

三步学 JAVA

——语言基础 + 面向对象 + 应用



赵志崑 编著

序言

Java 是互联网时代最重要的程序设计语言。学习 **Java** 已成为每个从事计算机软件的人的必修课。市面上充斥着大量介绍 **Java** 程序设计的图书。

这本书的特点是把学习 **Java** 分为三大步：语言基础、面向对象和应用。每一部分包含若干知识点，每个知识点既独立成章，前后章节之间又环环相扣循序渐进。语言基础部分主要介绍 **Java** 的背景、开发环境、程序基本概念、基本语句。基本语句又分为数据类型、运算符和流程控制三类来介绍。面向对象部分按三个层次展开，第一层次是类和对象等基本概念，第二层次是封装、继承、多态三个核心机制，第三层次是其它特性。应用部分介绍实现各种功能的机制，包括输入输出流、异常处理、集合框架、图形用户界面、事件处理、绘图、多线程、网络通信、序列化和数据库。

本书适合从零基础开始学习 **Java**。当然，如果您有一定的基础，比如 **C**，那么学起来会更快。如果您对程序设计有一定了解，可以跳过第三章“程序的基本概念”，不过如果您能再读一遍的话，应该也有新的收获。如果您对面向对象技术很熟悉，那么在阅读第二部分时可以跳过每一章前面的理论部分，直接看后面的 **Java** 语言如何实现面向对象技术。

本书是我十多年 **Java** 教学工作的总结，也是我在二十多年编程经验基础上对程序设计的理解。我从 90 年第一次接触编程，那时我刚上高一，从此喜欢上了编程一发不可收拾。后来本科硕士博士一气读完，专业都是计算机软件。**2003** 年博士毕业，留在中国科学院研究生院（现在改名为中国科学院研究生大学，之前还曾叫中国科技大学研究生院）工作，从那时起就开始讲授 **Java** 程序设计，一直到现在。

从我自己学习 **Java** 的经验来看，分成语言基础、面向对象和应用三大步是非常合理的。我在学 **Java** 之前接触过 **Basic**、**Fortran**、**Pascal**、**C** 等多种语言，但主要用 **C**。对面向对象技术只有一知半解，只知道基本概念，不知道细节。我就是按照这三大步的方式来学习 **Java** 的，半年左右就能够用 **Java** 完成比较大的项目了。通过 **Java** 学习，对面向对象技术也有了更深的理解。

在我教 **Java** 的过程中，也是按三大步的顺序来教授的，效果还不错。但有一个问题一直困扰着我，就是没有合适的教材。介绍 **Java** 的书虽然很多，但是没有和我的三大步的思想完全一致的，我用过几本书做教材，都不太满意。于是，我花了很大精力把我上课的讲稿整理出来，形成了本书。因此，这本书既反映了我自己学习 **Java** 的过程，也体现了我讲授 **Java** 的思路。希望我的经验能够对您学习 **Java** 起到帮助。

赵志崑
2014 年 1 月

目录

第一部分 Java 语言基础.....	5
第一章 Java 简介.....	5
1.1 Java 的诞生与发展.....	5
1.2 Java 相关的概念.....	6
1.3 Java 语言的特点.....	7
1.4 Java 语言流行的原因.....	8
1.5 Java 与 C++和 C#.....	9
第二章 开发环境.....	10
2.1 Java SDK 的安装.....	10
2.2 第一个 Java 程序.....	13
2.3 帮助文档.....	16
第三章 程序的基本概念.....	17
3.1 程序的组成.....	17
3.2 标识符.....	17
3.3 关键字.....	18
3.4 变量和值.....	19
3.5 表达式和运算符.....	19
3.6 子程序和函数.....	20
3.7 Java 程序的基本结构.....	20
第四章 基本语句.....	21
4.1 数据类型.....	21
4.2 运算符.....	27
4.3 流程控制语句.....	29
4.4 基本输入输出方法.....	34
第二部分 面向对象机制.....	36
第五章 类和对象.....	36
5.1 类和对象的概念.....	36
5.2 类.....	38
5.3 对象.....	39
5.4 引用与参数传递.....	42
5.5 实例变量和局部变量.....	44
5.6 类的管理.....	45
第六章 封装、继承、多态.....	50
6.1 封装.....	50
6.2 继承.....	51
6.3 多态.....	57
第七章 其它特性.....	60
7.1 final 关键字.....	60
7.2 类的静态成员.....	61
7.3 接口.....	65
7.4 抽象类.....	66
7.5 内部类.....	67

7.6 覆盖的规则.....	68
7.7 instanceof 运算符与引用的类型转换	69
7.8 包装类.....	70
7.9 降级.....	71
第八章 数组.....	72
8.1 一维数组.....	72
8.2 多维数组.....	75
第九章 常用工具类.....	77
9.1 String 类	77
9.2 Math 类.....	78
9.3 Class 类.....	80
9.4 Random 类	80
第三部分 应用.....	82
第十章 流与输入输出.....	82
10.1 流的概念.....	82
10.2 流相关的类.....	82
10.3 数据读写方法.....	84
10.4 标准流.....	87
10.5 示例——抽奖器.....	88
10.6 抽奖器的改进.....	92
第十一章 异常处理.....	96
11.1 为什么需要异常处理.....	96
11.2 异常处理流程.....	97
11.3 异常处理语句.....	98
11.4 异常类.....	104
11.5 异常处理原则.....	106
11.6 异常处理示例.....	107
第十二章 集合框架.....	111
12.1 集合框架简介.....	111
12.2 集合框架中的接口和类.....	113
12.3 List——列表	115
12.4 Set——集合	121
12.5 用列表改进抽奖器.....	125
12.6 常用算法.....	127
12.7 Comparable 接口与 Comparator 接口.....	128
第十三章 窗口程序.....	133
13.1 概述.....	133
13.2 创建窗口.....	134
13.3 设置窗口属性.....	135
13.4 为窗口添加菜单.....	137
13.5 向窗口中添加组件.....	141
13.6 布局管理器.....	144
13.7 面板.....	153
13.8 工具栏.....	158

13.9 组件边界.....	159
13.10 对话框.....	161
第十四章 事件处理.....	168
14.1 事件监听机制.....	168
14.2 事件监听器类的各种写法.....	171
14.3 窗口事件.....	173
14.4 事件类.....	176
14.5 键盘事件.....	177
14.6 鼠标事件.....	181
14.7 定时器事件.....	185
第十五章 绘图.....	187
15.1 绘图机制.....	187
15.2 绘制几何图形.....	188
15.3 颜色与字体.....	194
15.4 绘制图像.....	200
15.5 其它绘图功能.....	204
第十六章 多线程.....	208
16.1 线程的概念.....	208
16.2 创建线程.....	208
16.3 线程的状态.....	210
16.4 事件处理线程.....	213
16.5 线程与 swing 组件	216
16.6 线程的优先级.....	221
16.7 同步方法.....	222
16.8 wait 和 notify 机制	230
16.9 死锁.....	233
第十七章 网络通信.....	234
17.1 Socket 简介	234
17.2 有连接的 TCP 通信.....	235
17.3 简易聊天室.....	244
17.4 无连接的 UDP 通信.....	249
17.5 MVC 模式	252
第十八章 对象序列化.....	253
18.1 格式化数据输入输出.....	253
18.2 对象序列化.....	258
18.3 对象序列化的其它应用.....	265
第十九章 数据库.....	269
19.1 数据库系统与 JDBC.....	269
19.2 MySQL 数据库安装	270
19.3 建立连接.....	275
19.4 执行 SQL 语句	276
19.5 处理结果集.....	278
19.6 事务.....	281

第一部分 Java 语言基础

第一章 Java 简介

Java 语言是由 Sun 公司创造和发展的一种程序设计语言。可以说，Java 和 C++ 是目前软件开发中最重要的两门语言。Java 语言有两个最重要的特点，完全面向对象和跨平台。这两个特点使得 Java 比 C++ 简单易学，并且可以“一次开发，随处运行”，所以使用 Java 语言的 Java 平台，应用领域非常广泛，从桌面应用到 Web 应用，从嵌入式应用到移动应用，以及现在流行的 Web 服务，都可以使用 Java 开发。

据统计，到 2012 年，有 11 亿个桌面安装了 Java 平台；每年 Java 运行环境 9.3 亿次下载；有 30 亿部移动电话支持 Java 平台，是 Apple 和 Android 手机数量总和的 31 倍多；Java 为机顶盒、打印机、网络照相机、游戏、汽车导航系统、彩票终端、医疗设备、停车收费站等提供了强大功能。除了 Java 平台以外，很多非常流行的其它的平台也使用 Java 语言或类 Java 语言。比如，安卓手机操作系统使用 Java 语言，Flash 使用类 Java 的 Action Script 语言，HTML5 使用类似的 JavaScript 语言。这也说明了 Java 语言是目前最流行的程序设计语言之一。迄今为止，Java 平台已吸引了超过 900 万名软件开发人员。它在各个重要的行业部门得到了广泛的应用，出现在各种各样的设备、计算机和网络中。

1.1 Java 的诞生与发展

Java 最初的设想是 1991 年提出的。当时，Patrick Naughton 和 James Gosling 领导 Sun 的一个小组，着手设计一个用于消费类电子设备的计算机语言，这个项目命名为 Green。这个语言要满足两个要求，小和平台无关。他们采用了类似于当时最流行的 C++ 语言的语法，以便于学习，同时采用了虚拟机的思想实现跨平台。这个语言最初起名 Oak，源于设计者窗外的一棵橡树。后来改为 Java，Java 是印度尼西亚的一个岛，中国称为“爪哇”，那里出产世界闻名的咖啡。Java 的图标就是一杯咖啡，是希望程序员像喜欢咖啡一样喜欢 Java。

1992 年，Green 项目组提交了第一个产品。接着，项目组开始推销 Java，但是直到 1994 年，都没有找到一个客户。一个新生事物总是需要时间和机会去被人们接受。Java 项目组意识到，需要首先开发一个产品来证明 Java 的能力。当时正赶上互联网的热潮，所以项目组用 Java 开发了一个浏览器，这个浏览器叫 HotJava。1995 年 5 月，HotJava 在 Sun World'95 上展出，立即引起轰动。1995 年秋，Netscape, IBM, Inprise, Microsoft 等许多大公司先后宣布支持 Java，标志着 Java 正式诞生。

Java 从诞生到现在，版本一直在不断更新，几次大的改进包括：

- 1996 年初发布 1.0，两个月后发布 1.02，但不够实用，特别是图形库，被批评为“像玩具”。
- 1998 年 12 月发布 1.2，3 天后改名为 Java2，Java2 已经不仅是一个语言和虚拟机那么简单，成为了一个平台。而且针对不同的应用领域而分为三个版本，包括 J2SE, J2EE 和

J2ME(1.0)。Java2 已经相当实用，特别是 GUI 图形工具类库，至此 Java 核心平台固定下来。Java2 是第一个广泛应用的版本

- 2005 年发布 Java2 的 1.5，然后立即改为 Java5，仍然包括 SE,EE 和 ME 三个平台。
- 2006 年发布 Java 6。
- 2009 年发布 Java7。

1.2 Java 相关的概念

Java 语言。Java 语言是一种通用的程序设计语言。程序设计语言是人和计算机之间交流用的语言。人通过程序设计语言告诉计算机去完成一项工作，将完成的方法和过程用程序设计语言写下来就是程序。Java 语言是所有 Java 平台和应用都使用的程序设计语言。

Java SDK(Software Development ToolKit)。Java SDK 是能够编译用 Java 语言写的程序并生成二进制字节码的工具包，由 Sun 发布。

Java 虚拟机。Java 虚拟机是能够解释执行 Java 二进制字节码的软件。虚拟机是 Java 字节码文件和硬件之间的翻译，所以每个硬件平台都需要有 Java 虚拟机才能运行 Java 程序。

Java 平台。Java 平台是由 Java 虚拟机、编译器和类库构成的开发和运行环境。Java 有三个平台，标准版、企业版和微小版，分别面向不同的应用环境。不同平台之间虚拟机和编译器差别并不大，主要差别体现在类库上。

Java Script。Java Script 是 90 年代出现的一种能嵌在网页中运行的脚本语言，除语法与 Java 接近外没有其他关系。

Java Applet。Java Applet 叫 Java 小程序，是用 Java 语言编写的一种运行在支持 Java 的浏览器中的特殊程序，目的是在网页中实现类似于桌面应用的效果。

Java Servlet。Java Servlet 是运行在 Web 服务器端，能提供动态内容服务的 Java 小程序。Java Servlet 通过在 Java 程序中嵌入 Html 代码实现动态网页内容生成，是 Java 企业版的基础。

JSP(Java Server Page)。JSP 是对 Java Servlet 的升级，通过在 Html 代码中嵌入 Java 代码实现动态网页内容。服务器会先将 JSP 翻译成 Java Servlet，再执行。

根据不同的使用目标，Java 平台分为三个版本：

- **Standard Edition (JavaSE, 标准版):** 开发和部署在桌面、服务器和实时环境中使用的 Java 应用程序
- **Enterprise Edition (JavaEE, 企业版):** 开发企业应用系统，主要是基于动态 Web 的系统。
- **Micro Edition (JavaME, 微小版):** 移动设备和嵌入式设备，比如：手机、PDA、电视机顶盒。目前绝大多数手机厂商和移动服务商的网上商店都支持 Java 微小版的应用。但是，现在 JavaME 的新应用越来越少了，原因是安卓的崛起。安卓也是使用 Java 语言，虽然适用的机型比 JavaME 少，但是功能比 JavaME 强大的多。

下面这幅图说明了 Java 语言在 Java 体系中所处的位置。



图 1-1 Sun 的 Java 培训课程体系

Java 平台的核心是 Java 语言和 Java 虚拟机，学习 Java 语言通常以标准版为基础。学了 Java 语言才能去学习三个具体平台上的应用如何开发。本书以 Java 标准版为基础介绍 Java 语言。

1.3 Java 语言的特点

Java 语言有以下特点：

- **简单：**简单主要体现在几个方面，一是语法简单，类似 C++，但比 C++ 简单，易于学习；二是代码小，最小的可运行的基础解释器和类只有 40KB，可用于内存空间比较小的嵌入式环境；三是对象自动回收机制使程序员不用关心对象的释放了。
- **面向对象：**Java 是一个完全面向对象的语言，程序中一切皆是对象，不存在全局的变量和函数。完全面向对象使 Java 代码更加模块化，代码的存放也更加灵活。
- **可移植性、中立体系结构：**一致的数据类型大小，所有数据类型的大小都是固定的，不会随硬件平台变化。一致的界面风格，创建的图形用户界面在不同操作系统上风格一样，不会有变化。
- **解释型：**Java 源文件编译后生成字节码文件，字节码文件在虚拟机上解释执行。Java 实际上是一种先编译后解释的语言，在实现跨平台的前提下，效率也并不低。Java 的效率比纯解释型语言要高很多，比纯编译型语言要低一些。
- **动态性：**Java 的类加载是动态的，每个类的字节码单独保存在一个文件中，需要时才从文件加载到内存，不需要静态链接。这其实是一种完全动态的链接，实现了面向对象的组件模型，使用起来比 Windows 采用的 dll（动态链接库）要简单得多。Java 的面向对象的组件模型使设计插件型的程序非常简单。
- **反射性：**反射指由类的字节码文件可以得到类的信息，包括名字、属性和方法。反射可以理解为类似于反编译的过程，能使得程序的灵活性大大提高。
- **分布式：**Java 设计分布式程序是非常方便地，第一，它访问互联网上的 URL 资源就像访问本地资源一样方便；第二，它易于使用 Socket 进行 TCP/IP 编程；第三，它提供远程方法调用机制 RMI，调用远程对象的方法就像调用本地对象的方法一样简单。
- **多线程：**Java 提供的多线程编程接口很简单。
- **高性能：**Java 还有即时编译（JIT）机制，能够将字节码文件编译成特定平台的程序，被硬件直接执行，从而提高程序的性能。
- **健壮性：**Java 提供了很多机制保证程序的健壮性。编译时进行早期错误检查，执行时进行后期动态检查。对象自动回收机制可以有效防止内存泄漏。

- 安全性: Java 被称为最安全的语言, 提供了异常处理机制和安全机制, 可以防止堆栈溢出、访问进程空间外内存、Applet 访问本地文件等。

1.4 Java 语言流行的原因

Java 在程序设计语言家族中处于最新的位置。

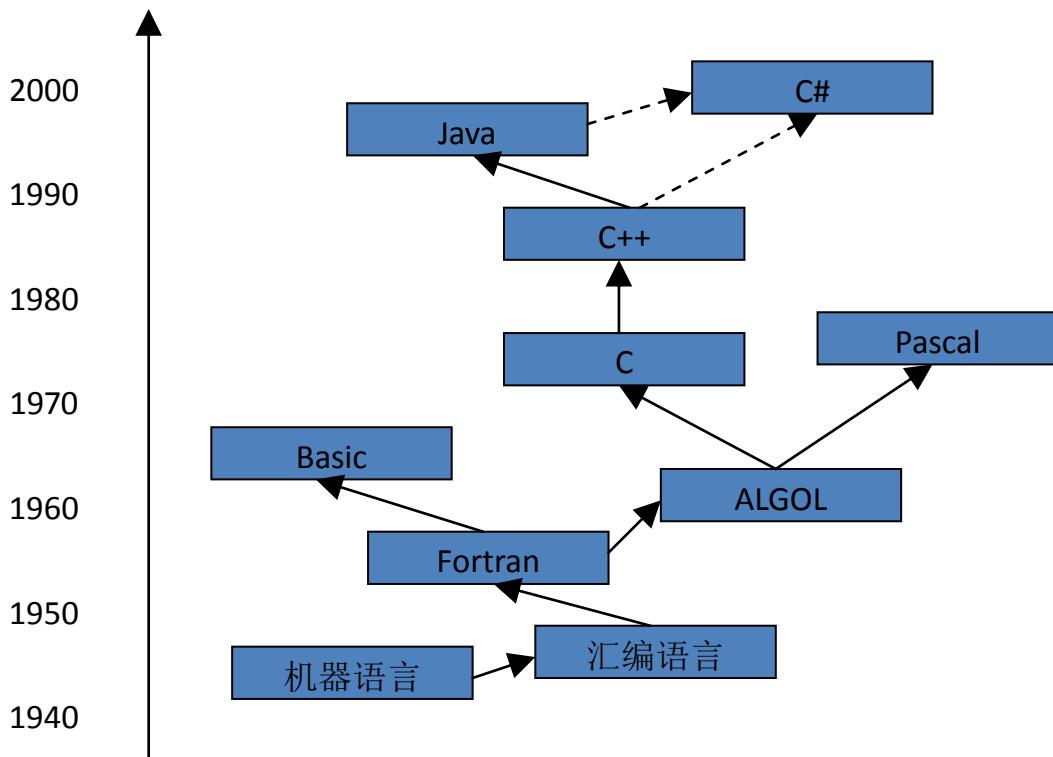


图 1-2 主要通用程序设计语言的发展历史

Java 语言之所以能够快速流行起来, 主要有以下几个原因:

首先, 它符合程序设计语言的发展趋势。程序设计语言经历了机器语言、汇编语言、高级语言, 程序设计思想也经历了面向过程、面向模块、面向对象, 其总的的趋势是越来越接近人的自然语言, 也就使得程序设计越来越接近人的思维方式。这样, 在人与计算机交流的过程中, 机器做得翻译工作越来越多, 人可以不用再去适应机器的工作方式, 从而使软件生产的效率越来越高。

其次, Java 有助于解决软件危机问题。软件危机指的是对软件需求的增长速度远远超高软件的生产速度。Java 的出现, 从两个方面有助于软件危机问题的解决。一是 Java 的跨平台性, 使得同样的程序, 只用开发一次, 就能在不同平台上运行, 成倍地减轻了软件开发的工作量。二是 Java 语言简单易学, 可以使大量的人员投入到软件开发中来, 提高了软件的开发总量。

再者, Java 恰好满足了互联网的需要。互联网的出现改变了软件的运行环境, 需要一种合适的程序设计语言。Java 恰好在此时出现, 并满足了互联网的需要, 比如其跨平台、分布式、安全性等特性。所以, Java 搭上了互联网飞速发展的顺风车, 也很快被大家所接受。

1.5 Java 与 C++和 C#

Java 借鉴了 C++语法，对于变量声明、参数传递、操作符、流控制等使用和 C++相同的传统，摒弃了 C 和 C++中许多不合理的内容。为了便于 C++程序员快速熟悉 Java，两者之间的一些具体的差别列举如下：

- 全局变量：Java 中没有全局变量，也没有全局方法，连 main 函数也不是全局的，而是类的函数。
- goto 语句：Java 中不支持 goto 语句，但有受限 goto 语句 break 和 continue。
- 指针：Java 不支持指针，但对象变量实际上都是指针，称为引用。
- 数据类型的支持：Java 在不同平台上数据类型都统一。
- 类型转换：Java 有类型相容性检查。
- 结构和联合：Java 只支持类，不再区分结构体和联合体。
- 多重继承：Java 只支持类的单一继承，多重继承的功能用接口实现。
- 内存管理：Java 的垃圾回收机制自动回收无用变量的内存，程序中只用分配内存，不用再释放内存。
- 头文件：Java 不支持头文件，类似的机制是 import。
- 宏定义和预处理：Java 不支持宏定义。

C#（称为 C Sharp）是 2000 年由 Microsoft 随着其 .Net 框架一起推出的一种程序设计语言。C#的含义是 C++++，在语法上和类库结构上，与 Java 非常相似。但 C#的语法比 Java 复杂，支持 foreach 语句、goto 语句，支持指针，支持运算符重载，而且可以在 .Net 框架中和其他语言一起进行多语言编程。

第二章 开发环境

开发 Java 程序最基本的环境是 Java SDK，包括 Java 的编译器、虚拟机和类库，采用基于命令行的界面，由 Sun 发布和维护。在 Java SDK 的基础上，有一些集成开发环境，比如 NetBean、Eclipse、JBuilder、JCreator 等。集成开发环境也是利用 Java SDK 中的编译器和虚拟机来编译和运行 Java 程序，只是提供了项目管理的功能，将源文件和类文件都管理起来，方便程序员开发。

本书是以 Java SDK 为开发环境的。这是因为 Java SDK 是最基本的开发环境，有助于去理解 Java 如何管理类的机制。学会了 Java SDK 环境，转到集成开发环境是很简单的事情，就像会用了手动相机再去用傻瓜相机一样。

2.1 Java SDK 的安装

编译运行 Java 程序，最基本的要求是安装 Java SDK。安装文件可以从以下网站下载：

<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

下载时注意选择正确的硬件和操作系统平台。目前硬件平台主要区分 32 位和 64 位，操作系统平台主要有 windows、Linux 和 Solaris 等。

安装过程主要有两个步骤，第一步是运行安装程序，第二步是将编译器和虚拟机程序所在目录添加到缺省路径。

以 windows XP 平台为例，目前 Java 的版本是 7，下载下来的程序为 jdk-7u3-windows-i586.exe。双击运行后，选择接受许可协议。JavaSDK 是免费的，采用免费软件的许可协议。然后选择安装位置，为避免不必要的错误，目录机构中最好不要出现中文字符，并且要记住安装的位置。安装界面如下图所示：



图 2-1 Java SDK 7 安装界面

安装程序运行结束后，Java 编译器、虚拟机和类库都安装好了。接下来需要将编译器和虚拟机所在目录添加到系统缺省路径中，才能在命令行中使用。在 windowsXP 下的具体方法是设置环境 path。启动控制面板，打开“系统”对话框，选择“高级”页面，点击“环境变量”按钮，打开环境变量设置对话框。在下面的“系统环境变量”列表框中，选中“path”变量（如果没有需要创建一个），然后点“编辑”按钮，将 Java SDK 安装目录下的 bin 子目录添加到 path 变量中去。比如，如果 Java SDK 安装到 D:\javasdk 目录，那么就在 path 变量值的文本框中，最后添加一个分号 “;”，再加上 “D:\javasdk\bin”。如下图所示：

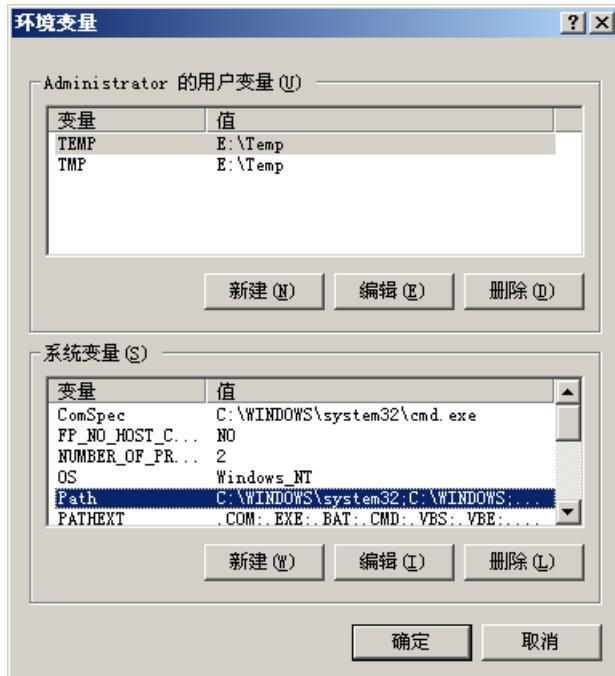


图 2-2 设置系统环境变量 Path

JavaSDK 安装完成后，应该测试一下。测试需要打开一个命令行窗口，又叫控制台窗口。有两种方法可以打开命令行窗口，一种是从“开始”菜单中选择“附件”——“命令提示符”，另一种是从“开始”菜单中选择“运行”并输入“cmd”。如下图

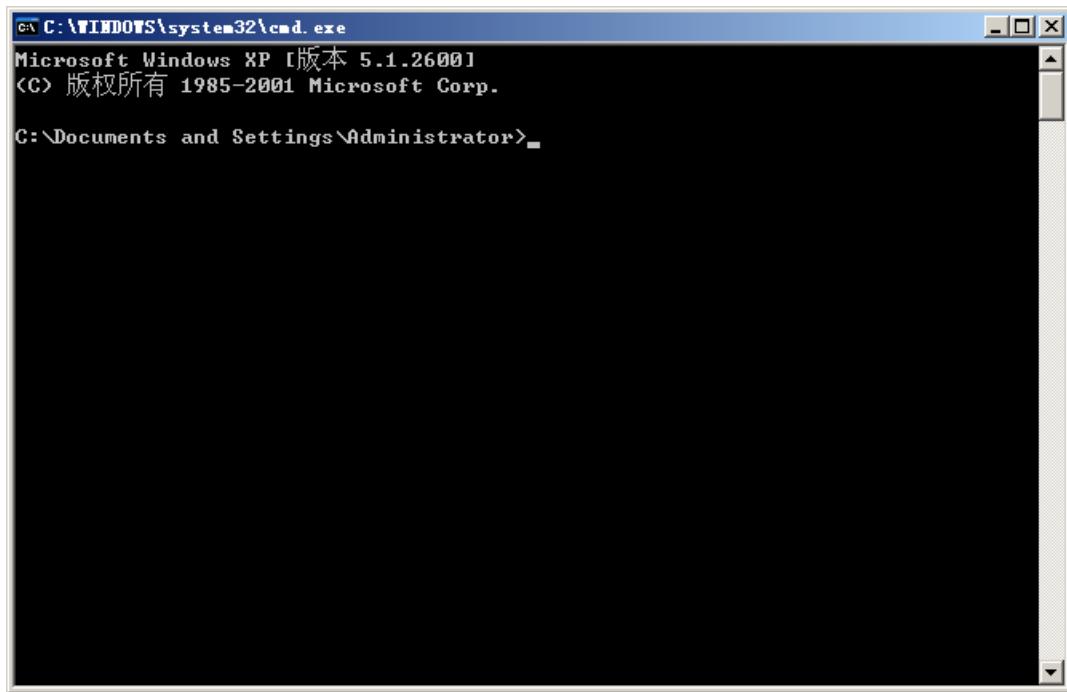


图 2-3 命令行窗口

打开命令行窗口后，在其中输入一行命令“javac”并回车，如果看到好多关于 javac 命令选项的提示信息，就说明安装正确。反之，如果提示找不到“javac”命令，就说明安装过程中出现了问题。如下图

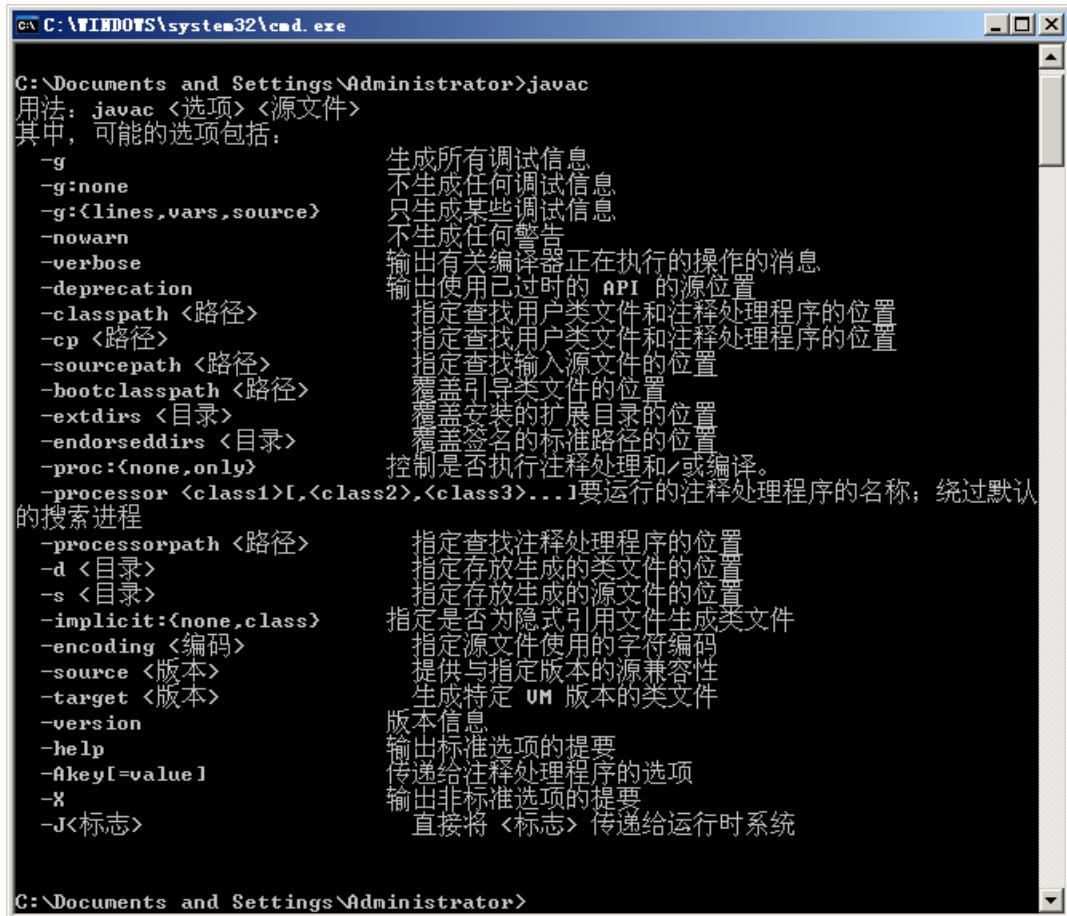


图 2-4 执行 javac 命令测试 Path 是否设置正确

出现问题很可能是没有正确地将 bin 目录添加到缺省目录中去，需要检查前面的步骤，改正后再做一遍。

2.2 第一个 Java 程序

Java 的源代码文件后缀为.java。源代码文件写好后，需要用编译器进行编译，生成字节码文件，后缀是.class。有了字节码文件后，再用虚拟机去执行。Java 程序从编写到运行的过程如下图：

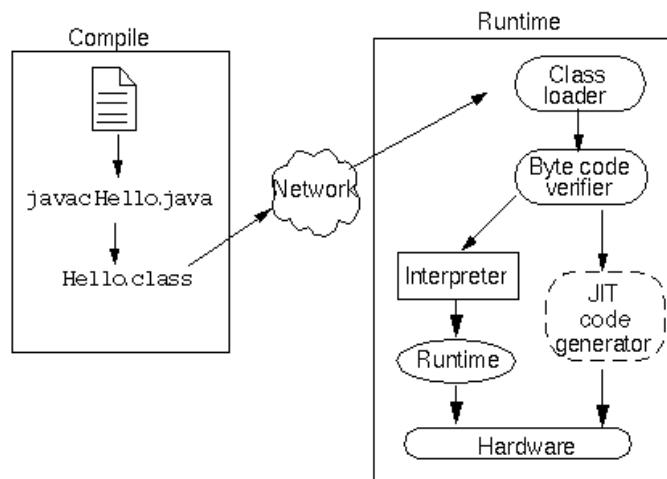


图 2-5 JavaSE 程序的编译运行

下面编写第一个 Java 程序，叫 HelloWorld，主要功能是在命令行窗口中输出一行字符串“Hello World!”。程序编写、编译和运行的步骤如下：

第一步是编写程序源代码。Java 源代码文件虽然后缀是.java，但其实是个文本文件，所以可以在任意一个文本编辑器中编写 Java 源代码，比如 Windows 记事本、UltraEdit 等。创建源文件的具体步骤如下：

- 1、编写程序时一般把一个项目中的所有文件放到一个目录下，所以首先在 D 盘创建一个文件夹，可以命名为 firstJava。
- 2、在该文件夹中，新建一个文本文件，然后改名为 HelloWorld.java。注意改名前应该设置资源管理器显示文件后缀，这样去掉原来的后缀.txt，否则文件名会被改为 HelloWorld.java.txt。显示文件后缀的方法是在上面的菜单中选择“工具”——“文件夹选项”——“查看”，如下图所示：

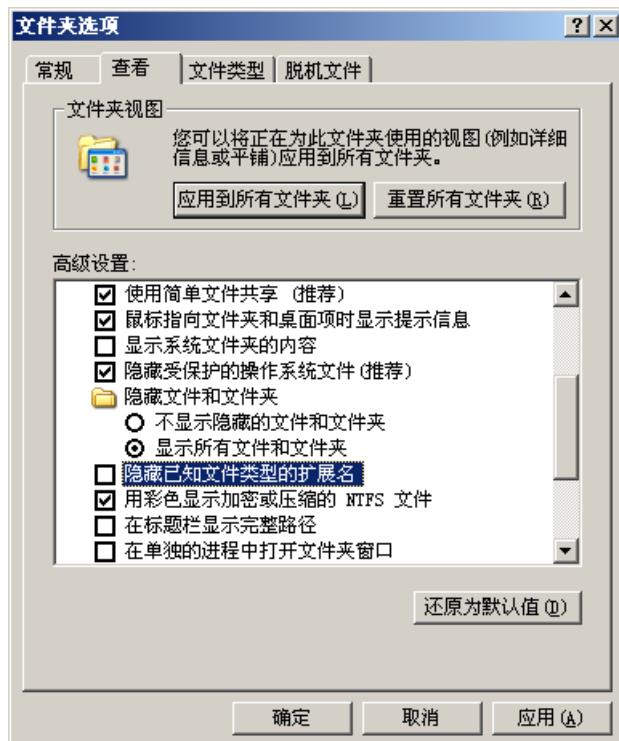


图 2-6 显示文件名后缀

3、用记事本打开 HelloWorld.java，在其中输入下面的代码：

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

代码输入后，保存文件即可。注意，先不用管代码是什么意思，文件名必须叫 HelloWorld，代码中字符的大小写不能错，任何一个符号都不能省略或出错，引号 “” 和分号 “;” 不能用成全角的中文符号。如下图所示：



图 2-7 HelloWorld 源代码

第二步是编译。首先打开一个命令行窗口，然后切换到源文件所在目录，再执行编译命令。具体步骤是：

- 1、打开命令行窗口。
- 2、切换盘符。命令行窗口一打开时当前目录是 C 盘的某个目录，需要先切换当前盘符，命令是 “D:”，即在命令行窗口中输入 “D:”，然后按回车，就会看到当前目录变成了 D 盘根目录。

3、切换目录。因为源文件不是放在 D 盘根目录下，而是在 firstJava 目录下，所以还应切换到该目录。命令是 “cd firstJava”。

4、编译 HelloWorld.java。用的命令是 “javac HelloWorld.java”

正确编译后会在同一个目录下生成一个 HelloWorld.class 文件。如果源代码中有错误，编译时也会给出出错的位置（第几行）和原因。需要仔细阅读出错信息后，检查源代码，找出错误并改正。

第三步是运行。继续在命令行窗口中，执行命令 “java HelloWorld”，会看到输出一行字符串 “Hello World!”。注意命令中的 HelloWorld 后面没有 “.class”。

编译运行的过程如下图：

The screenshot shows a Windows XP command prompt window titled 'cmd C:\WINDOWS\system32\cmd.exe'. The window displays the following command-line session:

```
C:\> Microsoft Windows XP [版本 5.1.2600]
(C) 版权所有 1985-2001 Microsoft Corp.

C:\Documents and Settings\Administrator>d:
D:>cd firstJava
D:\firstJava>javac HelloWorld.java
D:\firstJava>java HelloWorld
Hello World!
D:\firstJava>
```

图 2-8 HelloWorld 编译执行过程

编译运行的过程中，用到了一些命令行中的命令，包括切换盘符和切换目录。切换盘符的命令比较简单，只要输入目标盘符，加上冒号 “:” 即可。切换目录用的 “cd” 命令，格式是 “cd 目标目录”，但目标目录有几种不同的表示形式，主要包括：

- 1、使用相对路径，例如： cd firstJava，从当前目录开始计算目标目录。
- 2、使用绝对路径，例如： cd \firstJava，从根目录开始计算目标目录。
- 3、上一级目录，例如： cd..，切换到当前目录的上一级目录。
- 4、根目录，例如： cd\，直接返回到根目录。

编写第一个程序时容易犯的主要错误有：

- 1、拼写错误，包括大小写混淆。
 - 2、文件名和类名不一致，Java 要求文件名和类名要一致
- 出现错误后，应仔细阅读提示信息，检查源代码，改正后再编译运行。

2.3 帮助文档

写程序时会用到类库中的许多类，如果想知道某个类有哪些属性方法，每个方法怎么用，最快的办法就是查询帮助文档。

帮助文档是对 Java 类库中所有类的说明。帮助文档对每个类采用统一的格式说明类的功能、属性、方法等。帮助文档相当于一本字典，供程序员查询。

Sun 提供的 Java 帮助文档是 html 格式的，可以从下载 Java SDK 的网站下载，是英文版的。下载后，通常也解压缩到 Java SDK 安装目录下的 doc 文件夹中。另外，还有一些开发者做的 chm 格式的帮助文档，汉化过的，比 html 格式的更加好用。

比如，要查询 String 类的说明，只要打开帮助文档，搜索 String 类，就可以了，如下图：

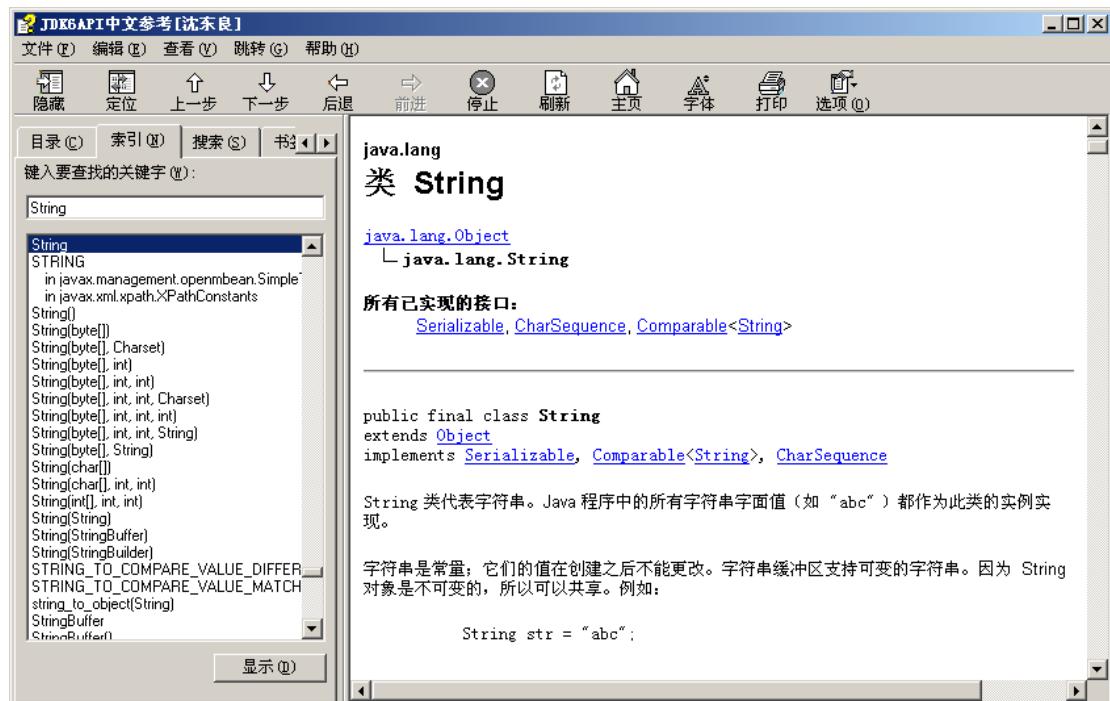


图 2-9 Java 帮助文档

这种帮助文档的格式统一清晰，便于开发者使用。Java 还提供了一套自动生成帮助文档的机制，让开发者可以为自己的类也编写出这样的帮助文档。开发者可以在源代码中用特殊注释的形式添加用于生成帮助文档的内容，格式是 “`/** <帮助文档内容>`”，这种注释可以加在类声明、属性声明、方法声明之前。然后用 “`javadoc`” 命令就可以自动生成标准格式的帮助文档。关于如何生成帮助文档，请查阅 `javadoc` 相关资料，本书不做详细介绍。

这种自动生成类帮助文档的机制，大大方便了类的共享。同时，因为帮助文档信息又可作为代码的注释，也方便了源代码的阅读。

第三章 程序的基本概念

本章简要介绍程序的一些基本概念，包括程序、语句、语句块、关键字和标识符。如果您对这些概念已经熟悉，可以跳过本章；如果您仍然阅读本章，应该可以加深对程序设计的理解。

3.1 程序的组成

关于程序的基本概念包括：程序、语句、语句块、关键字和标识符。

可以把一个程序看作一篇文章，由程序员写出来，告诉计算机如何一步一步完成某个任务。程序中的语句相当于文章中的句子，语句块相当于段落，关键字和标识符都相当于单词。关键字是事先约定好的有特殊含义的单词，可以被计算机用于理解整个程序。标识符是程序员起的一些名字，特定于要完成的任务。

一个程序是由若干个语句块构成的。每个程序块包含多条语句。一条语句是由分号终止的一串代码。语句又由关键字和标识符构成。关键字是程序设计语言保留的有特殊含义的字符串。标识符是用户定义的一些名称。

语句块是以大括号括起来的多条语句。一个语句块从外面可以看作一个整体，占一条语句的位置。比如说 `if-else` 语句后面有两个分支，每个分支可以是用大括号括起来的多条语句，就是两个语句块。块还可以用于类的定义。块语句可以嵌套，一个块可以包含其他块。

`Java` 语言对源代码格式要求不严格。程序中结构成分之间可以添加任意多的空白，例如语句和语句之间、变量和运算符之间都可以加任意多的空白。多条语句可以写在一行中，只要中间用分号隔开就行。虽然 `Java` 不要求严格的源代码格式，但是为了代码的美观、减少错误和阅读，我们应该形成良好的代码格式。特别是要注意缩进，即同一个嵌套层次上的语句前面的空格要一样多，嵌套层次越深空格越多。控制缩进通常使用空格键或 `Tab` 键。缩进可以有效避免左右括号不匹配的错误。

3.2 标识符

程序里用到的一些名称称为标识符。相当于文章中的名词或动词。`Java` 中的标识符用于给变量、类和方法命名。`Java` 规定标识符是以字符、下划线或美元符号开头的字符串。不允许以数字开头，因为会和数值混淆。标识符中的字符是区分大小写的，“`String`” 和 “`string`” 就是两个不同的标识符，要注意区分。标识符没有最大长度限制。

下面这些例子都是些合法的标识符：

```
identifier  userName  User_name  _sys_var1  $change
```

关于标识符，还有一些需要注意的地方：

- 1、Java 采用的是 16 位的 Unicode 编码，所以可以用中文做标识符名字，但不鼓励这么做。
- 2、标识符不能和关键字冲突。关键字是用来区分程序结构的。标识符里可以把关键字作为一部分，例如 `this` 不能做标识符，但 `thisone` 可以做标识符。
- 3、\$ 符号尽量不要用，是 Java 用于给内部类命名的。

关于 Java 的标识符，还有一些约定俗成的规矩：

- 1、类名用名词或名词短语，第一个字符要大写，如 `String` 和 `System`。
- 2、变量用名词或名词短语，第一个字符要小写。
- 3、方法用动词或动词短语，第一个字符要小写。
- 4、当用词组作为标识符时，从第二个单词开始首字母大写，作为分词的标志，比如 `moveTo`, `fileName` 等。

这些规矩不是强制的，但一般 Java 程序员都会这样去命名，而且 Java 类库中的类、变量和方法就是这样命名的。这样写出来的程序更像自然语言，易于理解。

3.3 关键字

关键字是对于 Java 编译器有特殊含义的字符串。Java 语言用的关键字有很大部分和 C++ 相同，因为 Java 就是在 C++ 的基础上设计出来的。Java 中的关键字有以下这些：

数据成分：

```
boolean, int, byte, short, long, char, float, double, void, null,  
class, extends, abstract, interface, implements,  
private, protected, public, super, this,,  
const, final, static, transient, volatile, native, synchronized,
```

运算加工成分：

```
new, =,  
+, -, *, /, %, ++, --, ?:, &, |, ^, ~, >>, <<, >>>,  
==, <, >, <=, >=, !=, instanceof,
```

控制成分：

```
for, while, do, if, else, switch, case, default, break, continue, goto, return,  
throw, throws, try, catch, finally,
```

程序结构成分：

```
import, package, (,), [], {}, ;, ", ;//, /*, */
```

这些关键字可以分成四大类来看。第一类是和数据描述有关的，包括数据类型、类、可见性等。第二类是和运算有关的，就是怎样去处理数据的操作。第三类是流程控制的，就是控制代码执行走向的。第四类是程序结构有关的。

关键字不用死记硬背，用得多了自然而然就记住了。

和 C++ 有一些细微的差别。

- 1、`true`、`false` 和 `null` 都是小写。
- 2、没有 `sizeof` 运算符，因为不需要计算内存长度，只要用 `new` 运算符就可以。
- 3、`goto` 和 `const` 是关键字，但是没有实际意义了。

3.4 变量和值

变量在程序中用于保存数据，如输入的参数、计算的结果。每个变量有一个标识符作为名字，通过变量的名字可以访问变量的内容，读取或写入都可以。根据保存数据的格式的不同，变量具有不同的类型，在程序设计语言中成为数据类型。Java 中有 8 个基本数据类型，能够保存整数、浮点数、布尔值、文本字符格式的数据。

变量是从高级程序设计语言中出现的概念，之前的机器语言和汇编语言都没有变量的概念，汇编语言中的“变量”只能算是内存地址的助记符。程序运行时，每个变量实际上对应于一块内存空间，变量保存的数据就存放在这块内存中，读取/写入变量的值就是读取/写入内存的内容，数据在内存中的保存方式就是数据类型。从抽象的变量到具体的内存地址的映射是由编译程序完成的，不需要程序员知道。正是通过这种方式，变量把程序员从具体繁杂的内存地址中解放出来，减轻了程序员对硬件知识的依赖，从而提高了程序编写的效率。

每个变量都有生命周期，即运行的时候变量对应的内存从分配到回收的时间段。从源代码中来看，就是每个变量都有作用范围，从变量声明时开始，到变量所在的程序块结束。另外，变量在内存中所处的位置不同，对其生命周期也有影响。程序的内存分为栈内存和堆内存。栈内存主要用于函数调用时的参数传递，每个函数调用在栈上占一块内存，函数中的局部变量都在栈上。堆内存主要用于程序中的动态内存分配，一个程序只有一个堆，比如，所有对象都分配在堆上。

值是变量中保存的内容。在源代码中，对于不同的数据类型，有不同的表示数值的方法。

3.5 表达式和运算符

表达式是由值、变量、运算符按有意义的排列方式组成的式子。程序中的表达式类似于数学上的函数，其输入为变量的定值，输出则为根据表达式运算之后所产生的数值。表达式的运算结果可以通过赋值运算符再写入变量保存。

表达式运算需要读取变量的值、执行运算符、将结果赋值给另一个变量，这正好构成了计算机主要的工作循环——读取内存中的数据、加工、结果保存回内存。可以说，表达式体现了计算机处理数据的过程。

运算符代表了计算机的处理器加工数据的方法。程序中的运算符一般包括加减乘除等算数运算符、判断大小的逻辑运算符、与或非的布尔运算符等。正是因为处理器有这样的能力，程序设计语言中才会有这些运算符。

3.6 子程序和函数

子程序体现的是分而治之的思想。如果一个程序功能很多，代码特别长，就不便于阅读和维护。如果按照功能，把大的程序分成很多小的部分，每个部分实现一项较简单的功能，就容易阅读和维护得多，这些小的部分就是子程序。一个子程序是一段相对独立封闭的代码，由子程序名来标识，由入口参数来给出需要的各种变量值，计算结果由返回值返回。

要执行子程序需要“调用”，即给出子程序名和入口参数。对于子程序的返回值，也可以保存到变量中进一步处理。通过使用不同的入口参数调用子程序，可以重复执行子程序的代码，避免相同的代码在程序中出现多次，便于程序维护。

函数是子程序的另一种叫法，因为子程序和数学函数类似：子程序名就相当于函数名，入口参数相当于函数自变量，返回值相当于函数结果。最早的高级语言中只有子程序的概念，没有函数。`Pascal` 语言中，把没有返回值的子程序称为子程序，有返回值的子程序成为函数。`C` 语言将子程序统称为函数，分为有返回值的和没有返回值的。

整个程序也可以看做一个函数，一般要规定一个固定的函数名作为整个程序的入口，通常是 `main` 函数。

对于函数有一个重要的概念称为重载，指两个函数具有相同的名字，但是参数表不同，函数的返回值类型可以相同也可以不同。重载是一个操作对于不同的对象有不同的处理方法。比如，加法“`+`”操作就具有很多重载，两个整数相加是一种计算方法，两个有理数相加是一种计算方法，两个分数相加又是另一种计算方法。加法“`+`”操作的返回值类型对于不同的操作数也不相同，整数相加还是整数，有理数相加是有理数，分数相加是分数。

调用有重载的函数时，编译器无法通过函数名来区分两个函数，只能通过匹配参数表来确定调用的是哪个函数。

3.7 Java 程序的基本结构

在 `HelloWorld` 例子中，给出了一个最基本的 `Java` 程序。其中涉及了三个主要的名字：文件名、类的名字和主函数。如下图：

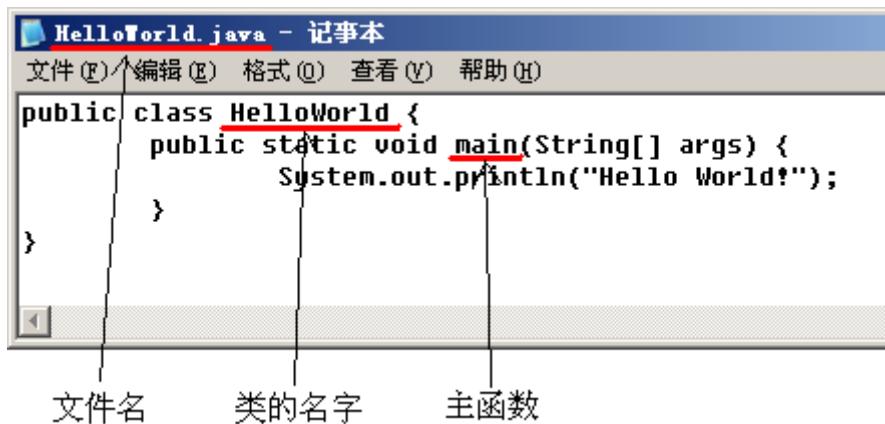


图 3-1 Java 程序基本结构

在编写 Java 程序时，先要确定类的名字，此例中为 `HelloWorld`。注意，Java 是区分大小写的，也就是说，`HelloWorld` 和 `helloworld` 是两个不同的名字。确定了类的名字后，Java 规定文件名要和类的名字相同，所以源代码文件的名字应该命名为 `HelloWorld.java`。否则，编译的时候会报错。编译以后生成的字节码文件也对应的叫做 `HelloWorld.class`。这种对应关系决定了，基本上一个类对应一个源代码文件和一个字节码文件。

每个类可以有一个作为启动程序入口的 `main` 函数。如果一个程序包含多个类，那么可能有多个类有 `main` 函数，程序就有多个执行的入口。

类名字前面的修饰词 `public class` 的意思是声明一个公开的类，后面大括号中的内容为类体。`main` 函数前面的修饰词 `public static void` 的意思是公开的、静态的、无返回值的，后面的圆括号中的内容为函数的参数，`String[]` 的意思是字符串数组，`args` 是参数变量的名字。再后面大括号中的内容就是 `main` 函数的函数体，里面编写各种语句。

Java 程序的基本结构就是这样的，每个程序的类和主函数都是这样声明。这些声明的含义在后面的章节中还会作详细介绍。

第四章 基本语句

4.1 数据类型

所有的通用程序设计语言都要有三个方面：数据类型、运算符和流程控制。但是每种语言在这三个方面又有细微的差别。

数据类型是从高级语言出现以后才有的一个概念。数据类型是帮助程序员去管理内存的。用计算机程序来解决实际问题时，需要先把问题抽象为数学模型，就会涉及到很多数据。这些数据如何保存到计算机中去，要靠数据类型。比如说一个整数要保存到计算机中，可以占 1 个字节，也可以占 2 个字节、4 个字节或 8 个字节，占用不同的内存大小就要选择不同的数据类型。因此，数据类型决定数据如何保存到计算机中。

每种高级程序设计语言都有自己设计好的数据类型，这些数据类型是根据用计算机程序解决问题时的具体需要而设计的。所以，基本上所有高级程序设计语言的数据类型都很相似，但有小的差别。

Java 有八个基本数据类型：`boolean`、`byte`、`short`、`int`、`long`、`float`、`double`、`char`，分别称为布尔型、字节型、短整型、整型、长整型、单精度浮点型、双精度浮点型、字符型。`boolean` 用于保存布尔型的值，即真和假。`byte`、`short`、`int`、`long` 用于保存整数，但能表示的整数的范围不同。`float` 和 `double` 用于保存有理数，能够表示的数的大小和精度都不同。`char` 用于保存一个字符。

Java 中整数类型的最大特点有两个：

- 长度固定：不管在什么平台上，都是占用这么多字节，不会变化。
- 带符号：所有整数都是采用带符号方式保存的，没有无符号数，即最高位是符号位。

4.1.1 boolean 型

`boolean` 型变量只有两种取值，即真和假，在 Java 中表示为 `true` 和 `false`。`boolean` 型变量描述对一个事情真假的判断，或者是真的，或者是假的。下面的例子：

```
boolean truth = true;
```

声明了一个布尔型变量 `truth`，取值为真。

需要注意的是，Java 中 `boolean` 型不能和整型混用，这一点上和 C++ 不同。对 `boolean` 型要求更严格，可以防止一些程序中的错误。C++ 中一个常见的错误是把布尔运算的等于符号“`==`”（双等号）写成赋值符号“`=`”（单等号）。例如，要判断变量 `x` 是否为 1，应该写成

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

如果一疏忽，就很容易写成

```
If ( x=1)
```

如果允许 `boolean` 型和整型混用，那么这种情况下编译时不会报错，但会造成判断结果永远为真，而且 `x` 的值被修改为 1，这种错误会很难发现。Java 不允许 `boolean` 型和整型混用，对于这种情况在编译时就会报错，程序员就会很容易发现和改正这种错误。这就是采用严格的数据类型后带来的好处，比降低了灵活性的损失要大得多。

4.1.2 char 型

在 Java 中，`char` 型用于保存一个 16 个二进制位的 `Unicode` 字符。`Unicode` 是一种对字符进行编码的方式，可以表示世界上所有语言中的字符。在 `Unicode` 出现之前，计算机中对字符编码采用的是 8 个二进制位的 `ASCII` 码，只能表示英文字母、数字以及一些符号。随着用计算机处理其它语言文字的需要，才制定出了 `Unicode` 编码。

`Unicode` 编码可以有不同长度，Java 采用的是 16 个二进制位，即两个字节来表示一个字符，所以 Java 中能够用一个 `char` 型变量保存一个汉字。例如，下面的变量声明和赋值是允许的：

```
char c = '中';
```

这在 C++ 中是不行的。

字符的值必须用英文的单引号（半角）“'”引起来，如上例中的“'中'”。不能是双引号，也不能是中文字符的单引号（全角）“‘”，这一点应该特别注意，在输入英文单引号时，先关闭中文输入法。此外，程序中还有很多地方使用英文的分号“;”和双引号“""”，不能用成中文的分号“；”和双引号““”。这些中英文符号间看上去有差别，但是很不容易分辨，如果输错，编译时会报错。如果编译时报告“非法字符”的错误信息，就应该检查出错的那一行中是否把英文符号输成了中文符号。

在字符的值里，还有一个特殊的表示符号——转意字符“\”。转意字符后的下一个字符不再表示原来的意思，而是转意成了其它的意思。比如，'\t' 不表示字符“t”，而是表示制表符，对应于键盘上的 Tab 键，会向右空出一定距离并和其它行在纵向上对齐。'\\' 转意后表示'\'，如果没有这一规定，那么字符'\' 就无法表示了。另外，转意字符后还可以跟字符的 Unicode 编码，用于表示一些不方便从键盘输入的特殊字符，如'\u005c' 表示反斜杠'/'，编码用 16 进制表示。要查 Unicode 编码的详细信息，请访问网站：<http://www.unicode.org>。

设置 char 类型的目的是为了让计算机能够方便地处理文字。

4.1.3 整数类型

程序里最常用的数据是整数类型。下面分别从整型变量和整数值来介绍。

整型变量

Java 里的整数类型分为四种：byte、short、int、long，分别占用 1、2、4、8 个字节。占用的字节个数不同，能够表示的数的范围就不同。显然，占的字节数越多，能够表示的范围越大。

这四种整数类型能够表示的整数范围如下：

长度	类型	范围
8 bits	byte	-2 ⁷ (128) ... 2 ⁷ -1
16 bits	short	-2 ¹⁵ (32768) ... 2 ¹⁵ -1
32 bit	int	-2 ³¹ (21 亿) ... 2 ³¹ -1
64 bits	long	-2 ⁶³ (900 亿) ... 2 ⁶³ -1

整数值

整数值可以有三种进制表示，分别是十进制、八进制和十六进制，通过不同的前缀区分。以非 0 的数字开头的数是十进制，以 0 开头的数是八进制，以 0x 开头的数是十六进制。例如：

2 十进制数值 2

077 首位的 0 表示这是一个八进制的数值，换算成十进制为 63

0xBAAC 首位的 0x 表示这是一个 16 进制的数值

整数值的保存方式还有区别。缺省情况下，一个整数值用 4 个字节保存，可以看作是 `int` 型。如果要表示一个很大的整数值，4 个字节保存不下，那么需要把整数值声明为 `long` 型，方法是在数值后面加上“`L`”或“`l`”。例如，下面的写法是正确的：

```
long population = 6000000000L;
```

而下面的写法是错误的：

```
long population = 6000000000;
```

编译时会报错，因为 4 个字节保存不下 `6000000000` 这个数。最后加上“`L`”后，表示这个数字要保存在 8 个字节中，就可以保存下了。

4.1.4 浮点类型

浮点类型用于表示有理数。下面分别从浮点型变量和浮点值来介绍。

浮点变量

浮点变量有两种：`float` 和 `double`。`float` 叫作单精度浮点型，简称单精度型或浮点型。`double` 叫作双精度浮点型，简称双精度型。`float` 型占用 4 个字节，`double` 型占用 8 个字节。

浮点类型是用有效值和指数两部分来表示一个有理数。除了表示范围外，浮点数还有一个表示精度的问题，即有效值的小数点后保留的有效位数。占用内存越多，保留的有效位数越多，精度越高。

这两种浮点类型的表示范围和精度如下：

长度	类型	范围	精度
32 bits	<code>float</code>	约± <code>3.40282347E+38</code>	(有效小数 6-7 位)
64 bits	<code>double</code>	约± <code>1.797693134862317E+308</code>	(有效小数 15 位)

因为浮点类型的变量有精度的问题，所以判断两个浮点类型的变量是否相等不应使用“`==`”号，因为有的情况下两个表达式的运算结果在理论上相等，但实际上却会因为计算机的浮点运算误差而有很小的差别。

如果 `x` 和 `y` 是两个 `double` 型变量，要判断它们两个的值是否相等，应该用：

```
if( Math.abs(x-y) < 0.00001 )
```

其中，`Math.abs(x-y)` 是求 `x` 和 `y` 之间差的绝对值，`0.00001` 是精度要求。意思是只要 `x` 和 `y` 之间的误差在精度要求范围内，就认为 `x` 和 `y` 相等。不应该直接用下面的方式：

```
if (x == y)
```

浮点值

浮点值缺省为 `double` 型，用 8 个字节存储。也可以在浮点值后面加上“`D`”或“`d`”显式说明浮点值是 `double` 型。目前，后面加或不加“`D/d`”是等价的。如果想把一个浮点值声明为 `float` 型，需要在值的后面加上“`F`”或“`f`”，那么就会用 4 个字节存储。浮点值可以采用科学计数法表示，用字符“`E`”或“`e`”隔开有效数字和指数。

下面是一些正确的浮点值写法：

3.14	一个采用缺省方式的 double 值
3.14D	一个采用显式声明的 double 值
4.02E23	一个采用科学计数法的 double 值, 4.02×10^{23}
2.718F	一个 float 值

Java 类库中还定义了一些特殊的浮点数:

最大值: Double.MAX_VALUE	$((2-2^{-52}) \times 2^{1023})$
最小值: Double.MIN_VALUE	(2^{-1074})
正无穷大: Double.POSITIVE_INFINITY	无穷大表示溢出, 如 0 除任何数。
负无穷大: Double.NEGATIVE_INFINITY	
非数字: Double.NaN	非数字表示出错, 如 $0/0$ 或负数开方。

4.1.5 数据类型转换

在运算过程中, 一种类型存储的数据可能需要转换为按另一种类型存储。例如, 一个 int 型数据和一个 double 型数据相加, 需要先把 int 型数据转换为 double 型, 再执行两个 double 型数据间的加法。这是因为计算机 CPU 的硬件只支持同类型数据间的运算。

数据类型转换分为两大类, 一类是可以自动进行的转换, 不需要显式声明, 前提条件是转换后的类型的表示范围大于转换前的。另一类是不能自动进行的转换, 因为转换后的类型的表示范围小于转换前的, 此时因为可能有信息损失, 若要转换需要显式地指出, 称为强制类型转换或造型。

自动类型转换

可以自动进行的类型转换如下图所示:

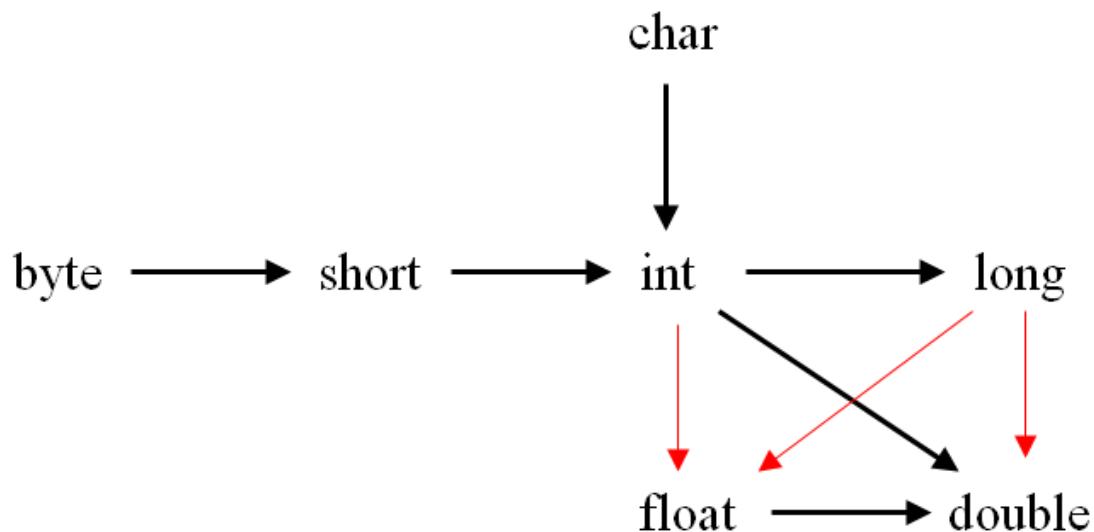


图 4-1 自动类型转换

虚线箭头表示有可能有精度损失的转换。

图中横向的主线是整型数据间可以“由小到大”转换。`byte` 型可以自动转成 `short` 型，因为 1 个字节中的数据保存到 2 个字节中也没有问题。同样道理，2 字节 `short` 型可以自动转成 4 字节 `int` 型，4 字节 `int` 型可以自动转成 8 字节 `long` 型。这些转换都不会损失精度。

纵向来看，`char` 型可以自动转成 `int` 型，因为 `char` 型占两个字节，但是无符号的整数编码，有 16 个二进制位，而 `short` 型虽然也是占 16 个二进制位，但却有一个位用作符号位，只有 15 个位来保存有效值，所以无法保存 `char` 型数据，至少要保存到 4 字节的 `int` 型，才不会损失精度。

`float` 到 `double` 的转换也不会超出范围和损失精度。`float` 用 3 个字节保存有效值，1 个字节保存指数；`double` 用 6 个字节保存有效值，2 个字节保存指数，都比 `float` 大。

`int` 到 `double` 的转换不会超出范围和损失精度。`int` 用 4 个字节保存有效位，而 `double` 用 6 个字节，大于 `int`。

`int` 到 `float` 的转换和 `long` 到 `double` 的转换不超出范围，但有可能损失精度。`int` 到 `float` 的转换是把 4 字节的有效值保存到 3 字节中；而 `long` 到 `double` 的转换是将 8 字节的有效值保存到 6 字节中。

下面是可以自动转换的例子：

```
int salary = 2000;  
float output = salary;
```

`int` 型的变量 `salary` 被自动转换为 `float` 型。

强制类型转换（造型）

除了能自动进行的类型转换外，其它的类型转换都需要显式声明，就是在数据前面加上用圆括号括起来的类型。例如：

```
double salary = 2000.85;  
int input = (int)salary; //input 值为 2000
```

将浮点类型转换为整数类型时，采用的是去尾法，即舍弃小数点后的部分。若要采用四舍五入法，需使用条件判断语句或者 `Math.round()` 方法。例如，对上例中的 `salary` 四舍五入应写为：

```
int input = (int)Math.round(salary); //2001
```

当转换结果超出了目标类型的范围时，结果是被截取的不同的值。例如：

```
byte count1 = (byte)300;//count1 值为 44  
byte count2 = (byte)-400;//count2 值为 -112
```

`boolean` 类型不参与造型。若要对 `boolean` 类型进行转换，使用`(?:)`操作符，如

```
boolean result = true;  
int score = result?1:0; //score 值为 1
```

4.1.6 数据类型小结

关于变量的使用，Java 中有一些规定和习惯。在 Java 程序中，任何变量都必须经初始化后才能被使用。强制变量先初始化再使用，可以避免一些忘记初始化的错误。变量声明与初始化赋值可以分开，也可以一起进行。变量可以在函数一开始全部声明，也可以在使用之前再声明，以程序易于阅读修改为原则。现在一般的习惯是使用前定义。

下面是一些正确的使用数据类型进行变量声明和赋值的例子：

```
int x, y;  
float z = 3.414F;  
double w = 3.1415;  
boolean truth = true;  
char c='中';
```

下面是一些错误的例子：

```
int y = 3.1415926;      // 3.1415926 不是整型  
double w = 175,000;     // 数值中不能出现逗号  
boolean truth = 1;       // boolean 型和整型不能混用  
float z = 3.14;          // 3.14 为 double 型浮点值，占 8 个字节，用 float 型变量无法保  
存，应改为：float z=3.14f;
```

下面是一个变量未初始化就使用的例子：

```
int x=10,y,z;  
if (x>9) y=x;  
z = y;
```

编译时报错，原因是 y 可能没有初始化就使用。即使有一个 if 分支初始化，编译器也认为有另一个分支，存在不初始化的可能。

4.2 运算符

4.2.1 运算符及其含义

运算符描述的是对数据的操作。

Java 中的运算符以及它们的含义如下：

- 算术运算符： +, -, *, /, %
- 递增递减运算符： ++, --
 - 建议不要和其他表达式混用。
- 关系运算符： ==, >, <, >=, <=, !=, instanceof
- 三元运算符： ?: (选择运算)
 - 最适合于求最大(小)值，如： z = x>y?x:y;

- 布尔运算符: `&&, ||, !`
 - 布尔运算采用短路方式, 例如: `if(a!=0 && b/a>2)`
- 位运算符: `&, |, ^, ~, >>, <<, >>>`
 - 移位运算符的简化操作: 若左侧操作数为 `int`, 则将右侧的操作数模 32 简化; 若左侧操作数为 `long`, 则将右侧操作数模 64 简化。即 `int x=100; x>>>32; x` 的值不变, 而不是 0。
 - `>>>` 运算符仅被允许用在整数类型, 并且仅对 `int` 和 `long` 值有效。如果用在 `short` 或 `byte` 值上, 则在应用`>>>`之前, 该值将通过带符号的向上类型转换被升级为一个 `int`。
- 数学函数和常量:
 - `Math` 类中提供 `sqrt, pow, sin, cos, tan, atan, atan2, exp, log, PI, E`。
 - 若要保证平台无关性, 应使用 `StrictMath` 类。

4.2.2 运算符的优先级

当一个表达式中有很多运算符的时候, 先执行哪个运算, 就涉及运算符的优先级。

`Java` 语言对表达式的执行也是采用左结合的方式。

优先级由高到低为:

```
[ ] . ( )
~ ++ -- +(正号) -(负号) (造型) new
* / %
+
-
> < >= <= instanceof
== !=

&&
||
?:
=
+= -= *= /=
```

明白优先级顺序可以省略括号, 如

表达式 `x+y>3` 先计算 `x+y`, 再计算是否大于 3, 利用了“+”优先级比“>”高。

表达式 `X+y>3 && x+y<10` 先分别计算两个加法, 再分别计算布尔表达式, 最后再进行与运算, 利用了优先级顺序“+”大于“>”大于“`&&`”。

如果写程序时对某些运算符之间的优先级不是很确定, 那么建议用圆括号括起来, 可以保证运算的先后顺序, 避免一些可能的错误, 也使阅读程序时, 对于计算顺序更明确。

和赋值符号“=”有关的运算符的优先级是最低的, 所以表达式都是先计算“=”右边的部分的结果, 再赋给左边的变量。

4.2.3 类型转换规则

二元运算中还有一个数据类型转换的问题，因为当使用二元运算符把两个值结合到一起时，需要在运算前将两个数据转换成相同的类型。

Java 运算中的数据类型转换遵循如下规则：

- 只要有一个数是 double 型，则另一个会转换成 double 型。
- 否则，只要有一个数是 float 型，则另一个会转换成 float 型。
- 否则，只要有一个数是 long 型，则另一个会转换成 long 型。
- 否则，两个数都转换成 int 型。

其中的规律是将表达范围小的数据类型转换成表达范围大的数据类型，这样计算结果超出表示范围的可能性才比较小。整数运算时至少按照 int 型进行，因为 Java 是按照 32 位机来设计的。

例如：如果有以下变量定义

```
byte count1, count2;  
float price;  
short money;
```

那么，`count1+count2` 为 int 型，因此如果写一条语句

```
byte sum = count1 + count2;
```

会出错，因为相当于将 int 型数据往 byte 型变量里保存。修改的方式是：

```
int sum = count1 + count2; 或 byte sum = (byte) (count1 + count2);
```

另外，`count1*price` 为 float 型，`money-(count1+count2)*price` 为 float 型。

4.3 流程控制语句

流程控制语句用来控制程序中语句的执行顺序。也就是说，执行完一条语句之后，下一步应该执行哪条语句。如果没有流程控制，程序中的语句默认是从上到下依次执行，这是因为计算机硬件在设计时就是这样设计的。

在很多情况下，需要根据前面的语句的执行情况决定下一步执行哪条语句。比如根据变量 `x` 的值的大小，当 `x` 大于等于 0 时如何处理，当 `x` 小于 0 时如何处理，这时就需要对程序的执行流程进行控制。

流程控制语句可以分为两种：分支和循环。分支语句就是程序执行到此处后有多个可选的分支路线，根据条件选择其中之一执行。循环语句的作用是重复执行一个代码段。

除了分支和循环外，还有一些配合循环语句一起使用的特殊流程控制语句。

4.3.1 分支语句

分支语句包括两条：两路分支语句 `if` 和多路分支语句 `switch`。

`if` 语句是两路分支语句，其格式是：

```
if(条件表达式){  
    分支一  
} else {  
    分支二  
}
```

其中，条件表达式的计算结果必须是 `boolean` 型的，如果结果为真，就执行分支一，结果为假就执行分支二。

`if` 语句的 `else` 分支可以省略。这种情况下，如果条件表达式结果为假，那么没有任何分支可执行。

`if` 语句可以连着使用，写成下面的形式：

```
if(条件表达式一){  
    分支一  
}else if(条件表达式二){  
    分支二  
}else {  
    分支三  
}
```

此时，如果条件表达式一为真，执行分支一，分支二和三都不再执行。如果条件表达式一为假、条件表达式二为真，执行分支二。如果条件表达式一和条件表达式二都为假，执行分支三。还可以有更多的 `if` 连接使用。

`switch` 语句是多路分子语句，其格式是：

```
switch (条件表达式){  
    case 结果一:  
        分支一  
        break;  
    case 结果二:  
        分支二  
        break;  
    default:  
        默认分支  
        break;  
}
```

其中，`switch` 后面的条件表达式和 `case` 后面的结果表达式的值必须是 `int` 类型，或者通过自动类型转换能够转换为 `int` 类型。如果条件表达式的结果与结果一相等，就执行分支一；如果条件表达式的结果与结果二相等，就执行分支二；依次类推。如果和所有结果都不相等，

就执行 `default` 后面的默认分支。只要有一个 `case` 分支执行，`default` 分支就不被执行。

如果在每个分支结束时不加 `break` 语句，那么继续执行下一分支。`default` 分支是可有可无的，如果没有 `default` 分支，并且条件表达式和所有结果都不相等，那么所有分支都不执行。

有的情况下在 `case` 语句块末尾特意不加 `break` 语句，继续执行下一个 `case` 块。比如已知某一天是 `m` 月 `d` 日，求这一天是这一年的第几天，例 1 月 1 日是第一天，2 月 1 日是第 32 天。这个程序如下：

```
int n=d;
switch(m) {
    case 12: n += 30; // 12 月的日期需要计算 11 月的所有天数
    case 11: n += 31; // 11 月的日期需要计算 10 月的所有天数
    case 10: n += 30;
    case 9: n += 31;
    case 8: n += 31;
    case 7: n += 30;
    case 6: n += 31;
    case 5: n += 30;
    case 4: n += 31;
    case 3: n += 28; // 3 月的日期需要计算 2 月的所有天数，暂不考虑闰年
    case 2: n += 31; // 2 月的日期需要计算 1 月的所有天数
}
```

4.3.2 循环语句

循环语句有三条：`for`、`while` 和 `do-while`。

`for` 语句的格式是：

```
for (初始化语句; 循环条件; 更新语句) {
    循环体
}
```

`for` 语句的执行过程是：

- 1、先执行初始化语句；
- 2、判断循环条件，若结果为真转第 3 步，若结果为假转第 5 步；
- 3、执行循环体中的语句；
- 4、执行更新语句，转第 2 步；
- 5、结束循环。

例如，求整数 1 到 100 的和的代码如下：

```
int sum=0;
for ( int i=1; i<=100; i++) {
```

```
    sum += i;  
}
```

其中的变量 `i` 叫循环变量，初始化语句主要是给循环变量赋初值，循环条件主要是对循环变量的值进行判断，更新语句改变循环变量的值，循环体中使用循环变量的值。

`for` 语句一般用于循环次数事先确定，循环变量在循环体中值不会改变的情况。如果在循环体中改变循环变量的值，那么循环次数就不好确定，也容易出现死循环的情况。初始化语句和更新语句中可以使用分号“，”分隔多个操作。

`while` 语句的格式是：

```
while ( 循环条件 ) {  
    循环体  
}
```

`while` 语句的执行过程是：

- 1、判断循环条件，若结果为真转第 2 步，若结果为假转第 3 步；
- 2、执行循环体中的语句，转第 1 步；
- 3、结束循环。

例如，求整数 1 到 100 的和的代码用 `while` 语句写的话如下：

```
int i=1;  
int sum=0;  
while ( i<=100 ) {  
    sum += i;  
    i++;  
}
```

`while` 语句一般用在循环次数事先不可确定且循环体可以一次也不执行的情况，因为是先判断循环条件再执行循环体。使用时需注意：确认循环控制变量正确初始化，控制变量必须被正确更新以防止死循环。在上面的例子中，循环控制变量为 `i`，更新它的语句是循环体中的 `i++`。

`do-while` 语句的格式如下：

```
do {  
    循环体  
} while ( 循环条件 )
```

`do-while` 语句的执行过程是：

- 1、执行循环体中的语句；
- 2、判断循环条件，若结果为真转第 1 步，若结果为假退出循环。

`do-while` 语句一般用在循环次数事先不可确定且循环体至少执行一次的情况，因为是先执行一遍循环体后才判断循环条件。

4.3.3 特殊流程控制语句

有三条特殊的流程控制语句，可以和循环语句配合使用。分别是 `break`、`continue` 和 `label`。

`break` 语句的作用是跳出 `switch` 语句或循环语句。在 `switch` 语句中，`break` 语句通常夹在每个 `case` 块的末尾，当程序执行到此处时，结束分支，跳出 `switch` 语句。`break` 语句在循环中的作用也类似。

一个使用 `break` 的例子是用循环找出 100 以内 5183 的一个因子，代码如下：

```
int x=2;
while ( x<100 ) {
    if ( 5183 % x == 0 ) break;
    x++;
}
```

在 `while` 循环中，`x` 的值会从 2 增长到 100，一旦发现 5183 能被 `x` 除尽，则执行 `break` 语句跳出循环。此时 `x` 的值就是 5183 的一个因子。

`continue` 语句的作用是开始下一次循环。一个例子是求 100 以内所有偶数的和，代码如下：

```
int sum=0;
for ( int i=1; i<=100; i++ ) {
    if ( i % 2 > 0 ) continue;
    sum += i;
}
```

在循环中，如果发现 `i` 是奇数就执行 `continue` 跳到下一次循环，后面的语句不会被执行。

`label` 用于标记一个循环(位置)，和 `break` 和 `continue` 一起使用。在多层循环中用来指出 `break` 和 `continue` 对应的是哪层循环。例如：

```
test: for (i=1; i<10; i++) {
    ...
    while (...) {
        if (j > 10) {
            continue test; //跳到 for 的更新语句 i++
            break test; //跳出 test 块，即执行 x=j
        }
    } // end while
} // end for
x = j;
```

第一行的 `test` 是一个 `label`，标记最外层的 `for` 循环。后面的 `continue test` 语句就表示继续执行 `for` 循环的下一个循环，及 `for` 循环的更新语句 `i++`。如果 `continue` 后面没有 `test` 标签的话，应该是继续 `continue` 所在的最内层循环的下一个循环，即 `while` 循环。`break test` 语句也是跳出 `for` 循环。

这三条语句合起来实现了一部分 `goto` 跳转语句的功能，都是将代码的执行从一个位置跳到

另一个位置，但是跳转的范围被限制在一个代码块内，跳转的位置被限制在代码块的头部或尾部，所以它们又称为受限的 `goto` 语句。这样设计既提供了一定跳转的功能，又没有违反结构化程序设计的要求。

4.4 基本输入输出方法

输入输出是指程序和使用者之间传递数据。输入是让数据从程序外进入到程序内，输出是让数据从程序内流出到程序外。没有输入输出，程序就无法读取数据和显示结果，其运行就失去了意义。

本章将分别介绍控制台输入输出方法和对话框输入输出方法。

4.4.1 控制台输入输出方法

一个控制台输入输出的例子如下：

```
//源文件应命名为 ScanerTest.java, 如叫其它名字编译会出错
public class ScanerTest {
    public static void main(String[] args) {
        java.util.Scanner sc = new java.util.Scanner(System.in);      //创建对象
        System.out.println("Please enter a string:");
        String s = sc.next();           // 从控制台输入一个字符串
        System.out.println(s);
        System.out.println("Please enter an int:");
        int i = sc.nextInt();
        System.out.println(i);
        System.out.println("Please enter a float:");
        float f = sc.nextFloat();
        System.out.println(f);
    }
}
```

控制台输入主要靠 `java.util.Scanner` 类，使用时先创建一个对象，再调用其方法。主要有：

- `next()`: 输入一个字符串
- `nextInt()`: 输入一个整数
- `nextFloat()`: 输入一个单精度浮点数

函数返回值就是输入的内容。详细信息可以查阅帮助文档。

控制台输出比较简单，就是两个方法：`System.out.print` 和 `System.out.println`。这两个方法都只有一个参数，可以是字符串，也可以是八种基本数据类型；区别是前者输出后不换行，后者输出后换行。

4.4.2 对话框输入输出方法

另一种输入输出方式是通过对话框。

```
//源文件应命名为 IOTest.java, 如叫其它名字编译会出错
import javax.swing.JOptionPane; //引入 JOptionPane 类
public class IOTest {
    public static void main(String[] args) {
        String name = JOptionPane.showInputDialog("What's your name?");
        //输入一个字符串
        JOptionPane.showMessageDialog(null, name); //输出一个字符串
        String str = JOptionPane.showInputDialog("Enter a integer:"); //输入一个字符串
        int y = Integer.parseInt(str); //将字符串转化为 int 型
        JOptionPane.showMessageDialog(null, y); //输出一个整数
        String Stri = JOptionPane.showInputDialog("Enter a double:");
        //输入一个字符串
        double z = Double.parseDouble(Stri); //将字符串转化为 double 型
        JOptionPane.showMessageDialog(null, z); //输出一个双精度数
    }
}
```

对话框输入输出使用 `javax.swing` 包中的 `JOptionPane` 类。其方法 `showInputDialog` 用于输入，调用它会弹出一个对话框，可以在文本框中输入一个字符串，点击确定后，这个方法就将输入的字符串返回。其方法 `showMessageDialog` 用于输出，调用它会弹出一个对话框，显示一个字符串，点击确定后，对话框关闭。

`showInputDialog` 方法都是以字符串格式输入数据，要想输入整数、浮点数等其它基本数据类型，必须将字符串类型转化为相应的类型，需要使用该类型对应的包装类提供的转化方法。每个基本数据类型在类库中都有一个对应的包装类，该类的名字是基本数据类型关键字的第一个字符大写，除了 `int` 型对应的包装类是 `Integer`。例如，`byte` 型对应 `Byte` 类，`float` 型对应 `Float` 类。每个包装类都有一个 `parseXXX` 方法，`XXX` 代表包装类的名字，能够将字符串转化为该类型。

`showMessageDialog` 方法有两个参数，简单起见可以将第一个设为 `null`，第二个为要输出的内容，可以是各种数据类型，会将数据先转化为字符串再显示在对话框中。

还应注意，要使用 `JOptionPane` 类，须先将此类引入，也就是告诉编译器去哪个包中寻找此类。引入类用的是 `import` 语句，并且在程序的开始，见程序中第一行。

第二部分 面向对象机制

现在的软件开发主要采用面向对象的思想，主要目的是提高软件生产率。在面向对象的思想出现之前，软件生产采用的是面向过程的思想。面向对象和面向过程主要的差别在于，程序的基本构件是类和对象，不再是过程或函数。例如，C 语言是面向过程的语言，C 语言的程序是由很多函数构成的；C++ 和 Java 是面向对象的语言，程序是由很多类和对象构成的。

从问题分析的角度来看，过程是一个动作，而类是一个概念，对象是概念的实例。因此，要分析一个问题时，面向过程的思想是找出问题中的动词来，而面向对象的思想是找出问题中的名词来。这是一种分析问题的方法上的改变。

学习 Java 语言学习的是面向对象的程序设计，只是面向对象的软件开发方法的一部分。面向对象的软件开发方法包括面向对象的分析、面向对象的设计、面向对象的程序开发和面向对象的维护。在实际软件开发中，分析和设计是主要的工作，程序开发只占整个工作量的不到 30%。

Java 的面向对象机制可以分三个层次来学习。第一个层次是面向对象的基本概念，包括类、对象和包。第二个层次是面向对象的核心机制，包括封装、继承和多态。第三个层次是面向对象的高级特性，是由基本概念和核心机制衍生出来的一些具体问题。

使用了类和对象的程序，可以称为基于对象的程序。应用了封装、继承、多态等核心机制的程序，才可以称为面向对象的程序。

第五章 类和对象

5.1 类和对象的概念

面向对象程序设计里最基本的概念是类和对象。

从程序设计语言发展历史来看，类可以看作是结构体（struct）的扩展。结构体是由多个数据类型组合而成的一种自定义数据类型，但结构体中只能包含数据，不能包含函数。结构体的主要缺点是造成数据和操作的分离。类是在结构体的基础上，不仅可以包含数据，而且允许定义操作这些数据的函数。类使得数据和操作定义在一起，给程序设计带来了方便。因此，类在刚出现时经常被称为“扩展的结构体”。实际上，类最早也是基于结构体来实现的，将函数以函数指针的形式保存在结构体中。

从问题分析的角度看，类是问题中涉及的概念。类中定义的变量对应于概念的属性，又叫类的属性。类中定义的函数对应于概念的功能操作，又叫类的方法或操作。属性和方法都是类的成员。

同结构体一样，类能够声明变量，所以类也可以看作一种自定义的数据类型。数据类型的两个基本方面就是数据的表示和操作。类的属性中保存数据，类中定义的方法就是数据上的操作。类又被称为抽象数据类型，使用类时，可以不用知道属性是如何保存的，操作是如何实现的，只要知道如何使用就行。

类声明出来的变量称为对象。类和对象之间的关系是抽象概念和具体事物的关系，比如“人”是一个抽象概念，而“迈克尔乔丹”是一个具体的人，所以对象又称为类的实例。反过来看，类是一个样板，以表示和操作的形式定义了一组对象的行为。

对象的三个基本要素是变量、方法和消息。对象的变量是类的属性的实例化，也就是类中包含的数据类型声明出来的变量，所以对象是占用具体的内存空间的，对象的所有变量组合起来占用一块连续的内存空间。对象的方法就是在类中定义的方法，是对象的功能单元。消息是对象之间互动的方式，是面向对象中的说法。对象 A 向对象 B 发送一条消息，实际上就是 A 调用 B 的一个方法。所以，一条消息包括三个部分：接收消息的对象、要调用的方法和方法需要的参数。

从状态转换的角度来看，变量用于保存对象的状态，方法用于改变对象的状态，消息触发状态的改变。计算机是一个基于状态的系统，由亿万个二进制位寄存器构成。每个寄存器能够保存 0 或 1 的状态，所以整个计算机有一个状态。这个状态会定期改变，改变的频率就是 CPU 的主频，比如主频是 1G 的芯片就是每秒改变 10 亿次。基于这样的硬件设计出的软件也都是状态转换系统。比如，一个 byte 型的变量占用 8 个二进制位，即 1 个字节，所以有 256 种状态，也就是有 256 种取值；运算可以使得状态发生变化，但不会超出这 256 种之外。一个对象可能占用多个内存中的字节，这些内存的状态就是对象的状态。占用的内存越大，可能的状态就越多。

面向对象的程序设计思想如下：

- 1、程序是有一堆对象组成的，对象之间通过消息联系，每个对象都知道自己能做什么，这是一种分而治之的思想，将复杂问题先分解为多个小的问题。
- 2、每个对象有一个类型，同一类对象具有相同的功能，这是举一反三的思想，解决一个问题的方法可以用来解决同类问题。
- 3、先从小的类开始编写，所有类编写完以后，组合起来就是整个程序，这是由小到大的思想，所有小问题解决后，整个大问题也就自然解决了。

分而治之、举一反三和由小到大是人们解决复杂问题的基本思想。面向对象的程序设计就是将程序看作一个复杂问题，利用上面的三个基本思想来解决。

Java 是一种完全面向对象的语言，程序完全由类来组成，变量或函数必须定义在类内，不允许在类的外面定义全局变量或函数，比如 main 函数都定义在类内，不是全局的，这一点和 C++ 不同。

学习面向对象程序设计时有个说法是：类和对象的概念，一天就能学会，但要用一生去体会。意思是这两个概念需要在编写程序的过程中去体会其含义。

5.2 类

5.2.1 定义类

定义类采用下面的语句：

```
class 类名 {  
    成员定义  
}
```

类的成员包括变量和函数，而且可以有多个。成员函数可以使用成员变量。对于成员函数来说，成员变量类似于 C++ 中的全局变量，但其作用范围仅限于类内部。成员函数也可以调用其它成员函数。一个类就划定了一个范围，在此范围内有“全局变量”和一些函数。

例如，下面的语句定义了一个名字叫 Student 的类：

```
class Student {  
    long id;          //学号  
    char gender;     //性别  
    int classID;     //班级号，注意不能用 class 作属性名  
    void changeClass(int c) { //更改班级  
        classID = c;  
    }  
}
```

Student 类有三个属性和一个函数。在函数 changeClass 中，可以使用变量 classID。

关于类和源文件的关系，Java 是这样规定的：

- 1、一个源文件中可以写多个类。
- 2、和源文件名字相同的类叫作主类，只有主类声明前面可以有 public。
- 3、一个源文件中可以没有主类。
- 4、将源文件编译后，每个类都会生成一个.class 文件。

5.2.2 构造函数

为了在创建对象时初始化对象的状态，每个类都应有一个构造函数。构造函数是在类中定义的一个特殊的函数，在创建对象时被调用，具有以下特点：

- 1、函数名和类名相同；
- 2、不需要说明返回值类型（不是没有返回值的 void 类型）；

可以为上节中的 Student 类定义构造函数，代码如下：

```
class Student{  
    long id;          //学号  
    char gender;     //性别  
    int classID;     //班级号
```

```

void changeClass(int c) { //更改班级
    classID = c;
}
Student() { //构造函数，函数名与类名相同，不需要返回值
    id = 0;
    gender = 'F';
    classID = 0;
}
}

```

一个类可以有多个构造函数，形成构造函数的重载，通过参数表的不同来区分。例如：

```

class Student{
    long id;          //学号
    char gender;      //性别
    int classID;      //班级号
    void changeClass(int c) { //更改班级
        classID = c;
    }
    Student() {       //构造函数，函数名与类名相同，不需要返回值
        id = 0;
        gender = 'F';
        classID = 0;
    }
    Student(long aID, char aGender, int aClassID) { //构造函数，与第一个参数表不同
        id = aID;
        gender = aGender;
        classID = aClassID;
    }
}

```

每个类都至少要有一个构造函数，否则无法创建对象。所以，如果程序中没有为类定义构造函数，Java 会自动为其定义一个缺省的构造函数。缺省构造函数不带任何参数。一旦程序中为类定义了构造函数，则系统不会再为其定义缺省构造函数。

5.3 对象

5.3.1 创建对象

在 Java 中，用类声明的变量不是对象，而是对象引用，简称引用。引用实际上相当于指向对象的指针，一个引用占 4 个字节。例如：

```
Student xiaoZhang; // 声明一个 Student 类型的引用
```

声明引用并不创建对象，要创建对象须使用 new 运算符，格式是：

new 类的构造函数；

例如：

new Student(); // 调用 Student 类的无参数构造函数，创建一个 Student 对象
可以调用不同的构造函数创建对象，通过给出不同的参数实现。例如：

new Student(1, 'M', 2); // 调用 Student 类的有参构造函数，创建一个 Student 对象

创建了对象后，对对象的操作都要通过引用进行。所以，仅创建对象还不行，必须让一个引用指向这个对象。采用的方法是将创建的对象的地址赋值给引用。例如：

xiaoZhang = new Student();

访问对象的成员使用“.”运算符，格式是：

对象引用.成员

例如：

xiaoZhang.id = 200328013203194L;
xiaoZhang.changeClass(1);

前面说过，发给对象的一条消息包括三个部分：接收消息的对象、要调用的方法和方法需要的参数。对于上面例子中的方法调用来说，xiaoZhang 是接收消息的对象，changeClass 是要调用的方法，1 是方法需要的参数。这三个部分又类似于自然语言句子中的主语、谓语、宾语。所以说，面向对象的方式比面向过程的方式更加接近于人的思维方式。

引用和对象指针虽然很相似，但不允许指针运算，例如下面的代码是不允许的：

xiaoZhang ++;

综合一下，Java 中使用类和对象的过程包括以下步骤：

- 1、定义类
- 2、声明引用
- 3、创建对象
- 4、让引用指向对象
- 5、使用对象成员

完整的程序如下：

```
// 程序代码应保存在文件 Student.java 中
class Student {
    long id;          //学号
    char gender;      //性别
    int classID;      //班级号，注意不能用 class 作属性名
    void changeClass(int c) { //更改班级
        classID = c;
    }
    Student() {       //构造函数，函数名与类名相同，不需要返回值
        id = 0;
        gender = 'F';
        classID = 0;
    }
}
```

```

    }
    Student(long aID, char aGender, int aClassID) { //构造函数，与第一个参数表不同
        id = aID;
        gender = aGender;
        classID = aClassID;
    }
    public static void main(String[] args) {
        Student xiaoZhang;           // ①
        xiaoZhang = new Student();    // ②
        xiaoZhang.id = 200328013203194L; // ③
        xiaoZhang.changeClass(1);     // ④
    }
}

```

程序代码应保存在文件 `Student.java` 中，编译的命令是：

`javac Student.java`

执行的命令是：

`java Student`

程序运行时，内存中的变化过程是这样的：

- 1、先执行 `main` 函数中的语句①，声明一个引用，从内存中分配 4 个字节给引用 `xiaoZhang`，如图 a。
- 2、执行语句②，先执行 `new Student()`，在内存中创建一个 `Student` 对象，如图 b；再执行赋值运算，让引用 `xiaoZhang` 指向这个对象，如图 c。
- 3、执行语句③，修改对象的 `id` 字段，如图 d。
- 4、执行语句④，调用对象的 `changeClass` 方法，修改 `classID` 字段。



图 5-1 对象创建过程

一开始学习 Java 容易忘掉语句②，特别是用惯了 C++ 的程序员，那么对象就没有被实际创建，Java 编译器会报错，信息是变量 `xiaoZhang` 没有初始化就使用。所以，使用 Java 要记住类声明出来的变量是对象引用，创建对象必须用 `new` 运算。

另外需注意的是，`new` 运算后面是对构造函数的调用，所以要和类中构造函数的定义相匹配。如果程序中没有为 `Student` 类定义无参构造函数，只定义了有参构造函数，那么语句②就会出错。但是，如果程序中没有为 `Student` 定义任何构造函数，由于此时 Java 会自动为其定义一个缺省的无参构造函数，语句②就不会出错。

5.3.2 对象回收

由于对象占用一块内存，所以当对象不再使用时，应该将其占用的内存归还给操作系统，称为对象回收或释放。

在 Java 中，对象回收是自动进行的，不用程序员去管理。Java 提供了一个垃圾回收机制，在处理器不忙时起动，检查内存中的对象哪些不再使用，自动释放无用对象所占内存。判断对象不会再被使用的标准是没有任何引用指向该对象，因为对象的访问都是通过对象引用进行的。

对象自动回收机制是 Java 语言的一大特点，一方面能有效避免内存泄漏错误，另一方面也大大减轻了程序员管理内存的工作量。如果没有对象自动回收机制，程序员需要确保释放每个不再使用的对象，这是一项很复杂的工作。比如，程序中有很多分支，要确保每个分支都释放分支前分配的对象。再比如，可能在一个函数中分配对象，在另外的函数中释放对象。只要有一个对象忘记释放，其内存就会被程序永远占用但又不使用。如果这个过程不断循环，就会使得程序占用的内存不断变大，直到将系统的所有内存都耗尽。内存泄漏的后果是很严重的，而且这类错误很难查找。对象自动回收机制能够保证所有无用对象都被释放，承担了绝大部分内存管理的工作，使程序员可以随意“丢弃”对象，大大降低了程序设计的工作量和难度，从而提高了编写程序的效率。

5.4 引用与参数传递

本节介绍引用在函数调用的参数传递中的作用。

首先要明确一点，两个引用可以指向同一个对象。比如下面的例子：

```
// 程序代码应保存在文件 Student.java 中
class Student {
    long id;          //学号
    char gender;      //性别
    int classID;      //班级号，注意不能用 class 作属性名
    void changeClass(int c) { //更改班级
        classID = c;
    }
}
```

```

    }
    public static void main(String[] args) {
        Student xiaoMing = new Student(); // ①
        Student xiaoFang = xiaoMing; // ②
        xiaoMing.gender = 'M'; // ③
        xiaoFang.gender = 'F'; // ④
        System.out.println(xiaoMing.gender); // ⑤
    }
}

```

语句②使得引用 `xiaoFang` 和 `xiaoMing` 指向同一个对象，如图所示。所以语句③和语句④修改的是同一个变量的内容，语句⑤的输出是“F”。

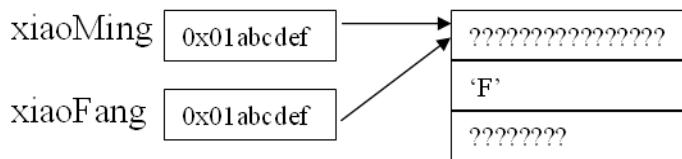


图 5-2 两个引用指向同一个对象

当引用作为函数调用的参数进行传递时，传递的是引用的值，就相当于让两个引用指向同一个对象，因此，在函数内修改了对象的内容，在函数退出后还是有效的。例如下面的例子：

```

// 程序代码应保存在文件 Student.java 中
class Student {
    long id; //学号
    char gender; //性别
    int classID; //班级号，注意不能用 class 作属性名
    void changeClass(int c) { //更改班级
        classID = c;
    }
    void modifyStudent(Student stu, long aID) {
        stu.id = aID;
    }
    void modifyInt(int x) {
        x = 100;
    }
    public static void main(String[] args) {
        Student xiaoMing = new Student(); // ①
        xiaoMing.modifyStudent(xiaoMing, 200328013203194L); // ②
        System.out.println(xiaoMing.id); // ③
        int y = 10; // ④
        xiaoMing.modifyInt(y); // ⑤
        System.out.println(y); // ⑥
    }
}

```

语句②调用函数 `modifyStudent` 时，将引用 `xiaoMing` 的值传递给函数的形参 `stu`，那么 `stu` 和 `xiaoMing` 指向同一个对象。在函数 `modifyStudent` 中，通过引用 `stu` 修改对象的 `id` 变量，相当于修改了 `xiaoMing` 指向的对象。函数退出后，对象的修改仍然有效，所以，语句③输出“200328013203194”。

如果函数调用时传递的不是引用，而是基本数据类型，那么在函数内修改变量的值不影响函数外的实参的值。语句⑤调用函数 `modifyInt`，将 `y` 的值传递给函数的形参 `x`，`x` 和 `y` 是两个变量，在函数中将 `x` 的值改为 `100`，`y` 的值不会受影响。所以，语句⑥输出“10”。

`Java` 在函数调用时只支持值传递，但同样是值传递，传递基本数据类型的值和传递引用的值带来的效果却是不一样的。因为引用相当于对象指针，所以传递引用时实际上相当于 `C++` 里的传递地址。

5.5 实例变量和局部变量

根据变量声明的位置不同，程序中的变量分为局部变量和实例变量。局部变量是在函数内声明的变量。实例变量是在函数外且类的内部声明的变量。

局部变量也被称作自动变量、临时变量或栈变量。称为自动变量是因为它占用的内存是自动管理的，不用写程序的人去关心。称为临时变量是因为它只在其作用域内起作用，出了作用域就消失了。称为栈变量是因为它占用的内存处于进程的栈上，以便支持函数的递归调用。

局部变量的作用域就是声明它的语句块。比如下面的例子：

```
int sum( int m ) {
    int y = 0;
    for (int x=0; x<=m; x++) {
        y += x;
    }
    return y;
}
```

变量 `y` 所在的语句块是函数 `sum` 的函数体，所以它的作用域是从声明起到函数体结束。变量 `x` 所在的语句块是 `for` 语句，所以他的作用域是从声明起到 `for` 语句结束，如果在 `for` 语句后访问 `x` 就会出错。函数的入口参数是局部变量，所以 `m` 是局部变量。

局部变量在声明时被创建，在作用域结束时被清除。局部变量使用之前必须初始化，否则编译时将出错。

实例变量也可以称为类的成员变量。实例变量在对象创建时被创建。因为 `Java` 在创建对象时会将对象占用的内存清 0，所以实例变量有默认的初始值，可以不用显式初始化。对于基本数据类型，其默认的初始值如下：

<code>boolean</code>	<code>false</code>	<code>Char</code>	<code>'\u0000'(null)</code>	<code>float</code>	<code>0.0f</code>	<code>double</code>	<code>0.0d</code>
<code>byte</code>	<code>(byte)0</code>	<code>short</code>	<code>(short)0</code>	<code>int</code>	<code>0</code>	<code>long</code>	<code>0L</code>

引用的默认初始值为 `null`。

两个局部变量不允许重名，两个实例变量也不允许重名，都属于变量重复定义的错误。但一个局部变量和一个实例变量可以重名。当局部变量和实例变量重名时，局部变量可以取消实例变量的作用。即缺省为局部变量；此时若要使用实例变量，应使用 `this` 关键字，`this` 关键字用于指代当前对象。例如：

```
class Student{  
    long id;  
    char gender;  
    int classID;  
    void changeClass(int classID) {  
        this.classID = classID;  
        int y = 0;  
        for (int x=0; x<10; x++) { y += x; }  
        for (int x=0; x<100; x++) { y += x; }  
    }  
};
```

`id`、`gender`、`classID`（函数外的）是实例变量，`x`、`y`、`classID`（函数的形参）是局部变量。在函数中直接使用 `classID` 指的是作为形参的 `classID`，要使用实例变量 `classID`，应该用 `this.classID`，意思是当前对象的 `classID` 变量。第一个 `x` 变量的作用域是第一条 `for` 语句，第二个 `x` 变量的作用域是第二条 `for` 语句，其作用域没有重合，所以不会有变量重复定义的问题。

5.6 类的管理

5.6.1 包

包是和类相关的一个很重要的概念，是 Java 用来对类进行分组管理用的。当程序很大时会包含很多的类，特别是由多人合作开发时，如何管理这些类就很复杂。Java 管理类的思想是分组管理，根据类的功能把类组织到不同的包中，这也是人们管理很多东西时最常采用的思想。所以，包是多个功能上联系密切的类收集到一起成为一组。

包可以嵌套，即包中可以再包含另外的包，从而形成层次。Java 对类和包的管理，类似于操作系统对文件和文件夹的管理。包相当于文件夹，类相当于文件。包中可以有类，也可以有其它包。

使用包时主要涉及三个问题：

- 1、如何将一个类放到某个包中。
- 2、类在包中的可见性，即一个类是只能被包内的类使用，还是能够被包外的类使用。
- 3、如何从包的外面使用包中的类。

下面对这三个问题分别进行介绍。

要将一个类放到某个包中，需使用 `package` 语句，格式为：

```
package 包名;
```

这条语句必须是源文件的第一条语句，在定义类之前。有了这条语句后，此源文件中定义的所有类都会被放入 `package` 后面的包中。在一个源文件中，至多只能有一条 `package` 语句，因为不允许将一个类同时放入两个包中。一个源文件中可以没有 `package` 语句，此时将类放入缺省的包，缺省的包没有名字，类似于文件系统中当前目录的概念。在不同的源文件中定义的类，可以放到同一个包中。

包中的类可以分为两种：对外可见的和对外不可见的。要使一个类包外可见，须在类的定义前面加上“`public`”描述。如果类的定义前面没有 `public`，那么此类在包外不可见，只能被同一个包内的类使用。这种类的可见性只对包外面的类有效，在同一个包内的所有类之间可以互相使用，不受类可见性的限制。

下面是一个将类放入包中的例子：

```
//程序代码应保存在 Student.java 中，否则编译出错
package school; //将此文件中定义的所有类放入 school 包
public class Student { //定义一个公开的类 Student
    long id;
    char gender;
    int classID;
    Course javaProgramming; //定义一个 Course 类的引用
    void initClass() { //更改班级
        javaProgramming = new Course(); //创建一个 Course 类的对象
        javaProgramming.name = "Java Programming";
    }
    public static void main(String[] args) {
        Student xiaoMing = new Student();
        xiaoMing.initClass();
        System.out.println(xiaoMing.javaProgramming.name);
    }
}
class Course { //定义一个不公开的类 Course
    long id;
    String name;
    float score;
}
```

代码定义了两个类：`Student` 和 `Course`，并将它们都放入一个叫作 `school` 的包中。`Student` 为包中公开的类，所以可以被不在此包中的类使用；`Course` 为不公开的类，只能被 `school` 包中的类使用。但是 `Student` 和 `Course` 在一个包中，所以 `Student` 中可以使用 `Course` 类。

编译程序使用命令“`javac Student.java`”。执行程序不能再用“`java Student`”，因为该命令表示执行缺省包中的 `Student` 类的 `main` 函数，而例子中的 `Student` 类定义在 `school` 包中，所以执行时 `java` 虚拟机会报错。

要执行程序，正确的命令应该是“`java school.Student`”。这样 `java` 虚拟机才知道是执行 `school` 包中的 `Student` 类的 `main` 函数。由此可见，有了包的概念后，类的全称应该是包层次+类名，包层次间用“.”隔开，相当于文件系统中绝对路径的概念。

但是，输入正确的命令后，`java` 虚拟机仍然报错，这是因为虚拟机找不到类的 `class` 文件。`Java` 虚拟机寻找类文件时，是将类所在的包和类文件所存放的文件夹挂钩的。所以，执行“`java school.Student`”命令，虚拟机需要寻找 `school` 包中的 `Student` 类，对应的文件是 `school` 文件夹下的 `Student.class` 文件（从当前文件夹出发）。因此，需要在当前目录下新建一个 `school` 文件夹，将 `Student.class` 和 `Course.class` 移动到该文件夹中，然后再执行“`java school.Student`”命令，虚拟机就能找到类，并执行了。

`java` 规定，和源文件名字相同的类叫作主类，只有主类可以是 `public` 的。也就是说，一个 `.java` 文件中只能有一个类声明前面有 `public`，而且这个类的名字必须和文件名相同，其它的类都不能是 `public` 的。

在包的外面使用包中的类，有两种方式。第一种方式是使用类的全称，即每次使用类时都给出类所在的包来。例如，`Date` 是 `java` 类库中 `java.util` 包中的一个用于处理日期和时间的类，要用它声明引用并创建对象，可以使用下面的语句：

```
java.util.Date today = new java.util.Date();
```

但是每次都给出类的全称，显得很啰嗦，因此，`Java` 又提供了类的“快捷方式”，可以在程序开头指出类的全称，后面使用类时只要给出类名就可以。指出类的全称使用 `import` 语句，格式如下：

```
import 类的全称;
```

例如：

```
import java.util.Date;
```

`import` 语句必须在所有类定义之前。有了这条语句，后面再出现 `Date` 类，`Java` 编译器就知道指的是 `java.util.Date` 了，相当于在 `Date` 是 `java.util.Date` 的快捷方式。

有时程序用到很多其它包中的类，对每个类都写一条 `import` 语句会很啰嗦。`Java` 为 `import` 语句提供了通配符“*”，可以一次引入包中的所有类。例如：

```
import java.util.*;
```

该语句将 `java.util` 包中的所有类都引入，即每个类都建立快捷方式，直接使用类名即可。但是“*”只能代表一个包中的所有类，并不能嵌套引入下层包。例如，下面的代码是错误的：

```
import java.*.*;
```

想用第一个“*”来代表 `java` 包中的所有包是不允许的。或者说，`import` 中只能出现一次“*”，并且是在最右边。

使用 `import` 语句可能会碰到不同包中的同名类的问题。比如下面的代码编译时会报错：

```
import java.util.*;
import java.sql.*;
.....
Date today; //编译错误
```

出错的原因是 `java.util` 包中有个类叫 `Date`，`java.sql` 包中也有一个类叫 `Date`，两条 `import` 语句为 `Date` 一个名字建立了指向这两个类的快捷方式，当使用 `Date` 这个名字时，编译器不知

道指的是 `java.util.Date` 还是 `java.sql.Date`。要解决这个问题，可以再用一条 `import` 语句明确指出后面用到的是哪个类，例如下面的代码就没有错误了：

```
import java.util.*;
import java.sql.*;
import java.util.Date; //明确指出后面用到 Date 时指的是 java.util.Date
.....
Date today; //编译错误
```

如果后面这两个类都会用到，就只能用类的全称了。

接着前面的例子，在 `school` 包中定义了 `Student` 和 `Course` 类后，现在要在 `app` 包中的 `Test` 类中使用 `Student` 类。代码如下：

```
//程序代码应保存在文件 Test.java 中，否则编译出错
package app; //将此文件中定义的所有类放入 app 包
import school.Student; // 引入 school 包中的 Student 类
public class Test { // 定义一个公开的类
    public static void main(String[] args) {
        Student stu = new Student(); // 使用 Student 类
        stu.main(args); // 调用 Student 类的 main 函数
    }
}
```

将 `Test.java` 放在当前目录，原来的 `Student.class` 和 `Course.class` 放在当前目录下的 `school` 文件夹中，否则编译 `Test.java` 时将因为找不到 `Student.class` 而报错。编译程序使用命令：`javac Test.java`。生成 `Test.class` 文件后，创建 `app` 文件夹，并将 `Test.class` 移动到 `app` 文件夹。运行程序使用命令：`java app.Test`。

包可以用于区分不同开发者写的类。如果一个程序由多人合作开发，可以先为每个开发者都分配好一个独有的包名。开发者把自己开发的类都放入自己的包中，这样通过包名就能知道类是哪个开发者写的，同时可以有效解决不同开发者间类重名的问题。

Java 的类库是放在包中管理的，最高层的两个包是 `java` 和 `javax`，下面又包含很多包。Java 类库中有很多类，但每个包中的类的数量不是很多，也便于按功能去学习掌握。为了类库的安全，Java 不允许开发者把自己的类放到类库中，也就是不允许把类放到 `java` 或 `javax` 这两个包及其嵌套的包中。

对于每个 `java` 源文件，编译器自动引入 `java.lang` 包中的所有类。因为这些类是关于 Java 语言的，是最常用的。

5.6.2 类文件的管理

上节中编译运行 `java` 程序的方式很繁琐，因为每次编译后都需要移动 `class` 文件。而在程序编写过程中，需要多次编译和运行测试，都这样移动 `class` 文件的话会带来很多不必要的工作。为此，应该改变编译运行程序的方式，以避免移动 `class` 文件。

第一种方法是将源代码文件也放到所在的包对应的文件夹中。比如上例中的 `Student.java` 文件也放到 `school` 文件夹中，此时仍然在原来的目录下编译，但不能再用“`javac Student.java`”命令，而是应该用“`javac school\Student.java`”命令，这样生成的 `Student.class` 文件就直接在 `school` 文件夹中。同样的，`Test.java` 也放到 `app` 文件夹中，用“`javac app\Test.java`”命令编译。

第二种方法是使用 `javac` 命令的“`-d`”选项，指定生成的 `class` 文件的位置。可以将源代码文件放在一个文件夹中，不用放在包名对应的相对路径中，编译时使用“`-d`”选项让生成的 `class` 文件自动放到相应的路径下。比如上例中，可以将 `Student.java` 和 `Test.java` 都放在当前目录，但是编译时分别使用命令“`javac -d . Student.java`”和“`javac -d . Test.java`”，“`-d`”后面设置的目的路径是“`.`”，表示当前目录，所以生成的 `Student.class` 和 `Course.class` 会放到`\school` 文件夹中，`Test.class` 会放到`\app` 文件夹中。

编译和运行 Java 程序的关键是让编译器和虚拟机能够找到所有用到的类。对于类库中的类，保存在 `jre/lib` 和 `jre/lib/ext` 目录下的 `rt.jar` 和其他的 `jar` 文件中，编译器和虚拟机自动会首先搜索这些位置，所以不用关心它们的位置。对于用户自己写的类（类库以外的类），前面的例子都是将它们放到执行编译/运行命令的目录下的以包名为相对路径的文件夹中，这样编译器和虚拟机才能找到。这种方式不够灵活，比如一些工具类可能在好多程序中都用到，就需要在每个程序的目录中复制一份，一旦对工具类进行了修改，需要重新复制到每个程序目录。解决此问题的办法是使用 `javac` 和 `java` 命令的“`-classpath`”（也可简写为“`-cp`”）选项，指定查找用户类文件的位置。例如：

```
java -classpath .;hello Hello;
```

用“`-classpath`”选项可以指定多个位置，用分号“`;`”隔开，编译器/虚拟机将依次从这些目录查找需要的类。在加载类的时候查找 `classpath` 是有顺序的，如果在 `classpath` 中有多个条目都有同一个名称的类，那么在较前位置的类会被加载，后面的会被忽略。所以应该注意的是，如果一个类文件有几个副本，都处于 `classpath` 的查找位置中，修改时要修改第一个查找到的位置，否则修改后的版本不会被编译器/虚拟机使用。

编译器和虚拟机查找类的方式还有差别。第一，编译器会先检查当前目录，再搜索 `classpath`，而虚拟机不会，所以在 `java` 命令使用“`-cp`”选项时，通常要把当前目录“`.`”作为第一个查找位置。第二，编译器会同时检查类的源文件，如果在相应的位置找到源文件，就比较源文件和类文件的最后更新时间，若源文件比类文件新，则说明源文件有了修改，要重新编译源文件生成类文件。

如果每次编译/执行时都加“`-classpath`”选项，会使得命令很复杂，可以将“`classpath`”设置成环境变量，编译/执行时就会自动读出该环境变量，并加到编译/执行命令中。设置的方法是：

注意，`classpath` 因为编译器和虚拟机都使用，而虚拟机不自动搜索当前目录，所以设置 `classpath` 时，为照顾虚拟机需将当前目录“`.`”设置为第一个查找位置。

`javac` 和 `java` 命令还有一些其它的选项，可以在控制台窗口中直接输入“`javac`”或“`java`”命令就能列出，并有解释。

第六章 封装、继承、多态

封装、继承和多态是面向对象程序设计的核心机制。这些机制都是为了让程序设计更加方便，改善程序的安全性、灵活性、易读性等。

6.1 封装

封装的意思是将类的成员组合在一起，并控制哪些成员是从类外部可见的，哪些是不可见的。控制成员的可见性，通过在成员声明前加上可见性描述来实现。封装的思想在生活中很常见，比如电器外面都有一个外壳，这就是一种封装。

封装的好处是：第一、可以隐藏复杂的实现细节，让使用者只要知道其对外接口就知道如何使用，不用了解内部的具体实现。第二，可以保护内部成员和结构，使它们对外部使用者不可见，也就无法破坏它们。第三，可以统一界面，让使用者通过公开的接口去使用类，而私有的内部实现可以更改，使得类和使用者之间的耦合度降低，更符合模块化程序设计的思想。

在控制类成员的可见性时，还要考虑类和类之间的关系紧密程度。一个类的成员可以对关系密切的类可见，对关系不密切的类不可见。这种设置可以方便程序设计。从亲疏远近的角度，类和类之间可以有朋友关系（同一个包中的两个类）、父子关系（一个类继承另一个类）和没有关系。因此类的成员可以有四种可见性，分别是 `private`、`friendly`、`protected` 和 `public`。这四种可见性的范围是由小到大的，`private` 可见的成员只在类内可见，`friendly` 可见的成员还对朋友类可见，`protected` 可见的成员还对子类可见，`public` 可见的成员对于所有类可见。

每种可见性描述的可见范围列在下表：

修饰符	同类	同包	子类	不同包 非子类
<code>public</code>	可见	可见	可见	可见
<code>protected</code>	可见	可见	可见	
<code>friendly</code>	可见	可见		
<code>private</code>	可见			

可见性描述加在每个成员声明的前面。`friendly` 比较特殊，并没有 `friendly` 这个关键字，而在成员声明前面不加任何可见性描述时，就表示朋友可见。例如下面的例子：

```
class Student{  
    private long id;      // 变量一般都声明为私有  
    private char gender;  // 以防止其他对象任意更改  
    protected int classID; // 有的变量可以设为子类可见  
    void changeClass(int aClassID){ // 朋友可见  
        classID = aClassID;  
    }  
}
```

```
    }
    public long getID() {return id;} // 方法一般是对外提供服务的
    public boolean setID(long aID) { // 所以声明为公共的
        if(aID>=0){id = aID;return true;}
        else {return false;}
    }
    public static void main(String[] args) {
        ...
    }
}
```

变量 `id` 和 `gender` 的可见性为 `private`, 只有类内可见。方法 `changeClass` 的可见性为 `friendly`, 同一包中的类都可见。变量 `classID` 的可见性为 `protected`, 子类可见。方法 `getID`、`setID` 的可见性为 `public`, 所有类可见。`main` 函数必须是 `public` 的, 这样才能被虚拟机从类外调用。

包中公开的类的构造函数的可见性大多数情况下应该为 `public`, 因为需要从包外的类中调用构造函数来创建对象。

6.2 继承

6.2.1 继承的概念与使用

继承是在已有的类的基础上定义新类的一种方法。作为基础的类称为基类、父类或超类, 新定义的类称为派生类或子类。子类继承父类除构造函数外的所有成员, 并可以再定义新的成员。类 A 继承类 B, 也称为类 B 派生类 A。引入继承机制的主要目的是为了代码重用, 能够以类为单位把已有代码拿来使用, 并根据需要修改。代码重用可以通过积累代码提高软件生产率。

子类与父类之间的关系是特化与泛化的关系。就是说, 子类对应的概念比父类更特殊, 反过来, 父类对应的概念比子类更广泛。如果把类能够对应的所有对象看作一个集合, 那么子类对象集合是父类对象集合的子集。因此, 判断类 A 可以作为类 B 的子类的标准是“所有 A 都是 B”说法成立。

例如, 自行车是一个概念, 山地自行车、运动自行车、双座自行车都是更特殊的概念, 所以可以作为自行车的子类, 具有自行车的所有特性。

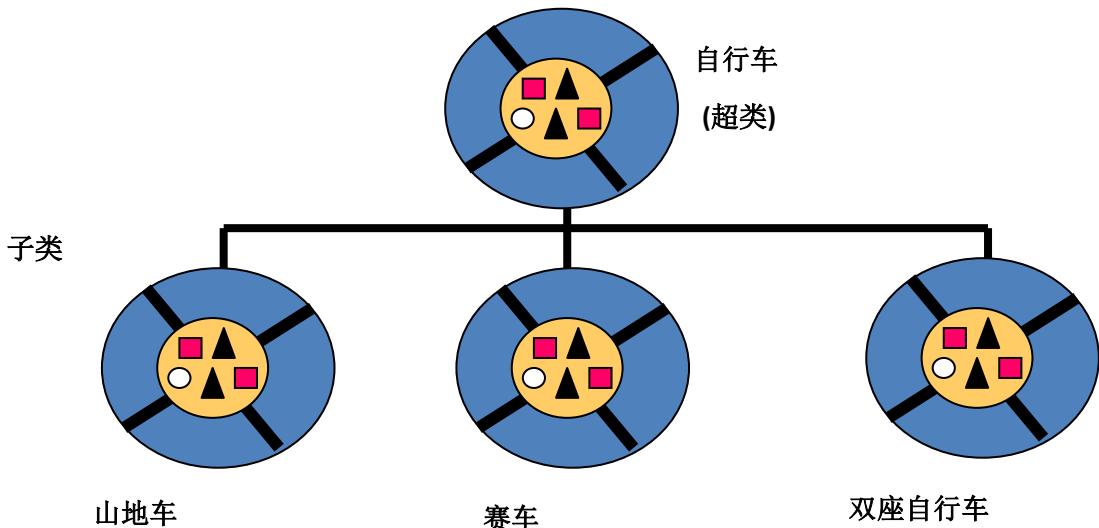


图 6-1 继承的例子

要将一个类声明为另一个类的子类，需要用 `extends` 关键字。格式如下：

```
class 子类名 extends 父类名 {
    子类成员定义
}
```

例如：

```
class Car {
    int colorNumber;
    int doorNumber;
    int speed;
    void pushBreak() {}
    void addOil() {}
}

class TrashCar extends Car {
    double amount;
    void fillTrash() {}
    public static void main(String[] args) {
        TrashCar myCar = new TrashCar();
        myCar.addOil();
        myCar.fillTrash();
    }
}
```

`TrashCar` 是 `Car` 的子类，继承 `Car` 的所有方法，所以 `TrashCar` 的对象 `myCar` 可以使用 `addOil()` 方法，虽然在 `TrashCar` 类中并没有定义这个方法。同时，`TrashCar` 类中又可以定义自己的成员 `fillTrash()`，所以 `TrashCar` 的对象 `myCar` 也可以使用 `fillTrash()` 方法。

虽然子类继承父类的所有成员，但父类的成员的可见性对其能否在子类中使用有影响。只有父类中的 `protected` 和 `public` 成员在子类中可见。对于父类中的 `private` 成员，子类继承过来，但由于可见性的原因无法使用。对于父类中的 `friendly` 成员在子类中是否可见，要看子类和

父类是否在同一个包中。

与 C++ 相比，Java 中的继承机制有三个特点：

- 1、只支持单一继承，不支持多重继承。单一继承指一个类仅能直接扩展自一个其它类，即 `extends` 关键字后面只能跟一个类名。多重继承指一个类可以有多个直接父类，子类继承多个父类的所有成员。程序中使用多重继承的机会不多，但多重继承会使得继承的复杂性大大增加，所以 Java 取消了多重继承。
- 2、只支持公开继承，不支持私有继承和受保护继承。公开继承规定父类中 `private` 可见性的成员在子类中不可见，其它可见性的成员继承到子类中后可见性不变。私有继承和受保护继承会使得父类成员继承到子类中后可见性发生变化。程序设计中用到的绝大部分继承都是公开继承，所以 Java 不再支持私有继承和受保护继承，简化了继承机制。
- 3、Java 中规定，如果类声明时没有声明父类，那么缺省为 `Object` 类的子类。这样，Java 中的所有类都以 `Object` 类为祖先，继承 `Object` 类的所有特性。`Object` 类定义了类的一些基本行为，如 `clone`、`toString`、`equals` 等，Java 中的所有类都具有这些行为。

Java 中的所有类构成一棵以 `Object` 为根的继承树，称为单根继承，可以提供一些对所有类和对象统一管理的方式。

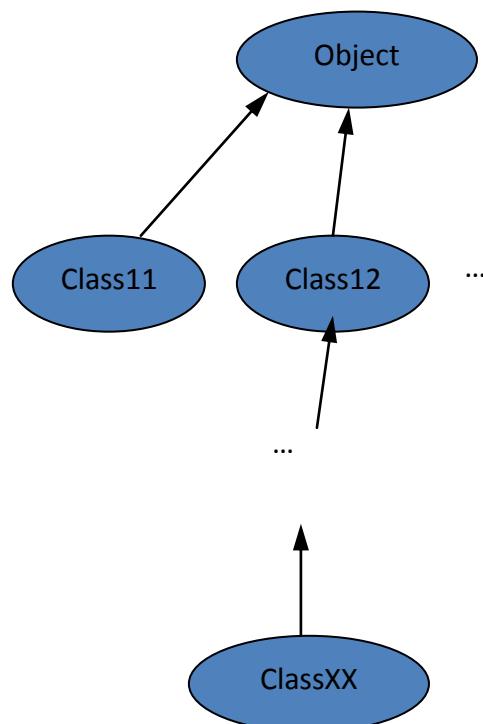


图 6-2 Java 类的继承树

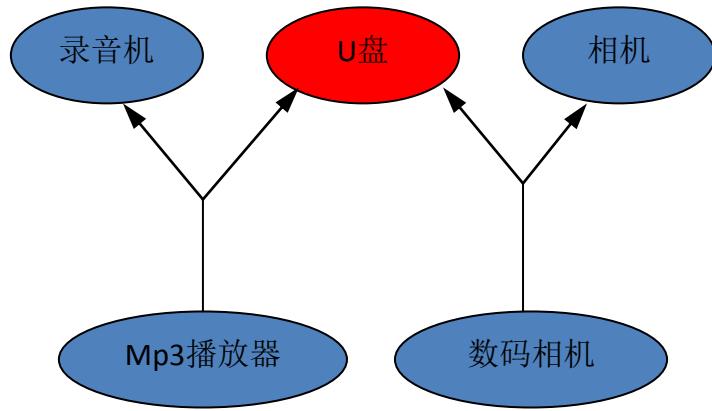


图 6-3 多重继承的例子

在类库的帮助文档中，只详细说明在本类中定义的成员。对于从父类中继承过来的成员，在父类中说明，查阅文档时不要忽略这些继承来的成员。

6.2.2 覆盖

子类继承父类的所有行为，但有时子类的某个行为与父类又不一样，此时应允许子类修改从父类继承过来的行为。覆盖就是为满足这一要求而设计的。覆盖允许在子类中对从父类继承来的方法重新定义，创建一个与父类方法具有相同的名称、返回值类型和参数表，但有不同函数实现的方法。

例如：

```

class Car {
    int colorNumber;
    int doorNumber;
    int speed;
    void pushBreak() { speed=0; }
    void addOil() {}
}

class TrashCar extends Car {
    double amount;
    void fillTrash() {}
    void pushBreak() { speed=speed-10; }
}

```

本来，子类 `TrashCar` 继承父类 `Car` 的 `pushBreak` 方法，但现在子类 `TrashCar` 中又重新定义了一个 `pushBreak` 方法，名称、返回值类型和参数表都相同，但具体实现不同。此时，子类 `TrashCar` 中定义的 `pushBreak` 方法覆盖从父类 `Car` 中继承来的 `pushBreak` 方法，从而完成了对继承过来的父类行为的修改。

覆盖的要求是两个方法的名字、返回值类型和参数表都相同，如果有一项不一样就不能形成覆盖。参数表相同指的是参数的个数和类型对应相同，不要求形参变量的名字也一模一样。如果两个方法名称相同而参数表不同，则不是覆盖，而是重载。如果两个方法名称和参数表相同，但返回值类型不同，则是方法定义错误。

覆盖针对的是在子类中可见的那些父类的方法，对于不可见的方法，如父类中的 `private` 的方法，不存在覆盖。

覆盖时还有两个要求：第一，子类的方法只能具有与父类相同或更大可见性，这是多态机制的要求；第二，子类的方法不能抛出比父类更多的异常，这是异常处理机制的要求。

有些情况下，子类的行为是对父类行为的扩展。此时也是使用覆盖，但是需要在子类的方法中调用父类中被覆盖的方法，然后再执行扩展部分的行为。调用父类的方法使用 `super` 关键字。例如：

```
class Car {  
    int colorNumber;  
    int doorNumber;  
    int speed;  
    void pushBreak() { speed=0; }  
    void addOil() {}  
}  
class TrashCar extends Car {  
    double amount;  
    void fillTrash() {}  
    void pushBreak() { super.pushBreak(); speed=speed-10; }  
}
```

子类 `TrashCar` 的 `pushBreak` 方法覆盖父类 `Car` 的 `pushBreak` 方法，但是在子类的 `pushBreak` 方法中通过“`super.pushBreak()`”语句先调用了父类的 `pushBreak` 方法，然后才执行后面的扩展部分。如果语句中没有 `super` 的话，就成了函数自己调用自己，是递归了。

覆盖和重载是两个相近的概念，它们的相同点是都涉及两个名字相同的函数。它们的区别是：

- 1、所处的类的层次不一样。覆盖涉及的是父类和子类中的两个方法，而重载涉及的是同一个类中的两个方法。
- 2、对于参数表和返回值类型的要求不一样。覆盖要求两个函数的参数表和返回值类型都必须一样，而重载要求两个函数的参数表必须不一样，返回值类型可以一样可以不一样。

6.2.3 构造函数与重载、覆盖

构造函数可以重载，在介绍构造函数时已经给出过例子。既然一个类可以有多个构造函数，那么，构造函数之间是否可以互相调用？答案是可以，用 `this` 关键字。

例如：

```

class Student{
    long id;          //学号
    char gender;      //性别
    int classID;      //班级号
    void changeClass(int aClassID) { //更改班级
        classID = aClassID;
    }
    public Student() { //构造函数 1
        this(0, 'M', 1); //调用构造函数 2
    }
    public Student(long aID, char aGender, int aClassID) { //构造函数 2
        id = aID;
        gender = aGender;
        classID = aClassID;
    }
}

```

Student 类有两个构造函数，在构造函数 1 中用 this 关键字调用了构造函数 2。

在一个构造函数中调用另一个构造函数时，this 语句必须是函数中的第一条语句。构造函数互相调用时，不能形成环，如 A 调用 B，B 调用 C，C 再调用 A。

构造函数有没有覆盖的问题？答案是没有，因为构造函数不继承。那么，子类如何初始化从父类继承过来的成员变量？特别是父类中的 private 成员变量，子类继承但无法直接访问。这是通过在子类构造函数中调用父类构造函数来完成的，用 super 关键字。

例如，定义 Student 的子类 GraduateStudent 如下：

```

class GraduateStudent extends Student {
    public GraduateStudent() {
        super(0, 'M', 1);
    }
}

```

在其构造函数中通过 super(0, 'M', 1) 语句调用了 Student 类的构造函数 2。

在调用父类构造函数时，super 语句必须是构造函数的第一条语句。这是因为对象初始化的顺序应该是先初始化父类中定义的部分，再初始化子类中定义的部分。

为保证对象中在所有类层次上所定义的变量都被正确初始化，Java 规定：如果类的构造函数中的第一条语句既不是调用其它构造函数的 this 语句，也不是调用父类构造函数的 super 语句，编译器自动插入一条 super() 语句，意思是调用父类的缺省构造函数。有了这一规定，对象初始化时，就会按由高到低的顺序依次调用继承层次上的类的构造函数。例如，类的继承关系是 A 派生 B，B 派生 C，C 派生 D，那么创建类 D 的对象时会依次调用 A、B、C、D 的构造函数中的内容。

关于继承和构造函数的规定相对较多，所以在设计类的继承时，对于类的构造函数的使用要

注意。一个比较容易让初学继承的人困惑的错误是这样的：

```
class GeometricObject{  
    private String name;  
    public GeometricObject(String aName){name = aName;}  
}  
class Ellipse extends GeometricObject{  
}
```

上面的程序在编译时会报告错误的构造函数调用。原因如下：

- 1、类 `Ellipse` 没有定义构造函数，Java 自动为其定义一个 `Ellipse()`。
- 2、构造函数 `Ellipse()` 既没有调用 `super` 也没有调用 `this`，编译器自动插入一个对父类构造函数的调用 `super()`，相当于调用 `GeometricObject()`。
- 3、类 `GeometricObject` 定义了构造函数 `GeometricObject(String aName)`，系统不会再为其定义缺省构造函数 `GeometricObject()`。

所以，编译器会将上面的程序补充为：

```
class GeometricObject{  
    private String name;  
    public GeometricObject(String aName){name = aName;}  
}  
class Ellipse extends GeometricObject{  
    public Ellipse(){  
        super();  
    }  
}
```

错误的原因是，`Ellipse` 类的构造函数中的 `super()` 语句调用的 `GeometricObject` 类的构造函数 `GeometricObject()`，但 `GeometricObject` 类中没有这样的构造函数。

修改的方法有两种：一种是为 `GeometricObject` 类定义一个无参的构造函数；另一种方法是将 `Ellipse` 类的构造函数的第一条语句改为对父类的有参构造函数的调用 `super("name")`。

6.3 多态

多态指一个概念有多种具体表现形式，或一项功能有多种实现方式。在面向对象的程序设计中表现为：父类型的引用可以指向子孙类型的对象。多态的作用在于可以将不同子类型的对象都当作父类来看，可以屏蔽子类型之间的差异，写出通用的代码。这样，针对基类写出的代码，可以适用于所有子孙类。

例如：

```
class Car {  
    int colorNumber;  
    int doorNumber;  
    int speed;  
    void pushBreak() { ... }  
    void addOil() { ... }
```

```

}

class TrashCar extends Car {      // TrashCar 是 Car 类的子类
    double amount;
    fill_trash() { ... }
}

class SportsCar {               // SportsCar 是 Car 类的子类
}

Car myCar = new TrashCar();     // Car 类型的引用可以指向 TrashCar 类型的对象
myCar = new SportsCar();       // Car 类型的引用也可以指向 SportsCar 类型的对象

```

TrashCar 是 Car 类的子类，多态性允许 Car 类型的引用指向 TrashCar 类型的对象。SportsCar 也是 Car 类的子类，所以 Car 类型的引用也可以指向 SportsCar 类型的对象。由于一个类可以有多个子类，所以一个父类型的引用可以指向多种子类型的对象。

多态性使得引用的类型和其所指向的对象的类型可能不同，此时对象的类型决定具体调用的方法，引用的类型决定能够看到哪些方法。

例如：

```

class Shape {
    public void draw(){ System.out.println("Shape.draw"); }
}

class Ellipse extends Shape{
    public void draw(){ System.out.println("Ellipse.draw"); }
    public void getCenter(){}
}

class Circle extends Ellipse{
    public void draw(){ System.out.println("Circle.draw"); }
    public double getArea(){ return 0; }
}

Shape g = new Circle(); //父类型引用指向子类型对象
g.draw();              //draw 调用的是哪个类的方法？
                      //如果 g 换成 Circle 类引用呢？
                      //如果 Circle 类不覆盖 draw 方法呢？
double d = g.getArea(); //有没有问题？

```

虽然引用 g 是 Shape 类型，但语句 g.draw() 调用的不是 Shape 类的 draw 方法，而是 Circle 类的 draw 方法，因为 g 所指向的对象是 Circle 类型的。多态时，调用哪个类的方法是由对象的类型决定的，跟引用的类型无关，这样才使得一项功能调用对于不同类型的对象有不同的实现。

同时，上例中的语句 g.getArea() 通过 Shape 类型的引用去调用 Circle 类型的对象的 getArea 方法。getArea 方法与 draw 方法的区别在于，在 Shape 类中有 draw 方法而没有 getArea 方法。此时，允许通过 g 调用 draw 方法但不允许调用 getArea 方法，编译时会报告错误。这就是引用的类型决定能够看到哪些方法的含义。之所以这样规定，是因为 Shape 类型的引用

也可能会指向 `Ellipse` 类型或 `Shape` 类型的对象，而这两种类型的对象都没有 `getArea` 方法，如果允许用 `Shape` 类型的引用调用 `getArea` 方法，此时就不知道要调用哪段代码。如果在 `Shape` 类中定义了 `getArea` 方法，就可以用 `Shape` 类型的引用调用该方法，因为其子类中必然有 `getArea` 方法，或者通过继承，或者进行覆盖。

可以将多态性理解为，大的概念的性质适用于其包含的所有更小的概念，但每个小概念在该性质上的表现形式可能不同；反过来，小的概念的性质不一定适用于包含它的更大的概念。例如，

A 对 B 说：我有一辆车。（可能是汽车、电动车、或自行车，都是车的子类）

B 可以问 A：你的车是什么颜色的？（颜色是车的属性，所有子类都具有）

但是 B 不能问 A：你的车排量多少？（排量只是汽车的属性，不是所有车都具有，A 可能只有一辆自行车）

在上例中，如果想要调用 `Circle` 对象的 `getArea` 方法，可以先将 `Shape` 类型的引用 g “还原”为 `Circle` 类型的引用，再去调用，代码如下：

```
Circle c = (Circle)g;  
c.getArea();
```

将引用 g 由 `Shape` 类型还原为 `Circle` 类型的语句类似于数据类型转换的语句。转换引用的类型时，需注意只有其指向的对象是目标引用类型或其子类型才能进行，否则的话编译时不报错但运行时出现异常。例如：

```
Shape g = new Ellipse();  
Circle c = (Circle)g; // 运行时异常
```

为支持多态性，Java 调用一个对象的方法时采用动态绑定的方式。动态绑定指在运行时确定调用方法的代码。与之对应的是静态绑定，指在编译时确定调用方法的代码。两者相比较，动态绑定更加灵活，静态绑定运行速度更快。C++采用静态绑定。

可以将动态绑定表示为如下的示意图：

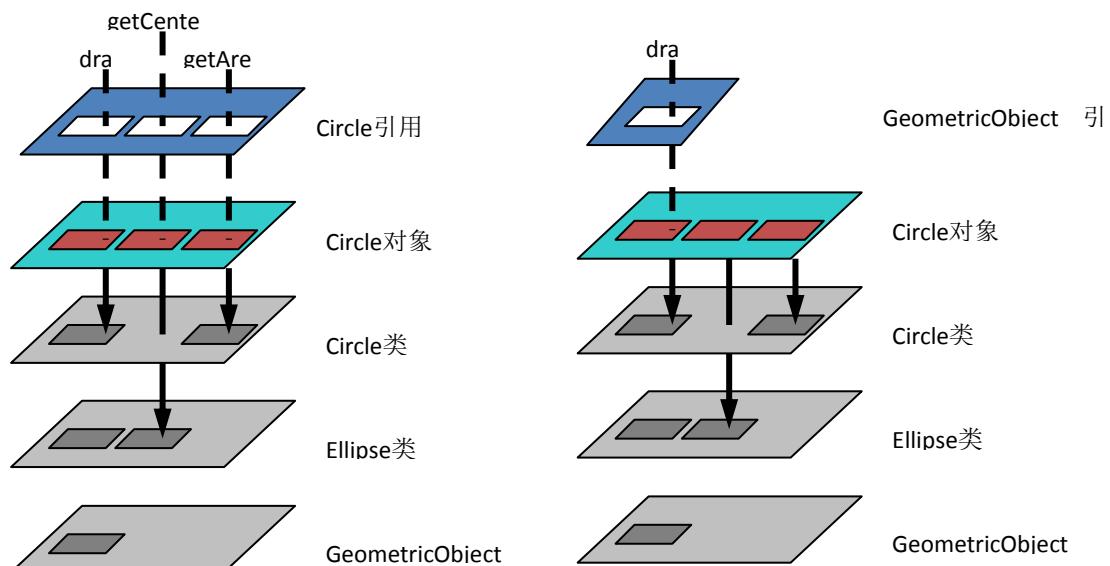


图 6-4 动态绑定示意图

在示意图中，引用是最上面的一层纸，引用的类型中定义的方法是纸上的洞，只有引用的类型中定义了该方法，从上面才能穿过方法对应的洞看到对象的方法，即引用的类型决定能够看到哪些方法。不同的引用类型，决定看到的方法不同，如图 a 和 b 的区别。对象是第二层，对象所属的类是第三层，该类的祖先依次是下面几层。对象的某个方法对应的代码要从其父类开始向下寻找，如果子类覆盖了父类的方法，那么从上面就只能看见子类的方法，看不到父类的方法了。

在 Java 中，Object 类是所有类的祖先，所以 Object 类型的引用可以指向所有对象。基于 Object 类的代码，可以适用于所有类。这是 Java 单根继承的最大的好处。

多态性是建立在继承机制之上的，没有继承机制就谈不上多态。覆盖是实现多态性的主要方式，它使得一项功能对于不同类型的对象有不同的实现。从更广义上来说，重载也是多态的一种，是同一项功能对于不同的数据类型有不同的实现。

第七章 其它特性

除了基本概念和核心机制外，面向对象程序设计中还有许多其它问题。本章将对这些方面一一进行介绍。

7.1 final 关键字

final 关键字可以用来修饰类、方法和变量，将他们“密封”。final 关键字用在可见性描述之后，类型之前。

final 类不能派生子类。如果一个类的实现细节不允许改变且不需要派生子类，那么就可以声明为 final 的。例如，类库中的 `java.lang.String`，其声明如下：

```
public final class String
```

final 方法可以继承，但不能被覆盖。将方法声明为 final 的，可以防止任何继承类修改其功能和实现。此外，final 方法的效率较高，因为 Java 在调用 final 方法时起动内嵌机制，采用把方法的实现代码全部复制到代码调用的地方的方式，省去了子程序调用的“保存现场”步骤。声明 final 方法的格式如下：

```
public class A {  
    public final void B(){...}  
}
```

final 变量相当于常量，一旦完成初始化则它在该对象整个生命周期里保持不变。final 变量可以在定义时进行初始化，也可以在运行时进行初始化，但只能赋值一次。声明常量的例子如下：

```
public final int MAX_ARRAY_SIZE = 25;
```

如果将引用类型的变量标记为 `final`, 那么该引用变量的值不能改变, 即不能再指向其它对象, 但可以改变对象的内容, 因为只有引用本身是 `final` 的。例如:

```
class Student {  
    int id;  
    public static void main(String[] args) {  
        final Student s = new Student(); // 引用 s 为 final 的  
        s = new Student() // 错误, 不允许改变 final 引用的值  
        s.id = 100; // 允许, 可以改变引用指向的对象的内容  
    }  
}
```

`final` 关键字实际上是对类、方法和变量添加了一定的限制, 这些限制主要是出于程序健壮性方面的考虑。

7.2 类的静态成员

7.2.1 静态变量

Java 不允许有全局变量, 但有时对象之间又需要有共享的变量。比如, 要编写一个 `Student` 类, 用其对象保存学生信息, 并要求所有 `Student` 对象有连续的编号。第一个创建的对象编号为 1, 第二个对象编号为 2, 以此类推。这就需要有一个所有 `Student` 对象都能访问的变量 `count`, 而且变量 `count` 在所有 `Student` 对象间共享。当一个 `Student` 对象创建时, 在其构造函数中增加 `count` 值, 并作为对象的编号, 下一个对象创建时使用增加过的值。

要声明一个在所有 `Student` 对象间共享的 `count` 变量, 可以将 `count` 声明为 `Student` 类的静态变量, 在声明时加上 `static` 关键字。`static` 关键字要加在可见性和类型之间。静态变量又叫类变量。`Student` 类的代码如下:

```
public class Student {  
    private int serialNumber;  
    private static int count = 0;  
    public Student() {  
        count++;  
        serialNumber = count;  
    }  
    public printSN() {  
        System.out.println(serialNumber);  
    }  
    public static void main(String[] args) {  
        Student s1 = new Student();  
        Student s2 = new Student();  
        s1.printSN();  
        s2.printSN();  
    }  
}
```

```
    }  
}
```

类变量在该类的所有对象之间共享，具有部分 C 语言中全局变量的功能。类也可以有可见性。如果被标记为 `public`，不需要类的实例就可以访问，如 `Math` 类中定义的常量 `PI` 和 `E`，可以直接通过类的名字去访问：

```
double a = Math.PI * r * r; // 根据半径 r 求圆的面积
```

7.2.1 静态方法

类的方法在声明时加上 `static` 关键字，该方法就变为静态方法，也叫类方法。同类变量一样，类方法不需要类的实例就可以访问，因此工具函数通常编写为类方法。例如 `Math` 类中的数学函数，一个例子如下：

```
double x = Math.sqrt(b); // 求 b 的平方根
```

`main` 函数是静态的，因为它必须在任何实例化发生前被访问，以便应用程序能运行。例如“`java Hello`”命令就是让虚拟机调用 `Hello.main(...)`。

类方法因为不需要类的实例就可以访问，所以类方法中只能访问本身的参数和类变量，不能访问类的非静态成员。因此，类方法是与对象状态没有关系的方法，方法的返回结果只和调用参数和类变量有关系。类方法中访问非静态成员会引起编译错误。例如直接在 `main` 函数中调用非静态方法：

```
public class Student {  
    private int serialNumber;  
    private static int count = 0;  
    public Student() {  
        count++;  
        serialNumber = count;  
    }  
    public printSN() {  
        System.out.println(serialNumber);  
    }  
    public static void main(String[] args) {  
        printSN(); // 静态方法中调用非静态的方法，编译错误  
    }  
}
```

构造函数不能是静态的，因为构造函数是创建对象时调用。在设计模式中，有一种工厂模式，在类函数中创建对象，代码如下：

```
public class Student {  
    private int serialNumber;  
    private static int count = 0;
```

```

public Student() {
    count++;
    serialNumber = count;
}
public static Student createStudent() { // 工厂方法
    return newStudent();
}
public static void main(String[] args) {
    printSN();
}
}

```

7.2.3 静态初始化块

静态初始化块主要用于类变量的初始化。类变量显然不能在构造函数中初始化，因为类变量只需在该类第一个对象创建之前初始化一次，不需要每个对象创建时都初始化，那样也失去了类变量的作用。如前面的例子中，类变量可以在声明时初始化，但只能执行一条语句，无法满足较复杂的需求。比如，如果希望对象编号不是从 1 开始，而是从 0 到 1000 之间的随机数开始，那么 count 的初始化需要多条语句，这些语句可以写在类的静态初始化块中。

静态初始化块是直接写在类中并且前面有 `static` 修饰的语句块。实现从随机数开始编号的代码如下：

```

import java.util.*;
class Student {
    private int serialNumber;
    private static int count;      //类变量，保存当前编号
    static { // 类的静态初始化块
        Random generator = new Random();
        count = generator.nextInt(1000); //随机数作为初始编号
    }
    public Student{
        serialNumber = count++; //每个对象创建时编号加1
    }
};

```

静态初始化块在类的第一个对象创建之前执行一次，是对初始化类变量的合适的地方。此外，也可在静态初始化块中输出类的版权信息，这样此类被使用时就会在控制台窗口中输出版权信息，且只在一开始输出一次。

如果静态初始化块前面没有 `static`，那么就是动态初始化块。动态初始化块在每个对象初始化时都执行。一个类的声明中可以包含多个初始化块，它们会按先后顺序依次执行。

类的数据字段就有三种初始化方式：

- 在变量声明时赋值
- 在初始化块中赋值
- 在构造函数中赋值

三种方式的执行顺序是按上面的顺序。

动态初始化块因为功能与构造函数类似，而且容易出错，所以不常使用。

7.2.4 静态成员的继承

子类继承父类的静态变量和静态方法，但不能将静态方法覆盖成非静态的。子类调用父类的初始化块。

前面的例子中，如果 `Student` 类有子类，子类会和父类统一编号。代码如下：

```
class Student {  
    private int serialNumber;  
    private static int count = 0;  
    public Student() {  
        count++;  
        serialNumber = count;  
    }  
    public printSN() {  
        System.out.println(serialNumber);  
    }  
}  
class GraduateStudent extends Student {  
}  
class Test {  
    public static void main(String[] args) {  
        Student s1 = new Student(); //serialNumber=1  
        GraduateStudent g1 = new GraduateStudent(); //serialNumber=2  
        Student s2 = new Student(); //serialNumber=3  
        GraduateStudent g2 = new GraduateStudent(); //serialNumber=4  
        s1.printSN();  
        g1.printSN();  
        s2.printSN();  
        g2.printSN();  
    }  
}
```

7.3 接口

Java 只支持单重继承，但是在有些情况下又有“多重继承”的需要。例如下图中的几个概念，MP3 是录音机的子类，数码相机是照相机的子类，MP3 和数码相机之间不存在继承关系，但是它们之间又有一个共性——都可以当作 U 盘使用。此时，如果按继承关系的话，MP3 应该既继承录音机的特性，又继承 U 盘的特性；数码相机既继承照相机的特性，又继承 U 盘的特性。但是 Java 不允许一个类继承自两个类，那么如何体现 MP3 和数码相机之间的共性呢？

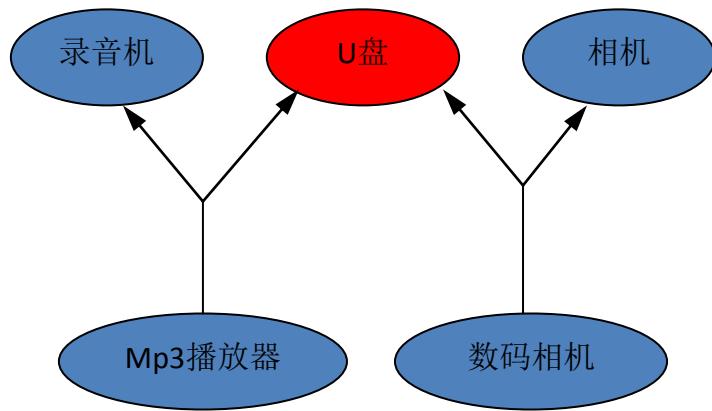


图 7-1 “多重继承”的需要

Java 采用接口来实现多重继承的需要。接口可以看作一个特殊的类，接口中的方法只给出原型声明，不给出实现。类和接口的关系是：类可以实现一个或多个接口。在上面的例子中，可以将 U 盘定义为一个接口，MP3 类继承录音机类并实现 U 盘接口，数码相机类继承照相机类并实现 U 盘接口，可以理解为 MP3 和数码相机都有 U 盘的 USB 接口。

声明接口的方式和声明类很相似，但是用的是 `interface` 关键字。接口中可以定义变量和方法，接口中的方法和变量自动都是 `public` 的。接口中的变量自动(且只能)为 `static final` 成员变量。接口中的方法只给出原型声明，不直接定义方法的内容。例如下面的例子：

```
interface Comparable {
    int compareTo(Object other);
}

class Student implements Comparable {
    private long id;
    public void setID(long aID){...};
    public int compareTo(Object otherObject) {
        Student other = (Student)otherObject;
        if(id < other.id) return -1;
        if(id > other.id) return 1;
        return 0;
    }
}
```

类实现接口需要在类定义中使用 `implements` 关键字。类要实现接口，必须在类的定义中给出接口中所定义方法的实际实现。上面的例子中，`Student` 类实现 `Comparable` 接口，那么 `Student` 类中必须给出 `compareTo` 方法的实现。

接口不是类，所以不能用接口创建对象，即不能用 `new` 运算符。例如：

```
x = new Comparable(); // 错误
```

可以声明接口类型的引用，该引用只能指向实现了该接口的对象。这也是多态性的一种形式。此时，接口的类型决定能够看到哪些方法。当用 `Comparable` 类型的接口指向 `Student` 类型的对象时，能够看到其 `compareTo` 方法，因为在接口 `Comparable` 中有定义，但无法看到 `setID` 方法。

```
Comparable x = new Student(); // 接口声明引用，指向实现了接口的对象
x.compareTo(o); // 可以，能够看到 compareTo 方法
x.setID(1L); // 不可以，不能看到 setID 方法
```

接口也可以继承，同类继承一样，也使用 `extends` 关键字。如：

```
interface ListIterator extends Iterator {
    void add(Object o);
}
```

子接口继承父接口的所有变量和方法声明，用于扩大接口的功能。实现子接口的类必须实现父接口和子接口中的所有方法。

接口用于下面一些情况：

- 声明方法，期望一个或更多的类来实现该方法。
- 决定一个对象的编程界面，而不揭示类的实际程序体。
- 捕获无关类之间的相似性，而不强迫类关系。
- 回调功能，如图形界面中的事件处理、排序(sort)方法需要用到比较(compareTo)方法等。

接口是用来实现多重继承的需要的，同时接口克服了多重继承带来的很多复杂的问题。

7.4 抽象类

抽象类是一种特殊的类，类中有一部分方法能够实现，还有一部分方法不知道如何实现。抽象类可以认为是介于普通类和接口之间的一种类。普通类的所有方法都必须实现，接口的所有方法都不用实现。

抽象类对应于问题中的抽象概念，比如说“几何形状”是一个抽象概念，虽然我们知道每个几何形状都有面积，但是在设计类时却无法在“几何形状”这个类中给出求面积的方法，因为还不知道具体是什么形状，所以“几何形状”应设计为抽象类。抽象类一般处于类的继承层次中比较高的层次。

抽象类中还不知道如何实现的方法是抽象方法，如“几何形状”类的“求面积”方法。抽象

方法只用给出声明，不用给出如何实现，即函数体。抽象方法要在子类中给出其实现，即抽象方法必须被覆盖，除非子类也是抽象类。例如“圆”是“几何形状”的子类，在“圆”类中可以给出“求面积”的方法实现。

声明抽象类和抽象方法使用 `abstract` 关键字。例如：

```
abstract class Shape {  
    abstract double getArea();  
}  
class Circle extends Shape {  
    double r;  
    double getArea() {  
        return Math.PI * r * r;  
    }  
}
```

抽象类在使用上和接口类似，不能创建抽象类的对象，因为有抽象方法没有实现；但可以创建抽象类的引用，用他们指向抽象类的子类的对象，遵从多态的规则。接口实际上相当于完全抽象类，所有方法都是抽象方法。例如：

```
Shape s = new Circle();
```

不能有抽象构造函数或抽象静态方法。一个抽象类可以不包含抽象方法，可以包含非抽象方法和变量，但一个非抽象类不能包含抽象方法。

`abstract` 关键字和 `final` 关键字在意义上是对立的：`abstract` 类必须被继承才能使用，`final` 类不能被继承；`abstract` 方法必须被覆盖，`final` 方法不能被覆盖。所以，`abstract` 和 `final` 不能同时使用。

7.5 内部类

如果一个类的定义是在另一个类的内部，那么叫作内部类。内部类中可以访问嵌套它的外面的类的成员，包括私有成员，但反过来不行。内部类和嵌套类之间没有继承关系，但内部类又能访问嵌套类的成员，这样就形成一种类似于“寄生”的关系。对于其他类来说，内部类隐藏在嵌套类内部，必须通过嵌套类才能使用。

内部类在事件监听器中应用最多，事件监听是 Java 中的事件处理机制，事件发生时由事件监听器进行相应。通常会将事件监听器类设计为窗口类的内部类，因为在事件监听器中需要操作窗口中的组件。例如：

```
public class MyFrame extends Frame{  
    Button myButton;  
    private int sum;  
    class ButtonListener implements ActionListener{  
        public void actionPerformed(ActionEvent e){  
            sum++;  
        }  
    }  
}
```

```
    }
}
}
```

内部类定义在嵌套类的内部，相当于嵌套类的成员，因此可以有 `public`、`private`、`protected` 和 `friendly` 等可见性描述，决定从嵌套类的外面能不能访问内部类。可见性描述不影响内部类使用嵌套类的成员。

内部类还可以加 `static` 描述，表示静态内部类，相当于类的静态成员。如果不加 `static` 描述，就是动态内部类。但是，内部类不能声明任何 `static` 成员，只有嵌套类可以声明 `static` 成员。因此，一个需要 `static` 成员的内部类必须使用来自嵌套类的成员。

内部类还可以被定义在方法中，相当于方法中的局部变量。此时，内部类除了可以使用嵌套类的成员外，还可以使用嵌套它的块语句中的局部变量。

内部类的名称必须与嵌套它的类不同。编译后，内部类的文件名是：嵌套类名\$内部类名.class。所以，Java 允许“\$”符号出现在类名中，主要是用于内部类。

7.6 覆盖的规则

在类的继承那一章介绍了覆盖机制，即子类可以对从父类继承来的方法进行重写。覆盖时，两个方法的名字、参数表和返回值类型必须相同。除此以外，还必须满足下面的要求：

- 覆盖方法不能比被覆盖的方法可见性差。
- 覆盖方法不能比被覆盖的方法抛出更多的异常。

如果覆盖使得方法的可见性变小，那么由于多态性使得会使用父类中被覆盖的方法的可见性去访问子类中的覆盖方法，可能会造成对私有方法的访问。例如下面的例子：

```
public class Parent {
    public void method() {}      //public 方法
}

public class Child extends Parent {
    private void method() {}    //用 private 可见性覆盖 public 可见性
}

public class UseBoth {
    public void otherMethod() {
        Parent p2 = new Child(); //父类型引用指向子类型对象
        p2.method();           //导致可能调用子类型的 private 方法!
    }
}
```

异常处理将在后面的章节中介绍。覆盖方法不能比被覆盖的方法抛出更多的异常的原因也类似，多态性使得会使用处理父类中被覆盖的方法可能抛出的异常的机制去处理子类中覆盖方法抛出的异常。如果子类中的方法抛出更多异常，针对父类方法写的异常处理机制无法处理。

7.7 instanceof 运算符与引用的类型转换

`instanceof` 运算符用于检查对象与类之间的关系，形式为：

```
if ( obj instanceof classname )
```

当 `obj` 为 `classname` 类或其子类的对象时，运算返回 `true`。否则返回 `false`。如果 `obj` 没有指向对象，出错。

一个例子如下：

```
public class Employee {};
public class Manager extends Employee {}; // Manager 是 Employee 的子类
public void method(Employee e) {
    if (e instanceof Manager) { // e 指向 Manager 对象
        } else { // e 指向 Employee 对象
        }
}
```

由于多态性，一个对象可以用不同类型的引用指向它，可以是所属的类、其父类或更高层的超类。所以，有时需要进行引用的类型转换。引用的类型转换分为两个方向：

- 向上的类型转换：将子类型的引用转换为父类型（或更高层类型）。
- 向下的类型转换：将父类型（或更高层类型）的引用转换为子类型。

向上的类型转换是可以自动进行的，因为引用所能指的类型扩大了，相当于用了一个更抽象的概念来描述一个事物。例如：

```
public class Employee {};
public class Manager extends Employee {}; // Manager 是 Employee 的子类
Manager m = new Manager();
Employee e = m; // 向上的类型转换：由 Manager 型转换为 Employee 型
```

向下的类型转换必须明确说明，因为引用所能指的类型减小了，而且向下的类型转换不一定都能进行。例如：

```
Employee e = new Manager();
Manager m = (Manager) e; // 向下的类型转换：由 Employee 型转换为 Manager 型
```

此处如果 `e` 指向的不是 `Manager` 对象，则向下的类型转换无法进行，产生错误。

```
Employee e = new Employee();
Manager m = (Manager) e; // e 指向的对象不是 Manager 型，不能转换，造成错误
为了避免这种错误，可以先用 instanceof 运算符进行判断，代码如下：
```

```
if (e instanceof Manager) { // e 指向 Manager 对象
    Manager m = (Manager) e;
}
```

向下的类型转换用于恢复对象的全部功能，因为引用的类型决定能够看到对象的哪些方法。例如：

```
class GeometricObject{
    public void draw(){...}
}

class Ellipse extends GeometricObject{
    public void draw(){...}
    public void getCenter(){...}
}

class Circle extends Ellipse{
    public void draw(){...}
    public double getArea(){...}
}

GeometricObject g = new Circle(); //多态，父类型引用指向子类型对象
double d = g.getArea();           //错误，编译时无法通过
Circle c = (Circle) g;           //向下类型转换恢复对象的所有功能
double d = c.getArea();           //正确，编译通过
```

引用的类型转换和基本数据类型的转换有相似之处，都是范围由小变大时可以自动进行，由大变小时必须明确说明。

7.8 包装类

基本数据类型不是类，所以基本数据类型的变量不能当做对象来使用。但是，有时又希望将基本数据类型的变量和对象混合在一起进行管理。为此，Java 在类库中编写了与基本数据类型一一对应的八个包装类。可以用包装类的对象来保存基本数据类型的数据，相当于将基本数据类型的变量转化为对象。

这八个包装类是： Boolean、 Byte、 Short、 Integer、 Long、 Float、 Double、 Char。

使用包装类的例子如下：

```
int x = 500;
Object o = x;      // 错误，不能将基本数据类型的变量当做对象
Integer w = new Integer(x); // 先将基本数据类型的变量转化为包装类的对象
Object o = w;      // 再统一当做对象管理
```

包装类中还提供了一些和对应的数据类型有关的操作。例如将字符串转化为整数的操作在 int 类型对应的包装类 Integer 中实现，叫作 parseInt。

```
String s = "100";
int x = Integer.parseInt(s); // x 的值为 100
```

提供包装类的主要目的是使基本数据类型也能和类一起管理，从而使用一些通用编程的好处，例如在链表等常用数据结构中保存基本数据类型的数据。

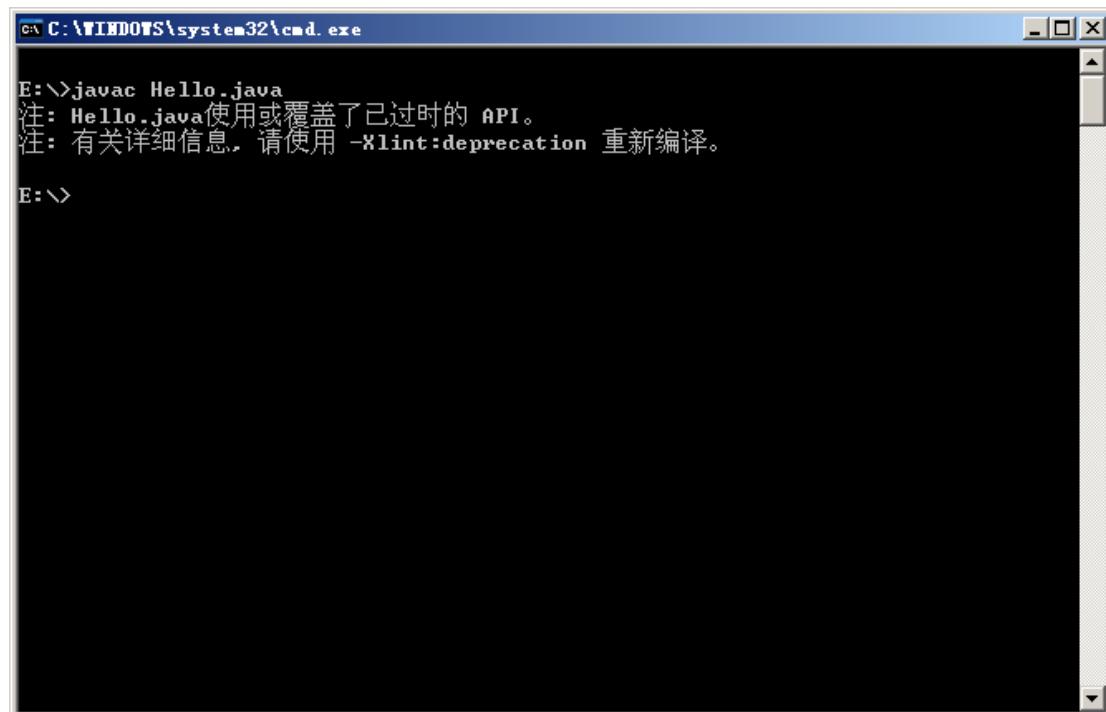
7.9 降级

随着 Java 平台版本的不断升级，类库中的类和方法也在不断更新。有些老版本中的类和方法在新版本中被新的类和方法取代了。但是，为了保持向前的兼容性，即老版本上开发的程序在新版本上也能运行，这些类又不能去掉。Java 对这些类和方法采用了“降级”的方式。被降级的类和方法不推荐使用，但不强制，所以可以继续使用。

因为被降级的类和方法可能会造成程序的兼容性变小或执行效率降低，所以新编写的程序不应使用它们。如果使用了，编译时会给出警告，例如下面的程序：

```
import javax.swing.*;
public class Hello {
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.show();
    }
}
```

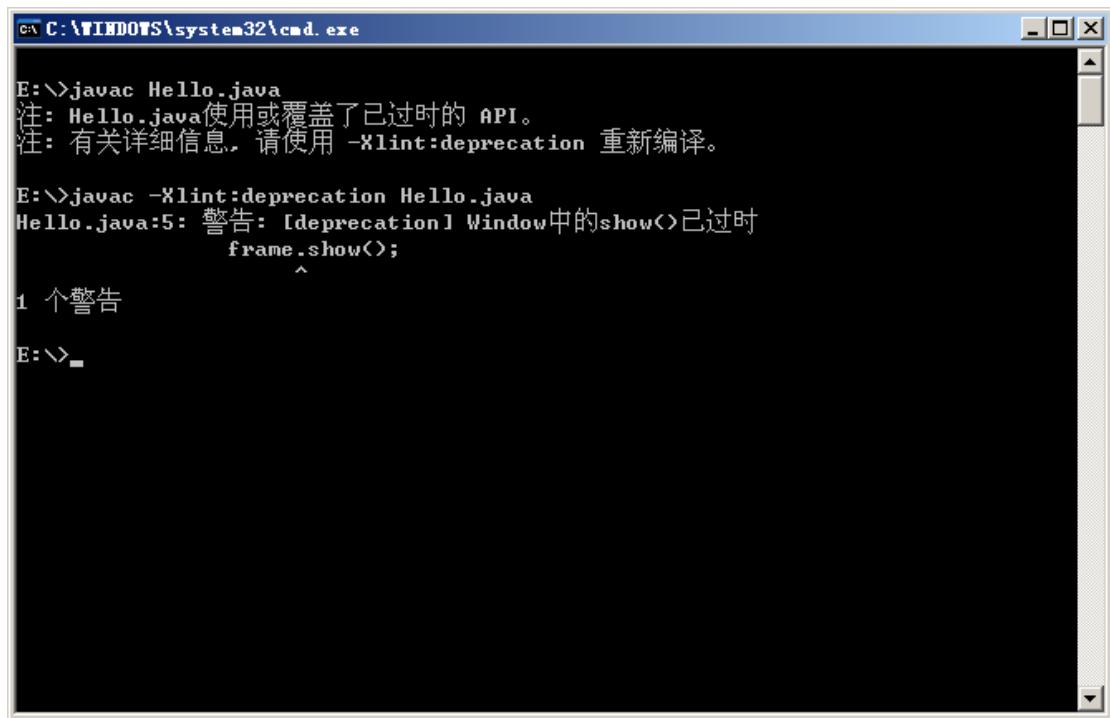
由于 `JFrame` 的 `show` 方法被降级了，所以编译时给出警告：



The screenshot shows a Windows command prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The command entered is 'E:\>javac Hello.java'. The output shows two warning messages: '注: Hello.java 使用或覆盖了已过时的 API.' and '注: 有关详细信息, 请使用 -Xlint:deprecation 重新编译。'. The window has standard Windows UI elements like a title bar, minimize, maximize, and close buttons.

图 7-2 编译时的降级警告

如果想知道详细信息，可用`-deprecation` 选项来重新编译，就会给出具体使用了哪些被降级的方法。



The screenshot shows a Windows Command Prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The command 'javac Hello.java' is run, resulting in the following output:

```
E:\>javac Hello.java
注: Hello.java 使用或覆盖了已过时的 API。
注: 有关详细信息, 请使用 -Xlint:deprecation 重新编译。

E:\>javac -Xlint:deprecation Hello.java
Hello.java:5: 警告: [deprecation] Window中的show()已过时
        frame.show();
                  ^
1 个警告

E:\>
```

图 7-3 降级信息

根据警告的提示, 查阅 JDK 帮助文档可知, `show` 方法被 `setVisible` 方法取代, 因此, 改为:

```
frame.setVisible(true);  
后警告消失。
```

类似的, 如果想将老版本的 Java 程序移植到新版本上, 只要用 `-deprecation` 选项来重新编译, 找出哪些类或方法被降级了, 替换为新的类或方法即可。当然, 不做任何改动, 老版本的程序也可直接在新版本上运行, 只是效率和兼容性可能稍差。

第八章 数组

8.1 一维数组

数组用于保存批量的数据。Java 语言在数组的使用上非常有特色, 与 C++ 差别很大。

要使用数组, 必须先声明数组变量。声明数组变量时, 声明的是一个引用, 并没有实际分配内存空间。所以, 声明数组变量时不必给出数组的大小。下面的语句是正确的:

```
char name[];           //方括号在变量名之后
float salary[], total; //salary 为数组, total 为单个变量
int[] scores;          //scores 为数组
Student[] class1Students, class2Students; //都是数组 c
```

表示数组变量有两种方式:

- 方括号在变量名之后表示该变量为数组。
- 方括号在类型之后表示后面的变量全部为数组，推荐使用这种形式，因为能体现 Java 数组的特点。

声明数组变量时不能给出数组的大小，下面的语句是错误的：

```
int num[100]; // 错误的定义数组变量的形式
```

数组变量实际上只是一个引用，还要创建数组所占用的内存空间才能使用。使用关键字 `new` 创建数组：

`float[] salary = new float[30]; // 创建一个数组，可保存 30 个 float 型数据`
和对象的创建类似，`new` 运算符为数组分配内存空间，然后需要将数组空间的地址赋给数组变量。如图所示。数组中能保存的元素个数称为数组的大小，或数组的长度。数组的大小保存在 `length` 属性中，例如要得到上面 `salary` 数组的大小可以使用 `salary.length`。

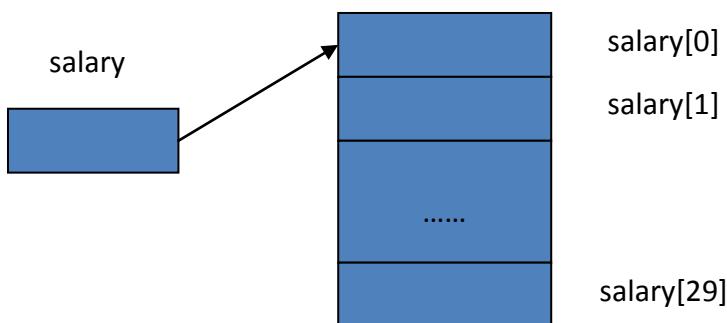


图 8-1 数组引用与数组内存

数组中每个元素相当于一个变量，通过数组变量名和下标来访问，下标可以是整数型变量。
例如：

```
float[] salary = new float[30]; // 创建一个数组，可保存 30 个 float 型数据
salary[0] = 2105.3;
int x = 10;
salary[x] = 3245.8;
```

数组下标从 0 到 `length-1`。Java 会做下标是否越界的检查，如果下标超出范围，Java 会报出异常。例如下面的例子

```
float[] salary = new float[30]; // 创建一个数组，可保存 30 个 float 型数据
int x = 30;
salary[x] = 10892.5; // 下标越界，产生异常
```

在这一点上，Java 程序的健壮性比 C++ 要好，避免了下标越界可能带来的问题。

对象引用型数组创建时只创建引用，使用前应该为引用创建对象。`Student` 为一个类，下面的代码创建一个 `Student` 数组，并为数组中前两个元素创建两个 `Student` 对象。

```
Student student = new Student[100];
student[0] = new Student();
student[1] = new Student();
```

上面的代码执行过程中的内存分配过程如下图：

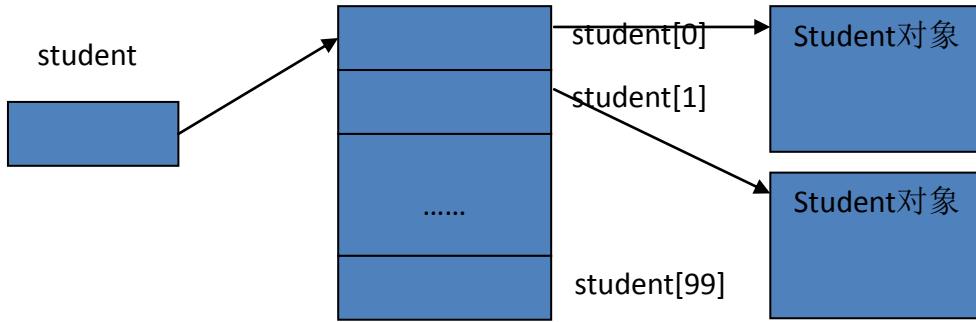


图 8-2 对象数组

每个数组元素在使用前应保证初始化。数组的初始化可以与声明分离，就像前面的例子一样，创建数组后，将每个数组元素当做一个变量进行赋值。数组的初始化也可以与声明同时进行，如：

```

float[] salary = { //基本数据类型数组声明和初始化
    1350.5F,
    2000,
}
String[] name = { "Andy", "Tom", "Jack" } // 字符串数组声明和初始化
Student[] students = { // 对象数组声明和初始化
    new Student(),
    new Student(),
    new Student(),
}

```

可以使数组变量指向一个新的数组。原数组所占空间将自动回收。

```

int[] myArray = new int [6];
myArray = new int [10];

```

数组复制可以采用对应元素依次复制的形式，用一个循环语句就可实现。数组复制还可以使用 `System.arraycopy` 方法：

```
static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)
```

方法有五个参数，第一个参数是源数组，第二个参数是从源数组的什么位置开始复制，第三个参数是目标数组，第四个参数是复制到目标数组的什么位置，第五个参数是复制几个元素。例如：

```

int[] myArray = { 1, 2, 3, 4, 5, 6 };
int[] hold = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };
System.arraycopy(myArray, 1, hold, 2, 4);
//操作后 hold 内容为{10, 9, 2, 3, 4, 5, 4, 3, 2, 1}

```

若数组为对象数组，则复制的是对象引用，而不是对象。

```

Student[] myArray = {new Student(), new Student()};
Student[] hold = new Student[10];
System.arraycopy(myArray, 0, hold, 1, 1);

```

```
//操作后 hold[1]和 myArray[0]指向同一个对象。
```

上面的代码执行后，内存如下图所示。Java 中的内存管理是很复杂的，若没有自动垃圾回收机制，内存管理将是很困难的，内存泄漏是很难避免的。

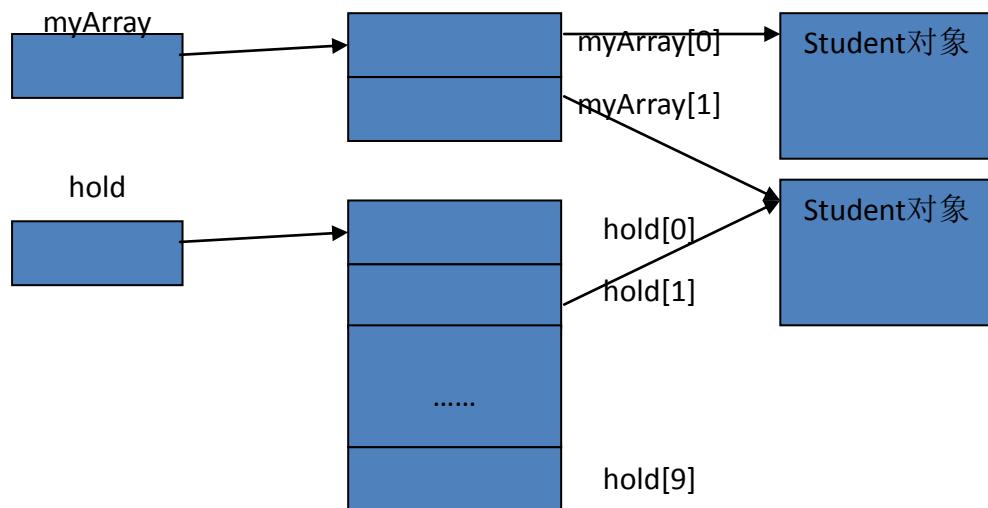


图 8-3 对象数组复制时，复制的是对象的引用

8.2 多维数组

Java 将多维数组看作“数组的数组”来管理。批量的一维数组变量放在一起，就构成二维数组。同样的道理，批量的二维数组变量放在一起，就构成三维数组。依次类推，可以构成更高维的数组。

创建一个 4×5 的整型二维数组可以使用下面的代码：

```
int[][] twoDim = new int [4][5];  
twoDim[0][1] = 10;
```

创建后的内存如下图所示：

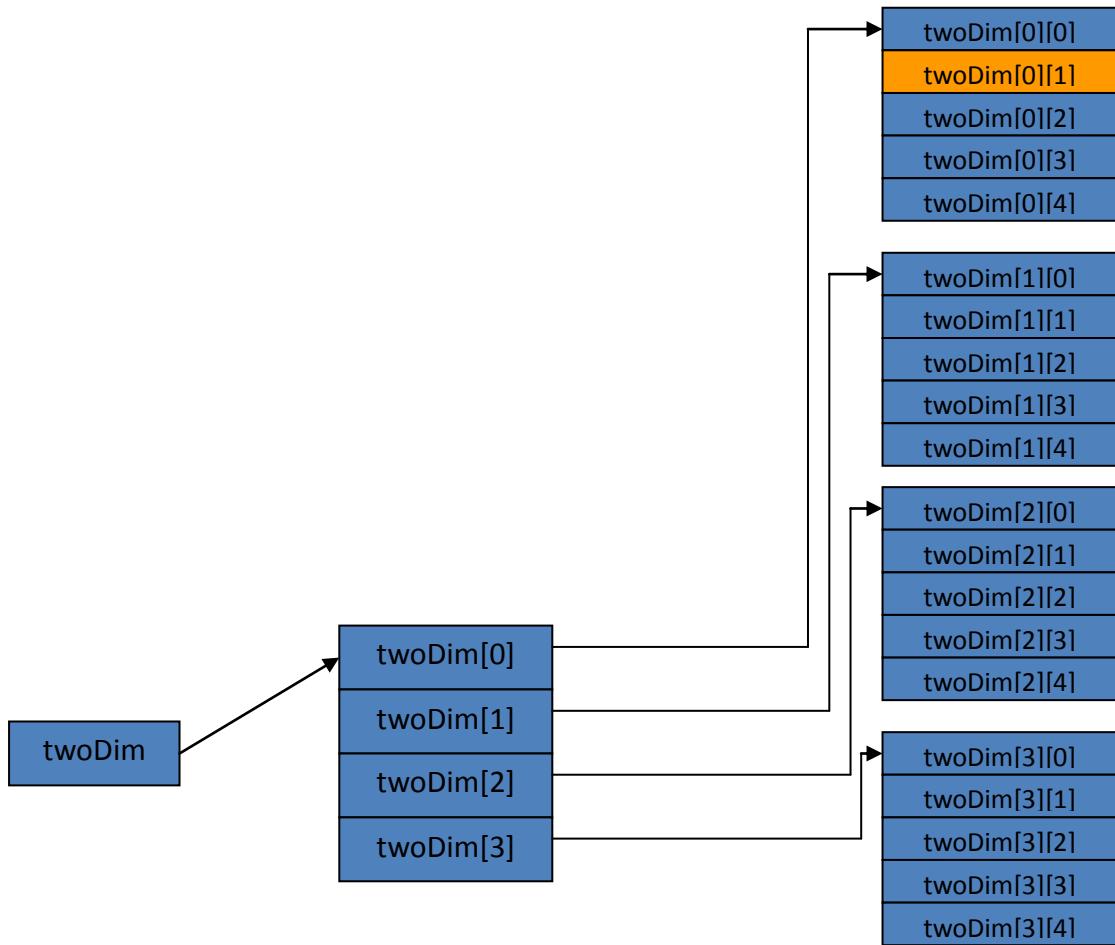


图 8-4 二维数组

如果将二维数组看作一维数组的数组，还可以先创建一个一维数组，数组中的每个元素又是一个一维数组，代码如下：

```

int[][] twoDim = new int[4][];
twoDim[0] = new int[5];
twoDim[1] = new int[5];

```

从上面的例子可以看出，Java 的多维数组是按照从高维到低维（从左到右）的顺序创建的。关于二维数组创建的一些错误的写法如下，应该避免：

```

int[][] twoDim = new int [][4]; // 非法，只给出低维，没有给出高维，不符合顺序
int[][] twoDim = new int [][]; // 非法，没有给出任何维度的大小

```

将二维数组看作一维数组的数组，还可以创建出不规则的数组。例如：

```

int[][] twoDim = new int[4][];
twoDim[0] = new int[1];
twoDim[1] = new int[2];
twoDim[2] = new int[3];
twoDim[3] = new int[4];

```

上述代码创建出的数组在内存中如下图所示：

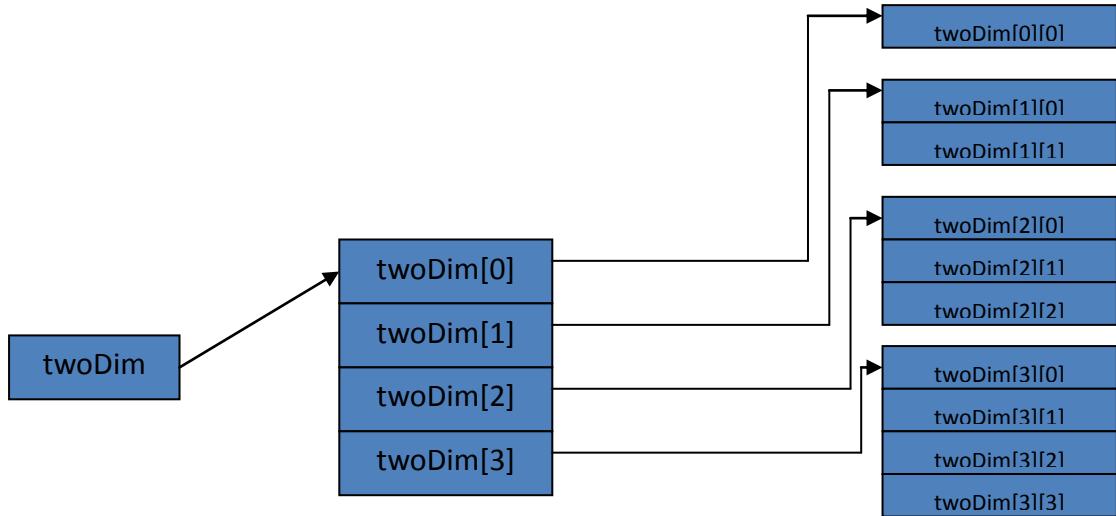


图 8-5 不规则数组

不规则数组使用时要特别注意下标不要越界，因为每一维的大小是不一样的。

第九章 常用工具类

9.1 String 类

`String` 是字符串类，用于处理文本数据。`Java` 并没有将字符串设计成像 `int` 那样的基本数据类型，也没有当成 `char` 型数组，而是设计成了一个类。这个类实际上是用 `char` 型数组保存字符串，并在此基础上设计了一些操作。

`String` 虽然不是基本数据类型，但是在使用上和基本数据类型很相似，主要体现在变量声明、字符串表示和字符串运算三个方面。

字符串的声明和赋值采用下面的形式：

```
String greeting = "Good Morning! ";
```

字符串的值是用双引号 “” 引起来的一串字符。和 `char` 型值的区别是，单个字符用单引号 ‘’ 引起来。

因为字符串相当于多个 `char` 型数据，所以 `char` 型数据的表示法在字符串中都可以使用，比如转意字符 “\”。

要对字符串进行操作，就要用到 `String` 类型上的运算。常用的有字符串的连接、求子串、判断两个字符串是否相等。

字符串的连接是将两个字符串合成一个字符串，用的运算符是 “+”。例如：

```
String greeting = "Good Morning! ";
```

```
String name = "Tom";
```

```
String helloString = greeting + name; // 结果为"Good Morning!Tom"
```

连接运算允许字符串和其它类型的数据连接，此时会先将非 String 类型自动转换为 String。如：

```
int times = 2;  
String name = "Tom" + times; //name 为"Tom2"
```

因此，在使用 System.out.println(...)方法输出时，给的参数是一个字符串，可以是用“+”连接的多个输出内容。先将其作为表达式求出多个字符串连接后的值，再输出到控制台窗口。例如：

```
int times = 2;  
System.out.println("times=" + times); // 输出 times=2
```

注意，System.out.println(...)方法的多个输出内容间不能用“，”号隔开，那样就被编译器理解为调用方法时提供了多个参数，而 System.out.println(...)方法只接受一个参数。

求子串操作是从字符串中取出一部分，需要用 String 类的 substring 方法。这个方法有两个参数，第一个参数为子串开始位置，第二个参数为子串结束位置。例如：

```
String person = "Tiger Woods";  
String name = person.substring(0,5); // name 为"Tiger"
```

判断两个字符串是否相等用 String 类的 equals 方法，不能用“==”运算符。如：

```
if ( name.equals("Tiger") ) {.....}
```

equals 方法判断两个字符串的内容是否相等，而“==”判断的是两个字符串型引用是否指向同一个字符串。其区别见下面的示意图。

关于字符串的其他操作，参见 String 类文档。

9.2 Math 类

Java 中的 Math 类属于 java.lang 包，Math 的所有方法均用 static 声明，所以使用该类中的方法时，可以直接使用包名.方法名，如：Math.sin()。Math 类包含用于执行基本数学运算的方法，如初等指数、对数、平方根和三角函数。

首先，java.lang.Math 中的四舍五入方法：

round(float a) 返回最接近参数的 int。

例如：

```
System.out.println(Math.round(5.4d));  
System.out.println(Math.round(5.5d));
```

输出结果为：

```
5  
6
```

向上取整：

ceil(double a) 返回最小的（最接近负无穷大）double 值，该值大于等于参数，并

等于某个整数。

例如：

```
System.out.println(Math.ceil(5.4d));  
System.out.println(Math.ceil(5.5d));
```

输出结果为：

```
6.0  
6.0
```

向下取整：

`floor(double a)` 返回最大的（最接近正无穷大）`double` 值，该值小于等于参数，并等于某个整数。

例如：

```
System.out.println(Math.floor(5.4d));  
System.out.println(Math.floor(5.5d));
```

输出结果为：

```
5.0  
5.0
```

使用此类中的三角函数方法时，应注意方法的灵活运用，及传递合适的参数。以 $\sin 30^\circ$ 为例，JDK 中 `Math.sin(double a)` 的源代码：

```
public static double sin(double a)  
参数： a - 以弧度表示的角。
```

因此此时首先要把参数 30 转换为是弧度表示的角。使用方法 `sin(30*Math.PI/180)`, PI 在数学方法中为 π ，而此时的 π 在角度里为 180° ，`Math.PI/180` 就为 1° ，所以 $30*Math.PI/180$ 就相当于了 30° 。例如：

```
System.out.println(Math.sin(30*Math.PI/180));
```

输出结果为：

```
0.4999999999999999
```

因为 `Math` 类中的方法直接计算出来的值均为近似值，若想得到精确值，需要使用 `Math` 中的四舍五入方法 `round()`。即为：

```
double sin30 = Math.round(Math.sin(30*Math.PI/180)*100);  
System.out.println("sin30=" + sin30/100);
```

输出结果为：

```
sin30 ° = 0.5
```

查看 `Math.round(double a)` 的源代码：

```
public static long round(double a)
```

返回最接近参数的 `long`。结果将舍入为整数：加上 $1/2$ ，对结果调用 `floor` 并将所得结果强制转换为 `long` 类型。换句话说，结果等于以下表达式的值：

```
(long) Math.floor(a + 0.5d).
```

即 $\text{Math.sin}(30*\text{Math.PI}/180) = 0.4999999999999994$ ，乘以 100 后得 49.99999999，然后 49.99999999 加上 0.5 得 50.49999999，此时再利用 `Math` 类中的另一方法 `floor()`，向下取整，得到 50.0，将 50.0 返回给 `double sin30`，然后 $\sin 30^\circ = \sin 30/100$ ，整除以 100，即得到 $\sin 30^\circ$ 的精确值 0.50。此时我们事先知道 $\sin 30^\circ$ 的结果为一个小于 1 的小数，故使用 `round()` 方法，乘以 100，四舍五入，然后再除去 100。

同理，当求反正弦时，如：`asin0.5`

```
Math.asin(0.5)*(180/Math.PI)
```

此时的 PI 即 π 为圆周率的值，所以 $(180/Math.PI)$ 得到的就是一弧度值，而后 `Math.asin(0.5)` 再乘以该一弧度的值，得到的就是弧度值，此时再利用 `round()` 方法，`Math.round(Math.asin(0.5)*(180/Math.PI)*100)`, 结果在整除以 100 即得到 `asin0.5` 的精确值为 30° 。

当我们预测到结果的整数部分为一位或 0 的小数时，我们若需取得精确值可使用 `round()` 的四舍五入方法，同时计算时需要使用先乘以 100，后整除以 100 的操作。倘若结果的整数部分为两位及其以上时，此时使用 `round()` 四舍五入方法，既不必使用先乘以 100，后才整除以 100 的操作。因为四舍五入方法四舍五入之后进的是整数位，即整数位的首位加 1。

`Math` 类在实现时利用了硬件，而不同硬件平台上实现时有可能会有微小的差别，出现计算结果的不同。要保证同样的公式在不同平台上结果完全相同，可以用类似的 `StrictMath` 类。

9.3 Class 类

`Class` 类是 `java.lang` 包中的类，其对象用来描述一个 `java` 类装入后的信息。当一个类被加载，相关的 `Class` 的对象就会自动创建。`Object` 类有个 `getClass()` 方法，所以任何对象都可以通过 `getClass()` 方法得到其所属类的 `Class` 对象。通过 `Class` 对象，可以获得类的名字、成员变量、成员方法、构造函数等信息，这种机制称为反射（reflect），`java` 提供 `java.lang.reflect` 包。

使用 `Class` 类还可以动态加载类和创建对象，相关方法包括：

- 1、用 `Class.forName(String className)` 创建 `Class` 对象。
- 2、用 `Class` 对象的 `newInstance()` 创建一个类实例（`Object` 对象），也可以通过调用类的某个构造函数来创建类实例，构造函数可以有参数。
- 3、用 `Class` 对象的 `invokeMethod` 方法可以通过方法的名字来调用方法。

使用这种动态加载类和创建对象的方法，可以编写出非常灵活的 `Java` 程序。

9.4 Random 类

`Random` 类在 `java.util` 包中，实现的随机算法是伪随机，也就是有规则的随机。在进行随机时，随机算法的起源数字称为种子数(`seed`)，在种子数的基础上进行一定的变换，从而产生需要的随机数字。

相同种子数的 `Random` 对象，相同次数生成的随机数字是完全相同的。也就是说，两个种子数相同的 `Random` 对象，第一次生成的随机数字完全相同，第二次生成的随机数字也完全相同。这点在生成多个随机数字时需要特别注意。

要生成随机数，先要创建 `Random` 对象。`Random` 类包含两个构造方法：

a、`public Random():` 使用一个和当前系统时间对应的相对时间有关的数字作为种子数，然后使用这个种子数构造 `Random` 对象。

b、`public Random(long seed):` 该构造方法可以通过制定一个种子数进行创建。

示例代码：

```
Random r = new Random();
Random r1 = new Random(10);
```

再次强调：种子数只是随机算法的起源数字，和生成的随机数字的区间无关。

`Random` 类中的方法比较简单，每个方法的功能也很容易理解。需要说明的是，`Random` 类中各方法生成的随机数字都是均匀分布的，也就是说区间内部的数字生成的几率是均等的。下面是一些常用的方法：

a、`public boolean nextBoolean()`

该方法的作用是生成一个随机的 `boolean` 值，生成 `true` 和 `false` 的值几率相等，也就是都是 50% 的几率。

b、`public double nextDouble()`

该方法的作用是生成一个随机的 `double` 值，数值介于 [0,1.0] 之间。

c、`public int nextInt()`

该方法的作用是生成一个随机的 `int` 值，该值介于 `int` 的区间，也就是 -2^{31} 到 $2^{31}-1$ 之间。

如果需要生成指定区间的 `int` 值，则需要进行一定的数学变换，具体可以参看下面的使用示例中的代码。

d、`public int nextInt(int n)`

该方法的作用是生成一个随机的 `int` 值，该值介于 [0,n) 的区间，也就是 0 到 n 之间的随机 `int` 值，包含 0 而不包含 n。

如果想生成指定区间的 `int` 值，也需要进行一定的数学变换，具体可以参看下面的使用示例中的代码。

e、`public void setSeed(long seed)`

该方法的作用是重新设置 `Random` 对象中的种子数。设置完种子数以后的 `Random` 对象和相同种子数使用 `new` 关键字创建出的 `Random` 对象相同。

此外，还有 `nextDouble`、`nextFloat` 等方法生成随机浮点数。

其实在 `Math` 类中也有一个 `random` 方法，该 `random` 方法的工作是生成一个 [0,1.0] 区间的随机小数。通过阅读 `Math` 类的源代码可以发现，`Math` 类中的 `random` 方法就是直接调用 `Random` 类中的 `nextDouble` 方法实现的。只是 `random` 方法的调用比较简单，所以很多程序员都习惯使用 `Math` 类的 `random` 方法来生成随机数字。

第三部分 应用

第十章 流与输入输出

10.1 流的概念

流是 Java 程序与外部环境交换数据的方式。这种数据交换可能发生在程序和硬盘上的文件之间、程序通过网络和其它的程序之间、程序和键盘或显示器等输入输出设备之间。这些数据交换之间有一个共同特点：数据都是以一串二进制字节序列的方式进行转移，就好像水在管道中流动一样，所以称为“流”。例如，从一个硬盘文件中读取数据时，数据是从文件流到程序中；往一个硬盘文件中写数据时，数据是从程序流到文件中。流是在文件、网络传输、键盘输入、显示器输出等基础上提炼出来的更抽象的概念，因此应用范围更广。

本章主要以文件流为例来介绍 Java 中的输入输出流。每个流在程序中实际上是一个对象，使用流就要先创建流对象，然后通过流对象的方法来操作流，进行数据读写。

Java 中的流有很多，可以进行分类。根据数据流向，可以分为输入流和输出流。输入流是数据从程序外部流向程序内部的流，如读文件、网络接收数据、键盘输入。输出流是数据从程序内部流向程序外部的流，如写文件、网络发送数据、屏幕输出。根据流的入口和出口，可以分为节点流和过滤器流。节点流是只有入口或只有出口、本身可以从一个特定的地方读写数据的流，例如磁盘或者一块内存。过滤器流则同时有入口和出口，即用一个到已存在的输入流的连接创建的。当试图从过滤器流读数据时，它会从另一个输入流读取数据并进行加工后返回给调用者。

10.2 流相关的类

Java 中的所有输入输出流类都派生自四个基类：`InputStream`、`OutputStream`、`Reader`、`Writer`。其中，`InputStream` 和 `OutputStream` 是所有字节输入输出流的基类，`Reader` 和 `Writer` 是所有 `Unicode` 输入输出流的基类。字节流只能处理单字节的字符，而 `Unicode` 流能够处理多字节的 `Unicode` 编码，比如说汉字。这四个类都是抽象类，所以不能用来创建对象。

从这四个类派生出了六十多个和流相关的类。之所以提供这么多类，Java 类库的设计者声称是用强制的方法减少编程错误。这些类全部定义在 `java.io` 包中，所以使用时要声明：

```
import java.io.*;
```

流的继承关系如下图：

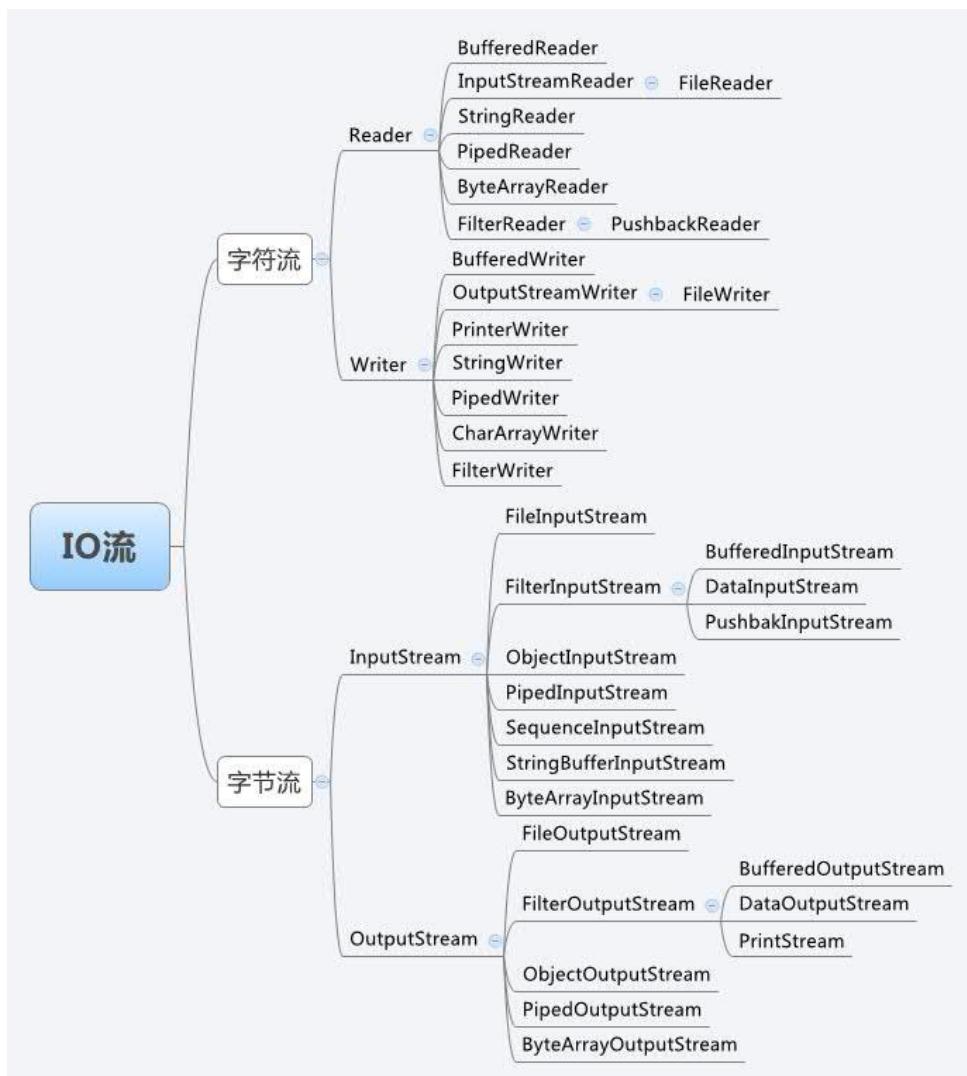


图 10-1 IO 相关的类

由 `InputStream` 派生出一些比较常用的类有：

- `FileInputStream`: 以字节流的方式读取文件。
- `BufferedInputStream`: 缓冲区输入流，为一个输入流增加缓冲区。
- `DataInputStream`: 以二进制数的方式读取数据。
- `RandomAccessFile`: 随机读取文件

由 `OutputStream` 派生出的一些比较常用的类有：

- `FileOutputStream`: 以字节流方式写文件。
- `BufferedOutputStream`: 缓冲区输出流，为一个输出流增加缓冲区。
- `DataOutputStream`: 以二进制数的方式写入数据。
- `PrintStream`: 打印输出流，用于向打印机或控制台窗口输出字符。

由 `Reader` 派生出的比较常用的有：

- `FileReader`: 以 Unicode 流的方式读取文件。
- `BufferedReader`: 为一个 Unicode 输入流增加缓冲区。

由 **Writer** 派生出的比较常用的有：

- **FileWriter**: 以 Unicode 流的方式写文件。
- **BufferedWriter**: 为一个 Unicode 输出流增加缓冲区。

10.3 数据读写方法

由于与流相关的所有类都是从四个基类派生出来的，所以大部分数据读写方法都在这四个类中定义。只要将这四个类的主要方法掌握了，再了解其他类的特有方法，就基本掌握了 Java 输入输出流的使用。本节将以文件读写为例介绍流的数据读写方法。

读入数据主要用 **InputStream** 类的方法，主要有：

- **int read()**: 读一个字节。
- **int read(byte[] b)**: 读多个字节到数组 **b** 中。
- **int read(byte[] b,int off,int len)**: 读多个字节到数组 **b** 中从 **off** 开始长度为 **len** 的位置。
- **long skip(long n)**: 跳过流中若干字节数。
- **int available()**: 返回流中可用字节数。
- **void mark(int readlimit)**: 在流中标记一个位置。
- **void reset()**: 返回到标记过的位置。
- **boolean markSupport()**: 是否支持标记和复位操作。
- **void close()**: 关闭流，解除对流资源的占用。

Reader 类的方法与 **InputStream** 类似。

读取一个文本文件的内容的代码如下：

```
// 源文件命名为 FileInputStream.java
import java.io.*;
public class FileInputStream {
    public static void main(String[] args) throws IOException { //异常
        FileInputStream in = new FileInputStream("FileInputStream.java"); //创建流
        int length = in.available(); //获得流中字节数(文件长度)
        char c;
        for(int i = 0; i < length; i++) {
            c = (char)in.read(); //读取一个字符
            System.out.print(c);
        }
        in.close(); //关闭流
    }
}
```

上述代码应保存在 **FileInputStream.java** 文件中，编译运行后会读取 **FileInputStream.java** 文件中的内容并输出到屏幕上。程序中首先创建一个 **FileInputStream** 类型的输入流，构造函数的参数是要读取的文件的名字，该文件在运行时应位于当前目录下。文件输入流创建后，调用 **available** 方法获取文件长度，然后用 **for** 循环依次调用 **read** 方法读出流中的每个字符，并打印到屏幕。最后调用 **close** 方法关闭流，解除对文件的占用。

`main` 函数后面多了“`throws IOException`”，意思是将所有异常交给虚拟机处理。异常指的是程序运行时遇到的非正常的情况，比如文件不存在或文件读写错误等。由于这些异常情况是不可能完全避免的，所以必须进行处理，而异常处理机制此前还未介绍，所以这里采用了一种最简单的方式，只要在 `main` 函数后面加上“`throws IOException`”，就不用关心异常的处理了。本章后面的例子也是这样做的。异常处理机制将在下一章介绍。

运行时，对于注释中的汉字无法正确输出，因为用的是字节输入流，无法处理多字节编码。

输出数据主要用 `OutputStream` 类的方法，主要有：

- `abstract void write(int b):` 将一个字节输出到流中。
- `void write(byte b[]):` 将数组中的数据输出到流中。
- `void write(byte b[], int off,int len):` 将数组 `b` 中从 `off` 开始 `len` 长度的数据输出到流中。
- `void flush():` 将缓冲区中的数据强制送出。
- `void close():` 关闭流。

`Writer` 类的方法与 `OutputStream` 类似。

输出二十六个英文字母到一个文件的代码如下：

```
// 源文件命名为 FileOutputStream.java
import java.io.*;
public class FileOutputStream {
    public static void main(String[] args) throws IOException { //异常
        FileOutputStream out = new FileOutputStream("abc.txt"); //创建流
        for(int i = 'a'; i <= 'z'; i++)
            out.write(i); //写入一个字节
        out.close(); //关闭流
    }
}
```

上述代码编译运行后，会将 `a` 到 `z` 的二十六个字母写入文件 `abc.txt`。程序中先创建一个 `FileOutputStream` 类型的文件输出流，构造函数的参数是要写入的文件的名字。然后用一个 `for` 循环依次调用 `write` 方法，将数据写入流中。最后调用 `close` 方法关闭流，确保数据写入到了文件。

在读写文件时，可以加上缓冲区。一方面可以减少硬盘访问次数，提高效率，另一方面也可以使用缓冲区流的一些特有方法，比如一次读取一行的 `readLine` 方法。加上缓冲区的方法是在原有文件输入输出流的后面再接上缓冲区输入输出流，如下图所示：

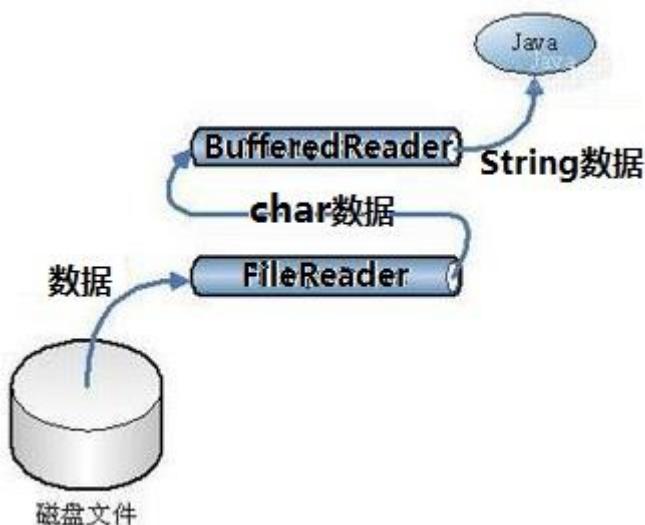


图 10-2 流的嵌套

带缓冲区读文件的代码如下：

```
import java.io.*;
public class FileInputStream {
    public static void main(String[] args) throws IOException {
        FileInputStream in = new FileInputStream("FileInput.java");
        BufferedInputStream bufIn = new BufferedInputStream(in); // 连接 in
        int length = bufIn.available();
        for(int i = 0; i < length; i++) {
            char c = (char)bufIn.read();
            System.out.print(c);
        }
        bufIn.close();
        in.close();
    }
}
```

程序在文件输入流 `in` 的后面又接上了一个缓冲区输入流 `bufin`，具体方法是构造 `bufin` 对象时将 `in` 作为构造函数的参数。同时应注意，最后关闭流的时候是先关闭 `bufin` 再关闭 `in`，即后接的流先关闭，否则会出错。

带缓冲区写文件的代码如下：

```
import java.io.*;
public class FileOutputStream {
    public static void main(String[] args) throws IOException {
        FileOutputStream out = new FileOutputStream("abc.txt");
        BufferedOutputStream bufOut = new BufferedOutputStream(out);
        for(int i = 'a'; i <= 'z'; i++)
            bufOut.write(i);
        bufOut.close();
        out.close();
    }
}
```

```
    }  
}
```

程序在文件输出流 `out` 的前面又接上了一个缓冲区输出流 `bufOut`, 具体方法是构造 `bufOut` 对象时将 `out` 作为构造函数的参数。最后关闭时也是先关闭 `bufOut` 再关闭 `out`。

10.4 标准流

每个 Java 程序运行时, 虚拟机都会为其创建标准输出流、标准输入流、标准错误流三个流, 这三个流对象保存在 `System` 类的静态变量中。

- `System.out`: 把输出送到缺省的显示(通常是显示器)。
- `System.in`: 从标准输入获取输入(通常是键盘)。
- `System.err`: 把错误信息送到缺省的显示。

每当 `main` 方法被执行时, 就自动生成上述三个对象, 所以能够直接使用。

`System.out` 对象属于 `PrintStream` 类, 输出数据大都用的是 `print` 方法或 `println` 方法, `print` 方法输出后不换行而 `println` 方法输出后换行。要了解 `System.out` 对象有哪些方法可用, 只要查 `PrintStream` 类的文档。`PrintStream` 类的主要方法有:

- `void print(...)`: 对于各种输入参数类型都有对应的重载函数, 所以能输出各种形式的数据。如: `print(String s); print(char c);`
- `void println(...)`: 类似于 `print` 方法, 但在输出的最后自动换行。
- `void write(byte[] buf, int off, int len)`: 写入多个字节。
- `void write(int b)`: 写入一个字节。
- `void close()`: 关闭流。
- `void flush()`: 将缓冲区中的数据强制送出。

再如, 从键盘读入一个字符串要用到 `System.in`, 代码如下:

```
InputStreamReader ins = new InputStreamReader(System.in);  
BufferedReader br = new BufferedReader(ins);  
String str = br.readLine(); // 读取一行
```

上面一段代码在节点流 `System.in` 后面接上了两个过滤器流 `ins` 和 `br`, 调用 `br` 的 `readLine` 方法时, 该方法会再调用 `ins` 的数据读取方法, 而该方法又会调用 `System.in` 的数据读取方法。如下图所示:

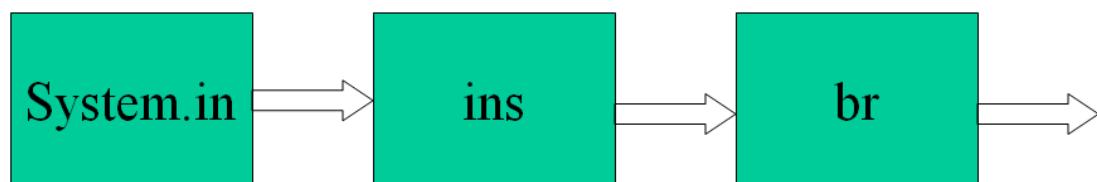


图 10-3 从标准输入流读入数据

10.5 示例——抽奖器

本节将以抽奖器程序为例，演示流输入输出的方法。抽奖器程序的功能是从一个文本文件中读取参与抽奖的全部对象的数据，然后从中随机选择出一定个数来，最后将选出的对象输出到屏幕并保存到文件中。抽奖器可以用于抽奖、课堂随机点名或作业抽查等。抽奖器的数据流程如下：

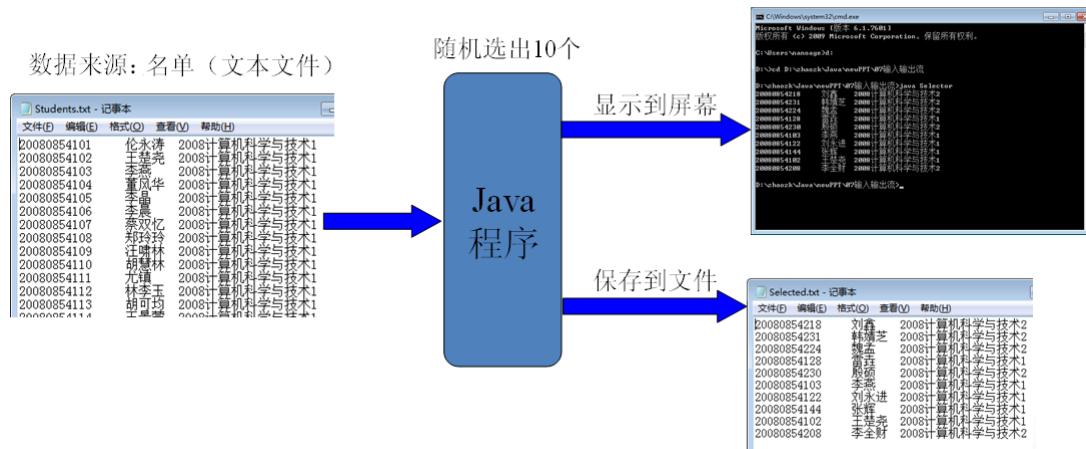


图 10-4 抽奖器数据流程

设计抽奖器程序的关键点有三个：

- 1、文件读写：如何从文件中读出数据，如何将数据写入文件。
 - 2、数据保存：候选对象信息如何在文件和程序中保存。
 - 3、随机选取：如何从一组数据中随机选择一个。

文件读写的方法在本章前面已经介绍过，应采用文件流。因为数据中有汉字，所以应该用 Unicode 流，即 `FileReader` 和 `FileWriter` 两个类。在文本文件中，每个候选对象占一行，要一次读取一个候选对象，就要一次从文件中读取一行，这就需要使用缓冲区流，即 `BufferedReader`，它有 `readLine` 方法。文本文件可以用 Windows 的记事本打开，内容如下图：

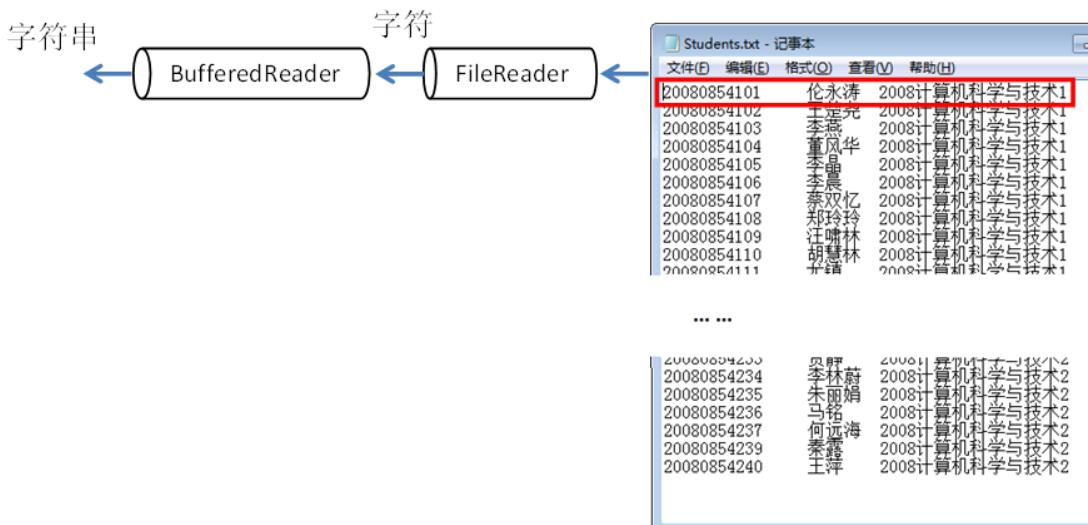


图 10-5 读取一行的方法

每个候选对象的数据都是文本描述，所以可以用一个字符串（String）保存。程序中要保存批量的候选对象，所以应该用 `String` 数组保存。输出时，也是作为一个字符串输出，占一行。这样处理，实际上并不关心字符串中内容为什么，也就是不关心候选对象的描述格式。

随机选取使用随机数发生器来完成。随机数发生器是 Java 中的一个类 `java.util.Random`，它是一个用于产生伪随机数的类，可以产生各种类型的随机数，如 `int`, `float`, `double`, `boolean` 等。使用方法如下：

```

Random rand = new Random(); // 创建一个随机数发生器对象
int j = rand.nextInt(100); // 产生一个 0-99 之间的整数
boolean b = rand.nextBoolean(); // 产生 true 或 false

```

要产生随机数就要先创建一个随机数发生器对象，然后调用它产生各种类型的随机数的方法。

产生伪随机数利用了数学中的混沌函数，随机数序列是迭代生成的。即初始值为 x_1 ，那么第二个值是 $x_2=f(x_1)$ ，第三个值是 $x_3=f(x_2)$ ，依次类推。混沌函数的特点是函数值与自变量间几乎没有规律，所以在一个随机数序列中，如果不知道函数 f 几乎无法预测下一个数。但是如果两个随机数序列的初始值相同，那么这两个序列就将完全相同。为避免程序两次运行产生相同的随机序列，初始值 x_1 采用 `Random` 对象构造时的系统时钟值生成，精确到毫秒。因两次运行的时刻肯定不同，所以产生的随机数序列也将不同。

抽奖器程序的三个关键点都有方法解决，下面开始编写程序。将抽奖器类命名为 `Selector`，并编写 `main` 函数框架。代码如下：

```

public class Selector {
    public static void main(String[] args) throws IOException {
    }
}

```

首先编写从文件读入数据的代码，因为不读入数据其它部分都无法运行。代码如下：

```

public class Selector {

```

```

public static void main(String[] args) throws IOException {
    //读入学生数据
    FileReader fin = new FileReader("Students.txt");
    BufferedReader in = new BufferedReader(fin);
    String line = in.readLine(); // 读取一行
    while(line != null) { // 如果读取结果为 null 表示文件结束
        System.out.println(line); // 暂时先打印读出的结果，检验代码正确性
    }
    line = in.readLine();
    in.close();
    fin.close();
}

```

编写完这一部分后，可以编译运行，看是否正确。正确的话，应该能将文件中的所有候选对象打印到屏幕上。

接着编写用 `String` 数组保存数据的代码。由于事先知道文件中候选对象个数不超过 100 个，所以用一个大小为 100 的数组就足以保存下所有候选对象。同时用变量 `count` 记录实际读入了多少个候选对象。代码修改后如下：

```

class Selector {
    public static void main(String[] args) throws IOException {
        String[] students = new String[100]; // 创建 String 数组
        int count = 0; // 用变量 count 记录候选对象个数
        //读入学生数据，保存到 students 数组中
        FileReader fin = new FileReader("Students.txt");
        BufferedReader in = new BufferedReader(fin);
        String line = in.readLine();
        while(line != null) {
            students[count] = line; // 数据保存到数组中
            count++; // 个数加 1
            if(count>=100)break; // 如果超出数组容量就停止
            line = in.readLine(); // 读入下一行
        }
        in.close();
        fin.close();
    }
}

```

再添加随机选取部分的代码。若要从 `count` 个候选对象中随机选出一个，只要用随机数发生器产生一个 0 到 `count`（包括 0 但不包括 `count`）之间的整数 `r`，`students` 数组中下标为 `r` 的元素即为选出的。如果要选出多个，则需要重复上述选取的过程。同时为避免多次选中同一对象，在对象选中后要将其从数组中移除。移除的方法是将该对象拿走，然后用数组中最后一个元素挪到拿走后留下的空位上。代码如下：

```
int selectedCount = 10;
```

```

Random rand = new Random();
for(int i = 0; i < selectedCount; i++) {
    int j = rand.nextInt(count); //从学生中随机挑选一个
    line = students[j]; //选出的学生
    System.out.println(line); //打印输出
    students[j] = students[count-1]; //将该学生去除
    count--; //总人数减 1
}

```

最后再将选出的对象保存到文件。可以在上面的选取循环中，直接加入写文件的语句。代码如下：

```

FileWriter fout = new FileWriter("Selected.txt"); // 创建文件输出流
BufferedWriter out = new BufferedWriter(fout); // 接上缓冲区输出流
for(int i = 0; i < selectedCount; i++) {
    int j = rand.nextInt(count);
    line = students[j];
    System.out.println(line);
    out.write(line,0,line.length()); // 将字符串写入文本文件
    out.newLine(); // 换行
    students[j] = students[count-1];
    count--;
}
out.close(); // 关闭缓冲区输出流
fout.close(); // 关闭文件输出流

```

至此，抽奖器程序就编写完成了。运行时，它从 `students.txt` 文件中读取候选对象数据，然后随机选择 10 个，将它们打印到屏幕上，并将它们保存到文件 `selected.txt` 中。编写程序时，要采用由小到大一块一块添加的增量式开发方法，可以避免出现过多错误。完整的程序代码如下：

```

import java.io.*;
import java.util.*;
class Selector {
    public static void main(String[] args) throws IOException {
        String[] students = new String[100];
        int count = 0;
        //读入学生数据，保存到 students 数组中
        FileReader fin = new FileReader("Students.txt");
        BufferedReader in = new BufferedReader(fin);
        String line = in.readLine();
        while(line != null) {
            line = line.trim(); //去除字符串中前后空格
            if( line.length() > 0 ){ //检查是否是空行
                students[count] = line;
                count++;
            }
        }
    }
}

```

```

        if(count>=100)break;
    }
    line = in.readLine();
}
in.close();
fin.close();

//随机选出所需人数，并保存到文件中
int selectedCount = 10;
Random rand = new Random();

FileWriter  fout = new FileWriter("Selected.txt");
BufferedWriter  out = new BufferedWriter(fout);
for(int i = 0; i < selectedCount; i++) {
    int j = rand.nextInt(count);
    line = students[j];
    System.out.println(line);
    out.write(line,0,line.length());
    out.newLine();
    students[j] = students[count-1];
    count--;
}
out.close();
fout.close();
}
}

```

10.6 抽奖器的改进

上节中编写的抽奖器能完成最简单的功能，还有许多可以改进的地方。本节将从以下几个方面对其进行改进：

- 1、自动生成文件名：第一次抽奖结果保存在 no1.txt 中，第二次结果保存在 no2.txt 中，...
- 2、调整抽取的个数：将要抽取的个数作为程序运行时的参数
- 3、数据格式化：将每个候选对象数据不再笼统的当作字符串处理，而是切分出不同部分来。

自动生成文件名实际上采用了下面的规则：

- 如果不存在 no1.txt 文件，则结果为 no1.txt；
- 如果不存在 no2.txt 文件，则结果为 no2.txt；
-

判断一个文件存不存在可以使用类库中 java.io 包中的 File 类，用文件名为参数构造该类的一个对象，就对应于硬盘上的一个文件，通过该对象的 exists() 方法可以判断文件是否存在。同时，还可以用 File 对象直接作为 FileWriter 类构造函数的参数。所以，此部分代码如下：

```

int no = 0;
File outf;
do {
    no++;                                // 文件编号
    outf = new File("No"+no+".txt");        // 构造 File 对象
} while(outf.exists());                  // 判断文件是否已经存在
FileWriter fout = new FileWriter(outf);   // File 对象作为 FileWriter 类构造函数的参数

```

每次要抽取的个数可以通过程序启动参数来输入，这些参数会作为 main 函数的参数传入。例如，如果程序启动时使用的命令是：

```
java Selector 15
```

那么，main 函数后面的参数 args 是一个字符串数组，在 args 中有一个元素，其内容为字符串“15”。可以将抽奖器设计为，如果输入抽取个数就按该数量抽取，如果不输入就按默认情况抽取 10 个。代码如下：

```

if( args.length > 0 ) selectedCount = Integer.parseInt(args[0]);
// 此语句加入 selectedCount 定义之后

```

数据格式化部分主要是对文件中一行上的字符串内容进行分析，切分出各个字段来，可以为程序增加一些按字段排序和抽取的功能。一行字符串切分成多个字段后，就需要保存在一个对象中，为此需要设计该对象所属的类，这里命名为 Student。该类定义如下：

```

class Student {
    private String id;                      //学号
    private String name;                     //姓名
    private String department;               //学院
    public void parseStudent(String str) {...} //字符串解析
    public String toString() {...}           //生成字符串
}

```

在 Student 类中，每个字段对应用一个变量保存。此外，该类还有两个方法，parseStudent 方法将一个字符串按分隔符切分成几个字段，分别放到相应的变量中保存，toString 方法再将多个字段合并为一个字符串，用于输出。

parseStudent 方法可以用类库中字符串切词类 StringTokenizer，该类提供的方法可以根据空格符等分隔符将字符串切分成多个词。代码如下：

```

public void parseStudent(String str) {
    int tokenCount;
    StringTokenizer t = new StringTokenizer(str); // 用要切分的字符串构造对象
    tokenCount = t.countTokens();                // 该字符串包含几个部分
    id = t.nextToken();                        // 第一部分为学号
    name = t.nextToken();                      // 第二部分为姓名
    department = t.nextToken();                // 第三部分为班级
}

```

toString 方法代码如下：

```
public String toString() {
```

```

String s = id + " " + name;
for(int l = s.length(); l < 21; l++) s += " " ; //对齐，引号中为两个空格
return s + department;
}

```

Student 类编写好以后，再来修改抽奖器的代码，主要修改三处地方。

第一，数据不再保存在 String 数组中，而是保存在 Student 类型的对象数组中：

```
Student[] students = new Student[100];
```

第二，数据读入时，需要将读入的字符串转换为 Student 对象：

```
Student student = new Student();
student.parseStudent(line);
```

第三，数据输出时，需要将 Student 对象转换为字符串：

```
line = students[j].toString();
```

做完上面的三处修改后，抽奖器的完整代码如下：

```

import java.io.*;
import java.util.*;
class Selector {
    public static void main(String[] args) throws IOException {
        Student[] students = new Student[100];
        int count = 0;
        //读入学生数据，保存到 Student 对象数组中
        FileReader fin = new FileReader("Students.txt");
        BufferedReader in = new BufferedReader(fin);
        String line = in.readLine();
        while(line != null) {
            line = line.trim(); //去除字符串中前后空格
            if(line.length() > 0){ //检查是否是空行
                Student student = new Student();
                student.parseStudent(line);
                students[count] = student;
                count++;
                if(count>=100)break;
            }
            line = in.readLine();
        }
        in.close();
        fin.close();
    }
}
//确定输出文件名
int no = 0;

```

```

File outf;
do {
    no++;
    outf = new File("No"+no+".txt");
}while(outf.exists());
FileWriter  fout = new FileWriter(outf);
BufferedWriter  out = new BufferedWriter(fout);

//随机选出所需人数
int selectedCount = 10;
if( args.length > 0 ) selectedCount = Integer.parseInt(args[0]);
Random rand = new Random();
for(int i = 0; i < selectedCount; i++) {
    int j = rand.nextInt(count);
    line = students[j].toString();
    System.out.println(line);
    out.write(line,0,line.length());
    out.newLine();
    students[j] = students[count-1];
    count--;
}
out.close();
fout.close();
}

class Student {
    private String id;
    private String name;
    private String department;
    public void parseStudent(String str) {
        int tokenCount;
        StringTokenizer  t = new StringTokenizer(str);
        tokenCount = t.countTokens();
        id = t.nextToken();           //学号
        name = t.nextToken();         //姓名
        department = t.nextToken();   //学院
    }
    public String toString() {
        String s = id + "  " + name;
        for(int l = s.length();l<21; l++) s += "  "; //对齐
        return  s + department;
    }
}

```

第十一章 异常处理

11.1 为什么需要异常处理

异常处理机制提供处理程序运行时遇到的异常情况的方法。异常处理机制是 Java 中非常重要的一个机制，因为对遇到的异常情况进行处理对于提高程序的健壮性是非常必要的。

程序在运行过程中完全避免异常情况出现是不可能的，主要有两方面的原因：第一，程序运行时要和外部环境交互，而程序对于外部环境是无法完全控制的，所以外部环境有可能出现异常情况，这就导致程序运行过程中可能遇到异常情况。比如需要从硬盘上读取文件时文件不存在，或者通过网络传输数据时网络出现问题，等等。第二，人不可能不犯错误，所以人写的代码不可能完全没有 bug，比如数组下标越界、使用空引用等等。

既然这些异常情况无法完全避免，那么关键就是如何去处理这些异常情况。比较好的方式是按照下面的层次去处理异常：第一层次，通知用户发生了异常，发生了什么样的异常，也就是发生异常的原因。这样，用户才能根据这些信息去进行相应的处理，使得程序能够再正常运行。第二层次，允许用户保存当前工作，不至于因为异常而导致前面的工作白做了。用户处理完异常后，重新启动程序，可以继续异常出现前的工作。第三层次，应该能顺利终止程序，如果出现程序无法终止甚至造成整个系统死机，则会影响其他程序的运行，可能给用户带来更大损失。所以，程序运行时若碰到异常情况，至少应通知用户异常原因，如果能进一步让用户保存当前工作，并能正常退出程序，则程序的健壮性就比较高了。

一个好的程序应该把所有可能出错的情况都处理到。在编写程序之前就应把程序运行时可能遇到的异常情况都考虑到，并在程序中添加相应的代码进行处理。与异常不可避免的原因相对应，程序运行时可能遇到的异常情况分为两大类：环境异常和代码错误。

环境异常包括：

- 输入输出错误：如用户输入格式错误、文件读写错误。
- 设备错误：如打印机卡纸。
- 物理限制：硬盘空间用尽，内存耗尽。

代码错误包括：

- 无效的数组下标。
- 使用空引用访问对象。
- 被 0 除。

环境异常是无法避免的，只能在程序中做相应处理。代码错误可以通过修改源代码消除，但一般难以查找。因为这些代码错误不像语法错误在编译时直接报告出来，而是需要根据程序运行表现去分析源代码，属于程序逻辑上的错误。

11.2 异常处理流程

异常处理机制处理异常情况的流程是这样的：如果没有异常情况出现，程序就完全按照正常流程运行；一旦遇到异常情况，就中断正常流程，转到异常处理流程。因此，在异常处理中，把会导致中断程序正常流程的事件称为异常。

比如，从文件中读取数据的正常流程是：打开文件、获得文件大小、分配内存、将文件内容读取到内存、关闭文件。代码如下：

```
openTheFile;  
determine its size;  
allocate memory;  
read the file into memory;  
closeTheFile;
```

这个过程中的每一步都有可能遇到异常。比如打开文件时文件不存在，由于分区表损坏等原因文件大小无法获得，内存不足，硬盘上内容无法读取出来等。

异常处理的基本方式是用一条异常处理语句把正常的流程包起来，并在后面针对每一种可能的异常写一段异常处理代码。比如，对上面的例子的异常处理方式如下：

```
try {  
    openTheFile;  
    determine its size;  
    allocate memory;  
    read the file into memory;  
    closeTheFile;  
}  
catch(fileopenFailed) { dosomething; }  
catch(sizeDetermineFailed) {dosomething;}  
catch(memoryAllocateFailed){ dosomething;}  
catch(readFailed){ dosomething;}  
catch(fileCloseFailed) { dosomething; }
```

在异常处理机制之前，程序中的异常是用 `if` 语句通过判断函数返回值的方式来处理的。如果采用这种传统的方式来处理上面例子中的异常，代码如下：

```
openTheFile;  
if (theFilesOpen) {  
    determine its size;  
    if (gotTheFileLength){  
        allocate memory;  
        if (gotEnoughMemory) {  
            read the file into memory;  
            if (readFailed) errorCode=-1;  
        }else errorCode=-2;  
    }else errorCode=-3;
```

```
closeTheFile;  
if (closeFailed) errorCode=-4;  
}else errorCode=-5;
```

从这个例子可以看出，异常处理方式和传统的返回值方式相比，具有以下优点：

- 传统方法将大部分精力花在出错处理上了；而异常方式则可以先写出正常流程，再添加错误处理程序。
- 传统方法将错误处理和正常处理流程混在一起，造成程序可读性差；而异常方式则是互相独立的，程序可读性好。
- 传统方法都是使用分支语句来处理错误；而异常方式则按错误类型和错误差别分组。
- 传统方法出错返回信息量太少，通常只有一个数字作为错误代号；而异常方式返回的是一个对象，可以传递各种信息。
- 传统方法只把能够想到的错误考虑到，对以外的情况无法处理；而异常方式则能够部分地处理这些情况，如异常方式中可以定义不管发生何种情况一定会执行的代码。

11.3 异常处理语句

异常在 Java 中实际上是一个对象，称为异常对象。Java 中和异常处理有关的语句有三条：

- **try-catch-finally** 语句：捕获异常，是异常的终点。
- **throws** 语句：向上传递异常，用于函数声明的后面，向上指函数的调用者。
- **throw** 语句：抛出异常，是异常的源头。

11.3.1 try-catch-finally 语句

try 语句用于捕获一段代码中可能产生的异常，格式为：

```
try {  
    需要监视的语句块  
} catch (异常类 异常标识符){  
    异常处理代码  
} .....  
catch (异常类 异常标识符){  
    异常处理代码  
} finally {  
    最后一定会执行的代码  
}
```

try 后面的语句块中是正常流程的代码；后面可以有多个 **catch** 子句，每个 **catch** 子句中是对一种异常的处理代码；最后有一个可选的 **finally** 子句，其中的代码不管出不出现异常一定会执行。

执行过程中，如果 **try** 块中没有任何代码产生一个异常，那么程序就会跳过所有 **catch** 子句。只要 **try** 块中的任意代码产生一个异常，就会跳过 **try** 块中的剩余代码，执行相应 **catch** 子句。

中的异常处理代码。

具体对应执行那个 `catch` 子句，是用异常对象和 `catch` 子句中声明的异常类进行匹配来判断的。`catch` 子句后面的括号内的部分类似于一个参数表，异常类是参数类型，异常标识符是形参。当 `try` 块中产生异常时，Java 会将异常信息保存到一个异常对象中，并按顺序依次用异常对象的类型和 `catch` 子句的异常类匹配，如果异常对象属于该异常类或其子类，那么匹配成功，后面的 `catch` 子句就不再匹配了。匹配成功后，将异常对象赋给异常标识符，执行该 `catch` 子句中的异常处理代码，可以通过异常标识符访问异常对象。可以说，异常种类是靠异常对象的类型来区分的。如果所有 `catch` 子句都和异常对象匹配不上，说明这条 `try-catch` 语句无法捕获这种异常，此时应把整个这条 `try-catch` 语句看作一条语句，而这条语句产生了一个异常，需要更外层的语句去处理。

一个使用 `try` 语句捕获异常的例子如下：

```
import java.io.*;
public class Test1 {
    public static void main(String[] args) {
        try {
            FileReader fin = new FileReader("Students.txt");
            BufferedReader in = new BufferedReader(fin);
            String line = in.readLine();
            System.out.println(line);
            in.close();
            fin.close();
        } catch(FileNotFoundException e) {
            e.printStackTrace();
        } catch(IOException e) {
            System.out.println(e);
        }
    }
}
```

此例中，`FileReader` 的构造函数在文件 `Students.txt` 不存在时产生 `FileNotFoundException` 类型的异常，而 `in.readLine()` 语句和 `in.close()` 语句可能产生 `IOException` 类型的异常，对这两类异常在后面的两个 `catch` 子句中分别进行了处理。处理的方式是将异常信息显示给用户，这里又分为两种方法。一种是调用异常对象的 `printStackTrace()` 方法，既显示异常类型和原因，又显示产生异常的语句位置。另一种是直接用 `System.out.println(e)` 输出异常对象，相当于调用了异常对象的 `toString()` 方法，只显示异常类型和原因，不显示产生异常的语句位置。如下图。

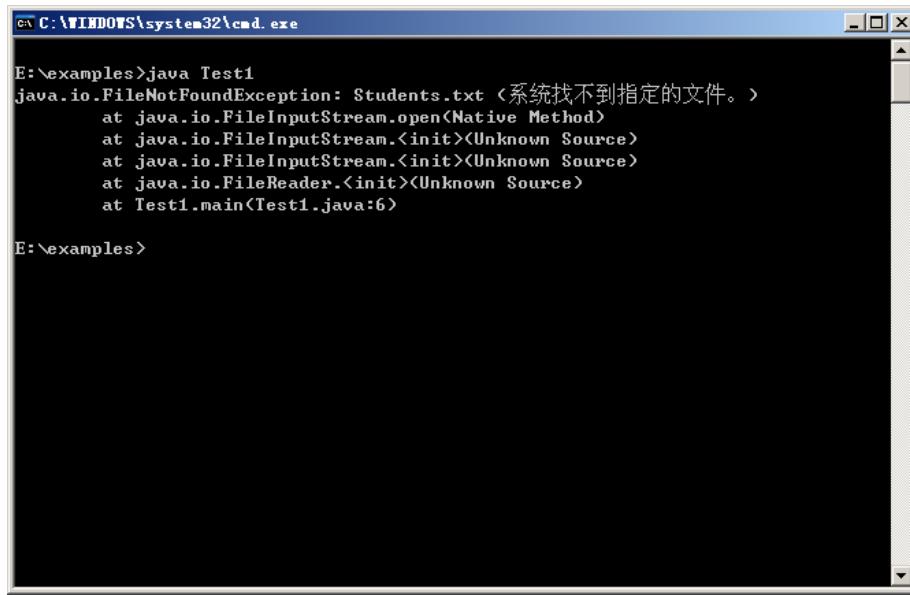


图 11-1 异常信息

那么，如何知道一段代码中会产生哪些种类的异常呢？Java 强制对所有异常进行处理，所以对于代码中可能产生但没有捕获的异常，编译时会报错，指出哪些方法可能产生哪些种类的异常。然后再查帮助文档，在每个方法说明的最下方会有一个部分叫“抛出”，会列出该方法可能产生的异常，并作解释。由此，就可以知道需要处理哪些异常了。例如：



图 11-2 帮助文件中的异常说明

try 语句可以带有 finally 子句，表示一定执行的代码块，不管发生不发生异常，也不管发生何种类型的异常。例如，文件流打开后，不管在读写过程中是否出现异常，最后都应该关闭文件流。为此，前面的例子应该改为：

```
import java.io.*;
public class Test7 {
    public static void main(String[] args){
        FileReader fin = null;
```

```

BufferedReader in = null;
try {
    fin = new FileReader("Students.txt");
    in = new BufferedReader(fin);
    String line = in.readLine();
    System.out.println(line);
}
catch(IOException e) {
    System.out.println(e);
}
finally {
    try {
        if(in != null) in.close();
        if(fin != null) fin.close();
    }
    catch(IOException e) {
        System.out.println("关闭文件异常");
    }
}
}

```

将关闭文件流的代码放到 `finally` 子句中，不管出不出现异常，都会被执行。为防止对没有创建的文件流执行关闭操作，用 `if` 语句判断文件流引用是否为空，如果文件流成功创建，则该引用不为空。同时，由于 `close()` 方法可能会抛出异常，所以在 `finally` 子句中应该再用一条 `try` 语句捕获 `close()` 语句的异常。这也说明 `try` 语句是可以嵌套使用的，在 `try` 语句块、`catch` 子句和 `finally` 子句中都可以再嵌套 `try` 语句。

11.3.2 throws 语句

Java 规定，对于一个函数，函数内的语句可能产生的所有异常，必须采用捕获或者向上传递的方式处理。对于在函数内知道如何处理的异常，应采用 `try-catch` 语句捕获，在函数内处理。对于在函数内不知道如何处理的异常，应将异常向上传递，即把异常交给函数的调用者去处理。

向上传递异常采用 `throws` 关键字，格式为：

```

函数声明 throws 异常类, 异常类
{ 函数体 }

```

在方法外部(从方法调用者的角度)看来，就相当于调用此方法的语句会产生这些类型的异常。例如：

```

import java.io.*;
public class Test2 {
    public static void main(String[] args) {

```

```

try {
    read();
} catch(IOException e) {
    System.out.println(e);
}

static void read() throws IOException {
    FileReader fin = new FileReader("Students.txt");
    BufferedReader in = new BufferedReader(fin);
    String line = in.readLine();
    System.out.println(line);
    in.close();
    fin.close();
}
}

```

此例的 `read` 函数中，不知道如何处理 `IOException` 异常，就用在函数声明后面用 `throws` 将其传递给调用者。在 `main` 函数中调用 `read` 函数时，需要对 `IOException` 异常进行处理。

一个方法可以将异常向上传递给调用者，调用者可以再传递给更上级的调用者，这样就形成一个逐级上传的层次。最后上传到 `main` 函数，`main` 函数声明后面如果有 `throws` 声明就可以再传递给 Java 虚拟机。虚拟机可以最终处理所有异常。异常在传递过程中，一旦遇到匹配的 `catch` 子句，就被捕获处理并停止传递，所以异常可以在适当的层次被处理。异常只有捕获或者传递两种处理方法，二者必居其一，所以保证了所有的异常都被处理。如图所示：

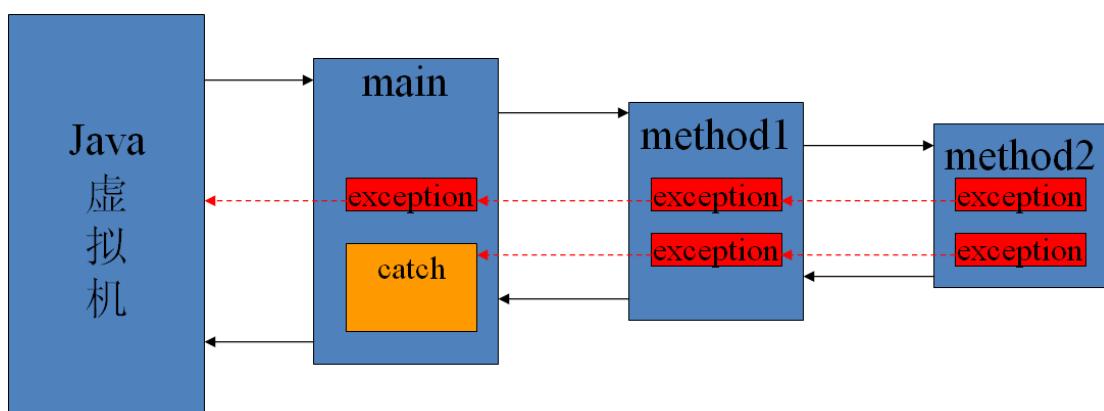


图 11-3 异常传递机制

11.3.3 throw 语句

当方法执行中检测到非正常情况，而在方法内部又处理不了时，就需要抛出异常。抛出异常实际上等于先产生出一个异常，然后将异常传递给调用者。

抛出异常使用 `throw` 语句，分为三步：找到一个合适的异常类，创建该类的一个对象，抛出

该对象(转去执行异常处理程序)。

例如，编写一个从文件中读取一定个数的字符的函数，如果要读取的个数大于文件中的字符个数，那么就可以认为是一种异常情况，用 `throw` 语句抛出异常。

```
import java.io.*;
public class Test3 {
    public static void main(String[] args) {
        try {
            read("Test3.txt",20);
        } catch(EOFException e) {
            System.out.println(e);
        } catch(IOException e) {
            System.out.println(e);
        }
    }

    static String read(String aFileName, int aCount) throws EOFException,IOException {
        FileInputStream fin = new FileInputStream(aFileName);
        BufferedInputStream in = new BufferedInputStream(fin);
        if(aCount>in.available()) {
            throw new EOFException("文件中没有这么多字符。");
        }
        String line = "";
        for(int i=0; i<aCount; i++) {
            line += in.read();
        }
        in.close();
        fin.close();
        return line;
    }
}
```

此例中，检测异常情况的方法是用要读取的字符数和文件流中的字符数比较。发现异常后，选择 `EOFException` 类作为异常种类，该类是 Java 类库中的一个异常类，描述读取数据时遇到了文件结尾。然后，用 `new` 创建该类的一个异常对象，构造函数的参数是一个描述异常情况的字符串。最后用 `throw` 语句抛出该对象。此外，在 `read()` 方法中可能抛出 `EOFException`，所以应该在方法声明后的 `throws` 部分，应加入 `EOFException`。

在 `catch` 子句中捕获的异常，也可以用 `throw` 语句再次抛出，这种方式使得在方法中能够捕获异常进行处理，然后再将异常向上传递给调用者去处理，一个异常可以在多个地方处理。例如：

```
import java.io.*;
public class Test4 {
    public static void main(String[] args) {
        try {
            read();
        }
```

```

        } catch(IOException e) {
            System.out.println("main:" + e);
        }
    }

    static void read() throws IOException {
        try {
            FileReader fin = new FileReader("Students.txt");
            BufferedReader in = new BufferedReader(fin);
            String line = in.readLine();
            System.out.println(line);
            in.close();
            fin.close();
        } catch(IOException e) {
            System.out.println("read:" + e);
            throw e;
        }
    }
}

```

此例中，在 `read` 方法中，捕获了 `IOException` 异常，输出一个以“`read:`”开头的异常信息后，再将异常向上传递。在 `main` 方法中，再次捕获该 `IOException` 异常，输出一个以“`main:`”开头的异常信息。

虽然异常处理相关的语句只有三条，但是这三条语句可以组合、嵌套使用，构成非常灵活的异常处理方式。

11.4 异常类

在异常机制中，每种异常的类型代表了一种异常的情况。`Java` 在类库中为每一种异常情况定义了一个异常类。这些类可以分为以下几大类，如图：

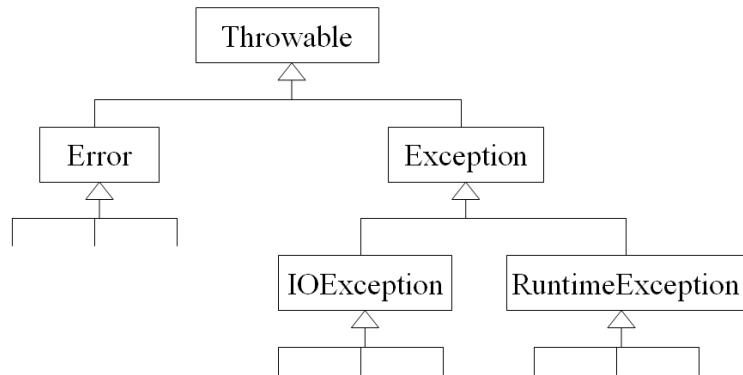


图 11-4 `Java` 中的异常类

最顶层的类叫 `Throwable`，所有异常类都是它的后代。`Throwable` 有两个子类：`Error` 和

`Exception`。`Exception` 又分为两个分支：`IOException` 和 `RuntimeException`。

`Error` 用于描述运行时系统的内部错误和资源耗尽错误。这种错误发生时，除了通知用户并终止程序外没有别的办法，所以不应该抛出这种类型的错误。

产生 `RuntimeException` 的原因是编程错误，如：错误的类型转换、越界数组访问、空引用访问、被 0 除等。这些错误应该通过修改程序进行改正，如果不处理，编译时也不会报错误，运行时会报异常。

产生 `IOException` 的原因是正确的程序遇到了意外的外部环境，如：打开不存在的文件、打开错误的 URL、读写异常等。这些异常必须用异常处理机制进行处理，如果不处理，编译时会报“未处理异常”错误。

`Exception` 分支是编写程序时处理的重点。对于 `RuntimeException` 应修改程序进行改正，对于 `IOException` 应采用异常处理机制进行处理。

当类库中的异常类不足以描述出现的异常情况时，可以自己创建异常类。自定义异常一样需要捕获或传递。创建异常类只需从 `Exception` 或其子类派生一个子类即可。该类一般具有一个无参数构造函数和一个带字符串参数的构造函数。

例如，抽奖器的例子中，如果要选出的对象个数大于对象总个数，这是一个异常情况，但类库中没有合适的异常类来描述这种情况。所以，自定义一个异常类 `SelectingException`。该类是 `IOException` 类的子类，在其带字符串参数的构造函数中，调用父类构造函数，将字符串设置为异常提示信息。该异常类的使用和类库中的异常类完全一样。

```
import java.io.*;
public class Test5 {
    private String[] students = new String[100];
    private int totalCount = 0;
    public static void main(String[] args){
        Test5 selector = new Test5();
        try {
            selector.select(200);
        }
        catch(SelectingException e) {
            System.out.println(e);
        }
    }
    public void select(int aCount) throws SelectingException {
        if(aCount>totalCount) {
            throw new SelectingException("无法从 "+totalCount+" 名同学中选出 "+aCount+" 名。");
        }
    }
}
```

```
class SelectingException extends IOException {  
    public SelectingException(){};  
    public SelectingException(String msg) {  
        super(msg);  
    }  
}
```

11.5 异常处理原则

异常处理时有一些原则，主要有下面三条：

- 异常不能代替简单测试
- 不要进行小粒度的异常处理
- 不要压制异常

因为异常处理过程比条件判断语句的简单测试更耗时间，所以能用 if 语句直接处理就尽量不要用异常处理机制。例如，Test7 中的 finally 子句中，用 if 语句判断文件流是否已创建，代码如下：

```
if(in != null) in.close();  
if(fin != null) fin.close();
```

不应写成下面的异常处理形式：

```
try {  
    in.close();  
    fin.close();  
}  
catch(NullPointerException e){  
    .....  
}
```

小粒度的异常处理会使代码变得冗长罗嗦，也没有将正常流程与异常处理分开，所以不应进行小粒度的异常处理。例如，Test1 中的异常处理，不应写成下面的形式：

```
try {  
    FileReader fin = new FileReader("Students.txt");  
} catch(FileNotFoundException e) { System.out.println(e); }  
try {  
    BufferedReader in = new BufferedReader(fin);  
} catch(IOException e) { System.out.println(e); }  
try {  
    String line = in.readLine();  
} catch(IOException e) { System.out.println(e); }  
System.out.println(line);  
try {  
    in.close();  
    fin.close();  
}
```

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

try 块中的语句最好形成一个完整的功能流程，不能对每条语句单独进行异常处理。

压制异常指捕获了异常又不做处理。压制异常会使得程序对异常情况不做任何处理，造成运行时即使遇到异常情况用户也无法发现。压制异常的例子如下：

```
try {
    FileReader fin = new FileReader("Students.txt");
    BufferedReader in = new BufferedReader(fin);
    String line = in.readLine();
    System.out.println(line);
    in.close();
    fin.close();
} catch(FileNotFoundException e) {          // 捕获异常又不做处理
} catch(IOException e) {
}
```

异常处理中，至少应该将异常信息通知用户。

11.6 异常处理示例

本节将为 10.6 节中的抽奖器添加异常处理代码。

Selector 程序中需要处理的异常包括：

- **NumberFormatException:** 输入参数 arg[0] 不是整数格式。
- **ArrayIndexOutOfBoundsException:** Student.txt 文件中学生数大于 100。
- **SelectingException:** 要选出的学生数大于学生名单中的学生数。
- **FileNotFoundException:** 文件 Student.txt 不存在。
- **IOException:** 读写文件时发生错误。

处理异常的方法是告知用户发生错误的原因，并终止程序的运行。为了便于进行异常处理，对于程序结构上也做了改动，将各功能模块分割成为多个方法。最终的程序代码如下：

```
import java.io.*;
import java.util.*;

class Selector {
    private Student[] students = new Student[100];
    private int totalCount = 0;
    private Student[] selected = new Student[20];
    private int selectedCount = 0;

    public static void main(String[] args) {
        Selector selector = new Selector();
        int selectedCount = 10;
```

```

try { //正常处理流程
    if( args.length > 0 ) selectedCount = Integer.parseInt(args[0]);
    selector.loadStudents("Students.txt");
    selector.randomSelect(selectedCount);
    selector.printSelected();
    String name = selector.getFileName();
    selector.saveSelected(name);
}
catch(NumberFormatException e) {
    System.out.println(args[0]+"不是一个合法的整数。");
}
catch(ArrayIndexOutOfBoundsException e) {
    System.out.println(e);
}
catch>SelectingException e) {
    System.out.println(e);
}
catch(FileNotFoundException e) {
    System.out.println("FileNotFoundException: " + e.getMessage());
}
catch(IOException e) {
    System.out.println(e);
}
}

/** 读入学生数据 */
public void loadStudents(String aFileName) throws IOException {
    FileReader fin = new FileReader(aFileName);
    BufferedReader in = new BufferedReader(fin);
    String line = in.readLine();
    while(line != null) {
        line = line.trim(); //去除字符串中前后空格
        if( line.length() > 0 ){ //检查是否是空行
            Student student = new Student();
            student.parseStudent(line);
            students[totalCount] = student;
            totalCount++;
            if(totalCount>=100) {
                throw new ArrayIndexOutOfBoundsException("无法容纳多于 100
个学生。");
            }
        }
        line = in.readLine();
    }
}

```

```

        in.close();
        fin.close();
    }

    /** 将选出的学生名单保存到文件 */
    public void saveSelected(String aFileName) throws IOException {
        FileWriter  fout = new FileWriter(aFileName);
        BufferedWriter  out = new BufferedWriter(fout);
        for(int i = 0; i < selectedCount; i++){
            String line = selected[i].toString();
            out.write(line,0,line.length());
            out.newLine();
        }
        out.close();
        fout.close();
    }

    /** 打印选出的学生名单 */
    public void printSelected() {
        for(int i = 0; i < selectedCount; i++){
            System.out.println(selected[i]);
        }
    }

    /** 随机选出 aCount 名学生 */
    public void randomSelect(int aCount) throws SelectingException{
        Random rand = new Random();
        if(aCount > 20)      {
            throw new ArrayIndexOutOfBoundsException("因容量关系无法选出多于 20
个学生。");
        }
        if(aCount > totalCount) {
            throw new SelectingException("无法从 "+totalCount+" 个学生中选出
"+aCount+" 个。");
        }
        selectedCount = aCount;
        for(int i = 0; i < selectedCount; i++){
            int j = rand.nextInt(totalCount);
            selected[i] = students[j];
            students[j] = students[totalCount-1];
            totalCount--;
        }
    }
}

```

```

/** 确定输出文件名 */
private String getFileName() {
    int no = 0;
    File outf;
    do {
        no++;
        outf = new File("No"+no+".txt");
    }while(outf.exists());
    return(outf.getName());
}

class SelectingException extends Exception {
    public SelectingException(){}
    public SelectingException(String msg) {
        super(msg);
    }
}

class Student {
    private String id;
    private String name;
    private String department;
    public void parseStudent(String str) {
        int tokenCount;
        StringTokenizer t = new StringTokenizer(str);
        tokenCount = t.countTokens();
        id = t.nextToken();           //学号
        name = t.nextToken();         //姓名
        department = t.nextToken();   //学院
    }
    public String toString() {
        String s = id + " " + name;
        for(int l = s.length();l<21;l++) s += " "; //对齐
        return s + department;
    }
}

```

第十二章 集合框架

12.1 集合框架简介

因为计算机适合处理批量数据，所以“数据集合”可以说是程序设计中最常用的数据结构。此处的“集合”不同于数学中集合的概念，是允许出现重复元素的。为了支持这种数据结构，Java 提供了集合框架（Collection Framework）。集合框架是为表示和操作数据集合而规定的一种统一的标准的体系结构，包含三大块内容：对外的接口、接口的实现和数据集合上的算法。

Java 的集合框架利用了 Java 的单根继承性。数据集合中的数据统一用 `Object` 类型的引用表示，由于 `Object` 是所有类的祖先，根据多态性 `Object` 类型的引用可以指向任何类型的 Java 对象，这样数据集合中就可以保存任何类型的 Java 对象。但是由于八个基本数据类型不是对象，所以数据集合中不能直接保存基本数据类型的数据，可以将基本数据类型的数据转换为对应的包装类的对象，就可以保存到数据集合中了。

在数据结构中通常采用逻辑结构和物理结构分开的思想。逻辑结构是从外部看到的模型的功能，即模型可以如何使用。物理结构是模型内部的实现机制。一种逻辑结构可以有多种物理结构来实现。Java 集合框架也采用这种思想，用接口来描述模型的逻辑结构，用类来实现物理结构，一个接口可以由不同的类来实现。

比如，队列的逻辑结构如图所示，其主要功能为入队、出队和求队长。

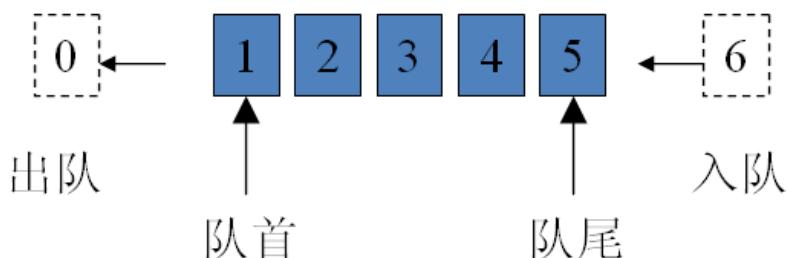


图 12-1 队列的逻辑结构

在 Java 中，用接口定义队列的逻辑结构如下

```
Interface Queue {  
    void add(Object obj);      // 入队  
    Object remove();          // 出队  
    int size();                // 求队长  
}
```

队列的逻辑结构可以用链表来实现，如图：

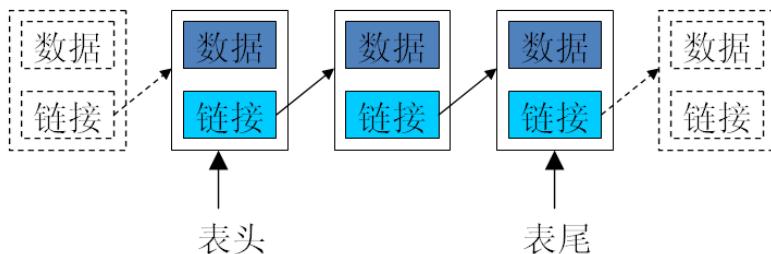


图 12-2 链表

在 Java 中可以将链表实现用 `LinkedList` 类来描述如下：

```
class LinkedList implements Queue {
    public void add (Object obj){...};
    public Object remove (){...};
    public int size (){...};
    private LinkedList header; // 表头
    private LinkedList tail; // 表尾
    private class Entry { // 表中元素
        Object element; // 数据
        Entry next; // 指针
    }
}
```

除了用链表外，队列的逻辑结构还可以用循环数组来实现，如图。可以定义另外一个类来用循环数组实现队列的功能。

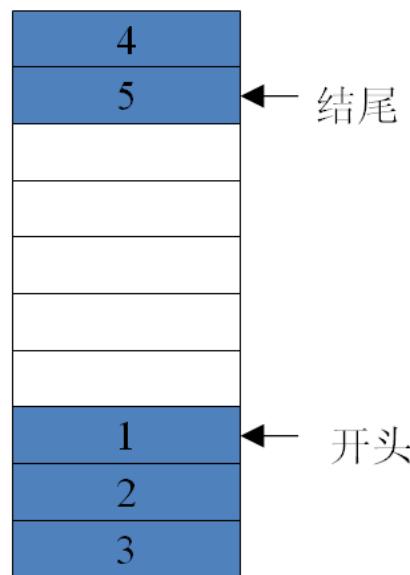


图 12-3 循环数组

队列的例子说明，一种逻辑功能可以有多种实现方式。这一点在 Java 中体现为一个接口可以被多个类实现。逻辑功能相同的情况下，不同的物理实现有不同的效率。例如，线性表功能可以用链表或顺序表来实现，链表对于插入和移除效率较高，对于随机访问效率较低，而

顺序表对于插入和移除效率较低，对于随机访问效率较高。采用这种功能定义和物理实现分开的方式，可以提高程序的灵活性，很容易从一种实现改为另一种实现。对程序员来说，了解接口即可完成所需功能，了解具体实现则可以提高程序的效率。

12.2 集合框架中的接口和类

集合框架中的接口和类都定义在 `java.util` 包中。接口主要有：

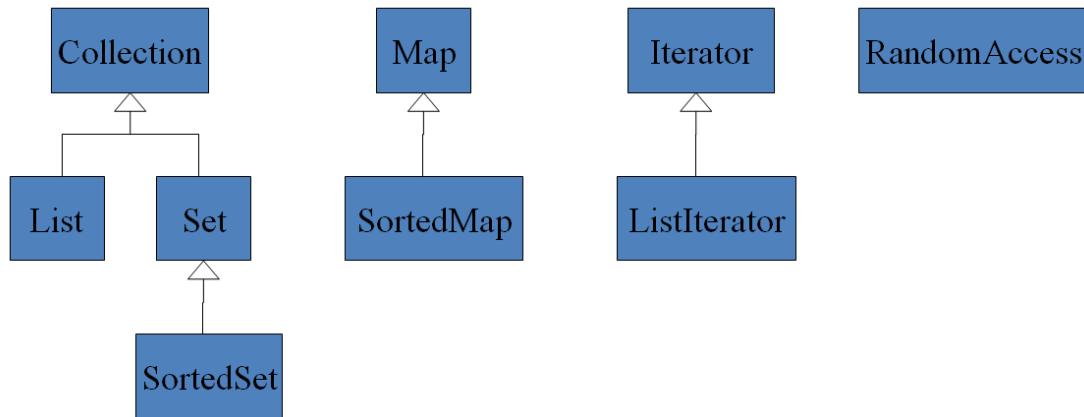


图 12-4 集合框架中的接口

`Collection` 接口定义的是“数据集合”的公共功能。

`List` 接口定义的是线性表的逻辑结构。

`Set` 接口定义的是数学上的集合。`SortedSet` 接口定义的是有序集合。

`Map` 接口定义的是二元组集合，`SortedMap` 定义的是有序二元组集合。

`Iterator` 和 `ListIterator` 是两个枚举器接口，用于枚举或遍历数据集合中的所有元素。

`RandomAccess` 接口是一个标志，不定义任何功能，实现此接口的类随机访问效率较高。

集合框架是从 Java2 开始才有的，但之前已有一些“旧类”，这些类现在也被纳入集合框架中。这些“旧类”包括：

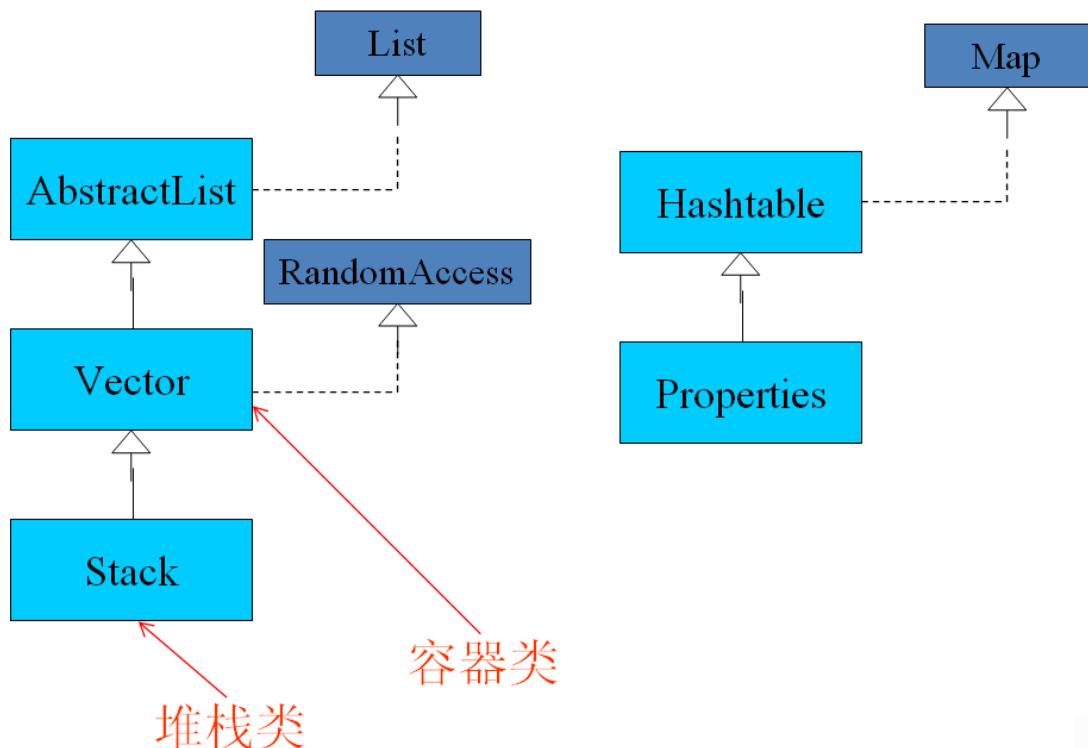


图 12-5 集合框架中的旧类

Java2 新加入集合框架的类包括:

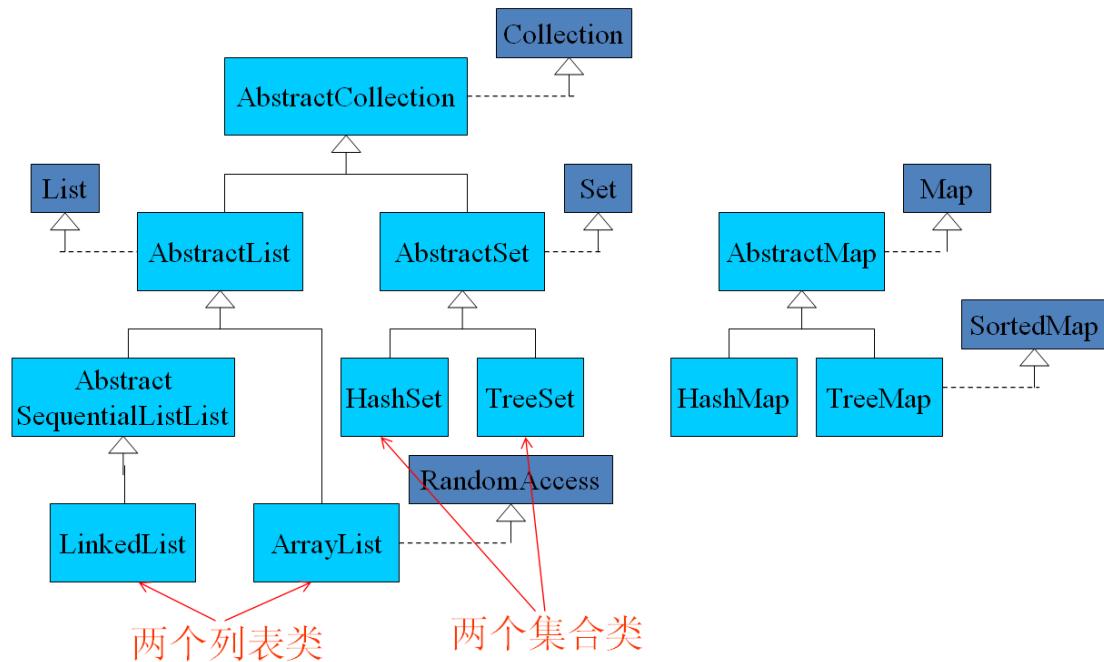


图 12-6 集合框架中的新类

这些类中，最常用的是三个实现 List 接口的类：Vector 和 ArrayList 用数组实现 List 接口，LinkedList 用双向循环链表实现 List 接口。

两个实现 Set 接口的类是 HashSet 和 TreeSet，分别用散列表和平衡二叉树来实现集合的功能，主要是完成查找重复元素的操作效率较高。

12.3 List—列表

List 接口定义的是线性表的逻辑结构。List 中可以有重复元素，元素有先后顺序，又称为列表。其逻辑结构如图



图 12-7 列表的逻辑结构

List 接口中定义的方法主要有：

```
public interface List extends Collection {  
    boolean add(Object o); // 将元素 o 添加到列表最后。  
    void clear(); // 清空列表内容。  
    boolean contains(Object o); // 判断列表中是否包含元素 o。  
    Object get(int index); // 获取列表中 index 位置的元素。  
    int indexOf(Object o); // 获取元素 o 在列表中的位置。  
    boolean isEmpty(); // 列表是否为空。  
    int lastIndexOf(Object o); // 获取列表中最后一个 o 出现的位置。  
    ListIterator listIterator(); // 获取一个枚举器。  
    Object remove(int index); // 去除位置 index 的元素。  
    boolean remove(Object o); // 去除列表中第一个出现的 o 元素。  
    Object set(int index, Object element); // 用 element 取代位置 index 的元素。  
    int size(); // 返回列表中元素个数。  
    List subList(int fromIndex, int toIndex); // 得到一个子范围。  
    Object[] toArray(); // 将列表转化为一个数组。  
}
```

从 List 接口中定义的 get、indexOf、lastIndexOf、remove、set 这些方法来看，List 中的元素是有位置的概念的，可以由重复元素。此外，从所有方法来看，List 中的元素都是 Object 类型的引用，可以指向任何类型的对象。

实现 List 接口的类主要有三个：LinkedList、ArrayList 和 Vector。

LinkedList 类采用双向循环链表来实现线性表的功能。其关键部分的源代码如下：

```
public class LinkedList extends AbstractSequentialList
```

```

    implements List, Cloneable, java.io.Serializable{
private static class Entry {
    Object element;
    Entry next;
    Entry previous;
    Entry(Object element, Entry next, Entry previous) {
        this.element = element;
        this.next = next;
        this.previous = previous;
    }
}
private Entry addBefore(Object o, Entry e) {
    Entry newEntry = new Entry(o, e, e.previous);
    newEntry.previous.next = newEntry;
    newEntry.next.previous = newEntry;
    size++;
    modCount++;
    return newEntry;
}
}

```

链表的一个节点结构在 `Entry` 中定义，数据用 `Object` 类型的引用 `element` 保存，既有后向指针 `next` 又有前向指针 `previous`。另外，在插入节点的 `addBefore` 方法中，既需要调整前一个节点的后向指针，又需要调整后一个节点的前向指针。所以，可以看出 `LinkedList` 采用的物理结构是双向循环链表，如图

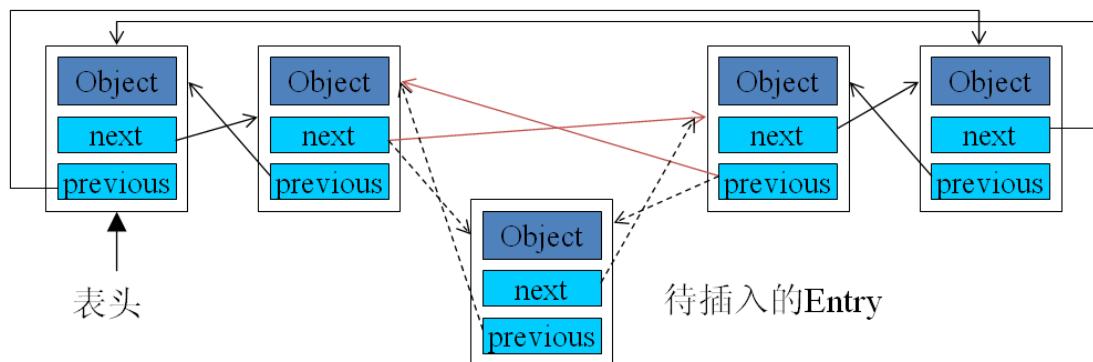


图 12-8 双向循环链表的插入操作

`ArrayList` 类采用数组来实现线性表的功能。其关键部分代码如下：

```

public class ArrayList extends AbstractList
    implements List, RandomAccess, Cloneable, java.io.Serializable {
private transient Object elementData[];
    public void ensureCapacity(int minCapacity) {
        modCount++;
        int oldCapacity = elementData.length;
        if (minCapacity > oldCapacity) {
            Object oldData[] = elementData;

```

```

        int newCapacity = (oldCapacity * 3)/2 + 1;
        if (newCapacity < minCapacity)
            newCapacity = minCapacity;
        elementData = new Object[newCapacity];
        System.arraycopy(oldData, 0, elementData, 0, size);
    }
}

```

用数组来实现线性表需要解决的关键问题是数组容量的限制。从上面的代码中可以看出，当数组容量被用光而又有元素需要插入时，会申请一个容量更大的数组，并把原数组中的所有元素复制到新数组，这样就又有容量存放新元素了。新数组的容量是原数组的 1.5 倍。如图

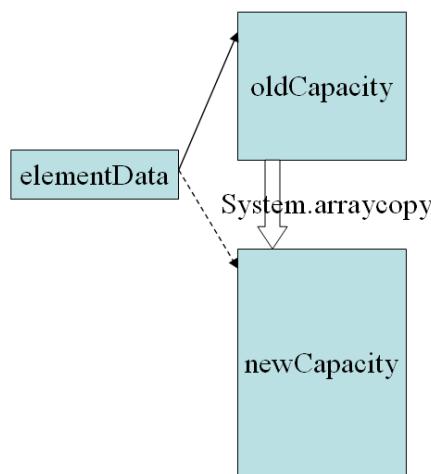


图 12-9 数组列表的扩容

`Vector` 类的实现机制与 `ArrayList` 相同，区别在于 `Vector` 支持多线程同步读写，`ArrayList` 不支持。为了支持线程同步，`Vector` 需要多做一些操作，所以其效率比 `ArrayList` 低。

下面是一个使用列表的例子：创建一个链式列表，保存整数对象 0-9，然后输出。代码如下：

```

//Test1_1.java
import java.util.*;
public class Test1_1 {
    public static void main(String[] args) {
        LinkedList l = new LinkedList(); // 创建一个 LinkedList 对象
        for(int i=0;i<10;i++) {
            l.add(new Integer(i)); // 将 int 变量包装成 Integer 对象，加入列表
        }
        String s = "";
        for(int i=0; i<l.size(); i++) {
            Object o = l.get(i); // 获得列表中的第 i 个元素
            s += o + " ";
        }
        System.out.println(s);
    }
}

```

```
}
```

这里遍历列表中元素用的是 `get` 方法依次获取各个位置上的元素。还可以先将列表转化为数组，在通过数组输出，代码如下：

```
Object[] pl = l.toArray();
for(int i=0;i<pl.length;i++) {
    Object o = pl[i];
    s += o + " ";
}
```

遍历 `List` 中的元素还可以用 `Iterator`（枚举器）接口。`Iterator` 接口中定义的方法有：

```
public interface Iterator {
    boolean hasNext();      是否已经枚举到结尾。
    Object next();         下一个元素。
    void remove();         从集合中删除刚枚举过的元素
}
```

从其方法定义可以看出，`Iterator` 是一个单向枚举的接口，在枚举的过程中指针只能从前向后移动。

使用枚举器遍历前面的例子中的列表的代码如下：

```
Iterator iter = l.iterator();
while(iter.hasNext()) {
    Object o = iter.next();
    s += o + " ";
}
```

代码中，先获得列表的枚举器，再用一个 `while` 循环来遍历列表中的元素，循环的结束条件是枚举器是否到了结尾，循环中用枚举器的 `next` 方法得到下一个元素。

`Iterator` 派生出一个子接口 `ListIterator`，其定义如下：

```
public interface ListIterator extends Iterator {
    void add(Object o);      在当前位置插入一个元素。
    boolean hasNext();       正向是否还有其他元素。
    boolean hasPrevious();   反向是否还有其他元素
    Object next();          下一个元素。
    int nextIndex();         返回下一个元素的位置。
    Object previous();      上一个元素
    int previousIndex();    返回上一个元素的位置。
    void remove();          删除刚刚访问过的元素。
    void set(Object o);     用 o 覆盖刚刚访问过的元素。
}
```

从其方法定义可以看出，`ListIterator` 是一个双向枚举的接口，在枚举的过程中指针既可以向后移动又可以向前移动。

对于前面的例子，可以很方便的改为用数组列表来实现，只需要将 `main` 函数中第一条语句改为：

```
ArrayList l = new ArrayList();
```

其它代码都不用改动。根据多态性，这条语句还可以写成：

```
List l = new ArrayList();
```

如果要用 `Vector` 来实现，也很简单：

```
List l = new Vector();
```

这些代码见 `Test1_2.java`

由于 `Object` 类型的引用可以指向任何类型的对象，列表中可以放入不同类型的对象。如下面的代码：

```
// Test1_3.java
import java.util.*;
public class Test1_3 {
    public static void main(String[] args) {
        List l = new LinkedList();
        l.add(new Integer(5));
        l.add(new Float(2.3f));
        l.add(new Boolean(true));
        l.add("string");
        String s = "";
        Iterator iterator = l.iterator();
        while(iterator.hasNext()) {
            Object o = iterator.next();
            s += o + " ";
        }
        System.out.println(s);
    }
}
```

此例中，一个列表中保存了 `Integer`、`Float`、`Boolean`、`String` 四种不同类型的对象，遍历时也能依次输出。

列表中存放不同类型的对象时都是用 `Object` 类型的引用保存的，这样在遍历时很可能需要通过造型恢复对象的类型，如果弄错类型就会产生造型异常。也有可能将一些对象误放入列表，带来不稳定的因素。为此，Java 为列表提供了类型检查机制，在声明列表引用、创建列表对象和声明枚举器引用时，可以声明列表中元素的类型。声明类型的方式是：`<类型>`。类型检查机制用于列表中元素都是同一类型的对象时。

为了程序健壮性的考虑，Java 编译器对于没有声明元素类型的列表会做出警告。所以，前面的例子编译时，都会有警告，如图：

```
C:\WINDOWS\system32\cmd.exe
E:\>cd examples
E:\examples>javac Test1_3.java
注: Test1_3.java 使用了未经检查或不安全的操作。
注: 有关详细信息, 请使用 -Xlint:unchecked 重新编译。
E:\examples>
```

图 12-10 类型检查警告

根据警告的提示，使用选项再编译，编译器就会指出具体哪些语句上可以声明列表中元素的类型，如图：

```
C:\WINDOWS\system32\cmd.exe
E:\examples>javac -Xlint:unchecked Test1_3.java
Test1_3.java:6: 警告: [unchecked] 对作为原始类型List的成员的add(E)的调用未经过检查
        l.add(new Integer(5));
               ^
其中, E是类型变量:
    E扩展已在接口 List中声明的Object
Test1_3.java:7: 警告: [unchecked] 对作为原始类型List的成员的add(E)的调用未经过检查
        l.add(new Float(2.3f));
               ^
其中, E是类型变量:
    E扩展已在接口 List中声明的Object
Test1_3.java:8: 警告: [unchecked] 对作为原始类型List的成员的add(E)的调用未经过检查
        l.add(new Boolean(true));
               ^
其中, E是类型变量:
    E扩展已在接口 List中声明的Object
Test1_3.java:9: 警告: [unchecked] 对作为原始类型List的成员的add(E)的调用未经过检查
        l.add("string");
               ^
其中, E是类型变量:
```

图 12-11 类型检查警告

为本节第一个例子添加类型检查后，代码如下：

```
// Test1_4
```

```

import java.util.*;
public class Test1_4 {
    public static void main(String[] args) {
        List<Integer> l = new LinkedList<Integer>();          // 添加类型检查
        for(int i=0;i<10;i++) {
            l.add(new Integer(i));
        }
        String s = "";
        Iterator<Integer> iterator = l.iterator();           // 添加类型检查
        while(iterator.hasNext()) {
            Integer o = iterator.next();                    // 直接得到 Integer 型
            s += o+" ";
        }
        System.out.println(s);
    }
}

```

代码中在声明列表引用、创建列表对象和声明枚举器引用时，在类型后面都添加了<Integer>，表示列表中的元素为 Integer 对象。同时，在遍历列表中元素时，调用枚举器的 next 方法直接得到 Integer 型引用，不再是 Object 型引用。

12.4 Set—集合

Set 接口描述一个数学概念上的集合。Set 中不能有重复元素，元素没有先后顺序之分。其逻辑结构如图：

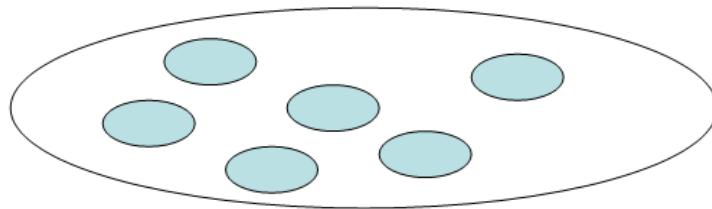


图 12-12 集合的逻辑结构

Set 接口中定义的主要方法有：

```

public interface Set extends Collection {
    boolean add(Object o); 将元素 o 加入集合。
    boolean addAll(Collection c); 将集合 c 中元素并入集合。
    void clear(); 清空集合。
    boolean contains(Object o); 判断集合中是否包含元素 o。
    boolean containsAll(Collection c); 判断集合是否全部包含 c 中元素。
    boolean equals(Object o); 判断两个集合是否相等。
    boolean isEmpty(); 判断集合是否为空。
    Iterator iterator(); 返回一个枚举器。
}

```

```
    boolean remove(Object o); 从集合中删除元素 o。  
    int size(); 返回集合元素个数。  
    Object[] toArray(); 将集合转化为一个数组。  
}
```

从 **Set** 接口中定义的方法可以看出，**Set** 中的元素是没有位置概念的，不能有重复元素。**Set** 中的元素都是 **Object** 类型。

SortedSet 接口描述一个自动排序的集合，除了 **Set** 中的功能外，还能够自动为集合中的元素排序。

实现 **Set** 接口的有两个类：**HashSet** 和 **TreeSet**。

HashSet 类用散列表实现 **Set** 接口。散列表查找重复元素的时间复杂度为 $O(C)$ ，而链表和数组中查找重复元素的时间复杂度为 $O(n)$ 。**HashSet** 的实现主要利用了 **HashMap**，**HashMap** 的关键部分代码如下：

```
public class HashMap extends AbstractMap  
    implements Map, Cloneable, Serializable {  
    Entry[] table;  
    static class Entry implements Map.Entry {  
        final Object key;          Object value;  
        final int hash;            Entry next;  
    }  
    public boolean containsKey(Object key) {  
        Object k = maskNull(key);  
        int hash = hash(k);  
        int i = indexFor(hash, table.length);  
        Entry e = table[i];  
        while (e != null) {  
            if (e.hash == hash && eq(k, e.key))  
                return true;  
            e = e.next;  
        }  
        return false;  
    }  
}
```

在向 **HashSet** 中添加一个元素时，首先要用 **containsKey** 方法查找集合中是否已有同样的元素存在。散列表的原理是将表中元素保存在多个列表上，没个列表对应一个整数。查找时，首先根据元素内容运用 **hash** 函数生成一个整数，再遍历该整数对应的列表中是否存在同样的元素即可。**hash** 函数对于相同元素生成的整数是相同的，所以如果集合中存在同样的元素的话一定放在同一个整数对应的列表上，这样就不用遍历其它列表了，大大缩小了搜索空间。如图

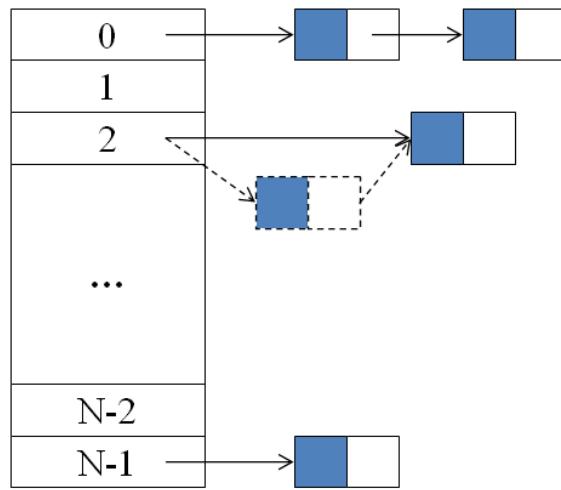


图 12-13 Hash 表

`TreeSet` 类用平衡二叉树实现 `SortedSet` 接口，其查找重复元素的操作的时间复杂度为 $O(\log_2 n)$ 。`TreeSet` 类的实现主要利用了 `TreeMap` 类，`TreeMap` 类的关键部分代码如下：

```
public class TreeMap extends AbstractMap
    implements SortedMap, Cloneable, java.io.Serializable {
    private transient Entry root = null;
    .....
    private Entry getEntry(Object key) {
        Entry p = root;
        while (p != null) {
            int cmp = compare(key, p.key);
            if (cmp == 0)
                return p;
            else if (cmp < 0)
                p = p.left;
            else
                p = p.right;
        }
        return null;
    }
}
```

平衡二叉树是一棵有序树，每个元素的左子树中元素都比本节点小，右子树中元素都比本节点大，而且左右子树长度之差不超过 1。因此查找时用要查找的元素与根节点比较大小，若想等则根节点即为要查找的节点，若小于根节点则查找左子树，否则查找右子树，这是一个递归的过程。上面的代码实现的就是此查找的过程。因为平衡二叉树的层数= $\lceil \log_2 n \rceil + 1$ ，所以查找操作的时间复杂度为 $O(\log n)$ 。如图：

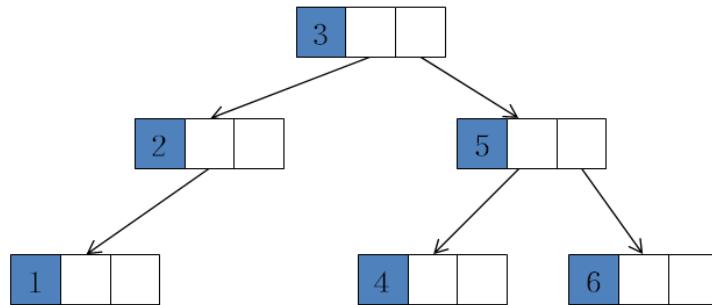


图 12-14 平衡二叉树

下面的例子创建一个 `HashSet`, 保存整数对象 0-9, 然后用枚举器输出。代码如下:

```

// Test2_1.java
import java.util.*;
public class Test2_1 {
    public static void main(String[] args) {
        HashSet set = new HashSet();
        for(int i=0;i<10;i++) {
            set.add(new Integer(i));
        }
        String s = "";
        Iterator iterator = set.iterator();
        while(iterator.hasNext()) {
            Object o = iterator.next();
            s += o+ " ";
        }
        System.out.println(s);
    }
}
  
```

如果要改为用 `TreeSet` 实现, 只需将 `main` 函数中第一条语句改为:

```
TreeSet set = new TreeSet();
```

根据多态性, 该语句还可以改为:

```
Set set = new HashSet();
```

即使改变元素的添加顺序, 比如把第一个 `for` 循环改为从 9 到 0, 遍历时输出的顺序也不会发生变化, 这一点与列表不同。另外, 即使将元素添加两次, 输出时也只输出一个, 比如可以把第一个 `for` 循环再复制一次。因为第二次添加时由于集合中已有同样的元素就不再执行了, 这一点也与列表不同。

`Set` 也可以添加类型检查, 方法与 `List` 类似。为上面的例子添加类型检查后, 代码如下:

```

// Test2_2.java
import java.util.*;
public class Test2_2 {
    public static void main(String[] args) {
        Set<Integer> set = new HashSet<Integer>();
  
```

```

        for(int i=0;i<10;i++) {
            set.add(new Integer(i));
        }
        String s = "";
        Iterator<Integer> iterator = set.iterator();
        while(iterator.hasNext()) {
            Integer o = iterator.next();
            s += o+" ";
        }
        System.out.println(s);
    }
}

```

12.5 用列表改进抽奖器

前面介绍的抽奖器的例子是用数组来保存学生信息，可以改为用列表来实现，这样就不再受数组容量的限制了。具体需要改动下面几处：

将学生信息保存在列表中：

```
List students = new ArrayList();
```

读入信息时用 add 方法将信息添加到列表中：

```
students.add(student);
```

学生数量用 size()方法获得：

```
int j = rand.nextInt(students.size());
```

选出学生用 get()方法，注意要造型：

```
Student selectedOne = (Student)(students.get(j));
```

将选出的学生从列表中删除用 remove()方法：

```
students.remove(j);
```

完整的程序代码如下。为了和原来代码比较，并没有将原来的代码删除，而是把它们改为了注释。

```

// Selector1.java
import java.io.*;
import java.util.*;

class Selector1 {
    public static void main(String[] args) throws IOException {
        //Student[] students = new Student[100];
        //int count = 0;
        List students = new ArrayList();
        //读入学生数据，保存到 Student 对象数组中
        FileReader fin = new FileReader("Students.txt");
        BufferedReader in = new BufferedReader(fin);

```

```

String line = in.readLine();
while(line != null) {
    line = line.trim(); //去除字符串中前后空格
    if( line.length() > 0 ){ //检查是否是空行
        Student student = new Student();
        student.parseStudent(line);
        //students[count] = student;
        //count++;
        //if(count>=100)break;
        students.add(student);
    }
    line = in.readLine();
}
in.close();
fin.close();

//确定输出文件名
int no = 0;
File outf;
do {
    no++;
    outf = new File("No"+no+".txt");
}while(outf.exists());
FileWriter fout = new FileWriter(outf);
BufferedWriter out = new BufferedWriter(fout);

//随机选出所需人数
int selectedCount = 10;
if( args.length > 0 ) selectedCount = Integer.parseInt(args[0]);
Random rand = new Random();
for(int i = 0; i < selectedCount; i++) {
    //int j = rand.nextInt(count);
    //line = students[j].toString();
    int j = rand.nextInt(students.size());
    Student selectedOne = (Student)students.get(j);
    line = selectedOne.toString();
    System.out.println(line);
    out.write(line,0,line.length());
    out.newLine();
    //students[j] = students[count-1];
    //count--;
    students.remove(j);
}
out.close();

```

```

        fout.close();
    }
}

class Student {
    private String id;
    private String name;
    private String department;
    public void parseStudent(String str) {
        int tokenCount;
        StringTokenizer t = new StringTokenizer(str);
        tokenCount = t.countTokens();
        id = t.nextToken();           //学号
        name = t.nextToken();         //姓名
        department = t.nextToken();   //学院
    }
    public String toString() {
        String s = id + " " + name;
        for(int l = s.length();l<21;l++) s += " "; //对齐
        return s + department;
    }
}

```

如果为此例中的列表添加类型检查，完整程序代码见 Selector2.java。

12.6 常用算法

Java 提供了一些针对数据集合的算法，这些算法在 `java.util.Collections` 类的静态方法中实现。`Collections` 类中定义的常用方法有：

```

public class Collections {
    static int binarySearch(List list, Object key)      二分查找
    static void fill(List list, Object obj)             填充
    static Object max(Collection coll)                查找最大元素
    static Object min(Collection coll)                查找最小元素
    static boolean replaceAll(List list, Object oldVal, Object newVal) 替换
    static void reverse(List list)                     反向
    static void rotate(List list, int distance)        循环移动
    static void shuffle(List list)                    随机打乱
    static void sort(List list)                      排序
    static void swap(List list, int i, int j)          交换
}

```

下面用一个例子来说明各个方法如何使用。将 0 到 9 封装为 Integer 类型的对象放到一个列表中，然后调用 shuffle 方法将顺序打乱。代码如下：

```
// Test4.java
import java.util.*;
public class Test4 {
    public static void main(String[] args) {
        List l = new LinkedList(); // 创建列表
        for(int i=0;i<10;i++) {
            l.add(new Integer(i));
        }
        Collections.shuffle(l); // 打乱列表内容
        printList(l);
    }
    public static void printList(List l) { // 将列表中内容输出到一行上
        String s = "";
        Iterator iterator = l.iterator();
        while(iterator.hasNext()) {
            s += iterator.next()+" ";
        }
        System.out.println(s);
    }
}
```

在此例子的基础上，可以继续测试其它方法。比如：

```
Collections.max(l); // 求列表中最大值
Collections.sort(l); // 排序
Collections.reverse(l); // 将列表中元素反向排列
Collections.rotate(l,1); // 所有元素向右移动 1 个位置
l.remove(new Integer(2)); // 移除元素 2
Collections.sort(l); // 注意运用二分查找法之前必须保证列表有序
Collections.binarySearch(l,new Integer(3)); //二分查找法查找元素 3 所在位置
```

其中， sort 方法用的是快速排序算法，在 Java SDK 安装目录下的 demo\applet\sortDemo 子目录下有个 applet，可以演示冒泡排序、双向冒泡排序和快速排序在效率上的差别。

此外还应注意， binarySearch 方法采用的是二分查找算法，所以调用该方法前，应确保列表中元素是有序的，否则结果是不正确的。

12.7 Comparable 接口与 Comparator 接口

上一节中， Integer 类型的对象可以按照数值大小排序，那么一般对象是否也能排序呢？比如前面例子中用到的 Student 对象。可以做一个试验，将 Collections 的 sort 方法应用于 Student 对象，代码如下：

```

// Test5.java
import java.io.*;
import java.util.*;
class Test5 {
    private List<Student> students = new ArrayList<Student>();
    public static void main(String[] args) throws IOException {
        //读入学生数据，保存到 Student 对象数组中
        Test5 t = new Test5();
        FileReader fin = new FileReader("Students.txt");
        BufferedReader in = new BufferedReader(fin);
        String line = in.readLine();
        while(line != null) {
            line = line.trim();          // 去除字符串中前后空格
            if( line.length() > 0 ) {   // 检查是否是空行
                Student student = new Student();
                student.parseStudent(line);
                t.students.add(student);
            }
            line = in.readLine();
        }
        in.close();
        fin.close();
        // 打乱后再排序
        Collections.shuffle(t.students);
        Collections.sort(t.students);
        t.printStudents();
    }
    public void printStudents() {
        Iterator<Student> iter = students.iterator();
        while(iter.hasNext()) {
            Student s = iter.next();
            System.out.println(s);
        }
    }
}

class Student {
    private String id;
    private String name;
    private String department;
    public void parseStudent(String str) {
        int tokenCount;
        StringTokenizer t = new StringTokenizer(str);
        tokenCount = t.countTokens();

```

```

        id = t.nextToken();           //学号
        name = t.nextToken();        //姓名
        department = t.nextToken();   //学院
    }
    public String toString() {
        String s = id + " " + name;
        for(int l = s.length();l<21;l++) s += " ";
        return s + department;
    }
}

```

编译上面的代码，编译器会报一个错误，如下图：

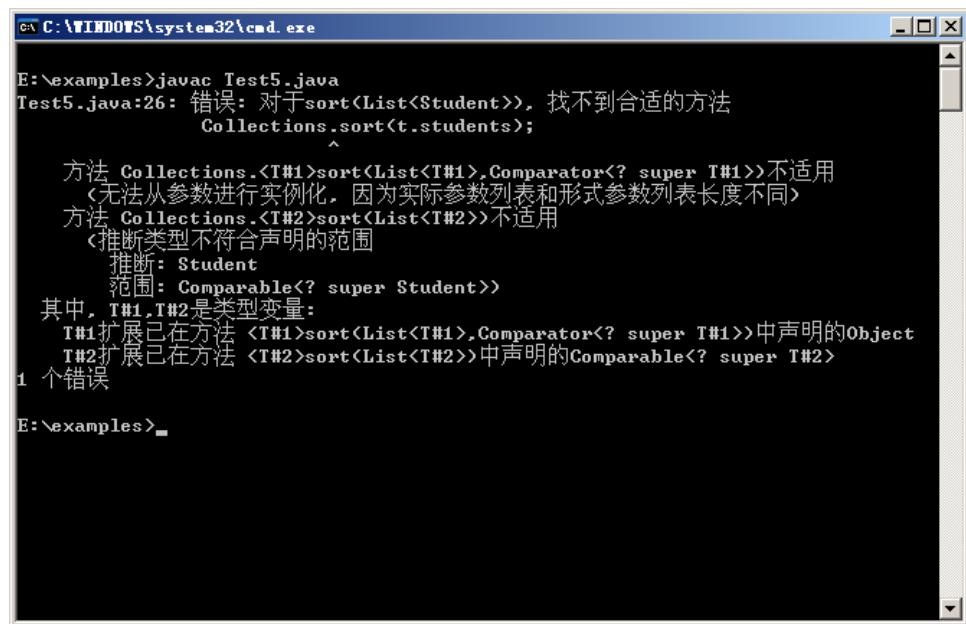


图 12-15 Collections.sort 方法的要求

察看 Collections.sort 方法的帮助文档，就明白造成这个错误的原因：sort 方法应用的对象必须实现 Comparable 接口。从理论上来说，能够排序的前提条件是任意两个对象之间能够比较大小。Java 中定义对象之间比较大小功能的就是 Comparable 接口，该接口中定义了一个方法需要实现：

```
public int compareTo(Object o);
```

该方法的要求是将当前对象与另一个对象 o 比较，当前对象大于、等于、小于另一个对象时分别返回正整数、0、负整数。

因此，要想对 Student 类型的对象排序，就要为 Student 类实现 Comparable 接口。具体如何实现，要看功能需要，比如是按学号排序，还是按姓名排序。实现按姓名排序的 Student 类的代码如下：

```

class Student implements Comparable{
    private String id;
    private String name;
    private String department;
}

```

```

public void parseStudent(String str) {
    int tokenCount;
    StringTokenizer t = new StringTokenizer(str);
    tokenCount = t.countTokens();
    id = t.nextToken();           //学号
    name = t.nextToken();         //姓名
    department = t.nextToken();   //学院
}
public String toString() {
    String s = id + " " + name;
    for(int l = s.length();l<21;l++) s += " "; // 对齐
    return s + department;
}
public int compareTo(Object o) {
    Student stud = (Student)o;
    return name.compareTo(stud.name); // 将当前对象的 name 与另一个对象
                                    // 的 name 比较。
}
}

```

实现 Comparable 时也可以添加类型检查，代码如下：

```

class Student implements Comparable<Student> { // 有类型检查
    private String id;
    private String name;
    private String department;
    public void parseStudent(String str) {
        int tokenCount;
        StringTokenizer t = new StringTokenizer(str);
        tokenCount = t.countTokens();
        id = t.nextToken();           //学号
        name = t.nextToken();         //姓名
        department = t.nextToken();   //学院
    }
    public String toString() {
        String s = id + " " + name;
        for(int l = s.length();l<21;l++) s += " "; //对齐
        return s + department;
    }
    public int compareTo(Student o) { // 参数直接为 Student 类型
        return( name.compareTo(o.name) );
    }
}

```

用 Comparable 接口只能实现按一种标准排序。如果在一个程序中，需要同时实现按多种标

准排序，比如有时按学号排序、有时按姓名排序，此时应该如何实现？Java 提供的方法是使用不同的比较器。

前面的例子中用的 `sort` 方法只有一个参数，就是要排序的数据集合。其实 `sort` 方法还有一个重载，有两个参数，是要排序的数据集合以及比较器。比较器是一个实现了 `Comparator` 接口的对象，可以为 `sort` 方法提供比较两个对象大小的方法。这就好比是，`Comparable` 接口让两个对象自己可以比较大小，而比较器是让两个对象之外的第三方来比较大。

为了使用比较器，必须编写一个比较器类。因为比较器需要访问被比较的对象的数据，所以可以将比较器类写为内部类。例如，`Student` 对象的比较器类可以作为 `Student` 类的内部类，而内部类又可以是静态内部类和动态内部类，所以下面的例子写了两个比较器，分别按姓名和院系排序，并且一个作为静态内部类、一个作为动态内部类，代码如下：

```
class Student {  
    private String id;  
    private String name;  
    private String department;  
    public void parseStudent(String str) {  
        int tokenCount;  
        StringTokenizer t = new StringTokenizer(str);  
        tokenCount = t.countTokens();  
        id = t.nextToken();           //学号  
        name = t.nextToken();         //姓名  
        department = t.nextToken();   //学院  
    }  
    public String toString() {  
        String s = id + " " + name;  
        for(int l = s.length();l<21;l++) s += " "; //对齐  
        return s + department;  
    }  
    public static class NameComparator implements Comparator<Student> {  
        public int compare(Student s1, Student s2) {  
            return s1.name.compareTo(s2.name);  
        }  
    }  
    public class DepartComparator implements Comparator<Student> {  
        public int compare(Student s1, Student s2) {  
            return s1.department.compareTo(s2.department);  
        }  
    }  
}
```

用静态内部类创建比较器对象时，可以用外面嵌套类的名字加上内部类的名字，代码如下：

```
Comparator<Student> c = new Student.NameComparator(); // 静态内部类创建对象  
Collections.sort(t.students,c); // 使用比较器排序
```

用动态内部类创建比较器对象时，要用外面嵌套类的一个对象加上内部类的名字，代码如下：

```
Student student = new Student();
Comparator<Student> c = student.new DepartComparator(); // 动态内部类创建对象
Collections.sort(t.students,c); // 使用比较器排序
```

比较器还可用于在数据集合中查找特定对象。比如，在 `Student` 对象的数据集合中，查找 `name` 字段等于某个字符串的对象，也就是按姓名查找。因为用 `binarySearch` 方法查找时，需要提供一个对象作为查找标准，但此时只知道 `name` 字段，所以需要构造一个 `Student` 对象，其 `name` 字段值为要查找的学生的姓名，`id` 和 `department` 字段的值在查找过程中不会用到，都为空即可。并且查找前要用姓名比较器对数据集合进行排序，查找时也要提供姓名比较器。具体代码如下：

```
Comparator<Student> c = new Student.NameComparator();
Collections.sort(students,c);
Student student = new Student();
student.setName("杨晓非");
int index = Collections.binarySearch(students,student,c);
```

查找的返回值是对象在数据集合中的位置，再通过数据集合的 `get` 方法就可以得到要查找的对象，代码如下：

```
students.get(index)
```

完整的代码见 `Test9.java`。

`Collections` 类中提供的是对数据集合的常用操作，类似的，`Arrays` 类提供对数组的常用操作。使用 `Arrays` 类的例子见 `Test10.java`。

第十三章 窗口程序

13.1 概述

Java 类库中和创建窗口程序有关的包有两个：`java.awt` 和 `javax.swing`。

`awt` 的意思是 `Abstract Window Toolkit`，是在 Java 平台一出现时就有的。但是，`awt` 中的组件是利用底层操作系统(如 windows、Linux)的组件来实现的，处理图形用户界面元素的方法是把这些元素的创建和行为委托给每个目标平台上的本地 GUI 处理。由于不同操作系统平台提供的图形用户界面元素都不一样，所以 `awt` 包功能有限，不同平台上界面风格不一致，而且在跨平台时有许多 Bug。

鉴于此，在 `JDK1.1` 以后，Java 提供了 `swing` 包，用纯 Java 代码实现了包中所有的组件，用户界面元素都绘制在空白窗口上，绘制和行为都由 `swing` 类自己完成。各平台之间唯一不同的就是最外层窗口的创建。这样，就解决了 Java 图形用户界面程序跨平台的问题。所以，现在编写窗口程序，都使用 `swing` 组件，尽管 `awt` 的用户界面组件仍然可以用，但是建议最

好不要使用。

在窗口程序中响应用户的操作需要用到事件处理模型。Java 目前仍然沿用 awt 的事件处理模型。awt 的事件处理模型在 Java1.1 版进行了大的改动后，到目前的版本基本没变。

综上所述，现在编写 Java 图形界面程序，使用 swing 组件 + awt 事件处理模型。

13.2 创建窗口

每个图形用户界面程序都有一个窗口，Java 中顶层窗口称作框架（Frame），这个窗口是由操作系统绘制和管理的。Swing 包中用于创建框架的类是 JFrame。Swing 包中的组件类一般以 J 开头，以区别于原来 awt 包中的组件。例如，awt 包中用于创建框架的类是 Frame。

一个窗口就是一个 JFrame 的对象，所以要创建一个窗口，只需要创建一个 JFrame 的对象，并调用它的 setVisible 方法即可。创建一个最简单的窗口的代码如下：

```
import javax.swing.*;
public class Hello {
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setVisible(true);
    }
}
```

方法 setVisible 可以设置窗口可见或不可见，由 boolean 型参数控制。运行后出现的窗口如图：



图 13-1 最简单的窗口

窗口很小，没有内容，而且点了右上角的关闭按钮（“×”）后窗口没了但程序还不退出。之所以会这样，是因为窗口的很多属性都用了默认值。在用 frame.setVisible(true) 之前，可以用 JFrame 类的方法来设置窗口的属性，添加属性设置语句后代码如下：

```
//Hello.java
import javax.swing.*;
public class Hello {
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setSize(300,200);      // 设置窗口宽 300 个像素，高 200 个像素
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // 设置窗口关闭操作
        frame.getContentPane().add(new JLabel("Hello World!")); // 在窗口中添加文本
        frame.setVisible(true);
    }
}
```

再运行程序，窗口如图：



图 13-2 普通的窗口

基于面向对象的思想，应该将与一个窗口有关的代码封装到一个类中，这个类称为窗口类，通常是 `JFrame` 的子类。上面的代码应改写为：

```
// HelloWin.java
import javax.swing.*;
public class HelloWin extends JFrame {      // 窗口类通常是 JFrame 的子类
    public static void main(String[] args){
        JFrame helloFrame = new HelloWin(); // 使用窗口时只需创建窗口对象
        helloFrame.setVisible(true);
    }
    public HelloWin(){      // 在构造函数中设置窗口的属性
        setSize(300,200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        getContentPane().add(new JLabel("Hello World!"));
    }
}
```

上述代码的运行结果没有任何变化，但比较前后两种写法可以看出，采用窗口类的写法更加清晰、简洁、灵活。一般在窗口类的构造函数中设置窗口的属性，添加窗口中需要的组件。使用窗口时只需要创建窗口对象，再调用其 `setVisible` 方法即可。这种方式使得一个窗口成为一个单独的模块，可以设计和使用分开，并且这个窗口可以作为一个类直接在其它程序中使用。

本章只介绍如何设计窗口，至于如何响应用户操作将在下一章事件处理中介绍。

13.3 设置窗口属性

设置窗口属性需要使用 `JFrame` 类的方法。`JFrame` 类的继承关系如图：

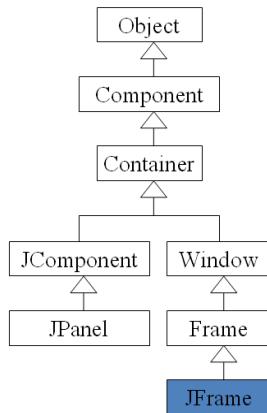


图 13-3 JFrame 类的继承层次

JFrame 类中用于设置窗口属性的方法主要有以下这些：

继承自 Frame

- `public void setTitle(String title):` 设置标题
- `public void setIconImage(Image image):` 设置图标
- `public void setMenuBar(MenuBar mb):` 设置菜单
- `public void setResizable(boolean resizable):` 设置可变大小
- `public void setState(int state):` 设置窗口状态

继承自 Component

- `public void setLocation(int x,int y):` 设置位置
- `public void setSize(int width,int height):` 设置尺寸
- `public void setBounds(int x,int y,int width,int height)`

可以通过下面的例子来演示这些方法如何使用。创建一个窗口，使它的宽和高都占整个屏幕的一半，初始位置在屏幕中央，标题为“Hello”，图标为一幅图片，不能改变大小。代码如下：

```

// HelloWin1.java
import javax.swing.*;
import java.awt.*;
public class HelloWin1 extends JFrame {
    public HelloWin1()  {
        Toolkit kit = Toolkit.getDefaultToolkit();      // 获取默认 Toolkit
        Dimension screenSize = kit.getScreenSize(); // 获取屏幕分辨率
        setSize(screenSize.width/2,screenSize.height/2);
        setLocation(screenSize.width/4,screenSize.height/4);
        Image img = kit.getImage("graph.bmp");
        setIconImage(img);
        setTitle("Hello");
        setResizable(false);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}

```

要满足这些条件，首先要获取整个屏幕的大小，即分辨率信息，可以用 `Toolkit` 类的方法。首先通过 `Toolkit` 类的静态方法 `getDefalutToolkit` 获取默认的 `Toolkit` 对象，然后再用该对象的 `getScreenSize` 方法获取屏幕分辨率。获取到的分辨率是一个 `Dimension` 对象，通过它的 `width` 和 `height` 属性就可以得出屏幕的宽和高。有了屏幕的宽和高，就可以计算窗口的宽和高，以及初始位置。窗口的位置用左上角在屏幕坐标系中的位置表示，单位为像素。屏幕坐标系以屏幕左上角为(0,0)点，向右为 X 轴，向下为 Y 轴。计算好以后，使用 `setSize` 和 `setLocation` 方法来设置。如图：

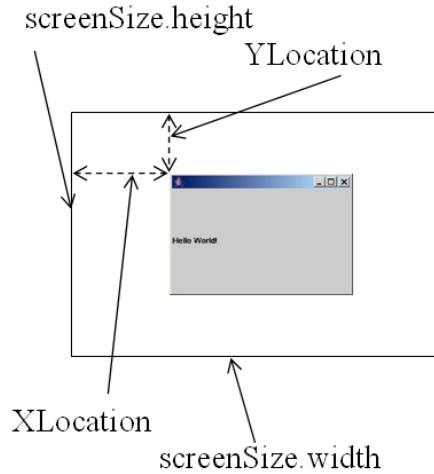


图 13-4 窗口在屏幕上的位置坐标

设置图标用 `setIconImage` 方法，参数是一个 `Image` 对象。`Image` 对象代表了内存中的一幅图片，可以使用前面的 `Toolkit` 对象的 `getImage` 方法，从硬盘上把一个图片文件装到内存中，构造出一个 `Image` 对象。可以使用任意大小的图片作为窗口的图标，Java 会将其缩小到合适的尺寸。本例子中使用了一幅 `graph.bmp` 图片，运行例子时这个图片文件应该放在同一个目录下，即执行 `java` 命令时的当前目录。

13.4 为窗口添加菜单

窗口菜单涉及的概念以及对应的类包括菜单栏（`JMenuBar`）、菜单（`JMenu`）、菜单项（`JMenuItem`）、快捷键和加速键。如图：

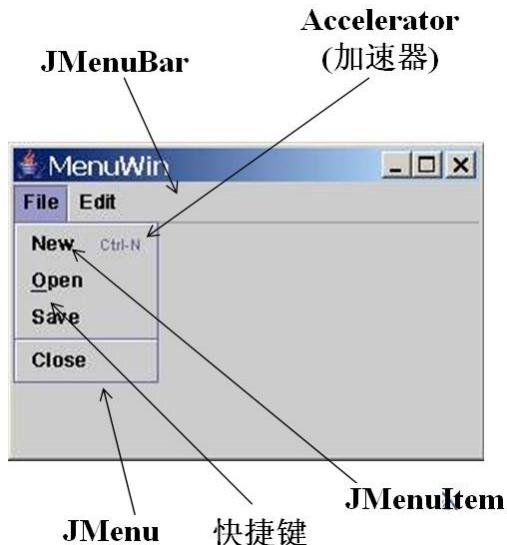


图 13-5 窗口菜单

为窗口添加菜单需要按以下步骤依次创建上述对象：

1. 创建一个 **JMenuBar** 对象，用 **Frame** 的 **setJMenuBar** 方法设置为框架菜单。
2. 创建 **JMenu** 对象，添加到 **JMenuBar** 对象中。
3. 创建 **JMenuItem** 对象，添加到 **JMenu** 对象中。

创建图中菜单的代码如下：

```
//MenuWin.java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class MenuWin extends JFrame {
    public static void main(String[] args){
        JFrame menuFrame = new MenuWin();
        menuFrame.setVisible(true);
    }

    public MenuWin() {
        setTitle("MenuWin");
        setSize(300,200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JMenuBar mainMenu = new JMenuBar(); //创建一个菜单条
        setJMenuBar(mainMenu);           //设置为框架的菜单条
        JMenu fileMenu = new JMenu("File"); //创建一个 File 菜单
        mainMenu.add(fileMenu);          //添加到菜单条中
        JMenuItem newItem = new JMenuItem("New"); //创建一个菜单项
        newItem.setAccelerator(KeyStroke.getKeyStroke(new
Character('N'),InputEvent.CTRL_MASK)); //设置菜单项的快捷键
    }
}
```

```

        fileMenu.add(newItem); //将该菜单项添加到 File 菜单中
        fileMenu.add(new JMenuItem("Open",'O')); //在 File 菜单中添加一个 Open 菜单项
    }

    fileMenu.add(new JMenuItem("Save"));
    fileMenu.addSeparator();           //在 File 菜单中添加一条分隔线
    fileMenu.add(new JMenuItem("Close"));

    JMenu editMenu = new JMenu("Edit"); //创建一个 Edit 菜单
    mainMenu.add(editMenu);           //添加到菜单条中
    editMenu.add("Cut");
    editMenu.add("Copy");
    editMenu.add("Paste");
}
}

```

给菜单项添加加速键用 `JMenuItem` 的 `setAccelerator` 方法，参数是一个 `KeyStroke` 对象。该对象由 `KeyStroke` 类的静态方法 `getKeyStroke` 创建，创建时给出两个参数，第一个参数是字母，第二个参数是组合键。字母通过 `Character` 对象给出，组合键通过 `InputEvent` 类中定义的静态常量给出。

给菜单项添加快捷键的方法是在构造菜单项对象时，在 `JMenuItem` 的构造函数中给出快捷键字母，通常是菜单项文本中的一个字符，将会添加下划线标识。

加速键和快捷键的区别在于，加速键在菜单没有弹出时就起作用，而快捷键只有在菜单弹出后才起作用。比如本例中，在窗口中直接按 `CTRL-N` 就相当于在 `File` 菜单中点了 `New`，即使 `File` 菜单没有弹出来也起作用；而只有 `File` 菜单弹出来后，再按 `O` 键才会点中 `Open` 项。

除了上面的最基本的菜单功能外，还可以为菜单添加其他功能，包括：菜单项图标、复选框菜单项、单选钮菜单项、多级菜单、菜单项启用和禁用、删除菜单项。使用这些功能的例子代码如下：

```

//MenuWin1.java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class MenuWin1 extends JFrame {
    public static void main(String[] args) {
        JFrame menuFrame = new MenuWin1();
        menuFrame.setVisible(true);
    }

    public MenuWin1() {
        setTitle("MenuWin");
        setSize(300,200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}

```

```

JMenuBar mainMenu = new JMenuBar();
setJMenuBar(mainMenu);
JMenu fileMenu = new JMenu("File");
mainMenu.add(fileMenu);
JMenuItem newItem = new JMenuItem("New");
newItem.setIcon(new ImageIcon("graph.jpg")); //设置图标
newItem.setAccelerator(KeyStroke.getKeyStroke(new
Character('N'),InputEvent.CTRL_MASK));
fileMenu.add(newItem);
fileMenu.add(new JMenuItem("Open",'O'));
//创建一组单选菜单项
JMenuItem saveItem = new JRadioButtonMenuItem("Save");
JMenuItem closeItem = new JRadioButtonMenuItem("Close");
fileMenu.add(saveItem);
fileMenu.addSeparator();
fileMenu.add(closeItem);
ButtonGroup bg = new ButtonGroup(); //逻辑功能上分为一组
bg.add(saveItem);
bg.add(closeItem);
//创建一组复选菜单项
JMenu editMenu = new JMenu("Edit");
mainMenu.add(editMenu);
editMenu.add(new JCheckBoxMenuItem("Cut"));
editMenu.add(new JCheckBoxMenuItem("Copy"));
JMenuItem pasteItem = new JCheckBoxMenuItem("Paste");
editMenu.add(pasteItem);
pasteItem.setEnabled(false); //禁用菜单项
//创建多级菜单
JMenu optionMenu = new JMenu("Options");
optionMenu.add("ReadOnly");
optionMenu.add("Insert");
optionMenu.add("Overwrite");
editMenu.addSeparator();
editMenu.add(optionMenu);
//删除菜单项
// optionMenu.remove(1);
}
}

```

为菜单项设置图标用 `JMenuItem` 的 `setIcon` 方法，参数可以是一个 `ImageIcon` 对象，可以用一个图片文件生成。

复选框菜单项不是 `JMenuItem` 对象，而是其子类 `JCheckBoxMenuItem` 的对象。类似的，单选钮菜单项是 `JRadioButtonMenuItem` 对象。一个菜单中可以同时选中多个复选框菜单项，

在菜单项前面的方框中打勾表示。但是，一个菜单中只能选中一个单选钮菜单项，在菜单项前面的圆圈中填充圆点表示。若再选中其它的单选钮菜单项，则自动取消原有的单选钮菜单项的选中状态。

多级菜单是当选中一个菜单项时，会再弹出一个菜单来。所以，实际上是用一个菜单代替了原来一个菜单项的位置，子菜单的标题会作为父菜单的一个菜单项显示。要创建多级菜单，只要把一个菜单当作菜单项加到另一个菜单中即可。

菜单项的启用和禁用使用的是 `JMenuItem` 的 `setEnabled` 方法，其 `boolean` 型的参数控制菜单项的状态。

从菜单中移除菜单项用的是 `JMenu` 的 `remove` 方法，参数是菜单项在菜单中的顺序位置。

13.5 向窗口中添加组件

程序与用户的交互，要通过窗口中的组件来完成。每个组件都是一个对象。每种类型的组件在 `swing` 包中都有一个类对应。`Swing` 包中常用的组件及其对应的类包括：

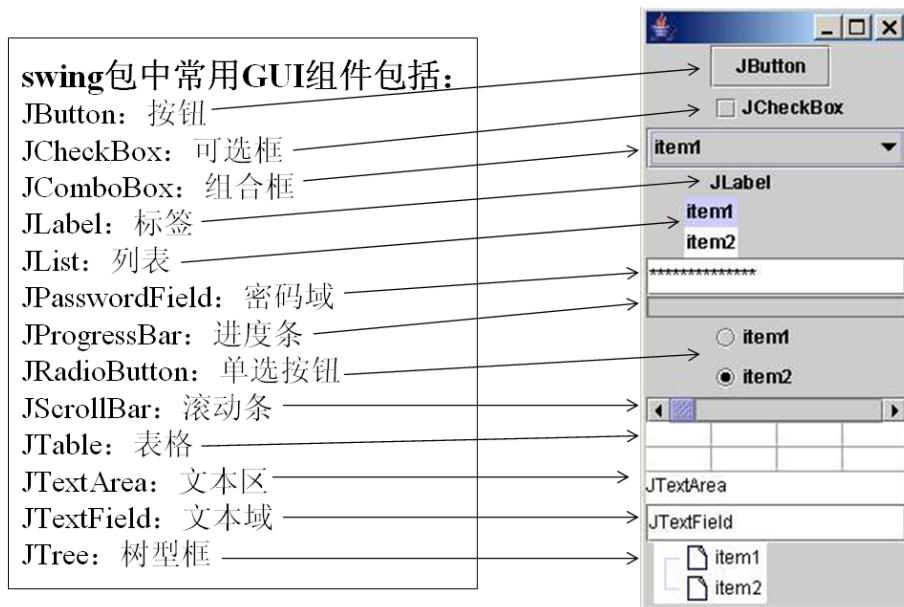


图 13-6 常用 GUI 组件

各组件的外观和内容都是由组件的属性决定的，如按钮大小、上面的文字等。改变和读取组件的属性通过组件的方法进行，设置属性一般使用 `set` 开头的方法，如 `JTextField` 组件的 `setText` 方法可以设置文本输入框中的内容；读取组件属性使用 `get` 开头的方法，如 `JTextField` 组件的 `getText` 方法可以读取用户输入的内容。

带有列表内容的组件，如组合框 `JComboBox`，其列表内容一般通过 `addItem` 和 `removeItem` 方法添加和删除。具体的方法还需查阅 Java 帮助文档。

组件对象并不是直接添加到窗口对象，而是添加到窗口的内容容器中。如图：

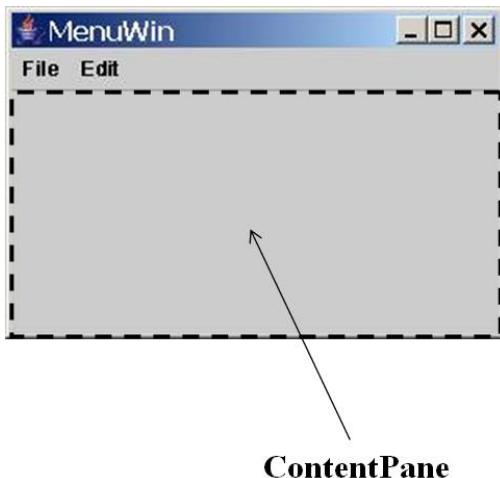


图 13-7 窗口的内容容器

窗口的内容容器是一个 `Container` 对象，通过窗口对象的 `getContentPane` 方法获得。要把组件加入容器，要使用容器的 `add` 方法，参数是要加入的组件。例如，要在窗口中显示字符串“Hello World！”，需要把一个 `JLabel` 组件添加到内容容器中。代码如下：

```
// ContentTest.java
import javax.swing.*;
import java.awt.*;
public class ContentTest extends JFrame {
    public static void main(String[] args) {
        JFrame helloFrame = new ContentTest();
        helloFrame.setVisible(true);
    }
    public ContentTest() {
        setSize(300,200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container contentPane = getContentPane();
        contentPane.add(new JLabel("Hello World!"));
    }
}
```

窗口中可以有很多组件，那么如何安排它们的大小和位置呢？有两种方式，第一种方式是直接设置组件的大小和位置属性，可以称为“手动方式”，第二种方式是使用 Java 提供的布局管理器，可以称为“自动方式”。本节接下来介绍“手动方式”，布局管理器将在下一节中介绍。

因为 Java 的容器都有默认的布局管理器，它会自动计算组件的大小和位置。所以，要使用手动方式安排组件大小和位置，就必须先关掉布局管理器，将布局管理器设为 `null` 即可。关掉布局管理器后就可以使用每个组件的 `setLocation` 和 `setSize` 方法来安排其位置和大小。如果不关掉布局管理器，使用这两个方法设置组件的位置和大小是没有效果的，因为会被布局管理器重新计算。采用的坐标系是窗口内容容器左上角为坐标原点(0,0)，向右为 X 轴，向下

为 Y 轴，单位为像素点。如图：

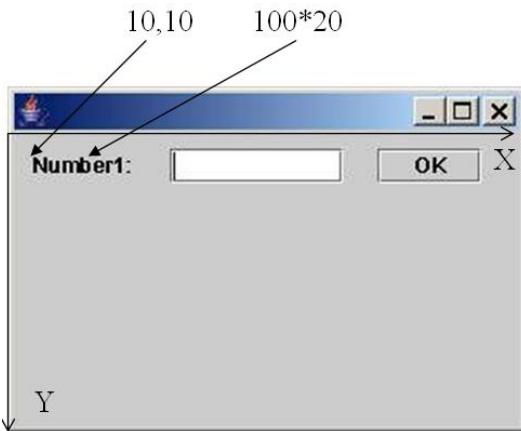


图 13-8 组件在容器中的绝对位置和尺寸

创建上面的窗口的代码如下：

```
// LocationTest.java
import javax.swing.*;
import java.awt.*;
public class LocationTest extends JFrame {
    public static void main(String[] args) {
        JFrame helloFrame = new LocationTest();
        helloFrame.setVisible(true);
    }
    public LocationTest() {
        setSize(300,200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container contentPane = getContentPane();
        contentPane.setLayout(null);
        JLabel label1 = new JLabel("Number1:");
        label1.setLocation(10,10);
        label1.setSize(80,20);
        JTextField textField1 = new JTextField();
        textField1.setBounds(90,10,100,20);
        JButton button1 = new JButton("OK");
        button1.setBounds(210,10,60,20);
        contentPane.add(label1);
        contentPane.add(textField1);
        contentPane.add(button1);
    }
}
```

因为需要访问窗口中的组件以获得其属性，所以组件变量一般需要声明为窗口类的成员变量，而不能声明为方法中的临时变量。

13.6 布局管理器

13.6.1 概述

使用“手动方式”安排组件大小和位置虽然直接，但是有个很大的问题，就是无法根据窗口大小自动排列窗口元素。很多情况下窗口大小是可以变化的，此时窗口内的组件也应该随之调整，比如 Windows 的资源管理器。更重要的是，Java 是跨平台的语言，所以其窗口程序运行的环境千差万别，比如可能运行在各种屏幕分辨率的手机上、Pad 上，此时更需要一种机制来自动调整组件的大小和位置。

布局管理器就是 Java 为实现这一目的而为容器（Container）添加的一个对象。每个容器都可以有一个布局管理器对象，当容器大小变化时，此对象将自动调整容器中各组件的大小和位置，目的是使各组件在不同平台上排列和显示都正常。为容器设置布局管理器使用其 `setLayout` 方法，参数是一个布局管理器对象。语句 `setLayout(null)` 会将容器的布局管理器设为空，即不使用布局管理器。

Java 类库中提供了多个布局管理器类，用于创建不同的布局管理器对象，满足不同的布局要求。常用的布局管理器有以下几个：

- `FlowLayout`: 流式布局管理器
- `BorderLayout`: 边界布局管理器
- `GridLayout`: 网格布局管理器
- `GridBagLayout`: 网格组布局管理器
- `BoxLayout`: 箱式布局管理器

此外，用户还可以创建自己的布局管理器，只要实现 `LayoutManager` 接口。下面就逐个介绍这些布局管理器是如何管理组件大小和位置的。

为满足布局管理器自动布局的需要，与组件大小有关的属性有以下四个：

- `Size`: 组件的当前实际尺寸
- `MaximumSize`: 组件的最大尺寸
- `MinimumSize`: 组件的最小尺寸
- `PreferredSize`: 组件的最佳尺寸，缺省值和组件内容有关。

`Size` 属性保存组件的实际大小，我们前面用到的 `setSize` 方法就是设置这个属性。后面三个属性分别保存组件可以调整到的最大尺寸、最小尺寸和最佳尺寸。如图：

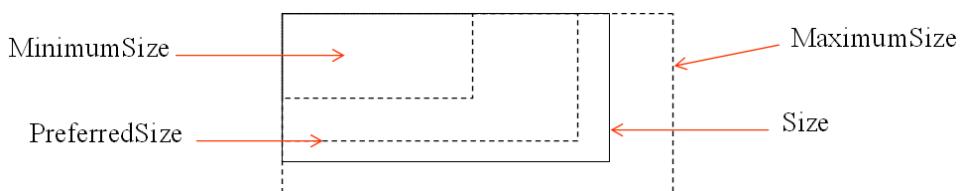


图 13-9 组件和尺寸有关的属性

布局管理器会根据这三个属性以及当前容器的大小来计算组件的实际大小，并保存到 `Size` 属性中，不同的布局管理器计算方法不一样。因此，在有布局管理器的情况下，设置 `Size`

属性是不起作用的，因为布局管理器会自动重新计算，并覆盖掉 `Size` 的值。上面的四个尺寸属性都可以通过 `get/set` 方法来读取/设置，例如 `getSize`、`setSize`、`getMaximumSize`、`setMaximumSize`。

13.6.2 流式布局管理器——FlowLayout

流式布局管理器按如下方式安排容器中的组件：

- 将各组件设置为最佳尺寸（`PreferredSize`，有时由组件内容决定）；
- 按组件加入容器的顺序在一行上水平排列组件，直到没有足够剩余空间，另起一行，直至所有组件都排完；
- 可以决定组件在一行上是左对齐、居中、右对齐、或两边对齐；
- 可以决定组件的水平间距和垂直间距。

流式布局方式类似于在 Word 里写文本，是从左到右排版，一行排满后转到下一行，可以决定是两边对齐、左对齐、右对齐、还是居中，可以设置行距和文字间距。

一个流式布局的例子如下：

```
// FlowLayoutTest.java
import javax.swing.*;
import java.awt.*;
public class FlowLayoutTest extends JFrame {
    public static void main(String[] args) {
        JFrame helloFrame = new FlowLayoutTest();
        helloFrame.setVisible(true);
    }
    public FlowLayoutTest() {
        setSize(300,200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());
        // contentPane.setLayout(new FlowLayout(FlowLayout.LEFT));
        // contentPane.setLayout(new FlowLayout(FlowLayout.LEFT,10,10));
        JLabel label1 = new JLabel("Number1:");
        //无效      label1.setBounds(10,10,80,20);
        JTextField textField1 = new JTextField("12345");
        //无效      textField1.setBounds(90,10,100,20);
        // textField1.setPreferredSize(new Dimension(100,20)); //设置推荐大小
        JButton button1 = new JButton("OK");
        //无效      button1.setBounds(210,10,60,20);
        contentPane.add(label1);
        contentPane.add(textField1);
        contentPane.add(button1);
```

```

        contentPane.add(new JButton("ButtonGreen"));
        contentPane.add(new JButton("ButtonYellow"));
        contentPane.add(new JButton("ButtonRed"));
    }
}

```

运行生成的窗口如图：



图 13-10 流式布局管理器的效果

13.6.3 边界布局管理器——BorderLayout

边界布局管理器如下安排容器中的组件：

- 将容器划分为东西南北中五个区域，如图，每个区域可以放置一个组件，或者不放；
- 南北两个组件的宽度随容器宽度变化，高度采用最佳大小；
- 东西两个组件的高度随容器高度变化，宽度采用最佳大小；
- 中间组件占满剩余空间；
- 可以决定组件的水平间距和垂直间距。

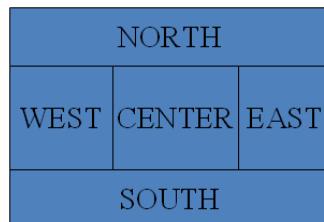


图 13-11 边界布局管理器

边界布局中最多只能放置五个组件，每个组件在放入容器时必须指出要放入哪个区域。因此，使用边界布局管理器时，容器的 add 方法需要两个参数，一个是组件对象，一个是区域标识。区域标识通过 BorderLayout 类的五个静态常量 NORTH、SOUTH、WEST、EAST、CENTER 来指定。如果将两个组件放入同一个区域，那么后面的组件会替换掉前面的组件。如果一个区域中没有组件，那么这个区域就没有了，其位置会被其它区域占据。这种布局方式的一个实际例子是 Windows 资源管理器，其工具栏占据北区，状态栏占据南区（如果有的话），导航栏占据西区，东区为空，文件列表占据中央，当窗口大小变化时，各区域会随之变化。

边界布局管理器的例子如下：

```
// BorderLayoutTest.java
```

```

import javax.swing.*;
import java.awt.*;
public class BorderLayoutTest extends JFrame {
    public static void main(String[] args) {
        JFrame helloFrame = new BorderLayoutTest();
        helloFrame.setVisible(true);
    }

    public BorderLayoutTest() {
        setSize(300,200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container contentPane = getContentPane();
        contentPane.setLayout(new BorderLayout());
        // contentPane.setLayout(new BorderLayout(10,10)); //水平距离和垂直距离
        JLabel label1 = new JLabel("Number1:");
        label1.setPreferredSize(new Dimension(80,20));
        contentPane.add(label1,BorderLayout.WEST);
        JTextField textField1 = new JTextField("");
        contentPane.add(textField1,BorderLayout.NORTH);
        JButton button1 = new JButton("OK");
        contentPane.add(button1,BorderLayout.EAST);
        contentPane.add(new JButton("ButtonGreen"),BorderLayout.SOUTH);
        contentPane.add(new JButton("ButtonYellow"),BorderLayout.CENTER);
    }
}

```

运行生成的窗口如图：

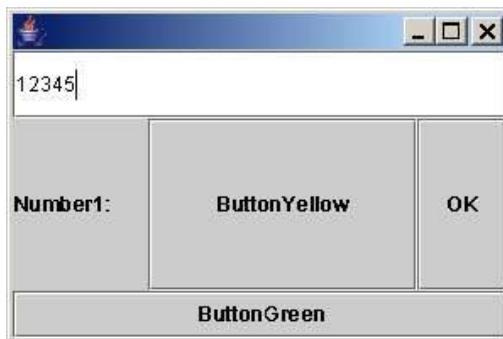


图 13-12 边界布局管理器的效果

13.6.4 网格布局管理器——GridLayout

网格布局管理器如下安排容器中的组件：

- 将容器平均划分为一个 m 行 n 列的表格，各方格大小相同。

- 组件按加入的顺序被依次放入第一行第一格，第一行第二格，……
- 可以决定网格的水平间距和垂直间距。

行数 m 和列数 n 在创建网格布局管理器时作为构造函数的参数给出。如果需要设置水平间距和垂直间距，也在创建网格布局管理器时作为构造函数的参数给出。边界布局管理器的例子如下：

```
// GridLayoutTest.java
import javax.swing.*;
import java.awt.*;
public class GridLayoutTest extends JFrame {
    public static void main(String[] args) {
        JFrame helloFrame = new GridLayoutTest();
        helloFrame.setVisible(true);
    }
    public GridLayoutTest() {
        setSize(300,200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container contentPane = getContentPane();
        // contentPane.setLayout(new GridLayout(4,3)); //4 行 3 列
        contentPane.setLayout(new GridLayout(4,3,5,5)); //水平距离和垂直距离
        contentPane.add(new JButton("7"));
        contentPane.add(new JButton("8"));
        contentPane.add(new JButton("9"));
        contentPane.add(new JButton("4"));
        contentPane.add(new JButton("5"));
        contentPane.add(new JButton("6"));
        contentPane.add(new JButton("1"));
        contentPane.add(new JButton("2"));
        contentPane.add(new JButton("3"));
        contentPane.add(new JButton("*"));
        contentPane.add(new JButton("0"));
        contentPane.add(new JButton("R"));
    }
}
```

运行生成的窗口如图：



图 13-13 网格布局管理器的效果

13.6.5 网格组布局管理器——**GridBagLayout**

网格组布局管理器如下安排容器中的组件：

- 将容器平均划分为一个 m 行 n 列的表格，各方格大小相同。
- 一个组件可以占用多个方格组成的矩形区域。

组件占用哪些格子由一个 **GridBagConstraints** 对象指定，该对象的属性如下：

- **gridx:** 左上角方格 x 坐标，即在第几列
- **gridy:** 左上角方格 y 坐标，即在第几行
- **gridwidth:** 横向占用几个格
- **gridheight:** 纵向占用几个格
- **fill:** 组件在区域内的填充方式；
- **anchor:** 组件在区域内的位置；
- **weightx:** 伸缩时的横向权重；
- **weighty:** 伸缩时的纵向权重；

网格组布局管理器的例子如下：

```
// GridBagLayoutTest.java
import javax.swing.*;
import java.awt.*;
public class GridBagLayoutTest extends JFrame {
    public static void main(String[] args) {
        JFrame helloFrame = new GridBagLayoutTest();
        helloFrame.setVisible(true);
    }
    public GridBagLayoutTest() {
        setSize(300,200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container contentPane = getContentPane();
        contentPane.setLayout(new GridBagLayout());
        GridBagConstraints constraints = new GridBagConstraints();
        constraints.weightx = 100;
        constraints.weighty = 100;
        constraints.fill = GridBagConstraints.BOTH;
        constraints.anchor = GridBagConstraints.CENTER;
        constraints.gridx = 0;
        constraints.gridy = 0;
        constraints.gridwidth = 2;
        constraints.gridheight = 1;
        contentPane.add(new JButton("1"),constraints);
    }
}
```

```
constraints.weightx = 100;
constraints.weighty = 100;
constraints.fill = GridBagConstraints.BOTH;
constraints.anchor = GridBagConstraints.CENTER;
constraints.gridx = 2;
constraints.gridy = 0;
constraints.gridwidth = 1;
constraints.gridheight = 2;
contentPane.add(new JButton("2"),constraints);
constraints.weightx = 100;
constraints.weighty = 100;
constraints.fill = GridBagConstraints.BOTH;
constraints.anchor = GridBagConstraints.CENTER;
constraints.gridx = 1;
constraints.gridy = 2;
constraints.gridwidth = 2;
constraints.gridheight = 1;
contentPane.add(new JButton("3"),constraints);
constraints.weightx = 100;
constraints.weighty = 100;
constraints.fill = GridBagConstraints.BOTH;
constraints.anchor = GridBagConstraints.CENTER;
constraints.gridx = 0;
constraints.gridy = 1;
constraints.gridwidth = 1;
constraints.gridheight = 2;
contentPane.add(new JButton("4"),constraints);
constraints.weightx = 100; //若要 Button5 所占方格不改变大小，可将此处设
为 0。
```

```
constraints.weighty = 100;
constraints.fill = GridBagConstraints.BOTH;
constraints.anchor = GridBagConstraints.CENTER;
// constraints.fill = GridBagConstraints.NONE; //若要 Button5 不改变大小，可这样
写
// constraints.anchor = GridBagConstraints.NORTHWEST; //若要 Button5 在区域
左上角，可这样写
constraints.gridx = 1;
constraints.gridy = 1;
constraints.gridwidth = 1;
constraints.gridheight = 1;
contentPane.add(new JButton("5"),constraints);
}
}
```

运行生成的窗口如图：

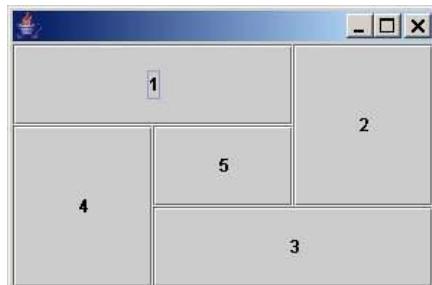


图 13-14 网格组布局管理器的效果

容器中的网格划分如图：

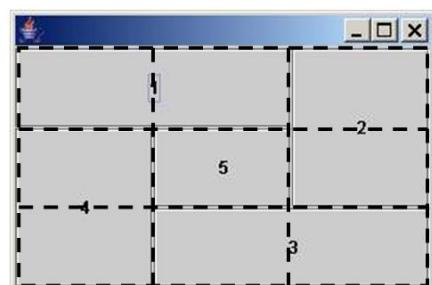


图 13-15 网格组布局管理器的网格

13.6.6 箱式布局管理器——BoxLayout

箱式布局管理器分为水平（横向）和竖直（纵向）两种。水平箱式布局按如下方式安排容器中的组件（竖直箱式布局类似，只是改为纵向）：

- 将容器划分为 n 行，称为 n 个横箱子，各行高度可以不同；
- 每个横箱子内可以水平排列组件；
- 可以插入三种特殊不可见组件——支柱、固定区、胶水，来决定组件的位置。

支柱用于在组件之间支撑起一段间隔。固定区用于在箱子之间加入一段间隔。胶水用于填满组件之间空出的位置。

使用箱式布局时，应创建需要的箱子，然后在每个箱子中放入组件，再把所有箱子加入到容器。一个箱式布局管理器的例子如下：

```
// BoxLayoutTest.java
import javax.swing.*;
import java.awt.*;

public class BoxLayoutTest extends JFrame {
    public static void main(String[] args) {
        JFrame helloFrame = new BoxLayoutTest();
        helloFrame.setVisible(true);
    }
}
```

```
public BoxLayoutTest() {  
    setSize(300,200);  
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
    Container contentPane = getContentPane();  
    contentPane.setLayout(new BoxLayout(contentPane,BoxLayout.Y_AXIS));  
  
    Box hBox1 = Box.createHorizontalBox();//Box1  
    hBox1.add(new JLabel("Name:"));  
    JTextField nameField = new JTextField("");  
    nameField.setPreferredSize(new Dimension(100,20));  
    nameField.setMaximumSize(new Dimension(100,20));  
    hBox1.add(nameField);  
  
    Box hBox2 = Box.createHorizontalBox();//Box2  
    hBox2.add(new JLabel("Password:"));  
    hBox2.add(Box.createHorizontalStrut(10));  
    JPasswordField passwordField = new JPasswordField("");  
    passwordField.setPreferredSize(new Dimension(100,20));  
    passwordField.setMaximumSize(new Dimension(100,20));  
    hBox2.add(passwordField);  
  
    Box hBox3 = Box.createHorizontalBox();//Box3  
    hBox3.add(Box.createGlue());  
    hBox3.add(new JButton("OK"));  
    hBox3.add(Box.createHorizontalStrut(20));  
    hBox3.add(new JButton("Cancel"));  
    hBox3.add(Box.createGlue());  
  
    contentPane.add(Box.createRigidArea(new Dimension(2,10)));  
    contentPane.add(hBox1);  
    contentPane.add(Box.createRigidArea(new Dimension(2,10)));  
    contentPane.add(hBox2);  
    contentPane.add(Box.createRigidArea(new Dimension(2,20)));  
    contentPane.add(hBox3);  
}  
}
```

运行生成的窗口如图：



图 13-16 箱式布局管理器的效果

其中的箱子、支柱、固定区、胶水等不可见部分如图：

支柱、固定区、胶水。

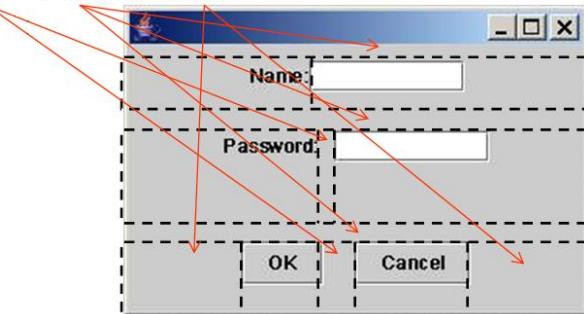


图 13-17 箱式布局管理器中的支柱、固定区和胶水

13.7 面板

面板（JPanel）是一个特殊的组件，从外面看它是一个组件，可以放到其它容器中，而从内部看它又是一个容器，可以容纳其它组件。每个面板可以有自己的布局管理器。有了面板，就实现了容器嵌套。也就是把一个大的区域采用一种布局划分为许多小的区域进行管理，每个小的区域可以再采用一种布局管理。

如图所示的窗口首先采用边界布局管理，按钮的面板占据 SOUTH，文本域占据 CENTER。按钮面板内部采用流式布局，依次排列三个按钮组件。这样，当窗口大小变化时，面板的高度不变，只有宽度随窗口变。

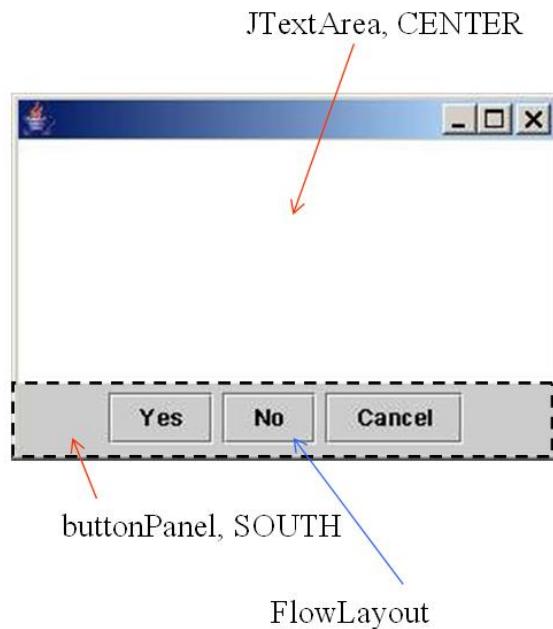


图 13-18 使用面板进行布局管理

实现上述窗口的代码如下：

```
// PanelTest.java
import javax.swing.*;
import java.awt.*;

public class PanelTest extends JFrame {
    public static void main(String[] args) {
        JFrame helloFrame = new PanelTest();
        helloFrame.setVisible(true);
    }

    public PanelTest() {
        setSize(300,200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container contentPane = getContentPane();
        contentPane.setLayout(new BorderLayout());

        JPanel buttonPanel = new JPanel(new FlowLayout());
        buttonPanel.add(new JButton("Load"));
        buttonPanel.add(new JButton("Save"));
        buttonPanel.add(new JButton("Exit"));

        contentPane.add(buttonPanel,BorderLayout.SOUTH);
        contentPane.add(new JTextArea(),BorderLayout.CENTER);
    }
}
```

Java 还提供了一些实现特定功能的面板，常用的包括滚动面板、切分面板和分页面板。

滚动面板（`JScrollPane`）用于实现滚动条的功能，即从一个小窗口中观察大组件时，用滚动条控制显示哪一部分。使用滚动面板的方法很简单，只需先将组件加入滚动面板，再将滚动面板放入容器，相当于在组件的外面套上了一个滚动面板再放到容器中。滚动面板会自动判断，在需要时加上滚动条。

滚动面板的例子如下：

```
// ScrollPaneTest.java
import javax.swing.*;
import java.awt.*;

public class ScrollPaneTest extends JFrame {
    public static void main(String[] args) {
        JFrame helloFrame = new ScrollPaneTest();
        helloFrame.setVisible(true);
    }

    public ScrollPaneTest() {
        setSize(300,200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container contentPane = getContentPane();
        contentPane.setLayout(new BorderLayout());

        JPanel imgPanel = new JPanel();
        imgPanel.setBackground(Color.white);
        imgPanel.setBounds(0,0,600,400);

        JPanel bgPanel = new JPanel();
        bgPanel.setPreferredSize(new Dimension(620,420));
        bgPanel.setLayout(null);
        bgPanel.add(imgPanel);

        JScrollPane imgScrollPane = new JScrollPane(bgPanel);//创建滚动面板

        contentPane.add(imgScrollPane,BorderLayout.CENTER);
    }
}
```

运行生成的窗口如图：

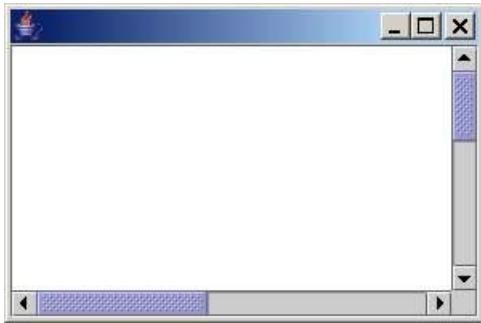


图 13-19 滚动面板

切分面板（JSplitPane）用一个分割条将一个区域一分为二，并且可以通过拖动分割条改变两个区域的大小。切分后每个区域可以放一个组件。切分的方向可以是横向，也可以是纵向。使用切分面板时，将两个组件加入切分面板，再将切分面板放到容器中。一个例子如下：

```
// SplitPaneTest.java
import java.awt.*;
import javax.swing.*;

public class SplitPaneTest extends JFrame {
    public static void main(String[] args) {
        JFrame helloFrame = new SplitPaneTest();
        helloFrame.setVisible(true);
    }
    public SplitPaneTest() {
        setSize(300,200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container contentPane = getContentPane();
        contentPane.setLayout(new BorderLayout());

        JLabel label1=new JLabel("Label1",JLabel.CENTER);
        label1.setBackground(Color.green);
        label1.setOpaque(true);

        JLabel label2=new JLabel("Label2",JLabel.CENTER);
        label2.setBackground(Color.blue);
        label2.setOpaque(true);

        JSplitPane splitPane1=new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,true,label1,label2); //true 表示在拖动过程中，两边的组件是否即时更新
        splitPane1.setDividerLocation(0.3); //分隔条的初始位置
        splitPane1.setOneTouchExpandable(true); //是否有快速改变位置的按钮
        splitPane1.setDividerSize(6); //设置分隔条宽度

        contentPane.add(splitPane1);
    }
}
```

```
    }  
}
```

运行生成的窗口如图：

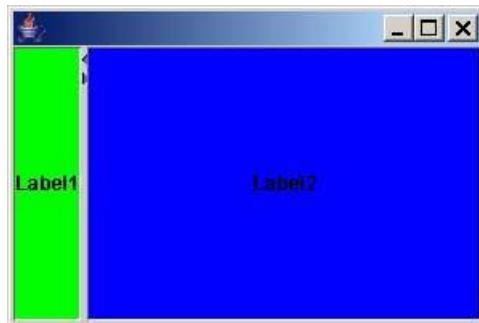


图 13-20 切分面板

分页面板（JTabbedPane）将一个区域分为很多页，每页可以放置一个组件。使用分页面板时，将组件加入分页面板，再把分页面板加入容器。一个例子如下：

```
// TabbedPaneTest.java  
import java.awt.*;  
import javax.swing.*;  
  
public class TabbedPaneTest extends JFrame {  
    public static void main(String[] args) {  
        JFrame helloFrame = new TabbedPaneTest();  
        helloFrame.setVisible(true);  
    }  
    public TabbedPaneTest() {  
        setSize(300,200);  
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        Container contentPane = getContentPane();  
        contentPane.setLayout(new BorderLayout());  
  
        JLabel label1=new JLabel("Label1",JLabel.CENTER);  
        label1.setBackground(Color.green);  
        label1.setOpaque(true);  
  
        JLabel label2=new JLabel("Label2",JLabel.CENTER);  
        label2.setBackground(Color.blue);  
        label2.setOpaque(true);  
  
        JTabbedPane tabbedPane1=new JTabbedPane(); //创建分页面板  
        tabbedPane1.addTab("tab1",label1); //添加第一页  
        tabbedPane1.addTab("tab2",label2); //添加第二页  
  
        contentPane.add(tabbedPane1);
```

```
    }  
}
```

运行生成的窗口如图：



图 13-21 分页面板

13.8 工具栏

工具栏是窗口中提供最常用功能的快捷按钮栏。工具栏可以贴附在窗口的四条边的任意一条上，也可以独立小窗口存在，通过直接拖动工具栏来改变。

为窗口创建工具栏使用 `JToolBar` 类。快捷按钮是 `JButton` 对象，将它们添加到工具栏对象中，再将工具栏添加到窗口内容容器中，此时内容容器须采用边界布局，工具栏占四个边之一。

使用工具栏的例子如下：

```
// ToolWin.java  
import javax.swing.*;  
import java.awt.*;  
import java.awt.event.*;  
  
public class ToolWin extends JFrame {  
    public static void main(String[] args) {  
        JFrame menuFrame = new ToolWin();  
        menuFrame.setVisible(true);  
    }  
  
    public ToolWin() {  
        setTitle("ToolWin");  
        setSize(300,200);  
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        //创建工具栏  
        Container contentPane = getContentPane();  
        JToolBar bar = new JToolBar("ToolBar");  
        contentPane.add(bar,BorderLayout.NORTH);  
    }  
}
```

```

        bar.add(new JButton("B1")); //普通按钮
        bar.add(new JButton(new ImageIcon("graph.jpg"))); //图标按钮
        bar.addSeparator(); //分隔
        bar.add(new JButton("B3")); //普通按钮
    }
}

```

运行后的窗口如图：



图 13-22 工具栏的效果

13.9 组件边界

有时组件需要一个可见的边界，如包含一组选择按钮的面板。所有 `swing` 组件都可设置边界，用 `setBorder` 方法，参数为边界对象，可以由 `BorderFactory` 的静态方法创建，可以创建多种样式的边界，如凹进、凸起、文字标题等。

边界可以嵌套，比如先创建一个凹进效果的边界，可以再嵌套一个文字标题边界。例子代码如下：

```

// BorderTest.java
import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;

public class BorderTest extends JFrame {
    public static void main(String[] args) {
        JFrame f = new BorderTest();
        f.setVisible(true);
    }
    public BorderTest() {
        setSize(300,200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container contentPane = getContentPane();
        contentPane.add(new radioPanel(),BorderLayout.NORTH);
    }
}

```

```
}

class radioPanel extends JPanel {
    public radioPanel() {
        JRadioButton rb1 = new JRadioButton("S1");
        JRadioButton rb2 = new JRadioButton("S2");
        JRadioButton rb3 = new JRadioButton("S3");
        JRadioButton rb4 = new JRadioButton("S4");
        add(rb1);
        add(rb2);
        add(rb3);
        add(rb4);

        Border etched = BorderFactory.createEtchedBorder();
        Border titled = BorderFactory.createTitledBorder(etched,"Selections");
        setBorder(titled);

        ButtonGroup g1 = new ButtonGroup();
        g1.add(rb1);
        g1.add(rb2);
        ButtonGroup g2 = new ButtonGroup();
        g2.add(rb3);
        g2.add(rb4);
    }
}
```

运行后窗口如图：

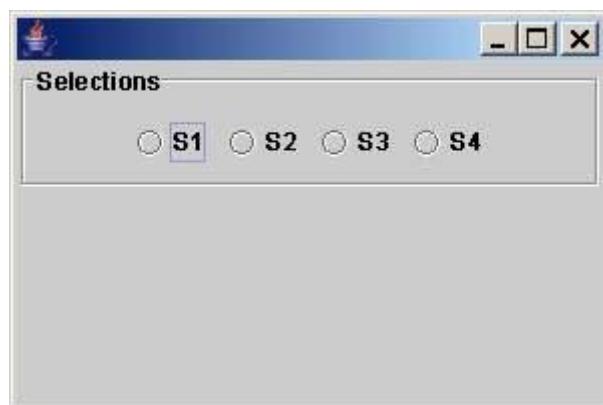


图 13-23 组件边界

13.10 对话框

13.10.1 创建对话框

对话框是一种特殊窗口。它与框架（`JFrame`）的区别主要是，对话框可以依附于某个框架，模式对话框可以强制用户输入信息或阅读信息，只有关闭对话框才能将输入焦点切换到其它子窗口。

编写对话框需要自 `JDialog` 派生一个类，构造函数中必须用 `super` 调用父类构造函数。调用时需给出三个参数，第一个是依附的框架，第二个是对话框的标题，第三个是是否有模式。

`JDialog` 在类库中的继承关系如图：

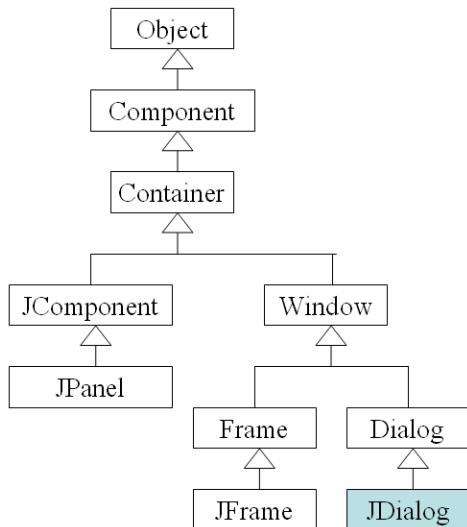


图 13-24 对话框类的继承层次

使用对话框的例子如下：

```
// TestDialog.java
import java.awt.*;
import javax.swing.*;

public class TestDialog extends JDialog {
    public static void main(String[] args) {
        TestDialog testDialog = new TestDialog(null);
        testDialog.setVisible(true);
        System.exit(0);
    }
    public TestDialog(JFrame aOwner) {
        super(aOwner, "About", true);
        setSize(300,200);
        Container contentPane = getContentPane();
    }
}
```

```
    contentPane.add(new JLabel("Test Dialog 1.0"),BorderLayout.CENTER);
    contentPane.add(new JButton("OK"),BorderLayout.SOUTH);
}
}
```

运行后窗口如图：

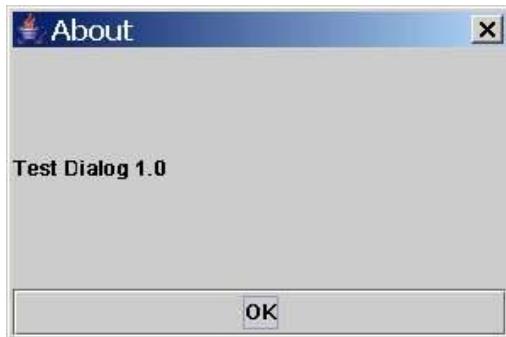


图 13-25 对话框

例子中，在调用父类构造函数时，将依附的框架窗口设为 `null`，表示没有窗口。如果有框架，那么要将其设为第一个参数。

为方便起见，Java 将一些经常会用到的对话框做成了标准对话框，放到了 `swing` 包中。当需要使用时直接拿来用就可以，不用再自己设计。这些标准对话框包括选项对话框、文件选择对话框和颜色选择对话框。

13.10.2 选项对话框

选项对话框（`JOptionPane`）有四个静态方法来显示常用的一些简单的选项对话框：

- 方法 `showMessageDialog`: 显示一条消息并等待用户点击 `OK`。
- 方法 `showConfirmDialog`: 显示一条消息并等待用户选择 `OK/Cancel`。
- 方法 `showOptionDialog`: 显示一条消息并让用户在一组选项中选择。
- 方法 `showInputDialog`: 显示一条消息并让用户输入一行文本。

选项对话框将对话框分为四个区域，如图。每一区域的内容或式样都可以通过参数设置，参数是 `int` 型的，都用事先定义好的常量表示。方法的返回值也是 `int` 型，表示用户按了哪个按钮或选项。

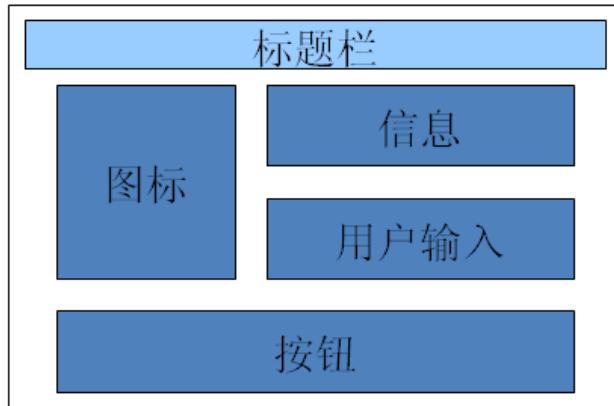


图 13-26 选项对话框的布局

图标部分可选的常量有：

ERROR_MESSAGE
INFORMATION_MESSAGE
WARNING_MESSAGE
QUESTION_MESSAGE
PLAIN_MESSAGE

按钮部分可选的常量有：

DEFAULT_OPTION
YES_NO_OPTION
YES_NO_CANCEL_OPTION
OK_CANCEL_OPTION

返回值部分的常量有：

OK_OPTION
CANCEL_OPTION
YES_OPTION
NO_OPTION
CLOSED_OPTION

使用选项对话框的例子如下：

```

// OptionPaneTest.java
import javax.swing.*;
import java.awt.*;

public class OptionPaneTest extends JFrame {
    public static void main(String[] args) {
        JFrame helloFrame = new OptionPaneTest();
        helloFrame.setVisible(true);
    }

    public OptionPaneTest() {

```

```

setSize(300,200);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
Container contentPane = getContentPane();
contentPane.setLayout(null);
JLabel label1 = new JLabel("操作: ");
label1.setLocation(10,10);
label1.setSize(80,20);
JTextField textField1 = new JTextField();
textField1.setBounds(90,10,200,20);
contentPane.add(label1);
contentPane.add(textField1);

//显示一个 ConfirmDialog
int result = JOptionPane.showConfirmDialog(this,"部分改动内容没有保存，现在
保存？","请确认",JOptionPane.YES_NO_CANCEL_OPTION,JOptionPane.WARNING_MESSAGE);
String s = null;
switch(result) {
    case JOptionPane.YES_OPTION: s = "保存退出";
        break;
    case JOptionPane.NO_OPTION: s = "不保存退出";
        break;
    case JOptionPane.CANCEL_OPTION: s = "不退出";
        break;
}
textField1.setText(s);
}
}

```

运行生成的对话框如图：



图 13-27 选项对话框

在第 4.4.2 节介绍的对话框输入输出方式就是用的此对话框。

13.10.3 文件选择对话框

文件选择对话框（JFileChooser）让用户可以选择磁盘上的文件，既可以用于打开文件，也可以用于保存文件。文件选择对话框是模式对话框。

在使用文件选择对话框时，先创建一个 JFileChooser 对象，然后可以设置当前目录、当前文件、文件过滤器、是否支持多选等属性，再调用 showOpenDialog 或 showSaveDialog 方法就可以显示对话框，方法中的参数是当前窗口。方法返回后，可以通过 JFileChooser 对象的 getSelectedFile 得到选中的文件。例子代码如下：

```
// FileSelector.java
import javax.swing.*;
import java.awt.*;
import java.io.*;
import javax.swing.filechooser.FileFilter;

public class FileSelector extends JFrame {
    public static void main(String[] args) {
        JFrame helloFrame = new FileSelector();
        helloFrame.setVisible(true);
    }

    public FileSelector() {
        setSize(300,200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container contentPane = getContentPane();
        contentPane.setLayout(null);
        JLabel label1 = new JLabel("File:");
        label1.setLocation(10,10);
        label1.setSize(80,20);
        JTextField textField1 = new JTextField();
        textField1.setBounds(90,10,200,20);
        contentPane.add(label1);
        contentPane.add(textField1);

        //显示一个文件选择对话框
        JFileChooser chooser = new JFileChooser(); //创建文件选择对话框
        chooser.setCurrentDirectory(new File(".")); //设置当前目录
        chooser.setSelectedFile(new File("FileSelector.java")); //设置当前文件
        chooser.setFileFilter(new JavaSourceFilter()); //设置文件过滤器
        chooser.setMultiSelectionEnabled(false); //设置是否支持多选
        chooser.showOpenDialog(this); //显示对话框，父窗口为 helloFrame
        String fileName = chooser.getSelectedFile().getPath(); //得到选中的文件
    }
}
```

```

        textField1.setText(fileName);
    }

}

class JavaSourceFilter extends FileFilter {
    public boolean accept(File f) {
        return f.getName().toLowerCase().endsWith(".java") || f.isDirectory();
    }
    public String getDescription() {
        return "Java Source File";
    }
}

```

运行后生成的对话框如图：



图 13-28 文件选择对话框

13.10.4 颜色选择对话框

颜色选择对话框（JColorChooser）让用户可以以多种方式选择一种颜色。颜色选择对话框可以是模式的，也可以是无模式的。还可以将颜色选择器组件添加到一个容器中。

使用颜色选择对话框可以直接调用 JColorChooser 的静态方法 showDialog，该方法有三个参数，第一个是父窗口，第二个是对话框的标题，第三个是当前颜色。例子代码如下：

```

// ColorSelector.java
import javax.swing.*;
import java.awt.*;

public class ColorSelector extends JFrame {
    public static void main(String[] args) {
        JFrame helloFrame = new ColorSelector();
        helloFrame.setVisible(true);
    }
}

```

```

}

public ColorSelector() {
    setSize(300,200);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    Container contentPane = getContentPane();
    contentPane.setLayout(null);
    JLabel label1 = new JLabel("File:");
    label1.setLocation(10,10);
    label1.setSize(80,20);
    JTextField textField1 = new JTextField();
    textField1.setBounds(90,10,200,20);
    contentPane.add(label1);
    contentPane.add(textField1);

    //显示一个有模式颜色选择对话框
    Color selectedColor = JColorChooser.showDialog(this, "选择颜色", null);

    //设置窗口内容容器的背景颜色
    contentPane.setBackground(selectedColor);
}

}

```

运行生成的对话框如下：



图 13-29 颜色选择对话框

第十四章 事件处理

14.1 事件监听机制

图形用户界面（GUI）程序设计必须使用事件处理机制，因为程序的执行不是一条线，而是根据用户操作选择不同代码段来执行。支持 GUI 的操作环境会不断监视用户操作，将用户操作封装成事件，并把事件报告给正在运行的程序。每个程序自己决定如何响应这些事件。

Java 的事件处理采用的是事件监听器方式。涉及的主要概念有三个：事件、事件源和事件监听器。

- 事件：保存了某些事情发生的信息的对象，如用户操作。
- 事件源：能够产生事件的对象，如按钮等组件，当用户点击时，就会产生事件。
- 事件监听器：保存事件处理代码的对象，这些代码保存在对象的一个方法中。

事件处理的过程是：当某件事情发生，如用户按了某个按钮，事件源（即按钮）就会检测到这个操作，并将操作的信息封装在一个事件对象中，然后给对应的事件监听器发送通知，即调用它的某个方法。不同事件源能够产生不同种类的事件。每个事件源拥有自己的事件监听器，一个事件可以有多个监听器响应。

事件监听器是一个实现了监听器接口的对象，监听器接口中定义了事件处理方法，但只有方法定义没有具体实现。编程人员要做的就是编写事件监听器类，在其中实现事件处理方法，然后创建一个事件监听器对象，并添加到相应的事件源。

上一章曾提到，编写 Java 图形界面程序，使用 swing 组件 + awt 事件处理模型。所以，和事件处理有关的类在 awt 和 awt.event 这两个包中。

下面是一个响应按钮事件的例子。事件源是 testButton，按钮按下对应的监听器接口是 ActionListener，该接口只有一个事件处理方法 actionPerformed 表示按钮按下，监听器类是 MyListener1，将监听器添加到事件源用的是 testButton 的 addActionListener 方法。代码如下：

```
// Example1.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class Example1 extends JFrame {
    public Example1() {
        setSize(320,200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container contentPane = getContentPane();
        JButton testButton = new JButton("Test");
        testButton.addActionListener(new MyListener1());
        contentPane.setLayout(new BorderLayout());
```

```

        contentPane.add(testButton,BorderLayout.SOUTH);
    }
}

class MyListener1 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        JOptionPane.showMessageDialog(null,
            "ActionCommand:"+e.getActionCommand());
    }
}

```

事件源和事件监听器之间的关系是很灵活的。一个事件源可以对应多个事件监听器，一个事件监听器也可以对应多个事件源。下面的例子中，有三个 Red、Green、Blue 按钮，也就是有三个事件源；有两个事件监听器，监听器 myListener1 弹出一个显示事件信息的对话框，监听器 myListener2 改变窗口内容容器的背景色。两个事件监听器都添加到了 Red 按钮，而且 myListener2 还添加到了另外两个按钮，如图。

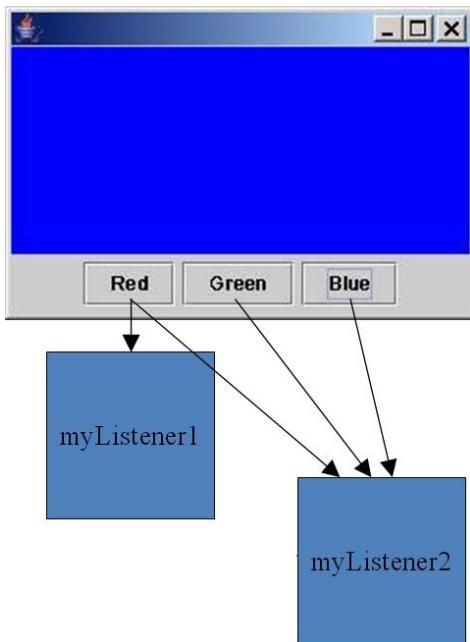


图 14-1 事件源和事件监听器之间的关系

那么，当 Red 按钮按下时，两个监听器会先后响应，即先弹出对话框，再改变背景色，相应的顺序与监听器添加到事件源的顺序正好相反，即后添加的先响应。而另外两个按钮按下时，只有 myListener2 响应，只改变背景色。那 myListener2 对应三个按钮，如何判断是那个按钮触发的事件呢？通过事件对象，即 actionPerformed 方法的参数，是一个 ActionEvent 类型的对象，它的方法 getActionCommand 返回按钮的文本，据此就可以区分事件源了。

例子的代码如下：

```

// Example2.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

```

```

public class Example2 extends JFrame {
    public static void main(String[] args) {
        JFrame win = new Example2();
        win.setVisible(true);
    }
    public Example2() {
        setSize(300,200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container contentPane = getContentPane();
        contentPane.setLayout(new BorderLayout());

        JButton redButton = new JButton("Red");
        JButton greenButton = new JButton("Green");
        JButton blueButton = new JButton("Blue");

        //三个按钮使用同一个监听器对象
        ActionListener myListener2 = new MyListener2();
        redButton.addActionListener(myListener2);
        greenButton.addActionListener(myListener2);
        blueButton.addActionListener(myListener2);

        //red 按钮还有另外一个监听器对象
        redButton.addActionListener(new MyListener1());

        Container buttonPanel = new JPanel();
        buttonPanel.add(redButton);
        buttonPanel.add(greenButton);
        buttonPanel.add(blueButton);
        contentPane.add(buttonPanel, BorderLayout.SOUTH);
    }
    class MyListener2 implements ActionListener { //将监听器声明为内部类
        public void actionPerformed(ActionEvent e) {
            String actionCommand = e.getActionCommand();
            Color bkColor = null;
            if(actionCommand.equals("Red"))bkColor = new Color(255,0,0);
            else if(actionCommand.equals("Green"))bkColor = new Color(0,255,0);
            else if(actionCommand.equals("Blue"))bkColor = Color.blue;
            getContentPane().setBackground(bkColor); //调用外部类的方法
        }
    }
}

class MyListener1 implements ActionListener {

```

```

public void actionPerformed(ActionEvent e) {
    JOptionPane.showMessageDialog(null,"ActionCommand:"+e.getActionCommand());
}
}

```

在这个例子中，第二个事件监听器类 `MyListener2` 定义成了窗口类的内部类。这样做的好处是在 `MyListener2` 中可以访问窗口类的所有成员，此例子中需要通过窗口类得到其内容容器。通常在事件监听器中都需要访问窗口的成员，所以大部分情况下事件监听器类都会写为窗口类的内部类。

通过这两个例子，我们对 Java 的事件监听机制有了个大致了解。可以用下面的示意图来描述 Java 的事件监听机制。

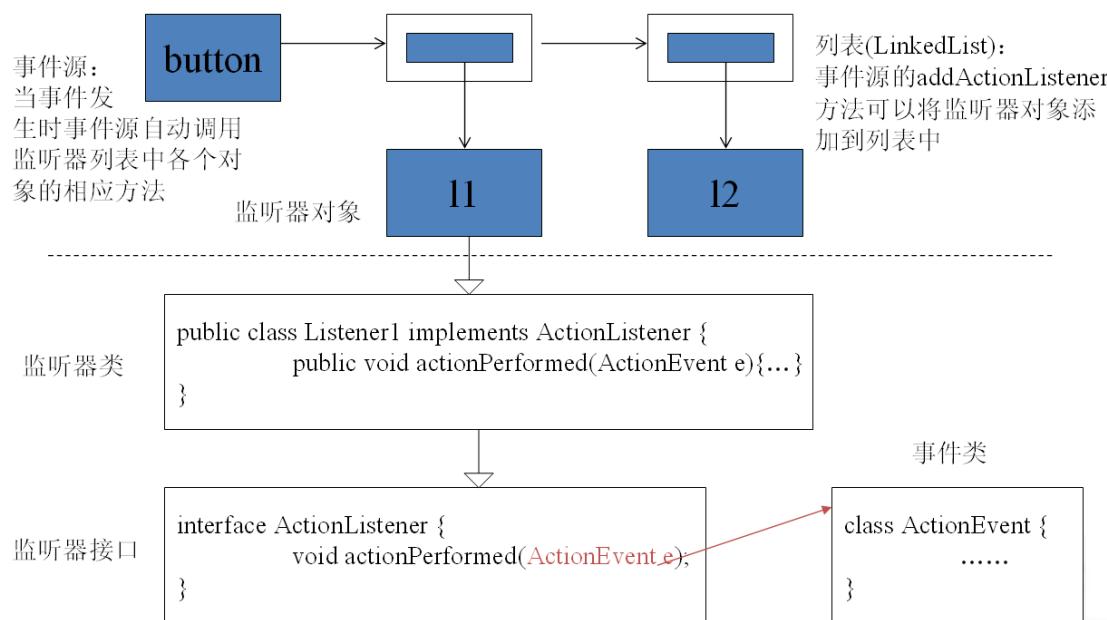


图 14-2 Java 的事件监听器模型

14.2 事件监听器类的各种写法

事件监听器类可以有四种写法：

- 将事件监听器作为单独的类。
- 将事件监听器作为组件的内部类。
- 直接使用已有类(通常是事件源组件)作为事件监听器。
- 使用匿名内部类。

前两个写法在上一节中已经演示过了，本节再介绍后两种写法。

因为只要实现了监听器接口的类就可以作为监听器类，所以可以用一个程序中已有的类来作为监听器类，通常可以用事件源类，因为在事件响应中要处理的数据通常都在事件源对象中，在前面的例子中就是窗口类。只要让窗口类再实现 `ActionListener` 接口，即实现

`actionPerformed` 方法。这种情况下，窗口对象也是监听器对象，在将监听器对象添加到事件源时，应该用语句 `testButton.addActionListener(this)`。因为这条语句在窗口对象的构造函数中，所以 `this` 表示窗口对象。

第四种写法是将监听器类写成匿名内部类。由于事件监听器是特定于事件和事件处理方法的，所以往往整个程序中一个监听器类只用来创建一个监听器对象。一个窗口程序中往往有许多事件需要处理，也就是说需要写很多事件监听器类，如果每个监听器类都不用起名字的话，可以减少命名的工作量。省略掉类名后，直接用类体来声明对象，这样带来的另一个好处是在添加事件处能直接找到事件处理代码，不用再通过类名去找了。

将事件监听器类写成匿名内部类的例子如下：

```
// Example3.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Example3 extends JFrame {
    public static void main(String[] args) {
        JFrame win = new Example3();
        win.setVisible(true);
    }
    public Example3() {
        setSize(300,200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container contentPane = getContentPane();
        contentPane.setLayout(new BorderLayout());

        Container buttonPanel = new JPanel();
        JButton redButton = makeButton(buttonPanel,"Red",Color.red);
        makeButton(buttonPanel,"Green",Color.green);
        makeButton(buttonPanel,"Blue",Color.blue);
        //三个按钮各自有一个事件监听器对象，三个对象同属于一个事件监听器类

        redButton.addActionListener(new ActionListener() { //匿名内部类监听器，即
Example2 中的 MyListener1
            public void actionPerformed(ActionEvent e) {
                JOptionPane.showMessageDialog(null,"ActionCommand:"+e.getActionCommand());
            }
        });

        contentPane.add(buttonPanel,BorderLayout.SOUTH);
    }

    //本方法创建一个按钮，该按钮带有单独的事件监听器对象
```

```

private JButton makeButton(Container aContainer, String aName, final Color aBackColor)
{
    JButton tButton = new JButton(aName);
    aContainer.add(tButton);
    tButton.addActionListener(new ActionListener() {
        //将 Example2 中的 MyListener2 改造为匿名内部类。
        public void actionPerformed(ActionEvent e) {
            getContentPane().setBackground(aBackColor);
        }
    });
    return tButton;
}

```

因为匿名内部类是最简洁的写法，所以是目前最流行的。

监听器类和监听器对象之间的关系也是很灵活的。一个监听器类可以创建一个监听器对象，同时为多个同类型的事件源服务（像例子 2 那样），此时通过事件处理方法的参数——即事件对象来区分不同事件源。一个监听器类也可以创建多个监听器对象，每个监听器对象为一个事件源服务，此时可以通过构造函数为不同的监听器对象赋予不同的属性，来进行区分。

14.3 窗口事件

前面介绍的 `ActionListener` 接口非常简单，用于按钮事件处理。相应的，窗口事件比较复杂，对应的接口是 `WindowListener`。`WindowListener` 定义的方法及其对应的操作如下：

- `void windowActivated(WindowEvent e)` : 窗口被激活为当前窗口
- `void windowClosed(WindowEvent e)` : 窗口关闭
- `void windowClosing(WindowEvent e)` : 窗口将要被关闭
- `void windowDeactivated(WindowEvent e)` : 窗口变为不激活
- `void windowDeiconified(WindowEvent e)` : 窗口从最小化还原
- `void windowIconified(WindowEvent e)` : 窗口最小化
- `void windowOpened(WindowEvent e)` : 窗口第一次打开

要编写窗口事件监听器类，需要实现此接口，也就是要给出所有这些方法的实现。如果只想响应其中的某个操作，其它的方法也不能省略，可以写成空方法。

下面是一个响应窗口事件的例子，响应的是 `windowClosing` 操作，弹出一个窗口询问用户是否真的要关闭，如果用户选择“是”，那么关闭窗口退出程序；如果用户选择“否”，那么就不关闭。退出程序用的是 `System` 类的静态方法 `exit`，一般用参数 0 表示正常退出。例子代码如下：

```

// Example5.java
import javax.swing.*;
import java.awt.*;

```

```

import java.awt.event.*;

public class Example5 extends JFrame {
    public static void main(String[] args) {
        JFrame win = new Example5();
        win.setVisible(true);
    }
    public Example5() {
        setSize(320,200);
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
        Container contentPane = getContentPane();
        JButton testButton = new JButton("Test");
        contentPane.setLayout(new BorderLayout());
        contentPane.add(testButton,BorderLayout.SOUTH);
        addWindowListener(new MyWindowListener());
    }
    private class MyWindowListener implements WindowListener {
        public void windowOpened(WindowEvent e){}
        public void windowClosing(WindowEvent e) {
            int result = JOptionPane.showConfirmDialog(
                Example5.this,"确定要退出程序吗？","退出程序",
                JOptionPane.OK_CANCEL_OPTION,
                JOptionPane.WARNING_MESSAGE);
            if(result == JOptionPane.OK_OPTION) {
                System.exit(0);
            }
        }
        public void windowClosed(WindowEvent e){}
        public void windowIconified(WindowEvent e){}
        public void windowDeiconified(WindowEvent e){}
        public void windowActivated(WindowEvent e){}
        public void windowDeactivated(WindowEvent e){}
    }
}

```

这个例子的代码中有个问题，就是仅仅需要响应一个操作，但却要写另外六个空方法。有没有方法可以让另外六个方法不用写呢？有，Java 提供了适配器类。窗口事件对应的适配器类是 `WindowAdapter`，它实现了 `WindowListener` 接口，但是里面所有方法都是空方法。写窗口事件监听器的时候，可以从 `WindowAdapter` 派生，这样就自动继承所有方法，然后对需要的事件处理方法进行覆盖即可，就不用写其它方法了。

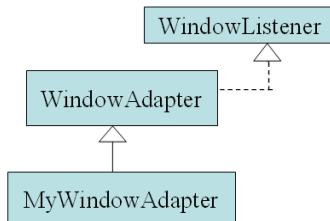


图 14-3 适配器类的作用

使用适配器类后，上面例子的代码可以改为：

```

// Example6.java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Example6 extends JFrame {
    public static void main(String[] args) {
        JFrame win = new Example6();
        win.setVisible(true);
    }
    public Example6() {
        setSize(320,200);
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
        Container contentPane = getContentPane();
        JButton testButton = new JButton("Test");
        contentPane.setLayout(new BorderLayout());
        contentPane.add(testButton, BorderLayout.SOUTH);
        addWindowListener(new MyWindowListener());
        /* 也可以使用匿名内部类，写法如下
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                int result = JOptionPane.showConfirmDialog(Example6.this,"确定要退出
程序吗？","退出程序",JOptionPane.OK_CANCEL_OPTION,JOptionPane.WARNING_MESSAGE);
                if(result == JOptionPane.OK_OPTION) {
                    System.exit(0);
                }
            }
        });
        */
    }
    private class MyWindowListener extends WindowAdapter {
        public void windowClosing(WindowEvent e) {
            int result = JOptionPane.showConfirmDialog(
                Example6.this,"确定要退出程序吗？","退出程序",
                JOptionPane.OK_CANCEL_OPTION,

```

```

        JOptionPane.WARNING_MESSAGE);
        if(result == JOptionPane.OK_OPTION) {
            System.exit(0);
        }
    }
}

```

Java 中每个具有不止一个方法的监听器接口都对应有一个适配器类。采用适配器类后，事件监听器类也可以写成匿名内部类，参见上面的例子。

14.4 事件类

Java 中所有事件类都从 `java.util.EventObject` 类继承而来。事件类定义在两个包中：

- `java.awt.event`: awt 的事件、监听器及适配器定义。
- `java.swing.event`: 专门用于 swing 组件的附加事件。

要查找组件能够产生哪些事件，可以查看组件类的帮助文档，查找组件能够添加哪些事件监听器(`add***Listener`)。根据该方法的参数查找相应的接口，即可知道具体的事件含义。

`java.awt.event` 包中定义了 11 个监听器接口，分别是：

- `ActionListener`
- `WindowListener`
- `AdjustmentListener`
- `ComponentListener`
- `ContainerListener`
- `FocusListener`
- `ItemListener`
- `KeyListener`
- `MouseListener`
- `MouseMotionListener`
- `TextListener`

其中有 7 个接口有对应的适配器类，分别是：

- `WindowAdapter`
- `ComponenAdapter`
- `ContainerAdapter`
- `FocusAdapter`
- `KeyAdapter`
- `MouseAdapter`
- `MouseMotionAdapter`

虽然接口和类比较多，但使用方法都一样。这些事件可以分为语义事件和低级事件。语义事

件是有明确意义的事件，包括：

- **ActionEvent**: 按钮按下、菜单选择、选择列表项、文本域中按回车
- **AdjustmentEvent**: 调整滚动条
- **ItemEvent**: 从一组选择框或列表项中选择一个
- **TextEvent**: 文本域或文本框中内容发生变化

低级事件是较小的基本事件，包括：

- **ComponentEvent**: 组件被显示、隐藏、改变位置、改变大小
- **KeyEvent**: 键盘上的一个键被按下或者释放
- **MouseEvent**: 鼠标按键的按下和释放，鼠标移动或拖动
- **FocusEvent**: 组件得到焦点或失去焦点
- **WindowEvent**: 窗口被显示、隐藏、关闭、激活、图标化、还原
- **ContainerEvent**: 容器中加入或移除一个组件

14.5 键盘事件

键盘事件描述键盘上的按键的按下或弹起。一个组件要产生键盘事件，必须获得输入焦点。不同组件获得输入焦点的外在表现不同，如文本区中出现光标、按钮上有虚线框等。组件可以用鼠标点击获得输入焦点，也可以用 **Tab** 键使输入焦点在窗口中所有组件间循环。

键盘事件监听器接口是 **KeyListener**，有三个方法：

- **void keyPressed(KeyEvent e)**: 按键按下时调用。
- **void keyReleased(KeyEvent e)**: 按键弹起时调用。
- **void keyTyped(KeyEvent e)**: 结合上述两个操作，在按键敲击时调用。

三个方法的参数——键盘事件类 **KeyEvent** 的主要方法有：

- **char getKeyChar()**: 得到按键对应的字符。
- **int getKeyCode()**: 得到按键对应的扫描码。
- **static String getKeyText(int keyCode)**: 将扫描码转化为说明字符串。

通过 **KeyEvent** 的这些方法，就可以获得按下或弹起的是哪个键。扫描码是每个键盘上的按键都有的一个整数编号，为了区分键盘上的按键。在 **KeyEvent** 类中用静态常量定义了所有按键的扫描码，如：**VK_A**、**VK_SHIFT**、**VK_F10**、**VK_ENTER**、**VK_LEFT**、**VK_NUMPAD1**。判断 **SHIFT**、**CONTROL**、**ALT** 等组合键的状态可以使用 **KeyEvent** 的 **isShiftDown**, **isControlDown**, **isAltDown** 方法。

下面是一个处理键盘事件的例子。窗口中有一个按钮和一个文本区，键盘事件监听器添加到了按钮上，如图。当输入焦点在按钮上时，按下或弹起某个键会触发键盘事件，并调用事件监听器。如果输入焦点在文本区中，则不会触发按钮的键盘事件。事件监听器中将按键的信息显示在文本区中。在 **keyPressed** 方法中还做了判断，如果按下 **Alt+F3**，就退出程序。

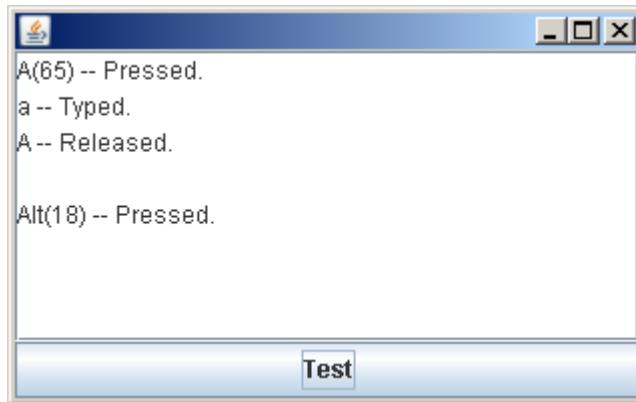


图 14-4 键盘事件

例子的源代码如下：

```
// KeyExample.java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class KeyExample extends JFrame {
    private JTextArea keyTextArea = new JTextArea();
    public static void main(String[] args) {
        JFrame win = new KeyExample();
        win.setVisible(true);
    }
    public KeyExample() {
        setSize(320,200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container contentPane = getContentPane();
        contentPane.setLayout(new BorderLayout());
        JButton testButton = new JButton("Test");
        testButton.addKeyListener(new KeyListener() { //键盘监听器，添加到 Test 按钮；所以，只有输入焦点在 Test 上时才起作用
            public void keyPressed(KeyEvent e) {
                int keyCode = e.getKeyCode(); //得到按键扫描码
                keyTextArea.append(KeyEvent.getKeyText(keyCode)) //将按键扫描码转化为按键描述字符串
                    + "(" + keyCode + ")" + " -- Pressed.\n";
                if(keyCode == KeyEvent.VK_F3 && e.isAltDown()) { //按下 Alt+F3，程序退出
                    System.exit(0);
                }
            }
            public void keyTyped(KeyEvent e) {
                keyTextArea.append(e.getKeyChar()+" -- Typed.\n");
            }
        });
    }
}
```

```

        }
        public void keyReleased(KeyEvent e) {
            keyTextArea.append(KeyEvent.getKeyText(e.getKeyCode())+""
--Released.\n\n");
        }
    });
    contentPane.add(testButton,BorderLayout.SOUTH);
    contentPane.add(new JScrollPane(keyTextArea),BorderLayout.CENTER);
}
}

```

键盘事件只在事件源组件获得输入焦点时触发。但有些组件缺省情况下没有输入焦点，如 JPanel。此时若要让组件能产生键盘事件，应该让组件可以得到输入焦点，用组件的 setFocusable 方法，参数为 true 就是能得到输入焦点。注意，窗口中不能同时出现其他能够获得输入焦点的组件，否则输入焦点还是无法转移到 JPanel 上。

例子代码如下：

```

// KeyExample1.java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class KeyExample1 extends JFrame {
    private JLabel keyLabel = new JLabel();
    public static void main(String[] args) {
        JFrame win = new KeyExample1();
        win.setVisible(true);
    }
    public KeyExample1() {
        setSize(320,200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container contentPane = getContentPane();
        contentPane.setLayout(new BorderLayout());
        JPanel testPanel = new JPanel();
        testPanel.setFocusable(true); //若缺少此句，则不能接收键盘事件
        testPanel.setBackground(Color.blue);
        testPanel.setPreferredSize(new Dimension(50,50));
        testPanel.addKeyListener(new KeyListener() {
            public void keyPressed(KeyEvent e) {
                int keyCode = e.getKeyCode(); //得到按键扫描码
                keyLabel.setText(KeyEvent.getKeyText(keyCode)) //将按键扫描码转化为
按键描述字符串
                + "(" + keyCode + ")" + " -- Pressed.\n");
                if(keyCode == KeyEvent.VK_F3 && e.isAltDown()) { //按下 Alt+F3，程

```

序退出

```
        System.exit(0);
    }
}
public void keyTyped(KeyEvent e) {}
public void keyReleased(KeyEvent e) {}
});
contentPane.add(testPanel,BorderLayout.SOUTH);
contentPane.add(keyLabel,BorderLayout.CENTER); //不能有其他能获得焦点的组件，所以选择使用 JLabel，而不使用 JTextArea。
//contentPane.add(new JTextField(), BorderLayout.NORTH); // 若添加此句，则 JPanel 无法获得输入焦点
}
}
```

有些情况下还需要“销毁”键盘事件。比如，设计一个输入电话号码在文本域，只允许输入数字，输入其它字符是无效的。销毁键盘事件要在监听器的 `keyTyped` 方法中，使用 `KeyEvent` 事件对象的 `consume` 方法。例子代码如下：

```
// KeyExample2.java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class KeyExample2 extends JFrame {
    public static void main(String[] args) {
        JFrame win = new KeyExample2();
        win.setVisible(true);
    }
    public KeyExample2() {
        setSize(320,200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container contentPane = getContentPane();
        contentPane.setLayout(new BorderLayout());
        JTextField numField = new JTextField();
        numField.addKeyListener(new KeyAdapter() {
            public void keyTyped(KeyEvent e) {
                char ch = e.getKeyChar();
                if(ch < '0' || ch > '9') e.consume();
            }
        });
        contentPane.add(numField,BorderLayout.NORTH);
    }
}
```

14.6 鼠标事件

14.6.1 鼠标事件

鼠标事件描述鼠标按键的按下和释放，以及当时的鼠标位置。鼠标的移动有专门的鼠标移动事件，将在本节的后半部分介绍。

鼠标事件监听器接口是 `MouseListener`，有五个方法：

- `void mousePressed(MouseEvent e)`: 鼠标按键按下时调用。
- `void mouseReleased(MouseEvent e)` : 鼠标按键释放时调用。
- `void mouseClicked(MouseEvent e)` : 结合上述两个方法，鼠标按键点击时调用。
- `void mouseEntered(MouseEvent e)` : 鼠标移动进入事件源组件时调用。
- `void mouseExited(MouseEvent e)` : 鼠标移出事件源组件时调用。

上面五个方法的参数——鼠标事件类 `MouseEvent` 的主要方法有：

- `int getButton()`: 得到发生动作的按键(BUTTON1—左, BUTTON2—中, BUTTON3—右)。
- `int getClickCount()`: 得到点击次数，主要用于双击事件。
- `Point getPoint()`(或 `int getX(), int getY()`): 得到事件发生时鼠标的相对位置。
- `int getModifiers ()`: 得到鼠标按键和键盘组合键的状态。

在鼠标事件中判断键盘组合键 `SHIFT`、`CONTROL`、`ALT` 的状态可以使用 `MouseEvent` 的 `isShiftDown`, `isControlDown`, `isAltDown` 方法。

下面的例子演示鼠标事件的处理。窗口中有一个按钮和一个文本区，鼠标事件监听器添加到了按钮上，如图。当按钮发生鼠标事件时，监听器会将事件信息显示在文本区中。鼠标事件包括：在按钮上按下或弹起鼠标按键，以及鼠标进入或移出按钮区域。在 `mousePressed` 方法中还做了判断，如果按住 `Shift` 键，并在 `testButton` 左边 50 像素之内双击鼠标左键，则退出程序。

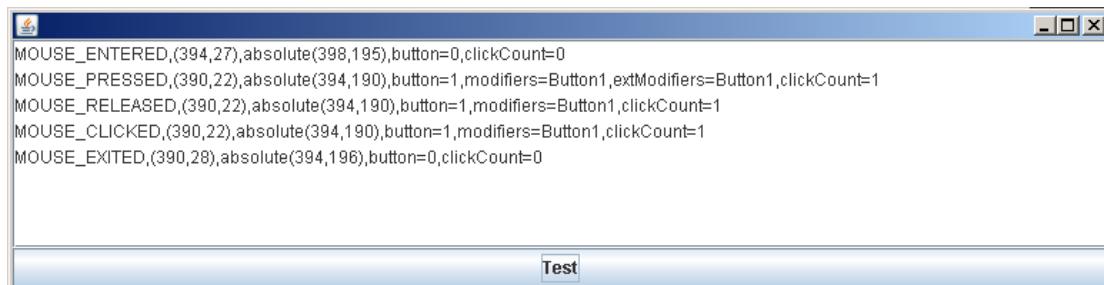


图 14-5 鼠标事件

例子的代码如下：

```
// MouseExample.java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
```

```

public class MouseExample extends JFrame {
    private JTextArea mouseTextArea = new JTextArea();
    public static void main(String[] args) {
        JFrame win = new MouseExample();
        win.setVisible(true);
    }
    public MouseExample() {
        setSize(320,200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container contentPane = getContentPane();
        contentPane.setLayout(new BorderLayout());
        JButton testButton = new JButton("Test");
        // testButton.setCursor(Cursor.getPredefinedCursor(Cursor.HAND_CURSOR));
        testButton.addMouseListener(new MouseListener() { //鼠标事件监听器，添加到
Test 按钮
            public void mousePressed(MouseEvent e) {
                mouseTextArea.append(e paramString()+"\n");
            }
            public void mouseClicked(MouseEvent e) {
                mouseTextArea.append(e paramString()+"\n");
                if(e.getButton()==MouseEvent.BUTTON1 && //鼠标左键
                    e.getClickCount()==2 && //双击
                    e.getX()<50 && //在 testButton 左边 50 像素
                    e.isShiftDown()) { //按下键盘 Shift 键
                    System.exit(0);
                }
            }
            public void mouseReleased(MouseEvent e) {
                mouseTextArea.append(e paramString()+"\n");
            }
            public void mouseEntered(MouseEvent e) {
                mouseTextArea.append(e paramString()+"\n");
                setCursor(Cursor.getPredefinedCursor(Cursor.HAND_CURSOR));
            }
            public void mouseExited(MouseEvent e) { //鼠标移出 testButton
事件
                mouseTextArea.append(e paramString()+"\n");
                int modifiers = e.getModifiers();
                int exitMask = MouseEvent.BUTTON3_MASK | //鼠标右键
                    MouseEvent.CTRL_MASK | //按下键盘 Ctrl 键
                    MouseEvent.BUTTON1_MASK; //鼠标左键
                if((modifiers & exitMask)==exitMask) {
                    System.exit(0);
                }
            }
        });
    }
}

```

```

        }
    //      setCursor(Cursor.getPredefinedCursor(Cursor.DEFAULT_CURSOR));
    }
});

contentPane.add(testButton,BorderLayout.SOUTH);
contentPane.add(new JScrollPane(mouseTextArea),BorderLayout.CENTER);
}
}

```

有时需要判断多个鼠标按键的状态，比如左键和右键同时按住，可以使用 `MouseEvent` 的 `getModifiers` 得到鼠标按键的状态。该方法返回一个整数，转换成二进制后，用一个二进制位表示一个按键的状态，1 表示按下，0 表示未按下。例如，左键 `BUTTON1` 对应右起第 1 位，中键 `BUTTON2` 对应右起第 2 位，右键 `BUTTON3` 对应右起第 3 位。同时，键盘上的组合键也有对应的位，如 `Ctrl` 对应右起第 4 位。这样，通过判断各个状态位，就可以判断是否满足需要的条件。如图。

modifiers	... 1 0 0 1 1 0 1
exitMask	... 0 0 0 1 1 0 1
BUTTON1	... 0 0 0 0 0 0 1
BUTTON3	... 0 0 0 0 1 0 0
CTRL	... 0 0 0 1 0 0 0

图 14-6 判断鼠标按键状态

上面的例子还判断下面的条件：按住鼠标左键和右键，按住键盘 `Ctrl` 键，将鼠标从 `testButton` 移出，则退出程序。因为事件由鼠标移出触发，所以对应于 `mouseExited` 方法。代码见上面例子中的 `mouseExited` 方法。

14.6.2 鼠标形状

鼠标的形状也可以改变。比如，要让前面的例子中鼠标移动到 `testButton` 上时变为手的形状，可以使用组件的 `setCursor` 方法，参数是一个 `Cursor` 对象描述鼠标形状。鼠标有很多事先定义好的形状，如手形、十字等，可以通过 `Cursor` 类的静态方法 `getPredefinedCursor` 生成 `Cursor` 对象，方法的参数是 `Cursor` 类中的静态常量。语句如下：

```
testButton.setCursor(Cursor.getPredefinedCursor(Cursor.HAND_CURSOR));
```

也可以在事件监听器的 `mouseEntered` 和 `mouseExited` 方法中设置窗口的鼠标图标。代码如下（完整代码见上一节的例子）：

```

public void mouseEntered(MouseEvent e) {
    setCursor(Cursor.getPredefinedCursor(Cursor.HAND_CURSOR));
}

public void mouseExited(MouseEvent e) {

```

```
    setCursor(Cursor.getPredefinedCursor(Cursor.DEFAULT_CURSOR));
}
```

预定义的鼠标形状见 `Cursor` 类的文档。

14.6.3 鼠标移动事件

鼠标移动事件是独立于鼠标事件的，专门描述鼠标的移动和拖动，对应的事件监听器接口 `MouseMotionListener` 有两个方法：

- `void mouseDragged(MouseEvent e)`: 鼠标拖动时调用，即鼠标在组件上按下，然后移动位置，可以移动出组件之外。
- `void mouseMoved(MouseEvent e)`: 鼠标移动时调用，即没有按钮按下，且鼠标只在组件内部移动位置。

上面两个方法的参数与 `MouseListener` 相同，也是 `MouseEvent` 类型。`mouseDragged` 方法通常用于组件内部的某些物体形状、位置等改变。`mouseMoved` 方法通常用于组件内部鼠标光标的改变。在鼠标移动事件监听器中，也是使用 `MouseEvent` 的各种方法来获得鼠标的当前状态。

下面是一个处理鼠标移动事件的例子。在窗口底部显示鼠标移动事件信息，在窗口顶部显示鼠标拖动事件信息。窗口中放了两个 `JPanel` 组件，左边的设置为蓝色，显示的鼠标事件的事件源是右侧的 `JPanel`，这样就可以测试从一个组件移动或拖动出组件范围时的情况。



图 14-7 鼠标移动事件和拖动事件

例子代码如下：

```
// MouseMoveExample.java
```

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class MouseMoveExample extends JFrame {
    private JLabel mouseMovedInfo = new JLabel("mouseMoved:");
    private JLabel mouseDraggedInfo = new JLabel("mouseDragged:");
    public static void main(String[] args) {
        JFrame win = new MouseMoveExample();
        win.setVisible(true);
    }
    public MouseMoveExample() {
        setSize(600,400);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container contentPane = getContentPane();
        contentPane.setLayout(new BorderLayout());
        JPanel testPanel = new JPanel();
        contentPane.add(mouseMovedInfo,BorderLayout.SOUTH);
        contentPane.add(mouseDraggedInfo,BorderLayout.NORTH);
        JPanel leftPanel = new JPanel();
        leftPanel.setPreferredSize(new Dimension(50,100));
        leftPanel.setBackground(Color.blue);
        contentPane.add(leftPanel,BorderLayout.WEST);
        contentPane.add(testPanel,BorderLayout.CENTER);
        testPanel.addMouseListener(new MouseAdapter() {
            public void mouseMoved(MouseEvent e) {
                mouseMovedInfo.setText(e paramString());
            }
            public void mouseDragged(MouseEvent e) {
                mouseDraggedInfo.setText(e paramString());
            }
        });
        testPanel.addMouseListener(new MouseAdapter() {
            public void mouseReleased(MouseEvent e) {
                mouseDraggedInfo.setText(e paramString());
            }
        });
    }
}

```

14.7 定时器事件

定时器是一个事件源对象，可以实现每隔一段时间触发一个事件。Java 类库中有两个定时器

类，`javax.swing.Timer` 和 `java.util.Timer`，注意类名的区别。本节介绍的是 `javax.swing.Timer` 类，使用起来比较简单，而且对 swing 组件安全，没有多线程冲突问题。

`Timer` 类的主要方法有：

- `Timer(int delay, ActionListener listener)`: 构造函数
- `void start()`: 启动定时器
- `void stop()`: 停止定时器

使用定时器时，先创建定时器对象，然后使用其 `start` 和 `stop` 方法控制定时器的启动和停止。构造函数中，第一个参数为时间间隔，单位为毫秒；第二个参数为事件监听器对象，实现的接口为 `ActionListener`，同按钮的一样。

下面是一个定时器的例子，每隔一秒钟触发一次，改变一下窗口中文本标签的数字，看上去就像一个时钟。如图：

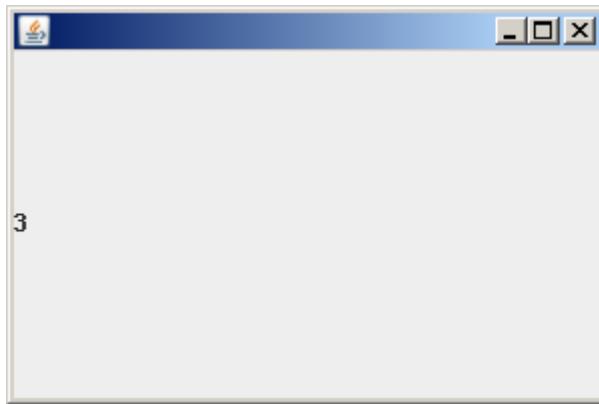


图 14-8 定时器事件

例子代码如下：

```
// TimerExample.java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class TimerExample extends JFrame {
    private JLabel infoLabel = new JLabel("Time:");
    private int counter;
    public static void main(String[] args) {
        JFrame win = new TimerExample();
        win.setVisible(true);
    }
    public TimerExample() {
        setSize(300,200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container contentPane = getContentPane();
        contentPane.setLayout(new BorderLayout());
    }
}
```

```

        contentPane.add(infoLabel);
        Timer t = new Timer(1000,new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                infoLabel.setText(String.valueOf(counter++));
            }
        });
        t.start();
    }
}

```

第十五章 绘图

15.1 绘图机制

Java 中允许在任意 Swing 组件上绘制图形，但一般绘制在面板(JPanel)上，因为面板是空白的。在面板上绘制图形需要三步：

- 自 JPanel 派生一个新类；
- 覆盖 paintComponent 方法，将绘图的语句写在这个方法中；
- 用新类创建一个对象，作为组件添加到要显示的容器中。

下面是一个绘图的例子。在窗口中放了一个 JPanel 组件，在 JPanel 上画了一个矩形。如图：

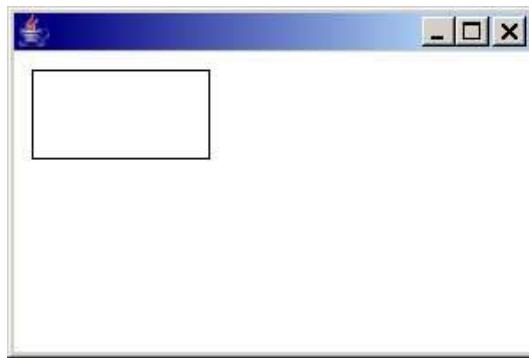


图 15-1 在面板上绘图

例子的代码如下：

```

// PanelExample.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class PanelExample extends JFrame {
    public static void main(String[] args) {
        JFrame win = new PanelExample();
        win.setVisible(true);
    }
}

```

```

public PanelExample() {
    setSize(300,200);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    Container contentPane = getContentPane();
    contentPane.add(new MyPanel());
}
}

class MyPanel extends JPanel {
    public void paintComponent(Graphics g) {
        super.paintComponent(g); // 
        g.drawRect(10,10,100,50);
    }
}

```

在覆盖 `paintComponent` 方法时，应首先调用父类的 `paintComponent`，以便画出组件原来的样子。绘图时，使用 `paintComponent` 方法的参数——`Graphics` 对象，称为绘图对象或画布，所有绘图操作都要通过绘图对象的方法进行。`Graphics` 提供了以下几类绘图方法：

- 绘制简单几何图形，如矩形、椭圆等；
- 绘制图像，如图片；
- 绘制文字；
- 设置画笔属性，如颜色、文字字体、绘图模式等。

这些方法如何使用将在本章后面的章节中介绍。

Java 绘制窗口的机制是依次调用每个组件的 `paintComponent` 方法，每个组件都在自己的 `paintCompoient` 方法中绘制出自己。具体来说，第一步先创建出最外层窗口，这个是和操作系统平台相关的。第二步调用窗口的布局管理器计算各组件的大小和位置。第三步依次调用每个组件的 `paintComponent` 方法绘出组件。如果窗口中有其它容器，对于容器也会重复布局和绘制的过程。这一绘制过程在需要时会被调用，包括以下情况：

- 面板首次显示时；
- 面板尺寸变化时；
- 其它窗口遮住面板时；
- 组件的 `repaint()`方法被调用时。

`paintComponent` 方法是先确定了在哪里调用，再在开发组件的时候给出实现，所以这种方法又称为回调方法。类似的情况还有事件监听器的方法，也属于回调。也可以将绘制看作是发给组件的一个事件，`paintComponent` 是其处理方法。

绘图坐标系以组件左上角为坐标原点，向右为 X 轴，向下为 Y 轴，单位为点。

15.2 绘制几何图形

`Graphics` 对象能够绘制的几何图形包括直线、矩形、圆角矩形、椭圆、弧线和多边形。

绘制直线的方法是 `drawLine`, 定义如下:

```
void drawLine(int x1, int y1, int x2, int y2)
```

参数 `x1, y1` 和 `x2, y2` 分别是两个端点。例如, `g.drawLine(10,20,60,50)` 绘出的直线如图。

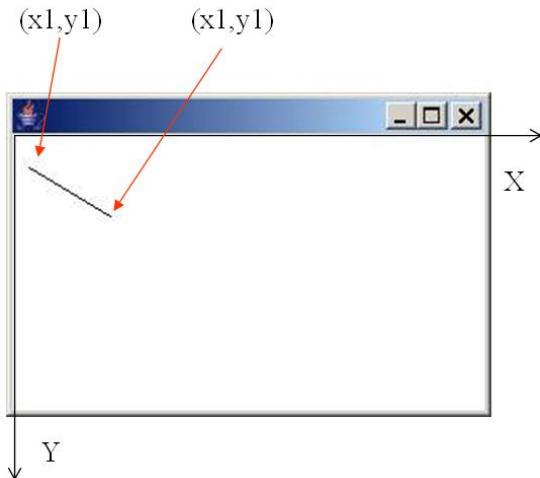


图 15-2 绘制直线

绘制矩形的方法是 `drawRect`, 定义如下:

```
void drawRect(int x, int y, int width, int height)
```

参数 `x, y` 给出矩形左上角的坐标, `width` 和 `height` 给出矩形的宽和高。`drawRect` 只画出矩形的边框, 还有一个填充矩形的方法是 `fillRect`, 定义如下:

```
void fillRect(int x, int y, int width, int height)
```

例如, 下面的语句绘出的图形如图:

```
g.drawRect(10,20,60,50);
```

```
g.fillRect(80,20,60,50);
```

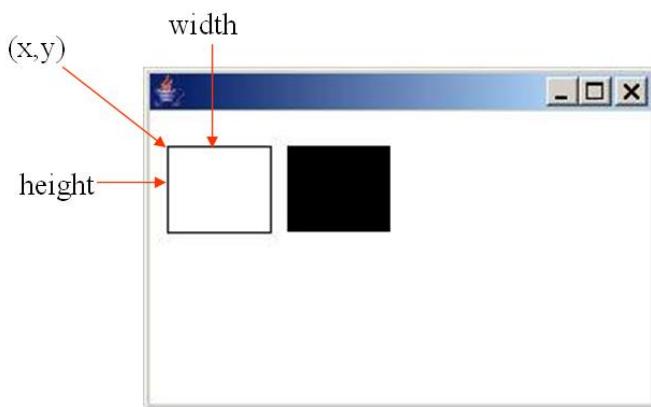


图 15-3 绘制矩形

绘制和填充圆角矩形的方法是 `drawRoundRect` 和 `fillRoundRect`, 定义如下:

```
void drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)
```

```
void fillRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)
```

参数 `x, y, width, height` 给出矩形的位置和大小, `arcWidth` 和 `arcHeight` 给出圆角的大小。这里将圆角看作椭圆的四分之一, 如图。

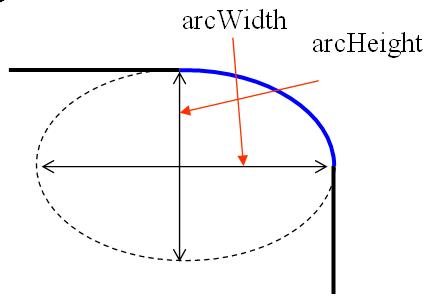


图 15-4 圆角是椭圆的四分之一

下面的语句绘出的图形如图

```
g.drawRoundRect(10,10,100,50,20,15);  
g.fillRoundRect(160,10,100,50,20,15);
```

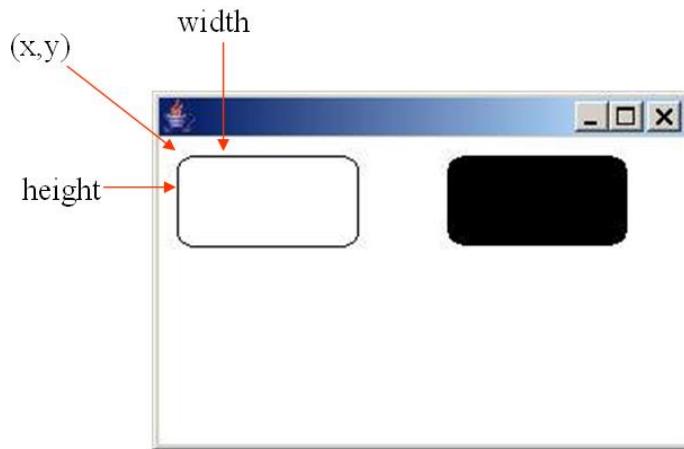


图 15-5 绘制圆角矩形

绘制椭圆和填充椭圆的方法是 `drawOval` 和 `fillOval`, 定义如下:

```
void drawOval(int x, int y, int width, int height)  
void fillOval(int x, int y, int width, int height)
```

参数给出的是椭圆的外切矩形。

下面的代码绘出的图形如图

```
g.drawOval(10,10,100,50);  
g.fillOval(160,10,100,50);
```

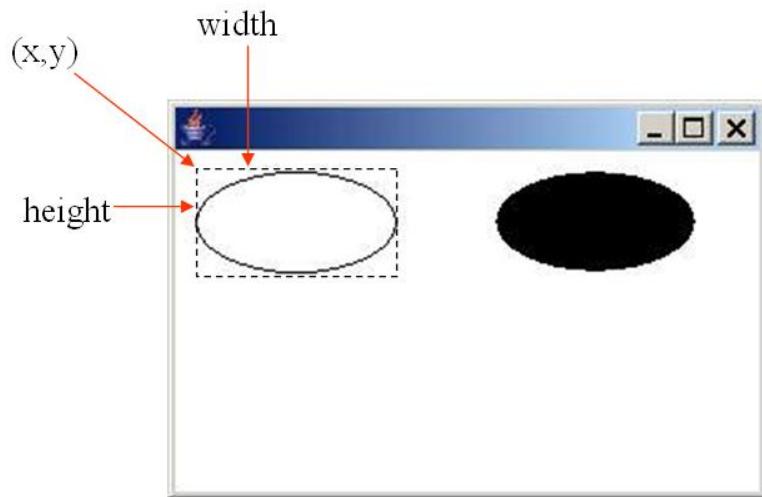


图 15-6 绘制椭圆

绘制弧线和填充扇形的方法是 `drawArc` 和 `fillArc`, 定义如下:

```
void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)
void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)
```

弧线是椭圆的一个弧段, 所以, 参数 `x`, `y`, `width`, `height` 给出椭圆外切矩形的位置和大小, 也就决定了椭圆的位置和大小; `startAngle` 和 `arcAngle` 给出弧段开始的角度和转过的度数, 是按逆时针角度旋转的, 以 X 轴为 0 度, 度数单位为 360 度一圈。

下面的代码绘出的图形如图

```
g.drawArc(10,10,100,50,0,60);
g.fillArc(160,10,100,50,0,60);
```

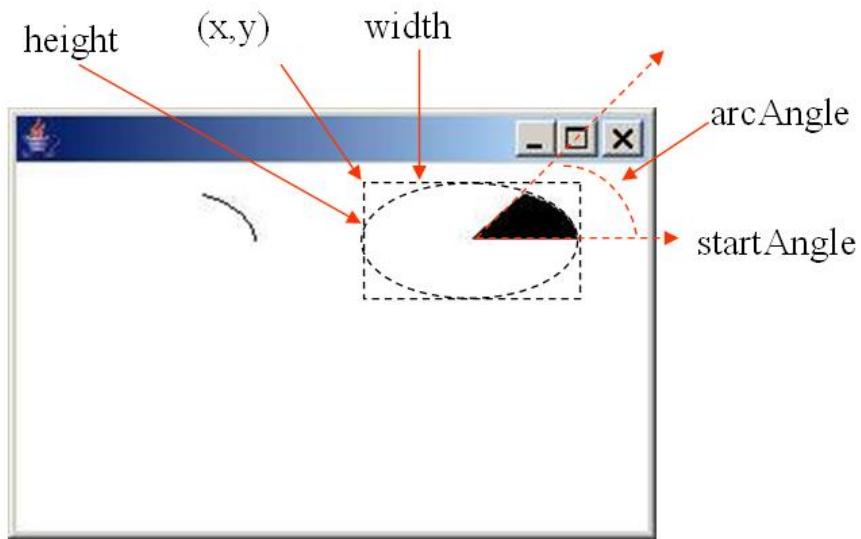


图 15-7 绘制圆弧

绘制和填充多边形的方法是 `drawPolygon` 和 `fillPolygon`, 定义如下:

```
void drawPolygon(Polygon p)
void fillPolygon(Polygon p)
```

参数是一个 `Polygon` 对象，给出多边形的顶点列表。`Polygon` 对象的 `addPoint` 方法可以依次加入多边形的各个顶点坐标。`Polygon` 还支持坐标平移，可以用 `translate` 方法将多边形的所有顶点平移，后面的参数是 X 和 Y 两个方向上平移的距离。

下面的代码绘出的多边形如图

```
Polygon p = new Polygon();
p.addPoint(10,10);
p.addPoint(100,30);
p.addPoint(50,50);
p.addPoint(100,70);
p.addPoint(30,100);
g.drawPolygon(p);
p.translate(150,0);
g.fillPolygon(p);
```

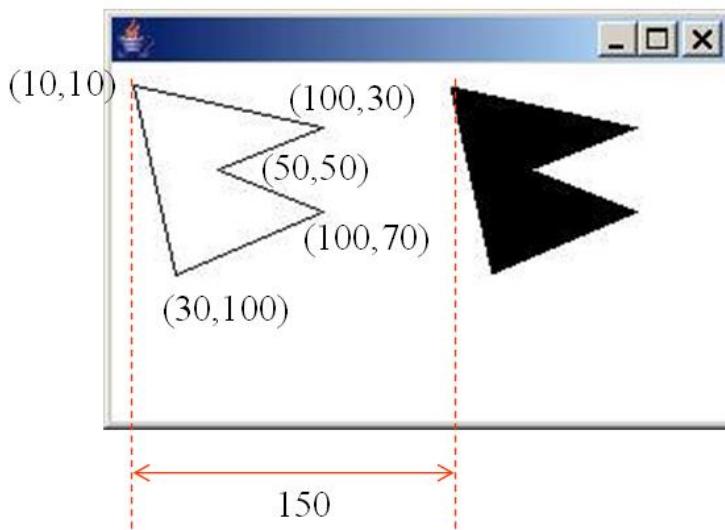


图 15-8 绘制多边形

上面所有绘图的语句都集成到了一个例子中。该例子通过启动时 `main` 函数的参数来控制绘制的几何形状，用数字 1-6 分别控制画直线、矩形、圆角矩形、椭圆、弧和多边形。完整的代码如下：

```
// GraphicsExample.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class GraphicsExample extends JFrame {
    public static void main(String[] args) {
        if(args.length == 0) {
            System.out.println("必须输入要绘制的图形的代码(1-6。");
        }
        else {
```

```

        JFrame win = new GraphicsExample(Integer.parseInt(args[0]));
        win.setVisible(true);
    }
}

public GraphicsExample(int aObjectToDraw) {
    setSize(300,200);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    Container contentPane = getContentPane();
    contentPane.add(new MyPanel(aObjectToDraw));
}
}

class MyPanel extends JPanel {
    private int objectToDraw = 0; //要绘制的图形的代码
    public MyPanel(int aObjectToDraw) {
        objectToDraw = aObjectToDraw;
    }
    public void paintComponent(Graphics g) {
        //绘图代码中测试 g 的各种方法。
        super.paintComponent(g);
        switch(objectToDraw) {
            case 1: //绘制直线
                g.drawLine(10,20,60,50);
                break;
            case 2: //绘制矩形
                g.drawRect(10,20,60,50);
                g.fillRect(80,20,60,50);
                break;
            case 3: //绘制圆角矩形
                g.drawRoundRect(10,10,100,50,20,15);
                g.fillRoundRect(160,10,100,50,20,15);
                break;
            case 4: //绘制椭圆
                g.drawOval(10,10,100,50);
                g.fillOval(160,10,100,50);
                break;
            case 5: //绘制弧线
                g.drawArc(10,10,100,50,0,60);
                g.fillArc(160,10,100,50,0,60);
                break;
            case 6: //绘制多边形
                Polygon p = new Polygon();
                p.addPoint(10,10);
                p.addPoint(100,30);
        }
    }
}

```

```

        p.addPoint(50,50);
        p.addPoint(100,70);
        p.addPoint(30,100);
        g.drawPolygon(p);
        p.translate(150,0); //将
        g.fillPolygon(p);
        break;
    }
}
}

```

15.3 颜色与字体

绘图对象可以改变当前画笔颜色，之后再绘制的图形就是新的颜色。改变画笔颜色用的方法是 `void setColor(Color c)`。参数是一个 `Color` 对象，可以用构造函数通过红、绿、蓝三元色值给出，也可以用 `Color` 类中预定义好的静态变量，如 `Color.blue`, `Color.yellow`, `Color.orange` 等。颜色值的范围为 0-255。

获得当前画笔颜色用绘图对象的 `getColor()`方法，返回值为 `Color` 对象。

下面的例子绘制 4 行每行 16 个小方块。第一行是红色由浅到深，第二行是绿色，第三行是蓝色，第四行是三种颜色同时变化。如图

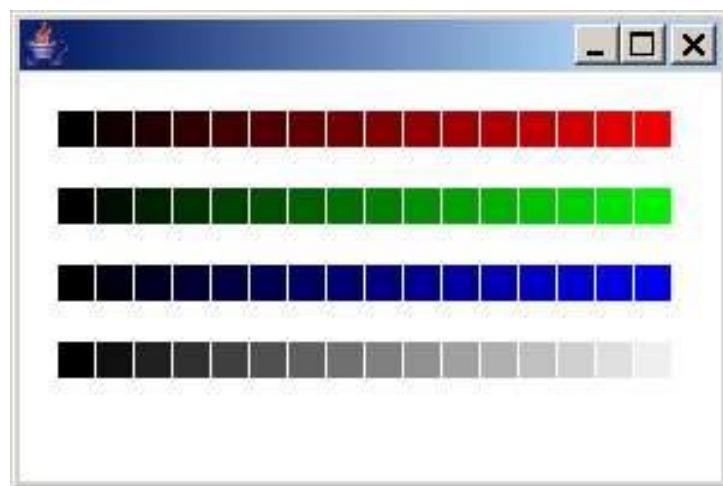


图 15-9 设置画笔颜色

例子的代码如下：

```

// ColorExample.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ColorExample extends JFrame {

```

```

public static void main(String[] args) {
    JFrame win = new ColorExample();
    win.setVisible(true);
}
public ColorExample() {
    setSize(300,200);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    Container contentPane = getContentPane();
    contentPane.add(new MyPanel());
}
}

class MyPanel extends JPanel {
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        int red = 0;
        int green = 0;
        int blue = 0;
        int gray = 0;
        for(red = 0; red <= 255; red += 16) {
            g.setColor(new Color(red,green,blue));
            g.fillRect(red+16,16,15,15);
        }
        red = 0;
        for(green = 0; green <= 255; green += 16) {
            g.setColor(new Color(red,green,blue));
            g.fillRect(green+16,48,15,15);
        }
        green = 0;
        for(blue = 0; blue <= 255; blue += 16) {
            g.setColor(new Color(red,green,blue));
            g.fillRect(blue+16,80,15,15);
        }
        blue = 0;
        for(gray = 0; gray <= 255; gray += 16) {
            g.setColor(new Color(gray,gray,gray));
            g.fillRect(gray+16,112,15,15);
        }
    }
}
}

```

绘图对象可以绘制文字，方法是 `drawString`，定义如下：

```
void drawString(String str, int x, int y)
```

参数 `str` 是要显示的文字字符串，`x`, `y` 是文字左上角的坐标。

下面的代码绘制的文字如图

```
g.drawString("Hello World!",50,50);
g.drawString("世界你好！ ",50,100);
```



图 15-10 绘制文字

文字的字体、尺寸和样式都可以设置。使用绘图对象的 `setFont` 方法，定义如下：

```
void setFont(Font font)
```

参数是一个 `Font` 对象，要用构造函数创建，构造函数定义如下：

```
Font(String name, int style, int size)
```

参数 `name` 是字体名称；`style` 是字体形式，为 `Font.PLAIN`, `Font.BOLD`, `Font.ITALIC` 或者其组合；`size` 是字体的高度，单位为像素。这三个字体的属性与 Word 里面的字体属性是一致的。

可以用下面的方法获取系统支持的所有字体：

```
String[] GraphicsEnvironment.getAvailableFontFamilyNames()
```

返回值是一个字符串数组，保存了所有字体的名字，每一个都可以用于构造 `Font` 对象。

获取绘图对象当前字体用 `getFont` 方法，定义如下：

```
Font getFont()
```

`Font` 对象也可以应用于组件上显示的字体。

下面的例子演示了设置字体的功能。在窗口中放置三个组合框，分别用于选择字体名称、字体形式和字体高度。字体名称组合框中可以选择系统支持的所有字体。字体形式可以选正常、粗体、斜体、粗斜体。字体高度可以选 1-100。改变一个选项就会影响窗口中字符串的显示。如图。



图 15-11 设置文字的字体、样式和尺寸

程序中每个组合框都添加了事件监听器，当选项变化时触发。监听器触发后会根据选项内容设置绘图对象的字体，并重新绘制窗口中的文字。代码如下：

```
// FontExample.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class FontExample extends JFrame {
    private MyPanel fontPanel = new MyPanel();      //显示字体用的 Panel
    private JComboBox nameBox = new JComboBox(); //字体名称
    private JComboBox styleBox = new JComboBox(); //字体形式
    private JComboBox sizeBox = new JComboBox(); //字体大小

    public static void main(String[] args) {
        JFrame win = new FontExample();
        win.setVisible(true);
    }

    public FontExample() {
        setSize(320,200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container contentPane = getContentPane();

        JPanel paramPanel = new JPanel();
        paramPanel.add(nameBox);
        paramPanel.add(styleBox);
        paramPanel.add(sizeBox);

        contentPane.add(paramPanel,BorderLayout.NORTH);
        contentPane.add(fontPanel);
    }
}
```

```

//设置字体名称组合框的选项和事件监听器
String[] fontNames = 
GraphicsEnvironment.getLocalGraphicsEnvironment().getAvailableFontFamilyNames();
for(int i = 0; i < fontNames.length; i++)
    nameBox.addItem(fontNames[i]);
nameBox.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e){
        Object o = nameBox.getSelectedItem();
        String name = (String)o;
        fontPanel.setFontName(name);
    }
});
nameBox.setSelectedIndex(0);

//设置字体形式组合框的选项和事件监听器
styleBox.addItem("普通");
styleBox.addItem("加粗");
styleBox.addItem("斜体");
styleBox.addItem("加粗斜体");
styleBox.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        int i = styleBox.getSelectedIndex();
        int style = 0;
        switch(i) {
            case 0:   style = Font.PLAIN;
                       break;
            case 1:   style = Font.BOLD;
                       break;
            case 2:   style = Font.ITALIC;
                       break;
            case 3:   style = Font.BOLD + Font.ITALIC;
                       break;
        }
        fontPanel.setFontStyle(style);
    }
});
styleBox.setSelectedIndex(0);

//设置字体大小组合框的选项和事件监听器
for(int i = 1; i < 100; i++)
    sizeBox.addItem(String.valueOf(i));
sizeBox.addActionListener( new ActionListener() {
    public void actionPerformed(ActionEvent e) {

```

```

        int i = sizeBox.getSelectedIndex() + 1;
        fontPanel.setFontSize(i);
    }
});

sizeBox.setSelectedIndex(13);
}

}

class MyPanel extends JPanel {
    private String fontName;
    private int fontStyle;
    private int fontSize;
    public void setFontParam(String aFontName, int aFontStyle, int aFontSize) {
        fontName = aFontName;
        fontStyle = aFontStyle;
        fontSize = aFontSize;
    }
    public MyPanel() {
        setFontParam("SansSerif",Font.PLAIN,14);
    }
    public void paintComponent(Graphics g) {
        super.paintComponent(g); //调用父类的绘制函数。
        g.setFont(new Font(fontName,fontStyle,fontSize)); //设置字体
        g.drawString("Hello World!",50,50);
        g.drawString("世界你好！ ",50,100);
    }
    public void setFontName(String aFontName) {
        fontName = aFontName;
        repaint();
    }
    public void setFontStyle(int aFontStyle) {
        fontStyle = aFontStyle;
        repaint();
    }
    public void setFontSize(int aFontSize) {
        fontSize = aFontSize;
        repaint();
    }
}

```

15.4 绘制图像

绘图对象的 `drawImage` 方法能够绘制将图片中内容绘制到窗口，定义如下：

```
drawImage(Image img, int dx, int dy, ImageObserver observer)
```

参数 `img` 是 `Image` 对象，可以从磁盘上的图片文件装入生成；`dx`, `dy` 是将图片绘制到什么位置，指的是左上角的坐标；`observer` 一般情况下可以给 `null`，但图片需要自动更新时，如动态 gif 图片，`observer` 应给的值是当前窗口。

下面的例子将磁盘上的图片文件 `globe.gif` 装入内存的 `Image` 对象，然后再绘制到窗口中。如图。

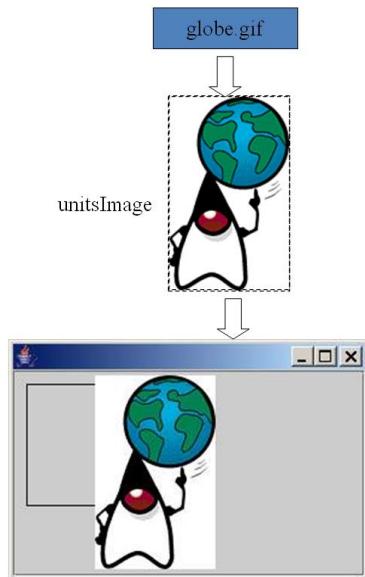


图 15-12 绘制图片

例子的代码如下：

```
// ImageExample.java
import java.awt.*;
import javax.swing.*;
import java.awt.image.*;
import java.awt.event.*;

public class ImageExample extends JFrame {
    private JPanel imagePanel = new JPanel();
    public static void main(String[] args) {
        JFrame win = new ImageExample();
        win.setVisible(true);
    }
    public ImageExample() {
        setSize(300,200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container contentPane = getContentPane();
```

```

contentPane.add(new JScrollPane(imagePanel));
imagePanel.addMouseListener( new MouseAdapter() {
    public void mouseClicked(MouseEvent e) {
        int x = e.getX();
        int y = e.getY();
        imagePanel.setUnitLocation(x,y);
    }
});
}
}

class ImagePanel extends JPanel {
private Image unitsImage = null;
private int unitX;
private int unitY;
public ImagePanel() {
    unitX = 10;
    unitY = 10;
    Toolkit kit = Toolkit.getDefaultToolkit();
    unitsImage = kit.getImage("globe.gif");
}
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.drawRect(10,10,100,100);
    g.drawImage(unitsImage,unitX,unitY,null);
}
public void setUnitLocation(int aX, int aY) {
    unitX = aX;
    unitY = aY;
    repaint();
}
}

```

方法 `draw` 有一个重载的方法，可以将图片的一部分绘制到组件上，定义如下：

```
drawImage(Image img, int dx1, int dy1, int dx2, int dy2, int sx1, int sy1, int sx2, int sy2,
ImageObserver observer)
```

此方法将 `img` 图片上的一个矩形区域绘制到画布上的一个矩形区域，`dx1`, `dy1` 和 `dx2`, `dy2` 是画布上目标矩形区域的左上角和右下角坐标，`sx1`, `sy1` 和 `sx2`, `sy2` 是源图片上一个矩形区域的左上角和右下角坐标。如果两个矩形区域大小不一样，会自动进行拉伸。如图

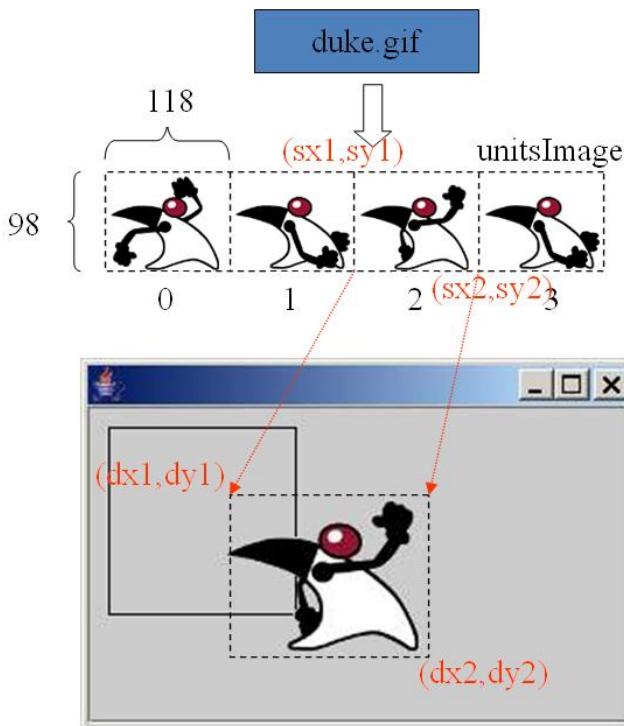


图 15-13 绘制部分图片

在上图的例子中，源图片分为四个相同大小的矩形区域，每个区域是一个行走的小人图像。在窗口中添加了一个鼠标事件，每次点击鼠标，就会在鼠标点击的位置绘制四幅图像之一，且第一次绘制第一幅，第二次绘制第二幅。这样，如果点的有节奏，就会出现小人移动的动画。同时，gif 图片支持透明色，所以源图片将小人轮廓之外的部分设成了透明色，绘制图片时这些区域不会覆盖后面的背景。

例子代码如下：

```
//ImageExample1.java
import java.awt.*;
import javax.swing.*;
import java.awt.image.*;
import java.awt.event.*;

public class ImageExample1 extends JFrame {
    private JPanel imagePanel = new JPanel();
    public static void main(String[] args) {
        JFrame win = new ImageExample1();
        win.setVisible(true);
    }
    public ImageExample1() {
        setSize(300,200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container contentPane = getContentPane();
        contentPane.add(new JScrollPane(imagePanel));
    }
}
```

```

        imagePanel.addMouseListener( new MouseAdapter() {
            public void mouseClicked(MouseEvent e) {
                int x = e.getX();
                int y = e.getY();
                imagePanel.setUnitLocation(x,y);
            }
        });
    }
}

class ImagePanel extends JPanel {
    private Image unitsImage = null;
    private int unitX = 0;
    private int unitY = 0;
    private int unitIndex = 0;
    public ImagePanel() {
        Toolkit kit = Toolkit.getDefaultToolkit();
        unitsImage = kit.getImage("duke.gif");
        //以下代码用于等待图片装入
        MediaTracker tracker = new MediaTracker(this);
        tracker.addImage(unitsImage,0);
        try {
            tracker.waitForID(0);
        }
        catch(Exception e) {
            System.out.println(e);
        }
    }
    public void paintComponent(Graphics g) {
        super.paintComponent(g);

        g.drawRect(10,10,100,100);

        int sx1 = 118 * unitIndex;
        int sy1 = 0;
        int sx2 = sx1 + 117;
        int sy2 = sy1 + 97;
        int dx1 = unitX;
        int dy1 = unitY;
        int dx2 = dx1 + 117;
        int dy2 = dy1 + 97;
        g.drawImage(unitsImage,dx1,dy1,dx2,dy2,sx1,sy1,sx2,sy2,null);
    }
    public void setUnitLocation(int aX, int aY) {

```

```

        unitX = aX;
        unitY = aY;
        unitIndex = (unitIndex + 1) % 4;
        repaint();
    }
}

```

在这个例子中，加载文件的时候使用了 **MediaTracker** 来跟踪图片的加载过程。这是因为 Java 加载图片文件(`kit.getImage`)的时候，采用异步的方式，即图片文件可能还没有加载完，`getImage` 方法就返回了，并继续执行后面的语句。这是为了适应网络速度慢的时候下加载图片时间比较长的情况，有时本地磁盘上图片文件较大时也需要较长加载时间，而且可以几幅图片一起加载。但异步方式带来的问题是，有可能图片还没加载完就被使用。因此需要一种机制等待图片加载完成。

使用 **MediaTracker** 的方法的步骤是：

- 1、创建一个 **MediaTracker** 对象；
- 2、用 `addImage` 方法将正在装入的图片加入到 **MediaTracker** 对象，并赋予一个编号；
- 3、用 `waitForID(编号)`方法等待图片装入完成。

Java 支持三种图片格式：

- **JPEG: Joint Photographic Experts Group**，支持全 24 位色彩。它是通过精确地记录每个像素的光亮但同时平均它们的色调的方法压缩图片，是有损压缩。
- **GIF: Graphics Interchange Format**，采用颜色索引的方式存储图片。一个 **GIF** 图片中只能有不多于 256 种的色彩，因此无法存储高质量照片。一个 **GIF** 文件可以包含几张图形以及每张图形的持续值，以产生动画效果。它也有有限度的可透明性：调色板中的某个色彩可被指定为透明色。
- **PNG: Portable Network Graphics**，无损压缩，适合在网络中传播；具有 8 位、24 位和 32 位三中色彩深度；支持 Alpha 通道透明（32 位）和色彩索引透明（8 位）。

Java 支持 **GIF**、**JPEG**、**PNG** 图形文件的读取，但只支持 **JPEG**、**PNG** 图形文件的写入。

15.5 其它绘图功能

Graphics 类是一个抽象类，所以 `paintComponent(Graphics g)`方法中的参数 `g` 不可能是一个 **Graphics** 类型的对象。参数 `g` 实际上是 **Graphics** 的子类 **Graphics2D** 类型的对象。**Graphics2D** 类提供的功能比 **Graphics** 类强大，包括：

- 支持绘制更复杂的形状，如二次曲线、三次曲线；
- 支持更复杂的坐标变换，如旋转等；
- 支持设置线型，如实线、虚线、线条粗细；
- 支持更复杂的填充方法，如多种颜色着色。

要使用 **Graphics2D** 类的功能时，可以直接进行造型，代码是：

```
Graphics2D g2 = (Graphics2D) g;
```

然后就可以通过 g2 使用 Graphics2D 的所有方法了。Graphics2D 的方法可以查阅类库文档。

Java 的绘图速度并不慢，因为其底层实现应用了快速引擎，在 Windows 平台是基于 DirectX 实现的。

采用 Java 也可以实现缓冲区绘图，即先将所有内容绘制到内存中的图片对象，再将该图片一次性复制到窗口中显示。在内存中创建能绘图的图片对象用的是 BufferedImage 类(Image 类的子类)，该类还能实现取颜色和保存到文件的功能，这些都是直接在窗口画布上绘图不容易实现的。

采用双缓冲区绘图时按照以下步骤：

1. 创建一个 BufferedImage 对象，作为内存中的绘图画布；
2. 所有绘图操作均在 BufferedImage 对象上进行；
3. 绘制完成后，将 BufferedImage 对象整个绘制到 JPanel 上；

这样，BufferedImage 对象中的图像和 JPanel 的图像将完全一样。如图

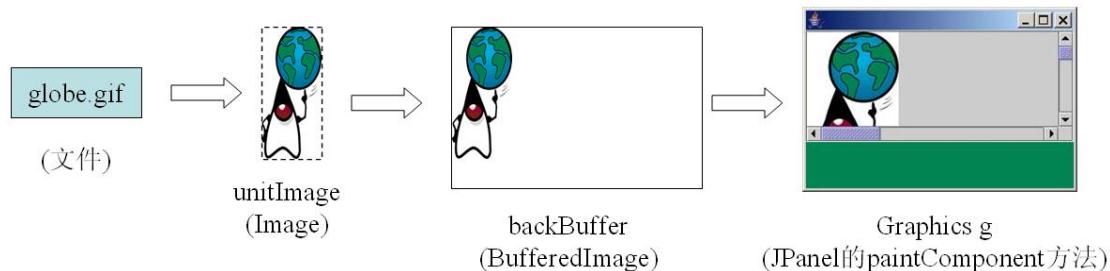


图 15-14 缓冲区绘图

因为 BufferedImage 和 JPanel 的图像完全相同，所以，若想从 JPanel 图像某个位置取色，可以直接从 BufferedImage 对象同一个位置取。要将 JPanel 图像保存到文件，和将 BufferedImage 对象的图像保存到文件是一样的。

BufferedImage 取色的方法定义如下：

```
int getRGB(int x, int y)
```

参数 x, y 定义了要取颜色的点的位置。返回一个 int 值，用四个字节分别表示红、绿、蓝和透明度。Color 类有一个构造函数 Color(int c)，可以直接用此 int 值构造一个颜色对象。

将图片保存到文件使用 ImageIO 类的静态方法 write，定义如下：

```
static boolean write(RenderedImage im, String formatName, File output)
```

参数 RenderedImage 为一个接口，BufferedImage 类实现了此接口，所以可以用 BufferedImage 对象作为参数；formatName 为字符串“JPEG”或“PNG”，表示图片格式；output 为文件对象。

下面的例子使用了缓冲区绘图，并在此基础上实现了取色和保存到文件的功能。当用鼠标点击图片时，该处的颜色会被设置成下面的面板的颜色。如果按下 Save 按钮，则会将图片内容保存到磁盘上的一个文件中。如图



图 15-15 保存绘图内容

例子代码如下：

```
// DoubleBufferExample.java
import java.awt.*;
import javax.swing.*;
import java.awt.image.*;
import java.awt.event.*;
import javax.imageio.*;
import java.io.*;

public class DoubleBufferExample extends JFrame {
    private PhotoPanel photoPanel = new PhotoPanel();
    private JPanel colorPanel = new JPanel();
    public static void main(String[] args) {
        JFrame win = new DoubleBufferExample();
        win.setVisible(true);
    }
    public DoubleBufferExample() {
        setSize(300,200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container contentPane = getContentPane();
        contentPane.add(new JScrollPane(photoPanel));
        contentPane.add(colorPanel, BorderLayout.SOUTH);
        photoPanel.addMouseListener( new MouseAdapter() {
            public void mouseClicked(MouseEvent e) {
                Color c = photoPanel.getColor(e.getX(),e.getY());
                colorPanel.setBackground(c);
                colorPanel.repaint();
            }
        });
        colorPanel.setPreferredSize(new Dimension(20,50));
    }
}
```

```

        JButton saveButton = new JButton("Save");
        colorPanel.add(saveButton,BorderLayout.EAST);
        saveButton.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                photoPanel.saveToFile("example.jpg");
            }
        });
    }
}

class PhotoPanel extends JPanel {
    private BufferedImage backImage = new
    BufferedImage(800,600,BufferedImage.TYPE_BYTE_INDEXED);
    private Image photoImage = null;
    public PhotoPanel() {
        Toolkit kit = Toolkit.getDefaultToolkit();
        photoImage = kit.getImage("globe.gif");
        setPreferredSize(new Dimension(backImage.getWidth(),backImage.getHeight()));
        setLayout(null);
    }
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        Graphics backG = backImage.getGraphics();
        backG.drawImage(photoImage,0,0,null);
        backG.dispose();
        g.drawImage(backImage,0,0,null);
    }
    public Color getColor(int x, int y) {
        int c = backImage.getRGB(x,y);
        return new Color(c);
    }
    public void saveToFile(String aFileName) {
        File f = new File(aFileName);
        try {
            ImageIO.write(backImage,"JPG",f);
        }
        catch(Exception e) {
            System.out.println(e);
        }
    }
}

```

第十六章 多线程

16.1 线程的概念

线程是一段执行着的程序。任何程序都是在线程中执行的。实际上我们前面设计的程序也都是在线程中执行的。在用 java 命令运行 class 文件的时候，虚拟机会创建一个线程，并在线程中运行该 class 文件的 main 函数。

线程是为了在程序中实现多任务并发。一个运行的程序可以有多个线程，每个线程都有自己的流程，可以是相互独立的作业，彼此之间互不干扰。多个线程在多处理器系统上可以并行执行；也可以在单处理器系统上轮转并发执行，因为轮转的很快，所以从外部来看就像同时在执行。这样，编写代码时按每个线程的流程单独编写，运行时由操作系统调度其交替执行。不用在编写代码时考虑流程的交替执行，所以方便了多任务程序的设计。

一个程序中的多个线程是共享程序的数据区的，如图。这有许多好处：一是线程切换时不用切换数据区，减小了开销；二是线程间交换数据方便，通过共享变量就能实现。但是，这也有缺点，当两个以上的线程使用同一个资源时，容易引起冲突。

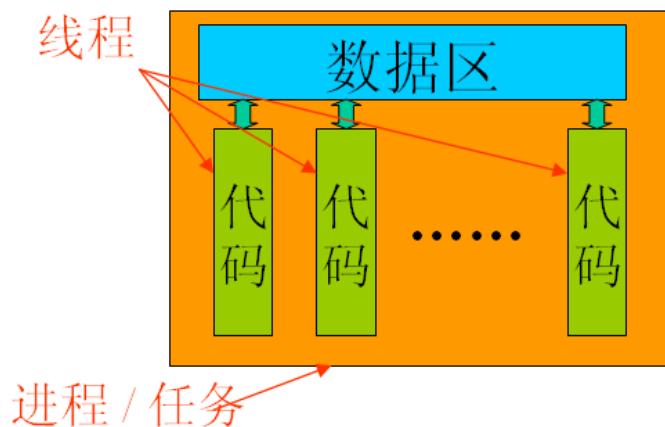


图 16-1 多线程模型

Java 支持多线程，每个线程当作一个对象来管理，所以创建线程就像创建对象一样非常简单，也提供了对冲突解决的支持。

16.2 创建线程

Java 中创建线程主要有两种方式，一种是用 Thread 类，一种是用 Runnable 接口。

用 Thread 类创建线程分为两步：

1. 由 Thread 类派生一个类，覆盖其 public void run()方法，run 方法中就是要在线程中执行的语句；
2. 创建一个该类的对象，调用对象的 start 方法。

下面是一个创建线程的例子。从 `Thread` 类派生出了一个新类 `MyThread`，覆盖了 `run` 方法，打印 100 行文本。在 `main` 函数中创建了一个 `MyThread` 的对象，并调用了其 `start` 方法。程序启动时，有一个线程在执行 `main` 函数，可以称此线程为 `main` 线程或主线程。当创建 `MyThread` 对象时，又创建了一个线程，调用 `start` 方法将其投入运行。线程运行完 `run` 方法后程序会正常退出。程序运行时的线程结构如图：

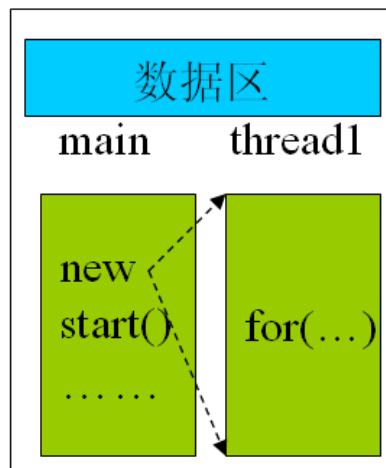


图 16-2 创建线程

例子代码如下：

```
// ThreadCreate1.java
public class ThreadCreate1 {
    public static void main(String[] args) throws Exception {
        MyThread thread1 = new MyThread();
        thread1.start();
        System.out.println("Main: Thread started");
    }
}

class MyThread extends Thread {
    public void run() {
        for(int i = 0; i < 100; i++) {
            System.out.println("Thread: running " + i);
        }
    }
}
```

当要用一个已经有了父类、不方便再用 `Thread` 作父类的类来作线程类时，可以用 `Runnable` 接口。用 `Runnable` 接口创建线程的步骤是：

1. 用一个类实现 `Runnable` 接口，实现接口中的 `run` 方法，`run` 方法中就是要在线程中执行的语句；
2. 用一个该类的对象，作为 `Thread` 构造函数的参数去创建一个 `Thread` 对象，调用其 `start`

方法。

下面是一个用 `Runnable` 接口创建线程的例子，实现的功能和上一个例子相同。例子代码如下：

```
// ThreadCreate2.java
public class ThreadCreate2 {
    public static void main(String[] args) throws Exception {
        Thread thread1 = new Thread(new MyThread());
        thread1.start();
        System.out.println("Main: Thread started");
    }
}

class MyThread implements Runnable {
    public void run() {
        for(int i = 0; i < 100; i++) {
            System.out.println("Thread: running " + i);
        }
    }
}
```

16.3 线程的状态

如果有多个线程需要执行，但 CPU 或者内核只有一个，到底是如何执行的？这个问题涉及操作系统的线程调度机制。通常采用时间片轮转的方式来执行线程，也就是说将 CPU 或内核的执行时间分为许多很小的片段，用一个时间片执行一个线程，时间片结束后会切换到执行另一个线程。另外，一个线程在执行过程中如果由于某种原因没事可做了，会主动放弃 CPU 或内核，让其去执行其它线程。

下面是一个演示时间片轮转的例子。在上一节例子的基础上，再创建一个同样的线程，也是打印 100 行文本，看两个线程同时执行时的情况。注：在单核的系统上，这两个线程会被调度交替执行，执行结果反映的是线程调度的情况；在多核的系统上，这两个线程是在不同的核上并行执行的，不用时间片轮转，此时反映的是两个线程抢占控制台输出的情况。

例子的代码如下：

```
// MultiThread1.java
public class MultiThread1 {
    public static void main(String[] args) throws Exception {
        MyThread thread1 = new MyThread(1); // 创建第一个线程
        thread1.start();
        System.out.println("Main: thread1 started");
        MyThread thread2 = new MyThread(2); // 创建第二个线程
        thread2.start();
    }
}
```

```

        System.out.println("Main: thread2 started");
    }
}

class MyThread extends Thread {
    private int index; //保存本线程的编号
    public MyThread(int aIndex) {
        super();
        index = aIndex;
    }
    public void run() {
        for(int i = 0; i < 100; i++) {
            System.out.println("Thread" + index + ": running " + i);
        }
    }
}

```

从运行结果可以看出，两个线程的输出是交替的，说明是交替执行的。如果执行多次，每次的执行结果是不一样的，说明线程的切换时机每次都不一样。这是由操作系统的线程调度机制决定的，不同操作系统调度机制不同。一般无法精确预料切换的时机，在设计多线程程序时要特别注意这一点。

修改一下这个例子，就可以演示线程主动让出 CPU 或内核的情况。将两个线程都改为每隔 10 毫秒输出一个字符串，这样在输出完一行后，线程就有 10 毫秒无事可做，就会主动让出 CPU 或内核。让线程等待一段时间用 `Thread` 类的 `sleep` 方法，定义如下：

```
public static native void sleep(long millis) throws InterruptedException;
```

如果是用 `Thread` 类来创建线程，在 `run` 方法中可以直接调用 `sleep(10)`。如果是用 `Runnable` 接口来创建线程，应该写成 `Thread.sleep(10)`。注意捕获可能抛出的异常。例子代码如下：

```

// MultiThread2.java
public class MultiThread2 {
    public static void main(String[] args) throws Exception {
        MyThread thread1 = new MyThread(1); // 创建第一个线程
        thread1.start();
        System.out.println("Main: thread1 started");
        MyThread thread2 = new MyThread(2); // 创建第二个线程
        thread2.start();
        System.out.println("Main: thread2 started");
    }
}

class MyThread extends Thread {
    private int index; //保存本线程的编号
    public MyThread(int aIndex) {

```

```

        super();
        index = aIndex;
    }
    public void run() {
        for(int i = 0; i < 100; i++) {
            System.out.println("Thread" + index + ": running " + i);
            try {
                sleep(10);
                //如果 MyThread 不是由 Thread 派生而来，而是实现 Runnable 接口，则应该写成下面的形式：
                //Thread.sleep(10);
            }
            catch(InterruptedException e) { //捕获异常
                System.out.println(e);
            }
        }
    }
}

```

线程有四个状态，分别是：

- 新线程：刚刚用 `new` 操作符创建了一个线程对象，代码尚未开始执行；此时可以进行一些属性设置。
- 可运行：新线程调用 `start` 方法后，就成为可运行的，参与到整个线程调度中，有资格获得 CPU 或者内核来执行。可运行的线程可能获得了 CPU 或者内核正在运行，也可能没有获得 CPU 或者内核还在排队。
- 被中断：可运行的线程因为某些原因无法继续运行，就进入被中断状态，此时已不在调度队列中。当某些条件满足后，再重新回到可运行状态。
- 死线程：当线程的 `run` 方法运行结束，或者出现了未捕获的异常时，可运行的线程成为死线程。

线程四个状态间的转换关系如图。

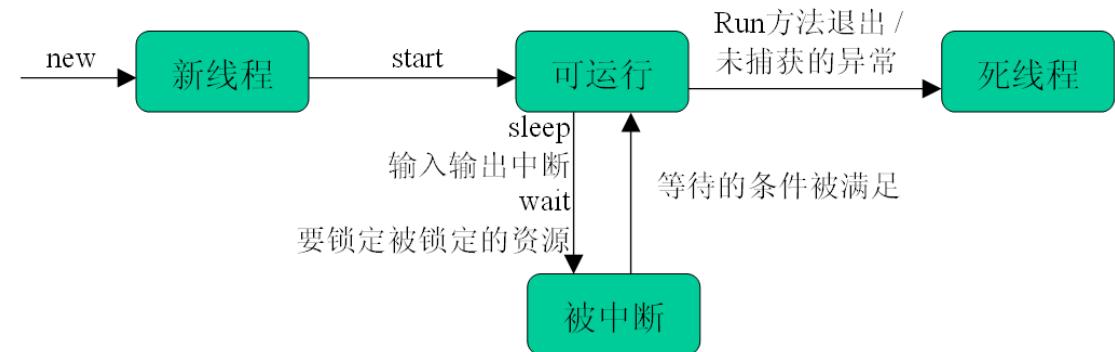


图 16-3 线程的状态转换

16.4 事件处理线程

实际上，一个 Java 程序运行时自动创建的不只 main 线程，还有垃圾回收线程和事件处理线程。垃圾回收线程会在系统空闲时运行，回收内存中不再使用的对象。事件处理线程负责根据事件调用相应的事件监听器。所以，Java 程序运行时的线程模型如图。

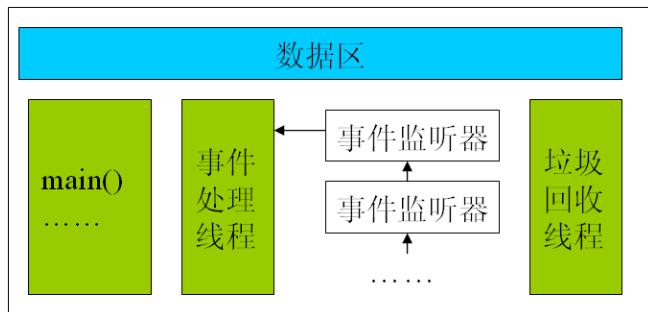


图 16-4 事件处理线程

事件处理线程负责依次调用事件队列中的待处理操作。因为所有的事件处理都使用这一个线程，所以要一个事件处理完才能处理另一个事件。可以通过下面的例子来验证这一点。在窗口中添加一个鼠标事件监听器，在事件监听器中每隔 1 秒输出一行字符串显示鼠标点击的位置，共循环 5 次。如图。

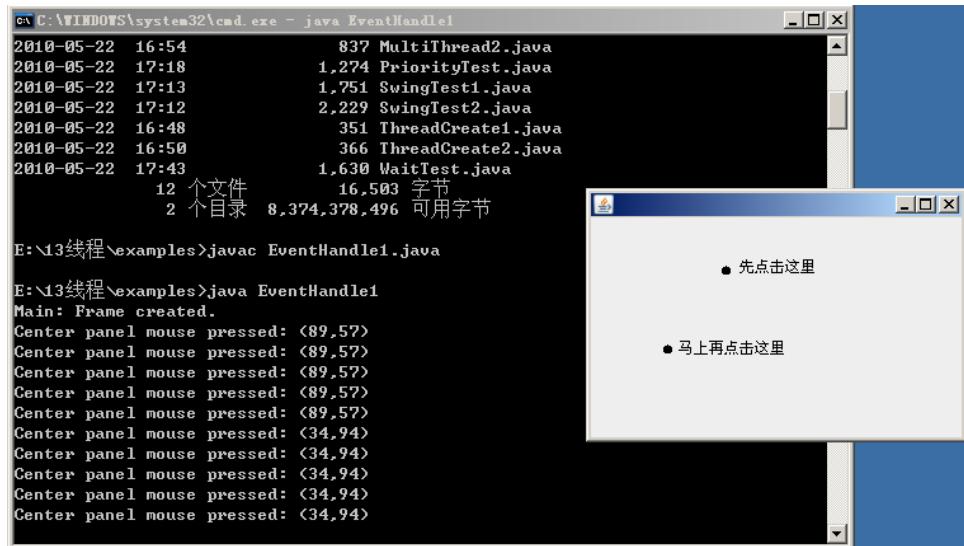


图 16-5 事件中进行耗时操作

这样，一个事件的响应需要 5 秒钟，如果在一个事件的响应还没有执行完的时候就又点击，会如何响应呢？运行结果表明，对后一次点击的响应要等前一次响应完成后才会开始。带来的启示是：事件处理要尽量少占用时间，否则会影响其它事件的处理。

例子的代码如下：

```
// EventHandle1.java
import java.awt.*;
import java.awt.event.*;
```

```

import javax.swing.*;

public class EventHandle1 extends JFrame {
    public static void main(String[] args) throws Exception {
        JFrame win = new EventHandle1();
        win.setVisible(true);
        System.out.println("Main: Frame created.");
    }
    public EventHandle1() {
        setSize(300,200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container contentPane = getContentPane();

        JPanel cPanel = new JPanel();
        contentPane.add(cPanel);

        cPanel.addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent e) {
                int x = e.getX();
                int y = e.getY();
                for(int i = 0; i < 5; i++) {
                    System.out.println("Center panel mouse pressed: (" + x + "," + y +
                ")");
                }
            }
        });
    }
}

```

如果事件处理中必需进行耗时的工作，应该启动新的线程来做，从而让事件处理线程可以处理其他事件。修改上面的例子，将输出 5 个字符串的工作放到另外的线程中去执行，那么多次点击就可以同时响应了。程序代码如下：

```

// EventHandle2.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class EventHandle2 extends JFrame {

```

```

public static void main(String[] args) throws Exception {
    JFrame win = new EventHandle2();
    win.setVisible(true);
    System.out.println("Main: Frame created.");
}

public EventHandle2() {
    setSize(300,200);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    Container contentPane = getContentPane();

    JPanel cPanel = new JPanel();
    contentPane.add(cPanel);

    cPanel.addMouseListener(new MouseAdapter() {
        public void mousePressed(MouseEvent e) {
            int x = e.getX();
            int y = e.getY();
            String s = "Center panel mouse pressed: (" + x + "," + y + ")";
            Thread t = new EventThread(s);
            t.start();
        }
    });
}
}

class EventThread extends Thread {
    private String outString;
    public EventThread(String s) {
        outString = s;
    }
    public void run() {
        for(int i = 0; i < 5; i++) {
            System.out.println(outString);
            try {
                Thread.sleep(1000);
            }
            catch(Exception exc) {
                exc.printStackTrace();
            }
        }
    }
}

```

16.5 线程与 swing 组件

如果在线程中操作 swing 组件，需要特别注意，swing 组件不是线程安全（thread safe）的。也就是说，多个线程同时修改一个 swing 组件的属性时，会发生冲突。组件是响应 repaint 事件而绘制的，所以是在事件处理线程中绘制的。如果在其他线程中操作 swing 组件的属性，将会有不可预知的异常产生。

下面的例子演示了这种冲突的发生。窗口中有一个 JList 组件，有两个线程，各自随机地向 JList 中添加或删除选项（删除时保证组件中有选项可以删除）。运行时将不一定在什么时候产生异常，类型为数组下标 OutOfBounds。

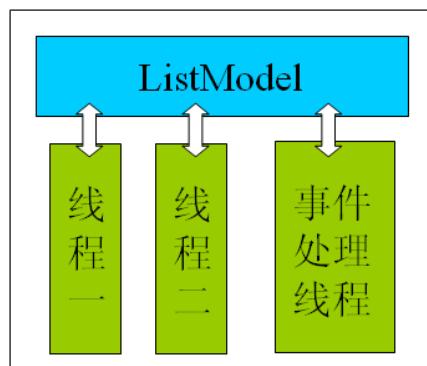


图 16-6 多线程和 swing 组件

异常产生的过程是这样的：

- 1、事件处理线程要绘制列表；
- 2、事件处理线程调用 size() 得到列表项个数 n；
- 3、线程切换，两个线程从列表中删除掉了某些列表项；
- 4、事件处理线程依次调用 get(0) 到 get(n-1) 方法取得列表内容；
- 5、而此时列表中实际内容少于 n 项，所以 OutOfBounds。

例子的代码如下：

```
// SwingTest1.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;
import java.util.List;

public class SwingTest1 extends JFrame {
    private MyListModel aListModel = new MyListModel();
    private JList aList = new JList(aListModel);
    private Random generator = new Random();
    public static void main(String[] args) throws Exception {
        JFrame win = new SwingTest1();
```

```

        win.setVisible(true);
    }

    public SwingTest1() {
        setSize(300,200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container contentPane = getContentPane();

        contentPane.add(new JScrollPane(aList));

        //创建两个线程来操作 JList 的内容
        Thread t1 = new OpThread();
        t1.start();
        Thread t2 = new OpThread();
        t2.start();
    }

    class OpThread extends Thread {
        public void run() {
            try {
                for(int j = 0; j < 10000; j++) {
                    int i = generator.nextInt(2); //产生随机数
                    if(i == 1) {//50%的概率添加一行
                        aListModel.append("step "+ j);
                    }
                    else { //50%的概率删除一行
                        if(aListModel.getSize()>0) //删除之前先判断是否有行可
                            delete
                        aListModel.remove(0);
                    }
                    sleep(1);
                }
            }
            catch(Exception e) {
                e.printStackTrace();
            }
        }
    }
}

class MyListModel extends AbstractListModel {
    private List<String> data = new Vector<String>();
    public int getSize() {
        return data.size();
    }
}

```

```

public String getElementAt(int i) {
    return data.get(i);
}
public void append(String o) {
    data.add(o);
    int n = data.size();
    fireIntervalAdded(this,n-1,n-1);
}
public void remove(int i) {
    int n = data.size(); //得到列表中元素个数
    if(i < n) { //保证删除时列表中有元素
        data.remove(i);
        fireIntervalRemoved(this,i,i); //通知窗口更新界面
    }
}
}

```

如果需要在线程中操作 Swing 组件，安全的方法是将操作添加到事件队列中，让事件处理线程来执行这些操作。一个操作是一个实现了 `Runnable` 接口的对象，操作的指令在 `run` 方法中。将一个操作添加到事件队列使用 `EventQueue` 类的静态方法 `invokeLater`。如果将上面的例子中，线程添加或删除列表框选项的操作都做成事件添加到事件队列中，那么就不会出现异常了。

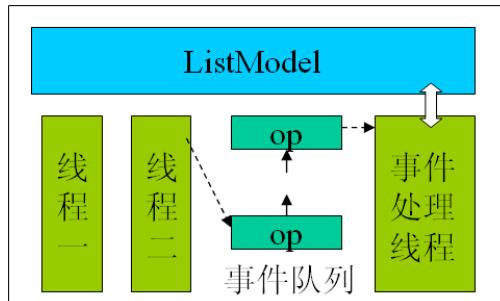


图 16-7 使用事件队列访问 swing 组件

代码如下：

```

// SwingTest2.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;
import java.util.List;

public class SwingTest2 extends JFrame {
    private MyListModel aListModel = new MyListModel();
    private JList aList = new JList(aListModel);
    private Random generator = new Random();

    public void actionPerformed(ActionEvent e) {
        int index = generator.nextInt(10);
        aListModel.append("Item " + index);
    }
}

```

```

public static void main(String[] args) throws Exception {
    JFrame win = new SwingTest2();
    win.setVisible(true);
}
public SwingTest2() {
    setSize(300,200);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    Container contentPane = getContentPane();

    contentPane.add(new JScrollPane(aList));

    Thread t1 = new OpThread();
    t1.start();
    Thread t2 = new OpThread();
    t2.start();
}

class OpThread extends Thread {
    public void run() {
        try {
            for(int j = 0; j < 10000; j++) {
                int i = generator.nextInt(2);
                if(i == 1) {
                    //aListModel.append("step "+ j);
                    Runnable op = new OpList(aListModel, 1, "step"+j);
                    EventQueue.invokeLater(op);
                }
                else {
                    //if(aListModel.getSize()>0)
                    //  aListModel.remove(0);
                    Runnable op = new OpList(aListModel, 2, null);
                    EventQueue.invokeLater(op);
                }
                sleep(1);
            }
        }
        catch(Exception e) {
            e.printStackTrace();
        }
    }
}

class OpList implements Runnable { //操作 JList 的对象
    private MyListModel listModel;

```

```

private int opCode;
private String text;
public OpList(MyListModel l, int o, String t) {//o 为 1 表示将 t 添加到 l 中, o 为 2
表示从 l 中删除第一行
    listModel = l;
    opCode = o;
    text = t;
}
public void run() {
    switch(opCode) {
        case 1: listModel.append(text);
        break;
        case 2: listModel.remove(0);
        break;
    }
}
}

class MyListModel extends AbstractListModel {
    private List<String> data = new Vector<String>();
    public int getSize() {
        return data.size();
    }
    public String getElementAt(int i) {
        return data.get(i);
    }
    public void append(String o) {
        data.add(o);
        int n = data.size();
        fireIntervalAdded(this,n-1,n-1);
    }
    public void remove(int i) {
        int n = data.size();
        if(i < n) { //保证删除时列表中有元素
            data.remove(i);
            fireIntervalRemoved(this,i,i);
        }
    }
}

```

16.6 线程的优先级

为了区分不同任务的紧急程度，线程添加了优先级的属性。优先级高的线程调度时获得的执行时间较多，所以执行较快。但并不是说优先级低的线程完全无法获得执行，而是获得的执行时间较少，执行得较慢。

Java 中线程的优先级分为 10 级(1-10)，线程创建后默认为 5。设置线程优先级使用 Thread 类的 `setPriority(int newPriority)` 方法。如果操作系统支持的级别小于 10 个，某些 Java 线程级别会映射为相同的操作系统线程级别。

下面的例子演示了优先级对线程执行速度的影响。在窗口中有 10 个进度条，分别用 10 个线程控制这 10 个进度条从 0 到 100%。当所有线程优先级相同时，执行结果如左图。当 10 个线程优先级分别为 1-10 时，执行结果如右图。可以看出，优先级越高执行得越快，并且有的优先级映射到了相同的操作系统线程优先级。

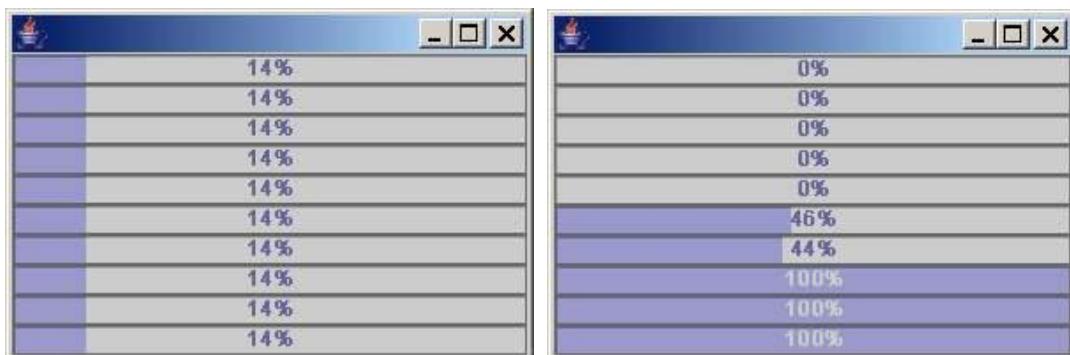


图 16-8 线程优先级

例子的代码如下：

```
// PriorityTest.java
import javax.swing.*;
import java.awt.*;

public class PriorityTest extends JFrame {
    private JProgressBar[] progBar = new JProgressBar[10];
    public static void main(String[] args) {
        JFrame win = new PriorityTest();
        win.setVisible(true);
    }
    public PriorityTest() {
        setSize(300,200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container contentPane = getContentPane();
        contentPane.setLayout(new GridLayout(10,1));
```

```

        for(int i=0; i<10; i++)    {
            progBar[i] = new JProgressBar(0,100000);
            progBar[i].setStringPainted(true); //显示中间的文字 **%
            contentPane.add(progBar[i]);
            Thread r = new RunnerThread(i);
            //r.setPriority(i+1);      //设置线程优先级
            r.start();
        }
    }

    class RunnerThread extends Thread {
        private int index;
        public RunnerThread(int aIndex)  {
            index = aIndex;
        }
        public void run() {
            try {
                for(int i = 0; i <= 100000; i++) {
                    EventQueue.invokeLater(new Op(progBar[index],i)); // 调用
                }
            } catch(Exception e) {
                System.out.println(e);
            }
        }
    }

    class Op implements Runnable {
        private JProgressBar prog;
        private int val;
        public Op(JProgressBar p, int v) {
            prog = p;
            val = v;
        }
        public void run() {
            prog.setValue(val);
        }
    }
}

```

16.7 同步方法

两个以上的线程同时访问和修改同一个对象的状态时，会造成对象的状态不确定。这种冲突

需要通过临界区机制来解决。要访问或修改该对象的状态必须先进入临界区，而同一时刻只允许有一个线程进入临界区。如果其它线程也想进入临界区，必须先等待第一个线程退出临界区。这样就保证了同一时刻只有一个线程在操作该对象，从而避免了冲突。临界区在 Java 中通过同步方法或同步代码段来实现，属于线程同步机制。

下面通过一个电影院售票系统来解释线程同步机制。一家电影院有多个售票机，同时销售一个大厅的票。每个售票机是一台客户端，都连接到后台的一个服务器获取数据。用户购票流程是：查询空座，选择座位，实际交易。和用户的交互都有售票机完成，但最终都要通过网络发送到服务器实际执行。服务器对应于每个售票机有一个服务线程（在网络通信一章会介绍），这些线程都需要修改售票数据，而这些数据保存在服务器的内存中的一个对象中。这就造成了多个线程同时访问一个对象的冲突，如图

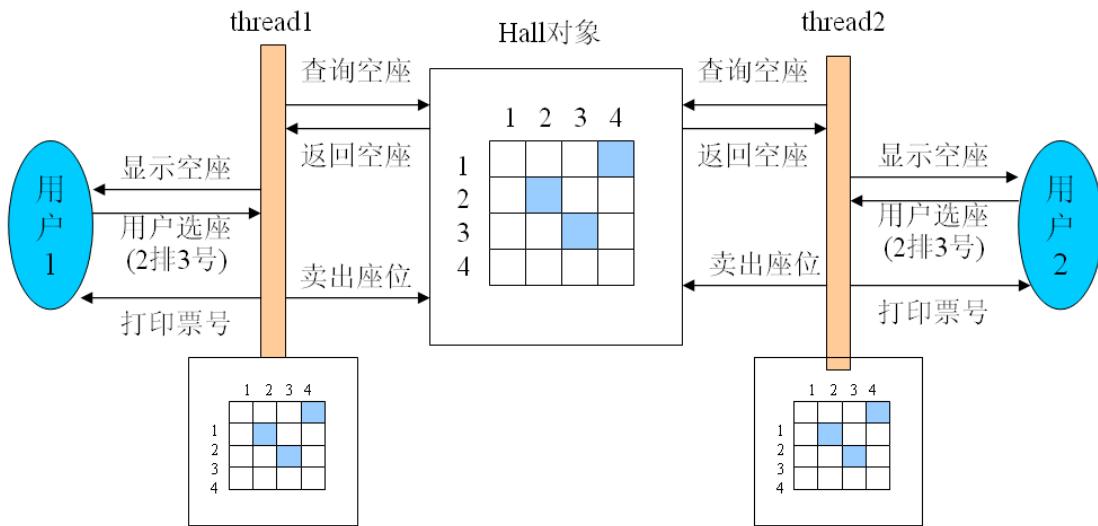


图 16-9 电影院售票系统

为了突出线程同步部分，例子中省略了网络通信部分。在一个程序中模拟了两个售票机，两个售票线程，以及保存售票数据的对象。图中，中间的 Hall 对象用一个布尔型的二维数组保存座位是否售出的数据，数组下标对应座位的排和号。左右两边各有一个售票线程，都需要访问和修改中间 Hall 对象。同时，每个线程保存有一个售票数据副本，模拟售票机通过网络通信从服务器得到的数据。

服务器和售票机中的数据会显示在一个窗口中，如图，左右两边是售票机的界面，中间显示服务器数据，用白色方块表示尚未卖出的座位，蓝色方块表示已卖出。在售票机界面的下方可以点“查询”按钮查询服务器售票数据，可以输入要购买的座位排号，然后点“购买”确认。

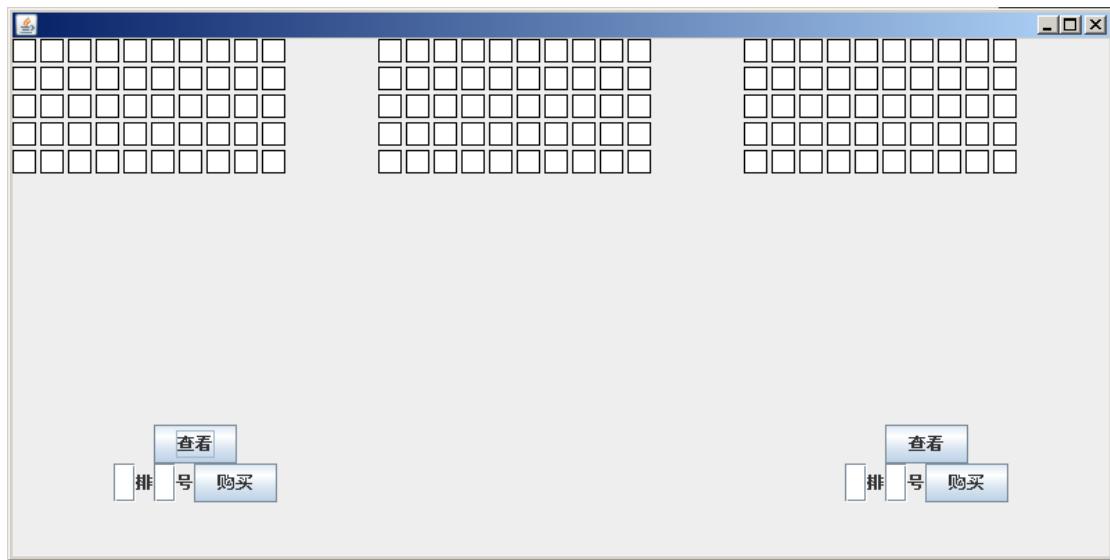


图 16-10 电影院售票系统界面

例子的代码如下：

```
// Cinema1.java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class Cinema1 extends JFrame {
    public static void main(String[] args) {
        JFrame win = new Cinema1();
        win.setVisible(true);
    }
    public Cinema1() {
        setSize(800,400);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container contentPane = getContentPane();
        contentPane.setLayout(new GridLayout(1,3));

        HallPanel server = new HallPanel(5,10);
        SellPanel client1 = new SellPanel(server);
        SellPanel client2 = new SellPanel(server);
        contentPane.add(client1);
        contentPane.add(server);
        contentPane.add(client2);
    }
}

class SellPanel extends JPanel {
```

```
private HallPanel localHallPanel = new HallPanel(5,10);
private HallPanel serverHallPanel;
public SellPanel(HallPanel aServer) {
    serverHallPanel = aServer;

    setLayout(new BorderLayout());
    add(localHallPanel, BorderLayout.CENTER);
    JPanel controlPanel = new ControlPanel();
    add(controlPanel, BorderLayout.SOUTH);

}

class ControlPanel extends JPanel {
    private JTextField rowText = new JTextField();
    private JTextField columnText = new JTextField();
    private JLabel info = new JLabel();
    public ControlPanel() {
        setPreferredSize(new Dimension(10,100));

        Box b1 = Box.createHorizontalBox();
        JButton lookupButton = new JButton("查看");
        b1.add(lookupButton);

        Box b2 = Box.createHorizontalBox();
        rowText.setPreferredSize(new Dimension(16,16));
        b2.add(rowText);
        b2.add(new JLabel("排"));
        columnText.setPreferredSize(new Dimension(16,16));
        b2.add(columnText);
        b2.add(new JLabel("号"));
        JButton buyButton = new JButton("购买");
        b2.add(buyButton);

        Box b3 = Box.createHorizontalBox();
        info.setPreferredSize(new Dimension(32,32));
        b3.add(info);

        Box vBox = Box.createVerticalBox();
        vBox.add(b1);
        vBox.add(b2);
        vBox.add(b3);
        add(vBox);

        lookupButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {

```

```

        localHallPanel.setSits(serverHallPanel.getSits());
    }
});

buyButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        Thread t = new RunnerThread();
        t.start();
    }
});
}

class RunnerThread extends Thread {
    public RunnerThread() {
    }
    public void run() {
        try {
            int row = Integer.parseInt(rowText.getText());
            int column = Integer.parseInt(columnText.getText());

            if(localHallPanel.sellSit(row,column)) { //卖票前查询本地数据
                serverHallPanel.sellSit(row,column);
            }

            if(serverHallPanel.sellSit(row,column)) { //卖票前查询服务器数据
                localHallPanel.sellSit(row,column);
            }

            info.setText(row+"排"+column+"号座位售出");
        }
        else {
            info.setText("座位已卖出");
        }
    }
    catch(Exception e) {
        e.printStackTrace();
    }
}
}

class HallPanel extends JPanel {
    private boolean[][] sits;
    public HallPanel(int rows, int columns) {
        sits = new boolean[columns][rows];
        for(int y = 0; y < rows; y++) {

```

```

        for(int x = 0; x < columns; x++) {
            sits[x][y] = false;    //初始都为空座
        }
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        int columns = sits.length;
        int rows = sits[0].length;
        for(int y = 0; y < rows; y++) {
            for(int x = 0; x < columns; x++) {
                if(sits[x][y]) { //已卖出用蓝色
                    g.setColor(Color.blue);
                } else {           //未卖出用白色
                    g.setColor(Color.white);
                }
                g.fillRect(x*20, y*20, 16, 16);
                //加上黑色边框
                g.setColor(Color.black);
                g.drawRect(x*20, y*20, 16, 16);
            }
        }
    }

    public boolean[][] getSeats() { //返回座位出售情况
        //复制一份座位出售情况
        int columns = sits.length;
        int rows = sits[0].length;
        boolean[][] newSeats = new boolean[columns][rows];
        for(int y = 0; y < rows; y++) {
            for(int x = 0; x < columns; x++) {
                newSeats[x][y] = sits[x][y];
            }
        }
        return newSeats;
    }

    public void setSeats(boolean[][] aSeats) {
        sits = aSeats;
        repaint();
    }

    public boolean sellSeat(int row, int column) throws Exception { //卖出一个座位
        //public synchronized boolean sellSeat(int row, int column) throws Exception { //卖 出
        //一个座位
        if(sits[column][row]) { //座位已经卖出

```

```

        return false;
    }
    else {
        Thread.sleep(1000);
        sits[column][row] = true;
        repaint();
        return true;
    }
}
}

```

用一台售票机进行买票操作没有问题，问题出在两个用户从两台售票机同时买同一个座位的票时。比如下面的流程，当售票机 1 查询完，用户正在选择座位，还没有提交确认；此时售票机 2 进行了查询，然后用户也选择座位，再提交确认。两个售票机查到的数据是一样的，所以两个用户同时看到一个座位还未卖出，比如 2 排 2 号。这样两个用户可以都选择购买 2 排 2 号，并确认提交成功买到票。但问题是两个人买到了同一个座位的票。

这个问题是因为售票机仅仅根据自己的数据副本来自售造成的一，一台售票机卖出座位后另一台并不知道座位已卖出。正确的方法应该是根据服务器上的数据来售票，即售票机将用户选定的购票排号提交到服务器，由服务器确定是否能够售出该座位。流程修改为如图。

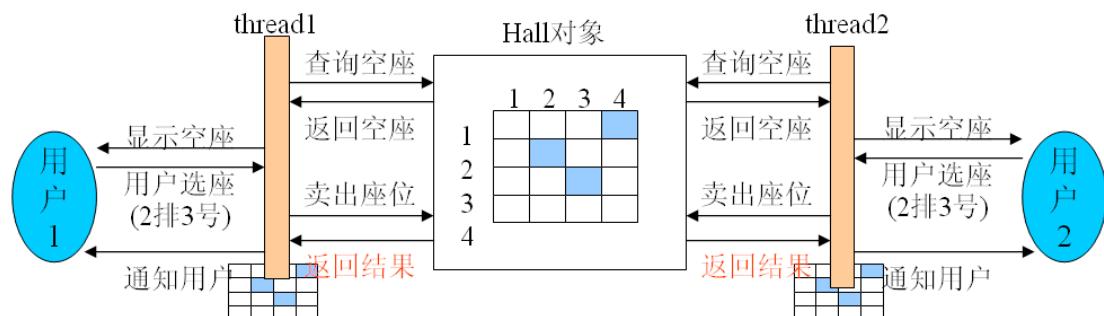


图 16-11 更改后的电影院售票流程

代码做如下修改，将原来例子中 90-91 行改为：

```

if(serverHallPanel.sellSit(row,column)) {//卖票前查询服务器数据
    localHallPanel.sellSit(row,column);
}

```

这样做是否就高枕无忧了呢？回答是否定的，现在才到了关键的地方。服务器上确定是否能够售出一张票的流程是，先用 if 语句判断座位是否售出，如果已经售出，就返回售出失败；如果没有售出就做上售出的标记，并返回售出成功。注意，服务器上两个这样的线程同时在运行，有可能出现这样的情况：一个线程在执行完 if 语句发现座位没有售出，但是在做上售出标记之前发生了线程切换；另一个线程对同一个座位执行 if 语句也发现座位没有售出，并执行后面的售出分支。如图。这样，两个线程还是会售出同一个座位的票。

```

public boolean sellSit(int row, int column)
    ① if(sits[column][row]) return false;
    ② else{ Thread.sleep(1000);
        ⑤ sits[column][row] = true;
        ⑥ repaint();
        ⑦ return true;
    }
}

```



```

public boolean sellSit(int row, int column)
    ③ if(sits[column][row]) return false;
    ④ else{ Thread.sleep(1000);
        ⑧ sits[column][row] = true;
        ⑨ repaint();
        ⑩ return true;
    }
}

```

图 16-12 线程切换造成问题

发生上述情况的概率很小，为了演示这种情况，可以人为的让线程在 `if` 语句后面进行切换，即添加一条 `sleep` 语句，比如让线程睡眠 2 秒。那么，如果在点完一台售票机的“购买”按钮后，2 秒钟之内在另一台售票机购买同一个座位的票，两边都会成功。代码如上图所示。

虽然在这个例子中，如果不人为线程切换的话，发生一票二卖的情况微乎其微，但如果售票线程很多的时候概率将大大增加，比如网上火车票销售系统。而且有时发生后造成的后果会很严重。所以，编写程序时应该完全避免这种情况发生。原理很简单，就是不让线程切换发生在 `if` 语句和做标记的语句之间，具体方法是将这两条语句所在的方法声明成同步方法，或者将这两条语句所在的代码段加上同步标记。

要将该方法声明为同步方法，只需在方法声明时加上 `synchronized` 关键字。代码如下：

```

public synchronized boolean sellSit(int row, int column) throws Exception {//卖出一个座位
    if(sits[column][row]) { //座位已经卖出
        return false;
    }
    else {
        Thread.sleep(1000);
        sits[column][row] = true;
        repaint();
        return true;
    }
}

```

要将一段代码声明为同步的，只需在代码外面加上 `synchronized(this){}` 括起来。

变成同步方法后，同一时刻只能有一个线程执行进入同一个对象的该方法。这里需要强调的是同一个对象，比如两个线程同时执行对象 `a` 的同步方法 `f`，只能有一个进入，另一个须等待；如果 `a` 和 `b` 是同类型的对象，一个线程执行 `a` 的同步方法 `f`，另一个线程执行 `b` 的同步方法 `f`，不需要等待，因为不是同一个对象。

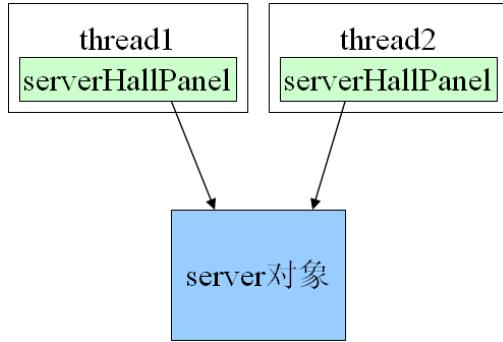


图 16-13 同步方法只对同一对象起作用

同步代码段的规则是一样的，但是可以指定同步对象。同步方法默认使用当前对象作为同步对象，而同步代码段的声明中，`synchronized` 后面的括号内可以指定一个同步对象，用 `this` 的话指方法所在的对象，不是线程对象。

同步机制在 Java 中通过对象锁来实现。当线程调用对象的同步方法时，对象将转为“锁定”状态。此时如果其它线程调用该对象的同步方法，由于对象已经锁定，线程将进入被中断状态，等待对象解锁。同步方法退出后，对象才被解锁。

一个对象可能有多个同步方法，调用任何一个，都会给对象加锁。一个线程可以在同步方法中调用另外一个同步方法，这种情况下，直到退出最外层的同步方法后，对象才解锁。如果在同步方法中发生异常造成退出方法时，仍然会为对象解锁。非同步方法没有这些限制。对象锁机制保证了同步方法操作的完整性：只有一个线程完成操作后，其它线程才能进行操作。

16.8 wait 和 notify 机制

线程同步机制除了同步方法外，还有 `wait` 和 `notify` 机制，线程在执行过程中需要等待某个条件成立，比如一个线程等待另外两个线程的计算结果。此时使用 `wait` 和 `notify` 方法机制，`wait` 和 `notify` 是 `Object` 类的两个方法，即任何对象都有此方法。这两个方法只能在对象的 `synchronized` 方法中使用。当线程需要等待某个条件时调用 `wait` 方法，它会将当前线程放入该对象的等待队列中。而 `notify` 方法则唤醒对象等待队列中的一个线程，在对象属性变化后调用，可以唤醒等待的线程，让其判断继续运行的条件是否满足。若要唤醒等待队列中的所有线程，用 `notifyAll` 方法。如图

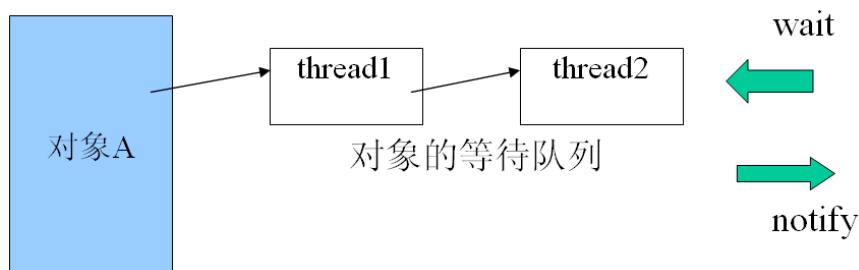


图 16-14 wait/notify 机制

下面的例子演示了 `wait` 和 `notify` 方法的使用。修改线程优先级的例子，让 10 个线程不是同

时启动，而是让线程 2-10 等待线程 1 的进度，线程 2 在线程 1 执行到 10%的时刻启动，线程 2 在 20%时启动，依次类推。执行的结果如图

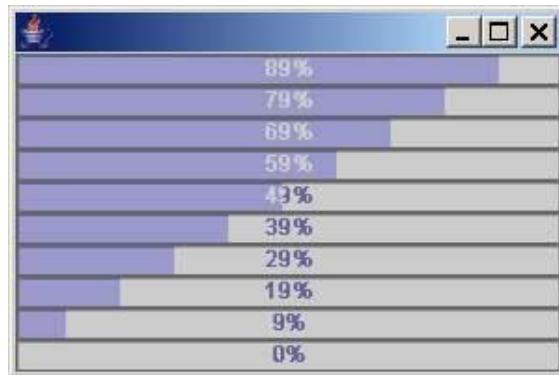


图 16-15 使用 notify 唤醒线程

例子代码如下：

```
// WaitTest.java
import javax.swing.*;
import java.awt.*;

public class WaitTest extends JFrame {
    private JProgressBar[] progBar = new JProgressBar[10];
    private Signal signal = new Signal();
    public static void main(String[] args) {
        JFrame win = new WaitTest();
        win.setVisible(true);
    }
    public WaitTest() {
        setSize(300,200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container contentPane = getContentPane();
        contentPane.setLayout(new GridLayout(10,1));

        for(int i=0; i<10; i++) {
            progBar[i] = new JProgressBar(0,100000);
            progBar[i].setStringPainted(true); //显示中间的文字 ***
            contentPane.add(progBar[i]);
            Thread r = new RunnerThread(i);
            r.start();
        }
    }
    class RunnerThread extends Thread {
        private int index;
        public RunnerThread(int aIndex) {
            index = aIndex;
        }
        public void run() {
            for(int i=0; i<100000; i+=10000) {
                if(index == 0) {
                    signal.signal();
                }
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

```

    }
    public void run() {
        try {
            signal.waitForValue(index);
            for(int i = 0; i <= 100000; i++) {
                EventQueue.invokeAndWait(new Op(progBar[index],i)); // 调用
            }
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}

class Op implements Runnable {
    private JProgressBar prog;
    private int val;
    public Op(JProgressBar p, int v) {
        prog = p;
        val = v;
    }
    public void run() {
        prog.setValue(val);
    }
}

class Signal {
    private int value = 0;
    public synchronized void setValue(int v) {
        if(value != v) {
            value = v;
            notifyAll();
        }
    }
    public synchronized void waitForValue(int v) {
        try {
            while(value < v) wait();
        } catch(Exception e) {

```

```
    }  
}  
}
```

需要注意，有 `wait` 就一定要有 `notify`，使用时要注意，不要造成“死等”。

16.9 死锁

使用同步方法和 `wait/notify` 机制都有可能造成死锁。多个线程同时运行，每个线程占用部分资源，同时等待另外的资源。比如，线程 1 占用对象 `a`，等待对象 `b`；而线程 2 占用对象 `b`，等待对象 `a`；此时谁都无法继续执行，造成死锁。

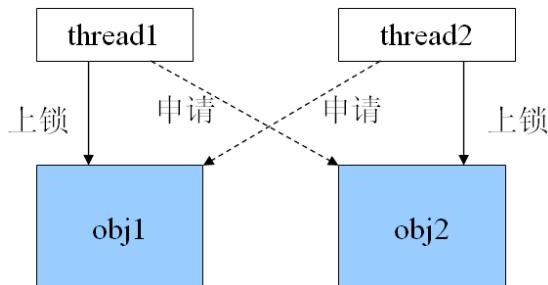


图 16-16 死锁

死锁问题有时很难发现，特别是线程较多且同步机制使用较多时。单个线程需要的资源数量应该不超过系统中资源的总数量，如果某个线程占用所有资源，肯定不会死锁。所以，只从单个线程的角度考虑是无法发现死锁的，只有从全局所有线程的角度出发，才能发现死锁。Java 平台不能自动检测出死锁问题，所以需要程序设计时特别注意。

下面的例子演示了一个同步方法造成的死锁。代码如下：

```
// DeadLock1.java  
public class DeadLock1 {  
    public static void main(String[] args) throws Exception {  
        MyObject obj1 = new MyObject();  
        MyObject obj2 = new MyObject();  
        MyThread thread1 = new MyThread(obj1,obj2); //thread1 在 obj1 的同步方法中去调用 obj2 的同步方法  
        MyThread thread2 = new MyThread(obj2,obj1); //thread2 在 obj2 的同步方法中去调用 obj1 的同步方法  
        thread1.start();  
        thread2.start();  
        System.out.println("Main: Threads started");  
    }  
}  
  
class MyThread extends Thread {  
    private MyObject obj1;
```

```
private MyObject obj2;
public MyThread(MyObject aObj1, MyObject aObj2) {
    obj1 = aObj1;
    obj2 = aObj2;
}
public void run() {
    obj1.operate(obj2);
}
}

class MyObject {
    public synchronized void operate(MyObject other) {
//    public void operate(MyObject other) { //若不使用同步方法则不会死锁
        try {
            if(other != null) {
                Thread.sleep(1000);
                other.operate(null);
            }
        }
        catch(Exception e) {
            e.printStackTrace();
        }
    }
}
}
```

第十七章 网络通信

17.1 Socket 简介

现在互联网上和局域网中传输数据，使用最广泛的是 TCP/IP 协议。通信的两端，每一端都有一个 Socket。Socket API 是使用 TCP/IP 协议进行网络通信程序设计的编程接口。

在 TCP/IP 通信协议中，每台计算机用一个 IP 地址标识，用四个字节，如 210.226.15.38。同一台计算机上有多个程序/进程需要通信时，用端口号进行区分，是一个两字节无符号整数。通信两端，发送方和接收方，每一端都有 IP 和 Port。如图

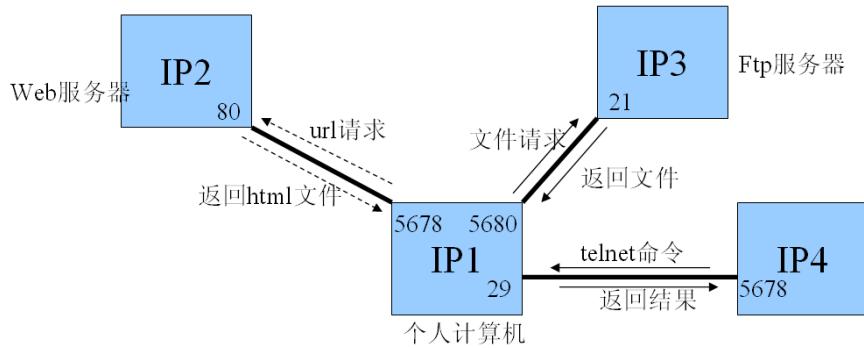


图 17-1 TCP/IP 协议

通信分为两种方式：

- 有连结(TCP): 先在两台计算机间建立一个双向数据流管道，再传输数据。
- 无连结(UDP / Datagram, 数据报): 不用建立数据流管道，将数据组织为一个个的包，以包为单位发送。

TCP 是一种有连结的数据传输协议，传输数据前先建立可靠的连结。连接就像一个管道，数据能够顺着管道流到另一端。数据传输是可靠的，如果传输不到接收方，发送方能够检测到。数据传输是有序的，先发送的数据肯定先到达接收方。因为要建立连结并通过一套协议机制保证数据传输的有效性和可靠性，所以耗费资源比较多。类似于打电话，讲话之前要先拨号并等待对方接听。

UDP 是一种无连结的数据传输协议，不需要事先建立连结。数据传输是不可靠的，数据包一旦发出，就和发送方没有关系了，能否到达接收方不可知。数据传输顺序是不定的，先发送的数据包可能后到达接收方。耗费资源比较少。数据包是有大小限制的，通过交换机上的参数设置，大约四百多个字节。类似于发邮件或短信，发送后就不管了，也不知道对方是否收到。

基于两种通信方式各自的特点，分别应用于不同用途。TCP 方式用于对数据可靠性要求较高的情况，如浏览器使用的 HTTP 协议、文件下载使用的 FTP 等协议。UDP 用于对速度和节省资源要求较高的情况，如多人游戏、QQ 等。

17.2 有连接的 TCP 通信

17.2.1 实现服务器端

Socket 将通信双方抽象为服务器端和客户端，分别用 `ServerSocket` 和 `Socket` 类来描述。如图

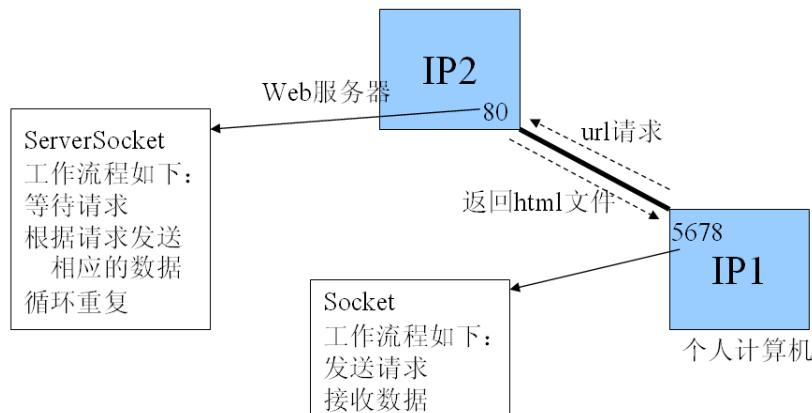


图 17-2 Socket 的工作流程

实现 TCP 服务器端的步骤如下：

1. 创建一个 `ServerSocket` 对象；
2. 调用其 `accept` 方法等待客户端连结，有连接后会自动生成 `Socket` 对象；
3. 有连结后，通过 `Socket` 对象的 `getInputStream` 和 `getOutputStream` 方法获得通信的输入、输出流；
4. 通过两个流进行数据传输。

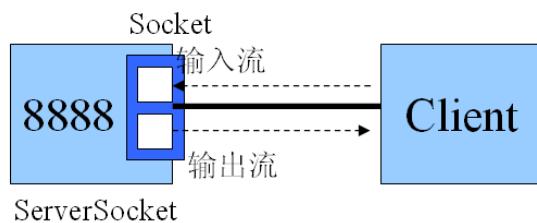


图 17-3 Socket 服务器端

下面的例子实现了一个这样的 TCP 服务器端，代码如下：

```
//Server1.java
import java.io.*;
import java.net.*;
public class Server1 {
    public static void main(String[] args){
        try {
            ServerSocket s = new ServerSocket(8888);
            Socket incoming = s.accept();
            BufferedReader in = new BufferedReader(new InputStreamReader(
                incoming.getInputStream()));
            PrintWriter out = new PrintWriter(incoming.getOutputStream(),true);
            out.println("Welcome to "+s.getInetAddress()+"("+s.getLocalPort()+")");
            System.out.println("Accept "+incoming.getInetAddress()+
                "("+incoming.getPort()+")");
        } catch(IOException e) {
            System.out.println(e);
        }
    }
}
```

```
    }
}
}
```

要测试这个程序，可以使用 Windows 自带的 Telnet 程序。首先运行服务器端程序，如图 A。然后另外启动一个命令行窗口，输入命令：telnet 127.0.0.1 8888。即可看到服务器端的反应，如图 B。Telnet 实际上就是一个 TCP 客户端，以纯文本方式和服务器端通信。命令后面的两个参数分别是服务器端 IP 地址和端口，127.0.0.1 是一个指代本机的专用 IP，因为此处服务器端和客户端运行在同一台计算机上。

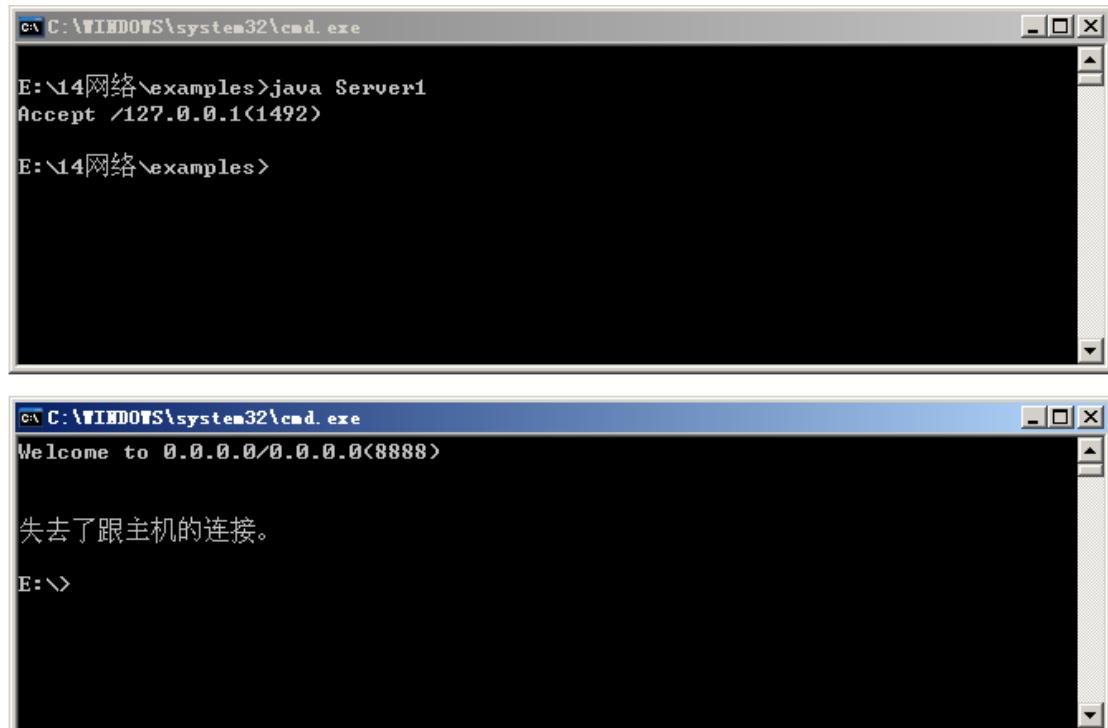


图 17-4 Socket 通信过程

上面的例子中，服务器端只能响应一个客户端的通信请求，而且一个回合就结束退出了。要实现对一个客户端的多个回合的服务，服务器端必须循环处理输入流的内容。要实现对多次连结的服务，服务器端必须循环处理所有建立的连结。这样，就给上面的流程套上一个二重循环。可以采用通信中的某个特定词作为退出命令，如 exit 退出连接，quit 退出程序。代码修改成下面这样：

```
// Server2.java
import java.io.*;
import java.net.*;
public class Server2 {
    public static void main(String[] args){
        try {
            ServerSocket s = new ServerSocket(8888);
            boolean flag = true;
            while(flag){ //循环等待建立连结
                Socket incoming = s.accept(); //直到有一个连结建立， accept 返回
                ...
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```

        System.out.println("Accept
"+incoming.getInetAddress()+"("+incoming.getPort()+")");
        BufferedReader in = new BufferedReader(new
InputStreamReader(incoming.getInputStream()));
        PrintWriter out = new PrintWriter(incoming.getOutputStream(),true);
        out.println("Welcome to "+s.getInetAddress()+"("+s.getLocalPort()+")");

        String line;
        while(true){ //循环处理客户端发来的数据
            line = in.readLine(); //等待读入一行
            if(line != null) {
                System.out.println(line);
                if(line.equals("quit")){
                    //quit=服务器端结束，退出程序
                    flag = false;
                    break;
                }
                if(line.equals("exit")) //exit=一个连结结束，继续等待下一个连结
                    break;
            }
        }
        incoming.close();
    }
}
catch(IOException e){
    System.out.println(e);
}
}
}

```

这个服务器端程序可以和客户端进行多个回合的通信，并且可以依次和多个客户端建立连接，但是不能同时和多个客户端通信。如果正在和一个客户端通信，而另一个客户端发起了建立连接的请求（再打开一个命令行窗口并执行 telnet），只能等第一个客户端退出后才能建立连接。因为服务器端这时才会执行到 accept 方法来和客户端建立连接。要想同时和多个客户端建立连接，就必须使用多线程让 accept 方法随时都可以执行。

17.2.2 采用多线程

服务器端采用多线程技术可以实现同时与多个客户端建立连接。一个线程主要负责循环执行 accept 方法，等待客户端连接。一旦一个连接建立后，该线程就启动一个新的线程处理该连接的通信。这样，每个与客户端的连接都有一个处理线程，各线程独立并发运行，就实现了和多个客户端同时通信。代码如下：

```

// Server3.java
import java.io.*;
import java.net.*;
public class Server3{
    public static void main(String[] args){
        Thread listeningThread = new ListeningThread(8888);
        listeningThread.start();
    }
}
class ListeningThread extends Thread{
    private int port;
    private boolean flag = true;
    private ServerSocket lServerSocket;
    public ListeningThread(int aPort) {
        port = aPort;
    }
    public void run(){
        try{
            lServerSocket = new ServerSocket(port);
            System.out.println("Start listening.....");
            while(flag){
                Socket incoming = lServerSocket.accept();
                System.out.println("Accept
"+incoming.getInetAddress()+"("+incoming.getPort()+")");
                Thread t = new ServiceThread(incoming);
                t.start();
            }
            lServerSocket.close();
            System.exit(0);
        }
        catch(IOException e){
            System.out.println(e);
        }
    }
}
class ServiceThread extends Thread{
    private Socket lSocket;
    public ServiceThread(Socket aSocket){
        lSocket = aSocket;
    }
    public void run(){
        try{
            BufferedReader in = new BufferedReader(new
InputStreamReader(lSocket.getInputStream()));
            PrintWriter out = new PrintWriter(lSocket.getOutputStream(),true);

```

```
        out.println("Welcome to
"+lServerSocket.getInetAddress()+"("+lServerSocket.getLocalPort()+")");
        String line;
        while(true){
            line = in.readLine();
            if(line!=null){
                System.out.println(line);
                if(line.equals("quit")){
                    flag = false;
                    break;
                }
                if(line.equals("exit"))    break;
            }
        }
        lSocket.close();
    }
    catch(IOException e){
        System.out.println(e);
    }
}
}
```

要测试和多个客户端同时通信，只需再启动几个命令行窗口，并用 telnet 命令和服务器端建立连接。如图

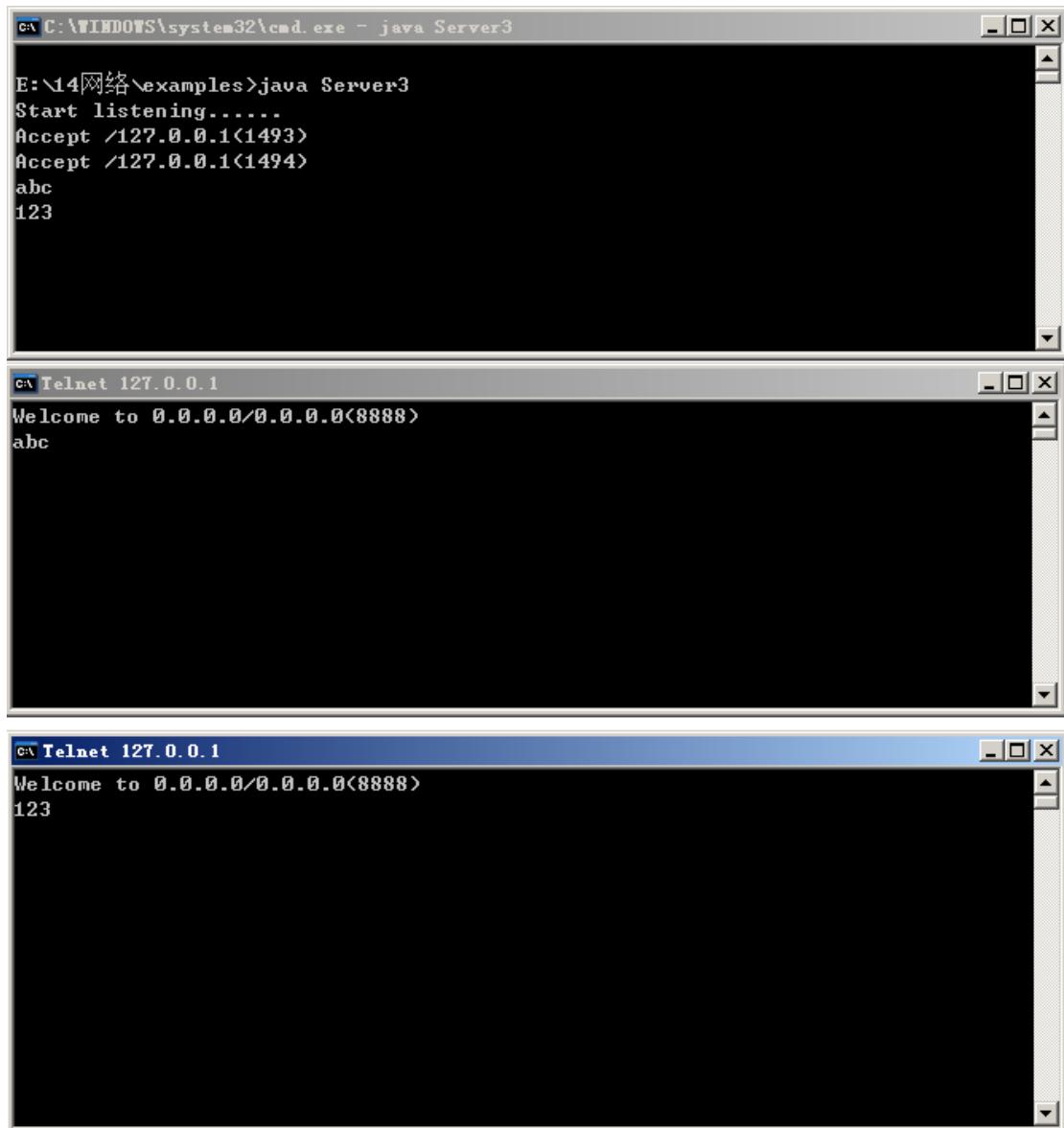


图 17-5 服务器端用多线程同时和多个客户端通信

与前面的例子类似，也是用 `exit` 退出当前连接，用 `quit` 退出程序。但是实际上，在客户端输入 `quit` 并发送后，服务器并不能马上退出，而是要等到下一个客户端连接到来时才退出。这是因为服务器端的线程在等待 `accept` 方法返回，只有一个客户端连接到来时，`accept` 方法才返回，从而让程序能执行到 `while` 循环的条件判断部分，并退出程序。

解决 `accept` 方法死等的办法是为 `ServerSocket` 设置超时参数。有了超时设置，`accept` 方法在等待一段时间后，如果没有客户端连接到来，也会退出，但是不是正常退出，而是异常退出，异常类是 `SocketTimeoutException`。

加入超时设置后，监听线程的代码如下：

```
public void run(){
    try{
        IServerSocket = new ServerSocket(port);
```

```

I ServerSocket.setSoTimeout(1000);
System.out.println("Start listening.....");
while(flag){
    try{
        Socket incoming = I ServerSocket.accept();
        System.out.println("Accept
"+incoming.getInetAddress()+"("+incoming.getPort()+"')");
        Thread t = new ServiceThread(incoming);
        t.start();
    }
    catch(SocketTimeoutException e){
        if(!flag)break;
    }
}
I ServerSocket.close();
System.exit(0);
}
catch(IOException e){
    System.out.println(e);
}
}

```

至此，服务器端程序基本完善了。

17.2.3 实现客户端

有了服务器端以后，接下来编写客户端程序。实现 TCP 客户端的步骤如下：

1. 根据服务器 ip 和 port 创建一个 Socket 对象，创建 Socket 对象的过程就是建立连接的过程；
2. 建立连结后，通过 Socket 对象获得输入、输出流；
3. 通过两个流进行数据传输。

下面的例子实现了一个 TCP 客户端，代码如下：

```

// Client1.java
import java.io.*;
import java.net.*;
public class Client1 {
    public static void main(String[] args){
        try {
            //建立 Socket 对象
            Socket client = new Socket("127.0.0.1",8888);
            //建立网络输入输出流

```

```

        BufferedReader in = new BufferedReader(new
InputStreamReader(client.getInputStream()));
        PrintWriter out = new PrintWriter(client.getOutputStream());

        //建立键盘读入流
        BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
        boolean flag = true;
        while(flag) {
            System.out.println(in.readLine());
            String str = br.readLine();
            out.println(str);
            out.flush();
            if(str!=null && str.equals("quit")) flag = false;
        }
        client.close();
    }
    catch(Exception e) {
        System.out.println(e);
    }
}
}

```

此客户端可以连接上一节中编写的服务器端进行测试。先用一个命令行窗口启动服务器端，再用另一个命令行窗口运行客户端，可以在多个命令行窗口中启动多个客户端。

在前面的例子中，客户端和服务器端通信必须采用一问一答的方式，不够灵活。如果服务器主动给客户端发送一条信息，则客户端无法及时收到。如果客户端需要读入数据而服务器没有发送，则客户端死等。这是因为发送数据和接收数据都在一个线程中进行。如果需要，不管是服务器端还是客户端，都可以将发送数据和接收数据分在两个线程中。

例子中没有处理各种网络通信的异常。可能发生的异常情况包括：

1. 服务器端程序整个退出，客户端再发送信息时抛出 `java.net.SocketException`；
2. 服务器端服务线程退出，客户端再发送信息时抛出 `java.net.SocketException`；
3. 客户端退出，服务器端立即抛出 `java.net.SocketException`；
4. 客户端在连结服务器时如果服务器未启动，则抛出 `java.net.ConnectionException`。

本节中编写的客户端也无法马上检测到服务器端的退出，这是因为客户端用 `PrintWriter` 来操作网络输出流，该类的对象检测不到连接中断的异常。如果改用 `OutputStream` 等能够检测到连接异常的类来操作网络输出，则能够马上检测到服务器端退出等情况。

17.3 简易聊天室

在前面的例子的基础上，很容易实现一个简单的聊天室系统。聊天室的功能特点是：服务器从一个客户端接收的信息，要转发到所有客户端。为实现这一功能，聊天服务器端应该用一个列表结构记下所有的客户端连接，这样，接收线程就能够将接收到的信息发往列表中保存的所有客户端连接。客户端也有两个线程，一个接收线程负责从服务器接收数据，另一个主线程接收用户的键盘输入并发往服务器。如图

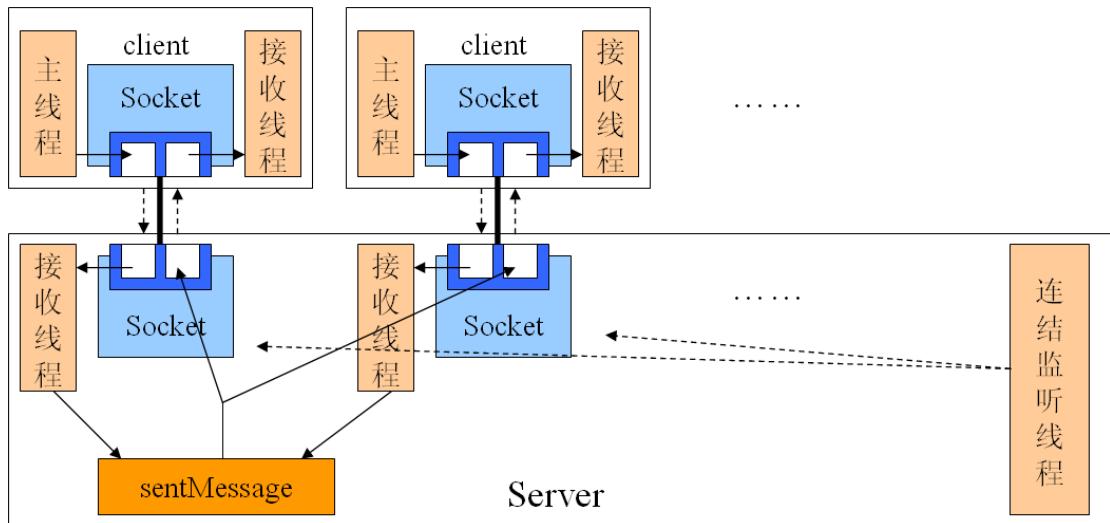


图 17-6 聊天室的结构

服务器端代码如下：

```
// ChatServer.java
import java.io.*;
import java.net.*;
import java.util.*;
public class ChatServer {
    public static void main(String[] args){
        Thread listeningThread = new ListeningThread(8888);
        listeningThread.start();
    }
}
class ListeningThread extends Thread {
    private int port;
    private boolean flag = true;
    private ServerSocket lServerSocket;
    private List<PrintWriter> connections = new Vector<PrintWriter>(); //保存所有连结
    public ListeningThread(int aPort){
        port = aPort;
    }
    public void run(){
        try {
```

```

IServerSocket = new ServerSocket(port);
IServerSocket.setSoTimeout(1000);
System.out.println("Start listening.....");
while(flag) {
    try {
        Socket incoming = IServerSocket.accept();
        System.out.println("Accept
"+incoming.getInetAddress()+"("+incoming.getPort()+")");
        PrintWriter out = new
PrintWriter(incoming.getOutputStream(),true);
        connections.add(out); //有新连结则将其输出流添加到连结列
表中
        out.println("Welcome to
"+IServerSocket.getInetAddress()+"("+IServerSocket.getLocalPort()+")");
        out.flush();
        Thread t = new ServiceThread(incoming); //启动接收数据线程
        t.start();
    }
    catch(SocketTimeoutException e) {
        if(!flag)break;
    }
}
IServerSocket.close();
System.exit(0);
}
catch(IOException e) {
    System.out.println(e);
}
}

public synchronized void chatMessage(String msg) {
    Iterator<PrintWriter> iter = connections.iterator();
    while(iter.hasNext()) {
        try {
            PrintWriter out = iter.next();
            out.println(msg);
            out.flush();
        }
        catch(Exception e) {
            iter.remove(); //如果发送中出现异常，则将连结移除
        }
    }
}
class ServiceThread extends Thread {
    private Socket ISocket;

```

```

public ServiceThread(Socket aSocket){
    ISocket = aSocket;
}
public void run(){
    try {
        BufferedReader      in      =      new      BufferedReader(new
InputStreamReader(ISocket.getInputStream()));
        String line;
        while(flag){
            line = in.readLine();
            if(line != null){
                System.out.println(line);
                chatMessage("Chat:"+line);
                if(line.equals("quit")){
                    flag = false; //退出整个程序
                    break;
                }
                if(line.equals("exit")) {
                    break; //只退出本线程
                }
            }
        }
        ISocket.close();
        System.out.println("Thread stoped.");
    }
    catch(IOException e){
        System.out.println(e);
    }
}
}
}

```

客户端代码如下：

```

//ChatClient.java
import java.io.*;
import java.net.*;
public class ChatClient {
    public static void main(String[] args) {
        try {
            Socket client = new Socket("127.0.0.1",8888);

            //建立数据接收线程
            Thread t = new ServiceThread(client);
            t.start();
        }
    }
}

```

```

//建立网络输出流
PrintWriter out = new PrintWriter(client.getOutputStream());
//建立键盘读入流
BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));

//开始键盘读取输入的循环
while(true) {
    String str = br.readLine();
    if(str != null) {
        out.println(str);
        out.flush();
        if(str.equals("close")) break;
    }
}
client.close();
}

catch(Exception e) {
    System.out.println(e);
}
System.exit(0);
}

}

class ServiceThread extends Thread {
private Socket lSocket;
public ServiceThread(Socket aSocket) {
    lSocket = aSocket;
}
public void run() {
try {
    BufferedReader in = new BufferedReader(new
InputStreamReader(lSocket.getInputStream()));
    String line;
    while(true) {
        line = in.readLine();
        if(line != null) {
            System.out.println(line);
            if(line.equals("chat:close"))break;
            if(line.equals("chat:quit"))break;
        }
    }
    lSocket.close();
}
}

```

```
        catch(IOException e) {
            System.out.println(e);
            System.exit(0);
        }
    }
}
```

运行后，如图。在任何一个客户端窗口中输入的字符串将被服务器转发到所有客户端窗口。

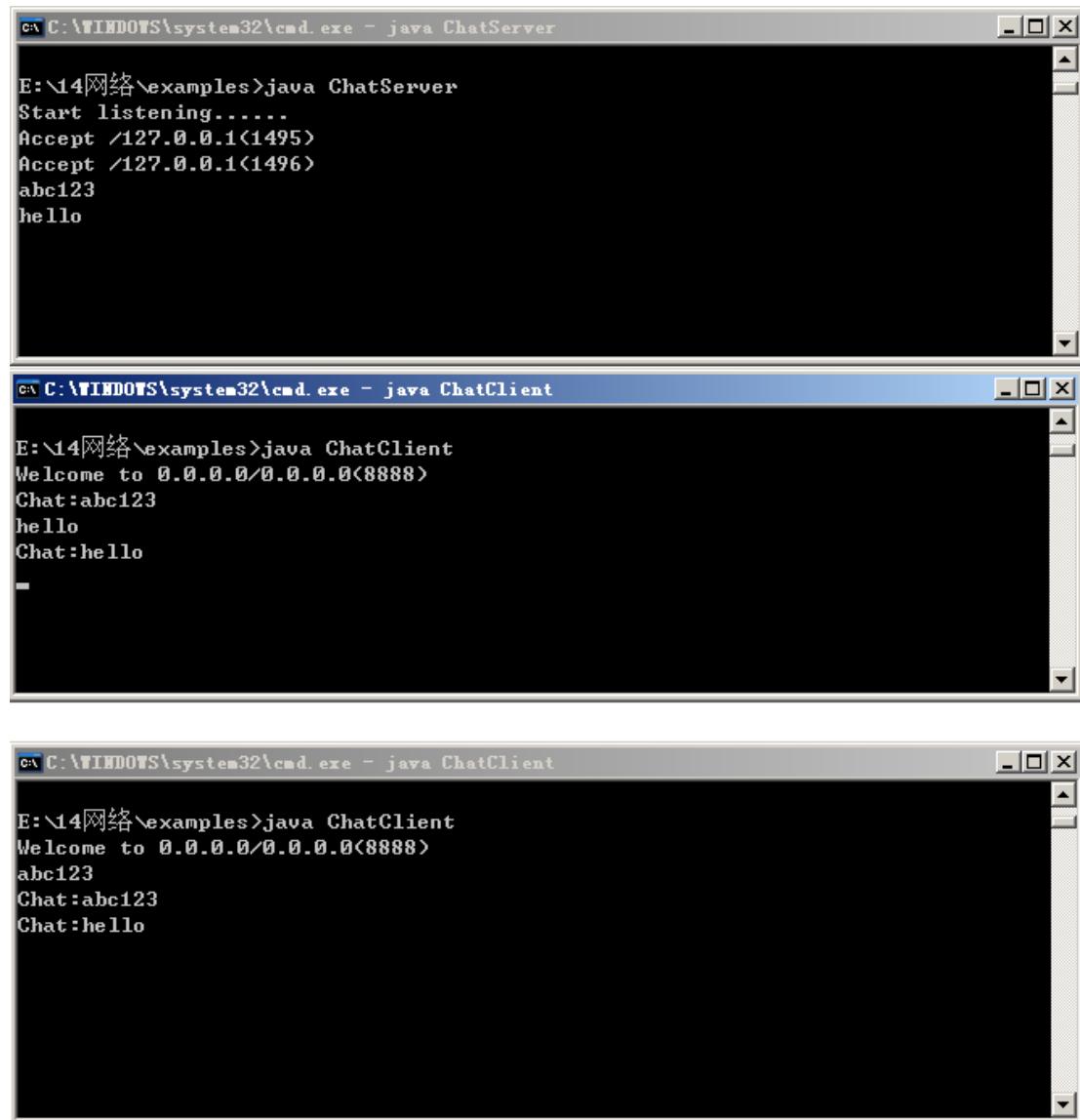


图 17-7 聊天过程

17.4 无连接的 UDP 通信

17.4.1 实现服务器端

UDP 通信服务器端使用 DatagramSocket 类的 receive 方法来接收 UDP 包。实现 Datagram 接收方的步骤如下：

1. 创建一个 DatagramSocket 对象，需指出监听的端口；
2. 创建一个空的 DatagramPacket，作为参数调用 receive 方法等待数据包到达；
3. 有数据包到达后，可以从数据包得到发送方的 ip 和 port，以及包中的数据，从而进行处理。

创建空的 DatagramPacket 时，需指定一块缓冲区用户存放数据，可以用一个 byte 型数组。
代码如下：

```
// DatagramServer.java
import java.io.*;
import java.net.*;
import java.util.*;
public class DatagramServer {
    public static void main(String[] args) throws IOException {
        new DatagramServerThread().start();
    }
}
class DatagramServerThread extends Thread {
    protected int number = 0; //引用计数变量
    public void run(){
        try {
            DatagramSocket socket = new DatagramSocket(8888);
            while(true){
                //建立一个空的数据包，准备接收 UDP 包
                byte[] buf = new byte[256];
                DatagramPacket packet = new DatagramPacket(buf, buf.length);
                socket.receive(packet);
                //收到包后，将其数据转化为字符串输出
                String s = new String(packet.getData()).trim();
                System.out.println(packet.getAddress()+"."+packet.getPort()+"."+s);
                if(s.equals("exit"))break;
            }
            socket.close();
        }
        catch (IOException e) {
            System.out.println(e);
        }
    }
}
```

```
    }  
}
```

服务器端编写好后，需要再编写客户端，两个一起测试。

17.4.2 实现客户端

实现 Datagram 发送方的步骤如下：

1. 创建一个 DatagramSocket 对象；
2. 创建一个 DatagramPacket，指定接收方 IP，Port 和数据；
3. 调用 DatagramSocket 对象的 send 方法发送。

所有信息都保存在数据包 DatagramPacket 中，所以创建时应给出这些参数。接收方 IP 封装在一个 InetAddress 对象中，Port 是一个整数，数据要保存在一个 byte 型数组中。

下面的例子实现了一个发送方，不断从键盘读取数据，打包发送到服务器端。代码如下：

```
// DatagramClient.java  
import java.io.*;  
import java.net.*;  
import java.util.*;  
public class DatagramClient {  
    public static void main(String[] args) throws IOException{  
        //建立 DatagramSocket  
        DatagramSocket socket = new DatagramSocket();  
        //建立键盘读入流  
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));  
        while(true){  //键盘输入循环  
            //读取一行输入  
            String str = br.readLine();  
            //建立 UDP 包  
            byte[] buf = str.getBytes();  
            InetAddress address = InetAddress.getByName("127.0.0.1");  
            DatagramPacket packet = new DatagramPacket(buf, buf.length, address,  
8888);  
            //发送  
            socket.send(packet);  
            if(str.equals("close"))break;  
        }  
        socket.close();  
    }  
}
```

客户端编好后，和服务器端一起测试。先在一个命令行窗口中启动服务器端，再在另一个命令行窗口中运行客户端。如图

The image shows two separate command-line windows. The top window, titled 'C:\WINDOWS\system32\cmd.exe - java DatagramServer', displays the command 'java DatagramServer' followed by two lines of output: '/127.0.0.1,1497,abcd' and '/127.0.0.1,1497,1234'. The bottom window, titled 'C:\WINDOWS\system32\cmd.exe - java DatagramClient', displays the command 'java DatagramClient' followed by two lines of output: 'abcd' and '1234'.

图 17-8 客户端

从程序结构上来看，TCP 协议可以在一台计算机的一个端口上可以建立多个连结，而且需要为每个连结建立一个线程。UDP 协议没有连结的概念，一台计算机的一个端口上只需建立一个线程就可接收所有数据。如图

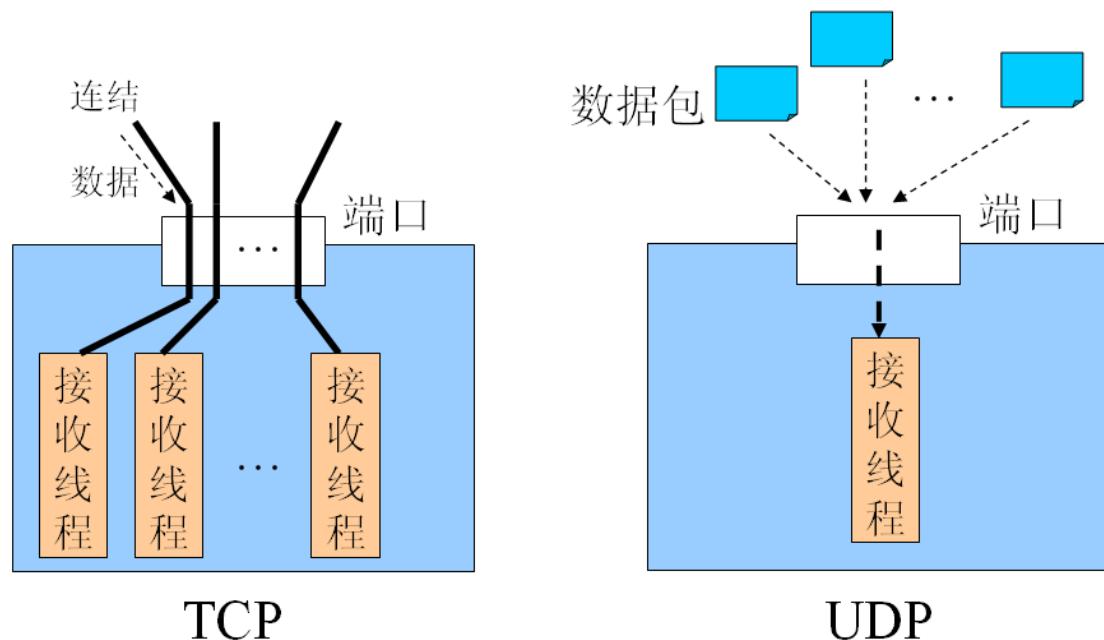


图 17-9 TCP 和 UDP 方式对比

17.5 MVC 模式

编写网络多用户程序最常用的设计模式是 MVC 模式。MVC 代表 Model-View-Controller，即模型-视图-控制器模式。模型包含核心功能和数据，视图向用户显示信息，控制器处理用户输入。其主要特点是：从外部来看，多个控制器共同控制一个模型，多个视图共同显示一个模型；从内部来看，实际上是每个客户端都有一个模型，各客户端之间通过变更传播机制维持模型的一致。



图 17-10 MVC 模式

MVC 的主要思想是：开始时每个客户端都建立一个初始状态完全相同的模型，后续对模型的每一点改动都要所有客户端上的模型同时进行，这样各个模型就整齐划一了。这种变更传播机制的好处是，在模型的运行过程中，每一轮只用传输改动命令即可，不用传输整个模型的数据，可以用很小的通信流量维持很大数据量的模型的一致。

变更传播机制如图，步骤如下：

1. 各模型开始状态完全相同；客户端根据自己的模型更新视图；
2. 各客户端对模型的控制不是直接对自己的模型进行，而是将变更操作发送到服务器；
3. 服务器每隔一定时间将收集到的操作请求同时发送到各个客户端；
4. 客户端收到服务器的变更操作后，对自己的模型进行更新，更新完成后向服务器发送确认消息。
5. 服务器收到所有客户端的确认消息后，转第 2 步开始下一个循环。
6. 这样各客户端上模型的变更操作序列也完全相同，所以始终保持一致。
7. MVC 模式在网络间传输的是模型的变更操作，优点是数据量小。

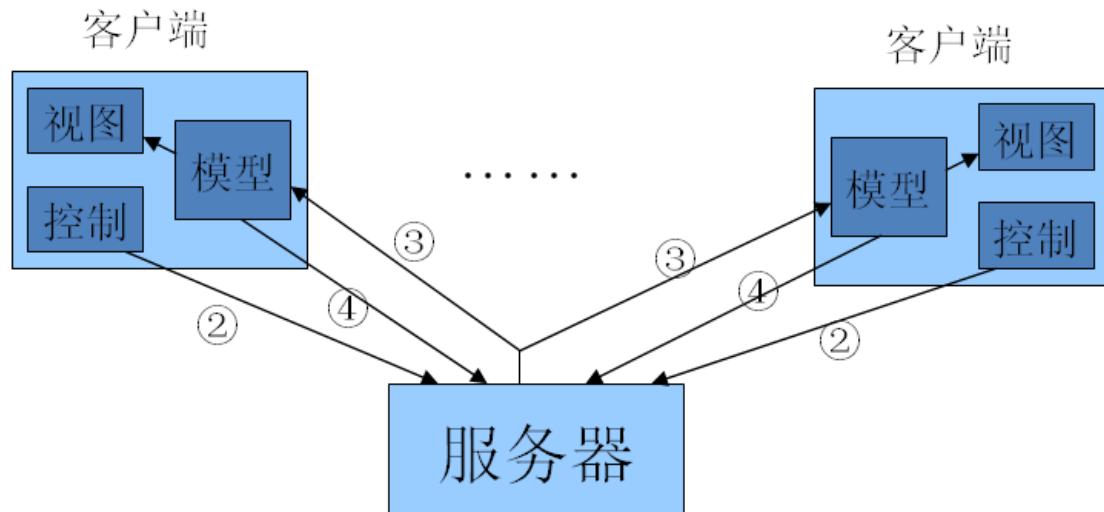


图 17-11 MVC 模式中的变更传播机制

聊天室系统也可以采用 MVC 模型。所有用户输入的字符历史记录构成了模型，一个用户输入的一行字符串就是模型的更新命令。传输更新命令比传输整个模型需要的网络流量要小得多。

第十八章 对象序列化

18.1 格式化数据输入输出

18.1.1 基本数据类型数据的输入输出

本书前面章节中介绍过抽奖器的例子，使用输入输出流和文件进行数据交换，采用的是文本格式。这种方式适用于数据主要是字符串型、用户希望能够方便的直接查看和修改数据文件的情况。

如果数据主要是数值型，再采用字符流输入输出，那就需要进行数值型和字符串型之间的转换。若数据量很大，则会花费很多时间。这种情况下，可以改用数据流来输入输出，以数据在内存中的二进制表示形式进行传输，从而省去转换的时间。但是，这种格式用户不方便直接查看和修改数据文件。

数据输入流和输出流分别对应 `DataInputStream` 和 `DataOutputStream` 两个类。这两个类对于 8 种基本数据类型，分别使用 8 个不同的方法操作，`DataInputStream` 提供 `read` 方法从输入流读取数据，而 `DataOutputStream` 提供 `write` 方法向输出流写入数据。使用数据流向文件中保存和读取数据时，保存和读取的格式要一一对应。

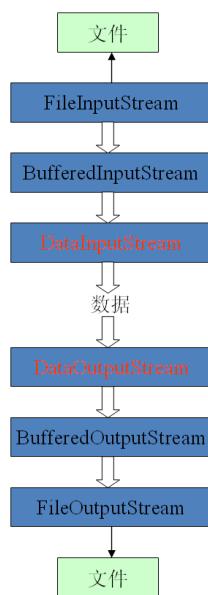


图 18-1 基本数据类型的输入输出

下面是一个用数据流向文件中保存数据和从文件中读取数据的例子。Account 类有两个字段，分别是 long 型和 double 型。要将一个 Account 对象的数据保存到文件，需要分别将这两个字段的数据写入数据流。要再从文件中将数据读出来，需要分别读出这两种格式的数据。例子代码如下：

```
// Account.java
import java.io.*;
public class Account {
    private long id;
    private double value;
    public Account() {
        id = 0;
        value = 0;
    }
    public Account(long aID, double aValue) {
        id = aID;
        value = aValue;
    }
    public String toString() {
        return String.valueOf(id) + " " + String.valueOf(value);
    }
    public void saveToData(DataOutputStream out) throws IOException {
        out.writeLong(id);
        out.writeDouble(value);
    }
    public void loadData(DataInputStream in) throws IOException {
        id = in.readLong();
        value = in.readDouble();
    }
    public void saveToText(BufferedOutputStream out) throws IOException {
        String line = String.valueOf(id) + " " + String.valueOf(value) + "\r\n";
        out.write(line.getBytes());
    }
}
// DataSave.java
import java.io.*;
import java.util.*;
public class DataSave {
    public static void main(String[] args) throws IOException {
        int flag = 0;
        if(args.length > 0) {
            flag = Integer.parseInt(args[0]);
        }
        Random rand = new Random();
```

```

FileOutputStream  fout = new FileOutputStream("data.dat");
BufferedOutputStream  bout = new BufferedOutputStream(fout);
DataOutputStream dout = new DataOutputStream(bout);
for(int i = 0; i < 10; i++) {
    long id = i;
    double value = rand.nextDouble();
    Account account = new Account(id, value);
    switch(flag) {
        case 0:  account.saveToData(dout);
        break;
        case 1:  account.saveToText(bout);
        break;
    }
}
dout.close();
bout.close();
fout.close();
}

// DataRead.java
import java.io.*;
import java.util.*;
public class DataRead {
    public static void main(String[] args) throws IOException {
        FileInputStream  fin = new FileInputStream("data.dat");
        BufferedInputStream bin = new BufferedInputStream(fin);
        DataInputStream din = new DataInputStream(bin);
        for(int i = 0; i < 10; i++) {
            Account account = new Account();
            account.loadData(din);
            System.out.println(account);
        }
        din.close();
        bin.close();
        fin.close();
    }
}

```

运行时，先运行 `DataSave`，会将一个 `Account` 对象的数据通过数据流写到文件中。再运行 `DataRead`，会将文件中的数据通过数据流读出，并重建 `Account` 对象。运行结果如图：

```
C:\WINDOWS\system32\cmd.exe
E:\15对象序列化\examples>javac DataSave.java
E:\15对象序列化\examples>java DataSave
E:\15对象序列化\examples>javac DataRead.java
E:\15对象序列化\examples>java DataRead
0 0.6940166180487355
1 0.6510228853161715
2 0.03331419058706342
3 0.427548957153119
4 0.7248740616721315
5 0.5093812028561263
6 0.7504105492879559
7 0.8829584519897273
8 0.5073487743994886
9 0.7244631226002133
E:\15对象序列化\examples>
```

图 18-2 基本数据类型的输入输出

8 种基本数据类型的数据都可以用数据流进行输入输出。采用数据流输入输出比转化成文本再输入输出速度要快，而且数据文件要小。修改一下上面的例子，随机创建 1000000 个 Account，分别用数据流和文本流的方式保存到文件，会发现使用数据流比文本流速度快且文件小。

18.1.2 对象的输入输出

程序中经常需要将一个对象的数据完整的保存到文件中，如果用数据流，需要将对象的每个字段都保存，并且读出时要也要对应读出。若对象数据字段非常多，则比较繁琐，且容易出错。为此，Java 提供了对象输入输出流，可以直接将对象写入对象输出流或从对象输入流中读取对象。对象输入输出流对应的类是 ObjectInputStream 和 ObjectOutputStream。

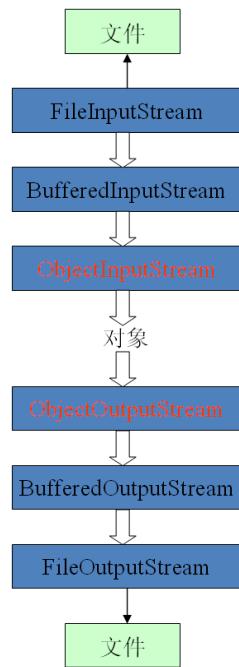


图 18-3 对象的输入输出

要写入对象流的对象必须实现 `Serializable`（序列化）接口，但该接口不需要实现任何方法，只是一个表明对象可以序列化的标识。之所以不是默认所有对象都可以序列化，是出于安全方面的考虑。将对象保存到文件时，创建 `ObjectOutputStream`，调用 `writeObject` 方法。将对象从文件读入时，创建 `ObjectInputStream`，调用 `readObject` 方法；读入返回的是对象的 `Object` 类型的引用，需要通过造型回复对象的原有类型。

下面的例子用对象流将对象保存到文件再读出。代码如下：

```

// ObjectSave.java
import java.io.*;
import java.util.*;
public class ObjectSave {
    public static void main(String[] args) throws IOException {
        Random rand = new Random();
        FileOutputStream  fout = new FileOutputStream("data.dat");
        BufferedOutputStream  bout = new BufferedOutputStream(fout);
        ObjectOutputStream oout = new ObjectOutputStream(bout);
        for(int i = 0; i < 10; i++) {
            long id = i;
            double value = rand.nextDouble();
            Account account = new Account(id, value);
            oout.writeObject(account);
        }
        oout.close();
        bout.close();
        fout.close();
    }
}

```

```

}

class Account implements Serializable {
    private long id;
    private double value;
    public Account() {
        id = 0;
        value = 0;
    }
    public Account(long aID, double aValue) {
        id = aID;
        value = aValue;
    }
    public String toString() {
        return String.valueOf(id) + " " + String.valueOf(value);
    }
}

// ObjectRead.java
import java.io.*;
import java.util.*;
public class ObjectRead {
    public static void main(String[] args) throws Exception {
        FileInputStream fin = new FileInputStream("data.dat");
        BufferedInputStream bin = new BufferedInputStream(fin);
        ObjectInputStream oin = new ObjectInputStream(bin);
        for(int i = 0; i < 10; i++) {
            Account account = (Account)(oin.readObject());
            System.out.println(account);
        }
        oin.close();
        bin.close();
        fin.close();
    }
}

```

18.2 对象序列化

18.2.1 对象序列化机制

在向对象流中写入对象时，是先将对象转化为了一块 byte 序列的数据，再将数据依次写入流中。这个过程中，对象状态转化为 byte 序列，所以称为对象序列化。读出时则是一个相反的过程，称为反序列化。

对象序列化并不是简单的将对象成员依次按基本数据类型保存，而是使用一种比较完善复杂的特殊格式来存储对象数据。存储格式中包括标志、版本号、序列号、类标识符、校验特征码等。对象序列化机制完全由 Java 自动完成，并且提供向上版本的兼容性。

在使用对象序列化机制时，需要说明的是：

- 一个流中可以写入或读出多个对象，不同类的对象也可以保存到一个流中或从流中读出。
- 对象的静态成员（static）和临时成员（transient）不会被保存。
- 对象之间的相互引用关系也会保存，但进入对象流的所有对象都必须实现 Serializable 接口。
- 从接口的继承关系上来看，ObjectInput 和 ObjectOutput 分别是 DataInput 和 DataOutput 的子接口，所以 ObjectOutputStream 和 ObjectInputStream 也可以直接存取原始数据类型。
- 序列化机制使得程序编写上简单，但速度相对来说比直接保存数据慢。
- 可以为对象定义四个方法，来控制序列化的过程：readObject, writeObject, readExternal, writeExternal。

后面的几节，将通过一些例子来说明对象序列化机制的上述特点。

18.2.2 对象与基本数据类型混合保存

使用对象流，可以将多个不同类型的对象、以及基本数据类型的数据混合保存到一个文件中，并且能再正确的读出。

下面的例子先向对象流中写入一个整数 n，然后随机生成 n 个对象写入对象流，这 n 个对象或者是 Human 类型或者是 Computer 类型，是随机的。读出时，先读出整数 n，再读出 n 个对象，并还原每个对象引用的类型。代码如下：

```
// MultiObjectSave.java
import java.io.*;
import java.util.*;
public class MultiObjectSave {
    public static void main(String[] args) throws IOException {
        Random rand = new Random();
        FileOutputStream   fout = new FileOutputStream("data.dat");
        BufferedOutputStream   bout = new BufferedOutputStream(fout);
        ObjectOutputStream oout = new ObjectOutputStream(bout);
        //oout.writeInt(10);
        for(int i = 0; i < 10; i++) {
            if(rand.nextInt()<5) {
                Human h = new Human(i, 'M');
                oout.writeObject(h);
            }
        }
    }
}
```

```

        }
        else {
            Computer c = new Computer(1.0F,i);
            oout.writeObject(c);
        }
    }
    oout.close();
    bout.close();
    fout.close();
}
}

class Human implements Serializable {
    private int age;
    private char gender;
    public Human(int aAge, char aGender) {
        age = aAge;
        gender = aGender;
    }
    public String toString() {
        return "Human[age=" + age + ", gender=" + gender + "]";
    }
}

class Computer implements Serializable {
    private float cpuSpeed;
    private int memSize;
    public Computer(float aCpuSpeed, int aMemSize) {
        cpuSpeed = aCpuSpeed;
        memSize = aMemSize;
    }
    public String toString() {
        return "Computer[cpuSpeed=" + cpuSpeed + ", memSize=" + memSize + "]";
    }
}

// MultiObjectRead.java
import java.io.*;
import java.util.*;

public class MultiObjectRead {
    public static void main(String[] args) throws Exception {
        FileInputStream fin = new FileInputStream("data.dat");
        BufferedInputStream bin = new BufferedInputStream(fin);
        ObjectInputStream oin = new ObjectInputStream(bin);
        //int n = oin.readInt();
        int n = 10;

```

```

        for(int i=0; i<n; i++) {
            Object o = oin.readObject();
            System.out.println(o);
        }
        oin.close();
        bin.close();
        fin.close();
    }
}

```

18.2.3 对象间引用关系的保存

对象间引用关系保存起来比较复杂。例如，一个 `Human` 对象的成员中有对另一个 `Computer` 对象的引用，那么引用中保存的是 `Computer` 对象的内存地址。那么，将两个对象保存到文件后，下次读出时，无法保证 `Computer` 被还原到同一内存地址；如果 `Human` 对象中的引用还使用原来的内容，将指向错误的地址。

`Java` 序列化机制也自动解决对象间引用关系保存的问题。解决的原理是：

- 为每个对象分配一个唯一的序列号；
- 对象中有引用字段指向其他对象，就在引用的位置保存该对象的序列号。
- 对象写入对象流之前，先检查同一对象是否已经写入过，如果没有再将对象写入。

上面实际上就是序列化机制的实现原理，而且这个过程是递归的。就是说对象 `a` 引用对象 `b`（如图 a），那么写入对象 `a` 时，自动将 `b` 也写入。如果 `b` 又引用了其它对象，也会自动写入。如果对象 `a` 和 `b` 都引用了对象 `c`（如图 b），那么写入 `a` 和 `b` 时，对象 `c` 也自动被写入，而且只写入一次，不会写入两次。

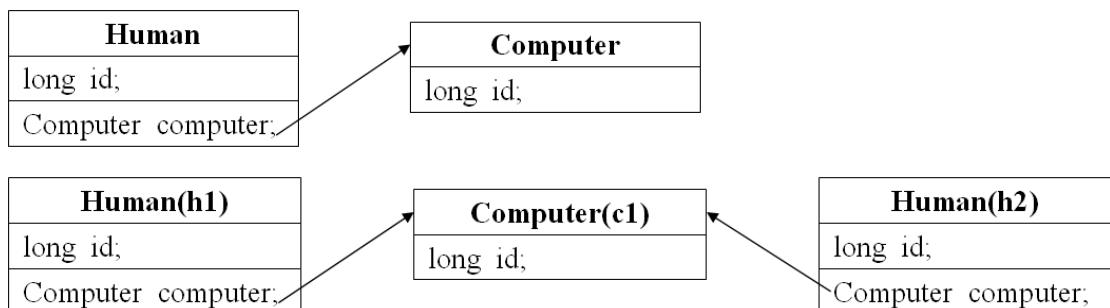


图 18-4 对象间的引用关系

下面的例子演示了这两种情况。代码如下：

```

// RelationSave.java
import java.io.*;
import java.util.*;
public class RelationSave {
    public static void main(String[] args) throws IOException {

```

```

Random rand = new Random();
FileOutputStream   fout = new FileOutputStream("data.dat");
BufferedOutputStream   bout = new BufferedOutputStream(fout);
ObjectOutputStream oout = new ObjectOutputStream(bout);

Computer c1 = new Computer();
Human h1 = new Human(c1);
oout.writeObject(h1);

//Human h2 = new Human(c1);
//oout.writeObject(h2);

oout.close();
bout.close();
fout.close();
}

}

class Human implements Serializable {
    private static long index = 0;
    private long id;
    private Computer computer;
    public Human(Computer aComputer) {
        index++;
        id = index;
        computer = aComputer;
    }
    public String toString() {
        return "Human[id=" + id + ", computer=" + computer + "]";
    }
    public boolean equalComputer(Human other) {
        return(computer == other.computer);
    }
}

class Computer implements Serializable {
    private static long index = 0;
    private long id;
    public Computer() {
        index++;
        id = index;
    }
    public String toString(){
        return "Computer[id=" + id + "]";
    }
}

```

```

// RelationRead.java
import java.io.*;
import java.util.*;

public class RelationRead {
    public static void main(String[] args) throws Exception {
        FileInputStream fin = new FileInputStream("data.dat");
        BufferedInputStream bin = new BufferedInputStream(fin);
        ObjectInputStream oin = new ObjectInputStream(bin);

        Human h1 = (Human)(oin.readObject());
        System.out.println(h1);

        //Human h2 = (Human)(oin.readObject());
        //System.out.println(h2);
        //System.out.println(h2.equalComputer(h1));

        oin.close();
        bin.close();
        fin.close();
    }
}

```

数组变量在 Java 中也是当作引用来处理的，所以也能被正确序列化。比如，类中有一个数组成员，该类的对象能被正确序列化。

基于序列化机制，对象之间的引用关系可以被序列化机制保存，不管这种引用关系多么复杂。比如列表（List）对象都能够被序列化，列表中保存的所有数据都被写入到对象流中。无论多么复杂的引用关系，都能够正常保存和读取。前面的抽奖器程序中，可以直接将 LinkedList 对象 students 通过序列化保存到文件和读取，甚至将保存程序中的 LinkedList 改为 ArrayList 都不影响读取程序的工作。

列表对象序列化的代码如下：

```

// StudentSave.java
import java.io.*;
import java.util.*;
class StudentsSave {
    public static void main(String[] args) throws IOException {
        List<Student> students = new LinkedList<Student>();
        //List<Student> students = new ArrayList<Student>();
        //读入学生数据，保存到 Student 对象数组中
        FileReader fin = new FileReader("Students.txt");
        BufferedReader in = new BufferedReader(fin);
        String line = in.readLine();

```

```

        while(line != null) {
            line = line.trim(); //去除字符串中前后空格
            if( line.length() > 0 ) { //检查是否是空行
                Student student = new Student();
                student.parseStudent(line);
                students.add(student);
            }
            line = in.readLine();
        }
        in.close();
        fin.close();

        ObjectOutputStream out = new ObjectOutputStream(new
BufferedOutputStream(new FileOutputStream("students.dat")));
        out.writeObject(students);
        out.close();
    }
}

class Student implements Serializable {
    private String id;
    private String name;
    private String department;
    public void parseStudent(String str) {
        int tokenCount;
        StringTokenizer t = new StringTokenizer(str);
        tokenCount = t.countTokens();
        id = t.nextToken();           //学号
        name = t.nextToken();         //姓名
        department = t.nextToken();   //学院
    }
    public String toString() {
        String s = id + " " + name;
        for(int l = s.length();l<21; l++) s += " "; //对齐
        return s + department;
    }
}

// StudentsRead.java
import java.io.*;
import java.util.*;
class StudentsRead {
    public static void main(String[] args) throws Exception {
        ObjectInputStream out = new ObjectInputStream(new BufferedInputStream(new
FileInputStream("students.dat")));

```

```

List students = (List)( out.readObject() );
out.close();
Iterator iter = students.iterator();
while(iter.hasNext()){
    System.out.println(iter.next());
}
}
}
}

```

由此可见，对象序列化是一个使用简单但功能强大的机制。除了可以用于将内存中的对象状态保存到文件外，还有许多其它用途。

18.3 对象序列化的其它应用

18.3.1 在网络传输中的应用

输入输出流可以是文件流，也可以是 Socket 流，所以序列化机制也可以用到网络传输数据上。利用序列化机制，一台计算机上的对象可以通过对象流传输到另一台计算机上，就好象对象从一台计算机移动到了另一台计算机上。

要使用对象流在网络上传输对象，创建 Socket 流以后，套接一个对象流即可。如图

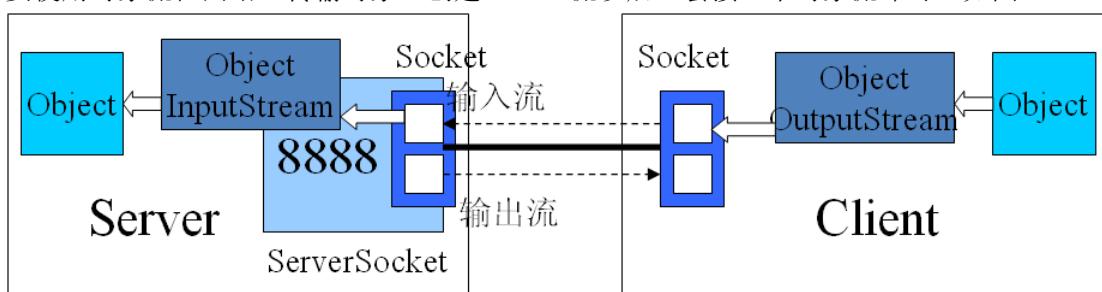


图 18-5 通过 TCP 连接传输对象

下面的例子使用对象流在网络上传输对象。代码如下：

```

// Server.java
import java.io.*;
import java.net.*;
public class Server {
    public static void main(String[] args) {
        Thread listeningThread = new ListeningThread(8888);
        listeningThread.start();
    }
}
class ListeningThread extends Thread {
    private int port;

```

```

private boolean flag = true;
private ServerSocket lServerSocket;
public ListeningThread(int aPort) {
    port = aPort;
}
public void run() {
    try {
        lServerSocket = new ServerSocket(port);
        lServerSocket.setSoTimeout(1000);
        System.out.println("Start listening.....");
        while(flag) {
            try {
                Socket incoming = lServerSocket.accept();
                System.out.println("Accept
"+incoming.getInetAddress()+"("+incoming.getPort()+"')");
                Thread t = new ServiceThread(incoming);
                t.start();
            }
            catch(SocketTimeoutException e) {
                if(!flag)break;
            }
        }
        lServerSocket.close();
        System.exit(0);
    }
    catch(IOException e) {
        System.out.println(e);
    }
}
class ServiceThread extends Thread {
    private Socket lSocket;
    public ServiceThread(Socket aSocket) {
        lSocket = aSocket;
    }
    public void run() {
        try {
            //将网络输入流套接为 ObjectInputStream
            ObjectInputStream in = new
ObjectInputStream(lSocket.getInputStream());
            PrintWriter out = new PrintWriter(lSocket.getOutputStream(),true);
            out.println("Welcome to
"+lServerSocket.getInetAddress()+"("+lServerSocket.getLocalPort()+"')");
            Object o = in.readObject(); //接收一个对象
            if(o!=null) {

```

```

        String line = o.toString();
        System.out.println(line); // 打印该对象
        out.println("Received."); // 发送应答消息
        out.flush();
    }
    ISocket.close();
    System.out.println("Thread stoped.");
}
catch(Exception e) {
    System.out.println(e);
}
}
}

// Client.java
import java.io.*;
import java.net.*;
import java.util.*;
public class Client {
    public static void main(String[] args) {
        try {
            // 建立 Socket 对象
            Socket client = new Socket("127.0.0.1", 8888);
            // 建立网络输入输出流
            BufferedReader in = new BufferedReader(new InputStreamReader(client.getInputStream()));
            // 将网络输出流套接为 ObjectOutputStream
            ObjectOutputStream out = new ObjectOutputStream(client.getOutputStream());
            // 读入欢迎词
            System.out.println(in.readLine());

            // 读入 students 对象
            ObjectInputStream fout = new ObjectInputStream(new BufferedInputStream(new FileInputStream("students.dat")));
            List students = (List)(fout.readObject());
            fout.close();

            // 向网络输出 students 对象
            out.writeObject(students);
            out.flush(); // 确保将缓冲区中数据发送，若不调用此方法，则有可能对
象不发送
        }
    }
}

```

```

//接收应答信息
System.out.println(in.readLine());

client.close();
}
catch(Exception e) {
    e.printStackTrace();
}
}
}
}

```

这里需要注意的是，在接收端要求也能访问到对象的 `class` 文件，否则无法反序列化，因为反序列化时要用到类的信息。所以，上面的例子执行时，同一目录下要有 `Student.class` 文件。

18.3.2 在对象克隆中的应用

对象克隆指创建一个与自身完全相同的对象。对象克隆中的难点是如何处理引用到的其它对象。如果不克隆这些引用到的对象，那么称为浅克隆，如图(a)；如果也克隆这些引用到的对象，那么称为深克隆，如图(b)。如果不使用序列化机制，实现对象的深克隆是很麻烦的。



图 18-6 浅克隆和深克隆

Java 中用 `Cloneable` 接口来标志一个类是可克隆的。这个接口和 `Serializable` 接口一样，只是一个标识，不需要实现任何方法。

Java 在 `Object` 类中定义了一个 `clone` 方法，用序列化机制实现了对象的深克隆。原理是先将当前对象序列化输出到一个内存字节流中，再将内存字节流读入，反序列化成一个新对象。如图

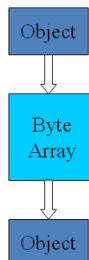


图 18-7 序列化在对象克隆中的应用

所有类都可从 Object 类继承了 clone 方法，所以能够实现深克隆。

第十九章 数据库

19.1 数据库系统与 JDBC

数据库系统提供数据存储和访问的支持。现在使用最多的是关系型数据库，采用二维表格的形式保存数据。数据库系统支持大量结构化数据的存储，支持多用户同时访问，支持远程访问。如图。比较有名的数据库系统有 Oracle, SQL Server, DB2, MySQL 等。

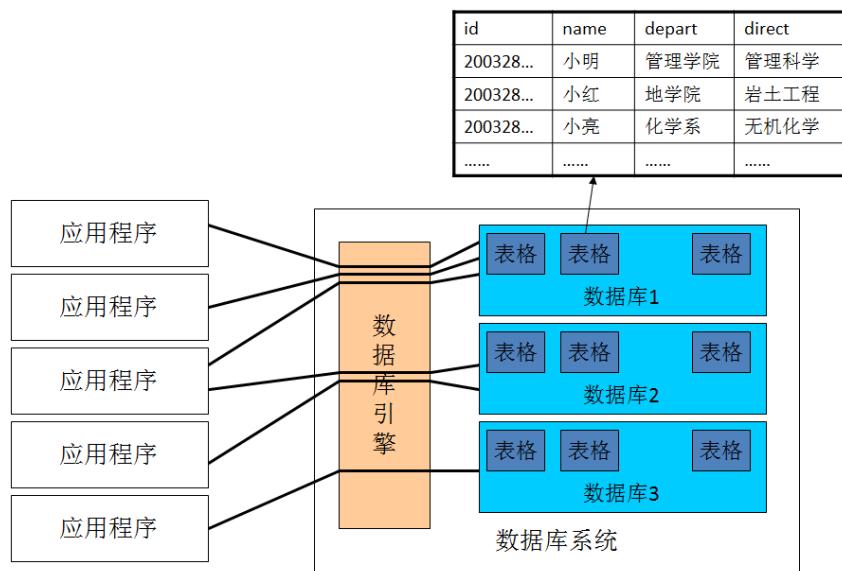


图 19-1 数据库系统

应用程序远程操作后台数据库的需要先和数据库管理系统建立连接，一般是 TCP 连接，比如 MySQL 使用 3306 端口。应用程序作为客户端，数据库管理系统作为服务器端。通过连接，客户端可以向服务器发送 SQL 命令，服务器将结果返回给客户端。

客户端一般不用直接去管理这个连接，因为这样的话编写客户端需要熟悉数据库系统接收命令和返回数据的格式。现在一般由数据库系统为客户端提供一些访问数据库的 API，客户端利用这些 API 发送 SQL 命令与处理返回的数据。这套 API 称为连接接口或驱动。如图。

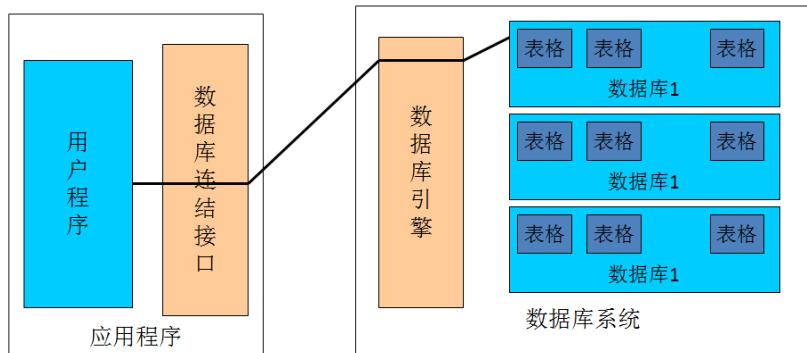


图 19-2 数据库连接接口

Java 的数据库驱动称为 JDBC (Java Data Base Connectivity, java 数据库连接), 是一种用于执行 SQL 语句的 Java API, 可以为多种不同的数据库系统提供统一的访问接口, 它由一组用 Java 语言编写的类和接口组成。JDBC 驱动程序共有 4 种连接后台数据库系统的模式, 一般使用与数据库系统直接连接的模式。

要使用 JDBC, 需要先安装数据库的 JDBC 驱动。本章基于 MySQL 数据库及其 JDBC 驱动来介绍如何用 Java 操作数据库。

19.2 MySQL 数据库安装

19.2.1 安装 MySQL

MySQL 是现在流行的关系数据库中的一种, 相比其它的数据库管理系统(DBMS)来说, MySQL 具有小巧、功能齐全、查询迅捷等优点, 关键的是它是免费的, 可以在 Internet (<http://www.mysql.cn>) 上免费下载到, 并可免费使用, 对于一般中小型, 甚至大型应用都能够胜任。

MySQL 分安装版和免安装版。安装过程非常简单, 安装版只要按照提示一步一步操作; 免安装版直接解压即可。安装后的主要目录有:

- **Bin:** 该目录中包含了 MySQL 的常用命令。其中 mysqld-max-nt.exe 用于启动 MySQL 服务器; mysql.exe 启动后可以通过控制台操作 MySQL。
- **Data:** 该目录存放 MySQL 创建的数据库。
- **Docs:** 存放 MySQL 的使用手册。

19.2.2 启动 MySQL

启动 MySQL 服务器只需运行 bin 目录下的 mysqld-max-nt.exe。成功启动后会出现一个命令行窗口, 并显示启动成功。如果此窗口没有出现或一闪即逝, 说明没能正常启动, 应检查是否已经启动或 3306 端口被占用。

为了方便使用，可以将 MySQL 安装成一个 windows 服务，这样每次系统启动时，MySQL 都能自动随系统一起启动。要安装 Windows 服务，请进入命令行，切换工作目录到 MySQL 的 bin 目录，执行：mysql -max-nt - install 命令即可。要卸载 windows 服务，则请执行 mysql -max-nt - remove。

MySQL 服务器启动后，只是启动了服务端，要操作数据库中的数据，还要通过一个客户端。MySQL 自带一个命令行客户端，叫 mysql，在 bin 目录下。要启动这个客户端，需要进入命令行窗口，切换工作目录到 MySQL 的 bin 目录，然后在命令行输入 mysql -u root -p，则将以 root 用户身份登录 MySQL，并要求输入密码验证身份。

也有一些图形界面的客户端可以使用，比如 MySQL-Front。命令行客户端的具体用法这里不详细介绍，可以参阅其它文档。

19.2.3 创建数据库表

本节介绍使用 MySQL-Front 来创建表格。

先启动 MySQL-Front，并建立到数据库的连接。建立连接前，请确保 MySQL 服务器已经启动。MySQL-Front 启动后，首先出现下图所示对话框，这里首先输入一个连接名称。这个名称是任意命名，但建议最好采用要连接的数据库名。

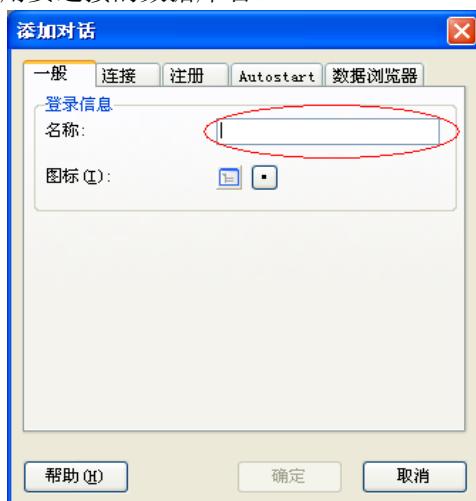


图 19-3 对话名称

然后单击“连接”切换到连接标签，如下图所示，在服务器输入框中输入 localhost 或 MySQL 所在的主机 IP 地址。



图 19-4 对话连接设置

下一步请单击“注册”切换到注册标签，如下图所示。这里请输入用户名 root，密码为空，然后单击“数据库”右边的按钮，如果用户名和密码输入准确的话，单击按钮后将列出指定主机（连接标签中设置）上的所有数据库，请从列表中选择一个数据库，确定后，您选择数据名称将出现在数据库输入框。

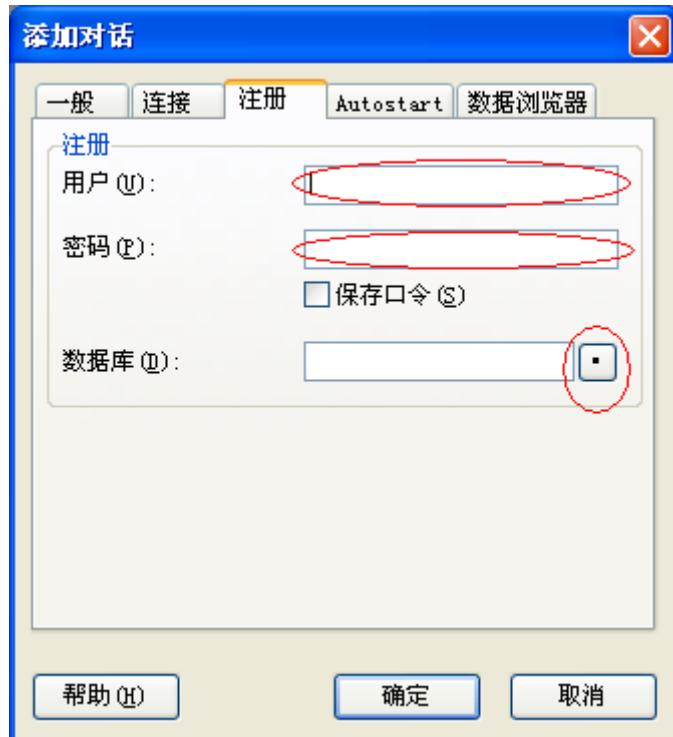


图 19-5 数据库访问用户名和密码

下一步再回到连接标签。这次，在这里选择字符集 gb2312。最后单击确定，完成到数据库的连接。

创建完成到数据库的连接后既可通过 MySQL-Front 操作 MySQL 数据库了。具体包括创建数据库、创建表、添加/修改/查询/删除记录等操作。

1、创建数据库

要创建一个新的数据库，可以在导航栏（主界面左边视图）的主机上单击鼠标右键，选择“新建”→“数据库…”（如下图 a 所示），或者单击“数据库”菜单，选择“新建”→“数据库…”（如下图 b 所示）。

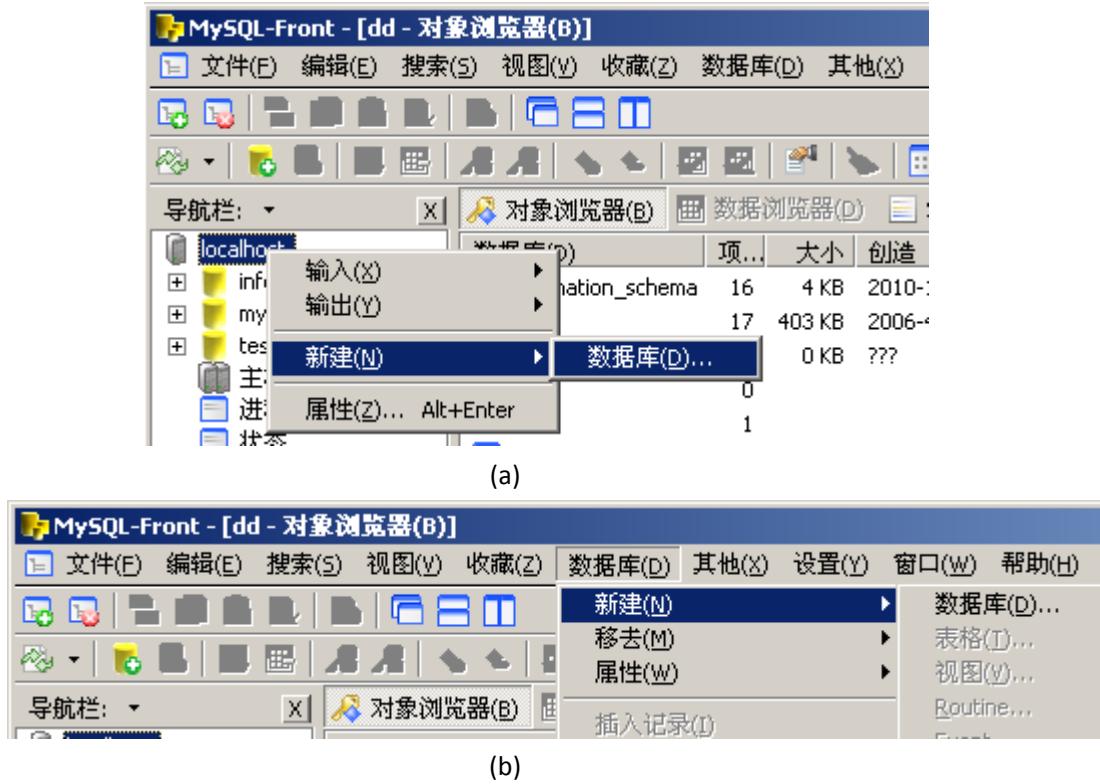


图 19-6 创建数据库

系统将弹出如下图所示对话框，在该对话框中输入相应的数据库名和选择需要的字符集即可。注意，Mysql-Front 不支持中文数据库名称。



图 19-7 数据库名称

2、创建数据表

创建数据库后，可以进一步在该数据库中创建相应数据表，具体操作如下：

在要创建数据表的数据库上单击鼠标右键，选择“新建”→“表格…”（如下图所示）

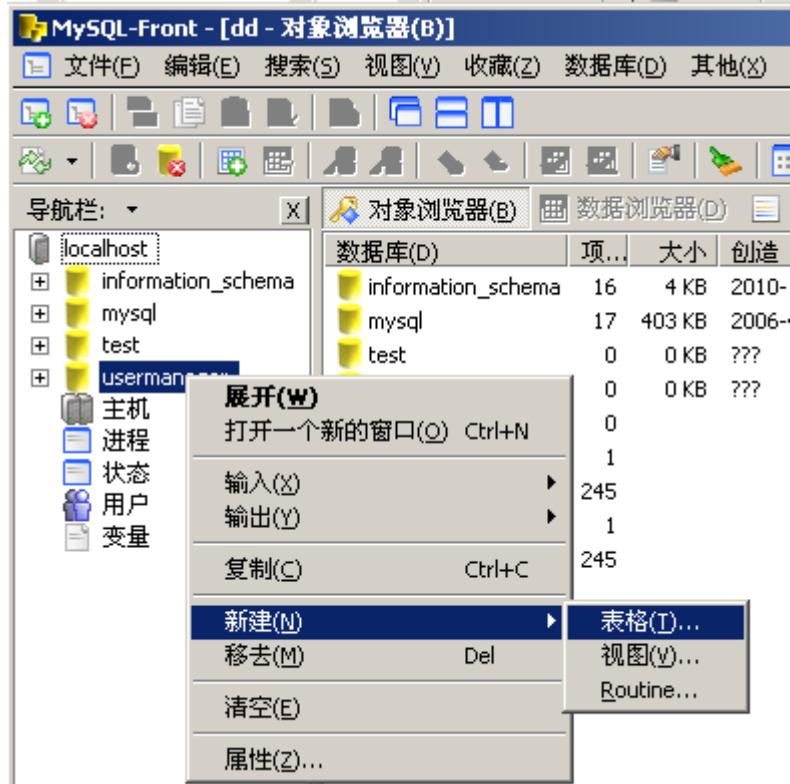


图 19-8 创建数据表

将弹出如下图所示对话框。该对话框中包括一般、字段、索引和约束四个标签。输入数据表的“一般”信息后，单击“字段”输入数据表的字段信息，如下图所示。

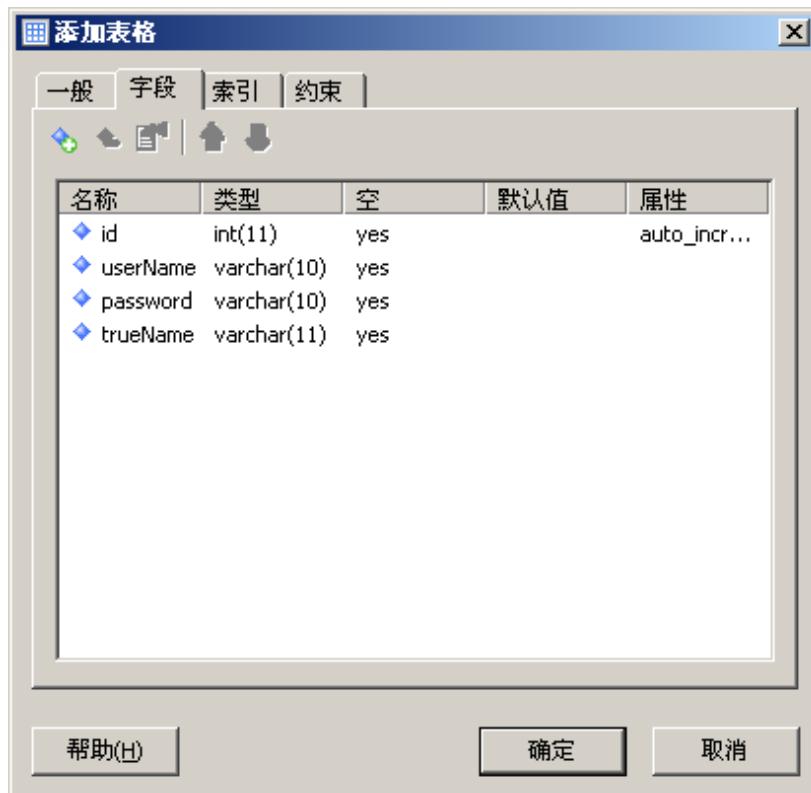


图 19-9 数据表字段

这样，数据表就创建好了。

19.3 建立连接

JDBC 主要概念包括：连接（Connection）、语句（Statement）和结果集（ResultSet）。

Connection 代表与数据库的连接。连接过程包括所执行的 SQL 语句和在该连接上所返回的结果。一个应用程序可与单个数据库有一个或多个连接，或者可与许多数据库有连接。连接一旦建立，就可用来向它所涉及的数据库传送 SQL 语句。JDBC 提供了三个类（Statement、PreparedStatement、CallableStatement），用于向数据库发送 SQL 语句。如果声明对象执行的是 Select 语句，则将返回一个结果集（ResultSet）对象。ResultSet 是一个存储查询结果的对象，但是结果集并不仅仅具有存储的功能，它同时还具有操纵数据的功能，可完成对数据的更新等。

JDBC 的具体使用步骤如下：

- 1) 注册加载一个 driver 驱动；
- 2) 创建数据库连接（Connection）；
- 3) 创建语句对象 Statement、PreparedStatement 或 CallableStatement（发送 sql）、执行 sql 语句；
- 4) 处理 sql 结果集 ResultSet（select 语句）；
- 5) 关闭 Statement；
- 6) 关闭连接 Connection。

注意：5)、6) 两个步骤是必须要做的，因为这些资源不会自动释放，必须要自己关闭。

在编写程序前，先安装 JDBC 驱动。MySQL Connector 是 MySQL 的 JDBC 驱动，最简单的安装方法是：先解压缩，然后将 com 目录所有内容复制到 classpath 目录或程序当前目录。

要使用数据库，首先要在应用程序和数据库之间建立连结。需要两个步骤：

1. 向 JDBC 驱动程序管理器注册驱动程序；
2. 建立与数据库的连结，需给出数据库的地址、用户名和口令。

注册驱动程序实际上是将 com.mysql.jdbc.Driver 类装入内存，使用 Class 类的 `forName` 方法可以将一个类装入内存。

建立连接用的是 `DriverManager` 类的 `getConnection` 方法，需要三个参数：第一个参数是采用 url 格式描述的数据库地址字符串，一般为“`jdbc:mysql://服务器 ip 地址/数据库名称`”；第二、三两个参数是登录数据库用的用户名和密码。

下面的例子建立一个数据库连接，代码如下：

```
Class.forName("com.mysql.jdbc.Driver");
String url="jdbc:mysql://localhost:3306/stuManager";
```

```

string user="root";
string pass="root";
Connection con=DriverManager.getConnection(url,user,pass);

```

下表中列出了常用数据库的驱动和 URL:

数据库	驱动	URL
SQLServer	com.microsoft.sqlserver.jdbc.SQLSe rverDriver	jdbc:sqlserver://localhost:1433; DatabaseName=dbName
Oracle	oracle.jdbc.driver.OracleDriver	jdbc:oracle:thin:@IP:Port:SID
MySQL	com.mysql.jdbc.Driver	jdbc:mysql://IP:Port/dbName
Access	sun.jdbc.odbc.JdbcOdbcDriver	jdbc:odbc:driver={Microsoft Access Driver (*.mdb)};DBQ=dbFileName

建立连接够就可以通过连接执行 SQL 命令了。其常用主要命令如下：

- 读取表中记录: `SELECT * FROM table1`
- 查询满足特定条件的记录: `SELECT * FROM table1 WHERE ...`
- 向表中插入一条记录: `INSERT INTO table1 VALUES(...)`
- 从表格删除记录: `DELETE FROM table1 WHERE ...`
- 修改记录: `UPDATE table1 SET ... WHERE ...`

在处理完对数据库的操作后,一定要将 `Connection` 对象关闭,以释放 JDBC 占用的系统资源。关闭连接使用 `Connection` 对象的 `close` 方法。

在不关闭 `connection` 对象的前提下再次用 `DriverManager` 静态类初始化新的 `Connection` 对象会产生系统错误。而一个已经建立连接的 `Connection` 对象可以同时初始化多个 `Statement` 对象。对应不同的数据库管理系统的 `Connection` 对象可以初始化 `Statement` 对象的个数是不同的。

要在 MySQL 中存储中文字符,需要注意以下事项:

- 1、 创建数据库时,字符集选择 gbk;
- 2、 创建数据表时,字符集选择 gbk;
- 3、 在程序中连接数据库时使用下面的 URL:
`"jdbc:mysql://127.0.0.1:3306/login?user=root&password=&characterEncoding=gbk"`
该 URL 中设置了字符集编码,意思是告诉 JDBC 传输的 SQL 语句中有中文。

19.4 执行 SQL 语句

操作数据通过 SQL 语句进行,其常用主要命令如下:

- 读取表中记录: `SELECT * FROM table1`

- 查询满足特定条件的记录: `SELECT * FROM table1 WHERE ...`
- 向表中插入一条记录: `INSERT INTO table1 VALUES(...)`
- 从表格删除记录: `DELETE FROM table1 WHERE ...`
- 修改记录: `UPDATE table1 SET ... WHERE ...`

执行 SQL 语句是通过语句对象完成的。语句对象有三种: `Statement` (执行简单的、无参数的 SQL 语句)、`PreparedStatement` (预编译语句对象)、`CallableStatement` (用来执行存储过程)。这里主要介绍前两个的用法。

1. Statement 对象

该对象可以通过 `Connection` 对象的 `createStatement` 方法创建, `Statement` 对象的主要方法有:

- `executeQuery` 方法: 执行查询语句 (`select`), 将查到的记录以结果集 (`ResultSet`) 的方式返回。
- `executeUpdate` 方法: 执行 `insert`、`update`、`delete` 操作。
- `execute` 方法: 执行查询语句 (可以返回多个结果集), 也可以执行更新操作。

建立 `Statement` 对象并执行插入一条记录的 SQL 语句的例子如下:

```
Statement stmt = con.createStatement();
String sql = "INSERT INTO table1 VALUES( '200328000500074', '小明', '地球
科学学院', '岩土工程' )"; // 要执行的 sql 语句
stmt.executeUpdate(sql);
```

以上代码实现了向 `table1` 表中插入一条新记录。

2. PreparedStatement 对象

`PreparedStatement` 实例包含已编译的 SQL 语句。这就是使语句“准备好”。包含于 `PreparedStatement` 对象中的 SQL 语句可具有一个或多个 IN 参数。IN 参数的值在 SQL 语句创建时未被指定。相反的, 该语句为每个 IN 参数保留一个问号 (“?”) 作为占位符。每个问号的值必须在该语句执行之前, 通过适当的 `setXXX` 方法来提供。

由于 `PreparedStatement` 对象已预编译过, 所以其执行速度要快于 `Statement` 对象。因此, 多次执行的 SQL 语句经常创建为 `PreparedStatement` 对象, 以提高效率。

PreparedStatement 举例

```
PreparedStatement pstmt = con.prepareStatement("Update Favorites set favorite=? where
id=?");
pstmt.setLong(1,"Song");
pstmt.setLong(2,1);
pstmt.executeUpdate();
```

以上代码实现了将 `Favorites` 表中 `id` 值等于 1 的记录的 `favorite` 字段的值更新为 `Sing`。一旦设置了给定语句的参数值, 就可以用它多次执行该语句, 直到调用 `clearParameters` 方法为止。

19.5 处理结果集

一般情况下，Statement 对象的 executeQuery 方法返回一个 ResultSet 对象。ResultSet 对象中存放的是满足查询条件的结果集。虽然说结果集(ResultSet)是一个存储查询结果的对象，但是结果集并不仅仅具有存储的功能，他同时还具有操纵数据的功能，可能完成对数据的更新等。

结果集读取数据的方法主要是 ResultSet 对象的 getXXX(int)或 getXXX (String)，其中 XXX 代表某种数据类型，如 Int、Float、String、Date、Boolean 等，其参数可以是整型表示第几列（是从 1 开始的），也可以是字符串形式的列名；该方法的返回值是对应的 XXX 类型的值。结果集从其使用的特点上可以分为四类，这四类的结果集的所具备的特点都是和 Statement 语句的创建有关，因为结果集是通过 Statement 语句执行后产生的。

1. 仅顺序读取的 ResultSet

第一种 ResultSet 是仅完成查询结果的存储功能，而且只能读取一次，不能够来回的滚动读取。这种结果集的创建方式如下：

```
Statement st = conn.createStatement(); // conn,连接对象  
ResultSet rs = Statement.executeQuery(sqlStr); // sqlStr, 要执行的 select 查询语句
```

由于这种结果集不支持滚动的读取功能，所以，如果获得这样一个结果集，只能使用它里面的 next()方法，逐个的读取数据。

2. 可滚动的 ResultSet

该类型支持向后滚动取得记录 next()、向前滚动取记录 previous()、回到第一条记录 first()、移动到最后一条记录 last()、取第 n 条记录 absolute(int n)和相对当前行移动 n 条记录 relative(int n)。要实现这样的 ResultSet，在创建 Statement 时用如下的方法。

```
Statement st = conn.createStatement(int resultSetType, int resultSetConcurrency)
```

```
ResultSet rs = st.executeQuery(sqlStr)
```

其中两个参数的意义是：

1) resultSetType 是设置 ResultSet 对象的类型可滚动，或者是不可滚动。取值如下：

```
ResultSet.TYPE_FORWARD_ONLY 只能向前滚动。
```

ResultSet.TYPE_SCROLL_INSENSITIVE 和 ResultSet.TYPE_SCROLL_SENSITIVE 这两个方法都能够实现任意的前后滚动，使用各种移动的 ResultSet 指针的方法。二者的区别在于前者对于修改不敏感，而后者对于修改敏感。

2) resultSetConcurrency 是设置 ResultSet 对象是否能够修改，取值如下：

```
ResultSet.CONCUR_READ_ONLY 设置为只读类型的参数。
```

```
ResultSet.CONCUR_UPDATABLE 设置为可修改类型的参数。
```

所以如果只是想要可以滚动的类型的 Result，只要把 Statement 如下赋值就行了。

```
Statement st = conn.createStatement(ResultsetType_SCROLL_INSENSITIVE,  
                                    Resultset.CONCUR_READ_ONLY);
```

```
ResultSet rs = st.executeQuery(sqlStr);
```

用这个 Statement 执行的查询语句得到的就是可滚动的 ResultSet。

3. 可更新的 ResultSet

这样的 ResultSet 对象可以完成对数据库中表的修改，但是我们知道 ResultSet 只是相当于数据库中表的视图，所以并不是所有的 ResultSet 只要设置了可更新就能够完成更新的，能够完成更新的 ResultSet 的 SQL 语句必须要具备如下的属性：

只引用了单个表。

不含有 join 或者 group by 子句。

那些列中要包含主关键字。

具有上述条件的，可更新的 ResultSet 可以完成对数据的修改，可更新的结果集的创建方法是：

```
Statement st = createstatement(Result.TYPE_SCROLL_INSENSITIVE,  
                               Result.CONCUR_UPDATABLE)
```

这样的 Statement 的执行结果得到的就是可更新的结果集。更新的方法是，把 ResultSet 的游标移动到您要更新的行，然后调用 updateXXX()，这个方法 XXX 的含义和 getXXX() 是相同的。updateXXX() 方法有两个参数：第一个是要更新的列，可以是列名或者序号；第二个是要更新的数据，这个数据类型要和 XXX 相同。每完成对一行的 update 要调用 updateRow() 完成对数据库的写入。

要完成对数据库的插入，首先调用 moveToInsertRow() 移动到插入行，然后调用 updateXXX 的方法完成对各列数据的更新，最后调用 insertRow() 方法将数据写入到数据库。

4. 可保持的 ResultSet

正常情况下如果使用 Statement 执行完一个查询，又去执行另一个查询时第一个查询的结果集就会被关闭。也就是说，所有的 Statement 的查询对应的结果集是一个，如果调用 Connection 的 commit() 方法也会关闭结果集。可保持性就是指当 ResultSet 的结果被提交时，是被关闭还是不被关闭。JDBC2.0 和 1.0 提供的都是提交后 ResultSet 就会被关闭。不过在 JDBC3.0 中，我们可以设置 ResultSet 是否关闭。要完成这样的 ResultSet 的对象的创建，要使用的 Statement 的创建要具有三个参数，如下所示：

```
Statement st=conn.createStatement(int resultsetscrollable,  
                               int resultsetupdateable,  
                               int resultsetSetHoldability)
```

```
ResultSet rs = st.executeQuery(sqlStr);
```

前两个参数和两个参数的 createStatement 方法中的参数是完全相同的，这里只介绍第三个参数：

resultSetHoldability 表示在结果集提交后结果集是否打开，取值有两个：

ResultSet.HOLD_CURSORS_OVER_COMMIT：表示修改提交时，不关闭数据库。

ResultSet.CLOSE_CURSORS_AT_COMMIT：表示修改提交时 ResultSet 关闭。

不过这种功能只是在 JDBC3.0 的驱动下才能成立。

使用 ResultSet 对象读取数据的例子如下：

```
String sql = "select * from Student";  
.....  
try {  
    stmt = con.createStatement();  
    rs = stmt.executeQuery(sql);
```

```

        while(rs.next()){
            System.out.print(rs.getInt("id"));
            System.out.print(rs.getString("userNmae"));
            System.out.print(rs.getString("password"));
            System.out.print(rs.getString("trueName"));
            System.out.print(rs.getInt("sex"));
            System.out.print(rs.getString("favors"));
            System.out.print(rs.getDate("birthday"));
            System.out.print(rs.getInt("classId"));
            System.out.print(rs.getString("introduction"));
            System.out.println();
        }

        .....
    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

使用 ResultSet 对象更新数据的例子如下：

```

// 创建 Statement 对象
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                         ResultSet.CONCUR_UPDATABLE);
// 使用 ResultSet 对象更新最后一条记录
ResultSet result = stmt.executeQuery("SELECT * FROM user ");
result.last(); // 移动光标到想要修改的记录
result.updateString("username", "lin");
result.updateString("password", "1234");
result.updateRow(); // 使用 updateRow()让更新生效

```

使用 ResultSet 对象添加新数据的例子如下

```

ResultSet result = stmt.executeQuery("SELECT * FROM user ");
result.moveToInsertRow(); // 移至新增数据处
result.updateString("username", "newUserName");
result.updateString("password", "newPassword");
// update other fields
result.insertRow(); // 执行插入操作

```

使用 ResultSet 对象删除记录的例子如下：

```

ResultSet result = stmt.executeQuery("SELECT * FROM user ");
result.last(); // 将光标移至要删除的记录
result.deleteRow(); // 执行删除操作

```

19.6 事务

有时需要同时修改两个以上数据表的内容，并且这种修改是一个不可分割的操作，即或者两个表同时修改成功，或者同时不修改，不能出现一个表修改成功而另一个表不成功的情况，这通常是应用中数据一致性的要求。数据库系统中的“事务”就是为这种需要而设计的。

在 JDBC 的数据库操作中，一项事务是由一条或是多条 SQL 语句所组成的一个不可分割的工作单元。

事务具有 ACID4 个特性：

1. 原子性 (Atomicity): 事务中的全部操作在数据库中是不可分割的，要么全部完成，要么均不执行。
2. 一致性 (Consistency): 几个并行执行的事务，其执行结果必须与按某一顺序串行执行的结果相一致。
3. 隔离性 (Isolation): 事务的执行不受其他事务的干扰，事务执行的中间结果对其他事务必须是透明的。
4. 持久性 (Durability): 对于任意已提交事务，系统必须保证该事务对数据库的改变不被丢失，即使数据库出现故障。

已提交事务是指成功执行完毕的事务，未能成功完成的事务称为中止事务，对中止事务造成的变更需要进行撤销处理，称为事务回滚。我们通过提交 `commit()` 或是回滚 `rollback()` 来结束事务的操作。

在 JDBC 中，事务操作缺省是自动提交。也就是说，一条对数据库的更新语句代表一项事务操作，操作成功后，系统将自动调用 `commit()` 来提交，否则将调用 `rollback()` 来回滚。

在 JDBC 中，可以通过调用 `setAutoCommit(false)` 来禁止自动提交。之后就可以把多个数据库操作的表达式作为一个事务，在操作完成后调用 `commit()` 来进行整体提交，倘若其中一个表达式操作失败，都不会执行到 `commit()`，并且将产生响应的异常；此时就可以在异常捕获时调用 `rollback()` 进行回滚。这样做可以保持多次更新操作后，相关数据的一致性。

通过 JDBC 使用事务的步骤：

- 1、调用 `SetAutoCommit(false)` 来关闭自动提交；
- 2、执行若干条 sql 语句；
- 3、结束这个事务：提交或回滚；
- 4、调用 `SetAutoCommit(true)` 来恢复 JDBC 事务的默认提交方式。

JDBC API 支持事务对数据库的加锁，并且提供了 5 种操作支持，2 种加锁，下表中给出了具体的操作，其中，最后一项为表加锁，其余 3~4 项为行加锁。

TRANSACTION_NONE	禁止事务操作和加锁
TRANSACTION_READ_UNCOMMITTED	允许脏数据读写、重复读写和影象读写
TRANSACTION_READ_COMMITTED	禁止脏数据读写，允许重复读写和影象读写

TRANSACTION_REPEATABLE_READ	禁止脏数据读写和重复读写，允许影象读写
TRANSACTION_SERIALIZABLE	禁止脏数据读写、重复读写和允许影象读写

脏数据读写 (**dirty reads**): 当一个事务修改了某一数据行的值而未提交时，另一事务读取了此行值。倘若前一事务发生了回滚，则后一事务将得到一个无效的值（脏数据）。

重复读写 (**repeatable reads**): 当一个事务在读取某一数据行时，另一事务同时在修改此数据行。则前一事务在重复读取此行时将得到一个不一致的值。

影象读写 (**phantom reads**): 当一个事务在某一表中进行数据查询时，另一事务恰好插入了满足了查询条件的数据行，则前一事务在重复读取满足条件的值时，将得到一个额外的“影象”值。

JDBC 根据数据库提供的缺省值来设置事务支持及其加锁，当然，也可以手工设置：

```
setTransactionIsolation (TRANSACTION_READ_UNCOMMITTED) ;
```

可以查看数据库的当前设置：

```
getTransactionIsolation()
```

需要注意的是，在进行手动设置时，数据库及其驱动程序必须得支持相应的事务操作操作才行。

上述设置随着值的增加，其事务的独立性增加，更能有效的防止事务操作之间的冲突；同时也增加了加锁的开销，降低了用户之间访问数据库的并发性，程序的运行效率也回随之降低。因此得平衡程序运行效率和数据一致性之间的冲突。一般来说：

1. 对于只涉及到数据库的查询操作时，可以采用 **TRANSACTION_READ_UNCOMMITTED** 方式；
2. 对于数据查询远多于更新的操作，可以采用 **TRANSACTION_READ_COMMITTED** 方式；
3. 对于更新操作较多的，可以采用 **TRANSACTION_REPEATABLE_READ**；
4. 在数据一致性要求更高的场合再考虑最后一项，由于涉及到表加锁，因此会对程序运行效率产生较大的影响。