Java 程序员工资高吗?

年薪 10 万是一个很正常的薪资水平，你的目标应该是年薪 30 万+，这样就可以在北京、深圳、上海等城市拥有自己的房子、车子。可以让垂垂老矣的父母更好的享受人生；经常带着爱人去看爱情海...

我为什么学不好，找不到高薪工作？

也许你会在学无所用之中度过一生，因为你不知道大型企业现在和未来需要什么样的人才，**方向错了一切努力等于白费**。

如何才能找到高人指点迷津？

Java 学习群：72030155，一年 365 天，每天晚上 8：30-11:30 有**腾讯、百度、阿里内部首席架构师来免费讲课**

我们为什么每天都要免费讲课？

1、口碑宣传

2、塑造品牌

3、免费课程都很棒，那么有更好的收费课程，你也会选择的

第1部分　Java语言

第 1 章　Java 的起源

对于计算机语言的发展史，业界一般认为：B语言导致了C语言的诞生，C语言演变出了C++语言，而C++语言将让位于Java语言。要想更好地了解Java语言，就必须了解它产生的原因、推动它发展的动力，以及它对其他语言的继承。像以前其他成功的计算机语言一样，Java继承了其他语言的先进原理，同时又因其独特的环境要求而提出了一些创新性的概念。在这本书的其他各章中，将从实用的角度，对Java语言、库及应用程序进行包括语法在内的详细介绍。在本章里，我们将介绍Java语言产生的背景、发展过程，以及使它变得如此重要的原因。

尽管Java语言已和Internet的在线环境密不可分，但首先应该注意到的最重要一点是：它是一种程序语言。计算机语言的革新和发展需要2个基本因素的驱动：

- 适应正在变化的环境和需求
- 实现编程艺术的完善与提高

下面你将看到，Java也正是在这两个因素的驱动下产生的。

1.1　Java的由来

Java总是和C++联系在一起，而C++则是从C语言派生而来的，所以Java语言继承了这两种语言的大部分特性。Java的语法是从C继承的，Java许多面向对象的特性受到C++的影响。事实上，Java中几个自定义的特性都来自于或可以追溯到它的前驱。而且，Java语言的产生与过去30年中计算机语言细致改进和不断发展密切相关。基于这些原因，本节将按顺序回顾促使Java产生的事件和推动力。正如你将看到的一样，每一次语言设计的革新都是因为先前的语言不能解决目前遇到的基本问题而引起。Java也不例外。

1.1.1　现代的编程语言的诞生：C语言

C语言的产生震撼了整个计算机界。它的影响不应该被低估，因为它从根本上改变了编程的方法和思路。C语言的产生是人们追求结构化、高效率、高级语言的直接结果，可用它替代汇编语言开发系统程序。当设计一种计算机语言时，经常要从以下几方面进行权衡：

- 易用性与功能
- 安全性和效率性
- 稳定性和可扩展性

C语言出现以前，程序员们不得不经常在有优点但在某些方面又有欠缺的语言之间做出选择。例如，尽管公认FORTRAN在科学计算应用方面可以编写出相当高效的程序，但它不适于编写系统程序。BASIC虽然容易学习，但功能不够强大，并且谈不上结构化，这使它应用到大程序的有效性受到怀疑。汇编语言虽能写出高效率的程序，但是学习或有效地使用它却是不容易的。而且，调试汇编程序也相当困难。

另一个复杂的问题是，早期设计的计算机语言（如BASIC，COBOL，FORTRAN等）没有考虑结构化设计原则，使用GOTO语句作为对程序进行控制的一种主要方法。这样做的结果是，用这些语言编写的程序往往成了“意大利面条式的程序代码”，一大堆混乱的跳转语句和条件分支语句使得程序几乎不可能被读懂。Pascal虽然是结构化语言，但它的设计效率比较低，而且缺少几个必需的特性，因而无法在大的编程范围内使用（特别是，给定的Pascal的标准语言在特定时间是可用的，但将Pascal作为系统级编码是不切实际的）。

因此，在C语言产生以前，没有任何一种语言能完全满足人们的需要，但人们对这样一种语言的需要却是迫切的。在20世纪70年代初期，计算机革命开始了，对软件的需求量日益增加，使用早期的计算机语言进行软件开发根本无法满足这种需要。学术界付出很多努力，尝试创造一种更好的计算机语言。但是，促使C语言诞生的另一个，也许是最重要的因素，是计算机硬件资源的富余带来了机遇。计算机不再像以前那样被紧锁在门里，程序员们可以随意使用计算机，可以随意进行自由尝试，因而也就有了可以开发适合自己使用的工具的机会。所以，在C语言诞生的前夕，计算机语言向前飞跃的时机已经成熟。

在Dennis Ritchie第一个发明和实现在DEC PDP-11上运行UNIX操作系统时，一种更古老的由Martin Richards设计的BCPL语言导致了C语言的产生。受BCPL语言的影响，由Ken Thompson发明的B语言，在20世纪70年代逐渐向C语言发展演变。在此后的许多年里，由Brian Kernighan和Dennis Ritchie编写的《The C Programming Language》(Prentice-Hall，1978)被认为是事实上的C语言标准，该书认为C只是支持UNIX 操作系统的一种语言。1989年12月，美国国家标准化组织(ANSI)制定了C语言的标准，C语言被正式标准化。

许多人认为C语言的产生标志着现代计算机语言时代的开始。它成功地综合处理了长期困扰早期语言的矛盾属性。C语言是功能强大、高效的结构化语言，简单易学，而且它还包括一个无形的方面：它是程序员自己的语言。在C语言出现以前，计算机语言要么被作为学术实验而设计，要么由官僚委员会设计。而C语言不同。它的设计、实现、开发由真正的从事编程工作的程序员来完成，反映了现实编程工作的方法。它的特性经由实际运用该语言的人们不断去提炼、测试、思考、再思考，使得C语言成为程序员们喜欢使用的语言。确实，C语言迅速吸引了许多狂热的追随者，因而很快受到许多程序员的青睐。简言之，C语言是由程序员设计并由他们使用的一种语言。正如你将看到的，Java继承了这个思想。

1.1.2　对C++的需要

在20世纪70年代末和80年代初，C成为了主流的计算机编程语言，至今仍被广泛使用。你也许会问，既然C是一种成功且有用的语言，为什么还需要新的计算机语言？答案是复杂性（complexity）。程序越来越复杂这一事实贯穿编程语言的历史。C++正是适应了这一需求。下面介绍为什么对程序复杂性的更好管理是C++产生的基本条件。

自从计算机发明以来，编程方法经历了戏剧性的变化。例如，当计算机刚发明出来时，编程是通过面板触发器用人工打孔的办法输入二进制机器指令来实现的。对于只有几百行的程序，这种办法是可行的。随着程序不断增大，人们发明了汇编语言，它通过使用符号来代替机器指令，这样程序员就能处理更大、更复杂的程序。随着程序的进一步增大，高级语言产生了，它给程序员提供了更多的工具来处理复杂性问题。

第一个被广泛使用的高级语言当然是FORTRAN。尽管FORTRAN最初给人留下了深刻的印象，但它无法开发出条理清楚易于理解的程序。20世纪60年代提出了结构化编程方法。这种结构化的编程思想被像C这样的语言所应用，第一次使程序员可以相对轻松地编写适度复杂的程序。然而，当一个工程项目达到一定规模后，即使使用结构化编程方法，编程人员也无法对它的复杂性进行有效管理。20世纪80年代初期，许多工程项目的复杂性都超过了结构化方法的极限。为解决这个问题，面向对象编程（object-oriented programming，OOP）新方法诞生了。面向对象的编程在这本书的后面详细讨论，但在这里给出一个简短的定义：面向对象的编程是通过使用继承性、封装性和多态性来帮助组织复杂程序的编程方法。

总之，尽管C是世界上伟大的编程语言之一，但它处理复杂性的能力有限。一旦一个程序的代码超过25 000~100 000行，就很难从总体上把握它的复杂性了。C++突破了这个限制，帮助程序员理解并且管理更大的程序。

1979年，当Bjarne Stroustrup在新泽西州的Murray Hill实验室工作时，发明了C++。Stroustrup 最初把这种新语言称为“带类的C”。1983年，改名为C++。C++通过增加面向对象的特性扩充了C。因为C++产生在C的基础之上，因此它包括了C所有的特征、属性和优点。这是C++作为语言成功的一个关键原因。C++的发明不是企图创造一种全新的编程语言，而是对一个已经高度成功的语言的改进。C++在1997年11月被标准化，目前的标准是ANSI/ISO。

1.1.3　Java出现的时机已经到来

在20世纪80年代末和90年代初，使用面向对象编程的C++语言占主导地位。的确，有一段时间程序员似乎都认为已经找到了一种完美的语言。因为C++有面向对象的特征，又有C语言高效和格式上的优点，因此它是一种可以被广泛应用的编程语言。然而，就像过去一样，推动计算机语言进化的力量正在酝酿。在随后的几年里，万维网（WWW）和Internet达到临界状态。这个事件促成编程的另一场革命。

1.2 Java的产生

Java是由James Gosling，Patrick Naughton，Chris Warth，Ed Frank和Mike Sheridan于1991年在Sun Microsystems公司设计出来的。开发第一个版本花了18个月。该语言开始名叫“Oak”，于1995年更名为“Java”。从1992 的秋天Oak问世到1995的春天公开发布Java语言，许多人对Java的设计和改进做出了贡献。Bill Joy，Arthur van Hoff，Jonathan Payne，Frank Yellin和Tim Lindholm是主要的贡献者，正是他们的贡献使最初原型得以成熟。

说起来多少有些令人吃惊，Java的最初推动力并不是因特网！而是源于对独立于平台（也就是体系结构中立）语言的需要，这种语言可创建能够嵌入微波炉、遥控器等各种家用电器设备的软件。用作控制器的CPU芯片是多种多样的，但C和C++以及其他绝大多数语言的缺点是只能对特定目标进行编译。尽管为任何类型的CPU芯片编译C++程序是可能的，但这样做需要一个完整的以该CPU为目标的C++编译器，而创建编译器是一项既耗资巨大又耗时较长的工作。因此需要一种简单且经济的解决方案。为了找到这样一种方案，Gosling和其他人开始一起致力于开发一种可移植、跨平台的语言，该语言能够生成运行于不同环境、不同CPU芯片上的代码。他们的努力最终促成了Java的诞生。

在Java的一些细节被设计出来的同时，第二个并且也是最重要的因素出现了，该因素将对Java的未来起着至关重要的作用。这第二个因素当然就是万维网（WWW）。如果万维网（WWW）的成型和Java的实现不是同时发生的话，那么Java可能保持它有用、但默默无闻的用于电子消费品编程语言的状态。然而，随着万维网的出现，Java被推到计算机语言设计的最前沿，因为万维网也需要可移植的程序。

绝大多数程序员在涉足编程领域时就知道可移植的程序像他们的理想一样难以捉摸。尽管人们对高效的、可移植的（独立于平台）编程方式的追寻几乎和编程历史一样久远，但它总是让位于其他的更为紧迫的问题。此外，因为计算机业被Intel，Macintosh和UNIX这3个竞争对手垄断，大多数程序员都在其中的某个领域内长期工作，所以对可移植语言的需求就不是那么迫切。但是，随着因特网和Web的出现，关于可移植性语言的旧问题又被提了出来。毕竟，因特网由不同的、分布式的系统组成，其中包括各种类型的计算机、操作系统和CPU。尽管许多类型的平台都可以与因特网连接，但用户仍希望他们能够运行同样的程序。曾经是一个令人烦恼却无需优先考虑的问题现在变成了急需解决的问题。

1993年，Java设计小组的成员发现他们在编制嵌入式控制器代码时经常遇到的可移植性问题，在编制因特网代码的过程中也出现了。事实上，开始被设计为解决小范围问题的Java语言同样可以被用在大范围的因特网上。这个认识使他们将Java的重心由电子消费品转移到Internet编程。因此，中立体系结构编程语言的需要是促使Java诞生的源动力，而Internet却最终导致了Java的成功。

正如前面提到的，Java的大部分特性是从C和C++中继承的。Java设计人员之所以故意这么做，主要是因为他们觉得，在新语言中使用熟悉的C语法及模仿C++面向对象的特性，将使他们的语言对经验丰富的C/C++程序员有更大的吸引力。除了表面类似外，其他一些促使C和C++成功的因素也帮了Java的忙。首先，Java的设计、测试、精炼由真正从事编程

工作的人员完成，它根植于设计它的人员的需要和经验，因而也是一个程序员自己的语言。其次，Java是紧密结合的且逻辑上是协调一致的。最后，除了那些Internet环境强加的约束以外，Java给了编程人员完全的控制权。如果你程序编的好，你编写的程序就能反映出这一点。相反，如果你的编程手法拙劣，也能在你的程序中反映出来。换一种说法，Java并不是训练新手的语言，而是供专业编程人员使用的语言。

由于Java和C++之间的相似性，容易使人将Java简单地想象为“C++的版本”。但其实这是一种误解。Java在实践和理论上都与C++有重要的不同点。尽管Java受到C++的影响，但它并不是C++的增强版。例如，Java与C++既不向上兼容，也不向下兼容。当然，Java与C++的相似之处也是很多的，如果你是一个C++程序员，你会感觉到对Java非常熟悉。另外一点是：Java并不是用来取代C++的，设计Java是为了解决某些特定的问题，而设计C++是为了解决另外一类完全不同的问题。两者将长时间共存。

正如本章开始提到的，计算机语言的革新靠两个因素驱动：对计算环境改变的适应和编程艺术的进步。环境的变化促使Java这种独立于平台的语言注定成为Internet上的分布式编程语言。同时，Java也改变了人们的编程方式，特别是Java对C++使用的面向对象范例进行的增强和完善。所以，Java不是孤立存在的一种语言，而是计算机语言多年来的演变结果。仅这个事实就足以证明Java在计算机语言历史上的地位。Java对Internet编程的影响就如同C对系统编程的影响一样：革命的力量将改变世界。

1.3　Java对Internet为什么重要

Internet使Java成为网上最流行的编程语言，同时Java对Internet的影响也意义深远。原因相当简单：Java扩展了可以在赛百空间自由流动的对象的世界。在网络中，有两大类对象在服务器和个人计算机之间传输：被动的信息和动态的、主动的程序。例如，当你阅读电子邮件时，你在看被动的数据。甚至当你下载一个程序时，该程序的代码也是被动的数据，直到你执行它为止。但是，可以传输到个人计算机的另一类对象却是：动态的、可自运行的程序，虽然这类程序是客户机上的活动代理，但却是由服务器来初始化的。例如，被服务器用来正确地显示服务器传送数据的程序。

网上程序在动态性上是令人满意的，但它们在安全性和可移植性方面也显示出严重的缺陷。在Java产生以前，当前赛百空间有一半的对象实体无法进入网络世界，是Java为它们打开了便利之门，而且在这个过程中定义了一种全新的程序形式：applet(小应用程序)。

1.3.1　Java小应用程序和应用程序

Java可用来生成两类程序：应用程序（applications）和Java applet（小应用程序）。应用程序是可以在你的计算机的操作系统中运行的程序，从这一方面来说，用Java编制的应用程序多多少少与使用C或C++编制的应用程序有些类似。在创建应用程序时，Java与其他计算机语言没有大的区别。而Java的重要性就在于它具有编制小应用程序的功能。小应用程序是可以在Internet中传输并在兼容Java的Web浏览器中运行的应用程序。小应用程序实际上就是小型的Java程序，能像图像文件、声音文件和视频片段那样通过网络动态下载，

它与其他文件的重要差别是，小应用程序是一个智能的程序，能对用户的输入作出反应，并且能动态变化，而不是一遍又一遍地播放同一动画或声音。

如果Java不能解决两个关于小应用程序的最棘手的问题：安全性和可移植性，那么小应用程序就不会如此令人激动。在继续下一个话题之前，让我们先说明以下这两个术语对Internet的意义。

1.3.2 安全性

正如你知道的那样，每次当你下载一个“正常”的程序时，你都要冒着被病毒感染的危险。在Java出现以前，大多数用户并不经常下载可执行的程序文件；即使下载了程序，在运行它们以前也都要进行病毒检查。尽管如此，大多数用户还是担心他们的系统可能被病毒感染。除了病毒，另一种恶意的程序也必须警惕。这种恶意的程序可通过搜索你计算机本地文件系统的内容来收集你的私人信息，例如信用卡号码、银行账户结算和口令。Java在网络应用程序和你的计算机之间提供了一道防火墙（firewall），消除了用户的这些顾虑。

当使用一个兼容Java的Web浏览器时，你可以安全地下载Java小应用程序，不必担心病毒的感染或恶意的企图。Java实现这种保护功能的方式是，将Java程序限制在Java运行环境中，不允许它访问计算机的其他部分，后面将介绍这个过程是如何实现的。下载小应用程序并能确保它对客户机的安全性不会造成危害是Java的一个最重要的方面。

1.3.3 可移植性

正如前面所讨论的，许多类型的计算机和操作系统都连接到Internet上。要使连接到Internet上的各种各样的平台都能动态下载同一个程序，就需要有能够生成可移植性执行代码的方法。很快你将会看到，有助于保证安全性的机制同样也有助于建立可移植性。实际上，Java对这两个问题的解决方案是优美的也是高效的。

1.4 Java的魔力：字节码

Java解决上述两个问题——安全性和可移植性的关键在于Java编译器的输出并不是可执行的代码，而是字节码（bytecode）。字节码是一套设计用来在Java运行时系统下执行的高度优化的指令集，该Java运行时系统称为Java虚拟机(JavaVirtual Machine，JVM)。在其标准形式下，JVM 就是一个字节码解释器。这可能有点让人吃惊，因为像C++之类语言的编译结果是可执行的代码。事实上，出于对性能的考虑，许多现代语言都被设计为编译型，而不是解释型。然而，正是通过JVM运行Java程序才有助于解决在Internet上下载程序的主要问题。这就是Java输出字节码的原因。

将一个Java程序翻译成字节码，有助于它更容易地在一个大范围的环境下运行程序。原因非常直接：只要在各种平台上都实现Java虚拟机就可以了。在一个给定的系统中，只要系统运行包存在，任何Java程序就可以在该系统上运行。记住：尽管不同平台的Java虚拟机的细节有所不同，但它们都解释同样的Java字节码。如果一个Java程序被编译为本机代码，那么对于连接到Internet上的每一种CPU类型，都要有该程序的对应版本。这当然不是一个

可行的解决方案。因此，对字节码进行解释是编写真正可移植性程序的最容易的方法。

对Java程序进行解释也有助于它的安全性。因为每个Java程序的运行都在Java虚拟机的控制之下，Java虚拟机可以包含这个程序并且能阻止它在系统之外产生副作用。正如你将看到的，Java语言特有的某些限制增强了它的安全性。

被解释的程序的运行速度通常确实会比同一个程序被编译为可执行代码的运行速度慢一些。但是对Java来说，这两者之间的差别不太大。使用字节码能够使Java运行时系统的程序执行速度比你想象的快得多。

尽管Java被设计为解释执行的程序，但是在技术上Java并不妨碍动态将字节码编译为本机代码。SUN公司在Java 2发行版中提供了一个字节码编译器——JIT（Just In Time，即时）。JIT是Java虚拟机的一部分，它根据需要、一部分一部分地将字节码实时编译为可执行代码。它不能将整个Java程序一次性全部编译为可执行的代码，因为Java要执行各种检查，而这些检查只有在运行时才执行。记住这一点是很重要的，因为JIT只编译它运行时需要的代码。尽管如此，这种即时编译执行的方法仍然使性能得到较大提高。即使对字节码进行动态编译后，Java程序的可移植性和安全性仍能得到保证，因为运行时系统（该系统执行编译）仍然能够控制Java程序的运行环境。不管Java程序被按照传统方式解释为字节码，还是被动态编译为可执行代码，其功能是相同的。

1.5　Java常用语

不介绍Java常用语，对Java的总体介绍就是不完整的。尽管促使Java诞生的源动力是可移植性和安全性，但在Java语言最终成型的过程中，其他一些因素也起了重要的作用。Java设计开发小组的成员总结了这些关键因素，称其为Java的专门用语，包括下面几个：

- 简单（Simple）
- 安全（Secure）
- 可移植（Portable）
- 面向对象（Object-oriented）
- 健壮（Robust）
- 多线程（Multithreaded）
- 体系结构中立（Architecture-neutral）
- 解释执行（Interpreted）
- 高性能（High performance）
- 分布式（Distributed）
- 动态（Dynamic）

在这些特性中，安全和可移植已经在前面介绍过了，下面让我们看看其他特性的含义。

1.5.1　简单

Java的设计目的是让专业程序员觉得既易学又好用。假设你有编程经历，你将不觉得

Java难掌握。如果你已经理解面向对象编程的基本概念，学习Java将更容易。如果你是一个经验丰富的C++程序员，那就最好了，学习Java简直不费吹灰之力。因为Java继承C/C++语法和许多C++面向对象的特性，大多数程序员在学习Java时都不会觉得太难。另外，C++中许多容易混淆的概念，或者被Java弃之不用了，或者以一种更清楚、更易理解的方式实现。

除了和C/C++类似以外，Java的另外一个属性也使它更容易学习：设计人员努力使Java中不出现显得让人吃惊的特性。在Java中，很少明确地告诉你如何才能完成一项特定的任务。

1.5.2 面向对象

尽管受到其前辈的影响，但Java没被设计成兼容其他语言源代码的程序。这允许Java开发组自由地从零开始。这样做的一个结果是，Java语言可以更直接、更易用、更实际的接近对象。 通过对近几十年面向对象软件优点的借鉴，Java设法在纯进化论者的“任何事物都是一个对象”和实用主义者的“不讨论对象不对象”的论点之间找到了平衡。Java的对象模型既简单又容易扩展，对于简单数据类型，例如整数，它保持了高性能，但不是对象。

1.5.3 健壮

万维网上多平台的环境使得它对程序有特别的要求，因为程序必须在许多系统上可靠地执行。这样，在设计Java时，创建健壮的程序被放到了高度优先考虑的地位。为了获得可靠性，Java在一些关键的地方限制你，强迫你在程序开发过程中及早发现错误。同时，Java使你不必担心引起编程错误的许多最常见的问题。因为Java是一种严格的类型语言，它不但在编译时检查代码，而且在运行时也检查代码。事实上，在运行时经常碰到的难以重现的、难以跟踪的许多错误在Java中几乎是不可能产生的。要知道，使程序在不同的运行环境中以可预见的方式运行是Java的关键特性。

为更好理解Java是如何具有健壮性的，让我们考虑使程序失败的两个主要原因：内存管理错误和误操作引起的异常情况(也就是运行时错误)。在传统的编程环境下，内存管理是一项困难、乏味的任务。例如，在C/C++中，程序员必须手工地分配并且释放所有的动态内存。这有时会导致问题，因为程序员可能忘记释放原来分配的内存，或者释放了其他部分程序正在使用的内存。Java通过替你管理内存分配和释放，可以从根本上消除这些问题（事实上，释放内存是完全自动的，因为Java为闲置的对象提供内存垃圾自动收集）。在传统的环境下，异常情况可能经常由“被零除”或“文件未找到”这样的情况引起，而我们又必须用既繁多又难以理解的一大堆指令来对它们进行管理。Java通过提供面向对象的异常处理机制来解决这个问题。一个写得很好的Java程序，所有的运行时错误都可以并且应该被你的程序自己进行管理。

1.5.4 多线程

设计Java的目标之一是为了满足人们对创建交互式网上程序的需要。为此，Java支持多线程编程，因而你用Java编写的应用程序可以同时执行多个任务。Java运行时系统在多线程

同步方面具有成熟的解决方案，这使你能够创建出运行平稳的交互式系统。Java的多线程机制非常好用，因而你只需关注程序细节的实现，不用担心后台的多任务系统。

1.5.5　结构中立

Java设计者考虑的一个主要问题是程序代码的持久性和可移植性。程序员面临的一个主要问题是，不能保证今天编写的程序明天能否在同一台机器上顺利运行。操作系统升级、处理器升级以及核心系统资源的变化，都可能导致程序无法继续运行。Java设计者对这个问题做过多种尝试，Java虚拟机（JVM）就是试图解决这个问题的。他们的目标是“只要写一次程序，在任何地方、任何时间该程序永远都能运行”。在很大程度上，Java实现了这个目标。

1.5.6　解释性和高性能

前面已提到，通过把程序编译为Java字节码这样一个中间过程，Java可以产生跨平台运行的程序。字节码可以在提供Java虚拟机（JVM）的任何一种系统上被解释执行。早先的许多尝试解决跨平台的方案对性能要求都很高。其他解释执行的语言系统，如BASIC，Tcl，PERL都有无法克服的性能缺陷。然而，Java却可以在非常低档的CPU上顺利运行。前面已解释过，Java确实是一种解释性语言，Java的字节码经过仔细设计，因而很容易便能使用JIT编译技术将字节码直接转换成高性能的本机代码。Java运行时系统在提供这个特性的同时仍具有平台独立性，因而“高效且跨平台”对Java来说不再矛盾。

1.5.7　分布式

Java为Internet的分布式环境而设计，因为它处理TCP/IP协议。事实上，通过URL地址存取资源与直接存取一个文件的差别是不太大的。Java原来的版本(Oak) 包括了内置的地址空格消息传递(intra-address-space)特性。这允许位于两台不同的计算机上的对象可以远程地执行过程。Java最近发布了叫做远程方法调用（Remote Method Invocation ，RMI）的软件包，这个特性使客户机/服务器编程达到了无与伦比的抽象级。

1.5.8　动态

Java程序带有多种的运行时类型信息，用于在运行时校验和解决对象访问问题。这使得在一种安全、有效的方式下动态地连接代码成为可能，对小应用程序环境的健壮性也十分重要，因为在运行时系统中，字节码内的小段程序可以动态地被更新。

1.6　继 续 革 命

Java的最初发布本不亚于一场革命，但是它并不标志着Java快速革新时代的结束。与大多数其他软件系统经常进行小的改进不同，Java继续以爆炸式的步伐向前发展。在Java 1.0发布不久，Java的设计者已经创造出了Java 1.1。Java 1.1新增的特性远比普通意义上的版本修订有意义，内容要丰富许多。Java 1.1增加了许多新的库元素，重新定义了小应用程序处

理事件的方法，并且重新设置了1.0版中库的许多特性。它也放弃了原来由Java1.0定义的若干过时的特征。因此，Java 1.1不但增加了Java 1.0中没有的属性，同时也抛弃了一些原有的属性。

Java的第二个主要发布版本是Java 2。Java 2 是一个分水岭，它标志这个快速演变语言“现代时代”的开始！Java 2第一版本的版本号是1.2。这似乎有点奇怪。原因是它参考了原来Java库的版本，对于整个版本来说，它本身没有多大变化。Java 2增加了很多对新特性的支持，例如Swing和类集框架，并且它提高了Java虚拟机和各种编程工具的性能。Java 2也包含了一些不赞成继续使用的内容，主要是不赞成使用线程类中suspend( )，resume( )和stop( )这些方法。

Java的当前版本是Java 2，1.3版。Java的这个版本是对Java 2原来版本的第一次最主要的升级。该版本增强了Java大部分现有的功能，并且限制了它的开发环境。总的来说，版本1.2和版本1.3的程序源代码是兼容的。尽管与前面3个版本相比，版本1.3作了一些小的改变，但这是无关紧要的。

本书适合Java 2的1.2和1.3版。当然，大多数内容也适用于Java早期的版本。在本书中，当一个特性只适用于Java的一个特定的版本时，会被注明。否则，你就可以认为它适用于一般的Java版本。另外，对于适用于Java 2两个版本的那些特性，本书中将简单地使用术语Java 2，而不注明版本号。

1.7 Java不是增强的HTML

在继续讲解前，有必要澄清一个普遍的误解。因为Java被用来创建网页，所以初学者有时将Java与超文本标记语言（HTML）混淆，或认为Java仅仅是对HTML的一些改进。幸好，这只是误解。实质上，HTML是一种定义信息逻辑的组织方式并提供相关信息的链接（叫超文本链接）。你可能知道，超文本链接（hypertext link）（也叫超链接）是把一个超文本与另一个超文本文档链接起来的工具，而这个被链接的超文本文档可能在本地或万维网上其他地方。超文本文档要素的定义是通过选择该超文本文档与另一个相关文档的链接，在用户搜索各种路径后，该超文本文档可以非线性的方式阅读。

尽管HTML允许用户以动态方式阅读文档，但HTML永远无法成为一种编程语言。当然，HTML确实帮助和推进了万维网的普及，HTML是促使Java诞生的催化剂，但它没有直接从概念上影响Java语言的设计。HTML与Java的惟一联系是，HTML提供Java小应用程序标记，该标记启动Java小应用程序。这样，就可以在超文本文档中嵌入能启动Java小应用程序的指令。

第 2 章　Java 语言概述

像所有其他的计算机语言一样，Java的各种要素不是独立存在的，它们作为一个整体共同构成了Java语言。这种关联使得不讲其他方面而单独描述Java的某一方面是困难的。讨论一个特性经常要先具有另外一个特性的知识。因此，本章先对Java的若干主要特性做简单综述。这里描述的主题将给你一个立足点：能够使你编写和理解简单的Java程序。大多数讨论话题将在第1部分的其他章节详细叙述

2.1　面向对象编程

Java的核心是面向对象编程。事实上，所有的Java程序都是面向对象的，你别无选择。这一点与C++不同，因为在那里你可以选择是否面向对象编程。面向对象编程与Java密不可分，因此，在你编写哪怕是最简单的Java程序以前，也必须理解它的基本原则。因此，本章先从面向对象编程的概念讲起。

2.1.1　两种范型

我们知道，所有的计算机程序都由两类元素组成：代码和数据。此外，从概念上讲，程序还可以以它的代码或是数据为核心进行组织编写。也就是说，一些程序围绕“正在发生什么”编写，而另一些程序则围绕“谁将被影响”编写。这两种范型决定程序的构建方法。第一种方法被称为面向过程的模型（process-oriented model），用它编写的程序都具有线性执行的特点。面向过程的模型可认为是代码作用于数据，像C这样的过程式语言采用这个模型是相当成功的。然而，正如在第1章提到的，当程序变得更大并且更复杂时，就会出现问题。

为了管理不断增加的复杂性，第二种方式，也就是面向对象的编程（object-oriented programming）被构思出来了。面向对象的编程围绕它的数据（即对象）和为这个数据严格定义的接口来组织程序。面向对象的程序实际上是用数据控制对代码的访问。下面你将看到，将控制的实体变换为数据，可使程序在组织结构上从若干方面受益。

2.1.2　抽象

面向对象编程的一个实质性的要素是抽象。人们通过抽象（abstraction）处理复杂性。例如，人们不会把一辆汽车想象成由几万个互相独立的部分所组成的一套装置，而是把汽车想成一个具有自己独特行为的对象。这种抽象使人们可以很容易地将一辆汽车开到杂货店，而不会因组成汽车各部分零件过于复杂而不知所措。他们可以忽略引擎、传动及刹车系统的工作细节，将汽车作为一个整体来加以利用。

使用层级分类是管理抽象的一个有效方法。它允许你根据物理意义将复杂的系统分解

为更多更易处理的小块。从外表看，汽车是一个独立的对象。一旦到了内部，你会看到汽车由若干子系统组成：驾驶系统，制动系统，音响系统，安全带，供暖，便携电话，等等。再进一步细分，这些子系统由更多的专用元件组成。例如，音响系统由一台收音机、一个CD播放器、或许还有一台磁带放音机组成。从这里得到的重要启发是，你通过层级抽象对复杂的汽车（或任何另外复杂的系统）进行管理。

复杂系统的分层抽象也能被用于计算机程序设计。传统的面向过程程序的数据经过抽象可用若干个组成对象表示，程序中的过程步骤可看成是在这些对象之间进行消息收集。这样，每一个对象都有它自己的独特行为特征。你可以把这些对象当作具体的实体，让它们对告诉它们做什么事的消息作出反应。这是面向对象编程的本质。

面向对象的概念是Java 的核心，对程序员来讲，重要的是要理解这些概念怎么转化为程序。你将会发现，在任何主要的软件工程项目中，软件都不可避免地要经历概念提出、成长、衰老这样一个生命周期，而面向对象的程序设计，可以使软件在生命周期的每一个阶段都处变不惊，有足够的应变能力。例如，一旦你定义好了对象和指向这些对象的简明的、可靠的接口，你就能很从容很自信地解除或更替旧系统的某些组成部分。

2.1.3 面向对象编程的3个原则

所有面向对象的编程语言都提供帮助你实现面向对象模型的机制，这些机制是封装，继承及多态性。现在让我们来看一下它们的概念。

封装

封装（Encapsulation）是将代码及其处理的数据绑定在一起的一种编程机制，该机制保证了程序和数据都不受外部干扰且不被误用。理解封装性的一个方法就是把它想成一个黑匣子，它可以阻止在外部定义的代码随意访问内部代码和数据。对黑匣子内代码和数据的访问通过一个适当定义的接口严格控制。如果想与现实生活中的某个事物作对比，可考虑汽车上的自动传送。自动传送中包含了有关引擎的数百比特的信息，例如你正在以什么样的加速度前进，你行驶路面的坡度如何，以及目前的档位。作为用户，你影响这个复杂封装的方法仅有一个：移动档位传动杆。例如，你不能通过使用拐弯信号或挡风玻璃擦拭器影响传动。所以档位传动杆是把你和传动连接起来的惟一接口。此外，传动对象内的任何操作都不会影响到外部对象，例如，档位传动装置不会打开车前灯！因为自动传动被封装起来了，所以任何一家汽车制造商都可以选择一种适合自己的方式来实现它。然而，从司机的观点来看，它们的用途都是一样的。与此相同的观点能被用于编程。封装代码的好处是每个人都知道怎么访问它，但却不必考虑它的内部实现细节，也不必害怕使用不当会带来负面影响。

Java封装的基本单元是类。尽管类将在以后章节详细介绍。现在仍有必要对它作一下简单的讨论。一个类（class）定义了将被一个对象集共享的结构和行为（数据和代码）。一个给定类的每个对象都包含这个类定义的行为和结构，好像它们是从同一个类的模子中铸造出来似的。因为这个原因，对象有时被看作是类的实例（instances of a class）。所以，类是一种逻辑结构，而对象是真正存在的物理实体。

当创建一个类时，你要指定组成那个类的代码和数据。从总体上讲，这些元素都被称

为该类的成员（members）。具体地说，类定义的数据称为成员变量（member variables）或实例变量（instance variables）。操作数据的代码称为成员方法（member methods）或简称方法（methods）。如果你对C/C++熟悉，可以这样理解：Java程序员所称的方法，就是C/C++程序员所称的函数（function）。在完全用Java编写的程序中，方法定义如何使用成员变量。这意味着一个类的行为和接口是通过方法来定义的，类这些方法对它的实例数据进行操作。

既然类的目的是封装复杂性，在类的内部就应该有隐藏实现复杂性机制。类中的每个方法或变量都可以被标记为私有（private）或公共（public）。类的公共接口代表类的外部用户需要知道或可以知道的每件事情；私有方法和数据仅能被一个类的成员代码所访问，其他任何不是类的成员的代码都不能访问私有的方法或变量。既然类的私有成员仅能被程序中的其他部分通过该类的公共方法访问，那么你就能保证不希望发生的事情就一定不会发生。当然，公共接口应该小心仔细设计，不要过多暴露类的内部内容（见图2-1）。

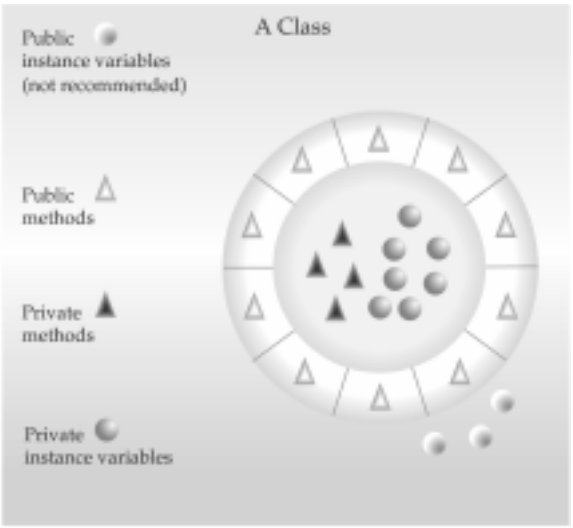


图 2-1　封装：可用来保护私有数据的公共方法

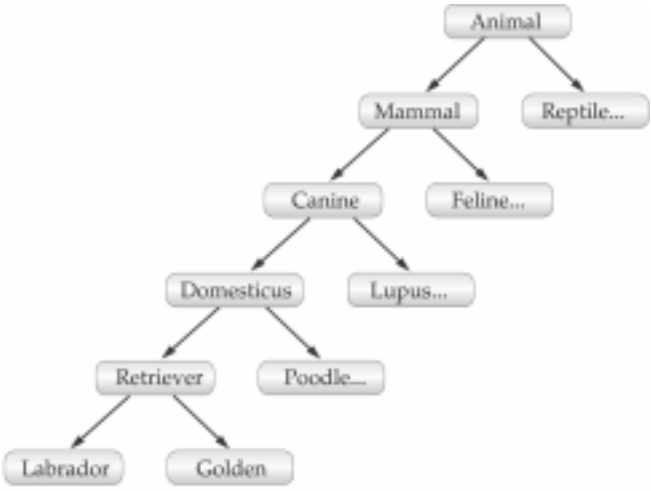
继承

继承（Inheritance）是一个对象获得另一个对象的属性的过程。继承很重要，因为它支持了按层分类的概念。如前面提到的，大多数知识都可以按层级（即从上到下）分类管理。例如，尊贵的猎犬是狗类的一部分，狗又是哺乳动物类的一部分，哺乳动物类又是动物类的一部分。如果不使用层级的概念，我们就不得不分别定义每个动物的所有属性。使用了继承，一个对象就只需定义使它在所属类中独一无二的属性即可，因为它可以从它的父类那儿继承所有的通用属性。所以，可以这样说，正是继承机制使一个对象成为一个更具通用类的一个特定实例成为可能。下面让我们更具体地讨论这个过程。

大多数人都认为世界是由对象组成的，而对象又是按动物、哺乳动物和狗这样的层级结构相互联系的。如果你想以一个抽象的方式描述动物，那么你可以通过大小、智力及骨

胳系统的类型等属性进行描述。动物也具有确定的行为，它们也需要进食、呼吸，并且睡觉。这种对属性和行为的描述就是对动物类的定义。

如果你想描述一个更具体的动物类，比如哺乳动物，它们会有更具体的属性，比如牙齿类型、乳腺类型等。我们说哺乳类动物是动物的子类（subclass），而动物是哺乳动物的超类（superclass）。



由于哺乳动物类是需要更加精确定义的动物，所以它可以从动物类继承（inherit）所有的属性。一个深度继承的子类继承了类层级（class hierarchy）中它的每个祖先的所有属性。

继承性与封装性相互作用。如果一个给定的类封装了一些属性，那么它的任何子类将具有同样的属性，而且还添加了子类自己特有的属性（见图2-2）。这是面向对象的程序在复杂性上呈线性而非几何性增长的一个关键概念。新的子类继承它的所有祖先的所有属性。它不与系统中其余的多数代码产生无法预料的相互作用。

多态性

多态性（Polymorphism，来自于希腊语，表示“多种形态”）是允许一个接口被多个同类动作使用的特性，具体使用哪个动作与应用场合有关，下面我们以一个后进先出型堆栈为例进行说明。假设你有一个程序，需要3种不同类型的堆栈。一个堆栈用于整数值，一个用于浮点数值，一个用于字符。尽管堆栈中存储的数据类型不同，但实现每个栈的算法是一样的。如果用一种非面向对象的语言，你就要创建3个不同的堆栈程序，每个程序一个名字。但是，如果使用Java，由于它具有多态性，你就可以创建一个通用的堆栈程序集，它们共享相同的名称。

多态性的概念经常被说成是“一个接口，多种方法”。这意味着可以为一组相关的动作设计一个通用的接口。多态性允许同一个接口被必于同一类的多个动作使用，这样就降低了程序的复杂性。选择应用于每一种情形的特定的动作（specific action）（即方法）是编译器的任务，程序员无需手工进行选择。你只需记住并且使用通用接口即可。

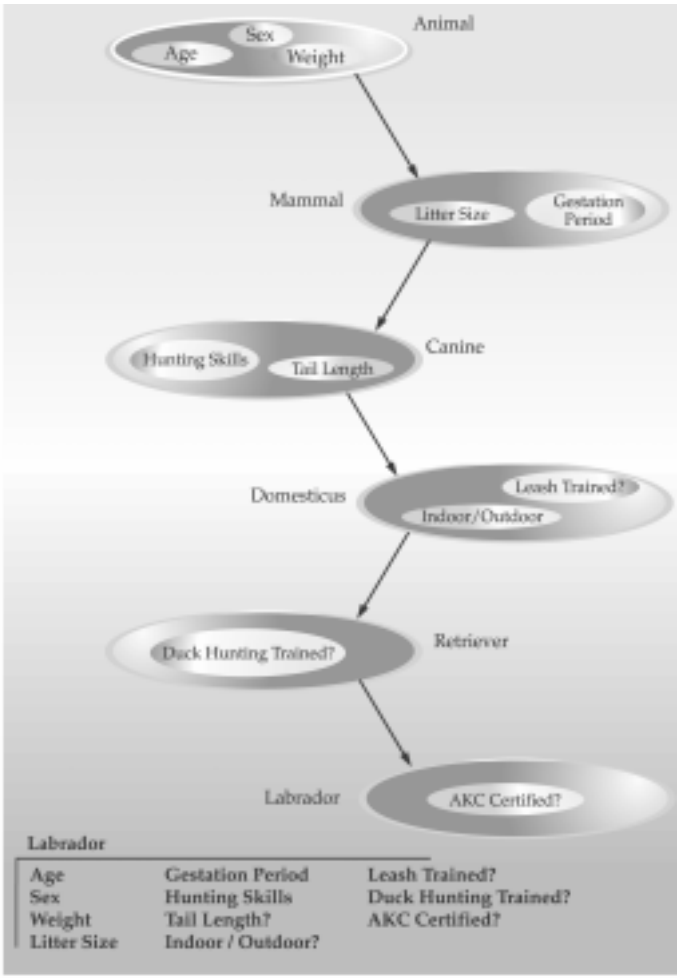


图 2-2　拉不拉多猎犬继承所有其超类的封装

再拿狗作比喻，一条狗的嗅觉是多态的。如果狗闻到猫的气味，它会在吠叫并且追着它跑。如果狗闻到食物的气味，它将分泌唾液并向盛着食物的碗跑去。两种状况下是同一种嗅觉器官在工作，差别在于闻到了什么气味，也就是有两种不同类型的数据作用于狗的鼻子！在一个Java程序中使用方法时，也可以采用这个通用的概念。

多态性、封装性与继承性相互作用

如果用得当，在由多态性、封装性和继承性共同组成的编程环境中可以写出比面向过程模型环境更健壮、扩展性更好的程序。精心设计的类层级结构是重用你花时间和努力改进并测试过的程序的基础，封装可以使你在不破坏依赖于类公共接口的代码基础上对程序进行升级迁移，多态性则有助于你编写清楚、易懂、易读、易修改的程序。

在前面两个与现实生活有关的实例中，汽车更能全面说明面向对象设计的优点，为介绍继承而用狗作类比也很有趣。总的来说，汽车与程序很相似，所有的驾驶员依靠继承性很快便能掌握驾驶不同类型（子类）车辆的技术。不管是接送学生的校车，或是默西迪斯

私家轿车，或是保时捷汽车，或是家庭汽车，司机差不多都能找到方向盘、制动闸和加速器，并知道如何操作。经过一段驾驶，大多数人甚至能知道手动档与自动档之间的差别，因为他们从根本上理解这两个档的超类——传动。

人们在汽车上看见的总是封装好的特性。刹车和踏脚板隐蔽着不可思议的复杂性，但接口却是如此简单，你的脚就可以操作它们！引擎、制动闸及轮胎的大小对于你如何定义踏脚板类的接口没有任何影响。

最后的属性，多态性，在汽车制造商基于相同的交通工具所提供的多种选择的能力上得到了充分反映。例如，刹车系统有正锁和反锁之分，方向盘有带助力或不带助力之分，引擎有4缸、6缸或8缸之分。无论设置如何，你都得脚踩刹车板来停车，转动方向盘来转向，按离合器来制动。同样的接口能被用来控制许多不同的实现过程。

正如你所看到的，通过封装、继承及多态性原理，各个独立部分组成了汽车这个对象。这在计算机程序设计中也是一样的。通过面向对象原则的使用，可以把程序的各个复杂部分组合成一个一致的、健壮的、可维护的程序整体。

正如本节开始时提到的，所有的Java程序都是面向对象的。或者，更精确地说，每个Java程序都具有封装性、继承性及多态性。尽管在本章将要介绍的简单示例程序及以后几章的示例程序中并未体现所有这些特性，但也有所体现。你将看到，Java提供的许多特性是它的内置类库的一部分，这个库使封装性、继承性及多态性得到更广泛应用。

2.2 第1个简单程序

既然Java面向对象的基础已经被讨论过了，接下来让我们看一些实际的Java程序。让我们从编译及运行下面这个简短示例程序开始。你将看到，这个程序的功能比你想像的要多。

```
/*
This is a simple Java program.
Call this file "Example.java".
*/
class Example {
  // Your program begins with a call to main().
  public static void main(String args[]) {
System.out.println("This is a simple Java program.");
  }
}
```

注意：在下面的介绍中，将使用标准JDK（Java Developer's Kit，Java 开发工具包），它可从Sun Microsystems公司得到。如果你正在使用其他的Java开发环境，则Java程序编译、运行过程可能有所不同。在这种情况下，请你查阅编译器的用户手册来获得具体指导。

2.2.1 键入程序

对大多数计算机语言，包含程序源代码的文件名是任意的，但对于Java就不行。关于Java，你需要知道的第一件事就是源文件的名字非常重要。对这个例子，源程序文件名应

该是Example.java。下面我们将解释其中的原因。

在Java中，一个源程序文件被称为一个编译单元（compilation unit）。它是一个包含一个或多个类定义的文本文件。Java编译器要求源程序文件使用.java文件扩展名。请注意，文件扩展名长度是4个字符。所以，你的操作系统一定要有支持长文件名的能力。这意味着DOS和Windows 3.1是不支持Java的（至少在目前是这样)。当然，它可在Windows 95/98和Windows NT/2000下正常工作。

从上述示例程序中可以看出，程序中定义的类名也是Example。这不是巧合。在Java中，所有的代码都必须驻留在类中。按照约定，类名必须与源程序的文件名相同。你也应该确保文件名的大小写字母与类名一样，这是因为Java是区分大小写的。虽然文件名与类名必须一致的约定似乎有点专制，但是这个约定有助于你轻松地维护及组织程序。

2.2.2　编译程序

要编译示例程序Example，须运行编译器程序javac，并在命令行上指定源程序文件名，格式如下所示：

```
C:\>javac Example.java
```

编译器javac产生了一个名为Example.class的文件，该文件包含程序的字节码。前面已讨论过，Java字节码中包含的是Java解释程序将要执行的指令码。因此，javac的输出并不是可以直接运行的代码。

要真正运行该程序，你必须使用名叫java的Java解释器。具体做法是把类名Example作为一个命令行参数输入，格式如下所示：

```
C:\>java Example
```

运行这个程序，将输出如下内容：

```
This is a simple Java program.
```

当Java源代码被编译后，每个单独的类都被放入自己的输出文件中，并以类的名字加“.class”扩展名为其文件名。这就是为什么Java源程序文件必须与其中包含的类同名的原因——源程序文件将与“.class”文件相同。运行Java解释器实际上是指定你想要解释器运行的类的名字，它会自动搜索包含该名字且带有.class扩展名的文件。如果找到，它将运行包含在该指定类中的代码。

2.2.3　详细讨论第1个示例程序

尽管Example.java很短，但它包括了所有Java程序具有的几个关键特性。让我们仔细分析该程序的每个部分。

程序开始于以下几行：

```
/*
   This is a simple Java program.
   Call this file "Example.java".
*/
```

这是一段注释（comment）。像大多数其他的编程语言一样，Java也允许你在源程序文件中加注释。注释中的内容将被编译器忽略。事实上，注释是为了给任何阅读源代码程序的人说明或解释程序的操作。在本例中，注释对程序进行说明，并提醒你该源程序的名字叫做Example.java。当然，在真正的应用中，注释通常用来解释程序的某些部分如何工作或某部分的特殊功能。

Java支持3种类型的注释。在示例程序顶部的注释称为多行注释（multiline comment）。这类注释开始于“/*”，结束于“*/”。这两个注释符间的任何内容都将被编译器忽略。正如“多行注释”名字所示，一个多行注释可以包含若干行文字。

程序的下一行代码如下所示：

```
class Example {
```

该行使用关键字class声明了一个新类，Example是类名标识符，整个类定义（包括其所有成员）都将位于一对花括号（{}）之间，花括号在Java中的使用方式与C或C++相同，目前，不必考虑类的细节，只是有一点要注意，在Java中，所有程序活动都发生在类内，这就是为什么Java程序是面向对象的。

下面一行程序是单行注释：

```
// Your program begins with a call to main().
```

这是Java支持的第二种类型的注释。单行注释（single-line comment）始于“//”，在该行的末尾结束。通常情况下，程序员们对于较长的注释使用多行注释，而对于简短的、一行一行的注释则使用单行注释。

下一行代码如下所示：

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

该行开始于main( )方法。正如它前面的注释所说，这是程序将要开始执行的第一行。所有的Java应用程序都通过调用main( )开始执行（这一点同C/C++一样），我们在此还不能对该行的每一个部分作出精确的解释，因为这需要详细了解Java封装性的特点，但是，由于本书第1部分中的大多数例子都用到这一行代码，我们将对各部分作一个简单介绍。

关键字public是一个访问说明符（access specifier），它允许程序员控制类成员的可见性。如果一个类成员前面有public，则说明该成员能够被声明它的类之外的代码访问（与public相对的是private，它禁止成员被所属类之外的代码访问）。在本例中，main( )必须被定义为public类型，因为当程序开始执行时它需要被它的类之外的代码调用。关键字static允许调用main( )而不必先实现该类的一个特殊实例。这是必要的，因为在任何对象被创建之前，Java解释器都会调用main()。关键字void仅通知编译器main()不返回任何值。你将看到，方法也可以有返回值。如果这一切似乎有一点令人费解，别担心。所有这些概念都将在随后的章节中详细讨论。

前面已经介绍过，main()是Java程序开始时调用的方法。请记住，Java是区分大小写的。因此，main与Main是不同的。Java编译器也可以编译不包含main()方法的类，但是Java解释程序没有办法运行这些类。因此，如果你输入了Main而不是main，编译器仍将编译你的程序，但Java解释程序将报告一个错误，因为它找不到main()方法。

你要传递给方法的所有信息由方法名后面括号中指定的变量接收，这些变量被称为参数（parameters）。即使一个方法不需要参数，你仍然需要在方法名后面放置一对空括号。在main()中，只有一个参数，即String args[]，它声明了一个叫做args的参数，该参数是String类的一个实例数组（注：数组是简单对象的集合）。字符串类型的对象存储字符的串。在本例中，args接收程序运行时显示的任何命令行参数。本例中的这个程序并没有使用这些信息，但是本书后面讲到的其他一些程序将使用它们。

该行的最后一个字符是“{”。它表示了main()程序体的开始。一个方法中包含的所有代码都将包括在这对花括号中间。

另外，main( )仅是解释器开始工作的地方。一个复杂的程序可能包含几十个类，但这些类仅需要一个main( )方法以供解释器开始工作。当你开始引用被嵌入在浏览器中的Java小应用程序时，你根本不用使用main( )方法，因为Web浏览器使用另一种不同的方法启动小应用程序。

接下来的代码行如下所示。请注意，它出现在main( )内。

```
System.out.println("This is a simple Java program.");
```

本行在屏幕上输出字符串“This is a simple Java program.”，输出结果后面带一个空行。输出实际上是由内置方法 println ( )来实现的。在本例中，println ( )显示传递给它的字符串。你将会看到，println ( )方法也能用来显示其他类型的信息。该行代码开始于System.out，现在对它作详细说明为时尚早，需涉及很多复杂内容。简单的说，System是一个预定义的可访问系统的类，out是连接到控制台的输出流。

可能你已经猜到了，控制台输出（输入）在实际的Java程序和小应用程序中并不经常使用。因为绝大多数现代计算环境从本质上讲都是窗口和图形界面的，控制台I/O主要被用简单的实用工具程序和演示程序使用。在本书后面，你将会学到用Java生成输出的其他方法。但是目前，我们将继续使用控制台I/O方法。

请注意，println( )语句以一个分号结束。在Java中，所有的语句都以一个分号结束。该程序的其他行没有以分号结束，这是因为从技术上讲，它们并不是程序语句。

程序中的第一个“}”号结束了main( )，而最后一个“}”号结束类Example的定义。

2.3　第2个示例程序

对于编程语言来说，变量是一个最为基本的概念。你可能知道，变量是一个有名字的内存位置，它能够被赋值。而且，在程序的运行过程中，变量的值是可以改变的。下一个程序将介绍如何声明变量，如何给变量赋值。另外，该程序也说明了控制台输出的某些新特点。从程序开始的注释可以看出，你应该把这个文件命名为Example2.java。

```
/*
  Here is another short example.
  Call this file "Example2.java".
*/
class Example2 {
  public static void main(String args[]) {
```

```
    int num; // this declares a variable called num

    num = 100; // this assigns num the value 100

    System.out.println("This is num: " + num);

    num = num * 2;

    System.out.print("The value of num * 2 is ");
    System.out.println(num);
  }
}
```

运行该程序时，你将会看到如下的运行结果：

```
This is num: 100
The value of num * 2 is 200
```

让我们来进一步查看这个结果是如何产生的。我们重点考虑与前一示例不同的代码，在上一个程序中未出现的第一行代码是：

```
int num; // this declares a variable called num
```

该行声明了一个名为num的整型变量。和其他大多数语言一样，在Java中一定要先声明变量，然后再使用变量。

下面是声明变量的一般形式：

type var-name;

在这里，type表示所要声明的变量的类型，var-name是所要声明变量的名称。如果你要声明多个属于同一类型的变量，只需用逗号将各个变量名分开即可。Java定义了几种数据类型：整型（integer），字符型（character），浮点型（floating-point）。关键字int指的是整数类型。

在程序中，下面这一行代码将100赋予变量num。

```
num = 100; // this assigns num the value 100
```

在Java中，赋值符号是等号。

下面的这行程序在输出变量值之前，先输出字符串“This is num:”。

```
System.out.println("This is num: " + num);
```

在这个语句中，变量num之前的加号“+”的作用是，让num的取值与它前面的字符串相连接，然后再输出结果字符串的内容（实际上，变量num先被它赋值再超值转换成字符串，然后再和加号之前的字符串相连接。这个过程将在本书的后面详细讨论）。这种方法可以被推广。通过加号“+”的连接操作，你可以在println（）这个方法之内将尽可能多的字符串内容连在一起。

接下来的语句行将变量num乘2以后的结果重新赋值给变量num。和其他大多数语言一样，Java用“*”符号来表示乘法运算。在执行这行语句之后，变量num的值变成了200。

本程序接下来的两行代码是：

```
System.out.print("The value of num * 2 is ");
System.out.println(num);
```

在这两行中有几个新内容。首先，内置方法print( )被用来显示字符串“The value of num * 2 is”。该字符串后面不换行，这意味着如果生成第二个输出，它将在同一行中开始输出。方法print ( )和方法println ( )类似，只是它在每次调用后并不输出一个新行（即换行）。其次，在调用println ( )时，注意变量num可以被自身使用。方法print ( )和方法println ( )都能够用来输出Java的任何内置类型的值。

2.4　两个控制语句

尽管将在第5章仔细讨论控制语句，我们还是在这里先简单介绍2条控制语句，以便能在第3章、第4章中的例子程序中使用它们，并且它们也将帮助说明Java的一个重要特点：程序块。

2.4.1　if控制语句

Java中if控制语句与其他语言中的IF语句非常相似。并且，它与C/ C++语言中的if语句的语法完全相同。它的最简单形式如下：

if(condition) statement;

这里，条件condition是一个布尔型表达式。如果条件为真，那么执行语句statement；如果条件为假，则语句statement将被绕过而不被执行。下面是一个例子：

```
if(num < 100) println("num is less than 100");
```

在这个例子中，如果变量num的值小于100，那么条件表达式的值为真，方法 println ( )将被调用执行。如果变量num的值大于或等于100，那么方法println ( )被绕过而不被执行 。

在第4章，中你将看到Java在条件语句中用到的所有的关系运算符，下面是其中一部分：

运算符	含　义
<	小于
>	大于
==	等于

注意，判断是否相等的关系运算符是两个等号“==”。

下面的程序说明了if控制语句的用法：

```
/*
  Demonstrate the if.

  Call this file "IfSample.java".
*/
class IfSample {
  public static void main(String args[]) {
```

```
    int x, y;

    x = 10;
    y = 20;

    if(x < y) System.out.println("x is less than y");

    x = x * 2;
    if(x == y) System.out.println("x now equal to y");

    x = x * 2;
    if(x > y) System.out.println("x now greater than y");
    // this won't display anything
    if(x == y) System.out.println("you won't see this");
  }
}
```

该程序产生的结果如下所示：

```
x is less than y
x now equal to y
x now greater than y
```

这个程序中另一个需要注意的地方是：

```
int  x , y ;
```

该程序行使用逗号来分隔变量列表，定义了2个变量x和y。

2.4.2 for循环

你可能从先前的编程经验已经知道，在几乎所有的编程语言中，循环语句都是其重要组成部分。Java也不例外。事实上，你将在第5章中看到，Java提供了一套功能强大的循环结构。For循环也许是最通用的。如果你对C或C++熟悉，那么你应该感到高兴，因为Java的for循环和其他语言中的for循环操作完全一样。如果你不熟悉C/C++，for循环也是容易使用的。最简单的for循环结构如下所示：

```
for(initialization; condition; iteration) statement;
```

在这个最常见的形式中，循环体的初始化部分（initialization）设置循环变量并为变量赋初始值。条件判断部分（condition）是测试循环控制变量的布尔表达式。如果测试的结果为真，循环体（statement）继续反复执行；如果测试的结果为假，循环结束。迭代部分（iteration）的表达式决定循环控制变量在每次循环后是如何改变的。下面的短程序说明了for循环的使用方法：

```
/*
  Demonstrate the for loop.

  Call this file "ForTest.java".
*/
class ForTest {
  public static void main(String args[]) {
```

```
    int x;

    for(x = 0; x<10; x = x+1)
      System.out.println("This is x: " + x);
  }
}
```

这个程序产生的结果如下：

```
This is x: 0
This is x: 1
This is x: 2
This is x: 3
This is x: 4
This is x: 5
This is x: 6
This is x: 7
This is x: 8
This is x: 9
```

在这个例子中，x是循环控制变量。它在for的初始化部分被初始化为零。在每次重复迭代（包括第一次）的开始，执行条件测试x< 10。如果测试的结果为真，println ( )语句被执行，然后执行循环体的迭代部分。这个过程将持续进行下去，直到条件测试的结果为假。

有趣的是，在Java专业程序员编写的程序中，循环体的迭代部分一般不会像前面程序示例那样。即你很少会看到下面的语句：

```
x = x + 1;
```

原因是Java有一个特殊的增量运算符，能够更高效地执行这项操作。该增量运算符是“++”(即2个加号）。递增运算符每次使其作用对象加1。通过使用递增运算符，上条语句可以这样写：

```
x++;
```

这样，前述的for循环语句通常写成这样：

```
for(x = 0; x<10; x++)
```

你可以将上一个程序的for循环语句改写成这样试一下。你将看到，运行结构与以前相同。

Java也提供一个递减运算符：“--”（即2个减号）。递减运算符使其作用对象每次减1。

2.5　使用程序块

在Java中，可以将2个或2个以上的语句组成一组，这样的一组语句称为程序块（Codeblocks）。程序块是通过将所属语句放在花括号中来实现。一旦创建了程序块，它就成为一个逻辑单元，可以作为一个单独的语句来使用。例如，程序块可以作为Java中if控制语句和for控制语句的目标。我们来看一看下面的if控制语句：

```
if(x < y) { // begin a block
  x = y;
  y = 0;
} // end of block
```

本例中，如果x小于y，那么在程序块内的两条语句都将执行。因此，程序块中的这2条语句组成一个逻辑单元，不能一条语句运行，而另一条语句不运行。其中的关键一点是如果你需要将两个或多个语句在逻辑上连接起来 ，你就可以将其放入一个程序块中。

让我们看另外的例子。下面的程序将for循环作为一个程序块使用。

```
/*
  Demonstrate a block of code.

  Call this file "BlockTest.java"
*/
class BlockTest {
  public static void main(String args[]) {
    int x, y;

    y = 20;

    // the target of this loop is a block
    for(x = 0; x<10; x++) {
      System.out.println("This is x: " + x);
      System.out.println("This is y: " + y);
      y = y - 2;
    }
  }
}
```

这个程序产生的结果如下所示：

```
This is x: 0
This is y: 20
This is x: 1
This is y: 18
This is x: 2
This is y: 16
This is x: 3
This is y: 14
This is x: 4
This is y: 12
This is x: 5
This is y: 10
This is x: 6
This is y: 8
This is x: 7
This is y: 6
This is x: 8
This is y: 4
This is x: 9
This is y: 2
```

在本例中，for循环作为一个程序块使用，而并不是一个单独的语句。这样，每循环一

次，块内的3条语句都要运行一次。这个事实当然被程序的执行结果证实了。

在本书的后面，你会看到程序块的其他性质和用法。当然，它们存在的主要原因是为了创建逻辑上独立的代码单元。

2.6　基 本 词 汇

既然你已经看过了几个短的Java程序，现在让我们更正式的介绍Java的基本元素。Java程序由空白分隔符、标识符、注释、文字、运算符、分隔符，以及关键字组成。运算符将在下一章详细讨论，本节讨论其他的元素。

2.6.1　空白分隔符（whitespace）

Java 是一种形式自由的语言。这意味着你不需要遵循任何特殊的缩进书写规范。例如,例子程序的所有代码都可以在一行上，你也可以按自己喜欢的方式输入程序代码，前提是必须在已经被运算符或分隔符描述的标记之间至少留出一个空白分隔符。在Java中，空白分隔符可以是空格，Tab跳格键或是换行符。

2.6.2　标识符（identifiers）

标识符是赋给类、方法或是变量的名字。一个标识符可以是大写和小写字母、数字、下划线、美元符号的任意顺序组合，但不能以一个数字开始。否则容易与数字、常量相混淆。再次强调一下，Java是区分大小写的，VALUE和Value是两个不同的标识符。下面是一些有效的标识符：

```
AvgTemp        count        a4        $test        this_is_ok
```

下面是一些无效的变量名：

```
2count        high-temp        Not/ok
```

2.6.3　常量（literal）

在Java中，常量用literal表示。例如，下面是一些常量：

```
100        98.6        'X'        "This is a test"
```

从左到右，第一个表示一个整数，第二个是浮点值，第三个是一个字符常数，最后是一个字符串。常量能在任何地方被它所允许的类型使用，代表的是所属类型的一个值。

2.6.4　注释(comments)

Java定义了3种注释的类型。其中2种注释类型你已经知道了：单行注释和多行注释。第3种注释类型被称为文档注释（documentation comment）。这类注释以HTML文件的形式为你的程序作注释。文档注释以“/**”开始，以“*/”结束。在附录A中对文档注释作了解释。

2.6.5 分隔符（separators）

在Java中 ，有一些字符被当作分隔符使用，最常用的分隔符是分号(；)，用来分隔语句。下面是常用的分隔符。

符 号	名称	用途
（）	圆括号	在定义和调用方法时用来容纳参数表。在控制语句或强制类型转换组成的表达式中用来表示执行或计算的优先权
{ }	花括号、大括号	用来包括自动初始化的数组的值。也用来定义程序块、类、方法以及局部范围
[ ]	方括号、中括号	用来声明数组的类型，也用来表示撤消对数组值的引用
；	分号	用来终止一个语句
，	逗号	在变量声明中，用于分隔变量表中的各个变量。在for控制语句中，用来将圆括号内的语句连接起来
.	句号（点）	用来将软件包的名字与它的子包或类分隔。也用来将引用变量与变量或方法分隔

2.6.6 Java关键字

目前Java语言一共定义了48个保留关键字（参见表2-1 ）。这些关键字与运算符和分隔符的语法一起构成Java语言的定义。这些关键字不能用于变量名、类名或方法名。

关键字const和goto虽然被保留但未被使用。在Java语言的早期，还有几个其他关键字被保留以备以后使用。但是目前Java定义的关键字如表2-1所示。

除了上述关键字，Java还有以下保留字：true，false，null。这些词是Java定义的值。你也不能用这些词作为变量名，类名等等。

表 2-1 Java 保留关键字

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

2.6.7 Java类库

在本章的示例程序中用到了Java的两个内置方法：println ( )和print( )。前面提到过，这些方法是System类的成员，它已经被Java预定义且自动地包括在你的程序中。Java环境依靠几个内置的类库，这些类库包含许多内置的方法，用以提供对诸如输入/输出（I/O）、字符

串处理、网络、图形的支持。标准的类还提供对窗口输出的支持。因此，作为一个整体，Java是Java语言本身和它的标准类的组合体。你将会看到，Java类库提供了Java的许多功能。毫无疑问，要成为一个Java程序员，其中的一部分工作就是学会使用标准的Java类。在本书第1部分，需要时会对标准库类库和方法的各种元素进行介绍。在本书的第2部分，将对类库作详细地描述。

第 3 章　数据类型、变量、数组

本这章分析Java语言中3个最基本的元素:数据类型，变量和数组。就像所有的现代编程语言一样，Java支持多种数据类型。你可以使用这些类型声明变量或创建数组。你将看到，Java对这些项目的处理方法是清楚、有效且连贯的。

3.1　Java语言是强类型语言语言

首先我们要声明Java语言强类型语言的重要性。确实，Java的安全和健壮性部分来自于它是该类型语言这一事实。让我们看这意味着什么。首先，每个变量有类型，每个表达式有类型，而且每种类型是严格定义的。其次，所有的数值传递，不管是直接的还是通过方法调用经由参数传过去的都要先进行类型相容性的检查。有些语言没有自动强迫进行数据类型相容性的检查或对冲突的类型进行转换的机制。Java编译器对所有的表达式和参数都要进行类型相容性的检查以保证类型是兼容的。任何类型的不匹配都是错误的，在编译器完成编译以前，错误必须被改正。

注意：如果你有C或C++的背景，一定要记住Java对数据类型兼容性的要求比任何语言都要严格。例如，在C/C++中你能把浮点型值赋给一个整数。在Java中则不能。另外，C语言中，在一个参数和一个自变量之间没有必然的强制的类型检查。在Java中则有。起初你可能发现Java的强制类型检查有点繁烦。但是要记住，从长远来说它将帮助你减少程序出错的可能性。

3.2　简单数据类型

Java定义了8个简单（或基本）的数据类型：字节型（byte），短整型（short），整型（int），长整型（long），字符型（char），浮点型（float），双精度型（double），布尔型（boolean），这些类型可分为4组：

- 整数：该组包括字节型（byte），短整型（short），整型（int），长整型（long），它们有符号整数。
- 浮点型数：该组包括浮点型（float），双精度型（double），它们代表有小数精度要求的数字。
- 字符：这个组包括字符型（char），它代表字符集的符号，例如字母和数字。
- 布尔型：这个组包括布尔型（boolean），它是一种特殊的类型，表示真/假值。

你可以按照定义使用它们，也可以构造数组或类的类型来使用它们。这样，他们就形

成了你可能创建的所有其他类型数据的基础。

简单数据类型代表单值，而不是复杂的对象。Java是完全面向对象的，但简单数据类型不是。他们类似于其他大多数非面向对象语言的简单数据类型。这样做的原因是出于效率方面的考虑。在面向对象中引入简单数据类型不会对执行效率产生太多的影响。

简单类型的定义有明确的范围，而且有数学特性。像C和C++这样的语言，整数大小根据执行环境的规定而变化。然而，Java不是这样。因为Java可移植性的要求，所有的数据类型都有一个严格的定义的范围。例如，不管是基于什么平台，整型（int）总是32位。这样写的程序在任何机器体系结构上保证都可以运行。当然严格地指定一个整数的大小在一些环境上可能会损失性能，但为了达到可移植性，这种损失是必要的。

让我们依次讨论看每种数据类型。

3.3　整 数 类 型

Java定义了4个整数类型：字节型（byte），短整型（short），整型（int），长整型（long）。这些都是有符号的值，正数或是负数。Java 不支持仅仅是正的无符号的整数。许多其他计算机语言，包括C/C++，支持有符号或无符号的整数。然而，Java的设计者感到无符号整数是不必要的。具体地说，他们感到无符号（unsigned）概念主要被用来指定高位（high-order bit）状态，它定义了当int表示一个数字时的符号。你将在第4章中看到，Java对高位含义的管理是不同的，它通过增加一个专门的“无符号右移”运算符来管理高位。这样，就不需要无符号整数了。

整数类型的长度（width）不应该被理解为它占用的存储空间，而应该是该类变量和表达式的行为（behavior）。只要你对类型进行了说明，Java的运行环境对该类的大小是没有限制的。事实上，为了提高性能，至少字节型和短整型的存储是32位（而非8位和16位），因为这是现在大多数计算机使用的字的大小。

这些整数类型的长度和变化范围如表3-1所示：

表 3-1　整数的各种类型及特性

名称	长度	数的范围
长整型	64	–9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807
整型	32	–2,147,483,648 ~ 2,147,483,647
短整型	16	–32,768 ~ 32,767
字节型	8	–128~127

让我们分别讨论整数的每种类型。

3.3.1　字节型（byte）

最小的整数类型是字节型。它是有符号的8位类型，数的范围是-128～127。当你从网络或文件处理数据流的时候，字节类型的变量特别有用。当你处理可能与Java的其他内置类型不直接兼容的未加工的二进制的数据时，它们也是有用的。

通过“byte”这个关键字的使用来定义字节变量。例如，下面定义了2个变量，称为b和c：

```
byte b, c;
```

3.3.2 短整型（short）

short是有符号的16位类型，数的范围是 –32,768～32,767 。因为它被定义为高字节优先（称为big-endian格式），它可能是Java中使用得最少的类型。这种类型主要适用于16位计算机，然而这种计算机现在已经很少见了。

下面是声明Short变量的一些例子：

```
short s;
short t;
```

注意： “Endianness ”描述像short，int，和 long这些多字节数据类型是如何被存储在存储器中的。如果用2 个字节代表short，那么哪个字节在前，是高字节位(最重要的字节)还是低字节位(最不重要的字节)？说一台机器是big-endian，那意味着这个机器中最重要的字节在前，最不重要的字节在后。例如 SPARC 和 PowerPC 的机器是 big-endian ，而Intel x86 系列是 little-endian。

3.3.3 整型（int）

最常用的整数类型是int。它是有符号的32位类型，数的范围是-2,147,483,648～2,147,483,647。int类型的变量通常被用来控制循环及作数组的下标。任何时候你的整数表达式包含byte，short，int及字面量数字，在进行计算以前，所有表达式的类型被提升（promoted）到整型。

整型是最通用并且有效的类型，当你想要计数用作或数组下标或进行整数计算时，你应该使用整型。似乎使用字节型和短整型可以节约空间，但是不能保证 Java 不会内部把那些类型提升到整型。记住，类型决定行为，而不是大小（惟一的例外是数组，字节型的数据保证每个数组元素只占用一个字节，短整型使用 2 个字节，整型将使用4个。）

3.3.4 长整型（long）

long是有符号的64位类型，它对于那些整型不足以保存所要求的数值时是有用的。长整型数的范围是相当大的。这使得大的、整个数字都被需要时，它是非常有用的。例如，下面的程序是计算光在一个指定的天数旅行的英里数。

```
// Compute distance light travels using long variables.
class Light {
  public static void main(String args[]) {
    int lightspeed;
    long days;
    long seconds;
    long distance;

    // approximate speed of light in miles per second
```

```
        lightspeed = 186000;

        days = 1000; // specify number of days here

        seconds = days * 24 * 60 * 60; // convert to seconds

        distance = lightspeed * seconds; // compute distance

        System.out.print("In " + days);
        System.out.print(" days light will travel about ");
        System.out.println(distance + " miles.");
    }
}
```

这个程序运行的结果如下：

```
In 1000 days light will travel about 16070400000000 miles.
```

很清楚，计算结果超出了整型数的表达范围。

3.4　浮点型（Floating-Point Types）

浮点数字，也就是人们知道的实数（real），当计算的表达式有精度要求时被使用。例如，计算平方根，或超出人类经验的计算如正弦和余弦，它们的计算结果的精度要求使用浮点型。Java实现了标准（IEEE-754）的浮点型和运算符集。有2种浮点型，单精度浮点型（float）及双精度（double）浮点型。

他们的长度和变化范围如表3-2所示：

表 3-2　浮点型分类及其特性

名称	位数	数的范围
DOUBLE	64	1.7E–308～1.7E+308
float	32	3.4E–038～3.4E+038

下面讨论浮点型的每一种类型。

3.4.1　单精度浮点型（float）

单精度浮点型（float）专指占用32位存储空间的单精度（single-precision）值。单精度在一些处理器上比双精度更快而且只占用双精度一半的空间，但是当值很大或很小的时候，它将变得不精确。当你需要小数部分并且对精度的要求不高时，单精度浮点型的变量是有用的。例如，当表示美元和分时，单精度浮点型是有用的。

这是一些声明单精度浮点型变量的例子：

```
float hightemp, lowtemp;
```

3.4.2 双精度型（double）浮点型

双精度型，正如它的关键字“double”表示的，占用64位的存储空间。在一些现代的被优化用来进行高速数学计算的处理器上双精度型实际上比单精度的快。所有超出人类经验的数学函数，如sin( )，cos( )，和sqrt( )均返回双精度的值。当你需要保持多次反复迭代的计算的精确性时，或在操作值很大的数字时，双精度型是最好的选择。

下面的短程序用双精度浮点型变量计算一个圆的面积：

```
// Compute the area of a circle.
class Area {
  public static void main(String args[]) {
    double pi, r, a;

    r = 10.8; // radius of circle
    pi = 3.1416; // pi, approximately
    a = pi * r * r; // compute area
    System.out.println("Area of circle is " + a);
  }
}
```

3.5 字 符

在Java中，存储字符的数据类型是char 。但是，C/C++程序员要注意：Java的char与C或C++中的char不同。在C/C++中，char的宽是8位整数。但Java的情况不同。Java使用Unicode码代表字符。Unicode定义的国际化的字符集能表示迄今为止人类语言的所有字符集。它是几十个字符集的统一，例如拉丁文，希腊语，阿拉伯语， 古代斯拉夫语，希伯来语，日文片假名，匈牙利语等等，因此它要求16位。这样，Java中的char 类型是16位，其范围是0～65,536，没有负数的char。人们熟知的标准字符集ASCII 码的范围仍然是0～127，扩展的 8 位字符集ISO-Latin-1的范围是0～255。既然Java被设计为允许其开发的applet（小应用程序）在世界范围内使用，因此使用Unicode码代表字符是说得通的。当然，Unicode的使用对于英语、德语、西班牙语或法语的语言是有点低效，因为它们的字符能容易地被包含在8位以内。但是为了全球的可移植性，这一点代价是必须的。

注意：有关Unicode码的更多信息可在网址http://www.unicode.org上找到。

下面的程序示范了char 变量：

```
// Demonstrate char data type.
class CharDemo {
  public static void main(String args[]) {
    char ch1, ch2;

    ch1 = 88;  // code for X
    ch2 = 'Y';

    System.out.print("ch1 and ch2: ");
    System.out.println(ch1 + " " + ch2);
```

```
  }
}
```

该程序的输出结果如下：

```
ch1 and ch2: X Y
```

注意变量ch1被赋值88，它是ASCII码（Unicode码也一样）用来代表字母X的值。前面已提到，ASCII字符集占用了Unicode字符集的前127个值。因此以前你使用过的一些字符技巧在Java中同样适用。

尽管char不是整数，但在许多情况中你可以对它们进行运算操作就好像他们是整数一样。这允许你可以将2个字符相加，或对一个字符变量值进行增量操作。例如，考虑下面的程序：

```
// char variables behave like integers.
class CharDemo2 {
  public static void main(String args[]) {
    char ch1;

    ch1 = 'X';
    System.out.println("ch1 contains " + ch1);

    ch1++; // increment ch1
    System.out.println("ch1 is now " + ch1);
  }
}
```

这个程序的输出结果如下所示：

```
ch1 contains X
ch1 is now Y
```

在该程序中，ch1首先被赋值为X。然后变量ch1递增（自增量1）。结果是ch1变成了代表Y，即在ASCII（以及 Unicode）字符集中的下一个字符。

3.6　布 尔 型

Java有一种表示逻辑值的简单类型，称为布尔型。它的值只能是真或假这两个值中的一个。它是所有的诸如a<b这样的关系运算的返回类型。布尔类型对管理像if、for这样的控制语句的条件表达式也是必需的。

下面的程序说明了布尔类型的使用：

```
// Demonstrate boolean values.
class BoolTest {
  public static void main(String args[]) {
    boolean b;

    b = false;
    System.out.println("b is " + b);
    b = true;
```

```
        System.out.println("b is " + b);

        // a boolean value can control the if statement
        if(b) System.out.println("This is executed.");

        b = false;
        if(b) System.out.println("This is not executed.");

        // outcome of a relational operator is a boolean value
        System.out.println("10 > 9 is " + (10 > 9));
    }
}
```

这个程序的运行结果如下所示：

```
b is false
b is true
This is executed.
10 > 9 is true
```

关于这个程序有3件有趣的事情要注意。首先，你已经看到，当用方法println ( )输出布尔的值时，显示的是“true”或“false”。第二，布尔变量的值本身就足以用来控制if语句。没有必要将if语句写成像下面这样：

```
if(b == true) ...
```

第三，关系运算符（例如<）的结果是布尔值。这就是为什么表达式10>9的显示值是“true”。此外，在表达式10>9的两边额外的加上括号是因为加号“+”运算符比运算符“>”的优先级要高。

3.7 进一步研究字面量

在第2章中曾简要地提及字面量，现在已经讲述了内置的类型，让我们进一步研究它们。

3.7.1 整数字面量

整数可能是在典型的程序中最常用的类型。任何一个数字的值就是一个整数字面量。例如1，2，3和42。这些都是十进制的值，这意味着对他们的描述基于数字10。还有另外2种进制被整数字面量使用，八进制（octal，基数是8）和十六进制（hexadecimal，基数是16）。在Java中对八进制的值通过在它的前面加一个前导0来表示。正常的十进制的数字不用前导零。这样，看起来有效的值09将从编译器产生一个错误，因为9超出了八进制的范围0～7。程序员对数字更常用的是十六进制，它清楚地与8的大小相匹配，如8，16，32，和64位。

通过前导的0x或0X表示一个十六进制的字面量。十六进制数的范围是0～15，这样用A～ F（或a～f）来替代10～15。

整数字面量产生int 值，在Java中它是32位的整数值。既然Java对类型要求严格，你可能会纳闷，将一个整数字面量赋给Java的其他整数类型如byte或long而没有产生类型不匹配的错误，怎么可能呢。庆幸的是，这个问题很好解决。当一个字面量的值被赋给一个byte 或short型的变量时，如果字面量的值没有超过对应类型的范围时不会产生错误，所以，一个

字面量总是可以被赋给一个long变量。但是，指定一个long字面量，你需要明白地告诉编译器字面量的值是long型，你可以通过在字面量的后面加一个大写或小写的L来做到这一点。例如0x7fffffffffffffL或9223372036854775807L 就是long型中最大的。

3.7.2　浮点字面量

浮点数代表带小数部分的十进制值。他们可通过标准记数法或科学记数法来表示。标准记数法（Standard notation）由整数部分加小数点加小数部分组成。例如2.0，3.14159，和0.6667 都是有效的标准记数法表示的浮点数字。科学记数法（Scientific notation）是浮点数加一表明乘以10的指定幂次的后缀，指数是紧跟E或 e 的一个十进制的数字，它可以是正值或是负值。例子如6.022E23，314159E-05，及2e+100。

Java中的浮点字面量默认是双精度。为了指明一个浮点字面量，你必须在字面量后面加F或f。你也可以通过在字面量后面加D或d来指明一个浮点字面量。这样做当然是多余的。默认的双精度类型要占用64位存储空间，而精确低些的浮点类型仅仅需要32位。

3.7.3　布尔型字面量

布尔型字面量很简单。布尔型字面量仅仅有2个逻辑值，真或假。真值或假值不会改变任何数字的表示。Java中，真字面量的值不等于1，假字面量的值也不等于0，他们仅仅能被赋给已定义的布尔变量，或在布尔的运算符表达式中使用。

3.7.4　字符字面量

Java用Unicode字符集来表示字符。Java的字符是16位值，可以被转换为整数并可进行像加或减这样的整数运算。通过将字符包括在单引号之内来表示字符字面量。所有可见的ASCII字符都能直接被包括在单引号之内，例如'a'，'z'，and '@'。对于那些不能直接被包括的字符，有若干转义序列，这样允许你输入你需要的字符，例如'\'代表单个引号字符本身，'\n'代表换行符字符。为直接得到八进制或十六进制字符的值也有一个机制。对八进制来说，使用反斜线加3个阿拉伯数字。例如，'\141'是字母'a'。对十六进制来说，使用反斜线和u（\u）加4个十六进制阿拉伯数字。例如，'\u0061'因为高位字节是零，代表ISO-Latin-1字符集中的'a'。'\ua432'是一个日文片假名字符。表3-3列出了字符转义序列。

表 3-3　字符转义序列

转义序列	说明
\ddd	八进制字符（ddd）
\uxxxx	十六进制Unicode码字符
\'	单引号
\"	双引号
\\	反斜杠
\r	回车键
\n	换行

续表

转义序列	说明
\f	换页
\t	水平制表符
\b	退格

3.7.5 字符串字面量

Java中的字符串字面量和其他大多数语言一样——将一系列字符用双引号括起来。字符串的例子如：

```
"Hello World"
"two\nlines"
"\"This is in quotes\""
```

为字符串定义的字符转义序列和八进制/十六进制记法在字符串内的工作方法一样。关于Java字符串应注意的一件重要的事情是它们必须在同一行开始和结束。不像其他语言有换行连接转义序列。

注意：你可能知道，在大多数其他语言中，包括C/C++，字符串作为字符的数组被实现。然而，在Java中并非如此。在Java中，字符串实际上是对象类型。在这本书的后面你将看到，因为Java对字符串是作为对象实现的，因此，它有广泛的字符串处理能力，而且功能既强又好用。

3.8 变 量

变量是Java程序的一个基本存储单元。变量由一个标识符，类型及一个可选初始值的组合定义。此外，所有的变量都有一个作用域，定义变量的可见性，生存期。接下来讨论变量的这些元素。

3.8.1 声明一个变量

在Java中，所有的变量必须先声明再使用。基本的变量声明方法如下：

```
type identifier [ = value][, identifier [= value] ...] ;
```

type是Java的基本类型之一，或类及接口类型的名字（类和接口类型在本书第1部分的后部讨论）。标识符（identifier）是变量的名字，指定一个等号和一个值来初始化变量。请记住初始化表达式必须产生与指定的变量类型一样（或兼容）的变量。声明指定类型的多个变量时，使用逗号将各变量分开。

以下是几个各种变量声明的例子。注意有一些包括了初始化。

```
int a, b, c;                    // declares three ints, a, b, and c.
int d = 3, e, f = 5;            // declares three more ints, initializing
                                // d and f.
byte z = 22;                    // initializes z.
```

```
double pi = 3.14159;            // declares an approximation of pi.
char x = 'x';                   // the variable x has the value 'x'.
```

你选择的标识符名称没有任何表明它们类型的东西。许多读者记得FORTRAN预先规定从I到N的所有标识符都为整型，其他的标识符为实型。Java允许任何合法的标识符具有任何它们声明的类型。

3.8.2　动态初始化

尽管前面的例子仅将字面量作为其初始值，Java也允许在变量声明时使用任何有效的表达式来动态地初始化变量。

例如，下面的短程序在给定直角三角形两个直角边长度的情况下，求其斜边长度。

```
// Demonstrate dynamic initialization.
class DynInit {
    public static void main(String args[]) {
      double a = 3.0, b = 4.0;

      // c is dynamically initialized
      double c = Math.sqrt(a * a + b * b);

      System.out.println("Hypotenuse is " + c);
    }
}
```

这里，定义了3个局部变量a，b，c。前两个变量a和b初始化为常量。然而直角三角形的斜边c被动态地初始化（使用勾股定理）。该程序用了Java另外一个内置的方法sqrt()，它是Math类的一个成员，计算它的参数的平方根。这里关键的一点是初始化表达式可以使用任何有效的元素，包括方法调用、其他变量或字面量。

3.8.3　变量的作用域和生存期

到目前为止，我们使用的所有变量都是在方法main()的后面被声明。然而，Java允许变量在任何程序块内被声明。在第2章中已解释过了，程序块被包括在一对大括号中。一个程序块定义了一个作用域（scope）。这样，你每次开始一个新块，你就创建了一个新的作用域。你可能从先前的编程经验知道，一个作用域决定了哪些对象对程序的其他部分是可见的，它也决定了这些对象的生存期。

大多数其他计算机语言定义了两大类作用域：全局和局部。然而，这些传统型的作用域不适合Java的严格的面向对象的模型。当然将一个变量定义为全局变量是可行的，但这是例外而不是规则。在Java中2个主要的作用域是通过类和方法定义的。尽管类的作用域和方法的作用域的区别有点人为划定。因为类的作用域有若干独特的特点和属性，而且这些特点和属性不能应用到方法定义的作用域，这些差别还是很有意义的。因为有差别，类（以及在其内定义的变量）的作用域将被推迟到第6章当讨论类时再来讨论。到现在为止，我们将仅仅考虑由方法或在一个方法内定义的作用域。

方法定义的作用域以它的左大括号开始。但是，如果该方法有参数，那么它们也被包括在该方法的作用域中。本书在第5章将进一步讨论参数，因此，现在可认为它们与方法中

其他变量的作用域一样。

作为一个通用规则，在一个作用域中定义的变量对于该作用域外的程序是不可见（即访问）的。因此，当你在一个作用域中定义一个变量时，你就将该变量局部化并且保护它不被非授权访问和/或修改。实际上，作用域规则为封装提供了基础。

作用域可以进行嵌套。例如每次当你创建一个程序块，你就创建了一个新的嵌套的作用域。这样，外面的作用域包含内部的作用域。这意味着外部作用域定义的对象对于内部作用域中的程序是可见的。但是，反过来就是错误的。内部作用域定义的对象对于外部是不可见的。

为理解嵌套作用域的效果，考虑下面的程序：

```
// Demonstrate block scope.
class Scope {
    public static void main(String args[]) {
      int x; // known to all code within main

      x = 10;
      if(x == 10) { // start new scope
      int y = 20; // known only to this block

      // x and y both known here.
      System.out.println("x and y: " + x + " " + y);
      x = y * 2;
    }
    // y = 100; // Error! y not known here

    // x is still known here.
    System.out.println("x is " + x);
  }
}
```

正如注释中说明的那样，在方法main()的开始定义了变量x，因此它对于main()中的所有的随后的代码都是可见的。在if程序块中定义了变量y。因为一个块定义一个作用域，y仅仅对在它的块以内的其他代码可见。这就是在它的块之外的程序行y=100;被注释掉的原因。如果你将该行前面的注释符号去掉，编译程序时就会出现错误，因为变量y在它的程序块之外是不可见的。在if程序块中可以使用变量x，因为块（即一个嵌套作用域）中的程序可以访问被其包围作用域中定义的变量。

变量可以在程序块内的任何地方被声明，但是只有在他们被声明以后才是合法有效的。因此，如果你在一个方法的开始定义了一个变量，那么它对于在该方法以内的所有程序都是可用的。反之，如果你在一个程序块的末尾声明了一个变量，它就没有任何用处，因为没有程序会访问它。例如，下面这个程序段就是无效的，因为变量count在它被定义以前是不能被使用的。

```
// This fragment is wrong!
count = 100; // oops! cannot use count before it is declared!
int count;
```

另一个需要记住的重要之处是：变量在其作用域内被创建，离开其作用域时被撤消。

这意味着一个变量一旦离开它的作用域，将不再保存它的值了。因此，在一个方法内定义的变量在几次调用该方法之间将不再保存它们的值。同样，在块内定义的变量在离开该块时也将丢弃它的值。因此，一个变量的生存期就被限定在它的作用域中。

如果一个声明定义包括一个初始化，那么每次进入声明它的程序块时，该变量都要被重新初始化。例如，考虑这个程序：

```
// Demonstrate lifetime of a variable.
class LifeTime {
  public static void main(String args[]) {
    int x;

    for(x = 0; x < 3; x++) {
      int y = -1; // y is initialized each time block is entered
       System.out.println("y is: " + y); // this always prints -1
       y = 100;
       System.out.println("y is now: " + y);
    }
  }
}
```

该程序运行的输出如下：

```
y is: -1
y is now: 100
y is: -1
y is now: 100
y is: -1
y is now: 100
```

可以看到，每次进入内部的for循环，y都要被重新初始化为-1。即使它随后被赋值为100，该值还是被丢弃了。

最后一点：尽管程序块能被嵌套，你不能将内部作用域声明的变量与其外部作用域声明的变量重名。在这一点上，Java不同于C和C++。下面的例子企图为两个独立的变量起同样的名字。在Java中，这是不合法的。但在C/C++中，它将是合法的，而且2个变量bar将是独立的。

```
// This program will not compile
class ScopeErr {
  public static void main(String args[]) {
    int bar = 1;
    { // creates a new scope
      int bar = 2; // Compile-time error - bar already defined!
    }
  }
}
```

3.9　类型转换与强制类型转换

如果你以前有编程经验，那么你已经知道把一种类型的值赋给另外类型的一个变量是

相当常见的。如果这2种类型是兼容的，那么Java将自动地进行转换。例如，把int类型的值赋给long类型的变量，总是可行的。然而，不是所有的类型都是兼容的，因此，不是所有的类型转换都是可以隐式实现的。例如，没有将double型转换为byte型的定义。幸好，获得不兼容的类型之间的转换仍然是可能的。要达到这个目的，你必须使用一个强制类型转换，它能完成两个不兼容的类型之间的显式变换。让我们看看自动类型转换和强制类型转换。

3.9.1 Java的自动转换

如果下列2个条件都能满足，那么将一种类型的数据赋给另外一种类型变量时，将执行自动类型转换（automatic type conversion）：

- 这2种类型是兼容的。
- 目的类型数的范围比来源类型的大。

当以上2个条件都满足时，拓宽转换（widening conversion）发生。例如，int型的范围比所有byte型的合法范围大，因此不要求显式强制类型转换语句。

对于拓宽转换，数字类型，包括整数（integer）和浮点（floating-point）类型都是彼此兼容的，但是，数字类型和字符类型（char）或布尔类型（bollean）是不兼容的。字符类型（char）和布尔类型（bollean）也是互相不兼容的。

3.9.2 不兼容类型的强制转换

尽管自动类型转换是很有帮助的，但并不能满足所有的编程需要。例如，如果你需要将int型的值赋给一个byte型的变量，你将怎么办？这种转换不会自动进行，因为byte型的变化范围比int型的要小。这种转换有时称为“缩小转换”（），因为你肯定要将源数据类型的值变小才能适合目标数据类型。

为了完成两种不兼容类型之间的转换，你就必须进行强制类型转换。所谓强制类型转换只不过是一种显式的类型变换。它的通用格式如下：

```
(target-type)value
```

其中，目标类型（target-type）指定了要将指定值转换成的类型。例如，下面的程序段将int型强制转换成byte型。如果整数的值超出了byte型的取值范围，它的值将会因为对byte型值域取模（整数除以byte得到的余数）而减少。

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

当把浮点值赋给整数类型时一种不同的类型转换发生了：截断（truncation）。你知道整数没有小数部分。这样，当把浮点值赋给整数类型时，它的小数部分会被舍去。例如，如果将值1.23赋给一个整数，其结果值只是1，0.23被丢弃了。当然，如果浮点值太大而不能适合目标整数类型，那么它的值将会因为对目标类型值域取模而减少。

下面的程序说明了强制类型转换：

```
// Demonstrate casts.
class Conversion {
  public static void main(String args[]) {
    byte b;
    int i = 257;
    double d = 323.142;

    System.out.println("\nConversion of int to byte.");
    b = (byte) i;
    System.out.println("i and b " + i + " " + b);

    System.out.println("\nConversion of double to int.");
    i = (int) d;
    System.out.println("d and i " + d + " " + i);

    System.out.println("\nConversion of double to byte.");
    b = (byte) d;
    System.out.println("d and b " + d + " " + b);
  }
}
```

该程序的输出如下：

```
Conversion of int to byte.
i and b 257 1

Conversion of double to int.
d and i 323.142 323

Conversion of double to byte.
d and b 323.142 67
```

让我们看看每一个类型转换。当值257被强制转换为byte变量时，其结果是257除以256（256是byte类型的变化范围）的余数1。当把变量d转换为int型，它的小数部分被舍弃了。当把变量d转换为byte型，它的小数部分被舍弃了，而且它的值减少为256的模，即67。

3.10　表达式中类型的自动提升

除了赋值，还有另外一种类型变换：在表达式中。想要知道原因，往下看。在表达式中，对中间值的精确要求有时超过任何一个操作数的范围。例如，考虑下面的表达式：

```
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c;
```

中间项结果a*b很容易超过它的任何一个byte型操作数的范围。为处理这种问题，当分析表达式时，Java自动提升各个byte型或short型的操作数到int型。这意味着子表达式a*b使用整数而不是字节型来执行。这样，尽管变量a和b都被指定为byte型，50*40中间表达式的结果2000是合法的。

自动类型提升有好处，但它也会引起令人疑惑的编译错误。例如，这个看起来正确的程序却会引起问题：

```
byte b = 50;
b = b * 2; // Error! Cannot assign an int to a byte!
```

该程序试图将一个完全合法的byte型的值50*2再存储给一个byte型的变量。但是当表达式求值的时候，操作数被自动地提升为int型，计算结果也被提升为int型。这样，表达式的结果现在是int型，不强制转换它就不能被赋为byte型。确实如此，在这个特别的情况下，被赋的值将仍然适合目标类型。

在你理解溢出的后果的情况下，你应该使用一个显式的强制类型转换，例如：

```
byte b = 50;
b = (byte)(b * 2);
```

它产生出正确的值100 。

3.10.1 类型提升的约定

除了将byte型和shorts型提升到int型以外，Java定义了若干适用于表达式的类型提升规则（type promotion rules）。首先，如刚才描述的，所有的byte型和short型的值被提升到 int型。其次，如果一个操作数是long型，整个表达式将被提升到long型；如果一个操作数是float型，整个表达式将被提升到float型；如果有一个操作数是double型，计算结果就是double型。

下面的程序表明：在表达式中的每个值是如何被提升以匹配各自二进制运算符的第二个参数：

```
class Promote {
  public static void main(String args[]) {
    byte b = 42;
    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);
  }
}
```

让我们进一步看看发生在下列程序行的类型提升：

```
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，该子表达式的结果当然是float类型。接下来，在子表达式i/c，中，变量c被提升为int类型，该子表达式的结果当然是int类型。然后，子表达式d*s中的变量s被提升为double类型，该子表达式的结果当然也是double类型。最后，考虑三个中间值，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类型减去提升为double类型的double类型，该表达式的最后结果是double类

型。

3.11　数　组

数组（array）是相同类型变量的集合，可以使用共同的名字引用它。数组可被定义为任何类型，可以是一维或多维。数组中的一个特别要素是通过下标来访问它。数组提供了一种将有联系的信息分组的便利方法。

注意：如果你熟悉C/C++，请注意，Java数组的工作原理与它们不同。

3.11.1　一维数组

一维数组（one-dimensional array）实质上是相同类型变量列表。要创建一个数组，你必须首先定义数组变量所需的类型。通用的一维数组的声明格式是：

```
type var-name[ ];
```

其中，type定义了数组的基本类型。基本类型决定了组成数组的每一个基本元素的数据类型。这样，数组的基本类型决定了数组存储的数据类型。例如，下面的例子定义了数据类型为int，名为month_days的数组。

```
int month_days[];
```

尽管该例子定义了month_days是一个数组变量的事实，但实际上没有数组变量存在。事实上，month_days的值被设置为空，它代表一个数组没有值。为了使数组month_days成为实际的、物理上存在的整型数组，你必须用运算符new来为其分配地址并且把它赋给month_days。运算符new是专门用来分配内存的运算符。

你将在后面章节中更进一步了解运算符new，但是你现在需要使用它来为数组分配内存。当运算符new被应用到一维数组时，它的一般形式如下：

```
array-var = new type[size];
```

其中，type指定被分配的数据类型，size指定数组中变量的个数，array-var 是被链接到数组的数组变量。也就是，使用运算符new来分配数组，你必须指定数组元素的类型和数组元素的个数。用运算符new分配数组后，数组中的元素将会被自动初始化为零。下面的例子分配了一个12个整型元素的数组并把它们和数组month_days 链接起来。

```
month_days = new int[12];
```

通过这个语句的执行，数组month_days将会指向12个整数，而且数组中的所有元素将被初始化为零。

让我们回顾一下上面的过程：获得一个数组需要2步。第一步，你必须定义变量所需的类型。第二步，你必须使用运算符new来为数组所要存储的数据分配内存，并把它们分配给数组变量。这样Java中的数组被动态地分配。如果动态分配的概念对你陌生，别担心，它将在本书的后面详细讨论。

一旦你分配了一个数组，你可以在方括号内指定它的下标来访问数组中特定的元素。

所有的数组下标从零开始。例如，下面的语句将值28赋给数组month_days 的第二个元素。

```
month_days[1] = 28;
```

下面的程序显示存储在下标为3的数组元素中的值。

```
System.out.println ( month_days [ 3 ]);
```

综上所述，下面程序定义的数组存储了每月的天数。

```
// Demonstrate a one-dimensional array.
class Array {
  public static void main(String args[]) {
    int month_days[];
    month_days = new int[12];
    month_days[0] = 31;
    month_days[1] = 28;
    month_days[2] = 31;
    month_days[3] = 30;
    month_days[4] = 31;
    month_days[5] = 30;
    month_days[6] = 31;
    month_days[7] = 31;
    month_days[8] = 30;
    month_days[9] = 31;
    month_days[10] = 30;
    month_days[11] = 31;
    System.out.println("April has " + month_days[3] + " days.");
  }
}
```

当你运行这个程序时，它打印出4月份的天数。如前面提到的，Java数组下标从零开始，因此4月份的天数数组元素为month_days[3]或30。

将对数组变量的声明和对数组本身的分配结合起来是可以的，如下所示：

```
int month_days[] = new int[12];
```

这将是你通常看见的编写Java程序的专业做法。

数组可以在声明时被初始化。这个过程和简单类型初始化的过程一样。数组的初始化（array initializer）就是包括在花括号之内用逗号分开的表达式的列表。逗号分开了数组元素的值。Java会自动地分配一个足够大的空间来保存你指定的初始化元素的个数，而不必使用运算符new。例如，为了存储每月中的天数，下面的程序定义了一个初始化的整数数组：

```
// An improved version of the previous program.
class AutoArray {
  public static void main(String args[]) {
    int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31,
                         30, 31 };
    System.out.println("April has " + month_days[3] + " days.");
  }
}
```

当你运行这个程序时，你会看到它和前一个程序产生的输出一样。

Java严格地检查以保证你不会意外地去存储或引用在数组范围以外的值。Java的运行系统会检查以确保所有的数组下标都在正确的范围以内（在这方面，Java与C/C++从根本上不同，C/C++不提供运行边界检查）。例如，运行系统将检查数组month_days的每个下标的值以保证它包括在0和11之间。如果你企图访问数组边界以外（负数或比数组边界大）的元素，你将引起运行错误。

下面的例子运用一维数组来计算一组数字的平均数。

```
// Average an array of values.
class Average {
  public static void main(String args[]) {
    double nums[] = {10.1, 11.2, 12.3, 13.4, 14.5};
    double result = 0;
    int i;

    for(i=0; i<5; i++)
      result = result + nums[i];

    System.out.println("Average is " + result / 5);
  }
}
```

3.11.2　多维数组

在Java中，多维数组（multidimensional arrays）实际上是数组的数组。你可能期望，这些数组形式上和行动上和一般的多维数组一样。然而，你将看到，有一些微妙的差别。定义多维数组变量要将每个维数放在它们各自的方括号中。例如，下面语句定义了一个名为twoD的二维数组变量。

```
int twoD[][] = new int[4][5];
```

该语句分配了一个4行5列的数组并把它分配给数组twoD。实际上这个矩阵表示了int类型的数组的数组被实现的过程。概念上，这个数组可以用图3-1来表示。

下列程序从左到右，从上到下为数组的每个元素赋值，然后显示数组的值：

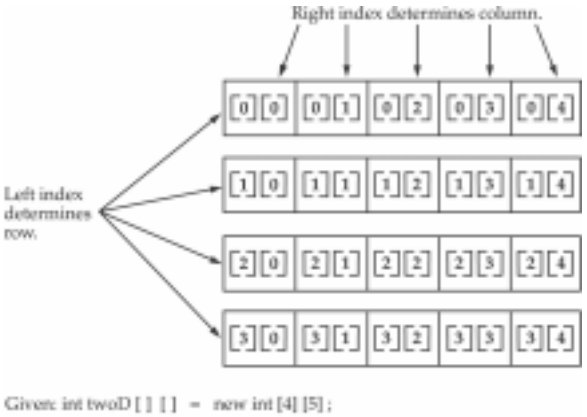


图 3.1　二维数组（4 行 5 列）的概念性表示

```
// Demonstrate a two-dimensional array.
class TwoDArray {
  public static void main(String args[]) {
    int twoD[][]= new int[4][5];
    int i, j, k = 0;

    for(i=0; i<4; i++)
      for(j=0; j<5; j++) {
        twoD[i][j] = k;
        k++;
    }

    for(i=0; i<4; i++) {
      for(j=0; j<5; j++)
        System.out.print(twoD[i][j] + " ");
      System.out.println();
    }
  }
}
```

程序运行的结果如下：

```
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
```

当你给多维数组分配内存时，你只需指定第一个（最左边）维数的内存即可。你可以单独地给余下的维数分配内存。例如，下面的程序在数组twoD被定义时给它的第一个维数分配内存，对第二维则是手工分配地址。

```
int twoD[][] = new int[4][];
twoD[0] = new int[5];
twoD[1] = new int[5];
twoD[2] = new int[5];
twoD[3] = new int[5];
```

尽管在这种情形下单独地给第二维分配内存没有什么优点，但在其他情形下就不同了。例如，当你手工分配内存时，你不需要给每个维数相同数量的元素分配内存。如前面所说，既然多维数组实际上是数组的数组，每个数组的维数在你的控制之下。例如，下列程序定义了一个二维数组，它的第二维的大小是不相等的。

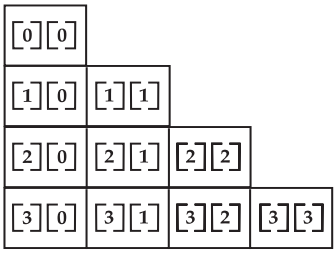
```
// Manually allocate differing size second dimensions.
class TwoDAgain {
  public static void main(String args[]) {
    int twoD[][] = new int[4][];
    twoD[0] = new int[1];
    twoD[1] = new int[2];
    twoD[2] = new int[3];
    twoD[3] = new int[4];

    int i, j, k = 0;
```

```
    for(i=0; i<4; i++)
      for(j=0; j<i+1; j++) {
        twoD[i][j] = k;
        k++;
      }

    for(i=0; i<4; i++) {
      for(j=0; j<i+1; j++)
        System.out.print(twoD[i][j] + " ");
      System.out.println();
    }
  }
}
```

该程序产生的输出如下：

```
0
1 2
3 4 5
6 7 8 9
```

该程序定义的数组可以表示如下：



对于大多数应用程序，我们不推荐使用不规则多维数组，因为它们的运行与人们期望的相反。但是，不规则多维数组在某些情况下使用效率较高。例如，如果你需要一个很大的二维数组，而它仅仅被稀疏地占用（即其中一维的元素不是全被使用），这时不规则数组可能是一个完美的解决方案。

初始化多维数组是可能的。初始化多维数组只不过是把每一维的初始化列表用它自己的大括号括起来即可。下面的程序产生一个矩阵，该矩阵的每个元素包括数组下标的行和列的积。同时注意在数组的初始化中你可以像用字面量一样用表达式。

```
// Initialize a two-dimensional array.
class Matrix {
  public static void main(String args[]) {
    double m[][] = {
      { 0*0, 1*0, 2*0, 3*0 },
      { 0*1, 1*1, 2*1, 3*1 },
      { 0*2, 1*2, 2*2, 3*2 },
      { 0*3, 1*3, 2*3, 3*3 }
    };
    int i, j;
```

```
    for(i=0; i<4; i++) {
      for(j=0; j<4; j++)
        System.out.print(m[i][j] + " ");
      System.out.println();
    }
  }
}
```

当你运行这个程序时，你将得到下面的输出：

```
0.0  0.0  0.0  0.0
0.0  1.0  2.0  3.0
0.0  2.0  4.0  6.0
0.0  3.0  6.0  9.0
```

正如你看到，数组中的每一行就像初始化表指定的那样被初始化。

让我们再看一个使用多维数组的例子。下面的程序首先产生一个3×4×5的3维数组，然后装入用它的下标之积生成的每个元素，最后显示了该数组。

```
// Demonstrate a three-dimensional array.
class threeDMatrix {
  public static void main(String args[]) {
    int threeD[][][] = new int[3][4][5];
    int i, j, k;

    for(i=0; i<3; i++)
      for(j=0; j<4; j++)
        for(k=0; k<5; k++)
          threeD[i][j][k] = i * j * k;

      for(i=0; i<3; i++) {
        for(j=0; j<4; j++) {
          for(k=0; k<5; k++)
            System.out.print(threeD[i][j][k] + " ");
          System.out.println();
        }
        System.out.println();
    }
  }
}
```

该程序的输出如下：

```
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0

0 0 0 0 0
0 1 2 3 4
0 2 4 6 8
0 3 6 9 12

0 0 0 0 0
0 2 4 6 8
```

```
0 4 8 12 16
0 6 12 18 24
```

3.11.3　另一种数组声明语法

声明数组还有第二种格式：

```
type[ ] var-name;
```

这里，方括号紧跟在类型标识符type的后面，而不是跟在数组变量名的后面。例如，下面的两个定义是等价的：

```
int a1[] = new int[3];
int[] a2 = new int[3];
```

下面的两个定义也是等价的：

```
char twod1[][] = new char[3][4];
char[][] twod2 = new char[3][4];
```

包含这种数组声明格式主要是为了方便。

3.12　字符串的简单介绍

你可能注意到了，在前面关于数据类型和数组的讨论中没有提到字符串或字符串数据类型。这不是因为Java不支持这样一种类型，它支持。只是因为Java的字符串类型，叫做字符串（String），它不是一种简单的类型，它也不是简单的字符数组（在C/C++中是）。字符串（String）在Java中被定义为对象，要完全理解它需要理解几个和对象相关的特征。因此，有关字符串（String）的讨论被放到本书的后面，在对象被描述后再讲字符串（String）。但是，为了在例子程序中使用简单的字符串，下面简单的按顺序介绍。

字符串（String）类型被用来声明字符串变量。你也可以定义字符串数组。一个被引号引起来的字符串字面量可以被分配给字符串变量。一个字符串类型的变量可被分配给另一个字符串类型的变量。你可以像用方法println()的参数一样用字符串（String）类型。例如，考虑下面的语句：

```
String str = "this is a test";
System.out.println(str);
```

这里，str是字符串（String）类型的一个对象，它被分配给字符串“this is a test”，该字符串被println( )语句显示。

读者你将会在后面看到，字符串对象有许多特别的特征和属性，这使得它们功能非常强大而且易用。然而，在后面的几章中，你只能用它们最简单的形式。

3.13　C/C++程序员请注意指针的用法

如果你是一个经验丰富的C/C++程序员，那么你知道这些语言提供对指针的支持。然

而，在本章中没有提到指针。这样做的道理很简单：Java不支持或不允许指针（或者更恰当地说，Java不支持程序员来访问或修改指针）。Java不允许指针，是因为这样做将允许Java applet（小应用程序）突破Java运行环境和主机之间的防火墙（要知道，指针能放在内存的任何地址，即使是Java运行系统之外的地址）。既然C/C++广泛的使用指针，你可能认为损失对指针的使用是Java的一个很重要的不利条件。然而，事实并非如此。Java以这种方式设计，只要你在Java的执行环境范围内，你决不需要使用指针如何进行处理，指针也不会有任何好处。至于将C/C++上的程序转化到Java中来，包括指针，参看第28章。

第4章　运 算 符

Java提供了丰富的运算符环境。Java有4大类运算符：算术运算、位运算、关系运算和逻辑运算。Java还定义了一些附加的运算符用于处理特殊情况。本章将描述Java所有的运算符，而比较运算符instanceof将在第12章讨论。

注意：如果你对C/C++熟悉，你将会高兴，因为Java的绝大多数运算符和C/C++中的用法一样。但有一些微妙的差别，所以提醒你要仔细阅读。

4.1　算术运算符

算术运算符用在数学表达式中，其用法和功能与代数学（或其他计算机语言）中一样，Java定义了下列算术运算符（见表4-1）：

表 4.1　算术运算符及其含义

运算符	含义
+	加法
–	减法（一元减号）
*	乘法
/	除法
%	模运算
++	递增运算
+=	加法赋值
–=	减法赋值
*=	乘法赋值
/=	除法赋值
%=	模运算赋值
--	递减运算

算术运算符的运算数必须是数字类型。算术运算符不能用在布尔类型上，但是可以用在char类型上，因为实质上在Java中，char类型是int类型的一个子集。

4.1.1　基本算术运算符

基本算术运算符——加、减、乘、除可以对所有的数字类型操作。减运算也用作表示单个操作数的负号。记住对整数进行“/”除法运算时，所有的余数都要被舍去。

下面这个简单例子示范了算术运算符，也说明了浮点型除法和整型除法之间的差别。

```
// Demonstrate the basic arithmetic operators.
class BasicMath {
  public static void main(String args[]) {
    // arithmetic using integers
    System.out.println("Integer Arithmetic");
    int a = 1 + 1;
    int b = a * 3;
    int c = b / 4;
    int d = c - a;
    int e = -d;
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    System.out.println("c = " + c);
    System.out.println("d = " + d);
    System.out.println("e = " + e);

    // arithmetic using doubles
    System.out.println("\nFloating Point Arithmetic");
    double da = 1 + 1;
    double db = da * 3;
    double dc = db / 4;
    double dd = dc - a;
    double de = -dd;
    System.out.println("da = " + da);
    System.out.println("db = " + db);
    System.out.println("dc = " + dc);
    System.out.println("dd = " + dd);
    System.out.println("de = " + de);
  }
}
```

当你运行这个程序，你会看到输出如下：

```
Integer Arithmetic
a = 2
b = 6
c = 1
d = -1
e = 1

Floating Point Arithmetic
da = 2.0
db = 6.0
dc = 1.5
dd = -0.5
de = 0.5
```

4.1.2 模运算符

模运算符%，其运算结果是整数除法的余数。它能像整数类型一样被用于浮点类型（这不同于C/C++，在C/C++中模运算符%仅仅能用于整数类型）。下面的示例程序说明了模运算符%的用法：

```
// Demonstrate the % operator.
```

```
class Modulus {
  public static void main(String args[]) {
    int x = 42;
    double y = 42.25;

    System.out.println("x mod 10 = " + x % 10);
    System.out.println("y mod 10 = " + y % 10);
  }
}
```

当你运行这个程序，你会看到输出如下：

```
x mod 10 = 2
y mod 10 = 2.25
```

4.1.3 算术赋值运算符

Java提供特殊的算术赋值运算符，该运算符可用来将算术运算符与赋值结合起来。你可能知道，像下列这样的语句在编程中是很常见的：

```
a = a +4;
```

在Java中，你可将该语句重写如下：

```
a += 4;
```

该语句使用“+=”进行赋值操作。上面两行语句完成的功能是一样的：使变量a的值增加4 。下面是另一个例子：

```
a = a % 2;
```

该语句可简写为：

```
a %= 2;
```

在本例中，%=算术运算符的结果是a/2的余数，并把结果重新赋给变量a。

这种简写形式对于Java的二元（即需要两个操作数的）运算符都适用。其语句格式为：

```
var= var op expression;
```

可以被重写为：

```
var op= expression;
```

这种赋值运算符有两个好处。第一，它们比标准的等式要紧凑。第二，它们有助于提高Java的运行效率。由于这些原因，在Java的专业程序中，你经常会看见这些简写的赋值运算符。

下面的例子显示了几个赋值运算符的作用：

```
// Demonstrate several assignment operators.
class OpEquals {
  public static void main(String args[]) {
    int a = 1;
    int b = 2;
    int c = 3;
```

```
        a += 5;
        b *= 4;
        c += a * b;
        c %= 6;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
    }
}
```

该程序的输出如下：

```
a = 6
b = 8
c = 3
```

4.1.4 递增和递减运算

在第2章中曾经介绍过，“++”和“--”是Java的递增和递减运算符。下面将对它们做详细讨论。它们具有一些特殊的性能，这使它们变得非常有趣。我们先来复习一下递增和递减运算符的操作。

递增运算符对其运算数加1，递减运算符对其运算数减1。因此：

```
x = x + 1;
```

运用递增运算符可以重写为：

```
x++;
```

同样，语句：

```
x = x - 1;
```

与下面一句相同：

```
x--;
```

在前面的例子中，递增或递减运算符采用前缀（prefix）或后缀（postfix）格式都是相同的。但是，当递增或递减运算符作为一个较大的表达式的一部分，就会有重要的不同。如果递增或递减运算符放在其运算数前面，Java就会在获得该运算数的值之前执行相应的操作，并将其用于表达式的其他部分。如果运算符放在其运算数后面，Java就会先获得该操作数的值再执行递增或递减运算。例如：

```
x = 42 ;
y =++x ;
```

在这个例子中，y将被赋值为43，因为在将x的值赋给y以前，要先执行递增运算。这样，语句行y =++x ;和下面两句是等价的：

```
x = x + 1;
y = x;
```

但是，当写成这样时：

```
x = 42;
y = x++;
```

在执行递增运算以前，已将x的值赋给了y，因此y的值还是42 。当然，在这两个例子中，x都被赋值为43。在本例中，程序行y =x++；与下面两个语句等价：

```
y = x;
x = x + 1;
```

下面的程序说明了递增运算符的使用：

```
// Demonstrate ++.
class IncDec {
  public static void main(String args[]) {
    int a = 1;
    int b = 2;
    int c;
    int d;
    c = ++b;
    d = a++;
    c++;
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    System.out.println("c = " + c);
    System.out.println("d = " + d);
  }
}
```

该程序的输出如下：

```
a = 2
b = 3
c = 4
d = 1
```

4.2　位 运 算 符

Java定义的位运算（bitwise operators）直接对整数类型的位进行操作，这些整数类型包括long，int，short，char，and byte。表4-2列出了位运算：

表 4.2　位运算符及其结果

运算符	结果
~	按位非（NOT）（一元运算）
&	按位与（AND）
|	按位或（OR）
^	按位异或（XOR）
>>	右移
>>>	右移，左边空出的位以0填充

续表

运算符	结果
<<	左移
&=	按位与赋值
|=	按位或赋值
^=	按位异或赋值
>>=	右移赋值
>>>=	右移赋值，左边空出的位以0填充
<<=	左移赋值

既然位运算符在整数范围内对位操作，因此理解这样的操作会对一个值产生什么效果是重要的。具体地说，知道Java是如何存储整数值并且如何表示负数的是有用的。因此，在继续讨论之前，让我们简短概述一下这两个话题。

所有的整数类型以二进制数字位的变化及其宽度来表示。例如，byte型值42的二进制代码是00101010，其中每个位置在此代表2的次方，在最右边的位以2^0开始。向左下一个位置将是2^1，或2，依次向左是2^2，或4，然后是8，16，32等等，依此类推。因此42在其位置1，3，5的值为1（从右边以0开始数）；这样42是$2^1+2^3+2^5$的和，也即是2+8+32 。

所有的整数类型（除了char类型之外）都是有符号的整数。这意味着他们既能表示正数，又能表示负数。Java使用大家知道的2的补码（two's complement）这种编码来表示负数，也就是通过将与其对应的正数的二进制代码取反（即将1变成0，将0变成1），然后对其结果加1。例如，-42就是通过将42的二进制代码的各个位取反，即对00101010取反得到11010101，然后再加1，得到11010110，即-42 。要对一个负数解码，首先对其所有的位取反，然后加1。例如-42，或11010110取反后为00101001，或41，然后加1，这样就得到了42。

如果考虑到零的交叉（zero crossing）问题，你就容易理解Java（以及其他绝大多数语言）这样用2的补码的原因。假定byte类型的值零用00000000代表。它的补码是仅仅将它的每一位取反，即生成11111111，它代表负零。但问题是负零在整数数学中是无效的。为了解决负零的问题，在使用2的补码代表负数的值时，对其值加1。即负零11111111加1后为100000000。但这样使1位太靠左而不适合返回到byte类型的值，因此人们规定，-0和0的表示方法一样，-1的解码为11111111。尽管我们在这个例子使用了byte类型的值，但同样的基本的原则也适用于所有Java 的整数类型。

因为Java使用2的补码来存储负数，并且因为Java中的所有整数都是有符号的，这样应用位运算符可以容易地达到意想不到的结果。例如，不管你如何打算，Java用高位来代表负数。为避免这个讨厌的意外，请记住不管高位的顺序如何，它决定一个整数的符号。

4.2.1 位逻辑运算符

位逻辑运算符有“与”（AND）、“或”（OR）、“异或（XOR）”、“非（NOT）”，分别用“&”、“|”、“^”、“~”表示，4-3表显示了每个位逻辑运算的结果。在继续讨论之前，请记住位运算符应用于每个运算数内的每个单独的位。

表 4-3　位逻辑运算符的结果

A	B	A | B	A & B	A ^ B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

按位非（NOT）

按位非也叫做补，一元运算符NOT“~”是对其运算数的每一位取反。例如，数字42，它的二进制代码为：

```
00101010
```

经过按位非运算成为

```
11010101
```

按位与（AND）

按位与运算符“&”，如果两个运算数都是1，则结果为1。其他情况下，结果均为零。看下面的例子：

```
  00101010      42
 &00001111      15
--------------
  00001010      10
```

按位或（OR）

按位或运算符“|”，任何一个运算数为1，则结果为1。如下面的例子所示：

```
  00101010      42
 | 00001111     15
--------------
  00101111      47
```

按位异或（XOR）

按位异或运算符“^”，只有在两个比较的位不同时其结果是 1。否则，结果是零。下面的例子显示了“^”运算符的效果。这个例子也表明了XOR运算符的一个有用的属性。注意第二个运算数有数字1的位，42对应二进制代码的对应位是如何被转换的。第二个运算数有数字0的位，第一个运算数对应位的数字不变。当对某些类型进行位运算时，你将会看到这个属性的用处。

```
  00101010      42
 ^ 00001111     15
-------------
  00100101      37
```

位逻辑运算符的应用

下面的例子说明了位逻辑运算符：

```
// Demonstrate the bitwise logical operators.
class BitLogic {
  public static void main(String args[]) {
    String binary[] = {
      "0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111",
      "1000", "1001", "1010", "1011", "1100", "1101", "1110", "1111"
    };
    int a = 3; // 0 + 2 + 1 or 0011 in binary
    int b = 6; // 4 + 2 + 0 or 0110 in binary
    int c = a | b;
    int d = a & b;
    int e = a ^ b;
    int f = (~a & b) | (a & ~b);
    int g = ~a & 0x0f;

    System.out.println(" a = " + binary[a]);
    System.out.println(" b = " + binary[b]);
    System.out.println(" a|b = " + binary[c]);
    System.out.println(" a&b = " + binary[d]);
    System.out.println(" a^b = " + binary[e]);
    System.out.println("~a&b|a&~b = " + binary[f]);
    System.out.println(" ~a = " + binary[g]);
  }
}
```

在本例中，变量a与b对应位的组合代表了二进制数所有的 4 种组合模式：0-0，0-1，1-0，和1-1。“|”运算符和“&”运算符分别对变量a与b各个对应位的运算得到了变量c和变量d的值。对变量e和f的赋值说明了“^”运算符的功能。字符串数组binary代表了0到15对应的二进制的值。在本例中，数组各元素的排列顺序显示了变量对应值的二进制代码。数组之所以这样构造是因为变量的值n对应的二进制代码可以被正确的存储在数组对应元素binary[n]中。例如变量a的值为3，则它的二进制代码对应地存储在数组元素binary[3]中。~a的值与数字0x0f （对应二进制为0000 1111）进行按位与运算的目的是减小~a的值，保证变量g的结果小于16。因此该程序的运行结果可以用数组binary对应的元素来表示。该程序的输出如下：

```
          a     = 0011
          b     = 0110
        a|b     = 0111
        a&b     = 0010
        a^b     = 0101
 ~a&b|a&~b      = 0101
         ~a     = 1100
```

4.2.2 左移运算符

左移运算符<<使指定值的所有位都左移规定的次数。它的通用格式如下所示：

```
value << num
```

这里，num指定要移位值value移动的位数。也就是，左移运算符<<使指定值的所有位都左移num位。每左移一个位，高阶位都被移出（并且丢弃），并用0填充右边。这意味着当左移的运算数是int类型时，每移动1位它的第31位就要被移出并且丢弃；当左移的运算数是long类型时，每移动1位它的第63位就要被移出并且丢弃。

在对byte和short类型的值进行移位运算时，你必须小心。因为你知道Java在对表达式求值时，将自动把这些类型扩大为 int型，而且，表达式的值也是int型 。对byte和short类型的值进行移位运算的结果是int型，而且如果左移不超过31位，原来对应各位的值也不会丢弃。但是，如果你对一个负的byte或者short类型的值进行移位运算，它被扩大为int型后，它的符号也被扩展。这样，整数值结果的高位就会被1填充。因此，为了得到正确的结果，你就要舍弃得到结果的高位。这样做的最简单办法是将结果转换为byte型。下面的程序说明了这一点：

```
// Left shifting a byte value.
class ByteShift {
  public static void main(String args[]) {
    byte a = 64, b;
    int i;

    i = a << 2;
    b = (byte) (a << 2);

    System.out.println("Original value of a: " + a);
    System.out.println("i and b: " + i + " " + b);
  }
}
```

该程序产生的输出下所示：

```
Original value of a: 64
i and b: 256 0
```

因变量a在赋值表达式中，故被扩大为int型，64（0100 0000）被左移两次生成值256（10000 0000）被赋给变量i。然而，经过左移后，变量b中惟一的1被移出，低位全部成了0，因此b的值也变成了0。

既然每次左移都可以使原来的操作数翻倍，程序员们经常使用这个办法来进行快速的2的乘法。但是你要小心，如果你将1移进高阶位（31或63位），那么该值将变为负值。下面的程序说明了这一点：

```
// Left shifting as a quick way to multiply by 2.
class MultByTwo {
  public static void main(String args[]) {
    int i;
    int num = 0xFFFFFFE;

    for(i=0; i<4; i++) {
      num = num << 1;
      System.out.println(num);
    }
  }
```

```
}
```

该程序的输出如下所示：

```
536870908
1073741816
2147483632
-32
```

初值经过仔细选择，以便在左移 4 位后，它会产生-32。正如你看到的，当1被移进31位时，数字被解释为负值。

4.2.3 右移运算符

右移运算符>>使指定值的所有位都右移规定的次数。它的通用格式如下所示：

```
value >> num
```

这里，num指定要移位值value移动的位数。也就是，右移运算符>>使指定值的所有位都右移num位。

下面的程序片段将值32右移2次，将结果8赋给变量a:

```
int a = 32;
a = a >> 2; // a now contains 8
```

当值中的某些位被“移出”时，这些位的值将丢弃。例如，下面的程序片段将35右移2次，它的2个低位被移出丢弃，也将结果8赋给变量a:

```
int a = 35;
a = a >> 2; // a still contains 8
```

用二进制表示该过程可以更清楚地看到程序的运行过程：

```
00100011  35
>> 2
00001000  8
```

将值每右移一次，就相当于将该值除以2并且舍弃了余数。你可以利用这个特点将一个整数进行快速的2的除法。当然，你一定要确保你不会将该数原有的任何一位移出。

右移时，被移走的最高位（最左边的位）由原来最高位的数字补充。例如，如果要移走的值为负数，每一次右移都在左边补1，如果要移走的值为正数，每一次右移都在左边补0，这叫做符号位扩展（保留符号位）（sign extension），在进行右移操作时用来保持负数的符号。例如，–8 >> 1 是–4，用二进制表示如下：

```
11111000  -8
>>1
11111100  -4
```

一个要注意的有趣问题是，由于符号位扩展（保留符号位）每次都会在高位补1，因此-1右移的结果总是–1。

有时你不希望在右移时保留符号。例如，下面的例子将一个byte型的值转换为用十六

进制表示。注意右移后的值与0x0f进行按位与运算，这样可以舍弃任何的符号位扩展，以便得到的值可以作为定义数组的下标，从而得到对应数组元素代表的十六进制字符。

```
// Masking sign extension.
class HexByte {
  static public void main(String args[]) {
    char hex[] = {
      '0', '1', '2', '3', '4', '5', '6', '7',
      '8', '9', 'a', 'b', 'c', 'd', 'e', 'f''
    };
    byte b = (byte) 0xf1;

    System.out.println("b = 0x" + hex[(b >> 4) & 0x0f] + hex[b & 0x0f]);
  }
}
```

该程序的输出如下：

```
b = 0xf1
```

4.2.4 无符号右移

正如上面刚刚看到的，每一次右移，>>运算符总是自动地用它的先前最高位的内容补它的最高位。这样做保留了原值的符号。但有时这并不是我们想要的。例如，如果你进行移位操作的运算数不是数字值，你就不希望进行符号位扩展（保留符号位）。当你处理像素值或图形时，这种情况是相当普遍的。在这种情况下，不管运算数的初值是什么，你希望移位后总是在高位（最左边）补0。这就是人们所说的无符号移动（unsigned shift）。这时你可以使用Java的无符号右移运算符>>>，它总是在左边补0。

下面的程序段说明了无符号右移运算符>>>。在本例中，变量a被赋值为-1，用二进制表示就是32位全是1。这个值然后被无符号右移24位，当然它忽略了符号位扩展，在它的左边总是补0。这样得到的值255被赋给变量a。

```
int a = -1;
a = a >>> 24;
```

下面用二进制形式进一步说明该操作：

```
11111111  11111111  11111111  11111111      int型- 1的二进制代码
>>> 24                                      无符号右移24位
00000000  00000000  00000000  11111111      int型255的二进制代码
```

由于无符号右移运算符>>>只是对32位和64位的值有意义，所以它并不像你想象的那样有用。因为你要记住，在表达式中过小的值总是被自动扩大为int型。这意味着符号位扩展和移动总是发生在32位而不是8位或16位。这样，对第7位以0开始的byte型的值进行无符号移动是不可能的，因为在实际移动运算时，是对扩大后的32位值进行操作。下面的例子说明了这一点：

```
// Unsigned shifting a byte value.
class ByteUShift {
  static public void main(String args[]) {
```

```
    char hex[] = {
      '0', '1', '2', '3', '4', '5', '6', '7',
      '8', '9', 'a', 'b', 'c', 'd', 'e', 'f'
    };
    byte b = (byte) 0xf1;
    byte c = (byte) (b >> 4);
    byte d = (byte) (b >>> 4);
    byte e = (byte) ((b & 0xff) >> 4);
    System.out.println("      b = 0x"
      + hex[(b >> 4) & 0x0f] + hex[b & 0x0f]);
    System.out.println("      b >> 4 = 0x"
      + hex[(c >> 4) & 0x0f] + hex[c & 0x0f]);
    System.out.println("      b >>> 4 = 0x"
      + hex[(d >> 4) & 0x0f] + hex[d & 0x0f]);
    System.out.println("( b & 0xff) >> 4 = 0x"
      + hex[(e >> 4) & 0x0f] + hex[e & 0x0f]);
    }
}
```

该程序的输出显示了无符号右移运算符>>>对byte型值处理时，实际上不是对byte型值直接操作，而是将其扩大到int型后再处理。在本例中变量b被赋为任意的负byte型值。对变量b右移4位后转换为byte型，将得到的值赋给变量c，因为有符号位扩展，所以该值为0xff。对变量b进行无符号右移4位操作后转换为byte型，将得到的值赋给变量d，你可能期望该值是0x0f，但实际上它是0xff，因为在移动之前变量b就被扩展为int型，已经有符号扩展位。最后一个表达式将变量b的值通过按位与运算将其变为8位，然后右移4位，然后将得到的值赋给变量e，这次得到了预想的结果0x0f。由于对变量d（它的值已经是0xff）进行按位与运算后的符号位的状态已经明了，所以注意，对变量d再没有进行无符号右移运算。

```
          B       = 0xf1
     b >> 4       = 0xff
     b >>> 4      = 0xff
(b & 0xff) >> 4   = 0x0f
```

4.2.5 位运算符赋值

所有的二进制位运算符都有一种将赋值与位运算组合在一起的简写形式。例如，下面两个语句都是将变量a右移4位后赋给a：

```
a = a >> 4;
a >>= 4;
```

同样，下面两个语句都是将表达式a OR b运算后的结果赋给a：

```
a = a | b;
a |= b;
```

下面的程序定义了几个int型变量，然后运用位赋值简写的形式将运算后的值赋给相应的变量：

```
class OpBitEquals {
  public static void main(String args[]) {
    int a = 1;
```

```
        int b = 2;
        int c = 3;

        a |= 4;
        b >>= 1;
        c <<= 1;
        a ^= c;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
    }
}
```

该程序的输出如下所示：

```
a = 3
b = 1
c = 6
```

4.3　关系运算符

关系运算符（relational operators）决定值和值之间的关系。例如决定相等不相等以及排列次序。关系运算符如表4-4所示：

表 4-4　关系运算符及其意义

运算符	意义
==	等于
!=	不等于
>	大于
<	小于
>=	大于等于
<=	小于等于

这些关系运算符产生的结果是布尔值。关系运算符常常用在if控制语句和各种循环语句的表达式中。

Java中的任何类型，包括整数，浮点数，字符，以及布尔型都可用“==”来比较是否相等，用“!=”来测试是否不等。注意Java（就像C和C++一样）比较是否相等的运算符是2个等号，而不是一个（注意：单等号是赋值运算符）。只有数字类型可以使用排序运算符进行比较。也就是，只有整数、浮点数和字符运算数可以用来比较哪个大哪个小。

前面已经说过，关系运算符的结果是布尔（boolean）类型。例如，下面的程序段对变量c的赋值是有效的：

```
int a = 4;
int b = 1;
boolean c = a < b;
```

在本例中，a<b（其结果是false）的结果存储在变量c中。

如果你有C/C++语言知识的背景，请注意下面的几条语句。在C/C++中，这些类型的语句是很常见的：

```
int done;
// ...
if(!done) ... // Valid in C/C++
if(done) ...  // but not in Java.
```

在Java中，这些语句必须写成下面这样：

```
if(done == 0)) ... // This is Java-style.
if(done != 0) ...
```

这样做的原因是Java定义真和假的方法和C/C++中的不一样。在C/C++中，真是任何非0的值而假是值0。在Java中，真值假值是非数字的，它和0或非0联系不到一起。因此，为了测试0值或非0值，你必须明确地用一个或多个关系运算符。

4.4 布尔逻辑运算符

布尔逻辑运算符的运算数只能是布尔型。而且逻辑运算的结果也是布尔类型（见表4-5）。

表 4-5 布尔逻辑运算符及其意义

运算符	含义
&	逻辑与
|	逻辑或
^	异或
||	短路或
&&	短路与
!	逻辑反
&=	逻辑与赋值（赋值的简写形式）
|=	逻辑或赋值（赋值的简写形式）
^=	异或赋值（赋值的简写形式）
==	相等
!=	不相等
?:	三元运算符（IF-THEN-ELSE）

布尔逻辑运算符“&”、“ |”、“ ^”，对布尔值的运算和它们对整数位的运算一样。逻辑运算符“!”的结果表示布尔值的相反状态：!true == false 和 !false == true。各个逻辑运算符的运算结果如表4-6所示：

表 4-6　布尔逻辑运算符的运算

A	B	A|B	A&B	A^B	!A
False	False	False	False	False	True
True	False	True	False	True	False
False	True	True	False	True	True
True	True	True	True	False	False

下面的例子和前面讲的位逻辑运算的几乎一样，只不过本例中的运算数是布尔逻辑值而不是二进制的位值：

```
// Demonstrate the boolean logical operators.
class BoolLogic {
  public static void main(String args[]) {
    boolean a = true;
    boolean b = false;
    boolean c = a | b;
    boolean d = a & b;
    boolean e = a ^ b;
    boolean f = (!a & b) | (a & !b);
    boolean g = !a;
    System.out.println("        a   = " + a);
    System.out.println("        b   = " + b);
    System.out.println("        a|b = " + c);
    System.out.println("        a&b = " + d);
    System.out.println("        a^b = " + e);
    System.out.println("!a&b|a&!b   = " + f);
    System.out.println("        !a  = " + g);
  }
}
```

在你运行该程序后，你会发现逻辑运算符对布尔值和位运算值的逻辑规则一样。而且从下面的输出你也可以看出Java中的布尔值是字符串常量“true”或“false”：

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

4.4.1　短路（short-circuit）逻辑运算符

Java 提供了两个在大多数其他计算机语言中没有的有趣的布尔运算符。这就是逻辑AND和逻辑OR的特殊的短路版本。从上表可以看出，在逻辑OR的运算中，如果第一个运算数A为真，则不管第二个运算数B是真是假，其运算结果为真。同样，在逻辑AND的运算中，如果第一个运算数A为假，则不管第二个运算数是真是假，其运算结果为假。如果运用||和&&形式，而不是|和&，那么一个运算数就能决定表达式的值，Java的短路版本就不会对第二个运算数求值，只有在需要时才对第二个运算数求值。为完成正确的功能，当右

边的运算数取决于左边的运算数是真或是假时，短路版本是很有用的。例如，下面的程序语句说明了短路逻辑运算符的优点，用它来防止被0除的错误：

```
if (denom != 0 && num / denom > 10)
```

既然用了短路AND运算符，就不会有当denom为0时产生的意外运行时错误。如果该行代码使用标准AND运算符（&），它将对两个运算数都求值，当出现被0除的情况时，就会产生运行时错误。

既然短路运算符在布尔逻辑运算中有效，那么就在布尔逻辑运算中全用它，而标准的AND和OR运算符（只有一个字符）仅在位运算中使用。然而，这条规则也有例外。例如，考虑下面的语句：

```
if(c==1 & e++ < 100) d = 100;
```

这里，使用标准AND运算符（单个的&）来保证不论c是否等于1，e都被自增量。

4.5 赋值运算符

在第2章中你已经使用过赋值运算符。下面我们正式讨论它。赋值运算符是一个等号“=”。它在Java中的运算与在其他计算机语言中的运算一样，其通用格式为：

```
var = expression;
```

这里，变量var的类型必须与表达式expression的类型一致。

赋值运算符有一个有趣的属性，你或许并不熟悉：它允许你对一连串变量赋值。例如，请看下面的例子：

```
int x, y, z;
x = y = z = 100; // set x, y, and z to 100
```

该例子使用一个赋值语句对变量 x、y、z 都赋值为100。这是因为“=”运算符产生右边表达式的值，因此 z = 100 的值是 100，然后该值被赋给 y ，并依次被赋给 x 。使用“串赋值”是给一组变量赋同一个值的简单办法。

4.6 ？运算符

Java提供一个特别的三元运算符（ternary）经常用于取代某个类型的if-then-else 语句。这个运算符就是?，并且它在Java中的用法和在C/C++中的几乎一样。该符号初看起来有些迷惑，但是一旦掌握了它，用？运算符是很方便高效的。？运算符的通用格式如下：

```
expression1 ? expression2 : expression3
```

其中，expression1是一个布尔表达式。如果expression1为真，那么expression2被求值；否则，expression3被求值。整个？表达式的值就是被求值表达式（expression2或expression3）的值。expression2和expression3是除了void以外的任何类型的表达式，且它们的类型必须相

同。

下面是一个利用？运算符的例子：

```
ratio = denom == 0 ? 0 : num / denom;
```

当Java计算这个表达式时，它首先看问号左边的表达式。如果 denom 等于0，那么在问号和冒号之间的表达式被求值，并且该值被作为整个？表达式的值。如果 denom 不等于零，那么在冒号之后的表达式被求值，并且该值被作为整个？表达式的值。然后将整个？表达式的值赋给变量ratio。

下面的程序说明了？运算符，该程序得到一个变量的绝对值。

```
// Demonstrate ?.
class Ternary {
  public static void main(String args[]) {
    int i, k;
    i = 10;
    k = i < 0 ? -i : i; // get absolute value of i
    System.out.print("Absolute value of ");
    System.out.println(i + " is " + k);
    i = -10;
    k = i < 0 ? -i : i; // get absolute value of i
    System.out.print("Absolute value of ");
    System.out.println(i + " is " + k);
  }
}
```

该程序的输出如下所示：

```
Absolute value of 10 is 10
Absolute value of -10 is 10
```

4.7　运算符优先级

表4-7显示了Java 运算符从最高到最低的优先级。注意第一行显示的项你通常不能把它们作为运算符:圆括号，方括号，点运算符。圆括号被用来改变运算的优先级。从前面章节我们知道，方括号用来表示数组的下标。点运算符用来将对象名和成员名连接起来，这将在本书的后面讨论。

表 4-7　Java 运算符优先级

最高			
()	[]	.	
++	−−	~	!
*	/	%	
+	−		
>>	>>>	<<	
>	>=	<	<=

续表

==	!=
&	
^	
|	
&&	
||	
?:	
=	op=
最低	

4.8 使用圆括号

圆括号（Parentheses）提高了括在其中的运算的优先级。这常常对于获得你需要的结果是必要的。例如，考虑下列表达式：

```
a >> b + 3
```

该表达式首先把 3 加到变量 b，得到一个中间结果，然后将变量a右移该中间结果位。该表达式可用添加圆括号的办法重写如下：

```
a >> (b + 3)
```

然而，如果你想先将a右移b位，得到一个中间结果，然后对该中间结果加3，你需要对表达式加如下的圆括号：

```
(a >> b) + 3
```

除了改变一个运算的正常优先级外，括号有时被用来帮助澄清表达式的含义。对于阅读你程序代码的人来说，理解一个复杂的表达式是困难的。对复杂表达式增加圆括号能帮助防止以后的混乱。例如，下面哪一个表达式更容易读呢？

```
a | 4 + c >> b & 7
(a | (((4 + c) >> b) & 7))
```

另外一点：圆括号（不管是不是多余的）不会降低你程序的运行速度。因此，添加圆括号可以减少含糊不清，不会对你的程序产生消极影响。

第 5 章　程序控制语句

编程语言使用控制（control）语句来产生执行流，从而完成程序状态的改变，如程序顺序执行和分支执行。Java的程序控制语句分为以下几类:选择，重复和跳转。根据表达式结果或变量状态选择（Selection）语句来使你的程序选择不同的执行路径。重复（Iteration）语句使程序能够重复执行一个或一个以上语句（也就是说，重复语句形成循环）。跳转（Jump）语句允许你的程序以非线性的方式执行。下面将分析Java 的所有控制语句。

如果你熟悉C/C++，那么掌握Java的控制语句将很容易。事实上，Java的控制语句与C/C++中的语句几乎完全相同。当然它们还是有一些差别的——尤其是break语句与continue语句。

5.1　Java的选择语句

Java支持两种选择语句：if语句和switch语句。这些语句允许你只有在程序运行时才能知道其状态的情况下，控制程序的执行过程。如果你没有C/C++的编程背景，你将会为这两个语句的强大功能和灵活性而感到吃惊。

5.1.1　if语句

if语句曾在第2章中介绍过，我们将在这里对它进行详细讨论。if语句是Java中的条件分支语句。它能将程序的执行路径分为两条。if语句的完整格式如下：

```
if (condition) statement1;
else statement2;
```

其中，if和else的对象都是单个语句（statement），也可以是程序块。条件condition可以是任何返回布尔值的表达式。else子句是可选的。

if语句的执行过程如下:如果条件为真，就执行if的对象（statement1）；否则，执行else的对象（statement2）。任何时候两条语句都不可能同时执行。考虑下面的例子：

```
int a, b;
// ...
if(a < b) a = 0;
else b = 0;
```

本例中，如果a小于b，那么a被赋值为0；否则，b被赋值为0。任何情况下都不可能使a和b都被赋值为0。

通常，用于控制if语句的表达式都包含关系运算符。当然，这在技术上并不是必要的。仅用一个布尔值来控制if语句也是可以的，如下面的程序段：

```
boolean dataAvailable;
```

```
// ...
if (dataAvailable)
  ProcessData();
else
  waitForMoreData();
```

记住，直接跟在if 或else语句后的语句只能有一句。如果你想包含更多的语句，你需要建一个程序块，如下面的例子：

```
int bytesAvailable;
// ...
if (bytesAvailable > 0) {
  ProcessData();
  bytesAvailable -= n;
} else
  waitForMoreData();
```

这里，如果变量bytesAvailable 大于0，则if块内的所有语句都会执行。

一些程序员觉得在使用if语句时在其后跟一个大括号是很方便的，甚至在只有一条语句的时侯也使用大括号。这使得在日后添加别的语句变得容易，并且你也不必担心忘记括号。事实上，当需要定义块时而未对其进行定义是一个导致错误的普遍原因。例如，考虑下面的程序段：

```
int bytesAvailable;
// ...
if (bytesAvailable > 0) {
  ProcessData();
  bytesAvailable -= n;
} else
  waitForMoreData();
  bytesAvailable = n;
```

由于编排的原因，看起来似乎 bytesAvailable = n语句应该在else子句中执行。然而，当你调用时，空白对Java无关紧要，编译器无法知道你的意图。这段程序会通过编译，但运用时会出错。上述例子应修改如下：

```
int bytesAvailable;
// ...
if (bytesAvailable > 0) {
  ProcessData();
  bytesAvailable -= n;
} else {
  waitForMoreData();
  bytesAvailable = n;
}
```

嵌套 if 语句

嵌套（nested）if语句是指该if语句为另一个if或者else语句的对象。在编程时经常要用到嵌套if语句。当你使用嵌套if语句时，需记住的要点就是：一个else语句总是对应着和它在同一个块中的最近的if语句，而且该if语句没有与其他else语句相关联。下面是一个例子：

```
if(i == 10) {
  if(j < 20) a = b;
  if(k > 100) c = d;     // this if is
  else a = c;            // associated with this else
}
else a = d;              // this else refers to if(i == 10)
```

如注释所示，最后一个else语句没有与if（j <20）相对应，因为它们不在同一个块（尽管if（j <20）语句是没有与else配对最近的if语句）。最后一个else语句对应着if（i==10）。内部的else语句对应着if（k>100），因为它是同一个块中最近的if语句。

if-else-if 阶梯

基于嵌套if语句的通用编程结构被称为 if-else-if 阶梯。它的语法如下：

```
if(condition)
  statement;
else if(condition)
  statement;
else if(condition)
  statement;
.
.
.
else
  statement;
```

条件表达式从上到下被求值。一旦找到为真的条件，就执行与它关联的语句，该阶梯的其他部分就被忽略了。如果所有的条件都不为真，则执行最后的else语句。最后的else语句经常被作为默认的条件，即如果所有其他条件测试失败，就执行最后的else语句。如果没有最后的else语句，而且所有其他的条件都失败，那程序就不做任何动作。

下面的程序通过使用if-else-if阶梯来确定某个月是什么季节。

```
// Demonstrate if-else-if statements.
class IfElse {
  public static void main(String args[]) {
    int month = 4; // April
    String season;

    if(month == 12 || month == 1 || month == 2)
      season = "Winter";
    else if(month == 3 || month == 4 || month == 5)
      season = "Spring";
    else if(month == 6 || month == 7 || month == 8)
      season = "Summer";
    else if(month == 9 || month == 10 || month == 11)
      season = "Autumn";
    else
      season = "Bogus Month";

    System.out.println("April is in the " + season + ".");
  }
}
```

该程序产生如下输出：

```
April is in the Spring.
```

在往下继续讲之前，你可能想要先试验这个程序。你将看到，不管你给month什么值，该阶梯中有而且只有一个语句执行。

5.1.2 switch语句

switch语句是Java的多路分支语句。它提供了一种基于一个表达式的值来使程序执行不同部分的简单方法。因此，它提供了一个比一系列if-else-if语句更好的选择。switch语句的通用形式如下：

```
switch (expression) {
  case value1:
  // statement sequence
  break;
case value2:
  // statement sequence
  break;
.
.
.
case valueN:
  // statement sequence
  break;
default:
  // default statement sequence
}
```

表达式expression必须为byte，short，int或char类型。每个case语句后的值value必须是与表达式类型兼容的特定的一个常量（它必须为一个常量，而不是变量）。重复的case值是不允许的。

switch语句的执行过程如下：表达式的值与每个case语句中的常量作比较。如果发现了一个与之相匹配的，则执行该case语句后的代码。如果没有一个case常量与表达式的值相匹配，则执行default语句。当然，default语句是可选的。如果没有相匹配的case语句，也没有default语句，则什么也不执行。

在case语句序列中的break语句将引起程序流从整个switch语句退出。当遇到一个break语句时，程序将从整个switch语句后的第一行代码开始继续执行。这有一种“跳出” switch语句的效果。

下面是一个使用switch语句的简单例子：

```
// A simple example of the switch.
class SampleSwitch {
  public static void main(String args[]) {
    for(int i=0; i<6; i++)
      switch(i) {
        case 0:
          System.out.println("i is zero.");
          break;
```

```
      case 1:
        System.out.println("i is one.");
        break;
      case 2:
        System.out.println("i is two.");
        break;
      case 3:
          System.out.println("i is three.");
          break;
      default:
        System.out.println("i is greater than 3.");
    }
  }
}
```

该程序的输出如下：

```
i is zero.
i is one.
i is two.
i is three.
i is greater than 3.
i is greater than 3.
```

从中可以看出，每一次循环，与i值相配的case常量后的相关语句就被执行。其他语句则被忽略。当i大于3时，没有可以匹配的case语句，因此执行default语句。

break语句是可选的。如果你省略了break语句，程序将继续执行下一个case语句。有时需要在多个case语句之间没有break语句。例如下面的程序：

```
// In a switch, break statements are optional.
class MissingBreak {
  public static void main(String args[]) {
    for(int i=0; i<12; i++)
      switch(i) {
        case 0:
        case 1:
        case 2:
        case 3:
        case 4:
          System.out.println("i is less than 5");
          break;
        case 5:
        case 6:
        case 7:
        case 8:
        case 9:
          System.out.println("i is less than 10");
          break;
        default:
          System.out.println("i is 10 or more");
      }
  }
}
```

该程序产生的输出如下：

```
i is less than 5
i is less than 5
i is less than 5
i is less than 5
i is less than 5
i is less than 10
i is less than 10
i is less than 10
i is less than 10
i is less than 10
i is 10 or more
i is 10 or more
```

正如该程序所演示的那样，如果没有break语句，程序将继续执行下面的每一个case语句，直到遇到break语句（或switch语句的末尾）。

当然该例子是为了示例而人为构造的，省略break语句在真实的程序中有许多实际的应用。为了说明它更现实的用法，让我们考虑下例对以前显示季节例子的重写。这个重写的版本使用switch语句来使程序的执行更高效。

```
// An improved version of the season program.
class Switch {
  public static void main(String args[]) {
     int month = 4;
     String season;
     switch (month) {
        case 12:
        case 1:
        case 2:
            season = "Winter";
            break;
        case 3:
        case 4:
        case 5:
            season = "Spring";
            break;
        case 6:
        case 7:
        case 8:
            season = "Summer";
            break;
        case 9:
        case 10:
        case 11:
            season = "Autumn";
            break;
        default:
            season = "Bogus Month";
       }
       System.out.println("April is in the " + season + ".");
    }
}
```

嵌套 switch 语句

可以将一个switch语句作为一个外部switch语句的语句序列的一部分，这称为嵌套switch语句。因为一个switch语句定义了自己的块，外部switch语句和内部switch语句的case常量不会产生冲突。例如，下面的程序段是完全正确的：

```
switch(count) {
  case 1:
    switch(target) { // nested switch
      case 0:
        System.out.println("target is zero");
        break;
      case 1: // no conflicts with outer switch
        System.out.println("target is one");
        break;
  }
  break;
case 2: // ...
```

本例中，内部switch语句中的 case 1：语句与外部switch语句中的case 1：语句不冲突。变量count仅与外层的case语句相比较。如果变量count为1，则变量target与内层的case语句相比较。

概括起来说，switch语句有3个重要的特性需注意：

- switch语句不同于if语句的是switch语句仅能测试相等的情况，而if语句可计算任何类型的布尔表达式。也就是switch语句只能寻找case常量间某个值与表达式的值相匹配。
- 在同一个switch语句中没有两个相同的case常量。当然，外部switch语句中的case常量可以和内部switch语句中的case常量相同。
- switch语句通常比一系列嵌套if语句更有效。

最后一点尤其有趣，因为它使我们知道Java编译器如何工作。当编译一个switch语句时，Java编译器将检查每个case常量并且创造一个“跳转表”，这个表将用来在表达式值的基础上选择执行路径。因此，如果你需要在一组值中做出选择，switch语句将比与之等效的if-else语句快得多。编译器可以这样做是因为它知道case常量都是同类型的，所要做的只是将它与switch表达式相比较看是否相等。对于一系列的if表达式，编译器就无此功能。

5.2　循 环 语 句

Java的循环语句有for，while和 do-while。这些语句创造了我们通常所称的循环（loops）。你可能知道，一个循环重复执行同一套指令直到一个结束条件出现。你将看到，Java有适合任何编程所需要的循环结构。

5.2.1 while语句

while语句是Java最基本的循环语句。当它的控制表达式是真时，while语句重复执行一个语句或语句块。它的通用格式如下：

```
while(condition) {
   // body of  loop
}
```

条件condition可以是任何布尔表达式。只要条件表达式为真，循环体就被执行。当条件condition为假时，程序控制就传递到循环后面紧跟的语句行。如果只有单个语句需要重复，大括号是不必要的。

下面的while循环从10开始进行减计数，打印出10行“tick”。

```
// Demonstrate the while loop.
class While {
  public static void main(String args[]) {
    int n = 10;

    while(n > 0) {
      System.out.println("tick " + n);
      n--;
    }
  }
}
```

当你运行这个程序，它将“tick”10次：

```
tick 10
tick 9
tick 8
tick 7
tick 6
tick 5
tick 4
tick 3
tick 2
tick 1
```

因为while语句在循环一开始就计算条件表达式，若开始时条件为假，则循环体一次也不会执行。例如，下面的程序中，对println( )的调用从未被执行过：

```
int a = 10, b = 20;

while(a > b)
   System.out.println("This will not be displayed");
```

while循环（或Java的其他任何循环）的循环体可以为空。这是因为一个空语句（null statement)（仅由一个分号组成的语句）在Java的语法上是合法的。例如，下面的程序：

```
// The target of a loop can be empty.
class NoBody {
  public static void main(String args[]) {
```

```
    int i, j;

    i = 100;
    j = 200;

    // find midpoint between i and j
    while(++i < --j) ; // no body in this loop

    System.out.println("Midpoint is " + i);
  }
}
```

该程序找出变量i和变量j的中间点。它产生的输出如下：

```
Midpoint is 150
```

该程序中的while循环是这样执行的。值i自增，而值j自减，然后比较这两个值。如果新的值i仍比新的值j小，则进行循环。如果i等于或大于j，则循环停止。在退出循环前，i将保存原始i和j的中间值（当然，这个程序只有在开始时i比j小的情况下才执行）。正如你看到的，这里不需要循环体。所有的行为都出现在条件表达式自身内部。在专业化的Java代码中，一些可以由控制表达式本身处理的短循环通常都没有循环体。

5.2.2　do-while循环

如你刚才所见，如果while循环一开始条件表达式就是假的，那么循环体就根本不被执行。然而，有时需要在开始时条件表达式即使是假的情况下，while循环至少也要执行一次。换句话说，有时你需要在一次循环结束后再测试中止表达式，而不是在循环开始时。幸运的是，Java就提供了这样的循环：do-while循环。do-while循环总是执行它的循环体至少一次，因为它的条件表达式在循环的结尾。它的通用格式如下：

```
do {
  // body of loop
} while (condition);
```

do-while循环总是先执行循环体，然后再计算条件表达式。如果表达式为真，则循环继续。否则，循环结束。对所有的Java循环都一样，条件condition必须是一个布尔表达式。

下面是一个重写的“tick”程序，用来演示do-while循环。它的输出与先前程序的输出相同。

```
// Demonstrate the do-while loop.
class DoWhile {
  public static void main(String args[]) {
    int n = 10;

    do {
      System.out.println("tick " + n);
      n--;
    } while(n > 0);
  }
}
```

该程序中的循环虽然在技术上是正确的，但可以像如下这样编写更为高效：

```
do {
  System.out.println("tick " + n);
} while(--n > 0);
```

在本例中，表达式“-- n > 0 ”将n值的递减与测试n是否为0组合在一个表达式中。它的执行过程是这样的。首先，执行-- n 语句，将变量n递减，然后返回n的新值。这个值再与0比较，如果比0大，则循环继续。否则结束。

do-while循环在你编制菜单选择时尤为有用，因为通常都想让菜单循环体至少执行一次。下面的程序是一个实现Java选择和重复语句的很简单的帮助系统：

```
// Using a do-while to process a menu selection
class Menu {
  public static void main(String args[])
    throws java.io.IOException {
    char choice;

    do {
      System.out.println("Help on:");
      System.out.println(" 1. if");
      System.out.println(" 2. switch");
      System.out.println(" 3. while");
      System.out.println(" 4. do-while");
      System.out.println(" 5. for\n");
      System.out.println("Choose one:");
      choice = (char) System.in.read();
    } while( choice < '1' || choice > '5');

    System.out.println("\n");

    switch(choice) {
      case '1':
        System.out.println("The if:\n");
        System.out.println("if(condition) statement;");
        System.out.println("else statement;");
      break;
    case '2':
      System.out.println("The switch:\n");
      System.out.println("switch(expression) {");
      System.out.println(" case constant:");
      System.out.println(" statement sequence");
      System.out.println(" break;");
      System.out.println(" // ...");
      System.out.println("}");
      break;
    case '3':
      System.out.println("The while:\n");
      System.out.println("while(condition) statement;");
      break;
    case '4':
      System.out.println("The do-while:\n");
      System.out.println("do {");
```

```
      System.out.println(" statement;");
      System.out.println("} while (condition);");
      break;
    case '5':
      System.out.println("The for:\n");
      System.out.print("for(init; condition; iteration)");
      System.out.println(" statement;");
      break;
    }
  }
}
```

下面是这个程序执行的一个样本输出：

```
Help on:
  1. if
  2. switch
  3. while
  4. do-while
  5. for
Choose one:
4
The do-while:
do {
  statement;
} while (condition);
```

在程序中，do-while循环用来验证用户是否输入了有效的选择。如果没有，则要求用户重新输入。因为菜单至少要显示一次，do-while循环是完成此任务的合适语句。

关于此例的其他几点：注意从键盘输入字符通过调用System.in.read( )来读入。这是一个Java 的控制台输入函数。尽管Java的终端 I/O （输入/输出)方法将在第12章中详细讨论，在这里使用 System.in.read ( )来读入用户的选择。它从标准的输入读取字符（返回整数，因此将返回值choice定义为字符型）。默认地，标准输入是按行进入缓冲区的，因此在你输入的任何字符被送到你的程序以前，必须按回车键。

Java的终端输入功能相当有限且不好使用。进一步说，大多数现实的Java程序和applets（小应用程序）都具有图形界面并且是基于窗口的。因此，这本书使用终端的输入并不多。然而，它在本例中是有用的。另外一点:因为使用System.in.read ( )，程序必须指定throws java.io.IOException子句。这行代码对于处理输入错误是必要的。这是Java的异常处理的一部分，将在第10章讨论。

5.2.3　for循环

在第2章曾使用过一个for循环的简单格式。你将看到，for循环是一个功能强大且形式灵活的结构。下面是for循环的通用格式：

```
for(initialization; condition; iteration) {
  // body
}
```

如只有一条语句需要重复，大括号就没有必要。

for循环的执行过程如下。第一步，当循环启动时，先执行其初始化部分。通常，这是设置循环控制变量值的一个表达式，作为控制循环的计数器。重要的是你要理解初始化表达式仅被执行一次。下一步，计算条件condition的值。条件condition必须是布尔表达式。它通常将循环控制变量与目标值相比较。如果这个表达式为真，则执行循环体；如果为假，则循环终止。再下一步执行循环体的反复部分。这部分通常是增加或减少循环控制变量的一个表达式。接下来重复循环，首先计算条件表达式的值，然后执行循环体，接着执行反复表达式。这个过程不断重复直到控制表达式变为假。

下面是使用for循环的“tick”程序：

```
// Demonstrate the for loop.
class ForTick {
  public static void main(String args[]) {
    int n;

    for(n=10; n>0; n--)
      System.out.println("tick " + n);
  }
}
```

在 for 循环中声明循环控制变量

控制for循环的变量经常只是用于该循环，而不用在程序的其他地方。在这种情况下，可以在循环的初始化部分中声明变量。例如，下面重写了前面的程序，使变量 n 在for循环中被声明为整型：

```
// Declare a loop control variable inside the for.
class ForTick {
  public static void main(String args[]) {

    // here, n is declared inside of the for loop
    for(int n=10; n>0; n--)
      System.out.println("tick " + n);
  }
}
```

当你在for循环内声明变量时，必须记住重要的一点:该变量的作用域在for语句执行后就结束了（因此，该变量的作用域就局限于for循环内）。在for循环外，变量就不存在了。如果你在程序的其他地方需要使用循环控制变量，你就不能在for循环中声明它。

由于循环控制变量不会在程序的其他地方使用，大多数程序员都在for循环中来声明它。例如，以下为测试素数的一个简单程序。注意由于其他地方不需要i，所以循环控制变量i在for循环中声明。

```
// Test for primes.
class FindPrime {
  public static void main(String args[]) {
    int num;
    boolean isPrime = true;

    num = 14;
```

```
    for(int i=2; i <= num/2; i++) {
      if((num % i) == 0) {
        isPrime = false;
        break;
      }
    }
    if(isPrime) System.out.println("Prime");
    else System.out.println("Not Prime");
  }
}
```

使用逗号

你可能经常需要在初始化和for循环的反复部分包括超过一个变量的声明。例如，考虑下面程序的循环部分：

```
class Sample {
  public static void main(String args[]) {
    int a, b;
    b = 4;
    for(a=1; a<b; a++) {
      System.out.println("a = " + a);
      System.out.println("b = " + b);
      b--;
    }
  }
}
```

如你所看到的，循环被两个相互作用的变量控制。由于循环被两个变量控制，如果两个变量都能被定义在for循环中，而变量b不需要通过人工处理将是很有用的。幸好，Java提供了一个完成此任务的方法。为了允许两个或两个以上的变量控制循环，Java允许你在for循环的初始化部分和反复部分声明多个变量，每个变量之间用逗号分开。

使用逗号，前面的for循环将更高效，改写后的程序如下：

```
// Using the comma.
class Comma {
  public static void main(String args[]) {
    int a, b;

    for(a=1,b=4; a<b; a++,b--) {
      System.out.println("a = " + a);
      System.out.println("b = " + b);
    }
  }
}
```

在本例中，初始化部分把两个变量a和 b都定义了。在循环的反复部分，用两个逗号分开的语句在每次循环重复时都执行。程序输出如下：

```
a = 1
b = 4
a = 2
b = 3
```

注意：如果你对C/C++熟悉，你就会知道逗号是一个运算符，能在任何有效的表达式中使用。然而，在Java 中不是这样。在 Java中 ，逗号仅仅是一个分隔符，只适用于for循环。

5.2.4 for循环的一些变化

for循环支持一些变化，这增加了它的功能和灵活性。for循环这样灵活是因为它的3部分（初始化部分，条件测试部分和反复部分）并不仅用于它们所限定的那些目的。事实上，for 循环的3部分能被用于你需要的任何目的。让我们看一些例子。

最普通的变化之一包含在条件表达式中。具体地说，条件表达式可以不需要用循环变量和目标值的比较来测试循环条件。事实上，控制for循环的条件可以是任何布尔表达式。例如，考虑下列程序片段：

```
boolean done = false;

for(int i=1; !done; i++) {
  // ...
  if(interrupted()) done = true;
}
```

在本例中，for循环将一直运行，直到布尔型变量done被设置为真。for循环的条件部分不测试值i。

下面是for循环的另外一个有趣的变化。在Java中可以使for循环的初始化、条件或者反复部分中的任何或者全部都为空，如下面的程序：

```
// Parts of the for loop can be empty.
class ForVar {
  public static void main(String args[]) {
    int i;
    boolean done = false;

    i = 0;
    for( ; !done; ) {
      System.out.println("i is " + i);
      if(i == 10) done = true;
      i++;
    }
  }
}
```

本例中，初始化部分和反复部分被移到了for循环以外。这样，for循环的初始化部分和反复部分是空的。当这个简单的例子中，for循环中没有值，确实，这种风格被认为是相当差的，有时这种风格也是有用的。例如，如果初始条件由程序中其他部分的复杂表达式来定义，或者循环控制变量的改变由发生在循环体内的行为决定，而且这种改变是一种非顺序的方式，这种情况下，可以使for循环的这些部分为空。

下面是for循环变化的又一种方式。如果for循环的三个部分全为空，你就可以创建一个无限循环（从来不停止的循环）。例如：

```
for( ; ; ) {
  // ...
}
```

这个循环将始终运行，因为没有使它终止的条件。尽管有一些程序，例如操作系统命令处理器，需要无限循环，但大多数“无限循环”实际上是具有特殊终止要求的循环。在不久你将看到如何不用正常的条件表达式来终止这种类型的循环。

5.2.5　循环嵌套

和其他编程语言一样，Java允许循环嵌套。也就是，一个循环在另一个循环之内。例如，下面的程序就是循环嵌套：

```
// Loops may be nested.
class Nested {
  public static void main(String args[]) {
    int i, j;

    for(i=0; i<10; i++) {
      for(j=i; j<10; j++)
        System.out.print(".");
      System.out.println();
    }
  }
}
```

该程序产生的输出如下所示：

```
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . .
. . . .
. . .
. .
.
```

5.3　跳 转 语 句

Java 支持 3 种跳转语句：break，continue和return。这些语句把控制转移到程序的其他部分。下面对每一种语句进行讨论。

注意：除了这里讨论的跳转语句，Java还支持另一种能改变你程序执行流程的方法:通过异常处理。异常处理提供了一种结构化的方法，通过该方法可以使你的程序捕获并处理运行时刻错误。它由下列五个关键字来控制：try，catch，throw，throws，和 finally。实质上，异常处理机制允许你的程序完成一个非局部的分支跳转。由于异常处理是一个大话题，我们将在第 10 章专门讨论。

5.3.1 使用break语句

在Java中，break语句有3种作用。第一，你已经看到，在switch语句中，它被用来终止一个语句序列。第二，它能被用来退出一个循环。第三，它能作为一种“先进”的goto 语句来使用。下面对最后 2种用法进行解释。

使用 break 退出循环

可以使用break语句直接强行退出循环，忽略循环体中的任何其他语句和循环的条件测试。在循环中遇到break语句时，循环被终止，程序控制在循环后面的语句重新开始。下面是一个简单的例子：

```
// Using break to exit a loop.
class BreakLoop {
  public static void main(String args[]) {
    for(int i=0; i<100; i++) {
      if(i == 10) break; // terminate loop if i is 10
      System.out.println("i: " + i);
    }
    System.out.println("Loop complete.");
  }
}
```

该程序产生如下的输出：

```
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
Loop complete.
```

正如你看到的那样，尽管for循环被设计为从 0执行到99，但是当i等于10时，break语句终止了程序。

break语句能用于任何 Java循环中，包括人们有意设置的无限循环。例如，将上一个程序用while循环改写如下。该程序的输出和刚才看到的输出一样。

```
// Using break to exit a while loop.
class BreakLoop2 {
  public static void main(String args[]) {
    int i = 0;

    while(i < 100) {
      if(i == 10) break; // terminate loop if i is 10
      System.out.println("i: " + i);
      i++;
    }
```

```
    System.out.println("Loop complete.");
  }
}
```

在一系列嵌套循环中使用break语句时，它将仅仅终止最里面的循环。例如：

```
// Using break with nested loops.
class BreakLoop3 {
  public static void main(String args[]) {
    for(int i=0; i<3; i++) {
      System.out.print("Pass " + i + ": ");
      for(int j=0; j<100; j++) {
        if(j == 10) break; // terminate loop if j is 10
        System.out.print(j + " ");
      }
      System.out.println();
    }
    System.out.println("Loops complete.");
  }
}
```

该程序产生如下的输出：

```
Pass 0: 0 1 2 3 4 5 6 7 8 9
Pass 1: 0 1 2 3 4 5 6 7 8 9
Pass 2: 0 1 2 3 4 5 6 7 8 9
Loops complete.
```

从中可以看出，在内部循环中的break语句仅仅终止了该循环，外部的循环不受影响。

关于break，在这里要记住两点。首先，一个循环中可以有一个以上的break语句。但要小心，太多的break语句会破坏你的代码结构。其次，switch语句中的break仅仅影响该switch语句，而不会影响其中的任何循环。

注意：break不是被设计来提供一种正常的循环终止的方法。循环的条件语句是专门用来终止循环的。只有在某类特殊的情况下，才用break语句来取消一个循环。

把 break 当作 goto 的一种形式来用

break语句除了在switch语句和循环中使用之外，它还能作为goto 语句的一种“文明”形式来使用。Java中没有 goto语句，因为goto语句提供了一种改变程序运行流程的非结构化方式。这通常使程序难以理解和难于维护。它也阻止了某些编译器的优化。但是，有些地方goto语句对于构造流程控制是有用的而且是合法的。例如，从嵌套很深的循环中退出时， goto语句就很有帮助。因此，Java定义了break语句的一种扩展形式来处理这种情况。通过使用这种形式的break，你可以终止一个或者几个代码块。这些代码块不必是一个循环或一个switch语句的一部分，它们可以是任何的块。而且，由于这种形式的break语句带有标签，你可以明确指定执行从何处重新开始。你将看到，break带给你的是goto的益处，并舍弃了goto语句带来的麻烦。

标签break语句的通用格式如下所示：

```
break label;
```

这里，标签label是标识代码块的标签。当这种形式的break执行时，控制被传递出指定的代码块。被加标签的代码块必须包围break语句，但是它不需要是直接的包围break的块。这意味着你可以使用一个加标签的break语句退出一系列的嵌套块。但是你不能使用break语句将控制传递到不包含break语句的代码块。

要指定一个代码块，在其开头加一个标签即可。标签（label）可以是任何合法有效的Java标识符后跟一个冒号。一旦你给一个块加上标签后，你就可以使用这个标签作为break语句的对象了。这样做会使执行在加标签的块的结尾重新开始。例如，下面的程序示例了 3个嵌套块，每一个都有它自己的标签。break语句使执行向前跳，调过了定义为标签second的代码块结尾，跳过了2个 println ( )语句。

```
// Using break as a civilized form of goto.
class Break {
  public static void main(String args[]) {
    boolean t = true;

    first: {
      second: {
        third: {
          System.out.println("Before the break.");
          if(t) break second; // break out of second block
          System.out.println("This won't execute");
        }
        System.out.println("This won't execute");
      }
      System.out.println("This is after second block.");
    }
  }
}
```

运行该程序，产生如下的输出：

```
Before the break.
This is after second block.
```

标签break语句的一个最普遍的用法是退出循环嵌套。例如，下面的程序中，外层的循环只执行了一次：

```
// Using break to exit from nested loops
class BreakLoop4 {
  public static void main(String args[]) {
    outer: for(int i=0; i<3; i++) {
      System.out.print("Pass " + i + ": ");
      for(int j=0; j<100; j++) {
        if(j == 10) break outer; // exit both loops
        System.out.print(j + " ");
      }
        System.out.println("This will not print");
    }
    System.out.println("Loops complete.");
  }
}
```

该程序产生如下的输出：

```
Pass 0: 0 1 2 3 4 5 6 7 8 9 Loops complete.
```

你可以看到，当内部循环退到外部循环时，两个循环都被终止了。

记住如果一个标签不在包围break的块中定义，你就不能break到该标签。例如，下面的程序就是非法的，且不会被编译：

```
// This program contains an error.
class BreakErr {
  public static void main(String args[]) {

    one: for(int i=0; i<3; i++) {
      System.out.print("Pass " + i + ": ");
    }

    for(int j=0; j<100; j++) {
      if(j == 10) break one; // WRONG
      System.out.print(j + " ");
    }
  }
}
```

因为标签为one的循环没有包围break语句，所以不能将控制传递到该块。

5.3.2　使用continue语句

有时强迫一个循环提早反复是有用的。也就是，你可能想要继续运行循环，但是要忽略这次重复剩余的循环体的语句。实际上，goto只不过是跳过循环体，到达循环的尾部。continue语句是break语句的补充。在while和do while循环中，continue语句使控制直接转移给控制循环的条件表达式，然后继续循环过程。在for循环中，循环的反复表达式被求值，然后执行条件表达式，循环继续执行。对于这3种循环，任何中间的代码将被旁路。

下例使用continue语句，使每行打印2个数字：

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

该程序使用%（模）运算符来检验变量i是否为偶数，如果是，循环继续执行而不输出一个新行。该程序的结果如下：

```
0 1
2 3
4 5
```

```
6 7
8 9
```

对于break语句，continue可以指定一个标签来说明继续哪个包围的循环。下面的例子运用continue语句来打印0到9的三角形乘法表：

```
// Using continue with a label.
class ContinueLabel {
  public static void main(String args[]) {
outer: for (int i=0; i<10; i++) {
       for(int j=0; j<10; j++) {
         if(j > i) {
           System.out.println();
           continue outer;
          }
          System.out.print(" " + (i * j));
        }
     }
     System.out.println();
  }
}
```

在本例中的continue语句终止了计数j的循环而继续计数i的下一次循环反复。该程序的输出如下：

```
0
0 1
0 2 4
0 3 6 9
0 4 8 12 16
0 5 10 15 20 25
0 6 12 18 24 30 36
0 7 14 21 28 35 42 49
0 8 16 24 32 40 48 56 64
0 9 18 27 36 45 54 63 72 81
```

很好的利用continue语句的情况很少，一个原因是Java提供了一系列丰富的循环语句，可以适用于绝大多数应用程序。但是，对于那些需要提早反复的特殊情形，continue语句提供了一个结构化的方法来实现。

5.3.3 使用return语句

最后一个控制语句是return。return语句用来明确地从一个方法返回。也就是，return语句使程序控制返回到调用它的方法。因此，将它分类为跳转语句。尽管对return语句的详细讨论在第 7 章开始，这里对其作简要地介绍。

在一个方法的任何时间，return语句可被用来使正在执行的分支程序返回到调用它的方法。下面的例子说明这一点。下例中，由于是Java 运行系统调用main()，因此，return语句使程序执行返回到Java运行系统。

```
// Demonstrate return.
class Return {
```

```
  public static void main(String args[]) {
    boolean t = true;

    System.out.println("Before the return.");

    if(t) return; // return to caller

    System.out.println("This won't execute.");
  }
}
```

该程序的结果如下：

```
Before the return.
```

正如你看到的一样，最后的println( )语句没有被执行。一旦return语句被执行，程序控制传递到它的调用者。

最后一点：在上面的程序中，if(t)语句是必要的。没有它，Java编译器将标记“执行不到的代码”（unreachable code）错误，因为编译器知道最后的println（）语句将永远不会被执行。为阻止这个错误，为了这个例子能够执行，在这里使用if语句来“蒙骗”编译器。

第6章　介 绍 类

类是Java的核心和本质。它是Java语言的基础，因为类定义了对象的本性。既然类是面向对象程序设计Java语言的基础，因此，你想要在Java程序中实现的每一个概念都必须封装在类以内。

因为类是Java的基础，所以在本章和以后几章中对其进行介绍。本章将介绍类的基本元素，并学习如何运用类来创建对象。同时也将学习方法、构造函数及this这个关键字。

6.1　类 基 础

从本书的开始我们就使用类了。当然，使用的都是非常简单的类。在前面几章中创造的类主要都包含在main( )方法中，用它来表明Java句法的基础。你将看到，类的功能实质上比你到目前为止看到的要强大得多。

也许理解类的最重要的事情就是它定义了一种新的数据类型。一旦定义后，就可以用这种新类型来创建该类型的对象。这样，类就是对象的模板（template），而对象就是类的一个实例（instance）。既然一个对象就是一个类的实例，所以你经常看到object和instance这两个词可以互换使用。

6.1.1　类的通用格式

当你定义一个类时，你要声明它准确的格式和属性。你可以通过指定它包含的数据和操作数据的代码来定义类。尽管非常简单的类可能只包含代码或者只包含数据，但绝大多数实际的类都包含上述两者。你将看到，类的代码定义了该类数据的接口。

使用关键字class来创建类。在这一点上，类实际上被限制在它的完全格式中。类可以（并且常常）是一个组合体。类定义的通用格式如下所示：

```
class classname  {
  type instance-variable1;
type instance-variable2;
// ...
  type instance-variableN;

type methodname1(parameter-list) {
// body of method
}
type methodname2(parameter-list) {
  // body of method
  }
// ...
  type methodnameN(parameter-list) {
// body of method
```

```
  }
}
```

在类中，数据或变量被称为实例变量（instance variables），代码包含在方法（methods）内。定义在类中的方法和实例变量被称为类的成员（members）。在大多数类中，实例变量被定义在该类中的方法操作和存取。这样，方法决定该类中的数据如何使用。

定义在类中的变量被称为实例变量，这是因为类中的每个实例(也就是类的每个对象)都包含它自己对这些变量的拷贝。这样，一个对象的数据是独立的且是惟一的。关于这一点我们将马上讨论，但是这是一个重要的概念，因此要早一点学习。

所有的方法和我们到目前为止用过的方法main()的形式一样。但是，以后讲到的方法将不仅仅是被指定为static或public。注意类的通用格式中并没有指定main()方法。Java类不需要main( )方法。main()方法只是在定义您程序的起点时用到。而且，Java小应用程序也不要求main( )方法。

注意：如果你有C++编程经验请注意，类的声明和方法的实现要存储在同一个地方并且不能被单独定义。由于所有类的定义必须全部定义在一个单个的源文件中，这有时会生成很大的.java文件。在Java中设计这个特征是因为从长远来说，在一个地方指明，定义以及实现将使代码更易于维护。

6.1.2　一个简单的类

让我们先从一个简单的例子来开始对类的研究。下面定义了一个名为box的类，它定义了3个实例变量：width，height和depth。当前，box类不包含任何方法（但是随后将增加一些）。

```
class Box {
  double width;
  double height;
  double depth;
}
```

前面已经说过，一个类定义一个新的数据类型。在本例中，新的数据类型名为Box。你可以使用这个名字来声明Box类型的对象。记住类声明只是创建一个模板（或类型描述），它并不会创建一个实际的对象。因此，上述代码不会生成任何Box类型的对象实体。

要真正创建一个Box对象，你必须使用下面的语句：

```
Box mybox = new Box(); // create a Box object called mybox
```

这个语句执行后，mybox就是Box的一个实例了。因此，它将具有“物理的”真实性。现在，先不必考虑这个语句的一些细节问题。

每次你创建类的一个实例时，你是在创建一个对象，该对象包含它自己的由类定义的每个实例变量的拷贝。因此，每个Box对象都将包含它自己的实例变量拷贝，这些变量即width，height，和 depth。要访问这些变量，你要使用点号“.”运算符。点号运算符（dot operator）将对象名和成员名连接起来。例如，要将mybox的width变量赋值为100，使用下面的语句：

```
mybox.width = 100;
```

该语句告诉编译器对mybox对象内包含的width变量拷贝的值赋为100。通常情况下，你可以使用点号运算符来访问一个对象内的实例变量和方法。

下面是使用Box类的完整程序：

```
/* A program that uses the Box class.

   Call this file BoxDemo.java
*/
class Box {
  double width;
  double height;
  double depth;
}

// This class declares an object of type Box.
class BoxDemo {
  public static void main(String args[]) {
    Box mybox = new Box();
    double vol;

    // assign values to mybox's instance variables
    mybox.width = 10;
    mybox.height = 20;
    mybox.depth = 15;

    // compute volume of box
    vol = mybox.width * mybox.height * mybox.depth;

    System.out.println("Volume is " + vol);
  }
}
```

你应该把包含该程序的的文件命名为BoxDemo.java，因为main( )方法在名为 BoxDemo 的类中，而不是名为Box的类中。当你编译这个程序时，你会发现生成了两个“.class”文件，一个属于box，另一个属于BoxDemo。Java编译器自动将每个类保存在它自己的“ .class ”文件中。没有必要分别将Box类和Boxdemo类放在同一个源文件中。你可以分别将它们放在各自的文件中，并分别命名为Box.Java和 BoxDemo.java。

要运行这个程序，你必须执行BoxDemo.class。运行该程序后，你会看见如下输出：

```
Volume is 3000.0
```

前面已经讲过，每个对象都含有它自己的、由它的类定义的实例变量的拷贝。因此，假设你有两个Box对象，每个对象都有其自己的depth，width和height拷贝。改变一个对象的实例变量对另外一个对象的实例变量没有任何影响，理解这一点是很重要的。例如，下面的程序定义了两个Box对象：

```
// This program declares two Box objects.

class Box {
  double width;
  double height;
```

```
  double depth;
}

class BoxDemo2 {
  public static void main(String args[]) {
    Box mybox1 = new Box();
    Box mybox2 = new Box();
    double vol;

    // assign values to mybox1's instance variables
    mybox1.width = 10;
    mybox1.height = 20;
    mybox1.depth = 15;

    /* assign different values to mybox2's
       instance variables */
    mybox2.width = 3;
    mybox2.height = 6;
    mybox2.depth = 9;

    // compute volume of first box
    vol = mybox1.width * mybox1.height * mybox1.depth;
    System.out.println("Volume is " + vol);

    // compute volume of second box
    vol = mybox2.width * mybox2.height * mybox2.depth;
    System.out.println("Volume is " + vol);
  }
}
```

该程序产生的输出如下所示：

```
Volume is 3000.0
Volume is 162.0
```

你可以看到，mybox1的数据与mybox2的数据完全分离。

6.2　声 明 对 象

正如刚才讲过的，当你创建一个类时，你创建了一种新的数据类型。你可以使用这种类型来声明该种类型的对象。然而，要获得一个类的对象需要两步。第一步，你必须声明该类类型的一个变量，这个变量没有定义一个对象。实际上，它只是一个能够引用对象的简单变量。第二步，该声明要创建一个对象的实际的物理拷贝，并把对于该对象的引用赋给该变量。这是通过使用new运算符实现的。new运算符为对象动态分配（即在运行时分配）内存空间，并返回对它的一个引用。这个引用或多或少的是new分配给对象的内存地址。然后这个引用被存储在该变量中。这样，在Java中，所有的类对象都必须动态分配。让我们详细看一下该过程。

在前面的例子中，用下面的语句来声明一个Box类型的对象：

```
Box mybox = new Box();
```

本例将上面讲到的两步组合到了一起，可以将该语句改写为下面的形式，以便将每一步讲的更清楚：

```
Box mybox; // declare reference to object
mybox = new Box(); // allocate a Box object
```

第一行声明了mybox，把它作为对于Box类型的对象的引用。当本句执行后，mybox 包含的值为null，表示它没有引用对象。这时任何引用mybox的尝试都将导致一个编译错误。第二行创建了一个实际的对象，并把对于它的引用赋给mybox 。现在，你可以把mybox作为Box的对象来使用。但实际上，mybox仅仅保存实际的Box对象的内存地址。这两行语句的效果如图6-1所示。

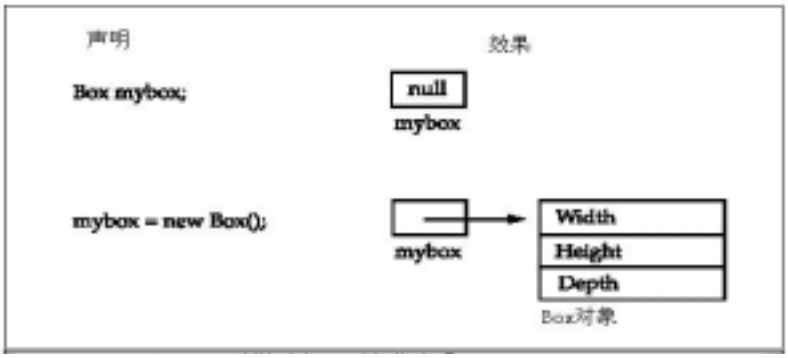


图 6-1 声明 Box 类型的对象

注意：那些熟悉C/C++语言的读者，可能已经注意到了对象的引用看起来和指针类似。这种怀疑实质上是正确的。一个对象引用和内存指针类似。主要的差别（也就是Java安全的关键）是你不能像实际的指针那样来操作它。这样，对于对象引用，你就不能像指针那样任意分配内存地址，或像整数一样操作它。

6.2.1 深入研究new运算符

刚才已经解释过，new运算符动态地为一个对象分配地址。它的通用格式如下：

```
class-var = new classname( );
```

其中，class-var 是所创建类类型的变量。classname 是被实例化的类的名字。类的后面跟的圆括号指定了类的构造函数。构造函数定义当创建一个类的对象时将发生什么。构造函数是所有类的重要组成部分，并有许多重要的属性。大多数类在他们自己的内部显式地定义构造函数。如果一个类没有显式的定义它自己的构造函数，那么Java将自动地提供一个默认的构造函数。对类Box的定义就是这种情况。现在，我们将使用默认的构造函数。不久，你将看到如何定义自己的构造函数。

这时，你可能想知道为什么对整数或字符这样的简单变量不使用new运算符。答案是Java的简单类型不是作为对象实现的。出于效率的考虑，它们是作为“常规”变量实现的。你将看到，对象有许多特性和属性，使Java对对象的处理不同于简单类型。由于对处理对

象和处理简单类型的开销不同，Java能更高效地实现简单类型。后面，你将看见，对于那些需要完全对象类型的情况下，简单类型的对象版本也是可用的。

理解new运算符是在运行期间为对象分配内存的是很重要的。这样做的好处是你的程序在运行期间可以创建它所需要的内存。但是，内存是有限的，因此new有可能由于内存不足而无法给一个对象分配内存。如果出现这种情况，就会发生运行时异常（你将在第10章学习如何处理这种异常以及其他异常情况）。对于本书中的示例程序，你不必担心内存不足的情况，但是在实际的编程中你必须考虑这种可能性。

让我们再次复习类和对象之间的区别。类创建一种新的数据类型，该种类型能被用来创建对象。也就是，类创建了一个逻辑的框架，该框架定义了它的成员之间的关系。当你声明类的对象时，你正在创造该类的实例。因此，类是一个逻辑构造，对象有物理的真实性（也就是对象占用内存空间）。弄清楚这个区别是很重要的。

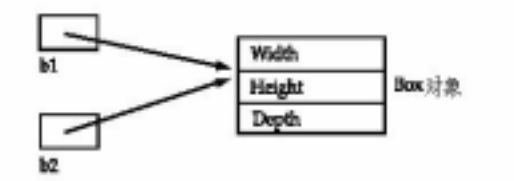
6.3　给对象引用变量赋值

对象变量的赋值和你直觉期望的不同。例如，你认为下面的程序段是做什么呢？

```
Box b1 = new Box();
Box b2 = b1;
```

你可能认为，变量b2被赋值为变量b1对象引用的一个拷贝。也就是，你可能认为b1和b2引用的是不同的对象，但实际情况却相反，b1和b2将引用同样的对象。将b1赋值给b2并没有分配任何内存或对原对象做任何部分的拷贝。由于它们是同一个对象，因此通过变量b2对对象的改变也将影响b1所对应的对象。

这种情况描绘如下：



尽管b1和b2都引用同一个对象，但是他们之间没有任何其他的关系。例如，接下来对b1的赋值仅仅使b1脱离（unhook）初始对象，而没有影响对象或影响b2。

```
Box b1 = new Box();
Box b2 = b1;
// ...
b1 = null;
```

这里，b1被设置为空，但是b2仍然指向原来的对象。

注意：当你将一个对象引用赋值给另一个对象引用时，你并没有创建该对象的一个拷贝，而是仅仅对引用的一个拷贝。

6.4 方 法

在本章的开始提到，类通常由两个要素组成:实例变量和方法。方法是个很大的话题，因为Java给他们如此大的功能和灵活性。事实上，下一章的大部分都用来介绍方法。然而，你现在需要学习一些基础以便你能开始把方法加到你的类中。

这是方法一般的形式：

```
type name(parameter-list) {
  // body of method
}
```

其中，type指定了方法返回的数据类型。这可以是任何合法有效的类型，包括你创建的类的类型。如果该方法不返回任何值，则它的返回值type必须为void 。方法名由name指定。除了被当前作用域中的其他项使用的标识符以外，方法名可以是任何合法的标识符。parameter-list （自变量列表）是一系列类型和标识符对，用逗号分开。自变量本质上是变量，它接收方法被调用时传递给方法的参数值。如果方法没有自变量，那么自变量列表就为空。

对于不返回void类型的方法，使用下面格式的return语句，方法返回值到它的调用程序：

```
return value;
```

其中，value是返回的值。

接下来，你将看到怎样创建多种类型的方法，包括带参数的和那些有返回值的方法。

6.4.1 为Box类添加一个方法

尽管创建一个仅包含数据的类是相当不错的事情，但这样的情况很少发生。大部分情况是你将使用方法存取由类定义的实例变量。事实上，方法定义大多数类的接口。这允许类实现函数可以把内部数据结构的特定布局隐蔽到方法抽象后面。除了定义提供数据的存取的方法，你也可以定义被类的内部自己所使用的方法。

让我们由对Box类增加一个方法开始。回顾一下前面计算盒子体积的例子，你会发现用Box类有时会比使用BoxDemo类能更好地处理这个问题。不管怎么说，一个盒子的体积依赖于盒子的大小，这就是我们想到用Box类来计算盒子的体积。为了做到这一点，你必须对Box类增加一个方法，示例如下：

```
// This program includes a method inside the box class.

class Box {
  double width;
  double height;
  double depth;

  // display volume of a box
  void volume() {
    System.out.print("Volume is ");
```

```
    System.out.println(width * height * depth);
  }
}

class BoxDemo3 {
  public static void main(String args[]) {
    Box mybox1 = new Box();
    Box mybox2 = new Box();

    // assign values to mybox1's instance variables
    mybox1.width = 10;
    mybox1.height = 20;
    mybox1.depth = 15;

    /* assign different values to mybox2's
       instance variables */
    mybox2.width = 3;
    mybox2.height = 6;
    mybox2.depth = 9;

    // display volume of first box
    mybox1.volume();

    // display volume of second box
    mybox2.volume();
  }
}
```

该程序产生的输出如下，与先前版本程序的输出一样。

```
Volume is 3000.0
Volume is 162.0
```

注意看下面两行程序：

```
mybox1.volume ();
mybox2.volume ();
```

该例的第一行调用mybox1的volume()方法。也就是，它使用对象名加点号运算符调用mybox1对象的volume()方法。这样，调用mybox1.volume( )显示mybox1 定义的盒子的体积，调用mybox2.volume（）将显示mybox2 定义的盒子的体积。每次调用volume()，它都会显示指定对象的体积。

如果你对方法调用的概念比较陌生，下列的讨论将有助于澄清该概念。当mybox1.volume ( )被执行时，Java运行系统将程序控制转移到volume( )定义内的代码。当volume( )内的语句执行后，程序控制返回调用者，然后执行程序调用的下一行语句。Java执行方法的过程类似于子程序的运行。

在volume()方法中有一些需要注意的地方：实例变量width，height和depth被直接引用，并没有在它们前面加对象名或点号运算符。当一个方法使用由它的类定义的实例变量时，它可以直接这样做，而不必使用显式的对象引用和使用点号运算符。这是很容易理解的。一个方法总是被它的类的对象调用。只要这个调用过程一发生，对象就是可见的。因此，在方法中就没有必要二次指定对象了。这意味着，volume()中的width，height和depth已经

隐含地引用了调用volume()方法中的这些变量的拷贝。

让我们复习一下:当一个实例变量不是被该实例变量所在类的部分代码访问时，它必须通过该对象加点运算符来访问。但是当一个实例变量被定义该变量的类的代码访问时，该变量可以被直接引用。同样的规则也适用于方法。

6.4.2 返回值

执行volume()方法确实将计算盒子体积的值返回到Box类，但这并不是最好的方法。例如，你的程序的其他部分如何知道一个盒子的体积，而不显示它的值？一个更好地实现volume()的方法是将它计算的盒子体积的结果返回给它的调用者。下面的例子是对前面程序的改进，它正是这样做的：

```
// Now, volume() returns the volume of a box.

class Box {
  double width;
  double height;
  double depth;

  // compute and return volume
  double volume() {
return width * height * depth;
  }
}

class BoxDemo4 {
  public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;

// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;

/* assign different values to mybox2's
   instance variables */
mybox2.width = 3;
mybox2.height = 6;
mybox2.depth = 9;

// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);

// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
  }
}
```

在这个程序中，当volume()被调用时，它被放在赋值语句的右边。左边是接收volume()返回值的变量。因此，当下面的语句执行后，

```
vol = mybox1.volume();
```

变量mybox1.volume ( )的值是 3，000，且该值被保存在vol中。

对于返回值的理解，要注意下面两件重要的事情：

- 方法返回的数据类型必须与该方法指定的返回类型相兼容。例如，如果一个方法的返回值是布尔型，就不可能返回整数。
- 接收方法返回值的变量 (例如本例中的变量 vol)也必须与指定方法返回值的类型相兼容。

另外一点：因为实际上不需要vol变量，前面的程序可以被写得更高效一些。对volume( )方法的调用可以直接用在 println ( )语句中，如下面的语句：

```
System.out.println("Volume is " + mybox1.volume());
```

在本例中，当println ( )被执行时，mybox1.volume ( )将自动地被调用，而且它的值会被传递给println ( )。

6.4.3　加入带自变量的方法

大多数方法不需要自变量。自变量对方法没有特殊要求。也就是说，带自变量的方法，可以完成各种数据操作，它还可以用在很多有微妙差别的情况。为了说明这一点，让我们举一个非常简单的例子。下面的方法返回数字10的平方：

```
int square()
{
  return 10 * 10;
}
```

运行该方法，确实返回了10 的平方的值，但它的使用是很有限的。然而，如下所示，如果你修改该方法，以便它带一个自变量，这样square( )就更有用了。

```
int square(int i)
{
  return i * i;
}
```

现在，square()可以返回任何调用它的值的平方。也就是说，square()现在是可以计算任何整数值的平方的一个通用方法，而不单纯是数字10 。

下面是一个例子：

```
int x, y;
x = square(5); // x equals 25
x = square(9); // x equals 81
y = 2;
x = square(y); // x equals 4
```

在第一次调用square( )时，值5被传递给自变量i。在第二次调用时，i接收到值9。第三次调用时，将值传递给y，在本例中是 2 。如这些例子所示，square ( )可以返回传递给它的任何数据的平方。

区分自变量（parameter）和参数（argument）这两个术语是很重要的。自变量是方法定义的一个变量，当方法被调用时，它接收一个值。例如在square()中，i就是一个自变量。参数是当一个方法被调用时，传递给该方法的值。例如，square(100)把100作为参数传递。在square()中，自变量i接收该值。

可以使用一个带自变量的方法来改进Box类。在前面的例子中，每个盒子的尺寸不得不用单独的语句顺序的来设置，例如：

```
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
```

本例中的代码在执行时，它在两个方面比较麻烦。首先，它笨拙且容易发生错误。例如，很容易忘记设置其中的一个尺寸。其次，在设计的很好的Java程序中，实例变量应该仅仅由定义类的方法来存取。在后面，你可以改变一个方法的行为，但是你不能改变一个暴露的实例变量的行为。

这样，设置一个盒子尺寸的更好的途径是创建一个自变量代表盒子尺寸的方法，而且适当地设置每个实例变量。下面的例子实现了这个想法。

```
// This program uses a parameterized method.

class Box {
  double width;
  double height;
  double depth;

  // compute and return volume
  double volume() {
    return width * height * depth;
  }

  // sets dimensions of box
  void setDim(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }
}

class BoxDemo5 {
  public static void main(String args[]) {
    Box mybox1 = new Box();
    Box mybox2 = new Box();
    double vol;

    // initialize each box
    mybox1.setDim(10, 20, 15);
    mybox2.setDim(3, 6, 9);
```

```
    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume is " + vol);

    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume is " + vol);
  }
}
```

正如你看到的，setDim方法用来设置每个盒子的尺寸，例如，当下面的语句执行后：

```
mybox1.setDim(10, 20, 15);
```

10被拷贝进参数w，20被拷贝进h，15被拷贝进d。在setDim( )内的w、h、d的值分别赋给width、height和depth。

许多读者，特别是那些有C/C++经验的读者，对前面章节中的概念会比较熟悉。但是，如果像方法调用、参数、自变量这些概念对你来说比较新的话，在继续学习以前，你要花些时间来练习。方法调用，自变量，返回值这些概念是Java编程的基础。

6.5 构 造 函 数

每次在创建实例变量，对类中的所有变量都要初始化是很乏味的。即使你对setDim ( )这样的方法增加有用的功能时，你也不得不这样做。如果在一个对象最初被创建时就把对它的设置做好，那样的话，程序将更简单并且更简明。因为对初始化的要求是共同的，Java允许对象在他们被创造时初始化自己。这种自动的初始化是通过使用构造函数来完成的。

构造函数（constructor）在对象创建时初始化。它与它的类同名，它的语法与方法类似。一旦定义了构造函数，在对象创建后，在new运算符完成前，构造函数立即自动调用。构造函数看起来有点奇怪，因为它没有任何返回值，即使是void型的值也不返回。这是因为一个类的构造函数内隐藏的类型是它自己类的类型。构造函数的任务就是初始化一个对象的内部状态，以便使创建的实例变量能够完全初始化，可以被对象马上使用。

你可以重写Box例子程序，以便当对象创建时盒子的尺寸能被自动地初始化。为了达到这个目的，用构造函数代替setDim。让我们由定义仅仅将每个盒子的尺寸设置为同样值的一个简单的构造函数开始。示例如下：

```
/* Here, Box uses a constructor to initialize the
   dimensions of a box.
*/
class Box {
  double width;
  double height;
  double depth;

  // This is the constructor for Box.
  Box() {
    System.out.println("Constructing Box");
```

```
    width = 10;
    height = 10;
    depth = 10;
  }

  // compute and return volume
  double volume() {
    return width * height * depth;
  }
}

class BoxDemo6 {
  public static void main(String args[]) {
    // declare, allocate, and initialize Box objects
    Box mybox1 = new Box();
    Box mybox2 = new Box();

    double vol;

    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume is " + vol);

    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume is " + vol);
  }
}
```

运行该程序，产生如下的结果：

```
Constructing Box
Constructing Box
Volume is 1000.0
Volume is 1000.0
```

正如你能看到的一样，当mybox1和mybox2被创建时，它们两个都被Box构造函数初始化。因为构造函数将所有的盒子赋为一样的尺寸，长、宽、高都是10，mybox1 和 mybox2 将有一样的体积。在Box( )内的println( )语句仅仅是为说明的缘故。大多数构造函数的功能不显示任何东西，他们仅简单地初始化一个对象。

在继续学习前，让我们再考察new运算符。你已经知道，当分配一个对象时，使用下面的通用格式：

```
class-var = new classname( );
```

现在你可以理解为什么在类的名字后面需要圆括号。圆括号的作用是调用该类的构造函数。这样，在下面的这行中

```
Box mybox1 = new Box();
```

new Box( )调用Box( )构造函数。如果你不显式为类定义一个构造函数，Java将为该类创建一个默认的构造函数。这就是本行程序在Box早期版本没有定义构造函数工作的原因。

默认构造函数自动地将所有的实例变量初始化为零。默认构造函数对简单的类是足够的，但是对更复杂的类它就不能满足要求了。一旦你定义了你自己的构造函数，默认构造函数将不再被使用。

6.5.1　带自变量的构造函数

虽然在前面的例子中，Box构造函数确实初始化了Box对象，但它不是很有用，因为所有的盒子都是一样的尺寸。我们所需要的是一种能够构造各种各样尺寸盒子对象的方法。比较容易的解决办法是对构造函数增加自变量。你可能已经猜到，这将使他们更有用。例如，下面版本的Box程序定义了一个自变量构造函数，它根据自变量设置每个指定盒子的尺寸。特别注意Box对象是如何被创建的。

```
/* Here, Box uses a parameterized constructor to
   initialize the dimensions of a box.
*/
class Box {
  double width;
  double height;
  double depth;

  // This is the constructor for Box.
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  // compute and return volume
  double volume() {
    return width * height * depth;
  }
}

class BoxDemo7 {
  public static void main(String args[]) {
    // declare, allocate, and initialize Box objects
    Box mybox1 = new Box(10, 20, 15);
    Box mybox2 = new Box(3, 6, 9);

    double vol;

    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume is " + vol);

    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume is " + vol);
  }
}
```

该程序的输出如下：

```
Volume is 3000.0
Volume is 162.0
```

正如你看到的，每个对象被它的构造函数指定的参数初始化。例如，在下行中，

```
Box mybox1 = new Box(10, 20, 15);
```

当new创建对象时，值10，20，15传递到Box()构造函数。这样，mybox1 的拷贝width、height、depth将分别包含值10、20、15。

6.6 this关键字

有时一个方法需要引用调用它的对象。为此，Java定义了this这个关键字。this可以在引用当前对象的所有方法内使用。也就是，this总是调用该方法对象的一个引用。你可以在当前类的类型所允许对象的任何地方将this作为一个引用。

为了更好理解this引用什么，考虑下面版本的Box()：

```
// A redundant use of this.
Box(double w, double h, double d) {
  this.width = w;
  this.height = h;
  this.depth = d;
}
```

本例中的box( )和它的更早版本完成同样的操作。使用this是冗余的，但是完全正确。在Box( )内，this总是引用调用的对象。虽然在本例中它是冗余的，但在另外的环境中，它是有用的，其中的一种用法在下一小节解释。

6.6.1 隐藏的实例变量

你知道，在同一个范围或一个封装范围内，定义二个重名的局部变量在Java中是不合法的。有趣的是，局部变量，包括传递到方法的正式的自变量，可以与类的实例变量的名字重叠。在这种情况下，局部变量名就隐藏（hide）了实例变量名。这就是在Box类中，width、height、depth没有作为Box()构造函数自变量名字的原因。如果它们是，那么width将正式的引用自变量，而隐蔽实例变量width。由于通常简单地使用不同的名字更容易，对这种状况还有其他的解决办法。因为this可以使你直接引用对象，你能用它来解决可能在实例变量和局部变量之间发生的任何同名的冲突。例如，下面的例子是另外一个版本的Box（）程序，它用width、height、depth作为自变量的名字，然后使用this关键字来存取同名的实例变量：

```
// Use this to resolve name-space collisions.
Box(double width, double height, double depth) {
  this.width = width;
  this.height = height;
  this.depth = depth;
}
```

注意，在这样的环境下使用this有时会引起混淆。有些程序员比较小心，不使用和局部

变量、正式的自变量同名的隐藏的实例变量。当然，另外的程序员则相反，相信用this来“揭开”与局部变量、自变量同名的实例变量是一个好习惯。这取决于你的爱好。

尽管在上面的例子中，this没有有什么意义，但它在某种状况下是很有用的。

6.7　垃 圾 回 收

由于使用new运算符来为对象动态地分配内存，你可能想知道这些对象是如何撤消的以及他们的内存在以后的重新分配时是如何被释放的。在一些语言，例如C++中，用delete运算符来手工地释放动态分配的对象的内存。Java使用一种不同的、自动地处理重新分配内存的办法：垃圾回收(garbage collection)技术，它是这样工作的：当一个对象的引用不存在时，则该对象被认为是不再需要的，它所占用的内存就被释放掉。它不像C++那样需要显式撤消对象。垃圾回收只在你的程序执行过程中偶尔发生。它不会因为一个或几个存在的对象不再被使用而发生。况且，Java不同的运行时刻会产生各种不同的垃圾回收办法，但是对你编写的大多数程序，你不必须考虑垃圾回收问题。

6.8　finalize( )方法

有时当撤消一个对象时，需要完成一些操作。例如，如果一个对象正在处理的是非Java资源，如文件句柄或window字符字体，这时你要确认在一个对象被撤消以前要保证这些资源被释放。为处理这样的状况，Java提供了被称为收尾（finalization）的机制。使用该机制你可以定义一些特殊的操作，这些操作在一个对象将要被垃圾回收程序释放时执行。

要给一个类增加收尾（finalizer），你只要定义finalize ( )方法即可。Java回收该类的一个对象时，就会调用这个方法。在finalize ( )方法中，你要指定在一个对象被撤消前必须执行的操作。垃圾回收周期性地运行，检查对象不再被运行状态引用或间接地通过其他对象引用。就在对象被释放之前，Java运行系统调用该对象的finalize( )方法。

finalize()方法的通用格式如下：

```
protected void finalize( )
{
// finalization code here
}
```

其中，关键字protected是防止在该类之外定义的代码访问finalize()标识符。该标识符和其他标识符将在第7章中解释。

理解finalize( )正好在垃圾回收以前被调用非常重要。例如当一个对象超出了它的作用域时，finalize( )并不被调用。这意味着你不可能知道何时——甚至是否——finalize( )被调用。因此，你的程序应该提供其他的方法来释放由对象使用的系统资源，而不能依靠finalize( )来完成程序的正常操作。

注意：如果你熟悉C++，那你知道C++允许你为一个类定义一个撤消函数（destructor），它在对象正好出作用域之前被调用。Java不支持这个想法也不

提供撤消函数。finalize( )方法只和撤消函数的功能接近。当你对Java有丰富经验时，你将看到因为Java使用垃圾回收子系统，几乎没有必要使用撤消函数。

6.9 一个堆栈类

尽管Box类在说明一个类的必要的元素时是有用的，但它实际应用的价值并不大。为了显示出类的真实的功能，本章将用一个更复杂的例子来说明类的强大功能。如果你回忆起在第2章中讲过的面向对象编程的讨论，你就会想起对象编程的最重要的好处之一是对数据和操作该数据的代码的封装。你已经知道，在Java中，就是通过类这样的机制来完成封装性。在创建一个类时，你正在创建一种新的数据类型，不但要定义数据的属性，也要定义操作数据的代码。进一步，方法定义了对该类数据相一致的控制接口。因此，你可以通过类的方法来使用类，而没有必要担心它的实现细节或在类的内部数据实际上是如何被管理的。在某种意义上，一个类像“一台数据引擎”。你可以通过操纵杆来控制使用引擎，而不需要知道引擎内是如何工作的。事实上，既然细节被隐蔽，当需要时，它的内部工作可以被改变。只要你的代码通过类的方法来使用它，内部的细节可以改变而不会对类的外部带来负面影响。

为了看看前面讨论概念的一个实际的应用，让我们开发一个封装的典型例子：堆栈（stack）。堆栈用先进后出的顺序存储数据。堆栈通过两个传统的操作来控制：压栈（push）和出栈（pop）。在堆栈的上面加入一项，用压栈，从堆栈中取出一项，用出栈。你将看到，将整个堆栈机制封装是很容易的。

下面是一个叫做Stack的类，实现整数的堆栈。

```
// This class defines an integer stack that can hold 10 values.
class Stack {
  int stck[] = new int[10];
  int tos;

  // Initialize top-of-stack
  Stack() {
    tos = -1;
  }

  // Push an item onto the stack
  void push(int item) {
    if(tos==9)
      System.out.println("Stack is full.");
    else
      stck[++tos] = item;
  }

  // Pop an item from the stack
  int pop() {
    if(tos < 0) {
      System.out.println("Stack underflow.");
      return 0;
    }
```

```
    else
      return stck[tos--];
  }
}
```

正如你看到的，Stack类定义了两个数据项、三个方法。整数堆栈由数组stck存储。该数组的下标由变量tos控制，该变量总是包含堆栈顶层的下标。Stack()构造函数将tos初始化为-1，它指向一个空堆栈。方法push()将一个项目压入堆栈。为了重新取回压入堆栈的项目，调用pop()。既然存取数据通过push()和pop()，数组中存储堆栈的事实实际上和使用的堆栈不相关。例如，堆栈可以被存储在一个更复杂的数据结构中，例如一个链表，但push()和pop()定义的接口仍然是一样的。

下面示例的类TestStack，验证了Stack类。该类产生两个整数堆栈，将一些值存入，然后将它们取出。

```
class TestStack {
  public static void main(String args[]) {
    Stack mystack1 = new Stack();
    Stack mystack2 = new Stack();

    // push some numbers onto the stack
    for(int i=0; i<10; i++) mystack1.push(i);
    for(int i=10; i<20; i++) mystack2.push(i);

    // pop those numbers off the stack
    System.out.println("Stack in mystack1:");
    for(int i=0; i<10; i++)
      System.out.println(mystack1.pop());

    System.out.println("Stack in mystack2:");
    for(int i=0; i<10; i++)
      System.out.println(mystack2.pop());
  }
}
```

该程序产生的输出如下：

```
Stack in mystack1:
9
8
7
6
5
4
3
2
1
0
Stack in mystack2:
19
18
17
16
```

```
15
14
13
12
11
10
```

你已经看到，每个堆栈中的内容是分离的。

关于Stack类的最后一点。正如它现在执行的一样，通过Stack类外面的代码可以改变保存堆栈的数组stck。这样的Stack是开放的，容易误用或损坏。在下一章中，你将会看到如何补救这种情况。

第 7 章　进一步研究方法和类

在本章中我们接着上一章继续研究方法和类。我们先研究几个有关方法的主题，包括方法重载、参数传递和递归。然后研究类，讨论存取控制，关键字static的用法，以及Java最重要的内置类之一：String。

7.1　方 法 重 载

在Java中，同一个类中的2个或2个以上的方法可以有同一个名字，只要它们的参数声明不同即可。在这种情况下，该方法就被称为重载（overloaded），这个过程称为方法重载（method overloading）。方法重载是Java实现多态性的一种方式。如果你以前从来没有使用过一种允许方法重载的语言，这个概念最初可能有点奇怪。但是你将看到，方法重载是Java最激动人心和最有用的特性之一。

当一个重载方法被调用时，Java用参数的类型和（或）数量来表明实际调用的重载方法的版本。因此，每个重载方法的参数的类型和（或）数量必须是不同的。虽然每个重载方法可以有不同的返回类型，但返回类型并不足以区分所使用的是哪个方法。当Java调用一个重载方法时，参数与调用参数匹配的方法被执行。

下面是一个说明方法重载的简单例子：

```
// Demonstrate method overloading.
class OverloadDemo {
  void test() {
    System.out.println("No parameters");
  }

  // Overload test for one integer parameter.
  void test(int a) {
    System.out.println("a: " + a);
  }

  // Overload test for two integer parameters.
  void test(int a, int b) {
    System.out.println("a and b: " + a + " " + b);
  }

  // overload test for a double parameter
  double test(double a) {
    System.out.println("double a: " + a);
    return a*a;
  }
}

class Overload {
```

```
  public static void main(String args[]) {
    OverloadDemo ob = new OverloadDemo();
    double result;

    // call all versions of test()
    ob.test();
    ob.test(10);
    ob.test(10, 20);
    result = ob.test(123.25);
    System.out.println("Result of ob.test(123.25): " + result);
  }
}
```

该程序产生如下输出：

```
No parameters
a: 10
a and b: 10 20
double a: 123.25
Result of ob.test(123.25): 15190.5625
```

从上述程序可见，test()被重载了四次。第一个版本没有参数，第二个版本有一个整型参数，第三个版本有两个整型参数，第四个版本有一个double型参数。由于重载不受方法的返回类型的影响，test()第四个版本也返回了一个和重载没有因果关系的值。

当一个重载的方法被调用时，Java在调用方法的参数和方法的自变量之间寻找匹配。但是，这种匹配并不总是精确的。在一些情况下，Java的自动类型转换也适用于重载方法的自变量。例如，看下面的程序：

```
// Automatic type conversions apply to overloading.
class OverloadDemo {
  void test() {
    System.out.println("No parameters");
  }

  // Overload test for two integer parameters.
  void test(int a, int b) {
    System.out.println("a and b: " + a + " " + b);
  }

  // overload test for a double parameter
  void test(double a) {
    System.out.println("Inside test(double) a: " + a);
  }
}

class Overload {
  public static void main(String args[]) {
    OverloadDemo ob = new OverloadDemo();
    int i = 88;

    ob.test();
    ob.test(10, 20);
```

```
    ob.test(i); // this will invoke test(double)
    ob.test(123.2); // this will invoke test(double)
  }
}
```

该程序产生如下输出：

```
No parameters
a and b: 10 20
Inside test(double) a: 88
Inside test(double) a: 123.2
```

在本例中，OverloadDemo 的这个版本没有定义test(int)。因此当在Overload内带整数参数调用test()时，找不到和它匹配的方法。但是，Java可以自动地将整数转换为double型，这种转换就可以解决这个问题。因此，在test(int)找不到以后，Java将i扩大到double型，然后调用test(double)。当然，如果定义了test(int)，当然先调用test(int)而不会调用test(double)。只有在找不到精确匹配时，Java的自动转换才会起作用。

方法重载支持多态性，因为它是Java实现 “一个接口，多个方法”范型的一种方式。要理解这一点，考虑下面这段话：在不支持方法重载的语言中，每个方法必须有一个惟一的名字。但是，你经常希望实现数据类型不同但本质上相同的方法。可以参考绝对值函数的例子。在不支持重载的语言中，通常会含有这个函数的三个及三个以上的版本，每个版本都有一个差别甚微的名字。例如，在C语言中 ，函数abs( )返回整数的绝对值，labs( )返回long型整数的绝对值( )，而fabs( )返回浮点值的绝对值。尽管这三个函数的功能实质上是一样的，但是因为C语言不支持重载，每个函数都要有它自己的名字。这样就使得概念情况复杂许多。尽管每一个函数潜在的概念是相同的，你仍然不得不记住这三个名字。在Java中就不会发生这种情况，因为所有的绝对值函数可以使用同一个名字。确实,Java的标准的类库包含一个绝对值方法，叫做abs ( )。这个方法被Java的math类重载，用于处理数字类型。Java根据参数类型决定调用的abs()的版本。

重载的价值在于它允许相关的方法可以使用同一个名字来访问。因此，abs这个名字代表了它执行的通用动作（general action）。为特定环境选择正确的指定（specific）版本是编译器要做的事情。作为程序员的你，只需要记住执行的通用操作就行了。通过多态性的应用，几个名字减少为一个。尽管这个例子相当简单，但如果你将这个概念扩展一下，你就会理解重载能够帮助你解决更复杂的问题。

当你重载一个方法时,该方法的每个版本都能够执行你想要的任何动作。没有什么规定要求重载方法之间必须互相关联。但是，从风格上来说，方法重载还是暗示了一种关系。这就是当你能够使用同一个名字重载无关的方法时，你不应该这么做。例如，你可以使用sqr这个名字来创建一种方法，该方法返回一个整数的平方和一个浮点数值的平方根。但是这两种操作在功能上是不同的。按照这种方式应用方法就违背了它的初衷。在实际的编程中，你应该只重载相互之间关系紧密的操作。

7.1.1　构造函数重载

除了重载正常的方法外，构造函数也能够重载。实际上，对于大多数你创建的现实的

类，重载构造函数是很常见的，并不是什么例外。为了理解为什么会这样，让我们回想上一章中举过的Box类例子。下面是最新版本的Box类的例子：

```
class Box {
  double width;
  double height;
  double depth;

  // This is the constructor for Box.
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  // compute and return volume
  double volume() {
    return width * height * depth;
  }
}
```

在本例中，Box()构造函数需要三个自变量，这意味着定义的所有Box对象必须给Box()构造函数传递三个参数。例如，下面的语句在当前情况下是无效的：

```
Box ob = new Box();
```

因为Box( )要求有三个参数，因此如果不带参数的调用它则是一个错误。这会引起一些重要的问题。如果你只想要一个盒子而不在乎 (或知道)它的原始的尺寸该怎么办？或，如果你想用仅仅一个值来初始化一个立方体，而该值可以被用作它的所有的三个尺寸又该怎么办？如果Box类是像现在这样写的，与此类似的其他问题你都没有办法解决，因为你只能带三个参数而没有别的选择权。

幸好，解决这些问题的方案是相当容易的：重载Box构造函数，使它能处理刚才描述的情况。下面程序是Box的一个改进版本，它就是运用对Box构造函数的重载来解决这些问题的：

```
/* Here, Box defines three constructors to initialize
   the dimensions of a box various ways.
*/
class Box {
  double width;
  double height;
  double depth;
  // constructor used when all dimensions specified
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  // constructor used when no dimensions specified
  Box() {
    width = -1;  // use -1 to indicate
```

```
    height = -1; // an uninitialized
    depth = -1;  // box
  }

  // constructor used when cube is created
  Box(double len) {
    width = height = depth = len;
  }

  // compute and return volume
  double volume() {
    return width * height * depth;
  }
}

class OverloadCons {
  public static void main(String args[]) {
    // create boxes using the various constructors
    Box mybox1 = new Box(10, 20, 15);
    Box mybox2 = new Box();
    Box mycube = new Box(7);

    double vol;

    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume of mybox1 is " + vol);

    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume of mybox2 is " + vol);
    // get volume of cube
    vol = mycube.volume();
    System.out.println("Volume of mycube is " + vol);
  }
}
```

该程序产生的输出如下所示：

```
Volume of mybox1 is 3000.0
Volume of mybox2 is -1.0
Volume of mycube is 343.0
```

在本例中，当new执行时，根据指定的自变量调用适当的构造函数。

7.2　把对象作为参数

到目前为止，我们都使用简单类型作为方法的参数。但是，给方法传递对象是正确的，也是常用的。例如，考虑下面的简单程序：

```
// Objects may be passed to methods.
class Test {
  int a, b;
```

```
  Test(int i, int j) {
    a = i;
    b = j;
  }

  // return true if o is equal to the invoking object
  boolean equals(Test o) {
    if(o.a == a && o.b == b) return true;
    else return false;
  }
}

class PassOb {
  public static void main(String args[]) {
    Test ob1 = new Test(100, 22);
    Test ob2 = new Test(100, 22);
    Test ob3 = new Test(-1, -1);

    System.out.println("ob1 == ob2: " + ob1.equals(ob2));

    System.out.println("ob1 == ob3: " + ob1.equals(ob3));
  }
}
```

该程序产生如下输出：

```
ob1 == ob2: true
ob1 == ob3: false
```

在本程序中，在Test中的equals()方法比较两个对象的相等性，并返回比较的结果。也就是，它把调用的对象与被传递的对象作比较。如果它们包含相同的值，则该方法返回值为真，否则返回值为假。注意equals中的自变量o指定Test作为它的类型。尽管Test是程序中创建的类的类型，但是它的使用与Java的内置类型相同。

对象参数的最普通的使用涉及到构造函数。你经常想要构造一个新对象，并且使它的初始状态与一些已经存在的对象一样。为了做到这一点，你必须定义一个构造函数，该构造函数将一个对象作为它的类的一个参数。例如，下面版本的Box允许一个对象初始化另外一个对象：

```
// Here, Box allows one object to initialize another.

class Box {
  double width;
  double height;
  double depth;

  // construct clone of an object
  Box(Box ob) { // pass object to constructor
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
  }
```

```
  // constructor used when all dimensions specified
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  // constructor used when no dimensions specified
  Box() {
    width = -1;  // use -1 to indicate
    height = -1; // an uninitialized
    depth = -1;  // box
  }

  // constructor used when cube is created
  Box(double len) {
    width = height = depth = len;
  }

  // compute and return volume
  double volume() {
    return width * height * depth;
  }
}

class OverloadCons2 {
  public static void main(String args[]) {
    // create boxes using the various constructors
    Box mybox1 = new Box(10, 20, 15);
    Box mybox2 = new Box();
    Box mycube = new Box(7);

    Box myclone = new Box(mybox1);

    double vol;
    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume of mybox1 is " + vol);
    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume of mybox2 is " + vol);

    // get volume of cube
    vol = mycube.volume();
    System.out.println("Volume of cube is " + vol);

    // get volume of clone
    vol = myclone.volume();
    System.out.println("Volume of clone is " + vol);
  }
}
```

在本程序中你能看到，当你开始创建你自己的类的时候，为了方便高效的构造对象，必须为同一构造函数方法提供多种形式。

7.3 参数是如何传递的

总的来说，计算机语言给子程序传递参数的方法有两种。第一种方法是按值传递（call-by-value）。这种方法将一个参数值（value）复制成为子程序的正式参数。这样，对子程序的参数的改变不影响调用它的参数。第二种传递参数的方法是引用调用（call-by-reference）。在这种方法中，参数的引用（而不是参数值）被传递给子程序参数。在子程序中，该引用用来访问调用中指定的实际参数。这样，对子程序参数的改变将会影响调用子程序的参数。你将看到，根据传递的对象不同，Java将使用这两种不同的方法。

在Java中，当你给方法传递一个简单类型时，它是按值传递的。因此，接收参数的子程序参数的改变不会影响到该方法之外。例如，看下面的程序：

```
// Simple types are passed by value.
class Test {
  void meth(int i, int j) {
    i *= 2;
    j /= 2;
  }
}
class CallByValue {
  public static void main(String args[]) {
    Test ob = new Test();

    int a = 15, b = 20;

    System.out.println("a and b before call: " +
                       a + " " + b);

    ob.meth(a, b);

    System.out.println("a and b after call: " +
                       a + " " + b);
  }
}
```

该程序的输出如下所示：

```
a and b before call: 15 20
a and b after call: 15 20
```

可以看出，在meth( )内部发生的操作不影响调用中a和b的值。它们的值没在本例中没有变为30和10。

当你给方法传递一个对象时，这种情形就会发生戏剧性的变化，因为对象是通过引用传递的。记住，当你创建一个类类型的变量时，你仅仅创建了一个类的引用。因此，当你将这个引用传递给一个方法时，接收它的参数将会指向该参数指向的同一个对象。这有力地证明了对象是通过引用调用传递给方法的。该方法中对象的改变确实影响了作为参数的对象。例如，考虑下面的程序：

```
// Objects are passed by reference.

class Test {
  int a, b;

  Test(int i, int j) {
    a = i;
    b = j;
  }
  // pass an object
  void meth(Test o) {
    o.a *=  2;

    o.b /= 2;
  }
}

class CallByRef {
  public static void main(String args[]) {
    Test ob = new Test(15, 20);

    System.out.println("ob.a and ob.b before call: " +
                ob.a + " " + ob.b);

    ob.meth(ob);

    System.out.println("ob.a and ob.b after call: " +
                ob.a + " " + ob.b);
  }
}
```

该程序产生下面的输出：

```
ob.a and ob.b before call: 15 20
ob.a and ob.b after call: 30 10
```

正如你所看到的，在这个例子中，在 meth ( )中的操作影响了作为参数的对象。

有趣的一点是，当一个对象引用被传递给方法时，引用本身使用按值调用被传递。但是，因为被传递的值指向一个对象，该值的拷贝仍然指向它相应的参数所指向的同一个对象。

注意：当一个简单类型传递给一个方法时，使用按值传递。对象传递则按引用传递。

7.4　返 回 对 象

方法能够返回任何类型的数据，包括你创建的类的类型。例如，在下面的程序中，incrByTen()方法返回一个对象，在该对象中的值a比调用对象中的值a大10。

```
// Returning an object.
class Test {
```

```
  int a;

  Test(int i) {
    a = i;
  }

  Test incrByTen() {
    Test temp = new Test(a+10);
    return temp;
  }
}

class RetOb {
  public static void main(String args[]) {
    Test ob1 = new Test(2);
    Test ob2;

    ob2 = ob1.incrByTen();
    System.out.println("ob1.a: " + ob1.a);
    System.out.println("ob2.a: " + ob2.a);

    ob2 = ob2.incrByTen();
    System.out.println("ob2.a after second increase: "
                + ob2.a);
  }
}
```

该程序产生的输出如下所示：

```
ob1.a: 2
ob2.a: 12
ob2.a after second increase: 22
```

正如你看到的，每次调用incrByTen()，就产生一个新对象，同时将它的引用返回到调用子程序。

上面的程序还有另外重要的一点:既然所有的对象用关键字new动态地分配内存，你不必担心一个对象会出范围，因为它被其创建的方法终止。只要你程序中有它的一个引用，该对象将会继续存在。当没有该对象的引用时，在下一次垃圾回收发生时该对象将被回收。

7.5 递 归

Java支持递归（recursion）。递归就是依照自身定义事物的过程。在Java编程中，递归是允许方法调用自身调用的属性。调用自身的方法称为是递归的（recursive）。

递归的典型例子是数字的阶乘。数字N的阶乘是1到N之间所有整数的乘积。例如3的阶乘就是1× 2×3，或者是6。下面的程序使用递归来计算数字的阶乘。

```
// A simple example of recursion.
class Factorial {
  // this is a recursive function
  int fact(int n) {
```

```
    int result;

    if(n==1) return 1;
    result = fact(n-1) * n;
    return result;
  }
}

class Recursion {
  public static void main(String args[]) {
    Factorial f = new Factorial();

    System.out.println("Factorial of 3 is " + f.fact(3));
    System.out.println("Factorial of 4 is " + f.fact(4));
    System.out.println("Factorial of 5 is " + f.fact(5));
  }
}
```

该程序产生的输出如下所示：

```
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120
```

如果你对递归的方法比较陌生，那么fact( )的操作可能看起来似乎有点糊涂。它是这样工作的：当fact( )带着参数1被调用时，该方法返回1；否则它返回fact(n-1)与n的乘积。为了对这个表达式求值，fact()带着参数n-1被调用。重复这个过程直到 n 等于 1，且对该方法的调用开始返回。

为了更好地理解fact( )方法是如何工作的,让我们通过一个短例子来说明。例如当计算 3 的阶乘时，对fact()的第一次调用引起参数2的第二次调用。这个调用将引起fact以参数1的第三次调用，这个调用返回1，这个值接着与2（第二次调用时n的值）相乘。然后该结果（现为2）返回到fact()的最初的调用，并将该结果与3（n的初始值）相乘。这时得到答案，6。如果你在fact()中插入println()语句，显示每次调用的阶数以及中间结果，你会觉得很有意思。

当一个方法调用它自身的时候，堆栈就会给新的局部变量和自变量分配内存，方法代码就带着这些新的变量从头执行。递归调用并不产生方法新的拷贝。只有参数是新的。每当递归调用返回时，旧的局部变量和自变量就从堆栈中清除，运行从方法中的调用点重新开始。递归方法可以说是像“望远镜”一样，可以自由伸缩。

许多子程序的递归版本执行时会比它们的迭代版本要慢一点,因为它们增加了额外的方法调用的消耗。对一个方法太多的递归调用会引起堆栈崩溃。因为自变量和局部变量的存储都在堆栈中，每次调用都创建这些变量新的拷贝，堆栈有可能被耗尽。如果发生这种情况，Java的运行时系统就会产生异常。但是，除非递归子程序疯狂运行，否则你大概不会担心这种情况。

递归的主要优点在于：某些类型的算法采用递归比采用迭代算法要更加清晰和简单。例如快速排序算法按照迭代方法是很难实现的。还有其他一些问题，特别是人工智能问题，就依赖于递归提供解决方案。最后，有些人认为递归要比迭代简单。

当编写递归方法时，你必须使用if条件语句在递归调用不执行时来强制方法返回。如果你不这么做，一旦你调用方法，它将永远不会返回。这类错误在使用递归时是很常见的。尽量多地使用println()语句，使你可以了解程序的进程；如果发现错误，立即中止程序运行。

下面是递归的又一个例子。递归方法 printArray ( )打印数组values中的前i个元素。

```
// Another example that uses recursion.

class RecTest {
  int values[];

  RecTest(int i) {
    values = new int[i];
  }

  // display array - recursively
  void printArray(int i) {
    if(i==0) return;
    else printArray(i-1);
    System.out.println("[" + (i-1) + "] " + values[i-1]);
  }
}

class Recursion2 {
  public static void main(String args[]) {
    RecTest ob = new RecTest(10);
    int i;

    for(i=0; i<10; i++) ob.values[i] = i;

    ob.printArray(10);
  }
}
```

该程序产生如下的输出：

```
[0] 0
[1] 1
[2] 2
[3] 3
[4] 4
[5] 5
[6] 6
[7] 7
[8] 8
[9] 9
```

7.6 介绍访问控制

我们知道，封装将数据和处理数据的代码连接起来。同时，封装也提供另一个重要属性：访问控制（access control）。通过封装你可以控制程序的哪一部分可以访问类的成员。通过控制访问，可以阻止对象的滥用。例如，通过只允许适当定义的一套方法来访问数据，

你能阻止该数据的误用。因此，如果使用得当，可以把类创建一个“黑盒子”，虽然可以使用该类，但是它的内部机制是不公开的，不能修改。但是，本书前面创建的类可能不会完全适合这个目标。例如，考虑在第6章末尾示例的Stack类。方法push( )和pop()确实为堆栈提供一个可控制的接口，这是事实，但这个接口并没被强制执行。也就是说，程序的其他部分可以绕过这些方法而直接存取堆栈，这是可能的。当然，如果使用不当，这可能导致麻烦。本节将介绍能精确控制一个类各种各样成员的访问的机制。

一个成员如何被访问取决于修改它的声明的访问指示符（access specifier）。Java提供一套丰富的访问指示符。存取控制的某些方面主要和继承或包联系在一起（包，package，本质上是一组类）。Java的这些访问控制机制将在以后讨论。现在，让我们从访问控制一个简单的类开始。一旦你理解了访问控制的基本原理，其他部分就比较容易了。

Java的访问指示符有public（公共的，全局的）、private（私有的，局部的）、和protected（受保护的）。Java也定义了一个默认访问级别。指示符protected仅用于继承情况中。下面我们描述其他两个访问指示符。

让我们从定义public和private开始。当一个类成员被public指示符修饰时，该成员可以被你的程序中的任何其他代码访问。当一个类成员被指定为private时，该成员只能被它的类中的其他成员访问。现在你能理解为什么main( )总是被public指示符修饰。它被在程序外面的代码调用，也就是由Java运行系统调用。如果不使用访问指示符，该类成员的默认访问设置为在它自己的包内为public，但是在它的包以外不能被存取（包将在以后的章节中讨论）。

到目前为止，我们开发的类的所有成员都使用了默认访问模式，它实质上是public。然而，这并不是你想要的典型的方式。通常，你想要对类数据成员的访问加以限制，只允许通过方法来访问它。另外，有时你想把一个方法定义为类的一个私有的方法。

访问指示符位于成员类型的其他说明的前面。也就是说，成员声明语句必须以访问指示符开头。下面是一个例子：

```
public int i;
private double j;

private int myMethod(int a, char b) { // ...
```

要理解public和private对访问的作用，看下面的程序：

```
/* This program demonstrates the difference between
   public and private.
*/
class Test {
  int a; // default access
  public int b; // public access
  private int c; // private access

  // methods to access c
  void setc(int i) { // set c's value
    c = i;
  }
  int getc() { // get c's value
```

```
    return c;
  }
}

class AccessTest {
  public static void main(String args[]) {
    Test ob = new Test();

    // These are OK, a and b may be accessed directly
    ob.a = 10;
    ob.b = 20;

    // This is not OK and will cause an error
//  ob.c = 100; // Error!

    // You must access c through its methods
    ob.setc(100); // OK
    System.out.println("a, b, and c: " + ob.a + " " +
                ob.b + " " + ob.getc());
  }
}
```

可以看出，在Test类中，a使用默认访问指示符，在本例中与public相同。b被显式地指定为public。成员c被指定为private，因此它不能被它的类之外的代码访问。所以，在AccessTest类中不能直接使用c。对它的访问只能通过它的public方法：setc()和getc()。如果你将下面语句开头的注释符号去掉，

```
//  ob.c = 100; // Error!
```

则由于违规，你不能编译这个程序。

为了理解访问控制在实际中的应用，我们来看在第6章末尾所示的Stack类的改进版本。

```
// This class defines an integer stack that can hold 10 values.
class Stack {
  /* Now, both stck and tos are private. This means
    that they cannot be accidentally or maliciously
    altered in a way that would be harmful to the stack.
  */
  private int stck[] = new int[10];
  private int tos;

  // Initialize top-of-stack
  Stack() {
    tos = -1;
  }

  // Push an item onto the stack
  void push(int item) {
    if(tos==9)
      System.out.println("Stack is full.");
    else
      stck[++tos] = item;
  }
  // Pop an item from the stack
```

```
  int pop() {
    if(tos < 0) {
      System.out.println("Stack underflow.");
      return 0;
    }
    else
      return stck[tos--];
  }
}
```

在本例中，现在存储堆栈的stck和指向堆栈顶部的下标tos，都被指定为private。这意味着除了通过push()或pop()，它们不能够被访问或改变。例如，将tos指定为private，阻止你程序的其他部分无意中将它的值设置为超过stck 数组下标界的值。

下面的程序表明了改进的Stack类。试着删去注释前面的线条来证明stck和tos成员确实是不能访问的。

```
class TestStack {
  public static void main(String args[]) {
    Stack mystack1 = new Stack();
    Stack mystack2 = new Stack();

    // push some numbers onto the stack
    for(int i=0; i<10; i++) mystack1.push(i);
    for(int i=10; i<20; i++) mystack2.push(i);

    // pop those numbers off the stack
    System.out.println("Stack in mystack1:");
    for(int i=0; i<10; i++)
      System.out.println(mystack1.pop());

    System.out.println("Stack in mystack2:");
    for(int i=0; i<10; i++)
      System.out.println(mystack2.pop());

    // these statements are not legal
    // mystack1.tos = -2;
    // mystack2.stck[3] = 100;
  }
}
```

尽管由类定义的方法通常提供对数据的访问，但情况并不总是这样。当需要时允许一个实例变量为public是完全合适的。例如,为简单起见，本书中大多数的简单类在创建时不关心实例变量的存取。然而，在大多数实际应用的类中，你将有必要仅仅允许通过方法来对数据操作。下一章将回到访问控制的话题。你将看到，在继承中访问控制是至关重要的。

7.7　理解static

有时你希望定义一个类成员，使它的使用完全独立于该类的任何对象。通常情况下，类成员必须通过它的类的对象访问，但是可以创建这样一个成员，它能够被它自己使用，

而不必引用特定的实例。在成员的声明前面加上关键字static(静态的)就能创建这样的成员。如果一个成员被声明为static，它就能够在它的类的任何对象创建之前被访问，而不必引用任何对象。你可以将方法和变量都声明为static。static成员的最常见的例子是main( )。因为在程序开始执行时必须调用main()，所以它被声明为static。

声明为static的变量实质上就是全局变量。当声明一个对象时，并不产生static变量的拷贝，而是该类所有的实例变量共用同一个static变量。

声明为static的方法有以下几条限制：

- 它们仅能调用其他的static方法。
- 它们只能访问static数据。
- 它们不能以任何方式引用this或super（关键字super与继承有关，在下一章中描述）。

如果你需要通过计算来初始化你的static变量，你可以声明一个static块，Static块仅在该类被加载时执行一次。下面的例子显示的类有一个static方法，一些static变量，以及一个static初始化块：

```
// Demonstrate static variables, methods, and blocks.
class UseStatic {
  static int a = 3;
  static int b;

  static void meth(int x) {
    System.out.println("x = " + x);
    System.out.println("a = " + a);
    System.out.println("b = " + b);
  }

  static {
    System.out.println("Static block initialized.");
    b = a * 4;
  }

  public static void main(String args[]) {
    meth(42);
  }
}
```

一旦UseStatic类被装载，所有的static语句被运行。首先，a被设置为3，接着static块执行(打印一条消息)，最后，b被初始化为a*4或12。然后调用main()，main()调用meth()，把值42传递给x。3个println ( )语句引用两个static变量a和b，以及局部变量x 。

注意：在一个static方法中引用任何实例变量都是非法的。

下面是该程序的输出：

```
Static block initialized.
x = 42
a = 3
b = 12
```

在定义它们的类的外面，static方法和变量能独立于任何对象而被使用。这样，你只要在类的名字后面加点号运算符即可。例如，如果你希望从类外面调用一个static方法，你可以使用下面通用的格式：

```
classname.method( )
```

这里，classname 是类的名字，在该类中定义static方法。可以看到，这种格式与通过对象引用变量调用非static方法的格式类似。一个static变量可以以同样的格式来访问——类名加点号运算符。这就是Java如何实现全局功能和全局变量的一个控制版本。

下面是一个例子。在main()中，static方法callme()和static变量b在它们的类之外被访问。

```
class StaticDemo {
  static int a = 42;
  static int b = 99;
  static void callme() {
    System.out.println("a = " + a);
  }
}

class StaticByName {
  public static void main(String args[]) {
    StaticDemo.callme();
    System.out.println("b = " + StaticDemo.b);
  }
}
```

下面是该程序的输出：

```
a = 42
b = 99
```

7.8　介绍final

一个变量可以声明为final，这样做的目的是阻止它的内容被修改。这意味着在声明final变量的时候，你必须初始化它（在这种用法上，final类似于C/C++中的const）。例如：

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
final int FILE_SAVE = 3;
final int FILE_SAVEAS = 4;
final int FILE_QUIT = 5;
```

你的程序的随后部分现在可以使用 FILE_OPEN等等，就好像它们是常数，不必担心它们的值会被改变。

为final变量的所有的字符选择大写是一个普遍的编码约定。声明为final的变量在实例中不占用内存。这样，一个final变量实质上是一个常数。

关键字final也可以被应用于方法，但是它的意思和它被用于变量实质上是不同的。final的第二种用法将在下一章描述继承时解释。

7.9 重新温习数组

在此之前已经在本书中介绍过数组了。现在既然你已了解了类，可以介绍关于数组的重要的一点：数组是作为对象来实现的。因此，你可能想要利用数组的一种特别的属性，具体地说，就是一个数组的大小——也就是，一个数组能保存的元素的数目——可以在它的length实例变量中找到。所有的数组都有这个变量，并且它总是保存数组的大小。下面的程序示例了这个性质：

```
// This program demonstrates the length array member.
class Length {
  public static void main(String args[]) {
    int a1[] = new int[10];
    int a2[] = {3, 5, 7, 1, 8, 99, 44, -10};
    int a3[] = {4, 3, 2, 1};

    System.out.println("length of a1 is " + a1.length);
    System.out.println("length of a2 is " + a2.length);
    System.out.println("length of a3 is " + a3.length);
  }
}
```

该程序显示如下输出：

```
length of a1 is 10
length of a2 is 8
length of a3 is 4
```

可以看出，每个数组的大小都被显示。要记住length的值和数组实际使用的元素的个数没有关系。length仅反映了数组能够包含的元素的数目。

在许多情况下，你可以好好利用length。例如，下面的程序是Stack类的改进版本。你可能回忆起，该类的早期的版本总是要产生一个10个元素的堆栈。下面的版本可以让你产生任意长度的堆栈。stck.length的值用来防止堆栈溢出。

```
// Improved Stack class that uses the length array member.
class Stack {
  private int stck[];
  private int tos;

  // allocate and initialize stack

  Stack(int size) {
    stck = new int[size];
    tos = -1;
  }

  // Push an item onto the stack
  void push(int item) {
    if(tos==stck.length-1) // use length member
      System.out.println("Stack is full.");
```

```
    else
      stck[++tos] = item;
  }

  // Pop an item from the stack
  int pop() {
    if(tos < 0) {
      System.out.println("Stack underflow.");
      return 0;
    }
    else
      return stck[tos--];
  }
}

class TestStack2 {
  public static void main(String args[]) {
    Stack mystack1 = new Stack(5);
    Stack mystack2 = new Stack(8);
    // push some numbers onto the stack
    for(int i=0; i<5; i++) mystack1.push(i);
    for(int i=0; i<8; i++) mystack2.push(i);

    // pop those numbers off the stack
    System.out.println("Stack in mystack1:");
    for(int i=0; i<5; i++)
      System.out.println(mystack1.pop());

    System.out.println("Stack in mystack2:");
    for(int i=0; i<8; i++)
      System.out.println(mystack2.pop());
  }
}
```

注意，该程序创建了两个堆栈：一个有5个元素，另一个有8个元素。可以看出，数组保持它们自己长度信息的事实使创建任何大小的堆栈很容易。

7.10　介绍嵌套类和内部类

在另一个类中定义的类就是嵌套类（nested classes）。嵌套类的范围由装入它的类的范围限制。这样，如果类B被定义在类A之内，那么B为A所知，然而不被A的外面所知。嵌套类可以访问嵌套它的类的成员，包括private成员。但是，包围类不能访问嵌套类的成员。

嵌套类一般有2种类型：前面加static标识符的和不加static标识符的。一个static的嵌套类有static修饰符。因为它是static，所以只能通过对象来访问它包围类的成员。也就是说，它不能直接引用它包围类的成员。因为有这个限制，所以static嵌套类很少使用。

嵌套类最重要的类型是内部类（inner class）。内部类是非static的嵌套类。它可以访问它的外部类的所有变量和方法，它可以直接引用它们，就像外部类中的其他非static成员的功能一样。这样，一个内部类完全在它的包围类的范围之内。

下面的程序示例了如何定义和使用一个内部类。名为Outer的类有一个名为outer_x的示例变量，一个名为test()的实例方法，并且定义了一个名为Inner的内部类。

```
// Demonstrate an inner class.
class Outer {
  int outer_x = 100;

  void test() {
    Inner inner = new Inner();
    inner.display();
  }

  // this is an inner class
  class Inner {
    void display() {
      System.out.println("display: outer_x = " + outer_x);
    }
  }
}

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

该程序的输出如下所示：

```
display: outer_x = 100
```

在本程序中，内部类Inner定义在Outer类的范围之内。因此，在Inner类之内的任何代码可以直接访问变量outer_x。实例方法display()定义在Inner的内部，该方法以标准的输出流显示 outer_x。InnerClassDemo的main( )方法创建类Outer的一个实例并调用它的test( )方法。创建类Inner和display()方法的一个实例的方法被调用。

认识到Inner类只有在类Outer的范围内才是可知的是很重要的。如果在类Outer之外的任何代码试图实例化Inner类，Java编译器会产生一条错误消息。总体来说，一个嵌套类和其他任何另外的编程元素没有什么不同：它仅仅在它的包围范围内是可知的。

我们解释过，一个内部类可以访问它的包围类的成员，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，而且不能被外部类使用。例如：

```
// This program will not compile.
class Outer {
  int outer_x = 100;

  void test() {
    Inner inner = new Inner();
    inner.display();
  }

  // this is an inner class
```

```
  class Inner {
    int y = 10; // y is local to Inner
    void display() {
      System.out.println("display: outer_x = " + outer_x);
    }
  }

  void showy() {
    System.out.println(y); // error, y not known here!
  }
}

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

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

尽管我们强调嵌套类在它的外部类的范围之内声明，但在几个程序块的范围之内定义内部类是可能的。例如，在由方法定义的块中，或甚至在for循环体内部，你也可以定义嵌套类，如下面的程序所示：

```
// Define an inner class within a for loop.
class Outer {
  int outer_x = 100;

  void test() {
    for(int i=0; i<10; i++) {
      class Inner {
        void display() {
          System.out.println("display: outer_x = " + outer_x);
        }
      }
      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
display: outer_x = 100
display: outer_x = 100
```

```
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
```

尽管嵌套类在日常的大多数编程中不使用，但当处理applet（小应用程序）时是特别有帮助的。在第20章中我们将继续嵌套类的话题。在那里你将看到对于某些类型的事件内部类如何被用来简化代码。你也将了解匿名内部类（anonymous inner classes）,它是一个没有名字的内部类。

最后一点：嵌套类在Java的最初的1.0版本中是不允许的。直到Java 1.1中才添加了嵌套类。

7.11 探索String类

尽管String类将在本书的第2部分深入地研究，但因为我们将在第1部分末尾的一些例子程序中使用字符串，因此，现在应该对它做一个简单的探索。String类是Java类库中最常用的类，其中最明显的原因是字符串在编程语言中是很重要的部分。

有关字符串的最重要一点是，你创建的每一个字符串实际上都是String类型的一个对象，即使是字符串常量实际上也是String对象。

```
System.out.println("This is a String, too");
```

字符串“This is a String，too”是一个字符串常数。幸好，Java处理字符串常数和其他计算机语言处理“正常”的字符串的方法一样，因此你不必担心这个。

字符串的另一个特点是，String类型的对象是不可改变的；一旦创建了一个字符串对象，它的内容是不能被改变的。这看起来是一个严格的限制，但实际上不是，因为这有两个原因：

- 如果你需要改变一个字符串，你可以创建一个新的字符串，其中包含修改后的字符串即可。
- Java定义了一个和String类同等的类叫StringBuffer，它允许字符串改变，因此所有正常的字符串操作在Java中还是可用的（StringBuffer在本书的第2部分描述）。

字符串可以通过多种方法构造。最容易的一种用如下的语句：

```
String myString = "this is a test";
```

一旦你创建了一个字符串对象，你可以在任何允许字符串的地方使用它，例如下面这条语句显示myString ：

```
System.out.println(myString);
```

Java定义了一个String对象的运算符：“+”。它用来连接两个字符串。例如，下面这

条语句：

```
String myString = "I" + " like " + "Java.";
```

的结果是myString包含“I like Java.”

下面的程序表明了前面的概念：

```
// Demonstrating Strings.
class StringDemo {
  public static void main(String args[]) {
    String strOb1 = "First String";
    String strOb2 = "Second String";
    String strOb3 = strOb1 + " and " + strOb2;

    System.out.println(strOb1);
    System.out.println(strOb2);
    System.out.println(strOb3);
  }
}
```

该程序产生的输出如下所示：

```
First String
Second String
First String and Second String
```

String类包含许多操作字符串的方法。例如下面就是其中一些。你可以用equals()来检验两个字符串是否相等。你可以调用方法length()来获得一个字符串的长度。你可以调用charAt()来获得一个字符串指定索引的字符。这三个方法的通用格式如下所示：

```
boolean equals(String object)
int length( )
char charAt(int index)
```

下面的程序示例了这些方法：

```
// Demonstrating some String methods.
class StringDemo2 {
  public static void main(String args[]) {
    String strOb1 = "First String";
    String strOb2 = "Second String";
    String strOb3 = strOb1;

    System.out.println("Length of strOb1: " +
                       strOb1.length());

    System.out.println("Char at index 3 in strOb1: " +
                       strOb1.charAt(3));

    if(strOb1.equals(strOb2))
      System.out.println("strOb1 == strOb2");
    else
      System.out.println("strOb1 != strOb2");
```

```
    if(strOb1.equals(strOb3))
      System.out.println("strOb1 == strOb3");
    else
      System.out.println("strOb1 != strOb3");
  }
}
```

该程序产生如下的输出：

```
Length of strOb1: 12
Char at index 3 in strOb1: s
strOb1 != strOb2
strOb1 == strOb3
```

当然，与其他对象类型一样，strings也可以组成数组，例如：

```
// Demonstrate String arrays.
class StringDemo3 {
  public static void main(String args[]) {
    String str[] = { "one", "two", "three" };

    for(int i=0; i<str.length; i++)
      System.out.println("str[" + i + "]: " +
                         str[i]);
  }
}
```

下面是该程序产生的输出：

```
str[0]: one
str[1]: two
str[2]: three
```

在下节里你将看到，字符串数组在许多Java程序中起重要的作用。

7.12 使用命令行参数

有时你想在运行程序时将信息传递到一个程序中。这通过将命令行参数（command-line arguments）传递给main()来实现。命令行参数是程序执行时在命令行中紧跟在程序名后的信息。在Java程序中访问命令行参数是相当容易的——它们作为字符串存储在传递给main()的String数组中。例如,下面的程序显示了调用的所有的命令行参数：

```
// Display all command-line arguments.
class CommandLine {
  public static void main(String args[]) {
    for(int i=0; i<args.length; i++)
      System.out.println("args[" + i + "]: " +
      args[i]);
  }
}
```

尝试执行这个程序，命令如下所示：

```
java Commandline this is a test 100  -1
```

执行后，你会看到如下输出：

```
args[0]: this
args[1]: is
args[2]: a
args[3]: test
args[4]: 100
args[5]: -1
```

注意：所有的命令行参数都是以字符串的形式传递的。你必须手工把数字值变换到它们的内部形式,这将在第14章解释。

第 8 章　继　承

继承是面向对象编程技术的一块基石，因为它允许创建分等级层次的类。运用继承，你能够创建一个通用类，它定义了一系列相关项目的一般特性。该类可以被更具体的类继承，每个具体的类都增加一些自己特有的东西。在Java术语学中，被继承的类叫超类（superclass），继承超类的类叫子类（subclass）。因此，子类是超类的一个专门用途的版本，它继承了超类定义的所有实例变量和方法，并且为它自己增添了独特的元素。

8.1　继承的基础

继承一个类，只要用extends关键字把一个类的定义合并到另一个中就可以了。为了理解怎样继承，让我们从简短的程序开始。下面的例子创建了一个超类A和一个名为B的子类。注意怎样用关键字extends来创建A的一个子类。

```
// A simple example of inheritance.

// Create a superclass.
class A {
  int i, j;

  void showij() {
    System.out.println("i and j: " + i + " " + j);
  }
}

// Create a subclass by extending class A.
class B extends A {
  int k;

  void showk() {
    System.out.println("k: " + k);
  }
  void sum() {
    System.out.println("i+j+k: " + (i+j+k));
  }
}

class SimpleInheritance {
  public static void main(String args[]) {
    A superOb = new A();
    B subOb = new B();

    // The superclass may be used by itself.
    superOb.i = 10;
    superOb.j = 20;
```

```
    System.out.println("Contents of superOb: ");
    superOb.showij();
    System.out.println();

    /* The subclass has access to all public members of
       its superclass. */
    subOb.i = 7;
    subOb.j = 8;
    subOb.k = 9;
    System.out.println("Contents of subOb: ");
    subOb.showij();
    subOb.showk();
    System.out.println();

    System.out.println("Sum of i, j and k in subOb:");
    subOb.sum();
  }
}
```

该程序的输出如下：

```
Contents of superOb:
i and j: 10 20

Contents of subOb:
i and j: 7 8
k: 9

Sum of i, j and k in subOb:
i+j+k: 24
```

像你所看到的，子类B包括它的超类A中的所有成员。这是为什么subOb可以获取i和j以及调用showij( )方法的原因。同样，sum( )内部，i和j可以被直接引用，就像它们是B的一部分。

尽管A是B的超类，它也是一个完全独立的类。作为一个子类的超类并不意味着超类不能被自己使用。而且，一个子类可以是另一个类的超类。

声明一个继承超类的类的通常形式如下：

```
class subclass-name extends superclass-name {
 // body of class
}
```

你只能给你所创建的每个子类定义一个超类。Java不支持多超类的继承（这与C++不同，在C++中，你可以继承多个基础类）。你可以按照规定创建一个继承的层次。该层次中，一个子类成为另一个子类的超类。然而，没有类可以成为它自己的超类。

8.1.1　成员的访问和继承

尽管子类包括超类的所有成员，它不能访问超类中被声明成private的成员。例如，考虑下面简单的类层次结构：

```
/* In a class hierarchy, private members remain
```

```
   private to their class.

   This program contains an error and will not
   compile.
*/

// Create a superclass.
class A {
  int i; // public by default
  private int j; // private to A

  void setij(int x, int y) {
    i = x;
    j = y;
  }
}

// A's j is not accessible here.
class B extends A {
  int total;
  void sum() {
    total = i + j; // ERROR, j is not accessible here
  }
}

class Access {
  public static void main(String args[]) {
    B subOb = new B();

    subOb.setij(10, 12);

    subOb.sum();
    System.out.println("Total is " + subOb.total);
  }
}
```

该程序不会编译，因为B中sum( )方法内部对j的引用是不合法的。既然j被声明成private，它只能被它自己类中的其他成员访问。子类没权访问它。

注意：一个被定义成private的类成员为此类私有，它不能被该类外的所有代码访问，包括子类。

8.1.2 更实际的例子

让我们看一个更实际的例子，该例子有助于阐述继承的作用。这里，前面章节改进的Box类的最后版本将被扩展。它包括第四成员名为weight。这样，新的类将包含一个盒子的宽度、高度、深度和重量。

```
// This program uses inheritance to extend Box.
class Box {
  double width;
  double height;
  double depth;
```

```
  // construct clone of an object
  Box(Box ob) { // pass object to constructor
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
  }

  // constructor used when all dimensions specified
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  // constructor used when no dimensions specified
  Box() {
    width = -1;  // use -1 to indicate
    height = -1; // an uninitialized
    depth = -1;  // box
  }

  // constructor used when cube is created
  Box(double len) {
    width = height = depth = len;
  }

  // compute and return volume
  double volume() {
    return width * height * depth;
  }
}

// Here, Box is extended to include weight.
class BoxWeight extends Box {
  double weight; // weight of box

  // constructor for BoxWeight
  BoxWeight(double w, double h, double d, double m) {
    width = w;
    height = h;
    depth = d;
    weight = m;
  }
}
class DemoBoxWeight {
  public static void main(String args[]) {
    BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
    BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
    double vol;

    vol = mybox1.volume();
    System.out.println("Volume of mybox1 is " + vol);
    System.out.println("Weight of mybox1 is " + mybox1.weight);
    System.out.println();
```

```
    vol = mybox2.volume();
    System.out.println("Volume of mybox2 is " + vol);
    System.out.println("Weight of mybox2 is " + mybox2.weight);
  }
}
```

该程序的输出显示如下：

```
Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3

Volume of mybox2 is 24.0
Weight of mybox2 is 0.076
```

BoxWeight继承了Box的所有特征并为自己增添了一个weight成员。没有必要让BoxWeight重新创建Box中的所有特征。为满足需要我们只要扩展Box就可以了。

继承的一个主要优势在于一旦你已经创建了一个超类，而该超类定义了适用于一组对象的属性，它可用来创建任何数量的说明更多细节的子类。每一个子类能够正好制作它自己的分类。例如，下面的类继承了Box并增加了一个颜色属性：

```
// Here, Box is extended to include color.
class ColorBox extends Box {
  int color; // color of box

  ColorBox(double w, double h, double d, int c) {
    width = w;
    height = h;
    depth = d;
    color = c;
  }
}
```

记住，一旦你已经创建了一个定义了对象一般属性的超类，该超类可以被继承以生成特殊用途的类。每一个子类只增添它自己独特的属性。这是继承的本质。

8.1.3 超类变量可以引用子类对象

超类的一个引用变量可以被任何从该超类派生的子类的引用赋值。你将发现继承的这个方面在很多条件下是很有用的。例如，考虑下面的程序：

```
class RefDemo {
  public static void main(String args[]) {
    BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
    Box plainbox = new Box();
    double vol;

    vol = weightbox.volume();
    System.out.println("Volume of weightbox is " + vol);
    System.out.println("Weight of weightbox is " +
                       weightbox.weight);
    System.out.println();
```

```
    // assign BoxWeight reference to Box reference
    plainbox = weightbox;

    vol = plainbox.volume(); // OK, volume() defined in Box
    System.out.println("Volume of plainbox is " + vol);

    /* The following statement is invalid because plainbox
       does not define a weight member. */
//  System.out.println("Weight of plainbox is " + plainbox.weight);
  }
}
```

这里，weightbox是BoxWeight对象的一个引用，plainbox是Box对象的一个引用。既然BoxWeight是Box的一个子类，允许用一个weightbox对象的引用给plainbox赋值。

理解是引用变量的类型——而不是引用对象的类型——决定了什么成员可以被访问。也就是说，当一个子类对象的引用被赋给一个超类引用变量时，你只能访问超类定义的对象的那一部分。这是为什么plainbox不能访问weight的原因，甚至是它引用了一个BoxWeight对象也不行。仔细想一想，这是有道理的，因为超类不知道子类增加的属性。这就是本程序中的最后一行被注释掉的原因。Box的引用访问weight域是不可能的，因为它没有定义。

尽管前面部分看起来有一点深奥，它是很重要的实际应用——本章后面将讨论的两种应用之一。

8.2　使用super

在前面的例子中，从Box派生的类并没有体现出它们的实际上是多么有效和强大。例如，BoxWeight构造函数明确的初始化了Box( )的width、height和depth成员。这些重复的代码在它的超类中已经存在，这样做效率很低，而且，这意味着子类必须被同意具有访问这些成员的权力。然而，有时你希望创建一个超类，该超类可以保持它自己实现的细节（也就是说，它保持私有的数据成员）。这种情况下，子类没有办法直接访问或初始化它自己的这些变量。既然封装是面向对象的基本属性，Java提供了该问题的解决方案是不值得奇怪的。任何时候一个子类需要引用它直接的超类，它可以用关键字super来实现。

super有两种通用形式。第一种调用超类的构造函数。第二种用来访问被子类的成员隐藏的超类成员。下面分别介绍每一种用法。

8.2.1　使用super调用超类构造函数

子类可以调用超类中定义的构造函数方法，用super的下面形式：

```
super(parameter-list);
```

这里，parameter-list定义了超类中构造函数所用到的所有参数。super( )必须是在子类构造函数中的第一个执行语句。

为了了解怎样运用super( )，考虑下面BoxWeight( )的改进版本：

```
// BoxWeight now uses super to initialize its Box attributes.
```

```
class BoxWeight extends Box {
  double weight; // weight of box

  // initialize width, height, and depth using super()
  BoxWeight(double w, double h, double d, double m) {
    super(w, h, d); // call superclass constructor
    weight = m;
  }
}
```

这里，BoxWeight( )调用带w、h和d参数的super( )方法。这使Box( )构造函数被调用，用w、h和d来初始化width, height, 和 depth。BoxWeight不再自己初始化这些值。它只需初始化它自己的特殊值：weight。这种方法使Box可以自由的根据需要把这些值声明成private。

上面的例子，调用super( )用了三个参数。既然构造函数可以被重载，可以用超类定义的任何形式调用super( )，执行的构造函数将是与所传参数相匹配的那一个。例如，下面是BoxWeight一个完整的实现，BoxWeight具有以不同方法构造盒子的构造函数。在每种情况下，用适当的参数调用super( )。注意width, height, and depth在Box是私有的。

```
// A complete implementation of BoxWeight.
class Box {
  private double width;
  private double height;
  private double depth;

  // construct clone of an object
  Box(Box ob) { // pass object to constructor
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
  }
  // constructor used when all dimensions specified
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  // constructor used when no dimensions specified
  Box() {
    width = -1;  // use -1 to indicate
    height = -1; // an uninitialized
    depth = -1;  // box
  }

  // constructor used when cube is created
  Box(double len) {
    width = height = depth = len;
  }

  // compute and return volume
  double volume() {
    return width * height * depth;
  }
```

```
}

// BoxWeight now fully implements all constructors.
class BoxWeight extends Box {
  double weight; // weight of box

  // construct clone of an object
  BoxWeight(BoxWeight ob) { // pass object to constructor
    super(ob);
    weight = ob.weight;
  }

  // constructor when all parameters are specified
  BoxWeight(double w, double h, double d, double m) {
    super(w, h, d); // call superclass constructor
    weight = m;
  }
  // default constructor
  BoxWeight() {
    super();
    weight = -1;
  }

  // constructor used when cube is created
  BoxWeight(double len, double m) {
    super(len);
    weight = m;
  }
}

class DemoSuper {
  public static void main(String args[]) {
    BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
    BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
    BoxWeight mybox3 = new BoxWeight(); // default
    BoxWeight mycube = new BoxWeight(3, 2);
    BoxWeight myclone = new BoxWeight(mybox1);
    double vol;

    vol = mybox1.volume();
    System.out.println("Volume of mybox1 is " + vol);
    System.out.println("Weight of mybox1 is " + mybox1.weight);
    System.out.println();

    vol = mybox2.volume();
    System.out.println("Volume of mybox2 is " + vol);
    System.out.println("Weight of mybox2 is " + mybox2.weight);
    System.out.println();

    vol = mybox3.volume();
    System.out.println("Volume of mybox3 is " + vol);
    System.out.println("Weight of mybox3 is " + mybox3.weight);
    System.out.println();

    vol = myclone.volume();
```

```
    System.out.println("Volume of myclone is " + vol);
    System.out.println("Weight of myclone is " + myclone.weight);
    System.out.println();
    vol = mycube.volume();
    System.out.println("Volume of mycube is " + vol);
    System.out.println("Weight of mycube is " + mycube.weight);
    System.out.println();
  }
}
```

该程序产生下面的输出：

```
Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3
Volume of mybox2 is 24.0
Weight of mybox2 is 0.076
Volume of mybox3 is -1.0
Weight of mybox3 is -1.0
Volume of myclone is 3000.0
Weight of myclone is 34.3
Volume of mycube is 27.0
Weight of mycube is 2.0
```

特别注意BoxWeight( )中的这个构造函数：

```
// construct clone of an object
BoxWeight(BoxWeight ob) { // pass object to constructor
  super(ob);
  weight = ob.weight;
}
```

注意super( )被用一个BoxWeight类型而不是Box类型的对象调用。这仍然调用了构造函数Box(Box ob)。前面已经提醒过，一个超类变量可以引用作为任何一个从它派生的对象。因此，我们可以传递一个BoxWeight对象给Box构造函数。当然，Box只知道它自己成员的信息。

让我们复习super( )中的关键概念。当一个子类调用super( )，它调用它的直接超类的构造函数。这样，super( )总是引用调用类直接的超类。这甚至在多层次结构中也是成立的。还有，super( )必须是子类构造函数中的第一个执行语句。

8.2.2 Super的第2种用法

Super的第2种形式，除了总是引用它所在子类的超类，它的行为有点像this。这种用法有下面的通用形式：

```
super.member
```

这里，member既可以是1个方法也可以是1个实例变量。

Super的第2种形式多数是用于超类成员名被子类中同样的成员名隐藏的情况。思考下面简单的层次：

```
// Using super to overcome name hiding.
class A {
```

```
  int i;
}

// Create a subclass by extending class A.
class B extends A {
  int i; // this i hides the i in A

  B(int a, int b) {
    super.i = a; // i in A
    i = b; // i in B
  }

  void show() {
    System.out.println("i in superclass: " + super.i);
    System.out.println("i in subclass: " + i);
  }
}

class UseSuper {
  public static void main(String args[]) {
    B subOb = new B(1, 2);

    subOb.show();
  }
}
```

该程序输出如下：

```
i in superclass: 1
i in subclass: 2
```

尽管B中的实例变量i隐藏了A中的i，使用super就可以访问超类中定义的i。你将会看到，super也可以用来调用超类中被子类隐藏的方法。

8.3　创建多级类层次

到目前为止，我们已经用到了只含有一个超类和一个子类的简单类层次结构。然而，你可以如你所愿的建立包含任意多层继承的类层次。前面提到，用一个子类作为另一个类的超类是完全可以接受的。例如，给定三个类A，B和C。C是B的一个子类，而B又是A的一个子类。当这种类型的情形发生时，每个子类继承它的所有超类的属性。这种情况下，C继承B和A的所有方面。为了理解多级层次的用途，考虑下面的程序。该程序中，子类BoxWeight用作超类来创建一个名为Shipment的子类。Shipment继承了BoxWeight和Box的所有特征，并且增加了一个名为cost的成员，该成员记录了运送这样一个小包的费用。

```
// Extend BoxWeight to include shipping costs.

// Start with Box.
class Box {
  private double width;
  private double height;
```

```
  private double depth;

  // construct clone of an object
  Box(Box ob) { // pass object to constructor
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
  }

  // constructor used when all dimensions specified
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  // constructor used when no dimensions specified
  Box() {
    width = -1;  // use -1 to indicate
    height = -1; // an uninitialized
    depth = -1;  // box
  }

  // constructor used when cube is created
  Box(double len) {
    width = height = depth = len;
  }

  // compute and return volume
  double volume() {
    return width * height * depth;
  }
}

// Add weight.
class BoxWeight extends Box {
  double weight; // weight of box

  // construct clone of an object
  BoxWeight(BoxWeight ob) { // pass object to constructor
    super(ob);
    weight = ob.weight;
  }
  // constructor when all parameters are specified
  BoxWeight(double w, double h, double d, double m) {
    super(w, h, d); // call superclass constructor
    weight = m;
  }

  // default constructor
  BoxWeight() {
    super();
    weight = -1;
  }
  // constructor used when cube is created
```

```
  BoxWeight(double len, double m) {
    super(len);
    weight = m;
  }
}

// Add shipping costs
class Shipment extends BoxWeight {
  double cost;

  // construct clone of an object
  Shipment(Shipment ob) { // pass object to constructor
    super(ob);
    cost = ob.cost;
  }

  // constructor when all parameters are specified
  Shipment(double w, double h, double d,
           double m, double c) {
    super(w, h, d, m); // call superclass constructor
    cost = c;
  }

  // default constructor
  Shipment() {
    super();
    cost = -1;
  }

  // constructor used when cube is created
  Shipment(double len, double m, double c) {
    super(len, m);
    cost = c;
  }
}

class DemoShipment {
  public static void main(String args[]) {
    Shipment shipment1 =
              new Shipment(10, 20, 15, 10, 3.41);
    Shipment shipment2 =
              new Shipment(2, 3, 4, 0.76, 1.28);

    double vol;

    vol = shipment1.volume();
    System.out.println("Volume of shipment1 is " + vol);
    System.out.println("Weight of shipment1 is "
                       + shipment1.weight);
    System.out.println("Shipping cost: $" + shipment1.cost);
    System.out.println();

    vol = shipment2.volume();
    System.out.println("Volume of shipment2 is " + vol);
    System.out.println("Weight of shipment2 is "
```

```
                    + shipment2.weight);
    System.out.println("Shipping cost: $" + shipment2.cost);
  }
}
```

下面是该程序的输出：

```
Volume of shipment1 is 3000.0
Weight of shipment1 is 10.0
Shipping cost: $3.41
Volume of shipment2 is 24.0
Weight of shipment2 is 0.76
Shipping cost: $1.28
```

因为继承关系，Shipment可以利用原先定义好的Box 和BoxWeight类，仅为自己增加特殊用途的其他信息。这体现了继承的部分价值；它允许代码重用。

该例阐述了另一个重要的知识点：super( )总是引用子类最接近的超类的构造函数。Shipment中super( )调用了BoxWeight的构造函数。BoxWeight中的super( )调用了Box中的构造函数。在类层次结构中，如果超类构造函数需要参数，那么不论子类它自己需不需要参数，所有子类必须向上传递这些参数。

注意：在前面的例子中，整个类层次，包括Box，BoxWeight和Shipment，在一个文件中显示。这仅仅根据简便程度而定。Java中所有三个类可以被放置在它们自己的文件中且可以独立编译。实际上，在创建类层次结构的时候，使用分离的文件是常见的，不是罕见的。

8.4 何时调用构造函数

类层次结构创建以后，组成层次结构的类的构造函数以怎样的顺序被调用？举个例子来说，给定一个名为B的子类和超类A，是A的构造函数在B的构造函数之前调用，还是情况相反？回答是在类层次结构中，构造函数以派生的次序调用，从超类到子类。而且，尽管super( )必须是子类构造函数的第一个执行语句，无论你用到了super( )没有，这个次序不变。如果super( )没有被用到，每个超类的默认的或无参数的构造函数将执行。下面的例子阐述了何时执行构造函数：

```
// Demonstrate when constructors are called.

// Create a super class.
class A {
  A() {
    System.out.println("Inside A's constructor.");
  }
}

// Create a subclass by extending class A.
class B extends A {
  B() {
    System.out.println("Inside B's constructor.");
```

```
  }
}

// Create another subclass by extending B.
class C extends B {
  C() {
    System.out.println("Inside C's constructor.");
  }
}

class CallingCons {
  public static void main(String args[]) {
    C c = new C();
  }
}
```

该程序输出如下：

```
Inside A’s constructor
Inside B’s constructor
Inside C’s constructor
```

如你所见，构造函数以派生的顺序被调用。

仔细考虑，构造函数以派生的顺序执行是很有意义的。因为超类不知道任何子类的信息，任何它需要完成的初始化是与子类的初始化分离的，而且它可能是完成子类初始化的先决条件。因此，它必须最先执行。

8.5　方 法 重 载

类层次结构中，如果子类中的一个方法与它超类中的方法有相同的方法名和类型声明，称子类中的方法重载（override）超类中的方法。从子类中调用重载方法时，它总是引用子类定义的方法。而超类中定义的方法将被隐藏。考虑下面程序：

```
// Method overriding.
class A {
  int i, j;
  A(int a, int b) {
    i = a;
    j = b;
  }

  // display i and j
  void show() {
    System.out.println("i and j: " + i + " " + j);
  }
}

class B extends A {
  int k;

  B(int a, int b, int c) {
```

```
    super(a, b);
    k = c;
  }

  // display k - this overrides show() in A
  void show() {
    System.out.println("k: " + k);
  }
}

class Override {
  public static void main(String args[]) {
    B subOb = new B(1, 2, 3);

    subOb.show(); // this calls show() in B
  }
}
```

程序输出如下：

```
k: 3
```

当一个B类的对象调用show( )时，调用的是在B中定义的show( )版本。也就是说，B中的show( )方法重载了A中声明的show( )方法。

如果你希望访问被重载的超类的方法，可以用super。例如，在下面的B的版本中，在子类中超类的show( )方法被调用。这使所有的实例变量被显示。

```
class B extends A {
  int k;

  B(int a, int b, int c) {
    super(a, b);
    k = c;
  }
  void show() {
    super.show(); // this calls A's show()
    System.out.println("k: " + k);
  }
}
```

如果你用该版本的A代替先前的版本形式，将会得出下面输出：

```
i and j: 1 2
k: 3
```

这里，super.show( )调用了超类的show( )方法。

方法覆盖仅在两个方法的名称和类型声明都相同时才发生。如果它们不同，那么两个方法就只是重载。例如，考虑下面的程序，它修改了前面的例子：

```
// Methods with differing type signatures are overloaded - not
// overridden.
class A {
  int i, j;
```

```
  A(int a, int b) {
    i = a;
    j = b;
  }

  // display i and j
  void show() {
    System.out.println("i and j: " + i + " " + j);
  }
}

// Create a subclass by extending class A.
class B extends A {
  int k;

  B(int a, int b, int c) {
    super(a, b);
    k = c;
  }

  // overload show()
  void show(String msg) {
    System.out.println(msg + k);
  }
}

class Override {
  public static void main(String args[]) {
    B subOb = new B(1, 2, 3);

    subOb.show("This is k: "); // this calls show() in B
    subOb.show(); // this calls show() in A
  }
}
```

该程序的输出显示如下：

```
This is k: 3
i and j: 1 2
```

B中show( )带有一个字符串参数。这是它的类型标签与A中的不同，A中的show( )没有带参数。因此没有覆盖（或名称隐藏）发生。

8.6　动态方法调度

前面的例题说明了方法重载机制，但并没有显示它们的作用。实际上，如果方法重载只是一个名字空间的约定，那么它最多是有趣的，但是没有实际价值的。然而，情况并不如此。方法重载构成Java的一个最强大的概念的基础：动态方法调度（dynamic method dispatch）。动态方法调度是一种在运行时而不是编译时调用重载方法的机制。动态方法调度是很重要的，因为这也是Java实现运行时多态性的基础。

让我们从重述一个重要的原则开始：超类的引用变量可以引用子类对象。Java用这一事实来解决在运行期间对重载方法的调用。过程如下：当一个重载方法通过超类引用被调用，Java根据当前被引用对象的类型来决定执行哪个版本的方法。如果引用的对象类型不同，就会调用一个重载方法的不同版本。换句话说，是被引用对象的类型（而不是引用变量的类型）决定执行哪个版本的重载方法。因此，如果超类包含一个被子类重载的方法，那么当通过超类引用变量引用不同对象类型时，就会执行该方法的不同版本。

下面是阐述动态方法调度的例子：

```
// Dynamic Method Dispatch
class A {
   void callme() {
     System.out.println("Inside A's callme method");
  }
}

class B extends A {
  // override callme()
  void callme() {
    System.out.println("Inside B's callme method");
  }
}

class C extends A {
  // override callme()
  void callme() {
    System.out.println("Inside C's callme method");
  }
}

class Dispatch {
  public static void main(String args[]) {
    A a = new A(); // object of type A
    B b = new B(); // object of type B
    C c = new C(); // object of type C
    A r; // obtain a reference of type A

    r = a; // r refers to an A object
    r.callme(); // calls A's version of callme

    r = b; // r refers to a B object
    r.callme(); // calls B's version of callme
    r = c; // r refers to a C object
    r.callme(); // calls C's version of callme
  }
}
```

该程序的输出如下：

```
Inside A’s callme method
Inside B’s callme method
Inside C’s callme method
```

程序创建了一个名为A的超类以及它的两个子类B和C。子类B和C重载A中定义的callme( )方法。main( )主函数中，声明了A、B和C类的对象。而且，一个A类型的引用r也被声明。就像输出所显示的，所执行的callme( )版本由调用时引用对象的类型决定。如果它是由引用变量r的类型决定的，你将会看到对A的callme( )方法的三次调用。

熟悉C++的读者会认同Java中的重载方法与C++中的虚函数类似。

8.6.1　为什么要重载方法

前面声明过，重载方法允许Java支持运行时多态性。多态性是面向对象编程的本质，原因如下：它允许通用类指定方法，这些方法对该类的所有派生类都是公用的。同时该方法允许子类定义这些方法中的某些或全部的特殊实现。重载方法是Java实现它的多态性——“一个接口，多个方法”的另一种方式。

成功应用多态的关键部分是理解超类和子类形成了一个从简单到复杂类层次。正确应用多态，超类提供子类可以直接运用的所有元素。多态也定义了这些派生类必须自己实现的方法。这允许子类在加强一致接口的同时，灵活的定义它们自己的方法。这样，通过继承和重载方法的联合，超类可以定义供它的所有子类使用的方法的通用形式。

动态的运行时多态是面向对象设计代码重用的一个最强大的机制。现有代码库在维持抽象接口同时不重新编译的情况下调用新类实例的能力是一个极其强大的工具。

8.6.2　应用方法重载

让我们看一个运用方法重载的更实际的例子。下面的程序创建了一个名为Figure的超类，它存储不同二维对象的大小。它还定义了一个方法area( )，该方法计算对象的面积。程序从Figure派生了两个子类。第一个是Rectangle，第二个是Triangle。每个子类重载area( )方法，它们分别返回一个矩形和一个三角形的面积。

```
// Using run-time polymorphism.
class Figure {
  double dim1;
  double dim2;

  Figure(double a, double b) {
    dim1 = a;
    dim2 = b;
  }

  double area() {
    System.out.println("Area for Figure is undefined.");
    return 0;
  }
}

class Rectangle extends Figure {
  Rectangle(double a, double b) {
    super(a, b);
  }

  // override area for rectangle
```

```
  double area() {
    System.out.println("Inside Area for Rectangle.");
    return dim1 * dim2;
  }
}

class Triangle extends Figure {
  Triangle(double a, double b) {
    super(a, b);
  }

  // override area for right triangle
  double area() {
    System.out.println("Inside Area for Triangle.");
    return dim1 * dim2 / 2;
  }
}

class FindAreas {
  public static void main(String args[]) {
    Figure f = new Figure(10, 10);
    Rectangle r = new Rectangle(9, 5);
    Triangle t = new Triangle(10, 8);

    Figure figref;

    figref = r;
    System.out.println("Area is " + figref.area());

    figref = t;
    System.out.println("Area is " + figref.area());

    figref = f;
    System.out.println("Area is " + figref.area());
  }
}
```

该程序输出如下：

```
Inside Area for Rectangle.
Area is 45
Inside Area for Triangle.
Area is 40
Area for Figure is undefined.
Area is 0
```

通过继承和运行时多态的双重机制，可以定义一个被很多不同却有关的对象类型运用的一致的接口。这种情况下，如果一个对象是从Figure派生，那么它的面积可以由调用area( )来获得。无论用到哪种图形的类型，该操作的接口是相同的。

8.7　使用抽象类

有些情况下，你希望定义一个超类，该超类定义了一种给定结构的抽象但是不提供任何完整的方法实现。也就是说，有时你希望创建一个只定义一个被它的所有子类共享的通用形式，由每个子类自己去填写细节。这样的类决定了子类所必须实现的方法的本性。这类情形下一种可能发生的情况是超类不能创建一个方法的有意义的实现。前面的例子中用到的类Figure就属于这种情况。area( )的定义仅是一个占位符。它不会计算和显示任何类型对象的面积。

当创建自己的类库时你会看到，超类中的方法没有实际意义并不罕见。你有两种方法可以处理这种情况。第一种，如前面的例子所示，仅仅是报告一个出错消息。尽管这种方式在某些场合是有用的——例如调试——但是它不是很适用的。你还有一种方法就是通过子类重载该方法以使它对子类有意义。考虑Triangle类，如果不定义area( )它是毫无意义的。这种情况下，你希望有方法确保子类真正重载了所有必须的方法。Java对于这个问题的解决是用抽象方法(abstract method)。

你可以通过指定abstract类型修饰符由子类重载某些方法。这些方法有时被作为子类责任（subclasser responsibility）引用，因为它们没有在超类中指定的实现。这样子类必须重载它们——它们不能简单地使用超类中定义的版本。声明一个抽象方法，用下面的通用形式：

```
abstract type name(parameter-list);
```

正如你所看到的，不存在方法体。

任何含有一个或多个抽象方法的类都必须声明成抽象类。声明一个抽象类，只需在类声明开始时在关键字class前使用关键字abstract。抽象类没有对象。也就是说，一个抽象类不能通过new操作符直接实例化。这样的对象是无用的，因为抽象类是不完全定义的。而且，你不能定义抽象构造函数或抽象静态方法。所有抽象类的子类都必须执行超类中的所有抽象方法或者是它自己也声明成abstract。

下面是具有一个抽象方法类的简单例题。该类后面是一个执行抽象方法的类：

```
// A Simple demonstration of abstract.
abstract class A {
  abstract void callme();

  // concrete methods are still allowed in abstract classes
  void callmetoo() {
    System.out.println("This is a concrete method.");
  }
}

class B extends A {
  void callme() {
    System.out.println("B's implementation of callme.");
  }
```

```
}

class AbstractDemo {
  public static void main(String args[]) {
    B b = new B();

    b.callme();
    b.callmetoo();
  }
}
```

注意程序中声明A的对象。刚刚讲过，实例化一个抽象类是不可能的。另外一点要注意：类A实现一个具体的方法callmetoo( )。这是完全可接受的，抽象类可以包括它们合适的很多实现。

因为Java的运行时多态是通过使用超类引用实现的，所以尽管抽象类不能用来实例化，它们可以用来创建对象引用。这样，创建一个抽象类的引用是可行的，这样它可以用来指向一个子类对象。在下面的程序中你将会看到这种特性的运用。

运用抽象类，你可以改善前面所显示的Figure类。因为对于一个未定义的二维图形，面积的概念是没有意义的，下面的程序在Figure内将area( )定义成抽象方法。这样当然意味着从Figure派生的所有类都必须重载area( )方法。

```
// Using abstract methods and classes.
abstract class Figure {
  double dim1;
  double dim2;

  Figure(double a, double b) {
    dim1 = a;
    dim2 = b;
  }

  // area is now an abstract method
  abstract double area();
}

class Rectangle extends Figure {
  Rectangle(double a, double b) {
    super(a, b);
  }

  // override area for rectangle
  double area() {
    System.out.println("Inside Area for Rectangle.");
    return dim1 * dim2;
  }
}
```

```
class Triangle extends Figure {
  Triangle(double a, double b) {
    super(a, b);
  }

  // override area for right triangle
  double area() {
    System.out.println("Inside Area for Triangle.");
    return dim1 * dim2 / 2;
  }
}

class AbstractAreas {
  public static void main(String args[]) {
  // Figure f = new Figure(10, 10); // illegal now
    Rectangle r = new Rectangle(9, 5);
    Triangle t = new Triangle(10, 8);
    Figure figref; // this is OK, no object is created

    figref = r;
    System.out.println("Area is " + figref.area());

    figref = t;
    System.out.println("Area is " + figref.area());
  }
}
```

Main（）内的注释暗示，定义Figure类型的对象不再是可能的了，因为现在它是抽象类。而且，所有Figure的子类都必须重载area( )方法。为证明这点，试着创建不重载area( )的子类。你会收到一个编译时错误。

尽管不可能创建一个Figure类型的对象，你可以创建一个Figure类型的引用变量。变量figref声明成Figure的一个引用，意思是说它可以用来引用任何从Figure派生的对象。刚才解释过的，通过超类引用变量重载方法在运行时解决。

8.8　继承中使用final

Final关键字有三个用途。第一，它可以用来创建一个已命名常量的等价物。这个用法在前面的章节中已有描述。Final的其他两个用法是应用于继承的，这两种用法都会在下面阐述。

8.8.1　使用final阻止重载

尽管方法重载是Java的一个最强大的特性，有些时候你希望防止它的发生。不接受方法被重载，在方法前定义final修饰符。声明成final的方法不能被重载。下面的程序段阐述

了final的用法：

```
class A {
  final void meth() {
    System.out.println("This is a final method.");
  }
}

class B extends A {
  void meth() { // ERROR! Can't override.
    System.out.println("Illegal!");
  }
}
```

因为meth( )被声明成final，它不能被B重载，如果你试图这样做，将会生成一个编译时错误。

定义成final的方法有时可以提高程序性能：编译器可以自由的内嵌调用final方法因为它知道这些方法不能被子类重载。当一个小的final函数被调用，通常Java编译器可以通过调用方法的编译代码直接内嵌来备份子程序的字节码，这样减小了与方法调用有关的昂贵开销。内嵌仅仅是final方法的一个可选项。通常，Java在运行时动态的调用方法，这叫做后期绑定（late binding）。然而既然final方法不能被重载，对方法的调用可以在编译时解决，这叫做早期绑定（early binding）。

8.8.2 使用final阻止继承

有时你希望防止一个类被继承。做到这点只需在类声明前加final。声明一个final类含蓄的宣告了它的所有方法也都是final。你可能会想到，声明一个既是abstract的又是final的类是不合法的，因为抽象类本身是不完整的，它依靠它的子类提供完整的实现。

下面是一个final类的例子：

```
final class A {
  // ...
}

// The following class is illegal.
class B extends A { // ERROR! Can't subclass A
  // ...
}
```

像注释暗示的，B继承A是不合法的，因为A声明成final。

8.9 Object类

有一种由Java定义的特殊的类Object。所有其他的类都是Object的子类。也就是说，Object是所有其他类的超类。这意味着一个Object类型的引用变量可以引用其他任何一个类的对象。同样，因为数组像类一样执行，Object类型变量可以引用任何数组。

Object定义了下面的方法，意味着它们可以被用于任何对象，如表8-1所示。

表 8-1　Object 类定义的方法及其用途

方法	用途
Object clone( )	创建一个和被复制的对象完全一样的新对象
boolean equals(Object object)	判定对象是否相等
void finalize( )	在一个不常用的对象被使用前调用
Class getClass( )	获取运行时一个对象的类
int hashCode( )	返回调用对象有关的散列值
void notify( )	恢复一个等待调用对象线程的执行
void notifyAll( )	恢复所有等待调用对象线程的执行
String toString( )	返回描述对象的一个字符串
void wait( )	等待另一个线程的执行
void wait(long milliseconds)	
void wait(long milliseconds, int nanoseconds)	

getClass( )，notify( )，notifyAll( )和wait( )方法被定义成final。你可以重载除这些方法以外的其他方法。这些方法在本书的其他地方有所描述。然而，现在注意两个方法：equals( )和toString( )。equals( )方法比较两个对象的内容。如果对象是相等的，它返回true，否则返回false。toString( )方法返回一个包含调用它的对象描述的字符串。而且，该方法在对象用println( )输出时自动调用。很多类重载该方法。这样做使它们生成它们创建对象类型的一个特殊描述。要了解更多toString( )信息请参看第13章。

第9章　包 和 接 口

本章我们讲述Java最具有革新性的两个特点：包和接口。包（package）是类的容器，用来保存划分的类名空间。例如，一个包允许你创建一个名为List的类，你可以把它保存在你自己的包中而不用考虑和其他地方的某个名为List的类相冲突。包以分层方式保存并被明确的引入新的类定义。

在前面的章节你已经了解了怎样在类中定义数据接口的方法。通过运用关键字interface，Java允许你充分抽象它实现的接口。用接口，你可以定义一系列的被一个类或多个类执行的方法。接口自己不定义任何实现。尽管它们与抽象类相似，接口有一个特殊的功能：类可以实现多个接口。与之相反，类只能继承一个超类（抽象类或其他）。

包和接口是Java程序的两个基本组成。一般来说，Java源程序可以包含下面的四个内部部分的任何一个（或所有）。

- 单个接口声明（可选）
- 任意数目的引入语句（可选）
- 单个公共类声明（必须）
- 对包来说是私有的任意数目的类（可选）

其中只有一个——单个公共类声明——在前面的程序中被用到。本章将探究剩下的三个部分。

9.1　包

在前面的章节，每个例题类名从相同的名称空间获得。意思是说为避免名称冲突每个类都必须用惟一的名称。下面，没有管理名称空间的办法，你可能觉得不方便，因为每个单独的类都有描述性的名称。你还需要有确保你选用的类名是独特的且不和其他程序员选择的类名相冲突的方法（假想一小组程序员为用“Foobar”作类名而争斗。或者，设想整个Internet团体为谁最先为类取名为“Espresso”而争论）。感谢上帝，Java提供了把类名空间划分为更多易管理的块的机制。这种机制就是包。包既是命名机制也是可见度控制机制。你可以在包内定义类，而且在包外的代码不能访问该类。这使你的类相互之间有隐私，但不被其他世界所知。

9.1.1　定义包

创建一个包是很简单的：只要包含一个package命令作为一个Java源文件的第一句就可以了。该文件中定义的任何类将属于指定的包。package语句定义了一个存储类的名字空间。如果你省略package 语句，类名被输入一个默认的没有名称的包（这是为什么在以前你不

用担心包的问题的原因）。尽管默认包对于短的例子程序很好用，但对于实际的应用程序它是不适当的。多数情况，需要为自己的代码定义一个包。

下面是package 声明的通用形式：

```
package pkg;
```

这里，pkg 是包名。例如，下面的声明创建了一个名为MyPackage的包。

```
package MyPackage;
```

Java用文件系统目录来存储包。例如，任何你声明的MyPackage中的一部分的类的.class文件被存储在一个MyPackage 目录中。记住这种情况是很重要的，目录名必须和包名严格匹配。

多个文件可以包含相同package声明。package声明仅仅指定了文件中定义的文件属于哪一个包。它不拒绝其他文件的其他方法成为相同包的一部分。多数实际的包伸展到很多文件。

你可以创建包层次。为做到这点，只要将每个包名与它的上层包名用点号“.”分隔开就可以了。一个多级包的声明的通用形式如下：

```
package pkg1[.pkg2[.pkg3]];
```

包层次一定要在Java开发系统的文件系统中有所反映。例如，一个由下面语句定义的包：

```
package java.awt.image;
```

需要在你的UNIX、Windows或Macintosh文件系统的 java/awt/image, java\awt\image或java:awt:image中分别保存。一定要仔细选用包名。你不能在没有对保存类的目录重命名的情况下重命名一个包。

9.1.2　理解类路径（CLASSPATH）

在介绍运用包的例子之前，关于类路径环境变量的简单讨论是必要的。当包从访问控制和名称-空间-冲突中解决很多问题时，在编译和运行程序时它们导致某些古怪的难点。这是因为Java编译器考虑的特定位置作为包层次的根被类路径(CLASSPATH)控制。直到现在，你在同样的未命名的默认包中保存所有的类。这样做允许你仅仅通过在命令行键入类名编译源文件和运行Java解释器，并得到结果。这种情况下它还会工作是因为默认的当前工作目录（.）通常在类路径环境变量中为Java运行时间默认定义。然而，当有包参与时，事情就不这么简单。下面是原因。

假设你在一个test包中创建了一个名为PackTest 的类。因为你的目录结构必须与包相匹配，你创建一个名为test的目录并把PackTest.java装入该目录。然后使test 成为当前目录并编译PackTest.java。这导致PackTest.class被存放在test目录下。当你试图运行PackTest时，java解释器报告一个与“不能发现PackTest类”相似的错误消息。这是因为该类现在被保存在test包中。不再能简单用PackTest来引用。必须通过列举包层次来引用该类。引用包层次时用逗号将包名隔开。该类现在必须叫做test.PackTest。然而，如果你试图用test.PackTest，你将

仍然收到一个与“不能发现test/PackTest类”相似的出错消息。

仍然收到错误消息的原因隐藏在类路径变量中。记住，类路径设置顶层类层次。问题在于在当前工作目录下不存在test子目录，因为你是工作在test目录本身。

在这个问题上你有两个选择：改变目录到上一级然后用java test.PackTest，或者在类路径环境变量增加你的开发类层次结构的顶层。然后可以用java test.PackTest，Java将发现正确的.class文件。例如，如果你的源代码在目录C:\myjava下，那么设置类路径为：

```
.;C:\myjava;C:\java\classes
```

9.1.3　一个简短的包的例子

记住前面的讨论，试试下面简单的包：

```
// A simple package
package MyPack;

class Balance {
  String name;
  double bal;

  Balance(String n, double b) {
    name = n;
    bal = b;
  }

  void show() {
    if(bal<0)
      System.out.print("--> ");
    System.out.println(name + ": $" + bal);
  }
}

class AccountBalance {
  public static void main(String args[]) {
    Balance current[] = new Balance[3];

    current[0] = new Balance("K. J. Fielding", 123.23);
    current[1] = new Balance("Will Tell", 157.02);
    current[2] = new Balance("Tom Jackson", -12.33);

    for(int i=0; i<3; i++) current[i].show();
  }
}
```

称该文件名为 AccountBalance.java，把它存放在MyPack目录中。

接着，编译文件。确信结果文件.class同样在MyPack 目录中。然后用下面的命令行执行AccountBalance 类：

```
java MyPack.AccountBalance
```

记住，当你执行该命令时你必须在MyPack的上级目录，或者把类路径环境变量设置成合适的值。

如上所述，AccountBalance现在是MyPack包的一部分。这意味着它不能自己执行。也就是说你不能用下面的命令行：

```
java AccountBalance
```

AccountBalance必须和它的包名一起使用。

9.2　访 问 保 护

前面已经学习了Java的访问控制机制的很多方面和它的访问说明符。例如，你已经知道一个类的private成员仅可以被该类的其他成员访问。包增加了访问控制的另一个维度。如你所看到的，Java提供很多级别的保护以使在类、子类和包中有完善的访问控制。

类和包都是封装和容纳名称空间和变量及方法范围的方法。包就像盛装类和下级包的容器。类就像是数据和代码的容器。类是Java的最小的抽象单元。因为类和包的相互影响，Java将类成员的可见度分为四个种类：

- 相同包中的子类
- 相同包中的非子类
- 不同包中的子类
- 既不在相同包又不在相同子类中的类

三个访问控制符，private、public和protected，提供了多种方法来产生这些种类所需访问的多个级别，表9-1总结了它们之间的相互作用。

表 9-1　类成员访问

	Private成员	默认的成员	Protected成员	Public成员
同一类中可见	是	是	是	是
同一个包中对子类可见	否	是	是	是
同一个包中对非子类可见	否	是	是	是
不同包中对子类可见	否	否	是	是
不同的包中对非子类可见	否	否	否	是

Java的访问控制机制看上去很复杂，我们可以按下面方法简化它。任何声明为public的内容可以被从任何地方访问。被声明成private的成员不能被该类外看到。如果一个成员不含有一个明确的访问说明，它对于子类或该包中的其他类是可见的。这是默认访问。如果你希望一个元素在当前包外可见，但仅仅是元素所在类的子类直接可见，把元素定义成protected。

表9-1仅适用于类成员。一个类只可能有两个访问级别：默认的或是公共的。如果一个类声明成public，它可以被任何其他代码访问。如果该类默认访问控制符，它仅可以被相同包中的其他代码访问。

9.2.1　一个访问的例子

下面的例子显示了访问修饰符的所有组合。该例有两个包和五个类。记住这两个不同包中的类需要被存储在以它们的包p1、p2命名的目录下。

第一个包定义了三个类： Protection, Derived, 和 SamePackage。第一个类以合法的保护模式定义了四个int 变量。变量n声明成默认受保护型。n_pri是private型，n_pro是protected，n_pub是public的。

该例中每一个后来的类试图访问该类一个实例中的变量。根据访问权限不编译的行用单行注释//。在每个这样的行之前都是列举该级保护将允许访问的地点的注释。

第二个类，Derived是同样包p1中Protection类的子类。这允许Derived访问Protection中的除n_pri以外的所有变量，因为它是private。第三个类，SamePackage，不是Protection的子类，但是是在相同的包中，也可以访问除n_pri以外的所有变量。

下面是Protection.java文件：

```
package p1;

public class Protection {
  int n = 1;
  private int n_pri = 2;
  protected int n_pro = 3;
  public int n_pub = 4;

  public Protection() {
    System.out.println("base constructor");
    System.out.println("n = " + n);
    System.out.println("n_pri = " + n_pri);
    System.out.println("n_pro = " + n_pro);
    System.out.println("n_pub = " + n_pub);
  }
}
```

下面是Derived.java文件：

```
package p1;

class Derived extends Protection {
  Derived() {
    System.out.println("derived constructor");
    System.out.println("n = " + n);

//  class only
//  System.out.println("n_pri = " + n_pri);

    System.out.println("n_pro = " + n_pro);
    System.out.println("n_pub = " + n_pub);
  }
}
```

下面是SamePackage.java文件：

```
package p1;
```

```
class SamePackage {
  SamePackage() {
    Protection p = new Protection();
    System.out.println("same package constructor");
    System.out.println("n = " + p.n);

//  class only
//  System.out.println("n_pri = " + p.n_pri);
    System.out.println("n_pro = " + p.n_pro);
    System.out.println("n_pub = " + p.n_pub);
  }
}
```

下面是另一个包p2的源代码。p2中定义的两个类重载了另两种受访问控制影响的情况。第一个类Protection2是p1.Protection的子类。这允许访问p1.Protection中除n_pri（因为它是private的）和n之外的所有变量，n是定义成默认保护型的。记住，默认型的只能允许类中或包中的代码访问。最后，OtherPackage类只访问了一个变量n_pub，它是定义成public型的。

下面是Protection2.java文件：

```
package p2;

class Protection2 extends p1.Protection {
  Protection2() {
    System.out.println("derived other package constructor");

//  class or package only
//  System.out.println("n = " + n);

//  class only
//  System.out.println("n_pri = " + n_pri);

    System.out.println("n_pro = " + n_pro);
    System.out.println("n_pub = " + n_pub);
  }
}
```

下面是OtherPackage.java文件：

```
package p2;

class OtherPackage {
  OtherPackage() {
    p1.Protection p = new p1.Protection();
    System.out.println("other package constructor");

//  class or package only
//  System.out.println("n = " + p.n);

//  class only
//  System.out.println("n_pri = " + p.n_pri);

//  class, subclass or package only
//  System.out.println("n_pro = " + p.n_pro);
```

```
    System.out.println("n_pub = " + p.n_pub);
  }
}
```

如果你希望试试这两个包，下面是两个可以用的测试文件。包p1的测试文件如下：

```
// Demo package p1.
package p1;

// Instantiate the various classes in p1.
public class Demo {
  public static void main(String args[]) {
    Protection ob1 = new Protection();
    Derived ob2 = new Derived();
    SamePackage ob3 = new SamePackage();
  }
}
```

p2的测试文件如下：

```
// Demo package p2.
package p2;
// Instantiate the various classes in p2.
public class Demo {
  public static void main(String args[]) {
    Protection2 ob1 = new Protection2();
    OtherPackage ob2 = new OtherPackage();
  }
}
```

9.3　引 入 包

包的存在是划分不同类的好的机制，了解为什么所有Java内部的类都存在包中是很简单的。在未命名的默认包中是没有核心Java类的；所有的标准类都存储在相同的包中。既然包中的类必须包含它们的包名才能完全有效，为每个你想用的包写一个长的逗点分离的包路径名是枯燥的。因为这点，Java包含了import语句来引入特定的类甚至是整个包。一旦被引入，类可以被直呼其名的引用。import语句对于程序员是很方便的而且在技术上并不需要编写完整的Java程序。如果你在程序中将要引用若干个类，那么用import 语句将会节省很多打字时间。

在Java源程序文件中，import语句紧接着package语句（如果package 语句存在），它存在于任何类定义之前，下面是import声明的通用形式：

```
import pkg1[.pkg2].(classname|*);
```

这里，pkg1是顶层包名，pkg2是在外部包中的用逗点（.）隔离的下级包名。除非是文件系统的限制，不存在对于包层次深度的实际限制。最后，你要么指定一个清楚的类名，要么指定一个星号（*），该星号表明Java编译器应该引入整个包。下面的代码段显示了所用的两种形式：

```
import java.util.Date;
import java.io.*;
```

警告：星号形式可能会增加编译时间——特别是在你引入多个大包时。因为这个原因，明确的命名你想要用到的类而不是引入整个包是一个好的方法。然而，星号形式对运行时间性能和类的大小绝对没有影响。

所有Java包含的标准Java类都存储在名为java的包中。基本语言功能被存储在java包中的java.lang包中。通常，你必须引入你所要用到的每个包或类，但是，既然Java在没有java.lang中的很多函数时是无用的，因此通过编译器为所有程序隐式引入java.lang是有必要的。这与下面的在你所有程序开头的一行是一样的：

```
import java.lang.*;
```

如果在你用星号形式引用的两个不同包中存在具有相同类名的类，编译器将保持沉默，除非你试图运用其中的一个。这种情况下，你会得到一个编译时错误并且必须明确的命名指定包中的类。

任何你用到类名的地方，你可以使用它的全名，全名包括它所有的包层次。例如，下面的程序使用了一个引入语句：

```
import java.util.*;
class MyDate extends Date {
}
```

没有import 语句的例子如下：

```
class MyDate extends java.util.Date {
}
```

如表9-1种所示，当一个包被引入，仅仅是该包中声明成public的项目可以在引入代码中对非子类可用。例如，如果你希望前面显示的MyPack 包中的Balance类在MyPack外可以被独立的类运用，那么你需要声明它为public型，并把它存在自己的文件中，如下：

```
package MyPack;
/* Now, the Balance class, its constructor, and its
   show() method are public.  This means that they can
   be used by non-subclass code outside their package.
*/
public class Balance {
  String name;
  double bal;

  public Balance(String n, double b) {
    name = n;
    bal = b;
  }

  public void show() {
    if(bal<0)
      System.out.print("--> ");
    System.out.println(name + ": $" + bal);
```

```
  }
}
```

如你所见，Balance类现在是public。而且，它的构造函数和show( )方法也是public。这意味着它们可以被任何类型的MyPack包之外的代码访问。例如下面TestBalance引入了MyPack，那么它可以利用Balance类：

```
import MyPack.*;

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

    /* Because Balance is public, you may use Balance
       class and call its constructor. */
    Balance test = new Balance("J. J. Jaspers", 99.88);

    test.show(); // you may also call show()
  }
}
```

作为一个试验，从Balance 类移去public 修饰符，然后编译TestBalance，和分析得到的结论一样，将会产生错误。

9.4　接口(interface)

用关键字interface，你可以从类的实现中抽象一个类的接口。也就是说，用interface，你可以指定一个类必须做什么，而不是规定它如何去做。接口在语句构成上与类相似，但是它们缺少实例变量，而且它们定义的方法是不含方法体的。实际上，这意味着你可以定义不用假设它们怎样实现的接口。一旦接口被定义，任何类成员可以实现一个接口。而且，一个类可以实现多个接口。

要实现一个接口，接口定义的类必须创建完整的一套方法。然而，每个类都可以自由的决定它们自己实现的细节。通过提供interface关键字，Java允许你充分利用多态性的“一个接口，多个方法”。

接口是为支持运行时动态方法解决而设计的。通常，为使一个方法可以在类间调用，两个类都必须出现在编译时间里，以便Java编译器可以检查以确保方法特殊是兼容的。这个需求导致了一个静态的不可扩展的类环境。在一个系统中不可避免会出现这类情况，函数在类层次中越堆越高以致该机制可以为越来越多的子类可用。接口的设计避免了这个问题。它们把方法或方法系列的定义从类层次中分开。因为接口是在和类不同的层次中，与类层次无关的类实现相同的接口是可行的。这是实现接口的真正原因所在。

注意：接口增添了很多应用程序所需的功能。在一种语言例如C++中这些应用程序通常借助于多重继承来完成。

9.4.1　接口定义

接口定义很像类定义。下面是一个接口的通用形式：

```
access interface name {
       return-type method-name1(parameter-list);
       return-type method-name2(parameter-list);
       type final-varname1 = value;
       type final-varname2 = value;
       // ...
       return-type method-nameN(parameter-list);
       type final-varnameN = value;
}
```

这里，access要么是public，要么就没有用修饰符。当没有访问修饰符时，则是默认访问范围，而接口是包中定义的惟一的可以用于其他成员的东西。当它声明为public时，则接口可以被任何代码使用。name是接口名，它可以是任何合法的标识符。注意定义的方法没有方法体。它们以参数列表后面的分号作为结束。它们本质上是抽象方法；在接口中指定的方法没有默认的实现。每个包含接口的类必需实现所有的方法。

接口声明中可以声明变量。它们一般是final 和static型的，意思是它们的值不能通过实现类而改变。它们还必须以常量值初始化。如果接口本身定义成public ，所有方法和变量都是public的。

下面是一个接口定义的例子。它声明了一个简单的接口，该接口包含一个带单个整型参数的callback( )方法。

```
interface Callback {
  void callback(int param);
}
```

9.4.2　实现接口

一旦接口被定义，一个或多个类可以实现该接口。为实现一个接口，在类定义中包括implements 子句，然后创建接口定义的方法。一个包括implements 子句的类的一般形式如下：

```
access class classname [extends superclass]
                        [implements interface [,interface...]] {
     // class-body
}
```

这里，access要么是public的，要么是没有修饰符的。如果一个类实现多个接口，这些接口被逗号分隔。如果一个类实现两个声明了同样方法的接口，那么相同的方法将被其中任一个接口客户使用。实现接口的方法必须声明成public。而且，实现方法的类型必须严格与接口定义中指定的类型相匹配。

下面是一个小的实现Callback接口的例子程序：

```
class Client implements Callback {
  // Implement Callback's interface
  public void callback(int p) {
```

```
    System.out.println("callback called with " + p);
  }
}
```

注意callback( )用public 访问修饰符声明。

注意：当实现一个接口方法时，它必须声明成public。

类在实现接口时定义它自己的附加的成员，既是允许的，也是常见的。例如，下面的Client版本实现了callback( )方法，并且增加了nonIfaceMeth( )方法。

```
class Client implements Callback {
  // Implement Callback's interface
  public void callback(int p) {
    System.out.println("callback called with " + p);
  }

  void nonIfaceMeth() {
    System.out.println("Classes that implement interfaces " +
                       "may also define other members, too.");
  }
}
```

通过接口引用实现接口

你可以把变量定义成使用接口的对象引用而不是类的类型。任何实现了所声明接口的类的实例都可以被这样的一个变量引用。当你通过这些引用调用方法时，在实际引用接口的实例的基础上，方法被正确调用。这是接口的最显著特性之一。被执行的方法在运行时动态操作，允许在调用方法代码后创建类。调用代码在完全不知“调用者”的情况下可以通过接口来调度。这个过程和第8章描述的用超类引用来访问子类对象很相似。

警告：因为Java中在运行时动态查询方法与通常的方法调用相比会有一个非常庞大的花费，所以在对性能要求高的代码中不应该随意的使用接口。

下面的例子通过接口引用变量调用callback( )方法：

```
class TestIface {
  public static void main(String args[]) {
    Callback c = new Client();
    c.callback(42);
  }
}
```

该程序的输出如下：

```
callback called with 42
```

注意变量c被定义成接口类型Callback，而且被一个Client实例赋值。尽管c可以用来访问Callback（）方法，它不能访问Client类中的任何其他成员。一个接口引用变量仅仅知道被它的接口定义声明的方法。因此，c不能用来访问nonIfaceMeth( )，因为它是被Client定义的，而不是由Callback定义。

前面的例子机械的显示了一个接口引用变量怎样访问一个实现对象，它没有说明这样的引用的多态功能。为演示这个用途，首先创建Callback的第二个实现，如下：

```
// Another implementation of Callback.
class AnotherClient implements Callback {
  // Implement Callback's interface
  public void callback(int p) {
    System.out.println("Another version of callback");
    System.out.println("p squared is " + (p*p));
  }
}
```

现在，试试下面的类：

```
class TestIface2 {
  public static void main(String args[]) {
    Callback c = new Client();
    AnotherClient ob = new AnotherClient();

    c.callback(42);

    c = ob; // c now refers to AnotherClient object
    c.callback(42);
  }
}
```

程序输出如下：

```
callback called with 42
Another version of callback
p squared is 1764
```

如你所见，被调用的callback( )的形式由在运行时c引用的对象类型决定。这是一个非常简单的例子，下面你将会看到另一个例子，它更实用。

局部实现

如果一个类包含一个接口但是不完全实现接口定义的方法，那么该类必须定义成abstract型。例如：

```
abstract class Incomplete implements Callback {
  int a, b;
  void show() {
    System.out.println(a + " " + b);
  }
  // ...
}
```

这里，类Incomplete没有实现callback( )方法，必须定义成抽象类。任何继承Incomplete的类都必须实现callback( )方法或者它自己定义成abstract类。

9.4.3　应用接口

为理解接口的功能，让我们看一个更实际的例子。我们曾开发过一个名为Stack的类，该类实现了一个简单的固定大小的堆栈。然而，有很多方法可以实现堆栈。例如，堆栈的大小可以固定也可以不固定。堆栈还可以保存在数组、链表和二进制树中等。无论堆栈怎样实现，堆栈的接口保持不变。也就是说，push( )和pop( )方法定义了独立实现细节的堆栈的接口。因为堆栈的接口与它的实现是分离的，很容易定义堆栈接口，而不用管每个定义实现细节。让我们看下面的两个例子。

首先，下面定义了一个整数堆栈接口，把它保存在一个IntStack.java文件中。该接口将被两个堆栈实现使用。

```
// Define an integer stack interface.
interface IntStack {
  void push(int item); // store an item
  int pop(); // retrieve an item
}
```

下面的程序创建了一个名为FixedStack的类，该类实现一个固定长度的整数堆栈：

```
// An implementation of IntStack that uses fixed storage.
class FixedStack implements IntStack {
  private int stck[];
  private int tos;

  // allocate and initialize stack
  FixedStack(int size) {
    stck = new int[size];
    tos = -1;
  }

  // Push an item onto the stack
  public void push(int item) {
    if(tos==stck.length-1) // use length member
      System.out.println("Stack is full.");
    else
      stck[++tos] = item;
  }
  // Pop an item from the stack
  public int pop() {
    if(tos < 0) {
      System.out.println("Stack underflow.");
      return 0;
    }
    else
      return stck[tos--];
  }
}

class IFTest {
  public static void main(String args[]) {
    FixedStack mystack1 = new FixedStack(5);
    FixedStack mystack2 = new FixedStack(8);
```

```
    // push some numbers onto the stack
    for(int i=0; i<5; i++) mystack1.push(i);
    for(int i=0; i<8; i++) mystack2.push(i);

    // pop those numbers off the stack
    System.out.println("Stack in mystack1:");
    for(int i=0; i<5; i++)
      System.out.println(mystack1.pop());

    System.out.println("Stack in mystack2:");
    for(int i=0; i<8; i++)
      System.out.println(mystack2.pop());
  }
}
```

下面是IntStack 的另一个实现。通过运用相同的接口定义IntStack 创建了一个动态堆栈。这种实现中，每一个栈都以一个初始长度建造。如果初始化长度被超出，那么堆栈的大小将增加。每一次需要更多的空间，堆栈的大小成倍增长。

```
// Implement a "growable" stack.
class DynStack implements IntStack {
  private int stck[];
  private int tos;
  // allocate and initialize stack
  DynStack(int size) {
    stck = new int[size];
    tos = -1;
  }

  // Push an item onto the stack
  public void push(int item) {
    // if stack is full, allocate a larger stack
    if(tos==stck.length-1) {
      int temp[] = new int[stck.length * 2]; // double size
      for(int i=0; i<stck.length; i++) temp[i] = stck[i];
      stck = temp;
      stck[++tos] = item;
    }
    else
      stck[++tos] = item;
  }

  // Pop an item from the stack
  public int pop() {
    if(tos < 0) {
      System.out.println("Stack underflow.");
      return 0;
    }
    else
      return stck[tos--];
  }
}
```

```
class IFTest2 {
  public static void main(String args[]) {
    DynStack mystack1 = new DynStack(5);
    DynStack mystack2 = new DynStack(8);

    // these loops cause each stack to grow
    for(int i=0; i<12; i++) mystack1.push(i);
    for(int i=0; i<20; i++) mystack2.push(i);

    System.out.println("Stack in mystack1:");
    for(int i=0; i<12; i++)
      System.out.println(mystack1.pop());
    System.out.println("Stack in mystack2:");
    for(int i=0; i<20; i++)
      System.out.println(mystack2.pop());
  }
}
```

下面的类运用了FixedStack 和DynStack 实现。它通过一个接口引用完成。意思是说对push( ) 和 pop( )的调用在运行时解决而不是在编译时解决。

```
/* Create an interface variable and
   access stacks through it.
*/
class IFTest3 {
  public static void main(String args[]) {
    IntStack mystack; // create an interface reference variable
    DynStack ds = new DynStack(5);
    FixedStack fs = new FixedStack(8);

    mystack = ds; // load dynamic stack
    // push some numbers onto the stack
    for(int i=0; i<12; i++) mystack.push(i);

    mystack = fs; // load fixed stack
    for(int i=0; i<8; i++) mystack.push(i);


    mystack = ds;
    System.out.println("Values in dynamic stack:");
    for(int i=0; i<12; i++)
      System.out.println(mystack.pop());

    mystack = fs;
    System.out.println("Values in fixed stack:");
    for(int i=0; i<8; i++)
      System.out.println(mystack.pop());
  }
}
```

该程序中，mystack是IntStack接口的一个引用。因此，当它引用ds时，它使用DynStack实现所定义的push( )和pop( )方法。当它引用fs时，它使用FixedStack定义的push( )和pop( )方法。已经解释过，这些决定是在运行时做出的。通过接口引用变量获得接口的多重实现

是Java完成运行时多态的最有力的方法。

9.4.4　接口中的变量

你可以使用接口来引入多个类的共享常量，这样做只需要简单的声明包含变量初始化想要的值的接口就可以了。如果你的一个类中包含那个接口（就是说当你实现了接口时），所有的这些变量名都将作为常量看待。这与在C/C++中用头文件来创建大量的 #defined 常量或const 声明相似。如果接口不包含方法，那么任何包含这样接口的类实际并不实现什么。这就像类在类名字空间引入这些常量作final变量。下面的例子运用了这种技术来实现一个自动的“作决策者”：

```
import java.util.Random;

interface SharedConstants {
  int NO = 0;
  int YES = 1;
  int MAYBE = 2;
  int LATER = 3;
  int SOON = 4;
  int NEVER = 5;
}

class Question implements SharedConstants {
  Random rand = new Random();
  int ask() {
    int prob = (int) (100 * rand.nextDouble());
    if (prob < 30)
      return NO;          // 30%
    else if (prob < 60)
      return YES;         // 30%
    else if (prob < 75)
      return LATER;       // 15%
    else if (prob < 98)
      return SOON;        // 13%
    else
      return NEVER;       // 2%
  }
}

class AskMe implements SharedConstants {
  static void answer(int result) {
    switch(result) {
      case NO:
        System.out.println("No");
        break;
      case YES:
        System.out.println("Yes");
        break;
      case MAYBE:
        System.out.println("Maybe");
        break;
      case LATER:
```

```
        System.out.println("Later");
        break;
      case SOON:
        System.out.println("Soon");
        break;
      case NEVER:
        System.out.println("Never");
        break;
    }
  }

  public static void main(String args[]) {
    Question q = new Question();
    answer(q.ask());
    answer(q.ask());
    answer(q.ask());
    answer(q.ask());
  }
}
```

注意该程序利用了Java的一个标准类：Random，该类提供伪随机数。它包含若干个方法。通过这些方法你可以获得你程序所需形式的随机数。该例中，用到了nextDouble( )方法。它返回0.0到1.0之间的随机数。

该例子程序中，定义了两个类Question和AskMe。这两个类都实现了SharedConstants接口。该接口中定义了NO、YES、MAYBE、SOON、LATER和 NEVER。每个类中，代码就像自己定义或继承了它们一样直接引用了这些变量。下面是该程序的输出示例。注意每次运行结果不同。

```
Later
Soon
No
Yes
```

9.4.5　接口可以扩展

接口可以通过运用关键字extends被其他接口继承。语法与继承类是一样的。当一个类实现一个继承了另一个接口的接口时，它必须实现接口继承链表中定义的所有方法。下面是一个例子：

```
// One interface can extend another.
interface A {
  void meth1();
  void meth2();
}

// B now includes meth1() and meth2() -- it adds meth3().
interface B extends A {
  void meth3();
}

// This class must implement all of A and B
class MyClass implements B {
```

```
  public void meth1() {
    System.out.println("Implement meth1().");
  }

  public void meth2() {
    System.out.println("Implement meth2().");
  }

  public void meth3() {
    System.out.println("Implement meth3().");
  }

}

class IFExtend {
  public static void main(String arg[]) {
    MyClass ob = new MyClass();

    ob.meth1();
    ob.meth2();
    ob.meth3();
  }
}
```

作为一个实验你也许希望移走MyClass中meth1( )的实现。这将导致编译时错误。前面讲过，任何实现接口的类必须实现该接口定义的所有方法，包括从其他接口继承的任何方法。

尽管我们在本书中包括的例子没有很频繁的用到包和接口，这两个工具是Java编程环境中的重要部分。实质上所有用Java编写的实际的程序和小应用程序都被包含在包中。一个数字也可能实现接口。因此，游刃有余的运用这些工具是非常有用的。

第 10 章　异 常 处 理

本章介绍Java的异常处理机制。异常(exception)是在运行时代码序列中产生一种异常情况。换句话说，异常是一个运行时错误。在不支持异常处理的计算机语言中，错误必须被手工的检查和处理——典型的是通过错误代码的运用等等。这种方法既很笨拙也很麻烦。Java的异常处理避免了这些问题，而且在处理过程中，把运行时错误的管理带到了面向对象的世界。

10.1　异常处理基础

Java异常是一个描述在代码段中发生的异常（也就是出错）情况的对象。当异常情况发生，一个代表该异常的对象被创建并且在导致该错误的方法中被引发（throw）。该方法可以选择自己处理异常或传递该异常。两种情况下，该异常被捕获（caught）并处理。异常可能是由Java运行时系统产生，或者是由你的手工代码产生。被Java引发的异常与违反语言规范或超出Java执行环境限制的基本错误有关。手工编码产生的异常基本上用于报告方法调用程序的出错状况。

Java异常处理通过5个关键字控制：try、catch、throw、throws和 finally。下面讲述它们如何工作的。程序声明了你想要的异常监控包含在一个try块中。如果在try块中发生异常，它被抛出。你的代码可以捕捉这个异常（用catch）并且用某种合理的方法处理该异常。系统产生的异常被Java运行时系统自动引发。手动引发一个异常，用关键字throw。任何被引发方法的异常都必须通过throws子句定义。任何在方法返回前绝对被执行的代码被放置在finally块中。

下面是一个异常处理块的通常形式：

```
try {
  // block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
           // exception handler for ExceptionType1
       }
catch (ExceptionType2 exOb) {
           // exception handler for ExceptionType2
       }
       // ...
       finally {
        // block of code to be executed before try block ends
      }
```

这里，ExceptionType 是发生异常的类型。下面将介绍怎样应用这个框架。

10.2　异 常 类 型

所有异常类型都是内置类Throwable的子类。因此，Throwable在异常类层次结构的顶层。紧接着Throwable下面的是两个把异常分成两个不同分支的子类。一个分支是Exception。该类用于用户程序可能捕捉的异常情况。它也是你可以用来创建你自己用户异常类型子类的类。在Exception分支中有一个重要子类RuntimeException。该类型的异常自动为你所编写的程序定义并且包括被零除和非法数组索引这样的错误。

另一类分支由Error作为顶层，Error定义了在通常环境下不希望被程序捕获的异常。Error类型的异常用于Java运行时系统来显示与运行时系统本身有关的错误。堆栈溢出是这种错误的一例。本章将不讨论关于Error类型的异常处理，因为它们通常是灾难性的致命错误，不是你的程序可以控制的。

10.3　未被捕获的异常

在你学习在程序中处理异常之前，看一看如果你不处理它们会有什么情况发生是很有好处的。下面的小程序包括一个故意导致被零除错误的表达式。

```
class Exc0 {
  public static void main(String args[]) {
    int d = 0;
    int a = 42 / d;
  }
}
```

当Java运行时系统检查到被零除的情况，它构造一个新的异常对象然后引发该异常。这导致Exc0的执行停止，因为一旦一个异常被引发，它必须被一个异常处理程序捕获并且被立即处理。该例中，我们没有提供任何我们自己的异常处理程序，所以异常被Java运行时系统的默认处理程序捕获。任何不是被你程序捕获的异常最终都会被该默认处理程序处理。默认处理程序显示一个描述异常的字符串，打印异常发生处的堆栈轨迹并且终止程序。

下面是由标准javaJDK运行时解释器执行该程序所产生的输出：

```
java.lang.ArithmeticException: / by zero
        at Exc0.main(Exc0.java:4)
```

注意，类名Exc0，方法名main，文件名Exc0.java和行数4是怎样被包括在一个简单的堆栈使用轨迹中的。还有，注意引发的异常类型是Exception的一个名为ArithmeticException的子类，该子类更明确的描述了何种类型的错误方法。本章后面部分将讨论，Java提供多个内置的与可能产生的不同种类运行时错误相匹配的异常类型。

堆栈轨迹将显示导致错误产生的方法调用序列。例如，下面是前面程序的另一个版本，它介绍了相同的错误，但是错误是在main( )方法之外的另一个方法中产生的：

```
class Exc1 {
```

```
  static void subroutine() {
    int d = 0;
    int a = 10 / d;
  }
  public static void main(String args[]) {
    Exc1.subroutine();
  }
}
```

默认异常处理器的堆栈轨迹结果表明了整个调用栈是怎样显示的：

```
java.lang.ArithmeticException: / by zero
     at Exc1.subroutine(Exc1.java:4)
     at Exc1.main(Exc1.java:7)
```

如你所见，栈底是main的第7行，该行调用了subroutine( )方法。该方法在第4行导致了异常。调用堆栈对于调试来说是很重要的，因为它查明了导致错误的精确的步骤。

10.4 使用try和catch

尽管由Java运行时系统提供的默认异常处理程序对于调试是很有用的，但通常你希望自己处理异常。这样做有两个好处。第一，它允许你修正错误。第二，它防止程序自动终止。大多数用户对于在程序终止运行和在无论何时错误发生都会打印堆栈轨迹感到很烦恼（至少可以这么说）。幸运的是，这很容易避免。

为防止和处理一个运行时错误，只需要把你所要监控的代码放进一个try块就可以了。紧跟着try块的，包括一个说明你希望捕获的错误类型的catch子句。完成这个任务很简单，下面的程序包含一个处理因为被零除而产生的ArithmeticException 异常的try块和一个catch子句。

```
class Exc2 {
  public static void main(String args[]) {
    int d, a;
  try { // monitor a block of code.
     d = 0;
     a = 42 / d;
     System.out.println("This will not be printed.");
    } catch (ArithmeticException e) { // catch divide-by-zero error
     System.out.println("Division by zero.");
    }
    System.out.println("After catch statement.");
  }
}
```

该程序输出如下：

```
Division by zero.
After catch statement.
```

注意在try块中的对println( )的调用是永远不会执行的。一旦异常被引发，程序控制由try

块转到catch块。执行永远不会从catch块“返回”到try块。因此，“This will not be printed。”将不会被显示。一旦执行了catch语句，程序控制从整个try/catch机制的下面一行继续。

一个try和它的catch语句形成了一个单元。catch子句的范围限制于try语句前面所定义的语句。一个catch语句不能捕获另一个try声明所引发的异常（除非是嵌套的try语句情况）。被try保护的语句声明必须在一个大括号之内（也就是说，它们必须在一个块中）。你不能单独使用try。

构造catch子句的目的是解决异常情况并且像错误没有发生一样继续运行。例如，下面的程序中，每一个for循环的反复得到两个随机整数。这两个整数分别被对方除，结果用来除12345。最后的结果存在a中。如果一个除法操作导致被零除错误，它将被捕获，a的值设为零，程序继续运行。

```
// Handle an exception and move on.
import java.util.Random;

class HandleError {
  public static void main(String args[]) {
    int a=0, b=0, c=0;
    Random r = new Random();

    for(int i=0; i<32000; i++) {
      try {
        b = r.nextInt();
        c = r.nextInt();
        a = 12345 / (b/c);
      } catch (ArithmeticException e) {
        System.out.println("Division by zero.");
        a = 0; // set a to zero and continue
      }
      System.out.println("a: " + a);
    }
  }
}
```

10.4.1　显示一个异常的描述

Throwable重载toString( )方法（由Object定义），所以它返回一个包含异常描述的字符串。你可以通过在println( )中传给异常一个参数来显示该异常的描述。例如，前面程序的catch块可以被重写成

```
catch (ArithmeticException e) {
  System.out.println("Exception: " + e);
  a = 0; // set a to zero and continue
}
```

当这个版本代替原程序中的版本，程序在标准javaJDK解释器下运行，每一个被零除错误显示下面的消息：

```
Exception: java.lang.ArithmeticException: / by zero
```

尽管在上下文中没有特殊的值，显示一个异常描述的能力在其他情况下是很有价值的

——特别是当你对异常进行实验和调试时。

10.5 使用多重catch 语句

某些情况，由单个代码段可能引起多个异常。处理这种情况，你可以定义两个或更多的catch子句，每个子句捕获一种类型的异常。当异常被引发时，每一个catch子句被依次检查，第一个匹配异常类型的子句执行。当一个catch语句执行以后，其他的子句被旁路，执行从try/catch块以后的代码开始继续。下面的例子设计了两种不同的异常类型：

```
// Demonstrate multiple catch statements.
class MultiCatch {
  public static void main(String args[]) {
    try {
      int a = args.length;
      System.out.println("a = " + a);
      int b = 42 / a;
      int c[] = { 1 };
      c[42] = 99;
    } catch(ArithmeticException e) {
      System.out.println("Divide by 0: " + e);
    } catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("Array index oob: " + e);
    }
    System.out.println("After try/catch blocks.");
  }
}
```

该程序在没有命令行参数的起始条件下运行导致被零除异常，因为a为0。如果你提供一个命令行参数，它将幸免于难，把a设成大于零的数值。但是它将导致ArrayIndexOutOfBoundsException异常，因为整型数组c的长度为1，而程序试图给c[42]赋值。

下面是运行在两种不同情况下程序的输出：

```
C:\>java MultiCatch
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.
C:\>java MultiCatch TestArg
a = 1
Array index oob: java.lang.ArrayIndexOutOfBoundsException
After try/catch blocks.
```

当你用多catch语句时，记住异常子类必须在它们任何父类之前使用是很重要的。这是因为运用父类的catch语句将捕获该类型及其所有子类类型的异常。这样，如果子类在父类后面，子类将永远不会到达。而且，Java中不能到达的代码是一个错误。例如，考虑下面的程序：

```
/* This program contains an error.

   A subclass must come before its superclass in
```

```
   a series of catch statements. If not,
   unreachable code will be created and a
   compile-time error will result.
*/
class SuperSubCatch {
  public static void main(String args[]) {
    try {
      int a = 0;
      int b = 42 / a;
    } catch(Exception e) {
      System.out.println("Generic Exception catch.");
    }
    /* This catch is never reached because
       ArithmeticException is a subclass of Exception. */
    catch(ArithmeticException e) { // ERROR - unreachable
      System.out.println("This is never reached.");
    }
  }
}
```

如果你试着编译该程序，你会收到一个错误消息，该错误消息说明第二个catch语句不会到达，因为该异常已经被捕获。因为ArithmeticException 是Exception的子类，第一个catch语句将处理所有的面向Exception的错误，包括ArithmeticException。这意味着第二个catch语句永远不会执行。为修改程序，颠倒两个catch语句的次序。

10.6　嵌套try语句

Try语句可以被嵌套。也就是说，一个try语句可以在另一个try块内部。每次进入try语句，异常的前后关系都会被推入堆栈。如果一个内部的try语句不含特殊异常的catch处理程序，堆栈将弹出，下一个try语句的catch处理程序将检查是否与之匹配。这个过程将继续直到一个catch语句匹配成功，或者是直到所有的嵌套try语句被检查耗尽。如果没有catch语句匹配，Java的运行时系统将处理这个异常。下面是运用嵌套try语句的一个例子：

```
// An example of nested try statements.
class NestTry {
  public static void main(String args[]) {
    try {
      int a = args.length;

      /* If no command-line args are present,
         the following statement will generate
         a divide-by-zero exception. */
      int b = 42 / a;

      System.out.println("a = " + a);

      try { // nested try block
```

```
        /* If one command-line arg is used,
           then a divide-by-zero exception
           will be generated by the following code. */
        if(a==1) a = a/(a-a); // division by zero

        /* If two command-line args are used,
           then generate an out-of-bounds exception. */
        if(a==2) {
          int c[] = { 1 };
          c[42] = 99; // generate an out-of-bounds exception
        }
      } catch(ArrayIndexOutOfBoundsException e) {
        System.out.println("Array index out-of-bounds: " + e);
      }

    } catch(ArithmeticException e) {
      System.out.println("Divide by 0: " + e);
    }
  }
}
```

如你所见，该程序在一个try块中嵌套了另一个try块。程序工作如下：当你在没有命令行参数的情况下执行该程序，外面的try块将产生一个被零除的异常。程序在有一个命令行参数条件下执行，由嵌套的try块产生一个被零除的错误。因为内部的块不匹配这个异常，它将把异常传给外部的try块，在那里异常被处理。如果你在具有两个命令行参数的条件下执行该程序，由内部try块产生一个数组边界异常。下面的结果阐述了每一种情况：

```
C:\>java NestTry
Divide by 0: java.lang.ArithmeticException: / by zero
C:\>java NestTry One
a = 1
Divide by 0: java.lang.ArithmeticException: / by zero
C:\>java NestTry One Two
a = 2
Array index out-of-bounds: java.lang.ArrayIndexOutOfBoundsException
```

当有方法调用时，try语句的嵌套可以很隐蔽的发生。例如，你可以把对方法的调用放在一个try块中。在该方法内部，有另一个try语句。这种情况下，方法内部的try仍然是嵌套在外部调用该方法的try块中的。下面是前面例子的修改，嵌套的try块移到了方法nesttry( )的内部：

```
/* Try statements can be implicitly nested via
   calls to methods. */
class MethNestTry {
  static void nesttry(int a) {
    try { // nested try block
      /* If one command-line arg is used,
```

```
        then a divide-by-zero exception
        will be generated by the following code. */
      if(a==1) a = a/(a-a); // division by zero

      /* If two command-line args are used,
        then generate an out-of-bounds exception. */
      if(a==2) {
       int c[] = { 1 };
       c[42] = 99; // generate an out-of-bounds exception
      }
    } catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("Array index out-of-bounds: " + e);
    }
  }

  public static void main(String args[]) {
    try {
      int a = args.length;

      /* If no command-line args are present,
        the following statement will generate
        a divide-by-zero exception. */
      int b = 42 / a;
      System.out.println("a = " + a);

      nesttry(a);
    } catch(ArithmeticException e) {
      System.out.println("Divide by 0: " + e);
    }
  }
}
```

该程序的输出与前面的例子相同。

10.7　引发（throw）

到目前为止，你只是获取了被Java运行时系统引发的异常。然而，程序可以用throw语句引发明确的异常。Throw语句的通常形式如下：

```
throw ThrowableInstance;
```

这里，ThrowableInstance一定是Throwable类类型或Throwable子类类型的一个对象。简单类型，例如int或char，以及非Throwable类，例如String或Object，不能用作异常。有两种可以获得Throwable对象的方法：在catch子句中使用参数或者用new操作符创建。

程序执行在throw语句之后立即停止；后面的任何语句不被执行。最紧紧包围的try块用来检查它是否含有一个与异常类型匹配的catch语句。如果发现了匹配的块，控制转向该语句；如果没有发现，次包围的try块来检查，以此类推。如果没有发现匹配的catch块，默认异常处理程序中断程序的执行并且打印堆栈轨迹。

下面是一个创建并引发异常的例子程序，与异常匹配的处理程序再把它引发给外层的处理程序。

```
// Demonstrate throw.
class ThrowDemo {
  static void demoproc() {
    try {
      throw new NullPointerException("demo");
    } catch(NullPointerException e) {
      System.out.println("Caught inside demoproc.");
      throw e; // rethrow the exception
    }
  }

  public static void main(String args[]) {
    try {
      demoproc();
    } catch(NullPointerException e) {
      System.out.println("Recaught: " + e);
    }
  }
}
```

该程序有两个机会处理相同的错误。首先，main（）设立了一个异常关系然后调用demoproc( )。demoproc( )方法然后设立了另一个异常处理关系并且立即引发一个新的NullPointerException实例，NullPointerException在下一行被捕获。异常于是被再次引发。下面是输出结果：

```
Caught inside demoproc.
Recaught: java.lang.NullPointerException: demo
```

该程序还阐述了怎样创建Java的标准异常对象，特别注意下面这一行：

```
throw new NullPointerException("demo");
```

这里，new用来构造一个NullPointerException实例。所有的Java内置的运行时异常有两个构造函数：一个没有参数，一个带有一个字符串参数。当用到第二种形式时，参数指定描述异常的字符串。如果对象用作 print( )或println( )的参数时，该字符串被显示。这同样可以通过调用getMessage( )来实现，getMessage( )是由Throwable定义的。

10.8 throws

如果一个方法可以导致一个异常但不处理它，它必须指定这种行为以使方法的调用者可以保护它们自己而不发生异常。做到这点你可以在方法声明中包含一个throws子句。一

个 throws 子句列举了一个方法可能引发的所有异常类型。这对于除 Error 或 RuntimeException及它们子类以外类型的所有异常是必要的。一个方法可以引发的所有其他类型的异常必须在throws子句中声明。如果不这样做，将会导致编译错误。

下面是包含一个throws子句的方法声明的通用形式：

```
type method-name(parameter-list) throws exception-list
{
  // body of method
}
```

这里，exception-list是该方法可以引发的以有逗号分割的异常列表。

下面是一个不正确的例子。该例试图引发一个它不能捕获的异常。因为程序没有指定一个throws子句来声明这一事实，程序将不会编译。

```
// This program contains an error and will not compile.
class ThrowsDemo {
  static void throwOne() {
    System.out.println("Inside throwOne.");
    throw new IllegalAccessException("demo");
  }
  public static void main(String args[]) {
    throwOne();
  }
}
```

为编译该程序，需要改变两个地方。第一，需要声明throwOne( )引发IllegalAccess Exception异常。第二，main( )必须定义一个try/catch 语句来捕获该异常。

正确的例子如下:

```
// This is now correct.
class ThrowsDemo {
  static void throwOne() throws IllegalAccessException {
    System.out.println("Inside throwOne.");
    throw new IllegalAccessException("demo");
  }
  public static void main(String args[]) {
    try {
      throwOne();
    } catch (IllegalAccessException e) {
      System.out.println("Caught " + e);
    }
  }
}
```

下面是例题的输出结果：

```
inside throwOne
caught java.lang.IllegalAccessException: demo
```

10.9 finally

当异常被引发，通常方法的执行将作一个陡峭的非线性的转向。依赖于方法是怎样编码的，异常甚至可以导致方法过早返回。这在一些方法中是一个问题。例如，如果一个方法打开一个文件项并关闭，然后退出，你不希望关闭文件的代码被异常处理机制旁路。finally关键字为处理这种意外而设计。

finally创建一个代码块。该代码块在一个try/catch 块完成之后另一个try/catch出现之前执行。finally块无论有没有异常引发都会执行。如果异常被引发，finally甚至是在没有与该异常相匹配的catch子句情况下也将执行。一个方法将从一个try/catch块返回到调用程序的任何时候，经过一个未捕获的异常或者是一个明确的返回语句，finally子句在方法返回之前仍将执行。这在关闭文件句柄和释放任何在方法开始时被分配的其他资源是很有用的。finally子句是可选项，可以有也可以无。然而每一个try语句至少需要一个catch或finally子句。

下面的例子显示了3种不同的退出方法。每一个都执行了finally子句：

```
// Demonstrate finally.
class FinallyDemo {
  // Through an exception out of the method.
  static void procA() {
    try {
      System.out.println("inside procA");
      throw new RuntimeException("demo");
    } finally {
      System.out.println("procA's finally");
    }
  }

  // Return from within a try block.
  static void procB() {
    try {
      System.out.println("inside procB");
      return;
    } finally {
      System.out.println("procB's finally");
    }
  }
 // Execute a try block normally.
  static void procC() {
    try {
      System.out.println("inside procC");
    } finally {
      System.out.println("procC's finally");
    }
```

```
  }

  public static void main(String args[]) {
    try {
      procA();
    } catch (Exception e) {
      System.out.println("Exception caught");
    }
    procB();
    procC();
  }
}
```

该例中，procA( )过早地通过引发一个异常中断了try。Finally子句在退出时执行。procB( )的try语句通过一个return语句退出。在procB( )返回之前finally子句执行。在procC（）中，try语句正常执行，没有错误。然而，finally块仍将执行。

注意：如果finally块与一个try联合使用，finally块将在try 结束之前执行。

下面是上述程序产生的输出：

```
inside procA
procA’s finally
Exception caught
inside procB
procB’s finally
inside procC
procC’s finally
```

10.10　Java的内置异常

在标准包java.lang中，Java定义了若干个异常类。前面的例子曾用到其中一些。这些异常一般是标准类RuntimeException的子类。因为java.lang实际上被所有的Java程序引入，多数从RuntimeException派生的异常都自动可用。而且，它们不需要被包含在任何方法的throws列表中。Java语言中，这被叫做未经检查的异常（unchecked exceptions ）。因为编译器不检查它来看一个方法是否处理或引发了这些异常。 java.lang中定义的未经检查的异常列于表10-1。表10-2列出了由 java.lang定义的必须在方法的throws列表中包括的异常，如果这些方法能产生其中的某个异常但是不能自己处理它。这些叫做受检查的异常（checked exceptions）。Java定义了几种与不同类库相关的其他的异常类型。

表 10-1　Java 的 java.lang 中定义的未检查异常子类

异常	说明
ArithmeticException	算术错误，如被0除
ArrayIndexOutOfBoundsException	数组下标出界

续表

异常	说明
ArrayStoreException	数组元素赋值类型不兼容
ClassCastException	非法强制转换类型
IllegalArgumentException	调用方法的参数非法
IllegalMonitorStateException	非法监控操作，如等待一个未锁定线程
IllegalStateException	环境或应用状态不正确
IllegalThreadStateException	请求操作与当前线程状态不兼容
IndexOutOfBoundsException	某些类型索引越界
NullPointerException	非法使用空引用
NumberFormatException	字符串到数字格式非法转换
SecurityException	试图违反安全性
StringIndexOutOfBounds	试图在字符串边界之外索引
UnsupportedOperationException	遇到不支持的操作

表 10-2 java.lang 中定义的检查异常

异常	意义
ClassNotFoundException	找不到类
CloneNotSupportedException	试图克隆一个不能实现Cloneable接口的对象
IllegalAccessException	对一个类的访问被拒绝
InstantiationException	试图创建一个抽象类或者抽象接口的对象
InterruptedException	一个线程被另一个线程中断
NoSuchFieldException	请求的字段不存在
NoSuchMethodException	请求的方法不存在

10.11 创建自己的异常子类

尽管Java的内置异常处理大多数常见错误，你也许希望建立你自己的异常类型来处理你所应用的特殊情况。这是非常简单的：只要定义Exception的一个子类就可以了（Exception当然是Throwable的一个子类）。你的子类不需要实际执行什么——它们在类型系统中的存在允许你把它们当成异常使用。

Exception类自己没有定义任何方法。当然，它继承了Throwable提供的一些方法。因此，所有异常，包括你创建的，都可以获得Throwable定义的方法。这些方法显示在表10-3中。你还可以在你创建的异常类中覆盖一个或多个这样的方法。

表 10-3 Throwable 定义的方法

方法	描述
Throwable fillInStackTrace( )	返回一个包含完整堆栈轨迹的Throwable对象，该对象可能被再次引发

续表

方法	描述
String getLocalizedMessage( )	返回一个异常的局部描述
String getMessage( )	返回一个异常的描述
void printStackTrace( )	显示堆栈轨迹
void printStackTrace(PrintStreamstream)	把堆栈轨迹送到指定的流
void printStackTrace(PrintWriterstream)	把堆栈轨迹送到指定的流
String toString( )	返回一个包含异常描述的String对象。当输出一个Throwable对象时，该方法被println( )调用

下面的例子声明了Exception的一个新子类，然后该子类当作方法中出错情形的信号。它重载了toString( )方法，这样可以用println( )显示异常的描述。

```
// This program creates a custom exception type.
class MyException extends Exception {
  private int detail;

  MyException(int a) {
    detail = a;
  }

  public String toString() {
    return "MyException[" + detail + "]";
  }
}

class ExceptionDemo {
  static void compute(int a) throws MyException {
    System.out.println("Called compute(" + a + ")");
    if(a > 10)
      throw new MyException(a);
    System.out.println("Normal exit");
  }

  public static void main(String args[]) {
    try {
      compute(1);
      compute(20);
    } catch (MyException e) {
      System.out.println("Caught " + e);
    }
  }
}
```

该例题定义了Exception的一个子类MyException。该子类非常简单：它只含有一个构造函数和一个重载的显示异常值的toString( )方法。ExceptionDemo类定义了一个compute( )方法。该方法引发一个MyException对象。当compute( )的整型参数比10大时该异常被引发。main( )方法为MyException设立了一个异常处理程序，然后用一个合法的值和不合法的值调用compute( )来显示执行经过代码的不同路径。下面是结果：

```
Called compute(1)
Normal exit
Called compute(20)
Caught MyException[20]
```

10.12 使 用 异 常

异常处理为控制具有很多动态运行时特性的复杂程序提供了一个强大的机制。把try，throw，和catch当成处理错误简洁及程序逻辑上的反常边界条件是很重要的。如果你像多数程序员一样，那么你可能习惯于在方法失败时返回一个错误代码。在你用Java编程时，你应该打破这个习惯。当方法可能失败时，引发一个异常。这是处理失败模式的一个更简洁的方法。

最后说明一点：Java的异常处理语句不应该被当作是一个非本地分支的通常机制，如果你这样认为，它将困扰你的代码并使代码难于维护。

第 11 章　多线程编程

和其他多数计算机语言不同，Java内置支持多线程编程（multithreaded programming。多线程程序包含两条或两条以上并发运行的部分。程序中每个这样的部分都叫一个线程（thread），每个线程都有独立的执行路径。因此，多线程是多任务处理的一种特殊形式。

你一定知道多任务处理，因为它实际上被所有的现代操作系统所支持。然而，多任务处理有两种截然不同的类型：基于进程的和基于线程的。认识两者的不同是十分重要的。对很多读者，基于进程的多任务处理是更熟悉的形式。进程(process)本质上是一个执行的程序。因此，基于进程(process-based) 的多任务处理的特点是允许你的计算机同时运行两个或更多的程序。举例来说，基于进程的多任务处理使你在运用文本编辑器的时候可以同时运行Java编译器。在基于进程的多任务处理中，程序是调度程序所分派的最小代码单位。

在基于线程(thread-based) 的多任务处理环境中，线程是最小的执行单位。这意味着一个程序可以同时执行两个或者多个任务的功能。例如，一个文本编辑器可以在打印的同时格式化文本。所以，多进程程序处理“大图片”，而多线程程序处理细节问题。

多线程程序比多进程程序需要更少的管理费用。进程是重量级的任务，需要分配它们自己独立的地址空间。进程间通信是昂贵和受限的。进程间的转换也是很需要花费的。另一方面，线程是轻量级的选手。它们共享相同的地址空间并且共同分享同一个进程。线程间通信是便宜的，线程间的转换也是低成本的。当Java程序使用多进程任务处理环境时，多进程程序不受Java的控制，而多线程则受Java控制。

多线程帮助你写出CPU最大利用率的高效程序，因为空闲时间保持最低。这对Java运行的交互式的网络互连环境是至关重要的，因为空闲时间是公共的。举个例子来说，网络的数据传输速率远低于计算机处理能力，本地文件系统资源的读写速度远低于CPU的处理能力，当然，用户输入也比计算机慢很多。在传统的单线程环境中，你的程序必须等待每一个这样的任务完成以后才能执行下一步——尽管CPU有很多空闲时间。多线程使你能够获得并充分利用这些空闲时间。

如果你在Windows 98 或Windows 2000这样的操作系统下有编程经验，那么你已经熟悉了多线程。然而，Java管理线程使多线程处理尤其方便，因为很多细节对你来说是易于处理的。

11.1　Java线程模型

Java运行系统在很多方面依赖于线程，所有的类库设计都考虑到多线程。实际上，Java使用线程来使整个环境异步。这有利于通过防止CPU循环的浪费来减少无效部分。

为更好的理解多线程环境的优势可以将它与它的对照物相比较。单线程系统的处理途径是使用一种叫作轮询的事件循环方法。在该模型中，单线程控制在一无限循环中运行，

轮询一个事件序列来决定下一步做什么。一旦轮询装置返回信号表明，已准备好读取网络文件，事件循环调度控制管理到适当的事件处理程序。直到事件处理程序返回，系统中没有其他事件发生。这就浪费了CPU时间。这导致了程序的一部分独占了系统，阻止了其他事件的执行。总的来说，单线程环境，当一个线程因为等待资源时阻塞（block，挂起执行），整个程序停止运行。

Java多线程的优点在于取消了主循环/轮询机制。一个线程可以暂停而不影响程序的其他部分。例如，当一个线程从网络读取数据或等待用户输入时产生的空闲时间可以被利用到其他地方。多线程允许活的循环在每一帧间隙中沉睡一秒而不暂停整个系统。在Java程序中出现线程阻塞，仅有一个线程暂停，其他线程继续运行。

线程存在于好几种状态。线程可以正在运行（running）。只要获得CPU时间它就可以运行。运行的线程可以被挂起（suspend），并临时中断它的执行。一个挂起的线程可以被恢复（resume，允许它从停止的地方继续运行。一个线程可以在等待资源时被阻塞（block）。在任何时候，线程可以终止（terminate），这立即中断了它的运行。一旦终止，线程不能被恢复。

11.1.1　线程优先级

Java给每个线程安排优先级以决定与其他线程比较时该如何对待该线程。线程优先级是详细说明线程间优先关系的整数。作为绝对值，优先级是毫无意义的；当只有一个线程时，优先级高的线程并不比优先权低的线程运行的快。相反，线程的优先级是用来决定何时从一个运行的线程切换到另一个。这叫“上下文转换”(context switch)。决定上下文转换发生的规则很简单：

- 线程可以自动放弃控制。在I/O未决定的情况下，睡眠或阻塞由明确的让步来完成。在这种假定下，所有其他的线程被检测，准备运行的最高优先级线程被授予CPU。
- 线程可以被高优先级的线程抢占。在这种情况下，低优先级线程不主动放弃，处理器只是被先占——无论它正在干什么——处理器被高优先级的线程占据。基本上，一旦高优先级线程要运行，它就执行。这叫做有优先权的多任务处理。

当两个相同优先级的线程竞争CPU周期时，情形有一点复杂。对于Windows98这样的操作系统，等优先级的线程是在循环模式下自动划分时间的。对于其他操作系统，例如Solaris 2.x，等优先级线程相对于它们的对等体自动放弃。如果不这样，其他的线程就不会运行。

警告：不同的操作系统下等优先级线程的上下文转换可能会产生错误。

11.1.2　同步性

因为多线程在你的程序中引入了一个异步行为，所以在你需要的时候必须有加强同步性的方法。举例来说，如果你希望两个线程相互通信并共享一个复杂的数据结构，例如链表序列，你需要某些方法来确保它们没有相互冲突。也就是说，你必须防止一个线程写入数据而另一个线程正在读取链表中的数据。为此目的，Java在进程间同步性的老模式基础

上实行了另一种方法：管程（monitor）。管程是一种由C.A.R.Hoare首先定义的控制机制。你可以把管程想象成一个仅控制一个线程的小盒子。一旦线程进入管程，所有线程必须等待直到该线程退出了管程。用这种方法，管程可以用来防止共享的资源被多个线程操纵。

很多多线程系统把管程作为程序必须明确的引用和操作的对象。Java提供一个清晰的解决方案。没有“Monitor”类；相反，每个对象都拥有自己的隐式管程，当对象的同步方法被调用时管程自动载入。一旦一个线程包含在一个同步方法中，没有其他线程可以调用相同对象的同步方法。这就使你可以编写非常清晰和简洁的多线程代码，因为同步支持是语言内置的。

11.1.3　消息传递

在你把程序分成若干线程后，你就要定义各线程之间的联系。用大多数其他语言规划时，你必须依赖于操作系统来确立线程间通信。这样当然增加花费。然而，Java提供了多线程间谈话清洁的、低成本的途径——通过调用所有对象都有的预先确定的方法。Java的消息传递系统允许一个线程进入一个对象的一个同步方法，然后在那里等待，直到其他线程明确通知它出来。

11.1.4　Thread 类和Runnable 接口

Java的多线程系统建立于Thread类，它的方法，它的共伴接口Runnable基础上。Thread类封装了线程的执行。既然你不能直接引用运行着的线程的状态，你要通过它的代理处理它，于是Thread 实例产生了。为创建一个新的线程，你的程序必须扩展Thread 或实现Runnable接口。

Thread类定义了好几种方法来帮助管理线程。本章用到的方法如表11-1所示：

表 11-1　管理线程的方法

方法	意义
getName	获得线程名称
getPriority	获得线程优先级
jsAlive	判定线程是否仍在运行
join	等待一个线程终止
run	线程的入口点.
sleep	在一段时间内挂起线程
start	通过调用运行方法来启动线程

到目前为止，本书所应用的例子都是用单线程的。本章剩余部分解释如何用Thread 和Runnable 来创建、管理线程。让我们从所有Java程序都有的线程：主线程开始。

11.2　主 线 程

当Java程序启动时，一个线程立刻运行，该线程通常叫做程序的主线程（main thread），

因为它是程序开始时就执行的。主线程的重要性体现在两方面：

- 它是产生其他子线程的线程
- 通常它必须最后完成执行，因为它执行各种关闭动作。

尽管主线程在程序启动时自动创建，但它可以由一个Thread对象控制。为此，你必须调用方法currentThread()获得它的一个引用，currentThread()是Thread类的公有的静态成员。它的通常形式如下：

```
static Thread currentThread( )
```

该方法返回一个调用它的线程的引用。一旦你获得主线程的引用，你就可以像控制其他线程那样控制主线程。

让我们从复习下面例题开始：

```
// Controlling the main Thread.
class CurrentThreadDemo {
  public static void main(String args[]) {
    Thread t = Thread.currentThread();

    System.out.println("Current thread: " + t);

    // change the name of the thread
    t.setName("My Thread");
    System.out.println("After name change: " + t);

    try {
      for(int n = 5; n > 0; n--) {
        System.out.println(n);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Main thread interrupted");
    }
  }
}
```

在本程序中，当前线程（自然是主线程）的引用通过调用currentThread()获得，该引用保存在局部变量t中。然后，程序显示了线程的信息。接着程序调用setName()改变线程的内部名称。线程信息又被显示。然后，一个循环数从5开始递减，每数一次暂停一秒。暂停是由sleep()方法来完成的。Sleep()语句明确规定延迟时间是1毫秒。注意循环外的try/catch块。Thread类的sleep()方法可能引发一个InterruptedException异常。这种情形会在其他线程想要打搅沉睡线程时发生。本例只是打印了它是否被打断的消息。在实际的程序中，你必须灵活处理此类问题。下面是本程序的输出：

```
Current thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
5
4
3
```

```
2
1
```

注意t作为语句println()中参数运用时输出的产生。该显示顺序：线程名称，优先级以及组的名称。默认情况下，主线程的名称是main。它的优先级是5，这也是默认值，main也是所属线程组的名称。一个线程组（thread group）是一种将线程作为一个整体集合的状态控制的数据结构。这个过程由专有的运行时环境来处理，在此就不赘述了。线程名改变后，t又被输出。这次，显示了新的线程名。

让我们更仔细的研究程序中Thread类定义的方法。sleep()方法按照毫秒级的时间指示使线程从被调用到挂起。它的通常形式如下：

```
static void sleep(long milliseconds) throws InterruptedException
```

挂起的时间被明确定义为毫秒。该方法可能引发InterruptedException异常。

sleep()方法还有第二种形式，显示如下，该方法允许你指定时间是以毫秒还是以纳秒为周期。

```
static void sleep(long milliseconds, int nanoseconds) throws
InterruptedException
```

第二种形式仅当允许以纳秒为时间周期时可用。

如上述程序所示，你可以用setName()设置线程名称，用getName()来获得线程名称（该过程在程序中没有体现）。这些方法都是Thread 类的成员，声明如下：

```
final void setName(String threadName)
final String getName( )
```

这里，threadName 特指线程名称。

11.3　创 建 线 程

大多数情况，通过实例化一个Thread对象来创建一个线程。Java定义了两种方式：

- 实现Runnable 接口。
- 可以继承Thread类。

下面的两小节依次介绍了每一种方式。

11.3.1　实现Runnable接口

创建线程的最简单的方法就是创建一个实现Runnable 接口的类。Runnable抽象了一个执行代码单元。你可以通过实现Runnable接口的方法创建每一个对象的线程。为实现Runnable 接口，一个类仅需实现一个run()的简单方法，该方法声明如下：

```
public void run( )
```

在run()中可以定义代码来构建新的线程。理解下面内容是至关重要的：run()方法能够

像主线程那样调用其他方法，引用其他类，声明变量。仅有的不同是run()在程序中确立另一个并发的线程执行入口。当run()返回时，该线程结束。

在你已经创建了实现Runnable接口的类以后，你要在类内部实例化一个Thread类的对象。Thread 类定义了好几种构造函数。我们会用到的如下：

```
Thread(Runnable threadOb, String threadName)
```

该构造函数中，threadOb是一个实现Runnable接口类的实例。这定义了线程执行的起点。新线程的名称由threadName定义。

建立新的线程后，它并不运行直到调用了它的start()方法，该方法在Thread 类中定义。本质上，start() 执行的是一个对run()的调用。 Start（）方法声明如下：

```
void start( )
```

下面的例子是创建一个新的线程并启动它运行：

```
// Create a second thread.
class NewThread implements Runnable {
  Thread t;

  NewThread() {
    // Create a new, second thread
    t = new Thread(this, "Demo Thread");
    System.out.println("Child thread: " + t);
    t.start(); // Start the thread
  }

  // This is the entry point for the second thread.
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Child Thread: " + i);
        Thread.sleep(500);
      }
    } catch (InterruptedException e) {
      System.out.println("Child interrupted.");
    }
    System.out.println("Exiting child thread.");
  }
}

class ThreadDemo {
  public static void main(String args[]) {
    new NewThread(); // create a new thread

    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Main Thread: " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Main thread interrupted.");
    }
```

```
    System.out.println("Main thread exiting.");
  }
}
```

在NewThread 构造函数中，新的Thread对象由下面的语句创建：：

```
t = new Thread(this, "Demo Thread");
```

通过前面的语句this 表明在this对象中你想要新的线程调用run()方法。然后，start() 被调用，以run()方法为开始启动了线程的执行。这使子线程for 循环开始执行。调用start()之后，NewThread 的构造函数返回到main()。当主线程被恢复，它到达for 循环。两个线程继续运行，共享CPU，直到它们的循环结束。该程序的输出如下：

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

如前面提到的，在多线程程序中，通常主线程必须是结束运行的最后一个线程。实际上，一些老的JVM，如果主线程先于子线程结束，Java的运行时间系统就可能“挂起”。前述程序保证了主线程最后结束，因为主线程沉睡周期1000毫秒，而子线程仅为500毫秒。这就使子线程在主线程结束之前先结束。简而言之，你将看到等待线程结束的更好途径。

11.3.2　扩展Thread

创建线程的另一个途径是创建一个新类来扩展Thread类，然后创建该类的实例。当一个类继承Thread时，它必须重载run()方法，这个run()方法是新线程的入口。它也必须调用start()方法去启动新线程执行。下面用扩展thread类重写前面的程序：

```
// Create a second thread by extending Thread
class NewThread extends Thread {

  NewThread() {
    // Create a new, second thread
    super("Demo Thread");
    System.out.println("Child thread: " + this);
    start(); // Start the thread
  }

  // This is the entry point for the second thread.
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Child Thread: " + i);
```

```
        Thread.sleep(500);
      }
    } catch (InterruptedException e) {
      System.out.println("Child interrupted.");
    }
    System.out.println("Exiting child thread.");
  }
}

class ExtendThread {
  public static void main(String args[]) {
    new NewThread(); // create a new thread

    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Main Thread: " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Main thread interrupted.");
    }
    System.out.println("Main thread exiting.");
  }
}
```

该程序生成和前述版本相同的输出。子线程是由实例化NewThread对象生成的，该对象从Thread类派生。注意NewThread 中super()的调用。该方法调用了下列形式的Thread构造函数：

```
public Thread(String threadName)
```

这里，threadName指定线程名称。

11.3.3　选择合适方法

到这里，你一定会奇怪为什么Java有两种创建子线程的方法，哪一种更好呢。所有的问题都归于一点。Thread类定义了多种方法可以被派生类重载。对于所有的方法，惟一的必须被重载的是run()方法。这当然是实现Runnable接口所需的同样的方法。很多Java程序员认为类仅在它们被加强或修改时应该被扩展。因此，如果你不重载Thread的其他方法时，最好只实现Runnable 接口。这当然由你决定。然而，在本章的其他部分，我们应用实现runnable接口的类来创建线程。

11.4　创建多线程

到目前为止，我们仅用到两个线程：主线程和一个子线程。然而，你的程序可以创建所需的更多线程。例如，下面的程序创建了三个子线程：

```
// Create multiple threads.
class NewThread implements Runnable {
```

```
  String name; // name of thread
  Thread t;

  NewThread(String threadname) {
    name = threadname;
    t = new Thread(this, name);
    System.out.println("New thread: " + t);
    t.start(); // Start the thread
  }

  // This is the entry point for thread.
  public void run() {
   try {
      for(int i = 5; i > 0; i--) {
        System.out.println(name + ": " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println(name + "Interrupted");
    }
    System.out.println(name + " exiting.");
  }
}

class MultiThreadDemo {
  public static void main(String args[]) {
    new NewThread("One"); // start threads
    new NewThread("Two");
    new NewThread("Three");

    try {
      // wait for other threads to end
      Thread.sleep(10000);
    } catch (InterruptedException e) {
      System.out.println("Main thread Interrupted");
    }

    System.out.println("Main thread exiting.");
  }
}
```

程序输出如下所示：

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Three: 3
Two: 3
```

```
One: 2
Three: 2
Two: 2
One: 1
Three: 1
Two: 1
One exiting.
Two exiting.
Three exiting.
Main thread exiting.
```

如你所见，一旦启动，所有三个子线程共享CPU。注意main()中对sleep(10000)的调用。这使主线程沉睡十秒确保它最后结束。

11.5　使用isAlive()和join()

如前所述，通常你希望主线程最后结束。在前面的例子中，这点是通过在main()中调用sleep()来实现的，经过足够长时间的延迟以确保所有子线程都先于主线程结束。然而，这不是一个令人满意的解决方法，它也带来一个大问题：一个线程如何知道另一线程已经结束？幸运的是，Thread类提供了回答此问题的方法。

有两种方法可以判定一个线程是否结束。第一，可以在线程中调用isAlive()。这种方法由Thread定义，它的通常形式如下：

```
final boolean isAlive( )
```

如果所调用线程仍在运行，isAlive()方法返回true，如果不是则返回false。

但isAlive（）很少用到，等待线程结束的更常用的方法是调用join()，描述如下：

```
final void join( ) throws InterruptedException
```

该方法等待所调用线程结束。该名字来自于要求线程等待直到指定线程参与的概念。join()的附加形式允许给等待指定线程结束定义一个最大时间。

下面是前面例子的改进版本。运用join()以确保主线程最后结束。同样，它也演示了isAlive()方法。

```
// Using join() to wait for threads to finish.
class NewThread implements Runnable {
  String name; // name of thread
  Thread t;

  NewThread(String threadname) {
    name = threadname;
    t = new Thread(this, name);
    System.out.println("New thread: " + t);
    t.start(); // Start the thread
  }

  // This is the entry point for thread.
  public void run() {
```

```
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println(name + ": " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println(name + " interrupted.");
    }
    System.out.println(name + " exiting.");
  }
}

class DemoJoin {
  public static void main(String args[]) {
    NewThread ob1 = new NewThread("One");
    NewThread ob2 = new NewThread("Two");
    NewThread ob3 = new NewThread("Three");

    System.out.println("Thread One is alive: "
                       + ob1.t.isAlive());
    System.out.println("Thread Two is alive: "
                       + ob2.t.isAlive());
    System.out.println("Thread Three is alive: "
                       + ob3.t.isAlive());
    // wait for threads to finish
    try {
      System.out.println("Waiting for threads to finish.");
      ob1.t.join();
      ob2.t.join();
      ob3.t.join();
    } catch (InterruptedException e) {
      System.out.println("Main thread Interrupted");
    }

    System.out.println("Thread One is alive: "
                       + ob1.t.isAlive());
    System.out.println("Thread Two is alive: "
                       + ob2.t.isAlive());
    System.out.println("Thread Three is alive: "
                       + ob3.t.isAlive());

    System.out.println("Main thread exiting.");
  }
}
```

程序输出如下所示:

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
Thread One is alive: true
Thread Two is alive: true
Thread Three is alive: true
Waiting for threads to finish.
One: 5
```

```
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Two: 3
Three: 3
One: 2
Two: 2
Three: 2
One: 1
Two: 1
Three: 1
Two exiting.
Three exiting.
One exiting.
Thread One is alive: false
Thread Two is alive: false
Thread Three is alive: false
Main thread exiting.
```

如你所见，调用join()后返回，线程终止执行。

11.6　线程优先级

线程优先级被线程调度用来判定何时每个线程允许运行。理论上，优先级高的线程比优先级低的线程获得更多的CPU时间。实际上，线程获得的CPU时间通常由包括优先级在内的多个因素决定（例如，一个实行多任务处理的操作系统如何更有效的利用CPU时间）。一个优先级高的线程自然比优先级低的线程优先。举例来说，当低优先级线程正在运行，而一个高优先级的线程被恢复（例如从沉睡中或等待I/O中），它将抢占低优先级线程所使用的CPU。

理论上，等优先级线程有同等的权利使用CPU。但你必须小心了。记住，Java是被设计成能在很多环境下工作的。一些环境下实现多任务处理从本质上与其他环境不同。为安全起见，等优先级线程偶尔也受控制。这保证了所有线程在无优先级的操作系统下都有机会运行。实际上，在无优先级的环境下，多数线程仍然有机会运行，因为很多线程不可避免的会遭遇阻塞，例如等待输入输出。遇到这种情形，阻塞的线程挂起，其他线程运行。但是如果你希望多线程执行的顺利的话，最好不要采用这种方法。同样，有些类型的任务是占CPU的。对于这些支配CPU类型的线程，有时你希望能够支配它们，以便使其他线程可以运行。

设置线程的优先级，用setPriority()方法，该方法也是Tread 的成员。它的通常形式为：

```
final void setPriority(int level)
```

这里，level指定了对所调用的线程的新的优先权的设置。Level的值必须在MIN_PRIORITY到MAX_PRIORITY范围内。通常，它们的值分别是1和10。要返回一个线

程为默认的优先级，指定NORM_PRIORITY，通常值为5。这些优先级在Thread中都被定义为final型变量。

你可以通过调用Thread的getPriority()方法来获得当前的优先级设置。该方法如下：

```
final int getPriority( )
```

当涉及调度时，Java的执行可以有本质上不同的行为。Windows 95/98/NT/2000 的工作或多或少如你所愿。但其他版本可能工作的完全不同。大多数矛盾发生在你使用有优先级行为的线程，而不是协同的腾出CPU时间。最安全的办法是获得可预先性的优先权，Java获得跨平台的线程行为的方法是自动放弃对CPU的控制。

下面的例子阐述了两个不同优先级的线程，运行于具有优先权的平台，这与运行于无优先级的平台不同。一个线程通过Thread.NORM_PRIORITY设置了高于普通优先级两级的级数，另一线程设置的优先级则低于普通级两级。两线程被启动并允许运行10秒。每个线程执行一个循环，记录反复的次数。10秒后，主线程终止了两线程。每个线程经过循环的次数被显示。

```
// Demonstrate thread priorities.
class clicker implements Runnable {
  int click = 0;
  Thread t;
  private volatile boolean running = true;

  public clicker(int p) {
    t = new Thread(this);
    t.setPriority(p);
  }

  public void run() {
    while (running) {
      click++;
    }
  }

  public void stop() {
    running = false;
  }

  public void start() {
    t.start();
  }
}

class HiLoPri {
  public static void main(String args[]) {
    Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
    clicker hi = new clicker(Thread.NORM_PRIORITY + 2);
    clicker lo = new clicker(Thread.NORM_PRIORITY - 2);

    lo.start();
    hi.start();
    try {
```

```
      Thread.sleep(10000);
    } catch (InterruptedException e) {
       System.out.println("Main thread interrupted.");
    }

    lo.stop();
    hi.stop();

    // Wait for child threads to terminate.
    try {
      hi.t.join();
      lo.t.join();
    } catch (InterruptedException e) {
      System.out.println("InterruptedException caught");
    }

    System.out.println("Low-priority thread: " + lo.click);
    System.out.println("High-priority thread: " + hi.click);
  }
}
```

该程序在Windows 98下运行的输出，表明线程确实上下转换，甚至既不屈从于CPU，也不被输入输出阻塞。优先级高的线程获得大约90%的CPU时间。

```
Low-priority thread: 4408112
High-priority thread: 589626904
```

当然，该程序的精确的输出结果依赖于你的CPU的速度和运行的其他任务的数量。当同样的程序运行于无优先级的系统，将会有不同的结果。

上述程序还有个值得注意的地方。注意running前的关键字volatile。尽管volatile 在下章会被很仔细的讨论，用在此处以确保running的值在下面的循环中每次都得到验证。

```
while (running) {
  click++;
}
```

如果不用volatile，Java可以自由的优化循环：running的值被存在CPU的一个寄存器中，每次重复不一定需要复检。volatile的运用阻止了该优化，告知Java running可以改变，改变方式并不以直接代码形式显示。

11.7　线 程 同 步

当两个或两个以上的线程需要共享资源，它们需要某种方法来确定资源在某一刻仅被一个线程占用。达到此目的的过程叫做同步（synchronization）。像你所看到的，Java为此提供了独特的，语言水平上的支持。

同步的关键是管程（也叫信号量semaphore）的概念。管程是一个互斥独占锁定的对象，或称互斥体（mutex）。在给定的时间，仅有一个线程可以获得管程。当一个线程需要锁定，它必须进入管程。所有其他的试图进入已经锁定的管程的线程必须挂起直到第一个线程退

出管程。这些其他的线程被称为等待管程。一个拥有管程的线程如果愿意的话可以再次进入相同的管程。

如果你用其他语言例如C或C++时用到过同步，你会知道它用起来有一点诡异。这是因为很多语言它们自己不支持同步。相反，对同步线程，程序必须利用操作系统源语。幸运的是Java通过语言元素实现同步，大多数的与同步相关的复杂性都被消除。

你可以用两种方法同步化代码。两者都包括synchronized关键字的运用，下面分别说明这两种方法。

11.7.1　使用同步方法

Java中同步是简单的，因为所有对象都有它们与之对应的隐式管程。进入某一对象的管程，就是调用被synchronized关键字修饰的方法。当一个线程在一个同步方法内部，所有试图调用该方法（或其他同步方法）的同实例的其他线程必须等待。为了退出管程，并放弃对对象的控制权给其他等待的线程，拥有管程的线程仅需从同步方法中返回。

为理解同步的必要性，让我们从一个应该使用同步却没有用的简单例子开始。下面的程序有三个简单类。首先是Callme，它有一个简单的方法call( )。call( )方法有一个名为msg的String参数。该方法试图在方括号内打印msg 字符串。有趣的事是在调用call( ) 打印左括号和msg字符串后，调用Thread.sleep(1000)，该方法使当前线程暂停1秒。

下一个类的构造函数Caller，引用了Callme的一个实例以及一个String，它们被分别存在target 和 msg 中。构造函数也创建了一个调用该对象的run( )方法的新线程。该线程立即启动。Caller类的run( )方法通过参数msg字符串调用Callme实例target的call( ) 方法。最后，Synch类由创建Callme的一个简单实例和Caller的三个具有不同消息字符串的实例开始。Callme的同一实例传给每个Caller实例。

```
// This program is not synchronized.
class Callme {
  void call(String msg) {
    System.out.print("[" + msg);
    try {
      Thread.sleep(1000);
    } catch(InterruptedException e) {
      System.out.println("Interrupted");
    }
    System.out.println("]");
  }
}

class Caller implements Runnable {
  String msg;
  Callme target;
  Thread t;

  public Caller(Callme targ, String s) {
    target = targ;
    msg = s;
    t = new Thread(this);
    t.start();
```

```
  }

  public void run() {
    target.call(msg);
  }
}

class Synch {
  public static void main(String args[]) {
    Callme target = new Callme();
    Caller ob1 = new Caller(target, "Hello");
    Caller ob2 = new Caller(target, "Synchronized");
   Caller ob3 = new Caller(target, "World");

    // wait for threads to end
    try {
      ob1.t.join();
      ob2.t.join();
      ob3.t.join();
    } catch(InterruptedException e) {
      System.out.println("Interrupted");
    }
  }
}
```

该程序的输出如下：

```
Hello[Synchronized[World]
]
]
```

在本例中，通过调用sleep( )，call( )方法允许执行转换到另一个线程。该结果是三个消息字符串的混合输出。该程序中，没有阻止三个线程同时调用同一对象的同一方法的方法存在。这是一种竞争，因为三个线程争着完成方法。例题用sleep( )使该影响重复和明显。在大多数情况，竞争是更为复杂和不可预知的，因为你不能确定何时上下文转换会发生。这使程序时而运行正常时而出错。

为达到上例所想达到的目的，必须有权连续的使用call( )。也就是说，在某一时刻，必须限制只有一个线程可以支配它。为此，你只需在call( ) 定义前加上关键字synchronized，如下：

```
class Callme {
   synchronized void call(String msg) {
   ...
```

这防止了在一个线程使用call( )时其他线程进入call( )。在synchronized加到call( )前面以后，程序输出如下：

```
[Hello]
[Synchronized]
[World]
```

任何时候在多线程情况下，你有一个方法或多个方法操纵对象的内部状态，都必须用

synchronized 关键字来防止状态出现竞争。记住，一旦线程进入实例的同步方法，没有其他线程可以进入相同实例的同步方法。然而，该实例的其他不同步方法却仍然可以被调用。

11.7.2　同步语句

尽管在创建的类的内部创建同步方法是获得同步的简单和有效的方法，但它并非在任何时候都有效。这其中的原因，请跟着思考。假设你想获得不为多线程访问设计的类对象的同步访问，也就是，该类没有用到synchronized方法。而且，该类不是你自己，而是第三方创建的，你不能获得它的源代码。这样，你不能在相关方法前加synchronized修饰符。怎样才能使该类的一个对象同步化呢？很幸运，解决方法很简单：你只需将对这个类定义的方法的调用放入一个synchronized块内就可以了。

下面是synchronized语句的普通形式：

```
synchronized(object) {
   // statements to be synchronized
  }
```

其中，object是被同步对象的引用。如果你想要同步的只是一个语句，那么不需要花括号。一个同步块确保对object成员方法的调用仅在当前线程成功进入object管程后发生。

下面是前面程序的修改版本，在run( )方法内用了同步块：

```
// This program uses a synchronized block.
class Callme {
  void call(String msg) {
    System.out.print("[" + msg);
    try {
      Thread.sleep(1000);
    } catch (InterruptedException e) {
      System.out.println("Interrupted");
    }
    System.out.println("]");
  }
}

class Caller implements Runnable {
  String msg;
  Callme target;
  Thread t;

  public Caller(Callme targ, String s) {
    target = targ;
    msg = s;
    t = new Thread(this);
    t.start();
  }

  // synchronize calls to call()
  public void run() {
    synchronized(target) { // synchronized block
      target.call(msg);
    }
```

```
  }
}

class Synch1 {
  public static void main(String args[]) {
    Callme target = new Callme();
    Caller ob1 = new Caller(target, "Hello");
    Caller ob2 = new Caller(target, "Synchronized");
    Caller ob3 = new Caller(target, "World");

    // wait for threads to end
    try {
      ob1.t.join();
      ob2.t.join();
      ob3.t.join();
    } catch(InterruptedException e) {
      System.out.println("Interrupted");
    }
  }
}
```

这里，call( )方法没有被synchronized修饰。而synchronized是在Caller类的run( )方法中声明的。这可以得到上例中同样正确的结果，因为每个线程运行前都等待先前的一个线程结束。

11.8　线程间通信

上述例题无条件的阻塞了其他线程异步访问某个方法。Java对象中隐式管程的应用是很强大的，但是你可以通过进程间通信达到更微妙的境界。这在Java中是尤为简单的。

像前面所讨论过的，多线程通过把任务分成离散的和合乎逻辑的单元代替了事件循环程序。线程还有第二优点：它远离了轮询。轮询通常由重复监测条件的循环实现。一旦条件成立，就要采取适当的行动。这浪费了CPU时间。举例来说，考虑经典的序列问题，当一个线程正在产生数据而另一个程序正在消费它。为使问题变得更有趣，假设数据产生器必须等待消费者完成工作才能产生新的数据。在轮询系统，消费者在等待生产者产生数据时会浪费很多CPU周期。一旦生产者完成工作，它将启动轮询，浪费更多的CPU时间等待消费者的工作结束，如此下去。很明显，这种情形不受欢迎。

为避免轮询，Java包含了通过wait( )，notify( )和notifyAll( )方法实现的一个进程间通信机制。这些方法在对象中是用final方法实现的，所以所有的类都含有它们。这三个方法仅在synchronized方法中才能被调用。尽管这些方法从计算机科学远景方向上来说具有概念的高度先进性，实际中用起来是很简单的：

- wait( ) 告知被调用的线程放弃管程进入睡眠直到其他线程进入相同管程并且调用 notify( )。
- notify( ) 恢复相同对象中第一个调用 wait( ) 的线程。
- notifyAll( ) 恢复相同对象中所有调用 wait( ) 的线程。具有最高优先级的线程最先

运行。

这些方法在Object中被声明，如下所示：

```
final void wait( ) throws InterruptedException
final void notify( )
final void notifyAll( )
```

wait( )存在的另外的形式允许你定义等待时间。

下面的例子程序错误的实行了一个简单生产者/消费者的问题。它由四个类组成：Q，设法获得同步的序列；Producer，产生排队的线程对象；Consumer，消费序列的线程对象；以及PC，创建单个Q，Producer，和Consumer的小类。

```
// An incorrect implementation of a producer and consumer.
class Q {
  int n;

  synchronized int get() {
    System.out.println("Got: " + n);
    return n;
  }

  synchronized void put(int n) {
    this.n = n;
    System.out.println("Put: " + n);
  }
}

class Producer implements Runnable {
  Q q;

  Producer(Q q) {
    this.q = q;
    new Thread(this, "Producer").start();
  }

  public void run() {
    int i = 0;

    while(true) {
      q.put(i++);
    }
  }
}

class Consumer implements Runnable {
  Q q;

  Consumer(Q q) {
    this.q = q;
    new Thread(this, "Consumer").start();
  }

  public void run() {
```

```
    while(true) {
      q.get();
    }
  }
}

class PC {
  public static void main(String args[]) {
    Q q = new Q();
    new Producer(q);
    new Consumer(q);

    System.out.println("Press Control-C to stop.");
  }
}
```

尽管Q类中的put( )和get( )方法是同步的，没有东西阻止生产者超越消费者，也没有东西阻止消费者消费同样的序列两次。这样，你就得到下面的错误输出（输出将随处理器速度和装载的任务而改变）：

```
Put: 1
Got: 1
Got: 1
Got: 1
Got: 1
Got: 1
Put: 2
Put: 3
Put: 4
Put: 5
Put: 6
Put: 7
Got: 7
```

生产者生成1后，消费者依次获得同样的1五次。生产者在继续生成2到7，消费者没有机会获得它们。

用Java正确的编写该程序是用wait( )和notify( )来对两个方向进行标志，如下所示：

```
// A correct implementation of a producer and consumer.
class Q {
  int n;
  boolean valueSet = false;

  synchronized int get() {
    if(!valueSet)
      try {
        wait();

     } catch(InterruptedException e) {
        System.out.println("InterruptedException caught");
      }

      System.out.println("Got: " + n);
      valueSet = false;
```

```
      notify();
      return n;
    }

    synchronized void put(int n) {
      if(valueSet)
        try {
          wait();
        } catch(InterruptedException e) {
          System.out.println("InterruptedException caught");
        }

        this.n = n;
        valueSet = true;
        System.out.println("Put: " + n);
        notify();
    }
  }

  class Producer implements Runnable {
    Q q;

    Producer(Q q) {
      this.q = q;
      new Thread(this, "Producer").start();
    }

    public void run() {
      int i = 0;

      while(true) {
        q.put(i++);
      }
    }
  }

  class Consumer implements Runnable {
    Q q;

    Consumer(Q q) {
      this.q = q;
      new Thread(this, "Consumer").start();
    }

    public void run() {
      while(true) {
        q.get();
      }
    }
  }

  class PCFixed {
    public static void main(String args[]) {
      Q q = new Q();
      new Producer(q);
```

```
    new Consumer(q);

    System.out.println("Press Control-C to stop.");
  }
}
```

内部get( ), wait( )被调用。这使执行挂起直到Producer 告知数据已经预备好。这时，内部get( ) 被恢复执行。获取数据后，get( )调用notify( )。这告诉Producer可以向序列中输入更多数据。在put( )内，wait( )挂起执行直到Consumer取走了序列中的项目。当执行再继续，下一个数据项目被放入序列，notify( )被调用，这通知Consumer它应该移走该数据。

下面是该程序的输出，它清楚的显示了同步行为：

```
Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
Got: 3
Put: 4
Got: 4
Put: 5
Got: 5
```

11.8.1　死锁

需要避免的与多任务处理有关的特殊错误类型是死锁（deadlock）。死锁发生在当两个线程对一对同步对象有循环依赖关系时。例如，假定一个线程进入了对象X的管程而另一个线程进入了对象Y的管程。如果X的线程试图调用Y的同步方法，它将像预料的一样被锁定。而Y的线程同样希望调用X的一些同步方法，线程永远等待，因为为到达X，必须释放自己的Y的锁定以使第一个线程可以完成。死锁是很难调试的错误，因为：

- 通常，它极少发生，只有到两线程的时间段刚好符合时才能发生。
- 它可能包含多于两个的线程和同步对象（也就是说，死锁在比刚讲述的例子有更多复杂的事件序列的时候可以发生）。

为充分理解死锁，观察它的行为是很有用的。下面的例子生成了两个类，A和B，分别有foo( )和bar( )方法。这两种方法在调用其他类的方法前有一个短暂的停顿。主类，名为Deadlock，创建了A和B的实例，然后启动第二个线程去设置死锁环境。foo( )和bar( )方法使用sleep( )强迫死锁现象发生。

```
// An example of deadlock.
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("A Interrupted");
    }

    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("B Interrupted");
    }

    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");
    Thread t = new Thread(this, "RacingThread");
    t.start();

    a.foo(b); // get lock on a in this thread.
    System.out.println("Back in main thread");
  }

  public void run() {
    b.bar(a); // get lock on b in other thread.
    System.out.println("Back in other thread");
  }

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

运行程序后，输出如下：

```
MainThread entered A.foo
RacingThread entered B.bar
MainThread trying to call B.last()
RacingThread trying to call A.last()
```

因为程序死锁，你需要按CTRL-C来结束程序。在PC机上按CTRL-BREAK（或在Solaris下按CTRL-\）你可以看到全线程和管程缓冲堆。你会看到RacingThread在等待管程a时占用管程b，同时，MainThread占用a等待b。该程序永远都不会结束。像该例阐明的，你的多线程程序经常被锁定，死锁是你首先应检查的问题。

11.9　挂起、恢复和终止线程

有时，线程的挂起是很有用的。例如，一个独立的线程可以用来显示当日的时间。如果用户不希望用时钟，线程被挂起。在任何情形下，挂起线程是很简单的，一旦挂起，重新启动线程也是一件简单的事。

挂起，终止和恢复线程机制在Java 2和早期版本中有所不同。尽管你运用Java 2的途径编写代码，你仍需了解这些操作在早期Java环境下是如何完成的。例如，你也许需要更新或维护老的代码。你也需要了解为什么Java 2会有这样的变化。因为这些原因，下面内容描述了执行线程控制的原始方法，接着是Java 2的方法。

11.9.1　Java 1.1或更早版本的线程的挂起、恢复和终止

先于Java2的版本，程序用Thread 定义的suspend() 和 resume() 来暂停和再启动线程。它们的形式如下：

```
final void suspend( )
final void resume( )
```

下面的程序描述了这些方法：

```
// Using suspend() and resume().
class NewThread implements Runnable {
  String name; // name of thread
  Thread t;

  NewThread(String threadname) {
    name = threadname;
    t = new Thread(this, name);
    System.out.println("New thread: " + t);
    t.start(); // Start the thread
  }

  // This is the entry point for thread.
  public void run() {
    try {
      for(int i = 15; i > 0; i--) {
        System.out.println(name + ": " + i);
```

```
        Thread.sleep(200);
      }
    } catch (InterruptedException e) {
      System.out.println(name + " interrupted.");
    }
    System.out.println(name + " exiting.");
  }
}

class SuspendResume {
  public static void main(String args[]) {
    NewThread ob1 = new NewThread("One");
    NewThread ob2 = new NewThread("Two");

    try {
      Thread.sleep(1000);
      ob1.t.suspend();
      System.out.println("Suspending thread One");
      Thread.sleep(1000);
      ob1.t.resume();
      System.out.println("Resuming thread One");
      ob2.t.suspend();
      System.out.println("Suspending thread Two");
      Thread.sleep(1000);
      ob2.t.resume();
      System.out.println("Resuming thread Two");
    } catch (InterruptedException e) {
      System.out.println("Main thread Interrupted");
    }

    // wait for threads to finish
    try {
      System.out.println("Waiting for threads to finish.");
      ob1.t.join();
      ob2.t.join();
    } catch (InterruptedException e) {
      System.out.println("Main thread Interrupted");
    }
    System.out.println("Main thread exiting.");
  }
}
```

程序的部分输出如下：

```
New thread: Thread[One,5,main]
One: 15
New thread: Thread[Two,5,main]
Two: 15
One: 14
Two: 14
One: 13
Two: 13
One: 12
Two: 12
One: 11
```

```
Two: 11
Suspending thread One
Two: 10
Two: 9
Two: 8
Two: 7
Two: 6
Resuming thread One
Suspending thread Two
One: 10
One: 9
One: 8
One: 7
One: 6
Resuming thread Two
Waiting for threads to finish.
Two: 5
One: 5
Two: 4
One: 4
Two: 3
One: 3
Two: 2
One: 2
Two: 1
One: 1
Two exiting.
One exiting.
Main thread exiting.
```

Thread类同样定义了stop() 来终止线程。它的形式如下：

```
void stop( )
```

一旦线程被终止，它不能被resume() 恢复继续运行。

11.9.2　Java 2中挂起、恢复和终止线程

Thread定义的suspend()，resume()和stop()方法看起来是管理线程的完美的和方便的方法，它们不能用于新Java版本的程序。下面是其中的原因。Thread类的suspend()方法在Java 2中不被赞成，因为suspend()有时会造成严重的系统故障。假定对关键的数据结构的一个线程被锁定的情况，如果该线程在那里挂起，这些锁定的线程并没有放弃对资源的控制。其他的等待这些资源的线程可能死锁。

Resume()方法同样不被赞同。它不引起问题，但不能离开suspend()方法而独立使用。

Thread类的stop()方法同样在Java 2中受到反对。这是因为该方法可能导致严重的系统故障。设想一个线程正在写一个精密的重要的数据结构且仅完成一个零头。如果该线程在此刻终止，则数据结构可能会停留在崩溃状态。

因为在Java 2中不能使用suspend()，resume()和stop() 方法来控制线程，你也许会想那就没有办法来停止，恢复和结束线程。其实不然。相反，线程必须被设计以使run() 方法定期检查以来判定线程是否应该被挂起，恢复或终止它自己的执行。有代表性的，这由建立

一个指示线程状态的标志变量来完成。只要该标志设为“running”，run()方法必须继续让线程执行。如果标志设为“suspend”，线程必须暂停。若设为“stop”，线程必须终止。当然，编写这样的代码有很多方法，但中心主题对所有的程序应该是相同的。

下面的例题阐述了从Object继承的wait()和notify()方法怎样控制线程的执行。该例与前面讲过的程序很像。然而，不被赞同的方法都没有用到。让我们思考程序的执行。

NewTread 类包含了用来控制线程执行的布尔型的实例变量suspendFlag。它被构造函数初始化为false。Run()方法包含一个监测suspendFlag 的同步声明的块。如果变量是true，wait()方法被调用以挂起线程。Mysuspend()方法设置suspendFlag为true。Myresume()方法设置suspendFlag为false并且调用notify()方法来唤起线程。最后，main()方法被修改以调用mysuspend()和myresume()方法。

```
// Suspending and resuming a thread for Java2
class NewThread implements Runnable {
  String name; // name of thread
  Thread t;
  boolean suspendFlag;

  NewThread(String threadname) {
    name = threadname;
    t = new Thread(this, name);
    System.out.println("New thread: " + t);
    suspendFlag = false;
    t.start(); // Start the thread
  }

  // This is the entry point for thread.
  public void run() {
    try {
      for(int i = 15; i > 0; i--) {
        System.out.println(name + ": " + i);
        Thread.sleep(200);
        synchronized(this) {
          while(suspendFlag) {
            wait();
          }
        }
      }
    } catch (InterruptedException e) {
      System.out.println(name + " interrupted.");
    }
    System.out.println(name + " exiting.");
  }

  void mysuspend() {
    suspendFlag = true;
  }

  synchronized void myresume() {
    suspendFlag = false;
    notify();
  }
```

```
}

class SuspendResume {
  public static void main(String args[]) {
    NewThread ob1 = new NewThread("One");
    NewThread ob2 = new NewThread("Two");

    try {
      Thread.sleep(1000);
      ob1.mysuspend();
      System.out.println("Suspending thread One");
      Thread.sleep(1000);
      ob1.myresume();
      System.out.println("Resuming thread One");
      ob2.mysuspend();
      System.out.println("Suspending thread Two");
      Thread.sleep(1000);
      ob2.myresume();
      System.out.println("Resuming thread Two");
    } catch (InterruptedException e) {
      System.out.println("Main thread Interrupted");
    }

    // wait for threads to finish
    try {
      System.out.println("Waiting for threads to finish.");
      ob1.t.join();
      ob2.t.join();
    } catch (InterruptedException e) {
      System.out.println("Main thread Interrupted");
    }

    System.out.println("Main thread exiting.");
  }
}
```

该程序的输出与前面的程序相同。此书的后面部分，你将看到用Java 2机制控制线程的更多例子。尽管这种机制不像老方法那样“干净”，然而，它是确保运行时不发生错误的方法。它是所有新的代码必须采用的方法。

11.10　使用多线程机制

如果你和大多数程序员一样，那么在语言中加入多线程支持对你来说是很新鲜的事物。有效运用这种支持的关键是并发思考而不是连续思考。例如，当你的程序有两个可以并行执行的子系统，创建它们各自的线程。仔细的运用多线程，你能编写出非常有效的程序。然而要注意：如果你创建太多的线程，你可能会减弱而不是加强程序的性能。记住，上下文转换是需要开销的。如果你创建了太多的线程，更多的CPU时间会用于上下文转换而不是用来执行程序。

第 12 章　输入/输出、小应用程序和其他主题

本章介绍Java的两个重要的包：io和applet。io包支持Java的基本I/O（输入/输出）系统，包括文件的输入/输出。applet包支持applet（小应用程序）。对输入/输出和applet的支持是来源于Java的内核API库，而不是语言关键字。因为这个原因，关于这些主题的深入讨论在本书的第2部分可以见到，这些讨论验证了Java的API类。本章讨论这两个子系统的基础部分，这样你可以看到它们怎样融入Java语言，怎样符合Java编程和执行环境的大量内容。本章同样介绍了Java的最后的关键字：transient，volatile，instanceof，native以及strictfp。

12.1　输入/输出基础

在阅读前面的第11章时你也许注意到在例题中输入/输出没有很多的应用。实际上，除了print()和println()，基本没有运用输入/输出方法。原因很简单：很多实际的Java应用程序不是基于文本的控制台程序。相反，它们是与用户交流的依赖于抽象窗口工具集（AWT）的用于绘图的小应用程序。尽管基于文本的程序作为教学实例是很出色的，它们无法胜任JAVA在实际中的重要应用。同样Java对外设输入/输出的支持也是有限的，并且用起来有些笨拙——甚至是在简单的例子程序中。基于文本的控制台输入/输出对于Java程序并不是十分重要。

尽管在前面的章节中Java 没有提供与文件和网络相关的强大的和灵活的输入/输出支持，Java的输入/输出系统是紧密相连并且是具有一致性的。实际上，一旦你理解了它的基本原理，输入/输出系统的其他部分就变得易于掌握了。

12.1.1　流的概念

Java程序通过流来完成输入/输出。流是生产或消费信息的抽象。流通过Java的输入/输出系统与物理设备链接。尽管与它们链接的物理设备不尽相同，所有流的行为具有同样的方式。这样，相同的输入/输出类和方法适用于所有类型的外部设备。这意味着一个输入流能够抽象多种不同类型的输入：从磁盘文件，从键盘或从网络套接字。同样，一个输出流可以输出到控制台，磁盘文件或相连的网络。流是处理输入/输出的一个洁净的方法，例如它不需要代码理解键盘和网络的不同。Java中流的实现是在java.io包定义的类层次结构内部的。

注意：如果你熟悉C/C++,你已经对流的概念很熟悉了。JAVA中流的实现跟C/C++中有些相似。

12.1.2　字节流和字符流

Java 2 定义了两种类型的流：字节类和字符类。字节流（byte stream）为处理字节的

输入和输出提供了方便的方法。例如使用字节流读取或书写二进制数据。字符流（character stream）为字符的输入和输出处理提供了方便。它们采用了统一的编码标准，因而可以国际化。当然，在某些场合，字符流比字节流更有效。

Java的原始版本（Java 1.0）不包括字符流，因此所有的输入和输出都是以字节为单位的。Java 1.1中加入了字符流，某些字节形式的类和方法不受欢迎。这也是为什么没用字符流的老代码在适当的地方需要更新的原因。

需要声明：在最底层，所有的输入/输出都是字节形式的。基于字符的流只为处理字符提供方便有效的方法。

下面是对字节流和字符流的概述。

字节流类

字节流由两个类层次结构定义。在顶层有两个抽象类：InputStream 和 OutputStream。每个抽象类都有多个具体的子类，这些子类对不同的外设进行处理，例如磁盘文件，网络连接，甚至是内存缓冲区。字节流类显示于表12-1中。本章的后面将讨论一些这样的类。其他的类的描述在第2部分。记住，要使用流类，必须导入Java.io包。

表 12-1 字节流类

流类	含义
BufferedInputStream	缓冲输入流
BufferedOutputStream	缓冲输出流
ByteArrayInputStream	从字节数组读取的输入流
ByteArrayOutputStream	向字节数组写入的输出流
DataInputStream	包含读取Java标准数据类型方法的输入流
DataOutputStream	包含编写Java 标准数据类型方法的输出流
FileInputStream	读取文件的输入流
FileOutputStream	写文件的输出流
FilterInputStream	实现 InputStream
FilterOutputStream	实现 OutputStream
InputStream	描述流输入的抽象类
OutputStream	描述流输出的抽象类
PipedInputStream	输入管道
PipedOutputStream	输出管道
PrintStream	包含print( ) 和 println( )的输出流
PushbackInputStream	支持向输入流返回一个字节的单字节的“unget”的输入流
RandomAccessFile	支持随机文件输入/输出
SequenceInputStream	两个或两个以上顺序读取的输入流组成的输入流

抽象类InputStream 和 OutputStream定义了实现其他流类的关键方法。最重要的两种方法是read()和write()，它们分别对数据的字节进行读写。两种方法都在InputStream 和

OutputStream中被定义为抽象方法。它们被派生的流类重载。

字符流类

字符流类由两个类层次结构定义。顶层有两个抽象类：Reader和Writer。这些抽象类处理统一编码的字符流。Java中这些类含有多个具体的子类。字符流类如表12-2所示。

表 12-2　字符流的输入/输出类

流类	含义
BufferedReader	缓冲输入字符流
BufferedWriter	缓冲输出字符流
CharArrayReader	从字符数组读取数据的输入流
CharArrayWriter	向字符数组写数据的输出流
FileReader	读取文件的输入流
FileWriter	写文件的输出流
FilterReader	过滤读
FilterWriter	过滤写
InputStreamReader	把字节转换成字符的输入流
LineNumberReader	计算行数的输入流
OutputStreamWriter	把字符转换成字节的输出流
PipedReader	输入管道
PipedWriter	输出管道
PrintWriter	包含print( )和println( )的输出流
PushbackReader	允许字符返回到输入流的输入流
Reader	描述字符流输入的抽象类
StringReader	读取字符串的输入流
StringWriter	写字符串的输出流
Writer	描述字符流输出的抽象类

抽象类Reader和Writer定义了几个实现其他流类的关键方法。其中两个最重要的是read()和write()，它们分别进行字符数据的读和写。这些方法被派生流类重载。

12.1.3　预定义流

所有的Java程序自动导入java.lang包。该包定义了一个名为System的类，该类封装了运行时环境的多个方面。例如，使用它的某些方法，你能获得当前时间和与系统有关的不同属性。System 同时包含三个预定义的流变量，in，out和err。这些成员在System中是被定义成public 和static型的，这意味着它们可以不引用特定的System对象而被用于程序的其他部分。

System.out是标准的输出流。默认情况下，它是一个控制台。System.in是标准输入，默认情况下，它指的是键盘。System.err指的是标准错误流，它默认是控制台。然而，这些流

可以重定向到任何兼容的输入/输出设备。

System.in 是inputStream的对象；System.out和System.err是PrintStream的对象。它们都是字节流，尽管它们用来读写外设的字符。如果愿意，你可以用基于字符的流来包装它们。

前面的章节在例题中已经用到过System.out。你可以以同样的方式使用System.err。下面对此进行解释，你会看到使用System.in有一点复杂。

12.2 读取控制台输入

在Java 1.0中，完成控制台输入的惟一途径是字节流，使用该方法的老代码依然存在。今天，运用字节流读取控制台输入在技术上仍是可行的，但这样做需要用到不被赞成的方法，这种做法不值得推荐。Java 2中读取控制台输入的首选方法是字符流，它使程序容易符合国际标准并且易于维护。

注意：Java没有像标准C的函数scanf()或C++输入操作符那样的统一的控制台输入方法。

Java中，控制台输入由从System.in读取数据来完成。为获得属于控制台的字符流，在BufferedReader对象中包装System.in。BufferedReader 支持缓冲输入流。它最常见的构造函数如下：

```
BufferedReader(Reader inputReader)
```

这里，inputReader是链接被创建的BufferedReader实例的流。Reader是一个抽象类。它的一个具体的子类是InputStreamReader，该子类把字节转换成字符。为获得链接System.in的一个InputStreamReader的对象，用下面的构造函数：

```
InputStreamReader(InputStream inputStream)
```

因为System .in引用了InputStream 类型的对象，它可以用于inputStream。综上所述，下面的一行代码创建了与键盘相连的BufferedReader对象。

```
BufferedReader br = new BufferedReader(new
                    InputStreamReader(System.in));
```

当该语句执行后，br是通过System.in生成的链接控制台的字符流。

12.2.1 读取字符

从BufferedReader读取字符，用read()。我们所用的read()版本如下：

```
int read( ) throws IOException
```

该方法每次执行都从输入流读取一个字符然后以整型返回。当遇到流的末尾时它返回-1。你可以看到，它要引发一个IOException异常。

下面的例子程序演示了read()方法，从控制台读取字符直到用户键入“q”：

```
// Use a BufferedReader to read characters from the console.
```

```
import java.io.*;

class BRRead {
  public static void main(String args[])
    throws IOException
  {
    char c;
    BufferedReader br = new
           BufferedReader(new InputStreamReader(System.in));
    System.out.println("Enter characters, 'q' to quit.");


    // read characters
    do {
      c = (char) br.read();
      System.out.println(c);
    } while(c != 'q');
  }
}
```

下面是程序运行：

```
Enter characters, 'q' to quit.
123abcq
1
2
3
a
b
c
q
```

程序的输出看起来与预想的略有不同，因为System.in在默认情况下是以行来缓冲的。这意味着在你键入ENTER以前实际上是没有输入的。你能猜想，这不能充分体现交互式控制台输入条件下read()的独特价值。

12.2.2　读取字符串

从键盘读取字符串，使用readLine()。它是BufferedReader 类的成员。它的通常形式如下：

```
String readLine( ) throws IOException
```

它返回一个String对象。

下面的例子阐述了BufferedReader类和readLine()方法;程序读取和显示文本的行直到键入“stop”：

```
// Read a string from console using a BufferedReader.
import java.io.*;

class BRReadLines {
  public static void main(String args[])
    throws IOException
```

```
  {
    // create a BufferedReader using System.in
    BufferedReader br = new BufferedReader(new
                        InputStreamReader(System.in));
    String str;

    System.out.println("Enter lines of text.");
    System.out.println("Enter 'stop' to quit.");
    do {
      str = br.readLine();
      System.out.println(str);
    } while(!str.equals("stop"));
  }
}
```

下面的例题生成了一个小文本编辑器。它创建了一个String对象的数组，然后依行读取文本，把文本每一行存入数组。它将读取到100行或直到你按“stop”才停止。该例运用一个BufferedReader类来从控制台读取数据。

```
// A tiny editor.
import java.io.*;

class TinyEdit {
  public static void main(String args[])
    throws IOException
  {
    // create a BufferedReader using System.in
    BufferedReader br = new BufferedReader(new
                        InputStreamReader(System.in));
    String str[] = new String[100];

    System.out.println("Enter lines of text.");
    System.out.println("Enter 'stop' to quit.");
    for(int i=0; i<100; i++) {
      str[i] = br.readLine();
      if(str[i].equals("stop")) break;
    }

    System.out.println("\nHere is your file:");

    // display the lines
    for(int i=0; i<100; i++) {
      if(str[i].equals("stop")) break;
      System.out.println(str[i]);
    }
  }
}
```

下面是输出部分：

```
Enter lines of text.
Enter ‘stop’ to quit.
This is line one.
This is line two.
```

```
Java makes working with strings easy.
Just create String objects.
stop
Here is your file:
This is line one.
This is line two.
Java makes working with strings easy.
Just create String objects.
```

12.3　向控制台写输出

控制台输出由前面描述过的print( ) 和 println( )来完成最为简单，它们被用在本书的大多数例题中。这两种方法由PrintStream(System.out引用的对象类型)定义。尽管System.out是一个字节流，用它作为简单程序的输出是可行的。字符流输出在下节介绍。

因为PrintStream是从OutputStream派生的输出流，它同样实现低级方法write( )，write( )可用来向控制台写数据。PrintStream 定义的write( )的最简单的形式如下：

```
void write(int byteval)
```

该方法按照byteval指定的数向文件写字节。尽管byteval 定义成整数，但只有低位的8个字节被写入。下面的短例用 write( )向屏幕输出字符“A”，然后是新的行。

```
// Demonstrate System.out.write().
class WriteDemo {
  public static void main(String args[]) {
    int b;

    b = 'A';
    System.out.write(b);
    System.out.write('\n');
  }
}
```

一般不常用write( )来完成向控制台的输出（尽管这样做在某些场合非常有用），因为print( )和println( ) 更容易用。

12.4　PrintWriter类

尽管Java允许用System.out向控制台写数据，但建议仅用在调试程序时或在例题中，这在本书中得到充分体现。对于实际的程序，Java推荐的向控制台写数据的方法是用PrintWriter流。PrintWriter是基于字符的类。用基于字符类向控制台写数据使程序更为国际化。

PrintWriter定义了多个构造函数，我们所用到的一个如下：

```
PrintWriter(OutputStream outputStream, boolean flushOnNewline)
```

这里，outputStream是OutputStream类的对象，flushOnNewline控制Java是否在println( )

方法被调用时刷新输出流。如果flushOnNewline为true，刷新自动发生，若为false，则不发生。

PrintWriter支持所有类型（包括Object）的print( )和println( )方法，这样，你就可以像用System.out那样用这些方法。如果遇到不同类型的情况，PrintWriter方法调用对象的toString( )方法并打印结果。

用PrintWriter向外设写数据，指定输出流为System.out并在每一新行后刷新流。例如这行代码创建了与控制台输出相连的PrintWriter类。

```
PrintWriter pw = new PrintWriter(System.out, true);
```

下面的应用程序说明了用PrintWriter处理控制台输出的方法：

```
// Demonstrate PrintWriter
import java.io.*;

public class PrintWriterDemo {
  public static void main(String args[]) {
    PrintWriter pw = new PrintWriter(System.out, true);
    pw.println("This is a string");
    int i = -7;
    pw.println(i);
    double d = 4.5e-7;
    pw.println(d);
  }
}
```

该程序的输出如下：

```
This is a string
-7
4.5E-7
```

记住，在你学习Java或调试程序时用System.out向控制台写简单文本输出是没有错的。但是使用PrintWriter使实际的应用程序更容易国际化。因为在本书所示的例题程序中用PrintWriter并没有多大的优势，所以我们继续用System.out来向控制台输出。

12.5 文件的读写

Java为你提供了一系列的读写文件的类和方法。在Java中，所有的文件都是字节形式的。Java提供从文件读写字节的方法。而且，Java允许在字符形式的对象中使用字节文件流。这项技术在第2部分描述。本章说明基本的文件输入/输出。

两个最常用的流类是FileInputStream和FileOutputStream，它们生成与文件链接的字节流。为打开文件，你只需创建这些类中某一个类的一个对象，在构造函数中以参数形式指定文件的名称。这两个类都支持其他形式的重载构造函数。下面是我们将要用到的形式：

```
FileInputStream(String fileName) throws FileNotFoundException
FileOutputStream(String fileName) throws FileNotFoundException
```

这里，fileName指定需要打开的文件名。当你创建了一个输入流而文件不存在时，引发FileNotFoundException异常。对于输出流，如果文件不能生成，则引发FileNotFoundException异常。如果一个输出文件被打开，所有原先存在的同名的文件被破坏。

注意：在早期的Java版本中，当输出文件不能创建时FileOutputStream()引发一个IOException异常。这在Java 2中有所修改。

当你对文件的操作结束后，需要调用close( )来关闭文件。该方法在FileInputStream和FileOutputStream中都有定义。如下：

```
void close( ) throws IOException
```

为读文件，可以使用在FileInputStream中定义的read( )方法。我们用到的如下：

```
int read( ) throws IOException
```

该方法每次被调用，它仅从文件中读取一个字节并将该字节以整数形式返回。当读到文件尾时，read( )返回-1。该方法可以引发IOException异常。

下面的程序用read()来输入和显示文本文件的内容，该文件名以命令行形式指定。注意try/catch块处理程序运行时可能发生的两个错误——未找到指定的文件或用户忘记包括文件名了。当你用命令行时也可以用这样的方法。

```
/* Display a text file.

   To use this program, specify the name
   of the file that you want to see.
   For example, to see a file called TEST.TXT,
   use the following command line.

   Java ShowFile TEST.TXT

*/

import java.io.*;

class ShowFile {
  public static void main(String args[])
    throws IOException
  {
    int i;
    FileInputStream fin;

    try {
      fin = new FileInputStream(args[0]);
    } catch(FileNotFoundException e) {
      System.out.println("File Not Found");
      return;
    } catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("Usage: ShowFile File");
      return;
    }
```

```
    // read characters until EOF is encountered
    do {
      i = fin.read();
      if(i != -1) System.out.print((char) i);
    } while(i != -1);

    fin.close();
  }
}
```

向文件中写数据，需用FileOutputStream定义的write()方法。它的最简单形式如下：

```
void write(int byteval) throws IOException
```

该方法按照byteval指定的数向文件写入字节。尽管byteval作为整数声明，但仅低8位字节可以写入文件。如果在写的过程中出现问题，一个IOException被引发。下面的例子用write()拷贝一个文本文件：

```
/* Copy a text file.

   To use this program, specify the name
   of the source file and the destination file.
   For example, to copy a file called FIRST.TXT
   to a file called SECOND.TXT, use the following
   command line.

   Java CopyFile FIRST.TXT SECOND.TXT
*/

import java.io.*;

class CopyFile {
  public static void main(String args[])
    throws IOException
  {
    int i;
    FileInputStream fin;
    FileOutputStream fout;

    try {
      // open input file
      try {
        fin = new FileInputStream(args[0]);
      } catch(FileNotFoundException e) {
        System.out.println("Input File Not Found");
        return;
      }

      // open output file
      try {
        fout = new FileOutputStream(args[1]);
      } catch(FileNotFoundException e) {
        System.out.println("Error Opening Output File");
        return;
```

```
      }
    } catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("Usage: CopyFile From To");
      return;
    }

    // Copy File
    try {
      do {
        i = fin.read();
        if(i != -1) fout.write(i);
      } while(i != -1);
    } catch(IOException e) {
      System.out.println("File Error");
    }

    fin.close();
    fout.close();
  }
}
```

注意本程序中和前面ShowFile程序中处理潜在输入/输出错误的方法。不像其他的计算机语言，包括C和C++，这些语言用错误代码报告文件错误，而Java用异常处理机制。这样不仅是文件处理更为简洁，而且使Java正在执行输入时容易区分是文件出错还是EOF条件问题。在C/C++中，很多输入函数在出错时和到达文件结尾时返回相同的值（也就是说，在C/C++中，EOF情况与输入错误情况映射相同）。这通常意味着程序员必须还要编写特殊程序语句来判定究竟是哪种事件发生。Java中，错误通过异常引发，而不是通过read( )的返回值。这样，当read( )返回-1时，它仅表示一点：遇到了文件的结尾。

12.6　小应用程序基础

本书中前面所有的例子都是Java应用程序。然而，应用程序只是Java程序的一种。另一种类型的程序是applet（小应用程序）。如第1章提到的，小应用程序（applet）是访问internet服务器，在internet上传播的，自动安装的，作为部分Web文档运行的小应用程序。当小应用程序到达客户端，它被限制访问资源，以使它能够在不受病毒威胁和破坏数据完整性的情况下生成一个二进制的多媒体用户界面以及完成复杂的计算。

用到applet包时，很多关于创建和使用小应用程序的问题会在第2部分见到。然而，有关创建小应用程序的基础问题在这里描述，因为小应用程序与以前所用的程序具有不同的结构。你将看到，小应用程序在几处关键地方与应用程序不同。

让我们从下面的简单小应用程序开始：

```
import java.awt.*;
import java.applet.*;

public class SimpleApplet extends Applet {
  public void paint(Graphics g) {
    g.drawString("A Simple Applet", 20, 20);
```

```
  }
}
```

这个小应用程序以两个import语句开始。第一个导入抽象窗口工具集（AWT）类。小应用程序是通过AWT与用户交流的，而不是通过基于控制台的输入/输出类。AWT包含了对基于视窗的图形界面的支持。你能猜想，AWT是非常庞大和复杂的，关于它的详尽的讨论在本书的第2部分花了好几章。幸运的是，这个简单的小应用程序仅用到了AWT的一点点内容。第二个import语句导入了applet包，该包包含Applet类。每一个小应用程序都必须是Applet的子类。

程序的下面一行声明了SimpleApplet类。该类必须为public型，因为它的代码会在程序外面被引用。

在SimpleApplet内部声明了paint()。该方法由AWT定义且必须被小应用程序重载。小应用程序每次重新显示输出时都要调用paint()。发生这种情况有多种原因。例如，小应用程序运行的窗口可以被另一窗口重写然后覆盖。或者，小应用程序窗口可以最小化然后恢复。paint()方法在小应用程序启动时也被调用。无论什么原因，当小应用程序需要重画输出时，paint()总被调用。paint()方法有一个Graphics类型的参数，该参数包含描绘小应用程序运行的图形环境的内容。一旦小应用程序需要输出，该内容被用到。

在paint( )内调用Graphics类成员drawString()，该方法从指定的X，Y坐标处输出一个字符串。它有下面的常用形式：

```
void drawString(String message, int x, int y)
```

这里message是以x,y为输出起点的字符串。在Java窗口中，左上角的位置为0，0。在小应用程序中DrawString()的调用使得在坐标20，20处开始显示消息“A Simple Applet”。

注意小应用程序没有main()方法，不像Java应用程序，小应用程序不以main()为程序起始。实际上，大多数小应用程序甚至不含main()方法。相反，当小应用程序类名被传输到小应用程序阅读器(applet view)或网络浏览器时它开始执行。

在你键入SimpleApplet的源代码后，用你以前编译程序的方法编译该程序。但是，运行SimpleApplet包含一个完全不同的过程。实际上，有两种方法可以运行小应用程序。

- 在一个兼容Java的网络浏览器，例如Netscape Navigator中执行小应用程序
- 使用小应用程序阅读器，例如标准JDK工具，小应用程序阅览器（appletviewer）。一个小应用程序阅读器在窗口中执行小应用程序。这是检测小应用程序最快和最简单的方法。

上述方法在下面都有阐述。

为在一个网络浏览器中执行小应用程序，需要编写包含适当APPLET标记的简短的HTML文档。下面是执行SimpleApplet的HTML文件：

```
<applet code="SimpleApplet" width=200 height=60>
</applet>
```

width 和height语句指定了小应用程序用到的显示区域的尺寸（APPLET标记包括几个其他的选项，这在第2部分有详细的描述）。创建文件后，你可以启动浏览器并加载可以执

行SimpleApplet的文件。

为使用小应用程序阅读器执行SimpleApplet，你也需执行前面的HTML文件。例如前面所述的HTML文档叫做RunApp.html，则下面的命令行将运行SimpleApplet：

```
C:\>appletviewer RunApp.html
```

然而，存在一个更方便的方法使测试更快的完成。仅仅在你包含APPLET标记的Java源代码的开头加入一个命令。这样做，你的代码就被一个必要的HTML语句原型证明，你只需启动含有JAVA源码文件的小应用程序阅读器就可以测试你编译过的小应用程序。如果你使用该方法，SimpleApplet源文件如下：

```
import java.awt.*;
import java.applet.*;
/*
<applet code="SimpleApplet" width=200 height=60>
</applet>
*/

public class SimpleApplet extends Applet {
  public void paint(Graphics g) {
    g.drawString("A Simple Applet", 20, 20);
  }
}
```

总的来说，你可以使用下面三步来应用小应用程序：

1. 编写Java源程序。
2. 编译程序。
3. 执行小应用程序阅览器，指定小应用程序源文件名称。小应用程序阅览器将在注释中遇到APPLET标记并执行小应用程序。

SimpleApplet生成的窗口，在小应用程序阅览器中显示。该窗口如下面插图：

关于小应用程序的专题在本书后面有更详尽的讨论，下面是需要记住的关键点：

- 小应用程序不一定包含 main( ) 方法。
- 小应用程序必须在小应用程序阅读器或兼容JAVA的浏览器中运行。
- 用户输入/输出不是由Java的输入/输出流类来完成的。相反，小应用程序运用AWT提供的界面。

12.7 Transient和volatile修饰符

Java定义了两类有趣的修饰符：transient和volatile，这些修饰符用来处理特殊的情况。

如果用transient声明一个实例变量，当对象存储时，它的值不需要维持。例如：

```
class T {
  transient int a; // will not persist
  int b; // will persist
}
```

这里，如果T类的一个对象被写入一个持久的存储区域，a的内容不被保存，但b将被保存。

Volatile修饰符告诉编译器被volatile修饰的变量可以被程序的其他部分改变。一种这样的情形是多线程程序（参看第11章的例子）。在多线程程序里，有时两个或更多的线程共享一个相同的实例变量。考虑效率的问题，每个线程可以自己保存该共享变量的私有拷贝。实际的（或主要的）变量副本在不同的时候更新，例如当进入synchronized方法时。当这种方式运行良好时，它在时间上会是低效的。在某些情况，真正要紧的是变量主副本的值会体现当前的状态。为保证这点，仅需把变量定义成volatile型，它告诉编译器它必须总是使用volatile变量的主副本（或者至少总是保持一些私有的最新的主副本的拷贝，反之亦然），同时，对主变量的获取必须以简洁次序执行，就像执行私有拷贝一样。

注意：Java中的volatile或多或少与C/C++中的类似。

12.8 使用instanceof

有时，在运行时间内知道对象类型是很有用的。例如，你有一个执行线程生成各种类型的对象，其他线程处理这些对象。这种情况下，让处理线程在接受对象时知道每一个对象的类型是大有裨益的。另一种在运行时间内知道对象的类型是很有用的情形是强制类型转换。Java中非法强制类型转换导致运行时错误。很多非法的强制类型转换在编译时发生。然而包括类层次结构的强制类型转换可能产生仅能在运行时间里被察觉的非法强制类型转换。例如，一个名为A的父类能生成两个子类B和C。这样，在强制B对象转换为类型A或强制C对象转换为类型A都是合法的，但强制B对象转换为C对象（或相反）都是不合法的。因为类型A的一个对象可以引用B或C。但是你怎么知道，在运行时，在强制转换为C之前哪类对象被引用？它可能是A，B或C的一个对象。如果它是B的对象，一个运行时异常被引发。Java 提供运行时运算符instanceof来解决这个问题。

instanceof运算符具有下面的一般形式：

```
object instanceof type
```

这里，object是类的实例，而type是类的类型。如果object是指定的类型或者可以被强制转换成指定类型，instanceof将它评估成true，若不是，则结果为false。这样，instanceof是

你的程序获得对象运行时类型信息的方法。

下面的程序说明了instanceof的应用：

```
// Demonstrate instanceof operator.
class A {
  int i, j;
}

class B {
  int i, j;
}

class C extends A {
  int k;
}

class D extends A {
  int k;
}

class InstanceOf {
  public static void main(String args[]) {
    A a = new A();
    B b = new B();
    C c = new C();
    D d = new D();

    if(a instanceof A)
      System.out.println("a is instance of A");
    if(b instanceof B)
      System.out.println("b is instance of B");
    if(c instanceof C)
      System.out.println("c is instance of C");
    if(c instanceof A)
      System.out.println("c can be cast to A");

    if(a instanceof C)
      System.out.println("a can be cast to C");

    System.out.println();

    // compare types of derived types
    A ob;

    ob = d; // A reference to d
    System.out.println("ob now refers to d");
    if(ob instanceof D)
      System.out.println("ob is instance of D");

    System.out.println();

    ob = c; // A reference to c
    System.out.println("ob now refers to c");
```

```
    if(ob instanceof D)
      System.out.println("ob can be cast to D");
    else
      System.out.println("ob cannot be cast to D");

    if(ob instanceof A)
      System.out.println("ob can be cast to A");

    System.out.println();

    // all objects can be cast to Object
    if(a instanceof Object)
      System.out.println("a may be cast to Object");
    if(b instanceof Object)
      System.out.println("b may be cast to Object");
    if(c instanceof Object)
      System.out.println("c may be cast to Object");
    if(d instanceof Object)
      System.out.println("d may be cast to Object");
  }
}
```

程序输出如下：

```
a is instance of A
b is instance of B
c is instance of C
c can be cast to A

ob now refers to d
ob is instance of D

ob now refers to c
ob cannot be cast to D
ob can be cast to A

a may be cast to Object
b may be cast to Object
c may be cast to Object
d may be cast to Object
```

多数程序不需要instanceof运算符，因为，一般来说，你知道你正在使用的对象类型。但是，在你编写对复杂类层次结构对象进行操作的通用程序时它是非常有用的。

12.9 strictfp

Java 2向Java语言增加了一个新的关键字strictfp。与Java 2同时产生的浮点运算计算模型很轻松的使某些处理器可以以较快速度进行浮点运算例如奔腾处理器。特别指明，在计算过程中，一个不需要切断某些中介值的新的模型产生了。用strictfp来修饰类或方法，可以确保浮点运算（以及所有切断）正如它们在早期Java版本中那样准确。切断只影响某些操作的指数。当一个类被strictfp修饰，所有该类的方法都自动被strictfp修饰。

举例来说，下面的程序段告诉Java在MyClass中定义的所有方法都用原始浮点运算模型来计算：

```
strictfp class MyClass { //...
```

坦白地说，很多程序员从未用过strictfp，因为它只对非常少的问题有影响。

12.10　本 机 方 法

尽管这种情况极少发生，你也许希望调用不是用Java语言写的子程序。通常，这样的子程序是CPU的或是你所工作环境的执行代码——也就是说，本机代码。例如，你希望调用本机代码子程序来获得较快的执行时间。或者，你希望用一个专用的第三方的库，例如统计学包。然而，因为Java程序被编译为字节码，字节码由Java运行时系统解释（或动态编译），看起来在Java程序中调用本机代码子程序是不可能的。幸运的是，这个结论是错误的。Java提供了native关键字，该关键字用来声明本机代码方法。一旦声明，这些方法可以在Java程序中被调用，就像调用其他Java方法一样。

为声明一个本机方法，在该方法之前用native修饰符，但是不要定义任何方法体。例如：

```
public native int meth() ;
```

声明本机方法后，必须编写本机方法并要执行一系列复杂的步骤使它与Java代码链接。

很多本机方法是用C写的。把C代码结合到Java 程序中的机制是调用Java Native Interface (JNI)。该方法学由Java 1.1创建并在Java 2中增强。（Java 1.0是用不同的方法，该方法已经过时），关于JNI的详尽描述超出了本书的范围。但是下面的描述为多数应用程序提供了足够的信息。

注意：所需执行的精确的步骤随Java 环境和版本的不同而不同，它还依赖于所要实现的本机方法使用的语言。下面的讨论假定在Windows 95/98/NT/2000环境下。所要实现的本机方法是用C写的。

理解该过程的最简单的方法是完成一个例子。开始，输入下面的短程序，该程序使用了一个名为test( )的native方法。

```
// A simple example that uses a native method.
public class NativeDemo {
  int i;
  public static void main(String args[]) {
    NativeDemo ob = new NativeDemo();

    ob.i = 10;
    System.out.println("This is ob.i before the native method:" +
                       ob.i);
    ob.test(); // call a native method
    System.out.println("This is ob.i after the native method:" +
                       ob.i);
  }
  // declare native method
```

```
  public native void test() ;

  // load DLL that contains static method
  static {
    System.loadLibrary("NativeDemo");
  }
}
```

注意test( )方法声明为native且不含方法体。简而言之这是我们用C语言实现的方法。同时注意static块。像本书前面解释过的，一个static块仅在程序开始执行时执行（更为简单的说，当它的类被加载时执行）。这种情况下，它用来加载包含本地执行方法test( )的动态链接库（你不久就会看到怎样创建这个库）。

该库由loadLibrary( )方法加载。loadLibrary( )方法是System类的组成单元。它的一般形式为：

```
static void loadLibrary(String filename)
```

这里，filename是指定保存该库文件名的字符串。在Windows环境下，该文件的扩展名为.DLL。

写完程序后，编译它生成NativeDemo.class。然后，你必须用javah.exe生成一个文件：NativeDemo.h（javah.exe包含在JDK中）。在执行test( )时你要包含NativeDemo.h。为生成NativeDemo.h，用下面的命令：

```
javah -jni NativeDemo
```

该命令生成名为NativeDemo.h的头文件。该文件必须包含在实现test()的C文件中。该命令的输出结果如下：

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class NativeDemo */

#ifndef _Included_NativeDemo
#define _Included_NativeDemo
#ifdef _ _cplusplus
extern "C" {
#endif
/*
 * Class:     NativeDemo
 * Method:    test
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_NativeDemo_test
  (JNIEnv *, jobject);

#ifdef _ _cplusplus
}
#endif
#endif
```

请特别注意下面一行，该行定义了所要创建的test( )函数的原型：

```
JNIEXPORT void JNICALL Java_NativeDemo_test(JNIEnv *, jobject);
```

注意函数的名称是Java_NativeDemo_test( )。调用本机函数你必须用这样的名字。也就是说，不是生成一个名为test( )的C函数，而是创建一个名为Java_NativeDemo_test( )函数。加入前缀NativeDemo是因为它把test( )方法作为NativeDemo类的一部分。记住，其他类可以定义它们自己的与NativeDemo定义的完全不同的本地test( )方法。前缀中包括类名的方法解决了区分不同版本的问题。作为一个常规方法，给本机函数取名，前缀中必须包括声明它们的类名。

生成了必备的头文件后，可以编写test( )执行文件并把它存在一个名为NativeDemo.c的文件中：

```
/* This file contains the C version of the
   test() method.
*/

#include <jni.h>
#include "NativeDemo.h"
#include <stdio.h>

JNIEXPORT void JNICALL Java_NativeDemo_test(JNIEnv *env, jobject obj)
{
  jclass cls;
  jfieldID fid;
  jint i;

  printf("Starting the native method.\n");
  cls = (*env)->GetObjectClass(env, obj);
  fid = (*env)->GetFieldID(env, cls, "i", "I");

  if(fid == 0) {
    printf("Could not get field id.\n");
    return;
  }
  i = (*env)->GetIntField(env, obj, fid);
  printf("i = %d\n", i);
  (*env)->SetIntField(env, obj, fid, 2*i);
  printf("Ending the native method.\n");
}
```

注意此文件包含具有接口信息的jni.h文件。该文件由你的Java 编译器提供。头文件NativeDemo.h预先已由javah创建。

该函数中，GetObjectClass( )方法用来获得一个含有NativeDemo类信息的C结构。GetFieldID( )方法返回一个包含该类域名“i”信息的C结构。GetIntField()检索该域原来的值。SetIntField( )存储该域的一个更新值（别的处理其他数据类型的方法参看文件jni.h）。

生成NativeDemo.c文件后，必须编译它生成一个DLL文件。用微软C/C++编译器来做，使用下面的命令行：

```
Cl /LD NativeDemo.c
```

它生成了一个名为NativeDemo.dll的文件。该步骤完成，你可以执行Java 程序。该程

序输出如下：

```
This is ob.i before the native method: 10
Starting the native method.
i = 10
Ending the native method.
This is ob.i after the native method: 20
```

注意：使用native的特殊环境是依赖于实现和环境的。而且，与JAVA代码接口的指定方式必须改变。你必须仔细考虑完成你Java开发系统文件的本机方法。

12.11 使用本机方法的问题

本机方法看起来提供巨大承诺，因为它们使你有权使用已经存在的库程序，而且使快速执行成为可能。但是本机方法同样引入了两个重大问题：

- 潜在的安全隐患　因为本机方法执行实际的机器代码，它有权使用主机系统的任何资源。也就是说，本机代码不受Java执行环境的限制。例如，它可能允许病毒入侵。因为这个原因，小应用程序不能使用本机方法。同样，DLL文件的加载被限制，它们的加载必须经过安全管理器的同意。
- 丧失了可移植性　因为本机代码是包含在DLL文件中的，它必须存在于执行Java程序的机器上。而且，因为每一个本机方法都依赖于CPU和操作系统，每一个DLL文件在本质上都是不可可移植性的。这样，一个运用本机方法的Java程序只能在一个已经安装了可兼容的DLL的机器上运行。

本机方法的使用是受限的，因为它使Java程序丧失了可移植性且造成重大安全隐患。

第2部分　Java库

第 13 章　字符串处理

在第7章已对Java的字符串处理做了简要的介绍。本章将对此做详细论述。像大多数其他计算机语言一样，Java中的字符串也是一连串的字符。但是与许多其他的计算机语言将字符串作为字符数组处理不同，Java将字符串作为String类型对象来处理。

将字符串作为内置的对象处理允许Java提供十分丰富的功能特性以方便处理字符串。例如，Java语言中有多种方法用于比较两个字符串，搜索子字符串，连接字符串以及改变字符串中字母的大小写。也有许多途径可以构造出String对象，使得当需要时，能够容易得到字符串。

有些出乎意料的是当创建一个String对象时，被创建的字符串是不能被改变的。这也就是说一旦一个String对象被创建，将无法改变那些组成字符串的字符。表面上看起来，这好像是一个严格的约束。然而事实并非如此。你仍能够执行各种类型的字符串操作。区别在于每次需要改变字符串时都要创建一个新的String对象来保存新的内容。原始的字符串不变。之所以采用这种方法是因为实现固定的，不可变的字符串比实现可变的字符串更高效。对于那些想得到改变的字符串的情况，有一个叫做StringBuffer的String类的友类。它的对象包含了在创建之后可被改变的字符串。

String类和StringBuffer类都在java.lang中定义。因此它们可以自动的被所有程序利用。两者均被说明为final，这意味着两者均不含子类。从而使得某些增强性能的优化可作用于普通字符串操作。

最后需要指明一点：包含在类型String对象中的字符串的不可改变意味着String实例一旦被建立，它的内容将不能被改变。然而在任何时候，被说明为String引用的变量可以被改变以指向另外的一些字符串（String）对象。

13.1　String构造函数

String类支持几种构造函数。将创建空String的构造函数称为默认构造函数。例如，

```
String s = new String();
```

将创建一个String实例，该实例中不包含字符。

通常希望创建含有初始值的字符串。String类提供了各种构造函数来完成这项功能。使

用如下的构造函数可以创建一个被字符数组初始化的字符串（String）：

```
String(char chars[ ])
```

下面是一个例子：

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
```

这个构造函数用字符串“abc”初始化s。

使用下面的构造函数可以指定字符数组的一个子区域作为初始化值。

```
String(char chars[ ], int startIndex, int numChars)
```

这里，startIndex指定了子区域开始的下标，numChars指定所用字符的个数。下面是一个例子：

```
char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
String s = new String(chars, 2, 3);
```

该例子用字符cde初始化s。

用下面的构造函数可以构造一个String对象，该对象包括了与另一个String对象相同的字符序列。

```
String(String strObj)
```

这里strObj是一个String对象，请看如下例子：

```
// Construct one String from another.
class MakeString {
  public static void main(String args[]) {
    char c[] = {'J', 'a', 'v', 'a'};
    String s1 = new String(c);
    String s2 = new String(s1);

    System.out.println(s1);
    System.out.println(s2);
  }
}
```

程序的输出如下所示：

```
Java
Java
```

正如你所能看到的，s1和s2包含了相同的字符串。

尽管Java的Char类型使用16位（bit）表示Unicode编码字符集，在Internet中，字符串的典型格式使用由ASCII字符集构成的8位字节数组。因为8位ASCII字符串是共同的，当给定一个字节（byte）数组时，String类提供了初始化字符串的构造函数。它们的形式如下：

```
String(byte asciiChars[ ])
String(byte asciiChars[ ],  int startIndex, int numChars)
```

这里asciiChars指定了字节数组。第二种形式允许指定一个子区域。在这些构造函数中，通过使用操作平台默认的字符编码实现了由字节到字符的转换，下面的程序举例说明了这些构造函数：

```
// Construct string from subset of char array.
class SubStringCons {
  public static void main(String args[]) {
    byte ascii[] = {65, 66, 67, 68, 69, 70 };

    String s1 = new String(ascii);
    System.out.println(s1);

    String s2 = new String(ascii, 2, 3);
    System.out.println(s2);
  }
}
```

该程序运行产生如下的输出：

```
ABCDEF
CDE
```

字节-字符串转换的扩展版本也有定义，使用该版本，你可以指定实现字节-字符串转换的字符编码方式。不过，大多数情况下，一般会选择操作平台提供的默认编码。

注意：当从一个数组创建一个String对象时，数组的内容将被复制。在字符串被创建以后，如果改变数组的内容，String将不会随之改变。

13.2　字符串长度

字符串的长度是指其所包含的字符的个数。调用如下的length( )方法可以得到这个值：

```
int length( )
```

下面的程序段输出“3”，因为在字符串s中有三个字符。

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
System.out.println(s.length());
```

13.3　特殊的字符串操作

因为字符串是程序中的一个通用和重要的部分，Java为字符串操作在语法中增加了特殊的支持。这些操作包括从字符串常量自动创建新的String实例。通过+运算符连接多个String对象以及将其他的数据类型转换成字符串形式。尽管有显式的方法去执行这些函数，而为了方便程序员，Java会自动执行它们。

13.3.1 字符串文字

前面的例子说明了如何通过使用new运算符从一个字符数组明确地创建一个String实例。然而这是一种早期的使用字符串常量的处理方法。对于程序中的每一个字符串常量，Java会自动创建String对象。因此，可以使用字符串常量初始化String对象。例如，如下的程序代码段创建两个相等的字符串。

```
char chars[] = { 'a', 'b', 'c' };
String s1 = new String(chars);

String s2 = "abc"; // use string literal
```

由于对应每一个字符串常量，有一String对象被创建，因此在使用字符串文字的任何地方，都可以使用String对象。例如，可以直接对引用字符串调用方法，如同它是一个对象引用，如下面语句所显示的那样。以字符串“abc”调用方法length( )，正如所料，输出“3”。

```
System.out.println("abc".length());
```

13.3.2 字符串连接

通常，Java不允许对String对象进行操作。这一规则的一个例外是+运算符，它可以连接两个字符串，产生一个String对象。也允许使用一连串的+运算符，例如下面的程序段将三个字符串连接起来：

```
String age = "9";
String s = "He is " + age + " years old.";
System.out.println(s);
```

输出为字符串“He is 9 years old.”。

字符串连接的一个实际使用是当创建一个很长的字符串时，可以将它拆开，使用+将它们连接起来，避免源代码中长字符串的换行，下面是一个例子：

```
// Using concatenation to prevent long lines.
class ConCat {
  public static void main(String args[]) {
    String longStr = "This could have been " +
      "a very long line that would have " +
      "wrapped around.  But string concatenation " +
      "prevents this.";

    System.out.println(longStr);
  }
}
```

13.3.3 字符串与其他类型数据的连接

字符串可以和其他类型的数据连接。例如，考虑与前面的例子略有差别的一个例子：

```
int age = 9;
String s = "He is " + age + " years old.";
System.out.println(s);
```

在这里，age是一个整型（int）而不是另一个字符串（String）型值，但是程序的输出与前面的例子相同。这是因为在字符串（String）对象中age的整型（int）值自动转换为它的字符串（String）形式。然后这个字符串就和前面一样被连接。只要+运算符的一个运算数是字符串（String）实例，编译器就将另一个运算数转换为它的字符串形式。

应当小心的是当你将其他类型的操作与字符串连接表达式混合时，有可能得到意想不到的结果。看下面例子：

```
String s = "four: " + 2 + 2;
System.out.println(s);
```

程序显示：

```
four: 22
```

而不是

```
four: 4
```

虽然这可能是你期望得到的。这是为什么呢？这是因为运算符的优先级造成了首先是“four”和与2相应的字符串的连接，这个连接的结果再和与2相应的字符串连接。若要先实现整数加，就应该使用圆括号。像下面这样：

```
String s = "four: " + (2 + 2);
```

现在s包含了字符串“four：4”。

13.3.4　字符串转换和toString( )

当Java在连接时将数据转换为其字符串形式时，它是通过调用一个由字符串（String）定义的字符串转换方法valueOf( )的重载来完成的。valueOf( )方法对所有简单的类型和类型Object重载。对于简单类型，valueOf( )方法返回一个字符串，该字符串包含了相应其被调用的值的可读值。对于对象，valueOf( )方法调用toString( )方法。在本章后面我们将更仔细地分析valueOf( )方法。这里让我们讨论toString( )方法，因为通过它可以确定所创建类的对象的字符串形式。

每一个类都执行toString( )方法，因为它是由Object定义的。然而toString( )方法的默认实现是不够的。对于所创建的大多数类，通常想用你自己提供的字符串表达式重载toString( )方法。幸运的是这很容易做到。toString( )方法具有如下的一般形式：

```
String toString( )
```

实现toString( )方法，仅仅返回一个String对象，该对象包含描述类中对象的可读的字符串。

通过对所创建类的toString( )方法的覆盖，允许将得到的字符串完全集成到Java的程序设计环境中。例如它们可以被用于print( )和println( )语句以及连接表达式中。下面的程序中用Box类重载toString( )方法说明了这些。

```
// Override toString() for Box class.
class Box {
```

```
  double width;
  double height;
  double depth;

  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  public String toString() {
    return "Dimensions are " + width + " by " +
           depth + " by " + height + ".";
  }
}

class toStringDemo {
  public static void main(String args[]) {
    Box b = new Box(10, 12, 14);
    String s = "Box b: " + b; // concatenate Box object

    System.out.println(b); // convert Box to string
    System.out.println(s);
  }
}
```

该程序的输出如下所示：

```
Dimensions are 10.0 by 14.0 by 12.0
Box b: Dimensions are 10.0 by 14.0 by 12.0
```

正如你能看到的那样，当Box对象在连接表达式中使用或出现在调用println( )中时，Box的toString( )方法被自动调用。

13.4 字 符 截 取

String类提供了许多从String对象中截取字符的方法。下面逐一介绍。尽管在一个String对象中构成字符串的字符不能像字符数组一样被索引，许多字符串（String）方法利用下标（或偏移）对字符串进行操作。和数组一样，字符串下标从0开始。

13.4.1 charAt( )

为了从一个字符串（String）中截取一个字符，可以通过charAt( )方法直接引用单个字符。其一般形式如下：

```
char charAt(int where)
```

这里，where是想要得到的字符的下标。where的值必须是非负的，它指定了在字符串中的位置。charAt( )方法返回指定位置的字符。例如，

```
char ch;
```

```
ch = "abc".charAt(1);
```

将“b”赋给ch。

13.4.2　getChars( )

如果想一次截取多个字符，可以使用getChars( )方法。它有如下的一般形式：

```
void getChars(int sourceStart, int sourceEnd, char target[ ], int
targetStart)
```

这里sourceStart指定了子字符串开始的下标，sourceEnd指定了子字符串结束的下一个字符的下标。因此子字符串包含了从sourceStart到sourceEnd–1的字符。获得字符的数组由target所指定。将被复制子字符串于其中的target的下标由targetStart指定。注意必须确保的是数组target应该足够大以保证能容纳被指定子字符串中的字符。

下面的程序说明了getChars( )方法：

```
class getCharsDemo {
  public static void main(String args[]) {
    String s = "This is a demo of the getChars method.";
    int start = 10;
    int end = 14;
    char buf[] = new char[end - start];

    s.getChars(start, end, buf, 0);
    System.out.println(buf);
  }
}
```

该程序的输出如下所示：

```
demo
```

13.4.3　getBytes( )

有一种称为getBytes( )的方法，它是实现将字符存放于字节数组中的getChars( )方法的替代，它使用平台提供的默认的字符到字节的转换。下面是它的最简单形式：

```
byte[ ] getBytes( )
```

也可使用getBytes( )方法的其他形式。在将字符串（String）值输出到一个不支持16位Unicode编码的环境时，getBytes( )是最有用的。例如，大多数Internet协议和文本文件格式在文本交换时使用8位ASCII编码。

13.4.4　toCharArray( )

如果想将字符串（String）对象中的字符转换为一个字符数组，最简单的方法就是调用toCharArray( )方法。对应整个字符串，它返回一个字符数组。其一般形式为：

```
char[ ] toCharArray( )
```

这个函数是为了便于使用而提供的，因此也可以用getChars( )方法获得相同的结果。

13.5 字符串比较

String类包括了几个用于比较字符串或字符串内子字符串的方法。下面分别对它们进行介绍。

13.5.1 equals( )和 equalsIgnoreCase( )

使用equals( )方法比较两个字符串是否相等。它具有如下的一般形式：

```
boolean equals(Object str)
```

这里str是一个用来与调用字符串（String）对象做比较的字符串（String）对象。如果两个字符串具有相同的字符和长度，它返回true，否则返回false。这种比较是区分大小写的。

为了执行忽略大小写的比较，可以调用equalsIgnoreCase( )方法。当比较两个字符串时，它会认为A-Z和a-z是一样的。其一般形式如下：

```
boolean equalsIgnoreCase(String str)
```

这里，str是一个用来与调用字符串（String）对象做比较的字符串（String）对象。如果两个字符串具有相同的字符和长度，它也返回true，否则返回false。

下面的例子说明了equals( )和equalsIgnoreCase( )方法：

```
// Demonstrate equals() and equalsIgnoreCase().
class equalsDemo {
  public static void main(String args[]) {
    String s1 = "Hello";
    String s2 = "Hello";
    String s3 = "Good-bye";
    String s4 = "HELLO";
    System.out.println(s1 + " equals " + s2 + " -> " +
                       s1.equals(s2));
    System.out.println(s1 + " equals " + s3 + " -> " +
                       s1.equals(s3));
    System.out.println(s1 + " equals " + s4 + " -> " +
                       s1.equals(s4));
    System.out.println(s1 + " equalsIgnoreCase " + s4 + " -> " +
                       s1.equalsIgnoreCase(s4));
  }
}
```

该程序的输出如下所示：

```
Hello equals Hello -> true
Hello equals Good-bye -> false
Hello equals HELLO -> false
Hello equalsIgnoreCase HELLO -> true
```

13.5.2　regionMatches( )

regionMatches( )方法将一个字符串中指定的区间和另一字符串中指定的区间进行比较。它的重载形式允许在比较时忽略大小写。下面给出这两种方法的一般形式：

```
boolean regionMatches(int startIndex, String str2,
                                    int str2StartIndex, int numChars)
boolean regionMatches(boolean ignoreCase,
                                    int startIndex, String str2,
                                    int str2StartIndex, int numChars)
```

对于这两种形式，startIndex指定了调用字符串（String）对象内区间开始的下标。用于比较的字符串（String）由str2指定的。在str2内，开始比较区间的下标由str2StartIndex指定。用于比较的子字符串的长度在numChars中。在第二种方案中，如果ignoreCase是true，字符的大小写被忽略。否则，大小写是有意义的。

13.5.3　startsWith( )和endsWith( )

字符串（String）定义两个例程，它们或多或少是regionMatches( )方法的特殊形式。startsWith( )方法判断一个给定的字符串（String）是否从一个指定的字符串开始。相反地，endsWith( )方法判断所讨论的字符串（String）是否是以一个指定的字符串结尾。它们具有如下的一般形式：

```
boolean startsWith(String str)
boolean endsWith(String str)
```

这里，str是被测试的字符串（String）。如果字符串匹配，返回true。否则返回false。例如：

```
"Foobar".endsWith("bar")
```

和

```
"Foobar".startsWith("Foo")
```

都是true。

下面给出startsWith( )方法的第二种形式。这种形式允许指定一个起始点：

```
boolean startsWith(String str, int startIndex)
```

这里，startIndex指定了调用字符串开始搜索的下标。例如，

```
"Foobar".startsWith("bar", 3)
```

返回true。

13.5.4　equals( )与==的比较

理解equals( )方法和==运算符执行的是两个不同的操作是重要的。如同刚才解释的那样，equals( )方法比较字符串（String）对象中的字符。而==运算符比较两个对象引用看它们是否引用相同的实例。下面的程序说明了两个不同的字符串（String）对象是如何能够包

含相同字符的，但同时这些对象引用是不相等的：

```
// equals() vs ==
class EqualsNotEqualTo {
  public static void main(String args[]) {
    String s1 = "Hello";
    String s2 = new String(s1);

    System.out.println(s1 + " equals " + s2 + " -> " +
                       s1.equals(s2));
    System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2));
  }
}
```

变量s1指向由“Hello”创建的字符串（String）实例。s2所指的的对象是以s1作为初始化而创建的。因此这两个字符串（String）对象的内容是一样的。但它们是不同的对象，这就意味着s1和s2没有指向同一的对象，因此它们是不==的，上面例子的结果如下：

```
Hello equals Hello -> true
Hello == Hello -> false
```

13.5.5 compareTo( )

通常，仅仅知道两个字符串是否相同是不够的。对于排序应用来说，必须知道一个字符串是大于、等于还是小于另一个。一个字符串小于另一个指的是它在字典中先出现。而一个字符串大于另一个指的是它在字典中后出现。字符串（String）的compareTo( )方法实现了这种功能。它的一般形式如下：

```
int compareTo(String str)
```

这里str是与调用字符串（String）比较的字符串（String）。比较的结果返回并被解释如表13-1所示：

表 13.1 字符串比较的结果及其含义

值	含义
小于0	调用字符串小于str
大于0	调用字符串大于str
等于0	两个字符串相等

下面是一个对字符串数组进行排序的例子程序。程序中在冒泡法排序中使用compareTo( )方法确定排序的顺序：

```
// A bubble sort for Strings.
class SortString {
  static String arr[] = {
    "Now", "is", "the", "time", "for", "all", "good", "men",
    "to", "come", "to", "the", "aid", "of", "their", "country"
  };
  public static void main(String args[]) {
    for(int j = 0; j < arr.length; j++) {
```

```
        for(int i = j + 1; i < arr.length; i++) {
          if(arr[i].compareTo(arr[j]) < 0) {
            String t = arr[j];
            arr[j] = arr[i];
            arr[i] = t;
          }
        }
        System.out.println(arr[j]);
      }
    }
}
```

程序的输出是如下的单词表：

```
Now
aid
all
come
country
for
good
is
men
of
the
the
their
time
to
to
```

正如从这个例子的输出所能看到的，compareTo( )方法区分单词的大小写。单词“Now”因为是以大写字母开始的而出现在其他单词的前面，这意味着它在ASCII字符集中具有更小的值。

如果想在比较两个字符串时，忽略大小写，可以使用如下的compareToIgnoreCase( )方法：

```
int compareToIgnoreCase(String str)
```

除了忽略大小写之外，该方法的返回值与compareTo( )方法相同。该方法是在Java 2中新增加的。可以在前面的程序中换成这个方法。这样做了之后，“Now”将不再是第一个输出了。

13.6　搜索字符串

String类提供了两个方法，允许在字符串中搜索指定的字符或子字符串：

- indexOf( )　搜索字符或子字符串首次出现。
- lastIndexOf( )　搜索字符或子字符串的最后一次出现。

这两种方法被几种不同的方法重载。在所有这些情况下，方法返回字符或子字符串被发现的位置的下标，当搜索失败时，返回-1。

搜索字符首次出现用

```
int indexOf(int ch)
```

搜索字符最后一次出现用

```
int lastIndexOf(int ch)
```

这里ch是被查找的字符。

搜索子字符串首次或最后一次出现用

```
int indexOf(String str)
int lastIndexOf(String str)
```

这里子字符串由str指定

可以使用如下这些形式指定搜索的起始点：

```
int indexOf(int ch, int startIndex)
int lastIndexOf(int ch, int startIndex)

int indexOf(String str, int startIndex)
int lastIndexOf(String str, int startIndex)
```

这里startIndex指定了搜索开始点的下标。对于indexOf( )方法，搜索从startIndex开始到字符串结束。对于lastIndexOf( )方法，搜索从startIndex开始到下标0。

下面的例子说明如何利用不同的索引方法在字符串（String）的内部进行搜索：

```
// Demonstrate indexOf() and lastIndexOf().
class indexOfDemo {
  public static void main(String args[]) {
    String s = "Now is the time for all good men " +
               "to come to the aid of their country.";

    System.out.println(s);
    System.out.println("indexOf(t) = " +
                       s.indexOf('t'));
    System.out.println("lastIndexOf(t) = " +
                       s.lastIndexOf('t'));
    System.out.println("indexOf(the) = " +
                       s.indexOf("the"));
    System.out.println("lastIndexOf(the) = " +
                       s.lastIndexOf("the"));
    System.out.println("indexOf(t, 10) = " +
                       s.indexOf('t', 10));
    System.out.println("lastIndexOf(t, 60) = " +
                       s.lastIndexOf('t', 60));
    System.out.println("indexOf(the, 10) = " +
                       s.indexOf("the", 10));
    System.out.println("lastIndexOf(the, 60) = " +
                       s.lastIndexOf("the", 60));
  }
```

```
}
```

下面是该程序的输出结果：

```
Now is the time for all good men to come to the aid of their country.
indexOf(t) = 7
lastIndexOf(t) = 65
indexOf(the) = 7
lastIndexOf(the) = 55
indexOf(t, 10) = 11
lastIndexOf(t, 60) = 55
indexOf(the, 10) = 44
lastIndexOf(the, 60) = 55
```

13.7　修改字符串

因为字符串（String）对象是不可改变的，每当想修改一个字符串（String）时，就必须采用或者将它复制到StringBuffer或者使用下面字符串（String）方法中的一种，这些方法都将构造一个完成修改的字符串的拷贝。

13.7.1　substring( )

利用substring( )方法可以截取子字符串，它有两种形式。其中第一种形式如下：

```
String substring(int startIndex)
```

这里startIndex指定了子字符串开始的下标。这种形式返回一个从startIndex开始到调用字符串结束的子字符串的拷贝。

substring( )方法的第二种形式允许指定子字符串的开始和结束下标：

```
String substring(int startIndex, int endIndex)
```

这里startIndex指定开始下标，endIndex指定结束下标。返回的字符串包括从开始下标直到结束下标的所有字符，但不包括结束下标对应的字符。

下面的程序使用substring( )方法完成在一个字符串内用一个子字符串替换另一个子字符串的所有实例的功能：

```
// Substring replacement.
class StringReplace {
  public static void main(String args[]) {
    String org = "This is a test. This is, too.";
    String search = "is";
    String sub = "was";
    String result = "";
    int i;

    do { // replace all matching substrings
      System.out.println(org);
      i = org.indexOf(search);
      if(i != -1) {
        result = org.substring(0, i);
```

```
        result = result + sub;
        result = result + org.substring(i + search.length());
        org = result;
      }
    } while(i != -1);

  }
}
```

这个程序的输出如下所示：

```
This is a test. This is, too.
Thwas is a test. This is, too.
Thwas was a test. This is, too.
Thwas was a test. Thwas is, too.
Thwas was a test. Thwas was, too.
```

13.7.2 concat( )

使用concat( )可以连接两个字符串，一般形式如下：

```
String concat(String str)
```

这个方法创建一个新的对象，该对象包含调用字符串。而str的内容跟在调用字符串的后面。concat( )方法与+运算符执行相同的功能。例如，

```
String s1 = "one";
String s2 = s1.concat("two");
```

将字符串“onetwo”赋给s2。它和下面的程序段产生相同的结果：

```
String s1 = "one";
String s2 = s1 + "two";
```

13.7.3 replace( )

replace( )方法用另一个字符代替调用字符串中一个字符的所有具体值。它具有如下的一般形式：

```
String replace(char original, char replacement)
```

这里original指定被由replacement指定的字符所代替的字符，返回得到的字符串。例如：

```
String s = "Hello".replace('l', 'w');
```

将字符串“Hewwo”赋给s。

13.7.4 trim( )

trim( )方法返回一个调用字符串的拷贝，该字符串是将位于调用字符串前面和后面的空白符删除后的剩余部分。它的一般形式如下：

```
String trim( )
```

这里是一个例子：

```
String s = "   Hello World    ".trim();
```

将字符串“Hello World”赋给s。

trim( )方法在处理用户输入的命令时，是十分有用的。例如，下面的程序提示用户输入一个州名后显示该州的首府名。程序中使用trim( )方法删除在用户输入期间，不经意间输入的任何前缀或后缀空白符。

```
// Using trim() to process commands.
import java.io.*;

class UseTrim {
  public static void main(String args[])
    throws IOException
  {
    // create a BufferedReader using System.in
    BufferedReader br = new
      BufferedReader(new InputStreamReader(System.in));
    String str;

    System.out.println("Enter 'stop' to quit.");
    System.out.println("Enter State: ");
    do {
      str = br.readLine();
      str = str.trim(); // remove whitespace

      if(str.equals("Illinois"))
        System.out.println("Capital is Springfield.");
      else if(str.equals("Missouri"))
        System.out.println("Capital is Jefferson City.");
      else if(str.equals("California"))
        System.out.println("Capital is Sacramento.");
      else if(str.equals("Washington"))
        System.out.println("Capital is Olympia.");
      // ...
    } while(!str.equals("stop"));
  }
}
```

13.8　利用valueOf( )方法实现数据转换

valueOf( )方法将数据的内部格式转换为可读的形式。它是一种静态方法，对于所有Java内置的类型，在字符串（String）内被重载，以便每一种类型都能被转换成字符串。valueOf( )方法还被类型Object重载，所以创建的任何形式类的对象也可被用作一个参数（我们知道Object是所有的类的超类）。这里是它的几种形式：

```
static String valueOf(double num)
static String valueOf(long num)
static String valueOf(Object ob)
static String valueOf(char chars[ ])
```

与前面的讨论一样，调用valueOf( )方法可以得到其他类型数据的字符串形式——例如在进行连接操作时。对各种数据类型，可以直接调用这种方法得到合理的字符串（String）形式。所有的简单类型数据转换成相应于它们的普通字符串（String）形式。任何传递给valueOf( )方法的对象都将返回对象的toString( )方法调用的结果。事实上，也可以通过直接调用toString( )方法而得到相同的结果。

对大多数数组，valueOf( )方法返回一个相当晦涩的字符串，这说明它是一个某种类型的数组。然而对于字符（char）数组，它创建一个包含了字符（char）数组中的字符的字符串（String）对象。valueOf( )方法有一种特定形式允许指定字符（char）数组的一个子集。它具有如下的一般形式：

```
static String valueOf(char chars[ ], int startIndex, int numChars)
```

这里chars是存放字符的数组，startIndex是字符数组中期望得到的子字符串的首字符下标，numChars指定子字符串的长度。

13.9 改变字符串内字符的大小写

toLowerCase( )方法将字符串内的所有字符从大写字母改写为小写字母。而toUpperCase( )方法将字符串内所有字符从小写字母改写为大写字母。对于那些非字母字符，如数字等则不受影响。下面是这些方法的一般形式：

```
String toLowerCase( )
String toUpperCase( )
```

两种方法返回与调用字符串（String）对应的大写或小写的字符串（String）对象。

下面是一个使用toLowerCase( )和toUpperCase( )方法的例子：

```
// Demonstrate toUpperCase() and toLowerCase().

class ChangeCase {
  public static void main(String args[])
  {
    String s = "This is a test.";

    System.out.println("Original: " + s);

    String upper = s.toUpperCase();
    String lower = s.toLowerCase();

    System.out.println("Uppercase: " + upper);
    System.out.println("Lowercase: " + lower);
  }
}
```

程序的输出显示如下：

```
Original: This is a test.
Uppercase: THIS IS A TEST.
```

```
Lowercase: this is a test.
```

13.10　StringBuffer

StringBuffer是提供了大量的字符串功能的字符串（String）类的对等类。正如你所知，字符串（String）表示了定长，不可变的字符序列。相反，StringBuffer表示了可变长的和可写的字符序列。StringBuffer可有插入其中或追加其后的字符或子字符串。StringBuffer可以针对这些添加自动地增加空间，同时它通常还有比实际需要更多的预留字符，从而允许增加空间。Java大量使用这两种类，但是多数程序员仅仅处理字符串（String）而通过重载+运算符让Java在后台处理StringBuffer。

13.10.1　StringBuffer构造函数

StringBuffer定义了下面三个构造函数：

```
StringBuffer( )
StringBuffer(int size)
StringBuffer(String str)
```

默认构造函数（无参数）预留了16个字符的空间。该空间不需再分配。第二种形式接收一个整数参数，清楚地设置缓冲区的大小。第三种形式接收一个字符串（String）参数，设置StringBuffer对象的初始内容，同时不进行再分配地多预留了16个字符的空间。当没有指定缓冲区的大小时，StringBuffer分配了16个附加字符的空间，这是因为再分配在时间上代价很大。而且频繁地再分配可以产生内存碎片。StringBuffer通过给一些额外的字符分配空间，减少了再分配操作发生的次数。

13.10.2　length( )和capacity( )

通过调用length( )方法可以得到当前StringBuffer的长度。而通过调用capacity( )方法可以得到总的分配容量。它们的一般形式如下：

```
int length( )
int capacity( )
```

这里是一个例子：

```
// StringBuffer length vs. capacity.
class StringBufferDemo {
  public static void main(String args[]) {
    StringBuffer sb = new StringBuffer("Hello");

    System.out.println("buffer = " + sb);
    System.out.println("length = " + sb.length());
    System.out.println("capacity = " + sb.capacity());
  }
}
```

下面是这个程序的输出，它说明了StringBuffer如何为另外的处理预留额外的空间：

```
buffer = Hello
length = 5
capacity = 21
```

由于sb在创建时由字符串“Hello”初始化，因此它的长度为5。因为给16个附加的字符自动增加了存储空间，因此它的存储容量为21。

13.10.3 ensureCapacity( )

如果想在构造StringBuffer之后为某些字符预分配空间，可以使用ensureCapacity( )方法设置缓冲区的大小。这在事先已知要在StringBuffer上追加大量小字符串的情况下是有用的。ensureCapacity( )方法具有如下的一般形式：

```
void ensureCapacity(int capacity)
```

这里capacity指定了缓冲区的大小。

13.10.4 setLength( )

使用setLength( )方法在StringBuffer对象内设置缓冲区的大小。其一般形式如下：

```
void setLength(int len)
```

这里len指定了缓冲区的长度。这个值必须是非负的。

当增加缓冲区的大小时，空字符将被加在现存缓冲区的后面。如果用一个小于length( )方法返回的当前值的值调用setLength( )方法，那么在新长度之后存储的字符将被丢失。后面的setCharAtDemo例子程序使用setLength( )方法缩短StringBuffer。

13.10.5 charAt( )和setCharAt( )

使用charAt( )方法可以从StringBuffer中得到单个字符的值。可以通过setCharAt( )方法给StringBuffer中的字符置值。它们的一般形式如下：

```
char charAt(int where)
void setCharAt(int where, char ch)
```

对于charAt( )方法，where指定获得的字符的下标。对于setCharAt( )方法，where指定被置值的字符的下标，而ch指定了该字符的新值。对于这两种方法，where必须是非负的，同时不能指定在缓冲区之外的位置。

下面的例子说明了charAt( )和setCharAt( )方法：

```
// Demonstrate charAt() and setCharAt().
class setCharAtDemo {
  public static void main(String args[]) {
    StringBuffer sb = new StringBuffer("Hello");
    System.out.println("buffer before = " + sb);
    System.out.println("charAt(1) before = " + sb.charAt(1));
      sb.setCharAt(1, 'i');
      sb.setLength(2);
      System.out.println("buffer after = " + sb);
      System.out.println("charAt(1) after = " + sb.charAt(1));
```

```
  }
}
```

下面是该程序的输出结果：

```
buffer before = Hello
charAt(1) before = e
buffer after = Hi
charAt(1) after = i
```

13.10.6　getChars( )

使用getChars( )方法将StringBuffer的子字符串复制给数组。其一般形式如下：

```
void getChars(int sourceStart, int sourceEnd, char target[ ],
              int targetStart)
```

这里，sourceStart指定子字符串开始时的下标，而sourceEnd指定了该子字符串结束时下一个字符的下标。这意味着子字符串包含了从sourceStart到sourceEnd–1位置上的字符。接收字符的数组由target指定。在target内将被复制子字符串的位置下标由targetStart传递。应注意确保target数组足够大以便能够保存指定的子字符串所包含的字符。

13.10.7　append( )

append( )方法将任一其他类型数据的字符串形式连接到调用StringBuffer对象的后面。对所有内置的类型和Object，它都有重载形式。下面是其几种形式：

```
StringBuffer append(String str)
StringBuffer append(int num)
StringBuffer append(Object obj)
```

每个参数调用String.valueOf( )方法获得其字符串表达式。结果追加在当前StringBuffer对象后面。对每一种append( )形式，返回缓冲区本身。它允许后续的调用被连成一串，下面的例子说明了这一点：

```
// Demonstrate append().
class appendDemo {
  public static void main(String args[]) {
    String s;
    int a = 42;
    StringBuffer sb = new StringBuffer(40);

    s = sb.append("a = ").append(a).append("!").toString();
    System.out.println(s);
  }
}
```

程序的输出如下所示：

```
a = 42!
```

当对字符串（String）对象使用+运算符时，append( )方法是最常被调用的。Java自动地

改变对字符串（String）实例的修改，就像对StringBuffer实例的操作一样。因此，连接调用StringBuffer对象的append( )方法。在执行连接之后，编译器插入对toString( )方法的调用，将修改的StringBuffer返回到一个不变的字符串（String）中。所有这一切看起来是很复杂的。为什么不是仅仅只有一个其操作或多或少地像StringBuffer的字符串类呢？答案是性能。Java运行时执行的许多优化是知道字符串（String）对象是不可改变的。值得欣慰的是Java隐藏了大多数复杂的String与StringBuffer之间的转换。实际上，大多数的程序员从没有直接感觉到需要使用StringBuffer，而可以根据应用于字符串（String）变量上的+运算符表示大多数的操作。

13.10.8 insert( )

insert( )方法将一个字符串插入另一个字符串中。它被重载而接收所有简单类型的值，包括String和Object。和append( )方法一样，它调用String.valueOf( )方法得到调用它的值的字符串表达式。随后这个字符串被插入所调用的StringBuffer对象中。下面是它们的几种形式：

```
StringBuffer insert(int index, String str)
StringBuffer insert(int index, char ch)
StringBuffer insert(int index, Object obj)
```

这里index指定将字符串插入所调用的StringBuffer对象中的插入点的下标。

下面的例子程序完成在“I”和“Java”之间插入“like”的功能。

```
// Demonstrate insert().
class insertDemo {
  public static void main(String args[]) {
    StringBuffer sb = new StringBuffer("I Java!");

    sb.insert(2, "like ");
    System.out.println(sb);
  }
}
```

程序的输出结果如下所示：

```
I like Java!
```

13.10.9 reverse( )

可以使用reverse( )方法将StringBuffer对象内的字符串翻转，其一般形式如下：

```
StringBuffer reverse( )
```

这种方法返回被调用对象的翻转对象。下面的程序说明了reverse( )方法：

```
// Using reverse() to reverse a StringBuffer.
class ReverseDemo {
  public static void main(String args[]) {
    StringBuffer s = new StringBuffer("abcdef");
```

```
    System.out.println(s);
    s.reverse();
    System.out.println(s);
  }
}
```

程序的输出结果如下所示：

```
abcdef
fedcba
```

13.10.10　delete( )和deleteCharAt( )

Java 2在StringBuffer中增加了用于删除字符串的方法delete( )和deleteCharAt( )。这些方法的一般形式如下：

```
StringBuffer delete(int startIndex, int endIndex)
StringBuffer deleteCharAt(int loc)
```

delete( )方法从调用对象中删除一串字符。这里startIndex指定了需删除的第一个字符的下标，而endIndex指定了需删除的最后一个字符的下一个字符的下标。因此要删除的子字符串从startIndex到endIndex–1，返回结果的StringBuffer对象。

deleteCharAt( )方法删除由loc指定下标处的字符，返回结果的StringBuffer对象。

这里是一个说明delete( )和deleteCharAt( )方法的程序。

```
// Demonstrate delete() and deleteCharAt()
class deleteDemo {
  public static void main(String args[]) {
    StringBuffer sb = new StringBuffer("This is a test.");

    sb.delete(4, 7);
    System.out.println("After delete: " + sb);

    sb.deleteCharAt(0);
    System.out.println("After deleteCharAt: " + sb);
  }
}
```

程序输出如下所示：

```
After delete: This a test.
After deleteCharAt: his a test.
```

13.10.11　replace( )

Java 2在StringBuffer中增加的另一个方法是replace( )。它完成在StringBuffer内部用一组字符代替另一组字符的功能。它的形式如下：

```
StringBuffer replace(int startIndex, int endIndex, String str)
```

被替换的子字符串由下标startIndex和endIndex指定。因此从startIndex到endIndex-1的子字符串被替换。替代字符串在str中传递。返回结果的StringBuffer对象。

下面的程序说明了replace( )方法：

```
// Demonstrate replace()
class replaceDemo {
  public static void main(String args[]) {
    StringBuffer sb = new StringBuffer("This is a test.");

    sb.replace(5, 7, "was");
    System.out.println("After replace: " + sb);
  }
}
```

输出如下所示：

```
After replace: This was a test.
```

13.10.12 substring( )

Java 2也增加了substring( )方法，它返回StringBuffer的一部分值。它具有如下的两种形式：

```
String substring(int startIndex)
String substring(int startIndex, int endIndex)
```

第一种形式返回调用StringBuffer对象中从startIndex下标开始直至结束的一个子字符串。第二种形式返回从startIndex开始到endIndex–1结束的子字符串。这些方法与前面在String中定义的那些方法具有相同的功能。

第 14 章　java.lang 研究

本章讨论那些由java.lang定义的类和接口。正如你所知道的那样，java.lang被自动导入所有的程序。它所包含的类和接口对所有实际的Java程序都是必要的。它是Java最广泛使用的包。

java.lang包括了下面这些类：

Boolean	Long	StrictMath (Java 2,1.3)
Byte	Math	String
Character	Number	StringBuffer
Class	Object	System
ClassLoader	Package (Java 2)	Thread
Compiler	Process	>ThreadGroup
Double	Runtime	ThreadLocal (Java 2)
Float	>RuntimePermission (Java 2)	Throwable
>InheritableThreadLocal (Java 2)	SecurityManager	Void
>Integer	>Short	>

另外还有两个由Character定义的类：Character.Subset和Character.UnicodeBlock，它们是在Java 2中新增加的。

java.lang也定义了如下的接口：

- Cloneable
- Comparable
- Runnable

其中Comparable接口是在Java 2中新增加的。

java.lang中的几个类包含了过时的方法，其中的大多数可以追溯到Java 1.0。在Java2中仍然提供了这些方法，用于支持逐渐减少的老程序，而这些方法在新程序中不被推荐使用。大多数的过时方法出现在Java 2之前，因此在这里不讨论这些方法。而在Java 2中出现的那些过时的方法将被提及。

Java 2也在java.lang包中增加了几个新的类和方法，这些新类和方法被说明如下。

14.1　简单类型包装器

在本书的第1部分，我们提到因为性能的原因，Java使用简单的类型，例如整型（int）和字符（char）。这些数据类型不是对象层次结构的组成部分。它们通过值传递给方法而

不能直接通过引用传递。而且，也没有办法使两种方法对整型（int）引用同一实例（same instance）。有时需要对这些简单的类型建立对象表达式。例如在第15章中讨论的仅仅处理对象的枚举类；如果要将简单类型存储到这些类中的一个，需要在类中包装简单类型。为了满足这种需要，Java提供了与每一个简单类型相应的类。本质上，这些类在类中包装（wrap）简单类型。因此，它们通常被称作类型包装器（wrappers）。

14.1.1 Number

抽象类Number定义了一个由包装数字类型字节型（byte），短整型（short），整型（int），长整型（long），浮点型（float）和双精度型（double）的类实现的超类。Number有返回上面不同数字格式的对象值的抽象方法。也就是，doubleValue( )方法返回双精度（double）值，floatValue( )方法返回浮点（float）值等。这些方法如下：

```
byte byteValue( )
double doubleValue( )
float floatValue( )
int intValue( )
long longValue( )
short shortValue( )
```

这些方法的返回值可以被舍入。

Number有6个具体的子类包含了6种数字类型的显式值：双精度型（Double），浮点型（Float），字节型（Byte），短整型（Short），整型（Integer）和长整型（Long）。

14.1.2 Double和Float

双精度（Double）和浮点（Float）分别是对类型double和类型float的浮点值的包装器。浮点（Float）构造函数如下所示：

```
Float(double num)
Float(float num)
Float(String str) 引发NumberFormatException异常
```

正如你所看到的，浮点（Float）对象可以由类型float或类型double的值创建。它们也能由浮点数的字符串表达式创建。

双精度（Double）构造函数如下：

```
Double(double num)
Double(String str) 引发NumberFormatException异常
```

双精度（Double）对象可以被双精度（double）值或包含了浮点值的字符串创建。

由浮点（Float）定义的方法在表14-1中列出。由双精度（Double）定义的方法在表14-2中列出。浮点（Float）和双精度（Double）都定义了下面的常数：

MAX_VALUE	最大正值
MIN_VALUE	最小正值
NaN	非数字

POSITIVE_INFINITY	正无穷
NEGATIVE_INFINITY	负无穷
TYPE	浮点（float）或双精度（double）的类（Class）对象

表 14-1 由 Float 定义的方法

方法	描述
byte byteValue（）	返回调用对象的值（字节型）
int compareTo（Float f）	将调用对象的数值与f中的数值进行比较，如果两者相等，返回0。如果调用对象的值小于f的值，则返回负值。如果调用对象的值大于f的值，则返回正值（在Java 2中新增加的）
int compareTo（object obj）	当obj是类Float中的对象时，该方法与compareTo（Float）的功能相同。否则，引发一个ClassCastException异常（在Java 2中新增加的）
double doubleValue（）	返回调用对象的值(双精度型)
boolean equals（Object FloatObj）	如果float调用对象与FloatObj相等，则返回true。否则返回false
static int float ToIntBits（float num）	返回与num相应的与IEEE兼容的单精度位模式
Float floatValue（）	返回调用对象的值（浮点型）
int hashCode（）	返回调用对象的散列值
static float intBitsToFloat（int num）	返回由num指定的，与IEEE兼容的单精度位模式的等价浮点（float）值
int intValue（）	返回整型（int）形式的调用对象值
boolean isInfinite（）	如果调用对象包含有无穷大值，则返回true。否则返回false
static boolean isInfinite（float num）	如果num指定了一个无穷大值，则返回true。否则返回false
boolean isNaN（）	如果调用对象中包含了非数字值，则返回true。否则返回false
static boolean isNaN（float num）	如果num指定了一个非数字值，则返回true。否则返回false
long longValue()	返回调用对象的值（长整型）
static float parseFloat（String str） throws NumberFormatException	以10为基数，返回包含在由str指定的字符串中的数字的等价浮点值（在Java 2中新增加的）
short shortValue（）	返回调用对象值（短整型）
String toString（）	返回调用对象的等价字符串形式
static String to String（float num）	返回由num指定的值的等价字符串
static Float valueOf（String str） throws NumberForamtException	返回包含了由str中的字符串指定的值的float对象

表 14-2 由 Double 定义的方法

方法	描述
byte byteValue（）	返回调用对象的值（字节型）
int compareTo（Double d）	将调用对象的值与d的数值进行比较。如果这两个值相等，则返回0。如果调用对象的数值小于d的数值，则返回负值。如果调用对象的数值大于d的数值，则返回正值（在Java 2中新增加的）
Int compareTo（Object obj）	如果obj属于类Double，其操作与compareTo（Double）相同。否则，引发一个ClassCastException异常（在Java 2中新增加的）
static long doubleToLongBits（double num）	返回与num相应的与IEEE兼容的双精度位模式
double doubleValue（）	返回调用对象的值（双精度）
boolean equals（Object DoubleObj）	如果double调用对象与DoubleObj相等，则返回true。否则，返回false
float floatValue（）	返回调用对象的值（浮点型）
int hashcode（）	返回调用对象的散列码
int intValue（）	返回调用对象的值（整型）
boolean isInfinite（）	如果调用对象包含了一个无穷大值，则返回true。否则，返回false
static boolean isInfinite（double num）	如果num指定了一个无穷大值，则返回true。否则，返回false
boolean is NaN（）	如果调用对象包含了一个非数字值，则返回true。否则，返回false
static boolean isNaN（double num）	如果num指定了一个非数字值，则返回true。否则，返回false
static double longBitsToDouble（long num）	返回由num指定的，与IEEE兼容的双精度位模式的双精度（double）等价值
long longValue（）	返回调用对象的值（长整型）
static double parseDouble（String str） throws NumberFormatException	以10为基数，返回包含在由str指定的字符串中的数字的等价双精度（double）形式（在Java 2中新增加的）
short shortValue（）	返回调用对象的值（短整型）
String toString（）	返回调用对象的等价字符串形式
Static String toString（double num）	返回由num指定的值的等价字符串形式
Static Double valueOf（String str） throws NumberFormatException	返回包含了由str中的字符串指定的值的double对象

在下面的例子中创建两个double对象——一个通过使用双精度（double）值实现，另一个通过传递一个可以被解析为双精度（double）的字符串来实现。

```
class DoubleDemo {
  public static void main(String args[]) {
    Double d1 = new Double(3.14159);
    Double d2 = new Double("314159E-5");

    System.out.println(d1 + " = " + d2 + " -> " + d1.equals(d2));
  }
}
```

正如从下面的输出中可以看到的那样，如同通过equals( )方法返回true，两种构造函数创建相同的双精度（double）实例。

```
3.14159 = 3.14159 -> true
```

理解 isInfinite( )和 isNaN( )

浮点（Float）和双精度（Double）提供了isInfinite( )和isNaN( )方法，这些方法会有助于操作两个特殊的双精度（double）和浮点（float）值，这些方法检验两个由IEEE浮点规范定义的独特的值：无穷和NaN（非具体数字）。当被检验的值为无穷大或无穷小值时，isInfinite( )方法返回true。当被检验值为非数字时，isNaN( )方法返回true。

在下面的例子中构造了两个Double对象；一个是无穷，另一个是非数字：

```
// Demonstrate isInfinite() and isNaN()
class InfNaN {
  public static void main(String args[]) {
    Double d1 = new Double(1/0.);
    Double d2 = new Double(0/0.);

    System.out.println(d1 + ": " + d1.isInfinite() + ", " + d1.isNaN());
    System.out.println(d2 + ": " + d2.isInfinite() + ", " + d2.isNaN());
  }
}
```

程序运行产生如下的输出：

```
Infinity: true, false
NaN: false, true
```

14.1.3　Byte, Short, Integer 和 Long

Byte，Short，Integer，和Long类分别是字节型（byte），短整型（short），整型（int）和长整型（long）整数类型的包装器。它们的构造函数如下：

```
Byte(byte num)
Byte(String str) 引发一个NumberFormatException异常
Short(short num)
Short(String str) 引发一个NumberFormatException异常
Integer(int num)
```

```
Integer(String str) 引发一个NumberFormatException异常
Long(long num)
Long(String str) 引发一个NumberFormatException异常
```

正如你能看到的，这些对象可由数值或含有有效整数值的字符串创建。

由这些类定义的方法列在表14-3到表14-6中。正如你能看到的，它们定义方法以便从字符串解析整数和将字符串转换为整数。为方便起见，这些方法提供的变量可以用来指定radix，也称为基数。通常二进制（binary）的基数是2，八进制（octal）的基数是8，十进制（decimal）的基数是10，而十六进制（hexadecimal）的基数为16。

表 14-3 由 Byte 定义的方法

方法	描述
byte byteValue（）	返回调用对象值（字节型）
int compareTo（Byte b）	将调用对象的数值与b的数值进行比较。如果这两个数值相等，则返回0。如果调用对象的数值小于b的数值，则返回负值。如果调用对象的数值大于b的数值，则返回正值（在Java 2中新增加的）
int compareTo（Object obj）	如果obj属于类Byte，其操作与compareTo（Byte）相同。否则，引发一个ClassCastException异常（在Java 2中新增加的）
static Byte decode（String str） throws NumberFormatException	返回一个包含了由str中的字符串指定的值的Byte对象
double doubleValue（）	返回调用对象值（双精度度型）
boolean equals（Object ByteObj）	如果Byte调用对象与ByteObj相等，则返回true。否则，返回false
float floatValue（）	返回调用对象值（浮点型）
int hashCode（）	返回调用对象的散列码
int intValue（）	返回调用对象值（整型）
long longValue（）	返回调用对象值（长整型）
static byte parseByte（String str） throws NumberFormatException	以10为基数，返回包含在由str指定的字符串中的数字的等价字节（byte）形式
static byte parseByte（String str，int radix） throws NumberFormatException	以指定的基数（radix）为底，返回包含在由str指定的字符串中的数字的等价字节
short shortValue（）	返回调用对象值（短整型）
String toString（）	返回一个包含了调用对象的等价十进制形式的字符串
static String toString（byte num）	返回一个包含了num的等价十进制形式的字符串
static Byte valueOf（String str） throws NumberFormatException	返回一个包含了由str中的字符串指定的值的Byte对象
static Byte valueOf（String str，int radix） throws NumberFormatException	以指定的基数（radix）为底，返回一个包含了由str中的字符串指定的值的Byte对象

表 14-4　由 Short 定义的方法

方法	描述
byte byteValue（）	返回调用对象值（字节型）
int compareTo（Short s）	将调用对象的数值和s的数值进行比较。如果这两个值相等，则返回0。如果调用对象的数值小于s的数值，则返回负值 如果调用对象的数值大于s的数值，则返回正值（在Java 2中新增加的）
int compareTo（Object obj）	如果obj属于类Short，其操作与compareTo（Short）相同。否则，引发一个ClassCastException异常（在Java 2中新增加的）
static Short decode（String str） throws NumberFormatException	返回一个包含了由str中的字符串指定值的Short对象
double doubleValue（）	返回调用对象值（双精度型）
boolean equals（Object ShortObj）	如果整型（Interger）调用对象与ShortObj相等，则返回true。否则，返回false
float floatValue（）	返回调用对象值（浮点值）
int hashCode（）	返回调用对象的散列码
int intValue（）	返回调用对象值（整型）
long longValue（）	返回调用对象值（长整型）
static short parseShort（String str） throws NumberFormatException	以10为基数，返回包含在由str指定的字符串中的数字的等价短整型（Short）数
static short parseShort（String str，int radix） throws NumberFormatException	以指定的基数（radix）为底，返回包含在由str指定的字符串中的数字的等价短整型（Short）数
short shortValue（）	返回调用对象值（短整型）
String toString（）	返回一个包含了调用对象的等价十进制形式的字符串
static String to String（short num）	返回一个包含了num的等价十进制形式的字符串
static Shortvalue Of（Stringstr）throws NumberFormatException	以10为基数，返回一个包含了由str中的字符串指定的值的Short对象
static Short valueOf（String str，int radix） throws NumberFormatException	以指定的基数（radix）为底，返回一个包含了由str中的字符串指定的值的Short对象

表 14-5　由 Integer 定义的方法

方法	描述
byte byteValue（）	返回调用对象值（字节型）
int compareTo（Integer i）	将调用对象的数值与i的数值进行比较。如果这两个值相等，则返回0。如果调用对象的数值小于i的数值，则返回负值。如果调用对象的数值大于i的数值，则返回正值（在Java 2中新增加的）

续表

方法	描述
int compareTo（Object obj）	如果obj属于类Integer，其操作与compareTo（Integer）相同。否则，引发一个ClassCastException异常（在Java 2中新增加的）
static Integer decode（String str） throws NumberFormatException	返回一个包含了由str中的字符串指定值的Integer对象
double doubleValue（）	返回调用对象值（双精度型）
boolean equals（Object IntegerObj）	如果调用Interger对象与IntegerObj相等，则返回true。否则，返回false
float floatValue（）static Integer getInteger （String propertyName）	返回调用对象值（浮点型） 返回与由propertyname指定的环境属性相关联的值，调用失败返回null
static Integer getInteger （String propertyName，int default）	返回与由propertyname指定的环境属性相关联的值，调用失败返回default值
static Integer getInteger （String propertyName，Integer default）	返回与由propertyname指定的环境属性相关联的值，调用失败返回default值
int hashCode（）	返回调用对象的散列码
int intValue（）	返回调用对象值（整型）
long longValue（）	返回调用对象值（长整型）
static int parseInt（String str） throws NumberFormatException	以10为基数，返回包含在由str指定的字符串中的数字的等价整数（integer）值
static int parseInt（String str，int radix） throws NumberFormatException	以指定的基数（radix）为底，返回包含在由str指定的字符串中的数字的等价整数值
short shortValue（）	返回调用对象值（短整型）
static String toBinaryString（int num）	返回一个包含了num的等价二进制形式的字符串
static String toHexString（int num）	返回一个包含了num的等价十六进制形式的字符串
static String toOctalString（int num）	返回一个包含了num的等价八进制形式的字符串
String toString（）	返回一个包含了调用对象的等价十进制形式的字符串
static String toString（int num）	返回一个包含了num的等价十进制形式的字符串
static String toString（int num，int radix）	以指定的基数（radix）为底，返回一个包含了num的等价十进制形式的字符串
static Integer valueOf（String str） throws NumberFormatException	返回一个包含了由str中的字符串指定的值的Integer对象
static Integer valueOf（String str，int radix） throws NumberFormatException	以指定的基数（radix）为底，返回一个包含了由str中的字符串指定的值的Integer对象

表 14-6　由 Long 定义的方法

方法	描述
byte byteValue（）	返回调用对象值（字节型）
int compareTo（Long l）	将调用对象的数值和l的数值进行比较，如果这两个值相等，则返回0。如果调用对象的数值小于l的数值，则返回负值。如果调用对象的数值大于l的数值，则返回正值（在Java 2中新增加的）
int compareTo（Object obj）	如果obj属于类long，其操作与compareTo（Long）相同。否则，引发一个ClassCastException异常（在Java 2中新增加的）
static Long decode（String str） throws NumberFormatException	返回一个包含了由str中的字符串指定的值的Long对象
double doubleValue（）	返回调用对象值（双精度型）
boolean equals（Object LongObj）	如果调用Long对象与LongObj相等，则返回true。否则，返回false
float floatValue（）	返回调用对象值（浮点型）
static Long getLong（String propertyname）	返回与由propertyname指定的环境属性相关联的值，调用失败则返回null
static Long getLong（String propertyname， long default）	返回与由propertyname指定的环境属性相关联的值，调用失败则返回default的值
static long getLong（String propertyname， Long default）	返回与由propertyname指定的环境属性相关联的值，调用失败则返回default的值
int hashcode（）	返回调用对象的散列码
int intValue（）	返回调用对象值（整型）
long longValue（）	返回调用对象值（长整型）
static long parseLong（String str） throws NumberFormatException	以10为基数，返回包含在由str指定的字符串中的数字的等价长整型（Long）数
static long parseLong（String str，int radix） throws NumberFormatException	以指定的基数（radix）为底，返回包含在由str指定的字符串中的数字的等价长整型（Long）数
short shortValue（）	返回调用对象值（短整型）
static String toBinaryString（long num）	返回一个包含了num的等价二进制形式的字符串
static String toHexString（long num）	返回一个包含了num的等价十六进制形式的字符串
static String toOctalString（long num）	返回一个包含了num的等价八进制形式的字符串
String toString（）	返回一个包含了调用对象的等价十进制形式的字符串
static String toString（long num）	返回一个包含了num的等价十进制形式的字符串
static String toString（long num，int radix）	以指定的基数（radix）为底，返回一个包含了num的等价十进制形式的字符串
static Long valueOf（String str） throws NumberFormatException	返回一个包含了由str中的字符串指定的值的Long对象

续表

方法	描述
static Long valueOf（String str，int radix） throws NumberFormatException	以指定的基数（radix）为底，返回一个包含了由str中的字符串指定的值的Long对象

定义下面的常数：

MIN_VALUE	最小值
MAX_VALUE	最大值
TYPE	字节（Byte），短整型（short），整型（int）或长整型（long）的类（Class）对象

数字和字符串的转换

程序设计中一个最常见的任务是将一个数字的字符串表达式转换成内部的二进制格式。幸运的是Java提供了一个方便的方法去完成这项任务。Byte，Short，Integer和Long类分别提供了parseByte( )，parseShort( )，parseInt( )和parseLong( )方法。这些方法返回与调用它们的数值字符串相应的字节（byte），短整型（short），整型（int）和长整型（long）值（在Float和Double类中也有相似的方法）。

下面的程序说明了parseInt( )方法。该程序完成对用户输入的一系列整数的求和。在程序中通过使用readLine( )方法读取整数，使用parseInt( )方法将这些字符串转换成与之相应的整型（int）值。

```
/* This program sums a list of numbers entered
   by the user.  It converts the string representation
   of each number into an int using parseInt().
*/

import java.io.*;

class ParseDemo {
  public static void main(String args[])
    throws IOException
  {
    // create a BufferedReader using System.in
    BufferedReader br = new
      BufferedReader(new InputStreamReader(System.in));
    String str;
    int i;
    int sum=0;

    System.out.println("Enter numbers, 0 to quit.");
    do {
      str = br.readLine();
      try {
        i = Integer.parseInt(str);
      } catch(NumberFormatException e) {
        System.out.println("Invalid format");
        i = 0;
      }
```

```
      sum += i;
      System.out.println("Current sum is: " + sum);
    } while(i != 0);
  }
}
```

为了将一个整数转换为一个十进制的字符串，可以使用在Byte，Short，Integer或Long类中定义的toString( )方法。Integer和Long类还同时提供了toBinaryString( )，toHexString( )和toOctalString( )方法，可以分别将一个值转换成二进制，十六进制和八进制字符串。

下面的程序说明了向二进制，十六进制和八进制的转换：

```
/* Convert an integer into binary, hexadecimal,
   and octal.
*/

class StringConversions {
  public static void main(String args[]) {
    int num = 19648;

    System.out.println(num + " in binary: " +
                       Integer.toBinaryString(num));

    System.out.println(num + " in octal: " +
                       Integer.toOctalString(num));

    System.out.println(num + " in hexadecimal: " +
                       Integer.toHexString(num));
  }
}
```

程序的输出结果如下所示：

```
19648 in binary: 100110011000000
19648 in octal: 46300
19648 in hexadecimal: 4cc0
```

14.1.4　Character

字符（Character）是围绕字符型（char）的一个简单的包装器。字符（Character）的构造函数如下：

```
Character(char ch)
```

这里ch指定了被创建的字符（Character）对象所包装的字符。

调用如下的charValue( )方法可以获得包含在字符（Character）对象中的字符型（char）值。

```
char charValue( )
```

调用的结果返回字符。

字符（Character）类定义了几个常数，包括下面这些：

MAX_RADIX	最大基数
MIN_RADIX	最小基数
MAX_VALUE	最大字符值
MIN_VALUE	最小字符值
TYPE	字符型（char）的类（Class）对象

字符（Character）包括了几个静态方法，这些方法完成将字符分类并改变它们的大小写。这些方法在表14-7中列出。下面的例子说明了这些方法。

```
// Demonstrate several Is... methods.

class IsDemo {
  public static void main(String args[]) {
    char a[] = {'a', 'b', '5', '?', 'A', ' '};

    for(int i=0; i<a.length; i++) {
      if(Character.isDigit(a[i]))
        System.out.println(a[i] + " is a digit.");
      if(Character.isLetter(a[i]))
        System.out.println(a[i] + " is a letter.");
      if(Character.isWhitespace(a[i]))
        System.out.println(a[i] + " is whitespace.");
      if(Character.isUpperCase(a[i]))
        System.out.println(a[i] + " is uppercase.");
      if(Character.isLowerCase(a[i]))
        System.out.println(a[i] + " is lowercase.");
    }
  }
}
```

程序的输出结果如下所示：

```
a is a letter.
a is lowercase.
b is a letter.
b is lowercase.
5 is a digit.
A is a letter.
A is uppercase.
  is whitespace.
```

表 14-7 各种字符（Character）方法

方法	描述
static boolean isDefined（char ch）	如果ch是由Unicode定义的，则返回true，否则，返回false
static boolean isDigit（char ch）	如果ch是一个数字，则返回true，否则，返回false
static boolean isIdentifierIgnorable（char ch）	如果在一个标识符中ch应被忽略，则返回true，否则，返回false
static boolean isISOControl（char ch）	如果ch是一个ISO控制字符，则返回true，否则，返回false

续表

方法	描述
static boolean isJavaIdentifierPart（char ch）	如果ch被做为Java标识符的一部分（除了第一个字符），则返回true。否则，返回false
static boolean isJavaIdentifierStart（char ch）	如果ch被做为Java标识符的首字符，则返回true。否则返回false
static boolean isLetter（char ch）	如果ch是一个字母，则返回true。否则返回false
static boolean isLetterOrDigit（char ch）	如果ch是一个字母或一个数字，则返回true。否则返回false
static boolean isLowerCase（char ch）	当ch是小写字母时，返回true。否则返回false
static boolean isSpaceChar（char ch）	如果ch是Unicode编码的空格字符，则返回true。否则返回false
static boolean isTitleCase（char ch）	如果ch是Unicode编码的标题字符，则返回true。否则返回false
static boolean isUnicodeIdentifierPart（char ch）	如果ch被做为Unicode编码标识符的一部分（除了第一个字符），则返回true。否则，返回false
static boolean isUnicodeIdentifierStart（char ch）	如果ch被做为一个Unicode标识符的首字符，则返回true。否则返回false
static boolean isUpperCase（char ch）	如果ch是一个大写字母，则返回true。否则返回false
static boolean isWhitespace（char ch）	如果ch是一个空白符，则返回true。否则，返回false
static char toLowerCase（char ch）	返回ch的小写等价形式
static char toTitleCase（char ch）	返回ch的标题等价形式
static char toUpperCase（char ch）	返回ch的大写等价形式

字符（Character）定义了如下形式的forDigit( )和digit( )方法：

```
static char forDigit(int num, int radix)
static int digit(char digit, int radix)
```

forDigit( )方法返回与num的值关联的数字字符。而转换的基数由radix指定。digit( )方法按照给定的基数，返回与指定字符（该字符可能是一个数字）相关联的整数值。

由Character类定义的另一个方法是compareTo( )，该方法具有如下的两种形式：

```
int compareTo(Character c)
int compareTo(Object obj)
```

第一种形式当调用对象与c具有相同值时返回0。当调用对象具有比c小的值时返回一个负值。否则它将返回一个正值。在第二种形式中，当obj是对Character类的一个引用时，其功能与第一种形式一样。否则它将引发一个ClassCastException异常。这些方法是在Java 2中新增加的。

Character类还定义了equals( )和hashCode( )方法。

另两个与字符有关的类是Character.Subset和Character.UnicodeBlock，其中

Character.Subset类用于描述Unicode编码的一个子集，而Character.UnicodeBlock类中包含了Unicode 2.0编码的字符块。

14.1.5 Boolean

Boolean是一个围绕布尔（boolean）值的非常细小的包装器，主要用在通过引用传递布尔（boolean）变量的场合。它包含了常数TRUE和FALSE，这些常数定义了布尔（Boolean）对象的真与假。Boolean也定义了TYPE域，它是boolean的Class对象。在Boolean中定义了如下的构造函数：

```
Boolean(boolean boolValue)
Boolean(String boolString)
```

在第一种形式中，boolValue要么是true，要么是false。在第二种形式中，如果在boolString中包含了字符串“true”（无论是大写形式还是小写形式），则新的布尔（Boolean）对象将为真，否则为假。

Boolean定义了如表14-8中列出的方法。

表 14-8 由 Boolean 定义的方法

方法	描述
boolean booleanValue( )	返回布尔（boolean）等价形式
boolean equals(Object boolObj)	如果调用对象与boolObj相等，则返回true。否则返回false
static boolean getBoolean(String propertyName)	如果由propertyName指定的系统属性为true，则返回true。否则返回false
int hashCode( )	返回调用对象的散列码
String toString( )	返回调用对象的字符串等价形式
static Boolean valueOf(String boolString)	如果在boolString中包含了“true”（以大写或小写形式），则返回true。否则返回false

14.2 Void

Void类有一个TYPE域，该域保存对类型void的Class对象的引用。这样做将不创建类的实例。

14.3 Process

抽象类Process封装了一个进程（process）——也就是说一个正在执行的程序。它主要被当作由Runtime类中的exec( )方法所创建的对象的类型的超类。Runtime类将在下面介绍。在抽象类Process中，包含了如下表14-9中列出的抽象方法。

表 14-9　由 Process 定义的抽象方法

方法	描述
void destroy( )	中断进程
int exitValue( )	返回一个从子进程获得的退出码
InputStream getErrorStream( )	返回一个从进程的err输出流中读输入的输入流
InputStream getInputStream( )	返回一个从进程的out输出流中读输入的输入流
OutputStream getOutputStream( )	返回一个从进程的in输入流中写输出的输出流
Int waitFor( ) throws InterruptedException	返回由进程返回的退出码。这个方法直到调用它的进程中止，才会返回

14.4　Runtime

Runtime类封装了运行时环境。一般不实例化一个Runtime对象。但是可以通过调用静态方法Runtime.getRuntime( )而获得对当前Runtime对象的引用。一旦获得了对当前对象的引用，就可以调用几个控制Java虚拟机的状态和行为的方法。小应用程序（Applets）和其他不可信赖的编码由于没有引起一个安全异常（SecurityException）而不能调用任何的Runtime方法。

表14-10给出了由Runtime定义的方法。Java 2中不赞成使用方法runFinalizersOnExit( )。这种方法是在Java 1.1中增加的，但被认为是一种不稳定的方法。

表 14-10　由 Runtime 定义的常用方法

方法	描述
void addShutdownHook(Thread thrd)	当Java虚拟机终止时，寄存器thrd作为线程而运行
Process exec(String progName) throws IOException	将由progName指定的程序作为独立的进程来执行。返回描述新进程的类型Process的对象
Process exec(String progName, String environment[ ]) throws IOException	将由progName指定的程序作为独立的进程来执行。该独立进程的环境由environment指定。返回描述新进程的类型Process的对象
Process exec(String comLineArray[ ], String environment[ ]) throws IOException	将由comLineArray中的字符串指定的命令行作为独立的进程来执行。运行环境由environment指定。返回描述新进程的类型Process的对象
void exit(int exitCode)	暂停执行并且向父进程返回exitCode的值，按照约定，0表示正常中止，所有的其他值表示有某种形式的错误
long freeMemory( )	返回Java运行系统可以利用的空闲内存的大概字节数
void gc( )	初始化垃圾回收站
static Runtime getRuntime( )	返回当前的Runtime对象

续表

方法	描述
void halt(int code)	立即终止Java虚拟机，不执行任何的终止线程和善后处理程序。code的值返回给调用进程（在Java 2的1.3版中新增加的）
void load(String libraryFileName)	载入库中文件由libraryFileName指定的动态库，必须指定它的完全路径
void loadLibrary(String libraryName)	载入库名为libraryName的动态库
boolean removeShutdownHook(Thread thrd)	当Java虚拟机中止，从线程列表中移出thrd的运行。如果成功，也就是说如果线程被移出，则返回true（在Java 2的1.3版中新增加的）
void runFinalization( )	调用未用的但还不是回收站中对象的finalize()方法
long totalMemory( )	返回程序可以利用的内存的总字节数
void traceInstructions(boolean traceOn)	根据traceOn的值，打开或关闭指令跟踪。如果traceOn值为true，跟踪被显示。如果traceOn值为false，跟踪被关闭
void traceMethodCalls(boolean traceOn)	根据traceOn的值，打开或关闭调用跟踪的方法。如果traceOn的值为true，跟踪被显示。如果traceOn的值为false，跟踪被关闭

让我们来看一看Runtime类的两个最普遍的用法：内存管理和执行附加进程。

14.4.1 内存管理

尽管Java提供了自动垃圾回收，有时也想知道对象堆的大小以及它还剩下多少。可以利用这些信息检验你的代码的效率，或估计对某些类型，有多少对象可以被实例化。为了获得这些值，可以使用totalMemory( )和freeMemory( )方法。

正如我们在第1部分提及的，Java的垃圾回收器周期性地运行将不再使用的对象放入回收站。然而有时想在收集器的下一个指定循环之前收集被丢弃的对象。可以通过调用gc( )方法按照要求运行垃圾回收器。一个好的尝试是调用gc( )方法，然后再调用freeMemory( )方法以获得内存使用的底线。接着执行你的程序，并再一次调用freeMemory( )方法看分配了多少内存。下面的例子说明了这个思想。

```
// Demonstrate totalMemory(), freeMemory() and gc().

class MemoryDemo {
  public static void main(String args[]) {
    Runtime r = Runtime.getRuntime();
    long mem1, mem2;
    Integer someints[] = new Integer[1000];

    System.out.println("Total memory is: " +
                       r.totalMemory());

    mem1 = r.freeMemory();
    System.out.println("Initial free memory: " + mem1);
```

```
    r.gc();
    mem1 = r.freeMemory();
    System.out.println("Free memory after garbage collection: "
                       + mem1);

    for(int i=0; i<1000; i++)
      someints[i] = new Integer(i); // allocate integers

    mem2 = r.freeMemory();
    System.out.println("Free memory after allocation: "
                       + mem2);
    System.out.println("Memory used by allocation: "
                       + (mem1-mem2));

    // discard Integers
    for(int i=0; i<1000; i++) someints[i] = null;

    r.gc(); // request garbage collection

    mem2 = r.freeMemory();
    System.out.println("Free memory after collecting" +
                       " discarded Integers: " + mem2);

  }
}
```

这个例子的一个输出样本如下（当然，你的实际运行结果可能会与之不同）：

```
Total memory is: 1048568
Initial free memory: 751392
Free memory after garbage collection: 841424
Free memory after allocation: 824000
Memory used by allocation: 17424
Free memory after collecting discarded Integers: 842640
```

14.4.2　执行其他的程序

在可靠的环境中，可以在你的多任务操作系统中使用Java去执行其他特别繁重的进程（也即程序）。exec( )方法的几种形式允许命名想运行的程序以及它们的输入参数。exec( )方法返回一个Process对象，这个对象可以被用来控制你的Java程序如何与这个正在运行的新进程相互作用。因为Java可以运行在多种平台和多种操作系统的情况下，exec( )方法本质上是依赖于环境的。

下面的例子使用exec( )方法装入Window的简单文本编辑器——notepad。显而易见，这个例子必须在Windows操作系统下运行。

```
// Demonstrate exec().
class ExecDemo {
  public static void main(String args[]) {
    Runtime r = Runtime.getRuntime();
    Process p = null;

    try {
      p = r.exec("notepad");
```

```
    } catch (Exception e) {
      System.out.println("Error executing notepad.");
    }
  }
}
```

exec( )方法有几个形式可用，而在本例子中展示的是最常用的一种。在新程序开始运行之后，由exec( )方法返回的Process对象可以被Process方法使用。可以使用destroy( )方法杀死子进程。waitFor( )方法暂停你的程序直至子进程结束。当子进程结束后，exitValue( )方法返回子进程返回的值。如果没有问题发生，它通常返回0。下面是前面关于exec( )方法例子的改进版本。例子被修改为等待直至正在运行的进程退出：

```
// Wait until notepad is terminated.
class ExecDemoFini {
  public static void main(String args[]) {
    Runtime r = Runtime.getRuntime();
    Process p = null;

    try {
      p = r.exec("notepad");
      p.waitFor();
    } catch (Exception e) {
      System.out.println("Error executing notepad.");
    }
    System.out.println("Notepad returned " + p.exitValue());
  }
}
```

当子进程正在运行时，可以从它的标准输入输出进行读和写。getOutputStream( )方法和getInputStream( )方法返回子进程的标准输入（in）和输出（out）的句柄（关于I/O将在第17章详细讨论）。

14.5 System

System类保存静态方法和变量的集合。标准的输入，输出和Java运行时错误输出存储在变量in，out和err中。由System类定义的方法列在表14-11中。注意当所做操作是安全方式所不允许的时，许多方法引发一个安全异常（SecurityException）。应当注意的另一点是：Java 2不赞成使用runFinalizersOnExit( )方法。该方法是在Java 1.1中增加的，同时也被证明是不可靠的。

让我们看一看System类的一些普遍用法。

表 14-11　由 Sysem 定义的方法

方法	描述
static void arraycopy(Object source, int sourceStart, Object target, int targetStart, int size)	复制数组。被复制的数组由source传递，而source中开始复制数组时的下标由sourceStart传递。接收复制的数组由target传递。而target中开始复制数组时的下标由targetStart传递。Size是被复制的元素的个数
static long currentTimeMillis( )	返回自1970年1月1日午夜至今的时间，时间单位为毫秒。
static void exit(int exitCode)	暂停执行，返回exitCode值给父进程（通常为操作系统）。按照约定，0表示正常退出，所有其他的值代表某种形式的错误
static void gc( )	初始化垃圾回收
static Properties getProperties( )	返回与Java运行系统有关的属性类（Properties class）将在第15章中介绍）
static String getProperty(String which)	返回与which有关的属性。如果期望的属性没有被发现，返回一个空对象（null object）
static String getProperty(String which, String default)	返回一个与which有关的属性。如果期望的属性没有被发现，则返回default
static SecurityManager getSecurityManager( )	返回当前的安全管理程序，如果没有安装安全管理程序，则返回一个空对象（null object）
static native int identityHashCode(Object obj)	返回obj的特征散列码
static void load(String libraryFileName)	载入其文件由libraryFileName指定的动态库，必须指定其完全路径
static void loadLibrary(String libraryName)	载入其库名为libraryName的动态库
static String mapLibraryName(String lib)	对应名为lib的库，返回一个指定平台的名字（在Java 2中新增加的）
static void runFinalization( )	启动调用不用的但还不是回收站中的对象的finalize( )方法。
static void setErr(PrintStream eStream)	设置标准的错误（err）流为iStream
static void setIn(InputStream iStream)	设置标准的输入（in）流为oStream
static void setOut(PrintStream oStream)	设置标准的输出（out）流eStream
static void setProperties(Properties sysProperties)	设置由sysProperties指定的当前系统属性
Static String setProperty(String which, String v)	将v值赋给名为which的属性（在Java 2中新增加的）
static void setSecurityManager (SecurityManager secMan)	设置由secMan指定的安全管理程序

14.5.1 使用currentTimeMillis( )记录程序执行的时间

可以发现System类的一个特别有意义的用法是利用currentTimeMillis( )方法来记录你的程序的不同部分的执行时间。currentTimeMillis( )方法返回自从1970年1月1号午夜起到现在的时间，时间单位是毫秒。如果要记录你的程序中一段有问题程序的运行时间可以在这段程序开始之前存储当前时间，在该段程序结束之际再次调用currentTimeMillis( )方法。执行该段程序所花费的时间为其结束时刻的时间值减去其开始时刻的时间值。下面的程序说明了这一点：

```
// Timing program execution.

class Elapsed {
  public static void main(String args[]) {
    long start, end;

    System.out.println("Timing a for loop from 0 to 1,000,000");

    // time a for loop from 0 to 1,000,000
    start = System.currentTimeMillis(); // get starting time
    for(int i=0; i < 1000000; i++) ;
    end = System.currentTimeMillis(); // get ending time

    System.out.println("Elapsed time: " + (end-start));
  }
}
```

这里是程序运行的一个输出样本（记住你的程序的运行结果可能与此不同）：

```
Timing a for loop from 0 to 1,000,000
Elapsed time: 10
```

14.5.2 使用arraycopy( )

使用arraycopy( )方法可以将一个任意类型的数组快速地从一个地方复制到另一个地方。这比使用Java中编写的循环要快的多。下面是一个用arraycopy( )方法复制两个数组的例子。首先，将数组a复制给数组b，接下来，数组a中的所有元素向后移一位，然后数组b中元素向前移一位。

```
// Using arraycopy().

class ACDemo {
  static byte a[] = { 65, 66, 67, 68, 69, 70, 71, 72, 73, 74 };
  static byte b[] = { 77, 77, 77, 77, 77, 77, 77, 77, 77, 77 };

  public static void main(String args[]) {
    System.out.println("a = " + new String(a));
    System.out.println("b = " + new String(b));
    System.arraycopy(a, 0, b, 0, a.length);
    System.out.println("a = " + new String(a));
    System.out.println("b = " + new String(b));
    System.arraycopy(a, 0, a, 1, a.length - 1);
```

```
    System.arraycopy(b, 1, b, 0, b.length - 1);
    System.out.println("a = " + new String(a));
    System.out.println("b = " + new String(b));
  }
}
```

正如从下面的输出中看到的那样，可以使用相同的源和目的在任一方向进行复制：

```
a = ABCDEFGHIJ
b = MMMMMMMMM
a = ABCDEFGHIJ
b = ABCDEFGHIJ
a = AABCDEFGHI
b = BCDEFGHIJJ
```

14.5.3　环境属性

下面的属性在Java 2的所有环境中可以使用：

file.separator	java.vendor.url	os.arch
java.class.path	java.version	os.name
java.class.version	java.vm.name	os.version
java.ext.dirs	java.vm.specification.name	Path.separator
java.home	java.vm.specification.vendor	User.dir
java.specification.name	java.vm.specification.version	User.home
java.specification.vendor	java.vm.vendor	User.name
java.specification.version	java.vm.version	
java.vendor	line.separator	

可以通过调用System.getProperty( )方法来获得不同环境变量的值。例如下面的程序显示当前用户目录的路径：

```
class ShowUserDir {
  public static void main(String args[]) {
    System.out.println(System.getProperty("user.dir"));
  }
}
```

14.6　Object

正如我们在第1部分所提及的，Object类是其他所有类的一个超类。表14-12给出了Object类中定义的方法，这些方法对于每一个对象都是可用的。

表 14-12 由 Object 定义的方法

方法	描述
Object clone( ) Throws CloneNotSupportedException	创建一个与调用对象一样的新对象
Boolean equals(Object object)	如果调用对象等价于object返回true
void finalize( ) throws Throwable	默认的finalize( )方法。常常被子类重载
final Class getClass( )	获得描述调用对象的Class对象
int hashCode( )	返回与调用对象关联的散列码
final void notify( )	恢复等待调用对象的线程的执行
final void notifyAll( )	恢复等待调用对象的所有线程的执行
String toString( )	返回描述对象的一个字符串
final void wait( ) throws InterruptedException	等待另一个执行的线程
final void wait(long milliseconds) throws InterruptedException	等待直至指定毫秒数的另一个执行的线程
final void wait(long milliseconds, int nanoseconds) throws InterruptedException	等待直至指定毫秒加毫微秒数的另一个执行的线程

14.7 使用clone( )和Cloneable接口

由Object类定义的绝大部分方法在本书其他部分讨论。而一个特别值得关注的方法是clone( )。clone( )方法创建调用它的对象的一个复制副本。只有那些实现Cloneable接口的类能被复制。

Cloneable接口没有定义成员。它通常用于指明被创建的一个允许对对象进行位复制（也就是对象副本）的类。如果试图用一个不支持Cloneable接口的类调用clone( )方法，将引发一个CloneNotSupportedException异常。当一个副本被创建时，并没有调用被复制对象的构造函数。副本仅仅是原对象的一个简单精确的拷贝。

复制是一个具有潜在危险的操作，因为它可能引起不是你所期望的副作用。例如，假如被复制的对象包含了一个称为obRef的引用变量，当副本创建时，副本中的obRef如同原对象中的obRef一样引用相同的对象。如果副本改变了被obRef引用的对象的内容，那么对应的原对象也将被改变。这里是另一个例子。如果一个对象打开一个I/O流并被复制，两个对象将可操作相同的流。而且，如果其中一个对象关闭了流，而另一个对象仍试图对I/O流进行写操作的话，将导致错误。

由于复制可能引起问题，因此在Object内，clone( )方法被说明为protected。这就意味着它必须或者被由实现Cloneable的类所定义的方法调用，或者必须被那些类显式重载以便它

是公共的。让我们看关于下面每一种方法的例子。

下面的程序实现Cloneable接口并定义cloneTest( )方法，该方法在Object中调用clone( )方法：

```
// Demonstrate the clone() method.

class TestClone implements Cloneable {
  int a;
  double b;

  // This method calls Object's clone().
  TestClone cloneTest() {
    try {
      // call clone in Object.
      return (TestClone) super.clone();
    } catch(CloneNotSupportedException e) {
      System.out.println("Cloning not allowed.");
      return this;
    }
  }
}

class CloneDemo {
  public static void main(String args[]) {
    TestClone x1 = new TestClone();
    TestClone x2;

    x1.a = 10;
    x1.b = 20.98;

    x2 = x1.cloneTest(); // clone x1

    System.out.println("x1: " + x1.a + " " + x1.b);
    System.out.println("x2: " + x2.a + " " + x2.b);
  }
}
```

这里，方法cloneTest( )在Object中调用clone( )方法并且返回结果。注意由clone( )方法返回的对象必须被强制转换成它的适当类型（TestClone）。

下面的例子重载clone( )方法以便它能被其类外的程序所调用。为了完成这项功能，它的存取说明符必须是public，如下所示：

```
// Override the clone() method.

class TestClone implements Cloneable {
  int a;
  double b;

  // clone() is now overridden and is public.
  public Object clone() {
    try {
      // call clone in Object.
      return super.clone();
```

```
      } catch(CloneNotSupportedException e) {
        System.out.println("Cloning not allowed.");
        return this;
      }
    }
  }

  class CloneDemo2 {
    public static void main(String args[]) {
      TestClone x1 = new TestClone();
      TestClone x2;

      x1.a = 10;
      x1.b = 20.98;

      // here, clone() is called directly.
      x2 = (TestClone) x1.clone();

      System.out.println("x1: " + x1.a + " " + x1.b);
      System.out.println("x2: " + x2.a + " " + x2.b);
    }
  }
```

由复制带来的副作用最初一般是比较难发现的。通常很容易想到的是类在复制时是很安全的，而实际却不是这样。一般在没有一个必须的原因的情况下，对任何类都不应该执行Cloneable。

14.8 Class

Class封装对象或接口运行时的状态。当类被加载时，类型Class的对象被自动创建。不能显式说明一个类（Class）对象。一般地，通过调用由Object定义的getClass( )方法来获得一个类（Class）对象。由Class定义的一些最常用的方法列在表14-13中。

表 14-13 由 Class 定义的一些方法

方法	描述
static Class forName(String name) throws ClassNotFoundException	返回一个给定全名的Class对象
static Class forName(String name, Boolean how, ClassLoader ldr) throws ClassNotFoundException	返回一个给定全名的Calss对象。对象由ldr指定的加载程序加载。如果how为true，对象被初始化，否则它将不被初始化（在Java 2中新增加的）
Class[ ] getClasses( )	对每一个公共类和接口，返回一个类（Class）对象。这些公共类和接口是调用对象的成员
ClassLoader getClassLoader( )	返回一个加载类或接口的ClassLoader对象，类或接口用于实例化调用对象

续表

方法	描述
Constructor[ ] getConstructors( ) throws SecurityException	对这个类的所有的公共构造函数，返回一个Constructor对象
Constructor[ ] getDeclaredConstructors( ) throws SecurityException	对由这个类所声明的所有构造函数，返回一个Constructor对象
Field[ ] getDeclaredFields( ) throws SecurityException	对由这个类所声明的所有域，返回一个Field对象
Method[ ] getDeclaredMethods( ) throws SecurityException	对由这个类或接口所声明的所有方法，返回一个Method对象
Field[ ] getFields( ) throws SecurityException	对于这个类的所有公共域，返回一个Field对象
Class[ ] getInterfaces( )	当调用对象时，这个方法返回一个由该对象的类类型实现的接口数组。当调用接口时，这个方法返回一个由该接口扩展的接口数组
Method[ ] getMethods( ) throws SecurityException	对这个类中的所有公共方法，返回一个Method对象
String getName( )	返回调用对象的类或接口的全名
ProtectionDomain getProtectionDomain( )	返回与调用对象有关的保护范围（在Java 2中新增加的）
Class getSuperclass( )	返回调用对象的超类。如果调用对象是类型Object的，则返回值为空（null）
Boolean isInterface( )	如果调用对象是一个接口，则返回true。否则返回false
Object newInstance( ) throws IllegalAccessException, InstantiationException	创建一个与调用对象类型相同的新的实例（即一个新对象）。这相当于对类的默认构造函数使用new。返回新对象
String toString( )	返回调用对象或接口的字符串表达式

由Class定义的方法经常用在需要知道对象的运行时类型信息的场合。如同表14-13中所说明的那样，由Class提供的方法确定关于特定的类的附加信息。例如它的公共构造函数，域以及方法。这对于本书后面将要讨论的Java Beans函数是很重要的。

下面的程序说明了getClass( )（从Object继承的）和getSuperclass( )方法（从Class继承的）：

```
// Demonstrate Run-Time Type Information.

class X {
  int a;
  float b;
}

class Y extends X {
  double c;
}
```

```
class RTTI {
  public static void main(String args[]) {
    X x = new X();
    Y y = new Y();
    Class clObj;

    clObj = x.getClass(); // get Class reference
    System.out.println("x is object of type: " +
                       clObj.getName());

    clObj = y.getClass(); // get Class reference
    System.out.println("y is object of type: " +
                       clObj.getName());
    clObj = clObj.getSuperclass();
    System.out.println("y's superclass is " +
                       clObj.getName());
  }
}
```

这个程序的输出如下所示：

```
x is object of type: X
y is object of type: Y
y’s superclass is X
```

14.9 ClassLoader

抽象类ClassLoader规定了类是如何加载的。应用程序可以创建扩展ClassLoader的子类，实现它的方法。这样做允许使用不同于通常由Java运行时系统加载的另一些方法来加载类。由ClassLoader定义的一些方法列在表14-14中。

表 14-14 由 CalssLoader 定义的一些方法

方法	描述
final Class defineClass(String str, byte b[ ], int index, int numBytes) throws ClassFormatError	返回一个类（Class）对象，类的名字在str中，对象包含在由b指定的字节数组中。该数组中对象开始的位置下标由index指定，而该数组的长度为numBytes。b中的数据必须表示一个有效的对象
final Class findSystemClass(String name) throws ClassNotFoundException	返回一个给定名字的类（Class）对象
abstract Class loadClass(String name, boolean callResolveClass) throws ClassNotFoundException	如果callResolveClass为true，这个抽象方法的实现工具必须加载一个给定名字的类，并调用resolveClass( )方法
final void resolveClass(Class obj)	用obj引用的类被解析（即，它的名字被输入在类名字空间中）

14.10　Math

Math类保留了所有用于几何学，三角学以及几种一般用途方法的浮点函数。Math定义了两个双精度（double）常数：E（近似为2.72）和PI（近似为3.14）。

14.10.1　超越函数

下面的三种方法对一个以弧度为单位的角度接收一个双精度（double）参数并且返回它们各自的超越函数的结果：

方法	描述
Static double sin(double arg)	返回由以弧度为单位由arg指定的角度的正弦值
static double cos(double arg)	返回由以弧度为单位由arg指定的角度的余弦值
static double tan(double arg)	返回由以弧度为单位由arg指定的角度的正切值

下面的方法将超越函数的结果作为一个参数，按弧度返回产生这个结果的角度值。它们是其非弧度形式的反。

方法	描述
static double asin(double arg)	返回一个角度，该角度的正弦值由arg指定
static double acos(double arg)	返回一个角度，该角度的余弦值由arg指定
static double atan(double arg)	返回一个角度，该角度的正切值由arg指定
static double atan2(double x, double y)	返回一个角度，该角度的正切值为x/y

14.10.2　指数函数

Math定义了下面的指数方法：

方法	描述
static double exp(double arg)	返回arg的e
static double log(double arg)	返回arg的自然对数值
static double pow(double y, double x)	返回以y为底数，以x为指数的幂值；例如pow(2.0, 3.0)返回8.0
static double sqrt(double arg)	返回arg的平方根

14.10.3　舍入函数

Math类定义了几个提供不同类型舍入运算的方法。这些方法列在表14-15中。

表 14-15 由 Math 定义的舍入方法

方法	描述
static int abs(int arg)	返回arg的绝对值
static long abs(long arg)	返回arg的绝对值
static float abs(float arg)	返回arg的绝对值
static double abs(double arg)	返回arg的绝对值
static double ceil(double arg)	返回大于或等于arg的最小整数
static double floor(double arg)	返回小于或等于arg的最大整数
static int max(int x, int y)	返回x和y中的最大值
static long max(long x, long y)	返回x和y中的最大值
static float max(float x, float y)	返回x和y中的最大值
static double max(double x, double y)	返回x和y中的最大值
static int min(int x, int y)	返回x和y中的最小值
static long min(long x, long y)	返回x和y中的最小值
static float min(float x, float y)	返回x和y中的最小值
static double min(double x, double y)	返回x和y中的最小值
static double rint(double arg)	返回最接近arg的整数值
static int round(float arg)	返回arg的只入不舍的最近的整型（int）值
static long round(double arg)	返回arg的只入不舍的最近的长整型（long）值

14.10.4 其他的数学方法

除了给出的方法，Math还定义了下面这些方法：

```
static double IEEEremainder(double dividend, double divisor)
static double random( )
static double toRadians(double angle)
static double toDegrees(double angle)
```

IEEEremainder( )方法返回dividend/divisor的余数。random( )方法返回一个伪随机数，其值介于0与1之间。在大多数情况下，当需要产生随机数时，通常用Random类。toRadians( )方法将角度的度转换为弧度。而toDegrees( )方法将弧度转换为度。这后两种方法是在Java 2中新增加的。

下面是一个说明toRadians( )和toDegrees( )方法的例子：

```
// Demonstrate toDegrees() and toRadians().
class Angles {
  public static void main(String args[]) {
    double theta = 120.0;

    System.out.println(theta + " degrees is " +
                       Math.toRadians(theta) + " radians.");

    theta = 1.312;
    System.out.println(theta + " radians is " +
```

```
                    Math.toDegrees(theta) + " degrees.");
  }
}
```

程序输出如下所示：

```
120.0 degrees is 2.0943951023931953 radians.
1.312 radians is 75.17206272116401 degrees.
```

14.11　StrictMath

在Java 2的1.3版本中增加了StrictMath类。这个类定义一个与Math中的数学方法类似的一套完整的数学方法。两者的区别在于StrictMath中的方法对所有Java工具保证产生精确一致的结果，而Math中的方法更大程度上是为了提高性能。

14.12　Compiler

Compiler类支持创建将字节码编译而非解释成可执行码的Java环境。常规的程序不使用它。

14.13　Thread，ThreadGroup和Runnable

Runnable接口以及Thread和ThreadGroup类支持多线程编程。下面分别予以说明。

注意：关于管理线程，实现Runnable接口以及创建多线程程序的概述已在第11章中介绍过。

14.13.1　Runnable接口

Runnable接口必须由启动执行的独立线程的类所实现。Runnable仅定义了一种抽象方法，叫做run( )。该方法是线程的入口点。它的形式如下所示：

```
abstract void run( )
```

所创建的线程必须实现该方法。

14.13.2　Thread

Thread创建一个新的执行线程。它定义了如下的构造函数：

```
Thread( )
Thread(Runnable threadOb)
Thread(Runnable threadOb, StringthreadName)
Thread(String threadName)
Thread(ThreadGroup groupOb, Runnable threadOb)
Thread(ThreadGroup groupOb, Runnable threadOb, String threadName)
Thread(ThreadGroup groupOb, String threadName)
```

threadOb是实现Runnable接口的类的一个实例，它定义了线程运行开始的地方。线程的名字由threadName指定。当名字未被指定时，Java虚拟机将创建一个。groupOb指定了新线程所属的线程组。当没有线程组被指定时，新线程与其父线程属于同一线程组。

下面的常数由Thread定义：

```
MAX_PRIORITY
MIN_PRIORITY
NORM_PRIORITY
```

正如所期望的那样，这些常数指定了最大，最小以及默认的线程优先权。

由Thread定义的方法列在表14-16中。在比Java 2早的版本中，Thread中也包括了stop( )，suspend( )以及resume( )方法。然而正如在第11章中解释的那样，这些方法由于其固有的不稳定性而在Java 2中被摈弃了。在Java 2中摈弃的还有countStackFrames( )方法，因为它调用了suspend( )方法。

表 14-16 由 Thread 定义的方法

方法	描述
static int activeCount( )	返回线程所属的线程组中线程的个数
void checkAccess( )	引起安全管理程序检验当前的线程能访问和/或能改变在其上checkAccess( )方法被调用的线程
static Thread currentThread( )	返回一个Thread对象，该对象封装了调用这个方法的线程
void destroy( )	终止线程
static int enumerate(Thread threads[ ])	将当前线程组中的所有Thread对象的拷贝放入threads中。返回线程的个数
ClassLoader getContextClassLoader( )	返回用于对这个线程加载类和资源的类加载程序（在Java 2中新增加的）
final String getName( )	返回线程名
final int getPriority( )	返回线程的属性设置
final ThreadGroup getThreadGroup( )	返回调用线程是其中一个成员的ThreadGroup对象
void interrupt( )	中断线程
static boolean interrupted( )	如果当前执行的线程已经被预先设置了中断，则返回true；否则，返回false
final boolean isAlive( )	如果线程仍在运行中，则返回true；否则返回false
final boolean isDaemon( )	如果线程是一个后台进程线程（Java运行系统的一部分），则返回true；否则返回false
boolean isInterrupted( )	如果线程被中断，则返回true，否则返回false
final void join( ) throws InterruptedException	等待直至线程终止

续表

方法	描述
final void join(long milliseconds) throws InterruptedException	等待直到为终止线程而指定的以毫秒计时的时间
final void join(long milliseconds, int nanoseconds) throws InterruptedException	等待直到为终止线程而指定的以毫秒加毫微秒计时的时间
void run( )	开始线程的执行
void setContextClassLoader(ClassLoader cl)	设置将被调用线程用于cl的类加载程序（在Java 2中新增加的）
final void setDaemon(boolean state)	标记线程为后台进程线程
final void setName(String threadName)	将线程的名字设置为由threadName指定的名字
final void setPriority(int priority)	设置由priority指定的线程优先权
static void sleep(long milliseconds) throws InterruptedException	以指定的毫秒为单位的时间长度挂起执行的线程
static void sleep(long milliseconds, int nanoseconds) throws InterruptedException	以指定的毫秒加毫微秒为单位的时间长度挂起执行的线程
void start( )	开始线程的执行
String toString( )	返回线程的等价字符串形式
static void yield( )	调用线程将CPU让给其他的线程

14.13.3　ThreadGroup

线程组（ThreadGroup）创建了一组线程。它定义了如下的两个构造函数：

```
ThreadGroup(String groupName)
ThreadGroup(ThreadGroup parentOb, String groupName)
```

对于两种形式，groupName指定了线程组的名字。第一种形式创建一个新的线程组，该线程组将当前的线程作为它的父线程。在第二种形式中，父线程由parentOb指定。

由ThreadGroup定义的方法列在表14-17中。在比Java 2更早出现的Java版本中，ThreadGroup中也包括了stop( )，suspend( )以及resume( )方法。这些方法由于其本身固有的不稳定性，而在Java 2中被摈弃。

表 14-17　由 ThreadGroup 定义的方法

方法	描述
int activeCount( )	返回线程组加上以这个线程作为父类的所有线程组中线程的个数
int activeGroupCount( )	返回调用线程是父类的线程的组数
final void checkAccess( )	引起安全管理程序检验调用线程能访问和/或能改变在其上checkAccess( )方法被调用的线程组

续表

方法	描述
final void destroy( )	撤消被调用的线程组（以及任一子线程组）
int enumerate(Thread group[ ])	将构成调用线程组的线程放入group数组中
int enumerate(Thread group[ ], boolean all)	将构成调用线程组的线程放入group数组中。如果all为true，那么线程组的所有子线程组中的线程也被放入group中
int enumerate(ThreadGroup group[ ])	将调用线程组的子线程组放入group数组中
int enumerate(ThreadGroup group[ ], boolean all)	将调用线程组的子线程组放入group数组中。如果all为true，所有子线程组的子线程组（等等）也被放入group中
final int getMaxPriority( )	返回对线程组设置的最大优先权
final String getName( )	返回线程组名
final ThreadGroup getParent( )	如果调用ThreadGroup对象没有父类，则返回null；否则返回调用对象的父类
final void interrupt( )	调用线程组中所有线程的interrupt( )方法（在Java 2中新增加的）
final boolean isDaemon( )	如果线程组是一个端口后台进程组，则返回true；否则返回false
boolean isDestroyed( )	如果线程组已经被破坏，则返回true；否则，返回false
void list( )	显示关于线程组的信息
final boolean parentOf(ThreadGroup group)	如果调用线程是group的父线程（或group本身），则返回true；否则返回false
final void setDaemon(boolean isDaemon)	如果isDaemon为true，那么调用线程组被标记为一个端口后台进程组
final void setMaxPriority(int priority)	对调用线程组设置最大优先权priority
String toString( )	返回线程组的字符串等价形式
void uncaughtException(Thread thread, Throwable e)	当一个异常未被捕获时，该方法被调用

线程组提供了一种方便的方法，可以将一组线程当做一个单元来管理。这在想挂起或恢复一些相关的线程的情况下，是特别有用的。例如假想在一个程序中，有一组线程被用来打印文档，另一组线程被用来将该文档显示在屏幕上，同时另一组线程将文档保存为磁盘文件。如果打印被异常中止了，想用一种很简单的方法停止所有与打印有关的线程。线程组为这种处理提供了方便。下面的程序说明了这种用法，在程序中创建两个线程组，每一线程组中有两个线程：

```
// Demonstrate thread groups.
class NewThread extends Thread {
  boolean suspendFlag;
```

```
  NewThread(String threadname, ThreadGroup tgOb) {
    super(tgOb, threadname);
    System.out.println("New thread: " + this);
    suspendFlag = false;
    start(); // Start the thread
  }

  // This is the entry point for thread.
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println(getName() + ": " + i);
        Thread.sleep(1000);
        synchronized(this) {
          while(suspendFlag) {
            wait();
          }
        }
      }
    } catch (Exception e) {
      System.out.println("Exception in " + getName());
    }
    System.out.println(getName() + " exiting.");
  }

  void mysuspend() {
    suspendFlag = true;
  }

  synchronized void myresume() {
    suspendFlag = false;
    notify();
  }
}

class ThreadGroupDemo {
  public static void main(String args[]) {
    ThreadGroup groupA = new ThreadGroup("Group A");
    ThreadGroup groupB = new ThreadGroup("Group B");

    NewThread ob1 = new NewThread("One", groupA);
    NewThread ob2 = new NewThread("Two", groupA);
    NewThread ob3 = new NewThread("Three", groupB);
    NewThread ob4 = new NewThread("Four", groupB);

    System.out.println("\nHere is output from list():");
    groupA.list();
    groupB.list();
    System.out.println();

    System.out.println("Suspending Group A");
    Thread tga[] = new Thread[groupA.activeCount()];
    groupA.enumerate(tga); // get threads in group
    for(int i = 0; i < tga.length; i++) {
      ((NewThread)tga[i]).mysuspend(); // suspend each thread
```

```
    }

    try {
      Thread.sleep(4000);
    } catch (InterruptedException e) {
      System.out.println("Main thread interrupted.");
    }

    System.out.println("Resuming Group A");
    for(int i = 0; i < tga.length; i++) {
      ((NewThread)tga[i]).myresume(); // resume threads in group
    }

    // wait for threads to finish
    try {
      System.out.println("Waiting for threads to finish.");
      ob1.join();
      ob2.join();
      ob3.join();
      ob4.join();
    } catch (Exception e) {
      System.out.println("Exception in Main thread");
    }

    System.out.println("Main thread exiting.");
  }
}
```

该程序的一个输出样本如下所示：

```
New thread: Thread[One,5,Group A]
New thread: Thread[Two,5,Group A]
New thread: Thread[Three,5,Group B]
New thread: Thread[Four,5,Group B]
Here is output from list():
java.lang.ThreadGroup[name=Group A,maxpri=10]
  Thread[One,5,Group A]
  Thread[Two,5,Group A]
java.lang.ThreadGroup[name=Group B,maxpri=10]
  Thread[Three,5,Group B]
  Thread[Four,5,Group B]
Suspending Group A
Three: 5
Four: 5
Three: 4
Four: 4
Three: 3
Four: 3
Three: 2
Four: 2
Resuming Group A
Waiting for threads to finish.
One: 5
Two: 5
Three: 1
```

```
Four: 1
One: 4
Two: 4
Three exiting.
Four exiting.
One: 3
Two: 3
One: 2
Two: 2
One: 1
Two: 1
One exiting.
Two exiting.
Main thread exiting.
```

注意在这个程序中，线程组A被挂起四秒。由于输出确认，造成线程One和线程Two暂停，但是线程Three和线程Four仍然运行。四秒钟之后，线程One和线程Two被恢复。注意线程组A是如何被挂起和恢复的。首先通过对线程组A调用enumerate( )方法得到线程组A中的线程。然后每一个线程重复通过得到的数组而被挂起。为了恢复线程组A中的线程，序列再一次被遍历，每一个线程被恢复。最后一点：这个例子使用了Java 2推荐使用的方法去完成挂起和恢复线程的任务。而没有用在Java 2中被摈弃的方法suspend( )和resume( )。

14.14　ThreadLocal和InheritableThreadLocal

在Java 2的java.lang中增加了两个与线程有关的类：

- ThreadLocal　用于创建线程局部变量。每个线程都拥有自己局部变量的拷贝。
- InheritableThreadLocal　创建可以被继承的线程局部变量。

14.15　Package

在Java 2中增加了一个称为Package的类。这个类封装了与包有关的版本数据。包版本信息由于包的增值以及由于Java程序可能需要知道哪些包版本可以利用而变得更加重要。Package中定义的方法列在表14-18中。下面的程序通过显示程序当前已知的包而说明了Package。

表 14-18　由 Package 定义的方法

方法	描述
String getImplementationTitle( )	返回调用包的标题
String getImplementationVendor( )	返回调用包的实现程序的程序名
String getImplementationVersion( )	返回调用包的版本号
String getName( )	返回调用包的名字
Static Package getPackage(String pkgName)	返回一个由pkgName指定的Package对象

续表

方法	描述
Static Package[ ] getPackages( )	返回调用程序当前已知的所有包
String getSpecificationTitle( )	返回调用包的规格说明的标题
String getSpecificationVendor( )	返回对调用包的规格说明的所有者的名字
String getSpecificationVersion( )	返回调用包的规格说明的版本号
Int hashCode( )	返回调用包的散列码
Boolean isCompatibleWith(String verNum) throws NumberFormatException	如果verNum小于或等于调用包的版本号，则返回true
Boolean isSealed( )	如果调用包被封，则返回true；否则返回false
Boolean isSealed(URL url)	如果调用包相对于url被封，则返回true；否则返回false。
String toString( )	返回调用包的等价字符串形式

```
// Demonstrate Package
class PkgTest {
  public static void main(String args[]) {
    Package pkgs[];

    pkgs = Package.getPackages();

    for(int i=0; i < pkgs.length; i++)
      System.out.println(
            pkgs[i].getName() + " " +
            pkgs[i].getImplementationTitle() + " " +
            pkgs[i].getImplementationVendor() + " " +
            pkgs[i].getImplementationVersion()
      );

  }
}
```

14.16 RuntimePermission

在Java 2的java.lang中也新增加了RuntimePermission。它与Java的安全机制有关，这里不做进一步的讨论。

14.17 Throwable

Throwable类支持Java的异常处理系统，它是派生所有异常类的类。在本书第10章已经讨论过它。

14.18　SecurityManager

SecurityManager是一个子类可以实现的抽象类，它用于创建一个安全管理程序。一般不需要实现自己的安全管理程序，如果非要这样做，需要查阅与你的Java开发系统一起得到的相关文档。

14.19　Comparable接口

Java 2在java.lang中新增加了一个接口：Comparable。实现Comparable的类的对象可以被排序。换句话说，实现Comparable的类包含了可以按某种有意义的方式进行比较的对象。Comparable接口说明了一个方法，该方法用于确定Java 2调用一个类的实例的自然顺序。该方法如下所示：

```
int compareTo(Object obj)
```

这个方法比较调用对象和obj。如果他们相等，就返回0。如果调用对象比obj小，则返回一个负值。否则返回一个正值。

该接口由前面已经介绍的几种类实现。特别是Byte，Character，Double，Float，Long，Short，String以及Integer类定义了compareTo( )方法。另外，下一章将会介绍到，实现这个接口的对象可以被使用在不同的集合中。

14.20　java.lang.ref和java.lang.reflect包

在Java中定义了两个java.lang的子包：java.lang.ref和java.lang.reflect。下面分别予以简单介绍。

14.20.1　java.lang.ref

在前面学到过，在Java中，垃圾回收工具自动确定何时对一个对象，没有引用存在。然后这个对象就被认为是不再需要的，同时它所占的内存也被释放。在Java 2中新增加的java.lang.ref包中的类对垃圾回收处理提供更加灵活的控制。例如，假设你的程序创建了大量的在后面某个时间又想重新使用的对象，可以持续保持对这些对象的引用，但是这可能需要更多的内存开销。

作为替代，可以对这些对象定义“软”引用。如果可以利用的内存接近用完的话，一个可以“软实现”的对象可以从垃圾回收工具中释放。在那种情况下，垃圾回收工具将这个对象的“软”引用设为空（null）。否则，垃圾回收工具保存对象以便以后使用。

程序设计人员具有确定是否一个“软实现”的对象被释放的能力。如果它被释放了，可以重新创建它。如果没有释放，该对象对于后面的应用将一直是可以利用的。也可以为对象创建“弱”（weak）和“假想”（phantom）引用，不过关于这些以及java.lang.ref包中

其他特性的讨论已经超过了本书的范围。

14.20.2 java.lang.reflect

Reflection是一个程序分析自己的能力。包java.lang.reflect提供了获得关于一个类的域、构造函数、方法和修改符的能力。需要这些信息去创建可以使你利用Java Beans组件的软件工具。这个工具使用映射动态地确定组件的特征。这个主题将在第25章中讨论。

另外，包java.lang.reflect包括了一个可以动态创建和访问数组的类。

第 15 章　java.util 第 1 部分：类集框架

java.util包中包含了一些在Java 2中新增加的最令人兴奋的增强功能：类集。一个类集（collection）是一组对象。类集的增加使得许多java.util中的成员在结构和体系结构上发生根本的改变。它也扩展了包可以被应用的任务范围。类集是被所有Java程序员紧密关注的最新型的技术。

除了类集，java.util还包含了支持范围广泛的函数的各种各样的类和接口。这些类和接口被核心的Java包广泛使用，同时当然也可以被你编写的程序所使用。对它们的应用包括产生伪随机数，对日期和时间的操作，观测事件，对位集的操作以及标记字符串。由于java.util具有许多特性，因此它是Java中最被广泛使用的一个包。

java.util中包含的类如下。在Java 2中新增加的一些也被列出：

AbstractCollection (Java 2)	EventObject	Random
AbstractList (Java 2)	GregorianCalendar	ResourceBundle
AbstractMap (Java 2)	HashMap (Java 2)	SimpleTimeZone
AbstractSequentialList (Java 2)	HashSet (Java 2)	Stack
AbstractSet (Java 2)	Hashtable	StringTokenizer
ArrayList (Java 2)	LinkedList (Java 2)	Timer (Java 2, v1.3)
Arrays (Java 2)	ListResourceBundle	TimerTask (Java 2, v1.3)
BitSet	Locale	TimeZone
Calendar	Observable	TreeMap (Java 2)
Collections (Java 2)	Properties	TreeSet (Java 2)
Date	PropertyPermission (Java 2)	Vector
Dictionary	PropertyResourceBundle	WeakHashMap (Java 2)

java.util定义了如下的接口。注意其中大多数是在Java 2中新增加的。

Collection (Java 2)	List (Java 2)	Observer
Comparator (Java 2)	ListIterator (Java 2)	Set (Java 2)
Enumeration	Map (Java 2)	SortedMap (Java 2)
EventListener	Map.Entry (Java 2)	SortedSet (Java 2)
Iterator (Java 2)		

ResourceBundle类，ListResourceBundle类和PropertyResourceBundle类帮助具有特定地区资源的大型程序国际化。关于这些类的讨论，在这里从略。授权对系统属性进行读/写的PropertyPermission类也超过了本书的讨论范围。EventObject和EventListener类将在第20章讨论。下面将对剩下的类和接口做详细的讨论。

由于java.util包非常大，关于它的讨论将分成两章进行。本章讨论那些与对象的类集有关的成员。在第16章讨论其他的类和接口。

15.1 类 集 概 述

Java的类集（Collection）框架使你的程序处理对象组的方法标准化。在Java 2出现之前，Java提供了一些专门的类如Dictionary，Vector，Stack和Properties去存储和操作对象组。尽管这些类非常有用，它们却缺少一个集中，统一的主题。因此例如说使用Vector的方法就会与使用Properties的方法不同。以前的专门的方法也没有被设计成易于扩展和能适应新的环境的形式。而类集解决了这些（以及其他的一些）问题。

类集框架被设计用于适应几个目的。首先，这种框架是高性能的。对基本类集（动态数组，链接表，树和散列表）的实现是高效率的。一般很少需要人工去对这些“数据引擎”编写代码（如果有的话）。第二点，框架必须允许不同类型的类集以相同的方式和高度互操作方式工作。第三点，类集必须是容易扩展和/或修改的。为了实现这一目标，类集框架被设计成包含一组标准的接口。对这些接口，提供了几个标准的实现工具（例如LinkedList，HashSet和TreeSet），通常就是这样使用的。如果你愿意的话，也可以实现你自己的类集。为了方便起见，创建用于各种特殊目的的实现工具。一部分工具可以使你自己的类集实现更加容易。最后，增加了允许将标准数组融合到类集框架中的机制。

算法（Algorithms）是类集机制的另一个重要部分。算法操作类集，它在Collections类中被定义为静态方法。因此它们可以被所有的类集所利用。每一个类集类不必实现它自己的方案，算法提供了一个处理类集的标准方法。

由类集框架创建的另一项是Iterator接口。一个迭代程序（iterator）提供了一个多用途的，标准化的方法，用于每次访问类集的一个元素。因此迭代程序提供了一种枚举类集内容（enumerating the contents of a collection）的方法。因为每一个类集都实现Iterator，所以通过由Iterator定义的方法，任一类集类的元素都能被访问到。因此，稍作修改，循环通过集合的程序代码也可以被用来循环通过列表。

除了类集之外，框架定义了几个映射接口和类。映射（Maps）存储键/值对。尽管映射在对项的正确使用上不是“类集”，但它们完全用类集集成。在类集框架的语言中，可以获得映射的类集“视图”（collection-view）。这个“视图”包含了从存储在类集中的映射得到的元素。因此，如果选择了一个映射，就可以将其当做一个类集来处理。

对于由java.util定义的原始类，类集机制被更新以便它们也能够集成到新的系统里。所以理解下面的说法是很重要的：尽管类集的增加改变了许多原始工具类的结构，但它却不会导致被抛弃。类集仅仅是提供了处理事情的一个更好的方法。

最后的一点：如果你对C++比较熟悉的话，那么你可以发现Java的类集技术与在C++中定义的标准模板库（STL）相似。在C++中叫做容器（container），而在Java中叫做类集。

15.2　类 集 接 口

类集框架定义了几个接口。本节对每一个接口都进行了概述。首先讨论类集接口是因为它们决定了collection类的基本特性。不同的是，具体类仅仅是提供了标准接口的不同实现。支持类集的接口总结在如下的表中：

接口	描述
Collection	能使你操作对象组，它位于类集层次结构的顶层
List	扩展Collection去处理序列（对象的列表）
Set	扩展Collection去处理集合，集合必须包含唯一元素
SortedSet	扩展Set去处理排序集合

除了类集接口之外，类集也使用Comparator，Iterator和ListIterator接口。关于这些接口将在本章后面做更深入的描述。简单地说，Comparator接口定义了两个对象如何比较；Iterator和ListIterator接口枚举类集中的对象。

为了在它们的使用中提供最大的灵活性，类集接口允许对一些方法进行选择。可选择的方法使得使用者可以更改类集的内容。支持这些方法的类集被称为可修改的（modifiable）。不允许修改其内容的类集被称为不可修改的（unmodifiable）。如果对一个不可修改的类集使用这些方法，将引发一个UnsupportedOperationException异常。所有内置的类集都是可修改的。

下面讨论类集接口。

15.2.1　类集接口

Collection接口是构造类集框架的基础。它声明所有类集都将拥有的核心方法。这些方法被总结在表15-1中。因为所有类集实现Collection，所以熟悉它的方法对于清楚地理解框架是必要的。其中几种方法可能会引发一个UnsupportedOperationException异常。正如上面解释的那样，这些发生在当类集不能被修改时。当一个对象与另一个对象不兼容，例如当企图增加一个不兼容的对象到一个类集中时。将产生一个ClassCastException异常。

表 15-1　由 Collection 定义的方法

方法	描述
boolean add(Object obj)	将obj加入到调用类集中。如果obj被加入到类集中了，则返回true；如果obj已经是类集中的一个成员或类集不能被复制时，则返回false
boolean addAll(Collection c)	将c中的所有元素都加入到调用类集中，如果操作成功（也就是说元素被加入了），则返回true；否则返回false
void clear( )	从调用类集中删除所有元素
boolean contains(Object obj)	如果obj是调用类集的一个元素，则返回true，否则，返回false

续表

方法	描述
boolean containsAll(Collection c)	如果调用类集包含了c中的所有元素，则返回true；否则，返回false
boolean equals(Object obj)	如果调用类集与obj相等，则返回true；否则返回false
int hashCode( )	返回调用类集的散列码
boolean isEmpty( )	如果调用类集是空的，则返回true；否则返回false
Iterator iterator( )	返回调用类集的迭代程序
Boolean remove(Object obj)	从调用类集中删除obj的一个实例。如果这个元素被删除了，则返回true；否则返回false
Boolean removeAll(Collection c)	从调用类集中删除c的所有元素。如果类集被改变了（也就是说元素被删除了），则返回true；否则返回false
Boolean retainAll(Collection c)	删除调用类集中除了包含在c中的元素之外的全部元素。如果类集被改变了（也就是说元素被删除了），则返回true，否则返回false
int size( )	返回调用类集中元素的个数
Object[ ] toArray( )	返回一个数组，该数组包含了所有存储在调用类集中的元素。数组元素是类集元素的拷贝
Object[ ] toArray(Object array[ ])	返回一个数组，该数组仅仅包含了那些类型与数组元素类型匹配的类集元素。数组元素是类集元素的拷贝。如果array的大小与匹配元素的个数相等，它们被返回到array。如果array的大小比匹配元素的个数小，将分配并返回一个所需大小的新数组，如果array的大小比匹配元素的个数大，在数组中，在类集元素之后的单元被置为null。如果任一类集元素的类型都不是array的子类型，则引发一个ArrayStoreException异常

调用add( )方法可以将对象加入类集。注意add( )带一个Object类型的参数。因为Object是所有类的超类，所以任何类型的对象可以被存储在一个类集中。然而原始类型可能不行。例如，一个类集不能直接存储类型int，char，double等的值。当然如果想存储这些对象，也可以使用在第14章中介绍的原始类型包装器之一。可以通过调用addAll( )方法将一个类集的全部内容增加到另一个类集中。

可以通过调用remove( )方法将一个对象删除。为了删除一组对象，可以调用removeAll( )方法。调用retainAll( )方法可以将除了一组指定的元素之外的所有元素删除。为了清空类集，可以调用clear( )方法。

通过调用contains( )方法，可以确定一个类集是否包含了一个指定的对象。为了确定一个类集是否包含了另一个类集的全部元素，可以调用containsAll( )方法。当一个类集是空的时候，可以通过调用isEmpty( )方法来予以确认。调用size( )方法可以获得类集中当前元素的个数。

toArray( )方法返回一个数组，这个数组包含了存储在调用类集中的元素。这个方法比它初看上去的能力要更重要。经常使用类数组语法来处理类集的内容是有优势的。通过在类集和数组之间提供一条路径，可以充分利用这两者的优点。

调用equals( )方法可以比较两个类集是否相等。“相等”的精确含义可以不同于从类集到类集。例如，可以执行equals( )方法以便用于比较存储在类集中的元素的值，换句话说，equals( )方法能比较对元素的引用。

一个更加重要的方法是iterator( )，该方法对类集返回一个迭代程序。正如你将看到的那样，当使用一个类集框架时，迭代程序对于成功的编程来说是至关重要的。

15.2.2　List接口

List接口扩展了Collection并声明存储一系列元素的类集的特性。使用一个基于零的下标，元素可以通过它们在列表中的位置被插入和访问。一个列表可以包含复制元素。

除了由Collection定义的方法之外，List还定义了一些它自己的方法，这些方法总结在表15-2中。再次注意当类集不能被修改时，其中的几种方法引发UnsupportedOperationException异常。当一个对象与另一个不兼容，例如当企图将一个不兼容的对象加入一个类集中时，将产生ClassCastException异常。

表 15-2　由 List 定义的方法

方法	描述
void add(int index, Object obj)	将obj插入调用列表，插入位置的下标由index传递。任何已存在的，在插入点以及插入点之后的元素将前移。因此，没有元素被覆盖
boolean addAll(int index, Collection c)	将c中的所有元素插入到调用列表中，插入点的下标由index传递。在插入点以及插入点之后的元素将前移。因此，没有元素被覆盖。如果调用列表改变了，则返回true；否则返回false
Object get(int index)	返回存储在调用类集内指定下标处的对象
int indexOf(Object obj)	返回调用列表中obj的第一个实例的下标。如果obj不是列表中的元素，则返回-1
int lastIndexOf(Object obj)	返回调用列表中obj的最后一个实例的下标。如果obj不是列表中的元素，则返回-1
ListIterator listIterator( )	返回调用列表开始的迭代程序
ListIterator listIterator(int index)	返回调用列表在指定下标处开始的迭代程序
Object remove(int index)	删除调用列表中index位置的元素并返回删除的元素。删除后，列表被压缩。也就是说，被删除元素后面的元素的下标减一
Object set(int index, Object obj)	用obj对调用列表内由index指定的位置进行赋值
List subList(int start, int end)	返回一个列表，该列表包括了调用列表中从start到end–1的元素。返回列表中的元素也被调用对象引用

对于由Collection定义的add( )和addAll( )方法，List增加了方法add(int, Object)和

addAll(int, Collection)。这些方法在指定的下标处插入元素。由Collection定义的add(Object)和addAll(Collection)的语义也被List改变了，以便它们在列表的尾部增加元素。

为了获得在指定位置存储的对象，可以用对象的下标调用get( )方法。为了给类表中的一个元素赋值，可以调用set( )方法，指定被改变的对象的下标。调用indexOf( )或lastIndexOf( )可以得到一个对象的下标。

通过调用subList( )方法，可以获得列表的一个指定了开始下标和结束下标的子列表。正如你能想象到的，subList( )方法使得列表处理十分方便。

15.2.3 集合接口

集合接口定义了一个集合。它扩展了Collection并说明了不允许复制元素的类集的特性。因此，如果试图将复制元素加到集合中时，add( )方法将返回false。它本身并没有定义任何附加的方法。

15.2.4 SortedSet接口

SortedSet接口扩展了Set并说明了按升序排列的集合的特性。除了那些由Set定义的方法之外，由SortedSet接口说明的方法列在表15-3中。当没有项包含在调用集合中时，其中的几种方法引发NoSuchElementException异常。当对象与调用集合中的元素不兼容时，引发ClassCastException异常。如果试图使用null对象，而集合不允许null时，引发NullPointerException异常。

表 15-3 由 SortedSet 定义的方法

方法	描述
Comparator comparator( )	返回调用被排序集合的比较函数，如果对该集合使用自然顺序，则返回null
Object first( )	返回调用被排序集合的第一个元素
SortedSet headSet(Object end)	返回一个包含那些小于end的元素的SortedSet，那些元素包含在调用被排序集合中。返回被排序集合中的元素也被调用被排序集合所引用
Object last( )	返回调用被排序集合的最后一个元素
SortedSet subSet(Object start, Object end)	返回一个SortedSet，它包括了从start到end–1的元素。返回类集中的元素也被调用对象所引用
SortedSet tailSet(Object start)	返回一个SortedSet，它包含了那些包含在分类集合中的大于等于start的元素。返回集合中的元素也被调用对象所引用

SortedSet定义了几种方法，使得对集合的处理更加方便。调用first( )方法，可以获得集合中的第一个对象。调用last( )方法，可以获得集合中的最后一个元素。调用subSet( )方法，可以获得排序集合的一个指定了第一个和最后一个对象的子集合。如果需要得到从集合的第一个元素开始的一个子集合，可以使用headSet( )方法。如果需要获得集合尾部的一个子集合，可以使用tailSet( )方法。

15.3　Collection类

现在，你已经熟悉了类集接口，下面开始讨论实现它们的标准类。一些类提供了完整的可以被使用的工具。另一些类是抽象的，提供主框架工具，作为创建具体类集的起始点。没有Collection类是同步的，但正如你将在本章后面看到的那样，有可能获得同步版本。

标准的Collection类总结在下面的表中。

类	描述
AbstractCollection	实现大多数Collection接口
AbstractList	扩展AbstractCollection并实现大多数List接口
AbstractSequentialList	为了被类集使用而扩展AbstractList，该类集使用连续而不是随机方式访问其元素
LinkedList	通过扩展AbstractSequentialList来实现链接表
ArrayList	通过扩展AbstractList来实现动态数组
AbstractSet	扩展AbstractCollection并实现大多数Set接口
HashSet	为了使用散列表而扩展AbstractSet
TreeSet	实现存储在树中的一个集合。扩展AbstractSet

注意：除了Collection类外，还有几个从以前版本遗留下来的类，如Vector，Stack和Hashtable均被重新设计成支持类集的形式。这些内容将在本章后面讨论。

下面讨论具体的Collection类，举例说明它们的用法。

15.3.1　ArrayList类

ArrayList类扩展AbstractList并执行List接口。ArrayList支持可随需要而增长的动态数组。在Java中，标准数组是定长的。在数组创建之后，它们不能被加长或缩短，这也就意味着你必须事先知道数组可以容纳多少元素。但是，你直到运行时才能知道需要多大的数组。为了解决这个问题，类集框架定义了ArrayList。本质上，ArrayList是对象引用的一个变长数组。也就是说，ArrayList能够动态地增加或减小其大小。数组列表以一个原始大小被创建。当超过了它的大小，类集自动增大。当对象被删除后，数组就可以缩小。

注意：动态数组也被从以前版本遗留下来的类Vector所支持。关于这一点，将在本章后面介绍。

ArrayList有如下的构造函数：

```
ArrayList( )
ArrayList(Collection c)
ArrayList(int capacity)
```

其中第一个构造函数建立一个空的数组列表。第二个构造函数建立一个数组列表，该

数组列表由类集c中的元素初始化。第三个构造函数建立一个数组列表，该数组有指定的初始容量（capacity）。容量是用于存储元素的基本数组的大小。当元素被追加到数组列表上时，容量会自动增加。

下面的程序展示了ArrayList的一个简单应用。首先创建一个数组列表，接着添加类型String的对象（回想一个引用字符串被转化成一个字符串（String）对象）。接着列表被显示出来。将其中的一些元素删除后，再一次显示列表。

```
// Demonstrate ArrayList.
import java.util.*;

class ArrayListDemo {
  public static void main(String args[]) {
    // create an array list
    ArrayList al = new ArrayList();

    System.out.println("Initial size of al: " +
                       al.size());

    // add elements to the array list
    al.add("C");
    al.add("A");
    al.add("E");
    al.add("B");
    al.add("D");
    al.add("F");
    al.add(1, "A2");

    System.out.println("Size of al after additions: " +
                       al.size());

    // display the array list
    System.out.println("Contents of al: " + al);

    // Remove elements from the array list
    al.remove("F");
    al.remove(2);

    System.out.println("Size of al after deletions: " +
                       al.size());
    System.out.println("Contents of al: " + al);
  }
}
```

该程序的输出如下所示：

```
Initial size of al: 0
Size of al after additions: 7
Contents of al: [C, A2, A, E, B, D, F]
Size of al after deletions: 5
Contents of al: [C, A2, E, B, D]
```

注意a1开始时是空的，当添加元素后，它的大小增加了。当有元素被删除后，它的大

小又会变小。

在前面的例子中，使用由toString( )方法提供的默认的转换显示类集的内容，toString( )方法是从AbstractCollection继承下来的。尽管它对简短的例子程序来说是足够了，然而很少使用这种方法去显示实际中的类集的内容。通常编程者会提供自己的输出程序。但在下面的几个例子中，仍将采用由toString( )方法创建的默认输出。

尽管当对象被存储在ArrayList对象中时，其容量会自动增加。仍可以通过调用ensureCapacity( )方法来人工地增加ArrayList的容量。如果事先知道将在当前能够容纳的类集中存储许许多多的项时，你可能会想这样做。在开始时，通过一次性地增加它的容量，就能避免后面的再分配。因为再分配是很花时间的，避免不必要的处理可以改善性能。ensureCapacity( )方法的特征如下所示：

```
void ensureCapacity(int cap)
```

这里，cap是新的容量。

相反地，如果想要减小在ArrayList对象之下的数组的大小，以便它有正好容纳当前项的大小，可以调用trimToSize( )方法。该方法说明如下：

```
void trimToSize( )
```

从数组列表（ArrayList）获得数组（Array）

当使用ArrayList时，有时想要获得一个实际的数组，这个数组包含了列表的内容。正如前面解释的那样，可以通过调用方法toArray( )来实现它。下面是几个为什么可能想将类集转换成为数组的原因：

- 对于特定的操作，可以获得更快的处理时间。
- 为了给方法传递数组，而方法不必重载去接收类集。
- 为了将新的基于类集的程序与不认识类集的老程序集成。

无论何种原因，如下面的例子程序所示，将ArrayList转换成数组是一件繁琐的事情。

```
// Convert an ArrayList into an array.
import java.util.*;

class ArrayListToArray {
  public static void main(String args[]) {
    // Create an array list
    ArrayList al = new ArrayList();

    // Add elements to the array list
    al.add(new Integer(1));
    al.add(new Integer(2));
    al.add(new Integer(3));
    al.add(new Integer(4));

    System.out.println("Contents of al: " + al);

    // get array
```

```
    Object ia[] = al.toArray();
    int sum = 0;

    // sum the array
    for(int i=0; i<ia.length; i++)
      sum += ((Integer) ia[i]).intValue();

    System.out.println("Sum is: " + sum);
  }
}
```

该程序的输出如下所示：

```
Contents of al: [1, 2, 3, 4]
Sum is: 10
```

程序开始时创建一个整数的类集。正如上面做出的解释那样，由于不能将原始类型存储在类集中，因此类型Integer的对象被创建并被保存。接下来，toArray( )方法被调用，它获得了一个Objects数组。这个数组的内容被置为整型（Integer），接下来对这些值进行求和。

15.3.2 LinkedList类

LinkedList类扩展AbstractSequentialList并执行List接口。它提供了一个链接列表数据结构。它具有如下的两个构造函数，说明如下：

```
LinkedList( )
LinkedList(Collection c)
```

第一个构造函数建立一个空的链接列表。第二个构造函数建立一个链接列表，该链接列表由类集c中的元素初始化。

除了它继承的方法之外，LinkedList类本身还定义了一些有用的方法，这些方法主要用于操作和访问列表。使用addFirst( )方法可以在列表头增加元素；使用addLast( )方法可以在列表的尾部增加元素。它们的形式如下所示：

```
void addFirst(Object obj)
void addLast(Object obj)
```

这里，obj是被增加的项。

调用getFirst( )方法可以获得第一个元素。调用getLast( )方法可以得到最后一个元素。它们的形式如下所示：

```
Object getFirst( )
Object getLast( )
```

为了删除第一个元素，可以使用removeFirst( )方法；为了删除最后一个元素，可以调用removeLast( )方法。它们的形式如下所示：

```
Object removeFirst( )
Object removeLast( )
```

下面的程序举例说明了几个LinkedList支持的方法。

```
// Demonstrate LinkedList.
import java.util.*;

class LinkedListDemo {
  public static void main(String args[]) {
    // create a linked list
    LinkedList ll = new LinkedList();

    // add elements to the linked list
    ll.add("F");
    ll.add("B");
    ll.add("D");
    ll.add("E");
    ll.add("C");
    ll.addLast("Z");
    ll.addFirst("A");

    ll.add(1, "A2");

    System.out.println("Original contents of ll: " + ll);

    // remove elements from the linked list
    ll.remove("F");
    ll.remove(2);

    System.out.println("Contents of ll after deletion: "
                       + ll);

    // remove first and last elements
    ll.removeFirst();
    ll.removeLast();

    System.out.println("ll after deleting first and last: "
                       + ll);

    // get and set a value
    Object val = ll.get(2);
    ll.set(2, (String) val + " Changed");

    System.out.println("ll after change: " + ll);
  }
}
```

该程序的输出如下所示：

```
Original contents of ll: [A, A2, F, B, D, E, C, Z]
Contents of ll after deletion: [A, A2, D, E, C, Z]
ll after deleting first and last: [A2, D, E, C]
ll after change: [A2, D, E Changed, C]
```

因为LinkedList实现List接口，调用add(Object)将项目追加到列表的尾部，如同addLast( )方法所做的那样。使用add( )方法的add(int, Object)形式，插入项目到指定的位置，如例子

程序中调用add（1，“A2”）的举例。

注意如何通过调用get( )和set( )方法而使得ll中的第三个元素发生了改变。为了获得一个元素的当前值，通过get( )方法传递存储该元素的下标值。为了对这个下标位置赋一个新值，通过set( )方法传递下标和对应的新值。

15.3.3 HashSet类

HashSet扩展AbstractSet并且实现Set接口。它创建一个类集，该类集使用散列表进行存储。正像大多数读者很可能知道的那样，散列表通过使用称之为散列法的机制来存储信息。在散列（hashing）中，一个关键字的信息内容被用来确定唯一的一个值，称为散列码（hash code）。而散列码被用来当做与关键字相连的数据的存储下标。关键字到其散列码的转换是自动执行的——你看不到散列码本身。你的程序代码也不能直接索引散列表。散列法的优点在于即使对于大的集合，它允许一些基本操作如add( )，contains( )，remove( )和size( )方法的运行时间保持不变。

下面的构造函数定义为：

```
HashSet( )
HashSet(Collection c)
HashSet(int capacity)
HashSet(int capacity, float fillRatio)
```

第一种形式构造一个默认的散列集合。第二种形式用c中的元素初始化散列集合。第三种形式用capacity初始化散列集合的容量。第四种形式用它的参数初始化散列集合的容量和填充比（也称为加载容量）。填充比必须介于0.0与1.0之间，它决定在散列集合向上调整大小之前，有多少能被充满。具体的说，就是当元素的个数大于散列集合容量乘以它的填充比时，散列集合被扩大。对于没有获得填充比的构造函数，默认使用0.75。

HashSet没有定义任何超过它的超类和接口提供的其他方法。

重要的是，注意散列集合并没有确保其元素的顺序，因为散列法的处理通常不让自己参与创建排序集合。如果需要排序存储，另一种类集——TreeSet将是一个更好的选择。

这里是一个说明HashSet的例子。

```
// Demonstrate HashSet.
import java.util.*;

class HashSetDemo {
  public static void main(String args[]) {
    // create a hash set
    HashSet hs = new HashSet();

    // add elements to the hash set
    hs.add("B");
    hs.add("A");
    hs.add("D");
    hs.add("E");
    hs.add("C");
    hs.add("F");
```

```
    System.out.println(hs);
  }
}
```

下面是该程序的输出：

```
[A, F, E, D, C, B]
```

如上面解释的那样，元素并没有按顺序进行存储。

15.3.4　TreeSet类

TreeSet为使用树来进行存储的Set接口提供了一个工具，对象按升序存储。访问和检索是很快的。在存储了大量的需要进行快速检索的排序信息的情况下，TreeSet是一个很好的选择。

下面的构造函数定义为：

```
TreeSet( )
TreeSet(Collection c)
TreeSet(Comparator comp)
TreeSet(SortedSet ss)
```

第一种形式构造一个空的树集合，该树集合将根据其元素的自然顺序按升序排序。第二种形式构造一个包含了c的元素的树集合。第三种形式构造一个空的树集合，它按照由comp指定的比较函数进行排序（比较函数将在本章后面介绍）。第四种形式构造一个包含了ss的元素的树集合

这里是一个说明TreeSet的例子。

```
// Demonstrate TreeSet.
import java.util.*;

class TreeSetDemo {
  public static void main(String args[]) {
    // Create a tree set
    TreeSet ts = new TreeSet();

    // Add elements to the tree set
    ts.add("C");
    ts.add("A");
    ts.add("B");
    ts.add("E");
    ts.add("F");
    ts.add("D");

    System.out.println(ts);
  }
}
```

这个程序的输出如下所示：

```
 [A, B, C, D, E, F]
```

正如上面解释的那样，因为TreeSet按树存储其元素，它们被按照排序次序自动安排，

如程序输出所示。

15.4 通过迭代函数访问类集

通常希望循环通过类集中的元素。例如，可能会希望显示每一个元素。到目前为止，处理这个问题的最简单方法是使用iterator，iterator是一个或者实现Iterator或者实现ListIterator接口的对象。Iterator可以完成循环通过类集，从而获得或删除元素。ListIterator扩展Iterator，允许双向遍历列表，并可以修改单元。Iterator接口说明的方法总结在表15-4中。ListIterator接口说明的方法总结在表15-5中。

表 15-4 由 Iterator 定义的方法

方法	描述
boolean hasNext( )	如果存在更多的元素，则返回true，否则返回false
Object next( )	返回下一个元素。如果没有下一个元素，则引发NoSuchElementException异常
void remove( )	删除当前元素，如果试图在调用next( )方法之后，调用remove( )方法，则引发IllegalStateException异常

表 15-5 由 ListIterator 定义的方法

方法	描述
void add(Object obj)	将obj插入列表中的一个元素之前，该元素在下一次调用next( )方法时，被返回
boolean hasNext( )	如果存在下一个元素，则返回true；否则返回false
boolean hasPrevious( )	如果存在前一个元素，则返回true；否则返回false
Object next( )	返回下一个元素，如果不存在下一个元素，则引发一个NoSuchElementException异常
int nextIndex( )	返回下一个元素的下标，如果不存在下一个元素，则返回列表的大小
Object previous( )	返回前一个元素，如果前一个元素不存在，则引发一个NoSuchElementException异常
int previousIndex( )	返回前一个元素的下标，如果前一个元素不存在，则返回-1
void remove( )	从列表中删除当前元素。如果remove( )方法在next( )方法或previous( )方法调用之前被调用，则引发一个IllegalStateException异常
void set(Object obj)	将obj赋给当前元素。这是上一次调用next( )方法或previous( )方法最后返回的元素

15.4.1 使用迭代函数

在通过迭代函数访问类集之前，必须得到一个迭代函数。每一个Collection类都提供一个iterator( )函数，该函数返回一个对类集头的迭代函数。通过使用这个迭代函数对象，可以访问类集中的每一个元素，一次一个元素。通常，使用迭代函数循环通过类集的内容，

步骤如下：

1. 通过调用类集的iterator( )方法获得对类集头的迭代函数。
2. 建立一个调用hasNext( )方法的循环，只要hasNext( )返回true，就进行循环迭代。
3. 在循环内部，通过调用next( )方法来得到每一个元素。

对于执行List的类集，也可以通过调用ListIterator来获得迭代函数。正如上面解释的那样，列表迭代函数提供了前向或后向访问类集的能力，并可让你修改元素。否则，ListIterator如同Iterator功能一样。

这里是一个实现这些步骤的例子，说明了Iterator和ListIterator。它使用ArrayList对象，但是总的原则适用于任何类型的类集。当然，ListIterator只适用于那些实现List接口的类集。

```
// Demonstrate iterators.
import java.util.*;

class IteratorDemo {
  public static void main(String args[]) {
    // create an array list
    ArrayList al = new ArrayList();

    // add elements to the array list
    al.add("C");
    al.add("A");
    al.add("E");
    al.add("B");
    al.add("D");
    al.add("F");

    // use iterator to display contents of al
    System.out.print("Original contents of al: ");
    Iterator itr = al.iterator();
    while(itr.hasNext()) {
      Object element = itr.next();
      System.out.print(element + " ");
    }
    System.out.println();

    // modify objects being iterated
    ListIterator litr = al.listIterator();
    while(litr.hasNext()) {
      Object element = litr.next();
      litr.set(element + "+");
    }

    System.out.print("Modified contents of al: ");
    itr = al.iterator();
    while(itr.hasNext()) {
      Object element = itr.next();
      System.out.print(element + " ");
    }
    System.out.println();
```

```
    // now, display the list backwards
    System.out.print("Modified list backwards: ");
    while(litr.hasPrevious()) {
      Object element = litr.previous();
      System.out.print(element + " ");
    }
    System.out.println();
  }
}
```

程序的输出如下所示：

```
Original contents of al: C A E B D F
Modified contents of al: C+ A+ E+ B+ D+ F+
Modified list backwards: F+ D+ B+ E+ A+ C+
```

特别值得注意的是：列表是如何被反向显示的。在列表被修改之后，litr指向列表的末端（记住，当到达列表末端时，litr.hasNext( )方法返回false）。为了以反向遍历列表，程序继续使用litr，但这一次，程序检测它是否有前一个元素。只要它有前一个元素，该元素就被获得并被显示出来。

15.5 将用户定义的类存储于Collection中

为了简单，前面的例子在类集中存储内置的对象，如String或Integer。当然，类集并没有被限制为只能存储内置的对象。完全相反的是，类集的能力是它能存储任何类型的对象，包括你所创建的类的对象。例如，考虑下面的例子，在这个例子中使用LinkedList存储信箱地址。

```
// A simple mailing list example.
import java.util.*;

class Address {
  private String name;
  private String street;
  private String city;
  private String state;
  private String code;

  Address(String n, String s, String c,
          String st, String cd) {
    name = n;
    street = s;
    city = c;
    state = st;
    code = cd;
  }

  public String toString() {
    return name + "\n" + street + "\n" +
           city + " " + state + " " + code;
  }
```

```
}

class MailList {
  public static void main(String args[]) {
    LinkedList ml = new LinkedList();

    // add elements to the linked list
    ml.add(new Address("J.W. West", "11 Oak Ave",
                       "Urbana", "IL", "61801"));
    ml.add(new Address("Ralph Baker", "1142 Maple Lane",
                       "Mahomet", "IL", "61853"));
    ml.add(new Address("Tom Carlton", "867 Elm St",
                       "Champaign", "IL", "61820"));

    Iterator itr = ml.iterator();
    while(itr.hasNext()) {
      Object element = itr.next();
      System.out.println(element + "\n");
    }
    System.out.println();
  }
}
```

程序的输出如下所示：

```
J.W. West
11 Oak Ave
Urbana IL 61801

Ralph Baker
1142 Maple Lane
Mahomet IL 61853

Tom Carlton
867 Elm St
Champaign IL 61820
```

除了在类集中存储用户定义的类之外，关于上面程序的另一个重要的，值得注意的事情是它是非常短的。当考虑用50行代码建立一个能够实现存储，检索，以及处理信箱地址的链表时，类集框架的能力就变得显而易见了。正如大多数读者知道的那样，如果所有这些功能都必须用人工编写代码的话，程序将比现在的长好几倍。类集对许多不同的编程问题提供了现成的解决方案。每当情况出现时，就可以用它们。

15.6　处 理 映 射

正如在本章开始时所谈到的，除了类集，Java 2还在java.util中增加了映射。映射（map）是一个存储关键字和值的关联或者说是关键字/值对的对象。给定一个关键字，可以得到它的值。关键字和值都是对象。关键字必须是唯一的。但值是可以被复制的。有些映射可以接收null关键字和null值。而有的则不行。

15.6.1 映射接口

因为映射接口定义了映射的特征和本质，因此关于映射的讨论从这里开始。下面的接口支持映射：

接口	描述
Map	映射唯一关键字给值
Map.Entry	描述映射中的元素（关键字/值对）。这是Map的一个内部类
SortedMap	扩展Map以便关键字按升序保持

下面对每个接口依次进行讨论。

Map 接口

Map接口映射唯一关键字到值。关键字（key）是以后用于检索值的对象。给定一个关键字和一个值，可以存储这个值到一个Map对象中。当这个值被存储以后，就可以使用它的关键字来检索它。由Map说明的方法总结在表15-6中。当调用的映射中没有项存在时，其中的几种方法会引发一个NoSuchElementException异常。而当对象与映射中的元素不兼容时，引发一个ClassCastException异常。如果试图使用映射不允许使用的null对象时，则引发一个NullPointerException异常。当试图改变一个不允许修改的映射时，则引发一个UnsupportedOperationException异常。

表 15-6 由 Map 定义的方法

方法	描述
void clear( )	从调用映射中删除所有的关键字/值对
boolean containsKey(Object k)	如果调用映射中包含了作为关键字的k，则返回true；否则返回false
boolean containsValue(Object v)	如果映射中包含了作为值的v，则返回true；否则返回false
Set entrySet( )	返回包含了映射中的项的集合（Set）。该集合包含了类型Map.Entry的对象。这个方法为调用映射提供了一个集合“视图”
Boolean equals(Object obj)	如果obj是一个Map并包含相同的输入，则返回true；否则返回false
Object get(Object k)	返回与关键字k相关联的值
int hashCode( )	返回调用映射的散列码
boolean isEmpty( )	如果调用映射是空的，则返回true；否则返回false
Set keySet( )	返回一个包含调用映射中关键字的集合（Set）。这个方法为调用映射的关键字提供了一个集合“视图”
Object put(Object k, Object v)	将一个输入加入调用映射，覆盖原先与该关键字相关联的值。关键字和值分别为k和v。如果关键字已经不存在了，则返回null；否则，返回原先与关键字相关联的值
void putAll(Map m)	将所有来自m的输入加入调用映射
Object remove(Object k)	删除关键字等于k的输入

续表

方法	描述
int size( )	返回映射中关键字/值对的个数
Collection values( )	返回一个包含了映射中的值的类集。这个方法为映射中的值提供了一个类集“视图”

映射循环使用两个基本操作：get( )和put( )。使用put( )方法可以将一个指定了关键字和值的值加入映射。为了得到值，可以通过将关键字作为参数来调用get( )方法。调用返回该值。

正如前面谈到的，映射不是类集，但可以获得映射的类集“视图”。为了实现这种功能，可以使用entrySet( )方法，它返回一个包含了映射中元素的集合（Set）。为了得到关键字的类集“视图”，可以使用keySet( )方法。为了得到值的类集“视图”，可以使用values( )方法。类集“视图”是将映射集成到类集框架内的手段。

SortedMap 接口

SortedMap接口扩展了Map，它确保了各项按关键字升序排序。由SortedMap说明的方法总结在表15-7中。当调用映射中没有的项时，其中的几种方法引发一个NoSuchElementException异常。当对象与映射中的元素不兼容时，则引发一个ClassCastException异常。当试图使用映射不允许使用的null对象时，则引发一个NullPointerException异常。

表 15-7　由 SortedMap 定义的方法

方法	描述
Comparator comparator( )	返回调用排序映射的比较函数。如果调用映射使用的是自然顺序的话，则返回null
Object firstKey( )	返回调用映射的第一个关键字
SortedMap headMap(Object end)	返回一个排序映射，该映射包含了那些关键字小于end的映射输入
Object lastKey( )	返回调用映射的最后一个关键字
SortedMap subMap(Object start, Object end)	返回一个映射，该映射包含了那些关键字大于等于start同时小于end的输入
SortedMap tailMap(Object start)	返回一个映射，该映射包含了那些关键字大于等于start的输入

排序映射允许对子映射（换句话说，就是映射的子集）进行高效的处理。使用headMap( )，tailMap( )或subMap( )方法可以获得子映射。调用firstKey( )方法可以获得集合的第一个关键字。而调用lastKey( )方法可以获得集合的最后一个关键字。

Map.Entry 接口

Map.Entry接口使得可以操作映射的输入。回想由Map接口说明的entrySet( )方法，调用该方法返回一个包含映射输入的集合（Set）。这些集合元素的每一个都是一个Map.Entry

对象。表15-8总结了由该接口说明的方法。

表 15-8 由 Map.Entry 定义的方法

方法	描述
boolean equals(Object obj)	如果obj是一个关键字和值都与调用对象相等的Map.Entry，则返回true
Object getKey( )	返回该映射项的关键字
Object getValue( )	返回该映射项的值
int hashCode( )	返回该映射项的散列值
Object setValue(Object v)	将这个映射输入的值赋给v。如果v不是映射的正确类型，则引发一个ClassCastException 异常。如果v存在问题，则引发一个IllegalArgumentException异常。如果v是null而映射又不允许null关键字，则引发一个NullPointerException异常。如果映射不能被改变，则引发一个UnsupportedOperationException异常。

15.6.2 映射类

有几个类提供了映射接口的实现。可以被用做映射的类总结如下：

类	描述
AbstractMap	实现大多数的Map接口
HashMap	将AbstractMap扩展到使用散列表
TreeMap	将AbstractMap扩展到使用树
WeakHashMap	将AbstractMap扩展到使用弱关键字散列表

注意AbstractMap对三个具体的映射实现来说，是一个超类。WeakHashMap实现一个使用“弱关键字”的映射，它允许映射中的元素，当该映射的关键字不再被使用时，被放入回收站。关于这个类，在这里不做更深入的讨论。其他的类将在下面介绍。

HashMap 类

HashMap类使用散列表实现Map接口。这允许一些基本操作如get( )和put( )的运行时间保持恒定，即便对大型集合，也是这样的。

下面的构造函数定义为：

```
HashMap( )
HashMap(Map m)
HashMap(int capacity)
HashMap(int capacity, float fillRatio)
```

第一种形式构造一个默认的散列映射。第二种形式用m的元素初始化散列映射。第三种形式将散列映射的容量初始化为capacity。第四种形式用它的参数同时初始化散列映射的容量和填充比。容量和填充比的含义与前面介绍的HashSet中的容量和填充比相同。

HashMap实现Map并扩展AbstractMap。它本身并没有增加任何新的方法。

应该注意的是散列映射并不保证它的元素的顺序。因此，元素加入散列映射的顺序并

不一定是它们被迭代函数读出的顺序。

下面的程序举例说明了HashMap。它将名字映射到账目资产平衡表。注意集合“视图”是如何获得和被使用的。

```
import java.util.*;

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

    // Create a hash map
    HashMap hm = new HashMap();

    // Put elements to the map
    hm.put("John Doe", new Double(3434.34));
    hm.put("Tom Smith", new Double(123.22));
    hm.put("Jane Baker", new Double(1378.00));
    hm.put("Todd Hall", new Double(99.22));
    hm.put("Ralph Smith", new Double(-19.08));

    // Get a set of the entries
    Set set = hm.entrySet();

    // Get an iterator
    Iterator i = set.iterator();

    // Display elements
    while(i.hasNext()) {
      Map.Entry me = (Map.Entry)i.next();
      System.out.print(me.getKey() + ": ");
      System.out.println(me.getValue());
    }
    System.out.println();

    // Deposit 1000 into John Doe's account
    double balance =  ((Double)hm.get("John Doe")).doubleValue();
    hm.put("John Doe", new Double(balance + 1000));
    System.out.println("John Doe's new balance: " +
      hm.get("John Doe"));
  }
}
```

该程序的输出如下所示：

```
Todd Hall: 99.22
Ralph Smith: -19.08
John Doe: 3434.34
Jane Baker: 1378.0
Tom Smith: 123.22

John Doe’s current balance: 4434.34
```

程序开始创建一个散列映射，然后将名字的映射增加到平衡表中。接下来，映射的内容通过使用由调用函数entrySet( )而获得的集合“视图”而显示出来。关键字和值通过调用

由Map.Entry定义的getKey( )和getValue( )方法而显示。注意存款是如何被制成John Doe的账目的。put( )方法自动用新值替换与指定关键字相关联的原先的值。因此，在John Doe的账目被更新后，散列映射将仍然仅仅保留一个“John Doe”账目。

TreeMap 类

TreeMap类通过使用树实现Map接口。TreeMap提供了按排序顺序存储关键字/值对的有效手段，同时允许快速检索。应该注意的是，不像散列映射，树映射保证它的元素按照关键字升序排序。

下面的TreeMap构造函数定义为：

```
TreeMap( )
TreeMap(Comparator comp)
TreeMap(Map m)
TreeMap(SortedMap sm)
```

第一种形式构造一个空树的映射，该映射使用其关键字的自然顺序来排序。第二种形式构造一个空的基于树的映射，该映射通过使用Comparator comp来排序（比较函数Comparators将在本章后面进行讨论）。第三种形式用从m的输入初始化树映射，该映射使用关键字的自然顺序来排序。第四种形式用从sm的输入来初始化一个树映射，该映射将按与sm相同的顺序来排序。

TreeMap实现SortedMap并且扩展AbstractMap。而它本身并没有另外定义其他方法。

下面的程序重新使前面的例子运转，以便在其中使用TreeMap：

```
import java.util.*;

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

    // Create a tree map
    TreeMap tm = new TreeMap();

    // Put elements to the map
    tm.put("John Doe", new Double(3434.34));
    tm.put("Tom Smith", new Double(123.22));
    tm.put("Jane Baker", new Double(1378.00));
    tm.put("Todd Hall", new Double(99.22));
    tm.put("Ralph Smith", new Double(-19.08));

    // Get a set of the entries
    Set set = tm.entrySet();

    // Get an iterator
    Iterator i = set.iterator();

    // Display elements
    while(i.hasNext()) {
      Map.Entry me = (Map.Entry)i.next();
      System.out.print(me.getKey() + ": ");
      System.out.println(me.getValue());
```

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

    // Deposit 1000 into John Doe's account
    double balance =  ((Double)tm.get("John Doe")).doubleValue();
    tm.put("John Doe", new Double(balance + 1000));
    System.out.println("John Doe's new balance: " +
      tm.get("John Doe"));
  }
}
```

下面是该程序的输出结果：

```
Jane Baker: 1378.0
John Doe: 3434.34
Ralph Smith: -19.08
Todd Hall: 99.22
Tom Smith: 123.22

John Doe’s current balance: 4434.34
```

注意对关键字进行了排序。然而，在这种情况下，它们用名字而不是用姓进行了排序。可以通过在创建映射时，指定一个比较函数来改变这种排序。在下一节将介绍如何做。

15.7　比 较 函 数

TreeSet和TreeMap都按排序顺序存储元素。然而，精确定义采用何种“排序顺序”的是比较函数。通常在默认的情况下，这些类通过使用被Java称之为“自然顺序”的顺序存储它们的元素，而这种顺序通常也是你所需要的（A在B的前面，1在2的前面，等等）。如果需要用不同的方法对元素进行排序，可以在构造集合或映射时，指定一个Comparator对象。这样做为你提供了一种精确控制如何将元素储存到排序类集和映射中的能力。

Comparator接口定义了两个方法：compare( )和equals( )。这里给出的compare( )方法按顺序比较了两个元素：

```
int compare(Object obj1, Object obj2)
```

obj1和obj2是被比较的两个对象。当两个对象相等时，该方法返回0；当obj1大于obj2时，返回一个正值；否则，返回一个负值。如果用于比较的对象的类型不兼容的话，该方法引发一个ClassCastException异常。通过覆盖compare( )，可以改变对象排序的方式。例如，通过创建一个颠倒比较输出的比较函数，可以实现按逆向排序。

这里给出的equals( )方法，测试一个对象是否与调用比较函数相等：

```
boolean equals(Object obj)
```

obj是被用来进行相等测试的对象。如果obj和调用对象都是Comparator的对象并且使用相同的排序。该方法返回true。否则返回false。重载equals( )方法是没有必要的，大多数简单的比较函数都不这样做。

15.7.1 使用比较函数

下面是一个说明定制的比较函数能力的例子。该例子实现compare( )方法以便它按正常顺序的逆向进行操作。因此，它使得一个树集合按逆向的顺序进行存储。

```
// Use a custom comparator.
import java.util.*;

// A reverse comparator for strings.
class MyComp implements Comparator {
  public int compare(Object a, Object b) {
    String aStr, bStr;

    aStr = (String) a;
    bStr = (String) b;

    // reverse the comparison
    return bStr.compareTo(aStr);
  }

  // no need to override equals
}

class CompDemo {
  public static void main(String args[]) {
    // Create a tree set
    TreeSet ts = new TreeSet(new MyComp());

    // Add elements to the tree set
    ts.add("C");
    ts.add("A");
    ts.add("B");
    ts.add("E");
    ts.add("F");
    ts.add("D");

    // Get an iterator
    Iterator i = ts.iterator();

    // Display elements
    while(i.hasNext()) {
      Object element = i.next();
      System.out.print(element + " ");
    }
    System.out.println();
  }
}
```

正如下面的输出所示，树按照逆向顺序进行存储：

```
F E D C B A
```

仔细观察实现Comparator并覆盖compare( )方法的MyComp类（正如前面所解释的那样，覆盖equals( )方法既不是必须的，也不是常用的）。在compare( )方法内部，String方法

compareTo( )比较两个字符串。然而由bStr——不是aStr——调用compareTo( )方法，这导致比较的结果被逆向。

对应一个更实际的例子，下面的例子是用TreeMap程序实现前面介绍的存储账目资产平衡表例子的程序。在前面介绍的程序中，账目是按名进行排序的，但程序是以按照名字进行排序开始的。下面的程序按姓对账目进行排序。为了实现这种功能，程序使用了比较函数来比较每一个账目下姓的排序。得到的映射是按姓进行排序的。

```
// Use a comparator to sort accounts by last name.
import java.util.*;

// Compare last whole words in two strings.
class TComp implements Comparator {
  public int compare(Object a, Object b) {
    int i, j, k;
    String aStr, bStr;

    aStr = (String) a;
    bStr = (String) b;

    // find index of beginning of last name
    i = aStr.lastIndexOf(' ');
    j = bStr.lastIndexOf(' ');

    k = aStr.substring(i).compareTo(bStr.substring(j));
    if(k==0) // last names match, check entire name
      return aStr.compareTo(bStr);
    else
      return k;
  }

  // no need to override equals
}

class TreeMapDemo2 {
  public static void main(String args[]) {
    // Create a tree map
    TreeMap tm = new TreeMap(new TComp());

    // Put elements to the map
    tm.put("John Doe", new Double(3434.34));
    tm.put("Tom Smith", new Double(123.22));
    tm.put("Jane Baker", new Double(1378.00));
    tm.put("Todd Hall", new Double(99.22));
    tm.put("Ralph Smith", new Double(-19.08));

    // Get a set of the entries
    Set set = tm.entrySet();

    // Get an iterator
    Iterator itr = set.iterator();

    // Display elements
    while(itr.hasNext()) {
```

```
      Map.Entry me = (Map.Entry)itr.next();
      System.out.print(me.getKey() + ": ");
      System.out.println(me.getValue());
    }
    System.out.println();

    // Deposit 1000 into John Doe's account
    double balance =  ((Double)tm.get("John Doe")).doubleValue();
    tm.put("John Doe", new Double(balance + 1000));
    System.out.println("John Doe's new balance: " +
      tm.get("John Doe"));
  }
}
```

这里是程序的输出结果，注意此时的账目是按姓进行排序的：

```
Jane Baker: 1378.0
John Doe: 3434.34
Todd Hall: 99.22
Ralph Smith: -19.08
Tom Smith: 123.22

John Doe's new balance: 4434.34
```

比较函数类TComp比较两个包含姓和名的字符串。它首先比较姓。具体是这样做的，它首先寻找每一个字符串中最后一个空格的下标，然后比较从这个位置开始的每一个元素的子字符串。当两个字符串中姓完全相等时，它再比较两个名。这样就形成了一个先按姓进行排序，在姓相同的情况下，再按名字进行排序的树型映射。通过程序的输出中Ralph Smith出现在Tom Smith之前的结果可以看到这一点。

15.8 类 集 算 法

类集框架定义了几种能用于类集和映射的算法。在Collections类中，这些算法被定义为静态方法。表15-9中列出了这些算法。当试图比较不兼容的类型时，其中的一些算法引发一个ClassCastException异常；而当试图改变一个不可改变的类集时，则引发一个UnsupportedOperationException异常。

表 15-9 由 Collections 定义的算法

方法	描述
static int binarySearch(List list, Object value,Comparator c)	按照c的次序在list中搜寻value。如果value在list内，则返回value在list的位置。如果在list中没有发现value，则返回-1
static int binarySearch(List list, Object value)	在list中搜寻value，列表（list）必须被排序。如果value在list内，则返回value的位置。如果在list中没有发现value，则返回-1
static void copy(List list1, List list2)	将list2中的元素复制给list1

续表

方法	描述
static Enumeration enumeration(Collection c)	返回c的一个枚举（参看本章后面的“枚举接口”）。
static void fill(List list, Object obj)	将obj赋给list中的每一个元素
Static Object max(Collection c,Comparator comp)	返回由comp确定的c中的最大元素
static Object max(Collection c)	返回按自然顺序确定的c中的最大元素。类集不必被排序
static Object min(Collection c,Comparator comp)	返回由comp确定的c中的最小元素。类集不必被排序
static Object min(Collection c)	返回按自然顺序确定的c中的最小元素
static List nCopies(int num, Object obj)	返回包含在不可改变的列表中的obj的num个拷贝。num必须大于等于0
static void reverse(List list)	将list中的序列逆向
static Comparator reverseOrder( )	返回一个逆向比较函数（即将两个元素比较的结果进行逆向的比较函数）
static void shuffle(List list, Random r)	用r作为随机数的源，对list中的元素进行混淆（也即随机化）
static void shuffle(List list)	对list中的元素进行混淆（也即随机化）
static Set singleton(Object obj)	返回一个不可改变的集合obj。这是一个实现将单个对象变成集合的简单办法
static List singletonList(Object obj)	返回一个不可改变的列表obj。这是一个实现将单个对象变成列表的简单办法（在Java 2的1.3版中新增加的）
static Map singletonMap(Object k, Object v)	返回一个不可改变的关键字/值对映射k/v。这是一个实现将单个关键字/值对变成映射的简单办法（在Java 2的1.3版中新增加的）
static void sort(List list, Comparator comp)	按comp对list中的元素进行排序
static void sort(List list)	按自然顺序对list中的元素进行排序
static Collection synchronizedCollection(Collection c)	返回一个被c支持的安全线程类集
static List synchronizedList(List list)	返回一个被list支持的安全线程列表
static Map synchronizedMap(Map m)	返回一个被m支持的安全线程映射
static Set synchronizedSet(Set s)	返回一个被s支持的安全线程集合
static SortedMap synchronizedSortedMap(SortedMap sm)	返回一个被sm支持的安全线程排序集合
static SortedSet synchronizedSortedSet(SortedSet ss)	返回一个被ss支持的安全线程集合
static Collection unmodifiableCollection(Collection c)	返回一个被c支持的不可变类集
Static List unmodifiableList(List list)	返回一个被list支持的不可变列表

续表

方法	描述
static Map unmodifiableMap(Map m)	返回一个被m支持的不可变映射
static Set unmodifiableSet(Set s)	返回一个被s支持的不可变集合
Static SortedMap unmodifiable SortedMap(SortedMap sm)	返回一个被sm支持的不可变排序映射
static SortedSet unmodifiableSortedSet(SortedSet ss)	返回一个被ss支持的不可变排序集合

注意其中的几种方法，如synchronizedList( )和synchronizedSet( )被用来获得各种类集的同步（安全线程）拷贝。正如前面解释的那样，没有任何一个标准类集实现是同步的。必须使用同步算法来为其提供同步。另一种观点：同步类集的迭代函数必须在synchronized块内使用。

以unmodifiable开头的一组方法返回不能被改变的各种类集“视图”。这些方法当将一些进程对类集设为只读形式时很有用的。

Collections定义了三个静态变量：EMPTY_SET，EMPTY_LIST和EMPTY_MAP。它们都是不可改变的。EMPTY_MAP是在Java 2的1.3版中新增加的。

下面的程序说明了其中的一些算法。该程序创建和初始化了一个链表。reverseOrder( )方法返回一个对Integer对象的比较进行逆向的Comparator函数。列表中的元素按照这个比较函数进行排序并被显示出来。接下来，调用shuffle( )方法对列表进行随机排列。然后显示列表的最大和最小值。

```
// Demonstrate various algorithms.
import java.util.*;

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

    // Create and initialize linked list
    LinkedList ll = new LinkedList();
    ll.add(new Integer(-8));
    ll.add(new Integer(20));
    ll.add(new Integer(-20));
    ll.add(new Integer(8));

    // Create a reverse order comparator
    Comparator r = Collections.reverseOrder();

    // Sort list by using the comparator
    Collections.sort(ll, r);

    // Get iterator
    Iterator li = ll.iterator();

    System.out.print("List sorted in reverse: ");
    while(li.hasNext())
      System.out.print(li.next() + " ");
```

```
    System.out.println();

    Collections.shuffle(ll);

    // display randomized list
    li = ll.iterator();
    System.out.print("List shuffled: ");
    while(li.hasNext())
      System.out.print(li.next() + " ");
    System.out.println();

    System.out.println("Minimum: " + Collections.min(ll));
    System.out.println("Maximum: " + Collections.max(ll));
  }
}
```

该程序的输出如下所示：

```
List sorted in reverse: 20 8 -8 -20
List shuffled: 20 -20 8 -8
Minimum: -20
Maximum: 20
```

注意min( )和max( )方法是在列表被混淆之后，对其进行操作的。两者在运行时，都不需要排序的列表。

15.9　Arrays （数组）

Java 2在java.util中新增加了一个叫做Arrays的类。这个类提供了各种在进行数组运算时很有用的方法。尽管这些方法在技术上不属于类集框架，但它们提供了跨越类集和数组的桥梁。在这一节中，分析由Arrays定义的每一种方法。

asList( )方法返回一个被指定数组支持的List。换句话说，列表和数组访问的是同一个单元。它具有如下的形式：

```
static List asList(Object[ ] array)
```

这里array是包含了数据的数组。

binarySearch( )方法使用二进制搜索寻找指定的值。该方法必须应用于排序数组。它具有如下的形式：

```
static int binarySearch(byte[ ] array, byte value)
static int binarySearch(char[ ] array, char value)
static int binarySearch(double[ ] array, double value)
static int binarySearch(float[ ] array, float value)
static int binarySearch(int[ ] array, int value)
static int binarySearch(long[ ] array, long value)
static int binarySearch(short[ ] array, short value)
static int binarySearch(Object[ ] array, Object value)
static int binarySearch(Object[ ] array, Object value, Comparator c)
```

这里，array是被搜索的数组，而value是被查找的值。当array中包含的元素是不可比较的（例如Double和StringBuffer）或者当value与array中的类型不兼容时，后两种形式引发一个ClassCastException异常。在最后一种形式中，比较函数（Comparator）c用于确定array中的元素的顺序。在所有的形式中，如果array中含有value，则返回该元素的下标。否则，返回一个负值。

当两个数组相等时，equals( )方法返回true；否则返回false。equals( )方法具有下面的一些形式：

```
static boolean equals(boolean array1[ ], boolean array2[ ])
static boolean equals(byte array1[ ], byte array2[ ])
static boolean equals(char array1[ ], char array2[ ])
static boolean equals(double array1[ ], double array2[ ])
static boolean equals(float array1[ ], float array2[ ])
static boolean equals(int array1[ ], int array2[ ])
static boolean equals(long array1[ ], long array2[ ])
static boolean equals(short array1[ ], short array2[ ])
static boolean equals(Object array1[ ], Object array2[ ])
```

这里array1和array2是两个用来比较看是否相等的数组。

fill( )方法将一个值赋给数组中的所有元素。换句话说，它用一个指定的值填充数组。fill( )方法有两种形式。第一种形式具有下面的一些形式，填充整个数组：

```
static void fill(boolean array[ ], boolean value)
static void fill(byte array[ ], byte value)
static void fill(char array[ ], char value)
static void fill(double array[ ], double value)
static void fill(float array[ ], float value)
static void fill(int array[ ], int value)
static void fill(long array[ ], long value)
static void fill(short array[ ], short value)
static void fill(Object array[ ], Object value)
```

这里value被赋给数组array中的每一个元素。

fill( )方法的第二种形式将一个值赋给数组的一个子集。它的几种形式如下：

```
static void fill(boolean array[ ], int start, int end, boolean value)
static void fill(byte array[ ], int start, int end, byte value)
static void fill(char array[ ], int start, int end, char value)
static void fill(double array[ ], int start, int end, double value)
static void fill(float array[ ], int start, int end, float value)
static void fill(int array[ ], int start, int end, int value)
static void fill(long array[ ], int start, int end, long value)
static void fill(short array[ ], int start, int end, short value)
static void fill(Object array[ ], int start, int end, Object value)
```

这里，value是赋给数组array中从start开始到end–1结束的子集的值。这些方法当start大于end时，都能引发一个IllegalArgumentException异常；而当start或end出界时，都能引发一个ArrayIndexOutOfBoundsException异常。

sort( )方法对数组进行排序，以便数组能够按升序进行排列。sort( )方法有两种形式。下面给出的第一种形式对整个数组进行排序：

```
static void sort(byte array[ ])
static void sort(char array[ ])
static void sort(double array[ ])
static void sort(float array[ ])
static void sort(int array[ ])
static void sort(long array[ ])
static void sort(short array[ ])
static void sort(Object array[ ])
static void sort(Object array[ ], Comparator c)
```

这里，array是被排序的数组。在最后的一种形式中，c是一个用来规定array中元素顺序的比较函数(Comparator)。当用于排序的数组中的元素不可比较时，这些对Object的数组进行排序的sort( )方法将引发一个ClassCastException异常。

sort( )方法的第二种形式允许在一个数组内，指定一个想要进行排序的范围。它的具体形式如下：

```
static void sort(byte array[ ], int start, int end)
static void sort(char array[ ], int start, int end)
static void sort(double array[ ], int start, int end)
static void sort(float array[ ], int start, int end)
static void sort(int array[ ], int start, int end)
static void sort(long array[ ], int start, int end)
static void sort(short array[ ], int start, int end)
static void sort(Object array[ ], int start, int end)
static void sort(Object array[ ], int start, int end, Comparator c)
```

这里，数组中想要进行排序的范围从start到end–1。在最后一种形式中，c是一个用来规定array中元素顺序的Comparator。如果start大于end，所有这些方法都能引发一个IllegalArgumentException异常；而当start或end出界时，又都能引发一个ArrayIndexOutOfBoundsException异常。当用于排序的数组中的元素不可比较时，最后两种形式也能引发一个ClassCastException异常。

下面的程序举例说明了如何使用Arrays类中的一些方法：

```
// Demonstrate Arrays
import java.util.*;

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

    // allocate and initialize array
    int array[] = new int[10];
    for(int i = 0; i < 10; i++)
      array[i] = -3 * i;

    // display, sort, display
    System.out.print("Original contents: ");
    display(array);
    Arrays.sort(array);
    System.out.print("Sorted: ");
    display(array);
```

```
    // fill and display
    Arrays.fill(array, 2, 6, -1);
    System.out.print("After fill(): ");
    display(array);

    // sort and display
    Arrays.sort(array);
    System.out.print("After sorting again: ");
    display(array);

    // binary search for -9
    System.out.print("The value -9 is at location ");
    int index =
      Arrays.binarySearch(array, -9);
    System.out.println(index);
  }

  static void display(int array[]) {
    for(int i = 0; i < array.length; i++)
      System.out.print(array[i] + " ");
    System.out.println("");
  }
}
```

下面是该程序的输出结果：

```
Original contents: 0 -3 -6 -9 -12 -15 -18 -21 -24 -27
Sorted: -27 -24 -21 -18 -15 -12 -9 -6 -3 0
After fill(): -27 -24 -1 -1 -1 -1 -9 -6 -3 0
After sorting again: -27 -24 -9 -6 -3 -1 -1 -1 -1 0
The value -9 is at location 2
```

15.10 从以前版本遗留下来的类和接口

正如本章开始时介绍的那样，java.util的最初版本中不包括类集框架。取而代之，它定义了几个类和接口提供专门的方法用于存储对象。随着在Java 2中引入类集，有几种最初的类被重新设计成支持类集接口。因此它们与框架完全兼容。尽管实际上没有类被摈弃，但其中某些仍被认为是过时的。当然，在那些重复从以前版本遗留下来的类的功能性的地方，通常都愿意用类集编写新的代码程序。一般地，对从以前版本遗留下来的类的支持是因为仍然存在大量使用它们的基本代码。包括现在仍在被Java 2的应用编程接口（API）使用的程序。

另一点，没有一个类集类是同步的。但是所有的从以前版本遗留下来的类都是同步的。这一区别在有些情况下是很重要的。当然，通过使用由Collections提供的算法也很容易实现类集同步。

由java.util定义的从以前版本遗留下来的类说明如下：

Dictionary	Hashtable	Properties	Stack	Vector

有一个枚举（Enumeration）接口是从以前版本遗留下来。在下面依次介绍Enumeration和每一种从以前版本遗留下来的类。

15.10.1　Enumeration接口

Enumeration接口定义了可以对一个对象的类集中的元素进行枚举（一次获得一个）的方法。这个接口尽管没有被摈弃，但已经被Iterator所替代。Enumeration对新程序来说是过时的。然而它仍被几种从以前版本遗留下来的类（例如Vector和Properties）所定义的方法使用，被几种其他的API类所使用以及被目前广泛使用的应用程序所使用。

Enumeration指定下面的两个方法：

```
boolean hasMoreElements( )
Object nextElement( )
```

执行后，当仍有更多的元素可提取时，hasMoreElements( )方法一定返回true。当所有元素都被枚举了，则返回false。nextElement( )方法将枚举中的下一个对象做为一个类属Object的引用而返回。也就是每次调用nextElement( )方法获得枚举中的下一个对象。调用例程必须将那个对象置为包含在枚举内的对象类型。

15.10.2　Vector

Vector实现动态数组。这与ArrayList相似，但两者不同的是：Vector是同步的，并且它包含了许多不属于类集框架的从以前版本遗留下来的方法。随着Java 2的公布，Vector被重新设计来扩展AbstractList和实现List接口，因此现在它与类集是完全兼容的。

这里是Vector的构造函数：

```
Vector( )
Vector(int size)
Vector(int size, int incr)
Vector(Collection c)
```

第一种形式创建一个原始大小为10的默认矢量。第二种形式创建一个其原始容量由size指定的矢量。第三种形式创建一个其原始容量由size指定，并且它的增量由incr指定的矢量。增量指定了矢量每次允许向上改变大小的元素的个数。第四种形式创建一个包含了类集c中元素的矢量。这个构造函数是在Java 2中新增加的。

所有的矢量开始都有一个原始的容量。在这个原始容量达到以后，下一次再试图向矢量中存储对象时，矢量自动为那个对象分配空间同时为别的对象增加额外的空间。通过分配超过需要的内存，矢量减小了可能产生的分配的次数。这种次数的减少是很重要的，因为分配内存是很花时间的。在每次再分配中，分配的额外空间的总数由在创建矢量时指定的增量来确定。如果没有指定增量，在每个分配周期，矢量的大小增一倍。

Vector定义了下面的保护数据成员：

```
int capacityIncrement;
int elementCount;
Object elementData[ ];
```

增量值被存储在capacityIncrement中。矢量中的当前元素的个数被存储在elementCount中。保存矢量的数组被存储在elementData中。

除了由List定义的类集方法之外，Vector还定义了几个从以前版本遗留下来的方法，这些方法列在表15-10中。

表 15-10 由 Vector 定义的方法

方法	描述
final void addElement(Object element)	将由element指定的对象加入矢量
int capacity( )	返回矢量的容量
Object clone( )	返回调用矢量的一个拷贝
Boolean contains(Object element)	如果element被包含在矢量中，则返回true；如果不包含于其中，则返回false
void copyInto(Object array[ ])	将包含在调用矢量中的元素复制到由array指定的数组中
Object elementAt(int index)	返回由index指定位置的元素
Enumeration elements( )	返回矢量中元素的一个枚举
Object firstElement( )	返回矢量的第一个元素
int indexOf(Object element)	返回element首次出现的位置下标。如果对象不在矢量中，则返回-1
int indexOf(Object element, int start)	返回element在矢量中在start及其之后第一次出现的位置下标。如果该对象不属于矢量的这一部分，则返回-1
void insertElementAt(Object element, int index)	在矢量中，在由index指定的位置处加入element
boolean isEmpty( )	如果矢量是空的，则返回true。如果它包含了一个或更多个元素，则返回false
Object lastElement( )	返回矢量中的最后一个元素
int lastIndexOf(Object element)	返回element在矢量中最后一次出现的位置下标。如果对象不包含在矢量中，则返回-1
int lastIndexOf(Object element,int start)	返回element在矢量中，在start之前最后一次出现的位置下标。如果该对象不属于矢量的这一部分，则返回-1
void removeAllElements( )	清空矢量，在这个方法执行以后，矢量的大小为0
boolean removeElement(Object element)	从矢量中删除element。对于指定的对象，矢量中如果有其多个实例，则其中第一个实例被删除。如果成功删除，则返回true；如果没有发现对象，则返回false
void removeElementAt(int index)	删除由index指定位置处的元素
void setElementAt(Object element, int index)	将由index指定的位置分配给element
void setSize(int size)	将矢量中元素的个数设为size。如果新的长度小于老的长度，元素将丢失；如果新的长度大于老的长度，则在其后增加null元素
int size( )	返回矢量中当前元素的个数

续表

方法	描述
String toString( )	返回矢量的字符串等价形式
void trimToSize( )	将矢量的容量设为与其当前拥有的元素的个数相等

因为Vector实现List，所以可以像使用ArrayList的一个实例那样使用矢量。也可以使用它的从以前版本遗留下来的方法来操作它。例如，在后面实例化Vector，可以通过调用addElement( )方法而为其增加一个元素。调用elementAt( )方法可以获得指定位置处的元素。调用firstElement( )方法可以得到矢量的第一个元素。调用lastElement( )方法可以检索到矢量的最后一个元素。使用indexOf( )和lastIndexOf( )方法可以获得元素的下标。调用removeElement( )或removeElementAt( )方法可以删除元素。

下面的程序使用矢量存储不同类型的数值对象。程序说明了几种由Vector定义的从以前版本遗留下来的方法，同时它也说明了枚举（Enumeration）接口。

```
// Demonstrate various Vector operations.
import java.util.*;

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

    // initial size is 3, increment is 2
    Vector v = new Vector(3, 2);

    System.out.println("Initial size: " + v.size());
    System.out.println("Initial capacity: " +
                       v.capacity());

    v.addElement(new Integer(1));
    v.addElement(new Integer(2));
    v.addElement(new Integer(3));
    v.addElement(new Integer(4));

    System.out.println("Capacity after four additions: " +
                       v.capacity());
    v.addElement(new Double(5.45));
    System.out.println("Current capacity: " +
                       v.capacity());
    v.addElement(new Double(6.08));
    v.addElement(new Integer(7));

    System.out.println("Current capacity: " +
                       v.capacity());
    v.addElement(new Float(9.4));
    v.addElement(new Integer(10));

    System.out.println("Current capacity: " +
                       v.capacity());
    v.addElement(new Integer(11));
    v.addElement(new Integer(12));
```

```
    System.out.println("First element: " +
                       (Integer)v.firstElement());
    System.out.println("Last element: " +
                       (Integer)v.lastElement());

    if(v.contains(new Integer(3)))
      System.out.println("Vector contains 3.");

    // enumerate the elements in the vector.
    Enumeration vEnum = v.elements();

    System.out.println("\nElements in vector:");
    while(vEnum.hasMoreElements())
      System.out.print(vEnum.nextElement() + " ");
    System.out.println();
  }
}
```

该程序的输出如下所示：

```
Initial size: 0
Initial capacity: 3
Capacity after four additions: 5
Current capacity: 5
Current capacity: 7
Current capacity: 9
First element: 1
Last element: 12
Vector contains 3.

Elements in vector:
1 2 3 4 5.45 6.08 7 9.4 10 11 12
```

随着Java 2的公布，Vector增加了对迭代函数的支持。现在可以使用迭代函数来替代枚举去遍历对象（正如前面的程序所做的那样）。例如，下面的基于迭代函数的程序代码可以被替换到上面的程序中：

```
// use an iterator to display contents
Iterator vItr = v.iterator();

System.out.println("\nElements in vector:");
while(vItr.hasNext())
  System.out.print(vItr.next() + " ");
System.out.println();
```

因为建议不要使编写枚举新的程序代码，所以通常可以使用迭代函数来对矢量的内容进行枚举。当然，业已存在的大量的老程序采用了枚举。不过幸运的是，枚举和迭代函数的工作方式几乎相同。

15.10.3 Stack

Stack是Vector的一个子类，它实现标准的后进先出堆栈。Stack仅仅定义了创建空堆栈

的默认构造函数。Stack包括了由Vector定义的所有方法，同时增加了几种它自己定义的方法，具体总结在表15-11中。

表 15-11　由 Stack 定义的方法

方法	描述
boolean empty( )	如果堆栈是空的，则返回true，当堆栈包含有元素时，返回false
Object peek( )	返回位于栈顶的元素，但是并不在堆栈中删除它
Object pop( )	返回位于栈顶的元素，并在进程中删除它
Object push(Object element)	将element压入堆栈，同时也返回element
int search(Object element)	在堆栈中搜索element，如果发现了，则返回它相对于栈顶的偏移量。否则，返回-1

调用push( )方法可将一个对象压入栈顶。调用pop( )方法可以删除和返回栈顶的元素。当调用堆栈是空的时，如果调用pop( )方法，将引发一个EmptyStackException异常。调用peek( )方法返回但不删除栈顶的对象。调用empty( )方法，当堆栈中没有元素时，返回true。search( )方法确定一个对象是否存在于堆栈，并且返回将其指向栈顶所需的弹出次数。下面是一个创建堆栈的例子，在例子中，将几个整型（Integer）对象压入堆栈，然后再将它们弹出。

```
// Demonstrate the Stack class.
import java.util.*;

class StackDemo {
  static void showpush(Stack st, int a) {
    st.push(new Integer(a));
    System.out.println("push(" + a + ")");
    System.out.println("stack: " + st);
  }

  static void showpop(Stack st) {
    System.out.print("pop -> ");
    Integer a = (Integer) st.pop();
    System.out.println(a);
    System.out.println("stack: " + st);
  }

  public static void main(String args[]) {
    Stack st = new Stack();

    System.out.println("stack: " + st);
    showpush(st, 42);
    showpush(st, 66);
    showpush(st, 99);
    showpop(st);
    showpop(st);
    showpop(st);
    try {
      showpop(st);
```

```
      } catch (EmptyStackException e) {
        System.out.println("empty stack");
      }
    }
  }
```

下面是由该程序产生的输出。注意对于EmptyStackException的异常处理程序是如何被捕获以便于能够从容地处理堆栈的下溢：

```
stack: [ ]
push(42)
stack: [42]
push(66)
stack: [42, 66]
push(99)
stack: [42, 66, 99]
pop -> 99
stack: [42, 66]
pop -> 66
stack: [42]
pop -> 42
stack: [ ]
pop -> empty stack
```

15.10.4 Dictionary

字典（Dictionary）是一个表示关键字/值存储库的抽象类，同时它的操作也很像映射（Map）。给定一个关键字和值，可以将值存储到字典（Dictionary）对象中。一旦这个值被存储了，就能够用它的关键字来检索它。因此，与映射一样，字典可以被当做关键字/值对列表来考虑。尽管在Java 2中并没有摈弃字典（Dictionary），由于它被映射（Map）所取代，从而被认为是过时的。然而由于目前Dictionary被广泛地使用，因此这里仍对它进行详细的讨论。

由Dictionary定义的抽象方法在表15-12中列出。

表 15-12 由 Dictionary 定义的抽象方法

方法	描述
Enumeration elements( )	返回对包含在字典中的值的枚举
Object get(Object key)	返回一个包含与key相连的值的对象。如果key不在字典中，则返回一个空对象
boolean isEmpty( )	如果字典是空的，则返回true；如果字典中至少包含一个关键字，则返回false
Enumeration keys( )	返回包含在字典中的关键字的枚举
Object put(Object key, Object value)	将一个关键字和它的值插入字典中。如果key已经不在字典中了，则返回null；如果key已经在字典中了，则返回与key相关联的前一个值

续表

方法	描述
Object remove(Object key)	删除key和它的值。返回与key相关联的值。如果key不在字典中，则返回null
int size( )	返回字典中的项数

使用put( )方法在字典中增加关键字和值。使用get( )方法检索给定关键字的值。当分别使用keys( )和elements( )方法进行枚举（Enumeration）时，关键字和值可以分别逐个地返回。size( )方法返回存储在字典中的关键字/值对的个数。当字典是空的时候，isEmpty( )返回true。使用remove( )方法可以删除关键字/值对。

注意：Dictionary类是过时的。应该执行Map接口去获得关键字/值存储的功能。

15.10.5　Hashtable

散列表（Hashtable）是原始java.util中的一部分同时也是Dictionary的一个具体实现。然而，Java 2重新设计了散列表（Hashtable）以便它也能实现映射（Map）接口。因此现在Hashtable也被集成到类集框架中。它与HashMap相似，但它是同步的。

和HashMap一样，Hashtable将关键字/值对存储到散列表中。使用Hashtable时，指定一个对象作为关键字，同时指定与该关键字相关联的值。接着该关键字被散列，而把得到的散列值作为存储在表中的值的下标。

散列表仅仅可以存储重载由Object定义的hashCode( )和equals( )方法的对象。hashCode( )方法计算和返回对象的散列码。当然，equals( )方法比较两个对象。幸运的是，许多Java内置的类已经实现了hashCode( )方法。例如，大多数常见的Hashtable类型使用字符串（String）对象作为关键字。String实现hashCode( )和equals( )方法。

Hashtable的构造函数如下所示：

```
Hashtable( )
Hashtable(int size)
Hashtable(int size, float fillRatio)
Hashtable(Map m)
```

第一种形式是默认的构造函数。第二种形式创建一个散列表，该散列表具有由size指定的原始大小。第三种形式创建一个散列表，该散列表具有由size指定的原始大小和由fillRatio指定的填充比。填充比必须介于0.0和1.0之间，它决定了在散列表向上调整大小之前散列表的充满度。具体地说，当元素的个数大于散列表的容量乘以它的填充比时，散列表被扩展。如果没有指定填充比，默认使用0.75。最后，第四种形式创建一个散列表，该散列表用m中的元素初始化。散列表的容量被设为m中元素的个数的两倍。默认的填充因子设为0.75。第四种构造函数是在Java 2中新增加的。

除了Hashtable目前实现的，由Map接口定义的方法之外，Hashtable定义的从以前版本遗留下来的方法列在表15-13中。

表 15-13 由 Hashtable 定义的从以前版本遗留下来的方法

方法	描述
void clear( )	复位并清空散列表
Object clone( )	返回调用对象的复制
boolean contains(Object value)	如果一些值与存在于散列表中的value相等的话，则返回true；如果这个值不存在，则返回false
boolean containsKey(Object key)	如果一些关键字与存在于散列表中的key相等的话，则返回true；如果这个关键字不存在，则返回false
boolean containsValue(Object value)	如果一些值与散列表中存在的value相等的话，返回true；如果这个值没有找到，则返回false（是一种为了保持一致性而在Java 2中新增加的非Map方法）
Enumeration elements( )	返回包含在散列表中的值的枚举
Object get(Object key)	返回包含与key相关联的值的对象。如果key不在散列表中，则返回一个空对象
boolean isEmpty( )	如果散列表是空的，则返回true；如果散列表中至少包含一个关键字，则返回false
Enumeration keys( )	返回包含在散列表中的关键字的枚举
Object put(Object key, Object value)	将关键字和值插入散列表中。如果key已经不在散列表中，返回null。如果key已经存在于散列表中，则返回与key相连的前一个值
void rehash( )	增大散列表的大小并且对其关键字进行再散列。
Object remove(Object key)	删除key及其对应的值。返回与key相关联的值。如果key不在散列表中，则返回一个空对象
int size( )	返回散列表中的项数
String toString( )	返回散列表的等价字符串形式

下面的例子重写前面介绍的关于银行账目的程序。在重写的程序中，使用Hashtable储存银行存款人的名字和他们当前的资产平衡表：

```
// Demonstrate a Hashtable
import java.util.*;
class HTDemo {
  public static void main(String args[]) {
    Hashtable balance = new Hashtable();
    Enumeration names;
    String str;
    double bal;

    balance.put("John Doe", new Double(3434.34));
    balance.put("Tom Smith", new Double(123.22));
    balance.put("Jane Baker", new Double(1378.00));
    balance.put("Todd Hall", new Double(99.22));
    balance.put("Ralph Smith", new Double(-19.08));
```

```
    // Show all balances in hash table.
    names = balance.keys();
    while(names.hasMoreElements()) {
      str = (String) names.nextElement();
      System.out.println(str + ": " +
                         balance.get(str));
    }

    System.out.println();

    // Deposit 1,000 into John Doe's account
    bal = ((Double)balance.get("John Doe")).doubleValue();
    balance.put("John Doe", new Double(bal+1000));
    System.out.println("John Doe's new balance: " +
                       balance.get("John Doe"));
  }
}
```

该程序的输出如下所示：

```
Todd Hall: 99.22
Ralph Smith: -19.08
John Doe: 3434.34
Jane Baker: 1378.0
Tom Smith: 123.22

John Doe’s new balance: 4434.34
```

重要的一点是：和映射类一样，Hashtable不直接支持迭代函数。因此，上面的程序使用枚举来显示balance的内容。然而，我们可以获得允许使用迭代函数的散列表的集合视图。为了实现它，可以简单地使用由Map定义的一个类集“视图”方法，如entrySet( )或keySet( )方法。例如，可以获得关键字的一个集合“视图”，并遍历这些关键字。下面是采用这种技术后重新编写的程序：

```
// Use iterators with a Hashtable.
import java.util.*;

class HTDemo2 {
  public static void main(String args[]) {
    Hashtable balance = new Hashtable();
    String str;
    double bal;

    balance.put("John Doe", new Double(3434.34));
    balance.put("Tom Smith", new Double(123.22));
    balance.put("Jane Baker", new Double(1378.00));
    balance.put("Todd Hall", new Double(99.22));
    balance.put("Ralph Smith", new Double(-19.08));

    // show all balances in hashtable
    Set set = balance.keySet(); // get set-view of keys

    // get iterator
```

```
    Iterator itr = set.iterator();
    while(itr.hasNext()) {
      str = (String) itr.next();
      System.out.println(str + ": " +
                         balance.get(str));
    }

    System.out.println();

    // Deposit 1,000 into John Doe's account
    bal = ((Double)balance.get("John Doe")).doubleValue();
    balance.put("John Doe", new Double(bal+1000));
    System.out.println("John Doe's new balance: " +
                       balance.get("John Doe"));
  }
}
```

15.10.6 Properties

属性（Properties）是Hashtable的一个子类。它用来保持值的列表，在其中关键字和值都是字符串（String）。Properties类被许多其他的Java类所使用。例如，当获得系统环境值时，System.getProperties( )返回对象的类型。

Properties定义了下面的实例变量：

```
Properties defaults;
```

这个变量包含了一个与属性（Properties）对象相关联的默认属性列表。Properties定义了如下的构造函数：

```
Properties( )
Properties(Properties propDefault)
```

第一种形式创建一个没有默认值的属性（Properties）对象。第二种形式创建一个将propDefault作为其默认值的对象。在这两种情况下，属性列表都是空的。

除了Properties从Hashtable中继承下来的方法之外，Properties自己定义的方法列在表15-14中。Properties也包含了一个不被赞成使用的方法：save( )。它被store( )方法所取代，因为它不能正确地处理错误。

表 15-14 由 Properties 定义的从以前版本遗留下来的方法

方法	描述
String getProperty(String key)	返回与key相关联的值。如果key既不在列表中，也不在默认属性列表中，则返回一个null对象
String getProperty(String key, String defaultProperty)	返回与key相关联的值。如果key既不在列表中，也不在默认属性列表中，则返回defaultProperty
void list(PrintStream streamOut)	将属性列表发送给与streamOut相链接的输出流
void list(PrintWriter streamOut)	将属性列表发送给与streamOut相链接的输出流

续表

方法	描述
void load(InputStream streamIn) throws IOException	从与streamIn相链接的输入数据流输入一个属性列表
Enumeration propertyNames( )	返回关键字的枚举，也包括那些在默认属性列表中找到的关键字
Object setProperty(String key, String value)	将value与key关联。返回与key关联的前一个值，如果不存在这样的关联，则返回null（为了保持一致性，在Java 2中新增加的）
void store(OutputStream streamOut, String description)	在写入由description指定的字符串之后，属性列表被写入与streamOut相链接的输出流（在Java 2中新增加的）

Properties类的一个有用的功能是可以指定一个默认属性，如果没有值与特定的关键字相关联，则返回这个默认属性。例如，默认值可以与关键字一起在getProperty( )方法中被指定——如getProperty(“name”，“default value”)。如果“name”值没有找到，则返回“default value”。当构造一个Properties对象时，可以传递Properties的另一个实例做为新实例的默认值。在这种情况下，如果对一个给定的Properties对象调用getProperty(“foo”)，而“foo”并不存在时，Java在默认Properties对象中寻找“foo”。它允许默认属性的任意层嵌套。

下面的例子说明了Properties。该程序创建一个属性列表，在其中关键字是美国的州名，值是这些州的首府的名字。注意试图寻找包括默认值的Florida的首府时的情况。

```
// Demonstrate a Property list.
import java.util.*;

class PropDemo {
  public static void main(String args[]) {
    Properties capitals = new Properties();
    Set states;
    String str;

    capitals.put("Illinois", "Springfield");
    capitals.put("Missouri", "Jefferson City");
    capitals.put("Washington", "Olympia");
    capitals.put("California", "Sacramento");
    capitals.put("Indiana", "Indianapolis");

    // Show all states and capitals in hashtable.
    states = capitals.keySet(); // get set-view of keys
    Iterator itr = states.iterator();

    while(itr.hasNext()) {
      str = (String) itr.next();
      System.out.println("The capital of " +
                         str + " is " +
                         capitals.getProperty(str)
                         + ".");
    }
```

```
    System.out.println();

    // look for state not in list -- specify default
    str = capitals.getProperty("Florida", "Not Found");
    System.out.println("The capital of Florida is "
                       + str + ".");
  }
}
```

该程序的输出如下所示：

```
The capital of Missouri is Jefferson City.
The capital of Illinois is Springfield.
The capital of Indiana is Indianapolis.
The capital of California is Sacramento.
The capital of Washington is Olympia.

The capital of Florida is Not Found.
```

由于Florida不在列表中，所以使用了默认值。

尽管当调用getProperty( )方法时，使用默认值是十分有效的，正如上面的程序所展示的那样，对大多数属性列表的应用来说，有更好的方法去处理默认值。为了更大的灵活性，当构造一个属性（Properties）对象时，指定一个默认的属性列表。如果在主列表中没有发现期望的关键字，将会搜索默认列表。例如，下面是对前面程序稍作修改的程序。在该程序中，有一个指定州的默认列表。在这种情况下，当搜索Florida时，将在默认列表中找到它。

```
// Use a default property list.
import java.util.*;

class PropDemoDef {
  public static void main(String args[]) {
    Properties defList = new Properties();
    defList.put("Florida", "Tallahassee");
    defList.put("Wisconsin", "Madison");

    Properties capitals = new Properties(defList);
    Set states;
    String str;

    capitals.put("Illinois", "Springfield");
    capitals.put("Missouri", "Jefferson City");
    capitals.put("Washington", "Olympia");
    capitals.put("California", "Sacramento");
    capitals.put("Indiana", "Indianapolis");

    // Show all states and capitals in hashtable.
    states = capitals.keySet(); // get set-view of keys
    Iterator itr = states.iterator();

    while(itr.hasNext()) {
      str = (String) itr.next();
      System.out.println("The capital of " +
                         str + " is " +
```

```
                    capitals.getProperty(str)
                    + ".");
    }

    System.out.println();

    // Florida will now be found in the default list.
    str = capitals.getProperty("Florida");
    System.out.println("The capital of Florida is "
                    + str + ".");
  }
}
```

15.10.7　使用store( )和load( )

Properties的一个最有用的方面是可以利用store( )和load( )方法方便地对包含在属性（Properties）对象中的信息进行存储或从盘中装入信息。在任何时候，都可以将一个属性（Properties）对象写入流或从中将其读出。这使得属性列表特别方便实现简单的数据库。例如，下面的程序使用属性列表创建一个简单的用计算机处理的存储着姓名和电话号码的电话本。为了寻找某人的电话号码，可输入他或者她的名字。程序使用store( )和load( )方法来存储和检索列表。当程序执行时，它首先试着从一个叫做phonebook.dat的文件中装入列表。如果这个文件存在，列表就被装入。然后就可以增加列表。如果这样做了，当终止程序时，新列表就会被保存。注意：实现一个小且实用的计算机化的电话号码本只需要很少的程序代码。

```
/* A simple telephone number database that uses
   a property list. */
import java.io.*;
import java.util.*;

class Phonebook {
  public static void main(String args[])
    throws IOException
  {
    Properties ht = new Properties();
    BufferedReader br =
      new BufferedReader(new InputStreamReader(System.in));
    String name, number;
    FileInputStream fin = null;
    boolean changed = false;

    // Try to open phonebook.dat file.
    try {
      fin = new FileInputStream("phonebook.dat");
    } catch(FileNotFoundException e) {
      // ignore missing file
    }

    /* If phonebook file already exists,
       load existing telephone numbers. */
    try {
      if(fin != null) {
        ht.load(fin);
```

```
        fin.close();
      }
    } catch(IOException e) {
      System.out.println("Error reading file.");
    }

    // Let user enter new names and numbers.
    do {
      System.out.println("Enter new name" +
                         " ('quit' to stop): ");
      name = br.readLine();
      if(name.equals("quit")) continue;

      System.out.println("Enter number: ");
      number = br.readLine();

      ht.put(name, number);
      changed = true;
    } while(!name.equals("quit"));

    // If phone book data has changed, save it.
    if(changed) {
      FileOutputStream fout = new FileOutputStream("phonebook.dat");

      ht.store(fout, "Telephone Book");
      fout.close();
    }

    // Look up numbers given a name.
    do {
      System.out.println("Enter name to find" +
                         " ('quit' to quit): ");
      name = br.readLine();
      if(name.equals("quit")) continue;

      number = (String) ht.get(name);
      System.out.println(number);
    } while(!name.equals("quit"));
  }
}
```

15.11 类 集 总 结

类集框架为程序员提供了一个功能强大的设计方案以解决编程过程中面临的大多数任务。下一次当你需要存储和检索信息时，可考虑使用类集。记住，类集不仅仅是专为那些“大型作业”，例如联合数据库，邮件列表或产品清单系统等所专用的。它们对于一些小型作业也是很有效的。例如，TreeMap可以给出一个很好的类集以保留一组文件的字典结构。TreeSet在存储工程管理信息时是十分有用的。坦白地说，对于采用基于类集的解决方案而受益的问题种类只受限于你的想象力。

第 16 章　java.util 第 2 部分：更多的实用工具类

本章将通过浏览那些不属于类集框架的类和接口的方式来继续研究java.util。这其中包括了标记字符串，处理日期，计算随机数以及观测事件。也包括了在本章后面简要提及的java.util.zip包和java.util.jar包。

16.1　StringTokenizer（字符串标记）

对文本的处理经常包括对格式化的输入字符串进行语法分析。语法分析（Parsing）将文本划分为一组不连续的部分，或标记（tokens），在一个确定的序列中，标记可以表达语义。StringTokenizer类提供了语法分析处理的第一步。经常被称为lexer（词法分析程序）或scanner（扫描程序）。StringTokenizer实现枚举（Enumeration）接口。因此，给定一个输入字符串，可以使用StringTokenizer对包含于其中的单独标记进行枚举。

使用StringTokenizer时，指定一个输入字符串和一个包含了分割符的字符串。分割符（Delimiters）是分割标记的字符。分割符字符串中的每一个字符被当做一个有效的分割符——例如，“,;:”建立逗号，分号和冒号分割符。默认建立的分割符有空白符字符，空格，tab键，换行以及回车。

StringTokenizer的构造函数如下所示：

```
StringTokenizer(String str)
StringTokenizer(String str, String delimiters)
StringTokenizer(String str, String delimiters, boolean delimAsToken)
```

在上述三种形式中，str都表示将被标记的字符串。在第一种形式中，使用默认的分割符。在第二种和第三种形式中，delimiters是用来指定分割符的一个字符串。在第三种形式中，如果delimAsToken为true，当字符串被分析时，分割符也被作为标记而被返回；否则，不返回分割符。在第一种和第二种形式中，分割符不会作为标记而被返回。

一旦创建了StringTokenizer对象之后，nextToken( )方法被用于抽取连续的标记。当有更多的标记被抽取时，hasMoreTokens( )方法返回true。因为StringTokenizer实现枚举（Enumeration），因此hasMoreElements( )和nextElement( )方法也被实现，同时它们的作用也分别与hasMoreTokens( )和nextToken( )方法相同。StringTokenizer方法列在表16-1中。

下面是一个创建用于分析“key=value”对的StringTokenizer的例子。连续的多组“key=value”对将用分号分开。

```
// Demonstrate StringTokenizer.
import java.util.StringTokenizer;

class STDemo {
  static String in = "title=Java: The Complete Reference;" +
```

```
    "author=Schildt;" +
    "publisher=Osborne/McGraw-Hill;" +
    "copyright=2001";

  public static void main(String args[]) {
    StringTokenizer st = new StringTokenizer(in, "=;");

    while(st.hasMoreTokens()) {
      String key = st.nextToken();
      String val = st.nextToken();
      System.out.println(key + "\t" + val);
    }
  }
}
```

表 16-1 由 StringTokenizer 定义的方法

方法	描述
int countTokens( )	使用当前分割符集，该方法确定还没被分析的标记的个数并返回结果
boolean hasMoreElements( )	如果在字符串中包含有一个或多个标记，则返回true；如果在字符串中不包含标记，则返回false
boolean hasMoreTokens( )	如果在字符串中包含有一个或多个标记，则返回true；如果在字符串中不包含标记，则返回false
Object nextElement( )	将下一个标记作为Object返回
String nextToken( )	将下一个标记作为String返回
String nextToken(String delimiters)	将下一个标记作为String返回并且将分割符字符串设为由delimiters指定的字符串

该程序的输出如下所示：

```
title  Java: The Complete Reference
author  Schildt
publisher  Osborne/McGraw-Hill
copyright  2001
```

16.2 BitSet（置位）

BitSet类创建一个专用类型的数组，该数组包含位的值。而该数组的大小可以按需要进行增加。这使得它与位矢量相似。BitSet的构造函数如下所示：

```
BitSet( )
BitSet(int size)
```

第一种形式创建一个默认的对象。第二种形式允许指定其初始大小（也就是说，它所能包含的位的个数）。所有的位被初始化为0。

BitSet实现Cloneable接口并且定义了表16-2中列出的方法。

表 16-2　由 BitSet 定义的方法

方法	描述
void and(BitSet bitSet)	将调用BitSet对象的内容与由bitSet指定的那些内容进行相与（AND）运算，结果被放置在调用对象中
void andNot(BitSet bitSet)	对应bitSet中的每一位，清除调用BitSet中相应的位（在Java 2中新增加的）
void clear(int index)	对由index指定的位进行置0
Object clone( )	复制调用BitSet对象
Boolean equals(Object bitSet)	如果调用位集合与由bitSet传递的位集合相等，则返回true；否则，该方法返回false
Boolean get(int bitIndex)	返回指定下标处位的当前状态
int hashCode( )	返回调用对象的散列值
int length( )	返回包含调用BitSet的内容所需的位的个数。该值由最后1位的位置确定（在Java 2中新增加的）
void or(BitSet bitSet)	将调用对象的内容与由bitSet指定的内容进行“或”（OR）运算，结果被放置在调用对象中
void set(int index)	设置由index指定的位
int size( )	返回调用BitSet对象中位的个数
String toString( )	返回调用BitSet对象的字符串等价形式
void xor(BitSet bitSet)	将调用BitSet对象的内容与由bitSet指定的内容进行“异或”（XOR）运算，结果被放置在调用对象中

这里是一个说明BitSet的例子。

```
// BitSet Demonstration.
import java.util.BitSet;

class BitSetDemo {
  public static void main(String args[]) {
    BitSet bits1 = new BitSet(16);
    BitSet bits2 = new BitSet(16);

    // set some bits
    for(int i=0; i<16; i++) {
      if((i%2) == 0) bits1.set(i);
      if((i%5) != 0) bits2.set(i);
    }

    System.out.println("Initial pattern in bits1: ");
    System.out.println(bits1);
    System.out.println("\nInitial pattern in bits2: ");
    System.out.println(bits2);

    // AND bits
    bits2.and(bits1);
```

```
    System.out.println("\nbits2 AND bits1: ");
    System.out.println(bits2);

    // OR bits
    bits2.or(bits1);
    System.out.println("\nbits2 OR bits1: ");
    System.out.println(bits2);

    // XOR bits
    bits2.xor(bits1);
    System.out.println("\nbits2 XOR bits1: ");
    System.out.println(bits2);
  }
}
```

下面是该程序的输出。当用toString( )方法将一个BitSet对象转换成其相应的字符串形式后，每组位就由其位置表示。被清除的位将不被显示。

```
Initial pattern in bits1:
{0, 2, 4, 6, 8, 10, 12, 14}

Initial pattern in bits2:
{1, 2, 3, 4, 6, 7, 8, 9, 11, 12, 13, 14}

bits2 AND bits1:
{2, 4, 6, 8, 12, 14}

bits2 OR bits1:
{0, 2, 4, 6, 8, 10, 12, 14}

bits2 XOR bits1:
{}
```

16.3 Date（日期）

Date类封装当前的日期和时间。在开始检验Date类之前，需要着重强调的是：这里讨论的Date类与其在Java 1.0中所定义的原始版本相比，已经发生了本质上的改变。在Java 1.1公布的时候，许多原来由原始Date类执行的函数功能被移入到Calendar和DateFormat类中，这样做的结果就导致许多最初1.0中的Date方法被摈弃。在Java 2中对时间和日期类增加了几个新的方法，但是以与Java 1.1中相同的形式来实现它们。由于被摈弃的Java 1.0中的方法不被新的程序所使用，这里就不讨论它们了。

Date支持下面的构造函数：

```
Date( )
Date(long millisec)
```

第一种形式的构造函数用当前的日期和时间初始化对象。第二种形式的构造函数接收一个参数，该参数等于从1970年1月1日午夜起至今的毫秒数的大小。由Date类定义的未被摈弃的方法列在表16-3中。随着Java 2的出现，Date也实现了Comparable接口。

表 16-3　由 Date 定义的未被摈弃的方法

方法	描述
boolean after(Date date)	如果调用Date对象所包含的日期迟于由date指定的日期，则返回true；否则返回false
boolean before(Date date)	如果调用Date对象所包含的日期早于由date指定的日期，则返回true；否则返回false
Object clone( )	复制调用Date对象
int compareTo(Date date)	将调用对象的值与date的值进行比较。如果这两者数值相等，则返回0；如果调用对象的值早于date的值，则返回一个负值；如果调用对象的值晚于date的值，则返回一个正值（在Java 2中新增加的）
int compareTo(Object obj)	如果obj属于类Date，其操作与compareTo(Date)相同；否则，引发一个ClassCastException异常（在Java 2中新增加的）
boolean equals(Object date)	如果调用Date对象包含的时间和日期与由date指定的时间和日期相同，则返回true；否则，返回false
long getTime( )	返回自1970年1月1日起至今的毫秒数的大小
int hashCode( )	返回调用对象的散列值
void setTime(long time)	按time的指定，设置时间和日期，表示自1970年1月1日午夜至今的以毫秒为单位的时间值
String toString( )	将调用Date对象转换成字符串并且返回结果

正如从表16-3中看到的那样，Date功能部件不允许单独获得日期或时间分量。通过下面的程序可以看到，仅能获得以毫秒为单位的日期和时间或通过调用toString( )方法来获得其默认的字符串表达式。为了获得关于日期和时间的更加详细的信息，可以使用Calendar类。

```
// Show date and time using only Date methods.
import java.util.Date;

class DateDemo {
  public static void main(String args[]) {
    // Instantiate a Date object
    Date date = new Date();

    // display time and date using toString()
    System.out.println(date);

    // Display number of milliseconds since midnight, January 1, 1970 GMT
    long msec = date.getTime();
    System.out.println("Milliseconds since Jan. 1, 1970 GMT = " + msec);
  }
}
```

程序的输出如下所示：

```
Thu Jan 25 15:06:40 CST 2001
```

```
Milliseconds since Jan. 1, 1970 GMT = 980456763420
```

16.3.1 比较日期

有三种方法可用于比较两个Date对象。首先，可以对两个对象使用getTime( )方法获得它们各自自1970年1月1日午夜起至今的毫秒数的大小。然后比较这两个值的大小。其次，可以使用before( )，after( )以及equals( )方法。例如，由于每个月的12号出现在18号之前，所以new Date(99, 2, 12).before(new Date (99, 2, 18))将返回true。最后，可以使用由Comparable接口定义，被Date实现的compareTo( )方法。

16.4 Calendar（日历）

抽象Calendar类提供了一组方法，这些方法允许将以毫秒为单位的时间转换为一组有用的分量。一些可以提供信息的类型是：年，月，日，小时，分和秒。Calendar的子类能提供特定的功能，以便按照它们本身的规则去解释时间信息，这是能够写出在几个国际环境下都能运行的程序的Java类库的一个方面。这种子类的一个例子是GregorianCalendar。

Calendar提供非公共的构造函数。

Calendar定义了几个受保护的实例变量。areFieldsSet是一个指示时间分量是否已经建立的boolean型变量。fields是一个包含了时间分量的ints数组。isSet是一个指示特定时间分量是否已经建立的boolean数组。time是一个包含了该对象的当前时间的long型变量。isTimeSet是一个指示当前时间是否已经建立的boolean型变量。

由Calendar定义的一些常用的方法列在表16-4中。

表 16-4 由 Calendar 定义的常见方法

方法	描述
abstract void add(int which, int val)	将val加到由which指定的时间或日期分量。为了实现减功能，可以加一个负数。which必须是由Calendar定义的域之一，例如Calendar.HOUR
boolean after(Object calendarObj)	如果调用Calendar对象所包含的日期晚于由calendarObj指定的日期，则返回true；否则，返回false
boolean before(Object calendarObj)	如果调用Calendar对象所包含的日期早于由calendarObj指定的日期，则返回true；否则返回false
final void clear( )	对调用对象的所有时间分量置0
final void clear(int which)	在调用对象中，对由which指定的时间分量置0
Object clone( )	返回对调用对象的复制
Boolean equals(Object calendarObj)	如果调用Calendar对象所包含的日期与由calendarObj指定的日期相等，则返回true；否则返回false
final int get(int calendarField)	返回调用对象的一个分量的值。该分量由calendarField指定。可以被请求的分量的示例有：Calendar.YEAR，Calendar.MONTH，Calendar.MINUTE等等

续表

方法	描述
static Locale[ ] getAvailableLocales( )	返回一个Locale对象的数组，其中包含了可以使用日历的地区
static Calendar getInstance( )	对默认的地区和时区，返回一个Calendar对象
static Calendar getInstance(Locale locale)	对由locale指定的地区，返回一个Calendar对象，而时区使用默认的时区
static Calendar getInstance(TimeZone tz,Locale locale)	对由tz指定的时区，同时由locale指定的地区返回一个Calendar对象
final Date getTime( )	返回一个与调用对象的时间相等的Date对象
TimeZone getTimeZone( )	返回调用对象的时区
final boolean isSet(int which)	如果指定的时间分量被设置，则返回true；否则返回false
final void set(int which, int val)	在调用对象中，将由which指定的日期和时间分量赋给由val指定的值。Which必须是由Calendar定义的域之一。例如Calendar.HOUR
final void set(int year, int month, int dayOfMonth)	设置调用对象的各种日期和时间分量
final void set(int year, int month, int dayOfMonth, int hours, int minutes)	设置调用对象的各种日期和时间分量
final void set(int year, int month, int dayOfMonth, int hours, int minutes, int seconds)	设置调用对象的各种日期和时间分量
final void setTime(Date d)	设置调用对象的各种日期和时间分量。该信息从Date对象d中获得
void setTimeZone(TimeZone tz)	将调用对象的时区设置为由tz指定的时区

Calendar定义了下面的int常数。这些常数用于得到或设置日历分量：

AM	FRIDAY	PM
AM_PM	HOUR	SATURDAY
APRIL	HOUR_OF_DAY	SECOND
AUGUST	JANUARY	SEPTEMBER
DATE	JULY	SUNDAY
DAY_OF_MONTH	JUNE	THURSDAY
DAY_OF_WEEK	MARCH	TUESDAY
DAY_OF_WEEK_IN_MONTH	MAY	UNDECIMBER
DAY_OF_YEAR	MILLISECOND	WEDNESDAY
DECEMBER	MINUTE	WEEK_OF_MONTH
DST_OFFSET	MONDAY	WEEK_OF_YEAR

ERA	MONTH	YEAR
FEBRUARY	NOVEMBER	ZONE_OFFSET
FIELD_COUNT	OCTOBER	

下面的程序举例说明了几个Calendar方法：

```
// Demonstrate Calendar
import java.util.Calendar;

class CalendarDemo {
  public static void main(String args[]) {
    String months[] = {
            "Jan", "Feb", "Mar", "Apr",
            "May", "Jun", "Jul", "Aug",
            "Sep", "Oct", "Nov", "Dec"};

    // Create a calendar initialized with the
    // current date and time in the default
    // locale and timezone.
    Calendar calendar = Calendar.getInstance();

    // Display current time and date information.
    System.out.print("Date: ");
    System.out.print(months[calendar.get(Calendar.MONTH)]);
    System.out.print(" " + calendar.get(Calendar.DATE) + " ");
    System.out.println(calendar.get(Calendar.YEAR));

    System.out.print("Time: ");
    System.out.print(calendar.get(Calendar.HOUR) + ":");
    System.out.print(calendar.get(Calendar.MINUTE) + ":");
    System.out.println(calendar.get(Calendar.SECOND));

    // Set the time and date information and display it.
    calendar.set(Calendar.HOUR, 10);
    calendar.set(Calendar.MINUTE, 29);
    calendar.set(Calendar.SECOND, 22);

    System.out.print("Updated time: ");
    System.out.print(calendar.get(Calendar.HOUR) + ":");
    System.out.print(calendar.get(Calendar.MINUTE) + ":");
    System.out.println(calendar.get(Calendar.SECOND));
  }
}
```

该程序的输出显示如下：

```
Date: Jan 25 2001
Time: 11:24:25
Updated time: 10:29:22
```

16.5　GregorianCalendar（标准阳历）

GregorianCalendar是Calendar的一个实现大家所熟悉的标准日历（格列高利历）的具体工具。Calendar的getInstance( )方法返回用默认的地区和时区的当前日期和当前时间所初始化的GregorianCalendar（标准日历）。

GregorianCalendar定义了两个域：AD和BC。它们代表由公历定义的两个纪元。

对GregorianCalendar对象，也有几个构造函数。默认的GregorianCalendar( )方法用默认的地区和时区的当前日期和当前时间初始化对象。提供的三种构造函数如下：

```
GregorianCalendar(int year, int month, int dayOfMonth)
GregorianCalendar(int year, int month, int dayOfMonth, int hours,
                                  int minutes)
GregorianCalendar(int year, int month, int dayOfMonth, int hours,
                                  int minutes, int seconds)
```

三种形式中，都设置了日，月和年。这里，year指定了从1900年起的年数。month指定了月，以0表示一月。月中的日由dayOfMonth指定。第一种形式以午夜设置时间。第二种形式以小时和分钟设置时间，第三种形式增加了秒。

也可以通过指定地区和/或时区来构造GregorianCalendar对象。下面的构造函数创建了用指定的时区和/或地区的当前日期和当前时间来初始化的对象。

```
GregorianCalendar(Locale locale)
GregorianCalendar(TimeZone timeZone)
GregorianCalendar(TimeZone timeZone, Locale locale)
```

GregorianCalendar对Calendar的所有抽象方法提供了实现工具。它也提供了一些另外的方法。其中最令人感兴趣的大概是isLeapYear( )，该方法用于测试某年是否是闰年。它的形式如下：

```
boolean isLeapYear(int year)
```

当year是一个闰年时，该方法返回true；否则返回false。

下面的程序说明了GregorianCalendar。

```
// Demonstrate GregorianCalendar
import java.util.*;

class GregorianCalendarDemo {
  public static void main(String args[]) {
    String months[] = {
            "Jan", "Feb", "Mar", "Apr",
            "May", "Jun", "Jul", "Aug",
            "Sep", "Oct", "Nov", "Dec"};
    int year;
```

```
    // Create a Gregorian calendar initialized
    // with the current date and time in the
    // default locale and timezone.
    GregorianCalendar gcalendar = new GregorianCalendar();

    // Display current time and date information.
    System.out.print("Date: ");
    System.out.print(months[gcalendar.get(Calendar.MONTH)]);
    System.out.print(" " + gcalendar.get(Calendar.DATE) + " ");
    System.out.println(year = gcalendar.get(Calendar.YEAR));

    System.out.print("Time: ");
    System.out.print(gcalendar.get(Calendar.HOUR) + ":");
    System.out.print(gcalendar.get(Calendar.MINUTE) + ":");
    System.out.println(gcalendar.get(Calendar.SECOND));

    // Test if the current year is a leap year
    if(gcalendar.isLeapYear(year)) {
      System.out.println("The current year is a leap year");
    }
    else {
      System.out.println("The current year is not a leap year");
    }
  }
}
```

该程序的输出如下所示：

```
Date: Jan 25 2001
Time: 11:25:27
The current year is not a leap year
```

16.6 TimeZone（时区）

另一个与时间有关的类是TimeZone。TimeZone类允许给出相对于格林威治时间（GMT），也称为世界时间（UTC）的时区差。它也计算夏令时。TimeZone仅仅提供默认的构造函数。

由TimeZone定义的一些方法总结在表16-5中。

表 16-5 由 TimeZone 定义的一些方法

方法	描述
Object clone( )	返回clone( )方法的特定时区版本
Static String[ ] getAvailableIDs( )	返回一个表示所有时区的名字的字符串（String）对象的数组

续表

方法	描述
Static String[ ] getAvailableIDs(int timeDelta)	返回一个表示所有时区的名字的字符串（String）对象的数组，时区名为相对于GMT的timeDelta偏移
Static TimeZone getDefault( )	返回一个表示被主机使用的默认时区的TimeZone对象
String getID( )	返回调用TimeZone对象的名字
Abstract int getOffset(int era, int year, int month, int dayOfMonth, int dayOfWeek, int millisec)	返回计算当地时间而在GMT中加入的偏移量。该值为夏令时而做调整。该方法的参数表示日期和时间分量
Abstract int getRawOffset( )	返回计算当地时间而在GMT中加入的未处理的偏移量。该值不对夏令时做调整
static TimeZone getTimeZone(String tzName)	对名字为tzName的时区返回TimeZone对象
abstract boolean inDaylightTime(Date d)	如果调用对象中用d表示的日期对应夏令时，则返回true；否则返回false
static void setDefault(TimeZone tz)	设置主机使用的默认时区。tz是对使用的TimeZone对象的一个引用
void setID(String tzName)	将时区的名字（也就是它的ID）设置为由tzName指定的名字
abstract void setRawOffset(int millis)	确定相对GMT的以毫秒为单位的偏移量
abstract boolean useDaylightTime( )	如果调用对象使用夏令时，则返回true；否则返回false

16.7　SimpleTimeZone

SimpleTimeZone类是TimeZone的一个方便的子类。它实现TimeZone的抽象方法，并允许对公历进行时区操作。它也计算夏令时。

SimpleTimeZone定义了三个构造函数。其中之一为：

```
SimpleTimeZone(int timeDelta, String tzName)
```

该构造函数创建了SimpleTimeZone对象。与格林威治标准时间（GMT）的偏移量是timeDelta。时区为tzName。

第二个SimpleTimeZone构造函数为：

```
SimpleTimeZone(int timeDelta, String tzId, int dstMonth0,
                               int dstDayInMonth0, int dstDay0, int time0,
                               int dstMonth1, int dstDayInMonth1, int
dstDay1,
                               int time1)
```

这里相对于GMT的偏移量由timeDelta指定。时区名由tzId传递。夏令时的开始由参数dstMonth0，dstDayInMonth0，dstDay0和time0指出。而夏令时的结束由dstMonth1，dstDayInMonth1，dstDay1和time1指出。

第三个SimpleTimeZone构造函数为：

```
SimpleTimeZone(int timeDelta, String tzId, int dstMonth0,
                                int dstDayInMonth0, int dstDay0, int time0,
                                 int dstMonth1, int dstDayInMonth1, int
dstDay1,
                                int time1, int dstDelta)
```

这里dstDelta是在夏令时期间保存的毫秒数。

16.8 Locale（地区）

Locale类被实例化以生成其中每一个描述一个地理或文化区域的对象。它是提供了编写在不同的国际环境下都能运行的程序的几个类之一。例如，用于显示各个区域的日期，时间和数字区别的格式设计。

国际化是一个超过本书讨论范围的大课题。然而大多数的程序仅仅需要处理其中的一些基本要求，包括设置当前的地区。

Locale类定义了下面的一些常数用于处理最常见的地区。

CANADA	GERMAN	KOREAN
CANADA_FRENCH	GERMANY	PRC
CHINA	ITALIAN	SIMPLIFIED_CHINESE
CHINESE	ITALY	TAIWAN
ENGLISH	JAPAN	TRADITIONAL_CHINESE
FRANCE	JAPANESE	UK
FRENCH	KOREA	US

例如，表达式Locale.CANADA代表了对应加拿大的Locale对象。

Locale的构造函数为：

```
Locale(String language, String country)
Locale(String language, String country, String data)
```

这些构造函数建立一个代表指定语言（language）和国家（country）的Locale对象。这些值必须包含ISO标准语言和国家代码。辅助浏览器和特定供应商的信息可以在data中提供。

Locale定义了几种方法。其中最重要的一种是setDefault( )，说明如下：

```
static void setDefault(Locale localeObj)
```

它将默认区域设置为由localeObj指定的值。

其他的一些比较感兴趣的方法如下：

```
final String getDisplayCountry( )
final String getDisplayLanguage( )
final String getDisplayName( )
```

它们返回可用于显示国家名字，语言种类和对地区做完整描述的可读的字符串。

通过使用getDefault( )方法可以获得默认的区域，说明如下：

```
static Locale getDefault( )
```

Calendar 和 GregorianCalendar 是按地区方式工作的类的例子。DateFormat 和 SimpleDateFormat也与地区有关。

16.9　Random

Random类是伪随机数的产生器。之所以称之为伪随机数是因为它们是简单的均匀分布序列。Random定义了下面的构造函数。

```
Random( )
Random(long seed)
```

第一种形式创建一个使用当前时间作为起始值或称为初值的数字发生器。第二种形式允许人为指定一个初值。

如果用初值初始化了一个Random对象，就对随机序列定义了起始点。如果用相同的初值初始化另一个Random对象，将获得同一随机序列。如果要生成不同的序列，应当指定不同的初值。实现这种处理的最简单的方法是使用当前时间作为产生Random对象的初值。这种方法减少了得到相同序列的可能性。

由Random定义的公共方法列在表16-6中。

表 16-6　由 Random 定义的方法

方法	描述
boolean nextBoolean( )	返回下一个布尔（boolean）随机数（在Java 2中新增加的）
void nextBytes(byte vals[ ])	用随机产生的值填充vals
double nextDouble( )	返回下一个双精度（double）随机数
float nextFloat( )	返回下一个浮点（float）随机数
double nextGaussian( )	返回下一个高斯随机数
int nextInt( )	返回下一个整型（int）随机数
int nextInt(int n)	返回下一个介于0和n之间的整型（int）随机数（在Java 2中新增加的）
long nextLong( )	返回下一个长整型（long）随机数
void setSeed(long newSeed)	将由newSeed指定的值作为种子值（也就是随机数产生器的开始值）

正如你能看到的，从Random对象中可以提取七种类型的随机数。从nextBoolean( )方法中可以获得随机布尔数。通过调用nextBytes( )方法可以获得随机字节数。通过调用nextInt( )方法可以获得随机整型数。通过调用nextLong( )方法可以获得均匀分布的长整型随机数。通过调用nextFloat( )和nextDouble( )方法可以分别得到在0.0到1.0之间的，均匀分布的float和double随机数。最后，调用nextGaussian( )方法返回中心在0.0，标准偏差为1.0的double值，这就是著名的钟型曲线。

下面是一个说明由nextGaussian( )方法产生序列的例子。该程序得到100个随机高斯数值，并计算它们的平均值。该程序也统计落在正或负两种标准偏差，且对每一种分类按增量为0.5递增的值的个数，程序运行结果在屏幕上靠左（或靠右）用图形方式画出。

```
// Demonstrate random Gaussian values.
import java.util.Random;

class RandDemo {
  public static void main(String args[]) {
    Random r = new Random();
    double val;
    double sum = 0;
    int bell[] = new int[10];

    for(int i=0; i<100; i++) {
      val = r.nextGaussian();
      sum += val;
      double t = -2;
      for(int x=0; x<10; x++, t += 0.5)
        if(val < t) {
          bell[x]++;
          break;
        }
    }
    System.out.println("Average of values: " +
                       (sum/100));

    // display bell curve, sideways
    for(int i=0; i<10; i++) {
      for(int x=bell[i]; x>0; x--)
        System.out.print("*");
      System.out.println();
    }
  }
}
```

这里是程序运行的结果，正如你能看到的那样，结果获得了数字的一个钟型分布。

```
Average of values: 0.0702235271133344
**
*******
******
***************
******************
*****************
*************
**********
********
***
```

16.10　Observable （观测）

Observable类用于创建可以观测到你的程序中其他部分的子类。当这种子类的对象发生变化时，观测类被通知。观测类必须实现定义了update( )方法的Observer接口。当一个观测程序被通知到一个被观测对象的改变时，update( )方法被调用。

Observable定义了表16-7中的方法。一个被观测的对象必须服从下面的两个简单规则。第一，如果它被改变了，它必须调用setChanged( )方法。第二，当它准备通知观测程序它的改变时，它必须调用notifyObservers( )方法。这导致了在观测对象中对update( )方法的调用。注意——当对象在调用notifyObservers( )方法之前，没有调用setChanged( )方法，就不会有什么动作发生。在update( )被调用之前，被观测对象必须调用setChanged( )和notifyObservers( )两种方法。

表 16-7　由 Observable 定义的方法

方法	描述
void addObserver(Observer obj)	将obj增加到观测调用对象的对象列表中
protected void clearChanged( )	调用该方法返回调用对象的状态为“未改变的”
int countObservers( )	返回观测调用对象的对象的个数
void deleteObserver(Observer obj)	从观测调用对象的对象列表中删除obj
void deleteObservers( )	删除对调用对象的所有观测程序
boolean hasChanged( )	如果调用对象已经被改变了，则返回true；如果它没有被改变，则返回false
void notifyObservers( )	通过调用update( )方法，将调用对象的改变通知其所有观测程序。null被传递给update( )方法作为其第二个参数
void notifyObservers(Object obj)	通过调用update( )方法，将调用对象的改变通知其所有观测程序。obj被传递给update( )方法作为其第二个参数
protected void setChanged( )	当调用对象发生改变时被调用

注意notifyObservers()有两种形式：一种带有参数而另一种没有。当用参数调用notifyObservers( )方法时，该对象被传给观测程序的update( )方法作为其第二个参数。否则，将给update( )方法传递一个null。可以使用第二个参数传递适合于你的应用程序的任何类型的对象。

16.10.1　观测接口

为了观测一个可观测的对象，必须实现Observer接口。这个接口仅仅定义了如下所示的一个方法。

```
void update(Observable observOb, Object arg)
```

这里，observOb是被观测的对象，而arg是由notifyObservers( )方法传递的值。当被观测

对象发生了改变，调用update( )方法。

16.10.2 观测程序举例

这里是一个说明可观测对象的例子。该程序创建了一个叫做Watcher的类，该类实现了Observer接口。被监控的类叫做BeingWatched，它扩展了Observable。在BeingWatched里，是counter( )方法，该方法仅是从一个指定的值开始递减计数。它使用sleep( )方法在两次计数中间等待十分之一秒。每次计数改变时，notifyObservers( )方法被调用，而当前的计数被作为参数传递给notifyObservers( )方法。这导致了Watcher中的update( )方法被调用，显示当前的计数值。在main( )内，分别调用observing和observed的Watcher和BeingWatched对象被创建。然后，observing被增加到对observed的观测程序列表。这意味着每次counter( )调用notifyObservers( )方法时，observing.update( )方法将被调用。

```
/* Demonstrate the Observable class and the
   Observer interface.
*/

import java.util.*;

// This is the observing class.
class Watcher implements Observer {
  public void update(Observable obj, Object arg) {
    System.out.println("update() called, count is " +
                    ((Integer)arg).intValue());
  }
}

/ This is the class being observed.
class BeingWatched extends Observable {
  void counter(int period) {
    for( ; period >=0; period--) {
      setChanged();
      notifyObservers(new Integer(period));
      try {
        Thread.sleep(100);
      } catch(InterruptedException e) {
        System.out.println("Sleep interrupted");
      }
    }
  }

}

class ObserverDemo {
  public static void main(String args[]) {
    BeingWatched observed = new BeingWatched();
    Watcher observing = new Watcher();

    /* Add the observing to the list of observers for
       observed object.  */
    observed.addObserver(observing);
```

```
    observed.counter(10);
  }
}
```

该程序的输出如下所示：

```
update() called, count is 10
update() called, count is 9
update() called, count is 8
update() called, count is 7
update() called, count is 6
update() called, count is 5
update() called, count is 4
update() called, count is 3
update() called, count is 2
update() called, count is 1
update() called, count is 0
```

有多个对象可以用作观测程序。例如下面程序实现了两个观测类并且将每个类中的一个对象增加到BeingWatched观测程序列表中。第二个观测程序等待直到计数为0，随后振铃。

```
/* An object may be observed by two or more
   observers.
*/

import java.util.*;

// This is the first observing class.
class Watcher1 implements Observer {
  public void update(Observable obj, Object arg) {
    System.out.println("update() called, count is " +
                       ((Integer)arg).intValue());
  }
}

// This is the second observing class.
class Watcher2 implements Observer {
  public void update(Observable obj, Object arg) {
    // Ring bell when done
    if(((Integer)arg).intValue() == 0)
      System.out.println("Done" + '\7');
  }
}

// This is the class being observed.
class BeingWatched extends Observable {
  void counter(int period) {
    for( ; period >=0; period--) {
      setChanged();
      notifyObservers(new Integer(period));
      try {
        Thread.sleep(100);
      } catch(InterruptedException e) {
        System.out.println("Sleep interrupted");
      }
    }
```

```
  }
}

class TwoObservers {
  public static void main(String args[]) {
    BeingWatched observed = new BeingWatched();
    Watcher1 observing1 = new Watcher1();
    Watcher2 observing2 = new Watcher2();

    // add both observers
    observed.addObserver(observing1);
    observed.addObserver(observing2);

    observed.counter(10);
  }
}
```

Observable类和Observer接口允许实现基于文档/视图方法的高级程序结构。它们也适用于多线程情况。

16.11 Timer和TimerTask

Java 2的1.3版在java.util中增加了一个有趣又有用的功能部件：提供了提前安排将来某时间要执行任务的能力。支持这项功能的类是Timer和TimerTask。使用这些类可以创建一个工作于后台的线程，该线程等待一段指定的时间。当指定的时间到来时，与该线程相连的任务被执行。不同的选项允许安排一个任务重复执行，或安排一个任务在指定的时间运行。尽管永远都可能使用Thread类利用手工方法创建一个在指定的时间执行的任务，但是使用Timer和TimerTask却大大简化了这一过程。

Timer和TimerTask一起工作。Timer是一个用于安排一个将来执行的任务的类。被安排的任务必须是TimerTask的一个实例。因此，为了安排一个任务，首先应该创建一个TimerTask对象，然后使用Timer的一个实例安排执行它。

TimerTask实现了Runnable接口；因此它可以被用于创建一个执行线程。它的构造函数如下所示：

```
TimerTask( )
```

TimerTask定义了表16-8中的方法。注意run( )是一个抽象方法，这意味着它可以被覆盖。由Runnable接口定义的run( )方法包含了将被执行的程序代码。因此创建一个定时器任务的最简单的办法是扩展TimerTask和重载run( )。

表 16-8 由 TimerTask 定义的方法

方法	描述
boolean cancel( )	终止任务。如果任务的执行被阻止了，则返回true；否则返回false
abstract void run( )	包含了定时器任务的代码
long scheduledExecutionTime( )	返回被安排最后执行的任务的时间

一旦任务被创建，它将通过一个类型Timer的对象被安排执行。Timer的构造函数如下：

```
Timer( )
Timer(boolean DThread)
```

第一种形式创建一个以常规线程方式运行的Timer对象。第二种形式当DThread为true时，使用后台进程线程。只要剩下的程序继续运行，后台进程线程就会执行。由Timer定义的方法列在表16-9中。

表 16-9 由 Timer 定义的方法

方法	描述
void cancel( )	终止定时器线程
void schedule(TimerTask TTask, long wait)	TTask被安排在由参数wait传递的周期之后执行wait参数的单位是毫秒
void schedule(TimerTask TTask, long wait, long repeat)	TTask被安排在由参数wait传递的周期之后执行。任务随后在由repeat指定的时间间隔重复执行。参数wait和repeat的单位都是毫秒
void schedule(TimerTask TTask, Date targetTime)	TTask被安排在由targetTime指定的时间执行
void schedule(TimerTask TTask, Date targetTime, long repeat)	TTask被安排在由targetTime指定的时间执行。接着任务在由repeat传递的时间间隔重复执行。repeat参数的单位是毫秒
void scheduleAtFixedRate(TimerTask TTask, long wait, long repeat)	TTask被安排在经过由参数wait传递的周期之后执行。任务随后在由repeat指定的时间间隔重复执行。参数wait和repeat的单位都是毫秒。每一次重复的时间是和第一次执行，而不是和前一次执行的时间有关。因此执行的总速度是固定的
void scheduleAtFixedRate(TimerTask TTask, Date targetTime, long repeat)	TTask被安排在由targetTime指定的时间执行。任务随后在由repeat指定的时间间隔重复执行。参数repeat的单位是毫秒。每一次重复的时间是和第一次执行，而不是和前一次执行的时间有关。因此执行的总速度是固定的

一旦Timer被创建，将可以通过调用创建的Timer的schedule( )方法来安排任务。正如表16-9所示的那样，这里有几种schedule( )方法的形式，这些形式允许用各种办法来安排任务。

如果创建了一个非后台进程任务，当你的程序结束时，你可能希望调用cancel( )方法来结束任务。如果不这样做的话，你的程序可能被“挂起”一个周期时间。

下面的程序说明了Timer和TimerTask。该程序定义了一个定时器任务，它的run( )方法显示消息“Timer task executed.”。该任务被安排在最初的一秒延时后，恰好半秒钟运行一次。

```
// Demonstrate Timer and TimerTask.
```

```
import java.util.*;

class MyTimerTask extends TimerTask {
  public void run() {
    System.out.println("Timer task executed.");
  }
}

class TTest {
  public static void main(String args[]) {
    MyTimerTask myTask = new MyTimerTask();
    Timer myTimer = new Timer();

    /* Set an initial delay of 1 second,
       then repeat every half second.
    */
    myTimer.schedule(myTask, 1000, 500);

    try {
      Thread.sleep(5000);
    } catch (InterruptedException exc) {}

    myTimer.cancel();
  }
}
```

16.12 java.util.zip包

java.util.zip包提供了读、写流行的ZIP格式或GZIP格式文件的能力。ZIP和GZIP都输入和输出可用的数据流。另外一些类实现了用于压缩和解压缩的ZLIB算法。

16.13 java.util.jar包

java.util.jar包提供了读、写Java存档（JAR）文件的能力。在第25章将会看到，JAR文件被用来包含被称作Java Beans的软件及任何相关文件。

第 17 章　输入/输出：探究 java.io

本章探究为输入/输出操作提供支持的java.io包。在第12章，我们介绍了Java 的输入/输出系统。现在，我们要更深入的研究Java输入/输出系统。

所有的程序员都知道，多数程序在不获取外部数据的情况下不能顺利完成目标。数据从一个输入源获得。程序的结果被送到输出目的地。Java中，这些源和目的地被广泛的定义。例如一个网络连接器，内存缓冲区或磁盘文件可以被Java输入/输出类熟练的操作。尽管从物理上很难说明，这些外设都由相同的抽象体流（stream）来处理。流，在第12章解释过，是一个生产或消费信息的逻辑实体。流通过Java输入/输出系统与物理设备相连。尽管与之相连的实际的物理设备各不相同，所有的流都以同样的方式运转。

注意：Java流式输入/输出的概述，参看第12章。

17.1　Java输入/输出类和接口

java.io定义的输入/输出类列于下表：

BufferedInputStream	FileWriter	PipedInputStream
BufferedOutputStream	FilterInputStream	PipedOutputStream
BufferedReader	FilterOutputStream	PipedReader
BufferedWriter	FilterReader	PipedWriter
ByteArrayInputStream	FilterWriter	PrintStream
ByteArrayOutputStream	InputStream	PrintWriter
CharArrayReader	InputStreamReader	PushbackInputStream
CharArrayWriter	LineNumberReader	PushbackReader
DataInputStream	ObjectInputStream	RandomAccessFile
DataOutputStream	ObjectInputStream.GetField	Reader
File	ObjectOutputStream	SequenceInputStream
FileDescriptor	ObjectOutputStream.PutField	SerializablePermission
FileInputStream	ObjectStreamClass	StreamTokenizer
FileOutputStream	ObjectStreamField	StringReader
FilePermission	OutputStream	StringWriter
FileReader	OutputStreamWriter	Writer

ObjectInputStream.GetField和ObjectOutputStream.PutField 是Java 2新添的内部类。java.io包还包含两个不受java 2欢迎的类，这两个类没有在上表中列出：LineNumberInputStream和StringBufferInputStream。新代码不应该使用两个类。

下面是由java.io定义的接口：

DataInput	FilenameFilter	ObjectOutput
DataOutput	ObjectInput	ObjectStreamConstants
Externalizable	ObjectInputValidation	Serializable
FileFilter		

FileFilter接口是Java 2新增的。

java.io包中有很多类和接口。包括字节和字符流，对象序列化（对象的存储和释放）。本章讲述几个最常用的I/O成员，从最独特的File开始。

17.2 File（文件类）

尽管java.io定义的大多数类是实行流式操作的，File类不是。它直接处理文件和文件系统。也就是说，File类没有指定信息怎样从文件读取或向文件存储；它描述了文件本身的属性。File对象用来获取或处理与磁盘文件相关的信息，例如权限，时间，日期和目录路径。此外，File还浏览子目录层次结构。

很多程序中文件是数据的根源和目标。尽管它们在小应用程序中因为安全原因而受到严格限制，文件仍是存储固定和共享信息的主要资源。Java中的目录当成File 对待，它具有附加的属性——一个可以被 list( )方法检测的文件名列表。

下面的构造函数可以用来生成File 对象：

```
File(String directoryPath)
File(String directoryPath, String filename)
File(File dirObj, String filename)
```

这里，directoryPath是文件的路径名，filename 是文件名，dirObj 一个指定目录的File 对象。

下面的例子创建了三个文件：f1，f2，和f3。第一个File对象是由仅有一个目录路径参数的构造函数生成的。第二个对象有两个参数——路径和文件名。第三个File对象的参数包括指向f1文件的路径及文件名。f3和f2指向相同的文件。

```
File f1 = new File("/");
File f2 = new File("/","autoexec.bat");
File f3 = new File(f1,"autoexec.bat");
```

注意：Java 能正确处理UNIX和Windows/DOS约定路径分隔符。如果在Windows版本的Java下用斜线（/），路径处理依然正确。记住，如果你用Windows/DOS使用反斜线（\）的约定，你需要在字符串内使用它的转义序列（\\）。Java约定是用UNIX和URL风格的斜线来作路径分隔符。

File 定义了很多获取File对象标准属性的方法。例如getName( )返回文件名，getParent( )返回父目录名，exists( )在文件存在的情况下返回true，反之返回false。然而File类是不对称的。说它不对称，意思是虽然存在允许验证一个简单文件对象属性的很多方法，但是没有

相应的函数来改变这些属性。下面的例子说明了几个File方法：

```
// Demonstrate File.
import java.io.File;

class FileDemo {
  static void p(String s) {
    System.out.println(s);
  }

  public static void main(String args[]) {
    File f1 = new File("/java/COPYRIGHT");
    p("File Name: " + f1.getName());
    p("Path: " + f1.getPath());
    p("Abs Path: " + f1.getAbsolutePath());
    p("Parent: " + f1.getParent());
    p(f1.exists() ? "exists" : "does not exist");
    p(f1.canWrite() ? "is writeable" : "is not writeable");
    p(f1.canRead() ? "is readable" : "is not readable");
    p("is " + (f1.isDirectory() ? "" : "not" + " a directory"));
    p(f1.isFile() ? "is normal file" : "might be a named pipe");
    p(f1.isAbsolute() ? "is absolute" : "is not absolute");
    p("File last modified: " + f1.lastModified());
    p("File size: " + f1.length() + " Bytes");
  }
}
```

运行该程序，你将看到下面的结果：

```
File Name: COPYRIGHT
Path: /java/COPYRIGHT
Abs Path: /java/COPYRIGHT
Parent: /java
exists
is writeable
is readable
is not a directory
is normal file
is absolute
File last modified: 812465204000
File size: 695 Bytes
```

大多数File方法是自说明的，但isFile( )和isAbsolute( )不是。isFile( )在被文件调用时返回true，在被目录调用时返回false。并且，isFile( )被一些专用文件调用时返回false，例如设备驱动程序和命名管道，所以该方法可用来判定文件是否作为文件执行。isAbsolute( )方法在文件拥有绝对路径时返回true，若是相对路径则返回false。

File 还包括两个有用的实用工具方法。第一个是renameTo( )，显示如下：

```
boolean renameTo(File newName)
```

这里，由newName指定的文件名变成了所调用的File 对象的新的名称。如果更名成功则返回ture，文件不能被重命名（例如，你试图重命名文件以使它从一个目录转到另一个目

录，或者你使用了一个已经存在的文件名），则返回false。

第二个实用工具方法是delete( )，该方法删除由被调用的File对象的路径指定的磁盘文件。它的形式如下：

```
boolean delete( )
```

同样可以在目录为空时用delete( )删除目录。如果删除了文件，delete( )返回true，如果文件不能被删除则返回false。

Java 2 为File 类增添了一些新的方法，你会发现在某些场合这些新增方法很有用。一些最有趣的方法显示如下：

方法	描述
void deleteOnExit( )	在java虚拟机终止时删除与调用对象相关的文件
boolean isHidden( )	如果调用的文件是隐藏的，返回true；否则返回 false。
boolean setLastModified(long millisec)	设置由millisec指定的调用文件的时间标志，Millisec是从1970年1月1号开始的标准时间(UTC)的毫秒数
boolean setReadOnly( )	设置调用文件为只读

并且，因为File 类现在支持Comparable 接口，compareTo( )方法也被支持。

17.2.1 目录

目录是一个包含其他文件和路径列表的File 类。当你创建一个File 对象且它是目录时，isDirectory( ) 方法返回ture。这种情况下，可以调用该对象的list( )方法来提取该目录内部其他文件和目录的列表。该方法有两种形式。第一种形式如下：

```
String[ ] list( )
```

文件列表在一个String 对象数组中返回。

下面显示的程序说明怎样用list( )来检查一个目录的内容：

```
// Using directories.
import java.io.File;

class DirList {
  public static void main(String args[]) {
    String dirname = "/java";
    File f1 = new File(dirname);

    if (f1.isDirectory()) {
      System.out.println("Directory of " + dirname);
      String s[] = f1.list();

      for (int i=0; i < s.length; i++) {
        File f = new File(dirname + "/" + s[i]);
        if (f.isDirectory()) {
          System.out.println(s[i] + " is a directory");
        } else {
          System.out.println(s[i] + " is a file");
```

```
        }
      }
    } else {
      System.out.println(dirname + " is not a directory");
    }
  }
}
```

下面是程序的样本输出（当然，目录下的内容不同，输出也不同）：

```
Directory of /java
bin is a directory
lib is a directory
demo is a directory
COPYRIGHT is a file
README is a file
index.html is a file
include is a directory
src.zip is a file
.hotjava is a directory
src is a directory
```

17.2.2　使用FilenameFilter

你总是希望能够限制由list( )方法返回的文件数目，使它仅返回那些与一定的文件名方式或者过滤(filter)相匹配的文件。为达到这样的目的，必须使用list( )的第二种形式：

```
String[ ] list(FilenameFilter FFObj)
```

该形式中，FFObj是一个实现FilenameFilter接口的类的对象。

FilenameFilter仅定义了一个方法，accept( )。该方法被列表中的每个文件调用一次。它的通常形式如下：

```
boolean accept(File directory, String filename)
```

当被directory 指定的目录中的文件（也就是说，那些与filename 参数匹配的文件）包含在列表中时，accept( )方法返回true ，当这些文件没有包括在列表中时，accept( )返回false。

下面显示的OnlyExt 类实现FilenameFilter接口，它被用来修饰前面的程序，限制由 list( )返回的文件名的可见度，把对象被创建时以指定扩展名结束的文件归档。

```
import java.io.*;

public class OnlyExt implements FilenameFilter {
  String ext;

  public OnlyExt(String ext) {
    this.ext = "." + ext;
  }

  public boolean accept(File dir, String name) {
    return name.endsWith(ext);
  }
}
```

修改过的目录列表程序显示如下。现在它只显示以.html 为扩展名的文件。

```
// Directory of .HTML files.
import java.io.*;

class DirListOnly {
  public static void main(String args[]) {
    String dirname = "/java";
    File f1 = new File(dirname);
    FilenameFilter only = new OnlyExt("html");
    String s[] = f1.list(only);

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

17.2.3 listFiles（）方法

Java 2增加了list( )方法的一个变化形式，名为listFiles( )，你会发现该方法很有用。listFiles( )形式如下：

```
File[ ] listFiles( )
File[ ] listFiles(FilenameFilter FFObj)
File[ ] listFiles(FileFilter FObj)
```

上述三种形式以File对象矩阵的形式返回文件列表，而不是用字符串形式返回。第一种形式返回所有的文件，第二种形式返回满足指定FilenameFilter接口的文件。除了返回一个File 对象数组，这两个listFiles( )方法就像list( )方法一样工作。

第三种listFiles( )形式返回满足指定FileFilter的路径名的文件。FileFilter只定义了一个accept( )方法，该方法被列表中的每个文件调用一次。它的通常形式如下：

```
boolean accept(File path)
```

如果文件被包括在列表中（即与path参数匹配的文件），accept( )方法返回true，如果不被包括，则返回false。

17.2.4 创建目录

另外两个有用的File类的方法是mkdir( )和mkdirs( )。mkdir( )方法创建了一个目录，创建成功返回true，创建失败返回false。创建失败是指File对象指定的目录已经存在，或者是因为整个路径不存在而不能创建目录。创建路径不存在的目录，用mkdirs( )的方法。它创建目录以及该目录所有的父目录。

17.3 流 类

Java 的流式输入/输出建立在四个抽象类的基础上：InputStream, OutputStream, Reader

和Writer。这些类在第12章中有过简要的讨论。它们用来创建具体流式子类。尽管程序通过具体子类执行输入/输出操作，顶层的类定义了所有流类的基础通用功能。

InputStream 和OutputStream 设计成字节流类。Reader 和Writer 为字符流设计。字节流类和字符流类形成分离的层次结构。一般说来，处理字符或字符串时应使用字符流类，处理字节或二进制对象时应用字节流类。

下面分别讲述字节流和字符流类。

17.4　字 节 流

字节流类为处理字节式输入/输出提供了丰富的环境。一个字节流可以和其他任何类型的对象并用，包括二进制数据。这样的多功能性使得字节流对很多类型的程序都很重要。因为字节流类以InputStream 和OutputStream为顶层，我们就从讨论这两个类开始。

17.4.1　InputStream（输入流）

InputStream 是一个定义了Java流式字节输入模式的抽象类。该类的所有方法在出错条件下引发一个IOException 异常。表17-1显示了InputStream的方法。

表 17-1　InputStream 定义的方法

方法	描述
int available( )	返回当前可读的输入字节数
void close( )	关闭输入源。关闭之后的读取会产生IOException异常
void mark(int numBytes)	在输入流的当前点放置一个标记。该流在读取numBytes个字节前都保持有效
boolean markSupported( )	如果调用的流支持mark( )/reset( )就返回true
int read( )	如果下一个字节可读则返回一个整型，遇到文件尾时返回-1
int read(byte buffer[ ])	试图读取buffer.length个字节到buffer中，并返回实际成功读取的字节数。遇到文件尾时返回-1
int read(byte buffer[ ], int offset, int numBytes)	试图读取buffer中从buffer[offset]开始的numBytes个字节，返回实际读取的字节数。遇到文件尾时返回-1
void reset( )	重新设置输入指针到先前设置的标志处
long skip(long numBytes)	忽略numBytes个输入字节，返回实际忽略的字节数

17.4.2　OutputStream（输出流）

OutputStream是定义了流式字节输出模式的抽象类。该类的所有方法返回一个void 值并且在出错情况下引发一个IOException异常。表17-2显示了OutputStream的方法。

表 17-2 OutputStream 定义的方法

方法	描述
void close( )	关闭输出流。关闭后的写操作会产生IOException异常
void flush( )	定制输出状态以使每个缓冲器都被清除，也就是刷新输出缓冲区
void write(int b)	向输出流写入单个字节。注意参数是一个整型数，它允许你不必把参数转换成字节型就可以调用write（）
void write(byte buffer[ ])	向一个输出流写一个完整的字节数组
void write(byte buffer[ ], int offset, int numBytes)	写数组buffer以buffer[offset]为起点的numBytes个字节区域内的内容

注意：多数在表17-1和表17-2中描述的方法由InputStream 和OutputStream的子类实现，但mark()和reset()方法除外。注意下面讨论的每个子类中它们的使用和不用情况。

17.4.3 FileInputStream（文件输入流）

FileInputStream 类创建一个能从文件读取字节的InputStream 类，它的两个常用的构造函数如下：

```
FileInputStream(String filepath)
FileInputStream(File fileObj)
```

它们都能引发FileNotFoundException异常。这里，filepath 是文件的全称路径，fileObj是描述该文件的File对象。

下面的例子创建了两个使用同样磁盘文件且各含一个上述构造函数的FileInputStreams类：

```
FileInputStream f0 = new FileInputStream("/autoexec.bat")
File f = new File("/autoexec.bat");
FileInputStream f1 = new FileInputStream(f);
```

尽管第一个构造函数可能更常用到，第二个构造函数允许在把文件赋给输入流之前用File方法更进一步检查文件。当一个FileInputStream被创建时，它可以被公开读取。FileInputStream重载了抽象类InputStream的六个方法，mark( )和reset( )方法不被重载，任何关于使用FileInputStream的reset()尝试都会生成IOException异常。

下面的例题说明了怎样读取单个字节、字节数组以及字节数组的子界。它同样阐述了怎样运用available( )判定剩余的字节个数及怎样用skip( )方法跳过不必要的字节。该程序读取它自己的源文件，该源文件必定在当前目录中。

```
// Demonstrate FileInputStream.
import java.io.*;

class FileInputStreamDemo {
  public static void main(String args[]) throws Exception {
```

```
    int size;
    InputStream f =
      new FileInputStream("FileInputStreamDemo.java");

    System.out.println("Total Available Bytes: " +
                       (size = f.available()));
    int n = size/40;
    System.out.println("First " + n +
                       " bytes of the file one read() at a time");
    for (int i=0; i < n; i++) {
      System.out.print((char) f.read());
    }
    System.out.println("\nStill Available: " + f.available());
    System.out.println("Reading the next " + n +
                       " with one read(b[])");
    byte b[] = new byte[n];
    if (f.read(b) != n) {
      System.err.println("couldn't read " + n + " bytes.");
    }
    System.out.println(new String(b, 0, n));
    System.out.println("\nStill Available: " + (size = f.available()));
    System.out.println("Skipping half of remaining bytes with skip()");
    f.skip(size/2);
    System.out.println("Still Available: " + f.available());
    System.out.println("Reading " + n/2 + " into the end of array");
    if (f.read(b, n/2, n/2) != n/2) {
      System.err.println("couldn't read " + n/2 + " bytes.");
    }
    System.out.println(new String(b, 0, b.length));
    System.out.println("\nStill Available: " + f.available());
    f.close();
  }
}
```

下面是该程序的输出：

```
Total Available Bytes: 1433
First 35 bytes of the file one read() at a time
// Demonstrate FileInputStream.
im
Still Available: 1398
Reading the next 35 with one read(b[])
port java.io.*;

class FileInputS

Still Available: 1363
Skipping half of remaining bytes with skip()
Still Available: 682
Reading 17 into the end of array
port java.io.*;
read(b) != n) {
S

Still Available: 665
```

这个有些刻意创作的例子说明了怎样读取数据的三种方法，怎样跳过输入以及怎样检查流中可以获得数据的数目。

17.4.4 FileOutputStream（文件输出流）

FileOutputStream 创建了一个可以向文件写入字节的类OutputStream，它常用的构造函数如下：

```
FileOutputStream(String filePath)
FileOutputStream(File fileObj)
FileOutputStream(String filePath, boolean append)
```

它们可以引发IOException或SecurityException异常。这里filePath是文件的全称路径，fileObj是描述该文件的File对象。如果append为true，文件以设置搜索路径模式打开。

FileOutputStream的创建不依赖于文件是否存在。在创建对象时FileOutputStream在打开输出文件之前创建它。这种情况下你试图打开一个只读文件，会引发一个IOException异常。

下面的例子创建一个样本字节缓冲器。先生成一个String对象，接着用getBytes( )方法提取字节数组对等体。然后创建了三个文件。第一个file1.txt将包括样本中的各个字节。第二个文件是file2.txt，它包括所有字节。第三个也是最后一个文件file3.txt，仅包含最后的四分之一。不像FileInputStream类的方法，所有FileOutputStream类的方法都返回一个void类型值。在出错情况下，这些方法将引发IOException异常。

```
// Demonstrate FileOutputStream.
import java.io.*;

class FileOutputStreamDemo {
  public static void main(String args[]) throws Exception {
    String source = "Now is the time for all good men\n"
      + " to come to the aid of their country\n"
      + " and pay their due taxes.";
    byte buf[] = source.getBytes();
    OutputStream f0 = new FileOutputStream("file1.txt");
    for (int i=0; i < buf.length; i += 2) {
      f0.write(buf[i]);
    }
    f0.close();

    OutputStream f1 = new FileOutputStream("file2.txt");
    f1.write(buf);
    f1.close();

    OutputStream f2 = new FileOutputStream("file3.txt");
    f2.write(buf,buf.length-buf.length/4,buf.length/4);
    f2.close();
  }
}
```

下面是运行该程序之后，每个文件的内容，首先是file1.txt：

```
Nwi h iefralgo e
t oet h i ftercuty n a hi u ae.
```

接着，是file2.txt：

```
Now is the time for all good men
 to come to the aid of their country
 and pay their due taxes.
```

最后，file3.txt

```
nd pay their due taxes.
```

17.4.5　ByteArrayInputStream（字节数组输入流）

ByteArrayInputStream是把字节数组当成源的输入流。该类有两个构造函数，每个构造函数需要一个字节数组提供数据源：

```
ByteArrayInputStream(byte array[ ])
ByteArrayInputStream(byte array[ ], int start, int numBytes)
```

这里，array是输入源。第二个构造函数创建了一个InputStream类，该类从字节数组的子集生成，以start指定索引的字符为起点，长度由numBytes决定。

下面的例子创建了两个ByteArrayInputStream，用字母表的字节表示初始化它们：

```
// Demonstrate ByteArrayInputStream.
import java.io.*;

class ByteArrayInputStreamDemo {
  public static void main(String args[]) throws IOException {
    String tmp = "abcdefghijklmnopqrstuvwxyz";
    byte b[] = tmp.getBytes();
    ByteArrayInputStream input1 = new ByteArrayInputStream(b);
    ByteArrayInputStream input2 = new ByteArrayInputStream(b, 0,3);
  }
}
```

input1对象包含整个字母表中小写字母，input2仅包含开始的三个字母。

ByteArrayInputStream实现mark( )和reset( )方法。然而，如果 mark( )不被调用，reset( )在流的开始设置流指针——该指针是传递给构造函数的字节数组的首地址。下面的例子说明了怎样用reset( )方法两次读取同样的输入。这种情况下，我们读取数据，然后分别用小写和大写字母打印“abc”。

```
import java.io.*;

class ByteArrayInputStreamReset {
  public static void main(String args[]) throws IOException {
    String tmp = "abc";
    byte b[] = tmp.getBytes();
    ByteArrayInputStream in = new ByteArrayInputStream(b);

    for (int i=0; i<2; i++) {
      int c;
      while ((c = in.read()) != -1) {
        if (i == 0) {
```

```
          System.out.print((char) c);
        } else {
          System.out.print(Character.toUpperCase((char) c));
        }
      }
      System.out.println();
      in.reset();
    }
  }
}
```

该例先从流中读取每个字符，然后以小写字母形式打印。然后重新设置流并从头读起，这次在打印之前先将字母转换成大写字母。下面是输出：

```
abc
ABC
```

17.4.6 ByteArrayOutputStream（字节数组输出流）

ByteArrayOutputStream是一个把字节数组当作输出流的实现。ByteArrayOutputStream有两个构造函数，如下：

```
ByteArrayOutputStream( )
ByteArrayOutputStream(int numBytes)
```

在第一种形式里，一个32位字节的缓冲器被生成。第二个构造函数生成一个跟指定numBytes相同位数的缓冲器。缓冲器保存在ByteArrayOutputStream的受保护的buf 成员里。缓冲器的大小在需要的情况下会自动增加。缓冲器保存的字节数是由ByteArrayOutputStream的受保护的count域保存的。

下面的例子说明了ByteArrayOutputStream：

```
// Demonstrate ByteArrayOutputStream.
import java.io.*;

class ByteArrayOutputStreamDemo {
  public static void main(String args[]) throws IOException {
    ByteArrayOutputStream f = new ByteArrayOutputStream();
    String s = "This should end up in the array";
    byte buf[] = s.getBytes();

    f.write(buf);
    System.out.println("Buffer as a string");
    System.out.println(f.toString());
    System.out.println("Into array");
    byte b[] = f.toByteArray();
    for (int i=0; i<b.length; i++) {
      System.out.print((char) b[i]);
    }
    System.out.println("\nTo an OutputStream()");
    OutputStream f2 = new FileOutputStream("test.txt");

    f.writeTo(f2);
    f2.close();
```

```
    System.out.println("Doing a reset");
    f.reset();
    for (int i=0; i<3; i++)
      f.write('X');
    System.out.println(f.toString());
  }
}
```

运行程序后，生成下面的输出。注意在调用reset( )之后，三个X怎样结束。

```
Buffer as a string
This should end up in the array
Into array
This should end up in the array
To an OutputStream()
Doing a reset
XXX
```

该例用 writeTo( )这一便捷的方法将f 的内容写入test.txt，检查在前面例子中生成的test.txt文件内容，结果如下：

```
This should end up in the array
```

17.4.7　过滤字节流

过滤流（filtered stream）仅仅是底层透明地提供扩展功能的输入流（输出流）的包装。这些流一般由普通类的方法（即过滤流的一个超类）访问。典型的扩展是缓冲，字符转换和原始数据转换。这些过滤字节流是FilterInputStream 和FilterOutputStream。它们的构造函数如下：

```
FilterOutputStream(OutputStream os)
FilterInputStream(InputStream is)
```

这些类提供的方法和InputStream 及OutputStream类的方法相同。

17.4.8　缓冲字节流

对于字节流，缓冲流（buffered stream），通过把内存缓冲器连到输入/输出流扩展一个过滤流类。该缓冲器允许Java对多个字节同时进行输入/输出操作，提高了程序性能。因为缓冲器可用，所以可以跳过、标记和重新设置流。缓冲字节流类是BufferedInputStream 和BufferedOutputStream。PushbackInputStream 也可实现缓冲流。

BufferedInputStream（缓冲输入流）

缓冲输入/输出是一个非常普通的性能优化。Java 的BufferedInputStream 类允许把任何InputStream 类“包装”成缓冲流并使它的性能提高。

BufferedInputStream 有两个构造函数：

```
BufferedInputStream(InputStream inputStream)
BufferedInputStream(InputStream inputStream, int bufSize)
```

第一种形式生成了一个默认缓冲长度的缓冲流。第二种形式缓冲器大小是由bufSize传入的。使用内存页或磁盘块等的若干倍的缓冲区大小可以给执行性能带来很大的正面影响。但这是依赖于执行情况的。最理想的缓冲长度一般与主机操作系统、可用内存空间及机器配置有关。合理利用缓冲不需要特别复杂的操作。一般缓冲大小为8192个字节，给输入/输出流设定一个更小的缓冲器通常是好的方法。用这样的方法，低级系统可以从磁盘或网络读取数据块并在缓冲器中存储结果。因此，即使你在InputStream外同时读取字节数据时，也可以在超过99.9%的时间里获得快速存储操作。

缓冲一个输入流同样提供了在可用缓冲器的流内支持向后移动的必备基础。除了在任何InputStream类中执行的read( )和skip( )方法外，BufferedInputStream 同样支持mark( ) 和reset( )方法。BufferedInputStream.markSupported( )返回true是这一支持的体现。

下面的例子设计了一种情形，该情形下，我们可以使用mark( )来记忆我们在输入流中的位置，然后用reset( )方法返回该位置。这个例子分析了HTML实体的引用为版权信息的情况。这个引用以一个&符号开始以分号（；）结束，没有任何空格。例子输入由两个& 符号来说明何处reset( )发生，何处不发生的情况。

```
// Use buffered input.
import java.io.*;

class BufferedInputStreamDemo {
  public static void main(String args[]) throws IOException {
    String s = "This is a &copy; copyright symbol " +
      "but this is &copy not.\n";
    byte buf[] = s.getBytes();
    ByteArrayInputStream in = new ByteArrayInputStream(buf);
    BufferedInputStream f = new BufferedInputStream(in);
    int c;
    boolean marked = false;

    while ((c = f.read()) != -1) {
      switch(c) {
      case '&':
        if (!marked) {
          f.mark(32);
          marked = true;
        } else {
          marked = false;
        }
        break;
      case ';':
        if (marked) {
          marked = false;
          System.out.print("(c)");
        } else
          System.out.print((char) c);
        break;
      case ' ':
        if (marked) {
          marked = false;
          f.reset();
```

```
            System.out.print("&");
          } else
            System.out.print((char) c);
          break;
        default:
          if (!marked)
             System.out.print((char) c);
          break;
         }
       }
     }
}
```

注意该例运用mark(32)，该方法保存接下来所读取的32个字节（这个数量对所有的实体引用都足够）。下面是程序的输出：

```
This is a (c) copyright symbol but this is &copy not.
```

警告：在缓冲器中使用 mark()是受限的。意思是说你只能给mark()定义一个小于流缓冲大小的参数。

BufferedOutputStream（缓冲输出流）

BufferedOutputStream与任何一个OutputStream相同，除了用一个另外的flush( ) 方法来保证数据缓冲器被写入到实际的输出设备。因为BufferedOutputStream是通过减小系统写数据的时间而提高性能的，可以调用flush( )方法生成缓冲器中待写的数据。

不像缓冲输入，缓冲输出不提供额外的功能，Java中输出缓冲器是为了提高性能的。下面是两个可用的构造函数：

```
BufferedOutputStream(OutputStream outputStream)
BufferedOutputStream(OutputStream outputStream, int bufSize)
```

第一种形式创建了一个使用512字节缓冲器的缓冲流。第二种形式，缓冲器的大小由bufSize参数传入。

PushbackInputStream（推回输入流）

缓冲的一个新颖的用法是实现推回（pushback）。Pushback用于输入流允许字节被读取然后返回（即“推回”）到流。PushbackInputStream类实现了这个想法。它提供了一种机制来“窥视”在没有受到破坏的情况下输入流生成了什么。

PushbackInputStream有两个构造函数：

```
PushbackInputStream(InputStream inputStream)
PushbackInputStream(InputStream inputStream, int numBytes)
```

第一种形式创建了一个允许一个字节推回到输入流的流对象。第二种形式创建了一个具有numBytes长度缓冲区的推回缓冲流。它允许多个字节推回到输入流。

除了具有与InputStream相同的方法，PushbackInputStream提供了unread( )方法，表示如下：

```
void unread(int ch)
void unread(byte buffer[ ])
void unread(byte buffer, int offset, int numChars)
```

第一种形式推回ch的低位字节，它将是随后调用read( )方法所返回的下一个字节。第二种形式返回buffer缓冲器中的字节。第三种形式推回buffer中从offset处开始的numChars个字节。如果在推回缓冲器为满时试图返回一个字节，IOException异常将被引发。

Java 2 对PushbackInputStream作了一些小的修改：它实现skip( )方法。

下面的例子演示一个编程语言解析器怎样用PushbackInputStream和unread( )来处理==操作符和=操作符之间的不同的。

```
// Demonstrate unread().
import java.io.*;

class PushbackInputStreamDemo {
  public static void main(String args[]) throws IOException {
    String s = "if (a == 4) a = 0;\n";
    byte buf[] = s.getBytes();
    ByteArrayInputStream in = new ByteArrayInputStream(buf);
    PushbackInputStream f = new PushbackInputStream(in);
    int c;

    while ((c = f.read()) != -1) {
      switch(c) {
      case '=':
        if ((c = f.read()) == '=')
          System.out.print(".eq.");
        else {
          System.out.print("<-");
          f.unread(c);
        }
        break;
     default:
       System.out.print((char) c);
       break;
      }
    }
  }
}
```

下面是例子程序的输出。注意==被“.eq”代替而=被“<-”代替。

```
if (a .eq. 4) a <- 0;
```

注意：PushbackInputStream具有使InputStream生成的 mark() 或 reset()方法失效的副作用。用markSupported()来检查你运用mark()/reset()的任何流类。

17.4.9 SequenceInputStream（顺序输入流）

SequenceInputStream类允许连接多个InputStream流。SequenceInputStream的构造不同于任何其他的InputStream。SequenceInputStream构造函数要么使用一对InputStream，要么用

InputStream的一个Enumeration，显示如下：

```
SequenceInputStream(InputStream first, InputStream second)
SequenceInputStream(Enumeration streamEnum)
```

操作上来说，该类满足读取完第一个InputStream后转去读取第二个流的读取要求。使用Enumeration的情况下，它将继续读取所有InputStream流直到最后一个被读完。

下面是用SequenceInputStream输出两个文件内容的例子程序：

```
// Demonstrate sequenced input.
import java.io.*;
import java.util.*;

class InputStreamEnumerator implements Enumeration {
  private Enumeration files;
  public InputStreamEnumerator(Vector files) {
    this.files = files.elements();
  }

  public boolean hasMoreElements() {
    return files.hasMoreElements();
  }

  public Object nextElement() {
    try {
      return new FileInputStream(files.nextElement().toString());
    } catch (Exception e) {
      return null;
    }
  }
}

class SequenceInputStreamDemo {
  public static void main(String args[]) throws Exception {
    int c;
    Vector files = new Vector();

    files.addElement("/autoexec.bat");
    files.addElement("/config.sys");
    InputStreamEnumerator e = new InputStreamEnumerator(files);
    InputStream input = new SequenceInputStream(e);

    while ((c = input.read()) != -1) {
      System.out.print((char) c);
    }
    input.close();
  }
}
```

该例创建了一个Vector向量并向它添加了两个文件名。它把名字向量传给InputStreamEnumerator类，设计该类是为了提供向量包装器，向量返回的元素不是文件名，而是用这些名称打开FileInputStream流。SequenceInputStream依次打开每个文件，该程序打印了两个文件的内容。

17.4.10 PrintStream（打印流）

PrintStream具有本书开始以来我们在System文件句柄使用过的System.out所有的格式化性能。PrintStream有两个构造函数：

```
PrintStream(OutputStream outputStream)
PrintStream(OutputStream outputStream, boolean flushOnNewline)
```

当flushOnNewline控制Java每次刷新输出流时，输出一个换行符（\n）。如果flushOnNewline为true，自动刷新。若为false，刷新不能自动进行。第一个构造函数不支持自动刷新。

Java的PrintStream对象支持包括Object在内的各种类型的 print( ) 和 println( )方法。如果参数不是一个简单类型，PrintStream方法将调用对象的 toString( ) 方法，然后打印结果。

17.4.11 RandomAccessFile（随机访问文件类）

RandomAccessFile包装了一个随机访问的文件。它不是派生于InputStream和OutputStream，而是实现定义了基本输入/输出方法的DataInput和DataOutput接口。它同样支持定位请求——也就是说，可以在文件内部放置文件指针。它有两个构造函数：

```
RandomAccessFile(File fileObj, String access)
   throws FileNotFoundException
RandomAccessFile(String filename, String access)
   throws FileNotFoundException
```

第一种形式，fileObj指定了作为File 对象打开的文件的名称。第二种形式，文件名是由filename参数传入的。两种情况下，access 都决定允许访问何种文件类型。如果是“r”，那么文件可读不可写，如果是“rw”，文件以读写模式打开。

下面所示的seek( )方法，用来设置文件内部文件指针的当前位置：

```
void seek(long newPos) throws IOException
```

这里，newPos 指文件指针从文件开始以字节方式指定新位置。调用seek( )方法后，接下来的读或写操作将在文件的新位置发生。

RandomAccessFile 实现了用来读写随机访问文件的标准的输入和输出方法。下面是Java 2增添的新方法setLength( )。它有下面的形式：

```
void setLength(long len) throws IOException
```

该方法通过指定的len设置正在调用的文件的长度。该方法可以增长或缩短一个文件。如果文件被加长，增加的部分是未定义的。

17.5 字 符 流

尽管字节流提供了处理任何类型输入/输出操作的足够的功能，它们不能直接操作Unicode字符。既然Java的一个主要目的是支持“只写一次，到处运行”的哲学，包括直接

的字符输入 / 输出支持是必要的。本节将讨论几个字符输入 / 输出类。如前所述，字符流层次结构的顶层是Reader 和Writer 抽象类。我们将从它们开始。

注意：如第12章讨论过的，字符输入／输出类是在java 的1.1版本中新加的。由此，你仍然可以发现遗留下的程序代码在应该使用字符流时却使用了字节流。当遇到这种代码，最好更新它。

17.5.1　Reader

Reader是定义Java的流式字符输入模式的抽象类。该类的所有方法在出错情况下都将引发IOException 异常。表17-3给出了Reader类中的方法。

表 17-3　Reader 定义的方法

方法	描述
abstract void close( )	关闭输入源。进一步的读取将会产生IOException异常
void mark(int numChars)	在输入流的当前位置设立一个标志。该输入流在numChars个字符被读取之前有效
boolean markSupported( )	该流支持mark( )/reset( )则返回true
int read( )	如果调用的输入流的下一个字符可读则返回一个整型。遇到文件尾时返回-1
int read(char buffer[ ])	试图读取buffer中的buffer.length个字符，返回实际成功读取的字符数。遇到文件尾返回-1
abstract int read(char buffer[ ],int offset, int numChars)	试图读取buffer中从buffer[offset]开始的numChars个字符，返回实际成功读取的字符数。遇到文件尾返回-1
boolean ready( )	如果下一个输入请求不等待则返回true，否则返回false
void reset( )	设置输入指针到先前设立的标志处
long skip(long numChars)	跳过numChars个输入字符，返回跳过的字符数

17.5.2　Writer

Writer 是定义流式字符输出的抽象类。所有该类的方法都返回一个void 值并在出错条件下引发IOException 异常。表17-4 给出了Writer类中方法。

表 17-4　Writer 定义的方法

方法	描述
abstract void close( )	关闭输出流。关闭后的写操作会产生IOException异常
abstract void flush( )	定制输出状态以使每个缓冲器都被清除。也就是刷新输出缓冲
void write(int ch)	向输出流写入单个字符。注意参数是一个整型，它允许你不必把参数转换成字符型就可以调用write（）
void write(char buffer[ ])	向一个输出流写一个完整的字符数组

续表

方法	描述
abstract void write(char buffer[ ], int offset,int numChars)	向调用的输出流写入数组buffer以buffer[offset]为起点的numChars个字符区域内的内容
void write(String str)	向调用的输出流写str
void write(String str, int offset, int numChars)	写数组str中以制定的offset为起点的长度为numChars个字符区域内的内容

17.5.3 FileReader

FileReader类创建了一个可以读取文件内容的Reader类。它最常用的构造函数显示如下：

```
FileReader(String filePath)
FileReader(File fileObj)
```

每一个都能引发一个FileNotFoundException异常。这里，filePath是一个文件的完整路径，fileObj是描述该文件的File 对象。

下面的例子演示了怎样从一个文件逐行读取并把它输出到标准输入流。例子读它自己的源文件，该文件一定在当前目录。

```
// Demonstrate FileReader.
import java.io.*;

class FileReaderDemo {
  public static void main(String args[]) throws Exception {
    FileReader fr = new FileReader("FileReaderDemo.java");
    BufferedReader br = new BufferedReader(fr);
    String s;

    while((s = br.readLine()) != null) {
      System.out.println(s);
    }

    fr.close();
  }
}
```

17.5.4 FileWriter

FileWriter 创建一个可以写文件的Writer 类。它最常用的构造函数如下：

```
FileWriter(String filePath)
FileWriter(String filePath, boolean append)
FileWriter(File fileObj)
```

它们可以引发IOException或SecurityException异常。这里，filePath是文件的完全路径，fileObj是描述该文件的File对象。如果append为true，输出是附加到文件尾的。

FileWriter类的创建不依赖于文件存在与否。在创建文件之前，FileWriter将在创建对象

时打开它来作为输出。如果你试图打开一个只读文件，将引发一个IOException异常。

下面的例子是前面讨论FileOutputStream时用到例子的字符流形式的版本。它创建了一个样本字符缓冲器，开始生成一个String，然后用getChars( )方法提取字符数组。然后该例创建了三个文件。第一个file1.txt，包含例子中的隔个字符。第二个file2.txt，包含所有的字符。最后，第三个文件file3.txt，只含有最后的四分之一。

```
// Demonstrate FileWriter.
import java.io.*;

class FileWriterDemo {
  public static void main(String args[]) throws Exception {
    String source = "Now is the time for all good men\n"
      + " to come to the aid of their country\n"
      + " and pay their due taxes.";
    char buffer[] = new char[source.length()];
    source.getChars(0, source.length(), buffer, 0);

    FileWriter f0 = new FileWriter("file1.txt");
    for (int i=0; i < buffer.length; i += 2) {
      f0.write(buffer[i]);
    }
    f0.close();

    FileWriter f1 = new FileWriter("file2.txt");
    f1.write(buffer);
    f1.close();

    FileWriter f2 = new FileWriter("file3.txt");

    f2.write(buffer,buffer.length-buffer.length/4,buffer.length/4);
    f2.close();
  }
}
```

17.5.5　CharArrayReader

CharArrayReader 是一个把字符数组作为源的输入流的实现。该类有两个构造函数，每一个都需要一个字符数组提供数据源：

```
CharArrayReader(char array[ ])
CharArrayReader(char array[ ], int start, int numChars)
```

这里，array是输入源。第二个构造函数从你的字符数组的子集创建了一个Reader，该子集以start指定的索引开始，长度为numChars。

下面的例子用到了上述CharArrayReader的两个构造函数：

```
// Demonstrate CharArrayReader.
import java.io.*;

public class CharArrayReaderDemo {
  public static void main(String args[]) throws IOException {
    String tmp = "abcdefghijklmnopqrstuvwxyz";
```

```
    int length = tmp.length();
    char c[] = new char[length];

    tmp.getChars(0, length, c, 0);
    CharArrayReader input1 = new CharArrayReader(c);
    CharArrayReader input2 = new CharArrayReader(c, 0, 5);

    int i;
    System.out.println("input1 is:");
    while((i = input1.read()) != -1) {
      System.out.print((char)i);
    }
    System.out.println();

    System.out.println("input2 is:");
    while((i = input2.read()) != -1) {
      System.out.print((char)i);
    }
    System.out.println();
  }
}
```

input1对象由全部的小写字母构造，而input2 值包含最初的5个字符。下面是输出：

```
input1 is:
abcdefghijklmnopqrstuvwxyz
input2 is:
abcde
```

17.5.6 CharArrayWriter

CharArrayWriter 实现了以数组作为目标的输出流。CharArrayWriter 有两个构造函数：

```
CharArrayWriter( )
CharArrayWriter(int numChars)
```

第一种形式，创建了一个默认长度的缓冲器。第二种形式，缓冲器长度由numChars指定。缓冲器保存在CharArrayWriter的buf 成员中。缓冲器大小在需要的情况下可以自动增长。缓冲器保持的字符数包含在CharArrayWriter的count 成员中。buf 和count 都是受保护的域。

下面的例子阐述了CharArrayWriter，我们继续使用前面显示的ByteArrayOutputStream例子中演示的程序。它的输出与以前的例子输出相同：

```
// Demonstrate CharArrayWriter.
import java.io.*;

class CharArrayWriterDemo {
  public static void main(String args[]) throws IOException {
    CharArrayWriter f = new CharArrayWriter();
    String s = "This should end up in the array";
    char buf[] = new char[s.length()];

    s.getChars(0, s.length(), buf, 0);
```

```
    f.write(buf);
    System.out.println("Buffer as a string");
    System.out.println(f.toString());
    System.out.println("Into array");

    char c[] = f.toCharArray();
    for (int i=0; i<c.length; i++) {
      System.out.print(c[i]);
    }

    System.out.println("\nTo a FileWriter()");
    FileWriter f2 = new FileWriter("test.txt");
    f.writeTo(f2);
    f2.close();
    System.out.println("Doing a reset");
    f.reset();
    for (int i=0; i<3; i++)
      f.write('X');
    System.out.println(f.toString());
  }
}
```

17.5.7　BufferedReader

BufferedReader 通过缓冲输入提高性能。它有两个构造函数：

```
BufferedReader(Reader inputStream)
BufferedReader(Reader inputStream, int bufSize)
```

第一种形式创建一个默认缓冲器长度的缓冲字符流。第二种形式，缓冲器长度由bufSize传入。

和字节流的情况相同，缓冲一个输入字符流同样提供支持可用缓冲器中流内反向移动的基础。为支持这点，BufferedReader 实现了mark( )和reset( )方法，并且BufferedReader.markSupported( ) 返回true.。

下面的例子改写了前面的BufferedInputStream例子，它用一个BufferedReader字符流而不是用一个缓冲字节流。和以前一样，它用mark( )和reset( )方法解析一个作为版权记号的HTML实体引用的流。这样的引用以&符号开始，以分号（；）结束，没有任何空格。例子输入有两个&字符，用来显示何处reset( )发生，何处不发生的情况。输出与前面的输出相同。

```
// Use buffered input.
import java.io.*;

class BufferedReaderDemo {
  public static void main(String args[]) throws IOException {
    String s = "This is a &copy; copyright symbol " +
      "but this is &copy not.\n";
    char buf[] = new char[s.length()];
    s.getChars(0, s.length(), buf, 0);
    CharArrayReader in = new CharArrayReader(buf);
    BufferedReader f = new BufferedReader(in);
    int c;
    boolean marked = false;
```

```
    while ((c = f.read()) != -1) {
      switch(c) {
      case '&':
        if (!marked) {
          f.mark(32);
          marked = true;
        } else {
          marked = false;
        }
        break;
      case ';':
        if (marked) {
          marked = false;
          System.out.print("(c)");
        } else
          System.out.print((char) c);
        break;
      case ' ':
        if (marked) {
          marked = false;
          f.reset();
          System.out.print("&");
        } else
          System.out.print((char) c);
        break;
       default:
         if (!marked)
            System.out.print((char) c);
         break;
       }
     }
   }
}
```

17.5.8 BufferedWriter

BufferedWriter是一个增加了flush( )方法的Writer。flush( )方法可以用来确保数据缓冲器确实被写到实际的输出流。用BufferedWriter 可以通过减小数据被实际的写到输出流的次数而提高程序的性能。

BufferedWriter有两个构造函数：

```
BufferedWriter(Writer outputStream)
BufferedWriter(Writer outputStream, int bufSize)
```

第一种形式创建了使用默认大小缓冲器的缓冲流。第二种形式中，缓冲器大小是由bufSize参数传入的。

17.5.9 PushbackReader

PushbackReader类允许一个或多个字符被送回输入流。这使你可以对输入流进行预测。下面是它的两个构造函数：

```
PushbackReader(Reader inputStream)
PushbackReader(Reader inputStream, int bufSize)
```

第一种形式创建了一个允许单个字节被推回的缓冲流。第二种形式，推回缓冲器的大小由bufSize参数传入。

PushbackReader 提供了unread( )方法。该方法返回一个或多个字符到调用的输入流。它有下面的三种形式：

```
void unread(int ch)
void unread(char buffer[ ])
void unread(char buffer[ ], int offset, int numChars)
```

第一种形式推回ch传入的字符。它是被并发调用的read( )返回的下一个字符。第二种形式返回buffer中的字符。第三种形式推回buffer中从offset开始的numChars 个字符。如果在推回缓冲器为满的条件下试图返回一个字符，一个IOException异常将被引发。

下面的例子重写了前面的PushBackInputStream例子，用PushbackReader代替了PushBackInputStream。和以前一样，它演示了一个编程语言解析器怎样用一个推回流处理用于比较的==操作符和用于赋值的=操作符之间的不同。

```
// Demonstrate unread().
import java.io.*;

class PushbackReaderDemo {
  public static void main(String args[]) throws IOException {
    String s = "if (a == 4) a = 0;\n";
    char buf[] = new char[s.length()];
    s.getChars(0, s.length(), buf, 0);
    CharArrayReader in = new CharArrayReader(buf);
    PushbackReader f = new PushbackReader(in);
    int c;

    while ((c = f.read()) != -1) {
      switch(c) {
      case '=':
        if ((c = f.read()) == '=')
          System.out.print(".eq.");
        else {
          System.out.print("<-");
          f.unread(c);
        }
        break;
      default:
        System.out.print((char) c);
        break;
      }
    }
  }
}
```

17.5.10　PrintWriter

PrintWriter本质上是PrintStream的字符形式的版本。它提供格式化的输出方法print( )和

println( )。PrintWriter有四个构造函数：

```
PrintWriter(OutputStream outputStream)
PrintWriter(OutputStream outputStream, boolean flushOnNewline)
PrintWriter(Writer outputStream)
PrintWriter(Writer outputStream, boolean flushOnNewline)
```

flushOnNewline控制Java是否在每次输出换行符（\n）时刷新输出流。如果flushOnNewline为true，刷新自动发生。若为false，不进行自动刷新。第一个和第三个构造函数不能自动刷新。

Java的PrintWriter对象支持包括用于Object在内的各种类型的print( )和println( )方法。如果语句不是一个简单类型，PrintWriter的方法将调用对象的toString（）方法，然后输出结果。

17.6 使用流式输入/输出

下面的例子演示了几个Java的输入/输出字符流类和方法。该程序执行标准wc（字数统计）命令。程序有两个模式：如果没有语句提供的文件名存在，程序对标准输入流进行操作。如果一个或多个文件名被指定，程序对每一个文件进行操作。

```
// A word counting utility.
import java.io.*;

class WordCount {
  public static int words = 0;
  public static int lines = 0;
  public static int chars = 0;

  public static void wc(InputStreamReader isr)
    throws IOException {
    int c = 0;
    boolean lastWhite = true;
    String whiteSpace = " \t\n\r";

    while ((c = isr.read()) != -1) {
      // Count characters
      chars++;
      // Count lines
      if (c == '\n') {
        lines++;
      }
      // Count words by detecting the start of a word
      int index = whiteSpace.indexOf(c);
      if(index == -1) {
        if(lastWhite == true) {
          ++words;
        }
        lastWhite = false;
      }
      else {
```

```
        lastWhite = true;
      }
    }
    if(chars != 0) {
      ++lines;
    }
  }

  public static void main(String args[]) {
    FileReader fr;
    try {
      if (args.length == 0) { // We're working with stdin
        wc(new InputStreamReader(System.in));
      }
      else { // We're working with a list of files
        for (int i = 0; i < args.length; i++) {
          fr = new FileReader(args[i]);
          wc(fr);
        }
      }
    }
    catch (IOException e) {
      return;
    }
    System.out.println(lines + " " + words + " " + chars);
  }
}
```

wc( )方法对任何输入流进行操作并且计算字符数，行数和字数。它在lastNotWhite里追踪字数的奇偶和空格。

当在没有参数的情况下执行时，WordCount以System.in为源流生成一个InputStream Reader对象。该流然后被传递到实际计数的 wc( )方法。当在有一个或多个参数的情况下执行时，WordCount 假设这些文件名存在并给每一个文件创建FileReader，传递保存结果的FileReader对象给wc( ) 方法。两种情况下，在退出之前都打印结果。

17.6.1　用StreamTokenizer （流标记）来改善wc( )

在输入流中一个更好的寻找模式的方法使用Java的另一个输入/输出类：Stream Tokenizer。与第16章的StringTokenizer相似，StreamTokenizer把InputStream拆解到被字符组界定的标记（token）中。它有下面的构造函数：

```
StreamTokenizer(Reader inStream)
```

这里，inStream必须具有Reader的某种形式。

StreamTokenizer定义了几个方法。该例中，我们仅用到少数几个。为重置分隔符的默认设置，我们使用 resetSyntax( )方法。分隔符的默认设置与标记表征的Java程序完美和谐，而且是为该例专用的。我们规定我们的标记，即“字”，是两边都有空格的明显字符组成的连续的字符串。

我们用eolIsSignificant（）来保证换行符作为标记被传递，所以我们可以和计算字数一

样计算行数。它的通常形式如下：

```
void eolIsSignificant(boolean eolFlag)
```

如果eolFlag为true，行结束符作为标记返回；若为false，行结束符被忽略。

wordChars( )方法用来指定可以用于字的字符范围。它的通常形式如下：

```
void wordChars(int start, int end)
```

这里，start和end指定了有效字符的范围。程序中，从33到255范围内的字符都是有效字符。

空格符由 whitespaceChars( )说明。它的一般形式如下：

```
void whitespaceChars(int start, int end)
```

这里，start和end指定了有效空格符的范围。

下一个标记通过调用nextToken( )从输入流获得，它返回标记的类型。

StreamTokenizer定义个四个int型常量：TT_EOF，TT_EOL，TT_NUMBER和TT_WORD。有三个实例变量。nval是一个公开的double 型变量，用来保存可识别的字数的值。sval是一个public String 型变量，用来保存可识别的的字的值。ttype是一个int型变量，说明刚刚被nextToken( )方法读取的标记的类型。如果标记是一个字，ttype等于TT_WORD。如果标记为一个数，ttype等于TT_NUMBER。如果标记是单一字符，ttype包含该字符的值。如果遇到一个行结束情况，ttype等于TT_EOL（这假定了参数为true调用eolIsSignificant（））。如果遇到流的结尾，ttype 等于TT_EOF。

用StreamTokenizer 修改过的字数计算程序显示如下：

```
// Enhanced word count program that uses a StreamTokenizer
import java.io.*;

class WordCount {
  public static int words=0;
  public static int lines=0;
  public static int chars=0;

  public static void wc(Reader r) throws IOException {
    StreamTokenizer tok = new StreamTokenizer(r);

    tok.resetSyntax();
    tok.wordChars(33, 255);
    tok.whitespaceChars(0, ' ');
    tok.eolIsSignificant(true);

    while (tok.nextToken() != tok.TT_EOF) {
      switch (tok.ttype) {
        case tok.TT_EOL:
          lines++;
          chars++;
          break;
        case tok.TT_WORD:
          words++;
```

```
        default: // FALLSTHROUGH
          chars += tok.sval.length();
          break;
      }
    }
  }

  public static void main(String args[]) {
    if (args.length == 0) { // We're working with stdin
      try {
        wc(new InputStreamReader(System.in));
        System.out.println(lines + " " + words + " " + chars);
      } catch (IOException e) {};
    } else { // We're working with a list of files
      int twords = 0, tchars = 0, tlines = 0;
      for (int i=0; i<args.length; i++) {
        try {
          words = chars = lines = 0;
          wc(new FileReader(args[i]));
          twords += words;
          tchars += chars;
          tlines += lines;
          System.out.println(args[i] + ": " +
            lines + " " + words + " " + chars);
        } catch (IOException e) {
          System.out.println(args[i] + ": error.");
        }
      }
      System.out.println("total: " +
      tlines + " " + twords + " " + tchars);
    }
  }
}
```

17.7　序 列 化

序列化（serialization）是把一个对象的状态写入一个字节流的过程。当你想要把你的程序状态存到一个固定的存储区域例如文件时，它是很管用的。稍后一点时间，你就可以运用序列化过程存储这些对象。

序列化也要执行远程方法调用（RMI）。RMI允许一台机器上的Java对象调用不同机器上的Java 对象方法。对象可以作为一个参数提供给那个远程方法。发送机序列化该对象并传送它。接受机反序列化它（关于RMI的更多内容请参看第24章）。

假设一个被序列化的对象引用了其他对象，同样，其他对象又引用了更多的对象。这一系列的对象和它们的关系形成了一个顺序图表。在这个对象图表中也有循环引用。也就是说，对象X可以含有一个对象Y的引用，对象Y同样可以包含一个对象X的引用。对象同样可以包含它们自己的引用。对象序列化和反序列化的工具被设计出来并在这一假定条件下运行良好。如果你试图序列化一个对象图表中顶层的对象，所有的其他的引用对象都被循环的定位和序列化。同样，在反序列化过程中，所有的这些对象以及它们的引用都被正

确的恢复。

下面是支持序列化的接口和类的概述。

17.7.1 Serializable接口

只有一个实现Serializable接口的对象可以被序列化工具存储和恢复。Serializable接口没有定义任何成员。它只用来表示一个类可以被序列化。如果一个类可以序列化，它的所有子类都可以序列化。

声明成transient的变量不被序列化工具存储。同样，static变量也不被存储。

17.7.2 Externalizable接口

Java的序列化和反序列化的工具被设计出来，所以很多存储和恢复对象状态的工作自动进行。然而，在某些情况下，程序员必须控制这些过程。例如，在需要使用压缩或加密技术时，Externalizable接口为这些情况而设计。

Externalizable 接口定义了两个方法：

```
void readExternal(ObjectInput inStream)
    throws IOException, ClassNotFoundException
void writeExternal(ObjectOutput outStream)
    throws IOException
```

这些方法中，inStream是对象被读取的字节流，outStream是对象被写入的字节流。

17.7.3 ObjectOutput接口

ObjectOutput 继承DataOutput接口并且支持对象序列化。它定义的方法显示于表17-5中。特别注意writeObject( )方法，它被称为序列化一个对象。所有这些方法在出错情况下引发IOException 异常。

表 17-5 ObjectOutput 定义的方法

方法	描述
void close( )	关闭调用的流。关闭后的写操作会产生IOException异常
void flush( )	定制输出状态以使每个缓冲器都被清除。也就是刷新输出缓冲区
void write(byte buffer[ ])	向调用的流写入一个字节数组
void write(byte buffer[ ], int offset, int numBytes)	写入数组buffer中从buffer[offset]位置开始的numBytes个字节长度区域内的数据
void write(int b)	向调用的流写入单个字节。写入的是b的低位字节
void writeObject(Object obj)	向调用的流写入obj对象

17.7.4 ObjectOutputStream类

ObjectOutputStream类继承OutputStream 类和实现ObjectOutput 接口。它负责向流写入对象。该类的构造函数如下：

```
ObjectOutputStream(OutputStream outStream) throws IOException
```

参数outStream 是序列化的对象将要写入的输出流。

该类中最常用的方法列于表17-6。它们在出错情况下引发IOException异常。Java 2给ObjectOuputStream增加了一个名为PutField的内部类。该类有助于持久域的编写，它的用法超出了本书的范围。

表 17-6　ObjectOutputStream 定义的常用方法

方法	描述
void close( )	关闭调用的流。关闭后的写操作会产生IOException异常
void flush( )	定制输出状态以使每个缓冲器都被清除。也就是刷新输出缓冲区
void write(byte buffer[ ])	向调用的流写入一个字节数组
void write(byte buffer[ ], int offset, int numBytes)	写入数组buffer中从buffer[offset]位置开始的numBytes个字节长度区域内的数据
void write(int b)	向调用的流写入单个字节。写入的是b的低位字节
void writeBoolean(boolean b)	向调用流写入一个布尔型值
void writebyte(int b)	向调用的流写入字节。写入的是b的低位字节
Void writeBytes（String str）	通过str向输入流写入字节
void writeChar(int c)	向调用流写入字符型值
void writeChars(String str)	通过str向调用流写入字符
void writeDouble(double d)	向调用流写入双精度值
void writeFloat(float f)	向调用流写入浮点数
void writeInt(int i)	向调用流写入整型数
void writeLong(long l)	向调用流写入长整型数
final void writeObject(Object obj)	向调用流写入obj
void writeShort(int i)	向调用流写入short型

17.7.5　ObjectInput

ObjectInput 接口继承DataInput接口并且定义了表17-7中的方法。它支持对象序列化。特别注意 readObject( )方法，它叫反序列化对象。所有这些方法在出错情况下引发IOException 异常。

表 17-7　ObjectInput 定义的方法

方法	描述
int available( )	返回输入缓冲区中现在可访问的字节数
void close( )	关闭调用流。关闭后的读取操作会产生IOException异常
int read( )	返回代表下一个输入字节的整数，遇到文件尾返回-1
int read(byte buffer[ ])	试图读取buffer中的buffer.length长度的字节，返回实际成功读取的字节数，遇到文件尾返回-1

续表

方法	描述
int read(byte buffer[ ], int offset, int numBytes)	试图读取buffer中buffer[offset]为起点的numBytes个字节，返回实际成功读取的字节数。遇到文件尾时返回-1
Object readObject( )	从调用流读取一个对象。
long skip(long numBytes)	忽略（跳过）调用流的numBytes个字节，返回实际忽略的字节数

17.7.6 ObjectInputStream

ObjectInputStream 继承InputStream 类并实现ObjectInput 接口。ObjectInputStream 负责从流中读取对象。该类的构造函数如下：

```
ObjectInputStream(InputStream inStream)
     throws IOException, StreamCorruptedException
```

参数inStream 是序列化对象将被读取的输入流。

该类中最常用的方法列于表17-8。它们在出错情况下将引发IOException异常。Java 2 为ObjectInputStream 增加了一个名为GetField的内部类。它帮助读取持久域，它的用途超出了本书的范围。并且，Java 2 中不被赞成的readLine( )方法不应该继续使用。

表 17-8 ObjectInputStream 定义的常用方法

方法	描述
int available( )	返回输入缓冲中现在可访问的的字节数
void close( )	关闭调用流。关闭后的读取操作会产生IOException异常
int read( )	返回表示下一个输入字节的整数，遇到文件尾返回-1
int read(byte buffer[ ], int offset, int numBytes)	试图读取buffer中buffer[offset]为起点的numBytes个字节，返回实际成功读取的字节数。遇到文件尾时返回-1
boolean readBoolean( )	从调用流读取并返回一个boolean型值
byte readByte( )	从调用流读取并返回一个byte型值
char readChar( )	从调用流读取并返回一个char型值
double readDouble( )	从调用流读取并返回一个double型值
float readFloat( )	从调用流读取并返回一个float型值
void readFully(byte buffer[ ])	读取buffer中buffer.length个字节。仅当所有字节被读取时会返回
void readFully(byte buffer[ ], int offset,int numBytes)	读取buffer中从buffer[offset]开始的numBytes个字节，仅当numBytes个字节被读取时有返回
int readInt( )	从调用流读取并返回一个int型值
long readLong( )	从调用流读取并返回一个long型值
final Object readObject( )	从调用流读取并返回一个对象
short readShort()	从调用流读取并返回一个short型值
int readUnsignedByte()	从调用流读取并返回一个无符号byte型值
int readUnsignedShort()	从调用流读取一个无符号short型值

17.7.7　序列化示例

下面的程序说明了怎样实现对象序列化和反序列化。它由实例化一个MyClass类的对象开始。该对象有三个实例变量，它们的类型分别是String，int和double。这是我们希望存储和恢复的信息。

FileOutputStream被创建，引用了一个名为“serial”的文件。为该文件流创建一个ObjectOutputStream。ObjectOutputStream 的writeObject( )方法用来序列化对象。对象的输出流被刷新和关闭。

然后，引用名为“serial”的文件创建一个FileInputStream类并为该文件创建一个ObjectInputStream类。ObjectInputStream 的readObject( )方法用来反序列化对象。然后对象输入流被关闭。

注意MyClass被定义成实现Serializable接口。如果不这样做，将会引发一个NotSerializableException异常。试图做一些把MyClass实例变量声明成transient的实验。那些数据在序列化过程中不被保存。

```
import java.io.*;

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

    // Object serialization
    try {
      MyClass object1 = new MyClass("Hello", -7, 2.7e10);
      System.out.println("object1: " + object1);
      FileOutputStream fos = new FileOutputStream("serial");
      ObjectOutputStream oos = new ObjectOutputStream(fos);
      oos.writeObject(object1);
      oos.flush();
      oos.close();
    }
    catch(Exception e) {
      System.out.println("Exception during serialization: " + e);
      System.exit(0);
    }

    // Object deserialization
    try {
      MyClass object2;
      FileInputStream fis = new FileInputStream("serial");
      ObjectInputStream ois = new ObjectInputStream(fis);
      object2 = (MyClass)ois.readObject();
      ois.close();
      System.out.println("object2: " + object2);
    }
    catch(Exception e) {
```

```
      System.out.println("Exception during deserialization: " + e);
      System.exit(0);
    }
  }
}

class MyClass implements Serializable {
  String s;
  int i;
  double d;
  public MyClass(String s, int i, double d) {
    this.s = s;
    this.i = i;
    this.d = d;
  }
  public String toString() {
    return "s=" + s + "; i=" + i + "; d=" + d;
  }
}
```

该程序说明了object1 和 object2 的实例变量是一样的。输出如下：

```
object1: s=Hello; i=-7; d=2.7E10
object2: s=Hello; i=-7; d=2.7E10
```

17.8 流 的 益 处

Java的输入输出的流式接口为复杂而繁重的任务提供了一个简洁的抽象。过滤流类的组合允许你动态建立客户端流式接口来配合数据传输要求。继承高级流类InputStream、InputStreamReader、Reader和Writer 类的Java程序在将来（即使创建了新的和改进的具体类）也能得到合理运用。就像你将在下一章看到的，这种模式在我们从基于文件系统的流转换到网络和套接字流时工作良好。最后，对象序列化有望在未来的Java编程中扮演一个越来越重要的角色。Java的序列化输入/输出类为这些有时显得十分复杂的任务提供了便携的解决方法。

第 18 章　网　络

本章讨论支持网络应用的java.net包。它的创建者称Java为“为网络编程”。尽管它只占Java的一小部分，与C++或FORTRAN相比，这种编程语言更有利于编写网络程序。使Java成为好的网络语言的正是java.net包定义的类。

这些类包装了由加州大学的伯克利分校的BSD引入的“套接字（socket）”范型。如果不简单介绍UNIX和BSD套接字的历史对Internet网络库的讨论就是不完整的。

18.1　网 络 基 础

1969年，Ken Thompson和Dennis Ritchie在Murray Hill，New Jersey的贝尔电话实验室开发了与C语言一致的UNIX。很多年来，UNIX的发展停留在贝尔实验室和一些大学及研究机构，用特意设计的DEC PDP机器运行。到了1978 年，Bill Joy在Cal Berkeley领导了一个项目，给UNIX增添新的特性，例如虚拟内存和全屏显示功能。到了1984年早期，当Bill正准备建立Sun Microsystems，它发明了4.2BSD，即众所周知的Berkeley UNIX。

4.2BSD带有快速文件系统、可靠信号处理、进程间通信以及最重要的网络功能。最先在4.2中发现的网络支持后来成为了实际的Internet标准。Berkeley的TCP/IP实现保留了在Internet内通信的最初的标准。进程间和网络通信的套接字范型被Berkeley以外的系统广泛采用。甚至Window和Macintosh在20世纪80年代晚期也开始和“Berkeley 套接字”谈话。

18.1.1　套接字概述

网络套接字（network socket）有一点像电源插座。网络周围的各式插头有一个标准方法传输它们的有效负载。理解标准协议的任何东西都能够插入套接字并进行通信。对于电源插座，不论你插入一个电灯或是烤箱，只要它们使用60HZ,115伏电压，设备将会工作。思考一下你的用电账单是怎样生成的。在你的房子和电网支架间可能有1米的距离，经过这一米的每千瓦电都将列入账单。账单到达你的“地址”。所以，虽然电流在电源插座周围是自由流动的，你房子的所有插头都是有特定的地址的。

除了我们谈论的是TCP/IP包和IP地址而不是电器和街道地址外，同样的思想被应用到网络套接字。Internet Protocol（IP）是一种低级路由协议。该协议将数据分解成小包然后通过网络传到一个地址，它并不确保传输的信息包一定到达目的。传输控制协议（TCP）是一种较高级的协议，它把这些信息包有力的捆绑在一起，在必要的时候，排序和重传这些信息包以获得可靠的数据传输。第三种协议，用户数据报协议（UDP）几乎与TCP协议相当，并能够直接用来支持快速的、无连接的、不可靠的信息包传输。

18.1.2 客户/服务器模式

你经常在与网络有关的话题中听说客户/服务器（client/server）这个术语。在一些产品说明中，这个概念似乎非常复杂，其实它的含义很简单。 服务器（server）就是能够提供共享资源的任何东西。现在有计算服务器，提供计算功能；打印服务器，管理多个打印机；磁盘服务器，提供联网的磁盘空间；以及Web服务器，用来存储网页。客户（client）是简单的任何有权访问特定服务器的实体。客户和服务器之间的连接就像电灯和电源插头的连接。房间的电源插座是服务器，电灯是客户。服务器是永久的资源，在访问过服务器之后，客户可以自由的“拔去插头”。

在Berkeley套接字中，套接字的概念允许单个计算机同时服务于很多不同的客户，并能够提供不同类型信息的服务。该种技术由引入的端口(port)处理，此端口是一个特定机器上的被编号的套接字。服务器进程是在“监听”端口直到客户连到它。尽管每个客户部分是独特的，一个服务器允许在同样端口接受多个客户。为管理多个客户连接，服务器进程必须是多线程的，或者有同步输入/输出处理多路复用技术的其他方法。

18.1.3 保留套接字

一旦连接成功，一个高级的协议跟着生效，该协议与所使用的端口有关。TCP/IP 为特定协议保留了低端的1024 个端口。如果你在网络上冲浪有一些时间了，那么这中间的很多你已经很熟悉了。端口21是为FTP的，23是Telnet，25是为e-mail，79是为finger的，80是HTTP，119是为网络新闻的——等等。下面该轮到讲述每个协议决定客户如何与端口交互了。

举例来说，HTTP是网络浏览器及服务器用来传输超文本网页和图像的协议。它是基本网页浏览服务器的一个非常简单的协议。下面是它的工作原理。当一个客户向一个HTTP服务器请求一个文件时，即一个点击动作，它仅仅以一种特定格式向预先指定的端口打印文件名然后读回文件的内容。服务器同样对状态代码编号反应，告诉客户请求是否被执行以及原因。下面是一个例子。客户请求单个文件/index.html，服务器回应它已成功找到该文件并且把文件传输到客户：

服务器	客户
监听80端口	与端口80连接
接受连接	写“GET /index.html HTTP/1.0\n\n”
读取数据直到遇到第二个换行符（\n）	
知道GET是一个命令，HTTP/1.0是有效的协议	
读取名为/ index.html的本地文件	
写“HTTP/1.0 200 OK\n\n”	“200”意味着“文件来了”
向套接字复制文件内容	读取文件内容并显示
挂起	挂起

很明显，HTTP协议比该例显示的要复杂的多，但这是一个实际的和附近Web服务器的传输。

18.1.4　代理服务器

一个代理服务器（proxy server）以客户端协议与其他服务器通信。这在客户与服务器连接受到某些限制的情况下经常是必需的。这样，客户可以连接代理服务器，代理服务器没有这些限制并且代理服务器也会依次和客户通信。代理服务器具有过滤某些请求或缓存一些这样的请求的结果以备后用的额外功能。一个缓冲代理HTTP 服务器可用来减小局域网连向Internet的带宽要求。若一个流行网站的网址被成百上千个用户点击，代理服务器可以一次获得该网络服务器的流行网页，节省昂贵的Internet网络传输，同时为用户快速提供对这些网页的访问。

本章的后面部分，我们将实际建立一个完整的缓冲代理HTTP服务器。这个程序有趣的是它既是客户又是服务器。为服务于某些网页，它必须像客户那样向其他服务器获取被请求内容的一个拷贝。

18.1.5　Internet 编址

Internet上的每一台计算机都有一个地址。Internet地址是网络上标识每台计算机的惟一定义的数。IP地址有32位，我们通常把它们分成4个从0到255的，有点号（.）隔开的序列。这使它们易于记忆，因为它们不是随机指派的——它们是按层次结构指派的。最开始的字节定义了网络属于A、B、C、D或E哪个等级。多数Internet用户使用C级，因为有多于两百万的网络是在C类。C类网的开始字节从192到224，最后字节实际上标识了256之中可以上单个C类网的独立计算机。这种安排允许在C类网中可以有5亿的设备。

域名服务（DNS）

如果每台机器必须用数字作为它们的地址，Internet不是一个漫游的友好场所。例如，设想在广告的底部看见“http://192.9.9.1/”，这一定使人头昏脑胀。感谢上帝，存在一个与所有这些数字相伴的一个平行层次的名称的交换所，叫做域名服务（DNS）。就像IP地址中从左到右描绘网络层次的四个数字一样，Internet 地址的名称，域名，在名称空间从右到左描述了机器的地址。例如，www.osborne.com是在COM域（为美国商业企业保留）中的，叫做osborne（公司名称）的，www是Osborne的Web服务器的特定计算机的名称。www在意义上相应于IP地址的最右边的数字。

18.2　JAVA和网络

现在准备工作已经完成，让我们看看Java怎样和所有的这些网络概念相联系。Java 通过扩展第17章介绍的已有的流式输入/输出接口和增加在网络上建立输入/输出对象特性这两个方法支持TCP/IP。Java支持TCP和UDP协议族。TCP用于网络的可靠的流式输入/输出。UDP支持更简单的、快速的、点对点的数据报模式。

18.2.1　网络类和接口

java.net 包所包含的类如下：

Authenticator (Java 2)	JarURLConnection (Java 2)	SocketPermission
ContentHandler	MulticastSocket	URL
DatagramPacket	NetPermission	URLClassLoader (Java 2)
DatagramSocket	PasswordAuthentication (Java 2)	URLConnection
DatagramSocketImpl	ServerSocket	URLDecoder (Java 2)
HttpURLConnection	Socket	URLEncoder
InetAddress	SocketImpl	URLStreamHandler

java.net包的接口如下：

ContentHandlerFactory	SocketImplFactory	URLStreamHandlerFactory
FileNameMap	SocketOptions	DatagramSocketImplFactory (Java 2, v1.3中添加的)

下面我们将讨论主要的网络类并且一些应用例子。

18.3 InetAddress类

无论你是在打电话、发送邮件或建立与Internet的连接，地址是基础。InetAddress 类用来封装我们前面讨论的数字式的IP地址和该地址的域名。你通过一个IP主机名与这个类发生作用，IP主机名比它的IP地址用起来更简便更容易理解。InetAddress 类内部隐藏了地址数字。

18.3.1 工厂方法

InetAddress 类没有明显的构造函数。为生成一个InetAddress对象，必须运用一个可用的工厂方法。工厂方法（factory method）仅是一个类中静态方法返回一个该类实例的约定。这是在一个带有各种参数列表的重载构造函数完成的，当持有惟一方法名时可使结果更清晰。对于InetAddress，三个方法 getLocalHost( )、getByName( )以及getAllByName( )可以用来创建InetAddress的实例。三个方法显示如下：

```
static InetAddress getLocalHost( )
   throws UnknownHostException
static InetAddress getByName(String hostName)
   throws UnknownHostException
static InetAddress[ ] getAllByName(String hostName)
   throws UnknownHostException
```

getLocalHost( )仅返回象征本地主机的InetAddress 对象。getByName( )方法返回一个传给它的主机名的InetAddress。如果这些方法不能解决主机名，它们引发一个UnknownHostException异常。

在Internet上，用一个名称来代表多个机器是很常有的事。Web服务器中，也有方法提供一定程度的缩放。getAllByName( )工厂方法返回代表由一个特殊名称分解的所有地址的InetAddresses类数组。在不能把名称分解成至少一个地址时，它将引发一个UnknownHostException异常。

下面的例子打印了本地机的地址和名称以及两个著名的Internet网址：

```
// Demonstrate InetAddress.
import java.net.*;

class InetAddressTest
{
  public static void main(String args[]) throws UnknownHostException {
    InetAddress Address = InetAddress.getLocalHost();
    System.out.println(Address);
    Address = InetAddress.getByName("osborne.com");
    System.out.println(Address);
    InetAddress SW[] = InetAddress.getAllByName("www.nba.com");
    for (int i=0; i<SW.length; i++)
      System.out.println(SW[i]);
  }
}
```

下面是该程序的输出（当然，你所看到的结果可能有一些不同）：

```
default/206.148.209.138
osborne.com/198.45.24.130
www.nba.com/204.202.130.223
```

18.3.2　实例方法

InetAddress 类也有一些非静态的方法，列于下面，它们可以用于讨论过的方法返回的对象：

boolean equals(Object other)	如果对象具有和other相同的Internet地址则返回true。
byte[ ] getAddress( )	返回代表对象的Internert地址的以网络字节为顺序的有四个元素的字节数组。
String getHostAddress( )	返回代表与InetAddress对象相关的主机地址的字符串。
String getHostName( )	返回代表与InetAddress对象相关的主机名的字符串。
int hashCode( )	返回调用对象的散列码。
boolean isMulticastAddress( )	如果Internet地址是一个多播地址返回true；否则返回false。
String toString( )	返回主机名字符串和IP地址。

Internet地址在分层的缓存服务器系列中被找到。这意味着你的本地机可能像知道它自己和附近的服务器一样知道一个名称-IP地址的自动映射。对于其他名称，它可能向一个本地DNS服务器询问IP地址信息。如果那个服务器不含一个指定的地址，它可以到一个远程的站点去询问。这可以一路通到名为InterNIC（internic.net）的根服务器。该过程可能需要比较长的时间，所以结构化你的代码以使你在本地存储IP地址信息而不是重复向上查找信息是一个明智之举。

18.4 TCP/IP客户套接字

TCP/IP 套接字用于在主机和Internet之间建立可靠的、双向的、持续的、点对点的流式连接。一个套接字可以用来建立Java 的输入/输出系统到其他的驻留在本地机或Internet上的任何机器的程序的连接。

注意：小应用程序只建立回到下载它的主机的套接字连接。存在这个限制的原因是：穿过防火墙的小应用程序有权使用任何机器是很危险的事情。

Java中有两类TCP套接字。一种是服务器端的，另一种是客户端的。ServerSocket类设计成在等待客户建立连接之前不做任何事的“监听器”。Socket类为建立连向服务器套接字以及启动协议交换而设计。

一个Socket对象的创建隐式建立了一个客户和服务器的连接。没有显式的说明建立连接细节的方法或构造函数。下面是用来生成客户套接字的两个构造函数：

Socket(String hostName, int port)	创建一个本地主机与给定名称的主机和端口的套接字连接，可以引发一个UnknownHostException异常或IOException异常。
Socket(InetAddress ipAddress, int port)	用一个预先存在的InetAddress对象和端口创建一个套接字，可以引发IOException异常。

使用下面的方法，可以在任何时候检查套接字的地址和与之有关的端口信息：

InetAddress getInetAddress( )	返回和Socket对象相关的InetAddress。
Int getPort( )	返回与该Socket对象连接的远程端口。
Int getLocalPort( )	返回与该Socket连接的本地端口。

一旦Socket对象被创建，同样可以检查它获得访问与之相连的输入和输出流的权力。如果套接字因为网络的连接中断而失效，这些方法都能够引发一个IOException异常。

InputStream getInputStream( )	返回与调用套接字有关的InputStream类。
OutputStream getOutputStream( )	返回与调用套接字有关的OutputStream类。
void close( )	关闭InputStream和OutputStream。

18.4.1 Whois

下面的例子打开了一个InterNIC服务器上“whois”端口的连接，传输命令行语句到套接字，然后打印返回的数据。InterNIC将努力寻找作为已注册的Internet域名的参数，然后传输回IP地址和该地点的联系信息。

```
//Demonstrate Sockets.
import java.net.*;
import java.io.*;

class Whois {
```

```
  public static void main(String args[]) throws Exception {
    int c;
    Socket s = new Socket("internic.net", 43);
    InputStream in = s.getInputStream();
    OutputStream out = s.getOutputStream();
    String str = (args.length == 0 ? "osborne.com" : args[0]) + "\n";
    byte buf[] = str.getBytes();
    out.write(buf);
    while ((c = in.read()) != -1) {
      System.out.print((char) c);
    }
    s.close();
  }
}
```

例如，如果你在命令行键入osborne.com，你将会获得下面相似的结果：

```
Whois Server Version 1.3

Domain names in the .com, .net, and .org domains can now be registered
with many different competing registrars. Go to http://www.internic.net
for detailed information.

   Domain Name: OSBORNE.COM
   Registrar: NETWORK SOLUTIONS, INC.
   Whois Server: whois.networksolutions.com
   Referral URL: www.networksolutions.com
   Name Server: NS1.EPPG.COM
   Name Server: NS2.EPPG.COM
   Updated Date: 07-apr-2000

>>> Last update of whois database: Fri, 6 Oct 2000 10:03:36 EDT <<<

The Registry database contains ONLY .COM, .NET, .ORG, .EDU domains and
Registrars.
```

18.5　URL

最后的例子有一点含糊，因为现代Internet不是围绕老的协议，如whois、finger及FTP的，而是关于WWW（万维网）的。Web是一个由Web浏览器统一的高级协议和文件格式的松散集合。Web中最重要的一个方面是Tim Berners-Lee设计了一个定位所有网络资源的弹性方法。一旦你能够可靠的命名一个事物，它将成为一个功能强大的范型，统一资源定位（URL）就做这些。

URL提供了一个相当容易理解的形式来惟一确定或对Internet上的信息进行编址。URL是无所不在的；每一个浏览器用它们来识别Web上的信息。实际上，Web是用URL 和HTML为所有资源编址的同样陈旧的Internet 。在Java的网络类库中，URL 类为用URL在Internet上获取信息提供了一个简单的、简洁的用户编程接口（API）。

18.5.1 格式化（Format）

http://www.osborne.com/ 和 http://www.osborne.com:80/index.htm是URL 的两个例子。一个URL规范以四个元素为基础。第一个是所用到的协议，用冒号（：）来将它与定位符的其他部分相隔离。尽管现在所有的事情都通过HTTP(实际上，如果你在URL规范中不用“http://”，大多数浏览器都能正确执行)完成，但它不是惟一的协议，常见的协议有http、ftp、gopher和文件。第二个元素是主机名或所用主机的IP地址，这由左边的双斜线（//）和右边的单斜线（/）或可选冒号（：）限制。第三个成分，端口号，是可选的参数，由主机名左边的冒号（：）和右边的斜线（/）限制（它的默认端口为80，它是预定义的HTTP 端口；所以“:80”是多余的）。第四部分是实际的文件路径。多数HTTP服务器将给URL附加一个与目录资源相关的index.html或 index.htm 文件。所以，http://www.osborne.com/ 与 http://www.osborne.com/index.htm 是相同的。

Java的URL类有多个构造函数，每个都能引发一个MalformedURLException异常。一个常见形式是用与你在浏览器中看到的相同的字符串指定URL：

```
URL(String urlSpecifier)
```

下面的两个构造函数形式允许你把URL分裂成它的组成部分：

```
URL(String protocolName, String hostName, int port, String path)
URL(String protocolName, String hostName, String path)
```

另一个经常用到的构造函数允许你用一个已经存在的URL作为引用上下文，然后从该上下文中创建一个新的URL。尽管这听起来有些别扭，它实际上是很简单而有用的。

```
URL(URL urlObj, String urlSpecifier)
```

下面的例子中，我们为Osborne的下载页面创建一个URL，然后检查它的属性：

```
// Demonstrate URL.
import java.net.*;
class URLDemo {
  public static void main(String args[]) throws MalformedURLException {
    URL hp = new URL("http://www.osborne.com/download");

    System.out.println("Protocol: " + hp.getProtocol());
    System.out.println("Port: " + hp.getPort());
    System.out.println("Host: " + hp.getHost());
    System.out.println("File: " + hp.getFile());
    System.out.println("Ext:" + hp.toExternalForm());
  }
}
```

运行该程序，你将获得下面输出：

```
Protocol: http
Port: -1
Host: www.osborne.com
File: /download
Ext:http://www.osborne.com/download
```

注意端口是-1，这意味着该端口没有被明确设置。现在我们已经创建了一个URL对象，我们希望找回与之相连的数据。为获得URL的实际比特或内容信息，用它的openConnection( )方法从它创建一个URLConnection对象，如下：

```
url.openConnection()
```

openConnection( ) 有下面的常用形式：

```
URLConnection openConnection( )
```

与调用URL对象相关，它返回一个URLConnection对象。它可能引发IOException异常。

18.6　URLConnection类

URLConnection是访问远程资源属性的一般用途的类。如果你建立了与远程服务器之间的连接，你可以在传输它到本地之前用URLConnection来检察远程对象的属性。这些属性由HTTP协议规范定义并且仅对用HTTP协议的URL对象有意义。我们这里将验证URLConnection最有用的原理。

下面的例子中我们用一个URL对象的openConnection( )方法创建了一个URLConnection类，然后用它来检查文件的属性和内容：

```
// Demonstrate URLConnection.
import java.net.*;
import java.io.*;
import java.util.Date;

class UCDemo
{
  public static void main(String args[]) throws Exception {
    int c;
    URL hp = new URL("http://www.osborne.com");
    URLConnection hpCon = hp.openConnection();

    System.out.println("Date: " + new Date(hpCon.getDate()));
    System.out.println("Content-Type: " + hpCon.getContentType());
    System.out.println("Expires: " + hpCon.getExpiration());
    System.out.println("Last-Modified: " +
                       new Date(hpCon.getLastModified()));
    int len = hpCon.getContentLength();
    System.out.println("Content-Length: " + len);
    if (len > 0) {
      System.out.println("=== Content ===");
      InputStream input = hpCon.getInputStream();
      int i = len;
      while (((c = input.read()) != -1) && (--i > 0)) {
        System.out.print((char) c);
      }
      input.close();
    } else {
      System.out.println("No Content Available");
```

```
    }
  }
}
```

该程序建立了一个经过端口80通向www.osborne.com 的HTTP 连接。然后列出了标头值并检索内容。下面是输出的前几行：

```
Date: Fri Oct 06 14:17:10 CDT 2000
Content-Type: text/html
Expires: 0
Last-Modified: Tue Oct 26 01:36:57 CDT 1999
Content-Length: 529
=== Content ===
<html>
<head>
<title>Osborne/McGraw-Hill: Required Reading for the Information
Age</title>
</head>
```

URL和URLConnection类对于希望建立与HTTP服务器的连接来获取信息的简单程序来说是非常好的。对于更复杂的应用程序，你会发现学习HTTP协议规范，实现你自己的包装程序是比较好的。

18.7 TCP/IP服务器套接字

如我们在前面提到的，Java具有用来创建服务器应用程序的不同的套接字类。ServerSocket类用来创建服务器，服务器监听本地或远程客户程序通过公共端口的连接。既然Web驱动着Internet中的大部分活动，本节就开发一个可运行的Web服务器（http）。

ServerSocket与通常的Sockets类完全不同。当创建一个ServerSocket类，它在系统注册自己对客户连接感兴趣。ServerSocket的构造函数反映了希望接受连接的端口号及你希望排队等待上述端口的时间（该项可选）。队列长度告诉系统多少与之连接的客户在系统拒绝连接之前可以挂起。队列的默认长度是50。构造函数在不利情况下可以引发IOException异常。下面是构造函数：

ServerSocket(int port)	在指定端口创建队列长度为50的服务器套接字。
ServerSocket(int port, int maxQueue)	在指定端口创建一个最大队列长度为maxQueue的服务器套接字。
ServerSocket(int port, int maxQueue, InetAddress localAddress)	在指定端口创建一个最大队列长度为maxQueue的服务器套接字。在一个多地址主机上，localAddress指定该套接字约束的IP地址。

ServerSocket有一个额外的accept()方法，该方法是一个等待客户开始通信的模块化调用，然后以一个用来与客户通信的常规Socket返回。

18.8　缓存代理HTTP服务器

本节的剩余部分，我们讲述一个简单的缓存代理HTTP服务器，名为http，来演示客户与服务器套接字。http只支持GET操作及硬编码的MIME类型的一小部分（MIME类型是多媒体内容的类型描述符）。代理HTTP服务器是单线程的，该线程中每一个请求依次被处理，其他请求等待。这是缓存的相当天真的策略——它在RAM永久保存所有信息。http作为一个代理服务器时，它还拷贝每一个它获取的文件到本地缓存中。对于本地缓存，它没有用于刷新和无用单元回收的策略。除此之外，http代表了客户和服务器套接字的一个多产的例子，它是值得探究和容易扩展的。

18.8.1　源代码

HTTP服务器是通过5个类和一个接口实现的。更完善的实现方案可能在主类httpd外分裂很多的方法，以使组成结构更抽象。考虑本书的容量，多数功能是在单个类中实现的，小的支持类仅作为数据结构。我们仔细学习每一个类和方法来了解该服务器怎样工作，由支持类开始，终止于主程序。

MimeHeader.java

MIME是通过电子邮件系统传达多媒体内容的一个Internet标准。该标准是由Nat Borenstein在1992年创建的。HTTP协议运用并扩展了MIME标头的概念，在HTTP客户和服务器之间传输常规的属性/值对。

构造函数　该类是Hashtable 的一个子类，所以它能方便的存储和检索与MIME标头有关的关键字/值对。它有两个构造函数。一个创建一个不含关键字的空的MimeHeader。另一个以一个格式化的字符串作为MIME标头，然后把它解析为对象的初始内容。参看下面的parse( )。

parse( )　parse( )方法用来获取一个原始MIME格式的字符串，并使它的关键字/值对进入一个给定的MimeHeader实例。它用StringTokenizer 把输入数据分解成独立的由CRLF(\r\n)序列标记的行。然后用规范的while... hasMoreTokens( ) ... nextToken( ) 序列遍历每一行。

对于MIME标头的每一行，parse( )通过冒号（：）把该行分解成两个字符串。两个变量key和val由substring( )方法设置，用来提取冒号前后的字符及后面的空隔。当两个字符串被提取后，使用put( )方法存储Hashtable中的关键字和值之间的关联。

toString( )　toString( )方法（用于String 串联操作，+）只是parse( )的反方法。它获取当前存储在MimeHeader 中的关键字/值对，返回一个MIME格式的字符串描述，然后打印关键字，跟着是冒号和空隔，然后是值，最后是CRLF。

put( ), get( ), AND fix( )　如果不是特殊的任务，Hashtable中的put( )和get( )方法将运行良好。MIME规范定义了几个重要的关键字例如Content-Type和Content-Length。一些早期的MIME 系统设备，特别是网络浏览器，对这些成员的大小写是自由的。一些用Content-type，另一些用content-type。为避免灾祸，我们的HTTP服务器努力将所有的输入和输出的

MimeHeader关键字转换成规范形式Content-Type。因此，我们在它们进入Hashtable和寻找给定关键字之前用fix()方法重载put( )和get( )，转变值的大小写。

代码 下面是MimeHeader的源代码：

```
import java.util.*;

class MimeHeader extends Hashtable {
  void parse(String data) {
    StringTokenizer st = new StringTokenizer(data, "\r\n");

    while (st.hasMoreTokens()) {
      String s = st.nextToken();
      int colon = s.indexOf(':');
      String key = s.substring(0, colon);
      String val = s.substring(colon + 2); // skip ": "
      put(key, val);
    }
  }

  MimeHeader() {}

  MimeHeader(String d) {
    parse(d);
  }

  public String toString() {
    String ret = "";
    Enumeration e = keys();

    while(e.hasMoreElements()) {
      String key = (String) e.nextElement();
      String val = (String) get(key);
      ret += key + ": " + val + "\r\n";
    }
    return ret;
  }

  // This simple function converts a mime string from
  // any variant of capitalization to a canonical form.
  // For example: CONTENT-TYPE or content-type to Content-Type,
  // or Content-length or CoNTeNT-LENgth to Content-Length.
  private String fix(String ms) {
    char chars[] = ms.toLowerCase().toCharArray();
    boolean upcaseNext = true;

    for (int i = 0; i < chars.length - 1; i++) {
      char ch = chars[i];
      if (upcaseNext && 'a' <= ch && ch <= 'z') {
        chars[i] = (char) (ch - ('a' - 'A'));
      }
      upcaseNext = ch == '-';
    }
    return new String(chars);
  }
```

```
  public String get(String key) {
    return (String) super.get(fix(key));
  }

  public void put(String key, String val) {
    super.put(fix(key), val);
  }
}
```

HttpResponse.java

HttpResponse类是所有与HTTP服务器应答有关的事物的包装程序。它被httpd类的代理部分使用。当你向一个HTTP服务器发送一个请求时，它以一个存储在statusCode中的整数形式的代码以及一个存储在reasonPhrase中的文本应答（这些变量名在正式的HTTP规范中规定）。这个单行的响应后面跟随着一个包含进一步应答信息的MIME头。我们用以前解释过的MimeHeader对象来解析这个字符串。MimeHeader对象存储在HttpResponse类的mh变量中。这些变量不是私有的，所以httpd可以直接使用它们。

构造函数　如果用一个字符串参数创建一个HttpResponse类对象，它被用来作为一个HTTP服务器的原始响应，并传向下面描述的parse( )来初始化对象。你还可以传入一个预计算的状态码，原因语句以及MIME标头。

parse( )　parse( )方法获得从HTTP服务器上读取的原始数据，从第一行解析出statusCode 和reasonPhrase ，然后在剩下的行外部创建一个MimeHeader 。

toString( )　toString( )方法是parse( )的逆方法。它获取HttpResponse对象的当前值并返回一个字符串，HTTP客户希望从服务器读回该字符串。

代码　下面是HttpResponse 的源代码：

```
import java.io.*;
/*
 * HttpResponse
 * Parse a return message and MIME header from a server.
 * HTTP/1.0 302 Found  =  redirection, check Location for where.
 * HTTP/1.0 200 OK = file data comes after mime header.
 */

class HttpResponse
{
  int statusCode;     // Status-Code in spec
  String reasonPhrase; // Reason-Phrase in spec
  MimeHeader mh;
  static String CRLF = "\r\n";

  void parse(String request) {
    int fsp = request.indexOf(' ');
    int nsp = request.indexOf(' ', fsp+1);
    int eol = request.indexOf('\n');
    String protocol = request.substring(0, fsp);
    statusCode = Integer.parseInt(request.substring(fsp+1, nsp));
    reasonPhrase = request.substring(nsp+1, eol);
    String raw_mime_header = request.substring(eol + 1);
```

```
    mh = new MimeHeader(raw_mime_header);
  }

  HttpResponse(String request) {
    parse(request);
  }

  HttpResponse(int code, String reason, MimeHeader m) {
    statusCode = code;
    reasonPhrase = reason;
    mh = m;
  }

  public String toString() {
    return "HTTP/1.0 " + statusCode + " " + reasonPhrase + CRLF +
      mh + CRLF;
  }
}
```

UrlCacheEntry.java

为在服务器上保存文档的内容，必须在用于找回文档的URL和文档自身描述之间建立联系。一个文档由它的MimeHeader和原始数据描述。例如一副图像可以被一个Content-Type: image/gif样式的MimeHeader描述，而原始图像数据就是一个字节数组。同样，一个网页在它的MimeHeader中有Content-Type:text/html关键字/值对，而原始数据就是HTML页的内容。再次申明，实例变量不是私有的，所以httpd可以自由的访问它们。

构造函数 UrlCacheEntry 对象的构造函数需要用URL作为关键字以及一个与之相关的MimeHeader。如果MimeHeader内部有一个名为Content-Length成员（大多数情况下如此），数据区域被预先分配足够大的空间来保存它的内容。

append( ) append( ) 方法用来给UrlCacheEntry对象增添数据的。它不是一个简单的setData( )方法，原因是数据可能流经网络且需要在一定时间被存储成块。append( )方法处理三种情形。第一种，数据缓冲区根本没有分配。第二种情形，数据缓冲区对于引入的数据来说太小，所以它被重新分配。最后一种情况，引入的数据正好可以插入缓冲区。在任何时候，length成员变量保存数据缓冲区当前的有效大小值。

代码 下面是UrlCacheEntry的源代码：

```
class UrlCacheEntry
{
  String url;
  MimeHeader mh;
  byte data[];
  int length = 0;

  public UrlCacheEntry(String u, MimeHeader m) {
    url = u;
    mh = m;
    String cl = mh.get("Content-Length");
    if (cl != null) {
      data = new byte[Integer.parseInt(cl)];
    }
```

```
  }

  void append(byte d[], int n) {
    if (data == null) {
      data = new byte[n];
      System.arraycopy(d, 0, data, 0, n);
      length = n;
    } else if (length + n > data.length) {
      byte old[] = data;
      data = new byte[old.length + n];
      System.arraycopy(old, 0, data, 0, old.length);
      System.arraycopy(d, 0, data, old.length, n);
    } else {
      System.arraycopy(d, 0, data, length, n);
      length += n;
    }
  }
}
```

LogMessage.java

LogMessage是一个简单的接口，它只定义了一个方法log( )，该方法只有一个String型参数。它用来抽象从httpd获得消息的输出。在应用程序条件下，该方法用来打印标准应用程序起始处控制台的输出。在小应用程序情况下，数据被送到一个视窗文本缓冲区。

代码　下面是LogMessage的源程序：

```
interface LogMessage {
  public void log(String msg);
}
```

httpd.java

这真是一个具有很多功能的大类。我们将一个方法一个方法的讲解它。

构造函数　存在5个主要的实例变量：port, docRoot，log，cache和stopFlag，它们都是私有的。其中的三个可以由httpd的独立构造函数设置，显示如下：

```
httpd(int p, String dr, LogMessage lm)
```

它初始化监听端口，初始化检索文件的目录以及初始化发送消息的接口。

第四个实例变量cache，是在RAM中保存所有文件的Hashtable，它是在对象创建时被初始化的。stopFlag 控制程序的执行。

静态部分　该类中有几个重要的静态变量。MIME标头中的“Server”域报告的版本在变量version中被发现。接着定义了一些常量：HTML文件的MIME类型，mime_text_html；MIME的结束顺序，CRLF；代替原始目录请求返回的HTML文件名，indexfile以及在输入/输出中用到的数据缓冲区的大小， buffer_size。

然后mt 定义了一系列文件扩展名和这些文件相应的MIME 类型。types Hashtable在下一个块中被静态初始化，以用来包含作为可选关键字和值的数组mt。接着可以用fnameToMimeType( )方法来返回传入的每个filename的合适的MIME类型。如果filename 不

含mt 表中的任何一个扩展名，该方法返回defaultExt或“text/plain.”。

统计计算器 下面，我们声明另外5个实例变量。它们是没有private 修饰符，所以一个外部监控器可以检查这些值并以图形形式显示它们（我们将在后面演示）。这些变量表示了我们的Web服务器所用的统计资料。点击数和提供的字节的原始数目被存储在hits_served 和bytes_served中。通常存储在高速缓存中的文件和字节数被存放在files_in_cache 和bytes_in_cache中。最后，我们把成功在高速缓存外部提供服务的点击数目存放在hits_to_cache中。

toBytes( ) 接着，我们有一个方便的程序，toBytes( )。该程序把它的字符串转变成一个字节数组。这是十分必要的，因为Java的String 对象是以统一编码的字符形式存储的，而Internet 协议中的混合语例如HTTP是老式的8位ASCII码。

makeMimeHeader( ) MakeMimeHeader()方法是另一个方便的方法，它用来创建由一些关键字值填充的MimeHeader对象。该方法返回的MimeHeader在Date成员中含有当前时间和时期，在Server成员中有服务器的名称和版本，Content-Type成员中有type参数，Content-Length成员中有length参数。

error( ) error( )方法用来格式化HTML页并返回提出不能完成请求的Web客户。第一个参数code，是返回的出错代码。一般它在400到499之间。我们的服务器返回404和405错误。它用HttpResponse 类和适当的MimeHeader来封装返回的代码。该方法返回字符串表示是与HTML页有关的响应。该页包括易于人读的错误代码信息msg 和导致错误的url请求。

getRawRequest( ) GetRawRequest( )方法是很简单的。它从流读取数据直到它获得两个连续的换行符。它忽略回车符号并且只寻找换行符。一旦它已经发现了连续的两个换行符，它使字节数组转向一个String对象并返回该对象。如果输入流在结束之前没有生成两个连续的换行符，它将返回null。这说明了HTTP服务器和客户的消息是怎样被格式化的。它们以状态的一行开始然后立即跟着一个MIME头。MIME头的结尾被两个换行符从剩余的内容中分离。

logEntry( ) logEntry( )方法用来报告标准格式下的HTTP服务器的每个点击数。该方法生成的格式也许看起来有一点奇怪，但是它和HTTP日志文件的当前标准相匹配。该方法有若干个用来格式化每个日志项的日期戳的辅助变量和方法。months数组用来把月份转换成字符串。当host变量接受一个给定主机的连接时它由主HTTP循环设置。fmt02d( )方法把0到9的整数格式化成两位的，第一位为零的数，然后结果字符串通过LogMessage接口变量log传输。

writeString( ) 另一个方便的方法writeString( )，用来隐藏字符到字节数组的转变，以使它可以被写入流。

writeUCE( ) writeUCE( )方法占取一个OutputStream和一个UrlCacheEntry。它从高速缓存项提取信息，以便给网络客户传送消息。消息中包含适当的响应代码，MIME标头和内容。

serveFromCache( ) 这个布尔方法试图在高速缓存中发现一个特殊的URL。如果成功，缓存项的内容被写给客户，hits_to_cache变量递增，调用者被返回true。否则，它只返回false。

loadFile( ) 该方法占用了一个InputStream、与之相应的url以及该URL的MimeHeader。

用存储在MimeHeader的信息创建一个新的UrlCacheEntry。输入流在buffer_size字节块中被读取并传入UrlCacheEntry。结果的UrlCacheEntry 存储在高速缓存中。files_in_cache和bytes_in_cache变量更新，UrlCacheEntry返回调用者。

readFile( )　readFile( )方法就loadFile( )方法来说看起来有些多余，实际不然。该方法严格的从本地文件系统读取文件。在本地文件系统中，loadFile( )方法用来与各种类型的流交流。如果File对象f存在，将会为它创建一个InputStream类。文件的大小是决定了的，MIME类型来自文件名。当loadFile( )被调用来做实际的读取和缓存高速工作时，有两个变量用来创建合适的MimeHeader。

writeDiskCache( )　writeDiskCache( )方法占用一个UrlCacheEntry 对象并且把它持久的写入本地磁盘。它从URL中建立一个目录名，确保用与系统有关的separatorChar代替斜线（/）字符。然后它调用mkdirs( )方法来保证这个URL的本地磁盘路径存在。最后，它打开一个FileOutputStream，向它写入所有的数据然后关闭它。

handleProxy( )　HandleProxy（）程序是该服务器的两个主要模式之一。基础思想是：如果你把你的浏览器设置成把该服务器当成代理服务器，则要传给它的请求将包括完整的URL，URL中常规GET方法可删除“http://”和主机名部分。我们仅把完整的URL拆碎，寻找“：//”序列，其次是斜线（/），然后是使用非标准端口号的服务器的另一个冒号（：）。一旦我们发现了这些字符，我们就可以知道已经所需要的主机和端口号以及我们需要获取的URL。然后我们可以试图从RAM高速缓存中转载一个先前保存过的文档版本。如果失败，我们可以试图从文件系统装载它到RAM高速缓存并且再尝试从RAM高速缓存装载它。如果此举失败，那么事情变得很有趣，因为我们必须从远程站点读取该文件。

为此，我们打开一个远程站点和端口的套接字。我们发送一个GET请求，要求传给我们的URL。无论我们从远程站点获得什么响应标头信息，我们把它传给客户。如果代码是200，对于成功的文件传输，我们还把确认数据流读到一个新的UrlCacheEntry类并且把它写入客户套接字。然后，我们调用writeDiskCache( )来保存传输结果到本地磁盘。我们记录传输日志，关闭套接字，然后返回。

handleGet( )　当http后台进程像一个普通的Web服务器一样工作时，handleGet( )被调用。它有一个服务于文件的本地磁盘文件根目录。handleGet( )的参数告诉它向何处写结果，何处访问URL以及何处请求网络浏览器的MimeHeader。这个MIME标头将包括用户代理字符串和其他有用的属性。开始，我们试图在RAM高速缓存外为URL提供服务。如果此举失败，为寻找URL，我们顺序访问文件系统。如果文件不存在或不可读，我们向Web客户报告一个错误。否则，我们就用readFile( )方法获得文件的内容并把它们输入到高速缓存。然后调用writeUCE( )方法以用来传输文件内容到客户套接字。

doRequest( )　每次连接服务器时都会调用一次 doRequest( )方法。它解析请求字符串和引入的MIME标头。它在请求字符串中是否存在“：//”的基础上判定是调用handleProxy( )还是 handleGet( )方法。如果不用GET，而使用任何其他的方法例如HEAD或POST，该程序向客户返回一个405错误。注意如果stopFlag 是true时HTTP请求被忽略。

run( )　run( )方法在服务器线程启动时被调用。它在指定端口生成一个新的ServerSocket，在服务器套接字处进入一个调用accept( )的无限循环，把结果Socket传给doRequest( )作检查。

start( ) AND stop( ) 存在两个用来启动和终止服务器过程的方法。这些方法设置stopFlag的值。

main( ) 你可以在命令行用main( )方法来运行应用程序。它为服务器自身设置LogMessage参数，然后为log( )提供简单的控制台输出执行。

代码 下面是httpd的源代码：

```
import java.net.*;
import java.io.*;
import java.text.*;
import java.util.*;

class httpd implements Runnable, LogMessage {
  private int port;
  private String docRoot;
  private LogMessage log;
  private Hashtable cache = new Hashtable();
  private boolean stopFlag;

  private static String version = "1.0";
  private static String mime_text_html = "text/html";
  private static String CRLF = "\r\n";
  private static String indexfile = "index.html";
  private static int buffer_size = 8192;
  static String mt[] = {  // mapping from file ext to Mime-Type
    "txt", "text/plain",
    "html", mime_text_html,
    "htm", "text/html",
    "gif", "image/gif",
    "jpg", "image/jpg",
    "jpeg", "image/jpg",
    "class", "application/octet-stream"
  };
  static String defaultExt = "txt";
  static Hashtable types = new Hashtable();
  static {
    for (int i=0; i<mt.length;i+=2)
      types.put(mt[i], mt[i+1]);
  }

  static String fnameToMimeType(String filename) {
    if (filename.endsWith("/"))     // special for index files.
      return mime_text_html;
    int dot = filename.lastIndexOf('.');
    String ext = (dot > 0) ? filename.substring(dot + 1) : defaultExt;
    String ret = (String) types.get(ext);
    return ret != null ? ret : (String)types.get(defaultExt);
  }

  int hits_served = 0;
  int bytes_served = 0;
  int files_in_cache = 0;
  int bytes_in_cache = 0;
  int hits_to_cache = 0;
```

```
private final byte toBytes(String s)[] {
  byte b[] = s.getBytes();
  return b;
}

private MimeHeader makeMimeHeader(String type, int length) {
  MimeHeader mh = new MimeHeader();
  Date curDate = new Date();
  TimeZone gmtTz = TimeZone.getTimeZone("GMT");
  SimpleDateFormat sdf =
    new SimpleDateFormat("dd MMM yyyy hh:mm:ss zzz");
  sdf.setTimeZone(gmtTz);
  mh.put("Date", sdf.format(curDate));
  mh.put("Server", "JavaCompleteReference/" + version);
  mh.put("Content-Type", type);
  if (length >= 0)
    mh.put("Content-Length", String.valueOf(length));
  return mh;
}

private String error(int code, String msg, String url) {
  String html_page = "<body>" + CRLF +
             "<h1>" + code + " " + msg + "</h1>" + CRLF;
  if (url != null)
    html_page += "Error when fetching URL: " + url + CRLF;
  html_page += "</body>" + CRLF;
  MimeHeader mh = makeMimeHeader(mime_text_html, html_page.length());
  HttpResponse hr = new HttpResponse(code, msg, mh);

  logEntry("GET", url, code, 0);
  return hr + html_page;
}

// Read 'in' until you get two \n's in a row.
// Return up to that point as a String.
// Discard all \r's.
private String getRawRequest(InputStream in)
  throws IOException {
  byte buf[] = new byte[buffer_size];
  int pos=0;
  int c;
  while ((c = in.read()) != -1) {
    switch (c) {
      case '\r':
      break;
      case '\n':
        if (buf[pos-1] == c) {
          return new String(buf,0,pos);
        }
      default:
        buf[pos++] = (byte) c;
    }
  }
  return null;
```

```
}

static String months[] = {
  "Jan", "Feb", "Mar", "Apr", "May", "Jun",
  "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
};
private String host;
// fmt02d is the same as C's printf("%02d", i)
private final String fmt02d(int i) {
  if(i < 0) {
    i = -i;
    return ((i < 9) ? "-0" : "-") + i;
  }
  else {
    return ((i < 9) ? "0" : "") + i;
  }
}
private void logEntry(String cmd, String url, int code, int size) {
  Calendar calendar = Calendar.getInstance();
  int tzmin = calendar.get(Calendar.ZONE_OFFSET)/(60*1000);
  int tzhour = tzmin / 60;
  tzmin -= tzhour * 60;
  log.log(host + " - - [" +
    fmt02d(calendar.get(Calendar.DATE) ) + "/" +
    months[calendar.get(Calendar.MONTH)] + "/" +
    calendar.get(Calendar.YEAR) + ":" +
    fmt02d(calendar.get(Calendar.HOUR) ) + ":" +
    fmt02d(calendar.get(Calendar.MINUTE) ) + ":" +
    fmt02d(calendar.get(Calendar.SECOND)) + " " +
    fmt02d(tzhour) + fmt02d(tzmin) +
    "] \"" +
    cmd + " " +
    url + " HTTP/1.0\" " +
    code + " " +
    size + "\n");
  hits_served++;
  bytes_served += size;
}

private void writeString(OutputStream out, String s)
  throws IOException {
  out.write(toBytes(s));
}

private void writeUCE(OutputStream out, UrlCacheEntry uce)
  throws IOException {
    HttpResponse hr = new HttpResponse(200, "OK", uce.mh);
    writeString(out, hr.toString());
    out.write(uce.data, 0, uce.length);
    logEntry("GET", uce.url, 200, uce.length);
}

private boolean serveFromCache(OutputStream out, String url)
  throws IOException {
  UrlCacheEntry uce;
```

```
  if ((uce = (UrlCacheEntry)cache.get(url)) != null) {
    writeUCE(out, uce);
    hits_to_cache++;
    return true;
  }
  return false;
}

private UrlCacheEntry loadFile(InputStream in, String url,
                               MimeHeader mh)
  throws IOException {

  UrlCacheEntry uce;
  byte file_buf[] = new byte[buffer_size];
  uce = new UrlCacheEntry(url, mh);
  int size = 0;
  int n;
  while ((n = in.read(file_buf)) >= 0) {
    uce.append(file_buf, n);
    size += n;
  }
  in.close();
  cache.put(url, uce);
  files_in_cache++;
  bytes_in_cache += uce.length;
  return uce;
}

private UrlCacheEntry readFile(File f, String url)
  throws IOException {

  if (!f.exists())
    return null;
  InputStream in = new FileInputStream(f);
  int file_length = in.available();
  String mime_type = fnameToMimeType(url);
  MimeHeader mh = makeMimeHeader(mime_type, file_length);
  UrlCacheEntry uce = loadFile(in, url, mh);
  return uce;
}

private void writeDiskCache(UrlCacheEntry uce)
  throws IOException {

  String path = docRoot + uce.url;
  String dir = path.substring(0, path.lastIndexOf("/"));
  dir.replace('/', File.separatorChar);
  new File(dir).mkdirs();
  FileOutputStream out = new FileOutputStream(path);
  out.write(uce.data, 0, uce.length);
  out.close();
}

// A client asks us for a url that looks like this:
// http://the.internet.site/the/url
```

```
// we go get it from the site and return it...
private void handleProxy(OutputStream out, String url,
                         MimeHeader inmh) {
  try {
    int start = url.indexOf("://") + 3;
    int path = url.indexOf('/', start);
    String site = url.substring(start, path).toLowerCase();
    int port = 80;
    String server_url = url.substring(path);
    int colon = site.indexOf(':');
    if (colon > 0) {
      port = Integer.parseInt(site.substring(colon + 1));
      site = site.substring(0, colon);
    }
    url = "/cache/" + site + ((port != 80) ? (":" + port) : "") +
         server_url;
    if (url.endsWith("/"))
      url += indexfile;

    if (!serveFromCache(out, url)) {
      if (readFile(new File(docRoot + url), url) != null) {
        serveFromCache(out, url);
        return;
      }

      // If we haven't already cached this page, open a socket
      // to the site's port and send a GET command to it.
      // We modify the user-agent to add ourselves... "via".

      Socket server = new Socket(site, port);
      InputStream server_in = server.getInputStream();
      OutputStream server_out = server.getOutputStream();
      inmh.put("User-Agent", inmh.get("User-Agent") +
              " via JavaCompleteReferenceProxy/" + version);
      String req = "GET " + server_url + " HTTP/1.0" + CRLF +
                 inmh + CRLF;
      writeString(server_out, req);
      String raw_request = getRawRequest(server_in);
      HttpResponse server_response =
                   new HttpResponse(raw_request);
      writeString(out, server_response.toString());
      if (server_response.statusCode == 200) {
        UrlCacheEntry uce = loadFile(server_in, url,
                                     server_response.mh);
        out.write(uce.data, 0, uce.length);
        writeDiskCache(uce);
        logEntry("GET", site + server_url, 200, uce.length);
      }
      server_out.close();
      server.close();
    }
  } catch (IOException e) {
    log.log("Exception: " + e);
  }
}
```

```
private void handleGet(OutputStream out, String url,
                       MimeHeader inmh) {
  byte file_buf[] = new byte[buffer_size];
  String filename = docRoot + url +
                 (url.endsWith("/") ? indexfile : "");
  try {
    if (!serveFromCache(out, url)) {
      File f = new File(filename);
      if (! f.exists()) {
        writeString(out, error(404, "Not Found", filename));
        return;
      }
      if (! f.canRead()) {
        writeString(out, error(404, "Permission Denied", filename));
        return;
      }
      UrlCacheEntry uce = readFile(f, url);
      writeUCE(out, uce);
    }
  } catch (IOException e) {
    log.log("Exception: " + e);
  }
}

private void doRequest(Socket s) throws IOException {
  if(stopFlag)
    return;
  InputStream in = s.getInputStream();
  OutputStream out = s.getOutputStream();
  String request = getRawRequest(in);
  int fsp = request.indexOf(' ');
  int nsp = request.indexOf(' ', fsp+1);
  int eol = request.indexOf('\n');
  String method = request.substring(0, fsp);
  String url = request.substring(fsp+1, nsp);
  String raw_mime_header = request.substring(eol + 1);

  MimeHeader inmh = new MimeHeader(raw_mime_header);

  request = request.substring(0, eol);

  if (method.equalsIgnoreCase("get")) {
    if (url.indexOf("://") >= 0) {
      handleProxy(out, url, inmh);
    } else {
      handleGet(out, url, inmh);
    }
  } else {
    writeString(out, error(405, "Method Not Allowed", method));
  }
  in.close();
  out.close();
}
```

```
  public void run() {
    try {
      ServerSocket acceptSocket;
      acceptSocket = new ServerSocket(port);
      while (true) {
        Socket s = acceptSocket.accept();
        host = s.getInetAddress().getHostName();
        doRequest(s);
        s.close();
      }
    } catch (IOException e) {
      log.log("accept loop IOException: " + e + "\n");
    } catch (Exception e) {
      log.log("Exception: " + e);
    }
  }

  private Thread t;
  public synchronized void start() {
    stopFlag = false;
    if (t == null) {
      t = new Thread(this);
      t.start();
    }
  }

  public synchronized void stop() {
    stopFlag = true;
    log.log("Stopped at " + new Date() + "\n");
  }

  public httpd(int p, String dr, LogMessage lm) {
    port = p;
    docRoot = dr;
    log = lm;
  }

  // This main and log method allow httpd to be run from the console.
  public static void main(String args[]) {
    httpd h = new httpd(80, "c:\\www", null);
    h.log = h;
    h.start();
    try {
      Thread.currentThread().join();
    } catch (InterruptedException e) {};
  }

  public void log(String m) {
    System.out.print(m);
  }
}
```

HTTP.java

这里是一个给HTTP服务器一个“前面板”功能的applet类。该applet有两个可以用来配

置服务器的参数：port和docroot。这是一个很简单的小应用程序。它创建了一个httpd实例，把它作为LogMessage 接口传入自己。然后创建了两个面板。一个面板在顶部有一个简单的标签，在中部是一个TextArea 文本区以显示LogMessage；另一个面板在底部有两个按钮及一个标签。小应用程序的start( )和stop( )方法调用httpd相应的方法。标注有“Start”和“Stop”的按钮调用httpd中它们相应的方法。在任何时候只要一条消息被记录，右下角的Label对象就更新，这样它包含httpd最新的统计数据。

```
import java.util.*;
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class HTTP extends Applet implements LogMessage,
                                    ActionListener
{
  private int m_port = 80;
  private String m_docroot = "c:\\www";
  private httpd m_httpd;
  private TextArea m_log;
  private Label status;

  private final String PARAM_port = "port";
  private final String PARAM_docroot = "docroot";
  public HTTP()  {
  }

  public void init()  {
    setBackground(Color.white);
    String param;

    // port: Port number to listen on
    param = getParameter(PARAM_port);
    if (param != null)
      m_port = Integer.parseInt(param);

    // docroot: web document root
    param = getParameter(PARAM_docroot);
    if (param != null)
      m_docroot = param;

    setLayout(new BorderLayout());

    Label lab = new Label("Java HTTPD");

    lab.setFont(new Font("SansSerif", Font.BOLD, 18));
    add("North", lab);
    m_log = new TextArea("", 24, 80);
    add("Center", m_log);
    Panel p = new Panel();
    p.setLayout(new FlowLayout(FlowLayout.CENTER,1,1));
    add("South", p);
    Button bstart = new Button("Start");
    bstart.addActionListener(this);
```

```
    p.add(bstart);
    Button bstop = new Button("Stop");
    bstop.addActionListener(this);
    p.add(bstop);
    status = new Label("raw");
    status.setForeground(Color.green);
    status.setFont(new Font("SansSerif", Font.BOLD, 10));
    p.add(status);
    m_httpd = new httpd(m_port, m_docroot, this);
  }

  public void destroy()  {
    stop();
  }

  public void paint(Graphics g)  {
  }

  public void start()  {
    m_httpd.start();
    status.setText("Running  ");
    clear_log("Log started on " + new Date() + "\n");
  }

  public void stop()  {
    m_httpd.stop();
    status.setText("Stopped  ");
  }

  public void actionPerformed(ActionEvent ae) {
    String label = ae.getActionCommand();
    if(label.equals("Start")) {
      start();
    }
    else {
      stop();
    }
  }

  public void clear_log(String msg) {
    m_log.setText(msg + "\n");
  }

  public void log(String msg) {
    m_log.append(msg);
    status.setText(m_httpd.hits_served + " hits (" +
           (m_httpd.bytes_served / 1024) + "K), " +
           m_httpd.files_in_cache + " cached files (" +
           (m_httpd.bytes_in_cache / 1024) + "K), " +
           m_httpd.hits_to_cache + " cached hits");
    status.setSize(status.getPreferredSize());
  }
}
```

注意：httpd.java 和HTTP.java文件中，代码是在假定文件根部是“c:\www”的

基础上建立的。你可能需要改变配置的值。因为这个小应用程序写入一个日志文件，它只能在被信任的条件下工作。例如，一个小应用程序在它可以获得用户CLASSPATH时被信任。

18.9　数 据 报

对于你的大多数网络需求，你会对TCP/IP型网络很满意。它提供了有序的、可预测和可靠的信息包数据流。但是，这样做的代价也很大。TCP包含很多在拥挤的网络中处理拥塞控制的复杂算法以及信息丢失的悲观的预测。这导致了一个效率很差的传输数据方式。数据报是一种可选的替换方法。

数据报（Datagrams）是在机器间传递的信息包，它有些像从一个训练有素但是很盲目的捕手投出一记有力的传球给三垒。一旦数据报被释放给它们预定的目标，不保证它们一定到达目的地，甚至不保证一定存在数据的接收者。同样，数据报被接受时，不保证它在传输过程不受损坏，不保证发送它的机器仍在那儿等待响应。

Java通过两个类实现UDP协议顶层的数据报：DatagramPacket对象是数据容器，DatagramSocket是用来发送和接受DatagramPackets的机制。

18.9.1　DatagramPacket

DatagramPackets 可以用四个构造函数中的一个创建。第一个构造函数指定了一个接收数据的缓冲区和信息包的容量大小。它通过DatagramSocket接收数据。第二种形式允许你在存储数据的缓冲区中指定一个偏移量。第三种形式指定了一个用于DatagramSocket决定信息包将被送往何处的目标地址和端口。第四种形式从数据中指定的偏移量位置开始传输数据包。想象前两种形式是建立在“盒内”的，后两种形式形成了填塞物，并为一个信封注明了地址。下面是四个构造函数：

```
DatagramPacket(byte data[ ], int size)
DatagramPacket(byte data[ ], int offset, int size)
DatagramPacket(byte data[ ], int size, InetAddress ipAddress, int port)
DatagramPacket(byte data[ ], int offset, int size, InetAddress ipAddress,
int port)
```

存在几种方法可获取DatagramPacket内部状态。它们对信息包的目标地址和端口号以及原始数据和数据长度有完全的使用权，下面是它们的概述：

InetAddress getAddress( )	返回目标文件InetAddress，一般用于发送。
Int getPort( )	返回端口号。
byte[ ] getData( )	返回包含在数据包中的字节数组数据。多用于在接收数据之后从数据包来检索数据。
Int getLength( )	返回包含在将从getData( )方法返回的字节数组中有效数据长度。通常它与整个字节数组长度不等。

18.9.2 数据报服务器和客户

下面的例子实现了一个非常简单的联网通信的客户和服务器。消息被写入服务器的窗口并通过网络写入客户端，在此它们被显示。

```
// Demonstrate Datagrams.
import java.net.*;

class WriteServer {
  public static int serverPort = 666;
  public static int clientPort = 999;
  public static int buffer_size = 1024;
  public static DatagramSocket ds;
  public static byte buffer[] = new byte[buffer_size];

  public static void TheServer() throws Exception {
    int pos=0;
    while (true) {
      int c = System.in.read();
      switch (c) {
        case -1:
          System.out.println("Server Quits.");
          return;
        case '\r':
          break;
        case '\n':
          ds.send(new DatagramPacket(buffer,pos,
             InetAddress.getLocalHost(),clientPort));
          pos=0;
          break;
        default:
          buffer[pos++] = (byte) c;
      }
    }
  }

  public static void TheClient() throws Exception {
    while(true) {
      DatagramPacket p = new DatagramPacket(buffer, buffer.length);
      ds.receive(p);
      System.out.println(new String(p.getData(), 0, p.getLength()));
    }
  }

  public static void main(String args[]) throws Exception {
    if(args.length == 1) {
      ds = new DatagramSocket(serverPort);
      TheServer();
    } else {
      ds = new DatagramSocket(clientPort);
      TheClient();
    }
  }
}
```

该程序被DatagramSocket 构造函数限制在本地机的两个端口间运行。试着运行该程序，在一个窗口中键入

```
java WriteServer
```

这是在客户端的。然后运行

```
java WriteServer 1
```

这是在服务器端的。在服务器窗口中打印的任何东西在接受回车之后将被送到客户窗口。

注意：该例题需要你的计算机与Internet相连。

18.10　网 络 价 值

给你5个类InetAddress，Socket，ServerSocket，DatagramSocket和DatagramPacket，你可以编写现有的任何Internet协议。它们同样为Internet连接提供功能强大的低级控制。Java的网络包也完美地反映了Internet的状态。

第 19 章　Applet 类

在这一章里，将介绍Applet类的概念，它为小应用程序提供了必不可少的支持。在第12章里，曾介绍了小应用程序的一般形式和对其进行编译和运行所需要的步骤。现在，让我们更仔细的看一看小应用程序。

Applet类被包含在名叫java.applet的类库里，它为你提供了几种方法，用它们可以在你的小应用程序的执行过程中进行更严密的控制。除此以外，java.applet还定义了一些接口如：AppletContext, AudioClip, 和AppletStub等。

现在让我们先来回顾一下组成一个小应用程序的基本元素以及生成一个小应用程序并对其进行测试的必需的步骤。

19.1　Applet基础

所有的小应用程序都是Applet类的子类。因此，所有的小应用程序都必须引用java.applet类库。回想一下，AWT代表Abstract Window Toolkit（抽象窗口工具包）。既然所有的小应用程序都运行在一个窗口中，那么引入对这个窗口的支持类库就是必不可少的。需要注意的是，小应用程序并不是被基于控制台的Java运行环境的解释器所执行的，而是由Web浏览器或小应用程序阅读器所执行。本章你所看到的图像就是由标准的小应用程序阅读器appletviewer生成的，appletviewer由JDK(java开发工具)提供。但你能够按自己的喜好选择任何小应用程序阅读器或浏览器。

与大多数程序不同的是，一个小应用程序的执行不是从main()开始的。实际上，没有多少小应用程序使用main()。正相反，小应用程序的执行用一种完全不同的机制启动和控制，我们接下来将对这种机制进行介绍。对小应用程序窗口的输出并不是由函数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，小应用程序窗口的输入与一般的应用程序不同。

只要小应用程序经过编译，它就被包含在一个HTML文件中，并使用APPLET标记。这之后当支持Java的Web浏览器遇到HTML文件中的APPLET标记时，小应用程序就能被执行。为了更方便的观察和测试小应用程序，只需在你编制的Java源程序代码的头部加入一个包含APPLET标记的注释即可。用这种方法，你的代码就能用你的小应用程序所需的HTML语句记述下来，这样，只要你启动小应用程序阅读器并指定你的Java源代码文件为目标文件，就可以测试经过编译的小应用程序了。这里有一个例子告诉你如何加入注释：

```
/*
<applet code="MyApplet" width=200 height=60>
</applet>
*/
```

这条注释包含了一个APPLET标记，意思是指在一个200个像素宽，60个像素高的窗口中运行一个叫做MyApplet的小应用程序。由于加入一个APPLET命令能使测试小应用程序更简单，因此在这本书里所有的小应用程序都有适当的APPLET标记嵌套在一条注释语句中。

19.1.1　Applet类

Applet类定义了如表19-1所示的一些方法。Applet类为小应用程序的执行，如启动，中止等提供了所有必需的支持。它还提供了装载和显示图像的方法，以及装载和播放语音片断的方法。Applet扩展了AWT类中的Panel。依此类推，Panel扩展了类Container，Container扩展了Component。些类都为Jva的基于窗口的图形接口提供了支持。这样，Applet为基于窗口的所有活动提供了支持（有关AWT将在下面几章里详细介绍）。

表 19-1　由 Applet 定义的方法

方法	描述
void destroy( )	在一个小应用程序结束之前被浏览器调用。你的小应用程序在被删除之前如果需要完成任何清除工作则会重载此方法
AccessibleContext getAccessibleContext( )	为调用对象返回可访问的上下文
AppletContext getAppletContext( )	返回与此小应用程序相关的上下文关系
String getAppletInfo( )	返回一个描述此小应用程序的字符串
AudioClip getAudioClip(URL url)	返回一个AudioClip对象，它封装了在由url所指定的地方找到的音频片断
AudioClip getAudioClip(URL url, String clipName)	返回一个AudioClip对象，它封装了在由url所指定的地方找到的名为clipName的音频片断
URL getCodeBase( )	返回与调用小应用程序相关的URL
URL getDocumentBase( )	返回调用此小应用程序的HTML文档的URL
Image getImage(URL url)	返回一个Image对象，它封装了在由url所指定的位置找到的图像
Image getImage(URL url, String imageName)	返回一个Image对象，它封装了在由url所指定的地方找到的名为imageName的图像
Locale getLocale( )	返回一个Locale对象，它被许多的对位置敏感的类和方法使用
String getParameter(String paramName)	返回与paramName相关的参数。如果所指定的参数未能找到的话，则返回null
String[ ] [ ] getParameterInfo( )	返回一个描述由此小应用程序所识别的参数的String表。表中的每一条必须包含三个字符串，分别包含了参数名，类型或范围描述以及用途说明
void init( )	在小应用程序开始执行时被调用。它是任何小应用程序调用的第一个方法

续表

方法	描述
boolean isActive( )	如果小应用程序已经启动则返回 true。如果小应用程序被中止则返回 false
static final AudioClip newAudioClip(URL url)	返回一个 AudioClip 对象，它封装了在url所指定的位置找到的音频片断。此方法类似于getAudioClip( )除了它是静态的且可无需Applet对象就可被执行(在 Java 2中被加入)
void play(URL url)	如果在url指定的位置能找到一个音频片断的话，则此片断被播放
void play(URL url, String clipName)	如果在url设定的位置能找到一个音频片断且名为clipName的话，则此片断被播放
void resize(Dimension dim)	根据dim指定的尺寸调整小应用程序的大小。dimension是一个存储在java.awt中的类。它包含了两个整数域：width 和 height
void resize(int width, int height)	根据width和 height设定的尺寸调整小应用程序的大小
final void setStub(AppletStub stubObj)	使 stubObj 成为小应用程序的存根。此方法由实时运行系统使用并且通常不被小应用程序调用。一个存根是提供小应用程序和浏览器之间的连接的小段代码
void showStatus(String str)	在浏览器或小应用程序阅读器的状态窗口显示str。如果浏览器不支持状态窗口，则无任何动作发生
void start( )	当小应用程序开始（或重新）执行时，被浏览器调用。它在小应用程序刚开始时在init之后被自动调用
void stop( )	被浏览器调用以将小应用程序挂起。一旦被停止，则当浏览器调用start（）时，小应用程序被启动

19.2 Applet体系结构

所谓的小应用程序是一种基于窗口的程序。所以，它的体系结构与本书第1部分所介绍的一般的基于控制台的程序是不同的。如果你对Windows编程很熟悉，在深入小应用程序编程时就能够得心应手。如果不是这样的话，你就必须先要理解几个重要的概念。

首先，小应用程序是由事件驱动的。尽管我们在这里并不关心事件处理的机制，但是对于事件驱动机制如何影响到小应用程序的设计这一问题，获得一般性的理解还是很重要的。一个小应用程序类似于系列提供中断服务的子程序的集合。程序就是这样运行的。在事件发生之前，小应用程序一直处于等待状态中。一旦事件发生，小应用程序就会采取相应措施并迅速将控制权交给AWT。这一点是很关键的。在大部分时间里，小应用程序都不会进入操作运行模式而长久的保持控制权。相反的，它必须针对特定的事件作出相应的动作并把控制交给AWT的运行环境。在有些情况下，你的小应用程序需要独立完成一些重复的作业（比如，在窗口中显示滚动信息），这时，你必须再启动一个额外的线程（你将在

本章后面部分看到一个例子）。

其次，用户可以与小应用程序进行交互，而不是通过其他方式。我们都知道，在一个非窗口界面的程序中，当程序需要输入时，它会提示用户并调用一定的输入方法，例如readLine()。而在小应用程序中，并不是这样运作的。相反，用户可以按照自己的喜好随意的与小应用程序进行交互。这些交互被送至小应用程序，作为小应用程序必须作出响应的事件。例如，当用户在小应用程序的窗口中点击鼠标时，一个鼠标点击事件就产生了。如果用户在小应用程序窗口中的焦点处按下一个键，一个按键事件就被产生。在以后的几章里，你就会看到，小应用程序包含了各种各样的控件，如下压式按钮，复选框等。当与某一控件进行交互时，一个事件就会产生。

虽然小应用程序的体系结构并不像基于控制台的程序那样易于理解，但Java的AWT使得它变得非常简单。如果你曾编写过Windows程序，你一定知道那种编程环境是怎样的恶劣。但幸运的是，Java的AWT提供了一种简洁得多的方法，从而更易于掌握。

19.3　Applet主框架

几乎大多数的小应用程序都重载一套方法，这些方法提供了浏览器或小应用程序阅读器与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( )是由Applet所定义的。另一个方法，paint( )是由AWT组件类定义的。所有这些方法的具体实现也都被提供。小应用程序并不需要重载那些它们没有用到的方法。但是，只有非常简单的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，如下图所示。

```
// An Applet skeleton.
import java.awt.*;
import java.applet.*;
/*
<applet code="AppletSkel" width=300 height=100>
</applet>
*/

public class AppletSkel extends Applet {
  // Called first.
  public void init() {
    // initialization
  }

  /* Called second, after init().  Also called whenever
     the applet is restarted. */
  public void start() {
    // start or resume execution
  }

  // Called when the applet is stopped.
  public void stop() {
    // suspends execution
  }
```

```
  /* Called when applet is terminated.  This is the last
     method executed. */
  public void destroy() {
    // perform shutdown activities
  }

  // Called when an applet's window must be restored.
  public void paint(Graphics g) {
    // redisplay contents of window
  }
}
```

虽然这个主框架不能完成任何功能，但它可以被编译和执行。但被执行时，就能产生下面的窗口，可以用小应用程序阅读器看到。

19.3.1 Applet的初始化与终止

懂得程序主框架中所示的各种方法的排列顺序是很重要的。当一个小应用程序开始执行时，AWT就以如下顺序调用以下方法：

1. init( )
2. start( )
3. paint( )

当一个小应用程序终止时，下列方法就以下列顺序被调用：

1. stop()
2. destroy()

现在让我们更仔细的来看一看这些方法。

init()

init（）方法是被调用的第一个方法。这是初始化变量的地方。这个方法在你的小应用程序运行期间仅被调用一次。

start()

start()方法是在init()之后被调用。它也在小应用程序被终止后重新启动时调用。Init()

仅在小应用程序第一次被装载时调用一次，而start()却在每一次小应用程序的HTML文档被显示在屏幕上时都被调用。因此，如果用户离开一个网页之后重新进入时，小应用程序就会从start()开始重新执行。

paint( )

在每一次你的小应用程序的输出必须重画窗口，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复原。paint()方法也在小应用程序开始执行时被调用。不管是什么原因，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，这个上下文将被用到。

stop( )

当Web浏览器离开包含小应用程序的HTML文件时，stop()方法就被调用，如在浏览器去另一个页面时。当stop()被调用时，小应用程序很可能在运行。你应该使用stop()来挂起一些在小应用程序不可见时不需要运行的线程。当用户回到此页面时，你能重新启动它们。

destroy( )

当环境决定了你的小应用程序需要完全从内存中移去时，destroy()方法被调用。在这时候，你应该释放任何小应用程序可能用到的资源。stop()方法总是在destroy（）之前被调用。

19.3.2　重载update()方法

在某些情况下，你的小应用程序可能需要覆盖另外一个AWT所定义的方法，叫做update()。这个方法在你的小应用程序要求窗口的一部分要被重画时被调用。默认的update()的方法是先用默认的背景颜色填充小应用程序窗口，再调用paint()方法。如果你在填充背景时用的颜色与paint()方法中使用的不同，那么在每次update()被调用时，也就是只要窗口被重画时，用户将会感觉到默认背景的闪动。避免这个问题的一种方法是重载update()方法，从而使它完成所有必要的显示功能。然后，使paint()简单的调用update()。这样，在一些应用中，小应用程序将重载paint()和update()，如下所示：

```
public void update(Graphics g) {
  // redisplay your window, here.
}

public void paint(Graphics g) {
  update(g);
}
```

对于本书中所举的例子，我们只在需要时重载update()方法。

19.4 简单的小应用程序显示方法

我们曾提到，小应用程序用一个窗口来显示，并使用AWT来完成输入和输出。有一系列的方法，过程和技术对于控制AWT视窗环境是十分必要的，尽管我们将在接下来的几章中对它们进行深入研究，这里先介绍其中的几个，我们用它们来编写一些示例小应用程序。

我们在第12章里介绍过，为了向小应用程序输出字符串，可以使用drawString（）方法，这个方法是Graphics类中的一个。一般来说，它在update()或paint()中被调用。它的一般形式如下所示：

```
void drawString(String message, int x, int y)
```

在这里，message是要输出的字符串，它从坐标为x，y的地方开始输出。在一个Java窗口中，左上角的座标为0，0。drawString()方法不识别换行符，所以如果你想要在下一行开始新的输出时，你必须手工来完成这件事，精确的设定你想要新行开始的位置的X,Y座标。（你将在以后的章节中看到，有一些技术可以简化这个过程）。

为设定小应用程序窗口的背景颜色，可以用setBackground()方法。设定窗口的前景颜色（例如，所输出文字的底色），可用setForeground()方法。这些方法在组件类中被定义，它们的使用方法如下：

```
void setBackground(Color newColor)
void setForeground(Color newColor)
```

这里，newColor规定了新的颜色。Color类规定了以下所示的常数，它们被用来设定颜色：

Color.black	Color.magenta
Color.blue	Color.orange
Color.cyan	Color.pink
Color.darkGray	Color.red
Color.gray	Color.white
Color.green	Color.yellow
Color.lightGray	

例如，下面的语句设定了背景为绿色，字体为红色。

```
setBackground(Color.green);
setForeground(Color.red);
```

设定背景和前景颜色的一个好办法是使用init()方法。当然，你可以在你的小应用程序执行时随心所欲的改变颜色。前景的默认颜色是黑色，而背景的默认颜色是浅灰色。

你能通过分别调用getBackground()和getForeground()方法来获得背景和前景颜色的当前设定。它们也是在Component类中被定义的，如下所示：

```
Color getBackground( )
```

```
Color getForeground( )
```

下面是一个很简单的小应用程序，它设定背景颜色为蓝绿色，前景颜色为红色，并显示了一条消息来说明当小应用程序启动时init()，start()，paint()方法被调用的顺序。

```
/* A simple applet that sets the foreground and
   background colors and outputs a string. */
import java.awt.*;
import java.applet.*;
/*
<applet code="Sample" width=300 height=50>
</applet>
*/

public class Sample extends Applet{
  String msg;

  // set the foreground and background colors.
  public void init() {
    setBackground(Color.cyan);
    setForeground(Color.red);
    msg = "Inside init( ) --";
  }

  // Initialize the string to be displayed.
  public void start() {
    msg += " Inside start( ) --";
  }

  // Display msg in applet window.
  public void paint(Graphics g) {
    msg += " Inside paint( ).";
    g.drawString(msg, 10, 30);
  }
}
```

这个小应用程序生成的窗口如下所示：

方法stop()和destroy()这里没有阐述，因为在简单的小应用程序中不需要用到它们。

19.5　请 求 重 画

一般来说，只有当小应用程序调用update()和paint()方法时，它才能够向窗口进行写操作。这会产生一个有趣的问题：当小应用程序窗口的信息发生改变时，它如何完成窗口的

更新？例如，如果小应用程序正在显示一条移动的横幅，它使用什么机制在横幅移动时将窗口更新？记住，对小应用程序有一条体系结构上的基本规定，就是小应用程序必须迅速将控制权交给AWT运行环境。例如，它不能在paint()中产生循环从而使横幅滚动。这样就会阻止控制权交还给AWT。在这个规定下，向你的小应用程序窗口的输出看起来会有些难度。可幸运的是，这种情况并不会发生。不管什么时候你的小应用程序需要更新窗口所显示的信息，它仅仅调用repaint（）就够了。

repaint()方法是AWT所定义的。它使得AWT的实时运行环境执行对小应用程序的update()方法的调用，update()方法在它的默认方式下会调用paint()。这样，只需存储输出结果，之后调用repaint()。AWT将接着执行对paint()的调用，以显示所存储的信息。例如，如果你的小应用程序需要输出一个字符串，可以先将字符串存储在String变量中，之后调用repaint()。在paint()中，你将用drawString()输出这个字符串。

repaint()方法有四种形式，让我们逐个看一看。其中最简单的一种形式如下所示：

```
void repaint( )
```

这种方式使得整个窗口都被重画。下面的另一种方式指定了窗口中需要被重画的区域：

```
void repaint(int left, int top, int width, int height)
```

这里区域左上角的坐标由left和top两个参数规定，而区域的宽与高由weight和height所规定。上述尺寸以像素为单位。通过规定需要重画的区域你可以节省时间。窗口更新的代价在时间上来看是十分昂贵的。如果你仅需要更新窗口的很小一部分，那么只重画那一部分将会更高率。

调用repaint()在本质上是要求你的小应用程序窗口在不久以后某时被重画。但是，如果你的系统很慢或正忙，update()将不会立即被调用。在短时间内如果出现多个重画请求，它们可能会被AWT以某种方式瓦解。因此，update()方法仅仅被零散的调用。这在很多情况下会造成问题，在包括动画演示时，需要不断进行update()调用。解决这个问题的一个办法是使用repaint()方法的如下形式：

```
void repaint(long maxDelay)
void repaint(long maxDelay, int x, int y, int width, int height)
```

这里，参数maxDelay规定了update()被调用之前的所能经过的最大的毫秒数。要小心，如果时间在update()能被调用之前已经过去，则update()将不被调用。这时既没有返回值，也没有异常，因此你必须要小心。

注意：使用除paint()或update()以外的其它方法向小应用程序窗口进行输出也是可能的。要这样做的话，就必须调用getGraphics()（由Component类）获得图形信息,再利用这些信息向窗口输出。但对于大多数应用来说，当窗口的内容发生变化时通过paint()将结果送到窗口并调用repaint()效果将更好且更简单。

19.5.1 一个简单的banner小应用程序

为了说明repaint()方法，这里开发了一个简单的banner小应用程序例子程序。这个小应

用程序在窗口中显示了一条从左向右滚动的消息。由于消息的滚动是重复的，它以一个单独的线程来实现，这条线程在小应用程序初始化时生成。Banner小应用程序如下所示：

```
/* A simple banner applet.

   This applet creates a thread that scrolls
   the message contained in msg right to left
   across the applet's window.
*/
import java.awt.*;
import java.applet.*;
/*
<applet code="SimpleBanner" width=300 height=50>
</applet>
*/

public class SimpleBanner extends Applet implements Runnable {
  String msg = " A Simple Moving Banner.";
  Thread t = null;
  int state;
  boolean stopFlag;

  // Set colors and initialize thread.
  public void init() {
    setBackground(Color.cyan);
    setForeground(Color.red);
  }

  // Start thread
  public void start() {
    t = new Thread(this);
    stopFlag = false;
    t.start();
  }

  // Entry point for the thread that runs the banner.
  public void run() {
    char ch;

    // Display banner
    for( ; ; ) {
      try {
        repaint();
        Thread.sleep(250);
        ch = msg.charAt(0);
        msg = msg.substring(1, msg.length());
        msg += ch;
        if(stopFlag)
          break;
      } catch(InterruptedException e) {}
    }
  }

  // Pause the banner.
```

```
  public void stop() {
    stopFlag = true;
    t = null;
  }

  // Display the banner.
  public void paint(Graphics g) {
    g.drawString(msg, 50, 30);
  }
}
```

以下是上例的输出：

让我们仔细看一看这个applet是如何操作的：首先，SimpleBanner扩展了Applet，并且还实现了Runnable接口。因为小应用程序要生成另一条线程来完成横幅的滚动，这一点是十分必要的。前景和背景颜色的设置在init()中完成。

初始化之后，AWT的运行期间系统调用start()来启动小应用程序的运行。在start()方法中，一个新的线程被生成，被命名为线程t。接着，控制小应用程序执行的逻辑变量stopFlag被设为false。然后，线程t通过调用t.start()被启动。t.start（）会调用一个由Thread所定义的方法，使run()函数开始执行，而不会引起对Applet定义的start（）的调用。这是两种独立的方法。

在run（）中，包含在msg中的字符串中的字符不停的查复向左旋转。在每两次旋转之间，对repaint()的调用都要被执行一次。每两次循环之间，run()方法都要休眠0.25秒。run()方法执行的效果是msg的内容在屏幕上不停的从右向左移动。stopFlag 变量在每个循环中被检查，当它的值为true时，run()方法被中止。

如果浏览器在显示这个小应用程序窗口时，要查看一个新页面，则stop()方法就被调用，它将stopFlag设为true，使run()方法中止。这就是当线程的页面被关闭时用来中止线程的方法。当小应用程序窗口再次被浏览时，start()方法被再次调用，从而生成新的线程来执行banner程序。

19.6 使用状态窗口

小应用程序除了在自己的窗口上显示信息之外，还能向它所在运行的浏览器或阅读器的状态窗口输出消息。完成此项功能，只需调用showStatus()方法并加上你想显示的字符串即可。状态窗口是一个很好的工具，它能反馈给用户小应用程序进行的情况，提供选项，或报告错误类型。状态窗口还能帮助调试，因为它能很容易的输出有关你的小应用程序的

信息。

下面的小应用程序示范了showStatus()方法：

```
// Using the Status Window.
import java.awt.*;
import java.applet.*;
/*
<applet code="StatusWindow" width=300 height=50>
</applet>
*/

public class StatusWindow extends Applet{
  public void init() {
    setBackground(Color.cyan);
  }

  // Display msg in applet window.
  public void paint(Graphics g) {
    g.drawString("This is in the applet window.", 10, 20);
    showStatus ("This is shown in the status window.");
  }
}
```

程序的标准输出如下所示：

19.7　HTML 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，但本书使用APPLET）。

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Netscape Navigator，Internet Explorer和HotJava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。现在我们来更仔细的看一看。

标准的APPLET标记的语法如下所示，被括起来的项是可选的：

```
< APPLET
  [CODEBASE = codebaseURL]
  CODE = appletFile
  [ALT = alternateText]
  [NAME = appletInstanceName]
```

```
  WIDTH = pixels HEIGHT = pixels
  [ALIGN = alignment]
  [VSPACE = pixels] [HSPACE = pixels]
>
[< PARAM NAME = AttributeName VALUE = AttributeValue>]
[< PARAM NAME = AttributeName2 VALUE = AttributeValue>]
 . . .
[HTML Displayed in the absence of Java]
</APPLET>
```

我们来看一看每一部分的含义：

CODEBASE CODEBASE是一项可选的属性，它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。

CODE 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文件的名子。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）。

ALT ATL标记是个可选项，用来指定一条短的文本消息，当浏览器能理解APPLET标记但当前不能运行小应用程序时，这条消息就会被显示。这种方法与你为那些不支持小应用程序的浏览器提供替换的HTML文件是截然不同的。

NAME NAME是个可选项，用来指定小应用程序实例的名字。Applet的命名必须使与同一页上的所有的小应用程序能够通过名字互相找到并通信。为获得一个小应用程序程序的名字可用getApplet()方法，它由AppletContent接口所定义。

WIDTH和HEIGHT WIDTH 和HEIGHT是必要的属性,它们给出了小应用程序显示区域的尺寸。

ALIGN ALIGN是个可选项，它规定了小应用程序的对齐方式。它像HTML的IMG标记一样来被处理。ALIGN有以下的可能取值：LEFT，RIGHT，TOP，BOTTOM， MIDDLE，BASELINE，TEXTTOP，ABSMIDDLE，和ABSBOTTOM。

VSPACE 和 HSPACE VSPACE和HSPACE属性是可选的。VSPACE规定了小应用程序的以像素为单位的上下空间大小。HSPACE规定了小应用程序的以像素为单位的左右空间大小。它们都像IMG标记的VSPACE和HSPACE属性一样来被处理。

PARAM NAME 和VALUE PARAM标记允许你来指定小应用程序在HTML页面中的某些特定的属性。小应用程序用getParameter()方法可以获得它们的属性。

处理较早的浏览器 一些较早的Web浏览器不能执行小应用程序和识别APPLET标记。虽然这样的浏览器现在已经很少了（已被Java兼容的版本所代替），但你可能还要在一段时间内面对这个问题。在遇到这种浏览器时，设计你的HTML网页的最好的办法是在你的<applet></applet>标记中加入HTML文本。如果小应用程序的标记不能被你的浏览器识别，你可以看看另一个标记。如果Java可用的话，它将耗尽<applet></applet>标记之间的所有标记并忽视预备的标记。

这儿是一个HTML文件，它在Java中启动了一个名为SampleApplet的小应用程序，并将

消息显示在老版本的浏览器上。

```
<applet code="SampleApplet" width=200 height=40>
 If you were driving a Java powered Navigator,
 you'd see &quote;A Sample Applet&quote; here.<p>
</applet>
```

19.8　向小应用程序传递参数

刚才讨论过，HTML中的APPLET标记允许你向你的小应用程序传递参数。接受参数可用getParameter()方法。它以字符串的方式从特定的参数中返回值。这样，对于数字或逻辑值，你需要将它们从字符串形式转为原来的形式。这儿有一个例子说明参数的传递。

```
// Use Parameters
import java.awt.*;
import java.applet.*;
/*
<applet code="ParamDemo" width=300 height=80>
<param name=fontName value=Courier>
<param name=fontSize value=14>
<param name=leading value=2>
<param name=accountEnabled value=true>
</applet>
*/

public class ParamDemo extends Applet{
  String fontName;
  int fontSize;
  float leading;
  boolean active;

  // Initialize the string to be displayed.
  public void start() {
    String param;

    fontName = getParameter("fontName");
    if(fontName == null)
      fontName = "Not Found";

    param = getParameter("fontSize");
    try {
      if(param != null) // if not found
        fontSize = Integer.parseInt(param);
      else
        fontSize = 0;
    } catch(NumberFormatException e) {
      fontSize = -1;
    }

    param = getParameter("leading");
    try {
      if(param != null) // if not found
```

```
        leading = Float.valueOf(param).floatValue();
      else
        leading = 0;
    } catch(NumberFormatException e) {
      leading = -1;
    }

    param = getParameter("accountEnabled");
    if(param != null)
      active = Boolean.valueOf(param).booleanValue();
  }

  // Display parameters.
  public void paint(Graphics g) {
    g.drawString("Font name: " + fontName, 0, 10);
    g.drawString("Font size: " + fontSize, 0, 26);
    g.drawString("Leading: " + leading, 0, 42);
    g.drawString("Account Active: " + active, 0, 58);
  }
}
```

程序的标准输出如下所示：

如程序所示，用getParameter()方法测试返回值，如果参数无效，将返回null。向数字类型的转换必须在try语句中进行，try语句能捕获NumberFormatException，捕获不到的异常情况在小应用程序中决不会发生。

19.8.1 对程序Banner Applet的改进

使用参数来改进前面所示的banner 小应用程序是可能的。在以前的版本中，滚动的消息是通过硬编码进小应用程序的。但如果将消息作为一个参数传递，就能允许banner 小应用程序在每次执行时显示一个不同的消息。被改进的版本如下所示。注意，文件顶部的APPLET标记指定了名为message的参数，它与括号里的字符串相关联。

```
// A parameterized banner
import java.awt.*;
import java.applet.*;
/*
<applet code="ParamBanner" width=300 height=50>
<param name=message value="Java makes the Web move!">
</applet>
*/
```

```
public class ParamBanner extends Applet implements Runnable {
  String msg;
  Thread t = null;
  int state;
  boolean stopFlag;

  // Set colors and initialize thread.
  public void init() {
    setBackground(Color.cyan);
    setForeground(Color.red);
  }

  // Start thread
  public void start() {
    msg = getParameter("message");
    if(msg == null) msg = "Message not found.";
    msg = " " + msg;
    t = new Thread(this);
    stopFlag = false;
    t.start();
  }

  // Entry point for the thread that runs the banner.
  public void run() {
    char ch;

    // Display banner
    for( ; ; ) {
      try {
        repaint();
        Thread.sleep(250);
        ch = msg.charAt(0);
        msg = msg.substring(1, msg.length());
        msg += ch;
        if(stopFlag)
          break;
      } catch(InterruptedException e) {}
    }
  }

  // Pause the banner.
  public void stop() {
    stopFlag = true;
    t = null;
  }

  // Display the banner.
  public void paint(Graphics g) {
    g.drawString(msg, 50, 30);
  }
}
```

19.9 getDocumentBase( )和getCodeBase( )

你常常会生成一些小应用程序需要装载媒体和文本。Java将允许小应用程序从启动小应用程序的HTML文件所在的路径（文件数据库）装载数据，同样也允许小应用程序从它的类文件所被装载的路径（代码数据库）装载数据。这些路径都由url对象用getDucumentBase()方法和getCodeBase()方法返回。它们能与你想要装载的文件的名字相联系。要实际装载另一个文件，用AppletContext接口定义的方法showDocument()，它将在下一章讨论。

下面的小应用程序说明了这些方法：

```
// Display code and document bases.
import java.awt.*;
import java.applet.*;
import java.net.*;
/*
<applet code="Bases" width=300 height=50>
</applet>
*/

public class Bases extends Applet{
  // Display code and document bases.
  public void paint(Graphics g) {
    String msg;

    URL url = getCodeBase(); // get code base
    msg = "Code base: " + url.toString();
    g.drawString(msg, 10, 20);

    url = getDocumentBase(); // get document base
    msg = "Document base: " + url.toString();
    g.drawString(msg, 10, 40);
  }
}
```

程序输出如下所示：

19.10　AppletContext 和showDocument( )

Java的一个应用是使用活动图像和动画为浏览Web提供一种图形方法，这比超文本所使用的带蓝色下划线的文字有趣的多。为了允许你的小应用程序将控制权交给另一个URL，你必须使用由AppletContext接口定义的方法showDocument()。AppletContext是使你从小应用程序的执行环境中获取信息的接口。这些由AppletContext所定义的方法如表19-2所示。当前正在执行的小应用程序的环境上下文可通过对getAppletContext()方法的调用来获得，此方法由Applet所定义。

表 19-2　由 AppletContext 接口定义的抽象方法

方法	描述
Applet getApplet(String appletName)	如果它在当前的小应用程序上下文中的话，返回名为 appletName 的小应用程序，否则，null被返回
Enumeration getApplets( )	返回当前小应用程序上下文中所有的小应用程序的枚举
AudioClip getAudioClip(URL url)	返回一个AudioClip 对象，它封装了在url所指定的位置找到的音频片断
Image getImage(URL url)	返回一个Image对象，它封装了在url所指定的位置找到的图像
void showDocument(URL url)	浏览由url所指定的URL中的文件。此方法不能被小应用程序阅读器支持
void showDocument(URL url, String where)	浏览由url所指定的URL中的文件。此方法不能被小应用程序阅读器支持。文件的位置由where设定
void showStatus(String str)	显示状态窗口中的str

在一个小应用程序中，一旦你获得了小应用程序的上下文，你就可以通过调用方法showDocument()来阅读其他文件。这个方法没有返回值，如果不成功也没有异常出现，因此使用要小心。showDocument( )方法有两种形式，showDocument(URL)在特定的URL中显示文档，showDocument(URL，where)在浏览器窗口的特定位置显示特定的文件，where的有效值包括“_self”（在当前框架中显示），“_parent”（在父框架中显示）“_top”（在顶端框架中显示），“_blank”（在新的浏览器窗口中显示）。你还可以指定一个名字，使文件在一个以它命名的浏览器窗口中显示。

下面的小应用程序说明了AppletContext和showDocument( )方法。在执行时，它获得了当前小应用程序的上下文，并用上下文将控制权交给一个名为Test.html的文件。这个文件必须与小应用程序在同一目录下。Test.html.能包含任何有效的超文本。

```
/* Using an applet context, getCodeBase(),
   and showDocument() to display an HTML file.
*/

import java.awt.*;
import java.applet.*;
```

```
import java.net.*;
/*
<applet code="ACDemo" width=300 height=50>
</applet>
*/

public class ACDemo extends Applet{
  public void start() {
    AppletContext ac = getAppletContext();
    URL url = getCodeBase(); // get url of this applet

    try {
      ac.showDocument(new URL(url+"Test.html"));
    } catch(MalformedURLException e) {
      showStatus("URL not found");
    }
  }
}
```

19.11 AudioClip接口

AudioClip接口定义了如下方法：play()(从开始播放音频片断)，stop()（停止播放音频片断），loop()（连续循环的播放音频片断）。在用getAudioClip( )装载了一个音频片断后就可以用这些方法来播放它。

19.12 AppletStub 接口

AppletStub 接口提供了小应用程序与浏览器之间通信的方式。你的程序代码通常不实现这个接口。

19.13 向控制台的输出

虽然向小应用程序窗口的输出必须由AWT方法来完成，例如drawString()，在一些特殊的情况下，如为了调试的目的，在小应用程序中用控制台输出也是可能的。在小应用程序中，当你调用一个System.out.println( )方法时，输出将不被送到你的小应用程序窗口。相反，它或者出现在你启动小应用程序阅读器的控制台部分，或者在一些浏览器可用的控制台出现。如果不是以调试为目的而使用控制台输出，则效果并不令人满意，因为它违反了大多数用户认可的图形接口的设计准则。

第 20 章　事 件 处 理

本章讲述一个与Java小应用程序有关的重要方面：事件。正如我们在19章所说的那样，小应用程序是基于事件驱动的。那么，事件处理就是编写一个成功的小应用程序的核心了。你的许多小应用程序需要响应的事件是被用户触发的。这些事件以各种各样的方式传递给你的小应用程序，而特定的方法依赖于实际的事件。事件有很多种类型。经常处理的事件是那些由鼠标，键盘和按钮等各种控件触发的事件。这些事件在java的java.awt.event包中被提供。

在这一章，我们先看一下Java的事件处理机制。然后再讲述主要的事件类和接口，并开发几个有关基本事件处理过程的示例程序。这一章中也解释了如何使用adapter类，inner类和匿名的inner类去顺利地编写事件处理代码。在本书剩下的部分提供的例子中频繁的利用了这些技术。

20.1　两种事件处理机制

在我们开始讨论事件处理之前，必须明确一点：Java原始的1.0版和现在开始于1.1版的版本之间在小应用程序处理事件的方式上有了根本的变化。1.0版的事件处理方法仍然被支持，但是不推荐在新的程序中应用。同时，许多支持老的1.0事件处理模型的方法已经不被推荐使用。新的方法应该被所有新的程序中应用，其中也包括那些为Java2编写的程序，因而也被这本书中所提供的程序所使用。

20.2　授权事件模型

现在处理事件的方法是基于授权事件模型（delegation event model）的，这种模型定义了标准一致的机制去产生和处理事件。它的概念十分简单:一个源（source）产生一个事件（event）并把它送到一个或多个的监听器（listeners）那里。在这种方案中，监听器简单地等待，直到它收到一个事件。一旦事件被接受，监听器将处理这些事件，然后返回。这种设计的优点是那些处理事件的应用程序可以明确地和那些用来产生那些事件的用户接口程序分开。一个用户接口元素可以授权一段特定的代码处理一个事件。

在授权事件模型中，监听器为了接受一个事件通知必须注册。这样有一个重要的好处：通知只被发送给那些想接受的它们的监听器那里。这是一种比Java 1.0版设计的方法更有效的处理事件的方法。以前，一个事件按照封装的层次被传递直到它被一个组件处理。这需要组件接受那些它们不处理的事件，所以这样浪费了宝贵的时间。而授权事件模型去掉了这个开销。

注意：Java也允许你处理事件而不采用授权事件模型。这可以通过扩展一个awt组件来实现。这种技术在第22章的结尾被讨论。然而，授权事件模型是首选的方案。

接下来给出了事件的定义并且描述了事件源与监听器的任务。

20.2.1 事件

在授权事件模型中，一个事件是一个描述了事件源的状态改变的对象。它可以作为一个人与图形用户接口相互作用的结果被产生。一些产生事件的活动可以是通过按一个按钮，用键盘输入一个字符，选择列表框中的一项，点击一下鼠标。许多别的用户操作也能作为例子列出。

事件可能不是由于用户接口的交互而直接发生的。例如，一个事件可能由于在定时器到期，一个计数器超过了一个值，一个软件或硬件错误发生，或者一个操作被完成而产生。你还可以自由地定义一些适用于你的应用程序的事件。

20.2.2 事件源

一个事件源是一个产生事件的对象。当这个对象内部的状态以某种方式改变时,事件就会产生。事件源可能产生不止一种事件。

一个事件源必须注册监听器以便监听器可以接受关于一个特定事件的通知。每一种事件有它自己的注册方法。这里是通用的形式：

```
public void addTypeListener(TypeListener el)
```

在这里，type是事件的名称，而el是一个事件监听器的引用。例如，注册一个键盘事件监听器的方法被叫做addKeyListener( )，注册一个鼠标活动监听器的方法被叫做addMouseMotionListener( )，当一个事件发生时，所有被注册的监听器都被通知并收到一个事件对象的拷贝。这就是大家知道的多播（multicasting）。在所有的情况下，事件通知只被送给那些注册接受它们的监听器。

一些事件源可能只允许注册一个监听器。这种方法的通用形式如下所示:

```
public void addTypeListener(TypeListener el)
    throws java.util.TooManyListenersException
```

在这里，type是事件的名称而el是一个事件监听器的引用。当这样一个事件发生时，被注册的监听器被通知。这就是大家知道的单播事件。

一个事件源必须也提供一个允许监听器注销一个特定事件的方法。这个方法的通用形式如下所示：

```
public void removeTypeListener(TypeListener el)
```

这里，type是事件的名字而el是一个事件监听器的引用。例如，为了注销一个键盘监听器，你将调用removeKeyListener( )函数。

这些增加或删除监听器的方法被产生事件的事件源提供。例如，component类提供了那

些增加或删除键盘和鼠标事件监听器的方法。

20.2.3　事件监听器

一个事件监听器是一个在事件发生时被通知的对象。它有两个要求。首先，为了可以接受到特殊类型事件的通知它必须在事件源中已经被注册。第二，它必须实现接受和处理通知的方法。

用于接受和处理事件的方法在java.awt.event中被定义为一系列的接口。例如，MouseMotionListener接口定义了两个在鼠标被拖动时接受通知的方法。如果实现这个接口，任何对象都可以接受并处理这些事件的一部分。许多别的监听器接口以后将在别的章中被讨论。

20.3　事 件 类

Java事件处理机制的核心是这些代表事件的类。因而，我们从浏览事件类开始学习事件处理。正如你将看到的，它们提供一个一致而又易用的封装事件的方法。

在java.util中被封装的EventObject类是Java事件类层次结构的根节点。它是所有事件类的父类。它的一个构造函数如下所示:

```
EventObject(Object src)
```

这里，src是一个可以产生事件的对象。

EventObject类包括两个方法：getSource( )和toString( )。GetSource( )方法返回的是事件源。它通常的形式如下所示:

```
Object getSource( )
```

正如所期望的一样，返回的是等价于事件的一个字符串。

在java.awt包中被定义的AWTEvent类是EventObject类的子类。同时作为所有基于awt的事件的父类（不论直接还是间接），它在授权事件模型中被使用。它的getID()方法可以被用来决定事件的类型。这个方法的形式如下所示：

```
int getID( )
```

关于AWTEvent类的细节，在第22章将进一步讨论。需要明确的是在本节中我们讨论的所有其他类都是AWTEvent子类。

小结：

- EventObject 是所有时间类的父类
- AWTEvent是所有在授权事件模型中处理的AWT事件类的父类

java.awt.event这个包定义了一些能被各种用户接口单元产生的事件类型。在表20-1中列举了这些事件类中最重要的一些并对它们的产生条件进行了简要的描述。每一类中最常用的构造函数及其他方法将在下一节中讲述。

表 20-1 java.awt.event 中的主要事件类

事件类	描述
ActionEvent	通常在按下一个按钮，双击一个列表项或者选中一个菜单项时发生
AdjustmentEvent	当操作一个滚动条时发生
ComponentEvent	当一个组件隐藏，移动，改变大小或成为可见时发生
ContainerEvent	当一个组件从容器中加入或删除时发生
FocusEvent	当一个组件获得或失去键盘焦点时发生
InputEvent	所有组件的输入事件的抽象超类
ItemEvent	当一个复选框或列表项被点击时发生；当一个选择框或一个可选择菜单的项被选择或取消时发生
KeyEvent	当输入从键盘获得时发生
MouseEvent	当鼠标被拖动，移动，点击，按下，释放时发生；或者在鼠标进入或退出一个组件时发生
TextEvent	当文本区和文本域的文本改变时发生
WindowEvent	当一个窗口激活，关闭，失效，恢复，最小化，打开或退出时发生

20.3.1 ActionEvent 类

在一个按钮被按下，列表框中的一项被选择，或者是一个菜单项被选择时都会产生一个ActionEvent类型的事件。在ActionEvent类中定义了四个用来表示功能修改的整型常量：ALT_MASK，CTRL_MASK，META_MASK和SHIFT_MASK。除此之外，还有一个整型常量ACTION_PERFORMED用来标识事件。

ActionEvent类有两个构造函数：

```
ActionEvent(Object src, int type, String cmd)
ActionEvent(Object src, int type, String cmd, int modifiers)
```

在这里，src是一个事件源对象的引用。事件的类型由type指定，cmd是它的命令字符串，modifiers这个参数显示了在事件发生时，ALT, CTRL, META, 或 SHIFT中的哪一个修改键被按下。

你可以通过调用ActionEvent对象的getActionCommand( )方法来获得命令的名字，下面是这个方法。

```
String getActionCommand( )
```

例如，当一个按钮被按下时，一个ActionEvent类事件被产生，它的命令名和按钮上的标签相同。

```
int getModifiers( )
```

这个方法返回了一个值，它表示了在事件产生时ALT, CTRL, META, 或 SHIFT这些修改键哪一个被按下。

20.3.2　AdjustmentEvent 类

一个AdjustmentEvent类的事件由一个滚动条产生。调整事件有五种类型。在AdjustmentEvent类中定义了用于标识它们的整型常量。这些常量和意义在下面列出：

BLOCK_DECREMENT	用户点击滚动条内部减少这个值
BLOCK_INCREMENT	用户点击滚动条内部增加这个值
TRACK	滑块被拖动
UNIT_DECREMENT	滚动条端的按钮被点击减少它的值
UNIT_INCREMENT	滚动条端的按钮被点击增加它的值

除此之外，还有一个整数常量ADJUSTMENT_VALUE_CHANGED，它用来表示改变已经发生。

AdjustmentEvent类有两个构造函数：

```
AdjustmentEvent(Adjustable src, int id, int type, int data)
```

在这里，src是一个产生事件的对象的引用。id等于ADJUSTMENT_VALUE_CHANGED这个常量。事件的类型由type指定， data是与它相关的数据。

getAdjustable( )方法返回了产生事件的对象。它的形式如下所示：

```
Adjustable getAdjustable( )
```

通过getAdjustmentType( )方法，可以获得调整事件的类型。它返回被AdjustmentEvent定义的常量之一。下面是通常的形式：

```
int getAdjustmentType( )
```

调整数量可以通过getValue( )方法获得，它的原形如下所示：

```
int getValue( )
```

例如，当一个滚动条被调整时，这个方法返回了代表变化的值。

20.3.3　ComponentEvent 类

一个ComponentEvent事件通常在一个组件的大小、位置或者是可视性发生了改变时产生。组件的事件类型有四种。ComponentEvent这个类定义了用于标识它们的整型常量。这些常量和它们的意义如下所示：

COMPONENT_HIDDEN	组件被隐藏
COMPONENT_MOVED	组件被移动
COMPONENT_RESIZED	组件被改变大小
COMPONENT_SHOWN	组件被显示

ComponentEvent类有这样一个构造函数：

```
ComponentEvent(Component src, int type)
```

在这里，src是产生事件的对象的引用。Type指定了事件的类型。

ComponentEvent 类是 ContainerEvent, FocusEvent, KeyEvent, MouseEvent 和 WindowEvent这几个类的父类。

getComponent( )方法返回了产生事件的组件。它的形式如下所示：

```
Component getComponent( )
```

20.3.4 ContainerEvent 类

一个ContainerEvent事件是在容器中被加入或删除一个组件时产生的。容器有两种事件类型。在ContainerEvent类中定义了用于标识它们的整型常量：COMPONENT_ADDED和COMPONENT_REMOVED。它们表示了在容器中加入和删除一个组件。

ContainerEvent是ComponentEvent类的子类，它有如下所示构造函数：

```
ContainerEvent(Component src, int type, Component comp)
```

在这里，src是产生事件的容器的引用。Type指定了事件的类型。Comp指定了从容器中被加入或删除的组件。

你可以通过调用getContainer( )方法来获得产生这个事件的容器的一个引用，它的形式如下所示：

```
Container getContainer( )
```

通过调用getChild( )方法可以返回在容器中被加入或删除的组件。它的通常形式如下所示：

```
Component getChild( )
```

20.3.5 FocusEvent 类

一个 FocusEvent 是在一个组件获得或失去输入焦点时产生。这些事件用 FOCUS_GAINED 和 FOCUS_LOST这两个整型变量来表示。

FocusEvent类是ComponentEvent类的子类，它有两个构造函数：

```
FocusEvent(Component src, int type)
FocusEvent(Component src, int type, boolean temporaryFlag)
```

在这里，src是产生事件的组件的引用。Type指定了事件的类型。如果焦点事件是暂时的，那么参数temporaryFlag被设为true。否则，它是false（一个暂时焦点事件被作为另一个用户接口操作的结果产生。例如，假如焦点在一个文本框中，如果用户移动鼠标去调整滚动条，这个焦点就会被暂时失去。）

通过调用isTemporary( )方法可以知道焦点的改变是否是暂时的。它的调用形式如下所示：

```
boolean isTemporary( )
```

如果这个改变是暂时的，那么这个方法返回true，否则返回false。

20.3.6　InputEvent类

InputEvent抽象类是ComponentEvent类的子类，同时是一个组件输入事件的父类。它的子类包括：KeyEvent类和MouseEvent类。在InputEvent类中定义了如下所示的八个整型常量，它们被用来获得任何和这个事件有关的修改符的信息。

ALT_MASK	BUTTON2_MASK	META_MASK
ALT_GRAPH_MASK	BUTTON3_MASK	SHIFT_MASK
BUTTON1_MASK	CTRL_MASK	

isAltDown( )，isAltGraphDown( )，isControlDown( )，isMetaDown( )和isShiftDown( )等方法用来测试是否在事件发生时相应的修改符被按下。这些方法如下所示：

```
boolean isAltDown( )
boolean isAltGraphDown( )
boolean isControlDown( )
boolean isMetaDown( )
boolean isShiftDown( )
```

通过调用方法可以返回一个值，包含这个事件所有修改符的标志。如下所示：

```
int getModifiers( )
```

20.3.7　ItemEvent类

一个ItemEvent事件是当一个复选框或者列表框被点击，或者是一个可选择的菜单项被选择或取消选定时产生（复选框和列表框在本书的后面将作论述）。这个项事件有两种类型，它们可以用如下所示的整型常量标识。

DESELECTED	用户取消选定的一项
SELECTED	用户选择一项

除此之外，ItemEvent类还定义了一个整型常量ITEM_STATE_CHANGED，用它来表示一个状态的改变。

ItemEvent类有这样一个构造函数：

```
ItemEvent(ItemSelectable src, int type, Object entry, int state)
```

在这里，src是一个产生事件组件的引用。例如，它可能是一个列表或可选择元素。Type指定了事件的类型。产生该项事件的特殊项在entry中被传递。该项当前的状态由state表示。

GetItem（）方法能被用来获得一个产生事件的项的引用。如下所示：

```
Object getItem( )
```

getItemSelectable( )方法能被用来获得一个产生事件的ItemSelectable对象的引用。如下所示：

```
ItemSelectable getItemSelectable( )
```

列表框和可选框就是实现了ItemSelectable接口的用户接口元素的例子。

getStateChange( )方法返回了事件对应的状态（如选择或取消）。如下所示：

```
int getStateChange( )
```

20.3.8 KeyEvent类

一个KeyEvent事件是当键盘输入发生时产生。键盘事件有三种，它们分别用整型常量：KEY_PRESSED，KEY_RELEASED和KEY_TYPED来表示。前两个事件在任何键被按下或释放时发生。而最后一个事件只在产生一个字符时发生。请记住，不是所有被按下的键都产生字符。例如，按下SHIFT键就不能产生一个字符。

还有许多别的整型常量在KeyEvent类中被定义。例如，从VK_0到VK_9和从VK_A到VK_Z定义了与这些数字和字符等价的ASCII码。这里还有一些其他的：

VK_ENTER	VK_ESCAPE	VK_CANCEL	VK_UP
VK_DOWN	VK_LEFT	VK_RIGHT	VK_PAGE_DOWN
VK_PAGE_UP	VK_SHIFT	VK_ALT	VK_CONTROL

VK常量指定了虚拟键值（virtual key codes）并且与任何control，shift或alt修改键不相关。

KeyEvent类是InputEvent类的子类，它有这样两个构造函数：

```
KeyEvent(Component src, int type, long when, int modifiers, int code)
KeyEvent(Component src, int type, long when, int modifiers, int code, char
ch)
```

在这里，src是一个产生事件的组件的引用。Type指定了事件的类型。当这个键被按下时，系统时间在when里被传递。参数Modifiers 决定了在键盘事件发生时那一个修改符被按下。像VK_UP和VK_A这样的虚拟键值在code中传递。如果与这些虚拟键值相对应的字符存在，则在ch中被传递，否则ch中是CHAR_UNDEFINED。对于KEY_TYPED事件，code将是VK_UNDEFINED。

KeyEvent类定义了一些方法，但是其中用的最多的是用来返回一个被输入的字符的方法和用来返回键值的方法getKeyCode()。它们的通常形式如下所示：

```
char getKeyChar( )
int getKeyCode( )
```

如果没有合法的字符可以返回，getKeyChar( )方法将返回CHAR_UNDEFINED。同样，在一个KEY_TYPED事件发生时，getKeyCode()方法返回的是VK_UNDEFINED。

20.3.9 MouseEvent 类

鼠标事件有7种类型。在MouseEvent类中定义了如下所示的整型常量来表示它们：

MOUSE_CLICKED	用户点击鼠标
MOUSE_DRAGGED	用户拖动鼠标

MOUSE_ENTERED	鼠标进入一个组件内
MOUSE_EXITED	鼠标离开一个组件
MOUSE_MOVED	鼠标移动
MOUSE_PRESSED	鼠标被按下
MOUSE_RELEASED	鼠标被释放

MouseEvent类是InputEvent类的子类。它有如下所示的构造函数：

```
MouseEvent(Component src, int type, long when, int modifiers,
                        int x, int y, int clicks, boolean triggersPopup)
```

在这里，src是一个产生事件的组件的引用。Type指定了事件的类型。鼠标事件发生时的系统事件在when中被传递。参数modifiers决定了在鼠标事件发生时哪一个修改键被按下。鼠标的坐标在x，y中传递。点击的次数在clicks中传递。triggersPopup标志决定了是否由这个事件引发在平台上弹出一个弹出式菜单。

在这个类中用的最多的方法是getX( )和getY( )。它们返回了在事件发生时，对应的鼠标所在坐标点的X和Y。形式如下所示：

```
int getX( )
int getY( )
```

相应的，你也可以用getPoint( )方法去获得鼠标的坐标。形式如下所示：

```
Point getPoint( )
```

它返回了一个Point对象，在这个对象中以整数成员变量的形式包含了x和y坐标。

translatePoint( )方法可以改变事件发生的位置。它的形式如下所示：

```
void translatePoint(int x, int y)
```

在这里，参数x和y被加到了该事件的坐标中。

getClickCount( )方法可以获得这个事件中鼠标的点击次数。如下所示：

```
int getClickCount( )
```

isPopupTrigger( )方法可以测试是否这个事件将引起一个弹出式菜单在平台中弹出。如下所示：

```
boolean isPopupTrigger( )
```

20.3.10　The TextEvent Class

这个类的实例描述了文本事件。当字符被用户或程序输入到文本框或文本域时，它们产生了文本事件。TextEvent类定义了整数常量：TEXT_VALUE_CHANGED。

这个类的一个构造函数如下所示：

```
TextEvent(Object src, int type)
```

在这里，src是一个产生事件的对象的引用。Type指定了事件的类型。

TextEvent类不包括在产生事件的文本组件中现有的字符。相反，你的程序必须用其他的与文本组件相关的方法来获得这些信息。这不同于那些其他的在本节中被讨论的事件对象。由于这个原因，这里没有TextEvent类的方法可以讨论。可以想到，一个文本事件通知作为监听器的信号将可以从一个特定的文本组件获得信息。

20.3.11 WindowEvent类

窗口事件有七种类型。在WindowEvent类中定义了用来表示它们的整数常量。这些常量和它们的意义如下所示：

WINDOW_ACTIVATED	窗口被激活
WINDOW_CLOSED	窗口已经被关闭
WINDOW_CLOSING	用户要求窗口被关闭
WINDOW_DEACTIVATED	窗口被禁止
WINDOW_DEICONIFIED	窗口被恢复
WINDOW_ICONIFIED	窗口被最小化
WINDOW_OPENED	窗口被打开

WindowEvent类是ComponentEvent类的子类。它的构造函数如下所示：

```
WindowEvent(Window src, int type)
```

在这里，src是一个产生事件的对象的引用。Type指定了事件的类型。

在这个类中用的最多的方法是getWindow( )。它返回的是产生事件的Window对象。其一般形式如下所示：

```
Window getWindow( )
```

20.4 事 件 源

在表20-2中列举了一些可以产生我们在前面所描述的事件的用户接口组件。除了这些图形用户接口元素之外，其他组件，如一个小应用程序，也可以产生事件。例如，你可以在一个小应用程序中获得键盘和鼠标事件（你可能也建立了你自己的组件，它们也可以产生事件）。在本章中我们将只处理鼠标和键盘事件，但是接下来的两章将处理在表20-2中所列的事件源所产生的事件。

表 20-2 事件源举例

事件源	描述
Button	在按钮被按下时产生动作事件
Checkbox	在复选框被选中或取消时产生项目事件
Choice	在选择项改变时产生项目事件
List	在一项被双击时，产生动作事件，被选择或取消时产生项目事件

续表

事件源	描述
Menu item	菜单项被选中时产生动作事件，当可复选菜单项被选中或取消时产生项目事件
Scrollbar	在滚动条被拖动时产生调整事件
Text components	当用户输入字符时产生文本事件
Window	窗口被激活，关闭，失效，恢复，最小化，打开或退出时产生窗口事件

20.5　事件监听器接口

正如我们前面所解释的，在授权事件模型中有两部分：事件源和监听器。事件源是通过实现一些在java.awt.event包中被定义的接口而生成的。当一个事件产生的时候，事件源调用被监听器定义的相应的方法并提供一个事件对象作为参数。在表20-3中列出了通常用到的监听器接口，同时还简要的说明了它们所定义的方法。接下来将解释每一个接口包含的一些特殊方法。

表 20-3　通常使用的事件监听器接口

接口	描述
ActionListener	定义了一个接受动作事件的方法
AdjustmentListener	定义了一个接受调整事件的方法
ComponentListener	定义了四个方法来识别何时隐藏、移动、改变大小、显示组件
ContainerListener	定义了两个方法来识别何时从容器中加入或除去组件
FocusListener	定义了两个方法来识别何时组件获得或失去焦点
ItemListener	定义了一个方法来识别何时项目状态改变
KeyListener	定义了三个方法来识别何时键按下、释放和键入字符事件
MouseListener	定义了五个方法来识别何时鼠标单击，进入组件，离开组件，按下和释放事件
MouseMotionListener	定义了两个方法来识别何时鼠标拖动和移动
TextListener	定义了一个方法来识别何时文本值改变
WindowListener	定义了七个方法来识别何时窗口激活、关闭、失效、最小化、还原、打开和退出

20.5.1　ActionListener接口

在这个接口中定义了actionPerformed( )方法，当一个动作事件发生时，它将被调用。一般形式如下所示：

```
void actionPerformed(ActionEvent ae)
```

20.5.2 AdjustmentListener 接口

在这个接口中定义了adjustmentValueChanged( )方法，当一个调整事件发生时，它将被调用。其一般形式如下所示：

```
void adjustmentValueChanged(AdjustmentEvent ae)
```

20.5.3 ComponentListener 接口

在这个接口中定义了四个方法，当一个组件被改变大小、移动、显示或隐藏时，它们将被调用。其一般形式如下所示：

```
void componentResized(ComponentEvent ce)
void componentMoved(ComponentEvent ce)
void componentShown(ComponentEvent ce)
void componentHidden(ComponentEvent ce)
```

注意：AWT处理改变大小和移动事件。componentResized( )**和**componentMoved( )**方法只用来提供通知。**

20.5.4 ContainerListener 接口

在这个接口中定义了两个方法，当一个组件被加入到一个容器中时，componentAdded( )方法将被调用。当一个组件从一个容器中删除时，componentRemoved( )方法将被调用。这两个方法的一般形式如下所示：

```
void componentAdded(ContainerEvent ce)
void componentRemoved(ContainerEvent ce)
```

20.5.5 FocusListener 接口

在这个接口中定义了两个方法，当一个组件获得键盘焦点时，focusGained( )方法将被调用。当一个组件失去键盘焦点时，focusLost( )方法将被调用。这两个方法的一般形式如下所示：

```
void focusGained(FocusEvent fe)
void focusLost(FocusEvent fe)
```

20.5.6 ItemListener 接口

在这个接口中定义了itemStateChanged( )方法，当一个项的状态发生变化时，它将被调用。这个方法的原型如下所示：

```
void itemStateChanged(ItemEvent ie)
```

20.5.7 KeyListener 接口

在这个接口中定义了三个方法。当一个键被按下和释放时，相应地keyPressed( )方法和keyReleased( )方法将被调用。当一个字符已经被输入时，keyTyped( )方法将被调用。

例如，如果一个用户按下和释放A键，通常有三个事件顺序产生：键被按下，键入和

释放。如果一个用户按下和释放HOME键时，通常有两个事件顺序产生：键被按下和释放。

这些方法的一般形式如下所示：

```
void keyPressed(KeyEvent ke)
void keyReleased(KeyEvent ke)
void keyTyped(KeyEvent ke)
```

20.5.8　MouseListener 接口

在这个接口中定义了五个方法，当鼠标在同一点被按下和释放时，mouseClicked( )方法将被调用。当鼠标进入一个组件时，mouseEntered( )方法将被调用。当鼠标离开组件时，mouseExited( )方法将被调用。当鼠标被按下和释放时，相应的mousePressed( )方法和mouseReleased( )方法将被调用。

这些方法的一般形式如下所示：

```
void mouseClicked(MouseEvent me)
void mouseEntered(MouseEvent me)
void mouseExited(MouseEvent me)
void mousePressed(MouseEvent me)
void mouseReleased(MouseEvent me)
```

20.5.9　MouseMotionListener 接口

在这个接口中定义了两个方法，当鼠标被拖动时，mouseDragged( )方法将被调用多次。当鼠标被移动时，mouseMoved( )方法将被调用多次。这些方法的一般形式如下所示：

```
void mouseDragged(MouseEvent me)
void mouseMoved(MouseEvent me)
```

20.5.10　TextListener 接口

在这个接口中定义了textChanged( )方法，当文本区或文本域发生变化时，它将被调用。这个方法的一般形式如下所示：

```
void textChanged(TextEvent te)
```

20.5.11　WindowListener 接口

在这个接口中定义了七个方法。当一个窗口被激活或禁止时，windowActivated( )方法或windowDeactivated( )方法将相应地被调用。如果一个窗口被最小化，windowIconified( )方法将被调用。当一个窗口被恢复时，windowDeIconified( )方法将被调用。当一个窗口被打开或关闭时，windowOpened( )方法或windowClosed( )方法将相应地被调用。当一个窗口正在被关闭时，windowClosing( )方法将被调用。

这些方法的一般形式如下所示：

```
void windowActivated(WindowEvent we)
void windowClosed(WindowEvent we)
void windowClosing(WindowEvent we)
void windowDeactivated(WindowEvent we)
void windowDeiconified(WindowEvent we)
```

```
void windowIconified(WindowEvent we)
void windowOpened(WindowEvent we)
```

20.6 使用授权事件模型

现在你已经学习了授权事件模型的原理，并且对它的各种组件有了总体的认识。下面让我们来具体实践一下。采用了授权事件模型编程的小应用程序确实十分简单。只需要如下所示两步：

1. 在监听器中实现相应的监听器接口，以便接受相应的事件。
2. 实现注册或注销（如果必要）监听器的代码，以便可以得到事件的通知。

请记住，一个事件源可能产生多种类型的事件。每一个事件都必须分别注册。当然，一个对象可以注册接受多种事件，但是它必须实现相应的所有事件监听器的接口。

为了明白授权事件模型实际上是如何工作的，我们将分析一个例子，在这个例子中处理了两个最常用的事件产生器：鼠标和键盘。

20.6.1 处理鼠标事件

为何处理鼠标事件，你必须实现MouseListener接口和MouseMotionListener接口。接下来的小应用程序说明了这个过程。它在小应用程序所在窗口的状态栏中显示了鼠标的当前坐标。每当鼠标按钮被按下，在鼠标指针所在的位置将显示“Down”。而当鼠标按钮释放时，将显示“Up”。如果鼠标按钮被点击，“Mouse clicked”将被显示在小应用程序显示区域的左上角。

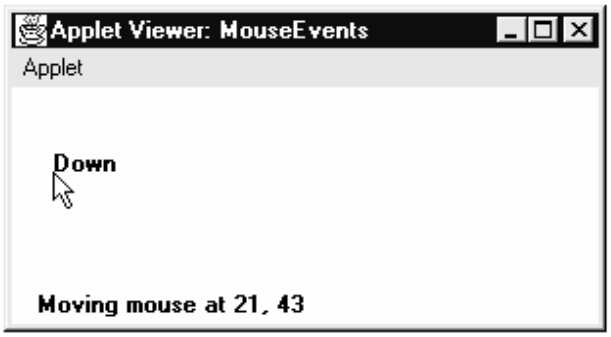
当鼠标进入或退出小应用程序窗口时，在小应用程序显示区域的左上角将显示一个消息。当你拖动鼠标时，跟随着被拖动的鼠标一个*将被显示。在这里注意两个变量：mouseX和mouseY，它们存放着在鼠标的按下，释放或拖动事件发生时鼠标的位置。接下来，这些坐标将在paint()方法中被使用，以便在这些事件发生的点显示输出。

```
// Demonstrate the mouse event handlers.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="MouseEvents" width=300 height=100>
  </applet>
*/

public class MouseEvents extends Applet
  implements MouseListener, MouseMotionListener {

  String msg = "";
  int mouseX = 0, mouseY = 0; // coordinates of mouse

  public void init() {
     addMouseListener(this);
     addMouseMotionListener(this);
```

```
}

// Handle mouse clicked.
public void mouseClicked(MouseEvent me) {
  // save coordinates
  mouseX = 0;
  mouseY = 10;
  msg = "Mouse clicked.";
  repaint();
}

// Handle mouse entered.
public void mouseEntered(MouseEvent me) {
  // save coordinates
  mouseX = 0;
  mouseY = 10;
  msg = "Mouse entered.";
  repaint();
}

// Handle mouse exited.
public void mouseExited(MouseEvent me) {
  // save coordinates
  mouseX = 0;
  mouseY = 10;
  msg = "Mouse exited.";
  repaint();
}

// Handle button pressed.
public void mousePressed(MouseEvent me) {
  // save coordinates
  mouseX = me.getX();
  mouseY = me.getY();
  msg = "Down";
  repaint();
}

// Handle button released.
public void mouseReleased(MouseEvent me) {
  // save coordinates
  mouseX = me.getX();
  mouseY = me.getY();
  msg = "Up";
  repaint();
}

// Handle mouse dragged.
public void mouseDragged(MouseEvent me) {
  // save coordinates
  mouseX = me.getX();
  mouseY = me.getY();
  msg = "*";
  showStatus("Dragging mouse at " + mouseX + ", " + mouseY);
  repaint();
```

```
  }

  // Handle mouse moved.
  public void mouseMoved(MouseEvent me) {
    // show status
    showStatus("Moving mouse at " + me.getX() + ", " + me.getY());
  }

  // Display msg in applet window at current X,Y location.
  public void paint(Graphics g) {
    g.drawString(msg, mouseX, mouseY);
  }
}
```

这个例子的输出如下所示：



让我们来仔细看看这个例子。MouseEvents类扩展了Applet类，同时实现了MouseListener接口和MouseMotionListener接口。这两个接口包括了接受并处理各种鼠标事件的方法。请注意，在这里小应用程序不但是事件源，同时也是这些事件的监听器。这是因为支持addMouseListener( )方法和addMouseMotionListener( )方法的Component类是Applet的超类。所以小应用程序不但是事件源而且还是监听器。

在init()方法中，这个小应用程序注册它自己为鼠标事件的监听器。这些是通过调用addMouseListener( )方法和addMouseMotionListener( )方法来实现的，正如我们已经提到的，它们是类的成员方法，它们的原型如下所示：

```
synchronized void addMouseListener(MouseListener ml)
synchronized void addMouseMotionListener(MouseMotionListener mml)
```

在这里，ml是一个接受鼠标事件的对象的引用，而mml是一个接受鼠标运动事件的对象的引用。在这个程序里，它们是相同的一个对象。

接下来，这个小应用程序实现了在MouseListener接口和MouseMotionListener接口中定义的所有方法，以便对这些鼠标事件进行处理。每一个方法都处理了相应的事件，然后返回。

20.6.2 处理键盘事件

你可以采用与在前一章中鼠标事件范例相同的结构去处理键盘事件。当然，不同的是，

你必须实现相应的KeyListener接口。

在分析这个例子之前，让我们回顾一下键盘事件是如何产生的。当一个键被按下时，一个KEY_PRESSED事件被产生。这就使keyPressed( )这个事件处理方法被调用。当这个键被释放时，一个KEY_RELEASED事件产生，相应的事件处理方法keyReleased( )被执行。如果一个字符被按键产生，那么一个KEY_TYPED事件将被产生，并且事件处理方法keyTyped( )将被调用。因此，每次用户按下一个键时，通常有两个或三个事件被产生。如果你关心的只是字符，那么你可以忽略由按键和释放键所产生的信息。然而，如果你的程序需要处理特殊的键，比如方向键，那么你必须通过调用keyPressed( )这个事件处理方法来处理它们。

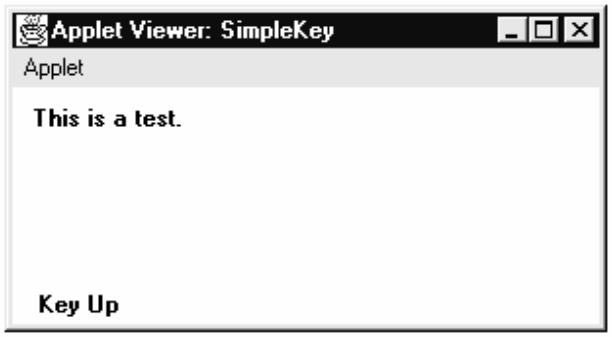
另外，在你的程序处理键盘事件之前，你的程序必须要获得输入焦点。通过调用requestFocus( )方法可以获得焦点，这个方法在Component类中被定义。如果你忽略了这一点，你的程序将不会获得任何键盘事件。

下面的程序演示了键盘输入的处理。它将回显按键到小应用程序窗口，并在窗口的状态栏上显示每一个按键被按下或释放的状态。

```
// Demonstrate the key event handlers.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="SimpleKey" width=300 height=100>
  </applet>
*/

public class SimpleKey extends Applet
  implements KeyListener {

  String msg = "";
  int X = 10, Y = 20; // output coordinates

  public void init() {
    addKeyListener(this);
    requestFocus(); // request input focus
  }

  public void keyPressed(KeyEvent ke) {
    showStatus("Key Down");
  }

  public void keyReleased(KeyEvent ke) {
    showStatus("Key Up");
  }

  public void keyTyped(KeyEvent ke) {
    msg += ke.getKeyChar();
    repaint();
  }

  // Display keystrokes.
```

```
  public void paint(Graphics g) {
    g.drawString(msg, X, Y);
  }
}
```

这个例子的输出如下所示：

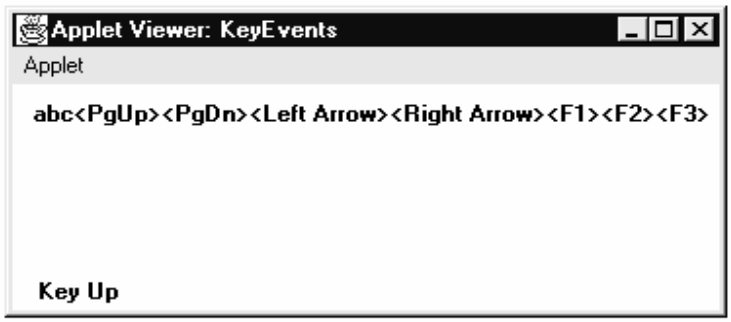


如果你想处理特殊的键，比如方向键，你需要在keyPressed( )这个事件处理方法中进行处理。它们不能通过 keyTyped( )这个事件处理方法来处理。为了表示这些键，你需要使用它们的虚拟键值。作为说明，接下来的这个小应用程序将输出一些特殊键的名字。

```
// Demonstrate some virtual key codes.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="KeyEvents" width=300 height=100>
  </applet>
*/

public class KeyEvents extends Applet
  implements KeyListener {

  String msg = "";
  int X = 10, Y = 20; // output coordinates

  public void init() {
    addKeyListener(this);
    requestFocus(); // request input focus
  }

  public void keyPressed(KeyEvent ke) {
    showStatus("Key Down");

    int key = ke.getKeyCode();
    switch(key) {
      case KeyEvent.VK_F1:
        msg += "<F1>";
        break;
      case KeyEvent.VK_F2:
        msg += "<F2>";
```

```
        break;
      case KeyEvent.VK_F3:
        msg += "<F3>";
        break;
      case KeyEvent.VK_PAGE_DOWN:
        msg += "<PgDn>";
        break;
      case KeyEvent.VK_PAGE_UP:
        msg += "<PgUp>";
        break;
      case KeyEvent.VK_LEFT:
        msg += "<Left Arrow>";
        break;
      case KeyEvent.VK_RIGHT:
        msg += "<Right Arrow>";
        break;
    }

    repaint();
  }

  public void keyReleased(KeyEvent ke) {
    showStatus("Key Up");
  }

  public void keyTyped(KeyEvent ke) {
    msg += ke.getKeyChar();
    repaint();
  }

  // Display keystrokes.
  public void paint(Graphics g) {
    g.drawString(msg, X, Y);
  }
}
```

这个例子的输出如下所示：



在键盘和鼠标事件处理的例子中展示的程序过程，可以被用于任何类型的事件处理，包括那些由控件产生的事件。在最后一章，你将看到许多处理其他类型事件的例子，但是它们采用的基本结构同前面所描述的例子程序是一样的。

20.7 Adapter类

Java提供了一个适配器类（adapter class），它可以使一些情况下的事件处理变得简单。一个适配器类实现并提供了一个事件监听器接口中所有的方法，但这些方法都是空方法。当你只想接受和处理特定的事件监听器接口对应的一部分事件时，适配器类将十分有用。你可以定义一个扩展了相应适配器类的新类来作为事件监听器，然后只实现那些你感兴趣的事件处理方法。

例如，MouseMotionAdapter类有两个方法：mouseDragged( )方法和mouseMoved( )方法。这些空方法的声明被定义在MouseMotionListener接口中。如果你只对鼠标的拖动事件感兴趣，那么你可以简单地继承MouseMotionAdapter类并实现mouseDragged( )方法。MouseMotionAdapter类中实现mouseMoved( )方法的空方法将替你处理鼠标运动事件。

在表20-4中列出了在java.awt.event包中定义的适配器类，并且注明了它们所实现的接口。

表 20-4 适配器类实现的监听器接口

适配器类	监听器接口
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseListener
WindowAdater	WindowListener

接下来的例子将采用一个适配器。当鼠标被点击或拖动时，相应的信息将被显示在小应用程序查看器和浏览器的状态栏上，而其他鼠标事件将被忽略掉。这个程序有三个类。AdapterDemo类扩展了Applet类。它的init( )方法产生了一个MyMouseAdapter类的实例，并且注册这个对象去接受鼠标事件通知。它也生成了一个类的实例，并且注册这个对象去接受鼠标运动事件通知。它们的构造函数都是以一个小应用程序的引用作为参数的。

MyMouseAdapter类实现了mouseClicked( )方法。其他鼠标事件在继承MouseAdapter类的代码中被忽略。

MyMouseMotionAdapter类实现了mouseDragged( )方法。别的鼠标运动事件在从MouseMotionAdapter类继承到的代码中被忽略。

请注意到我们的这两个事件监听器类都保存了一个小应用程序的引用。这个被作为参数传递给构造函数的信息在下面将被用来调用showStatus( )方法。

```
// Demonstrate an adapter.
import java.awt.*;
import java.awt.event.*;
```

```
import java.applet.*;
/*
  <applet code="AdapterDemo" width=300 height=100>
  </applet>
*/

public class AdapterDemo extends Applet {
  public void init() {
     addMouseListener(new MyMouseAdapter(this));
     addMouseMotionListener(new MyMouseMotionAdapter(this));
  }
}

class MyMouseAdapter extends MouseAdapter {
  AdapterDemo adapterDemo;
  public MyMouseAdapter(AdapterDemo adapterDemo) {
    this.adapterDemo = adapterDemo;
  }

  // Handle mouse clicked.
  public void mouseClicked(MouseEvent me) {
    adapterDemo.showStatus("Mouse clicked");
  }
}

class MyMouseMotionAdapter extends MouseMotionAdapter {
  AdapterDemo adapterDemo;
  public MyMouseMotionAdapter(AdapterDemo adapterDemo) {
    this.adapterDemo = adapterDemo;
  }

  // Handle mouse dragged.
  public void mouseDragged(MouseEvent me) {
    adapterDemo.showStatus("Mouse dragged");
  }
}
```

正如你看到的，我们不必去实现MouseMotionListener接口和MouseListener接口定义的全部方法，这就让我们省去了相当多的处理，也预防了由空方法带来的代码混乱。作为一个练习，你最好能用KeyAdapter类来重写一下前面键盘输入中的一个例子。

20.8　Inner 类

在第7章，我们解释了内部类的基础概念。在这里，你将看到它们为何很重要。我们知道一个内部类（inner class）是一个定义在其他类或者甚至是表达式中的类。在这一章我们将说明在使用事件适配器类时，如何通过使用内部类来简化代码。

为了理解使用内部类的好处，让我们看看如下所示列出的这个小应用程序。它没有使用一个内部类。它主要是在鼠标被按下时，在小应用程序查看器或浏览器的状态栏上显示“Mouse Pressed”这个字符串。在这个程序中有两个并列的类。MousePressedDemo类扩展

了Applet类，而MyMouseAdapter类扩展了MouseAdapter类。MousePressedDemo类的init( )方法产生了一个MyMouseAdapter类的实例，并且将这个对象作为参数提供给addMouseListener( )方法。

请注意，一个小应用程序的引用被作为参数提供给了MyMouseAdapter类的构造函数。这个引用被存储在这个实例的成员变量中，以便以后在mousePressed( )方法中使用。当鼠标被按下时，在mousePressed( )方法中通过被存储的小应用程序的引用，调用了小应用程序的showStatus( )方法。换句话说，showStatus( )方法被存储在MyMouseAdapter类中的小应用程序的引用调用了。

```
// This applet does NOT use an inner class.
import java.applet.*;
import java.awt.event.*;
/*
  <applet code="MousePressedDemo" width=200 height=100>
  </applet>
*/

public class MousePressedDemo extends Applet {
  public void init() {
    addMouseListener(new MyMouseAdapter(this));
  }
}

class MyMouseAdapter extends MouseAdapter {
  MousePressedDemo mousePressedDemo;
  public MyMouseAdapter(MousePressedDemo mousePressedDemo) {
    this.mousePressedDemo = mousePressedDemo;
  }
  public void mousePressed(MouseEvent me) {
    mousePressedDemo.showStatus("Mouse Pressed.");
  }
}
```

接下来，让我们看看如何通过使用内部类来改进前面的这个程序。在这个程序里，InnerClassDemo类是一个扩展了Applet类的最高类。MyMouseAdapter类是一个扩展了MouseAdapter类的内部类。由于MyMouseAdapter类被定义在InnerClassDemo类的范围之内，所以它可以访问这个类中的所有成员变量和方法。因此，mousePressed( )方法可以直接调用showStatus( )方法。这就不再需要通过存储一个小应用程序的引用来完成这些工作了。因此，也就不再需要为了调用这个对象，而给MyMouseAdapter( )方法传递一个小应用程序的引用。

```
// Inner class demo.
import java.applet.*;
import java.awt.event.*;
/*
  <applet code="InnerClassDemo" width=200 height=100>
  </applet>
*/

public class InnerClassDemo extends Applet {
```

```
  public void init() {
    addMouseListener(new MyMouseAdapter());
  }
  class MyMouseAdapter extends MouseAdapter {
    public void mousePressed(MouseEvent me) {
      showStatus("Mouse Pressed");
    }
  }
}
```

20.8.1　匿名内部类

一个匿名内部类（Anonymous inner class）是一个没有指定名称的类。在这一节中，我们将说明一个匿名内部类如何有利于处理事件程序的编写。让我们看一下下面的这个小应用程序。像以前一样，它还是在鼠标被按下时，在小应用程序查看器或浏览器的状态栏上显示“Mouse Pressed”这个字符串。

```
// Anonymous inner class demo.
import java.applet.*;
import java.awt.event.*;
/*
  <applet code="AnonymousInnerClassDemo" width=200 height=100>
  </applet>
*/

public class AnonymousInnerClassDemo extends Applet {
  public void init() {
    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent me) {
        showStatus("Mouse Pressed");
      }
    });
  }
}
```

在这个程序中只有一个最高类：AnonymousInnerClassDemo类。在init( )方法中调用了方法addMouseListener( )，它的参数是一个定义并产生一个匿名内部类的表达式。让我们来仔细分析一下这个表达式。

new MouseAdapter( ) { ... }的语法意义是告诉编译器在括号中的代码定义了一个匿名内部类。此外，这个类扩展了MouseAdapter类。这个新类没有名称，但是在这个表达式被执行时，自动实例化。

由于这个匿名内部类被定义在AnonymousInnerClassDemo类的范围之内，它可以访问这个类中的所有成员变量和方法。因此，也就可以直接的调用showStatus( )方法了。

像刚才所说明的那样，不论是有名称的内部类还是匿名内部类，都以一种简单而有效的方式解决了一个让人讨厌的问题。它们也让你的代码更加有效。

第 21 章　介绍 AWT：使用窗口、图形、文本

在第19章中，我们介绍了抽象窗口工具包（AWT），它对小应用程序提供了支持。在这一章中，我们将进一步介绍它。AWT包含了许多类和方法，通过它们我们可以生成和管理窗口。如果要对AWT包进行详尽的描述，恐怕要用整整一本书。因此，我们不可能详细描述AWT中的每一个方法、实例变量和类。然而，在这一章和接下来的两章，我们将介绍所有在你创建小应用程序或独立程序时有效地利用AWT所需要的知识。之后，你将可以独立的学习AWT的其他部分。

在本章中，你将学会如何创建和管理窗口、管理字体、输出文本以及使用图像。在第22章中，描述了各种各样AWT所支持的组件，比如滚动条和按钮。同时，还解释了Java事件处理机制的更多的方面。在第23章中，介绍了AWT中的图像子系统以及动画。

虽然，AWT的主要目的是支持小应用程序窗口，它也可以被用来创建独立运行在GUI环境中的窗口。由于大多数的例子采用了小应用程序的形式，所以为了运行它们，你需要用一个小应用程序查看器或者是兼容Java的浏览器。还有一小部分的例子示范了独立窗口程序的创建。

注意：如果你还没有阅读第20章，那么请先阅读。它讲述了事件处理的基本知识，而这一章中所用的许多例子都采用了事件处理。

21.1　AWT 类

AWT类被定义在java.awt包中。它是Java中最大的包之一。但由于采用从上到下分层方式组织，所以理解和使用起来都比较容易。表21-1列出了一些AWT的类。

表 21-1　一些 AWT 类

类（Class）	描述
AWTEvent	封装AWT事件
AWTEventMulticaster	分配事件到多个事件监听器
BorderLayout	边界布局管理器。边界布局使用了五个组件：North，South，East，West和Center
Button	产生一个下压式按钮控件
Canvas	一个空白，可以自由使用的窗口
CardLayout	卡片布局管理器。卡片布局仿效索引卡片。只有顶部的卡片可以看到
Checkbox	产生一个复选框
CheckboxGroup	产生一个复选框控件组
CheckboxMenuItem	产生一个开/关菜单项

续表

类（Class）	描述
Choice	生产一个弹出式列表
Color	用可移植的、跨平台的方式来管理颜色
Component	各种AWT组件的抽象的超类
Container	一个可以用来存放其他组件的组件类的子类
Cursor	封装一个位图光标
Dialog	产生一个对话框窗口
Dimension	确定一个对象的尺寸，宽度存放在变量width中，高度存放在变量height中
Event	封装事件
EventQueue	给事件排队
FileDialog	产生一个用于选择文件的窗口
FlowLayout	流动布局管理器。流动布局从左到右，从上到下的定位组件
Font	封装字体
FontMetrics	封装各种和字体有关的信息。这些信息有助你在窗口中显示文本
Frame	产生一个具有标题栏，调整大小的角以及一个菜单栏的标准窗口
Graphics	封装图形上下文。这个上下文被各种输出方法使用来在一个窗口中显示输出
GraphicsDevice	描述一个图形设备，比如一个屏幕和一个打印机
GraphicsEnvironment	描述各种Font和GraphicsDevice对象的集合
GridBagConstraints	定义各种与GridBagLayout类相关的常量
GridBagLayout	网格包布局管理器。网格包布局通过有GridBagConstraints提供的限制来显示组件
GridLayout	网格布局管理器。网格布局管理器用二维的网格来显示组件
Image	封装一个图形图像
Insets	封装一个容器的边框
Label	产生一个显示字符串的标签
List	产生一个用户可以选择的列表。与标准的窗口列表框相似
MediaTracker	管理媒体对象
Menu	产生一个下拉式菜单
MenuBar	产生一个菜单栏
MenuComponent	一个被各种菜单类所实现的抽象类
MenuItem	产生菜单项
MenuShortcut	封装与菜单项相应的快捷键
Panel	容器类的最简单的具体子类
Point	封装一个笛卡儿坐标对，分别存储在变量x和y中
Polygon	封装一个多边形
PopupMenu	产生一个弹出式菜单
PrintJob	代表一个打印机任务的抽象类

续表

类（Class）	描述
Rectangle	封装一个矩形
Robot	支持自动测试基于AWT的应用程序。（Java2,v1.3新增）
Scrollbars	产生一个滚动条控件
ScrollPane	为另一个组件提供水平和/或垂直滚动条的容器。
SystemColor	存放窗口，滚动条，文本以及其他GUI小部件的颜色
TextArea	生成多行编辑控件
TextComponent	TextArea 和 TextField的一个超类
TextField	生成一个单行编辑控件
Toolkit	由AWT实现的抽象类
Window	生成一个无框架，无菜单栏，无标题的窗口

虽然在Java 1.0之后，AWT的基本结构没有发生过变化，但是在Java 1.1发布后，已经不赞成使用一些旧的方法了，并且提供了新的方法代替它们。由于要向后兼容，Java 2仍然支持那些旧的方法。但是，由于这些方法已不适合在新的编码中使用，所以本书将不再讲述它们。

21.2 窗口基本原理

AWT根据类的层次定义窗口，并在每一层添加了特定的功能。在这些窗口中，用得最普遍的是在小应用程序派生于Panel类的窗口和派生于Frame类的独立窗口。这些窗口的功能大多数来自于它们的父类。因此，与Panel和Frame这两个类相关的类结构的描述是我们理解它们的基础。在图21-1中展示了Panel和Frame类的结构。现在让我们分别来看一下这些类。

21.2.1 组件（Component）

在AWT类层次结构的顶部是Component类。Component类是一个封装了一个可视组件的所有属性的抽象类。在屏幕上显示的所有用于用户交互的用户界面元素都是Component类的子类。这个类定义了一百多个用于事件管理的公共方法，这些事件包括鼠标或键盘的输入，窗口位置或大小的改变以及重绘窗口(在第19章和第20章中创建小应用程序时，你已经使用了其中的许多方法了)。一个Component对象可以保存当前的前景色、背景色以及被选择的文本的字体。

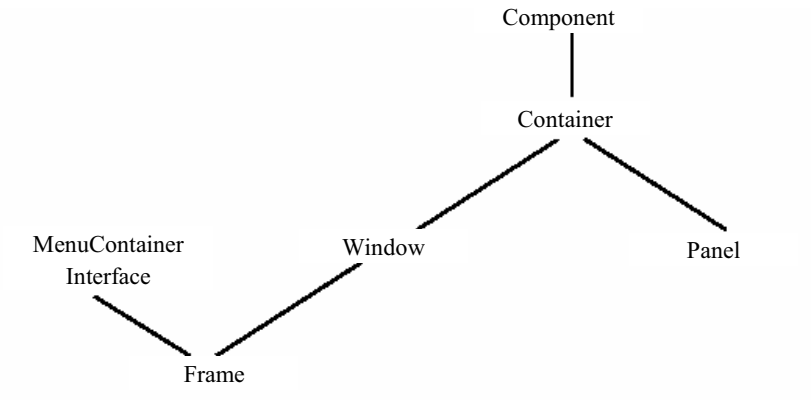


图 21-1 Panel 和 Frame 的类层次结构

21.2.2 容器（Container）

Container类是Component类的子类。这个类有一些附加方法，允许别的Component对象嵌套在Container类的对象中。当然，其他的Container对象可以被存放在一个Container对象中（因为它们也是Component类的实例）。这就形成了一个多层包容机制。容器主要负责布置它所包含的组件的位置。而它是通过使用一些设计管理器来完成这个功能的，你将在第22章中学习这些设计管理器。

21.2.3 面板（Panel）

Panel类是Container类的一个具体的子类。它没有添加任何新的方法；它只是简单的实现了Container类。一个Panel对象可以被看作是一个递归嵌套的具体的屏幕组件。Panel类是Applet类的子类。当屏幕输出直接传递给一个小应用程序时，它将在一个Panel对象的表面被画出。实际上，一个Panel对象是一个不包含标题栏、菜单栏以及边框的窗口。这就是为什么在浏览器中运行一个小应用程序时，你看不见标题栏、菜单栏以及边框的原因。而当你用小应用程序查看器来运行一个小应用程序时，小应用程序查看器提供了标题和边框。

其他的组件可以通过调用Panel类的add()方法被加入到一个Panel对象中，这个方法是从Container类继承来的。一旦这些组件被加入，那么你通常就可以通过调用在Component类中定义了的setLocation( ), setSize( )以及setBounds( )方法来改变这些组件的位置和大小。

21.2.4 窗口（Window）

窗口类产生一个顶级窗口（Window）。顶级窗口不包含在任何别的对象中，它直接出现在桌面上。通常，你将不会直接产生Window对象。相反，你将使用Window类的子类，这就是Frame类。

21.2.5 框架（Frame）

Frame类封装了窗口通常所需要的一切组件，它是Window类的子类，并且拥有标题栏、菜单栏、边框以及可以调整大小的角。如果你在一个小应用程序中创建了一个Frame对象，

它将包含一个例如“Java Applet Window”的警告消息给用户，表示一个小应用程序窗口已经被创建。这个消息警告用户，他们看见的窗口是由小应用程序启动的，而不是被运行在他们机器上的软件所启动（一个伪装基于主机的应用程序的小应用程序将可以用于在用户不知道的情况下获得密码和其他敏感信息）。当一个Frame窗口被程序而不是小应用程序创建时，就创建了一个通常的窗口。

21.2.6 画布（Canvas）

虽然画布不是小应用程序和frame窗口的层次结构的一部分，但是Canvas这种类型的窗口是很有用的。Canvas类封装了一个你可以用来绘制的空白窗口。你将在这本书的后面看到一个有关Canvas的例子。

21.3 用Frame窗口工作

在小应用程序之中，你最常创建的窗口来自于Frame类。你将用它在小应用程序中创建子窗口，在应用程序中创建顶级或子窗口。正如前面所提到的那样，它会生成一个标准样式的窗口。

Frame的构造函数如下所示：

```
Frame( )
Frame(String title)
```

第一种形式用于创建一个不含标题的标准窗口。第二种形式用于创建一个含有标题的窗口，这个标题是由title变量指定的。请注意你不能在创建时指定窗口的大小，你必须在窗口被创建后再设置窗口的大小。

这里有几个方法在你使用Frame窗口时将会用到。下面我们举例说明。

21.3.1 设置窗口大小

setSize( )这个方法用来设置窗口的大小，如下所示：

```
void setSize(int newWidth, int newHeight)
void setSize(Dimension newSize)
```

窗口的新的大小在变量newWidth和newHeight中被指定，或者在来自Dimension类的newSize对象的width和height这两个成员变量中被指定。这些大小使用像素为单位。

getSize( )这个方法被用来获得当前的窗口大小，如下所示：

```
Dimension getSize( )
```

这个方法返回一个Dimension对象，在这个对象的成员变量width和height中存放着当前窗口的大小。

21.3.2 隐藏和显示一个窗口

当一个frame窗口被创建以后，这个窗口默认是不可见的，除非你调用它的setVisible( )

方法。如下所示：

```
void setVisible(boolean visibleFlag)
```

如果这个方法的参数是true，那么调用它的组件是可见的。否则，就被隐藏。

21.3.3 设置窗口标题

你可以通过使用setTitle( )方法来改变一个frame窗口的标题。如下所示：

```
void setTitle(String newTitle)
```

在这里，参数newtitle是窗口的新标题。

21.3.4 关闭frame窗口

当使用一个frame窗口时，你的程序必须在它被关闭时通过调用setVisible(false)方法来将窗口从屏幕中除去。为了截获窗口关闭事件，你必须实现WindowListener监听器接口的windowClosing( )方法。在windowClosing( )方法中，你必须将窗口从屏幕中除去。在下一节中将用一个例子说明这种技术。

21.4 在小应用程序中创建一个frame窗口

简单地通过创建一个frame类的实例来创建一个窗口是可能的，但是你可能很少会这样做，因为对于这样的窗口你没有什么可以做的。例如，你将不能接受和处理在这个窗口中发生的事件或者不能简单的输出信息给它。大多数情况下，你将创建一个frame类的子类。这样做，你将会重载frame类的方法和事件处理。

在小应用程序中创建一个新的基于frame的窗口是很容易的。首先，创建一个frame类的子类。接下来，重载任何一个标准窗口方法，比如init( )方法，start( )方法，stop( )方法和paint( )方法。最后，实现windowListener这个监听器接口的windowClosing( )方法，在这个方法中，当窗口被关闭时，调用setVisible(false)方法将窗口从屏幕中除去。

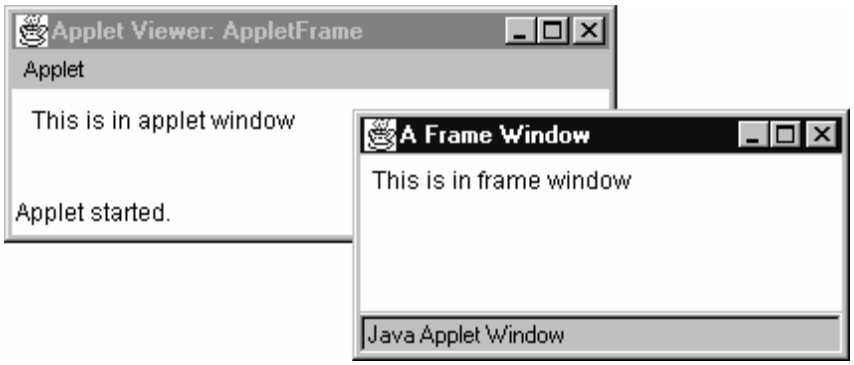
一旦你已经定义了一个frame类的子类，你就可以创建这个类的对象了。这就会产生一个基于frame的窗口，但是它在初始化中被设置为不可见的。你可以通过调用setVisible( )方法来使它可见。当这个窗口被创建后，它就有了一个默认的高度和宽度。你可以通过调用setSize( )方法来显式改变窗口的大小。

下面的小应用程序创建了一个叫做SampleFrame的Frame类的子类。这个子类的窗口在AppletFrame类的init()方法中实例化。请注意SampleFrame类调用了Frame类的构造函数。这将创建一个标准的frame窗口，它的标题由title参数决定。这个例子重载了小应用程序窗口的start()方法和stop()方法，所以在这两个方法中相应地显示和隐藏子窗口。这样当你在结束小应用程序时，关闭窗口，或者在浏览器中转移到另一页时，窗口都可以自动地被移去。而当你重新回到小应用程序时子窗口又被显示了出来。

```
// Create a child frame window from within an applet.
import java.awt.*;
```

```
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="AppletFrame" width=300 height=50>
  </applet>
*/

// Create a subclass of Frame.
class SampleFrame extends Frame {
  SampleFrame(String title) {
    super(title);
    // create an object to handle window events
    MyWindowAdapter adapter = new MyWindowAdapter(this);
     // register it to receive those events
    addWindowListener(adapter);
  }
  public void paint(Graphics g) {
    g.drawString("This is in frame window", 10, 40);
  }
}

class MyWindowAdapter extends WindowAdapter {
  SampleFrame sampleFrame;
  public MyWindowAdapter(SampleFrame sampleFrame) {
    this.sampleFrame = sampleFrame;
  }
  public void windowClosing(WindowEvent we) {
    sampleFrame.setVisible(false);
  }
}

// Create frame window.
public class AppletFrame extends Applet {
  Frame f;
  public void init() {
    f = new SampleFrame("A Frame Window");

    f.setSize(250, 250);
    f.setVisible(true);
  }
  public void start() {
    f.setVisible(true);
  }
  public void stop() {
    f.setVisible(false);
  }
  public void paint(Graphics g) {
    g.drawString("This is in applet window", 10, 20);
  }
}
```

这个例子的输出如下所示：



21.4.1 在Frame的窗口中处理事件

由于Frame类是Component类的子类，所以它继承了Component类的所有能力。这就意味着你可以像管理小应用程序主窗口一样地使用和管理frame窗口。例如，你可以重载paint()方法来显示输出，也可以在你需要恢复窗口时调用repaint()方法并且还可以重载所有的事件处理程序。无论何时，一个事件在窗口中发生时，由这个窗口定义的事件处理方法将被调用。每一个窗口都处理自己的事件。例如，接下来的程序创建了一个响应鼠标事件的窗口。小应用程序的主窗口中也响应鼠标事件。当你运行这个程序时，你将看到鼠标事件被发送给了那个产生该事件的窗口。

```
// Handle mouse events in both child and applet windows.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="WindowEvents" width=300 height=50>
  </applet>
*/

// Create a subclass of Frame.
class SampleFrame extends Frame
  implements MouseListener, MouseMotionListener {

  String msg = "";
  int mouseX=10, mouseY=40;
  int movX=0, movY=0;

  SampleFrame(String title) {
    super(title);
    // register this object to receive its own mouse events
    addMouseListener(this);
    addMouseMotionListener(this);
    // create an object to handle window events
    MyWindowAdapter adapter = new MyWindowAdapter(this);
    // register it to receive those events
    addWindowListener(adapter);
  }

  // Handle mouse clicked.
```

```
public void mouseClicked(MouseEvent me) {
}

// Handle mouse entered.
public void mouseEntered(MouseEvent evtObj) {
  // save coordinates
  mouseX = 10;
  mouseY = 54;
  msg = "Mouse just entered child.";
  repaint();
}

// Handle mouse exited.
public void mouseExited(MouseEvent evtObj) {
  // save coordinates
  mouseX = 10;
  mouseY = 54;
  msg = "Mouse just left child window.";
  repaint();
}

// Handle mouse pressed.
public void mousePressed(MouseEvent me) {
  // save coordinates
  mouseX = me.getX();
  mouseY = me.getY();
  msg = "Down";
  repaint();
}
// Handle mouse released.
public void mouseReleased(MouseEvent me) {
  // save coordinates
  mouseX = me.getX();
  mouseY = me.getY();
  msg = "Up";
  repaint();
}

// Handle mouse dragged.
public void mouseDragged(MouseEvent me) {
  // save coordinates
  mouseX = me.getX();
  mouseY = me.getY();
  movX = me.getX();
  movY = me.getY();
  msg = "*";
  repaint();
}

// Handle mouse moved.
public void mouseMoved(MouseEvent me) {
  // save coordinates
  movX = me.getX();
  movY = me.getY();
  repaint(0, 0, 100, 60);
```

```
  }

  public void paint(Graphics g) {
    g.drawString(msg, mouseX, mouseY);
    g.drawString("Mouse at " + movX + ", " + movY, 10, 40);
  }
}

class MyWindowAdapter extends WindowAdapter {
  SampleFrame sampleFrame;
  public MyWindowAdapter(SampleFrame sampleFrame) {
    this.sampleFrame = sampleFrame;
  }
  public void windowClosing(WindowEvent we) {
    sampleFrame.setVisible(false);
  }
}

// Applet window.
public class WindowEvents extends Applet
  implements MouseListener, MouseMotionListener {

  SampleFrame f;
  String msg = "";
  int mouseX=0, mouseY=10;
  int movX=0, movY=0;

  // Create a frame window.
  public void init() {
    f = new SampleFrame("Handle Mouse Events");
    f.setSize(300, 200);
    f.setVisible(true);

    // register this object to receive its own mouse events
    addMouseListener(this);
    addMouseMotionListener(this);
  }

  // Remove frame window when stopping applet.
  public void stop() {
    f.setVisible(false);
  }

  // Show frame window when starting applet.
  public void start() {
    f.setVisible(true);
  }

  // Handle mouse clicked.
  public void mouseClicked(MouseEvent me) {
  }

  // Handle mouse entered.
  public void mouseEntered(MouseEvent me) {
    // save coordinates
```

```
    mouseX = 0;
    mouseY = 24;
    msg = "Mouse just entered applet window.";
    repaint();
  }

  // Handle mouse exited.
  public void mouseExited(MouseEvent me) {
    // save coordinates
    mouseX = 0;
    mouseY = 24;
    msg = "Mouse just left applet window.";
    repaint();
  }

  // Handle button pressed.
  public void mousePressed(MouseEvent me) {
    // save coordinates
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "Down";
    repaint();
  }

  // Handle button released.
  public void mouseReleased(MouseEvent me) {
    // save coordinates
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "Up";
    repaint();
  }

  // Handle mouse dragged.
  public void mouseDragged(MouseEvent me) {
    // save coordinates
    mouseX = me.getX();
    mouseY = me.getY();
    movX = me.getX();
    movY = me.getY();
    msg = "*";
    repaint();
  }

  // Handle mouse moved.
  public void mouseMoved(MouseEvent me) {
    // save coordinates
    movX = me.getX();
    movY = me.getY();
    repaint(0, 0, 100, 20);
  }

  // Display msg in applet window.
  public void paint(Graphics g) {
    g.drawString(msg, mouseX, mouseY);
```

```
    g.drawString("Mouse at " + movX + ", " + movY, 0, 10);
  }
}
```

这个程序的输出如下所示：

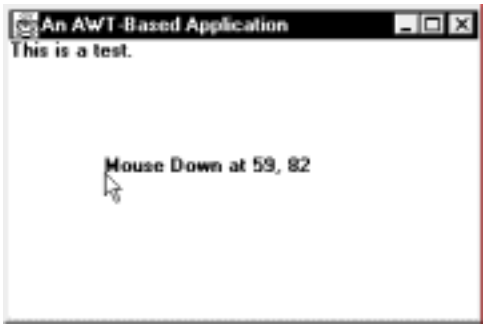
21.5 创建一个基于窗口的程序

虽然Java的AWT常用来创建小应用程序，但是它也一样可以被用来创建独立的基于AWT的应用程序。这可以通过在main()方法中简单的创建一个你需要的窗口实例或窗口来实现。例如，接下来的程序创建了frame窗口，它可以响应鼠标点击和键盘击键。

```
// Create an AWT-based application.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

// Create a frame window.
public class AppWindow extends Frame {
  String keymsg = "";
  String mousemsg = "";
  int mouseX=30, mouseY=30;

  public AppWindow() {
    addKeyListener(new MyKeyAdapter(this));
    addMouseListener(new MyMouseAdapter(this));
    addWindowListener(new MyWindowAdapter());
  }

  public void paint(Graphics g) {
    g.drawString(keymsg, 10, 40);
    g.drawString(mousemsg, mouseX, mouseY);
  }

  // Create the window.
```

```
  public static void main(String args[]) {
    AppWindow appwin = new AppWindow();

    appwin.setSize(new Dimension(300, 200));
    appwin.setTitle("An AWT-Based Application");
    appwin.setVisible(true);
  }
}

class MyKeyAdapter extends KeyAdapter {
  AppWindow appWindow;
  public MyKeyAdapter(AppWindow appWindow) {
    this.appWindow = appWindow;
  }
  public void keyTyped(KeyEvent ke) {
    appWindow.keymsg += ke.getKeyChar();
    appWindow.repaint();
  };
}

class MyMouseAdapter extends MouseAdapter {
  AppWindow appWindow;
  public MyMouseAdapter(AppWindow appWindow) {
    this.appWindow = appWindow;
  }
  public void mousePressed(MouseEvent me) {
    appWindow.mouseX = me.getX();
    appWindow.mouseY = me.getY();
    appWindow.mousemsg = "Mouse Down at " + appWindow.mouseX +
                         ", " + appWindow.mouseY;
    appWindow.repaint();
  }
}

class MyWindowAdapter extends WindowAdapter {
  public void windowClosing(WindowEvent we) {
    System.exit(0);
  }
}
```

这个程序的输出如下所示：



一旦被创建，frame窗口就有了自己的生命特征。请注意main()方法结束时调用了

appwin.setVisible(true)方法。所以，无论怎样，在关闭窗口之前这个程序将一直保持运行状态。事实上，当创建一个基于窗口的应用程序时，你将用main()方法来启动它的顶级窗口。在此之后，你的程序将作为一个基于GUI的应用程序运行，而不是开始时的基于控制台的应用程序。

21.6 在窗口中显示信息

在大多数情况下，一个窗口是一个信息的容器。虽然我们在前面的例子中已经输出了一些文本到窗口中，但是我们还没有开始利用一个窗口的优势去显示高质量的文本和图像。实际上，AWT的许多功能来自于对它们的支持。在本章剩余部分，我们将讨论Java的文字、图形和字体处理能力。正如你将看到的那样，它们不但功能强大而且很灵活。

21.7 使 用 图 形

AWT支持图形方法。所有的图形被画到相关联的窗口中，而这个相关联的窗口可能是一个小应用程序的主窗口，也可能是一个小应用程序的子窗口，或者是一个独立应用程序的窗口。每一个窗口的原点都位于窗口的左上角，以像素为单位坐标为(0,0)的点。在这个窗口里，所有的输出都是通过一个图形上下文（graphics context）来产生的。图形上下文是由Graphics类封装的，它可以通过两种方法获得：

- 当某一方法如paint( )和update( )被调用时，它被传递给小应用程序。
- 它可以由Component类的getGraphics( )方法返回。

在本章剩下的例子中，我们将在一个小应用程序的主窗口中示范图形。不过，请注意，这些技术也可以被用到别的窗口中。

Graphics类定义了一些绘图函数。每一个图形都可以只画边框或者被填充。这些对象用当前选择的颜色来绘制和填充，黑色是默认的颜色。当一个被绘制图形对象的尺寸超过了窗口的大小时，超出的那部分输出将会自动被剪去。让我们来看几个绘图方法吧。

21.7.1 画线

通过drawLine( )方法我们可以画线，语法如下所示：

```
void drawLine(int startX, int startY, int endX, int endY)
```

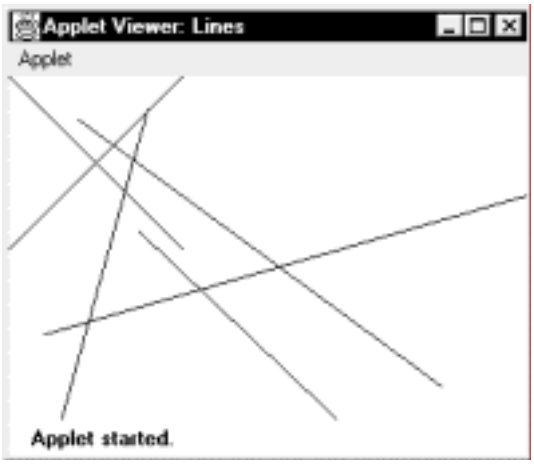
drawLine( )方法将用当前的颜色以(startX,startY)为起点，(endX,endY)为终点画一条直线。

接下来的这个小应用程序演示了如何画线。

```
// Draw lines
import java.awt.*;
import java.applet.*;
/*
```

```
<applet code="Lines" width=300 height=200>
</applet>
*/
public class Lines extends Applet {
  public void paint(Graphics g) {
    g.drawLine(0, 0, 100, 100);
    g.drawLine(0, 100, 100, 0);
    g.drawLine(40, 25, 250, 180);
    g.drawLine(75, 90, 400, 400);
    g.drawLine(20, 150, 400, 40);
    g.drawLine(5, 290, 80, 19);
  }
}
```

这个例子的输出如下所示：



21.7.2 画矩形

drawRect( )方法和fillRect( )方法分别可以用来绘制一个矩形的轮廓和一个被填充的矩形。语法如下所示：

```
void drawRect(int top, int left, int width, int height)
void fillRect(int top, int left, int width, int height)
```

矩形的左上角在（top,left），矩形的大小由参数width和heigh来确定。.

为了绘制一个圆角矩形，可以用drawRoundRect( )方法或者fillRoundRect( )方法，语法如下所示：

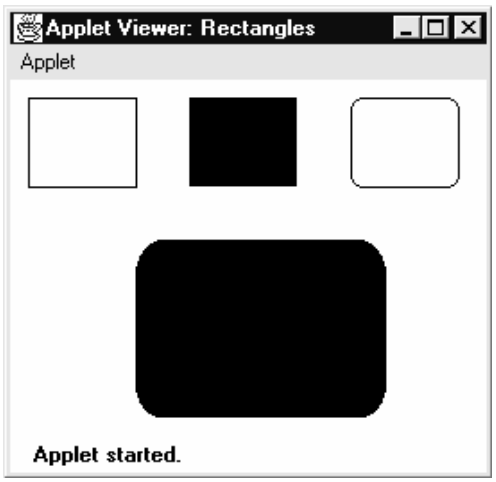
```
void drawRoundRect(int top, int left, int width, int height,
                                        int xDiam, int yDiam)
void fillRoundRect(int top, int left, int width, int height,
                                    int xDiam, int yDiam)
```

一个圆角矩形的角是圆的。这个矩形的左上角是在(top,left)。这个矩形的大小有参数width和height来确定。X方向圆弧的直径由参数xDiam确定。Y方向圆弧的直径由参数yDiam确定。

下面的例子演示了如何画这些矩形。

```
// Draw rectangles
import java.awt.*;
import java.applet.*;
/*
<applet code="Rectangles" width=300 height=200>
</applet>
*/

public class Rectangles extends Applet {
  public void paint(Graphics g) {
    g.drawRect(10, 10, 60, 50);
    g.fillRect(100, 10, 60, 50);
    g.drawRoundRect(190, 10, 60, 50, 15, 15);
    g.fillRoundRect(70, 90, 140, 100, 30, 40);
  }
}
```

程序输出如下所示：



21.7.3 绘制椭圆和圆

用drawOval( )方法可以绘制一个椭圆。而用fillOval( )方法可以填充一个椭圆。这些方法的语法如下所示：

```
void drawOval(int top, int left, int width, int height)
void fillOval(int top, int left, int width, int height)
```

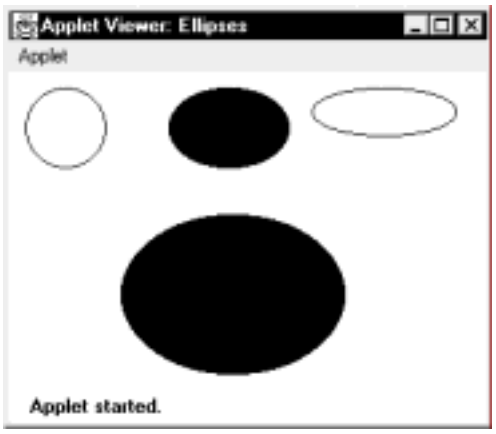
椭圆被绘制在一个矩形范围内，这个矩形的左上角是(top,left)，而大小由参数width和height确定。绘制圆形时，我们只需指定矩形为一个正方形。

接下来的例子演示了如何绘制椭圆。

```
// Draw Ellipses
import java.awt.*;
```

```
import java.applet.*;
/*
<applet code="Ellipses" width=300 height=200>
</applet>
*/

public class Ellipses extends Applet {
  public void paint(Graphics g) {
    g.drawOval(10, 10, 50, 50);
    g.fillOval(100, 10, 75, 50);
    g.drawOval(190, 10, 90, 30);
    g.fillOval(70, 90, 140, 100);
  }
}
```

这个例子的输出如下所示：



21.7.4 画圆弧

通过drawArc( )方法和fillArc( )方法我们可以绘制圆弧。它们的原型如下所示：

```
void drawArc(int top, int left, int width, int height, int startAngle,
                          int sweepAngle)
void fillArc(int top, int left, int width, int height, int startAngle,
                     int sweepAngle)
```

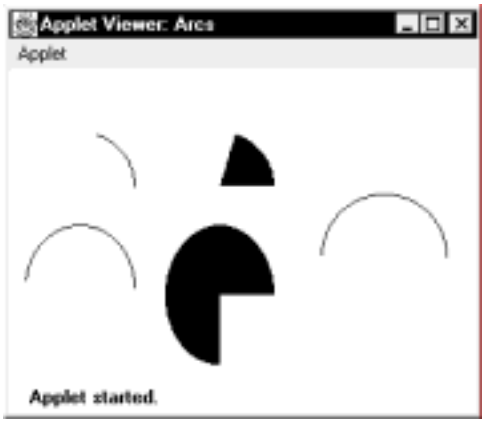
圆弧被绘制在一个矩形范围内，这个矩形的左上角是(top,left)点，而大小由参数width和height确定。圆弧是以startAngle为开始的角度，sweepAngle为转过的角度而绘制的。这些角是以度为单位的。0度指水平方向上，类似钟表上三点钟的时针位置。如果参数sweepAngle是正的，圆弧将被逆时针绘制，否则将被顺时针绘制。因此，为了画出一个从12点到6点的圆弧，我们应该设置开始的角度为90°而转过的角度为180°。

接下来的例子演示了如何绘制圆弧。

```
// Draw Arcs
import java.awt.*;
import java.applet.*;
/*
```

```
<applet code="Arcs" width=300 height=200>
</applet>
*/

public class Arcs extends Applet {
  public void paint(Graphics g) {
    g.drawArc(10, 40, 70, 70, 0, 75);
    g.fillArc(100, 40, 70, 70, 0, 75);
    g.drawArc(10, 100, 70, 80, 0, 175);
    g.fillArc(100, 100, 70, 90, 0, 270);
    g.drawArc(200, 80, 80, 80, 0, 180);
  }
}
```

这个例子的输出如下所示：



21.7.5 绘制多边形

通过使用drawPolygon( )方法和fillPolygon( )方法，我们可以绘制出任意的形状，这些方法的语法如下所示：

```
void drawPolygon(int x[ ], int y[ ], int numPoints)
void fillPolygon(int x[ ], int y[ ], int numPoints)
```

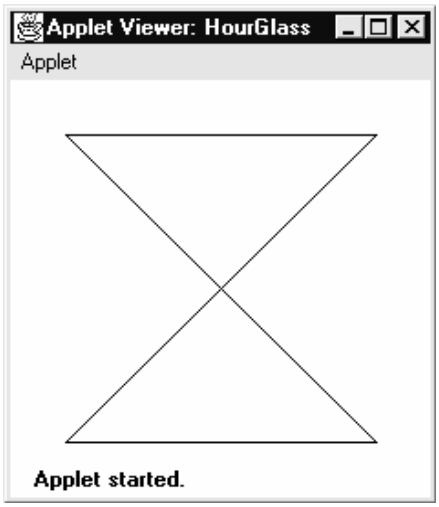
多边形的顶点是由数组x和数组y中相对应的数字组成的坐标来指定的。而数组x，y中定义的点的个数是由参数numPoints确定的。这些方法的另一种形式是通过ploygon对象来指定多边形。

接下来的例子演示了如何绘制一个沙漏的形状。

```
// Draw Polygon
import java.awt.*;
import java.applet.*;
/*
<applet code="HourGlass" width=230 height=210>
</applet>
*/

public class HourGlass extends Applet {
  public void paint(Graphics g) {
```

```
    int xpoints[] = {30, 200, 30, 200, 30};
    int ypoints[] = {30, 30, 200, 200, 30};
    int num = 5;

    g.drawPolygon(xpoints, ypoints, num);
  }
}
```

这个例子的输出如下所示：



21.7.6 改变图像的大小

我们经常想去改变一个图形对象，以便它的大小与它所在的窗口的大小匹配。为此，我们首先要通过窗口对象的getSize( )方法获得当前窗口的尺寸。这个方法返回窗口的尺寸并封装在Dimension对象中。一旦你获得当前窗口的尺寸，便可以相应的调整你的图形输出的大小。

为了演示这个技术，我们让一个小应用程序开始时在200×200的像素范围内，并且每用鼠标点击一次，小应用程序的范围的长和宽分别会增加25个像素点，直到小应用程序的范围大于500×500的像素范围。将不断接下来的点击又使小应用程序的范围不断减小，直至回到了开始时200×200的像素范围，这一过程将不断重复。在小应用程序的窗口中，沿着窗口的内边框绘制了一个矩形，在矩形的内部还绘制了一个X，填满整个窗口。这个小应用程序要在appletviewer中才能工作，在浏览器中它不运行。

```
// Resizing output to fit the current size of a window.
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/*
  <applet code="ResizeMe" width=200 height=200>
  </applet>
 */

public class ResizeMe extends Applet {
```

```
  final int inc = 25;
  int max = 500;
  int min = 200;
  Dimension d;

  public ResizeMe() {
    addMouseListener(new MouseAdapter() {
      public void mouseReleased(MouseEvent me) {
        int w = (d.width + inc) > max?min :(d.width + inc);
        int h = (d.height + inc) > max?min :(d.height + inc);
        setSize(new Dimension(w, h));
      }
    });
  }
  public void paint(Graphics g) {
    d = getSize();

    g.drawLine(0, 0, d.width-1, d.height-1);
    g.drawLine(0, d.height-1, d.width-1, 0);
    g.drawRect(0, 0, d.width-1, d.height-1);
  }
}
```

21.8 使 用 颜 色

Java以一种方便、设备无关的方式支持颜色。AWT的颜色系统允许你指定任何你需要的颜色。在执行程序或小应用程序时，虽然受到了显示硬件的限制，但是它将找到与之最相近的颜色。因此，编写代码时无需关心由于不同的硬件设备所支持的方式不同而引起的颜色差别。颜色被封装在Color类中。

正如你在第19章中所看到的，Color类定义了一些常量(比如，Color.black)来指定一些常用的颜色。你也可以通过Color类的构造函数来生产你自己的颜色。这些方法的形式如下所示：

```
Color(int red, int green, int blue)
Color(int rgbValue)
Color(float red, float green, float blue)
```

第一个构造函数使用了三个分别代表红、绿、蓝的整数来表示它们混合的颜色。这些值像下面这个例子中的一样，必须在0~255之间。

```
new Color(255, 100, 100); // light red.
```

第二个构造函数采用一个由红、绿、蓝按照一定的格式压缩成的整数来表示颜色。这个整数的0-7位代表蓝，8-15位代表绿，16-23位代表红。这里是一个使用这个构造函数的例子：

```
int newRed = (0xff000000 | (0xc0 << 16) | (0x00 << 8) | 0x00);
Color darkRed = new Color(newRed);
```

最后一个构造函数Color(float, float, float)，用了三个浮点数指定红、绿、蓝的相对混合。

一旦你已经生成了一个颜色，你将可以通过调用在19章中描述过的setForeground( )方法和setBackground( )方法使它作为前景色或者背景色。你也可以用该颜色来作为当前的绘图颜色。

21.8.1 有关颜色的方法

Color类定义了几种使用颜色的方法。下面我们将讨论它们。

使用色相、饱和度、亮度

HSB（hue-saturation-brightness）颜色模型是除了RGB模型外的另一种可以用来指定特定颜色的方式。其中，色相好比是颜色的轮子，是由一个0.0~1.0之间的数来确定的（颜色大约有：红色，橙色，黄色，绿色，蓝色，靛青，紫色）。饱和度是另一个介于0.0~1.0之间的值，它代表了相应的色相的深浅或鲜艳程度。亮度也是一个介于0.0~1.0之间的值，当为1时表示明亮，而0表示黑暗。Color类支持两个方法，它们可以让你在RGB和HSB之间进行转换。它们的原型如下所示：

```
static int HSBtoRGB(float hue, float saturation, float brightness)
static float[ ] RGBtoHSB(int red, int green, int blue, float values[ ])
```

HSBtoRGB( )方法返回了一个与构造函数Color(int)兼容的被压缩的RGB值。

RGBtoHSB( )方法返回了一个与RGB值相应的HSB值的浮点数组。如果values不是null，那么这个数组返回的是HSB值；否则，产生一个新的数组并且在里面放着HSB的值。在数组中，色相的下标为0，饱和度的下标为1，亮度的下标为2。

getRed( ), getGreen( ), getBlue( )

你可以通过调用getRed( )方法、getGreen( )方法和getBlue( )方法来获得一个颜色中包含的红、绿、蓝的成份。语法如下所示：

```
int getRed( )
int getGreen( )
int getBlue( )
```

这些方法中的每一个都返回在低8位整数中Color对象的RGB颜色中的相应的成份。

getRGB( )

为了获得一个颜色的RGB值，我们可以调用getRGB( )方法，格式如下所示：

```
int getRGB( )
```

这个方法的返回值和前面所描述的一样。

21.8.2 设置当前图形颜色

默认情况下，图形对象是用当前的前景色来绘制的。你可以改变这个颜色，这是通过调用Graphics类的setColor( )方法来实现的：

```
void setColor(Color newColor)
```

这里，newcolor指定了新的绘图颜色。

你也可以通过调用getColor( )方法来获得当前的颜色，格式如下所示：

```
Color getColor( )
```

21.8.3 一个有关颜色的例子

接下来的这个小应用程序创建了几种颜色，并且用这些颜色绘制几个图形对象。

```
// Demonstrate color.
import java.awt.*;
import java.applet.*;
/*
<applet code="ColorDemo" width=300 height=200>
</applet>
*/

public class ColorDemo extends Applet {
  // draw lines
  public void paint(Graphics g) {
    Color c1 = new Color(255, 100, 100);
    Color c2 = new Color(100, 255, 100);
    Color c3 = new Color(100, 100, 255);

    g.setColor(c1);
    g.drawLine(0, 0, 100, 100);
    g.drawLine(0, 100, 100, 0);

    g.setColor(c2);
    g.drawLine(40, 25, 250, 180);
    g.drawLine(75, 90, 400, 400);

    g.setColor(c3);
    g.drawLine(20, 150, 400, 40);
    g.drawLine(5, 290, 80, 19);

    g.setColor(Color.red);
    g.drawOval(10, 10, 50, 50);
    g.fillOval(70, 90, 140, 100);

    g.setColor(Color.blue);
    g.drawOval(190, 10, 90, 30);
    g.drawRect(10, 10, 60, 50);

    g.setColor(Color.cyan);
    g.fillRect(100, 10, 60, 50);
    g.drawRoundRect(190, 10, 60, 50, 15, 15);
  }
}
```

21.9 设置绘图模式

绘图模式（paint mode）决定了对象是如何被画在窗口中的。默认情况下，对一个窗口的新的输出将覆盖该窗口中任何已经存在的内容。然而，通过调用setXORMode( )方法设置绘图模式，我们可以使一个新的对象以异或操作的方式加入到窗口中，语法如下所示：

```
void setXORMode(Color xorColor)
```

在这里，xorColor指定的是绘制对象时与窗口进行异或操作的颜色。异或模式的优点是新的对象总是可以保证被看见，无论这个对象是用什么颜色画的。

如果你想回到覆盖模式，那么你可以调用setPaintMode( )方法，语法如下所示：

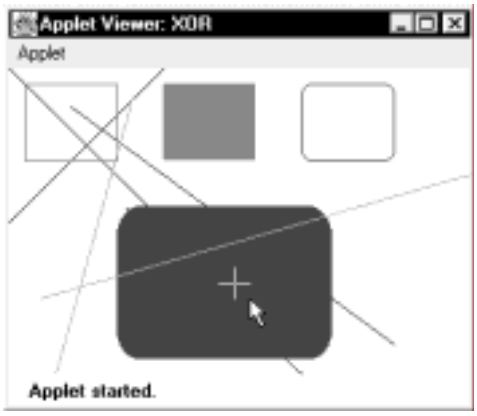
```
void setPaintMode( )
```

通常，你将希望对一般的输出用覆盖模式，而在特定的情况下采用异或模式。例如，接下来的程序中显示了一个跟踪鼠标指针的十字。这个十字被异或到窗口中，所以无论指针下面是什么颜色它都总是可见的。

```
// Demonstrate XOR mode.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="XOR" width=400 height=200>
  </applet>
*/

public class XOR extends Applet {
  int chsX=100, chsY=100;

  public XOR() {
    addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseMoved(MouseEvent me) {
        int x = me.getX();
        int y = me.getY();
        chsX = x-10;
        chsY = y-10;
        repaint();
      }
    });
  }

  public void paint(Graphics g) {
    g.drawLine(0, 0, 100, 100);
    g.drawLine(0, 100, 100, 0);
    g.setColor(Color.blue);
    g.drawLine(40, 25, 250, 180);
    g.drawLine(75, 90, 400, 400);
    g.setColor(Color.green);
    g.drawRect(10, 10, 60, 50);
```

```
    g.fillRect(100, 10, 60, 50);
    g.setColor(Color.red);
    g.drawRoundRect(190, 10, 60, 50, 15, 15);
    g.fillRoundRect(70, 90, 140, 100, 30, 40);
    g.setColor(Color.cyan);
    g.drawLine(20, 150, 400, 40);
    g.drawLine(5, 290, 80, 19);

    // xor cross hairs
    g.setXORMode(Color.black);
    g.drawLine(chsX-10, chsY, chsX+10, chsY);
    g.drawLine(chsX, chsY-10, chsX, chsY+10);
    g.setPaintMode();
  }
}
```

这个例子的输出如下所示：



21.10 使 用 字 体

AWT支持多种字体。字体已从传统的排版领域发展为产生计算机文档和显示的重要部分。AWT通过提炼字体处理提供了灵活的操作，并且允许字体的动态选择。

从Java 2开始，字体有了一个姓名、一个逻辑字体名和一个外形名。姓名（family name）是字体最通常的名字，比如courier。逻辑名（logical name）指定了一类字体，比如monospaced。而外形名（face name）指定了一个特定的字体，比如courier italic。

字体被封装在font类中。在表21-2中，列出了一些font类定义的方法。

表 21-2 一些 font 定义的方法

方法	描述
static Font decode(String str)	返回给定名称的字体
boolean equals(Object FontObj)	如果调用的对象包含了由FontObj指定的字体，返回true；否则返回false
String getFamily( )	返回调用的字体所属于的字体家族的名字

续表

方法	描述
static Font getFont(String property)	返回由property指定的系统属性相应的字体。如果property不存在，返回null
static Font getFont(String property, Font defaultFont)	返回由property指定的系统属性相应的字体。如果property不存在，返回由defaultFont指定的字体
String getFontName()	返回调用字体的外形名(Java2增加)
String getName()	返回调用字体的逻辑名
int getSize()	返回调用字体的大小（以点为单位）
int getStyle()	返回调用字体的类型值
int hashCode( )	返回与调用对象相关的散列值。
boolean isBold( )	如果字体包含粗体类型（BOLD）则返回true；否则返回false
boolean isItalic( )	如果字体包含斜体类型（ITALIC）则返回true；否则返回false
boolean isPlain( )	如果字体包含无格式类型（PLAIN）则返回true；否则返回false
String toString( )	返回与调用字体相应的字符串

Font类定义了以下变量：

变量	意义
String name	字体的名字
float pointSize	字体的尺寸（以磅计）
int size	字体的尺寸（以磅计）
int style	字体的样式

21.10.1 决定可用的字体

当使用字体时，你经常需要知道哪一种字体在你的机器上是可用的。为了获得这些信息，你可以调用在GraphicsEnvironment类中定义的getAvailableFontFamilyNames( )方法。如下所示：

```
String[ ] getAvailableFontFamilyNames( )
```

这个方法返回了一个字符串数组，它包含了可以使用的字体集合的名字。

除此之外，在GraphicsEnvironment类中定义的getAllFonts( )方法也可以完成这个功能。语法如下所示：

```
Font[ ] getAllFonts( )
```

这个方法返回一个字体对象的数组，这个数组中包含了所有可以使用的字体。

因为这些方法都是类的成员方法，所以你需要用一个GraphicsEnvironment类的引用去调用它们。你可以通过调用getLocalGraphicsEnvironment( )这个静态方法来获得一个GraphicsEnvironment类的引用，这个方法被定义在GraphicsEnvironment类中，如下所示：

```
static GraphicsEnvironment getLocalGraphicsEnvironment( )
```

下面的这个小应用程序演示了如何获得可用的字体集合的名字。

```
// Display Fonts
/*
<applet code="ShowFonts" width=550 height=60>
</applet>
*/
import java.applet.*;
import java.awt.*;

public class ShowFonts extends Applet {
  public void paint(Graphics g) {
    String msg = "";
    String FontList[];

    GraphicsEnvironment ge =
      GraphicsEnvironment.getLocalGraphicsEnvironment();
    FontList = ge.getAvailableFontFamilyNames();
    for(int i = 0; i < FontList.length; i++)
      msg += FontList[i] + " ";

    g.drawString(msg, 4, 16);
  }
}
```

这个例子的输出如下图所示。然而，当你运行这个程序时，你可能看到不同于图中显示的字体的列表。

注意：在Java 2之前，需要使用在Toolkit类中定义的getFontList( )方法来获得字体的列表。现在已经反对再使用这个方法，所以不应该在新的程序中使用它。

21.10.2 创建和选择一种字体

为了选择一个新的字体，你必须首先创建一个描述了字体的Font对象。Font类的构造函数的一种形式如下所示：

```
Font(String fontName, int fontStyle, int pointSize)
```

在这里，参数fontname指定了希望使用的字体的名字。这个名字既可以是逻辑名也可以是外形名。所有的Java环境都将支持如下的字体：Dialog, DialogInput, Sans Serif, Serif, Monospaced, 以及 Symbol。Dialog是你的系统对话框使用的字体。如果你不明确的指定字体，Dialog也是默认的字体。你也可以用任何其他被你的特殊环境支持的字体，但是请小心，这些字体未必都是可用的。

字体的样式是由参数fontstyle指定的。它可以由Font.PLAIN， Font.BOLD以及

Font.ITALIC这三个常量中的一个和多个组成。为了结合各种样式，可以把它们用or连接在一起。例如，Font.BOLD | Font.ITALIC指定了一个由粗体和斜体组成的样式。

以磅为单位的字体，大小由pointsize指定。

为了使用你创建的字体，你必须通过调用setFont( )方法来选择该字体，这个方法被定义在Component类中，如下所示：

```
void setFont(Font fontObj)
```

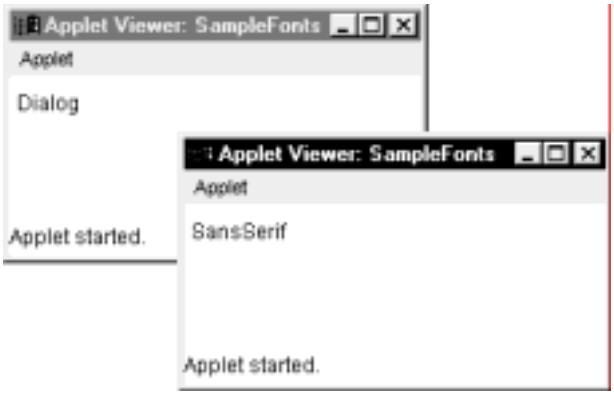
在这里，fontObj是包含了你希望使用的字体的对象。

下面的这个程序输出了每一个标准字体的例子。每一次你在窗口中点击鼠标时，一个新的字体将被选择，同时它的名字将被显示出来。

```
// Show fonts.
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/*
  <applet code="SampleFonts" width=200 height=100>
  </applet>
*/

public class SampleFonts extends Applet {
  int next = 0;
  Font f;
  String msg;
  public void init() {
    f = new Font("Dialog", Font.PLAIN, 12);
    msg = "Dialog";
    setFont(f);
    addMouseListener(new MyMouseAdapter(this));
  }

  public void paint(Graphics g) {
    g.drawString(msg, 4, 20);
  }
}

class MyMouseAdapter extends MouseAdapter {
  SampleFonts sampleFonts;
  public MyMouseAdapter(SampleFonts sampleFonts) {
    this.sampleFonts = sampleFonts;
  }
  public void mousePressed(MouseEvent me) {
    // Switch fonts with each mouse click.
    sampleFonts.next++;
    switch(sampleFonts.next) {
    case 0:
      sampleFonts.f = new Font("Dialog", Font.PLAIN, 12);
      sampleFonts.msg = "Dialog";
      break;
    case 1:
      sampleFonts.f = new Font("DialogInput", Font.PLAIN, 12);
```

```
        sampleFonts.msg = "DialogInput";
        break;
      case 2:
        sampleFonts.f = new Font("SansSerif", Font.PLAIN, 12);
        sampleFonts.msg = "SansSerif";
        break;
      case 3:
        sampleFonts.f = new Font("Serif", Font.PLAIN, 12);
        sampleFonts.msg = "Serif";
        break;
      case 4:
        sampleFonts.f = new Font("Monospaced", Font.PLAIN, 12);
        sampleFonts.msg = "Monospaced";
        break;
      }
      if(sampleFonts.next == 4) sampleFonts.next = -1;
      sampleFonts.setFont(sampleFonts.f);
      sampleFonts.repaint();
    }
  }
```

这个例子的输出如下图所示：



21.10.3 获取字体信息

如果你想去获得当前被选择的字体的有关信息，必须首先通过调用getFont( )方法来获得当前的字体，这个方法被定义在Graphics类中，如下所示：

```
Font getFont( )
```

一旦你获知了当前被选择的字体是哪一种字体，那么你就可以通过调用Font类定义的各种方法来获得它的有关信息了。例如，下面的这个小应用程序显示当前被选择的字体的名字、所属集合、大小以及样式。

```
// Display font info.
import java.applet.*;
import java.awt.*;
/*
<applet code="FontInfo" width=350 height=60>
```

```
</applet>
*/

public class FontInfo extends Applet {
  public void paint(Graphics g) {
    Font f = g.getFont();
    String fontName = f.getName();
    String fontFamily = f.getFamily();
    int fontSize = f.getSize();
    int fontStyle = f.getStyle();

    String msg = "Family: " + fontName;
    msg += ", Font: " + fontFamily;
    msg += ", Size: " + fontSize + ", Style: ";
    if((fontStyle & Font.BOLD) == Font.BOLD)
      msg += "Bold ";
    if((fontStyle & Font.ITALIC) == Font.ITALIC)
      msg += "Italic ";
    if((fontStyle & Font.PLAIN) == Font.PLAIN)
      msg += "Plain ";

    g.drawString(msg, 4, 16);
  }
}
```

21.11 通过FontMetrics来管理文本输出

正如我们所期望的一样，Java支持许多字体。对于大部分字体，字符并不具有完全相同的尺寸，许多字体的尺寸是可以调整的。每一个字符的高、字母（descenders）的长度（字母的悬挂部分，例如y）以及行与行之间间距对与每一种字体都是不同的。此外，字体的磅大小也可以被改变。那些可变的属性不是很重要，除非Java要求程序员用手工管理所有的文本输出。

每一种字体的尺寸可能不同，并且在你的程序运行时，字体可能会改变，所以必须有一些方式来决定字体尺寸和当前被选择的字体的其他属性。例如，为了在一行文字之后再写一行就意味着你得知道当前字体的高度和行间距的大小。为了满足这个需要，AWT提供了FontMetrics类，这个类封装了关于字体的一些信息。下面让我们先来了解一些描述字体时常用的术语。

Height	在字体中最高的字符从底部到顶部的距离
Baseline	所有字符的底部所对齐的线
Ascent	从基线到字符顶部的距离
Descent	从基线到字符底部的距离
Leading	一行文字的底部到下一行的顶部的距离

正如你所知道的，我们在前面的许多例子中都使用了drawString( )方法。这个方法用当前的字体和颜色从一个特定的位置开始画出一个字符串。然而，这个位置是这些字符的基

线的最左边，而不是像别的一些绘图方法那样在左上角。所以不能把画框的坐标当作是画字符串的坐标。例如，你画了一个矩形在你的小应用程序中的(0,0)点，你将可以看到整个的矩形。但是如果你在同样的坐标处开始画一个字符串“Typesetting”，你将只能看到这些字符中的y,p和g的尾巴。你将看到，通过使用FontMetrics类，你可以为每一个字符串设置合适的显示位置。

FontMetrics定义了几种方法，这些方法可以帮助你管理文字的输出。在下面的表21-3中列出了我们常用的一些方法。这些方法可以帮助你在一个窗口中的适当位置显示文字。下面让我们来看几个例子。

表 21-3 FontMetrics 定义的一些方法

方法	描述
int bytesWidth(byte b[],int start, int numBytes)	返回字符串numBytes的宽度，并存放在数组b中，字符串start处
int charWidth(char c[],int start, int numChars)	返回字符串numChars的宽度，并存放在数组c中，字符串开始于start处
int charWidth(char c)	返回字符c的宽度
int charWidth(int c)	返回整数c的宽度
int getAscent()	返回字体从基线到字符顶部的距离
int getDescent()	返回字体从基线到字符底部的距离
Font getFont()	返回字体
int getHeight()	返回一行文字的高度。这个值可以被用于在一个窗口中输出多行文字
int getLeading()	返回文字行之间的间距
int getMaxAdvance()	返回最宽的字符的宽度，如果该值不可用返回-1
int getMaxAscent()	返回从基线到字符顶部的最大距离
int getMaxDescent()	返回从基线到字符底部的最大距离
int[] getWidths()	返回前256个字符的宽度
int stringWidth(String str)	返回特定字符串str的宽度
String toString()	返回与调用对象相应的字符串

21.11.1 显示多行文字

通常，FontMetrics类最常用的功能就是用来决定各行文字之间的距离。还有就是用来决定一个将被显示的字符串的长度。在这里，你将看到如何来实现这些功能的例子。

一般为了显示多行文字，你的程序通常必须不断的跟踪当前输出点的位置。每一次开始一个新行时，Y坐标必须高于下一行的开始点。每当一个字符串被显示之后，这个坐标X必须被设为这个字符串结尾的点。这样就可以使下一个字符串在前一个的结尾写出下。

为了决定行间距，你可以使用由getLeading( )方法返回的值。为了决定字体的整体高度，你可以通过由getAscent( )方法返回的值加上由getDescent( )方法返回的值来实现。然后，你可以用这些值确定每一行文本输出的位置。然而，在很多情况下，你将不需要用这些独立

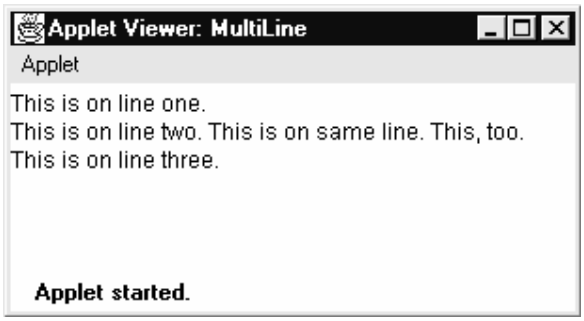
的值。当然，所有你需要知道的是一行的整体高度，它是行间距、字体上部分高、字体下部分高的和。获得这个值的最简单的方法是调用getHeight( )方法。在你想开始输出下一行文本时，你需要简单地增加这个值的Y坐标。

为了在同一行中前一部分的结束处开始输出，你必须知道每一个显示的字符串占有多少像素。为了获得这个值，你可以调用stringWidth( )方法。在每次显示一行时，你可以用这个值来决定X坐标。

接下来的这个小应用程序演示了如何在一个窗口中输出多行文字。它也显示了如何在一行中输出多个句子。请注意变量curX和curY，它们用于跟踪当前文本的输出位置。

```
// Demonstrate multiline output.
import java.applet.*;
import java.awt.*;
/*
<applet code="MultiLine" width=300 height=100>
</applet>
*/

public class MultiLine extends Applet {
  int curX=0, curY=0; // current position

  public void init() {
    Font f = new Font("SansSerif", Font.PLAIN, 12);
    setFont(f);
  }
  public void paint(Graphics g) {
    FontMetrics fm = g.getFontMetrics();

    nextLine("This is on line one.", g);
    nextLine("This is on line two.", g);
    sameLine(" This is on same line.", g);
    sameLine(" This, too.", g);
    nextLine("This is on line three.", g);
  }

  // Advance to next line.
  void nextLine(String s, Graphics g) {
    FontMetrics fm = g.getFontMetrics();

    curY += fm.getHeight(); // advance to next line
    curX = 0;
    g.drawString(s, curX, curY);
    curX = fm.stringWidth(s); // advance to end of line
  }

  // Display on same line.
  void sameLine(String s, Graphics g) {
    FontMetrics fm = g.getFontMetrics();

    g.drawString(s, curX, curY);
    curX += fm.stringWidth(s); // advance to end of line
  }
}
```

这个例子的输出如下所示：

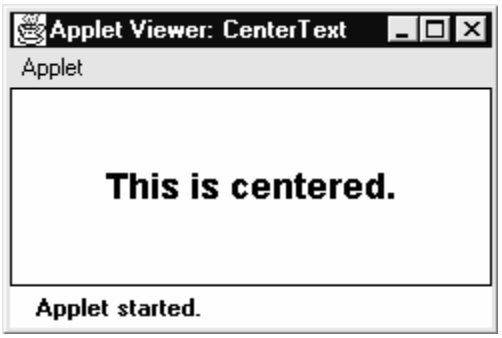


21.11.2 居中

这里有一个在窗口中文本居中（从左到右，从上到下）的例子。它获得了一个字符串的上半部高，下半部高和宽度，并且计算了使它居中显示的位置。

```
// Center text.
import java.applet.*;
import java.awt.*;
/*
  <applet code="CenterText" width=200 height=100>
  </applet>
*/

public class CenterText extends Applet {
  final Font f = new Font("SansSerif", Font.BOLD, 18);

  public void paint(Graphics g) {
    Dimension d = this.getSize();

    g.setColor(Color.white);
    g.fillRect(0, 0, d.width,d.height);
    g.setColor(Color.black);
    g.setFont(f);
    drawCenteredString("This is centered.", d.width, d.height, g);
    g.drawRect(0, 0, d.width-1, d.height-1);
  }

  public void drawCenteredString(String s, int w, int h,
                                 Graphics g) {
    FontMetrics fm = g.getFontMetrics();
    int x = (w - fm.stringWidth(s)) / 2;
    int y = (fm.getAscent() + (h - (fm.getAscent()
            + fm.getDescent()))/2);
    g.drawString(s, x, y);
  }
}
```

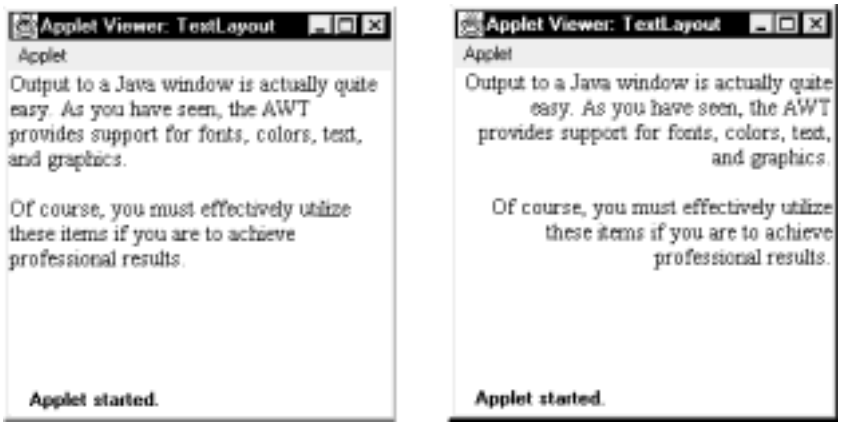
这个例子的输出如下所示：



21.11.3 多行文本对齐

如果你曾经使用过文字处理软件，你将已经看到过文字对齐的效果，即一行和多文字的边在一条直线上。例如，许多字处理软件可以左对齐或者是右对齐文本。大多数也可以中间对齐文本。在接下来的这个程序中，你将看到如何实现这些功能。

在这个程序中，要被调整的字符串被分成独立的文字。对于每个字，程序跟踪当前字体的长度，并且在当前行无法全部容纳该字时，自动的移至下一行。所有的输出行都在窗口中用当前选择的对齐样式来显示。每一次你在小应用程序窗口中单击鼠标，对齐样式都会改变。下面的这个例子演示了这些功能。




```
// Demonstrate text alignment.
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
/* <title>Text Layout</title>
   <applet code="TextLayout" width=200 height=200>
   <param name="text" value="Output to a Java window is actually
     quite easy.
     As you have seen, the AWT provides support for
     fonts, colors, text, and graphics. <P>  Of course,
     you must effectively utilize these items
     if you are to achieve professional results.">
    <param name="fontname" value="Serif">
```

```
  <param name="fontSize" value="14">
 </applet>
*/

public class TextLayout extends Applet {
 final int LEFT = 0;
 final int RIGHT = 1;
 final int CENTER = 2;
 final int LEFTRIGHT =3;
 int align;
 Dimension d;
 Font f;
 FontMetrics fm;
 int fontSize;
 int fh, bl;
 int space;
 String text;

 public void init() {
   setBackground(Color.white);
   text = getParameter("text");
   try {
     fontSize = Integer.parseInt(getParameter("fontSize"));}
   catch (NumberFormatException e) {
     fontSize=14;
   }
   align = LEFT;
   addMouseListener(new MyMouseAdapter(this));
 }

 public void paint(Graphics g) {
   update(g);
 }

 public void update(Graphics g) {
   d = getSize();
   g.setColor(getBackground());
   g.fillRect(0,0,d.width, d.height);
   if(f==null) f = new Font(getParameter("fontname"),
                        Font.PLAIN, fontSize);
   g.setFont(f);
   if(fm == null) {
       fm = g.getFontMetrics();
       bl = fm.getAscent();
       fh = bl + fm.getDescent();
       space = fm.stringWidth(" ");
   }

   g.setColor(Color.black);
   StringTokenizer st = new StringTokenizer(text);
   int x = 0;
   int nextx;
   int y = 0;
   String word, sp;
   int wordCount = 0;
```

```
  String line = "";
  while (st.hasMoreTokens()) {
    word = st.nextToken();
    if(word.equals("<P>")) {
      drawString(g, line, wordCount,
                 fm.stringWidth(line), y+bl);
      line = "";
      wordCount = 0;
      x = 0;
      y = y + (fh * 2);
    }
    else {
      int w = fm.stringWidth(word);
      if(( nextx = (x+space+w)) > d.width ) {
        drawString(g, line, wordCount,
                   fm.stringWidth(line), y+bl);
        line = "";
        wordCount = 0;
        x = 0;
        y = y + fh;
      }
      if(x!=0) {sp = " ";} else {sp = "";}
      line = line + sp + word;
      x = x + space + w;
      wordCount++;
    }
  }
  drawString(g, line, wordCount, fm.stringWidth(line), y+bl);
}

public void drawString(Graphics g, String line,
                       int wc, int lineW, int y) {
  switch(align) {
    case LEFT: g.drawString(line, 0, y);
      break;
   case RIGHT: g.drawString(line, d.width-lineW ,y);
      break;
    case CENTER: g.drawString(line, (d.width-lineW)/2, y);
      break;
    case LEFTRIGHT:
      if(lineW < (int)(d.width*.75)) {
        g.drawString(line, 0, y);
      }
      else {
        int toFill = (int)((d.width - lineW)/wc);
        int nudge = d.width - lineW - (toFill*wc);
        int s = fm.stringWidth(" ");
        StringTokenizer st = new StringTokenizer(line);
        int x = 0;
        while(st.hasMoreTokens()) {
          String word = st.nextToken();
          g.drawString(word, x, y);
          if(nudge>0) {
            x = x + fm.stringWidth(word) + space + toFill + 1;
            nudge--;
```

```
            } else {
              x = x + fm.stringWidth(word) + space + toFill;
            }
          }
        }
        break;
      }

  }

}

class MyMouseAdapter extends MouseAdapter {
  TextLayout tl;
  public MyMouseAdapter(TextLayout tl) {
    this.tl = tl;
  }
  public void mouseClicked(MouseEvent me) {
    tl.align = (tl.align + 1) % 4;
    tl.repaint();
  }
}
```

让我们来仔细的看一下这个小应用程序是如何工作的。这个小应用程序首先产生一些用来表示对齐方式的常量，然后定义了几个变量。在init()方法中获得了将被显示的文本。然后它用了一个try-catch块来初始化字体的大小，如果在html中没有写入fontSize参数，那么字体的大小将被设置为14。Text参数是一个很长的用HTML标记<P>来分段的字符串。

update( )方法在这个例子中是关键。它通过一个font metrics对象设置了字体，获得了基线和字体的高度。接下来，它产生了一个StringTokenizer对象，用它来接受在由text指定的字符串中的下一个分隔符。如果下一个分隔符是<p>，它将增加竖直空间。否则，update()将检查在当前字体中的分隔符的长度是否超过了列的宽度。如果没有分隔符且行已满，将调用自定义的drawString( )方法来输出此行。

在drawString( )方法中的头三种情况是比较简单的。每一次由line传进来的字符串通过由参数style决定对齐方式为左对齐、右对齐或者居中。当选择LEFTRIGHT方式时，字符串左右两边同时对齐。这意味着我们需要计算剩余的空间（即字符串的宽度和列的宽度之间的差），然后等距离的分布这些字。该类中的最后一个方法在每一次你在小应用程序窗口中点击鼠标时改变参数style。

21.12 关于文字和图形

虽然本章覆盖了当你显示文字和图形时将会用到的最重要的属性和最常用的方法，但是它也只是触及到了Java能力的表面。在这个领域，Java将做的越来越好。例如，Java 2为AWT增加了一个被叫做Java 2D的子系统。Java 2D支持更多的关于图形的组件，其中包括像坐标转换、旋转和伸缩等。它也提供了更多的图像特性。如果你对图像处理感兴趣，那么你将可以仔细研究Java 2D。

第 22 章　使用 AWT 控件、布局管理器和菜单

本章继续介绍抽象窗口工具包（AWT）。在这里我们将要学习Java定义的标准控件和布局管理器，讨论菜单和菜单栏，以及两个高级组件：对话框和文件对话框。同时，还要介绍事件处理。

控件（controls）是允许用户同你的应用程序用各种方式进行交互的组件，例如，一个常用的控件是下压式按钮。布局管理器自动安排组件在容器中的位置。这样，窗口的外观就可以由它所包含的控件来决定，并可以通过布局管理器来排放各个控件。

除了这些控件以外，框架窗口也能包含一个标准形式的菜单栏。每进入一个菜单栏就会激发一个下拉式菜单选项，用户可以从中选择。菜单栏总是位于窗口的顶部。虽然外观不同，但菜单栏与其他控件的处理方式是大致相同的。

虽然手工定位窗口中的组件是可能的，但做起来十分枯燥。布局管理器会自动完成此项任务。本章的第一节将介绍各种不同的控件，在这里将用到默认的布局管理器，该管理器使用从左到右、从上到下的方式来组织容器里的控件。一旦控件被覆盖，布局管理器将被检查。通过学习你将了解如何更好的管理控件的位置。

22.1　基 本 控 件

AWT支持下列类型的控件：

- 标签
- 下压式按钮
- 复选框
- 选择列表
- 列表框
- 滚动条
- 文本框

这些控件是Component的子类。

22.1.1　增加和删除控件

为了在窗口中包含一个控件，你必须将它加入窗口。实际上，你必须首先生成所需控件的实例，然后通过调用add()方法将它加入到窗口中，此方法是在Container类中定义的。Add()方法有几种形式。下面这种形式是本章前面部分所用到的：

```
Component add(Component compObj)
```

在这里，参数compObj是你将要加入的控件的一个实例，执行以上语句后一个compObj对象的引用被返回。一旦一个控件被加入，无论何时，只要父窗口被显示，它都会自动显示出来。

有时，当控件不再需要时，需要将它从窗口中删除。为此，你可以调用remove()方法。这个方法也是在Container类中定义的，如下所示：

```
void remove(Component obj)
```

在这里，参数obj是一个对你想要删除的控件的引用。你可以通过调用removeAll()方法删除所有的控件。

22.1.2　对控件的响应

除了标签这种被动的控件之外，所有的控件被用户访问时都会产生事件。例如，当用户点击按钮时，一个与下压式按钮有关的相应事件就被送出。一般来说，你的程序只需简单地实现相应的接口，并为每个你要监听的控件注册一个事件监听器。在第20章中我们介绍过，一旦一个事件监听器被安装，相应的事件就会被自动地发送给它。在下面几节中，对每一个控件都指定了相应的接口。

22.2　标　签

使用起来最简单的控件是标签。标签是Label类的对象，它包含了要显示的字符串。标签是被动的控件，不支持与用户的交互。Label类定义了以下的构造函数：

```
Label( )
Label(String str)
Label(String str, int how)
```

第一种形式生成一个空白标签；第二种形式生成一个包含由参数str所设定的字符串的标签，这个字符串是左对齐的；第三种形式生成一个包含由参数str所设定的字符串的标签，并由整数how决定了对齐方式。How的值必须为以下常量之一：Label.LEFT,Label.RIGHT或Label.CENTER。

你能通过使用setText()方法来设定或改变标签中的文本。通过调用getText()方法，你可以获得当前的标签。这些方法如下所示：

```
void setText(String str)
String getText( )
```

对于方法setText()来说，参数str指定了新的标签。对于getText()方法来说，当前的标签中的文本被返回。

在标签中，你还可以通过调用setAlignment()方法来设定字符串的对齐方式。同时，你也可以通过调用getAlignment()方法来获得当前的对齐方式。这些方法如下所示：

```
void setAlignment(int how)
int getAlignment( )
```

这里，整数how必须是前面介绍的对齐方式中的一种。

下面的例子生成了三个标签，并将它们加入到一个小应用程序中。

```
// Demonstrate Labels
import java.awt.*;
import java.applet.*;
/*
<applet code="LabelDemo" width=300 height=200>
</applet>
*/

public class LabelDemo extends applet {
  public void init() {
  Label one = new Label("One");
    Label two = new Label("Two");
    Label three = new Label("Three");
    // add labels to applet window
    add(one);
    add(two);
    add(three);
  }
}
```

下面是由LabelDemo这个小应用程序所生成的窗口。请注意，在窗口中，标签由默认的布局管理器组织。稍后，你将看到如何更精确地控制标签的位置。

22.3 使 用 按 钮

使用最广泛的控件是下压式按钮。下压式按钮是一种包含一个标签并能在被按下时产生事件的控件。下压式按钮是Button类的对象。Button类定义了如下两种构造函数：

```
Button( )
Button(String str)
```

第一种方式生成一个空的按钮。第二种方式生成了一个以参数str为标签的按钮。

在按钮被生成之后，你能通过调用setLabel()方法设定它们的标签。同时，你也可以通过调用getLabel()方法来获得它的标签。这些方法如下所示：

```
void setLabel(String str)
```

```
String getLabel( )
```

这里，参数str成为按钮的新标签。

22.3.1　按钮事件处理

每当按钮被按下时，就产生一个动作事件。动作事件被发送给任何先前注册过的事件监听器，这些监听器为了从组件获得事件通知而被注册。每一个事件监听器都实现了ActionListener接口。这个接口中定义了actionPerformed()方法，它在事件产生时被调用。一个ActionEvent对象作为参数被提供给此方法。它既包含了对产生事件的按钮的引用，也包含了对按钮标签的字符串的引用。通常，这两者中的任何一个都能用来惟一确定这个按钮，这一点你将会在下面看到。

下面的这个例子产生了三个标签分别为“Yes”，“No”和“Undecided”的按钮。每当按钮被按下时，将显示一条消息，用于报告是哪一个按钮被按下。在这个版本中，按钮的标签被用来判断哪一个按钮被按下。标签可以通过调用方法getActionCommand()来获得，这个getActionCommand( )方法属于被传递给actionPerformed( )方法的ActionEvent对象。

```
// Demonstrate Buttons
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="ButtonDemo" width=250 height=150>
  </applet>
*/

public class ButtonDemo extends applet implements ActionListener {
  String msg = "";
  Button yes, no, maybe;
  public void init() {
    yes = new Button("Yes");
    no = new Button("No");
    maybe = new Button("Undecided");

    add(yes);
    add(no);
    add(maybe);

    yes.addActionListener(this);
    no.addActionListener(this);
    maybe.addActionListener(this);
  }

  public void actionPerformed(ActionEvent ae) {}
    String str = ae.getActionCommand();
    if(str.equals("Yes")) {
      msg = "You pressed Yes.";
    }
    else if(str.equals("No")) {
      msg = "You pressed No.";
    }
```

```
    else {
      msg = "You pressed Undecided.";
    }
    repaint();
  }

  public void paint(Graphics g) {
    g.drawString(msg, 6, 100);
  }
}
```

ButtonDemo小应用程序的标准输出如图22-1所示。

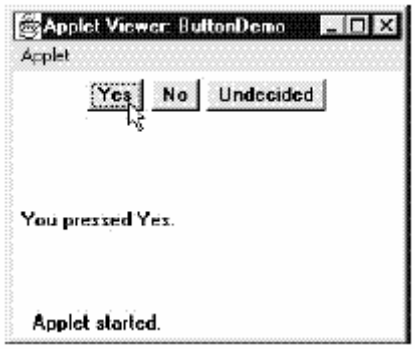


图 22-1 ButtonDemo 小应用程序的输出

我们提到过，除了比较按钮标签之外，还可以用另一种方式来确定是哪一个按钮被按下，第2种方式是通过比较你加入到窗口的按钮对象与调用getSource()所获得的对象来进行的。为完成此功能，你必须在对象被加入时保存它们的列表。下面的小应用程序示范了如何使用这个方法：

```
// Recognize Button objects.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="ButtonList" width=250 height=150>
  </applet>
*/
public class ButtonList extends applet implements ActionListener {
  String msg = "";
  Button bList[] = new Button[3];

  public void init() {
    Button yes = new Button("Yes");
    Button no = new Button("No");
    Button maybe = new Button("Undecided");

    // store references to buttons as added
    bList[0] = (Button) add(yes);
    bList[1] = (Button) add(no);
```

```
    bList[2] = (Button) add(maybe);

    // register to receive action events
    for(int i = 0; i < 3; i++) {
      bList[i].addActionListener(this);
    }
  }

  public void actionPerformed(ActionEvent ae) {
    for(int i = 0; i < 3; i++) {
      if(ae.getSource() == bList[i]) {
        msg = "You pressed " + bList[i].getLabel();
      }
    }
    repaint();
  }

  public void paint(Graphics g) {
     g.drawString(msg, 6, 100);
  }
}
```

在这个例子中，程序在按钮被加入到小应用程序窗口时，就以数组方式保存了每个按钮的引用（注意当按钮被加入时add()方法返回它的引用）。接下来在actionPerformed()方法中，此数组被用来确定被按下的按钮。

对于简单的小应用程序来说，用标签来识别按钮通常要容易得多。但在某些情况下，例如如果你要在程序的执行过程中改变按钮中的标签，或者使用了有相同标签的按钮，这时使用它的对象引用来判断按压了哪一个按钮就容易得多。

22.4　使用复选框

复选框（Checkbox）是一个用来将选项开启或关闭的控件。它由一个小框组成，其中可能包含一个复选标记。每一个复选框都有一个标签来描述它所代表的选项。你可以通过点击复选框来改变它的状态。复选框可以单独使用，也可作为一个组的一部分来使用。复选框是Checkbox类的对象：

Checkbox支持以下这些构造函数：

```
Checkbox( )
Checkbox(String str)
Checkbox(String str, boolean on)
Checkbox(String str, boolean on, CheckboxGroup cbGroup)
Checkbox(String str, CheckboxGroup cbGroup, boolean on)
```

第一种形式生成了一个所含标签初始为空白的复选框，复选框未被选中。第二种形式生成了一个标签由参数str指定的复选框，复选框未被选中。第三种形式允许设定复选框的初始状况，如果参数on为true，则复选框初始被选中；否则它被清除。第四和第五种形式生成了一个由参数str设定的标签，它所属的组由参数cbGroup确定。如果复选框不属于一个组，

参数cbGroup必须为null（复选框组在下一节讲述）。参数on的值决定了复选框的初始状态。

为了获得复选框的当前状态，可以调用getState()方法。如果要设定复选框的状态，可以调用setState()。你还能通过调用getLabel()方法来获得与复选框相关的当前标签。为了设置复选框的标签，可以调用setLabel()方法。这些方法如下所示：

```
boolean getState( )
void setState(boolean on)
String getLabel( )
void setLabel(String str)
```

在这里，如果参数on为true，则复选框被选中；如果为false，则被清除。参数str传递的字符串成为与所调用的复选框相关的新标签。

22.4.1 处理复选框

每次复选框被选中或被取消选定时，就产生一个事件。该事件被发送给任何先前注册过的事件监听器，这些监听器为了从组件获得事件通知而被注册。每个监听器都实现了一个ItemListener接口。这个接口定义了itemStateChanged()方法。一个ItemEvent对象被作为参数提供给这个方法，其中包含了关于事件的信息（例如，是否它被选中）。

下面的程序生成了四个复选框。第一个复选框的初始状态是被选中。每一个复选框的状态都被显示。每次你改变复选框的状态时，状态显示都被更新。

```
// Demonstrate check boxes.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="CheckboxDemo" width=250 height=200>
  </applet>
*/

public class CheckboxDemo extends applet implements ItemListener {
  String msg = "";
  Checkbox Win98, winNT, solaris, mac;

  public void init() {
    Win98 = new Checkbox("Windows 98", null, true);
    winNT = new Checkbox("Windows NT/2000");
    solaris = new Checkbox("Solaris");
    mac = new Checkbox("MacOS");

    add(Win98);
    add(winNT);
    add(solaris);
    add(mac);

    Win98.addItemListener(this);
    winNT.addItemListener(this);
    solaris.addItemListener(this);
    mac.addItemListener(this);
  }
```

```
  public void itemStateChanged(ItemEvent ie) {
    repaint();
  }

  // Display current state of the check boxes.
  public void paint(Graphics g) {
    msg = "Current state: ";
    g.drawString(msg, 6, 80);
    msg = "  Windows 98: " + Win98.getState();
    g.drawString(msg, 6, 100);
    msg = "  Windows NT/2000: " + winNT.getState();
    g.drawString(msg, 6, 120);
    msg = "  Solaris: " + solaris.getState();
    g.drawString(msg, 6, 140);
    msg = "  MacOS: " + mac.getState();
    g.drawString(msg, 6, 160);
  }
}
```

这个例子的输出如图22-2所示。

图 22-2　CheckboxDemo 小应用程序的输出

22.5　复 选 框 组

生成一系列互斥的复选框是可能的，在任何时间其中只能有一个复选框被选中。这些复选框通常被称为单选按钮，之所以起这个名字，是因为它们的行为就像是汽车上无线广播的选台器，任何时间只能选一个电台。为生成一系列互斥的复选框，你在构造复选框时必须先定义它们所属的组，并对这个组进行设定。

通过调用getSelectedCheckbox( )方法可以确定组中哪个复选框被选中。通过调用setSelectedCheckbox( )方法，你能设定哪个复选框被选中。这些方法如下所示：

```
Checkbox getSelectedCheckbox( )
void setSelectedCheckbox(Checkbox which)
```

在这里，参数which是你想要选定的复选框。先前选中的复选框将被关闭。
下面是一个使用了复选框作为组的一部分的例子：

```
// Demonstrate check box group.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="CBGroup" width=250 height=200>
  </applet>
*/

public class CBGroup extends applet implements ItemListener {
  String msg = "";
  Checkbox Win98, winNT, solaris, mac;
  CheckboxGroup cbg;

  public void init() {
    cbg = new CheckboxGroup();
    Win98 = new Checkbox("Windows 98", cbg, true);
    winNT = new Checkbox("Windows NT/2000", cbg, false);
   solaris = new Checkbox("Solaris", cbg, false);

   mac = new Checkbox("MacOS", cbg, false);

    add(Win98);
    add(winNT);
    add(solaris);
    add(mac);

    Win98.addItemListener(this);
    winNT.addItemListener(this);
    solaris.addItemListener(this);
    mac.addItemListener(this);
  }

  public void itemStateChanged(ItemEvent ie) {
    repaint();
  }

  // Display current state of the check boxes.
  public void paint(Graphics g) {
    msg = "Current selection: ";
    msg += cbg.getSelectedCheckbox().getLabel();
    g.drawString(msg, 6, 100);
  }
}
```

CBGroup小应用程序产生的输出如图22-3所示。注意，这里的复选框是圆形的。

图 22-3　CBGroup 小应用程序的输出

22.6　选择框控件

Choice类用来生成弹出式的列表项，用户可以从中选择。因此，可以说选择框（Choice）就是一种形式的菜单。当未被激发时，选择框仅占据刚刚够用的空间来显示当前的被选项。当用户点击它时，所有选项的列表将被弹出，可以从中作出新的选择。列表中的每一项都是一个字符串，以左对齐的标签形式出现，并按被加入选择框对象的顺序排列。选择框仅仅定义了默认的构造函数，这个构造函数将产生一个空列表。

为了将新的选项加入列表，可以调用addItem()或add()方法。如下所示：

```
void addItem(String name)
void add(String name)
```

在这里，参数name是被加入选项的名字。被加入到列表中的选项的显示顺序是由调用add()或addItem()方法的顺序决定的。

为了确定当前哪些选项被选中，你可以调用getSelectedItem( ) 或 getSelectedIndex( )方法。这些方法如下所示。

```
String getSelectedItem( )
int getSelectedIndex( )
```

GetSelectedItem()方法返回一个包含相应项名字的字符串，而GetSelectedIndex()方法返回选项索引。第一个选项的索引为零。在默认的情况下，第一个被加入列表的选项被选中。

为了获得列表中选项的数目，你可以调用方法getItemCount()。通过调用select()方法，并给出一个从零开始的整数索引，或一个与列表中的名字相匹配的字符串，你能设定当前的被选项。这些方法如下所示：

```
int getItemCount( )
void select(int index)
```

```
void select(String name)
```

通过给出一个索引，你可以调用方法getItem()获得此索引所代表的选项的名字。这个方法如下所示：

```
String getItem(int index)
```

在这里，参数index设定了所要求的选项的索引。

22.6.1 处理选择框列表

每当一个选择框选项被选时，就产生了一个选项事件。该事件被发送给任何先前注册过的事件监听器，这些监听器为了从组件获得事件通知而被注册。每个监听器都实现了一个ItemListener接口。此接口定义了itemStateChanged()方法。一个ItemEvent对象被作为参数提供给这个方法。

这里有一个生成两个选择框菜单的例子。一个选择框用来选择操作系统，另一个则用于选择浏览器：

```
// Demonstrate Choice lists.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="ChoiceDemo" width=300 height=180>
  </applet>
*/

public class ChoiceDemo extends applet implements ItemListener {
  Choice os, browser;
  String msg = "";

  public void init() {
    os = new Choice();
    browser = new Choice();

    // add items to os list
    os.add("Windows 98");
    os.add("Windows NT/2000");
    os.add("Solaris");
    os.add("MacOS");

    // add items to browser list
    browser.add("Netscape 1.1");
    browser.add("Netscape 2.x");
    browser.add("Netscape 3.x");
    browser.add("Netscape 4.x");

    browser.add("Internet Explorer 3.0");
    browser.add("Internet Explorer 4.0");
    browser.add("Internet Explorer 5.0");

    browser.add("Lynx 2.4");
```

```
    browser.select("Netscape 4.x");

    // add choice lists to window
    add(os);

   add(browser);

// register to receive item events
    os.addItemListener(this);
    browser.addItemListener(this);
  }

  public void itemStateChanged(ItemEvent ie) {
    repaint();
  }

  // Display current selections.
  public void paint(Graphics g) {
    msg = "Current OS: ";
    msg += os.getSelectedItem();
    g.drawString(msg, 6, 120);
    msg = "Current Browser: ";
    msg += browser.getSelectedItem();
    g.drawString(msg, 6, 140);
  }
}
```

这个例子的输出如图22-4所示。

图 22-4　ChoiceDemo 小应用程序的输出

22.7　使用列表框

列表框（List）类提供了一个简洁的多选项滚动列表。与Choice对象不同的是，Choice对象只能显示菜单中的单一的被选项，而List对象能被构造成在可见窗口显示任意数量选项

的对象。在列表框中可以同时选择多个选项。列表框提供了如下的构造函数：

```
List( )
List(int numRows)
List(int numRows, boolean multipleSelect)
```

第一种形式生成了一个List控件，它在任何时间仅允许一个选项被选中。在第二种形式中，参数numRows的值指定了列表框中一直可见的选项数量（其他的在需要时滚动可见）。在第三种形式中，如果参数multipleSelect 为true，则用户可一次选两个或更多的选项。如果它为false，则仅有一项可选。

为了向列表框加入一个选项，可以调用add()方法。它有以下两种形式：

```
void add(String name)
void add(String name, int index)
```

在这里，参数name是被加入列表框的项的名字。第一种形式在列表框尾加入选项。第二种形式在参数index所确定的索引处加入选项。索引从零开始。你可将它设为-1以将选项加入列表框的末尾。

对于仅允许一个选项被选中的列表框来说，你能通过调用getSelectedItem( ) 或getSelectedIndex( ) 来确定当前哪一项被选中。这些方法所示如下：

```
String getSelectedItem( )
int getSelectedIndex( )
```

getSelectedItem( )方法返回一个包含选项名字的字符串。如果多于一个的选项被选中或没有任何选项被选中，就返回null。getSelectedIndex( )返回选项的索引。第一项的索引为零。如果多个选项被选中，或者无选项被选中，则返回-1。

对于允许选中多个选项的列表框来说，你必须调用getSelectedItems( )或getSelectedIndexes( )方法（所示如下）来确定当前选项，如下所示：

```
String[ ] getSelectedItems( )
int[ ] getSelectedIndexes( )
```

getSelectedItems( )方法返回一个包含了当前被选项的名字的数组。getSelectedIndexes( )返回一个包含当前被选中的项的索引。

为获得列表框中项的数目，可以调用getItemCount( )方法。给定一个从零开始的整数索引，你能通过调用select()方法设定当前被选的项。这些方法如下所示：

```
int getItemCount( )
void select(int index)
```

如果给定索引，你能通过调用getItem()方法，获得与此索引相关的选项的名字。该方法的一般形式如下：

```
String getItem(int index)
```

这儿，index设定了想要的项的索引。

22.7.1　处理列表框

为了处理列表框事件，你需要实现ActionListener接口。每当一个列表框中的项被双击时，就产生一个ActionEvent对象。它的getActionCommand( )方法能被用来接收新的被选项的名字。而且，每当一项通过单击被选定或被取消选定时，一个ItemEvent对象就被产生。它的getStateChange( )方法能被用来确定一次选定或取消选定动作是否触发该事件。getItemSelectable( )返回触发这个事件的对象的引用。

下面是一个有关列表框的例子，它将上一节选择框控件转变为列表框控件：一个多选列表框框和一个单选列表框。

```
// Demonstrate Lists.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="ListDemo" width=300 height=180>
  </applet>
*/

public class ListDemo extends applet implements ActionListener {
  List os, browser;
  String msg = "";

  public void init() {
    os = new List(4, true);
    browser = new List(4, false);


   // add items to os list
os.add("Windows 98");
    os.add("Windows NT/2000");
    os.add("Solaris");
    os.add("MacOS");

    // add items to browser list
    browser.add("Netscape 1.1");
    browser.add("Netscape 2.x");
    browser.add("Netscape 3.x");
    browser.add("Netscape 4.x");

    browser.add("Internet Explorer 3.0");
    browser.add("Internet Explorer 4.0");
    browser.add("Internet Explorer 5.0");

    browser.add("Lynx 2.4");
    browser.select(1);

    // add lists to window
    add(os);
    add(browser);

    // register to receive action events
```

```
    os.addActionListener(this);
    browser.addActionListener(this);
  }

  public void actionPerformed(ActionEvent ae) {
    repaint();
  }

  // Display current selections.
  public void paint(Graphics g) {
    int idx[];

    msg = "Current OS: ";
    idx = os.getSelectedIndexes();
    for(int i=0; i<idx.length; i++)
      msg += os.getItem(idx[i]) + "  ";
    g.drawString(msg, 6, 120);
    msg = "Current Browser: ";
    msg += browser.getSelectedItem();
    g.drawString(msg, 6, 140);
  }
}
```

ListDemo applet的输出如图22-5所示。请注意浏览器列表框中有一个滚动条，这是由于生成列表框时指定的显示行数与实际项目总数不符所致。

图 22-5 ListDemo 小应用程序的输出

22.8 管理滚动条

滚动条（scroll bar）被用来对最大值与最小值之间的连续值进行选定。滚动条可被水平或垂直定向。一个滚动条实际上是几个部分的组合。每一端都有一个箭头，单击它可使滚动条的当前值向箭头所指的方向移动一个单位。滚动条的当前值距最大值和最小值的相对位置可由滚动条的滑块（或拇指）来提示。滑块能由用户拖向新的位置，拖动之后滚动

条将反映此值。用户可以点击滑块的任何一侧的背景区域使滑块沿那个方向以多于1的增幅移动。一般来说，这个动作可以使页面以多种形式向前或向后翻。滚动条被封装在Scrollbar类中。

Scrollbar定义了下面的构造函数：

```
Scrollbar( )
Scrollbar(int style)
Scrollbar(int style, int initialValue, int thumbSize, int min, int max)
```

第一种形式生成了一个垂直滚动条。第二和第三种形式允许你指定滚动条的方向。如果 style 为 Scrollbar.VERTICAL，则生成一个垂直滚动条。如果 style 被设为 Scrollbar.HORIZONTAL，则滚动条为水平的。在构造函数的第三种形式中，滚动条的初始值由initialValue传递。滑块的高度所代表的单元的个数由thumbSize所传递。滚动条的最大和最小值由min和max设定。

如果用前两个构造函数来构造滚动条，那么你需要在使用它前，通过使用setValues()来设定参数，这个方法如下所示：

```
void setValues(int initialValue, int thumbSize, int min, int max)
```

其中的参数与第三个构造函数的参数意义相同。

为了获得滚动条的当前值，可以调用getValue()方法。它返回当前的设置。为了设定当前值，可以调用setValue()。这些方法如下所示：

```
int getValue( )
void setValue(int newValue)
```

这里，参数newValue为滚动条指定了新值。当你设定一个新值，滚动条内部的滑块将被定位以反映新值。

你也能够通过getMinimum( ) 和getMaximum( )方式来查看最大值和最小值，如下所示：

```
void getMininum
void getMaximum
```

这两种方法返回所要求的数。

在默认的情况下，每次滚动条向上或向下滚动一行时，它的值都要加一或减一。你可以通过调用setUnitIncrement( )来改变这个增量。默认情况下，向上和向下滚动一页的增值为10。你能通过调用setBlockIncrement( )来改变此值。方法如下所示：

```
void setUnitIncrement(int newIncr)
void setBlockIncrement(int newIncr)
```

22.8.1　处理滚动条

为处理滚动条事件，你需要实现AdjustmentListener接口。每次用户与滚动条进行交互时，都将生成一个AdjustmentEvent对象。它的getAdjustmentType( )方法被用来确定调节类型。调节事件的类型如下所示：

BLOCK_DECREMENT	产生一个向下翻页事件
BLOCK_INCREMENT	产生一个向上翻页事件
TRACK	产生一个绝对跟踪事件
UNIT_DECREMENT	滚动条中的向下按钮被按下
UNIT_INCREMENT	滚动条中的向上按钮被按下

下面的例子生成了水平和垂直的滚动条并显示它们的当前设定。如果你在窗口中拖动鼠标，每次拖动事件的坐标将被用来更新滚动条。在当前的拖动位置上显示一个星号。

```
// Demonstrate scroll bars.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="SBDemo" width=300 height=200>
  </applet>
*/

public class SBDemo extends applet
  implements AdjustmentListener, MouseMotionListener {
  String msg = "";
  Scrollbar vertSB, horzSB;

  public void init() {
    int width = Integer.parseInt(getParameter("width"));
    int height = Integer.parseInt(getParameter("height"));

    vertSB = new Scrollbar(Scrollbar.VERTICAL,
                           0, 1, 0, height);
    horzSB = new Scrollbar(Scrollbar.HORIZONTAL,
                           0, 1, 0, width);
    add(vertSB);
    add(horzSB);

    // register to receive adjustment events
    vertSB.addAdjustmentListener(this);
    horzSB.addAdjustmentListener(this);

    addMouseMotionListener(this);
  }

  public void adjustmentValueChanged(AdjustmentEvent ae) {
    repaint();
  }

  // Update scroll bars to reflect mouse dragging.
  public void mouseDragged(MouseEvent me) {
    int x = me.getX();
    int y = me.getY();
    vertSB.setValue(y);
    horzSB.setValue(x);
    repaint();
  }
```

```
  // Necessary for MouseMotionListener
  public void mouseMoved(MouseEvent me) {
  }

  // Display current value of scroll bars.
  public void paint(Graphics g) {
    msg = "Vertical: " + vertSB.getValue();
    msg += ",  Horizontal: " + horzSB.getValue();
    g.drawString(msg, 6, 160);

    // show current mouse drag position
    g.drawString("*", horzSB.getValue(),
                vertSB.getValue());
  }
}
```

SBDemo小应用程序的输出如图22-6所示。

图 22-6　SBDemo 小应用程序的输出

22.9　使用文本区

文本区（TextField）类实现了一个单行的文本区，通常称为编辑控件。文本区允许用户输入字符串，使用箭头键、剪切和粘贴键，以及鼠标选定一些方式来编辑文本。TextField是TextComponent的一个子类。TextField定义了如下构造函数：

```
TextField( )
TextField(int numChars)
TextField(String str)
TextField(String str, int numChars)
```

第一种形式生成了一个默认的文本区。第二种形式生成了一个有参数numChars确定字符个数的文本区。第三种形式用参数str中包含的字符串来初始化文本区。第四种形式初始化一个文本区并设定它的宽度。

TextField（和它的超类 TextComponent）提供了多种初始化文本区的方法。为获得当前包含在文本区中的字符串，调用getText()方法；为设置文本，调用setText()。这些方法如下所示：

```
String getText( )
void setText(String str)
```

在这里，参数str为新的字符串。

用户能在文本区中选择一部分字符。而且，通过调用select()，还能在程序控制下选择一部分文本。通过调用getSelectedText()，你的程序能获得当前被选定的文本。这些方法如下所示：

```
String getSelectedText( )
void select(int startIndex, int endIndex)
```

GetSelectedText()方法返回被选定的文本。方法select()选择了从startindex开始至endIndex-1结束的字符。

你能通过调用setEditable()方法来控制文本区的内容是否可被用户改变。通过调用isEditable()方法，你能判定文本区的可编辑性。这些方法如下所示：

```
boolean isEditable( )
void setEditable(boolean canEdit)
```

如果文本能被改变，IsEditable()返回true；否则返回false。在setEditable()中，如果canEdit为true，文本将可被改变；如果为false，则文本不能改变。

有时候，你不想显示用户输入的文本，例如密码，则可以在字符被键入时调用setEchoChar()来禁止字符的回显。此方法指定了在文本区输入字符时所显示的字符（这样，实际键入的字符将不被显示）。你能用echoCharIsSet()方法来查看文本区是否在这种模式下。通过调用getEchoChar()方法，你能检索回显的字符。这些方法如下所示：

```
void setEchoChar(char ch)
boolean echoCharIsSet( )
char getEchoChar( )
```

这里，ch是指定的被回显的字符。

22.9.1 处理TextField

由于文本区执行它们自己的编辑函数，你的程序一般不会响应出现在文本区的单键事件。但是，你可能想让程序在用户按下ENTER键时作出响应，在这种情况下，就应该创建一个动作事件。

这里是一个生成经典用户名和密码屏幕的例子：

```
// Demonstrate text field.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="TextFieldDemo" width=380 height=150>
```

```
  </applet>
*/

public class TextFieldDemo extends applet
  implements ActionListener {

  TextField name, pass;

  public void init() {
    Label namep = new Label("Name: ", Label.RIGHT);
    Label passp = new Label("Password: ", Label.RIGHT);
    name = new TextField(12);
    pass = new TextField(8);
    pass.setEchoChar('?');

    add(namep);
    add(name);
    add(passp);
    add(pass);

    // register to receive action events
    name.addActionListener(this);
    pass.addActionListener(this);
  }

  // User pressed Enter.
  public void actionPerformed(ActionEvent ae) {
    repaint();
  }

  public void paint(Graphics g) {
     g.drawString("Name: " + name.getText(), 6, 60);
     g.drawString("Selected text in name: "
                + name.getSelectedText(), 6, 80);

     g.drawString("Password: " + pass.getText(), 6, 100);
  }
}
```

TextFieldDemo小应用程序的输出如图22-7所示。

图 22-7　TextFieldDemo 小应用程序的输出

22.10 使用TextArea

有时一个单行的文本输入对于一个给定的任务是不够的。为处理这些情况，AWT中提供了一个简单的多行编辑器，称为TextArea。下面是TextArea的构造函数：

```
TextArea( )
TextArea(int numLines, int numChars)
TextArea(String str)
TextArea(String str, int numLines, int numChars)
TextArea(String str, int numLines, int numChars, int sBars)
```

在这里，参数numLines指定了以行为单位的文本区的高度，参数numChars指定了以字符为单位的文本区宽度。初始文本由参数str指定。在第五种形式中，你能为控件指定滚动条。参数sBars必须为下列值中的一个：

SCROLLBARS_BOTH	SCROLLBARS_NONE
SCROLLBARS_HORIZONTAL_ONLY	SCROLLBARS_VERTICAL_ONLY

TextArea是TextComponent的一个子类，它支持前一节描述过的getText( ), setText( ), getSelectedText( ), select( ), isEditable( )和setEditable( )方法。

TextArea 加入了以下方法：

```
void append(String str)
void insert(String str, int index)
void replaceRange(String str, int startIndex, int endIndex)
```

append()方法向当前文本的末尾追加由str设定的字符串。insert()在设定的索引处插入由str所传递的字符串。调用replaceRange()方法，可替换文本，它以str所传递的文本替换从startIndex到endIndex-1处的文本。

文本区是自包含控件。你的程序实际上没有什么管理开销。文本区仅产生获得焦点和失去焦点的事件。通常，你的程序仅在需要时才去获取当前的文本。

下面的程序生成了一个TextArea控件：

```
// Demonstrate TextArea.
import java.awt.*;
import java.applet.*;
/*
<applet code="TextAreaDemo" width=300 height=250>
</applet>
*/

public class TextAreaDemo extends applet {
  public void init() {
      String val = "There are two ways of constructing " +
      "a software design.\n" +
      "One way is to make it so simple\n" +
      "that there are obviously no deficiencies.\n" +
      "And the other way is to make it so complicated\n" +
```

```
      "that there are no obvious deficiencies.\n\n" +
      "       -C.A.R. Hoare\n\n" +
      "There's an old story about the person who wished\n" +
      "his computer were as easy to use as his telephone.\n" +
      "That wish has come true,\n" +
      "since I no longer know how to use my telephone.\n\n" +
      "       -Bjarne Stroustrup, AT&T, (inventor of C++)";

    TextArea text = new TextArea(val, 10, 30);
    add(text);
  }
}
```

这里是TextAreaDemo小应用程序的输出：

22.11　理解布局管理器

到现在为止，我们介绍的所有组件都是由默认的布局管理器来布置的。我们在本章开头提到过，布局管理器会通过某种算法，自动的在窗口中安排控件。如果你曾为其他的GUI环境编过程序，例如Windows，那么你就会习惯于手工布置你的控件。虽然手工布置控件也是可能的，一般来说，你不想这样做。有两个原因，首先，手工布置巨大数量的组件是十分繁琐的。其次，有时候，在你需要安排控件时，高度、宽度信息会由于本地工具包组件还未实现而不可得。这是一个鸡与蛋的情况：何时使用一个给定尺寸的组件来相对于其他组件安放是十分令人困惑的。

每个Container对象都有一个与它相关的布局管理器。布局管理器是一个实现LayoutManager接口的任何类的实例。布局管理器由setLayout()方法设定。如果没有对setLayout()方法的调用，那么默认的布局管理器就被使用。每当一个容器被调整大小时(或第一次被形成时)，布局管理器都被用来布置它里面的组件。

setLayout()方法有以下基本形式：

```
void setLayout(LayoutManager layoutObj)
```

在这里，参数layout是所需布局管理器的一个引用。如果你想禁用布局管理器从而手工布置组件，则将layout赋值为null。如果这样做的话，你将需要使用Component定义的方法setBounds()来手工决定每个组件的形状和位置。一般来说，推荐使用布局管理器。

每个布局管理器都跟踪按名字存储的组件列表。每当你向一个容器加入一个组件时，布局管理器都会得到通知。每当容器需要调整大小时，布局管理器就通过它的方法minimumLayoutSize( )和preferredLayoutSize( ) 考虑该问题。每个被布局管理器管理的组件都包含getPreferredSize( )和getMinimumSize( )方法。这些方法分别返回显示每个组件所需的预设尺寸和最小尺寸。如果可能的话，布局管理器将尽可能的满足这些要求，同时维持布局策略的完整性。你可以在子类中为控件重载这些方法；否则默认的值将被提供。

Java有几种预定义的LayoutManager类，接下来就讨论这些类，以便你能使用最适合你的应用程序的布局管理器。

22.11.1 FlowLayout

流式布局管理器（FlowLayout）是默认的布局管理器。这是前面的例子所使用的布局管理器。流式布局管理器实现了一种简单的布局风格，它类似于在一个文本编辑器中文字的流动方式。组件从左上角开始，按从左向右、从上到下的方式布置。当更多的组件在一行上排列不下时，下一个组件就出现在下一行上。在每个组件的上边、下边、左边、右边都留有小的空间。下面是FlowLayout的构造函数：

```
FlowLayout()
FlowLayout(int how)
FlowLayout(int how, int horz, int vert)
```

第一种形式生成了默认的布局，它将组件置于中心，在每个组件之间留下五个像素的距离。第二种形式让你设定每一行组件的对齐方式。参数How的有效值如下所示：

```
FlowLayout.LEFT
FlowLayout.CENTER
FlowLayout.RIGHT
```

这些值，分别设定了左对齐、居中和右对齐方式。第三种形式允许你在horz和vert中分别设定组件间的水平和垂直距离。

下面是本章前面所示的CheckboxDemo的一个版本，它被修改成使用左对齐流式布局管理器。

```
// Use left-aligned flow layout.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="FlowLayoutDemo" width=250 height=200>
  </applet>
```

```
*/

public class FlowLayoutDemo extends applet
  implements ItemListener {

  String msg = "";
  Checkbox Win98, winNT, solaris, mac;

  public void init() {
    // set left-aligned flow layout
    setLayout(new FlowLayout(FlowLayout.LEFT));

    Win98 = new Checkbox("Windows 98", null, true);
    winNT = new Checkbox("Windows NT/2000");
    solaris = new Checkbox("Solaris");
    mac = new Checkbox("MacOS");

    add(Win98);
    add(winNT);
    add(solaris);
    add(mac);

    // register to receive item events
    Win98.addItemListener(this);
    winNT.addItemListener(this);
    solaris.addItemListener(this);
    mac.addItemListener(this);
 }

  // Repaint when status of a check box changes.
  public void itemStateChanged(ItemEvent ie) {
    repaint();
  }

  // Display current state of the check boxes.
  public void paint(Graphics g) {
    msg = "Current state: ";
    g.drawString(msg, 6, 80);
    msg = "  Windows 98: " + Win98.getState();
    g.drawString(msg, 6, 100);
    msg = "  Windows NT/2000: " + winNT.getState();
    g.drawString(msg, 6, 120);
    msg = "  Solaris: " + solaris.getState();
    g.drawString(msg, 6, 140);
    msg = "  Mac: " + mac.getState();
    g.drawString(msg, 6, 160);
  }
}
```

下面是FlowLayoutDemo小应用程序的输出：

请自行与前面的图22-2所示的CheckboxDemo小应用程序的输出作比较。

22.11.2 BorderLayout

BorderLayout类为最顶层的窗口实现了一个普通的布局风格。它在边缘为四个狭窄的、固定宽的控件，在中间为一个大的区域。四条边分别称为南、北、西、东。中间部分称为中。下面里是由BorderLayout所定义的构造函数：

```
BorderLayout( )
BorderLayout(int horz, int vert)
```

第一种形式生成了默认的边界布局管理器。第二种形式允许你分别在horz和vert中设定组件间的水平和垂直距离。

BorderLayout定义了下列常数以指定区域：

BorderLayout.CENTER	BorderLayout.SOUTH
BorderLayout.EAST	BorderLayout.WEST
BorderLayout.NORTH	

当加入组件时，你将在add()方法中使用这些常数，add()方法由Container定义：

```
void add(Component compObj, Object region);
```

这里，compObj是被加入的组件，并用region指定组件被加入的位置。

下面是一个关于BorderLayout的例子，在每个布局区域都有一个组件：

```
// Demonstrate BorderLayout.
import java.awt.*;
import java.applet.*;
import java.util.*;
/*
<applet code="BorderLayoutDemo" width=400 height=200>
</applet>
*/

public class BorderLayoutDemo extends applet {
```

```
  public void init() {
    setLayout(new BorderLayout());

    add(new Button("This is across the top."),
        BorderLayout.NORTH);
    add(new Label("The footer message might go here."),
        BorderLayout.SOUTH);
    add(new Button("Right"), BorderLayout.EAST);
    add(new Button("Left"),  BorderLayout.WEST);
    String msg = "The reasonable man adapts " +
      "himself to the world;\n" +
      "the unreasonable one persists in " +
      "trying to adapt the world to himself.\n" +
      "Therefore all progress depends " +
      "on the unreasonable man.\n\n" +
      "       - George Bernard Shaw\n\n";

    add(new TextArea(msg), BorderLayout.CENTER);
  }
}
```

BorderLayputDemo小应用程序的输出如下所示。

22.11.3　使用间隔

有时，在装组件的容器和包含该容器的窗口之间留出一定的空间。这时，就要重载由Container所定义的getInsets()方法。此函数当容器显示时，返回一个包含顶部、底部、左部和右部间隔的对象。Insets的构造函数如下所示：

```
Insets(int top, int left, int bottom, int right)
```

top，left，bottom，和right设定了容器和它的封装窗口之间的空间。

getInsets()方法的基本形式如下：

```
Insets getInsets( )
```

当重载这些方法之一时，你必须返回一个新的包含你所要求的间隔空间的Insets对象。

这里给出的是将前面的BorderLayout 进行修改后的示例，它就能在它的控件与每一个边界之间留出10个像素的空间。背景颜色被设为青色，以便间隔清晰可见。

```
// Demonstrate BorderLayout with insets.
import java.awt.*;
import java.applet.*;
import java.util.*;
/*
<applet code="InsetsDemo" width=400 height=200>
</applet>
*/

public class InsetsDemo extends applet {
  public void init() {
    // set background color so insets can be easily seen
    setBackground(Color.cyan);

    setLayout(new BorderLayout());

    add(new Button("This is across the top."),
        BorderLayout.NORTH);
    add(new Label("The footer message might go here."),
        BorderLayout.SOUTH);
    add(new Button("Right"), BorderLayout.EAST);
    add(new Button("Left"), BorderLayout.WEST);

    String msg = "The reasonable man adapts " +
      "himself to the world;\n" +
      "the unreasonable one persists in " +
      "trying to adapt the world to himself.\n" +
      "Therefore all progress depends " +
      "on the unreasonable man.\n\n" +
      "       - George Bernard Shaw\n\n";

    add(new TextArea(msg), BorderLayout.CENTER);
  }
  // add insets
  public Insets getInsets() {
    return new Insets(10, 10, 10, 10);
  }
}
```

InsetsDemo小应用程序的输出如下所示。

22.11.4　GridLayout

网格布局管理器（GridLayout）在一个二维的网格中布置组件。当你实例化一个网格布局管理器时，需要定义行数和列数。由网格布局管理器所支持的构造函数如下所示：

```
GridLayout( )
GridLayout(int numRows, int numColumns )
GridLayout(int numRows, int numColumns, int horz, int vert)
```

第一个形式生成了一个单列的网格布局管理器。第二种形式生成了一个设定了行数与列数的布局管理器。第三种形式允许你在参数horz和vert中分别设定组件之间的水平和垂直间隔。参数numRows或numColumns其中之一可为零。指定参数numRows为零则允许使用无限长度的列。设定参数numColumns为零，则允许使用无限长度的行。

下面例子生成4×4网格，并以15个按钮填充，每个按钮以它的索引为标签。

```
// Demonstrate GridLayout
import java.awt.*;
import java.applet.*;
/*
<applet code="GridLayoutDemo" width=300 height=200>
</applet>
*/

public class GridLayoutDemo extends applet {
  static final int n = 4;
  public void init() {
    setLayout(new GridLayout(n, n));

    setFont(new Font("SansSerif", Font.BOLD, 24));

    for(int i = 0; i < n; i++) {
      for(int j = 0; j < n; j++) {
        int k = i * n + j;
        if(k > 0)
```

```
            add(new Button("" + k));

        }
      }
    }
  }
```

下面是layoutDemo小应用程序的输出：

提示：你可以使用这个例子作为一个15个正方形迷宫的起点。

22.11.5 CardLayout

CardLayout（卡片布局）类在其他的布局管理器中是独一无二的，因为它存储了几个不同的布局管理器。每个布局管理器都可被看作是在一副可洗的牌中的具有单独索引的牌，这样在一个给定的时间总会有一个纸牌在顶层。当用户与可选的组件交互时这会很有用，这些组件能根据用户的输入被动态启用或禁用。你可以准备其他的布局管理器并将它们隐藏，以便在需要时被激活。

CardLayout提供了两种构造函数：

```
CardLayout( )
CardLayout(int horz, int vert)
```

第一种形式生成了一个默认的卡片布局管理器。第二种形式分别在horz和vert中设定控件之间的水平和垂直间隔。

使用卡片布局管理器比其他的布局管理器需要多做一些工作。卡片一般来说存放在Panel类的一个对象中。这个面板必须使用卡片布局管理器作为它的布局管理器。形成纸牌的卡片通常也是Panel类的对象。这样，你必须生成一个包含一副牌的面板，以及这副牌中的每张牌的面板。然后，你向适当的面板加入形成每张牌的组件。接着，向以卡片布局管理器为布局管理器的面板加入这些面板。最后，你将这个面板加入主小应用程序面板。这些步骤完成后，你必须给用户提供在卡片之间进行选择的方法。一个常用的方法是一副牌中的每张卡片都包含一个按钮。

当卡片面板被加入到一个面板时，它们通常被给定一个名字。这样，大多数时间里，

你在将卡片加入面板时都要使用add()方法的以下形式：

```
void add(Component panelObj, Object name);
```

在这里，参数name是指定卡片名字的字符串，卡片的面板由panelObj设定。

在生成一副牌之后，你的程序通过调用由CardLayout定义的下列方法之一来激活一张卡片：

```
void first(Container deck)
void last(Container deck)
void next(Container deck)
void previous(Container deck)
void show(Container deck, String cardName)
```

这里，deck是盛卡片的容器的一个引用（通常是一个面板），cardName是一张卡片的名字。

调用first()方法使得deck中的第一张牌被显示；若显示最后一张，调用last()；显示下一张时，调用next()；显示前一张时，调用previous()。next()和previous()都会自动的分别循环回到组的最顶部及最底部。show()方法显示了以cardName命名的卡片。

下面的例子生成了一个两级的牌，它允许用户选择操作系统。基于视窗的操作系统被显示在一张卡片中，Macintosh和Solaris显示在另一张卡片中。

```
// Demonstrate CardLayout.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="CardLayoutDemo" width=300 height=100>
  </applet>
*/

public class CardLayoutDemo extends applet
  implements ActionListener, MouseListener {

  Checkbox Win98, winNT, solaris, mac;
  Panel osCards;
  CardLayout cardLO;
  Button Win, Other;

  public void init() {
    Win = new Button("Windows");
    Other = new Button("Other");
    add(Win);
    add(Other);

    cardLO = new CardLayout();
    osCards = new Panel();
    osCards.setLayout(cardLO); // set panel layout to card layout

    Win98 = new Checkbox("Windows 98", null, true);
    winNT = new Checkbox("Windows NT/2000");
    solaris = new Checkbox("Solaris");
```

```
    mac = new Checkbox("MacOS");


    // add Windows check boxes to a panel
    Panel winPan = new Panel();
    winPan.add(Win98);
    winPan.add(winNT);

    // Add other OS check boxes to a panel
    Panel otherPan = new Panel();
    otherPan.add(solaris);
    otherPan.add(mac);

    // add panels to card deck panel
    osCards.add(winPan, "Windows");
    osCards.add(otherPan, "Other");

    // add cards to main applet panel
    add(osCards);

    // register to receive action events
    Win.addActionListener(this);
    Other.addActionListener(this);

    // register mouse events
    addMouseListener(this);
  }

  // Cycle through panels.
 public void mousePressed(MouseEvent me) {
   cardLO.next(osCards);
  }
  // Provide empty implementations for the other MouseListener methods.

  public void mouseClicked(MouseEvent me) {
  }
  public void mouseEntered(MouseEvent me) {
  }
  public void mouseExited(MouseEvent me) {
  }
  public void mouseReleased(MouseEvent me) {
  }

  public void actionPerformed(ActionEvent ae) {
    if(ae.getSource() == Win) {

      cardLO.show(osCards, "Windows");
}
    else {
      cardLO.show(osCards, "Other");
    }
  }
}
```

下面是CardLayoutDemo小应用程序的输出。每张卡片通过按下其按钮来激活。你也能

够通过点击鼠标遍历每张卡片。

22.12　菜单栏与菜单

一个顶级窗口可能会使用菜单栏。一个菜单栏显示了一系列的顶级菜单选项。每个选项都与一个下拉式菜单相联。这种概念由Java中的下列类实现：MenuBar, Menu和MenuItem。一般来说，一个菜单栏包含了一个或更多的Menu对象。每个Menu对象都包含了一系列的MenuItem对象。每个MenuItem对象都代表了可被用户选择的某种东西。由于Menu是MenuItem的子类，将生成一个嵌套的子菜单的层次结构。包含可复选的菜单项的菜单也是可能的，CheckboxMenuItem类的菜单选项在它们被选中时会有一个复选标记在旁边。

为创建一个菜单栏，首先要创建MenuBar的一个实例。此类仅仅定义了默认的构造函数。接下来，还应该创建定义菜单栏显示选项的Menu的实例。下面是Menu的构造函数：

```
Menu( )
Menu(String optionName)
Menu(String optionName, boolean removable)
```

这里，optionName设定了菜单选项的名字。如果removable为true，则弹出式菜单可被移开并自由漂动。否则，它将被附着在菜单栏上（可移动菜单是与实现有关的）。第一种形式创建一个空菜单。

每一个菜单项都属于MenuItem类。它定义了下列构造函数：

```
MenuItem( )
MenuItem(String itemName)
MenuItem(String itemName, MenuShortcut keyAccel)
```

这里，参数itemName是菜单中显示的名字，参数keyAccel是此项的菜单快捷键。

可以使用setEnabled()方法来启用或禁用一个菜单选项。它的形式如下所示：

```
void setEnabled(boolean enabledFlag)
```

如果参数enabledFlag为true，菜单选项就被启用；如果为false，菜单选项就被禁用。

可以通过调用isEnabled()方法判定一个选项的状态。方法如下所示：

```
boolean isEnabled( )
```

如果被调用的菜单选项被启用，则isEnabled()返回true；否则，返回false。

可以通过调用setLabel()方法改变一个菜单选项的名字。可以通过使用getLabel()方法查看当前菜单项的名字。这些方法如下所示：

```
void setLabel(String newName)
String getLabel( )
```

这里，参数newName成为调用它的菜单选项的新名字。getLabel()返回当前菜单项的名字。

可以通过使用CheckboxMenuItem的MenuItem的子类来生成一个可选的菜单选项。它有下列构造函数：

```
CheckboxMenuItem( )
CheckboxMenuItem(String itemName)
CheckboxMenuItem(String itemName, boolean on)
```

在这里，参数itemName是菜单中显示的名字。每当一项被选时，它的状态就改变。在前两种形式中，可复选的项是未被选中的。第三种形式中，如果on为true，可复选项的初始状态为被选中；否则，则被清除选中。

你能通过调用getState()方法获得可选项的状态。通过使用setState()方法，能够将它设为一个已知的状态。这些方法如下所示：

```
boolean getState( )
void setState(boolean checked)
```

如果项被选中，则getState()返回true；否则，它返回false。为了选中一项，将true传给setState()方法。为了清除一项，则传递false。

一旦你生成了一个菜单选项，必须用add()方法将它加入Menu对象。Add()方法有下列一般形式：

```
MenuItem add(MenuItem item)
```

这里，参数item是被加入的项。选项以add()方法调用的顺序被加入菜单，item被返回。

一旦你已将所有选项加入Menu对象，你就能用由MenuBar()定义的add()方法的以下形式将这个对象加入菜单栏：

```
Menu add(Menu menu)
```

这里，参数menu是被加入的菜单，Menu被返回。

菜单仅在MenuItem类或CheckboxMenuItem类的选项被选中时才会产生事件。例如，当访问一个菜单栏显示向下弹出的菜单时，它们就不会产生事件。每当一个菜单选项被选中时，一个ActionEvent对象就被产生。每当一个复选菜单项被选中或被取消选中时，一个ItemEvent对象就被产生。这样，你必须实现ActionListener和ItemEvent接口以处理这些菜单

事件。

ItemEvent的getItem()方法对产生事件的选项返回一个引用，该方法的一般形式如下所示：

```
Object getItem()
```

下面是一个例子，它将一系列嵌入式的菜单加入一个弹出式窗口。所选定的选项，也显示了两个复选菜单选项的状态：

```
// Illustrate menus.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="MenuDemo" width=250 height=250>
  </applet>
*/

// Create a subclass of Frame
class MenuFrame extends Frame {
  String msg = "";
  CheckboxMenuItem debug, test;

  MenuFrame(String title) {
    super(title);

    // create menu bar and add it to frame
    MenuBar mbar = new MenuBar();
    setMenuBar(mbar);

    // create the menu items
    Menu file = new Menu("File");

    MenuItem item1, item2, item3, item4, item5;
    file.add(item1 = new MenuItem("New..."));
    file.add(item2 = new MenuItem("Open..."));
    file.add(item3 = new MenuItem("Close"));
    file.add(item4 = new MenuItem("-"));
    file.add(item5 = new MenuItem("Quit..."));
    mbar.add(file);


    Menu edit = new Menu("Edit");
    MenuItem item6, item7, item8, item9;
    edit.add(item6 = new MenuItem("Cut"));
    edit.add(item7 = new MenuItem("Copy"));

    edit.add(item8 = new MenuItem("Paste"));
    edit.add(item9 = new MenuItem("-"));
    Menu sub = new Menu("Special");
    MenuItem item10, item11, item12;
    sub.add(item10 = new MenuItem("First"));

    sub.add(item11 = new MenuItem("Second"));
```

```
    sub.add(item12 = new MenuItem("Third"));
    edit.add(sub);

    // these are checkable menu items
    debug = new CheckboxMenuItem("Debug");
    edit.add(debug);
    test = new CheckboxMenuItem("Testing");
    edit.add(test);

    mbar.add(edit);

    // create an object to handle action and item events
    MyMenuHandler handler = new MyMenuHandler(this);
    // register it to receive those events
    item1.addActionListener(handler);

    item2.addActionListener(handler);
    item3.addActionListener(handler);
    item4.addActionListener(handler);
    item5.addActionListener(handler);
    item6.addActionListener(handler);
    item7.addActionListener(handler);
    item8.addActionListener(handler);
    item9.addActionListener(handler);

    item10.addActionListener(handler);
    item11.addActionListener(handler);
    item12.addActionListener(handler);
    debug.addItemListener(handler);
    test.addItemListener(handler);

    // create an object to handle window events
    MyWindowAdapter adapter = new MyWindowAdapter(this);
     // register it to receive those events
    addWindowListener(adapter);
  }

  public void paint(Graphics g) {
    g.drawString(msg, 10, 200);


   if(debug.getState())
     g.drawString("Debug is on.", 10, 220);
else
     g.drawString("Debug is off.", 10, 220);

    if(test.getState())
      g.drawString("Testing is on.", 10, 240);
    else
      g.drawString("Testing is off.", 10, 240);
  }
}

class MyWindowAdapter extends WindowAdapter {
  MenuFrame menuFrame;
```

```
  public MyWindowAdapter(MenuFrame menuFrame) {
    this.menuFrame = menuFrame;
  }
  public void windowClosing(WindowEvent we) {
    menuFrame.setVisible(false);
  }
}

class MyMenuHandler implements ActionListener, ItemListener {
  MenuFrame menuFrame;
  public MyMenuHandler(MenuFrame menuFrame) {
    this.menuFrame = menuFrame;
  }
  // Handle action events
  public void actionPerformed(ActionEvent ae) {
      String msg = "You selected ";
      String arg = (String)ae.getActionCommand();
      if(arg.equals("New..."))
        msg += "New.";
      else if(arg.equals("Open..."))
        msg += "Open.";
      else if(arg.equals("Close"))
        msg += "Close.";
      else if(arg.equals("Quit..."))
        msg += "Quit.";
      else if(arg.equals("Edit"))
        msg += "Edit.";
      else if(arg.equals("Cut"))
        msg += "Cut.";
      else if(arg.equals("Copy"))
        msg += "Copy.";
      else if(arg.equals("Paste"))
        msg += "Paste.";
      else if(arg.equals("First"))

        msg += "First.";
      else if(arg.equals("Second"))
        msg += "Second.";
      else if(arg.equals("Third"))
        msg += "Third.";
      else if(arg.equals("Debug"))
        msg += "Debug.";
      else if(arg.equals("Testing"))
        msg += "Testing.";

      menuFrame.msg = msg;
      menuFrame.repaint();
  }
  // Handle item events
  public void itemStateChanged(ItemEvent ie) {
    menuFrame.repaint();
  }
}

// Create frame window.
```

```
public class MenuDemo extends applet {
  Frame f;

  public void init() {
    f = new MenuFrame("Menu Demo");
    int width = Integer.parseInt(getParameter("width"));
    int height = Integer.parseInt(getParameter("height"));

    setSize(new Dimension(width, height));

    f.setSize(width, height);
    f.setVisible(true);
  }

  public void start() {
    f.setVisible(true);
  }

  public void stop() {
    f.setVisible(false);
  }
}
```

MenuDemo小应用程序的输出如图22-8所示。

图 22-8 MenuDemo 小应用程序的输出

还有另外一个与菜单相关的类，也许你会感兴趣：PopupMenu。它的功能与Menu一样，但产生一个能显示在特定位置的菜单。PopupMenu为某些类型的类型提供了灵活有用的方法。

22.13 对 话 框

通常，你需要使用对话框（Dialog box）来存放一系列相关的控件。对话框主要被用来获得用户输入。它们与框架窗口相似，惟一的差别在于对话框总是一个顶级窗口的子窗口，

其中不能有菜单栏。在其他方面，对话框与框架窗口类似，例如，你能用与将控件加入框架窗口同样的方法将控件加入对话框。对话框可以是模式化的或非模式的。当一个模式化的对话框处在激活状态时，所有的输入都直接给它，直到它被关闭。这意味着在你关闭对话框之前，不能访问程序的其他部分。当一个非模式的对话框被激活时，输入焦点可以直接转移到你的程序的其他窗口。这样，你的程序的其他部分仍处于激活状态并可访问。对话框属于Dialog类。最常用的构造函数如下所示：

```
Dialog(Frame parentWindow, boolean mode)
Dialog(Frame parentWindow, String title, boolean mode)
```

这里，parentWindow是对话框的所有者。如果mode为true，对话框是模式的；否则，它是非模式的。对话框的标题可由title传递。一般来说，你将生成Dialog的子类，为你的应用程序增添所需的功能。

下面是将先前的菜单程序修改后的一个版本，当New选项被选中时，它显示了一个非模式的对话框。注意，当该对话框被关闭时，调用了dispose()方法。此方法由Window所定义，它将释放所有与对话框相关的系统资源。

```
// Demonstrate Dialog box.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="DialogDemo" width=250 height=250>
  </applet>

*/

// Create a subclass of Dialog.
class SampleDialog extends Dialog implements ActionListener {
  SampleDialog(Frame parent, String title) {
    super(parent, title, false);
    setLayout(new FlowLayout());
    setSize(300, 200);

    add(new Label("Press this button:"));
    Button b;
    add(b = new Button("Cancel"));
    b.addActionListener(this);
  }

  public void actionPerformed(ActionEvent ae) {
     dispose();

  }

  public void paint(Graphics g) {
    g.drawString("This is in the dialog box", 10, 70);
  }
}

// Create a subclass of Frame.
```

```
class MenuFrame extends Frame {
  String msg = "";
  CheckboxMenuItem debug, test;

  MenuFrame(String title) {
    super(title);

    // create menu bar and add it to frame
    MenuBar mbar = new MenuBar();
    setMenuBar(mbar);

    // create the menu items
    Menu file = new Menu("File");
    MenuItem item1, item2, item3, item4;
    file.add(item1 = new MenuItem("New..."));
    file.add(item2 = new MenuItem("Open..."));
    file.add(item3 = new MenuItem("Close"));
    file.add(new MenuItem("-"));
    file.add(item4 = new MenuItem("Quit..."));
    mbar.add(file);

    Menu edit = new Menu("Edit");
    MenuItem item5, item6, item7;
    edit.add(item5 = new MenuItem("Cut"));
    edit.add(item6 = new MenuItem("Copy"));
    edit.add(item7 = new MenuItem("Paste"));
    edit.add(new MenuItem("-"));

    Menu sub = new Menu("Special", true);
    MenuItem item8, item9, item10;
    sub.add(item8 = new MenuItem("First"));
    sub.add(item9 = new MenuItem("Second"));
    sub.add(item10 = new MenuItem("Third"));
    edit.add(sub);

     // these are checkable menu items
    debug = new CheckboxMenuItem("Debug");
    edit.add(debug);
    test = new CheckboxMenuItem("Testing");
    edit.add(test);

    mbar.add(edit);

    // create an object to handle action and item events
    MyMenuHandler handler = new MyMenuHandler(this);
    // register it to receive those events
    item1.addActionListener(handler);
    item2.addActionListener(handler);
    item3.addActionListener(handler);
    item4.addActionListener(handler);
    item5.addActionListener(handler);
    item6.addActionListener(handler);
    item7.addActionListener(handler);
    item8.addActionListener(handler);
    item9.addActionListener(handler);
```

```
    item10.addActionListener(handler);
    debug.addItemListener(handler);
    test.addItemListener(handler);

    // create an object to handle window events
    MyWindowAdapter adapter = new MyWindowAdapter(this);
     // register it to receive those events
    addWindowListener(adapter);
  }
  public void paint(Graphics g) {
    g.drawString(msg, 10, 200);

    if(debug.getState())
      g.drawString("Debug is on.", 10, 220);
    else
      g.drawString("Debug is off.", 10, 220);

    if(test.getState())
      g.drawString("Testing is on.", 10, 240);
    else
    g.drawString("Testing is off.", 10, 240);
  }
}

class MyWindowAdapter extends WindowAdapter {
  MenuFrame menuFrame;
  public MyWindowAdapter(MenuFrame menuFrame) {
    this.menuFrame = menuFrame;
  }
  public void windowClosing(WindowEvent we) {
    menuFrame.dispose();
  }
}

class MyMenuHandler implements ActionListener, ItemListener {
  MenuFrame menuFrame;
  public MyMenuHandler(MenuFrame menuFrame) {
    this.menuFrame = menuFrame;
  }
  // Handle action events
  public void actionPerformed(ActionEvent ae) {
      String msg = "You selected ";
      String arg = (String)ae.getActionCommand();
     // Activate a dialog box when New is selected.
     if(arg.equals("New...")) {
       msg += "New.";
       SampleDialog d = new
         SampleDialog(menuFrame, "New Dialog Box");
       d.setVisible(true);
      }
      // Try defining other dialog boxes for these options.
      else if(arg.equals("Open..."))
       msg += "Open.";
      else if(arg.equals("Close"))
       msg += "Close.";
```

```
      else if(arg.equals("Quit..."))
        msg += "Quit.";
      else if(arg.equals("Edit"))
        msg += "Edit.";
      else if(arg.equals("Cut"))
        msg += "Cut.";
      else if(arg.equals("Copy"))
        msg += "Copy.";
      else if(arg.equals("Paste"))
        msg += "Paste.";
      else if(arg.equals("First"))
        msg += "First.";
      else if(arg.equals("Second"))
        msg += "Second.";
      else if(arg.equals("Third"))
        msg += "Third.";
      else if(arg.equals("Debug"))
        msg += "Debug.";
      else if(arg.equals("Testing"))
        msg += "Testing.";
      menuFrame.msg = msg;
      menuFrame.repaint();
  }
  public void itemStateChanged(ItemEvent ie) {
      menuFrame.repaint();
  }
}

// Create frame window.
public class DialogDemo extends applet {
  Frame f;

  public void init() {
    f = new MenuFrame("Menu Demo");
    int width = Integer.parseInt(getParameter("width"));
    int height = Integer.parseInt(getParameter("height"));

    setSize(width, height);

    f.setSize(width, height);
    f.setVisible(true);
  }

  public void start() {
    f.setVisible(true);
  }

  public void stop() {
    f.setVisible(false);
  }
}
```

下图是DiaglogDemo小应用程序的输出：

提示：你自己试一试为菜单的其他选项定义对话框。

22.14　文件对话框

Java提供了一个，允许用户指定文件的内置对话框。为创建一个文件对话框，需创建FileDiaglog类的实例，这可以显示文件对话框。通常，这是由操作系统提供的标准文件对话框。FileDialog提供了下面这些构造函数：

```
FileDialog(Frame parent, String boxName)
FileDialog(Frame parent, String boxName, int how)
FileDialog(Frame parent)
```

这里，parent是对话框的所有者，boxName是显示在对话框标题栏的名字。如果boxName被忽略，对话框的标题将为空。如果how为FileDialog.LOAD，那么对话框为读而选择一个文件。如果how为FileDialog.SAVE，对话框为写而选择文件。第三个构造函数创建一个为读而选择文件的对话框。

FileDialog()方法允许用户判定被选文件的名字和路径。这儿是两个例子：

```
String getDirectory( )
String getFile( )
```

这些方法分别返回目录和文件名。

下面的程序将激活标准文件对话框：

```
/* Demonstrate File Dialog box.

   This is an application, not an applet.
*/
import java.awt.*;
import java.awt.event.*;

// Create a subclass of Frame
```

```
class SampleFrame extends Frame {
  SampleFrame(String title) {
    super(title);
    // create an object to handle window events
    MyWindowAdapter adapter = new MyWindowAdapter(this);
    // register it to receive those events
    addWindowListener(adapter);
  }
}

class MyWindowAdapter extends WindowAdapter {
  SampleFrame sampleFrame;
  public MyWindowAdapter(SampleFrame sampleFrame) {
    this.sampleFrame = sampleFrame;
  }
  public void windowClosing(WindowEvent we) {
    sampleFrame.setVisible(false);
  }
}

// Create frame window.
class FileDialogDemo {
  public static void main(String args[]) {
    Frame f = new SampleFrame("File Dialog Demo");
    f.setVisible(true);
    f.setSize(100, 100);
    FileDialog fd = new FileDialog(f, "File Dialog");
    fd.setVisible(true);
  }
}
```

程序的输出如下所示：

22.15 扩展AWT组件进行事件处理

在我们结束学习AWT之前，还有一个话题要讨论：通过扩展AWT组件进行事件处理。授权事件模型在第20章中介绍过，目前为止本书中所有的程序都用到了这种设计。但是Java

也允许你通过生成AWT组件子类来处理事件。这样做允许你用与在Java的最初的1.0版本下进行事件处理大致相同的方法来处理事件。当然，这种技术是令人气馁的，因为它有与Java1.0的事件模型同样的缺点，其中主要一点是效率低。在这一节里讲述通过扩展AWT组件处理事件是为了保证内容的完整性。但是，此技术在本书的其他任何部分都未被用到。

为了扩展AWT组件，必须调用Component的enableEvent()方法。它的基本形式如下：

```
protected final void enableEvents(long eventMask)
```

参数eventMask是用来定义被传送给此组件的事件的一个位掩码。AWT类为生成这个掩码定义了int常数。其中几个如下所示：

ACTION_EVENT_MASK	ITEM_EVENT_MASK
ADJUSTMENT_EVENT_MASK	KEY_EVENT_MASK
COMPONENT_EVENT_MASK	MOUSE_EVENT_MASK
CONTAINER_EVENT_MASK	MOUSE_MOTION_EVENT_MASK
FOCUS_EVENT_MASK	TEXT_EVENT_MASK
INPUT_METHOD_EVENT_MASK	WINDOW_EVENT_MASK

为了处理事件，你必须从超类中重载合适的方法。表22-1列出了常用的方法及提供它们的类。

表 22-1　事件处理方法

类	处理方法
Button	processActionEvent( )
Checkbox	processItemEvent( )
CheckboxMenuItem	processItemEvent( )
Choice	processItemEvent( )
Component	processComponentEvent( ), processFocusEvent( ), processKeyEvent( ), processMouseEvent( ), processMouseMotionEvent( )
List	processActionEvent( ), processItemEvent( )
MenuItem	processActionEvent( )
Scrollbar	processAdjustmentEvent( )
TextComponent	processTextEvent( )

下面几节提供了如何扩展几种AWT组件的简单程序。

22.15.1　扩展按钮（Button）

下列程序创建了一个标签为“Test Button”的按钮。当按钮被按下时，字符串“action event:”，以及按钮被按次数被显示在小应用程序解释器或浏览器的状态栏上。

程序有一个名为ButtonDemo2的类，它扩展了小应用程序，定义了一个名为i的静态整

数变量并初始化为0。它记录了按钮按下的次数。Init()方法实例化了Mybutton 并将它加入小应用程序。

Mybutton是一个扩展Button的内部类。它的构造函数使用super来将按钮的标签传给超类的构造函数。它调用enableEvent()方法从而使动作事件能被这个对象接收。当一个动作事件产生时，processActionEvent()方法被调用。此方法在状态栏上显示一个字符串行并为超类调用processActionEvent()方法。由于Mybutton是一个内部类，它可以直接访问ButtonDemo2的showStatus()方法。

```
/*
* <applet code=ButtonDemo2 width=200 height=100>
* </applet>
*/
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class ButtonDemo2 extends applet {
  MyButton myButton;
  static int i = 0;
  public void init() {
    myButton = new MyButton("Test Button");
    add(myButton);
  }
  class MyButton extends Button {
    public MyButton(String label) {
      super(label);
      enableEvents(AWTEvent.ACTION_EVENT_MASK);
    }
    protected void processActionEvent(ActionEvent ae) {
      showStatus("action event: " + i++);
      super.processActionEvent(ae);
    }
  }
}
```

22.15.2 扩展复选框（Checkbox）

下面的程序生成了一个小应用程序，它显示三个标签分别为“Item1”，“Item2”和“Item3”的复选框。当一个复选框被选中或被取消选择时，在小应用程序解释器或浏览器的状态栏上显示一个包含该复选框的名字和状态的字符串。

程序有一个名为CheckboxDemo2的扩展小应用程序的顶级类。它的init()方法生成了三个MyCheckbox的实例并将它们加入小应用程序。扩展Checkbox的MyCheckbox是一个内部类。它的构造函数使用了super来将复选框的标签传给超类构造函数。它调用了enableEvents()。因此选项事件可被这个对象接收。当一个选项事件产生时，调用processItemEvent()。此方法在状态栏上显示了一个字符串，并为超类调用processItemEvent()。

```
/*
* <applet code=CheckboxDemo2 width=300 height=100>
* </applet>
```

```
*/
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class CheckboxDemo2 extends applet {
  MyCheckbox myCheckbox1, myCheckbox2, myCheckbox3;
  public void init() {
    myCheckbox1 = new MyCheckbox("Item 1");
    add(myCheckbox1);
    myCheckbox2 = new MyCheckbox("Item 2");
    add(myCheckbox2);
    myCheckbox3 = new MyCheckbox("Item 3");
    add(myCheckbox3);
  }
  class MyCheckbox extends Checkbox {
    public MyCheckbox(String label) {
      super(label);
      enableEvents(AWTEvent.ITEM_EVENT_MASK);
    }
    protected void processItemEvent(ItemEvent ie) {
      showStatus("Checkbox name/state: " + getLabel() +
                 "/" + getState());
      super.processItemEvent(ie);
    }
  }
}
```

22.15.3　扩展一个复选框组(Check box group)

下面的程序重写了前面的复选框示例程序，使多个复选框形成一个复选框组。任何时刻，只能有一个复选框被选中。

```
/*
* <applet code=CheckboxGroupDemo2 width=300 height=100>
* </applet>
*/
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class CheckboxGroupDemo2 extends applet {
  CheckboxGroup cbg;
  MyCheckbox myCheckbox1, myCheckbox2, myCheckbox3;
  public void init() {
    cbg = new CheckboxGroup();
    myCheckbox1 = new MyCheckbox("Item 1", cbg, true);
    add(myCheckbox1);
    myCheckbox2 = new MyCheckbox("Item 2", cbg, false);
    add(myCheckbox2);
    myCheckbox3 = new MyCheckbox("Item 3", cbg, false);
    add(myCheckbox3);
  }
  class MyCheckbox extends Checkbox {
```

```
    public MyCheckbox(String label, CheckboxGroup cbg,
                      boolean flag) {
      super(label, cbg, flag);
      enableEvents(AWTEvent.ITEM_EVENT_MASK);
    }
    protected void processItemEvent(ItemEvent ie) {
      showStatus("Checkbox name/state: " + getLabel() +
                 "/" + getState());
      super.processItemEvent(ie);
    }
  }
}
```

22.15.4 扩展选择框（choice）

下面的程序生成了显示选项标签为“Red”，“Green”和“Blue”的复选框的小应用程序。当一个选项被选时，在小应用程序解释器或浏览器的状态栏上显示一个包含颜色名字的字符串。

有一个名为ChoiceDemo2的顶级类扩展了小应用程序。它的init()方法生成了一个选择框元件并将它加入小应用程序。Mychoice是一个扩展Choice的内部类。它调用了enableEvent()方法，因此选项事件可被这个对象接收。当一个选项事件产生时，调用processItemEvent()，此方法在状态栏上显示了一个字符串并为超类调用processItemEvent()。

```
/*
* <applet code=ChoiceDemo2 width=300 height=100>
* </applet>
*/
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class ChoiceDemo2 extends applet {
  MyChoice choice;
  public void init() {
    choice = new MyChoice();
    choice.add("Red");
    choice.add("Green");
    choice.add("Blue");
    add(choice);
  }
  class MyChoice extends Choice {
    public MyChoice() {
      enableEvents(AWTEvent.ITEM_EVENT_MASK);
    }
    protected void processItemEvent(ItemEvent ie) {
      showStatus("Choice selection: " + getSelectedItem());
      super.processItemEvent(ie);
    }
  }
}
```

22.15.5　扩展列表框(List)

下面的程序修改了前面的例子，使用列表框而不是选择框菜单。有一个扩展小应用程序的名为ListDemo2的顶级类。它的init()方法生成了一个列表框元素，并将它加入小应用程序。MyList是一个扩展List的内部类。它调用enableEvent()方法因而动作事件和选项事件可被这个对象接收。当一个选项被选中或被取消选中时，调用processItemEvent()。当双击一个选项时，也调用processActionEvent()。两种方法显示了一个字符串并接着将控制权交给超类。

```
/*
* <applet code=ListDemo2 width=300 height=100>
* </applet>
*/
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class ListDemo2 extends applet {
  MyList list;
  public void init() {
    list = new MyList();
    list.add("Red");
    list.add("Green");
    list.add("Blue");
    add(list);
  }
  class MyList extends List {
    public MyList() {
      enableEvents(AWTEvent.ITEM_EVENT_MASK |
                   AWTEvent.ACTION_EVENT_MASK);
    }
    protected void processActionEvent(ActionEvent ae) {
      showStatus("Action event: " + ae.getActionCommand());
      super.processActionEvent(ae);
    }
    protected void processItemEvent(ItemEvent ie) {
      showStatus("Item event: " + getSelectedItem());
      super.processItemEvent(ie);
    }
  }
}
```

22.15.6　扩展滚动条(Scrollbar)

下面的程序生成了一个显示滚动条的小应用程序。当此控件被操作时，在小应用程序的解释器或浏览器的状态栏上显示一个字符串。此字符串包含了滚动条所代表的值。

有一个扩展小应用程序的名为ScrollbarDemo2的顶级类。它的init()方法生成了一个滚动条元素并将它加入小应用程序。MyScrollbar是一个扩展Scrollbar的内部类。它调用enableEvents()方法因而调整事件可被该对象接收。当操作滚动条时，调用processAdjustmentEvent( )，当选项被选中时，也调用processAdjustmentEvent( )。它显示了

一个字符串并将控制权交给超类。

```
/*
* <applet code=ScrollbarDemo2 width=300 height=100>
* </applet>
*/
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class ScrollbarDemo2 extends applet {
  MyScrollbar myScrollbar;
  public void init() {
    myScrollbar = new MyScrollbar(Scrollbar.HORIZONTAL,
                              0, 1, 0, 100);
    add(myScrollbar);
  }
  class MyScrollbar extends Scrollbar {
    public MyScrollbar(int style, int initial, int thumb,
                    int min, int max) {
      super(style, initial, thumb, min, max);
      enableEvents(AWTEvent.ADJUSTMENT_EVENT_MASK);
    }
    protected void processAdjustmentEvent(AdjustmentEvent ae) {
      showStatus("Adjustment event: " + ae.getValue());
      setValue(getValue());
      super.processAdjustmentEvent(ae);
    }
  }
}
```

22.16 开发控件、菜单和布局管理器

本章讨论了包含AWT控件的各个类、菜单、布局管理器。但是，AWT提供的是一个丰富的编程环境，你能用它进行自己的开发。下面是一些建议：

- 试一试在一个小应用程序面板中嵌入画布
- 开发FileDialog控件
- 用setBounds()方法尝试手工布置控件
- 在面板中嵌入控件以获得对布局管理器的更多控制
- 实现LayoutManager接口来创建你自己的布局管理器
- 开发PopupMenu

你对AWT组件懂得越多，你将对你的应用程序和小应用程序的外观、感觉和性能有越多的控制。

在下一章中，我们将介绍另一个AWT的类：Image。此类可用来支持画图和动画。

第23章　图　像

本章我们来学习AWT的Image类和java.awt.image包，它们为成像（对图像的显示和操作）提供了支持。一个image只是一个矩形的图形对象，而images是网页设计中的一个重要的组件。事实上，在NCSA（国际超级计算机应用中心）的Mosaic浏览器中包含<img>标记是促使网络从1993年以来蓬勃发展的原因。此标记用来将图像内嵌入超文本。Java发展了这个基本概念，允许图像受到程序控制。由于图像的重要性，Java为它提供了广泛的支持。

Images是Image类的对象，而Image类是java.awt包的一部分。Images由java.awt.image中的类对其进行操作。java.awt.image定义了一大批类和接口，一一对它们进行研究是不可能的，因此我们集中研究其中对成像起基础作用的那一部分。下面是将在本章里讨论的java.awt.image类：

CropImageFilter	MemoryImageSource
FilteredImageSource	PixelGrabber
ImageFilter	RGBImageFilter

这些是我们将用到的接口：

ImageConsumer	ImageObserver	ImageProducer

除此以外，我们还要讨论Media.Tracker类，它是java.awt的一部分。

23.1　文 件 格 式

最初，网页图像仅用GIF一种格式。这种GIF图像格式由CompuServe在1987年创建，从而使得图像能够在线浏览，因此它非常适合于Internet。每个GIF图像至多只能有256种颜色。这种局限使得主要的浏览器厂商在1995年增加了对JPEG图像的支持。JPEG格式是由一群专业摄影师为了存储全彩色光谱的连续色调的图像而创建的。这种图像，只要被正确的生成，不但能比由同一源图像编码生成的GIF图像更好的被压缩，而且具有更好的精度。在几乎所有的情况下，你不必关心或注意在你的程序中用了哪种格式。Java图像类可以抽象出接口之间的差异。

23.2　图像基本操作：创建，加载和显示

当你对图像进行操作时三种最常见的操作为：创建图像，加载图像和显示图像。在Java中，Image类用来指向内存中的图像以及那些必须从外部资源加载的图像。因此，Java为你提供了创建新的图像对象并加载它的方法，它也提供了显示图像的方法。让我们依次看一

看这些方法。

23.2.1 创建一个图像对象

你可能会期望用如下的工具创建一个内存图像(Memory Image)。

```
Image test = new Image(200, 100); // Error -- won't work
```

但并不是这样。因为图像必须最终被画在窗口中以便查看，而Image类没有关于它的环境的足够信息来为屏幕生成合适的数据格式。所以，java.awt的Component类有一个叫做createImage()的方法用来生成图像对象（记住所有的AWT组件都是Component类的子类，因此它们都支持该方法）。

CreateImage()方法有如下两种形式：

```
Image createImage(ImageProducer imgProd)
Image createImage(int width, int height)
```

第一种形式返回由imgProd产生的图像，imgProd是一个实现ImageProducer接口的类的对象（稍后我们将讨论producers）。第二种形式返回具有指定宽度和高度的空图像，如下例所示。

```
Canvas c = new Canvas();
Image test = c.createImage(200, 100);
```

上例生成了一个画布Canvas实例，然后调用createImage()方法来实际生成一个Image对象。这里，图像是空白的。以后你将会看到如何对它写数据。

23.2.2 加载一个图像

获得一个图像的另一种方法是加载图像。这通过使用由Applet类定义的getImage()方法来实现。它有以下形式：

```
Image getImage(URL url)
Image getImage(URL url, String imageName)
```

该方法的第一种形式将参数url所设定的路径下的图像装入一个Image类的对象并将它返回。第二种形式将参数url所设定的路径下的图像装入一个以参数imageName命名的Image类的对象并将它返回。

23.2.3 显示图像

只要你有一个图像，你就可以用drawImage()方法来显示它，drawImage()方法是Graphics类中的一员。它有几种形式，我们将要用到的一种如下所示：

```
boolean drawImage(Image imgObj, int left, int top, ImageObserver imgOb)
```

它显示了由imgObj所传递的图像，其左上角由left和top指定。imgOb是一个实现了ImageObserver接口的类的引用。这个接口由所有的AWT组件所实现。一个image observer是一个对象，它能够在图像被加载时对其进行监控。ImageObserver将在下一节讨论。

用getImage()方法和drawImage()方法很容易加载和显示图像。这里是一个加载和显示简单图像的小应用程序示例。文件seattle.jpg被加载，但你能随意用任何GIF或JPG文件来替代它（只要保证它与包含这个小应用程序的HTML文件在同一个目录下）。

```
/*
 * <applet code="SimpleImageLoad" width=248 height=146>
 *  <param name="img" value="seattle.jpg">
 * </applet>
 */
import java.awt.*;
import java.applet.*;

public class SimpleImageLoad extends Applet
{
  Image img;

  public void init() {
    img = getImage(getDocumentBase(), getParameter("img"));
  }

  public void paint(Graphics g) {
    g.drawImage(img, 0, 0, this);
  }
}
```

在init()方法中，img变量代表getImage()方法返回的图像。getImage()方法以getParameter()方法返回的字符串为图像的文件名。这个图像可从与getDocumentBase()的调用结果有关的URL中下载，这也是小应用程序所在的HTML页面的URL。方法getParameter（"img"）返回的文件名来自于小应用程序标记<param name="img" value="seattle.jpg">。除了速度稍慢以外，这与使用HTML标记<img src="seattle.jpg" width=248 height=146>是等效的。图23-1表示了运行程序所看到的结果。

图 23-1　SimpleImageLoad 程序的输出

当这个小应用程序运行时，它在init()方法中启动加载img。在屏幕上你能看到正从网络

上下载的图像，因为每当更多的图像数据到达时，Applet对ImageObserver接口的实现就会调用paint()方法。

用这种方法，只要图像被完整的加载，就能简单的在屏幕上立即显现。你可以一边用其他信息在屏幕上画图，同时用ImageObserver（我们接下来将要讨论）来监控图像的加载。如果你利用加载图像的时间去并行完成其他的事情，可能会更好。

23.3 ImageObserver

ImageObserver是一个接口，当图像被生成时用来接收消息。ImageObserver仅定义了一种方法：imageUpdate()。用一个图像监视器能允许你完成其他的操作，例如在你被告知下载进程时，显示进程消息或一个吸引人的屏幕。在图像正通过网络加载时，这种消息非常有用，在这种情况下，人们常常试图通过一个很慢的调制解调器来加载图像决不是好办法。

ImageUpdate()方法有以下所示的一般形式：

```
boolean imageUpdate(Image imgObj, int flags, int left, int top, int width,
int height)
```

这里，参数imgObj是被加载的图像，参数flags是表示更新状况的整数。四个整数，left，top，width和height代表一个矩形，它根据flags中所传递的不同值包含不同的数值。如果ImageUpdate()方法完成了加载，它将返回false，如果还有图像要处理，将返回true。

flags参数包含一个或更多的在ImageObserver接口中定义为静态变量的位标识。这些标识以及它们所提供的信息如表23-1中所列。

表 23-1 ImageUpdate()中定义的位标识

标识	意义
WIDTH	Width这个标识是很有用的，它表示了图像的宽度
HEIGHT	Height这个标识是很有用的，它表示了图像的高度
PROPERTIES	与图像相关的属性可以通过使用imgObg.getProperty()获得
SOMEBITS	用于画图的像素已经收到。参数left，right，width和height定义了包含新像素的矩形
FRAMEBITS	以前所画的一个多帧图像中的完整一帧已经收到。这个帧可以被显示。参数left，right，width和height没有使用
ALLBITS	图像已经被完成。参数left，right，width和height没有使用
ERROR	在异步跟踪一幅图像时发生了一个错误。图像未完成，不能显示。没有获得更多的图像信息。ABORT被设置，表示图像生成被异常中止
ABORT	在图像完成之前，一幅异步跟踪的图像异常中止。然而，如果没有发生错误，访问图像的任何数据都将重新开始生成图像

Applet类有一个为ImageObserver接口所实现的方法imageUpdate()，用来重画被加载的图像。你可以在你的类中覆盖此方法以改变它的操作。

这里是imageUpdate()的一个简单的例子：

```
public boolean imageUpdate(Image img, int flags,
                           int x, int y, int w, int h) {
  if ((flags & ALLBITS) == 0) {
    System.out.println("Still processing the image.");
    return true;
  } else {
    System.out.println("Done processing the image.");
    return false;
  }
}
```

23.3.1　ImageObserver示例

现在我们看一个实例，它重载imageUpdate()，生成SimpleImageLoad小应用程序的一个版本，它不会使屏幕闪烁的那么频繁。Applet中ImageUpdate()的默认的实现有几处问题。首先，每次新数据到来时它都要重画整个图像。这就造成了背景颜色与图像之间的闪动。其次，它使用了Applet.repaint()以使系统每十分之一秒左右重画图像一次。这就造成了一种急促，不均匀的感觉，图像好像是油画。最后，默认的实现对可能正常加载失败的图像一无所知。getImage()方法即使当设定图像并不存在时也能成功，许多Java程序员一开始会因此遭到失败。在imageUpdate()出现之前，你发现不了缺少图像。如果你使用imageUpdate()的默认实现，那么你决不会知道发生了什么。你的paint()方法在你调用g.drawImage时将什么也不作。

下面的例子改正了SimpleImageLoad程序代码中的所有三个问题。首先，它通过两个小的改动消除了闪动。它重载了update()方法，这样它能调用paint()方法而无须首先画背景颜色。背景通过init()方法中的setBackground()方法来设定，这样，初始颜色只需画一次。除此以外，它还使用了repaint()方法的一个实现，来设定一个矩形以确定需要重画的范围。系统将设定剪下的区域，这样矩形以外的部分都无需被重画。这就减少了重画引起的闪烁，改善了性能。

其次，它消除了对引入图像的急促的，不均匀的显示。其方法是在每次接到更新时都进行画图。这些更新是在逐行扫描的基础上进行，因此一个100像素高的图像将在被加载时被重画100次。注意这不是显示图像的最快的方法，仅是显示最平滑的方法。

最后，它控制了由于找不到想要的文件而造成的错误，其方法是为ABORT位检查flags参数。如果它被设定，那么，实例变量error就被设为true，接着就调用repaint()。修改paint()方法，从而在error为true时能够打印出一条以淡红色为背景的出错信息。

程序代码如下所示：

```
/*
 * <applet code="ObservedImageLoad" width=248 height=146>
 *  <param name="img" value="seattle.jpg">
 * </applet>
 */
import java.awt.*;
```

```
import java.applet.*;

public class ObservedImageLoad extends Applet {
  Image img;
  boolean error = false;
  String imgname;

  public void init() {
    setBackground(Color.blue);
    imgname = getParameter("img");
    img = getImage(getDocumentBase(), imgname);
  }

  public void paint(Graphics g) {
    if (error) {
      Dimension d = getSize();
      g.setColor(Color.red);
      g.fillRect(0, 0, d.width, d.height);
      g.setColor(Color.black);
      g.drawString("Image not found: " + imgname, 10, d.height/2);
    } else {
      g.drawImage(img, 0, 0, this);
    }
  }

  public void update(Graphics g) {
    paint(g);
  }

  public boolean imageUpdate(Image img, int flags,
                         int x, int y, int w, int h) {
    if ((flags & SOMEBITS) != 0) {  // new partial data
      repaint(x, y, w, h);         // paint new pixels
    } else if ((flags & ABORT) != 0) {
      error = true;               // file not found
      repaint();                 // paint whole applet
    }
    return (flags & (ALLBITS|ABORT)) == 0;
  }
}
```

图23-2给出了小应用程序运行时的两个单独的屏幕。上面的屏幕显示了被装载了一半的图像，下面的屏幕显示了在小应用程序中被敲错的文件名。

图 23-2　ObservedImageLoad 程序的输出

下面是imageUpdate( )的另一种实现。它直到图像完全被装入后，才重画到屏幕上。

```
public boolean imageUpdate(Image img, int flags,
                           int x, int y, int w, int h) {
  if ((flags & ALLBITS) != 0) {
    repaint();
  } else if ((flags & (ABORT|ERROR)) != 0) {
    error = true; // file not found
    repaint();
  }
  return (flags & (ALLBITS|ABORT|ERROR)) == 0;
}
```

23.4　双 缓 冲

图像不仅如我们刚才所示的能够存储图画，你还能用它们作为屏幕外的图形环境。这就允许你将任何包含文字和图形的图像传递给一个屏幕外的缓冲区，这样就可以在稍后的时间里显示它。这样做的好处是，图像可以只在它被完成时才被看到。画一个复杂的图像

可能需要几毫秒或更多的时间，在用户看来可能产生闪烁。这种闪烁使用户感到传递的图像要比实际上慢。使用画面外图像来减少闪动的方法叫做双缓冲，因为屏幕被看作像素的一个缓冲区，而屏幕外的图像被认为是第二个缓冲区，可以在那儿准备要显示的像素。

在本章前面，你看到了如何去生成一个空白的Image对象。现在你将会看到如何在这个图像上而不是在屏幕上画图。回想前几章我们曾讲过，这需要一个Graphics对象来使用任何一个Java的绘制方法。一种简易的方法是，你在一个图像上画图时所需要的Graphics对象可以通过getGraphics()方法来获得。这里是用来产生一个新图像的程序代码段，它能获得它的图片的上下文关系，并用红色像素填充整个图像。

```
Canvas c = new Canvas();
Image test = c.createImage(200, 100);
Graphics gc = test.getGraphics();
gc.setColor(Color.red);
gc.fillRect(0, 0, 200, 100);
```

一旦创建并填充了一个非屏幕图像，它是不可见的。为了真正显示这个图像，需调用drawImage()方法。下面示例绘制了一幅耗时的图像，以此证明在用户可察觉的画图时间里，使用双缓冲会使效果有所不同。

```
/*
 * <applet code=DoubleBuffer width=250 height=250>
 * </applet>
 */
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class DoubleBuffer extends Applet {
  int gap = 3;
  int mx, my;
  boolean flicker = true;
  Image buffer = null;
  int w, h;

  public void init() {
    Dimension d = getSize();
    w = d.width;
    h = d.height;
    buffer = createImage(w, h);
    addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseDragged(MouseEvent me) {
        mx = me.getX();
        my = me.getY();
        flicker = false;
        repaint();
      }
      public void mouseMoved(MouseEvent me) {
        mx = me.getX();
        my = me.getY();
        flicker = true;
        repaint();
```

```
      }
    });
  }

  public void paint(Graphics g) {
    Graphics screengc = null;

    if (!flicker) {
      screengc = g;
      g = buffer.getGraphics();
    }

    g.setColor(Color.blue);
    g.fillRect(0, 0, w, h);

    g.setColor(Color.red);
    for (int i=0; i<w; i+=gap)
      g.drawLine(i, 0, w-i, h);
    for (int i=0; i<h; i+=gap)
      g.drawLine(0, i, w, h-i);

    g.setColor(Color.black);
    g.drawString("Press mouse button to double buffer", 10, h/2);

    g.setColor(Color.yellow);
    g.fillOval(mx - gap, my - gap, gap*2+1, gap*2+1);

    if (!flicker) {
      screengc.drawImage(buffer, 0, 0, null);
    }
  }
  public void update(Graphics g) {
    paint(g);
  }
}
```

这个简单的小应用程序有一个复杂的paint方法，它用蓝色填充了背景，之后在上面画了红色的波纹。它在上面画了黑色的文字并以坐标为mx，my的点为圆心画了一个黄色的圆。MouseMoved()方法和MouseDragged()方法被重载以跟踪鼠标的位置。这些方法除了flicker逻辑变量的设定以外都是一样的。MouseMoved()将flicker设为true，而mouseDragged()将它设为false。这时调用repaint()方法在两种情况下有相同的效果：在鼠标移动而无按钮被按下时将flicker设为true，在鼠标移动并有按钮被按下时将flicker设为false。

在paint()方法被调用而flicker被设为true时，当它在屏幕上被执行时，我们可以看到每一个画图操作。按下鼠标键，paint()方法被调用而flicker被设为false，我们会看到一幅全然不同的图画。paint()方法将Graphics的引用g与访问在ini( )中所创建的画布之外的缓冲图形相交换。这样所有的画图操作都不可见。在paint()方法的最后，我们简单的调用drawImage()方法来一次性地显示所有这些画图方法的结果。

注意将null作为第四个参数传递给drawImage()方法是允许的。这个参数用来传递一个接收图像事件消息的ImageObserver对象。由于这不是一个从网络数据流产生的图像，我们

并不需要消息提示。图23-3左边的快照就是当按钮没有被按下时，小应用程序窗口看上去的样子。我们可以看到，当快照被拍下时图像正在被重画的过程中。右边的快照显示了当按钮被按下时，图像如何通过双缓冲区的办法保持完整和清晰的。

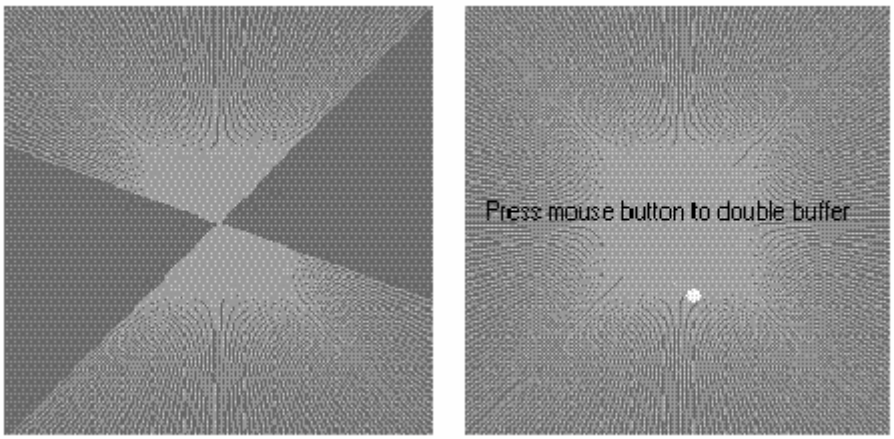


图 23-3 双缓冲区输出示意图（左图、右图分别是没有采用和采用双缓冲的示意图）

23.5 MediaTracker

许多早期的Java开发者发现，当有多媒体图像被加载时，ImageObserver接口太难于理解和掌握。开发者团体要求发明一种更简单的办法，允许程序员们同步加载他们所有的图像而不必担心imageUpdate()方法。应此要求，Sun 公司在JDK后来的版本中，在java.awt中增加了一个叫MediaTracker的类。一个MediaTracker对象能对任意数量的图像的状况进行并行检查。

使用MediaTracker时，你可以生成一个实例，用它的addImage()方法来跟踪图像的加载状况。addImage()方法有以下一般形式：

```
void addImage(Image imgObj, int imgID)
void addImage(Image imgObj, int imgID, int width, int height)
```

这里，参数imgObj是要被跟踪的图像。它的标识号由参数imgID传递。ID号并不需要是惟一的。你能用同一个数来标识几个图像作为将它们看作一个组的一种方法。在第二种形式中，参数width和height指定了对象显示时的尺寸。

一旦你已经注册了一个图像，就能检查它是否被装载，或等待被完全加载。检查图像的状况，调用checkID()方法。本章所用的版本如下所示：

```
boolean checkID(int imgID)
```

这里，imgID指定了你想要检查的图像的ID。这个方法在所有有此ID号的图像都已被加载时返回true（或当错误出现以及当用户中止了加载时）。否则，它将返回false。你可以用checkAll()方法来查看是否所有被跟踪的图像都已被加载。

当加载成批的图像时，你应该用MediaTracker。如果你所感兴趣的所有图像都没有被下载，你可以显示其他的东西以使用户乐意等待直到它们全部到达。

注意：如果你使用了MediaTracker，一旦你在图像中调用addImage()方法，MediaTracker中的引用就会阻止系统对它进行垃圾回收。如果你想要系统能够对被跟踪对图像进行垃圾回收，就要保证也能收集MediaTracker实例。

这里是一个装载7张图片幻灯片的例子，它用细条形表格显示了装载进程。

```
/*
 * <applet code="TrackedImageLoad" width=300 height=400>
 * <param name="img"
 * value="vincent+leonardo+matisse+picasso+renoir+seurat+vermeer">
 * </applet>
 */
import java.util.*;
import java.applet.*;
import java.awt.*;

public class TrackedImageLoad extends Applet implements Runnable {
  MediaTracker tracker;
  int tracked;
  int frame_rate = 5;
  int current_img = 0;
  Thread motor;
  static final int MAXIMAGES = 10;
  Image img[] = new Image[MAXIMAGES];
  String name[] = new String[MAXIMAGES];
  boolean stopFlag;

  public void init() {
    tracker = new MediaTracker(this);
    StringTokenizer st = new StringTokenizer(getParameter("img"),
                                        "+");

    while(st.hasMoreTokens() && tracked <= MAXIMAGES) {
      name[tracked] = st.nextToken();
      img[tracked] = getImage(getDocumentBase(),
                          name[tracked] + ".jpg");
      tracker.addImage(img[tracked], tracked);
      tracked++;
    }
  }

  public void paint(Graphics g) {
    String loaded = "";
    int donecount = 0;

    for(int i=0; i<tracked; i++) {
      if (tracker.checkID(i, true)) {
        donecount++;
        loaded += name[i] + " ";
      }
```

```
    }

    Dimension d = getSize();
    int w = d.width;
    int h = d.height;

    if (donecount == tracked) {
      frame_rate = 1;
      Image i = img[current_img++];
      int iw = i.getWidth(null);
      int ih = i.getHeight(null);
      g.drawImage(i, (w - iw)/2, (h - ih)/2, null);
      if (current_img >= tracked)
        current_img = 0;
    } else {
      int x = w * donecount / tracked;
      g.setColor(Color.black);
      g.fillRect(0, h/3, x, 16);
      g.setColor(Color.white);
      g.fillRect(x, h/3, w-x, 16);
      g.setColor(Color.black);
      g.drawString(loaded, 10, h/2);
    }
  }

  public void start() {
    motor = new Thread(this);
    stopFlag = false;
    motor.start();
  }

  public void stop() {
    stopFlag = true;
  }

  public void run() {
    motor.setPriority(Thread.MIN_PRIORITY);
    while (true) {
      repaint();
      try {
        Thread.sleep(1000/frame_rate);
      } catch (InterruptedException e) { };
      if(stopFlag)
        return;
    }
  }
}
```

这个例子在init()方法中生成了一个新的MediaTracker，然后用addImage()方法将每一个被命名的图像作为被跟踪图像加入。在paint()方法中，它对每一个我们跟踪的图像调用了checkID()方法。如果所有的图像都已加载，它们就显示。否则，显示一个简单的表示已加载图像数目的条形表格，在表格下面，显示了那些已被完全加载的图像的名字。图23-4显示了这个小应用程序运行的两种情景。一种是条形表，显示了三个图像已被加载。另一种

是演示过程中的Van Gogh的自画像。

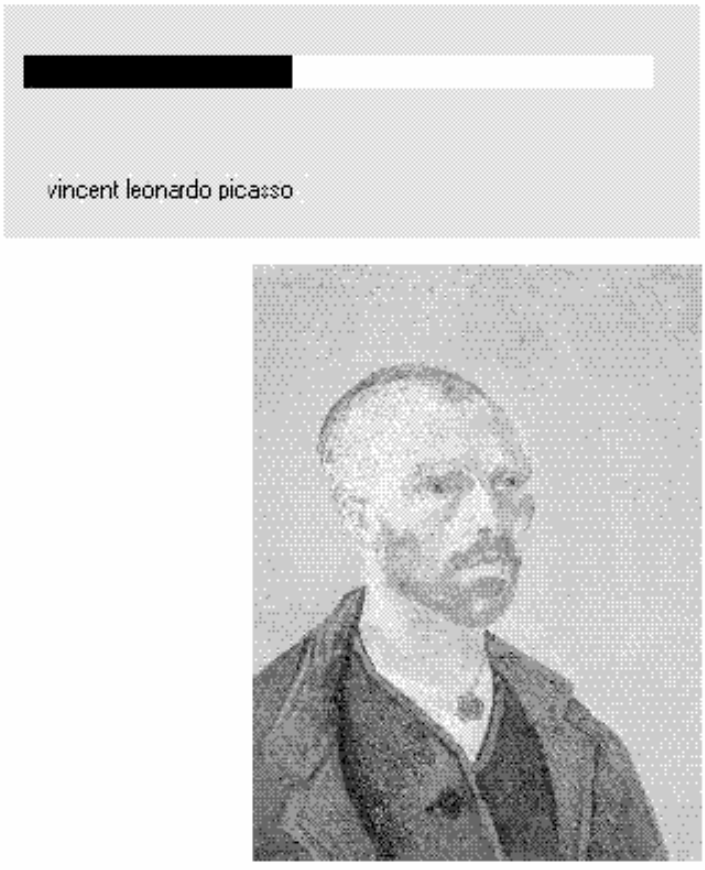


图 23-4　TrackedImageLoad 程序的输出

23.6　ImageProducer

ImagePruducer是一个用于生成图像数据对象的接口。一个实现该接口的对象将提供整数或字节类型的数组并生成Image对象。以前曾看到过，createImage()方法的一种形式是将一个ImageProducer对象作为它的参数。java.awt.image中包含两种图像生成器：MemoryImageSource和FilteredImageSource。这里，我们将研究MemoryImageSource并用小应用程序中产生的数据来生成一个新的Image对象。

23.6.1　MemoryImageSource

MemoryImageSource是用来从数据数组中生成新的图像的类。它定义了几种构造函数。这里是我们将用到的一种：

```
    MemoryImageSource(int width, int height, int pixel[ ], int offset, int
scanLineWidth)
```

MemoryImageSource对象是由整数数组构造出的，这些整数由pixel指定，在默认的RGB颜色模型中为Image对象生成数据。在默认的颜色模型中，一个像素是一个带有Alpha，Red，Green和Blue(0xAARRGGBB)的整数。Alpha值代表了像素的透明度，完全透明为0，完全不透明为255。生成图像的宽和高由width和height所设定。像素数组中最先被读的第一个点由offset所传递。扫描线（常与图像的宽相同）的宽由参数scanLineWidth所传递。

下面的小程序生成了一个MemoryImageSource对象，它使用了对一种简单算法进行改变后的算法（对每一点的x和y地址进行位异或），该算法来自于Prentice Hall于1988出版的Gerard J. Holzmann所著的《Beyond Photography, The Digital Darkroom》。

```
/*
 * <applet code="MemoryImageGenerator" width=256 height=256>
 * </applet>
 */
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

public class MemoryImageGenerator extends Applet {
  Image img;
  public void init() {
    Dimension d = getSize();
    int w = d.width;
    int h = d.height;
    int pixels[] = new int[w * h];
    int i = 0;

    for(int y=0; y<h; y++) {
      for(int x=0; x<w; x++) {
        int r = (x^y)&0xff;
        int g = (x*2^y*2)&0xff;
        int b = (x*4^y*4)&0xff;
        pixels[i++] = (255 << 24) | (r << 16) | (g << 8) | b;
      }
    }
    img = createImage(new MemoryImageSource(w, h, pixels, 0, w));
  }
  public void paint(Graphics g) {
    g.drawImage(img, 0, 0, this);
  }
}
```

MemoryImageSource所用的数据在init()方法中生成。生成一个整数数组以存储像素值；数据在嵌套的for循环中生成，在那儿r，g，b值被装入像素数组的相应像素值中。最后，调用createImage()方法，从作为参数的像素数据中生成一个MemoryImageSource的新的实例。图23-5显示了当我们运行小应用程序时所得到的图像。

图 23-5　MemoryImageGenerator 程序的输出

23.7　ImageConsumer

ImageConsumer是用于从图像中提取像素数据，并将这些数据以另一种数据类型方式表示出来的抽象接口。这与前面所描述的ImageProducer是显然相反的。一个实现ImageConsumer接口的对象将生成int和byte类型的数组，这些数组代表了一个Image对象的像素。我们将研究ImageConsumer接口的一个简单实现，PixelGrabber类。

23.7.1　PixelGrabber

PixelGrabber类是在java.lang.image中定义的。它与MemoryImageSource类相反。它并不从像素值数组中生成图像，它从现存的图像中提取像素数组。要使用PixelGrabber，首先需要生成一个足够大的int型的数组来存储像素数据，然后生成一个PixelGrabber实例，将你想要提取数据的对象放在一个矩形中传递给它。最后，在这个实例中调用grabPixels()方法。

本章用到的PixelGrabber构造函数如下所示：

```
PixelGrabber(Image imgObj, int left, int top, int width, int height,
             int pixel[ ],int offset, int scanLineWidth)
```

这里，参数imgObj要被提取像素的对象。left和top的值指定了矩形的左上角，width和height指定了矩形的尺寸，我们就是要从这个矩形中获得像素。这些像素将被存储在以offset开始的像素数组中。扫描线的宽（通常与图像的宽相同）由scanLineWidth所传递。

GrabPixels()方法的定义如下所示：

```
boolean grabPixels( )
    throws InterruptedException
```

```
boolean grabPixels(long milliseconds)
         throws InterruptedException
```

两种方法如果成功均返回true；否则返回false。在第二种形式中，参数milliseconds设定了方法将等待像素的时间是多长。

下面是一个示例，它从图像中抓取了像素，之后生成了像素亮度的柱状图。柱状图是对像素的亮度的一种简单计算，设定所有像素的亮度在0到255之间。在小应用程序画完图之后，它在其上绘柱状图。

```
/*
 * <applet code=HistoGrab.class width=341 height=400>
 * <param name=img value=vermeer.jpg>
 * </applet> */
import java.applet.*;
import java.awt.* ;
import java.awt.image.* ;

public class HistoGrab extends Applet {
  Dimension d;
  Image img;
  int iw, ih;
  int pixels[];
  int w, h;
  int hist[] = new int[256];
  int max_hist = 0;

  public void init() {
    d = getSize();
    w = d.width;
    h = d.height;

    try {
      img = getImage(getDocumentBase(), getParameter("img"));
      MediaTracker t = new MediaTracker(this);
      t.addImage(img, 0);
      t.waitForID(0);
      iw = img.getWidth(null);
      ih = img.getHeight(null);
      pixels = new int[iw * ih];
      PixelGrabber pg = new PixelGrabber(img, 0, 0, iw, ih,
                                         pixels, 0, iw);
      pg.grabPixels();
    } catch (InterruptedException e) { };

    for (int i=0; i<iw*ih; i++) {
      int p = pixels[i];
      int r = 0xff & (p >> 16);
      int g = 0xff & (p >> 8);
      int b = 0xff & (p);
      int y = (int) (.33 * r + .56 * g + .11 * b);
      hist[y]++;
    }
    for (int i=0; i<256; i++) {
```

```
        if (hist[i] > max_hist)
          max_hist = hist[i];
      }
    }

    public void update() {}

    public void paint(Graphics g) {
      g.drawImage(img, 0, 0, null);
      int x = (w - 256) / 2;
      int lasty = h - h * hist[0] / max_hist;
      for (int i=0; i<256; i++, x++) {
        int y = h - h * hist[i] / max_hist;
        g.setColor(new Color(i, i, i));
        g.fillRect(x, y, 1, h);
        g.setColor(Color.red);
        g.drawLine(x-1,lasty,x,y);
        lasty = y;
      }
    }
}
```

图23-6显示了内容为一幅著名的Vermeer油画的图像和柱状图。

图 23-6　HistoGrab 程序的输出

23.8　ImageFilter

有了ImageProducer和ImageConsumer这两个接口，和它们的具体的类MemoryImageSource和PixelGrabber，你就能生成任意一套转换过滤器从而提取像素，对它们进行修改并将它们传递给任意使用者。这种机制与具体的类从抽象的I/O类InputStream,

OutputStream，Reader和Writer（见第17章）中生成的方式是类似的。图像数据流的模型是由 ImageFilter 类的引入来完成的。在 java.awt.image 包中，ImageFilter 的子类有 AreaAveragingScaleFilter，CropImageFilter，ReplicateScaleFilter，和RGBImageFilter。还有 ImageProducer 的一个实现，叫做 FilterImageSource，能用任意 ImageFilter 并使它针对 ImageProducer来提取生成的像素。一个FilteredImageSource的实例能在对createImage的调用中被用作ImageProducer，这与BufferedInputStreams能被作为InputStreams采用的是几乎相同的方法。

23.8.1 CropImageFilter

CropImageFilter对象可以对一原始图像进行过滤，提取出一个矩形区域。在某种情况下这种过滤器是很有价值的，比如你想要使用一个更大的原始图像中的几个小图像。加载20个2K的图像要比加载一个40K的嵌有许多动画的图像需要长得多的时间。如果每个子图像都同样大小，只要小应用程序启动，那么你就能很容易的用CropImageFilter来分解块从而提取图像。下面是一个示例，它从一个图像中生成了16个图像。这些被提取出的图像通过任意的两两互换被重新拼凑32次了。

```
/*
 * <applet code=TileImage.class width=288 height=399>
 * <param name=img value=picasso.jpg>
 * </applet>
 */
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

public class TileImage extends Applet {
  Image img;
  Image cell[] = new Image[4*4];
  int iw, ih;
  int tw, th;

  public void init() {
    try {
      img = getImage(getDocumentBase(), getParameter("img"));
      MediaTracker t = new MediaTracker(this);
      t.addImage(img, 0);
      t.waitForID(0);
      iw = img.getWidth(null);
      ih = img.getHeight(null);
      tw = iw / 4;
      th = ih / 4;
      CropImageFilter f;
      FilteredImageSource fis;
      t = new MediaTracker(this);
      for (int y=0; y<4; y++) {
        for (int x=0; x<4; x++) {
          f = new CropImageFilter(tw*x, th*y, tw, th);
          fis = new FilteredImageSource(img.getSource(), f);
          int i = y*4+x;
```

```
          cell[i] = createImage(fis);
          t.addImage(cell[i], i);
        }
      }
      t.waitForAll();
      for (int i=0; i<32; i++) {
        int si = (int)(Math.random() * 16);
        int di = (int)(Math.random() * 16);
        Image tmp = cell[si];
        cell[si] = cell[di];
        cell[di] = tmp;
      }
    } catch (InterruptedException e) { };
  }

  public void update(Graphics g) {
    paint(g);
  }

  public void paint(Graphics g) {
    for (int y=0; y<4; y++) {
      for (int x=0; x<4; x++) {
        g.drawImage(cell[y*4+x], x * tw, y * th, null);
      }
    }
  }
}
```

图23-7显示了TileImage小应用程序拼成的一幅著名的Picasso的油画。

图 23-7　TileImage 程序的输出

23.8.2 RGBImageFilter

RGBImageFilter用来将一个图像逐个像素地改变颜色，从而转化为另一个图像。此过滤器可以用来增强图像的亮度，增加对比度，或将它转化为灰度。

为了说明RGBImageFilter，我们开发了一个稍复杂的示例程序，它在图像过滤过程中使用了动态嵌入的策略。我们曾为一般的图像过滤器生成了一个接口，这样我们的小应用程序就能够简单地在<param>标记的基础上加载这些过滤器，而无需提前了解所有的ImageFilters。这个例子包含了叫做ImageFilterDemo的小应用程序类，名为PlugInFilter的接口，和一个叫做LoadedImage的有用的类，此LoadedImage类装入了一些我们曾在本章用过的MediaTracker方法。除此以外还包含了三个能用RGBImageFilters对原始图像的颜色空间进行简单操作的过滤器，Grayscale，Invert和Contrast，以及两个做更复杂的“回旋”过滤的类，Blur和Sharpen，它们可以根据源数据中每个像素周围的其他像素的数据来改变这个像素的数据。Blur和Sharpen是一个名为Convolver的抽象类的子类。让我们来看一看例子的每一组成部分。

ImageFilterDemo.java

ImageFilterDemo类是标准图像过滤器的小应用程序框架。它用了一个简单的BorderLayout，在南边有一个Panel放置代表每一个过滤器的按钮。一个Label对象在北边的槽中用来显示有关过滤器进程的消息。中间是图像（它被装入LoadedImage Canvas的子类，我们接下来将会讨论）所在的地方。我们从filters <param>标记中分析了按钮/过滤器，使用StringTokenizer以＋'s将它们区分开。

actionPerformed( )方法非常有趣，因为它使用了来自于一个按钮的标签作为它想用(PlugInFilter) Class.forName(a).newInstance( )来装载的过滤器类的名字。这个方法十分强大，如果按钮不能与一个实现PlugInFilter的类相适应，它会采取适当的措施。

```
/*
 * <applet code=ImageFilterDemo width=350 height=450>
 * <param name=img value=vincent.jpg>
 * <param name=filters value="Grayscale+Invert+Contrast+Blur+ Sharpen">
 * </applet>
 */
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class ImageFilterDemo extends Applet implements ActionListener {
  Image img;
  PlugInFilter pif;
  Image fimg;
  Image curImg;
  LoadedImage lim;
  Label lab;
  Button reset;

  public void init() {
```

```
    setLayout(new BorderLayout());
    Panel p = new Panel();
    add(p, BorderLayout.SOUTH);
    reset = new Button("Reset");
    reset.addActionListener(this);
    p.add(reset);
    StringTokenizer st = new StringTokenizer(getParameter("filters"), "+");
    while(st.hasMoreTokens()) {
      Button b = new Button(st.nextToken());
      b.addActionListener(this);
      p.add(b);
    }

    lab = new Label("");
    add(lab, BorderLayout.NORTH);

    img = getImage(getDocumentBase(), getParameter("img"));
    lim = new LoadedImage(img);
    add(lim, BorderLayout.CENTER);

  }

  public void actionPerformed(ActionEvent ae) {
    String a = "";

    try {
      a = (String)ae.getActionCommand();
      if (a.equals("Reset")) {
        lim.set(img);
        lab.setText("Normal");
      }
      else {
        pif = (PlugInFilter) Class.forName(a).newInstance();
        fimg = pif.filter(this, img);
        lim.set(fimg);
        lab.setText("Filtered: " + a);
      }
      repaint();
    } catch (ClassNotFoundException e) {
      lab.setText(a + " not found");
      lim.set(img);
      repaint();
    } catch (InstantiationException e) {
      lab.setText("could't new " + a);
    } catch (IllegalAccessException e) {
      lab.setText("no access: " + a);
    }
  }
}
```

图23-8显示了小应用程序在用这个源文件头部所示的applet标记被第一次加载时的样子。

图 23-8 ImageFilterDemo 程序的正常输出显示

PlugInFilter.java

PlugInFilter是一个用来抽象图像过滤的接口。它仅有一种方法filter()，这个方法获得小应用程序和源图像并返回一个经过了某种过滤的新的图像。

```
interface PlugInFilter {
  java.awt.Image filter(java.applet.Applet a, java.awt.Image in);
}
```

LoadedImage.java

LoadedImage类是Canvas类的一个十分方便使用的子集，它在构造的时候获得图像并同时用MediaTracker加载它。由于LoadedImage重载了getPreferredSize( ) 和getMinimumSize( ) 方法，接着它在LayoutManager的控制下进行适当的动作。除此以外，它还有一个名为set() 的方法，用来在Canvas中设定一个要被显示的图像。这就是经过过滤的图像如何在嵌入完成后被显示的原理。

```
import java.awt.*;

public class LoadedImage extends Canvas {
  Image img;

  public LoadedImage(Image i) {
    set(i);
  }

  void set(Image i) {
    MediaTracker mt = new MediaTracker(this);
    mt.addImage(i, 0);
```

```
    try {
      mt.waitForAll();
    } catch (InterruptedException e) { };
    img = i;
    repaint();
  }

  public void paint(Graphics g) {
    if (img == null) {
      g.drawString("no image", 10, 30);
    } else {
      g.drawImage(img, 0, 0, this);
    }
  }

  public Dimension getPreferredSize()  {
    return new Dimension(img.getWidth(this), img.getHeight(this));
  }

  public Dimension getMinimumSize()  {
    return getPreferredSize();
  }
}
```

Grayscale.java

Grayscale过滤器是RGBImageFilter的一个子集，这意味着Grayscale能将自己作为FilteredImageSource的构造函数ImageFilter参数。然后它只需重载filterRGB()来改变引入的颜色值。它获取红，绿和蓝色的值并计算像素的亮度，在此过程中使用NTSC（National Television Standards Committee，国家电视标准委员会）的颜色亮度转换因子。之后，它仅返回一个与源像素具有同等亮度的灰色像素。

```
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

class Grayscale extends RGBImageFilter implements PlugInFilter {
  public Image filter(Applet a, Image in) {
    return a.createImage(new FilteredImageSource(in.getSource(), this));
  }

  public int filterRGB(int x, int y, int rgb) {
    int r = (rgb >> 16) & 0xff;
    int g = (rgb >> 8) & 0xff;
    int b = rgb & 0xff;
    int k = (int) (.56 * g + .33 * r + .11 * b);
    return (0xff000000 | k << 16 | k << 8 | k);
  }
}
```

invert.Java

Invert过滤器很简单。它将红，绿，蓝三色分成不同的通道，再用它们去减255，得到

的差作为新值。这些转换过的值被重新组合进一个像素值并被返回。

```
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

class Invert extends RGBImageFilter implements PlugInFilter {
  public Image filter(Applet a, Image in) {
    return a.createImage(new FilteredImageSource(in.getSource(), this));
  }
  public int filterRGB(int x, int y, int rgb) {
    int r = 0xff - (rgb >> 16) & 0xff;
    int g = 0xff - (rgb >> 8) & 0xff;
    int b = 0xff - rgb & 0xff;
    return (0xff000000 | r << 16 | g << 8 | b);
  }
}
```

图23-9显示了经过Invert过滤器转化后的图像。

图 23-9 ImageFilterDemo 经过 Invert 过滤器转化后的图像

Contrast.java

Contrast过滤器类似于Grayscale，除了它对filterRGB()的重载稍复杂以外。它所用到的增强对比度的算法是：分别获取红，绿，蓝三色的值，并将那些大于128的值增大1.2倍。这些被增大的值能通过multclamp()方法以255为界被适当地压缩。

```
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

public class Contrast extends RGBImageFilter implements PlugInFilter {
```

```
  public Image filter(Applet a, Image in) {
    return a.createImage(new FilteredImageSource(in.getSource(), this));
  }

  private int multclamp(int in, double factor) {
    in = (int) (in * factor);
    return in > 255 ? 255 : in;
  }

  double gain = 1.2;
  private int cont(int in) {
    return (in < 128) ? (int)(in/gain) : multclamp(in, gain);
  }

  public int filterRGB(int x, int y, int rgb) {
    int r = cont((rgb >> 16) & 0xff);
    int g = cont((rgb >> 8) & 0xff);
    int b = cont(rgb & 0xff);
    return (0xff000000 | r << 16 | g << 8 | b);
  }
}
```

图23-10显示了对比度被压缩后的图像。

图 23-10　ImageFilterDemo 经过对比度过滤器转化后的图像

Convolver.java

抽象类Convolver处理回旋过滤器的基本要素，它通过实现ImageConsumer接口将源像素移入一个称为imgpixel的数组中。它还能为过滤后的数据生成另一个称为newimgpixels的数组。Convolution过滤器以图像中每一个像素为回旋中心，抽取其周围的一个很小的矩形

范围的像素为样本，这称为covolution kernel。在此演示中，3×3像素的区域面积决定了此区域的中心像素如何变化。在下一节里介绍的两个具体的子类完全实现了convolve()方法，并使用imgpixel存储源数据，用newimgpixels存储结果。

注意：过滤器不能适当对imgpixels数组进行修改的原因是，扫描线上的下一个像素将尝试使用前一像素的原始值，而这一原始值可能已经过滤过了。

```
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

abstract class Convolver implements ImageConsumer, PlugInFilter {
  int width, height;
  int imgpixels[], newimgpixels[];

  abstract void convolve();  // filter goes here...

  public Image filter(Applet a, Image in) {
    in.getSource().startProduction(this);
    waitForImage();
    newimgpixels = new int[width*height];

    try {
      convolve();
    } catch (Exception e) {
      System.out.println("Convolver failed: " + e);
      e.printStackTrace();
    }

    return a.createImage(
      new MemoryImageSource(width, height, newimgpixels, 0, width));
  }

  synchronized void waitForImage() {
    try { wait(); } catch (Exception e) { };
  }

  public void setProperties(java.util.Hashtable dummy) { }
  public void setColorModel(ColorModel dummy) { }
  public void setHints(int dummy) { }

  public synchronized void imageComplete(int dummy) {
    notifyAll();
  }

  public void setDimensions(int x, int y) {
    width = x;
    height = y;
    imgpixels = new int[x*y];
  }

  public void setPixels(int x1, int y1, int w, int h,
    ColorModel model, byte pixels[], int off, int scansize) {
```

```
    int pix, x, y, x2, y2, sx, sy;

    x2 = x1+w;
    y2 = y1+h;
    sy = off;
    for(y=y1; y<y2; y++) {
      sx = sy;
      for(x=x1; x<x2; x++) {
        pix = model.getRGB(pixels[sx++]);
        if((pix & 0xff000000) == 0)
            pix = 0x00ffffff;
        imgpixels[y*width+x] = pix;
      }
      sy += scansize;
    }
  }

  public void setPixels(int x1, int y1, int w, int h,
    ColorModel model, int pixels[], int off, int scansize) {
    int pix, x, y, x2, y2, sx, sy;

    x2 = x1+w;
    y2 = y1+h;
    sy = off;
    for(y=y1; y<y2; y++) {
      sx = sy;
      for(x=x1; x<x2; x++) {
        pix = model.getRGB(pixels[sx++]);
        if((pix & 0xff000000) == 0)
            pix = 0x00ffffff;
        imgpixels[y*width+x] = pix;
      }
      sy += scansize;
    }
  }
}
```

Blur.Java

Blur过滤器是Convolver的一个子类，它完全在源图像数组imgpixels中对每一像素进行运算，算出它周围3×3像素区域的平均值。newimgpixels中相应的输出像素就是此平均值。

```
public class Blur extends Convolver {
  public void convolve() {
    for(int y=1; y<height-1; y++) {
      for(int x=1; x<width-1; x++) {
        int rs = 0;
        int gs = 0;
        int bs = 0;

        for(int k=-1; k<=1; k++) {
          for(int j=-1; j<=1; j++) {
            int rgb = imgpixels[(y+k)*width+x+j];
            int r = (rgb >> 16) & 0xff;
            int g = (rgb >> 8) & 0xff;
```

```
            int b = rgb & 0xff;
            rs += r;
            gs += g;
            bs += b;
          }
        }

        rs /= 9;
        gs /= 9;
        bs /= 9;

       newimgpixels[y*width+x] = (0xff000000 |
                              rs << 16 | gs << 8 | bs);
      }
    }
  }
}
```

图23-11显示了经过Blur过滤后的小应用程序。

图 23-11 ImageFilterDemo 经过 Blur 过滤后的图像

Sharpen.java

Sharpen过滤器也是Convolver的一个子类，与Blur或多或少相反。它对源图像数组imgpixels中的每一像素进行运算，算出它周围的3×3区域的平均值，但不计入中心像素本身。newimgpixels中相应的输出像素在中心像素和其周围的均值之间存在差异。这主要说明了，如果一个像素比它周围亮30倍，此运算会使它再亮30倍。如果，它比周围暗10倍，则使它再暗10倍。这往往使得边界更尖锐，而使平滑的区域不发生改变。

```
public class Sharpen extends Convolver {
```

```
  private final int clamp(int c) {
    return (c > 255 ? 255 : (c < 0 ? 0 : c));
  }

  public void convolve() {
    int r0=0, g0=0, b0=0;
    for(int y=1; y<height-1; y++) {
      for(int x=1; x<width-1; x++) {
        int rs = 0;
        int gs = 0;
        int bs = 0;

        for(int k=-1; k<=1; k++) {
          for(int j=-1; j<=1; j++) {
            int rgb = imgpixels[(y+k)*width+x+j];
            int r = (rgb >> 16) & 0xff;
            int g = (rgb >> 8) & 0xff;
            int b = rgb & 0xff;
            if (j == 0 && k == 0) {
              r0 = r;
              g0 = g;
              b0 = b;
            } else {
              rs += r;
              gs += g;
              bs += b;
            }
          }
        }

        rs >>= 3;
        gs >>= 3;
        bs >>= 3;
        newimgpixels[y*width+x] = (0xff000000 |
                              clamp(r0+r0-rs) << 16 |
                              clamp(g0+g0-gs) << 8 |
                              clamp(b0+b0-bs));
      }
    }
  }
}
```

图23-12是使用了Sharpen过滤后的小应用程序。

图 23-12 ImageFilterDemo 经过 Sharpen 过滤后的图像

23.9 单 元 动 画

现在我们已经对图像API有了一般的了解，我们可以用它们来实现一个有趣的小应用程序，这个小应用程序实现一系列动画的功能。这个动画中的单元来自于一个图片，通过用<param>标记传入的参数rows和cols作为行数和列数组成的网格把它分开。图像中的每一个单元都像在TileImage例子中一样地处理。我们从参数sequence获得了显示单元的序列。那是一个单元数的列表，它以0为基数，从左到右，从上到下的分布。

一旦这个小应用程序分析了<param>标记，并载入了源图像，它就将图像分割成一系列的子图像。然后，一个线程开始按照sequence参数确定的顺序来显示图像。这个线程为维持framerate而休眠足够多的时间。源程序如下所示：

```
// Animation example.
import java.applet.*;
import java.awt.*;
import java.awt.image.*;
import java.util.*;

public class Animation extends Applet implements Runnable {
  Image cell[];
  final int MAXSEQ = 64;
  int sequence[];
  int nseq;
  int idx;
  int framerate;
  boolean stopFlag;

  private int intDef(String s, int def) {
```

```
  int n = def;
  if (s != null)
    try {
      n = Integer.parseInt(s);
    } catch (NumberFormatException e) { };
  return n;
}

public void init() {
  framerate = intDef(getParameter("framerate"), 5);
  int tilex = intDef(getParameter("cols"), 1);
  int tiley = intDef(getParameter("rows"), 1);
  cell = new Image[tilex*tiley];

  StringTokenizer st = new
          StringTokenizer(getParameter("sequence"), ",");
  sequence = new int[MAXSEQ];
  nseq = 0;
  while(st.hasMoreTokens() && nseq < MAXSEQ) {
    sequence[nseq] = intDef(st.nextToken(), 0);
    nseq++;
  }

  try {
    Image img = getImage(getDocumentBase(), getParameter("img"));
    MediaTracker t = new MediaTracker(this);
    t.addImage(img, 0);
    t.waitForID(0);
    int iw = img.getWidth(null);
    int ih = img.getHeight(null);
    int tw = iw / tilex;
    int th = ih / tiley;
    CropImageFilter f;
    FilteredImageSource fis;
    for (int y=0; y<tiley; y++) {
      for (int x=0; x<tilex; x++) {
        f = new CropImageFilter(tw*x, th*y, tw, th);
        fis = new FilteredImageSource(img.getSource(), f);
        int i = y*tilex+x;
        cell[i] = createImage(fis);
        t.addImage(cell[i], i);
      }
    }
    t.waitForAll();
  } catch (InterruptedException e) { };
}

public void update(Graphics g) { }

public void paint(Graphics g) {
    g.drawImage(cell[sequence[idx]], 0, 0, null);
}

Thread t;
public void start() {
  t = new Thread(this);
  stopFlag = false;
  t.start();
```

```
  }

  public void stop() {
    stopFlag = true;
  }

  public void run() {
    idx = 0;
    while (true) {
      paint(getGraphics());
      idx = (idx + 1) % nseq;
      try { Thread.sleep(1000/framerate); } catch (Exception e) { };
            if(stopFlag)
        return;
    }
  }
}
```

接下来的小应用程序标记展示了通过Eadweard Muybridge来进行著名的运动研究，Eadweard Muybridge证明马跑的时候的确四蹄离地（当然，你可以在你的小应用程序中选择另一个图像）。

```
<applet code=Animation width=67 height=48>
<param name=img value=horse.gif>
<param name=rows value=3>
<param name=cols value=4>
<param name=sequence value=0,1,2,3,4,5,6,7,8,9,10>
<param name=framerate value=15>
</applet>
```

图23-13展示了这个小应用程序运行的结果。请注意源图像已经通过<img>标记显示在了小应用程序的下面。

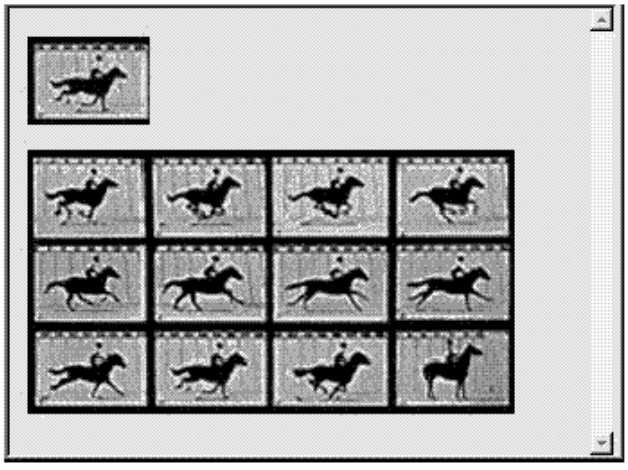


图 23-13 程序 Animation 的输出

23.10　其他Java 2图像类

除了在本章中描述的图像类以外，Java 2还支持其他的一些更强大的进行图像处理和高级图像技术的类。如果想了解更多的复杂图像输出，你可以在java.awt.image包中浏览其他的图像类。

第 24 章　附 加 的 包

在Java 1.0发布时，它包括一系列的8个包，它们被叫做核心应用编程接口(API)。其中一些包已在前面各章中介绍过，而且你每天编程时经常用到它们。现在，每一个后续版本发布时，就会增加到核心API中。Java的API已经包括了很多包。在这些新的包中，有许多包支持本书中没有讲述的各专业领域。然而，这里将对以下三个包做简要论述：java.lang.reflect，java.rmi和java.text。它们分别支持自省，远程方法调用和文本格式化。

所谓的自省是软件分析自己的能力。它是在第25章中将被讨论的Java Beans技术的主要部分。这里我们提供了介绍这个概念的例子。远程方法调用(RMI)则允许我们建立一个分布式的Java应用程序。在本章，提供了一个简单的客户机/服务器的例子，这个例子使用了RMI技术。

24.1　核心Java API 包

在表24-1中，我们列出了所有被Java 2定义的Java核心API包并且对它们进行了简要的描述。

表 24-1　核心 Java API 包

包（Package）	主要功能
java.applet	支持小应用程序的结构
java.awt	提供图形用户接口的能力
java.awt.color	支持颜色空间和外形
java.awt.datatransfer	与系统的剪贴板交换数据
java.awt.dnd	支持拖拉操作
java.awt.event	处理事件
java.awt.font	描述多种字体类型
java.awt.geom	允许你使用几何形状
java.awt.im	允许编辑组件中日文，中文，韩文的输入
java.awt.im.spi	支持二选一的输入设备 (在 Java 2, v1.3中加入)
java.awt.image	处理图像
java.awt.image.renderable	支持独立显示图像
java.awt.print	支持一般的打印功能
java.beans	允许你建立软件组件
java.beans.beancontext	为bean 提供可执行环境
java.io	输入输出数据

续表

包（Package）	主要功能
java.lang	提供核心功能
java.lang.ref	使能与垃圾回收交互
java.lang.reflect	运行时分析代码
java.math	处理大整数和十进制数
java.net	支持网络功能
java.rmi	支持远程方法调用
java.rmi.activation	激活永久对象
java.rmi.dgc	管理分布垃圾回收
java.rmi.registry	映射名称到远程对象引用
java.rmi.server	支持远程方法调用
java.security	处理证书，密钥，摘要，签名和其他安全功能
java.security.acl	管理访问控制列表
java.security.cert	分析和管理证书
java.security.interfaces	为DSA（数字签名算法）定义接口
java.security.spec	设定密钥和算法参数
java.sql	与SQL（结构化查询语言）数据库交互
java.text	格式化，查询和处理文本
java.util	包含一般工具
java.util.jar	生成和打开JAR文件
java.util.zip	读写压缩或解压缩文件

24.2　自　省

自省是软件分析自己的能力。这个功能由Class类中的java.lang.reflect包和元素所提供。自省是一个重要的功能，当我们使用被称为Java Beans的组件时将用到它。它允许你在运行时而不是在编译时分析一个软件组件并且动态的描述它的功能。例如，通过使用自省，你能决定一个类支持哪些方法、构造函数和成员属性。

包java.lang.reflect有一个叫做Member的接口，在这个接口中定义了那些允许获得类中域、构造函数或方法信息的方法。在这个包中也有8个类。在表24-2中列出了它们。

表 24-2　java.lang.reflect　中定义的类

类（Class）	主要功能
AccessibleObject	允许你绕过默认的访问控制检查 (在 Java 2中加入)
Array	允许你动态生成和处理数组
Constructor	提供有关构造函数的信息
Field	提供有关域的信息

续表

类（Class）	主要功能
Method	提供关于方法的信息
Modifier	提供有关类和成员访问修饰符的信息
Proxy	支持动态代理类 (在 Java 2, v1.3中加入)
ReflectPermission	允许一个类中私有的和被保护的成员思考(在Java 2中加入)

接下来的这个应用程序演示了如何简单的使用Java自省功能。这个程序输出了java.awt.Dimension类中的构造函数、成员属性和方法的信息。该程序开始时通过使用class对象的forname方法获得了一个java.awt.Dimension类的对象。在获得这个对象之后，getConstructors( )，getFields( )和getMethods( )方法被用来分析这个类的对象，它们分别返回了用来提供相应对象信息的Constructor，Field和Method对象的数组。在Constructor，Field和Method类中定义了一些可以用来获得一个对象信息的方法。然而，它们都支持toString( )方法。因此，将Constructor，Field和Method对象作为参数来调用println( )方法是最简单的，下面的例子正是这样做的。

```
// Demonstrate reflection.
import java.lang.reflect.*;
public class ReflectionDemo1 {
  public static void main(String args[]) {
    try {
      Class c = Class.forName("java.awt.Dimension");
      System.out.println("Constructors:");
      Constructor constructors[] = c.getConstructors();
      for(int i = 0; i < constructors.length; i++) {
        System.out.println(" " + constructors[i]);
      }

      System.out.println("Fields:");
      Field fields[] = c.getFields();
      for(int i = 0; i < fields.length; i++) {
        System.out.println(" " + fields[i]);
      }

      System.out.println("Methods:");
      Method methods[] = c.getMethods();
      for(int i = 0; i < methods.length; i++) {
        System.out.println(" " + methods[i]);
      }
    }
    catch(Exception e) {
      System.out.println("Exception: " + e);
    }
  }
}
```

这个程序的输出如下所示：

```
Constructors:
```

```
 public java.awt.Dimension(java.awt.Dimension)
 public java.awt.Dimension(int,int)
 public java.awt.Dimension()
Fields:
 public int java.awt.Dimension.width
 public int java.awt.Dimension.height
Methods:
 public final void java.lang.Object.wait()
   throws java.lang.InterruptedException
 public final void java.lang.Object.wait(long,int)
   throws java.lang.InterruptedException
 public final native void java.lang.Object.wait(long)
   throws java.lang.InterruptedException
 public final native java.lang.Class java.lang.
                       Object.getClass()
 public final native void java.lang.Object.notify()
 public final native void java.lang.Object.notifyAll()
 public java.lang.Object java.awt.geom.Dimension2D.clone()
 public void java.awt.geom.Dimension2D.
               setSize(java.awt.geom.Dimension2D)
 public int java.awt.Dimension.hashCode()
 public boolean java.awt.Dimension.equals(java.lang.Object)
 public java.lang.String java.awt.Dimension.toString()
 public void java.awt.Dimension.setSize(int,int)
 public void java.awt.Dimension.setSize(java.awt.Dimension)
 public void java.awt.Dimension.setSize(double,double)
 public double java.awt.Dimension.getWidth()
 public double java.awt.Dimension.getHeight()
 public java.awt.Dimension java.awt.Dimension.getSize()
```

接下来的这个例子使用了Java的自省功能来获得一个类的公用方法。这个程序首先实例化A类。getClass()方法被应用到这个对象的引用，并且返回对于A类的class对象。然后，通过调用后面这个对象的方法来获得一个Method对象数组，这个数组只用来描述A类中说明的方法。从超类继承来的方法例如Object将不在其中。

然后处理Methods对象数组中的每一个元素。getModifiers( )方法返回了一个整数，这个整数包括了那些用来描述这个元素访问修饰符的标志。Modifier类提供了一系列的可以用来检查这个值的方法，它们在表24-3中列出。例如，当它的参数包括了public访问控制值时，静态方法isPublic( ) 返回true，否则就返回false。在接下来的程序中，如果方法支持public访问，它的名字将通过getName( )方法获得，然后打印出来。

```
// Show public methods.
import java.lang.reflect.*;
public class ReflectionDemo2 {
  public static void main(String args[]) {
    try {
      A a = new A();
      Class c = a.getClass();
      System.out.println("Public Methods:");
      Method methods[] = c.getDeclaredMethods();
      for(int i = 0; i < methods.length; i++) {
        int modifiers = methods[i].getModifiers();
        if(Modifier.isPublic(modifiers)) {
```

```
            System.out.println(" " + methods[i].getName());
          }
        }
      }
      catch(Exception e) {
        System.out.println("Exception: " + e);
      }
    }
  }

  class A {
    public void a1() {
    }
    public void a2() {
    }
    protected void a3() {
    }
    private void a4() {
    }
  }
```

这个程序的输出如下所示：

```
Public Methods:
  a1
  a2
```

表 24-3 Modifier 中定义的获得访问控制修饰符的方法

方法	描述
static boolean isAbstract(int val)	如果val设定了abstract 标识，则返回true；否则返回false
static boolean isFinal(int val)	如果val设定了final 标识，则返回true；否则返回false
static boolean isInterface(int val)	如果val设定了interface 标识，则返回true；否则返回false
static boolean isNative(int val)	如果val设定了native 标识，则返回true；否则返回false
static boolean isPrivate(int val)	如果val设定了private 标识，则返回true；否则返回false
static boolean isProtected(int val)	如果val设定了protected 标识，则返回true；否则返回false
static boolean isPublic(int val)	如果val设定了public 标识，则返回true；否则返回false
static boolean isStatic(int val)	如果val设定了static 标识，则返回true；否则返回false
static boolean isStrict(int val)	如果val设定了strict 标识，则返回true；否则返回false
static boolean isSynchronized(int val)	如果val设定了synchronized 标识，则返回true；否则返回false
static boolean isTransient(int val)	如果val 设定了transient 标识，则返回true；否则返回false
static boolean isVolatile(int val)	如果val 设定了volatile 标识，则返回true；否则返回false

24.3 远程方法调用

远程方法调用（RMI）允许运行在一台机器上的Java对象可以调用运行在另一台机器上的Java对象的一个方法。这是Java的一个重要特性，因为它允许你去建立分布式的应用程

序。虽然对于RMI的完整讨论已经超出了这本书的范围，但是接下来的例子描述了涉及到的与之相关的基本原则。

24.3.1　一个使用了RMI技术的简单的客户机/服务器应用程序

下面一步一步的说明了如何创建一个简单的使用RMI技术的客户机/服务器应用程序。服务器从一个客户端获得一个请求，然后处理它并且返回一个结果。在这个例子中，收到的是两个数字。服务器把它们相加，然后返回它们的和。

第一步：输入并且编译源代码

这个应用程序使用了四个源文件。AddServerIntf.java是第一个源文件，在这个文件中定义了服务器提供的远程接口。这个接口包括一个接受两个double参数然后返回其和的方法。所有的远程接口必须继承Remote接口，这个接口是java.rmi包的一部分。Remote类没有成员变量。它的目的就是简单的说明一个使用了远程方法的接口。所有的远程方法都可以引发一个RemoteException异常。

```
import java.rmi.*;
public interface AddServerIntf extends Remote {
  double add(double d1, double d2) throws RemoteException;
}
```

第二个源文件是AddServerImpl.java，它实现了远程接口。Add()方法的实现是很简单的。所有远程对象必须扩展UnicastRemoteObject类，在这个类中提供了对象可以被远程调用的功能。

```
import java.rmi.*;
import java.rmi.server.*;
public class AddServerImpl extends UnicastRemoteObject
  implements AddServerIntf {

  public AddServerImpl() throws RemoteException {
  }
  public double add(double d1, double d2) throws RemoteException {
    return d1 + d2;
  }
}
```

第三个源文件是AddServer.java，在这个源文件中包括了服务器端的主程序。它的主要功能是更新机器上的RMI注册表。这些是通过调用java.rmi包中Naming类的rebind()方法来实现的。这个方法可以将一个名字和一个对象引用联系起来。rebind()的第一个参数是服务器的名字，第二个参数是一个AddServerImpl类的实例的引用。

```
import java.net.*;
import java.rmi.*;
public class AddServer {
  public static void main(String args[]) {
    try {
      AddServerImpl addServerImpl = new AddServerImpl();
      Naming.rebind("AddServer", addServerImpl);
```

```
    }
    catch(Exception e) {
      System.out.println("Exception: " + e);
    }
  }
}
```

第四个源文件是AddClient.java，这个文件实现了这个分布式应用程序的客户端。AddClient.java需要三个命令行参数。第一个是服务器机器的名字或者是IP地址。第二个和第三个是那两个用于计算的数。

应用程序首先形成一个URL形式的字符串。这个URL使用了RMI协议。这个字符串包括服务器的IP地址或者名字和字符串“AddServer”。然后这个程序调用Naming类的lookup( )方法。这个方法以一个RMI的URL为参数，返回一个AddServerIntf对象的引用。接着，所有的远程方法就可以通过它来调用了。

接下来，这个程序显示了它的参数，然后调用远程方法add()，最后从这个方法返回两数相加之和，然后打印出来。

```
import java.rmi.*;
public class AddClient {
  public static void main(String args[]) {
    try {
      String addServerURL = "rmi://" + args[0] + "/AddServer";
      AddServerIntf addServerIntf =
                  (AddServerIntf)Naming.lookup(addServerURL);
      System.out.println("The first number is: " + args[1]);
      double d1 = Double.valueOf(args[1]).doubleValue();
      System.out.println("The second number is: " + args[2]);

      double d2 = Double.valueOf(args[2]).doubleValue();
      System.out.println("The sum is: " + addServerIntf.add(d1, d2));
    }
    catch(Exception e) {
      System.out.println("Exception: " + e);
    }
  }
}
```

在你输入了所有代码之后，用javac编译这四个源程序。

第二步：生成存根和主框架

在你可以使用客户端和服务器之前，你必须产生必要的存根（stub）。你同样也可能需要产生一个主框架。在RMI上下文中，一个存根是一个存放在客户端机器上的Java对象，它的功能是提供和远程服务器相同的接口。由客户端初始化的远程方法实际上就是存根。这个存根和RMI系统的其他部分一起工作产生一个要送到远程机器的请求。

一个远程方法可以接受那些简单类型或者对象参数。在后一种情况中，这个对象可能包括别的对象的引用。所有这些信息必须被发送到远程机器。作为远程方法调用的一个参数，一个对象必须被序列化之后再被送到远程机器。在第17章中，序列化的功能也递归处理了所有被引用的对象。

在Java 2中，主框架（Skeletons）不再被需要。然而，它们在Java 1.1的RMI模型中是必要的。正是由于这个原因，为了兼容Java 1.1和Java 2，主框架仍然是需要的。一个主框架是一个存放在服务器机器上的Java对象。它和Java 1.1的RMI系统的其他部分一起工作来接受请求，执行逆序列化和调用服务器上相应的代码。与Java 1.1不同，主框架机制对于Java 2的代码不是必须的。因为许多读者想产生主框架，所以下面的例子使用了主框架。

如果一个响应必须要返回给客户端，进程按相反的方向进行工作。请注意，如果一个对象被返回给客户端，那么也使用序列化和逆序列化功能。

为了产生存根和主框架，你需要用远程方法调用编译器（RMI compiler），它可以像下面这样在命令行被调用：

```
rmic AddServerImpl
```

这个命令生产了两个新的文件：AddServerImpl_Skel.class(主框架)和AddServerImpl_Stub.class(存根)。当使用rmic时，请确信设置CLASSPATH包括了当前目录。正如你所看到的，默认情况下，rmic同时产生一个存根和一个主框架文件。如果你不需要主框架，你可以有选择的放弃它。

第三步：安装文件到客户端和服务器

复制AddClient.class，AddServerImpl_Stub.class和AddServerIntf.class到客户端机器上的一个目录中。复制AddServerIntf.class, AddServerImpl.class, AddServerImpl_Skel.class, AddServerImpl_Stub.class和AddServer.class到服务器的一个目录中。

注意：RMI可以动态地加载类，但是它们不能被目前的例子使用。相反，所有的被客户端和服务器程序使用的文件必须安装在这些机器上。

第四步：在服务器端启动 rmi 的注册程序

JDK提供了一个叫做rmiregistry的程序，这个程序在服务器上执行。它可以映射名字到对象的引用。首先，检查包括当前文件夹的CLASSPATH环境变量。然后，用命令行启动RMI注册，方式如下所示：

```
start rmiregistry
```

当这个命令行返回时，你将看到产生了一个新的窗口。你需要保持这个窗口打开，直到你用RMI的例子做完了试验。

第五步：启动服务器

服务器的程序使用下面的命令行启动：

```
java AddServer
```

重新调用AddServer代码创建AddServerImpl的实例并且用“AddServer”这个名字注册这个对象。

第六步：启动客户端

AddClient程序需要三个参数：服务器的名字或IP，两个加数。你可以通过以下两种方式中的一种从命令行来调用它。

```
java AddClient server1 8 9
java AddClient 11.12.13.14 8 9
```

在第一种方式中，提供服务器的名字，在第二种方式中则使用了它的IP地址。

你可以尝试不用远程服务器来执行这个例子。你可以将所有的程序安装在一台机器上，启动rmiregistry，启动AddSever，然后用下面的方式执行AddClient。

```
java AddClient 127.0.0.1 8 9
```

在这里，地址127.0.0.1是本机的回送地址。用这个地址可以让你练习RMI的机制而不用安装一个远程的计算机。

无论那一种情况，这个例子的输出都如下所示：

```
The first number is: 8
The second number is: 9
The sum is: 17
```

24.4 文本的格式化

java.text包可以让你格式化，查找和操作文本。在这一节简单地介绍了这个包中最常用的类，它们可以用来格式化日期和时间信息。

24.4.1 DateFormat类

DateFormat类是一个提供了格式化和解析日期和时间功能的抽象类。getDateInstance( )方法可以返回一个DateFormat类的实例，这个对象可以格式化日期信息。这个方法如下所示：

```
static final DateFormat getDateInstance( )
static final DateFormat getDateInstance(int style)
static final DateFormat getDateInstance(int style, Locale locale)
```

在这里，参数style是下列值中的一个：DEFAULT, SHORT, MEDIUM, LONG或者FULL。这些都是DateFormat类定义的整数常量。它们代表着日期显示的不同方式。参数locale是由Locale类(请参考第16章了解更多细节)定义的静态引用之一。如果style或者locale没有被指定，将使用默认方式。

在这个类中最常用的方法是format()。它有几种重载方式，其中的一种如下所示：

```
final String format(Date d)
```

这个方法的参数是一个将要显示的Date对象。这个方法返回一个包含了格式化信息的字符串。

接下来的例子说明了如何格式化日期信息。它首先产生了一个Date对象，通过这个对象我们获得了当前的日期和时间信息。然后它用不同的style和locale输出日期信息。

```
// Demonstrate date formats.
import java.text.*;
import java.util.*;

public class DateFormatDemo {
  public static void main(String args[]) {
    Date date = new Date();
    DateFormat df;

    df = DateFormat.getDateInstance(DateFormat.SHORT, Locale.JAPAN);
    System.out.println("Japan: " + df.format(date));

    df = DateFormat.getDateInstance(DateFormat.MEDIUM, Locale.KOREA);
    System.out.println("Korea: " + df.format(date));

    df = DateFormat.getDateInstance(DateFormat.LONG, Locale.UK);
    System.out.println("United Kingdom: " + df.format(date));

    df = DateFormat.getDateInstance(DateFormat.FULL, Locale.US);
    System.out.println("United States: " + df.format(date));
  }
}
```

这个例子的输出如下所示：

```
Japan: 01/02/19
Korea: 2001-02-19
United Kingdom: 19 February 2001
United States: Monday, February 19, 2001
```

getTimeInstance( )方法返回了一个DateFormat的实例，这个实例可以用来格式化时间信息。这个方法如下所示：

```
static final DateFormat getTimeInstance( )
static final DateFormat getTimeInstance(int style)
static final DateFormat getTimeInstance(int style, Locale locale)
```

参数style是DEFAULT，SHORT，MEDIUM，LONG或FULL中的一个。这些整数常量是由DateFormat类定义的。它们决定了时间显示的不同方式。参数locale是由Locale类定义的静态引用之一。如果style或者locale没有被指定，将使用默认方式。

接下来的例子说明了如何格式化时间信息。它首先产生了一个Date对象，通过这个对象我们获得了当前的日期和时间信息。然后它用不同的style和locale输出时间信息。

```
// Demonstrate time formats.
import java.text.*;
import java.util.*;
public class TimeFormatDemo {
  public static void main(String args[]) {
    Date date = new Date();
    DateFormat df;
```

```
        df = DateFormat.getTimeInstance(DateFormat.SHORT, Locale.JAPAN);
        System.out.println("Japan: " + df.format(date));

        df = DateFormat.getTimeInstance(DateFormat.LONG, Locale.UK);
        System.out.println("United Kingdom: " + df.format(date));

        df = DateFormat.getTimeInstance(DateFormat.FULL, Locale.CANADA);
        System.out.println("Canada: " + df.format(date));
      }
    }
```

这个例子的输出如下所示：

```
Japan: 20:25
United Kingdom: 20:25:14 GMT-05:00
Canada: 8:25:14 o'clock PM EST
```

getDateTimeInstance( )方法返回了一个DateFormat的实例，这个实例即可以用来格式化日期信息也可以用来格式化时间信息。你可以自己练习一下。

24.4.2 SimpleDateFormat类

SimpleDateFormat类是DateFormat类的一个子类，它可以让你定义自己用来显示日期和时间信息的格式化模型。

它的一个构造函数如下所示：

```
SimpleDateFormat(String formatString)
```

formatString参数描述了如何显示日期和时间信息，使用它的一个例子如下所示：

```
SimpleDateFormat sdf = SimpleDateFormat("dd MMM yyyy hh:mm:ss zzz");
```

在格式化字符串中使用的格式决定了信息的显示方式。在表24-4中列举了这些符号并且分别给出了解释。

表 24-4 SimpleDateFormat 中用于格式化字符串的符号

符号（Symbol）	描述
a	上午（AM） 或下午（ PM）
d	一个月中的某天(1–31)
h	上午或下午的某小时(1–12)
k	一天中的某小时 (1–24)
m	一小时里的某分钟 (0–59)
s	一分钟里的某一秒(0–59)
w	一年中的某星期 (1–52)
y	年
z	时区
D	一年里的某一天 (1–366)

续表

符号（Symbol）	描述
E	一星期里的某天(例如, 星期四)
F	某月的工作日数
G	纪元(即AD或BC)
H	一天中的某小时(0~23)
K	上午或下午的某小时(0~11)
M	月份
S	秒中的毫秒
W	某月中的某个星期 (1–5)
'	取消字符

在大多数情况下，字符数中一个符号重复的次数决定如何显示日期。如果模式字母被重复次数不超过4次，那么文本信息将用压缩的形式显示。否则，将使用没有压缩的形式显示。例如，一个zzzz模式可以显示太平洋白天时间，所以一个zzz模式可以显示PDT。

对于数字，时间数字中一个模式字符被重复的次数决定了多少数字将出现。例如，hh:mm:ss可以代表01:51:15,但是h:m:s显示相同的值为1:51:15。

最后，M或者是MM将使月用一个和两个数字来显示。然而，三个以上M的重复将使月作为文本字符来显示。

接下来的程序中演示了如何使用这个类。

```
// Demonstrate SimpleDateFormat.
import java.text.*;
import java.util.*;

public class SimpleDateFormatDemo {
  public static void main(String args[]) {
    Date date = new Date();
    SimpleDateFormat sdf;
    sdf = new SimpleDateFormat("hh:mm:ss");
    System.out.println(sdf.format(date));
    sdf = new SimpleDateFormat("dd MMM yyyy hh:mm:ss zzz");
    System.out.println(sdf.format(date));
    sdf = new SimpleDateFormat("E MMM dd yyyy");
    System.out.println(sdf.format(date));
  }
}
Sample output from this program is shown here:
```

这个例子的输出如下所示：

```
11:51:50
19 Feb 2001 11:51:50 CST
Mon Feb 19 2001
```

第3部分　Java软件开发技术

第 25 章　Java Beans

本章介绍Java编程技术的前沿：Java Beans。Beans非常重要，它允许开发人员由软件组件构造复杂系统。这些软件组件可以自己开发，也可以由一个或多个不同厂商提供。Java Beans定义了组件块互操作行为的体系结构。

为了更好的理解Beans的价值，对比考虑下列情况：硬件开发人员可以选择多个元件，例如电阻，电容和电感，使用这些元件，硬件开发人员可以很容易的构建系统。集成电路提供了更高级的功能。所有的集成电路芯片都可以重用，在构建新系统时，不必也不太可能重建所有元件的功能。而且相同的芯片可以用在不同类型的电路中。硬件开发人员可以这样做是因为这些硬件元器件的特性已知，且有相关文档可供查询。

不幸的是，软件工业还没有得到这种可重用性和互操作性。大应用程序复杂度剧增，难以维护和升级。部分原因是因为，到目前为止，还没有一个标准的，可移植的方式用于编写软件组件。为了得到组件软件的好处，首先需要一个组件体系结构，这个体系结构允许由不同厂商提供的组件组装成应用程序。设计者必须能够选择一个组件，理解其功能，将其集成进一个应用程序中。当一个组件的新版本出现后，这个新版本组件应非常容易的与其他已存在的代码互操作。幸运的是，Java Beans提供了一个这样的体系结构。

本章以Java Beans的定义开始，描述了Java Beans提供给软件设计者的有利条件。同时，介绍了来自JavaSoft的Beans Developer Kit (BDK)开发工具。这个工具可以连接多个Beans。最后，介绍如何编写一个简单的Beans，并在BDK中使用这个Beans。

25.1　Java Bean是什么

Java Bean是一个软件组件，被设计成可以在不同的环境里重复使用。Bean的功能没有限制。一个Bean可以完成一个简单的功能，如检查一个文件的拼写，也可以完成复杂功能，如预测一只股票的业绩。Bean对最终用户是可见的，如图形用户界面上的一个按钮。Bean也可能对用户不可见，如实时多媒体解码软件。最后，一个Bean可以被设计成在用户工作站上独立工作，也可以与其他一组分布式组件协调工作。由一组数据点生成饼图的软件是一个可以本地工作的Bean的例子。而提供实时股票价格信息或是日用品流通中心的实时价格信息的Bean必须与其他分布式软件合作以获得数据。

下面将要介绍的是如何改动一个类使之成为一个Java Bean。然而，Java设计者的目标

之一是简化这项技术。因此代码的改动量很小。

25.2　Java Beans的优点

一个软件组件体系结构提供标准机制处理软件构件块。下面列举的优点是Java技术提供给组件开发人员的：

- 一个Bean拥有Java的“一次编程，随处运行”的特性。
- 可以使用工具控制一个Bean的属性、事件和方法。
- Bean在不同地区都可以正常工作，这点对将产品推广至全球非常重要。
- 辅助软件可以帮助使用者配置Bean。仅在设计时为组件设置参数时才需要此辅助软件，运行环境中无须包括。
- Bean的配置工作保存在永久存储区域中，在使用时恢复即可。
- Bean注册接收来自其他对象的事件，并能产生事件送往其他对象。

25.3　应用程序开发工具

在使用Java Bean时，多数开发者使用应用程序开发工具(application builder tool)，这个工具可以配置一组Bean，将它们互联在一起，生成一个可工作的应用程序。这个工具的主要功能部件如下：

- 一个调色板，列出全部可用的Bean。其他自己开发的Bean或是购买的Bean，可以被加到这个调色板中。
- 一个工作平台，设计者可以在此排列Bean。设计者可以在调色板和工作平台之间拖放Bean。
- 专用的编辑器和定制的工具可以配置Bean。这个机制使Bean的行为可以适应特殊的环境。
- 允许设计者查询Bean的状态和行为的命令。当一个Bean加入调色板时这些信息自动变为可用。
- 连接Bean的能力。这意味着一个组件产生的事件可以映射成其他组件的方法调用。
- 当一组Bean配置连接完毕，它将所有的信息保存在一个稳定存储区域中。在使用时，利用这些信息恢复应用程序的状态。

25.4　Bean开发工具包(BDK)

Bean 开发工具包（Bean Developer Kit，BDK）可以从JavaSoft站点下载，这是一个简单工具可以用来创建，配置，连接一组Bean。这个工具中还包括一些带源码的Bean示例。本节提供了安装和使用的逐步说明。

注意：本章中，说明都是Windows环境中的。UNIX平台的过程与此类似，但是某些命令不同。

25.4.1 安装BDK

在安装BDK之前必须先安装JDK，确保JDK工具可以访问。

首先，从JavaSoft 站点 (http://java.sun.com)下载BDK。一般，BDK被打包成一个自解压的文档。其次，按照说明安装BDK。本书的其余部分假设BDK被安装在bdk目录中。如果你的系统中的BDK目录不是这样的，请替换成正确的目录。

25.4.2 启动BDK

按下列步骤，启动BDK：

1. 进入目录c:\bdk\beanbox。
2. 执行批处理文件run.bat.如图25-1，BDK显示三个窗口。ToolBox列出BDK中包括的所有Bean，BeanBox提供了一个可以编排连接Bean的区域。Properties属性窗口提供了配置Bean的能力。可能还有一个叫做Method Tracer的窗口，但本节中并不使用这个窗口。

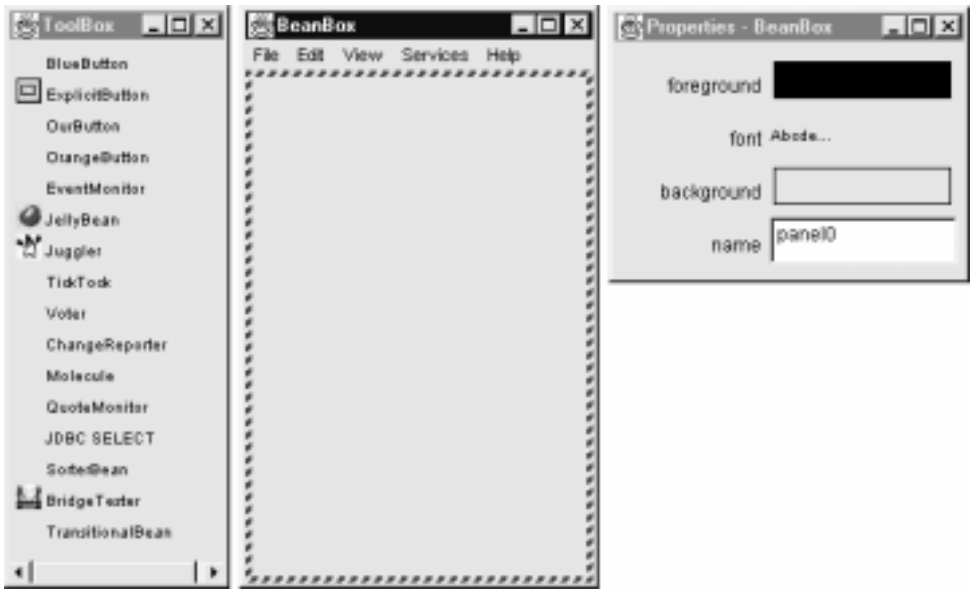


图 25-1 Bean 开发工具（BDK）

25.4.3 使用BDK

本节描述如何使用BDK提供的Bean创建一个应用程序。首先，Molecule Bean显示一个三维的分子模型。这个bean可被配置成下列分子：透明质酸，苯，环己烷，乙烷或水分子。这个组件还提供使分子沿X或Y轴旋转的方法。其次，OurButton Bean提供了一个按钮功能，可以将一个按钮标为“旋转X”，使分子沿X轴旋转，另一个按钮标为“选择Y”，使分子

沿Y轴旋转。

图25-2显示应用程序的外观。

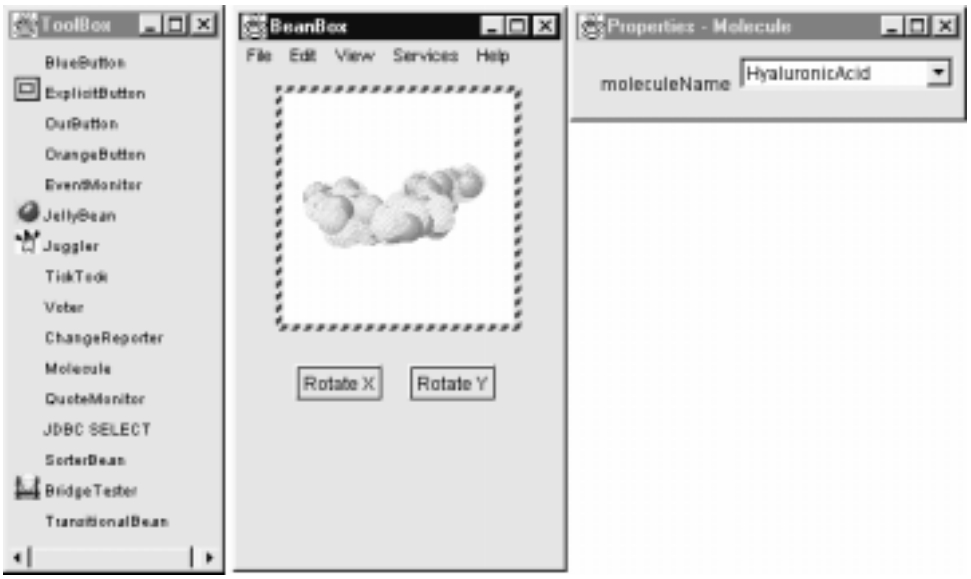


图 25-2　分子模型和 OurButton Beans

创建并配置 Molecule Bean 的一个实例

按下列步骤创建配置Molecule Bean的一个实例：

1. 将光标移至ToolBox中标为Molecule的条目上，点击鼠标左键。光标变成十字形。
2. 将光标移至BeanBox的显示区域，在希望放置Bean的位置点击鼠标左键。这时，一个包含分子的3维模型的矩形出现。这个区域环绕着阴影线框，指示此为当前选择项。
3. 将光标定位在一个阴影线框中，拖曳Bean，可以重新放置Molecule Bean。
4. 在属性窗口中改变选择，可以改变此分子。注意当改变所选的分子时，Bean外观立即改变。

创建并配置 OurButton 的一个实例

按下列步骤创建并配置OurButton Bean的一个实例，并将此实例与Molecule Bean实例连接：

1. 将光标移至ToolBox中标为OurButton的项上，点击鼠标左键。光标变成十字形。
2. 将光标移至BeanBox的显示区域，在希望放置Bean的位置点击鼠标左键。这时，一个包含按钮的矩形出现。这个区域环绕着阴影线框，指示此为当前选择项。
3. 将光标定位在一个阴影线框中，拖曳Bean，可以重新放置OurButton Bean.
4. 在属性窗口中，将Bean的标签改为“旋转X”。当属性改变时，按钮的外观立即改变。

5. 在BeanBox的菜单栏上，选择Edit | Events | action | actionPerformed后，出现一条从按钮到光标的线。注意线的一端随着光标移动，而另一端固定在按钮上。
6. 将光标移至Molecule Bean的显示区域，点击鼠标左键，Event Target Dialog对话框出现。
7. 在对话框中选择一个按钮被按下时所调用的方法。选择标为“沿X旋转”项，然后点击OK按钮。短时内会出现一个消息框，说明工具正在“生成并编译改编后的类”

测试应用程序。每次按下按钮，分子将沿着坐标轴移动一定角度。

现在创建OurButton Bean的另一个实例，标识为“旋转Y”，将其映射到Molecule Bean的“沿Y旋转”事件上。这个步骤与刚才描述的“旋转X”按钮非常类似。

测试应用程序，按下按钮观察分子的移动。

25.5 JAR文件

在开发自己的Bean之前，了解JAR（Java存档文档）文件是非常必要的，因为与BDK类似的工具希望Bean被打包成JAR文件。一个JAR文件可以装入一组类及它们的相关资源。例如，开发者完成另一个多媒体应用程序，这个应用程序使用多个声音和图像文件。一组Bean控制如何和何时使用这些信息。所有这些都可以放入一个JAR文件中。

JAR技术使发布、安装软件更容易。而且，JAR文件的组件都经过压缩，这样下载一个JAR文件要比分开下载多个未压缩文件要快得多。JAR文件中的组件可以附带数字签名。客户可以确定组件是否来自一个特定组织或个人。

注意：java.util.zip包中包括读写JAR文件的类。

25.5.1 清单文件（Manifest Files）

开发者必须提供一个清单文件，清单文件指示JAR文件中的Java Bean组件。下面是一个清单文件的例子。它定义了一个包含4个.gif文件和一个.class文件，最后一项是一个Bean。

```
Name: sunw/demo/slides/slide0.gif
Name: sunw/demo/slides/slide1.gif
Name: sunw/demo/slides/slide2.gif
Name: sunw/demo/slides/slide3.gif
Name: sunw/demo/slides/Slides.class
Java-Bean: True
```

一个清单文件可能涉及多个.class文件。如果一个.class文件是一个Java Bean，这一行必须立即跟随“Java-Bean: True”行。

25.5.2 JAR工具

这个工具用来生成JAR文件，其语法如下：

```
jar options files
```

表25-1列出所有选项和选项的含义。下面的例子说明如何使用这个工具。

表 25-1　JAR 命令选项

选项	描述
c	创建一个新存档文件
C	在命令执行期间改变目录
f	文件列表中的第一个元素是要创建或是访问的存档文件名字
i	应提供的索引信息
m	文件列表中的第二个元素是外部的清单文件名
M	不创建清单文件
t	存档文件的内容应制成表格
u	更新存在的JAR文件
v	当工具执行时显示详细信息
x	从归档文件中展开文件（如果只有一个文件，那就是归档文件的名字，其中的所有文件都被展开。否则，文件列表中的第一个元素是归档文件名字，其余的元素是应从归档文件中展开的文件）
0	不压缩

创建一个 JAR 文件

下面的命令创建一个名为Xyz.jar的文件，此文件包括当前目录下的所有.class 和.gif文件。

```
jar cf Xyz.jar  *.class  *.gif
```

如果存在一个可用的清单文件如Yxz.mf，使用时用下面的命令：

```
jar cfm Xyz.jar Yxz.mf  *.class  *.gif
```

列出 JAR 文件的内容

下面的命令列出Xyz.jar的内容：

```
jar tf Xyz.jar
```

展开一个 JAR 文件

下面的命令展开Xyz.jar的内容并将文件放在当前目录下：

```
jar xf Xyz.jar
```

更新一个存在的 JAR 文件

下面的命令将文件file1.class加入Xyz.jar文件:

```
jar -uf Xyz.jar file1.class
```

递归目录

下面的命令将directoryX目录下的所有文件加入Xyz.jar文件：

```
jar -uf Xyz.jar -C directoryX *
```

25.6 内 省

内省（Introspection）是分析一个Bean确定其功能的过程。这是Java Beans API的一个基本特点，因为内省机制允许应用开发工具将一个组件的信息提供给软件设计者。没有内省机制，Java Beans 技术无法运转。

开发者可以通过两种方式说明Bean的属性、事件和那些应该对应用开发工具公开的方法。第一种方式是使用简单的命名规则。这允许内省机制推断出一个Bean的信息。第二种方式是提供一个附加类，这个类负责信息提供。这里先介绍第一种方法。第二种方法随后介绍。

下面说明属性和事件的设计模式，属性和事件决定了一个Bean的功能。

25.6.1 属性的设计模式

属性（property）是Bean状态的一个子集。属性值决定了组件的行为和表现。本小节讨论三种类型的属性：简单属性，布尔属性和索引属性

简单属性

一个简单属性只有一个值。可由下列设计模式确定，其中N是属性名，T是类型：

```
public T getN( );
public void setN(T arg);
```

一个读/写属性有读/写两种方法来访问其属性值。一个只读属性只有一个get方法。一个只写属性只有一个set方法。

下面例子有三个读/写简单属性类:

```
public class Box {
  private double depth, height, width;
  public double getDepth( ) {
    return depth;
  }
  public void setDepth(double d) {
    depth = d;
  }
  public double getHeight( ) {
    return height;
  }
  public void setHeight(double h) {
    height = h;
  }
  public double getWidth( ) {
```

```
    return width;
  }
  public void setWidth(double w) {
    width = w;
  }
}
```

布尔属性

一个布尔属性的值为true或者false，可由下列设计模式确定，其中N是属性名。

```
public boolean isN( );
public boolean getN( );
public void setN(boolean value);
```

第一种模式或第二种模式都可用作获取一个布尔属性的值。然而，如果一个类有这两种方法，则必须使用第一种模式。

下面的例子是一个有一个布尔属性的类:

```
public class Line {
  private boolean dotted = false;
  public boolean isDotted( ) {
    return dotted;
  }
  public void setDotted(boolean dotted) {
    this.dotted = dotted;
  }
}
```

索引属性

一个索引属性包括多个值，可由下列设计模式确定，其中N是属性名，T为属性值。

```
public T getN(int index);
public void setN(int index, T value);
public T[ ] getN( );
public void setN(T values[ ]);
```

下面的例子是一个有一个读/写索引属性的类：

```
public class PieChart {
  private double data[ ];
  public double getData(int index) {
    return data[index];
  }
 public void setData(int index, double value) {
    data[index] = value;
  }
  public double[ ] getData( ) {
    return data;
  }
  public void setData(double[ ] values) {
    data = new double[values.length];
    System.arraycopy(values, 0, data, 0, values.length);
  }
```

```
}
```

25.6.2 事件的设计模式

在本书的前面章节中已经讨论过使用授权事件模型的Bean。Bean生成事件，将事件发送给其他对象。这些由下面的设计模式确定，其中T是事件的类型。

```
public void addTListener(TListener eventListener);
public void addTListener(TListener eventListener) throws TooManyListeners;
public void removeTListener(TListener eventListener);
```

事件监听器使用这些方法注册对特定类型的事件的兴趣。第一种模式指示一个Bean可以将一个事件传送给多个监听器。第二种模式指示一个Bean可以一个事件单独传送给一个监听器。当一个事件监听器不再接收一种类型的事件通知时，它使用第三种模式。

下面的例子简略说明了一个类，它在一个温度值超出一定范围时会通知其他对象。两种方法指示其他实现TemperatureListener接口的对象也可以接收事件通知。

```
public class Thermometer {
  public void addTemperatureListener(TemperatureListener tl) {
    ...
  }
  public void removeTemperatureListener(TemperatureListener tl) {
    ...
  }
}
```

25.6.3 方法

设计模式不能用来命名一个非属性的方法。内省机制发现一个Bean的全部public方法，而对Protected和private方法不作介绍。

25.7 开发一个简单的Bean

本节介绍一个例子，说明如何开发一个简单的Bean并通过BDK将其与其他组件相连。

新组件的名字是Colors Bean。它看起来像一个填满一种颜色的长方形或椭圆。在Bean开始执行时，随机选择一种颜色。可以调用一个public方法改变这个颜色。每次鼠标点击这个Bean，则再随机选择一种颜色。其形状是由一个布尔读/写属性决定的。

使用BDK来构造有一个Colors Bean实例和一个OurButton Bean实例的应用程序。按钮标识为“改变”，每次按下该按钮，则图形的颜色改变。

图25-3显示这个应用程序的外观。

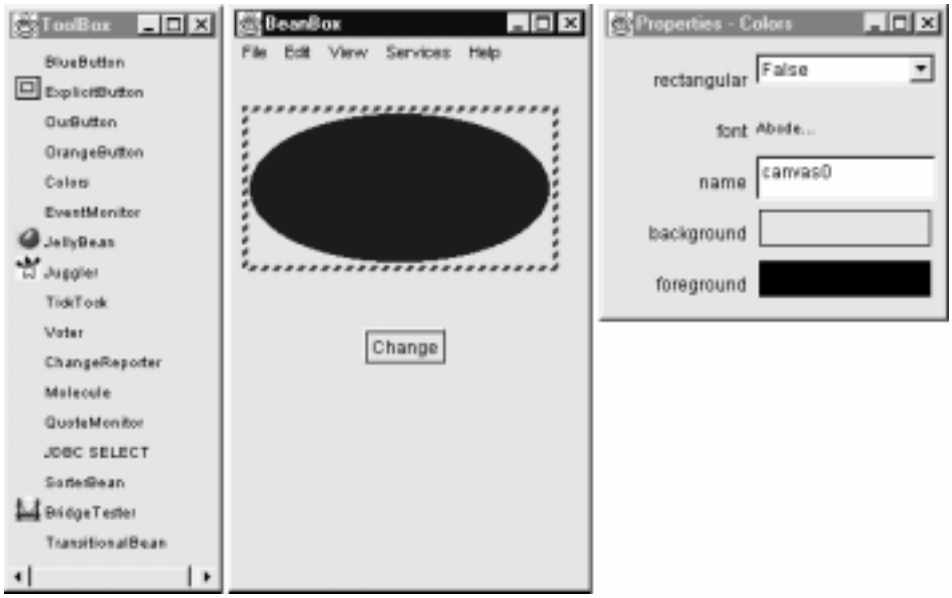


图 25-3　Colors 和 OutButton Beans

25.7.1　创建一个新Bean

创建一个新Bean必须遵照下列步骤：

1. 为新Bean创建一个目录。
2. 创建Java源文件（一个或多个）。
3. 编译源文件。
4. 创建一个清单文件。
5. 生成一个JAR文件。
6. 启动BDK。
7. 测试。

下面讨论这些步骤的细节。

为新 Bean 创建一个目录

需要为Bean 建立一个目录，按照这个例子的步骤，创建c:\bdk\demo\sunw\demo\colors，然后进入此目录。

为新 Bean 创建 Java 源文件

Colors 组件的源码如下所示。它被保存在Colors.java文件中.

文件开头的import声明指明文件在一个名为sunw.demo.colors的包中。回忆第9章的内容，目录层次对应着包的层次。因此这个文件必须放在相对于CLASSPATH环境变量的子目录sunw\demo\colors下。

组件的颜色由private Color变量color决定，外形由private boolean变量rectangular决定。

构造函数定义一个匿名的内部类，这个内部类继承了MouseAdapter类，并覆盖了mousePressed( )方法。当鼠标按下时调用change( )方法。组件初始化为一个长200像素宽100像素的长方形，并调用change( )方法选择一种随机颜色重画组件。

方法getRectangular( )和setRectangular( ) 可以访问Bean的一个属性。change( ) 方法调用randomColor( )选择一种颜色，然后调用repaint( )使此种改变可见。注意paint( )方法使用rectangular和color变量决定如何显示Bean。

```
// A simple Bean.
package sunw.demo.colors;
import java.awt.*;
import java.awt.event.*;
public class Colors extends Canvas {
 transient private Color color;
  private boolean rectangular;
  public Colors() {
    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent me) {
        change();
      }
    });
    rectangular = false;
    setSize(200, 100);
    change();
  }
  public boolean getRectangular() {
    return rectangular;
  }
  public void setRectangular(boolean flag) {
    this.rectangular = flag;
    repaint();
  }
  public void change() {
    color = randomColor();
    repaint();
  }
  private Color randomColor() {
    int r = (int)(255*Math.random());
    int g = (int)(255*Math.random());
    int b = (int)(255*Math.random());
    return new Color(r, g, b);
  }
  public void paint(Graphics g) {
    Dimension d = getSize();
    int h = d.height;
    int w = d.width;
    g.setColor(color);
    if(rectangular) {
      g.fillRect(0, 0, w-1, h-1);
    }
    else {
      g.fillOval(0, 0, w-1, h-1);
    }
  }
}
```

编译新 Bean 的源代码

编译源代码，产生一个class文件，键入下列命令:

```
javac Colors.java.
```

创建一个清单文件

现在必须创建一个清单文件。首先进入c:\bdk\demo目录。这个目录是BDK演示例程的清单文件的目录。将清单文件的源码键入colors.mft文件。源码如下所示：

```
Name: sunw/demo/colors/Colors.class
Java-Bean: True
```

这个文件说明在JAR文件中有一个.class文件，并且它是一个Java Bean。注意，Colors.class文件在sunw.demo.colors包中，而且在子目录sunw\demo\colors下。

生成一个 JAR 文件

只有当Bean在c:\bdk\jars目录下的JAR文件中时，才能出现在BDK的ToolBoox窗口中。这些文件由jar工具生成。键入下列命令：

```
jar cfm ..\jars\colors.jar colors.mft sunw\demo\colors\*.class
```

这个命令创建colors.jar文件并将其放在c:\bdk\jars目录下（可以将此命令放入批处理文件，以便以后使用）。

启动 BDK

加入c:\bdk\beanbox目录，键入run。启动BDK。BDK启动后出现三个窗口：ToolBox, BeanBox，和属性Properties。ToolBox窗口中应该包括一个为新Bean增加的名为“Colors”的项。

创建 Colors Bean 的一个实例

在完成前面所说的步骤后，在BeanBox窗口创建一个Colors Bean的实例。在其边界内随意按鼠标以测试新组件。它的颜色应该随鼠标按动即时改变。使用属性窗口改变rectangular的属性，从false改为true，其形状即时改变。

创建并配置 OurButton Bean 的一个实例

在BeanBox窗口创建一个OurButton Bean的实例。然后执行下列步骤：

1. 进入属性窗口，将Bean的标志改为“改变”。在属性改变时按钮的外观立即改变。
2. 在BeanBox的菜单栏中选择Edit | Events | action | actionPerformed。
3. 将光标移入Colors Bean的显示区域，点击鼠标左键。Event Target Dialog对话框弹出。
4. 使用这个对话框选择方法响应按钮。选择标明为“改变”的项，然后按下OK按钮。一个消息框快速闪现，指示工具正在“生成并编译改编后的类”。
5. 点击按钮，颜色改变。

在继续向前之前，你应该试一试Colors Bean。

25.8 使用限制属性

一个拥有限制属性（bound property）的Bean在属性改变时产生一个事件。这个事件是PropertyChangeEvent，且被发送给所有事先注册接收这样的事件通知的对象。

TickTock Bean由BDK提供。它每N秒生成一个属性改变事件。N是这个Bean的属性，可以通过BDK的属性窗口改变。下一个例子将创建一个使用TickTock Bean自动控制Colors Bean的应用程序。图25-4显示了这个应用程序。

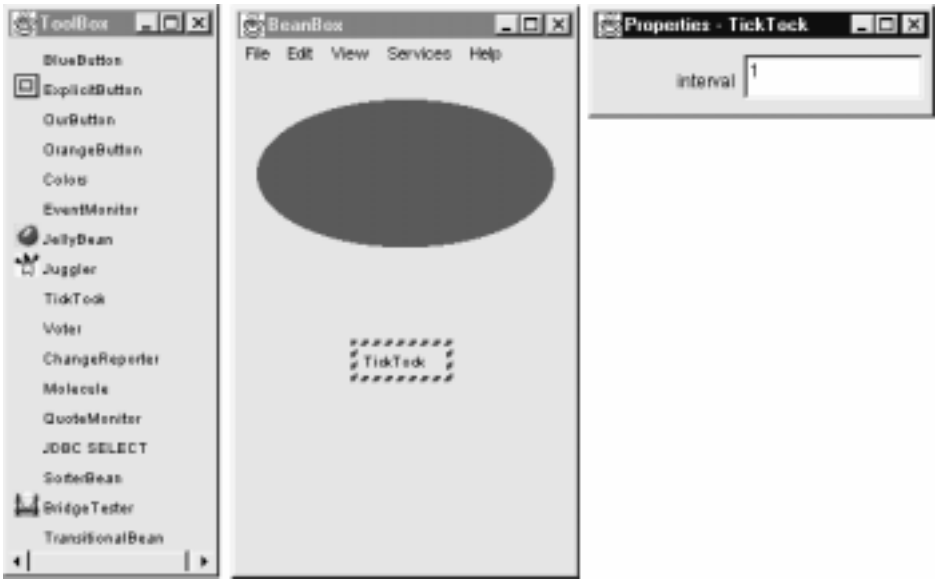


图 25-4 Colors 和 TickTock Beans

25.8.1 步骤

启动BDK，在BeanBox窗口中创建一个Colors Bean 的实例。

创建一个TickTock Bean的实例。属性窗口将显示这个组件的一个属性。其名字为“Interval”，其初始值为5。这代表按秒计算的TickTock Bean生成的属性改变事件的间隔时间。将属性值改为1。

现在将TickTock Bean 产生的事件映射为Colors Bean 的方法调用。步骤如下：

1. 在BeanBox的菜单栏中选择Edit | Events | propertyChange | propertyChange，出现一条从按钮延伸至光标的线。
2. 将光标移入Colors Bean的显示区域，点击鼠标左键，弹出Event Target Dialog对话框。
3. 使用这个对话框选择事件出现时希望被调用的方法。选择标为“改变”的项，按OK键。一个消息框快速闪现，指示工具正在“生成并编译改编后的类”。

现在组件每秒改变一次颜色。

25.9　使用BeanInfo接口

在前面的例子中，使用设计模式确定提供给Bean使用者的信息。本节描述开发人员如何使用BeanInfo接口明确的控制这一过程。

这个接口定义了几个方法，包括：

```
PropertyDescriptor[ ] getPropertyDescriptors( )
EventSetDescriptor[ ] getEventSetDescriptors( )
MethodDescriptor[ ] getMethodDescriptors( )
```

这些方法返回的对象数组提供了一个Bean的属性、事件和方法信息。通过实现这些方法，开发人员可以明确的指明提供给用户的信息。

SimpleBeanInfo是一个提供默认实现BeanInfo接口的类，它包括了刚才说明的三个方法。开发人员可以扩展这个类，重载一个或多个方法。下面的例子描述了以前开发的Colors Bean 如何完成这个步骤。ColorsBeanInfo 是 SimpleBeanInfo 的子类。它覆盖了getPropertyDescriptors( )以便指明到底提供给Bean用户哪个属性。这个方法为rectangular属性创建了一个PropertyDescriptor对象。PropertyDescriptor对象构造函数如下：

```
PropertyDescriptor(String property, Class beanCls)
      throws IntrospectionException
```

这里第一个参数是属性的名字，第二个参数是Bean的类。

```
// A Bean information class.
package sunw.demo.colors;
import java.beans.*;
public class ColorsBeanInfo extends SimpleBeanInfo {
  public PropertyDescriptor[] getPropertyDescriptors() {
    try {
      PropertyDescriptor rectangular = new
         PropertyDescriptor("rectangular", Colors.class);
      PropertyDescriptor pd[] = {rectangular};
      return pd;
    }
    catch(Exception e) {
    }
    return null;
  }
}
```

必须在BDK\demo目录编译这个文件，或者设置CLASSPATH 变量包括c:\bdk\demo. 否则，编译器将无法找到Colors.class 文件属性。在编译完文件后，如下所示更新colors.mft文件：

```
Name: sunw/demo/colors/ColorsBeanInfo.class
Name: sunw/demo/colors/Colors.class
Java-Bean: True
```

使用JAR工具创建一个新的colors.jar文件。重起BDK，在BeanBox中创建一个Colors Bean的实例。

内省工具被设计为首先寻找BeanInfo类。如果BeanInfo类存在，其行为明确指定了提供给Bean用户的信息。否则，使用设计模式来推断应提供给Bean用户的信息。

图25-5显示窗口属性的目前状态。对比图24-3，可以看到Colors Bean中从Component继承的属性不再出现。只有rectangular 属性出现在窗口中。

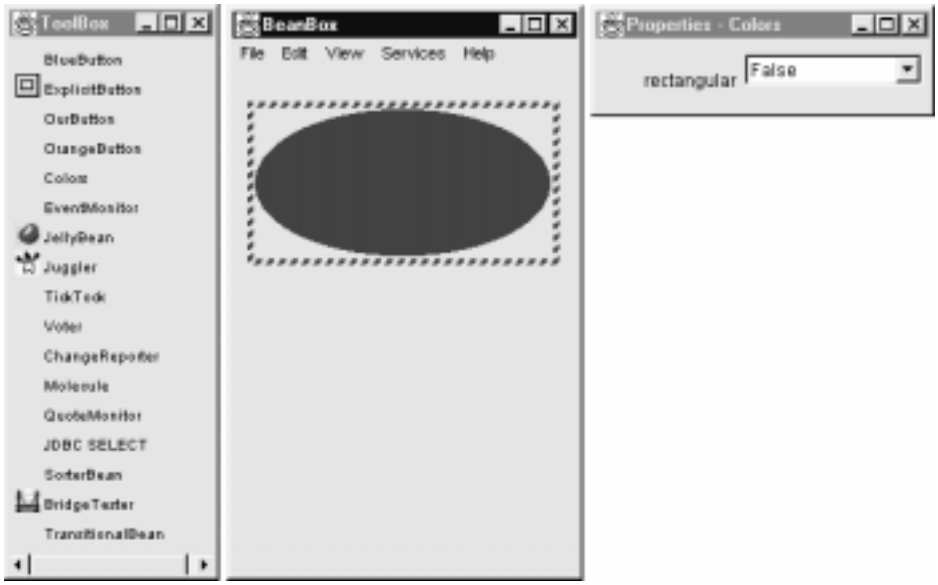


图 25-5 使用 ColorsBeanInfo 类

25.10 受 限 属 性

如果一个Bean有一个强制（Constrained）属性，当其属性值改变时生成一个事件。这个事件为PropertyChangeEvent。这个事件被发送给所有事先注册接收这样的事件通知的对象。其他对象能够拒绝改变。这个能力使Bean根据运行环境进行不同操作。关于受限属性的详细讨论超出了本书的范围。

25.11 持 久 性

所谓持久性（Persistence），是指将Bean保存在一个稳定的存储器中，以便在以后恢复的能力。这个信息对配置状态特别重要。

首先考察一下如何用BDK保存已经配置连接完毕的一组Bean构成的应用程序，回忆一下前面涉及Colors和TickTock Beans的例子。Colors Bean的rectangular属性被改成了true, TickTock Bean的interval属性被改成一秒。这些改变被保存来起来。

在BeanBox的菜单栏中选择File | Save，保存应用程序。一个对话框弹出，利用这个对

话框可以指定保存Bean和其配置参数的文件。填好文件名，点击OK按钮，退出BDK。

再次启动BDK。在BeanBox的菜单栏中选择File | Load，恢复应用程序。一个对话框弹出，利用这个对话框可以指定恢复应用程序所用的文件名。填好文件名，点击OK按钮。现在应用程序完全恢复，可以正常工作。而且Colors Bean的rectangular属性为true, TickTock Bean的interval属性等于一秒。

Java类库使用对象的序列化功能为Bean提供持久性。如果一个Bean直接或非直接的继承java.awt.Component，它就能自动的序列化, 因为这个类实现了java.io.Serializable接口。如果一个Bean没有集成一个Serializable接口，开发人员必须自己完成序列化工作。否则容器就不能保存组件的配置。

关键字transient指示Bean的数据成员不必序列化。Colors类的变量color正是这样一个例子。

25.12　定 制 器

一个开发者可以通过BDK的属性窗口改变一个Bean的属性。然而，对于一个有多个相关属性的复杂组件，属性窗口可能不是一个最好的用户界面。因此，Bean的开发者提供一个定制器（customizer），帮助其他开发人员配置软件。一个定制器提供一个在特定环境中使用组件的过程向导。定制器同时提供在线文档。一个Bean的开发人员可以灵活的开发定制器，以区别于市场上的其他产品。

25.13　Java Beans API

Java Bean的功能由java.beans包中的类和接口集合提供。本节简略介绍一下这些内容。表25-2列出了java.beans中接口，并进行了简要的功能描述。表25-3列出了java.beans中的类。

表 25-2　java.beans 中定义的接口

接口	描述
AppletInitializer	本接口中的方法用来初始化同时是小应用程序的Bean
BeanInfo	这个接口允许Bean设计者指定Bean的属性、事件和方法信息
Customizer	这个接口允许Bean设计者通过一个图形用户界面配置Bean
DesignMode	这个接口中的方法决定Bean是否按设计模式运行
PropertyChangeListener	当限制属性改变时调用接口中的方法
PropertyEditor	实现这个接口的对象允许设计者修改并显示属性值
VetoableChangeListener	当受限属性改变时调用接口中的方法
Visibility	这个接口中的方法允许Bean在一个非图形界面的环境中执行

表 25-3 在 java.beans 包中定义的类

类	描述
BeanDescriptor	这个类提供Bean的信息。它允许定制器和Bean相连
Beans	这个类获取Bean的信息
EventSetDescriptor	这个类的实例描述Bean生成的事件
FeatureDescriptor	这个类是PropetryDescriptor,EventSetDescriptor和MethodDescriptor类的父类
IndexedPropetryDescriptor	这个类的实例描述Bean的索引属性
IntrospectionException	当分析Bean时如果出现问题，生成这种类型的异常
Introspector	这个类分析一个Bean并构建描述组件的BeanInfo对象
MethodDescriptor	这个类的实例描述一个Bean的方法
ParameterDescripor	这个类的实例描述一个Bean的参数
PropertyChangeEvent	当限定属性或受限属性改变时生成这个事件。这个事件被送往注册对此感兴趣的对象，这些对象或者实现PropertyChangeListener接口或者实现VetoalbeChangeListener接口
PropertyChangeSupport	支持限制属性的Bean可以使用这个类通知PropertyChangeListener
PropertyDescripor	这个类的实例描述一个Bean的属性
PropertyEditorManager	这个类为给定类型定位PropertyEditor对象
PropertyEditorSupport	这个类提供写属性编辑器所使用的功能
PropertyVetoException	当受限属性被拒绝的时候产生此种类型的异常
SimpleBeanInfo	这个类提供写BeanInfo类时所需的功能
VetoableChangeSupport	支持强制属性的Bean可以使用这个类通知VetoableChangeListener对象

关于这些类和接口的详细说明超出了本书的范围。下面的程序举例说明Introspector, BeanDescriptor, PropertyDescriptor, EventSetDescriptor 类和BeanInfo接口。这个程序列举出本章前面小节完成的Colors Bean的属性和事件。

```
// Show properties and events.
package sunw.demo.colors;
import java.awt.*;
import java.beans.*;
public class IntrospectorDemo {
  public static void main(String args[]) {
    try {
      Class c = Class.forName("sunw.demo.colors.Colors");
      BeanInfo beanInfo = Introspector.getBeanInfo(c);
      BeanDescriptor beanDescriptor = beanInfo.getBeanDescriptor();

      System.out.println("Bean name = " +
                         beanDescriptor.getName());

      System.out.println("Properties:");
      PropertyDescriptor propertyDescriptor[] =
        beanInfo.getPropertyDescriptors();
```

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

      System.out.println("Events:");
      EventSetDescriptor eventSetDescriptor[] =
        beanInfo.getEventSetDescriptors();
      for(int i = 0; i < eventSetDescriptor.length; i++) {
        System.out.println("\t" + eventSetDescriptor[i].getName());
      }
    }
    catch(Exception e) {
      System.out.println("Exception caught. " + e);
    }
  }
}
```

这个程序的输出如下所示：

```
Bean name = Colors
Properties:
          rectangular
Events:
          propertyChange
          component
          mouseMotion
          mouse
          hierarchy
          key
          focus
          hierarchyBounds
          inputMethod
```

第 26 章　Swing

本书的第3部分介绍了如何使用AWT开发用户界面。本章将介绍一个新的可以替代AWT的图形界面类Swing。Swing是AWT的扩展，它提供了更强大和更灵活的组件集合。除了我们已经熟悉的组件如按钮、复选框和标签外，Swing还包括许多新的组件，如选项板、滚动窗口、树、表格。许多一些开发人员已经熟悉的组件，如按钮，在Swing都增加了新功能。而且，按钮的状态改变时按钮的图标也可以随之改变。

与AWT组件不同，Swing组件实现不包括任何与平台相关的代码。Swing组件是纯Java代码，因此与平台无关。一般用轻量级（lightweight）这个术语描述这类组件。

在Swing包中的类和接口的数量众多，本章只对其中的一部分简要描述。Swing包是开发人员希望自己仔细研究的部分。

下面是本书介绍的Swing组件：

类	描述
AbstractButton	按钮的抽象类
ButtonGroup	封装一组互斥的按钮
ImageIcon	封装图标
JApplet	Swing版的Applet
JButton	Swing 的按钮类
JCheckBox	Swing的复选框类
JComboBox	封装组合框(下拉式菜单和文本框的组合).
JLabel	Swing版的标记
JRadioButton	Swing版的单选按钮
JScrollPane	封装滚动窗口
JTabbedPane	封装选项窗口
JTable	封装表格控件
JTextField	Swing版的文本域
Jtree	封装树型控件

与Swing相关的类包含在javax.swing包及其子包中，如javax.swing.tree。其他与Swing相关的类和接口，本章都未加描述。

本章的其余部分介绍各个Swing组件，并通过小应用程序的例子进行说明。

26.1　JApplet

Swing的基础是JApplet类，Japplet扩展了Applet类。使用Swing的小应用程序必须是

JApplet的子类。JApplet增加了许多Applet没有的功能。例如，JApplet支持多种窗格，如内容窗格，透明窗格和根窗格。本章中的例子并没有使用太多的JApplet增强功能。但是必须强调Applet和JApplet的一个不同之处，因为本章的小应用程序例子使用到这个特性。这就是，在JApplet实例中增加一个组件，不是调用小应用程序的add()方法，而是先调用add( )增加一个内容窗格。可以通过如下方法得到内容窗格：

```
Container getContentPane( )
```

然后使用容器的add( )方法在内容窗格中增加一个组件。方法如下所示：

```
void add(comp)
```

这里，comp是增加到内容窗格中的组件。

26.2　图标和标记

在Swing中，图标由ImageIcon类封装，这个类负责将一个图片制成图标。ImageIcon类的两个构造函数如下所示：

```
ImageIcon(String filename)
ImageIcon(URL url)
```

第一个构造函数使用名为filename的文件中的图片。第二个使用资源定位符url所指示的图片。

ImageIcon类实现Icon接口，其方法如下所示：

方法	描述
int getIconHeight( )	返回图标的高度，以像素为单位
int getIconWidth( )	返回图标的宽度，以像素为单位
void paintIcon(Component comp, Graphics g, int x, int y)	在图形上下文g的x, y 位置显示图标。在组件comp中提供关于制作图标的附加信息

Swing标记是Jlabel类的实例，Jlabel类是JComponent类的子类，它可以显示文字和/或图标。Jlabel类的构造函数如下所示：

```
JLabel(Icon i)
Label(String s)
JLabel(String s, Icon i, int align)
```

其中，s和i是标记使用的文字和图标。参数align为LEFT, RIGHT, CENTER, LEADING, 或者 TRAILING。这些常数和其他Swing类使用的常数定义在SwingConstants接口中。

与标记相关的文字和图标由下列方法读写：

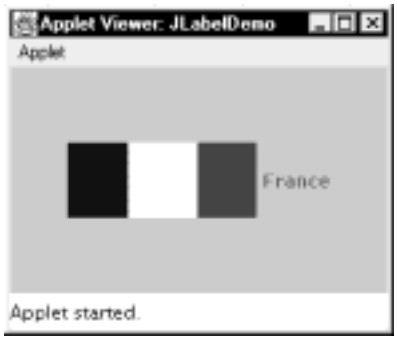
```
Icon getIcon( )
String getText( )
void setIcon(Icon i)
void setText(String s)
```

其中，i和s分别是图标和文字。

下面的例子说明如何创建并显示包含一个图标和一个字符串的标记。首先小应用程序获取内容窗格，然后利用france.gif文件创建ImageIcon对象。这个对象实例是JLabel构造函数的第二个参数。JLabel构造函数的第一个参数和最后一个参数是标记的文字和其排列方式。最后将标记加入内容窗格。

```
import java.awt.*;
import javax.swing.*;
/*
  <applet code="JLabelDemo" width=250 height=150>
  </applet>
*/

public class JLabelDemo extends JApplet {

  public void init() {
    // Get content pane
    Container contentPane = getContentPane();

    // Create an icon
    ImageIcon ii = new ImageIcon("france.gif");

    // Create a label
    JLabel jl = new JLabel("France", ii, JLabel.CENTER);

    // Add label to the content pane
    contentPane.add(jl);
  }
}
```

这个小应用程序的输出如下所示：



26.3 文 本 域

Swing的文本域被封装为JTextComponent类，JTextComponent类是JComponent的子类。它提供了Swing文本组件的公共功能。它的一个子类是JTextField, JtextField类可以编辑单行文本，它的构造函数如下所示：

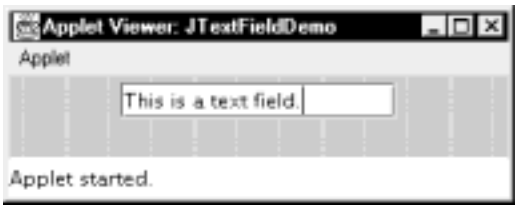
```
JTextField( )
JTextField(int cols)
JTextField(String s, int cols)
JTextField(String s)
```

其中，s 是要显示的字符串，cols是文本域中的列数。

下面的例子说明如何创建一个文本域。小应用程序获得内容窗格，然后分配一个流布局作为布局管理器。接着，创建一个JTextField对象，将其加入内容窗格。

```
import java.awt.*;
import javax.swing.*;
/*
  <applet code="JTextFieldDemo" width=300 height=50>
  </applet>
*/

public class JTextFieldDemo extends JApplet {
  JTextField jtf;

  public void init() {

    // Get content pane
    Container contentPane = getContentPane();
    contentPane.setLayout(new FlowLayout());

    // Add text field to content pane
    jtf = new JTextField(15);
    contentPane.add(jtf);
  }
}
```

这个小应用程序的输出如下所示：



26.4　按　钮

Swing的按钮相对于AWT中Button类提供了更多功能。例如，可以用一个图标修饰Swing的按钮。Swing的按钮是AbstractButton的子类，AbstractButton类扩展JComponent类。AbstractButton类包含多种方法，用于控制按钮行为，检查复选框和单选按钮。例如，当按钮被禁止，按下或选择时，可以将其显示为不同的图标。还可以定义一个图标为“rollover”图标，当鼠标移动到按钮上时显示。下面是控制这类行为的方法：

```
void setDisabledIcon(Icon di)
void setPressedIcon(Icon pi)
```

```
void setSelectedIcon(Icon si)
void setRolloverIcon(Icon ri)
```

其中，di，pi，si和ri是不同状态下使用的图标。

可以通过下列方法读写与按钮相关的文字：

```
String getText( )
void setText(String s)
```

其中，s是与按钮相关的文字。

AbstractButton抽象类的子类在按钮被按下时生成行为事件。通过如下的方法注册和注销这些事件的监听器：

```
void addActionListener(ActionListener al)
void removeActionListener(ActionListener al)
```

其中，al是动作监听器。

AbstractButton是按钮、复选框和单选按钮的父类。按钮、复选框和单选按钮将在下面的章节中介绍。

26.4.1 JButton类

JButton类提供一个按钮的功能。JButton类允许用图标，或字符串或两者同时与下压式按钮相关联的功能。类的构造函数如下所示：

```
JButton(Icon i)
JButton(String s)
JButton(String s, Icon i)
```

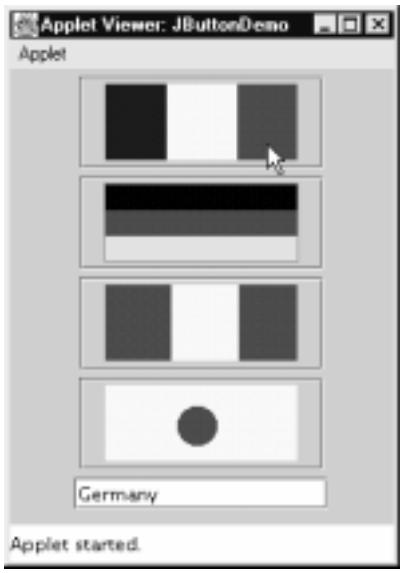
其中s和i是按钮使用的字符串和图标。

下面的例子显示四个按钮和一个文本域。每个按钮显示为一个图标，使用的图标是一个国家的国旗。当按下按钮时，在文本域中显示国家的名字。小应用程序程序先获取内容窗格，然后设置窗格的布局管理器。创建四个图形按钮，加入内容窗格。然后，小应用程序注册接收按钮产生的事件。创建文本域，将其加入小应用程序。最后，动作事件的处理程序将与按钮相关的命令字符串显示在文本域中。

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
  <applet code="JButtonDemo" width=250 height=300>
  </applet>
*/

public class JButtonDemo extends JApplet
implements ActionListener {
  JTextField jtf;

  public void init() {

    // Get content pane
```

```
    Container contentPane = getContentPane();
    contentPane.setLayout(new FlowLayout());

    // Add buttons to content pane
    ImageIcon france = new ImageIcon("france.gif");
    JButton jb = new JButton(france);
    jb.setActionCommand("France");
    jb.addActionListener(this);
    contentPane.add(jb);

    ImageIcon germany = new ImageIcon("germany.gif");
    jb = new JButton(germany);
    jb.setActionCommand("Germany");
    jb.addActionListener(this);
    contentPane.add(jb);

    ImageIcon italy = new ImageIcon("italy.gif");
    jb = new JButton(italy);
    jb.setActionCommand("Italy");
    jb.addActionListener(this);
    contentPane.add(jb);

    ImageIcon japan = new ImageIcon("japan.gif");
    jb = new JButton(japan);
    jb.setActionCommand("Japan");
    jb.addActionListener(this);
    contentPane.add(jb);

    // Add text field to content pane
    jtf = new JTextField(15);
    contentPane.add(jtf);
  }

  public void actionPerformed(ActionEvent ae) {
    jtf.setText(ae.getActionCommand());
  }
}
```

小应用程序的输出如下所示：



26.4.2 复选框

JCheckBox类提供复选框的功能，它是AbstractButton抽象类的子类。其构造函数如下所示：

```
JCheckBox(Icon i)
JCheckBox(Icon i, boolean state)
JCheckBox(String s)
JCheckBox(String s, boolean state)
JCheckBox(String s, Icon i)
JCheckBox(String s, Icon i, boolean state)
```

其中，i是按钮的图标。文字由s指定。如果state为true，复选框在初始化时状态为被选中，否则相反。

通过下列方法改变复选框状态：

```
void setSelected(boolean state)
```

其中，如果复选框的state为true，则复选框被选中。

下面的例子说明如何创建包括四个复选框和一个文本域的小应用程序。当选中一个按钮，其对应文字显示在文字域中。JApplet对象的内容窗格包括在内，同时分配了一个流布局给布局管理器。然后，在内容面板中加入四个按钮的复选框，并为正常(normal)，滚过（rollover）和选中(selected)状态分配图标。小应用程序注册以接受事件。最后在内容窗格中加入文本域。

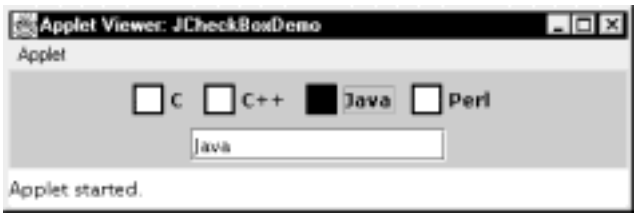
当选中或取消复选框时，生成一个项目事件。这个事件由itemStateChanged( )处理。在itemStateChanged( )内部，getItem( )方法获取产生事件的JCheckBox对象。getText( )方法获取复选框的文字，并用这个文字设置文本域。

```
import java.awt.*;
import java.awt.event.*;
```

```
import javax.swing.*;
/*
  <applet code="JCheckBoxDemo" width=400 height=50>
  </applet>
*/

public class JCheckBoxDemo extends JApplet
implements ItemListener {
  JTextField jtf;

  public void init() {

    // Get content pane
    Container contentPane = getContentPane();
    contentPane.setLayout(new FlowLayout());

    // Create icons
    ImageIcon normal = new ImageIcon("normal.gif");
    ImageIcon rollover = new ImageIcon("rollover.gif");
    ImageIcon selected = new ImageIcon("selected.gif");

    // Add check boxes to the content pane
    JCheckBox cb = new JCheckBox("C", normal);
    cb.setRolloverIcon(rollover);
    cb.setSelectedIcon(selected);
    cb.addItemListener(this);
    contentPane.add(cb);

    cb = new JCheckBox("C++", normal);
    cb.setRolloverIcon(rollover);
    cb.setSelectedIcon(selected);
    cb.addItemListener(this);
    contentPane.add(cb);

    cb = new JCheckBox("Java", normal);
    cb.setRolloverIcon(rollover);
    cb.setSelectedIcon(selected);
    cb.addItemListener(this);
    contentPane.add(cb);

    cb = new JCheckBox("Perl", normal);
    cb.setRolloverIcon(rollover);
    cb.setSelectedIcon(selected);
    cb.addItemListener(this);
    contentPane.add(cb);

    // Add text field to the content pane
    jtf = new JTextField(15);
    contentPane.add(jtf);
  }

  public void itemStateChanged(ItemEvent ie) {
    JCheckBox cb = (JCheckBox)ie.getItem();
    jtf.setText(cb.getText());
  }
```

```
}
```

小应用程序的输出如下所示：



26.4.3 单选按钮

单选按钮由JRadioButton类支持，JRadioButton也是AbstractButton抽象类的子类。其构造函数如下所示：

```
JRadioButton(Icon i)
JRadioButton(Icon i, boolean state)
JRadioButton(String s)
JRadioButton(String s, boolean state)
JRadioButton(String s, Icon i)
JRadioButton(String s, Icon i, boolean state)
```

其中，i是按钮的图标，文字由s指定，如果state为true，按钮被初始化为选中状态，否则为非选中状态。

单选按钮必须配置成组。每次一个组内只能选中一个按钮。例如，如果用户按下组内的一个按钮，则组内先前被选中的按钮自动变成非选中状态。实例化ButtonGroup类以创建一个按钮组，为此要调用其默认构造函数。然后使用下面的方法把元素加入按钮组。

```
void add(AbstractButton ab)
```

其中，ab是加入单选框组的按钮的引用。

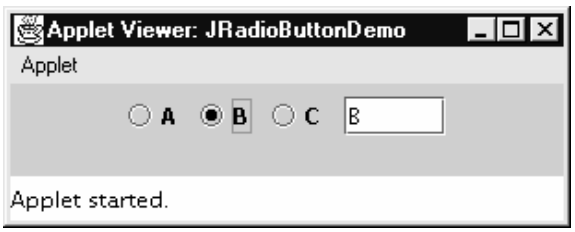
下面的例子说明如何使用单选按钮。创建三个单选按钮和一个文本域。当一个单选按钮被按下时，其文字显示在文本域中。首先包括JApplet对象的内容窗格，分配流布局给布局管理器。然后，在内容窗格中加入三个单选按钮，再定义一个按钮组，将按钮加入。最后在内容窗格中加入文本域。

单选按钮按下产生的行为事件，由actionPerformed( )处理。getActionCommand( ) 方法获取与单选按钮相关的文字，并用此文字设置文本域。

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
  <applet code="JRadioButtonDemo" width=300 height=50>
  </applet>
*/

public class JRadioButtonDemo extends JApplet
implements ActionListener {
  JTextField tf;
```

```
  public void init() {

    // Get content pane
    Container contentPane = getContentPane();
    contentPane.setLayout(new FlowLayout());

    // Add radio buttons to content pane
    JRadioButton b1 = new JRadioButton("A");
    b1.addActionListener(this);
    contentPane.add(b1);

    JRadioButton b2 = new JRadioButton("B");
    b2.addActionListener(this);
    contentPane.add(b2);

    JRadioButton b3 = new JRadioButton("C");
    b3.addActionListener(this);
    contentPane.add(b3);

    // Define a button group
    ButtonGroup bg = new ButtonGroup();
    bg.add(b1);
    bg.add(b2);
    bg.add(b3);

    // Create a text field and add it
    // to the content pane
    tf = new JTextField(5);
    contentPane.add(tf);
  }

  public void actionPerformed(ActionEvent ae) {
    tf.setText(ae.getActionCommand());
  }
}
```

小应用程序的输出如下所示：



26.5　组 合 框

Swing通过JComboBox类支持组合框（combo box，一个文本域和下拉列表的组合），JComboBox类是JComponent的子类。组合框通常显示一个可选条目，但可允许用户在一个

下拉列表中选择多个不同条目。用户也可以在文本域内键入选择项。JComboBox的两个构造函数如下所示：

```
JComboBox( )
JComboBox(Vector v)
```

其中，v 是初始化选择框的矢量。

使用addItem( ) 方法在列表中增加选项，其形式如下：

```
void addItem(Object obj)
```

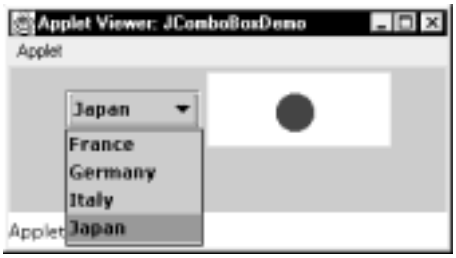
其中，obj 是加入组合框的对象。

下面的例子包括一个组合框和一个标签。标签显示一个图标。组合框的可选条目是“France”，“Germany”，“Italy”和“Japan”。当选择了一个图标，标签更新为这个国家的国旗。

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
  <applet code="JComboBoxDemo" width=300 height=100>
  </applet>
*/

public class JComboBoxDemo extends JApplet
implements ItemListener {
  JLabel jl;
  ImageIcon france, germany, italy, japan;

  public void init() {

    // Get content pane
    Container contentPane = getContentPane();
    contentPane.setLayout(new FlowLayout());

    // Create a combo box and add it
    // to the panel
    JComboBox jc = new JComboBox();
    jc.addItem("France");
    jc.addItem("Germany");
    jc.addItem("Italy");
    jc.addItem("Japan");
    jc.addItemListener(this);
    contentPane.add(jc);

    // Create label
    jl = new JLabel(new ImageIcon("france.gif"));
    contentPane.add(jl);
  }

  public void itemStateChanged(ItemEvent ie) {
    String s = (String)ie.getItem();
    jl.setIcon(new ImageIcon(s + ".gif"));
```

```
  }
}
```

小应用程序的输出如下所示：



26.6　选 项 窗 格

选项窗格（tabbed pane）组件表现为一组文件夹。每个文件夹都有标题。当用户使用文件夹时，显示它的内容。每次只能选择组中的一个文件夹。选项窗格一般用作设置配置选项。

选项窗格被封装为JTabbedPane类，JTabbedPane是JComponent的子类。使用默认构造函数时，选项的定义方法如下所示：

```
void addTab(String str, Component comp)
```

其中，str是标签的标题，comp是应加入标签的组件。典型情况下，加入的是JPanel 或其子类。

在小应用程序中使用选项窗格的一般过程如下所示：

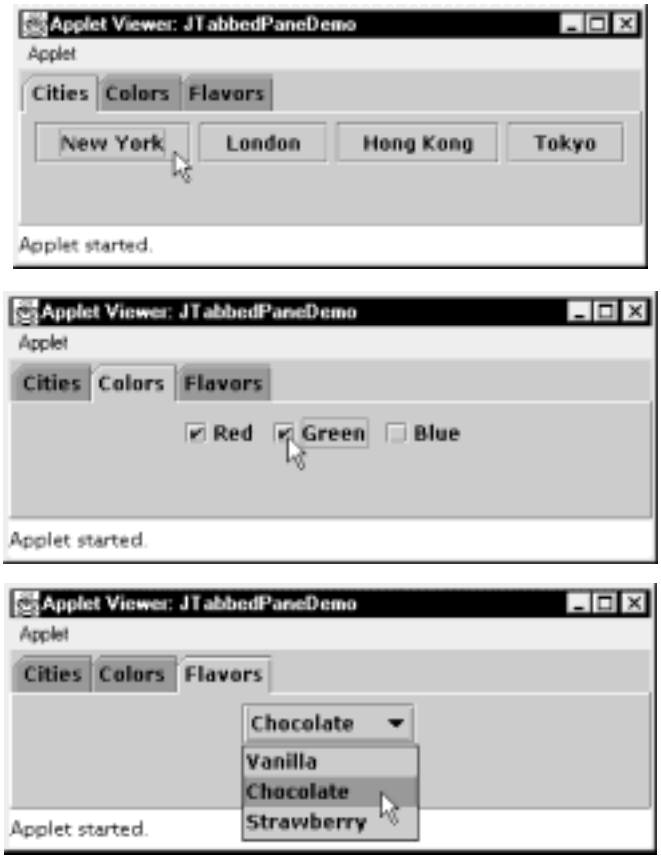
1. 创建JTabbedPane对象。
2. 调用addTab( ) 方法在窗格中增加一个标签（这个方法的参数是标签的标题和它包含的组件）。
3. 重复步骤2，增加标签。
4. 将选项窗格加入小应用程序的内容窗格。

下面的例子说明如何创建一个选项窗格。第一个标签的标题为“Cities”，包含四个按钮。每个按钮显示一个城市名。第一个标签的题目为“Colors”，包含三个按钮的复选框，每个按钮显示一种颜色名。第三个标签的标题为“Flavors”包含一个组合框，用户可以选择一种风格。

```
import javax.swing.*;
/*
  <applet code="JTabbedPaneDemo" width=400 height=100>
  </applet>
*/


public class JTabbedPaneDemo extends JApplet {
```

```
  public void init() {

    JTabbedPane jtp = new JTabbedPane();
    jtp.addTab("Cities", new CitiesPanel());
    jtp.addTab("Colors", new ColorsPanel());
    jtp.addTab("Flavors", new FlavorsPanel());
    getContentPane().add(jtp);
  }
}

class CitiesPanel extends JPanel {

  public CitiesPanel() {

    JButton b1 = new JButton("New York");
    add(b1);
    JButton b2 = new JButton("London");
    add(b2);
    JButton b3 = new JButton("Hong Kong");
    add(b3);
    JButton b4 = new JButton("Tokyo");
    add(b4);
  }
}

class ColorsPanel extends JPanel {

  public ColorsPanel() {

    JCheckBox cb1 = new JCheckBox("Red");
    add(cb1);
    JCheckBox cb2 = new JCheckBox("Green");
    add(cb2);
    JCheckBox cb3 = new JCheckBox("Blue");
    add(cb3);
  }
}

class FlavorsPanel extends JPanel {

  public FlavorsPanel() {

    JComboBox jcb = new JComboBox();
    jcb.addItem("Vanilla");
    jcb.addItem("Chocolate");
    jcb.addItem("Strawberry");
    add(jcb);
  }
}
```

小应用程序的输出如下所示：



26.7　滚 动 窗 格

滚动窗格组件是一个可以容纳其他组件的矩形区域，在必要的时候提供水平和/或垂直的滚动条。Swing中的滚动窗格由JScrollPane类实现，JScrollPane扩展了JComponent类。其构造函数如下所示：

```
JScrollPane(Component comp)
JScrollPane(int vsb, int hsb)
JScrollPane(Component comp, int vsb, int hsb)
```

其中，comp 是加入滚动窗格的组件。vsb 和hsb 为整型常数，定义滚动窗口的水平和垂直条。这些常数由ScrollPaneConstants 接口定义。这些常数的例子如下所示：

常数	描述
HORIZONTAL_SCROLLBAR_ALWAYS	总是提供水平滚动条
HORIZONTAL_SCROLLBAR_AS_NEEDED	在需要时，提供水平滚动条
VERTICAL_SCROLLBAR_ALWAYS	总是提供垂直滚动条
VERTICAL_SCROLLBAR_AS_NEEDED	在需要时，提供垂直滚动条

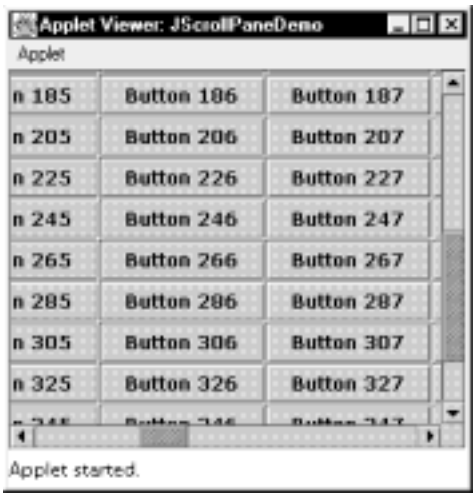
按下列步骤在小应用程序中增加滚动窗口：

1. 创建JComponent对象。
2. 创建JScrollPane 对象（构造函数的参数指定组件和水平、垂直滚动条的策略）。
3. 将滚动窗格加入小应用程序的内容窗格中。

下面的例子说明滚动窗格。首先，获取创建JApplet对象的内容窗格，给布局管理器分配一个边界布局。然后，创建JPanel对象，加入400个按钮，这400个按钮排列成20列。将面板加到滚动窗格中，将滚动窗格加入内容窗格。面板中出现水平和垂直滚动条。现在可以用滚动条来查看所有的按钮。

```
import java.awt.*;
import javax.swing.*;
/*
  <applet code="JScrollPaneDemo" width=300 height=250>
  </applet>
*/

public class JScrollPaneDemo extends JApplet {

  public void init() {

    // Get content pane
    Container contentPane = getContentPane();
    contentPane.setLayout(new BorderLayout());

    // Add 400 buttons to a panel
    JPanel jp = new JPanel();
    jp.setLayout(new GridLayout(20, 20));
    int b = 0;
    for(int i = 0; i < 20; i++) {
      for(int j = 0; j < 20; j++) {
        jp.add(new JButton("Button " + b));
        ++b;
      }
    }

    // Add panel to a scroll pane
    int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
    int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
    JScrollPane jsp = new JScrollPane(jp, v, h);

    // Add scroll pane to the content pane
    contentPane.add(jsp, BorderLayout.CENTER);
  }
}
```

小应用程序的输出如下所示：



26.8 Trees

树对象提供了用树型结构分层显示数据的视图。用户可以扩展或收缩视图中的单个子树。树由Swing中的JTree类实现，JTree是JComponent的子类。其构造函数如下所示：

```
JTree(Hashtable ht)
JTree(Object obj[ ])
JTree(TreeNode tn)
JTree(Vector v)
```

第一种模式创建一个树，散列表ht中的每个元素是树的一个子节点。第二种模式中对象数组obj中的每一个元素都是树的子节点。第三种模式中树节点tn是树的根节点。最后一种模式中向量v中的元素是树的子节点。

当节点扩展或收缩时，JTree对象生成事件。addTreeExpansionListener( )和removeTreeExpansionListener( )方法注册或注销监听这些通知的监听器。其使用方法如下所示：

```
void addTreeExpansionListener(TreeExpansionListener tel)
void removeTreeExpansionListener(TreeExpansionListener tel)
```

其中，tel 是监听器对象。

getPathForLocation( )方法将鼠标点击点转换为树的路径，其使用方法如下：

```
TreePath getPathForLocation(int x, int y)
```

其中x和y是鼠标点击的坐标。返回值是一个TreePath对象，TreePath对象封装了用户选择的树节点的信息。

TreePath类封装树中特定节点的路径信息。这个类提供了几个构造函数和方法。在本书中，只使用了其中的toString( )方法，它返回一个等价于树路径的字符串。

TreeNode接口定义了获取树节点信息的方法。例如，它能够得到关于父节点的引用，或者一个子节点的枚举。MutableTreeNode接口扩展了TreeNode接口。它定义了插入和删除子节点或者改变父节点的方法。

DefaultMutableTreeNode类实现了MutableTreeNode接口。它代表树中的一个节点。其构造函数如下所示：

```
DefaultMutableTreeNode(Object obj)
```

其中，obj 是包括在树节点中的对象。新的节点既没有父节点也没有子节点。

要创建树节点的层次结构，需使用DefaultMutableTreeNode的add( )方法。其使用方式如下所示：

```
void add(MutableTreeNode child)
```

其中，child 是一个可变的树节点，被当作当前节点的子节点插入。

树的扩展事件由javax.swing.event包中的TreeExpansionEvent类描述。这个类的getPath( )

方法返回一个TreePath对象，TreePath对象描述了改变节点的路径。其使用方式如下所示：

```
TreePath getPath( )
```

TreeExpansionListener 接口提供下列的两个方法：

```
void treeCollapsed(TreeExpansionEvent tee)
void treeExpanded(TreeExpansionEvent tee)
```

其中，tee 是树的扩展事件。当一个子树隐藏时，调用第一个方法。当一个子树变为可见时，调用第二个方法。

下面是在小应用程序中使用树组件时应遵循的步骤：

1. 创建一个JTree 对象。
2. 创建一个JScrollPane 对象（构造函数的参数指定树和水平和垂直滚动条的策略）。
3. 将树加入滚动窗口。
4. 将滚动窗口加入小应用程序的内容面板。

下面的例子说明如何创建一个树，并识别其上的鼠标点击。init( )方法获取小应用程序的内容窗格。创建一个标志为“Options”的DefaultMutableTreeNode对象。这是树分层结构的顶层。创建其他的树节点，调用add( )方法将这些节点加入树。这个树的顶点的引用作为参数提供给JTree的构造函数。然后，将树当作参数提供给Jscrollpane的构造函数。将滚动窗格加入小应用程序。接着，创建一个文本域，并加入小应用程序。鼠标点击事件的信息在此文本域内显示。为了接收树组件上的鼠标事件，调用JTree对象的addMouseListener( )方法。这个方法的第一个参数是一个匿名的内部类，这个内部类扩展MouseAdapter，并重载mouseClicked( )方法。

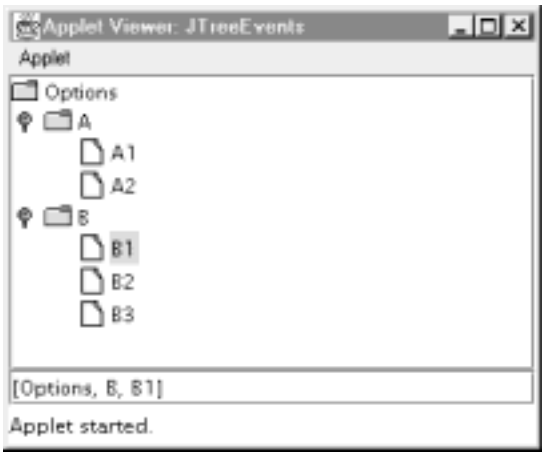
doMouseClicked( )方法处理鼠标点击事件。它调用getPathForLocation( )方法将鼠标点击的坐标转换成TreePath对象。如果鼠标点击并没有选择到一个树节点，则这个方法的返回值为null。否则，树的路径被转换成字符串，显示在文本域中。

```
Import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.tree.*;
/*
  <applet code=”JtreeEvents” width=400 height=200>
  </applet>
*/


public class JtreeEvents extends Japplet {
  Jtree tree;
  JtextField jtf;

  public void init() {
```

```
// Get content pane
Container contentPane = getContentPane();

// Set layout manager
contentPane.setLayout(new BorderLayout());

// Create top node of tree
DefaultMutableTreeNode top = new DefaultMutableTreeNode("Options");

// Create subtree of "A"
DefaultMutableTreeNode a = new DefaultMutableTreeNode("A");
top.add(a);
DefaultMutableTreeNode a1 = new DefaultMutableTreeNode("A1");
a.add(a1);
DefaultMutableTreeNode a2 = new DefaultMutableTreeNode("A2");
a.add(a2);

// Create subtree of "B"
DefaultMutableTreeNode b = new DefaultMutableTreeNode("B");
top.add(b);
DefaultMutableTreeNode b1 = new DefaultMutableTreeNode("B1");
b.add(b1);
DefaultMutableTreeNode b2 = new DefaultMutableTreeNode("B2");
b.add(b2);
DefaultMutableTreeNode b3 = new DefaultMutableTreeNode("B3");
b.add(b3);

// Create tree
tree = new Jtree(top);

// Add tree to a scroll pane
int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
JscrollPane jsp = new JscrollPane(tree, v, h);

// Add scroll pane to the content pane
contentPane.add(jsp, BorderLayout.CENTER);

// Add text field to applet
jtf = new JtextField("", 20);
contentPane.add(jtf, BorderLayout.SOUTH);

// Anonymous inner class to handle mouse clicks
tree.addMouseListener(new MouseAdapter() {
```

```
      public void mouseClicked(MouseEvent me) {
        doMouseClicked(me);
      }
    });
  }

  void doMouseClicked(MouseEvent me) {
    TreePath tp = tree.getPathForLocation(me.getX(), me.getY());
    if(tp != null)
      jtf.setText(tp.toString());
    else
      jtf.setText("");
  }
}
```

小应用程序的输出如下所示：



在文字域中显示的字符串描述类从树对待顶部节点到所选节点的路径。

26.9　表　格

表格（table）组件提供了以行和列的形式显示数据的视图。可以在表格的列边界上拖曳鼠标以改变列的大小，也可以将列拖放到新位置。表格由JTable类实现，JTable类是JComponent的子类，它的一个构造函数如下所示：

```
JTable(Object data[ ][ ], Object colHeads[ ])
```

其中，data是一个三维数组，包含要显示的信息，colHeads是一个一维数组，其中信息是列标头。

按下列步骤在小应用程序中加入表格：

1. 创建一个JTable 对象。

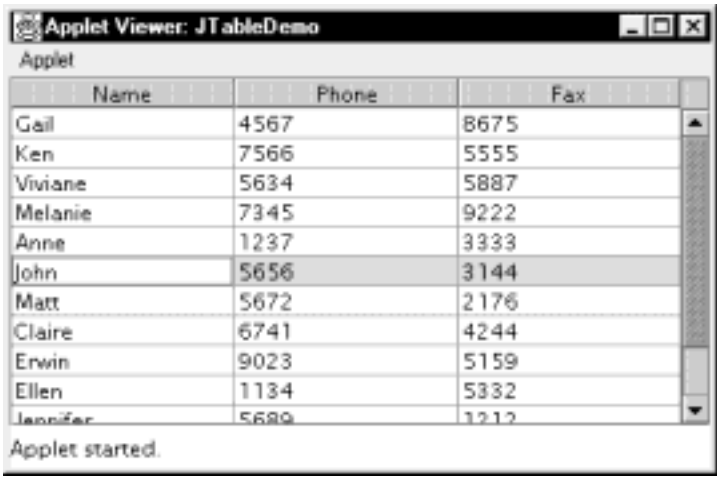
2. 创建一个JScrollPane 对象（构造函数中的参数指定表格及水平和垂直滚动条的策略）。
3. 将表格加入滚动窗格。
4. 将滚动窗格加入小应用程序的内容窗格中。

下面的例子说明如何创建并使用表格。首先获取创建JApplet对象所需的内容窗格，分配一个边界布局给布局管理器。创建一个一维的字符串数组作列标头。本例中的表格有三列。为表格元素创建一个两维的字符串数组。数组中的每个元素都是一个三个字符串的数组。将这些数组传递给JTable的构造函数。然后将表格加入滚动窗格，将滚动窗格加入内容窗格。

```
import java.awt.*;
import javax.swing.*;
/*
  <applet code="JTableDemo" width=400 height=200>
  </applet>
*/


public class JTableDemo extends JApplet {

  public void init() {

    // Get content pane
    Container contentPane = getContentPane();

    // Set layout manager
    contentPane.setLayout(new BorderLayout());

    // Initialize column headings
    final String[] colHeads = { "Name", "Phone", "Fax" };

    // Initialize data
    final Object[][] data = {
      { "Gail", "4567", "8675" },
      { "Ken", "7566", "5555" },
      { "Viviane", "5634", "5887" },
      { "Melanie", "7345", "9222" },
      { "Anne", "1237", "3333" },
      { "John", "5656", "3144" },
      { "Matt", "5672", "2176" },
      { "Claire", "6741", "4244" },
      { "Erwin", "9023", "5159" },
      { "Ellen", "1134", "5332" },
      { "Jennifer", "5689", "1212" },
      { "Ed", "9030", "1313" },
      { "Helen", "6751", "1415" }
    };

    // Create the table
    JTable table = new JTable(data, colHeads);
```

```
    // Add tree to a scroll pane
    int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
    int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
    JScrollPane jsp = new JScrollPane(table, v, h);

    // Add scroll pane to the content pane
    contentPane.add(jsp, BorderLayout.CENTER);
  }
}
```

小应用程序的输出如下所示：



26.10　深入研究Swing

在前面已经提到，Swing是一个巨大的系统，它有许多值得仔细研究的特性。例如Swing提供工具栏、工具提示和进度栏。同时，Swing组件能够提供可插入的外观感觉，这意味着可以方便替换一个元素的外观和行为。这种替换可以动态实现。用户甚至可以设计自己的外观和感觉。坦白的讲，在不久的将来，利用Swing机制实现GUI组件将完全取代AWT类，因此开发人员应该从现在开始熟悉Swing。

Swing 是Java基础类JFC（Java Foundation Calsses）的一部分。开发者应该深入研究其他JFC特性。Accessibility API可用来开发方便残疾人士使用的应用程序。在本书前面提到的Java 2-D API可提供强大的图形、文本功能。拖拉（Drag-and-Drop）API允许Java和非Java程序之间的信息交换。

第 27 章　从 C++到 Java

本章介绍当C++的开发人员转移到Java时必须注意的一些问题。许多Java程序员都有C++背景，因此利用原来的C++的技巧、技术和源代码是很自然的事情。虽然C++和Java是为不同的环境而设计的，但许多编程技术，算法和优化机制是相同的。然而，正如本书的第一部分所述，Java并不是一个Internet版本的C++。两种编程语言有许多相同之处，但它们仍旧存在着许多不同。本章将探讨这些不同之处并描述如何处理其中比较有挑战性的问题。

27.1　C++ 和Java的区别

在讨论特定环境之前，先探讨一下C++ 和Java的基本区别。这些不同之处大致分为三类：

- Java不支持的C++特性
- Java的独特之处
- C++和Java都有但是却不相同的特性

这里将分别讨论这三类区别。

27.1.1　Java摈弃的C++内容

有些C++的内容特性Java不再支持。在某些情况下，一个特殊的C++特性和Java的运行环境不相关。另一些情况下，Java的设计者舍弃了C++中某些重复的内容。还有一些情况是，出于对Internet applet的运行安全问题的考虑，Java不再支持某些C++特点。

C++与Java之间的最大不同可能是Java不再支持指针。指针使C++语言成为功能最强大最重要的一种编程语言。但同时指针在使用不正确的情况下也是C++中最危险的部分。Java不支持指针的原因主要有两点：

- 指针本身就非常不安全。例如，使用C++的状态指针，可以访问程序代码和数据段外的内存地址。一个恶意程序可以利用这个特点破坏系统，完成非法访问（如获取口令等），或者违反安全限制。
- 即使指针可以被限制在Java的运行系统中（这在理论上是可行的，因为Java程序是解释执行的），Java的设计者相信它们仍然是麻烦所在。

注意：既然Java中不存在指针，那么也不存在->操作符。

这里还列举了一些非常重要的“省略”：

- Java不再包含结构或者联合。在可以完全包含它们的类出现之后，这些结构成为冗余部分。
- Java不再支持操作符重载。操作符重载在某些情况下造成类C++程序的含糊不清，Java的设计人员感觉它带来的麻烦要比它带来的好处多。
- Java不再包括预处理，也不再支持预处理指令。预处理在C++语言中已经不如在C中那么重要，Java的设计者认为是彻底消除预处理的时候了。
- Java不支持自动的类型转换，因为这种类型转换导致精度降低。例如当从长整型转换为整型时，必须显式强制执行类型转换。
- 在Java中的代码必须封装在一个或多个类中。因此Java中不再包含所谓的全局变量或全局函数。
- Java不再允许默认参数。在C++中，当过程调用时存在没有对应值的参数时可以使用一个预先指定的值。Java不再允许这样的操作。
- Java不支持多重继承，即不允许一个子类继承多个父类。
- 虽然Java支持构造函数，但不再支持析构函数。但是，Java增加了finalize( )函数。
- Java不再支持typedef。
- 在Java中不再可能声明无符号整数。
- Java不再支持goto语句。
- Java不再有delete操作符 。
- Java中的<<and>>不再重载I/O操作。
- Java中，对象只能由引用传递，C++中对象可由值或引用传递。

27.1.2　Java的新特性

Java中的许多特性是C++中没有的。其中最重要的三个方面是多线程、包和接口，还有其他的许多独特之处都丰富了Java编程环境。

- 正如本书的前面章节中讨论过的那样，多线程允许两个或者多个相同线程并发运行。而且，这是一种在语言级支持的并发机制。C++中没有类似的机制。如果需要并发执行一个C++程序，必须利用操作系统的功能手工启动。虽然这两种方法都可以同时执行两个或者多个线程，但Java的方法清楚而且便于使用。
- C++中没有可以与Java包对应的特性。最近似的是用一个公用头文件的一组库函数。然而，在C++中构建和使用函数库与在Java中构建包和使用包是完全不同的。
- Java的接口与C++的抽象类相似（C++中的抽象类是包括至少一个纯虚函数的类）。例如，C++的抽象类和Java的接口都不能创建实例。两者都用于指定一个子类实现的一致接口。两者之间最大的不同之处在于接口更清晰的表明了这个概念。
- Java提供一个流线型的内存分配机制。与C++类似，Java支持new这个关键字。但是，不支持delete关键字。当对象的最后一个引用撤销时，对象本身被自动地删除，并进行内存垃圾回收。
- Java丢弃了C++的标准库，将它替换成自己的API类集合。它们功能上有相似之处，但名字和参数有显著不同。同时，因为所有的Java API库都是面向对象的，而C++

库只有部分是，所以库例程调用的方式不同。

- Java增强了break和continue语句以接收标记。
- Java中的char类型是16位的Unicode字符。这与C++中的wchar_t类型相似。使用Unicode字符增强了代码的可移植性。
- Java增加了>>> 操作，完成无符号的右移。
- 除支持单行和多行的注释之外，Java增加了第三种注释方法：文档注释。文档注释以/**开头，以*/结尾。
- Java包含一个内置的字符串类型叫做String。String在某种程度上和C++提供的标准string类很相象。当然C++中的string只有在程序中声明后方可使用，它并不是内置的类型。

27.1.3 Java和C++的不同特性

下面是Java和C++共有的特性，但是两种语言在处理上略有不同：

- Java和C++都支持布尔类型的数据，但Java实现true和false的方式和C++不同。在C++中，true是非零值，false是零。在Java中，true和false都是预先定义好的常量，而且是一个布尔表达式能得到的惟一的两个值。虽然C++也定义了true和false，并指定为布尔变量，但C++自动将非零值转换为true，零值转换为false。这种情况在Java中不会出现。
- 在创建C++类时，访问说明符应用到一组声明中。而Java中，访问说明符仅应用于其限定的声明中。
- C++支持异常处理，这与Java类似。但是在C++中无需捕捉一个被引发的异常。

有了这些添加、删除和区别做背景，本章的余下部分进一步描述在将C++代码转换为Java代码时必须处理的关键问题。

27.2 消 除 指 针

在将C++程序改写为Java程序时，大部分的改写工作都是由指针引起的。大多数C++代码依赖指针操作。离开指针几乎无法编写任何重要的C++程序。

在C++代码中可能有下列4种常规的指针用法：

- 做函数的参数。虽然C++支持引用参数，但有许多遗留下来的用C语言编写的代码。C不支持引用参数。在C语言中，如果函数需要改变其自变量的值，它必须显式将指针传递给变量。因此，在从C移植过来的C++代码中，指针参数非常普遍。同时，在某些情况下，一些函数库被C和C++代码共享，因此不能使用引用参数。而且许多支持C++的标准库函数向后兼容C。当这些基于C语言的函数需要一个地址参数时，一般情况下使用指针做自变量。于是在这些参数内部，就通过指针访问这个自变量。
- 提供更有效的实现结构——特别是数组下标。例如，使用指针在数组中移动要比使

用数组下标更为有效。现在的编译器实现了高效的优化策略，指针仍然能够提供显著的性能提高。因此，在C++中使用指针访问数组是非常普遍的现象。

- 支持内存分配。在C++中，分配内存时，返回的是内存的地址（也就是一个指针）。这个地址必须是一个指针变量。一旦这样做了，这个指针可以指向所分配的内存的任何部分或是其他的任何地方，在这种情况下，完全由指针算法决定。在Java中，当用new分配一个对象时，返回的是对象的引用。这个引用必须是一个兼容类型的引用变量。Java的引用变量隐式指向用new操作符分配的对象，因为引用变量不能象C++中的指针那样自由操作。他们不能指向Java运行环境以外的内存。
- 提供对任意机器地址的访问，可以调用一个ROM例程或是直接读写内存。因为Java有意地拒绝类似行为，这种指针用法在Java中没有对等方法。如果是编写一个非小应用程序的应用程序，通常可以使用Java的本机功能（在本书的第一部分中描述过）来获取可以访问这样的系统资源的例程。

现在考虑一下基于指针的C++代码改写成Java代码的两种情况。

27.2.1　转换指针参数

在多数情况下，将使用指针的C++函数转换为对等的Java方法是非常容易的。因为Java使用引用方式传递对象，某些时候只需要将C++中的指针操作符去掉。例如，这个C++程序逆转Coord对象的符号，Coord对象存储的是笛卡儿坐标。函数reverseSign( )传递一个指针给将逆转的Coord对象。可以看到，C++使用*，&和->指针操作符完成操作。

```
// Reverse the signs of a coordinate - C++ version.
#include <iostream>
using namespace std;

class Coord {
public:
  int x;
  int y;
};

// Reverse the sign of the coordinates.
void reverseSign(Coord *ob) {
  ob->x = -ob->x;
  ob->y = -ob->y;
}

int main()
{
  Coord ob;

  ob.x = 10;
  ob.y = 20;

  cout << "Original values for ob: ";
  cout << ob.x << ", " << ob.y << "\n";

  reverseSign(&ob);
```

```
  cout << "Sign reversed values for ob: ";
  cout << ob.x << ", " << ob.y << "\n";

  return 0;
}
```

这个程序可以被重新编码，成为下面的Java版本。如看到的那样，最多的转换工作是删除C++的指针。因为Java用引用传递对象，参数的改变自然影响到自变量。

```
// Reverse the signs of a coordinate - Java version.
class Coord {
  int x;
  int y;
};

class DropPointers {
  // Reverse the sign of the coordinates.
  static void reverseCoord(Coord ob) {
    ob.x = -ob.x;
    ob.y = -ob.y;
  }
  public static void main(String args[]) {
    Coord ob = new Coord();

    ob.x = 10;
    ob.y = 20;

    System.out.println("Original values for ob: " +
      ob.x + ", " + ob.y);

    reverseCoord(ob);

    System.out.println("Sign reversed values for ob: " +
      ob.x + ", " + ob.y);
  }
}
```

这两个程序的输出完全相同，如下所示：

```
Original values for ob: 10, 20
Sign reversed values for ob: -10, -20
```

27.2.2 转换用于数组操作的指针

理论上，将基于C++指针的数组访问转为对应的Java兼容的数组下标是非常直接的——简单地用合适的数组下标语句替代就可以了。然而，实际中，这需要一些考虑。基于指针的数组访问有点难以跟踪，因为标准的C++代码鼓励密集的、复杂的表达式。例如，这个简短的C++程序将一个数组的内容拷贝给另一个数组。它用0指示数组的结尾。注意指针表达式。甚至在这个简单的例子中，如果不知道这个程序是将nums的内容拷贝到copy中（然后打印出数组内容），在完全理解代码功能之前必须仔细考虑再开始动手改写。

```
// Copy an array in C++ using pointers.
```

```
#include <iostream>
using namespace std;

int main()
{
  int nums[] = {10, 12, 24, 45, 23, 19, 44,
              88, 99, 65, 76, 12, 89, 0};
  int copy[20];

  int *p1, *p2; // integer pointers

  // copy array
  p1 = nums; // p1 points to start of nums array
  p2 = copy;
  while(*p1) *p2++ = *p1++;
  *p2 = 0; // terminate copy with zero

  // Display contents of each array.
  cout << "Here is the original array:\n";
  p1 = nums;
  while(*p1) cout << *p1++ << " ";
  cout << endl;

  cout << "Here is the copy:\n";
  p1 = copy;
  while(*p1) cout << *p1++ << " ";
  cout << endl;

  return 0;
}
```

虽然这是非常简单的C++代码例子，先仔细看看这一行：

```
while(*p1) *p2++ = *p1++;
```

仍然需要考虑一会才能明白其确切操作。Java的一个优点是它不鼓励写这样的表达式。下面是这个程序的Java版本，可以看到，程序的目的和结果是非常清晰的。

```
// Array copy without pointers using Java.
class CopyArray {
  public static void main(String args[]) {
    int nums[] = {10, 12, 24, 45, 23, 19, 44,
                88, 99, 65, 76, 12, 89, 0};
    int copy[] = new int[14];
    int i;

    // copy array
    for(i=0; nums[i]!=0; i++)
      copy[i] = nums[i];
    nums[i] = 0; // terminate copy with zero

    // Display contents of each array.
    System.out.println("Here is the original array:");
    for(i=0; nums[i]!=0; i++)
      System.out.print(nums[i] + " ");
```

```
    System.out.println();

    System.out.println("Here is the copy:");
    for(i=0; nums[i]!=0; i++)
      System.out.print(copy[i] + " ");
    System.out.println();
  }
}
```

两个版本的程序都产生下面的结果：

```
Here is the original array:
10 12 24 45 23 19 44 88 99 65 76 12 89
Here is the copy:
10 12 24 45 23 19 44 88 99 65 76 12 89
```

许多C++代码分散而模糊，指针表达式难以理解。虽然这些表达式加快了执行速度，但是它们带来了C++程序难以维护的问题。而且当将这些代码转换为Java时，也非常困难。当面对一个复杂的指针表达式时，某些时候，先将其分解为子表达式将有助于确定其操作。

27.3 C++引用参数与Java引用参数

前面的小节提供了一个使用指针参数的C++程序例子。而在Java中，指针参数必须转换成引用参数。当然，C++也支持引用参数。前面提到过，大多数的C++代码中的指针参数是从C语言程序中遗留下来的。几乎所有新的C++代码在函数需要访问自变量时引用参数（本质上讲，虽然指针参数很普遍，在多数的C++代码中都是个时代错误）。虽然Java和C++都支持引用参数，但将使用引用参数的C++函数移植为Java方法时，仍然涉及一些改变。不幸的是，情况不总是一样。为了更深入的理解这点，下面的例子将移植一个C++程序，这个程序使用引用参数交换两个Coord对象的内容：

```
// Swap coordinates -- C++ version.
#include <iostream>
using namespace std;

class Coord {
public:
  int x;
  int y;
};

// Swap contents of two Coord objects.
void swap(Coord &a, Coord &b) {
  Coord temp;

  // swap contents of objects
  temp = a;
  a = b;
  b = temp;
}
```

```
int main()
{
  Coord ob1, ob2;

  ob1.x = 10;
  ob1.y = 20;

  ob2.x = 88;
  ob2.y = 99;

  cout << "Original values:\n";
  cout << "ob1: " << ob1.x << ", " << ob1.y << "\n";
  cout << "ob2: " << ob2.x << ", " << ob2.y << "\n";
  cout << "\n";
  swap(ob1, ob2);

  cout << "Swapped values:\n";
  cout << "ob1: " << ob1.x << ", " << ob1.y << "\n";
  cout << "ob2: " << ob2.x << ", " << ob2.y << "\n";

  return 0;
}
```

下面是这个程序的输出，可以看到，对象ob1 和ob2 的内容相互交换过了：

```
Original values:
ob1: 10, 20
ob2: 88, 99
Swapped values:
ob1: 88, 99
ob2: 10, 20
```

在Java中，所有的对象都通过对象引用变量访问。因此，当一个对象被传递给一个方法，只有它的引用被传递。这意味着所有的对象都可由引用自动传递给Java方法。如果不仔细考虑系统的实际操作，可能有人会将前面的程序转换成下面的（不正确的）版本。

```
// Swap program incorrectly converted to Java.
class Coord {
  int x;
  int y;
};

class SwapDemo {
  static void swap(Coord a, Coord b) {
    Coord temp = new Coord();

    // this won't swap contents of a and b!
    temp = a;
    a = b;
    b = temp;
  }

  public static void main(String args[]) {
    Coord ob1 = new Coord();
```

```
    Coord ob2 = new Coord();

    ob1.x = 10;
    ob1.y = 20;

    ob2.x = 88;
    ob2.y = 99;

    System.out.println("Original values:");
    System.out.println("ob1: " +
          ob1.x + ", " + ob1.y);
    System.out.println("ob2: " +
          ob2.x + ", " + ob2.y + "\n");

    swap(ob1, ob2);

    System.out.println("Swapped values:");
    System.out.println("ob1: " +
          ob1.x + ", " + ob1.y);
    System.out.println("ob2: " +
          ob2.x + ", " + ob2.y + "\n");
  }
}
```

由这个不正确的程序产生的输出如下所示：

```
Original values:
ob1: 10, 20
ob2: 88, 99
Swapped values:
ob1: 10, 20
ob2: 88, 99
```

如上所示，main( )中的ob1和ob2值并没有交换。虽然在开始的时候觉得有点违反直觉，实际上一旦理解了当一个对象引用被传递给方法时的实际操作过程，其发生错误的原因就非常明显。Java使用通过值调用的机制将自变量参数传递给方法。这意味着，制作一份自变量的拷贝传递进方法，因此在方法内部的拷贝对调用方法时使用的变量没有任何作用。然而，这种情况在对象引用时变得有点含混。

正如上面所说的那样，将对象的引用传递给方法时，系统制作了一份引用变量的拷贝。这意味着在方法内部的参数使用的对象和方法外部自变量使用的对象是相同的。因此，通过参数操作对象可以影响自变量所指的对象（因为它们是同一个对象）。但是在引用参数上的操作只影响参数自己。因此，前面的程序试图通过交换a和b对象指针来交换对象，而实际上只有参数（也就是自变量的拷贝）交换了它们所指向的对象，但是这样做并不影响在main( )中的ob1和ob2。

为了改正这个程序，需要重写swap( )方法，交换对象的内容而不是参数所指的对象，下面是swap( )的正确版本：

```
// Corrected version of swap().
static void swap(Coord a, Coord b) {
  Coord temp = new Coord();
```

```
  // swap contents of objects
  temp.x = a.x;
  temp.y = a.y;
  a.x = b.x;
  a.y = b.y;
  b.x = temp.x;
  b.y = temp.y;
}
```

如果用这个版本的swap( )替换前面程序，就可得到正确的结果。

27.4　将C++抽象类转换为Java的接口

Java技术最为创新独特的一个方面是接口。在书的前面章节中已经介绍过，接口定义了一组方法，但并没有指定任何实现细节。实现接口的类必须自己创建由接口声明的方法。因此，在Java中，一个接口意味着定义一个通用的类，这个类的所有特殊版本必须遵循同样的规则。接口是Java支持多态性的一种方式。

C++中没有与接口对应的部分。但是，C++中使用抽象类定义一个没有实现细节的类。C++中的抽象类与Java中的抽象类类似：它们都不包括实现细节。在C++语言中，一个抽象类包含至少一个纯虚函数。纯虚函数没有定义实现；只定义了函数原型。因此，一个C++中的纯虚函数本质上与Java中的抽象方法相同。在C++中，抽象类与Java的接口完成类似功能。因此，在移植C++代码到Java中时，必须注意这种区别。虽然许多C++抽象类都能转换成Java的接口，但不是所有的C++抽象类都能转换成Java的接口。下面让我们看两个例子：

这个短C++程序使用一个名为IntList的抽象类，定义了一个整数列表形式。这个类的实现由IntArray类创建，IntArray类用一个数组实现类整数列表。

```
0; // pure virtual functions
  virtual void putOnList(int i) = 0;
};

// Create an implementation of an integer list.
class IntArray : public IntList {
  int storage[100];
  int putIndex, getIndex;
public:
  IntArray() {
    putIndex = 0;
    getIndex = 0;
  }

  // Return next integer in list.
  int getNext() {
    if(getIndex >= 100) {
      cout << "List Underflow";
      exit(1);
    }
    getIndex++;
```

```
    return storage[getIndex-1];
  }

  // Put an integer on the list.
  void putOnList(int i) {
    if(putIndex < 100) {
      storage[putIndex] = i;
      putIndex++;
    }
    else {
      cout << "List Overflow";
      exit(1);
    }
  }
};

int main()
{
  IntArray nums;
  int i;

  for(i=0; i<10; i++) nums.putOnList(i);

  for(i=0; i<10; i++)
    cout << nums.getNext() << endl;

  return 0;
}
```

这个程序中，抽象类IntList只定义一个整数列表形式。它只包含纯虚函数，并没有声明任何数据。因此，当程序移植到Java时，这个类被作成一个接口，具体实现如下所示：

```
// Here, IntList is made into an interface which IntArray implements.

// Define interface for an integer list.
interface IntListIF {
  int getNext();
  void putOnList(int i);
}

// Create an implementation of an integer list.
class IntArray implements IntListIF {
  private int storage[];
  private int putIndex, getIndex;

  IntArray() {
    storage = new int[100];
    putIndex = 0;
    getIndex = 0;
  }

  // Create an implementation of an integer list.
  public int getNext() {
    if(getIndex >= 100) {
      System.out.println("List Underflow");
```

```
      System.exit(1);
    }
    getIndex++;
    return storage[getIndex-1];
  }

  // Put an integer on the list.
  public void putOnList(int i) {
    if(putIndex < 100) {
      storage[putIndex] = i;
      putIndex++;
    }
    else {
      System.out.println("List Overflow");
      System.exit(1);
    }
  }
}

class ListDemo {
  public static void main(String args[]) {
    IntArray nums = new IntArray();
    int i;

    for(i=0; i<10; i++) nums.putOnList(i);

    for(i=0; i<10; i++)
      System.out.println(nums.getNext());
  }
}
```

如上所示，在C++抽象类IntList和Java的IntListIF接口之间几乎是一一对应的。因为IntList抽象类仅包含虚函数，因此可以将其转换为IntListIF接口，这是非常关键的一点。如果IntList抽象类还包含任何数据或函数实现，它就没有资格转换成接口。

当转换或者改编C++代码时，首先检查那些仅含有纯虚函数的抽象类。它们是Java接口的主要转换对象。但是不要忽视含有少量实现函数或者数据的抽象C++类。这些实现函数或者数据并不一定必须归入抽象类，而是可以定义为单独的实现。因为C++中没有接口结构，因此C++的程序员不可能按接口形式设计程序。

在某些时候，抽象类将一个成员包括在内，仅仅是为了容易实现，而不是因为逻辑上这个成员属于这个类。例如，下面的C++抽象类：

```
0;
  virtual void f2(int i) = 0;
  virtual double f3() = 0;
  virtual int f4(int a, char ch) = 0;
};
```

仅仅因为isOK成员的存在，这个类不能转换成一个Java接口。基本上可以判断，isOK是用作指示这个类的相关状态。然而，如果仔细考虑一些，是没有理由将isOK定义成一个变量的。相反，可以定义一个isOK方法，然后返回其状态。在这种方式中，isOK可以和其

他方法一样，由其他类实现。因此，可以将前面的C++抽象类转换为下面的Java接口：

```
interface SomeClass {
  int f1();
  void f2(int i);
  double f3();
  int f4(int a, char ch);
  boolean isOK();
}
```

在移植过程中，许多C++的抽象类可以（也应该）转换成Java的接口。这样做，可以发现类的层次结构变得清晰了。

27.5 转换默认自变量

默认函数自变量，这个在C++中广泛应用的特性，Java并不支持。例如，在下面C++程序中使用的area( )函数在有两个自变量时计算矩形的面积，否则在只有一个自变量时计算正方形的面积。

```
// C++ program that uses default arguments.
#include <iostream>
using namespace std;

/* Compute area of a rectangle.  For a square,
   pass only one argument.
*/
double area(double l, double w=0) {
  if(w==0)  return l * l;
  else return l * w;
}

int main()
{
  cout << "Area of 2.2 by 3.4 rectangle: ";
  cout << area(2.2, 3.4) << endl;
  cout << "Area of 3.0 by 3.0 square: ";
  cout << area(3.0) << endl;
  return 0;
}
```

如上所示，当只用一个自变量调用area( )函数时，第二个自变量默认为0。此时，函数简单地使用第一个自变量作为矩形的长度和高度。

虽然很方便，但默认自变量机制不是必须的。本质上，默认自变量函数重载的简约版，其中一个函数和另一个函数的参数数目不同。因此，只需将包含一个或者多个默认自变量的C++函数转换为Java中的重载方法即可。在下面的例子中，有两个版本的area( )，其中一个有两个自变量，另一个只有一个自变量。按这种方式，重写前面的程序：

```
// Java version of area program.
class Area {
  // Compute area of a rectangle.
```

```
  static double area(double l, double w) {
    if(w==0)  return l * l;
    else return l * w;
  }

  // Overload area( ) for a square.
  static double area(double l) {
    return l * l;
  }

  public static void main(String args[]) {
    System.out.println("Area of 2.2 by 3.4 rectangle: " +
                       area(2.2, 3.4));

    System.out.println("Area of 3.0 by 3.0 square: " +
                      area(3.0));
  }
}
```

27.6　转换C++的多重继承层次结构

C++允许一个类同时继承两个或多个基本类。但Java不允许。为了更明确的理解C++和Java在继承关系上的不同，下面是两种层次结构的图形描述：

在两种情况下，子类C都继承了类A和B。但是在左边的层次结构中，C同时继承A和B。而在右边的结构中，B继承A，然后C继承B。Java通过不允许单个子类继承多个父类，简化了层次结构模型。多重继承带来了许多必须处理的问题。多重继承只对编程人员有益，却增加了编译器和运行环境的负担。

因为C++支持多重继承，而Java不支持。在移植C++应用程序时，必须处理这个问题。虽然每个程序的情况各不相同，这里给出两个较为通用的建议。第一，在多数情况下，在C++程序中的多重继承其实并不需要。在这种情况下，只要把类结构转换为单一继承结构就可以了。例如，下面定义了一个名为House类的C++类结构。

```
class Foundation {
  // ...
};

class Walls {
  // ...
};

class Rooms {
```

```
  // ...
};

class House : public Foundation, Walls, Rooms {
  // ...
};
```

注意，House多重继承了Foundation，Walls和Rooms。像这样的C++层次结构并没有任何错误，但是没有必要。例如，下面是Java中的同样的一组类结构：

```
class Foundation {
  // ...
}

class Walls extends Foundation {
  // ...
}

class Rooms extends Walls {
  // ...
}

class House extends Rooms {
  // ...
}
```

其中，每个类扩展类前面的类，House类是最后一个扩展。

有些时候，在最后一个对象中包括多重继承类的对象更加容易转换多重继承结构。下面是Java中构建House的另一种方法：

```
class Foundation {
  // ...
}

class Walls{
  // ...
}

class Rooms {
  // ...
}

/* 现在， House将Foundation, Walls, 和Rooms当作其对象成员
*/
class House {
  Foundation f;
  Walls w;
  Rooms r;
  // ...
}
```

其中，Foundation，Walls和Rooms是House的对象成员而不是由House继承。

另外一点，某些时候，一个C++程序包含一个多重继承仅仅因为最初设计不当。移植

程序正是纠正这种设计缺陷的好时候。

27.7　析构函数与finalize()

在从C++到Java的移植过程中，一个非常微妙的，但是又非常重要的必须面对的差别是C++的析构函数和Java的finalize( )方法。虽然它们在很多方面相似，但是它们的实际操作非常不同。首先，回顾一下C++析构函数和Java的finalize( )方法的目的和效果。

在C++中，当一个对象超出其有效范围时，它就会被撤消。在它被撤消前，调用其析构函数（如果它有的话）。这是一个确定而且快速的规则。没有任何例外。再仔细考察一下这个规则的每部分：

- 每个对象超出作用域都会被撤消。因此如果在一个函数中定义了一个本地对象，在这个函数返回时，本地的对象自动被撤消。函数返回的对象和函数参数也同样对待。
- 在被撤消前，调用对象的析构函数。这个操作立即执行，而且在任何其他程序的语句执行前执行。因此，C++的析构函数的执行方式非常确定。程序员总是知道析构函数的执行时间和地点。

在Java中，对象的撤消和其finalize( )方法的调用之间几乎没有联系。在Java中，对象执行完毕，并不显式撤消对象。而是在没有任何引用的时候，将对象标志为不再使用。因此，程序员无法确切的知道何时何地调用finalize( )。甚至执行gc( )调用（垃圾回收）时，也不保证立即执行finalize( )方法。

虽然在多数情况下，不必关心C++析构函数的确定性行为和finalize()的不确定性行为之间的差别，但它们有时会相互影响。例如下面的C++程序：

```
// This C++ program can call f() indefinitely.
#include <iostream>
#include <cstdlib>
using namespace std;

const int MAX = 5;
int count = 0;

class X {
public:
  // constructor
  X() {
   if(count<MAX) {
     count++;
    }
    else  {
      cout << "Error -- can't construct";
      exit(1);
    }
  }

  // destructor
  ~X() {
```

```
    count--;
  }
};

void f()
{
  X ob; // allocate an object
  // destruct on way out
}

int main()
{
  int i;

  for(i=0; i < (MAX*2); i++) {
    f();
    cout << "Current count is: " << count << endl;
  }

  return 0;
}
```

下面是这个程序的输出：

```
Current count is: 0
Current count is: 0
Current count is: 0
Current count is: 0
Current count is: 0
Current count is: 0
Current count is: 0
Current count is: 0
Current count is: 0
Current count is: 0
```

注意观察X的构造函数和析构函数。构造函数在count比MAX小之前，增加count的值。析构函数减少count的值。因此在X对象创建时count值增加，在X对象撤消时，count值减少。但是MAX对象在任何时候都存在。在main( )中，调用MAX*2次f( ) ，程序没有任何问题！下面解释其原因。在f( )中，创建X对象导致count值增加，然后函数返回。这引起对象超出其作用域，对象的析构函数被调用，count值减少。因此，调用f( )不影响count的值。这意味着可以无限次的调用f( )。然而在转换为Java时，情况发生了改变。

下面是前面程序的Java版本：

```
// This Java program will fail after 5 calls to f().

class X {
  static final int MAX = 5;
  static int count = 0;

  // constructor
  X() {
   if(count<MAX) {
     count++;
```

```
    }
    else {
      System.out.println("Error -- can't construct");
      System.exit(1);
    }
  }

  // finalization
  protected void finalize() {
    count--;
  }

  static void f()
  {
    X ob = new X(); // allocate an object
    // destruct on way out
  }

  public static void main(String args[]) {
    int i;

    for(i=0; i < (MAX*2); i++) {
      f();
      System.out.println("Current count is: " + count);
    }
  }
}
```

在程序5次调用f( )后，将出现错误，其输出如下：

```
Current count is: 1
Current count is: 2
Current count is: 3
Current count is: 4
Current count is: 5
Error — can't construct
```

程序失败的原因是每次f( )返回时垃圾回收并不出现。因此，没有调用finalize( )方法，count的值也没有减少。在5次调用方法后，count到达其最大值，因此程序出错。

强调垃圾回收出现的精确时间是与实现相关的这一点非常重要。在某些平台的某些Java实现中，前面的程序可以与C++版本的程序一样，可以正常运行。但是例子中指出的不同之处仍然存在：在C++中，程序设计者指定析构函数调用的时间和地点，但在Java中，程序设计者不知道finalize( )执行的时间和地点。因此，在移植C++代码时，必须注意那些依赖于精确时间执行的析构函数的实例。

第4部分　应用Java

第 28 章　DynamicBillboard 小应用程序

Robert Temple是Starwave公司的软件工程师，他设计了多个在世界范围内广泛使用的小应用程序。他的工作包括ESPNETSportsZone“HitCharts”和“Batter vs. Pitcher”小应用程序。最让Starwave公司的人印象深刻的，而且因此邀请他加入公司的小应用程序是DynamicBillboard，这是他还在佛罗里达的Embry-Riddle 航空大学时编写的。

DynamicBillboard小应用程序重复的显示一组图片，每隔一定时间就更换屏幕上的图片。图片的交替可以用不同的效果实现。其中一个例子是SmashTransition，新图片从老图片的顶部落下，将老图片挤压变形。这个小应用程序通过与每个图片相关联的URL链接到其他页面。当用户在小应用程序上点击鼠标时，浏览器就链接到与当前图片相关联的新页面上。DynamicBillboard为web站点提供了一个在静态页面上显示滚动广告、标志、广告牌的最好方法。

Robert做了许多有趣的优化工作。没有他的精湛技艺，这个小应用程序的功能不会如此强大。在这个小应用程序的源码中有足够多的技巧值得参考。

28.1　APPLET标记

DynamicBillboard的APPLET标记相当容易配置。和大多数的小应用程序一样，在code参数中指定主类，然后指定宽度和高度，就可以了：

```
<applet code=DynamicBillboard width=392 height=72>
```

为使小应用程序正常工作还需要指定几个参数。没有了这几个参数，小应用程序什么也不能做。同时，注意如果没有正确的命名文件，那么结果可能就不怎么样：不是什么都不显示，就是一些广告板区域一片空白。下列的参数必须为：

```
<param name=parameter_name value="your value here">
```

- bgcolor　这个参数是第一个图片装载之前小应用程序设定的背景色。使用这个参数可以尽快避免屏幕上出现灰色的小应用程序区域。.
- delay　这个参数指定广告牌间隔的毫秒数。典型情况下，是5000或10000，意味着五到十秒。
- billboards　这个参数指定希望循环播放的广告牌数目。

- bill#　这是缩写形式的bill0，bill1，bill2等，直到比指定的广告牌数目少一个（Robert是一个典型的从0开始计数的程序员）。一共有与billboards参数中指定数目一致的bill#。每个bill#的值是由逗号分隔的一对字符串。第一个是这个广告牌中显示的图片名，第二个是用户点击这个广告牌时连接的URL。例子如下所示：

```
<param name="bill0" value="starwave.jpg,http://www.starwave.com/">
```

- transitions　这是个列表，开头是一个整数表示的项数，其后是Transition子类名字的列表。下面是一个例子：

```
<param name="transitions" value="2,TearTransition,SmashTransition">
```

下面是一个完整的APPLET标记的例子，这个标记中包含了本章所要讨论的全部转换：

```
<applet code=DynamicBillboard width=392 height=72>
<param name="bgcolor" value="#ffffff">
<param name="delay" value="5000">
<param name="billboards" value="5">
<param name="bill0"
   value="robert.jpg,http://www.starwave.com/people/robertt/">
<param name="bill1"
   value="kenna.jpg,http://www.starwave.com/people/naughton/kenna/">
<param name="bill2"
    value="lavallee.jpg,http://www.starwave.com/people/lavallee/">
<param name="bill3"
   value="handbook.jpg,http://www.starwave.com/people/naughton/book/">
<param name="bill4"
   value="family.jpg,http://www.starwave.com/people/naughton/family/">
<param name="transitions"
   value="5,ColumnTransition,FadeTransition,TearTransition,
                        SmashTransition,UnrollTransition">
</applet>
```

28.2　源代码概述

Robert希望设计一个加载速度快的小应用程序。他尽量保持小应用程序为最小，由此减少通过网络传输的代码量。同时他尝试将某些小应用程序的加载和初始化工作延迟到第一幅图片显示之后。正像用户希望的那样，即使还有许多工作要做，在第一幅图片完全显示之后小应用程序开始运行。

小应用程序包括三个主要的类和多个转换类。这三个主要的类是DynamicBillboard，BillData和BillTransition。DynamicBillboard是一个顶层的小应用程序子类，使用所有的其他类。BillData类封装了一些广告牌的属性，包括图片和与图片相关的URL。BillTransition是一个抽象类，包含所有转换类的公共属性和方法。下面分别描述这三个主要的类和五个流行的转换形式。

28.2.1　DynamicBillboard.java

这是一个主要的小应用程序类，它实现Runnable接口以包括一个线程，这个线程控制

连续过程的创建和动态转换。transition_classes数组中存放转换类名字的字符串。使用字符串的原因是可以使用java.lang.Class.forName(String)方法动态加载这些类。这样，直到这些类第一次实例化前，小应用程序可以推迟加载这些类。

init( )

当小应用程序第一次加载时自动调用init( )。多数小应用程序用这种方法完成所有必须的初始化工作。然而，Robert决定将初始化分为两种方法：init( )和finishInit( )。这种分开初始化过程的背后是为了用最短时间在小应用程序中显示第一幅图像，减少小应用程序在加载和初始化时，显示灰色空白矩形框的时间。在init( )中进行的都是屏幕初始内容所绝对必须的处理过程，因为浏览器在init( )返回之后才调用paint( )。

Robert在init( )中最先做的工作是改变小应用程序和嵌入小应用程序的父帧的背景色。通常，在小应用程序加载和初始化时，屏幕上小应用程序使用的区域显示为不透明的灰色矩形。从1994年开始，创建每个页时，倾向使用灰色以外的背景色来突出这个框体。Robert发现这种方法的一个问题，一般小应用程序总有个嵌入的父容器。在Netscape Navigator和Internet Explorer中，这个父容器来自核心Java类：java.awt.Container。Robert 使用从java.awt.Component——setBackground( ) 和 repaint( )——继承来的方法改变小应用程序参数bgcolor的值从而改变背景色值。这使小应用程序区域比其为灰色时略为隐蔽。所有这些在小应用程序第一次加载图像前已经完成了。

> **注意**：市场上使用Netscape 3.0以上的新版本浏览器，其帧的背景不再默认地设为灰色而是使用当前页的背景色。将来，随着这种浏览器的广泛使用，改变背景色对小应用程序没有益处。这里只是说明这种技术，供你以后需要时使用。

改变背景色后，Robert的小应用程序读入一个参数，这个参数指示共有多少个广告牌，然后根据这个参数为BillData对象分配一个数组。

使用Math.random( )，随机选择一个广告牌作为开始。调用parseBillData( )来解析这个广告牌的参数。

parseBillData( )

这种方法创建和初始化小应用程序使用的下一个广告牌(BillData) 对象。只有在广告牌对象还未被创建时才调用此方法（在广告牌数组中与下一个广告牌对象相对应的元素为空）。

通常情况下，在创建新对象后，parseBillData( ) 调用BillData方法中的 initPixels( ) 来初始化在BillData对象中的像素数组。然而，在第一次调用此方法时，小应用程序仍在努力尽早将第一幅图像显示于屏幕中。这是因为绘制小应用程序的图像的引用值仍为null。所以在第一幅图像加载后，设置这个图像（image）变量并等待调用需要大量占用处理器资源的initPixel( )方法直至第一幅图像被完全加载。

finishInit( )

在第一幅图像显示在屏幕之后，小应用程序可完成剩余的初始化。这包括初始化所有

转换类名字、初始化第一个广告牌的像素数组和读取目标参数。

finishInit( )是被小应用程序的run( )方法调用的。每次用户离开和返回此页时，run( )方法从顶端开始运行。这时，再次调用finishInit( )。因为小应用程序已经完成初始化，所以Robert不想再重新初始化每件事情。这也是小应用程序检查delay变量是否已经初始化的原因。如果已经初始化，小应用程序忽略余下的初始化工作。

start( ) 和 stop( )

用户访问或离开页面时，分别调用start( )和stop( )方法。它们保证运行转换的小应用程序线程的开启和关闭。

如果在小应用程序运行一个转换当中调用stop( )，有些数据可能会处于不正确状态。在start( ) 中重新设置一些变量以保证小应用程序正确运行一个新转换。

在start( )过程中，鼠标的光标变为手型。所以如果鼠标移至一个小应用程序上，它将显示为一个链接。

run( )

在等待第一幅图像完全加载时，run( )方法开始一个循环。然后，调用finishInit( )完成小应用程序的初始化工作。从这里，进入程序的主循环。

主循环决定广告牌之间的转换。使用从HTML传递来的延迟参数，小应用程序开始计算下一个转换的开始时间。等待时它即开始准备转换。准备工作的开始是决定下一个显示的广告牌，如果这个广告牌的HTML参数还没有被解析，那么先解析参数。然后，随机选择下一次运行的转换，注意不要让小应用程序连续运行相同的转换。

一旦小应用程序决定下一次运行的转换，它通过String名字动态加载转换类并创建类的新实例，这种方法创建了转换类的新实例。动态加载转换类对小应用程序的总加载时间影响很大。在小应用程序启动前不必下载全部类，最初仅需要三个类： DynamicBillboard, BillData和BillTransition。其他的转换类只在小应用程序需要时下载。这样做显著地减少了小应用程序的下载时间。如果用户迅速离开页面，甚至无需下载某些类文件。

最后，小应用程序调用转换对象的init( )方法，将小应用程序和当前及下一个广告牌的图形像素作为参数传递。这样创建了用在动画转换中的单个帧。在转换工作准备完毕后，小应用程序仅需等待合适的时间启动转换。

小应用程序使用单帧动画完成转换——在屏幕上每次绘制一个单元，在每个帧之间延迟一个短暂的时间。Applet调用工具包方法sync( )以确保单元绘制发生在前一个单元完全绘制完毕之后。在最后一个单元显示完毕，小应用程序在屏幕上绘制下一个广告牌的图像完成本次转换。

随后，检查mouse_over_applet标志看光标是否在小应用程序上。如果光标是在小应用程序上，在状态栏中显示的前一个广告牌的URL必须更新为当前广告牌的URL。这是通过调用小应用程序的showStatus( )方法完成的。至此，小应用程序完成本次转换，准备开始下次转换。

mouseMoved( ) 和 mouseExited( )

使用mouseMoved( )和mouseExited( )方法改变状态栏中的文字。当鼠标光标移到小应用程序上时，状态栏应该显示当前广告牌连接的URL。因此当调用mouseMoved( )时，小应用程序在状态栏中显示URL。在调用mouseExited( )时，状态栏中消除URL。这两个方法都设置布尔变量mouse_inside_ applet。这个变量用在转换执行后的run( )方法中。如果在转换完成时，鼠标的位置停留在小应用程序上，小应用程序在状态栏中显示新广告牌的URL。

mouseReleased( )

当按在小应用程序上的鼠标键松开时，调用mouseReleased( )方法。小应用程序使用getAppletContext( ).show Document( )将浏览器转到新广告牌指向的URL。正如Robert发现的那样，某些浏览器需要很长的时间才能显示出新的页面。在等待新页面下载的时候为防止小应用程序运行新的转换，调用stop( )强制主线程退出。为了让用户指定小应用程序正在下载新的页面，鼠标的光标变为等待的光标。

值得注意的一点是用户可能从新的页面返回到原来的页面。在用户返回时，仍然显示等待光标。调用start( )后，小应用程序将光标重置为手形。

源代码

下面是DynamicBillboard类的源代码：

```
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.awt.*;
import java.awt.image.*;

public class DynamicBillboard
      extends java.applet.Applet
      implements Runnable {

  BillData[] billboards;
  int current_billboard;
  int next_billboard;

  String[] transition_classes;
  Thread thread = null;
  Image image = null;
  long delay = -1;
  boolean mouse_inside_applet;
  String link_target_frame;
  boolean stopFlag;

  public void init() {
    String s = getParameter("bgcolor");
    if(s != null) {
      Color color = new Color(Integer.parseInt(s.substring(1), 16));
      setBackground(color);
      getParent().setBackground(color);
```

```
    getParent().repaint();
  }
  billboards = new
    BillData[Integer.parseInt(getParameter("billboards"))];
  current_billboard = next_billboard
                = (int)(Math.random() *billboards.length);
  parseBillData();
}

void parseBillData() {
  String s = getParameter("bill" + next_billboard);
  int field_end = s.indexOf(",");
  Image new_image = getImage(getDocumentBase(),
                        s.substring(0, field_end));
  URL link;
  try {
    link = new URL(getDocumentBase(),
                s.substring(field_end + 1));
  }
  catch (java.net.MalformedURLException e) {
    e.printStackTrace();
    link = getDocumentBase();
  }
  billboards[next_billboard] = new BillData(link, new_image);
  if(image == null) {
    image = new_image;
  }
  else {
    prepareImage(new_image, this);
    billboards[next_billboard].initPixels(getSize().width,
                                    getSize().height);
  }
}

void finishInit() {
  if(delay != -1) {
    return;
  }
  delay = Long.parseLong(getParameter("delay"));

  link_target_frame = getParameter("target");
  if(link_target_frame == null) {
    link_target_frame = "_top";
  }

  String s = getParameter("transitions");
  int field_end = s.indexOf(",");

  int trans_count = Integer.parseInt(s.substring(0, field_end));
  transition_classes = new String[trans_count];
  for(--trans_count; trans_count > 0; --trans_count) {
    s = s.substring(field_end + 1);
    field_end = s.indexOf(",");
    transition_classes[trans_count] = s.substring(0, field_end);
  }
```

```
    transition_classes[0] = s.substring(field_end + 1);
    billboards[next_billboard].initPixels(getSize().width,
                                          getSize().height);
    mouse_inside_applet = false;
  }

  public void paint(Graphics g) {
    g.drawImage(image, 0, 0, this);
  }

  public void update(Graphics g) {
    paint(g);
  }

  public void start() {
    next_billboard = current_billboard;
    image = billboards[current_billboard].image;
    setCursor(new Cursor(Cursor.HAND_CURSOR));
    if(thread == null) {
      thread = new Thread(this);
      thread.start();
    }
  }

  public void stop() {
    if(thread != null) {
      stopFlag = true;
    }
  }

  public void run() {
    while((checkImage(image, this) & ImageObserver.ALLBITS) == 0) {
      try { Thread.sleep(600); } catch (InterruptedException e) {}
    }
    finishInit();

    addMouseListener(new MyMouseAdapter());
    addMouseMotionListener(new MyMouseMotionAdapter());

    int last_transition_type = -1;
    BillTransition transition;
    long next_billboard_time;
    while(true) {
      if(stopFlag)
        return;
      next_billboard_time = System.currentTimeMillis() + delay;
      current_billboard = next_billboard;
      if(++next_billboard >= billboards.length) {
        next_billboard = 0;
      }
      if(billboards[next_billboard] == null) {
        parseBillData();
        try { Thread.sleep(120); } catch (InterruptedException e) {}
      }
      int transition_type = (int)(Math.random() *
```

```
                          (transition_classes.length - 1));
    if(transition_type >= last_transition_type) {
      ++transition_type;
    }
    last_transition_type = transition_type;

    try {
      String trans = transition_classes[last_transition_type];
      transition = (BillTransition)Class.forName(trans)
                                  .newInstance();
    }
    catch(Exception e) {
      e.printStackTrace();
      continue;
    }

    transition.init(this,billboards[current_billboard].image_pixels,
      billboards[next_billboard].image_pixels);

    if(System.currentTimeMillis() < next_billboard_time) {
      try {
        Thread.sleep(next_billboard_time -
                    System.currentTimeMillis());
      } catch (InterruptedException e) { };
    }
    Graphics g = getGraphics();
    for(int c = 0; c < transition.cells.length; ++c) {
      image = transition.cells[c];
      g.drawImage(image, 0, 0, null);
      getToolkit().sync();
      try { Thread.sleep(transition.delay); }
     catch(InterruptedException e) { };
    }
    image = billboards[next_billboard].image;
    g.drawImage(image, 0, 0, null);
    getToolkit().sync();
    g.dispose();
    if(mouse_inside_applet == true) {
      showStatus(billboards[next_billboard].link.toExternalForm());
    }
    transition = null;
    try { Thread.sleep(120); } catch (InterruptedException e) {}
  }
}
public class MyMouseAdapter extends MouseAdapter {
  public void mouseExited(MouseEvent me) {
    mouse_inside_applet = false;
    showStatus("");
  }
  public void mouseReleased(MouseEvent me) {
    stop();
    setCursor(new Cursor(Cursor.WAIT_CURSOR));
    getAppletContext().showDocument(billboards[current_billboard].link,
                               link_target_frame);
  }
```

```
    }
    public class MyMouseMotionAdapter extends MouseMotionAdapter {
      public void mouseMoved(MouseEvent me) {
        mouse_inside_applet = true;
        showStatus(billboards[current_billboard].link.toExternalForm());
      }
    }
  }
```

28.2.2 BillData.java

BillData类基本上只是一个封装与每个广告牌相关的属性的数据结构。它包括三个变量。第一个变量存储广告牌链接的URL，第二个变量存储小应用程序用来绘制屏幕的Image对象，第三个变量包括一个RGB格式的像素数组。

这个像素数组被转换用作与另一个BillData像素数组组合来创建转换动画中的单元。这个数组只是一个一维数组。其中的像素是这样组织的，数组中的第一个元素是图像的顶部的左上角。第二个元素是第一个元素的右边。每个元素都紧挨着前一个元素的右边，以此类推，直到最右边的像素。然后，是第二行的最左边的像素。这样连续不断，数组的最后一个元素对应图像底部的右下角。

可以注意到，Robert将这个类中的所有变量定义为公用的（public）。正常情况下，比较好的编程习惯是隐藏数据成员，只允许其他类读取。应该是定义变量为protected或private，然后创建返回这些变量的引用。不幸的是，在Java中即使将仅有一行的方法定义为final并优化代码的编译，这样做也会增加类字节码的长度。因此为了保持小应用程序代码量少，下载快的优点，Robert将数据成员定义为公用。

构造函数

BillData对象的构造函数使用传递进来的两个参数初始化URL和Image变量。因为初始化像素数组的工作需要大量使用处理器资源，所以这部分工作是在另外的方法中完成的。这使得小应用程序可以仅在需要时初始化像素数组。

initPixels( )

initPixels( )方法使用Java的核心类：java.awt.image.PixelGrabber，从图像中创建像素数组。

程序代码

下面是BillData类的源代码：

```
import java.net.*;
import java.awt.*;
import java.awt.image.*;

public class BillData {
  public URL link;
  public Image image;
  public int[] image_pixels;
```

```
  public BillData(URL link, Image image) {
    this.link = link;
    this.image = image;
  }

  public void initPixels(int image_width, int image_height) {
    image_pixels = new int[image_width * image_height];
    PixelGrabber pixel_grabber = new
    PixelGrabber(image.getSource(), 0, 0,
         image_width, image_height, image_pixels, 0, image_width);
    try {
      pixel_grabber.grabPixels();
    }
    catch (InterruptedException e) {
      image_pixels = null;
    }
  }
}
```

28.2.3　BillTransition.java

BillTransition类是其他转换类的基础。其他类在两个单独的广告牌图像之间创建转换单元。这个抽象类包括所有转换的公用变量和方法。

BillTransition类没有构造函数。这是因为小应用程序不使用“new”来创建新的实例，而是使用制造方法（factory method）java.lang.Class.newInstance( )。用这种方法创建的对象不能使用构造函数中的参数初始化。这种制造方法使用一个没有任何参数的默认构造函数间接创建对象。BillTransition类提供一些重载init( )方法来用参数初始化实例。

在以前的DynamicBillboard版本中，Robert在不同的转换中使用静态变量存储只需初始化一次的数据。然而却发现，当一个web服务器存在多个小应用程序实例时，小应用程序们会共享这些静态变量。这会导致一些问题，譬如小应用程序的大小不同时，一个小应用程序会需要与其他小应用程序不同的静态值。一个例子是，用FadeTransition类创建数组时，其数组大小依赖于小应用程序的大小。而DynamicBillboard的数组要比前一个小应用程序小，它会用一个比前一个小应用程序小的数组重写这个数组。这将导致前一个小应用程序的崩溃。

Robert在这一版本的小应用程序中引入一个名为object_table静态的散列表来解决这个问题。目前转换类可将数据存入散列表，其中关键字为转换名和小应用程序的大小。当需要使用这些数字时，小应用程序在散列表中查询看是否存在，如果不存在，则创建数据并存入散列表中以方便以后使用。如果目前web服务器中有不止一个的小应用程序，而且有两个大小相同，则只需初始化一个数据。

init( )

init( )方法被三次重载。第一个方法带三个参数，是一个抽象方法，必须被从这个类派生的类重载。其余的两种方法初始化类中的数据成员。Robert的目的是凡是继承这个类并有init( )方法的类调用这两个方法中的一个初始化数据成员BillTransition。

createCellFromWorkPixels( )

createCellFromWorkPixels( )方法完成将work_pixels数组转换为Image对象的公用任务。注意它使用owner变量完成这一任务。这是转换类需要owner变量的惟一理由。当一个转换类需要将work_pixels数组组装成一个新单元时，调用此方法。

源代码

下面是BillTransition类的源代码：

```
import java.util.*;
import java.awt.*;
import java.awt.image.*;

public abstract class BillTransition {
  static Hashtable object_table = new Hashtable(20);

  public Image[] cells;
  public int delay;

  Component owner;
  int cell_w;
  int cell_h;
  int pixels_per_cell;
  int[] current_pixels;
  int[] next_pixels;
  int[] work_pixels;

  public abstract void
  init(Component owner, int[] current_pixels, int[] next_pixels);

  final protected void
  init(Component owner, int[] current_pixels, int[] next_pixels,
                    int number_of_cells, int delay) {
    this.delay = delay;
    this.next_pixels = next_pixels;
    this.current_pixels = current_pixels;
    this.owner = owner;

    cells = new Image[number_of_cells];
    cell_w = owner.getSize().width;
    cell_h = owner.getSize().height;
    pixels_per_cell = cell_w * cell_h;
    work_pixels = new int[pixels_per_cell];
  }

  final protected void
  init(Component owner, int[] current_pixels, int[] next_pixels,
                    int number_of_cells) {
    init(owner, current_pixels, next_pixels, number_of_cells, 120);
  }

  final void createCellFromWorkPixels(int cell) {
    cells[cell] = owner.createImage(
```

```
        new MemoryImageSource(cell_w, cell_h,
                              work_pixels, 0, cell_w));
      owner.prepareImage(cells[cell], null);
    }
  }
```

28.2.4　ColumnTransition.java

ColumnTransition类在切换图像时将新图像的列逐步扩大来覆盖旧图形。图像的列向左扩展，相同的像素总是绘制在每一列的左边。这使得广告牌看起来像是从旧的广告牌后面通过一个垂直通道滑动出来的。

为了创建这个转换类的单元，广告牌的空间分割成一定数目的列，每一类为24像素宽。转换类创建的每七个图形单元含有来自旧图像的左边的像素和新图像右边的像素。第一个被创建的单元只有从新图像中提取的每个列的三个右边像素。在随后的单元中，逐步填充从新图像中提取的三个像素。最后一个单元只有三个来自旧图像每列最左边的像素。

因为图像的宽度并不总是可以正好被24整除，所以图像的右边总是有些残留的像素。在每个单元中这些像素用变量rightmost_columns_max_width 和rightmost_columns_x_start表示。

init( )

init( )函数以调用基类的init( )方法开始，初始化包含在基类中的变量。它继续初始化与右边列相关的变量，然后将当前广告牌中的像素拷贝到工作像素数组中。随后的循环创建全部的单元帧。 nextCell( ) 方法改变work_pixels数组，这个方法继承于BillTransition 类。使用createCellFromWorkPixels( )方法将像素转换为图像。因为创建单元的过程大量的使用CPU资源，所以Robert允许线程暂时睡眠以允许其他线程运行。

nextCell( )

nextCell( )方法为下一个单元修改work_pixels数组。它从下往上的循环处理图形的每一行，从下一个广告牌中拷贝像素至work_pixels数组，以填充图像的每一列。它根本不需要从旧广告牌中拷贝像素，因为旧广告牌的像素已经被init( )方法拷贝进数组。

值得反复强调的是形成图像的像素数组是一维数组。每width个像素代表图像的一个水平行。

源代码

下面是ColumnTransition类的源代码：

```
import java.awt.*;
import java.awt.image.*;

public class ColumnTransition extends BillTransition {
  final static int CELLS = 7;
  final static int WIDTH_INCREMENT = 3;
  final static int MAX_COLUMN_WIDTH = 24;

  int rightmost_columns_max_width;
```

```
  int rightmost_columns_x_start;
  int column_width = WIDTH_INCREMENT;

  public void init(Component owner, int[] current, int[] next) {
    init(owner, current, next, CELLS, 200);

    rightmost_columns_max_width = cell_w % MAX_COLUMN_WIDTH;
    rightmost_columns_x_start = cell_w - rightmost_columns_max_ width;

    System.arraycopy(current_pixels, 0,
                  work_pixels, 0, pixels_per_cell);

    for(int c = 0; c < CELLS; ++c) {
      try { Thread.sleep(100); } catch (InterruptedException e) {}
      NextCell();
      try { Thread.sleep(100); } catch (InterruptedException e) {}
      createCellFromWorkPixels(c);
      column_width += WIDTH_INCREMENT;
    }
    work_pixels = null;
  }

  void NextCell() {
    int old_column_width = MAX_COLUMN_WIDTH - column_width;
    for(int p = pixels_per_cell - cell_w; p >= 0; p -= cell_w) {
      for (int x = 0; x < rightmost_columns_x_start; x +=
          MAX_COLUMN_WIDTH) {
        System.arraycopy(next_pixels, x + p, work_pixels,
             old_column_width + x + p, column_width);
      }
      if(old_column_width <= rightmost_columns_max_width) {
        System.arraycopy(next_pixels, rightmost_columns_x_start + p,
                      work_pixels, rightmost_columns_x_start +
                                old_column_width + p - 1,
                                rightmost_columns_max_width -
                                old_column_width + 1);
      }
    }
  }
}
```

下面是列转换之前、转换期间和转换之后的显示：

28.2.5　FadeTransition.java

FadeTransition类在变换图形时随机的将下一个广告牌的新像素插入连续的单元帧。这使得下一个广告牌好象是逐步地替代了老的广告牌。

这个转换的中心是一个名为random的二维short类型的整数数组。这个数组保持着下一个广告牌的像素数组的每个元素的下标。这些下标随机的分配在一个两维数组中。这个数组的第一维中的8个元素在单元创建时使用，每个新单元使用一个元素。实际上并没有真正使用最后一个元素，因为只有7个单元。在数组创建时包括这个单元是为了确保下标随机分布的正确性。

FadeTransition类使用这个数组在下一个广告牌中挑选像素以覆盖旧的广告牌。第一个单元中，work_pixels数组仅包含旧广告牌中的像素，其中1/8的像素被替换成新广告牌的像素。在随后的单元使用相同的work_pixels数组，再替换1/8的新像素。这个单元中就有1/4的像素来自新广告牌，而其余的仍是旧广告牌的像素。这个过程持续到最后一个单元（单元七），单元七有7/8的像素来自新广告牌。注意，在最后一个单元后，DynamicBillboard 小应用程序简单的使用下一个像素的完整图像来完成转换。

因为二维数组的大小由小应用程序大小决定，所以每个小应用程序必须保持一个独立的数组。使用静态变量保存数组是不可行的，因为不同大小的小应用程序会共享这个数组。因为创建这个数组要花相当长的时间，所以在直觉上，这个小应用程序不应每次使用这个转换的时候都重新创建数组。

这正是父类中的静态变量，object_table起作用的时候。一旦数组建立，它可以以小应用程序的大小为关键字存储在这个散列表中，当需要使用这个数组时，小应用程序可以从散列表中获取合适的数据。如果散列表中不存在合适数据，则小应用程序创建数组，并将创建的数组存放在散列表中留待以后使用。这看起来有点麻烦，但实际上，web站点会用这个小应用程序作为一个标准布局尺寸在大量页面上实现标题广告。因此，它节省了大量的高速缓存这些表的内存和CPU时间。

createRandomArray( )

createRandomArray( )静态方法创建一个二维的随机数组。它利用两个描述小应用程序大小的参数。因为它的原始版本非常慢，所以它被高度优化。它包括一个自己的快速，但范围狭窄的随机数生成器。因为这个随机生成器非常复杂，超出了本书的范围。其基本思想是Java内部的随机数生成器比较擅长产生真正的随机发布的随机数，但是对这个应用来说太慢了。而且，用户并不关心这个转换的随机程度，因此Robert的自制随机数生成器完全可以满足要求。

init( )

这个转换的init( )方法的启动方式和其他的转换一样，调用基类的init( )方法。然后像其

他转换一样，将旧广告牌的像素拷入work_pixels数组。

按这个小应用程序大小从object_table散列表中抽出一个两维的随机数组。如果不存在这样的一个数组，则创建并将新创建的数组存放在object_table表中。用手中的随机数据数组，init()方法循环处理每个单元和数组中的每个随机数，将新广告牌的像素拷入工作数组。

程序代码

下面是FadeTransition 类的源代码：

```
import java.awt.*;
import java.awt.image.*;

public class FadeTransition extends BillTransition {
  private static final int CELLS = 7;
  private static final int MULTIPLIER = 0x5D1E2F;

  private static short[][] createRandomArray(int number_pixels,
                                     int cell_h) {
    int total_cells = CELLS + 1;
    int new_pixels_per_cell = number_pixels / total_cells;
    short[][] random = new short[total_cells][new_pixels_per_cell];
    int random_count[] = new int[total_cells];
    for(int s = 0; s < total_cells; ++s) {
      random_count[s] = 0;
    }

    int cell;
    int rounded_new_pixels_per_cell =
         new_pixels_per_cell * total_cells;
    int seed = (int)System.currentTimeMillis();

    int denominator = 10;
    while((new_pixels_per_cell % denominator > 0 ||
      cell_h % denominator == 0) && denominator > 1) {
      --denominator;
    }
    int new_randoms_per_cell = new_pixels_per_cell / denominator;
    int new_randoms = rounded_new_pixels_per_cell / denominator;

    for(int p = 0; p < new_randoms_per_cell; ++p) {
      seed *= MULTIPLIER;
      cell = (seed >>> 29);
      random[cell][random_count[cell]++] = (short)p;
    }
    seed += 0x5050;
    try { Thread.sleep(150); } catch (InterruptedException e) {}

    for(int p = new_randoms_per_cell; p < new_randoms; ++p) {
      seed *= MULTIPLIER;
      cell = (seed >>> 29);

      while(random_count[cell] >= new_randoms_per_cell) {
        if(++cell >= total_cells) {
```

```
          cell = 0;
        }
      }
      random[cell][random_count[cell]++] = (short)p;
    }

    for(int s = 0; s < CELLS; ++s) {

      for(int ps = new_randoms_per_cell; ps < new_pixels_per_cell;
           ps += new_randoms_per_cell) {

        int offset = ps * total_cells;

        for(int p = 0; p < new_randoms_per_cell; ++p) {
          random[s][ps + p] = (short)(random[s][p] + offset);
        }
      }
      try { Thread.sleep(50); } catch (InterruptedException e) {}
    }
    random[CELLS] = null;
    return random;
  }

  public void init(Component owner, int[] current, int[] next) {
    init(owner, current, next, CELLS);
    System.arraycopy(current_pixels, 0, work_pixels,
                  0, pixels_per_cell);

    short random[][] = (short[][])object_table.get(
           getClass().getName() + pixels_per_cell);

    if(random == null) {
      random = createRandomArray(pixels_per_cell, cell_h);
      object_table.put(getClass().getName() + pixels_per_cell,
                    random);
    }

    for(int c = 0; c < CELLS; ++c) {
      try { Thread.sleep(100); } catch (InterruptedException e) {}
      int limit = random[c].length;
      for(int p = 0; p < limit; ++p) {
        int pixel_index = random[c][p];
        work_pixels[pixel_index] = next_pixels[pixel_index];
      }
      try { Thread.sleep(50); } catch (InterruptedException e) {}
      createCellFromWorkPixels(c);
    }
    work_pixels = null;
  }
}
```

下面是逐步淡化变换之前、淡化期间和淡化之后的显示：

28.2.6 SmashTransition.java

SmashTransition类在转换图像时将新图像从上方投入旧图像，旧图像看起来像是经受不住新图像的重量而挤压变形。

使用两个实例变量drop_amount和location创建帧。location变量保持被挤压的图像开始的像素。drop_amount变量保存新图像投入图像的每个帧的像素数目。换句话说，drop_amount是每个帧中增加到location变量中的数目。调用一个名为fill_pixels的静态数组把work_pixels数组中整行的像素转换为白色。

挤压的效果是通过将旧图像绘制成折叠形式而达到的。以将旧图像的第一行偏移至右边开始。随后的每一行再向右偏移一点。这样直至达到最大的偏移量。然后，逐行减少偏移量直到偏移量为零。连续进行这种操作直至被挤压图像的所有行都绘制完毕。

SmashTransition类并没有整行的绘制旧图像的行，它使用的长度略短于实际长度。

每一帧绘制的被挤压图像的行数都在减少，看起来旧图像好象越来越被挤压缩小。这个变换使用旧图像中均匀分布的行。这样保证被挤压图像看起来不象是从小应用程序的底部陷落或是在新图像下滑走。

setupFillPixels( )

使用setupFillPixels( )静态方法确保fill_pixels数组的初始化，而且数组的长度至少和小应用程序的行宽一样。如果这个数组没有初始化或是不够长，这个方法负责重建或是创建并填充这个数组。如果有多个小应用程序实例在运行，则小应用程序可以共享这个fill_pixels数组，但数组的长度至少和最宽的小应用程序的行宽一样。

init( )

这个转换的init( )方法和其他的转换一样，调用基类的init( )方法。随后，调用前面描述的setupFillPixels( )方法。然后计算变量drop_amount和location的初始值。在这之后，init( )方法进入循环创建每个帧。它实际上按照逆序的方式工作，先创建最后一个帧。它不必完全按逆序工作。然而，逆序循环可以节省class文件的一个字节码。在创建每个单元后，

location变量增加至下一个合适位置。

Smash( )

Smash( )方法为下一个单元修改work_pixels数组。它在work_pixels数组中构造旧广告牌的被挤压的图像，然后绘制新图像的像素。这个方法只有一个参数，max_fold，即这个参数折叠时的最大右偏移量。用行宽减去这个参数得到绘制折叠图像的长度。

这个方法从新图像中拷贝像素至work_ pixels数组开始，然后初始化一些绘制被挤压图像时要使用的变量。通过循环一行一行的绘制被挤压的图像。在这个循环中，它首先将整行涂白，然后拷贝旧广告牌的对应行至此行。为了得到折叠效果，该方法不在绘制白色行的起始位置绘制像素。相反，它将所有像素偏移向右。在绘制完整行后，增大偏移计数器。接着做一个边界检查，看偏移量是否超出最大或最小偏移量。如果超出，它取反每行增加的偏移量。这样做的效果看起来就是偏移方向反转。

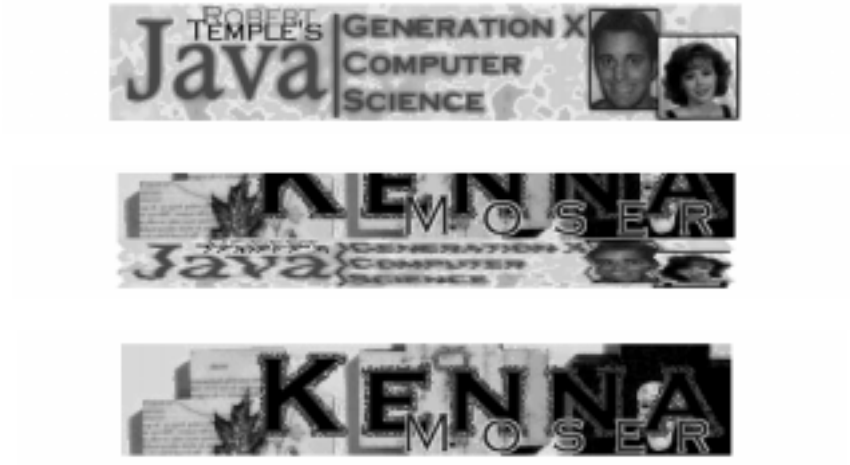
程序代码

下面是SmashTransition类的源代码：

```
import java.awt.*;
import java.awt.image.*;

public class SmashTransition extends BillTransition {
  final static int CELLS = 8;
  final static float FOLDS = 8.0f;
  static int[] fill_pixels;

  static void setupFillPixels(int width) {
    if(fill_pixels != null && fill_pixels.length <= width) {
      return;
    }
    fill_pixels = new int[width];
    for(int f = 0; f < width; ++f) {
      fill_pixels[f] = 0xFFFFFFFF;
    }
  }

  int drop_amount;
  int location;

  public void init(Component owner, int[] current, int[] next) {
    init(owner, current, next, CELLS, 160);
    setupFillPixels(cell_w);
    drop_amount =  (cell_h / CELLS) * cell_w;
    location = pixels_per_cell - ((cell_h / CELLS) / 2) * cell_w;
    for(int c = CELLS - 1; c >= 0; --c) {
      try { Thread.sleep(100); } catch (InterruptedException e) {}
      Smash(c + 1);
      try { Thread.sleep(150); } catch (InterruptedException e) {}
      createCellFromWorkPixels(c);
      location -= drop_amount;
    }
```

```
      work_pixels = null;
    }

    void Smash(int max_fold) {
      System.arraycopy(next_pixels, pixels_per_cell - location,
                   work_pixels, 0, location);
      int height = cell_h - location / cell_w;
      float fold_offset_adder = (float)max_fold * FOLDS / (float)height;
      float fold_offset = 0.0f;
      int fold_width = cell_w - max_fold;
      float src_y_adder = (float)cell_h / (float)height;
      float src_y_offset = cell_h - src_y_adder / 2;
      for(int p = pixels_per_cell - cell_w; p >= location; p -=
          cell_w) {
        System.arraycopy(fill_pixels, 0, work_pixels, p, cell_w);
        System.arraycopy(current_pixels, (int)src_y_offset * cell_w,
                     work_pixels, p + (int)fold_offset, fold_width);
        src_y_offset -= src_y_adder;
        fold_offset += fold_offset_adder;
        if(fold_offset < 0.0 || fold_offset >= max_fold) {
          fold_offset_adder *= -1.0f;
        }
      }
    }
  }
```

下面是挤压变换之前、挤压期间和挤压之后的显示：



28.2.7 TearTransition.java

TearTransition创造出当前广告牌破碎成纸片的幻觉效果。旧广告牌破碎着向上向左展现出下面的下一个广告牌图像。

这个变换中只有一个变量，x_cross。这个参数作为乘数创建破碎效果。这个变量值越大，其呈现的破碎效果越小。

这个变换有许多优化方法。其中一个显著优化是按逆序构建单元帧，先构建最后一个

单元帧，最后构建第一个帧。在正常顺序中，每个后面的单元帧展开一点下面的新图像。如果帧按正常顺序创建，绘制破碎效果时新图像的像素在当前帧中展开，这将覆盖了前一个帧的破碎效果。而按逆序创建单元帧，则只需重画每个单元帧的破碎效果。例如，最先创建的最后一个帧，只用最左上角的图像显示破碎效果，而其余的像素则完全取自新广告牌图像。在第二个创建的倒数第二帧中，新的破碎效果多绘制了一些像素在图像左上角，而其余的仍然保持不变。随后的单元帧一点一点的增加图像的破碎效果，但是总是在上个帧的基础上添加。

init( )

这个转换的init( )方法的启动方式和其他的转换一样，调用基类的init( )方法。然后将新广告牌中的像素拷贝进work_pixels数组，并将旧广告牌像素的第一行拷贝在work_pixels数组的第一行。在x_cross变量初始化之后，init( )方法按逆序产生每个单元。在循环中，它创建单元然后减少x_cross变量的值。

Tear( )

Tear( )方法为下一个单元修改work_pixels数组。它在工作像素上绘制破碎效果。它按行绘制破碎效果。每一行，这个方法将旧图像的像素拷贝至work_pixels数组。它使用两个计数器，一个是work_pixels数组的下标，destination，另一个是旧广告牌像素数组的下标，source。两个计数器的初始值都为零。目的（destination）计数器每次加一。可是源（source）计数器的增值为浮点数，这些数字常常大于1。运行循环直至目的下标大于行宽，其结果是源下标比目的下标增长得快。其总体效果是只有源图像左边的一部分像素被拷贝至目的地。从源图像抽取像素时，会跳过一些像素，因此源图像抽取的像素均匀的分布在行间。

单元帧的每一行使用比上一行的浮动指针更大的值。这使得底部的行中被绘制的像素比顶部的行中被绘制的像素少，这构成了破碎效果。

这个方法使用了一个较大的优化方法。当使用数组中的元素时，必须进行边界检查以确保下标在数组的边界中。在边界检查中设计一个性能点击数。标准的Java类System，提供了一个在数组间拷贝数据和拷贝单个数组元素几乎一样快的方法。这种方法用于加速在单元帧中创建行的操作。但是这个方法只能用在小应用程序知道源像素是连续的情况下。如果小应用程序在源图像像素之间跳跃，则只能使用标准的循环方法。一个x_increment值低于0.5指示每次源下标计数器增加1.5，特殊行使用这种数组拷贝方法可以提高速度。

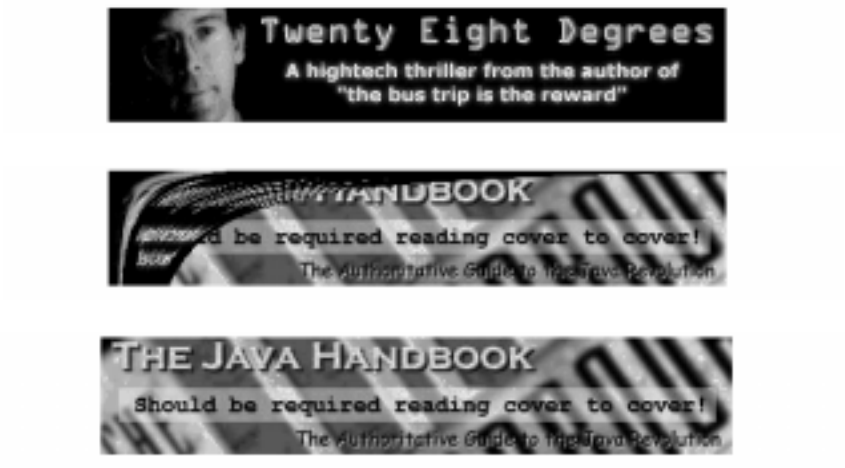
程序代码

下面是TearTransition类的源代码：

```
import java.awt.*;
import java.awt.image.*;
public class TearTransition extends BillTransition {
  static final int CELLS = 7;
  static final float INITIAL_X_CROSS = 1.6f;
  static final float X_CROSS_DIVISOR = 3.5f;
  float x_cross;
```

```
  public void init(Component owner, int[] current, int[] next) {
    init(owner, current, next, CELLS);
    System.arraycopy(next_pixels, 0, work_pixels, 0,
                     pixels_per_cell);
    System.arraycopy(current_pixels, 0, work_pixels, 0, cell_w);

    x_cross = INITIAL_X_CROSS;

    for(int c = CELLS - 1; c >= 0; --c) {
      try { Thread.sleep(100); } catch (InterruptedException e) {}
      Tear();
      try { Thread.sleep(150); } catch (InterruptedException e) {}
      createCellFromWorkPixels(c);
      x_cross /= X_CROSS_DIVISOR;
    }
    work_pixels = null;
  }

  final void Tear() {
    float x_increment;
    int p, height_adder;

    p = height_adder = cell_w;
    for (int y = 1; y < cell_h; ++y) {
      x_increment = x_cross * y;
      if(x_increment >= 0.50f) {
        float fx = 0.0f;
        x_increment += 1.0f;
        int x = 0;
        do {
          work_pixels[p++] = current_pixels[height_adder + x];
          x = (int)(fx += x_increment);
        } while(x < cell_w);
      }
      else {
        float overflow = 1.0f / x_increment;
        float dst_end = overflow / 2.0f  + 1.49999999f;
        int dst_start = 0, src_offset = 0, length = (int)dst_end;
        while(dst_start + src_offset + length < cell_w) {
          System.arraycopy(current_pixels, p + src_offset,
                           work_pixels, p, length);
          ++src_offset;
          dst_end += overflow;
          p += length;
          dst_start += length;
          length = (int)dst_end - dst_start;
        }
        length = cell_w - src_offset - dst_start;
        System.arraycopy(current_pixels, p + src_offset,
                         work_pixels, p, length);
      }
      p = height_adder += cell_w;
    }
  }
}
```

下面是破碎变换之前、变换期间和变换之后的显示：



28.2.8　UnrollTransition.java

UnrollTransition看起来像一个卷起来的海报被放置在小应用程序底下，然后向上展开，逐渐的展开下一幅图像覆盖老图像。为了增强这种打开效果，卷积部分的体积在向上展开的过程中逐渐缩小。

在创建展开转换中，使用两个变量实例。location变量指示像素数组中的像素，它存储着卷积部分将要展现的当前像素。unroll_amount数组变量告诉类每一帧中卷积部分应向上移动的垂直像素数。

构造每个单元帧中最困难的部分是绘制卷积部分。每一帧中其他任务所要完成的就是将新图像的像素填满前一帧卷积部分腾出的空间。

卷积部分用新图像的扫描线绘制。卷积部分的第一行使用位于Y坐标上的扫描行，其中Y是卷积部分在小应用程序的位置。例如，如果一个单元帧中的卷积部分位于第10行，则使用新图像的第9行绘制卷积部分的第1行。卷积部分的后来行用位于前面一行上的新图像的上一行绘制。就是，接着上例，使用新图像中的第8行绘制卷积部分的第2行。

卷积部分的三维外观是通过每一行略微向左偏移一点而达到的。在卷积部分中的行比其上下的行的偏移量都要大一点。卷积部分的顶部和底部的行带一点阴影看起来好象小应用程序顶部有灯光照射。这样的结果是顶部的行比卷积部分的其他行要亮一点而底部的行要黑一点。

createUnrollAmountArray( )

这个变换中的连续单元帧比前一个单元帧略少打开一点卷积部分。使用createUnrollAmountArray( )静态方法计算指示每个单元帧中应打开多少卷积部分。

init( )

这个转换的init( )方法的启动方式和其他的转换一样，调用基类的init( )方法。然后初始化location变量为指向像素数组的最后一个像素的下标。其后跟着将所有旧广告牌中的像素

拷贝至work_pixels数组。

按小应用程序的高度在object_table散列表中抽出一个存放每个帧中展开的像素的数组。如果表中没有这样的数组，创建数组且将数组存放在object_table表中。

然后，init( )方法循环处理每个单元，通过消减location变量将卷积部分向上展开，绘制每个单元帧。在处理器密集操作的单元帧创建工作之前和之后，当前线程终止一会儿，以运行Java多线程环境中的其他线程。在调用createCellFromWorkPixels( )在work_pixels数组中创建单元帧后，用新图像的像素绘制卷积部分。这为下一帧准备了work_pixels数组。

Unroll( )

Unroll( )方法为下一单元修改Unroll( )数组。它在工作像素数组上绘制卷积部分。这个方法首先计算需要绘制的卷积部分每一行的偏移量，然后循环处理卷积部分的每一行，将新图像的扫描行拷贝至work_pixels数组中。对应于每一行的左偏移量的展开的像素由静态数组fill_pixels的像素填充。

另一个循环处理卷积部分顶部和底部的每个像素值，加亮顶部的像素，增黑底部的像素。

程序代码

下面是UnrollTransition类的源代码：

```
import java.awt.*;
import java.awt.image.*;

public class UnrollTransition extends BillTransition {
  final static int CELLS = 9;
  static int fill_pixels[] = { 0xFFFFFFFF, 0xFF000000,
                          0xFF000000, 0xFFFFFFFF };

  private static int[] createUnrollAmountArray(int cell_h) {
    float unroll_increment =
      ((float)cell_h / (float)(CELLS + 1)) /
      ((float)(CELLS + 2) / 2.0f);

    int total = 0;
    int unroll_amount[] = new int[CELLS + 1];
    for(int u = 0; u <= CELLS; ++u) {
      unroll_amount[u] = (int)(unroll_increment * (CELLS - u + 1));
      total += unroll_amount[u];
    }
    if(total < 0) {
      unroll_amount[0] -= 1;
    }
    return unroll_amount;
  }

  int location;
  int[] unroll_amount;
  public void init(Component owner, int[] current, int[] next) {
    init(owner, current, next, CELLS, 220);
    location = pixels_per_cell;
```

```
  System.arraycopy(current_pixels, 0,
                   work_pixels, 0, pixels_per_cell);
  unroll_amount = (int[])object_table.get(getClass().getName() +
                   cell_h);
  if(unroll_amount == null) {
    unroll_amount = createUnrollAmountArray(cell_h);
    object_table.put(getClass().getName() + cell_h, unroll_amount);
  }

  for(int c = 0; c < CELLS; ++c) {
    location -= unroll_amount[c] * cell_w;
    try { Thread.sleep(150); } catch (InterruptedException e) {}
    Unroll(c);
    try { Thread.sleep(100); } catch (InterruptedException e) {}
    createCellFromWorkPixels(c);
    System.arraycopy(next_pixels, location,
                     work_pixels, location,
                     unroll_amount[c] * cell_w);
  }
  work_pixels = null;
}

void Unroll(int c) {
  int y_flip = cell_w;
  int offset[] = new int[unroll_amount[c]];
  for(int o = 0; o < unroll_amount[c]; ++o) {
    offset[o] = 4;
  }
  offset[0] = 2;
  if(unroll_amount[c] > 1) {
    offset[1] = 3;
  }
  if(unroll_amount[c] > 2) {
    offset[unroll_amount[c] - 1] = 2;
  }
  if(unroll_amount[c] > 3) {
    offset[unroll_amount[c] - 2] = 3;
  }

  int offset_index = 0;
  int end_location = location + unroll_amount[c] * cell_w;
  for(int p = location; p < end_location; p += cell_w) {

    System.arraycopy(next_pixels,
                     p - y_flip + offset[offset_index],
                     work_pixels,
                     p, cell_w - offset[offset_index]);

    System.arraycopy(fill_pixels, 0,
                     work_pixels,
                     p + cell_w - offset[offset_index],
                     offset[offset_index]);

    ++offset_index;
    y_flip += cell_w + cell_w;
  }

  for(int x = location + cell_w - 1; x > location; --x) {
```

```
        work_pixels[x] |= 0xFFAAAAAA;
        work_pixels[x + unroll_amount[c]] &= 0xFF555555;
      }
    }
  }
```

下面是展开变换之前、变换期间和变换之后的显示：

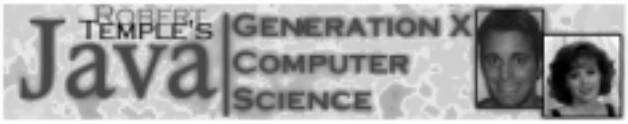
28.3 动 态 代 码

Robert展示了如何在Java的明显受限的环境中开发高性能的交互式图形的成功范例。他显示如何使用System.arraycopy( )有效地四处拷贝像素数据。他还显示了如何正确使用协同操作的多线程完成计算和在用户不等待的背景中进行网络传输。他显示了任何动态装载类文件而不出现总和Java 小应用程序相关的起始“灰色等待状态”的病症。Robert证明了只要足够细心，就能用Java实现高性能的直接像素操作算法。

除包含有趣的代码之外，DynamicBillboard是用户或是其他非程序员喜欢的小应用程序。HTML编辑器可以方便的配置这个小应用程序，Java编程人员可以很容易扩展这个小应用程序，web用户也觉得这个小应用程序很有趣。在这个“点击然后进入”的广告时代，广告人员只想为那些通过内容站点而进入本站的流量付费，Robert的小应用程序可以增加流量而最终带来收入。


Family
Planet


Java
COMPUTER
SCIENCE


ROBERT
TEMPLE'S
Java
GENERATION X
COMPUTER
SCIENCE


KENNA
MOSER
Java


KENNA
MOSER


Twenty Eight Degrees
A hightech thriller from the author of
"the bus trip is the reward"


HANDBOOK
be required reading cover to cover!


THE JAVA HANDBOOK
Should be required reading cover to cover!

第 29 章　ImageMenu：一个基于图像的 Web 菜单

ImageMenu小应用程序是一个简单程序，它显示一个图像菜单，这个菜单有个垂直列表包含任意数目的选项。当用户用鼠标点击这些选项，在光标下的选项的外观发生变化，指示它为选中项。当用户点击一个选项后，web浏览器根据选择项选择一个新文档显示。ImageMenu由David LaVallée编写, 他还完成了多个有趣的小应用程序。图29-1显示的是一个ImageMenu的实例。

图 29-1　一个 ImageMenu 的实例

ImageMenu使用AppletContext类的showDocument( )方法，将超文本文件显示在新的页面上。ImageMenu的创新之处在于它使用了一个单独源图像的不同部分在屏幕上绘制菜单。将菜单基于图像而不是文本，可使用户挑选自己喜欢的字体或图像来设计菜单。同时，用户可以提供不同类型的选择反馈，而不再拘泥于AWT的有限功能。

ImageMenu 小应用程序受一个名为Navigation的小应用程序启发，Navigation 小应用程序由一个出类拔萃的Java程序员Sean Welch完成。Navigation和ImageMenu的不同之处在于带宽的使用效率和applet标记的说明。Navigation小应用程序使用一个源图像，图像的宽度等于小应用程序的宽度乘以可选项的宽度，用以显示小应用程序的所有状态。两个小应用程序都下载一个图像，这样比通过Internet下载多个文件更有效率。一个有7个选择项的菜单的Navigation小应用程序(100×140像素)需要700×140大小的图形。而在这里描述的小应用程序，ImageMenu，使用2倍于小应用程序宽度的图像，即200×140。大多数的Web设计人员讨厌敲键盘，因此导致了Navigation 和ImageMenu之间的第二个显著区别：简短的小应用程序参数。

ImageMenu的效率较高，它只使用一个较小的源图像和更少字节的参数，而Welch的Navigation有一个无法取代的特征——它可以显示单个的选择状态并将其扩展到下一个菜

单项区域中。ImageMenu小应用程序需要限制每个菜单项在一个矩形区域不能侵占其邻近选项的区域。这禁止了在一行内上行字母（如h）与上行的下行字母（如j）之间的叠置。

29.1　源 图 像

虽然本书没有给出Navigation的源代码，观察下面的GIF图片可以清楚地了解它的功能。在图29-2中的Navigation的源图像显示了一个7列的菜单，每一列提供了一个可视化的选项。然而那个可选的条目只有两个状态，因此每一行有5个冗余的不能选择的状态。

图 29-2　Navigation 小应用程序的源图像

ImageMenu的源图像显示在图29-3中。这个图像简单的提供了一个6个选项的菜单的7种可能状态。首先，drawImage( )显示了源图像的左半边，其状态是没有一个选项被选中。如果选中其中任何一项，被选项周围显示一个剪切矩形框，然后使用drawImage( )显示图像右半边。它只绘制由剪切框限定的单元。

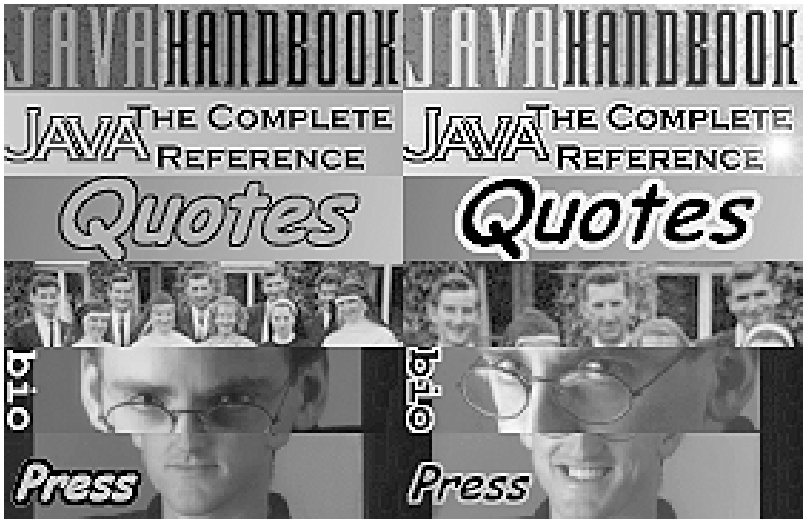


图 29-3　ImageMenu 的源图像

29.2 APPLET标记

ImageMenu的APPLET 标记包含多条信息。使用java.util.StringTokenizer 方法读出urlList和targetList帧参数，这些参数用加号分隔标记。同时计算每个菜单的坐标，将小应用程序的高度除以urlList中URL的个数。为了APPLET标记的可读性，当转移到新页面时，允许使用前缀和后缀和一个URL相连。

```
<applet code="ImageMenu" width=140 height=180 hspace=0 vspace=0>
<param name="img" value="menu.jpg">
<param name="urlPrefix"
         value="http://www.osborne.com/">
<param name="urlList"
         value="aboutus/+whatsnew/+download/+feedback/">
<param name="targetList"
         value="_self+_self+_self+_self+_self+_self">
<param name="urlSuffix" value="index.htm">
</applet>
```

29.3 方 法

这是一个小的小应用程序——只有100行Java源代码。我们将仔细研究全部的8个方法，并在本章结束之前显示全部源代码。

29.3.1 init( )

在小应用程序初始化时，init( )方法将大小保存在Dimension的d变量中，并解析小应用程序的param标记。然后使用StringTokenizer解析由加号分隔的字符串以创建字符串数组url和target。解析出的URL数目用作计算菜单项的高度。在完成计算之后，init( )将菜单项的数目和高度分别存放在cells和cellH变量中。

29.3.2 update( )

禁止小应用程序的update( )方法以避免闪烁。正如第23章中提到的那样，小应用程序父类的update( )方法在调用paint( )之前为矩形区域填充背景色。因为我们不使用repaint( )方法，所以这里完全取消更新。

29.3.3 lateInit( )

lateInit( )私有方法创建屏幕外的Image对象，利用这个对象双重缓冲菜单显示。这个方法同时使用MediaTracker对象同步获取源图像。

29.3.4 paint( )

paint( )方法非常简单。首先检查是否已经创建屏幕外的缓冲区。如果还没有，调用lateInit( )方法创建缓冲区并载入菜单图像。

随后，paint( )方法在屏幕外缓存区中绘制图像的左半部。当然，菜单图像应是小应用程序宽度的两倍。这样，在调用drawImage(img, 0, 0, null)时，小应用程序简单的剪切图像菜单的右半部。然后，如果有菜单项被选中(selectedCell >= 0)，小应用程序将剪切矩形设置在所选菜单项的边框上。可以注意到每次paint( )获取屏幕外图像的图形内容。这对重新设置图像边框的剪切区域有一定的影响。AWT缺乏一个resetClip( )方法，因此需要特殊的编码方案。

下一步，整个图像被重新绘制，但是这次，通过调用drawImage(img, -d.width, 0, null)方法偏移小应用程序宽度到左边。这相当于仅放置右边剪切框中的高亮度的菜单项。最后，屏幕外缓冲区被拷贝到小应用程序的窗口中。

注意：大多数的图形显示速度由CPU直接访问屏幕的速度决定。而且，许多现代的显示卡优化了拷贝存储器内的矩形区域显示在支持的窗口系统中的操作。因此最好在屏幕外的缓冲区中绘制矩形区域而不是直接在屏幕上拷贝数据。在类似的PC系统中，随着像素深度和显卡结构的不同，每秒可切换10到400个缓冲区图形。

29.3.5　mouseExited( )

mouseExited( )方法需要特殊处理，因为这个方法禁止所有的菜单项。必须将selectedCell和oldCell设成-1，这使得随后的paint( )方法将这些菜单项显示为不能选择。将oldCell设成-1意味着下次鼠标进入小应用程序将调用mouseMoved( )方法，正确绘制菜单的第一项。

29.3.6　mouseDragged( )

在按键被按下时鼠标移动将调用mouseDragged( )方法。在这个小应用程序中，对鼠标拖曳或移动的反应相同，因此简单地直接调用mouseMoved( )，并传递接收到的参数。

29.3.7　mouseMoved( )

鼠标移动时，mouseMoved( )方法检查y坐标看是否有菜单项被选中。如果selectedCell与oldCell不同，这意味着用户从一个单元移动到另一个单元，则重新绘制菜单。这是一个优化，避免了每次鼠标移动时引起的对同一屏幕的重画。注意，这里并没有调用repaint( )。程序走了个捷径，在从getGraphics( )中获取了Graphics内容后，通过正常的小应用程序协议直接调用paint( )。这样做使程序的反应增加迅速。在菜单绘制完毕后，重新设置状态栏以反映保存在oldCell中的最新选项，

29.3.8　mouseReleased( )

mouseReleased( )方法让浏览器显示当前被选菜单项所对应的URL。首先构建所需的URL。如果APPLET标记中的URL的格式不正确，状态栏中显示异常，然后放弃交换文件返回。showDocument( )方法将URL指定的文件放置在target数组列出的帧中。作为最后一个特点，调用MouseEvent的isShiftDown( )方法将检查SHIFT键的状态。如果已经按下SHIFT键，则不是在所指定的target帧中，而是在一个新的空白窗口中打开URL。

29.3.9 程序代码

ImageMenu的源代码如下所示：

```
import java.awt.* ;
import java.awt.event.*;
import java.applet.*;
import java.util.*;
import java.net.*;

public class ImageMenu extends Applet {
  Dimension d;

  Image img, off;
  Graphics offg;
  int MAXITEMS = 64;
  String url[] = new String[MAXITEMS];
  String target[] = new String[MAXITEMS];
  String urlPrefix, urlSuffix;
  int selectedCell = -1;
  int oldCell = -1;
  int cellH;
  int cells;

  public void init() {
    d = getSize();
    urlPrefix = getParameter("urlPrefix");
    urlSuffix = getParameter("urlSuffix");
    StringTokenizer st;
    st = new StringTokenizer(getParameter("urlList"), "+");
    int i=0;
    while(st.hasMoreTokens() && i < MAXITEMS)
       url[i++] = st.nextToken();
    cells = i;
    cellH = d.height/cells;
    st = new StringTokenizer(getParameter("targetList"), "+");
    i=0;
    while(st.hasMoreTokens() && i < MAXITEMS)
      target[i++] = st.nextToken();
    addMouseListener(new MyMouseAdapter());
    addMouseMotionListener(new MyMouseMotionAdapter());
  }

  private void lateInit() {
    off = createImage(d.width, d.height);
    try {
      img = getImage(getDocumentBase(), getParameter("img"));
      MediaTracker t = new MediaTracker(this);
      t.addImage(img, 0);
      t.waitForID(0);
    } catch(Exception e) {
      showStatus("error: " + e);
    }
  }
```

```
  public void update(Graphics g) {}
  public void paint(Graphics g) {
    if(off == null)
      lateInit();

    offg = off.getGraphics();
    offg.drawImage(img, 0, 0, this);
    if (selectedCell >= 0) {
      offg.clipRect(0, selectedCell * cellH, d.width, cellH)
      offg.drawImage(img, -d.width, 0, this);
    }
    g.drawImage(off, 0, 0, this);
  }

  class MyMouseMotionAdapter extends MouseMotionAdapter {
    public void mouseDragged(MouseEvent me) {
      mouseMoved(me);
    }
    public void mouseMoved(MouseEvent me) {
      int y = me.getY();
      selectedCell = (int)(y/(double)d.height*cells);
      if (selectedCell != oldCell) {
        paint(getGraphics());
        showStatus(urlPrefix + url[selectedCell] + urlSuffix);
        oldCell = selectedCell;
      }
    }
  }

  class MyMouseAdapter extends MouseAdapter {
    public void mouseExited(MouseEvent me) {
      selectedCell = oldCell = -1;
      paint(getGraphics());
      showStatus("");
    }

    public void mouseReleased(MouseEvent me) {
      URL u = null;
      try {
        u = new URL(urlPrefix + url[selectedCell] + urlSuffix);
      } catch(Exception e) {
        showStatus("error: " + e);

     }
      if (me.isShiftDown())
        getAppletContext().showDocument(u, "_blank");
      else
        getAppletContext().showDocument(u, target[selectedCell]);
    }
  }
}
```

29.4 小 结

在使用中，ImageMenu小应用程序效果很好，为许多小程序提供了可以平衡的选择。使用小应用程序中的showDocument（URL u，String target）方法是Web页设计时的一个优化策略。如果在一个基于帧结构的HTML中将ImageMenu小应用程序放入一个帧中，利用这个小应用程序将文件发送到第二个帧中，不必重载小应用程序，提高了效率，满足了用户需求。

第 30 章　Lavatron 小应用程序：运动竞技场的显示牌

Lavatron是一个运动竞技场显示程序。正常情况下，一个小应用程序并没有很多的历史，但是这个小应用程序却有。David LaVallée，第29章中ImageMenu 小应用程序的作者，希望达到这种效果已经很长时间了。Lavatron的历史可以追溯到1974年，当LaVallée还是一个美国曲棍球联合会的加利福尼亚黄金海岸队的忠实小球迷时。David回忆到，“我们的记分牌只显示分数。比赛是这样的一样东西：即使有呜呜哇哇的风琴声也无法转移曲棍球迷的注意力。”

1979年，当LaVallée是在加拿大国家体育馆中(Toronto Blue Jays总是在那里比赛) 运行记分牌程序的DEC的（数字设备公司）PDP 11/34的维修人员时，对编写一个图形化的记分牌程序的想法入了迷。那个记分牌是用普通的100瓦的家中常见的日光灯组成。1991年，多伦多市在Skydome装上了索尼公司的Jumbotron高清晰度电视（HDTV）计分牌：真实的色彩，图像，视频，大小是Hard Rock咖啡馆高度的三倍。1992年，LaVallée用面向对象的C和PostScript编写了第一版的Lavatron。1995年，用Java重写了Lavatron，它经过了一些性能上的调整和反复。目前在这里展示的是用Java 2更新过的版本。

仍然有许多可以尝试的工作可以增强Lavatron（见图30-1），例如在存储器中动态的绘制源图片而不是下载图片，或是滚动图片实现动画效果。但是这是一个可以利用的有趣而且生动的显示小应用程序。

图 30-1　Lavatron 小应用程序运行状态——其源图像在小应用程序的下面显示

30.1 Lavatron的工作过程

Lavatron使用了小技巧，可以在屏幕上显示非常有趣的图片，它的一个特点是加载小应用程序的速度非常的快。其加载速度快的原因是因为不需要通过网络传输很多的数据。源图像是一个JPEG的图像，比显示出来的图像小64倍。源图像中的每个像素被扩展成一个8×8的正方形。这是一个Lavatron用来生成日光灯效果的技巧。一个透明圆周环绕黑色斜面的8×8像素图形，带一个破折号形式的高亮度区域，是扩展的颜色像素。作为一个优化策略，灯泡可以预先装配成一次绘制一列的图像。图31-2显示灯泡扩大的效果。两个白色的像素是高亮度的。在角落里的黑色像素是不透明的。而在中心的灰色像素是透明的，允许灯泡颜色穿过。

图 30-2 一个放大的灯泡图像

因为Lavatron不必重新绘制已经画好的区域，所有它的绘画速度很快。拷贝屏幕区域是项好技术，它和只绘制图形的新的部分的技术使用在图形滚动的常见操作中。awt.Graphics中的copyArea( )功能将图形的矩形区域以当前位置为起始位置，偏移x,y。正如图形速度的优化方法，copyArea( )方法非常成功。它比其他的图形绘制技术，如drawImage( )，或是使用clipRect( )的drawImage( )，要好得多。构建一个比小应用程序自身大许多的由多个源图像连接而成的一个单一图像，然后使用copyArea将其移入位置，剪切其结果并显示在屏幕上，是一个快速的Java绘图技术。

30.2 源 代 码

Lavatron以初始化数据开始，初始化工作包括加载源图像，创建灯泡图像的列表。初始化工作的最后阶段是绘制一个全是灰暗（黑色）灯泡的屏幕外（备份存储）图形，作为图形显示的开始。随后，使用copyArea( )将图形当前位置向左移动一个列宽度并加至右边。读出下一列的像素值作为填充小应用程序右边的8×8矩形列的颜色。绘制灯泡的透明列，

然后在屏幕上画出整个图像。因为小应用程序除了滚动图像外不做其他的工作，所以它放弃了普通的repaint( )循环，而是创建一个线程，让这个线程重复调用paint( )方法，只在调用yield( )时暂停好让其他线程运行。

30.2.1　APPLET标记

源代码以Lavatron的APPLET标记开始，如下所示。当小应用程序的宽度是灯泡的偶数倍，高度是灯泡的尺寸乘以源图像的高度时这个小应用程序的外观最好。Applet的惟一参数是源图像文件的名字，存放在img中。

```
<applet code=Lavatron.class width=560 height=128>
<param name="img" value="swsm.jpg">
</applet>
```

30.2.2　Lavatron.java

主小应用程序很小，大约100行Java源代码。然而，它需要一个支持的类，在下一小节中描述。

init( )

首先，init( )方法使用getSize( )方法获取小应用程序的大小，将其四舍五入为灯泡大小的整数倍，存放在offw和offh变量中，灯泡的大小由bulbS指定。接着，按此大小创建一个名为offscreen的图像，作为显示图像的存储备份。利用Graphics对象在offscreen上绘制图像，然后保存在offGraphics中。小应用程序的大小，以灯泡为单位而不是像素，存放在bulbsW和bulbsH变量中。

在接下来的步骤中，传递图像大小参数，调用createBulbs( )创建灯泡组的图像。然后，载入在小应用程序参数img中指明的图像。这是通过将getImage( )的结果传递给MediaTracker的 addImage( )方法，然后调用waitForID( )，等待图像完全载入后再返回等一系列步骤完成的。

要绘制一个放大版本的图形，init( )需要重新获取图形中每个像素的颜色信息。首先，使用getWidth( ) 和getHeight( )取得图形的大小，将宽度存放在pixscan中。然后分配pixels为一个pixscan * h个整数的新数组。创建PixelGrabber。在调用grabPixels( )时，用颜色值填充数组。

init( ) 的最后一个步骤是在offscreen图像上绘制黑色的灯泡，当图像从右边滚动显示被点亮的灯泡时，黑色的灯泡使得显示效果更加生动。

createBulbs( )

createBulbs( )方法是init( )的辅助方法，它返回了一个图像的灯泡堆栈，可用这个图像屏蔽出一组着色的正方形，使其看起来像点亮的日光灯。它用了一点技巧，但是非常的有效。

首先，它分配了一个数目适当的整数数组来存放像素。然后，它声明了另一个数组，这个数组是用数字0，1和2表示的单个球形图片。其中，0代表黑色，1代表透明的像素，2

代表白色的加亮区。接着，createBulbs( )方法中定义了一个短数组——bulbCLUT（球形颜色查询表）。它将0，1和2映射为32比特的像素值。0xff000000是完全黑。高字节是alpha或透明。0x00c0c0c0是完全透明浅灰色，0xffffffff为不透明的白色。

for循环处理每个像素，从基于列位置的bulbBits中装载合适的0，1或2。这通过使用取模操作(%)完成。使用这个值在bulbCLUT中查询颜色。在确定像素数组后，createBulbs( )返回createImage( )输出，传入在由刚刚构建的像素所准备的MemoryImageSource对象中。

color( )

color( )方法返回一个源图像中x,y位置的像素颜色的Color对象。因为这个小应用程序连续运行，所有在每次绘制灯泡时不是简单的创建一个新的Color对象。这样滥用了垃圾回收。相反，每个不同的Color对象存放在散列表中。散列表中Color对象的个数最多可达源图像的高乘以宽那么多，但实际上通常要小于这个数目。

update( )

Lavatron重载update( )方法，使之为空，因为不希望出现AWT实现引起的闪烁现象。

paint( )

paint( )方法非常简单。第一步调用copyArea( )将所有的列向左移一列的宽度。然后使用一个for循环使用color()方法在最右边的列中填入Color中适当像素。然后在新列上绘制bulb图形条。接着，当前滚动位置，scrollX，以宽度pixscan为模，被更新为右边的列，

start( ), stop( )和 run( )

当小应用程序开始运行后，它创建并启动一个名为t的新线程。这个线程调用run( )，run( )将尽可能快的调用paint( )，同时保持运行调用yield( )使其他线程可以运行。当调用小应用程序的stop( )方法时，stopFlag被设为true。run( )方法中的无限循环语句检查stopFlag变量。在stopFlag变为true时，程序跳出循环，终止。

一个非常有趣的改进策略是引入一个线程帧速率，例如30 fps（每秒的帧数），如果显示的速率过快则不调用yield( )而是调用sleep( )。如果不采取方法保持固定的帧速率，小应用程序看起来会过快。

程序代码

下面是Lavatron类的源代码：

```
import java.applet.*;
import java.awt.* ;
import java.awt.image.* ;

public class Lavatron extends Applet implements Runnable {
  int scrollX;
  int bulbsW, bulbsH;
  int bulbS = 8;
  Dimension d;
  Image offscreen, bulb, img;
```

```
Graphics offgraphics;
int pixels[];
int pixscan;
IntHash clut = new IntHash();
boolean stopFlag;

public void init() {
  d = getSize();
  int offw = (int) Math.ceil(d.width/bulbS) * bulbS;
  int offh = (int) Math.ceil(d.height/bulbS) * bulbS;
  offscreen = createImage(offw, offh);
  offgraphics = offscreen.getGraphics();
  bulbsW = offw/bulbS;
  bulbsH = offh/bulbS;

  bulb = createBulbs(bulbS, bulbsH*bulbS);
  try {
    img = getImage(getDocumentBase(), getParameter("img"));
    MediaTracker t = new MediaTracker(this);
    t.addImage(img, 0);
    t.waitForID(0);
    pixscan = img.getWidth(null);
    int h = img.getHeight(null);
    pixels = new int[pixscan * h];
    PixelGrabber pg = new PixelGrabber(img, 0, 0, pixscan, h,
                                pixels, 0, pixscan);
    pg.grabPixels();
  } catch (InterruptedException e) { };
  scrollX = 0;
  // paint black bulbs on the offscreen image
  offgraphics.setColor(Color.black);
  offgraphics.fillRect(0, 0, d.width, d.height);
  for (int x=0; x<bulbsW; x++)
    offgraphics.drawImage(bulb, x*bulbS, 0, null);
}

Image createBulbs(int w, int h) {
  int pixels[] = new int[w*h];
  int bulbBits[] = {
    0,0,1,1,1,1,0,0,
    0,1,2,1,1,1,1,0,
    1,2,1,1,1,1,1,1,
    1,1,1,1,1,1,1,1,
    1,1,1,1,1,1,1,1,
    1,1,1,1,1,1,1,1,
    0,1,1,1,1,1,1,0,
    0,0,1,1,1,1,0,0
  };
  int bulbCLUT[] = { 0xff000000, 0x00c0c0c0, 0xffffffff };
  for (int i=0; i<w*h; i++)
    pixels[i] = bulbCLUT[bulbBits[i%bulbBits.length]];
  return createImage(new MemoryImageSource(w, h, pixels, 0, w));
}

public final Color color(int x, int y) {
```

```
    int p = pixels[y*pixscan+x];
    Color c;
    if ((c=(Color)clut.get(p)) == null)
      clut.put(p, c = new Color(p));
    return c;
  }

  public void update() {}

  public void paint(Graphics g) {
    offgraphics.copyArea(bulbS, 0, bulbsW*bulbS-bulbS, d.height,
                         -bulbS, 0);
    for (int y=0; y<bulbsH; y++) {
      offgraphics.setColor(color(scrollX, y));
      offgraphics.fillRect(d.width-bulbS, y*bulbS, bulbS, bulbS);
    }
    offgraphics.drawImage(bulb, d.width-bulbS, 0, null);
    g.drawImage(offscreen, 0, 0, null);
    scrollX = (scrollX + 1) % pixscan;
  }

  Thread t;
  public void run() {
    while (true) {
      paint(getGraphics());
      try{t.yield();} catch(Exception e) { };
      if(stopFlag)
        break;
    }
  }

  public void start() {
    t = new Thread(this);
    t.setPriority(Thread.MIN_PRIORITY);
    stopFlag = false;
    t.start();
  }

  public void stop() {
    stopFlag = true;
  }
}
```

30.2.3 IntHash( )

正如前面小节中提到的那样，Color对象存放在散列表中而不是一个接着一个的创建相同实例。而且为了更进一步的优化，程序实现了一个自己的Java版本的Hashtable类，使用正常的整数而不是对象句柄作为关键字。

相对于Color对象，整数需要较少的存储空间，因此使用散列表作为按单个像素的整数值来查询Color对象的机制。使用每个像素的整数值动态创建一个Color对象非常昂贵，因为它制造了许多必须回收的内存垃圾。一个可行的解决方案是使用Java的散列表，希望这么做可以仅制造一定数目的垃圾。因为在一个标准版的Java散列表中，只有对象才能做关键

字。因此，为了能在Java的散列表中存放整数，必须创建一个新的Integer对象作为匹配操作的关键字。像Lavatron这样任务繁重的小应用程序，每秒都会创建数千个垃圾Integer对象。这并不是一个很好的解决方案。

正确的方法是构造自己的散列表，IntHash，这个散列表使用整数而不是Integer对象做关键字。IntHash大约是60行的代码。IntHash类复制了java.util.Hashtable类的接口，但其put( )和get( )方法的参数类型是整型而不是对象。在这一章中没必要解释散列表的工作原理，只要说put(42，"Hello") == get(42)就足够了。

程序代码

下面是IntHash类的源代码：

```
class IntHash {
  private int capacity;
  private int size;
  private float load = 0.7F;
  private int keys[];
  private Object vals[];

  public IntHash(int n) {
    capacity = n;
    size = 0;
    keys = new int[n];
    vals = new Object[n];
  }

  public IntHash() {
    this(101);

  }

  private void rehash() {
    int newcapacity = capacity * 2 + 1;
    Object newvals[] = new Object[newcapacity];
    int newkeys[] = new int[newcapacity];
    for (int i = 0; i < capacity; i++) {
      Object o = vals[i];
      if (o != null) {
        int k = keys[i];
        int newi = (k & 0x7fffffff) % newcapacity;
        while (newvals[newi] != null)
          newi = (newi + 1) % newcapacity;
        newkeys[newi] = k;
        newvals[newi] = o;
      }
    }
    capacity = newcapacity;
    keys = newkeys;
    vals = newvals;
  }

  public void put(int k, Object o) {
```

```
    int i = (k & 0x7fffffff) % capacity;
    while (vals[i] != null && k != keys[i]) // hash collision.
      i = (i + 1) % capacity;
    if (vals[i] == null)
      size++;
    keys[i] = k;
    vals[i] = o;
    if (size > (int)(capacity * load))
      rehash();
  }

  public final Object get(int k) {
    int i = (k & 0x7fffffff) % capacity;
    while (vals[i] != null && k != keys[i]) // hash miss
      i = (i + 1) % capacity;
    return vals[i];
  }

  public final boolean contains(int k) {
    return get(k)!=null;
  }

  public int size() {
    return size;
  }

  public int capacity() {
    return capacity;
  }
}
```

30.3 小 结

这个小应用程序是另一个让Java具有令人吃惊的性能的小例子，只要开发人员足够仔细和勤奋，可以从Java中得到更多。David LaVallée使用了许多技巧来避免过量的内存分配和不必要的AWT绘画功能的调用。利用一个小的整数数组创建日光灯的掩码图像而不是使用一个下载的GIF图片节省来下载时间，增加了程序的灵活性。用paint(getGraphics( ))代替repaint( )明显地增加了帧播放速率。使用copyArea( )而不是重新绘制图形或是调用drawImage( )而带来的性能提高，其意义深远。最后，创建和使用IntHash类而不是强制系统经常的进行垃圾处理有助于提高性能。

第 31 章　Scrabblet：多玩家的猜字游戏

Scrabblet是一个多玩家，网络版，客户/服务器架构的游戏。它是本书中最复杂的小应用程序，它处理了Java编程中最难以处理的一些问题。Scrabblet包括11个类，1400多行代码。其中两个类是服务器端的小应用程序。其余的9个由Web浏览器下载，作为游戏的仿真部分。本书详细描述了游戏中的全部代码。在本章，我们将解剖每个类，向读者展示编写一个多玩家的游戏是件多么简单的事。

31.1　网络安全问题

现在大多数的网上小应用程序在下载之后就与网络无关了。其中一个原因是网络使得Java安全问题难以处理。大多数的Java小应用程序环境，例如Netscape Navigator 和Microsoft Internet Explorer，严格限制小应用程序对网络的使用权限。这种情况是由于TCP/IP在其多数的基本协议中缺乏认证机制而造成的。很多公司希望通过使用防火墙保护自己的私有数据，因此小心的管理这个Internet固有的缺陷。所谓防火墙，就是一个在Internet和私有网络之间的一台计算机。Internet与私有网络之间的所有连接都必须通过防火墙，而防火墙可以过滤或是拒绝所有进入、流出私有网络的连接和数据包。通过这种方法，一个试图访问一个内部网络端口的防火墙外的程序将被防火墙拒绝。如果没有防火墙时，系统管理员则必须审查内部网络每台机器的安全。在一个防火墙保护的网络中，只有防火墙自己需要安全措施，其余的机器都可以被认为是“友好的”，内部的机器之间不作任何防范。

这是Java可能造成安全隐患的地方。如果一个支持Java的浏览器允许小应用程序连接任何一个Internet地址，这个小应用程序就可以成为防火墙外的恶意程序的代理。一旦下载了这个小应用程序，这个小应用程序在web浏览器中自动运行，它可以连接邻近的机器和服务器。这些机器对内部机器没有任何防范，所以接收了这些连接。这样，这个小应用程序就可以自由地将敏感数据偷出，并将这些数据传送出防火墙，到达Internet中的恶意主机上。

因为存在这种情况，小应用程序只被允许连接一个主机：就是下载小应用程序的源主机。这可限制小应用程序在内部网中探听其他的消息。普林斯顿大学的研究者公布了一个非常知名的“Java安全攻击方法”，这个方法欺骗Java运行时系统，允许一个小应用程序打开一个通向被禁止机器的网络套接字。幸运的是，这个方法非常的难以重复，随后这个漏洞被堵上了。

一个多玩家的游戏都必须处理哪些安全问题呢？很多。最简单的方法是编写一个可以直接在游戏者之间通信的“点对点”网络环境中的多玩家游戏。这种方法，游戏并不依赖于任何服务器软件的运行。不幸的是，小应用程序只能和加载它的服务器相连。这意味着两个游戏者必须通过服务器才能彼此通信。

在本章中，展示一个简单服务器的源代码，这个服务器程序管理一个连接客户的列表，

并在客户之间传送消息。服务器对游戏的大部分内容并不了解。它只是愉快的在A点和B点之间传递消息。这个功能由两个类，Server和ClientConnection完成。在本章的结尾将详细描述这两个类。

31.2 游 戏

在玩家开始游戏之前，必须先选择一个对手。不需要在游戏之前先打个电话来找个朋友一起玩，这个小应用程序以其他方法解决这个问题。在它启动时，它提示用户输入他或是她的名字（见图31-1）。这个名字被传回服务器，服务器将参加者的名字广播给其他潜在的游戏者。然后这个用户可以看到一个全部可挑战的参加者列表（见图31-2），从中选择一个，并点击Challenge按钮。现在，不需要任何确认或是否认挑战的方法，挑战自动被接受。一旦完成接受挑战的工作，两个游戏者都能看到游戏画面，而其他的游戏参加者只是发现这两个游戏者的名字从可挑战的列表中消失了。

图 31-1 游戏者必须在游戏开始时键入他或她的名字

图 31-2　可选的对手名单

这是一个很简单的游戏，但很难战胜一个熟练的对手。呈现在游戏者面前的是一个15×15的正方形的网格，分配给每个游戏者一组带字母的七个方块（见图31-3）。这些方块是从100个方块的口袋中随机挑选出来的。这个方块可以用鼠标点击，拖曳到网格中的对应方块上。如果网格上的位置已经被占领了，那么这个方块会回到它的起始位置。在游戏者的一次操作中还可以调整位置，但是一次操作结束不能再移动方块了。

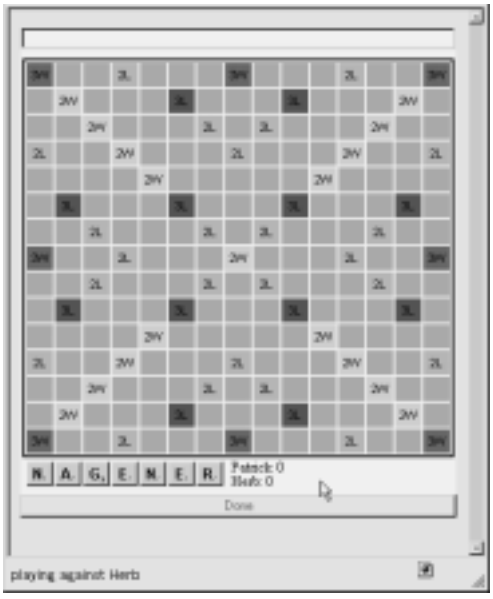


图 31-3　Patrick 和 Herb 准备开始比赛

第一个游戏者在棋盘上将方块排成一行，并组成一个英语单词。第一个单词必须覆盖中心的方块。其余的单词必须至少使用一个已经在棋盘上的方块。游戏者点击Done按钮结束自己的一次操作。如果游戏者找不到合适的单词，可以连击两次Done按钮。两个游戏者轮流拼单词直到方块用光。

棋盘如图31-3所示，在板上用简单的字母表明了分值。2L意味着加倍放置在这个格子中的方块分值。3L意味着可以三倍分值。2W意味着可以得到这个单词分值的两倍。3W意味着单词分值的三倍。如果小应用程序的的尺寸再大些，它会用更具有描述性的标记说明分值的计算方法，如图31-4所示。

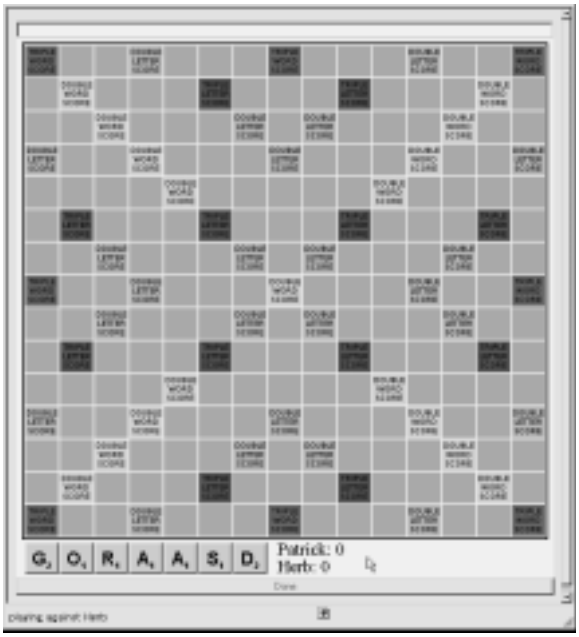


图 31-4 在尺寸较大的小应用程序中，显示更详细

31.2.1 计分方法

每个轮回之后计算分值。每个方块在字母旁边都有一个分值。这个分值可能会按方块放置的位置进行加倍或是三倍计分。如果单词中的一个方块放置在合适（颜色）的位置上，这个单词的分值可能会是两倍或是三倍。如果单词和其他与之相连的方块可以组成另外的单词，则分别计算分值。如果游戏参加者一次就使用了全部的七个单词，则有50分的鼓励。在游戏结束的时，得分高的人是赢家。

图31-5显示了一个已经经过若干回合的棋盘。Patrick以SIRE单词开始，得到8分。其中每个方块值一分，共四分，中心方块占据了一个可以双倍单词计分的位置，因此总分8分。接着，Herb拼出一个HIRE，HIRE中的I就用了SIRE中的I。这个单词共得七分，是四个方块分值的总合。注意Herb重复利用的Patrick的I是计分的，但是I方块下的双倍单词计分就失效了。在图31-5中，Patrick又拼出GREAT并按下了Done按钮完成他这一回合的操作。注意正在使用的方块的颜色比已经放置好的方块要明亮（见图31-5）。

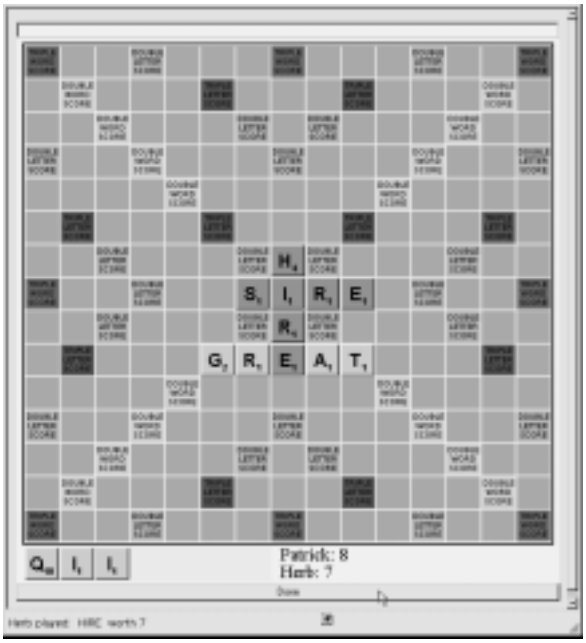


图 31-5　比赛开始时的 Scrabblet

图 31-6　Herb 正在放置 D 字母方块，完成单词 HEATED 的拼写

在游戏的过程中，游戏者可以通过在小应用程序顶部的文本输入区域进行文字交谈（见图31-7）。这些消息在一定时间后出现在另一个游戏者者浏览器的状态栏中，状态栏通常在浏览器的底部（见图31-8）。

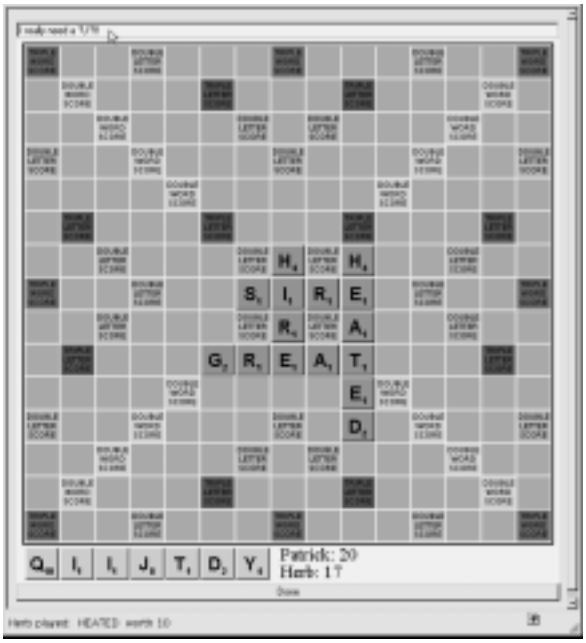


图 31-7 Patrick 正在抱怨被 Q 粘住了，但却没有 U

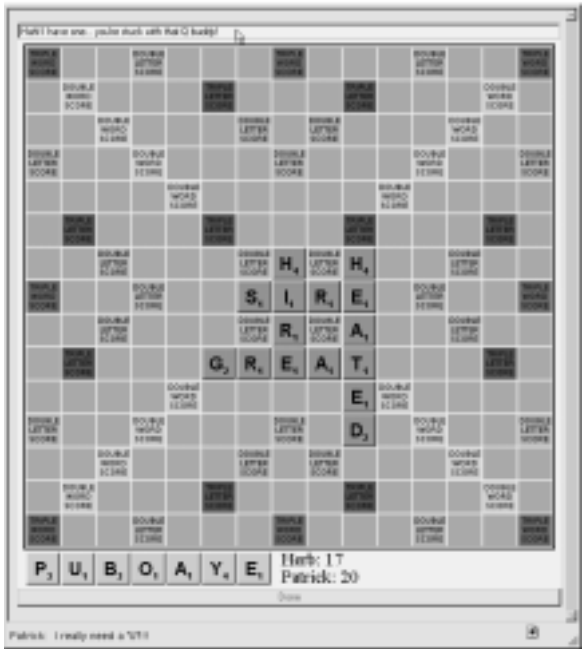


图 31-8 Herb 回答。注意屏幕底部 Patrick 的上一条消息

在分析源代码之前再解释一下这个游戏。要想取得高分，应该在一个方向构成单词的同时在另一个方向也构成一个单词。这第二个单词往往很短小，是两个字母的单词，但是分值可以累计。在图31-9中，Patrick的DEITY共得到21分，其中单词字面9分，加倍变为18

分，加上在垂直方向构成了AD这个单词，再得三分。记住，每次拼写的单词必须是一个真实的单词。最终，这个游戏会放松对有争议单词的界限或是自动的按字典检查以解决冲突。

图 31-9　Patrick 在两个方向都得了分

31.3　源 代 码

现在已经了解了游戏的使用方法，应该是学习这个游戏的源代码的时候了。因为一些类的代码非常长，本章将解释插入源代码中，而不是到最后才列出源码。

31.3.1　APPLET标记

这个游戏的APPLET标记很简单。只有主类的名字和小应用程序的尺寸。Scrabblet没有任何的<param>标记。注意，小应用程序的尺寸越大，棋盘就越好看。这个小应用程序的长宽比例应该是高度略大于宽度。

```
<applet code=Scrabblet.class width=400 height=450>
</applet>
```

31.3.2　Scrabblet.java

主要的applet类包含在Scrabblet.java文件中。虽然大部分的游戏规则都在本章后面部分会描述的Board类中，这个文件仍有大约300行，是个比较复杂的小应用程序类。

Scrabblet.java文件的开头是普通的输入语句，几乎加载了每一个标准Java包。然后，声明Scrabblet是实现ActionListener的Applet子类。

```
import java.io.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class Scrabblet extends Applet implements ActionListener {
```

接着，声明一个很大的实例变量集合。server是与运行游戏服务器的Web服务器的连接。机器的名字是ServerName，bag是游戏中共用的字母袋。两个游戏参与者有各自的装字母块的口袋，这两个口袋被初始化为相同的随机序列以此保持同步。board是棋盘的副本。参加游戏的两个人各有一个棋盘，在每一个回合后，由游戏保持这两个棋盘的同步。

如果不能访问网络服务器，则设置single标志，游戏可以在单用户模式下进行。在轮到这一方操作时，布尔变量ourturn为true。如果游戏者找不到合适的单词，可以不在棋盘上放置任何字母方块的情况下连续按下两次Done按钮。用seen_pass变量标记是否已经按下过Done按钮。

为了管理与远程对手的棋盘间的同步，在theirs中保存了已选择的方块拷贝。主要用theirs检查对手是否有作弊，因此小应用程序并不显示theirs的内容。两个字符串，name 和others_name，分别保持自己和对手的名字。

```
private ServerConnection server;
private String serverName;
private Bag bag;
private Board board;
private boolean single = false;
private boolean ourturn;
private boolean seen_pass = false;
private Letter theirs[] = new Letter[7];
private String name;
private String others_name;
```

接着，声明8个变量，用于用户界面管理。这些都是AWT的组件，由小应用程序操作控制。topPanel保存prompt ，而namefield保存在启动时得到的用户名。done按钮用来指示回合的结束。chat TextField用于输入对话的消息。idList显示可选的对手。用challenge按钮将自己和对手连接在一起。ican Canvas保存起始阶段显示的游戏名和版权声明。

```
private Panel topPanel;
private Label prompt;
private TextField namefield;
private Button done;
private TextField chat;

private List idList;
private Button challenge;
private Canvas ican;
```

init( )

init( )方法只被调用一次，这个方法建立BorderLayout，找出是从哪个Internet主机上下

载的这个小应用程序，然后创建一个splash屏幕画布。

```
public void init() {
  setLayout(new BorderLayout());
  serverName = getCodeBase().getHost();
  if (serverName.equals(""))
    serverName = "localhost";
  ican = new IntroCanvas();
}
```

start( )

在浏览器重新显示小应用程序所在的页面时调用start( )方法。在方法的开始部分的try块用来捕获网络连接错误。如果能够建立一个新的ServerConnection，则意味着以前没有运行过start( )方法，所以创建提示用户输入姓名的屏幕。在此时，经the splash screen, ican，放在窗口的中央。如果name非空，则意味着用户离开过这个页面，现在是重新返回到这个页面的。游戏假定已经获得用户的名字，可以跳过nameEntered( )。当用户在名字输入区域敲击返回时，调用这个nameEntered( )方法。在结尾处的validate( ) 方法确保所有AWT组件都被正确更新。

如果产生异常，则认为网络连接失败，进入单用户模式。调用start_game( )启动游戏。

```
public void start() {
  try {
    showStatus("Connecting to " + serverName);
    server = new ServerConnection(this, serverName);
    server.start();
    showStatus("Connected: " + serverName);

    if (name == null) {
      prompt = new Label("Enter your name here:");
      namefield = new TextField(20);
      namefield.addActionListener(this);
      topPanel = new Panel();
      topPanel.setBackground(new Color(255, 255, 200));
      topPanel.add(prompt);
      topPanel.add(namefield);
      add("North", topPanel);
      add("Center", ican);
    } else {
      if (chat != null) {
        remove(chat);
        remove(board);
        remove(done);
      }
      nameEntered(name);
    }
    validate();
  } catch (Exception e) {
    single = true;
    start_Game((int)(0x7fffffff * Math.random()));
  }
}
```

stop( )

在用户离开小应用程序所在页面时调用stop( )方法。这里，程序仅仅通知服务器用户已经离开。如果在用户重新返回页面，则在start( )方法中重新创建网络连接。

```
public void stop() {
 if (!single)
   server.quit();
}
```

add( )

在新用户进入游戏时调用add( )方法。将用户名加入List对象。特别注意add( )方法的字符串格式，在后面将使用这个字符串从选手列表中抽取ID。

```
void add(String id, String hostname, String name) {
  delete(id); // in case it is already there.
  idList.add("(" + id + ")  " + name + "@" + hostname);
  showStatus("Choose a player from the list");
}
```

delete( )

在用户不想将自己标志为可选对手时调用delete( )方法。当用户退出或是选好对手准备开始游戏时调用这个方法。在这个方法里，通过提取圆括号中的值在列表中逐个的寻找id字符串。如果列表中没有名字（而且并没有开始游戏：bag == null），则小应用程序显示一个特殊的消息通知用户挂起，直至找到对手。

```
void delete(String id) {
  for (int i = 0; i < idList.getItemCount(); i++) {
    String s = idList.getItem(i);
    s = s.substring(s.indexOf("(") + 1, s.indexOf(")"));
    if (s.equals(id)) {
      idList.remove(i);
      break;
    }
  }
  if (idList.getItemCount() == 0 && bag == null)
    showStatus("Wait for other players to arrive.");
}
```

getName( )

getName( )方法和delete( )很类似，除了它只是简单的提取名字然后返回。如果没有找到id，则返回null。

```
private String getName(String id) {
  for (int i = 0; i < idList.getItemCount(); i++) {
    String s = idList.getItem(i);
    String id1 = s.substring(s.indexOf("(") + 1, s.indexOf(")"));
    if (id1.equals(id)) {
      return s.substring(s.indexOf(" ") + 3, s.indexOf("@"));
    }
```

```
    }
    return null;
  }
```

challenge( )

在另一个用户向本地用户挑战时由ServerConnection调用challenge( )方法。本来可以将这个方法实现得更为复杂，允许提示用户接受或是拒绝挑战，但是这个方法实现为自动的接受挑战。注意，用来启动游戏的随机数初值，被传送到对手的accept( )方法中。这样，两方初始化了相同随机状态的两个字母方块袋以确保游戏的同步性。调用server.delete( )确保不再接受其他游戏者的挑战。同时注意，本地游戏者通过设置ourturn 为false放弃了先手。

```
  // we've been challenged to a game by "id".
  void challenge(String id) {
    ourturn = false;
    int seed = (int)(0x7fffffff * Math.random());
    others_name = getName(id);   // who was it?
    showStatus("challenged by " + others_name);

    // put some confirmation here...

    server.accept(id, seed);
    server.delete();
    start_Game(seed);
  }
```

accept( )

accept( )是远方用户回应刚才提到的server.accept( )调用的方法。与对手一样，必须调用server.delete( )将自己从可选的游戏者列表中删除。通过将ourturn设为true这个用户首先开始拼写。

```
  // our challenge was accepted.
  void accept(String id, int seed) {
    ourturn = true;
    others_name = getName(id);
    server.delete();
    start_Game(seed);
  }
```

chat( )

当对手在他或是她的对话窗口中输入时，服务器调用chat( )方法。在这个方法的实现中，只是简单的在浏览器的状态栏中显示对话消息。在将来的实现版本中，应改进为将消息记录在一个文本域内。

```
  void chat(String id, String s) {
    showStatus(others_name + ": " + s);
  }
```

move( )

对手每放置一个方块调用一次move( )方法。这个方法搜寻theirs找到使用的字母。如果那个格子已经被占用，方块被返回到原来放置的位置。否则，对手的字母永久移入棋盘上。接着，使用bag.takeOut( )在theirs中重新放置方块。如果bag空了，则显示一个状态消息。重画棋盘显示新的方块。注意此时还没有基于方块位置的记分。直到调用turn( )，小应用程序才计算总的分数。

```
// the other guy moved, and placed 'letter' at (x, y).
void move(String letter, int x, int y) {
  for (int i = 0; i < 7; i++) {
    if (theirs[i] != null && theirs[i].getSymbol().equals(letter)) {
      Letter already = board.getLetter(x, y);
      if (already != null) {
        board.moveLetter(already, 15, 15); // on the tray.
      }
      board.moveLetter(theirs[i], x, y);
      board.commitLetter(theirs[i]);
      theirs[i] = bag.takeOut();
      if (theirs[i] == null)
        showStatus("No more letters");
      break;
    }
  }
  board.repaint();
}
```

turn( )

在对手的方块全部移动完毕之后调用turn( )。远程的Scrabblet实例计算分数，并发送到本地，因此本地只需拷贝而无需再计算一遍。分数显示在状态栏中，setEnabled方法允许本地开始拼写。othersTurn( )将分数通知棋盘。棋盘此时显示新的分数。

```
void turn(int score, String words) {
  showStatus(others_name + " played: " + words + " worth " +
           score);
  done.setEnabled(true);
  board.othersTurn(score);
}
```

quit( )

当另一方明确退出时，调用quit( )方法。这个方法删除游戏的AWT组件，跳转回接下来描述的nameEntered( )，重新加入游戏者列表。

```
void quit(String id) {
  showStatus(others_name + " just quit.");
  remove(chat);
  remove(board);
  remove(done);
  nameEntered(name);
}
```

nameEntered( )

当提示输入用户名后按下回车时actionPerformed( )调用nameEntered( )。在此时先删除存在的任何AWT组件，然后创建新的List对象，idList，用以存储对手的名字。这个方法还在页面顶部增加了一个名为challenge的按钮，然后利用setName( )通知服务器，用户已经准备好了。

```
private void nameEntered(String s) {
  if (s.equals(""))
    return;
  name = s;
  if (ican != null)
    remove(ican);
  if (idList != null)
    remove(idList);
  if (challenge != null)
    remove(challenge);
  idList = new List(10, false);
  add("Center", idList);
  challenge = new Button("Challenge");
  challenge.addActionListener(this);
  add("North", challenge);
  validate();
  server.setName(name);
  showStatus("Wait for other players to arrive.");
  if (topPanel != null)
    remove(topPanel);
}
```

wepick( )和 theypick( )

简单的调用wepick( ) 和theypick( )方法开始游戏，这两个方法分别给每个游戏者分配七个方块。特别注意挑战的双方必须按正确的顺序调用过程，这个顺序取决于谁先开始。调用bag.takeOut( )可以在共享的口袋里取出单个字母。调用board.addLetter( )将方块放在盘子里，在另一方，theypick( )将字母保持在theirs中。

```
private void wepick() {
  for (int i = 0; i < 7; i++) {
    Letter l = bag.takeOut();
    board.addLetter(l);
  }
}

private void theypick() {
  for (int i = 0; i < 7; i++) {
    Letter l = bag.takeOut();
    theirs[i] = l;
  }
}
```

start_Game( )

在单用户模式中，start_Game( )在帧窗口中弹出splash图形。然后创建一个棋盘，在棋盘的构造函数中没有任何参数，指示是单用户模式。

在一对一的模式中，删除可选对手列表组件，然后在小应用程序中增加chat窗口，然后增加棋盘和一个Done按钮。接着，创建一个口袋，如果ourturn为真，则先调用wepick( )，然后调用theypick( )。在本用户不是先手的情况下，禁止棋盘和Done按钮，然后先调用theypick( )后调用wepick( )。最后强制重画棋盘。

```
private void start_Game(int seed) {
  if (single) {
    Frame popup = new Frame("Scrabblet");
    popup.setSize(400, 300);
    popup.add("Center", ican);
    popup.setResizable(false);
    popup.show();
    board = new Board();
    showStatus("no server found, playing solo");
    ourturn = true;
  } else {
    remove(idList);
    remove(challenge);
    board = new Board(name, others_name);
    chat = new TextField();
    chat.addActionListener(this);
    add("North", chat);
    showStatus("playing against " + others_name);
  }

  add("Center", board);
  done = new Button("Done");
  done.addActionListener(this);
  add("South", done);
  validate();

  bag = new Bag(seed);
  if (ourturn) {
    wepick();
    if (!single)
      theypick();
  } else {
    done.setEnabled(false);
    theypick();
    wepick();
  }
  board.repaint();
}
```

challenge_them( )

在按下challenge按钮时调用challenge_them( )方法。这个方法在idList中找到选中的对手，向他或她发送一个challenge( )消息。删除列表和按钮，准备开始游戏。

```
private void challenge_them() {
  String s = idList.getSelectedItem();
  if (s == null) {
    showStatus("Choose a player from the list then press Challenge");
  } else {
    remove(challenge);
    remove(idList);
    String destid = s.substring(s.indexOf('(')+1,
                          s.indexOf(')'));
    showStatus("challenging: " + destid);
    server.challenge(destid);  // accept will get called if
                          // they accept.
    validate();
  }
}
```

our_turn( )

在按下Done钮时，调用our_turn( )。首先它调用board.findwords( )检查方块是否放在合法的位置上，将结果存入word。如果word为空，则这个方块有问题，方法在状态栏中显示这个情况。如果word为“”，意味着本回合没有放置方块。在单用户模式，这可以忽略，在对抗模式中，如果接连两次Done按钮而没有放置方块，则将放弃本次机会轮到对手操作。

如果方块放置在合法位置，则本次回合结束，ourturn( )将字母提交到棋盘上。注意commit( )以server为参数。它用这种方法通知远程用户每个新字母的位置。然后方法重新放置使用过的字符。在多用户模式下，禁止己方的棋盘，调用server.turn( )通知另一方轮到他们行动了。

```
private void our_turn() {
  String word = board.findwords();
  if (word == null) {
    showStatus("Illegal letter positions");
  } else {
    if ("".equals(word)) {
      if (single)
        return;
      if (seen_pass) {
        done.setEnabled(false);
        server.turn("pass", 0);
        showStatus("You passed");
        seen_pass = false;
      } else {
        showStatus("Press done again to pass");
        seen_pass = true;
        return;
      }
    } else {
      seen_pass = false;
    }
    showStatus(word);
    board.commit(server);
    for (int i = 0; i < 7; i++) {
      if (board.getTray(i) == null) {
```

```
            Letter l = bag.takeOut();
            if (l == null)
              showStatus("No more letters");
            else
              board.addLetter(l);
          }
        }
        if (!single) {
          done.setEnabled(false);
          server.turn(word, board.getTurnScore());
        }
        board.repaint();
      }
    }
```

actionPerformed( )

使用actionPerformed( )方法获取小应用程序使用的各个组件的输入。它处理Challenge和Done按钮，名字输入框和对话输入框。

```
  public void actionPerformed(ActionEvent ae) {
    Object source = ae.getSource();
    if(source == chat) {
      server.chat(chat.getText());
      chat.setText("");
    }
    else if(source == challenge) {
      challenge_them();
    }
    else if(source == done) {
      our_turn();
    }
    else if(source == namefield) {
      TextComponent tc = (TextComponent)source;
      nameEntered(tc.getText());
    }
  }
}
```

31.3.3 IntroCanvas.java

Canvas的IntroCanvas子类非常简单。它仅重载paint( )以显示小应用程序的名字和一个简短的版权专用明。它创建一些定制的颜色和字体。为了清楚起见，显示字符串保存在静态变量中。

```
import java.awt.*;
import java.awt.event.*;

class IntroCanvas extends Canvas {
  private Color pink = new Color(255, 200, 200);
  private Color blue = new Color(150, 200, 255);
  private Color yellow = new Color(250, 220, 100);

  private int w, h;
```

```
private int edge = 16;
private static final String title = "Scrabblet";
private static final String name =
       "Copyright 1999 - Patrick Naughton";
private static final String book =
       "Chapter 31 from 'Java: The Complete Reference'";
private Font namefont, titlefont, bookfont;

IntroCanvas() {
  setBackground(yellow);
  titlefont = new Font("SansSerif", Font.BOLD, 58);
  namefont = new Font("SansSerif", Font.BOLD, 18);
  bookfont = new Font("SansSerif", Font.PLAIN, 12);
  addMouseListener(new MyMouseAdapter());
}
```

d( )

私有方法d( )是以按选择的等量偏移绘制居中文字。先向左偏移1绘制一个白色的字符串，然后向右偏移1绘制一个黑色的字符串，然后至少用粉红色不偏移的再画一次字符串，以此完成主标题的高亮度/阴影效果。

```
private void d(Graphics g, String s, Color c, Font f, int y,
               int off) {
  g.setFont(f);
  FontMetrics fm = g.getFontMetrics();
  g.setColor(c);
  g.drawString(s, (w - fm.stringWidth(s)) / 2 + off, y + off);
}

public void paint(Graphics g) {
  Dimension d = getSize();
  w = d.width;
  h = d.height;
  g.setColor(blue);
  g.fill3DRect(edge, edge, w - 2 * edge, h - 2 * edge, true);
  d(g, title, Color.black, titlefont, h / 2, 1);
  d(g, title, Color.white, titlefont, h / 2, -1);
  d(g, title, pink, titlefont, h / 2, 0);
  d(g, name, Color.black, namefont, h * 3 / 4, 0);
  d(g, book, Color.black, bookfont, h * 7 / 8, 0);
}
```

mousePressed( )

在下列的代码段中，注意MyMouseAdapter是扩展MouseAdapter的一个内部类。如果它被点击，它将重载mousePressed( )方法，从而引起画布父类调用hide( )方法。这个方法只在单用户模式有用，取消弹出的帧。

```
class MyMouseAdapter extends MouseAdapter {
  public void mousePressed(MouseEvent me) {
    ((Frame)getParent()).setVisible(false);
  }
}
```

```
}
```

31.3.4 Board.java

Board类封装了大部分的游戏规则和棋盘的外观。这是这个游戏中最大的类，有大约500行的代码。一些私有变量存储游戏的状态。一个名为board的15×15 的Letters数组存储棋盘上每个格子上的方块。tray数组保存当前棋盘上的Letters。回想一下，Scrabblet 小应用程序类保存着对手的7个Letters。Point对象orig 和here用来记住字母的位置。用name和others_name变量简单地显示计分牌上的名字。在单用户模式，这两个变量都为空。两个用户的得分存放在total_score 和others_score中，上一个回合的分数放在turn_score中。这个类有两个构造函数，一个构造函数创建用户的名字，否则在单用户模式中为空。

```
import java.awt.*;
import java.awt.event.*;

class Board extends Canvas {
  private Letter board[][] = new Letter[15][15];
  private Letter tray[] = new Letter[7];
  private Point orig = new Point(0,0);
  private Point here = new Point(0,0);
  private String name;
  private int total_score = 0;
  private int turn_score = 0;
  private int others_score = 0;
  private String others_name = null;

  Board(String our_name, String other_name) {
    name = our_name;
    others_name = other_name;
    addMouseListener(new MyMouseAdapter());
    addMouseMotionListener(new MyMouseMotionAdapter());
  }

  Board() {
    addMouseListener(new MyMouseAdapter());
    addMouseMotionListener(new MyMouseMotionAdapter());
  }
```

othersTurn( ), getTurnScore( ) 和 getTray( )

用这三个方法控制对几个私有变量的访问。首先，在其他游戏者结束回合时小应用程序调用othersTurn( )。这个方法增加游戏者的分数，重画棋盘反映分数的变换。在计分牌重画需要分数值时，getTurnScore( )方法返回存储的上个回合的分数。小应用程序使用这个方法将分数传递给对手，对手最后会在远程机器上调用othersTurn( )。getTray( )方法提供一个对私有tray数组的只读访问。

```
  void othersTurn(int score) {
    others_score += score;
    paintScore();
    repaint();
  }
```

```
int getTurnScore() {
  paintScore();
  return turn_score;
}

Letter getTray(int i) {
  return tray[i];
}
```

addLetter( )

使用addLetter( )方法在游戏者的盘子里放置字母。字母被放在第一个空格子里，如果这个方法找不到空格子，则返回false。

```
synchronized boolean addLetter(Letter l) {
  for (int i = 0; i < 7; i++) {
    if (tray[i] == null) {
      tray[i] = l;
      moveLetter(l, i, 15);
      return true;
    }
  }
  return false;
}
```

existingLetterAt( )

使用私有方法existingLetterAt( )检查棋盘上是否有不是本回合的字母。随后的findwords( )方法使用这个方法保证本回合拼写的单词中至少有一个字母是已经存在的字母。

```
private boolean existingLetterAt(int x, int y) {
  Letter l = null;
  return (x >= 0 && x <= 14 && y >= 0 && y <= 14
    && (l = board[y][x]) != null && l.recall() == null);
}
```

findwords( )

findwords( )是一个非常大的方法，检查棋盘的每一回合的状态。如果违反了字母放置规则，则返回null。如果本回合没有拼写任何单词，则返回“”。如果本回合的所有字母放置都合法，则返回这些字母组成的单词字符串，每个单词用空格隔开。更新turn_score 和total_score变量的实例，反映刚刚拼写出的单词的分值。

首先，findwords( )计算字母的分值ntiles，将其存放在一个名为atplay的数组中。接着，它检查头两个字母（如果本次拼出的单词多于一个字母）确定这个单词是垂直的或是水平的。然后，检查本回合的所以其他的字母，要确定它们在同一行。如果任何一个字母超出行或列，方法返回null。

```
synchronized String findwords() {
  String res = "";
```

```
turn_score = 0;

int ntiles = 0;
Letter atplay[] = new Letter[7];
for (int i = 0; i < 7; i++) {
  if (tray[i] != null && tray[i].recall() != null) {
    atplay[ntiles++] = tray[i];
  }
}
if (ntiles == 0)
  return res;

boolean horizontal = true; // if there's one tile,
                       // call it horizontal
boolean vertical = false;
if (ntiles > 1) {
  int x = atplay[0].x;
  int y = atplay[0].y;
  horizontal = atplay[1].y == y;
  vertical = atplay[1].x == x;
  if (!horizontal && !vertical) // diagonal...
    return null;
  for (int i = 2; i < ntiles; i++) {
    if (horizontal && atplay[i].y != y
      || vertical && atplay[i].x != x)
      return null;
  }
}
```

接着，这个方法查看每个字母以确保至少有一个字母是使用了已经存在的字母。一个特殊的情况是在游戏的开头，如果覆盖了中心的位置，且使用了多个方块，则这个回合合法。

```
// make sure that at least one played tile is
// touching at least one existing tile.
boolean attached = false;
for (int i = 0; i < ntiles; i++) {
  Point p = atplay[i].recall();
  int x = p.x;
  int y = p.y;
  if ((x == 7 && y == 7 && ntiles > 1) ||
     existingLetterAt(x-1, y) || existingLetterAt(x+1, y) ||
     existingLetterAt(x, y-1) || existingLetterAt(x, y+1)) {
    attached = true;
    break;
  }
}
if (!attached) {
  return null;
}
```

下一个循环遍历检查整个单词的每个字母，(i == –1)，然后检查每个字母(i == 0..ntiles)是否可能在与主方向垂直的另一个方向上构成新的单词，方向由horizontal管理。

```
// we use -1 to mean check the major direction first
// then 0..ntiles checks for words orthogonal to it.
for (int i = -1; i < ntiles; i++) {
  Point p = atplay[i==-1?0:i].recall(); // where is it?
  int x = p.x;
  int y = p.y;

  int xinc, yinc;
  if (horizontal) {
    xinc = 1;
    yinc = 0;
  } else {
    xinc = 0;
    yinc = 1;
  }
  int mult = 1;

  String word = "";
  int word_score = 0;
```

然后该方法选出每个字符，向左或向上移动找到每个单词的第一个字母。一旦找到单词的开头，方法向右或是向下移动，检查单词中的每个字母。在letters_seen中计算字母的分值。每个字母的分值由其下的增倍因子决定。仅在第一次使用这个格子时才应用增倍因子，否则按字母的分值计算。这些分数累计在word_score中。

```
  // here we back up to the top/left-most letter
  while (x >= xinc && y >= yinc &&
        board[y-yinc][x-xinc] != null) {
    x -= xinc;
    y -= yinc;
  }

  int n = 0;
  int letters_seen = 0; // letters we've just played.
  Letter l;
  while (x < 15 && y < 15 && (l = board[y][x]) != null) {
    word += l.getSymbol();
    int lscore = l.getPoints();
    if (l.recall() != null) {  // one we just played...
      Color t = tiles[y < 8 ? y : 14 - y][x < 8 ? x : 14 - x];
      if (t == w3)
        mult *= 3;
      else if (t == w2)
        mult *= 2;
      else if (t == l3)
        lscore *= 3;
      else if (t == l2)
        lscore *= 2;
      if (i == -1) {
        letters_seen++;
      }
    }
    word_score += lscore;
    n++;
```

```
        x += xinc;
        y += yinc;
      }
      word_score *= mult;
```

只对主要单词进行最后的错误检查。在遇到空白格子或是棋盘的边缘时循环结束，它应该覆盖所有的刚放置的字母方块以及以前放置的方块。如果它少检查了字母，这意味这字母之间可能有空格，是个非法位置，则返回空，如果通过检查，方法同时检查是否同时使用了7个方块，同时使用7个方块可以得到50分的奖励。在检查这个单词之后，findwords( )转换horizontal的方向，开始检查垂直方向的单词。

```
    if (i == -1) {     // first pass...

      // if we didn't see all the letters, then there was a gap,
      // which is an illegal tile position.
      if (letters_seen != ntiles) {
        return null;
      }

      if (ntiles == 7) {
        turn_score += 50;
      }

      // after the first pass, switch to looking the other way.
      horizontal = !horizontal;
    }
```

在findwords( )遍历整个单词时，必须确保只有至少由两个字母构成的单词才能计分。在这种情况下，将word_score加到turn_score上，将这个单词添加到结果字符串中。一旦遍历全部字母，统计总分数并返回。

```
      if (n < 2)  // don't count single letters twice.
        continue;

      turn_score += word_score;
      res += word + " ";
    }
    total_score += turn_score;
    return res;
  }
```

commit( ) 和 commitLetter( )

commit( )和commitLetter( )方法提交已经暂时放在棋盘上的字母。从游戏者的盘子中删除这些字母，在板上用深颜色画出。在字母提交后，commit( )调用move( )通知服务器每个字母的位置，根据这个消息更新对手的棋盘。

```
  synchronized void commit(ServerConnection s) {
    for (int i = 0 ; i < 7 ; i++) {
      Point p;
      if (tray[i] != null && (p = tray[i].recall()) != null) {
        if (s != null)  // there's a server connection
```

```
        s.move(tray[i].getSymbol(), p.x, p.y);
        commitLetter(tray[i]);  // marks this as not in play.
        tray[i] = null;
      }
    }
  }

  void commitLetter(Letter l) {
    if (l != null && l.recall() != null) {
      l.paint(offGraphics, Letter.DIM);
      l.remember(null);   // marks this as not in play.
    }
  }
```

update( ) 和 paint( )

在这里声明了多个私有变量提供访问棋盘的各个位置。这段代码同时定义了两个屏幕外缓冲区，一个用于缓存棋盘图像和所有的临时放置的字母方块，另一个是显示图像的备份。update( )方法调用paint( )以避免闪烁。paint( )方法快速调用checksize( )确保所有缓冲区都已创建，然后通过pick != null检查用户是否正在拖曳字母。如果正在拖曳字母，则paint( )拷贝屏幕外图像内容，画出正在拖曳的字母的外形，x0, y0, w0, h0。接着在屏幕图像内容上剪切同样的矩形。这种方法将每次移动鼠标时不得不移动的像素降为最少。

为了绘制屏幕，拷贝背景图像offscreen，然后按正常设置（NORMAL）绘制游戏者盘子里的字母块。正在拖曳的字母块按BRIGHT模式绘制。最后，拷贝备份缓冲区图像offscreen2到屏幕。

```
  private Letter pick;  // the letter being dragged around.
  private int dx, dy;   // offset to topleft corner of pick.
  private int lw, lh;   // letter width and height.
  private int tm, lm;   // top and left margin.
  private int lt;      // line thickness (between tiles).
  private int aw, ah;   // letter area size.

  private Dimension offscreensize;
  private Image offscreen;
  private Graphics offGraphics;
  private Image offscreen2;
  private Graphics offGraphics2;

  public void update(Graphics g) {
    paint(g);
  }

  public synchronized void paint(Graphics g) {
    Dimension d = checksize();
    Graphics gc = offGraphics2;
    if (pick != null) {
      gc = gc.create();
      gc.clipRect(x0, y0, w0, h0);
      g.clipRect(x0, y0, w0, h0);
    }
    gc.drawImage(offscreen, 0, 0, null);
```

```
    for (int i = 0 ; i < 7 ; i++) {
      Letter l = tray[i];
      if (l != null && l != pick)
        l.paint(gc, Letter.NORMAL);
    }
    if (pick != null)
      pick.paint(gc, Letter.BRIGHT);

    g.drawImage(offscreen2, 0, 0, null);
  }
```

LetterHit( )

LetterHit( )返回在x,y点的字母，如果那里没有字母则返回空。

```
  Letter LetterHit(int x, int y) {
    for (int i = 0; i < 7; i++) {
      if (tray[i] != null && tray[i].hit(x, y)) {
        return tray[i];
      }
    }
    return null;
  }
```

unplay( )

这个简单方法删除在棋盘上放置但没有提交的字母。

```
  private void unplay(Letter let) {
    Point p = let.recall();
    if (p != null) {
      board[p.y][p.x] = null;
      let.remember(null);
    }
  }
```

moveToTray( )

moveToTray( )方法简单的计算游戏者盘子中字母的屏幕位置。

```
  private void moveToTray(Letter l, int i) {
    int x = lm + (lw + lt) * i;
    int y = tm + ah - 2 * lt;
    l.move(x, y);
  }
```

dropOnTray( )

在向盘子里放方块或是将方块脱离棋盘时使用dropOnTray( )方法。这允许改变盘子中的方块顺序或是简单的从棋盘中返回字母方块。

```
  private void dropOnTray(Letter l, int x) {
    unplay(l); // unhook where we were.

    // find out what slot this letter WAS in.
```

```
    int oldx = 0;
    for (int i = 0 ; i < 7 ; i++) {
      if (tray[i] == l) {
        oldx = i;
        break;
      }
    }

    // if the slot we dropped on was empty,
    // find the rightmost occupied slot.
    if (tray[x] == null) {
      for (int i = 6 ; i >= 0 ; i--) {
        if (tray[i] != null) {
          x = i;
          break;
        }
      }
    }
    // if the slot we dropped on was from a tile already
    // played on the board, just swap slots with it.
    if (tray[x].recall() != null) {
      tray[oldx] = tray[x];
    } else {
      // we are just rearranging a tile already on the tray.
      if (oldx < x) {   // shuffle left.
        for (int i = oldx ; i < x ; i++) {
          tray[i] = tray[i+1];
          if (tray[i].recall() == null)
            moveToTray(tray[i], i);
        }
      } else {         // shuffle right.
       for (int i = oldx ; i > x ; i--) {
         tray[i] = tray[i-1];
         if (tray[i].recall() == null)
           moveToTray(tray[i], i);
        }
      }
    }
    tray[x] = l;
    moveToTray(l, x);
  }
```

getLetter( )

getLetter( )是一个简单的针对游戏板数组的只读包装方法。

```
  Letter getLetter(int x, int y) {
    return board[y][x];
  }
```

moveLetter( )

moveLetter( )方法处理将字母方块移到棋盘的指定位置或是将其放回游戏者盘子的情况。如果x,y点超出棋盘范围，则使用游戏者的盘子。当字母方块被移入棋盘，则必须是一

个空白的格子，否则字母方块被送回原地，原地的坐标由orig指定。

```
void moveLetter(Letter l, int x, int y) {
  if (y > 14 || x > 14 || y < 0 || x < 0) {
    // if we are off the board.
    if (x > 6)
      x = 6;
    if (x < 0)
      x = 0;
    dropOnTray(l, x);
  } else {
    if (board[y][x] != null) {
      x = orig.x;
      y = orig.y;
    } else {
      here.x = x;
      here.y = y;
      unplay(l);
      board[y][x] = l;
      l.remember(here);

      // turn it back into pixels
      x = lm + (lw + lt) * x;
      y = tm + (lh + lt) * y;
    }
    l.move(x, y);
  }
}
```

checksize( )

这个方法有个会误导人的名字。除了验证小应用程序的大小外，checksize( )方法作了很多工作，在确认小应用程序尺寸时，这个方法做一次性的初始化。这个方法包括主游戏模式的绘制编码。它绘制所有的格子，包括颜色和计分区域的文字。

```
private Color bg = new Color(175, 185, 175);
private Color w3 = new Color(255, 50, 100);
private Color w2 = new Color(255, 200, 200);
private Color l3 = new Color(75, 75, 255);
private Color l2 = new Color(150, 200, 255);
private Color tiles[][] = {
  {w3, bg, bg, l2, bg, bg, bg, w3},
  {bg, w2, bg, bg, bg, l3, bg, bg},
  {bg, bg, w2, bg, bg, bg, l2, bg},
  {l2, bg, bg, w2, bg, bg, bg, l2},
  {bg, bg, bg, bg, w2, bg, bg, bg},
  {bg, l3, bg, bg, bg, l3, bg, bg},
  {bg, bg, l2, bg, bg, bg, l2, bg},
  {w3, bg, bg, l2, bg, bg, bg, w2}
};

private Dimension checksize() {
  Dimension d = getSize();
  int w = d.width;
```

```
int h = d.height;

if (w < 1 || h < 1)
  return d;
if ((offscreen == null) ||
  (w != offscreensize.width) ||
  (h != offscreensize.height)) {
  System.out.println("updating board: " + w + " x " + h + "\r");

  offscreen = createImage(w, h);
  offscreensize = d;
  offGraphics = offscreen.getGraphics();
  offscreen2 = createImage(w, h);
  offGraphics2 = offscreen2.getGraphics();

  offGraphics.setColor(Color.white);
  offGraphics.fillRect(0,0,w,h);

  // lt is the thickness of the white lines between tiles.
  // gaps is the sum of all the whitespace.
  // lw, lh are the dimensions of the tiles.
  // aw, ah are the dimensions of the entire board
  // lm, tm are the left and top margin to center aw, ah in the applet.

  lt = 1 + w / 400;
  int gaps = lt * 20;

  lw = (w - gaps) / 15;
  lh = (h - gaps - lt * 2) / 16; // compensating for tray height;
  aw = lw * 15 + gaps;
  ah = lh * 15 + gaps;
  lm = (w - aw) / 2 + lt;
  tm = (h - ah - (lt * 2 + lh)) / 2 + lt;
  offGraphics.setColor(Color.black);
  offGraphics.fillRect(lm,tm,aw-2*lt,ah-2*lt);
  lm += lt;
  tm += lt;
  offGraphics.setColor(Color.white);
  offGraphics.fillRect(lm,tm,aw-4*lt,ah-4*lt);
  lm += lt;
  tm += lt;
  int sfh = (lh > 30) ? lh / 4 : lh / 2;
  Font font = new Font("SansSerif", Font.PLAIN, sfh);
  offGraphics.setFont(font);
  for (int j = 0, y = tm; j < 15; j++, y += lh + lt) {
    for (int i = 0, x = lm; i < 15; i++, x += lw + lt) {
      Color c = tiles[j < 8 ? j : 14 - j][i < 8 ? i : 14 - i];
      offGraphics.setColor(c);
      offGraphics.fillRect(x, y, lw, lh);
      offGraphics.setColor(Color.black);
      if (lh > 30) {
        String td = (c == w2 || c == l2) ? "DOUBLE" :
                    (c == w3 || c == l3) ? "TRIPLE" : null;
        String wl = (c == l2 || c == l3) ? "LETTER" :
                    (c == w2 || c == w3) ? "WORD" : null;
```

```
          if (td != null) {
            center(offGraphics, td, x, y + 2 + sfh, lw);
            center(offGraphics, wl, x, y + 2 * (2 + sfh), lw);
            center(offGraphics, "SCORE", x, y + 3 * (2 + sfh), lw);
          }
        } else {
          String td = (c == w2 || c == l2) ? "2" :
                      (c == w3 || c == l3) ? "3" : null;
          String wl = (c == l2 || c == l3) ? "L" :
                      (c == w2 || c == w3) ? "W" : null;
          if (td != null) {
            center(offGraphics, td + wl, x,
              y + (lh - sfh) * 4 / 10 + sfh, lw);
          }
        }
      }
    }
    Color c = new Color(255, 255, 200);
    offGraphics.setColor(c);
    offGraphics.fillRect(lm, tm + ah - 3 * lt, 7 * (lw + lt), lh +
                         2 * lt);

    Letter.resize(lw, lh);

    // if we already have some letters, place them.
    for (int i = 0; i < 7; i++) {
      if (tray[i] != null) {
        moveToTray(tray[i], i);
      }
    }
    paintScore();
  }
  return d;
}
```

center( )

checksize( )使用center( )将“Double Letter Score”文字居中显示。

```
private void center(Graphics g, String s, int x, int y, int w) {
  x += (w - g.getFontMetrics().stringWidth(s)) / 2;
  g.drawString(s, x, y);
}
```

paintScore( )

paintScore( )方法显示两个游戏者的分数或者是单用户模式下的一个用户分数。

```
private void paintScore() {
  int x = lm + (lw + lt) * 7 + lm;
  int y = tm + ah - 3 * lt;
  int h = lh + 2 * lt;
  Font font = new Font("TimesRoman", Font.PLAIN, h/2);
  offGraphics.setFont(font);
  FontMetrics fm = offGraphics.getFontMetrics();
```

```
    offGraphics.setColor(Color.white);
    offGraphics.fillRect(x, y, aw, h);
    offGraphics.setColor(Color.black);
    if (others_name == null) {
      int y0 = (h - fm.getHeight()) / 2 + fm.getAscent();
      offGraphics.drawString("Score: " + total_score, x, y + y0);
    } else {
      h/=2;
      int y0 = (h - fm.getHeight()) / 2 + fm.getAscent();
      offGraphics.drawString(name + ": " + total_score, x, y + y0);
      offGraphics.drawString(others_name + ": " + others_score,
                          x, y + h + y0);
    }
  }

  private int x0, y0, w0, h0;
```

selectLetter( )

selectLetter( )方法检查鼠标位置，看是否在字母上。如果在，则将其存放在pick中，并计算鼠标和左上角的字母的距离，这个距离存放在dx，dy中，同时在orig中保存字母的原始位置。

```
  private void selectLetter(int x, int y) {
    pick = LetterHit(x, y);
    if(pick != null) {
      dx = pick.x - x;
      dy = pick.y - y;
      orig.x = pick.x;
      orig.y = pick.y;
    }
    repaint();
  }
```

dropLetter( )

在dropLetter( )方法中，用户放下正在移动的字母。这个方法决定字母放下时占据的棋盘的格子。调用moveLetter( )将字母移进对应的格子。

```
  private void dropLetter(int x, int y) {
    if(pick != null) {
      // find the center of the tile
      x += dx + lw / 2;
      y += dy + lh / 2;
      // find the tile index
      x = (x - lm) / (lw + lt);
      y = (y - tm) / (lh + lt);

      moveLetter(pick, x, y);

      pick = null;
      repaint();
    }
```

```
  }
```

dragLetter( )

dragLetter( )方法与其他的鼠标相关事件的处理方式不同。这主要与性能有关。目标是尽可能的平滑与用户之间的交互作用。dragLetter( )计算在字符拖动前与当前位置的区域长宽。然后直接调用paint(getGraphics( ))。这是非标准的Java 小应用程序编程方式，但是性能可靠。

```
private void dragLetter(int x, int y) {
  if (pick != null) {
    int ox = pick.x;
    int oy = pick.y;
    pick.move(x + dx, y + dy);
    x0 = Math.min(ox, pick.x);
    y0 = Math.min(oy, pick.y);
    w0 = pick.w + Math.abs(ox - pick.x);
    h0 = pick.h + Math.abs(oy - pick.y);
    paint(getGraphics());
  }
}
```

mousePressed( )

在下面的代码段中，注意MyMouseAdapter是扩展MouseAdapter的内部类。它重载了mousePressed( ) 和mouseReleased( )方法。

mousePressed( )方法调用selectLetter( )方法做必要的处理。当前鼠标位置的x和y坐标由mousePressed( )方法以参数形式提供。

```
class MyMouseAdapter extends MouseAdapter {
  public void mousePressed(MouseEvent me) {
    selectLetter(me.getX(), me.getY());
  }
```

mouseReleased( )

mouseReleased( )方法调用dropLetter( )方法作必要的处理。当前鼠标位置的x和y坐标由mouseReleased( )方法以参数形式提供。

```
  public void mouseReleased(MouseEvent me) {
    dropLetter(me.getX(), me.getY());
  }
}
```

mouseDragged( )

在下面的代码段中，注意MyMouseMotionAdapter是扩展MouseMotionAdapter的内部子类。它重载了mouseDragged( )方法。

mouseDragged( )方法调用dropLetter( )方法作必要的处理。当前鼠标位置的x和y坐标由mouseDragged( )方法以参数形式提供。

```
class MyMouseMotionAdapter extends MouseMotionAdapter {
```

```
    public synchronized void mouseDragged(MouseEvent me) {
      dragLetter(me.getX(), me.getY());
    }
  }
}
```

31.3.5　Bag.java

相当于Board，Bag类非常简单明了。它是口袋字母的简单抽象。当创建Bag类时，传入一个随机数初值，这个相同的初值随机创建两个相同的口袋。随机数生成器存放在rand中。这个类包含两个有点怪的整数数组，名字分别是letter_counts 和letter_points。两个数组都是27个元素。数组的第0个元素代表空白方块，第1到26个元素代表A到Z。letter_counts数组代表口袋中的每个字母的个数。例如，letter_counts[1]为9，这就是说口袋里有9个A。同样，letter_points数组在字母和它们的分值之间做映射。A字母方块仅值1分，但一个Z值10分。100个字母存放在一个名为letters的数组中。在游戏中，口袋中实际剩余的字母数目存放在n中。

```
import java.util.Random;

class Bag {
  private Random rand;
  private int letter_counts[] = {
    2, 9, 2, 2, 4, 12, 2, 3, 2, 9, 1, 1, 4, 2,
    6, 8, 2, 1, 6, 4, 6, 4, 2, 2, 1, 2, 1
  };
  private int letter_points[] = {
    0, 1, 3, 3, 2, 1, 4, 2, 4, 1, 8, 5, 1, 3,
    1, 1, 3, 10, 1, 1, 1, 1, 4, 4, 8, 4, 10
  };
  private Letter letters[] = new Letter[100];
  private int n = 0;
```

Bag( )

Bag的构造函数有一个初值，方法使用这个随机初值生成一个Random对象。然后遍历letter_counts数组，构造合适数目的新的Letter对象，注意将空白的字母方块用星号代替。然后为每个字母调用putBack( )，将其放入口袋。

```
  Bag(int seed) {
    rand = new Random(seed);
    for (int i = 0; i < letter_counts.length; i++) {
      for (int j = 0; j < letter_counts[i]; j++) {
        Letter l = new Letter(i == 0 ? '*' : (char)('A' + i - 1),
                              letter_points[i]);
        putBack(l);
      }
    }
  }
```

takeOut( )

这个方法比较聪明，但是效率不高，还不是致命错陷。takeOut( )在0~n-1之间挑选一个随机数。然后用这个随机数做偏移量从letters数组中抽取字母。它调用System.arraycopy( )填补letters中的空洞。然后n减1，返回字符。

```
synchronized Letter takeOut() {
  if (n == 0)
    return null;
  int i = (int)(rand.nextDouble() * n);
  Letter l = letters[i];
  if (i != n - 1)
    System.arraycopy(letters, i + 1, letters, i, n - i - 1);
  n--;
  return l;
}
```

putBack( )

putBack( )方法是构造函数用来将字母方块放入原始口袋中的。在将来的游戏增强版中，也可以用这个方法让游戏者将不喜欢的字母放回袋中，当然这样做要以放弃一个回合为代价。这个方法简单地将字母放在数组的结尾。

```
  synchronized void putBack(Letter l) {
    letters[n++] = l;
  }
}
```

31.3.6 Letter.java

Letter类比较简单，它并不涉及游戏或棋盘。它只是封装位置和一个单个字母外观。它使用几个静态变量保持字体和大小的信息。这样做使得小应用程序不会一次在内存中装入100种字体。这还有一个副作用，那就是浏览器不能包含两个不同大小的Scrabblet 小应用程序实例。第二个Scrabblet 小应用程序的初始化必须覆盖这些静态变量的值。

变量w和h保存的是每个字符的宽度和高度。变量font和smfont是AWT字体对象的大字体和小字体的值。整数y0和ys0分别存放字母基线的偏移量和点数。几个常数传入paint( )描述绘画的颜色状态：NORMAL，DIM和BRIGHT模式。

```
import java.awt.*;

class Letter {
  static int w, h;
  private static Font font, smfont;
  private static int y0, ys0;
  private static int lasth = -1;
  static final int NORMAL = 0;
  static final int DIM = 1;
  static final int BRIGHT = 2;
```

colors[ ], mix( ), gain( ) 和 clamp( )

colors数组被用9个颜色对象——3个颜色一组的3组——静态初始化。mix( )方法的输入是一组RGB值如250, 220, 100，然后将它们转换为三个颜色，用以提供3-D效果的高亮度和低亮度。mix( )方法调用gain( )加重或降低一个给定颜色的亮度，调用clamp( )以确保仍处在合法范围之内。

```
private static Color colors[][] = {
  mix(250, 220, 100),   // normal
  mix(200, 150, 80),    // dim
  mix(255, 230, 150)    // bright
};

private static Color mix(int r, int g, int b)[] {
  Color arr[] = new Color[3];

  arr[NORMAL] = new Color(r, g, b);
  arr[DIM] = gain(arr[0], .71);
  arr[BRIGHT] = gain(arr[0], 1.31);
  return arr;
}
private static int clamp(double d) {
  return (d < 0) ? 0 : ((d > 255) ? 255 : (int) d);
}
private static Color gain(Color c, double f) {
  return new Color(
    clamp(c.getRed() * f),
    clamp(c.getGreen() * f),
    clamp(c.getBlue() * f));
}
```

变量实例

在第一次绘制Letter时，使用valid标志确保所有与大小相关的变量只被创建一次。这里缓存了多个变量以减少每次小应用程序绘制屏幕时的计算量，如x0，w0，xs0，ws0和gap这些都将在下面介绍。使用tile Point对象保存Letter所占据的15×15棋盘上的格子。如果这个变量为空，则Letter不在棋盘上。使用x,y对准确定位Letter。

```
private boolean valid = false;

// quantized tile position of Letter. (just stored here).
private Point tile = null;
int x, y;             // position of Letter.
private int x0;        // offset of symbol on tile.
private int w0;        // width in pixels of symbol.
private int xs0;       // offset of points on tile.
private int ws0;       // width in pixels of points.
private int gap = 1;    // pixels between symbol and points.
```

Letter( ), getSymbol( ) 和 getPoints( )

symbol是保持要显示字母的字符串，points是这个字符的分数值。这两个变量都在构造

函数中初始化，分别由包装方法getSymbol( ) 和getPoints( )返回。

```
private String symbol;
private int points;

Letter(char s, int p) {
  symbol = "" + s;
  points = p;
}
String getSymbol() {
  return symbol;
}

int getPoints() {
  return points;
}
```

move( ), remember( )和 recall( )

用move( )方法指示字母方块绘制的位置。但是，remember( )方法就复杂一些。如果它的参数为空，则意味着这个方块“忘记”了它原来的位置。这说明这个字符并没有使用。否则，则这个方法指示这个字母占据的棋盘位置坐标。调用recall( )可检查此状态。

```
void move(int x, int y) {
  this.x = x;
  this.y = y;
}

void remember(Point t) {
  if (t == null) {
    tile = t;
  } else {
    tile = new Point(t.x, t.y);
  }
}

Point recall() {
  return tile;
}
```

resize( )

棋盘调用resize( )方法一次，指示字母的大小。记住，w和h是静态的，所以这立刻影响所有的Letter实例。

```
static void resize(int w0, int h0) {
  w = w0;
  h = h0;
}
```

hit( )

如果xp，yp对落在这个Letter的范围内，则hit( )方法返回true。

```
boolean hit(int xp, int yp) {
```

```
    return (xp >= x && xp < x + w && yp >= y && yp < y + h);
  }
```

validate( )

使用validate( )方法装载字体，确定字体大小，决定绘制的位置。这些信息缓存在前面讨论过的私有变量中。这些计算的结果在下面的paint( )中使用。

```
  private int font_ascent;
  void validate(Graphics g) {
    FontMetrics fm;
    if (h != lasth) {
      font = new Font("SansSerif", Font.BOLD, (int)(h * .6));
      g.setFont(font);
      fm = g.getFontMetrics();
      font_ascent = fm.getAscent();

      y0 = (h - font_ascent) * 4 / 10 + font_ascent;

      smfont = new Font("SansSerif", Font.BOLD, (int)(h * .3));
      g.setFont(smfont);
      fm = g.getFontMetrics();
      ys0 = y0 + fm.getAscent() / 2;
      lasth = h;
    }
    if (!valid) {
      valid = true;
      g.setFont(font);
      fm = g.getFontMetrics();
      w0 = fm.stringWidth(symbol);
      g.setFont(smfont);
      fm = g.getFontMetrics();
      ws0 = fm.stringWidth("" + points);
      int slop = w - (w0 + gap + ws0);
      x0 = slop / 2;
      if (x0 < 1)
        x0 = 1;
      xs0 = x0 + w0 + gap;
      if (points > 9)
        xs0--;
    }
  }
```

paint( )

棋盘调用paint( )方法。一个整数i，被传入方法，这个i代表NORMAL, BRIGHT 或DIM中的一个。这个i是colors数组的下标，可以选择一个基础颜色。填充一个矩形创建一个3D外观的高亮度带阴影的按钮。如果points比零大，指明一个非空的字母，在绘制完主要字母后，在字母旁边绘制这个分数值。

```
  void paint(Graphics g, int i) {
    Color c[] = colors[i];
    validate(g);
    g.setColor(c[NORMAL]);
```

```
      g.fillRect(x, y, w, h);
      g.setColor(c[BRIGHT]);
      g.fillRect(x, y, w - 1, 1);
      g.fillRect(x, y + 1, 1, h - 2);
      g.setColor(Color.black);
      g.fillRect(x, y + h - 1, w, 1);
      g.fillRect(x + w - 1, y, 1, h - 1);
      g.setColor(c[DIM]);
      g.fillRect(x + 1, y + h - 2, w - 2, 1);
      g.fillRect(x + w - 2, y + 1, 1, h - 3);
      g.setColor(Color.black);
      if (points > 0) {
        g.setFont(font);
        g.drawString(symbol, x + x0, y + y0);
        g.setFont(smfont);
        g.drawString("" + points, x + xs0, y + ys0);
      }
    }
  }
```

31.3.7 ServerConnection.java

这个小应用程序客户方的最后一个类是ServerConnection，这个类封装与服务器和对手之间的通信。在类的开始部分声明了多个变量。与服务器相连的套接字端口号为6564。CRLF是Internet上的常数字符串，代表行结束。与服务器之间的I/O流分别为in和out。服务器给这个连接分配的惟一的ID存放在id中。对手的ID存放在toid中。连接的Scrabblet 小应用程序为scrabblet。

```
import java.io.*;
import java.net.*;
import java.util.*;

class ServerConnection implements Runnable {
  private static final int port = 6564;
  private static final String CRLF = "\r\n";
  private BufferedReader in;
  private PrintWriter out;
  private String id, toid = null;
  private Scrabblet scrabblet;
```

ServerConnection( )

ServerConnection构造函数利用一个Internet站点名，打开一个套接字连接对应主机上的端口。如果成功，它用InputStreamReader 和BufferedReader包装输入，用PrintWriter包装输出。如果连接失败，向调用者引发一个异常。

```
  public ServerConnection(Scrabblet sc, String site) throws
    IOException {
    scrabblet = sc;
    Socket server = new Socket(site, port);
    in = new BufferedReader(new
            InputStreamReader(server.getInputStream()));
    out = new PrintWriter(server.getOutputStream(), true);
```

```
}
```

readline( )

readline( ) 方法仅将原来readLine( )中的IOException变为一个为null 的返回值。

```
private String readline() {
  try {
    return in.readLine();
  } catch (IOException e) {
    return null;
  }
}
```

setName( ) 和 delete( )

setName( )方法通知服务器本地游戏者的名字，使用delete( )方法将自己从服务器保持的列表上删除。

```
void setName(String s) {
  out.println("name " + s);
}

void delete() {
  out.println("delete " + id);
}
```

setTo( ) 和 send( )

setTo( )方法绑定对手的ID。将来的send( )调用转移到游戏者。

```
void setTo(String to) {
  toid = to;
}

void send(String s) {
  if (toid != null)
    out.println("to " + toid + " " + s);
}
```

challenge( ), accept( ), chat( ), move( ), turn( )和 quit( )

下列的短小方法都是从客户端向服务器发送一行消息，然后服务器将这些消息发送给对手。用challenge消息初始启动游戏，accept消息用来响应挑战。每次移动一个字母，发送一个move消息，然后在每次回合结束时发送turn消息。如果客户退出或是离开小应用程序所在的页面，则发送quit消息。

```
void challenge(String destid) {
  setTo(destid);
  send("challenge " + id);
}

void accept(String destid, int seed) {
  setTo(destid);
```

```
    send("accept " + id + " " + seed);
  }

  void chat(String s) {
    send("chat " + id + " " + s);
  }

  void move(String letter, int x, int y) {
    send("move " + letter + " " + x + " " + y);
  }

  void turn(String words, int score) {
    send("turn " + score + " " + words);
  }

  void quit() {
    send("quit " + id);  // tell other player
    out.println("quit"); // unhook

  }
```

start( )

这个方法简单的启动线程管理客户方面的网络。

```
  // reading from server...

  private Thread t;

  void start() {
    t = new Thread(this);
    t.start();
  }
```

Keywords

在这里显示的静态变量和静态块被用来初始化keys Hashtable，这是使用这个散列表在keystrings中的字符串和数组位置之间映射——例如，keys.get("move") == MOVE来实现的。lookup( )方法负责将Integer对象解开为正确的整数，如果是-1，表示没有找到关键字。

```
  private static final int ID = 1;
  private static final int ADD = 2;
  private static final int DELETE = 3;
  private static final int MOVE = 4;
  private static final int CHAT = 5;
  private static final int QUIT = 6;
  private static final int TURN = 7;
  private static final int ACCEPT = 8;
  private static final int CHALLENGE = 9;
  private static Hashtable keys = new Hashtable();
  private static String keystrings[] = {
    "", "id", "add", "delete", "move", "chat",
    "quit", "turn", "accept", "challenge"
  };
```

```
static {
  for (int i = 0; i < keystrings.length; i++)
    keys.put(keystrings[i], new Integer(i));
}

private int lookup(String s) {
  Integer i = (Integer) keys.get(s);
  return i == null ? -1 : i.intValue();
}
```

run( )

run( )是游戏连接服务器的主循环。它进入一个阻塞调用的readline( )，这个调用在服务器返回一行文字时返回一个字符串。它使用StringTokenizer将一行文字拆为单词。switch语句基于输入行的第一个单词分配合适的代码。在协议中每个关键字解析不同的输入行，多数处理重新调用Scrabblet类完成工作。

```
public void run() {
  String s;
  StringTokenizer st;
  while ((s = readline()) != null) {
    st = new StringTokenizer(s);
    String keyword = st.nextToken();
    switch (lookup(keyword)) {
    default:
      System.out.println("bogus keyword: " + keyword + "\r");
      break;
    case ID:
      id = st.nextToken();
      break;
    case ADD: {
        String id = st.nextToken();
        String hostname = st.nextToken();
        String name = st.nextToken(CRLF);
        scrabblet.add(id, hostname, name);
      }
      break;
    case DELETE:
      scrabblet.delete(st.nextToken());
      break;
    case MOVE: {
        String ch = st.nextToken();
        int x = Integer.parseInt(st.nextToken());
        int y = Integer.parseInt(st.nextToken());
        scrabblet.move(ch, x, y);
      }
      break;
    case CHAT: {
        String from = st.nextToken();
        scrabblet.chat(from, st.nextToken(CRLF));
      }
      break;
    case QUIT: {
        String from = st.nextToken();
```

```
            scrabblet.quit(from);
          }
          break;
        case TURN: {
            int score = Integer.parseInt(st.nextToken());
            scrabblet.turn(score, st.nextToken(CRLF));
          }
          break;
        case ACCEPT: {
            String from = st.nextToken();
            int seed = Integer.parseInt(st.nextToken());
            scrabblet.accept(from, seed);
          }
          break;
        case CHALLENGE: {
            String from = st.nextToken();
            scrabblet.challenge(from);
          }
          break;
        }
      }
    }
  }
```

31.4 服务器程序代码

最后的这两个类不是小应用程序的部分。但是，必须加载并运行在小应用程序类下载的源web服务器上。在web站点上安装并运行这种叫做“守护程序”的程序是需要安全权限的，这个权限一般人是没有的。但幸运的是，多数读者使用这个游戏时，不必建立自己的类似服务器，只需连接到那些已经存在的服务器上。

31.4.1 Server.java

Server是Scrabblet游戏服务端方的主类。一旦在web服务器上安装了这个类，就可以使用服务器系统上的Java解释器的命令行（例如java.exe, jview.exe, 或是sj.exe）运行这个程序，下面是Windows 95/98/NT/2000上的例子：

```
C:\java\Scrabblet> jview Server
```

在运行期间，Server以下列消息响应：

```
Server listening on port 6564
```

Server类开头定义了几个变量。端口号必须相同，6564，如在ServerConnection中看到的那样。使用idcon Hashtable存储与所有的客户的连接。用散列表而不是数组更方便经常的插入和删除操作，这些操作需要大量的数组拷贝。在接受每个新连接的时候，id递增。这对应于前面在客户端看到的id实例变量。

```
import java.net.*;
import java.io.*;
```

```
import java.util.*;

public class Server implements Runnable {
  private int port = 6564;
  private Hashtable idcon = new Hashtable();
  private int id = 0;
  static final String CRLF = "\r\n";
```

addConnection( )

每当有新的客户连接到我们的小应用程序时就调用addConnection( )。这个方法创建一个ClientConnection的新实例（将在下面介绍）来管理客户。它传递给Server一个引用、客户连接的套接字和当前的id。最后，它增加id的值准备接收下一个连接。

```
  synchronized void addConnection(Socket s) {
    ClientConnection con = new ClientConnection(this, s, id);
    // we will wait for the ClientConnection to do a clean
    // handshake setting up its "name" before calling
    // set() below, which makes this connection "live."
    id++;
  }
```

set( )

从ClientConnection中调用set( )方法响应客户告诉了我们它的“名字”。set( )方法跟踪所有在idcon散列表中的连接，然后从这个表中删除这个id防止重复接受客户的名字。这个方法调用setBusy(false)表示这个连接已经可以开始游戏了。然后它通过列举idcon散列表的关键字的方法遍历所有的连接。对所有非忙的连接（那些等待对手的游戏者），set( )发出一个“add”消息通知这些游戏者这个新连接。

```
  synchronized void set(String the_id, ClientConnection con) {
    idcon.remove(the_id) ;  // make sure we're not in there twice.
    con.setBusy(false);
    // tell this one about the other clients.
    Enumeration e = idcon.keys();
    while (e.hasMoreElements()) {
      String id = (String)e.nextElement();
      ClientConnection other = (ClientConnection) idcon.get(id);
      if (!other.isBusy())
        con.write("add " + other + CRLF);
    }
    idcon.put(the_id, con);
    broadcast(the_id, "add " + con);
  }
```

sendto( )

在响应“to”消息时调用sendto( )方法。它将body字符串的内容直接写入由dest标志的连接。

```
  synchronized void sendto(String dest, String body) {
    ClientConnection con = (ClientConnection)idcon.get(dest);
    if (con != null) {
```

```
      con.write(body + CRLF);
    }
  }
```

broadcast( )

在body中，使用broadcast( )方法向所有的除了在exclude中（通常是发送者）标识的单个连接发送一个消息。

```
synchronized void broadcast(String exclude, String body) {
  Enumeration e = idcon.keys();
  while (e.hasMoreElements()) {
    String id = (String)e.nextElement();
    if (!exclude.equals(id)) {
      ClientConnection con = (ClientConnection) idcon.get(id);
      con.write(body + CRLF);
    }
  }
}
```

delete( )

使用delete( )方法通知所有连接的客户忘掉曾经听过的the_id。在客户已经开始游戏，并将自己从可选的挑战者表中删除时使用。

```
synchronized void delete(String the_id) {
   broadcast(the_id, "delete " + the_id);
}
```

kill( )

当一个客户发送“quit”消息，明确的退出游戏时，或是当用户简单的退出浏览器时调用kill( )方法。

```
synchronized void kill(ClientConnection c) {
  if (idcon.remove(c.getId()) == c) {
    delete(c.getId());
  }
}
```

run( )

run( )方法是服务器的主循环。它在端口6564建立一个新套接字，然后进入无限循环等待来自客户的套接字连接。在接受连接时调用addConnection( )。

```
public void run() {
  try {
    ServerSocket acceptSocket = new ServerSocket(port);
    System.out.println("Server listening on port " + port);
    while (true) {
      Socket s = acceptSocket.accept();
      addConnection(s);
    }
  } catch (IOException e) {
```

```
      System.out.println("accept loop IOException: " + e);
    }
  }
```

main( )

main( )当然是由Java命令行解释器运行的方法。它创建一个Server实例，然后启动一个新的线程来运行。

```
  public static void main(String args[]) {
    new Thread(new Server()).start();
    try {
      Thread.currentThread().join();
    } catch (InterruptedException e) { }
  }
}
```

31.4.2　ClientConnection.java

这个类是小应用程序中ServerConnection的镜像。为每个客户创建一个连接。它的工作是管理与客户之间的I/O操作。私有变量实例保持着这个客户的状态。Socket存储在sock中。缓冲区阅读器和输出流存放在in和out中。客户机的主机名保存在host中。创建这个客户的Server实例的引用变量存放在server中。在客户机上的游戏者名字存放在name中，自动分配的游戏者ID存放在id中。布尔变量busy保持这个客户是否已经开始游戏的信息。

```
import java.net.*;
import java.io.*;
import java.util.*;

class ClientConnection implements Runnable {
  private Socket sock;
  private BufferedReader in;
  private OutputStream out;
  private String host;
  private Server server;
  private static final String CRLF = "\r\n";
  private String name = null;    // for humans
  private String id;
  private boolean busy = false;
```

ClientConnection( )

构造函数保存服务器的引用变量，套接字和惟一的ID。将InputStreamReader 和BufferedReader包装在输入外，这样可以调用readLine( )。然后这个方法将id写回给客户以便让客户知道分配给它的号码。最后创建并启动一个新线程处理连接。

```
  public ClientConnection(Server srv, Socket s, int i) {
    try {
      server = srv;
      sock = s;
      in = new BufferedReader(new
                      InputStreamReader(s.getInputStream()));
```

```
    out = s.getOutputStream();
    host = s.getInetAddress().getHostName();
    id = "" + i;

    // tell the new one who it is...
    write("id " + id + CRLF);

    new Thread(this).start();
  } catch (IOException e) {
    System.out.println("failed ClientConnection " + e);
  }
}
```

toString( )

重载toString( ) 这样可以清楚的表示一个连接记录。

```
public String toString() {
  return id + " " + host + " " + name;
}
```

getHost( ), getId( ), isBusy( ) 和 setBusy( )

将host, id, 和busy 包装在公用方法中允许只读访问。

```
public String getHost() {
  return host;
}

public String getId() {
  return id;
}

public boolean isBusy() {
  return busy;
}

public void setBusy(boolean b) {
  busy = b;
}
```

close( )

如果客户显式退出或是在读套接字时捕获到一个异常则调用close( )方法。调用在服务器中的kill( )方法，将自己从任何列表中删除。然后关闭套接字，同时也就关闭输入流和输出流。

```
public void close() {
  server.kill(this);
  try {
    sock.close();   // closes in and out too.
  } catch (IOException e) { }
}
```

write( )

为了将字符串写入流，不得不使用getBytes( )方法将它转换成一个字节数组。

```
public void write(String s) {
  byte buf[];
  buf = s.getBytes();
  try {
    out.write(buf, 0, buf.length);
  } catch (IOException e) {
    close();
  }
}
```

readline( )

readline( ) 方法仅将原来readLine( )中的IOException转换为返回一个null值。

```
private String readline() {
  try {
    return in.readLine();
  } catch (IOException e) {
    return null;
  }
}
```

Keywords

它和ServerConnection类中的相同部分类似，解析另一方的消息。用这个散列表在keystrings中的字符串和数组位置之间的映射——例如，keys.get("quit") == QUIT，使用在这里显示的静态变量和静态块初始化keys Hashtable， lookup( )方法负责将Integer对象解开为适当的整数，如果是-1，表示没有找到关键字。

```
static private final int NAME = 1;
static private final int QUIT = 2;
static private final int TO = 3;
static private final int DELETE = 4;

static private Hashtable keys = new Hashtable();
static private String keystrings[] = {
  "", "name", "quit", "to", "delete"
};
static {
  for (int i = 0; i < keystrings.length; i++)
    keys.put(keystrings[i], new Integer(i));
}

private int lookup(String s) {
  Integer i = (Integer) keys.get(s);
  return i == null ? -1 : i.intValue();
}
```

run( )

run( )循环管理所有与客户通信的消息。它使用StringTokenizer解析输入的信息行，每行的第一个单词是关键字。使用刚刚介绍的lookup( )方法在keys散列表中查询第一个单词。然后基于关键字的整数值变换。在客户第一次获得人名标志时发送NAME消息。在服务器上调用set( )建立连接。当客户希望终止这个服务器会话时，发送QUIT消息。TO消息包含一个目标ID和发送给客户的消息体。在服务器上调用sendto( )向前传送消息。最后一个消息是DELETE，当用户希望继续连接但是不希望出现在可选的游戏者列表中时发送。run( )设置忙（busy）标志然后调用服务器上的delete( )，通知用户不再希望被调用。

```
public void run() {
  String s;
  StringTokenizer st;
  while ((s = readline()) != null) {
    st = new StringTokenizer(s);
    String keyword = st.nextToken();
    switch (lookup(keyword)) {
    default:
      System.out.println("bogus keyword: " + keyword + "\r");
      break;
    case NAME:
      name = st.nextToken() +
        (st.hasMoreTokens() ? " " + st.nextToken(CRLF) : "");
      System.out.println("[" + new Date() + "] " + this + "\r");
      server.set(id, this);
      break;
    case QUIT:
     close();
     return;
    case TO:
      String dest = st.nextToken();
      String body = st.nextToken(CRLF);
      server.sendto(dest, body);
      break;
    case DELETE:
      busy = true;
      server.delete(id);
      break;
    }
  }
  close();
 }
}
```

31.5 改进Scrabblet

这个小应用程序代表一个复杂的客户/服务器，多玩家棋盘游戏。在将来的版本中，将以多种方式扩展Server和ServerConnection中的代码。它能够支持其他基于回合的游戏。它能跟踪并保持记录每个游戏的最高得分榜。它应能够动态扩展以支持新的协议。例如以本

章介绍的游戏为例，游戏应该增加一个查找功能，能够根据存储在服务器上的字典检查一组提交的单词。服务器因此能够仲裁这样的纠纷：xyzy是不是一个合法的单词。也可以构造一个单词机器人，这个机器人在服务器上，但是可以扮演一个游戏对手，使用字典从当前的七个单词组中生成一个最佳的单词。这个机器人甚至可以利用一个经典对话表在对话窗口中与对手聊天。这些增强功能希望能由读者自己完成。

这个小应用程序是为了娱乐和教学目的而制作的。如与任何商业产品雷同，纯属巧合。

附录　使用 Java 的文档注释

在本书的第一部分已经提到过，Java支持三种形式的注释。前两种是// 和/* */。第三种方式被称为文档注释。它以“/**”开始，以“*”标志结束。文档注释提供将程序信息嵌入程序的功能。开发者可以使用javadoc工具将信息取出，然后转换为HTML文件。文档注释提供了编写程序文档的便利方式。javadoc工具生成的文档几乎人人都看过，因为Sun的Java API文档库就是这么生成的。

javadoc标记

javadoc实用程序识别下列标记：

Tag标记	意义
@author	确定类的作者
@deprecated	指示反对使用这个类或成员
{@docRoot}	指定当前文档的根目录路径 (Java 2的1.3版新增)
@exception	确定一个方法引发的异常
{@link}	插入对另一个主题的内部链接
@param	为方法的参数提供文档
@return	为方法的返回值提供文档
@see	指定对另一个主题的链接
@serial	为默认的可序列化字段提供文档
@serialData	为writeObject( )或者writeExternal( )方法编写的数据提供文档
@serialField	为ObjectStreamField组件提供文档
@since	当引入一个特定改变时，声明发布版本
@throws	与@exception相同
@version	指定类的版本

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

可以使用文档注释为类、接口、域、构造函数和方法提供文档。在所有这些情况中，文档注释必须紧接在被注释的项目之前。为变量做注释，可以使用@see, @since, @serial, @serialField, 和@deprecated文档标记。为类做注释，可以使用@see, @author, @since, @deprecated, 和@version文档标记。方法可以用@see, @return, @param, @since, @deprecated, @throws, @serialData, 和@exception做标记。而{@link} 或 {@docRoot}标记

可以用在任何地方。下面详细列出了每个标记的使用方法：

@ author

标记@author 指定一个类的作者，它的语法如下：

```
@author description
```

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

@deprecated

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

```
@deprecated description
```

其中description 是描述反对的信息。由@deprecated 标记指定的信息由编译器识别，包括在生成的.class文件中，因此在编译Java源文件时，程序员可以得到这个信息。@deprecated 标记可以用于变量，方法和类的文档中。

{@docRoot}

{@docRoot}指定当前文档的根目录路径。

@exception

@exception 标记描述一个方法的异常，其语法如下：

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

其中，异常的完整名称由exception-name指定，explanation 是描述异常如何产生的字符串。@exception 只用于方法的文档。

{@link}

{@link} 标记提供一个附加信息的联机超链接。其语法如下：

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

其中，name 是加入超链接的类或方法的名字， text 是显示的字符串。

@param

@param 标记注释一个方法的参数。其语法如下所示：

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

其中parameter-name指定方法的参数名。这个参数的含义由explanation描述。@param 标记只用在方法的文档中。

@return

@return 标记描述一个方法的返回值。其语法如下：

```
@@return explanation
```

其中，explanation 描述方法返回值的类型和含意。@return 标记只用在方法的文档中。

@see

@see 标记提供附加信息的引用。最常见的使用形式如下所示：

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

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

@serial

@serial 标记为默认的可序列化字段定义注释文档。其语法如下：

```
@serial description
```

其中， description 是字段的注释。

@serialData

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

```
@serialData description
```

其中，description 是数据的注释。

@serialField

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

```
@serialField name type description
```

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

@since

@since标记声明由一个特定发布版本引入的类或成员。其语法如下：

```
@since release
```

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

@throws

@throws 标记与@exception标记的含义相同。

@version

@version标记指示类的版本。其语法如下：

```
@version info
```

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

文档注释的一般形式

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

下面是一个类的文档注释的例子：

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

javadoc的输出

javadoc程序将Java程序的源文件作为输入，输出几个包含该程序文档的HTML文件。每个类的信息在其自己的HTML文件中。同时，javadoc还输出一个索引和一个层次结构树。Javadoc还可生成其他HTML文件。不同实现版本的javadoc可能工作方式有所不同，应该仔细阅读Java开发系统的说明书以了解此版本的细节处理。

一个使用文档注释的例子

下面是一个使用文档注释的程序范例。注意每个注释都在它描述的对象之前。在javadoc处理之后，SquareNum.html就是关于SquareNum类的文档。

```
import java.io.*;

/**
 * This class demonstrates documentation comments.
 * @author Herbert Schildt
 * @version 1.2
*/
public class SquareNum {
  /**
   * This method returns the square of num.
   * This is a multiline description.  You can use
   * as many lines as you like.
   * @param num The value to be squared.
   * @return num squared.
  */
  public double square(double num) {
    return num * num;
  }

  /**
   * This method inputs a number from the user.
   * @return The value input as a double.
   * @exception IOException On input error.
   * @see IOException
  */
  public double getNumber() throws IOException {
    // create a BufferedReader using System.in
    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader inData = new BufferedReader(isr);
    String str;

    str = inData.readLine();
    return (new Double(str)).doubleValue();
  }
  /**
   * This method demonstrates square().
   * @param args Unused.
   * @return Nothing.
   * @exception IOException On input error.
   * @see IOException
  */

  public static void main(String args[])
    throws IOException
  {
    SquareNum ob = new SquareNum();
    double val;

    System.out.println("Enter value to be squared: ");
    val = ob.getNumber();
    val = ob.square(val);

    System.out.println("Squared value is " + val);
  }
}
```