做结尾，如：

`9L` `156L`

### 浮点数常量：

浮点数常量有 `float`(32 位) 和 `double`(64 位) 两种类型，分别叫做单精度浮点数和双精度浮点数，表示浮点数时，要在后面加上 `f(F)` 或者 `d(D)`，用指数表示也可以。注意：由于小数常量的默认类型为 `double` 型，所以 `float` 类型的后面一定要加 `f(F)`，用以区分。如：

`2e3f` `3.6d`

`.4f` `0f`

`3.84d` `5.022e+23f`

都是合法的。

### 布尔常量：

布尔常量用于区分一个事物的正反两面，不是真就是假。其值只有两种：`true` 和 `false`。

#### 字符常量：

字符常量是由英文字母、数字、转义序列、特殊字符等的字符所表示，它的值就是字符本身，如：

`'a'` `'8'` `'\t'` `'\u0027'`

字符常量要用两个单引号括起来，Java 中的字符占用两个字节，是用 `unicode` 码表示的，我们也可以使用 `unicode` 码值加上 `\u` 来表示对应的字符。

#### 字符串常量：

字符串常量和字符型常量的区别就是：前者是用双引号括起来的常量，用于表示一连串的字符。而后者是用单引号括起来的，用于表示单个字符。下面是一些字符串常量：

`"Hello World"` `"123"` `"Welcome \nXXX"`

**& 多学两招：**有些时候，我们在无法直接往程序里面写一些特殊的按键，比如你想打印出一句带引号的字符串，或者判断用户的输入是不是一个回车键，等等。其实它们都可以用一些转义字符来表示，以下一些特殊字符的意义，供参考：

`\r` 表示接受键盘输入，相当于按下了回车键；

`\n` 表示换行；

`\t` 表示制表符，相当于 `tab` 键；

`\b` 表示退格键，相当于 `Back Space`；

`\'` 表示单引号，`\\"` 是双引号；

`\\"` 表示一个斜杠 “`\`”。

比如上面的“`Welcome \n XXX`”，它的运行结果是：

Welcome

XXX

#### null 常量：

`null` 常量只有一个值，用 `null` 表示，表示对象的引用为空。

## 2.2 变量及变量的作用域

### 2.2.1 变量的概念

变量就是系统为程序分配的一块内存单元，用来存储各种类型的数据。根据所存储的数据类型的不同，有各种不同类型的变量。

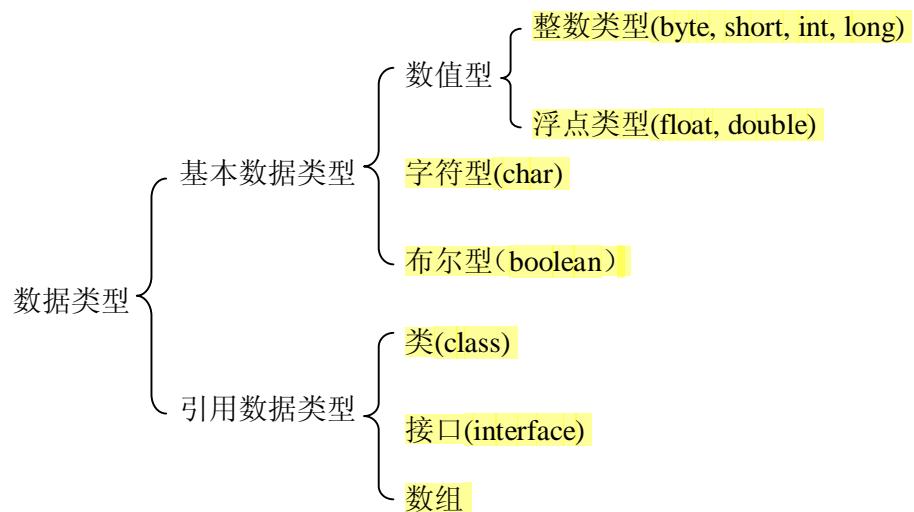
用一个变量定义一块内存以后，程序就可以用变量名代表这块内存中的数据。我们来看一下：

```
int x=0,y;  
y=x+3;
```

第一句代码分配了两块内存用于存储整数，分别用 `x`, `y` 作为这两块内存的变量名，并将 `x` 标识的内存中的数据置为 0，`y` 标识的内存中的数据为其原始状态，可以认为是一个未知数。第二句代码的执行过程，程序首先取出 `x` 代表的那块内存单元的数，加上 3，然后把结果放到 `y` 所在的那块内存单元，这样就完成了 `y=x+3` 的运算。

## 2.2.2 Java 的变量类型

在 Java 中内建有八种基本变量类型来存储整数、浮点数、字符和布尔值。



其中引用数据类型会在以后章节详细讲解，这里只讲基本数据类型。

与其他编程语言不同的是，Java 的基本数据类型在任何操作系统中都具有相同的大小和属性，不像 C 语言，在不同的系统中变量的取值范围不一样，在所有系统中，Java 变量的取值都是一样的，如表 2.1。这也是 Java 跨平台的一个特性。

有四种数据类型用来存储整数，它们具有不同的取值范围，分别如表 2.1 所示：

| 类型名   | 大小   | 取值范围                                     |
|-------|------|--|
| byte  | 8 位  | -128~127                                 |
| short | 16 位 | -32768~32767                             |
| int   | 32 位 | -2147483648~2147483647                   |
| long  | 64 位 | -9223372036854775808~9223372036854775807 |

表 2.1

这些类型都是有符号的，所有整数变量都无法可靠地存储其取值范围以外的数据值，因此定义数据类型时一定要谨慎。

有两种数据类型用来存储浮点数，它们是单精度浮点型（float）和双精度浮点型（double）。浮点数在计算机内存中的表示方式比较复杂，我们在后面的课程为大家进行分析，单精度浮点型和双精度浮点型的取值范围见表 2.2：

| 类型名    | 大小   | 取值范围                                   |
|--------|------|--|
| float  | 32 位 | 1.4E-45~3.4E+38, -1.4E-45~-3.4E+38     |
| double | 64 位 | 4.9E-324~1.7E+308, -4.9E-324~-1.7E+308 |

表 2.2

`char` 类型用来存储诸如字母、数字、标点符号及其他符号之类的单一字符。与 C 语言不同，Java 的字符占两个字节，是 uni code 编码的。



**独家见解：**计算机里只有数值，当你在内存中看到一个数值时，这个数值可能代表各种意义，比如你看到的文字、图像和听到的声音等都是使用数字形式来表示的。生活中的数值也可以代表其他意义，如 1234 可以代表密码，存款额，电报信息等。根据上下线索，我们能够知道这些数值代表的意义。其实，字符也是一个数字，当我们给一个字符变量赋值时，就可以直接用整数，如：97 对应字符'a'，我们使用 `char ch=97` 将字符'a'赋值给变量 `ch`。98 对应的是字符'b'，当在内存里面躺着一个 99 时，请问，它对应键盘上的哪个字母呢？大家都能够猜出就是字符'c'。

如果我们要将字符'x'赋给一个 `char` 变量，该填一个怎样的整数呢？显然，我们不太容易记住每个字符所对应的数字，所以，我们就用单引号加上这个字符本身来表示那个字符对应的数字，如 `char ch='x'`。

`boolean` 类型用来存储布尔值，在 Java 里布尔值只有两个，要么是 `true`，要么就是 `false`。

Java 里的这八种基本类型都是小写的，有一些与它们同名但大小写不同的类，例如 `Boolean` 等，它们在 Java 里具有不同的功能，切记不要互换使用。

### 2.2.3 注意变量的有效取值范围

系统为不同的变量类型分配不同的空间大小，如 `double` 型常量在内存中占八个字节，`float` 的变量占四个字节，`byte` 型占一个字节等，如图 2.1 所示：

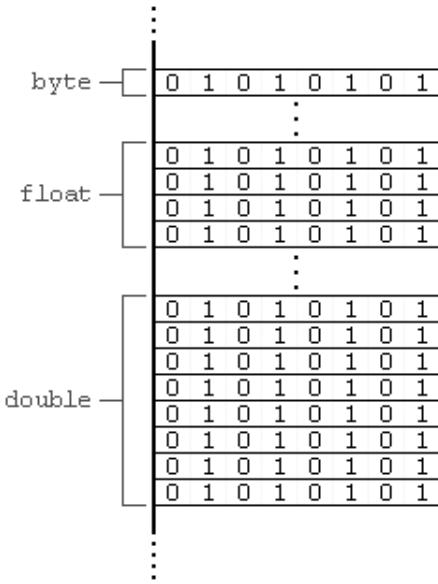


图 2.1

`byte b=129; //` 编译报错，因为 129 超出了 `byte` 类型的取值范围。

`float f=3.5;` // 编译报错，因为小数常量的默认类型为 `double` 型。`double` 型常量在内存中占八个字节，而 Java 只为 `float` 的变量分配四个字节的空间，要将八个字节的内容装入四个字节的容器，显然有问题。改为：

`float f=3.5f;`

编译就可以通过了，因为 `3.5f` 是一个 `float` 型常数，在内存中只占 4 个字节。

### 2.2.4 基本数据类型之间的转换

在编写程序过程中，我们经常会遇到的一种情况，就是需要将一种数据类型的值赋给另一种不同数据类型的变量，由于数据类型有差异，在赋值时就需要进行数据类型的转换，这里就涉及到两个关于数据转换的概念：自动类型转换和强制类型转换。

### ◆ 自动类型转换（也叫隐式类型转换）

要实现自动类型转换，需要同时满足两个条件，第一是两种类型彼此兼容，第二是目标类型的取值范围要大于源类型。例如，当 byte 型向 int 型转换时，由于 int 型取值范围大于 byte 型，就会发生自动转换。所有的数字类型，包括整型和浮点型彼此都可以进行这样的转换。

请看下面的例子：

```
byte b=3;  
int x=b; //没有问题，程序把 b 的结果自动转换成了 int 型了
```

### ◆ 强制类型转换（也叫显式类型转换）

当两种类型彼此不兼容，或目标类型取值范围小于源类型时，自动转换无法进行，这时就需要进行强制类型转换。强制类型转换的通用格式如下：

目标类型 变量=(目标类型)值

例如：

```
byte a;  
int b;  
a = (byte) b;
```

这段代码的含义就是先将 int 型的变量 b 的取值强制转换成 byte 型，再将该值赋给变量 a，注意，变量 b 本身的数据类型并没有改变。由于这类转换中，源类型的值可能大于目标类型，因此强制类型转换可能会造成你的数值不准确，从下面的图中就可以看出强制类型转换时数据传递的过程。

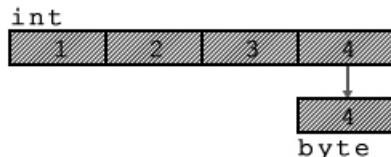


图 2.2

请看下面的程序：

程序清单：Conversion.java

```
public class Conversion  
{  
    public static void main(String[] args)  
    {  
        byte b ;  
        int i = 266 ;  
        b = (byte)i ;  
        System.out.println("byte to int is"+b) ;  
    }  
}
```

程序输出如下：

```
byte to int is 10
```

字符串可以使用加号 (+) 同其他的数据类型相连而形成一个新的字符串，读者只要明白二

进制与十进制数之间的转换关系，就不难明白上面程序打印的结果了。

**\$ 独家见解：**要理解类型转换，大家可以这么想像，大脑前面是一片内存，源和目标分别是两个大小不同的内存块（由变量及数据的类型来决定），将源数据赋值给目标内存的过程，就是用目标内存块去套取源内存中的数据，能套多少算多少。

## 2.2.5 表达式的数据类型自动提升

我们先看下面一个错误程序：

程序清单：Test.java

```
class Test
{
    public static void main(String[] args)
    {
        byte b = 5;
        b = (b-2);
        System.out.println(b);
    }
}
```

这段代码中，`5-2` 的值并未超出 `byte` 型取值范围，然而当执行这段代码时，Java 报出如下错误：

```
Test.java:6: possible loss of precision
found   : int
required: byte
        b = (b-2);
               ^
1 error
```

这是因为在表达式求值时，变量值被自动提升为 `int` 型，表达式的结果也就成了 `int` 型，这时要想把它赋给 `byte` 型变量就必须强制转换了。因此前面代码中粗体的部分就应该改成：

```
b = (byte)(b-2);
```

这种特殊情况在编程过程中如果遇到了，只要知道怎么解决就可以了。

关于类型的自动提升，Java 定义了若干适用于表达式的类型提升规则。

第一，所有的 `byte` 型、`short` 型和 `char` 的值将被提升到 `int` 型。

第二，如果一个操作数是 `long` 型，计算结果就是 `long` 型；

第三，如果一个操作数是 `float` 型，计算结果就是 `float` 型；

第四，如果一个操作数是 `double` 型，计算结果就是 `double` 型。

以下代码演示了 Java 的类型自动提升规则：

程序清单：Promote.java

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

```

byte b = 50;
char c = 'a';
short s = 1024;
int i = 50000;
float f = 5.67f;
double d = .1234;
double result = (f * b) + (i / c) - (d * s);
System.out.println((f * b) + " + " + (i / c) + " - " + (d * s));
System.out.println("result = " + result);
}
}

```

我们看看在下列代码行的类型提升：

```
double result = (f * b) + (i / c) - (d * s);
```

在第一个子表达式  $f * b$  中，变量  $b$  被提升为  $float$  类型，该子表达式的结果也提升为  $float$  类型。接下来，在子表达式  $i/c$  中，变量  $c$  被提升为  $int$  类型，该子表达式的结果提升为  $int$  类型。

然后，子表达式  $d*s$  中的变量  $s$  被提升为  $double$  类型，该子表达式的结果提升为  $double$  类型。

最后，这三个结果值类型分别是  $float$  类型， $int$  类型，和  $double$  类型。 $float$  类型加  $int$  类型的结果是  $float$  类型。然后  $float$  类型减去  $double$  类型，该表达式的最后结果就是  $double$  类型。

## & 多学两招：

`System.out.println()`除了可以在屏幕上打印字符串外，还可以直接在屏幕上打印其他类型的数据，读者再想想 `System.out.println('a'+1)` 在屏幕上的打印结果与 `System.out.println("")+'a'+1)` 在屏幕上的打印结果有什么不同呢？前者将字符'a'与整数1相加后得到的结果是整数98，后者将一个空字符串与字符'a'相连后再与整数1相连形成一个新的字符串“a1”。

### 2.2.6 变量的作用域

大多数组设计语言都提供了“变量作用域”（Scope）的概念。在 C、C++ 和 Java 里，一对花括号中间的部分就是一个代码块，代码块决定其中定义的变量的作用域。代码块由若干语句组成，必须用大括号包起来形成一个复合语句，多个复合语句可以嵌套在另外的一对大括号中形成更复杂的复合语句。如：

```
{
    int x=0;
{
    int y=0;
    y=y+1;
}
    x=x+1;
}
```

代码块决定了变量的作用域，作用域决定了变量的“可见性”以及“存在时间”。

参考下面这个例子：

程序清单：TestScope.java

```
public class TestScope
{
```

```

public static void main(String[] args)
{
    int x = 12;
    {
        int q = 96; // x 和 q 都可用
        System.out.println("x is "+x);
        System.out.println("q is "+q);
    }
    q = x; /* 错误的行，只有 x 可用， q 超出了作用域范围 */
    System.out.println("x is "+x);
}
}

```

**q** 作为在里层的代码块中定义的一个变量，只有在那个代码块中位于这个变量定义之后的语句，才可使用这个变量，**q=x** 语句已经超过了 **q** 的作用域，所以编译无法通过。记住这样的一个道理，在定义变量的语句所属于的那层大括号之间，就是这个变量的有效作用范围，但不能违背变量先定义后使用的原则。

**M** 脚下留心：下面这样书写的代码在 C 和 C++ 里是合法的：

```

{
    int x = 12;
    {
        int x = 96;
        x = x + 4; // x 运算后的结果为 100
    }
    x = x - 5; // x 运算后的结果为 7，而不是 95。
}

```

在 C 和 C++ 里面，上面的两个 **x** 相当于定义了两个变量，第二层大括号里面的代码对 **x** 的操作，都是对第二个 **x** 的操作，不会影响到第一个 **x**。第一层大括号里面的代码对 **x** 的操作，都是对第一个 **x** 的操作，跟第二个 **x** 没有任何关系。

但这种做法在 Java 里是不允许的，因为 Java 的设计者认为这样做使程序产生了混淆，编译器会认为变量 **x** 已在第一层大括号中被定义，不能在第二层大括号被重复定义。

## 2.2.7 局部变量的初始化

在一个函数或函数里面的代码块中定义的变量称为局部变量，局部变量在函数或代码块被执行时被创建，在函数或代码块结束时被销毁。局部变量在进行取值操作前必须被初始化或进行过赋值操作，否则会出现编译错误，如下面的代码：

程序清单：TestVar.java

```

public class TestVar
{
    public static void main(String [] args)
    {
        int x;//应改为 int x=0;
        x=x+1; //这个 x 由于没有初始化，编译会报错。
        System.out.println("x is "+x);
    }
}

```

## 2.3 函数与函数的重载

### 2.3.1 函数

假设我们有一个游戏程序，在程序运行过程中，我们要不断地发射炮弹。发射炮弹的动作都需要使用一段百行左右的程序代码，我们在每次发射炮弹的地方都要重复加入这一段百行程序代码，程序会变得非常臃肿，程序的可读性也就非常差。假如我们要修改发射炮弹的程序代码，需要修改每个发射炮弹的地方，很可能就会发生遗漏。几乎所有的编程语言中都要碰到这个问题，各种编程语言都将发射炮弹的程序代码从原来的主程序中单独拿出来，做成一个子程序，并为这个子程序安排一个名称，在主程序中需要使用到子程序的功能的每个地方，只要写上子程序的名称就行了，计算机便会去执行子程序中的程序代码，当子程序中的代码执行完后，计算机又会回到主程序中接着往下执行。在 Java 中，我们将这种子程序叫函数。

我们来写一个程序来说明函数的作用与编写方式，程序在窗口上打印出 3 个由\*组成的矩形，如下图所示。

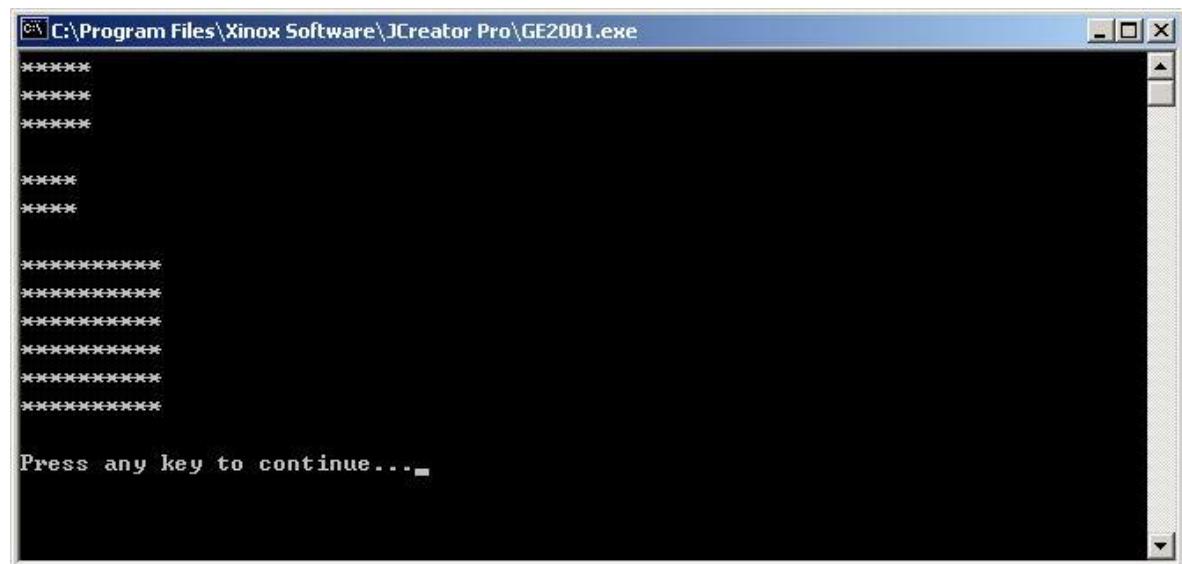


图 2.3

我们可以用下面的程序来实现：

程序清单：Func1.java

```
public class Func1
{
    public static void main(String [] args)
    {
        //下面是打印出第一个矩形的程序代码
        for(int i=0;i<3;i++)
        {
            for(int j=0;j<5;j++)
            {
                System.out.print("*");
            }
            System.out.println(); //换行
        }
        System.out.println();
        //下面是打印出第二个矩形的程序代码
        for(int i=0;i<2;i++)
        {
```

```

        for( int j=0; j<4; j++)
        {
            System.out.print(" * ");
        }
        System.out.println();
    }

    System.out.println();
    //下面是打印出第三个矩形的程序代码
    for( int i=0; i<6; i++)
    {
        for( int j=0; j<10; j++)
        {
            System.out.print(" * ");
        }
        System.out.println();
    }

    System.out.println();
}
}

```

看不懂上面程序中的 `for` 语句的读者，可以先看本章后面关于 `for` 语句的讲解，即使你现在不愿意去看，也没多大关系，你还是能够发现，上面程序中的每一段打印出矩形的代码，除了宽度和高度不一样，其他的地方都一样。我们可以把打印出矩形的代码作为子程序单独从程序中提出来，并用一个“名称”来标记这段代码，以后再碰到要打印矩形时，直接调用这个“名称”就可以了。这样，我们对打印矩形的程序代码只写了一遍，而不用重复书写那么多次了。代码如下：

程序清单：Func2.java

```

public class Func2
{
    public static void drawRectangle(int x,int y)
    {
        for( int i=0; i<x; i++)
        {
            for( int j=0; j<y; j++)
            {
                System.out.print(" * ");
            }
            System.out.println(); //换行
        }

        System.out.println();
    }

    public static void main(String [] args)
    {
        drawRectangle(3,5);
    }
}

```

```
    drawRectangle(2,4);
    drawRectangle(6,10);
}
}
```

以上代码中，我们提出的这段代码就是函数体，用来标记这段代码的“名称”(drawRectangle)，就是函数名，函数名和函数体共同组成了函数，在 Java 中，我们也称之为方法。这个函数需要接受两个整数类型的参数，一个代表矩形的宽度，另一个代表这个矩形的高度。有时候，函数还需要返回一个结果，如果我们要编写一个求解矩形面积的函数，该函数就要返回一个代表面积的结果，函数的返回结果都是有类型的。所以一个函数的定义必须由三部分组成，定义一个函数的格式如下：

```
返回值类型 函数名 (参数类型 形式参数 1, 参数类型 形式参数 2, ....)
{
    程序代码
    return 返回值;
}
```

其中：

**形式参数**：在方法被调用时用于接收外部传入的数据的变量。

**参数类型**：就是该形式参数的数据类型。

**返回值**：方法在执行完毕后返还给调用它的程序的数据。

**返回值类型**：函数要返回的结果的数据类型。

**实参**：调用函数时实际传给函数形式参数的数据。

如果一个函数不需要返回值，我们可以省略最后的 return 语句。如果你的函数里没有 return 语句，则编译时系统会自动在函数的最后添加一个“return;”。



**独家见解：**如何理解函数返回值类型为 void 的情况，如上面打印矩形的函数，不用返回任何结果，那么，返回值类型那里该填什么呢？对于这种情况，我们就用 void 作为返回值类型，意思是“不知道是什么类型，可定义函数时又非要填写一个返回值类型，就用它充数吧！”

下面我们再编写一个求矩形面积的函数，来了解一下函数有返回值的情况。

程序清单：Func3.java

```
public class Func3
{
    public static int getArea(int x,int y)
    {
        return x*y;
    }

    public static void main(String [] args)
    {
        int area = getArea(3,5);
        System.out.println("first Acreage is " + area);
        System.out.println("second Acreage is "+ getArea(2,4));
        getArea(6,10);
    }
}
```

```
}
```

在上面的 getArea 函数中，用到了一个“return (返回值)”语句，用于终止函数的运行并返回该函数的结果给调用者。

如果函数没有返回值或调用程序不关心函数的返回值，可以用下面这样的格式调用定义的函数：

**函数名 (实参 1, 实参 2, ....)**

如上面的 getArea(6, 10); 语句。

如果调用程序需要函数的返回结果，我们要用下面这样的格式调用定义的函数::

**变量 = 函数名 (实参 1, 实参 2, ....)**

如上面 int area = getArea(3, 5); 语句。

对于有返回值的函数调用，我们也可以在程序中直接使用返回的结果，如这一句：

```
System.out.println("second Acreage is " + getArea(2, 4));
```

我们还可以在函数的中间使用 return 语句提前返回，如打印矩形面积的函数，首先应检查传入的参数（即宽度和高度）是否为负数，为负则提前返回。修改上面的 Func3.java，如下：

程序清单：Func4.java

```
public class Func4
{
    public static int getArea(int x,int y)
    {
        if(x<=0|y<=0)
            return -1;
        return x*y;
    }

    public static void main(String [] args)
    {
        int area = getArea(3,5);
        System.out.println("first Acreage is " + area);
        System.out.println("second Acreage is " + getArea(2,4));
        getArea(6,10);
    }
}
```

这样的程序对传入的参数值进行了检查控制，明显要专业得多，也是软件编码规范中的一个起码要求。很多程序错误都是由非法参数引起的，我们应该充分理解并有效地使用类似上面的方式来防止此类错误。

Java 中所有函数都包含在类里面，在 Java 的一个类中定义的函数也叫这个类的方法(method)，本书中提到的函数就是方法，方法就是函数。

### 2.3.2 函数的参数传递过程

下面我们来看一下函数的参数传递过程，如图：

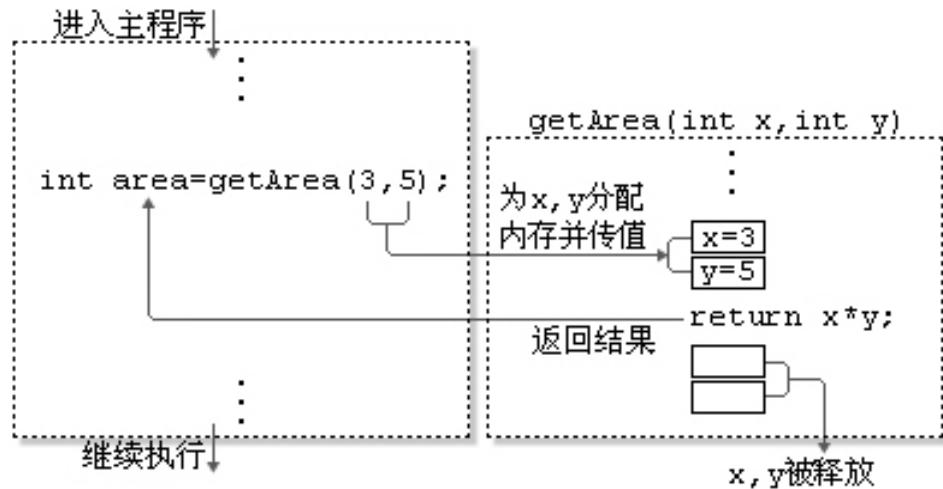


图 2.4

前面讲过，Java 程序运行时虚拟机会先找到这个程序的 `main` 函数，接着从 `main` 函数里面取出一条一条代码来执行，以上面的程序 `Func4.java` 为例，当执行到 `int area=getArea(3,5);` 这个语句时，程序会跳转到 `getArea(int x,int y)` 这个函数的内部去执行，先把实参 `(3,5)` 分别赋值给形式参数 `(int x,int y)`，然后返回 `x*y` 的结果 (`return x*y;`)，接着回到 `main` 里面，把结果赋值给整型变量 `area`。这就是函数的参数传值整个过程。

**& 多学两招：**形式参数 `x` 和 `y` 就相当于函数 `getArea` 中定义的两个局部变量，在函数被调用时创建，并以传入的实参作为初始值，函数调用结束时也就被释放了，不会影响到主程序中其他的 `x` 和 `y`（如果有的话），因为它们属于不同作用域中的变量，它们是互不相干的变量。

### 2.3.3 函数的重载

函数的重载就是在同一个类中允许同时存在一个以上的同名函数，只要它们的参数个数或类型不同即可。在这种情况下，该函数就叫被重载 (overloaded) 了，这个过程称为函数的重载 (method overloading)。

如下面的程序：

程序清单：Test.java

```
public class Test
{
    public static void main(String [] args)
    {
        int isum;
        double fsum;
        isum=add(3,5);
        isum=add(3,5,6);
        fsum=add(3.2,6.5);
    }
}
```

```

public static int add(int x,int y)
{
    reutrn x+y;
}
public static int add(int x,int y,int z)
{
    return x+y+z;
}
public static double add(double x,double y)
{
    return x+y;
}
}

```

Java 的编译器很聪明，能够根据调用函数时所传递的参数的个数和类型选择相应的函数。重载函数的参数列表必须不同，要么是参数的个数不同，要么是参数的类型不同。重载函数的返回值类型可以相同，也可以不同。

思考一下：如果两个方法的参数类型和个数完全一样，返回值类型不同，行不行呢？如果你是 Java 的设计者，而且你的用户在程序里编写了这样的两个方法，在调用时，你能根据他所传递的参数来为他选择到底该用那个吗？显然没有办法吧！那就是不能这样做呗！学编程不需要死记硬背，靠的是动脑筋来思考，这样的学习才能做到举一反三、触类旁通。

## 2.4 Java 中的运算符

运算符是一种特殊符号，用以表示数据的运算、赋值和比较。一般由一至三个字符组成，但 Java 将其视为一个符号。运算符共分以下几种：

- | 算术运算符
- | 赋值运算符
- | 比较运算符
- | 逻辑运算符
- | 移位运算符

### 2.4.1 算术运算符

加减乘除的四则运算相信读者都很熟悉了，在此不再赘述，这里给出一个表来供读者参考：

| 算术运算符 |    |          |    |
|-------|----|----------|----|
| 运算符   | 运算 | 范例       | 结果 |
| +     | 正号 | +3       | 3  |
| -     | 负号 | b=4; -b; | -4 |
| +     | 加  | 5+5      | 10 |
| -     | 减  | 6-4      | 2  |
| *     | 乘  | 3*4      | 12 |
| /     | 除  | 5/5      | 1  |

|    |       |             |          |
|----|-------|-------------|----------|
| %  | 取模    | 5%5         | 0        |
| ++ | 自增（前） | a=2; b=++a; | a=3; b=3 |
| ++ | 自增（后） | a=2; b=a++; | a=3; b=2 |
| -- | 自减（前） | a=2; b=--a  | a=1; b=1 |
| -- | 自减（后） | a=2; b=a--  | a=1; b=2 |
| +  | 字符串相加 | "He"+"llo"  | "Hello"  |

表 2.3

“+”除字符串相加功能外，还能把非字符串转换成字符串，条件是该表达式中至少有一个字符串对象，如：“x”+123; 的结果是“x123”。

## M 脚下留心：

- | 由于 Java 运算符有从左到右的优先顺序，这里请特别注意一下++a 和 a++以及--a 和 a--的区别，  
b=++a 是 a 先自增，a 自己变了后才赋值给 b，而 b=a++ 是先赋值给 b，a 后自增。
- | 如果对负数取模，可以把模数负号忽略不记，如：5%-2=1。但被模数是负数就另当别论了。
- | 对于除号 “/”，它的整数除和小数除是有区别的：整数之间做除法时，只保留整数部分而舍弃小数部分。猜一下这三句代码的结果：int x=3510;x=x/1000\*1000;System.out.println(x)，再运行一下，看看结果。再想一下为什么？你猜想的结果可能是 3510，但实际运行结果是 3000，其实很简单，在程序运行到表达式 “x/1000”的时候，它的结果是 3，而不是 3.51。

## & 多学两招：

读者在日常的编程当中，尽量多读一些别人优秀的源程序，积累一些好的算法，这样对你以后的工作会有很大帮助。如下面的留言板分页问题，和走马灯问题，有的程序员可能会用到一大堆的 if else 和 for 语句来判断和循环，作者在这里给出两个经典的算法。

思考题 1：某个培训中心要为新到的学员安排房间，假设共有 x 个学员，每个房间可以住 6 人，让你用一个公式来计算他们要住的房间数（千万不要像向我以前的有些学员开玩笑，说男生和女生是不能分在一块的，我们就不在这考虑了）？

答案：(x+5)/6。这种算法还可用在查看留言板的分页显示上：其中 x 是总共的留言数，6 是每页显示的留言数，结果就是总共有多少页。

思考题 2：假设你要让 x 的值在 0 至 9 之间循环变化，请写出相应的程序代码。

答案：

```
int x=0;
while(true)
{
    x = (x+1)%10;
}
```

这样，x 的取值总在 0 与 9 之间循环。

## 2.4.2 赋值运算符

如下表：

| 赋值运算符 |     |                 |           |
|-------|-----|-----------------|-----------|
| 运算符   | 运算  | 范例              | 结果        |
| =     | 赋值  | a=3; b=2;       | a=3; b=2; |
| +=    | 加等于 | a=3; b=2; a+=b; | a=5; b=2; |

|                 |     |                              |                        |
|-----------------|-----|------------------------------|------------------------|
| <code>-=</code> | 减等于 | <code>a=3; b=2; a-=b;</code> | <code>a=1; b=2;</code> |
| <code>*=</code> | 乘等于 | <code>a=3; b=2; a*=b;</code> | <code>a=6; b=2;</code> |
| <code>/=</code> | 除等于 | <code>a=3; b=2; a/=b</code>  | <code>a=1; b=2;</code> |
| <code>%=</code> | 模等于 | <code>a=3; b=2; a%=b</code>  | <code>a=1; b=2;</code> |

表 2.4

注：在 JAVA 里可以把赋值语句连在一起，如：

`x = y = z = 5 ;`

在这个语句中，所有三个变量都得到同样的值 5。

还可以把 `x = x + 3` 简写成 `x += 3`，所有运算符都可以此类推，看多了，写多了就习惯了。

### 2.4.3 比较运算符

比较运算符的结果都是 boolean 型的，也就是要是 true，要是 false，如下表：

| 比较运算符                    |           |  |       |
|--------------------------|-----------|--|-------|
| 运算符                      | 运算        | 范例                                     | 结果    |
| <code>==</code>          | 相等于       | <code>4==3</code>                      | False |
| <code>!=</code>          | 不等于       | <code>4!=3</code>                      | True  |
| <code>&lt;</code>        | 小于        | <code>4&lt;3</code>                    | False |
| <code>&gt;</code>        | 大于        | <code>4&gt;3</code>                    | True  |
| <code>&lt;=</code>       | 小于等于      | <code>4&lt;=3</code>                   | False |
| <code>&gt;=</code>       | 大于等于      | <code>4&gt;=</code>                    | False |
| <code> instanceof</code> | 检查是否是类的对象 | <code>"Hello" instanceof String</code> | True  |

表 2.5

**M**脚下留心：比较运算符 “`==`” 不能误写成 “`=`”，特别是在 C/C++ 中，如果你少写了一个 “`=`”，那就不是比较了，整个语句变成了赋值语句，所以有经验的人干脆写成 `int x=5; if(3==x)` 这种样式，将常量放在 “`==`” 前面，万一不小心而少写了一个 “`=`”，就变成了给常量赋值，编译器报错。对于 Java，则不用这样处心积虑，即使写成 `if(x =3)`，编译器也会报错，因为 Java 的条件判断只接受布尔值，要么是 true，要么是 false。不过，也不能掉以轻心，如果你正好定义了一个布尔变量 b，比如 `b=false`；然后用 `if(b = true)`；这一句，编译器就不会报错了，显然与你的愿望背道而驰。

### 2.4.4 逻辑运算符

逻辑运算符用于对 boolean 型结果的表达式进行运算，运算的结果都是 boolean 型，如下表：

| 逻辑运算符                   |          |                                  |                    |
|-------------------------|----------|----------------------------------|--------------------|
| 运算符                     | 运算       | 范例                               | 结果                 |
| <code>&amp;</code>      | AND (与)  | <code>false&amp;true</code>      | <code>false</code> |
| <code> </code>          | OR (或)   | <code>false true</code>          | <code>true</code>  |
| <code>^</code>          | XOR (异或) | <code>true^false</code>          | <code>true</code>  |
| <code>!</code>          | Not (非)  | <code>!true</code>               | <code>false</code> |
| <code>&amp;&amp;</code> | AND(短路)  | <code>false&amp;&amp;true</code> | <code>false</code> |
| <code>  </code>         | OR(短路)   | <code>false  true</code>         | <code>true</code>  |

表 2.6

“`&`”和“`&&`”的区别在于，如果使用前者连接，那么无论任何情况，“`&`”两边的表达式都会参与计算。如果使用后者连接，当“`&&`”的左边为`false`，则将不会计算其右边的表达式。

请看下面的例子：

```
public class TestAnd
{
    public static void main(String[] args)
    {
        int x=0;
        int y=0;
        if(x!=0 && y==y/x)
            System.out.println("y = "+y);
    }
}
```

上面例子中，由于`while`语句的判断条件中的第一个布尔表达式是不成立的，程序就不会判断第二个布尔表达式的值，这就是“短路”。如果两个表达式之间用“`&`”来连接，而且恰好又碰到上面所示的特殊情况，程序运行时就会出错。

OR 运算符叫逻辑或，由“`|`”或“`||`”连接两个布尔表达式，只要运算符两边任何一个布尔表达式为真，该组合就会返回`true`值。

“`|`”和“`||`”的区别与“`&`”和“`&&`”的区别一样。

XOR 运算符叫做异或，只有当“`^`”连接的两个布尔表达式的值不相同时，该组合才返回`true`值。如果两个都是`true`或都是`false`，该组合将返回`false`值。

## 2.4.5 位运算符

我们知道，任何信息在计算机中都是以二进制的形式保存的，“`&`”、“`|`”和“`^`”除了可以作为逻辑运算符，也可以做为位算符，它们对两个操作数中的每一个二进制位都进行运算。

只有参加运算的两位都为`1`，`&`运算的结果才为`1`，否则就为`0`。

只有参加运算的两位都为`0`，`|`运算的结果才为`0`，否则就为`1`。

只有参加运算的两位不同，`^`运算的结果才为`1`，否则就为`0`。

除了这些位运算操作外，我们还可以对数据按二进制位进行移位操作，Java 的移位运算符有三种：

`<<` 左移

`>>` 右移

`>>>` 无符号右移

左移很简单，就不用多说了，右移则有些道道要讲了。对于 C 语言来说，无符号位的整数右移时，左边移空的高位都会填入`0`。有符号的数右移时，如果最高位为`0`，左边移空的高位都会填入`0`，但对最高位为`1`时，左边移空的高位是填入`0`还是`1`，要取决于所在的计算机系统，有的系统填入`0`，也有的系统是填入`1`，这也是 C 语言不跨平台的一个小的方面。

Java 语言是跨平台的，不能像 C 语言有这样的二义性，对 Java 来说，有符号的数据（Java 语言中没有无符号的数据类型）用“`>>`”移位时，如果最高位是`0`，左边移空的高位就填入`0`，如果最高位是`1`，左边移空的高位就填入`1`。同时，Java 也提供了一个新的移位运算符“`>>>`”，不管通过“`>>>`”移位的整数最高位是`0`还是`1`，左边移空的高位填入`0`。

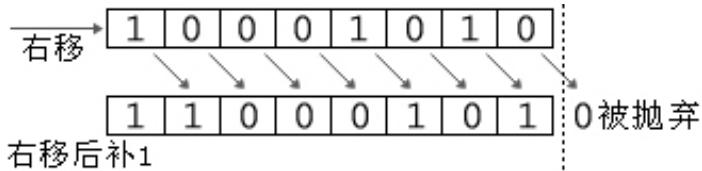


图 2.5

为了让读者能够亲身感受到这“`>>`”和“`>>>`”的区别，我们编写一个程序，来查看其输出结果。

程序清单：`ShiftTest.java`

```
public class ShiftTest
{
    public static void main(String [] args)
    {
        int x=0x80000000;
        int y=0x80000000;
        x=x>>1;
        y=y>>>1;
        System.out.println("0x80000000>>1 = " + Integer.toHexString(x));
        System.out.println("0x80000000>>>1 = " + Integer.toHexString(y));
    }
}
```

运行结果如下：

```
0x80000000>>1 = c0000000
0x80000000>>>1 = 40000000
```

在上面的程序中，我们用 `0x80000000` 这个十六进制表示的整数来作为我们的用例数据，因为我们很容易看出它的二进制形式 `1000,0000, 0000, 0000, 0000, 0000, 0000, 0000`。

为了直观，对移位后的结果，我们也用十六进制来显示，Java 提供的 `Integer.toHexString` 这个静态函数正好可以完成这种需求。

`0xc0000000` 的二进制形式为 `1100,0000, 0000, 0000, 0000, 0000, 0000, 0000`。

`0x40000000` 的二进制形式为 `0100,0000, 0000, 0000, 0000, 0000, 0000, 0000`。

通过上面的例子，我们很清楚的看到了“`>>`”和“`>>>`”运算符的区别。

读者可能会问：你怎么想到将整数转换成十六进制的字符串，要用 `Integer.toHexString` 方法呢？

还记得我们在第一章中讲过的 JDK 文档工具吗？当我们要完成某个功能时，可以通过模糊查询来查找具体的信息。`Hex` 是十六进制的英文单词，我首先在 `chm` 格式的帮助文档的“索引”栏中输入 `hex`，没有发现有价值的提示，如图 2.6：

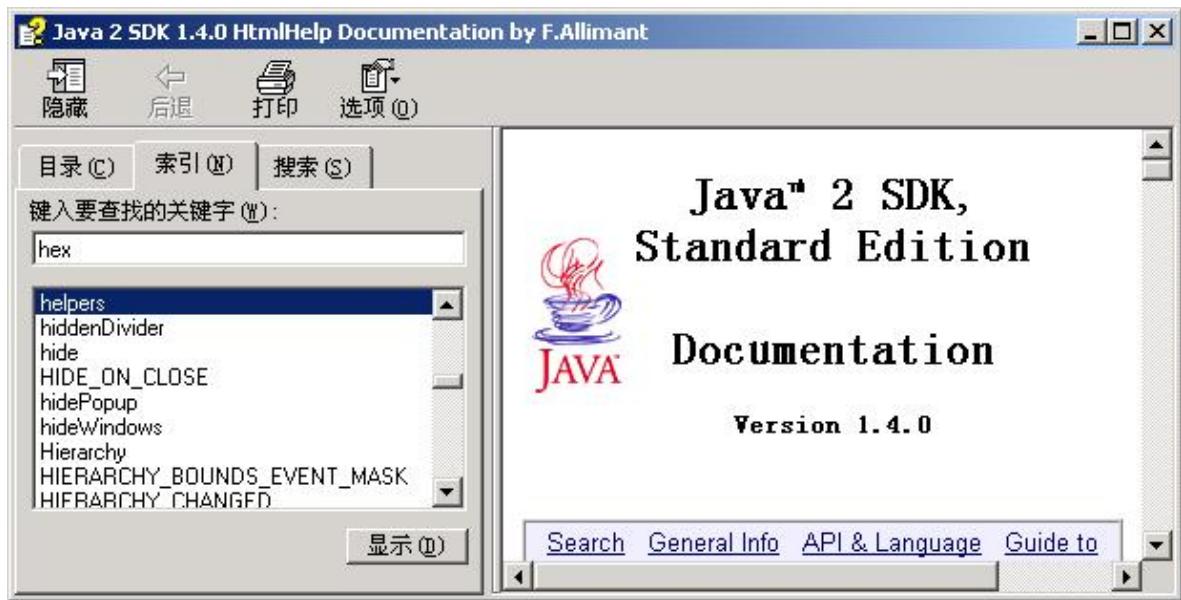


图 2.6

接着，我想到了 `tohex`(到十六进制)，在“索引”栏中输入 `tohex`，果然就找到了我期望的东西，如图 2.7

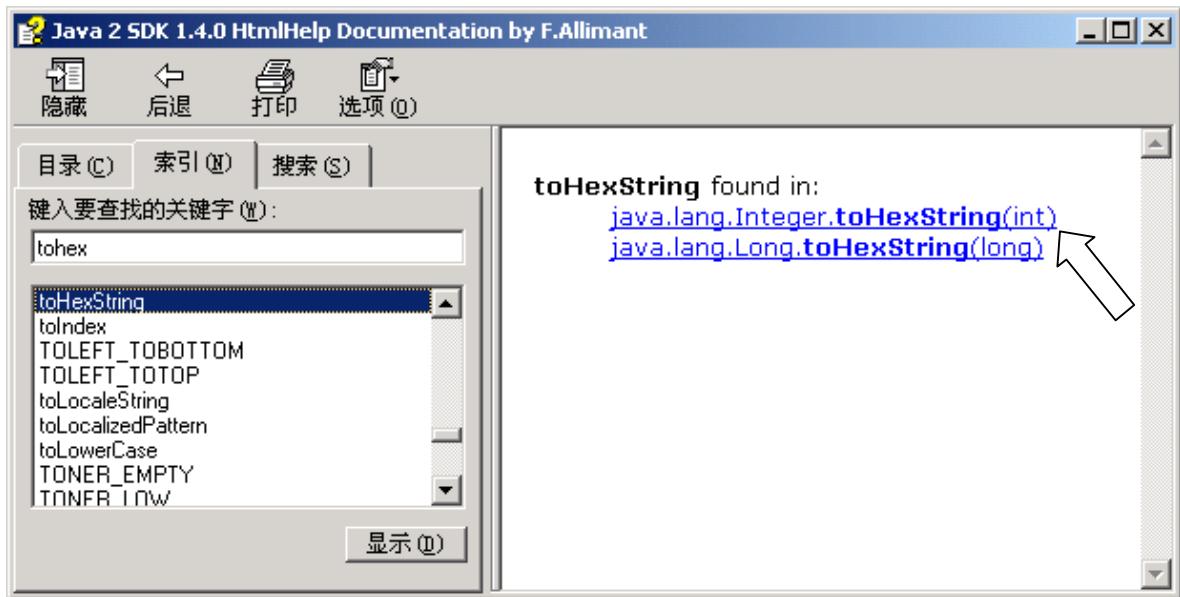


图 2.7

查阅文档没有什么难的，唯一要求的就是你的英语水平不要太低，一般说来词汇量要到英语三级的水平。等用多了，你就会变得非常熟悉这种方式的，初学者没必要为此担心！

**注意：**以上移位运算符适用数据类型有 `byte`、`short`、`char`、`int`、`long`。

1. 对低于 `int` 型的操作数将先自动转换为 `int` 型再移位。
2. 对于 `int` 型整数移位 `a>>b`，系统先将 `b` 对 32 取模，得到的结果才是真正移位的位数。  
例如：`a>>33` 和 `a>>1` 结果是一样的，`a>>32` 的结果还是 `a` 原来的数字。
3. 对于 `long` 型整数移位时 `a>>b`，则是先将移位位数 `b` 对 64 取模。

**&** **多学两招：**移位能为我们实现整数除以或乘以 2 的  $n$  次方的效果，如：

`x>>1` 的结果和 `x/2` 的结果是一样的, `x<<2` 和 `x*4` 的结果也是一样的。总之, 一个数左移  $n$  位, 就是等于这个数乘以 2 的  $n$  次方, 一个数右移  $n$  位, 就是等于这个数除以 2 的  $n$  次方。请思考: 屏幕上每个像素的颜色用一个 8 位的二进制数据表示, 那么屏幕上最多能显示多少种颜色? 这其实就是如何用程序实现求 2 的  $x$  次方的问题。答案:  $y = 1<< x;$

**M** 脚下留心: 移位不会改变变量本身的值。如 `a>>1;` 在一行语句中单独存在, 毫无意义, 因为它没有改变 `a` 的值, 也没有把它赋值给别的变量。作者在深更半夜编程时, 常犯这错误, 也帮不少人排除过类似的错误。

## 2.4.6 运算符的优先级

以上介绍的运算符都有不同的优先级, 所谓优先级就是在表达式运算中的运算顺序, 表 2.3 列出了包括分隔符在内的所有运算符的优先级顺序, 上一行中的运算符总是优先于下一行的:

|     |     |     |     |        |    |      |      |    |    |   |
|-----|-----|-----|-----|--------|----|------|------|----|----|---|
| .   | [ ] | ( ) | { } | ;      | ,  |      |      |    |    |   |
| ++  | --  | ~   | !   | (数据类型) |    |      |      |    |    |   |
| *   | /   | %   |     |        |    |      |      |    |    |   |
| +   | -   |     |     |        |    |      |      |    |    |   |
| <<  | >>  | >>> |     |        |    |      |      |    |    |   |
| <   | >   | <=  | >=  |        |    |      |      |    |    |   |
| ==  | !=  |     |     |        |    |      |      |    |    |   |
| &   |     |     |     |        |    |      |      |    |    |   |
| ^   |     |     |     |        |    |      |      |    |    |   |
|     |     |     |     |        |    |      |      |    |    |   |
| &&  |     |     |     |        |    |      |      |    |    |   |
|     |     |     |     |        |    |      |      |    |    |   |
| ? : |     |     |     |        |    |      |      |    |    |   |
| =   | *=  | /=  | %=  | +=     | -= | <=>= | >=>= | &= | ^= | = |

表 2.3

根据上面表中显示的优先级, 我们来分析一下 `int a =2; int b = a + 3*a;` 语句的执行过程, 程序先执行  $3*a$  后再与 `a` 相加, 最后将结果赋值给等号左边的 `b`, 所以 `b` 的结果为 8。我们可以使用括号改变运算符的优先级, 如我们将第二句修改成 `int b = (a+3) * a;` 后, 程序先执行括号中的  $a+3$  后再与 `a` 相乘, 最后将结果赋值给等号左边的 `b`, 所以 `b` 的结果为 10。

对于 `int a =2; int b= a + 3 * a++;` 这样的语句, `b` 最终等于多少呢? 作者试验得到的结果是 8。

对于 `int a =2; int b= (a ++)+ 3 * a;` 这样的语句, `b` 最终等于多少呢? 作者试验得到的结果是 11。

其中的原因难以理解, 与其去思考其中的原因, 我们还不如将上面的第二句改为几条语句来实现我们想要的结果。我可以给大家一个宝贵的经验, 不要在一行中编写太复杂的表达式, 也就是不要在一行中进行太多的运算, 在一行中进行太多的运算并不能为你带来什么好处, 相反只能带来坏处, 它并不比改成几条语句的运行速度快, 它除可读性差外, 还极容易出错。

对于这些优先级的顺序, 读者不用刻意去记, 有个印象就行。如果你实在弄不清这些运算先后关系的话, 就用括号或是分成多条语句来完成你想要的功能, 因为括号的优先级是最高的, 多使用括号能增加程序的可读性, 也是一种良好的编程习惯, 这也是软件编码规范的一个要求。

## 2.5 程序的流程控制

从结构化程序设计角度出发，程序有三种结构：

- | 顺序结构
- | 选择结构
- | 循环结构

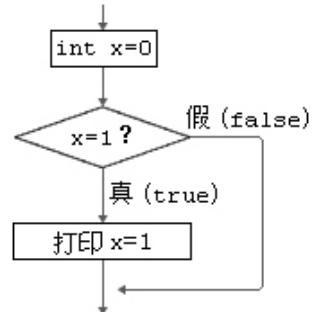
### 2.5.1 顺序结构

顾名思义，顺序结构就是程序从上到下一行一行执行的结构，中间没有判断和跳转，直到程序结束。

### 2.5.2 if 选择语句

第一种应用：

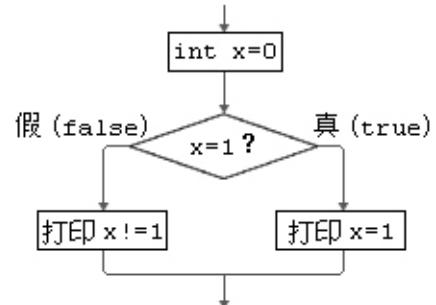
if 语句是条件判断语句，下面的代码是 if 语句的一种形式：  
int x=0;  
if (x==1)  
 System.out.println("x=1");  
如果 x 的值等于 1 则打印出 “x=1” ，否则什么也不做。



第二种应用：

下面的代码是 if 语句的另外一种形式，其中包括了 else 关键字：

```
int x=0;  
if(x==1)  
    System.out.println("X=1");  
else  
    System.out.println("X!=1");  
如果 x 的值等于 1 则打印出 “x=1” ，否则将打印出  
“x!=1” 。
```



### & 多学两招：

还有一种更专业的写法：变量 = 布尔表达式？语句 1:语句 2;

我们看一下下面的代码：

```
if(x>0)  
    y=x;  
else  
    y=-x;
```

这段代码也可以简写成下面的形式：

```
y = x>0?x:-x;
```

这是一个求绝对值的语句，如果  $x > 0$ ，就把  $x$  赋值给变量  $y$ ，如果  $x$  不大于  $0$ ，就把  $-x$  赋值给前面的  $y$ 。也就是：如果问号 “?” 前的表达式为真，则计算问号和冒号中间的表达式，并把结果赋

值给变量 y，否则将计算冒号后面的表达式，并把结果赋值给变量 y，这样的写法在 C 语言中经常用到，好处在于代码简洁，并且有一个返回值。

### 第三种应用：

if 语句中还可以有多个语句的情况：

```
int x=0;
if(x==1)
{
    System.out.println("X=1");
    System.out.println("Yes");
}
else
{
    System.out.println("X!=1");
    System.out.println("No");
}
```

由此可以看出，多个语句必须用大括号括起来形成一个复合语句。

### 第四种应用：

我们还可以用 else if 进行更多的选择，下面是使用 else if 语句的情况。

```
if (x==1)
    System.out.println("X=1");
else if (x==2)
    System.out.println("X=2");
else if (x==3)
    System.out.println("X=3");
else
    System.out.println("other");
```

程序首先判断 x 是否等于 1，如果是，就执行打印“x=1”，如果不是，程序将继续判断 x 是否等于 2，如果 x 等于 2，则打印“x=2”，如果也不等于 2，程序将判断 x 是否等于 3，如果是，则打印“x=3”，如果还不等于，就执行 else 后的语句。也可以不要 else 语句，那就是上面的条件都不满足时，就什么也不做。

### 第五种应用：

if 语句还可以嵌套使用，如：

```
if (x == 1)
    if(y == 1)
        System.out.println("x = 1,y = 1");
    else
        System.out.println("x = 1,y != 1");
else
    if(y == 1)
        System.out.println("x != 1,y = 1");
    else
        System.out.println("x != 1,y != 1");
```

在使用 if 嵌套语句时，最好使用 {} 来确定相互的层次关系，如下面的语句：

```
if (x == 1)
```

```

if(y == 1)
    System.out.println("x = 1,y = 1");
else
    System.out.println("x = 1,y != 1");
else if(x != 1)
    if(y == 1)
        System.out.println("x != 1,y = 1");
    else
        System.out.println("x != 1,y != 1");

```

我们很难判定最后的 `else` 语句到底属于哪一层的，编译器是不能根据书写格式来判定的，我们可以使用`{}`来加以明确。

```

if (x == 1)
{
    if(y == 1)
        System.out.println("x = 1,y = 1");
    else
        System.out.println("x = 1,y != 1");
}
else if(x != 1)
{
    if(y == 1)
        System.out.println("x != 1,y = 1");
    else
        System.out.println("x != 1,y != 1");
}

```

或者改为下面的格式，来表达另外的一种意思。

```

if (x == 1)
{
    if(y == 1)
        System.out.println("x = 1,y = 1");
    else
        System.out.println("x = 1,y != 1");
}
else if(x != 1)
{
    if(y == 1)
        System.out.println("x != 1,y = 1");
}
else
    System.out.println("x != 1,y != 1");

```

在 Java 中，`if()` 和 `else if()` 括号中的表达式的结果必须是布尔型的（即 `true` 或者 `false`），这一点和 C、C++ 不一样。

### 2.5.3 switch 选择语句

我们可以把上面含有两个 `else if` 语句的代码写成 `switch` 语句的格式：

```
int x=2;  
switch(x)  
{  
    case 1:  
        System.out.println("x=1");  
        break;  
    case 2:  
        System.out.println("x=2");  
        break;  
    case 3:  
        System.out.println("x=3");  
        break;  
    default:  
        System.out.println("Sorry,I don't Know X.");  
}
```

程序结果：

**x=2**

上面代码中，`default` 语句是可选的，它接受除上面接受值以外的其他值，通俗的讲，就是谁也不要的都归它。`switch` 语句判断条件可以接受 `int, byte, char, short` 型，不可以接受其他类型。

注意，不要混淆 `case` 与 `else if`。`else if` 是一旦匹配就不在执行后面的 `else` 语句，而 `switch` 一旦碰到第一次 `case` 匹配，就会开始顺序执行以后所有的程序代码，而不管后面的 `case` 条件是否匹配，后面 `case` 条件下的所有代码都将被执行，直到碰到 `break` 语句为止。所以，如果不写 `break` 语句，例如删除 `System.out.print("x=2")` 后面的 `break` 语句，程序运行的结果将是：

**x=2**

**x=3**

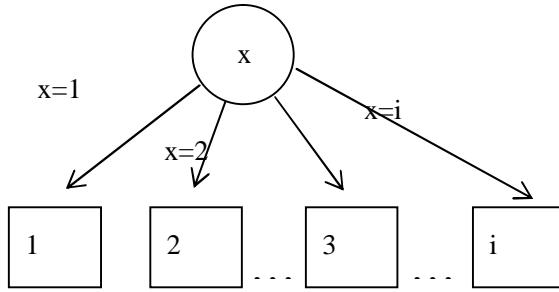
所以一定要记住用 `break` 退出 `switch`。`case` 后面可以跟多个语句，这些语句可以不用大括号括起来，如果你喜欢，非要将多个语句用大括号括起来当然也可以。

思考题：用同一段语句来处理多个 `case` 条件，程序该如何编写？答案如下：

```
case 1:  
case 2:  
case 3:  
    System.out.println("you are very bad");  
    System.out.println("you must make great efforts");  
    break;  
case 4:  
case 5:  
    System.out.println("you are good");
```

不要死记硬背，要从原理上去思考，还记得前面刚才讲的“`case` 是一旦碰到第一次匹配，如果没有 `break`，就会继续执行”这个原理吗？

## 2.5.4 while 循环语句

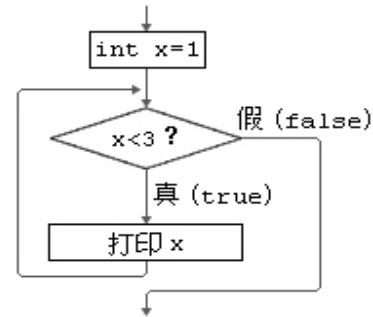


`while` 语句是循环语句，也是条件判断语句，条件满足时执行，不满足时退出，请看下面代码：

```
int x=1;
while(x<3)
{
    System.out.println("x="+x);
    x++;
}
```

运行结果如下：

```
x=1
x=2
```



**M**脚下留心：`while` 表达式的括号后一定不要加“；”，如：

```
int x=3;
while(x==3);
System.out.println("x=3");
```

这是初学者常犯的一个毛病，程序将认为要执行一条空语句，而进入无限循环，永远不去执行后面的代码，Java 编译器又不会报错。所以你可能要浪费许多时间来调试也不知道错在哪。

### do while 语句

我们将上面 `while` 循环的代码改成：

```
int x=1;
do
{
    System.out.println("x="+x);
    x++;
}
while(x<3);
```

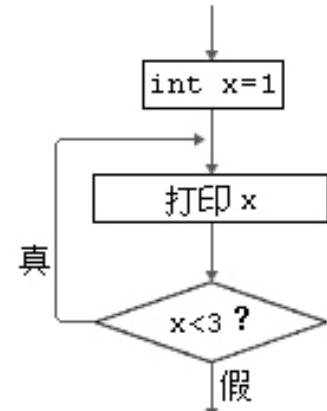
程序运行结果如下：

```
x=1
x=2
```

我们发现这两次的运行结果是一样的，区别只是 `do while` 的判断语句在结尾，所以中间代码是无论条件是否成立都要至少执行一次的。下面例子的 `while` 循环里面的代码就没有机会执行了，但尽管条件不成立，`do while` 循环中的代码还是执行了一次：

程序清单：`TestDo.java`

```
public class TestDo
{
    public static void main(String[] args)
    {
        int x=3;
        while(x==0)
        {
            System.out.println("ok1");
        }
    }
}
```



```

        x++;
    }
    int y=3;
    do
    {
        System.out.println("ok2");
        y++;
    }
    while(y==0);
}
}

```

程序运行结果如下：

```
ok2
```

## 2.5.5 for 循环语句

先看一个例子：

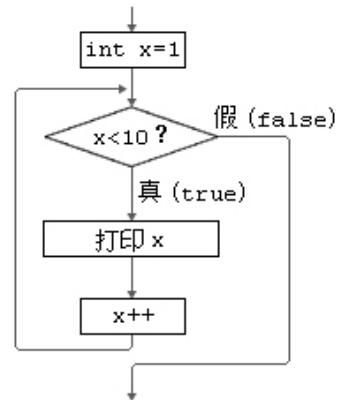
```

for(int x=1;x<10;x++)
{
    System.out.println("x="+x);
}

```

程序运行结果如下：

```
x=1
x=2
...
x=9
```



在这里，我们介绍一下 `for` 语句后面小括号中的部分，这部分内容又被“;”隔离成三部分，其中第一部分 `x=1` 是给 `x` 赋一个初值，只在刚进入 `for` 时执行一次；第二部分 `x<3` 是一个条件语句，满足就进入 `for` 循环，循环执行一次后又回来执行这条语句，直到条件不成立为止；第三部分 `x+=1` 是对变量 `x` 的操作，在每次循环的末尾执行，读者可以把 `x+=1` 分别换成 `x+=2` 和 `x-=2` 来试验一下每次加 2 和每次减 2 的情景。

如上所述，上面的代码可以改写为：

```

int x=1;
for(;x<10;)
{
    System.out.println("x="+x);
    x++;
}

```

通过这样改写，读者应该能够更好地理解 `for` 后面小括号中的三部分的各自作用了。

## & 多学两招：

`for` 语句还有下面一种特定的语法格式：

```
for(;;)
```

```
.....  
}
```

同样意义的还有 `while(true)`, 这些都是无限循环, 需要用 `break` 语句跳出循环, 我们在以后的编程中都会遇到。

例如上面的代码又可以改为:

```
for(;;)  
{  
    if(x<10)  
        break;  
    x++;  
}
```

## 2.5.6 `break` 与 `continue` 语句

### `break` 语句

`break` 语句可以中止循环中的子语句和 `switch` 语句。一个无标号的 `break` 语句会把控制传给当前(最内)循环(`while`, `do`, `for` 或 `Switch`)的下一条语句。如果有标号, 控制会被传递给当前方法中的带有这一标号的语句。如:

```
st:while(true)  
{  
    while(true)  
    {  
        break st;  
    }  
}
```

执行完 `break st;` 语句后, 程序会跳出外面的 `while` 循环, 如果不使用 `st` 标号, 程序只会跳出里面的 `while` 循环。

### `continue` 语句

`continue` 语句只能出现在循环语句(`while`, `do`, `for`)的子语句块中, 无标号的 `continue` 语句的作用是跳过当前循环的剩余语句块, 接着执行下一次循环。

请看下面打印 1 到 10 之间的所有奇数的例子, 当 `i` 是偶数时就跳过本次循环后的代码, 直接执行 `for` 语句中的第三部分, 然后进入下一次循环的比较, 是奇数就打印 `i`:

程序清单: `PrintOddNum.java`

```
public class PrintOddNum  
{  
    public static void main(String [] args)  
    {  
        for(int i=0;i<10;i++)  
        {  
            if(i%2==0)  
                continue;  
            System.out.println(i);  
        }  
    }  
}
```

}

## 2.6 数组

### 2.6.1 数组的基本概念

我们还是通过一个应用问题来引入我们对数组的讲解，来让大家了解什么是数组以及数组的作用。假设你需要在程序中定义一百个整数变量，并要求出这些变量相加的结果，如果你没有用过数组，这个程序该怎么写呢？至少我们需要定义一百个整数变量，如下所示：

```
int x0;  
int x1;  
.....  
.....  
int x98;  
int x99;
```

一下就要定义 100 个相似的变量，然后还要将这些变量一个一个的相加，这是一件令人畏惧的事情。有什么简单的方法来替代上述变量的定义方式呢？

我们可以将上面的定义改写为：

```
int x[] = new int[100];
```

上述语句的意义相当于一下子就定义了一百个 int 变量，变量的名称分别为 x[0], x[1], ..., x[98], x[99]。注意，第一个变量名为 x[0]，而不是 x[1]，最后一个变量名为 x[99]，而不是 x[100]，这种定义变量的方式就是数组。定义了这个数组，我们接着就可以使用简单的 for 循环语句来实现数组中的所有元素的相加了，程序代码如下：

```
int sum=0;  
for(int i=0;i<100;i++)  
    sum += x[i];
```

**& 多学两招：**为了充分和深入了解数组，我必须向大家讲解有关内存分配的一些背后知识。Java 把内存划分成了两种：一种是栈内存，另一种是堆内存。

我们在函数中定义的一些基本类型的变量和对象的引用变量都是在函数的栈内存中分配，当我们在一段代码块（也就是一对花括号 {} 之间）定义一个变量时，Java 就在栈中为这个变量分配内存空间，当超过变量的作用域后，Java 会自动释放掉为该变量所分配的内存空间，该内存空间可以立即被另作他用，我们先前讲到的知识都属于栈中分配的变量。

堆内存用来存放由 new 创建的对象和数组，在堆中分配的内存，由 Java 虚拟机的自动垃圾回收器来管理（关于自动垃圾回收器请参看第一章中的垃圾回收器介绍）。在堆中产生了一个数组或对象后，我们还可以在栈中定义一个特殊的变量，让栈中的这个变量的取值等于数组或对象在堆内存中的首地址，栈中的这个变量就成了数组或对象的引用变量，我们以后就可以在程序中使用栈中的引用变量来访问堆中的数组或对象，引用变量就相当于是我们为数组或对象起的一个名称（叫代号也行）。引用变量是普通的变量，定义时在栈中分配，引用变量在程序运行到其作用域之外后被释放。而数组和对象本身在堆中分配，即使程序运行到使用 new 产生数组和对象的语句所在的代码块之外，数组和对象本身占据的内存不会被释放，数组和对象在没有引用变量指向它时，才会变为垃圾，不能再被使用，但仍然占据内存空间不放，在随后一个不确定的时间被垃圾回收器收走（释放掉）。这也是 Java 比较吃内存的原因。

数组是多个相同类型数据的组合，实现对这些数据的统一管理，数组中的每一个数据也叫数组

的一个元素。我们来解释下面这句代码的语法。

```
int x[] = new int[100];
```

等号左边的 `int x[]` 相当于定义了一个特殊的变量符号 `x`, `x` 的数据类型是一个对 `int` 型的数组对象的引用, `x` 就是一个数组的引用变量, 其引用的数组的元素个数不定, 就象我们定义一个基本类型的变量, 变量值开始也是不确定的。等号右边的 `new int[100]` 就是在堆内存中创建一个具有 100 个 `int` 变量的数组对象。`int x[] = new int[100];` 就是将右边的数组对象赋值给左边的数组引用变量。因此, 我们也可以将这一行代码分成两行来写。

```
int x[]; // 定义了一个数组 x, 这条语句执行完后的内存状态如图 2.8 所示。
```

```
x=new int[100]; // 数组初始化, 这条语句执行完后的内存状态如图 2.9 所示。
```

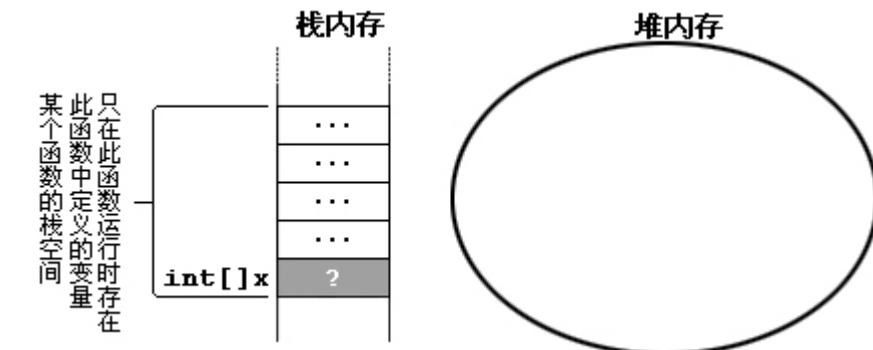


图 2.8

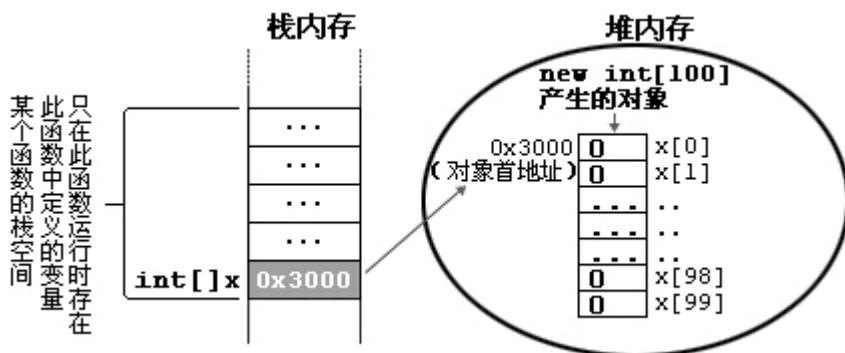


图 2.9

执行第二句 (`x=new int[100];`) , 在堆里面创建了一个数组对象, 为这个数组对象分配了 100 个整数单元, 并将数组对象赋值给了数组引用变量 `x`。精通 C 语言的读者可能已经明白了, 数组引用变量不就是 C 语言中的指针变量吗? 数组对象不就是指针变量要指向的那个内存块吗? 是的, Java 内部还是有指针的, 只是把指针的概念对用户隐藏起来了。

我们也可以改变 `x` 的值, 让它指向另外一个数组对象, 或者不指向任何数组对象。要想让 `x` 不指向任何数组对象, 我们只需要将常量 `null` 赋值给 `x` 即可。如: `x=null;` , 这条语句执行完后的内存状态如图 2.10 所示。

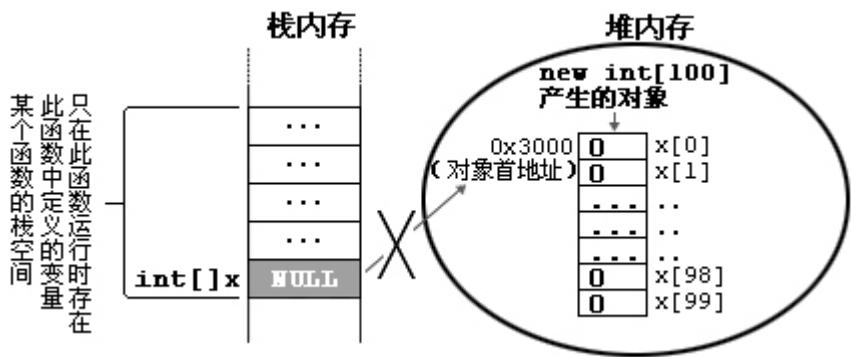


图 2.10

执行完 `x=null;` 语句后，原来通过 `new int[100]` 产生的数组对象不再被任何引用变量所引用，变成了“孤儿”，也就成了垃圾，直到垃圾回收器来将它释放掉。

`new int[100]` 产生的数组对象中的每个元素的初始值都是 0，读者可以用下面的程序来测试一下：

程序清单：TestArray.java

```
public class TestArray
{
    public static void main(String [] args)
    {
        int x[];
        x=new int[100];
        for(int i=0;i<100;i++)
        {
            System.out.println("x"+i+" is "+x[i]);
        }
    }
}
```

在 Java 中，我们还可以用下面的方式定义数组：

`int [] x;` // 方括号 ([ ]) 位于变量名之前。

`x=new int[100];`

两种定义数组方式的效果是完全一样的，差别只是人们的使用习惯不同罢了。

## 2.6.2 数组的静态初始化

我们也可以在定义数组的同时就为数组元素分配空间并赋值，也就是对数组的静态初始化。如这一句：

`int ia[] ={1,2,3,4};`

等号右边相当于产生了一个数组对象，该数组有 4 个元素变量，这 4 个变量的取值分别是整数 1、2、3、4。数组的长度等于右边 {} 中的元素的个数。我们有时也能见到下面定义数组的方式：

`int ia[]={1,2,3,4,5};`

**注意：在 Java 语言中声明数组时，无论用何种方式定义数组，都不能指定其长度，例如下面的定义将是非法的。**

`int a[5]; // 编译时将出错`

### 2.6.3 使用数组时要注意的问题

必须对数组引用变量赋予一个有效的数组对象(通过 new 产生或是用{}静态初始化而产生)后,才可以引用数组中的每个元素,下面的代码将会导致运行时出错,如图 2.11。

```
public class TestArray
{
    public static void main(String [] args)
    {
        int a[] = null;
        a[0]=1;
        System.out.println(a[0]);
    }
}
```



图 2.11

上面的错误告诉我们,运行时会有空指针异常错误(NullPointerException,关于异常的知识我们放在后面的章节讲解),因为 a 还没有指向任何数组对象(相当于 c 语言中的指针还没有指向任何内存块),所以还无法引用其中的元素。

还有一点请初学者注意,譬如说我们下面的代码:

```
int ia[] = new int[]{1,2,3,4,5};
```

这行代码中我们定义了一个 ia 数组,它里面包含了 5 个元素,它们分别是:

```
ia[0]=1  
ia[1]=2  
ia[2]=3  
ia[3]=4  
ia[4]=5
```

也就是说数组的第一个元素是 ia[0],而不是 ia[1]。最后一个元素是 ia[4],而不是 ia[5],如果我们不小心使用了 ia[5],如下面的程序:

程序清单: TestArray.java

```
public class TestArray
{
    public static void main(String [] args)
    {
        int ia[] = new int[]{1,2,3,4,5};
        System.out.println(ia[5]);
    }
}
```

就会发生“数组越界异常 (ArrayIndexOutOfBoundsException)”，如图 2.12。读者以后必须学会根据程序所报出的异常来判断究竟出了什么错误，并且看到这样的错误，就应该明白错误的原因所在了。

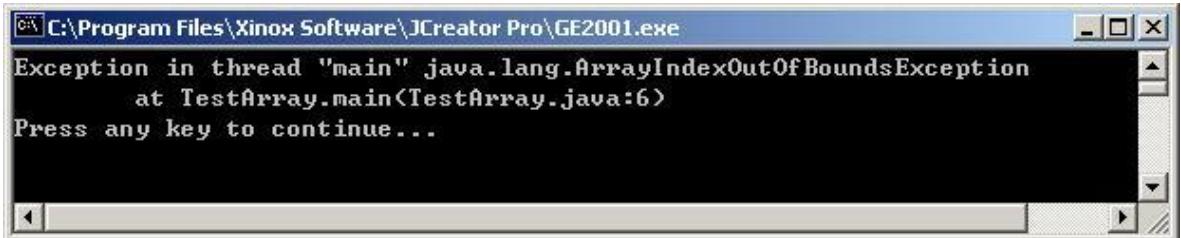


图 2.12

要想避免“数组越界异常”这样的错误，我们必须要知道数组长度。数组引用对象的 `length` 属性可以返回数组的长度。示例程序代码如下：

```
public class TestArrayLength
{
    public static void main(String [] args)
    {
        int ia[] = new int[]{1,2,3,4,5};
        System.out.println(ia.length);
        for(int i=0; i<ia.length; i++)
        {
            System.out.println("ia["+i+"] is "+ia[i]);
        }
    }
}
```

#### 2.6.4 多维数组

在 Java 中并没有真正的多维数组，只有数组的数组，虽然在应用上很象 C 语言中的多维数组，但还是有区别的。在 C 语言中定义一个二维数组，必须是一个  $x \times y$  二维矩阵块，类似我们通常所见到的棋盘，如图 2.13 所示：

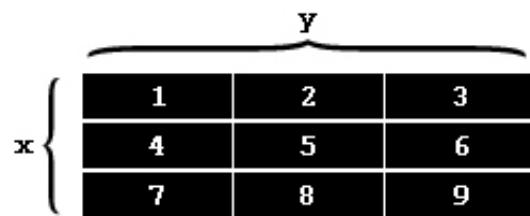


图 2.13

Java 中多维数组不一定是规则矩阵形式，如图 2.14 所示：

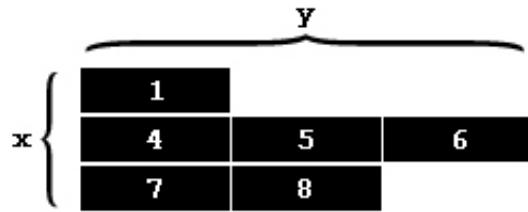


图 2.14

我们这样定义一个二维数组：

```
int xx[][];
```

表示定义了一个数组引用变量 xx，第一个元素变量为 xx[0]，第 n 个元素变量为 xx[n-1]。xx 中的每个元素变量（xx[0] 到 xx[n-1]）正好又是一个整数类型的数组引用变量，注意，这里只是要求每个元素都是一个数组引用变量，并没有要求它们所引用数组的长度是多少，也就是每个引用数组的长度可以不一样，我们还是看看下面的程序代码。

```
int[][] xx;  
xx=new int[3][];
```

这两句代码表示数组 xx 有三个元素，每个元素都是 int [] 类型的一维数组。相当于定义了三个数组引用变量，分别为 int xx[0][], int [] xx[1], int [] xx[2]。你完全可以把 xx[0] 想成是一个普通的变量名，只是正好是 x、x、[、0、]这五个字母的组合罢了，另外，作者是故意对 int xx[0][], int [] xx[1] 这两个数组引用变量用不一样的方式书写，如果你还记得前面讲过的一维数组的几种表示方式，就知道作者想达到一箭双雕的目的，既帮助读者巩固了前面所学的知识，又便于读者通过对比而真正理解作者“把 xx[1] 想成是一个普通的变量名”的思想。

由于 xx[0], xx[1], xx[2] 都是数组引用变量，必须对它们赋值，指向真正的数组对象，才可以引用这些数组中的元素。

```
xx[0]=new int[3];  
xx[1]=new int[2];
```

注意，xx[0] 和 xx[1] 的长度可以不一样，数组对象中也可以只有一个元素。程序运行到这之后的内存分配情况如图 2.15 所示：

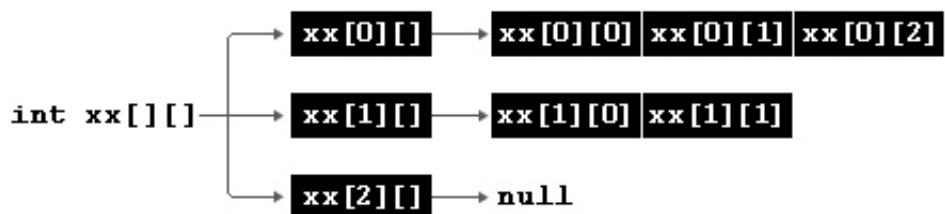


图 2.15

xx[0] 中的第二个元素用 xx[0][1] 来表示，如果我们要将整数 5 赋值给 xx[0] 中的第二个元素，写法如下：

```
xx[0][1] = 5;
```

如果数组对象正好是一个  $x \times y$  形式的规则矩阵，我们不必向上面的程序一样，先产生高维的数组对象后，再逐一产生低维的数组对象，完全可以用一句代码在产生高维数组对象的同时，产生所有的低维数组对象。

```
int xx[][]=new int[2][3];
```

上面的代码产生了一个  $2 \times 3$  形式的二维数组，其内存布局如图 2.16：

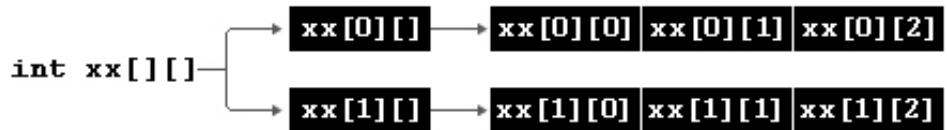


图 2.16

我们也可以象一维数组一样，在定义数组的同时就为多维数组元素分配空间并赋值，也就是对多维数组的静态初始化。如下面这句代码

```
int[][] xx={{3,2,7},{1,5},{6}};
```

定义了一个如图 2.17 中的多维数组：



图 2.17

与一维数组一样，在声明多维数组时不能指定其长度，例如下面的定义将是非法的。

```
int xx[3][2] = {{3,2},{1,3},{7,5}}; //编译出错
```

## 2.6.5 一些与数组操作相关的函数

既然学到数组，我们就介绍两个与数组操作相关的函数：

1. 使用 System.arraycopy() 函数拷贝数组。方法如下：

```
System.arraycopy(source, 0, dest, 0, x);
```

这行代码的意思就是：复制源数组中从下标 0 开始的 x 个元素到目的数组，从目标数组的下标 0 所对应的位置开始存储。我们可以从 [JDK 文档中了解到 arraycopy 更详细的使用说明](#)，如图 2.18 所示：

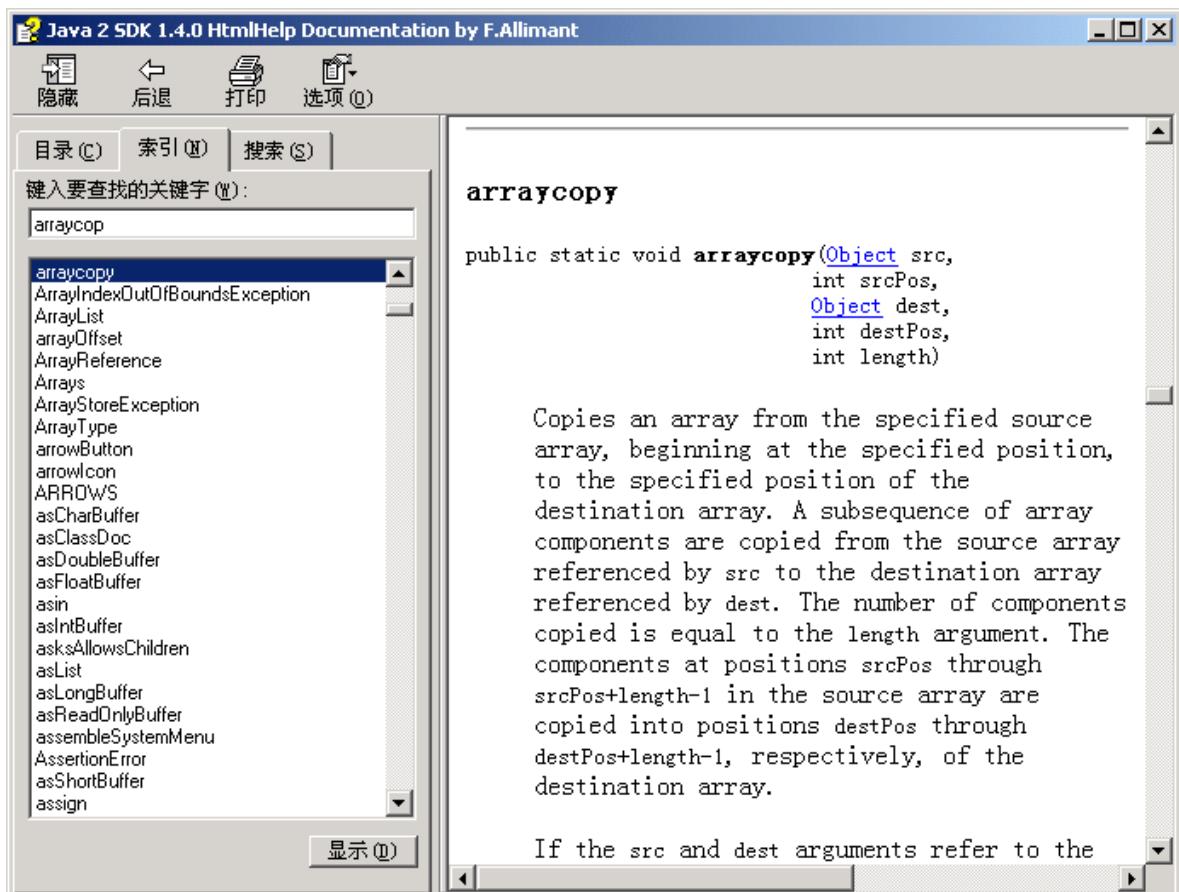


图 2.18

程序清单：TestArrayCopy.java

```
public class TestArrayCopy
{
    public static void main(String [] args)
    {
        int ia[] = new int[]{1,2,3,4,5};
        int ib[] = new int[]{9,8,7,6,5,4,3};

        System.arraycopy(ia,0,ib,0,3);
        // 复制源数组中从下标 0 开始的 3 个元素到目的数组，从下标 0 的位置开始存储。
        for(int i=0;i<ia.length;i++)
            System.out.print(ia[i]);
        System.out.println();

        for(int j=0;j<ib.length;j++)
            System.out.print(ib[j]);
        System.out.println();
    }
}
```

注意：你复制的数组元素的个数一定不要超过目的数组的长度。否则会有异常产生。

2. 用 `Arrays.sort` 来排序数组，把里面的元素按从小到大的顺序逐一排列，然后把排序后的数组输出在命令行窗口上：

程序清单：`ArrSort.java`

```
import java.util.*;  
//关于上面这条语句的细节，读者暂且不用理会，在后面的章节中会有详细的讲解。  
public class ArrSort  
{  
    public static void main(String [] args)  
    {  
        int ia[]={1,2,4,8,3};  
  
        Arrays.sort(ia); //对数组 ia 进行排序  
  
        for(int i=0;i<ia.length;i++)  
            System.out.print(ia[i]);//输出数组 ia  
    }  
}
```

程序运行结果是：

**12348**

关于 `Arrays.sort` 方法的帮助信息，读者现在应该可以自己动手查阅 JDK 文档解决了吧！

|                             |    |
|-----------------------------|----|
| 第 2 章 Java 编程基础 .....       | 24 |
| 2.1 Java 基本语法格式 .....       | 24 |
| 2.1.1 Java 代码的落脚点 .....     | 24 |
| 2.1.2 Java 是严格区分大小写的 .....  | 24 |
| 2.1.3 Java 是一种自由格式的语言 ..... | 24 |
| 脚下留心： 1.字符串的断行问题            |    |
| 2. 分号(;)问题                  |    |
| 2.1.4 java 程序的注释 .....      | 24 |
| 脚下留心： 程序注释中应注意的一些问题         |    |
| 不得不说： 养成良好的编程风格             |    |
| 2.1.5 Java 中的标识符 .....      | 25 |
| 2.1.6 Java 的关键字 .....       | 26 |
| 2.1.7 Java 中的常量 .....       | 26 |
| 多学两招： 特殊字符的表示               |    |
| 2.2 变量及变量的作用域 .....         | 27 |
| 2.2.1 变量的概念 .....           | 27 |
| 2.2.2 Java 的变量类型 .....      | 28 |
| 独家见解： 数值所代表的意义              |    |
| 2.2.3 注意变量的有效取值范围 .....     | 29 |
| 2.2.4 基本数据类型之间的转换 .....     | 29 |
| 独家见解： 轻松理解类型转换              |    |
| 2.2.5 表达式的数据类型自动提升 .....    | 31 |
| 多学两招： 字符串中的加号（+）问题          |    |

|                                  |    |
|----------------------------------|----|
| 2.2.6 变量的作用域.....                | 32 |
| 脚下留心：Java 与 C/C++的区别             |    |
| 2.2.7 局部变量的初始化.....              | 33 |
| 2.3 函数与函数的重载.....                | 33 |
| 2.3.1 函数 .....                   | 33 |
| 独家见解：如何理解函数返回值类型为 void 的情况       |    |
| 2.3.2 函数的参数传递过程.....             | 37 |
| 多学两招：形式参数的作用                     |    |
| 2.3.3 函数的重载.....                 | 38 |
| 2.4 Java 中的运算符 .....             | 39 |
| 2.4.1 算术运算符.....                 | 39 |
| 脚下留心：1.注意++a 和 a++以及--a 和 a--的区别 |    |
| 2.对负数取模的问题                       |    |
| 3.的整数除和小数除的区别                    |    |
| 多学两招：两个经典算法                      |    |
| 2.4.2 赋值运算符.....                 | 40 |
| 2.4.3 比较运算符.....                 | 41 |
| 脚下留心：“==” 运算符的注意事项               |    |
| 2.4.4 逻辑运算符.....                 | 41 |
| 2.4.5 位运算符.....                  | 42 |
| 多学两招：移位的特殊功能                     |    |
| 脚下留心：使用移位运算符应注意的事项               |    |
| 2.4.6 运算符的优先级.....               | 45 |
| 2.5 程序的流程控制.....                 | 46 |
| 2.5.1 顺序结构.....                  | 46 |
| 2.5.2 if 选择语句 .....              | 46 |
| 多学两招：一种更简单的写法                    |    |
| 2.5.3 switch 选择语句 .....          | 48 |
| 2.5.4 while 循环语句.....            | 49 |
| 脚下留心：while 表达式的注意事项              |    |
| 2.5.5 for 循环语句.....              | 51 |
| 多学两招：无限循环                        |    |
| 2.5.6 break 与 continue 语句 .....  | 52 |
| 2.6 数组 .....                     | 53 |
| 2.6.1 数组的基本概念.....               | 53 |
| 多学两招：内存分配的奥秘                     |    |
| 2.6.2 数组的静态初始化.....              | 55 |
| 2.6.3 使用数组时要注意的问题.....           | 56 |
| 2.6.4 多维数组.....                  | 57 |
| 2.6.5 一些与数组操作相关的函数.....          | 59 |

# 第3章 面向对象（上）

## 3.1 面向对象的概念

面向对象是令大多数人都只可意会，难以言宣的大概念。我下面的讲解主要是冲着帮你迅速理解面向对象的内涵而来的，不见得非常严谨和完全正确，但我个人认为是合理的，是我的体会和认识，我也曾经用过同样的方式，让许多人仿佛一下子就明白了什么是面向对象。什么是面向对象，这是一个相对概念，是相对面向过程而言的。就象要理解什么是幸福，要先理解什么是痛苦一样的道理，在讲解面向对象之前，我要先简单介绍一下什么是面向过程。

### 3.1.1 面向过程

在早期出现的编程语言中，如大家都非常熟悉的 C 语言，当我们要用这种语言来定义一个复杂的数据类型，譬如 Windows 窗口时，可以用结构体（struct）来实现，只要在结构体中使用那些基本的数据类型来定义窗口的大小，位置，颜色，背景等属性就可以了。如果我们要对一个 Windows 窗口进行各种操作，如隐藏窗口，移动窗口，最小化窗口等等，我们要为每个操作都定义一个函数，这些函数与窗口本身的定义没有任何关系，如 HideWindow, MoveWindow, MinimizeWindow，这些函数都需要接受一个参数，即要操作的窗口。这是一种谓语与宾语的关系，“隐藏”、“移动”等是谓语，“窗口”是宾语。程序的重心集中在函数（即谓语）上。

### 3.1.2 面向对象

在 C++, java 语言中，我们可以将一个窗口当作一个主体（对象）来看，定义窗口时，除了要指定在面向过程中规定的那些属性，如大小，位置，颜色，背景等外，还要指定该窗口可能具有的动作，如隐藏，移动，最小化等。我们在定义窗口时，就要定义好对应这些动作的函数（也叫方法），如 Hide, Move, Minimize，注意体会这些函数名称与上面的名称的区别，这是作者刻意这么安排的，让读者更好地去对比，从函数名称上就能看出，这些函数都不再接受代表窗口的参数。这些函数被调用时，都是以某个窗口要隐藏，某个窗口要移动，某个窗口要最小化的语法格式来使用的。这是一种主语与谓语的关系，程序的重点集中在主体/对象（主语）上。虽然读者在下面的讲解中能够发现面向对象的知识远不止这一点（刚才讲的只是面向对象的一个方面，即封装性，用类封装了数据与函数），真正能体现面向对象的强大优势的地方，是在面向对象的继承与多态性方面，但作者认为，封装性是面向对象的根源和最根本的属性。

Java 类同其他面向对象的编程语言一样，也支持面向对象(OOP)的三个特征：

- 封装（Encapsulation）
- 继承（Inheritance）
- 多态（Polymorphism）

要想领悟面向对象的思想，不能把学习重点放在术语的死记硬背上，形而上学，而应该把精力主要放在实践和思考上，通过大量实践去理解和掌握。如果读者到现在还不能完全明白面向对象是什么，也没关系，学完下面的内容后，再回过头来看吧！

## 3.2 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面目尽可能的一致，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 3.1 所示就是一个典型的说明：

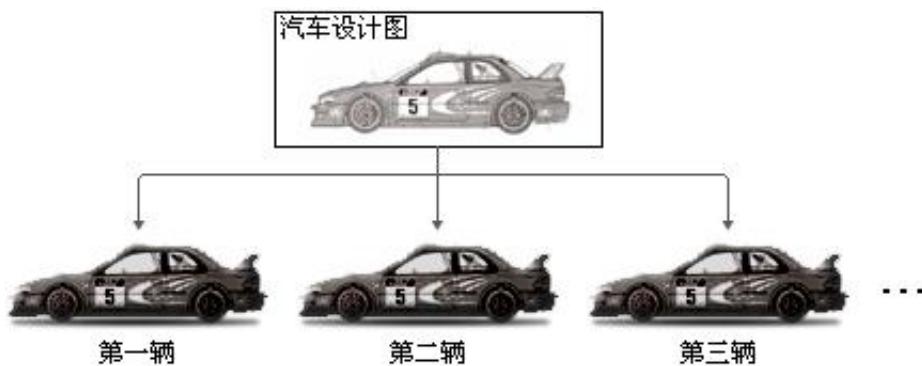


图 3.1

其中，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、图纸。对象（Object）是类（Class）的一个实例（Instance），**是一个实实在在的个体，一个类可以对应多个对象。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多个对象，刚开始的状态都应该是一样的，好比按照“奔驰 s600”型设计图纸生产出来的汽车刚开始都是一样的，其中一辆“奔驰 s600”汽车被改装后，是不会影响到同型号的其他“奔驰 s600”汽车的。但如果修改了“奔驰 s600”型的设计图纸，就会影响到以后所有出厂的“奔驰 s600”汽车。

### 3.2.1 类的定义

类可以将数据和函数封装在一起，**其中数据表示类的属性，函数表示类的行为。**定义类就是要定义类的属性与行为（方法）。请看这段代码：

```
class Person
{
    int age;
    void shout()
    {
        System.out.println("oh,my god! my age is " + age);
    }
}
```

其中，定义了一个 Person 类，该类有一个属性 age，一个方法 shout。类的属性也叫类成员变量，类的方法也叫类的成员函数。一个类中的方法可以直接访问同类中的任何成员（包括成员变量和成员函数），如 shout 方法可以直接访问同一个类中的 age 变量。

**M**脚下留心：如果一个方法中有与成员变量同名的局部变量，该方法中对这个变量名的访问是局部变量，而不再是成员变量。如：

```
class Person
{
    int age; //这是一个成员变量
    void shout()
    {
        int age=60; //这是函数内部又重新定义的一个局部变量
        System.out.println("oh,my god! my age is " + age);
    }
}
```

在这里，shout 方法的 System.out.println(“oh,my god! my age is “ + age);语句所访问的 age 就不再是成员变量 age，而是在 shout 方法中定义的局部变量 age。

### 3.2.2 对象的产生与使用

光有设计图是无法实现汽车的功能的，只有产生了实际的汽车才行，同样的，要想实现类的属性和行为，必须创建具体的对象。

要创建新的对象，需要使用 new 关键字和想要创建对象的类名，如：

```
Person p1 = new Person();
```

等号左边以类名 Person 做为变量类型定义了一个变量 p1，来指向等号右边通过 new 关键字创建的一个 Person 类的实例对象，变量 p1 就是对象的引用句柄，对象的引用句柄是在栈中分配的一个变量，对象本身是在堆中分配的，原理同上一章讲过的数组是一样的。注意：在 new 语句的类名后一定要跟着一对圆括号 ()，在本章的稍后部分，读者就会明白这个括号的意义的。这条语句执行完后的内存状态如图 3.2 所示：

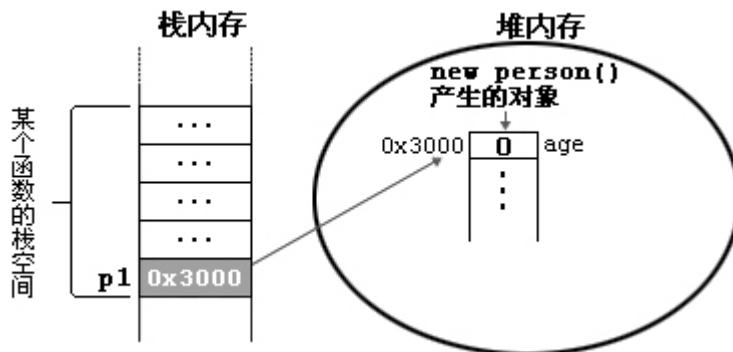


图 3.2

变量在被初始化之前是不能使用的，我们肯定还记得在一个方法内部的变量必须进行初始化赋值，否则编译无法通过的情况。当一个对象被创建时，会对其中各种类型的成员变量按表 3.1 自动进行初始化赋值。除了基本数据类型之外的变量类型都是引用类型，如上面的 Person 及前面讲过的数组。

| 成员变量类型                    | 初始值             |
|---------------------------|-----------------|
| <b>Byte</b>               | 0               |
| <b>Short</b>              | 0               |
| <b>Int</b>                | 0               |
| <b>long</b>               | 0L              |
| <b>float</b>              | 0.0F            |
| <b>double</b>             | 0.0D            |
| <b>char</b>               | '\u0000' (表示为空) |
| <b>boolean</b>            | False           |
| <b>All reference type</b> | Null            |

表 3.1

所以，我们看到对象内存状态图中的 age 成员变量的初始值为 0。

创建新的对象之后，我们就可以使用“对象名. 对象成员”的格式，来访问对象的成员（包括属性和方法），下面的程序代码演示了 Person 类对象的产生和使用方式。

```
class TestPerson
{
    public static void main(String[] args)
    {
        Person p1 = new Person();
        Person p2 = new Person();
        p1.age = -30;
        p1.shout();
        p2.shout();
    }
}
```

程序运行结果如下：

```
oh,my god! my age is -30
oh,my god! my age is 0
```

上面的程序代码，在 TestPerson.main 方法中创建了两个 Person 类的对象，并定义了两个 Person 类的对象引用句柄 p1、p2，分别指向这两个对象。接着，程序调用了 p1 和 p2 的方法和属性，p1、p2 是两个完全独立的对象，类中定义的成员变量，在每个对象都被单独实例化，不会被所有的对象共享，改变了 p1 的 age 属性，不会影响 p2 的 age 属性。调用某个对象的方法时，该方法内部所访问的成员变量，是这个对象自身的成员变量。上面程序运行的内存布局如图 3.3 所示：

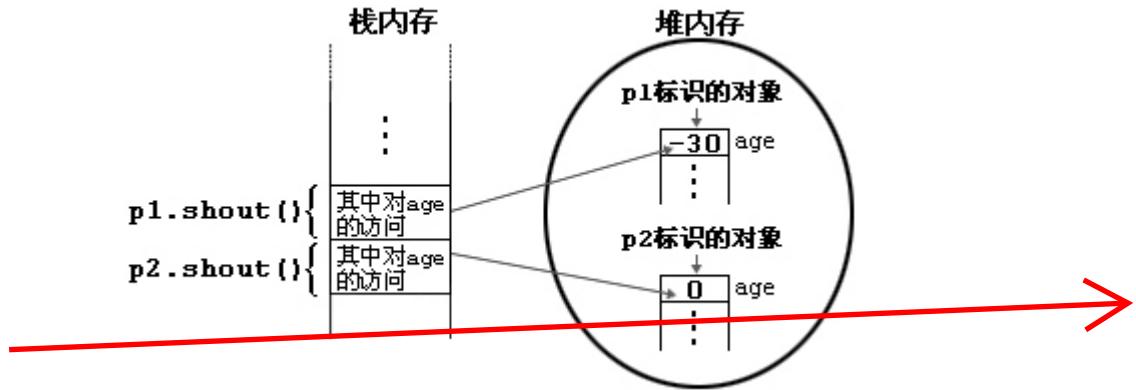


图 3.3

每个创建的对象都是有自己的生命周期的，对象只能在其有效的生命周期内被使用，当没有引用变量指向某个对象时，这个对象就会变成垃圾，不能再被使用。我们通过分析下面的几种程序代码，来了解对象何时会变成垃圾的情况。

第一种情况的程序代码：

```
{
    Person p1 = new Person();
    ...
}
```

程序执行完这个代码块后，也就是执行完这对大括号中的所有代码后，产生的 Person 对象就会变成垃圾，因为引用这个对象的句柄 p1 已超过其作用域，p1 已经无效，Person 对象就不再被任何句柄引用了。如图 3.4 所示：

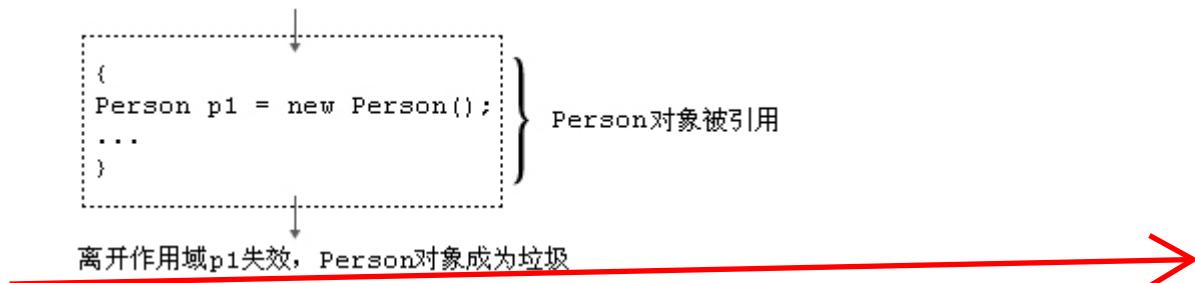


图 3.4

第二种情况的程序代码：

```
{
    Person p1 = new Person();
    p1 = null;
    ...
}
```

在执行完 `p1 = null;` 后，即使句柄 p1 还没有超出其作用域，仍然有效，但它已经被赋值为空，也就是说 p1 不再指向任何对象，这个 Person 对象也就成了孤儿，不再被任何句柄引用，变成了垃圾。如图 3.5 所示：

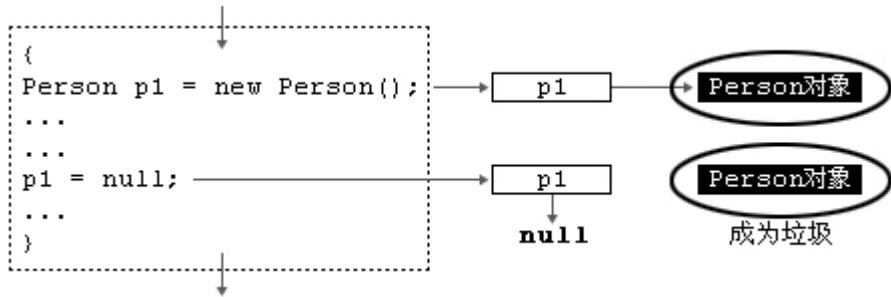


图 3.5

第三种情况的程序代码:

```

{
    Person p1 = new Person();
    Person p2 = p1;
    p1 = null;
    .....
}

```

执行完 `p1 = null;` 后，产生的 Person 对象不会变成垃圾，因为这个对象仍被 `p2` 所引用，直到 `p2` 超出其作用域而无效，产生的 Person 对象才会变成垃圾。如图 3.6 所示：

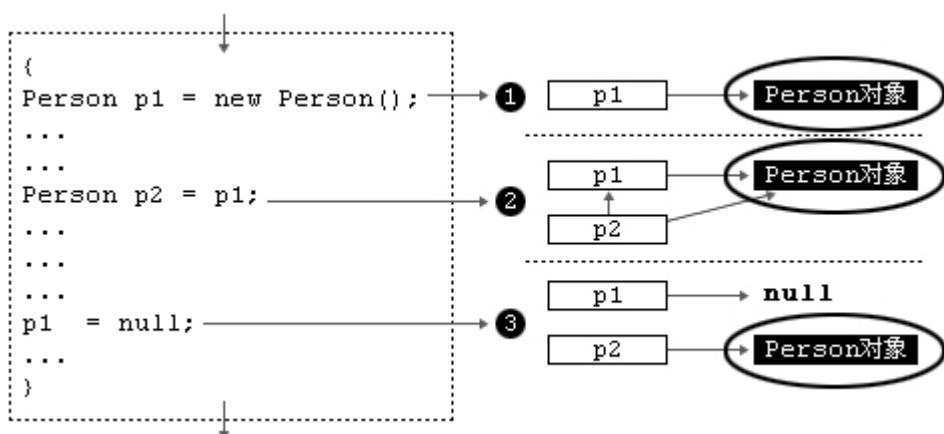


图 3.6

### 3.2.3 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变量的值是否相等，`equals()`方法用于比较两个对象的内容是否一致。我们在前面讲过“`==`”操作符用于基本数据类型的变量比较的情况，非常简单和容易理解。如果一个变量是用作指向一个数组或一个对象的句柄，那么这个变量的类型就是引用数据类型，“`==`”操作符用于比较两个引用数据类型变量的情况，对初学者来说，很容易造成一些混淆，我们来看看下面的程序代码：

```

class Compare
{
    public static void main(String[] args)
    {
        String str1 = new String("abc");
    }
}

```

```

String str2 = new String("abc");
String str3 = str1;
if(str1==str2)
    System.out.println("str1==str2");
else
    System.out.println("str1!=str2");
if(str1==str3)
    System.out.println("str1==str3");
else
    System.out.println("str1!=str3");
}
}

```

程序运行结果是：

```

str1!=str2
str1==str3

```

**str1** 和 **str2** 分别指向了两个新创建的 **String** 类对象，尽管创建的两个 **String** 实例对象看上去是一模一样，但他们是两个彼此独立的对象，是两个占据不同内存空间地址的不同对象。**str1** 和 **str2** 分别是这两个对象的句柄，也就是 **str1** 和 **str2** 的值分别是这两个对象的内存地址，显然他们的值是不相等的。将 **str1** 中的值直接赋给了 **str3**，**str1** 和 **str3** 的值当然是相等的。**str1** 和 **str2** 就好比是一对双胞胎兄弟的名称，尽管这对双胞胎兄弟长相完全一模一样，但他们不是同一个人，所以，是不能等同的，**str3** 就好比是为 **str1** 取的一个别名，**str3** 和 **str1** 代表的是同一个人，所以，它们是相等的。这个过程如图 3.7 所示。

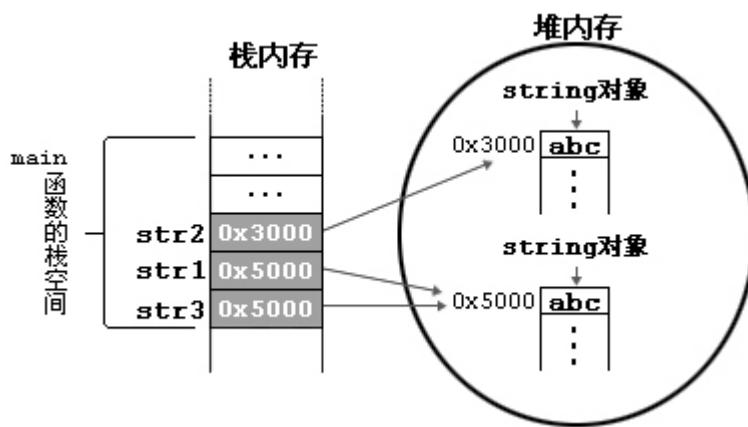


图 3.7

只要在脑子里想的是各个变量所对应的内存状态，就非常容易理解上面的结果了。

我们下面再来看看使用 `equals()` 方法的情况。

```

class Compare
{
    public static void main(String[] args)
    {
        String str1 = new String("abc");
        String str2 = new String("abc");
        String str3 = str1;
    }
}

```

```

if(str1.equals(str2))
    System.out.println("str1 equal str2");
else
    System.out.println("str1 not equal str2");
if(str1.equals(str3))
    System.out.println("str1 equal str3");
else
    System.out.println("str1 not equal str3");
}
}

```

程序运行结果是：

```

str1 equal str2
str1 equal str3

```

**equals()**方法是 String 类的一个成员方法，用于比较两个引用变量所指向的对象的内容是否相等，就像比较两个人的长相是否一样。关于 equals 更详细的信息，我们将在下一章讲解。

## & 多学两招：

同我们刚才讲的普通对象的比较方式一样，我们是不能用“==”运算符来比较两个数组对象的内容是否相等，但数组对象本身又没有 equals 方法，那我们怎样比较两个数组对象的内容是否相等呢？如果你还记得，在上一章中，我们曾经用过 Arrays.sort 方法对数组进行排序，我们为什么不去大胆设想 Arrays 里面可能提供了另外的方法，来帮助我们解决比较两个数组对象的内容是否相等的问题，查看 JDK 文档，如图 3.8 所示

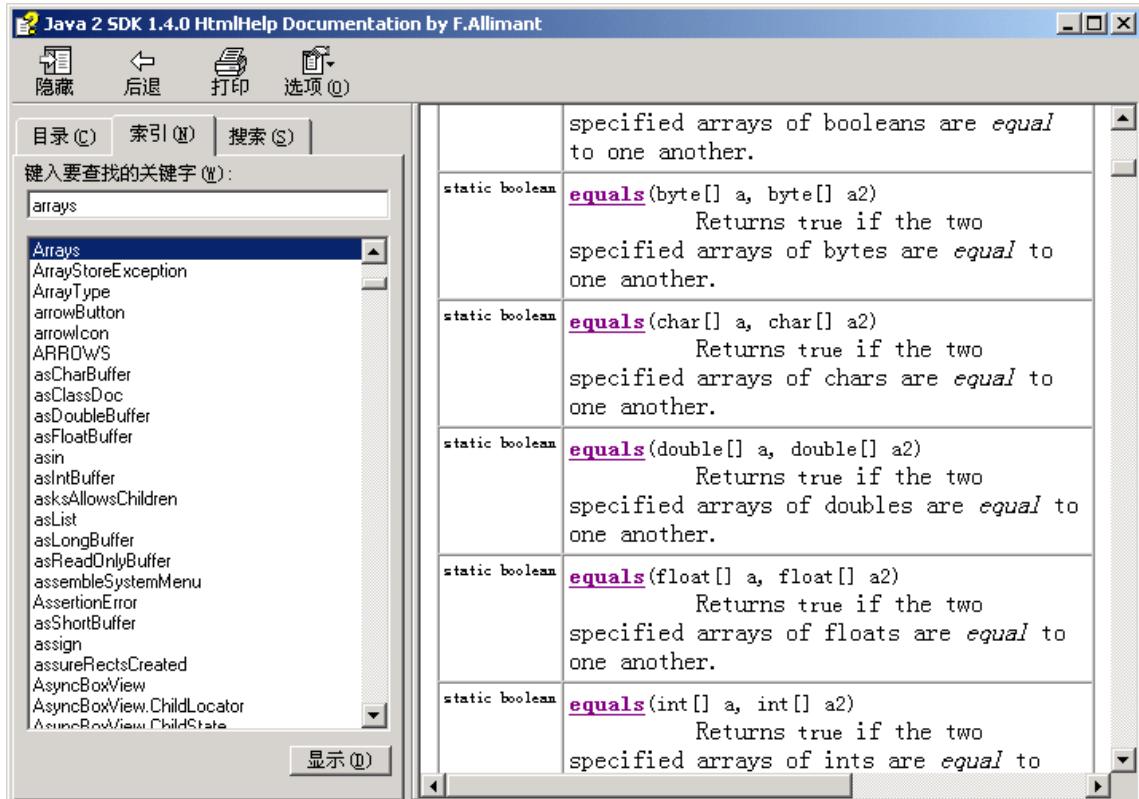


图 3.8

在上面的帮助信息中，我们果然发现该类提供了用于比较数组内容的 equals() 方法。

### 3.2.4 匿名对象

创建完对象，在调用该对象的方法时，我们也可以不定义对象的句柄，而直接调用这个对象的方法。这样的对象叫做匿名对象，我们把前面的 Person 程序中的代码：

```
Person p1=new Person();
p1.shout();
```

改写成：

```
new Person().shout();
```

这句代码没有产生任何句柄，而是直接用 new 关键字创建了 Person 类的对象并直接调用它的 shout() 方法，得出的结果和改写之前是一样的。这个方法执行完，这个对象也就变成了垃圾。

使用匿名对象的两种情况：

- 1). 如果对一个对象只需要进行一次方法调用，那么就可以使用匿名对象。
- 2). 我们经常将匿名对象作为实参传递给一个函数调用，比如程序中有一个 getSomeOne 函数，要接收一个 Person 类对象作为参数，函数定义如下：

```
public static void getSomeOne(Person p)
{
    .....
}
```

我们可以用下面的语句调用这个函数。

```
getSomeOne(new Person());
```

### 3.2.5 实现类的封装性

细心的读者可能已经发现，在上面定义的 Person 类产生的对象中，我们可以直接操作其中的成员变量 age，并且使用了

```
Person p1 = new Person();
p1.age = -30;
```

这样的代码段，显然在实际应用中，不应该出现这样的情况，这样做会导致数据的错误、混乱或安全性问题。如果外面的程序可以随意修改一个类的成员变量，会造成不可预料的程序错误，就象一个人的身高，不能被外部随意修改，只能通过各种摄取营养的方法去修改这个属性。

我们怎样对一个类的成员实现这种保护呢？我们只需要在定义一个类的成员（包括变量和方法）时，使用 private 关键字说明这个成员的访问权限，这个成员成了类的私有成员，只能被这个类的其他成员方法调用，而不能被其他的类中的方法所调用。先来看一段代码：

```
class Person
{
    private int age;
    public void shout()
    {
        System.out.println(age); //这一句没有错误
    }
}
class TestPerson
```

```

{
    public static void main(String[] args)
    {
        new Person().age = -30;
    }
}

编译时回出现下面的错误
E:\Person.java:13: age has private access in Person
    new Person().age = -30;
               ^
1 error

```

错误的意思是：age 是 Person 类里的私有变量，不能在其他类中直接调用和修改。虽然在类 TestPerson 中用的是“对象. 对象成员”的格式去访问 Person 类中的 age 属性，这是从其他类访问另外一个类实例对象的成员的必要格式，但这也是不行的，因为一个类里的成员（包括变量和方法）一旦用 private 加以修饰，就变成该类的私有成员，这个类之外的其它类就再也不能访问它了。

明白了 private 关键字，大家就自然明白了 public 关键字，如果用 public 修饰类里的成员，那么，这些成员就变成公有的，并可以在任意类中访问这种成员，当然，要在一个类外部访问这个类的成员，只能是用“对象. 对象成员”的格式。

为了实现良好的封装性，我们通常将类的成员变量声明为 private，再通过 public 的方法来对这个变量进行访问。对一个变量的操作，一般都有读取和赋值操作，我们分别定义两个方法来实现这两种操作，一个是 getXxx()（Xxx 表示要访问的成员变量的名字），用来读取这个成员变量操作，另外一个是 setXxx() 用来对这个成员变量赋值。按照刚才所讲的思路，我们按下面的代码来修改 Person 类的定义。

```

class Person
{
    private int age;
    public void setAge(int i)
    {
        if(i<0 || i>130)
            return;
        age = i;
    }
    public int getAge()
    {
        return age;
    }
}

public class TestPerson
{
    public static void main(String args[])
    {
        Person p1 = new Person();
        p1.setAge(3);
        p1.setAge(-6);
    }
}

```

```
        System.out.println(p1.getAge());
    }
}
```

这段程序的 Person 类里，成员变量 age 被定义成了私有（private）变量，这样就只有该类中的其他成员可以访问它，然后在该类中定义两个公有（public）的方法 setAge() 和 getAge() 供外部调用者访问，setAge() 方法可以接受一个外部调用者传入的值，当此值超出 0 到 130 的范围就被视为非法，就不再继续对 age 变量进行赋值操作，如果在此范围内就被赋值给成员变量 age，而 getAge() 方法可以给外部调用者返回 age 变量的值，**外部调用者只能访问这两个方法，不能直接访问成员变量 age**。我们通过将类的成员变量声明为私有的（private），再提供一个或多个公有（public）方法实现对该成员变量的访问或修改，这种方式就被称为封装。实现封装可以达到如下目的：

- 隐藏类的实现细节；
- 让使用者只能通过事先定制好的方法来访问数据，可以方便地加入控制逻辑，限制对属性的不合理操作；
- 便于修改，增强代码的可维护性；
- 可进行数据检查；

当然，在实际应用中，对于错误赋值的处理，我们不会只是简单地将变量赋 0 值就算完事，我们可以使用更有效的方式去处理这种非法调用，更有效的通知调用者非法调用的原因。这就是我们以后要讲到的**抛出异常的方式**，读者可以暂时不必细究这个问题，在后面的章节中会有详细的介绍的。

## F 指点迷津：

一个类通常就是一个小的模块，我们应该让模块仅仅公开必须要让外界知道的内容，而隐藏其它一切内容。我们在进行程序的详细设计时，应尽量避免一个模块直接修改或操作另一个模块的数据，**模块设计追求强内聚（许多功能尽量在类的内部独立完成，不让外面干预），弱耦合（提供给外部尽量少的方法调用）**。一支军队对总统只提供一个接收作战命令的 public 方法，至于这支军队在接到作战命令后所要执行的各个作战过程，由这支军队内部来完成就行了，没必要让总统来直接操作这支军队具体的作战过程，将这些具体的作战过程做成 private 方法，这样，总统在调遣这支军队时，只有一个方法可以被使用，这样的总统工作谁都会干，这不正是总统们期望的效果吗？

## & 多学两招：

我们在一个类中定义了一个 private 类型的成员变量，接着产生了这个类的两个实例对象，请问第一个对象的方法中，能否以“第二个对象.成员”的格式访问第二个对象的那个 private 成员变量？通过下面的试验程序代码，就可以找到答案。

```
class A
{
    private int x=3;
    public static void main(String [] args)
    {
        new A().func(new A());
    }
    public void func(A a)
```

```

    {
        System.out.println(a.x);
    }
}

```

编译运行上面程序，没有任何问题，在

```

new A().func(new A());
语句中，一共用 new A() 产生了两个对象，在第一个对象的 func 方法中成功地访问了第二个对象的 private 成员变量 x。

```

### 3.3 构造函数

#### 3.3.1 构造函数的定义与作用

我们先来看一个程序：

```

class Person
{
    public Person()
    {
        System.out.println("the constructor 1 is calling!");
    }

    private int age = 10;

    public void shout()
    {
        System.out.println("age is "+age);
    }
}

class TestPerson
{
    public static void main(String[] args)
    {
        Person p1=new Person();
        p1.shout();

        Person p2=new Person();
        p2.shout();

        Person p3=new Person();
        p3.shout();
    }
}

```

运行结果如下：

```

the constructor 1 is calling!
age is 10
the constructor 1 is calling!
age is 10
the constructor 1 is calling!

```

```
age is 10
```

通过运行的结果读者会发现，在 `TestPerson.main` 方法中并没有调用刚才新加的 `Person()` 方法，但它却被自动调用了，而且我们每创建一个 `Person` 对象，这个方法都会被自动调用一次。这就是“构造方法”。关于这个 `Person` 方法，有几点不同于一般方法的特征：

- | 它具有与类相同的名称；
- | 它不含返回值；
- | 它不能在方法中用 `return` 语句返回一个值。

在一个类中，具有上述特征的方法就是“构造方法”。构造方法在程序设计中非常有用，它可以为类的成员变量进行初始化工作，当一个类的实例对象刚产生时，这个类的构造方法就会被自动调用，我们可以在这个方法中加入要完成初始化工作的代码。这就好像我们规定每个“人”一出生就必须先洗澡，我们就可以在“人”的构造方法中加入完成“洗澡”的程序代码，于是每个“人”一出生就会自动完成“洗澡”，程序就不必再在每个人刚出生时一个一个地告诉他们要“洗澡”了。

虽然初始化在程序设计中是一项非常重要的工作，可是仍然还有很多程序员在编程过程中时常会疏忽对变量的初始化，使用构造方法就可以避免这种情况，并且可以实现一次完成对类的所有实例对象的初始化，这免除了调用程序对每个实例对象都要进行初始化的繁琐工作。

读者站在 Java 设计者的角度去想一想：构造方法的名称和类的名称为什么要相同呢？这是因为构造方法由 Java 编译器负责调用，而编译器必须知道哪一个才是构造方法，采用与类同名的方式应该是最简单合理的。

**M** 脚下留心：在构造方法里不含返回值的概念是不同于“`void`”的，对于“`public void Person()`”这样的写法就不再是构造方法，而变成了普通方法，很多人都会犯这样的错误，在定义构造方法时加了“`void`”，结果这个方法就不再被自动调用了。

### 3.3.2 | 构造方法的重载

上一章我们讲了方法的重载，构造方法也可以被重载，这种情况其实是很常见的，先来看这段程序：

```
class Person
{
    private String name="unknown";
    private int age = -1;
    public Person()
    {
        System.out.println("constructor1 is calling");
    }
    public Person(String n)
    {
        name = n;
        System.out.println("constructor2 is calling");
        System.out.println("name is "+name);
    }
    public Person(String n,int a)
    {
```

```

        name = n;
        age = a;
        System.out.println("constructor3 is calling");
        System.out.println("name and age is "+name+";"+age);
    }
    public void shout()
    {
        System.out.println("listen to me!!");
    }
}
class TestPerson
{
    public static void main(String[] args)
    {
        Person p1=new Person();
        p1.shout();
        Person p2=new Person("Jack");
        p2.shout();
        Person p3=new Person("Tom",18);
        p3.shout();
    }
}

```

在 Person 类里我们又添加了两个构造方法，其中一个构造方法接受外部传入的姓名，再赋值给类的成员变量，另外一个构造方法接受外部传入的姓名和年龄，再赋值给类的成员变量。上面定义了多个 Person 构造方法，这就是构造方法的重载。和一般的方法重载一样，它们具有不同个数或不同类型的参数，编译器就可以根据这一点判断出用 new 关键字产生对象时，该调用哪个构造方法了。

在 TestPerson.main 方法中，我们用 new 关键字产生了三个 Person 的实例对象，产生对象的格式是：

new 类名(参数列表)

Java 会根据我们在括号中传递的参数，自动为我们选择相应的构造方法，程序运行的结果如下：

```

constructor1 is calling
listen to me!!
constructor2 is calling
name is Jack
listen to me!!
constructor3 is calling
name and age is Tom;18
listen to me!!

```

这三个对象调用了不同的构造方法，可见，因为括号中传递的参数个数或类型不同，调用的构造方法也不同。

重载构造方法可以完成不同初始化的操作，当我们要创建一个 Person 实例对象的同时，就直接给对人的姓名和年龄赋值，我们就可以使用下面的方式去产生这个 Person 实例对象。

```
Person p1=new Person("Tom",18);
```

而不必先用 Person p1=new Person(); 语句产生 Person 实例对象，再单独对这个 Person 实例对象的姓名和年龄赋值。

我们来分析一下 p3=new Person("Tom",18); 语句都干了些什么？

它会做这样几件事：创建指定类的新实例对象，在堆内存中为实例对象分配内存空间，并调用指定类的构造方法，最后将实例对象的首地址赋值给引用变量 p3。

首先，等号左边定义了一个类 Person 类型的引用变量 p3，等号右边使用 new 关键字创建了一个 Person 类的实例对象。此时的内存状态如图 3.9 所示：

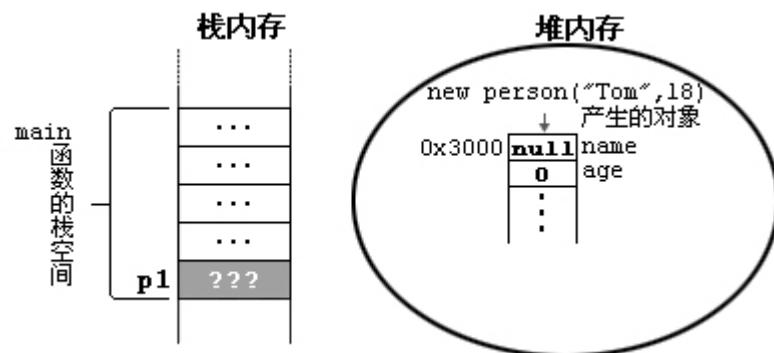


图 3.9

接着，调用相应的构造方法，构造方法接受外部传入的姓名和年龄，在执行构造方法中的代码之前，进行属性的显式初始化，也就是执行在定义成员变量时就对其进行赋值的语句，即程序 Person 类中的：

```
private String name="unknown"; //显式初始化  
private int age = -1; //显式初始化
```

此时的内存状态如图 3.10 所示：

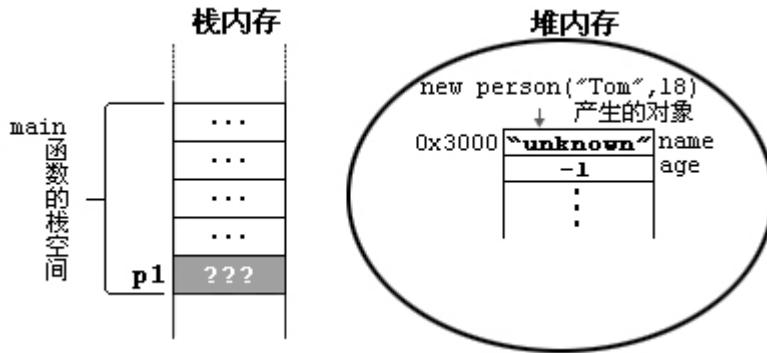


图 3.10

然后，执行构造方法中的代码，用从外部接受到的姓名和年龄对成员变量重新赋值。此时的内存状态如图 3.11 所示：

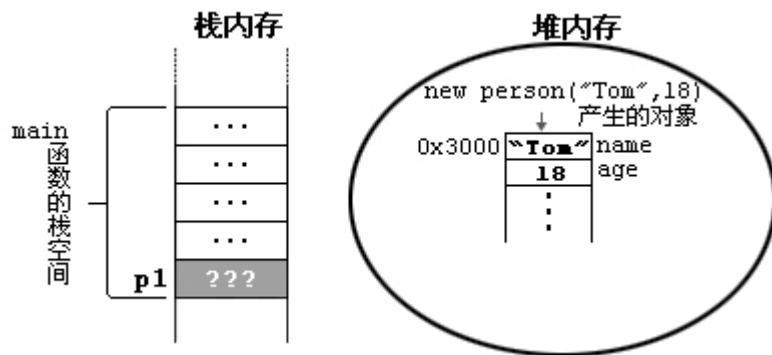


图 3.11

最后，把刚刚创建的对象赋给引用变量，如图 3.12 所示：

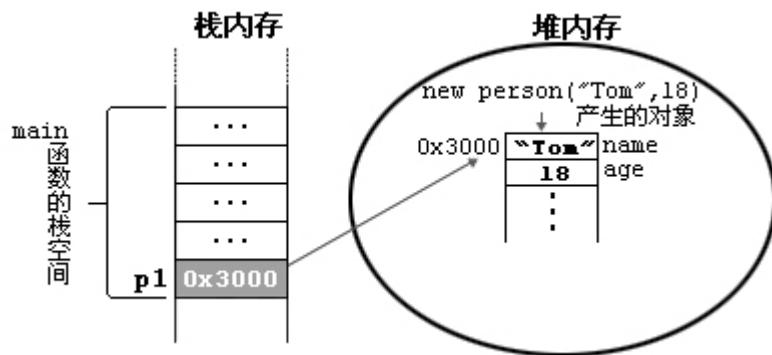


图 3.12

### 3.3.3 构造方法的一些细节

1). 在 Java 的每个类里都至少有一个构造方法，如果程序员没有在一个类里定义构造方法，系统会自动为这个类产生一个默认的构造方法，这个默认构造方法没有参数，在其方法体中也没有任何代码，即什么也不做。

下面程序的 Construct 类两种写法完全是一样的效果。

```
class Construct
{
}
class Construct
{
    public Construct(){}
}
```

对于第一种写法，类虽然没有声明构造方法，但我们可以用 new Construct() 语句来创建 Construct 类的实例对象。

由于系统提供的默认构造方法往往不能满足编程者的需求，我们可以自己定义类的构造方法，来满足我们的需要，一旦编程者为该类定义了构造方法，系统就不再提供默认的构造方法了。

```
class Construct
```

```

{
    int age;
    public Construct(int x)
    {
        age = x;
    }
}

```

上面的 `Construct` 类中定义了一个对成员变量赋初值的构造方法，该构造方法有一个形式参数，这时系统就不再产生默认的构造方法，我们再编写一个调用 `Construct` 类的程序。

```

class TestConstruct
{
    public static void main(String [] args)
    {
        Construct c = new Construct();
    }
}

```

编译上面的程序，出现如下错误：

```

E: \TestConstruct.java:13: cannot resolve symbol
symbol : constructor Construct ()
location: class Construct
    Construct c = new Construct();
                           ^
1 error

```

错误的原因就是，调用 `new Construct()` 创建 `Construct` 类的实例对象时，要调用的是没有参数的那个构造方法，而我们又没有定义无参数的构造方法，但我们定义了一个有参数的构造方法，系统不再为我们自动生成无参数的构造方法。针对这种情况，我们只要自己定义构造方法，都需要带上一个无参数的构造方法，否则，就会经常碰到上面这样的错误。

2). 思考一下，声明构造方法时，可以使用 `private` 访问修饰符吗？运行这段程序，看看有什么结果：

```

class Person
{
    private Person()
    {
        System.out.println("the constructor 1 is calling!");
    }
}
class TestPerson
{
    public static void main(String[] args)
    {
        Person p1=new Person();
    }
}

```

程序会报错：

```
E:\TestPerson.java:12: Person() has private access in Person
    Person p1=new Person();
               ^
1 error
```

这段信息是说 Person() 构造方法是私有的，不可以被外部调用，可见构造方法一般都是 public 的，因为它们在对象产生时会被系统自动调用的。

### 3.4 this 引用句柄

一个对象中的一个成员方法，可以引用另外一个对象的成员，如下面的程序代码：

```
class A
{
    String name;
    public A(String x)
    {
        name = x;
    }
    public void func1()
    {
        System.out.println("func1 of " + name + " is calling");
    }
    public void func2()
    {
        A a2 = new A("a2");
        a2.func1();
    }
}
class TestA
{
    public static void main(String [] args)
    {
        A a1 = new A("a1");
        a1.func2();
    }
}
```

在上面的程序中，我们一共产生了两个类 A 的实例对象，在 a1 的 func2 中，我们调用了 a2 的 func1。我们现在把眼光集中在 func2 中的代码上，我们已经看到了如何在 func2 中引用另外一个对象的成员，在 func2 运行期间的内存状态如图 3.13 所示。

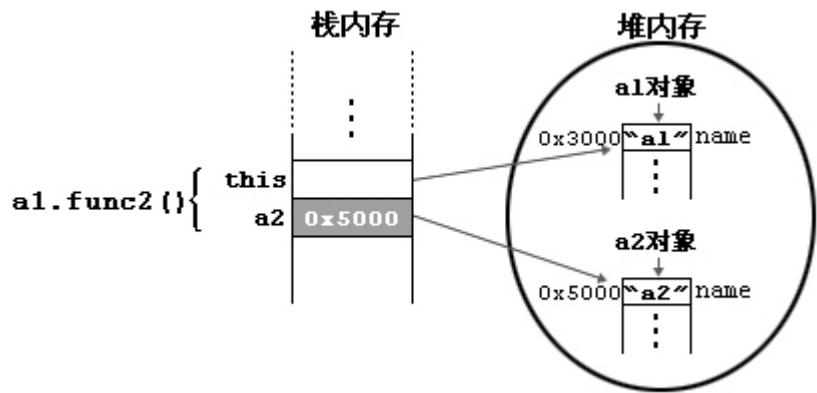


图 3.13

第一个要想明白的问题是，如果 func2 方法被调用，是不是一定要以“对象. func2”的方式进行呢？是的，那就是说，func2 被调用时，一定是事先已经有了一个存在的对象，func2 被作为那个对象的方法被使用。

第二个要想明白的问题是，我们在 func2 内部能引用别的对象，我们能不能在 func2 内部引用那个“事先存在并对 func2 进行调用”的对象呢？我们将这个“事先存在并对 func2 进行调用”的对象就叫 func2 所属的那个对象，对于一个方法来说，只要是对象，它就可以调用，它根本就不区分是不是自己所属的那个对象！

肯定搞清楚了上面的两个问题后，我们接着来思考下面的问题，在 func2 中，怎么引用自己所属的那个对象呢？也就是自己所属的那个对象的引用名称是什么呢？是 a1 吗？那个对象在 TestA.main 方法中叫 a1，但在 func2 内就不叫 a1 了，因为，我们在定义 func2 时，对象 a1 还没产生，我们不可能提前知道以后产生的类 A 的每个实例对象的引用名称。另外，如果我们在 func2 中用 a1 代表了 func2 所属的那个对象，那么，当我们调用 a2. func2() 时，这时的 func2 所属的那个对象就不再是 a1 了，而是 a2 了，显然我们在 func2 中不能用以后定义的对象引用名来引用它所属的那个对象。this 关键字在 java 程序里的作用和它的词义很接近，它在函数内部就是这个函数所属的对象的引用变量。

我们修改类 A 的程序代码，在 func2 中使用 this 关键字调用 func1 方法。

```
class A
{
    String name;
    public A(String x)
    {
        name = x;
    }
    public void func1()
    {
        System.out.println("func1 of " + name + " is calling");
    }
    public void func2()
    {
        A a2 = new A("a2");
        this.func1(); // 使用 this 关键字调用 func1 方法
        a2.func1();
    }
}
```

```
}
```

重新编译 TestA.java 后，运行类 TestA，结果如下：

```
func1 of a1 is calling  
func1 of a2 is calling
```

我们前面讲过，一个类中的成员方法可以直接调用同类中的其他成员，其实我们将 `this.func1();` 调用直接写成 `func1();` 调用，效果是一样的。

对于类 A 中的构造函数：

```
public A(String x)  
{  
    name = x;  
}
```

可以改写成如下形式：

```
public A(String x)  
{  
    this.name = x;  
}
```

在成员方法中，对访问的同类中成员前加不加 `this` 引用，效果都是一样的，这就好像同一个公司的职员彼此在提及和自己公司有关的事时，不必说出公司名一样，当然为了强调，可以加上“咱们公司...”这样的前缀，在程序里同样可以如此，而 `this` 就相当于“我所属于的那个对象”。每个成员方法内部，都有一个 `this` 引用变量，指向调用这个方法的对象，刚才我们所用的类 A 中的成员方法与 `this` 之间的关系如图 3.13 所示。

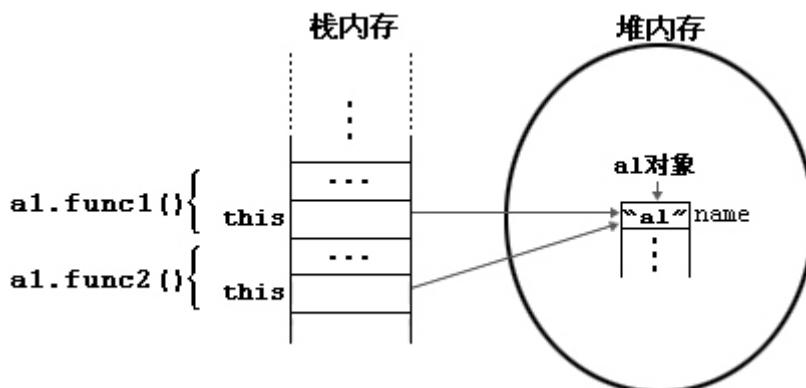


图 3.14

说了半天，读者应该总算明白 `this` 关键字的意义了，但好象用不用 `this`，程序效果都一样，那 `this` 还有多大的作用呢？在有些情况下，我们还是非得用 `this` 关键字不可的。

1). 我们想通过构造方法将外部传入的参数赋值给类成员变量，构造方法的形式参数名称与类的成员变量名相同。

```
class Person  
{  
    String name;  
    Public Person(String name)  
    {  
        name=name;  
    }  
}
```

在这段代码里，读者肯定会对 name=name; 这样的赋值语句莫名其妙，根本分不出哪个是成员变量，哪个是方法的变量。有人说，你定义构造方法的形式参数名称不要与类成员变量名相同，不就解决这个问题了吗？虽然在前面的例子中，我们都是这样做的，那是因为我们还没讲到 this 的原因，只是为了讲课而这么做的，这在实际应用中绝对不是一个好主意。软件工程学中要求我们不仅要完成程序的功能，还要追求程序代码良好的可理解性，让类的成员变量名和对其进行赋值的成员方法的形参变量同名是必要的，这样的代码谁看了都能明白这两个变量是彼此相关的，老手看到函数的定义，就能揣摩出函数中的代码，大大节省了别人和自己日后阅读程序的时间。

形式参数就是方法内部的一个局部变量，成员变量与方法中的局部变量同名时，在该方法中对同名变量的访问是指那个局部变量。如果明白了这个道理和 this 关键字的作用，我们就可以修改上面的程序代码，来达到我们的目的。

```
class Person
{
    String name;
    Public Person(String name)
    {
        this.name=name;
    }
}
```

2). 假设我们有一个容器类和一个部件类，在容器类的某个方法中要创建部件类的实例对象，而部件类的构造方法要接收一个代表其所在容器的参数，程序代码如下：

```
class Container
{
    Component comp;
    public void addComponent()
    {
        comp = new Component(this); //将 this 作为对象引用传递
    }
}
class Component
{
    Container myContainer;
    public Component(Container c)
    {
        myContainer = c;
    }
}
```

读者在脑子中，多想想各对象在内存中的状态，就很容易看懂上面的代码的。我不希望读者学完这本书后，还要我来提醒你：“读代码时，不是专盯代码本身，而是要看内存状态”，但在这本书中，我还得反复强调，反复地说。下面的图 3.15 所示是我在脑海中设想的内存状态，供你参考。

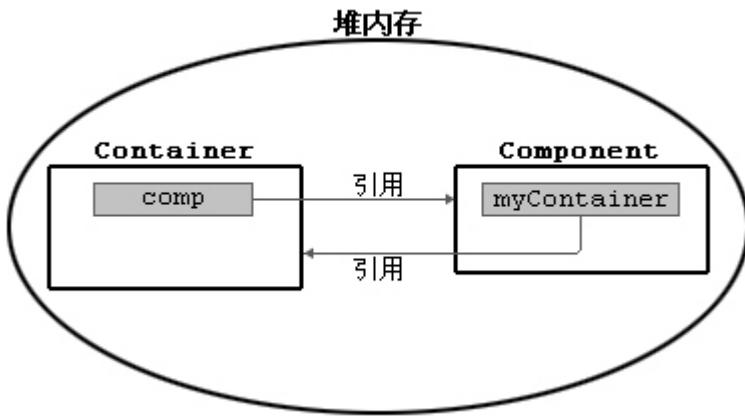


图 3.15

这就是通过 `this` 引用把当前的对象作为一个参数传递给其它的方法和构造方法的应用。

3). 构造方法是在产生对象时被 Java 系统自动调用的，我们不能在程序中象调用其他方法一样去调用构造方法。但我们可以在一个构造方法里调用其他重载的构造方法，不是用构造方法名，而是用 `this(参数列表)` 的形式，根据其中的参数列表，选择相应的构造方法。

```
public class Person
{
    String name;
    int age;
    public Person(String name)
    {
        this.name = name;
    }
    public Person(String name,int age)
    {
        this(name);
        this.age = age;
    }
}
```

} 在类 Person 的第二个构造方法中，我们通过 `this(...)` 调用，执行第一个构造方法中的代码。

## 3.5 与垃圾回收有关的知识

我们在第一章中详细介绍了垃圾回收的一些内部细节，在这里，我们只准备向读者讲解与垃圾回收有关的一些程序代码。

### 3.5.1 finalize 方法

`finalize()` 方法是 `Object` 类的一个方法，任何一个类都从 `Object` 那继承了这个方法，有关类 `Object` 和继承的内容，我们将在第四章中讲解，没有任何基础的读者现在可以不用理会。

为了便于理解 `finalize()` 方法的作用，我们先介绍一下 C++ 中的析构方法。在一个对象即将被从内存中释放时，析构方法被自动调用，这个方法里的程序代码也就得到了执行。有了析构方法，我们的程序就可以实现“每一个对象被释放前，自动去完成一些清理工作方面的事情”这样的功能。在 C++ 中，我们常用析构方法去释放对象在生存期间所占用的一些资源，就好比

我们要求每个人在临终之前都要写出他所知道的他的债权，不至于将这些只有他知道的东西在其死后成为永远的秘密，我们就可以在“人”类的析构函数中增加“写出他所知道的他的债权”的程序代码。只要某个对象的构造方法被调用了，我们就知道一个对象刚产生了，那么只要某个对象的析构方法被调用了，我们就知道这个对象即将消亡了。由于类代码的编写早于对象的产生，利用构造和析构方法，我们就可以提前对对象的产生和消亡过程进行一些控制。我经常问学员们这样一个问题，构造方法被调用时，对象产生了吗？析构方法被调用时，对象还在内存中存在吗？俗话说：“皮之不存，毛将附焉？”，如果对象都不存在，又怎么能够调用它的方法，所以，无论是构造方法被调用，还是析构方法被调用，对象都在内存中存在。

`finalize()`方法的作用类似 C++ 中的析构方法，但由于 Java 有垃圾回收的机制，当一个对象变成垃圾，他所引用的其他对象在没有被别的有效的句柄引用的话，也会随之变成垃圾，也就是不用程序员再去做“释放对象在生存期间所占用的一些资源”这样的事情了。`finalize()`方法的作用就如 C++ 中的析构方法那么重要了。`finalize()`方法是在对象被当作垃圾从内存中释放前调用，而不是在对象变成垃圾前调用，垃圾回收器的启用不由程序员控制，也无规律可循，并不会一产生了垃圾，它就被唤起，甚至有可能到程序终止，它都没有启动的机会。因此这并不是一个很可靠的机制，所以，我们无法保证每个对象的 `finalize()` 方法最终都会被调用。我们只要了解一下 `finalize()` 方法的作用就行了，不要期望 `finalize()` 方法去帮我们做“需要可靠完成”的工作。

下面，我们通过程序来了解 `finalize()` 方法：

```
class Person
{
    public void finalize()
    {
        System.out.println("the object is going!");
    }
    public static void main(String [] args)
    {
        new Person();
        new Person();
        new Person();
        System.out.println("the program is ending!");
    }
}
```

编译运行后的结果是：

`the program is ending!`

在上面的程序中，我们产生了三个匿名对象，这些对象在执行

```
System.out.println("the program is ending!");
```

语句前都变成了垃圾，但我们并没有看到垃圾回收时 `finalize` 方法被调用的效果。

### 3.5.2 System.gc 的作用

Java 的垃圾回收器被执行的偶然性有时候也会给程序运行带来麻烦，比如说在一个对象成为垃圾时需要马上被释放，或者程序在某段时间内产生大量垃圾时，释放垃圾占据的内存空间似乎成了一件棘手的事情，如果垃圾回收器不被启动，`finalize()` 方法也不会被调用。为此，Java 里提供了一个 `System.gc()` 方法，使用这个方法可以强制启动垃圾回收器来回收垃圾，就

象我们主动给环卫局打电话，通知他们提前来清扫垃圾的道理是一样的。我们将上面的程序作如下修改：

```
class Person
{
    public void finalize()
    {
        System.out.println("the object is going!");
    }
    public static void main(String [] args)
    {
        new Person();
        new Person();
        new Person();

        System.gc();
        System.out.println("the program is ending!");
    }
}
```

编译运行的结果如下：

```
the object is going!
the object is going!
the object is going!
the program is ending!
```

通过本节的讲解，读者对 Java 垃圾回收机制应该有了比较直观的基本认识。

## 3.6 函数的参数传递

### 3.6.1 基本数据类型的参数传递

我们在第二章中，已经讲了方法的参数传递过程，并强调了方法的形式参数就相当于方法中定义的局部变量，方法调用结束时也就被释放了，不会影响到主程序中同名的局部变量，我们看看下面的程序代码：

```
class PassValue
{
    public static void main(String [] args)
    {
        int x = 5;
        change(x);
        System.out.println(x);
    }

    public static void change(int x)
    {
        x = 3;
    }
}
```

请问：上面的程序打出的结果是 3 还是 5 呢？我们通过下面的图 3.16 来描述 change 方法被调用的内存状况。

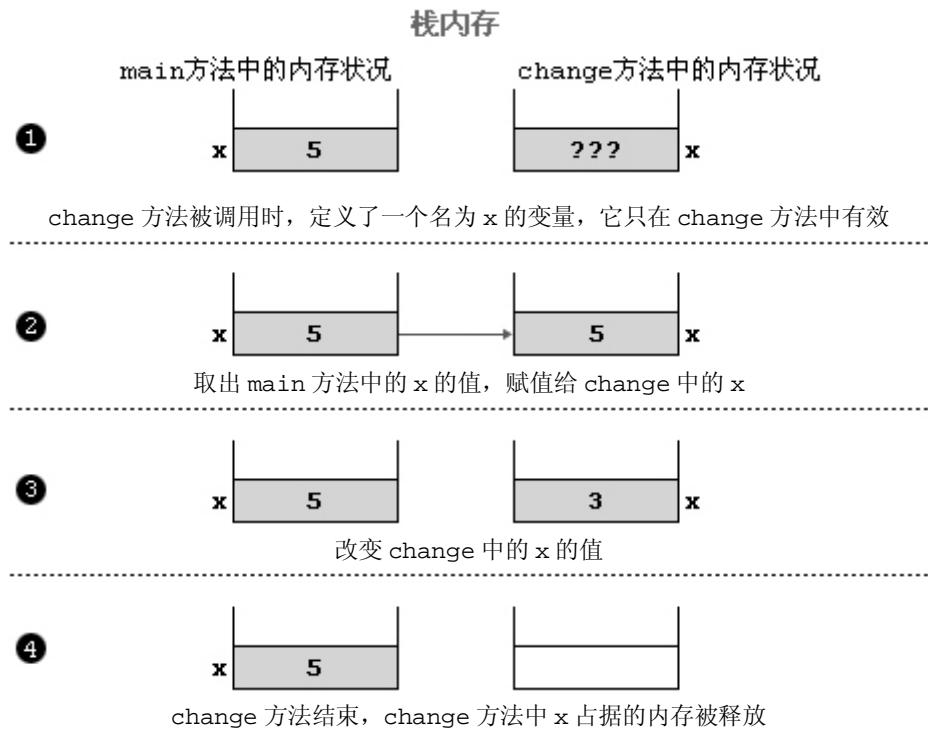


图 3.16

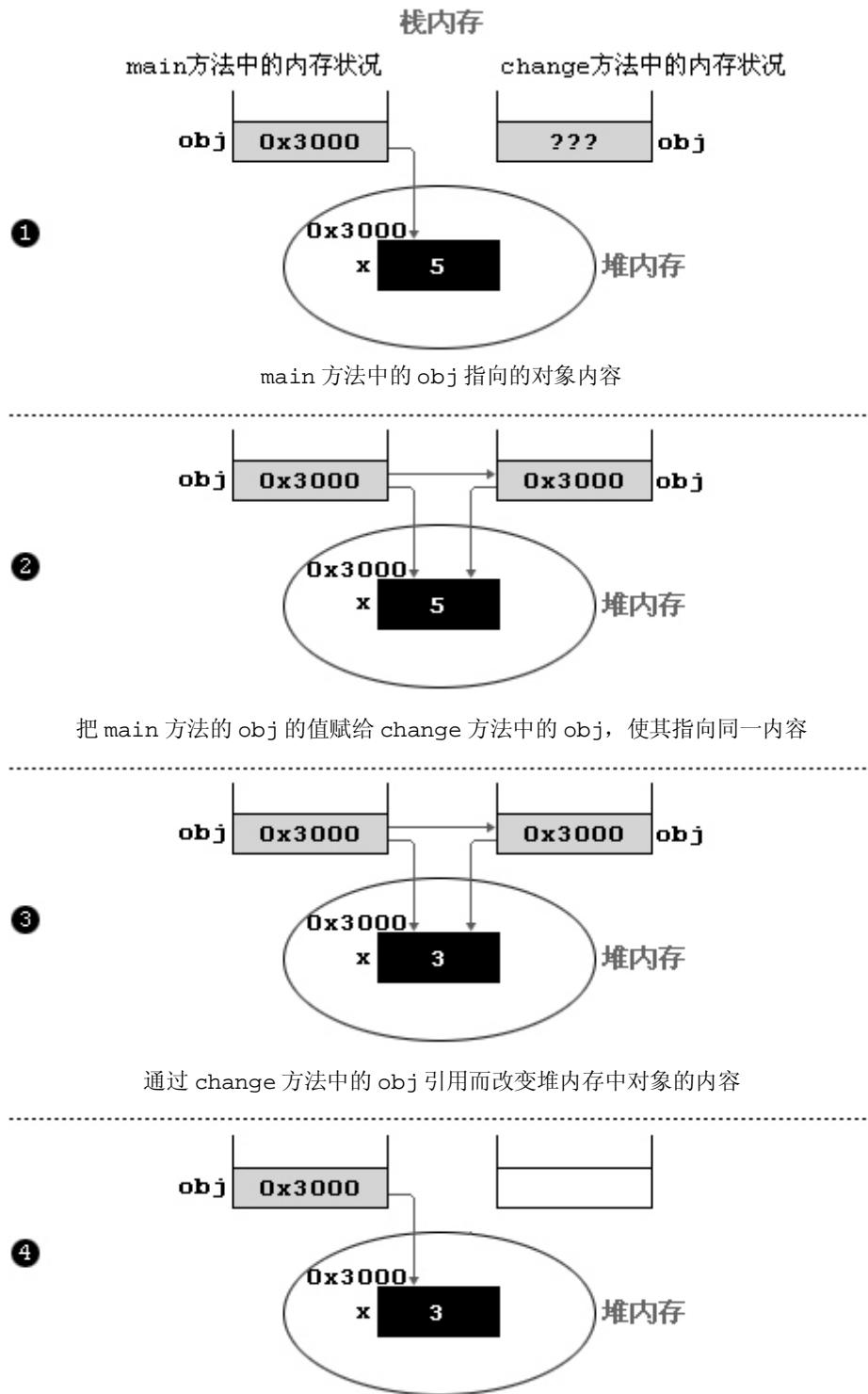
显然，change 方法从开始到结束的过程中并没有改变 main 方法中的 x 的值，所以打印出来的结果应该是 5。可见，基本类型的变量作为实参传递，并不能改变这个变量的值。

### 3.6.2 引用数据类型的参数传递

对象的引用变量并不是对象本身，它们只是对象的句柄（名称）。就好像一个人可以有多个名称一样（如中文名，英文名），一个对象可以有多个句柄，我们在前面已经讲过对象的生命期与引用变量之间的关系。

```
class PassRef
{
    int x ;
    public static void main(String [] args)
    {
        PassRef obj = new PassRef();
        obj.x = 5;
        change(obj);
        System.out.println(obj.x);
    }
    public static void change(PassRef obj)
    {
        obj.x=3;
    }
}
```

上面的程序打印出来的结果是多少呢？编译运行一下，打印的结果是 3，这是为什么呢？我们通过图 3.17 来看一下发生了什么：



change 方法结束，change 中的 obj 变量被释放，但堆内存的对象仍然被 main 方法中的 obj 引用，我们就会看到：在 main 方法中的 obj 所引用的对象的内容被改变

图 3.17

main方法中的obj值没有改变，这和3.16中的过程是一样的，所以还是指向那个对象，但指

向的对象的内容已在change方法中被改变。 change方法中的obj（这里用A标记）就好比main方法中的obj（这里用B标记）的别名，对A所引用的对象的任何操作就是对B所引用的对象的操作。例如有人名叫张小二，他的绰号是“五狗子”，说“五狗子”怎么怎么的，其实就是对张小二说三道四。

Java语言在给被调用方法的参数赋值时，只采用传值的方式。所以，基本类型数据传递的是该数据的值本身，引用类型数据传递的也是这个变量的值本身，即对象的引用（句柄），而非对象本身，通过方法调用，可以改变对象的内容，但是对象的引用是不能改变的。对于数组，也属于引用类型，将数组对象作为参数传递，和上面的例子有同样的效果，如：

```
class PassRef
{
    public static void main(String [] args)
    {
        int [] x= new int[1];
        x[0] = 5;
        change(x);
        System.out.println(x[0]);
    }
    public static void change(int [] x)
    {
        x[0] = 3;
    }
}
```

我们接着将上面传递对象引用的程序略作修改，大家来分析一下这个修改后的程序打印出来的结果是多少。

```
class PassRef
{
    int x ;
    public static void main(String [] args)
    {
        PassRef obj = new PassRef();
        obj.x = 5;
        change(obj);
        System.out.println(obj.x);
    }
    public static void change(PassRef obj)
    {
        obj = new PassRef();
        obj.x=3;
    }
}
```

这段程序中内存的分配情况如图 3.18 所示：

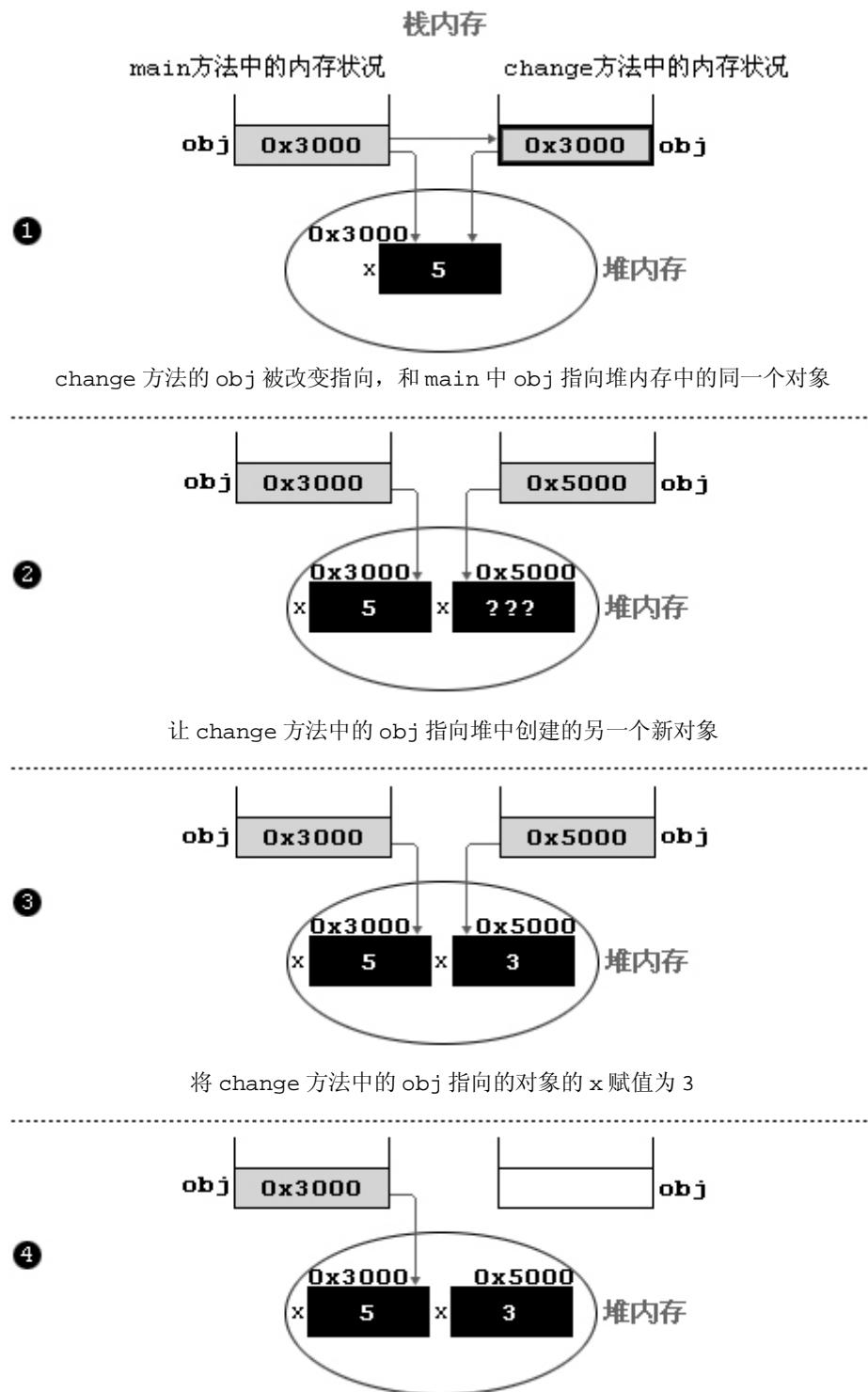


图 3.18

可见，我们在程序中编写的方法，如果不小心写出了类似上面的 change 方法中的代码，根本没有任何意义，因为这个方法的调用对程序的功能没有起到任何作用，在 Java 中，我们一般不会出现这样的错误，但在 C 语言的指向指针的指针参数传递过程中，少不了会出现这样的错误。

## 3.7 static 关键字

### 3.7.1 静态变量

当我们编写一个类时，其实就是在描述其对象的属性和行为，而并没有产生实质上的对象，只有通过 new 关键字才会产生出对象，这时系统才会分配内存空间给对象，其方法才可以供外部调用。

我们有时候希望无论是否产生了对象或无论产生了多少对象的情况下，某些特定的数据在内存空间里只有一份，例如所有的中国人都有个国家名称，每一个中国人都共享这个国家名称，不必在每一个中国人的实例对象中都单独分配一个用于代表国家名称的变量。如图 3.19 所示：

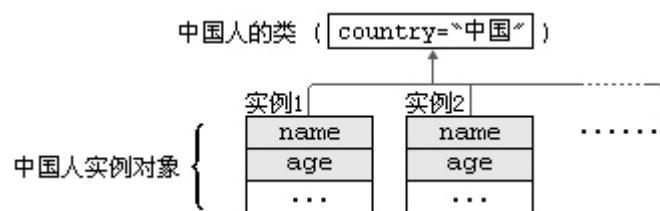


图 3.19

虽然，在各实例对象中没有分配 country 这个变量，但我们在程序中通过中国人的实例对象来访问这个变量，要实现这个效果，我们只需要在类中定义的 country 变量前面加上 static 关键字即可，我们称这种变量为静态成员变量。我们也可以直接使用类名来访问这个 country 这个变量，还可以在类的非静态的成员方法中象访问其他非静态成员变量一样去访问这个静态成员变量。静态变量在某种程度上与其它语言的全局变量相类似，如果不是私有的就可以在类的外部进行访问，此时不需要产生类的实例对象，只需要类名就可以引用。

```
class Chinese
{
    static String country="中国";
    String name;
    int age;
    void singOurCountry()
    {
        System.out.println("啊！，亲爱的" + country);
        //类中的成员方法也可以直接访问静态成员变量
    }
}
class TestChinese
{
    public static void main(String [] args)
    {
        System.out.println("Chinese country is " + Chinese.country);
        //上面的程序代码直接使用了“类名.成员”的格式
        Chinese ch1 = new Chinese();
        System.out.println("Chines country is " + ch1.country);
        //上面的程序代码直接使用了“对象名.成员”的格式
    }
}
```

```
    ch1.singOurCountry();
}
}
```

注意：我们不能把任何方法体内的变量声明为静态，如下面这样是不行的：

```
fun()
{
    static int i = 0;
}
```

用 `static` 标识符修饰的变量，它们在类被载入时创建，只要类存在，`static` 变量就存在。由于静态成员变量能被各实例对象所共享，所以我们可以用它来实现一些特殊效果，如我们想统计在程序中一共产生了多少某个类的实例对象，可以用下面的方法统计：

```
class A
{
    private static int count = 0;
    public A()
    {
        count = count + 1;
    }
}
```

在上面的程序中，每产生一个类 A 的实例对象，都会调用类 A 的构造方法，我们在构造方法中将 `count` 加 1，就可以统计出总共产生了多少个类 A 的实例对象，为了防止外面的程序直接修改 `count` 变量，我们用 `private` 关键字限定了 `count` 变量的访问权限。

我们如何统计一个类在程序中目前有多少个实例对象呢？我们只要在上面这个程序的基础上增加一些代码，在一个实例对象被释放时将 `count` 减 1。那么怎么能够提前知道一个对象在什么时候会被释放呢？在垃圾回收中，我们讲过如果一个对象的 `finalize` 方法被调用，就表示这个对象马上要被从内存中清除了。我们可以这样编写代码，来实现我们的需求。

```
class A
{
    private static int count = 0;
    public A()
    {
        count = count + 1;
    }
    public void finalize()
    {
        count = count - 1;
    }
}
```

我们讲过，垃圾回收器的启用不由程序员控制，也无规律可循，并不会一产生了垃圾，它就被唤起，甚至有可能到程序终止，它都没有启动的机会，因此利用垃圾回收来解决我们程序中的一些问题，并不是一个很可靠的机制。也就是说，我们很难真正实现“统计一个类在程序中目前有多少个实例对象”这个愿望。其实，我们也不需要去处理这个问题，因为，一个对象只要变成了垃圾（还会呆在内存中），我们就不用再管它了，剩下来的工作，都是 Java 系统的事情，跟我们的程序没有关系。我们真正可能会关心的一个问题是“统计一个类在程序中目前

有多少个有效（还没变成垃圾）的实例对象”，对于这个问题，我们只要知道一个对象变成垃圾时会调用对象的哪个方法，就可以实现我们的需求了。当没有引用指向一个对象时，这个对象就会变成垃圾。这个问题涉及一些更复杂的知识，不属于本书的讲解范围，我们就不在这作详细讲解了，作者之所以提出这个问题，只是为了开阔读者的思维，培养读者的编程思想。

### 3.7.2 静态方法

我们有时也希望不必创建对象就可以调用某个方法，换句话说也就是使该方法不必和对象绑在一起。要实现这样的效果，我们只需要在类中定义的方法前加上 `static` 关键字即可，我们称这种方法为静态成员方法。同静态成员变量一样，我们可以用类名直接访问静态成员方法，也可以用类的实例对象来访问静态成员方法，还可以在类的非静态的成员方法中象访问其他非静态方法一样去访问这个静态方法，如下面的程序代码：

```
class Chinese
{
    static void sing()
    {
        System.out.println("啊！");
    }
    void singOurCountry()
    {
        sing();
        //类中的成员方法也可以直接访问静态成员方法
    }
}
class TestChinese
{
    public static void main(String [] args)
    {
        Chinese.sing();
        //上面的程序代码直接使用了“类名.成员”的格式
        Chinese ch1 = new Chinese();
        ch1.sing();
        //上面的程序代码直接使用了“对象名.成员”的格式
        ch1.singOurCountry();
    }
}
```

类的静态成员经常被称作“类成员”(class members)，对于静态成员变量，我们叫类属性(class attributes)，对于静态成员方法，我们叫类方法(class methods)。采用 `static` 关键字说明类的属性和方法不属于类的某个实例对象，我们在前面的多个例子程序中反复用到的 `System.out.println()` 语句，其中 `System` 是一个类名，`out` 是 `System` 类的一个静态成员变量，`println()` 方法则是 `out` 所引用的对象的方法。`System.gc()` 语句中的 `gc()` 也是 `System` 类的一个静态方法。

在使用类的静态方法时，我们要注意以下几点：

- 1) 在静态方法里只能直接调用同类中其它的静态成员（包括变量和方法），而不能直接访问

类中的非静态成员。这是因为，对于非静态的方法和变量，需要先创建类的实例对象后才可使用，而静态方法在使用前不用创建任何对象。

2). 静态方法不能以任何方式引用 `this` 和 `super` 关键字（`super` 关键字在下一章讲解）。与上面的道理一样，因为静态方法在使用前不用创建任何实例对象，当静态方法被调用时，`this` 所引用的对象根本就没有产生。

3). `main()` 方法是静态的，因此 JVM 在执行 `main` 方法时不创建 `main` 方法所在的类的实例对象，因而在 `main()` 方法中，我们不能直接访问该类中的非静态成员，必须创建该类的一个实例对象后，才能通过这个对象去访问类中的非静态成员，这种情况，我们在以后的例子中会多次碰到。

对于上面的几个注意点，不必要死记硬背，只要从原理上去想为什么，反而很容易记住。

### 3.7.3 静态代码块

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

```
class StaticCode
{
    static String country;
    static
    {
        country = "china";
        System.out.println("StaticCode is loading");
    }
}
class TestStaticCode
{
    static
    {
        System.out.println("TestStaticCode is loading");
    }
    public static void main(String [] args)
    {
        System.out.println("begin executing main method");
        new StaticCode();
        new StaticCode();
    }
}
```

编译运行上面的程序，结果如下：

```
TestStaticCode is loading
begin executing main method
StaticCode is loading
```

类 `StaticCode` 中的静态代码块被自动执行，尽管我们产生了类 `StaticCode` 的两个实例对象，但其中的静态代码块只被执行了一次。上面的例子也反过来说明，当一个程序中用到了其

他的类，才会去装载那个类。因此，我们得出这个结论：类是在第一次被使用的时候才被装载，而不是在程序启动时就装载程序中所有可能要用到的类。

### 3.7.4 单态设计模式

设计模式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。设计模式就像是经典的棋谱，不同的棋局，我们用不同的棋谱，免得我们自己再去思考和摸索。失败为成功之母，但是要以大量的时间和精力为代价，如果有成功经验可借鉴，没有人再愿意去甘冒失败的风险，我们没有理由不去了解和掌握设计模式，这也是 Java 开发者提高自身素质的一个很好选择。使用设计模式也许会制约你去创新，不过真正有意义的创新只能出自少数天才，即使你就是那个天才，虽不必因循守旧，但也可能完全不去了解和借鉴前人的成功经验。

所谓类的单态设计模式，就是采取一定的方法保证在整个的软件系统中，对某个类只能存在一个对象实例，并且该类只提供一个取得其对象实例的方法。

如果我们要让类在一个虚拟机中只能产生一个对象，我们首先必须将类的构造方法的访问权限设置为 `private`，这样，就不能用 `new` 操作符在类的外部产生类的对象了，但在类内部仍可以产生该类的对象。因为在类的外部开始还无法得到类的对象，只能调用该类的某个静态方法以返回类内部创建的对象，静态方法只能访问类中的静态成员变量，所以，指向类内部产生的该类对象的变量也必须定义成静态的。下面是一个单态类的程序例子：

```
public class TestSingle
{
    private static final TestSingle onlyOne=new TestSingle();
    public static TestSingle getTestSingle()
    {
        return onlyOne;
    }
    private TestSingle(){}
}
```

对于上面的程序，我们在外面只能调用 `TestSingle.getTestSingle()` 方法获得 `TestSingle` 的对象。我的一些学员告诉我，有些用人单位对于考察应聘者这样的知识点乐此不疲，希望刚才这句话，能对那些想通过学习 Java 编程去找工作的读者有所帮助。

### 3.7.5 理解 main 方法的语法

在前面的章节中，我们已经多次看到，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` 方法的定义了。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。我们通过运行下面的程序来了解如何向类传递参数，程序又是如何取得这些参数的：

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

```

        for(int i=0;i<args.length;i++)
        {
            System.out.println(args[i]);
        }
    }
}

```

按下面的格式，运行这个程序。

```
Java TestMain first second
```

打印结果如下：

```
first
second
```

`args` 数组中元素的个数就是在命令行中给类传递的参数的个数，每个参数之间用空格分开，如果一个参数中有空格，将这个参数用双引号（"）引起来。参数与 `args` 数组的对应关系如图 3.20 所示：

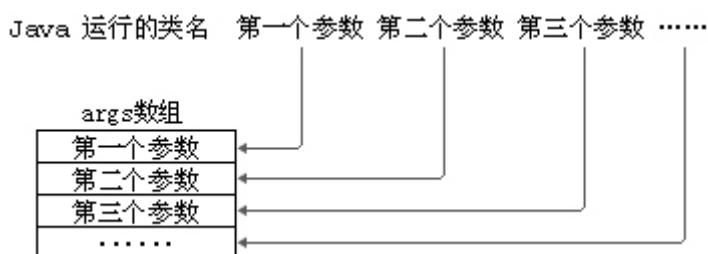


图 3.20

## 3.8 内部类

在一个类内部定义类，这就是嵌套类（nested classes），也叫内部类、内置类。读者以后不要死抠这些名称术语，不同的书籍有不同的翻译风格，你只要知道怎么回事就行了。我曾经碰到一个学员为了弄清楚方法与函数的区别（因为有的书叫方法，有的书叫函数，反而把他给弄糊涂了），翻阅了大量的书籍，都没有搞清楚这两者的区别，其实是同一个事务的两种不同叫法，因为他思考方式的问题，反而在他那变得复杂了。嵌套类可以直接访问嵌套它的类的成员，包括 `private` 成员，但是，嵌套类的成员却不能被嵌套它的类直接访问。

### 3.8.1 类中定义的内部类

在类中直接定义的嵌套类的使用范围，仅限于这个类的内部，也就是说，A类里定义了一个B类，那么B为A所知，却不被A的外面所知。内部类的定义和普通类的定义没什么区别，它可以直接访问和引用它的外部类的所有变量和方法，就像外部类中的其他非 `static` 成员的功能一样，和外部类不同的是，内部类可以声明为 `private` 或 `protected`。

下面的程序说明了如何定义和使用一个内部类。名为 `Outer` 的类定义了一个实例变量 `outer_i`，一个 `test()` 方法，和一个名为 `Inner` 的内部类。

```

class Outer
{
    int outer_i = 100;
}

```

```

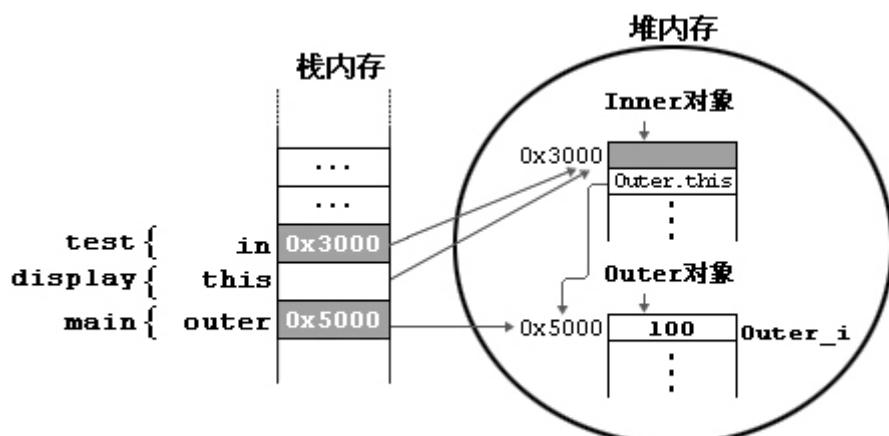
void test()
{
    Inner in = new Inner();
    in.display();
}
class Inner
{
    void display()
    {
        System.out.println("display: outer_i = " + outer_i);
    }
}
class InnerClassDemo
{
    public static void main(String[] args)
    {
        Outer outer = new Outer();
        outer.test();
    }
}

```

打印结果如下：

```
display: outer_i = 100
```

在程序中，内部类Inner定义在Outer类的范围之内。因此，在Inner类之内的display()方法可以直接访问Outer类的变量outer\_i。其实，在内部类对象保存了一个对外部类对象的引用，当内部类的成员方法中访问某一变量时，如果在该方法和内部类中都没有定义过这个变量，调用就会被传递给内部类中保存的那个外部类对象的引用，通过那个外部类对象的引用去调用这个变量，在内部类中调用外部类的方法也是一样的道理。上面的程序代码在内存中的布局，如图3.21所示：



注：`this`.变量不存在的情况下，传递给`Outer.this`.变量

图3.21

# F 指点迷津：

内部类使得程序代码更为紧凑，程序更具模块化。建议读者先自己试试如何将上面的Inner类改写成Outer类的外部去实现，如果你试图这样做了，并感觉到有些棘手的话，你就非常容易明白作者下面的结论了：当一个类中的程序代码要用到另外一个类的实例对象，而另外一个类中的程序代码又要访问第一个类中的成员，将另外一个类做成第一个类的内部类，程序代码就要容易编写得多，这样的情况在实际应用中非常之多！

一个内部类可以访问它的外部类的成员，但是反过来就不成立了。内部类的成员只有在内部类的范围之内是可知的，并不能被外部类使用。例如：

```
class Outer
{
    int outer_i = 100;
    void test()
    {
        Inner inner = new Inner();
        inner.display();
    }
    class Inner
    {
        int y = 10;
        void display()
        {
            System.out.println("display: outer_i = " + outer_i);
        }
        void showy()
        {
            System.out.println(y);
        }
    }
}
```

编译上面的程序，会出现如下错误：

```
G:\outer.java:19: cannot resolve symbol
symbol  : variable y
location: class Outer
System.out.println(y);
^
1 error
```

这里，y是作为Inner的一个实例变量来声明的，对于该类的外部它就是不可知的，因此不能被showy()使用。

如果用 static 修饰一个内部类，这个类就相当于是一个外部定义的类，所以 static 的内部类中可声明 static 成员，但是，非 static 的内部类中的成员是不能声明为 static 的。static 的内部类不能再使用外层封装类的非 static 的成员变量，这个道理不难想像！所以 static 嵌套类很少使用。

我们把前面程序中的 Inner 内部类声明为 static，来看看会出现什么样的错误：

```
class Outer
{
    int outer_i = 100;
    void test()
    {
        Inner in = new Inner();
        in.display();
    }
    static class Inner
    {
        void display()
        {
            System.out.println("display: outer_i = " + outer_i);
        }
    }
}
class InnerClassDemo
{
    public static void main(String[] args)
    {
        Outer outer = new Outer();
        outer.test();
    }
}
```

程序运行结果：

```
E:\TestOuter.java:13: non-static variable outer_i cannot be referenced from a
static context
    System.out.println("display: outer_i = " + outer_i);
                           ^
1 error
```

这段信息表明Outer类的非静态成员变量outer\_i不能被一个静态内部类的成员调用。

如果函数的局部变量（函数的形参也是局部变量），内部类的成员变量，外部类的成员变量重名，我们应该按下面的程序代码所使用的方式来明确指定我们真正要访问的变量。

```
public class Outer
{
    private int size;
    public class Inner
    {
        private int size;
        public void doStuff( int size)
        {
            size++; // 引用的是 doStuff 函数的形参
            this.size++; //引用的是 Inner 类中的成员变量
        }
    }
}
```

```
    Outer.this.size++; // 引用的 Outer 类中的成员变量
}
}
}
```

这些同名变量在内存中的分配情况如图3.22所示：

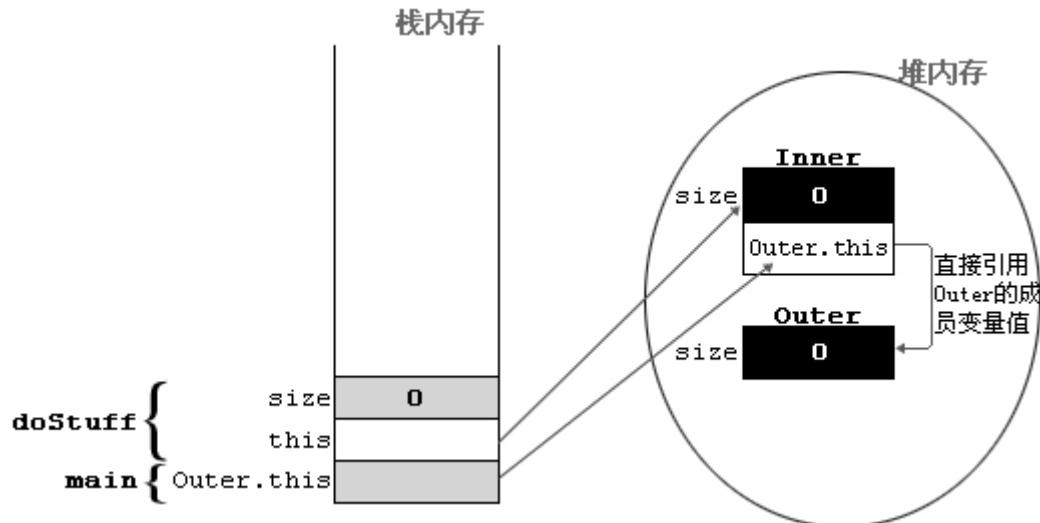


图3.22

### 3.8.2 内部类如何被外部引用

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，请看下面的程序：

```
class Outer
{
    private int size=10;
    public class Inner
    {
        public void doStuff()
        {
            System.out.println(++size);
        }
    }
}

public class TestInner
{
    public static void main( String[] args)
    {
        Outer outer = new Outer();
        Outer.Inner inner = outer.new Inner();
        inner.doStuff();
    }
}
```

程序中，内部类Inner被声明为public，在外部就可以创建其外部类Outer的实例对象，再通过Outer类的实例对象创建Inner类的实例对象，我们就可以使用Inner类的实例对象来调用内部类Inner中的方法了。

### 3.8.3 方法中定义的内部类

嵌套类并非只能在类里定义，也可以在几个程序块的范围之内定义内部类。例如，在方法中，或甚至在for循环体内部，都可以定义嵌套类，如下面的程序所示：

```
class Outer
{
    int outer_i = 100;
    void test()
    {
        for(int i=0; i<5; i++)
        {
            class Inner
            {
                void display()
                {
                    System.out.println("display: outer_i = "
                        + outer_i);
                }
            }
            Inner inner = new Inner();
            inner.display();
        }
    }
}

class InnerClassDemo
{
    public static void main(String args[])
    {
        Outer outer = new Outer();
        outer.test();
    }
}
```

该程序的这个版本的输出如下所示。

```
display: outer_x = 100
```

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：

```

class InOut
{
    String str= new String("Between");
    public void amethod(final int iArgs)
    {
        int it315;
        class Bicycle
        {
            public void sayHello()
            {
                System.out.println(str);
                System.out.println(iArgs);

            }//End of bicycle class
        }
    }//End of amethod
}

```

在内部类中的 sayHello 方法中，我们可以访问变量 iArgs 和 str，但不能访问 it315。

### 3.9 使用 Java 的文档注释

在前面的章节里我们介绍了 JDK 里主要的两个程序，也就是 Java 源文件编译程序 javac.exe 与 Java 字节码解释程序 java.exe。但是除了这两个程序之外，JDK 中还有一些其他的工具程序。本节中，我们为大家介绍一下 javadoc 的详细用法。

在使用javadoc工具之前，首先我们必须了解Java文档注释的作用。Java支持三种形式的注释，这已经在前面讲过，其中一种被称为文档注释，它以“`/**`”开始，以“`*/`”标志结束。文档注释提供将程序使用帮助信息嵌入到程序中的功能，开发者可以使用javadoc工具将这些信息取出，然后转换为HTML说明文档，由此可见文档注释提供了编写程序文档的便利方式。javadoc 工具生成的文档随处可见，比如Sun公司的Java API 文档库就是这么生成的。

本节中将通过几个简单的实例，来说明如何使用javadoc来生成文档注释文件。

我们首先来建立一个程序样本，在这里我们简要的列出程序内容，指出程序中存在的类及其方法，以及每个方法的功能，然后再为这个样本建立批注。最后再来利用javadoc程序来进行文档注释。程序样本如下：

```

//存储文件名为: engineer.java
01 import java.io.*;
02
03 public class engineer
04 {
05     public String Engineer_name;
06     public engineer(String name)
07     {
08         //初始化产生一个engineer，例如安排一个姓名.....
09     }

```

```

10  public int repairing(int sum,int alltime)
11  {
12      //进行修理的操作
13  }
14 }
```

上面的程序实例中声明了一个engineer的类，在类中又包含了两种方法：一个是用来产生engineer的方法，还有一个就是repairing的方法。

有了上面这样一个实例类之后，我们就利用Java提供的注释来为这个实例加上批注。其中我们可以利用Java提供的批注参数来标记一些特殊的属性及其相应的说明，这样，最后生成HTML说明文档时，就能看到相应的说明。特别提醒一点，在使用javadoc进行文档注释的时候，相应的信息和批注所对应的位置很重要！对于类的说明应在类定义之前，对于方法的说明应在方法的定义之前。我们接着就为上面的样本程序分段加上javadoc批注，顺便对其中的内容做相应解释。

在类的声明代码之前，应该加入类的注释，主要用来说明类的一些属性信息，如类名，编程者，编程时间，类封装的特性等等，在engineer类前的批注是这样的(批注位置：02行)

```

/***
 * Title: engineer 类<br>
 * Description: 一个模拟工程师的类<br>
 * Copyright: (c) 2003 www.it315.org<br>
 * Company: IT 人资讯交流网<br>
 * @author 张孝祥
 * @version 1.00
 */
```

说明一点，在这里<br>是HTML标签，表示回车换行，使生成的文档界面美观。而带有@标志的属性会自动换行，故结尾就省略了<br>。

@author<作者姓名>

用于类的说明，表示这个Java程序的作者。

@version<版本信息>

用于类的说明，表示了这个Java程序的开发版本。

接着进行产生engineer构造方法的批注（批注位置：05—06行之间）

```

/***
 * 这是engineer对象的构造函数
 * @param name engineer的名字
 */
```

其中，

@param<参数名称><参数说明>

用于方法的说明，表示方法所引入的参数，及其参数对应的说明。

最后，我们在来进行第二个方法repairing的批注（批注位置：09—10行之间）

```

/***
 * 这是repairing方法的说明
 * @param sum 需要修理的机器总数
 * @param alltime 需要修理的总时间
```

```
* @return 经过Repairing的数量
```

```
*/
```

其中，

@return<返回值说明>

用于方法的说明，表示此方法的返回值表示的意义。

到此，我们已经基本上了解了批注的格式与意义，那么就来看看样本程序完整的文档注释。

```
01 import java.io.*;
02 /**
03  * Title: engineer类<br>
04  * Description: 通过engineer类来说明Java中的文档注释<br>
05  * Copyright: (c) 2003 www.it315.org<br>
06  * Company: IT人资讯交流网<br>
07  * @author 张孝祥
08  * @version 1.00
09 */
10 public class engineer
11 {
12     public String Engineer_name;
13     /**
14      * 这是engineer对象的构造函数
15      * @param name engineer的名字
16      */
17     public engineer(String name)
18     {
19     }
20     /**
21      * 这是repairing方法的说明
22      * @param sum 需要修理的机器总数
23      * @param alltime 需要修理的总时间
24      * @return Repairing的数量
25      */
26     public int repairing(int sum, int alltime)
27     {
28     }
29 }
```

接下来就需要我们使用javadoc程序来产生这个样本程序的注释文件。

假设我们的样本程序engineer.java存放在C盘根目录下，我们在命令行窗口中进入engineer.java所在的目录。执行如下命令：

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

javadoc就会为我们生成这个类的说明文件。在上面的命令中，

-d用来制定说明文件存放的目录

engineer为说明文件所存放的目录名

-version代表要求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  
了解了上面的命令所起的作用，接着我们就来看看刚刚用javadoc程序生成的注释文档。进入我们刚才生成的engineer目录，打开index.html文件，如图3.23所示：



图 3.23

我们就可以看到利用javadoc程序所生成的类的文档说明，如图3.24所示：

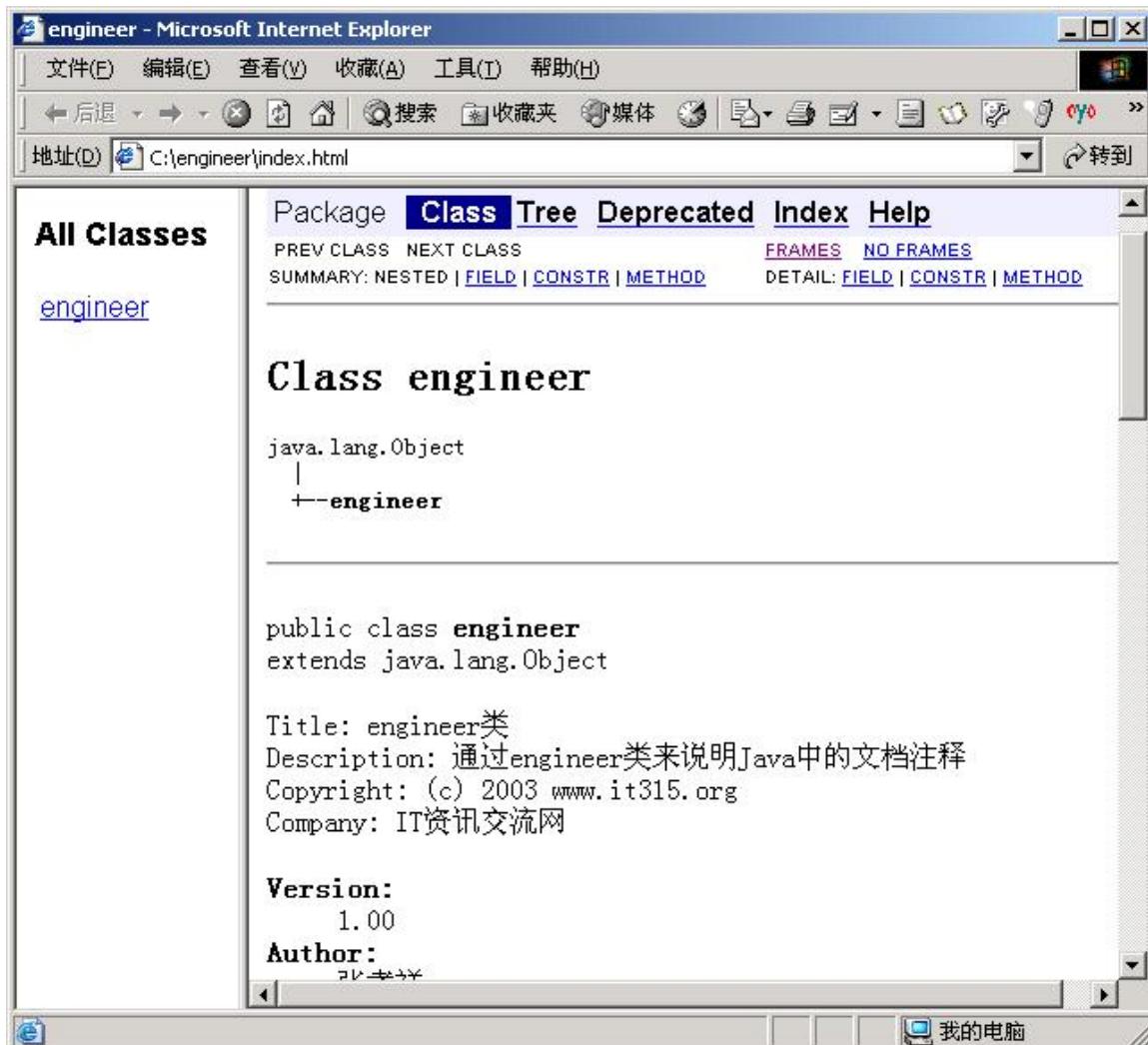


图3.24

你可以利用我们讲到的方法来熟悉 javadoc 的用法，更加详细的 javadoc 用法介绍，请读者在命令行窗口下运行“javadoc -help”来查看。

|                        |    |
|------------------------|----|
| 第3章 面向对象（上）.....       | 62 |
| 3.1 面向对象的概念.....       | 62 |
| 3.1.1 面向过程.....        | 62 |
| 3.1.2 面向对象.....        | 62 |
| 3.2 类与对象.....          | 63 |
| 3.2.1 类的定义.....        | 63 |
| 脚下留心：成员变量与局部变量同名的问题    |    |
| 3.2.2 对象的产生与使用.....    | 64 |
| 3.2.3 对象的比较.....       | 67 |
| 多学两招：怎样比较两个数组对象的内容是否相等 |    |
| 3.2.4 匿名对象.....        | 70 |
| 3.2.5 实现类的封装性.....     | 70 |
| 指点迷津：良好的封装             |    |
| 多学两招：Private 的试验       |    |

|                           |     |
|---------------------------|-----|
| 3.3 构造函数.....             | 73  |
| 3.3.1 构造函数的定义与作用.....     | 73  |
| 脚下留心：如何理解构造函数的返回值         |     |
| 3.3.2 构造方法的重载.....        | 74  |
| 3.3.3 构造方法的一些细节.....      | 77  |
| 3.4 this 引用句柄.....        | 79  |
| 3.5 与垃圾回收有关的知识.....       | 83  |
| 3.5.1 finalize 方法.....    | 83  |
| 3.5.2 System.gc 的作用 ..... | 84  |
| 3.6 函数的参数传递.....          | 85  |
| 3.6.1 基本数据类型的参数传递.....    | 85  |
| 3.6.2 引用数据类型的参数传递.....    | 86  |
| 3.7 static 关键字 .....      | 90  |
| 3.7.1 静态变量.....           | 90  |
| 3.7.2 静态方法.....           | 92  |
| 3.7.3 静态代码块.....          | 93  |
| 3.7.4 单态设计模式.....         | 94  |
| 3.7.5 理解 main 方法的语法.....  | 94  |
| 3.8 内部类 .....             | 95  |
| 3.8.1 类中定义的内部类.....       | 95  |
| 指点迷津：何时使用内部类              |     |
| 3.8.2 内部类如何被外部引用.....     | 99  |
| 3.8.3 方法中定义的内部类.....      | 100 |
| 3.9 使用 Java 的文档注释.....    | 101 |

# 第4章 面向对象（下）

## 4.1 类的继承

面向对象的重要特色之一就是能够使用以前建造的类的方法和属性。通过简单的程序代码来建造功能强大的类，会节省很多编程时间，而且更为重要的是，这样做可以减少代码出错的机会。类的继承讲的就是这方面的问题。

### 4.1.1 继承的特点

我们还是通过一个实际应用问题，来引出类的继承这个问题的讲解。

我们常常在编程中遇到下面的情况：

```
public class Person
{
    public String name;
    public int age;
    public String getInfo() {...}
}

public class Student
{
    public String name;
    public int age;
    public String school;
    public String getInfo() {...}
    public String study(){...}
}
```

在上面的程序中，我们先定义了一个 Person 类来处理个人信息，接着又定义一个 Student 类来处理学生信息，结果发现 Student 类中包含了 Person 类的所有属性和方法。

针对这种情况，Java 为我们引入了继承这个概念，我们只要表明类 Student 继承了类 Person 的所有属性与方法，我们就不用在类 Student 中重复书写类 Person 中的代码了，更确切的说是简化了类的定义。我们是通过 extends 关键字来表明类 Student 具有类 Person 的所有属性与方法，如果让 Student 类来继承 Person 类，Person 类里面的所有可继承的属性和方法（后面我们会讲到什么是可继承的，什么是受限制的），都可以在 Student 类里面使用了，也就是说 Student 这个类继承了 Person 类，拥有了 Person 类所有的特性，除此之外还有一些自己的特性，如：学生有学校名和学习的动作。因此，我们就说 Person 是 Student 的父类（也叫基类或超类），Student 是 Person 的子类。例如上面的两个类，我们就可以简写成下面的代码：

```
public class Person
{
    public String name;
    public int age;
    public String getInfo() {...}
}

public class Student extends Person
```

```
{  
    public String school;  
    public String study(){...}  
}
```

完整代码如下：

程序清单：Student.java

```
class Person  
{  
    public String name;  
    public int age;  
    public Person(String name,int age)  
    {  
        this.name=name;  
        this.age=age;  
    }  
    public Person() //如果你不写这个构造函数，看看对类 Student 有什么影响。  
    {  
    }  
    public void getInfo()  
    {  
        System.out.println(name);  
        System.out.println(age);  
    }  
}  
class Student extends Person  
{  
    public void study()  
    {  
        System.out.println("Studding");  
    }  
  
    public static void main(String[] args)  
    {  
        Person p=new Person();  
        p.name="person";  
        p.age=30;  
        p.getInfo();  
  
        Student s=new Student();  
        s.name="student";  
        s.age=16;  
        s.getInfo();  
        s.study();  
    }  
}
```

}

要在以前的类上构造新类，必须要在新类的声明中扩展以前的类。通过扩展一个超类，你可以得到这个类的一个拷贝，并可以在这个基础上添加新的属性和方法。如果你对这个新类并不做什么添加工作，那么它的工作情况会与超类完全相同。新类中会含有超类所声明或继承的所有属性和方法。在类的继承中，有这样的一些细节问题：

- 1). 通过继承可以简化类的定义，我们已经在上面的例子中了解到了。
- 2). Java 只支持单继承，不允许多重继承。在 Java 中，一个子类只能有一个父类，不允许一个类直接继承多个类，但一个类可以被多个类继承，如类 X 不可能既继承类 Y 又继承类 Z。
- 3). 可以有多层继承，即一个类可以继承某一个类的子类，如类 C 继承了类 A，类 B 又可以继承类 C，那么类 C 也间接继承了类 A。这种应用如下所示：

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

4). 子类继承父类所有的成员变量和成员方法，但不继承父类的构造方法。在子类的构造方法中可使用语句 `super(参数列表)` 调用父类的构造方法。如：我们为 `Student` 类增加一个构造方法，在这个构造方法中我们用 `super` 明确指定调用父类的某个构造方法。

```
class Student extends Person
{
    public Student(String name,int age,String school)
    {
        super(name,age);
        this.school=school;
    }
}
```

5). 如果子类的构造方法中没有显式地调用父类构造方法，也没有使用 `this` 关键字调用重载的其它构造方法，则在产生子类的实例对象时，系统默认调用父类无参数的构造方法。也就是，在下面的类 B 中定义的构造方法中，写不写 `super()` 语句效果是一样的。

```
Public class B extends A
{
    public B()
    {
        super(); //有没有这一句，效果都是一样的。
    }
}
```

如果子类构造方法中没有显式地调用父类构造方法，而父类中又没有无参数的构造方法（需要再次说明的是，如果父类没有显式的定义任何构造方法，系统将自动提供一个默认的没有参数的构造方法，这还是等于父类中有无参数的构造方法的），则编译出错。读者将前面的 `Person` 类中的无参数的构造方法注释掉，重新编译 `Student` 类，就能够看到这个错误效果了。所以，我们在定义类时，只要定义了有参数的构造方法，通常都还需要定义一个无参数的构造方法。

## 4.1.2 子类对象的实例化过程

对于许多 Java 老手来说，子类对象的实例化过程也不见得非常清楚，你可能并不需要完全了解子类对象的实例化过程，但了解后还是有好处的。

对象中的成员变量的初始化是按下列步骤进行的：

- 1). 分配成员变量的存储空间并进行默认的初始化，就是用 new 关键字产生对象后，对类中的成员变量按第三章的表 3.1 中的对应关系对对象中的成员变量进行初始化赋值。
- 2). 绑定构造方法参数，就是 new Person (实际参数列表) 中所传递进的参数赋值给构造方法中的形式参数变量。
- 3). 如有 this() 调用，则调用相应的重载构造方法（被调用的重载构造方法又从步骤 2 开始执行这些流程），被调用的重载构造方法的执行流程结束后，回到当前构造方法，当前构造方法直接跳转到步骤 6 执行。
- 4). 显式或隐式追溯调用父类的构造方法(一直到 Object 类为止，Object 是所有 Java 类的最顶层父类，在本章后面部分有详细讲解)，父类的构造方法又从步骤 2 开始对父类执行这些流程，父类的构造方法的执行流程结束后，回到当前构造方法，当前构造方法继续往下执行。
- 5). 进行实例变量的显式初始化操作，也就是执行在定义成员变量时就对其进行赋值的语句，如：

```
public Student extends Person
{
    String school = "it315"; // 显式初始化
    .....
}
```

将“it315”赋值给 school 成员变量。

- 6). 执行当前构造方法的方法体中的程序代码，如

```
public Student extends Person
{
    public Student(String name,int age,String school)
    {
        super(name,age);
        this.school=school;
    }
}
```

这一步将执行 this.school=school；这条语句，其中用到的 super() 或 this() 方法调用语句已在前面的步骤中执行过，这里就不再执行了。注意区别刚才所说的 this() 方法调用语句与 this.school=school 的区别，前者指调用其他的构造方法，后者是一个普通的赋值语句。

为了便于读者直观地看到子类对象的实例化过程，我将上面的流程用图 4.1 进行了重复描述。

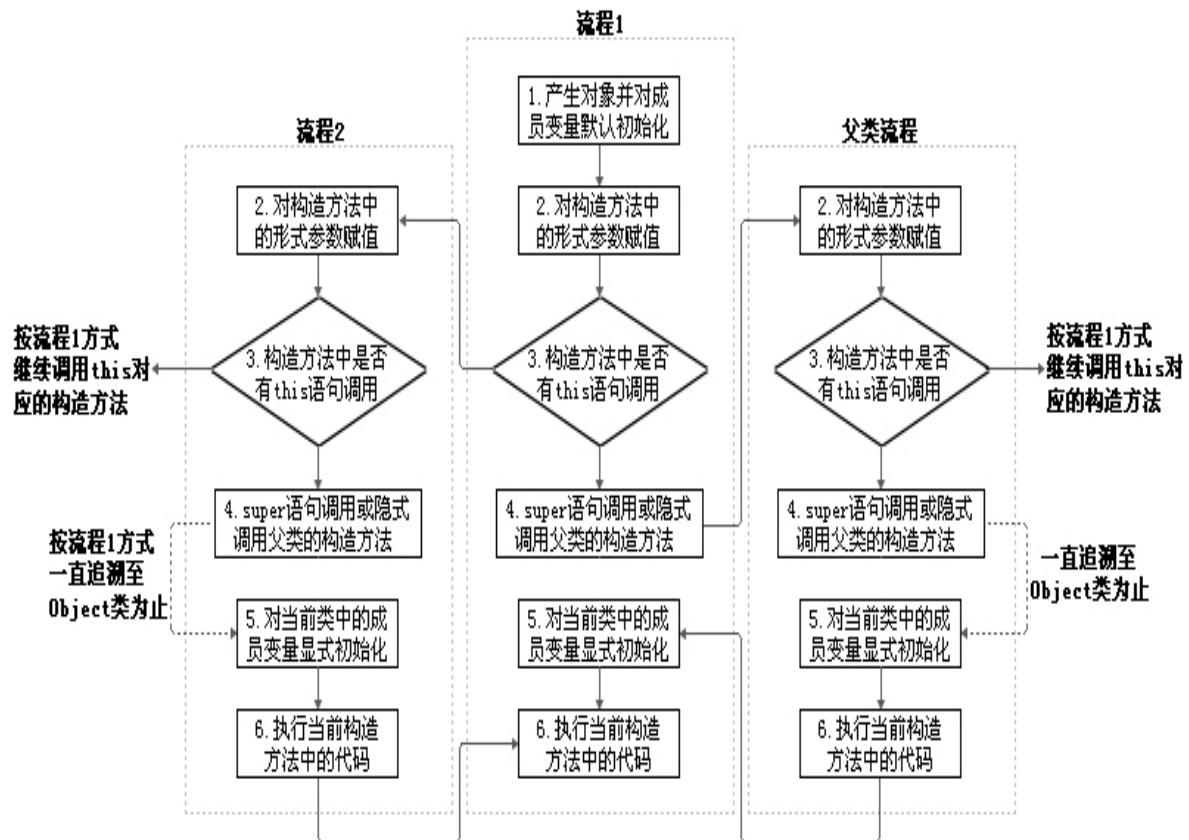


图 4.1

回过头来想一想：

- 1). 为什么 super(...)  
和 this(...)  
调用语句不能同时在一个构造函数中出现？
- 2). 为什么 super(...)  
或 this(...)  
调用语句只能作为构造函数中的第一句出现？

对照对象初始化实例变量过程，我们就发现这两种情况都违背上面的过程，所以读者应该明白上面两个问题的原因了。

### 4.1.3 覆盖父类的方法

在子类中可以根据需要对从父类中继承来的方法进行改造----方法的覆盖（也叫重写）。覆盖方法必须和被覆盖方法具有相同的方法名称、参数列表和返回值类型。

例如前面那个 Student 程序，它继承了 Person 类的 getInfo 方法，这个继承到的方法只能打印出学生的 name 和 age，不能打印出学生专有的信息，比如学校的名称等，这时我们就应该在类 Student 中重新编写一个 getInfo 方法，这就是方法的覆盖。程序修改后如下：

程序清单：Student1.java

```

class Person
{
    public String name;
    public int age;
    public void getInfo()
    {
        System.out.println(name);
    }
}

```

```

        System.out.println(age);
    }
}

class Student extends Person
{
    String school=new String();
    public void study()
    {
        System.out.println("Studding");
    }
    public void getInfo()
    {
        super.getInfo();
        System.out.println(school);
    }

    public static void main(String[] args)
    {
        Person p=new Person();
        p.name="person";
        p.age=30;
        p.getInfo();

        Student s=new Student();
        s.name="student";
        s.age=16;
        s.school="清华大学";
        s.getInfo();
        s.study();
    }
}

```

运行结果：

```

person
30
student
16
清华大学
Studding

```

从以上运行结果可以看出，`p.getInfo()`这一句中所用到的方法是父类 `Person` 的，而 `s.getInfo()` 这一句用的方法却是子类 `Student` 的。如果在子类中想调用父类中的那个被覆盖的方法，我们可以用 `super.` 方法的格式，如程序中的 `super.getInfo()`；

**注意：** 覆盖方法时，不能使用比父类中被覆盖的方法更严格的访问权限，如：父类中的方法是 `public` 的，子类的方法就不能是 `private` 的。关于访问权限中更多的知识，我们将在本章结尾时详细讲解，在这里大家只要有个印象就行了。

#### 4.1.4 final 关键字

- ü 在 Java 中声明类、属性和方法时，可使用关键字 `final` 来修饰。
- ü `final` 标记的类不能被继承。
- ü `final` 标记的方法不能被子类重写。
- ü `final` 标记的变量(成员变量或局部变量)即成为常量，只能赋值一次。

如：

```
final int x=3;  
x=4; //出错  
final 标记的成员变量必须在声明的同时或在该类的构造方法中显式赋值，然后才能使用。  
class Test  
{  
    final int x=3;  
}  
或者：  
class Test  
{  
    final int x;  
    Test()  
    {  
        x=3;  
    }  
}
```

- ü 方法中定义的内置类只能访问该方法内的 `final` 类型的局部变量，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，这种情况的应用，我们已在第三章中讲过。将一个形参定义成 `final` 也是可以的，这就限定了我们在方法中修改形式参数的值。

#### & 多学两招：

我们已经知道 `final` 标记的变量(成员变量或局部变量)即成为常量，只能赋值一次，但这个“常量”也只能在这个类的内部使用，不能在类的外部直接使用。

当我们用 `public static final` 共同标记常量时，这个常量就成了全局的常量。而且这样定义的常量只能在定义时赋值，即使在构造函数里面也不能对其进行赋值。如：

```
class Xxx  
{  
    public static final int x=3;  
    ....  
}
```

在 Java 中，我们看到的常量的定义方法，总是用 `public static final` 的组合方式进行标识。Java 中的全局常量也放在一个类中定义，给我们的使用带来了很大的方便，譬如，我们在程序中想使用最大的浮点小数，我们知道有个 `Float` 类封装了浮点小数的操作，我们就很容易想到在 `Float` 类的文档帮助中去查找这个常量的具体英文拼写。

## 4.2 抽象类与接口

### 4.2.1 抽象类

Java 中可以定义一些不含方法体的方法，它的方法体的实现交给该类的子类根据自己的情况去实现，这样的方法就是抽象方法，包含抽象方法的类就叫抽象类。一个抽象类中可以有一个或多个抽象方法。

抽象方法必须用 `abstract` 修饰符来定义，任何带有抽象方法的类都必须声明为抽象类。

#### 抽象类定义规则

- ü 抽象类必须用 `abstract` 关键字来修饰；抽象方法也必须用 `abstract` 来修饰。
- ü 抽象类不能被实例化，也就是不能用 `new` 关键字去产生对象。
- ü 抽象方法只需声明，而不需实现。
- ü 含有抽象方法的类必须被声明为抽象类，抽象类的子类必须覆盖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。

#### 抽象方法的写法：

```
abstract 返回值类型 抽象方法( 参数列表 );
```

#### 抽象类和抽象方法的例子：

```
abstract class A
{
    abstract int aa(int x,int y);
```

} 注意：含有抽象方法的类肯定是抽象类，抽象类中的某个方法不一定是抽象的。

### 4.2.2 接口(interface)

如果一个抽象类中的所有方法都是抽象的，我们就可以将这个类用另外一种方式来定义，也就是接口定义。接口是抽象方法和常量值的定义的集合，从本质上讲，接口是一种特殊的抽象类，这种抽象类中只包含常量和方法的定义，而没有变量和方法的实现。

下面是一个接口定义例子：

```
public interface Runner
{
    int ID = 1;
    void run();
}
```

在接口 `Runner` 的定义中，即使我们没有显示的将其中的成员用 `public` 关键字标识，但这些成员都是 `public` 访问类型的。接口里的变量默认是用 `public static final` 标识的，所以，接口中定义的变量就是全局静态常量。

我们可以定义一个新的接口用 `extends` 关键字去继承一个已有的接口，我们也可以定义一个类用 `implements` 关键字去实现一个接口中的所有方法，我们还可以去定义一个抽象类用 `implements` 关键字去实现一个接口中定义的部分方法。

```
interface Animal extends Runner
{
    void breathe();
```

```

}

class Fish implements Animal
{
    public void run()
    {
        System.out.println("fish is swimming");
    }

    public void breathe()
    {
        System.out.println("fish is bubbling");
    }
}

abstract LandAnimal implements Animal
{
    public void breathe()
    {
        System.out.println("LandAnimal is breathing");
    }
}

```

在上面的几个类和接口的定义中，`Animal` 是一个接口，`Animal` 接口具有 `Runner` 接口的特点，是对 `Runner` 接口的一种扩展。`Fish` 是一个类，具有 `Animal` 接口中定义的所有方法，必须实现 `Animal` 接口中的所有方法（包括从 `Runner` 接口继承到的方法）。`LandAnimal` 是一个抽象类，它实现了 `Animal` 接口中的 `breathe` 方法，但没有实现 `Run` 方法，`Run` 方法在 `LandAnimal` 中就成了一个抽象方法，所以 `LandAnimal` 应是一个抽象类。

在 Java 中，设计接口的目的是为了让类不必受限于单一继承的关系，而可以灵活地同时继承一些共有的特性，从而达到多重继承的目的，并且避免去了 C++ 中多重继承的复杂关系所产生的问题。多继承的危险性在于一个类有可能继承了同一个方法的不同实现，对接口来讲决不会发生这种情况，因为接口没有任何实现。

一个类可以在继承一个父类的同时，实现一个或多个接口，`extends` 关键字必须位于 `implements` 关键字之前，如我们可以这样定义类 `Student`。

```

class Student extends Person implements Runner
{
    .....
    public void run()
    {
        System.out.println("the student is running");
    }
    .....
}

```

下面是一个类实现多个接口的例子，我们在程序中再定义一个 `Flyer` 接口。

```

interface Flyer
{
    void fly();
}

```

```

class Bird implements Runner , Flyer
{
    public void run()
    {
        System.out.println("the bird is running");
    }
    public void fly()
    {
        System.out.println("the bird is flying");
    }
}

```

下面是关于接口中定义的常量的举例，

```

class TestFish
{
    public static void main(String args[])
    {
        Fish f =new Fish();
        int j=0;
        j=Runner.ID;//“类名.静态成员”的格式
        j=f.ID; // “对象名.静态成员”
        /*这两行粗体都正确，一个用类的实例对象，一个用接口类。/
        f.ID=2;//出错，不能为 final 变量重新赋值
    }
}

```

下面是对接口的实现及特点的小结：

- ü 实现一个接口就是要实现该接口的所有方法（抽象类除外）。
- ü 接口中的方法都是抽象的。
- ü 多个无关的类可以实现同一个接口，一个类可以实现多个无关的接口。

## 4.3 对象的多态性

### 4.3.1 对象的类型转换

前面第二章我们讲到了基本数据类型变量的类型转换问题，其实对象类型转换也差不多是一个道理。

1). 子类转换成父类

我们来看看下面的程序代码：

程序清单：C.java

```

class A
{
    public void func1()
    {
        System.out.println("A func1 is calling");
    }
}

```

```

public void func2()
{
    func1();
}
}

class B extends A
{
    public void func1()
    {
        System.out.println("B func1 is calling");
    }

    public void func3()
    {
        System.out.println("B func3 is calling");
    }
}

class C
{
    public static void main(String [] args)
    {
        B b=new B();
        A a = b;
        callA(a);
        callA(new B());
    }

    public static void callA(A a)
    {
        a.func1();
        a.func2();
    }
}

```

编译没有错误，我们发现，编译器能够自动将类 B 的实例对象 b 直接赋值给 A 类的引用类型变量，也就是子类能够自动转换成父类类型。另外，我们也看到了，程序可以直接创建一个类 B 的实例对象，传递给需要类 A 的实例对象作参数的 CallA 方法，在参数传递的过程中发生了隐式自动类型转换。子类能够自动转换成父类的道理非常容易理解，“男人”是“人”的子类，要把一个“男人”对象当作“人”去用，天经地义！

## 2). 父类转换成子类

如果我们在编程时，知道 CallA 方法中传递的形式参数 a 实际上就是子类 B 的一个引用对象，我们想在 CallA 方法中调用子类的特有方法，该如何做呢？

```

public static void CallA(A a)
{
    a.func1();
    a.func2();
    a.func3();
}

```

```
}
```

编译有问题，因为对编译器来说，它只分析程序语法，它只知道变量 a 的引用类型是类 A，而类 A 又没有 func3 这个方法，所以，编译无法通过。

我们将程序代码作如下修改：

```
public static void CallA(A a)
{
    B b=a;
    b.func1();
    b.func2();
    b.func3();
}
```

编译还是有问题，因为编译器是不能将父类对象自动转换成子类的。父类对象不能自动转换成子类的道理也很容易理解，要把一个“人”直接当作“男人”去用，是有点说不过去的。

最后更改如下：

```
public static void CallA(A a)
{
    B b= (B) a;
    b.func1();
    b.func2();
    b.func3();
}
```

编译就通过了。读者看到了，我们把对象 a 强制转换成了 B 类的引用类型，编译器对该对象进行了强制转换。我们通过下面的讲解来帮助读者了解强制类型转换。

比如有个函数为“去叫一个人来”，定义该函数时，其返回值只能是“人”，但实际来的“人”，不是“男人”，就是“女人”。即使叫来的是一个“女人”，但编译器只能根据函数定义的语法去确定来的是“人”，没法确定运行时返回的是“男人”，还是“女人”，如果将该函数的返回值直接赋给一个“女人”类型的变量，编译时将不会通过。程序员是很清楚运行时返回的是“男人”，还是“女人”的，他可以告诉严格执行语法检查的编译器：“我知道来的这个人确实是一个女人，你把他当作女人去处理吧，出了问题，我负责！”。这个过程就是强制类型转换，目的是让编译器进行语法检查时开点后门，放你过关，强制类型转换并不是要对内存中的对象大动手术，不是要将“男人”变成“女人”。

在类型转换时，程序员要对转换完的后果负责，要确保在内存中存在的对象本身确实可以被看成那种要转换成的类型，如果来的“人”是男人，我们将其转换成“女人”后，编译能够通过，但程序运行时将会出错，虽然骗过了编译器，但你没法调用“女人”独有的方法（比如怀孕），所以运行时就会产生类型转换异常。强制类型转换的前提是程序员提前就知道要转换的父类引用类型对象的本来面目确实是子类类型的。

作者在编码和调试时，总是用意境的方式，仿佛看到变量或对象在内存中的真实布局和状态，以及是如何进行转换的，这样编码时比较容易一气呵成，极少犯错。

有些时候，我们不能提前就知道这个“人”是不是“女人”，能不能事先判断一下呢？也就是这个父类的引用变量是否真的指向了要转换成的那个子类对象呢？我们用 instanceof 操作符就可以解决这个问题。

### 3). instanceof 操作符

可以用 `instanceof` 判断是否一个类实现了某个接口，也可以用它来判断一个实例对象是否属于一个类。还是用上面的代码来举例：

```
public static void CallA(A a)
{
    if(a instanceof B)
    {
        B b=(B)a;
        b.func1();
        b.func2();
        b.func3();
    }
    else
    {
        a.func1();
        a.func2();
    }
}
```

这样改的目的是要判断一下传入的“人”，是不是属于“女人”这个类的。如果是，则强制类型转换，如果不是就不转换。

#### **instanceof** 的用法：

对象 `instanceof` 类（或接口）

它的返回值是布尔型的，或真（`true`），或假（`false`）。

大家只要记住：一个“男人”肯定也是“人”，一个“人”却不一定“男人”的道理，就非常容易理解父类和子类之间的转换关系了。

### 4.3.2 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` 衍生出来的，所以 `Object` 的方法适用于所有类。

```
public class Person
{
    ...
}
```

等价于：

```
public class Person extends Object
{
    ...
}
```

`Object` 中有一个 `equals` 方法，用于比较两个对象是否相等，默认值为 `false`。由于上面的继承特性，我们就可以在我们的类中使用这个 `equals` 方法，但如果我们要比较的是我们自己类中的对象，结果就不一定准确了。因此，就必须覆盖掉 `Object` 类的 `equals` 方法。

请看下面的例子，在本类中，我们认为姓名和年龄都相同的两个学生是同一个人。所以，

我们用自己的 equals 覆盖掉了 Object 类中的 equals 方法，这样才达到了我们的编程的目的。

程序清单：Student.java

```
class Student
{
    String name;
    int age;
    boolean equals(Object obj)
    {
        Student st=null;
        if(obj instanceof Student)
            st = (Student)obj;
        else
            return false;
        if(st.name==this.name && st.age==this.age)
            return true;
        else
            return false;
    }

    public static void main(String[] args)
    {
        Student p=new Student();
        Student q=new Student();
        p.name="xyz";
        p.age=13;
        q.name="xyz";
        q.age=13;
        if(p.equals(q))
            System.out.println("p 与 q 相等");
        else
            System.out.println("p 与 q 不等");
    }
}
```

### 4.3.3 面向对象的多态性

在前面讲到的子类自动转换成父类的例子中，我们的调用程序是这么写的：

```
class C
{
    public static void main(String [] args)
    {
        B b=new B();
        A a = b;
        callA(a);
        callA(new B());
    }
}
```

```

    }
public static void callA(A a)
{
    a.func1();
    a.func2();
}
}

```

程序运行结果是：

```

B func1 is calling
B func1 is calling
B func1 is calling
B func1 is calling

```

尽管在 callA(A a)方法定义中，我们从字面上看是通过类 A（父类）的引用类型变量 a，去调用其中的 func1()，但实际上执行的是类 B（子类）中的 func1 方法，这是因为我们实际传递进来的对象确实是子类 B 的实例对象，所以程序调用的是类 B 中的 func1 方法。如果我们再做一个类 A 的子类 D，让 D 也覆盖类 A 中的 func1()方法，然后产生一个类 D 的实例对象，并传递给 callA(A a)方法，程序在 callA 方法中的 a.func1() 调用，也将是类 D 中的 func1 方法。这就好比我们有一个“叫某人来吃饭”的函数，在该函数内部，我们是这样写的：

```

void 叫某人来吃饭(人 p)
{
    p.吃饭();
}

```

当叫来一个中国人时，我们看到的是用筷子在吃饭，但是当叫来的是一个美国人时，我们看到的就是另外一番景象了，用的是叉子和小刀。同一段程序代码（单指“叫某人来吃饭”这个函数），却有两种截然不同的结果，这就是面向对象的多态性。可见，多态性有如下特点：

- 1). 应用程序不必为每一个派生类（子类）编写功能调用，只需要对抽象基类进行处理即可。这一招叫“以不变应万变”，可以大大提高程序的可复用性。
- 2). 派生类的功能可以被基类的方法或引用变量调用，这叫向后兼容，可以提高程序的可扩充性和可维护性。以前写的程序可以被后来程序调用不足为奇，现在写的程序（如 callA 方法）能调用以后写的程序（以后编写的一个类 A 的子类，如类 D）就很了不起了。

我们在深入分析多态性的另外一个方面，细心的读者可能已经注意到：在前面的程序中，子类 B 并没有覆盖父类 A 的 func2() 方法，直接从类 A 中继承了该方法，类 A 中的 func2() 方法调用的是类 A 自己的 func1 方法，但上面的结果显示，类 B 从类 A 继承来的 func2 里面调用的 func1 变成了子类 B 的 func1。还记得我说过，编程序的时候，眼睛就盯着内存去想问题，而不是盯着程序代码，肯定能想明白这个问题。

## \$ 独家见解：

接口在面向对象的设计与编程中应用的非常广泛，特别是实现软件模块间的插接方面有着巨大优势。其实，我们生活中也经常碰到接口的概念，大家想一想，我们在北京中关村的电子市场随便挑选了一块计算机主板和一块 PCI 卡（网卡，声卡等），结果，这块 PCI 卡能够很好地用到这块主板上，大家想过没有，这是什么原因造成的呢？主板厂商和 PCI 卡厂商是同一家吗？他们相互认识吗？答案是否定的。但他们都知道同一个标准，那就是 PCI 规范。做 PCI 卡的厂商严格按照 PCI 规范去实现他们的 PCI 卡，也就是与主板插槽的连接处是固定的格式，包括卡

的尺寸与连接电路线的排列顺序，但卡内部如何制造，就无所谓了，可以做成网卡，也可以做成声卡。做主板的厂商也要知道 PCI 规范，他们只要保留一个能使用 PCI 卡的插槽，也就是按照 PCI 卡的尺寸与连接电路线的排列顺序去使用可能插进来的 PCI 卡，而不必知道 PCI 卡的内部具体实现。

我们通过编写一段程序来模拟上述过程的实现，PCI 卡中的每个方法名称（相当于 PCI 卡的尺寸与连接电路线的排列顺序）必须是固定的，主板才能根据自己想执行的命令找到 PCI 卡中对应的方法，PCI 卡也必须具有主板可能调用到的所有命令方法。这正是“调用者和被调用者必须共同遵守某一限定，调用者按照这个限定进行方法调用，被调用者按照这个限定进行方法实现”的应用情况，在面向对象的编程语言中，这种限定就是通过接口类来表示的，主板和各种 PCI 卡就是按照 PCI 接口进行约定的。

程序清单：Interface.java

```
interface PCI
{
    void start();
    void stop();
}

class NetworkCard implements PCI
{
    public void start()
    {
        System.out.println("Send ...");
    }

    public void stop()
    {
        System.out.println("Network Stop.");
    }
}

class SoundCard implements PCI
{
    public void start()
    {
        System.out.println("Du du...");
    }

    public void stop()
    {
        System.out.println("Sound Stop.");
    }
}

class MainBoard
{
    public void usePCICard(PCI p)
    {
        p.start();
        p.stop();
    }
}
```

```

        }
    }

class Assembler
{
    public static void main(String [] args)
    {
        MainBoard mb=new MainBoard();
        NetworkCard nc=new NetworkCard();
        mb.usePCICard(nc);
        SoundCard sc=new SoundCard();
        mb.usePCICard(sc);
    }
}

```

在上面的程序代码中，类 Assembler 就是计算机组装者，他买了一块主板 mb 和一块网卡 nc，一块声卡 sc，无论是网卡还是声卡，他们都使用的是主板的 usePCICard 方法。由于 NetworkCard 与 SoundCard 都是 PCI 接口的子类，所以，他们的对象能直接传递给 usePCICard 方法中的 PCI 接口的引用变量 p，在参数传递的过程中发生了隐式自动类型转换。

通过这个例子，读者应该明白了一个类必须实现接口中的所有方法的原因，因为调用者有可能会用到接口中的每个方法，所以，被调用者必须实现这些方法。

**思考题：**如果 SoundCard 类中具有 PCI 接口中的所有方法，但没有明确声明它实现了 PCI 接口，SoundCard 的对象实例能不能直接传递给 MainBoard 的 usePCICard 方法使用？

编译器并不能根据一个类中有哪些方法，就知道它是某个类的子类的，编译器只能从 extends 和 implements 关键字上了解。MainBoard 的 usePCICard 方法要求的是 PCI 类型的对象，虽然 SoundCard 中具有 PCI 接口中的所有方法，但编译器并不能知道它就是 PCI 接口的子类对象，编译器又是严格进行语法检查的，所以我们不能将 SoundCard 的对象实例直接传递给 MainBoard 的 usePCICard 方法。即使我们将 SoundCard 的实例对象强制转换成 PCI 类型，编译时能够通过，但运行时会产生**类型转换异常**。

#### 4.3.4 匿名内部类

第三章中我们已经讲到了内部类，印象不深的读者请回到第三章复习一下，不然在这里你可能会更迷惑的。我们首先来看看内部类继承另外一个类的应用情况。下面的例子在类 Outer 中定义了一个内部类 Inner，Inner 类又继承了类 A。

```

abstract class A
{
    abstract public void fun1();
}

class Outer
{
    public static void main(String [] args)
    {
        class Inner extends A
        {
            public void fun1()

```

```

    {
        System.out.println("implement for fun1");
    }
}

new Outer().callInner(new Inner());
}

public void callInner(A a)
{
    a.fun1();
}
}

```

在上面的例子中，我们在 `Outer.main` 方法中调用了 `Outer.callInner` 方法，传递给 `callInner` 方法的参数要求是一个实现了类 A 中的方法的子类对象，为此，我们在 `Outer.main` 方法中定义的一个继承了类 A 的内部类 `Inner`，类 `Inner` 仅在此被使用了一次。内部类可以声明是抽象类或是一个接口，它可以被另外一个内部类来继承或实现。内部类可以继承外部类，也可以用 `final` 关键字修饰。

对于上面的程序，我们可以简写，在调用 `callInner` 方法时，不用事先定义类 `Inner`，我们可以在给 `callInner` 方法传递参数时，临时创建一个类 A 的匿名子类的实例对象。

```

class Outer
{
    public static void main(String [] args)
    {
        new Outer().callInner(new A()
        {
            public void fun1()
            {
                System.out.println("implement for fun1");
            }
        });
    }

    public void callInner(A a)
    {
        a.fun1();
    }
}

```

上面的写法，初看起来，有些复杂，上面的程序相当于定义了一个类 A 的子类，但没给这个子类起名字，接着又创建了这个子类的一个实例对象，这两步需要合在一块写。读者只要按下面的步骤去走，这个过程就很简单了。

首先，在 `callInner()` 方法调用中，写上 `new A(){};`，也就是在 `new A()` 的后面加上一对大括号，就表示要产生一个类 A 的一个匿名子类的实例对象，并传递给 `callInner` 方法。写完的效果如下：

```
new Outer().callInner(new A(){});
```

匿名子类的所有实现代码（包括成员方法和成员变量）都要在那对大括号间增加。

接着，为了有个直观的程序代码层次，我们先用回车键将那对大括号间分开成两行，效果如下：

```
new Outer().callInner(new A()  
{  
});
```

最后，在那对大括号间增加匿名子类的所有实现代码，效果如下：

```
new Outer().callInner(new A()  
{  
    public void fun1()  
    {  
        System.out.println("implement for fun1");  
    }  
});
```

就这么简单，见多了，用多了，你就非常习惯这种写法了，通过这种方式定义的类就是匿名内部类。

## 4.4 异常

### 4.4.1 了解异常

异常定义了程序中遇到的非致命的错误，而不是编译时的语法错误，如程序要打开一个不存在的文件、网络连接中断、操作数越界、装载一个不存在的类等。

我们先来看看下面的程序代码吧：

```
public class TestException  
{  
    public static void main(String [] args)  
    {  
        int result = new Test().devide( 3, 0 );  
        System.out.println("the result is" + result );  
    }  
}  
class Test  
{  
    public int devide(int x, int y)  
    {  
        int result = x/y;  
        return x/y;  
    }  
}
```

编译运行上面的程序，将出现如下错误：

```
Exception in thread "main" java.lang.ArithmetiсException: / by zero  
at Test.devide(TestException.java:14)
```

```
at TestException.main(TestException.java:5)
```

上面的程序运行的结果报告发生了算术异常 (ArithMeticException)，系统不再执行下去，提前结束，这种情况就是我们所说的异常。

#### 4.4.2 try... catch 语句

我们将上面的程序代码进行如下修改：

```
public class TestException
{
    public static void main(String [] args)
    {
        try
        {
            int reslut = new Test().devide( 3, 0 );
            System.out.println("the result is" + reslut );
        }
        catch(Exception e)
        {
            System.out.println(e.getMessage());
        }
        System.out.println("program is running here ,that is normal !");
    }
}

class Test
{
    public int devide(int x, int y)
    {
        int result = x/y;
        return x/y;
    }
}
```

程序运行结果如下：

```
/ by zero
program is running here ,that is normal !
```

我们看到程序在出现异常后，系统能够正常的继续运行，而没有异常终止。在上面的程序代码中，我们对可能会出现错误的代码用 try...catch 语句进行了处理，当 try 代码块中的语句发生了异常，程序就会跳转到 catch 代码块中执行，执行完 catch 代码块中的程序代码后，系统会继续执行 catch 代码块后的其他代码，但不会执行 try 代码块中发生异常语句后的代码，如程序中的 System.out.println("the result is" + reslut ); 不会再被执行。可见 Java 的异常处理是结构化的，不会因为一个异常影响整个程序的执行。

当 try 代码块中的程序发生了异常，系统将这个异常发生的代码行号，类别等信息封装到一个对象中，并将这个对象传递给 catch 代码块，所以我们看到 catch 代码块是下面的格式出现的。

```
catch(Exception e)
```

```

{
    System.out.println(e.getMessage());
}

catch (Exception e)
{
    System.out.println(ex.getMessage());
}

```

#### 4.4.3 throws 关键字

针对上面的例子，我们假设 TestException 类与 Test 类不是同一个人写的，写 TestException 类的人，在 main 方法中调用 Test 类的 devide 方法时，怎么能知道 devide 方法有可能出现异常情况呢？他又怎么能够想到要用 try...catch 语句去处理可能发生的异常呢？

问题可以这样解决，只要写 Test 类的人，在定义 devide 方法时，在 devide 方法参数列表后用 throws 关键字声明一下，该函数有可能发生异常及异常的类别。这样，调用者在调用该方法时，就必须用 try...catch 语句进行处理，否则，编译将无法通过。如下面的程序代码：

```

public class TestException
{
    public static void main(String [] args)
    {
        int result = new Test().devide( 3, 1 );
        System.out.println("the result is" + result );
    }
}

class Test
{
    public int devide(int x, int y) throws Exception
    {
        int result = x/y;
        return x/y;
    }
}

```

编译上面的程序，将出现如下的编译错误。

```

TestException.java:5: unreported exception java.lang.Exception; must be caught or
declared to be thrown
        int result = new Test().devide( 3, 1 );
                           ^
1 error

```

读者应注意一下出错的行号，就能够发现错误的位置。尽管我们已经将传给 devide 函数中的第二个参数改为了 1，程序在运行时不可能发生错误，但由于定义 devide 函数时声明了它有可能发生异常，调用者就必须使用 try...catch 语句进行处理，这叫防患于未然。

将程序作如下修改:

```
public class TestException
{
    public static void main(String [] args)
    {
        try
        {
            int result = new Test().devide( 3, 1 );
            System.out.println("the result is" + result );
        }
        catch(Exception e)
        {
            e.printStackTrace(); //很多人为了简单，不写这一句。
        }
    }
}
```

编译上面的程序，没有任何问题了。

**小经验：**很多人确信自己的程序中不会发生某种异常，try…catch语句似乎成了应对编译器而不得已的手段，为了简单，他们往往在catch代码块中什么也不写，就象上面标注的那样。try…catch成了一种摆设。作者强烈要求读者养成良好的编程习惯，不要回避这一点辛劳，贪图这么一点时间。在catch代码块中，最好有处理异常的代码，否则，一旦程序在运行过程中出现了异常，导致最终运行结果与我们期望的不一致，我们就很难发现问题的原因了。

### throws关键字的其他用处

如果一个方法中的语句执行时可能生成某种异常，但是并不能确定如何处理，则此方法应声明抛出异常，表明该方法将不对这些异常进行处理，而由该方法的调用者负责处理。也就是，程序中异常没有用try…catch捕捉异常、处理异常的代码，我们可以在程序代码所在的函数(方法)声明后用throws声明该函数要抛出异常，将该异常抛出到该函数的调用函数中，一直到main方法，JVM肯定要处理的，这样，编译就能通过了。用作者的话讲，就是将麻烦传递给了上级，村长传给乡长，乡长不处理就传给县长，县长不处理又传给市长，最后一直传到国务院，就截止了，因为国务院没有上级了。在Java程序中，异常传递到main方法后，就截止了，因为没有哪个方法会调用main方法的。

如上面的程序就可以改写成下面这样：

```
public class TestException
{
    public static void main(String [] args) throws Exception
    {
        int result = new Test().devide( 3, 1 );
        System.out.println("the result is" + result );

    }
}
```

虽然编译能够通过，但异常一旦发生，没有被处理，程序就会非正常终止。即使你能确信在你

的程序中不会发生某种异常，这种方法也能够简化程序的编写，但完全违背了 Java 设计异常的初衷，不建议在正规的程序中使用。用这种方法写点实验性的代码，倒是无可厚非。

#### 4.4.4 自定义异常与 Throw 关键字

我们通过查阅 JDK 文档资料，就能看到，Exception 类是 java.lang.Throwable 类的子类，Exception 类继承了 Throwable 类的所有方法。其实，我们在实际应用中，是使用 Exception 的子类来描述任何特定的异常的。Exception 类是所有异常类的父类，Java 语言为我们提供了许多 Exception 类子类，分别对应不同的异常类型，如我们在前面已经看到过以下几个异常。

- | ArithmeticException (在算术运算中发生的异常，如除以零)
- | NullPointerException (变量还没有指向一个对象，就引用这个对象的成员)
- | ArrayIndexOutOfBoundsException (访问数组对象中不存在的元素)

除了系统提供的异常，我们也可以定义自己的异常类，自定义的异常类必须继承 Exception 类。假设我们在上面程序的 devide 函数中，不允许有负的被除数，当 devide 函数接收到一个负的被除数时，程序返回一个自定义的异常（这里就叫负数异常吧！）通知调用者。我们可以这样定义这个负数异常类。

```
class DevideByMinusException extends Exception
{
    int devisor;
    public DevideByMinusException(String msg,int devisor)
    {
        super(msg);
        this.devvisor = devisor;
    }
    public int getDevisor()
    {
        return devisor;
    }
}
```

Java 是通过 throw 关键字抛出异常对象的，其语法格式是  
throw 异常对象；

如果我们的程序想跳转，我们可以抓住自己抛出的异常。否则，则此方法应声明抛出异常，而由该方法的调用者负责处理。

如果我们要在 devide 函数接收到的第二个参数（也就是被除数）为负数时，向调用者抛出自定义的 DevideByMinusException 对象，程序代码如下：

```
class Test
{
    public int devide(int x, int y)
        throws ArithmeticException, DevideByMinusException
    {
        if(y < 0)
            throw new DevideByMinusException("被除数为负",y);
        //这里抛出的异常对象，就是调用者在 catch (Exception e) {}语句中接收的变量 e。
        int result = x/y;
        return x/y;
    }
}
```

```
    }  
}
```

我们能够看到，上面的代码中，`devide` 方法声明时抛出了两个异常。java 中一个方法是可以被声明成抛出多个异常的。

下面再来看看调用程序应该如何对 `devide` 方法中的多个异常作出处理。

```
public class TestException  
{  
    public static void main(String [] args)  
    {  
        try  
        {  
            int result = new Test().devide( 3, 0 );  
            //int result = new Test().devide( 3, -1 );  
            //int result = new Test().devide( 3, 1 );  
            System.out.println("the result is " + result );  
        }  
        catch(DevideByMinusException e)  
        {  
            System.out.println("program is running into"+  
                "DevideByMinusException");  
            System.out.println(e.getMessage());  
            System.out.println("the devisor is " + e.getDevisor());  
        }  
        catch(ArithmeticException e)  
        {  
            System.out.println("program is running into"+  
                "DevideByMinusException");  
            System.out.println(e.getMessage());  
        }  
        catch(Exception e)  
        {  
            System.out.println("program is running into"+  
                "other unknowned Exception");  
            System.out.println(e.getMessage());  
        }  
        System.out.println("program is running here ,that is normal !");  
    }  
}
```

上面的程序使用一个 `try` 后面跟着多个 `catch` 来捕捉异常，每一个 `catch` 可以处理一个不同的异常类型。如果我们调用 `devide(3, 0)`，将发生 `ArithmeticException` 异常，程序将跳转至 `catch(ArithMeticException e)` 代码块中执行。如果我们调用 `devide(3, -1)`，将发生 `DevideByMi nusException` 异常，程序将跳转至 `catch(DevideByMi nusException e)` 代码块中执行。如果在 `devide` 方法中发生了 `Arithmeti cException`、`DevideByMi nusException` 之外的任何其他

异常，程序将跳转至 catch(Exception e) 代码块中执行，也就是还没有由前面的 catch 处理的异常，都将由 catch(Exception e) 代码块处理，不是因为该语句是最后一个 catch，而是由于 Exception 是所有异常类的父类，从语法上来讲，是能够处理所有异常的。所以，各种 catch 代码块的放置顺序非常重要，catch(Exception e) 不能放在其他 catch 语句的前面，否则后面的 catch 永远得不到执行，就没有什么意义了。大家可以用 if, else if, else if, else 结构去考虑 try...catch 问题。读者可以逐一去掉程序中被注释的 devide 语句的注释符，注释掉其他的 devide 语句，对比观察运行结果。

## & 多学两招：

我们可以在一个方法中使用 throw, try...catch 语句来实现程序的跳转，下面是描述这种方法的一段简要程序代码：

```
void fun()
{
    try
    {
        if(x==0)
            throw new YyyException("Yyy"); //跳转到代码块 1 处
        else(x==1)
            throw new XxxException("Xxx"); //跳转到代码块 2 处
    }
    catch(YyyException e)
    {
        代码块 1
    }
    catch(XxxException e)
    {
        代码块 2
    }
}
```

### 4.4.5 finally 关键字

在 try...catch 语句后，我们还可以有个 finally 语句，finally 语句中的代码块不管异常是否被捕获总是要被执行的。我们将上面的程序作如下修改，来看看 finally 语句的用法与作用。

```
public class TestException
{
    public static void main(String [] args)
    {
        try
        {
            int reslut = new Test().devide( 3, 0 );
            //int reslut = new Test().devide( 3, -1 );
            //int reslut = new Test().devide( 3, 1 );
            System.out.println("the result is" + result );
            //return ;
        }
    }
}
```

```

        }
        catch(DevideByMinusException e)
        {
            System.out.println("program is running into"+
                "DevideByMinusException");
            System.out.println(e.getMessage());
            System.out.println("the devisor is " + e.getDevisor());
            System.exit(0);
        }
        catch(ArithMeticException e)
        {
            System.out.println("program is running into"+
                "DevideByMinusException");
            System.out.println(e.getMessage());
            return ;
        }
        catch(Exception e)
        {
            System.out.println("program is running into"+
                "other unknowned Exception");
            System.out.println(e.getMessage());
        }
        finally
        {
            System.out.println("program is running into finally");
        }
        System.out.println("program is running here ,that is normal !");
    }
}

```

爱思考的读者也许会问：程序中的最后一句 `System.out.println("program is running here ,that is normal !");` 不论有没有异常发生，该语句都会执行，那何必还用 `finally` 语句呢？

`finally` 还是有其特殊之处的，即使 `try` 代码块和 `catch` 代码块中使用了 `return` 语句退出当前方法或 `break` 跳出某个循环，相关的 `finally` 代码块都要执行，读者可以逐一去掉程序中被注释的 `devide` 语句的注释符，注释掉其他的 `devide` 语句，对比观察运行结果。

`finally` 中的代码块不能被执行的唯一情况是：在被保护代码块中执行了 `System.exit(0)`。

每个 `try` 语句必须有一个或多个 `catch` 语句对应，`try` 代码块与 `catch` 代码块及 `finally` 代码块之间不能有其他语句。如下面的代码是非法的。

```

try
{
    .....
}

x = 3 + 5;//代码位置是非法的
catch(Exception e)

```

```
{  
    ....  
}
```

#### 4.4.6 异常的一些使用细节

- 1). 一个方法被覆盖时，覆盖它的方法必须扔出相同的异常或异常的子类。
- 2). 如果父类扔出多个异常，那么重写（覆盖）方法必须扔出那些异常的一个子集，也就是说，不能扔出新的异常。

#### 4.4.7 Java 引入异常的好处

以前，程序语言要求程序员使用函数调用的返回值来判断执行情况，这种方式有很多缺陷，因为对许多可能出现的异常情况需要不同程度的了解。有些语言使用一个全局变量来保存异常状态，但这种方式是不够的，因为后面的异常会在前一个异常还未处理之前就覆盖了那个全局变量的值。而且，更不可取的是，有些 C 程序借助 goto 语句来处理异常。Java 没有 goto 语句，它保留 goto 关键字只是为了让程序员不要搞混淆。Java 利用带标号的 break 和 continue 语句来取代 goto。Java 中严格定义的异常处理机制使 goto 没有再存在的必要，取消这种随意跳转的语句有利于优化代码以及保持系统的强健性和安全性。

Java 的异常是一个出现在代码中的描述异常状态的对象。每当出现一个异常情况，就创建一个异常对象并转入引起异常的方法中，这些方法根据不同的类型来捕捉异常，并防止由于异常而引起程序崩溃，还能在方法退出前执行特定的代码段。

作者感受最深的是：Java 异常强制我们去考虑程序的强健性和安全性，很多人在写程序时，总是想当然地认为，自己的程序会在一种理想的环境下运行，譬如，不会碰到硬盘空间不够的情况，不会碰到网络突然断线的情况，不会碰到用户输入非法数据的情况，不会……等等，通常都懒得去写处理这些意外情况的代码。一旦有个万一，碰到上面的意外情况，程序就只有崩溃的份儿了，就象许多 Windows 用户经常见到的情况，一个大红框冒出来后，程序就结束了，刚才做的工作还没保存呢，这些用户好不懊恼，气得直骂那个该死的、做这个软件的人！Java 对可能发生这些的意外情况的方法都用 throws 关键字进行了标记，程序员就不得不去编写处理意外的代码，想偷点懒都不成。使用 Java 语言编程，能使我们养成良好的编程风格！

### 4.5 包

读者可以想一想，sun 公司的 JDK，系统软件商，开发工具商都会提供成千上万个具有各种用途的类，我们也要管理自己的大型软件系统中数目众多的类。如果不对这些类进行分门别类的使用和存放，就象我们不用文件夹去管理众多的文件一样，在使用时将极度困难和不方便，也极易出现类的命名冲突问题。Java 是通过引入包(package)机制，提供类的多层次命名空间，来解决上述问题的。

#### 4.5.1 package 语句及应用

我们还是通过一段程序来开始 package 的讲解：

```
package org.it315;  
public class TestPackage
```

```

{
    public static void main(String [] args)
    {
        new Test().print();
    }
}
class Test
{
    public void print()
    {
        System.out.println("the program is demonstrating how to using package!");
    }
}

```

上面的程序第一条语句，指示这个源文件中的所有类都位于包 org.it315 中，位于包中的每个类的完整名称都应该是包名与类名的组合，如上面的类 TestPackage 的完整名称应是 org.it315.TestPackage。

同一个包中的类相互访问，不用指定包名，如程序中的 TestPackage 类访问 Test 类时，用的是 Test，而不是 org.it315.Test。如果从外部访问一个包中的类，必须使用类的完整名称。就好比北京人说：“我要去一趟上海！”，如果说成“我要去一趟中国.上海！”，同是中国的两个城市，使用时非要加个中国前缀，也是行得通的，但让人听来就有点像卖国贼了。美国纽约人则必须说：“我要去一趟中国.上海！”，因为这是两个不同国家的城市。

如我们要在命令行中启动 java 虚拟机解释运行 TestPackage 类，我们必须用：

**java org.it315.TestPackage**

而不能用

**java TestPackage**

在命令行窗口中，进入 TestPackage.java 源文件所在目录，

1. 执行 javac TestPackage.java 命令，编译 TestPackage.java 源文件，生成了 TestPackage.class 字节码文件。
2. 在当前目录下，执行 java TestPackage 命令，运行结果如下：

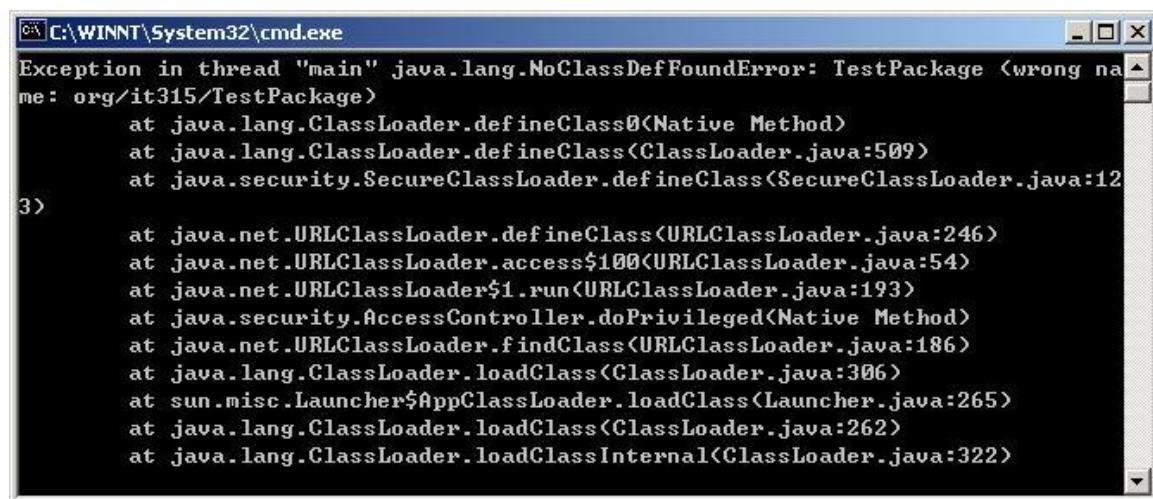


图 4.2

确信我们已经解决了 classpath 问题，这是怎么回事呢？

3. 哦！我们刚才讲过的，在包的外部访问包内的类时，应该使用类的完整名称，我们改为执行如下命令：

```
java org.it315.TestPackage
```

运行结果如下：

```
Exception in thread "main" java.lang.NoClassDefFoundError:  
org/it315/TestPackage
```

这又是怎么回事呢？不要怀疑我前面讲的知识，我肯定地告诉你，前面讲的没有任何错，要调用的类只能以 `org.it315.TestPackage` 完整名称的形式出现。

问题是这样的：位于在包中的类，在文件系统中（通俗地说，就是硬盘上）的存放位置，必须有与包名层次相对应的目录结构。在 `package` 语句中，用点（.）来指明包的层次，如：

```
package org.it315;
```

这个语句表示有一个 `org` 的包，该包中包含有一个名为 `it315` 的子包。如上面的程序，类 `TestPackage` 就在 `it315` 子包中，我们假设 `TestPackage.java` 源文件所在目录的名称为 `c:\myjava`，我们需要在 `c:\myjava` 目录中创建一个 `org` 的子目录，在 `org` 子目录中又要创建一个 `it315` 子目录，`TestPackage.class` 文件就要位于 `it315` 目录中，如图 4.3 所示：



图 4.3

当虚拟机要装载 `org.it315.TestPackage` 类时，它会沿着 `classpath` 环境变量指定的路径中去逐一查找，查找这些路径下是否有 `org` 子目录，接着在 `org` 目录中又去查找 `it315` 子目录，最后在 `it315` 目录中去查找 `TestPackage.class` 文件名，看其中是否包含 `org.it315.TestPackage` 类。虚拟机在装载带有包名的类时，会先找到 `classpath` 环境变量指定的目录，再在这些目录中，按照与包名层次相对应的目录结构去查找 `.class` 文件。

`classpath` 设置一定是指向顶层包名所对应目录的父目录 (`c:\myjava`)，而不是指向 `.class` 文件所在的目录 (`c:\myjava\org\it315`)。

**M** 脚下留心：

1).即使文件名都是 TestPackage.class，但其中包含的类的完整名称却不见得相同，如 org. i t315. TestPackage 类与 com. sun. TestPackage 类的文件名都 TestPackage.class，但却是两个完全不同的类，应存放在不同的目录结构中。

2).同一个包中的类不必位于同样的目录，如 org. i t315. Class1 与 org. i t315. Class2，完全可以一个位于 c 盘某个位置，另一个位于 d 盘某个位置，只要 classpath 分别指向这两个位置就可以了。有了包的概念，我们就可以理解 Java 实际上是通过 classpath 去寻找顶层包名的原理了。

4. 在当前目录下，创建 org\i t315 子目录，将 TestPackage.class 文件移动到 org\i t315 目录中，在当前目录下重复刚才的命令（在 win2000 下可以用上下键调出当前命令前后运行过的命令）。

```
java org.it315.TestPackage
```

结果就如我们所愿了。

5. 其实，我们也可以在编译时，让 javac 来生成与包名层次相对应的目录结构，而不必手工去创建。直接运行 javac 命令，可以看到 javac 的所有参数和作用。javac 命令有一个选项“-d”，用于指定编译生成的.class 文件存放的目录。读者已经从前面的内容中发现，当没有使用-d 选项时，.class 文件将存放在当前工作目录。我们将刚才创建的 org 目录彻底删除，运行下面命令：

```
javac -d . TestPackage.java
```

因为点(.)代表当前目录，也就是我们要将编译生成结果存放的当前工作目录下。这时，javac 替我们创建了 org\i t315 目录，并将.class 文件放在了该目录中，也就是 (-d) 选项还能生成与包名层次相对应的目录结构。读者已经看到了事实，就不要再问 javac -d . TestPackage.java 与 javac TestPackage.java 看起来不都是一回事吗，都是将编译生成的.class 文件放在当前目录，javac TestPackage.java 为什么不生成 org\i t315 目录？是的，作者就认为 javac TestPackage.java 应该为我们生成 org\i t315 目录，这对我们 java 程序员才叫方便，sun 公司应该做成这样才好！人无完人，当然也不存在没有问题的软件，这又该是 Sun 公司的一个疏漏吧！未来的程序员们，我们以后在自己编程中，一定要多为用户着想，尽量让用户使用起来方便，如果你的用户要用一些小窍门或一些繁琐的步骤，才能用得好你提供的软件，那么你的软件是不会受到欢迎的，你的技术支持也会累得喘不过气来的。

6. 再来做几个小实验，以便加深读者对包的更进一步的认识，这也是作者在长期的教学中发现许多学员总是容易混淆的地方。以下的实验都确信 classpath 环境变量中包含了当前目录。

1)在命令行中进入 org\i t315 目录中，运行 java TestPackage，结果会怎样？

提示：类的名称是 org. i t315. TestPackage，而不是 TestPackage，即使能够找到 TestPackage.class 文件，也不可能正常运行。

2)还是在 org\i t315 目录中，运行 java org. i t315. TestPackage 结果又怎样？

提示：我们要运行的类的完整名称为 org. i t315. TestPackage，所以 java 虚拟机会在 classpath 指定的路径下(包含当前目录 c:\myj ava\org\i t315)再去查找子目录 org\i t315 中的 TestPackage.class 文件，对于当前目录来说，java 虚拟机要找 c:\myj ava\org\i t315\org\i t315\TestPackage.class 文件，当然找不到。

3)将 TestPackage.java 源文件中的 package org. i t315; 语句注释掉，重新编译，将生成的.class 文件放在 c:\javawork\org\i t315 目录下，对，就是 c:\javawork，不是 c:\myj ava。设置 classpath 环境变量包含 c:\javawork 目录，然后运行 java org. i t315. TestPackage 命令。结果又会怎样？

**提示：**很多初学者常常误以为把一个类放在某个目录下，就等于这个目录名就成了这个类的包名。再次强调，不是有了目录结构，就等于有了包名，包名必须在程序中通过 package 语句指定，而不是靠目录结构来指定的，是先要有了包名后，才需要相应的目录结构。你刚才的类名是 TestPackage，所以你当然找不到 org.it315.TestPackage。但有的读者可能运行结果显示一切正常，要真这样就好了，你就能对我下面的分析容易理解多了和能够产生刻骨铭心的记忆了。这是因为：你运行上面命令时的当前目录还在 c:\myjava 下，且 classpath 环境属性包含当前目录（注意，即使没有显示包含 .，它也有可能的），你运行的就是你先前的那个正确的 org.it315.TestPackage 类，而不是你刚才重新编译的这个类。删除 c:\myjava\org 目录，重新运行，你就能够看到这个错误现象了。

## M 脚下留心：

下面这个问题也是在实际应用中经常发生的事情，虽然编译了一个修改过的.java 源文件，但运行可能是某个旧的.class 文件，特别是旧的.class 文件所在的目录在 classpath 环境变量中的位置，位于新的.class 文件所在的目录的前面，问题就更加隐蔽了。当我们在编程过程中遇到了问题，有时并不是程序本身所带来的问题，需要我们放眼全局，思路更加开阔一些，从多个方面去思考和解决问题。

### 有关包的其他细节：

package 语句作为 Java 源文件的第一条语句，指明该文件中定义的类所在的包，必须把包声明放在源文件的最前面，每个源文件只能声明一个包。

如果没有 package 语句，则为缺省无名包。但实际项目应用中，没有使用无名包的类，读者就把这当着是一个硬性规定吧，相信我，这不仅是一个良好的编程习惯，而且有助你在日后实际工作中避免许多可能碰到的麻烦。

### 4.5.2 import 语句及应用

了解完 package 语句的作用，我们来看看不同包的类之间是如何调用的。我们将 TestPackage.java 中的两个类放到不同的包中，按我们前面所讲，我们需要两个单独的.java 源文件来容纳这两个类，程序如下：

程序清单 1：TestPackage.java

```
package org.it315;
public class TestPackage
{
    public static void main(String [] args)
    {
        new Test().print(); //标记 1
    }
}
```

程序清单 2：Test.java

```
package org.it315.example;
public class Test
{
    public void print()
    {
```

```

        System.out.println("the program is demonstrating how to using package!");
    }
}

```

假设这两个源文件在同一目录，运行下面的命令（在 javac 命令中，是可以使用通配符来指定一次编译多个源文件的）：

```
javac -d . Test*.java
```

编译能够正常通过吗？如果通过了，肯定又是你没有清除先前实验所留下的 org\it315\Test.class 文件，先删除这个文件，重新编译，这下应该出现编译错误了，运行结果如下：

```

C:\WINNT\System32\cmd.exe
C:\myjava>javac -d . Test*.java
TestPackage.java:6: cannot resolve symbol
symbol : class Test
location: class org.it315.TestPackage
        new Test().print();//标记1
                           ^
1 error

C:\myjava>

```

图 4.4

上面的错误提示告诉我们，程序错误的位置是 TestPackage.java 源文件中的第 6 行，注意上面出错的行号，如果你用的是 EditPlus 之类的工具软件，我们可以使用工具栏上的“转到某行”按钮，或者快捷键 Ctrl+G 去定位程序的错误位置，如图 4.5。这类功能在我们以后的编程中，尤其是在查找较大程序的编译错误时特别好用的。

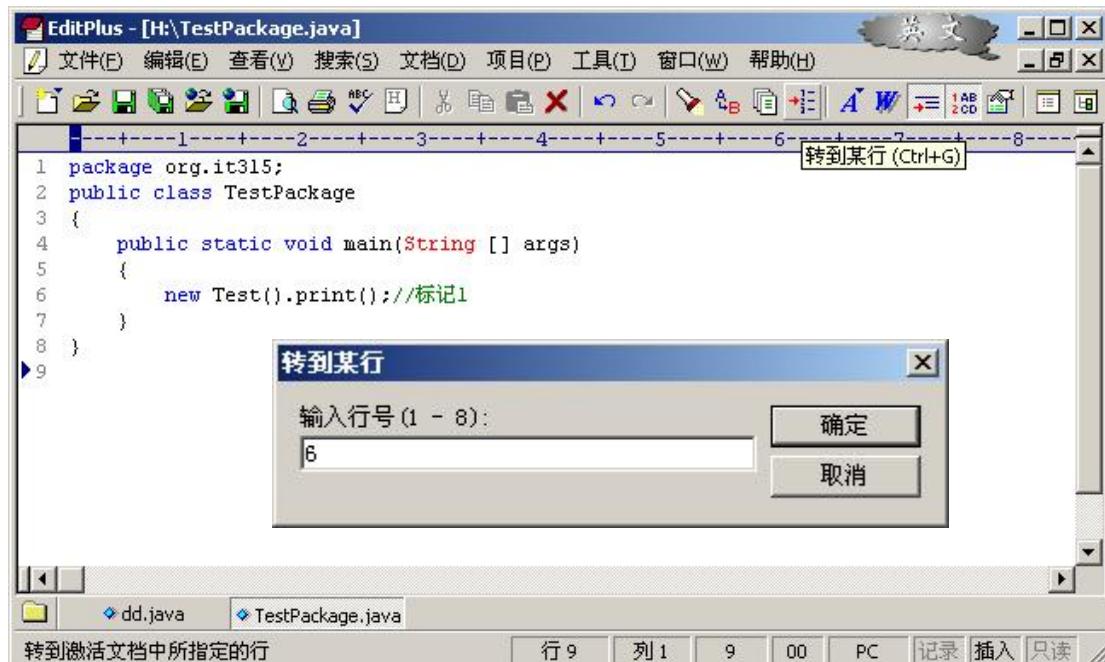


图 4.5

言归正传，回来看一下我们上面的错误，出错的原因在于：在类 TestPackage 中直接调用类 Test，编译器认为这两个类应在同一包中，即类 Test 完整名称为 org.it315.Test，实际上，我们是找不到这个类的。我们将标记 1 处的语句修改成：

```
new org.it315.example.Test().print();
```

重新编译，就一切正常了。

如果我们要在程序中多次用到 org.it315.example 包中的若干类名，我们每次用到其中的类名时，都要加上 org.it315.example 前缀，显然有点儿繁琐。在 java 中是使用 import 语句来简化这个问题的。

我们修改 TestPackage.java 源文件的程序代码，如下：

```
package org.it315;
import org.it315.example.*;
//import org.it315.example.Test;
public class TestPackage
{
    public static void main(String [] args)
    {
        new Test().print(); //标记 1
    }
}
```

重新编译，一切顺利，这个例子已经向读者显示了 import 语句的作用。有了 import 语句，我们在使用某个包里的类时，就不用写上一长串包名了（如 org.it315.example.Test）。我们也可以去掉第二行的注释，然后注释掉第一行，编译运行一下，发现与前面的结果是完全一样的。那么上面粗体的两行 import 语句有什么不一样呢？第一个 import 语句，导入了整个包中的所有类，那么我们的程序中凡是遇到的 org.it315.example 包中的类，都不用再写包名了，只需写一个类名就可以了。第二个 import 语句只导入了该包中的 Test 类，对该包中的其他类不起作用，在程序中用到 Test 类时，我们不用写包名了，但对于该包中的其他类，我们还必须写上完整的包名。

如果两个包中都含有同样的类名，如 java.util、java.sql 这两个包中都有 Date 这个类，程序又同时导入了这两个包中的所有类，如：

```
import java.util.*;
import java.sql.*;
```

这时候程序就不知道该用哪个包里面的 Date 类，编译器在碰到使用 Date 类的地方就会报错。对于这种情况，我们就需要在用到这个特殊的类的地方写上完整的包名，如  
new java.sql.Date()。

## & 多学两招：

父包和子包之间，能从语意上表示某种血缘和亲近关系，如 org.it315.\*，org.it315.mail.\*，从包名上可以看出这些类都是 it315 这个组织（或公司）开发的类，org.it315.mail.\*还能进一步地说明其中的类是专用于 mail 项目的。但父包和子包在使用上没有任何关系，如父包中的类调用子包中的类，必须引用子包的全名，而不能省略父包名部分。另外，当我们 import 了一个包中所有的类，并不会 import 这个包中的子包中的类，如果程序中用到了子包的类，需要再次对子包作单独引入，如我们以后经常会见到在同一程序中，经常同时出现下面的两条语句。

```
import java.awt.*;
import java.awt.event.*;
```

## \$ 小经验：

我们在编程中总是会碰到这样的错误：



图 4.6

不管是从学习班上，还是在各大网站的论坛上，这都是一个初学者常问的问题，从上面打出来的信息，我们知道错误的现象是找不到 Temp 类，相信很多读者都遇到过这样的错误。我们现在正好讲完了怎样导入包了，也讲完 classpath 了，现在我们总结一下吧，找不到类的原因大致有三种情况：

### 1) 把文件名字写错了

这种情况碰到的次数最多，也比较好排除，只要了解了 Java 的一些简单的书写规则，再细心一些，相信不会难倒读者的。如 Java 是大小写敏感的，但初学者很容易犯把 Test 写成 test 了之类的错误。

### 2) 没有 import 该类所在的包名

如果你能确认不是拼写的错误，那你可能忘记用 import 语句来引入该类所在包名了。

### 3) classpath 设置错误

如果上面两种错误的可能性，你都排除了，那就可能是你的计算机上压根儿就没有这个类文件，或者你的计算机上即使有这个类文件，你的 classpath 环境变量可能没有正确指向该类所在的 jar 包（参看本章最后的部分）或文件目录。我们在第一章中详细介绍了 classpath 的用法，不熟悉的读者请再返回去看一下。

## 4.5.3 JDK 中的常用包

Sun 公司在 JDK 中为我们提供了大量的各种实用类，通常称之为 API (application programming interface)，这些类按功能不同分别被放入了不同的包中，供我们编程使用，下面简要介绍其中最常用的六个包：

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。
2. `java.awt`----包含了构成抽象窗口工具集 (abstract window tool kits) 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. `java.applet`----包含 applet 运行所需的一些类。
4. `java.net`----包含执行与网络相关的操作的类。
5. `java.io`----包含能提供多种输入/输出功能的类。
6. `java.util`----包含一些实用工具类，如定义系统特性、使用与日期日历相关的函数。

注：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，如我们前面经常使用的 `System` 类。

## 4.6 访问控制

有了包的知识，我们就可以全面讲解访问控制修饰符了。访问控制修饰符共有 4 个，分别

是 public、protected、default、private。下面我们分别进行讲解。

### 4.6.1 类成员的访问控制

#### private 访问控制

在本章的前面，我们已经明白了 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。

#### 缺省访问控制

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，我们就称这个成员是缺省的 (default)，或是友元的(friendly)，或是包类型的(package)。对于缺省访问控制成员，可以被这个包中的其他类访问，如果一个子类与父类位于不同的包中，子类也不能访问父类中的缺省访问控制成员。

#### protected 访问控制

如果一个成员方法或成员变量名前使用了 protected 访问控制符，那么这个成员即可以被同一个包中的其他类访问，也可以被不同包中的子类访问。

#### public 访问控制

如果一个成员方法或成员变量名前使用了 public 访问控制符，那么这个成员即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，我们用一张图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 |         |         |           | √      |

表 4.1

### 4.6.2 类的访问控制

除了类中的成员有访问控制外，类本身也有访问控制，即在定义类的 class 关键字前加上访问控制符，但类本身只有两种访问控制，即 public 和默认，父类不能是 private 和 protected，否则子类无法继承。public 修饰的类能被所有的类访问，默认修饰（即 class 关键字前没有访问控制符）的类，只能被同一包中的所有类访问。

#### & 多学两招：

只要在 class 之前，没有使用 public 修饰符，源文件的名称可以是一切合法的名称。带有 public 修饰符的类的类名必须与源文件名相同，读者可以想一想，一个.java 源文件中能否包含多个 public 的类呢？

### 4.6.3 Java 的命名习惯

养成良好的命名习惯，意义重大，如果大家的习惯都一样，我们就能够很容易使用别人提供的类，别人也很容易理解我们的类，对此，我送给读者一句话，“勿以善小而不为，勿以恶小而为之”！下面是 Java 中的一些命名习惯，假设 xxx, yyy, zzz 分别是一个英文单词的拼写。

- u 包名中的字母一律小写, 如: xxxyyyyzzz。
- u 类名、接口名应当使用名词, 每个单词的首字母大写, 如: XxxYyyZzz。
- u 方法名, 第一个单词小写, 后面每个单词的首字母大写, 如: xxxYyyZzz。
- u 变量名, 第一个单词小写, 后面每个单词的首字母大写, 如: xxxYyyZzz。
- u 常量名中的每个字母一律大写, 如: XXXYYYYZZZ。

## 4.7 使用 jar 文件

我们用的 jdk 中的包与类主要在 jdk 的安装目录的 `jre\lib\rt.jar` 文件中, 由于 Java 虚拟机会自动找到这个 jar 包, 所以我们在使用这个 jar 包的类时, 无需再用 classpath 来指向它们的位置了。

### 4.7.1 jar 文件包

jar 文件就是 Java Archive File, 顾名思义, 它的应用是与 Java 息息相关的。jar 文件就是一种压缩文件, 与我们常见的 ZIP 压缩文件格式兼容, 习惯上称之为 jar 包。我们开发了许多类, 当需要把这些类提供给别人使用时, 通常都会将这些类压缩到一个 jar 文件中, 以 jar 包的方式提供给别人使用。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。我们用图 4.7 来形象地了解 jar 文件的内部结构。

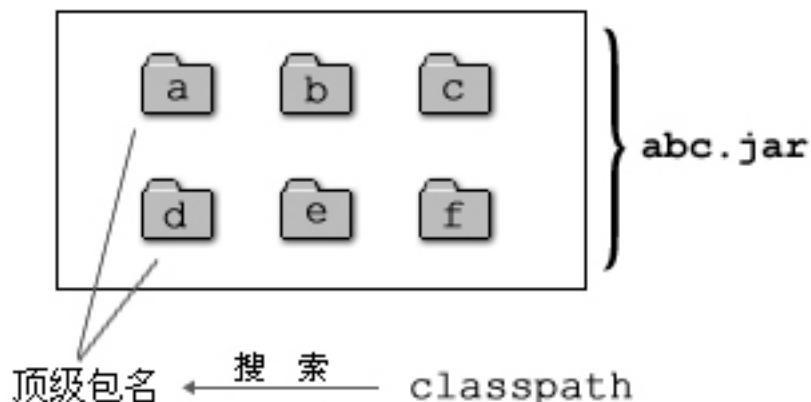


图 4.7

了解了 jar 文件之间的内部结构后, 我们就可以通过设置 classpath 环境变量来指向一个具体的 jar 文件 (配置 classpath 环境变量请读者参照第一章) 如图 4.8 所示。

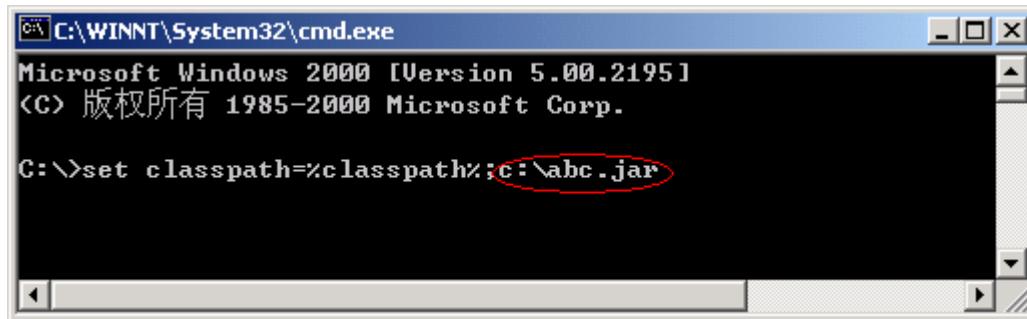


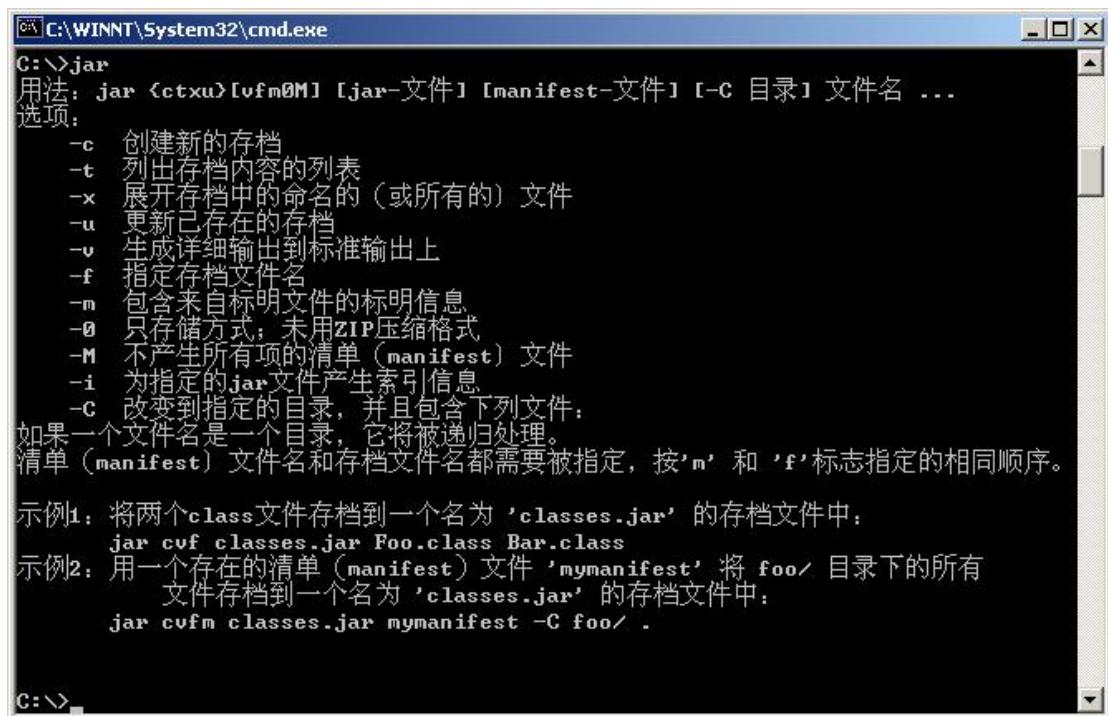
图 4.8

注意：在设置 classpath 时，不能使用相对路径。

#### 4.7.2 jar 命令详解

jar 命令是随 JDK 自动安装的，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。通过 jar 命令所生成的.jar 压缩文件有什么优点呢？一方面，可以方便我们管理大量的类文件，另一方面，进行了压缩也减少了文件所占的空间。对于我们来说，除了安装 JDK 之外什么也不需要做，因为 Sun 公司已经帮我们做好了。

了解了 jar 最基本的含义，接着我们在命令行窗口下运行 jar.exe 程序，就可以看到 jar 命令的用法如图 4.9 所示：



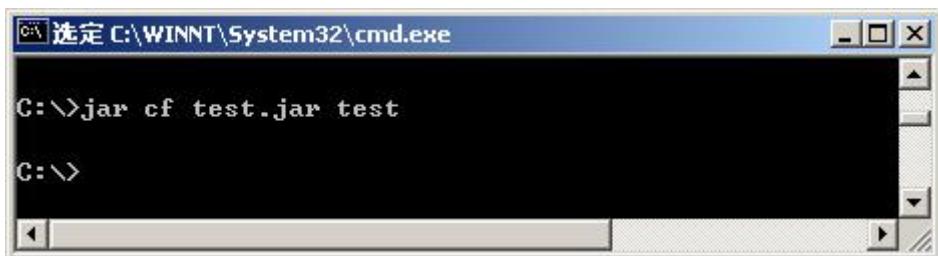
C:\>jar  
用法: jar <ctxu>[vfm0M] [jar-文件] [manifest-文件] [-C 目录] 文件名 ...  
选项:  
-c 创建新的存档  
-t 列出存档内容的列表  
-x 展开存档中的命名的(或所有的)文件  
-u 更新已存在的存档  
-v 生成详细输出到标准输出上  
-f 指定存档文件名  
-m 包含来自自明文的标明信息  
-0 只存储方式: 未用ZIP压缩格式  
-M 不产生所有项的清单(manifest)文件  
-i 为指定的jar文件产生索引信息  
-C 改变到指定的目录，并且包含下列文件:  
如果一个文件名是一个目录，它将被递归处理。  
清单(manifest)文件名和存档文件名都需要被指定，按'm' 和 'f' 标志指定的相同顺序。  
示例1: 将两个class文件存档到一个名为'classes.jar'的存档文件中:  
jar cvf classes.jar Foo.class Bar.class  
示例2: 用一个存在的清单(manifest)文件'mymanifest'将foo/目录下的所有  
文件存档到一个名为'classes.jar'的存档文件中:  
jar cvfm classes.jar mymanifest -C foo/ .

图 4.9

现在我们来举一些例子说明 jar 命令的基本用法，仍然在先前执行的命令行窗口环境中。读者可以任选一个目录，作者所用的是 c 盘根目录下的一个 test 子目录。

##### 1) jar cf test.jar test

该命令没有执行过程的显示，执行结果是在当前目录生成了 test.jar 文件。如果当前目录已经存在 test.jar，那么该文件将被覆盖。



C:\>jar cf test.jar test

图 4.10

2) jar cvf test.jar test

该命令与上例中的结果相同，但是由于 v 参数的作用，显示出了打包过程，如图 4.11 所示：



```
C:\>jar cvf test.jar test
标明显清单(manifest)
增加: test/<读入= 0> <写出= 0><存储了 0>
C:\>
```

图 4.11

6) jar tvf test.jar

该命令除显示图 4.11 中显示的内容外，还包括包内文件的详细信息，如图 4.12 所示：



```
C:\>jar tvf test.jar
 0 Mon Mar 31 10:37:12 CST 2003 META-INF/
 68 Mon Mar 31 10:37:12 CST 2003 META-INF/MANIFEST.MF
 0 Mon Mar 31 10:29:40 CST 2003 test/
C:\>
```

图 4.12

7) jar xf test.jar

解开 test.jar 到当前目录，不显示任何信息。



```
C:\>jar xf test.jar

C:\>
```

图 4.13

8) jar xvf test.jar

运行结果与 7) 相同，只是对于解压过程有详细信息显示，如图 4.14 所示：



图 4.14

9) 我们前面已经讲过了利用 tvf 参数的作用可以查看 jar 文件的内容。要是 jar 文件很大，包含的类很多，这时查看，可能会因为显示出的信息过多，屏幕刷新的速度过快，而不能正常浏览，如图 4.15 所示。这里就以 rt.jar 文件为例来进行讲解。rt.jar 文件是随着 JDK 安装就自动存在，包含了 Java 提供的所有基础类，它处在 JDK 安装目录里的\jre\lib 目录下。我们平时通过查看 rt.jar 文件就可以了解到 Java 里面种类繁多的类文件。

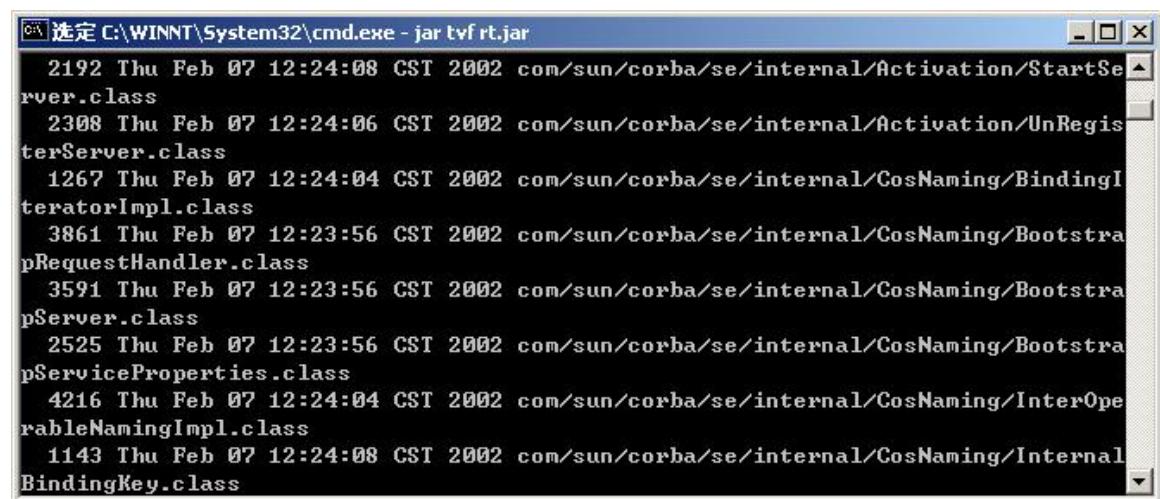


图 4.15

这是一个漫长的显示过程，并且最终只有少量的内容能够保留在屏幕上，我们难以查看想了解的内容。我们怎样才能避免这样的情况出现呢？我们不妨使用 dos 命令的重定向功能，将显示到屏幕上的内容定向到一个文件中。我们只需要执行 jar tvf rt.jar > c:\rt.txt 即可在 c 盘根目录下生成一个 rt.txt 的文本文件，如图 4.16、4.17 所示。我们以后如果想要查看 rt.jar 的内容，只需要查看 C 盘根目录下的 rt.txt 的内容就可以了。在这里符号“>”是重定向符，它的作用就是将显示输出到屏幕中的内容重定向到文件 rt.txt 中。



图 4.16

```
选定 C:\WINNT\System32\cmd.exe
C:\>dir
驱动器 C 中的卷是 操作系统
卷的序列号是 B8AC-6F61

C:\ 的目录

2003-03-18  16:52      186 boot.ini
2003-03-16  15:36      <DIR> Documents and Setting
2003-03-29  15:34      <DIR> engineer
2003-03-29  15:31      608 engineer.java
2003-03-16  15:29      <DIR> Inetpub
2003-03-29  10:29      <DIR> j2sdk1.4.0
2003-03-31  11:00      688,695 jar.txt
2003-03-31  10:44      <DIR> META-INF
2003-03-25  10:29      <DIR> My Music
2003-03-29  10:30      <DIR> Program Files
2003-03-31  11:05      688,695 rt.txt
2003-03-18  17:28      <DIR> sql2ksp3
2003-03-31  10:29      <DIR> test
2003-03-31  10:37      428 test.jar
2003-03-29  17:02      <DIR> WINNT
      5 个文件      1,378,612 字节
      10 个目录    6,014,111,744 可用字节

C:\>
```

图 4.17

#### 4.7.3 使用 WinRAR 对 jar 文件进行查看

我们可以使用 WinRAR 或者 WinZIP（这里我们以 WinRAR 来进行讲解，选择 WinRAR 是因为它能够看到 jar 文件内部的详细结构及其目录层次，而使用 WinZIP 却只能够看到一大堆的.class 文件，没有清晰的目录层次）来解压 jar 文件。

在介绍 jar 文件的时候就已经说过了，jar 文件实际上就是一种压缩文件，可以使用常见的一些解压工具来解压 jar 文件，如 Windows 下的 WinRAR、WinZIP 等。讲到这里可能会有细心的读者提问了：“上面讲过的 jar.exe 程序本身不就带有解压的功能吗？为什么我们还要借助于另外的工具呢？”这是因为如果使用 WinRAR 来进行 jar 文件的解压不仅方便快速，而且还便于我们轻松的浏览 jar 文件中的任意目录，并且目录层次清晰，读者可以参看下面的图 18、图 19。而在命令行窗口下运行 jar.exe 程序来进行 jar 文件的解压，解压效率和生成文件的层次感就差多了。

### F 指点迷津：

#### 1) 注意压缩后的 jar 文件中的目录结构

我们来举一个压包的过程，假设我们需要压缩的对象是 test1.Test1 类和 test2.Test2 类，需要将他们压缩成 myjava.jar 文件。我们首先要将这两个类的顶级包名所对应的目录（test1

和 test2) 放在同一个目录下, 作者所用的是 c:\myj ava 目录, 接着我们运行 c:\> jar cvf test.jar myj ava/\* 进行压缩, 我们再用 WinRAR 打开刚刚压缩好的 myj ava.jar, 这时我们会看到如图 4.18 所示的层次结构:



图 4.18

显然, 这并不是我们所需要的结果, 我们需要的是: 压缩生成后的 jar 文件中所包含最顶层目录应该是 test1.Test1 类和 test2.Test2 类的顶级包名所对应的文件夹 test1 和 test2, 而不应该是 myj ava 文件夹。象上面这种情况, 即使 classpath 环境变量中包含了 myj ava.jar, java 也不会找到这两个的类。

接下来, 我们还是在命令行窗口中, 运行 c:\>cd myj ava, 进入 myj ava 目录, 运行 jar cvf myj ava.jar \*. 这时在 myj ava 目录中也会生成一个 myj ava.jar 文件。我们用 WinRAR 查看这次压缩生成的 myj ava.jar 文件, 它的层次结构如图 4.19 所示。



图 4.19

可见，这次压缩生成的 jar 文件才是我们所需要的结果。

上面的实验告诉我们，使用 jar 压缩文件夹时，在生成的 jar 文件中会保留在 jar 命令中所出现的路径名，所以，使用 jar 命令时一定要注意路径问题！

## 2) 快速查看 jar 包中是否有我们想找的类

通过本节开始的介绍，我们已经基本明白了.jar 文件就相当于一个文件夹。Java 根据 classpath 环境变量的设置去查找一个类时，首先是在.jar “文件夹” 中去找这个类的顶级包名所对应的文件夹。所以，当我们运行 WinRAR 对 jar 文件进行查看时，如果首先就没有看到该类的顶级包名所对应的目录名，则说明这个 jar 文件里根本就没有我们要查找的类，就不用再继续查找下去了。假设我们要查看 org.it315.MyClass1 的类是否存在于上面的 myjava.jar 中，用 WinRAR 打开 myjava.jar 后的结果如图 4.19 所示，没有看到 org 文件夹，我们就可以肯定 myjava.jar 中没有我们想找的类，如果看到了 org 文件夹，我们可以继续展开 org 文件夹，看其中是否包含 it315 文件夹，依此类推，最后看能否找到 MyClass1.class 文件。

|                              |     |
|------------------------------|-----|
| 第 4 章 面向对象 (下) .....         | 106 |
| 4.1 类的继承.....                | 106 |
| 4.1.1 继承的特点.....             | 106 |
| 4.1.2 子类对象的实例化过程.....        | 109 |
| 4.1.3 覆盖父类的方法.....           | 110 |
| 4.1.4 final 关键字 .....        | 112 |
| 多学两招： Java 中的常量              |     |
| 4.2 抽象类与接口.....              | 113 |
| 4.2.1 抽象类.....               | 113 |
| 4.2.2 接口(interface).....     | 113 |
| 4.3 对象的多态性.....              | 115 |
| 4.3.1 对象的类型转换.....           | 115 |
| 4.3.2 Object 类 .....         | 118 |
| 4.3.3 面向对象的多态性.....          | 119 |
| 独家见解： 接口的概念                  |     |
| 4.3.4 匿名内部类.....             | 122 |
| 4.4 异常.....                  | 124 |
| 4.4.1 了解异常.....              | 124 |
| 4.4.2 try... catch 语句 .....  | 125 |
| 小经验： 发挥 try...catch 语句的作用    |     |
| 4.4.3 throws 关键字.....        | 126 |
| 4.4.4 自定义异常与 Throw 关键字 ..... | 128 |
| 多学两招： 如何实现程序的跳转              |     |
| 4.4.5 finally 关键字 .....      | 130 |
| 4.4.6 异常的一些使用细节.....         | 132 |
| 4.4.7 Java 引入异常的好处.....      | 132 |
| 4.5 包.....                   | 132 |
| 4.5.1 package 语句及应用.....     | 132 |
| 指点迷津： 1. 类名相同不等同于同一类         |     |
| 2. 同名包的目录位置                  |     |
| 脚下留心： 不要被旧文件所迷惑              |     |

|                                   |     |
|-----------------------------------|-----|
| 4.5.2 import 语句及应用 .....          | 136 |
| 多学两招：父包和子包之间的关系                   |     |
| 小经验：常见错误                          |     |
| 4.5.3 JDK 中的常用包 .....             | 139 |
| 4.6 访问控制.....                     | 139 |
| 4.6.1 类成员的访问控制.....               | 140 |
| 4.6.2 类的访问控制.....                 | 140 |
| 多学两招：类名与文件名                       |     |
| 4.6.3 Java 的命名习惯.....             | 140 |
| 4.7 使用 jar 文件 .....               | 141 |
| 4.7.1 jar 文件包 .....               | 141 |
| 4.7.2 jar 命令详解 .....              | 142 |
| 4.7.3 使用 WinRAR 对 jar 文件进行查看..... | 145 |
| 指点迷津：1.注意 jar 文件的相对路径问题           |     |
| 2.快速查看 jar 包中的类                   |     |

# 第 5 章 多线程

## 5.1 如何创建与理解线程

在讲解线程之前，我们先讲解一下什么是进程。简单地说，在多任务系统中，每个独立执行的程序称为进程，也就是“正在进行的程序”。我们现在使用的操作系统一般都是多任务的，即能够同时执行多个应用程序，如我们接触最多的 Windows、Linux、Unix。实际情况是，操作系统负责对 CPU 等设备资源进行分配和管理，虽然这些设备某一时刻只能做一件事，但以非常小的时间间隔交替执行多个程序，就可以给人以同时执行多个程序的感觉。如果我们同时运行记事本程序的两个实例，这就是两个不同的进程。我有一个朋友对我说，Windows 真好，我在从 C 盘向 D 盘拷贝文件的同时，又可从 E 盘向 F 盘拷贝文件，拷贝效率大为提高。大家对此话有何感想？我一听，就知道他充其量只能算是业余计算机爱好者了，因为 CPU 只有一个，每个进程都有独立的代码和数据空间（进程上下文），在两个文件拷贝进程间切换需要额外的开销，反而比先执行完 C 盘向 D 盘的拷贝，再启动 E 盘向 F 盘的拷贝慢。

### 5.1.1 了解线程概念

一个进程中又可以包含一个或多个线程，一个线程就是一个程序内部的一条执行线索。在单线程中，程序代码按调用顺序依次往下执行，在这种情况下，当主函数调用了子函数，主函数必须等待子函数返回后才能继续往下执行，不能实现两段程序代码同时交替运行的效果。如果要一程序中实现多段代码同时交替运行，就需产生多个线程，并指定每个线程上所要运行的程序代码段，这就是多线程。

当程序启动运行时，就自动产生了一个线程，主函数 main 就是在这个线程上运行的，当我们不再产生新的线程时，我们的程序就是单线程的，比如我们以前的例子，它们都是单线程的。

创建多线程有两种方法：继承 Thread 类和实现 Runnable 接口，在下面的小节里，我们分别进行讲解。

### 5.1.2 用 Thread 类创建线程

Java 的线程是通过 java.lang.Thread 类来控制的，一个 Thread 类的对象代表一个线程，而且只能代表一个线程，通过 Thread 类和它定义的对象，我们可以获得当前线程对象、获取某一线程的名称，可以实现控制线程暂停一段时间等功能，关于 Thread 类的具体应用与讲解，我们将在文中稍后的地方逐步涉及，在学完本章后，大家也需通读一下 JDK 文档中有关 Thread 类的方法及说明，并动手编写一些小程序对其中的某些方法测试验证一下，以便对 Thread 类有更全面的了解与认识。下面，我们就开始通过程序来详细讲解吧！

程序清单：ThreadDemo1.java

```
public class ThreadDemo1
{
    public static void main(String args[])
    {
        new TestThread().run();
        while(true)
    }
}
```

代码块 1

```

    {
        System.out.println("main thread is running");
    }
}

class TestThread
{
    public void run()
    {
        while(true)
        {
            System.out.println(Thread.currentThread().getName() +
                " is running");
        }
    }
}

```

代码块 2

一个代码段被执行，一定是在某个线程上运行的，代码与线程密不可分，同一段代码可以与多个线程相关联，在多个线程上执行的也可以是相同的一段代码，好比多个火车售票处按相同的操作流程（相当程序代码）同时售票一样。在上面的代码中，我们使用 `Thread.currentThread()` 静态函数获得该代码当前执行时对应的那个线程对象。得到当前线程对象后，我们又调用了线程对象的 `getName()` 方法，取出当前线程的名称字符串。

代码块 1 处的代码能否运行呢？编译 `ThreadDemo1.java` 文件，并运行一下，看看结果如何？



图 5.1

屏幕上不停地打印出 `main is running`，而不是 `main thread is running`，这说明代码块 1 处的程序没有运行，因为代码块 2 先于代码块 1 运行，且代码块 2 为无限循环，代码块 1 永远没有机会运行。同时，我们也能够看到当前线程的名称为 `main`。

我们将代码进行如下修改（为了达到对比讲解，保持上下连贯性的效果，我们对修改过的地方进行注释，而不是彻底删除掉）：

程序清单： `ThreadDemo2.java`

```

public class ThreadDemo2
{
    public static void main(String args[])
    {
        new TestThread ().start(); /*run()*/
    }
}

```

```

        while(true)
    {
        2. 调用 TestThread 类的 start 函数 (从 Thread 类继承而来的)

        System.out.println("main thread is running");
    }

}

class TestThread extends Thread
{
    1. 让 TestThread 类继承 Thread 类

    public void run()
    {
        while(true)
        {

            System.out.println(Thread.currentThread().getName() +
                " is running");
        }
    }
}

```

上面的代码让 TestThread 类继承了 Thread 类，也就是 TestThread 类具有了 Thread 类的全部特点，程序没有直接调用 TestThread 类对象的 run 方法，而是调用了该类对象从 Thread 类继承来的 start 方法。运行一下，我们能够看到两个 while 循环处的代码同时交替运行：

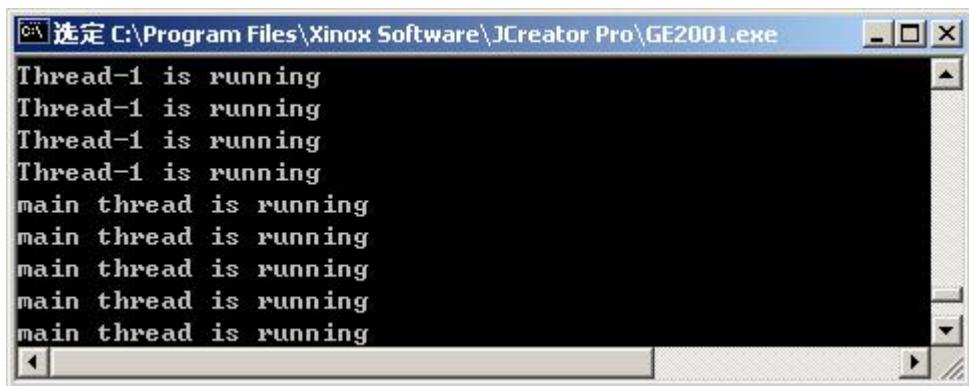


图 5.2

这就是我们要讲的多线程。单线程与多线程的区别，如图 5.3 所示。

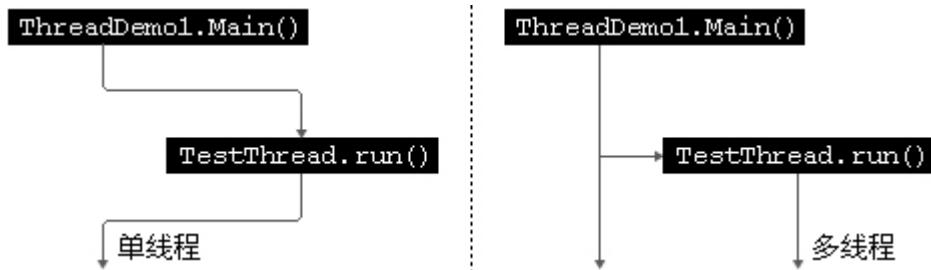


图 5.3

可见，在单线程中，main 函数必须等到 TestThread.run 函数返回后才能继续往下执行。而在多线程中，main 函数调用 TestThread.start()方法启动了 TestThread.run()函数后，main 函数不等待 TestThread.run 函数返回就继续运行，TestThread.run 函数在一边独自运行，不影响原来的 main 函数的运行，这好比将一个 1G 的 CPU 分成了两个 500M 的 CPU，在一个 CPU 上运行 main 函数，

而 `TestThread.run` 则在另一个 CPU 上运行，但它们都在向同一个显示器上输出，所以我们看到两个 `while` 循环处的代码同时交替运行。

同样，在代码段 `run()` 中，我们也可以通过线程的静态方法 `Thread.currentThread()` 得到当前线程实例对象。得到当前线程对象后，我们又调用了线程对象的 `getName()` 方法，取出当前线程的名称字符串

### 小结：

1. 要将一段代码在一个新的线程上运行，该代码应该在一个类的 `run` 函数中，并且 `run` 函数所在的类是 `Thread` 类的子类。倒过来看，我们要实现多线程，必须编写一个继承了 `Thread` 类的子类，子类要覆盖 `Thread` 类中的 `run` 函数，在子类的 `run` 函数中调用想在新线程上运行的程序代码。

2. 启动一个新的线程，我们不是直接调用 `Thread` 的子类对象的 `run` 方法，而是调用 `Thread` 子类对象的 `start`（从 `Thread` 类中继承的）方法，`Thread` 类对象的 `start` 方法将产生一个新的线程，并在该线程上运行该 `Thread` 类对象中的 `run` 方法，根据面向对象的多态性，在该线程上实际运行的是 `Thread` 子类（也就是我们编写的那个类）对象中的 `run` 方法。

3. 由于线程的代码段在 `run` 方法中，那么该方法执行完成以后线程也就相应的结束了，因而我们可以通过控制 `run` 方法中的循环条件来控制线程的终止。

查看 JDK 文档中 `Thread` 类，发现有许多构造方法，在我们上面的例子中，线程对象都是通过 `Thread()` 构造方法创建的，线程将调用线程对象中的 `run()` 方法作为其运行代码，具体细节大家已在前面看到了，请大家思考一个问题：如果 `Thread` 类的子类（在上面的例子中即 `TestThread` 类）没有覆盖 `run` 方法，编译和运行时有明显的错误或异常吗？运行结果是怎样的呢？请读者自己做试验来证明一下。

提示：程序会调用 `Thread` 类中的 `run` 方法，而该 `run` 方法什么也不做，所以，新的线程刚一产生就结束了，这样创建出来的线程对我们的程序来说毫无意义。

直接在程序中写 `new Thread().start();` 这样的语句，编译和运行时有明显的错误或异常吗？运行结果是怎样的呢？由于我们的线程对象不是通过 `Thread` 子类创建的，而是通过 `Thread` 类直接创建的，新的线程将直接调用 `Thread` 类中的 `run()` 方法，所以答案与上面的一样。

使用 `Thread()` 构造方法，适用于覆盖了 `run` 方法的 `Thread` 子类创建线程对象的情况，如我们前面的例子那样。

### 5.1.3 使用 `Runnable` 接口创建多线程

在 JDK 文档中，我们还看到了一个 `Thread(Runnable target)` 构造方法，从 JDK 文档中查看 `Runnable` 接口类的帮助，该接口中只有一个 `run()` 方法，当我们使用 `Thread(Runnable target)` 方法创建线程对象时，需为该方法传递一个实现了 `Runnable` 接口的类对象，这样创建的线程将调用那个实现了 `Runnable` 接口的类对象中的 `run()` 方法作为其运行代码，而不再调用 `Thread` 类中的 `run` 方法了。我们可以将上面的例子改写成下面这样：

程序清单：`ThreadDemo3.java`

```
public class ThreadDemo3
{
    public static void main(String args[])
    {
        //new TestThread ().start();
        TestThread tt= new TestThread(); //创建 TestThread 类的一个实例

        Thread t= new Thread(tt); //创建一个 Thread 类的实例
```

```

        t.start(); //使线程进入 Runnable 状态
        while(true)
        {
            System.out.println("main thread is running");
        }
    }

class TestThread implements Runnable //extends Thread
{
    public void run() //线程的代码段，当执行 start() 时，线程从此处开始执行
    {
        while(true)
        {
            System.out.println(Thread.currentThread().getName() +
                " is running");
        }
    }
}

```

运行的结果和前面一样。

#### 5.1.4 两种实现多线程方式的对比分析

既然直接继承 Thread 类和实现 Runnable 接口都能实现多线程，那么这两种实现多线程的方式在应用上有什么区别呢？我们到底该用哪一个好呢？

为了回答这个问题，我们通过编写一个应用程序，来进行比较分析。我们用程序来模拟铁路售票系统，实现通过四个售票点发售某日某次列车的 100 张车票，一个售票点用一个线程来表示。

我们首先这样编写这个程序：

程序清单： ThreadDemo4.java

```

public class ThreadDemo4
{
    public static void main(String [] args)
    {
        ThreadTest t=new ThreadTest();
        t.start();
        t.start();
        t.start();
        t.start();
    }
}

class ThreadTest extends Thread
{
    private int tickets=100;
    public void run()

```

```

{
    while(true)
    {
        if(tickets>0)
            System.out.println(Thread.currentThread().getName() +
                " is saling ticket " + tickets--);
    }
}
}

```

在上面的代码中，我们用 ThreadTest 类模拟售票处的售票过程，run 方法中的每一次循环都将总票数减 1，模拟卖出一张车票，同时该车票号打印出来，直到剩余的票数到零为止。在 ThreadDemo4 类的 main 方法中，我们创建了一个线程对象，并重复启动四次，希望通过这种方式产生四个线程，结果怎样呢？

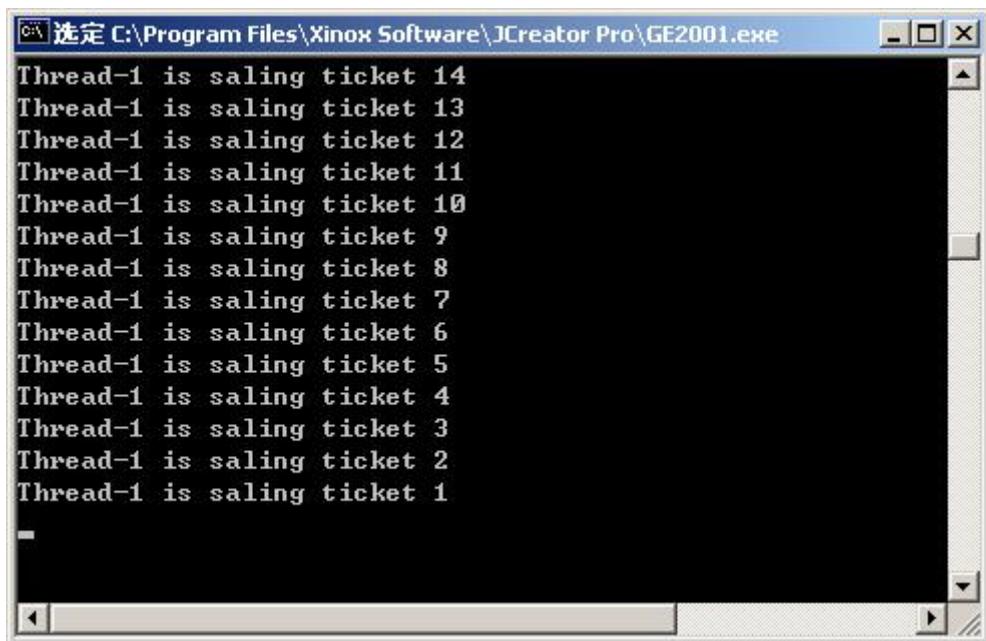


图 5.4

从上面运行结果上，我们发现其实只有一个线程在运行，这个结果告诉我们：一个线程对象只能启动一个线程，无论你调用多少遍 start() 方法，结果都只有一个线程。

我们接着修改 ThreadDemo4，在 main 方法中创建四个 ThreadTest 对象：

```

public class ThreadDemo4
{
    public static void main(String [] args)
    {
        new ThreadTest().start();
        new ThreadTest().start();
        new ThreadTest().start();
        new ThreadTest().start();
    }
}

```

这下达到我们的目的了吗？

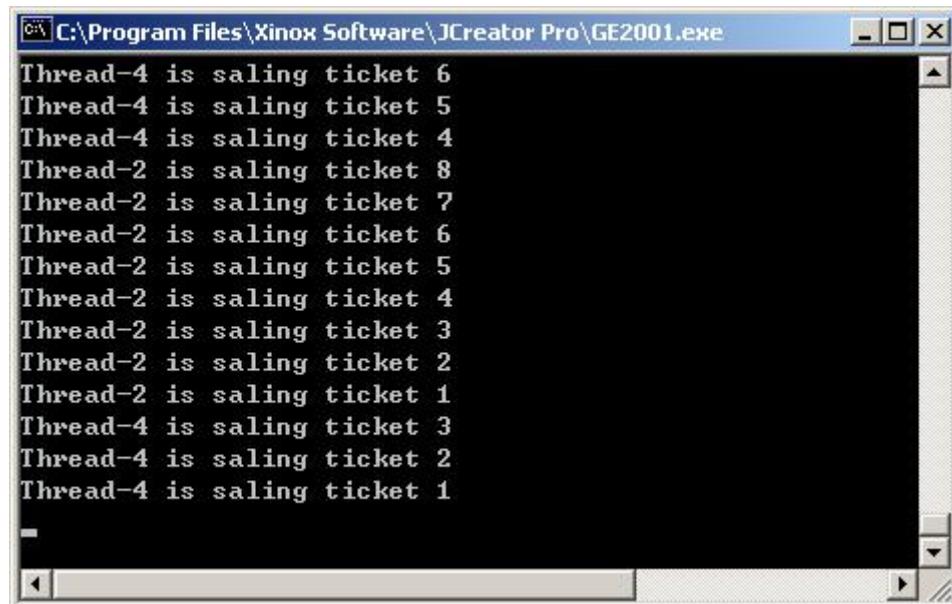


图 5.5

从上面结果上，我们看到的结果是每个票号都被打印了四遍，即四个线程各自卖各自的 100 张票，而不是去卖共同的 100 张票。这种情况是怎样造成的呢？我们需要的是，多个线程去处理同一资源，一个资源只能对应一个对象，在上面的程序中，我们创建了四个 ThreadTest 对象，就等于创建了四个资源，每个 ThreadTest 对象中都有 100 张票，每个线程在独立地处理各自的资源。

经过了这些曲折，这些实验，分析，我们可以总结出，要实现这个铁路售票模拟程序，我们只能创建一个资源对象（该对象中包含要发售的那 100 张票），但要创建多个线程去处理这同一个资源对象，并且每个线程上所运行的是相同的程序代码。再回顾一下使用接口编写多线程的过程，大家应该能够自己写出这个程序了，请大家自己写完后，再来参看我下面给出的程序代码。

程序清单：ThreadDemo5.java

```
public class ThreadDemo5
{
    public static void main(String [] args)
    {
        ThreadTest t=new ThreadTest();
        new Thread(t).start();
        new Thread(t).start();
        new Thread(t).start();
        new Thread(t).start();
    }
}
class ThreadTest implements Runnable
{
    private int tickets=100;
    public void run()
    {
        while(true)
        {
            if(tickets>0)
```

```

        System.out.println(Thread.currentThread().getName()
        + " is saling ticket " + tickets--);
    }
}
}

```

在上面的程序中，我们创建了四个线程，每个线程调用的是同一个 ThreadTest 对象中的 run() 方法，访问的是同一个对象中的变量(tickets)的实例，这个程序满足了我们的需求。我们在 Windows 上可以启动多个记事本程序，也就是多个进程使用的是同一个记事本程序代码，明白了这个道理后，大家就应该对多个线程上运行完全相同的程序代码不再难以理解了。

可见，实现 Runnable 接口相对于继承 Thread 类来说，有如下显著的好处：

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。我们经常碰到这样一种情况，即当我们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，这个类就只能采用实现 Runnable 接口的方式了。

3、有利于程序的健壮性，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，即它们共享相同的数据。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。

### 5.1.5 后台线程与联合线程

#### 1. 后台线程与 setDaemon 方法

在上面售票系统的例子中，我们在 main 方法中创建并启动新的线程后，main 方法便结束了，主线程也就随之结束了，这样的情况下，我们的这个 java 程序是否也随之结束呢？从程序运行的结果上，我们已经看到，虽然 main 线程结束了，但整个 java 程序没有随之结束，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运行，这个进程就会结束。前台线程是相对后台线程而言的，我们按前面的方式产生的线程都是前台线程，如果我们对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。我们下面来看看，进程中只有后台线程运行时，进程就会结束的情况。

```

程序清单：DaemonThread.java
public class DaemonThread
{
    public static void main(String[] args)
    {
        ThreadTest t=new ThreadTest();
        Thread tt=new Thread(t);
        tt.setDaemon(true);
        tt.start();
    }
}

```

```

class ThreadTest implements Runnable
{
    public void run()
    {
        while(true)
        {
            System.out.println(Thread.currentThread().getName()+" is running.");
        }
    }
}

```

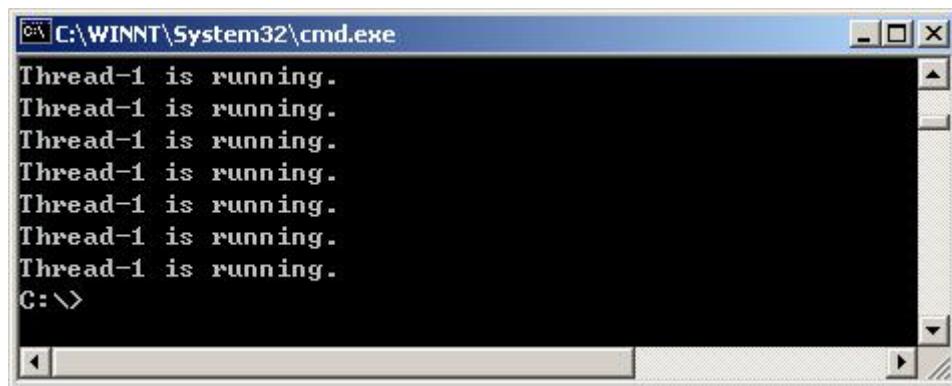


图 5.6

从上面的程序和运行结果上，我们看到：虽然我们创建了一个无限循环的线程，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。

## 2. 联合线程与 `join` 方法

在讲到联合线程之前，我们先来看下面的这段程序。

程序清单：`JoinThread.java`

```

public class JoinThread
{
    public static void main(String[] args)
    {
        ThreadTest t=new ThreadTest();
        Thread pp=new Thread(t);
        pp.start();
        int i=0;
        while(true)
        {
            if(i==100)
            {
                try
                {
                    pp.join();
                }

```

```

        catch(Exception e)
        {
            System.out.println(e.getMessage());
        }
    }
    System.out.println("main Thread "+i++);
}
}

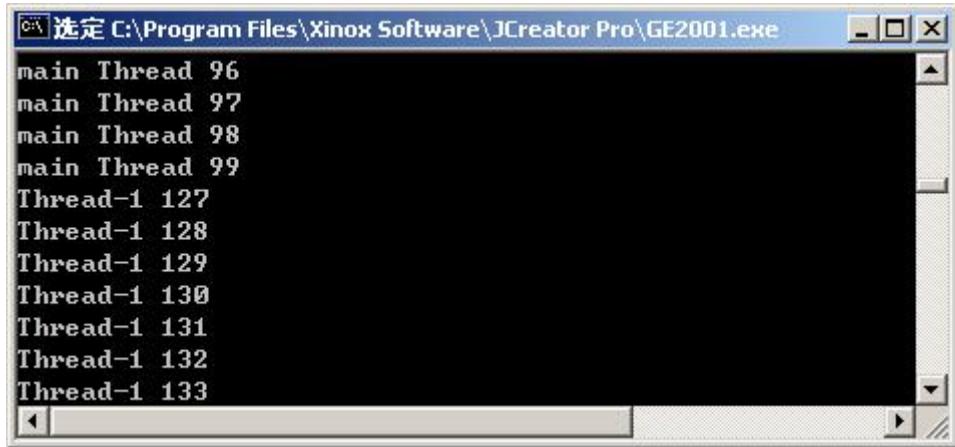
class ThreadTest implements Runnable
{
    public void run()
    {
        String str=new String();
        int i=0;
        while(true)
        {
            System.out.println(Thread.currentThread().getName()+" "+i++);
        }
    }
}

```

在上面的程序中用到了 `Thread` 类的 `join` 方法，即 `pp.join();` 语句，它的作用是把 `pp` 所对应的线程合并到调用 `pp.join();` 语句的线程中。在 `main` 线程中的循环计数达到 100 之前，我们看到 `main` 线程和 `Thread-1` 线程交替执行的情况，如图 5.7 中打印出来的结果。

图 5.7

在 `main` 线程中的循环计数达到 100 之后，我们看到只有 `Thread-1` 线程执行的情况，如图 5.7 中打印出来的结果。



The screenshot shows a terminal window titled "选定 C:\Program Files\Xinox Software\JCreator Pro\GE2001.exe". The window displays a series of text entries representing thread activity. The entries are as follows:

```
main Thread 96
main Thread 97
main Thread 98
main Thread 99
Thread-1 127
Thread-1 128
Thread-1 129
Thread-1 130
Thread-1 131
Thread-1 132
Thread-1 133
```

图 5.8

可见，Thread-1 线程中的代码被并入到了 main 线程中，也就是 Thread-1 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，我们发现，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，在看看程序运行的结果。

### 5.1.6 多线程在实际中的应用

多线程的用途非常广泛，我给大家举几个应用多线程的例子，以便大家更好地理解和运用多线程。

1. 当我们编写一个网络聊天程序时，如果使用的是单线程编程模式，通常的程序流程为图 5.9：

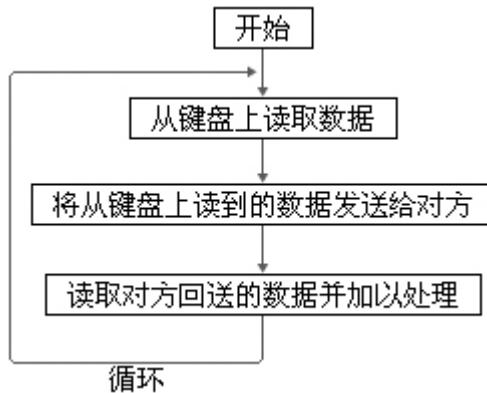


图 5.9

按上面的流程编写出来的程序将会产生这样的问题：

- 1) 如果一方从键盘上读取了数据并发送给了对方，程序运行到“读取对方回送的数据”并一直等待对方回送数据，如果对方没有回应，程序不能再做其他任何事情，这时程序处于阻塞状态，即使用户想正常终止程序运行都不可能，更不能实现“再给对方发送一条信息，催促对方赶快应答”这样的事情了。
- 2) 如果程序没有事先从键盘上读取数据并向外发送，程序将一直在“从键盘上读取数据”处阻塞，即使有数据从网上发送过来，程序无法到达“读取对方回送的数据”处，程序将不能收到别处先主动发送过来的数据。

我们用多线程模式编写这个网络聊天程序，其程序流程为图 5.10 所示：

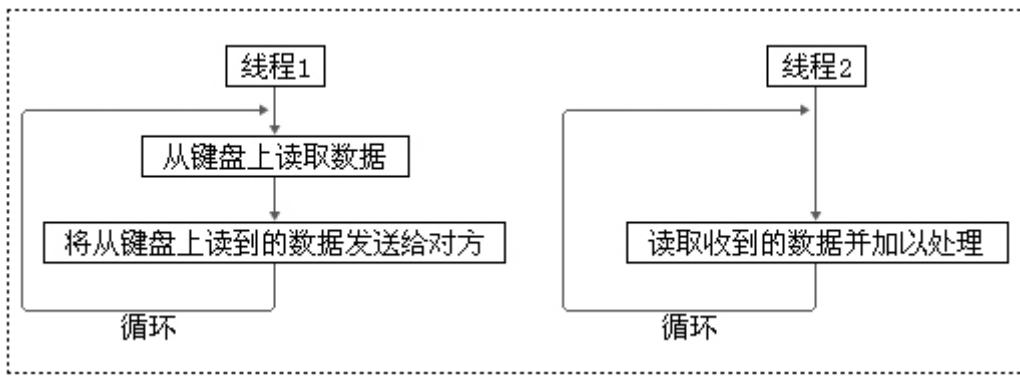


图 5.10

我们使用多线程模式编写这个程序后，发送过程与接收过程在两个不同的线程上运行，彼此之间没有任何约束，用户可以随心所欲地发送和接收数据了。

2. 如果我们程序要将数据库一个表中的所有记录复制到另外一个表中，我们利用循环将源表中的记录逐一取出并插入到目标表中，当表中的记录有百万条以上，这个复制过程的时间将非常长，当我们想放弃这次复制，在单线程模式下，我们所能做的只有一直无奈地等到这个费时的操作完成或是做出类似关机一样的野蛮行为了。我们可以用多线程编写这样的应用，由主线程负责表记录的复制，复制代码放在一个循环语句中，循环条件由一个 boolean 变量来控制。如：

```
boolean bFlag = true;
while(bFlag)
{
    // 复制程序
}
```

创建一个新的线程，该线程与用户交互，接收用户的键盘输入，当接收到用户的停止命令时，新线程将主线程的循环条件 `bFlag` 设置为假，即通知主线程在下次检查循环条件时结束复制过程，具体的程序代码见本章最后的小节（如何控制线程的生命）。

3. 多线程中的另外一个比较典型的例子就是 WWW 服务器，我们都应该知道，WWW 的服务器是可以同时为若干个浏览器服务的，这就需要它为每一个来访者都创建一个线程。如果它是单线程的话，在一个时间段就只能为一个人服务，其它人只有干等的份了。

## 5.2 多线程的同步

### 5.2.1 线程安全问题

在上面卖车票的程序代码中，我们极有可能碰到一种意外，这就是同一张票号被打印两次或多次，也可能出现打印出 0 甚至负数的票号。这个意外发生在下面这部分代码处：

```
if(tickets>0)
    System.out.println(Thread.currentThread().getName() +
        " is selling ticket " + 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() +  
    " is saling ticket " + tickets--);
```

但此刻 `tickets` 的值已变为 0，屏幕打印出的将是 0。

我们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 `ThreadTest` 代码如下：

```
class ThreadTest implements Runnable  
{  
    private int tickets=100;  
    public void run()  
    {  
        while(true)  
        {  
            if(tickets>0)  
            {  
                try  
                {  
                    Thread.sleep(10);  
                }  
                catch(Exception e)  
                {  
                    System.out.println(e.getMessage());  
                }  
                System.out.println(Thread.currentThread().getName() +  
                    " is saling ticket " + tickets--);  
            }  
        }  
    }  
}
```

在上面的程序代码中，我们故意造成线程执行完 `if(tickets>0)` 语句后，执行 `Thread.sleep(10)`，以让出 CPU 给别的线程，让读者直接看到在这一时刻发生线程切换的情况。查看 JDK 文档中关于 `Thread.sleep()` 方法的定义如下：

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

注：线程的睡眠是可以被打断的，通过 `Thread.interrupt()`，当然一个线程中可以调用另外一个线程的 `interrupt()`，线程的睡眠被打断后进入 `Runnable` 状态。由于 `Thread.sleep()` 的定义中通过 `throws` 关键字声明该方法中有可能引发异常，所以，我们的程序在调用该方法时，必须使用 `try...catch` 代码块处理，否则，编译将出错，这正是 java 语言强健性的一个方面。

编译运行上面的程序，屏幕上打出的最后几行结果如下：

```
Thread-2 is saling ticket 3  
Thread-3 is saling ticket 2  
Thread-4 is saling ticket 1  
Thread-1 is saling ticket 0  
Thread-2 is saling ticket -1
```

Thread-3 is saling ticket -2

票号被打印出来了负数，这就显示了同一张票被卖了四次的这种意外发生。

这种意外问题，就是我们有时听到专业人士谈到的“线程安全”问题，也许某天就有人问你，你写的类是线程安全的吗？就是说你编写的那个类的同一个实例对象的方法在多个线程被调用，是否会出现类似上面的意外，不要到时听不明白人家的意思，那会给人很业余的印象。

## 5.2.2 同步代码块

如何避免上面的这种意外？如何让我们的程序是线程安全的呢？这就是我们要为大家讲解的如何实现线程间的同步问题。要解决上面的问题，我们必须保证下面这段代码的原子性：

```
if(tickets>0)
    System.out.println(Thread.currentThread().getName() +
        " is saling ticket " + tickets--);
```

即当一个线程运行到 if(tickets>0)后，CPU 不去执行其他线程中的、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他线程中的有关代码块。这段代码就好比一座独木桥，任一时刻，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。

我们修改一下 ThreadTest 类，使其具有线程同步效果，代码如下：

```
class ThreadTest implements Runnable
{
    private int tickets=100;
    String str = new String ("");
    public void run()
    {
        while(true)
        {
            synchronized(str)
            {
                if(tickets>0)
                {
                    try
                    {
                        Thread.sleep(10);
                    }
                    catch(Exception e)
                    {
                        System.out.println(e.getMessage());
                    }
                    System.out.println(Thread.currentThread().getName()+
                        " is saling ticket " + tickets--);
                }
            }
        }
    }
}
```

在上面的代码中，我们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，其他线程才能进入同步代码块内运行。synchronized 语句的格式为：

```
synchronized(object){代码段} //object 可以是任意的一个对象
```

所以，程序中用 String str = new String ("")随便产生了一个对象，用在后面的同步代码块。编译运行后，程序在屏幕上打印出的内容如下：

```
Thread-2 is selling ticket 3  
Thread-3 is selling ticket 2  
Thread-4 is selling ticket 1
```

程序打印完最后的票号 1 后，就再也没有继续在屏幕上打印输出了，相当于这四个线程卖完最后一张票后，就停止了卖票的操作，这也向读者说明了改写后的程序是线程安全的了。

任意类型的对象都有一个标志位，该标志位具有 0、1 两种状态，其开始状态为 1，当执行 synchronized(object) 语句后，object 对象的标志位变为 0 状态，直到执行完整个 synchronized 语句中的代码块后又回到 1 状态。一个线程执行到 synchronized(object) 语句处时，先检查 object 对象的标志位，如果为 0 状态，表明已经有另外的线程的执行状态正在有关的同步代码块中，这个线程将暂时阻塞，让出 CPU 资源，直到另外的线程执行完有关的同步代码块，将 object 对象的标志位恢复到 1 状态，这个阻塞就被取消，线程能够继续往下执行，并将 object 对象的标志位变为 0 状态，防止其他线程再进入有关的同步代码块中。如果有多个线程因等待同一对象的标志位而处于阻塞状态时，当对象的标志位恢复到 1 状态时，只会有一个线程能够继续运行，其他线程仍然处于阻塞等待状态。我们反复提到有关的同步代码块，是指不仅同一个代码块在多个线程间可以实现同步（象上面例子一样），若干个不同的代码块也可以实现相互之间的同步，只要各 synchronized(object) 语句中的 object 完全是同一个对象就可以。上面的讲解主要是为了达到通俗易懂的目的，但有时与同行交流，或是参看相关书籍时，我们也不得不掌握一些专业术语，下面是对刚才的内容用专业术语进行的陈述。

当线程执行到 synchronized 的时候检查传入的实参对象，并得到该对象的锁旗标（就是我们上面讲的标志位）。如果得不到，那么此线程就会被加入到一个与该对象的锁旗标相关连的等待线程池中，一直等到该对象的锁旗标被归还，池中的等待线程就会得到该旗标，然后继续执行下去。当线程执行完成同步代码块时，就会自动释放它占有的同步对象的锁旗标。一个用于 synchronized 语句中的对象称为一个监视器，当一个线程获得了 synchronized(object) 语句中的代码块的执行权，即意味着它锁定了监视器，在一段时间内，只能有一个线程可以锁定监视器。所有其他的线程在试图进入已锁定的监视器时将被挂起，直到锁定了监视器的线程执行完 synchronized(object) 语句中的代码块，即监视器被解锁为止，另外的线程才可以进入并锁定监视器，一个刚锁定了监视器的线程在监视器被解锁后可以再次进入并锁定同一监视器，好比篮球运动员的篮球出手后可以再次去抢回来一样。另外当在同步块中遇到 break 语句或扔出异常时，线程也会释放该锁旗标。

其实，程序并不能控制 CPU 的切换，程序是不可能抱着 CPU 的大腿不让他走的。当 CPU 进入了一段同步代码块中执行，CPU 是可以切换到其他线程的，只是在准备执行其他线程的代码时，发现其他线程处于阻塞状态，CPU 又会回到先前的线程上。这个过程就类似于幸运之神刚一光顾其他有关线程，没想到吃了个闭门羹，便又离开了。

大家也看到同步处理后，程序的运行速度比原来没有使用同步处理前更慢了，因为系统要不停地对同步监视器进行检查，需要更多的开销。同步是以牺牲程序的性能为代价的，如果我们能够确定程序没有安全性的问题，就没必要使用同步控制。

我们将程序代码略作修改，改变 String str = new String ("") 这行代码的位置，将 str 对象放到 run 方法中定义：

```

class ThreadTest implements Runnable
{
    private int tickets=100;
    public void run()
    {
        String str = new String ("");
        while(true)
        {
            synchronized(str)
            {
                if(tickets>0)
                {
                    try
                    {
                        Thread.sleep(10);
                    }
                    catch(Exception e)
                    {
                        System.out.println(e.getMessage());
                    }
                    System.out.println(Thread.currentThread().getName()+
                        " is saling ticket " + tickets--);
                }
            }
        }
    }
}

```

编译运行后，发现结果又不正常了，问题出在什么地方呢？在这个程序中，`run` 方法被四个线程所调用，相当于 `run` 方法被调用了四次，对每一次调用，程序都产生一个不同的 `str` 局部对象，这四个线程使用的同步监视器完全是四个不同的对象，所以彼此之间不能同步。

### 5.2.3 同步函数

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，我们按下面的代码修改 `ThreadTest` 类：

```

class ThreadTest implements Runnable
{
    private int tickets=100;
    public void run()
    {
        while(true)
        {
            sale();
        }
    }
}

```

```

    }
    public synchronized void sale()
    {
        if(tickets>0)
        {
            try
            {
                Thread.sleep(10);
            }
            catch(Exception e)
            {
                System.out.println(e.getMessage());
            }
            System.out.println(Thread.currentThread().getName()+
                " is saling ticket " + tickets--);
        }
    }
}

```

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，可见，在函数定义前使用 `synchronized` 关键字也能够很好实现线程间的同步。

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 `synchronized` 修饰的方法（获得监视器），其他线程就不能进入同一个对象的所有使用了 `synchronized` 修饰的方法，直到第一个线程执行完它所进入的 `synchronized` 修饰的方法为止（离开监视器）。

#### 5.2.4 代码块与函数间的同步

我们掌握了同步代码块与同步函数两种方式，我们能否在代码块与函数之间实现同步呢？我们只要仔细分析线程同步的机理，线程同步靠的是检查同一对象的标志位，只要让代码块与函数使用同一个监视器对象，答案就应该是肯定的。那么，函数中用的监视器对象是哪个呢？如果你东想西想，左看右看，都没有发现合适的对象，那你为什么不去大胆的猜测？不管我们编写的类的内部结构怎样，类中的非静态方法始终都能访问到的一个对象就是这个对象本身，即 `this`，看来同步函数所用的监视器对象只能是 `this`。我们还是通过程序来验证一下我们的想法，我们在程序中产生两个线程，在线程的 `run` 方法中，我们通过检查 `str` 变量的取值，来决定是调用函数，还是调用代码块。第一个线程启动的 `run` 方法检查 `str` 变量的取值不等于“`method`”，就调用程序中的代码块运行。接着，我们将 `str` 变量的值修改成“`method`”，再启动第二个线程，这样第二个线程的 `run` 方法就会调用函数。

程序清单：ThreadDemo6.java

```

public class ThreadDemo6
{
    public static void main(String [] args)
    {
        ThreadTest t=new ThreadTest();
        new Thread(t).start(); //这个线程调用同步代码块
    }
}

```

```

        t.str=new String("method");
        new Thread(t).start(); //这个线程调用同步函数
    }
}

class ThreadTest implements Runnable
{
    private int tickets=100;
    String str = new String ("");
    public void run()
    {
        if(str.equals("method"))
        {
            while(true)
            {
                sale();
            }
        }
        else
        {
            synchronized(str)
            {
                if(tickets>0)
                {
                    try
                    {
                        Thread.sleep(10);
                    }
                    catch(Exception e)
                    {
                        System.out.println(e.getMessage());
                    }
                    System.out.println(Thread.currentThread().getName()+
                            " is saling ticket " + tickets--);
                }
            }
        }
    }

    public synchronized void sale()
    {
        if(tickets>0)
        {
            try
            {
                Thread.sleep(10);

```

```

        }
        catch(Exception e)
        {
            System.out.println(e.getMessage());
        }
        System.out.println(Thread.currentThread().getName()+
            " is saling ticket " + tickets--);
    }
}
}

```

注意，程序中的 `if(str.equals("method"))`，不能写成 `if(str=="method")`，这是新手在实际工作中常犯的错误，许多老手在这个问题上也都有过切肤之痛。其实，老手就是在大量的实践中，犯下众多错误，然后经过反复调试、观察、比较，最后总结、积累经验的过程中成长起来的。老手曾经碰到过的错误，一般也是新手极容易遇到的，本书除了讲解知识点本身，始终贯彻一种编程思想外，也穿插了许多老手曾经或经常碰到过的错误细节的分析，以达到尽量帮助初学者少走弯路的目的。这样，即使是初学者学完本书后，也能获得许多实际工作中的经验，那种仿佛自己真的经历过的感受一定会令人刻骨铭心，记忆深刻的。

上面程序的代码块使用的是 `str` 这个对象来作为同步监视器的，而我分析同步函数使用的监视器是 `this` 对象，这是两个不同的对象，程序运行的两个线程是不能同步的。因为我们这里只使用了两个线程，只要程序在屏幕上最后打印出来的票号为 0，就说明了这两个线程是不同步的，就达到了我的讲解过程中的第一个目的。读者对这个程序的实验结果有时候正如我所料，但有的时候也并非如此，程序在屏幕上最后打印出来的票号为 1，屏幕上打印的结果的最后几行如下：

```

Thread-2 is saling ticket 3
Thread-1 is saling ticket 2
Thread-2 is saling ticket 1

```

从运行的结果上来看，这两个线程之间似乎是同步的。在一次课堂上的讲解中，突然出现了这个“所分析的与所看到的结果不一样”的令人尴尬的问题！

凭借经验，我当场就开始怀疑这两个线程要么都是调用的代码块，要么都是调用的函数，不是我们期望的一个线程调用代码块，另外一个线程去调用函数。为了证实我的怀疑，我们只要在 `sale` 函数的开始处，随便增加一条打印一行检测字符串的语句，重新编译运行程序，如果屏幕每次打印票号时，也都打印出了我们的检测字符串，或者一次都没有打印出我们的检测字符串，就证实了我的怀疑。修改后的 `sale` 方法如下：

```

public synchronized void sale()
{
    if(tickets>0)
    {
        try
        {
            Thread.sleep(10);
        }
        catch(Exception e)
        {
            System.out.println(e.getMessage());
        }
    }
}

```

```

        System.out.print("Test: ");//新加的检测语句
        System.out.println(Thread.currentThread().getName()+
            " is saling ticket " + tickets--);
    }
}

```

打印的结果显示，果然这两个线程调用的都是 `sale` 方法，这种情况又是什么原因造成的呢？这是因为，我们产生并启动第一个线程，这个线程不见得马上就开始运行，CPU可能还在原来的 `main` 线程上运行，并将 `str` 变量设置成了"method"，等到第一个线程真正开始运行时，此刻检查到的 `str` 变量的值是"method"，所以，它也去调用 `sale` 方法来运行。为了让第一个线程启动后马上就能得到 CPU，我们可以在修改 `str` 变量值之前，让 `main` 线程暂停哪怕是仅仅的一毫秒就能达到我们的目的。对 `main` 方法进行修改后的代码如下：

```

public static void main(String [] args)
{
    ThreadTest t=new ThreadTest();
    new Thread(t).start(); //这个线程调用同步代码块
    /*让主线程暂停一毫秒，为直观，将 try...catch 都写在了一行上*/
    try{Thread.sleep(1); }catch(Exception e){}
    t.str=new String("method");
    new Thread(t).start(); //这个线程调用同步函数
}

```

重新编译后的运行结果的最后几行如下：

```

Test: Thread-2 is saling ticket 3
Thread-1 is saling ticket 2
Test: Thread-2 is saling ticket 1
Thread-1 is saling ticket 0

```

从运行的结果上，我们看到了与我们先前分析的结果一样，由于代码块和函数所使用的同步监视器对象不一样，所以，他们没有同步。

由于有的人没有考虑线程启动和运行的时机及共享数据的问题，会根据刚开始的程序的运行的结果，认为上面的程序是能够在线程间同步的，并作为一个经验心得向人炫耀。可见，在计算机编程过程中，我们有时候会因为自己知识的不全面而作出错误的结论。

我们接着将同步语句 `synchronized(str)` 改为 `synchronized(this)` 后运行，就能看到两个线程同步了，通过这两种运行结果的对比，验证了我们对同步函数所用的监视器对象就是 `this` 对象的猜测，因此，我们的结论是：完全可以实现代码块与函数之间的同步。

当多个线程共享数据时，必须防止一个线程对共享数据仅仅进行了部分操作就退出的情况出现，在这种情况下会破坏数据的一致性。读者也可以考虑一下另外一种应用情况，如果线程 1 要与线程 2 同步，线程 3 要与线程 4 同步，必须有两个不同对象，线程 1 与线程 2 的同步由一个对象来控制，线程 3 与线程 4 的同步由另外一个对象来控制，线程 1、2 与线程 3、4 之间没有任何关系。

## M 脚下留心：

- 所有访问共享数据的代码段，必须都使用 `synchronized` 关键字同步化同一个对象，从而实现对该代码段的互斥访问（也就是线程间的相互排斥，有我没你，有你没我）。如果程序中有类似下面的代码，我们也应立即引起警觉，这段程序代码在多线程中是很容易出现问题的。

```

public void push(char c)
{

```

```

        data[idx]=c;
        idx++;
    }
}

```

如果一个线程刚执行完 push 方法中的 data[idx]=c 语句，CPU 便切换到了另外一个线程上执行 push 方法，第二个线程将覆盖掉第一个线程执行的 data[idx]=c 语句的结果。

2. 另外，共享访问的数据，应当是类的 private 数据成员，从而禁止来自类外的随意访问破坏数据一的一致性。

在实际项目中，多线程安全问题会有许多。多线程访问共享数据，要十分小心，线程同步的错误十分隐蔽，总是不能马上发现，且极难排查。对一个大的程序，如果不事先仔细考虑，等到程序写完了，在运行过程中发现程序的不稳定性后，再去查找，就更是困难和费时了。只要程序没有多线程安全问题，我们就不应该使用同步技术，因为源程序调用了同步方法，需要额外的监视器检查，运行效率要低些。

### 5.2.5 死锁问题

死锁是一种少见的、而且难于调试的错误，在两个线程对两个同步对象具有循环依赖时，就会出现死锁。例如，一个线程进入对象 X 的监视器，而另一个对象进入了对象 Y 的监视器，这时进入 X 对象监视器的线程如果还试图进入 Y 对象的监视器就会被阻隔，接着进入 Y 对象监视器的线程如果试图进入 X 对象的监视器也会被阻隔，这样两个线程都处于挂起状态。程序发生死锁后最明显的特征就是程序的运行处于停滞不前状态。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，发生了下面的问题：

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。

在下面的例子中，我们创建了两个类 A 和 B，它们分别具有方法 foo 和 bar，在调用对方的方法前，foo 和 bar 都睡眠一会儿。主类 DeadLock 创建 A 和 B 实例，然后，产生第二个线程以构成死锁条件。foo 和 bar 使用 sleep 来强制死锁条件出现。在现实程序中死锁是较难发现的：

程序清单：DeadLock.java

```

class A
{
    synchronized void foo(B b)
    {
        String name=Thread.currentThread().getName();
        System.out.println(name+ " entered A.foo ");
        try
        {
            Thread.sleep(1000);
        }
        catch(Exception e)
        {
            System.out.println(e.getMessage());
        }
    }
}

```

```

        System.out.println(name+ " trying to call B.last()");
        b.last();
    }

    synchronized void last()
    {
        System.out.println("inside A.last");
    }
}

class B
{
    synchronized void bar(A a)
    {
        String name=Thread.currentThread().getName();
        System.out.println(name + " entered B.bar");
        try
        {
            Thread.sleep(1000);
        }
        catch(Exception e)
        {
            System.out.println(e.getMessage());
        }
        System.out.println(name + " trying to call A.last()");
        a.last();
    }

    synchronized void last()
    {
        System.out.println("inside A.last");
    }
}

class Deadlock implements Runnable
{
    A a=new A();
    B b=new B();
    Deadlock()
    {
        Thread.currentThread().setName("MainThread");
        new Thread(this).start();
        a.foo(b); //get lock on a in this thread.
        System.out.println("back in main thread");
    }

    public void run()
    {
        Thread.currentThread().setName("RacingThread");
    }
}

```

```

        b.bar(a); //get lock on a in other thread.
        System.out.println("back in other thread");
    }
}

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

```

运行结果如下：

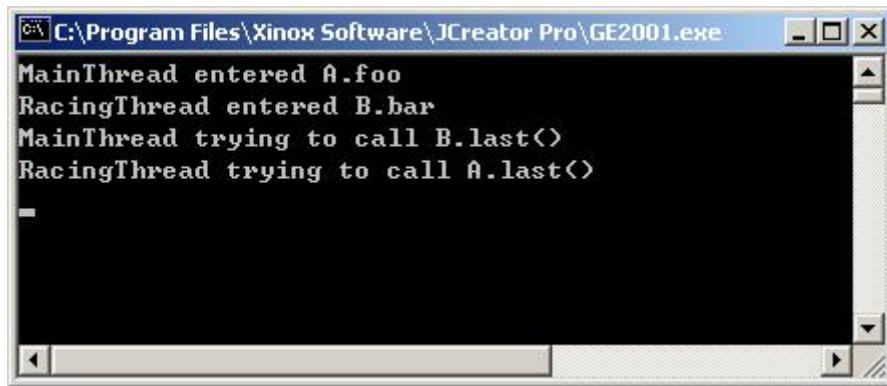


图 5.11

从运行结果可以看出，RacingThread 进入了 b 的监视器，然后又在等待 a 的监视器。同时 MainThread 进入了 a 的监视器并等待 b 的监视器。这个程序永远不会有完成。

## 5.3 线程间的通信

### 5.3.1 问题的引出

我们通过这样的一个应用来讲解线程间的通信。有一个数据存储空间，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。我们的应用包含两个线程，一个线程向数据存储空间添充数据（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我们考虑：

第一个意外，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 5.12 表示：



图 5.12

第二个意外，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。

### 5.3.2 问题如何解决

我们先来构思这个程序例子，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，因此我们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
        }
    }
}
```

当我们的程序写到这里，我们发现我们还需要定义一个新的数据结构来作为数据存储空间。

```
class Q
{
    String name;
    String sex;
}
```

Producer 和 Consumer 中的 run 函数都需要操作类 Q 的同一个对象实例，接下来，我们对 Producer 和 Consumer 这两个类作如下修改，顺便也写出程序的主调用类 ThreadCommunication。

程序清单：ThreadCommunication.java

```
class Producer implements Runnable
{
    Q q=null;
    public Producer(Q q)
    {
        this.q=q;
    }
    public void run()
    {
        int i=0;
        while(true)
        {
```

```

        if(i==0)
        {
            q.name="张孝祥";
            q.sex="男";
        }
        else
        {
            q.name="陈琼";
            q.sex="女";
        }
        i=(i+1)%2;
    }
}
class Q
{
    String name="陈琼";
    String sex="女";
}
class Consumer implements Runnable
{
    Q q=null;
    public Consumer(Q q)
    {
        this.q=q;
    }
    public void run()
    {
        while(true)
        {
            System.out.println(q.name + "---->" + q.sex);
        }
    }
}
public class ThreadCommunation
{
    public static void main(String [] args)
    {
        Q q=new Q();
        new Thread(new Producer(q)).start();
        new Thread(new Consumer(q)).start();
    }
}

```

在上面的代码中，Producer 和 Consumer 都定义了一个类 Q 的成员变量，并通过各自的构造函

数对其赋值。在主调用类中，定义了一个对象，并将其同时传给创建的 Producer 对象和 Consumer 对象，这样，Producer 和 Consumer 访问的就是同一个 Q 对象了。为了便于观察程序运行的效果，在 Producer 的 run 方法中，我们在每次循环中交替地存放两个人员数据内容：“张孝祥，男”，“陈琼，女”，为了让读者直接看到 CPU 在 Producer 线程只放了一部分数据就切换到 Consumer 线程的特殊情况（实际上就存在这样的可能性，只是我们不太容易捕捉到这种情况），我们在上面程序中的

```
q.name="张孝祥";  
q.sex="男";
```

两条语句之间加入暂停一段时间的程序代码，修改后的代码如下：

```
q.name="张孝祥";  
  
try  
{  
    Thread.sleep(10);  
}  
catch(Exception e)  
{  
    System.out.println(e.getMessage());  
}  
  
q.sex="男";
```

在以后的实际程序开发中不应有这一块。

编译并运行程序后的结果如下：

```
张孝祥---->女  
张孝祥---->女  
张孝祥---->女  
张孝祥---->女  
张孝祥---->女  
张孝祥---->女  
张孝祥---->女  
张孝祥---->女  
张孝祥---->女  
.....
```

Consumer 线程在屏幕上打出了“张孝祥”---->“女”这样的结果，显然又碰到了我们前面讲过的线程安全问题。如果大家真的掌握了前面讲过的线程同步的机理，应该能够自己解决这个问题了，不妨一试？我们需要将 Producer 和 Consumer 中的 run 方法中的有关代码块使用同一个对象的监视器进行同步，参考代码如下：

```
class Producer implements Runnable  
{  
    Q q=null;  
    public Producer(Q q)  
    {  
        this.q=q;  
    }  
    public void run()  
    {  
        int i=0;  
        while(true)
```

```

{
    synchronized(q)
    {
        if(i==0)
        {
            q.name="张孝祥";
            try
            {
                Thread.sleep(10);
            }
            catch(Exception e)
            {
                System.out.println(e.getMessage());
            }
            q.sex="男";
        }
        else
        {
            q.name="陈琼";
            q.sex="女";
        }
    }
    i=(i+1)%2;
}
}

class Q
{
    String name="陈琼";
    String sex="女";
}

class Consumer implements Runnable
{
    Q q=null;
    public Consumer(Q q)
    {
        this.q=q;
    }
    public void run()
    {
        while(true)
        {
            synchronized(q)
            {

```

```
        System.out.println(q.name + "---->" + q.sex);  
    }  
}  
}  
}  
}
```

对上面程序编译后运行，结果如下：

张孝祥---->男  
张孝祥---->男  
张孝祥---->男  
张孝祥---->男  
陈琼---->女  
陈琼---->女  
陈琼---->女  
陈琼---->女

程序中再没有打出“张孝祥”-->女这样的结果，说明我们解决了男女同步的问题。上面的程序虽然解决了线程同步的问题，但这样的程序结构比较混乱，显得条理不清，有点令人感到一种说不出来的别扭。我们为什么不在类 Q 中增加两个方法，一个对成员变量赋值，另一个取值，Producer 和 Consumer 分别调用这两个方法就行。如果要同步的两段代码或是两个函数放在同一个类中编写，是不是容易和清晰得多呢？我们再将代码修改成下面这样：

```
class Q
{
    private String name = "陈琼";
    private String sex = "女";
    public synchronized void put(String name, String sex)
    {
        this.name=name;
        try
        {
            Thread.sleep(10);
        }
        catch(Exception e)
        {
            System.out.println(e.getMessage());
        }
        this.sex=sex;
    }
    public synchronized void get()
    {
        System.out.println(name + "---->" + sex);
    }
}
class Producer implements Runnable
{
```

```

Q q=null;
public Producer(Q q)
{
    this.q=q;
}
public void run()
{
    int i=0;
    while(true)
    {
        if(i==0)
            q.put("张孝祥","男");
        else
            q.put("陈琼","女");
        i=(i+1)%2;
    }
}
class Consumer implements Runnable
{
    Q q=null;
    public Consumer(Q q)
    {
        this.q=q;
    }
    public void run()
    {
        while(true)
        {
            q.get();
        }
    }
}

```

上面修改过的程序结构是不是清楚多了？明显比先前的代码感觉令人舒畅！

现在读者应该明白这样的说法了：定义类时，尽量将其中的成员变量定义成 private 访问权限，对成员变量的访问都通过类中的具有 public 访问权限的方法来进行，这样定义的类才是面向对象的，类中的数据由类自身的方法来操作，保证了程序的高内聚性和健壮性。

从上面程序的执行结果上来看，Consumer 线程对 Producer 线程放入的一次数据连续读取了多次，并不符合我们的期望。我们要求的结果是，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，这就是我们要讲到的线程间的通信问题，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，由于所有的类都是从 Object 继承的，因此在任何类中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，直到其他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。用于类似饭馆有一个空位后通知所有等候就餐的顾客中的第一位可以入座的情况。

**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。用于类似某个不定期的培训班终于招生满额后，通知所有学员都来上课的情况。

如果想让上面的程序符合我们的需求，我们必须在类 `Q` 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 `Q` 类的程序代码如下：

```
class Q
{
    private String name="陈琼";
    private String sex="女";
    boolean bFull=false;
    public synchronized void put(String name,String sex)
    {
        if(bFull)
            wait();
        this.name=name;
        try
        {
            Thread.sleep(10);
        }
        catch(Exception e)
        {
            System.out.println(e.getMessage());
        }
        this.sex=sex;
        bFull=true;
        notify();
    }
    public synchronized void get()
    {
        if(!bFull)
            wait();
        System.out.println(name + "---->" + sex);
        bFull=false;
        Notify();
    }
}
```

编译并运行上面的程序，结果如下：

张孝祥---->男  
陈琼---->女  
张孝祥---->男

陈琼---->女  
张孝祥---->男  
陈琼---->女  
张孝祥---->男  
陈琼---->女  
张孝祥---->男  
陈琼---->女  
.....

上面的程序满足了我们的需求，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旗标，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，我们就可以分组有多个 `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 5.13 表示：

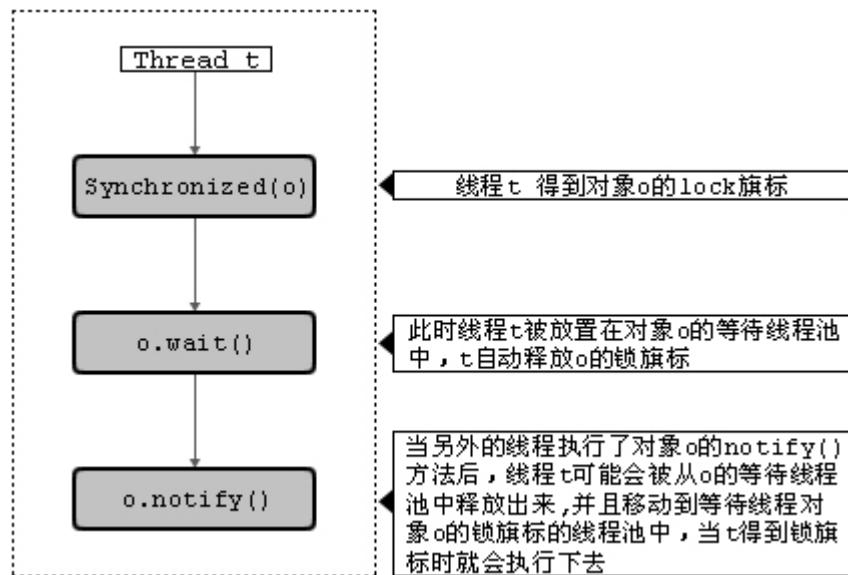


图 5.13

## 5.4 线程生命的控制

### 5.4.1 线程的生命周期

任何事物都有一个生命周期，线程也不例外，那么在一个程序中我们怎样控制一个线程的生命并让它更有效的为我们工作呢？要想控制它的生命，我们得先了解它的产生和消亡的整个过程。请读者结合我们前面讲的内容看看下面的这张图。

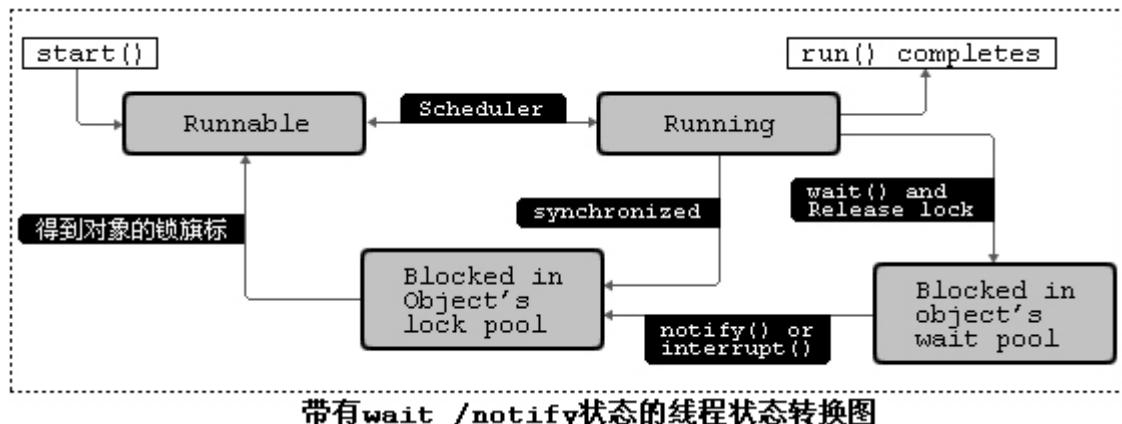


图 5.14

如图 5.14 所示，一个线程的产生是从我们调用了 start 方法开始进入 Runnable 状态，即可以被调度运行状态，并没有真正开始运行，调度器可以将 CPU 分配给它，真正运行其中的程序代码。线程在运行过程中，有以下几个可能的去向：

1. 没有遇到任何阻隔，运行完成直接结束，也就是 run() 方法执行完毕。
2. 调度器将 CPU 分配给其它线程，这个线程又变为 Runnable 状态。
3. 请求锁旗标，却得不到，这时候它要等待对象的锁旗标，得到锁旗标后又会进入 Runnable 状态开始运行。
4. 遇到 wait 方法，它会被放入等待池中继续等待，直到有 notify() 或 interrupt() 方法执行，它才会被唤醒或打断开始等待对象锁旗标，等到锁旗标后进入 Runnable 状态继续执行。

#### 5.4.2 如何控制线程的生命

了解了线程的生命周期，我们就不难理解如何控制线程生命的方法了吧？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。但我们不推荐使用这三个方法，其中，不推荐使用 suspend 和 resume 是因为：

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带来了另外的不足：

一个线程正在操作共享数据段，如果操作过程没有完成就 stop 了的话将会导致数据的完整性。

因此 stop 方法也被不提倡使用了！

既然这三个方法我们都不推荐使用，那么到底该用什么方法呢？请看下面的代码：

程序清单： ThreadLife.java

```

public class ThreadLife
{
    public static void main(String[] args)
    {
        ThreadTest t=new ThreadTest();
        new Thread(t).start();
        for(int i=0;i<100;i++)
        {
            if(i == 50)

```

```

        t.stopMe();
        System.out.println("mainThread is running");
    }
}
}

class ThreadTest implements Runnable
{
    private boolean bFlag = true;
    public void stopMe()
    {
        bFlag = false;
    }
    public void run()
    {
        while(bFlag)
        {
            System.out.println(Thread.currentThread().getName()+
                " is running ");
        }
    }
}

```

运行结果：

```

.....
Thread-1 is running
Thread-1 is running
Thread-1 is running
Thread-1 is running
mainThread is running
mainThread is running
mainThread is running
mainThread is running
.....

```

从上面的程序中我们定义了一个计数器 *i*，用来控制 *main* 线程的循环打印次数，在 *i* 的值从 0 到 50 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 50 的时候，程序调用了 *ThreadTest* 类的 *stopMe* 方法，而在 *stopMe* 方法中，我们将 *bFlag* 这个变量赋值为 *false*，也就是说终止了 *while* 循环，*run* 方法结束，*Thread-1* 线程随之结束。*main* 线程在计数器 *i* 等于 50 的时候，调用了 *ThreadTest* 类的 *stopMe* 方法后，CPU 不一定会马上会切换到 *Thread-1* 线程上，也就是说 *Thread-1* 线程不一定会马上终止，*main* 线程的计数器 *i* 可能到达五十几甚至六十几后，*Thread-1* 线程才真正结束。

综上所述，我们推荐使用控制 *run* 方法中循环的条件的方式来结束一个线程，这也是实际情况中用的最多的。

|                                 |     |
|---------------------------------|-----|
| 第 5 章 多线程 .....                 | 149 |
| 5.1 如何创建与理解线程 .....             | 149 |
| 5.1.1 了解线程概念 .....              | 149 |
| 5.1.2 用 Thread 类创建线程 .....      | 149 |
| 5.1.3 使用 Runnable 接口创建多线程 ..... | 152 |
| 5.1.4 两种实现多线程方式的对比分析 .....      | 153 |
| 5.1.5 后台线程与联合线程 .....           | 156 |
| 5.1.6 多线程在实际中的应用 .....          | 159 |
| 5.2 多线程的同步 .....                | 160 |
| 5.2.1 线程安全问题 .....              | 160 |
| 5.2.2 同步代码块 .....               | 162 |
| 5.2.3 同步函数 .....                | 164 |
| 5.2.4 代码块与函数间的同步 .....          | 165 |
| 多学两招：同步的注意事项                    |     |
| 5.2.5 死锁问题 .....                | 169 |
| 5.3 线程间的通信 .....                | 171 |
| 5.3.1 问题的引出 .....               | 171 |
| 5.3.2 问题如何解决 .....              | 171 |
| 5.4 线程生命的控制 .....               | 179 |
| 5.4.1 线程的生命周期 .....             | 179 |
| 5.4.2 如何控制线程的生命 .....           | 180 |

# 第6章 Java API

## 6.1 理解 API 的概念

### API 的概念

API (Application Programming Interface) 就是应用程序编程接口。

假设我们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，没有编过程序的人是很难想象出如何编写这样的程序的。但对于有经验的人来说，他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，我们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给我们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 操作系统提供给我们编写 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。我们在 Java 中，经常提到的 API，就是 JDK 中提供的各种功能的 Java 类。

### 学习编程语言与学汉语

作者认为，学习编程语言和学习汉语差不多，主要是从三个方面去掌握：

一、学好汉语，我们首先必须掌握汉语的语法。学好编程，我们也必须先学习该编程语言的语法，关于 Java 语言的语法，我们在本书前面的几个章节中所讲的知识点正是 Java 的语法。

二、学好汉语，我们还必须掌握大量的成语，虽然不要求掌握所有的成语，但至少也是成语掌握的越多，描述事务时用词也越发恰当、准确、快捷，文章写得也越发流畅，更能获得读者好评。同时，我们在学习成语的过程中，还能从成语本身上潜移默化地学到古人说明事物与问题时生动比喻的用词技巧，创造典故时的灵感与聪明才智。学习编程语言，掌握了大量的 API，就象我们学习汉语时掌握了大量的成语一样，我们在处理某些问题时将会轻而易举，同样，我们也能从这些 API 类中学会大师们组织 Java 类的方法，划分系统结构的技巧。掌握较多的实用类（即 API），就如掌握大量的成语一样有好处。

三、学好汉语，除了要掌握语法和成语外，还应该学会写文章的技巧和手法，找到写文章的灵感，才能写出好的作文。学习编程，也需要掌握分析与解决问题的手法，养成良好的编程习惯与风格，体会出一种编程的感觉。学习汉语，我们可以从大量优秀的唐诗宋词及现代优秀散文中去体会写文章的技巧，同时听取语文老师的讲解与分析。同样，学习编程，我们也要从阅读别人优秀的程序代码中去找感觉和经验，同时听取有经验的程序员的分析与讲解。注意，这里不是说编程老师，而是说有经验的程序员，因为，我们不得不承认的一个现实，一些大学老师从事实际项目开发的经验不太丰富，只能讲些语法和 API 方面的知识，没有能力帮你分析与讲解编程经验与体会的，就象我们许多小学老师自己都写不出好的作文来，但却可以成为语文老师一样的道理。

当然，除了上面所说的共同点之外，学习编程语言与学习汉语还是有些区别：

首先，汉语中的成语必须先记下来，才能在需要时用得起来，但对 API 来说，完全可以在需要时通过某种方式临时获取，现用现学。

另外，学习汉语，我们完全可以自学，自己多读优秀的文章，就可能写出好的文章来，事实上，很多作家都是这样自学成才的。但对编程来说，动手操作性很强，在某些方面又象学修车一样，闭门修炼还是比较难的，要是有机会经常看到老手的操作和听到他们的见解，经常与同行交流，这样的进步速度就非常快。

## 如何算学会

那么，对一门语言，学到什么程度就算掌握和学会了呢？一门编程语言可以用在生活中的方方面面，每个方面又都有自己的许多细节，所以，我们不可能了解一门语言中的各个方面和细节，也没有一个专家，敢说他能做到这一步的。其实我们也没有必要去了解一门语言中的各个方面和细节，虽然，我们了解的越多，我们的水平似乎就越高，但这都是要以时间和精力为代价的，学习到一定程度后，要适可而止，否则，一辈子都只有疲于学习的份儿了，就完全违背“学以致用”的初衷了。作者认为，当你掌握了一门语言的语法特点后，能够看懂一般的程序，和在需要时能够参照文档资料看懂以前还没接触过的某个方面的程序，能够自己写出一个有某种实际应用的小程序，你就算掌握了这门语言，剩下的就是你在工作中如何去积累经验的问题了。我刚才说的是能写一个程序就够了，如网络聊天，文件分割等，这在我们后面的课程中都会讲到的。会写一个程序的标准，不是象有的学员一样，完全把我讲的例子给背下来，这是不可取的。我要求的是你自己理解了我的思想后，再用自己的想法去独立写下来，对程序中的每个细节都是真正明白的。我们学习汉语，也是只要能够看懂一般性的文章，能写篇小文章，也就学会了吧，我不相信有人能够认识每一个汉字，知道每一个成语和典故，汉语博士在读到一篇文章时，也会有不认识的字，在必要时去查查字典就行了。专家不是学出来的，是有了一定的基础后，在工作中再总结、再学习的过程中成长起来的。

## 6.2 工具软件的介绍与使用

工欲善其事，必先利其器。在详细讲解这章之前，我们还需要在我们的计算机上安装一些新工具。

因为 Sun 的 JDK 是非常简洁的，没有提供图形化的开发环境，他把开发工具这一块肥肉让给了他的合作伙伴，之所以这样做，也有他的道理，除了让合作伙伴得到一些实惠之外，也激起了合作伙伴之间的激烈竞争，于是 Java 的开发工具也就越做越好，方便了我们这些 Java 程序员的同时，也为他们自己铺平了前进的道路。

于是，开发工具商、系统集成商们一拥而上。不过要论业绩呢，要数 Borland 公司的 JBuilder、IBM 的 Visual Age 和塞门铁克的 Visual Cafe 做的都很好，在业界颇有好评。但是，由于这些工具使用起来满复杂的，介绍这类工具的篇幅会很长，还请读者朋友去参考一些专门的书籍。作者在这里推荐的是一款简单而小巧的开发工具——JCreator Pro，它是一款共享软件，下载地址是：<http://www.JCreator.com>。

### JCreator

它的安装很简单，大体与其它的 windows 程序安装过程一样，我们就不在这里多说了。我们还是着重讲一下它的使用吧，下面是它在使用中的图示：

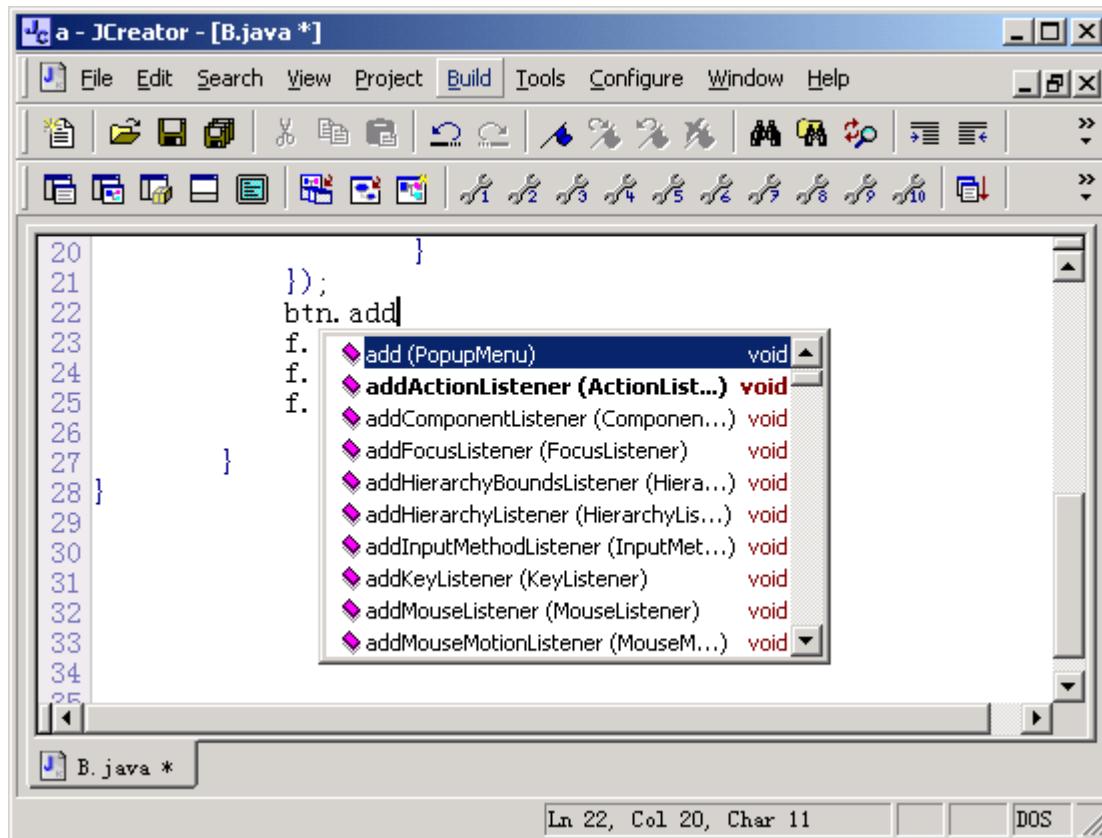


图 6.1

如果读者在以前接触或使用过 Borland 公司的 Jbuilder, 可能对这个软件就不会太陌生, 因为它的一些基本功能设计和 Jbuilder 差不多, 只是在支持 J2EE 上差了点儿, 对于 Java 的初学者来说是足够用了, 而且它形体小巧, 运行起来比较快。

和 EditPlus 之类的小工具相比起来呢, 它又具有动态随笔提示功能, 如上图, 在我们输入 “System.out.” 的时候, 它会出现关于该类中的方法和属性、方法的参数类型与返回值类型。这些功能对我们学习本章及本章以后的内容都很有帮助, 因为它能节省你查阅 JDK 文档的很多时间。

### 6.3 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, 我们不需要用 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、字符串与其他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, 可以将其他各种类型的数据增加、插入到字符串中, 也可以翻转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, 你就可以使用 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 (+), 可以把其他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();
```

## M 脚下留心：

```
String s1="hello";  
String s2="hello";
```

上面的两个等号“=”说明 s1 与 s2 是同一个对象，而下面这两句代码是创建两个对象，有了我们前面讲的知识，读者就不难理解：尽管它们内容相同，但却是两个不同的对象：

```
String s1=new String("hello");  
String s2=new String("hello");
```

## String 类的使用

我们还是编写一个程序来简单了解 String 类的使用，程序一行一行地读取从键盘上不停输入的字符串，并打印显示，直到输入一行“bye”为止。

程序清单：ReadLine.java

```
public class ReadLine  
{  
    public static void main(String [] args)  
    {  
        byte buf[] = new byte[1024];  
        String strInfo=null;  
        int pos=0;  
        int ch=0;  
        System.out.println("please enter info, input bye for exit:");  
        while(true)  
        {  
            try  
            {  
                ch=System.in.read();  
            }  
            catch(Exception e)  
            {  
                System.out.println(e.getMessage());  
            }  
            switch(ch)  
            {  
                case '\r':  
                    break;  
                case '\n':  
                    strInfo= new String(buf,0,pos);  
                    if(strInfo == "bye")  
                        break;  
            }  
        }  
    }  
}
```

```
        return ;
    else
        System.out.println(strInfo);
        pos=0;
        break;
    default:
        buf[pos++]=(byte)ch;
    }
}
```

上面程序中我们用到了 `String` 类的构造方法 `String(buf, 0, pos)`, 这个方法是把数组 `buf` 里面的值从 0 到 `pos` 取出, 用来创建一个新的 `String` 类对象。从这个构造方法中, 读者应该学会了如何将一个字节数组转换成字符串的办法。

无论我们怎样输入“bye”，程序就是无法退出，读者最好还是先自己找找原因。关于 equals 与 == 的区别，虽然我们在前面已经讲过，但对于初学者来说，就算你现在已经明白，日后在实际开发中，遇到要比较 String 类对象的内容是否等于某一字符串常量时，还是非常容易忘记使用 equals，而误用成 == 的。上面代码中的 if(strInfo== “bye” ) 改为 if(strInfo.equals( “bye” )), 程序就可以通过输入一行“bye”结束运行了。

String 类有几个比较常用的函数，如 equalsIgnoreCase、indexOf、substring，下面我们分别介绍一下：

`equalIgnoreCase (String anotherString)`是在比较两个字符串时忽略大小写，这在实际应用中用的非常多，比如我们上面输入和打印字符的例子，只能是输入小写的“bye”才能结束，其实我们还可以把它做的更友好一些，如输入“BYE”和“Bye”等都能正确退出。这时只需把程序中的`if(strInfo.equals("bye"))`改为`if (strInfo.equalsIgnoreCase("bye"))`就行了。

还有这样一种情况：我们在编写的程序中有一个用户登录界面，需要将用户的姓名与数据库里面的姓名进行比较，这时如果我们用以前的 equals 方式来比较，就可能出错，比如程序可能认为 Tom 和 tom 不是一个人，这样的程序就太不友好了。要想让程序友好一些，不出这样的错误，我们就要用方法 equalsIgnoreCase 来做姓名这个字段的验证。

`indexOf(int ch)`方法是用来返回一个字符在该字符串中的首次出现的位置，如果没有这个字符则返回“-1”。它的另一种形式`indexOf(int ch, int fromIndex)`返回的是从`fromIndex`指定的数值开始，`ch`字符首次出现的位置。该方法可以应用于文件和数据库内容的查找等功能，如Word等字处理软件等。

```
如: String str="hello world";  
System.out.println(str.indexOf('o'));
```

打印结果是 4

```
String str="hello world";
System.out.println(str.indexOf('l', 6));
```

打印结果页 3

`indexOf` 还有其他几种重载形式，如

indexOf(String str)

indexOf(String str, int fromIndex)

这两个函数返回一个子字符串在该字符串中的首次出现的位置。读者掌握了上面对首个

字符的进行操作的方法，自然也就明白了如何使用这两个函数。

`subString(int beginIndex)`方法返回的是在一个字符串中从 `beginIndex` 指定的数值到末尾的一个字符串，如果 `beginIndex` 指定的数值超过了当前字符串的长度，则返回一个空字符串。这个方法也有另外一种形式 `substring(int beginIndex, int endIndex)`，它返回的是当前字符串中从 `beginIndex` 开始到 `endIndex-1` 结束的一个字符串。如：

```
String str="hello world";
System.out.println(str.substring(6));
```

打印结果是 `world`。

```
String str="hello world";
System.out.println(str.substring(6,8));
```

打印结果是 `wo`。

关于这两个类的其它方法，读者要自己查阅 JDK 文档资料了。看完文档中所有 `String` 类的方法后，你也许会问：“`String` 类中的 `replace` 和 `toUpperCase` 方法不都能改变字符串的内容吗？这与你先讲的‘`String` 类对象的内容一旦被初始化就不能再改变’不是自相矛盾吗？”，请你再仔细看一下这两个函数的帮助，他们的返回类型都是 `String` 类，即生成一个新的字符串，而不是改变原来的字符串内容。

## 6.4 基本数据类型的对象包装类

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（也有的书中叫包裹类）。使用基本简单数据类型，可以改善系统的性能，也能够满足大多数的应用。但基本简单类型不具有对象的特性，不能满足某些特殊的需求。从 JDK 中，我们知道，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，我们又需要用这些方法来处理基本数据类型的数据，这时，我们就要用到包装类。比如，用 `Integer` 类来包装整数。关于这种应用，我们在后面讲解的 `Vector` 类的例子程序中就要碰到。

另外，包装类对象在进行基本数据类型的类型转换时也特别有用，如整数与字符串的转换。下面的程序用于在屏幕上打印出一个星号(\*)组成的矩形，矩形的宽度和高度通过运行时为程序传递的参数指定。

程序清单：`TestInteger.java`

```
class TestInteger
{
    public static void main(String[] args)
    {
        int w = Integer.parseInt(args[0]); //第一种方法
        int h = new Integer(args[1]).intValue(); //第二种方法
        //int h = Integer.valueOf(args[1]).intValue(); //第三种方法
        for(int i=0;i<h;i++)
        {
            StringBuffer sb=new StringBuffer();
            for(int j=0 ;j<w;j++)
            {
                sb.append('*');
            }
            System.out.println(sb);
        }
    }
}
```

```
        System.out.println(sb.toString());
    }
}
```

编译后运行“java TestInteger 20 10”，结果如下：

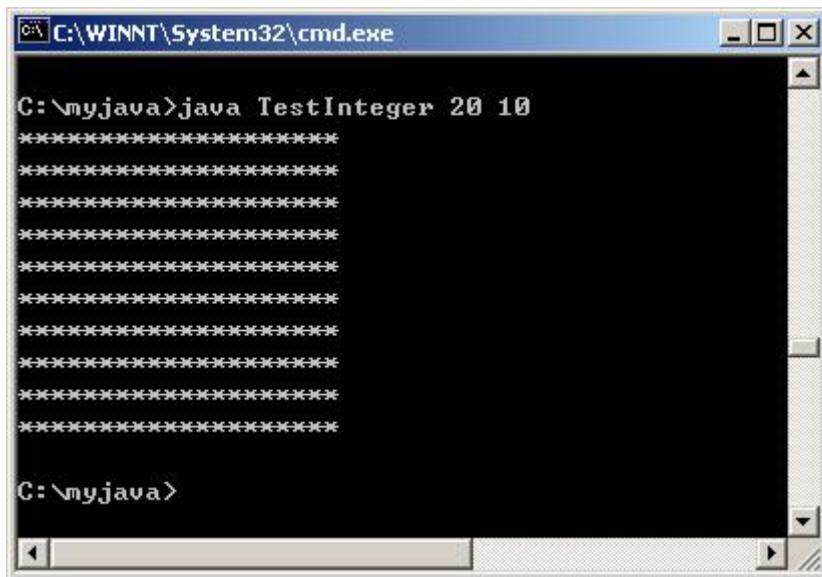


图 6.2

在上面的程序中，我们想打印出一个由\*号组成的矩形，这时需要指定矩形的长度和宽度两个值，我们可以在运行这个程序时，将矩形的长度和宽度作为程序参数传递进去。在程序中，可以直接从 args 数组中取出传递进的程序参数，但这些参数是字符串型的，不能直接使用，所以我们需要实现字符串到整数之间的转换。这时，我们就需要用到整数包装类 Integer，从 JDK 文档中，我们很容易发现有好几种方式都可以实现字符串与整数间的转换，上面的程序给出了作者从 JDK 文档中了解到的三种方法，供读者参考。注意，我们在打印的这个过程中用到了 StringBuffer 类的 append 方法，这也是复习我们前面讲的内容。

在 Java 中，八种基本数据类型都有其对应的包装类，下面是它们和其相应包装类的对照：

|         |           |
|---------|-----------|
| boolean | Boolean   |
| byte    | Byte      |
| char    | Character |
| short   | Short     |
| int     | Integer   |
| long    | Long      |
| float   | Float     |
| double  | Double    |

读者掌握了 Integer 这个类的用法，自然也就学会了其它几个包装类的用法，但读者要学会善于找出这些包装类的一些共性的地方，要达到举一反三，学一通百的效果。如：要将字符串转换成基本数据类型，几乎都是用“Xxx 包装类.parseXxx”方式实现（一个例外是对于 Boolean 类，用的是 getBoolean 方法）；要将包装类转换成基本数据，几乎都是“Xxx 包装类对象.xxxValue”方式。

## 6.5 集合类

在 Java 编程中，我们会经常用到 Vector, Enumeration, ArrayList, Collection, Iterator, Set, List 等集合类和接口。

### Vector 类与 Enumeration 接口

Vector 类是 Java 语言提供的一种高级数据结构，可用于保存一系列的对象，Java 不支持动态数组，Vector 类提供了一种与“动态数组”相近的功能。如果我们不能预先确定要保存的对象的数目，或是需要方便获得某个对象的存放位置时，Vector 类都是一种不错的选择。

如：我们在键盘上输入一个数字序列并存储在某种数据结构中，最后在屏幕上打印出每位数字相加的结果：

程序清单：TestVector.java

```
import java.util.*; //下面用到的 Vector 类和 Enumeration 接口都在此包中
public class TestVector
{
    public static void main(String [] args)
    {
        int b=0;
        Vector v=new Vector();
        System.out.println("Please Enter Number:");
        while(true)
        {
            try
            {
                b= System.in.read();
            }
            catch(Exception e)
            {
                System.out.println(e.getMessage());
            }
            if(b=='\r' || b=='\n')
                break;
            else
            {
                int num=b-'0';
                v.addElement(new Integer(num));
            }
        }
        int sum=0;
        Enumeration e=v.elements();
        while(e.hasMoreElements())
        {
            Integer intObj=(Integer)e.nextElement();
            sum += intObj.intValue();
        }
        System.out.println(sum);
    }
}
```

```
    }  
}
```

运行结果：

```
输入 32 打印 5  
输入 1234 打印 10
```

在上面的例子中，因为我们不能预先确定输入数字序列的位数，所以不能使用数组来存储每一个数值。正因为如此，我们选择了 `Vector` 类来保存我们的数据。`Vector.addElement` 只能接受对象类型的数据，我们先用 `Integer` 类包装了我们的整数后，再用 `Vector.addElement` 方法向 `Vector` 对象中加入这个整数对象。最后，我们要取出保存在 `Vector` 对象中的所有整数进行相加，首先必须通过 `Vector.elements` 方法返回一个 `Enumeration` 接口对象，再用 `Enumeration.nextElement` 方法逐一取出保存的每个整数对象，`Enumeration` 对象内部有一个指示器指向调用 `nextElement` 方法时要返回的对象的位置。

读者不要从字面上去理解 `nextElement` 方法，`nextElement` 不是返回下一个对象，而是返回指示器正指向的那个对象，并将指示器指向下一个对象。当指示器指向了一个空对象（表示没有对象可以返回）时，`Enumeration.hasMoreElement` 方法将返回 `false`，否则返回 `true`，调用 `nextElement` 方法之前，指示器指向第一个对象或空对象（`Enumeration` 中没有一个对象存在时）。`nextElement` 方法返回的是 `Object` 类型，我们需要对其进行类型转换，转换成 `Integer`。

`Enumeration` 是一个接口类，它提供了一种访问各种数据结构（`Vector` 类只是众多数据结构中的一种）中的所有数据的抽象机制，就是我们要访问各种数据结构对象中的所有元素时，都可以使用同样的方式，调用同样的方法。有了这样的数据结构接口，我们就很容易学一通百，以不变应万变。

### Collection 接口与 Iterator 接口

Java2 平台发布后，Java 设计者又推出了一套综合的数据结构，这些数据结构类的基本接口是 `Collection`，它的使用非常类似上面讲过的 `Vector` 类，只是方法的名称不同。我们要取出保存在实现了 `Collection` 接口对象中的所有对象，我们也必须通过 `Collection.iterator` 方法返回一个 `Iterator` 接口对象，`Iterator` 接口的功能与使用同 `Enumeration` 接口非常相似。Java2 平台的数据结构类设计人员本可以扩展 `Enumeration` 接口，而不用创建 `Iterator` 这个新接口。但他们不喜欢 `Enumeration` 接口方法冗长的名字，因而创建了 `Iterator` 这个新接口，并缩短了方法名长度。

按照 Java 的语法，我们不能直接用 `Collection` 接口类创建对象，而必须用实现了 `Collection` 接口的类来创建对象，`ArrayList` 类就是一个实现了 `Collection` 接口的类，我们将上面使用 `Vector` 和 `Enumeration` 的例子改为用 `ArrayList` 和 `Iterator` 编写，读者就应该基本明白了这些类之间的关系和用法。

程序清单：TestCollection.java

```
import java.util.*; //ArrayList 类和 Iterator 接口都在此包中  
public class TestCollection  
{  
    public static void main(String [] args)  
    {  
        int b=0;  
        ArrayList al=new ArrayList();  
        System.out.println("Please Enter Number:");
```

```

        while(true)
        {
            try
            {
                b= System.in.read();
            }
            catch(Exception e)
            {
                System.out.println(e.getMessage());
            }
            if(b=='\r' | b=='\n')
                break;
            else
            {
                int num=b-'0';
                al.add(new Integer(num));
            }
        }
        int sum=0;
        Iterator itr=al.iterator();
        while(itr.hasNext())
        {
            Integer intObj=(Integer)itr.next();
            sum += intObj.intValue();
        }
        System.out.println(sum);
    }
}

```

运行结果与前面的 TestVector.java 相同：

输入 32 打印 5

输入 1234 打印 10

关于这些类中的其他方法都无需解释，请参阅 JDK 文档资料中的 API 部分，我不想让读者变成只会照着书做、而不会自学和思考的人，更不想在这里浪费读者的时间。假设有个人真的把 SUN 公司发布的 Java API 全都背了下来，以后再遇到别的公司开发的 Java 类，不还是一样不会吗？

爱思考的读者也许要问：我什么时候用 Vector，什么时候用 ArrayList 呢？Vector 类中的所有方法都是线程同步的，两个线程并发访问 Vector 对象将是安全的，但只有一个线程访问 Vector 对象时，因为源程序仍调用了同步方法，需要额外的监视器检查，运行效率要低些。

ArrayList 类中的所有方法是非同步的，所以在没有多线程安全问题时，最好用 ArrayList，程序的效率会高些。在有线程安全问题，且我们的程序又没有自己处理（自己处理是指对调用 ArrayList 的代码或方法加上同步代码同步处理）时，只能用 Vector。

### 集合类接口的比较：

另外还有几个集合类接口，Set、List，下面是 Collection 和它们的比较：

**Collection** - 对象之间没有指定的顺序，允许重复元素

**Set** - 对象之间没有指定的顺序，不允许重复元素

**List** - 对象之间有指定的顺序，允许重复元素

这三个接口的继承关系图：

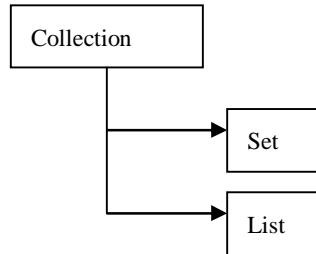


图 6.3

## & 多学两招：

由于在 **List** 接口中，对象之间有指定的顺序。因此我们可以对 **List** 接口的对象进行排序：

程序清单：TestSort.java

```
import java.util.*;
public class TestSort
{
    public static void main(String[] args)
    {
        ArrayList al=new ArrayList();
        al.add(new Integer(1));
        al.add(new Integer(3));
        al.add(new Integer(2));
        System.out.println(al.toString()); //排序前
        Collections.sort(al);
        System.out.println(al.toString()); //排序后
    }
}
```

运行结果：

```
[1, 3, 2]
[1, 2, 3]
```

在上面的程序中，我们向实现了 **List** 接口的 **ArrayList** 类对象中添加了 3 个成员。然后用 **Collections** 类的 **sort** 静态方法对其进行排序。

## 6.6 Hashtable 与 Properties 类

**Hashtable** 也是一种高级数据结构，用以快速检索数据。**Hashtable** 不仅可以像 **Vector** 一样动态存储一系列的对象，而且对存储的每一个对象（称为值）都要安排另一个对象（称

为关键字)与之相关联。例如，我们可以在 Hashtable 中存储若干国家的中文和英文名称，并且能够通过英文检索出对应的中文名称，这里中文就是值，英文就是关键字。

向 Hashtable 对象中存储数据，使用的是 Hashtable.put(Object key, Object value)方法，从 Hashtable 中检索数据，使用 Hashtable.get(Object key)方法。值和关键字都可以是任何类型的非空的对象。

下面的代码产生一个存储数字的 Hashtable，用英文数字作为关键字。

```
Hashtable numbers = new Hashtable();
numbers.put("one", new Integer(1));
numbers.put("two", new Integer(2));
numbers.put("three", new Integer(3));
```

要检索其中"two"关键字对应的数据，看下面的代码就能明白。

```
Integer n = (Integer)numbers.get("two");
if (n != null)
{
    System.out.println("two = " + n);
}
```

要想成功地从 Hashtable 中检索数据，用作关键字的对象必须正确覆盖了 Object.hashCode 方法和 Object.equals 方法。覆盖 Object.equals 理所当然，检索数据时，必须比较所用关键字是否与存储在 Hashtable 中的某个关键字相等，如果两个关键字对象不能正确判断是否相等，检索是不可能正确的。Object.hashCode 方法返回一个叫散列码的值，这个值是由对象的地址以某种方式转换来的。内容相同的两个对象，既然是两个对象，地址就不可能一样，所以 Object.hashCode 返回的值也不一样。要想两个内容相同的 Object 子类对象的 hashCode 方法返回一样的散列码，子类必须覆盖 Object.hashCode 方法，用于关键字的类，如果它的两个对象用 equals 方法是比较相等的，那么这两个对象的 hashCode 方法返回值也要一样，所以我们也要覆盖 hashCode 方法。因为 String 类已按关键字类的要求覆盖了这两个方法，如果两个 String 对象的内容不相等，它们的 hashCode 的返回值也不相等，如果两个 String 对象的内容相等，它们的 hashCode 的返回值也相等，所以，我们在实现自己编写的关键字类的 hashCode 方法时，可以调用这个关键字类的 String 类型的成员变量的 hashCode 方法来计算关键字类的 hashCode 返回值。注意：StringBuffer 类没有按照关键字类的要求覆盖 hashCode 方法，即使两个 StringBuffer 类对象的内容相等，但这两个对象的 hashCode 方法的返回值却不相等，所以，我们不能用 StringBuffer 作为关键字类。

下面是我们自己编写的一个类用于关键字时，是如何实现 equals 和 hashCode 这两个方法的。

```
class MyKey
{
    private String name;
    private int age;
    public MyKey(String name,int age)
    {
        this.name = name;
        this.age = age;
    }
    public String toString()
    {
```

```

        return new String(name + "," + age);
    }
    public boolean equals(Object obj)
    {
        if(name.equals(obj.name) && age==obj.age)
            return true;
        else
            return false;
    }
    public int hashCode()
    {
        return name.hashCode() + age;
    }
}

```

这个类实现了我们的目的：如果两个人名字和年龄都相同，我们就认为他们是同一个人。

下面我们讲解一下如何从一个 Hashtable 中，结合使用 MyKey 类作为关键字类，取出所有关键字的集合和取出所有值的集合：

程序清单：HashtableTest.java

```

import java.util.*;
public class HashtableTest
{
    public static void main(String[] args)
    {
        Hashtable numbers=new Hashtable();
        numbers.put(new MyKey("zhangsan", 18), new Integer(1));
        numbers.put(new MyKey("lisi", 15), new Integer(2));
        numbers.put(new MyKey("wangwu", 20), new Integer(3));
        Enumeration e=numbers.keys();
        while(e.hasMoreElements())
        {
            MyKey key=(MyKey)e.nextElement();
            System.out.print(key.toString()+"=");
            System.out.println(numbers.get(key).toString());
        }
    }
}

```

运行结果：

```

zhangsan, 18=2
lisi, 15=1
wangwu, 20=3

```

在上面这个程序中，我们定义了一个 Hashtable 的对象 numbers，然后给它放入了 3 个 MyKey 类对象和对应的阿拉伯数字。最后，我们在假装不知道该对象里面的值的情况下，利用 Enumeration 接口的对象 e 将值和关键字一一对应的取了出来。

Properties 是 Hashtable 的子类，它增加了将 Hashtable 对象中的关键字、值对保存到

文件和从文件中读取关键字/值对到 `Hashtable` 对象中的方法。大家可以想想在什么情况下可能会使用 `Properties` 类呢？大多数应用程序都有选项设置，就是在程序退出时将功能/设置值对存储到了文件，程序启动时将功能/设置值读到了内存，程序按新的设置运行，比如我们常用的 Microsoft Word 选项：

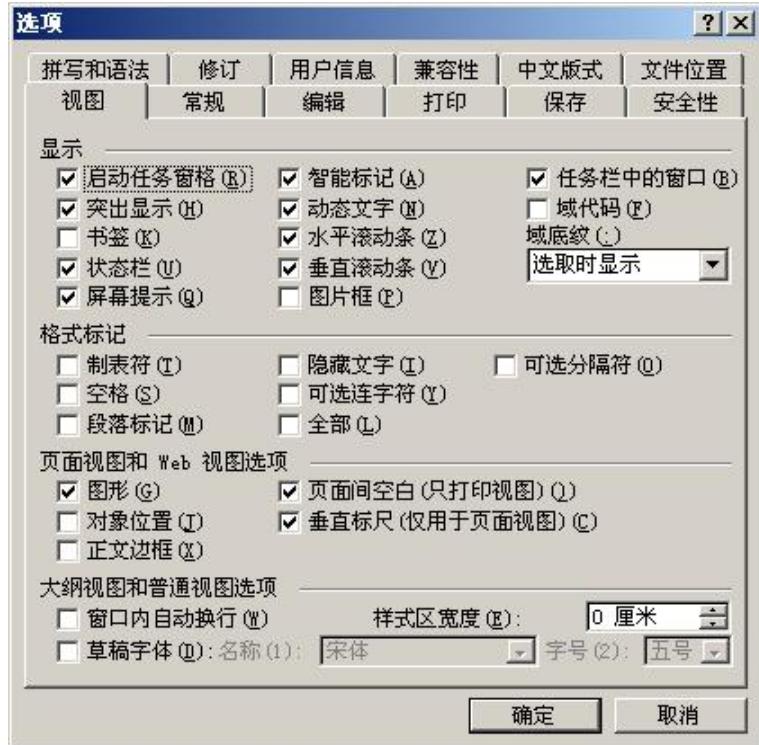


图 6.4

这不正好是 `Properties` 的用武之地吗？为了说明在 Java 中我们是如何使用 `Properties` 实现这一功能的，我们编写了下面的一个小程序，这个程序是把它的运行次数记录在某个文件中，每次运行时打印出它运行的次数。

程序清单：PropertiesFile.java

```
import java.util.*;
import java.io.*;
class PropertiesFile
{
    public static void main(String[] args)
    {
        Properties settings=new Properties();
        try
        {
            settings.load(new FileInputStream("c:\\\\count.txt"));
        }
        catch(Exception e)
        {
            settings.setProperty("Count",new Integer(0).toString());
        }
        int c=Integer.parseInt(settings.getProperty("Count"))+1;
        System.out.println("这是本程序第"+c+"次被使用");
    }
}
```

```

        settings.put("Count", new Integer(c).toString());
    try
    {
        settings.store(new FileOutputStream("c:\\count.txt"),
        "This Program is used:");
    }
    catch(Exception e)
    {
        System.out.println(e.getMessage());
    }
}
}

```

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，我们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。如果我们要用到 `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。一般有使用次数限制的共享软件的程序代码基本上都是这么做的，只不过它们把记录次数的这个文件隐藏在某个不容易发现的地方，例如保存在注册表里或某个系统文件里等等，我们只要找到这个地方，删除这个文件或注册表项，这个软件又可以被我们使用了。

## 6.7 System 类与 Runtime 类

### 6.7.1 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，当我们要引用这些变量和方法时，直接使用 `System` 类名作前缀，我们前面已经使用到了标准输入和输出的 `in` 和 `out` 变量，关于这两个变量的详细介绍，请看第七章 `I/O`。下面再介绍 `System` 类中几个方法，其他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。对于在用户正常操作下，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他各种类型，也就是说，我们平常用到的日期本质上就是一个数值，但通过这个数值，我们能够推算出其对应的具体日期时间。

#### & 多学两招：

我们还可用 `currentTimeMillis` 方法检测一段程序代码运行时所花费的时间：

```

Long startTime=System.currentTimeMillis();
.....
//代码段

```

```

Long endTime= System.currentTimeMillis();
System.out.println("total time expended is " + (starttime - endTime) +
                    "milliseconds");

```

### getProperties 方法与 Java 的环境属性

getProperties 方法获得当前虚拟机的环境属性。如果大家明白 Windows 的环境属性，如我们在第一章中讲到的 path 和 classpath 就是其中的两个环境变量，每一个属性都是变量与值成对的形式出现的。

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。

程序清单：TestProperties

```

import java.util.*;
public class SystemInfo
{
    public static void main(String[] args)
    {
        Properties sp=System.getProperties();
        Enumeration e=sp.propertyNames();
        while(e.hasMoreElements())
        {
            String key=(String)e.nextElement();
            System.out.println(key+" = "+sp.getProperty(key));
        }
    }
}

```

运行结果：

```

C:\Program Files\Xinox Software\JCreator Pro\GE2001.exe
java.runtime.name = Java(TM) 2 Runtime Environment, Standard Edition
sun.boot.library.path = C:\j2sdk1.4.0\jre\bin
java.vm.version = 1.4.0-b92
java.vm.vendor = Sun Microsystems Inc.
java.vendor.url = http://java.sun.com/
path.separator =
java.vm.name = Java HotSpot(TM) Client VM
file.encoding.pkg = sun.io
user.country = CN
sun.os.patch.level = Service Pack 2
java.vm.specification.name = Java Virtual Machine Specification
user.dir = e:\MyProjects\six\classes
java.runtime.version = 1.4.0-b92
java.awt.graphicsenv = sun.awt.Win32GraphicsEnvironment
java.endorsed.dirs = C:\j2sdk1.4.0\jre\lib\endorsed
os.arch = x86
java.io.tmpdir = C:\WINNT\Temp\
line.separator =
java.vm.specification.vendor = Sun Microsystems Inc.
user.variant =
os.name = Windows 2000
sun.java2d.fontpath =
java.library.path = C:\j2sdk1.4.0\bin;.;C:\WINNT\System32;C:\WINNT;C:\j2s

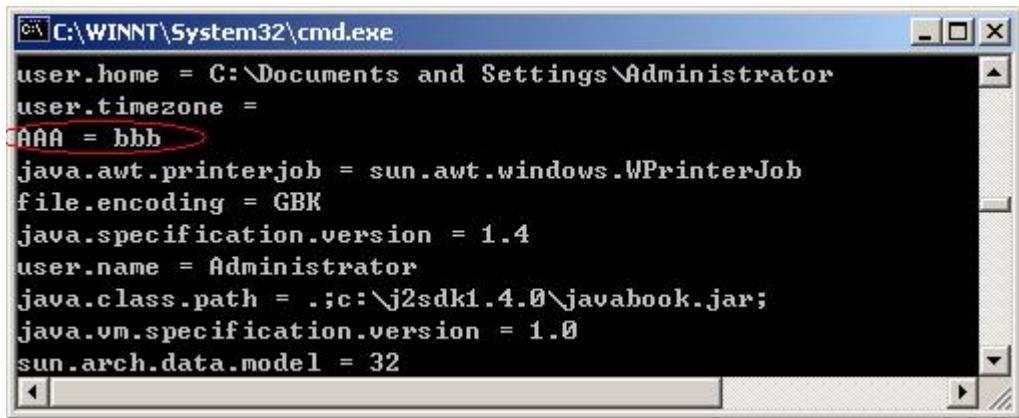
```

图 6.5

在 Windows 中，我们很容易增加一个新的环境属性，但我们如何为 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，我们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。我们按下面的格式运行：

```
java -DAAA=bbb TestProperties
```

运行后如下图：



```
C:\WINNT\System32\cmd.exe
user.home = C:\Documents and Settings\Administrator
user.timezone =
AAA = bbb
java.awt.printerjob = sun.awt.windows.WPrinterJob
file.encoding = GBK
java.specification.version = 1.4
user.name = Administrator
java.class.path = .;c:\j2sdk1.4.0\javabook.jar;
java.vm.specification.version = 1.0
sun.arch.data.model = 32
```

图 6.6

我们看到输出的结果中多了一行“AAA = bbb”，即 java 虚拟机中多了一个新的环境属性 AAA。要增加两个环境属性的格式例子是：

```
java -DAAA=bbb -DCCC=ddd TestProperties
```

**注意：**-D 与 AAA 之间没有空格。

讲解了 getProperties 方法，读者自己应该能够明白 setProperties 方法了。

### 6.7.2 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 中的方法相重复。我们不能直接创建 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。

Exec 方法，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，即执行其他程序时使用 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。

程序清单：TestRuntime.java

```
public class TestRuntime
{
    public static void main(String[] args)
    {
        Process p=null;
        try
        {
            p=Runtime.getRuntime().exec("notepad.exe TestRuntime.java");
            Thread.sleep(5000);
        }
        catch(Exception e)
```

```

    {
        System.out.println(e.getMessage());
    }
    p.destroy();
}
}

```

运行后程序启动一个子进程：用 Windows 的记事本程序打开了我们的源程序，并在 5 秒钟后销毁该子进程，记事本程序被关掉。

由于程序不能直接创建类 `Runtime` 的实例，所以可以保证我们只会产生一个 `Runtime` 的实例对象，而不能产生多个实例对象，这种情况就是我们前面曾经讲过的单态设计模式。读者可以按照前面讲过的单态设计模式思想，来设想一下 `Runtime` 类在内部是如何构造 `Runtime` 类的对象实例的。

## 6.8 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 `Data()`，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`。

`Calendar` 类是一个抽象基类，主要用于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 目前提供的一个唯一的 `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 315 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。

```

import java.util.*;
class TestCalendar
{
    public static void main(String[] args)
    {
        Calendar c1=Calendar.getInstance();
        //下面打印当前时间
        System.out.println(c1.get(c1.YEAR)+"年"+(c1.get(c1.MONTH)+1)+"
        "月"+c1.get(c1.DAY_OF_MONTH)+"日 "+c1.get(c1.HOUR)+"
        ":""+c1.get(c1.MINUTE)+":"+"+c1.get(c1.SECOND));
        c1.add(c1.DAY_OF_YEAR,315); //增加天数为 315
        //下面打印的是 315 天后的时间
        System.out.println(c1.get(c1.YEAR)+"年"+(c1.get(c1.MONTH)+1)+"
        "月"+c1.get(c1.DAY_OF_MONTH)+"日 "+c1.get(c1.HOUR)+"
        ":""+c1.get(c1.MINUTE)+":"+"+c1.get(c1.SECOND));
    }
}

```

}

我们也可以不通过 `Calendar.getInstance` 方法返回 `GregorianCalendar` 类对象，而直接使用 `GregorianCalendar` 类，构造一个 `GregorianCalendar` 类对象来完成上面的任务。

虽然 `Calendar` 类与上面的 `Runtime` 类都不能用 `new` 操作符直接在类外部产生对象实例，都是用类的静态方法返回一个对象实例，但两者的实现机理完全是不一样的。下面的例子演示了 `Calendar.getInstance()` 方法的实现过程。

```
public static Calendar getInstance()
{
    return new GregorianCalendar(TimeZone.getDefault(),
                                  Locale.getDefault());
}
```

虽然 `Calendar` 类几乎完全替代了 `Date` 类，但在某些情况下，我们仍有可能要用到 `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。我们有时要将用 `Date` 对象表示的日期用指定的格式输出和将特定格式的日期字符串转换成一个 `Date` 对象。

`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 `DateFormat` 子类，它是一个具体类，使用它就可以完成把 `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。

下面的例子将“2002-03-15”格式的日期字符串转换成“2002 年 03 月 15 日”。

程序清单：`TestDateFormat.java`

```
import java.util.*;
import java.text.*;
public class TestDateFormat
{
    public static void main(String[] args)
    {
        SimpleDateFormat sdf1=new SimpleDateFormat("yyyy-MM-dd");
        SimpleDateFormat sdf2=new SimpleDateFormat("yyyy 年 MM 月 dd 日");
        try
        {
            Date d=sdf1.parse("2003-03-15");
            System.out.println(sdf2.format(d));
        }
        catch(Exception e)
        {
            System.out.println(e.getMessage());
        }
    }
}
```

运行结果：

2003 年 03 月 15 日

`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，我们定义了一个 `SimpleDateFormat` 类的对象 `sdf1` 来接收和转换源格式字符串“2003-03-15”，随后又定义了该类的另一个对象 `sdf2` 来接收 `sdf1` 转换成的 `Date` 类

的对象，并按 sdf2 所定义的格式转换成字符串。

在这个过程中，我们已经实现了利用 SimpleDateFormat 类来把一个字符串转换成 Date 类对象，和把一个 Date 对象按我们指定的格式输出的两个功能。

注意：当我们要将一个字符串转换成 Date 类对象时，字符串中的代表年份的字符必须在模版中 yyyy 的位置，y 的个数指定了年的位数，在字符串中除了这些格式字符要被真实的日期数字替代之外，其它的字符必须原样出现，如：表示时间的字符串中的“-”的位置必须和创建 SimpleDateFormat 类对象时“-”的位置一样，同理，我们将日期转换成的字符串格式也是按照创建 SimpleDateFormat 类对象时指定的字符串格式而输出的。

## 6.9 Math 与 Random 类

Math 类包含了所有用于几何和三角几何的浮点运算函数，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 类是一个伪随机数产生器，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，在程序中我们看到的“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。

## 6.10 学习 API 的方法

本章讲解的是我们在编程中最常用的类，除此之外，JDK 中还有许多类，系统软件商、开发工具商也都会提供许多各种功能的类，大家不可能全部都去学习一遍的，而且也没有这个必要，到需要时再去掌握是完全来得及的。Java 的 API 非常多，必须规划好一个学习路线，才不会在浩瀚的 API 大海中迷失。有了某一领域的知识，再参看一些范例，很容易就掌握到一些新的 API。掌握了本章所讲的 API 和查阅文档资料的技巧，你就没必要再去看什么 Java API 大全之类的书籍了，那些大全无非是 JDK 文档的一些翻版罢了。

最聪明的人是最会利用工具和资源的人，要想做一个出色的程序员，读者必须学会查阅文档，同时也要结交一些程序员朋友，或上一些技术论坛，这些都是你解决问题的捷径。作者现在是 IT 人资讯交流网 ([www.it315.org](http://www.it315.org)) 的顾问，该网站就是针对一些问题，而为广大技术爱好者，从业人员提供一个交流技术的平台。该网站有针对初学者在学习 Java 过程中常常遇到的一些问题，进行详细的解答。当然，我们回答问题的侧重点将围绕本书进行。因为本书在编写过程中，已经包含了很多初学者常犯的错误，所以，在你提出你的问题之前，强烈建议你先读一下本书。

本书的重点在于讲解事情的来龙去脉、分析事物的本质和提供给读者解决问题的方法，而不是一些文档资料的翻录，我们把这个工作交给读者自己去完成，以便读者在以后真正的项目开发中，能够脱离开本书，自己去查阅官方的文档来解决问题。

一般的书籍，无外乎对 JDK 文档进行了摘录和翻译，远不及原始文档资料完整和准确，这样的书籍，根本不说明信息的来源，往往把初学者引入一条死记硬背的道路上去，往往会造成读者在碰到问题时，只知道去翻阅大量书籍，却根本解决不了问题，也就无法应对真实的开发过程了。大家根据自己的实际情况，可以提前通读一下 JDK 文档中大部分类及类中

的方法，做到遇到问题时心中有数，也可以暂时不读，只掌握原理，处理过程，解决方法，等到以后有具体的实际需求时，再来查阅 JDK 文档。

|   |     |
|---|-----|
| 第 6 章 Java API .....                    | 182 |
| 6.1 理解 API 的概念 .....                    | 182 |
| 6.2 工具软件的介绍与使用 .....                    | 183 |
| 6.3 String 类和 StringBuffer 类 .....      | 184 |
| 脚下留心：内容相同，对象不同                          |     |
| 6.4 基本数据类型的对象包装类 .....                  | 187 |
| 6.5 集合类 .....                           | 188 |
| 多学两招：对 List 接口的对象进行排序                   |     |
| 6.6 Hashtable 与 Properties 类 .....      | 192 |
| 6.7 System 类与 Runtime 类 .....           | 196 |
| 6.7.1 System 类 .....                    | 196 |
| 多学两招：检测程序段运行的时间                         |     |
| 6.7.2 Runtime 类 .....                   | 198 |
| 6.8 Date 与 Calendar, DateFormat 类 ..... | 199 |
| 6.9 Math 与 Random 类 .....               | 201 |
| 6.10 学习 API 的方法 .....                   | 201 |
|   | 202 |

# 第 7 章 IO/输入输出

大多数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，键盘被当作输入文件，显示器被当作输出文件使用。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。

## 7.1 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，我们能够创建、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。

在 Java 中，目录也被当作 `File` 使用，只是多了一些目录特有的功能——可以用 `list` 方法列出目录中的文件名。在 Unix 下的路径分隔符为 `(/)`，在 Dos 下的路径分隔符为 `(\)`，Java 可以正确处理 Unix 和 Dos 的路径分隔符，即使我们在 Windows 环境下使用 `(/)` 作为路径分隔符，Java 仍然能够正确处理。

我们用下面的一个简单应用来演示一下 `File` 类用法，判断某个文件是否存在，存在则删除，不存在则创建，读者可以在 Windows 的资源管理器下观察到这个变化。

程序清单： `FileTest.java`

```
import java.io.*;
public class FileTest
{
    public static void main(String[] args)
    {
        File f=new File("c:\\\\1.txt");
        if(f.exists())
            f.delete();
        else
            try
            {
                f.createNewFile();
            }
            catch(Exception e)
            {
                System.out.println(e.getMessage());
            }
        System.out.println("File name:"+f.getName());
        System.out.println("File path:"+f.getPath());
        System.out.println("Abs path:"+f.getAbsoluteFilePath());
        System.out.println("Parent:"+f.getParent());
        System.out.println(f.exists()?"exists":"does not exist");
        System.out.println(f.canWrite()?"is writeable":"not writeable");
    }
}
```

```

        is not writeable");
        System.out.println(f.canRead()?"is readable":"is not readable");
        System.out.println(f.isDirectory()?"is ":"is not"+" a directory");
        System.out.println(f.isFile()?"is normal file":"might be a named
pipe");
        System.out.println(f.isAbsolute()?"is absolute":"
is not absolute");
        System.out.println("File last modified:"+f.lastModified());
        System.out.println("File size:"+f.length()+" Bytes");
    }
}

```

当运行这个程序时会因为文件 1.txt 的存在和不存在而出现两种结果：

结果 1:

```

File name:1.txt
File path:c:\1.txt
Abs path:c:\1.txt
Parent:c:\\
exists
is writeable
is readable
is not a directory
is normal file
is absolute
File last modified:1051755103126
File size:0 Bytes

```

结果 2:

```

File name:1.txt
File path:c:\1.txt
Abs path:c:\1.txt
Parent:c:\\
does not exist
is not writeable
is not readable
is not a directory
might be a named pipe
is absolute
File last modified:0
File size:0 Bytes

```

注: delete 方法删除由 File 对象的路径所表示的磁盘文件。它只能删除普通文件，而不能删除目录，即使是空目录也不行。

关于 File 类的其它方法，是没法死记硬背的，读者在需要时自己查看 JDK 文档，应该能够明白怎么使用。初步接触了 File 类，我们发现 File 类不能访问文件的内容，即不能够从文件中读取数据或往文件里写数据，它只能对文件本身的属性进行操作。

## 7.2 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，我们可以跳转到文件的任意位置处读写数据。在你访问一个文件的时候，不想把文件从头读到尾，并希望像访问一个数据库一样的访问一个文本文件，使用 RandomAccessFile 类就是你的最佳选择。

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，我们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，不能访问其他的 I/O 设备，如网络，内存映象等。

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我们创建 RandomAccessFile 类对象的构造方式：

```
new RandomAccessFile(f, "rw"); //读写方式  
new RandomAccessFile(f, "r"); //只读方式
```

注：当我们的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，程序会为你创建它。

我们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，我们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，我们才能够准确定位每条记录在文件中的具体位置。假设 name 中有八个字符，少于八个则补空格（这里我们用"\u0000"），多于八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。

程序清单：RandomFileTest.java

```
import java.io.*;  
public class RandomFileTest  
{  
    public static void main(String [] args) throws Exception  
    {  
        Employee e1 = new Employee("zhangsan", 23);  
        Employee e2 = new Employee("Lisi", 24);  
        Employee e3 = new Employee("Wangwu", 25);  
        RandomAccessFile ra=new RandomAccessFile("c:\\\\1.txt", "rw");  
        ra.write(e1.name.getBytes());  
        ra.writeInt(e1.age);  
        ra.write(e2.name.getBytes());  
        ra.writeInt(e2.age);  
        ra.write(e3.name.getBytes());  
        ra.writeInt(e3.age);  
        ra.close();  
        RandomAccessFile raf=new RandomAccessFile("c:\\\\1.txt", "r");  
        int len=8;
```

```

raf.skipBytes(12); //跳过第一个员工的信息，其中姓名 8 字节
System.out.println("第二个员工信息: ");
String str="";
for(int i=0;i<len;i++)
    str=str+(char)raf.readByte();
System.out.println("name:"+str);
System.out.println("age:"+raf.readInt());

System.out.println("第一个员工的信息: ");
raf.seek(0); //将文件指针移动到文件开始位置
str="";
for(int i=0;i<len;i++)
    str=str+(char)raf.readByte();
System.out.println("name:"+str);
System.out.println("age:"+raf.readInt());

System.out.println("第三个员工的信息: ");
raf.skipBytes(12); //跳过第二个员工信息
str="";
for(int i=0;i<len;i++)
    str=str+(char)raf.readByte();
System.out.println("name:"+str.trim());
System.out.println("age:"+raf.readInt());

raf.close();
}
}

class Employee
{
    String name;
    int age;
    final static int LEN=8;
    public Employee(String name,int age)
    {
        if(name.length()>LEN)
        {
            name = name.substring(0,8);
        }
        else
        {
            while(name.length()<LEN)
                name=name+"\u0000";
        }
        this.name=name;
    }
}

```

```
        this.age=age;
    }
}
```

运行结果：

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
```

c 盘还多了个文件 1.txt：



图 7.1

上面的这个程序完成了我们想要的功能，演示了 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，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。在实际生活中，我们常用的数据库和数据库管理工具实际上就是这种原理。我们的 1.txt 就相当于数据库的数据文件，而我们这个程序提供了往这个数据文件写入和读取数据的功能。

## 7.3 节点流

### 7.3.1 理解流的概念

数据流是一串连续不断的数据的集合，就象水管里的水流，在水管的一端一点一点地供水，而在水管的另一端看到的是一股连续不断的水流。数据写入程序可以是一段、一段地向数据流管道中写入数据，这些数据段会按先后顺序形成一个长的数据流。对数据读取程序来说，看不到数据流在写入时的分段情况，每次可以读取其中的任意长度的数据，但只能先读取前面的数据后，再读取后面的数据。不管写入时是将数据分多次写入，还是作为一个整体一次写入，读取时的效果都是完全一样的。

我们将 IO 流类分为两个大类，节点流类和过滤流类（也叫处理流类）。程序用于直接操作目标设备所对应的类叫节点流类，程序也可以通过一个间接流类去调用节点流类，以达

到更加灵活方便地读写各种类型的数据，这个间接流类就是过滤流类（也叫处理流类），我更喜欢称之为包装类。不管叫什么，都只是一个代名词而已，读者不要太在意，你可以根据自己的习惯和喜好来定。

### 7.3.2 InputStream 与 OutputStream

程序可以从中连续读取字节的对象叫输入流，用 `InputStream` 类完成，程序能向其中连续写入字节的对象叫输出流，用 `OutputStream` 类完成。`InputStream` 与 `OutputStream` 对象是两个抽象类，还不能表明具体对应哪种 I/O 设备。它们下面有许多子类，包括网络，管道，内存，文件等具体的 I/O 设备，如 `FileInputStream` 类对应的就是文件输入流，是一个节点流类，我们将这些节点流类所对应的 I/O 源和目标称为流节点(Node)。

## F

### 指点迷津：

很多人搞不清程序要将 A 文件的内容写入 B 文件中，程序对 A 文件的操作所用的是输出类还是输入类这个问题。读者也先自己想想，再记住下面的话，输入输出类是相对程序而言的，而不是代表文件的，所以我们应该创建一个输入类来完成对 A 文件的操作，创建一个输出类来完成对 B 文件的操作。

`InputStream` 定义了 Java 的输入流模型。该类中的所有方法在遇到错误的时候都会引发 `IOException` 异常，下面是 `InputStream` 类中方法的一个简要说明：

- ü `int read()` 返回下一个输入字节的整型表示，如果返回 -1 表示遇到流的末尾，结束。
- ü `int read(byte[] b)` 读入 `b.length` 个字节放到 `b` 中并返回实际读入的字节数。
- ü `int read(byte[] b, int off, int len)` 这个方法表示把流中的数据读到，数组 `b` 中，第 `off` 个开始的 `len` 个数组元素中。
- ü `long skip(long n)` 跳过输入流上的 `n` 个字节并返回实际跳过的字节数。
- ü `int available()` 返回当前输入流中可读的字节数。
- ü `void mark(int readlimit)` 在输入流的当前位置处放上一个标志，允许最多再读入 `readlimit` 个字节。
- ü `void reset()` 把输入指针返回到以前所做的标志处。
- ü `boolean markSupported()` 如果当前流支持 `mark/reset` 操作就返回 `true`。
- ü `void close()` 在操作完一个流后要使用此方法将其关闭，系统就会释放与这个流相关的资源。

`InputStream` 是一个抽象类，程序中实际使用的是它的各种子类对象。不是所有的子类都会支持 `InputStream` 中定义的某些方法的，如 `skip`, `mark`, `reset` 等，这些方法只对某些子类有用。

## F

### 指点迷津：

一个对象在没有引用变量指向它时会变成垃圾，最终会被垃圾回收器从内存中清除。对于我们创建的流对象，干嘛还要“调用 `close` 方法将它关闭，以释放与其相关的资源”呢？这相关的资源到底是些什么呢？我们在程序中创建的对象都是对应现实世界中有形或无形的事物，计算机操作系统所产生的东西当然也是现实世界中的事物，也就是说，程序中的对象也可以对应计算机操作系统所产生的一个其他东西，专业地说，这些东西叫资源，流就是

操作系统产生的一种资源。当我们在程序中创建了一个 I/O 流对象，同时系统内也会创建了一个叫流的东西，在这种情况下，计算机内存中实际上产生了两个事物，一个是 Java 程序中的类的实例对象，一个是系统本身产生的某种资源，我们以后讲到的窗口，Socket 等都是这样的情况。Java 垃圾回收器只能管理程序中的类的实例对象，没法去管理系统产生的资源，所以程序需要调用 close 方法，去通知系统释放其自身产生的资源。

OutputStream 是一个定义了输出流的抽象类，这个类中的所有方法均返回 void，并在遇到错误时引发 IOException 异常。下面是 OutputStream 的方法：

- ü void write(int b) 将一个字节写到输出流。注意，这里的参数是 int 型，它允许 write 使用表达式而不用强制转换成 byte 型。
- ü void write(byte[] b) 将整个字节数组写到输出流中。
- ü void write(byte [] b, int off, int len) 将字节数组 b 中的从 off 开始的 len 个字节写到输出流。
- ü void flush 彻底完成输出并清空缓冲区。
- ü void close 关闭输出流。

## & 多学两招：

计算机访问外部设备，要比直接访问内存慢得多，如果我们每一次 write 方法的调用都直接写到外部设备（如直接写入硬盘文件），CPU 就要花费更多的时间等待外部设备；如果我们开辟一个内存缓冲区，程序的每一次 write 方法都是写到这个内存缓冲区中，只有这个缓冲区被装满后，系统才将这个缓冲区的内容一次集中写到外部设备。使用内存缓冲区有两个方面的好处，一是有效地提高了 CPU 的使用率，二是 write 并没有马上真正写入到外设，我们还有机会回滚部分写入的数据。使用缓冲区，能提高整个计算机系统的效率，但也会降低单个程序自身的效率，由于有这么一个中间缓冲区，数据并没有马上写入到目标中去，例如在网络流中，就会造成一些滞后。对于输入流，我们也可以使用缓冲区技术。在程序与外部设备之间到底用不用缓冲区，是由编程语言本身决定的，我们通常用的 C 语言默认情况下就会使用缓冲区，而在 Java 语言中，有的类使用了缓冲区，有的类没有使用缓冲区，我们还可以在程序中使用专门的包装类来实现自己的缓冲区。

flush 方法就是用于即使在缓冲区没有满的情况下，也将缓冲区的内容强制写入到外设，习惯上称这个过程为刷新。可见，flush 方法不是对所有的 OutputStream 子类都起作用的，它只对那些使用缓冲区的 OutputStream 子类有效。如果我们调用了 close 方法，系统在关闭这个流之前，也会将缓冲区的内容刷新到硬盘文件的。

作者开发过一个邮件服务器程序，需要 7\*24 小时不间断工作，这个服务器程序要面对 Internet 上各种可能的非法格式的数据输入和攻击，而我的程序正好又没考虑到某种非法格式的数据，一旦碰到这样的情况，程序就会崩溃。有经验的人都知道，为了找出服务器程序崩溃的原因，我们可以将程序每次接收到的数据都记录到一个文件中，当服务器程序崩溃后，我们便打开这个记录文件，查看最后记录的那条数据，这个数据就是让我的程序毙命的罪魁祸首，然后拿着这条数据一步步测试我们的程序，就很容易找出程序中的问题了。遗憾的是，我每次用最后记录的这条数据测试我的程序，程序均安然无恙。最后，我发现就是因为有缓冲区的原因，缓冲区的内容还没来得及刷新到硬盘文件，程序就崩溃了，所以，文件中并没有记录最后接收到的那些数据，我在文件中看到的最后一条记录并不是真正最后接收到的那条数据。发现了这个原因，我修改程序，在每一次调用 write 语句后，都立即调用 flush 语句，这样，我就终于找到了肇事元凶，并修复了程序的这个漏洞。

尽管我以前从来没有真正认真思考和编程试验过缓冲区问题，但是正因为还有那么一点点概念和印象，所以，在出现问题时，我才能从多方面去思考并最终解决问题。我建议读者花更多的时间去开阔自己的知识面和思维，了解更多的原理，而不是去花大量时间去死记硬背某些细节和术语，特别是一个类中的每个函数名的具体拼写、具体的参数形式，Java 中有哪些关键字等这些死板的东西，只要有个印象就足够了。

### 7.3.3 FileInputStream 与 FileOutputStream

这两个流节点用来操作磁盘文件，在创建一个 `FileInputStream` 对象时通过构造函数指定文件的路径和名字，当然这个文件应当是存在的和可读的。在创建一个 `FileOutputStream` 对象时指定文件如果存在将要被覆盖。

下面是对同一个磁盘文件创建 `FileInputStream` 对象的两种方式。其中用到的两个构造函数都可以引发 `FileNotFoundException` 异常：

```
FileInputStream inOne=new FileInputStream("hello.test");
File f = new File("hello.test");
FileInputStream inTwo = new FileInputStream(f);
```

尽管第一个构造函数更简单，但第二个构造函数允许在把文件连接到输入流之前对文件做进一步分析。

`FileOutputStream` 对象也有两个和 `FileInputStream` 对象具有相同参数的构造函数，创建一个 `FileOutputStream` 对象时，可以为其指定还不存在的文件名，但不能是存在的目录名，也不能是一个已被其他程序打开了的文件。`FileOutputStream` 先创建输出对象，然后再准备输出。

其实在上一章中讲 `Properties` 类的时候，我们已经使用过这两个类。在下面的例子中，我们用 `FileOutputStream` 类向文件中写入一串字符，并用 `FileInputStream` 读出。

```
程序清单：FileStream.java
import java.io.*;
public class FileStream
{
    public static void main(String[] args)
    {
        File f = new File("hello.txt");
        try
        {
            FileOutputStream out = new FileOutputStream(f);
            byte buf[] = "www.it315.org".getBytes();
            out.write(buf);
            out.close();
        }
        catch(Exception e)
        {
            System.out.println(e.getMessage());
        }
    }
}
```

```

    {
        FileInputStream in = new FileInputStream(f);
        byte [] buf = new byte[1024];
        int len = in.read(buf);
        System.out.println(new String(buf,0,len));
    }
    catch(Exception e)
    {
        System.out.println(e.getMessage());
    }
}
}

```

编译运行上面的程序，我们能够看到当前目录下产生了一个 hello.txt 的文件，用记事本程序打开这个文件，能看到我们写入的内容。随后，程序开始读取文件中的内容，并将读取到的内容打印出来。在这个例子中，我们演示了怎样用 `FileOutputStream` 往一个文件中写东西和怎样用 `FileInputStream` 从一个文件中将内容读出来。有一点不足的是，这两个类都只提供了对字节或字节数组进行读取的方法，对于字符串的读写，我们还需要进行额外的转换。

### 7.3.4 Reader 与 Writer

Java 中的字符是 unicode 编码，是双字节的，而 `InputStream` 与 `OutputStream` 是用来处理字节的，在处理字符文本时不太方便，需要编写额外的程序代码。Java 为字符文本的输入输出专门提供了一套单独的类，`Reader`、`Writer` 两个抽象类与 `InputStream`、`OutputStream` 两个类相对应，同样，`Reader`、`Writer` 下面也有许多子类，对具体 I/O 设备进行字符输入输出，如 `FileReader` 就是用来读取文件流中的字符。

对于 `Reader` 和 `Writer`，我们就不过多的说明了，大体的功能和 `InputStream`、`OutputStream` 两个类相同，但并不是它们的代替者，只是在处理字符串时简化了我们的编程。我们上面的程序改为使用 `FileWriter` 和 `FileReader` 来实现，修改后的程序代码如下：

```

import java.io.*;
public class FileStream
{
    public static void main(String[] args)
    {
        File f = new File("hello.txt");
        try
        {
            FileWriter out = new FileWriter(f);
            out.write("www.it315.org");
            out.close();
        }
        catch(Exception e)
        {
            System.out.println(e.getMessage());
        }
    }
}

```

```

    }

    try
    {
        FileReader in = new FileReader(f);
        char [] buf = new char[1024];
        int len = in.read(buf);
        System.out.println(new String(buf,0,len));
    }
    catch(Exception e)
    {
        System.out.println(e.getMessage());
    }
}
}

```

我们发现编译运行后的结果与先前没有什么两样，由于 `FileWriter` 可以往文件中写入字符串，我们不用将字符串转换为字节数组。相对于 `FileOutputStream` 来说，使用 `FileReader` 读取文件中的内容，并没有简化我们的编程工作，`FileReader` 的优势，要结合我们后面讲到的包装类才能体现出来。

### \$ 独家见解：

我们将程序中的 `out.close();` 语句注释掉后编译运行，在 `hello.txt` 文件中没有看到 `out.write` 语句写入的字符串，这可能就是我们前面谈到的缓冲区的原因，我们将 `out.close()` 改为 `out.flush` 后编译运行，在 `hello.txt` 文件中又能够看到 `out.write` 语句写入的字符串了，这更加证明了 `FileWriter` 使用了缓冲区。在使用 `FileOutputStream` 的例子程序中，我们同样注释掉 `out.close();` 语句，编译运行后，在 `hello.txt` 文件中能够看到 `out.write` 语句写入的字符串，这说明 `FileOutputStream` 没有使用缓冲区。

## 7.3.5 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。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。请看下面的例子：

程序清单： `PipeStreamTest.java`

```

import java.io.*;
public class PipeStreamTest
{
    public static void main(String args[])
    {
        try
        {
            Thread t1=new Sender();

```

```

        Thread t2=new Receiver();
        PipedOutputStream out = t1.getOutputStream();
        PipedInputStream in = t2.getInputStream();
        out.connect(in);
        t1.start();
        t2.start();
    }
    catch(IOException e)
    {
        System.out.println(e.getMessage());
    }
}
class Sender extends Thread
{
    private PipedOutputStream out=new PipedOutputStream();
    public PipedOutputStream getOutputStream()
    {
        return out;
    }
    public void run()
    {
        String s=new String("hello,receiver ,how are you");
        try
        {
            out.write(s.getBytes());
            out.close();
        }
        catch(IOException e)
        {
            System.out.println(e.getMessage());
        }
    }
}
class Receiver extends Thread
{
    private PipedInputStream in=new PipedInputStream();
    public PipedInputStream getInputStream()
    {
        return in;
    }
    public void run()
    {
        String s=null;

```

```

byte [] buf = new byte[1024];
try
{
    int len =in.read(buf);
    s = new String(buf,0,len);
    System.out.println("the      following      message      comes      from
sender:\n"+s);
    in.close();
}
catch(IOException e)
{
    System.out.println(e.getMessage());
}
}
}

```

运行结果：

```

the following message comes from sender:
hello,receiver ,how are you

```

JDK 还提供了 PipedWriter 和 PipedReader 这两个类来用于字符文本的管道通信，读者掌握了 PipedOutputStream 和 PipedInputStream 类，自然也就知道如何使用 PipedWriter 和 PipedReader 这两个类了。

## \$ 独家见解：

使用管道流类，可以实现各个程序模块之间的松耦合通信，我们可以灵活地将多个这样的模块的输出流与输入流相连接，以拼装成满足各种应用的程序，而不用对模块内部进行修改。

就象家庭的供水系统一样，我们可以把进水表的出水管与净化过滤器的进水管连在一起，然后，把净化过滤器的出水管同水箱的进水管连在一起拼凑成我们的供水管道系统。我们可以在这个供水管道系统中增加其他的水处理装置，也可以更换一个更大的水箱，甚至可以将进水表与水箱直连，而不经过净化过滤器，这一切都只需要各个水处理装置带有标准输入输出管道。

可见，使用管道流进行通信的模块具有“强内聚，弱耦合”的特点，一个模块被替换，或被拆卸不会影响其他模块。假设有一个使用了管道流的压缩或加密的模块，我们的调用程序只管向该模块的输入流中送入数据，从该模块的数据流中取得数据，就完成了我们数据的压缩或加密，这个模块完全就象黑匣子一样，我们根本不用去了解它的任何细节。

### 7.3.6 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数据源：

```

ByteArrayInputStream(byte[] buf)
ByteArrayInputStream(byte[] buf, int offset, int length)

```

第二个构造函数指定仅使用数组 `buf` 中的从 `offset` 开始的 `length` 个元素作为数据源。

`ByteArrayOutputStream` 是输出流的一种实现，它也有两个构造函数。

```
ByteArrayOutputStream()  
ByteArrayOutputStream(int)
```

第一种形式的构造函数创建一个 32 字节的缓冲区，第二种形式则是根据参数指定的大小创建缓冲区，缓冲区的大小在数据过多时能够自动增长。

这两个流的作用在于，用 I/O 流的方式来完成对字节数组内容的读写。爱思考的读者一定有过这样的疑问：对数组的读写非常简单，我们为什么不直接读写字节数组呢？我在什么情况下该使用这两个类呢？

有的读者可能听说过内存虚拟文件或者是内存映像文件，它们是把一块内存虚拟成一个硬盘上的文件，原来该写到硬盘文件上的内容会被写到这个内存中，原来该从一个硬盘文件上读取内容可以改为从内存中直接读取。如果程序在运行过程中要产生一些临时文件，就可以用虚拟文件的方式来实现，我们不用访问硬盘，而是直接访问内存，会提高应用程序的效率。

假设有一个别人已经写好了的压缩函数，这个函数接收两个参数，一个输入流对象，一个输出流对象，它从输入流对象中读取数据，并将压缩后的结果写入输出流对象中。我们的程序要将一台计算机的屏幕图像通过网络不断地传送到另外的计算机上，为了节省网络带宽，我们需要对一副屏幕图像的像素数据进行压缩后，再通过网络发送出去的。如果没有内存虚拟文件，我们就必须先将一副屏幕图像的像素数据写入到硬盘上的一个临时文件，再以这个文件作为输入流对象去调用那个压缩函数，接着又从压缩函数生成的压缩文件中读取压缩后的数据，再通过网络发送出去，最后删除压缩前后所生成的两个临时文件。可见这样的效率是非常低的。我们要在程序分配一个存储数据的内存块，通常都用定义一个字节数组来实现的，

JDK 中提供了 `ByteArrayInputStream` 和 `ByteArrayOutputStream` 这两个类可实现类似内存虚拟文件的功能，我们将抓取到的计算机屏幕图像的所有像素数据保存在一个数组中，然后根据这个数组创建一个 `ByteArrayInputStream` 流对象，同时创建一个用于保存压缩结果的 `ByteArrayOutputStream` 流对象，将这两个对象作为参数传递给压缩函数，最后从 `ByteArrayOutputStream` 流对象中返回包含有压缩结果的数组。

我们通过下面的例子程序来模拟上面的过程，我们并没有真正压缩输入流中的内容，只是把输入流中的所有英文字母变成对应的大写字母写入到输出流中。

```
程序清单：ByteArrayTest.java  
import java.io.*;  
public class ByteArrayTest  
{  
    public static void main(String[] args) throws Exception  
    {  
        String tmp="abcdefghijklmnopqrstuvwxyz";  
        byte [] src =tmp.getBytes(); //src 为转换前的内存块  
        ByteArrayInputStream input = new ByteArrayInputStream(src);  
        ByteArrayOutputStream output = new ByteArrayOutputStream();  
        new ByteArrayTest().transform(input,output);  
        byte [] result = output.toByteArray(); //result 为转换后的内存块  
        System.out.println(new String(result));  
    }  
    public void transform(InputStream in,OutputStream out)  
    {
```

```

int c=0;
try
{
    while((c=in.read())!=-1)//read 在读到流的结尾处返回-1
    {
        int C = (int)Character.toUpperCase((char)c);
        out.write(C);

    }
}
catch(Exception e)
{
    e.printStackTrace();
}
}

```

运行结果为：

**ABCDEFGHIJKLMNPQRSTUVWXYZ**

与 `ByteArrayInputStream` 和 `ByteArrayOutputStream` 类对应的字符串读写类分别是 `StringReader` 和 `StringWriter`。读者可以将上面的程序修改成由这两个类来完成，具体的程序代码就不在这里多说了。

### 7.3.7 IO 程序代码的复用

由于没有编码为-1的字符，所以，操作系统就使用-1作为硬盘上的每个文件的结尾标记，对于文本文件，我们的程序只要从文件中读取到了一个-1的字符值时，就可以确定已经到了这个文件结尾。注意，这种方式只能用于判断文本文件是否结束，不能判断一个二进制文件是否结束。尽管二进制文件的结尾标记也是-1，因为二进制文件中的每个字节可以是-128到127之间的任意取值，其中就包括-1，当程序读取到一个-1的字节时，就难以判定是文件结尾还是文件中的有效数据。对于标准的二进制文件，在文件开始部分，都有一个文件头指定文件的大小，程序就是凭借文件头中的这个大小来读取文件中的所有内容的。

我本人曾经为二进制文件和文本文件的区别困惑过很久，后来发现许多有一定软件开发经验的人也没完全搞清楚两者的区别。我们知道内存中的一个字节中数据可以是-128到127之间的任意值，实际上是以二进制形式存放的，文件就是一片内存的数据在硬盘上的另外一种存放形式，也都是二进制数据，所以，可以说每个文件都是二进制的。我们现在的每个字符由一个或多个字节组成，每个字节都是用的-128到127之间的部分数值来表示的，也就是说，-128到127之间还有一些数据没有对应任何字符的任何字节。如果一个文件中的每个字节中的内容都是可以表示成字符的数据，我们就可以称这个文件为文本文件，可见，文本文件只是二进制文件中的一种特例，为了与文本文件相区别，人们又把除了文本文件以外的文件称之为二进制文件。由于很难严格区分文本文件和二进制文件的概念，所以我们可以简单地认为，如果一个文件专用于存储文本字符的数据，没有包含字符之外的其他数据，我们就称之为文本文件，除此之外的文件就是二进制文件。

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文件，显示器可以被当作一个特殊的输出文件。当我们把键盘作为输入文件处理时，在 Windows 下，我们可以按下 **ctrl+z** 组合键来输入 -1 作为文件结束标记，在 Linux 下，我们可以按下 **ctrl+d** 组合键来输入 -1。

我们在编写流的程序时，应尽量考虑到程序代码的复用性，对于我们上面的程序代码，我们可以直接调用上面的 transform 方法，将键盘上输入的内容转变成大写字母后打印在屏幕上，程序代码如下：

```
import java.io.*;
public class ByteArrayTest
{
    public static void main(String[] args) throws Exception
    {
        new ByteArrayTest().transform(System.in, System.out);
    }

    public void transform(InputStream in, OutputStream out)
    {
        int c=0;
        try
        {
            while((c=in.read())!=-1)
            {
                int C = (int)Character.toUpperCase((char)c);
                out.write(C);

            }
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

我们没有修改 transform 方法中的任何代码，就利用它完成了我们期望的功能。我们还可以使用 transform 方法将一个文件中的内容全部变成大写字母后写入另外一个文件，也可以将键盘上输入的内容转变成大写字母后写入另外一个文件，这就是因为我们在 transform 方法中使用的是 InputStream 和 OutputStream 这两个抽象基类，而不是直接使用某个具体的子类，这样就达到了以不变应万变的效果。

如果我们平时从键盘上读取内容的程序代码也放在一个类似 transform 方法的函数中去完成，也是用 -1 来作为键盘输入的结束，在该函数中不直接使用 System.in，只是在调用该函数时，将 System.in 作为参数传递进去。这样，我们以后要从某个文件中读取数据，来代替手工键盘输入时，我们可以直接使用这个函数，程序就不用作太多的修改了。

## 7.4 过滤流与包装类

### 7.4.1 理解包装类的概念与作用

在前面的部分，我们接触到了许多节点流类，就以 `FileOutputStream` 和 `FileInputStream` 为例吧，这两个类只提供了读写字节的方法，我们通过它们只能往文件中写入字节或从文件中读取字节。在实际应用中，我们要往文件中写入或读取各种类型的数据，我们就必须先将其他类型的数据转换成字节数组后写入文件或是将从文件中读取到的字节数组转换成其他类型，这给我们的程序带来了一些困难和麻烦。如果有人给我们提供了一个中间类，这个中间类提供了读写各种类型的数据的各种方法，当我们需要写入其他类型的数据时，只要调用中间类中的对应的方法即可，在这个中间类的方法内部，它将其他数据类型转换成字节数组，然后调用底层的节点流类将这个字节数组写入目标设备。我们将这个中间类叫做过滤流类或处理流类，也叫包装类，如 IO 包中有一个叫 `DataOutputStream` 的包装类，下面是它所提供的部分方法的列表。

```
public final void writeBoolean(boolean v) throws IOException  
public final void writeShort(int v) throws IOException  
public final void writeChar(int v) throws IOException  
public final void writeInt(int v) throws IOException  
public final void writeLong(long v) throws IOException  
public final void writeFloat(float v) throws IOException  
public final void writeDouble(double v) throws IOException  
public final void writeBytes(String s) throws IOException
```

大家从上面的方法名和参数类型中，就知道这个包装类能帮我们往 IO 设备中写入各种类型的数据。包装类的调用过程如图 7.2 所示：

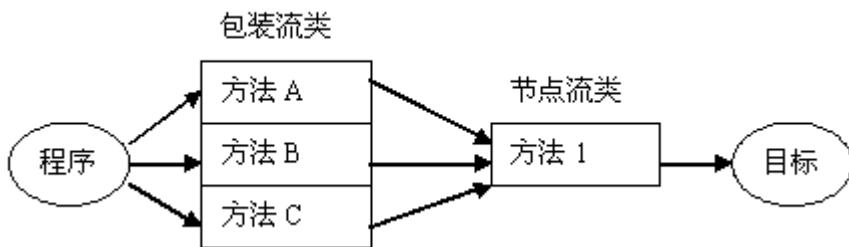


图 7.2

使用输出包装类的过程，就好比我们要给某个市长送礼，该市长向来不接受陌生人的礼品，但其夫人则是来者不拒的。我们只要将礼品送到市长夫人手中，就等于送到了市长的手中。市长夫人就是我们用到的输出包装类。

使用输入包装类过程，好比我们要想借用市长大人的一点权力，承揽一个假竞标的工程项目，我们是没法直接请动市长大人来替我们说话的，但我们可以让市长公子来替我们办好这件事，我们就间接借用了市长的权力。市长公子就是我们用到的输入包装类。

我们还可以用包装类去包装另外一个包装类，创建包装类对象时，必须指定它要调用的那个底层流对象，也就是这些包装类的构造函数中，都必须接收另外一个流对象作为参数。如 `DataOutputStream` 包装类的构造函数为：

```
public DataOutputStream(OutputStream out)
```

参数 out 就是 DataOutputStream 要调用的那个底层输出流对象。

### 7.4.2 BufferedInputStream 与 BufferedOutputStream

对 I/O 进行缓冲是一种常见的性能优化。缓冲流为 I/O 流增加了内存缓冲区。增加缓冲区有两个基本目的：

- ü 允许 Java 的 I/O 一次不只操作一个字节，这样提高了整个系统的性能。
- ü 由于有缓冲区，使得在流上执行 skip、mark 和 reset 方法都成为可能。

#### BufferedInputStream

Java 的 BufferedInputStream 类可以对任何的 InputStream 进行带缓冲区的封装以达到性能的改善。BufferedInputStream 有两个构造函数：

```
BufferedInputStream(InputStream in)  
BufferedInputStream(InputStream in, int size)
```

第一种形式的构造函数创建了一个带有 32 字节缓冲区的缓冲流，第二种形式的构造函数按指定的大小来创建缓冲区。通常缓冲区大小是内存、磁盘扇区或其它系统容量的整数倍，这样就可以充分提高 I/O 的性能。一个最优的缓冲区的大小，取决于它所在的操作系统、可用的内存空间以及机器的配置。

对输入流进行缓冲可以实现部分字符的回流。除了 InputStream 中常用的 read 和 skip 方法，BufferedInputStream 还支持 mark 和 reset 方法。mark 方法在流的当前位置作一个标记，该方法接收的一个整数参数用来指定从标记处开始，还能通过 read 方法读取的字节数，reset 方法可以让以后的 read 方法重新回到 mark 方法所作的标记处开始读取数据。

**M** 脚下留心：mark 只能限制在建立的缓冲区内。

#### BufferedOutputStream

往 BufferedOutputStream 输出和往 OutputStream 输出完全一样，只不过 BufferedOutputStream 有一个 flush 方法用来将缓冲区的数据强制输出完。与缓冲区输入流不同，缓冲区输出流没有增加额外的功能。在 Java 中使用输出缓冲也是为了提高性能。它也有两个构造函数：

```
BufferedOutputStream(OutputStream out)  
BufferedOutputStream(OutputStream out, int size)
```

第一种形式创建一个 32 字节的缓冲区，第二种形式以指定的大小来创建缓冲区。

### 7.4.3 DataInputStream 与 DataOutputStream

这两个类提供了可以读写各种基本数据类型的数据的各种方法，这些方法是使用非常简单，在前面我们已经看到了 DataOutputStream 类的大部分方法，在 DataInputStream 类有与这些 write 方法对应的 read 方法，读者可以在 JDK 文档帮助查看详细的信息。

DataOutputStream 类提供了三种写入字符串的方法，分别是

```
public final void writeBytes(String s) throws IOException  
public final void writeChars(String s) throws IOException  
public final void writeUTF(String str) throws IOException
```

这三种方法有什么区别呢？Java 中的字符是 unicode 编码，是双字节的，writeBytes 只将字符串中的每一个字符的低字节的内容写入目标设备中，而 writeChars 将字符串中的

每一个字符的两个字节的内容都写入到目标设备中。writeUTF 对字符串按照 UTF 格式写入目标设备, UTF 是带有长度头的, 最开始的两个字节是对字符串进行 UTF 编码后的字节长度, 然后才是每个字符的 UTF 编码。字符的 UTF 编码对应下列规则:

- Ø 假如字符 c 的范围在 \u0001 和 \u007f 之间, 对应的 UTF 码占一个字节, 内容为: (byte)c。
- Ø 假如字符 c 是 \u0000 或其范围在 \u0080 和 \u07ff 之间, 对应的 UTF 码占两个字节, 内容为: (byte)(0xc0 | (0x1f & (c >> 6))), (byte)(0x80 | (0x3f & c))。
- Ø 假如字符 c 的范围在 \u0800 和 \uffff 之间, 对应的 UTF 码占三个字节, 内容为: (byte)(0xe0 | (0x0f & (c >> 12))), (byte)(0x80 | (0x3f & (c >> 6))), (byte)(0x80 | (0x3f & c))

在与 DataOutputStream 类对应的输入流 DataInputStream 类中只提供了一个 readUTF 方法返回字符串, 也就是 DataInputStream 类中没有直接读取到 DataOutputStream 类的 writeBytes 和 writeChars 方法写入的字符串, 这又是为什么呢? 我们要在一个连续的字节流读取一个字符串(只是流中的一段内容), 如果没有特殊的标记作为一个字符串的结尾, 而且和我们事先也不知道这个字符串的长度, 我们是没法知道读取到什么位置才是这个字符串的结束。在 DataOutputStream 类中只有 writeUTF 方法向目标设备中写入了字符串的长度, 所以, 我们也只能准确地读回这个方法写入的字符串。

我们下面的程序使用了多个流对象来进行文件的读写, 这多个流对象形成了一个链, 我们称之为流栈, 如图 7.3 所示:

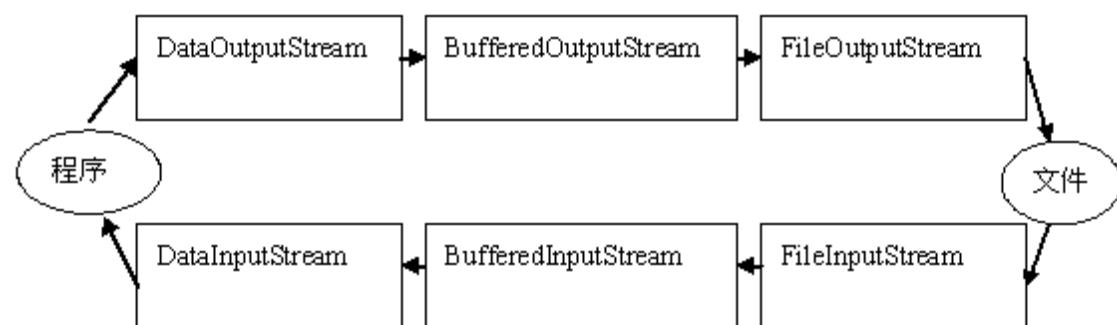


图 7.3

```
import java.io.*;
public class DataStreamTest
{
    public static void main(String[] args)
    {
        try
        {
            FileOutputStream fos = new FileOutputStream("hello.txt");
            BufferedOutputStream bos = new BufferedOutputStream(fos);
            DataOutputStream dos = new DataOutputStream(bos);

            dos.writeUTF("ab 中国");
            dos.writeBytes("ab 中国");
            dos.writeChars("ab 中国");
            dos.close();
        }
    }
}
```

```

        FileInputStream fis = new FileInputStream("hello.txt");
        BufferedInputStream bis = new BufferedInputStream(fis);
        DataInputStream dis = new DataInputStream(bis);
        System.out.println(dis.readUTF());
        /*byte [] buf=new byte[1024];
        int len = dis.read(buf);
        System.out.println(new String(buf,0,len));*/
        fis.close();
    }
    catch(Exception e)
    {
        System.out.println(e.getMessage());
    }
}
}

```

如果正在使用一个流栈，程序关闭最上面的一个流也就自动关闭了栈中的所有底层流，所以程序中只调用了 `DataInputStream` 与 `DataOutputStream` 这两个流对象的 `close` 方法。我们用记事本程序打开 `hello.txt` 文件，显示内容如下

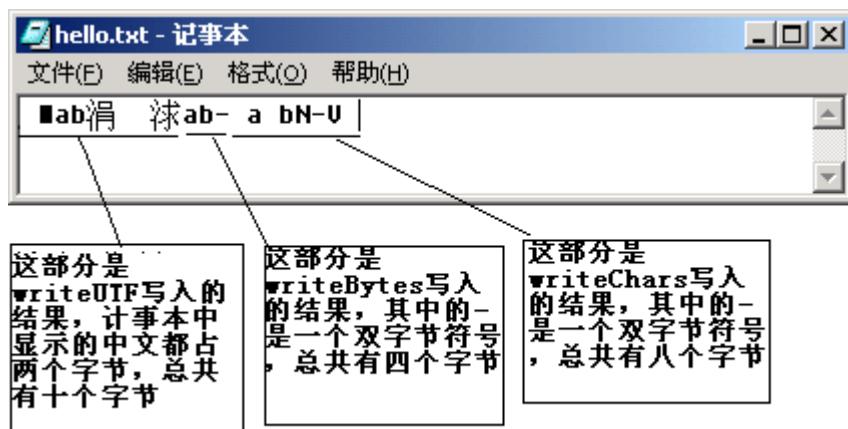


图 7.4

我们能看出其中的大概，就如同作者直接在上面的图中进行标注的那样，`writeChars` 写入的 `a` 字符都占用两个字节。尽管我们在记事本程序中看不出 `writeUTF` 写入的字符串是“ab 中国”，但程序通过 `readUTF` 读回后显示在屏幕上的仍是“ab 中国”，这个过程就好比一个写入函数把字符串加密后写入文件，我们用记事本程序是看不出其实际写入的内容的，但对应的读取函数却能正确返回先前写入的字符串，因为读取函数内部知道如何解密。`writeChars` 和 `writeBytes` 方法写入的字符串，我们要想读取回来，就没这么幸运了，读者可以借鉴作者在程序中注释掉的那段代码，运行后没有把我们写入的字符串打印出来，你就能够明白我们要将 `writeChars` 和 `writeBytes` 方法写入的字符串正确读取回来，实在太困难了，所以，`io` 包中专门提供了各种 `Reader` 和 `Writer` 类来操作字符串。

如果读者想仔细研究上面几个 `write` 方法写入的字符串在 `hello.txt` 文件中到底以何种形式存在的，可以使用 `UltraEdit` 打开 `hello.txt` 文件，显示的内容如下：

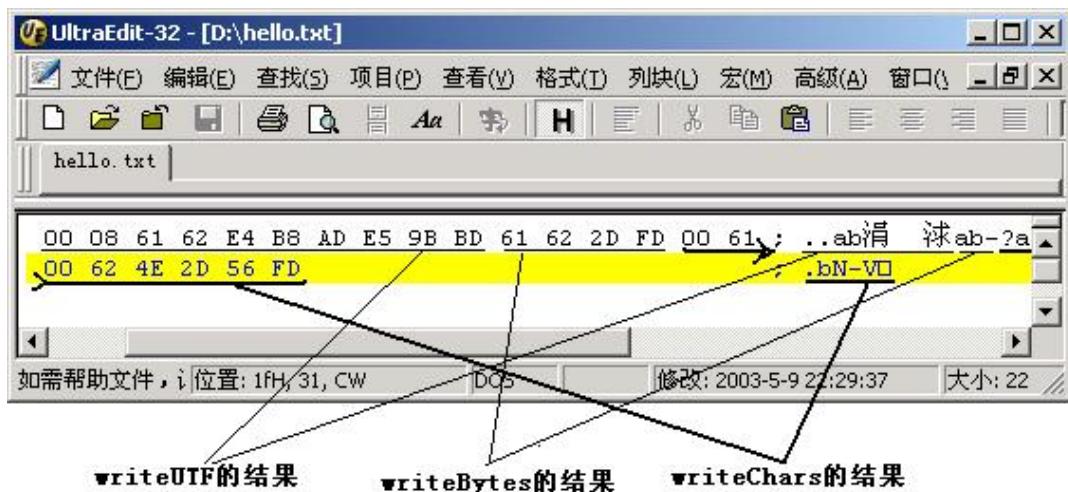


图 7.5

通过 UltraEdit，我们看到了每个字节所对应的具体数值，所以，有经验的人士经常用 UltraEdit 来查看和研究二进制文件的内容信息。

#### 7.4.4 PrintStream

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化成字符串输出。在前面，我们在程序中大量用到 “System.out.println” 语句中的 System.out 就是 PrintStream 类的一个实例对象，读者已经多次使用到这个类了。PrintStream 有 3 个构造函数：

```
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，我们在本章后面部分有详细的讨论。

println 方法与 print 方法的区别是：前者会在打印完的内容后多打印一个换行符(\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后再输出，而不是输出原始的字节内容，如：整数 123 的打印结果是字符‘1’、‘2’、‘3’所组合成的一个字符串，而不是整数 123 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 方法返回的字符串。

I/O 包中提供了一个与 PrintStream 对应的 PrintWriter 类，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"，如果我们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，我们就应该使用 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。

# F 指点迷津：

格式化输出是指将一个数据用其字符串格式输出，如我们使用 `print` 方法把 97 这个整数打印到一个文件中，该方法将把 ‘9’ 和 ‘7’ 这两个字符的 ASCII 码写入到文件中，也就是文件中会被写入两个字节，这两个字节中的数字分别为 57 (十六进制的 0x39) 和 55 (十六进制的 0x37)，在记事本程序中显示为 ‘9’ 和 ‘7’ 这两个字符。如果我们使用 `write` 方法把 97 这个整数写到一个文件中，只有一个字节会写入到这个文件中，字节中的数字就是 97，正好是字符 ‘a’ 的 ASCII 码，所以在记事本程序中显示为一个字符 ‘a’ 。

## 7.4.5 ObjectInputStream 与 ObjectOutputStream

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，我们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，我们只要读取到一个对象中原来保存的所有成员变量的取值，就等于读取到了一个对象。`ObjectInputStream` 与 `ObjectOutputStream` 类，可以帮我们完成保存和读取对象成员变量取值的过程，但要读写或存储的对象必须实现了 `Serializable` 接口，`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。`ObjectInputStream` 与 `ObjectOutputStream` 类不会保存和读取对象中的 `transient` 和 `static` 类型的成员变量，使用 `ObjectInputStream` 与 `ObjectOutputStream` 类保存和读取对象的机制叫序列化，如下面定义了一个可以被序列化的 `MyClass` 类：

```
public class MyClass implements Serializable
{
    public transient Thread t;
    private String customerID;
    private int total;
}
```

在 `MyClass` 类的实例对象被序列化时，成员变量 `t` 不会被保存和读取。

序列化的好处在于：它可以将任何实现了 `Serializable` 接口的对象转换为连续的字节数据，这些数据以后仍可被还原为原来的对象状态，即使这些数据通过网络传输也没问题。序列化能处理不同操作系统上的差异，我们可以在 Windows 上产生某个对象，将它序列化存储，然后通过网络传到 Linux 机器上，该对象仍然可以被正确重建出来，在这期间，我们完全不用担心不同机器上的不同的数据表示方式。

下面我们就创建一个学生对象，并把它输出到一个文件 (`mytext.txt`) 中，然后再把该对象读出来，将其还原后打印出来：

程序清单：`Serialization.java`

```
import java.io.*;
public class serialization
{
    public static void main(String args[])
        throws IOException, ClassNotFoundException
    {
        Student stu=new Student(19,"dintdding",50,"huaxue");
        FileOutputStream fos=new FileOutputStream("mytext.txt");
        ObjectOutputStream os=new ObjectOutputStream(fos);
```

```

try
{
    os.writeObject(stu);
    os.close();
}catch(IOException e)
{
    System.out.println(e.getMessage());
}
stu=null;
FileInputStream fi=new FileInputStream("mytext.txt");
ObjectInputStream si=new ObjectInputStream(fi);
try
{
    stu=(Student)si.readObject();
    si.close();
}catch(IOException e)
{
    System.out.println(e.getMessage());
}
System.out.println("ID is:"+stu.id);
System.out.println("name is:"+stu.name);
System.out.println("age is:"+stu.age);
System.out.println("department is:"+stu.department);
}

}

class Student implements Serializable
{
    int id;
    String name;
    int age;
    String department;
    public Student(int id,String name,int age,String department)
    {
        this.id=id;
        this.name=name;
        this.age=age;
        this.department=department;
    }
}

```

运行结果：

```

ID is:19
name is:dintdding

```

```
age is: 50  
department is: huaxue
```

从运行结果上看，我们刚刚读出来并还原的内容和我们原来创建时是一样的。我们到底写了些什么内容到 mytext.txt 文件中呢？我们用记事本程序打开 mytext.txt 文件时所看到的内容如图 7.6 所示：

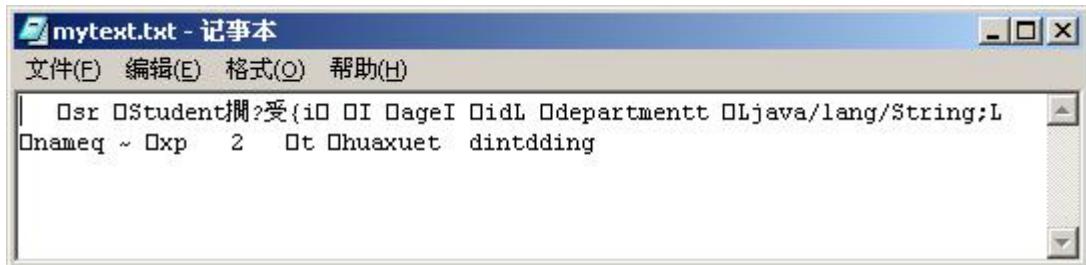


图 7.6

我们不用了解其中的详细细节，只要能够通过相应的方式正确地读取回来就足够了。

## F 指点迷津：

一个学员曾经问过我，他们公司买了一套美国人的地理信息系统，这个系统将采集到的地理数据存放在一个文件中，他有没有办法读取到这个文件中的内容？看来，他还没有完全明白这些 I/O 类能帮助我们做些什么。我告诉他，用我们的前面讲的 FileInputStream 类就能够读取到这个文件中的所有字节的数据，只是我们不明白这些数据代表的是什么意思罢了，也就是说我们不知道美国人存储数据的格式，读到了这些数据也是白读！只有开发那个系统的美国人自己知道这些数据的意义，他们才能正确地使用文件中保存的数据。就象 ObjectOutputStream 保存的数据一样，是专门给 ObjectInputStream 来读取的，我们通过别的方式读取到的数据毫无意义。

### 7.4.6 字节流与字符流的转换

前面我们讲过，Java 支持字节流和字符流，我们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，`OutputStreamWriter` 将写入的字符编码成字节后写入一个字节流。其中 `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) //接受以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 CharsetName) //接受以指定字符集名的字符串，  
//并用该字符集创建 OutputStreamWriter 对象
```

为了达到最好的效率，避免频繁的字符与字节间的相互转换，我们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`。例如：

```
BufferedWriter out=new BufferedWriter(new OutputStreamWriter(System.out));
```

```
BufferedReader in=new BufferedReader(new InputStreamReader(System.in));
```

我们接着从一个更实际的应用中来熟悉 `InputStreamReader` 的作用, 怎样用一种简单的方式一下就读取到键盘上输入的一整行字符? 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();
```

我们不可能什么时候都提前掌握了正好可以解决我们问题的各个小知识点, 作者在第一次碰到这种需求时, 就不知道可以用这种方式, 但作者在以前从没有接触的情况下, 也写出了上面的代码。首先, 要读取一行, 我马上想到在 `chm` 格式的 JDK 文档中去查类似 `readLine` 这样的英文单词的拼写组合, 查询的界面如图 7.7 所示:

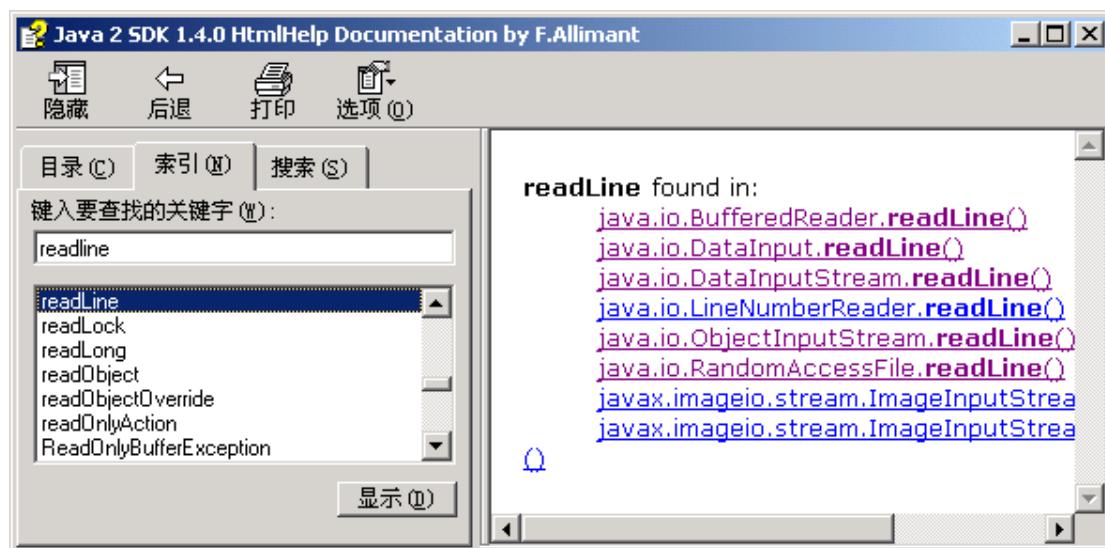


图 7.7

我们找到了 `BufferedReader` 这个类, 查看 `BufferedReader` 类的构造方法, 如图 7.8 所示:

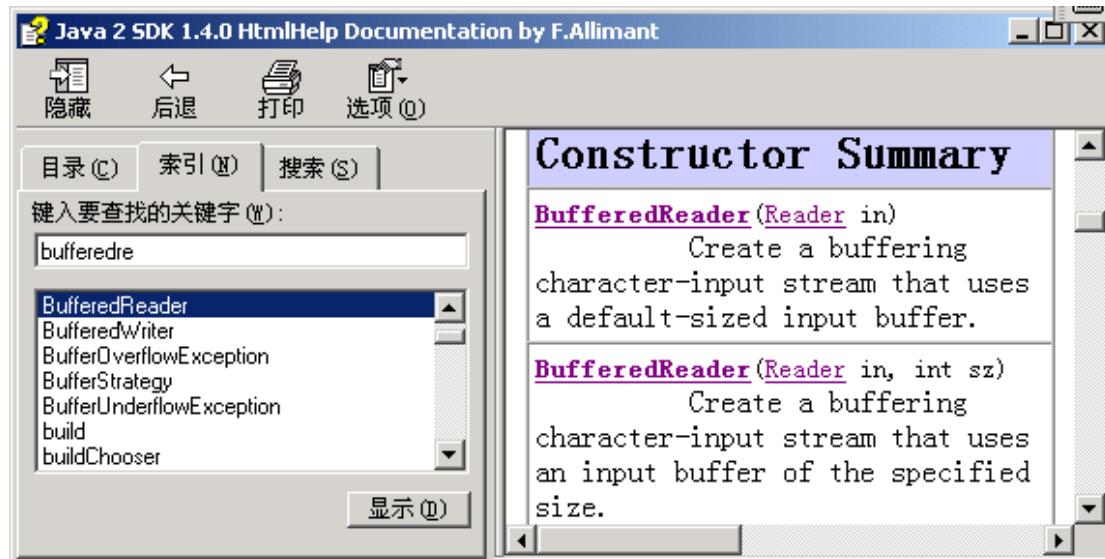


图 7.8

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, 解决问题的关键是, 我们还需要找到

将 `InputStream` 类型的流对象包装成 `Reader` 类型的包装类。作者眼尖（其实是作者读文档的一个习惯，也可以说是作者查文档的一点小经验吧），就在 `BufferedReader` 的帮助界面中，我看了如下的一段信息：

See Also:

`FileReader`, `InputStreamReader`

在这里，我看到了 `InputStreamReader` 这个关键的类，阅读其帮助后，最终写出了上面的程序代码，轻松解决了我从未碰到过的问题。在查阅文档时，经常顺便看看 See Also 部分也是很重要的，在那里往往都有解决相关问题的超链接。如果在 See Also 部分也没有提及 `InputStreamReader`，那我们只能去查 `I`O 包的帮助，浏览其中列出的每个类，也能发现 `InputStreamReader` 这个类就是我们所要找的类的。

`BufferedReader` 类可以读取一行文本，对应的 `BufferedWriter` 类也提供了一个 `newLine` 方法来向字符流中写入不同操作系统下的换行符，如果我们要向字符流中写入与平台相关的文本换行，就可以考虑使用 `BufferedWriter` 这个包装类了。

我们在前面用到的 `FileWriter` 和 `FileReader` 实际上都是包装类，`FileReader` 是 `InputStreamReader` 的子类，`FileWriter` 是 `OutputStreamWriter` 的子类。

#### 7.4.7 IO 包中的类层次关系图

1. 字节输入流类：

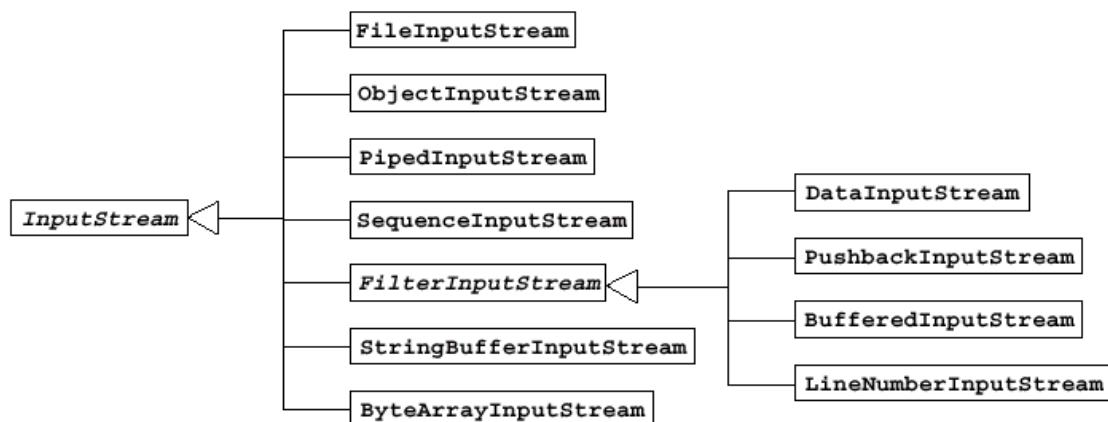


图 7.9

2. 字节输出流类：

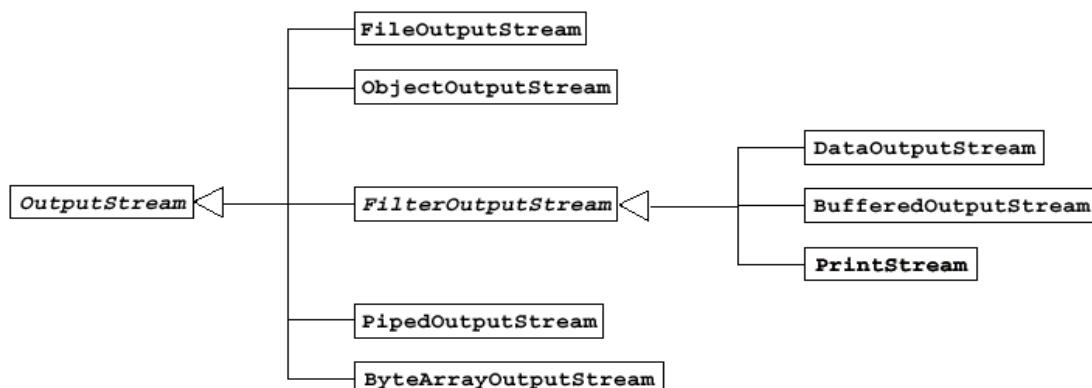


图 7.10

3. 字符输入流类:

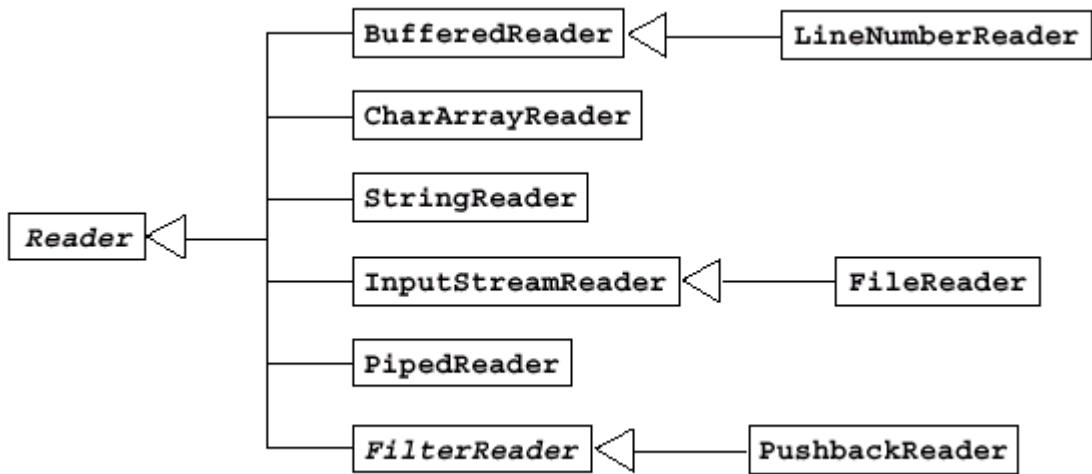


图 7.11

4. 字符输出流类:

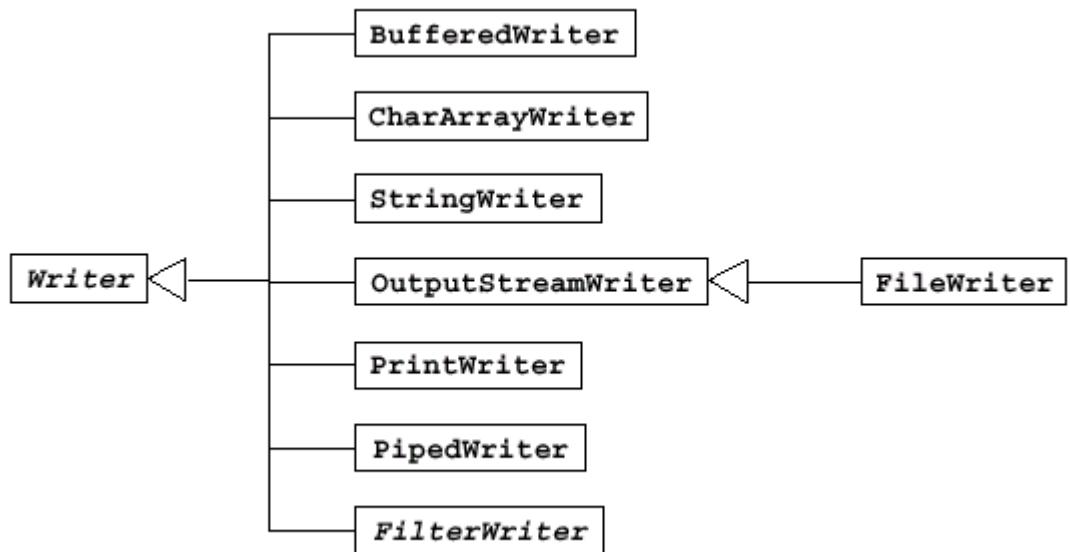


图 7.12

## 7.5 IO 中的高级应用

### 7.5.1 字符集的编码问题

计算机里只有数字，我们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我们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（美国标准信息交换码）。

随着计算机逐渐在其他国家的应用和普及，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，可见gbk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，我们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，使用各个国家和地区的本地化字符编码，已经给我们的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本地化字符编码带来的不利影响，人们将全世界所有的符号进行了统一编码，称之为uni code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，至于那个伊拉克能不能认识这个符号，就不是我们计算机所要解决的问题了。Uni code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)，我想一定是uni code编码中没有包括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。uni code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uni code编码共存的景象。既然本地化字符编码与uni code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uni code编码，Java技术在通过Uni code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，而我们显示输出和键盘输入都是采用的本地编码。

作者在上面的讲解中，写出了“中国”的“中”字在gbk，big5，uni code编码中分别对应的数字，读者是否对此感到奇怪过，是作者记忆力超群吗？非也！作者就是通过下面的实验而得到的这几个数字并借此帮助读者完全理解字符编码的问题。

步骤1，在UltraEdit中，输入“中国”，再按下工具栏上的“H”样的按钮，用十六进

制方式查看“中国”这两个字符在本地系统编码中所对应的字节数字，如图 7.13 所示：



图 7.13

步骤 2，编写并运行下面的程序代码

```
public class CharCode
{
    public static void main(String [] args) throws Exception
    {
        String strChina = "中国";
        for(int i=0;i<strChina.length();i++)
        {
            System.out.println(Integer.toHexString((int)strChina.charAt(i)));
        }
        byte [] buf=strChina.getBytes("gb2312");
        for(int i=0;i<buf.length;i++)
        {
            System.out.println(Integer.toHexString(buf[i]));
        }
        System.out.println(strChina);
        for(int i=0;i<buf.length;i++)
        {
            System.out.write(buf[i]);
        }
        System.out.println();//试试没有这一句的效果
    }
}
```

运行的结果如下：

```
4e2d
56fd
fffffd6
fffffd0
fffffb9
fffffa
中国
```

## 中国

Java 中的字符采用的是 uni code 编码，每个字符都占用两个字节，我们直接把每个字符中的内容对应着的整数打印出来，显示的结果就是这个字符的 uni code 码。String 类中的 getBytes 方法，并不是简单地将字符串中的每个字节数据存放到一个字节数组中去，而是将 uni code 码的字符串中的每个字符数字，转换成该字符在指定的字符集下的数字，最后将这些数字存放到一个字节数组中返回。将一个字符的 uni code 码转换成某种本地字符集码的过程叫编码，将 uni code 码成功地转换到本地字符集码，在 JDK 包中必须有对应的字符集编码器类。

打印出编码转换后的每个字节，我们就可以看到字符在该字符集下的编码。由于我们要将字节转换成整数后才能以十六进制的形式打印出来，从程序打印的结果上，我们看到 d6 以 ffffffd6 的形式打印出来，这说明如果字节的最高位(bi t)为 1，转换后的整数的三个高字节的每个 bi t 位的内容也都是 1，如果大家明白在计算机中是如何表示负数的，就不难想明白其中的道理了，如程序中打印的 ffffffd6 所对应的字节在内存中实际上是一个字节的数据 d6。我们只要取打印结果的最低字节的数字，就是我们原始字节中的数字，将这些数字与我们在步骤 1 看到的数字对比，我们就看到了 getBytes 成功地将 uni code 码转换成了 gb2312 码。

如果程序中没有最后的 System.out.println 语句，屏幕上不会打印出最后的“中国”，这是为什么？前面讲过，System.out 是包装类 PrintStream 的一个实例对象，包装类都是有缓冲的，PrintStream 类在调用了 println 方法后会自动刷新缓冲区的内容。从最后屏幕上正常打印出“中国”这两个字符，我们又可以得到这么一个结论：要正确地在屏幕上打印中文字符，我们写入屏幕输出流的字节内容必须是该中文字符的 gb2312 码，要将中文字符正确的存入硬盘文件也是一样的道理。我们还可以进而推断：System.out.println("中国") 中的 println 方法实际上是先把“中国”转换成其 gb2312 码的字节数组，然后调用 write 方法将这个字节数组写入到输出流中。那么，println 方法怎么知道要将“中国”转换成 gb2312 码，而不是 big5 码或其他的字符集呢？前面讲过，在创建 PrintStream 实例对象的构造方法中可以指定一个编码参数，在我们使用的中文版的 Windows 操作系统上，System.out 对象就是 Java 系统按照 gb2312 编码方式创建出来的 PrintStream 实例对象，这是一个非常底层的问题，有兴趣的读者可以去研究 JDK 中的源码。

步骤 3，String 类中有一个没有参数的 getBytes 方法，它将使用系统缺省的编码器对字符编码，我们将上面程序中的

```
byte [] buf=strChina.getBytes("gb2312");
```

修改为

```
byte [] buf=strChina.getBytes();
```

重新编译后运行的结果和修改前的结果一样，这说明我们系统的缺省编码方式就是 gb2312。在程序的开始处，我们增加下面的一条语句：

```
System.getProperties().list(System.out);
```

程序运行的结果如图 7.14 所示：

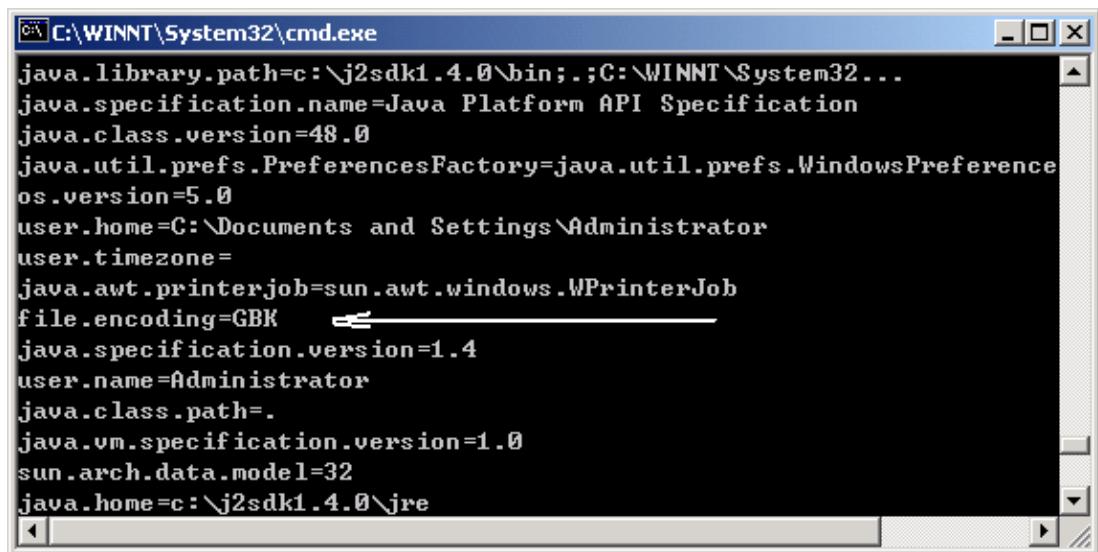


图 7.14

如图中的箭头所指，作者所用系统的缺省编码方式为 gbk，也就是 gb2312。

接着，我们修改系统的缺省编码方式，修改后的程序代码如下：

```
public class CharCode
{
    public static void main(String [] args) throws Exception
    {
        System.getProperties().put("file.encoding","iso8859-1");
        System.getProperties().list(System.out);

        String strChina = "中国";
        for(int i=0;i<strChina.length();i++)
        {
            System.out.println(Integer.toHexString((int)strChina.charAt(i)));
        }
        byte [] buf=strChina.getBytes();
        for(int i=0;i<buf.length;i++)
        {
            System.out.println(Integer.toHexString(buf[i]));
        }
        System.out.println(strChina);
        for(int i=0;i<buf.length;i++)
        {
            System.out.write(buf[i]);
        }
        System.out.println();//试试没有这一句的效果
    }
}
```

重新运行后，程序最后在屏幕上打印的几行如下：

.....

```
4e2d  
56fd  
3f  
3f  
中国  
??
```

我们成功地将系统的缺省编码方式修改成了 iso8859-1, iso8859-1 标准的西方英语国家字符集码，在 iso8859-1 字符集中是没有“中国”这样的字符的，编码的结果当然也就没有意义了，打印的结果也就不正常了。

步骤 4，我们将键盘字节输入流中的每个字节读取到一个字节数组中，然后将字节数组中的数据当作某种本地字符集码转换成 uni code 码的字符串，这个过程叫解码，要能够正确地完成解码工作，在 JDK 包中必须有对应的字符集解码器类。实验并观察下面的程序代码的运行结果。

```
import java.io.*;  
public class CharDecoder  
{  
    public static void main(String [] args) throws Exception  
    {  
        System.out.println("please enter a Chinese String: ");  
        byte [] buf=new byte[1024];  
        int ch=0;  
        int pos=0;  
        String strInfo=null;  
        while(true)  
        {  
            ch =System.in.read();  
            System.out.println(Integer.toHexString(ch));  
            switch(ch)  
            {  
                case '\r':  
                    break;  
                case '\n':  
                    strInfo= new String(buf,0,pos,"gb2312");  
                    for(int i=0;i<strInfo.length();i++)  
                    {  
                        System.out.println(Integer.toHexString((int)strInfo.charAt(i)));  
                    }  
                    System.out.println(strInfo);  
                    for(int i=0;i<pos;i++)  
                        System.out.write(buf[i]);  
                    System.out.println();//想想为什么要这一句  
                    return;  
                default:  
                    buf[pos++]=(byte)ch;  
            }  
        }  
    }  
}
```

```
        }
    }
}
}
```

编译运行后的效果如下：

```
please enter a Chinese String:
```

```
中国
```

```
d6
```

```
d0
```

```
b9
```

```
fa
```

```
d
```

```
a
```

```
4e2d
```

```
56fd
```

```
中国
```

```
中国
```

从运行的结果上，我们可以看出通过键盘输入的中文字符，在键盘输入流中的字节数据是这个字符的gb2312码对应的数字。另外，上面打印出的d和a分别是字符'\r'和'\n'的十六进制数据。

如果String类的构造函数中没有指定解码方式，它将使用系统缺省的解码器将字节数组中的数据解码成uni code码的字符串，我们将上面程序中的

```
strInfo= new String(buf,0,pos,"gb2312");
```

修改为

```
strInfo= new String(buf,0,pos);
```

重新编译后运行的结果和修改前的结果一样，这说明我们系统的缺省解码方式就是gb2312。

步骤5，我们将系统的缺省解码方式修改成了iso8859-1或在String类的构造函数中明确指定用iso8859-1解码，来观察程序运行的结果，修改后的程序代码如下：

```
import java.io.*;
public class CharDecoder
{
    public static void main(String [] args) throws Exception
    {
        System.getProperties().put("file.encoding","iso8859-1");
        System.out.println("please enter a Chinese String");
        byte [] buf=new byte[1024];
        int ch=0;
        int pos=0;
        String strInfo=null;
        while(true)
        {
            ch =System.in.read();
            System.out.println(Integer.toHexString(ch));
        }
    }
}
```

```

switch(ch)
{
    case '\r':
        break;
    case '\n':
        strInfo= new String(buf,0,pos);
        for( int i=0;i<strInfo.length();i++)
        {

            System.out.println(Integer.toHexString((int)strInfo.charAt(i)));
        }
        System.out.println(strInfo);
        for( int i=0;i<pos;i++)
            System.out.write(buf[i]);
        System.out.println();//想想为什么要这一句
        return;
    default:
        buf[pos++]=(byte)ch;
}
}
}
}

```

编译运行后打印的结果如下：

```

please enter a Chinese String
中国
d6
d0
b9
fa
d
a
d6
d0
b9
fa
???
中国

```

可见，装有“中国”这两个字符的gb2312码的字节数组，使用iso8859-1解码后的uni code字符串中的字符并不是“中国”这两个字符的uni code码，而是被解码成了四个字符，每个字符的低字节的内容都是原来字节数组中的数据，而高字节都是0，如程序中打印的d6所对应的字符在内存中实际上是两个字节00d6。

步骤6，我们在实际的开发中，经常会遇到诸如步骤5中出现的字符编码问题，假设别人给我们提供的某个方法返回的字符串是用的iso8859-1解码而成的，在遇到中文也会出现程序中System.out.println(strInfo);打印出的是乱码的问题，就拿上面的程序来说，我

们有没有办法修改程序中的打印语句，让其能打印出正确的中文字符呢？要注意到我们的前提条件：返回字符串的函数是别人提供的，我们不能修改生成字符串的那部分程序代码。字符串用的是 iso8859-1 解码而成的，我们只要将其按 iso8859-1 又编码成字节数组，字节数组中的内容就是中文字符最初的那个字符集（这里就以中国大陆地区的 gb2312 为例）的编码值，然后，我们对这个字节数组再按 gb2312 解码成 uni code 的字符串就行了。

我们只要将

```
System.out.println(strInfo);
```

修改成

```
String strChina = new String(strInfo.getBytes("gb2312"), "iso8859-1");
System.out.println(strChina);
```

打印中文的结果就正常了。

作者从上面的实验结果还得到了另外一些启发：先将字节数组解码成字符串，以后还可以将这个字符串又反向编码成最初的字符数组；但先将一个字符串的内容先编码成字节数组，却不一定能够反向解码成最初的字符串。其实细心的读者在步骤 3 中，就已经看到了把“中国”这个字符串按 iso8859-1 编码成字节数组后打印出的数据，不可能反向解码回去的。因为一个字符占两个字节，字符串中的原来两个字节的内容按 iso8859-1 编码后只有一个字节，两个字节能表示 65536 个数字，而一个字节只能表示 256 个数值，这个编码过程显然不可能一一对应，会有数据的丢失。

## F

### 指点迷津：

字符编码在很多人眼里是一个复杂的问题，很多“高手”也尽量避开与人交谈这个问题。如果你掌握了本节的内容，在遇到字符编码问题时，按照本节所演示的一些手段和程序代码，一定能分析出问题的原因的。例如有学员问我，他的 Java 程序从 oracle 数据库中读取到的中文显示为乱码，他通过 Java 程序写入 oracle 数据库的中文也为乱码，该怎么解决这个问题呢？虽然我以前没有碰到这样的问题，但我可以帮他分析问题的原因，首先用 oracle 自带的管理工具在数据库中插入“中国”这两个字符，然后用 Java 程序读出这个字符串，按前面所讲的方法打印出每个字节的内容，与我们这节程序打印出的结果对比，看看这个字符串是按什么编码生成的，最后将这个字符串转换成 gb2312 编码。oracle 数据库的中文写入问题，用相反的方式去做实验就可以解决了。

字节用于表示计算机内存中最原始的数据，不会涉及到编码问题，只有把字节中的内容当做字符来处理时，才会涉及编码问题，所以 InputStreamReader，InputStreamWriter，PrintStream，String 中都有一个可以指定字符集编码参数的构造函数，而 InputStream 的构造函数中则不存在这样的参数。

缺省的情况下，如果你构造与流相连的 Reader 和 Writer，字节和字符之间的转换规则使用缺省的平台字符编码和 Uni code，比如在英语国家字节码是用 ISO 8859-1，用户也可以指定编码格式，具体的格式参考 Sun 公司的 JDK 文档首页中的 Internationalization 超链接部分，在 JDK 文档首页中，你还能看到有关 java 的各种特性讲解的超链接，有空进去看看，一定会让你受益匪浅。如图 7.15 所示。

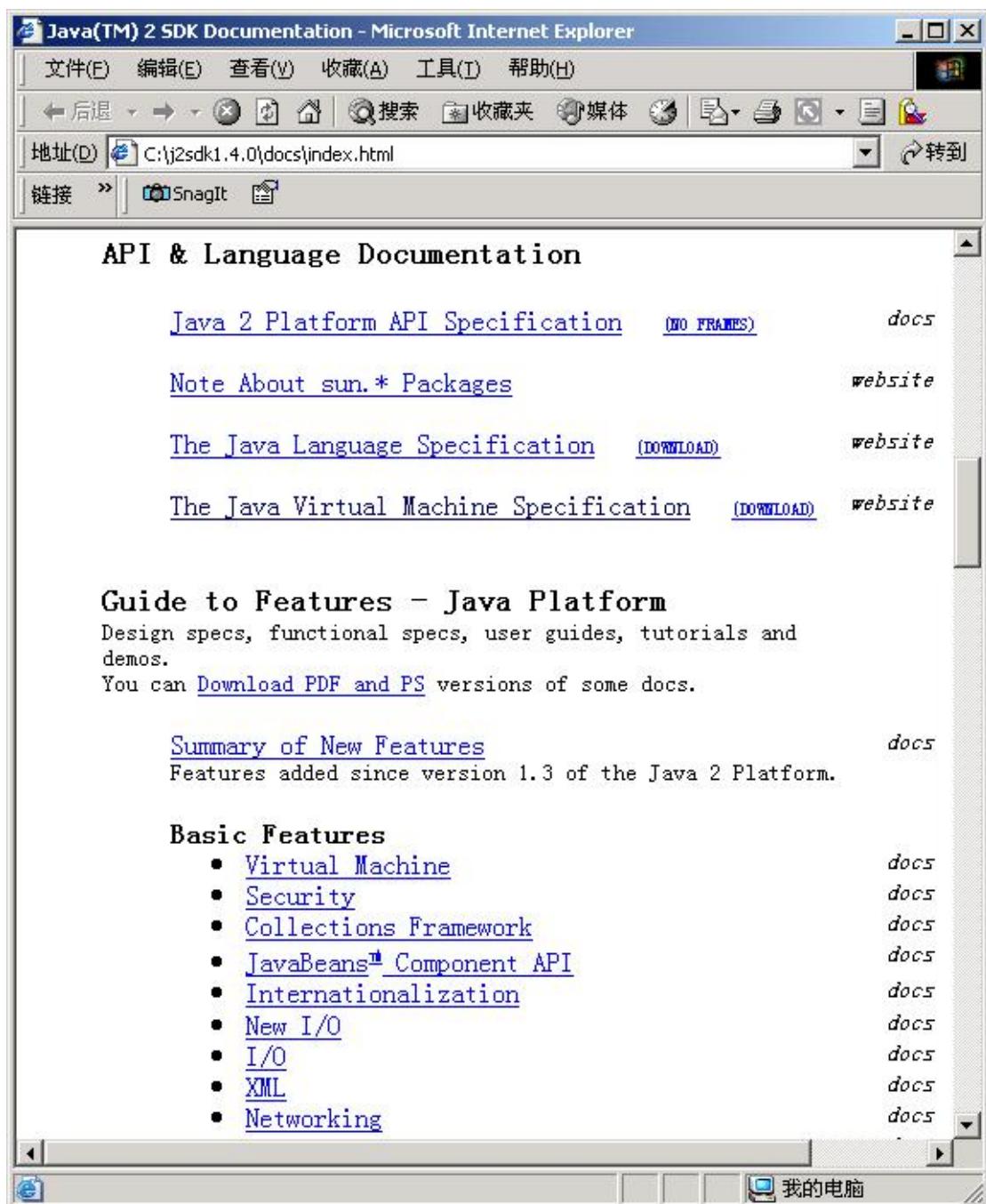


图 7.15

其中 Java 2 Platform API Specification 部分就是 JDK 中提供的各种类的帮助文档，Java 程序员在实际编码的过程中会经常来查阅这一部分，但作者现在使用 F. Alliant 整理成的 chm 格式的文档，也就是我们前面多次使用的那个帮助系统，会更方便，更有效。

我们再来看看为 InputStreamReader 指定其他字符集参数，也就是用 iso8859-1 替代缺省的 gb2312 字符集，分析观察程序的运行结果，来作为对本节讲解的结束和检查读者对本节内容的掌握情况。程序代码如下：

```
import java.io.*;
public class InputReader
{
    public static void main(String [] args) throws Exception
```

```

{
    InputStreamReader isr=
        new InputStreamReader(System.in, "iso8859-1");
    BufferedReader br = new BufferedReader(isr);
    String strLine=br.readLine();
    for(int i=0;i<strLine.length();i++)
    {

        System.out.println(Integer.toHexString((int)strLine.charAt(i)));
    }
    isr.close();
    System.out.println(strLine);
}
}

```

输入“中国”后，程序的运行结果如下：

中国

d6

d0

b9

fa

??ú

为了达到一种体验的效果，读者最好自己修改程序代码，让程序能够正常打印出输入中文字符。作者提前给出答案，照顾一下学得还不是很好的读者，读者可以用下面两种方式中的任意一种。

一. 将

```
InputStreamReader isr =new InputStreamReader(System.in,"iso8859-1");
```

修改成

```
InputStreamReader isr =new InputStreamReader(System.in,"gb2312");
```

在中国大陆的计算机系统上一般都是简体中文版的操作系统，缺省字符集为 gb2312，所以，我们也可以不指定字符集参数。

```
InputStreamReader isr =new InputStreamReader(System.in);
```

二. 不修改上面的部分，而是将

```
System.out.println(strLine);
```

修改成

```
System.out.println(new String(strLine.getBytes("iso8859-1"), "gb2312"));
```

广东地区的台商和港商的工厂较多，他们使用的系统的缺省字符集就不一定是 gb2312，使用这样系统的读者就不能照搬照套本节的内容了。

## 7.5.2 Decorator 设计模式

通过包装类，就可以用一个对象(the Decorators)包装另外的一个对象，比如：可以用 BufferedReader 来包装一个 FileReader，FileReader 仅仅提供了底层的读操作，比如

`read(char[] buffer)`。`BufferedReader` 实现了一个更高层次上的操作，比如 `readLine`，完成读取文件中的一个行。其实，这是一种被称为 `Decorator` 的设计模式。

我们要设计自己的 I/O 包装类，需要继承 `FilterXXX` 命名的类，从而扩展了对输入输出流的支持。比如我们设计一对类包装类：`RecordInputStream` 和 `RecordOutputStream`，来完成从数据库中读取记录和往数据库中写入记录。以后程序中就可以用它们来包装 `InputStream` 和 `OutputStream`，从而完成对数据库的操作。

包装类的使用非常灵活，我们来看看一个巧妙使用包装类的例子，从而借鉴一些思想。`Exception` 类从 `Throwable` 类继承的三个 `printStackTrace` 方法的定义如下：

```
public void printStackTrace()
public void printStackTrace(PrintStream s)
public void printStackTrace(PrintWriter s)
```

它们分别用把异常的详细信息打印到标准输出流（屏幕上），或者其他的 `PrintStream` 和 `PrintWriter` 流中。有时候，我们的应用需要把异常的详细信息放到一个字符串中，然后将这个包含异常详细信息的字符串通过网络发送出去，该怎么实现呢？我们先看看程序代码：

```
import java.io.*;
public class TestPrintWriter
{
    public static void main(String [] args)
    {
        try
        {
            throw new Exception("test");
        }
        catch(Exception e)
        {
            StringWriter sw=new StringWriter();
            e.printStackTrace(new PrintWriter(sw));
            String strException = sw.toString();
            System.out.println(strException);
        }
    }
}
```

在上面的程序中，用 `System.out.println` 将字符串打印在屏幕上简单模拟通过网络将这个字符串发送出去。`printStackTrace` 方法只能将异常详细信息写入一个 `PrintWriter` 对象中，写入 `PrintWriter` 对象的数据实际上会写入它所包装的一个 `Writer` 对象中，而写入 `StringWriter` 对象（一种 `Writer` 对象）的内容可以当作字符串取出来，所以用 `PrintWriter` 对象去包装一个 `StringWriter` 对象就可以解决我们的需求。解决问题的关键在于，我们如何能想到将这些流有机地串联起来。

### 7.5.3 Java 虚拟机读写其他进程的数据

我们在 Java 程序中可以产生其他的应用程序的进程，在 Java 程序中启动的进程称为子进程，启动子进程的 Java 程序称为父进程。子进程没有键盘和显示器，子进程的标准输入

和输出不再连接到键盘和显示器，而是以管道流的形式连接到父进程的一个输出流和输入流对象上，调用 Process 类的 `getOutputStream` 和 `getInputStream` 方法可以得到这个输出流和输入流对象。子进程从标准输入读取到的内容是父进程通过输出流对象写入管道的数据，子进程写入标准输出的数据通过管道传到了父进程的输入流对象中，父进程从这个输入流对象中读取到的内容就是子进程写入到标准输出的数据。

如 `javac.exe` 和 `java.exe` 这两个文件本身都是应用程序，我们在 Java 程序中也可以启动它们。请看下面的例子：

程序清单： `TestInOut.java`

```
import java.io.*;
public class TestInOut implements Runnable
{
    Process p=null;
    public TestInOut()
    {
        try
        {
            p=Runtime.getRuntime().exec("java MyTest");
            new Thread(this).start();
        }
        catch(Exception e)
        {
            System.out.println(e.getMessage());
        }
    }

    public void send()
    {
        try
        {

            OutputStream ops=p.getOutputStream();
            while(true)
            {
                ops.write("help\r\n".getBytes());
            }
        }
        catch(Exception e)
        {
            System.out.println(e.getMessage());
        }
    }
    public static void main(String [] args)
    {
        TestInOut tio=new TestInOut ();
    }
}
```

```

        tio.send();

    }

    public void run()
    {
        try
        {
            InputStream in = p.getInputStream();
            BufferedReader bfr=new BufferedReader(
                new InputStreamReader(in));
            while(true)
            {
                String strLine=bfr.readLine();
                System.out.println(strLine);
            }
        }
        catch(Exception e)
        {
            System.out.println(e.getMessage());
        }
    }
}

class MyTest
{
    public static void main(String [] args) throws IOException
    {
        while(true)
        {
            System.out.println("hi:"+
                new BufferedReader(new InputStreamReader(System.in)).readLine());
        }
    }
}

```

为了方便，我们将类 MyTest 与类 TestInout 放在同一个源文件中编译，运行后的结果如下：

```

.....
hi : hel p
hi : hel p
hi : l p
hi :
hi : hel p
hi : l p
hi :

```

```
hi : hel p
```

```
.....
```

可见，`TestInOut` 类中通过子进程的 `Process` 对象获得的输出和输入流对象，分别充当了 `MyTest` 类的“键盘”(`System.in`) 和“显示器”(`System.out`)。从运行的结果上，我们还看到由数据丢失的情况发生，这是因为管道是有一定大小的，这个大小其实就是 `PipedInputStream` 类中的缓冲区的大小，如果缓冲区满后，程序还没及时读取数据，就会发生数据丢失。我们对 `MyTest` 类的程序代码进行修改，提高其运行效率。

```
class MyTest
{
    public static void main(String [] args) throws IOException
    {
        BufferedReader bfr=new BufferedReader(
            new InputStreamReader(System.in));
        while(true)
        {
            System.out.println("hi :" +bfr.readLine());
        }
    }
}
```

重新编译运行后的结果如下：

```
hi : hel p
```

可见，在不同的地方定义变量和创建对象，用不同的代码顺序和结构，程序的运行效率大不一样。

## & 多学两招：

其实，我们在很多细微之处都可以提高程序的运行效率的，如

```
for(int i=0;i<str.length();i++){...}
```

的效率就不如改写成

```
int len=str.length();for(int i=0;i<len;i++){...}
```

的效率高，因为，上面的语句在每次循环时都要调用 `str.length()` 方法，而下面的语句只调用了一次。

我们在程序中还会经常碰到类似下面的应用，

```
byte [] buf= new byte[1024];while(true){对buf元素的操作语句}
```

```
while(true){ byte [] buf= new byte[1024]; 对buf元素的操作语句}
```

上面程序结构比下面的程序结构效率高，因为 `buf` 数组只被产生了一次。

在上面的程序中，我们没有考虑程序的退出方式，就只有用 `ctrl+c` 强制进程的结束这一招了。读者将这个程序多运行和退出几次，你会发现你的计算机慢如蜗牛了，查看 `Windows` 进程信息，如图 7.16 所示：

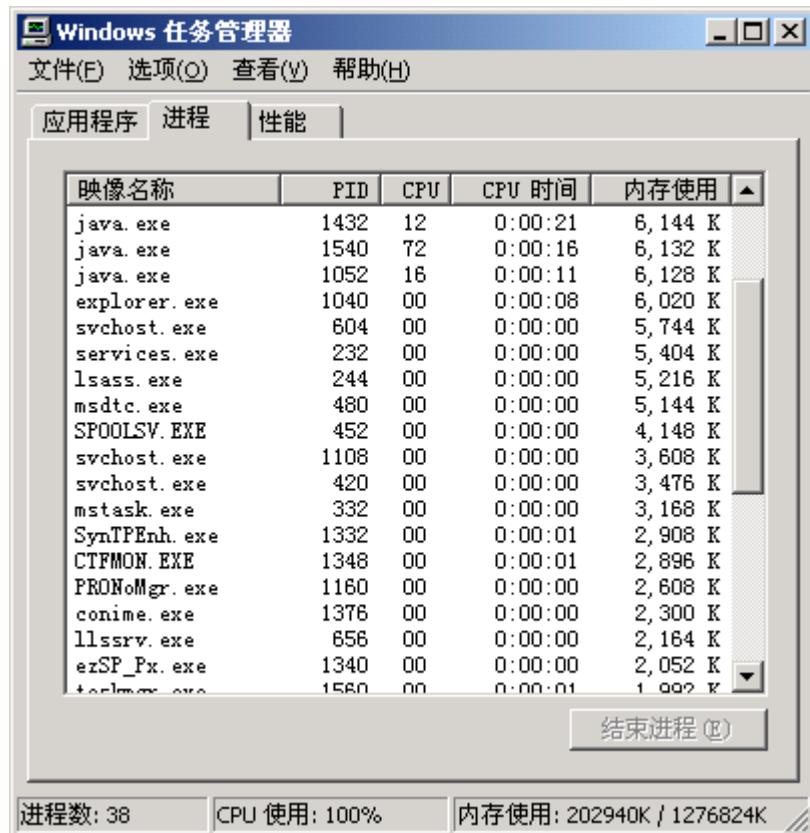


图 7.16

我们在进程列表中看到了多个 `java.exe` 程序，这说明我们每次启动的子进程仍在运行。在实际应用中，我们编写的程序应考虑退出方式，并调用 `Process` 类的 `destroy` 方法结束子进程的运行。

其实，我们所用的 `JCreator` 和 `JBUILDER` 这样的集成开发环境，都是调用 `JDK` 中的 `javac.exe` 和 `java.exe` 来编译和运行我们编写 `Java` 程序的，它们把 `javac.exe` 和 `java.exe` 做成了它们的一个子进程，并通过自己的图形界面中对子进程进行输入和显示其输出。

|  |            |
|--|------------|
| <b>第 7 章 IO/输入输出.....</b>                                | <b>203</b> |
| 7.1 File 类 .....   | 203        |
| 7.2 RandomAccessFile 类 .....                             | 205        |
| 7.3 节点流 .....  | 207        |
| 7.3.1 理解流的概念.....  | 207        |
| 7.3.2 InputStream 与 OutputStream.....                    | 208        |
| 指点迷津：1.如何选择输入与输出   |            |
| 2.为什么要调用 close 方法  |            |
| 多学两招：IO 中的缓冲区  |            |
| 7.3.3 FileInputStream 与 FileOutputStream .....           | 210        |
| 7.3.4 Reader 与 Writer .....                              | 211        |
| 独家见解：隐含的缓冲区  |            |
| 7.3.5 PipedInputStream 与 PipedOutputStream.....          | 212        |
| 独家见解：管道流类的作用   |            |
| 7.3.6 ByteArrayInputStream 与 ByteArrayOutputStream ..... | 214        |

|                      |   |     |
|----------------------|---|-----|
| 7.3.7                | IO 程序代码的复用 .....                                | 216 |
| 7.4                  | 过滤流与包装类.....                                    | 218 |
| 7.4.1                | 理解包装类的概念与作用.....                                | 218 |
| 7.4.2                | BufferedInputStream 与 BufferedOutputStream..... | 219 |
| 脚下留心：使用 mark 时应考虑的问题 |   |     |
| 7.4.3                | DataInputStream 与 DataOutputStream.....         | 219 |
| 7.4.4                | PrintStream.....                                | 222 |
| 指点迷津：何谓格式化输出         |   |     |
| 7.4.5                | ObjectInputStream 与 ObjectOutputStream .....    | 223 |
| 指点迷津：文件中的数据可读但不见得可用  |   |     |
| 7.4.6                | 字节流与字符流的转换.....                                 | 225 |
| 7.4.7                | IO 包中的类层次关系图 .....                              | 227 |
| 7.5                  | IO 中的高级应用 .....                                 | 228 |
| 7.5.1                | 字符集的编码问题.....                                   | 228 |
| 指点迷津：如何处理字符乱码问题      |   |     |
| 7.5.2                | Decorator 设计模式.....                             | 238 |
| 7.5.3                | Java 虚拟机读写其他进程的数据.....                          | 239 |
| 多学两招：提高程序的运行效率       |   |     |

# 第8章 GUI (上)

GUI 全称是 Graphical User Interface，即图形用户界面。顾名思义，就是应用程序提供给用户操作的图形界面，包括窗口、菜单、按钮、工具栏和其他各种屏幕元素。目前，图形用户界面已经成为一种趋势，它的好处自不必多说了，所以几乎所有的程序设计语言都提供了 GUI 设计功能。在 Java 里有两个包为 GUI 设计提供丰富的功能，它们是 AWT 和 Swing。AWT 是 Java 的早期版本，其中的 AWT 组件种类有限，可以提供基本的 GUI 设计工具，却无法完全实现目前 GUI 设计所需的所有功能。Swing 是 SUN 公司对早期版本的改进版本，它不仅包括 AWT 中具有的所有部件，并且提供了更加丰富的部件和功能，它足以完全实现 GUI 设计所需的一切功能。Swing 会用到 AWT 中的许多知识，掌握了 AWT，也就基本上掌握了 Swing，我们就从 AWT 开始我们的图形界面设计之旅吧！

## 8.1 初识 AWT

AWT 中定义了多种类和接口，用于在 Java 应用程序和 Java Applet 中进行 GUI 设计。我们首先通过下面的示例程序来感受一下 Java 的图形界面编程。

```
import java.awt.*;
public class TestFrame
{
    public static void main(String [] args)
    {
        Frame f=new Frame("IT 人资讯交流网");
        f.add(new Button("ok"));
        f.setSize(300,300);
        f.setVisible(true);
    }
}
```

图形界面程序中可以使用各种各样的图形界面元素，如文本框，按钮，列表框，对话框等等，我们将这些图形界面元素称为 GUI 组件。AWT 为各种 GUI 组件提供了对应的 Java 组件类，这些组件类都是 `java.awt.Component` 的直接或间接子类。其中，`Frame` 类用于产生一个具有标题栏的框架窗口。`Frame.setSize` 方法设置窗口的大小，`Frame.setVisible` 显示或隐藏窗口，程序运行后产生一个如图 8.1 所示的非常标准的框架窗口。用 AWT 编写 Java 的 GUI 程序的各种组件类都位于 JDK 的 `java.awt` 包中，程序开始必须导入 `java.awt` 包，可以导入整个 `java.awt` 包，也可以只导入程序中用到的那些组件类。编译运行此程序，结果如图 8.1 所示。



图 8.1

对于众多的 GUI 组件，根据其作用可以又分为两大类：基本组件（下面就全部简称为组件）和容器。

组件又被称为构件，它是诸如按钮、文本框之类的图形界面元素，在这些组件上不能容纳其他的组件。容器其实也是一种组件，是一种比较特殊的组件，它可以用来自容纳其他组件，如窗口，对话框等等，所有的容器类都是 `java.awt.Container` 的直接或间接子类。`Container` 类是 `Component` 类的一个子类，由此可见容器本身也具有组件的功能和特点，也可以被当作基本组件一样使用。在上面的程序中，`Frame` 就是一个容器，它容纳了一个 `Button` 部件。

## 8.2 AWT 线程

细心的读者也许注意到了，在运行上面写的那段程序时，主调用类的 `main` 方法执行 `f.setVisible(true)` 语句后就退出了，程序的 `main` 线程也随之结束了，但程序并没有结束，窗口不仅正常显示在桌面上，而且我们还可以对这些窗口进行一些常规操作，如拖动窗口，改变窗口的大小等。我们在多线程的课程中曾经讲过，对 Java 程序来说，只要还有一个前台线程在运行，整个进程就不会结束。这说明我们的程序还有其它线程在运行，那么其它线程是谁创建的？又是在什么时候创建的呢？读者可以简单地认为，程序在产生 `Frame` 对象时，创建了一个新的线程，我们称之为 AWT 线程。AWT 线程的内部实现，在不同的 JDK 版本下不太一样，从我们对 Java 的使用经验上来看，在不同的版本下，我们时常碰到我们的应用程序（不仅仅是 AWT）有不同的执行结果，这是令人很痛苦和无奈的事情，也是我们使用 Java 所要经常承担的风险。如果我们的程序调用 `Frame.dispose` 方法关闭了我们的框架窗口（具体实现细节，读者在本章稍后的部分能够看到），当程序放在 JDK1.3 下运行时，我们发现 AWT 线程没有结束，程序也不会自动结束。但在 JDK1.4 下运行这个程序，当框架窗口被关闭后，AWT 线程也结束了，程序随之结束。

搬出那些过时的小经验来给读者讲解，并不是什么好主意，甚至会造成误导。所以在这里，我的侧重点并不是要讲解与分析 AWT 线程在不同 JDK 版本下的差异的具体案例本身，因为百分之九十九的读者都不会再用到旧的 JDK 版本，也不会再碰到这样的问题的，既然人家在新的版本中已经做出了修改，你就没必要白费精力、花功夫去了解那些陈年旧事了。虽然

如此，但我还要在此提及这个问题，是要告诉大家下面的信息：在实际开发中，要注意到有时候碰到的一些莫名其妙的问题，并不是我们程序本身的问题，不妨换个角度去想，可能是开发工具或系统版本的问题，在通常的应用程序开发中，不要太相信你的系统是绝对的。

从事计算机软件开发，对绝大多数人来说，不是科学研究，而是一个工程项目的实施，我们使用的开发环境，开发工具，甚至编程语言本身就是我们的工程工具，这些工具不可能是完全理想和完美的，经常会有这样或那样的小问题，往往都是软件开发人员在应用过程中发现了它们在某个应用上的问题后，系统供应商再去修改他们的的开发环境，升级他们的开发工具，完善编程语言。在这之前，我们只能想别的办法来完成我们程序中的有关任务，或是通过别的手段来回避这些开发工具、环境、语言本身的问题。当新手碰到这些开发工具和语言本身的问题后，一般都不知所措，只会从程序的角度上去找错误，而不是怀疑开发工具或程序语言本身的问题。所以，在软件开发过程中，老手的经验是非常重要的，脱离实际应用，只学习编程语言本身的人们，往往都很难独当一面地从事软件开发。

## 8.3 AWT 事件处理

### 8.3.1 事件处理机制

我们在上面写的这个程序有一个明显的问题，鼠标点击窗口标题栏上的关闭按钮并不能够使窗口关闭和使程序结束，我们只能用操作系统自带的杀死进程的办法来关闭这个程序，在命令行窗口中用 `ctrl+c` 就可以强制结束正在运行的程序。其实，这也没有什么奇怪的，通过 GUI 组件，用户可以对应用程序进行各种操作，反之，应用程序可以通过 GUI 组件收集用户的操作信息，如用户在窗口上移动了鼠标、按下了键盘等。别忘了 GUI 的意义，是应用程序提供给用户操作的图形界面，而 GUI 本身并不对用户操作的结果负责。大家可以想想，我们在这个窗口上添加了一个按钮，当用户用鼠标点击这个按钮时，程序不也是什么都不做吗？如果我们想对鼠标点击按钮这个事件执行某种功能，就必须编写相应的处理程序代码。

对于这种 GUI 程序与用户操作的交互功能，Java 使用了一种自己专门的方式，称之为事件处理机制。在事件处理机制中，我们需要理解三个重要的概念：

- | 事件：用户对组件的一个操作，我们称之为一个事件。
- | 事件源：发生事件的组件就是事件源。
- | 事件处理器：负责处理事件的方法。

三者之间的关系如图 8.2 所示：

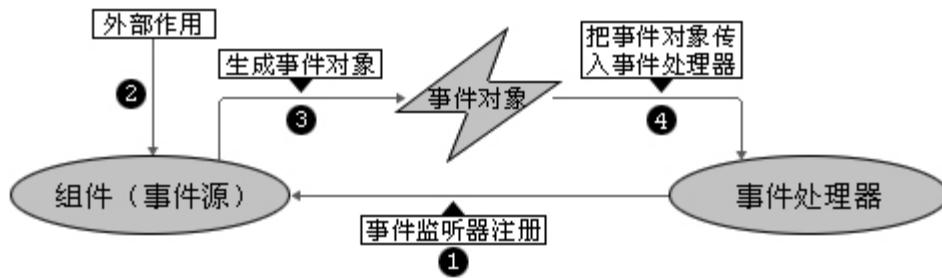


图 8.2

Java 程序对事件进行处理的方法是放在一个类对象中的，这个类对象就是事件监听器。

我们必须将一个事件监听器对象同某个事件源的某种事件进行关联，这样，当某个事件源上发生了某种事件后，关联的事件监听器对象中的有关代码才会被执行。我们把这个关联的过程称为向事件源注册事件监听器对象。从上面的图例中，我们能够看到，事件处理器（事

件监听器)首先与组件(事件源)建立关联,当组件接受外部作用(事件)时,组件就会产生一个相应的事件对象,并把此对象传给与之关联的事件处理器,事件处理器就会被启动并执行相关的代码来处理该事件。

基本上明白了 Java 的事件处理机制,我们接着详细介绍事件和事件监听器的一些编程方面的有关知识。

事件用以描述发生了什么事情。AWT 对各种不同的事件,按事件的动作(如鼠标操作)、效果(如窗口的关闭和激活)等进行了分类,一类事件对应一个 AWT 事件类。我们这里并不想象通常的书籍一样,给大家罗列各种各样的事件并解释一番。我在这里为大家简要介绍几个具有典型代表意义的事件,就足以让大家掌握相关的知识了,如果有人想了解所有的事件,不用去查找什么大全之类的书籍,在这里我告诉你一个简单的办法,你只要参阅 JDK 文档中的 `java.awt.event` 包,那里列出了所有的事件类。

- | MouseEvent 类对应鼠标事件,包括鼠标按下,鼠标释放,鼠标点击(按下后释放)等。
- | WindowEvent 类对应窗口事件,包括用户点击了关闭按钮,窗口得到与失去焦点,窗口被最小化等。
- | ActionEvent 类对应一个动作事件,它不是代表一个具体的动作,而是一种语义,如按钮或菜单被鼠标单击,单行文本框中按下回车键等都可以看作是 ActionEvent 事件。读者可以这么理解 ActionEvent 事件,如果用户的一个动作导致了某个组件本身最基本的作用发生了,这就是 ActionEvent 事件。菜单、按钮放在那就是用来发出某种动作或命令的,鼠标单击(也可以用键盘来操作)这些组件,只是表示要执行这种动作或命令的事情发生了,显然对于这种情况,我们并不关心是鼠标单击,还是键盘按下的。

通过各种事件类提供的方法,我们可以获得事件源对象,以及程序中对这一事件可能要了解的一些特殊信息,如对于鼠标事件,我们很可能要获得鼠标的坐标信息,经过查 JDK 文档就能知道用 `MouseEvent.getX`, `MouseEvent.getY` 这两个方法。小时候常听人讲“人有多大胆,地有多大产”,同样,对于我们编程中遇到的一般正常的需求,开发工具包都会提供,没有解决不了的,只有我们还没找到的。解决问题的关键就看我们有没有现用现找的本领了。

某一类事件,其中又包含触发这一事件的若干具体情况。对一类事件的处理由一个事件监听器对象来完成,对于触发这一事件的每一种情况,都对应着事件监听器对象中的一个不同的方法。某种事件监听器对象中的每个方法名称必须是固定的,事件源才能依据事件的具体发生情况找到事件监听器对象中对应的方法,事件监听器对象也包含事件源可能调用到的所有事件处理方法。这正是“调用者和被调用者必须共同遵守某一限定,调用者按照这个限定进行方法调用,被调用者按照这个限定进行方法实现”的应用规则,在面向对象的编程语言中,这种限定就是通过接口类来表示的。事件源和事件监听器对象就是通过事件监听器接口进行约定的,事件监听器对象就是实现了事件监听器接口的类对象。不同的事件类型对应不同的事件监听器接口。

事件监听器接口的名称与事件的名称是相对应的,非常容易记忆,如 `MouseEvent` 的监听器接口名为 `MouseListener`, `WindowEvent` 的监听器接口名为 `WindowListener`, `ActionEvent` 的监听器接口名为 `ActionListener`。

有许多书上将众多的事件分为两大类:低级事件和语义事件(又叫高级事件),并列出了所有属于低级事件的事件,所有属于高级事件的事件。凭作者的经验,不相信真有读者能够很清楚的记住哪些是高级事件和哪些是低级事件,包括那些书的作者们本人也做不到。虽然作者认为,记住哪些是高级事件和哪些是低级事件毫无意义,但作者也可以让你不用记忆,

就能够轻松地进行这种区分。如果某个事件的监听器接口中只有一个方法，那么这个事件就是语义事件，如 `ActionListener` 中只有一个方法，`ActionEvent` 就是一种语义事件，反之，则为低级事件。另外，从字面上，我们也能够想象，语义事件关心的是一个具有特殊作用的 GUI 组件对应的动作发生了，而不关心这个动作是怎样发生的。

### 8.3.2 用事件监听器处理事件

我们来看看如何为上面的程序添加窗口关闭的代码，从而学习事件处理的具体编码实现：

程序清单：`TestFrame.java`

```
import java.awt.*;
import java.awt.event.*;
public class TestFrame
{
    public static void main(String [] args)
    {
        Frame f=new Frame("IT人资讯交流网");
        f.setSize(300,300);
        f.setVisible(true);
        f.addWindowListener(new MyWindowListener());
    }
}
class MyWindowListener implements WindowListener
{
    public void windowClosing(WindowEvent e)
    {
        e.getWindow().setVisible(false);
        ((Window)e.getComponent()).dispose();
        System.exit(0);
    }
    public void windowActivated(WindowEvent e){}
    public void windowClosed(WindowEvent e){}
    public void windowDeactivated(WindowEvent e){}
    public void windowDeiconified(WindowEvent e){}
    public void windowIconified(WindowEvent e){}
    public void windowOpened(WindowEvent e){}
}
```

在上面的程序代码中，由于 AWT 中的事件类和监听器接口类都位于 `java.awt.event` 包中，所以在程序的开始处，`import` 了 `java.awt.event.*`。注意，`import` 一个包中的所有类，并没有 `import` 该包的子包中的类，所以，我们 `import` 了 `java.awt` 包，还要单独 `import` `java.awt.event` 包，其实，这个问题，即使我们不讲，大家也应该能够自己通过实验弄明白的。只要勇于实践，多动手编写些程序，通过观察编译运行的结果，就可以验证我们的许多想法或去除心中的疑惑，同时也锻炼和逐渐提高了我们的编程能力。

我们编写了一个新类 `MyWindowListener` 来实现窗口事件监听器对象的程序代码，并调用 `Window.addWindowListener` 方法将事件监听器对象注册到 `Frame` 类（`Frame` 类继承

`java.awt.Window` 类) 创建的框架窗口上。在 `WindowListener` 接口里有七个方法, 正好对应窗口事件的七种情况, 因为我们只想处理鼠标点击窗口标题栏上的关闭按钮这一事件, 对其他窗口事件我们并不关心, 所以, 在类 `MyWindowListener` 的代码中, 我们只对 `windowClosing` 方法进行了编码, 其他方法只是简单实现 (因为 Java 语法的要求, 必须实现接口的所有方法), 什么也没做。注意 `windowClosing` 方法和 `windowClosed` 方法的区别, `windowClosing` 对应用户想关闭窗口的情况, 而 `windowClosed` 对应窗口已经被关闭时的情况。

## F 指点迷津:

如我们要获得汽车类的发动机, 我们可以用一个汽车类的 `getEngine` 方法来实现, 显然该方法的返回值类型只能是发动机 (而不能是发动机的某个子类)。现在, 有个东风汽车类继承了汽车类, 如果我们直接使用从汽车类继承到的 `getEngine` 方法来获得一辆东风汽车的发动机, 尽管我们知道获得的发动机是东风发动机, 但对编译器来说, 它只能从 `getEngine` 方法的语法上知道返回的是发动机, 如果对于这个返回的发动机对象, 我们要使用东风发动机特有的功能, 我们就需要使用以前讲过的类型转换。如果东风汽车类专门增加了一个新的 `getDongFengEngine` 方法, 这个方法的返回值类型就是东风发动机, 返回的对象可以直接使用东风发动机特有的功能, 不用再作类型转换。

明白了东风汽车这个比喻, 读者就不难理解在上面程序的 `windowClosing` 方法中, 作者用了两种方式来返回那个窗口对象的代码了。下面是 `WindowEvent` 的继承层次图:

```
java.lang.Object
|
+--java.util.EventObject
|
+--java.awt.AWTEvent
|
+--java.awt.event.ComponentEvent
|
+--java.awt.event.WindowEvent
```

由于 `EventObject.getSource`、`ComponentEvent.getComponent`、`WindowEvent.getWindow` 等方法都可以返回事件源对象, 由于子类可以继承父类的方法, 所以 `WindowEvent` 事件对象, 返回事件源对象的方法可以有多个。不管用哪种方法, 返回的都是内存中的同一个对象, 只是对编译器来说, 返回的类型不同罢了, 越是底层的 `getXxx` 方法, 返回的类型越具体。

**小结:** 要处理 GUI 组件上的 `XxxEvent` 事件下的某种情况, 首先要编写一个实现了 `XxxListener` 接口的事件监听器类, 然后在 `XxxListener` 类和要处理的具体事件情况相对应的方法中编写处理程序代码, 最后将类 `XxxListener` 创建的对象通过 `addXxxListener` 方法注册到 GUI 组件上。`Xxx` 可以是各种不同的事件, 如 `Window`, `Mouse`, `Key`, `Action` 等。

**小经验:** 所有事件监听器方法返回的返回类型都是 `void`。

### 8.3.3 事件适配器

为简化编程, JDK 针对大多数事件监听器接口定义了相应的实现类, 我们称之为事件适配器 (Adapter) 类。在适配器类中, 实现了相应监听器接口中所有的方法, 但不做任何事情, 子类只要继承适配器类, 就等于实现了相应的监听器接口, 如果要对某类事件的某种情况进行处理, 只要覆盖相应的方法就可以了, 其他的方法再也不用“简单实现”了。可见,

如果想用作事件监听器的类已经继承了别的类，就不能再继承适配器类了，只能去实现事件监听器接口了。

修改 MyWindowListener，代码如下：

```
class MyWindowListener extends WindowAdapter
{
    public void windowClosing(WindowEvent e)
    {
        e.getWindow().setVisible(false);
        ((Window)e.getComponent()).dispose();
        System.exit(0);
    }
}
```

重新编译没有错误，可是，点击关闭按钮后程序并没有退出。这是怎么回事呢？要么是 WindowClosing 方法中的代码有问题，要么是 WindowClosing 方法没有调用，怎么判定这两种情况呢？我们只要在 WindowClosing 方法的开始处，添加 System.out.println("coming here!");，编译运行后再看看命令行窗口上（不是 GUI 窗口上）有没有打出“coming here!”，就知道 WindowClosing 方法是否被调用过。运行后屏幕上并没有打印出“coming here!”，可以确定 WindowClosing 方法没有被调用，这时又有两种可能性，一种是还没有将 MyWindowListener 对象注册到框架窗口上，所以即使框架窗口上发生了 WindowEvent，也不会来调用类 MyWindowListener 中的任何方法，仔细检查程序代码，我们可以排除这种可能。还有一种可能就是，WindowClosing 方法名拼写有问题，仔细比较后发现第一个字母应该是小写，即 windowClosing。哦！Java 是区分大小写的，MyWindowListener.WindowClosing 是一个与 WindowAdapter.windowClosing 毫无关系的方法，并没有起到对 WindowAdapter.windowClosing 方法覆盖的目的。将 WindowClosing 修改成 windowClosing 后编译运行，一切正常。

在上面的例子中，事件监听器的类 MyWindowListener 和产生 GUI 组件的类 TestFrame 是两个完全分开的类，事件监听器的类中的代码所访问到的对象也正好是事件源。如果我们要在件监听器的类中的代码中访问其他的非事件源的 GUI 组件，程序就得想别的办法了，如我们单击图 8.1 窗口中的 ok 按钮来关闭框架窗口并退出程序，这就要求程序在按钮的事件监听器代码中要访问框架窗口，我们又该怎么做呢？一种简单的办法就是将事件监听器的代码和产生 GUI 组件的代码放在同一个类中实现，程序代码如下：

```
import java.awt.*;
import java.awt.event.*;
public class TestFrame implements ActionListener
{
    public static void main(String [] args)
    {
        Frame f=new Frame("IT 人资讯交流网");
        Button btn=new Button("退出");
        btn.addActionListener(new TestFrame());//注册事件监听器对象
        f.add(btn); //将按钮增加到框架窗口上
        f.setSize(300,300);
        f.setVisible(true);
    }
}
```

```

    public void actionPerformed(ActionEvent e)
    {
        f.setVisible(false);
        f.dispose();
        System.exit(0);
    }
}

```

编译肯定有错，因为 `actionPerformed` 方法中不能访问在 `main` 方法中定义的 `f` 变量。修改程序，将 `Frame f=new Frame("IT 人资讯交流网");` 移动到 `main` 方法的上面，也就是将 `f` 定义成一个成员变量，编译还会有错，因为 `main` 是个静态方法，不能直接访问同类中的成员变量。再次修改程序，如下所示：

```

import java.awt.*;
import java.awt.event.*;
public class TestFrame implements ActionListener
{
    Frame f=new Frame("IT 人资讯交流网");
    public static void main(String [] args)
    {
        TestFrame tf=new TestFrame();
        Button btn=new Button("退出");
        btn.addActionListener(new TestFrame());
        tf.f.add(btn);
        tf.f.setSize(300,300);
        tf.f.setVisible(true);
    }
    public void actionPerformed(ActionEvent e)
    {
        f.setVisible(false);
        f.dispose();
        System.exit(0);
    }
}

```

上面的程序虽然编译没有问题，但程序代码规范极差，在 `main` 方法中，对于每一次 `f` 对象的引用，都必须在 `f` 前增加 `tf`，如 `tf.f.setSize`。如果添加到 `f` 上的 GUI 组件很多，那么将有很多的 `tf.f.xxx` 语句，这样的程序代码，相信你自己都不会满意。我们怎样将上面的代码修改得规范些呢？将程序改成下面这样。

```

import java.awt.*;
import java.awt.event.*;
public class TestFrame implements ActionListener
{
    Frame f=new Frame("IT 人资讯交流网");
    public static void main(String [] args)
    {
        TestFrame tf=new TestFrame();

```

```

        tf.init();
    }
    public void init()
    {
        Button btn=new Button("退出");
        btn.addActionListener(new TestFrame());
        f.add(btn);
        f.setSize(300,300);
        f.setVisible(true);
    }
    public void actionPerformed(ActionEvent e)
    {
        f.setVisible(false);
        f.dispose();
        System.exit(0);
    }
}

```

将在 `main` 方法中调用的 GUI 初始化代码全部放到了 `init` 方法中，在 `main` 方法中只要调用这个 `init` 方法就可以了，`init` 方法是非静态的，可以直接访问非静态的成员变量。编译运行后，退出按钮并不能关闭窗口，读者应先自己思考问题的原因并上机调试后，再来参考下面的解答。在上面的程序中，创建了两个 `TestFrame` 对象，桌面上显示的窗口是第一个 `TestFrame` 中的 `f` 对象，而退出按钮的事件处理代码中关闭的是第二个 `TestFrame` 中的 `f` 对象，所以我们看到的窗口没有关闭，将 `init` 方法中的 `btn.addActionListener(new TestFrame())` 改为 `btn.addActionListener(this)` 就可以了。

我们在这里再次强调要面向对象，脑子里想的就是程序中各对象在各个时刻的内存布局和状态，以及每行程序代码执行后对内存中的对象的影响。

我们通过将事件监听器的代码和产生 GUI 组件的代码放在同一个类中实现，就很轻松的解决了我们的问题。

### 8.3.4 事件监听器的匿名内置类实现方式

如果一个事件监听器类只用于在一个组件上注册监听器事件对象，为了让程序代码更为紧凑，我们可以用匿名内置类的语法来产生这个事件监听器对象，这也是一种经常使用的方法。

对于上面的例子，我们用匿名内置类的语法进行改写，程序代码如下：

```

import java.awt.*;
import java.awt.event.*;
class TestFrame
{
    Frame f=new Frame("IT人资讯交流网");
    public static void main(String [] args)
    {
        new TestFrame().init();
    }
    public void init()

```

```

{
    Button btn=new Button("退出");
    btn.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            f.setVisible(false);
            f.dispose();
            System.exit(0);
        }
    });
    f.add(btn);
    f.setSize(300,300);
    f.setVisible(true);
}
}

```

### 8.3.5 事件处理的多重运用

用户的一个操作，在触发了低级事件的同时，可能也会触发语义事件，这时令初学者困惑的是，不知道该选择哪种类型事件监听器来进行处理。其实道理很简单，一个员工打伤了他的同事，如果程序代表法律部门，要处理的就是打人这一低级事件，如果程序代表公司经理，要处理的便是违犯公司纪律这一语义事件，如果程序即代表法律部门，又代表公司经理，便需对这两个事件都做处理。一般情况下，如果对语义事件的处理能够满足我们的需求，我们就不再处理低级事件。用户在一个按钮上按下鼠标，触发了鼠标事件，也触发了按钮的动作事件，我们根据我们程序的需要来编码处理某一个事件或是两者都处理。例如当按钮被点击后，我们除了要执行该按钮对应的程序功能外，还希望鼠标按下时能改变按钮标题内容，鼠标释放时能恢复标题内容，我们在按钮上就要注册两个事件监听器对象，一个处理鼠标事件，另一个处理按钮的动作事件。一个 GUI 组件到底能够触发哪几种事件，我们没必要死记硬背，在一般的集成开发环境下，如 JBuilder，JCreator 等，当我们输入某个对象的成员分隔符，在下拉的成员列表提示框中，我们就能看到这个组件支持的事件，如图 8.3 所示的 JCreator 下的提示情况。

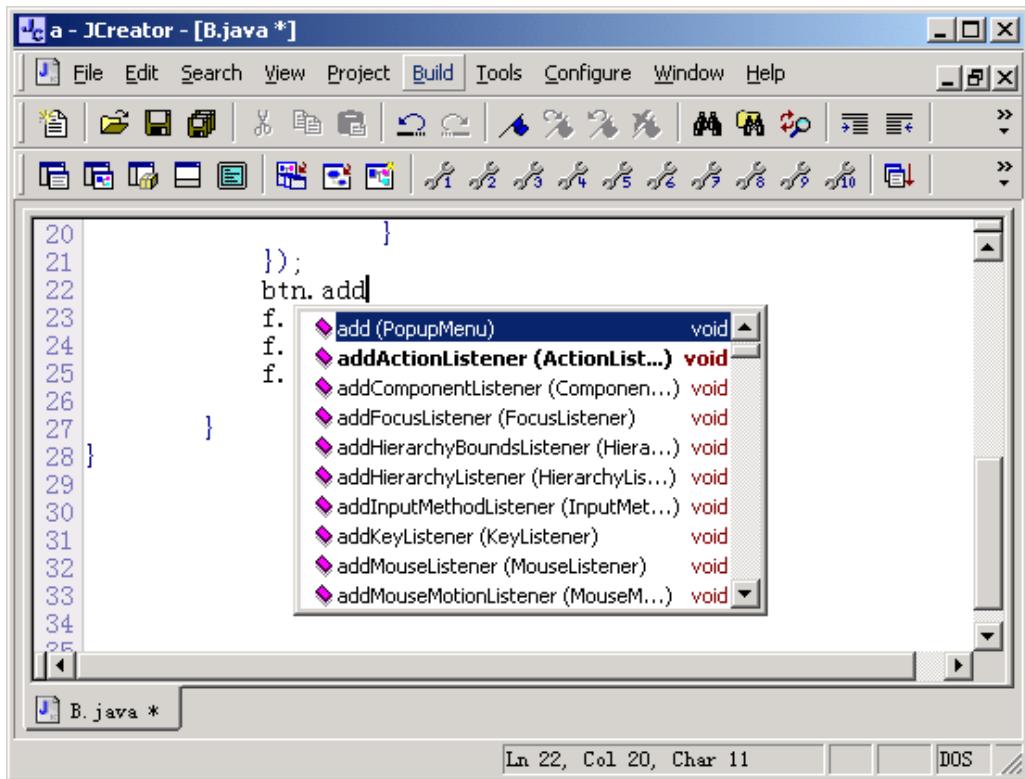


图 8.3

我们通过查看这些 `addXxxListener` 方法，就知道了这个组件所支持的事件类型。一个组件上的一个动作可以产生多种不同类型的事件，因而我们可以向同一个事件源上注册多种不同类型的监听器，如图 8.4 所示：

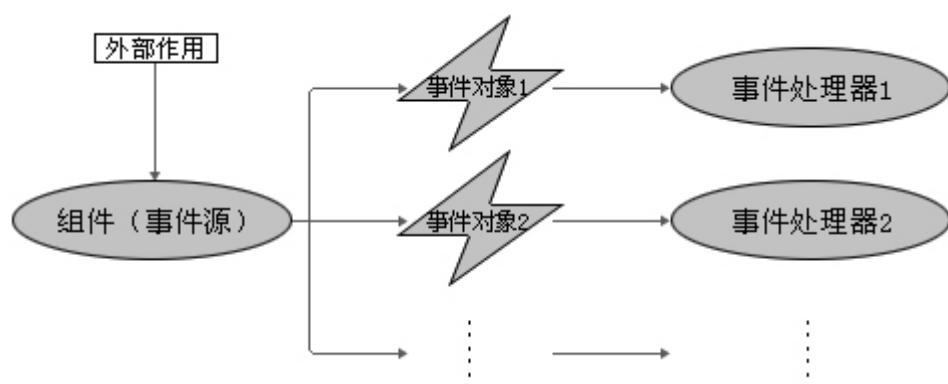


图 8.4

一个事件监听器对象可以注册到多个事件源上，即多个事件源的同一事件都由一个对象统一来处理，如图 8.5 所示：

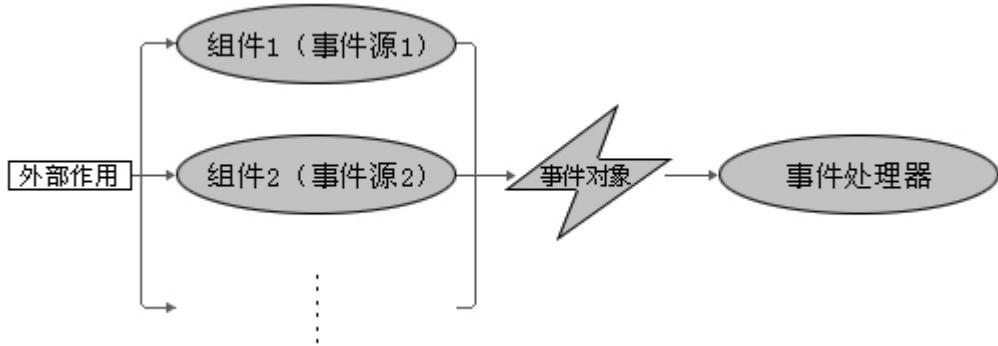


图 8.5

一个事件源上也可以注册对同一事件进行处理的多个事件监听器对象，当这一事件发生时，各事件监听器对象依次被调用，如图 8.6 所示：

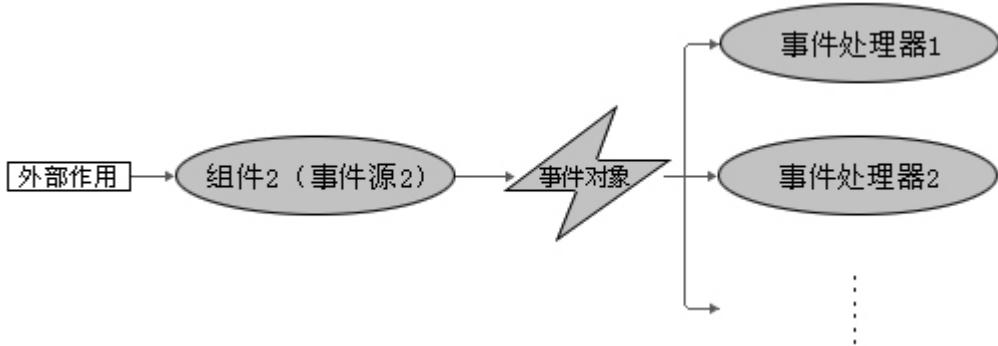


图 8.6

### 8.3.6 高级事件处理

默认情况下，组件屏蔽了对所有事件的响应，也就是不管发生了什么情况，事件都不会在这个组件上发生，组件都不会产生任何事件对象。只有在一个组件上注册某种事件的事件监听器对象后，组件才可以对这种事件做出响应，当发生了对应这种事件的情况后，事件才能在这个组件上发生，组件才会产生这种事件对象。

当一个组件上发生了某种事件后，系统会调用这个组件对象的 `processEvent` 方法来处理，缺省的 `processEvent` 方法将根据事件的类型调用相应的 `processXxxEvent` 方法，其中 `Xxx` 代表事件类型，`processXxxEvent` 方法接着将 `Xxx` 事件传递给注册的监听器去处理。例如，如果组件上发生了鼠标移动事件，组件对象的 `processEvent` 方法将调用 `processMouseEvent` 方法进行处理。

如果我们想改变某种组件的事件处理方式，我们需要覆盖该组件的 `processEvent` 方法或 `processXxxEvent` 方法，`processEvent` 是处理所有事件的总入口，而 `processXxxEvent` 是专用于处理某种事件的分岔入口。显然，如果要在一个方法中就改变所有事件的处理方式，我们需要覆盖 `processEvent`，如果只想改变某种或少数几种事件的处理方式，而不影响其它事件的处理方式，我们还是覆盖 `processXxxEvent` 方法，而不再直接覆盖 `processEvent` 方法为好。由于我们不可能直接进入到某个组件的 `processXxxEvent` 方法中去修改程序代码，我们需要定义一个继承了该组件的子类，在子类中覆盖 `processXxxEvent` 方法，并将原先创建的组件对象改为由这个子类创建，就可以达到我们的目的了。

就像前面讲的，如果我们没有在组件上注册 `Xxx` 事件监听器，组件就不会发生 `Xxx` 事件，我们的 `processXxxEvent` 方法根本就不可能被调用。即使没有在组件上注册事件监听器，我

们只要调用了 enableEvents 函数，设置组件能够进行响应的事件，在相应的情况发生后，组件仍然能够产生对应的事件。enableEvents 函数的定义如下：

```
protected final void enableEvents(long eventsToEnable)
```

其中，参数 eventsToEnable 指定了需要组件响应的事件类型所对应的数值。在我们的程序中经常要用到一类变量，这个变量里的每一位(bit)都对应某一种特性。当该变量的某位为 1 时，表示有该位对应的那种特性，当该位为 0 时，即没有那位所对应的特性。当变量中的某几位同时为 1 时，就表示同时具有那几种特性的组合。那种特性在我们这里就是要响应的事件类型，我们将不同的事件类型用一个不同的 long 型数值来表示，且每个这些数值中只有与其事件类型对应的那一位（一个 bit）为 1，其余的 bit 都为 0，这样，我们就可以对多个事件类型所对应的数值进行相或(|)操作，得到的结果正好就是这几种事件类型的组合。一个数值中的哪一位代表哪种事件类型，是不容易记忆的，所以我们就将这些表示不同事件类型的 long 型数值定义成常量，常量名就是根据事件类型的英文拼写去定义。可见，参数 eventsToEnable 可以是多个事件类型所对应的数值相或的结果，我们又怎样找到表示事件类型数值的常量名呢？如果你明白了在 Java 中是如何定义常量的，你就应该想到在 JDK 文档中去查找与事件有关的类，在其中的某个类中就会有这些常量的定义的。作者看到 processEvent((AWTEvent e)) 方法中的参数类型是 AWTEvent，就很自然地想到了去 AWTEvent 类中查找，果然在那里看到了这些常量的定义，如鼠标移动事件对应的常量为 MOUSE\_MOTION\_EVENT\_MASK，这样，当我们想让组件响应鼠标移动事件时，我们可以使用 enableEvents(AWTEvent.MOUSE\_MOTION\_EVENT\_MASK); 语句来完成。

明白了组件内部的事件处理过程和相关知识，我们现在就来编写一个这样的程序：在一个窗口上显示一个按钮，一旦鼠标移动到这个按钮上时，按钮就移动到了其他位置，这样，鼠标就永远无法点击到这个按钮。在写这个程序之前，我先讲个故事，我们假设有两个孙悟空，刚开始时第二个孙悟空使用了隐身术，所以你只能看见第一个孙悟空。当你靠近第一个孙悟空时，这个孙悟空马上使用了隐身术而隐藏，同时通知第二个孙悟空现身，你就会误以为第一个孙悟空跑到了第二个孙悟空的位置上。同样，当你靠近第二个孙悟空时，这个孙悟空马上使用了隐身术而隐藏，同时通知第一个孙悟空现身，这时，你以为还是第一个孙悟空又跑回到了他原来的位置。如此往复，你永远也抓不到孙悟空的，你却还以为只有一个孙悟空在跑来跑去的呢！作者就使用抓孙悟空的这个思路来编写我们的程序，首先要对按钮的鼠标移动事件进行处理，所以我定义了一个 Button 的子类 MyButton，在 MyButton 中对 processMouseEvent 方法进行覆盖，在该方法中隐藏自己，显示自己的伙伴。下面是程序代码，其中的细节在前面都分析过了，就不再多说了。

```
程序清单：TestMyButton.java
import java.awt.*;
import java.awt.event.*;
class MyButton extends Button
{
    private MyButton friend;
    public void setFriend(MyButton friend)
    {
        this.friend = friend;
    }
    public MyButton(String name)
    {
        super(name);
```

```

        enableEvents(AWTEvent.MOUSE_MOTION_EVENT_MASK);
    }
    protected void processMouseEvent(MouseEvent e)
    {
        setVisible(false);
        friend.setVisible(true);
    }
}
public class TestMyButton
{
    public static void main(String [] args)
    {
        MyButton btn1 =new MyButton("你来抓我呀!");
        MyButton btn2 =new MyButton("你来抓我呀!");
        btn1.setFriend(btn2);
        btn2.setFriend(btn1);
        btn1.setVisible(false);
        Frame f =new Frame("it315");
        f.add(btn1, "North");//将 btn1 增加到 f 的北部
        f.add(btn2, "South");//将 btn2 增加到 f 的南部
        f.setSize(300,300);
        f.setVisible(true);
        btn1.setVisible(false);
    }
}

```

## 8.4 GUI 组件上的图形操作

### 8.4.1 Graphics 类

我们有时需要在 GUI 组件上绘制图形，打印文字，显示图像等操作。组件对象本身不提供这些操作的方法，它只提供一个 `getGraphics` 方法，`getGraphics` 方法返回一个包含有该组件的屏幕显示外观信息的 `Graphics` 类对象，`Graphics` 类提供了在组件显示表面绘画图形，打印文字，显示图像等操作方法。

下面，我们通过编写一个画线程序来讲解在 GUI 组件上的图形操作，同时帮助大家更好地理解和掌握 AWT 事件处理机制。画线用的是 `Graphics.drawLine(int x1, int y1, int x2, int y2)` 方法，其中的参数意义，读者应该能够猜想得到，如果连这点想象力都没有，以后没法面对实际项目的开发。大家在使用陌生的方法时，可以反过来想，如果这个方法是你写的，你会提供给人家什么样的参数？这样一来，就基本上明白了这个方法的用法了。尽管我们有时在不得已的情况下，还是需要去仔细参看新方法的使用文档的，但经常多思考一下，我们能够潜移默化的从中体验出一些心得，借鉴到工具开发商的一些好的编码经验和思想。我们的程序要实现这样的功能：鼠标按下时的位置作为线的起始点，鼠标释放时的位置作为终止点，并在鼠标释放时画线，所以，我们需要对鼠标事件进行处理，在鼠标按下时记住鼠

标的坐标，鼠标释放时画线。程序代码如下：

```
程序清单：DrawLine.java
import java.awt.*;
import java.awt.event.*;
public class DrawLine
{
    Frame f= new Frame("IT 人资讯交流网");
    public static void main(String [] args)
    {
        new DrawLine().init();
    }
    public void init()
    {
        f.setSize(300,300);
        f.setVisible(true);
        f.addMouseListener(new MouseAdapter()
        {
            int orgX;
            int orgY;
            public void mousePressed(MouseEvent e)
            {
                orgX=e.getX();
                orgY=e.getY();
            }
            public void mouseReleased(MouseEvent e)
            {
                f.getGraphics().setColor(Color.red);
                //设置绘图颜色为红色
                f.getGraphics().drawLine(orgX,orgY,e.getX(),e.getY());
            }
        });
    }
}
```

在上面的程序代码中，读者也必须能够做到通过查阅 JDK 文档看懂 `f.getGraphics().setColor(Color.red);` 这句代码为什么是这么写，特别是 `Color.red` 部分。编译运行，线条被画出来了，但颜色却不是红色。这是一个非常隐蔽的问题，程序中的两处都用 `f.getGraphics()` 返回 `Graphics` 对象引用，返回的两个引用指向的并不是同一个 `Graphics` 对象，而是两个完全不同的对象，设置一个 `Graphics` 对象上的绘图颜色，不会影响另一个 `Graphics` 对象上的绘图输出。

我们接着为这个程序添加文本打印功能，顺便修正上面碰到的问题。

程序清单：DrawLine2.java

```
import java.awt.*;
import java.awt.event.*;
public class DrawLine2
```

```

{
    Frame f= new Frame("IT 人资讯交流网");
    public static void main(String [] args)
    {
        new DrawLine2().init();
    }
    public void init()
    {
        f.setSize(300,300);
        f.setVisible(true);
        f.addMouseListener(new MouseAdapter()
        {
            int orgX;
            int orgY;
            public void mousePressed(MouseEvent e)
            {
                orgX=e.getX();
                orgY=e.getY();
            }
            public void mouseReleased(MouseEvent e)
            {
                Graphics g=f.getGraphics();
                g.setColor(Color.red);//设置绘图颜色为红色
                g.setFont(new Font("隶书",Font.ITALIC|Font.BOLD,30));
                //设置文本的字体
                g.drawString(new String(orgX +", " +orgY),orgX,orgY);
                //打印鼠标按下时的坐标文本
                g.drawString(new String(e.getX() +", " +e.getY()))
                    ,e.getX(),e.getY());
                //打印鼠标释放时的坐标文本
                g.drawLine(orgX,orgY,e.getX(),e.getY());
            }
        });
    }
}

```

其实，上面的代码是作者一边查看 JDK 文档，一边现炒现卖的。在实际开发中，我们不可能以前就接触和掌握了程序中碰到的每个细节问题，这就需要我们在编程过程中遇到小的需求和问题时，马上能去查阅文档资料来解决，程序员就是这样工作的。上面的代码中，我们只调用了一次 `f.getGraphics()` 方法返回了一个 `Graphics` 对象，以后调用的全部是这个 `Graphics` 对象上的方法。当 Y 坐标小于 30 时，只有文本的下半部分被显示，是因为 `Graphics.drawString` 方法中的坐标是指整个文本块显示时的左下角位置，如图 8.7 所示：

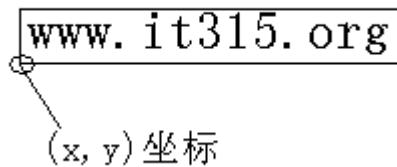


图 8.7

然而在其他语言中一般都指的是左上角位置，读者只要知道这个问题就够了，我们也不必把这作为程序问题去改正。

## F 指点迷津：

我们使用 `System.out.println()` 语句是不能够在图形窗口中打印字符文本的，只有使用 `Graphics.drawString()` 语句才能够在图形窗口中打印字符文本。在 GUI 程序中仍然可以使用 `System.out.println()` 语句打印字符文本，只是打印的字符文本会显示在命令行窗口中。

## \$ 独家见解：

我们以前讲过，在 Java 的命名习惯中，常量中的每个字母都大写。我们程序中用到了 `Color.red` 这个常量，可其中 `red` 的每个字母全都是小写。作者在很早以前，第一次碰到这个问题时，就断定：当初这一部分的设计者没有良好的命名习惯，弄了个监守自盗，贻笑大方了。果不其然，在 JDK1.4 的文档中，我们能够看到又增加了一个 `Color.RED` 常量，这正是 SUN 公司对以前的失误作出的弥补，所以，爱琢磨的读者不要再问 `Color.red` 与 `Color.RED` 有什么区别这个问题了，这是对同一事物的两种称呼，只是前者的命名不太正规，后者更为正规。

### 8.4.2 组件重绘

我们将程序窗口最小化后再恢复正常化显示，发现所绘图形全部消失了，这又是怎么回事呢？如何解决这个问题呢？在组件大小改变或隐藏后又显示，AWT 线程都会重新绘制组件，组件上原来绘制的图形也就不复存在了，这一过程称为“曝光”。要想让用户感觉到所绘的图形一直存在，我们只需在组件重新绘制后，立即将原来的图形重新画上去，这个过程是肉眼感觉不到的。我碰到过有个学员对此很不高兴，说这不是在骗人吗？记住，我们程序在于效果，而不关心你是用那种方式实现的，我不知道这位学员最后的工作情况，但我真的感受到了人的思维是千奇百怪的。AWT 线程在重新绘制组件后，会立即调用组件的 `paint` 方法，所以我们的图形重绘代码应该在 `paint` 方法中编写。

`paint` 方法是这样定义的：

```
public void paint(Graphics g)
```

可见，AWT 线程已经获得了组件上的 `Graphics` 对象，并将它传递给了 `paint` 方法，在 `paint` 方法中绘图时只能使用这个 `Graphics` 对象。

当我们想要执行 `paint` 方法中的程序代码时，应用程序不应直接调用 `paint` 方法，如果我们想执行 `paint` 方法中的程序代码，需调用 `Component.repaint` 方法，`Component.repaint` 方法调用 `Component.update` 方法，`Component.update` 再调用 `Component.paint`。组件重绘的调用关系如图 8.8 所示：

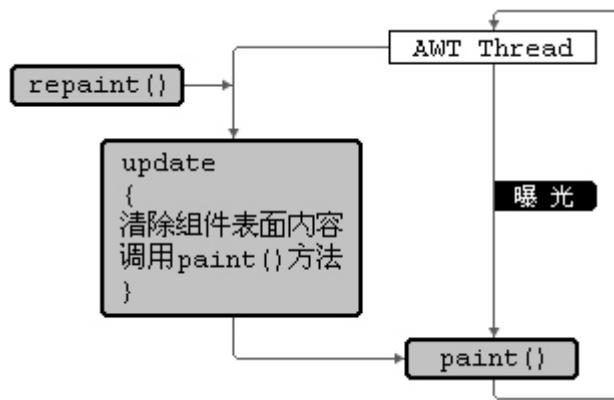


图 8.8

由于我们不可能直接进入到某个组件的 `paint` 方法中修改程序代码，我们需要定义一个继承了该组件的子类，在子类中覆盖 `paint` 方法，在新的 `paint` 方法中编写重绘图形程序代码，并将原先创建的组件对象改为由这个子类创建，就可以达到我们的目的了。我们修改上面的程序代码，使其具有重绘效果：

```

import java.awt.*;
import java.awt.event.*;
public class DrawLine extends Frame
{
    int orgX;
    int orgY;
    int endX;
    int endY;
    public static void main(String [] args)
    {
        DrawLine dl=new DrawLine();
        dl.init();
    }
    public void paint(Graphics g)
    {
        g.drawLine(orgX,orgY,endX,endY);
    }
    public void init()
    {
        /*基础差的读者如果难以理解下面这句代码，可将前面 this 处的注释去掉后再理解。也可这样理解，由于 DrawLine 是 Frame 的子类，所以可以直接调用 Frame 中的方法。*/
        /*this.* /addMouseListener(new MouseAdapter()
        {
            public void mousePressed(MouseEvent e)
            {
                orgX=e.getX();
                orgY=e.getY();
            }
            public void mouseReleased(MouseEvent e)

```

```

    {
        endX= e.getX();
        endY= e.getY();
        Graphics g=/*this.*getGraphics();
        g.setColor(Color.red);//设置绘图颜色为红色
        g.setFont(new Font("隶书",
            Font.ITALIC|Font.BOLD,30));//设置文本的字体
        g.drawString(new String(orgX +"," +orgY),
            orgX,orgY);//打印鼠标按下时的坐标文本
        g.drawString(new String(endX +"," +endY),
            endX, endY);//打印鼠标释放时的坐标文本
        g.drawLine(orgX,orgY, endX, endY);
    }
});
}
}

```

由于在 `paint` 方法中重绘直线时，需要两个点的坐标，所以，在上面的程序，又增加了两个成员变量 `endX`, `endY` 用于保存鼠标释放时的坐标。上面的程序只能重绘最后的那条直线，读者可以自己想想其中的道理。

如果我们想完全重绘窗口上的内容，我们需要将每一条直线的坐标保存到一个集合类中，在 `paint` 方法中取出该集合中的每一条直线的坐标，逐一绘制。首先我们创建一个新的类 `MyLine`，对应所画的一条直线，该类中定义了一个 `drawMe` 方法，用于将该直线在某个 `Graphics` 对应的组件上画出自己。

```

import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;
class MyLine
{
    private int x1;
    private int y1;
    private int x2;
    private int y2;
    public MyLine(int x1,int y1,int x2,int y2)
    {
        this.x1=x1;
        this.y1=y1;
        this.x2=x2;
        this.y2=y2;
    }
    public void drawMe(Graphics g)
    {
        g.drawLine(x1,y1,x2,y2);
    }
}

```

```

}

public class RerawAllLine extends Frame
{
    Vector vLines=new Vector();
    public static void main(String [] args)
    {
        RedrawAllLine f=new RedrawAllLine();
        f.init();
    }
    public void paint(Graphics g)
    {
        g.setColor(Color.red);
        Enumeration e=vLines.elements();
        while(e.hasMoreElements())
        {
            MyLine ln=(MyLine)e.nextElement();
            ln.drawMe(g);
        }
    }

    public void init()
    {
        this.addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent e)
            {
                ((Window)e.getSource()).dispose();
                System.exit(0);
            }
        });
        addMouseListener(new MouseAdapter(){
            int orgX;
            int orgY;
            public void mousePressed(MouseEvent e)
            {
                orgX=e.getX();
                orgY=e.getY();
            }
            public void mouseReleased(MouseEvent e)
            {
                Graphics g=e.getComponent().getGraphics();
                g.setColor(Color.red);
                g.drawLine(orgX,orgY,e.getX(),e.getY());
            }
        });
    }
}

```

```

        vLines.add(new MyLine(orgX,orgY,e.getX(),e.getY()));
    }
});
this.setSize(300,300);
setVisible(true);

}
}

```

### 8.4.3 图像操作

我们可以通过 `Graphics.drawImage(Image img, int x, int y, ImageObserver observer)` 方法在组件上显示图像，其中，`img` 参数是要显示的图像对象，`x, y` 是图像显示的左上角坐标，`observer` 是用于监视图像创建进度的一个对象。`drawImage` 是一个异步方法，即使 `img` 对应的图像还没有完全装载（在创建 `Image` 类对象时并没有在内存中创建图像数据）时，`drawImage` 也会立即返回。如果程序想了解图像创建的进度信息，我们需要编写一个实现了 `ImageObserver` 接口的类，并将该类创建的对象传递给 `drawImage` 方法，这个过程和原理与前面讲过的事件监听器相似。如果程序不关心图像创建的进度信息，我们可以传递要显示图像的那个组件对象，因为 `Component` 类已实现了 `ImageObserver` 接口。下面是一个显示图像的例子程序。

```

import java.awt.*;
import java.awt.event.*;
public class DrawImage
{
    public static void main(String [] args)
    {
        Frame f= new Frame("IT 人资讯交流网");
        Image img=f.getToolkit().getImage("c:\\test.gif");
        f.getGraphics().drawImage(img,0,0, f);
        f.setSize(300,300);
        f.setVisible(true);
        f.addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent e)
            {
                System.exit(0);
            }
        });
    }
}

```

喜爱思考的读者都知道，在实际工作中，可能会碰到我们课程中没有讲到的许多小细节，更希望知道老手们解决问题的方法和思想，对于程序中的 `Image img = f.getToolkit().getImage("c:\\test.gif")` 这行代码，可能会问：“如果是我自己写程序，我怎么能够知道产生一个 `Image` 对象，要用到这些方法和这些调用关系呢？”其实，最聪明的人是最会使用工具的人，特别是编写程序，如果我们有一个好的帮助系统，经常会有事半功倍的特效。我们在前面已经为大家讲解了 `chm` 格式的 JDK 文档。有了这样的帮助系

统，我们就可以进行模糊查询了。由于 `Image` 是抽象类，我们不能使用构造方法，只能通过某个方法来产生一个 `Image` 对象了，这个方法名，估计就是 `createImage`, `getImage` 之类的英文单词组合，按照这种猜想，通过帮助的索引查询，一般都会有较大的收获，我们查到了 `Toolkit.getImage(String path)` 方法，由于 `Toolkit` 是抽象类，不能直接创建，接着用 `getToolkit` 又找到了 `Component.getToolkit` 方法返回 `Toolkit` 对象，就这样顺藤摸瓜写出这个完整的语句了，由于很多东西，特别是一些老手的工作经验都是只可意会，不可言宣的，所以自己读书和参加面授培训的效果还是有区别的。

上面的程序在运行时会报告第 9 行产生了 `java.lang.NullPointerException`，一定要注意提示的行号，能够帮我们迅速定位错误。在 JDK 文档查找 `getGraphics` 的帮助，文档中说得非常清楚，只有在组件已显示在窗口上时，`getGraphics` 方法才能正确返回一个 `Graphics` 对象。将程序代码改成下面这样。

```
import java.awt.*;
import java.awt.event.*;
public class DrawImage
{
    public static void main(String [] args)
    {
        Frame f= new Frame("IT 人资讯交流网");
        f.setSize(300,300);
        f.setVisible(true);
        f.addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent e)
            {
                System.exit(0);
            }
        });
        Image img=f.getToolkit().getImage("c:\\\\test.gif");
        f.getGraphics().drawImage(img,0,0, f);
    }
}
```

运行时不再有错，但图像并没有显示出来。这是因为在窗口初始显示时也会被调用 `paint` 方法，`paint` 方法会擦除窗口上绘制的图形，这里的 `drawImage` 方法先于 `paint` 方法执行，所以，`drawImage` 方法绘制的图像被 `paint` 方法擦除掉了。读者不要误以为 `setVisible` 方法执行时，`paint` 方法就会立即执行，`paint` 方法是由 AWT 线程调度和管理的。顺便告诉大家另外一个小经验，放在 `Frame.setVisible(true)` 之后的 GUI 程序代码，在窗口初始显示时，都看不出期望的执行效果。例如，`f.setVisible(true); f.add(new Button("test"));` 这样的程序代码，我们在窗口初始显示时也是看不到窗口上放置的按钮的，改为 `f.add(new Button("test")); f.setVisible(true);` 这样的顺序则可以了。对于上面的问题，我们可以将图像放在 `paint` 方法中显示，修改后的程序如下：

```
import java.awt.*;
import java.event.*;
public class DrawImage extends Frame
{
    Image img=null;
```

```

    public static void main(String [] args)
    {
        DrawImage f= new DrawImage();
        f.init();
    }
    public void init()
    {
        setSize(300,300);
        setVisible(true);
        this.addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent e)
            {
                System.exit(0);
            }
        });
        img=this.getToolkit().getImage("c:\\test.gif");
    }
    public void paint(Graphics g)
    {
        getGraphics().drawImage(img,0,0,this);
    }
}

```

编译运行后，有时候图像立即就能够正常显示出来了，有时候图像并没有显示出来，而是在命令行窗口中出现了如下错误：

```

C:\>java DrawImage
java.lang.NullPointerException
    at sun.java2d.pipe.DrawImage.copyImage(DrawImage.java:48)
    at sun.java2d.pipe.DrawImage.copyImage(DrawImage.java:715)
    at sun.java2d.pipe.ValidatePipe.copyImage(ValidatePipe.java:147)
.....

```

改变窗口的大小，这时图像就显示出来了。原因在于，这时候正好碰到了 AWT 线程调用 `paint` 方法早于 `getImage` 方法的情况，而在 `paint` 方法中执行 `drawImage` 的时候，`img` 对象仍为 `null`。我们将调用 `getImage` 方法的语句放在 `setVisible` 语句之前，这样就万无一失，一切正常了。修改后的程序代码如下：

```

import java.awt.*;
import java.awt.event.*;
public class DrawImage extends Frame
{
    Image img=null;
    public static void main(String [] args)
    {
        DrawImage f= new DrawImage();
        f.init();
    }
}

```

```

public void init()
{
    img=this.getToolkit().getImage("c:\\\\test.gif");
    setSize(300,300);
    setVisible(true);
    this.addWindowListener(new WindowAdapter()
    {
        public void windowClosing(WindowEvent e)
        {
            System.exit(0);
        }
    });
}

public void paint(Graphics g)
{
    getGraphics().drawImage(img,0,0,this);
}
}

```

读者要逐渐学会作者对上面程序的反复实验和分析错误的过程，培养一种分析程序问题的思维，提高解决程序问题的手段和方法，积累调试程序问题的经验。

#### 8.4.4 双缓冲的技术

在画线重绘程序中，我们在窗口重画时，需要逐一重新绘制窗口上原来的图形（直线，文本），如果原来的图形非常多，这个过程就显得比较慢了。一种改进的办法是，我们可以调用 Component.createImage 方法，在内存中创建一个 Image 对象，当我们在组件上绘图时，也在这个 Image 对象上执行同样的绘制，即 Image 对象中的图像是组件表面内容的复制，当组件重画时，我们只需将内存中的这个 Image 对象在组件上画出，不管组件上原来的图形有多少，在重绘时都只是一幅图像而已，在组件上的图形非常多时，重绘速度明显提高，这就是一种被称为双缓冲的技术。下面是这个应用的例子程序。

程序清单：DrawLine.java

```

import java.awt.*;
import java.awt.event.*;
public class DrawLine extends Frame
{
    Image oimg=null;
    Graphics og=null;
    public static void main(String [] args)
    {
        new DrawLine().init();
    }
    public void init()
    {

```

```

setSize(300,300);
setVisible(true);
Dimension d=getSize();
oimg=createImage(d.width,d.height);
og=oimg.getGraphics();
addMouseListener(new MouseAdapter()
{
    int orgX;
    int orgY;
    public void mousePressed(MouseEvent e)
    {
        orgX=e.getX();
        orgY=e.getY();
    }
    public void mouseReleased(MouseEvent e)
    {
        Graphics g=getGraphics();
        g.setColor(Color.red);//设置绘图颜色为红色
        g.setFont(new Font("隶书",Font.ITALIC|Font.BOLD,30));
        //设置文本的字体
        g.drawString(new String(orgX +"," +orgY),orgX,orgY);
        //打印鼠标按下时的坐标文本
        g.drawString(new String(e.getX() +"," +e.getY()),
        e.getX(),e.getY());//打印鼠标释放时的坐标文本
        g.drawLine(orgX,orgY,e.getX(),e.getY());
        og.setColor(Color.red);//设置绘图颜色为红色
        og.setFont(new Font("隶书",Font.ITALIC|Font.BOLD,30));
        //设置文本的字体
        og.drawString(new String(orgX +"," +orgY),orgX,orgY);
        //打印鼠标按下时的坐标文本
        og.drawString(new String(e.getX() +"," +e.getY()),
        e.getX(),e.getY());//打印鼠标释放时的坐标文本
        og.drawLine(orgX,orgY,e.getX(),e.getY());
    }
}) ;
}
public void paint(Graphics g)
{
    if(oimg !=null)
        g.drawImage(oimg,0,0,this);
}
}

```

如果在 `paint` 方法中没有使用 `if` 语句对 `oimg` 进行是否为 `null` 的检查，就会出现如下的错误：

```
C:\>java DrawLine  
java.lang.NullPointerException  
    at sun.java2d.pipe.DrawImage.copyImage(DrawImage.java:48)  
    at sun.java2d.pipe.DrawImage.copyImage(DrawImage.java:715)  
    at sun.java2d.SunGraphics2D.drawImage(SunGraphics2D.java:2782)  
    at sun.java2d.SunGraphics2D.drawImage(SunGraphics2D.java:2772)  
....
```

`createImage` 是 `Component` 的一个方法，只有部件显示在桌面上之后才能调用这个方法，部件显示时会调用它的 `paint` 方法，也就是 `createImage` 方法只能在第一次 `paint` 方法调用之后才能被调用。我们在 `paint` 方法中的 `drawImage` 语句中，又要调用 `createImage` 方法产生的 `oimg` 对象，这又要求 `createImage` 方法应在 `paint` 方法之前调用。上面的矛盾就成了“鸡生蛋，蛋生鸡”的问题，所以，我们在程序中使用 `if` 语句对 `oimg` 进行是否为 `null` 的检查来解决这个矛盾。

|  |     |
|--|-----|
| 第 8 章 GUI (上) .....  | 244 |
| 8.1 初识 AWT.....  | 244 |
| 8.2 AWT 线程.....  | 245 |
| 8.3 AWT 事件处理.....  | 246 |
| 8.3.1 事件处理机制.....  | 246 |
| 8.3.2 用事件监听器处理事件.....<br>指点迷津：同一事件源的不同表示类型<br>小经验：事件监听器方法的返回类型 | 248 |
| 8.3.3 事件适配器.....   | 249 |
| 8.3.4 事件监听器的匿名内置类实现方式.....                                     | 252 |
| 8.3.5 事件处理的多重运用.....   | 253 |
| 8.3.6 高级事件处理.....  | 255 |
| 8.4 GUI 组件上的图形操作.....  | 257 |
| 8.4.1 Graphics 类 .....   | 257 |
| 指点迷津：如何打印字符文本<br>独家见解：JDK 的失误                                  |     |
| 8.4.2 组件重绘.....  | 260 |
| 8.4.3 图像操作.....  | 264 |
| 8.4.4 双缓冲的技术.....  | 267 |

# 第9章 GUI (下)

## 9.1 常用 AWT 组件

在上一章开始的第一个程序中，我们就用到了按钮（Button），这是最简单的组件之一了，在 AWT 里还有很多用于 GUI 设计的组件，我们现在就来了解更多的 GUI 组件。如图 9.1 所示描述了 AWT 中的组件及类层次关系图。

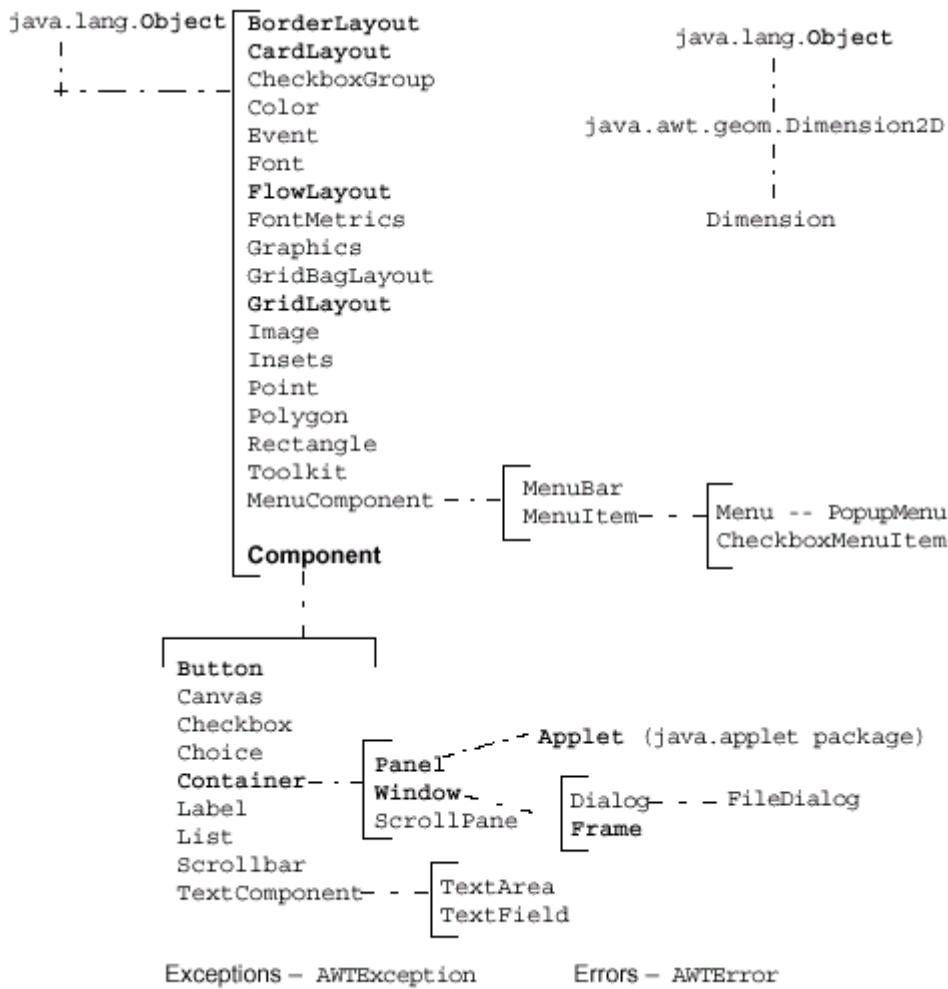


图 9.1

### 9.1.1 Component 类

Java 的图形用户界面的最基本组成部分是组件，组件是一个可以以图形化的方式显示在屏幕上并能与用户进行交互的对象，例如一个按钮，一个标签等。抽象类 `Component` 是所有 Java GUI 组件的共同父类。`Component` 类规定了所有 GUI 组件的基本特性，该类中定义的方法实现了作为一个 GUI 组件所应具备的基本功能。Java 程序要显示的 GUI 组件必须是抽象类 `Component` 或 `MenuComponent` 的子类。

### 9.1.2 Canvas

`Canvas` 代表屏幕上的一块空白的矩形区域，程序能够在这个部件表面绘图，也能够捕

获用户的操作，产生相应的事件，Canvas 可以说是具有最基本的和最简单的 GUI 功能的部件。当我们要设计一种自己定制的具有 GUI 功能的部件类，我们的这个类就可以继承 Canvas，这样，我们的部件类就已经完成了 GUI 的基本功能，我们只需要在这个基础上增加子类部件所专有的外观和功能的相关代码就行了，我们要想绘制子类部件的外观，我们必须覆盖 Canvas 的 paint 方法。

我们现在设计一个计时器部件，鼠标在部件上按下时，计时器开始计时，并在部件上显示计时时间，鼠标释放时，计时器停止计时。下面是这个计时器的程序代码，其中涉及到的技巧和知识点，都在前面有过讲解，这里就不作详细解释了。

程序清单：TestStopWatch.java

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.text.SimpleDateFormat;
class StopWatch extends Canvas implements Runnable
{
    private long startTime = 0;
    private long endTime = 0;
    private boolean bStart = false;
    public StopWatch()
    {
        enableEvents(AWTEvent.MOUSE_EVENT_MASK);
        setSize(80,30);
    }
    protected void processMouseEvent(MouseEvent e)
    {
        if(e.getID() == MouseEvent.MOUSE_PRESSED)
        {
            bStart = true;
            startTime = endTime = System.currentTimeMillis();
            repaint();
            new Thread(this).start();
        }
        else if(e.getID() == MouseEvent.MOUSE_RELEASED)
        {
            bStart = false;
            repaint();
        }
        super.processMouseEvent(e);
    }
    public void paint(Graphics g)
    {
        SimpleDateFormat sdf= new SimpleDateFormat("HH:mm:ss");
        Date elapsedTime =null;
```

```

        try
        {
            elapsedTime= sdf.parse("00:00:00");
        }catch(Exception e){}
        elapsedTime.setTime(endTime - startTime +
                           elapsedTime.getTime());
        String display = sdf.format(elapsedTime);
        g.drawRect(0,0,78,28);
        g.fill3DRect(2,2,75,25,true);
        g.setColor(Color.RED);
        g.drawString(display,10,20);
    }
    public void run()
    {
        while(bStart)
        {
            try
            {
                Thread.sleep(500);
            }catch(Exception e){e.printStackTrace();}
            endTime = System.currentTimeMillis();
            repaint();
        }
    }
}
public class TestStopWatch
{
    public static void main(String [] args)
    {
        Frame f =new Frame("StopWatch");
        f.add(new StopWatch());
        f.setSize(200,200);
        f.setVisible(true);
        f.addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent e)
            {
                System.exit(0);
            }
        });
    }
}

```

编译并运行这个程序，你一定要用鼠标按住计时器部件几秒钟，你就能看到我们期望的效果了，如图 9.2 所示。

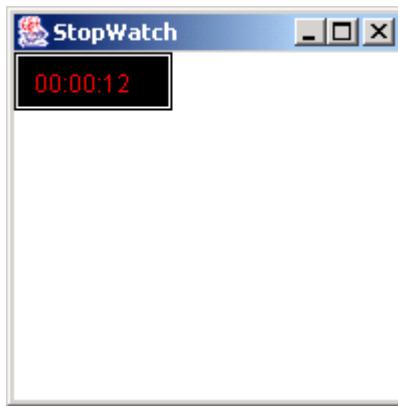


图 9.2

如果你想让这个计时器部件尽可能满足更多的应用需求，你还需要增加一些功能，如允许用户设置文本的颜色，文本的大小随部件大小的改变而改变等。

### 9.1.3 Checkbox

如果你熟悉 Windows 的应用，对单选按钮和多选按钮一定不会陌生，这两种按钮都有选中和不选两种状态，如图 9.3 所示。对多选按钮来说，如果有多个这样的按钮，每个按钮之间没有制约关系，可以同时选中其中的多个。而单选按钮则要求有一组按钮，这一组按钮中同时只能有一个为选中状态。

Java 里提供的这个 Checkbox 类来建立单选按钮和多选按钮，Checkbox 的使用很容易，如果要创建多选按钮，我们只要使用 `public Checkbox(String label, boolean state)` 这个构造函数来创建 Checkbox 对象就行了，创建多选按钮要用到两个参数，前一个是选框旁边的说明文字，后一个参数决定选框是否默认被选定。因为创建单选按钮需要一组按钮，所以在创建单选按钮时，我们还需要指定这个按钮所属于的组，使用

```
public Checkbox(String label,boolean state,CheckboxGroup group)
```

这个构造函数创建的就是单选按钮。其中，CheckboxGroup 类对象指定了这个单选按钮所属于的组。

对一般的程序来说，需要处理单选按钮和多选按钮的 ItemEvent 事件，从而获得用户选择的结果。处理 ItemEvent 事件的监听器接口为 ItemListener，其中只有一个 itemStateChanged 方法，显然，ItemEvent 是一种语义事件。

下面是一段创建多选按钮和单选按钮以及相关事件处理的程序代码：

程序清单：TestCheckbox.java

```
import java.awt.*;
import java.awt.event.*;
public class TestCheckbox
{
    Checkbox cb1=new Checkbox("你喜欢我吗？",true);
    CheckboxGroup cbg=new CheckboxGroup();
    Checkbox cb2=new Checkbox("喜欢",cbg,true);
    Checkbox cb3=new Checkbox("不喜欢",cbg,false);

    public void init()
    {
        Frame f=new Frame("TestCheckBox");
    }
}
```

```

//创建 FlowLayout 布局管理器, 关于布局管理器, 本章后面有专门的讲解, 看不明白
//的读者暂时可以不去下面两句代码的作用。
FlowLayout fl=new FlowLayout();
f.setLayout(fl);

f.add(cb1);
f.add(cb2);
f.add(cb3);
cb1.addItemListener(new CbItemListener());
cb2.addItemListener(new CbItemListener());
cb3.addItemListener(new CbItemListener());
f.setBounds(0,0,300,100);
f.setVisible(true);
f.addWindowListener(new WindowAdapter()
{
    public void windowClosing(WindowEvent e)
    {
        System.exit(0);
    }
});
}

class CbItemListener implements ItemListener
{
    public void itemStateChanged(ItemEvent e)
    {
        Checkbox cb = (Checkbox)e.getItemSelectable();
        if(cb.getLabel().equals("你喜欢我吗? "))
        {
            if(cb.getState() == true)
                System.out.println("我很高兴");
            else
                System.out.println("我很伤心");
        }
        /*else if(cb.getLabel().equals("喜欢"))
        {
            if(e.getStateChange() == ItemEvent.SELECTED)
                System.out.println("我也喜欢你");
            else
                System.out.println("我也不喜欢你");
        }*/
        else
        {
            Checkbox cbx =cbg.getSelectedCheckbox();
            if(cbx != null)

```

```

        System.out.println(cbx.getLabel());
    }
}
}

public static void main(String[] args)
{
    new TestCheckbox().init();
}
}

```

程序运行效果如图 9.3 所示：



图 9.3

如果我们用注释的程序代码来处理单选按钮，需要为每一个单选按钮都编写一段这样的代码，如果按钮的个数较多，程序就比较臃肿，但从这段注释代码中，读者可以了解到对同一问题的多种处理方式，也可以在正好有这方面的需要时参考。如果我们不想直接处理这些单选按钮的事件，而是在别的部件的事件处理代码中收集这些单选按钮的选择结果。譬如，我们经常在最后单击 ok 按钮的事件中再去收集设置对话框上的所有部件的设置结果，我们使用类似程序中 else 部分的代码，就是一个不错的方法。

#### 9.1.4 Choice

Choice 类用来制作用于单选的下拉列表，用起来也比较容易，来看这段程序：

程序清单：TestChoice.java

```

import java.awt.*;
import java.awt.event.*;
public class TestChoice
{
    Choice ch=new Choice(); //创建 Choice 对象
    TestChoice()
    {
        ch.add("choice1"); //用 add 方法向列表里加入选项
        ch.add("choice2"); //用 add 方法向列表里加入选项
        ch.add("choice3"); //用 add 方法向列表里加入选项
        FlowLayout fl=new FlowLayout();
        Frame f=new Frame("TestChoice");
        f.setLayout(fl);
        f.add(ch); //把列表加入到窗口
        f.setBounds(0,0,200,100);
        f.setVisible(true);
        f.addWindowListener(new WindowAdapter()

```

```

{
    public void windowClosing(WindowEvent e)
    {
        System.exit(0);
    }
}};

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

}

```

如图 9.4 所示是程序执行后展开列表的情况：



图 9.4

### 9.1.5 菜单

一个完整的菜单系统由菜单条、菜单和菜单项组成。他们与菜单的对应关系如图 9.5 所示：

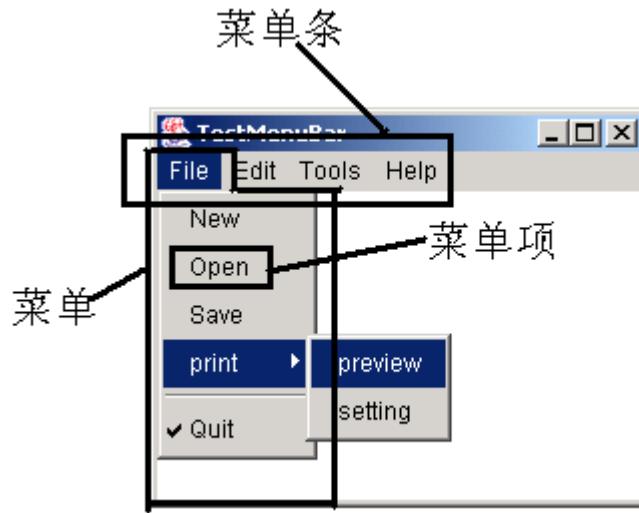


图 9.5

在图中，File、Edit、Tools、Help 各项叫做菜单，这些顶层菜单共同组合成菜单条，在 File 项的下拉菜单中 New、Open 等各项叫做菜单项。

Java 中与菜单相关的类主要有三个 `MenuBar`（菜单条）、`Menu`（菜单）、`MenuItem`（菜单项）。

我们来看看完成图 9.5 的菜单系统的程序代码。

程序清单：TestMenuBar.java

```
import java.awt.*;
import java.awt.event.*;
public class TestMenuBar
{
    MenuBar menubar=new MenuBar(); //创建菜单条对象
    Menu fileM=new Menu("File"); //创建各菜单
    Menu editM=new Menu("Edit"); //创建各菜单
    Menu toolsM=new Menu("Tools"); //创建各菜单
    Menu helpM=new Menu("Help"); //创建各菜单

    MenuItem fileMI1=new MenuItem("New"); //创建各菜单项
    MenuItem fileMI2=new MenuItem("Open");
    MenuItem fileMI3=new MenuItem("Save");
    CheckboxMenuItem fileMI5=new CheckboxMenuItem("Quit",true);
    //创建各菜单项

    Menu filePrint = new Menu("print");//创建子菜单
    MenuItem printM1 = new MenuItem("preview");
    MenuItem printM2 = new MenuItem("setting");

    TestMenuBar()
    {
        FlowLayout fl=new FlowLayout();

        Frame f=new Frame("TestMenuBar");
        f.setLayout(fl);

        menubar.add(fileM); //将菜单加入菜单条
        menubar.add(editM);
        menubar.add(toolsM);
        menubar.add(helpM);

        fileM.add(fileMI1); //将菜单项加入 file 菜单中
        fileM.add(fileMI2);
        fileM.add(fileMI3);

        filePrint.add(printM1); //将菜单项加入 print 菜单中
        filePrint.add(printM2);
        fileM.add(filePrint);
        //将 print 菜单作为一个菜单项加入 file 菜单中

        fileM.addSeparator(); //将一条分割线加入菜单中
```

```

        fileM.add(fileMI5); //将菜单项加入菜单中
        f.setMenuBar(menuBar); //把整个菜单系统显示在窗口中
        f.setBounds(0,0,250,200);
        f.setVisible(true);
        f.addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent e)
            {
                System.exit(0);
            }
        });
    }

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

```

在窗口上产生菜单的过程非常简单，我们首先要产生 `MenuBar` 对象，然后产生 `Menu` 对象，最后产生 `MenuItem` 对象。将 `MenuItem` 增加到 `Menu` 上后，再将 `Menu` 增加到 `MenuBar` 上，最后将 `MenuBar` 挂到 `Frame` 窗口上。要注意的一点是，`Menu` 类本身又是 `MenuItem` 的子类，一个 `Menu` 对象也可以作为一个菜单项增加到另外一个 `Menu` 对象上，这就是我们在上面看到的 `print` 子菜单。

关于程序中用到的复选菜单以及菜单分割条，读者不用刻意去记，用到的时候查一下 JDK 文档就可以找到，另外，在诸如 Jbuilder 的集成开发环境里，其可视化的生成工具就刻意帮我们产生这样程序代码，我们只要看得懂就行了。

程序不用处理菜单条和菜单的事件，但需要对菜单项的动作进行响应，单击一个菜单项，会发出 `ActionEvent` 事件。我们来看看如何对上面程序中的 `print` 子菜单的两个菜单项的动作进行处理，为了直观，下面只给出了新增部分的程序代码。

```

Menu filePrint = new Menu("print");//创建子菜单
MenuItem printM1 = new MenuItem("preview");
MenuItem printM2 = new MenuItem("setting");

MenuListener ml = new MenuListener();
printM1.addActionListener(ml);
printM2.addActionListener(ml);
class MenuListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        if(e.getActionCommand().equals("preview"))
            System.out.println("doing preview");
        else if(e.getActionCommand().equals("setting "))
            System.out.println("doing setting");
    }
}

```

```
    }  
}
```

对于发出 ActionEvent 事件的组件，我们可以调用 setActionCommand 方法为其关联一个字符串，用于指示这个动作想执行的命令。如果程序没有使用 setActionCommand 方法为组件关联一个命令字符串，则其命令字符串为组件的标题文本。ActionEvent 的 getActionCommand 方法就是用于返回这个命令字符串的。

使用命令字符串，我们可以用同一菜单来发出连接和断开的命令，在要发出的命令为连接前，我们用 MenuItem.setActionCommand 指定命令字符串为“connect”，在要发出的命令为断开前，我们指定命令字符串为“disconnect”，事件处理程序通过判断这个命令字符串，就知道该采取哪种动作了。如果我们程序中的菜单要针对不同的国家，用不同语言文字显示，我们不管菜单项标题上显示的是什么文字，只要用 setActionCommand 方法为这个菜单项指定一个命令字符串，我们就可以用同样的事件处理程序去处理这个用不同语言文字显示的菜单项的事件。

其实，其他一些组件也可以使用 getActionCommand 方法，到底有哪些，读者只要在 JDK 文档中查找 getActionCommand 就知道了，如图 9.6 所示：



图 9.6

### 9.1.6 Container 类

组件不能独立地显示出来，必须将组件放在一定的容器中才可以显示出来。象我们上面见到的窗口就是一个容器，类 Container 是所有容器的父类，容器（Container）实际上是 Component 的子类，因此容器类对象本身也是一个组件，具有组件的所有性质，另外还具有容纳其它组件和容器的功能。容器类对象可以使用方法 add() 添加组件。容器类的类层次图如图 9.7 所示：

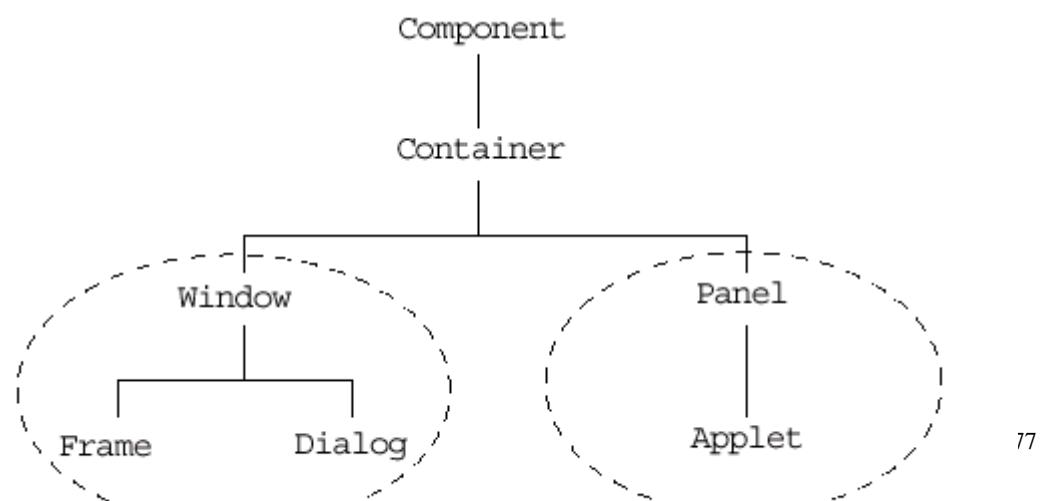


图 9.7

Container 有几个主要的子类: Window 类, Panel 类, ScrollPane 类。

### 9.1.7 Window 类

Window 类是可自由停泊的顶级窗口, 它没有边框和菜单条, 我们很少直接使用 Window 类, 而是使用它的两个子类: Frame 类和 Dialog 类。Frame 对象显示效果是一个“窗口”, 带有标题和尺寸重置角标, 默认初始化为不可见的, 可以使用 setVisible(true)方法使之变为可见, 在前面我们已经多次用到了 Frame 类。

### 9.1.8 Dialog

Dialog (对话框) 一般是一个临时的窗口, 用于显示提示信息或接收用户输入。在对话框中一般不需要菜单条, 也不需要改变窗口大小。有两种模式的对话框, 模态对话框和非模态对话框。模态对话框显示时, 用户不能操作其它窗口, 直到这个对话框被关闭。非模态对话框显示时, 用户还可以操作其它窗口。Dialog 类用来创建用户对话框, 对话框和框架 (Frame) 比较相似, 同样可以在对话框上添加其它的组件。

Dialog 类有两种常用的构造方法:

```
public Dialog(Frame owner, String title);
public Dialog(Frame owner, String title, boolean modal)
```

对话框不能独立存在, 它必须有一个上级窗口, 这个上级窗口就是对话框的拥有者。这两个构造方法的第一个参数代表对话框的拥有者, 第二个参数是对话框标题, 第三个参数设置对话框的模式, 如果是 true, 则为模态对话框, 如果是 false, 则为非模态对话框。

下面是一个关于对话框的例子程序。

```
程序清单: TestDialog.java
import java.awt.*;
import java.awt.event.*;
public class TestDialog
{
    TextField tf = new TextField(10);
    Button b1=new Button("模态显示");
    Button b2=new Button("非模态显示");
    Frame f=new Frame("TestDialog");

    Button b3=new Button("确定");
    Dialog dlg = new Dialog(f, "Dialog Title", true);
    FlowLayout fl=new FlowLayout();
    TestDialog()
```

```

{
f.setLayout(f1);
f.add(tf);
f.add(b1);
f.add(b2);
b1.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e)
    {
        dlg.setModal(true);
        dlg.setVisible(true);
        tf.setText("www.it315.org");
    }
}) ;
b2.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e)
    {
        dlg.setModal(false);
        dlg.setVisible(true);
        tf.setText("www.it315.org");
    }
}) ;
f.setBounds(0,0,400,200);
f.setVisible(true);
f.addWindowListener(new WindowAdapter(){
    public void windowClosing(WindowEvent e)
    {
        System.exit(0);
    }
}) ;

dlg.setLayout(f1);
dlg.add(b3);
b3.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        dlg.dispose();
    }
}) ;
dlg.setBounds(0,0,200,150);

}

```

```

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

}

```

程序运行效果如图 9.8 所示:



图 9.8

细心的读者能够发现，当我们用模态方式显示对话框后，我们不能再操作主框架窗口，程序中的 `tf.setText("www.it315.org")` 语句没有被执行，这条语句在当对话框关闭后才被执行，这说明 `dlg.setVisible(true)` 语句直到对话框关闭后才能返回。当我们用非模态方式显示对话框后，程序中的 `tf.setText("www.it315.org")` 语句立即被执行，我们也能操作主框架窗口（譬如对文本框进行输入），这说明 `dlg.setVisible(true)` 语句没有等到对话框关闭就立即返回了。

`Dialog` 类下面有一种用于文件存取对话框的子类，这就是 `FileDialog` 类，`FileDialog` 类能够产生标准的文件存取对话框，如图 9.9 所示：



图 9.9

它具有 `Dialog` 的一切特征。由于安全性限制，它不能在 `Applet` 中使用，只有在 `Application` 中才能使用。它典型的构造方法如下：

```
public FileDialog(Frame parent, String title, int mode);
```

这种方法中，前两个参数的作用和 `Dialog` 一样，第三个参数有两个值，分别是 `FileDialog.LOAD` 或 `FileDialog.SAVE`，这个参数用来确定产生的是读文件对话框还是写文

件对话框。

### 9.1.9 Panel 类

Panel 可作为容器容纳其它组件,但不能独立存在,必须被添加到其它容器中(如 Window 或 Applet)。Panel 是一个空白容器类,提供容纳组件的空间,通常用于集成其他的若干组件,使这些组件形成一个有机的整体,再增加到别的容器上。

### 9.1.10 ScrollPane 类

我们有时候需要在一个较小的容器窗口中,显示较大的子部件,这时就需要用到 ScrollPane 类。ScrollPane 也是一种容器,不能单独使用,通过滚动窗口可以利用滚动条查看大面积区域。ScrollPane 中只能放置一个组件,无布局管理器。我们要将多个组件添加到 ScrollPane 上,只能先将多个组件嵌套在一个 Panel 容器中,然后将这个 Panel 作为一个组件放置到 ScrollPane 上。下面是使用 ScrollPane 的一个例子程序。

程序清单: TestPane.java

```
import java.awt.*;
import java.awt.event.*;
public class TestPane
{
    TestPane()
    {
        Frame f=new Frame("TestDialog");
        ScrollPane sp = new ScrollPane();
        TextArea ta = new TextArea("",10,50,TextArea.SCROLLBARS_NONE);
        sp.add(ta);

        f.add(sp);
        f.setSize(200,200);
        f.setVisible(true);
        f.addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent e)
            {
                System.exit(0);
            }
        });
    }
    public static void main(String[] args)
    {
        new TestPane();
    }
}
```

由于有各种各样的组件,我们很难在这里一一介绍,其实读者只要掌握了其中几个具有典型意义的组件,其他的组件都可以在需要时临时查阅文档资料,很多东西是你自己一看都能够明白的。

## 9.2 布局管理器

### 9.2.1 了解布局管理

为了理解布局管理器，我们先来看一个程序的运行结果。

程序清单：TestLayout.java

```
import java.awt.*;
public class TestLayout
{
    public static void main(String [] args)
    {
        Frame f=new Frame("布局管理器");
        f.add(new Button("第一个按钮"));
        f.add(new Button("第二个按钮"));
        f.setSize(300,300);
        f.setVisible(true);
    }
}
```

程序运行后，显示效果如图 9.10 所示：



图 9.10

我们只看到了第二个按钮，为什么无法显示第一个按钮呢？在 AWT 中，每个组件在容器中都应该有一个具体的位置和大小，我们想在容器中排列若干组件时，会很难确定它们的大小和位置。为了简化编程者对容器上的组件的布局控制，一个容器内的所有组件的显示位置可以由一个“布局管理器”自动管理，我们可以为容器指定不同的布局管理器。在不同的布局管理器下，同一个组件将会有不同的显示效果，并且我们不能完全按自己的意愿设置组件的大小和位置了。

为了使我们生成的图形用户界面具有良好的平台无关性，Java语言中，提供了布局管理器这个工具来管理组件在容器中的布局，而不使用直接设置组件位置和大小的方式。每个容器都有一个布局管理器，当容器需要对某个组件进行定位或判断其大小尺寸时，就会调用其对应的布局管理器。

### 9.2.2 BorderLayout

BorderLayout 将容器划分为东、南、西、北、中五个区域，如图 9.11 所示。

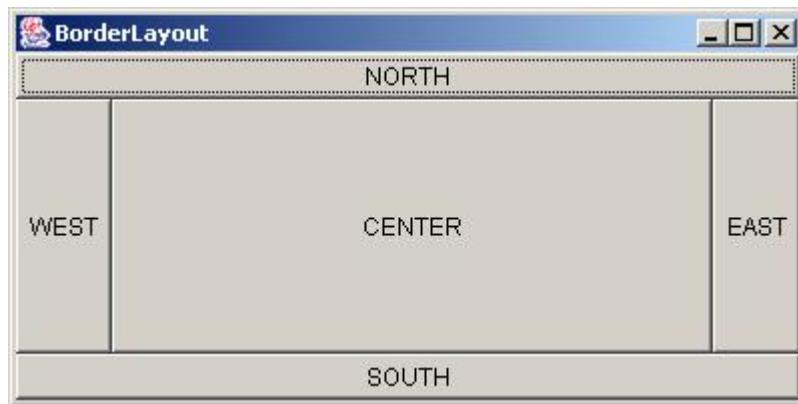


图 9.11

我们将组件添加到容器时，需要指定组件放置的区域。当改变容器大小时，北方和南方的组件只改变宽度，东方和西方的组件只改变高度，而中间组件宽度和高度都会改变。在 BorderLayout 布局管理下，这个管理器允许最多放置五个组件，如果我们想在窗口上放置更多的组件，可以将若干组件添加到一个 Panel 上，然后将这个 Panel 作为一个组件放置到窗口上。当容器上放置的组件少于五个，没有放置组件的区域将被相邻的区域占用。对 Frame 和 Dialog 来说，默认的布局管理器就是 BorderLayout 布局管理器。在上面的程序里，没有指定布局管理器，默认的 BorderLayout 布局管理器就把两个组件都放在中间了，所以实际上我们看到的是第二个按钮覆盖在第一个按钮上的效果。我们要想将上面例子中的第一个按钮放置在窗口上的北面，而第二个按钮放置在窗口的中间，程序修改如下：

```
程序清单：TestBorderLayout.java
import java.awt.*;
public class TestBorderLayout
{
    public static void main(String [] args)
    {
        Frame f=new Frame("布局管理器");
        f.add(new Button("第一个按钮"), "North");
        f.add(new Button("第二个按钮"));
        f.setSize(300,300);
        f.setVisible(true);
    }
}
```

程序运行的效果如图 9.12 所示：



图 9.12

可见，如果我们在使用 `Container.add` 方法，没有指定位置参数时，AWT 会用“Center”作为这个组件的放置位置。注意，位置参数的字符串的书写是非常严格的，不能有任何大小写问题，必须是大写。对于这个问题，作者又要发一些牢骚了，真不明白 Java 的设计人员当初怎么想的，在这些鸡毛蒜皮的小问题上都不让我们轻松，非要我们浪费一些时间去处理这些死板的细节。我们要是将“North”随手写成了“north”，也完全能够表达我们的意图啊，这对 SUN 公司的 Java 设计人员并未增加什么处理难度！读者以后在编写软件时，应多向 Microsoft 公司学习，在不增加自己多大的编程难度的情况下，尽量为用户提供方便和宽容，这反过来也为自己增加了用户的好感和信赖。

### 9.2.3 FlowLayout

`FlowLayout` 是一个简单的布局风格，组件从左到右，从上到下依次排列。如果一个组件在本行放不下，就自动换到下一行的开始。`FlowLayout` 是 `Panel` 和 `applet` 的默认布局管理器。调用 `Container.setLayout` 方法，就可以改变容器的布局管理器，下面是使用 `FlowLayout` 的例子程序。

程序清单： `TestFlowLayout.java`

```
import java.awt.*;
public class TestFlowLayout
{
    public static void main(String [] args)
    {
        Frame f=new Frame("布局管理器");
        f.setLayout(new FlowLayout());
        f.add(new Button("第一个按钮"),"North");
        f.add(new Button("第二个按钮"));
        f.add(new Button("第三个按钮"),"South");
        f.add(new Button("第四个按钮"));
        f.setSize(300,300);
        f.setVisible(true);
```

```
    }  
}
```

在 FlowLayout 布局中, Java 将忽略我们在 Container.add 方法中指定的位置参数, 如上面用黑体显示的程序行。

当容器窗口大小改变时, 组件的位置可能会发生变化, 但组件的尺寸不变, 如图 9.13 所示, 当窗口变大后, 第二行的组件排列到了第一行上。

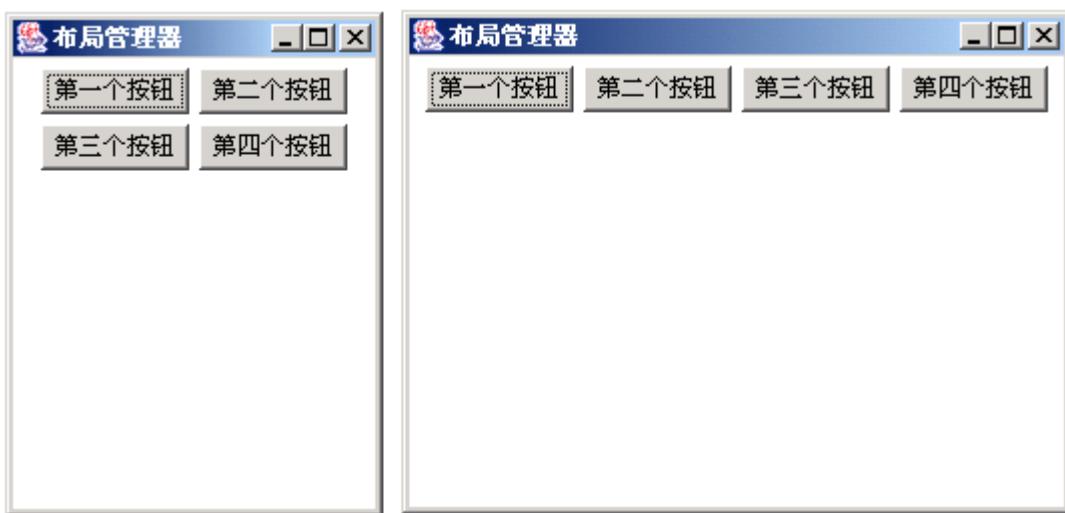


图 9.13

#### 9.2.4 GridLayout

GridLayout 将容器划分成若干行列的网格。在容器上添加组件时, 它们会按从左到右、从上到下的顺序在网格中排列。在 GridLayout 的构造方法里, 我们需要指定希望将容器划分成的网格的行、列数。GridLayout 布局管理器总是忽略组件的最佳大小, 所有单元的宽度是相同的, 是根据单元数对可用宽度进行平分而定的。同样地, 所有单元的高度是相同的, 是根据行数对可用高度进行平分而定的。其效果如图 9.14 所示。

|   |   |   |   |
|---|---|---|---|
| 1 | 2 | 3 | + |
| 4 | 5 | 6 | - |
| 7 | 8 | 9 | * |
| 0 | . | = | \ |

图 9.14

#### 9.2.5 CardLayout

CardLayout 布局管理器能够实现将多个组件放在同一容器区域内的交替显示，相当于多张卡片摞在一起，在任何时候都只有最上面的一个可见。CardLayout 提供了几个方法，可以显示特定的卡片，也可以按先后顺序依次显示，还可以直接定位到第一张或最后一张。如图 9.15 所示是我们要讲的例子程序界面。

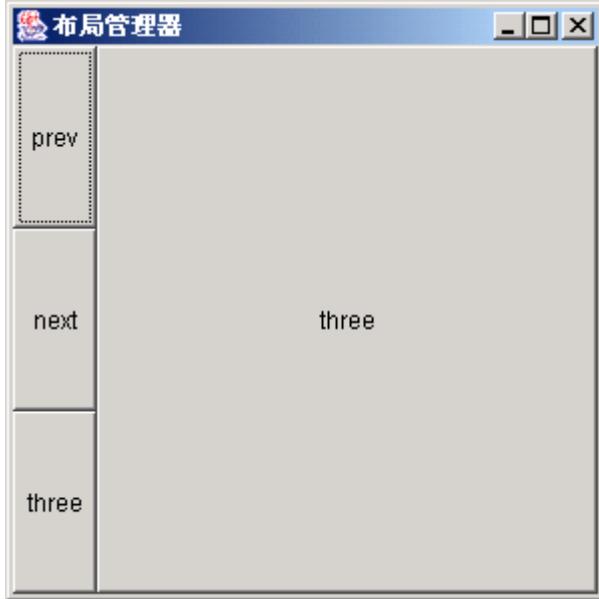


图 9.15

只有一个布局管理器来实现上面组件布局是相当的困难的，所以下面的例子联合了更多的布局类型。如果我们创建两个 Panel 对象，每个 Panel 上都能拥有一个布局管理器，在左边的 Panel 上使用 GridLayout 放置三个按钮，在右边的 Panel 上使用 CardLayout 来放置卡片，最后在窗口上使用 BorderLayout 放置这两个面板。CardLayout 容器中带有 5 张卡片(用 5 个按钮模拟)，按下 prev 按钮，依次向前显示，按下 next 按钮，依次向后显示，按下 three 按钮，显示第三张卡片。下面是程序代码：

程序清单：TestCardLayout.java

```
import java.awt.*;
import java.awt.event.*;
public class TestCardLayout
{
    CardLayout cl = new CardLayout();
    Panel plCenter = new Panel();
    public static void main(String [] args)
    {
        new TestCardLayout().init();
    }
    public void init()
    {
        Frame f=new Frame("布局管理器");
        Panel plWest = new Panel();
        f.add(plWest,"West");
        plCenter.setLayout(cl);
        plCenter.add("one",new JButton("one"));
        plCenter.add("two",new JButton("two"));
        plCenter.add("three",new JButton("three"));
        f.add(plCenter,"Center");
        f.pack();
        f.setVisible(true);
    }
}
```

```

f.add(plCenter);

plWest.setLayout(new GridLayout(3,1));
Button btnPrev = new Button("prev");
plWest.add(btnPrev);
Button btnNext = new Button("next");
plWest.add(btnNext);
Button btnThree = new Button("three");
plWest.add(btnThree);

plCenter.setLayout(cl);
plCenter.add(new Button("One"), "1");
plCenter.add(new Button("two"), "2");
plCenter.add(new Button("three"), "3");
plCenter.add(new Button("four"), "4");
plCenter.add(new Button("five"), "5");

class MyActionListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        if(e.getActionCommand().equals("prev"))
            cl.previous(plCenter);
        else if(e.getActionCommand().equals("next"))
            cl.next(plCenter);
        else if(e.getActionCommand().equals("three"))
            cl.show(plCenter, "3");
    }
}
MyActionListener ma = new MyActionListener();
btnPrev.addActionListener(ma);
btnNext.addActionListener(ma);
btnThree.addActionListener(ma);

f.setSize(300,300);
f.setVisible(true);
}
}

```

## 9.2.6 GridBagLayout

前面讲到的布局管理器的功能还是相当有限的，只能满足我们一些简单的需求。在复杂的布局要求下，我们需要使用 GridBagLayout 布局管理器。GridBagLayout 有布局管理器之王的说法，其功能非常强大，使用时也比较复杂，读者可以在 JDK 文档中了解到其详细说明及例子程序，考虑到一般的读者很少会使用到这种布局管理器，并且在 Swing 中我们可以有

更简单的办法来实现 GridLayout 布局管理器的功能，我们就不在这作更多的介绍了。如果你真的要进行这种复杂的布局设计，建议你使用 JBuilder 这样的集成开发环境来帮你完成。

### 9.2.7 取消布局管理器

我们也可以用绝对坐标的方式来指定组件的位置和大小，在这种情况下，我们首先要调用 Container.setLayout(null)方法取消布局管理器设置，然后调用 Component.setBounds 方法来设置每个组件的大小和位置。下面是不使用布局管理器的例子程序：

程序清单：TestNullLayout.java

```
import java.awt.*;
import java.awt.event.*;
public class TestNullLayout extends WindowAdapter
{
    TestNullLayout()
    {
        Button b1=new Button("第一个按钮");
        Button b2=new Button("第二个按钮");
        b1.setBounds(10,30,80,30);
        b2.setBounds(60,70,100,20);

        Frame f=new Frame("TestNullLayout");
        f.addWindowListener(this);
        f.setLayout(null);

        f.add(b1);
        f.add(b2);
        f.setBounds(0,0,200,200);
        f.setVisible(true);
    }
    public static void main(String[] args)
    {
        new TestNullLayout();
    }
    public void windowClosing(WindowEvent e)
    {System.exit(0);}
}
```

不使用布局管理器的一个潜在问题是，当容器大小改变，所有组件仍保持原来的位置和大小，将导致整个界面比较“难看”。如图 9.16 所示：



图 9.16

我们可以通过编程的手段来解决这个问题，一是限定用户改变容器的大小，二是容器大小改变时，容器上所有的组件按同等比例改变自己的大小和位置。

如果我们拥有一个类似 JBuilder 的集成开发环境，它都会为我们提供“所见即所得”的布局功能。我们可以在设计阶段通过图形操作界面，用鼠标直接在容器上拖动组件和拉动组件的边框，来指定容器上各个组件的绝对位置和大小。通过这种集成开发环境，我们能够方便的设计出用布局管理器难以实现的各种布局结构。所以，我们对布局管理器只要有个大致了解，明白其中的概念和属性就行了，在实际的 GUI 程序开发中，一般都是在诸如 JBuilder 的集成开发环境中进行的。

## 9.3 Swing

### 9.3.1 Swing 介绍

图形用户接口（GUI）库最初的设计目标是让程序员构建一个通用的 GUI，使其在所有平台上都能正常显示。但遗憾的是，AWT 产生的是在各系统看来都同样欠佳的图形用户接口。Java1.2 为老的 Java 1.0 AWT 添加了 Java 基础类（AWT），这是一个被称为“Swing”的 GUI 的一部分。Swing 是第二代 GUI 开发工具集，AWT 采用了与特定平台相关的实现，而绝大多数 Swing 组件却不是。Swing 是构筑在 AWT 上层的一组 GUI 组件的集合，为保证可移植性，它完全用 Java 语言编写，和 AWT 相比，Swing 提供了更完整的组件，引入了许多新的特性和能力。Swing 提供更多的组件库，如：JTable，JTree， JComboBox。Swing 也增强了 AWT 中组件的功能，这些增强的组件在 Swing 中的名称通常都是在 AWT 组件名前增加了一个“J”字母，如 awt 中的 Button 在 Swing 中是 JButton。

### 9.3.2 从 AWT 过渡到 Swing

在 Java 里用来设计 GUI 的组件和容器有两种，一种是早期版本的 AWT 组件，在 `java.awt` 包里，包括 `Button`、`CheckBox` 等，这些组件都是 `Component` 类的子类。

另一种是较新的 Swing 组件，在 `javax.swing` 包里，这些组件是 `Jcomponent` 类的子类。

我们看看如何将第八章开始的第一个程序中的 `java.awt.Button` 用 `java.Swing.JButton` 来替换。

程序清单：`TestFrame.java`

```
import java.awt.*;
import javax.swing.*;
public class TestFrame
{
    public static void main(String [] args)
    {
```

```

        Frame f=new Frame("IT 人资讯交流网");
        f.add(new JButton("ok"));
        f.setSize(300,300);
        f.setVisible(true);
    }
}

```

编译运行的结果如图 9.17 所示：



图 9.17

我们只是在原来程序的基础上增加了 `import javax.swing.*;` 语句和在原来程序中的“Button”前面增加了一个“J”字母。可见，Swing 在应用原理和方式上与 Awt 并没有多大的区别，我们学会了使用 AWT，基本上也就会了 Swing，关键是要学会自己查 JDK 文档帮助。

### 9.3.3 JFrame

我们再对上面程序作更进一步的改变，将其中的 `java.awt.Frame` 改为 `javax.swing.JFrame`，下面是修改过的程序代码。

```

import javax.swing.*;
public class TestFrame
{
    public static void main(String [] args)
    {
        JFrame f=new JFrame("IT 人资讯交流网");
        f.getContentPane().add(new JButton("ok"));
        f.setSize(300,300);
        f.setVisible(true);
    }
}

```

`JFrame` 与 `Frame` 的功能相当，但两者在使用上还是有很大的区别。我们不能直接在 `JFrame` 上增加子部件和设置布局管理器，而是必须先调用 `JFrame.getContentPane()` 方法

JFrame 中自带的 JRootPane 对象，JRootPane 是 JFrame 唯一的子组件，我们只能在这个 JRootPane 对象上增加子组件和设置布局管理器。当用户点击 JFrame 上的关闭窗口按钮时，JFrame 会自动隐藏这个框架窗口，但没有真正关闭这个窗口，这个窗口还在内存中，我们需要在 windowClosing 事件处理方法中，调用这个窗口对象的 dispose 方法来真正的关闭这个窗口。我们还可以调用 JFrame 的 setDefaultCloseOperation 方法，设置 JFrame 对这个事件的处理方式为 JFrame.EXIT\_ON\_CLOSE，当用户点击 JFrame 上的关闭窗口按钮时，直接关闭这个框架窗口并结束程序的运行。

### 9.3.4 JScrollPane

我们再来看看如何将第八章的 AWT 中的 ScrollPane 改为 swing 中的 JScrollPane:

程序清单：TestPane.java

```
import javax.swing.*;
import java.awt.event.*;
public class TestPane
{
    TestPane()
    {
        JFrame f=new JFrame("TestDialog");
        JScrollPane sp = new JScrollPane();
        JTextArea ta = new JTextArea(10,50);
        sp.setViewportView(ta);

        f.getContentPane().add(sp);
        f.setSize(200,200);
        f.setVisible(true);
        f.addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent e)
            {
                System.exit(0);
            }
        });
    }
    public static void main(String[] args)
    {
        new TestPane();
    }
}
```

### 9.3.5 对话框

我们在程序中经常要用一个简单的标准对话框来提示用户发生了什么事情，或者要求用户确认或取消一个动作。在 AWT 中，我们必须自己完全来实现这样的对话框界面和处理相关事件，Swing 为我们提供了一个 JOptionPane 类，JOptionPane 提供了若干个 showXxxDialog

静态方法，来帮我们完成这些功能。譬如，在程序开始启动时，弹出一个对话框提示用户，在用户关闭窗口时，我们询问用户是否真的要结束程序，下面是我们的程序代码。

```
程序清单：TestJDialog.java
import javax.swing.*;
import java.awt.event.*;
class TestJDialog
{
    public static void main(String [] args)
    {
        JOptionPane.showMessageDialog(null,"程序开始启动");
        final JFrame f = new JFrame("TestJDialog");
        //要被内置类访问，所以定义成 final
        f.setDefaultCloseOperation(
            WindowConstants.DO_NOTHING_ON_CLOSE);
        //Container c = f.getContentPane();
        f.addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent e)
            {
                int retval = JOptionPane.showConfirmDialog(f,
                    "你真的要结束吗？",
                    "结束程序",JOptionPane.YES_NO_OPTION);
                if(retval == JOptionPane.YES_OPTION)
                    System.exit(0);
            }
        });
        f.setSize(200,200);
        f.setVisible(true);
    }
}
```

在 Swing 中，有一个 JFileChooser 类专门用来实现文件存取对话框，它的使用非常简单，下面是从 JDK 文档中随手拷贝过来的一段代码，供读者参考：

```
JFileChooser chooser = new JFileChooser();
int returnVal = chooser.showOpenDialog(parent);
if(returnVal == JFileChooser.APPROVE_OPTION) {
    System.out.println("You chose to open this file: " +
        chooser.getSelectedFile().getName());
}
```

其实，如果读者习惯了去查看 JDK 文档，很多东西，我都只需提示一下就行的。如果大家想深入了解 JFileChooser 是如何设置文件列表过滤器的，可以仔细去读读（JDK 安装目录）\Demo\jfc\FileChooserDemo 目录下的源程序。

### 9.3.6 计算器界面的程序实现

我们通过实现一个如图 9.18 所示的计算器的界面，来看看如何使用 Swing 中的组件。

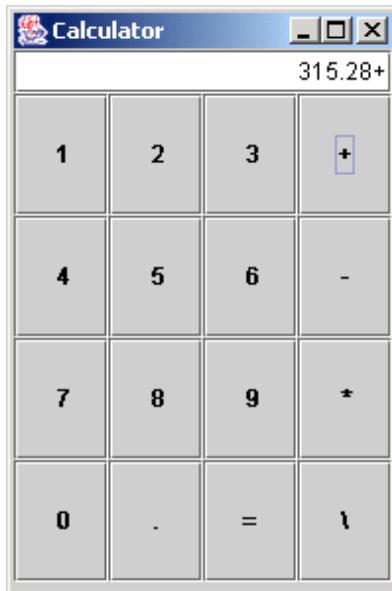


图 9.18

这个程序需要联合两个布局管理器，首先是将这 16 个按钮放在一个使用了 GridLayout 布局管理的 Panel 上，然后将这个 Panel 和文本框放在使用 BorderLayout 布局管理的主框架窗口上。文本框中的文字需要右对齐显示，才符合我们的习惯，而文本框中的文字默认是左对齐显示的，在 AWT 中的 TextField 是没有办法解决这个对齐问题的，但在 Swing 中的 JTextField 中，我们发现有一个 setHorizontalAlignment 方法可以解决这个问题。程序代码如下：

程序清单：Calculator.java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class Calculator implements ActionListener
{
    JFrame jf = new JFrame("Calculator");
    JTextField tf = new JTextField();
    public void init()
    {
        Container c = jf.getContentPane();
        tf.setHorizontalAlignment(JTextField.RIGHT);
        c.add(tf, "North");

        JPanel pnl=new JPanel();
        c.add(pnl, "Center");

        pnl.setLayout(new GridLayout(4,4));
        JButton b=new JButton("1");
        b.addActionListener(this);
        pnl.add(b);
        b=new JButton("2");
```

```
b.addActionListener(this);
pnl.add(b);
b=new JButton( "3" );
b.addActionListener(this);
pnl.add(b);
b=new JButton( "+" );
b.addActionListener(this);
pnl.add(b);
b=new JButton( "4" );
b.addActionListener(this);
pnl.add(b);
b=new JButton( "5" );
b.addActionListener(this);
pnl.add(b);
b=new JButton( "6" );
b.addActionListener(this);
pnl.add(b);
b=new JButton( "-" );
b.addActionListener(this);
pnl.add(b);
b=new JButton( "7" );
b.addActionListener(this);
pnl.add(b);
b=new JButton( "8" );
b.addActionListener(this);
pnl.add(b);
b=new JButton( "9" );
b.addActionListener(this);
pnl.add(b);
b=new JButton( "*" );
b.addActionListener(this);
pnl.add(b);
b=new JButton( "0" );
b.addActionListener(this);
pnl.add(b);
b=new JButton( "." );
b.addActionListener(this);
pnl.add(b);
b=new JButton( "=" );
b.addActionListener(this);
pnl.add(b);
b=new JButton( "\\" );
b.addActionListener(this);
pnl.add(b);
```

```

        jf.setSize(200,300);
        jf.setVisible(true);
    }

    public void actionPerformed(ActionEvent e)
    {
        tf.setText(tf.getText()+e.getActionCommand());
    }
    public static void main(String [] args)
    {
        new Calculator().init();
    }
}

```

### 9.3.7 BoxLayout 布局管理器

BoxLayout 是在 Swing 中新增加的一种布局管理器，它允许多个组件全部垂直摆放或全部水平摆放。嵌套组合多个使用 BoxLayout 布局管理器的 Panel，可以帮助我们实现类似 GridBagConstraints 的功能，但却要比直接使用 GridBagConstraints 简单许多。

作者在本章的讲解都只是告诉了大家怎样开发 GUI 的程序，重在向读者讲解一些基本的原理和开发技巧和思想，不可能将所有的组件都拿出来介绍。Swing 中的某些组件，在使用上与对应的 AWT 中的组件还有些区别，如果读者想从事专业的 GUI 程序开发，应尽量使用 Swing 下的组件，放弃 Awt 组件，以达到统一的效果。建议先在 JDK 文档中查阅 Swing 包，通读一下其中所有的组件，也许一个在 AWT 下实现起来有些费劲的功能，在 Swing 早就有一个很简单的组件已经可以帮你轻松搞定了。如果你要实现特殊的 GUI 功能和效果，你需要去临时掌握的一些特殊的组件，关于如何使用这些特殊的组件，最好和最快的办法就是，能够找到很好地使用了这些组件的例子程序，这时候，读者可以读读 JDK 下的一些 demo 程序，或是到 SUN 公司的网站上下载<<Java 指南>>(The Java Tutorial)，从中来了解那些组件的使用方法，参照别人成功的方式，这才是我们对 GUI 组件，甚至所有编程语言的学习之道。

|                           |     |
|---------------------------|-----|
| 第 9 章 GUI (下) .....       | 268 |
| 9.1 常用 AWT 组件.....        | 268 |
| 9.1.1 Component 类.....    | 268 |
| 9.1.2 Canvas.....         | 268 |
| 9.1.3 Checkbox .....      | 271 |
| 9.1.4 Choice .....        | 273 |
| 9.1.5 菜单 .....            | 274 |
| 9.1.6 Container 类.....    | 277 |
| 9.1.7 Window 类 .....      | 278 |
| 9.1.8 Dialog.....         | 278 |
| 9.1.9 Panel 类.....        | 281 |
| 9.1.10 ScrollPane 类 ..... | 281 |
| 9.2 布局管理器.....            | 282 |

|       |                      |     |
|-------|----------------------|-----|
| 9.2.1 | 了解布局管理.....          | 282 |
| 9.2.2 | BorderLayout .....   | 283 |
| 9.2.3 | FlowLayout .....     | 284 |
| 9.2.4 | GridLayout .....     | 285 |
| 9.2.5 | CardLayout .....     | 285 |
| 9.2.6 | GridBagLayout .....  | 287 |
| 9.2.7 | 取消布局管理器.....         | 288 |
| 9.3   | Swing .....          | 289 |
| 9.3.1 | Swing 介绍 .....       | 289 |
| 9.3.2 | 从 AWT 过渡到 Swing..... | 289 |
| 9.3.3 | JFrame.....          | 290 |
| 9.3.4 | JScrollPane.....     | 291 |
| 9.3.5 | 对话框.....             | 291 |
| 9.3.6 | 计算器界面的程序实现.....      | 292 |
| 9.3.7 | BoxLayout 布局管理器..... | 295 |

# 第 10 章 Applet

## 10.1 浏览器怎样显示网页

在学习 Applet 之前，我们先来看看浏览器是如何显示网页的。浏览器打开网页文件的过程同我们用记事本程序打开文本文件的过程是一样的，只是浏览器会对这个网页文件中的内容用特殊的方式显示。浏览器除了从本地硬盘上打开网页文件外，还可以从网络上的 WWW 服务器上打开网页文件。网页文件就是一个普通的文本文件，这个文本文件里的一些特殊字符序列被当作一种 html 标记，浏览器打开网页文件时，不是象记事本程序那样简单地显示文本文件里的内容，而是对这些标记作特殊处理。

我们先用一个实验来看看这种效果，用记事本程序创建一个 test.txt 文本文件，文件内容如下：

```
<marquee><font size=30 color=red>www.it315.org</font></marquee>
```

编写完上述代码后，我们将它存盘并将文件名改为 test.htm。然后用 IE 浏览器打开这个文件，我们可以看到在浏览器中显示的字体大小为 30 像素，颜色为红色，内容为 “www.it315.org” 的文字串在不停地水平滚动，但放在尖括号 (<>) 中的字符序列，如<marquee>、<font …> 并没在浏览器中显示出来，这些放在尖括号 (<>) 中的字符序列就是 html 标签。我们点击 IE 浏览器上的“查看” à “源文件” 菜单，又能够看到 test.htm 中的原始文本内容。所以说浏览器的基本功能就是根据 html 标签的含意，用特殊的效果去显示标签对中所引用的文本内容。

## 10.2 浏览器处理网页脚本代码的过程

现在我们再在刚才用“查看” à “源文件” 菜单打开的 test.htm 原始文本文件中增加下面的内容：

```
<script language="vbscript">
<!--
option explicit
private strStatus
private intSpace
private intDir

sub window_onload()
    strStatus= "www.it315.org"
    intSpace=0
    intDir=1
    window.setTimeout "Scroll",100
end sub
sub Scroll()
    dim strTemp
    intSpace=intSpace+1*intDir
    if intSpace>40 or intSpace<=0 then
        intDir=-1*intDir
    end if
    strTemp=string(intSpace, " ")

```

```

window.status=strTemp & strStatus
window.setTimeout "Scroll",100
end sub
-->
</script>

```

编辑结束后，我们存盘并刷新 IE 浏览器，重新显示 test.htm。这时我们能够看到在浏览器的状态栏上，“[www.it315.org](http://www.it315.org)”这几个文字在来回跑动。当 IE 浏览器碰到上面这段文本时，发现这是一段 vbscript 程序代码，便解释执行这段程序代码。注意，浏览器要解释执行这段程序代码，就必需内嵌有 vbscript 解释器，如果有些浏览器不具有 vbscript 解释器，就不能正确处理上面这段文本。

### 10.3 浏览器怎么处理 Applet

了解了上面这些内容，我们就比较容易理解什么是 Applet 了。简单的说，Applet 就是在 www 浏览器中执行的 Java 程序。

首先，编写如下的一个 Java 程序。

```

import java.applet.*;
import java.awt.*;
public class MyApplet extends Applet
{
    public void init()
    {
    }
    public void paint(Graphics g)
    {
        g.drawString("this is a simple applet",50,60);
    }
}

```

编译这个程序，并将生成的 `MyApplet.class` 文件与 `test.htm` 放在同一目录(这里我们放在 c 盘的 `test` 目录下)。修改 `test.htm` 的源文件，在最后处增加下面一行文本。

```
<applet code=MyApplet width=300 height=300></applet>
```

当浏览器碰到这行文本时，就知道要在网页的相同目录中装载一个名为 `MyApplet` 的 Java 类，这里的 `MyApplet` 在浏览器中占据 `300*300` 的显示和运行空间，如图 10.1 所示。



图 10.1

在上面这种情况下应用的Java程序就是Applet,与通常的Java程序不同,一个Applet小应用程序的执行不是从main()开始的,Applet小应用程序用一种与普通应用程序完全不同的机制来启动和执行,我们在后面的部分将对这种机制进行详细介绍。Applet小应用程序是一个GUI程序,在Applet小应用程序窗口中打印字符文本,并不是由函数System.out.println()完成的,而是由各种不同的AWT方法来实现,例如drawString(),这个方法可以向窗口的某个由X,Y坐标决定的特定位置输出一个字符串。同样的,Applet小应用程序接收数据输入也是按GUI方式进行的。

同网页中的script一样的道理,浏览器要正确显示并运行MyApplet,必需内嵌Java解释器。Applet的引用语句嵌入到HTML页面中,Applet在浏览器解释这段代码时被下载到客户端。

Applet是一个Java类,不同于其他的Java类。Applet是按下面的过程执行的。

- | 浏览器载入要访问的HTML文件的URL地址。
- | 浏览器载入HTML文件
- | 浏览器载入Applet的类字节代码
- | 启动Java虚拟机执行Applet。

一向喜欢玩弄垄断手法的微软曾经就去掉了IE浏览器中内嵌Java解释器,导致使用IE浏览器的用户没法正常显示带有Applet的网页,这样就会导致没人再使用Applet,以此报复老对手Sun公司,最后又引发了一场官司,最终以Sun获胜,Microsoft又不得不在最新的IE浏览器中重新内嵌Java解释器,以支持Applet。

如果读者使用某些版本的IE浏览器,可能会碰到Applet显示不了的情况,这是因为这个版本的IE浏览器中没有内嵌Java解释器的缘故。我们在安装JDK1.4的时候,会出现如图10.2所示的安装界面,意思是选择Java Plug-in软件来支持浏览器,就用来解决这种在浏览器中解释执行Applet的问题的。



图 10.2

有的时候我们正确的安装了 JDK，并且也选择了 Java Plug-in 软件，但是在 IE 浏览器中仍然不能正常显示 Applet 小程序。就让我们运行先前编写的 test.htm 来显示这种错误吧，如图 10.3 所示。

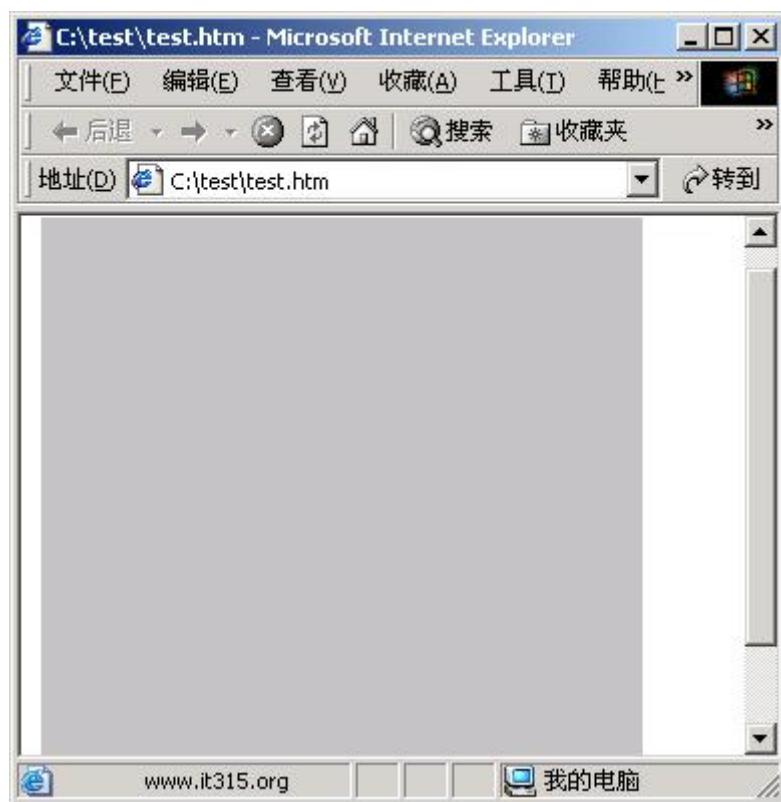


图 10.3

要解决这种问题其实很简单，我们只需要在 IE 浏览器的“工具”→“Internet 选项”→“高级”中选中“使用 Java2 用于 Applet”就可以了，如图 10.4 所示。

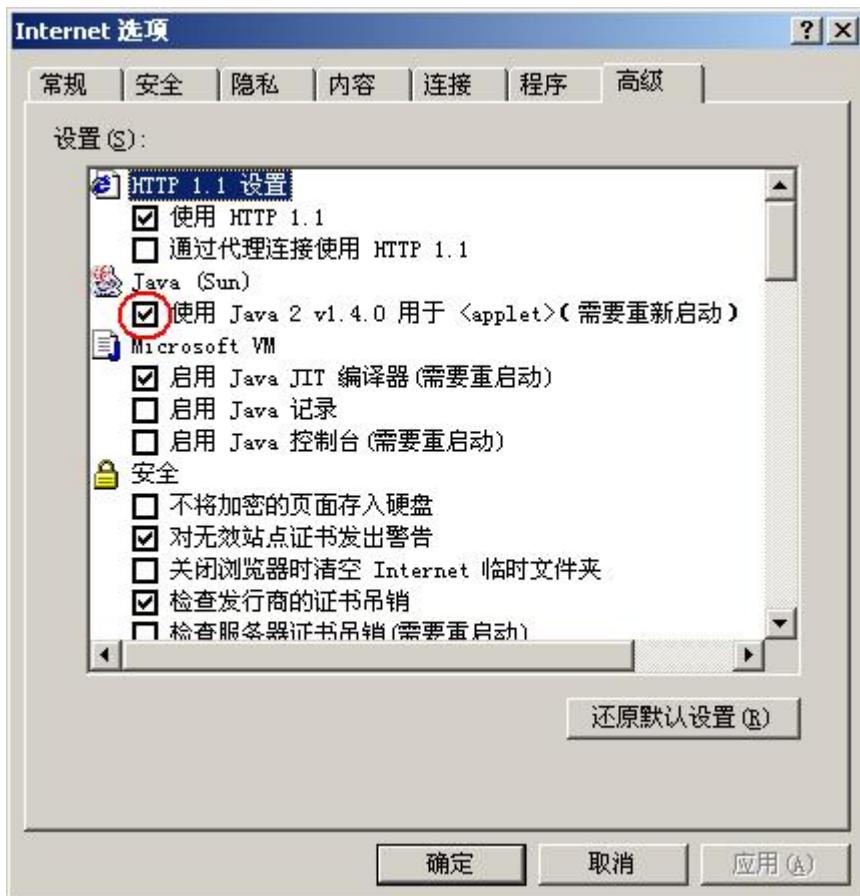


图 10.4

讲到这里大家应该对 Applet 有了一个基本的认识，接下来我们再来详细介绍 Applet 的类及其方法。

## 10.4 Applet 类及其方法

MyApplet 不是一个任意的 Java 类，必须是 `java.applet.Applet` 的子类。MyApplet 类就是一个 Applet，是具有图形用户界面的小程序，俗称 Java 小程序。

WWW 浏览器中内嵌 Java 解释器是按照下面的过程来使用 Applet 的。首先，WWW 浏览器装载网页并解释其中的内容，碰到 Applet 标签后，下载标签中指定的 Applet 类字节码并创建该类的实例对象。经过了 WWW 浏览器中内嵌 Java 解释器的处理，我们的浏览器就能够正常运行 Applet 小程序了。

几乎大多数的 Applet 小应用程序都重载一套方法，这些方法提供了浏览器或 Applet 小应用程序阅读器与 Applet 小应用程序之间的接口以及前者对后者的执行进行控制的基本机制。这套方法中的四个，`init()`, `start()`, `stop()` 和 `destroy()` 都是由 Applet 所定义的。另一个方法，`paint()` 是由 AWT 组件类定义的。Applet 小应用程序可以不重载那些它不想使用的方法，但是，只有非常简单的 Applet 小应用程序才不需要定义全部的方法。这五个方法组成了程序的基本主框架。

### 1. `init()` 方法

Applet 对象创建后，Java 会立即调用该对象的 `init()` 方法，以通知 Applet 对象进行初始化，尽管 Applet 小程序也可以像通常的类一样用构造方法进行初始化，但它习惯于在 `init` 方法中执行

所有的初始化。这个方法在 Applet 对象的生存期间只会被调用一次。我们可以按照如下格式书写方法 `init()`:

```
public void init()
{
    // Code here
}
```

### 2. `start()`方法

Java 在调用了 `init` 方法后, 会接着调用这个方法。在 WWW 浏览器每次离开创建此 Applet 对象的页面后, Java 所创建的 Applet 对象不会消失, 当 WWW 浏览器再回到创建此 Applet 对象的页面时, 又会调用这个方法。有些程序的功能在只有网页当前被显示时才要保持运行, 而浏览器离开此网页时应停止运行, 以节省浏览器的资源开销。`start` 方法中比较适合于放置这种功能的启动代码, 通常都与下面的 `stop` 方法配合使用。如网页正常显示时打开一个数据库连接或是启动一个音乐播放线程, 在浏览器离开该网页时应关闭数据库连接或是停止音乐播放线程。我们可以按照如下的方法书写方法 `start()`:

```
public void start()
{
    // Code here
}
```

### 3. `stop()`方法

终止和启动是成对出现的。在 WWW 浏览器每次离开创建此 Applet 对象的页面后, 去访问另一个页面时, Java 会调用 Applet 的 `stop` 方法。当 `stop()` 被调用时, 小应用程序还在运行, 你应该使用 `stop()` 来停止只有网页当前被显示时才要保持运行的功能。这样在 `stop` 方法中就可以停止先前由 `start` 方法启动的功能, 此时当用户回到此页面时, `start` 方法又会被调用, 你就可以重新启动先前已停止的功能。下面的例子显示了方法 `stop()` 的格式:

```
public void stop()
{
    // Code here
}
```

### 4. `destroy()`方法

当维护该 Applet 对象的 WWW 浏览器关闭时, Applet 对象也将被销毁。在 Applet 对象被销毁之前, `destroy` 方法会被调用, 该方法中通常用于释放 `init` 方法中初始化的资源, 调用该方法之前, 肯定已经调用了 `stop` 方法, 我们可以按如下的格式来书写方法 `destroy()`:

```
public void destroy()
{
    // Code here
}
```

## F 指点迷津:

你也许想知道 `destroy()` 与我们在第三章 “`finalize` 方法” 中介绍的 `finalize()` 究竟有什么不同。它们的区别在于, 方法 `destroy()` 只适合于 Applet (小应用程序); `finalize()` 是一个更加通用的方法, 它可用于任何类型的单个对象。

此外, Java 有一个自动的垃圾收集器来为你管理内存, 这也是我们前面讲过的内容。当程序使

用完资源之后，这个收集器会从这些资源中回收内存，但由于`finalize()`方法的不可靠，因此你通常需要使用`destroy()`方法，而不是用`finalize()`方法来释放资源。

### 5. `paint()`方法

我们先看看`Applet`类的继承层次关系

```
java.lang.Object
|
+---java.awt.Component
|
+---java.awt.Container
|
+---java.awt.Panel
|
+---java.applet.Applet
```

从上面的继承层次关系中，我们可以看到`Applet`是`Panel`的子类，因此具有`Panel`的所有功能，我们能够在上面添加AWT组件，显示图像，绘制图形，注册事件监听器。正因为`Applet`具有这种特性，我们在网页中使用`Applet`，就能让网页具有GUI程序的功能，这正是我们要使用`Applet`的原因。如在网页上显示一辆汽车，用户可以用鼠标拖动汽车旋转，从各个角度去欣赏这款汽车，也可以打开汽车的车门，查看汽车里面的结构。

就如我们在GUI章节中所讲的一样，在每一次你的小应用程序被重画后，`paint()`方法都被调用。这种情形的产生有几个原因。例如，小应用程序正在运行的窗口可能被另一个窗口覆盖，之后再恢复。或小应用程序窗口可能被缩小再还原。`paint()`方法也在小应用程序开始执行时被调用。不管是什原因，只要小应用程序必须重画窗口，`paint()`就被调用。`paint()`方法有一个`Graphics`类型的参数。这个参数包含了图像上下文，描述了小应用程序所运行的环境。在需要对小应用程序进行输出时，这个上下文都被用到。方法`paint()`看上去有如下格式：

```
public void paint(Graphics g)
{
    // Code here
}
```

## 10.5 一个显示动画的Applet的程序

了解了上面这些关于`Applet`的知识，我们接着修改上面的`MyApplet`类的源程序。新的`MyApplet`的主要功能是定时轮循显示十幅图像以产生动画效果，为了实现这个效果，我们首先必须要让程序具有定时功能。在这个程序中，我们是在一个新的线程中调用`Thread.sleep`方法来模拟定时器功能的，下面是`MyApplet`的源程序。

```
import java.applet.*;
import java.awt.*;
import java.net.*;
public class MyApplet extends Applet implements Runnable
{
    Image [] imgs=new Image[10];
    int index=0;
    public void init()
```

```

{
    try{
        for(int i=0;i<10;i++)
        {
            imgs[i]=getImage(new URL(getCodeBase(),"img\\T" +
                (i+1) +".gif"));
            /*imgs[i]=getToolkit().getImage(new URL(getCodeBase(),
                "img\\T" +(i+1) +".gif"));*/
        }
        new Thread(this).start();
    }
    catch(Exception e){e.printStackTrace();}
}

public void paint(Graphics g)
{
    g.drawImage(imgs[index],0,0,this);
    /*下面设置的字体必须是你的计算机中存在的字符,打开记事本程序的字体设置对话框,从其中可
    选的字体中复制一个到这里就行了.*/
    g.setFont(new Font("隶书",Font.ITALIC|Font.BOLD,30));
    index=(index+1)%10;
    g.drawString(" " + index,0,50);
}

public void run()
{
    while(true)
    {
        repaint();
        try
        {
            Thread.sleep(100);
        }catch(Exception e){}
    }
}
}

```

在上面的程序中，我们先用一个数组存储了要显示的十幅图像，接着启动了一个新的线程，新线程每隔 100 毫秒就会调用 Component.repaint 方法通知 Applet 重画，从而导致 paint 方法被调用，每一次 paint 调用都将显示变量 index 的值所对应的图像，程序中用 index=(index+1)%10; 这行代码让 index 变量的取值在 0-9 之间循环变化，也就实现程序所装载的十幅图像循环显示。当一个程序功能稍微有些复杂时，我们很难一气呵成，即使这样做了，万一程序不能按预期的结果运行，我们就不太好定位问题所在了。因此，作者在编写这个例子时，是分两个阶段来完成的，第一个阶段实现定时功能，运行后查看 index 的值是否每隔 100 毫秒在 0—9 之间循环变化显示，所有与图像有关的代码都没有编写。只有确认第一阶段的程序功能正常后，我们才进入第二个阶段，才开始编写

与图像有关的代码，实现图像显示。在第一个阶段，我们在 `paint` 方法中使用

```
g.drawString(" " + index, 0, 50);
```

这句程序代码，在 `Applet` 中打印出 `index` 的值以检查定时器功能是否正常。将整数转化成字符串除了用 `String.valueOf(整数)` 方法外，“”+ 整数（即用空字符串与整数相连），也是常用的一种方式。只有通过了第一阶段，我们才开始加入第二阶段的程序代码。

`Applet` 类自身就提供有一个 `getImage` 方法可以获得图像对象，我们在 AWT 中也使用过 `Component.getToolkit().getImage` 方法获得过图像对象，上面的程序中用注释给出了第二种方法在这里的应用，以供读者参考。要注意的是，网页与 `Applet` 类通常都是放在 WWW 服务器上的，`Applet` 所用到的资源（这里是图像文件）也应一并放在 WWW 服务器上，所以在这些的 `getImage` 方法中，我们不能用本地路径来指定图像文件的位置，如 `c:\img\T1.gif` 等，我们需用表示 Internet 网络资源路径的方式来指定图像文件的位置，也就是 URL。在讲解分析我们的程序代码之前，我们必须先对 URL 进行详细的讲解。

## 10.5.1 URL 类

URL（统一资源定位符，Uniform Resource Locator 的英文简写）用于表示 Internet 网络上资源的地址。URL 一般由协议名、资源所在的主机名和资源本身的名称等三个部分组成，例如下面的 URL：

<http://www.it315.org/home/welcome.html>

所用的协议是 http (Hypertext Transfer Protocol，超文本传输协议) 协议，资源所在的主机名为 [www.it315.org](http://www.it315.org)，资源名为 `/home/welcome.html`。

URL 还可以包含端口号来指定要连接的远端主机的端口，如果不指定端口号，则使用协议的默认端口号。例如，http 协议的默认端口号是 80，在上面这些 URL 例子中，使用的端口号就是 80。如果 http 服务器所用的端口号不是 80，而是 8080，在 URL 中要显式指定 http 服务器的端口号，指定端口号的 URL 形式如下：

<http://www.it315.org:8080/index.html>

URL 通常是大小写敏感的(除机器名外)，有些应用软件在开发时并没有太注意这个问题，有的可能是对整个 URL 都区分大小写，有的可能是 URL 的一部分区分大小写，有的对整个 URL 都不区分大小写。然而，要识别这些应用软件间的差异，并不容易，用户需要自己经常留意这个问题，并做到心中有数。

在许多应用中，我们也可以使用相对 URL。一个相对的 URL 不包括协议或主机地址信息，表示它的路径与当前文档（或其他的 Internet 资源）的访问协议、主机名相同，甚至有相同的目录路径。相对 URL 的使用，与我们本地计算机上的文件系统的相对目录的使用相似，用“/”开头，表示主机上的某种协议的根目录，用“.. /”开头表示当前资源所在目录的父目录，“... /”表示父目录的父目录。而直接使用目录名或文件名，表示当前目录下的子目录或文件，与用“./”开头的效果是一样的。

Java 在 `java.net` 包中专门为 URL 定义了对应的 URL 类。查看 JDK 文档帮助，其有多种形式的构造函数，我们的网页、`Applet` 以及这些图像文件可以放在不同网络地址的 WWW 服务器上的不同目录下，我们是没法确定这些图像文件的绝对路径的。但网页、`Applet` 以及这些图像文件的相对路径关系是必须固定的，而 `Applet` 类又提供了 `getCodeBase` 方法返回 `Applet` 的 URL，`getDocumentBase` 方法返回网页的 URL，我们便很容易想到用相对 URL 的方式来构造这些图像文件资源的 URL。按照这个想法，我们可以使用

```
public URL(URL context, String spec) throws MalformedURLException
```

这个构造函数来创建这些图像文件的 URL 对象。对于 `getCodeBase` 和 `getDocumentBase` 方法，我们到底用哪一个好呢？`Applet` 与网页的路径关系是可能变化的，也就是不一定在同一目录，这

取决于网页中<applet code=MyApplet codebase=...>标记的 codebase 可选属性的设置，如果 Applet 与网页在同一路径中，则不用设置 codebase 属性。图像文件与 Applet 类联系紧密，是作为一个整体提供的，所以，我们应该选用 getCodeBase 方法。关于<applet>标记的详细说明，请看本章后面的部分。读者可以试试将程序中的

```
imgs[i]=getImage(new URL(getCodeBase(),"img\\T"+(i+1)+".gif"));这行代码改为  
imgs[i]=getImage(new URL(getCodeBase(),"img\\T"+i+1+".gif"));看看运行结果有什么差别。
```

为了看到 MyApplet 的运行效果，在上面的例子中，我们的网页文件与 MyApplet.class 必须在同一文件目录下，该目录下还必需有个 img 的子目录，img 目录中保存有 T1.gif, T2.gif……T10.gif 等十副图像文件，作者用的是 JDK 中的例子程序所带的十副图像文件，它们位于：<jdk 安装目录>\demo\applets\Animator\images\Beans 目录下，将它们拷贝到 MyApplet.class 文件所在的目录下。用 WWW 浏览器打开网页文件，就可以看到 MyApplet 的运行效果了。

### 10.5.2 update 方法

读者可以回顾一下我们在 AWT 中讲过的部件重绘时的函数调用关系，如图 10.5 所示。

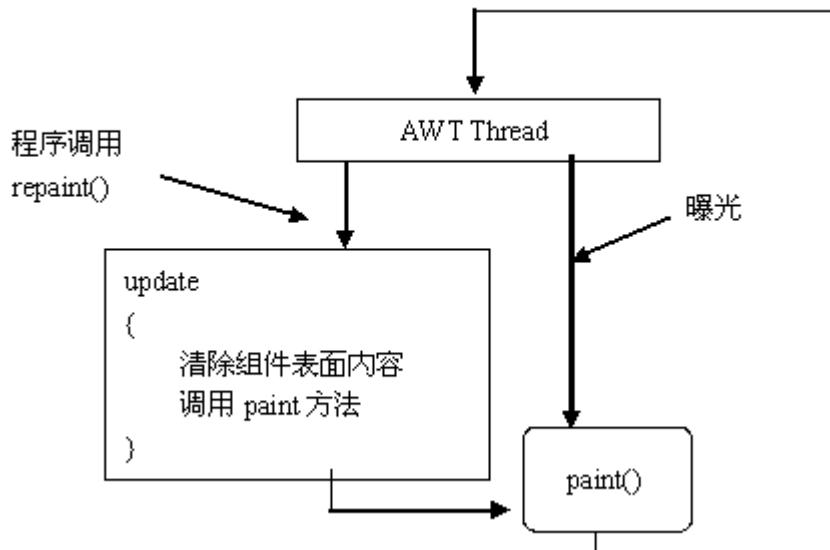


图 10.5

程序调用 repaint()方法导致的部件重绘过程中，AWT 线程调用 Component.update 方法，Component.update 再调用 paint 方法。我们正好可以借用这个程序来研究一下 repaint()方法的调用过程，来了解 Update 方法。

在 MyApplet 类中覆盖基类 Component 的 update 方法。MyApplet 类中的 update 方法的程序代码如下：

```
public void update(Graphics g)  
{  
}
```

这样，MyApplet 将又有怎样的运行效果呢？由于 MyApplet 覆盖了 Component.update 方法，按照对象的多态性，AWT 线程将调用 MyApplet 中的 update 方法，程序中的 update 方法什么代码也没有，不会再去调用 paint 方法了，所以没有动画显示的效果了。但窗口曝光时仍会调用 paint 方法，

所以第一幅图像和 index 的值还是应该被显示出来的。

重新编译 MyApplet 类，在 WWW 浏览器中刷新刚才打开的 MyApplet 网页，和先前运行的效果没有什么两样，MyApplet 仍然具有动画功能。这是因为在 WWW 浏览器中调用的还是程序修改前装载的那个 MyApplet 类，在前面讲过，浏览器关闭之前，java 所创建的 Applet 对象不会消失，当 WWW 浏览器再回到创建此 Applet 对象的页面时，不会再创建新的 Applet 对象。

关闭浏览器后，重新打开刚才的 MyApplet 网页，index 的值被打印出来了，但对应的图像并没有显示出来。index 的值被打印出来，并且在浏览器窗口最小化后再恢复正常化显示时，index 的值也变化了，这足以说明 paint 方法被调用了，也证明了图 10.5 中所示的曝光过程发生了，AWT 线程会直接调用 paint 方法。在 AWT 中讲过，drawImage 是一个异步函数，即使 img 对应的图像还没有完全装载，drawImage 也会立即返回。基于这些原理，我们试着刷新一下网页，这时，图像显示出来了。

我们再将 update 方法修改成下面这样，

```
public void update(Graphics g)
{
    paint(g);
}
```

MyApplet 的运行效果又会怎样呢？由于 update 再调用 paint 方法之前，没有清除部件表面原来的内容，所以，我们最终看到的效果是这十幅图像叠加在了一块。

通过上面对比讲解与实验，大家应该彻底明白 update 方法与 paint 方法的关系了。

## & 多学两招：

WWW 浏览器是 Applet 运行的容器，Applet 就是插件。插件的运行是完全由容器来控制和调度的，容器才是真正的应用程序，插件只是提供约定的函数方法供容器调用。容器与插件的程序架构设计模式，在软件开发中的应用非常广泛，如 WWW 服务器中的 JSP 技术，J2EE 中的 EJB 技术等。读者不仅要理解 Applet，掌握编写和运用 Applet 的过程，还应仔细体会容器与插件的这种程序架构设计模式，以便自己在以后的程序设计中借鉴这种思想。

读者对 Applet 已经有了一定的认识，也就是大家基本上明白了什么是 Applet 以及如何编写 Applet，如果你不想从事专业的 Applet 开发或者时间有限，可以跳过本章后面的部分，等到以后有需要时再回过头来看。现在我们再来继续讲解 Applet 中的一些细节问题。

# 10.6 关于 Applet 的一些细节

## 10.6.1 Applet 的运行环境

Applet 与 Application 的区别主要在于其执行方式的不同，Application 是从其中的 main() 方法开始运行的，一个 Applet 是无法单独运行的，必须在一个 HTML 页面中被引用，由 WWW 浏览器下载并调用其中的方法。

我们也可以用 JDK 自带的命令行工具 appletviewer 打开 HTML 文件运行 Applet 程序，还是以本章最开始的 test.htm 文件为例。

首先我们进入命令行窗口中，转入 test.htm 所在目录并运行：

```
appletviewer test.htm
```

运行结果如图 10.6 所示：

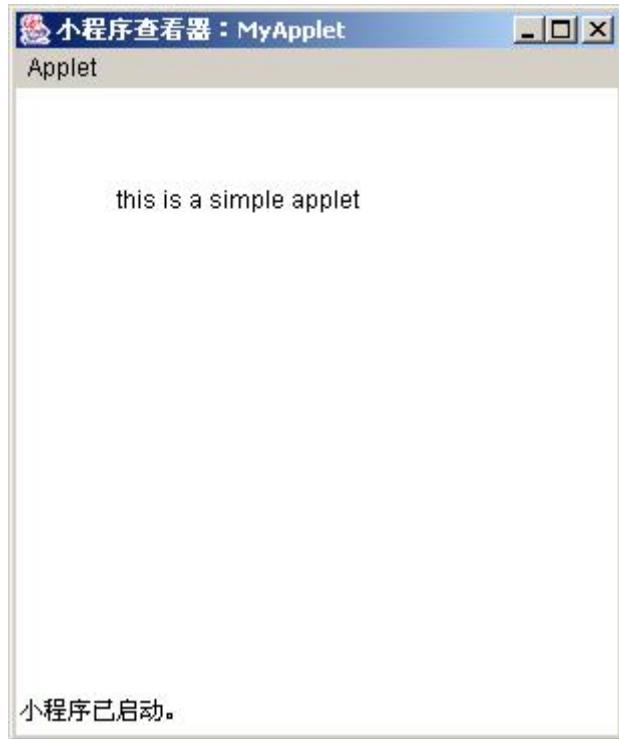


图 10.6

### 10.6.2 Applet 程序中如何使用控制台输出

我们使用 `System.out.println()` 语句是不能够在图形窗口中打印字符文本的，只有使用 `drawString()` 语句才能够在图形窗口中打印字符文本。我们知道，在程序中使用 `System.out.println()` 语句打印的字符文本会显示在命令行窗口中，如果我们用的是 appletviewer 执行 Applet 小程序，`System.out.println()` 语句的输出会显示在启动 appletviewer 程序的命令行窗口中。IE 浏览器也提供了专门用于显示 `System.out.println()` 语句的输出的机制，提供了一个专门的小窗口充当 Java 控制台。如果执行 Applet 小程序的解释器是 IE 浏览器自带的 Java 解释器，我们需要设置 IE 浏览器的选项，打开控制台窗口来显示 Applet 小程序中的 `System.out.println()` 语句的输出。如图 10.7 所示。

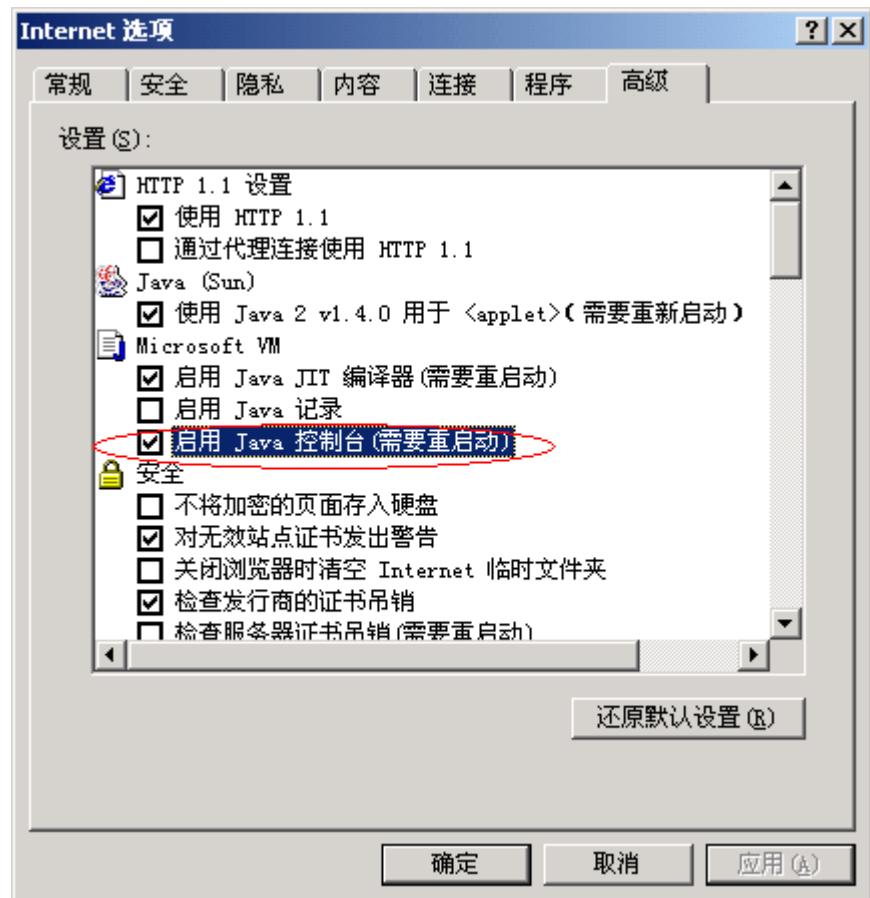


图 10.7

这样，当我们在程序中使用了 `System.out.println()` 语句时，在桌面上任务栏的右下角就会出现一个 Java 的小图标，双击此图标就能够看到 Java 控制台中显示的 `System.out.println()` 语句打印的字符文本，使用 `System.out.println()` 语句和控制台窗口在我们调试 Applet 时非常有用。如果程序中使用了 `System.out.println("this is an Applet example");` 语句，控制台窗口的结果如图 10.8 所示：

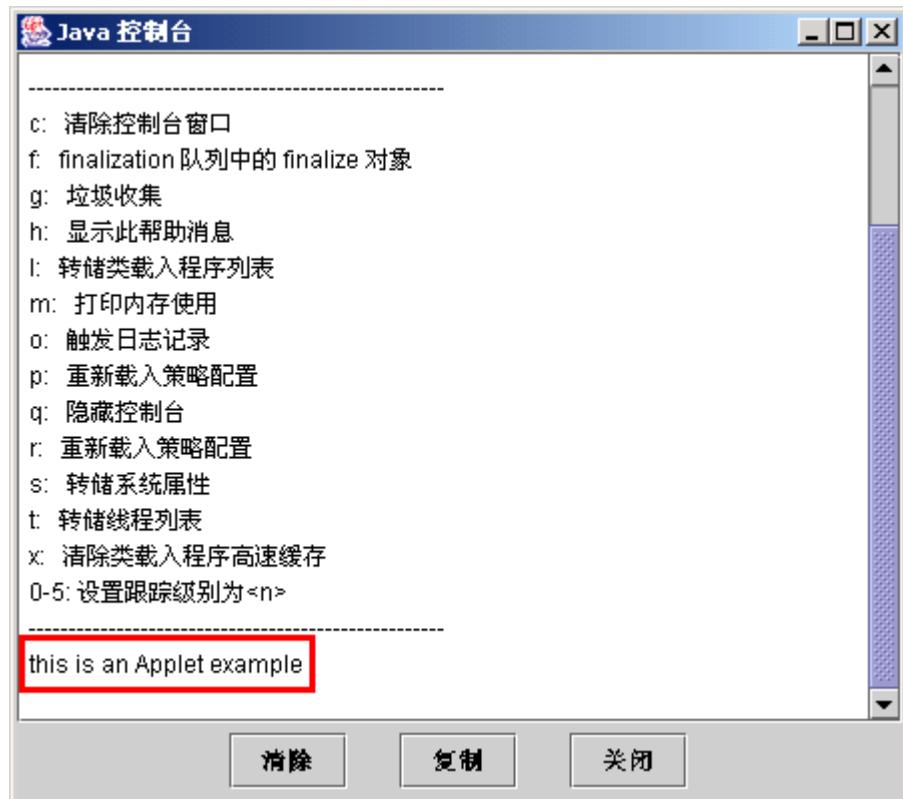


图 10.8

### 10.6.3 Applet 程序中需要注意的问题

一般情况下，Applet 执行时受下面的限制：

- | 不能调用其它的应用程序执行
- | 不能进行文件 I/O 操作
- | 不能调用本机代码
- | 不能与 Applet 所在的主机之外的其它机器进行网络连接

### 10.6.4 Applet 标记

APPLET标记被用来从HTML文件和小应用程序阅读器中启动一个小应用程序（新的OBJECT标记也有此种功能）。

一个小应用程序阅读器将执行它在一个单独窗口中所发现的每一个APPLET标记，而Internet Explorer等Web浏览器将允许在一个网页中有多个小应用程序。到目前为止，我们仅仅使用了APPLET标记的简单形式，现在我们来看一看标准的APPLET标记的更详细语法，如下所示。

```
<applet  
    [archive=archiveList]  
    [code=appletFile.class]  
    width=pixels height=pixels  
    [codebase=codebaseURL]  
    [alt=alternateText]  
    [name=appletInstanceName]
```

```

[align=alignment]
[vspace=pixels] [hspace=pixels]
>
[<param name=appletAttribute1 value=value>]
[<param name=appletAttribute2 value=value>]
. .
[alternateHTML]
</applet>

```

我们来看一看每一部分的含义：

- **archive = archiveList**

用来指示 Applet 执行前被预先载入的类字节代码或者是其它资源的压缩文件 (\*.jar 文件)。

当你创建了一个 jar 文件之后，ARCHIVE 属性可以和<APPLET>标记一起使用，这样来显示在哪里可以找到这个存档文件。具体来说我们可以用如下的格式使用带有标记的小应用程序的 jar 文件：

```

<applet code="MyApplet.class" archive="MyApplet.jar" width=50 height=50>
</applet>

```

这个标记指明了一个叫做 MyApplet.jar 的文件，它包含了这个小应用程序所使用的各个文件。当我们的浏览器碰到上述代码后，浏览器就会在这个 jar 文件中查找运行该小应用程序所需的文件。

- **code = appletFile.class**

指定要执行的Applet代码。code是一个必不可少的属性，它给定了你的小应用程序编译过的 class文件的名字。这个文件是与Applet的URL相关的，它是HTML文件所在的路径或由codebase 所说明的路径（如果被设定的话）。

- **width = pixels height = pixels**

制定 Applet 执行时的初始显示大小。

- **codebase = codebaseURL**

codebase指定了小应用程序代码的基本URL，从而表明小应用程序的可执行的类文件(由code标记所指定)的查询路径。如果这项属性没有被指定的话，则HTML文件的URL路径就被用作codebase。codebase不一定要在HTML文档所在的主机上。

- **alt = alternateText**

Applet 不能正常执行时显示的替代文本。

- **name = appletInstanceName**

Name用来指定小应用程序实例的名字，通过Applet的命名可以使同一页上的所有的小应用程序能够互相找到并通信。使用Applet的getApplet()方法可以通过名字获得相同网页中的另外一个Applet小程序实例对象。

- **align = alignment**

Applet 显示的对齐方式，如：left, right, top, texttop, middle, absmiddle, baseline, bottom, and absbottom。

- **vspace = pixels hspace = pixels**

Applet与显示边框之间的垂直、水平间隔。VSPACE规定了小应用程序的以像素为单位的上下空间大小

- **<param name = appletAttribute1 value = value>**

标记 Applet 执行时传递给它的参数，由 name 指定参数名，value 指定参数值。

详细介绍了标准的 Applet 标记的语法之后，我们再来看一个在 HTML 和 Applet 之间传递参数的例子

```

<html>
<applet code="Parameters.class" width=200 height=200>
<param name=speed value="12">
<param name=distance value="500m">
</applet>
</html>

```

在这段 html 代码中，为小应用程序 Parameters 定义了两个参数：一个名叫 speed，值为 12，另一个名为 distance，值为 500m。

现在我们就来看看下面的程序是如何取得参数和属性的。

```

import java.applet.*;
import java.awt.*;
public class Parameters extends Applet
{
    private String toDisplay;
    private int speed;
    public void init()
    {
        String pv;
        pv = getParameter("speed");
        if (pv == null)
        {
            speed = 10;
        }
        else
        {
            speed = Integer.parseInt (pv);
        }
        toDisplay = "Speed Parameter: " + speed;
    }
    public void paint(Graphics g)
    {
        g.drawString(toDisplay, 25, 25);
    }
}

```

当你的小应用程序被加载时，参数被传递到其中。在你的小应用程序的 init() 方法中，你可以通过使用方法 getParameter() 来取出这个参数。方法 getParameter 根据你传递进的参数名字，返回那个参数的对应值，这个返回的参数值也是字符串类型的。所以，程序中还刻意安排了将 getParameter() 方法取得 speed 的值转换成整数的语句。

## M 脚下留心：

在<PARAM>中指定的参数名与在 getParameter() 中的参数名必须完全匹配，如，<PARAM NAME= "distance" >与<PARAM NAME= "Distance" >是不同的。如果你的参数没有被正确地传递给你的小应用程序，请确认一下这个参数的大小写是否正确。

## 10.6.5 OBJECT 标记

OBJECT 标记属于 HTML4.0 标准，而 W3 协会建议人们用它来代替 APPLET 标记。OBJECT 标记有 35 个不同的属性，其中大多数只和编写动态 HTML 的人员有关。其中大量的定位标记，比如 ALIGN 和 HEIGHT 的用法完全像它们为 APPLET 标记所做的那样。OBJECT 标记中的关键属性是 CLASSID 属性，这个属性指定了 Java 解释器插件本身，这个插件不是网页中 Applet 小程序，而是执行 Applet 小程序的解释器，Applet 小程序被作为 OBJECT 的一个参数指定。

从<APPLET>到<OBJECT>的转换只需要在<APPLET>的位置放置<OBJECT>标记。

### APPLET 和 OBJECT 属性之间的转换

| APPLET     | OBJECT                          |
|------------|---------------------------------|
| ALT=…      | N/A                             |
| ARCHIVE=…  | <PARAM NAME="ARCHIVE" VALUE=…>  |
| CODE=…     | <PARAM NAME="CODE" VALUE=…>     |
| CODEBASE=… | <PARAM NAME="CODEBASE" VALUE=…> |
| OBJECT=…   | <PARAM NAME="OBJECT" VALUE=…>   |

Object 标记的使用还是非常复杂的，一般都不太容易记住其中的细节，但使用 Applet 标记，却相对容易得多。为此，Sun 公司的 JDK 中专门提供了一个 HTMLConverter 应用程序，它能把一个现有的 Web 页中的 Applet 标记转换成对应的 Object 标记。这样，我们在创建了一个 Applet 小程序后，在 Web 页中仍用 Applet 标记引用这个 Applet，之后，我们使用 HTMLConverter 将网页中的 Applet 标记转换成对应的 Object 标记，我们就不用去掌握 Object 中的细节知识了。

当使用 HTMLConverter 程序时，需要使用 HTML 文档的名字作为参数来进行转换。例如：

```
HTMLConverter test.htm
```

通过上述的命令，就会把 test.htm 中包含的所有 Applet 标记转化成 Object 标记。

对网页中的<applet code=MyApplet width=300 height=300></applet>部分，转换后对应的结果如图 10.9 所示：

The screenshot shows the UltraEdit-32 text editor window. The title bar reads "UltraEdit-32 - [C:\test\新建文件夹\test.htm]". The menu bar includes "文件(F)", "编辑(E)", "搜索(S)", "项目(P)", "视图(V)", "格式(T)", "列块(L)", "宏(M)", "高级(A)", "窗口(W)", and "帮助(H)". The toolbar contains various icons for file operations like Open, Save, Find, and Copy. The status bar at the bottom displays "如需帮助文件, 请按 F1", "行 1, 纵列 1, C0", "DOS", "修改: 2003-4-21 16:45:48", "大小: 1324", and "插入". The main editor area contains the following HTML code:

```
<!--"CONVERTED_APPLET"-->
<!-- HTML CONVERTER -->
<OBJECT
  classid="clsid:CAFEEFAC-0014-0000-0000-ABCDEFFEDCBA"
  WIDTH = 300 HEIGHT = 300
  codebase="http://java.sun.com/products/plugin/autodl/jinstall-1_4_0-win.cab#Version=1,4,0,0">
  <PARAM NAME = CODE VALUE = MyApplet >

  <PARAM NAME="type" VALUE="application/x-java-applet;jpi-version=1.4">
  <PARAM NAME="scriptable" VALUE="false">

  <COMMENT>
    <EMBED
      type="application/x-java-applet;jpi-version=1.4"
      CODE = MyApplet
      WIDTH = 300
      HEIGHT = 300
      scriptable=false
      pluginspage="http://java.sun.com/products/plugin/index.html#download"
      <NOEMBED>

        </NOEMBED>
    </EMBED>
  </COMMENT>
</OBJECT>

<!--
<APPLET CODE = MyApplet WIDTH = 300 HEIGHT = 300>

</APPLET>
-->

<!--"END_CONVERTED_APPLET"-->
```

图 10.9 (转换后)

由于转换后的文件格式是 Linux 操作系统的，它的换行符在 Windows 自带的文本编辑器中不能正常显示，我们建议大家使用 UltraEdit 这个工具软件来查看 HTMLConverter 程序转换的文档。

在 Object 的 classid 属性中指定了 Java 插件的代号（一个长长的数字序列），pluginspage 属性指定了这个插件的下载位置，如果你的机器上还没有安装过这个插件，在第一次使用这个插件时，浏览器会自动从 pluginspage 属性指定的位置下载插件并安装。如图 10.11、10.12 所示。



图 10.11



图 10.12

可见，使用 `Object` 标记比 `Applet` 标记多了自动下载和安装 Java 解释器（一种插件）的功能。不是每个上网的用户都是 Java 程序员，不是每个用户都会去主动安装 JDK 的，所以，我们需要某种机制去帮助那些想访问我们的 `Applet` 网页的用户自动安装 Java 解释器这个插件，这也就是我们为什么要使用 `Object` 标记的最根本的原因了。

## M 脚下留心：

HTMLConverter 工具重写了页面上已经存在的 HTML 代码，如果处于某种原因你还想使用原来的版本，你应该先备份这个 HTML 文件，然后再用 HTMLConverter 进行转换。

## 10.7 验证 Applet 对象在客户端如何存在的

我们在前面讲过，在 WWW 浏览器每次离开创建了某个 `Applet` 实例对象的页面后，Java 所创建的 `Applet` 对象不会消失，当 WWW 浏览器再回到创建此 `Applet` 对象的页面时，不用再创建新的 `Applet` 实例对象，会继续使用原来的那个 `Applet` 实例对象。如果两个网页中都用到了同一个 `Applet` 类，当用户访问了第一个网页后，这个 WWW 浏览器接着再去访问第二个网页时，是否还创建一个新的 `Applet` 实例对象，或是直接使用第一个网页创建的 `Applet` 实例对象？没有任何书籍和资料讲过这个问题，我们能否自己写点程序代码来解答心中的疑惑呢？我们知道，每创建一个 `Applet` 实例对象，这个 `Applet` 的构造方法和 `init` 方法就会被调用一次，不妨在 `Applet` 类中定义一个静态成员变量，

初始值为零，在 `init` 方法中将这个变量加 1，如果在 WWW 浏览器接着再去访问第二个网页时，这个变量的值仍为 1，说明 `init` 方法只被调用了一次，也即在浏览器中只产生了一个 Applet 实例对象；如果这个变量的值为 2，则说明 `init` 方法被调用了两次，即在浏览器中产生了两个 Applet 实例对象。为了保险起见，我们在构造方法中也进行了同样的操作，从更多的方面来观察我们的结果。

程序清单： `MyApplet.java`

```
import java.applet.*;
import java.awt.*;
public class MyApplet extends Applet
{
    static int count=0;
    static int count1=0;
    public MyApplet()
    {
        count1++;
    }
    public void init()
    {
        count++;
    }
    public void paint(Graphics g)
    {
        g.drawString(count + "," + count1, 50, 60);
    }
}
```

首先编译生成 `MyApplet.class` 文件，然后将本节中最开始编写的 `test.htm` 文件复制并改名为 `test1.htm` 和 `test2.htm`，首先打开 `test1.htm`，显示的结果是“1,1”，然后还是在这个浏览器窗口中打开 `test2.htm`，显示的结果为“2,2”，刷新一下 `test2.htm`，显示的结果为“3,3”。可见当不同的网页用到了同一个 Applet 类时，浏览器在访问每一个网页时，都会创建一个新的实例对象。我们在前面讲过，在 WWW 浏览器每次离开创建了某个 Applet 实例对象的页面后，Java 所创建的 Applet 对象不会消失，当 WWW 浏览器再回到创建此 Applet 对象的页面时，不用再创建新的 Applet 实例对象，而是再次调用这个 Applet 对象的 `start` 方法，尽管 Sun 公司的文档资料上是这么说的，并且作者在以前的 JDK 版本下的试验也确实如此。但我们这次的试验结果却又不是这样的了，我们仔细思考一下，我们每刷新一下网页，发现 `count` 与 `count1` 的值也随之加 1，这说明我们每次访问同一个网页时，都会重新创建一个新的实例对象，这里的试验结果显然与前面的说法相矛盾了。这种文档与实际相冲突的情况，我们在实际的项目开发中也经常碰到，那么在文档与试验结果之间，我们该做出怎样的抉择呢？作者的建议是：如果你能确信你的试验没有错误和考虑不周，你应该以你的试验结果为准，文档有时会过时和有疏漏的，你只能把文档当作参考资料，而不能当作真理。如果你有兴趣的话，也可以向 Sun 公司发送邮件，与他们探讨这个问题，一般的结果是，他们不小心疏漏了或是产品已经升级和修改了，但文档资料还没跟上。通过这个试验，我们重新认识了 `init()`, `start()`, `stop()`, `destroy()` 的调用时机，与先前讲的有些出入，作者通过更进一步的试验（在 `start` 方法中将变量加 1，在 `stop` 和 `destroy` 方法中将计数变量减 1 等），得出的结论如下：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。

通过上面的试验，读者应该更加相信，许多东西都不应死记硬背的，而是只能作为一种参考。

在 IT 领域，经常有这样的情况：“今天还是大伙都在宣扬的新的技术，明年的这个时候却成了历史的笑话”。这就是程序员的知识很容易老化的原因，程序员必须不停地学习新知识，抛弃老的东西，这样才不会被淘汰出局。

|                                  |     |
|----------------------------------|-----|
| 第 10 章 Applet.....               | 296 |
| 10.1 浏览器怎样显示网页.....              | 296 |
| 10.2 浏览器处理网页脚本代码的过程.....         | 296 |
| 10.3 浏览器怎么处理 Applet.....         | 297 |
| 10.4 Applet 类及其方法 .....          | 300 |
| 指点迷津：destroy() 与 finalize() 的区别  |     |
| 10.5 一个显示动画的 Applet 的程序.....     | 302 |
| 10.5.1 URL 类 .....               | 304 |
| 10.5.2 update 方法 .....           | 305 |
| 多学两招：学会容器与插件的设计思想                |     |
| 10.6 关于 Applet 的一些细节 .....       | 306 |
| 10.6.1 Applet 的运行环境 .....        | 306 |
| 10.6.2 Applet 程序中如何使用控制台输出 ..... | 307 |
| 10.6.3 Applet 程序中需要注意的问题 .....   | 309 |
| 10.6.4 Applet 标记 .....           | 309 |
| 脚下留心：参数名的大小写问题                   |     |
| 10.6.5 OBJECT 标记 .....           | 312 |
| 脚下留心：使用 HTMLConverter 工具的注意事项    |     |
| 10.7 验证 Applet 对象在客户端如何存在的 ..... | 314 |

# 第 11 章 网络编程

有人说，20世纪最伟大的发明并不是计算机，而是计算机网络。还有人说，如果你买了计算机而没有联网，就等于买了电话机却没有接电话线一样。

计算机网络就是实现了多个计算机互联的系统，相互连接的计算机之间彼此能够进行数据交换。正如城市道路系统总是伴随着城市交通规则来使用的道理，计算机网络总是伴随着计算机网络协议一起使用的。网络协议规定了计算机之间连接的物理、机械（网线与网卡的连接规则）、电气（有效的电平范围）等特性以及计算机之间的相互寻址规则、数据发送冲突的解决、长的数据如何分段传送与接收等。就象不同的城市可能有不同的交通规则一样，目前的网络协议也有多种，其中，TCP/IP 协议就是一个非常实用的网络协议，它是 Internet 所遵循的协议，是一个“既成事实”的标准，已广为人知并且广泛应用在大多数操作系统上，也可用于大多数局域网和广域网上。

网络应用程序，就是在已实现了网络互联的不同的计算机上运行的程序，这些程序相互之间可以交换数据。编写网络应用程序，首先必须明确网络程序所要使用的网络协议，TCP/IP 是网络应用程序的首选协议，大多数网络程序都是以这个协议为基础，本章关于网络程序编写的讲解，都是基于 TCP/IP 协议的。

## 11.1 网络编程的基础知识

### 11.1.1 TCP/IP 网络程序的 IP 地址和端口号

要想让网络中的计算机能够互相通信，必须为每台计算机指定一个标识号，通过这个标识号来指定要接收数据的计算机和识别发送数据的计算机，在 TCP/IP 协议中，这个标识号就是 IP 地址，目前 IP 地址在计算机中用四个字节，也就是 32 位的二进制数来表示，称为Ipv4。为了便于记忆和使用，我们通常取用每个字节的十进制数，并且每个字节之间用圆点隔开的文本格式来表示 IP 地址，如 192.168.8.1。随着计算机网络规模的不断扩大，用四个字节来表示 IP 地址已越来越不敷使用，人们正在实验和定制使用 16 个字节表示 IP 地址的格式，这就是Ipv6。由于Ipv6 还没有投入使用，现在网络上用的还都是Ipv4，我们这里的知识也只围绕着Ipv4 来展开。

因为一台计算机上可同时运行多个网络程序，IP 地址只能保证把数据送到该计算机，但不能保证把这些数据交给哪个网络程序，因此，每个被发送的网络数据包的头部都包含有一个称为“端口”的部分，它是一个整数，用于表示该数据帧交给哪个应用程序来处理。我们还必须为网络程序指定一个端口号，不同的应用程序接收不同端口上的数据，同一台计算机上不能有两个使用同一端口的程序运行。端口号范围为 0-65535 之间。0-1023 之间的端口号是用于一些知名的网络服务和应用，用户的普通网络应用程序应该使用 1024 以上的端口号，从而避免端口号已被另一个应用或系统服务所用。如果我们的一个网络程序指定了自己所用的端口号为 3150，那么其他网络程序发送给这个网络程序的数据包中必须指明接收程序的端口号为 3150，当数据到达第一个网络程序所在的计算机后，驱动程序根据数据包中的 3150 这个端口号，就知道要将这个数据包交给这个网络程序。

### 11.1.2 UDP 与 TCP

在 TCP/IP 协议栈中，有两个高级协议是我们网络应用程序编写者应该了解的，它们是“传输控制协议”（Transmission Control Protocol，简称 TCP）和“用户数据报协议”（User Datagram Protocol，简称 UDP）。

TCP 是面向连接的通信协议，TCP 提供两台计算机之间的可靠无错的数据传输。应用程序利用 TCP 进行通信时，源和目标之间会建立一个虚拟连接。这个连接一旦建立，两台计算机之间就可以

把数据当作一个双向字节流进行交换。就像我们打电话一样，互相能听到对方的说话，也知道对方的回应是什么。

UDP 是无连接通信协议，UDP 不保证可靠数据的传输，但能够向若干个目标发送数据，接收来自若干个源的数据。简单地说，如果一个主机向另外一台主机发送数据，这一数据就会立即发出，而不管另外一台主机是否已准备接收数据。如果另外一台主机收到了数据，它不会确认收到与否。就像传呼台给用户发信息一样，传呼台并不知道你是否能收到信息（为了避免丢失用户信息，他们常常将一条信息发送两遍）。

TCP、UDP 数据包（也叫数据帧）的基本格式如图 11.1 所示。

TCP、UDP 的数据帧格式简单图例：

|      |     |      |     |      |     |     |
|------|-----|------|-----|------|-----|-----|
| 协议类型 | 源IP | 目标IP | 源端口 | 目标端口 | 帧序号 | 帧数据 |
|------|-----|------|-----|------|-----|-----|

其中协议类型用于区分TCP, UDP

图 11.1

### 11.1.3 Socket

大家不要生硬和孤立地去理解什么是 Socket，就象我们不要让一个从来没有见到过大米与米饭的人去理解什么是“rice”一样的道理，任何一个事物和概念都得有个代名词，大家只有先理解和事物和概念本身，就自然理解了它的代名词。同样 Socket 是网络驱动层提供给应用程序编程的接口和一种机制，大家先掌握和理解了这个机制，自然就明白了什么是 Socket。

大家可以认为 Socket 是应用程序创建的一个港口码头，应用程序只要把装着货物的集装箱（在程序中就是要通过网络发送的数据）放到港口码头上，就算完成了货物的运送，剩下来的工作就由货运公司去处理了（在计算机中由驱动程序来充当货运公司）。

对接收方来说，应用程序也要创建的一个港口码头，然后就一直等待到该码头的货物到达，最后从码头上取走货物（发给该应用程序的数据）。

Socket 在应用程序中创建，通过一种绑定机制与驱动程序建立关系，告诉自己所对应的 Ip 和 Port。此后，应用程序送给 Socket 的数据，由 Socket 交给驱动程序向网络上发送出去。计算机从网络上收到与该 Socket 绑定的 IP+Port 相关的数据后，由驱动程序交给 Socket，应用程序便可从该 Socket 中提取接收到的数据。网络应用程序就是这样通过 Socket 进行数据的发送与接收的。

作者用下面的两个图例来帮助读者理解应用程序、Socket、网络驱动程序之间的数据传送过程与工作关系。

1. 数据发送过程如图 11.2 所示：

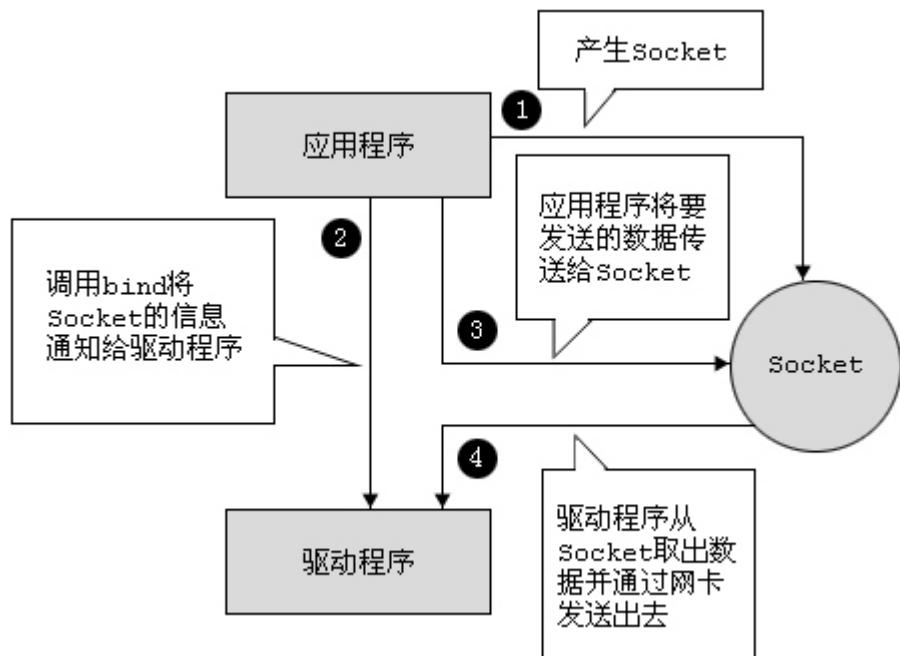


图 11.2

2. 数据接收过程如图 11.3 所示：

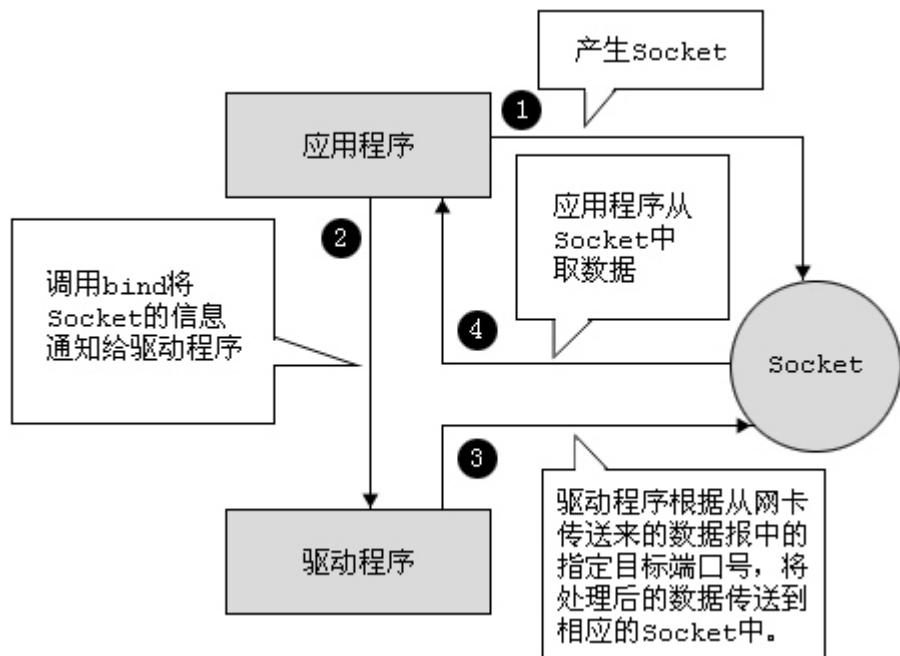


图 11.3

Java 分别为 UDP 和 TCP 两种通信协议提供了相应的编程类，这些类存放在 `java.net` 包中，与 UDP 对应的是 `DatagramSocket`，与 TCP 对应的是 `ServerSocket`(用于服务器端)和 `Socket` (用于客户端)。

网络通信，更确切的说，不是两台计算机之间在收发数据，而是两个网络程序之间在收发数据，我们也可以在一台计算机上进行两个网络程序之间的通信，这两个程序要使用不同的端口号。

## 11.2 Java 编写 UDP 网络程序

### 11.2.1 DatagramSocket

编写 UDP 网络程序，我们首先要用到 `java.net.DatagramSocket` 类，通过查阅 JDK 文档资料，看到 `DatagramSocket` 类的构造函数主要有如下几种形式：

```
public DatagramSocket() throws SocketException  
public DatagramSocket(int port) throws SocketException  
public DatagramSocket(int port, InetAddress laddr) throws SocketException
```

用第一个构造函数创建 `DatagramSocket` 对象，没有指定端口号，系统就会为我们分配一个还没有被其他网络程序所使用的端口号。用第二个构造函数创建 `DatagramSocket` 对象，我们就可以指定自己想要的端口号。用第三个构造函数创建 `DatagramSocket` 对象，我们除了指定自己想要的端口号外，还可以指定相关的 IP 地址，这种情况适用于计算机上有多块网卡和多个 IP 的情况，我们可以明确规定我们的数据通过哪块网卡向外发送和接收哪块网卡收到的数据。如果在创建 `DatagramSocket` 对象时，我们没有指定网卡的 IP 地址，在发送数据时，底层驱动程序会为我们选择其中一块网卡去发送，在接收数据时，我们会接收到所有网卡收到的与程序端口一致的数据，对于我们一般只有一块网卡的情况，我们就不用专门指定了，发送和接收时肯定都是它了。其实，对于只有一块网卡的情况，在这里指定了 IP 地址，反而会给我们的程序带来很大的不方便，你的这个网络程序只能在具有这个 IP 地址的计算机上运行，而不能在其他的计算机上运行。

当我们编写发送程序时，用哪个构造函数呢？我们在创建时 `DatagramSocket` 对象时，不指定端口号，系统就会为我们分配一个端口号，因此，我们可以用第一个构造函数，这样就相当于你给别人打电话时，你的电话可以是任意的，最好不要固定，如果你要用某个电话，那当别人正在用这个电话时，你就只有干等的份了。但作为接收程序，我们必须自己指定一个端口号，而不要让系统随机分配，我们可以用第二个构造函数，否则，我们就不能在程序运行前知道我们的端口号，并且每一次运行所分配的端口号都不一样，就象有朋友让你给他打电话，可他的电话号码不确定是不行的。

如果我们的程序不再使用某个 `Socket`，我们应该调用 `DatagramSocket.close()` 方法，关闭这个 `Socket`，通知驱动程序释放为这个 `Socket` 所保留的资源，系统就可以将这个 `Socket` 所占用的端口号重新分配给其他程序使用。

在发送数据时，我们用 `Datagram.send()` 方法，其完整的格式如下：

```
public void send(DatagramPacket p) throws IOException
```

在要接收数据时，我们用 `Datagram.receive()` 方法，其完整的格式如下：

```
public void receive(DatagramPacket p) throws IOException
```

`Datagram.send()` 和 `Datagram.receive()` 方法都需要我们传递一个 `DatagramPacket` 类的实例对象，如果把 `DatagramSocket` 比作创建的港口码头，那么 `DatagramPacket` 就是我们发送和接收数据的集装箱。

### 11.2.2 DatagramPacket

查阅 JDK 文档，`DatagramPacket` 类的构造函数主要有如下几种形式：

```
public DatagramPacket(byte[] buf, int length)  
public DatagramPacket(byte[] buf, int length, InetAddress address, int port)
```

用第一个构造函数创建的 `DatagramPakcet` 对象，只指定了数据包的内存空间和大小，相当于只定义了集装箱的大小。用第二个构造函数创建的 `DatagramPacket` 对象，不仅指定了数据包的内存空间和大小，而且指定了数据包的目标地址和端口。在接收数据时，我们是没法事先就知道哪个地址和端口的 `Socket` 会给我们发来数据，就象我们要准备一个集装箱去接收发给我们的货物时，是不用标明发货人或是收货人的地址信息的，所以我们应该用第一个构造函数来创建接收数据的

DatagramPakcet 对象。在发送数据时，我们必须指定接收方 Socket 的地址和端口号，就象我们要发送数据的集装箱上面必须标明接收人的地址信息一样的道理，所以我们应该用第二个构造函数来创建发送数据的 DatagramPakcet 对象。

### 11.2.3 InetAddress

在发送数据时， DatagramPacket 构造方法需要我们传递一个 InetAddress 类的实例对象， InetAddress 是用于表示计算机地址的一个类，我们习惯上表示计算机地址是用“192.168.0.1”或“[www.it315.org](http://www.it315.org)”的字符串格式，我们现在要做的就是根据这种习惯上的字符串地址格式来创建一个 InetAddress 类的实例对象，查阅 JDK 文档资料资料，我们发现 InetAddress.getByName() 这个静态方法能够根据我们的条件返回一个 InetAddress 类的实例对象。

另外，当我们将数据接收到 DatagramPacket 对象中后，我们想知道发送方的 IP 地址和端口号，该怎么办呢？到现在为止，我们应该学会了解决类似这样的小问题的最基本的思路了，大家应该很容易想到在 JDK 文档中去查 DatagramPacket 类的方法，看其中有没有解决我们问题的方法。在 JDK 文档中，我们又看到了 DatagramPacket.getInetAddress() 和 DatagramPacket.getPort() 方法。getInetAddress 方法返回的是 InetAddress 类型的对象，我们需要将它转换成用点（.）隔开的字符串型的 IP 地址。在 JDK 文档中去查 InetAddress 类的帮助，我们又可以看到 InetAddress.getHostAddress 方法能够以字符串的形式返回 InetAddress 对象中的 IP 地址。

### 11.2.4 最简单的 UDP 程序

有了前面这些网络编程的基本知识，我们接下来编写两个最简单的 UDP 程序，在一台计算机上相互发送和接收数据，接收程序所用的端口号为 3000，发送程序的端口号由系统分配，这里假设运行程序的计算机的 IP 地址是 192.168.0.213，读者应根据将程序中的这个地址，修改成你的计算机的实际地址后，编译运行。

发送程序: UdpSend.java

```
import java.net.*;
public class UdpSend
{
    public static void main(String [] args) throws Exception
    {
        DatagramSocket ds=new DatagramSocket();
        String str="hello world";
        DatagramPacket dp=new DatagramPacket(str.getBytes(),str.length(),
            InetAddress.getByName("192.168.0.213"),3000);
        ds.send(dp);
        ds.close();
    }
}
```

接收程序: UdpRecv.java

```
import java.net.*;
public class UdpRecv
{
    public static void main(String [] args) throws Exception
    {
```

```

DatagramSocket ds=new DatagramSocket(3000);
byte [] buf=new byte[1024];
DatagramPacket dp=new DatagramPacket(buf,1024);
ds.receive(dp);
String strRecv=new String(dp.getData(),0,dp.getLength()) + " from " +
    dp.getAddress().getHostAddress()+ ":"+dp.getPort();
System.out.println(strRecv);
ds.close();
}
}

```

由于创建 DatagramPacket 时，要求的数据格式都是 byte 型的数组，所以程序在发送数据时用到了 String.getBytes()方法将字符串转换成 byte 型的数组，在接收数据时用到了 String 类的 public String(byte[] bytes, int offset, int length)构造方法，将 byte 型的数组转换成字符串。我们为什么不用 public String(byte[] bytes)构造方法来将 byte 型的数组转换成字符串呢？因为我们在接收数据前，是没法知道对方实际发送的数据包的长度的，因此，在程序中定义 buf 数组具有 1024 个字节，即表示我们能够接收的数据包的大小最多为 1024 个字节，也就是确信对方每次发送的数据包不会超过 1024 个字节的。对方发送的数据的大小是不确定的，往往都不可能正好是 1024 个字节，如上面程序中，我们只收到的“hello world”，只有 11 个字节的数据，public String(byte[] bytes)是将数组中的所有元素都转换成字符串，即将这 1024 个字节都转换成字符串，包括那些根本没有被添充的单元。public String(byte[] bytes, int offset, int length) 是将字节数组中从 offset 开始，往后一共 length 个单元的内容转换成字符串，DatagramPacket.getLength() 方法可以返回数据包中实际收到的字节数。所以，接收程序中的 “String strRecv=new String(dp.getData(),0,dp.getLength()) + " from " + dp.getAddress().getHostAddress()+ ":" +dp.getPort();” 语句将接收到的数据转换成字符串，并在后面加上发送方的地址和端口。

## F

### 指点迷津：

UDP 数据的发送，类似发送寻呼一样的道理，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。就象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，我们要先运行 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，你调试成功了吗？

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。如果网络上没有数据发送过来，receive 方法也没有阻塞，肯定是你前面的程序出现了问题，通常都是使用了一个还在被其他程序占用的端口号，你的 DatagramSocket 绑定没有成功。

这两个网络程序当然也可以在两台计算机上运行，但要将发送方发送数据的目标 IP 设置成接收数据的计算机的 IP 地址。

## &

### 多学两招：

如果将 UdpSend 程序中发送的内容改为中文，如“我的程序”，接收到的内容有问题，请先想想为什么？

因为一个中文字符转换为字节时占用两个字节大小，而一个英文字符转换为字节时只有一个字节大小，所以，应将发送程序中的

```
DatagramPacket dp=new DatagramPacket(str.getBytes(),str.length(),
    InetAddress.getByName(),3000);
```

修改为：

```
DatagramPacket dp=new DatagramPacket(str.getBytes(),str.getBytes().length,
    InetAddress.getByName(),3000);
```

就行了。也就是说，在指定发送数据包的大小时，应按字节数组的大小来计算，而不是字符串中字符的个数。

### 11.2.5 用 UDP 编写聊天程序

掌握了 UDP 网络程序编写的基本过程，我们就可以结合前面的多线程、GUI 来编写一个更完善的网络应用程序，这个程序具有图形用户界面，如图 11.4 所示：

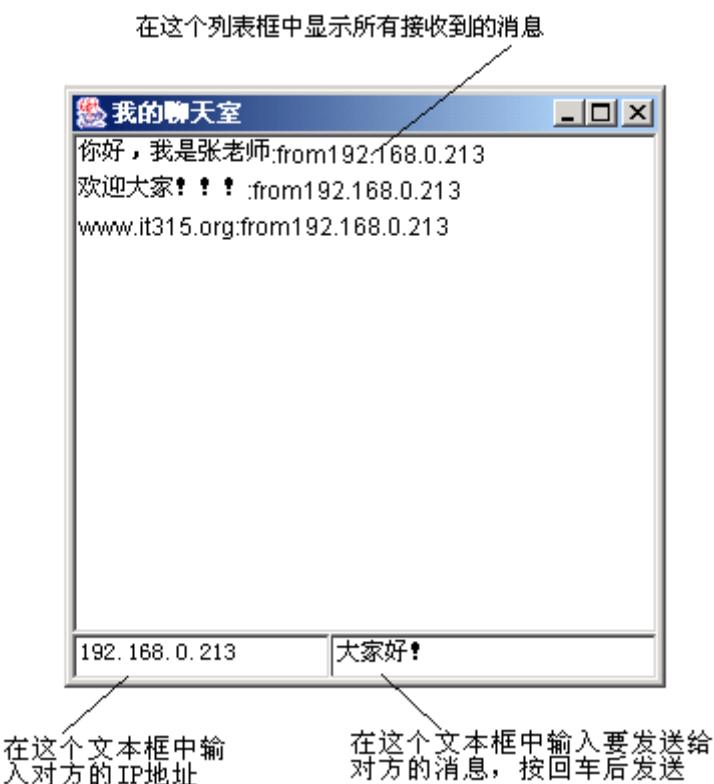


图 11.4

这个程序即可以发送数据，也可以接收数据。

在实际开发中，我们通常都会将一个大的问题分成若干小的问题来解决，对与上面这个程序，我们将其分为三个步骤来完成：

1. 编写图形用户界面部分
2. 编写网络消息发送部分
3. 编写网络消息接收部分

首先，我们编写图形用户界面部分，程序代码如下：

```
import java.awt.*;
import java.awt.event.*;
public class Chat
{
```

```

Frame f=new Frame("我的聊天室");
TextField tfIP=new TextField(15);
/* tfIP 是用于输入 IP 地址的文本框，在发送数据时，要取出其中的 IP 地址，所以将其定义成员变量，以便发送消息的程序代码访问。*/
List lst=new List(6);
/*lst 是用于显示接收消息的列表框，在接收到数据时，要向其中增加新的记录项，所以将其定义成员变量，以便接收消息的程序代码访问。*/
public static void main(String [] args)
{
    Chat chat=new Chat();
    chat.init();
}
public void init()
{
    f.setSize(300,300);
    f.add(lst);

    Panel p=new Panel();
    p.setLayout(new BorderLayout());
    p.add("West",tfIP);
    TextField tfData=new TextField(20);
    p.add("East",tfData);
    f.add("South",p);

    f.setVisible(true);
    f.setResizable(false); //限制用户改变窗口的大小

    //增加关闭窗口的事件处理代码
    f.addWindowListener(new WindowAdapter()
    {
        public void windowClosing(WindowEvent e)
        {
            f.setVisible(false);
            f.dispose();
            System.exit(0);
        }
    });
    //增加在消息文本框中按下回车键的事件处理代码
    tfData.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            //要在这里增加网络消息发送相关程序代码
            //下面的语句用于数据发送后，清空文本框中原来的内容
        }
    });
}

```

```
        ((TextField)e.getSource()).setText("");
    }
} );
}
}
```

编译并运行上面的程序，检查程序是否已经完成了图形用户界面的需求。我们接着编写网络消息发送部分的程序代码，这里用**黑体**标记新加的相关代码，以便读者于原来的代码相区别。

```
import java.awt.*;
import java.awt.event.*;
import java.net.*;
public class Chat
{
    Frame f=new Frame("我的聊天室");
    TextField tfIP=new TextField(15);
    List lst=new List(6);
    DatagramSocket ds;
    /*由于 DatagramSocket 的构造函数声明可能抛出异常，我们的程序需要用 try...catch 语句进行异常捕获处理，所以我们不能直接在这里调用 DatagramSocket 的构造函数对 ds 进行初始化，我们需要将 ds 的初始化放在 Chat 类的构造函数中去完成。*/
    public Chat()
    {
        try
        {
            ds=new DatagramSocket(3000);
        }catch(Exception ex){ex.printStackTrace();}
    }
    public static void main(String [] args)
    {
        Chat chat=new Chat();
        chat.init();
    }
    public void init()
    {
        f.setSize(300,300);
        f.add(lst);

        Panel p=new Panel();
        p.setLayout(new BorderLayout());
        p.add("West",tfIP);
        TextField tfData=new TextField(20);
        p.add("East",tfData);
        f.add("South",p);
        f.setVisible(true);
        f.setResizable(false); //限制用户改变窗口的大小
    }
}
```

```

//增加关闭窗口的事件处理代码
f.addWindowListener(new WindowAdapter()
{
    public void windowClosing(WindowEvent e)
    {
        ds.close(); //程序退出时，关闭 Socket，释放相关资源
        f.setVisible(false);
        f.dispose();
        System.exit(0);
    }
});

//增加在消息文本框中按下回车键的事件处理代码
tfData.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        //取出文本框中的消息字符串，并将其转换成字节数组
        byte[] buf;
        buf = e.getActionCommand().getBytes();
        DatagramPacket dp= new DatagramPacket(buf,buf.length,
        InetAddress.getByName(tfIP.getText()),3000);
        try
        {
            ds.send(dp);
        }catch(Exception ex){ex.printStackTrace();}
        /*上面的 Exception 的引用变量名不能为 e，而是改写成了 ex，因为 e 已经在
        actionPerformed 方法中作为形式参数变量名被定义过了。*/
        ((TextField)e.getSource()).setText("");
    }
});
}

```

我们接着编写网络消息接收部分的程序代码，接收程序代码在一个新的线程中完成，这样，在接收处于阻塞状态时，不会影响到程序的发送部分。新增加的代码为黑体显示部分。

```

import java.awt.*;
import java.awt.event.*;
import java.net.*;
public class Chat
{
    Frame f=new Frame("我的聊天室");
    TextField tfIP=new TextField(15);
    List lst=new List(6);
    DatagramSocket ds;

```

```

public Chat()
{
    try
    {
        ds=new DatagramSocket(3000);
    }catch(Exception ex){ex.printStackTrace();}
    new Thread(new Runnable()
    {
        public void run()
        {
            byte buf[]=new byte[1024];
            DatagramPacket dp= new DatagramPacket(buf,1024);
            while(true)
            {
                try
                {
                    ds.receive(dp);
                    lst.add(new String(buf,0,dp.getLength())+
                            ":from"+dp.getAddress().getHostAddress(),0);
                }catch(Exception e){e.printStackTrace();}
            }
        }
    }).start();
}
.....
}

```

在上面的程序中，我们使用的是 List 的 add(String item, int index) 方法将接收到的消息增加到列表框中，将 index 的值设置为 0，我们可以将最后接收到的消息作为列表框中的第一条记录项显示，为用户提供更友好方便的界面。

编译上面完整的程序，我们就实现了一个具有图形用户界面和收发功能的聊天程序，我们怎样来测试我们的这个程序是否正确呢？在这里，我向大家讲解另外两个小问题，顺便测试我们程序。

第一个问题是：我们这个网络程序能够自己给自己发送数据吗？当然可以，就象一个人非常孤单，自己可以给自己写信，一个网络程序也是可以给自己发送数据的，大家只要将上面的 IP 文本框中的目标 IP 指向自己的主机，你就能收到自己给自己发送的数据了。

第二个问题是：如何发送广播数据？只能在同一个网段中发送广播数据，将该网段 IP 地址的主机号部分的每个二进制位都设置为 1，这个 IP 地址就是这个网段的广播地址了，如果我们发送数据的目标地址是这个网段的广播地址，这个网段上的所有主机都可以接收到发送的数据。作者的主机所在的网络号为 192.168.0，子网掩码是 255.255.255.0，所以，这个网段的广播地址就是 192.168.0.255。顺便给大家补充点儿网络方面的知识，如果作者所在网络的子网掩码是 255.255.254.0，那么这个网段的广播地址就是 192.168.1.255，读者要将上面的 IP 文本框中的目标 IP 指向自己网段的广播地址，你也能收到你发送的数据，因为不管发送方是谁，只要是广播数据，你的主机都能接收的。

如果读者有多台计算机的网络环境，你可以试试多台计算机之间收发数据的情况，只要你在一台计算机上运行正常了，在多台计算机上也不会有什么问题的。

作者在教学中，碰到有的学员问过这样的问题，他的接收程序代码没有放在一个新的线程中，而是直接在 `main` 方法中调用，如下所示：

```
public static void main(String [] args)
{
    Chat chat=new Chat();
    chat.init();
    chat.run();
}

public void run()
{
    byte buf[]=new byte[1024];
    DatagramPacket dp= new DatagramPacket(buf,1024);
    while(true)
    {
        try
        {
            ds.receive(dp);
            lst.add(new String(buf,0,dp.getLength())+
                    " :from"+dp.getAddress().getHostAddress(),0);
        }catch(Exception e){e.printStackTrace();}
    }
}
```

程序也能运行得很好。程序并没有在接收部分阻塞，这是怎么回事呢？这是因为系统为我们创建了一个 AWT 线程，发送数据的程序代码正好在这个 AWT 线程上运行，而接收数据的程序代码就直接在 `main` 方法运行的线程上运行了，可见，发送和接收部分还是在两个不同的线程上运行的。

### 11.3 Java 编写 TCP 网络程序

利用 UDP 通信的两个程序是平等的，没有主次之分，两个程序代码可以是完全一样的。利用 TCP 协议进行通信的两个应用程序，是有主从之分的，一个称为服务器程序，另外一个称为客户机程序，两者的功能和编写方法大不一样。TCP 服务器程序类似 114 查号台，而 TCP 客户机程序类似普通电话。必须先有 114 查号台，普通电话才能拨打 114，在 114 查号台那边是先有一个总机，总机专门用来接听拨打进来的电话，并不与外面的电话直接对话，而是将接进来的电话分配到一个空闲的座机上，然后由这个座机去与外面的电话直接对话。总机在没有空闲的座机时，可以让对方排队等候，但等候服务的电话达到一定数量时，总机就会彻底拒绝以后再拨打进来的电话。Java 中提供的 `ServerSocket` 类用于完成类似 114 查号台总机的功能，`Socket` 类用于完成普通电话和 114 查号台端的座机功能。这个交互的过程如图 11.5 所示：

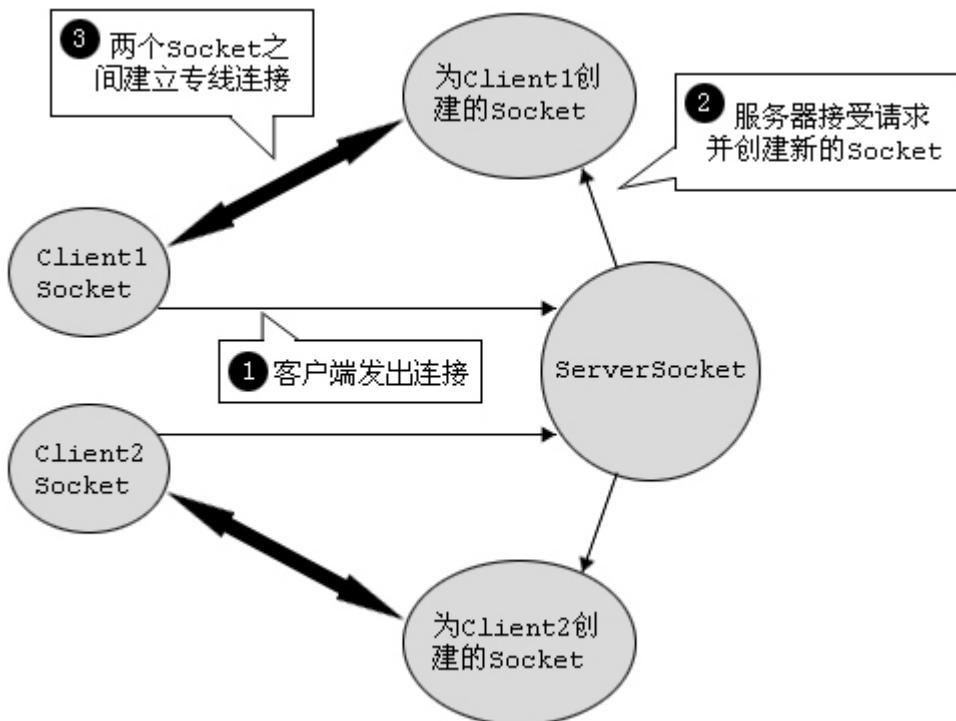


图 11.5

- 1). 服务器程序创建一个 ServerSocket，然后调用 accept 方法等待客户来连接。
- 2). 客户端程序创建一个 Socket 并请求与服务器建立连接。
- 3). 服务器接收客户的连接请求，并创建一个新的 Socket 与该客户建立专线连接。
- 4). 刚才建立了连接的两个 Socket 在一个单独的线程（由服务器程序创建）上对话。
- 5). 服务器开始等待新的连接请求。

### 11.3.1 ServerSocket

编写 TCP 网络服务器程序时，我们首先要用到 `java.net.ServerSocket` 类用以创建服务器 Socket，通过查阅 JDK 文档资料，看到 ServerSocket 类的构造函数有如下几种形式：

```
public ServerSocket() throws IOException
public ServerSocket(int port) throws IOException
public ServerSocket(int port,int backlog) throws IOException
public ServerSocket(int port,int backlog,InetAddress bindAddr) throws IOException
```

用第一个构造函数创建 ServerSocket 对象，没有与任何端口号绑定，不能被直接使用，还要继续调用 `bind` 方法，才能完成其他构造函数所完成的功能。

用第二个构造函数创建 ServerSocket 对象，我们就可以将这个 ServerSocket 绑定到一个指定的端口上，就象为我们的呼叫中心安排一个电话号码一样，如果在这里指定的端口号为 0，系统就会为我们分配一个还没有被其他网络程序所使用的端口号，作为服务器程序，端口号必须事先指定，其他客户才能根据这个号码进行连接，所以将端口号指定为 0 的情况并不常见。

用第三个构造函数创建 ServerSocket 对象，就是在第二个构造函数的基础上，我们根据 `backlog` 参数指定，在服务器忙时，可以与之保持连接请求的等待客户数量，对于第二个构造函数，没有指定这个参数，则使用默认的数量，大小为 50。

用第四个构造函数创建 ServerSocket 对象，我们除了指定第三个构造函数中的参数外，还可以指定相关的 IP 地址，这种情况适用于计算机上有多块网卡和多个 IP 的情况，我们可以明确规定我们的 ServerSocket 在哪块网卡或 IP 地址上等待客户的连接请求，在前面几个构造函数中，都没

有指定网卡的 IP 地址，底层驱动程序会为我们选择其中一块网卡或一个 IP 地址，显然，对于我们一般只有一块网卡的情况，我们就不用专门指定 IP 地址了。我们在前面的 DatagramSocket 部分就讲过，对于只有一块网卡的情况，在这里指定了 IP 地址，反而会给我们的程序带来很大的不方便，你的这个网络程序只能在具有这个 IP 地址的计算机上运行，而不能在其他的计算机上运行。

看完了上面几个 ServerSocket 构造函数的各自作用，对于通常情况的应用，我们不难作出选择，第二个构造方法来创建我们的 ServerSocket 对象是非常合适和方便的。

### 11.3.2 Socket

客户端要与服务器建立连接，首先必须创建一个 Socket 对象，查阅 JDK 文档资料，看到 Socket 类的构造函数有如下几种形式：

```
public Socket()
public Socket(String host,int port) throws UnknownHostException,IOException
public Socket(InetAddress address,int port) throws IOException
public Socket(String host,int port,InetAddress localAddr,int localPort)
    throws IOException
public Socket(InetAddress address,int port,InetAddress localAddr,
    int localPort) throws IOException
```

用第一个构造函数创建 Socket 对象，不与任何服务器建立连接，不能被直接使用，需要调用 connect 方法，才能完成和其他构造函数一样的功能。如果我们想用同一个 Socket 对象，去轮循连接多个服务器，可以使用这个构造函数创建 Socket 对象后，再不断调用 connect 方法去连接每个服务器。

用第二个和第三个构造函数创建 Socket 对象后，会根据参数去连接在特定地址和端口上运行的服务器程序，第二个构造函数接受字符串格式的地址，第三个构造函数接受 InetAddress 对象所包装的地址。

第四个和第五个构造函数在第二个和第三个构造函数的基础上，还指定了本地 Socket 所绑定的 IP 地址和端口号，由于客户端的端口号的选择并不重要，所以一般情况下，我们不会使用这两个构造函数，其中的原因，我们在前面已经讲了许多，这里就不再多说了。

看完了上面几个 Socket 构造函数的介绍，我们了解到，对于通常情况的应用，选择第二个构造函数来创建客户端的 Socket 对象并与服务器建立连接，是非常简单和方便的。

服务器端程序调用 ServerSocket.accept 方法等待客户的连接请求，一旦 accept 接收了客户连接请求，该方法将返回一个与该客户建立了专线连接的 Socket 对象，不用程序去创建这个 Socket 对象。

当客户端和服务器端的两个 Socket 建立了专线连接后，连接的一端能向另一端连续写入字节，也能从另一端连续读入字节，也就是建立了专线连接的两个 Socket 是以 I/O 流的方式进行数据交换的，Java 提供了 Socket.getInputStream 方法返回 Socket 的输入流对象，Socket.getOutputStream 方法返回 Socket 的输出流对象。只要连接的一端向该输出流对象写入了数据，连接的另一端就能从其输入流对象中读取到这些数据。

### 11.3.3 简单的 TCP 服务器程序

明白了 TCP 程序工作的过程，我们就可以编写一个非常简单的 TCP 服务端程序了。

```
import java.net.*;
import java.io.*;
public class TcpServer
{
```

```

public static void main(String [] args)
{
    try
    {
        ServerSocket ss=new ServerSocket(8001);
        Socket s=ss.accept();
        InputStream ips=s.getInputStream();
        OutputStream ops=s.getOutputStream();
        ops.write("welcome to www.it315.org! ".getBytes());
        byte [] buf = new byte[1024];
        int len = ips.read(buf);
        System.out.println(new String(buf, 0, len));
        ips.close();
        ops.close();
        s.close();
        ss.close();
    }catch(Exception e){e.printStackTrace();}
}
}

```

在这个程序中，我们创建了一个在 8001 端口上等待连接的 ServerSocket 对象，当接收到一个客户的连接请求后，程序从与这个客户建立了连接的 Socket 对象中获得输入输出流对象，通过输出流首先向客户端发送一串字符，然后通过输入流读取客户发送过来的信息，并将这些信息存放到一个字节数组中，最后关闭所有有关的资源。

## F

### 指点迷津：

就象必须先建立 114 查号台，客户才能拨打 114 一样的道理，我们要先运行 TCP 服务器程序，然后才能够运行 TCP 客户程序。

当 TCP 服务器程序运行到 ServerSocket.accept 方法等待客户连接时，在正常情况下，accept 方法将阻塞，一直等到有客户连接请求到来，该方法才会返回。如果没有客户连接请求到来的情况下，accept 方法没有发生阻塞，肯定是你前面的程序出现了问题，通常都是使用了一个还在被其他程序占用的端口号，你的 ServerSocket 绑定没有成功。

为了验证我们的服务器程序能否正常工作，还必须有一个客户端程序与之通信。我们每次编写完服务程序后，都还要自己编写一个客户端程序来测试，显然是不太现实的。其实，Windows 提供的 telnet 程序，就是一个 TCP 客户端程序，我们可以直接使用 telnet 程序对我们的服务器程序进行测试。我们只要在运行 telnet 时，指定所要连接的服务器程序的 IP 地址和端口号，telnet 程序就会按照指定参数去与服务器程序建立连接。连接建立后，在 telnet 程序窗口中键盘输入的内容会发送到服务器，从服务器那端接收到的数据会显示在窗口中。

首先运行上面的 TcpServer 程序，然后在一个新的命令行窗口中，运行 telnet 192.168.0.213 8001，注意，作者的计算机的 IP 地址是 192.168.0.213，读者应根据自己计算机的 IP 地址作相应调整，结果如 11.6 所示：

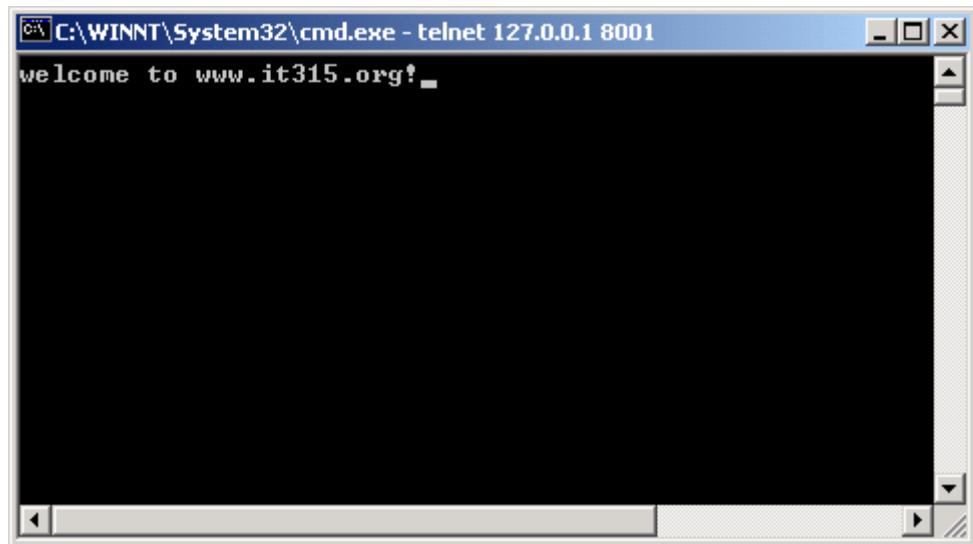


图 11.6

默认情况下，我们在 telnet 窗口键入的字符不会显示 telnet 本地窗口中，我们只好在 telnet 窗口中通过盲打敲入一串字符，如”abc”，但服务程序只收到一个字符 a 就退出了，如图 11.7 所示：



图 11.7

这主要是 telnet 的原因，只要有输入就发送，而不管有没有回车，我们一按下“a”键，它便马上将字符 a 发送过去了，而不等我们后面按回车后，将 abc 一次发送。我们在 10 那章中学过流的概念，不管发送方是将数据分多次送出，还是作为一个整体一次送出，对于接收方来说，效果应该是一样的。所以，如果服务器是想将接受到的数据按一行行的格式处理，需要自己在程序中编写额外的代码，而不能寄希望于客户端每次都是正好整行地发送数据。Java 为我们提供了一个 BufferedReader 类，可以帮我们非常方便地按行处理输入流的功能，修改后的程序代码如下：

```
import java.net.*;
import java.io.*;
public class TcpServer
{
    public static void main(String [] args)
    {
        try
        {

```

```

ServerSocket ss=new ServerSocket(8001);
Socket s=ss.accept();
InputStream ips=s.getInputStream();
OutputStream ops=s.getOutputStream();
ops.write("welcome to www.it315.org!".getBytes());

BufferedReader br = new BufferedReader(new InputStreamReader(ips));
System.out.println(br.readLine());

br.close(); //关闭包装类，会自动关闭包装类中所包装的底层类。所以不用调用 ips.close()
ops.close();
s.close();
ss.close();
}catch(Exception e){e.printStackTrace();
}
}

```

编译后运行上面的程序，我们就可以接收整行数据了。再说一遍，客户端程序可以是逐个逐个地发送一行中的所有字符，也可以是一次就发送一行，不管客户端程序是如何发送它的数据的，服务器程序处理的结果都应该是一样的。

我们编写了一个非常简单的 TCP 服务器程序，这个程序并没有什么实际用途，但达到了作者想用最少的代码来说明自己所要讲的问题的效果。

## & 多学两招：

telnet 程序可以被很好地用作测试 TCP 服务器功能的 TCP 客户端程序，但其默认情况下，不进行本地回显，我们只能用盲打的方式键入我们的字符，很不方便。其实，我们可以对 telnet 程序进行设置，让其对键入的字符进行本地回显。在命令行窗口中，运行 telnet 命令，接着执行 telnet 的 help 子命令查看所有的命令列表，我们就会发现，执行 set LOCAL\_ECHO 子命令可以打开 telnet 程序的本地回显功能，最后执行 quit 退出。以后，我们在 telnet 程序的命令窗口中的键盘输入就能够本地回显了。

### 11.3.4 完善的 TCP 服务器程序模型

我们接着修改上面的程序，让它能够接收多个客户的连接请求，并为每个客户连接创建一个单独的线程与客户进行对话。这程序是每个 TCP 服务器程序的基本框架和雏形，如 http, smtp, pop3, ftp 等服务器程序都会是这样的一种结构，也就是说，不同的服务器程序与客户端对话的方式几乎都是一样的，只是对话的内容不一样，最终完成的功能也就不一样了。

首先，一次 accept 方法调用只接收一个连接，accept 方法需要放在一个循环语句中，这样才可以接收多个连接。

每个连接的数据交换代码，也放在一个循环语句中，保证两者可以不停地交换数据。客户端每向服务器发送一个字符串，服务器就将这个字符串中的所有字符反向排列后回送给客户端，直到客户端向服务器端发送 quit 命令，结束两端的对话。

每个连接的数据交换代码，必须放在独立的线程中运行，否则，在这段代码运行期间，就没法执行其他的程序代码，accept 方法也得不到调用，新的连接就无法进入。我们用一个单独的类来实现服务器端与客户段的对话功能，这个类就叫 Servicer 吧！

完整的程序代码如下：

```

import java.net.*;
import java.io.*;
class Servicer implements Runnable
{
    Socket s;
    public Servicer(Socket s)
    {
        this.s = s;
    }
    public void run()
    {
        try
        {
            InputStream ips=s.getInputStream();
            OutputStream ops=s.getOutputStream();

            BufferedReader br = new BufferedReader(new InputStreamReader(ips));
            DataOutputStream dos = new DataOutputStream(ops);
            while(true)
            {
                String strWord = br.readLine();
                //System.out.println(strWord + ":" + strWord.length());
                if(strWord. equalsIgnoreCase("quit"))
                    break;
                String strEcho = (new StringBuffer(strWord).reverse()).toString();
                //dos.writeBytes(strWord + "---->" + strEcho + "\r\n");
                dos.writeBytes(strWord + "---->" + strEcho +
                               System.getProperty("line.separator"));
            }
            br.close(); //关闭包装类，会自动关闭包装类中所包装的底层类。所以不用调用
            ips.close();
            dos.close();
            s.close();
        }catch(Exception e){e.printStackTrace();}
    }
}
class TcpServer
{
    public static void main(String [] args)
    {
        try
        {
            ServerSocket ss=new ServerSocket(8001);
            while(true)
            {

```

```

        Socket s=ss.accept();
        new Thread(new Servicer(s)).start();
    }
    //ss.close();
}catch(Exception e){e.printStackTrace();}
}
}

```

在上面的程序中，我们使用了 `BufferedReader` 和 `DataOutputStream` 这两个 I/O 包装类，前者可以方便地从底层字节输入流中以整行的形式读取一个字符串，后者可以将一个字符串以字节数组的形式写入底层字节输出流中，合理地使用这些包装类，可以简化我们程序的编写。服务程序给客户端回送的结果也是行的形式发送的，以便客户端程序处理。Java 程序最大的优点就是具有跨平台性，在 Windows 的文本换行是“\r\n”，而 Linux 下的文本换行是“\n”，如果我们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“\r\n”作为文本换行，我们就不要在回送内容后硬性增加“\r\n”，而是应该通过 `System.getProperty("line.separator")` 方法调用，来根据不同的操作系统而返回相应的换行符。

这时，我们可以运行多个 telnet 程序与服务器对话，每一个 telnet 程序就是一个 tcp 客户。如图 11.8 所示：

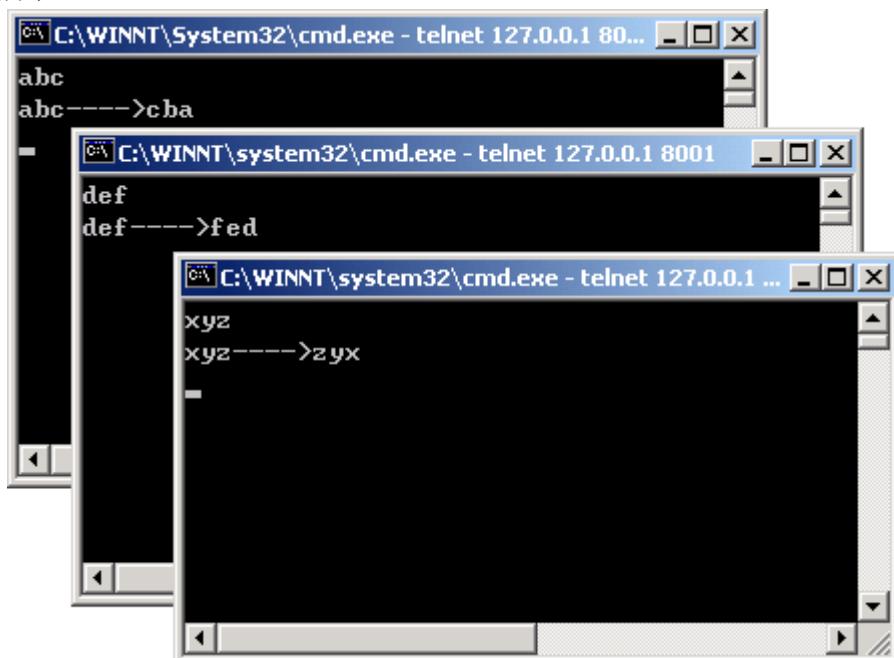


图 11.8

## F 指点迷津：

作者想向服务器端发送 abc，结果输成了 abd，发现第三个字符 c 敲错了 d，接着使用 backspace 键将 d 删除，然后重新键入 c，最后回车发给服务器。发现服务器回送的结果如下：

abc---->dba

读者想想为什么会出现这样的结果呢？作者花费了好长时间，才找出其中的原因，为了开阔读者的视野，培养读者解决实际问题的能力，所以将作者不妨将寻找其中原因的思路提供给大家。为了锻炼你分析问题，解决问题的能力，建议大家自己多想想，多试试后再来参考作者的讲解。

我们服务器端增加了一条打印出接收到的字符串的内容及其长度的语句，在程序中已用注释的

形式保留给了读者参考。对上面的情况，服务器上打印的结果是 abc: 5，尽管显示的内容是 abc，但服务器程序实际收到了 5 个字符，说明字符串的内容就是 abd{backspace}c，由于{backspace}的特殊显示效果，这个五个字符在窗口上的显示结果就是 abc，这个字符串的反向排序结果是 c{backspace}dba，服务器程序回送的结果实际上是 abd{backspace}c--->c{backspace}dba，由于{backspace}的特殊显示效果，所以我们在 telnet 窗口中显示的是 abc--->dba。可见，我们通过网络传送的数据，不能单凭打印的结果来判断，所以，在大多数网络应用程序中，对传送数据中的特殊字符都进行了编解码处理。

### 11.3.5 TCP 客户端程序

接下来，我们编写一个与上面的服务器程序通信的客户端程序。

```
import java.net.*;
import java.io.*;
public class TcpClient
{
    public static void main(String [] args)
    {
        try
        {
            //Socket s=new Socket(InetAddress.getByName("192.168.0.213"),8001);
            if(args.length < 2)
            {
                System.out.println("Usage:java TcpClient ServerIP ServerPort");
                return;
            }
            Socket s=new Socket(
                InetAddress.getByName(args[0]),Integer.parseInt(args[1]));
            InputStream ips=s.getInputStream();
            OutputStream ops=s.getOutputStream();

            BufferedReader brKey =
                new BufferedReader(new InputStreamReader(System.in));
            DataOutputStream dos = new DataOutputStream(ops);
            BufferedReader brNet = new BufferedReader(new InputStreamReader(ips));

            while(true)
            {
                String strWord = brKey.readLine();
                dos.writeBytes(strWord + System.getProperty("line.separator"));
                if(strWord.equalsIgnoreCase("quit"))
                    break;
                else
                    System.out.println(brNet.readLine());
            }
            dos.close();
        }
    }
}
```

```

        brNet.close();
        brKey.close();
        s.close();
    }catch(Exception e){e.printStackTrace();}
}
}

```

在上面的程序中，客户端要连接的服务器的 IP 地址和端口号都是在运行程序时通过参数指定的，这样为我们的程序提供了较好的灵活性和较高的通用性。首先确定服务器程序已经运行，接着按如下格式运行我们的这个客户程序：

```
java TcpClient 192.168.0.213 8001
```

我们可以运行多个这样的客户程序，每一个客户都可以同服务器单独对话，直到客户输入 quit 命令后结束。我们也可以同时运行几个这样的客户端程序和几个 telnet 程序，对服务器端来说都是一视同仁的。

## & 多学两招：

一台计算机上安装的网络应用程序越来越多，很可能我们指定的端口号已被别的程序占用，我们有时候碰到一个以前都能够运行得很好的网络程序，突然有一天就怎么都运行不起来了，这种情况的原因大都属于该网络程序的端口号被别的程序占用了。我们怎么知道自己的计算机上有哪些端口已被使用了呢？我们可以在命令行窗口下运行 netstat 命令，查看已被别的程序使用过的端口，关于这个命令的使用，读者可以运行 netstat -help 获得帮助。如果我们已经运行了前面的服务程序，这时，我们使用 netstat -na 就能看到该程序所使用的端口正处于监听状态，如图 11.9 所示。

| Proto | Local Address | Foreign Address | State     |
|-------|---------------|-----------------|-----------|
| TCP   | 0.0.0.0:25    | 0.0.0.0:0       | LISTENING |
| TCP   | 0.0.0.0:80    | 0.0.0.0:0       | LISTENING |
| TCP   | 0.0.0.0:135   | 0.0.0.0:0       | LISTENING |
| TCP   | 0.0.0.0:443   | 0.0.0.0:0       | LISTENING |
| TCP   | 0.0.0.0:445   | 0.0.0.0:0       | LISTENING |
| TCP   | 0.0.0.0:1025  | 0.0.0.0:0       | LISTENING |
| TCP   | 0.0.0.0:1026  | 0.0.0.0:0       | LISTENING |
| TCP   | 0.0.0.0:1028  | 0.0.0.0:0       | LISTENING |
| TCP   | 0.0.0.0:1030  | 0.0.0.0:0       | LISTENING |
| TCP   | 0.0.0.0:3372  | 0.0.0.0:0       | LISTENING |
| TCP   | 0.0.0.0:8001  | 0.0.0.0:0       | LISTENING |
| TCP   | 0.0.0.0:8733  | 0.0.0.0:0       | LISTENING |
| UDP   | 0.0.0.0:135   | *.*             |           |
| UDP   | 0.0.0.0:445   | *.*             |           |
| UDP   | 0.0.0.0:1027  | *.*             |           |
| UDP   | 0.0.0.0:1029  | *.*             |           |
| UDP   | 0.0.0.0:3456  | *.*             |           |

图 11.9

为了有效解决端口号冲突问题，我们也可以让先前编写的服务器程序的端口号通过程序参数来指定，在万一与某些程序冲突时，我们可以调整程序的端口号，而不用修改程序。为了避免用户每次运行程序时都要指定端口号的麻烦，我们同时也支持在用户没有指定端口号的情况下，使用一个默认值，修改的简要代码如下：

```
public class TcpServer
{
    public static void main(String [] args)
    {
        ServerSocket ss=null;
        if(args.length < 1)
            ss=new ServerSocket(8001);
        Else
            ss=ServerSocket(Integer.parseInt(args[0]));
        .....
        .....
    }
}
```

其实，最为理想的情况是，程序自动将程序上次运行时，用户所指定的端口号保存到一个文件中，用户下次运行时，直接从文件中读取那个端口号。这样，还解决了默认端口号与别的程序冲突时，用户也只需重新指定一次端口号的问题，程序如何实现，对认真学过本书前面所有章节的读者来说，应该不是什么难题，作者就不再多费口舌了。

## F

指点迷津：

### 1. 怎样理解应用程序协议和网络通信协议的关系

我碰到有人问我：FTP 协议支持 TCP 协议吗？我真不知道怎样回答这个问题，问这个问题的人可能是经常接触到一些网络方面的应用和术语，看过一些网络方面的书籍，了解了点皮毛，对网络原理方面的知识几乎就等于零。TCP 协议保证收发双方正确地数据传送，但没法保证接收方正确理解发送方的数据的意义。就如我们的电话能保证一个日本人的声音能传递到一个法国人的耳边，但却没法让这个法国人明白日本人的话的意思。要让电话两端的人彼此明白对方的意思，光听到对方的声音没有完全解决问题，通话的双方必须约定一种语言，通话的双方都必须熟悉这种语言，要求发送者按照一定的格式发音，而接收者才能理解。语言就是基于电话之上的协议（约定），在电话上通过的可以是各种语言，这些语言就是各种各样的协议，日语就是其中的一种协议，只要通话的双方都熟悉那种语言就行。有各种各样的语言，同样，在 TCP 上有各种网络应用，有的用于在两个程序之间传送邮件（smtp 和 pop3 协议），有的是在两个程序之间传送文件（ftp 协议），有的是在两个程序之间传送 WWW 网页（http 协议）。应用的双方都要按照他们都能知道的格式交换数据，不同的应用有不同的协议，每一种协议，都有针对自己的特殊应用的命令和数据格式，对话的两个应用程序必须使用同一种应用协议。

总之，我们把 TCP 协议比作电话，把各种网络应用协议（ftp, smtp, pop3, http 等）就可以比做是各种语言。就象我们的语言不一定非要通过电话系统传递一样的道理，我们的应用程序协议也可以在其他的网络通信协议（非 TCP 协议）上传送。

### 2. 怎样区分 ASP, JSP 与网络编程的概念

有很多书籍名叫什么网络编程高级技术和网络编程大全等等，打开一看，原来所谓的网络编程就是指 asp,jsp,php 等网站相关的脚本语言的讲解，以至于很多初学者认为 ASP, JSP 等就是网络编程。

我要告诉大家一个事实，我们这章讲的知识才是真正的网络编程。我们把网络编程想成一个卫星系统的制造，在卫星系统上可以传送电视节目，ASP、JSP 就相当于制造电视节目用的视频编辑系统。ASP、JSP 产生的网页内容通过网络程序传送，就好比视频编辑系统产生的电视节目在卫星系统上传送一样的道理，做电视节目的人绝对不能说自己是搞卫星制造的。ASP、JSP 是用于产生网站内容，而不是用于编写网络程序的。

|                                     |     |
|-------------------------------------|-----|
| 第 11 章 网络编程 .....                   | 317 |
| 11.1 网络编程的基础知识.....                 | 317 |
| 11.1.1 TCP/IP 网络程序的 IP 地址和端口号 ..... | 317 |
| 11.1.2 UDP 与 TCP .....              | 317 |
| 11.1.3 Socket.....                  | 318 |
| 11.2 Java 编写 UDP 网络程序.....          | 319 |
| 11.2.1 DatagramSocket.....          | 320 |
| 11.2.2 DatagramPacket .....         | 320 |
| 11.2.3 InetAddress .....            | 321 |
| 11.2.4 最简单的 UDP 程序.....             | 321 |
| 指点迷津： UDP 数据的发送原理                   |     |
| 多学两招： 中文信息的处理                       |     |
| 11.2.5 用 UDP 编写聊天程序.....            | 323 |
| 11.3 Java 编写 TCP 网络程序 .....         | 328 |
| 11.3.1 ServerSocket .....           | 329 |
| 11.3.2 Socket.....                  | 330 |
| 11.3.3 简单的 TCP 服务器程序 .....          | 330 |
| 指点迷津： TCP 程序的运行                     |     |
| 多学两招： 使用 telnet 测试 TCP 客户端程序        |     |
| 11.3.4 完善的 TCP 服务器程序模型 .....        | 333 |
| 指点迷津： 要编解码特殊字符                      |     |
| 11.3.5 TCP 客户端程序 .....              | 336 |
| 多学两招： 利用 netstat 命令， 查看程序使用的端口号     |     |
| 指点迷津： 1. 怎样理解应用程序协议和网络通信协议的关系       |     |
| 2. 怎样区分 ASP, JSP 与网络编程的概念           |     |