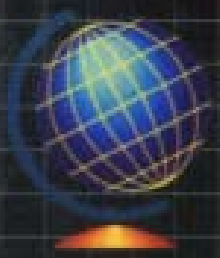


计算机实用技术丛书

jisuanji shiyong jishu congshu

Java 编程技术

谭浩强 主编

程龙 杨海兰 吴功宜 编

人民邮电出版社
POSTS & TELECOMMUNICATIONS PRESS

计 算 机 实 用 技 术 丛 书

Java 编程技术

人 民 邮 电 出 版 社

Java 编程技术/谭浩强主编；程龙，杨海兰，吴功宜编. —北京：人民邮电出版社，2003.3
ISBN 7-115-10895-1

Ⅰ. J… Ⅱ. ①谭… ②程… ③杨… ④吴… Ⅲ. JAVA 语言—程序技术 Ⅳ. TP312

中国版本图书馆 CIP 数据核字(2003)第 007567 号

内 容 提 要

Java 语言具有面向对象、平台无关性、安全性、健壮性和多线程等优良特性，为用户提供了良好的程序设计环境，特别适合因特网开发，成为网络时代最重要的语言之一。

本书对 Java 语言的内容、功能、特性以及实际应用作了深入浅出的全面介绍，对面向对象、多线程、异常处理、Applet 程序设计、数据库编程以及网络编程等作了重点阐述，并结合 Java 的最新发展，对 J2EE 和 J2ME 的开发也作了详细介绍。

本书在注重系统性和科学性的同时，力求突出实用性。在介绍相关的编程原理和基础知识的前提下，着重利用丰富实用的例子来演示 Java 编程技术的魅力。

本书可作为高等院校“Java 程序设计”课程的教材或教学参考书，也适合软件开发人员参考阅读。

计算机实用技术丛书

Java 编程技术

◆ 主　　编　谭浩强
编　　程龙　杨海兰　吴功宜
责任编辑　滑　玉

◆ 人民邮电出版社出版发行　　北京市崇文区夕照寺街 14 号

邮编　100061　　电子函件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
读者热线：010-67180876
北京汉魂图文设计有限公司制作
北京顺义向阳胶印厂印刷
新华书店总店北京发行所经销

◆ 开本：787×1092　1/16
印张：19
字数：460 千字　　2002 年 8 月第 1 版
印数：1－0 000 册　　2002 年 8 月北京第 1 次印刷

ISBN 7-115-10895 - 1/TP・3214

定价：25.00 元

21世纪是信息时代，每一个有文化的人都应当学习计算机和网络的知识，充分利用现代信息技术，改变我们的工作方式、学习方式和生活方式。

现在，计算机教育已经深入到每一个领域、每一个角落，学习计算机的人愈来愈广泛。从研究生、大学生到中小学生，从知识分子到公务人员，从青年人到老年人，都以空前的热情投入到学习计算机的热潮中。一般来说，计算机教育包括三个层次：

1．计算机入门教育。对象是原先没有学习过计算机的人，它带有扫盲的性质，使初学者初步具有计算机的基础知识，能进行计算机的初步操作。

2．计算机技术教育。对象是已具有一定的计算机基础知识、准备以计算机为工具去完成某些实际任务的人。

3．计算机专业教育。对象是计算机专业的大学生、研究生以及计算机的专业人员。他们应当具有比较系统而深入的计算机专业知识。

显然，不同层次的计算机教育应当有不同的学习内容和学习要求。

在我国第一次和第二次计算机普及高潮中，有数千万人初步学习了计算机知识，他们对计算机产生了浓厚的兴趣，现在他们之中有不少人强烈地希望进一步学习，并能把计算机技术用于实际工作，使所学的知识真正发挥作用。

针对这一情况，浩强创作室组织编写了这套《计算机实用技术丛书》，它的对象是已有一定的计算机基础知识的人，也就是说这套丛书属于第二层次——计算机技术教育的范围，要求学习者有一定的计算机基础知识。我们同时考虑到，学习这套丛书的读者多数原来是在非计算机专业的领域学习和工作的，不一定全部都具有系统的计算机知识，因此本丛书在写法上力求深入浅出。我们不采用传统编写教材的三部曲——“提出概念—解释概念—举例说明”，而采用面向实际应用的新三部曲——“提出问题—介绍解决问题的方法—归纳规律和结论”。实践证明，这是一种行之有效的方法。

本丛书不是全面、系统地介绍有关各方面的知识，而是紧紧扣住“应用”展开，以应用为目的，以应用为出发点。根据应用的需要，由浅入深地分为若干个单元，在每一单元中包括若干个任务，以具体任务驱动；读者只要按照书上介绍的步骤跟着做一遍，就能初步掌握有关的应用技术。在此基础上，进一步通过实践累积经验，举一反三，不断扩展自己的应用知识。

本丛书由浩强创作室策划和组织编写，参加策划、组织和编写的有谭浩强、薛淑斌、秦建中、吴功宜、廖彬山、尚晓航、邵丽萍、赵野军、赵丰年、孙立军、梁洵、俞必忠、邵光亚等同志，由谭浩强教授担任主编并最后审定各书的内容。

我们将根据计算机应用技术发展的需要，陆续增加其他内容的节目，以满足读者的需要。本丛书如有不足之处，希望得到专家和读者的指正。

谭浩强谨识

2002年4月于清华园

欢迎访问谭浩强网站：http://www.tanhaoqiang.com

编 者 的 话

Java 是 Sun 公司推出的面向对象的程序设计语言，Java 将面向对象、平台无关性、健壮性、安全性、多线程等诸多特性集于一身，特别是其与网络紧密结合的特点，使其已经成为网络时代最重要、最有前途的语言之一。

本书紧扣 Java 编程语言的特点，从平台独立型、面向对象、安全性、多线程和网络编程等多个方面逐一展开进行讲解，从不同方面展现了 Java“编写一次，随处运行”的精髓。

本书从 Java 语言的发展入手，讲述了 Java 编程中的各个方面。内容浅显易懂，循序渐进，即使是没有编程经验的新手，通过本书的阅读和学习也可以较快地掌握 Java 编程技术，不但掌握 J2SE 技术，更会对 J2EE、J2ME 等技术有全面的了解，并将本书介绍的大量应用实例应用到实际开发中去。

本书围绕 Java 技术的各个方面，从 Java 的标准版 J2SE 入手，详尽介绍了 Java 中最基础的编程知识，并用大量的实例加深读者对相关知识的理解。全书共 14 章，第 1 章介绍 Java 语言的发展历程、Java 语言的特性以及 Java 语言的运行环境；第 2 章介绍 Java 的基本数据类型、运算符和表达式；第 3 章介绍了 Java 语言的流程控制，前三章是 Java 的基本编程知识，是学习任何一门语言都必须掌握的；第 4 章介绍用 Java 如何实现面向对象编程，包括类、接口和包；和其它语言不同，Java 中字符串是用类来实现的，因此在介绍了类的概念之后，第 5 章介绍了数组、向量和字符串，这是 Java 的复杂数据类型；第 6 章介绍了 Java 的异常处理，其内容包括异常的概念，如何抛出、捕获和处理异常等；第 7 章介绍了 Java 的输入 / 输出流，通过本章的学习可以知道如何在程序中使用各种输入 / 输出流；第 8 章介绍了 Java 的线程程序设计方法，其内容包括线程的概念、线程的创建、线程的状态与同步等；第 9 章介绍了 Applet 编程技术；第 10 章介绍了 Java 如何实现与平台无关的图形用户界面；第 11 章介绍了如何使用 JDBC 编程技术访问数据库；由于 Java 是一种面向对象的编程语言，因此，第 12 章详细介绍了 Java 语言的网络编程技术。

上面介绍的内容都是 Java 标准版 J2SE，随着 Java 在开发企业应用系统方面的应用，Sun 公司推出了基于 Java 组件技术的企业应用系统开发规范 J2EE。第 13 章介绍了 J2EE 技术，包括 J2EE 的产生，J2EE 的体系结构、组件、平台服务以及容器等。为了满足消费电子和嵌入设备的需要，Sun 公司推出了 J2ME 技术。第 14 章介绍了 J2ME 技术和利用 Java 语言进行手机应用程序开发。

本书在内容的安排上力求通俗易懂，深入浅出，使读者能在短期内掌握 Java 语言的程序设计。即适合于广大 Java 爱好者参考和自学，也可作为高等院校 Java 程序设计课程的教材。

本书写作过程中，主要依据 Sun 公司的 Java 语言标准，并参考了 Patrick Naughton 所编著的《Java 使用手册》以及其他一些在因特网上公布的研究论文和相关资料，书中恕不一一注明出处。这些资料来源于众多的大学、研究机构、商业团体以及一些研究 Java 编程的个人，

对于他们推动 Java 编程的应用和发展表示衷心的感谢。写作过程中所参考的这些书籍资料，其原文版权属于原作者，特此声明。

由于本书编写的时间和作者自身的水平有限，所以书中难免有不足之处，敬请读者批评指教。

作　者

2002.8

目　录

第1章

Java 语言概述

了解一门编程语言最好先了解它的背景。本章介绍 Java 的起源及其发展，并概要介绍 Java 语言的特点。

本章的主要内容包括:

- Java 语言的起源及发展
- Java 语言的特点
- Java 的运行系统与 Java 虚拟机
- Java 开发环境
- 简单的 Java 程序

1.1 Java 语言的起源及发展

1991 年，为了对家用消费类电子产品进行交互式操作，Sun MicroSystems 公司（简称 Sun 公司）的 Jame Gosling 领导的 Green 开发小组开发了一个分布式代码系统，使用这个分布式代码系统可以把 E-mail 发给电冰箱、电视机等家用电器，并对它们进行控制。在研究过程中，Gosling 发现：消费类电子产品的用户不同于工作站的用户，他们不关心 CPU 的型号，他们只需要建立在标准基础之上的与硬件平台无关的一系列可选方案即可。最初，开发小组想用 C++语言作为他们的开发语言，但由于 C++语言太复杂，用 C++语言写的程序必须针对特定的计算机芯片进行编译，一旦编译好后就很难适应新的软件库，并且由于 C++语言中指针的使用、没有数组越界的检查、缺乏自动的内存管理以及过于灵活的数据类型转换等而导致其安全性很差，并不适合他们项目的要求。因此，开发小组基于 C++语言开发了一种能更适合于消费类电子设备软件开发的新的程序设计语言，称为 Oak（橡树），之后更名为 Java。

Java 语言初步设计完成后，Sun 公司使用 Java 语言开发了一些技术上非常成功的试验软件，但由于激烈的市场竞争和其他一些商业上的原因，Sun 公司没能将这些技术上成功的产品推向市场。直到 1994 年，面对 Internet 的迅猛发展以及环球信息网 WWW 的快速增长，Sun 公司审时度势，将 Java 定位到 Internet 的 WWW 应用开发上。同时，为了推动 Java 编程语言的发展，Sun 公司决定对 Java 采取免费提供的方式。Java 软件的免费下载网址为：http://java.sun.com/。如图 1.1 所示。

图 1.1 Sun 公司的 Java 站点主页

于是，Java 开发小组对 Java 语言进行了改进和包装，使之能更适合 Internet 的应用开发，并用 Java 语言开发了一个称作 HotJava 的 Web 浏览器。在当时，WWW 服务还只是静态的，而且只是一些静态的图像和文本，缺少交互性；虽然在 Web 页上也的确出现了一些实现诸如简单的绘画程序的 CGI 脚本，但实际上并没有交互性，而且客户端的请求还要送回服务器，这就给服务器增加了额外的负担。如果程序能够下载并能在客户端的浏览器上运行，那么服务器的负担就会减轻，HotJava 浏览器就是符合上述要求的浏览器。因此，1995 年 5 月 HotJava 浏览器推出后，引起了巨大的轰动。HotJava 是第一个能够自动装载和运行 Java 程序的浏览器，它的意义不仅仅在于使 Java 语言更加成熟，更重要的是向世人展示了 Java 的强大功能。从此，Java 语言便逐渐成为 Internet 上受欢迎的开发与编程语言。

1995 年，Sun 公司在“Sun World 95”大会上正式发布了 Java 技术。一些著名的计算机公司如 IBM、Netscape、Novell、Apple、DEC、Oracle、Borlan、SGI 等纷纷表示对 Java 语言的支持。Sun 公司与这些公司合作，共同致力于 Java 语言的发展。因此，Java 语言被美国的著名杂志 *PC Magazine* 评为 1995 年十大优秀科技产品（计算机类就此一项入选），随之出现了大量用 Java 语言编写的软件产品，受到工业界的重视与好评。微软公司总裁比尔·盖茨在悄悄地观察了一段时间后，不无感慨地说：“Java 语言是长时间以来最卓越的程序设计语言。比尔·盖茨确定微软整个软件开发的战略从 PC 单机转向以网络为中心的计算机，而购买 Java 使用权则是他的重大战略决策的实施部署。Sun 公司的总裁 Scott McNealy 认为：Java 为 Internet 和 WWW 开辟了一个崭新的时代。为了适应形势的需要，Sun 公司专门成立了一个子公司，即 SunSoft 公司，负责开发和推广 Java 技术，通过颁发许可证的办法来允许各家公司把 Java 虚拟机和 Java 的 Applets 类库嵌入到他们开发的操作系统中。这样，各类开发人员就能很容易地选择多种平台来使用 Java 语言编程，而且不同的用户也可以脱离 Web 浏览器来运行 Java 应用程序。这无疑为 Java 语言的应用开拓了极为广阔的前景。Java 是印尼的一个岛屿名，那里盛产咖啡，人们把那里产的咖啡称为 Java；而 Java 技术也的确像咖啡一样可口，令人耳目一新。

随着 Internet 技术的发展和应用的推广，Java 语言也不断有更新版本推出，以满足新的需求，图 1.2 简要地说明了 Java 版本的发展历程。

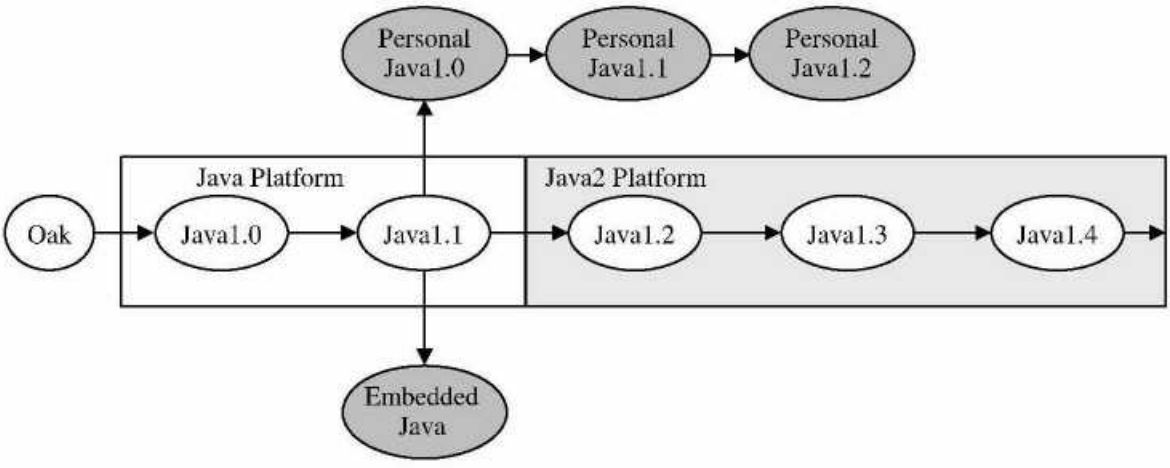


图 1.2 Java 版本发展历程

从图 1.2 中可以发现，自 Java 1.0 之后，Java 语言就被广泛地使用在桌上型应用程序以及 Applet 的开发上。但是，自 Java 1.1 开始，为了适应嵌入式开发的需要，Java 语言又增加了嵌入式系统上面的应用（又回到了它一开始的老路）。在当时，Sun 公司发表了 Embedded Java 与 Personal Java(也有人简称为 PJava)这两项规格。其中 Embedded Java 是为了资源十分有限，而且没有显示设备的装置而设计；而 Personal Java 则是为在能够与因特网联机并拥有显示系统(例如彩色 LCD)的消费性电子装置而设计。

接着 Java 的版本演进到 Java 2，这时，为了再明显区分各种 Java 的应用，又分割出了 J2EE、J2SE 以及 J2ME 三种版本，如图 1.3 所示。

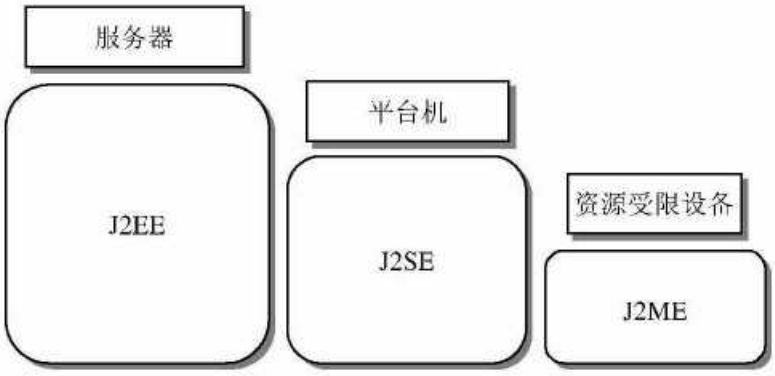


图 1.3　Java 版本的划分和各自针对的设备

（1）J2SE

J2SE（Java 2 Platform，Standard Edition）含有基本的 Java 2 SDK、工具、运行时（Runtime）和 API，开发者可以用来编写、部署和运行 Java 应用程序和 Applet。另外，它还包括了早期的 Java 开发工具包，即 Java Development Kit，简称 JDK，例如 JDK1.4。

（2）J2EE

J2EE(Java 2 Platform，Enterprise Edition)建立在 J2SE 的基础上，它是 JSP（Java Server Page）、Servlet、EJB（Enterprise JavaBean）、JTS（Java Transaction Service）、Java mail 以及 JMS（Java Message Service）等多项技术的混合体，并且还含有兼容性测试套件，主要用于开发分布式的、服务器端的多层结构的应用系统，例如电子商务网站。

（3）J2ME

J2ME（Java 2 Platform，Micro Edition）主要用于开发电子产品，例如移动电话、数字机顶盒、汽车导航系统等。

由于 Java 2 将 Java 的应用区分成 3 大块，Java 程序语言的发展不会再像 Java 1.1 那样如树枝状般扩散出去，这有助于人们理解 Java 的各种应用，而不会造成以后越发展下去越不可收拾的混乱局面。本书中，主要介绍标准 Java 版本 J2SE 的开发知识，并在最后两章中对 J2EE 和 J2ME 的程序开发作简要介绍。

1.2 Java 语言的特点

1.2.1 特点

Java 语言有下面一些特点：简单性、面向对象、面向网络、鲁棒性、安全性、结构中立性、可移植性、解释执行、高性能、多线程以及动态性。

1

Java 语言是一种面向对象的语言。它通过提供最基本的方法来完成指定的任务，只需理解一些基本的概念，就可以用它编写出适合于各种情况的应用程序；Java 是由 C++衍生而来的，其语言风格与 C++语言十分类似，有 C 或 C++语言基础的人可以十分容易地学会 Java 语言。同时，Java 略去了指针、结构、联合、隐式类型转换、头文件、运算符重载、多重继承等概念，并且通过实现自动垃圾回收，大大简化了程序设计者的内存管理工作，程序的可读性也更强。

2

Java 语言是一种完全面向对象的程序设计语言。除了几个基本数据类型外，其他类型都是对象。Java 的设计集中于对象及其接口，它提供了简单的类机制以及动态的接口模型。对象中封装了它的状态变量以及相应的方法，实现了模块化和信息隐藏；而类则提供了一类对象的原型，并且通过继承机制，子类可以使用父类所提供的方法，实现了代码的复用。

3

Java 是面向网络的语言。通过它提供的类库可以处理 TCP / IP 协议，用户可以通过 URL 地址在网络上很方便地访问其他对象。

4

Java 在编译和运行程序时，都要对可能出现的问题进行检查，以消除错误的产生。它提供自动垃圾收集来进行内存管理，防止程序员在管理内存时容易产生的错误。通过集成的面向对象的例外处理机制，即在编译时，Java 揭示出可能出现但未被处理的例外，帮助程序员正确地进行选择以防止系统的崩溃。另外，Java 在编译时还可捕获类型声明中的许多常见错误，防止动态运行时不匹配问题的出现。

5

用于网络、分布环境下的 Java 必须要考虑安全性，因此，Java 建立了下述一套严密的安全体系。

首先，Java 不支持指针，一切对内存的访问都必须通过对象的实例变量来实现，Java 程序员不能强制引用内存指针。这样就防止程序员使用欺骗手段访问对象的私有成员，同时也避免了指针操作中容易产生的错误。

其次，Java 内存分配延迟到运行时由运行系统决定，而不是像 C++语言一样由编译器决定。

这套严密的安全体系也是 Java 编译器的功能，它保证了 Java 的源代码不违反安全规则。Java 支持引入其他代码，并对此也提供了安全措施。Java 运行系统中有一个字节码验证器，

在字节码执行前对其进行代码验证，以确保代码段遵循如下规则：① 不存在伪造的指针；② 未违反访问权限；③ 严格遵循对象规范来访问对象；④ 用合适的参数调用方法；⑤ 没有栈溢出。

6

Java 解释器生成与体系结构无关的字节码指令。字节码指令十分类似于机器指令，但字节码指令又不是为某个特定的机器定义的，这些字节码指令对应于 Java 虚拟机中的表示。Java 语言编译后生成的字节码指令中还包含符号引用，即字节码中指令所有方法和变量的引用都还是名字，它们在执行时才被消解。只要安装了 Java 运行时系统，Java 程序就可在任意的处理器上运行，Java 解释器得到字节码后，对它进行转换，使之能够在不同的平台运行。

7

与平台无关的特性使 Java 程序可以方便地被移植到网络上的不同机器。Java 还定植了完全统一的语言文本，如 Java 的基本数据类型不会随目标机的变化而变化。同时，Java 的类库中也实现了与不同平台的接口，因而这些类库可以移植。另外，Java 编译器是由 Java 语言实现的，Java 运行时系统由标准 C 语言实现，这使得 Java 系统本身也具有可移植性。

8

Java 解释器直接对 Java 字节码进行解释执行。字节码本身携带了许多编译信息，使得连接过程更加简单。

9

一般情况下，可移植性、稳定性和安全性总是以牺牲性能为代价的，Java 语言很好地解决了这个问题。Java 编译生成的字节码与机器码十分接近，而且和其他解释执行的语言如 Basic、Tcl 不同，Java 字节码的设计使之能很容易地直接转换成对应于特定 CPU 的机器码，从而得到较高的性能。

10

多线程机制使应用程序能够并行执行，而且同步机制保证了对共享数据的正确操作。Java 在语言级嵌入了对多线程和并发的支持功能。通过使用多线程，程序设计者可以分别用不同的线程完成特定的行为，而不需要采用全局的事件循环机制，因而可以很容易地实现网络上的实时交互行为。

11

Java 的动态特性是其面向对象的延伸。Java 程序的基本组成单元为类，而 Java 的类又是在运行时动态装载的，这样可以保证 Java 在类库中自由地加入新的方法和实例变量而不会影响用户程序的执行。同时，Java 通过接口来支持多重继承，使之比严格的类继承具有更灵活的方式和扩展性。

1.2.2 Java 与 C 和 C++的比较

对于变量声明、参数传递、操作符、流控制等，Java 使用了和 C 及 C++语言相同的传统，使得熟悉 C 和 C++语言的程序员能很方便地进行编程。同时，Java 为了实现其简单、鲁棒、安全等特性，也摒弃了 C 和 C++语言中许多不合理的内容。

1

Java 程序中，不能在所有类之外定义全局变量，只能通过在一个类中定义公用、静态的

变量来实现一个全局变量。例如：

```
Class GlobalVar
{
    public static global_var;
}
```

在类GlobalVar中定义变量global_var为public static，使得其他类可以访问和修改该变量。同时Java对全局变量进行了更好的封装。而在C和C++语言中，依赖于不加封装的全局变量常常造成系统的崩溃。

2 goto

Java不支持C和C++语言中的goto语句，而是通过例外处理语句try、Catch、final等来代替C和C++语言中的goto语句处理遇到错误时跳转的情况，使程序更具可读性且更结构化。

3

指针是C和C++语言中最灵活也是最容易产生错误的数据类型。由指针所进行的内存地址操作常会造成不可预知的错误，同时，通过指针对某个内存地址进行显式类型转换后，可以访问一个C++语言中的私有成员，从而破坏安全性，造成系统的崩溃；而Java对指针进行完全的控制，程序员不能直接进行任何指针操作，例如把整数转化为指针，或者通过指针释放某一内存地址等。另外，数组作为类在Java中实现，更好地解决了数组访问越界这一C和C++语言中不作检查的错误。

4

C语言中，程序员通过库函数malloc()和free()来分配和释放内存，C++语言中则通过运算符new和delete来分配和释放内存。如果再次释放已释放的内存块或未被分配的内存块，则会造成系统的崩溃，同样，忘记释放不再使用的内存块也会逐渐耗尽系统资源。而在Java中，所有的数据结构都是对象，程序员通过运算符new为它们分配内存堆，通过new得到对象的处理权，而实际分配给对象的内存则可能随程序运行而改变，Java可以对此自动地进行管理并进行垃圾收集，这有效地防止了由于程序员的误操作而导致的错误，并且更好地利用了系统资源。

5

在C和C++语言中，对于不同的平台，编译器对于简单数据类型如int、float等分别分配不同长度的字节数，例如int在IBM PC中为16位，在VAX-11中为32位，这导致了代码的不可移植性。但Java对这些数据类型总是分配固定长度的位数，如int型总占32位，这就保证了Java的平台无关性。

6

在C和C++语言中，可以通过指针进行任意的类型转换，因而常常带来不安全性；而Java中，运行时系统对对象的处理要进行类型相容性检查，以防止不安全的转换。

7

C和C++用头文件来声明类的原型、全局变量以及库函数等；在大的系统中，要维护这些头文件是很困难的。而Java不支持头文件，类成员的类型和访问权限都封装在一个类中，运行时系统对访问进行控制，防止对私有成员的操作。同时，Java中用import语句来与其他类进行通信，以便使用它们的方法。

8

C 和 C++语言的结构和联合中的所有成员均为公有，这就带来了安全性问题。而 Java 中不包含结构和联合，所有的内容都封装在类中。

9

C 和 C++语言用宏定义实现的代码给程序的可读性带来了困难。而 Java 不支持宏，它通过关键字 final 来声明一个常量，以实现宏定义中广泛使用的常量定义。

1.3 Java 的运行系统与 Java 虚拟机

Java 中有两类应用程序，一类是有自己独立运行入口点的 Java 应用程序；另一类是被嵌入在 Web 页面中由 Web 浏览器来控制运行的 Java 小程序（Applet）。在执行时，它们都需要 Java 运行系统的支持，对于 Java 应用程序，Java 运行系统一般是指 Java 解释器；而对于 Applet，Java 运行系统一般是指能运行 Applet 的与 Java 兼容的 Web 浏览器，并且其中包含了支持 Applet 运行的环境。

Java 运行系统的功能是对字节码进行解释和执行，其工作过程可分为下述 3 步。

（1）由类装载器完成字节码的装载。在装载过程中，程序运行时所需要的所有代码都被装载（包括程序代码中调用到的所有类）。完成后，字节码中便保留了地址的符号引用信息，运行系统通过建立地址的符号引用信息与内存地址之间的对照表来确定程序的内存分配。

（2）由字节码检验器对字节码进行安全性检查。这种检查可以排除字节码中可能存在的违反访问权限、不规范数据类型以及非法调用等问题。

（3）字节码的翻译和执行。Java 字节码的运行可以有两种方式：一种是通过代码生成器，先将字节码翻译成适用于本系统的机器码，然后再送到硬件去执行，这是一种编译性工作方式；另一种是通过解释器将字节码翻译成机器码，然后由即时运行部件立即将机器码送硬件执行，这是一种解释性工作方式。Java 运行系统一般采用第二种方式，只有对那些运行速度要求高的软件，才采用编译性工作方式，这时需要使用特定的代码生成器来完成编译，从而更好地保证 Java 代码的高性能。Java 的运行时系统如图 1.4 所示。

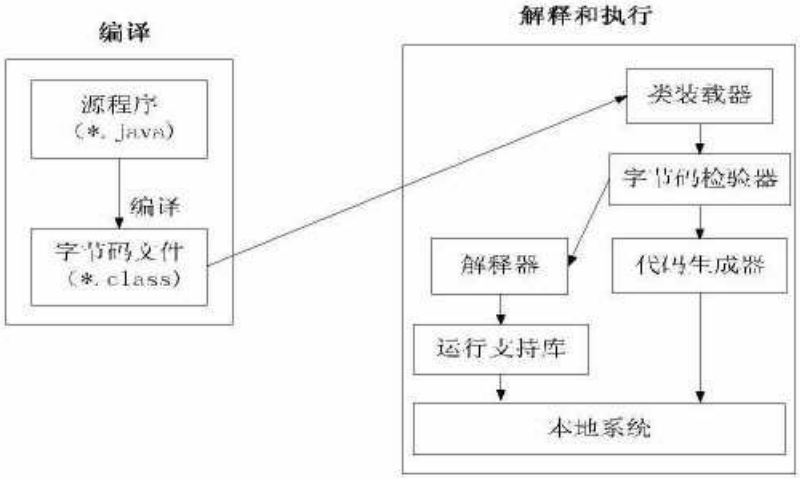


图 1.4 Java 的运行时系统

在执行时，由于 Java 的目标代码需要 Java 运行系统的支持，虽然 Java 运行系统被建立在各种不同的平台上，但为了做到 Java 的可移植性，被建立在不同平台上的 Java 运行系统的功能要求是统一的，为此 Java 引入了 Java 虚拟机（Java Virtual Machine，JVM）的概念，即在机器和编译程序之间加入了一层抽象的虚拟的机器。这台虚拟的机器在任何平台上都提供给编译程序一个共同的接口。编译程序只需要面向虚拟机，生成虚拟机能够理解的代码，然后由解释器来将虚拟机代码转换为特定系统的机器码执行。在 Java 中，这种供虚拟机理解的代码叫做字节码（ByteCode），它不面向任何特定的处理器，只面向虚拟机。从结构上看，Java 虚拟机由一组抽象的部件组成，这些部件包括指令集、寄存器组、类文件格式规定、堆栈、内存垃圾收集器和存储区 6 个部分。其中，指令集采用与平台无关的字节码形式，寄存器组中包含程序计数器、堆栈指针、运行环境指针和变量指针，类文件也与平台无关，堆栈用来传递参数和返回运行结果，垃圾收集器收集不再使用的内存片段，存储区则用来存放字节码。JVM 仅仅规定了部件的功能和规格，虽然这些功能和规格是统一的，但是并没有规定这些部件的具体实现技术，也就是说，可以用任何一种技术来实现。所以，JVM 是一种不具体的、能够使得任何一台实际机器运行 Java 字节码的规范机制。图 1.5 描述了在 Java 中，程序是如何在不同的机器上编译、运行的。

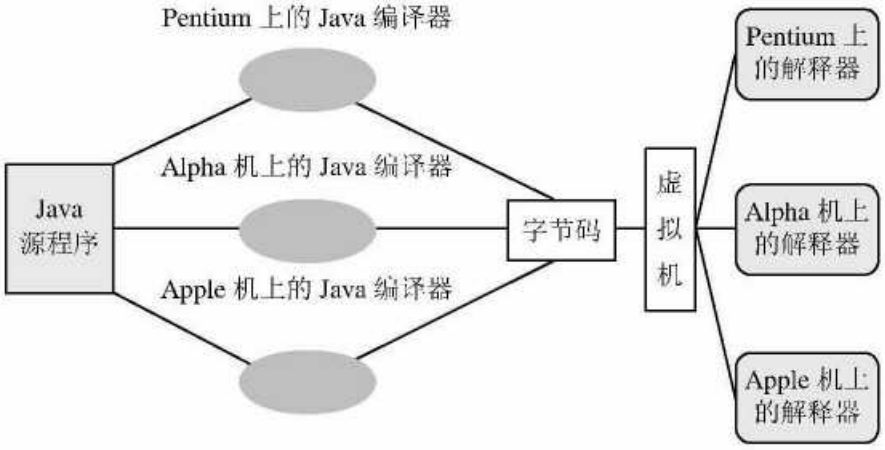


图 1.5 Java 程序在不同的机器上的编译执行过程

在 Java 推出初期，JVM 都是通过软件仿真的方法来实现的，这种方法目前仍被广泛采用，但是后来出现了硬件实现的 Java 虚拟机，如 Sun 公司的 Ultra Java。不管是用软件方法还是用硬件方法来实现 JVM，它们都必须符合并遵从 JVM 规范，而且只能执行统一格式的字节码。

1.4 Java 开发环境

1.4.1 JDK

在 Sun 公司推出 Java 语言的同时，也推出了 Java 的一系列开发工具，如 JDK（Java Developer's Kit）。JDK 是 Sun 公司推出的 Java 开发工具集；由于 Sun 公司是 Java 语言的创

始者和 Java 规范的制定者，所以 JDK 自然是最权威的 Java 开发工具。用户编写的 Java 程序是否符合 Java 语言规范的要求，用 JDK 提供的 Java 编译器一试便知。市场上其他公司的 Java 开发工具有的在某些地方并不完全遵从 Java 语言规范，例如微软公司的 Visual J++，但是 JDK 却绝对和 Java 语言规范保持一致，而且每当有新的 Java 语言规范推出时，就会有相应的 JDK 同时推出。用户可以从网上免费下载最新的 JDK 版本，其网址是：http://java.sun.com/downloads.html。下载完成后运行安装文件就可以很容易地完成安装了。

JDK 中提供了丰富的开发工具，其基本工具（Basic Tools）主要如表 1.1 所列。

表 1.1　　JDK 的基本开发工具

名　称	文 件 名	功 能 说 明
Java 编译器	javac.exe	将使用 Java 语言编写的程序编译成字节码
Java 解释器	java.exe	执行 Java 字节码，换句话说，它运行由 Java 语言编写的程序
Java 运行时刻解释器	jre.exe	类似于 Java 解释器，但是并不需要 java.exe 的所有的与开发相关的选项
Java 小程序观察器	appletviewer.exe	用来测试和运行小程序
Java 程序调试器	jdb.exe	用来帮助用户找到 Java 程序中的错误
类文件反编译器	javap.exe	反编译那些编译后的 Java 文件，并打印 Java 字节码的表示
Java 文档生成器	javadoc.exe	分析 Java 源文件中的声明和文档注释，并生成关于 Java 源文件的 HTML 页面，在 HTML 页面中描述了公有的和保护的类、接口、构造方法、方法、域。同时产生一个类的层次结构图，以及所有成员的一个索引
C 文件生成器	javah.exe	用于生成 C 语言头文件和存根文件（.h 文件和.c 文件），利用这些文件，编程人员可以把用 C 语言编写的源代码装载进 Java 应用程序
Java 压缩工具	jar.exe	将 Java 类文件和其他资源文件捆绑成一个单一的 jar 文件

除了上面的基本工具，JDK 中还包括数字签名工具（javakey——管理实体，包括它们的密钥、证明等）、国际化工具（native2ascii.exe——将基于本地字符集的代码文件转化成一个基于 ASCII 字符集的文件，并在新文件中使用本地字符的相应的 Unicode 代码\udddd 来代表本地字符）以及 RMI 工具（如 rmic.exe——根据包含远程对象实现的编译过的类文件名生成对象，rmiregistry.exe——在当前主机的指定端口创建和启动远程对象注册）等许多其他有用的工具。详细内容请参考相关的帮助文档。

Java 程序的编译程序是 javac.exe。javac 命令将 Java 程序编译成字节码，然后用 Java 解释器（即 Java 命令）来解释执行 Java 字节码。Java 程序源码必须存放在后缀为.java 的文件里。javac 可以将 Java 程序里的每一个类都生成与类具有相同名称但后缀为.class 的文件，并将.class 文件放在.java 文件的同一个目录里（除非用了-d 选项）。当用户引用到自己定义的类时，必须指明它们的存放目录，这就需要利用环境变量参数 CLASSPATH 来完成（环境变量

参数 CLASSPATH 由一些被分号隔开的路径名组成）。如果传递给 javac 编译器的源文件里引用到的类定义在本文件或传递的其他文件中找不到，则 javac 编译器会按 CLASSPATH 定义的路径来搜索。例如：

```
CLASSPATH=.;C:\Jdk1.4\classes
```

编译器先搜索当前目录，如果没搜索到，则继续搜索 C:\Jdk1.4\classes 目录。注意，除非用-classpath 选项来编译，系统总是将系统类的目录缺省地加在 CLASSPATH 后面。javac_g 是一个用于调试的未优化的编译器，功能与用法和 javac 一样。

另外，JDK 在 demo 目录中提供了大量丰富的例子程序，读者可以运行这些例子程序，并学习它们的源代码。

1.4.2 其他集成开发环境

除了 JDK 之外，其他一些著名的公司也相继推出了自己的 Java 集成开发工具，例如微软公司的 Visual J++，Borland 公司的 JBuilder，IBM 公司的 Visual Age for Java 以及 WebGain 的 Visual Cafe 等，这些工具都是 Java 的集成开发环境，而且大多数商业版本的功能都非常强大。

1 Forte for Java

作为 Java 技术的发明人和最主要的支持者，Sun 公司除了 JDK 之外，还推出集成开发环境 Forte for Java，它将调试器和用于 Web 应用开发系统的数据库组件整合到了统一的安装环境中。相关网站为 http://www.sun.com/forte/。

2 Borland JBuilder

Borland 是一个老牌的各类语言 IDE（集成开发环境）开发商。由 Borland 公司开发的 JBuilder 自然性能稳定、使用方便，特别适用于创建 Java 组件。JBuilder 的目标定位是代码开发人员而不是高级设计人员，所以 JBuilder 包含了大量的向导程序和其他针对中间层的快速开发工具。其网站是 http://www.borland.com/jbuilder。

3 IBM Visual Age for Java

IBM 的 Visual Age for Java 庞大，功能众多，所提供的是完整的解决方案。而且 IBM 使用自己的 JDK。相关网站是 http://www.ibm.com/software/vajava/。

4 WebGain Studio 4 with VisualCafe

VisualCafe 本是 Symantec 公司的著名产品，只不过 Symantec 公司已将其转让给了 WebGain 公司。VisualCafe 起到了全功能 Java 开发工具的作用，而且 WebGain 公司在其原来的基础上又增加了对 Enterprise JavaBean（EJB）和基本 UML 图表设计的支持。相关网站是 http://www.webgain.com/。

5 Oracle Internet Developer

这种工具只能针对 Oracle 数据库使用，尽管施加了这一限制，但其仍具有相当强大的中间层功能和采用 JSP 的 Web 应用程序开发功能。相关网站是 http://www.oracle.com/ip/develop/ids/index.html?jdeveloper.html。

对于初学者而言，与其一上来就面对那些复杂的 Java 集成开发环境，不如直接使用 JDK＋文本编辑器（如 UltrEditor）的方式学习，这样可以把主要的精力集中在 Java 语言本身。

1.5 简单的 Java 程序

下面我们先介绍两个简单的 Java 程序，并对其进行分析，了解一下 Java 程序究竟是什么样子，如何编写、编译和运行 Java 程序，从而建立一个总体的印象。

Java 程序主要有 Java 应用程序和 Applet 两种。

【例 1.1】HelloWorldApp.java

```
/*
  *这是一个简单的“Hello Word”程序
   */
public class HelloWorldApp
{
    public static void main (String args[])
    {
        //输出字符串“Hello World”到屏幕
        System.out.println(" Hello World!" );
    }
}
```

本程序的作用是输出下面一行信息：

```
Hello World!
```

编写 Java 程序首先用保留字 class 来声明一个新的类，其类名为 HelloWorldApp，它是一个公共类(public)。这里，类名与使用的文件名（包括字母的大小写）必须完全一样，因为 Java 解释器要求公共类必须放在与其同名的文件中。整个类定义用大括号{}括起来。在该类中定义了一个 main()方法，其中 public 表示访问权限，指明所有的类都可以使用这一方法；static 指明该方法是一个类方法，它可以通过类名直接调用；void 则指明 main()方法不返回任何值。对于一个应用程序来说，main()方法是必需的，而且必须按照如上的格式来定义。Java 解释器在没有生成任何实例的情况下，以 main()作为入口来执行程序。Java 程序中可以定义多个类，每个类中可以定义多个方法，但是最多只有一个公共类，main()方法也只能有一个作为程序的入口。与 C++类似，main 括号()中的 String args [] 是传递给 main()方法的参数，参数名为 args，它是类 String 的一个实例，参数可以为 0 个或多个，每个参数用“类名参数名”来指定，多个参数间用逗号分隔。在 main()方法的实现(大括号)中，只有一条语句：

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

它用来实现字符串的输出，这条语句实现的功能与 C 语言中的 printf 语句和 C++语言中的 cout<<语句的功能相同。

现在我们可以运行该程序。在 Windows / 95 / 98 / NT / 2000 中，单击“开始”，选择“程序” -> “MS-DOS 方式”，打开“命令行”窗口，进入到 HelloWorldApp.java 文件所在的目录，然后用下面的命令对它进行编译：

```
javac HelloWorldApp.java
```

编译的结果是生成字节码（bytecode）文件 HelloWorldApp.class，然后用 java 解释器来运行该字节码文件：

```
java HelloWorldApp
```

结果在屏幕上显示一行信息“Hello World!”，如图 1.6 所示。

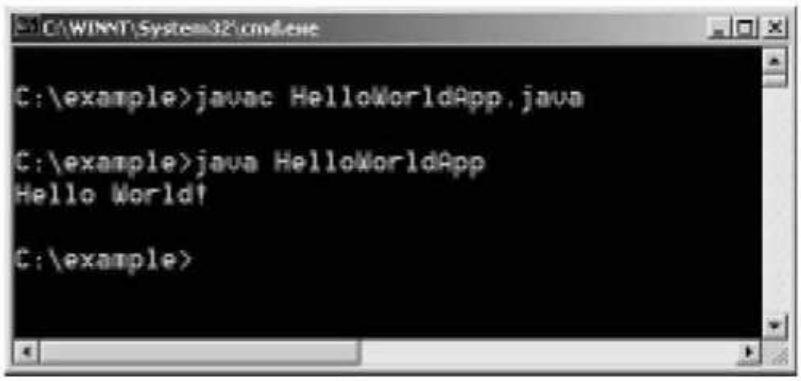


图 1.6 运行 Java 应用程序 HelloWorldApp 的输出结果

我们再来看下面的一个例子。

【例 1.2】

```
/*
  *这是一个简单的 Applet 小程序
  */
import java.awt.*;
import java.applet.*;
    public class HelloWorldApplet extends Applet{ //an applet
      public void paint(Graphics g){
            g.drawString(" Hello World!",20,20);
      }
}
```

这是一个简单的 Applet(小应用程序)。程序中，首先用 import 语句输入 java.awt 和 java.applet 下所有的包，使得该程序能够使用这些包中所定义的类，它类似于 C 语言中的 #include 语句。然后，声明一个公共类 HelloWorldApplet，用 extends 指明它是 Applet 的子类。在类中，我们重写父类 Applet 的 paint()方法，其中参数 g 为 Graphics 类，它表明当前作画的上下文。在 paint()方法中，调用 g 的方法 drawString()，在坐标(20,20)处输出字符串“Hello World!”，其中坐标是用像素点来表示的。

这个程序中没有实现 main()方法，这是 Applet 与应用程序 Application(如例 1.1)的区别之一。为了运行该程序，首先我们也要把它放在文件 HelloWorldApplet.java 中，然后对它进行编译：

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

得到字节码文件 HelloWorldApplet.class。由于 Applet 中没有 main()方法作为 Java 解释器的入口，我们必须编写 HTML 文件，把该 Applet 嵌入其中，然后用 appletviewer 来运行，或在支持 Java 的浏览器上运行。它的<HTML>文件如下：

```
<HTML>
<HEAD>
<TITLE> An Applet</TITLE>
</HEAD>
<BODY>
<applet code=" HelloWorldApplet.class"  width=200 height=40>
</applet>
</BODY>
</HTML>
```

其中用<applet>标记来启动 HelloWorldApplet，code 指明字节码所在的文件，width 和 height 指明 applet 所占的大小。我们把这个 HTML 文件存入 Example.html 中，然后运行：

```
C:\>appleviewer Example.html
```

这时屏幕上弹出一个窗口，其中显示 Hello World!。如图 1.7 所示。

图 1.7　运行 Java 小程序 HelloWorldApplet 的输出结果

从上述例子中可以看出，Java 程序是由类构成的，对于一个应用程序来说，必须有一个类中定义 main()方法；而对 applet 来说，它必须作为 Applet 的一个子类。在类的定义中，应包含类变量的声明和类中方法的实现。Java 在基本数据类型、运算符、表达式、控制语句等方面与 C 和 C++语言基本上是相同的，但它同时也增加了一些新的内容，在以后的各章中将详细介绍。本章中的介绍只是让大家对 Java 程序有一个初步的了解。

第2章

基本数据类型、操作符和表达式

数据类型、运算符和表达式是编程语言的基本元素，详细了解这些元素是掌握编程语言的基石。

本章的主要内容包括:

- 基本数据类型
- 运算符和表达式

2.1 基本数据类型

在 Java 编程语言中，主要有两种类型的数据：基本类型和引用类型。相应地，有两种类型的变量，它们既可以作为参数传递，也可以作为方法的返回值。另外，还有一种 null 类型，没有名字，因此不能声明 null 类型的变量，它通常被表达式用来描述空类型。对于那些熟悉编程，特别是用 C 和 C++语言编程的人来说，本章的内容是他们非常熟悉的，除了 Java 在一些地方更偏重于面向对象的思想外，几乎所有的基本内容都与 C++语言相同或是类似。

2.1.1 综述

基本数据类型是 Java 编程语言预先定义的、长度固定的、不能再分的类型，数据类型的名字被当作关键字保留。

Java 有 8 个基本类型（primitive type），其中 6 个是数值类型（4 个整数类型和 2 个浮点数类型）。在剩下的 2 个类型中，一个是字符类型（char），用来表示 Unicode 编码字符；另一个是 Boolean 类型，用来表示逻辑真和逻辑假，如图 2.1 所示。

表 2.1　　Java 的基本数据类型

类　型	描　述	初 始 值
byte	8 位带符号的整数，可表示的数的范围为-128～127	(byte)0
short	16 位带符号的整数，可表示的数的范围为-32768～32767	(float)0
int	32 位有符号的整数，可表示的数的范围为-2147483648～2147483647	0
long	64 位带符号的整数，其值为-9223372036854775808～9223372036854775807	0L
float	32 位单精度浮点数，使用 IEEE754－1985 标准	0.0f
double	64 位单精度浮点数，使用 IEEE754－1985 标准	0.0d
Boolean	只有两个值：真（true）与假（false）	false
char	16 位字符，其 ASCII 码的最高位为 0，它所表示的数字是无符号的 16 位值，在 0～65535 之间	'\u0000'

这些基本数据类型是 Java 所独有的，虽然其他编程语言也会有如表 2.1 所示的数据类型，但是，它们依赖于所使用的计算机所能支持的类型，而 Java 则不依赖于计算机对数据类型的支持，因此 Java 代码的可移植性很好。

2.1.2 整数类型

整型数据是最普通的数据类型，它的表现方式有：十进制、十六进制和八进制。基数为 10 的十进制与我们平时习惯使用的整数表示一样，以 10 为基准，数字只能包括 0～9 和符号；基数为 16 的十六进制表示法，也是程序设计中常常用到的，在十六进制中，每一位数字相当

于 4 个二进制 0 和 1 的组合，十六进制整数用 0～9 的 10 个数字和字母 A～F 代表 10～15 这些整数，十六进制整数必须以 0X 或者 0x 作为开头，可以包含 0～9、字母 A～F 和 a～f，以及符号，例如 0x23，0xab3c，0xff00 等；八进制整数以 8 为基准，数字只能包括 0～7 和符号，而且必须以 0 为前导，八进制常数通常用来进行位操作，机器中每一位八进制数字用三位二进制数表示。

int 整型数据占有 32 位的存储空间，即四个字节。这意味着 int 整型数据所表示的范围在 -2147483647～2147483648 之间，假如由于某些原因必须表示一个更大的数，64 位的长整型应该是足够的。

long

I

351 -12345689238971

2.1.3 浮点型

浮点数据用来代表一个带小数的十进制数，例如 1.5 或 43.7。它有如下几部分：10 进制整数、小数点、10 进制小数、指数和正负符号。它或者是标准形式，或者是用科学计数法的形式，下面是一些例子。

3.1415 0.1 .6 .02243 2.997E8

Java 有两种浮点数类型，即单精度浮点数和双精度浮点数。单精度浮点数的存储空间为 32 位，也就是 4 个字节；双精度浮点数的存储空间为 64 位。Java 通过在浮点数后面加描述符的方法来指明这两种浮点数。例如：

1.5f 或 1.5F 或 1.5	单精度浮点数
1.5d 或 1.5D	双精度浮点数

如果一个浮点数没有特别指明后缀，则为单精度浮点数。

2.1.4 布尔型

布尔类型是最简单的一种数据类型，布尔数据只有两种状态：真和假。通常用关键字 true 和 false 来表示这两种状态。与 C 语言或 C++语言不同的是，Java 的布尔类型只能是真（true）或假（false），不能代表整数。对 C 或 C++语言熟悉的人对这种严格的限制会觉得似乎有些不方便，但是，这样却避免了含糊的定义，减少了因查找这类错误而花费的大量时间。

改变布尔变量可以直接赋值，如：MyBoolean = true;

也可以利用其他变量赋值，例如，如果希望 YourBoolean 与 MyBoolean 的值相同，则：

```
YourBoolean = MyBoolean;
```

另外，还可以使用等式：

```
MyBoolean = 3 > 2;
```

2.1.5 字符型

字符型数据是由一对单引号括起来的单个字符，如：'a'，'b'。Java 语言使用的是 16 位的

Unicode 字符集，它不仅包括标准的 ASCII 字符集，还包括许多其他系统（如 Unix）通用的字符集。下面是字符的一些例子：

'x'，'6'，'+'，'M'，'%'

一些字符，例如空格，很难当作一个字符来使用，这类字符有特殊的意义，称为“转义符”（escape characters），引用方法为“\”加上特定的字符序列，如表 2.2 所示。

表 2.2　　Java 的转义字符

转义序列	含　义
\n	回车（\u000a）
\t	水平制表符（\u0009）
\b	空格（\u0008）
\r	换行（\u000d）
\f	换页（\u000c）
\'	单引号（\u0027）
\"	双引号（\u0022）
\\	反斜杠（\u005c）
\ddd	ddd 为三位八进制，ddd 值在 000～0377 之间，例如\007 代表 beep
\dddd	dddd 为四位 16 进制数

字符串数据类型是用一对双引号括起来的字符序列。需要指出的是，Java 中字符串数据实际上是由 String 类所实现，而不是 C 语言中所用的字符数组。我们将在第 5 章中详细介绍字符串。

2.1.6　数值类型之间的相互转换

Java 在进行异种类型的乘法运算时不会有什么问题，例如用一个整数乘以一个双精度数，结果会当作一个双精度数。更为常见的是，不同类型的数值进行二进制运算也是允许的，而且会根据下面规则加以对待：

① 如果其中一个运算对象是 double 类型，另一个也会转变为 double 类型；

② 否则，如果其中一个运算对象是 float 类型，另一个也会转变为 float 类型；

③ 否则，如果其中一个运算对象是 long 类型，另一个也会转变为 long 类型。

同样的道理也适用于整数类型：int、short 和 byte。

另一方面，有时候会遇到需要把 double 数作为整数处理的情况。尽管 Java 可以实现数值的相互转换，但是某些信息也可能丢失。记住整型变量和字符型变量位长不同是非常重要的，整型变量的位长是 32 位，字符型变量的位长是 16 位，所以当整型变量转换成字符型变量可能会丢失信息。同样，当把 64 位的长整型数转换为整型数时，由于长整型数可能有比 32 位更多的信息，也很可能会丢失信息。例如：

```
double x = 9.997;
int nx = (int)x;   //把 double 类型强制转换为 int 类型
```

然后，变量 nx 的值就变成了 9，因为将浮点数转换为整数后，会丢失其中的小数部分信息。

如果想把浮点数舍入成“最接近”的整数，则需要使用 Math.round()方法：

```
double x = 9.997
int nx = (int)Math.round(9.997)
```

现在 nx 的值变成 10 了。调用 round 方法时，仍需要使用强制类型转换（int），因为 round 方法返回值是一个 long 类型，而 long 类型只能通过强制类型转换赋给一个 int 类型变量，而其中存在着信息丢失的可能。

32

Java C/ C++

当进行类型转换时要注意使目标类型能够容纳原类型的所有信息，允许的转换包括：

byte -> short -> int -> long -> float -> double，以及 char -> int

如上所示，把位于左边的一种类型的变量赋给位于右边的类型的变量不会丢失信息。需要说明的是，当执行一个这里并未列出的类型转换时可能并不总会丢失信息，但进行一个理论上并不安全的转换总是很危险的。

2.2 运算符和表达式

2.2.1 综述

运算符，有时也称为操作符，用于对数据进行计算和处理，或改变特定对象的值。运算符按其操作数的数目来分，可分为一元运算符（如++，—）二元运算符(如+，>)和三元运算符(如? :)。按照运算符对数据的操作结果分类，运算符可以分为下述几种：

（1）算术运算符（++，-，*，/，%，++，--）；

（2）关系运算符（>，<，>=，<=，==，!=）；

（3）布尔逻辑运算符（!，&&，||）；

（4）位运算符（>>，<<，>>>，&，|，^，～）；

（5）赋值运算符（=，及其扩展赋值运算符如：+=）；

（6）条件运算符（？：）；

（7）其他运算符包括分量运算符“.”下标运算符“[]”，实例运算符“instance of”，内存分配运算符“new”，强制类型转换运算符（类型），以及方法调用运算符“()”等。

表达式是运算符、常量和变量的组合。Java 的表达式既可以单独组成语句，又可以出现于循环条件测试、变量说明、方法的调用参数等场合。一个表达式包含有一个或多个操作，操作的对象称作运算元，而操作本身是通过运算符来体现的。

2.2.2 算术运算符

算术运算符作用于整型数据或浮点型数据，用来完成算术运算。算术运算符按其操作数

的多少可分为一元算术运算符和二元算术运算符两类，一元算术运算符运算一次只对一个变量进行操作，二元算术运算符运算一次对两个变量进行操作。对于运算来说，如果有一个变量或操作数是长整型的，那么结果就肯定是长整型的，否则即使操作数没有确定是字节型（byte 型）、短整型或字符型，运算结果都是整型。

1

一元取反运算符（-）用来改变整数的正负号，加加运算符（++）和减减运算符（—）用来把变量的值加 1 或减 1。一元算术运算符如表 2.3 所示。

表 2.3　　一元算术运算符

运 算 符	用　法	描　述
+	+op	正值
-	-op	负值
++	++op，op++	加 1
—	—op，op—	减 1

注意“++”和“—”的使用，每次遇到“++”或“—”的时候，系统就把操作数的值相应地加 1 或减 1，++和—既是前置运算符也是后置运算符，这就是说，它们既可以放在操作数前面（++x），也可以放在操作数的后面（x++），如果它们被用在复合语句中，如：

i=x++；或 i=++x；

那么第一个语句中 *x* 把值赋给 *i* 以后再加 1，而第二个语句是先把 *x* 加 1，再把新的 *x* 值赋给 *i*。在这一点上，熟悉 C++语言的人应该很了解++和—的使用方法。

2

二元算术运算符并不改变操作数的值，而是返回一个必须赋给变量的值。二元算术运算符如表 2.4 所示。

表 2.4　　二元算术运算符

运 算 符	用　法	描　述
+	op1+op2	加
-	op1-op2	减
*	op1*op2	乘
/	op1/op2	除
%	op1%op2	取模(求余)

Java 对加运算符进行了扩展，使它能够进行字符串的连接，如“abc” + “de”，得到串“abcde”。

与 C、C++语言不同的是，在 Java 中，取模运算符“%”的操作数可以为浮点数，如 37.2%10=7.2。

例 2.1 说明了算术运算符的使用。

【例 2.1】

```
/*
  * Java 的算术运算符的使用
  */
public class operatorSample1
{
        public static void main( String args[])
        {
                int     a=8;                //a=8
                int     b=a*3;              //b=24
                int     c=b/4;              //c=6
                int     d=b-c;              //d=18
                int     e=-d;               //e=-18
                int     f=e%4;              //f=-2
                double          g=21.5;
                double          h=g%4;          //h=1.5
                int      i=3;
                int      j=i++;             //i=4，j=3
                int      k=++i;            //i=5，k=5
                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);
                System.out.println("f = "+f);
                System.out.println("g = "+g);
                System.out.println("h = "+h);
                System.out.println("I = "+i);
                System.out.println("j = "+j);
                System.out.println("k = "+k);
        }
}
```

其结果为：

```
C:\>java operatorSample1
    a=8
    b=24
    c=6
    d=18
```

e = -18

f = -2

g = 21.5

h = 1.5

i = 5

j = 3

k = 5

2.2.3 关系运算符

关系运算符用来比较两个值之间的关系，返回布尔类型的值 true 或 false。关系运算符都是二元运算符，如表 2.5 所示。

表 2.5 关系运算符

运 算 符	用 法	返回 true 的情况
>	op1>op2	op1 大于 op2
>=	op1>=op2	Op1 大于或等于 op2
<	op1<op2	op1 小于 op2
<=	op1<=op2	Op1 小于或等于 op2
==	op1==op2	op1 与 op2 相等
!=	op1!=op2	op1 与 op2 不等

与 C 和 C++不同的是，Java 中，任何数据类型的数据(包括基本类型和组合类型)都可以通过==或!=来比较是否相等。关系运算的结果返回 true 或 false，而 C 和 C++中关系运算的结果返回 1 或 0。

关系运算符常与布尔逻辑运算符一起使用，作为流控制语句的判断条件，如：

```
if(a>b&&b==c)
```

2.2.4 布尔逻辑运算符

布尔逻辑运算符进行布尔逻辑运算，用来连接关系表达式，如表 2.6 所示。

表 2.6 布尔逻辑运算符

op1	op2	Op1&&op2	op1||op2	!op1
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

&&、|| 为二元运算符，分别用来实现逻辑与、逻辑或。

！为一元运算符，用来实现逻辑非。

对于布尔逻辑运算，先求出运算符左边的表达式的值，若或运算左边表达式的值为 true，则不必对运算符右边的表达式再进行运算，整个表达式的结果为 true；若与运算左边表达式的值为 false，则不必对右边的表达式求值，整个表达式的结果为 false。

例 2.2 介绍了关系运算符和布尔逻辑运算符的使用方法。

【例 2.2】

```
/*
   * 关系运算符和布尔逻辑运算符的使用
   */
public class operatorSample2
{
    public static void main(String args[])
    {
        int a=25,b=35;
        boolean d=a<b;
        System.out.println("a<b="+d);
        int e=3;
        if(e!=0&&a/e>5)
            System.out.println("a/e="+a/e);
        int f = 0;
        if(f!=0&&a/f>5)
            System.out.println("a/f="+a/f);
        else
            System.out.println("f="+f);
    }
    }
```

其运行结果为：

```
C:\>java operatorSample2
    a<b=true
    a/e=8
    f=0
```

if　　0　　f!=0

false　　a/f

2.2.5　位运算符

在解释位运算符之前，需要先说明补码的概念。Java 使用补码来表示二进制数，在补码

表示中，最高位为符号位，正数的符号位为 0，负数为 1。补码的规则如下：

- 对正数来说，最高位为 0，其余各位代表数值本身(以二进制表示)，如+42 的补码为 00101010；
- 对负数而言，把该数绝对值的补码按位取反，然后对整个数加 1，即得该数的补码，如-42 的补码为 11010110 (00101010 按位取反，结果是 11010101 +1 11010110)；
- 用补码来表示数，0 的补码是惟一的，都为 00000000（而在原码和反码表示中，+0 和-0 的表示是不以惟一的）。而且可以用 111111 表示—1 的补码（这也是补码与原码和反码的区别）。

位运算符对整型数中的位进行测试、置位或移位处理，位运算符是对数据进行按位维护的手段。Java 中提供了如表 2.7 所示的位运算符。

表 2.7　　位运算符

运 算 符	用　法	描　述
~	~ op	按位取反
&	op1 & op2	按位与
|	op1 | op2	按位或
^	op1 ^ op2	按位异或
>>	op1 >> op2	op1 右移 op2 位
<<	op1 << op2	op1 左移 op2 位
>>>	op1 >>> op2	op1 无符号右移 op2 位

位运算符中，除 ~ 以外，其余均为二元运算符。

1

~是一元运算法，它对数据的二进制格式进行按位取反，即把 1 变为 0，把 0 变为 1。例如：~0010101 = 1101010

－　　21?-21

2　　&

参与运算的两个值，如果两个相应位都为 1，则按位“与”的结果为 1，否则为 0。即：0 & 0 = 0，0 &1 = 0，1 & 0 = 0，1 & 1 = 1。

例如：00101010 & 00010111 = 00000010

按位“与”可以用来对某些特定的位清零，例如对数 11010110 第 2 位和第 5 位清零，可将该数与 11101101 进行按位与运算：

11010110 & 11101101 = 11000100

按位“与”还可以用来取某个数中某些指定的位。例如：要取数 11010110 的第 2 位和第

5 位，则可将该数与 00010010 进行按位与运算：

11010110 & 00010010 = 00010010

3 |

参与运算的两个值，只要两个相应位中有一个为 1，则按位“或”的结果为 1。即 0 | 0 = 0，0 | 1 = 1，1 | 0 = 1，1 | 1 = 1。

例如：00101010 | 00010111 = 00111111

按位“或”可以用来把某些特定的位置 1，例如：对数 11010110 的第 4、5 位置 1，可将该数与 00011000 进行按位或运算：

11010110 | 00011000 = 11011110

4

参与运算的两个值，如果两个相应位相同，则按位“异或”的结果为 0，否则为 1。即 0 ^ 0 = 0，1 ^ 0 = 1，0 ^ 1 = 1，1 ^ 1 = 0。

例如： 00101010 ^ 00010111 = 00111101

按位“异或”可以用来使某些特定的位翻转，例如对数 11010110 的第 4、5 翻转，可以将该数与 00011000 进行按位“异或”运算：

11010110 ^ 00011000 = 11001110

通过按位“异或”运算，还可以实现两个值的交换，而不必使用临时变量，例如，交换两个整数 a、b 的值，可通过下列语句来实现：

a = 11010110，b = 01011001

a = a ^ b; // a = 10001111

b = b ^ a; // b = 11010110

a = a ^ b; // a = 01011001

5 <<

该运算符用来将一个数的各二进位全部左移若干位。例如 a = a<<2，使 a 的各二进制位左移两位，右补 0；若 a = 00001111，则 a<<2 = 00111100。高位左移后溢出，舍弃不起作用。

在不产生溢出的情况下，左移一位相当于乘 2，而且用左移来实现乘法比乘法运算速度要快。

6 >>

该运算符用来将一个数的各二进制位全部右移若干位。例如 a = a>>2，使 a 的各二进制位右移两位，移到右端的低位被舍弃，最高位则移入原来高位的值。例如：a = 00110111，则 a>>2=00001101，b = 11010011，则 b>>2 = 11110100。

右移一位相当于除 2 取商，而且用右移来实现除法比除法运算速度要快。

7 >>>

该运算符用来将一个数的各二进制位无符号右移若干位，与右移运算符>>相同，移出的低位被舍弃；但不同的是最高位补 0。例如 a = 00110111，则 a>>>2 = 00001101；b = 11010011，则 b>>>2=00110100。

8

如果两个数据长度不同(如 byte 型和 int 型)，对它们进行位运算时，例如：a & b，而 a

为 byte 型，b 为 int 型 ，则系统首先会将 a 的左侧 24 位填满：若 a 为正，则填满 0；若 a 为负，则左侧填满 1。

9

位运算可以用来处理一组布尔标志。如果一个程序中有几个不同的布尔标志分别代表对象的几个性质的状态，则可以把它们放在同一个变量中，通过对该变量进行位操作来实现对各布尔变量的访问。例 2.3 说明了这一过程。

【例 2.3】

```
/*
   * 位运算符的使用的例子
   */
public class operatorSample3
{
    static String binary[]={"0000","0001","0010","0011",
                            "0100","0101","0110","0111",
                            "1000","1001","1010","1011",
                            "1100","1101","1110","1111"};
    static final int FLAG1=1; //make FLAG1 a constant(0x0001)
    static final int FLAG2=2; //make FLAG2 a constant(0x0010)
    static final int FLAG4=8; //make FLAG4 a constant(0x1000)
    public static void main( String args[] )
    {
        int flags=0; //clear all flags
        System.out.println("Clear all flags... flags="+binary[flags]);
        flags=flags | FLAG4; //set flag4
        System.out.println("Set flag4... flags="+binary[flags]);
        flags=flags ^ FLAG1; //revert flag1
        System.out.println("Revert flag1... flags="+binary[flags]);
        flags=flags ^ FLAG2; //revert flag2
        System.out.println("Revert flag2... flags="+binary[flags]);
        int cf1=~FLAG1;
        flags=flags & cf1; //clear flag1
        System.out.println("Clear flag1... flags="+binary[flags]);
        int f4=flags & FLAG4;
        f4=f4>>>3; //get flag4
        System.out.println("Get flag4... flag4="+f4);
        int f1=flags & FLAG1; //get flag1
        System.out.println("Get flag1... flag1="+f1);
    }
}
```

其结果为：

```
C:\>java opratorSample3
Clear all flags... flags=0000
Set flag4... flags=1000
Revert flag1... flags=1001
Revert flag2... flags=1011
Clear flag1... flags=1010
Get flag4... flag4=1
Get flag1... flag1=0
```

2.2.6 赋值运算符

赋值运算符是二元运算符，赋值运算符左边的操作数必须是变量，右边的操作数必须是表达式，所有赋值操作符的优先级都相同。与其他运算符相比，赋值运算符的优先级最低，且具有右结合特性。

1

= 是最简单的赋值操作符，格式为：

变量名 = 表达式；

先计算表达式，然后再将结果赋给变量。使用赋值运算符时，尽量使变量和表达式的数据类型一致；否则，应先将表达式的值转换为变量的数据类型，然后再进行赋值。

2

复合赋值运算符是在基本赋值运算符的基础上加上另一个运算符。由于“=”的右结合特性，因此先计算右边表达式的值，再与左边的变量进行相应的操作，最后把操作的结果赋给左边的变量。当变量以前的值决定用户要赋的值的时候，使用复合赋值运算符使得程序简洁，而且效率更高。下面是一些常用的复合赋值运算符：

+=	加赋值运算符
-=	减赋值运算符
*=	乘赋值运算符
/=	除赋值运算符
%=	取模赋值运算符

例 2.4 是赋值运算符的一个例子。

【例 2.4】

```
/*
   * 赋值运算符的例子
   */
public class operatorSample4
{
   public static void main(String args[])
   {
      byte a = 60;
```

```
        short b = 4;
        int c =30;
        long d = 4L;
        long result = 0L;

        result += a – 8;
        System.out.println("result += a – 8 :    "    + result);
        Result *= b;
        System.out.println("result *= b :      " + result);
        result /= d + 1;
        System.out.println("result /= d + 1 :    "    + result);
        result -= c;
        System.out.println("result -= c :    "    + result);
        result %= d;
        System.out.println("result %= d :    "    + result);
          }
}
```

运行结果为：

```
result += a-8 :52
result *= b    :208
result /= d + 1    :41
result -= c    :11
result %= d :3
```

2.2.7 条件运算符

这是一种最独特的逻辑运算符，在 C 和 C++语言中也经常用到，其格式为：

条件 ？ 表达式 1 ： 表达式 2

例 2.5 为条件运算符的一个例子。如果条件的布尔值为真（true），则执行表达式 1，否则执行表达式 2。

【例 2.5】

```
/*
   * 条件运算符的例子
   */
public class operatorSample5
{
      public static void main(String args[])
      {
          int x = 0;
          Boolean isFalse = false;
```

```
        System.out.println("x = "+ x);
        x = isFalse? 4 : 7;
            System.out.println(" x = "+ x);
        }
}
```

运行结果为：

x = 0

x = 7

2.2.8 表达式与运算符优先级

简单地说，表达式就是操作数和运算符连接在一起的结果，通常用于对变量或值进行操作。表达式是变量、常量、运算符、方法调用的序列，它执行这些元素指定的计算并返回给某个值，如 a+b，f+d 等都是表达式。表达式用于计算、对变量赋值，并作为程序控制的条件。当表达式中有两个或多个运算符时，该表达式称为复杂表达式，其运算符执行的先后顺序由复杂表达式中各运算符的优先级和接合性决定。最简单的表达式只有一个常量或者变量，没有运算符。

表达式在运算时完成一个或多个操作，运算后，最终得到一个结果，且结果的数据类型由参加运算的数据的类型来决定。在对一个表达式进行运算时，要按照运算符的优先级顺序从高到低进行运算，运算符的优先级是指同一个表达式中多个运算符被执行的次序。Java 运算符的优先级与 C++语言的几乎完全一样，圆括号的优先级最高，赋值运算符的优先级最低。在运算过程中，优先级高的运算符先运算，优先级低的运算符后运算，可以使用圆括号改变运算的次序。Java 中运算符的优先级如表 2.8 所示，按照表从上到下的顺序优先级逐渐降低，同一行中的优先级相同。

表 2.8　　运算符的优先级

描　述	运 算 符
最高优先级	.[]()
一元	~!++--
乘、除、取模	*/%
加减	+-
移位	<<>>>>>
关系	<<=>=>
等于或不等于	==!=
按位与	&
按位异或	^
按位或	|

续表

描　述	运 算 符
条件与	&&
条件或	||
条件	?:
赋值	= 算术运算符=

另外，运算符还具有结合特性，即左结合特性（从左至右）和右结合特性（从右至左）。对左结合特性很好理解，表达式计算从左至右计算。对于右结合特性，例 2.6 中，赋值运算符的使用就是一个很好的例子。

【例 2.6】

```
/*
   * 运算符优先级的例子
   */
public class operatorSample6
{
    public static void mian(String args[])
    {
        int a, b;
        b = 5;
        a = b =10;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
          }
}
```

程序运行输出的结果为：

```
a = 10
b = 10
```

当计算 $a = b = 10$ 时，对于赋值运算符，先执行 $b = 10$，再执行 $a = b$，因此，a 和 b 的值都是 10。

第 3 章

程序的流程控制

流程控制构成了编程语言的逻辑，而对这些控制语句的灵活运用又能有助于编程逻辑的清晰整理。

本章的主要内容包括:

- 条件
- 循环
- 跳转

3.1 条件

程序是由多个语句组成的，Java 支持条件语句、循环语句和跳转语句，以便进行程序流控制。下面分别来介绍 Java 中的控制语句。

3.1.1 if 语句

Java 语言的 if 语句是最简单的流程控制形式。if 语句是一个条件表达式，如果条件表达式的值为真，则继续执行下面的代码块，否则跳过这个代码块，其语法为：

```
if（表达式）
    语句;
```

但 Java 语言和其他大多数的编程语言一样，在某个条件为真的情况下，经常都会执行多条语句，这时需要把这些代码放到同一个大括号中，其语法为：

```
if（表达式）
    {代码块}
```

【例 3.1】

```
/*
   * ifSample.java
   * if 条件语句的例子
   */
public class ifSample
{
     public static void main(String[] args)
     {
          int testscore = 76;
          char grade = 'B';
          if (testscore >= 90
          {
               grade = 'A';
               System.out.println("Grade = "+grade);
          }
          System.out.println("Grad = "+ grade);
}
```

运行结果为：

```
Grade = B
```

3.1.2 if...else 语句

if...else 语句根据判定条件的真假来执行两种操作中的一种，其语法为：

```
if（表达式）
```

```
    if 的语句;
else
    else 的语句
```

当 if 表达式的求值为假时，就会转向 else 部分去执行，两部分的代码只能有一部分被执行，当然，if...else 语句还可以嵌套使用，如：

```
if(firstValue == 0)
{
if(sencondValue == 1)
    Value++;
}
else
    Value—;
```

【例 3.2】

```
/*
  * ifelseSample.java
  *if...else...语句的例子
  */
import java.util.*;
public class ifelseSample{
    public static void main(String args[])
    {
        Calendar day = new GregorianCalendar();
        int today;

        today = day.get(Calendar.DAY_OF_WEEK);

        if(today == 0)
        System.out.println("Today is Monday");
        else if(today == 1)
        System.out.println("Today is Tuesday");
        else if(today == 2)
        System.out.println("Today is Wednesday");
        else if(today == 3)
        System.out.println("Today is Thursday");
        else if(today == 4)
        System.out.println("Today is Friday");
        else if(today == 5)
        System.out.println("Today is Saturday");
        else
        System.out.println("Today is Sunday");
```

```
        }
    }
```

程序中 day 是个日历实例，它的方法 get(Calendar.DAY_OF_WEEK)返回当天的星期数，if…else 语句根据这个星期数来输出字符串。

3.1.3 switch 语句

必须在多个备选方案中处理多项选择时，再用 if…else 结构就显得很繁琐。这时可以使用 switch 语句来实现同样的功能。switch 语句基于一个表达式条件来执行多个分支语句中的一个，它是一个不需要布尔求值的流程控制语句。switch 语句的一般格式如下：

```
switch(表达式)
{
    case 值 1：语句 1；
    break;
    case 值 2：语句 2；
     break;
     …
case 值 N：语句 N；
 break;
[default：上面情况都不符合情况下执行的语句；]
}
```

具体说明如下所述。

（1）表达式可以返回任意一个简单类型的值(如整型、字符型)，多分支语句把表达式返回的值与每个 case 子句中的值相比，如果匹配成功则执行该 case 子句后的语句序列。

（2）case 子句中的值必须是常量，而且所有 case 子句中的值是不同的。

（3）default 子句是任选的，当表达式的值与任意一个 case 子句中的值都不匹配时，程序执行 default 后面的语句。如果表达式的值与任意一个 case 子句中的值都不匹配且没有 default 子句时，则程序不做任何操作，并直接跳出 switch 语句。

（4）break 语句用来在执行完一个 case 分支后，使程序跳出 switch 语句，即终止 switch 语句的执行。由于 case 子句只是起到一个标号的作用，用来查找匹配的入口并从此处开始执行，它对后面的 case 子句不再进行匹配，而是直接执行其后面的语句序列，因此应该在每个 case 分支后，用 break 来终止后面的 case 分支语句的执行。


```
                                    case
        break
```

（5）case 分支中包括多个执行语句时，可以不用大括号“{}”括起来。

（6）switch 语句的功能可以用 if…else 来实现，但在某些情况下，使用 switch 语句更简炼、可读性强，而且可以提高程序的执行效率。

switch

• switch if switch

if

• switch case

3.2 循环

3.2.1 for 语句

for 语句是 Java 语言中最有用的循环结构，它循环执行下面的语句或者语句块，直到表达式的值为假。其语法为：

```
for（初始化；表达式；变化的步长）
{
    语句；
}
```

具体说明如下所述。

（1）for 语句执行时，首先执行初始化操作，然后判断终止条件是否满足。如果满足，则执行循环体中的语句，最后执行迭代部分。完成一次循环后，重新判断终止条件。

（2）可以在 for 语句的初始化部分声明一个变量，它的作用域为整个 for 语句。

（3）for 语句通常用来执行循环次数确定的情况(如对数组元素执行操作)，也可以根据循环结束条件执行循环次数不确定的情况。

（4）在初始化部分和迭代部分可以使用逗号语句，来进行多个操作。逗号语句是用逗号分隔的语句序列。例如：

```
for(i=0，j=10；i<j；i++，j--)
{
…
}
```

（5）初始化、终止和迭代部分都可以为空语句(但分号不能省)，三者均为空的时候，相当于一个无限循环。

C for C

for

Java Java C for

Java for

Java for

Java for

for

3.2.2 while 语句

while 语句测试一个表达式，如果表达式的值为真，则会重复执行下面的语句或语句块，直到表达式的值为假。其语法为：

```
while（表达式）
{
    语句;
}
```

例 3.3 演示了如何使用 while 语句。

【例 3.3】

```
/*
 * whileSample.java
 * while 语句的例子
 */
public class whileSample{
public static void main(String[] args)
{
    byte a[]=new byte[200];
    int length=0;

    try{
        length=System.in.read(a);
    }
    catch(Exception  e)
    {
        System.out.println(e.toString());
    }

    while(a[0]!='q')
    {
        System.out.println("length:" + (length-2));
        try{
                length=System.in.read(a);
```

```
            }
            catch(Exception   e)
            {
                System.out.println(e.toString());
            }
        }
    }
}
```

程序从标准输入读入字符串，然后输出字符串长度，如此循环，直到该字符串的第一个字符为'q'时退出。

3.2.3 do...while 语句

while 循环从顶部开始测试，因此，这个块中的代码可能永远也得不到执行。如果想让一个块至少执行一次，则需要从底部开始测试。do...while 语句可以实现“直到型”循环，它的一般格式为：

```
do
  {
    语句;
  } while（表达式）;
```

例 3.4 演示了如何使用 do...while 循环来实现例 3.3 相同的功能。

【例 3.4】

```
/*
  * dowhileSample.java
  * do...while 语句的例子
  */
public class dowhileSample
{

public static void main(String[] args)
{
        byte a[]=new byte[200];
        int length=0;

        try{
            length=System.in.read(a);
        }
        catch(Exception   e)
        {
          System.out.println(e.toString());
```

```
        }

        do
        {
            System.out.println("length:" + (length-2));

          try{
            length=System.in.read(a);
          }
          catch(Exception  e)
          {
              System.out.println(e.toString());
           }
        }while(a[0]!='q');
    }
}
```

该程序和例 3.3 惟一不同的是，由于至少会执行一次，第一次输入'q'打头的字符串时程序不会退出，而且会照例显示字符串长度。

3.3 跳转

3.3.1 break 语句

循环中的字语句块和 switch 语句均可以使用 break 来终止，break 会将控制传给当前（最内）循环语句后的第一个语句，这是 break 语句的第一个作用。break 语句的另外一个作用就是和标号一起实现 goto 语句的作用。

在 Java 中，可以为每个代码块加一个括号，一个代码块通常是用大括号｛｝括起来的一段代码。加标号的格式如下：

BlockLabel：{codeBlock}

然后在程序中可以使用 break 语句跳到它所指定的块，并从紧跟该块的第一条语句处执行。其格式为：

break　BlockLabel；

例如：

```
a：{…//标记代码块 a
    b：{…//标记代码块 b
        c：{…//标记代码块 c
              break      b；
              …//不会被执行
            }
```

```
        ...//不会被执行
    }
    ...//从这里开始执行
}
```

C C++ Java goto
Java
goto
goto Donald Knuth
goto goto
Java break goto

3.3.2 continue 语句

continue 语句只能出现在循环语句（while，do...while，for）的字语句块中，无标号的 continue 语句的作用是跳过当前循环的剩余语句块，直接执行下一个循环。对于 for 语句，在进行终止条件的判断前，还需要先执行迭代语句。也可以用 continue 跳转到括号指明的外层循环中，这时的格式为：

```
continue  outerLable;
```

例如：

```
outer: for(int i= 0;I < 10;i++){ //外层循环
    for(int j = 0;j < 20; j++){ //内层循环
        if(j>i){
            ...
            continue outer;
        }
        ...
    }
    ...
}
```

本例中，当满足 j>i 的条件时，程序执行完相应的语句后跳转到外层循环，执行外层循环的迭代语句 i++；然后开始下一次循环。

3.3.3 return 语句

return 语句的作用是执行程序从当前方法或程序中退出，返回到调用该方法的语句处，并从紧跟该语句的下一条语句继续程序的执行。返回语句有下述两种格式。

1　return expression

返回一个值给调用该方法的语句，返回值的数据类型必须和方法声明中的返回值类型一致。可以使用强制类型转换来使类型一致。

2　return

当方法说明中用 void 声明返回类型为空时，应使用这种格式，它不返回任何值。

return 语句通常用在一个方法体的最后，以退出该方法并返回一个值。Java 中，单独的 return 语句用在一个方法体的中间时，会产生编译错误，因为这时会有一些语句执行不到。但可以通过把 return 语句嵌入某些语句(如 if…else)来使程序在未执行完方法中的所有语句时退出。例如：

```
int   method(int   num){
      //return   num;          //will cause compile time error
      if(num>0)
          return num;
          …   //may or may not be executed，depending on the value of num
```

第4章

类、接口和包

类的概念是所有面向对象编程语言的基础，而 Java 的接口概念使得面向对象编程更加完善。

本章的主要内容包括:

- 面向对象的编程基础
- 类
- 接口
- 包

4.1 面向对象的编程基础

Java是一种面向对象的程序设计语言，学习Java编程，首先要了解面向对象程序设计的基本概念。面向对象程序设计（Object—Oriented Programming，OOP）是目前占据主流地位的一种先进的程序设计方法，它取代了早先所谓“结构化”的、以“过程（procedure）”为基础的编程技术，其编程思想是将事物抽象为对象，由于对象具有自己的状态和行为，Java语言便通过对象对消息的反应来完成一定的任务。Java语言支持面向对象的3种重要技术：封装（encapsulation）、继承（inheritance）和多态（polymorphism）。Java为了语言实现上的简单性，它不支持多重继承，但通过接口实现了多重继承的主要功能，避免了逻辑上的混乱。

4.1.1 对象（object）的概念

对象是面向对象技术的一个重要的概念。现实生活中任何物体都是对象，例如人、动植物、计算机以及从最简单的整数到复杂的航天飞机等等，都是对象。另外，一些看不见的事物、规格也是客观存在的对象。

任何一个物体都包括两个基本特点，一个是物体的内部构成（或属性），例如汽车的轮子、门和发动机等；另一个是物体的行为（或方式），即对该物体内部构成成分的操作或与外界信息的交换，例如汽车的发动、行驶和停车开门等。

面向对象编程（OOP）技术用对象表示现实中的物体，并与物体的两个基本特点相对应。一个对象就是一组成员变量（data）和相关的方法（method）的集合，其中成员变量表明对象的状态，方法表明对象所具有的行为。一个对象的变量构成这个对象的核心，包围在它外面的方法使这个对象和其他对象分离开来。面向对象的程序设计实现了对对象的封装，实现了模块化和信息隐藏，有利于程序的可移植性和安全性，同时也有利于对复杂对象的管理。

对象之间必须通过交互来实现复杂的行为。例如，要使汽车加速，必须发给它一个消息，告诉它进行何种动作（这里是加速），以及实现这种动作所需的参数(这里是需要达到的速度等)。一个消息包含三个方面的内容：消息的接收者、接收对象应采用的方法以及方法所需要的参数。

同时，接收消息的对象在执行相应的方法后，可能会给发送消息的对象返回一些信息。(如上例中，汽车的仪表上会出现已经达到的速度等)。由于任何一个对象的所有行为都可以用方法来描述，通过消息机制就可以完全实现对象之间的交互；同时，处于不同处理过程甚至不同主机的对象间也可以通过消息实现交互。

上面所说的对象是一个具体的事物（例如，每辆汽车都是一个不同的对象），但是多个对象常常具有一些共性（例如，所有的汽车都有轮子、方向盘、刹车装置等），于是，我们抽象出一类对象的共性，这就是类。类中定义一类对象共有的变量和方法。把一个类实例化即生成该类的一个对象（比如，我们可以定义一个汽车类来描述所有汽车的共性）。通过类的定义，人们可以实现代码的复用。我们不用去描述每一个对象(如某辆汽车)，而是通过创建类(如汽车类)的一个实例来创建该类的一个对象，这就大大减化了软件的设计。利用OOP编程，根据需要先定义类，指出类的名称、变量和方法，确定类成员被访问的权限等；其次，利用类创建相应的实例（或对象）；最后，在程序运行时，由系统根据需要与对象交换信息等。

4.1.2 类的封装

封装（encapsulation）是把类（或对象）的基本成分（数据和方法）封装在类体（或对象体）之中，使之与外界分隔开。

封装技术来源于现实生活中的许多实例。例如，计算机由许多零件组成，其零配件构造和运行过程非常复杂，这些都由计算机制造厂商进行负责，并把它们封装在机箱之内与外界分隔，只提供用户使用的计算机的接口，用户不能直接使用计算机箱内的各种构件，只能通过键盘、鼠标以及各种软件来操纵计算机。

OOP 模仿现实生活中的封装技术，把类的数据和方法封装在类对象体内。通常，将某些数据和方法说明为私有的，它的实现细节由软件开发者负责，用户不必了解这些细节，也不能直接访问这些数据。同时，还可以将一些数据和方法说明为公共的，可以作为外界访问类对象的接口，用户通过接口和类对象交换信息。

类对象的封装方法隐藏了其内部信息的细节，使内部信息不易受外界破坏，同时也为外界访问它提供了简单方便的接口，使类具有模块化的性质。

4.1.3 类的继承

通过对象、类实现了封装，通过了类可以实现继承。对于上例来说，公共汽车、出租车、货车等都是汽车，但它们是不同的汽车，除了具有汽车的共性外，它们还具有自己的特点(如不同的操作方法、不同的用途等)。这时，可以把它们作为汽车的子类来实现，它们继承父类(汽车)的所有状态和行为，同时增加自己的状态和行为。

类继承是指一个新的类继承原有类的基本特性。原有的类称为父类、超类（superclass）或基类，新的类称为原来类的子类或派生类。在子类中，既包含有父类的属性（数据）和方法，还可以增加新的属性和功能。这种子类继承父类的方法也被称为类的派生（deriving）。通过父类和子类，实现了类的层次，子类还可以派生新的子类，可以从最一般的类开始，逐步特殊化，定义一系列的子类。

类的继承提高了程序的可复用性，使程序的复杂性呈线性增长，而不是呈几何级数增长。根据类的继承机制，在父类中只定义了各层子类都需要的属性和功能，类派生时，只增加新的属性和功能即可，因此，父类的基本特征可被所有子类的对象共享，提高了类的重复利用率，这样，也极大地简化了软件的开发，有利于提高软件开发的效率。

C++

(

) Java Java

4.1.4 类的多态性

类的多态性（polymophism）是指一个名称具有多种功能，或者相同的接口有多重实现

方法。Java 编程语言通过方法的重载（overloading）、覆盖（overriding）和接口（Interface）来实现多态。

- 方法重载是指多个方法具有相同的名称，但各个方法的参数个数和参数类型不相同，根据不同的参数个数和类型来选择执行不同的方法。例如，对一个作图的类，它有一个 draw( ) 方法用来画图或输出文字，我们可以传递给它一个字符串、一个矩形、一个圆形，甚至还可以再指定作图的初始位置、图形的颜色等，对于每一种实现，只需实现一个新的 draw( )方法即可，而不需要新起一个名字，这大大简化了方法的实现和调用，程序员和用户都不需要记住很多的方法名，只需要传入相应的参数即可。
- 覆盖是指类派生过程中，子类与父类的方法不仅名称相同，参数也完全相同，但它们的功能不同，这时子类中的方法覆盖了父类中同名的方法，使其具有自己的特征。例如，对于汽车类的加速方法，其子类（如赛车）中可能增加了一些新的部件来改善和提高加速性能，这时可以在赛车类中重载父类的加速方法。重载隐藏了父类的方法，使了类拥有自己的具体实现，更进一步表明了与父类相比子类所具有的特殊性。
- 接口实际上是一种特殊的类，其中只有方法的原型，即只给出方法的名称、参数和返回值的类型，没有方法体。这些方法的实现在其子类中具体定义。

类的多态性使方法的调用更加容易、灵活和方便。

4.2 类

类是 Java 中的一种重要的复合数据类型，也是组成 Java 程序的基本要素。它封装了一类对象的变量和方法，是这一类对象的原型。创建一个新的类就是创建一个新的数据类型，实例化一个类就得到一个对象。在前面的例子中，我们已经定义了一些简单的类，如 HelloworldApp 类。

```
public class HelloWorldApp{
    public static void main( String args[ ] ){
        System.out.println("Hello World !");
    }
}
```

可以看出，一个类的实现包含两部分的内容：

```
classDeclaration {
    classBody
}
```

下面我们分别对每一部分做详细讲述。

4.2.1 类的声明

类声明的完整格式为：

[类修饰符] class 新类名 [extends 超类名] [implement 接口名]

其中，“[ ]”内的内容为可选项。

1

类声明中的修饰符决定了类在程序中被处理的方式。创建类时，可以接受缺省的修饰符，也可以根据需要制定一个或多个修饰符，这些修饰符可以是一个访问修饰符（public，protected，private），加上一个或多个类型修饰符（abstract，static，final，strictfp）。

（1）public

把一个类声明为公有的（public），类的成员可以被任何对象存取，这就意味着它可以被任何对象使用或扩展，无需考虑它所包含的包（package）。public 类必须在 .java 文件中定义。

（2）protected

限定类的成员仅仅可以被该类及其派生类访问。

（3）private

限定类的成员仅能被该类的其他成员使用，对于一些不需要外界知道的数据或方法，可以将它们声明为 private 类，这有利于数据的安全，同时也符合程序设计中隐藏内部信息处理细节的原则。

（4）缺省的访问修饰符

如果没有指定访问修饰符时，那么，这个类可以被其他类引用和扩展，但只有在相同包中的那些对象才可以使用这个类。

（5）abstract

abstract 类是一个未完成的（或不完全）的类，仅仅 abstract 类含有 abstract 方法，之所以称其为 abstract，就是因为在类中仅仅声明方法，而不实现方法。如果非 abstract 类含有 abstract 方法，则编译时会提示出错。abstract 类在下面三种情况下使用：

① 类中含有 abstract 方法；

② 类的超类声明了 abstract 方法，在当前类中没有实现；

③ 类 implement 的接口中声明或继承了一个方法，在类中或者没有继承，或者没有实现。

下面介绍的例 4.1 中，类 Point 必须声明为 abstract 类，因为它含有声明为 abstract 类的方法 alert。类 Point 的子类 ColorPoint 继承了 alert 的方法，但是没有实现，所以必须声明为 abstract 类；对于类 Point 的另一个子类 SimplePoint，由于实现了 alert 方法（虽然没有实际的代码），因此无需将其声明为 abstract 类。

【例 4.1】

```
/*
  * 抽象类的例子
  */
abstract class Point
{
    int x = 1, y = 1;
    void move (int dx, int dy)
    {
        x += dx;
        y += dy;
        alert();
```

```
    }
    abstract void alert();
}
abstract class ColoredPoint extends Point
{
    int color;
}
class SimplePoint extends Point
{
    void alert() { }
}
```

abstract　　abstract

如果：Point p = new Point();，则编译时将出错，但是，可以实例化 SimplePoint 类：

```
Point p = new SimplePoint();
```

（6）final

一个类被声明为 final 类，表明它的定义是完整的，而且不可以有任何子类。因此，同时声明一个类为 abstract 和 final 是明显错误的，因为这样的类永远都不可能完成。

final

例 4.2 是 java.lang.Math 类的一个局部，它是一个 final 类。

【例 4.2】

```
public final strictfp class Math
{
    private Math() {}  //任何人都不能初始化该方法
    public static final double E = 2.7182818284590452354;
    public static final double PI = 3.14159265358979323846;
    public static double sin (double a ) {
        return StrictMath.sin(a);
    }
    public static double cos(double a ) {
        return StrictMath.cos(a);
    }
...
}
```

（7）strictfp

strictfp 类的作用是使得类中所有的 float 和 double 表达式都是显式的满足 FP-strict，这意味着类中声明的所有方法和所有嵌套类型，都隐含地声明为 strictfp。

所谓 FP-strict，是针对 float 和 double 的表达式而言，表达式的所有中间值都必须是 float 或 double，并且表达式的操作符都在 IEEE754 中定义。

在 Java 类的声明中，经常会使用到 strictfp，例如：

```
public final strictfp class Math
```

2

与 Java 的其他标识符一样，类名有如下设置：

（1）以字母、字符“_”或“$”开头。

（2）只能含有大于十六进制 00C0 以上的 Unicode 字符。

（3）不能使用与 Java 关键字相同的类名。

（4）类名通常以大写字母开头，如果类名由多个单词组成，则每一个单词的开头字母也大写。

3

面向对象编程的最重要的特色之一就是能够使用以前创建的类的方法和域。通过简单的类来创建功能强大的类，从而大幅地节省编程时间；更重要的是，这样做可以减少代码出错的机会。

在 Java 中，所有的类都是通过直接或间接地继承 java.lang.Object 得到的。继承而得到的类为子类，所继承的类为父类，父类包括所有直接或间接被继承的类。子类继承父类的状态和行为，同时也可以修改父类的状态或重写父类的行为，并添加新的状态和行为，Java 中不支持多重继承。下面介绍在 Java 语言中如何实现继承。

（1）创建子类

通过在类的声明中加入 extends 子句来创建一个类的子类，其格式如下：

```
class 子类 extends 父类 {
    ...
}
```

把一个子类声明为父类的直接子类，子类可以继承所有父类的内容。如果缺省 extends 子句，则该类为 java.lang.Object 的子类。

子类可以继承父类中访问权限设定为 public、protected、“friendly”的成员变量和方法。但是不能继承访问权限为 private 的成员变量和方法。

（2）成员变量的隐藏和方法的重写

在子类中，重写的方法和父类中被重写的方法具有相同的名字、相同的参数表和相同的返回类型。子类通过成员变量的隐藏和方法的重写可以把父类的状态和行为改变为自身的状态和行为。

子类在隐藏了父类的成员变量或重写了父类的方法后，常常还要用到父类的成员变量，或在重写的方法中使用父类中被重写的方法以简化代码的编写，这时就要访问父类的成员变量或调用父类的方法，Java 中通过 super 来实现对父类成员的访问。

在 Java 中，this 用来引用当前对象，super 用来引用当前对象的父类。例 4.3 演示了这两个关键字的使用。

【例 4.3】

```
/*
  * thissuperSample.java
  * 演示 this 和 super 关键字的使用
  */
import java.awt.*;
public class thissuperSample extends Frame
{
    int b;
    public thissuperSample(String a)
    {
        this(a,0);
    }
    public thissuperSample(String a,int b)
    {
        super(a);
        this.b = b;
    }
}
```

4.2.2 类的成员变量

类的属性或者成员变量声明的完整格式为：

[修饰符] 类型 变量 [=初始值][变量];

其中，“[]”内的内容为可选项，初始值可以为一个表达式或者对象。

成员变量的类型可以是 Java 中的任意数据类型（包括简单类型、数组、类和接口）。在一个类中，成员变量应该是惟一的，但是成员变量的名字可以和类中某个方法的名字相同，例如：

```
class Point{
    int x,y;
    int x( ){
        return x;
    }
}
```

其中，方法 x( )和变量 x 具有相同的名字。

类的成员变量和在方法中所声明的局部变量是不同的，成员变量的作用域是整个类，而局部变量的作用域只是方法内部。

修饰符中同样含有前面已经介绍过的三个访问修饰符 public、protected 和 private。另外，还有 static、final、transient 和 volatile。同样的，如果在一个变量声明中相同修饰符出现不止一次或者出现多个访问修饰符，编译时都会提示出错。

（1）static

static 变量又称为类变量，如果变量不被声明为 static，则称为实例变量。如果类中的某个变量被声明为 static，那么，无论最终会创建多少个该类的实例，这些类实例中的 static 变量都只存在一个相同的副本。也就是说，这些类实例中的其中一个类的实例的 static 型的变量发生改变，其他实例中对应的 static 变量也具有相同的值。

【例 4.4】

```
/*
  * staticSample.java
  * static 类型变量的使用
  */
class intergerSample
{
    static int x;
    static int y;
    public int x() {
        return x;
    }
    public void setX(int newX){
        x = newX;
    }
    public int y() {
        return y;
    }
    public void setY(int newY){
        y = newY;
    }
}
class staticSample
{
   public static void main(String[] args)
   {
       intergerSample myXY = new intergerSample ();
       intergerSample anotherXY = new intergerSample ();
       myXY.setX(1);
       anotherXY.x = 2;
       myXY.setY(1);
       anotherXY.y = 2;
       System.out.println("myXY.x = "+ myXY.x());
       System.out.println("anotherXY.x = "+ anotherXY.x());
       System.out.println("myXY.y = "+ myXY.y());
```

```
        System.out.println(" anotherXY.y = " + anotherXY.y());
    }
}
```

程序的运行结果为：

```
myXY.x = 2
anotherXY.x = 2
myXY.y = 2
anotherXY.y = 2
```

（2）final

用来声明一个常量。对于用 final 声明的常量，在程序中不能改变它的值。通常常量名用大写字母表示。例如，下例中声明了常量 CONSTANT，并赋值为 50。

```
class FinalVar{
    final int CONSTANT = 50;
    …
}
```

（3）transient

用来声明一个暂时性变量。在缺省情况下，类中所有变量都是对象永久状态的一部分，当对象被存档时，这些变量必须同时被保存。用 transient 限定的变量则指示 Java 虚拟机，该变量并不属于对象的永久状态。例如，对于例 4.5，如果需要将类 Point 的实例对象存储到磁盘，那么仅仅存储 x 和 y，而不存储 temp01 和 temp02。

【例 4.5】

```
class Point
{
    int x ,y ;
    transient float temp01, temp02;
}
```

（4）volatile

声明一个共享变量。在 Java 编程语言中，允许访问共享变量的线程拥有该变量的一个私有的工作副本，这在多线程的环境中更加高效。这些工作副本需要在指定的同步点与共享主内存中的主副本同步（在对象加锁或解锁时）。为了保证共享变量能够一致和可靠地更新，线程应该保证以得到锁的方式来独占地使用这类变量。

当变量声明为 volatile 时，线程每一次访问该变量时，必须使工作副本和主副本同步，使得各个线程对该变量的访问能保持一致。

4.2.3 方法

方法声明了可以被调用的代码，传递固定数量的参数。方法的实现包括了两部分内容：方法声明和方法体，格式如下：

方法修饰符 结果类型 方法名（[参数列表]）{方法体}[throws 子句]

其中，返回类型可以是任何有效的 Java 数据类型，当一个方法不需要返回任何值时，返回类

型为 void；参数列表可以为空，也可以是以逗号隔开的、任何有效的数据类型的一些变量声明，以为方法的实现提供信息。

参数的类型可以是简单数据类型，也可以是引用数据类型（数组类型、类或接口）。对于简单数据类型，Java 实现的是值传送，方法接收参数的值，但并不能改变这些参数的值。如果要改变参数的值，则要用引用数据类型，因为引用数据类型传递给方法的是数据在内存中的地址，方法中对数据的操作可以改变数据的值。与其他一些编程语言不同的是，Java 语言不支持将方法作为参数传递给另一个方法，它只支持将对象传递给方法，然后，再调用对象上的方法。

有关 throws 子句的内容将在第 5 章介绍。

1

方法修饰符除了前面已经介绍的访问修饰符 public、protected 和 private 之外，还有 abstract、static、final、synchronized、native 和 strictfp。

如果在一个方法声明中含有不止一个访问修饰符，或者声明为 abstract 的方法还含有关键字 private、static、final、synchronized、native 和 strictfp 之一，则编译时都会出错。另外，声明的方法中同时含有 native 和 strictfp 关键字也会造成编译时的错误。

（1）abstract 方法

abstract 方法声明了一个方法（方法名、参数），返回值和 throws 子句，但是没有提供方法的实现，即方法体为空。abstract 方法必须出现在 abstract 类中，否则编译时会出错。

（2）static 方法

声明为 static 的方法被称为类方法（class method），类方法总是在不引用特定对象的情况下被调用，使用关键字 this 或 super 引用当前对象会导致编译时出错。

例如，在 Math 类中，含有许多数学计算方法（例如，指数、对数和三角函数等），使用时我们不可以使用 Math()方法来实例化 Math 类。对于 static 方法，只需要简单地将它们当作已经实例化的对象来对待，具体使用方法见例 4.6。

【例 4.6】

```
import java.lang.Math;
public class finalSample
{
    public static void main(String args[])
    {
        double myAngle = 2.319
        System.out.println("sin("+ myAngle + ") = " + Math.sin(myAngle));
    }
}
```

（3）final 方法

将方法声明为 final 可以防止其子类重载或隐藏该方法。Private 方法和 final 类中声明的所有方法都可以认为被隐含地声明为 final，因为不可能再重载它们。

（4）native 方法

native 方法用来把 Java 代码和其他语言的代码集成起来。native 方法是用依赖于平台的代码实现的。通常，native 方法体仅仅给出了一个分号，体被忽略。例如，在包 java.io 的

RandomAccessFile 类中，声明了 native 方法。

（5）strictfp 方法

strictfp 方法中，所有的 float 和 double 表达式都严格地遵守 FP-strict 的限制。

（6）synchronized 方法

synchronized 方法用来控制多个并发线程对共享数据的访问，在其执行之前要求一个锁。对于类方法，方法的类使用与类对象相联系的锁；对于实例变量，该变量使用与 this（调用方法的对象）相联系的锁。

【例 4.7】

```
/*
  * synchronSample.java
  * 演示 synchronize 方法的使用
  */
class synchronSample
{
    static int i = 0, j = 0;
    static synchronized void one()
    {
        i ++;
        j ++;
    }
    static synchronized void two()
    {
       if ( i < j )
            System.out.println("i = "+ i + ", j = "+ j);
    }
}
```

2

方法体是对方法的实现。它包括局部变量的声明以及所有合法的 Java 指令。方法体或者是一些代码块，或者是一个简单的分号（当方法声明为 abstract 或 native 时）。如果方法没有声明为 abstract 或 native，但方法体是一个分号，在编译时同样也会提示出错。

方法体中可以声明该方法中所用到的局部变量，它的作用域只在该方法内部，当方法返回时，局部变量也不再存在。如果局部变量的名字和类的成员变量的名字相同，则类的成员变量被隐藏。

对于一个方法，如果在声明中所指定的返回类型不为 void，则在方法体中必须包含 retrun 语句，返回指定类型的值。返回值的数据类型必须和声明中的返回类型一致，或者完全相同，或者是它的一个子类。当返回只是接口时，返回的数据类型必须实现该接口。

（1）方法的继承

在理解了面向对象编程的概念后，方法的继承也就很好理解了，方法的继承就是继承父类中所声明的、所有非 private 的方法。使用时必须注意方法的访问权限问题。

（2）方法的重载

方法重载是指多个方法有同样的名字，但是这些方法的参数必须不同，或者参数的个数不同，或者参数的类型不同。编译器根据参数的个数和类型来决定当前所使用的方法。

方法重载为程序调试带来方便，是 OOP 多态性的具体表现。在 Java 系统的类库中，对许多重要的方法进行了重载，这为用户使用这些方法提供方便。例如，System.out.println，它含有许多重载的类型，用户可以很方便地输出各种类型的数据，下面列出的就是重载 println 方法的类型。

```
public void println()
public void println(boolean x)
public void println(char x)
public void println(int x)
public void println(long x)
public void println(float x)
public void println(double x)
public void println(char x[])
public void println(String x)
public void println(Object x)
```

（3）方法的隐藏

如果类声明为 static 方法，则该方法的声明隐藏了父类和父接口中所有具有相同署名的方法。但是，如果 static 方法隐藏了实例方法，则会发生一个编译错误。对于被隐藏的方法而言，可以使用 super 关键字来访问。

（4）方法的继承与重载的限制

如果一个方法声明或者重载了另外一个方法的声明，则这两个方法的返回类型必须一致，而且这两个方法的 throws 子句也必须相同，否则，编译时会出错。

重载或隐藏方法的访问修饰符必须至少提供与其重载或隐藏方法一样的访问权限，否则，编译时会提示出错。

public

pubic

protected

public protected

private

由于 private 方法不可能被隐藏或重载，因此子类可以声明与其父类中该方法署名一模一样的方法，并且 throws 子句也可以不同。

4.2.4 类的构造

1

如果在每创建一个实例时都要初始化类中的所有变量就显得太繁琐了，因此，在创建对象时就对对象进行初始化是一种简单而有效的解决方法。在 Java 语言中，人们定义了一个特殊的方法，叫构造方法，用来初始化对象以使对象在创建后可以立即使用。构造方法具有和类名相同的名称，而且不返回任何数据类型，甚至 void 也不返回，这是因为构造方法的缺省返回类型是类的类型本身。例如，对于 Point 类，其构造方法可以是：

```
class Point
 {
   int x,y
   Point (int x,int y)
   {
        this.x = x;
        this.y = y;
   }
}
```

一旦定义了构造方法，在对象创建后和 new 操作完成之前，将自动调用构造方法。用构造方法进行初始化，避免了在生成对象后每次都要调用对象的初始化方法。如果没有实现类的构造方法，则 Java 运行时系统会自动提供缺省的构造方法，它没有任何参数，也不做任何事情。因此，如果我们希望执行某些初始化时，应该编写一些构造方法，完成所需的初始化工作。

Java

【例 4.8】

```
class Point
{
   int x,y
   Point (int x,int y)
   {
        this.x = x;
```

```
        this.y = y;
    }
    int x,y
    Point (){
        this.x = -1;
        this.y = -1;
     }
}
class PointCreate
{
    public static void main(String args[])
    {
        Point p = new Point();
        System.out.println(" x = " + p.x + " y = " + p.y);
    }
}
```

程序输出结果：

x = -1 y = -1

2

类的实例其实也就是类的对象，通过把类实例化，可以生成多个对象。创建它们的方法非常简单，格式为：

类 实例名 = 类的构造方法（[变量列表]）

new 运算符为对象分配内存空间，创建一个类的实例，调用该类的构造方法并返回对该对象的一个引用（即该对象所在的内存地址）。下面的例子是创建一个 Point 的 new 实例并保存到变量 p 中。

```
Point p = new Point();
```

这里 *p* 是 Point 实例的一个引用。在 Java 中，简单类型和对象在处理上有很大的不同。在上面的例子中，变量 *p* 值是对象的引用，它并不包含实际的对象。你可以为同一对象创建多个引用。下面的例子创建一个 Point 对象，但有两个变量引用它。

```
Point p1 = new Point ();
Point p2 = p1;
```

通过 p2 引用而改变了对象同样会影响 p1 所引用的同一对象，两者指的是同一块内存地址。

用 new 可以为一个类实例化多个不同的对象。这些对象占用不同的内存空间，因此改变其中一个对象的状态，不会影响到其他对象的状态。请看例 4.9。

【例 4.9】

```
/*
 * TwoPoints.java
 * 类的实例的例子
 */
```

```
class TwoPoints
{
    public static void main(String args[])
    {
        Point p1 = new Point();
        Point p2 = new Point();
        p1.x = 1;
        p1.y = 2;
        p2.x = 3;
        p2.y = 4;
        system.out.println("p1.x = " + p1.x + "    p1.y = " + p1.y);
        system.out.println("p2.x = " + p2.x + "    p2.y = " + p2.y);
    }
}
```

程序的输出结果是：

```
p1.x = 1 p1.y = 2
p2.x = 3 p2.y = 4
```

C　C++　Java

3

Java 运行时，系统通过垃圾收集周期性地释放无用对象所使用的内存，完成对象的清除。当不存在对一个对象的引用(当前的代码段不属于对象的作用域或把对象的引用赋值为 null，如 p=null)时，该对象成为一个无用对象。Java 的垃圾收集器自动扫描对象的动态内存区，对被引用的对象加标记，然后把没有引用的对象作为垃圾收集起来并释放。

垃圾收集器作为一个线程运行。当系统的内存用尽或程序中调用 system.gc()要求进行垃圾收集时，垃圾收集线程与系统同步运行。否则垃圾收集器在系统空闲时异步地执行。

在 C 语言中，通过 free 来释放内存，在 C++语言中则通过 delete 来释放内存，这种内存管理方法需要跟踪内存的使用情况，不仅复杂而且还容易造成系统的崩溃；Java 采用自动垃圾收集进行内存管理，使程序员不需要跟踪每个生成的对象，避免了上述问题的产生，这是 Java 的一大优点。

在对对象进行垃圾收集前，Java 运行时系统会自动调用对象的 finialize()方法来释放系统资源，如打开的文件或 socket。该方法的声明如下：

```
protected void finalize(){
    //cleanup
} throws Throwable
```

finalize()方法在类 java.lang.Object 中实现。如果要在一个自定义的类中实现该方法以释放该类所占用的资源（即要重写父类的 finalize()方法），则在对该类所使用的资源进行释放后，要调用父类的 finalize()方法以清除对象使用的所有资源，包括由于继承关系而获得的资源。

4.2.5 类的访问

类的访问包括引用类对象的成员变量和方法，通过运算符“.”，可以实现对变量的访问和方法的调用，对变量和方法可以通过设定一定的访问权限来允许或禁止其他对象对它的访问。

1

要访问类对象的某个变量，其格式为：

objectReference.variable

其中 objectReference 是类对象的一个引用，它可以是一个已经生成的类实例，也可以是能够生成类实例的表达式。

2

类的实例通过“.”运算符调用某个方法，其格式为：

objectReference.methodName([参数列表]);

其中 objectReference 是一个对象的引用变量，methodName 是由 objectReference 声明的类对象中的方法名。

4.2.6 嵌套类

在 Java 中，可以将一个类定义为另一个类的成员，这样的类被称为嵌套类（或内部类），如下例中类 ANestedClass 就是一个嵌套类，类 EnclosingClass 称为类 ANestedClass 的封装类：

```
class EnclosingClass {
        ...
    class ANestedClass {
        ...
    }
}
```

private

像其他类一样，嵌套类同样可以声明为 static，即静态嵌套类（static nested class）；而非静态的嵌套类被称为内部类（inner class）。在下例中，类 AStaticNestedClass 为静态的嵌套类，类 InnerClass 为内部类。

```
class EnclosingClass
{
```

```
    ...
    static class AStaticNestedClass
    {
        ...
    }
    class   InnerClass
    {
        ...
    }
}
```

与静态方法和变量相似，静态嵌套类总是与其封装类相联系，但不能直接引用其封装类中的实例变量或方法，仅仅可以通过对象来引用。另外，由于内部类与一个实例相联系，因此，不能定义任何 static 成员。

4.3 接口

接口(Interface)在语法上与类非常类似，但是没有实例变量和实现的方法。这就是说，我们可以定义接口而不必关心实现的细节。Java 通过接口使得处于不同层次甚至互不相关的类也可以具有相同的行为。换句话说，通过在接口中放置一些方法，可以用接口大致规划出类的共同行为，而把某些具体的实现留给各个具体类。接口就是方法定义和常量值的集合，它的用处主要体现在下面几个方面：

- 通过接口可以实现不相关类的相同行为，而不需要考虑这些类之间的层次关系；
- 通过接口可以指明多个类需要实现的方法；
- 通过接口可以了解对象的交互界面，而不需了解对象所对应的类。

Java 中使用接口的一个经典例子是 java.applet.AppletContext 接口。该接口定义了一个方法集，这些方法会根据 Applet 的运行环境返回信息。例如，AppletContext 定义了一个叫做 getImage 的方法，它使任何能够运行 Applet 的浏览器都可以下载图片。但问题是，对于不同的浏览器，例如 Microsoft Internet Explorer 和 Netscape Navigator，它们的工作方式是不同的，而且，即使是相同的浏览器，在不同平台上工作也可能是不同的。但幸运的是，每一个浏览器都实现了 AppletContext 接口，而 java.applet.AppletContext 类要依据在 AppletContext 接口中声明的方法工作，类无需考虑这些方法是如何工作的，也就是说，可以在不同的环境和不同的浏览器中使用相同的 applet 类和相同的方法。

当进行高级数据处理时，使用接口比使用类好得多。由于某些原因，与 C++语言不同的是，Java 语言不支持类的多重继承，而是用接口实现比多重继承更强的功能。多重继承指一个类可以是多个类的子类，它使得类的层次关系不清楚，而且当多个父类同时拥有相同的成员变量和方法时，子类的行为是不容易确定的，这给编程带来了困难。单一继承则清楚地表明了类的层次关系，指明了子类和父类各自的行为。接口则把方法的定义和类的层次区分开来，通过，程序可以在运行时动态地定位所调用的方法；同时，接口中可以实现“多重继承”，且

一个类可以实现多个接口。正是这些机制使得接口提供了比多重继承更简单、更灵活而且更强劲的功能。

static final

4.3.1 接口的定义

接口的定义包括接口声明和接口体两部分，

```
[public] interface 接口声明 [extends superInterfaceList]
{
   //接口体
    [type constName = Value;
   returnType methodName ([ParamList]);]
}
```

其中 [] 中的部分为可省略的，即最简单的接口声明如下：

```
interface 接口名{...}
```

1 public

public 指明任意类均可以使用这个接口，缺省情况下，只有与该接口定义在同一个包中的类才可以访问这个接口。

2

接口名的定义规则与类名的定义规则相同，要求名字的开头必须是字母、下划线或美元符号，且只能包含 Unicode 字符，不能与 Java 的关键字相同。接口通常以 able 或 ible 结尾，表明接口能完成一定的行为。

3

extends 子句与类声明中的 extends 子句基本相同，所不同的是，一个接口可以有多个父接口，用逗号隔开，而一个类只能有一个父类。子接口继承父接口中所有的常量和方法。

扩展其他接口时需要遵循的一条主要规则是：它们不可以定义方法的体，只能定义它们自己的方法，任何实现新接口的实现类必须定义父接口和子接口中所有方法的体。接口不能扩展类，因为接口所要扩展的类中一定会包含有方法体，而这恰好违反了接口的基本规定。类实现接口以继承它们的属性，而接口通过扩展其他的接口来继承属性。

4

接口体中包含常量定义和方法定义两部分。

常量定义的格式为：

```
type constantNAME = value;
```

其中 type 可以是任意类型，NAME 是常量名，通常用大写，value 是常量值。在接口中定义

的常量可以被实现该接口的多个类共享，它与 C 语言中用 #define 以及 C++语言中用 const 定义的常量是相同的。在接口中定义的常量具有 public, final ,static 的属性。

方法定义的格式为：

returnType methodName ([paramlist])；

接口中只进行方法的声明，而不提供方法的实现，所以方法定义没有方法体，而且用分号结尾。在接口中声明的方法具有 public 和 abstract 属性。另外，如果在子接口中定义了和父接口同名的常量或相同的方法，则父接口中的常量被隐藏，方法被重载。

下面给出了一个接口的定义：

```
interface Collection {
    int MAX_NUM=100;
    void add (Object obj);
    void delete (Object obj);
    Object find (Object obj);
    int currentCount ( );
}
```

该例定义了一个名为 Collection 的接口，其中声明了一个常量和 4 个方法。这个接口可以由队列、堆栈、链表等类来实现。

4.3.2 接口的实现

在类的声明中用 implements 子句来表示一个类使用某个接口，在类体中可以使用接口中定义的常量，而且必须实现接口中定义的所有方法。一个类可以实现多个接口，在 implements 子句中用逗号分隔。

例 4.10 讲解在类 queueSample 中实现上面所定义的接口 Collection。

【例 4.10】

```
class queueSample implements collection
{
    void add ( Object obj )
    {
        …
    }
    void delete( Object obj )
    {
        …
    }
    Object find( Object obj )
    {
        …
    }
    int currentCount
```

```
    {
        …
    }
}
```

在类中实现接口所定义的方法时，方法的声明必须与接口中所定义的完全一致。

4.3.3 接口类型

接口可以作为一种引用类型来使用。任何实现该接口的类的实例都可以存储在该接口类型的变量中，通过这些变量可以访问类所实现的接口中的方法。Java 运行时系统动态地确定应该使用哪个类中的方法。

把接口作为一种数据类型可以不需要了解对象所对应的具体的类，而着重于它的交互界面，以前面所定义的接口 Collection 和实现该接口的类 queueSample 为例，在例 4.11 中，以 Collection 作为引用类型来使用。

【例 4.11】

```
class InterfaceType
{
    public static void main( String args[] )
    {
        Collection c = new queueSample();
        …
        add( obj );
        …
    }
}
```

4.4 包

由于 Java 编译器为每个类生成一个字节码文件，且文件名与类名相同，因此同名的类有可能发生冲突。大量的、杂乱无章的类文件和接口文件放在一起时，就像一个不带子目录和文件夹的硬盘一样，所有的文件都放在一个文件夹中，极其混乱。为了解决这一问题，Java 提供包（Package）来管理类名空间，我们可以把每一个包看成一个子目录。Java 中的包是相关类的集合，例如，在一个称为 Transportation 的包中，包含有 car、boat、AirPlane、train 等类，很容易把 Transportation 包中的各个类引用到应用程序中去。包实际上提供了一种命名机制和可见性限制机制。

包在较大范围内实现了 OOP 的封装机制，它把一组类和接口封装在一个包之内，这为管理大量的类和接口提供了方便，也有利于这些类和接口的安全。

Java 编译器将包与文件系统的目录一一对应起来。例如，在名称为 MyPackage 的包中，它的类文件应在 MyPackage 目录中。在包语句中，用圆点“.”指明目录的层次。例如：

Package java.lang;

指明这个包中的类文件存储在目录 java\lang 下，该目录的 HOME 目录可以在 CLASSPATH 环境变量中指定，也可以在编译或运行时使用-classpath 选项。

.class .java

4.4.1 包的声明

声明包的格式为：

package 包的名称;

声明包的语句必须放在程序的最前面，在类声明之前。也就是说，在 package 这一行前面可以有注释或空白，但是不能有其他别的东西。该语句后面的所有类都被当作包的成员。例如：

```
package myPackage;
```

另外，包还可以嵌套在其他包中。

如果 Java 源程序文件前没有包定义语句，系统就把这个文件中定义的类放在一个缺省的包中，该缺省的包对应与源程序所在的目录，并且目录名和包名的大小写必须一致，否则编译时会出错。

不同程序文件中定义的类也可以放在一个包中，只要这些程序文件前都加上包的说明语句 package 即可。在例 4.12 中，class1.java 和 class2.java 源程序中定义的类都放置在包 MyPackage 中。

【例 4.12】

```
/*
  *  演示包的使用
  */
//class1.java 文件：
package MyPackage;
class1
{
    ...
}
//class2.java 文件：
package MyPackage;
class2
{
    ...
}
```

4.4.2 导入包的类

为了能使用 Java 中已提供的类，我们需要用 import 语句来引入所需要的类。import 语句的格式为：

```
import package1[.package2…]. (classname |*);
```

其中 package1[.package2…]表明包的层次，与 package 语句相同，它对应于文件目录，classname 则指明所要引入的类，如果要从一个包中引入多个类，则可以用星号(*)来代替。例如：

```
import java.awt.*;
import java.util.Date;
```

Java 编译器为所有程序自动引入包 java.awt，因此不必用 import 语句引入它包含的所有的类。若需要使用其他包中的类，必须用 import 语句引入，但是不能用下面的语句来导入 Java 中所有的类：

```
Import java.*;
```

原因是 import 语句不支持嵌套的包。

另外，在 Java 程序中使用类的地方，都可以指明包含它的包，这时就不必用 import 语句引入该类了。但是，如果引入的几个包中包括名字相同的类，则当使用该类时，必须指明包含它的包，使编译器能够载入特定的类。例如，类 Date 包含在包 java.util 中，可以用 import 语句引入它，以实现它的子类 myDate：

```
import java.util.*;
class myDate extends Date
{
    …
}
```

也可以直接引入该类：

```
class myDate extends java.util.Date
{
    …
}
```

两者是等价的。

可以使用通配符*来导入整个包，这样就可以存取包中的每一个类，非常方便。因为这样就不必写一大堆代码，否则必须是类似下面的代码：

```
import java.awt.Color;
import java.awt.graphics;
import java.awt.Image;
import java.awt.Button;
import java.awt.Canvas;
…
```

但是，导入整个包会存在两个弊端。

（1）导入整个包时，JVM 会保存包中所有元素的名字，这必然会使用额外的存储空间来

存储这些类和方法名。在基于 Java 的小型设备或使用 J2ME 的电子产品上，运行起来就会有问题，而且会使系统的性能有所降低。

（2）当导入不在本地计算机上的整个包时，浏览器不得不等到把包中所有的类文件均从网上下载下来之后，才能继续工作。如果下载了一个具有 20 个类的包，而只使用了其中 2 个，那么会浪费大量的资源。

使用类之前并不是非要实际地导入一个类不可。

null

class1　class2　　class1

class2　　class1

4.4.3 编译和运行包

如果用 package 语句指明一个包，则包的层次结构必须与文件目录的层次相同。例如，我们在 test 目录下创建了一个名为 packTest 的类放在包 test 中，然后保存在文件 packTest.java 中。对该文件进行编译后，我们得到字节码文件 packTest.class。如果我们直接在 test 目录下运行 java packTest，则解释器返回“can't find class packTest”(找不到类 packTest)，因为这时类 packTest 处于包 test 中，对它的引用应为 test.packTest，于是我们运行 java test.packTest，但解释器仍然返回“can't find class test\packTest”(找不到类 test\packTest)。这时，我们可以查看 CLASSPATH，发现它的值为 C:\java\classes，表明 Java 解释器在当前目录和 Java 类库所在目录 c:\java\classes 查找，即在\test\test 目录下查找类 packTest，因此找不到。改正的方法可以有下述两种：

（1）在 test 的上一级目录运行：

java test.packTest 或

（2）修改 CLASSPATH，使其包括当前目录的上一级目录。

由上例可以看出，运行一个包中的类时，必须指明包含这个类的包，而且要在适当的目录下运行，同时，还要正确地设定环境变量 CLASSPATH，使解释器能够找到指定的类。

4.4.4 访问权限

前面已经解释过，对类的成员变量和方法都有访问权限的控制。由于类中封装了数据和代码，所以 Java 提供了对类成员的访问权限的控制。由于包中封装了类和其他的包，Java 中的类单元之间就可以分为下述 4 种情况。

（1）同一个包中的子类；

（2）同一个包中的非子类；

（3）不同包中的子类；

（4）不同包中的非子类。

关键字 public、private 和 protected 可以按不同的组合形成不同的访问级别，如表 4.1 所示。

表 4.1　　3 个关键字按不同的组合成的不同的访问级别

	Private	No modifier	private protected	protected	public
同一类	yes	yes	yes	yes	yes
同一包中的子类	no	yes	yes	yes	yes
同一包中的非子类	no	yes	no	yes	yes
不同包中的子类	no	no	yes	yes	yes
不同包中的非子类	no	no	no	no	yes

下面是对表 4.1 的简单概括。

（1）可以从任何位置访问 public 单元；

（2）以 private 声明的单元不能在类外被访问；

（3）如果一个单元没有使用任何访问修饰符，那么它可以被子类及同包中的其他类访问，这是缺省情况；

（4）如果希望当前包中的某些类可见，而且只允许这个类的直接子类访问，那么应该声明成 protected；

（5）如果想进一步限制为只允许子类访问，那么应该声明成 private protected。

第5章

数组、Vector 与字符串

本章介绍了 Java 中常用到又比较特别的三种数据类型：数组、向量和字符串，面向对象思想优化了它们的操作。

本章的主要内容包括:

- 数组
- 向量 Vector
- 字符串

5.1 数组

所谓数组，就是一组有序数据的集合，数组中的每个元素都具有相同的数据类型，可以用一个统一的数组名和下标来惟一地确定数组中的元素。数组是 Java 语言中的特殊类型。它们存放能通过索引来引用的一系列对象。数组类型可以是基本类型如 int、double 等，也可以是引用类型。

另外，在 Java 编程语言中，数组（array）是一个可以动态创建的对象，可以向数组分配 Object 类型的变量，在数组上可以调用 Object 类的所有方法。另外，还可以定义数组的数组，从而实现多维数组的支持。

5.1.1 数组的创建与使用

Java 的数组是用 new 创建的，当数组不用时，和其他对象一样，并不需要在程序中显式地释放，而是由 Java 的垃圾收集器自动地回收所占的空间。

和变量一样，数组必须先定义。数组的定义有两个部分：数组类型和数组的名字。数组类型是指数组中各元素的类型，它可以是任意的 Java 类型，甚至可以是数组。数组的名字必须是合法的 Java 标识符。下面是数组定义的一般形式：

数组类型 数组名 [];

或

数组类型 [] 数组名;

示例如下：

```
int i[];
char c[];
float f[][];
```

声明数组类型的变量并不实际地创建数组对象或为数组分配任何空间，为了使用数组，还需要创建数组。Java 中有两种创建数组的方法，一种是与一般的对象创建一样，采用 new 操作符，例如下面语句为数组 a 创建有 10 个元素的存储空间：

```
s = new char[10];
```

在数组创建时，并没有创建存放在数组中的对象，因此数组创建并不需要调用构造方法。在上面形式的 new 操作符中，并没有参数列表，此时被创建的数组元素将初始化为该类型的缺省值。缺省值与数据类型相匹配，数值类型为 0，布尔类型为 false，字符类型为'\u0000'，类对象为 null。也可以在声明数组时直接初始化，例如：

```
char s [] = new char [10];
char [] s = new char [10];
```

在 Java 语言中，第二种创建数组的方法是用静态的初始化器，用这种方法创建数组的例子如下：

```
int itable = {1, 2, 3, 48, 9}
```

一旦数组被创建，那么数组对象的长度就不能再改变。

数组元素由数组名和下标组成，下标的下界为 0，上界为数组元素的个数减一，当要访问数组元素时，程序员可以通过在数组名后跟一个放在方括号中的下标值来实现，例如，a[10] 有 10 个元素，其元素分别为 a[0]，a[1]，…a[9]。在 Java 语言中，一个数组的大小一般是通过访问数组的 length()方法得到，例 5.1 中演示了 length 域的使用。

【例 5.1】

```
/*
 * ArrayTest.java
 * 演示数组 length 域的使用
 */
class ArrayTest
{
    public static void main(String[] arg)
    {
        String a[] = {"First", "Second", "Third"}; //初始化数组
        for (int I = 0; i < a.length; i ++)
            System.out.println(a[i]);
    }
}
```

程序的输出结果为：

```
First
Second
Third
```

5.1.2 多维数组

与 C 和 C++语言相同，在 Java 语言中，多维数组被看作数组的数组。例如，二维数组为一个特殊的一维数组，其每个元素又是一个一维数组。下面我们主要以二维数组为例来进行说明多维数组，高维的情况是类似的。

二维数组的定义方式为：

数组类型 多维数组名[ ][ ];

例如：

```
int intArray[ ][ ];
```

与一维数组一样，这时对数组元素也没有分配内存空间，同样要使用运算符 new 来分配内存，然后才可以访问每个元素。

对高维数组来说，分配内存空间有下面几种方法。

（1）直接为每一维分配空间，如：

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

（2）从最高维开始，分别为每一维分配空间，如：

```
int a[ ][ ] = new int[2][ ];
a[0] = new int[3];
```

a[1] = new int[3];

完成(1)中相同的功能。这一点与 C 和 C++语言是不同的，在 C 和 C++语言中必须一次指明每一维的长度。

对二维数组中每个元素，引用方式为：

arrayName[index1][index2]

其中 index1、index2 为下标，可以是整型常数或表达式，如 a[2][3]等。同一维数组一样，每一维的下标都从 0 开始。

同样，多维数组的初始化也有两种方式：

（1）直接对每个元素进行赋值；

（2）在定义数组的同时进行初始化，如：int a[][] = {{2,3}, {1,5}, {3,4}};

5.2 向量 Vector

Vector（向量）和数组非常类似，但是它可以存储多个对象，并且可以用索引值来检索这些对象。数组和 Vector 的最大区别是当空间用完后 Vector 会自动增长。同时，Vector 还提供了额外的方法来增加或者删除元素，而在数组中，必须用手工来完成这些工作。

5.2.1 创建 Vector

Vector 有 3 种构造方法。

① public Vector();

创建一个空 Vector。

② public Vector(int initialcapacity);

创建一个 Vector，其初始大小为 initialCapacity。

③ pubic Vector（int initialCapacity, int capacityIncrement）;

创建一个 Vector，其初始大小为 initialCapacity，当 Vector 需要增长时，其增长速度由 capacityIncrement 决定。

Vector Vector
Vector

5.2.2 访问和查找 Vector 中的对象

访问 Vector 中的对象与访问数组中的元素不同，它不使用由中括号 [] 括起来的索引值，而是使用 elementAt 方法来访问 Vector 中的元素。elementAt 的方法为：

public final synchronized Object elementAt(int index)

throws ArrayIndexOutOfBoundsException;

例如，对于数组元素 someArray [4]，其等价的 Vector 是 someVector.elementAt(4)。

另外，还可以使用 firstElement 和 lastElement 方法来访问 Vector 中的第一个和最后一个元素。也可以用 isEmpty 方法检查一个 Vector 是否为空。

虽然可以通过使用枚举和元素的逐个比较方法来手工搜索 Vector 中的对象，但是使用内置的搜索函数可以节省很多时间。Vector 中有以下几种查找函数。

① public final boolean contains(Object ob);

查找一个对象是否在一个 Vector 中，只要出现一次 Ob 指定的对象就返回 true，否则返回 false。

② public final int indexOf(Object ob);

查找 Ob 在 Vector 中第一次出现的位置，找不到则返回-1。

③ public final int lastindexOf(Object ob);

返回 Vector 中对象 ob 最后一次出现的位置，否则返回-1。

④ public final synchronized int indexOf(Object ob, int stratIndex);

throws ArrayIndexOutOfBoundsException;

查找从 startIndex 位置开始，对象 Ob 在 Vector 中第一次出现的位置，没有找到则返回-1。如果 startIndex 小于 0 或者大于 Vector 的长度则抛出 ArrayIndexOutOfBoundsException 异常。

⑤ public final synchronized int lastIndexOf(Object ob, int stratIndex)

throws ArrayIndexOutOfBoundsException;

查找从 startIndex 位置开始，对象 Ob 在 Vector 中最后一次出现的位置，没有找到则返回-1。如果 startIndex 小于 0 或者大于 Vector 的长度则抛出 ArrayIndexOutOfBoundsException 异常。

5.2.3 增加和移除 Vector 的对象

向 Vector 中加入一个新对象有两种方法：

① public final synchronized void addElement(Object newElemnet);

把加入的新对象作为 Vector 中的最后一个元素。

② pubic final synchronized void insertElementAt(Object newElement, int index) throws ArrayIndexOutOfBoundsException;

把新对象插入 index 指定的位置，如果该位置不存在，则抛出一个 ArrayIndexOutOfBounds Exception 异常。

③ 另外，还可以用 setElement 方法来改变对象到新位置：

public final synchronized void setElementAt(Object ob, int index) throws ArrayIndexOutOfBoundsException;

该方法和 insertElement 方法的作用几乎是一样的，其区别在于插入的新对象代替了 Vector 中指定位置的对象。

从 Vector 中移除一个对象有 3 种方法。

① public final synchronized void removeAllElements();

移走 Vector 中所有的对象。

② public final synchronized boolean removeElement(Object ob);

移走一个特定的对象 ob，如果对象在 Vector 中出现多次，只移走第一次出现的对象。成功返回 ture，否则返回 false。

③ public final synchronized void removeElementAt(int index) throws ArrayIndexOutOfBoundsException;

移走 index 指定位置上的对象，并移动其他对象来填充移走对象产生的空隙。如果试图从一个不存在的位置上移走对象，则会抛出 ArrayIndexOutOfBoundsException 异常。

5.2.4 改变 Vector 的大小

Vector 有两种大小的概念——当前存储的元素的个数（size）和 Vector 的最大容量（capacity），capacity 方法告诉用户 Vector 能容纳多少个对象。

```
public final int capacity();
```

ensureCapacity 方法可以增加 Vector 的大小，例如：

```
public final synchronized void ensureCapacity(int minimumCapacity);
```

告诉 Vector 至少能存储 minimumCapacity 个元素，如果 Vector 的当前容量比 mininmumCapacity 的小，将分配给 Vecotr 更多的空间；如果 Vecotr 的当前空间比 minimumCapacity 的大，Vector 也不会减少当前空间。

如果要减少 Vector 的容量，可以使用 trimToSize 方法：

```
public final synchronized void trimToSize();
```

该方法将 Vecotr 的容量减少至当前存储的元素的个数。

size 方法告诉 Vector 中当前含有多少个对象：

```
public final int size();
```

可以使用 setSize 方法来改变当前元素的个数：

```
public synchronized final void setSieze(int newSize);
```

如果新尺寸小于原来的尺寸，则新尺寸之后的 Vector 中原有元素将丢失；如果新尺寸大于原来的尺寸，则新增加的元素的值将被设置为 null。

5.3 字符串

字符串是用一对双引号括起来的字符序列，在 Java 语言中，字符串数据实际上是由 String 类所实现，而不是 C 语言中所用的字符数组，每一个字符串数据将产生一个 String 类的新的实例。在学习了类和对象的概念之后，Java 的字符串也比较容易理解了。为了高效地处理字符串，Java 将字符串分为两类：一类是在程序中不会被改变长度不变的字符串；另一类是在程序中会被改变长度的可变字符串，又称为缓冲字符串。Java 环境为了存储和维护这两类字符串提供了 String 和 StringBuffer 两个类。

5.3.1 创建字符串

1 String

根据在程序中使用的方法，String 对象可以隐式或显式地创建。对于隐式地创建 String，

我们可以简单地在程序中放置一个 String 语句，Java 将自动创建 String 对象，例如：

```
g.drwaString("This is a string", 50, 50 );
```

创建 String 的另外一种方法是显式地创建 String 对象，如：

```
String str = new String("This is a string");
```

也可以先定义 String 对象，之后在程序中设置它的值：

```
String str;
Str = "This is a String";
```

或者: `String str = "This is a String";`

正是由于编译器自动从字符串常量中创建新的 String 对象，所以可以直接用字符串常量初始化 String 对象。上面是最简单的显式创建 String 对象的方法，String 类的构造方法主要有：

```
public String();
public String(String value);
public String(char value[]);
public String(char value[],int offset, int count);
public String(byte ascii[], int hibyte, int offset, int count);
public String(StringBuffer buffer);
```

其中，参数 value、value []、ascii [] 和 buffer 均为字符串的初始化值，offset 为初始偏移值，count 为长度，hibyte 为每一个 16 位 Unicode 字符的前 8 位。在这 7 个参数中，后 3 个参数主要用于从字符串的初始化值中提取字符串。

2　StringBuffer

可变字符串和不可变字符串都是对象，与一般的对象一样，它们的创建也是利用 new 操作符。StringBuffer 类的构造方法有下面 3 种。

① public StringBuffer();

创建一个空的可变字符串。

② pubic StringBuffer（int length）;

创建一个具有指定长度的空可变字符串。

③ public StringBuffer(String str);

创建一个具有指定初始值的可变字符串。

5.3.2　得到字符串对象的信息

一旦创建了字符串对象，便可以通过调用 String 或 StringBuffer 对象的方法得到有关字符串的相关信息。下面以 String 类为例来讲解。

通过调用 length()方法得到 String 的长度：

```
String str = "This is a String";
int len = str.length();
```

StringBuffer 类的 capacity()方法与 String 类的 length()方法类似，但是它测试的是分配给 StringBuffer 的内存空间的大小，而不是当前被使用了的内存空间。

如果想确定字符串中指定字符或者子字符串在给定字符串中的位置，可以用 indexOf()

方法和 lastIndexOf()方法，其中 indexOf()方法从字符串的起始位置向前搜索，而 lastIndexOf()方法则是从字符串的尾部向后搜索。

```
String str = "This is a String";
int index1 = str.indexOf('i');   //index1 = 2
int index2 = str.indexOf('i',index +1); //index2 =5, 第二个'i'的位置
int index3 = str.lastIndexOf("I"); //index3 = 13
int index4 = str.indexOf("String"); //index4 =10
```

另外，在 String 类中还提供了 startsWith（string prefix）和 ends With (string suffix)方法，用 starts With(string prefix)方法判断在字符串的开始是否含有一个前缀，用 endsWith（string suffix）方法判断 String 对象的尾部是否以给定的字符串结尾等。

5.3.3 String 对象的比较和操作

1 String

String 类的 equals()方法用来确定两个字符串是否相等。

```
String str = "This is a String";
boolean result = str.equeals("This is another String"); //result = false
```

这种比较方法是大小写敏感的，String 类还提供了一种大小写不敏感的方法来比较两个字符串。

另外，String 提供了一些方法用以访问字符串的某些部分。

① 方法 charAt()用以得到指定位置的字符。

```
Stirng str = "This is a String";
Char chr = str.charAt(3); //chr = "i"
```

② 方法 getChars()用以得到字符串中的一部分字符串。

```
public void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)
Stirng str = "This is a String";
Char chr = new char[10];
str.getChars(5,12,chr,0); //chr = "is a St";
```

③ getBytes()方法与 getChars()方法一样，但是它使用的是 byte 数组。

④ subString()是提取字符串的另一种方法，它可以指定从何处开始提取字符串以及到何处结束。

2

另外，String 类还提供了一些方法可以操作字符串，例如替换字符串、合并字符串等。

① replace()方法可以将字符串中的一个字符串替换为另一个字符，例如：

```
String str1 = "This is a String";
String str2 = str1.replace('T', 't'); //str2 = "this is a String"
```

② concat()方法可以将两个字符串合并为一个字符串，例如：

```
String str1 = "This is a String";
String str2 = str1.concat("Test"); //str2 = "This is a String Test");
```

也可以直接用“+”运算符：

```
String str1 =" This is";
String str2 = " a String";
String str = str1 + str2;
```

③ toUpperCase()和 toLowerCase()方法分别实现字符串中字符大小写之间的转换。

```
String str1 = "THIS IS A STRING";
String str2 = str1.toLowerCase(); // str2 = "this is a string"
```

④ trim()方法可以将字符串中开头和结尾处的空格去掉，例如：

```
String str1 = " This is String   ";
String str2 = str1.trim(); //str2 = "This is a String"
```

⑤ String 类提供了静态方法 valueOf()，它可以将任何类型的数据对象转换为一个字符串，如：

```
system.out.println(String.valueOf(Math.PI);
```

5.3.4 修改可变字符串

不变字符串不能改变字符串的值，在 String 类中只提供了得到字符串信息的访问方法，而 StringBuffer 类中除了提供了相应方法外还提供了修改字符串的方法。StringBuffer 类为可变字符串的修改提供了 3 种方法：在字符串后面的追加，在字符串中间插入和改变某个位置所在的字符。

1

StrngBuffer 类提供了将各种数据类型（如浮点、布尔），以及将对象添加到字符串中的方法 append()，根据参数的不同，append()方法可以将各种对象加入到字符串中。

2

StringBuffer 类还提供了在可变字符串中间插入数据的方法 insert()，例如：

```
StringBuffer str = new StringBuffer("This is a String");
str.insert(9, "test");
system.out.println(str.toString());
```

这段代码将打印出：

```
This is a test String
```

3

setCharAt()方法的功能是在可变字符串的指定位置设置字符。

第6章

异常处理

程序运行中有可能会发生各种异常事件，如何处理这些事件是编程人员必须掌握的知识。

本章的主要内容包括:

- 异常处理概述
- 异常处理

6.1 异常处理概述

一个程序，除了按照用户需求完成所规定的功能外，还有可能在运行过程中发生各种异常事件，例如除0溢出、数组越界、文件找不到等，这些事件的发生将阻止程序的正常运行。为了加强程序的鲁棒性，设计程序时，必须考虑到可能发生的异常事件并做出相应的处理。当出现错误时，一种方法是终止程序的运行，但在某些情况下可能不是一种好的方法；另一种方法是在程序中引入错误检测代码，当检测到错误时就返回一个特定的值，但这种方法会将程序中进行正常处理的代码与错误检测代码混合在一起，使得程序变得复杂难懂，可靠性也会降低。

Java语言用异常处理为程序提供了错误处理方式。由发现错误但是不能处理的方法引发出一个异常，由该方法的直接或间接调用者来处理这个错误。Java的异常处理与返回错误码的处理方式完全不同，在例外情况发生时，Java的异常处理将抛出异常，并引起非局部的控制转移，即从引发异常的位置转移到可以捕获并处理异常的位置继续进行。异常为Java程序提供了一致的错误处理方式。

6.1.1 异常与异常对象

下面先看一个例子。

【例6.1】

```
/*
  * ExceptionSample.java
  * 演示异常的出现
  */
class ExceptionSample
{
    public static void main( String args[])
    {
        int x = 0;
        System.out.println(5/x);
    }
}
```

编译这个程序得到其字节码文件，然后运行它，结果如下：

```
c:> javac ExceptionDemo.java
c:>java ExceptionDemo
java.lang.ArthimeticException :/by zero
      at ExceptionDemo.mian(ExceptionDemo.java :4)
```

例6.1就出现了异常。异常（exception）是异常事件的缩写，它是程序执行过程中出现的违背正常指令流的事件。在通常情况下，异常处理应该明确指出终止程序执行的原因，并向方法的调用者发出错误产生的信号。与返回错误处理代码的处理方法不同，调用者可以忽

略所产生的异常。

C 语言中，通过使用 if 语句来判断是否出现了异常，同时，调用函数通过被调用函数的返回值感知在被调用函数中产生的异常事件并进行处理，全程变量 ErroNo 常常用来反映一个异常事件的类型。但是，这种错误处理机制既繁琐又费时，对那些本来执行机会就比较少的异常现象，却要花费大量的精力去编写异常程序，无疑是不值得的。传统的异常处理方法在一定程度上破坏了程序的可读性，也不利于程序的维护。

Java 编程语言通过面向对象的方法来处理异常，异常也被看成是对象，而且和一般对象没有什么区别，只不过异常必须是 Throwable 类及其子类所产生的对象实例。既然异常是一个类，那么它也像其他对象一样封装数据和方法。Throwable 对象在定义中包含一个字符串信息，而这个域可以被所有的异常类继承，它用于存放可读的描述异常条件的信息。该域是在异常对象创建的时候通过参数传递由构造方法设置的，可以用 throwable.getMessage()方法从异常对象中读取该信息。

Java 编程语言本身提供了错误检测机制，即异常处理机制。在 Java 程序的执行过程中，如果发生了异常事件，就会生成一个异常对象。该对象可能是由正在运行的方法生成，也可能是由 Java 虚拟机生成，其中包含异常事件的类型，以及当异常发生时程序的运行状态等信息。生成的异常对象被交给运行时系统，运行时系统寻找相应的代码来处理这一异常。我们把生成异常对象并把它提交给运行时系统的过程称为抛出(throw)一个异常。

在 Java 运行时系统得到一个异常对象时，它将会寻找处理这一异常的代码。寻找的过程从生成异常的方法开始，沿着方法的调用栈逐层回朔，直到找到包含相应异常处理的方法为止。然后运行时系统把当前异常对象交给这个方法进行处理。这一个过程称为捕获(catch)一个异常。如果查遍整个调用栈仍然没找到合适的异常处理方法，则运行系统将终止 Java 程序的执行。

6.1.2 异常类的层次

前面已经提到，Java 在处理错误时是采用面向对象的方法，用面向对象的方法处理异常，就必须建立类的层次。在 Java 中，每个异常都是一个对象，它是 Throwable 或其子类的实例对象，当一个方法出现异常后便抛出一个异常对象，该对象中包含异常信息，调用这个方法的方法可以捕获到这个异常对象并对它进行处理。图 6.1 表示的是异常处理的类层次。

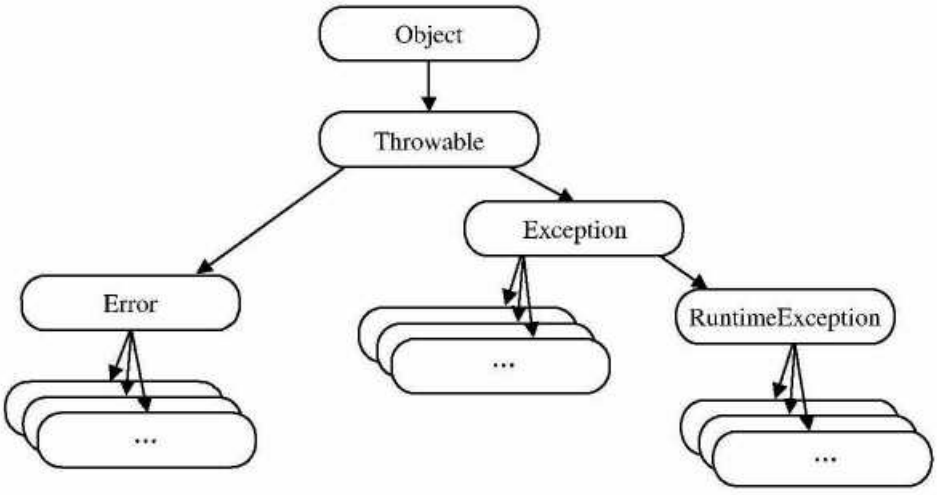


图 6.1 异常类层次

从图中可以看出，Throwable 类有两个标准子类：java.lang.Error 和 java.lang.Exception，即错误和异常。错误是指与虚拟机或动态装载等相关的问题，如系统崩溃、动态链接失败、虚拟机错误等，这类错误一般被认为是无法恢复和不可捕获的，将导致应用程序中断。异常是指一些可以被捕获而且可能恢复的异常情况，如访问下标越界、被零除等。

Exception 类对象是 Java 程序处理或抛出的对象。它有各种不同的子类分别对应于不同类型的异常。其中类 RuntimeException 代表运行时由 Java 虚拟机生成的异常，它是指 Java 程序在运行时发现的由 Java 解释器引发的各种异常，如算术运算异常 ArithmeticException、数组越界异常 ArrayIndexOutOfBoundsException 等；其他则为非运行时异常，是指能由编译器在编译时检测是否会发生在方法的执行过程中的异常，如输入/输出异常 IOException。Java 在其标准包 java.lang、java.util、java.io 和 java.net 中定义的异常类都是非运行时异常类。

Java 编译器要求 Java 程序必须捕获或声明所有的非运行时异常，如 FileNotFoundException、IOException 等，因为对于这类异常来说，如果程序不进行处理，可能会带来意想不到的结果。因此 Java 编译器要求程序必须捕获或者声明这种异常。但对运行时异常可以不做处理，因为这类异常事件的生成是很普遍的，要求程序全部对这类异常做出处理可能对程序的可读性和高效性带来不好的影响，因此 Java 编译器允许程序不对它们做出处理。表 6.1 所列的异常为常见的 Java 异常。

表 6.1　　常见的 Java 异常

异　常	引起的原因
ArithmeticException	数学错误，被零除
ArrayIndexOutOfBounds	错误的数组索引
ArrayStoreException	程序试图在数组中存储错误类型的数据
FileNotFoundException	企图访问一个不存在的文件
IOException	普通 I / O 故障，例如不能从文件中读
NullPointerException	涉及到一个空对象
NumberFormatException	在字符串和数字之间转换的故障
OutOfMemoryException	分配给新对象的内存太少
SecurityException	Applet 试图执行浏览器的安全设置不允许的动作
StackOverflowException	系统运行超出堆栈空间
StringIndexOutOfBoundsException	程序试图访问字符串中不存在的字符位置
SocketException	不能正常完成 socket 操作
ProtocolException	网络协议有错误

6.2 异常处理

Java 通过 5 个关键字来管理异常处理：try、catch、throw、throws 和 finally。通常在出现错误时用 try 来执行代码，系统引发（throws）一个异常后，可以根据异常的类型由 catch 来捕捉，或者用 finally 调用缺省异常处理。

6.2.1 捕获和处理异常

如果一个方法对某种类型的异常对象提供了相应的处理代码，则这个方法可捕获该种异常。Java 中通过使用 try...catch...finally 语句来捕获一个或多个异常，基本格式为：

```
try {
        //执行的代码块
} catch （Exceptiontype1 e）{
    //对异常类型 1 的处理
} cathch（Exceptiontype2 e）{
    //对异常类型 2 的处理
        throw（e）;      //重新抛出异常
} finally {
}
```

其中，catch 语句可以有一个或多个，而且至少要有一个 catch 语句或 finally 语句。

1 try

捕获异常的第一步是用 try{…}选定捕获异常的范围，在执行过程中，由 try 所指定的代码块中的语句会生成异常对象并被抛出。

2 catch

通常一个 try 块后面跟一个或多个包含异常指针的 catch 块，以处理指定的异常。catch 语句的参数类似于方法的声明，包括一个异常类型和一个异常对象。异常类型必须为 Throwable 类的子类，它指明了 catch 语句所处理的异常类型，异常对象则由运行时系统在 try 所指定的代码块中生成并被捕获，大括号中包含对象的处理，其中可以调用对象的方法。

catch 语句可以有多个，分别处理不同类的异常。Java 运行时系统从上到下分别对每个 catch 语句处理的异常类型进行检测，直到找到类型相匹配的 catch 语句为止。这里，类型匹配指 catch 所处理的异常类型与生成的异常对象的类型完全一致或者是它的父类，因此，catch 语句的排列顺序应该是从特殊到一般。

用一个 catch 语句也可以处理多个异常类型，这时它的异常类型参数应该是这多个异常类型的父类，在程序设计过程中，要根据具体的情况来选择 catch 语句的异常处理类型。

3 finally

在 try 所限定的代码中，当抛出一个异常时，其后的代码不会被执行。通过 finally 语句可以指定一块代码，使无论 try 所指定的程序块中抛出或不抛出异常，也无论 catch 语句的异常类型是否与所抛出的异常的类型一致，finally 所指定的代码都要被执行，它提供了统一的

出口。通常在 finally 语句中可以进行资源的清除工作，如关闭打开的文件等。

finally
try Java try
finally
try…catch…finally
try
Java try
try catch
finally
catch Java
try…catch…finally catch
Java
try
Java try try
finally

4

【例 6.2】

```
/*
 * TryCatchFinallySample.java
 * 演示 try、catch、finally 的用法
 */
public class TryCatchFinallySample
{
    static void Proc( int sel )
    {
        System.out.println("----- In Situation"+sel+" -----");
        Try
        {
            if( sel==0 ){
                System.out.println("no Exception caught");
                return;
            }
            else if( sel==1 )
            {
                int i=0;
```

```
                int j=4/i;
            }
            else if( sel==2 )
            {
                int iArray[]=new int[4];
                iArray[10]=3;
            }
        }catch( ArithmeticException e ){
            System.out.println("Catch "+e);
        }catch( ArrayIndexOutOfBoundsException e ){
            System.out.println("Catch "+e.getMessage());
        }catch( Exception e ){
            System.out.println("Will not be executed");
        }finally{
            System.out.println("in Proc finally");
        }
    }
    public static void main( String args[] )
    {
        Proc( 0 );
        Proc( 1 );
        Proc( 2 );
    }
}
```

运行结果为：

```
C:\>java TryCatchFinally
----- In Situation0 -----
no Exception caught
in Proc finally
----- In Situation1 -----
Catch java.lang.ArithmeticException: / by zero
in Proc finally
----- In Situation2 -----
Catch 10
in Proc finally
```

例 6.2 中，e.getMessage()调用了异常对象 e 的方法，输出它的有关信息。

6.2.2 用 throw 语句抛出异常

throw 语句将明显地引发一个异常。throw 语句只要求一个简单的参数，即一个可抛出的

对象。在 Java 系统中，可抛出对象是任何在 java.lang 中定义的可抛出类的子类的一个实例。如果抛出一个不可抛出的对象，则编译器将报错，并拒绝编译程序。下面是 throw 语句的一般格式：

```
throw throwableInstance;
```

首先，通过 catch 子句的参数或者使用 new 运算符获得一个 Throwable 实例的句柄。执行 throw 语句后，运行流程立即停止，throw 的下一条语句将暂停执行，系统转向内存的一个 try 语句块，并检查是否有 catch 子句能匹配 Throwable 实例。如果找到相匹配的实例，则系统转向该语句。如果没有找到，则转向上一层的 try 语句……这样逐层向上，直到最外层的异常处理程序终止程序。下面的程序是 throw 抛出一个异常，然后内部的异常处理程序又将同样的异常引发给外层的处理程序。

【例 6.3】

```
/*
  * ThrowSample.java
  * 抛出异常的例子
  */
class ThrowSample
{
        static void Example()
        {
            try
            {
                throw new NullPointerException("demo");
        }catch(NullPointerException e)
        {
                System.out.println("First catch inside Exception");
            throw e;
            }
    }
    public static void main (String args[])
    {
        try
        {
                Example();
        }catch(NullPointerException e)
        {
            System.out.println("recatch Exception: "+ e);
        }
    }
}
```

6.2.3 用 throws 子句声明异常

在方法中使用 try-catch-finally 可以由这个方法来处理它所生成的异常。在有些情况下，一个方法并不需要处理它所生成的异常，或者不知道该如何处理这一异常，这时它就向上传递，由调用它的方法来处理这些异常，这时就要用到 throws 子句。throws 子句包含在方法的声明中，其格式如下：

returnType methodName([paramlist]) throws exceptionList

其中，在 ExceptionList 中可以声明多个异常，用逗号隔开。Java 要求方法或者捕获所有可能出现的非运行异常，或者在方法定义中通过使用 throws 语句抛给上层调用者处理。Java 不要求声明除了非运行异常以外的异常，例如，每个运行在 Java 解释器上的方法都可能产生 InternalError 异常，该异常不需要在 throws 语句中声明。

请看例 6.4。

【例 6.4】

```
/*
  * throwsExceptionSample.java
  * 用 throws 子句声明异常的例子
  */
public class throwsExceptionSample
{
    static void Proc( int sel )
    throws ArithmeticException,ArrayIndexOutOfBoundsException
    {
        System.out.println("----- In Situation"+sel+" -----");
        if( sel==0 )
        {
            System.out.println("no Exception caught");
            return;
        }else if( sel==1 )
        {
            int iArray[]=new int[4];
            iArray[10]=3;
        }
    }
    public static void main( String args[] )
    {
        try
        {
            Proc( 0 );
            Proc( 1 );
```

```
            }catch( ArrayIndexOutOfBoundsException e ){
                System.out.println("Catch "+e);
            }finally{
                System.out.println("in Proc finally");
            }
        }
    }
```

运行结果为：

```
C:\>java throwsException
----- In Situation0 -----
no Exception caught
----- In Situation1 -----
Catch java.lang.ArrayIndexOutOfBoundsException: 10
in Proc finally
```

例 6.4 中，在 proc()方法中生成的异常通过调用栈传递给 main()方法，由 main()方法进行处理。

Java 包定义了几个运行异常类以方便用户使用，另外，Java 的程序设计者可以创建自己的错误和异常类或类集合来代表可能出现的问题。

Java

6.2.4 创建自己的异常类

在捕获一个异常前，必须有一段 Java 代码生成一个异常对象并把它抛出。抛出异常的代码可以是用户的 Java 程序，或者是 JDK 中某个类，或者是 Java 运行时系统。它们都是通过 throw 语句来实现的。throw 语句的格式为：

```
throw ThrowableObject;
```

其中 ThrowableObject 必须为 Throwable 类或其子类的对象。例如我们可以用

```
throw new ArithmeticExcption();
```

来抛出一个算术异常。

另外，我们还可以定义自己的异常类，并用 throw 语句来抛出它。通常在满足下述条件时可以创建新的异常类：

① 需要一个在 Java 开发环境中未被表示的异常类型；

② 新类集中的异常与其他异常有区别并使新类集的用户收益；

③ 新设计的代码抛出多个相关的异常；

④ 新设计的包是独立和自包含的。

【例 6.5】

```
/*
  * myExceptionSample.java
  * 创建自己的异常类的例子
  */
class MyExceptionSample extends Exception
{
    private int detail;
    MyException( int a )
    {
        detail = a;
    }
    public String toString( )
    {
        return "MyException "+detail;
    }
}

public 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);
        }
    }
}
```

运行结果为：

```
C:\>java ExceptionDemo
called compute(1)
normal exit
called compute(20)
Caught MyException 20
```

6.2.5 异常处理的优点和原则

本节将讨论异常处理机制的优点，并给出使用异常的一些原则。

1

（1）Java 通过面向对象的方法进行异常处理，把各种不同的异常事件进行分类，体现了良好的层次性，提供了良好的接口，这种机制对于具有动态运行特性的复杂程序提供了强有力的控制方式。

（2）Java 的异常处理机制使得处理异常的代码和“常规”代码分开，减少了代码的数量，增强了程序的可读性。

（3）Java 的异常处理机制使得异常事件可以沿调用栈自动向上传播，而不是 C 语言中通过函数的返回值来传播，这样可以传递更多的信息并且简化代码的编写。

（4）由于把异常事件当成对象来处理，利用类的层次性，可以把多个具有相同父类的异常统一处理，也可以区分不同的异常分别处理，使用非常灵活。

2

（1）对非运行时异常必须捕获或声明，而对运行时异常则不必，可以交给 Java 运行时系统来处理。

（2）对于自定义的异常类，通常把它做为类 Exception 子类，而不做为类 Error 的子类，因为 Error 类通常用于系统内严重的硬件错误。并且在多数情况下，不要把自定义的异常类作为运行时异常类 RuntimeException 子类。另外，自定义异常类的类名常常以 Exception 结尾。

（3）在捕获或声明异常时，要选取合适类型的异常类，注意异常的类层次，根据不同的情况使用一般或特殊的异常类。

（4）根据具体的情况来选择在何处处理异常。是在方法内捕获并处理还是用 throws 子句把它交给调用栈中上层的方法去处理。

（5）使用 finally 语句为异常处理提供统一的出口。

第7章

输入/输出处理

Java为各种输入/输出类设计了统一的接口，使得编程更加简单明了，但要重点注意不同类的实现差别。

本章的主要内容包括:

- 流和输入/输出处理的类层次
- 基本的输入/输出类
- 文件处理
- 内存的读/写
- 管道流
- 过滤流
- 标准输入/输出

7.1 流和输入/输出处理的类层次

输入/输出是计算机最基本的操作，计算机系统使用的所有数据从计算机的输入向输出流动，这种数据流动导致了术语流（stream）的产生。输入流（stream）是数据从外部设备（通常是键盘或鼠标）到计算机；输出流（output stream）是数据从计算机到外部设备（计算机屏幕或文件）。

计算机使用的外部设备各种各样，例如，磁带、磁盘、键盘、屏幕、终端等，Java 使用了一种称为“流”的逻辑设备来屏蔽不同设备间的差异。在行为上，所有的流都是类似的，Java 把不同类型的输入/输出抽象为流（stream），用统一的接口来表示，从而使程序设计简单明了。

流有两种类型：文本流和二进制流。文本流是一个字符序列，在文本流中，可按需要进行某些字符的转换，在被读写字符和外部设备中的字符之间不存在一一对应的关系，被读写字符的个数可能与外部设备中的字符个数不一样。二进制流是一个字节序列，它与外部设备中的字节存在着一一对应的关系，也就是说，不存在字符的转换，被读写字节的个数与外部设备中的字节个数是相同的。

在 Java 中，用类实现了流的输入/输出。其中一些最简单的类提供了基本的输入/输出，从该类派生出的其他类则针对某些特定种类的输入或输出，所有这些类都包含在 java.io 包中。图 7.1 是 Java 的输入/输出处理的类层次。

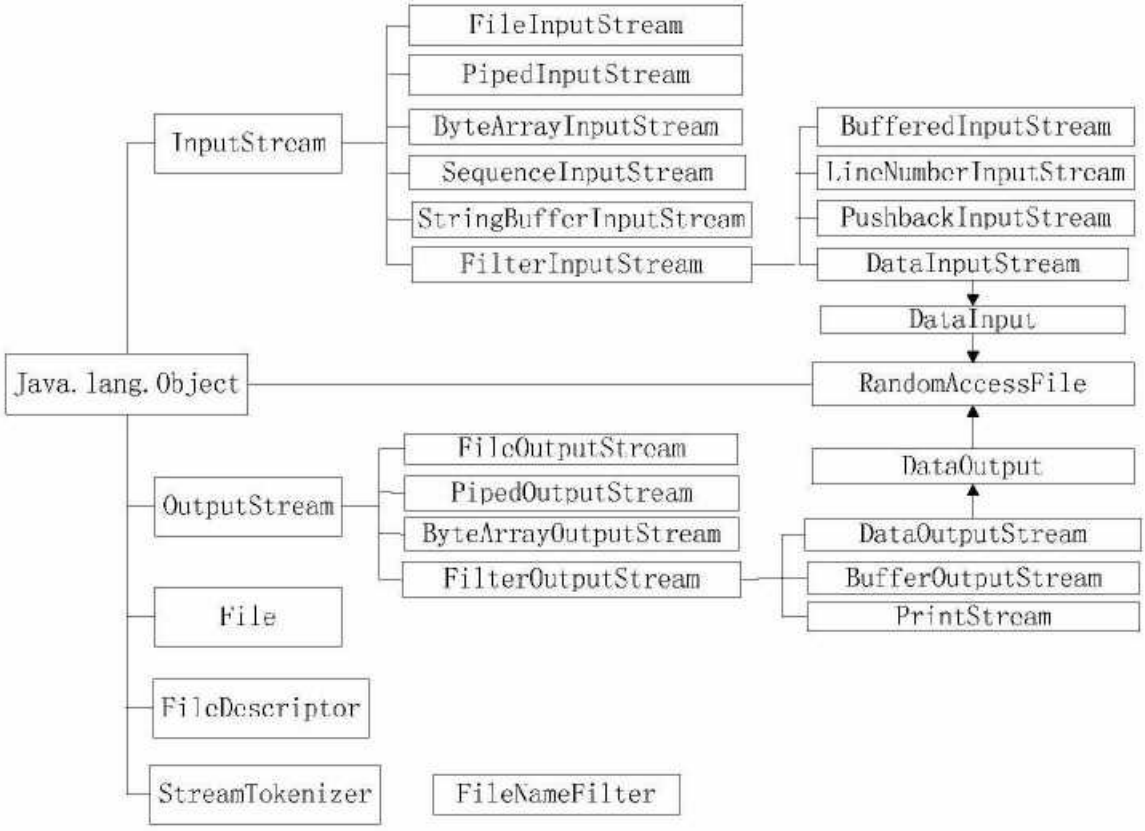


图 7.1 输入/输出处理的类层次

所有的输入流都是从抽象类 InputStream 继承而来的。在类 InputStream 中定义的方法包括：从流中读取数据、在流中标记某个位置、获取流中可得到的数据量，以及重置流中的读取位置等。

同样，所有的输出流都是从抽象类 OutputStream 继承而来的。在类 OutputStream 中定义的方法包括向流中写入数据以及清空流等。

其他的输入/输出流作为 InputStream 和 OutputStream 的子类，主要实现对特定流的输入和输出处理。

除了上面所说的流以外，包 java.io 中还提供了其他的一些类和接口。

（1）类 File 和 FileDescriptor 用于描述本地文件系统中的文件或目录。

（2）类 RandomAccessFile 用于描述一个随机的访问文件。

（3）类 Streamtokenizer 把流中的内容分解为记号（Token），通常用于文本文件的解析。

（4）当一个类实现了接口 DataInput 和 DataOutput 中定义的方法后，就可以用与机器无关的格式读写 Java 的基本数据类型。类 DataInputStream、DataOutputStream 和 RandomAccessFile 分别实现了这两个接口。

（5）接口 FilenameFilter 主要用于实现文件名查找模式的匹配。

7.2 基本的输入/输出类

InputStream 和 OutputStream 是抽象类，它们提供了输入/输出的基本功能，其他流类都是从这两个类派生而来的。惟一例外的是 RandomAccessFile 类，它允许对文件进行随机访问，即可以同时对文件进行读写。

7.2.1 InputStream 类

InputStream 类是一个抽象类，它定义了所有输入流所需的方法。

1

（1）int read();

从输入流中读取下一个字节，返回范围在 0～255 之间的一个整数，该方法的属性为 abstract，必须为子类所实现。

（2）int read(byte b []);

从输入流中读取长度为 b.length 的数据，写入字节数组 b，并返回读取的字节数。

（3）int read(byte b [], int off, int len);

从输入流中读取长度为 len 的数据，写入字节数组 b 中从索引 off 开始的位置，并返回读取的字节数。

对于以上方法，如果到达流的末尾位置，则返回-1 表明流的结束。

（4）int available();

返回从输入流中可以读取的字节数。

（5）long skip(long n);

输入流的当前读取位置向前移动 n 字节，并返回实际跳过的字节数。

2　Close()

关闭流并且释放与该流相关的系统资源。关闭流可以通过调用方法 close()显式进行，也可以由运行时系统在对流对象进行垃圾收集时隐式进行。

3

（1）void mark（int readlimit）;

在输入流的当前读取位置做标记。从该位置开始读取 readlimit 所指定长度的数据后，标记失效。

（2）void reset();

重置输入流的读取位置为方法 mark()所标记的位置。

（3）boolean markSupported();

返回一个 boolean 值，描述输入流是否支持方法 mark()和 reset()。

从以上方法中可以看出，InputStream 类中主要提供了对数据读取的基本支持，其中的方法通常都需要在子类中被重写，以提高效率或是适合于特定流的需要。对于输入流，其数据来源可以是文件、字符串、字节数组等，还有一种过滤的输入流是从其他的输入流中获取数据，并对这些数据进行转换或扩展后作为最终的输入数据。

7.2.2　OutputStream 类

和 InputStream 一样，OutputStream 是一个定义了输出流的抽象类。除了构造方法外，OutputStream 中封装的方法主要实现对输出数据的支持。对于输出流，其输出数据可以是文件、字节数组或将被其他输入流读的缓冲区，还有一种过滤的输出流是在写入到其他输出流之前对数据进行转换或扩展。

1

（1）void write(int b);

将指定的字节 b 写入到输出流。该方法的属性为 abstract，必须为子类所实现。注意参数中的 b 为 int 类型，如果 b 的值大于 255，则只输出它的低字节所表示的值。

（2）void write(byte b []);

将字节数组 b 中的 b.length 个字节写入到输出流。

（3）void write(byte b [], int off,　int len);

将字节数组 b 中从索引 off 开始的 len 个字节写入到输出流。

（4）void flush();

清空输出流，并输出所有被缓存的字节。

2

与类 InputStream 类似，可以用方法 close()显式地关闭输出流，也可以由运行时系统在对流对象进行垃圾处理时隐式关闭输出流。

通常 OutputStream 流中的方法都需要在子类中被重写，以提高效率或是适合于特定流的需要。

7.3　文件处理

在输入/输出处理中，最常见的是对文件的操作。文件也是一个逻辑概念，计算机系统的

所有设备都可以理解为一个文件。通过对一个文件的打开操作，流就与特定的文件建立了联系。一旦文件打开，那么在文件和程序之间就可以交换信息了。

值得注意的是，不是所有的文件都具有相同的功能，例如：磁盘文件支持随机存取，但打开的 socket 则不能。

如果文件支持随机存取，则打开该文件时，文件的位置指示符放在文件的首部。如果读写文件，则文件指示符将随之增加。

通过对一个文件的关闭操作，可以解除与特定文件的关联关系。

7.3.1 文件的输入/输出

任何希望保留其状态（包括任何已编辑文件的状态）的程序必须能够装载及存储文件。Java 提供了类 FileInputStream、FileOutputStream 和 RandomAccessFile。类 FileInputStream 和类 FileOutputStream 用来进行文件输入/输出处理，由它们所提供的方法可以打开本地主机上的文件，并进行读写；类 RandomAccessFile 提供了随机访问文件的支持。

1

如果读取的文件相对简单，可以使用 FileInputStream 类，它重写了父类 IputStream 中的方法 read()、skip()、available()和 close()，但不支持方法 mark()和 reset()。创建 FileInputStream 对象有下面 3 种方法。

（1）FileInputStream(String name);

（2）FileInputStream(File file);

（3）FileInputStream(FileDescriptor fdObj);

在生成类 FileInputStream 的对象时，如果指定的文件找不到，则会生成例外 FileNotFoundExceptio，它是非运行时例外，必须捕获或者声明抛弃。

例 7.1 演示了读取文件的方法，它显示源程序的内容。

【例 7.1】

```
/*
 * FileInputExample.java
 * 演示 FileInputStream 类的使用
 */
import java.io.*;
class FileInputExample
{
        public static void main(String args[])
        {
            byte buffer [] = new byte[2056];
```

```
        try
        {
            //创建文件的输入流对象
            FileInputStream fileInt = new FileInputStream("FileInputExample.java");
            //从文件的开头位置读取 2056 字节的内容到缓冲区 buffer
            int bytes = file.read(buffer,0,2056);
            String str = new String(buffer, 0 ,0 ,bytes);
            System.out.println(str);
        }
        catch (Exception e)        //捕获异常
            {
            String err = e.toString();
            System.out.println(err);
        }
    }
}
```

2

类 FileOutputStream 提供了写文件的功能，它重写了父类 OutputStream 中的方法 write() 和 close()。创建 FileOutputStream 对象也有 3 种方法。

（1）FileOutputStream(String name);

（2）FileOutputStream(File file);

（3）FileOutputStream(FileDescriptor fdObj);

FileOutputStream

例 7.2 演示了存储文件的方法，它将用户输入的一行信息存储在文件 test.txt 中。

【例 7.2】

```
/*
 * FileOutputExample.java
 * 演示 FileOutputStream 类的使用
 */
import java.io.*;
class FileOutputExample
{
    public static void main(String args[])
    {
        byte buf[] = new byte[80];
```

```
        try{
            System.out.println("Please input some words, it will storeinto test.txt");
            int bytes = System.in.read(buf); //读取用户的键盘输入
            //创建文件 test.txt 的输出流对象
            FileOutputStream fileOut = new FileOutputStream("test.txt");
            fileOut.write(buffer,0, bytes);
        }catch (Exception e){
            String err = e.toString();
            System.out.println(err);
        }
    }
}
3
```

对于类 FileInputStream 和类 FileOutputStream 来说，它们的实例都是顺序访问流，即只能进行顺序读写。随机文件则允许对文件内容的随机读写。类 RandomAccessFile 提供了随机访问文件的支持，它没有继承 InputStream 和 OutputStream，而是直接继承于 Object，并且实现接口 DataInput 和 DataOutput。

在接口 DataIput 中定义的方法主要包括：从流中读取基本类型的数据，读取一行数据，以及读取指定长度的字节数。如：readBoolean()、readInt()、readLine()、readFully()等。

接口 DataOutput 中定义的方法主要是向流中写入基本类型的数据，或写入一定长度的字节数组。如：writeChar()、writeDouble()等。

创建 RandomAccessFile 对象的方法有两种。

（1）RandomAccessFile(String name，String mode)

（2）RandomAccessFile(File file，String mode)

RandomAccessFile 类既可以用于读文件又可以用于写文件，它根据不同的参数区分是读还是写，例如：

```
RandomAccessFile rafR = new RandomAccessFile("fileApp.java","rw");
```

指出打开文件“fileApp.java”以进行输入和输出。

RandomAccessFile 类还支持文件指针（File pointer）的概念，文件指针用于指示当前文件中的位置。当文件第一次被创建时，文件指针是指向文件的开始位置，以后通过调用 read()和 write()方法根据读写的字节数调整文件指针。除了由通常的读 / 写操作隐式地移动文件指针外，RandomAccessFile 中还提供了一些方法以支持对文件指针的显式操作。例如：

```
long getFilePointer();          用于得到当前的文件指针
void seek(long pos);            用于移动文件指针到指定的位置
int skipBytes(int n);           使文件指针向前移动指定的 n 个字节
byte readByte()                 从文件读一个字节
```

例 7.3 演示了使用 RandomAccessFile 对象的方法，该程序的功能和例 7.1 一样，显示源程序文件的内容。

【例 7.3】

```
/*
 * RandomAccessFileDemo.java
 * 演示 RandomAccessFile 类的使用
 */
import java.io.*;
class RandomAccessFileExample
{
    public static void main(String args[])
    {
        byte buf [] = new byte[2056];
        try
        {
            //创建文件的 RandomAccessFile 对象
            RandomAccessFile fileRan = new
                    RandomAccessFile("RandomAccessFileExample.java", "r");
            long filePointer = 0;       //文件的初始位置
            long length = file.length();//得到文件的长度
            while (filePointer < length)
            {
                String str = file.readLine();       //从文件中读一行
                System.out.println(str);            //输出到屏幕
                filePointer = file.getFilePointer();            //得到文件指针当前的位置
            }
        }
        catch (Exception e)       //捕获异常
        {
            String err = e.toString();
            System.out.println(err);
        }
    }
}
```

7.3.2 File 类

File 类的核心概念是封装用户文件系统的某个文件或目录。另外，还包含许多用于执行文件常规操作的方法以及检查访问、删除、创建、更名指定的文件或目录。

File 类中有两个会经常用到的属性，一个是 separatorChar，它是一个系统的名字分割字符，在 UNIX 系统中为“ / ”，在 Windows 系统中为“\”；另一个是路径分隔符 pathSeparatorChar，在 UNIX 系统中为“：”，在 Windows 系统中为“；”。

1

创建 File 对象的构造方法如下所述。

（1）public File(Stirng pathname)

利用完整路径名 pathname 创建一个 File 对象。

（2）public File(String parent, String child)

利用一个单独的路径名 parent 和文件名 child 创建 File 对象。

（3）public file(File parent, String child)

利用一个单独的路径 parent 和文件名 child 创建 File 对象，该路径是一个 File 对象。

如果构造函数中的参数为 null，则会抛出一个 NullPointerException 异常。

2

一旦创建了一个文件对象，便可以使用以下 File 类的成员函数来获得文件相关信息，如图 7.1 所示。

表 7.1　　File 类的函数

方　法	功 能 解 释
public String getName()	得到文件或目录的名字，不含文件的路径
public String getParent()	得到当前文件或目录上一级的相对路径
public String getPath()	得到文件或目录的相对路径名
public boolean isAbsolute()()	判断文件或目录的相对路径是否是绝对路径
public String getAbsolutePath()	利用文件或目录的相对路径得到绝对路径
public String getCanonicalPath() throws IOException public File getCanonicalFile() throws IOException	利用文件或目录的相对路径得到规范的路径字符串，可以利用该方法得到某个文件或目录的绝对路径
public URL toURL() throws MalformedURLException	将相对路径转化为“file：URL”的格式
public boolean canRead()	测试文件或路径是否可读
public boolean canWrite()	测试文件或路径是否可写
public boolean exists()	测试文件或目录是否存在
public boolean isDirectory()	测试路径名所指定的是否是一个目录
public boolean isFile()	测试路径名所指定的是否是一个文件
public boolean isHidden()	测试文件或目录是否隐含
public long lastModified()	测试文件或目录的最后修改时间
public long length()	得到文件或目录的长度，单位为字节
public boolean createNewFile() throws IOException	创建一个新的空文件。如果要创建的文件不存在且创建成功，则返回 true；如果要创建的文件存在则返回 false。
Public boolean delete()	删除文件或目录，成功返回 true

续表

方　法	功 能 解 释
public void deleteOnExit()	当 JVM 退出时，删除文件或目录，删除的请求一经发出就无法取消
public boolean makedir()	创建一个目录
public boolean makedirs()	创建一个目录树
public boolean renameTo(File dest)	重命名文件或目录，成功返回 true
public boolean setLastModified(long time)	设置文件或目录的最后修改时间
public boolean setReadOnly()	将文件或目录设置为只读
public static File[] listRoots()	列出有效的文件系统的根
public File[] listFiles(FilenameFilter filter)	得到 File 对象所指定目录中的文件和目录
public static File createTempFile(String prefix,String suffix,File directory) throws IOException	在指定的目录 directory 中，使用指定的前缀 prefix 和后缀 suffix 创建一个新的临时文件。如果成功，则返回新创建的空文件
……	……

3

下面的例 7.4 演示了 File 类的使用，它的作用是显示文件的基本信息的程序，文件通过命令行参数传输。

【例 7.4】

```
/*
 * FileInfoExample.java
 * 演示 File 类的使用
 */
import java.io.*;
class FileInfoExample
{
    public static void main(String args[]) throws IOException
    {
        File fileToCheck;
        if(args.length>0)
        {
            for(int i=0;i<args.length;i++)
            {
                fileToCheck=new File(args[i]);
                info(fileToCheck);
            }
```

```
        }else{
                System.out.println("No file given.");
        }
    }
    public static void info(File f) throws IOException
    {
        System.out.println("Name: "+f.getName());
        System.out.println("Path: "+f.getPath());
        if(f.exists())
        {
            System.out.println("File exists.");
            System.out.print((f.canRead()?"and is Readable ":" "));
            System.out.print((f.canWrite()?"and is Writeable ":" "));
            System.out.println(".");
            System.out.println("File is "+f.lenght()+" bytes.");
        }else{
            System.out.println("File does not exist.");
          }
    }
}
```

7.4 内存的读/写

java.io 提供了对内存读写的支持，包括类 ByteArrayInputStream、ByteArrayOutputStream 和 StringBufferInputStream。

类 ByteArrayInputStream 可以从字节数组中读取数据，字节数组输入流的初始化是在给定的字节数组上完成的。类 ByteArrayInputStream 重写父类 InputStream 的方法 read()、available()、reset()和 skip()。其中方法 reset()重新设置输入流的读取位置为起始位置，即字节数组的起始位置。

类 ByteArrayOutputStream 可以向字节数组写入数据。类中提供了缓冲区以存放数据，并且该缓冲区的大小可随着数据的写入而自动增加。在构造方法中，可以指定初始缓冲区的大小，也可以使用缺省的 32 字节大小。在 ByteArrayOutputStream 实现的方法中，使用 size()可以得到输出流中的有效字节数；write()用于向输出流的字节数组缓冲区写入数据；通过 toByteArray()可以得到字节数组输出流缓冲区中的有效内容；reset()则清除字节数组输出流的缓冲区。另外，通过 writeTo()可以将流中的内容写入另一个指定的输出流中。

类 StringBufferInputStream 和 ByteArrayInputStream 基本类似，不同点是在于它是从字符串缓冲区 StringBuffer 中读取 16 位的 Unicode 数据，而不是 8 位的字节数据。

例 7.5 说明如何进行内存的读写。

【例 7.5】

```
/*
 * MemIOExample.java
 * 演示内存的读 / 写
 */
import java.io.*;
public class MemIOExample
{
public static ovid main(String args[])
{
    byte b[]= {65,66,67,68,69,70};
        int data;
    try
        {
        ByteArrayInputStream inStream = new ByteArrayInputStream(b);
        ByteArrayOutputStream outStream = new ByteArrayOutputStream();
        int backByte = inStream.available();
        inStream.skip(backByte/2);
        while((data = inStream.read())!=-1)
        {
            outStream.write(data);
        }
        System.out.println("read half datas" + outStream);
        inStream.reset();
        OutStream.reset();
        byte b1[]= new byte[b.length];
        inStream.read(b1,0,b.length);
        System.out.println("Read datas" +outStream.toString());
        outStream.write('\n');
        System.out.println("write to stdout…");
        FileOUtputStream stout = new fileOutputStream(filedescriptor.out);
        OutStream.writeTo(stdout);
        }catch(IOException e){
        System.out.println("error" + e);
        }
    }
}
```

7.5 管道流

管道用来把一个程序、线程或代码块的输出，并连接到另一个程序线程或代码块的输入。java.io 中提供了类 PipedInputStream 和 PipedOutputStream 作为管道的输入/输出部件。管道输入流作为一个通信管道的接收端，管道输出端则作为发送端。在使用管道前，管道输出流和管道输入流必须要进行连接。为此，PipedInputStream 和 PipedOutputStream 中提供了下面一些连接方法。

（1）在构造方法中，对于管道输入流和输出流来说，都可以给出对应的管道输出流和输入流作为参数进行连接，其接口分别为：

PipedInputStream(PipedOutputStream src)；

PipedOutputStream(PipedInputStream snk)；

（2）管道输入流和输出流还提供了方法 connect()以进行相应的连接。

类 PipedInputStream 的接口为：void connect(PipedOutpustStream src)；

类 PipedOutputStream 的接口为：void connect(PipedInputStream snk)；

同时，为了完成输入 / 输出操作，PipedInputStream 重写了方法 read()，PipedOutputStream 重写了 write()。

管道输入 / 输出提供了一种方便的途径，使一个程序产生的输出可以用作另一些程序的输入。例如，假设用户写一个类用以实现大量的文字处理（如排序、抽取特殊行以及倒装文字等），可能期望把这些方法的输出用作其他方法的输入，这样就可以把这些方法串起来以实现某些特定的功能。下面的方法 reverse()的作用是倒装输入的字符串，它从一个 InputStream 中读取数据，然后把 DataInputStream 映像到 source InputStream 中，这样就可以使用 DataInputStream 中的 readLine()方法来读取 source 文件中的每一行。然后用 reverse()方法创建一个 PipedOutputStream，把一个 PipedInputStream 与之相连（每一个 PipedOutputStream 必须有一个 PipedIputStream 与之相连，反之亦然）。Reverse()方法把一个 PrintStream 映像到 PipedOutputStream 后，就可以用 PrintStream 的 println 方法把字符串写入 PipedOutputStream 中。这样，reverse()方法从 source 中一行行地读取单词，用 reverseString()方法倒装处理之，然后将倒装后的字符串写入 PipedOutputStream 中。

当 reverse()关闭 PipedOutStream 后，所有写入 PipedOutputStream 中的数据被刷新到与之相连的 PipedInputStream 中。这个 PipedInputStream 就包含了已被倒装的单词序列，这时，reverse()方法就产生了一个适合于其他方法读的 PipeStream。

```
public static InputStream reverse(InputStream source)
{
    try{
        DataInputStream dis = new DataIputStream(source);
        String input;
        PipedOutputStream pos = new PipedOututStream();
        PipedInputStream pis = new PipedInputStream(pos);
        PrintStream ps = new PrinsStream(pos);
```

```
            While((input =dis.readLine()!=null)
            {
                ps.println(reversestring(input));
            }
            ps.close();
        }catch(Exception e){
                System.out.println("Reverse: "+ e);
        }
        return pis;
    }
```

7.6 过滤流

过滤流在从流中读取数据或者向流中写入数据的同时可以对数据进行处理，它提供了同步机制，使得某一时刻只有一个线程可以访问一个输入/输出流，以防止多个线程同时对一个输入/输出流进行操作所带来的意想不到的后果。

java.io 中提供类 FilterInputStream 和 FilterOutputStream，分别作为所有过滤输入流和输出流的父类，它们分别重写了父类 InputStream 和 OutputStream 的所有方法。为了使用一个过滤流，必须首先把过滤流连接到某个输入 / 输出流上，通常通过在构造方法的参数中指定所要连接的输入 / 输出流来实现。例如：FilterInputStream 和 FilterOutputStream 的构造方法分别为：

FilterInputStream(InputStream in)；

FilterOutputStream(OutputStream(OutputStream out)；

下面分别讲述几个过滤流的子类。

1　BufferedInputStream　　bufferOutputStream

类 BufferedInputStream 和 bufferOutputStream 实现了带缓冲的过滤流，由于提供了缓冲机制，把任意的输入流或输出流“捆绑”到缓冲流上会获得性能的提高。

在初始化 BufferedInputStream 和 bufferOutputStream 时，除了要指定所连接的输入/输出流之外，还可以指定缓冲区的大小。缺省的缓冲流使用 32 字节大小的缓冲区。通常，程序指定的缓冲区大小应是内存页或磁盘块等的整数倍，最优的缓冲区大小常依赖于主机操作系统、可使用的内存空间以及机器的配置等。

对于 BufferedInputStream，当读取数据时，数据按块读入缓冲区，其后的读操作则直接访问缓冲区。缓冲区的存在除了提高了性能外，还使得流 BufferedInputStream 可支持 mark()、reset()、skip()等方法。

BufferOutputStream()

flush()

2 LineNumberInputStream

除了提供对输入处理的支持外，LineNumberInputStream 可以记录当前的行号，将输入的每个换行字符“\n”，回车字符“\r”或者“\r\n”都处理为一个换行字符“\n”，行号从 0 开始，每遇到一个换行字符，则行号加 1。

使用方法 getLineNumber()可以得到当前的行号。

3 DataInputStream DataOutputStream

过滤流 DataInputStream 和 DataOutputStream 除了分别作为 FilterInputStream 和 FilterOutputStream 的子类外，还分别实现了接口 DataInput 和 DataOutput，因此 DataInputStream 可以从所连接的输入流中读取与机器无关的基本数据类型，DataOutputStream 可以向所连接的输出流写入基本类型数据。

4 PushbackInputStream

过滤流 PushbackInputStream 提供了一个字节的缓冲区，使得应用程序可以把刚刚读取的一个字符重新放回到输入流中，支持这一功能的方法是 unread()。

public void unread（int ch）throws IOException;

5 PrintStream

类 PrintStream 提供了方法 print()和 println()，可以方便地输出各种类型的数据，如 boolean、int、float、String 等。这些方法的参数还可以是 Ojbect 类型，这时，方法自动调用对象的 toString（）方法，然后输出得到的字符串。前面大量使用的 System.out 即为 PrintStream 类型。

6

许多程序需要实现自己的过滤流，以便在读 / 写操作时，可以对数据进行某些处理，这时的输入 / 输出流要继承 FilterInputStream 或 FilterOutputStream，并且输入流和输出流一般要成对实现。在实现的过程中，通常重写方法 read()、write()以及其他一些必要的方法。在 Java 中建立过滤输入 / 输出流的过程一般有以下几步。

（1）创建一个 FilterInputStream 和 FilterOutputStream 的子类，且输入流和输出流成对出现；

（2）置换 read()和 write()方法；

（3）置换所需的其他方法；

（4）确保输入流和输出流一起工作。

7.7 标准输入 / 输出

在前面已经讲过 Java 的标准输出流，如 println()。Java 中的标准输入 / 输出概念与 C 语言中的类似，有 3 种用 java.lang.system 类管理的标准流，即标准输入流 System.in、标准输出流 System.out 和标准错误流 System.err。

标准输出流和错误流可以向用户的终端窗口输出信息。标准输出流典型的应用是把命令输出，即输出命令的结果；标准错误流典型地用于打印程序运行过程中出现的错误，把程序的输出和错误放入两个不同的流中，并允许用户将它们送到不同的地方以区别对待。

标准输出流和标准错误流都是由 PrintStream 类导出的，所以可以用 PrintStream 的 3 个

方法 print()、println()和 write()来将信息送入流。Print()和 println()的方法相同，都是把字符串写入流中，惟一的区别是 println()自动在字符串后面加一个换行符，比如：

```
System.out.pirnt("ABCD\n");
```

等价于

```
System.out.println("ABCD");
```

print()和 println()方法都只有一个参数。因为 Java 语言支持方法的重载，所以这个参数可以是下述数据类型中的任何一种，这些数据类型是字符串、字符数组、整型、长整型、浮点数、布尔类型、对象等。

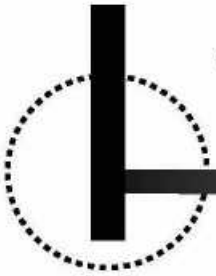
System.in 的 read()方法用于读取单个字符，其返回值是读到的字符；若没有读到的字符则返回值为－1。当一个程序从标准输入流读取数据时，程序被阻塞等待用户输入数据，直到用户输入结束标志（如回车键）时才继续执行。

例 7.6 是一个标准输入/输出的例子。

【例 7.6】

```
/*
 * StdOExample.java
 * 演示标准的输入输出流
 */
import java.io.*;
class StdIOExample
{
    public static void main(String args[]) throws IOException
    {
        int b;
        int count=0;
        while((b=System.in.read())!=-1)
        {
            count++;
            System.out.print((char) b);
        }
        System.out.println();//输出一空行
        System.err.println("counted"+count+"totalbytes.");
    }
}
```

第8章



线　　程

在需要同时运行多个任务时，线程变得十分有用，如何调度和避免死锁是使用线程的关键。

本章的主要内容包括:

- 线程概述
- 线程的创建和启动
- 与线程有关的类
- 线程的优先级和调度
- 线程的同步与死锁
- 线程组

8.1 线程概述

程序员对于编写顺序执行的程序都很熟悉，它们的特点是每个程序都有一个入口、一个出口和一个顺序执行的序列，在程序执行过程中的任何指定时间，都只有一个单独的执行点。线程和上面描述的顺序程序十分相似，但线程自己不能运行，必须在程序中运行，即一个线程是一个程序内部的顺序控制流。

多线程指的是在单个程序中，可以同时运行多个不同的线程，借以执行不同的任务。很多人可能对于多任务有一定的了解，即计算机看上去几乎同一时间内运行多个程序。而多线程的程序则在更接近于机器本身的基础上发展这一思想，这就是说，单个程序内部也可以在同一时刻进行多种运算。多线程并不等价于多次启动一个程序，操作系统也不把每个线程当作独立的进程来对待。虽然线程与进程都是顺序执行的指令序列，但它们之间是有区别的。一方面，进程是一个实体，每个进程都有自己独立的状态，并有自己的专用数据段；创建进程时，必须建立和复制其专用数据段。而线程则相互共享数据段，同一个程序中的所有线程只有一个数据段，以避免进行无谓的数据复制，因此线程的建立和线程间的切换速度大大优于进程。另一方面，线程又具备进程的大多数优点。但是，由于多个线程共享一个数据段，所以，也出现了数据访问过程的互斥和同步问题，使得系统管理功能变得相对复杂。

很多程序语言需要利用外部的线程软件包来实现多线程，而 Java 则内在支持多线程，它的所有类都是在多线程的思想下定义的。对多线程的支持是 Java 的一大特点，多线程可以大大地提高程序的运行效率。下面介绍 Java 中和线程有关的一些概念。

1

在多线程系统中，每个线程都被赋予一个执行优先级。优先级决定了线程被 CPU 执行的优先程序。优先级高的线程可以在一段的时间内获得比优先级低的线程更多的执行时间。这好像制造了不平等，然而却带来了高效率。如果线程的优先级完全相等，就按照“先来先用”的原则进行调度。

2

多个线程的并发执行实际上是通过一个调度程序来进行调度的。调度就是指，在各个线程之间分配 CPU 资源。线程调度有两种模型，即抢占式模型和分时模型。在分时模型中，CPU 资源是按照时间片来分配的，获得 CPU 资源的线程只能在指定的时间片内执行，一旦时间片使用完毕，就必须把 CPU 让给另一个处于就绪状态的线程；在分时模型中，线程本身不会让出 CPU。而在抢占式模型中，如果在一个低优先级的线程的执行过程中，又有一个高优先级的线程准备就绪，那么低优先级的线程就把 CPU 资源让给高优先级的线程。Java 支持的就是抢占式调度模型；因此，为了使低优先级的线程有机会运行，高优先级的线程应该不时地主动进入“睡眠”状态。

3

在 Java 中，线程可以共享数据，这就产生了同步的问题。假如两个线程 A 和 B 同时访问一个数据对象，线程 A 读这个数据对象而线程 B 写这个数据对象，或者两个线程同时写这个数据对象，就会导致诸如一致性、数据丢失等问题。这些问题在一些实际应用的领域（例

如，银行系统、电脑订票系统）尤其致命。Java 提供了一套同步化的机制，其基本思想是避免多个线程访问同一个资源。

4

Java 的线程从产生到灭亡，有以下几个状态。

（1）新建状态（newborn）

线程在已经创建但还未执行的这段时间里，处于一种特殊的新建状态中，此时，线程对象已经被分配了内存空间，私有数据已经被初始化，但是该线程尚未被调度。此时的线程可以被调度而变成可运行状态，也可以被杀死而变成停止状态。

（2）就绪状态或可运行状态（runnable）

这种状态表示线程正在等待 CPU 资源，随时可以被调度执行。处于就绪状态的线程实际上已经被调度，也就是说，处于就绪状态的线程已经被放到就绪队列中等待执行，至于该线程何时才被真正执行，则取决于线程的优先级和就绪队列的当前状况。

（3）运行状态（running）

运行状态表明线程正在运行，该线程已经拥有了对 CPU 的控制权。这个线程一直运行到运行完毕，除非该线程主动放弃 CPU 的控制权或者 CPU 的控制权被优先级更高的线程抢占。处在运行状态的线程在下列情况下将让出 CPU 的控制权：

- 线程运行完毕；
- 有比当前进程优先级更高的线程处于可运行状态；
- 线程主动睡眠一段时间；
- 线程在等待某一资源。

（4）挂起状态（blocked）

如果一个线程处于挂起状态，那么这个线程暂时无法进入就绪队列。处于挂起状态的线程通常需要由某些事件才能唤醒，至于由什么事件唤醒该线程，则取决于其挂起的原因。处于睡眠状态的线程必须被挂起一段固定的时间，当睡眠时间结束时就变成可运行状态；因等待资源或消息而被挂起的线程则需要由一个外来事件唤醒。

（5）停止状态（Dead）

表示线程已经退出运行状态，并且不再进入就绪队列。其中的原因可能是线程已经执行完毕，也可能是被另外一个进程强行杀死。

图 8.1 是这 5 个状态的相互转化图。

5

线程组是 Java 用以管理线程的概念。每个线程均属于某一个线程组，一个线程组可以包含多个线程或其他的线程组，从而形成线程之间的一种层次关系，这种层次关系有点儿类似于文件系统中目录的概念。一个线程可以访问其自身的线程组，却无法访问该线程组上层的线程组。

多线程程序设计允许单个程序创建多个并行执行的线程来完成各自的任务，相对于单线程程序设计而言，多线程的并行程序设计是比较困难的，利用 C 或 C++语言以及 OS 的多线程支持库很难保证线程的安全性。因为利用这种显式的多线程程序设计方式，程序员很难保证在需要的时候加锁，而在某个合适的时候又能恰当地将它释放。Java 在语言级提供了对线程的有效支持，通过语言和运行支持系统提供的复杂的同步机制，从而极大地方便了用户，

有效地减少了多线程并行程序设计的困难。

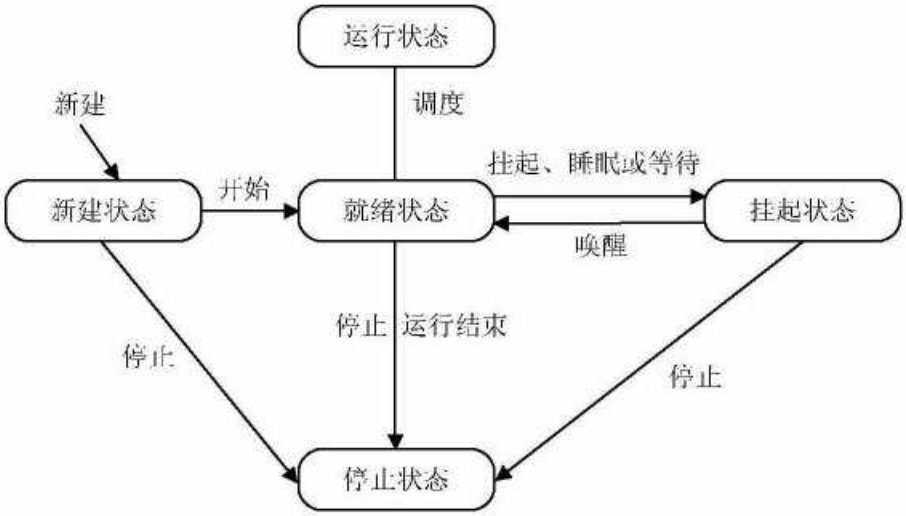


图 8.1　线程的生命周期

8.2　线程的创建和启动

当生成一个 Thread 类的对象之后，一个新的线程就诞生了。类 Thread 在包 java.lang 中定义，它的一个构造方法如下：

```
public Thread( ThreadGroup group, Runnable target, String name);
```

其中 group 给出了该线程所属的线程组；target 是执行线程体的目标对象，它必须实现接口 Runnable，name 为线程的名字。

当上述构造方法的某个参数为 null 时，可以得到下面几个构造方法：

```
public Thread();
public Thread(Runnable target);
public Thread(Runnable target, String name);
public Thread(String name);
public Thread(ThreadGroup group, Runnable target);
public Thread(ThreadGroup group, String name);
```

在接口 Runnable 中只定义了一个方法 run()作为线程体：

```
void run();
```

任何实现接口 Runnable 的对象都可以作为一个线程的目标对象，类 Thread 也实现了接口 Runnalbe，因此 Java 中建立线程有下述两种方法。

（1）定义一个线程类，它继承 Thread 类并重写其中的方法 run()，这时，在初始化这个类的实例时，目标对象可以为 null，表示由这个实例对象来执行线程体，由于 Java 只支持单重继承，用这种方法定义的类不能再继承其他父类。

（2）创建一个实现 Runnable 接口的类作为线程的目标对象，在初始化一个 Thread 类或

者 Thread 子类的线程对象时，把目标对象传递给这个线程实例，由该目标对象提供线程体 run()。这时，实现接口 Runnable 的类仍然可以继承其他父类。

Runnable

Applet

8.2.1 创建 Thread 类的子类

Thread 类是负责向其他类提供线程功能的最主要的类，为了向一个类增加线程功能，我们可以简单地从 Thread 类派生出一个类。在这种方法中，需要覆盖 run()方法来提供线程的执行代码，定义 Thread 类的成员变量来提供线程的数据。

run()方法是线程发生的地方，被称为线程体。线程执行时，从它的 run()方法中开始执行。run()方法是线程执行的起点，就像 main()方法是应用程序的执行起点，init()方法是小程序的执行起点一样。所以，程序员通过自己定义 run()方法来为线程提供代码。例如：

```
public class YourThread extends Thread
{
    public run()
    {
        //需要以线程方式运行的代码
    }
}
```

非常简单，为了在程序中实际地使用和设置线程，可以创建一个对象并调用 run()方法，如：

```
YourThread t = new YourThread();
t.start();
```

start()方法自动调用了 run()方法来执行线程的具体处理，线程将一直运行，直到退出 run()方法。

下面，通过例 8.1 来讲解如何使用创建 Thread 类的子类的方法来实现线程。

【例 8.1】

```
/*
 * TwoThreadExample.java
 * 演示 Thread 类的创建
 */
class TwoThreadsExample
{
    public static void main (String args[])
    {
        new SimpleThread("First").start();
```

```
            new SimpleThread("Second").start();
        }
    }
    class SimpleThread extends Thread
    {
        public SimpleThread(String str)
        {
            super(str);
        }
        public void run()
        {
            for (int i = 0; i <10; i++) {
                System.out.println(i + " " + getName());
                try {
                    sleep((int)(Math.random() * 1000)); //睡眠 0…1000 毫秒的时间
                } catch (InterruptedException e) {}
            System.out.println("Over! " + getName());
        }
    }
```

在这个例子中，类 TwoThreadsTest 的 main()方法中构造了两个 SimpleThread 类的线程，一个称为“First”，另一个为“Second”，并在构造后马上就调用了 start()方法来启动这两个线程。

类 SimpleThread 是类 Thread 的子类，它首先定义了一个构造方法，其中自变量为字符串类型，例如上面的“Second”，它的作用就是给线程取名，后面的方法 getName()是 Thread 类提供的方法，用来获得线程名。类 SimpleThread 中的第二个方法是 run()方法，它覆盖了类 Thread 中的 run()方法。run()方法中是一个 10 次的循环，每次循环中显示循环的次数和当前正运行的线程的名称，然后睡眠一个随机产生的时间间隔。在循环结束后，就显示“Over!”及线程名。

编译并运行这个程序，由于两个线程的调度情况是由操作系统动态决定的，因此，每次运行这个程序的结果不一定相同。一个可能的结果如下：

```
0 First
0 Second
1 Second
1 First
2 First
2 Second
3 Second
3 First
4 First
```

```
4 Second
5 First
5 Second
6 Second
6 First
7 First
7 Second
8 Second
9 Second
8 First
Over! Second
9 First
Over! First
```

可以看到：在输出结果中，两个线程的名称是混合在一起的，也没有顺序可循，这是因为这两个线程是同时运行的，并且是同时显示输出。

8.2.2 实现 Runnable 接口

创建一个类来实现 Runnable 接口，这种创建线程的方法更具灵活性，也使用户线程能够具有其他的一些类的特征，因此这种方法是最经常使用的。

Runnable 接口是定义在 java.lang 包中的一个接口，其中只提供了一个抽象的方法 run() 的声明，如下所示：

```
public interface java.lang.Runnable
{
    //Methods
    public abstract void run();
}
```

在类中实现 Runnable 接口的方法如下：

```
public class YourThread implements Runnable
{
    public run()
    {
        //需要以线程方式运行的代码
    }
}
```

或者

```
public class YourThread extends Applet implements Runnable
{
    public run()
    {
```

```
        //需要以线程方式运行的代码
    }
}
```

启动线程的方式也不一样：

```
YourThread your = new YourThread();
Thread t = new Thread(your);
t.start();
```

或者

```
YourThread your = new YourThread();
new Thread(your).start();
```

Thread Runnable run()
Thread Thread
Thread
Runnable run()
Thread Thread
Thread run()

例 8.2 是一个时钟 Applet。

【例 8.2】

```
public class Clock extends java.applet.Applet implements Runnable
{
    Thread clockThread;
    public void start()
    {
        if (clockThread == null)
        {
            clockThread = new Thread(this, "Clock");
            clockThread.start();
        }
    }
    public void run()
    {
        while (clockThread != null)
        {
            repaint();
```

```
                try
                {
                    clockThread.sleep(1000);
                } catch (InterruptedException e){}
            }
        }
        public void paint(Graphics g)
        {
            Date now = new Date();
            g.drawString(now.getHours()+ ":" + now.getMinutes()+ ":" +
            now.getSeconds(), 5, 10);
        }
        public void stop()
        {
            clockThread.stop();
            clockThread = null;
        }
    }
```

本程序的功能是在浏览器或者 appletviewer 中显示当前时间，它每隔一秒钟就更新一次时间显示。为了实现 clock applet，一方面，该 applet 的执行不能影响浏览器其他功能的执行，在 applet 显示时间的过程中，浏览器还可以执行其他功能（如滚动页面等），这就需要一个独立的线程来执行 clock applet；另一方面，由于它是 applet，因此，它需要从 java.applet.Applet 类继承，由于 Java 中不支持多重继承，所以它不能再继承类 Thread，而是利用了 Runnable 接口为其线程提供 run()方法。

首先，在 start()方法中构造了一个名为 clockThread 的线程 并调用 Thread.start()方法来启动这一线程，即在 Java 运行时建立系统的线程关系。下面语句建立了一个新的线程：

clockThread = new Thread(this, "Clock");

其中 this 是 Thread 构造方法中的第一个自变量，作为该线程的目标对象，它必须实现 Runnable 接口。在这种构造方法下，线程 clockThread 就以它的可运行的目标对象中的 run()方法为其 run()方法，在本例中，这个目标对象就是 Clock Applet。构造方法中的第二个变量为线程名。

线程被启动以后，调用目标对象的 run()方法，除非线程被停止。在 run()方法中实现了 while 循环，它在 clockThread 为 null 时退出。在循环体中，Applet 重绘本身，然后睡眠 1 秒，同时要捕获例外事件并进行处理。

当浏览器离开 applet 所在的页面后，系统将调用 applet 代码中的 stop()方法。Clock applet 中的 stop()方法停止线程的执行，并将线程置空，这样就停止了时钟的更新。当再次返回该页时，strat()方法又再次被调用，从而又将创建一个新的线程。

在具体应用中，采用哪种方法来构造线程体要视情况而定。通常，当一个线程已继承了另一个类时，就应该用第二种方法来构造，即实现 Runnable 接口。

8.3 与线程有关的类

通过前面的讲解，我们已经知道：Java 语言通过 Runnable 接口和 Thread、ThreadDeath 和 ThreadGroup 类提供了对线程的支持。本节将详细介绍这几个 Thread 类和 Runnable 接口，了解这些类对于深入掌握多线程编程是很有帮助的。

8.3.1 构造方法

Thread 类的构造方法很多，分别适应不同的需要。

（1）public Thread();

创建一个线程对象，此线程对象的名称是“Thread” +n 的形式，这里的 n 是一个整数。使用这个构造方法，用户必须创建 Thread 类的一个子类并覆盖其 run()方法。

（2）public Thread(Runnable target);

创建一个线程对象，此线程对象的名称是“Thread” +n 的形式，这里的 n 是一个整数。参数 target 的 run()方法将被线程对象调用，来作为其执行代码。

（3）public Thread(Runnable target, String name);

创建一个线程对象，此线程对象的名称是“Thread” +n 的形式，这里的 n 是一个整数。参数 target 的 run()方法将被线程对象调用，来作为其执行代码。参数 name 指定了新创建的线程的名称。

（4）public Thread(String name);

创建一个线程对象，参数 name 指定了线程的名称。

（5）public Thread(ThreadGroup group, Runnable target);

创建一个线程对象。参数 group 指定新建的线程所属的线程组，参数 target 的 run()方法将被线程对象调用，来作为其执行代码。如果当前线程不能在指定的组中创建线程，就会产生一个 SecurityException 异常。

（6）public Thread(ThreadGroup group,Runnable target, String name);

创建一个线程对象。参数 group 指定了线程所属的组。参数 target 的 run()方法将被线程对象作为执行代码执行。参数 name 指定了线程的名称。如果 group 是 null，则新建的线程和当前线程（即创建它的线程）属于同一个组，否则将调用 checkAccess 方法，检查当前线程是否能够在指定的组中创建一个线程，如果不能就抛出 SecurityException 异常。如果 target 是 null，则将 Thread 类的 run()方法作为其执行代码，否则调用 target 对象的 run()方法作为其执行代码。新创建的线程和创建它的线程拥有相同的优先级。

（7）public Thread(ThreadGroup group, String name);

创建一个线程对象，参数 group 和参数 name 的作用同前。

8.3.2 域

Thread 类定义了 3 个类常量。

public final static int MAX_PRIORITY;

public final static int MIN_PRIORITY;

public final static int NORM_PRIORITY;

这些常量规定了线程的优先级，它们的值分别为 10、5、1，值越大其优先级就越高。MIN_PRIORITY 表示最小可能的优先级，也就是说，在为线程指定优先级时，所指定的优先级不能小于 MIN_PRIORITY。MAX_PRIORITY 表示最大可能的优先级，即为线程指定的最大优先级不能超过此值。NORM_PRIORITY 表示线程的一般的优先级，也就是缺省的优先级。

8.3.3 方法

前面的例子中我们实际上已经使用了线程的一些方法，诸如 start()、run()、sleep 等。虽然这些方法是线程的一些非常重要的方法，但是 Thread 类还有其他一些有用的方法。

（1）public static int activeCount();

返回当前线程所在的线程组中的线程的数目。

（2）public void checkAccess();

检查当前运行的线程是否能够修改这个线程（指此方法所属的实例），如果不能，则抛出 SecurityException 异常。

（3）public int countStackFrames();

检查这个线程的堆栈框架并返回其数目。这个线程必须被挂起，否则会产生 IllegalThreadStateException 异常。

（4）public static Thread currentThread();

找到当前执行的线程并返回它的引用。

（5）public void destroy();

销毁一个线程，但是并不清除它。

（6）public static void dumpStack();

打印当前线程的堆栈情况，用来调试。

（7）public static int enumerate(Thread tarray[]);

将这个线程所在组的所有线程都拷贝到参数 tarray[]中去。

（8）public final String getName();

返回线程的名称。

（9）public final int getPriority();

返回线程的优先级。

（10）public final ThreadGroup getThreadGroup();

返回线程所在的线程组。

（11）public void interrupt();

中断当前线程。

（12）public static boolean interrupted();

如果当前线程被中断，则返回 true；否则返回 false。

（13）public final boolean isAlive();

判断当前线程是否活着，如果是，则返回 true；否则返回 false。线程活着是指线程已经被创建而且没有被杀死。

（14）public final boolean isDaemon();

判断这个线程是否是 Daemon 线程，如果是，则返回 true；否则返回 false。

（15）public boolean isInterrupted();

判断这个线程是否已经被中断，如果是，则返回 true；否则返回 false。

（16）public final void join();

等待这个线程死亡（被停止）。如果别的线程中断了当前线程，就会产生 InterruptedException 异常。

（17）public final void join(long millis);

等待这个线程死亡，最多等待 millis 毫秒。如果 millis 为 0，则意味着要无限等待。如果别的线程中断了当前线程，就会产生 InterruptedException 异常。

（18）public final void join(long millis, int nanos);

等待这个线程死亡，最多等待 millis 毫秒+nanos 纳秒。如果别的线程中断了当前线程，就会产生 InterruptedException 异常。

（19）public final void resume();

将一个线程由挂起状态变成可运行状态。如果当前线程不能修改这个线程，就会产生 SecurityException 异常。

（20）public void run();

如果这个线程实例是使用实现了 Runnable 接口的类的实例创建的，就调用这个类的实例的 run()方法，否则什么都不做并返回。

（21）public final void setDaemon(boolean on);

将一个线程设置成 Daemon 线程或者用户线程。Java 虚拟机在所有运行的线程都是 Daemon 线程的时候退出。这个方法必须在 start()方法被调用以前调用，否则如果在线程活着的时候调用就会产生 IllegalThreadStateException 异常。

（22）public final void setName(String name);

设置线程的名称。如果当前线程不能修改这个线程，就会产生 SecurityException 异常。

（23）public final void setPriority(int newPriority);

设置线程的优先级。如果当前线程不能修改这个线程，就会产生 SecurityException 异常。如果参数不在所要求的优先级范围之内，就会产生 IllegalArgumentException 异常。

（24）public static void sleep(long millis);

使当前执行的线程睡眠指定的时间。参数 millis 是线程睡眠的毫秒数。如果这个线程已经被别的线程中断，就会产生 InterruptedException 异常。

（25）public static void sleep(long millis, int nanos);

使当前执行的线程睡眠指定的时间。睡眠时间为 millis 毫秒+nanos 纳秒。如果这个线程已经被别的线程中断，就会产生 InterruptedException 异常。

（26）public void start();

使这个线程由新建状态变成可运行状态。如果该线程已经是可运行状态，就会产生 IllegalStartException 异常。

（27）public final void stop();

停止（杀死）这个线程。如果当前线程不能修改这个线程，就会产生 SecurityException

异常。即使这个线程刚刚新建，尚未变成可运行状态，也可以将这个线程停止。一个新建的ThreadDeath类的实例将作为异常被抛出。

（28）public final void stop(Throwable obj);

停止（杀死）这个线程。如果当前线程不能修改这个线程，就会产生SecurityException异常。即使这个线程刚刚新建，尚未变成可运行状态，也可以将这个线程停止。一个实现了Throwable接口的实例将作为异常被抛出。参数obj是要作为异常被抛出的实例。

（29）public final void suspend();

挂起一个线程。如果当前线程不能修改这个线程，就会产生SecurityException异常。

（30）public String toString();

返回一个代表该线程的字符串。

（31）public static void yield();

使当前执行的线程暂停执行，将CPU让给其他的线程。

8.4 线程的优先级和调度

8.4.1 线程的优先级和调度的基本机制

Java提供一个线程调度器来监控程序中启动后进入就绪状态的所有线程。线程调度器按照线程的优先级决定应调度哪些线程来执行，优先级决定了线程被CPU执行的优先程序。优先级高的线程可以在一段的时间内获得比优先级低的线程更多的执行时间。同时，线程的调度是抢先式的，如果在当前线程的执行过程中，一个具有更高优先级的线程进入就绪状态，则这个高优先级的线程立即被调度执行。Java中的每一个线程都有一个优先级，缺省情况下线程的优先级为5（即NORM_PRIORITY）；最高优先级为10（即MAX_PRIORITY）；最低优先级为1（即MIN_PRIORITY）。优先级高的线程先执行，优先级低的线程后执行。线程可以设置为守护线程（daemon）。

可以通过方法getPriority()来得到线程的优先级，也可以通过方法setPriority()在创建线程之后的任意时间改变线程的优先级。

```
int getPriority();
void setPriority(int newPriority);
```

当某一个时刻有多个线程处于Runnalbe状态时，Java运行系统从这些线程中选择优先级最高的线程执行。只有当运行中的高优先级线程停止或因其他原因使线程的状态变为Not Runnable时，低优先级的线程才可能运行。

Java的线程调度器支持不同优先级线程的抢先方式，但它本身不支持相同优先级线程的

时间片轮换。也就是说，如果有优先级高于正在执行线程的线程准备就绪时，Java 线程调度器将挂起当前正在执行的线程，而让优先级比它高的线程优先执行，但当准备就绪的线程的优先级与当前正在执行的线程的优先级相同时，如果 Java 运行系统所在的系统不支持时间片轮换方式，就不会挂起当前执行线程。如果在一个分时系统上实现 Java 运行系统，则可以使线程调度器支持相同优先级线程的时间片轮换方式。

例 8.3 的程序生成了 3 个不同的线程，其中线程在最低优先级下运行，另外两个线程在最高优先级下运行。

【例 8.3】

```
/*
 * ThreadPriorityExample.java
 * 演示 Thread 类的优先级设置
 */
class ThreadPriorityExample
{
    public static void main(String args[])
    {
        Thread Thread1 = new MyThread("Thread1");
        Thread1.setPriority(MIN_PRIORITY);
        Thread1.start();
        Thread Thread2 = new MyThread("Thread2");
        Thread2.setPriority(MAX_PRIORITY);
        Thread2.start();
        Thread Thread3 = new MyThread("Thread3");
        Thread3.setPriority(MAX_PRIORITY);
        Thread3.start();
    }
    class myThread extends Thread
    {
        String message;
        MyThread(String message)
        {
            this.message = message;
        }
        public void run()
        {
            for(int i = 0; i < 3; i++)
                System.out.println(message + ":" + getPriority());
        }
    }
```

8.4.2 Timer 类

为了方便在某个时刻以后台线程方式执行某种任务，Java 语言还提供了 Timer 类。Timer 类可以按照用户希望的方式在后台调度线程，让线程在将来某个时刻开始以一定的方式执行（执行一次或周期性地重复执行）。

Timer 类是线程安全，即多个线程可以共享一个定时器（Timer）对象，无需外部的同步。另外，timer 类并不保证线程执行的实时，它调度任务时使用的方法是 Object.wait(long)。

1

（1）public Timer();

创建一个新的 Timer 对象，与该对象相关的线程不作为守护线程运行。

（2）public Timer(boolean isDaemon);

创建一个新的 Timer 对象，参数 isDaemon 指定与该对象相关的线程是否作为守护线程运行。

2

（1）public void schedule(TimerTask task, long delay);

在指定的时间后，调度指定的任务执行。参数 task 为将被调度的任务，delay 为在任务执行之前以毫秒为单位的延时。如果 delay 为负数，或 delay 加 System.currentTimeMillis()为负数，将抛出 IlleaglArgumentException 异常；如果任务已经被调度或取消，或者定时器被取消，将抛出 IllegalStateException 异常。

（2）public void schedule(TimerTask task, Date time);

在指定的时间调度指定的任务。参数 task 为将被调度的任务，time 为执行任务的时间。如果 time.getTime()为负数，则抛出 IllegalArgumentEXception 异常；如果任务已经被调度或取消，或者定时器被取消，将抛出 IlegalStateException 异常。

【例 8.4】

```
Date timeToRun = new Date(System.currentTimeMillis()+500); //500 毫秒后执行
Timer timer1 = new Timer();
Timer.schedule(new timerTask(){
    Public void run(){
        //线程运行部分
    }
},timeToRun);
```

（3）public void schedule(TimerTask task, long delay, long period);

public void scheduleAtFixedrate(TimerTask task, long delay, long period);

在指定的延时之后，以固定的延时重复地调度指定的任务。参数 task 为将被调度的任务，delay 为在任务执行之前以毫秒为单位的延时，period 为连续执行任务时间的以毫秒为单位的时间间隔。如果 delay 为负数，或 delay 加 System.currentTimeMillis()为负数，将抛出 IllegalArgumentException 异常；如果任务已经被调度或者取消，或者定时器被取消，将抛出 IllegalStateException 异常。

（4）public void schedule(TimerTask task, Date firstTime, long period);

public void scheduleAtFixedRate(TimerTask task, Date firstTime, long period);

在指定的时刻，以固定的延时重复地调度指定的任务。参数 task 为将被调度的任务，firstTime 为第一次执行任务的时间，period 为连续执行任务时间的以毫秒为单位的时间间隔。参数 task 为将被调度的任务，time 为执行任务的时间。如果 time.getTime()为负数，将抛出 IllegalArgumentException 异常；如果任务已经被调度或者取消，或者定时器被取消，将抛出 IllegalStateException 异常。

（5）public void cancel();

中断当前的定时器，丢弃当前所有已调度的任务。

8.5 线程的同步与死锁

8.5.1 线程的同步

前面所提到的线程都是独立的，而且异步执行，也就是说每个线程都包含了运行时所需要的数据或方法，而不需要外部的资源或方法，也不必关心其他线程的状态或行为。但是经常有一些同时运行的线程需要共享数据，例如，一个线程向文件写数据，而同时另一个线程从同一文件中读取数据，因此就必须考虑其他线程的状态与行为，这时就需要实现同步来得到预期的结果。

我们把系统中使用某类资源的线程称为消费者，产生或释放同类资源的线程称为生产者，下面我们来讨论关于线程的同步问题的一般模型，即生产者—消费者问题。

在一个 Java 的应用程序中，生产者线程向文件中写数据，消费者线程从文件中读数据，这样，在这个程序中同时运行着的两个线程共享同一个文件资源。通过这个例子我们来讲解怎样使它们同步。

在这个例子中，生产者产生从 0～9 的整数，将它们存储在名为“CubbyHole”的对象中并打印出这些数来。然后调用 sleep()方法使生产者线程在一个随机产生的 0～100 秒的时间段内休息。

【例 8.5】

```
/*
 *  生产者消费者问题
 */
class Producer extends Thread
{
    private CubbyHole cubbyhole;
    private int number;
    public Producer(CubbyHole c, int number)
    {
        cubbyhole = c;
        this.number = number;
```

```
    }
    public void run()
    {
        for (int i = 0; i <10; i++)
        {
            cubbyhole.put(i);
            System.out.println("Producer #" + this.number + " put: " + i);
            try {
                sleep((int)(Math.random() * 100));
            } catch (InterruptedException e) { }
        }
    }
}
```

消费者线程则不断地从 CubbyHole 对象中取这些整数：

```
class Consumer extends Thread
{
    private CubbyHole cubbyhole;
    private int number;
    public Consumer(CubbyHole c, int number)
    {
        cubbyhole = c;
        this.number = number;
    }
    public void run()
    {
        int value = 0;
        for (int i = 0; i <10; i++)
        {
            value = cubbyhole.get();
            System.out.println("Consumer #" + this.number + " got: " + value);
        }
    }
}
```

在这个例子中，生产者与消费者通过 CubbyHole 对象来共享数据。但是，我们发现：不论是生产者线程还是消费者线程都无法保证生产者每产生一个数据，消费者就能及时取得该数据并且只取一次。通过 CubbyHole 中的 put()和 get()方法只能保证较低层次的同步，让我们来看一看可能发生的情况。

第一种情况是如果生产者比消费者快，那么在消费者来不及取前一个数据之前，生产者又产生了新的数据，于是，消费者很可能就会跳过前一个数据，这样就会有下面的结果：

…

Consumer #1 got: 3

Producer #1 put: 4

Producer #1 put: 5

Consumer #1 got: 5

…

第二种情况则反之，当消费者比生产者快时，消费者可能两次取同一个数据，可能会产生下面的输出结果：

…

Producer #1 put: 4

Consumer #1 got: 4

Consumer #1 got: 4

Producer #1 put: 5

…

上面两种输出结果都不是我们所希望的那样：生产者写一个数，消费者就取这个数。对于这种异步执行的多线程，由于希望同时进入同一对象中而发生错误结果的情况称为竞争条件(race condition)。为了避免竞争条件，线程在执行处理共享数据的代码段时需要进行同步，即保证不能有多个线程同时执行它。

1

在多线程程序设计中，将程序中那些不能被多个线程并发执行的代码段称为临界区，当某个线程已在临界区中时，其他的线程就不允许再进入临界区。

Java 语言专门引入了关键字 synchronized 来定义临界区，其一般定义格式如下：

synchronzied <expression> statement

其中可选的 expression 是对象或类的名字；statement 既可以是一个方法定义，也可以是一个语句块。当用 synchronized 来定义方法时，我们称该方法为同步方法；当用它来定义语句块时，称该语句为同步块。

在一个对象中，所有的同步对象或者同步块组成对象的临界区。在 Java 语言中，临界区的保护是由运行系统通过引入管程实现的，这种管程就像一把锁，当某个线程进入临界区时，系统将给临界区上锁，以阻止其他线程再次进入，直到已进入临界区的线程因为离开临界区等原因而释放锁，其他线程才能进入临界区。

2 Notify() wait()

有时，当某一个线程进入同步方法后，共享变量并不满足它需要的状态，该线程需要等待其他线程将共享变量改变为它需要的状态才能继续往下执行，但由于此时它占有了管程，其他线程进不到临界区中来改变该共享变量的值。Java 引入了 wait()和 notify()方法来克服这个问题。某线程需要在临界区中等待共享变量状态的改变时调用 wait()方法，这样该线程等待并暂时释放管程，其他线程也就可以进入临界区修改共享变量。当其他线程改变了共享变量后，只要调用方法 notify()，就可以通知正在等待的线程重新占用管程并唤醒该线程。

生产者—消费者问题中的类 CubbyHole 中提供了两个同步的方法：即 put()方法和 get()

方法。put()方法用来改变 CubbyHole 中的数据，get()方法则用来取数据；这样系统就把每个 CubbyHole 类的实例与一个 monitor 相对应。

【例 8.6】

```
class CubbyHole
{
    private int seq;
    private boolean available = false;
    public synchronized int get()
    {
        while (available == false)
        {
            try {
                wait(); // waits for notify() call from Producer
            } catch (InterruptedException e) {}
        }
        available = false;
        notify();
        return seq;
    }
    public synchronized void put(int value)
    {
        while (available == true)
        {
            try {
                wait(); // waits for notify() call from consumer
            } catch (InterruptedException e) {}
        }
        seq = value;
        available = true;
        notify();
    }
}
```

类 CubbyHole 有两个变量：seq 变量是 CubbyHole 的当前内容，布尔变量 available 指示当前内容是否可以取出。只有当 available 是真时，消费者才能取数据。为实现两线程同步，必须保证：

- 消费者接收数据的前提是 available 为真，即数据单元内容不空；
- 生产者发送数据的前提是数据单元内容为空。

在这个例子中，通过调用对象的 notify()和 wait()方法来保证 CubbyHole 中的每个数据只被读取一次。

（1）notify()方法。

在 get()方法返回前调用 notify()方法，它用来选择并唤醒等候进入监视器的线程。如果消费者线程调用了 get()方法，那么在整个执行过程中它将占据 monitor，在 get()方法结束前调用 notify()方法来唤醒处于等待状态的生产者线程。这样，生产者线程就占据了 monitor 并继续执行。

put( )方法与 get()相似，在一个线程结束前唤醒另一个线程。

与 notify()方法类似的还有 notifyAll()方法，不同的是它唤醒所有等待的线程，这些线程中的一个线程经过竞争进入 monitor，其他的继续等待。

（2）wait()方法

wait()方法使当前线程处于等待状态，直到别的线程调用 notify()方法来通知它。

get()方法包含了一个 while 循环，结束条件是 available 为真。如果 available 为假的话，消费者就知道生产者还没有产生新的数据，将继续等待。while()循环中调用 wait()方法，等待生产者线程发送消息。当 put()方法调用 notify()时，消费者线程被唤醒并继续 while()循环。put()方法中的 wait()方法作用相同。

get() available

monitor

monitor

Java monitor

monitor

monitor

主程序及输出结果如下：

```
class ProducerConsumerTest
{
public static void main(String args[])
{
CubbyHole c = new CubbyHole();
Producer p1 = new Producer(c, 1);
Consumer c1 = new Consumer(c, 1);
p1.start();
c1.start();
}
}
```

输出结果如下：

```
Producer #1 put: 0
0Consumer #1 got: 0
Producer #1 put: 1
```

```
Consumer #1 got: 1
Producer #1 put: 2
Consumer #1 got: 2
Producer #1 put: 3
Consumer #1 got: 3
Producer #1 put: 4
Consumer #1 got: 4
Producer #1 put: 5
Consumer #1 got: 5
Producer #1 put: 6
Consumer #1 got: 6
Producer #1 put: 7
Consumer #1 got: 7
Producer #1 put: 8
Consumer #1 got: 8
Producer #1 put: 9
Consumer #1 got: 9
```

由输出结果，我们可以看到：生产者和消费者两个线程实现了同步。

8.5.2 线程的死锁

由于系统资源有限，程序中多个线程互相等待对方资源，而在得到对方资源前不会释放自己的资源，造成都想得到资源而又都得不到资源，从而导致线程不能继续发展，这就是死锁问题。例如，一个线程持有对象 X，另一个线程持有对象 Y，第一个线程在持有对象 X，但必须拥有第二个线程所持有的对象 Y 才能执行；同样，第二个线程在持有对象 Y，但必须拥有第一个线程所持有的对象 X 才能执行，这两个线程就会无限期地阻塞，这时，线程的死锁就发生了。

在下面的例子中，我们创建了两个类 A 和 B，它们分别具有方法 fa 和 fb，在调用对方的方法前，fa 和 fb 都睡眠一会儿。主类 Deadlock 创建 A 和 B 实例，然后产生第二个线程以构成死锁条件。fa 和 fb 使用 sleep 来强制死锁条件出现。在现实程序中，错误的同步往往会出现死锁，而且是较难发现的。

【例 8.7】

```
/*
 * DeadlockExample.java
 * 演示死锁问题
 */
clas A
{
    synchronized void fa(B b)
    {
```

```
    try{
       Thread.sleep(1000);
    }catch (Exception e){};
    b.doFunc;
    synchronized void doFunc()
    {
        System.out.println("doing A.doFunc");
    }
}
class B
{
    synchronized void fb(A a)
    {
        try{
           Thread.sleep(1000);
        }catch(Exception e){};
        a.doFunc;
    }
    synchronized void doFunc()
    {
       System.out.println("doing B.doFunc");
    }
}
class DeadlockExample implements Runnable
{
    A a = new A();
    B b = new B();
    Deadlock();
    {
        Thread.currentThread().setName("MainThread");
        new Thread(this, "RacingThread").start();
        a.fa(b);
        System.out.println("back in MainThread");
    }
    public void run()
    {
       b.fb(a);
       System.out.println("bac to other thread");
    }
```

```
    public static void main(String args[])
    {
        new Deadlock();
    }
}
```

为了防止死锁问题,在进行多线程程序设计时需要遵循如下原则:

- 在指定的任务真正需要并行时才采用多线程来进行程序设计。
- 在对象的同步方法中需要调用其他同步方法时必须小心。
- 在 synchronized 封装的块中的时间尽可能的短，需要长时间运行的任务尽量不要放在 synchronized 封装的同步块中。

8.6 线程组

多线程编程对于很多程序员来说一件很头疼的事情，尤其是当运行的线程数量很多时，管理这些线程相当困难。线程组把多个线程集成为一个对象，通过线程组可以同时对其中的多个线程进行操作。线程组提供了将一组线程作为一个单独的对象进行统一处理或维护的手段，如通过线程组，程序员能用一个方法统一调用、启动或挂起线程组内的所有线程。

Java 的线程组由包 java.lang 中的类 ThreadGroup 实现。在生成线程时，可以指定将线程放在某个线程组中，也可以由系统将它放在某个缺省的线程组中。通常，缺省的线程组就是生成该线程所在的线程组。但是，一旦某个线程加入某个线程组，它将一直是这个线程组的成员，而不能被移出这个线程组。

在创建线程之前，可以创建一个 ThreadGroup 对象，下面代码是创建线程组并在其中加入两个线程。

```
ThreadGroup myThreadGroup = new ThreadGroup("Queen bee");
Thread myThread1 = new Thread(myThreadGroup, "worker bee 1");
Thread myThread2 = new Thread(myThreadGroup, "worker bee 2");
myThead1.start();
myThead2.start();
```

如上例所示，首先创建一个线程组，然后创建两个线程，并传递给 ThreadGroup 对象，称为线程组中的成员。每一个线程可以各自调用 start 方法启动。ThreadGroup 类并不提供一次启动所有线程的 start()方法，但是，可以通过调整线程组的优先级来挂起或者运行某个线程组中的所有线程。

调用方法 getThreadGroup()，可以获取一个线程所在的线程组。

```
ThreadGroup theGroup = myThread.getThreadGroup();
```

程序一旦得到一个线程所在的线程组，就可以得到关于该线程组的信息（如，该线程组中的其他线程等），并可以对其进行管理，如挂起、继续执行、终止线程等。TheadGroup 类提供了了管理一组线程的方法。

1

通过调用 Threadgroup 提供的方法可以查询线程组中的线程以及子线程组的信息，例如，调用方法 activeCount()可以取定线程组中所有的活动线程数目，包括子线程组中的活动线程。方法 activeCount()常常和方法 enumerate()一起使用，以获取线程组中的所有活动线程对象。请参看例 8.8。

【例 8.8】

```
/*
 * EnmuerateExample.java
 * 演示获取线程组中的所有活动线程对象
 */
class EnmuerateExample
{
void listCurrentThreas()
{
    ThreadGroup currentGroup = Thread.currentThread().getThreadGroup();
    Thread listOfThreads[];
    int num = currentGroup.activeCount();
    listOfThreads = new Thread[num];
    currentGroup.enumerate(listOfThreads);
    for(int I = 0; i < num; i ++)
    {
        System.out.println("Thread: " + i + "=" + listOfThreads[i].getName());
        }
}
}
```

2

类 ThreadGroup 支持以线程组为单位的操作，以设置或获取线程组的各种属性，例如，线程组中线程以了线程组的最大优先级、线程组的名字、父线程组的信息，以及是否为Daemon 线程组等。

（1）public final int getMaxPriority();

返回一个线程在线程组中能拥有的最高优先级。

（2）public final String getName();

返回该线程组的名字。

（3）public final ThreadGroup getParent();

返回该线程组的父线程组。

（4）public final boolean isDaemon();

判断是否是 Daemon 线程组。

（5）public boolean isDestroyed();

判断线程组是否被销毁。

（6）public final void setDaemon(boolean daemon);

改变线程组的 Daemon 状态，如果 daemon 是 true，则设置为 Daemon 状态，否则设置为正常状态。

（7）public final void setMaxPriority(int pri);

设置该线程组的最高优先级别，pri 是该线程组的新优先级。

（8）public final boolean parentOf(ThreadGroup g);

判断线程组是否是线程组 g 或者 g 的子线程组。

3

ThreadGroup 类提供了几个方法，用来改变线程组中的所有线程（包括子线程组中的线程）的当前状态，如 resume()、stop()、suspend()等。与 Thread 类一样，ThreadGroup 类中的 resume()、stop()、suspend()方法已不鼓励使用（deprecation），对线程组的挂起或者运行建议通过优先级的调整和让线程睡眠（sleep）来实现。

第9章

编写 Applet 程序

Applet 程序使得 Web 页面的交互性和动态性得到大大提高，它在页面中的嵌入很简单，同时也是个安全的技术。

本章的主要内容包括:

- 编写 Applet 程序概述
- Applet 的主类
- Applet 的生命周期
- Applet 类方法
- Applet 如何嵌入 Web 页面
- Applet 通讯
- Applet 在安全方面的限制
- Applet 的用户界面

9.1 编写 Applet 程序概述

Java 语言的特性使它可以最大限度地利用网络资源。Applet 是 Java 的小应用程序，它是动态、安全、跨平台的网络应用程序。Java Applet 嵌入 html 语言，通过主页发布到 Internet。网络用户访问服务器的 Applet 时，这些 Applet 从网络上进行下载，然后在支持 Java 的浏览器中运行。由于 Java 语言的安全机制，用户一旦载入 Applet，就可以放心地来生成多媒体的用户界面或完成复杂的计算，而不必担心病毒的入侵。

使用 Java 的小程序，可以大大的提高 Web 页的交互能力和动态执行能力。嵌入了 Applet 的 Web 页面被称为有 Java 能力的 Web 页，而可以运行小程序的浏览器被称为 Java 兼容的浏览器。目前，市场上占主流的浏览器——IE 和 Navigator，都是 Java 兼容的浏览器。不支持 Java 的浏览器也可以有 Java 能力的 Web 页，但是其中的小程序将被忽略。

虽然 Applet 可以和图像、声音、动画等一样从网络上下载，但它并不同于这些多媒体文件格式，它可以接收用户的输入，动态地进行改变，而不仅仅是动画的显示和声音的播放。

例 9.1 是一个简单的 Applet（小应用程序）。

【例 9.1】

```
import java.awt.*;
import java.applet.*;
public class HelloApplet extends Applet{
        //the applet class
       public void paint(Graphics g){
           g.drawString("Hello, this is the applet!",20,20);
       }
}
```

在程序中，首先用 import 语句引入 java.awt 和 java.applet 下所有的包，使得该程序可以使用这些包中所定义的类。然后声明一个公共类 HelloApplet，用 extends 指明它是 Applet 的子类。在类中，重写父类 Applet 的 paint()方法，其中参数 g 为 Graphics 类，它表明当前作画的上下文。在 paint()方法中，调用 g 的方法 drawString()，在坐标(20,20)处输出字符串“Hello, this is the applet!”，其中坐标是用像素点来表示的。

这个程序中没有实现 main()方法，这是 Applet 与应用程序 Application 的区别之一。为了运行该程序，首先也要把它放在文件 HelloApplet.java 中，然后对它进行编译：

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

得到字节码文件 HelloApplet.class。由于 Applet 中没有 main()方法作为 Java 解释器的入口，因此必须编写 html 文件，把该 Applet 嵌入其中，然后用 appletviewer 来运行，或在支持 Java 的浏览器上运行。它的<html>文件如下：

```
<html>
<head>
<title> an Applet</title>
```

```
</head>
<body>
<applet code=" HelloApplet "  width=200 height=40>
</applet>
</body>
</html>
```

其中用<applet>标记来启动 HelloApplet，code 指明字节码所在的文件，width 和 height 指明 Applet 所占的大小，把这个 html 文件存入 appletExample.html，然后运行：

```
C:\>appletviewer appletExample.html
```

这时，屏幕上弹出如图 9.1 所示的一个窗口，其中显示 Hello, this is the applet!。如图 9.1 所示。

图 9.1　Applet 示例

9.2　Applet 的主类

从上面的例子中我们可以看到，小应用程序是一个扩展了的小应用程序类（Applet）的 Java 类，该子类定义了小应用程序的行为框架，因此，也称之为 Applet 主类，Applet 类在软件包 java.apple 中定义，Applet 类在 Java 类中的继承层次如下：

```
java.lang.Object
    |___...java.awt.Component
            |__...java.awt.Panel
                    |__...java.applet.Applet
```

这样的层次结构决定了 Applet 的工作方式和工作能力，它与 awt 类密切相关。本章将介绍 Applet 的基础知识，而如何使用 awt 创建图形用户接口的详细内容请参考本书的相关章节。

实际上，Applet 类是 java.applet 包中的惟一的类，其他的是关于接口的定义，Applet 接口包括下述几项。

- AppletContext：用来帮助应用小程序与其他小程序进行通信。
- AppletStub　：用来编写应用小程序浏览器。
- AudioClip　：用来帮助应用小程序播放声音文件。

由于 Applet 类是公有的，因此，作为它的扩展类，Applet 主类也必须被说明为公有的，其他需要引入的类可以被说明为私有的也可以被说明为公有的。

当Java兼容的浏览器在Web页中发现有Applet时，它通过网络从服务器上下载Applet主类以及其他必要类。当Applet下载后，Java兼容的浏览器产生该Applet主类的一个实例，并将所有基于系统支持的方法送至该实例。同一Web页上的不同Applet以及不同Web页上的Applet都将产生不同的实例，运行在同一系统上的不同Applet具有各自独立的行为。

9.3 Applet的生命周期

Applet的生命周期是指从页面上的Applet被下载直至它被系统收回时所经历的过程。Applet的生命周期可分为四个阶段，各阶段分别由init()、start()、stop()和destroy()成员方法来具体体现，生命周期中的主事件发生时，Java兼容浏览器自动调用这些方法，Applet程序本身并不直接调用这些方法，在9.4.1节中将详细描述这四个方法。

在Applet生命周期中发生的引起Applet状态变化的关键事件称为主事件，在各个状态之间有如下事件.

（1）下载Applet

当一个Applet被下载到本地系统时会发生这个事件，它由3个子事件组成。

- 产生一个Applet主类的实例；
- 对Applet进行初始化工作；
- 启动Applet运行。

（2）离开或返回Applet所在的Web页

当离开Applet所在的Web页时，Applet将会停止运行，而当返回该页面时，Applet又会重新启动运行。

（3）重载Applet

有些浏览器允许重载Applet，在重载事件中，Applet先停止自身运行，接着释放Applet占用的所有资源，然后重新下载该Applet。

（4）关闭页面

当关闭页面或退出浏览器时，Applet先停止自身运行，然后释放Applet占用的所有资源，最后关闭页面或退出浏览器。

图9.2显示了Applet生命周期的各个阶段，以及主事件发生时的阶段转换关系。

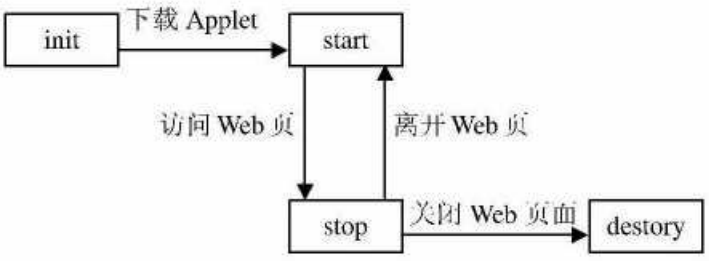


图9.2 Applet生命周期

9.4 Applet 类方法

9.4.1 生命周期方法

Applet 生命周期方法由下述四个成员方法来体现。

```
1
public void init(){
…
}
```

初始化发生于一个 Applet 首次被下载或重载时，初始化的主要任务是创建 Applet 所需要的对象，设置初始状态，装载声明的图像和字模(fonts)，设置参数等。Applet 程序员通过重载 init()方法来提供自己需要的初始化行为。

```
2
public void start() {
    …
}
```

当 Applet 初始化以后，启动方法会接着被调用，或者当离开该 Applet 所在的页面时，这个 Applet 被停止，当再次打开该页面时也会调用该 Applet 的启动方法。Applet 程序员通过重载 start()方法来提供自己需要的启动行为，例如，启动一个线程，发送消息给引入类对象，或者通知本 Applet 已经启动等。

```
3
public void stop() {
    …
}
```

当离开 Applet 所在的页面时，将调用 Applet 的停止方法，这时，Applet 将停止使用系统资源；若没有定义停止方法，则当用户离开时，Applet 也会继续使用它原先占用的系统资源。另一种情况是，在调用 Applet 的删除方法前要调用停止方法。Applet 程序员可以通过重载停止方法来提供自己需要的停止行为，例如，以某种方式通知该 Applet 已经停止。

```
4
public void destroy() {
    …
}
```

当 Applet 所在的页面被关闭或浏览器被关闭时，将调用 Applet 的停止方法，一般情况下不需要重载该方法，使用系统缺省的删除隐含行为，只有在必须释放某些系统特定的资源时才定义删除方法，例如，在启动时创建了线程，就必须在删除时释放这些资源。

9.4.2 绘制方法

绘制方法使一个 Applet 在窗口中显示某些信息，如文本、图像等。在 Applet 被初始化后，

以及该页面被重新放到前面，或者浏览器从一个地方被移到另一个地方，都需要调用绘制方法。简而言之，当一个 Applet 需要在屏幕上显示自己时，该方法被调用。格式为：

```
public void paint( Graphics g ) {
        …
}
```

绘制方法（paint()方法）带一个参数 g，g 为 Graphics 类的实例，该对象由浏览器产生并在这里传递给 Applet 类的 paint()方法，因此在 applet 文件首部必须声明引入 java.awt.Graphics 软件包（或者 java.awt.*也可以）。这样，在 paint()方法中就可以使用 Graphics 类的方法来进行绘图工作了。

9.4.3 html 页面方法

由于 Applet 都是运用于 html 页面中，因此 Applet 类也提供了一些方法来获取包含该 Applet 的 html 页面信息。如页面地址、页面传递的参数等。

（1）public URL getDocumentBase()；

返回包含该 Applet 的 html 页面的 URL。

（2）public URL getCodeBase()；

返回 Applet 自身的 URL，注意 Applet 的 URL 可以不同于页面的 URL。

（3）public String getParameter(string name)；

html 页面中的由<PARAM>标志中定义的参数可以通过这个方法来获取参数值。

【例 9.2】

```
<html>
<head>
…
</head>
<body>
…

<APPLET CODE="HelloApplet" WIDTH=200 HEIGHT=50>
<PARAM NAME=COLOR VALUE="RED">
</APPLET>
…
</body>
…
</html>
```

在 Applet 源码中可以调用 getParameter("COLOR")，返回值为“RED”，若未定义该参数则返回“NULL”。

9.4.4 多媒体支持方法

由于 Applet 类提供了从指定的 URL 获取图像和声音的方法，包含 Applet 的页面可以很

方便地使用网络上的各种多媒体文件。

（1）public void play(URL url);

直接演示指定 URL 上的声音文件，该参数是一个绝对的 URL。

（2）public void play(URL url, String name);

直接演示指定 URL 上指定的声音文件。

（3）public Image getImage(URL url);

返回参数 url 指定的图像，无论图像是否存在，该方法都会立即返回，当图像真正需要被显示时，图像数据才被装载，并由显示图像的图形原语画出图像。

（4）public Image getImage(URL url, String name);

返回指定 URL 上指定的图像文件。

（5）public AudioClip getAudioClip(URL url);

获得指定 URL 上的声音数据，返回一个 AudioClip 类型的对象，通过使用该对象可以实现声音的演示。

（6）public AudioClip getAudioClip(URL url, String name);

获得指定 URL 上指定的声音数据。

9.4.5 Applet 管理环境方法

Applet 类能够通过一些方法来管理它所在的环境，例如更换浏览器中显示的 Web 页面。但 Applet 的环境管理方法只能提供一些有限的支持，因为它无法确认 Applet 是否存在于 Web 浏览器中，或者是否能提供支持功能的浏览器，而 AppletContext 接口可以提供更好的方法。

（1）public AppletContext getAppletContext();

返回一个 AppletContext 实例，通过这个实例，Applet 可以管理它所在的环境。下面的 4 个方法是 AppletContext 接口提供的方法。

（2）public void showDocument(URL url);

用指定的 URL 代替当前的 Web 页面。

（3）public void showDocuments(URL url, String target);

按 target 指定的方法显示指定的 URL 页面，target 的取值可以为："_self"（当前窗口），"_parent"（父窗口），"_top"（最高一级的窗口），"_blank"（新建窗口）或"name"（指定名称的窗口）。

（4）public Applet getApplet(string name);

返回同一页面上的用<NAME>属性说明的名字为 name 的 Applet，如果该页面上不存在名字为 name 的 Applet，则该方法返回值为空。

（5）public Enumeration getApplets();

返回当前 Web 页面上所有的 Applet 的列表。为保证安全性，该方法只返回与调用此方法的 Applet 同一主机的 Applet。

9.4.6 Applet 信息报告方法

Applet 类提供一些方法来向用户报告本 Applet 的信息，以及在标准输出上显示诊断信息。

（1）public String getAppletInfo();

获得关于 Applet 的作者、版权、版本号等信息。

（2）public void ShowStatus(String status);

在标准输出上显示字符串。

（3）public String[][] getparameterInfo();

获得 Applet 中描述的所有参数，以字符串数组形式返回，数组中的每个元素都由 3 个字符串组成，形式如下：{name, type, comment}，其中，name 为参数名，type 为参数类型，comment 为参数含义的描述。

9.5 Applet 如何嵌入 Web 页面

在编写完了一个 Applet 程序并将它编译为字节码后，随后的工作便是将 Applet 嵌入到 Web 页面中，从第一小节的例子中，我们可以看到，Applet 是通过一种特殊的 html 标记<applet>嵌入的，当 Java 兼容浏览器访问到这个标记时便会使用其中的信息来找到 applet 类文件并执行它。

<applet>标记的结构包括 3 个部分，各部分的顺序如下：

```
<applet standard-attribute>
parameter-define
alternate-content
</applet>
```

首先是 applet 标记，其中的 standard-attribute 描述了 applet 的标准属性列表，其中一部分是必选属性，另一部分是可选属性。Parameter-define 定义了一系列 applet 参数。alternet-content 提供了可以在非 Java 兼容浏览器上显示的内容。

9.5.1 applet 标记

applet 标记中的标准属性列表的格式是：name=value。其中，name 是属性名，value 是属性值。该列表的完整语法如下（黑体表示是必选的，其他是可选的）：

```
<applet
    CODE = appletFile   or   OBJECT = serializedApplet
    WIDTH = pixels   HEIGHT = pixels
    ALIGN = alignment
    ALT = alternateText
    ARCHIVE = archiveList
    CODEBASE = codebaseurl
    NAME = appleInstanceName
    VSPACE = pixels   HSPACE = pixels
>
```

CODE OBJECT WIDTH HEIGHT

（1）CODE = appletFile

这个属性提供了包含 Applet 的编译好的 Applet 子类的文件名，它不是绝对的 URL，该文件名是相对于 Applet 的基本的 URL 的。文件名的形式如：appletclassname.class 或 packagename.appletclassname.class。CODE 和 OBJECT 中必须有而且只能有一个存在。

（2）OBJECT = serializedApplet

这个属性提供了包含 applet 序列化表示的文件名。该文件名指定的 Applet 将被序列化恢复，但在调用 start()方法之前，init()方法不会被调用。在源对象被序列化时，有效的属性不会被恢复。一般情况下，建议不要使用该属性。序列化之前应该停止该 Applet 。

（3）WIDTH = pixels　HEIGHT = pixels

这两个属性给出了 Applet 显示区域的初始宽度和高度，单位为像素，但不包括 Applet 内部显示的对话框和窗口。

（4）ALIGN = alignment

这个可选属性指定了 Applet 的对齐方式，该属性的可选值有：left, right, top, texttop, middle, absmiddle, baseline, bottom 和 absbottom。

（5）ALT = alternateText

这个可选属性指定了，在浏览器中能识别 Applet 标记但不能运行 applet 时显示的内容。

（6）ARCHIVE = archiveList

这个可选属性指定了一个或多个要预加载的类或其他资源归档，每个归档以逗号隔开。出于安全方面的考虑，这里只能加载与 CODEBASE 位于相同目录的文件，

（7）CODEBASE = codebaseurl

这个可选属性指定了 Java 类文件（.class）的路径或 URL。如果未指定该属性，则将使用该 html 页面文件所在目录。当页面文件与 Applet 类文件不在同一个目录时，要用到这个属性。

（8）NAME = appleInstanceName

这个可选属性指定了，在同一页上的不同 Applet 之间通信的符号名称。

（9）VSPACE = pixels　HSPACE = pixels

这两个可选属性指定 Applet 的上下和左右的像素数。

9.5.2 Applet 参数

Parameter-define 定义了一系列参数，参数的格式如下：

<PARAM NAME = appletparaname1 VALUE = appletparavalue1>

<PARAM NAME = appletparaname2 VALUE = appletparavalue2>

每个<PARAM>标志只能声明一个参数，这是指定 applet 特定参数的惟一方式，Applet 在源码中可以用 getParameter 方法得到参数值，该方法的入口参数即为这里的参数名，返回值即为这里的参数值；若在页面上的标记中没有找到该参数名，该方法返回空值。

9.5.3 在非 Java 兼容浏览器中显示辅助内容

Applet 只能在 Java 兼容浏览器中执行，但页面并不一定总会运行在 Java 兼容浏览器中。

当非 Java 兼容的浏览器访问含有 Applet 的 Web 页面时，就无法显示正常的 Applet 内容，这时应该给用户以提示，说明这里是一个 Applet；或告诉用户，他的浏览器由于不兼容 Java，因此无法看到应有的内容，但是可以用辅助 Applet 内容来显示这样的描述。

辅助内容是<applet>和</applet>之间除了<param>标志以外的任何 html 正文，Java 兼容浏览器将忽略这些辅助内容。

9.6 Applet 通讯

1 Applet

在同一个 Web 页面中可能有多个 Applet，它们之间通过包 java.applet 中提供的接口 AppletContext 来进行通讯，以获得本页面的其他 Applet 的图形、声音等文件。

通讯的过程是这样的，首先要通过方法 getAppletContext()来获取本页面的 html 环境，返回接口类 AppletContext 的实例，然后通过这个接口提供的方法 getApplet()或 getApplets()获得同页面的其他 Applet 对象，这两个方法定义如下：

```
public Applet getApplet( String name );
public Enumeration getApplets();
```

其中，name 为在<applet>标记中定义的属性 name 给出的 applet 名（见 9.5 节），方法 getApplet()返回本页面中一个属性名与入口参数一致的 applet 对象，若没有这样的 applet 则返回空值。

方法 getApplets()返回 Enumeration 类型的对象，然后可以通过接口 Enumeration 提供的方法 hasMoreElement()和 nextElement()来获得同页面中的所有 Applet 对象。

当一个 Applet 获得其他同页面的 Applet 对象以后，就可以调用它的方法，从而实现对象之间的通讯。

例 9.3 演示了小程序之间的通信，小程序 FirstApplet 和同一个 Web 页面中的其他小程序进行通信。在 FirstApplet 类的 mouseDown 方法中，两个小程序之间进行了通信。首先使用第一种方法，即使用 getApplet(String name)方法，Web 页面中的另外一个小程序名称为 anotherapplet，我们获得它的句柄，并像使用其他的对象一样使用它的方法。这里，小程序 FirstApplet 在小程序 anotherapplet 的区域中写上了一句话。然后使用第二种方法，从浏览器中获得了此 Web 页面中的所有的小程序的句柄，并把它们的名称在小程序 FirstApplet 的显示区域中打印出来。在程序中涉及的事件处理机制，将在第 10 章详细讲述。

【例 9.3】

```
//FirstApplet.java
import java.applet.*;
import java.awt.*;
import java.util.*;
public class FirstApplet extends Applet
{
```

```
    public FirstApplet(){}
    public String getAppletInfo()
    {
        String prop="FirstApplet\r\n";
        return prop;
    }
    public void init()
    {
        resize(450, 250);
    }
    public void destroy(){}
    public void paint(Graphics g){}
    public void start(){}
    public void stop(){}
    public boolean mouseDown(Event evt, int x, int y)
    {
        Font f = new Font("TimesRoman",Font.ITALIC|Font.BOLD,20);
        Graphics g = getGraphics();
        Applet another = getAppletContext().getApplet("anotherapplet");
        if (another!=null)
        {
            int by = another.size().height/2;
            Graphics g2 = another.getGraphics();
            g2.setFont(f);
            g2.drawString("I am class "+getClass().getName()+" in your region.",5,by);
        }
        g.setFont(f);
        Enumeration e = getAppletContext().getApplets();
int i=0;
        while(e.hasMoreElements())
        {
            Object o = e.nextElement();
            g.drawString(o.getClass().getName(),10,(i+1)*40);
            i++;
        }
        return true;
    }
}
```

```
//SecondApplet.java
import java.applet.*;
import java.awt.*;
import java.util.*;
public class SecondApplet extends Applet
{
    public SecondApplet(){}
    public void init()
    {
        resize(450, 250);
    }
    public void destroy(){}
    public void paint(Graphics g){}
    public void start(){}
    public void stop(){}
}

//AppletCommu.html
<html>
<head>
<title>AppletCommu</title>
</head>
<body>
<hr>
<applet
code=FirstApplet.class
name=FirstApplet
width=450
height=250 >
</applet>
<hr>
<applet
code=SecondApplet.class
name=anotherapplet
width=450
height=250 >
</applet>
<a href="FirstApplet.java">The source.</a>
</body>
```

</html>

在小程序 FirstApplet 的区域中用鼠标单击以后运行结果如图 9.3 所示。在小程序 FirstApplet 的区域中显示了在这个 Web 页面中的所有的小程序的类名。小程序 FirstApplet 在小程序 anotherapplet 的区域中显示图 9.3 中的一句话。

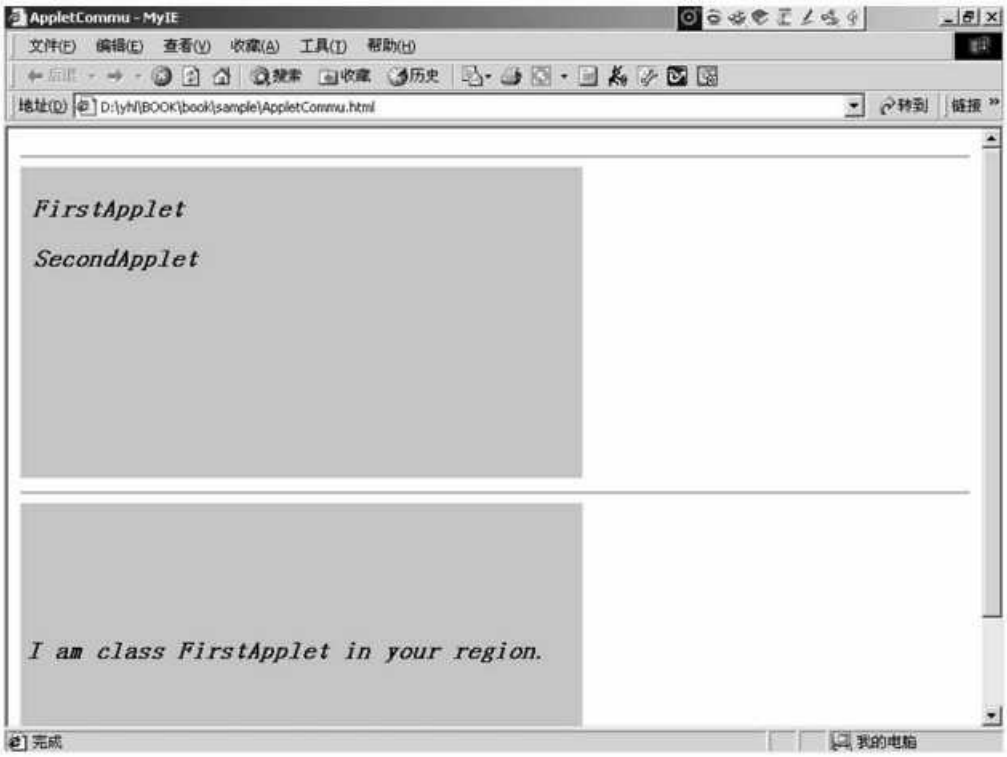


图 9.3 小程序之间的通讯

2 Applet

Applet 可通过管理环境方法和信息报告方法（见 9.4.5 和 9.4.6）与浏览器通讯。下面举一个例子来说明 Apple 是如何与浏览器通讯的。在例 9.4 中，当用鼠标单击时，将打开另一个页面。

【例 9.4】

```
import java.applet.*;
import java.awt.*;
import java.net.*;

public class BrowerComm extends Applet
{
//变量 count 用来记录鼠标点击小程序区的次数
//变量 m_href 用来保存 HTML 页面中参数 href 的实际的值
//字符串常量 PARAM_href 用来保存参数的名称
//字符串变量 data 是要显示在小程序区中的字符串
```

```
private int count=0;
private String m_href = "file:///Dlyhl/BOOK/Thinking in Java/index.htm";
private final String PARAM_href = "href";
private String data="Click here to go to other pages";

public BrowerComm()
{}

public String[][] getParameterInfo()
{
String[][] info ={{ PARAM_href, "String", "a hyper link" },};

return info;
}

public void init()
{
String param;

param = getParameter(PARAM_href);
if (param != null)
m_href = param;

resize(320, 240);
}

public void destroy()
{}

public void paint(Graphics g)
{
g.drawString(data, 10, 20);
}

public void start()
{}

public void stop()
{}
```

```
public boolean mouseDown(Event evt, int x, int y)
{
URL hrefURL = null;

try
{
hrefURL = new URL(m_href);
//在浏览器中显示状态行信息
getAppletContext().showStatus("Loading other pages, please wait...");
//根据单击小程序区次数的不同，用不同的方式显示其他的页面。
switch(count%6)
{
case 0:getAppletContext().showDocument(hrefURL,"_self");
       break;
case 1:getAppletContext().showDocument(hrefURL,"_parent");
       break;
case 2:getAppletContext().showDocument(hrefURL,"_top");
       break;
case 3:getAppletContext().showDocument(hrefURL,"_blank");
       break;
case 4:getAppletContext().showDocument(hrefURL,"NewWindow");
       break;
default:getAppletContext().showDocument(hrefURL);
}

count++;

//改变显示内容
data = "You have click this applet "+count+" times.";

//要求重绘小程序区
repaint();
}
catch(Exception e)
{
System.out.println("Could not go to URL");
}

return true;
```

```
}

public boolean mouseUp(Event evt, int x, int y)
{
return true;
}

}
```

图 9.4 表示用鼠标单击之前，图 9.5 表示用鼠标单击之后。

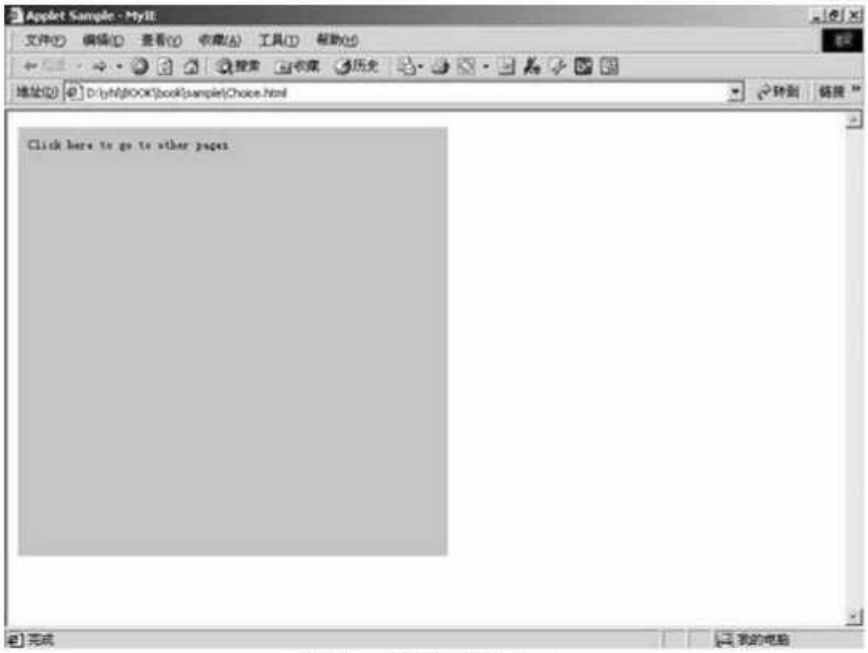


图 9.4 与浏览器通讯（一）

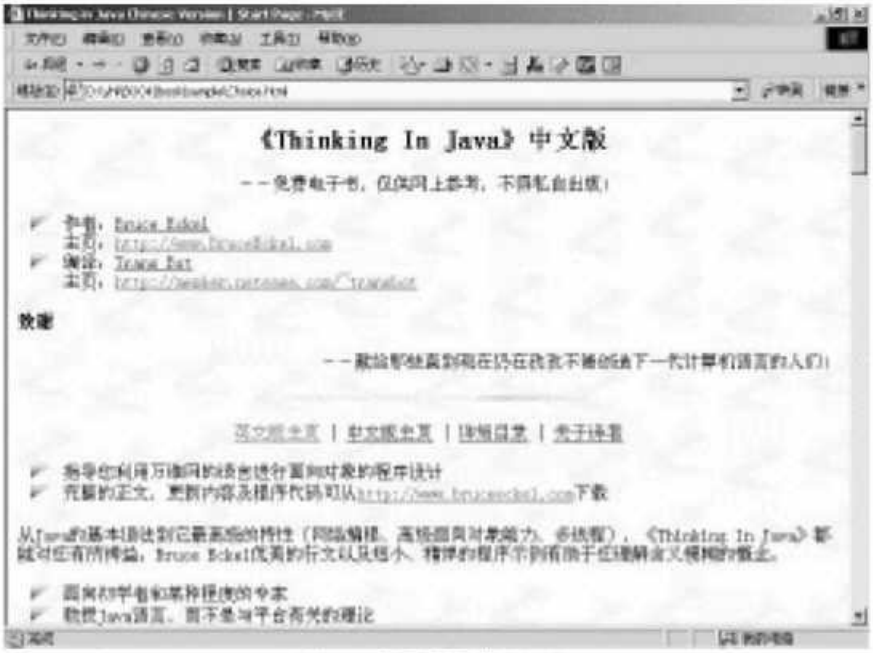


图 9.5 与浏览器通讯（二）

3

出于安全性的考虑，Applet 只能和提供它的服务器进行通讯，由软件包 java.net 中的类提供支持。

9.7 Applet 在安全方面的限制

Applet 作为一种应用于网络的客户端 Java 技术，一个主要目标就是要让用户在自己的浏览器中使用时感到安全，在缺省的情况下，Applet 在安全方面受到很多的限制，比如几乎不能对系统进行任何读写的操作，具体的限制包括下面一些内容。

① Applet 不能读本地系统上的文件。
② Applet 不能创建、修改或删除本地系统的文件。
③ Applet 不能检查本地文件是否存在。
④ Applet 不能在本地系统上创建目录。
⑤ Applet 不能查看本地系统上目录的内容。
⑥ Applet 不能检查文件的大小、类型、访问时间等属性。
⑦ Applet 不能执行本地系统上的任何程序。
⑧ Applet 不能装在动态库或定义本地方法的调用。
⑨ Applet 不能查看本地主机名。
⑩ Applet 不能创建或装载网络连接。

Applet 不能与除了创建它自己的服务器以外的主机进行网络通讯。
Applet 不能充当网络服务器，来监听或接受来自远程系统的连接请求。
Applet 不能操纵不在自己线程组中的任何线程。
Applet 不能关闭 JVM。
Applet 不能创建 SecurityManager 或 ClassLoader 实体。
Applet 不能定义类属于某一包（package）。

9.8 Applet 的用户界面

Web 网页之所以引入 Applet 的一个主要目的是为了实现动态可执行和可交互的内容，因此，用户界面是 Applet 设计的一个重要部分，Applet 通过图形、文本等方式和用户进行交互，显示状态和结果，以及播放各种类型的媒体文件等。

9.8.1 Applet 的 GUI 设计

从 9.2 节可以看到，applet 继承于 java.awt.panel，因此可以通过 AWT 类来实现 applet 的图形用户接口。

有几点要注意的，首先，Applet 是显示在父页面的浏览器窗口中，因此一般不需要自己再产生窗口；其次，Applet 的组件加入要遵循浏览器的窗口实现；再次，在 Applet 的生命周期中，它占用的空间大小是固定的，即在 html 文档中的<applet>标记中的 WIDTH 和 HEIGHT 中规定的大小。

我们前面介绍过 Applet 是在 paint()方法中绘制需要显示的信息，包括文本和图像，paint()方法带有一个 Graphics 类型的参数 g，g 的动作直接在 Applet 窗口中显示，Graphics 类支持两种类型的绘图功能，一是使用绘图原语来画几何图形和字符串，一是显示图像。Java.awt 包中的其他类也提供了一些有用的绘制方法。

1

Graphics 类的绘图原语。

drawLine()：画线段

drawRect()：画矩形

fillRect()：画实心矩形

draw3DRect()：画立体矩形

fill3Drect()：画实心立体矩形

drawOval()：画圆边矩形

fillOval()：画实心圆边矩形

drawArc()：画椭圆

fillArc()：画实心椭圆

drawPloygon()：画多边形

fillPloygon()：画实心多边形

drawBytes()：显示字节

drawChars()：显示字符

drawstring()：显示字符串

这些方法的具体用法可以参见 AWT 的使用手册。

另外，这些方法通常需要窗口大小作为参数，可在 paint()方法中用 size()方法来获得 applet 工作空间的大小，size()方法返回 Rectangle 类型，该类型包含 4 个域：分别是矩形框的 X 坐标、矩形框的 Y 坐标、矩形框的宽度、矩形框的高度。这里 X 坐标和 Y 坐标表示矩形框的左上角那个点，在 applet 内部的坐标系统中，坐标增长方式是 X 从左向右增长，Y 从上向下增长。

可以在 Applet 中设置适当的字体，通过 Graphics 类的 getFont()方法、getFontMetrics()方法和 setFont()方法，来获得当前字体对象和对应于当前字体的 FontMetrics 对象，并设置字体，然后可以调用 Font 类和 FontMetrics 类的方法来获得字体的信息或设置字体的属性。

2

要显示图像首先要获得一个表示图像的 java.awt.Image 类对象，AWT 能识别 GIF 格式和

JPEG 格式的图像文件，当知道该文件的文件名或 URL 后，可以通过 Toolkit 类的 getImage()方法或 Applet 类的 getImage()方法来获得这个对象，然后使用 Graphics 类的 drawImage()方法来显示它，关于这些方法的具体使用，请参看 AWT 的使用手册。

9.8.2 播放声音

在 java.applet 软件包中，Applet 类以及 AudioClip 接口提供了对声音演播的基本支持。方法可见 9.4.4 节中的多媒体支持方法。

在例 9.5 中当鼠标移到图片中时，会播放一段声音。test.gif 和 test.au 是放在本目录下的图片和声音文件。

【例 9.5】

```
import java.applet.*;
import java.awt.*;
import java.net.*;

public class AudioSample extends Applet
{
private String m_picture = "test.gif";
private String m_sound = "test.au";

Image image1;

public AudioSample()
{
}

public void init()
{
//装入图像
MediaTracker tracker = new MediaTracker(this);

try
{
image1 = getImage(getDocumentBase(),m_picture);
tracker.addImage(image1,0);
}catch(Exception e)
{}

try
{
```

```
tracker.waitForID(0);
}catch(Exception e)
{}

resize(280, 200);
}

public void destroy()
{
}

public void paint(Graphics g)
{
//显示图像
g.drawImage(image1,0,0,this);
}

public void start()
{
}

public void stop()
{
}

public boolean mouseEnter(Event evt, int x, int y)
{
//播放声音片段
try
{
play(getDocumentBase(),m_sound);
}
catch(Exception e)
{
System.out.println("Unable to play Sound");
}

return true;
}
```

}

运行结果如图 9.6 所示。

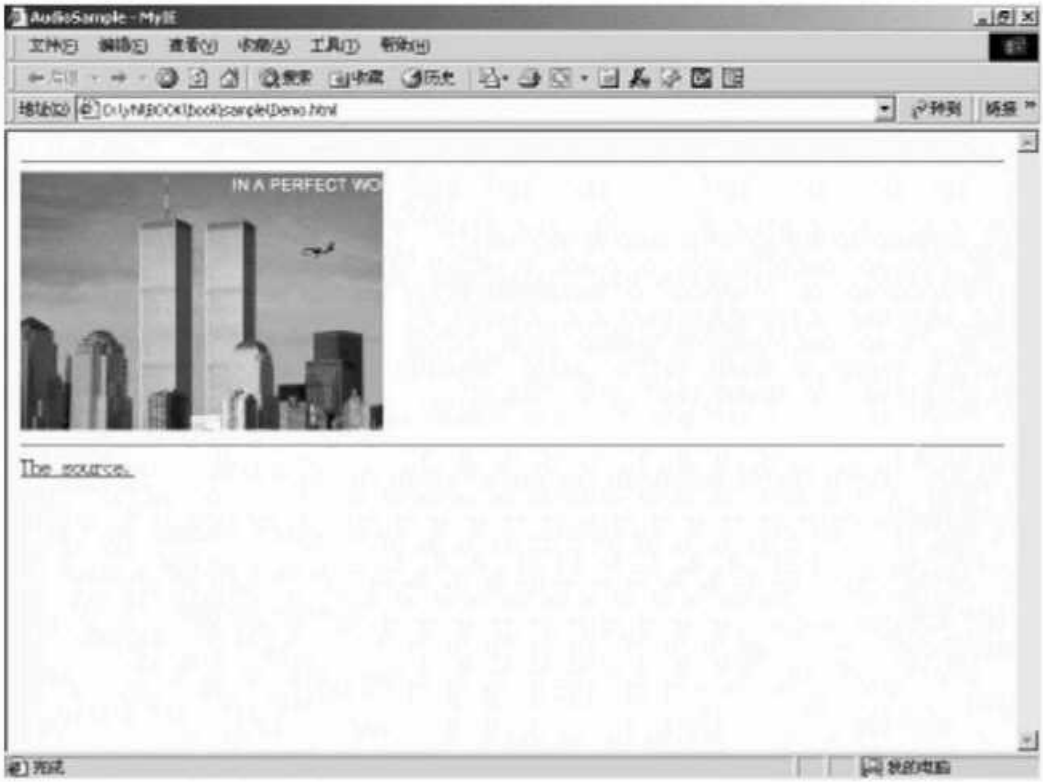


图 9.6　播放声音

第 10 章

图形用户接口

搭建美观明了的图形用户界面是很多交互性应用程序要做的事情，Java 的 GUI 库提供了各种标准部件和事件处理机制以供使用。

本章的主要内容包括:

- 图形用户接口概述
- AWT 简介
- AWT 组件
- AWT 容器与分布管理
- AWT 事件处理机制
- Swing 简介

10.1 图形用户接口概述

通过图形用户接口（Graphic User Interface，GUI），人和计算机程序可以方便友好地进行交互。使用了图形用户界面，人机交互变得更加形象化，也更加方便，人们可以方便地从计算机那里得到信息或者对它进行操作。

在 Java 1.0 中，图形用户接口库最初的设计目标是让程序员构建一个通用的 GUI，使其在所有平台上都能正常显示。它采用面向对象的方法，实现了一组用于图形用户界面设计的类，这些类都包含在称为抽象窗口工具集（Abstract Windowing Toolkit，AWT）的 java.awt 包中。和其他 GUI 工具集一样，AWT 中提供了各种用于 GUI 设计的标准构件，应用程序或小程序可以利用这些构件方便迅速地构造出自己所需的 GUI。但是，为了满足平台无关性的要求，AWT 中没有包含一些应用在特定平台上的 GUI 构件，因此显得有些简单。事实上，Java 1.0 版的 AWT 产生的是在各系统看来都同样欠佳的图形用户接口。除此之外，它还限制只能使用 4 种字体，并且不能访问操作系统中现有的高级 GUI 元素。同时，Jave1.0 版的 AWT 编程模型也不是面向对象的，很不成熟。这类情况在 Java1.1 版的 AWT 事件模型中得到了很好的改进，例如：更加清晰、面向对象的编程、遵循 Java Beans 的范例，以及一个可轻松创建可视编程环境的编程组件模型。Java1.2 为老的 Java 1.0 AWT 添加了 Java 基础类，这是一个被称为“Swing”的 GUI 的一部分。丰富的、易于使用和理解的 Java Beans 能经过拖放操作创建出能使程序员满意的 GUI。

本章主要讲解 Java1.1 版以后的 AWT，先讲解如何使用 AWT 工具，然后在 10.6 节讲解新的能像类库一样加入到 Java 1.1 版中的 JFC/Swing 组件。

10.2 AWT 简介

AWT 提供一套丰富的工具生成平台独立、容易使用的图形用户接口，它主要由组件（components）、容器（Containers）和布局管理（Layouts）3 部分组成，通过事件来表达和处理程序、系统和用户之间的动作与关系。

10.2.1 AWT 类层次

上面说到 AWT 是由组件、容器和布局管理 3 部分组成，除了菜单组件(menu)以外，其余大多数组件都是由 Component 类派生而来，Component 类是一个抽象类，Container 类是 Component 类的一个子类，容器是由 Container 类派生而来。图 10.1 是 AWT 组件的类层次图。

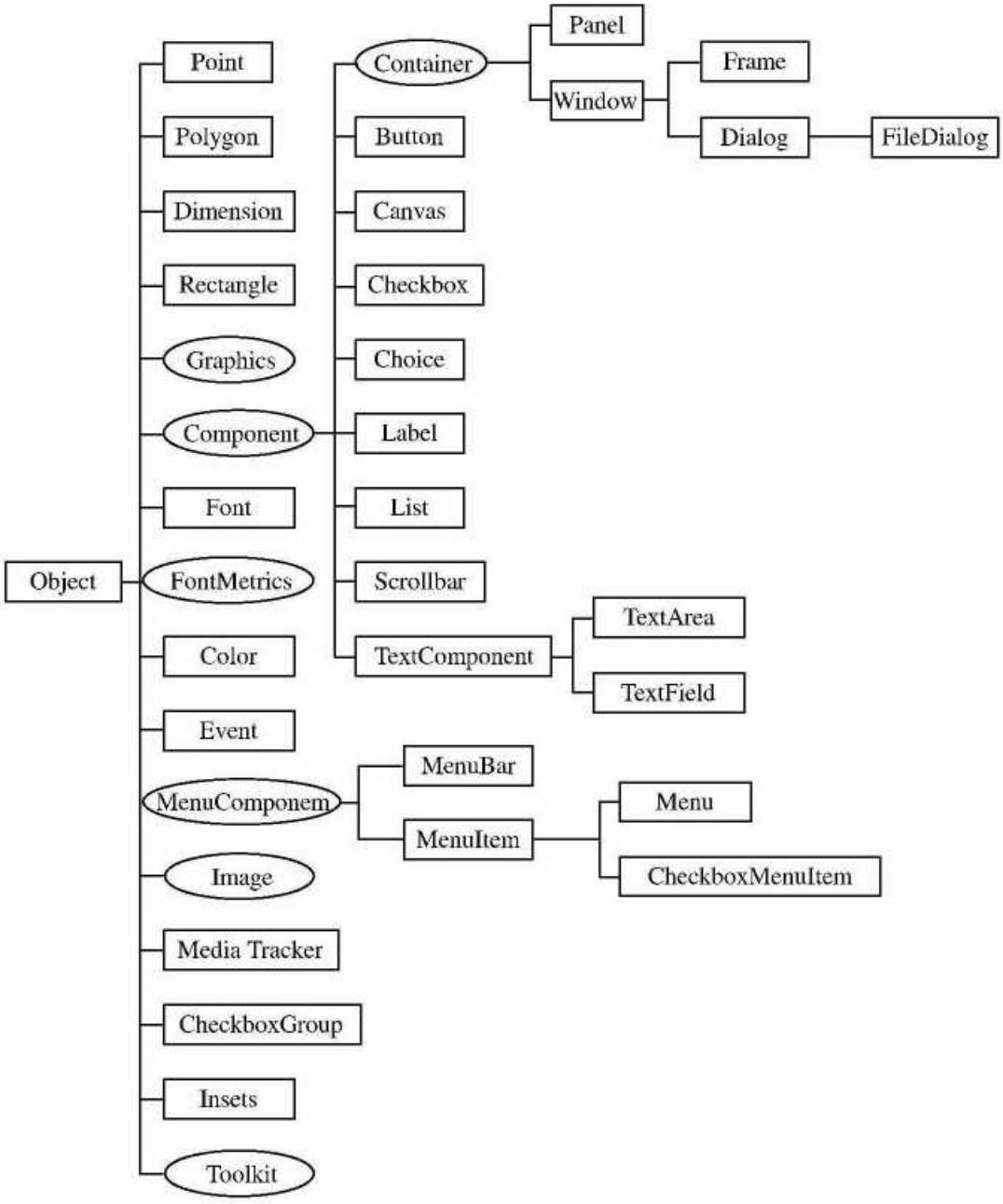


图 10.1　java.awt 包的类层次图

10.2.2　Component 类

Component 类是一个抽象类，它派生出容器类和几乎所有的组件类，它主要提供下面几种方法来支持 AWT 组件的功能。

1

paint()方法、repaint()方法和 update()方法用来在屏幕上绘制组件。其中，repaint()方法直接调用 update()方法，在缺省的 update()方法中，首先清除背景，然后设置前景色，最后调用 paint()方法完成绘制。

2

当用户对某个事件进行操作时，会生成一个事件对象（Event）。在 JDK1.0 中，通过 AWT 事件处理系统，组件得到这个事件并对它进行相应处理。Component 类提供了通用的事件方法 handleEvent()，以及其他一些对动作事件（action()）、鼠标事件(mouseDown())或键盘事件(keyUp())进行处理的方法。JDK1.1 以后的版本使用代理事件模型来处理事件。

3

外形控制包括字型控制和颜色控制。Component 类提供了 getFont()，setFont()，getColorModel()和 SetForeground()等方法。

4

布局管理器是决定组件大小和位置的主要因素，但组件也有一些方法来控制它自己的大小和位置，例如 minimumSize()方法。

5

Component 类的接口 ImageObserver 包含图像处理方法，例如 imageUpdate()方法用来进行图像跟踪等。

6

Component 类还提供一些方法来控制组件的状态，例如 setEnabled()方法控制组件是否能接受用户的输入，isEnabled()方法、isVisible()方法和 isValid()方法等报告组件的当前状态。

10.2.3 Container 类

Container 类是 Component 类的派生类，也是个抽象类，它派生出所有的 AWT 容器。容器既具有组件的功能又可以在其中包含其他的组件，并可以通过布局管理器（LayoutManager）来控制这些组件的大小和功能。每个容器都和一个布局管理器相关联，以确定容器内组件的布局。Container 类的方法可分为以下几种。

1

add()方法用来添加某个组件，remove()方法用来删除某个组件，getComponent()方法用来获得某个组件，countComponent()方法用来计算组件个数，paintComponent()方法用来绘制容器内的所有组件。

2

布局管理器类 LayoutManager 的实例都可以用来为容器作布局管理，它提供 setLayout()方法用来设置某种布局方式，例如 BorderLayout、GridLayout 和 FlowLayout 等。

10.2.4 AWT 程序结构

AWT 的程序分为两个部分，一部分是组件设计，另一部分是事件处理设计。

1

AWT 的组件设计是从高层往低层设置，先定义窗口，设置窗口的属性和窗口布局策略，然后定义窗口中的容器，设置容器的布局管理器，再一一定义容器中的组件，设置它们的属性、大小和位置。

组件设计完后，通过调用 show()方法来显示窗口和其中包含的所有组件。AWT 的显示次序和设计顺序相同，是从最高层组件开始显示，一直到最底层组件，例如先绘制窗口，最后

再一一绘制窗口中的容器，最后再一一绘制容器中的组件，这样做可以保证叶节点上的组件总在最前面。

2

组件设计和显示完后，用户可以看到界面的显示，但还无法和程序交互，这时就要进行事件处理设计。

当用户对一个组件进行某个动作时，例如用鼠标点击，拖动，按“回车”键等，都会产生一个事件对象，AWT 事件处理系统会将事件从该组件往上传递一直传到顶端窗口，这样，该组件和每个包含该组件的容器或窗口都有机会对发生的事件进行响应，AWT 程序就在需要进行相应的组件上添加事件处理方法，例如重新绘制图片、接受数据以及根据用户选择显示动画等，不需要响应该事件的组件就忽略之。

10.3 AWT 组件

10.3.1 基本组件

这里的基本组件是指除了容器类及其子类以外，Component 类派生的其他一些具体类。下面介绍这些组件，并使用 Applet 来演示如何生成和使用它们。

1

按钮（Button）是一个组件，它用于在被按下或被按下再释放时启动相应的操作，按钮可以在声明的时候用字符串来标识（label），字符串在显示上总是位于按钮对象的中央，也可以没有字符串标识。

（1）按钮的构造方法

① **Button**()：创建一个没有字符串标识的按钮。

② **Button**(String name)：创建一个标识为 name 的按钮。

（2）Button 类提供的几个主要方法

① public void setLabel(String newname)：设置新标识为 newname。

② public String getLabel()：获得按钮的标识。

③ public void setEnabled(boolean b)：设置按钮是否能用，true 为可用，false 为禁用，这是 Component 类的方法，Button 类继承使用。请参看例 10.1。

【例 10.1】

```
import java.awt.*;
import java.applet.*;

public class ButtonSample extends Applet {

    public void init() {
        Button button1 = new Button();
        Button button2 = new Button("second");
        button1.setLabel("first");
```

```
        button1.setEnabled(true);
        button2.setEnabled(false);
        add(button1);
        add(button2);
    }
}
```

运行结果如图 10.2 所示。

图 10.2　按钮

2

标签（Label）是最简单的 AWT 组件，用于在容器中加入文本字符串，在界面上不能对它进行操作。标签的位置可以指定为左、右或中间 3 种对齐方式。

（1）标签类的构造方法

① **Label**()：声明一个空白标签。

② **Label**(String name)：声明一个显示为 name 的标签。

③ **Label**(String name, int alignment)：声明一个显示为 name 的标签，对齐方式为 alignment，取值为 Label.LEFT、Label.RIGHT 和 Label.CENTER 3 个常量中的一个。

（2）Label 类提供的主要方法

① public void **setText**(String text)：设置标签内容。

② public void **setAlignment**(int alignment)：设置标签位置。

例 10.2 是个标签的小例子。

【例 10.2】

```
import java.awt.*;
import java.applet.*;

public class LabelSample extends Applet
{
    public void init() {
        Label label1 = new Label();
        Label label2 = new Label("second");
        Label label3 = new Label("third",Label.RIGHT);
        label1.setText("first");
        label1.setAlignment(Label.LEFT);
```

```
        label2.setAlignment(Label.CENTER);
        add(label1);
        add(label2);
        add(label3);
    }
}
```

运行结果如图 10.3 所示。注意，由于这里设置了对齐方式，所以 3 个标签的添加顺序无所谓，也就是说，不管先添加哪个标签，后添加哪个标签，运行结果都和图 10.3 所示的一样。

图 10.3　标签

3

画布（Canvas）是 Component 的简单子类，它与 Component 完全相同，除了它是个具体类以外，画布还可以看成是一个空白的矩形区域，可以将任何的组件和元素放在它上面，或者在它上面作图。在画布上面作图是在 paint()方法中实现的。

4

校验框（Checkbox）是一种处于两个状态（真或假）之一的按钮。从界面上看，一个 Checkbox 由一个字符串类型的标签和一个可供选择“true/false”的小按钮组成，以便用鼠标单击进行选择。

（1）Checkbox 类提供的构造方法

① Checkbox()：创建一个无标签的校验框。

② Checkbox(String)：创建一个标签为 String 的校验框，状态缺省为 false。

③ Checkbox(String, CheckboxGroup, boolean)：创建一个标签为 String，状态为 boolean，属于校验框组 CheckboxGroup 的校验框，如果 CheckboxGroup 为 null，则是独立的校验框。

校验框可以看作 VC 中的复选框，另一个类**校验框组**（CheckboxGroup）则可以看成 VC 中的单选框，首先声明一个校验框组对象，然后用 Checkbox(String, CheckboxGroup, boolean)声明若干个包含在这个校验框组中的校验框，则这几个校验框就是互相排斥的，每次只能有一个为“真”，即只能选择一个。

（2）Checkbox 类提供的主要方法

public boolean **getState**()：获取校验框的状态。

例 10.3 是一个校验框的例子。

【例 10.3】

```
import java.awt.*;
import java.applet.*;

public class CheckboxSample extends Applet
{
    public void init()
{
        Checkbox checkbox1 = new Checkbox();
        Checkbox checkbox2 = new Checkbox("check2");
        Checkbox checkbox3 = new Checkbox("check3", null, true);
        CheckboxGroup group1 = new CheckboxGroup();
        Checkbox checkbox5 = new Checkbox("check5", group1, true);
        Checkbox checkbox6 = new Checkbox("check6", group1, false);
        add(checkbox1);
        add(checkbox2);
        add(checkbox3);
        add(checkbox5);
        add(checkbox6);
    }
}
```

运行结果如图 10.4 所示。

图 10.4　校验框

5

选择框（Choice）是一个类似菜单的下拉选择列表，给用户提供文本字符串的选择，通过单击选择框上的小按钮来列出所有条目并选取，当前显示的字符串即为当前的选择项。

（1）Choice 类的构造方法

Choice()：声明一个空白的选择框。

（2）Choice 类提供的主要方法

① public void addItem(String name)：添加一个选择项名字为 name，根据添加顺序，每个选择项都有一个索引序号，从零开始。

② public void select(String name)：选择名字为 name 的条目。
③ public void select(int index)：选择索引为 index 的条目。
④ public int getItemCount()：返回选择项总数。
⑤ public String getItem(int index)：返回第 index 条的选择项的名字。
⑥ public String getSelectedItem()：返回当前选择项的名字。
⑦ public int getSelectedIndex()：返回当前选择项的序号。
⑧ public void remove(int index)：删除第 index 条选择项。

例 10.4 是个 Choice 的例子。

【例 10.4】

```
import java.awt.*;
import java.applet.*;

public class ChoiceSample extends Applet
{
    public void init()
{
        Choice city = new Choice();
        Choice occupation = new Choice();
        city.addItem("Beijing");
        city.addItem("Tianjin");
        city.addItem("Shanghai");
        occupation.addItem("teacher");
        occupation.addItem("executive");
        occupation.addItem("programer");
        occupation.addItem("businessman");
        occupation.select("programer");
        add(city);
        add(occupation);
    }
}
```

运行结果如图 10.5 和图 10.6 所示，分别是初始状态和单击状态。

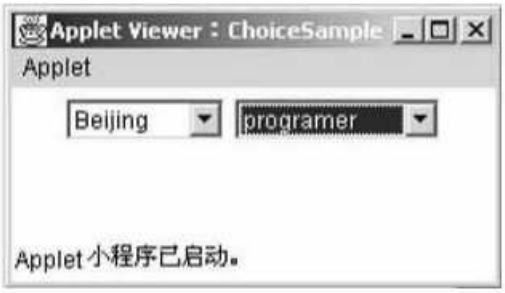


图 10.5　选择框

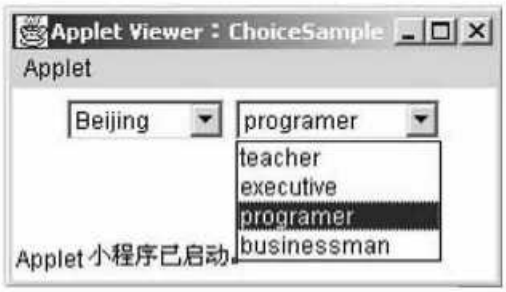


图 10.6　打开的选择框

6

列表(List)的功能类似于选择框，所不同的是，它可以一次选择多个条目，而且可选择的条目都是可见的，并且列在一个滚动区域内。

（1）List 类的构造方法

① **List()**：声明一个可滚动的、只能做单项选择的空列表，显示 4 个列表项。

② **List**(int number, boolean multipleMode)：声明一个可滚动的显示特定个数列表项的列表，第一个参数表示显示几个列表项，若为 0，则显示 4 个列表项；第二个参数表示可不可以做多项选择，true 表示可以多选。

（2）List 类提供的主要方法

① public void **add** (String item)：增加一个选择项在列表底部。

按添加顺序，每个列表位置都有一个索引，从 0 开始。

② public void **add** (String item, int index)：在位置 index 处增加一个选择项，若 index 非法（−1 或大于当前最大位置），则添加在列表底部。

③ public void **replaceItem**(String newitem, int index)：替换位置 index 处的选择项。

④ public void **remove**(int index)：删除位置 index 处的选择项。

⑤ public void **remove**(String item)：删除选择项 item。

⑥ public void **removeAll**()：删除列表中所有的选择项。

⑦ public int **getSelectedIndex**()：获得当前选取的选择项的 index 号。

⑧ public String **getSelectedItem**()：获得当前选取的选择项。

⑨ public Int[] **getSelectedIndexes**()：获得当前选取的多个选择项的 index 号。

⑩ public String[] **getSelectedItems**()：获得当前选取的多个选择项。

public void **select**(int index)：选取第 index 个选择项。

public void **deselect**(int index)：确保不选取第 index 个选择项。

public boolean **isIndexSelected**(int index)：判断第 index 个选择项是否被选取。

public void **setMultipleSelections**(boolean multipleMode)：设置是否能做多项选择。

public boolean **isMultipleMode**()：判断是否能做多项选择。

public void **makeVisible**(int index)：确保第 index 个选择项在列表窗口可见。

例 10.5 是一个列表类的例子

【例 10.5】

```
import java.awt.*;
import java.applet.*;

public class ListSample extends Applet
{
    public void init()
{
        List city = new List(0, true);
        List occupation = new List(0, false);
        city.add("Beijing");
```

```
        city.add("Tianjin");
        city.add("Shanghai");
        occupation.add("teacher");
        occupation.add("executive");
        occupation.add("programer");
        occupation.add("businessman");
        occupation.select(3);
        add(city);
        add(occupation);
    }
}
```

运行结果如图 10.7 和图 10.8 所示，图 10.7 是初始化时，图 10.8 显示了 city 可多选，而 occupation 只能单选。

图 10.7　列表（一）

图 10.8　列表（二）

7

滚动条(Scrollbar)类似一个滑尺，可用来设置数值，另外，当显示区域无法显示全部内容时，也可以使用滚动条来显示内容的一部分，如上面介绍的 List 类，在需要时自动出现滚动条。

（1）Scrollbar 的构造方法

① **Scrollbar**()：声明一个垂直方向的滚动条。

② **Scrollbar**(int orientation)：声明一个指定方向的滚动条。Scrollbar.VERTICAL 表示垂直方向的滚动条，Scrollbar.HORIZONTAL 表示水平方向的滚动条。

③ **Scrollbar**(int orientation, int value, int visible, int minimum, int maximum)：声明一个指定方向、显示位置、页面大小、最小值、最大值的滚动条。

（2）Scrollbar 提供的主要方法

① public int **getValue**()：获得滚动条当前的数值。

② public void **setValue**(int newvalue)：设置新数值。

③ public void **setValues**(int value, int visible, int minimum, int maximum)：设置滚动条的数

值、页面大小、最小值和最大值。

④ public void **setMaximum**(int newMaximum)：设置新的最大值。

⑤ public void **setMinimum**(int newMinimum)：设置新的最小值。

下面是一个滚动条的例子，其中设置布局方法 setLayout()将在后面的小节讲到。

运行结果如图 10.9 所示。

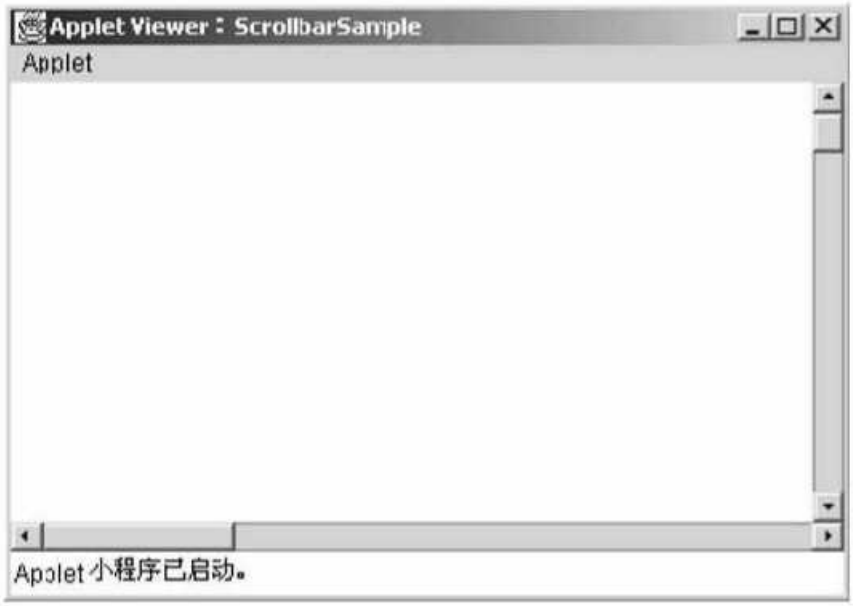


图 10.9　滚动条

例 10.6 中是源代码。

【例 10.6】

```
import java.awt.*;
import java.applet.*;

public class ScrollbarSample extends Applet
{
    public void init()
{
        setLayout(new BorderLayout());
        Scrollbar bar1 = new Scrollbar(Scrollbar.VERTICAL, 0, 5, 0, 50);
        Scrollbar bar2 = new Scrollbar(Scrollbar.HORIZONTAL, 0, 25, 0, 100);
        add("East",bar1);
        add("South",bar2);
        validate();
    }
}
```

8

文本区（TextField 和 TextArea）。TextField 是单行文本区，TextArea 是多行文本区，它们都是 TextComponent 类的子类。TextArea 自动支持滚动条。

（1）TextField 的构造方法

① **TextField**()：声明一个空文本区。

② **TextField**(String text)：声明一个文本区，内容为 text。

③ **TextField**(int length)：声明一个长度为 length 的文本区。

④ **TextField**(String text, int length)：声明一个长度为 length、内容为 text 的文本区。

（2）TextField 类提供的主要方法

① public void **setText**(String text)：设置文本区的内容。

② public void **setEchoChar**(Char c)：设置显示的响应字符。

③ public void **setEditable**(boolean)：设置是否能编辑。

④ public void **setColumns**(int columns)：设置长度。

⑤ public String **getText**()：获得文本区所有的内容。

⑥ public void **select**(int start, int end)：选择字符位置从 start 到 end 的文本内容。

⑦ public void **selectAll**()：选择所有的文本内容。

⑧ public void **getSelectedText**()：获得选择的文本内容。

（3）TextArea 的构造方法

① **TextArea**()：声明一个空文本区。

② **TextArea**(int rows, int columns)：声明一个行数为 rows、列数为 columns 的文本区。

③ **TextArea**(String text)：声明一个内容为 text 的文本区。

④ **TextArea**(String text, int rows, int columns)：声明一个内容为 text、行数为 rows、列数为 columns 的文本区。

⑤ **TextArea**(String text, int rows, int columns, int scrollbars)：声明一个内容为 text、行数为 rows、列数为 columns 的文本区。滚动条方式为指定方式，有以下几种。

- SCROLLBARS_BOTH：水平垂直都有滚动条。
- SCROLLBARS_HORIZONTAL_ONLY：只有水平滚动条。
- SCROLLBARS_VERTICAL_ONLY：只有垂直滚动条。
- SCROLLBARS_NONE：没有滚动条。

（4）TextArea 类提供的主要方法

① public void **setText**(String text)：设置文本区的内容。

② public void **setEditable**(boolean editMode)：设置是否能编辑。

③ public void **setColumns**(int columns)：设置列数。

④ public void **setRows**(int rows)：设置行数。

⑤ public void **appendText**(String text)：添加字符串。

⑥ public String **getText**()：获得文本区所有的内容。

⑦ public void **select**(int start, int end)：选择字符位置从 start 到 end 的文本内容。

⑧ public void **selectAll**()：选择所有的文本内容。

⑨ public void **getSelectedText**()：获得选择的文本内容。

例 10.7 是一个文本区的例子。

【例 10.7】

```
import java.awt.*;
import java.applet.*;

public class TextSample extends Applet
{
    public void init()
    {
        Label notice1 = new Label("please fill in the city you live:");
        TextField city = new TextField(10);
        Label notice2 = new Label("please fill in the your occupation:");
        TextField occupation = new TextField(10);
        TextArea comment = new TextArea("Please give some comment here:",5,50);

        add(notice1);
        add(city);
        add(notice2);
        add(occupation);
        add(comment);
    }
}
```

运行结果如图 10.10 所示。

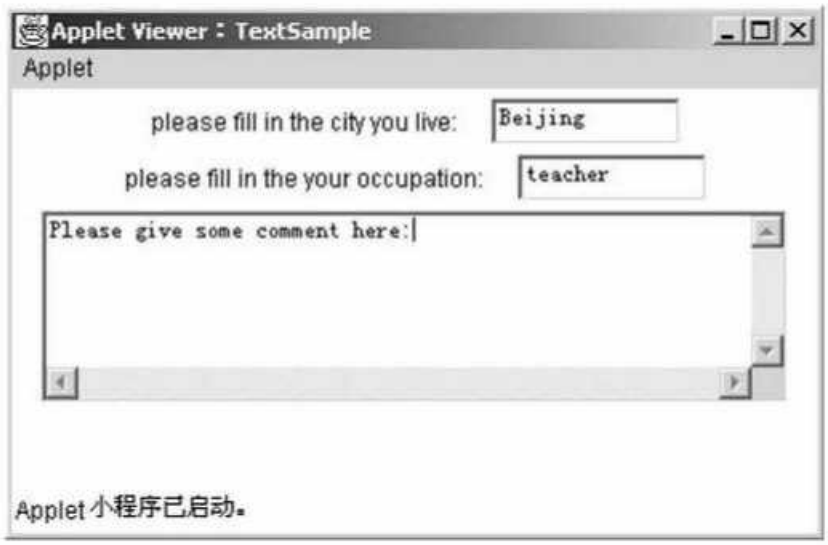


图 10.10　文本区

10.3.2 菜单

菜单作为一种独立的组件只能放在窗口容器上，为用户提供多种功能的选择和交互。菜单(Menu)和前面那些组件不同，它是由菜单条(Menubar)和菜单项(MenuItem)组成的，类的层次结构如图 10.11 所示（其中 CheckboxMenuItem 是一种类似于检查盒的菜单项）。

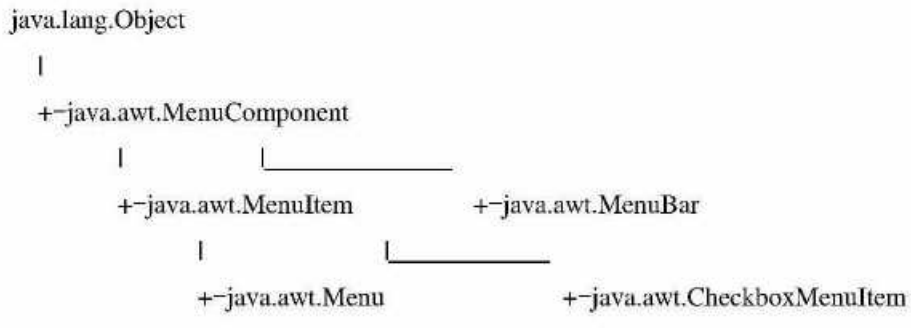


图 10.11 类的层次结构

从图 10.11 可以看出，基类是 MenuComponent，它派生出 MenuItem 和 MenuBar，MenuItem 派生出 Menu 和 CheckboxMenuItem。从运行后生成的菜单界面上看，菜单条（MenuBar）包含一个个的下拉菜单（Menu），每个菜单中包含一个个的菜单项（MenuItem or CheckboxMenuItem）和分割线，也可以包含下一级的下拉菜单。由于菜单不是从 Component 类派生而来的，因此不能用 add 方法在容器中添加菜单组件，为了能包含 MenuComponent 对象，Menu 和 MenuBar 都实现了一个 MenuContainer 接口，在后面讲到的 Frame 类也实现了这个接口，然后使用 setMenuBar 方法来添加菜单条。下面我们通过例 10.8 中的程序来说明菜单组件的使用。

【例 10.8】

```
import java.awt.*;
import java.applet.*;

public class MenuSample extends Applet
{
    public void init()
{
        int width = Integer.parseInt(getParameter("width"));
        int height = Integer.parseInt(getParameter("height"));
        Frame f = new Frame("FrameSample"); //声明一个框架来包含菜单
        f.setSize(width, height);

        MenuBar bar = new MenuBar();
        f.setMenuBar(bar); //在框架中设置菜单条
```

```
        Menu filemenu = new Menu("File"); //声明一个显示内容为 File 的下拉菜单
        filemenu.add(new MenuItem("New"));  //在菜单中添加一个菜单项
        filemenu.add(new MenuItem("Open"));
        filemenu.add(new MenuItem("Close"));
        filemenu.addSeparator(); //添加分割线
        filemenu.add(new MenuItem("Print"));
        bar.add(filemenu); //将下拉菜单 File 添加到菜单条中

        Menu editmenu = new Menu("Edit");
        editmenu.add(new MenuItem("Cut"));
        editmenu.add(new MenuItem("Copy"));
        editmenu.add(new MenuItem("Paste"));
        editmenu.addSeparator();
        Menu deletemenu = new Menu("Delete");
        deletemenu.add(new MenuItem("Delete"));
        MenuItem oneline = new MenuItem("Delete one line");
        oneline.setEnabled(false); //设置菜单项不激活
        deletemenu.add(oneline);
        MenuItem tohead = new MenuItem("Delete to head");
        tohead.setEnabled(true); //设置菜单项激活
        deletemenu.add(tohead);
        MenuItem totail = new MenuItem("Delete to tail");
        totail.setEnabled(true);
        deletemenu.add(totail);
        editmenu.add(deletemenu); //在菜单 Edit 中添加了菜单 Delete
        bar.add(editmenu);

        f.show();  //显示包含菜单的框架 s
    }
}
```

运行结果如图 10.12 和图 10.13 所示。

从以上程序中，我们可以总结出如何使用菜单组件。首先要生成一个框架，然后生成一个菜单条添加到框架上；随后在菜单条上逐一添加菜单，在菜单上逐一添加菜单项或子菜单；最后显示该框架。

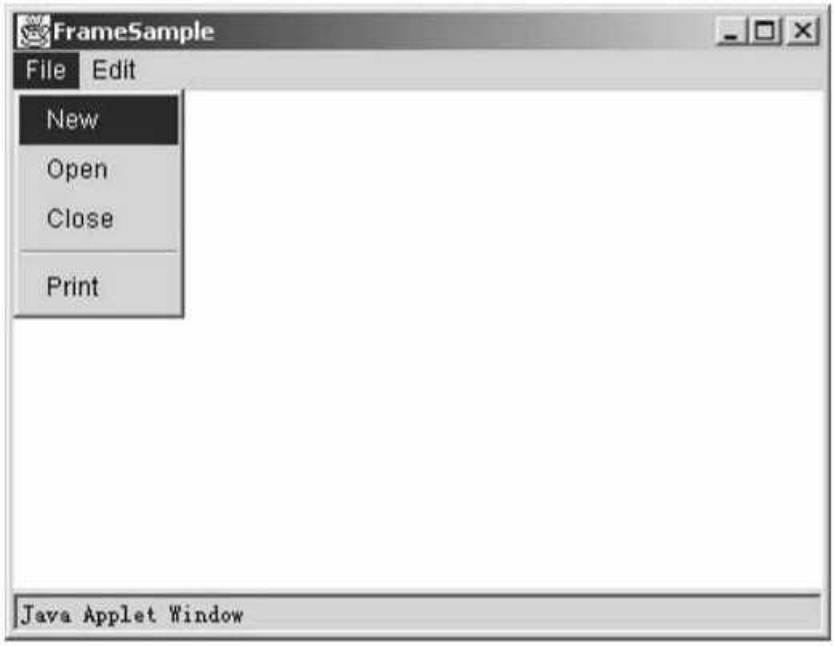


图 10.12　菜单（一）

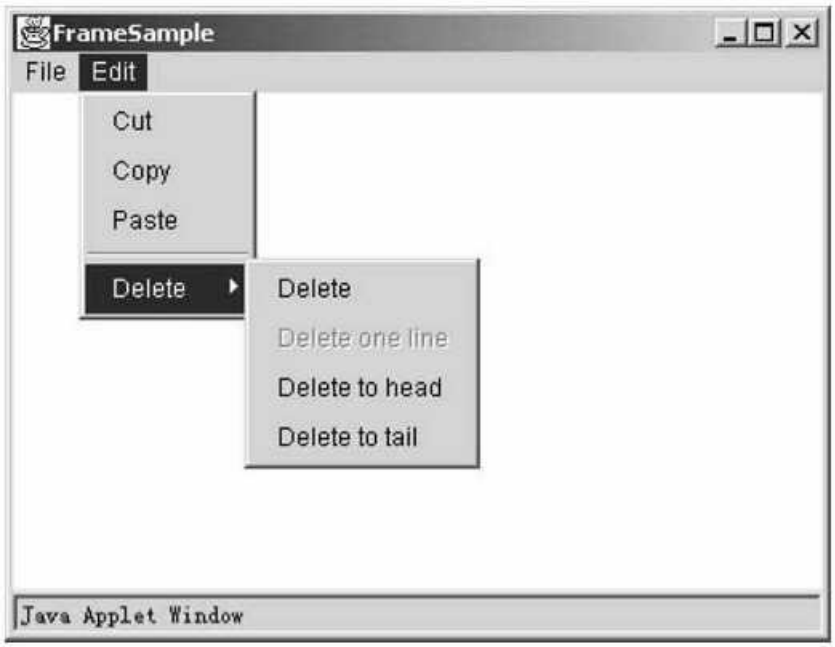


图 10.13　菜单（二）

10.4　AWT 容器与布局管理

在 10.2 讲到，AWT 容器主要用来容纳 AWT 组件，由该容器的布局管理器来决定容器中组件的大小和位置。

10.4.1 容器

AWT 容器是指能够容纳 AWT 组件的特殊组件，由于容器也是组件的一种，因此容器中也能容纳容器。容器组件既可以使用缺省的布局管理器，也可以通过 setLayout 方法来设置所需要的布局管理器。一旦确定了布局管理方式，容器组件就可以使用相应的 add 方法向其中加入其他 GUI 组件。本节讲述的是几种重要的容器类。

1　　Panel

面板不能单独存在，它必须包含在其他容器内，而且它是没有外观的，但它却是十分重要的容器，用来实现各种基本组件的组合，面板中也能放面板。Panel 的构造方法如下述。

① **Panel**()：构造一个空白面板，使用缺省的布局管理器 FlowLayout。

② **Panel**(LayoutManager layout)：构造一个指定布局管理器的面板。

构造完面板后，使用面板对象的 add()方法添加组件即可。

由于 Applet 就是 Panel 的子类，所以在前面的例子中，可以直接在 Applet 中使用 add()方法添加各种基本组件，就如同在 Panel 中添加一样。

2　　Windows

窗口类是最高一级的容器，这就是说它是独立存在的顶层容器，不能包含在其他容器之中。当一个窗口生成时是不可见的，需要通过调用方法 show()来显示它。

MenuContainer

Frame

3.　　Frame

框架类是窗口的子类，是一个功能完善的顶层容器。它可重定义尺寸大小，能最大化最小化，带边框，可以指定标题、图标和窗口光标。框架类可包含：菜单条，使用对话框，而且框架是使用这两种组件的惟一途径。框架缺省的布局管理器是 BorderLayout。

show()

Frame 的构造方法如下述。

① **Frame**()：构造一个空白框架，使用缺省布局管理器。

② **Frame**(GraphicsConfiguration gc)：构造一个使用特定图形配置的框架。

③ **Frame**(String title)：构造一个带标题的框架。

④ **Frame**(GraphicsConfiguration gc, String title)：构造一个使用特定图形配置带标题的框架。

4.　　Dialog

对话框也是窗口的子类，主要用于实现应用程序与用户的交流。与框架和窗口不同的是，

对话框必须由框架类或父对话框生成，而且当它的父框架或父对话框被关闭或最小化时，它也随之消失。对话框的缺省布局管理器是 BorderLayout。Dialog 的构造方法如下述。

① **Dialog**(Dialog parent)：在父对话框上构造一个对话框。

② **Dialog**(Dialog parent, String title)：在父对话框上构造一个标题为 title 的对话框。

③ **Dialog**(Dialog parent, String title, Boolean modal)：在父对话框上构造一个标题为 title 的对话框，并指定 modal 方式。上面两种构造方式中，对话框缺省为 modeless，当 modal 指定为 true 时，表示是一个 modal dialog，在该对话框出现时，用户不能对它以外的窗口进行操作，直到该对话框被关闭。当然，若该对话框又生成了子对话框和子窗口，用户是可以对这些子对话框和子窗口操作的。

④ **Dialog**(Frame parent)：在父框架上构造一个对话框。

⑤ **Dialog**(Frame parent, String title)：在父框架上构造一个带标题的对话框。

⑥ **Dialog**(Frame parent, String title, Boolean modal)：在父框架上构造一个带标题并指定 modal 方式的对话框。

5. FileDialog

文件对话框是对话框的子类，它的属性为 modal。用来让用户选择载入（Load）一个文件或保存（Save）一个文件。FileDialog 的构造方法如下述。

① **FileDialog**(Frame parent)：在父框架上构造一个文件对话框。

② **FileDialog**(Frame parent, String title)：在父框架上构造一个带标题的文件对话框。

③ **FileDialog**(Frame parent, String title, int mode)：在父框架上构造一个带标题的文件对话框，并指定模式，Load 表示是载入文件，Save 表示保存文件。

10.4.2 布局管理

AWT 提供了以下 5 种标准布局管理方式，即 5 个 LayoutManager 的实现类，为了形象起见，我们使用 SUN 公司的一个示例程序（例 10.9）来进行演示，程序源代码如下述，有兴趣的读者可以自己运行查看结果。

【例 10.9】

```
/*
 * @(#)CardTest.java 1.6 99/07/12
 *
 * Copyright (c) 1997 Sun Microsystems, Inc. All Rights Reserved.
 *
*/
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;

class CardPanel extends Panel {
    ActionListener listener;
```

```
Panel create(LayoutManager layout) {
  Button b = null;
  Panel p = new Panel();

  p.setLayout(layout);

  b = new Button("one");
  b.addActionListener(listener);
  p.add("North", b);

  b = new Button("two");
  b.addActionListener(listener);
  p.add("West", b);

  b = new Button("three");
  b.addActionListener(listener);
  p.add("South", b);

  b = new Button("four");
  b.addActionListener(listener);
  p.add("East", b);

  b = new Button("five");
  b.addActionListener(listener);
  p.add("Center", b);

  b = new Button("six");
  b.addActionListener(listener);
  p.add("Center", b);

  return p;
}

CardPanel(ActionListener actionListener) {
  listener = actionListener;
  setLayout(new CardLayout());
  add("one", create(new FlowLayout()));
  add("two", create(new BorderLayout()));
  add("three", create(new GridLayout(2, 2)));
```

```
        add("four", create(new BorderLayout(10, 10)));
        add("five", create(new FlowLayout(FlowLayout.LEFT, 10, 10)));
        add("six", create(new GridLayout(2, 2, 10, 10)));
    }

    public Dimension getPreferredSize() {
        return new Dimension(200, 100);
    }
}

public class CardTest extends Applet    implements ActionListener, ItemListener {
    CardPanel cards;

    public CardTest() {
        setLayout(new BorderLayout());
        add("Center", cards = new CardPanel(this));
        Panel p = new Panel();
        p.setLayout(new FlowLayout());
        add("South", p);

        Button b = new Button("first");
        b.addActionListener(this);
        p.add(b);

        b = new Button("next");
        b.addActionListener(this);
        p.add(b);

        b = new Button("previous");
        b.addActionListener(this);
        p.add(b);

        b = new Button("last");
        b.addActionListener(this);
        p.add(b);

        Choice c = new Choice();
        c.addItem("one");
        c.addItem("two");
```

```
 c.addItem("three");
 c.addItem("four");
 c.addItem("five");
 c.addItem("six");
 c.addItemListener(this);
 p.add(c);
}

public void itemStateChanged(ItemEvent e) {
 ((CardLayout)cards.getLayout()).show(cards,   (String)(e.getItem()));
}

public void actionPerformed(ActionEvent e) {
 String arg = e.getActionCommand();

 if ("first".equals(arg)) {
       ((CardLayout)cards.getLayout()).first(cards);
 } else if ("next".equals(arg)) {
       ((CardLayout)cards.getLayout()).next(cards);
 } else if ("previous".equals(arg)) {
       ((CardLayout)cards.getLayout()).previous(cards);
 } else if ("last".equals(arg)) {
       ((CardLayout)cards.getLayout()).last(cards);
 } else {
       ((CardLayout)cards.getLayout()).show(cards,(String)arg);
 }
}

public static void main(String args[]) {
 Frame f = new Frame("CardTest");
 CardTest cardTest = new CardTest();
 cardTest.init();
 cardTest.start();

 f.add("Center", cardTest);
 f.setSize(300, 300);
 f.show();
}
```

```
    public String getAppletInfo() {
        return "Demonstrates the different types of layout managers.";
    }
}
```

1　FlowLayout

流式布局管理（FlowLayout）是 AWT 中最简单的一种布局管理方法。顾名思义，它是以一种流程式的方式，自左向右、自上而下地布置容器中所包含的 GUI 组件。开发人员只需调用 setLayout 方法将容器组件的布局管理方式设置为流式布局管理，然后调用 add（Component comp）方法将组件加入到容器中即可。Panel 缺省的布局管理方式就是 FlowLayout。FlowLayout 可以指定布局对齐方式，Center 表示每一行都在窗口中间，这是缺省对齐方式，Left 表示每一行左对齐，Right 表示每一行右对齐，Leading 表示每一行对齐容器上面的边，TraiLing 表示每一行对齐容器下面的边。FlowLayout 还可以指定组件间的间隙，若不指定，缺省间隙为 5。

FlowLayout 的构造方法如下述。

① FlowLayout()：构造一个 FlowLayout 布局管理器。

② FlowLayout(int align)：构造一个 FlowLayout 布局管理器，并指定它的对齐方式。

③ FlowLayout(int align, int hgap, int vgap)：构造一个 FlowLayout 布局管理器，指定它的对齐方式，并指定组件间的水平间隙和垂直间隙。

如图 10.14 是 FlowLayout 布局管理器左对齐的样本。

图 10.14　FlowLayout 样本

2　BorderLayout

BorderLayout 用类似于地理区域的方式管理 GUI 组件的布局，这 5 个区域是东（East）、西（West）、南（South）、北（North）和中（Center）。

用户可以向这 5 个区域中加入相应的 GUI 组件。BorderLayout 是 Frame 和 Dialog 的缺省布局管理方式。与 FlowLayout 不同的是，如果使用 BorderLayout 进行布局管理，那么在加入 GUI 组件的时候，就需要明确指出加入的位置，例如：

```
setLayout( new BorderLayout() );
```

add("Center", new Button("Button"));

BorderLayout 的构造方法如下述。

① BorderLayout ()：构造一个 BorderLayout 布局管理器，缺省间隙为 0。

② BorderLayout (int hgap, int vgap)：构造一个 BorderLayout 布局管理器，并指定组件间的水平间隙和垂直间隙。

图 10.15 是 BorderLayout 布局管理器样本，图 10.16 是指定了间隔的 BorderLayout。

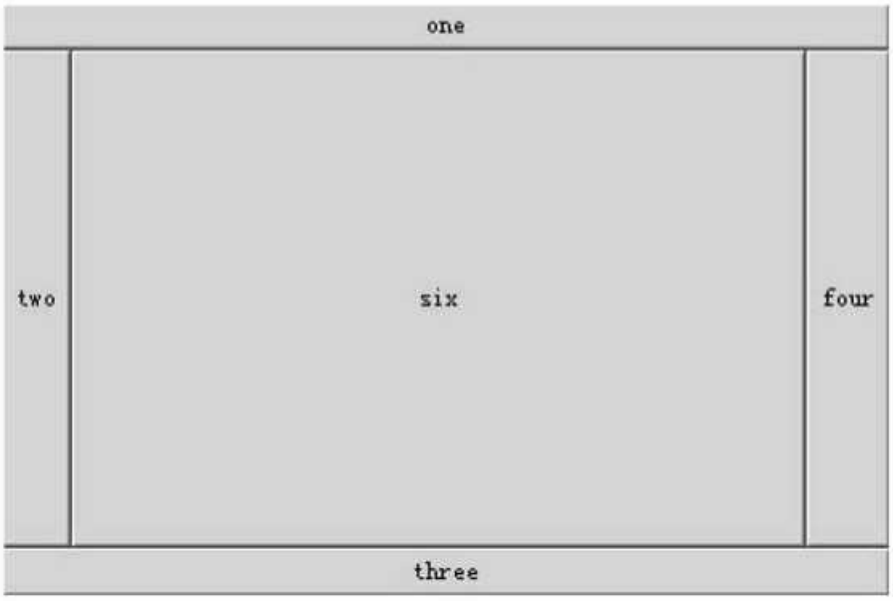


图 10.15　BorderLayout 样本（一）

图 10.16　BorderLayout 样本（二）

3　CardLayout

CardLayout 提供了一种基于卡片式的布局管理方式。在 AWT 应用中，可以将某一 GUI 组件加入到一个指定名字的卡片中，例如：

```
setLayout( new CardLayout() );
add("CardName", new Button("Card 1") );
```

这里，卡片名字是惟一的，可用于引用相应的卡片。一般而言，加入到卡片中的 GUI 组件通常是 Panel 对象，因为 Panel 对象又可以包含其他 GUI 组件，并按照其自身的布局管理方式来管理它们。为便于在一组卡片之间来回切换，CardLayout 定义了一系列方法，如 first、next、previous、show 等。

CardLayout 的构造方法如下述。

① CardLayout ()：构造一个 CardLayout 布局管理器，缺省间隙为 0。

② CardLayout (int hgap, int vgap)：构造一个 CardLayout 布局管理器，并指定组件间的水平间隙和垂直间隙。

在程序中，6 种布局形成一组 6 张的卡片，可以通过界面提供的按钮 first、next、previous,last 和选择框来回切换，如图 10.17 所示。

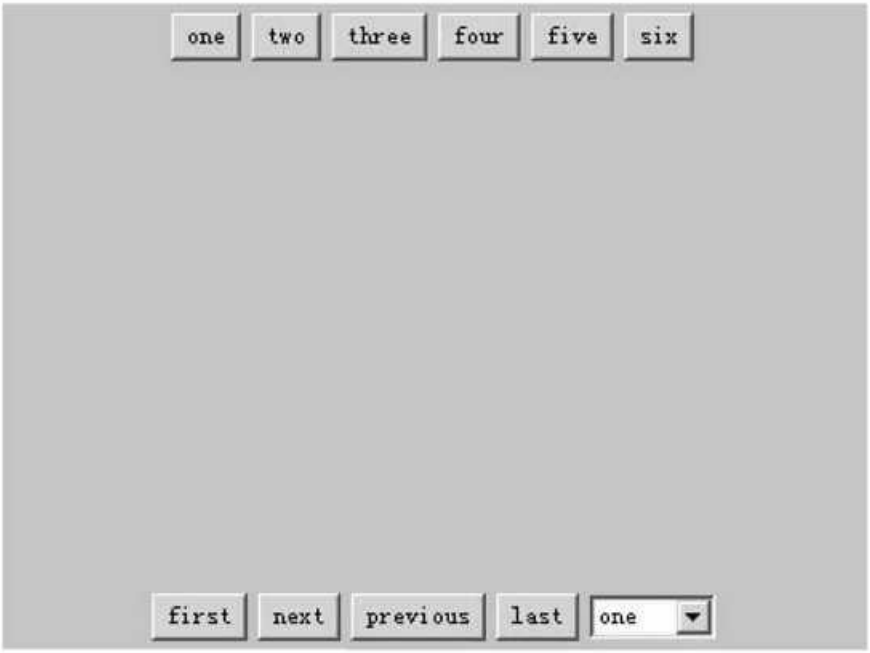


图 10.17　CardLayout 样本

4　GridLayout

GridLayout 提供了一种基于栅格的布局管理方式。栅格的行数和列数可以在创建 GridLayout 对象时指定。采用 GridLayout 进行布局管理，容器中的每个组件将占据大小完全相同的一个栅格。向栅格中布置 GUI 组件有两种方法：一种是使用缺省的布置顺序，即采用 add(Component comp)方法按照从左向右、从上到下的顺序加入 GUI 组件；另一种是采用 add(Component comp,int index)将组件加入到指定的栅格中。

GridLayout 的构造方法如下述。

① GridLayout()：构造一个 GridLayout，缺省单独一行，每个组件一列。

② GridLayout(int rows, int cols)：构造一个 rows 行 cols 列的 GridLayout。

③ GridLayout(int rows, int cols, int hgap, int vgap)：构造一个 rows 行 cols 列的 GridLayout，

并指定水平间隙和垂直间隙。

图 10.18 是 GridLayout 的样本，图 10.19 是加了间隙的 GridLayout。

图 10.18 GridLayout 样本（一）

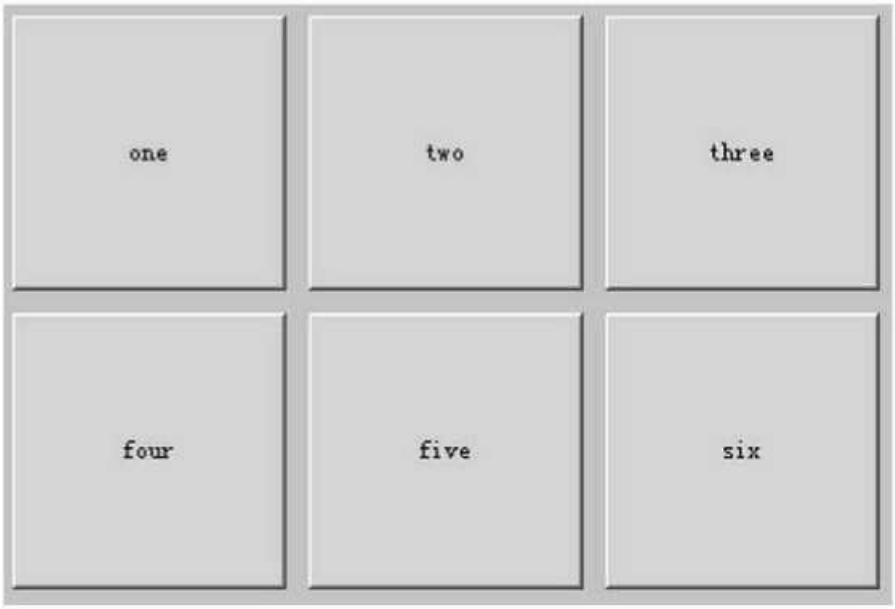


图 10.19 GridLayout 样本（二）

5 GridBagLayout

GridBagLayout 是建立在 GridLayout 之上的一种布局管理方式。较 GirdLayout 而言，GridBagLayout 更复杂，也更灵活，容器中的每一个组件可以占据一个或多个连续的栅格。GridBagLayout 布局管理器是借助于类 GridBagConstraints 来限制每个组件所占据的横向和纵向栅格的个数的。

GridBagLayout 的构造函数就是：**GridBagLayout()**。

如图 10.20 所示是一个 Sun 提供的例子，其源代码见例 10.10。

图 10.20　GridBagLayout 样本

【例 10.10】

```
import java.awt.*;
 import java.util.*;
 import java.applet.Applet;

 public class GridBagEx1 extends Applet {

     protected void makebutton(String name,
                               GridBagLayout gridbag,
                               GridBagConstraints c) {
         Button button = new Button(name);
         gridbag.setConstraints(button, c);
         add(button);
     }

     public void init() {
         GridBagLayout gridbag = new GridBagLayout();
         GridBagConstraints c = new GridBagConstraints();

         setFont(new Font("Helvetica", Font.PLAIN, 14));
         setLayout(gridbag);

         c.fill = GridBagConstraints.BOTH;
         c.weightx = 1.0;
         makebutton("Button1", gridbag, c);
         makebutton("Button2", gridbag, c);
         makebutton("Button3", gridbag, c);

          c.gridwidth = GridBagConstraints.REMAINDER; //end row
         makebutton("Button4", gridbag, c);
```

```
        c.weightx = 0.0;                //reset to the default
        makebutton("Button5", gridbag, c); //another row

        c.gridwidth = GridBagConstraints.RELATIVE; //next-to-last in row
        makebutton("Button6", gridbag, c);

        c.gridwidth = GridBagConstraints.REMAINDER; //end row
        makebutton("Button7", gridbag, c);

        c.gridwidth = 1;                //reset to the default
        c.gridheight = 2;
        c.weighty = 1.0;
        makebutton("Button8", gridbag, c);

        c.weighty = 0.0;                //reset to the default
        c.gridwidth = GridBagConstraints.REMAINDER; //end row
        c.gridheight = 1;               //reset to the default
        makebutton("Button9", gridbag, c);
        makebutton("Button10", gridbag, c);

        setSize(300, 100);
    }
    public static void main(String args[]) {
        Frame f = new Frame("GridBag Layout Example");
        GridBagEx1 ex1 = new GridBagEx1();
        ex1.init();
        f.add("Center", ex1);
        f.pack();
        f.setSize(f.getPreferredSize());
        f.show();
    }
}
```

10.5 AWT 事件处理机制

前面简要介绍了 AWT 提供的 GUI 组件和容器。但是，单单有了 GUI 组件还是无法构造一个应用系统的用户界面，因为这样构造出来的图形用户界面是无法与用户交互的，一个完

整的用户界面系统还必须具备事件处理能力。

从 JDK1.0 到 JDK1.1，AWT 的事件处理机制有了很大的变化。从 JDK1.1 开始，AWT 采用了一种新的事件处理模型——代理事件模型。较 JDK1.0 中的事件处理模型而言，新的事件处理模型不仅更为灵活，而且完全支持 JavaBeans。但是，我们这里仍会先介绍 JDK1.1 以前的事件处理机制，不仅仅因为这是后面的事件处理模型的基础，而且仍有很多源码是用 JDK1.1 以前的版本写的。

10.5.1 JDK1.1 以前的事件处理机制

1.

AWT 事件类的类层次图如图 10.21 所示。

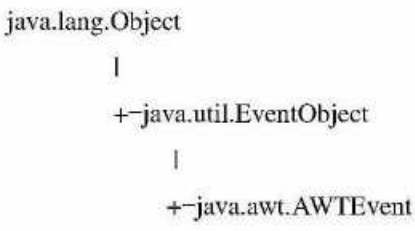


图 10.21 AWT 事件类的类层次图

事件是指用户对程序的某一种功能性操作，和平台无关。Java 中的事件主要有组件类事件（如 componentEvent、ContainerEvent、WindowEvent、FocusEvent、PaintEvent、MouseEvent 共 6 大类，它们均是当组件的状态发生变化时产生的）和动作类事件（如 ActionEvent、TextEvent、AdjustmentEvent、ItemEvent 共 4 类，它们均对应用户的某一种功能性操做动作）组成，Java 中的事件类都包含在 JDK 的 Java.awt.event 包中。

AWT 的每个事件都包含若干个域，从事件域中可以获知这个事件的详细内容。

（1）**id**：事件类型。AWT 中定义了下述事件类型。

① ACTION_EVENT：用户执行的某种动作。

② GOT_FOCUS：获得焦点。

③ KEY_ACTION：按下功能键。

④ KEY_ACTION_RELEASE：释放功能键。

⑤ KEY_PRESS：按下标准键。

⑥ KEY_RELEASE：释放标准键。

⑦ LIST_DESELECT：不选列表的某项。

⑧ LIST_SELECT：选中列表的某项。

⑨ LOAD_FILE：载入文件。

⑩ LOST_FOCUS：失去焦点。

MOUSE_DOWN：按下鼠标键。

MOUSE_DRAG：拖动鼠标，modifiers 域的 Alt_MASK 给出了鼠标是否按下。

MOUSE_ENTER：鼠标进入组件。

MOUSE_EXIT：鼠标离开组件。

MOUSE_MOVE：鼠标移动，此时没有按下鼠标键。

MOUSE_UP：释放鼠标键。

SAVE_FILE：保存文件。

SCROLL_ABSOLUTE：滚动条拖到某个位置。

SCROLL_LINE_DOWN：滚动条向下移一行。

SCROLL_LINE_UP：滚动条向上移一行。

SCROLL_PAGE_DOWN：滚动条向下移一页。

SCROLL_PAGE_UP：滚动条向上移一页。

WINDOW_DEICONIFY：窗口非图标化。

WINDOW_DESTROY：窗口销毁。

WINDOW_EXPOSE：窗口展开。

WINDOW_ICONIFY：窗口图标化。

WINDOW_MOVED：窗口移动。

（2）target：事件的作用对象，即该事件是在哪个组件上发生的。

（3）when：事件发生时间。

（4）x,y：事件发生的位置。

（5）key：键盘事件发生时键入的键值。

（6）**modifiers**：事件发生时的功能键值，指 Alt、Ctrl、Shift 键的状态。

① Alt_MASK：按下了 Alt 键。

② Ctrl_MASK：按下了 Ctrl 键。

③ Meta_MASK：在 UNIX 中按下了 Alt 键。

④ Shift_MASK：按下了 Shift 键。

（7）**arg**：事件的其他参数，根据不同的事件有所不同。

每个事件类都提供了相应的方法以便获得该事件的各种参数和状态。例如 ActionEvent 类提供的下述成员方法。

① public String **getActionCommand**()：获得该事件相关的命令字符串，例如按钮的标识字符串。

② public int **getModifiers**()：获得该事件发生时按下的功能键。

③ public String **paramString**()：获得表示该事件的参数字符串。

2.

用户编程定义每个特定事件发生时程序应做出何种响应，并且这些响应代码会在对应的事件发生时由系统自动调用。

在 JDK1.0 和以前的版本中，任何接受和处理事件的类都是 Component 类的子类。在一个图形用户界面中，容器中包含了组件或其他的容器(如 Panel)，事件可能发生在某个组件中，也可能发生在某个作为容器的窗口中。

当一个组件获得一个事件对象时，可忽略它不作任何处理，这时返回 false，事件会继续往上层传递(如包含它的容器 Panel 或 Frame)；也可对该事件进行处理，处理时调用事件对象提供的方法获得参数进行处理，处理完后，组件可以终止事件向上层传递，也可把事件继续传递给上层。若事件处理结果返回 true，则表明终止事件向上传递。

类 Component 中定义了一系列事件处理方法，这些方法的返回值均为 boolean 类型，用

以指明事件是否向上传递。

（1）动作处理方法

用来捕捉处理在特定部件中产生的事件。当用户作用于某些组件（如按钮、选择框、菜单项、滚动条等）时，产生的事件称为动作（action），AWT 把这类事件交由 action()方法处理。

public boolean **action**(Event evt，Object arg)

evt 参数给出当前事件实例，其中包含了事件的详细信息。arg 参数则是动作所作用的组件，值与 evt.arg 相同。

（2）键盘事件处理方法

① public boolean **keyDown**(Event evt, int key);

② public boolean **keyUp**(Event evt, int key);

其中 key 用来指明按下或释放的键的键值，与 evt.key 值相同。

（3）鼠标事件处理方法

① 按下鼠标键：public boolean **mouseDown**(Event evt, int x, int y)

② 按下鼠标键拖动鼠标：public boolean **mouseDrag**(Event evt, int x, int y)

③ 鼠标进入 Applet：public boolean **mouseEnter**(Event evt, int x, int y)

④ 鼠标离开 Applet：public boolean **mouseExit**(Event evt, int x, int y)

⑤ 移动鼠标：public boolean **mouseMove**(Event evt, int x, int y)

⑥ 鼠标按钮被释放：public boolean **mouseUp**(Event evt, int x, int y)

传递给事件处理方法的有 3 个参数，即 evt、x 和 y。

evt 是一个标识事件全部信息的类。

X 和 Y 坐标，这是发生鼠标事件的坐标地点。与 evt.x 和 evt.y 相同。

（4）通用事件处理方法

public boolean **handleEvent**(Event evt)

在类中重写的 action()、mouseUp()、keyUp()等方法实际上是被 handleEvent()方法简单地调用，这个方法是组件的事件处理开关板，鼠标、按键、动作等各种事件最先到 handleEvent()方法中，被分流到各种简单的处理方法，如 mouseUp()。

如果对一个组件要重写其 handleEvent()方法，为了使所有基本事件（鼠标、键盘或动作事件）都有机会被进行缺省处理，通常应在事件处理方法结束处调用父类的 handleEvent()方法。如：return super.handleEvent(evt)。

3

例 10.11 演示了如何接收鼠标事件 mouseDown 和键盘事件 keyDown。

【例 10.11】

```
import java.applet.*;
import java.awt.*;

public class KeyEvent extends Applet
{
    private int posx,posy;   //当前要写的字符的位置
    private char[] manykeys;   //记录键入字符的数组
```

```
private int keynumber;   //记录键入字符的数目
public KeyEvent()
{
}

public void init()
{
    resize(320, 240);
    posx = 10;
    posy = 10;
    manykeys = new char[100];
    keynumber = 0;
}

public void destroy()
{
}

public void paint(Graphics g)
{
    g.drawChars(manykeys, 0, keynumber, posx, posy);
}

public void start()
{
}

public void stop()
{
}

public boolean mouseDown(Event evt, int x, int y)
{
    //如果按下了鼠标，就在鼠标所单击处重新开始打字
    //记下鼠标的位置
    posx = x;
    posy = y;
    //清除字符数组
    keynumber = 0;
```

```
        return true;
    }

    public boolean mouseUp(Event evt, int x, int y)
    {
        return true;
    }

    //当按键是可显示字符时加入该字符并重绘窗口
    public boolean keyDown(Event evt, int nKey)
    {
        if (keynumber<100)
        {
            manykeys[keynumber]=(char)nKey;
            keynumber++;
        }

        repaint();
        return true;
    }
}
```

程序结果如图 10.22 所示。

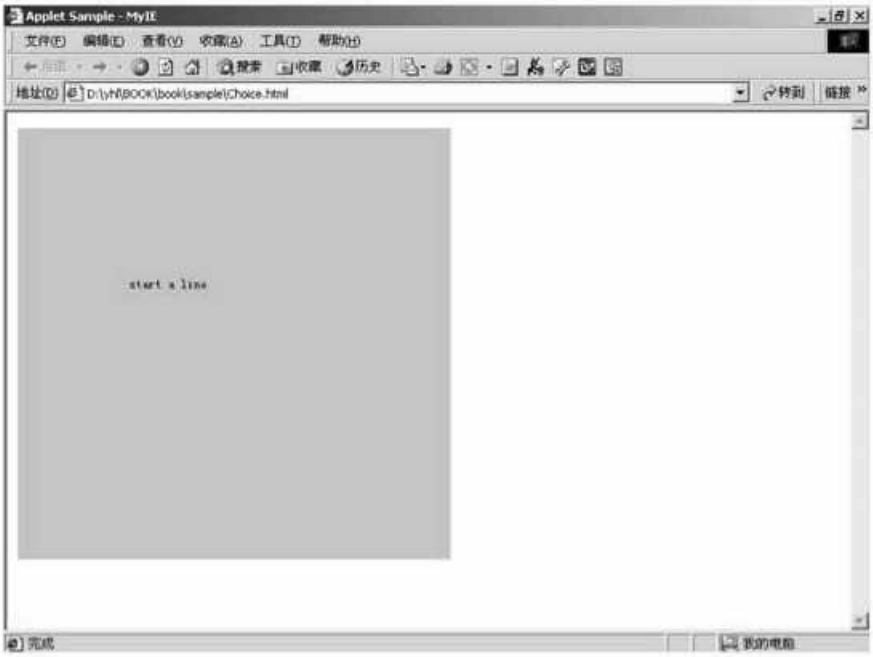


图 10.22　事件处理示例

10.5.2 JDK1.1 之后的事件处理机制

1 JDK1.1

JDK1.1 之后的代理事件模型中有 3 个角色：事件源、事件和事件监听器。

（1）事件源

事件源是产生或发出事件的对象。例如，当单击了图形用户界面上的一个按钮后，按钮就会产生一个鼠标按下的事件。此时，按钮就是事件源。前面提到，在 JDK1.0 和以前的版本中，任何接受和处理事件的类都是 Component 类的子类，但在 JDK1.1 以后就不是这样了，任何类都能接受和处理事件，接受之后，事件是由事件监听器来管理的。

（2）事件

AWT 中定义了各种事件类，如鼠标事件（MouseEvent）和窗口事件（WindowEvent）等，详见 10.6.1 节。

（3）事件监听器

在 AWT 中，事件监听器是 java.awt.event 包中的一类接口，其中定义了当被监听的事件发生时，系统要调用的方法。事件监听器用于监听某类事件的发生，它通过调用事件源的 add******方法（******代表某类事件监听器），来登记所需监听的事件。当该事件源发出此类事件时，事件监听器就会得到通知。

下面是一些常用的事件监听器。

① ActionListener：Button,List,MenuItem,TextField 接收 Action_Event 事件。

② ItemListener：Choice,CheckBox 接收 Action_Event 事件，接收所有的 List_事件。

③ FocusListener：接收 Got_Focus 和 Lost_Focus 事件。

④ KeyListener：接收所有的 Key_事件。

⑤ MouseMotionListener：接收 Mouse_Drag 和 Mouse_Move 事件。

⑥ MouseListener：接收除了 Mouse_Drag 和 Mouse_Move 以外的所有 Mouse_事件。

⑦ AdjustmentListener：接收所有的 Scroll_事件。

⑧ WindowListener：接收所有的 Window_事件，除了 Window_Moved。

⑨ ComponentListener：接收 Window_Moved 事件。

2

有了以上的接口，就可以通过下述 3 个步骤来使用它们接收和管理事件。

（1）为类声明需要的事件监听器接口

和其他的接口声明方法一样，

public class AppletSample extends Applet **implements ActionListener**

如果该类中需要一个以上的事件监听器，则可以根据需要声明多个。

（2）为组件登记接口

在代码中，当生成一个组件作为事件源后，就需要为这个组件登记该类已经声明过的相应的事件监听器，然后再加入该组件，所有的组件都有这样的登记方法，方法名的格式为 add******，******是事件监听器的名字。

例如：

```
Button button1 = new Button("OK");
```

```
button1.addActionListener(this);
add(button1);
```

这里 addActionListener 的参数为 this，表明当组件接收到 Action_Event 事件后，该类实现的事件监视器接口就会自动处理这个事件。

（3）最后在类中实现接口声明的事件处理方法

比如 ActionListener 声明的事件处理方法为 **actionPerformed()**，在该类中就要实现这个方法：

```
public void actionPerformed(ActionEvent event){
    //填写所需代码
}
```

10.6 Swing 简介

AWT 提供了构造 applet 和 application 图形用户界面的基本类库。通过采用“本地对等端”（native peer）模型（即 AWT 中的每一个可视化 GUI 组件都有一个相应的本地对等端来负责它的显示与外观），解决了跨各种软硬件平台显示 GUI 的问题，为 Java 最初的成功奠定了基础。

AWT 的“本地对等端”模型，满足了当时开发人员希望 Java 在不同平台上运行时，具有本地显示风格的要求，但同时也带来下述一些问题。

- “本地对等端”模型给 GUI 的绘画和事件处理带来了很多限制，使得某些方面的 GUI 编程很难实现，例如，采用透明背景色、改变 GUI 组件外形为其他形状等。
- AWT 提供的是本地显示风格，但是随着 Internet 的普及，更多的开发人员希望网络应用能够具有一致的显示风格。

为解决 AWT 的“本地对等端”模型带来的问题，Sun 公司与 Netscape 公司和 IBM 公司合作共同开发了新一代 Java 图形类库——JFC（Java Foundation Class）。JFC 实际上是 AWT 的超集，它提供了更丰富的 GUI 组件和更强的图形/图像处理能力，而且 JFC 完全向下兼容 AWT 的应用。JFC 的发布，使得 Java 在开发客户端应用方面又向前迈进了一大步。目前，JFC 的版本是 JFC1.1，它不但可以作为一个单独的包下载，而且已经成为 JDK1.2 的核心部分。

与 AWT 相比，JFC 提供了更丰富的 GUI 组件，即 Swing 组件。Swing 组件全部是“轻型”组件（即不需要“本地对等端”的组件），具备“可插接的外观和感觉”特性。此外，JFC 还提供了丰富的二维图形/图像支持、系统级的拖放功能和对辅助技术的支持。

10.6.1 Swing

Swing 组件集是 JFC 提供的一套新的 GUI 工具，它简化了基于图形界面的窗口系统的开发。Swing 组件是所谓的“轻型”（lightweight）组件，因为这些组件同 AWT 中的基于本地对等端的组件（即 heavyweight 组件）不同，这些组件没有对应的对等端，即不需要跟操作系统相关的本地 UI 代码。因此，实现这些组件所需要的代码量更少，而且减少了在不同平

台上运行可能带来的不一致性。

此外，开发人员使用 Swing 可以方便地选择自己需要的 GUI 风格，即本地的显示风格、统一的 Java 显示风格或用户定制的显示风格。总之，当开发客户端应用时，Swing 给程序员带来了更多的灵活性和更强大的功能。

事实上，Swing 是对 AWT 的扩展，而并非是 Swing 替代了 AWT。从体系结构的角度来说，Swing 构筑在 AWT（并非 AWT 的全部）之上。Swing 位于 JFC 的一系列 API 之上，例如 Java 2D API、Drag&Drop API 和 Accessibility API。因为这些 API 都需要依赖于本地代码来执行一些任务，而 Swing 的组件是不依赖于本地对等端的。因此，Java 2D API、Drag&Drop API 和 Accessibility API 都是 JFC 的组成部分，同时也是 AWT 的组成部分，但是它们都不属于 Swing。

10.6.2 Swing 组件介绍

Swing 提供了一套丰富的“轻型”GUI 组件，定义在 javax.swing 包中。Swing 组件涵盖了 AWT 中业已提供的 GUI 组件，这些组件的类名就是 AWT 的组件类名加上前缀“J”。此外，为了满足商业化桌面应用的需求，同时简化应用的开发和部署，Swing 还定义了一套新的高层次的 GUI 组件。其中，基本 GUI 组件新增加了 JTable、JTree、JSlide、JProgressBar、JToolTip 以及 JFileChooser 和 JColorChooser，并将 Choice 组件更名为大家熟悉的 ComboBox；而容器组件则新增加了 JSplitPane、JTabbedPane、JToolBar、JInternalFrame 和 JLayeredPane。

Swing 组件，无论是基本组件还是容器组件，都沿用了 AWT（JDK1.1）的事件处理模型。除了 java.awt.event 包中定义的事件类和事件监听器接口，Swing 还在 javax.swing.event 包中为新增加的组件定义了相应的事件类和事件监听器接口。

10.6.3 Swing 组件体系结构

Swing 组件早期采用的体系结构是“模式－视图－控制器”（model-view-controller），简称为 MVC。MVC 模型的应用可以追溯至 SmallTalk，它是 GUI 对象常用的一种结构设计。在 MVC 模型中，一个可视化应用可以分为以下 3 个方面。

- 模式：代表应用的数据。
- 视图：是应用数据的可视化表示。
- 控制器：接收用户在数据视图上的输入，并相应地改变数据模式。

Swing 开发小组在试图采用 MVC 作为 Swing 组件的结构模型过程中，发现 MVC 的视图和控制器部分密切相关，这种简单将应用切分为上述 3 部分的做法在实际工作中效果并不理想。因此，经过讨论，Swing 开发小组决定采用了一种改进的 MVC 模型，即“分离模式”模型。在这种结构模型中，模式独立出来，而视图和控制器则合并成为一个单一的 UI 对象，用以负责 Swing 组件的外观和感觉（look and feel，即 LAF）。

在这种结构模型中，包括 UI 管理器、UI 对象和模式 3 部分。

（1）UI 管理器

为了管理 Swing 组件的显示特性，Swing 在 javax.swing 包中定义了 UIManager 类，以跟踪和管理 Swing 组件当前的和缺省的 LAF。目前，Swing 提供 3 种 GUI 显示风格，即 Java LAF、Windows LAF 和 Motif LAF。开发人员可以通过 UIManager 的 setLookAndFeel 方法，动态地

设置 Swing 应用的显示风格。

（2）UI 对象/UI 代理

Swing 的每一个组件都有处理其自身的视图和控制器的能力。但是，每一个 Swing 组件类都不直接处理与其显示相关的部分，而是将它代理给当前安装的 LAF 模块所提供的 UI 对象，所以也称之为 UI 代理。UIManager 对 Swing 组件的外观和显示的控制，是通过它与 Swing 组件的 UI 对象的通信来实现的。UIManager 和 UI 代理与 Swing 组件的可插接的显示和感觉（Pluggable Look and Feel）特性密切相关。

（3）模式 Model

通常人们认为：一个设计良好的程序应当是以体系结构为中心的，而不是以用户界面为中心的。因而 Swing 在设计过程中以及其后的版本更新过程中，始终坚持了这一点。基于此，对每一个能够存储或操纵数据/值的组件，Swing 都定义了一个单独的 Model 接口，开发人员既可以使用系统提供的缺省数据模式，也可以实现自定义的数据模式。

在 Swing 中，Model 接口可以分为两类，即 GUI 状态模式接口和应用—数据模式接口。前者是定义了 GUI 组件的可见状态的接口，例如 ButtonModel 接口中定义了按钮当前的状态；而对于更复杂的 GUI 组件（如 JTree、JTable 等），则定义了应用—数据模式接口，这类接口用于管理复杂的数据信息。

实际上，Swing 的“分离模式”模型并未明确区分 GUI 状态模式接口和应用—数据模式接口。但是，了解这两种模式接口的不同，有利于根据应用的需要和组件的特点灵活地进行编程。

10.6.4 可插接的外观和感觉

可插接的外观和感觉（Pluggable look & Feel，PL&F）是 Swing 的一部分，它提供了一种灵活的机制，即开发人员既可以选择 Java LAF，在不同平台上获得一致的显示风格，也可以选择依赖于平台的 LAF。如果系统提供的 LAF 仍不满足应用的需要，开发人员可以利用 Swing 提供的“可插接的外观和感觉”特性来构造一个新的、定制化的 LAF。

PL&F 的实现与前面提到的 UIManager 和 UI 代理密不可分。每一个 Swing 组件特定于外观和感觉的部分都代理给相应的 UI 对象了。Swing 在 javax.swing.plaf 包中定义了一套抽象类来代表 UI 对象，其命名规范是将 Swing 组件类名的前缀“J”去掉，再加上后缀“UI”。每一个 LAF 模块（包括系统安装的和用户自定义的），都必须提供这些抽象类的实现。通常，UI 代理是在 Swing 组件的构造函数中被创建，开发人员可以通过 Swing 组件的 setUI / getUI 方法来设置/获取之。设置 Swing 组件的 UI 代理的过程实际上就是安装组件的 LAF 的过程。

当一个用 Swing 编写的应用程序被 JVM 载入的时候，Swing 缺省的动作是将组件的 LAF 初始化为 Java LAF。如果希望采用其他的 LAF，则只需要调用 UIManager 中定义的 setLookAndFeel 方法即可。UIManager 类还定义了各种方法，以方便用户获取有关 LAF 的各种信息，例如，本地系统采用的 LAF、当前的和缺省的 LAF 以及系统安装的所有 LAF 等。

除采用系统提供的 LAF 外，还可以定义自己需要的 LAF。Swing 定义了抽象类 javax.swing.LookAndFeel，它代表了实现一种 LAF 所需要的全部信息。所有的 LAF 都需要继承抽象类 LookAndFeel，并实现其中的抽象方法。通常，一个 LAF 模块（如 javax.swing .plaf.metal）包含 LookAndFeel 类的子类（如 MetalLookAndFeel）、所有 UI 代理类（例如 MetalButtonUI 等）和 LAF 应用类。

第11章

数据库编程

Java提供统一的应用程序接口访问关系数据库，编程人员不用关心底层数据库的细节差别，就可以实现方便的访问。

本章的主要内容包括:

- JDBC概述
- JDBC的接口和类
- JDBC的程序示例

11.1 JDBC 概述

11.1.1 JDBC 的出现

随着数据库技术的发展，大量有用的数据都存放在数据库中，由数据库管理系统（DBMS）进行存储和管理。目前，DBMS 的产品有很多，例如 DB2、ORACLE、SQLServer、Sybase、Informix、Mysql、Access 等，它们在各自的领域都有众多稳定的用户。

虽然都是关系数据库，而且也都支持标准 SQL，但是这些 DBMS 各自提供自己的数据访问方法，例如 DB2 的 CLI，ORACLE 的 OCI 和 PL/SQL，当用户需要访问这两种数据库时，就不得不使用两套 API，编写两套数据库访问程序；而如果需要将已编写好的项目再移植到另一种数据库上时（这种情况是经常发生的），就不得不再为同一个逻辑编写另一套数据库访问程序；而如果你的项目声称支持所有的数据库系统呢？实际上，数据库的访问差别在 ODBC 和 JDBC 没出现之前成了所有数据库程序员的噩梦。为什么不编写一个统一的接口来屏蔽这种差别呢？毕竟关系数据库的数据存放和访问方式是有标准的 SQL 可以遵循的。定义一个通用的 SQL 数据库存取框架，在各种各样的提供数据库连接模块上提供统一的界面是十分有意义的。

于是，出现了 ODBC（Open DataBase Connection，开放式数据库连接），这是一套数据库连接和访问的统一接口，这个标准制定得十分合理实用，在业界被广泛采用。

JDBC（Java DataBase Connection，Java 数据库连接）是按照 ODBC 的模式制定的，是一个通用低层的、支持基本 SQL 功能的 Java API，这使程序员可以面对单一的数据库界面，使数据库无关的 Java 工具和产品成为可能，使得数据库连接的开发者可以提供各种各样的连接方案。

11.1.2 什么是 JDBC

JDBC API 是一个标准 SQL（Structured Query Language，结构化查询语言）数据库访问接口，它使数据库开发人员能够用标准 Java API 编写数据库应用程序。

由 JavaSoft 定义的 JDBC API 的技术规范是由 SUN 和其 Java 技术合作伙伴开发的标准数据访问接口，这是一个使 Java 程序能够与数据库服务器通信的 Java 应用程序接口，它不针对任何厂家。JDBC 由一系列说明 API 的 Java 接口和一些由数据库厂家提供的使 Java 程序能够与数据库连接的驱动程序组成。

JDBC 对 Java 程序员而言是 API，对实现与数据库连接的服务提供商而言是接口模型。作为 API，JDBC 为程序开发提供标准的接口，并为数据库厂商及第三方中间件厂商实现与数据库的连接提供了标准方法。JDBC 使用已有的 SQL 标准并支持与其他数据库连接标准，如 ODBC 之间的桥接。JDBC 实现了所有这些面向标准的目标并且具有简单、严格类型定义且高性能实现的接口。

11.1.3 JDBC 的组成

Java 提供 3 种 JDBC 产品组件，它们是 Java 开发工具包（JDK）的组成部分，即 JDBC

驱动程序管理器、JDBC 驱动程序测试工具包和 JDBC-ODBC 桥，如图 11.1 所示。

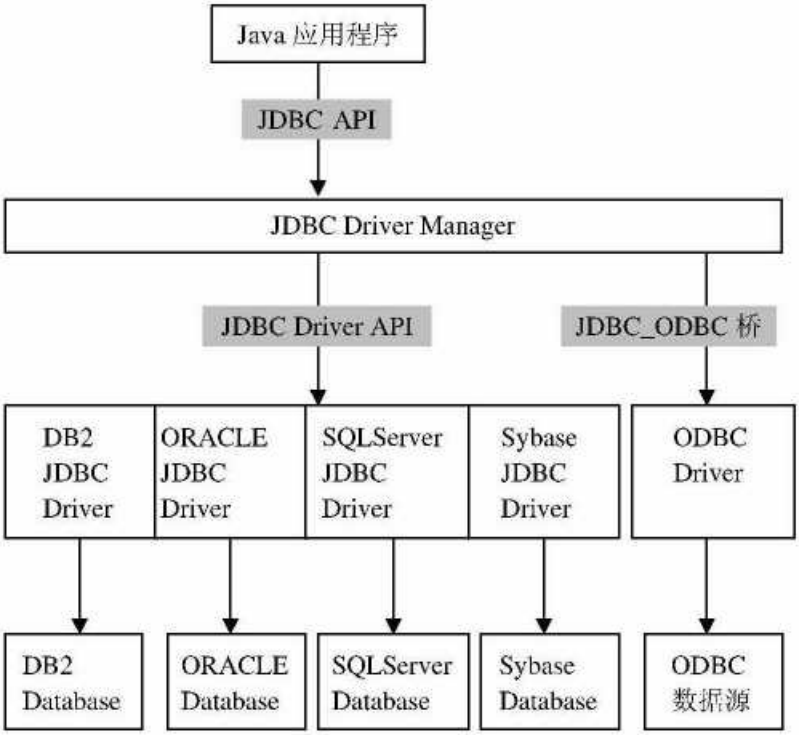


图 11.1　JDBC 的组成

1　JDBC API　JDBC

从功能上来看，JDBC 包含两部分与数据库独立的 API，即面向程序开发人员的 JDBC API 和面向底层的 JDBC Driver API。

JDBC API 定义了表示数据库连接、SQL 语句、结果集和数据库元数据的 Java 类。JDBC 按 ODBC 的结构模型化。这些类提供标准功能，其中包括：事务管理、查询、处理预编译语句、调用存储过程、对长字段数据的流式存取、访问数据库字典、游标描述和几个数据库的同时连接。

Java 应用程序使用 Java API 来发出 SQL 语句并对结果进行处理，Java API 是通过一个驱动程序管理器实现的，它可支持连接到不同数据库的多个驱动程序。JDBC 既可以全部由 Java 写成并作为 Applet 的一部分下载下来，也可以通过连接已有的数据库访问库的方法来实现。例如，Oracle 提供两类 JDBC 驱动程序：即作为 Java applet 的 JDBC-THIN 和作为 Java 应用程序的 JDBC-OCI。

JDBC 驱动程序可分为 4 类。

（1）JDBC-ODBC 桥

通过 ODBC 驱动程序提供 JDBC 访问，该驱动程序最适合于商业网络或三层体系结构中当应用服务器层的代码是由 Java 写成时的情况，该驱动程序的使用要求 ODBC 二进制代码必须在每个客户机上安装。

（2）部分 Java 技术的本地 API 驱动程序

将 JDBC 调用转换为客户端 API 对 DBMS 的调用，这些 DBMS 包括 Oracle、Sybase、Informix 和 DB2。这类驱动程序要求在每个客户端机器上都安装一定的二进制代码。

（3）全部基于 Java 技术的本地 API 驱动程序

将 JDBC 调用翻译为与 DBMS 无关的网络协议，然后网络服务器再将其翻译为 DBMS 协议。这类网络服务器中间件能够连接其所有的 Java 客户端到许多不同的数据库上，这是最灵活的 JDBC 驱动程序。

（4）全部基于 Java 技术的本地协议驱动程序

直接将 JDBC 调用转换为 DBMS 使用的网络协议。因为大多数这样的协议都是数据库专有的，数据库厂家将是这类驱动程序的最主要来源，一些数据库厂家正在开发这类驱动程序。

第（3）、（4）类驱动程序将成为从 JDBC 访问数据库的首选方法。第（1）、（2）类驱动程序在直接的纯 Java 驱动程序还没有上市前将会作为过渡方案来使用。对第（1）、（2）类驱动程序可能会有一些变种，这些变种要求有连接器，但通常这些是更加不可取的解决方案。第（3）、（4）类驱动程序提供了 Java 的所有优点，包括自动安装（例如，通过使用 JDBC 驱动程序的 applet 来下载该驱动程序）。

目前，已有几十个（1）类的驱动程序，即可与 JDBC-ODBC 桥联合使用的 ODBC 驱动程序的驱动程序。有大约十多个属于种类（2）的驱动程序是以 DBMS 的本地 API 为基础编写的。只有几个属于种类（3）的驱动程序，其首批提供者是 SCO、OpenHorizon、Visigenic 和 WebLogic。此外，JavaSoft 和数据库连接的领先提供者 Intersolv 还合作研制了 JDBC-ODBC 桥和 JDBC 驱动程序测试工具包。

Java 程序通过 JDBC API 访问 JDBC Driver Manager，JDBC Driver Manager 再通过 JDBC Driver API 访问不同的 JDBC 驱动程序，从而实现对不同数据库的访问。

JDBC 提供了一个通用的 JDBC Driver Manager，用来管理各数据库软件商提供的 JDBC 驱动程序，从而访问其数据库。此外，对没有提供相应 JDBC 驱动程序的数据库系统，开发了特殊的驱动程序，即 JDBC－ODBC 桥。现在越来越多的数据库厂商都提供其数据库的 JDBC 驱动程序。

2　JDBC-ODBC

前面说到，JDBC－ODBC 桥是一个特殊的驱动程序，它并不面对一种数据库，而是面对 ODBC 驱动程序，该驱动程序支持 JDBC 通过现有的 ODBC 驱动程序访问其数据库系统。JDBC－ODBC 桥的出现是历史决定的，由于 ODBC 的使用范围很大，绝大部分数据库管理系统都有自己的 ODBC 驱动程序，但有的比较少用的数据库系统的 JDBC 的驱动程序却不一定都有，为了使 JDBC 也能连接这些数据库，JDBC－ODBC 桥作为一种过渡方案出现了。

使用 JDBC-ODBC 桥必须将 ODBC 二进制代码（许多情况下还包括数据库客户机代码）加载到使用该驱动程序的每个客户机上。因此，这种类型的驱动程序最适合于企业网（这种网络上客户机的安装不是主要问题），或者是用 Java 编写的三层结构的应用程序服务器代码。

11.1.4　JDBC URL

1　JDBC URL

由于 URL 常引起混淆，我们将先对一般 URL 作简单说明，然后再讨论 JDBC URL。URL（统一资源定位符）提供在 Internet 上定位资源所需的信息，可将它想象为一个地址。URL 的第一部分指定了访问信息所用的协议，后面总是跟着冒号。常用的协议有“ftp”（代表“文件传输协议”）和“http”（代表“超文本传输协议”）。如果协议是“file”，则表示资源是在某个

本地文件系统上而非在Internet上(下例用于表示我们所描述的部分，它并非URL的组成部分)。

ftp：//Javasoft.com/docs/JDK-1_apidocs.zip

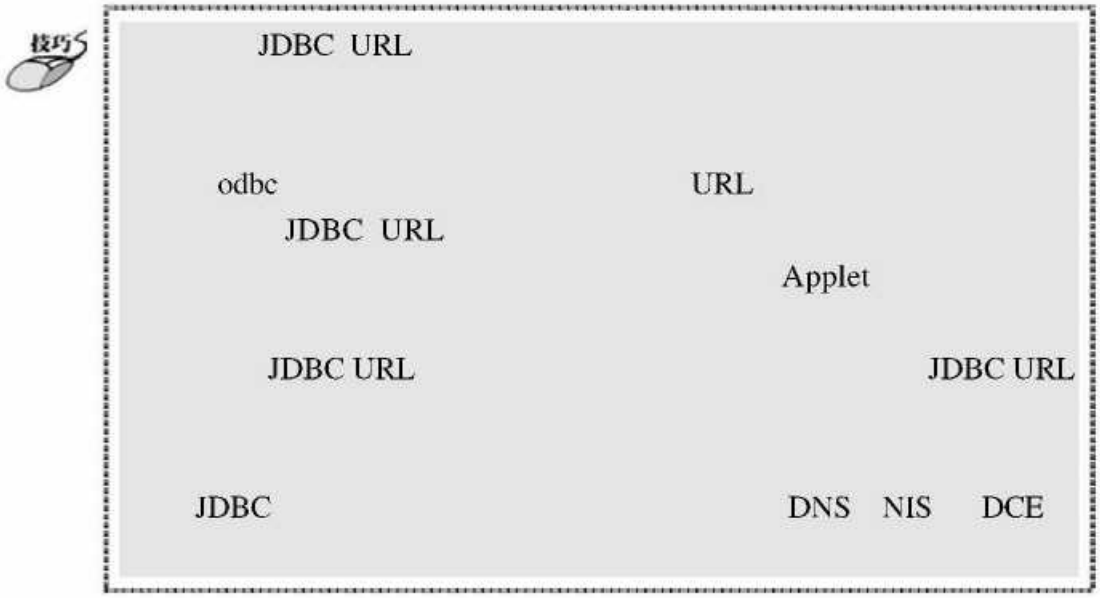
http：//Java.sun.com/products/jdk/CurrentRelease

file：/home/haroldw/docs/books/tutorial/summary.html

URL的其余部分（冒号后面的）给出了数据资源所处位置的有关信息。如果协议是file，则URL的其余部分就是文件的路径。对于ftp和http协议，URL的其余部分标识了主机并可选地给出某个更详尽的地址路径。例如，以下是JavaSoft主页的URL。该URL只标识了主机：http：//Java.sun.com。从该主页开始浏览，就可以进到许多其他的网页中，其中之一就是JDBC主页。JDBC主页的URL更为具体，它具体表示为：

http：//Java.sun.com/products/jdbc

JDBC URL提供了一种标识数据库的方法，可以使相应的驱动程序能识别该数据库并与之建立连接。实际上，驱动程序编程员将决定用什么JDBC URL来标识特定的驱动程序。用户不必关心如何形成JDBC URL；他们只须使用与所用的驱动程序一起提供的URL即可。JDBC的作用是提供某些约定，驱动程序编程员在构造他们的JDBC URL时应该遵循这些约定。



JDBC URL的标准语法如下所示（它由3部分组成，各部分间用冒号分隔）。

jdbc：<子协议>：<子名称>

JDBC URL的3个部分可分解如下所述。

（1）jdbc协议

JDBC URL中的协议总是jdbc。

（2）<子协议>

驱动程序名或数据库连接机制（这种机制可由一个或多个驱动程序支持）的名称。子协议名的典型示例是“odbc”，该名称是为用于指定ODBC风格的数据资源名称的URL专门保留的。例如，为了通过JDBC-ODBC桥来访问某个数据库，可以用如下所示的URL：

jdbc：odbc：test

本例中，子协议为“odbc”，子名称“test”是本地 ODBC 数据资源。

如果要用网络命名服务（这样 JDBC URL 中的数据库名称不必是实际名称），则命名服务可以作为子协议。例如，可用如下所示的 URL：

jdbc：dcenaming：accounts

本例中，该 URL 指定了本地 DCE 命名服务应该将数据库名称“accounts”解析为更为具体的可用于连接真实数据库的名称。

（3）<子名称>

一种标识数据库的方法。子名称可以依不同的子协议而变化。它还可以有子名称的子名称（含有驱动程序编程员所选的任何内部语法）。使用子名称的目的是为定位数据库提供足够的信息。然而，位于远程服务器上的数据库需要更多的信息。例如，如果数据库是通过 Internet 来访问的，则在 JDBC URL 中应将网络地址作为子名称的一部分包括进去，且必须遵循如下所示的标准 URL 命名约定：

//主机名：端口/子协议

假设“dbnet”是个用于将某个主机连接到 Internet 上的协议，则 JDBC URL 应为：

jdbc：dbnet：//ant：356/fred

2　Odbc

子协议 odbc 是一种特殊情况。它是为用于指定 ODBC 风格的数据资源名称的 URL 而保留的，并具有允许在子名称（数据资源名称）后面指定任意多个属性值的特性。odbc 子协议的完整语法为：

jdbc：odbc：<数据资源名称>[；<属性名>=<属性值>=

因此，以下都是合法的 jdbc：odbc 名称。

jdbc：odbc：mydb1

jdbc：odbc：mydb2

jdbc：odbc：mydb2；CacheSize=20；ExtensionCase=LOWER

jdbc：odbc：mydb3；UID=scott；PWD=tiger

11.1.5　事务

1

事务由一个或多个这样的语句组成，这些语句已被执行、完成并被提交或回滚。当调用方法 commit（提交）或 rollback（回滚）时，当前事务即告结束，另一个事务随即开始。缺省情况下，新连接将处于自动提交模式。也就是说，当执行完语句后，将自动对那个语句调用 commit 方法。这种情况下，由于每个语句都是被单独提交的，因此一个事务只由一个语句组成。如果禁用自动提交模式，事务将要等到 commit 或 rollback 方法被显式调用时才结束，因此它将包括上一次调用 commit 或 rollback 方法以来所有执行过的语句。对于第二种情况，事务中的所有语句将作为组来提交或回滚。

方法 commit 使 SQL 语句对数据库所做的任何更改成为永久性的，它还将释放事务持有的全部锁。而方法 rollback 将放弃那些更改。有时用户在另一个更改生效前不想让此更改生效。这可通过禁用自动提交并将两个更新组合在一个事务中来达到。如果两个更新都是成功，则调用 commit 方法，从而使两个更新结果成为永久性的；如果其中之一或两个更新都失败

了，则调用 rollback 方法，以便将值恢复为进行更新之前的值。

大多数 JDBC 驱动程序都支持事务。事实上，符合 JDBC 的驱动程序必须支持事务。DatabaseMetaData 给出的信息描述 DBMS 所提供的事务支持水平。

2

如果 DBMS 支持事务处理，它必须有某种途径来管理两个事务同时对一个数据库进行操作时可能发生的冲突。用户可指定事务隔离级别，以指明 DBMS 应该花多大精力来解决潜在冲突。例如，当事务更改了某个值而第二个事务却在该更改被提交或还原前读取该值时该怎么办。

假设第一个事务被还原后，第二个事务所读取的更改值将是无效的，那么是否可允许这种冲突？JDBC 用户可用以下代码来指示 DBMS 允许在值被提交前读取该值（“dirty 读取”），其中 con 是当前连接：

```
con.setTransactionIsolation（TRANSACTION_READ_UNCOMMITTED）;
```

事务隔离级别越高，为避免冲突所花的精力也就越多，Connection 接口定义了下述 5 个级别。

① TRANSACTION_NONE：无事务隔离级别。

② TRANSACTION_READ_UNCOMMITTED：可以读未提交的数据（脏数据）。

③ TRANSACTION_READ_COMMITTED：不可以读未提交的数据（避免读“脏数据”）。

④ TRANSACTION_REPEATABLE_READ：不可修改正在读的数据（避免了“不可重复读”）。

⑤ TRANSACTION_SERIALIZABLE：不可对正在操作的数据库做任何修改（包括插入新数据）。

其中，最低级别指定了根本不支持事务，而最高级别则指定当事务在对某个数据库进行操作时，任何其他事务不得对那个事务正在读取的数据进行任何更改。通常，隔离级别越高，应用程序执行的速度也就越慢（由于用于锁定的资源耗费增加了，而用户间的并发操作减少了）。在决定采用什么隔离级别时，开发人员必须在性能需求和数据一致性需求之间进行权衡。当然，实际所能支持的级别取决于所涉及的 DBMS 的功能。

当创建 Connection 对象时，其事务隔离级别取决于驱动程序，但通常是所涉及的数据库的缺省值。用户可通过调用 setIsolationLevel 方法来更改事务隔离级别。新的级别将在该连接过程的剩余时间内生效。要想只改变一个事务的事务隔离级别，必须在该事务开始前进行设置，并在该事务结束后进行复位。


```
commit
```

11.2 JDBC 的接口和类

JDBC 的接口和类定义都在包 java.sql 和 javax.sql 中。前者是 Java 标准版使用的包，后者是扩展的包，在 J2EE 中会用到，这里只介绍 java.sql 包。

J2SE 的 java.sql 包中定义的接口有下述几种。

- Array：SQL 数组类型数据在 Java 语言中的映射，定义了几个获得数组结果集的函数。

- Blob：SQL 二进制类型大数据在 Java 语言中的映射。
- CallableStatement：执行 SQL 存储过程的接口。
- Clob：SQL 字符类型大数据在 Java 语言中的映射。
- Connection：一个特定数据库的连接（会话）。
- DatabaseMetaData：数据库的综合信息。
- Driver：每个驱动类必须实现的接口。
- PreparedStatement：预编译的 SQL 语句。
- Ref：SQL 的 Ref 类型数据（结构化数据）在 Java 语言中的映射。
- ResultSet：SQL 语句的查询结果集。
- ResultSetMetaData：用来获得 ResultSet 对象的每列的数据类型和其他特性的对象。
- SQLData：用来定制用户自定义数据类型的接口。
- SQLInput：SQL 结构数据类型或其他特定数据类型的输入流。
- SQLOutput：向数据库写用户定义数据类型的特性的输出流。
- Statement：用来执行 SQL 语句和获得结果的接口。
- Struct：SQL 结构数据类型在 Java 语言中的映射。
- J2SE 的 java.sql 包中定义的类有下述几种。
- Date：用来表示 SQL 日期类型的毫秒级对象。
- DriverManager：用来管理 JDBC 驱动的基本服务对象。

JDBC2.0　　DataSource

- DriverPropertyInfo：连接数据库的驱动的特性。
- SQLPermission：用于 Applet 的安全管理器许可检查。
- Time：用来表示 SQL 时间类型的对象。
- Timestamp：用来表示 SQL 时间戳的对象。
- Types：包含一套通用的 SQL 类型，称为 JDBC 类型。

其中，Connection 接口、PreparedStatement 接口、ResultSet 接口、Statement 接口、DriverManager 接口是构成 JDBC 数据库访问框架的主要成分，下面进行详细讲解。

11.2.1 Connection

Connection 接口用于一个特定的数据库连接，它包含维持该连接的所有信息，并提供关于这个连接的方法。

（1）Statement **createStatement()** throws SQLException;

在本连接上生成一个 Statement 对象，该对象可对本连接的特定数据库发送 SQL 语句。

（2）PreparedStatement **prepareStatement**(String sql) throws SQLException;

在本连接上生成一个 PreparedStatement 对象。

（3）CallableStatement **prepareCall**(String sql) throws SQLException;

在本连接上生成一个 CallableStatement 对象。

（4）void **setAutoCommit**(boolean autoCommit) throws SQLException;

设置自动提交，设置后，在这个连接上的 SQL 语句都将自动提交，而不是等待 commit 或 rollback 才写数据库。

（5）boolean getAutoCommit() throws SQLException;

获得自动提交状态。

（6）void commit() throws SQLException;

提交数据库上当前的所有待提交的事务。

（7）void rollback() throws SQLException;

回滚数据库上当前的所有待提交的事务。

（8）void close() throws SQLException

关闭当前的 JDBC 数据库连接。

（9）boolean isClosed() throws SQLException;

查看是否已关闭。

11.2.2 PreparedStatement

前面提到，PreparedStatement 对象用于执行带或不带输入参数的预编译 SQL 语句，这种 SQL 语句可以在程序中动态改变参数。比如：Update table1 set a= 'standard' where b>=?。这里的“?”就是输入参数，可以在程序中动态指定它的值。因此，除了继承 Statement 类的上述方法外，PreparedStatement 还提供用于指定输入参数的值的方法。

（1）void setNull(int parameterIndex, int sqlType) throws SQLException;

（2）void setBoolean(int parameterIndex, boolean x) throws SQLException;

（3）void setByte(int parameterIndex, byte x) throws SQLException;

（4）void setShort(int parameterIndex, short x) throws SQLException;

（5）void setInt(int parameterIndex, int x) throws SQLException;

（6）void setLong(int parameterIndex, long x) throws SQLException;

（7）void setFloat(int parameterIndex, float x) throws SQLException;

（8）void setDouble(int parameterIndex, double x) throws SQLException;

（9）void setBigDecimal(int parameterIndex, BigDecimal x) throws SQLException;

（10）void setString(int parameterIndex, String x) throws SQLException;

（11）void setBytes(int parameterIndex, byte x[]) throws SQLException;

（12）void setDate(int parameterIndex, java.sql.Date x) throws SQLException;

（13）void setTime(int parameterIndex, java.sql.Time x) throws SQLException;

（14）void setTimestamp(int parameterIndex, java.sql.Timestamp x) throws SQLException;

（15）void setAsciiStream(int parameterIndex, java.io.InputStream x, int length) throws SQLException;

（16）void setUnicodeStream(int parameterIndex, java.io.InputStream x, int length) throws SQLException;

（17）void setBinaryStream(int parameterIndex, java.io.InputStream x, int length) throws SQLException;

（18）void setCharacterStream(int parameterIndex, java.io.Reader reader, int length) throws SQLException;

（19）void setRef (int i, Ref x) throws SQLException;

（20）void setBlob (int i, Blob x) throws SQLException;

（21）void setClob (int i, Clob x) throws SQLException;

（22）void setArray (int i, Array x) throws SQLException;

以上是 PreparedStatement 提供的一些规则的设置输入参数值的方法，方法名指定了参数类型，方法的第一个参数是该输入参数的位置，即它在 SQL 语句中是第几个“？”，方法的第二个参数是指定的值；有的方法还有第三个参数，当数据类型无法确定数据长度时，用第三个参数来指定输入数据的长度。

（23）void setObject(int parameterIndex, Object x, int targetSqlType, int scale)throws SQLException;

（24）void setObject(int parameterIndex, Object x, int targetSqlType)throws SQLException;

（25）void setObject(int parameterIndex, Object x) throws SQLException;

以上 3 个也是设置输入参数的方法，当并不知道（或并不确定）数据类型时，使用这 3 个方法，前两个参数含义和前面的方法一样，第三个参数可指定 SQL 数据类型，第四个参数进一步指定类型的精度。

（26）void clearParameters() throws SQLException;

清空输入参数缓冲区。

11.2.3 ResultSet

结果集是当执行 SQL 查询语句后的返回结果表，ResultSet 接口是用来处理查询结果的。每个 ResultSet 对象都包含一个游标，SQL 查询语句执行完并使 ResultSet 对象初始化后，游标是在第一行结果上，然后可以调用方法 next()来使游标移到下一行，当移到最后一行时，再使用方法 next()会由于 ResultSet 对象中已没有结果将返回 false。缺省的设置是结果集不可写，并且游标只能一个一个地向下移动。在 JDBC2.0 中，结果集可以更改，可以影响数据库，且游标可以来回移动，既可以向下移动也可以向上移动，还可以移到指定位置。本节列出的是 ResultSet 接口提供的主要方法。

（1）String getCursorName() throws SQLException：获得游标名字。

以下 6 个方法是游标移动方法，分别执行游标移到下一行、前一行、第一行、最后一行、第 row 行以及相对于当前行的第 rows 行。

（2）boolean next() throws SQLException ;

（3）boolean previous() throws SQLException;

（4）boolean first() throws SQLException;

（5）boolean last() throws SQLException;

（6）boolean absolute(int row) throws SQLException;

（7）boolean relative(int rows) throws SQLException;

下面这两个方法是将游标移到第一行之前和移到最后一行之后。

（8）void beforeFirst() throws SQLException;

（9）void afterLast() throws SQLException;

以下4个方法是查看游标是否在某个位置，根据名字可以知道是查看什么位置：

（10）boolean isBeforeFirst() throws SQLException;

（11）boolean isAfterLast() throws SQLException;

（12）boolean isFirst() throws SQLException;

（13）boolean isLast() throws SQLException;

（14）int findColumn(String columnName) throws SQLException;

根据列名获得列号，即第几列。

（15）int getRow() throws SQLException;

获得当前记录的位置，即第几行。

以下是获得游标指示的当前记录的字段值，可以根据列号或列名来获得，参数columnIndex表示列号，columnName表示列名，根据该字段的数据类型使用相应的方法。

（16）String getString(int columnIndex) throws SQLException;

（17）boolean getBoolean(int columnIndex) throws SQLException;

（18）byte getByte(int columnIndex) throws SQLException;

（19）short getShort(int columnIndex) throws SQLException;

（20）int getInt(int columnIndex) throws SQLException;

（21）long getLong(int columnIndex) throws SQLException;

（22）float getFloat(int columnIndex) throws SQLException;

（23）double getDouble(int columnIndex) throws SQLException;

（24）BigDecimal getBigDecimal(int columnIndex, int scale) throws SQLException;

（25）byte[] getBytes(int columnIndex) throws SQLException;

（26）java.sql.Date getDate(int columnIndex) throws SQLException;

（27）java.sql.Time getTime(int columnIndex) throws SQLException;

（28）java.sql.Timestamp getTimestamp(int columnIndex) throws SQLException;

（29）java.io.InputStream getAsciiStream(int columnIndex) throws SQLException;

（30）java.io.InputStream getUnicodeStream(int columnIndex) throws SQLException;

（31）java.io.InputStream getBinaryStream(int columnIndex) throws SQLException;

（32）String getString(String columnName) throws SQLException;

（33）boolean getBoolean(String columnName) throws SQLException;

（34）byte getByte(String columnName) throws SQLException;

（35）short getShort(String columnName) throws SQLException;

（36）int getInt(String columnName) throws SQLException;

（37）long getLong(String columnName) throws SQLException;

（38）float getFloat(String columnName) throws SQLException;

（39）double getDouble(String columnName) throws SQLException;

（40）BigDecimal getBigDecimal(String columnName, int scale) throws SQLException;

（41）byte[] getBytes(String columnName) throws SQLException;

（42）java.sql.Date getDate(String columnName) throws SQLException;

（43）java.sql.Time getTime(String columnName) throws SQLException;

（44）java.sql.Timestamp getTimestamp(String columnName) throws SQLException;

（45）java.io.InputStream getAsciiStream(String columnName) throws SQLException;

（46）java.io.InputStream getUnicodeStream(String columnName) throws SQLException;

（47）java.io.InputStream getBinaryStream(String columnName) throws SQLException;

（48）Object getObject(int columnIndex) throws SQLException;

（49）Object getObject(String columnName) throws SQLException;

（50）boolean wasNull() throws SQLException;

在使用了上面的 get***方法后，使用本方法来查看该列是否为空。

以下方法用来对当前记录的列做修改，第一个参数是列号，第二个参数是新值，从方法名可以知道字段的数据类型。另外可以用列名代替列号来做修改，除了第一个参数外，其他和这些方法没有不同，这里就不再列举了。

（51）void updateNull(int columnIndex) throws SQLException;

（52）void updateBoolean(int columnIndex, boolean x) throws SQLException;

（53）void updateByte(int columnIndex, byte x) throws SQLException;

（54）void updateShort(int columnIndex, short x) throws SQLException;

（55）void updateInt(int columnIndex, int x) throws SQLException;

（56）void updateLong(int columnIndex, long x) throws SQLException;

（57）void updateFloat(int columnIndex, float x) throws SQLException;

（58）void updateDouble(int columnIndex, double x) throws SQLException;

（59）void updateBigDecimal(int columnIndex, BigDecimal x) throws SQLException;

（60）void updateString(int columnIndex, String x) throws SQLException;

（61）void updateBytes(int columnIndex, byte x[]) throws SQLException;

（62）void updateDate(int columnIndex, java.sql.Date x) throws SQLException;

（63）void updateTime(int columnIndex, java.sql.Time x) throws SQLException;

以下 3 个方法用来将游标所指示的当前记录插入到数据库中或者在数据库中更新或者删除：

（64）void insertRow() throws SQLException;

（65）void updateRow() throws SQLException;

（66）void deleteRow() throws SQLException;

（67）void close() throws SQLException;

关闭本结果集，释放所有的资源。

11.2.4 Statement

Statement 对象用于将 SQL 语句发送到数据库中。实际上有 3 种 Statement 对象，即 Statement、PreparedStatement（它从 Statement 继承而来）和 CallableStatement（它从 PreparedStatement 继承而来），它们都作为在给定连接上执行 SQL 语句的包容器。分别用于发送特定类型的 SQL 语句，例如 Statement 对象用于执行不带参数的简单 SQL 语句，PreparedStatement 对象用于执行带或不带 IN 参数的预编译 SQL 语句，CallableStatement 对象用于执行对数据库存储过程的调用。

一个 Statement 对象只能有一个 ResultSet 对象，可以用于执行一个静态的 SQL 语句，或一批 SQL 语句，并得到执行结果。

Statement 接口提供了 3 种执行 SQL 语句的方法：executeQuery、executeUpdate 和 execute。使用哪一个方法由 SQL 语句的内容来决定。

方法 executeQuery 用于产生单个结果集的语句，例如 Select 语句。方法 executeUpdate 用于执行 INSERT、Update 或 Delete 语句以及 SQL Ddl（数据定义语言）语句，例如 CREATE Table 和 Drop Table。Insert、Update 或 Delete 语句的效果是修改表中一行或多行中的一列或多列，executeUpdate 的返回值是一个整数，指示受影响的行数（即更新计数）。对于 Create Table 或 Drop Table 等不操作行的语句，executeUpdate 的返回值总为零。

Statement

这意味着在重新执行 Statement 对象之前，需要完成对当前 ResultSet 对象的处理。应注意，继承了 Statement 接口中所有方法的 PreparedStatement 接口都有自己的 executeQuery、executeUpdate 和 execute 方法。Statement 对象本身不包含 SQL 语句，因而必须给 Statement.execute 方法提供 SQL 语句作为参数。

下面是 Statement 接口声明的主要方法。

（1）ResultSet **executeQuery**(String sql) throws SQLException;

执行一条 SQL 查询语句，返回查询结果集对象。

（2）int **executeUpdate**(String sql) throws SQLException;

执行一条 SQL 插入、更新或删除语句，返回操作影响的行数。

（3）void **close**() throws SQLException;

关闭连接的数据库。当一个 Statement 调用 close()后，它产生的 ResultSet 对象（如果有的话）也被关闭。

（4）void **setQueryTimeout**(int seconds) throws SQLException;

设置查询语句的执行等待超时时间，以秒为单位。

（5）int **getQueryTimeout**() throws SQLException;

获得查询语句的执行等待超时时间。

（6）void **cancel**() throws SQLException;

取消当前 SQL 语句（在驱动支持时）。

（7）SQLWarning getWarnings() throws SQLException;

获得当前的警告信息。该警告信息是每执行一条 SQL 就刷新的，所以只能获得最新的。

（8）void **setCursorName**(String name) throws SQLException;

设置游标，游标是当结果集中的记录多于一行时用来获取和标识记录的一个标志。

（9）boolean **execute**(String sql) throws SQLException;

执行一条 SQL 语句。

（10）ResultSet **getResultSet**() throws SQLException;

获得当前执行的结果集。

（11）int **getUpdateCount**() throws SQLException;

获得当前 SQL 执行后影响的行数。

（12）boolean **getMoreResults**() throws SQLException;

获得更多的结果集，该方法在 getResultSet()之后调用，当 SQL 查询有一个以上结果集时使用。

（13）void **setFetchDirection**(int direction) throws SQLException;

设置游标方向，direction 取值有：ResultSet.FETCH_FORWARD，ResultSet.FETCH_REVERSE 和 ResultSet.FETCH_UNKNOWN 3 种，这个游标方向对结果集的 next()方法起作用。

（14）int **getFetchDirection**() throws SQLException;

获得当前游标方向。

（15）void **setFetchSize**(int rows) throws SQLException;

设置当使用游标时要取的行数。

（16）int **getResultSetType**() throws SQLException;

获得结果集类型。

（17）void **addBatch**(String sql) throws SQLException;

在当前 SQL 语句集中添加一条 SQL。

（18）void **clearBatch** () throws SQLException;

清空当前 SQL 语句集。

（19）int[] executeBatch() throws SQLException;

执行一批 SQL 语句集。

（20）Connection getConnection() throws SQLException;

获得当前所在的 Connection 对象。

11.2.5 DriverManager

1 DriverManager

JDBC 的 DriverManager 如同一座桥梁，一方面，它面向程序员提供一个统一的连接数据库的接口；另一方面，它管理 JDBC 驱动程序。DriverManager 类就是这个管理层，对于程序员来说，最简单的连接数据库的方法就是调用 DriverManager 类的方法 getConnection。如果要关心其他连接细节，也可以使用 DriverManager 类的方法 getDriver、getDrivers 和 registerDriver 以及 Driver 类的方法 connect 来共同完成连接数据库的工作。另外，DriverManager 是个静态类。

下面是 DriverManager 类提供的主要方法。

（1）Public static Driver **getDriver**(String url);

根据指定 url 定位一个驱动。

（2）Public static Enumeration **getDrivers**();

获得当前调用访问的所有加载的 JDBC 驱动。

（3）Public static synchronized void **registerDriver**(java.sql.Driver dirver);

登记给定的驱动。

（4）Public static Connection **getConnection**(String url);

使用给定的 url 建立一个数据库连接，并返回一个 Connection 接口对象。

（5）Public static Connection **getConnection**(String url, java.util.Properties info);

同上，并给出一个属性字符串列表，至少包括用户名和密码属性。

（6）Public static Connection **getConnection**(String url, String uer, String password);

使用给定 url、用户名和密码建立一个数据库连接，返回一个 Connection 接口对象。

（7）Public static synchronized void **setLogWriter**(java.io.PrintWriter out);

设置 PrintWriter 类型的日志对象，该类型适用于 DriverManager 和所有的驱动程序。若参数设置为 null，则表示不写日志。

（8）Public static java.io.PrintWriter **getLogWriter**();

获得 PrintWriter 日志对象。

（9）Public static synchronized void **setLogStream**(java.io.PrintStream out);

设置 PrintStream 类型的日志对象，该类型适用于 DriverManager 和所有的驱动程序。若参数设置为 null，则表示不写日志。

（10）Public static java.io.PrintStream **getLogWriter**();

获得 PrintStream 日志对象。

（11）Public static synchronized void **println**(String message);

输出信息 message 到日志。

（12）Public static void **setLoginTimeout**(int seconds);

设置登录超时时间，以秒为单位。

（13）Public static int **getLoginTimeout**();

获得登录超时时间。

2　　DriverManger

（1）加载可用的驱动程序

DriverManager 类包含一列 Driver 类，它们已通过调用方法 DriverManager.registerDriver 对自己进行了注册。所有 Driver 类都必须包含有一个静态部分。它创建该类的实例，然后在加载该实例时将 DriverManager 类进行注册。这样，用户正常情况下将不会直接调用 DriverManager.registerDriver；而是在加载驱动程序时由驱动程序自动调用。加载 Driver 类，然后自动在 DriverManager 中注册的方式有两种。

① 调用方法 Class.forName

这将显式地加载驱动程序类。由于这与外部设置无关，因此推荐使用这种加载驱动程序的方法。以下代码加载类 ant.db.Driver：Class.forName（"ant.db.Driver"）。

如果将 ant.db.Driver 编写为加载时创建实例，并调用以该实例为参数的 DriverManager.registerDriver（本该如此），则它在 DriverManager 的驱动程序列表中，并可用于创建连接。

② 将驱动程序添加到 Java.lang.System 的属性 jdbc.drivers 中

这是一个由 DriverManager 类加载的驱动程序类名的列表，由冒号分隔，初始化 DriverManager 类时，它搜索系统属性 jdbc.drivers，如果用户已输入了一个或多个驱动程序，则 DriverManager 类将试图加载它们。以下代码说明程序员如何在~/.hotJava/properties 中输入

三个驱动程序类（启动时，HotJava 将把它加载到系统属性列表中）。

```
jdbc.drivers=server1.bah.Driver：ant.sql.Driver：bad.test.ourDriver;
```

对 DriverManager 方法的第一次调用将自动加载这些驱动程序类。

Class.forName

DriverManager

jdbc.drivers

在以上两种情况中，新加载的 Driver 类都要通过调用 DriverManager.registerDriver 类进行自我注册。如上所述，加载类时将自动执行这一过程。由于安全方面的原因，JDBC 管理层将跟踪哪个类加载器提供哪个驱动程序。这样，当 DriverManager 类打开连接时，它仅使用本地文件系统或与发出连接请求的代码相同的类加载器提供的驱动程序。

（2）建立连接

加载 Driver 类并在 DriverManager 类中注册后，它们即可用来与数据库建立连接。当调用 DriverManager.getConnection 方法发出连接请求时，DriverManager 将检查每个驱动程序，查看它是否可以建立连接。

有时可能有多个 JDBC 驱动程序可以与给定的 URL 连接。例如，与给定远程数据库连接时，可以使用 JDBC-ODBC 桥驱动程序、JDBC 到通用网络协议驱动程序或数据库厂商提供的驱动程序。在这种情况下，测试驱动程序的顺序至关重要，因为 DriverManager 将使用它所找到的第一个可以成功连接到给定 URL 的驱动程序。

首先 DriverManager 试图按注册的顺序使用每个驱动程序（jdbc.drivers 中列出的驱动程序总是先注册）。它将跳过代码不可信任的驱动程序，除非加载它们的源与试图打开连接的代码的源相同。它通过轮流在每个驱动程序上调用方法 Driver.connect，并向它们传递用户开始传递给方法 DriverManager.getConnection 的 URL 来对驱动程序进行测试，然后连接第一个认出该 URL 的驱动程序。这种方法初看起来效率不高，但由于不可能同时加载数十个驱动程序，因此每次连接实际只需几个过程调用即可。

以下代码是通常情况下用驱动程序（例如 JDBC-ODBC 桥驱动程序）建立连接所需所有步骤的示例：

```
Class.forName（"sun.jdbc.odbc.JdbcOdbcDriver"）；//加载驱动程序
String url = "jdbc：odbc：test"；
DriverManager.getConnection（url，"userID"，"passwd"）；
```

11.3 JDBC 程序示例

在编写 JDBC 程序之前，应该做两件事情，首先要确定本机是否能连到数据库上，然后要下载 JDBC 类库和该数据库的相应 JDBC 驱动类库，并把这些类库放到本机的 class 路径下。

驱动类库可以到该数据库公司的技术主页上下载，JDBC 类库可以到 Sun 的网站上下载最新版本：http://java.sun.com/products/jdbc/download.html。

JDBC 程序由以下步骤组成：

加载驱动程序→连接数据库→声明 SQL 语句→绑定输入参数→执行 SQL→操作结果集→关闭

Connection Statement

Java

例 11.1 是一个 JDBC 访问 oracle 数据库的例子。

【例 11.1】

```
//引入 JDBC 类库
import java.sql.*;

class JDBCSample
{
   public static void main (String args [])
         throws SQLException, ClassNotFoundException
   {
      try {
      // 加载 oracle 的 JDBC 驱动
           Class.forName ("oracle.jdbc.driver.OracleDriver");
         } catch(java.lang.ClassNotFoundException e) {
              System.err.print("ClassNotFoundException: ");
              System.err.println(e.getMessage());
         }

       try{
   // 连到数据库，在 connection URL 中，test 是数据库名，scott 和 tiger 是用户名和密码
   Connection conn = DriverManager.getConnection ("jdbc:oracle:oci8:@test", "scott", "tiger");

   // 声明一个 Statement 对象
   Statement stmt = conn.createStatement ();

   // 从 STUDENT 表中选择 NAME 列
   ResultSet rset = stmt.executeQuery ("select NAME from STUDENT");
```

```
    // 遍历结果集并将每条记录的 NAME 列打印出来
    while (rset.next ())
System.out.println (rset.getString (1));

        //关闭 stmt 和 conn
        stmt.close();
        conn.close();
} catch(SQLException ex) {
                System.err.println("SQLException: " + ex.getMessage());
            }
    }
}
```

第12章

网络编程

Java是针对网络而设计的编程语言，它的网络通讯类库屏蔽了平台的差别和低层细节，实现了强大的网络访问功能。

本章的主要内容包括:

- 网络技术基础
- URL
- Inet Address 类
- TCP Socket 编程
- UDP Socket 编程

12.1 网络技术基础

Java 是针对网络环境的程序设计语言，Java 中提供了强有力的网络支持机制。Java 通过采用面向对象的方法，隐藏了网络通讯程序设计中的一些繁琐的细节，并为用户提供了与平台无关的使用接口。

计算机在网络上应用 TCP 和 UDP 协议进行通讯，每个协议栈都由四层组成，如图 12.1 所示。每个层都有明确定义的功能及用途。每一层都能导出经准确定义的接口，在它上面或下面的层可通过该接口与之通信。这种分层结构具有多方面的优点。除了能简化协议栈的设计之外，还能简化它的使用设计。之所以能得到简化，是由于每个层都只与紧靠在它上面或下面的层打交道。只要一个层提供的服务确定下来，相应的接口也会确定下来，所以每个层都可以独立设计。如图 12.1 所示。

图 12.1 网络协议的层次结构

当使用 Java 编写网络通讯的程序时，通常是在应用层，不必关心 TCP 和 UDP 的层次的细节，只要用 java.net 包中的类即可实现与系统无关的网络通讯服务。

TCP 是一个可靠的、面向连接的、连续的、流的协议。它保证了数据从连接的一方传递到另一方，并且传递的顺序与发送的顺序一样，否则会提示出错。当应用程序需要一个可靠的、点对点的连接进行通讯时，就调用 TCP 协议，例如 HTTP、FTP、Telnet 等应用程序。

对许多应用程序来说，这种可靠性的保证是能否成功地传输信息的关键。然而，这种可靠的传输方式并非对所有的应用程序都适合。例如，如果 ping 指令需要以乱序的数据包来判断网络上的连接是否正常，可靠传输就会使这个服务完全失败。

用户数据报协议（UDP）是一种不可靠的通讯方法。UDP 不像 TCP，它没有检测错误的机制，也不重发丢失或损坏的数据。UDP 发送的数据包是相互独立的，这种数据包称为数据报。发送数据报就像寄信一样，信件的传递顺序并不重要，信件是彼此无关的。UDP 是面向非连接的。

端口（Port）与 IP 地址一起为网络通信的应用程序之间提供了一种确切的地址标识功能。计算机的网络地址是一个 32 位的 IP 地址，IP 地址用来把数据传送到网络上的目的计算机上。端口用一个 16 位的数来表示，TCP 和 UDP 都是通过这个端口来指明数据要发往的应用程序。

如图 12.2 所示。

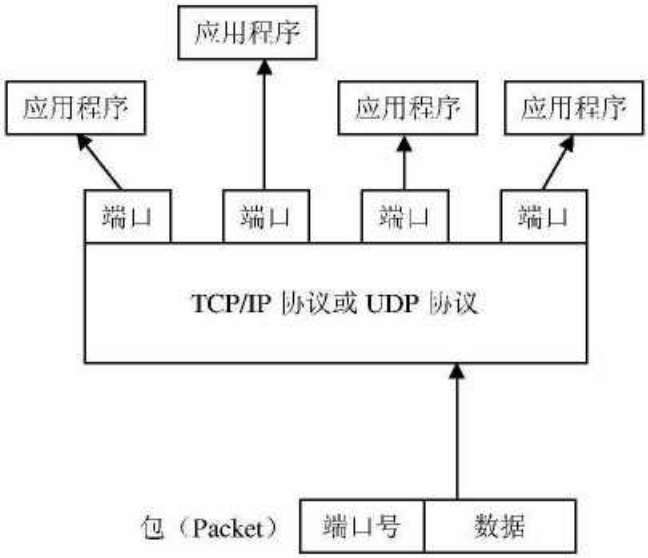


图 12.2 端口

TCP 和 UDP 协议利用端口把流入的数据映射到运行在计算机上的特定的进程。端口号是从 0 到 65535（因为端口是用 16 位数表示）。0～1023 的端口号被保留，它们被许多知名的服务占据，如：HTTP、ftp 和其他的系统服务。

Java.net 类中提供了两个不同层次的网络支持机制：用 URL 访问网络资源和用 socket 通讯。两层次上的网络支持机制分别面向两大类主要的应用需求：一类是针对访问 Internet 尤其是 WWW 资源的应用，Java 提供了支持用统一资源定位符 URL 访问网络资源的一组类，这些类对 HTTP 协议提供了更广泛的支持，用户不需要考虑 URL 中标识的协议处理细节；另一类主要是针对客户 / 服务器模式的应用和实现某些特殊的协议的应用，它们的通讯过程是基于 TCP / IP 协议中的传输层接口 socket 来实现的。通过 java.net 类库，Java 程序可以利用 TCP 和 UDP 在网络上通讯。URL、URLConnection、Socket、SocketSever 等类都是利用 TCP 在网络上通讯的。DatagramPacket 和 DatagramSocketr 以及 MulticastSocket 类是利用 UDP 进行通讯的。

Java 是面向网络环境发展而来的，因此其网络通讯支持能力还是比较强大的，Java.net packet 中提供的非常丰富的类对各种网络通讯需求提供了灵活方便的支持。本章简要介绍的是 Java 中最常用的一些网络程序设计方法，例如用 URL 类访问网络资源，以及 Java 的 socket 通讯机制等，其他更多详细内容请参考 java API 文档。

12.2 URL

12.2.1 URL 的概念

URL 是 Uniform Resource Locator（统一资源定位符）的缩写，是以一个字符串形式表示的 Internet 网络上某一资源的位置。URL 由两部分组成：协议标识符和资源名，两者用“:”分开。

协议标识符指出了访问该资源应使用的网络协议，如 http、ftp、gopher、news 等。资源名是网络资源的完整地址，包括主机名（域名）、端口号、文件名或文件内部的一个引用，但并非所有的 URL 都包含这些内容，其格式依赖于访问它所用的协议类型。下面是几个 URL 的例子：

http://java.sun.com

http://java.sun.com/j2se/1.4/download.html

http://www.ncsa.uiuc.edu:8080/demoweb/url-primer.html

java.net 包中包含了一个 URL 类，可以通过 URL 类创建的对象来代表一个网络资源。Java 应用程序中创建了 URL 对象之后，通过程序可以打开一条和资源链接的通讯链路，并利用该链路进行资源信息流的读写。

12.2.2 URL 类

1 URL

通过使用 URL 类的构造方法创建 URL 对象，该对象代表了用户需要访问的资源。URL 类提供了下面构造方法：

（1）public URL(String protocol, String host, int port, String file) throws MalformedURL-Exception;

（2）public URL(String protocol, String host, String file) throws MalformedURLException;

（3）public URL(String protocol, String host, int port, String file，URLStreamHandler handler) throws MalformedURLException;

（4）public URL(String spec) throws MalformedURLException;

（5）public URL(URL context, String spec) throws MalformedURLException;

（6）public URL(URL context, String spec, URLStreamHandler handler) throws Malformed-URLException;

表 12.1 列出了 URL 构造函数中各参数所代表的含义。

表 12.1　　URL 构造函数中各个参数的含义

参 数 名	含　义
protocol	协议的名称，如 http，ftp 等
host	主机名或 IP 地址
port	端口号
file	主机文件
handler	一个 URLStreamHandler 对象。URLStreamHandler 类是所有流协议处理类的超类，它是一个抽象类，所谓流协议处理就是为特定的协议类型（如 http）生成一个链接
spec	一个当作 URL 来分析的字符串
context	一个 URL 对象，和 spec 参数一起使用，通常作为 spec 参数的上下文，即当 spec 参数不完全时，提供一种补充。例如，当分析了 spec 参数后，如果没有指定协议名，则使用 context 参数中的协议名；如果两者的协议名相同，则 context 参数中的 host、port、file 等参数将在新创建的 URL 中使用

如果创建 URL 对象时所用的协议标识符是一个空协议或未知协议，则构造方法将抛出一个 MalformedURLException 异常，Java 应用程序中可以捕获并处理该异常。

URL 对象创建后就一直代表某一资源，不能动态改变其属性去代表另一资源。创建 URL 对象的例子如下：

```
URL url1 = new URL("http", "www.sun.com", 80, "index.html");
URL url2 = new URL("http", "www.sun.com", "index.html");
URL url3 = new URL("http://www.sun.com/index.html");
URL urlBase = new URL(http://www.sun.com/);
URL url4 = new URL(urlBase, "/j2se/1.4/download.html");
```

2　　URL

一个 URL 对象创建后，其属性是不能改变的，但是可以通过类 URL 提供的方法来获取这些属性，这些方法如下所述。

（1）public int getAuthority();得到 URL 对象的授权部分。

（2）public final Object getcontent() throws java.io.IOException;得到 URL 对象的内容。

（3）public int getdefaultPort();得到与 URL 对象相关的默认端口号。

（4）public String getHost();得到 URL 对象的主机名。

（5）public String getFile();得到 URL 对象的文件名部分。

（6）public String getPath();得到 URL 对象的路径部分。

（7）public String getProtocol();得到 URL 对象的协议名部分。

（8）public String getRef();得到 URL 对象的引用。

（9）public int getPort();得到 URL 对象的端口号。

…

URL　　URL
HTTP URL　　HTTP URL
URL　　URL　　HTTP

例如，我们可以用例 12.1 中的代码获取一个 URL 的信息。

【例 12.1】

```
import java.net.*;
import java.io.*;
class ParseURL
{
    public static void main(String[] args)
    {
        URL aURL = null;
        try
        {
```

```
            aURL = new
            URL("http://java.sun.com:80/tutorial/intro.html#DOWNLOADING");
            System.out.println("protocol = " + aURL.getProtocol());
            System.out.println("host = " + aURL.getHost());
            System.out.println("filename = " + aURL.getFile());
            System.out.println("port = " + aURL.getPort());
            System.out.println("ref = " + aURL.getRef());
        } catch (MalformedURLException e)
        {
            System.out.println("MalformedURLException: " + e);
        }
    }
}
```

3

URL 对象创建后，只是在应用程序中代表一个网络资源，而用户的主要目的是访问该资源的信息。URL 对象提供了一个 openStream（）方法，此方法调用成功将返回一个输入流类 InputStream 的对象，该对象联接着一条和资源方的通讯链路，这样，访问资源的过程就转变成了使用流对象的过程，因而可以使用标准的 InputStream 类方法来从 URL 中读取资源数据，就如同从输入流中读取资源数据一样方便。例 12.2 中的程序是使用 openStream()方法从 URL “http://java.sun.com/”中获取输入流，然后从输入流中读取内容，并把读到的内容显示出来。

【例 12.2】

```
import java.net.*;
import java.io.*;
class OpenStreamTest
{
    public static void main(String[] args)
    {
        try
        {
            URL sunUrl = new URL("http://java.sun.com/");
            DataInputStream dis = new
                    DataInputStream(sunUrl.openStream());
            String inputLine;
            while ((inputLine = dis.readLine()) != null)
            {
                System.out.println(inputLine);
            }
            dis.close();
        } catch (MalformedURLException me)
```

```
            {
                System.out.println("MalformedURLException: " + me);
            } catch (IOException ioe)
            {
            System.out.println("IOException: " + ioe);
            }
            }
}
```

运行这个程序，将会看到来自 http://java.sun.com/的 html 文件的 html 命令和文本内容显示到命令行的窗口上。

12.2.3 URLConnection 类

由 URL 的 openStream（）方法，只能从网络上读取数据。有些 URL，例如许多连接到 CGI 模板上的 URL，允许甚至要求用户向该 URL 上写入信息（例如：一个查询模板可能需要向 URL 上写入详细的查询数据以后才能实现查询），这时候就要用到 URLConnection 类了。

URLConnection 类也在包 java.net 中定义，它表示 Java 程序和 URL 在网络上的通讯链接。URL 对提供了一个 openConnection（）的方法，调用该方法将在 Java 应用程序和资源服务器之间按给定的访问协议建立一条通讯链路。方法调用成功时，将创建一个对象，用来代表建好的通讯链路，并返回一个 URLConnection 类型的对象的引用，否则产生 IOException 异常。该通讯链路的建立过程对用户隐含了 URL 中指定协议与资源服务器进行连接和握手的过程，而且在以后的数据传输过程中，该链路会自动将传给它的数据转换成协议要求的格式，而不必关心协议的处理方法，这大大简化了编程。

例 12.3 中的程序与例 12.2 实现的功能是一样的。但是，例 12.3 中的程序不是直接从 URL 上打开了一个流，而是显式地打开了一个到该 URL 的连接，从而获取到这个连接上的输入流，并从这个输入流读取。

【例 12.3】

```
import java.net.*;
import java.io.*;
class ConnectionTest
{
    public static void main(String[] args)
    {
        try
        {
            URL sunUrl = new URL("http://java.sun.com/");
            URLConnection sunConnection = sunUrl.openConnection();
            DataInputStream dis = new
                DataInputStream(sunConnection.getInputStream());
            String inputLine;
```

```
            while ((inputLine = dis.readLine()) != null)
            {
                System.out.println(inputLine);
            }
            dis.close();
        }
        catch (MalformedURLException me)
        {
            System.out.println("MalformedURLException: " + me);
        }
        catch (IOException ioe)
        {
            System.out.println("IOException: " + ioe);
        }
    }
}
```

例 12.4 中通过 URLConnection 对象来运行 java.sun.com 上一个叫 backwards 的模板，这个模板从标准输入读取一个字符串并将其反转后写到标准输出。这个模板需要的输入格式是：

```
string=string_to_reverse
```

这里，string_to_reverse 这个参数就是要反转的字符串。

【例 12.4】

```
import java.io.*;
import java.net.*;
public class ReverseTest
{
    public static void main(String[] args)
    {
        try
        {
            if (args.length != 1)
            {
                System.err.println("Usage: java ReverseTest string_to_reverse");
                System.exit(1);
            }
            String stringToReverse = URLEncoder.encode(args[0]);
            URL url = new URL("http://java.sun.com/cgi-bin/backwards");
            URLConnection connection = url.openConnection();
            PrintStream outStream = new
                        PrintStream(connection.getOutputStream());
```

```
                outStream.println("string=" + stringToReverse);
                outStream.close();
                DataInputStream inStream = new
                                DataInputStream(connection.getInputStream());
                String inputLine;
                while ((inputLine = inStream.readLine()) != null)
                {
                    System.out.println(inputLine);
                }
                inStream.close();
            }
            catch (MalformedURLException me)
            {
                System.err.println("MalformedURLException: " + me);
            }
            catch (IOException ioe)
            {
                System.err.println("IOException: " + ioe);
            }
        }
    }
```

该程序首先通过调用 URL 对象的 openConnection()方法创建了一条通讯链路，并返回 URLConnection 类型的对象；然后调用 URLConnection 的 getOutputStream()方法创建一个输出流对象，该流对象被联接到通讯链路上，从而也就被联接到资源方的 backwards 脚本的标准输入上，因此写往该流对象的数据将作为 backwards 的输入信息。资源方 backwards 脚本处理完输入数据后，将结果返回给用户程序，用户程序相应地调用 URLConnection 的 getInputStream()方法创建一个输入流对象，该对象通过通讯链路连接到资源方的 backwards 脚本的标准输出，因此 backwards 脚本的输出数据可以从该输入流中读出，最后将用户输入串的倒序形式打印在屏幕上。

例如，在程序运行时输入“Reverse Me”后，屏幕上将出现：

```
reversed is :
eM esreveR
```

12.3 InetAddress 类

要建立 Internet 网络上的连接，地址是必不可少的。Java 通过 InetAddress 对象来存储远程系统的 Internet 地址，该对象含有与 Java 网络编程有关的许多变量和方法。InetAddress 类没有显式的构造函数，为了创建一个 InetAddress 对象，必须使用构造方法，构造方法就是一

个能返回类实例的静态方法。有 3 个方法（getLocalHost、getByName 和 getAllByName）可以用来创建 InetAddress 实例，例如：

```
try
{
    InetAddress myHost = InetAddress.getLocalHost();
    InetAddress remoteAdderss = InetAddress.getByNamel("java.sun.com");
    InetAddress allremoteAddress = InetAddress.getAllByName("java.sun.com");
}catch(UnknownHostException excpt)
{
    System.err.println("Unknown host: " + excpt);
}
```

getLocalHost()方法返回代表本地主机的 InetAddress 对象。getByName（hostname）返回由 hostname 所指定机器的 InetAddress 对象。如果这个方法找不到 hostname 机器，则抛出一个 UnknownHostException 异常。在 Web 服务世界里，可以用同样的名字代表一组机器。getAllByName（hostname）方法返回一组具有相同名字的 InetAddress 对象，如果一台机器都没找到，同样会抛出 UnknownHostException 异常。

在 InetAddress 类中，还提供了下述一些常用的方法。

（1）public boolean isMulticastAddress();判断是否是一个 IP 多播地址。

（2）public String getHostName();得到 InetAddress 对象的主机名的字符串表示。

（3）public byte[] getAddress();得到 InetAddress 对象的地址。

（4）public String getHostAddress();得到 InetAddress 对象的“%d.%d.%d.%d”格式的四字节 Internet 地址，

例 12.5 是利用 InetAddress 类来得到域名对应的 IP 地址。

【例 12.5】

```
import java.io.*;
import java.net.*;
public class GetIP
{
    public static void main(String[] args)
    {
        if(args.length ! = 1)
        {
            System.err.println"Usage:getIP DNSname");
            System.exit(1);
        }
        InetAddress a = InetAddress.getByName(args[0]);
        System.out.println(a);
    }
}
```

12.4 TCP Socket 编程

12.4.1 Socket 通讯基础

Java 提供了 URL 类和 URLConnection 类来在一个相对比较高的层次上进行网络通讯，以实现对 Internet 上的资源的访问。有时候用户的程序需要低层的网络通讯，例如客户服务器应用程序以及某些特殊的协议的应用。

在各种网络的客户 / 服务器应用中，客户和服务器之间使用的通讯组件是多种多样的，大部分通讯组件内最低层的核心通讯机制都是使用传输层接口的 Socket 机制来实现的。Socket 是在两个程序间用来进行双向数据传输的网络通讯端点，一般由一个地址加上一个端口号来标识。每个服务程序在一个众所周知的端口上提供服务，

TCP 提供了一个可靠的、端到端的通讯通道，例如，client-server 应用程序在 Internet 上就使用这个通道来通讯。

12.4.2 TCP Socket 通讯程序的开发

采用 TCP 通讯方式的客户 / 服务器应用程序的开发主要由以下几个步骤来完成。

① 初始化服务器，建立 ServerSocket 对象，等待客户机的连接请求；

② 初始化客户机，建立 Socket 对象，向服务器发出连接请求；

③ 服务器响应客户机，建立连接；

④ 客户机发送请求数据到服务器方；

⑤ 服务器接收客户机请求数据；

⑥ 服务器处理请求数据，并返回处理结果给客户机；

⑦ 客户机接收服务器返回的结果；

⑧ 重复④~⑦步，直到客户机结束对话为止；

⑨ 中断连接，结束通讯。

对应上述过程，J2SE 中在 java.net 包中有两个类：Socket 和 ServerSocket，分别用于在客户机和服务器上创建和管理通讯 socket。另外，SocketImpl、InetAddress 和 SocketOptions 等也是重要的相关类。关于这些类的具体信息，可以查看有关的 Java API 文档，这里只是简单介绍一下客户和服务器程序的编写。

1

下面介绍开发客户机应用的通讯工作过程。

(1)调用 Socket 类的构造函数，以服务器主机地址信息和端口号为参数，创建一个 Socket 对象。创建过程包含了一个向服务器方请求建立通讯链路的过程，只有通讯链路建成后才能生成对象，该对象对应通讯链在客户机上的端点，以抽象的形式向用户隐藏了具体平台的细节。Socket 类的构造函数主要有以下几种形式：

① public Socket();

② protected Socket(SocketImpl impl) throws SocketExcetption;

③ public Socket(String host, int port);

④ public Socket(InetAddress adderss, int port) throws IOException;

⑤ public Socket(String host, int port, InetAddress localAddr, int localPort)throws IOException;

⑥ public Socket(InetAddress addersss, int port, InetAddress localAddr, int localPort)throws Ioexception;

⑦ public Socket(String host, int port, boolean stream)throws Ioexception;

⑧ public Socket(InetAddress host, int port ,boolean stream) throws IOException;

（2）有了客户机通讯 Socket 之后，下面就要读写数据了。Socket 类提供了方法 getInputStream()和 getOutputStream()来得到对应的输入／输出流，以进行读／写操作，这两个方法分别返回 InputStream 和 OutputStream 类对象。这样，通过使用 Socket 类，网络 I/O 也转变成为使用流对象的过程。

（3）使用流对象的方法读/写字节流数据。由于流连接着通讯所用的 Socket，Socket 又是和服务方建立连接的通讯链路的端点，因此数据将通过建好的通讯链路与服务器方进行传输。为了便于读／写数据，可以在返回的输入／输出流对象上建立过滤流，如：DataInputStream、DataOutputStream 或 PrintStream 类对象。

① PrintStream pStm = new PrintStream(new BufferedOutputStream(socket.getOutputSteam()));

② DataInputStream iStm = new DataIputStream(socket.getInputStream());

（4）工作完毕，关闭所有的与 Socket 相关的输入／输出流，用 Socket 对象的 close()方法关闭通讯 Socket。

例 12.5 中的客户端程序 EchoTest 通过一个 Socket 连接到标准 Echo 服务器上（在端口 7 上）。客户程序对 Socket 进行读/写操作。EchoTest 通过向 Socket 写文本，把从标准输入键入的所有文本发送给 Echo 服务器。服务器把它从客户端收到的所有输入通过到客户端的 Socket 回显给客户端，客户程序读取从服务器端送回的数据并显示这些数据。

【例 12.6】

```
import java.io.*;
import java.net.*;
public class EchoTest
{
    public static void main(String[] args)
    {
        Socket echoSocket = null;
        DataOutputStream os = null;
        DataInputStream is = null;
        DataInputStream stdIn = new DataInputStream(System.in);
        try
        {
            echoSocket = new Socket("netlab", 7);
```

```
            os = new DataOutputStream(echoSocket.getOutputStream());
            is = new DataInputStream(echoSocket.getInputStream());
        }
        catch (UnknownHostException e)
        {
            System.err.println("Don't know about host: netlab");
        }
        catch (IOException e)
        {
            System.err.println("Couldn't get I/O for the connection to: netlab");
        }
        if (echoSocket != null && os != null && is != null)
        {
        try
        {
                String userInput;
                while ((userInput = stdIn.readLine()) != null)
                {
                    os.writeBytes(userInput);
                    os.writeByte('\n');
                    System.out.println("echo: " + is.readLine());
                }
                os.close();
                is.close();
                echoSocket.close();
            }
            catch (IOException e)
            {
            System.err.println("I/O failed on the connection to: netlab");
            }
        }
    }
}
```

2

下面介绍用 Java 开发的服务器方程序进行 Socket 通讯的工作过程。

（1）调用 ServerSocket 类的构造方法以某端口号为参数，创建一个 ServerSocket 对象，代表服务器在指定端口监听客户机请求的 Socket。ServerSocket 类的构造方法有 3 种。

① public ServerSocket(int port) throws IOException;

在指定端口创建一个监听 ServerSocket，如果端口号为 0，则在任意一个空闲的端口创建

ServerSocket，缺省允许 50 个客户等待在连接队列中，如果队列满了，服务器将拒绝新的连接请求。

② public ServerSocket(int port, int count) throws IOException;

该构造函数可以改变连接队列的长度（count），允许更多或更少的客户等待服务器的处理。

③ public ServerSocket(int port, int count, InetAddress localAddr) throws IOException;

构造函数允许指定用于连接的本地监听端口。如果机器有多个 IP 地址，则这个构造函数允许提供指定的 IP 地址（localAddr）。

（2）服务程序使用 ServerSocket 对象的 accept 方法来接收来自某客户程序的连接请求。此时，服务方将一直阻塞，直到收到从客户程序发来的连接请求为止，然后该方法将返回一个新建 Socket 类的实例对象，该对象代表和客户方建立的通讯链路在服务程序内的通讯端点。

（3）使用新建的 Socket 对象创建输入/输出流对象。

（4）使用流对象的方法来完成和客户方的数据传输，按照约定协议处理客户方请求的数据，并将结果返回给客户方。

（5）客户方工作完毕，服务程序关闭和客户方通讯的流和通讯 Socket。

（6）服务程序运行结束前，还应调用 ServerSocket 对象关闭监听 Socket。

例 12.7 是一个服务程序的例子，该例子的功能和标准 echo 服务器程序相同，客户程序只需要在创建 Socket 对象时使用服务器的监听端口 7800 就可以使用此服务程序了。当输入“EXIT”时，该服务程序终止运行

【例 12.7】

```
import java.io.*;
import java.net.*;
public class EchoServer1
{
    public static void main(String[] args )
    {
        try
        {
            ServerSocket s = new ServerSocket(7800);
            Socket incoming = s.accept( );
            BufferedReader in = new BufferedReader
              (new InputStreamReader(incoming.getInputStream()));
            PrintWriter out = new PrintWriter
                  (incoming.getOutputStream(), true /* autoFlush */);
            out.println( "Hello! Enter EXIT to exit." );
            boolean done = false;
            while (!done)
            {
                String line = in.readLine();
```

```
                    if (line == null) done = true;
                    else
                    {
                        out.println("Echo: " + line);
                        if (line.trim().equals("EXIT"))
                            done = true;
                    }
                }
                incoming.close();
            }
            catch (Exception e)
            {
                System.out.println(e);
            }
        }
    }
```

12.5 UDP Socket 编程

12.5.1 概念

UDP Socket 与 TCP Socket 的不同之处在于它们的通信协议。TCP（Transport Control Protocol，传输控制协议）是面向连接的、可靠的，它在通讯双方之间建立了一条可靠的通讯链路，保证数据按序无误地到达目的地。而 UDP（User Datagram Protocol，用户数据报协议）是无连接的，每个数据报都是一个独立的信息单元，包括完整的源地址或目的地址，它在网络上以任何可能的路径传往目的地，因此能否到达目的地，到达目的地的时间以及内容的正确性都是不能保证的。

UDP

TCP

socket

TCP

UDP

64KB TCP

Socket

java.net 包中提供了两个类（DatagramSocket 和 DatagramPacket），用来支持数据报通讯。DatagramSocket 用于在程序之间建立传送数据报的通讯连接，DatagramPacket 用来表示一个数据报。

12.5.2 UDP Socket 通讯程序的开发

与 TCP 通讯方式相比，UDP Socket 程序的开发比较简单，双方不需要建立连接的过程。

1

（1）调用 DatagramSocket 类的构造函数，创建数据报通讯的 Socket 对象。DatagramSocket 类的构造函数有以下几种形式。

① public DatagramSocket();此时系统会自动给该 Socket 安排一个本地未用端口。

② public DatagramSocket(int port);以指定的 Socket 端口号创建一个 DatagramSocket 对象。

③ public DatagramSocket(int port，InetAddress laddr);创建一个 DatagramSocket 对象，以绑定到指定的本地地址上。

④ public DatagramSocket(SocketAddress bindaddr);创建一个 DatagramSocket 对象，以绑定到指定的本地 Socket 地址上。

（2）构造应用程序的载体——数据报。每个数据报是以一个 DatagramPacket 对的形式出现的，该对象封装了数据报数据、数据长度、数据报地址信息等。对发送方而言，DatagramPacket 对象的构造形式是：

public DatagramPacket(byte[] ibuf, int length, InetAddress iaddr, int iport);

其中，ibuf 中存放数据报数据，length 是数据报中数据的长度，iaddr 和 iport 给出了目的地址和端口号。

DatagramPacket 类还提供了下述一些方法以获取数据报中的信息。

① public synchronized int getLength();得到数据报中数据块的长度。

② public synchronized byte［］ getData();得到数据报中的数据。

③ public synchronized InetAddress getAddress();返回 InetAddress 对象，表示数据报地址信息。

（3）发送数据报。通过调用 DatagramSocket 对象的 send()方法，并以 DatagramPacket 对象为参数，将 DatagramPacket 对象中信息数据组成数据报送入网络。

（4）接收数据报。为了接收从服务器返回的结果数据报，需要再次创建一个 DatagramPacket 对象，由于要接收的数据报的信息只有在数据报被收到时才能确定，因此，此处使用 DatagramPacket 类的接收方构造函数：

DatagramPacket(byte[] ibuff, int ilenght);

其中 ibuff 接收数据报的数据部分，ilength 是该部分数据的长度。

以上面创建的 DatagramPacket 对象为参数，DatagramSocket 对象的 receive()方法可以完成接收数据报的工作。

（5）处理接收缓冲区内的数据，获取服务结果。

（6）通讯过程完毕，用 DatagramSocket 对象的 close()方法关闭数据报通讯 Socket。当然，Java 自己也会自动关闭 Socket，释放 DatagramSocket 和 DatagramPacket 所占用的资源。但是，作为一种良好的编程习惯，还是要显式地予以关闭。

2

不同于 TCP 通讯方式，在 UDP 通讯中，通讯双方并不需要建立连接。所以，服务器应

用程序的通讯过程与客户端应用程序的通讯过程使非常相似的，也要建立数据报通讯 DatagramSocket，构建数据报文包 DatagramPacket，接收数据报和发送数据报，以及处理接收缓冲区内的数据。通讯完毕后，关闭数据报通讯 Socket。

3

例 12.8 给出一个基于 UDP Socket 的 Daytime 例子，客户机向服务器 13 端口发出请求，服务器返回自己的当前时间。

【例 12.8】

（1）服务器

```
import java.io.*;
import java.net.*;
import java.util.*;
public class UdpDaytimeServer
{
        public static void main(String[] args)
{
        DatagramSocket timeSocket;
        Date currDate;
        byte[] inBuffer;
        byte[] outBuffer;
        DatagramPacket request;
        DatagramPacket reply;
        InetAddress clientAddress;
        int clientPort;
        try{
          //创建 socket
          timeSocket = new DatagramSocket(13);
          try{
            while(true){
              inBuffer = new byte[1];
              request = new DatagramPacket(inBuffer,inBuffer.length);
              timeSocket.receive(request);
              //得到客户机的 IP 地址和端口号
              clientAddress = request.getAddress();
              clientPort = request.getPort();
```

```
                    //创建响应数据报
                    currDate = new Date();
                    outBuffer = currDate.toString().getBytes();
                    reply = new DatagramPacket
(outBuffer, outBuffer.length, clientAddress, clientPort);
                        //将响应发回给客户机
                        timeSocket.send(reply);
                    }
                }finally {
                    timeSocket.close();    //关闭 socket
                }
            }catch(Exception e ){
                System.out.println(e);
            }
            System.exit(0);
        }
    }
```

（2）客户机

```
import java.io.*;
import java.net.*;
import java.util.*;
public class UdpDaytimeClient
{
    public static void main(String[] args)
    {
        String host;
        InetAddress hostAddress;
        byte[] inBuffer = new byte[512];
        byte[] outBuffer = new byte[1];
        DatagramPacket request;
        DatagramPacket reply;
        DatagramSocket timeSocket;
        if(args.length <1 )
        {
            System.out.println("Usage: UdpDaytime host");
        }
        else
        {
            try{
```

```
                    //获取主机 IP
                    host = args[0];
                    hostAddress = InetAddress.getByName(host);
                    //创建数据报
                    timeSocket = new DatagramSocket();
                    //创建请求和响应 buffer
                    request = new DatagramPacket(outBuffer, outBuffer.length, hostAddress, 13);
                    reply = new DatagramPacket(inBuffer, inBuffer.length);
                    //发送请求和读取响应
                    try
                    {
                        //将 socket 超时设置为 5 秒
                        timeSocket.setSoTimeout(5*1000);
                        //发送请求和得到响应
                        timeSocket.send(request);
                        timeSocket.receive(reply);
                        System.out.println("Reply received: "+ new String(inBuffer, 0, reply.getLength()));
                        }
                    catch(Exception e){
                    System.out.println(e);
                    }
                    finally{
                    timeSocket.close();
                    }
                }
                catch(Exception e){
                    System.out.println(e);
                }

                System.exit(0);
            }
        }
    }
```

12.5.3 IP 多播程序的开发

IP 多播（multicast）是针对点到点的传送和广播传送两种方式而言的，它是指在一定的组内对其成员进行的、有限的广播，组中的某个成员发出信息，接收对象为组中的其他所有成员。多播功能类似于一个单一信息发送到多个接收者的广播，但仅仅发送给那些等待它的接收者。224.0.0.0～239.255.255.255 之间的地址称为 D 类地址，专门用于支持 IP 多播。每

个多播地址都被看作一个组，当需要从某个地址获取信息时，加入该组即可。例如，可能为股票报价系统设置了地址 225.35.23.38 作为多播地址，如果想接收股票报价信息，就必须加入到该组中。IP 多播是基于 UDP 的，java.net 类库中的 MulticastSocket 类提供了对多播的支持，允许使用多播 IP 发送或接收 UDP 数据报。

1

（1）public MulticastSocket() throws IOException;

创建一个多播 Socket。

（2）pubic MulticastSocket(int port) throws IOException;

在指定端口创建一个多播 Socket。

2

（1）public void setTimeToLive(int tt) throws IOException;

（2）public int getTimeToLive() throws IOException;

（3）public void joinGroup(InetAddress mcastaddr) throws IOException;

（4）public void leaveGroup(InetAddress mcastaddr) throws IOException;

（5）public synchronized void send(DatagramPacket p, byte ttl) throws IOExcetpion;

（6）public synchronized void receive(DatagramPacket p, byte ttl) throws IOExcetpion;

（7）public void setIneterface(InetAddress inf) throws SocketExcetpion;

（8）public InetAddress getIneterface() throws SocketExcetpion;

3 MulticastSocket

创建 MulticastSocket 对象之后，必须创建一个相应格式的 DatagramPacket 对象，这样，数据报就可以使用 send（）发送了。例 12.9 中的代码演示了如何发送和接收 IP 多播数据报。

【例 12.9】

```
Int multiPort = 1111;    / / 端口号
Int ttl = 1;             / / TTL 值
InetAddress multiAddr = InetAddress.getByName("224.1.1.1"); / / 设置多播地址
byte[] multiBytes = {"n","e","t","l","a",b"};
/ / 创建多播数据报
DatagramPacket SmultiDatagram(multiBytes,multiBytes.length, multiAddr, multiPort);
MulticastSocket multiSocket = new MulticastSocket(); / / 创建多播 Socket
MultiSocket.send(SmultiDatagram, ttl); / / 发送数据报（不加入到组中）
……
Byte[] multiBytes = newbyte[256];    / / 定义一个长为 256 的空数据报
/ / 创建接收数据报
DatagramPacket RmultiDatagram = new DatagramPacket(multiAddr,multiPort);
multiSocket.joinGroup(multiAddr);    / / 加入到多播组中
multiSocket.receive(RmultiDatagram);    / / 接收 UDP 数据报
……
multiSocket.leaveGroup(multiAddr);    / / 离开多播组
multiSocket.close();                  / / 关闭多播 socket
```

第13章

J2EE

Java 在企业中的大量应用导致 J2EE 规范的诞生，EJB、JSP 和 Servlet 的配合能满足绝大多数企业应用的需要。

本章的主要内容包括:

- J2EE 的产生
- J2EE 体系结构
- J2EE 组件
- J2EE 平台服务
- J2EE 容器

13.1 J2EE 的产生

1996 年，Sun 公司推出 Java，当时它的主要用途是制作产生动态网页的 Applet。后来，人们发现 Java 的“一次开发，多次运行”、纯面向对象的特性、垃圾回收机制和内置的安全性特别适合于开发企业应用系统。于是，企业应用开发商纷纷在 Java 标准版的基础上各自扩展出许多企业应用 API，其结果导致基于 Java 的企业应用呈爆炸式增长。但是各企业系统 API 之间又不能相互兼容，从而破坏了 Java 的平台独立性。鉴于此，Sun 公司联合 IBM、Oracle、BEA 等大型企业应用系统开发商于 1998 年共同制订了一个基于 Java 组件技术的企业应用系统开发规范，该规范定义了一个多层企业信息系统的标准平台，旨在简化和规范企业应用系统的开发和部署。这一规范和其定义的平台就构成了 J2EE。

J2EE 全称是“Java2 平台企业版（Java 2 Platform Enterprise Edition）”，利用 J2EE 技术，公司或企业就能够建立完全符合自己商业逻辑的服务架构，从而以相应最短的时间、最低的投入，为最大范围的客户、雇员及供货商提供最方便的服务，而且这些服务是高质量、安全和可扩展的。

在 Sun 公司的 J2EE 主页 http://java.sun.com/j2ee/内可以找到 J2EE 的最新版本。目前，J2EE 的最新版本是 J2EE 1.3。需要注意的是，J2EE 本身是一个标准，而不是一个现成的产品，J2EE 由以下几个部分组成。

① J2EE 规范。该规范定义了 J2EE 平台的体系结构、平台角色及 J2EE 中每种服务和核心 API 的实现要求。它是 J2EE 应用服务器（Application Server）开发商的大纲。

② J2EE 兼容性测试站点。Sun 公司提供的一个测试 J2EE 应用服务器是否符合 J2EE 规范的站点，对通过该站点测试的产品，Sun 公司将发放兼容性证书。

③ J2EE 参考实现。即 J2EE SDK，它既是 Sun 公司自己对 J2EE 规范的一个非商业性实现，又是为开发基于 J2EE 企业级应用系统原型提供的一个免费的底层开发环境。

④ J2EE 实施指南。即 BluePrints 文档，该文档通过实例来指导开发人员如何去开发一个基于 J2EE 的多层企业应用系统。

根据 J2EE 标准开发的应用服务器拥有基于 Web 的应用系统需求的特点。

① 三层结构体系：最适合 Internet 环境，可以使系统有很强的可扩展性和可管理性。

② 分布式环境：可以保证系统的稳定性，同时拥有较高的性能。

③ 面向对象的模块化组件设计：可以提高开发速度，降低开发成本。

④ 采用 Java 技术：完全跨平台，适应 Internet 需要，并能得到大多数厂商支持，保护用户投资。

13.2 J2EE 体系结构

J2EE 定义了 4 个独立的层，分别是客户层、表示逻辑层、业务逻辑层和企业信息系统层，如图 13.1 所示。

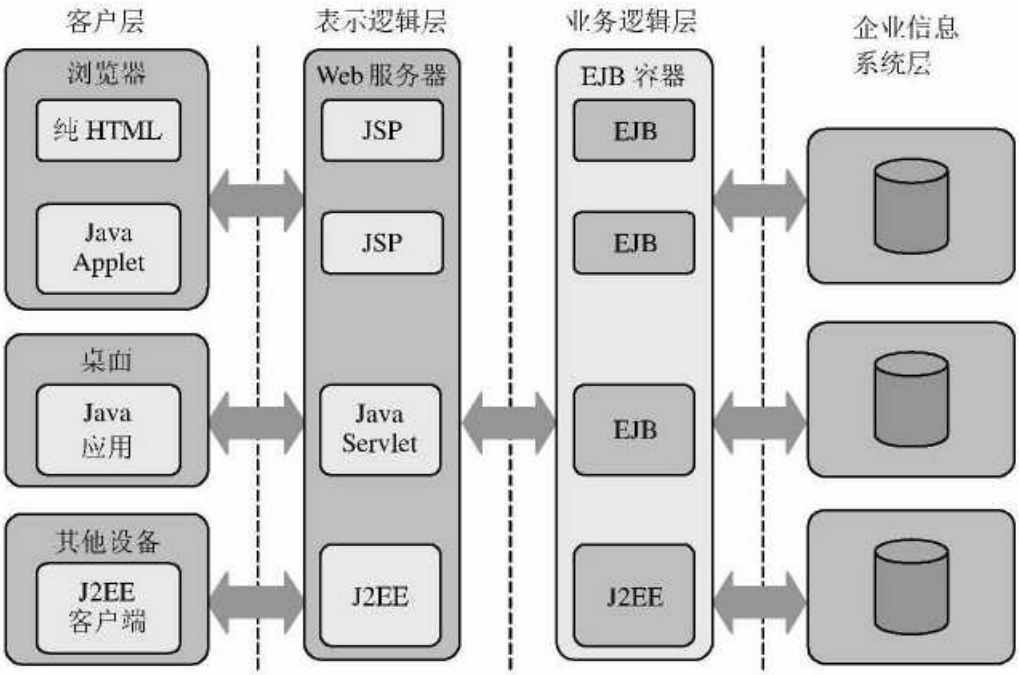


图 13.1　J2EE 体系结构

其中表示逻辑层和业务逻辑层同位于应用服务器区域，所谓应用服务器区域其实就是J2EE 平台的具体实施场所。J2EE 四层中的每一层都可被物理地部署在不同的场地中，而且，即使同在应用服务器区域内，表示逻辑层和业务逻辑层也可以分开来，分别被安装在不同的服务器上。

例如，我们可以将 Tomcat 作为 HTTP Web 服务器和表示逻辑容器，而在 Weblogic（BEA 公司的 J2EE 应用服务器产品）上部署业务逻辑组件。

关于层和容器两者的关系和区别，通常是将表示逻辑层和 Servlet 以及 JSP 容器相关联，而将业务逻辑层和企业 Javabean 容器相关联。或者，为了更易于理解，可以把层看作是概念实体——其功用是为了方便设计——而把容器看作物理意义上的软件实例也就是说，容器是为应用组件提供的运行时环境。

J2EE 多层体系结构的灵感来自模型—视图—控制器（Model-View-Controller，MVC）结构。MVC 模式是软件设计的典型结构。在这种设计结构下，一个复杂应用被分解为模型、视图和控制器 3 部分，分别对应于业务逻辑和数据、用户界面、用户请求处理和数据同步，这 3 个部分各自负责相应的功能。

如果开发一个企业级应用，只需要一种客户端的话，那么一切都非常容易解决。但真实情况是，人们必须面对运行在各种设备上的客户端（像 PDA、WAP 浏览器以及运行在桌面上的浏览器），这就不得不开发不同的应用程序来处理来自不同客户端的请求。数据访问与现实将混淆在一起，可能会出现重复的数据访问，从而导致整个开发周期没有必要的延长。

MVC 设计理念认为，在一个应用系统中，用户界面发生变动的可能性最大，控制部分变动次之，而业务逻辑是最稳定的。因此，为业务逻辑编写的代码不应该和反映用户界面的代码混杂在一起，而是彼此应该尽可能地独立，由控制器来担当两者交互的中介。MVC 认为，核心业务过程应该完全不依赖于特定的客户端程序，除了浏览器外，该任务也应该

能够被其他应用或银行办公系统调用。业务逻辑和视图元素之间的数据交互分配给控制器来完成。

基于 MVC 设计方法，J2EE 体系结构很自然地将业务逻辑层和表示逻辑层分开，控制器可置于其中任意一层，也可分置两层中。通过这种方式，J2EE 的业务逻辑组件获得相当高的可重用性。图 13.2 是 MVC 的模式结构。

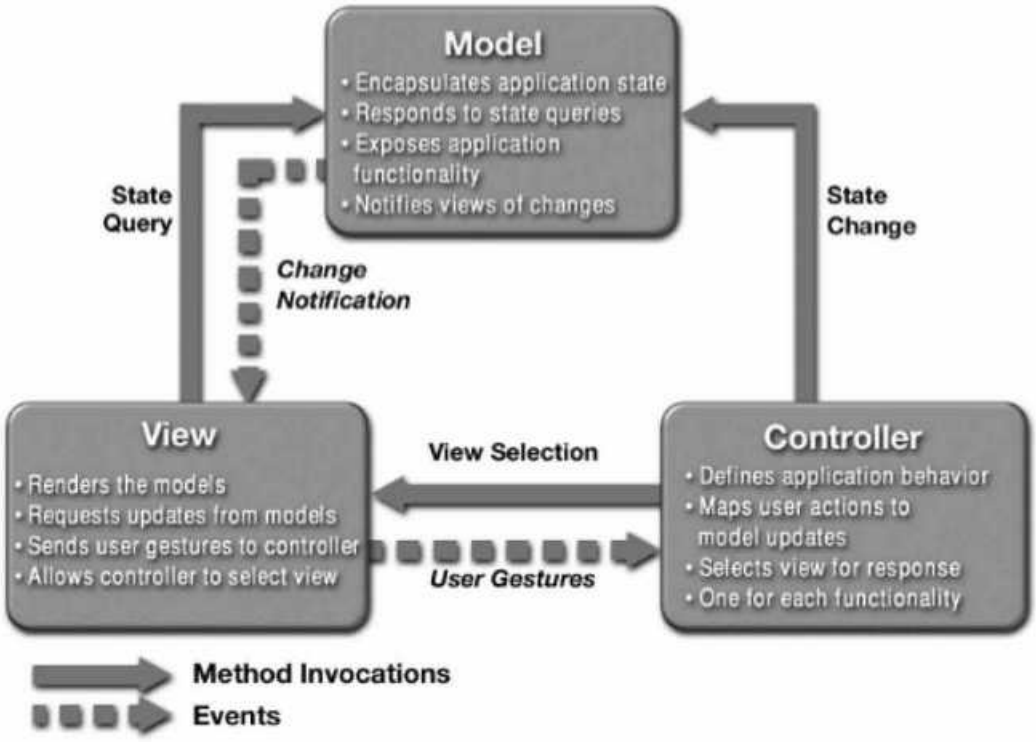


图 13.2 MVC 模式结构

13.3 J2EE 组件

在 J2EE 中，J2EE 的客户层、表示层和业务逻辑层各自有其适用的应用组件。

① 客户层的应用组件有：Applet、标准 Java 类和 Jar 文件。

② 表示逻辑层的应用组件有：Web 组件、标准 Java 类和 Jar 文件。

③ 业务逻辑层的应用组件有：EJB（Enterprise Javabean）、标准 Java 类和 Jar 文件。

J2EE 通过 Java 插件为 Applet 提供运行环境，客户端的组件容器就是本地 Java 虚拟机。Web 及 EJB 组件在服务器端运行。J2EE 中包含两种 Web 组件：JSP 和 Servlet。它们是 Web 服务器的功能扩展，都能生成动态 Web 页面。所不同的是，JSP 是将 Java 代码嵌入到 HTML 中，服务器负责解释执行，生成结果返回用户（与 ASP 技术相似）。而 Servlet 是单独的 Java 类，它动态生成 HTML 文件返回给客户。Web 组件的容器比较典型的就是基于 Java 的 Web 服务器。

下面将详细讲解 JSP，Servlet 和 EJB 这 3 个新概念。

13.3.1 EJB

1 EJB

EJB 是 J2EE 平台的核心，也是 J2EE 得到业界广泛关注和支持的主要原因。我们知道，J2EE 的一个主要目的就是简化企业应用系统的开发，使程序员将主要精力放在商业逻辑的开发上。EJB 正是基于这种思想的服务器端技术，它本身也是一种规范，该规范定义了一个可重用的组件框架来实现分布式的、面向对象的商业逻辑。EJB 的核心思想是将商业逻辑与底层的系统逻辑分开，使开发者只需关心商业逻辑，而由 EJB 容器实现目录服务、事务处理、持久性、安全性等底层系统逻辑。

EJB 技术定义了一组可重用的组件，即 Enterprise Bean 组件。利用这些组件可以建立分布式应用程序。当代码写好之后，这些组件就被组合到特定的文件中去。每个文件有一个或多个 Enterprise Bean，再加上一些配置参数。最后，这些 Enterprise Bean 被配置到一个装了 EJB 容器的平台上。客户能够通过这些 Bean 的 Home 接口，定位到某个 Bean，并产生这个 Bean 的一个实例。这样，客户就能够在程序中调用 Beans 的应用方法和远程接口了。

EJB 服务器作为容器和低层平台的桥梁管理着 EJB 容器和函数。它向 EJB 容器提供了访问系统服务的能力。例如：数据库的管理和事务的管理，或者对其他的 Enterprise 应用服务器的管理。

所有的 EJB 实例都运行在 EJB 容器中。容器提供了系统级的服务，控制了 EJB 的生命周期。

2 EJB

EJB 技术定义了 3 种 Bean，分别是 Session Bean、Entity Bean 和 Message Driven Bean。

（1）Session Bean

Session Bean 是一种作为单个的 Client 执行的对象。当远程的有一个任务请求时，容器便产生一个 Session Bean 的实例回应。一个 Session Bean 和一个 Client 对应。从某种程度上来说，一个 Session Bean 对于服务器来说就代表了它的那个 Client。Session Bean 也能用于事务，它能够更新共享的数据，但它不直接描绘这些共享的数据。

Session Bean 的生命周期是相对较短的。只有当 Client 保持会话的时候，Session Bean 才是激活的。一旦 Client 退出了，Session Bean 就不再与 Client 相联系了。Session Bean 被看成是瞬时的，因为如果容器崩溃了，那么 Client 必须重新建立一个新的 Session 对象来继续会话。

一个 Session Bean 显性地声明了与 Client 的互操作或者会话。也就是说，Session Bean 在客户会话期间，通过方法的调用，掌握 Client 的信息。一个具有状态的 Session Bean 称为有状态的 Session Bean。当 Client 终止与 Session Bean 互操作的时候，会话便终止了，而且 Bean 也不再拥有状态值。

一个 Session Bean 也可能是一个无状态的 Session Bean。无状态的 Session Bean 并不掌握它的客户的信息或者状态。Client 能够调用 Bean 的方法来完成一些操作。但是，Bean 只是在方法调用的时候才知道 Client 的参数变量。当方法调用完成以后，Bean 并不继续保持这些参数变量。这样，所有的无状态的 Session Bean 的实例都是相同的，除非它正在方法调用期间。因此，无状态的 Session Bean 就能够支持多个 Client。容器能够声明一个无状态的 Session

Bean，并能够将任何 Session Bean 指定给任何 Client。

（2）Entity Bean

Entity Bean 对数据库中的数据提供了一种对象的视图。例如：一个 Entity Bean 能够模拟数据库表中一行相关的数据。多个 Client 能够共享访问同一个 Entity Bean。多个 Client 也能够同时访问同一个 Entity Bean。Entity Bean 通过事务的上下文来访问或更新数据库中的数据。这样，数据的完整性就能够被保证。

Entity Bean 能存活相对较长的时间，并且状态是持续的。只要数据库中的数据存在，Entity Bean 就一直存活，而不是按照应用程序或者服务进程来说的。即使 EJB 容器崩溃了，Entity Bean 也是存活的。Entity Bean 生命周期能够被容器或者 Bean 自己管理。如果由 Bean 自己管理，就必须写 Entity Bean 的代码，包括访问数据库的调用。

Entity Bean 是由主键（primary key 数据库中的惟一对象标识符）标识的。通常，主键标识数据库中的一块数据，例如一个表中的一行。主键使 Client 能够定位特定的数据块。

（3）Message Driven Bean

在 EJB2.0 中，对规范的一个基础性更改是添加了一种全新的企业级 Bean 类型，即 Message Driven Bean。Message Driven Bean 专门设计用来处理入网的 JMS 消息。

JMS 是一种与厂商无关的 API，用来访问消息收发系统。它类似于 JDBC(Java Database Connectivity)，这里，JDBC 是可以用来访问许多不同关系数据库的 API，而 JMS 则提供同样与厂商无关的访问方法，以访问消息收发服务。许多厂商目前都支持 JMS，例如 IBM 的 MQSeries、BEA 的 Weblogic JMS service 和 Progress 的 SonicMQ。

EJB2.0 以两种方式支持 JMS 的集成，一种方式是将 JMS 作为一种 Bean 可用的资源，另一种方式是将 JMS 作为一个 Message Driven Bean。当将 JMS 用作一种资源时，使用 JMS API 的 Bean 就是消息的产生者或发送者。在这种情况下，Bean 将消息发送给称为主题或队列的虚拟通道。在第二种方式中，Message Driven Bean 是消息的使用者或接收者，它监听特定的虚拟通道（主题或队列），并处理发送给该通道的消息。为了更好地理解消息产生者和消息使用者的作用，可以用 Session Bean 发送一条使用 JMS 的消息，然后用一个新的 Message Driven Bean 来使用同一条消息。

3 EJB

（1）可部署的 EJB 组件

一个可部署的 EJB 组件包含下述 4 个部分。

① Remote 接口

Remote 接口定义 EJB 组件中提供的可供用户调用的方法，即通常所说的实现商业逻辑的函数或过程（如计算商品价格的函数），以供远程客户端调用。在 EJB 组件部署到容器的时候，容器会自动生成 Remote 接口相应的实例，即 EJB 对象，该对象负责代理用户的调用请求。

② Home 接口

Home 接口定义一组方法来创建新的 EJB 对象，用来查找、定位和清除已有的 EJB 对象。在 EJB 组件部署时容器，也会自动生成相应的 Home 对象，该对象负责查找和创建 EJB 对象，返回 EJB 对象的引用给客户；用户利用该引用调用 EJB 组件的方法，得到结果；最后 Home 对象清除 EJB 对象。我们可以形象地称 Home 接口为 EJB 对象的工厂。

③ Enterprise Bean 实现类

Enterprise Bean 类是商业逻辑的具体实现类，其可供用户调用的方法在 Remote 接口中定义。EJB 的实现类各接口要从不同的基类中继承下来。一个会话 Bean 必须实现基类 javax.ejb.SessionBean，而实体 Bean 必须实现基类 javax.ejb.EntiyBean。这些 EJB 的基类都是从 javax.ejb.EnterpriseBean 继承而来，而 javax.ejb.EnterpriseBean 又是从 java.io.Serializable 继承而来的。每一个 Enterprise Bean 都必须有一个 Remote 接口，Remote 接口定义了客户可以调用的逻辑操作，这些逻辑操作是可以由客户调用的公共的方法，通常由 Enterprise Beans 类来实现。

Enterprise bean	Bean	Remote

Enterprise Bean 类的 Remote 接口扩展了 javax.ejb.EJBObject 类的公共 Java 接口，而 Javax.ejb.EJBObject 是所有 Remote 接口的基类。其代码如下：

```
package javax.ejb;
public interface EJBObject extends java.rmi.Remote
{
public EJBHome getEJBHome() throws java.rmi.RemoteException;
public Object getPrimaryKey() throws java.rmi.RemoteException;
public void Remove() throws java.rmi.RemtoeException, java.rmi.RemoveException
public Handle getHandle() throws java.rmi.RemoteException;
boolean isIdentical (EJBObject p0) throws java.rmi.RemoteException;
}
```

getEJBHome()方法允许取得一个相关的 Home 接口。对于实体 Bean，用 getPrimaryKey()方法获得实体 Bean 的主键值。Remove()可以删除一个 Enterprise Bean。具体的语义在各种不同类型的 Enterprise Beans 中，由上下文来解释。GetHandle()方法返回了一个 Enterprise Bean 实例的持久的句柄。IsIndentical()方法允许比较 Enterprise Beans 是否相同。

④ 配置文件

部署编译好的 EJB 还需要相应的配置文件，是 XML 格式的，其中包含该 EJB 组件的配置信息，声明了 Enterprise Bean 的属性、绑定了 Bean 的 Class 文件（包括 stub 文件和 skeleton 文件）。

（2）EJB 的开发过程

下面我们通过一个实体 bean 的例子来说明 EJB 开发过程。一般需要下面几步。

① 编写 **Remote** 接口

```
import javax.ejb.EJBObject;
import java.rmi.RemoteException;
public interface EJBSample extends EJBObject {
public void method1() throws RemoteException;
```

```
public int method2() throws RemoteException;
}
```

注意：继承的是 EJBObject。生成 EJBSample.java。

② 编写 **Home** 接口

```
import javax.io.Serializable;
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;
public interface EJBSampleHome extends EJBHome {
Converter create() throws RemoteExceptoin, CreateException;
}
```

注意：继承的是 EJBHome。生成 EJBSampleHome.java。

③ 编写 **Enterprise Bean CLASS**

```
import java.rmi.RemoteException;
import javax.ejb. EntityBean;
import javax.ejb. EntityContext;
public class EJBSampleBean implements EntityBean{
public void method1() {
…
}
public int method2() {
…
}
}
```

生成 EJBSampleBean.java。

④ 生成主键文件

若是 Entity Bean，则还需要指定主键类型。如果是已知类型，例如 String，则在配置文件中指定；主键也可以是实体对象，可以包含大量的属性，此时还要生成 EJBSamplePK.java，并在配置文件中指定类 EJBSamplePK 为主键类型。

⑤ 生成 **xml** 配置文件

这里包名为 Sample1，dbname 为该实体 Bean 相关的 JDBC 数据库。

```
<?xml version="1.0"?>

<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 1.1//EN
http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd'>

<ejb-jar>
    <enterprise-beans>
      <entity>
```

```
<ejb-name> Sample1</ejb-name>
<home> Sample1.EJBSampleHome</home>
<remote> Sample1.EJBSample</remote>
<ejb-class> Sample1.EJBSampleBean</ejb-class>
<persistence-type>Bean</persistence-type>
<prim-key-class> java.lang.String</prim-key-class>
<reentrant>False</reentrant>

    <resource-ref>
     <res-ref-name>jdbc/dbname</res-ref-name>
     <res-type>javax.sql.DataSource</res-type>
     <res-auth>Container</res-auth>
    </resource-ref>

  </entity>
</enterprise-beans>
<assembly-descriptor>
  <container-transaction>
<method>
   <ejb-name> Sample1</ejb-name>
   <method-intf>Remote</method-intf>
   <method-name>*</method-name>
</method>
<trans-attribute>NotSupported</trans-attribute>
  </container-transaction>
</assembly-descriptor>
</ejb-jar>
```

生成以上文件并编译好，打成 JAR 包后，便可以发布到 J2EE 应用服务器的应用中了。

13.3.2 JSP

1 JSP

JSP 的全称是 Java Server Pages，是 Sun 公司于 1999 年 6 月在 Java 的基础上开发出的。它是一种以 Java 语言为主的跨平台 Web 开发语言，实现了动态页面与静态页面的分离，脱离了硬件平台的束缚，以其编译后运行的方式大大提高了其执行效率而逐渐成为因特网上的主流开发工具。

JSP 与微软的 Active Server Pages(ASP) 兼容，但是与 ASP 不同的是，它使用的是类似于 HTML 的 tags 以及 Java 程序代码段，而不是 VBScript 或 JavaScript。另外，ASP 有微软产品的通病（即只能在 Windows 平台下使用），虽然它可以通过增加控件而在 UNIX 平台下使用，但是其功能最强大的 DCOM 控件却不能使用。在这一方面，JSP 就比 ASP 更受广大的程序

员欢迎。

JSP 与 JavaScript 不同，当写好 JSP 代码后，第一次访问这个 JSP 网页时，网站服务器会自动将以 JSP 写成的 Java 程序代码段转换成 Java Servlets。而 Servlets 的字节代码只有在客户请求时才执行，所以，首次调用 JSP 时会有几秒钟的加载时间，但后续的请求会非常迅速，因为服务器已经缓存了运行的 servlets。当前的 JSP 服务器都带有 Java 即时编译器（JIT），因此，JSP 的执行速度比每次都要解释执行的 ASP 代码的执行速度要快，尤其是在代码中存在循环操作时，JSP 要比 ASP 快 1～2 个数量级。

2 JSP

由于 JSP 使用的字符集是 unicode，所以只要是 unicode 中序号大于 0xC0(十六进制)的所有符号，都可以组成合法的标识符，但是，标识符必须是以字母、下划线、与“$”为首字母。还有，标识符不能与 JSP 的保留字相同，JSP 的保留字如表 13.1 所示。

表 13.1 **JSP 保留字**

Abstract	Break	Byte	Boolean	Catch
Case	class	Char	continue	default
Double	Do	Else	extends	false
Final	Float	For	finally	If
Import	implements	Int	Interface	instanceof
Long	length	Native	New	null
Package	Private	Protected	public	return
Switch	Synchronized	Short	Static	Super
Try	true	This	Throw	Throws
Threadsafe	transient	Void	While	

3

（1）HTML 注释

在客户端显示一个注释，这种注释和 HTML 中很像，也就是它可以在“查看源代码”中看到。惟一不同的就是，在这个注释中可以用 Java 表达式（例 2 所示）。这个表达式是不确定的，根据页面不同而不同，各种表达式都能够使用，只要是合法的就行。

JSP 语法：

< ! — comment [< %= expression %>] —>

【例 13.1】

< ! -- This file displays the user login screen —>

在客户端的 HTML 源代码中产生和上面一样的数据，即：

< ! -- This file displays the user login screen —>

【例 13.2】

< !-- This page was loaded on < %= (new java.util.Date()).toLocaleString() %> —>

在客户端的 HTML 源代码中显示为：

<! --This page was loaded on August 30, 2002 -->

（2）隐藏注释

隐藏注释写在JSP程序中，但不是发给客户。用隐藏注释标记的字符会在JSP编译时被忽略掉。这个注释在希望隐藏或注释JSP程序时是很有用的。JSP编译器是不会对<%--and--%>之间的语句进行编译的，它不会显示在客户的浏览器中，也不会在源代码中看到。在< %— --%>之间，可以任意写注释语句，但是不能使用"--%>"；如果非要使用，请用"--%\>"。

JSP语法：

< %-- comment —%>

（3）声明

在JSP程序中声明合法的变量和方法。可以一次性声明多个变量和方法，只要以“;”结尾就行，当然这些声明在Java中应该是合法的。当声明方法或变量时，请注意以下的一些规则：声明必须以“;”结尾；可以直接使用在< % @ page %>中被包含进来的已经声明的变量和方法，不需要对它们重新声明；一个声明仅在一个页面中有效。如果希望每个页面都用到一些声明，最好把它们写成一个单独的文件，然后用< %@ include %>或< jsp:include >元素包含进来。

JSP语法：

< %! declaration; [declaration;]+ ... %>

【例13.3】

< %! int i = 0; %>

< %! int a, b, c; %>

（4）表达式

包含一个符合JSP语法的表达式，表达式元素表示的是一个在脚本语言中被定义的表达式，在运行后被自动转化为字符串，然后插入到这个表达示在JSP文件的位置显示。因为这个表达式的值已经被转化为字符串，所以能在一行文本中插入这个表达式（形式和ASP完全一样）。当在JSP中使用表达式时，请记住以下几点：

① 不能用一个分号（“;”）来作为表达式的结束符。但是同样的表达式用在scriptlet中就需要以分号来结尾了；

② 这个表达式元素能够包括任何在Java Language Specification中有效的表达式；

③ 有时候表达式也能做为其他JSP元素的属性值。

一个表达式能够变得很复杂，它可能由一个或多个表达式组成，这些表达式的顺序是从左到右。

JSP语法：

< %= expression %>

【例13.4】

<a href="<%=baseUrl%>/sample.jsp?<%=paraStr%>&pageNum=<%=pageNum - 1%>">示例

（5）Scriptlet

JSP语法：

< % code fragment %>

Java scriptlet JSP
JSP scriptlet
JSP
< jsp:useBean> JSP Java
Java Language Specification
HTML JSP scriptlet JSP
scriptlet scriptlet
out

【例 13.5】

```
<%
    boolean check = serPerson.checkLogin(request);
    String baseUrl = common.getSessionValue(request,"baseUrl");
    if(check == false)
    {
%>
<script language=javascript>
goToUrl("<%=baseUrl%>/login.jsp","top");
</script>
<%
    }
    else
    {
%>
<script language=javascript>
goToUrl("<%=baseUrl%>/false.jsp","top");
</script>
<%
    }
%>
```

（6）JSP 指令

JSP 指令都是这样的格式<%@.....%>，有下述几个指令。

① Include 指令

在 JSP 中包含一个静态的文件，同时解析这个文件中的 JSP 语句。静态的包含就是指这个被包含的文件将会被插入到 JSP 文件中去，这个包含的文件可以是 JSP 文件、HTML 文件以及文本文件。如果包含的是 JSP 文件，则这个包含的 JSP 的文件中的代码将会被执行。但是，在这个包含文件中不能使用< html>、< /html>、< body>以及< /body>标记，因为这将会影响在原 JSP 文件中同样的标记 ，这样做有时会导致错误。

JSP 语法：

< %@ include file="relativeURL" %>

【例 13.6】

include.jsp:

```
< html>
< head>< title>An Include Test< /title>< /head>
< body bgcolor="white">
< font color="blue">
The current date and time are
< %@ include file="date.jsp" %>
< /font>
< /body>
< /html>
```

date.jsp:

```
< %@ page import="java.util.*" %>
< %= (new java.util.Date() ).toLocaleString() %>
```

include.jsp 的显示：

```
Displays in the page:
The current date and time are
August 30, 2002 12:38:40
```

② Page 指令

定义 JSP 文件中的全局属性。作用于整个 JSP 页面，同样包括静态的包含文件。但是，< % @ page %>指令不能作用于动态的包含文件，例如 < jsp：include>包含的文件。在一个页面中可以使用多个<%@ page%>指令，但是其中的属性只能用一次，不过也有个例外，那就是 import 属性，因为 import 属性和 Java 中的 import 语句差不多(参照第一章)，所以就能多用该属性几次了。无论把<%@page%>指令放在 JSP 的文件的哪个地方，它的作用范围都是整个 JSP 页面，但为了 JSP 程序的可读性以及好的编程习惯，最好还是把它放在 JSP 文件的顶部。

JSP 语法：

```
< %@ page
 [ language="java" ]
 [ extends="package.class" ]
 [ import="{package.class | package.*}, ..." ]
 [ session="true | false" ]
 [ buffer="none | 8KB| sizeKB" ]
 [ autoFlush="true | false" ]
 [ isThreadSafe="true | false" ]
```

```
[ info="text" ]
[ errorPage="relativeURL" ]
[ contentType="mimeType [ ;charset=characterSet ]" | "text/html ; charset=ISO-8859-1" ]
[ isErrorPage="true | false" ]
%>
```

【例 13.7】

```
< %@ page import="java.util.*, java.lang.*" %>
< %@ page errorPage="error.jsp" %>
```

属性说明如下所述。

- **language**="java"声明脚本语言的种类，暂时只能用"java"。
- **extends**="package.class"标明 JSP 编译时需要加入的 Java Class 的全名，但是要慎重使用，因为它会限制 JSP 的编译能力。
- **import**="{package.class | package.* }, ..." 需要导入的 Java 包的列表，这些包就作用于程序段、表达式以及声明。

下面的包在 JSP 编译时已经导入了，所以就不需要在页面中再用 import 显式地导入 java.lang.*、javax.servlet.*、javax.servlet.jsp.*和 javax.servlet.http.*了。

- **session**="true | false"设置客户是否需要 HTTP Session。如果为 true，则 Session 是有用的。如果为 false，则不能使用 session 对象，也不能使用定义了 scope=session 的< jsp:useBean>元素。这样的使用会导致错误。缺省值是 true。
- **buffer**="none | 8KB | sizeKB"设置 buffer 的大小，buffer 被 out 对象用于处理执行后的 JSP 对客户浏览器的输出。缺省值是 8KB。
- **autoFlush**="true | false" 设置如果 buffer 溢出，是否需要强制输出。如果其值被定义为 true(缺省值)，则输出正常，如果被设置为 false，则 buffer 溢出，从而导致一个意外错误的发生。另外，如果把上面的参数 buffer 设置为 none，则不能把 autoFlush 设置为 false。
- **isThreadSafe**="true | false" 设置 JSP 文件是否能多线程使用。缺省值是 true，也就是说，JSP 能够同时处理多个用户的请求；如果设置为 false，一个 JSP 只能一次处理一个请求。
- **info**="text" 一个文本，一般用来对这个页面做一些说明，在执行 JSP 时，将会被逐字加入到 JSP 中，并能够在程序中使用 Servlet.getServletInfo 方法取回。
- **errorPage**="relativeURL" 设置处理异常事件的 JSP 文件。
- **isErrorPage**="true | false" 设置此页是否为出错页，如果被设置为 true，就能使用 exception 对象。
- **contentType**="mimeType [;charset=characterSet]" | "text/html;charset=ISO-8859-1" 设置 mime 类型。缺省 mime 类型是: text/html， 缺省字符集为 ISO-8859-1。

③ **Taglib** 指令

定义一个标签库并自定义标签的前缀。这里自定义的标签有标签和元素之分。由于 JSP 文件能够转化为 XML，所以了解标签和元素之间的联系很重要。标签只不过是一个在意义上被抬高了点的标记，是 JSP 元素的一部分。JSP 元素是 JSP 语法的一部分，和 XML 一样有开始标记和结束标记。元素也可以包含其他的文本、标记和元素。例如，一个 jsp:plugin 元素有< jsp:plugin>开始标记和< /jsp:plugin>结束标记，同样也可以有< jsp:params>和< jsp:fallback>

元素。必须在使用自定义标签之前使用<%@ taglib %>指令，而且可以在一个页面中多次使用，但是前缀只能使用一次。

JSP 语法：

```
<%@ taglib uri="URIToTagLibrary" prefix="tagPrefix" %>
```

属性说明如下所述。

- **uri**="URIToTagLibrary" Uniform Resource Identifier (URI)根据标签的前缀对自定义的标签进行惟一的命名，URI 可以是 Uniform Resource Locator (URL)或 Uniform Resource Name (URN)。
- **prefix**="tagPrefix" 在自定义标签之前的前缀，例如，有个自定义标签<public:loop>，public 为该标签的前缀，如果这里不声明 public，那么这个标签就是不合法的。请注意：不要用 jsp, jspx, java, javax, servlet, sun 和 sunw 作为前缀。

（7）JSP 动作

JSP 动作都是这样的格式<jsp:...>，有以下几个 JSP 动作。

① **<jsp:forward>**

重定向一个 HTML 文件、JSP 文件或者是一个程序段。<jsp:forward>标签以下的代码将不能执行。重定向能够向目标文件传送参数和值。

JSP 语法：

```
<jsp:forward page={"relativeURL" | "<%= expression %>"} /> 或
        <jsp:forward page={"relativeURL" | "<%= expression %>"} >
            <jsp:param name="parameterName"
            value="{parameterValue | <%= expression %>}" />+
        </jsp:forward>
```

属性如下所述。

- **page**="{relativeURL | <%= expression %>}"这里是一个表达式或是一个字符串用于说明将要定向的文件或 URL。这个文件可以是 JSP、程序段，或者是其他能够处理 Request 对象的文件(如 asp,cgi,php)。
- **<jsp:param** name="parameterName" **value**="{parameterValue | <%= expression %>}" />向一个动态文件发送一个或多个参数，这个文件一定是可以处理参数的动态文件。如果想传递多个参数，可以在一个 JSP 文件中使用多个<jsp:param>。name 指定参数名，Value 指定参数值。

② **<jsp:getProperty>**

获取 Bean 的属性值，用于显示在页面中。在使用<jsp:getProperty>之前，必须用<jsp:useBean>创建这个 Bean。同时应注意，不能使用<jsp:getProperty>来检索一个已经被索引了的属性；也不能与 Enterprise Bean 一起使用<jsp:getProperty>，但是能够和 JavaBeans 组件一起使用。

JSP 语法：

```
<jsp:getProperty name="beanInstanceName" property="propertyName" />
```

属性说明如下所述。

- **name**="beanInstanceName" Bean 的名字，由<jsp:useBean>指定。

- **property**="propertyName" 所指定的 Bean 的属性名。

【例 13.8】

```
< jsp:useBean id="score" scope="page" class="student.Score" />
< h2>
Score of < jsp:getProperty name="score" property="stuname" />
< /h2>
```

③ **< jsp:include>**

包含一个静态或动态文件。这两种包含文件的结果是不同的。如果文件仅是静态文件，那么这种包含仅仅是把包含文件的内容加到 JSP 文件中去；而如果这个文件是动态的，那么这个被包含文件也会被 JSP 编译器执行。不能从文件名上判断一个文件是动态的还是静态的，例如 aspcn.asp 就有可能只是包含一些信息而已，而不需要执行。< jsp:include>能够同时处理这两种文件。如果这个包含文件是动态的，那么还可以用< jsp:param>传递参数名和参数值。

JSP 语法：

< jsp:include page="{relativeURL | < %= expression%>}" flush="true" /> 或 < jsp:include page="{relativeURL | < %= expression %>}" flush="true" >

< jsp:param name="parameterName" value="{parameterValue | < %= expression %>}" />

< /jsp:include>

属性说明如下所述。

- **page**="{relativeURL | < %= expression %>}" 参数为一相对路径，或者是代表相对路径的表达式。
- **flush**="true" 这里必须使用 flush="true"，不能使用 false 值。缺省值为 false。
- **< jsp:param** name="parameterName" value="{parameterValue | < %= expression %> }" />+ < jsp:param>了句能传递一个或多个参数给动态文件。

④ **< jsp:plugin>**

执行一个 Applet 或 Bean，有可能的话还要下载一个 Java 插件用于执行它。当 JSP 文件被编译，送往浏览器时，< jsp:plugin>元素将会根据浏览器的版本替换成< object>或者< embed>元素。注意，< object>用于 HTML 4.0 ，< embed>用于 HTML 3.2。一般来说，< jsp:plugin>元素会指定对象是 Applet 还是 Bean，同样也会指定 class 的名字和位置，另外还会指定将从哪里下载这个 Java 插件。

JSP 语法：

```
< jsp:plugin
 type="bean | applet"
 code="classFileName"
 codebase="classFileDirectoryName"
 [ name="instanceName" ]
 [ archive="URIToArchive, ..." ]
 [ align="bottom | top | middle | left | right" ]
 [ height="displayPixels" ]
 [ width="displayPixels" ]
```

```
[ hspace="leftRightPixels" ]
[ vspace="topBottomPixels" ]
[ jreversion="JREVersionNumber | 1.1" ]
[ nspluginurl="URLToPlugin" ]
[ iepluginurl="URLToPlugin" ] >
[ < jsp:params>
[ < jsp:param name="parameterName" value="{parameterValue | < %= expression %>}" /> ]
< /jsp:params> ]
[ < jsp:fallback> text message for user < /jsp:fallback> ]
< /jsp:plugin>
```

属性如下所述。

- **type**="bean | applet" 将被执行的插件对象的类型，必须要指定这个是 Bean 还是 Applet，因为这个属性没有缺省值。
- **code**="classFileName" 将会被 Java 插件执行的 Java Class 的名字，必须以.class 结尾。这个文件必须位于 codebase 属性指定的目录中。
- **codebase**="classFileDirectoryName" 将会被执行的 Java Class 文件的目录（或者是路径），如果没有提供此属性，那么使用< jsp:plugin>的 JSP 文件的目录将会被使用。
- **name**="instanceName" 这个 Bean 或 Applet 实例的名字，它将会在 JSP 其他的地方调用。
- **archive**="URIToArchive, ..."一些由逗号分开的路径名，这些路径名用于预装一些将要使用的类，这会提高 Applet 的性能。
- **align**="bottom | top | middle | left | right"

图形、对象或 Applet 的位置，有 bottom，top，middle，left 和 right 等值。

- **height**="displayPixels" **width**="displayPixels" Applet 或 Bean 将要显示的长宽的值，此值为数字，单位为像素。
- **hspace**="leftRightPixels" **vspace**="topBottomPixels" Applet 或 Bean 显示时在屏幕左右，上下所需留下的空间，单位为像素。
- **jreversion**="JREVersionNumber | 1.1" Applet 或 Bean 运行所需的 Java Runtime Environment (JRE) 的版本。缺省值是 1.1。
- **nspluginurl**="URLToPlugin" Netscape Navigator 用户能够使用的 JRE 的下载地址，此值为一个标准的 URL，如< A href="http://www.aspcn.com/jsp" target="_NEW">http://www.aspcn.com/jsp< /A>。
- **iepluginurl**="URLToPlugin" IE 用户能够使用的 JRE 的下载地址，此值为一个标准的 URL，如< A href="http://www.aspcn.com/jsp" target="_NEW">http://www.aspcn.com/jsp< /A>。
- **< jsp:params>** [< jsp:param name="parameterName" value="{parameterValue | < %= expression %>}" />] < /jsp:params> 需要向 Applet 或 Bean 传送的参数或参数值。
- **< jsp:fallback>** text message for user < /jsp:fallback> 一段文字，用于 Java 插件不能启动时显示给用户的，如果 JSP 页面能够启动而 Applet 或 Bean 不能，那么浏览器会将这个出错信息弹出。

⑤ **< jsp:setProperty>**

设置 Bean 中的属性值。使用 Bean 给定的 setter 方法，在 Bean 中设置一个或多个属性值。在使用这个元素之前必须使用< jsp:useBean>声明此 Bean。因为，< jsp:useBean>和< jsp:setProperty>是一一对应的，同时它们使用的 Bean 实例的名字也应当相匹配。能使用多种方法利用< jsp:setProperty>来设置属性值，例如，可以通过用户输入的所有值(被做为参数储存中 request 对象中)来匹配 Bean 中的属性，也可以通过用户输入的指定的值来匹配 Bean 中指定的属性，还可以在运行时使用一个表达式来匹配 Bean 的属性。

JSP 语法：

```
< jsp:setProperty
 name="beanInstanceName"
 {
 property= "*" |
 property="propertyName" [ param="parameterName" ] |
 property="propertyName" value="{string |< %= expression %>}"
 }
/>
```

属性说明如下所述。

- **name**="beanInstanceName" 表示已经在< jsp:useBean>中创建的 Bean 实例的名字。
- **property**="*" 使用用户请求 JSP 页面时输入的所有值来匹配 Bean 中的属性。在 Bean 中的属性的名字必须和 Request 对象中的参数名一致。从客户传到服务器上的参数值一般都是字符类型，为了使这些字符串能够在 Bean 中匹配，就必须将其转换成其他的类型，表 13.2 列出了 Bean 属性的类型及其转换方法。

表 13.2　　Bean 中的类型转换方法

Property 类型	方　法
Boolean	Java.lang.Boolean.valueOf(String)
Byte	Java.lang.Byte.valueOf(String)
Char	Java.lang.Character.valueOf(String)
Double	Java.lang.Double.valueOf(String)
Integer	Java.lang.Integer.valueOf(String)
Float	Java.lang.Float.valueOf(String)
Long	Java.lang.Long.valueOf(String)

如果 Request 对象的参数值中有空值，那么对应的 Bean 属性将不会设置任何值。同样的，如果 Bean 中有一个属性没有与之对应的 Request 参数值，那么这个属性同样也不会设定。

- **property**="propertyName" [param="parameterName"] 使用 request 中的一个参数值来指定 Bean 中的一个属性值。在这个语法中，property 指定 Bean 的属性名，param 指定 Request

中的参数名。如果 Bean 属性和 Request 参数的名字不同，那么就必须要指定 property 和 param ，如果它们同名，那么就只需要指明 property 就行了。如果参数值为空（或未初始化），则对应的 Bean 属性不被设定。

- **property**="propertyName" value="{string | < %= expression %>}" 使用指定的值来设置 Bean 属性。这个值可以是字符串，也可以是表达式。如果是字符串，那么它就会被转换成 Bean 属性的类型（查看上面的表）；如果是一个表达式，那么它的类型就必须和它将要设置的属性值的类型一致；如果参数值为空，那么对应的属性值也不会被设置。另外，不能在一个< jsp:setProperty>中同时使用 param 和 value。

⑥ **< jsp:useBean>**

创建一个 Bean 实例并指定它的名字和作用范围。< jsp:useBean>首先会试图定位一个 Bean 实例，如果这个 Bean 不存在，那么< jsp:useBean>就会从一个 class 或模板中进行示例。定位或示例一个 Bean，< jsp:useBean>将进行以下步骤。

a）通过给定名字和范围试图定位一个 Bean。

b）对这个 Bean 对象的引用变量以指定的名字命名。

c）如果发现了这个 Bean，将会在这个变量中储存这个引用。如果指定了类型，那么这个 Bean 也设置为相应的类型。

d）如果没有发现这个 Bean，将会从指定的 class 中示例，并将此引用储存到一个新的变量中去。如果这个 class 的名字代表的是一个模板，那么这个 Bean 被 java.beans.Beans.instantiate 示例。

e）如果< jsp:useBean>已经示例（不是定位）了 Bean，同时< jsp:useBean>和< /jsp:useBean>中有元素，那么将会执行其中的代码。

< jsp:useBean>元素的主体通常包含有< jsp:setProperty>元素，用于设置 Bean 的属性值。正如上面第 e 步所说的，< jsp:useBean>的主体仅仅只有在< jsp:useBean>示例 Bean 时才会被执行，如果这个 Bean 已经存在，< jsp:useBean>能够定位它，那么主体中的内容将不会起作用。

JSP 语法：

```
< jsp:useBean
 id="beanInstanceName"
 scope="page | request | session | application"
 {
 class="package.class" |
 type="package.class" |
 class="package.class" type="package.class" |
 beanName="{package.class | < %= expression %>}" type="package.class"
 }
 {
 /> |
 > other elements < /jsp:useBean>
 }
```

属性说明及用法如下所述。

- **id**="beanInstanceName" 在所定义的范围中确认 Bean 的变量，能在后面的程序中使用此变量名来分辨不同的 Bean。这个变量名对大小写敏感，必须符合所使用的脚本语言的规定，在 Java Programming Language 中，这个规定在 Java Language 规范已经写明。如果这个 Bean 已经在别的< jsp:useBean>中创建，那么这个 id 的值必须与原来的那个 id 值一致。
- **scope**="page | request | session | application" Bean 存在的范围以及 id 变量名的有效范围，缺省值是 page，以下是详细说明。

page——能在包含< jsp:useBean>元素的 JSP 文件以及此文件中的所有静态包含文件中使用 Bean，直到页面执行完毕，并向客户端发回响应或转到另一个文件为止。

request——在任何执行相同请求的 JSP 文件中使用 Bean，直到页面执行完毕，并向客户端发回响应或转到另一个文件为止。能够使用 Request 对象访问 Bean，例如，request.getAttribute(beanInstanceName)。

session——从创建 Bean 开始，就能在任何使用相同 session 的 JSP 文件中使用 Bean。这个 Bean 存在于整个 Session 生存周期内，任何在分享此 Session 的 JSP 文件都能使用同一 Bean。注意：在创建 Bean 的 JSP 文件中的< % @ page %>指令中必须指定 session=true。

application——从创建 Bean 开始，就能在任何使用相同 application 的 JSP 文件中使用 Bean。这个 Bean 存在于整个 application 生存周期内，任何在分享此 application 的 JSP 文件都能使用同一个 Bean。

- **class**="package.class"使用 new 关键字以及 class 构造器从一个 class 中示例一个 bean。这个 class 不能是抽象的，必须有一个公用的，没有参数的构造器。这个 package 的名字区别大小写。
- **type**="package.class" 如果这个 Bean 已经在指定的范围中存在，那么这个 Bean 会作为一个新的数据类型。如果没有使用 class 或 beanName 指定 type，则 Bean 将不会被示例。package 和 class 的名字区分大小写。
- **beanName**="{package.class | < %= expression %>}" type="package.class" 使用 java.beans.Beans.instantiate 方法来从一个 class 或连续模板中示例一个 Bean，同时指定 Bean 的类型。

beanName 可以是 package 和 class 也可以是表达式，其值可以和 Bean 相同。package 和 class 名字区分大小写。

【例 13.9】

```
< jsp:useBean id="cart" scope="session" class="session.Carts" />
< jsp:setProperty name="cart" property="*" />
< jsp:useBean id="checking" scope="session" class="bank.Checking" >
< jsp:setProperty name="checking" property="balance" value="0.0" />
< /jsp:useBean>
```

4 JSP

内置对象是不需要声明的，可以直接在 JSP 中使用的对象，JSP 有以下几种内置对象。

① **request**：request 表示 HttpServletRequest 对象。它包含了有关浏览器请求的信息，并且提供了几个用于获取 cookie、 header 和 session 数据的有用的方法。

② **response**：response 表示 HttpServletResponse 对象，并提供了几个用于设置送回浏览

器的响应的方法，如 cookies、头信息等。

③ **out**：out 对象是 javax.jsp.JspWriter 的一个实例，并提供了几个方法能用于向浏览器回送输出结果。

④ **pageContext**：pageContext 表示一个 javax.servlet.jsp.PageContext 对象。它是用于方便存取各种范围的名字空间、Servlet 相关的对象的 API，并且包装了通用的 Servlet 相关功能的方法。

⑤ **session**：session 表示一个请求的 javax.servlet.http.HttpSession 对象。Session 可以存储用户的状态信息。

⑥ **application**：applicaton 表示一个 javax.servle.ServletContext 对象。这有助于查找有关 Servlet 引擎和 Servlet 环境的信息。

⑦ **config**：config 表示一个 javax.servlet.ServletConfig 对象。该对象用于存取 Servlet 实例的初始化参数。

⑧ **page**：page 表示从该页面产生的一个 Servlet 实例，它能够获得以上所有的对象。

13.3.3 Servlet

1 Servlet?

一个 Servlet 是一个 Java 类，用于扩展服务器的功能，为了避免 Servlet 编程者再去考虑网络连接、获取请求、产生格式正确的响应等细节问题，人们构建了 Servlet 容器。Servlet 容器也称 Servlet 引擎，它可把任何协议传输的请求变成 Servlet 能够理解的对象，并同时给 Servlet 一个能用来发送响应的对象。它为服务端代码和基于 Web 的客户之间的通信提供了一条方便的途径。

javax.sevlet 和 javax.http.sevlet 包提供了写 Servlet 所需要的接口和类，所有的 Servlet 都必须使用 Servlet 接口，这接口采用了一种叫生命周期的方法来管理。

2 Servlet

Servlet 容器负责处理客户请求，并把请求传给 Servlet，同时将把结果返回给客户。容器与 Servlet 之间的接口是由 Servlet API 定义的，这个接口定义了 Servlet 容器在 Servlet 上调用的方法以及传递给 Servlet 的对象。

3 Servlet

Servlet 的生命周期一般如下所述。

（1）Servlet 容器创建 Servlet 的一个实例；

（2）容器调用这个实例的 init()的一个方法；

（3）如容器对该 Servlet 有请求，则调用此实例的 service()方法；

（4）容器在销毁本实例前调用它的 destroy()方法；

（5）销毁并标记该实例以作为垃圾收集。

4 Servlet API

（1）Servlet 接口

前面说过，所有的 Servlet 都必须实现 Servlet 接口，必须实现以下几个方法。

① init() 方法

```
public void init(ServletConfig config) throws ServletException;
```

一旦对 Servlet 进行实例化后，Servlet 容器就调用 init()方法。接口规定，init()方法只能被调用一次。

② service()方法

public void service(ServletRequest req, ServletResponse res) throws ServletException,IOException;

只有在成功地初始化 Servlet 以后，才能调用 service()方法来处理用户请求。Service()方法的两个参数，即 ServletRequest 和 ServletResponse 对象，第一个提供了访问初始请求的数据的方法，第二个提供了 Servlet 构造响应的方法。

③ destroy()方法

public viod destroy();

Servlet 容器可以决定中止 Servlet 服务，但是在 Servlet 调用此方法之前，必须给 service()线程足够长的时间来结束执行，所以接口规定了，如果 Service()在运行，则 destroy()方法就不会执行。

④ getServletConfig()方法；

public ServletConfig getServletConfig()；

此方法允许访问初始化参数和 ServletContext 对象。getServletConfig()方法可以让 servlet 在任何时间获得该对象及其配置信息。

⑤ getServletInfo()方法

public String getServletInfo()；

它会返回一个 String 对象，其中包含了 Servlet 的信息（如开发者、创建日期、描述信息等）。该方法也适用于 servlet 容器。

（2）GenericServlet 类

此类提供了 Servlet 接口的基本实现部分。此类的声明为：

public abstract class GenericServlet implements Servlet,ServletConfig,Serializable

注意，此类被声明为了 abstract，说明如果想要扩展这个类，就必须自己实现 service()方法。

（3）HttpServlet 类

这个类包含在 javax.servlet.http 包内，它扩展了 GenericServlet 类，并为 Servlet 接口提供了 HTTP 更相关的实现代码，它有以下几个方法。

① service()方法

此方法被 HttpServlet 类用来作为 HTTP 请求的分发器，这个方法在任何时候都不能被重载。当请求到来时，此方法将决定请求的类型，并把请求分发给相应的处理方法。

② getLastModified()方法

此方法以毫秒形式返回自 GMT 时间（1970 年 1 月 1 日 0 时 0 分 0 秒）以来最近一次修改 Servlet 的时间，缺省值是一个负数，表示时间未知。

（4）HttpServletRequest 接口

想要理解此接口，就必须对 HTTP 是如何把数据传给 Web 服务器有所了解，使用这个可以处理 HTML 的表单传过来的信息。下面就是获取表单数据的基本方法。

① getParameter()方法

此方法会试图查询串中的关键字，并根据其参数返回相应的值。如对于给定的参数有多个值，则返回第一个。

② getParameterValues()方法

如有一个参数可返回多个值，则通过这个方法可以得到它所有的值。

③ getParameterNames()方法

返回一个 Enumeration 对象，包括对应请求的所有参数名字的列表。

（5）HttpServletResponse 接口

通过此对象及方法，Servlet 可修改响应头并返回结果，下面为其几个基本方法。

① setContentType()方法

设置 HTTP 响应的 mime 类型。

② getWriter()方法

返回 PrintWriter 对象，可把 Servlet 的结果作为文本返回给调用者。

③ getOutputStream()方法

返回 ServletOutputStream 对象获得输出对象，它是 java.io.OutputStream 的一个子类。

④ setHeader 方法

用来设置送回给客户的 HTTP 响应头。

13.4 J2EE 平台服务

一般认为，现代企业计算解决方案除了企业的业务逻辑外，还需要为一些基本服务提供支持，J2EE 环境的另一大特色就在于它提供了完善的企业级服务，以满足各类应用的需要。下面是一些主要服务以及提供服务的组件。

1

JDBC（Java Database Connectivity）：提供数据库连接和访问服务。

JCA（Java Connector Architecture）：提供与旧有遗留系统之间的连接。

2

Java 消息服务 JMS（Java Messaging Service）：提供层与组件之间的消息传递。

电子邮件服务（JAF/Javamail）：提供电子邮件服务。

Java IDL/RMI-IIOP：CORBA 兼容接口，提供 J2EE 与 CORBA 的通信服务。

JAX（Java XML APIs）：提供 XML 语法分析/绑定服务。

3

JNDI（Java Naming and Directory Interface）：提供分布式命名和目录服务。

4

JTS/JTA（Java Transaction Service）：提供事务处理/监控服务。

JAAS（Java Authentication and Authorization service）：提供访问控制等安全服务。

13.5 J2EE 容器

J2EE 应用组件各自运行在相应的运行时环境中，这些运行时环境在 J2EE 术语里被称为

“容器”。不同服务器厂商推出的容器产品不尽相同，但一定都符合通用接口标准，且均为 J2EE 中间件组件提供了许多必不可少的底层公共设施。

J2EE 标准共定义了以下 4 种容器，应用组件可通过配置工具部署到对应的容器中。

① Applet 容器：运行 Applet。

② 客户端应用程序容器：运行标准客户端 Java 应用程序（包括基于 Swing 的 GUI 客户端应用程序）。

③ Web 容器：运行表示逻辑层的 Servlet 和 JSP。

④ EJB 容器：运行业务逻辑层的企业 Javabean。

J2EE

典型地，容器提供的基础设施包括：内存管理、同步/分线程、垃圾收集、可用性、可伸缩性、负载平衡和容错。

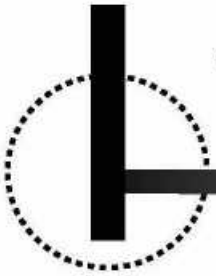
容器需要实现的基本接口和基础设施在 J2EE 规范中都有详细定义，但具体实施方案则因容器供应商的不同而不可能全然相同。因此 J2EE 在保持代码兼容性的同时，也为支持服务器的特性提供了一定的余地。

容器在业界的实现一般称为 J2EE 应用服务器，在 J2EE 应用服务器之上开发的代码，最大的特点是具有非常强的可移植性。例如，在 BEA 公司 Weblogic 服务器上开发的 Servlet 可以部署到 IBM 公司的 Websphere 服务器上，而不需要经过任何代码级修改。

为了确保在不同供应商服务器产品上开发的应用组件之间的兼容性，Sun 公司发布了 J2EE 许可证计划和 J2EE 兼容性测试包（CTS）。获得 J2EE 技术许可并通过 CTS 测试的供应商可以称其产品为“通过鉴定的 J2EE”。

目前，BEA 公司的 Weblogic 服务器和 IBM 的 Websphere 在 J2EE 应用服务器市场中占据绝对主导地位，紧随其后的主要选手是 Oracle 公司。另外，值得一提的是 Tomcat，这是个开放源码的应用服务器，在大学和研究机构应用广泛。

第14章



J2ME与手机编程

Java技术驱动的无线设备这一全世界范围的潮流导致，在几年之后，J2ME将会成为手持设备的首要开发平台。

本章的主要内容包括:

- J2ME基础知识
- J2ME配置（Configuration）
- J2ME简表（Profile）
- MIDP与手机应用程序开发

14.1 J2ME 基础知识

14.1.1 J2ME 概述

在今天，越来越多的人开始认为，经过了许多人共用一台计算机的大型主机时代，一人一台计算机的个人 PC 时代之后，下一个时代就是一人多台计算设备的普及计算时代（Ubiquitous Computing，IBM 称之为 Pervasive Computing）。从家电产品到个人随身携带或车载的电子设备，从移动电话、双向寻呼机到智能卡，从个人电脑记事本（personal organizer）到掌上电脑（palmtop），计算设备将从单一的 PC 扩展到各种功能不同、大小各异的信息处理设备。

网络是这个领域最重要的增值手段。只有当手机可以随时随地无线上网，PDA 可以和个人 PC 同步，车载 PC 可以获得网上 GPS 地图数据时，这些孤立的设备才能够发挥最大的作用。这些设备的发展趋势是成为目的特定的、资源有限的网络连接设备，并且能连接到 Internet 上。而平台无关性和网络功能正是 Java 的设计目标和能力（事实上，Java 最初的设计目标就是用于消费电子领域，但是后来在 PC 和服务器上得到了发展）。因此，为了满足这些信息设备日益扩大的需求，Sun 公司引进 J2ME（Java 2 Micro Edition）技术来扩大 Java 技术的使用范围，满足消费电子和嵌入设备的需要。

J2ME 保留了 Java“write once,run everywhere”的特性，对跨通讯产业、信息产业和消费类电子产品业的公司，还有对 Java 开发人员来说的确是个好消息。Java 技术将一大批设备（从服务器到台式机和移动设备）集中到一种语言和一种技术之下。虽然这些设备的应用不同，但 Java 技术为这些不同点起到了桥梁的作用，使原本致力于单一领域的开发人员能将其技能发挥到跨越不同设备和应用的领域。因此它一经问世就为业界看好并迅速得到各大厂商的支持。NOKIA 与 Borland 公司宣布共同推出 J2ME 平台兼容环境——JBuilder MobileSet（Nokia Edition），Samsung 公司和 Sun 公司宣布，两公司将共同努力，将 J2ME 和 MIDP 集成到 Samsung 的 CDMA 和 GSM 手机的全球系统中，Motorola 则在它的新一代手机 i85s 和 i50sx 中采用了 Java 技术，在它们的带动下众多无线服务商纷纷将目光转向 Java。

基于 Java 技术的服务能提供带有更强图形功能和高保真声音的动画游戏、聊天软件、带缩放地图的基于地点的服务、安全的移动商务以及商业支持程序等。用户可以从 Internet 上下载新的服务和应用程序来定制设备，如交互游戏、网上银行、订票系统和无线协作等应用程序（如图 14.1 所示）。向 Java 技术驱动的无线设备进行转向的这一全世界范围的潮流还在继续快步进行，因此很可能在几年之后，J2ME 将会成为手持设备的首要开发平台。

Sun 公司将 J2ME 定义为“一种以广泛的消费性产品为目标的的高度优化的 Java 运行时环境，包括寻呼机、移动电话、可视电话、数字机顶盒和汽车导航系统。”

自从 1999 年 6 月在 JavaOne Developer Conference 上声明之后，J2ME 为小型设备带来了 Java 语言的跨平台功能，允许移动无线设备共享应用程序。有了 J2ME，Sun 公司已经使 Java 平台能够适应集成了或基于小型计算设备的用户产品。据一些无线市场分析家预计，到 2004 年，J2ME 将成为无线设备的主导开发平台。全球著名的通讯设备制造商们如 Motorola、Nokia、西门子、Panasonic 纷纷宣布支持 J2ME 技术。美国著名的电信网络运营商（如 Nextel、SprintPCS、BT Cellnet 等）也热切地加入到 J2ME 的研发中来。在日本，NTT DoCoMo 推出的 I-mode 手机获得了极大

的成功，I-mode 手机部分使用了 J2ME 技术。据 Sun 公司统计，到 2001 年全球已经售出了 1500 万台支持 J2ME 技术的手机，目前有超过 200 家公司发售 J2ME 应用程序。

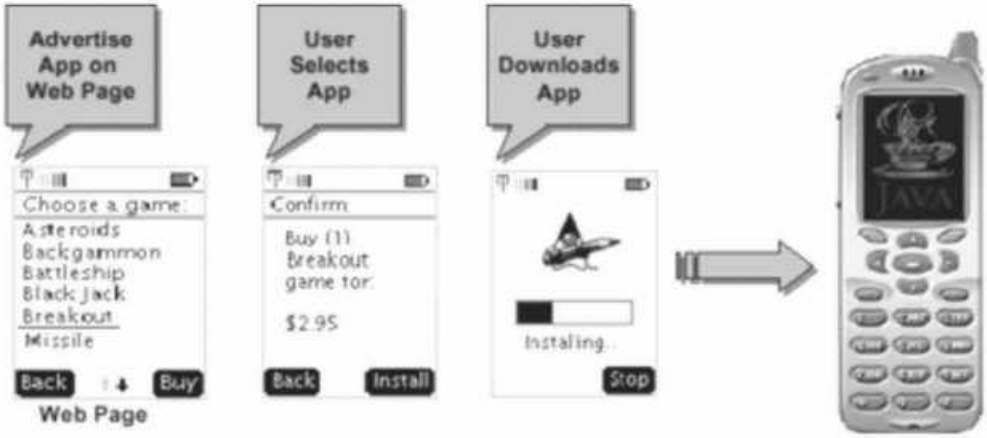


图 14.1　下载定制化的服务

14.1.2　J2ME 体系结构

首先考虑一下可能用到 J2ME 的各类设备。这类设备包括 PDA、蜂窝式电话和寻呼机、电视机机顶盒、远程遥控装置和许多其他嵌入式设备。很明显，要为所有这些设备定义一种最优化，或者接近最优化的单一技术是不可能的。它们之间处理器能源、内存、固定存储器和用户界面的差异非常之大。因此，Sun 公司在设计 J2ME 规格的时候，遵循着“对于各种不同的装置而造出一个单一的开发系统是没有意义的事”这个基本原则，将适合 J2ME 的设备的定义划分成各个部分，然后再进一步细分。

J2ME 体系结构是基于设备的系列和类别的。一个类别定义了一个特定种类的设备：移动电话、简单寻呼机和电脑记事本都是单独的类别。对存储器和处理能力有相近需求的若干类别的设备构成设备的一个系列。移动电话、简单寻呼机和简单个人电脑记事本一起就是占用资源很小的设备的一个系列。图 14.2 定义了在 J2ME 上下文环境中设备的系列和类别之间的关系。

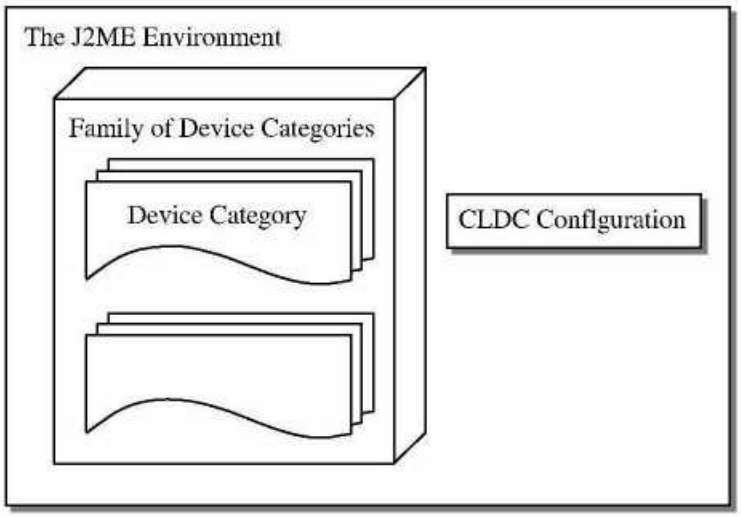


图 14.2　设备的系列（family）和类别（category）

为了支持资源受限设备系列所要求的那种灵活性和可定制部署，人们将 J2ME 体系结构设计成模块化的和可伸缩的。J2ME 分为 3 层，从下到上分别是 VM、Configuration（配置）和 Profile（简表）。VM 负责建立 Java 虚拟机，并解释 Java 代码；Configuration 负责建立核心类库，功能比较少（比如没有用户接口），主要面向水平市场；Profile 负责建立高级类库，主要功能丰富，面向垂直市场。J2ME 体系的一般结构是：由 Configuration 定义的 Java 虚拟机运行于设备的宿主操作系统之上，构成整个平台的基础；Configuration 提供了基本的语言特性，Profile 提供针对设备的特殊功能 API 和扩展类库；应用程序的运行环境需要一个 Configuration 和至少一个 Profile，多个 Profile 可以共存，也可以叠加。图 14.3 显示了 J2ME 体系结构。

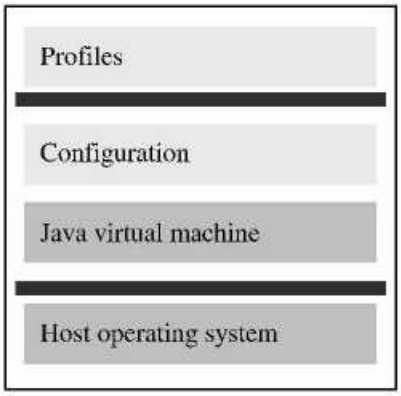


图 14.3　J2ME 体系结构

1　Java　　Java Virtual Machine Layer

这一层是 Java 虚拟机的一个实现，它是为特定设备的主机操作系统定制的，而且支持一个特定的 J2ME 配置（configuration）。为适应占用资源很小的（small-footprint）设备的特性，KVM 已经按以下方式修改：

（1）VM 的大小和类库已减小为 50KB～80KB 目标代码的标准；

（2）存储器占用已经减小为几十 KB 的标准；

（3）在具有 16 位和 32 位处理器的设备上，能有效运行；

（4）体系结构是高可移植的，特定于机器和 / 或平台的代码的总量很少；

（5）多线程和垃圾回收是独立于系统的；

（6）可以对虚拟机的组件进行配置，以适合于特定设备，从而增强了灵活性。

2　　Configuration Layer

J2ME 先将所有的嵌入式装置大体上区分为两种：一种是运算功能有限、电力供应也有限的嵌入式装置(比方说 PDA、手机)；另外一种则是运算能力相对较佳，并且在电力供应上相对比较充足的嵌入式装置(比方说冷气机、电冰箱、电视机顶盒)。Java 引入了一个叫做 Configuration 的概念，然后把上述运算功能有限、电力有限的嵌入式装置定义在 Connected Limited Device Configuration(有限连接设备配置，CLDC)规格之中；而另外一种装置则规范为 Connected Device Configuration(连接设备配置，CDC)规格。也就是说，J2ME 先把所有的嵌入式装置利用 Configuration 的概念分成两种抽象的型态。

其实也可以把 Configuration 当作是 J2ME 对于两种类型嵌入式装置的规格，而这些规格

之中定义了这些装置至少要符合的运算能力、供电能力、存储空间大小等规范，同时也定义了一组在这些装置上执行的 Java 程序所能使用的类别函数库。这些规范之中所定义的类别函数库为 Java 标准核心类别函数库的子集合，以及与该型态装置特性相符的扩充类别函数库。比如就 CLDC 的规范来说，可以支持的核心类别函数库为 java.lang.*、java.io.*、java.util.*，而支持的扩充类别函数库为 java.microedition.io.*。

总之，配置层定义了 Java 虚拟机功能的和特定类别设备上可用的 Java 类库的最小集。从某种程度上说，一个配置定义了 Java 平台功能部件和库的共同性，开发者可以假设这些功能部件和库在属于某一特定类别的所有设备上都是可用的。用户不太会见到这一层，但它对简表（Profile）实现者非常重要。

我们将在配置一节（Configuration）中详细介绍 CLDC 和 CDC。

3　　Profile Layer

区分出两种主要的 Configuration 之后，J2ME 接着定义出 Profile 的概念。Profile 是架构在 Configuration 之上的规格，是为了要更明确地区分出各种嵌入式装置上 Java 程序该如何开发以及它们应该具有哪些功能。简表在一个特定的配置上面实现。应用程序是针对特定的简表编写的，因此可以移植到支持该简表的任何设备上。一个设备可以支持多个简表。

Profile 之中定义了与特定嵌入式装置非常相关的扩充类别函数库，而 Java 程序在各种嵌入式装置上的用户界面该如何呈现就是定义在 Profile 里。Profile 之中所定义的扩充类别函数库是根据底层 Configuration 内所定义的核心类别函数库所建立。

简表定义了应用程序所支持的设备类型。J2ME 使用配置和简表定制 JRE (Java Runtime Environment,Java 运行时环境)。作为一个完整的 JRE，J2ME 由配置和简表组成，配置决定了使用的 JVM，而简表通过添加特定于域的类来定义应用程序。图 14.4 描述了不同的虚拟机、配置和简表之间的关系。它同时还把 J2SE API 和它的 Java 虚拟机放进来进行了比较。虽然 J2SE 虚拟机通常被称为一种 JVM，但是 J2ME 虚拟机、KVM 和 CVM 都是 JVM 的子集。KVM 和 CVM 均可被看作是一种 Java 虚拟机——它们是 J2SE JVM 的压缩版，并专为 J2ME 定制。

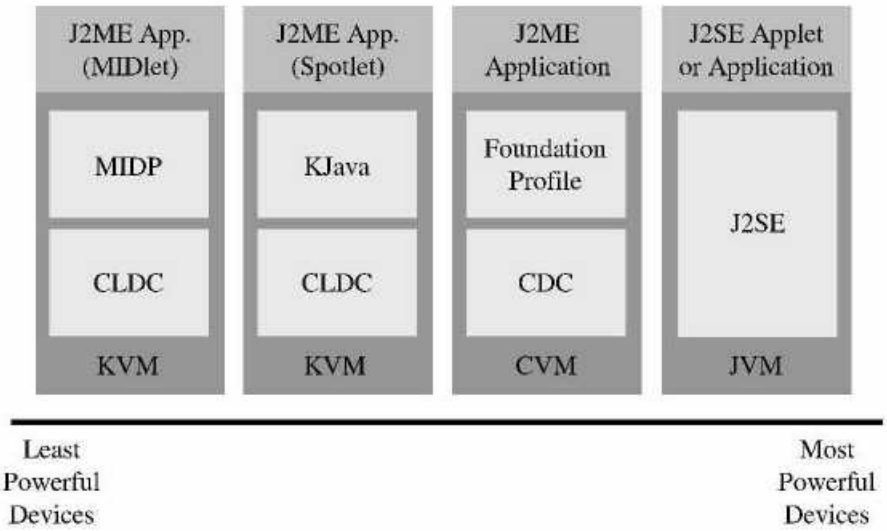


图 14.4　虚拟机、配置和简表之间的关系

14.1.3 J2ME 中的事件处理

J2ME 中的事件处理是以一系列屏幕显示为基础的，这与 Java 平台的台式机版本的事件处理有很大不同。每一屏显示特定的少量数据。

命令以每屏幕为基础提供给用户。Command 对象封装了与动作的语义相关的名称和信息。这个对象主要用于为用户提供动作选择。所产生的命令行为定义在与屏幕显示相关联的 CommandListener 中。

每一个 Command 包含 3 块信息：一个 label（标号）、一个 type（类型）和一个 priority（优先级）。label 用于命令的可视表示；type 和 priority 由系统使用，系统用它们来决定 Command 如何映射到具体用户界面。

图 14.5 显示了 J2ME 中的事件处理机制。

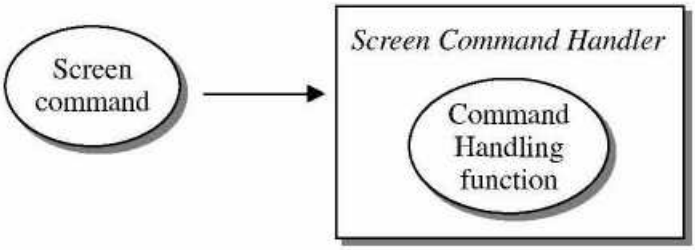


图 14.5 在 J2ME 中处理用户事件

14.1.4 其他概念

1 Personal Java Java Java Java

在 J2ME 出现之前，我们更多接触到的是 Java 卡（Java Card）、嵌入式 Java（Embedded Java）和实时 Java（Real Time Java）以及 PersonalJava 等。它们和 J2ME 有什么关系呢？

（1）Personal Java

Personal Java 是 Sun 公司首次为资源限制设备创建 Java 平台版本的尝试。Personal Java 应用程序环境目标是 Web 连接消费设备——常常执行来自网络的小应用程序。Personal Java 的规格其实是从 Java1.1 之中所分支出来，因此 Personal Java 的规格是根据 Java1.1 的规格而制定的。所以，Personal Java 的规格并没有定义在 CLDC 或者 CDC 下。J2ME 用 CDC 的 Personal Profile 来兼容 Personal Java 规范，从而慢慢取代之。但是 Personal Java 推出的比较早，市场上已经有大量采用这两种规格的作品，所以短期内 Personal Java 仍然会不断演进，一时之间还无法立刻归类到 J2ME 的特定 Profile 之下。

（2）Java Card

Java 卡是针对智能卡（Smart Cards）等设备而定制的最小 Java 子集，比 J2ME 还要小，移植性也不强。Smart Cards 的内存非常有限，为此限制了类的数目（例如，没有 Windows 系统类），并把整个 Java Card 规范在一个二进制核心里实现，然后用 Java 封装。由于智能卡的功能很简单且容易掌握，类不需要频繁改变，这一方法效果很好（对于 J2ME 显然不能使用这种方法）。另外，由于每一个 Java 卡应用的数据和代码是独立在沙箱（sand box）内执行

的，一张卡上可以有多个应用，彼此独立，并且很安全，这样就实现了一卡多用，不需要在钱包里装太多的卡。

虽然 Java 卡是为智能卡设计的，但也可以用于许多其他地方。例如，1998 年的 Java One 大会上的 Java 戒指（Java Ring，又名 Java-Powered iButton，事实上只是 Java Card 的另外一种型态，Dallas Semiconductor 公司将它制作成为钮扣的形状，进而镶嵌在戒指之上），或者世界上现在发布的数百万张 GSM SIM 卡。

（3）Embedded Java

Embedded Java 是历史上 Sun 公司另一次创立嵌入系统 Java 平台的尝试，针对有间断的网络连接或者没有网络连接的设备，这些设备经常是没有图形界面的。Embedded Java 的设计更接近于 J2SE，但是因此太大太慢，而且对系统要求过高。另外，Embedded Java 将控制其运行的平台，也就是接管所有的系统调用、所有的库以及所有的设备，它根本就无法移植。显然提供平台的公司不会喜欢这一点。

Personal Java 偎依在 J2ME 大伞之下，可为什么 Embedded Java 不呢？Embedded Java 不和 Personal Java 同在 J2ME 内，是因为在 Personal Java 和 Embedded Java 应用程序之间有一个基本的差别。Personal Java 应用程序期望连接到某类网络中下载并执行小应用程序。按照这种观点，Personal Java 设备就是一般用途的消费设备；它们的能力可以被扩展。

相比之下，Embedded Java 设备则执行的功能都非常具体的，基本没有必要提供下载新的代码到 Embedded Java 设备的能力。因此，Personal Java 设备使用可扩展 Java 应用程序接口；而 Embedded Java 设备则没有，因为没有必要使用。Embedded Java 基本上完全失败了。

（4）Real Time Java

实时 Java 是由 IBM 领导的 RTJE 组织（Real Time Specification for Java Experts Group，实时定制 Java 专家组）负责实施的，现在还在不断完善中。

2　KJava

另外，还有一个 Java 类库集，它现在差不多可以被认为是另一个简表了。当 Sun 公司为 Palm 开发第一个 KVM 时，他们需要一组类来开发 Palm 的演示程序。这套类库被封装进 com.sun.kjava 程序包，提供了 Palm 的图形用户接口、Palm 数据库访问以及简单的集合类等等。将两者结合，开发者就可以开发 Palm 上的 Java 应用，因此它受到了开发者的广泛欢迎，也有了大量教程和开发实例。

而事实上，到目前为止，这都是在 Palm 上开发 Java 程序的惟一合理途径。CLDC 没有定义用户界面等 API，MIDP 并不适合 PDA，对应的最合适 J2ME 标准是 PDA Profile，但是尚未完成。许多开发者强烈要求 Sun 公司在正式的 J2ME 标准中继续支持它，但是 Sun 公司已经放弃了这一技术，而且并不保证正式的 PDA Profile 拥有与其的兼容性。

在 CLDC 早期的开发中，这些类被广泛地用来测试和演示 J2ME。因为 KJava 是唯一的允许应用程序开发者使用 J2ME 和 KVM 开发应用程序的类，所以它就被广泛使用了。甚至到了今天，一个用于 PDA 或更特殊一点的 Palm 的简表已经在开发中，许多开发者仍然希望使用 KJava 类来开发 PDA 应用程序。尽管 KJava 类不被支持，并且仅仅用于设计测试程序或演示程序，并且它们将被一个即将到来的简表所替代，但是开发者们仍然热衷于使用它来开发。

正如前面提到的那样，KJava 类是最初提供的一个供测试用的类，在 Palm 设备上运行早

期的 KVM 和配置版本。它们将被 PDA 简表代替。KJava 类扩展了 CLDC 并且提供一个图形用户接口、Palm 数据库访问、简单集合类和一个三角法计算器。

14.1.5 J2ME 与 WAP 的关系

J2ME 刚刚提出的时候，经常会有人讨论，在无线 Internet 应用领域它是不是 WAP 的对手？其实，它们之间并没有直接竞争的关系，而是可以共生共存、互补互助的。

WAP 包括两层内容：一层是无线 Internet 传输层，它针对无线的特点制定了 WDP、WTP 等等协议；另一层是应用层，WAP 以 Browser 的方式来访问 Internet。为了适应低速网络的限制，WAP 简化了 Internet 标准的 HTML，制定了 WML 标准，并且只能访问 WML 脚本。而 J2ME 在传输层提供了一组面向应用程序的高层协议，以支持 HTTP 协议。在应用层，J2ME 提供了全功能的 Java 开发环境，可以用 Client/Server 方式来访问 Internet 上的所有数据，而与标记语言无关。

总的来说，WAP 是在线浏览技术，只能以 Browser/Server 模式运行于在线环境，而 J2ME 采用 Client/Server 模式，既可以用于在线环境，也可以用于离线应用。此外，WAP 论坛现在也正在尝试让现有 WAP 浏览器通过 WAP Profile 与 Java 交谈或直接用 Java 编写 WAP 浏览器。

由此可见，J2ME 与 WAP 并没有竞争关系，而是面向不同领域的两套解决方案。如果是面向浏览的应用，比如新闻、天气预报等，可以继续使用 WAP 方案；如果是需要智能处理的应用，比如收发 E-mail、股票信息等，就更适合用 J2ME 实现。

14.2 J2ME 配置（Configuration）

14.2.1 概述

正如前面所定义的，配置将基本运行时环境定义为一套核心类和一个运行在特定类型设备上的特定的 JVM。J2ME 可以在好几个不同的配置中进行配置。到目前为止，只有两种配置规范。通过 Java 规范定义的这两种配置是 Connected Limited Device Configuration(有限连接设备配置，CLDC)和 Connected Device Configuration(连接设备配置，CDC)。

使用 CLDC 开发的 J2ME 应用程序的目标设备通常具有以下特征：

（1）可供 Java 平台使用的 160KB～512KB 的总内存；

（2）功率有限，常常是电池供电；

（3）网络连通性，常常是无线的、不一致的连接并且带宽有限；

（4）用户接口混乱，程度参差不齐，有时根本就没有接口。

一些 CLDC 支持的设备，包括无线电话、寻呼机、主流个人数字助手(PDA)，以及小型零售支付终端。

使用 CDC 的目标设备通常具有以下特征：

（1）使用 32 位处理器；

（2）2MB 或更多可供 Java 平台使用的总内存；

（3）设备要求的 Java2“蓝皮书”虚拟机的全部功能；

（4）网络连通性，常常是无线的、不一致的连接，并且带宽有限；

（5）用户接口混乱，程度参差不齐，有时根本就没有接口。

一些 CDC 支持的设备，包括常驻网关、智能电话和通讯器、PDA、管理器、家用电器、销售网点终端以及汽车导航系统。

图 14.6 描述了支持 J2ME 应用程序的设备，同时说明了 J2ME 适合 Java 平台之处。

图 14.6 J2EE、J2SE 和 J2ME 的适应范围

属于同一类的设备计算能力是相似的，但是其他功能和条件还有非常大的区别。由于 Java 平台必须保证相容性，这就必须舍弃所有的设备特殊性。因此，Configuration 就是支持一组通用设备的最小 Java 平台，作为这些设备的最小公分母来保证不同设备间的平台相容性。这里的 Java 平台主要是指 Java 虚拟机（JVM）和核心库。

14.2.2 连接限制设备配置(CLDC)

CLDC 是由 Java Community Process 创建的。正如 Sun 公司的 Web 站点所定义的那样，它的标准是："轻便、覆盖区域最小的 Java 构建块，适合小型的、有资源限制的设备。"

CLDC 简要描述了高度限制设备上每个 J2ME 执行所要求的一套最基本的库和 Java 虚拟机特征。CLDC 主要面向那些网络连接速度慢、能源有限（经常是电池供电）、具有大于等于 128KB 的 ROM、以及大于等于 32KB 的 RAM 的设备。CLDC 设备使用 ROM 来存储运行时的库和 KVM，或存储为某个特殊设备创建的另一个虚拟机。RAM 被用来分配运行时的内存。

根据 CLDC 规范中所说，运行 CLDC 的设备应该有 512KB 或更少的内存空间、一个有限的电源供给（通常是使用电池）、有限的或断断续续的网络连接性（9600bit/s 或更少）以及多样化的用户界面，甚至没有用户界面。通常说来，这个配置是为个人化的、移动的、有限

连接信息设备而设计的，比如呼叫器、移动电话和 PDA 等。

CLDC 定义了下列要求：

（1）完整的 Java 语言支持（除浮点支持、最终定案和错误处理之外）；

（2）完整的 JVM 支持；

（3）CLDC 的安全性；

（4）有限国际化的支持；

（5）继承类——所有不针对 CLDC 的类都必须是 J2SE1.3 类的子类；

（6）针对 CLDC 的类都在名为 javax.microedition 的软件包和它的子包里。

除 javax.microedition 软件包以外，CLDC API 还由 J2SE 的子集 java.io，java.lang 以及 java.util 等软件包组成。我们将在 CLDC API 这一节中学习有关的细节问题，然后使用 CLDC API 来开发我们的绘图应用程序。

与 J2SE 相比，CLDC 缺少下列特征：

（1）AWT（抽象窗口开发包），Swing 或其他图形库；

（2）用户定义类装载器；

（3）类实例的 finalize 方法；

（4）weak references；

（5）RMI；

（6）Reflection（映射）。

CLDC 有 4 个包：java.lang、java.util、java.io 和 javax.microedition。除了 microedition 包以外，其他的这几个包都是 J2SE 包的核心子集，CLDC 采用这些 J2SE 类库，但是把其中一些在微型设备中用不到的类、属性、方法去掉了。

CLDC J2SE

CLDC 512KB

RMI

Java

J2SE

Float Double

CLDC

CLDC 要求其底层的虚拟机能够辨别并拒绝非法的 class 文件。但由于 CLDC 本身面向小内存消耗的小型设备这一前提，其类文件检测机制与 J2SE 中定义的标准类文件审核机制有所不同。CLDC 中有一步其特有的预审核（preverification）过程，这是 CLDC 区别于通常的类文件审核过程的关键。

如图 14.7 所示，当程序的源程序被编译后，必须被预审核器预审核，然后才能生成可以

被下载到目标设备上运行的类文件。之所以这么做，主要是为了减轻目标设备上虚拟机中审核器的负担，加快其审核速度，所以把一部分的审核任务放在预审核器中完成。请注意在CLDC/MIDP环境下开发程序，其程序经过编译后，必须经过预审核后才能运行。

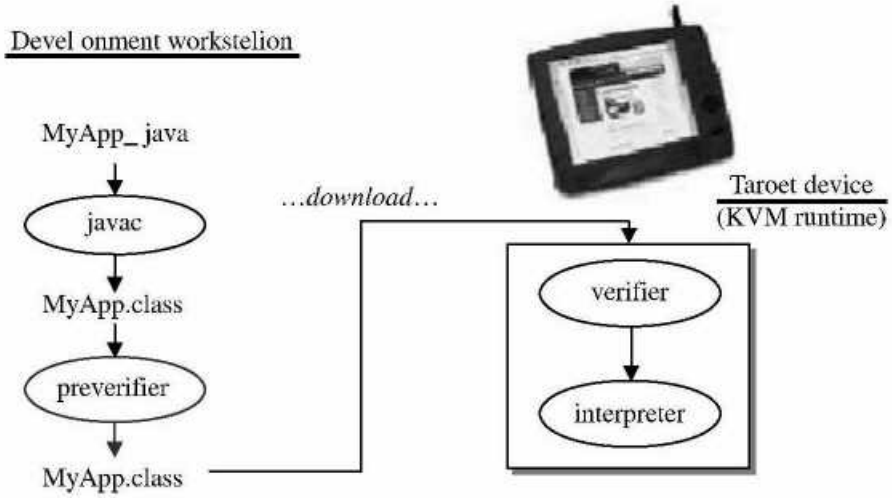


图 14.7　CLDC 的文件审核机制

14.2.3　CLDC API

如 14.2.2 节所述，CLDC API 实际上只是 J2SE 的一个子集，它包括 java.lang、java.io 和 java.util，另加一个新软件包 javax.microedition。我们将逐个来研究这些软件包，并突出显示每一个包中的重要的类。

尽管每一个类都在 J2SE 中，但是没有必要让每一个类的 CLDC 实现都能实现 J2SE 支持的所有方法。您可以检查 CLDC API 文档以确认哪些方法是受支持的。文档的副本位于安装 J2ME CLDC 时创建的 j2me_cldc/docs 目录下。它提供 PDF 和 javadoc 两种格式。

1　java.lang

CLDC java.lang 软件包是 J2SE java.lang 软件包的一个子集。与 J2SE 相比，它最引人注目的可能便是冗长的浮点操作了，特别是浮点(Float)和双精度(Double)类。如果使用浮点的话，这些冗余将涉及到所有其他的类。

相对于 J2SE v1.3 API，CLDC API 中删去了几个其他的类。其中包括 ClassLoader、Compiler、InheritableThreadLocal、Number、Package、Process、RuntimePermission、SecurityManager、StrictMath、ThreadGroup、ThreadLocal 和 Void。

我们描述了可从下面几页表中的 CLDC java.lang 软件包中获取的主要的类。Java 开发人员对所有这些类的使用都应该是非常熟悉了。

除这些核心类之外，读者还将看到 CLDC 支持的 Runnable 接口，正像 Exception、Error 和其他有关的类一样.

（1）java.lang 核心运行时类

java.lang 软件包的核心运行时类有下述几种。

- Class：表示正在运行的 Java 应用程序中的类和接口。
- Object：与在 J2SE 中相同，Object 是所有 Java 对象的基本类。
- Runtime：为 Java 应用程序提供一种与运行时环境（Java 应用程序在其中运行）进行

交互的方法。

- System：提供一些静态的帮助方法，就像为 J2SE 提供方法一样。
- Thread：定义 Java 程序的一个执行线程。
- Throwable：Java 语言中所有错误和异常的超级类。

（2）java.lang 核心数据类型类

java.lang 软件包中的核心数据类型类有下述几种。

- Boolean：包装 boolean 原始数据类型。
- Byte：包装 byte 原始数据类型。
- Character：包装 char 原始数据类型。
- Integer：包装 int 原始数据类型。
- Long：包装 long 原始数据类型。
- Short：包装 short 原始数据类型。

（3）java.lang 帮助类

java.lang 软件包的帮助类有下述几种。

- Math：包含执行基本数学运算的方法。请注意，所有执行浮点值运算的方法都被省略了，仅保留了关于 integers 和 longs 的方法：abs()、min()和 max()。
- String：在 Java 中代表对象 String，就像在 J2SE 中一样。
- StringBuffer：代表一个可以修改的串，就像在 J2SE 中一样。

2 java.io

（1）java.io 输入类

CLDC API 包含许多 J2SE 中共同使用的输入类。特别地，CLDC java.io 软件包中包括下面一些类。

- ByteArrayInputStream：包含一个内部缓冲器，它代表可能从输入流中读取的字节。
- DataInput：一个接口，从二进制输入流提供字节以供读取并把它们转换成原始 Java 数据类型。DataInputStream 提供该接口的实现。
- DataInputStream：允许应用程序以独立于平台的方式从基层输入流中读取原始 Java 数据类型。
- InputStream：一个抽象类，它是所有代表字节输入流的类的超级类。
- InputStreamReader：读取字节并把它们按照指定的字符编码方法转换成字符。
- Reader:一种读取字符流的抽象类。

J2SE

java.lang

（2）java.io 输出类

CLDC API 包含了许多 J2SE 中的共同使用的输出类。特别是，CLDC java.io 软件包中包括下面一些输出类.

- ByteArrayOutputStream：实现一个输出流，在此输出流中，数据被写入字节数组。

- DataOutput：一种接口，提供原始 Java 数据类型以供写入二进制输出流。DataOutputStream 提供该接口的实现。
- DataOutputStream：一个输出流，允许应用程序以一种便捷的方式编写原始 Java 数据类型。
- OutputStream：一个抽象类，它是所有代表字节输出流的类的超级类。
- OutputStreamReader：给出字符，并按指定的字符编码方法将其转换为字节。
- PrintStream：添加一种便捷的方法来打印数据值的文本表现形式。
- Writer：编写字符流的一个抽象类。

其中一些类可能不包含 J2SE 支持的所有方法，比如浮点方法和双精度方法。

3 java.util

CLDC java.util 软件包中包含 J2SE java.util 软件包中最常用的类。这些类中包括 4 个收集类（实际是 3 个收集类和一个接口），以及日期 / 时间和实用程序类。

（1）java.util 收集类

CLDC 支持的 java.util 收集类有下述几种。

- Enumeration：一个接口，通过项目集允许对例程进行重复调用。
- Hashtable：实现 hashtable，将键映射到值。
- Stack：代表了一个后进先出(LIFO)的对象集合或堆栈。
- Vector：代表可以调整大小的对象“数组”或者矢量。

（2）java.util 其他类

CLDC 支持的 java.util 类中的其余部分，包括日期和时间类，以及 Random 实用程序类。下表中简要列出了这些类。

- Calendar：一个抽象类，使用一套整型字段,如 YEAR、MONTH、DAY 等来获取和设置日期。
- Date：代表特定的时间和日期，精确到毫秒级。
- Random：一个实用程序类，用来生成 int 或 long 的随机值流。
- TimeZone：代表时区的偏移量，也可用于校正时间。

3 java.microedition.io

迄今为止，我们在 CLDC API 中看到的所有的类都是 J2SE API 的子类。CLDC 还包含一个附加的软件包，即 javax.microedition.io。

在这个包里惟一被定义的类就是 Connector 类，也称为工厂类，包含创建 Connection 对象或输入/输出流的方法，来替代许多 J2SE 网络输入/输出类。

当动态识别一个类的名字时，Connection 对象就被创建了。类名称的识别基于平台名称和被请求连接的协议。描述目标对象的参数串应该满足 RFC 2396 规范所要求的格式。请使用下列格式：

```
{scheme}:[{target}][{params}]
```

{scheme}是一个协议的名称，如 http 或 ftp。{target}通常是一个网络地址，但是面向非网络的协议则可能把它当作一个相当灵活的字段来处理。还有一些参数，如{params}被指定为一系列形如“;x=y”的分配形式（例如，;myParam=value）。下面列出使用同一个 Connector 类创建和打开五种不同类型的连接的方法：

（1）HTTP Connector.open("http://www.xyz.com")；

（2）套接字 Connector.open("socket://111.222.111.222:9000")；

（3）通讯端口 Connector.open（"comm:1；baudrate=9600"）；

（4）数据报 Connector.open（"datagram://111.222.111.222"）；

（5）文件 Connector.open（"file:/xyz.dat"）；

一般连接器结构提供给应用程序开发者一个到通用低水平硬件的简单的映射表。成功执行 open 语句将返回一个实现一般连接界面的对象。

除类属连接工厂类之外，javax.microedition.io 软件包中还包含下列面向连接的接口。

- Connection：定义了最基本的连接类型，这个接口也是此软件包中所有其他连接接口的基本类。
- ContentConnection：定义了一个可以通过内容的流连接。
- Datagram：定义了一个类属数据报接口。
- DatagramConnection：定义了类属数据报连接和它必须支持的性能。
- InputConnection：定义了一个类属输入流连接和它必须支持的性能。
- OutputConnection：定义了一个类属输出流连接和它必须支持的性能。
- StreamConnection：定义了一个类属流连接和它必须支持的性能。
- StreamConnectionNotifier：定义了一个流连接的通告程序必须具有的性能。

14.2.4 连接设备配置(CDC)

连接设备配置（CDC）被定义为一种添加了 CLDC 类的 Java2 标准版(J2SE)的简化版。因此，CDC 是建立在 CLDC 的基础之上，并且为 CLDC 设备开发的应用程序也可以运行在 CDC 设备上。

CDC 也是由 Java Community Process 开发的，它涵盖了个人电脑与有至少 512KB 内存的小型设备之间的中间地带。现在，这一类设备通常是共享的、固定的(不用移动)网络连接信息设备，像电视机机顶盒、网络电视系统、互联网电话与汽车导航/娱乐系统等。

CDC 基于 J2SE1.3 应用程序接口，包含所有定义在 CLDC 规范(包括 javax.microedition 程序包)中的 Java 语言应用程序接口。与 CLDC 相比，CLDC 所有缺少的特性和类在 CDC 中都被补齐，包含映射、最终化、所有的错误处理类、浮点数、属性、输入/输出(File、FileInputStream 等等)和弱的引用。一般说来，CDC 中预期的类包括一个 J2SE 子集和一个完整的 CLDC 超集，图 14.8 描述了 CDC 和 CLDC 之间的关系。同时该图也揭示了它们与整个 J2SE API 的关系。正如前面所说，CDC 是加上一些额外类的 J2SE 的子集，CLDC 是 CDC 的子集。

就像使用所有的配置一样，CDC 有基层虚拟机的具体的必要条件。根据 CDC 规范，基层虚拟机必须提供实现完整的 Java 虚拟机的支持。如果虚拟机实现有一个用于激活设备的本地方法的界面，则它必须兼容 JNI1.1 版本；如果虚拟机实现有一个调试界面，则它必须兼容 Java 虚拟机调试界面(JVMDI)规范；如果虚拟机有一个简表界面，则它必须兼容 Java 虚拟机简表界面(JVMPI)规范。可见，为了实现这些功能，CDC 肯定会变得很大，就不能称其为 K 虚拟机了，因此，通常称用于 CDC 的虚拟机为 CVM，这里的 C 代表 compact、connected、consumer。

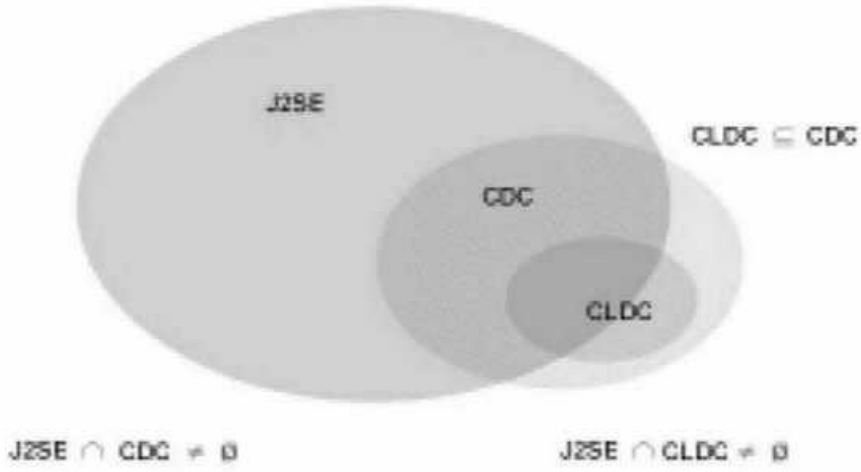


图 14.8　J2SE、CLDC 和 CDC 的关系

14.2.5　CDC API

CDC 运行在 C 虚拟机(CVM)的顶部，与基础表关联在一起。基础表是一套 Java API，专为要求定制用户界面(UI)的高端设备服务，通常由设备制造商提供。

CDC 是建立在 CLDC 顶部的 API，是整个 J2SE API 的一个更完整的子集。它还包含一个额外的软件包，即 javax.microedition.io 软件包，其中包含 CLDC 中定义的所有相同的类和接口等。

下面列出的是 CDC 中的一些更值得注意的功能是 CLDC 中所没有的。

- 支持浮点数（包括 java.lang.Float、java.lang.Double 和 java.lang.StrictMath 类）。
- classloader 类(java.lang.ClassLoader)。
- 支持本地进程(java.lang.Process)。
- 高级多线程支持（包括支持线程组和更多线程）。
- 串行化的类(java.io.Serialiable 和 java.io.Externalizable)。
- 映像 API（包括 java.lang.reflect 软件包）。
- 文件系统支持。
- 支持 J2SE 类型网络(java.net)。
- 对 J2SECollections API 更完全的支持。
- 为 javax.microedition.io 软件包增加一个 HttpConnection 接口，这样可为 HTTP 连接提供必要的方法和常量。
- 支持 J2SE 的 java.lang.ref、java.math、java.security、java.security.cert、java.text、java.util.jar 和 java.util.zip 软件包。

在资源限制的条件下，CDC 补充了 CLCD 的不足，并针对大于 2MB 内存的设备，它能支持标准 Java 虚拟机和 Java 编程语言的完整实现。简而言之，CDC 非常接近用户熟悉的 Java 规范。

当只需要兼容 CLDC 的虚拟机来支持标准 Java 虚拟机功能性的一个子集时，那么 CDC 指定的虚拟机必须和标准 JVM 特性兼容。这意味着，如果包括对本地方法调用的支持，CDCJVM（或 CVM）必须符合 JNI（Java 本地接口）1.1；如果包括对调试的支持，那就必须符合 JVMDI（Java 虚拟机调试界面）；如果需要包括简档支持，那就必须遵从 JVMPI（Java

虚拟机简档界面)。

在类库层中，CDC 提供支持全兼容 Java2 虚拟机所必需的最小 API 集。这一 API 集包括所有为 CLCD 定义的 API 和针对文件 I/O、连网、高级安全性、对象序列等 API。表 14.1 列出了在 CDC 规范下提供的包、每个包中的类和界面的数量和每个包的描述。

表 14.1　　在 CDC 规范下的所有包

包	描　述	类和界面
java.io	系统输入与输出	62
java.lang	Java 编程语言基本类	77
java.lang.ref	特别参考类	5
java.lang.reflect	反映支持	12
java.math	Math 支持	1
java.net	网络类和工具	23
java.security	安全支持	36
java.security.cert	证书支持	4
java.text	文本处理类	13
java.util	集合、日期和时间支持，各种实用工具类	47
java.util.jar	Jar 文件支持	7
java.util.zip	Zip 文件支持	9
javax.microedition	类属连接	10

很明显，表中缺少属于 java.awt 包中的类和界面。与 CLDC 的情形相同，CDC 不支持任何用户界面。这是由于设备与设备之间的用户界面差异很大所造成的，必须为 CDC 加入合适的简档以获得用户界面支持。

14.3　J2ME 简表（Profile）

在 Configuration 中舍弃了设备的特殊性来保证 Java 平台的相容性，但是仅仅有 Configuration 显然是不够的，因为特殊的具体设备其独有的功能和硬件条件都没有得到支持。为此，在 Configuration 的基础上，根据设备具体功能再进行一次划分，比如智能手机、PDA 等。功能的划分也就是其他硬件条件的划分，比如屏幕大小、内存、计算能力以及电力供应等都是从属于功能的，因此同功能的设备的硬件条件都是非常相似的。Profile（简表）就是针对每一类功能设备的特殊性定义的与设备特性相关的 API，建筑于 Configuration 之上，作为 Configuration 的扩展和补充。例如 MIDP，(Mobile Information Device Profile，移动信息设备 Profile)就定义了关于移动信息设备(主要指智能手机和一部分具有无线通信功能的 PDA)

的图形界面、输入和时间处理、持久性存储以及短消息等的 API，并且考虑到了移动信息设备的屏幕和内存限制。而正在制定中的 PDA Profile 则定义了针对 PDA 的 API，其屏幕和内存条件都要大于 MIDP，但是网络方面的要求则显然和手机有所不同。如图 14.9 所示，同属于一个 Configuration 的设备根据功能不同由不同的 Profile 来支持，Profile 体现设备的特殊性，但是都构筑在一个共同的基础 Configuration 平台之上。

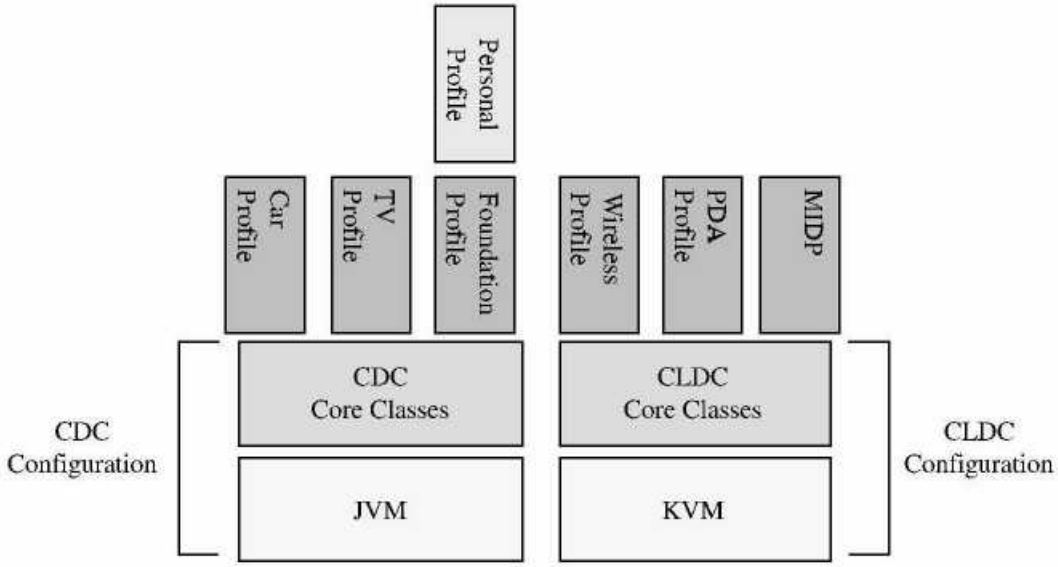


图 14.9　Configuration 和 Profile 之间的关系

Configuration 的分类是根据计算能力的不同来划分的，同类设备的计算能力相近。Configuration 是一个规范，定义了这类设备的共同 Java 平台，定义与设备无关的 Java 虚拟机和核心库，是平台相容性的基础。Profile 的分类是根据设备功能划分的，同类功能的设备其他各种硬件条件和需求也相近。Profile 是一组 API，在某一 Configuration 的基础上扩展了针对设备特定功能的 API，使得标准能够完全适应特殊的设备，彻底发挥设备的功能。简表为相同消费电子设备的不同的生产商提供了标准化的 Java 类库。事实上，虽然配置规范的开发由 Sun 公司领导，但是许多简表规范仍将继续由特殊设备的供应商领导。比如说，Motorola 领导了移动电话和呼叫器简表规范的开发，又如 Palm 领导 PDA 简表的开发。

现在，5 个已知简表已经有了规范，记住，每个简表的责任都是为了完善配置的不足，如表 14.2 所示。

表 14.2　　**已定义的简表**

简　表	完 善 配 置
Mobile information devices profile (MIDP)	移动电话和呼叫器 CLDC
Personal digital assistant profile	Palm 和 Handspring 的 PDA 设备 CLDC
Foundation profile	用于所有不需要 GUI 的 CDC 设备的标准简表 CDC
Personal profile	替代 PersonalJava 的 Foundation 完善的简表 CDC
RMI profile	提供 RMI 的 Foundation 完善的简表 CDC

既然 Profile 建构在 Configuration 之上，其意义就是说：Profile 之中所规范的配备需求不可能比 Configuration 还要低。同时，Profile 之中对于显示功能、网络功能以及耗电能力等相关需求将会比 Configuration 之中所规定的还要高。表 14.3 是一些 Profile 的配备需求。

表 14.3　　部分 Profile 的配备要求

配备\Profile	Foundation Profile	Personal Profile	MIDP
RAM	至少 512K	至少 1 MB	RAM 与 ROM 至少要为 512K
ROM	至少 1024K	至少 2.5MB	RAM 与 ROM 至少要为 512K
电源	不设限	不设限	通常是使用电池，所以电源有限
网络连接能力	部分功能	部分功能	具有低频宽的无线通讯能力
其他	要有额外的 RAM 或 ROM 供应用程序执行	要有额外的 RAM 或 ROM 供应用程序执行	要有额外的 RAM 或 ROM 供应用程序执行并储存资料

虽然 Sun 公司的官方文件中描绘了 J2ME 的美好远景，其实这些规格还不完善。目前，有关 Configuration 的规格只有完整的 CLDC 规格(v1.0)。而有关 Profile 的规格只有 Mobile Information Device Profile 的规格已经制定(v1.0)，而其他如 Personal Profile、Foundation Profile、RMI Profile 等，因为是根据 CDC 来制定，而 CDC 的规格尚未制定(v0.2)，所以这 3 个 Profile 的规格目前只有 v0.x 版。其余的 Profile 就连规格都还在草拟之中了。如果读者对这些正在草拟的规格有有兴趣的话，请到 Java Community Process(JCP) 的网页 http://java.sun.com/aboutJava/communityprocess/上查看 Java Specification Request(JSR)的后续进展。

1　MIDP

Mobile Information Device Profile(移动信息设备简表，简称 MIDP)是第一个实现的简表，MIDP 适合诸如蜂窝电话和寻呼机等移动设备。MIDP 和 KJava 一样，也是建立在 CLDC 之上的，并且提供一个标准的运行时环境，允许在终端用户设备上动态地部署新的应用程序和服务。

MIDP 是一个公共的、为移动设备设计的工业标准简表，它不依赖某个特定的商家。对于移动应用程序开发来说，它是一个完整的、受支持的基础。

MIDP 包含下列软件包，前面 3 个是核心 CLDC 软件包，另加 3 个是特定于 MIDP 的软件包。我们将在 14.4 节中讨论每个软件包。

java.lang

java.io

java.util

javax.microedition.io

javax.microedition.lcdui

javax.microedition.MIDlet

javax.microedition.rms

2　PDA Profile

Palm 公司是开发 PDA 简表规范的领头人，这个简表也是完善了 CLDC，在相当长的一段时间内，它都将是 KJava 类程序包的替代品。Java 规范建议这个 Profile 至少应当提供两个核心功能片段：一个用户界面显示工具包，适合于“有限的尺寸和深度显示”，另一个持久数据存储器机制。显示工具包应该是抽象窗口工具包的一个子集，而持久机制将为应用程序、数据、配置/环境信息提供简单的数据存储。

3　Foundation Profile

下面 3 种简表不是非常常见的，这 3 种简表的职责都是为了完善 CDC。Personal 和 RMI 简表实际上是 Foundation 简表的扩展。Foundation 简表的任务是担任一个基础简表，便于以后开发出来的提供图形用户接口、网络等功能的简表附着在它之上。除了用于基础简表，Foundation 简表还提供完整网络的支持，不管有没有使用图形用户接口。

4　Personal Profile

在当前的规范需求下，Personal 简表提供下一代 Personal Java 环境。这个简表允诺，提供互联网连接性和 Web 保真度以及一个能够运行 Java applets 的 GUI。

5　RMI Profile

回想一下 CDC 配置为共享的、固定网络连接信息设备提供最小的 Java 环境。RMI 简表将通过提供 Java 到 Java 的 RMI 来协助提供更好的网络连接性。通过使用 J2SE(1.2.x 或更高版本的)RMI，这个简表将允许这些网络设备与其他系统应用程序交互操作(这个系统不必也运行 J2ME)。

14.4　MIDP 与手机应用程序开发

回顾移动电话的发展历史，我们不难发现移动电话上应用软件的发展也经历了 3 个阶段。传统的移动电话通常只有通话和短消息功能，只能提供基本的语音服务。随后移动电话上又增加了一些简单的附加应用，如电话簿和电话铃声编辑功能等。而现在随着 WAP 技术的发展，移动电话增加了访问 Internet 的功能，使用户可以直接在手机上以无线方式浏览网页。然而，随着无线 Internet 新应用的出现，新的问题也随之而来。

首先面临的是开发瓶颈的问题。目前，手机类嵌入式系统普遍使用 C 语言和专用的实时操作系统，开发速度慢，也没有动态加载应用程序的能力。移动电话上应用程序的开发变得越来越困难，一方面单纯依靠手机厂商自身的软件开发能力难以满足市场的需求，而另一方面广大的软件开发商却又无法参与进来，开发适用于移动电话的应用程序。这无疑极大制约了新应用的推广与普及。

其次，移动电话访问 Internet 只能通过 WAP 方式，而 WAP 采用 Browser/Server 方式访问 Internet，功能有限。现在的 WAP 解决方案要求手机通过 WAP 网关才能访问 Internet，而且只能访问 WML 而不是主流的 HTML，也不能显示复杂格式的图形。此外，因为现有的 WAP 解决方案不够智能，而且不能访问本地存储区，如果进行在线交易会增加服务器负荷，反应速度慢，使无线 Internet 应用受到了很大的限制。

J2ME 的出现则使上述问题迎刃而解。因为 Java 语言是跨平台运行的，这一特性使第三

方软件开发商可以很容易地介入进来开发应用程序，也可以很方便地将应用程序安装移植到移动电话上，开发周期也大大缩短，而且还能支持应用程序的动态下载和升级。J2ME 提供了 HTTP 高级 Internet 协议，使移动电话能以 Client/Server 方式直接访问 Internet 的全部信息，不同的 Client 访问不同的文件，此外还能访问本地存储区，提供最高效率的在线交易。

如图 14.10 所示，J2ME 除了能够更好地增强完善移动电话上已有的应用外，还进一步增加了字典、图书、游戏、遥控家电和定时提醒等新的应用，并能访问电子邮件、即时消息、股票和电子地图等信息。

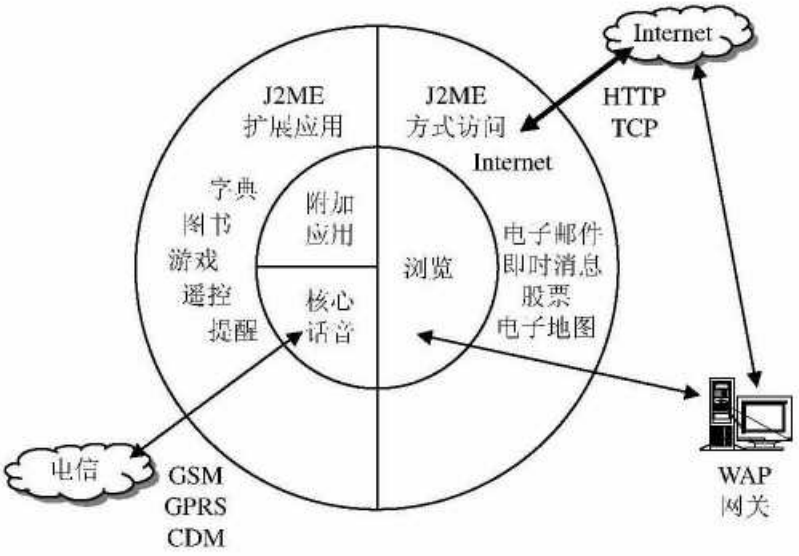


图 14.10　J2ME 在移动电话上的应用

14.4.1　MIDP

MIDP(Mobile Information Device Profile，移动信息设备简表)适合类似于移动电话和寻呼机这样的设备。MIDP 建立在 CLDC 之上，提供一种标准的运行时环境，允许在终端用户设备上动态地配置新的应用程序和服务。

1　MIDP API

（1）MIDP API 类的完整集合可以分为下述 3 个类别。

① 用于用户界面（UI）的 MIDPAPI。设计这些 API 是为了能以一系列屏幕显示为基础与用户进行交互操作，每一屏幕显示把适量的数据显示给用户。命令以每屏幕为基础提供给用户。

MIDP 包括一个低级的 UI API 和一个高级的 UI API。低级的 API 允许用户完全访问一个设备的显示屏，也允许访问按键和指针事件。然而，使用低级 API 时，没有可用的用户界面控件，应用程序必须精确地绘制出按钮和其他所有的控件。

相反，高级 API 提供简单的用户界面控件但不能直接访问原始的输入事件或显示屏。由于显示屏的尺寸和 MIDP 设备输入方法的差异，控件显得很抽象。MIDP 的实现确定了绘制控件的方法，也确定了如何管理用户输入。

② 用于处理数据库的 MIDP API。这些 API 负责组织和操作设备数据库，这个数据库由在 MIDlet 的多个调用之间跨越时保持持久的信息组成。

③ 底层的 CLDC API 用于处理字符串、对象和整数。还提供了 Java2 API 的一个子集，

用于处理 I/O 和网络通信。

（2）MIDP 包含 4 个核心 CLDC 软件包(java.lang、java.io、java.util 和 javax.microedition.io)，另加下面的 3 个特定于 MIDP 的软件包。

① javax.microedition.lcdui：定义用来控制 UI 的类。这个软件包既包含高级 UI 类（例如 Form、Command、DateField 和 TextField 等），又包含低级 UI 类（允许用低级方式控制 UI）。

② javax.microedition.MIDlet：包含 MIDP 主类中的一个 MIDlet 类，为 MIDP 应用程序提供访问关于其运行所在环境信息的权限。

③ javax.microedition.rms：定义一个类的集合，这些类为 MIDlet 提供了永久存储并随后重新得到数据的机制。

（3）除了上面新的软件包之外，MIDP 还向核心 CLDC 软件包添加了 4 个新类。

① java.util.Timer：用于为后台线程中将来要执行的任务确定时间。

② java.util.TimerTask：被 java.util.Timer 类使用，用来为后台线程中稍后的执行定义任务。

③ javax.microedition.io.HttpConnection：一个接口，为 HTTP 连接定义必要的方法和常量。

④ java.lang.IllegalStateException：一个 RuntimeException，指出在不合法或不合适的时间已经调用的一个方法。

2　MIDlet

遵照 MIDP 和 CLDC 规范编写的 Java 应用程序称为 MIDlet（这种命名方式如同我们熟悉的 Applet）。MID 简表的核心是一个 MIDlet 应用程序，这个应用程序继承了 MIDlet 类，以允许应用程序管理软件对 MIDlet 进行控制、从应用程序描述符检索属性以及对状态变化进行通知和请求。

与通常的 Java 程序相比，MIDlet 有比较大的不同。从某种意义上来说，MIDlet 更类似与 Applet。简而言之，与 J2SE 程序相比，MIDlet 没有 main()这个程序初始入口点，同时，MIDlet 也不能调用 Runtime.exit()和 System.exit()方法来中止虚拟机的运行。如果调用的话，将会抛出 SecurityException 异常。

MIDlet 是在 MID 设备上运行的 Java 程序，每一个 MIDlet 程序都必须继承自 javax.microedition.MIDlet.MIDlet 类，即运行时环境（应用程序管理器）和 MIDlet 应用程序代码之间的接口。MIDlet 的继承体系如图 14.11 所示。

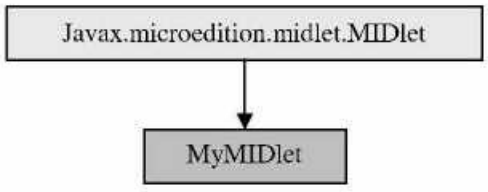


图 14.11　MIDlet 的继承体系

MIDlet 类提供了用于调用、暂停、重新启动和终止 MIDlet 应用程序的 API。因此每一个 MIDlet 程序都必须实现 startApp()、pauseApp()和 destroyApp()方法，这些方法类似于 J2SE 的 java.applet.Applet 类中的 start()、stop()和 destroy()方法。

（1）只要应用程序被激活、构造器执行完毕之后就会立即调用 startApp()方法而不是在应用程序最初启动的时候这样做。应用程序在一次运行的过程中会在活动和不活动状态之间多

次转变，这样就不必编写单独运行的初始化代码了，比如初始化用户界面的代码等，因为这类代码很可能会执行好多次。为此应该采用构造器来完成同一功能。

（2）管理器指示应用程序关闭之后就会调用 destroyApp()方法。和 startApp()方法不一样的是该方法只在应用程序生存期内调用一次，所以在这个方法内编写清除代码是很安全的。实际上，由于 MIDP 并没有为对象包括 finalize 函数，所以用户不得不在以上方法处执行清除功能。同时，由于典型的移动设备比通常情况下的标准平台欠缺稳定，经常被用户进行开关机或者复位操作。所以也不能真正指望 destroyApp()派上大用场。

（3）最后的抽象方法就是 pauseApp()了。该方法主要作用是发出这样的通知：因为用户转换到其他应用或者采用了设备的某项功能促使应用程序不能继续运行而暂时停止应用程序的运行。由于大多数移动设备都缺乏执行多任务的处理能力，以上的这类情况是完全可能发生的。所以在这个方法中应该编码释放所有资源。一旦应用程序重新开发运行，则应用程序管理器会再度调用 startApp()方法。

应用程序管理软件可以在运行时环境内管理多个 MIDlet 的活动。此外，MIDlet 可以自己发起一些状态变化，并把这些变化通知给应用程序管理软件。

除了扩充 javax.microedition.MIDlet.MIDlet 的主 MIDlet 类之外，一个 MIDP 应用程序通常还包括其他一些类，这些类能随它们的资源一起被打包成为 jar 文件，称之为 MIDlet 套件。一个 MIDlet 套件中的不同 MIDlet 能共享 jar 文件的资源，尽管不同套件中的 MIDlets 不能直接相互作用。

3 MIDlet

MIDlet 在应用程序生命周期中有 3 种可能的存在状态，即运行状态、暂停状态、销毁状态。运行状态，正如其名称所暗示的，意味着 MIDlet 正在运行中；这种状态始于 startApp 方法被调用时。在暂停状态中，MIDlet 持有的所有资源将被释放，但是它准备着再次被运行；调用 notifyPaused 方法时，MIDlet 处于暂停状态。在销毁状态中，MIDlet 已经永久地将其自身关闭，释放所有的资源，等待着废物清理程序的处理。它是通过 notifyDestroyed 方法来调用的。一个 MIDlet 程序的起始结束的流程如图 14.12 所示。

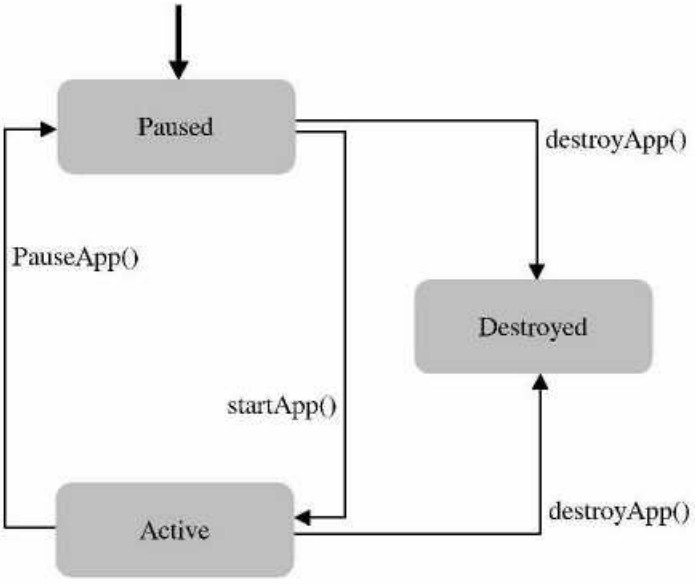


图 14.12 MIDlet 程序的生命周期

4 MIDlet

当然，通讯必须是双向有效的，MIDP 应用程序也不例外。MIDlet 提供了一组方法，用来同应用程序管理器进行通讯。

（1）NotifyDestroyed 告诉管理器应用程序要关闭了。调用该方法不会执行 destroyApp() 方法，所以必须手工调用它。

（2）NotifyPaused 通知管理器应用程序要暂停了。

（3）ResumeRequest 要求应用程序管理器重启暂停的应用程序。

（4）GetAppProperty 从输入或者应用程序描述文件中获取应用程序的配置信息。

14.4.2 开发 MIDlet

1

Sun 公司推出的一个工具包 J2ME Wireless Toolkit 简化了 MIDlet 的开发。该工具包支持在与移动信息设备框架 (MIDP) 兼容的设备（如蜂窝电话、双向寻呼机和掌上电脑）上运行的 Java 应用程序的开发。KtoolBar（随 J2ME Wireless Toolkit 提供）是带 GUI 的最小开发环境，可编译、打包和执行 MIDP 应用程序。其他工具仅需要第三方的 Java 源文件编辑器和调试器。同时 J2ME Wireless Toolkit 还可以与其他 IDE 进行集成，如 Sun ONE Studio 4, Mobile Edition（以前称为 Forte™ for Java™），可以在相同环境中编辑、编译、打包、执行或调试 MIDP 应用程序。

其中，如果用户要进行实际开发，必须选择 JavaTM 2 SDK、Standard Edition(J2SE SDK) 1.3 或更高版本；如果只计划运行演示应用程序，则选择 Java 2 Standard Edition Runtime Environment(J2SE JRE)即可。

图 14.13 说明了 MIDP 应用程序开发的各个阶段（编辑阶段除外）以及工具包在各个阶段中所起的作用。

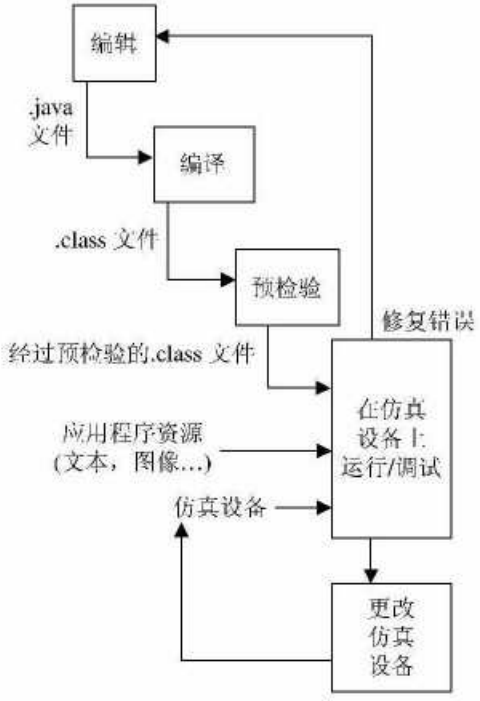


图 14.13 测试和开发 MIDlet 应用程序过程

① 首先，用编辑器编辑 java 源文件。

② 在 KToolbar 或工具包兼容环境（如 Sun ONE Studio 4, Mobile Edition）中编译源文件，该开发环境使用 Java 2 SDK, Standard Edition (J2SETM SDK) 编译器编译源文件。

③ 编译源文件之后，开发环境将生成的类文件传递给预校验器。由其重新整理类中的字节代码，以在 CLDC 虚拟机上简化字节代码校验的最后阶段。它还检查 CLDC 不支持的虚拟机功能的使用情况。

④ 在 KToolbar 或工具包兼容环境（如 Sun ONE Studio 4, Mobile Edition）中使用仿真器运行和调试应用程序。仿真器可以模拟用户在特定设备上运行应用程序的情形，并测试应用程序应用到不同的设备上的可移植性。

⑤ 打包和在真实设备上运行程序。

打包的目的是防止代码被反编译和减小 Java 字节代码的大小，从而产生较小的 JAR 文件，并使下载速度更快。打包之后，就可以在真实设备上运行程序了，如图 14.14 所示。

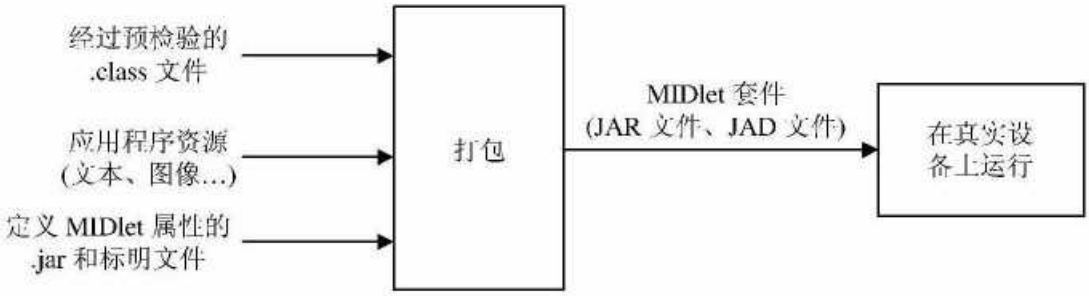


图 14.14　打包应用程序

运行时可共享资源的一组 MIDlet 称为 MIDlet 套件。开发环境如 KToolBar 会自动对 MIDlet 套件进行打包。图 14.15 说明 MIDlet 套件的组织方式。

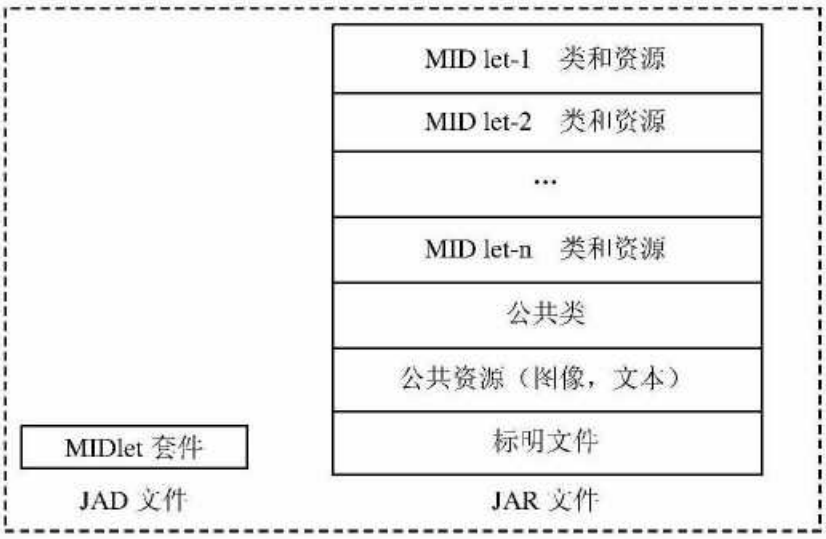


图 14.15　MIDlet 套件组件

Java 应用程序描述符 (JAD) 文件包含一组预定义的属性（由以“MIDlet-”开头的名称表示），这些属性允许应用程序管理软件识别、检索和安装 MIDlet。在 JAD 文件中出现的所有

属性都可用于 MIDlet。可定义自己的特定于应用程序的属性，并将其添加到 JAD 文件中。

Java 归档 (JAR) 文件包含：

① 套件中每个 MIDlet 的 Java 类；

② MIDlet 之间共享的 Java 类；

③ 由 MIDlet 使用的资源文件，如图像文件；

④ 描述 JAR 内容和指定属性（应用程序管理软件使用这些属性来标识和安装 MIDlet 套件）的标明文件。

MIDP 之所以定义 JAD 文件，就是因为如果只是通过 JAR 文件中的清单文件来描述 MIDlet，那么要获得 MIDlet 信息，势必得下载全部的 JAR 文件，这显然是划不来的（Jar 文件可能会较大）。基于这个原因，MIDP 把清单文件中的一部分属性和一些额外的属性集成到一个单个的 JAD 文件中，这样如果想要安装某 MIDlet suite，首先会下载 jad 文件，该文件要比 Jar 文件小得多，这就节省了大量的时间和宝贵的带宽，设备这时会根据 JAD 文件来显示 MIDlet suite 的内容，以供用户决定是否下载该 MIDlet suite。

2　MDllet

MIDP 规范定义了 MIDlet 的执行环境，在同一 MIDlet suite 中的所有 MIDlet 共享相同的执行环境，MIDlet suite 中的任一 MIDlet 都可与同一 MIDlet suite 中的其他 MIDlet 交互。在 MIDlet 的执行环境中，MIDlet 可以访问的内容为：

- 实现 CLDC 和 MIDP 的类以及它们的本地代码；
- MIDlet Suite 即 JAR 文件中的类；
- MIDlet Suite 中的资源文件（JAR 文件中所有不是类文件的文件，如图片和文本文件以及清单文件），可以通过 java.lang.Class.getResouceAsStream()方法来获得；
- 描述文件（即 JAD 文件）通过 javax.microedition.midlet.MIDlet.getAppPorperty()方法来访问；
- 通过 RMS 的 Record store 存储在永续存储介质中的内容。

图 14.16 展示了这种关系。

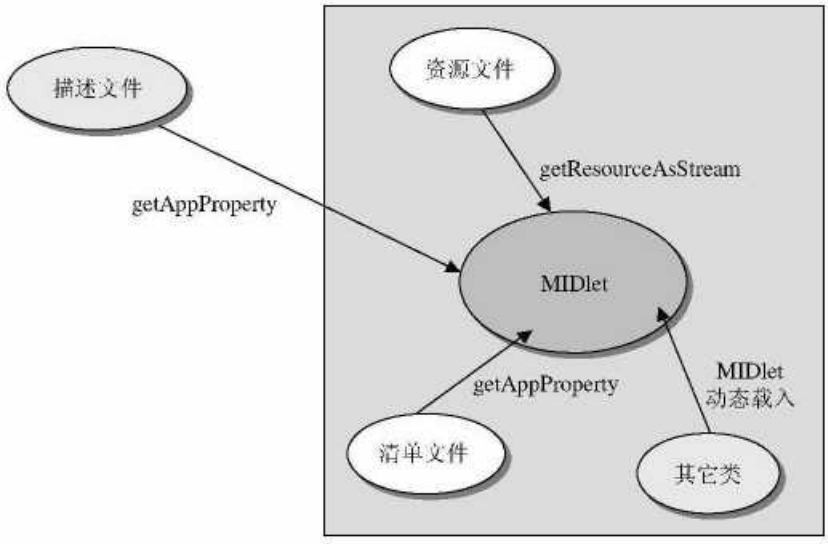


图 14.16　MIDlet 可以访问的内容

3

（1）安装英文版 J2MEWTK

有关更详细的安装说明，请参考 J2ME Wireless Toolkit 用户指南和发行说明。如果已安装有 J2ME Wireless Toolkit 的以前版本，则应在安装此版本之前将其卸载。在 Microsoft Windows 操作环境下的安装程序文件名为 j2me_wireless_toolkit-1_0_4_01-bin-win.exe。Solaris 操作环境下是自解压安装程序 j2me_wireless_toolkit-1_0_4_01-solsparc.bin。Linux 操作环境下是自解压安装程序 j2me_wireless_toolkit-1_0_4_01-linuxi386.bin。下面我们只介绍 Windows 下的安装。

双击安装文件 j2me_wireless_toolkit-1_0_4_01-bin-win.exe 来执行它。出现的安装界面如图 14.17 所示。

图 14.17　J2MEWTK 的安装欢迎界面

在版权等信息之后，安装程序会自动搜索 JDK 的安装路径，用户也可以手工来指明。如图 14.18 所示。

接着是指定安装的位置，系统默认的位置是 c:\wtk104 目录。表 14.4 是安装目录中主要的文件和目录。要注意的是，目录名不能包含有空格，否则运行会出错。

（2）安装中文包

J2ME Wireless Toolkit 发行版 1.0.4 已进行了日文、繁体中文和简体中文的本地化。本地化的内容包括：

① KToolbar/GUI 消息；

② 存储示例应用程序；

③ 用户指南文档；

④ 基本定制指南文档；

⑤ 发行说明。

用户可以从 http://java.sun.com/j2me 上下载 J2ME Wireless Toolkit 的中文本地化版本，文件

名为 j2me_wireless_toolkit-1_0_4_01-zh.zip。然后转至安装了英文版的 Wireless Toolkit 1.0.4 的目录下，将已下载的本地化压缩文件解压到 WTK104 目录。确保文件是带着目录路径解压的。

图 14.18　J2MEWTK 所需的 JDK 的路径

表 14.4　　**安装目录中的文件**

文件或目录	描　　述
BinaryLicense.html	许可证协议
BinaryReleaseNotes.html	发行说明
index.html	指向工具包文档的索引
appdb\	包含数据库文件的目录，如 RMS 文件和 ME 密钥数据库文件
apps\	目录，包含演示应用程序和由 KtoolBar 创建的其他应用程序
bin\	此工具的批处理文件和可执行文件
docs\	包含用户和 API 文档（包括本指南）的目录
lib\midpapi.zip	包含 CLDC 和 MIDP API 类的归档。应用程序源文件编译期间和应用程序类的字节代码预校验期间使用这些文件
sessions\	缺省目录，包含配置文件、内存监视文件和网络监视会话文件
wktlib\devices\	目录，包含由仿真器模拟的设备的属性文件

4　　KToolbar

KToolBar 是开发 MIDlet 套件的最小开发环境。KtoolBar 有以下功能：

① 创建一个新项目或打开一个现有项目；

② 生成、运行和调试用户的 MIDlet；

③ 微调用户的 MIDlet 应用程序；

④ 将用户的项目文件打包；

⑤ 修改 MIDlet 套件的属性。

从 Microsoft Windows“开始”菜单选择“程序”—>J2ME Wireless Toolkit 1.0.4—>Ktoolbar。KToolbar 启动后的界面如图 14.19 所示。

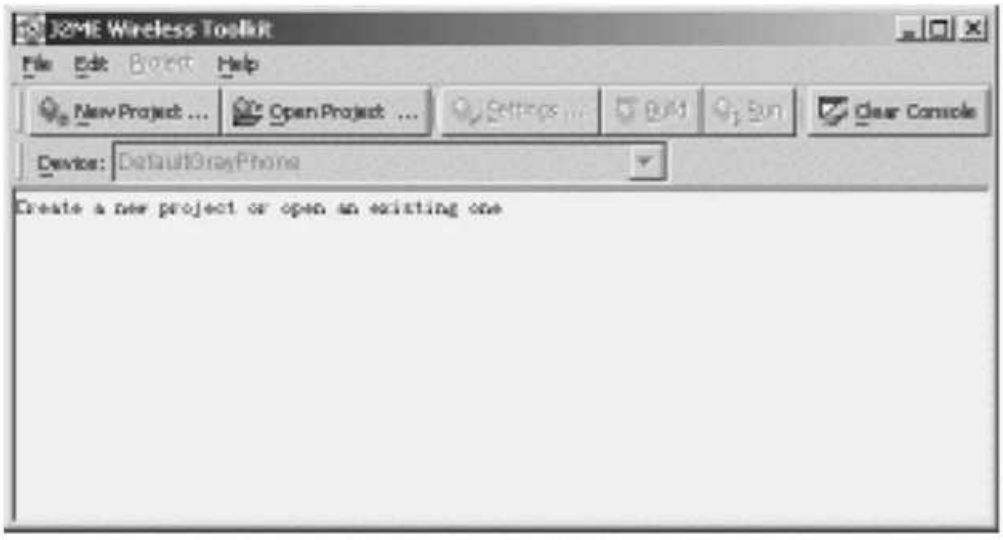


图 14.19 KToolbar 的界面

5

（1）从菜单中选择“File”—>“New Project”或单击工具栏上的“New Project”。出现“New Project”对话框。

（2）在“项目名称”字段中键入项目名称，在“MIDlet 类名称”字段中键入主 MIDlet 类的名称。例如，我们的项目称为 HelloWorld。

（3）单击“创建项目”。主窗口的原标题后面会追加上项目名称新项目的名称，如图 14.20 所示。

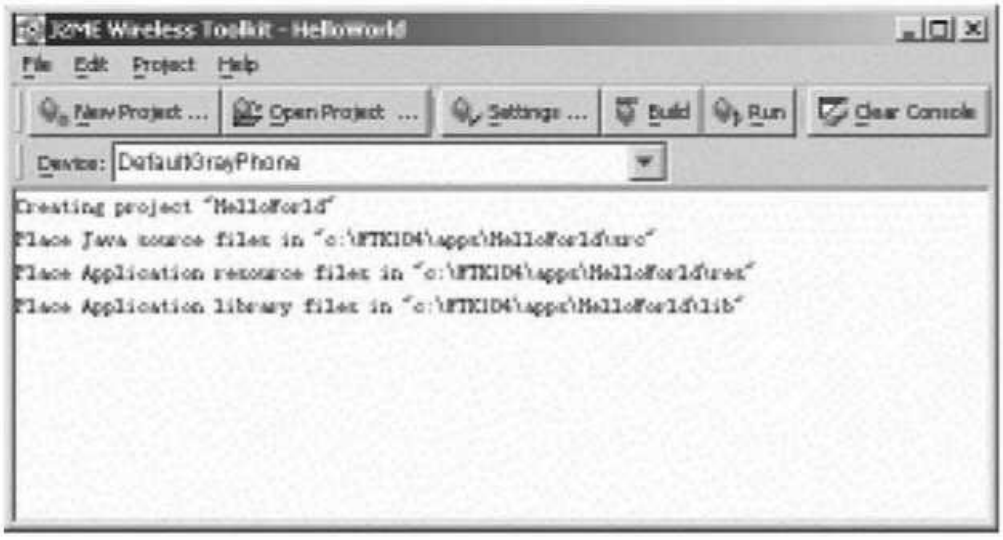


图 14.20 创建一个项目之后的控制台输出

控制台同时还会指示要将源、资源和程序库放置到何处。项目文件位于 Wireless Toolkit 安装目录中的项目子目录下。表 14.5 显示文件在项目目录{project.name}内的组织方式。

表 14.5　　项目文件组织

目　录	描　述
{j2mewtk.dir}\apps\{project.name}	包含项目的所有源、资源和二进制文件
{j2mewtk.dir}\apps\{project.name}\src	包含所有源文件
{j2mewtk.dir}\apps\{project.name}\res	包含所有资源文件
{j2mewtk.dir}\apps\{project.name}\bin	包含 JAR、JAD 和未打包的标明文件
{j2mewtk.dir}\apps\{project.name}\lib	包含特定项目的外部类程序库（JAR 或 ZIP 格式）
{j2mewtk.dir}\apps\lib	包含所有 KtoolBar 项目的外部类程序库（JAR 或 ZIP 格式）

6　　MIDlet

要编辑 MIDlet 套件的属性，请使用“项目设置”对话框，如图 14.21 所示。要显示此对话框，请从菜单中选择“Project”—>“Settings”，或单击工具栏上的“Settings”按钮。

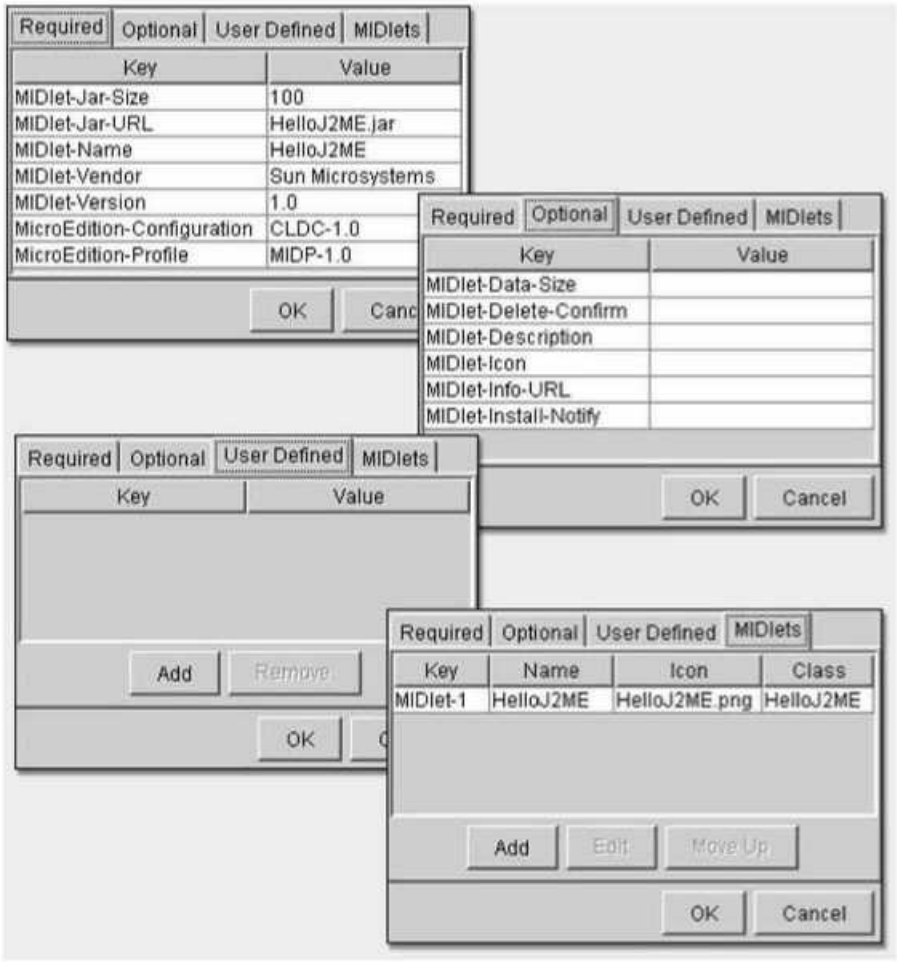


图 14.21　设置 MIDlet 属性

7 HelloWorld

我们从最基本的 HelloWorld 程序开始 MIDlet 程序的开发，HelloWorld 应用程序会在 MIDP 设备的显示屏上显示"Hello World!"和 Exit 按钮，单击“Exit”按钮会终止应用程序。

首先，在刚刚建立的应用程序目录下，如 C:\WTK104\apps\HelloWorld\src 下，新建文件 HelloWorld.java，然后用文本编辑器编辑该文件。

【例 14.1】

```
/*
* HelloWorld.java
*   一个“HelloWorld”的小例子，在屏幕上显示“HelloWorld”和 Exit 按纽
 */
import javax.microedition.MIDlet.*;
import javax.microedition.lcdui.*;
/*
 * 由于 HelloWorld 类是一个 MIDP 应用程序，它扩展了 MIDlet。
* 它也实现 CommandListener 接口来处理事件
*/
public class HelloWorld extends MIDlet implements CommandListener
{
    private Form form;
    /*
* 缺省构造程序，创建一个新表单，在上面初始化控件，然后显示出来。
*/
    private Form form;
    public HelloWorld()
    {
        // Create a new form on which to display our text
        form = new Form("HelloWorld");
        // Add the text "Hello World!" to the form
        form.append("Hello World!");
        // Add a command button labeled "Exit"
         form.addCommand( new Command( "Exit", Command.EXIT, 1 ) );
         // Register this object as a commandListener
         form.setCommandListener( this );
     }
/*
* 调用 startApp() 方法启动应用程序与小应用程序的启动方法很像。
* 在 MIDle 的一次执行中它可能会被调用多次。
* 如果 MIDlet 暂停，pauseApp() 将会被调用。要重新启动 MIDlet。
* 需调用 startApp()。仅需执行一次的主初始化代码应该放置在构造程序中。
```

```
*/
    public void startApp() {
        Display display = Display.getDisplay(this);
        display.setCurrent(props);
}
/*
* pauseApp() 被调用使得 MIDlet 处于暂停状态。
* 在此应用程序中，当进入暂停状态时，没执行任何操作；
* 但是仍然需要在 MIDlet 中实现 pauseApp()方法，
* 因为它是父 MIDlet 类中的抽象方法。
*/
public void pauseApp() { }
/*
* destroyApp() 被调用，破坏了 MIDlet 并使其处于销毁状态。
*/
public void destroyApp(boolean unconditional)
{
    form = null;//释放表单
}
/*
* commandAction()方法是事件处理程序，当单“Exit”按钮时发生的动作。
* 这里，它破坏了应用程序并通知应用程序管理软件 MIDlet 已经完成。
*/
public void commandAction(Command c, Displayable s)
{
    if (c == exitCommand)
        {
        destroyApp(true); // Destroy this MIDlet
        notifyDestroyed();
        }
    }
}
```

8 MIDlet

KToolbar 依次编译和预校验源代码。要编译和预校验源代码：选择“Project”—>“Build”或单击工具栏上的“Build”。如图 14.22 所示。

9

要在仿真器中使用 Ktoolbar 运行当前的 MIDlet 套件。

（1）使用“设备”菜单选择要模拟的设备，会列表显示适用于装入的应用程序的设备，如图 14.23 所示。

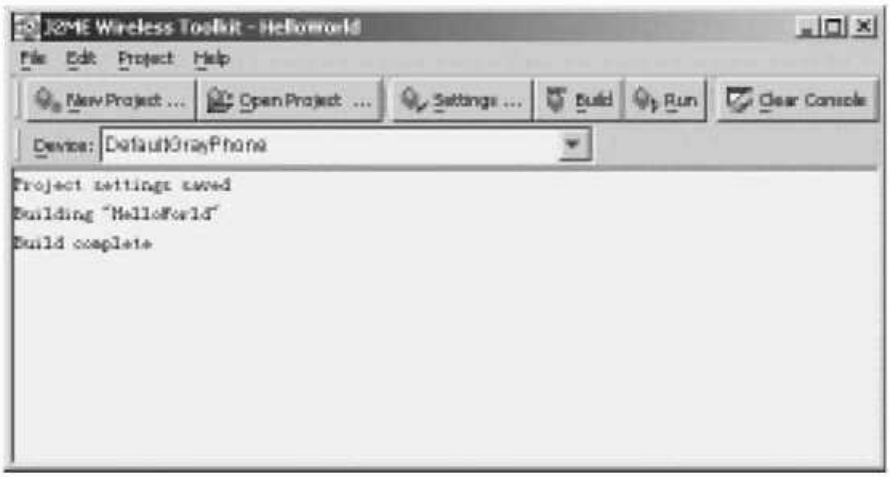


图 14.22 编译 MIDlet

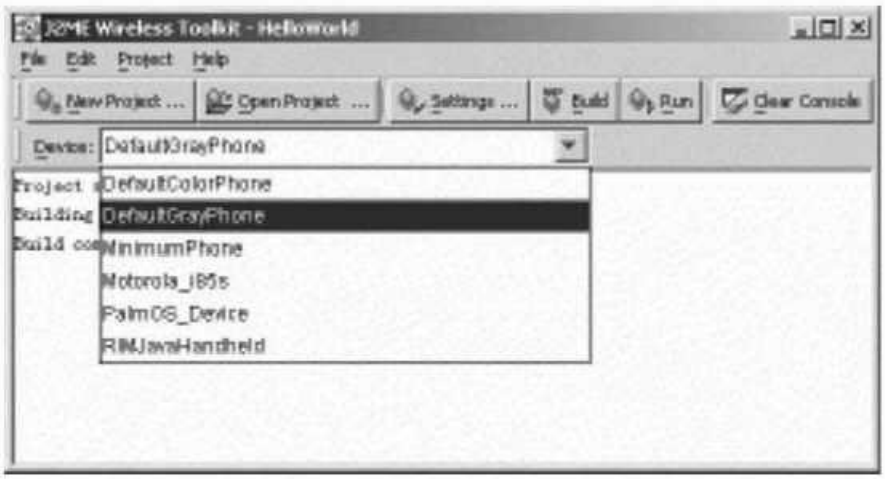


图 14.23 选择模拟设备

（2）选择 Ktoolbar 中的“项目”—>“运行”或单击其工具栏上的“运行”。出现仿真器，运行用户的 MIDlet 套件。控制台会同时显示 MIDlet 套件执行时系统和跟踪的输出。程序运行结果如图 14.24 所示。

图 14.24 程序的输出结果

14.4.3 GUI

在 HelloWorld 程序中，我们已经初步了解了 J2ME 界面开发的基本知识，J2ME 的 GUI 开发主要用到的是 javax.microedition.lcdui 包，它包含了 MIDP 应用程序要采用的用户界面元素。移动设备的用户界面(User interface,UI)与我们日常熟悉的 PC 机 UI 有很大不同，移动设备的显示范围相对要小的多，而且输出设备没有鼠标和键盘。对于 CLDC 来说，其本身并没有包含任何的 UI 包，这些 UI 包被定义在 MIDP 中。MIDP 中定义的包是专门针对移动设备而重新设计的。MIDP 之所以不支持已有的 GUI 类（如 AWT 和 Swing），原因是主要是因为 AWT 原本是为台式机设计的，而且 AWT 的许多功能对于移动设备并不必要，例如，AWT 提供了大量的功能来支持窗口管理，然而对于显示尺寸受限的手持设备来说，伸缩 resizing 窗口是不切实际的。因此，AWT 中的窗口与布局 layout 管理器对于手持设备来说显得很累赘。而 AWT 的事件处理模式，对 CPU 能力与内存大小都有限的移动设备来说更是不堪重荷。基于上述原因，MIDP 放弃了的即有 GUI 类的支持，针对移动设备而重新定义了新的 UI 包，即 javax.microedition.lcdui 包。

1 MIDP UI

MIDP 的 lcdui 包中包含了 high-level 与 low-level 这两种类型的 API，而且针对这两者分别定义了各自的事件处理模式。

（1）high-level API 面向于那些非常注重移动信息设备间可移植性（portability）的应用程序。为了达到这种可移植性，High-level API 采用了更高级的抽象。因此，对具体 GUI 可做的控制很有限，开发者不能自定义这些 high-level 组件的可视化外观（如形状、颜色、字体等）。注意，所有实现了 High-level API 的类都派生于 javax.microedition.lcdui.Screen 抽象类，其作用是提供传统的、Windows 风格的 GUI 控件。HelloWorld 程序所采用的 Form 对象就是 Screen 类的派生，其中包含和显示 GUI 控件。其他 Screen 组件还包括 Alert 对话框和显示多组选项的 List 以及容纳多行条目的 TextBox 等。

（2）low-level API 面向的是那些需要精确定位和控制图像元素以及需要获得 low-level 输入事件的应用程序。通过该 API，开发者可以全面控制 display 上的显示。访问 low-level API 的 MIDlet 不确保是可移植的，这是因为该 API 中提供了某些机制来访问一些底层细节，而这些细节可能与某种具体设备密切相关，这些就丧失了一部分的可移植性。这种 API 由 javax.microedition.lcdui.Canvas 与 javax.microedition.lcdui.Graphics 这两个类实现。

MIDP GUI 模型的核心是屏幕（screen），屏幕在 MIDP 规范中抽象为一个封装了设备特有的图形显示和用户输入的对象。而每个 MIDlet 会有惟一的一个 display 对象 javax.microedition.lcdui.Display，它管理着屏幕的显示。当调用 Display 对象的 setCurrent()方法时，就会显示屏幕。一个应用程序可以有多个屏幕，而在一个 Display 对象中同一时间只能显示一个屏幕。屏幕只是 MIDP 规范中定义的一个抽象，从开发者的角度来说，具体指的就是 displaybale 对象。共有下述两种类型的 Dispalyable 对象。

① Screen：封装高级的 UI 对象，如 Alert、List、TextBox、Form 等 UI 组件。

② Canvas：允许开发者自己定义处理图像和用户输入的低级对象。

全体图形控件都是由 Displayable 对象管理的，每一个应用程序都会访问这一对象的单一、私有实例。该实例可以通过静态的 Display.getDisplay 方法获得，该方法通常会把指向该

实例的引用保存在一个成员变量里，HelloWorld 在其构造器中就是这样做的。除了为特定屏幕元素设置焦点（setCurrent）和获取元素焦点的方法（getCurrent）之外，Display 还有一些用于获得设备显示能力信息的方法，比如是否显示彩色的（isColor）和支持显示色彩数量的（numColors）等方法。

MIDP 的 UI 类之间的层次结构如图 14.25 所示。

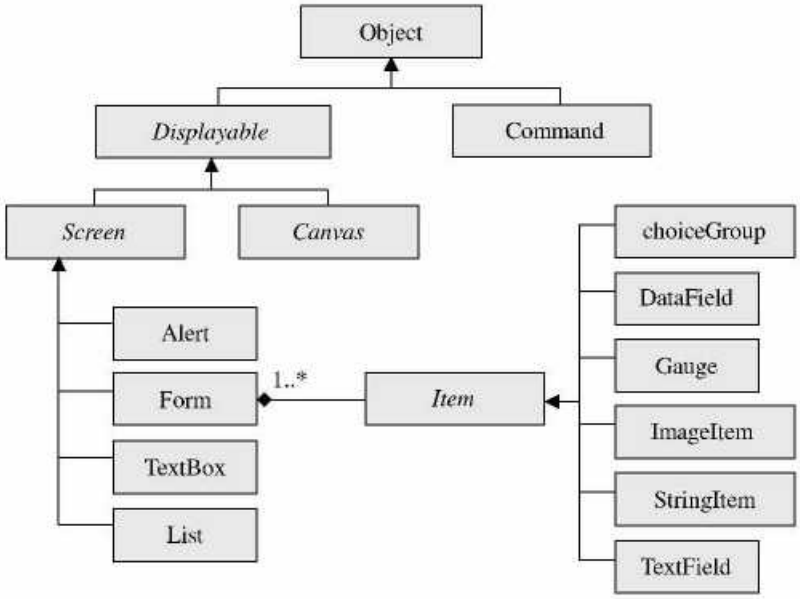


图 14.25 MIDP UI 库的类层次图

MIDP 的 GUI 类都包含在 javax.microedition.lcdui 包中，表 14.6 是一些最常见的 UI 元素的作用。

表 14.6 常见 UI 元素的作用

Alert	用于在屏幕上向用户显示关于异常情况或错误的信息
Choice	用于实现从既定数量的选项中进行选择
ChoiceGroup	提供一组相关选项
Form	作为其他 UI 元素的容器
List	提供一个选项列表
StringItem	充当只显（display-only）字符串之用
TextBox	是允许用户输入和编辑文本的屏幕显示
TextField	允许用户输入和编辑文本。多个 TextField 可放到一个 Form 中
DateField	是一个可编辑的组件，用于表示日期和时间信息。DateField 可以放到 Form 中
Ticker	用于文本的可滚动显示。

UI 元素的完整列表可在 MID Profile API 文档中找到，该文档随 J2ME Wireless Toolkit 一起提供，在\docs\api\目录下。

2　MIDP GUI

MIDP GUI　　Display
display　　setCurrent()　　Displayable
Display　　MIDlet
Display

通过给 Display 对象的静态方法 getDisplay()提供一个 MIDlet 的引用（reference），该 MIDlet 就获得了 Display 对象的实例，通常会在 MIDlet 的 startApp()方法中进行这一过程：

```
...
public void startApp( )
{
    display = Display.getDisplay(this);
}
...
```

然后通过把可以把 Displayable 对象传递给 Display 的 setCurrent()方法，来显示该 Dispalyable 对象：

```
public void setCurrent(Displayable d);
public void setCurrent(Alert alert,Displayable nextDisplayable);
```

同时 Display 对象还提供了 getCurrent()，可以用来返回当前被显示的 Displayable 对象的引用。

```
public Displayable getCurrent();
```

3　　Command　　Command

由于 MIDP 其 UI 需要适应数量广泛、特性可能互不相同的目标设备，所以 MIDP 需要高度抽象其输入机制。也就是说，在理想情况下，开发者不需要考虑设备的特殊性，例如键的数量与位置、键的绑定等。MIDP 定义了抽象的 Command 机制，而具体的实现根据与该设备相适合的机制来提供支持。对于开发者来说，Command 对象就类似于普通 GUI 编程的 button，可以设置其标题（如 OK、cancel、goback 等）。当用户调用该 Command 时，应用程序会做出适当的响应。而具体 Command 如何在屏幕上显示，以及如何与具体设备的按键等建立联系是具体实现的责任，开发者不需要考虑。

Command 对象有下述 3 个参数。

① Label：command 的标题，也就是 Command 所显示的内容，如 OK 等。

② CommandType：这个值用来表示 Command 的类型，大多数的设备都会有一个标准的键盘及其功能的映射方法，会根据这个值来映射 Command 对应的按键以及功能。例如，如果 CommnadType 被设定为 BACK，具体实现会将该 command 映射到设备的“goback”键和这个键对应的标准的“回退”操作上。

③ Priority：Command 的优先级。

下面是一个显示“OK”的 Command：

```
Command c =new Command(“OK”,Command.OK,0);
```

Command 的事件处理模式采取的是标准的 Listener 模式，需要使用 Displayable 对象的 setCommandListener()方法来注册 Command 的 Listener：

```
public void setCommandListener(CommandListener l)
```

这样 Command 事件就传送到 MIDlet 的 CommandListener()接口，应用通过实现 commandAction()来处理相应的 Command 事件：

```
public void commandAction(Command c, Displayable d);
```

例 14.1 说明了这一过程。

14.4.4 记录管理系统（RMS）

MIDP 提供在移动设备上存储持久数据的支持，并且 MID 简表还特意规定，兼容的移动设备必须提供至少 8KB 的非动态内存用于数据存储，事实上，大多数的 MIDP Java 设备提供的空间比这要求要多得多，这就允许一个 MIDlet 充分利用应用程序的持久数据。同时应该注意，这种数据存储能力与标准的 Java 是有区别的，J2ME 记录管理系统(Record Managerment System，RMS)允许数据流被储存并且在一个记录基础上访问数据，由应用程序开发者把每个记录解析到字段水平，RMS 程序包内部的接口支持一个在应用程序定义的基础上的比较与检索功能。

在面向记录的方法中，J2ME RMS 由多个记录存储构成。J2ME RMS 和 MIDlet 接口连接的概貌如图 14.26 所示。

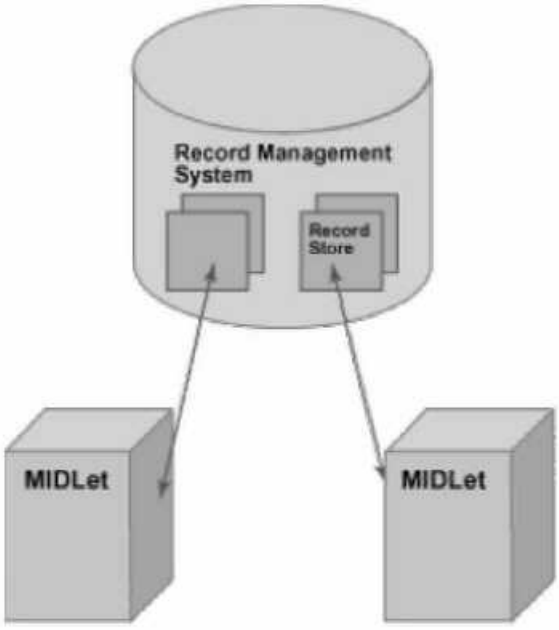


图 14.26 J2ME RMS 和 MIDlet 接口连接的概貌

可以将每个记录存储想像成一个记录集合，它将跨多个 MIDlet 调用持久存在。设备平台负责在平台正常使用的整个过程（包括重新启动、换电池等）中，尽全力维护 MIDlet 的记录存储的完整性。

记录存储在与平台相关的位置（比如非易失性设备存储器）创建，这些位置不直接公开给 MIDlet。RMS 类调用特定于平台的本机代码，这种本机代码使用标准 OS 数据管理器函数

来执行实际的数据库操作。

记录存储实现确保所有单个的记录存储操作都是原子的、同步的以及序列化的，因此多个访问将不会出现数据毁坏。记录存储被盖上时间戳来指示它上次被修改的时间。记录存储还维护版本（version），它是一个整数，修改记录存储内容的操作每发生一次，这个数就加一。版本和时间戳对于同步目的很有用。

当 MIDlet 使用多个线程访问一个记录存储时，协调该访问是 MIDlet 的责任；如果它不能这样做，则可能出现无法意料的结果。同样，如果一个平台使用试图同时访问记录存储的多个线程执行记录存储的同步，那么对 MIDlet 及其同步引擎之间的记录存储实施排外访问是平台的责任。

1　javax.microedition.rms

通过 javax.microedition.rms 包访问 J2ME 记录管理系统，这个包包括一个类 RecordStore 和好几个有用的接口。RecordStore 由大量的记录组成，这些记录在 MIDlet 的多个调用之间跨越时保持持久性。对 RecordStore 中的记录进行比较，或者从 RecordStore 中抽取若干组记录都是 Record Comparator 和 RecordFilter 接口提供的功能。表 14.7 给出的是 rms 包的接口。

表 14.7　　rms 包的接口

接　口	描　述
RecordComparator	一个接口，定义一个比较机制，比较两个记录(以一个实现定义的方式)，看它们是否匹配或它们的相对排序次序是什么样的
RecordEnumeration	一个接口，一个双向的纪录模拟器
RecordFilter	一个接口，定义了一个过滤器用于检查一个记录，看其是否匹配(基于一种应用程序定义的标准)
RecordListener	一个监听者接口，从一个记录存储器中接受记录更改/添加/删除事件

这些接口对于实现自定义搜索和检索功能很有用，这些接口中使用的最多的就是 RecordEnumeration。这个接口从 RecordStore.enumerateRecords()方法调用中返回并且被用于遍历一组从记录存储器中返回的记录。它包含 nextRecord()、previousRecord()、numRecords()和 hasNextElement()等方法。

2　　RecordStore

RecordStore 类是开发者开发基层 RMS 的接口，信息的实际的位置和存储细节应用程序开发者其实并不知道。记录存储可以使用一种简单的命名规则来访问：名称最多可以到 32 个 Unicode 字符长度，区分大小写和必须在一个 MIDlet 套件内惟一。在一个 MIDlet 套件内的所有 MIDlet 都有读/写一个记录存储的权限，只要它们知道正确的名称。一旦这个 MIDlet 套件被从该设备上删除，所有与这个套件关联的记录存储也将被删除。

（1）打开记录存储

要打开一个记录存储，调用 javax.microedition.rms.RecordStore 的 openRecordStore()方法。

public static RecordStore openRecordStore(String recordStoreName, boolean createIfNecessary);

这个方法使用两个参数：一个是字符串，表示记录存储的名称；另一个是布尔值，如果

为真的话，就会在记录存储不存在时创建一个。我们使下列方法调用创建新的 TestRecordSet 记录存储：

```
RecordStore rs = null;
rs = RecordStore.openRecordStore("TestRecordSet", true);
```

（2）插入一条数据

记录存储被创建好后，我们可以通过调用 RecordStore.addRecord()方法来向这个记录存储添加数据。addRecord()接受 3 个参数，如表 14.8 所示。

表 14.8　　addRecord()的参数

参　数	描　述
Byte[] data	一个储存在记录中的字节数据数组，通过 java.io.ByteArrayOutputStream 和 java.io.DataOutputStream 类把数据添加到这个字节数组。
Int offset	进入这个记录第一个关联字节数据缓冲区的索引。
Int numBytes	用于记录的数据缓冲区的字节数

一旦成功完成 addRecord()调用，这个方法就返回一个整数，指定这个记录在记录存储中的标识号。

（3）更新记录

更新一条特殊记录包括获取这个记录的句柄以及设置新信息。

public int getRecord(int recordId, byte[] buffer, int offset);

返回存储在由 buffer 代表的字节数组中给定记录的数据。public byte[] getRecord(int recorded)返回由 recordId 代表的数据的副本。public void setRecord(int recordId, byte[] newData, int offset, int numBytes)在 recordId 所代表记录的位置设置新信息，新信息是以 offset 作为它的起始索引，并以 numBytes 作为它的长度的字节流（newData）。例如：

```
String newappt = "update record";
Byte data = newappt.getBytes();
Rs.setRecord(1, data, 0, data.length());
```

（4）删除记录

MIDlet 调用 deleteRecord()方法来从记录存储中删除记录。

public void deleteRecord(int recordId);

删除由 recordId 代表的记录。这个记录的 recordId 接下来不能重用。例如：

```
Rs.deleteRecord(1);
```

（5）比较记录

MIDlet 实现 RecordComparator 接口，并定义 compare (byte[] rec1, byte[] rec2)方法来比较两个候选记录。这个方法的返回值必须指示这两条记录的顺序。下例比较记录并确定相对排序：

```
Int compare (byte[] b1, byte[] b2)
{
    String s1 = new String(b1);
    String s2 = new String(b2);
```

```
        If (s1.compareTo(s2) > 0)
        Return RecordComparator.FOLLOWS;
        Else if (s1.compareTo(s2) == 0)
        Return RecordComparator.EQUIVALENT;
        Else
       Return RecordComparator.PRECEDES;
}
```

（6）关闭 RecordStore

一旦所有操作完成，对 closeRecordStore() 的调用将关闭指定名称的记录存储。当一个记录存储被关闭时，不能进行进一步的操作。

```
Rs.closeRecordStore();
```

（7）删除一个 RecordStore

通过调用 deleteRecordStore() 方法可以删除指定名称的记录存储。

```
RecordStore.deleteRecordStore("TestRecordSet ");
```

RecordStore 还支持其他数据操作方法，包括 setRecord()和 deleteRecord()。

3

下面我们构造一个基本的股票行情查看应用程序，使用它，用户就可以从一个移动设备上查看股票行情。本节中只介绍从本地设备中存储的数据中读出相关的资料，在 14.4.6 节将介绍如何使用 J2ME 网络功能从一个 Web 服务器取回数据文件。

为了从用户界面逻辑中把数据存取逻辑分开，我们要创建两个类：一个是 StockDB 类封装所有的 RMS 代码；另一个是 StockBookMIDlet 类封装 GUI 代码。StockDB 类的代码见例 14.2。

【例 14.2】

```
/*
* StockDB.java
* 演示 J2ME 中 RMS 的使用
 */
import javax.microedition.rms.*;
import java.io.*
public class StockDB
{
    private static RecordStore rs = null;
    public StockDB()
    {
        try
        {
            rs = RecordStore.openRecordStore("Stockbook", true);
        }
        catch (RecordStoreException e)
        {
```

```
            System.out.println(e);
            e.printStackTrace();
        }
    }
    public void addStock(String Name, String Stock)
    {
        ByteArrayOutputStream os = new ByteArrayOutputStream();
        DataOutputStream output = new DataOutputStream(os);
        try
        {
            output.writeUTF(Name + "," + Stock);
        }
        catch (IOException e)
        {
            System.out.println(e);
            e.printStackTrace();
        }
        byte[] b = os.toByteArray();
        try
        {
            rs.addRecord(b, 0, b.length);
        }
        catch (RecordStoreException e)
        {
            System.out.println(e);
            e.printStackTrace();
        }
    }
    public static String getName(int index)
    {
        int counter = 1;
        int commalocation = 0;
        String name = null;
        try
        {
            RecordEnumeration enumRec =  rs.enumerateRecords(null, null, false);
            while ((counter <= index) && (enumRec.hasNextElement()))
            {
                String strTemp = new String(enumRec.nextRecord());
```

```
                commalocation = strTemp.indexOf(',');
                name = strTemp.substring(2, commalocation);
                counter++;
            }
        }
        catch (Exception e)
        {
            System.out.println(e);
            e.printStackTrace();
        }
        return name;
    }
    public static String getStock(int index)
    {
        int counter = 1;
        int commalocation = 0;
        String Stock = null;
        try
        {
            RecordEnumeration enumRec =    rs.enumerateRecords(null,null, false);
            while ((counter <= index) && (enumRec.hasNextElement()))
            {
                String strTemp = new String(enumRec.nextRecord());
                commalocation = strTemp.indexOf(',');
                Stock = strTemp.substring(commalocation + 1);
                counter++;
            }
        }
        catch (Exception e)
        {
            System.out.println(e);
            e.printStackTrace();
        }
        return Stock;
    }
    public static int recordCount()
    {
        int count = 0;
        try
```

```
            {
                count = rs.getNumRecords();
            }
            catch (Exception e)
            {
                System.out.println(e);
                e.printStackTrace();
            }
            return count;
        }
    }
```

StockDB 类包含好几个 public 访问方法，它们都对来自外部调用者的访问隐藏记录存储的细节。StockDB()构造程序调用 RecordStore.openRecordStore()，我们创建了 4 个方法用于访问基层记录存储：recordCount()、getStock()、getName()和 addStock()。注意，在本例子中 addStock()方法只是在 name/Stock 字段之间放了一个逗号。同样地，getStock()和 getName()从记录存储中取回 name/Stock 字段。

StockBookMIDlet 类(见例 14.3)包含一个 StockDB 对象、一个 List GUI 组件和一个 Back 命令和 Exit 命令。使用 StockDB 类中的 addStock()方法，把地址添加到数据库中。在 startApp()方法中，使用调用 List.append()和 StockDB.addStock()方法来填充 List。这是在 commandAction()内部完成的，其结果就是创建一个新的文本框并且添加显示出来。因为 cmdBack 命令对象是使用 Command.BACK 变量创建的，当又一个元素被添加显示时，环境知道显示一个“Back”命令按钮。然后通过把显示焦点设置回 mnuMain 列表对象，处理“back”命令事件。

【例 14.3】

```
import javax.microedition.MIDlet.*;
import javax.microedition.lcdui.*;
public class StockBookMIDlet extends MIDlet implementsCommandListener
{
    Display display = null;
    List mnuMain = null;
    TextBox txtStock = null;
    static final Command cmdBack = new Command("Back", Command.BACK,0);
    static final Command cmdExit = new Command("Exit", Command.STOP,3);
    StockDB dbStock = null;
    public StockBookMIDlet()
    {
        dbStock = new StockDB();
        dbStock.addStock("HP", "$50");
        dbStock.addStock("Apple", "$100");
        dbStock.addStock("IBM", "$150");
```

```
        dbStock.addStock("Microsoft", "$200");
        dbStock.addStock("Sun", "$300");
    }
    public void startApp() throws MIDletStateChangeException
    {
        display = Display.getDisplay(this);
        mnuMain = new List("Stocks", Choice.IMPLICIT);
        int count = dbStock.recordCount();
        for (int i=0; i < count; i++) {
        mnuMain.append(dbStock.getName(i+1), null);
    }
    mnuMain.addCommand(cmdExit);
    mnuMain.setCommandListener(this);
    display.setCurrent(mnuMain);
    }
    public void pauseApp()
    {
        display = null;
    }
    public void destroyApp(boolean unconditional)
    {
        notifyDestroyed();
    }
    public void commandAction(Command c, Displayable d)
    {
        String str = c.getLabel();
        if (str.equals("Exit"))
        {
            destroyApp(true);
        }
        else if (str.equals("Back"))
        {
            display.setCurrent(mnuMain);
        }
        else
        {
            List select = (List)display.getCurrent();
            String txtSelect =
            StockDB.getName(select.getSelectedIndex() + 1) + ", "
```

```
                    +dbStock.getStock(select.getSelectedIndex() + 1);
                txtStock = new TextBox("Stock", txtSelect, 255,TextField.ANY);
                txtStock.addCommand(cmdBack);
                txtStock.setCommandListener(this);
                display.setCurrent(txtStock);
            }
        }
    }
4
```

本例示范如何使用 J2ME 记录管理系统(RMS)构造一个基本的股票信息应用程序，支持局部数据存储的能力是 J2ME 与其他无线技术（例如，WML/WMLScript）的不同之处，当然，为了把来自移动设备的数据与一个企业数据库同步还需要增加网络和输入/输出性能。

14.4.5 J2ME 网络程序设计

上节介绍了通过记录管理系统(RMS)开发本地设备数据存储，J2ME 另外一个很重要的特性就是使用 J2ME 连接结构来打开网络连接，并传送数据的能力。javax.microedition.io 包内的这个结构包括 Connection 类和好几个很有用的接口(包括 StreamConnection、ContentConnection 和 HTTPConnection)。本节讨论这个包的设计并使用 StreamConnection 和 ContentConnection 接口增强前面介绍的 StockBookMIDlet 例子的功能。

1 javax.microedition.io

如果用户有使用 J2SE java.net 包开发程序的经验，就会知道不但它使用得非常广泛，而且它还提供一些非常高级的网络性能。遗憾的是，由于设备内存的大小不同，这些高级特性就不适合有限连接设备配置 CLDC。MID 简表定义了一个 HTTPConnection 接口，用于网络上的 HTTP 访问。

2 StockBookMIDlet

本节中的例程与 14.4.5 节中股票管理系统 StockBook 例程几乎一样，14.4.5 节的例子使用的是本地的数据文件，而现在要介绍的例子是使用 J2ME 网络功能从一个储存在互联网上的文本文件中取回股票价格信息。这个文本文件名为 stockbook.txt，文件中的股票与价格使用逗号分隔。例 14.3 使用两个不同的 J2ME 接口来执行传送数据 StreamConnection 和 ContentConnection。

3 StreamConnection

StreamConnection 接口定义了一个流连接必须有的最小功能。下面将对 StockBookMIDlet 应用程序做出修改。

（1）删除 StockBookMIDlet()构造程序中的 dbStock.addStock()方法调用，这个方法调用可以删除，因为新的程序没有必要自己动手向数据库中添加数据，新的程序将使用 J2ME 的网络功能取回储存在网上的股票价格。

（2）把特定的连接代码添加到 StockDB 构造程序中。例 14.3 中的特定连接代码只是简单地通过 TCP/IP 取回地址并手动地把每支股票添加到股票簿中。

```
StreamConnection connStream = null;
```

```
InputStream inStream = null;
byte[] b = new byte[255];
String stock, name;
int commalocation = 0;
try {
        connStream=(StreamConnection)Connector.open(
"http://localhost/stockbook.txt");
        inStream = connStream.openInputStream();
        int count = inStream.read(b);
        stock = new String(b);
        stock = stock.trim();
       StringTokenizer st = new StringTokenizer(stock, "");
       while (st.hasMoreTokens())
       {
        stock = st.nextToken();
              commalocation = stock.indexOf(',');
              name = stock.substring(0, commalocation);
              stock = stock.substring(commalocation + 1);
              addStock(name, stock);
        }
}
catch (IOException e)
{
        System.out.println(e);
        e.printStackTrace();
}
```

4　　ContentConnection

ContentConnection 接口与 StreamConnection 接口作用差不多，但是它提供了更多有用的方法。其中有一个非常有用的方法是 getLength()方法，它的功能是返回内容的长度，为了使用这个方法，需要声明一个 ContentConnection 变量：

```
ContentConnection connStream = null;
```

现在将字节数组移进 try{}子句中，并改变另外 3 项：

（1）把 Connector.open()方法的输出结果强制转化成一个 ContentConnection；

（2）取得 ContentConnection 后，调用 c.getLength()方法来取回这个数据的长度；

（3）一旦取得内容的长度，就可以使用这个长度动态地创建一个字节数组。

除此之外，其他的部分与原来的程序相同。

```
connStream=(ContentConnection)Connector.open("http://localhost/stockbook.txt");
int len = (int)connStream.getLength();
byte[] b = new byte[len];
```