从小白到大牛

Java 全新立体化图书

关东升 著

立体化
图书包含:
图书、在线服务
视频、课件
等等

版权信息

书名：Java从小白到大牛

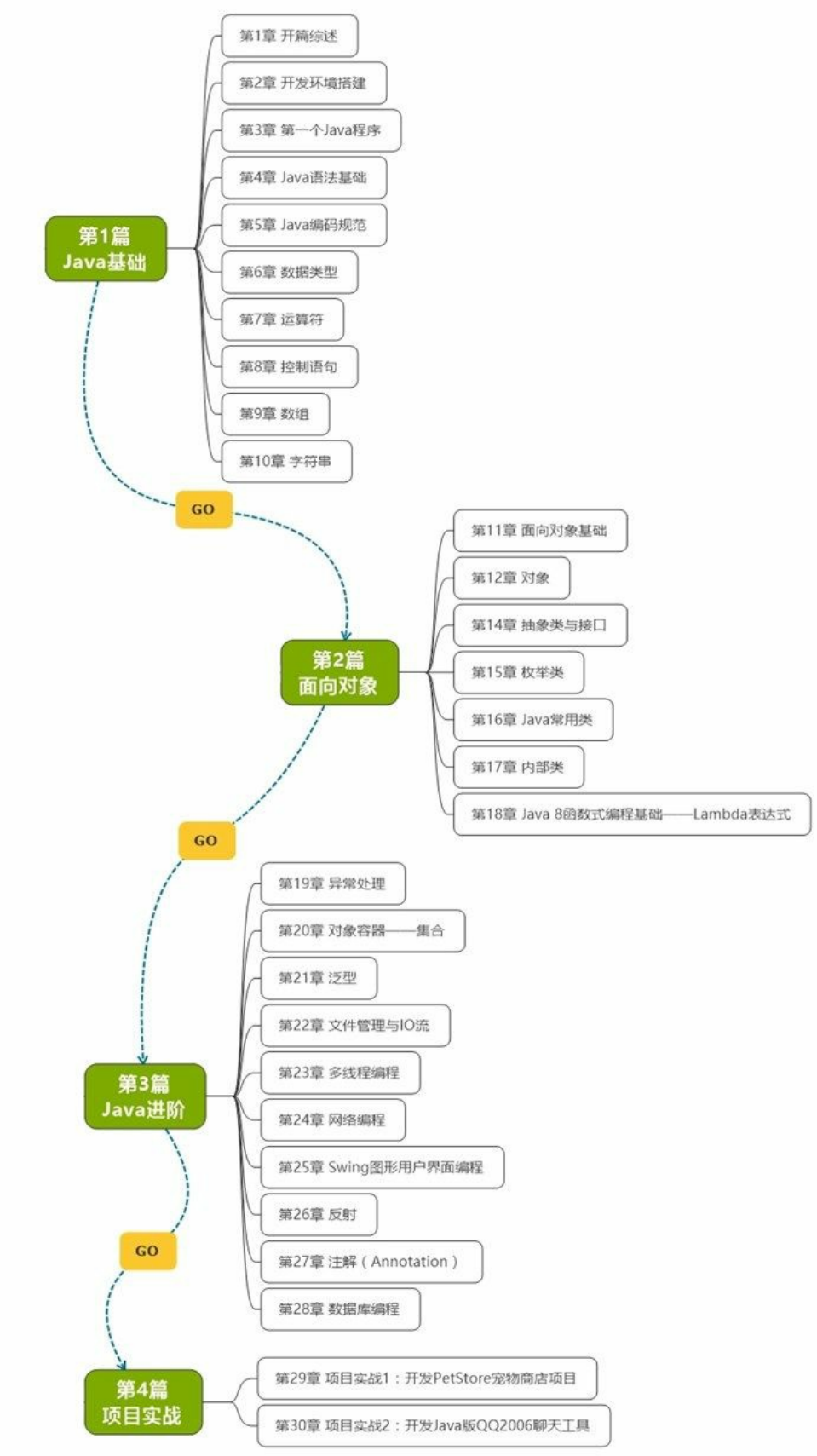
作者：关东升

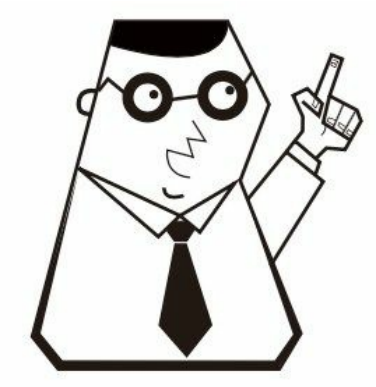
内容简介

本书是一本Java语言学习立体教程，读者群是零基础小白，通过本书的学习能够成为Java大牛。主要内容包括：Java语法基础、Java编码规范、数据类型、运算符、控制语句、数组、字符串、面向对象基础、继承与多态、抽象类与接口、枚举类、Java常用类、集合框架、泛型、反射机制、Annotation注解、Lambda表达式、异常处理、输入输出、多线程、网络编程和图形用户界面编程、反射、注解和数据库编程等技术。最后是项目实战，在部分系统地讲解了两个项目：PetStore宠物商店和Java版QQ2006聊天工具开发过程。

“立体教程”是指读者购买了我们图书后，我们同时赠送与此书配套的视频、课件和QQ答疑服务。

本书学习路线图：


第1篇
Java基础
第1章 开篇综述
第2章 开发环境搭建
第3章 第一个Java程序
第4章 Java语法基础
第5章 Java编码规范
第6章 数据类型
第7章 运算符
第8章 控制语句
第9章 数组
第10章 字符串
GO
第2篇
面向对象
第11章 面向对象基础
第12章 对象
第14章 抽象类与接口
第15章 枚举类
第16章 Java常用类
第17章 内部类
第18章 Java 8函数式编程基础——Lambda表达式
GO
第3篇
Java进阶
第19章 异常处理
第20章 对象容器——集合
第21章 泛型
第22章 文件管理与IO流
第23章 多线程编程
第24章 网络编程
第25章 Swing图形用户界面编程
第26章 反射
第27章 注解（Annotation）
第28章 数据库编程
GO
第4篇
项目实战
第29章 项目实战1：开发PetStore宠物商店项目
第30章 项目实战2：开发Java版QQ2006聊天工具



内容说明

全书分为4篇，共30章。

第一篇为基础篇，共**10**章内容，介绍了**Java**语言的一些基础知识。

第1章开篇综述。首先介绍了Java的历史、Java语言的特点，然后介绍了Java三大平台，最后介绍了Java虚拟机。

第2章开发环境搭建。介绍了Java开发环境搭建，其中重点介绍了Eclipse工具的下载、安装和使用。此外，还介绍了其他的一些工具：IntelliJ IDEA和NetBeans，以及文本编辑工具EditPlus+JDK的配置过程。

第3章第一个Java程序。介绍使用Eclipse和使用文本工具+JDK实现该示例具体过程。

第4章 Java语法基础。介绍了Java的一些基本语法，其中包括标识符、关键字、保留字、常量、变量、表达式等内容。

第5章 Java编码规范。介绍了Java的编码规范，包括命名规范、注释规范、声明规范和代码排版等内容。

第6章数据类型。介绍了Java中的数据类型，包括基本数据类型和引用数据类型，以及数值类型如何互相转换。

第7章运算符。介绍了Java语言的基本运算符，包括算术运算符、关系运算符、逻辑运算符、位运算符和其他运算符。

第8章控制语句。介绍了Java语言的控制语句，包括分支语句（if和switch）、循环语句（while、do-while、for和for-each）和跳转语句（break和continue）等。

第9章数组。介绍了Java的数组，包括一维数组和多维数组。另外，还介绍了不规则数组。

第10章字符串。介绍了Java中的字符串，Java字符串类分为：可变字符串类（String）和不可变字符串类（StringBuilder和StringBuffer）。然后分别介绍了这些字符串类的用法。

第二篇为面向对象篇，共**8**章，介绍了**Java**语言面向对象相关知识。

第11章面向对象基础。本章主要介绍了面向对象基础知识。首先介绍了面向对象一些基本概念，面向对象三个基本特性。然后介绍了类、包、方法重载和访问控制。最后介绍了静态变量、静态方法和静态代码块。

第12章对象。介绍了如何创建Java对象，如何理解构造方法的作用。此外，还介绍了this关键字的使用等。

第13章继承与多态。介绍了Java中的继承概念，在继承时会发生方法的覆盖、变量的隐藏。然后介绍了Java中的多态概念，以及引用类型检查和类型转换。最后还介绍了final关键字。

第14章抽象类与接口。介绍了抽象类和接口的概念，以及如何声明抽象类和接口，如何实现抽象类和接口。最后介绍了Java 8之后的接口的新变化。

第15章枚举类。介绍了Java中枚举的作用、特点和常用方法。

第16章 Java常用类。介绍了Object类、包装类、Math类、BigInteger类和BigDecimal类。最后还介绍了旧版本日期时间类和Java 8之后的日期时间类。

第17章内部类。介绍了内部类的概念，熟悉了内部类的划分，以及如何编写内部类。

第18章 Java 8函数式编程基础——Lambda表达式。介绍了Lambda表达式，为什么使用Lambda表达式，Lambda表达式的优点是什么，Lambda表达式标准语法，Lambda表达式的几个简写方式。

第三篇为**Java**进阶篇，共**10**章，介绍了**Java**语言的一些高级知识。

第19章异常处理。介绍了Java异常处理机制，其中包括Java异常类继承层次、捕获异常、释放资源、throws、throw和自定义异常类。

第20章对象容器——集合。介绍了Java中的集合，其中包括常用接口Collection、Set、List和Map，以及集合的遍历操作。

第21章泛型。介绍了Java中的泛型技术，包括泛型概念、在集合中使用泛型、自定义泛型类、自定义泛型接口和泛型方法等。

第22章文件管理与I/O流。本章主要介绍了Java文件管理和I/O流技术。其中包括File类使用、字节流（InputStream和OutputStream以及它们的子类）和字符流（Reader和Writer以及它们的子类）。

第23章多线程编程。首先介绍了线程相关的一些概念，然后介绍了如何创建子线程、线程状态、线程管理、线程安全和线程间通信等内容。

第24章网络编程。介绍了Java网络编程，首先介绍了一些网络方面的基本知识。然后重点介绍了TCP Socket编程和UDP Socket编程。此外，还介绍了数据交换格式，并重点介绍了JSON数据交换格式，由于Java官方没有提供JSON解码和编码库，需要是使用第三方库。最后介绍了使用URL类访问互联网资源。

第25章 Swing图形用户界面编程。介绍了Java中图形用户界面编程技术Swing。

第26章反射。介绍了通过反射机制创建对象、访问构造方法、访问方法和访问成员变量。

第27章注解（Annotation）。首先介绍了基本注解，接着介绍了元注解，最后介绍了自定义注解。读者需要掌握基本注解有哪些它们的用途，了解元注解、自定义注解，了解读取自定义注解信息的方法。另外，读者不要把注解与注释混淆了。

第28章数据库编程。首先介绍MySQL数据库的安装、配置和日常的管理命令，然后重点讲解了JDBC数据库编程技术。

第四篇为项目实战篇，共**2**章，介绍了**Java**项目开发过程中相关的技术。

第29章项目实战1：开发PetStore宠物商店项目。完整介绍PetStore宠物商店项目的设计和开发过程。

第30章项目实战2：开发Java版QQ2006聊天工具。完整介绍QQ聊天工具的设计和开发过程。

前言

1998年我在一本计算机杂志上看到介绍Java语言的文章，文中提到这种语言刚刚诞生就很快风靡全球，它的最大特点是跨平台，能够应用于Internet开发。抱着对Java语言好奇，我买了一本介绍Java语言的图书，很快我被它的特点吸引。正因为有了Java语言的基础，1999年我去了一家互联网公司，做Java Web程序员，那时候还没有JSP技术，而是使用Servlet技术，这一个干就是十多年Java。当初很多搞Java同事以及我的学生，现在不再写代码了，而我却依然在写代码。我使用Java从最初编写Web程序，到现在编写Android程序；从桌面到Web，再到移动平台，感叹Java语言的生命力，经过20多年的发展Java语言变得更加成熟、更加易用。

本书是智捷课堂开发的立体化图书的中一本，所谓“立体化图书”就是图书包含：书籍、视频、课件和服务等内容。智捷课堂真正地将广大读者看做自己的衣食父母，除了电子图书和纸质图书外，还提供真材实料的配套视频和教学课件，以及答疑服务，真正做到手把手教会读者学会书中内容。

关东升　2017年5月15日　于北京

第 1 章　开篇综述

Java诞生到现在已经有20多年了，但是Java仍然是非常热门的编程语言之一，很多平台中使用Java开发。表1-1所示的是TIOBE社区发布的2016年5月和2017年5月的编程语言排行榜，可见Java语言的热度，或许这也是很多人选择学习Java的主要原因。

表 **1-1**　**IOBE**编程语言排行榜

2017年5月	2016年5月	变化	编程语言	评级	评级变化
1	1		Java	14.639%	-6.320%
2	2		C	7.002%	-6.220%
3	3		C++	4.751%	-1.950%
4	5	^	Python	3.548%	-0.240%
5	4	v	C#	3.457%	-1.020%
6	10	^^	Visual Basic .NET	3.391%	1.070%
7	7		JavaScript	3.071%	0.730%
8	12	^^	Assembly language	2.859%	0.980%
9	6	v	PHP	2.693%	-0.300%
10	9	v	Perl	2.602%	0.280%
11	8	v	Ruby	2.429%	0.090%
12	13	^	Visual Basic	2.347%	0.520%
13	15	^	Swift	2.274%	0.680%
14	16	^	R	2.192%	0.860%
15	14	v	Objective-C	2.101%	0.500%
16	42	^^	Go	2.080%	1.830%
17	18	^	MATLAB	2.063%	0.780%
18	11	vv	Delphi/Object Pascal	2.038%	0.030%
19	19		PL/SQL	1.676%	0.470%
20	22	^	Scratch	1.668%	0.740%

1.1 Java语言历史

在正式学习Java语言之前，读者有必要先来了解一下Java的历史。1990年底美国Sun公司[1]成立了一个叫做Green的项目组，该Green项目主要目标是为消费类电子产品开发一种分布式系统，使之能够操控电冰箱、电视机等家用电器。

> [1]Sun Microsystems公司创建于1982年，主要产品是工作站及服务器。1986年在美国成功上市，1992年Sun推出了市场上第一台多CPU台式机，1993年进入财富500强，1995年开发了Java语言，2010年被Oracle（甲骨文）公司收购。现在Java技术是由甲骨文公司提供的。

消费类电子产品种类很多，包括掌上电脑（个人数字助理，Personal Digital Assistant，简称PDA）、机顶盒、手机等等，这些消费类电子产品所采用的处理芯片和操作系统基本上都是不相同的，存在跨平台等问题。开始Green项目组考虑采用C++语言来编写消费类电子产品的应用程序，但是C++语言过于复杂、庞大，而且安全性差。于是他们设计并开发出一种新的语言——Oak（橡树）。Oak这个名字来源于Green项目组办公室窗外的一棵橡树。由于Oak在进行注册商标时已经被注册，他们需要为这个新语言取一个新的名字，有一天，几位项目的成员正在咖啡馆喝着Java（爪哇）咖啡，其中一个人灵机一动说就叫Java怎么样？马上得到了其他人的同意，于是这个新的语言取名为Java。

Sun在1996年发布了Java 1.0，但是Java 1.0开发的应用速度很慢，并不适合做真正的应用开发，直到Java 1.1后速度有了明显的提升。Java设计之初是为消费类电子产品开发应用，但是真正使Java流行起来是在互联网上的Web应用程序，上个世纪90年代正在互联网发展起步阶段，互联网上设备差别很大，需要应用程序能够跨平台运行，那么Java语言具有“一经编写到处运行”的跨平台能力。

到本书编写时，Oracle公司已经发布了Java 8，Java 9将在2017年秋季发布。Java在20多年发展过程中，与时俱进，为了适应时代的需要，经历过两次重大的版本升级，一个是Java 5，Java 5提供了泛型等重要的功能；另一个是Java 8，Java 8中提供了Lambda表达式和枚举类等重要的功能。

1.2 Java语言特点

Java语言能够流行起来，并长久不衰，得益于Java语言有很多优秀的关键特点。这些特点包括：简单、面向对象、分布式、结构中立、可移植、解释执行、健壮、安全、高性能、多线程和动态。下面详细解释一下：

01. 简单

    Java设计目标之一就是能够方便学习，使用简单。由于当初C++程序员很多，介绍C++语言的书籍也很多，所以Java语言的风格设计成为类似于C++语言风格，但Java摒弃了C++中容易引发程序错误的地方，如指针、内存管理、运算符重载和多继承等。一方面C++程序员可以很快迁移到Java；另一方面没有编程经验的初学者也能很快学会Java。

02. 面向对象

    面向对象是Java最重要的特性。Java是彻底的、纯粹的面向对象语言，在Java中“一切都是对象”。Java完全具有面向对象三个基本特性：封装、继承和多态，其中封装性实现了模块化和信息隐藏，继承性实现了代码的复用，用户可以建立自己的类库。而且Java采用的是相对简单的面向对象技术，去掉了多继承等复杂的概念，只支持单继承。

03. 分布式

    Java语言就是为分布式系统而设计的。JDK(Java Development Kits，Java开发工具包)中包含了支持HTTP和FTP等基于TCP/IP协议的类库。Java程序可以凭借URL打开并访问网络上的对象，其访问方式与访问本地文件系统几乎完全相同。

04. 结构中立

    Java程序需要在很多不同网络设备中运行，这些设备有很多不同类型的计算机和操作系统。为能够使Java程序能在网络的任何地方运行，Java编译器编译生成了与机器结构（CPU和操作系统）无关的字节码（byte-code）文件。任何种类的计算机，只要可以运行Java虚拟机，字节码文件就可以在该计算机上运行。

05. 可移植

    体系结构的中立也使得Java程序具有可移植性。针对不同的CPU和操作系统Java虚拟机有不同的版本，这样就可以保证相同的Java字节码文件可以移植到多个不同的平台上运行。

06. 解释执行

    为实现跨平台，Java设计成为解释执行的，即Java源代码文件首先被编译成为字节码文件，这些字节码本身包含了许多编译时生成的信息，在运行时候Java解释器负责将字节码文件解释成为特定的机器码进行运行。

07. 健壮

    Java语言是强类型语言，它在编译时进行代码检查，使得很多错误能够在编译期被发现，不至于在运行期发生而导致系统崩溃。

    Java摒弃了C++中指针操作，指针是一种很多强大的技术，能够直接访问内存单元，但同时也很复杂，如果指针操控不好，会引起导致内存分配错误、内存泄漏等问题。而Java中则不会出现由指针所导致的问题。

    内存管理方面C/C++等语言采用手动分配和释放，经常会导致内存泄漏，从而导致系统崩溃。

而Java 采用自动内存垃圾回收机制[2]，程序员不再需要管理内存，从而减少内存错误的发生，提高了程序的健壮性。

08. 安全

    在Java程序执行过程中，类装载器负责将字节码文件加载到Java虚拟机中，这个过程中由字节码校验器检查代码中是否存在着非法操作。如果字节码校验器检验通过，由Java解释器负责把该字节码解释成为机器码进行执行，这种检查可以防止木马病毒。

    另外。Java虚拟机采用的是“沙箱”运行模式，即把Java程序的代码和数据都限制在一定内存空间里执行，不允许程序访问该内存空间外的内存。

09. 高性能

    Java编译器在编译时对字节码会进行一些优化，使之生成高质量的代码。Java字节码格式就是针对机器码转换而设计的，实际转换时相当简便。Java在解释运行时采用一种即时编译技术，可使Java程序的执行速度提升很大。多年的发展Java虚拟机也有很多改进这都使得Java程序的执行速度提升很大。

10. 多线程

    Java是为网络编程而设计的，这要求Java能够并发处理多个任务。Java支持多线程编程，多线程机制可以实现并发处理多个任务，互不干涉，不会由于某一任务处于等待状态而影响了其它任务的执行，这样就可以容易的实现网络上的实时交互操作。

11. 动态

    Java应用程序在运行过程中，可以动态的加载各种类库，即使是更新类库也不必重新编译使用这一类库的应用程序。这一特点使之非常适合于网络环境下运行，同时也非常有利于软件的开发。

> [2]在Java运行环境中，始终存在着一个系统级的线程，专门跟踪内存的使用情况，定期检测出不再使用的内存，并进行自动回收，避免了内存的泄露，也减轻了程序员的工作量。

1.3 Java平台

Java不仅是编程语言，还是一个开发平台，Sun公司根据Java应用领域的不同将Java分成三个平台：Java SE、Java EE和Java ME。

1.3.1 Java SE

Java SE是Java Standard Edition，主要目的是为台式机和工作站桌面应用（Application）程序的版本。Java SE是其他平台的基础，本书主要介绍的就是Java SE版本中的技术。

Java SE中主要包含了：JRE（Java SE Runtime Environment，Java SE运行环境）、JDK（Java Development Kit，Java开发工具包）和Java核心类库。如果只是运行Java程序，不考虑开发Java程序，那么只安装JRE就可以了。在JRE中包含了Java程序运行所需要的Java虚拟机（JVM，Java Virtual Machine）。JDK中包含了JRE和一些开发工具，这些工具包括：编译器、文档生成器和文件打包等工具。

另外，Java SE中还提供了Java应用程序开发需要的基本的和核心的类库，这些类库：字符串、集合、输入输出、网络通信和图形用户界面等。事实上学习Java就是在学习Java语法和Java类库使用。

1.3.2 Java EE

Java EE是Java Enterprise Edition，主要目的是为简化企业级系统的开发、部署和管理。Java EE是以Java SE为基础的，并提供了一套服务、API接口和协议，能够开发企业级分布式系统、Web应用程序和业务组件等，其中的包括：JSP、Servlet、EJB、JNI和Java Mail等。

1.3.3 Java ME

Java ME是Java Micro Edition，主要是面向消费类电子产品，为消费电子产品提供一个Java的运行平台，使得Java程序能够在手机、机顶盒、PDA等产品上运行。Java ME在早期的诺基亚塞班手机系统有很多应用，而现在的iOS和Android等智能手机中基本上没有它的用武之地。

1.4 Java虚拟机

Java应用程序能够跨平台运行，主要是通过Java虚拟机实现的。如图1-1所示，不同软硬件平台Java虚拟机是不同的，Java虚拟机往下是不同的操作系统和CPU，使用或开发时需要下载不同的JRE或JDK版本。Java虚拟机往上是Java应用程序，Java虚拟机屏蔽了不同软硬件平台，Java应用程序不需要修改，不需要重新编译直接可以在其他平台上运行。

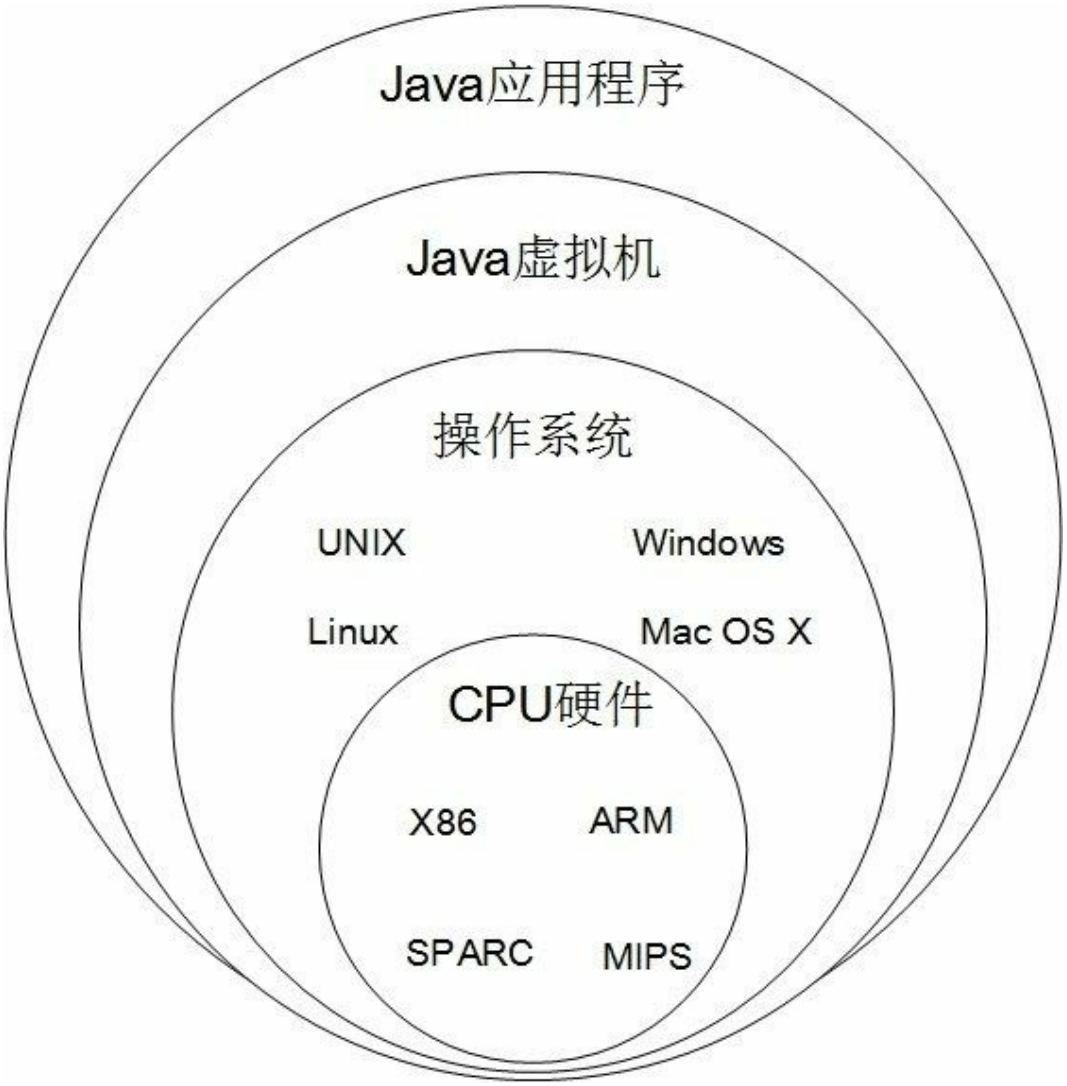


图**1-1 Java**虚拟机

Java虚拟机中包含了Java解释器，Java程序运行过程如图1-2所示，首先由编译器将Java源程序文件（.java文件）编译成为字节码文件（.class文件），然后再由Java虚拟机中的解释器将字节码解释成为机器码去执行。

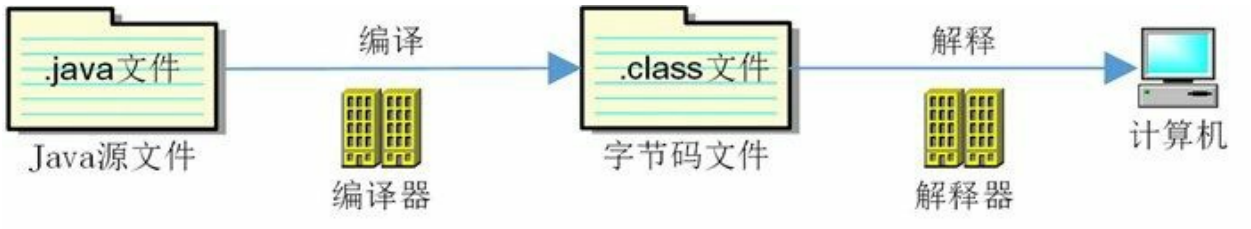


图**1-2　Java**程序运行过程

本章小结

本章首先介绍了解到Java的历史、Java语言的特点，然后介绍了Java三大平台，最后介绍了Java虚拟机。

第 2 章　开发环境搭建

《论语·魏灵公》曰：“工欲善其事，必先利其器”，做好一件事，准备工作非常重要。在开始学习Java技术之前，先介绍如何搭建Java开发环境是非常重要的一件事。

Oracle公司提供的JDK只是一个开发工具包，它不是一个IDE（Integrated Development　Environments，集成开发环境），IDE的开发工具将程序的编辑、编译、调试、执行等功能集成在一个开发环境中，使用户可以很方便地进行软件的开发，Java开发IDE工具有很多，其中主要有：Eclipse、IntelliJ IDEA和NetBeans等。

2.1 JDK工具包

JDK工具包是最基础的Java开发工具，很多Java IDE工具，如：Eclipse、IntelliJ IDEA和NetBeans等都依赖于JDK。也有一些人使用“JDK+文本编辑工具”编写Java程序。

2.1.1 JDK下载和安装

截止本书编写完成为止，Oracle公司对外发布的最新JDK 8。图2-1所示是JDK 8下载界面，它的下载地址是http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html。其中有很多版本，支持的操作系统有Linux、Mac OS X[1]、Solaris[2]和Windows。注意选择对应的操作系统，以及32位还是64位安装的文件。

[1]苹果桌面操作系统，基于UNIX操作系统，现在改名为macOS。

[2]原Sun公司UNIX操作系统，现在被Oracle公司收购。

如果你的电脑是Windows 10 64位系统，则首先选中Accept License Agreement（同意许可协议），然后单击jdk-8u131-windows-x64.exe下载JDK文件。

图**2-1** 下载**JDK8**

下载完成后就可以安装了，双击jdk-8u131-windows-x64.exe文件就可以安装了，安装过程中会弹出如图2-2所示的内容选择对话框，其中“开发工具”是JDK内容；“源代码”是安装Java SE源代码文件，如果安装源代码，安装完成后会见如图2-3所示的src.zip文件就是源代码文件；公共JRE就是Java运行环境了，

这里可以不安装，因为JDK文件夹中也会有一个JRE，如图2-3所示的jre文件夹。

图**2-2**　安装内容选择对话框

图**2-3** **JDK**安装后的内容

2.1.2 设置环境变量

完成之后，需要设置环境变量，主要包括：

01. JAVA_HOME环境变量，指向JDK目录，很多Java工具运行都需要的JAVA_HOME环境变量，所以笔者推荐大家添加这变量。

02. 将JDK\bin目录添加到Path环境变量中，这样在任何路径下都可以执行JDK提供的工具指令。

首先需要打开Windows系统环境变量设置对话框，打开该对话框有很多方式，如果Windows 10系统，则打开步骤是：右击屏幕左下角的Windows图标，单击“系统”菜单，然后弹出如图2-4所示的Windows系统对话框，单击左边的“高级系统设置”超链接，打开如图2-5所示的高级系统设置对话框。

图**2-4　Windows**系统对话框

图**2-5**　高级系统设置对话框

在如图2-5所示的高级系统设置对话框中，单击“环境变量”按钮打开环境变量设置对话框，如图2-6所示，可以在用户变量（上半部分，只配置当前用户）或系统变量（下半部分，配置所有用户）添加环境变量。一般情况下，在用户变量中设置环境变量。

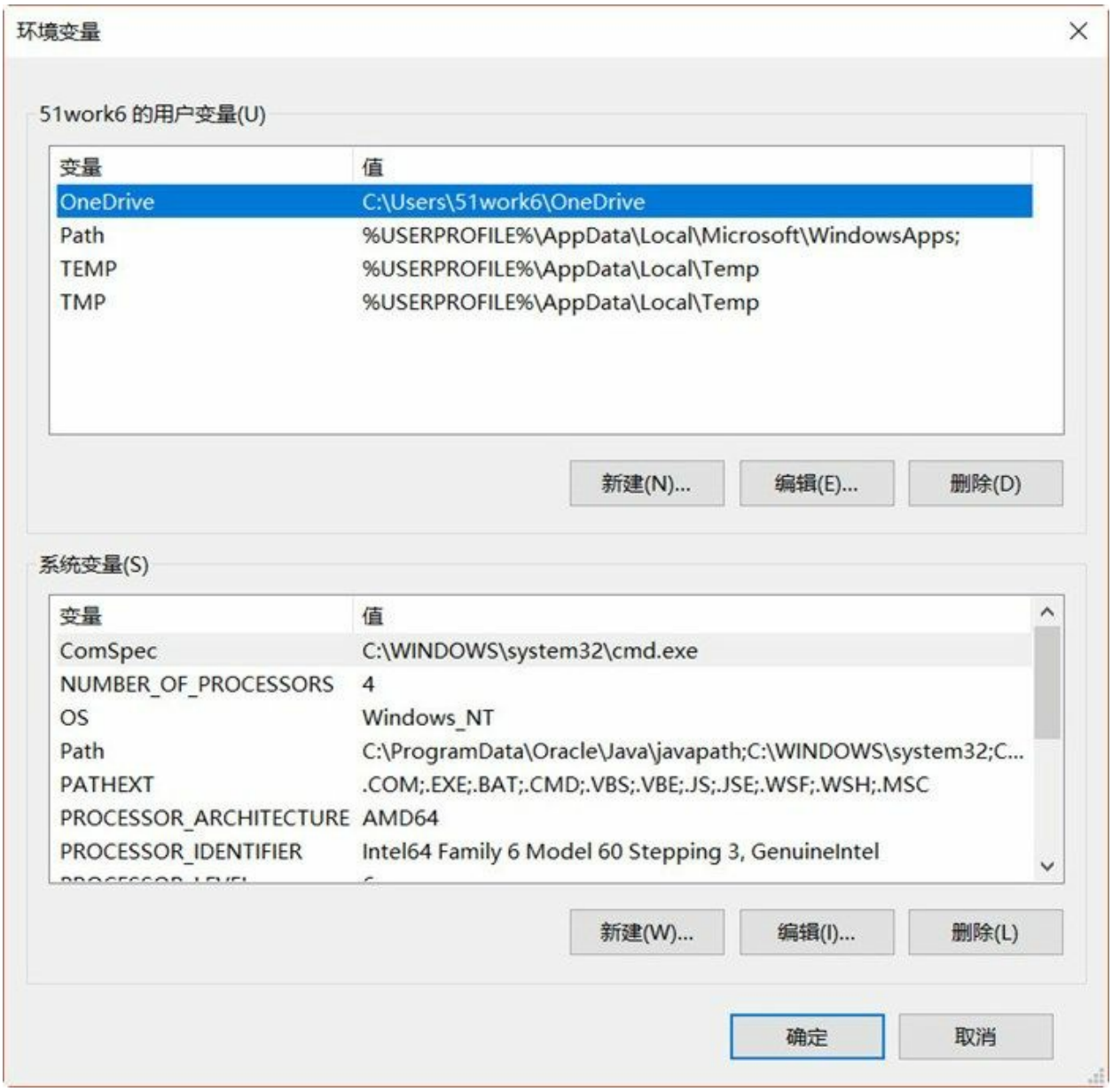


图**2-6**　环境变量设置对话框

在用户变量部分单击“新建”按钮，系统弹出对话框，如图2-7所示。设置“变量名”设置为JAVA_HOME，“变量值”设置为JDK安装路径。最后单击“确定”按钮完成设置。

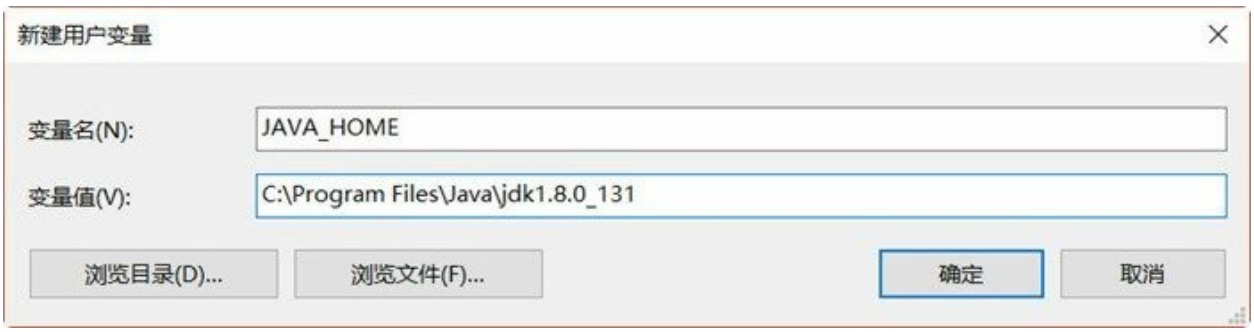


图**2-7**　设置**JAVA_HOME**

然后追加Path环境变量，在用户变量中找到Path，双击Path弹出Path变量对话框，如图2-8所示，追加%JAVA_HOME%\bin。注意多个变量路径之间用“;”（分号）分隔。最后单击“确定”按钮完成设置。

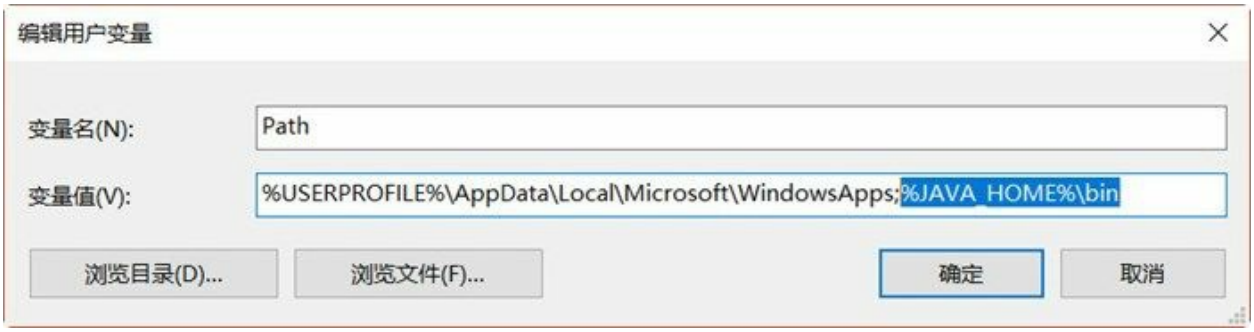


图**2-8**　添加**Path**变量对话框

下面测试一下环境设置是否成功，可以通过在命令提示行中输入javac指令，看是否能够找到该指令，如图2-9所示，则说明环境设置成功。

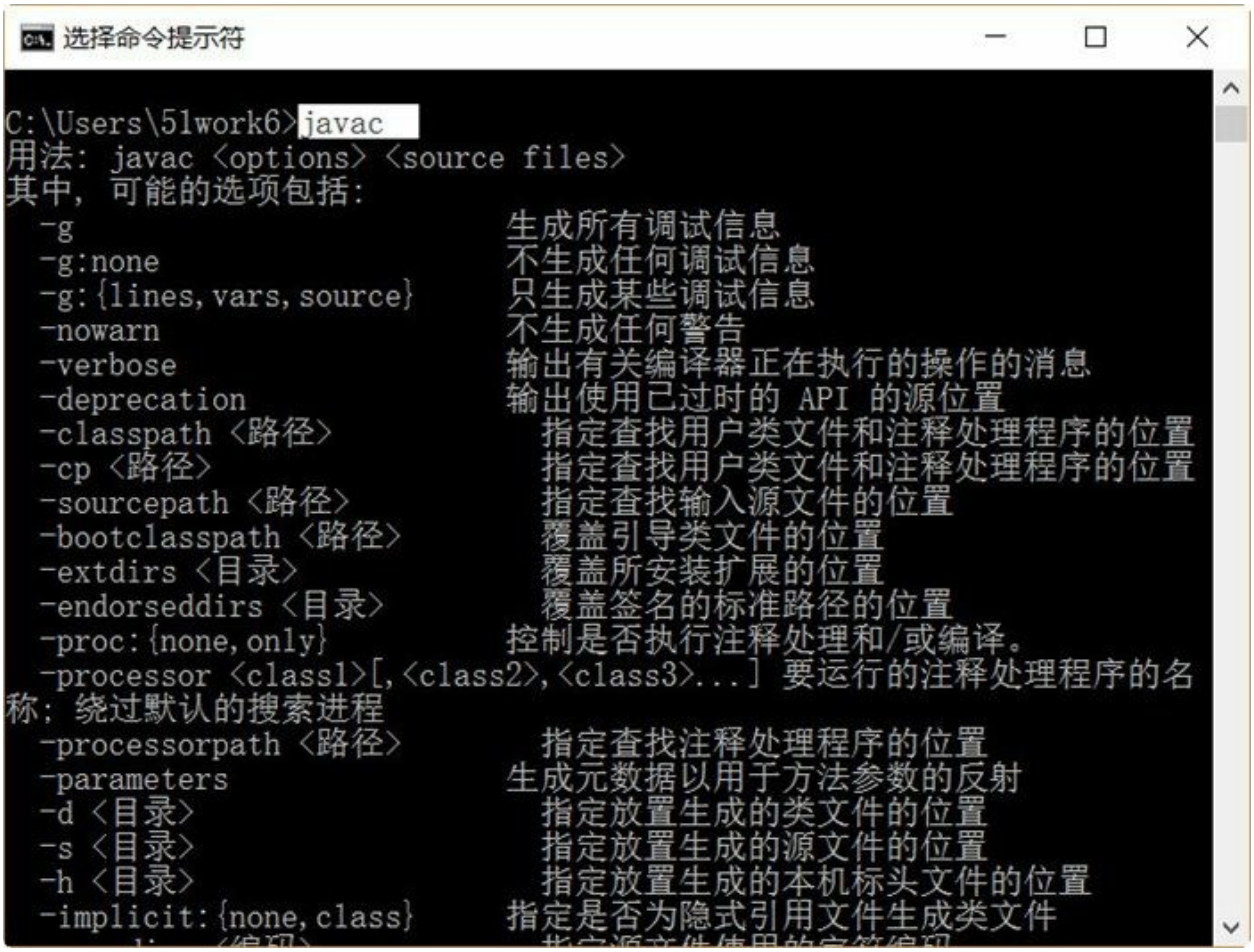


图**2-9**　通过命令提示行测试环境变量

提示　打开命令行工具，也可以通过右击屏幕左下角的Windows图标，单击“命令提示符”菜单

实现。

2.2 Eclipse开发工具

Eclipse是著名的跨平台IDE工具，最初Eclipse是IBM支持开发的免费Java开发工具，2001年11月贡献给开源社区，现在它由非营利软件供应商联盟Eclipse基金会管理。Eclipse的本身也是一个框架平台，它有着丰富的插件，例如C++、Python、PHP等开发其他语言的插件。另外，Eclipse是绿色软件不需要写注册表，卸载非常方便。

2.2.1 Eclipse下载和安装

本书采用Eclipse 4.6[3]版本作为IDE工具，Eclipse 4.6下载地址是http://www.eclipse.org/downloads/，如图2-10所示是Windows系统的下载Eclipse下载页面，单击“DOWNLOAD 64 bit”按钮页面会跳转到，如图2-11所示的选择下载镜像地址页面，单击Select Another Mirror连接可以改变下载镜像地址，然后单击DOWNLOAD按钮开始下载。

[3]Eclipse 4.6开发代号是Neon（氖气），Eclipse开发代号的首字母是按照字母顺序排列的。Eclipse 4.7开发代号是Oxygen（氧气）。

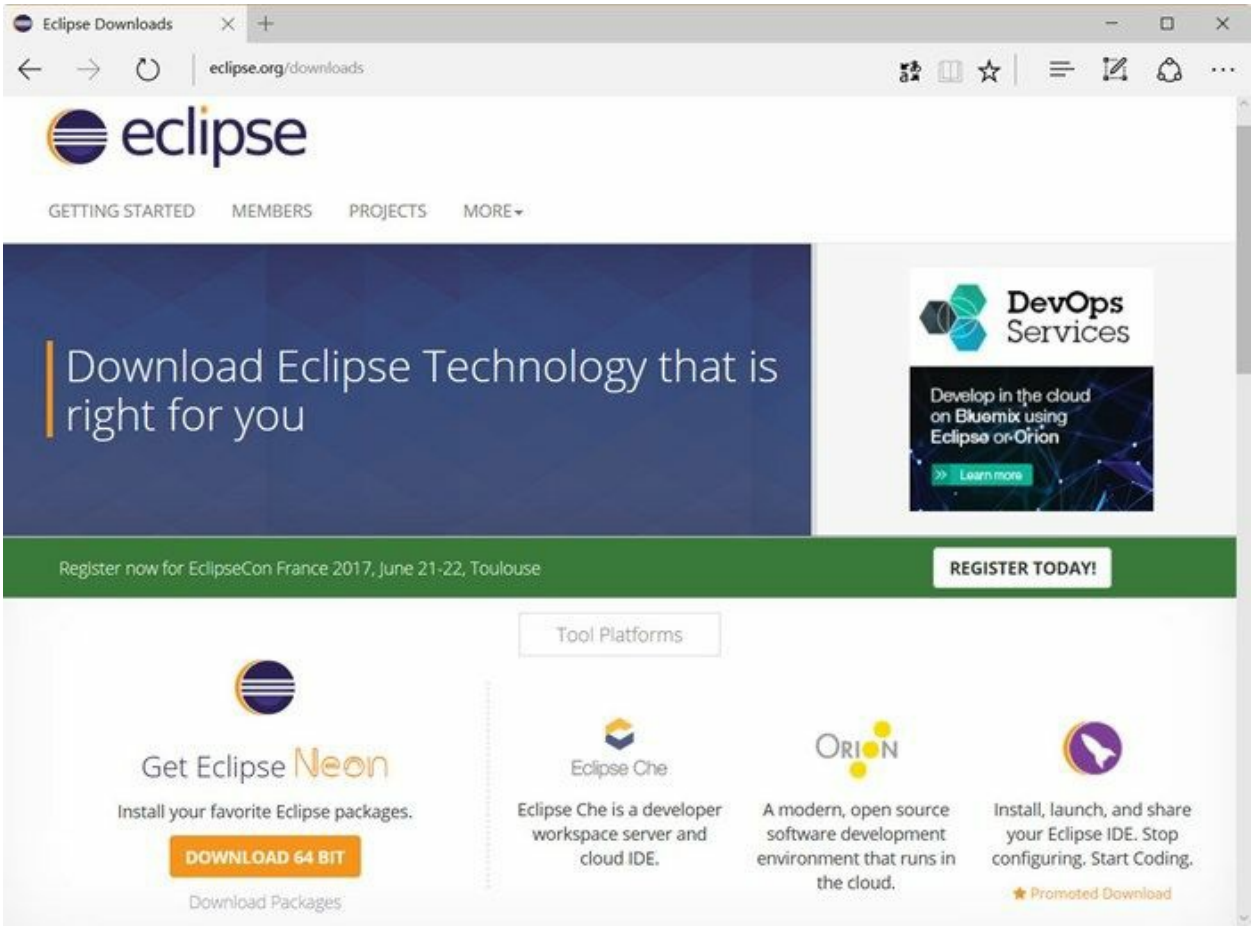


图2-10 Eclipse 4.6下载页面

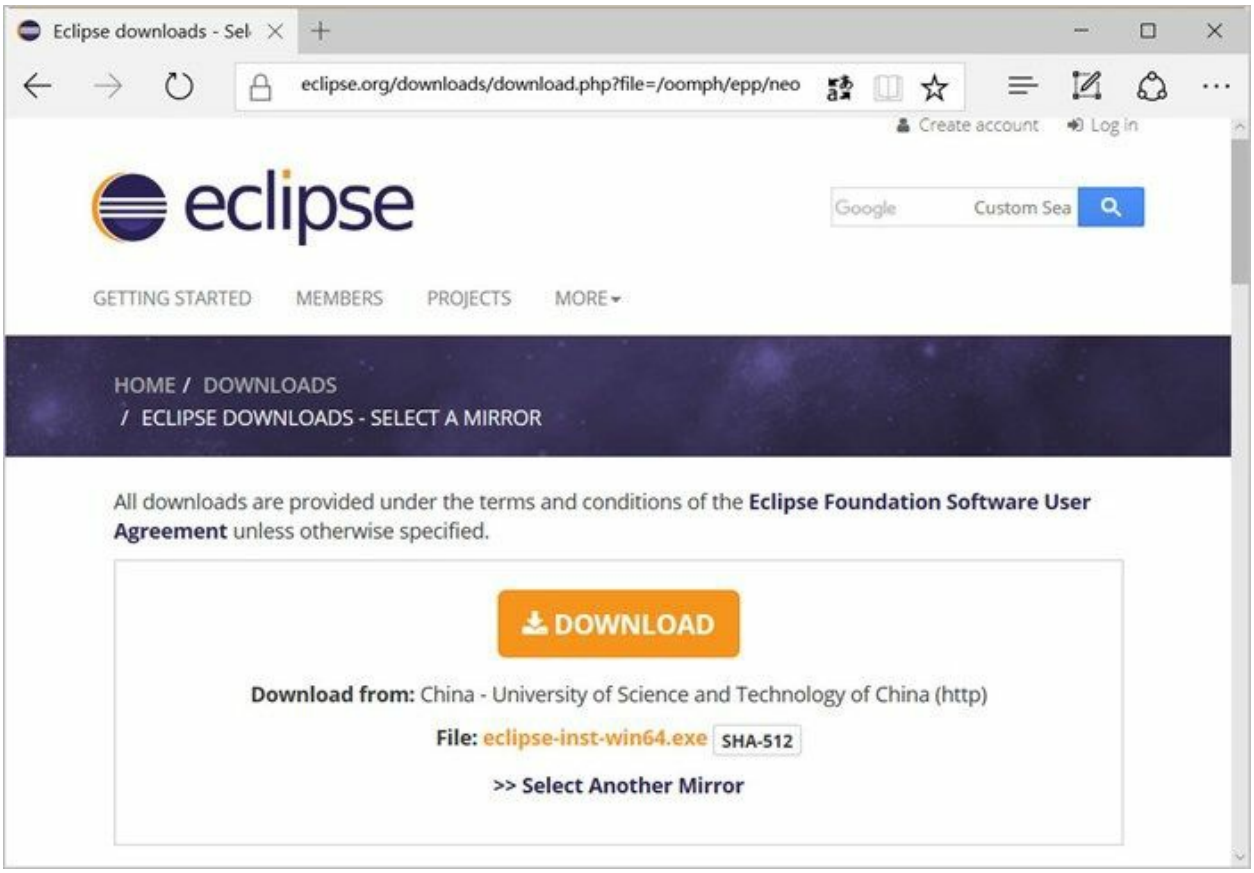


图**2-11**　选择下载镜像地址

下载完成后的文件是eclipse-inst-win64.exe，事实上eclipse-inst-win64.exe是安装各种Eclipse版本客户端，双击eclipse-inst-win64.exe弹出如图2-12所示的界面，选择Eclipse IDE for Java Developers进入如图2-13所示的界面，在该界面中Installation Folder可以改变安装目录，选中create start menu entry可以添加快捷方式到开始菜单，选中create desktop shortcut可以在桌面创建快捷方式，设置完成后单击INSTALL按钮开始安装，安装完成如图2-14所示，单击LAUNCH按钮启动Eclipse。

图**2-12**　安装各种**Eclipse**版本客户端

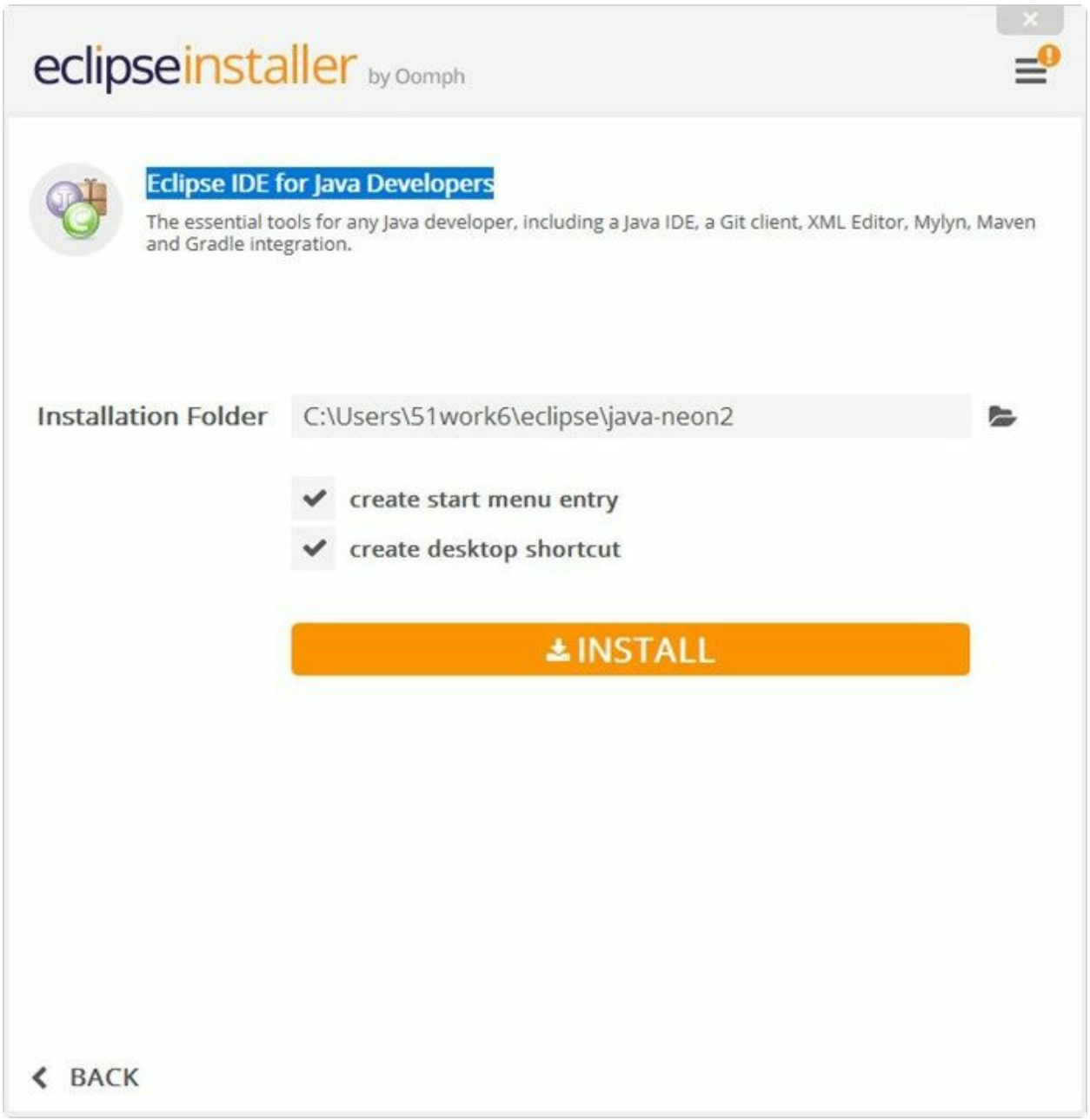


图**2-13　Eclipse**安装

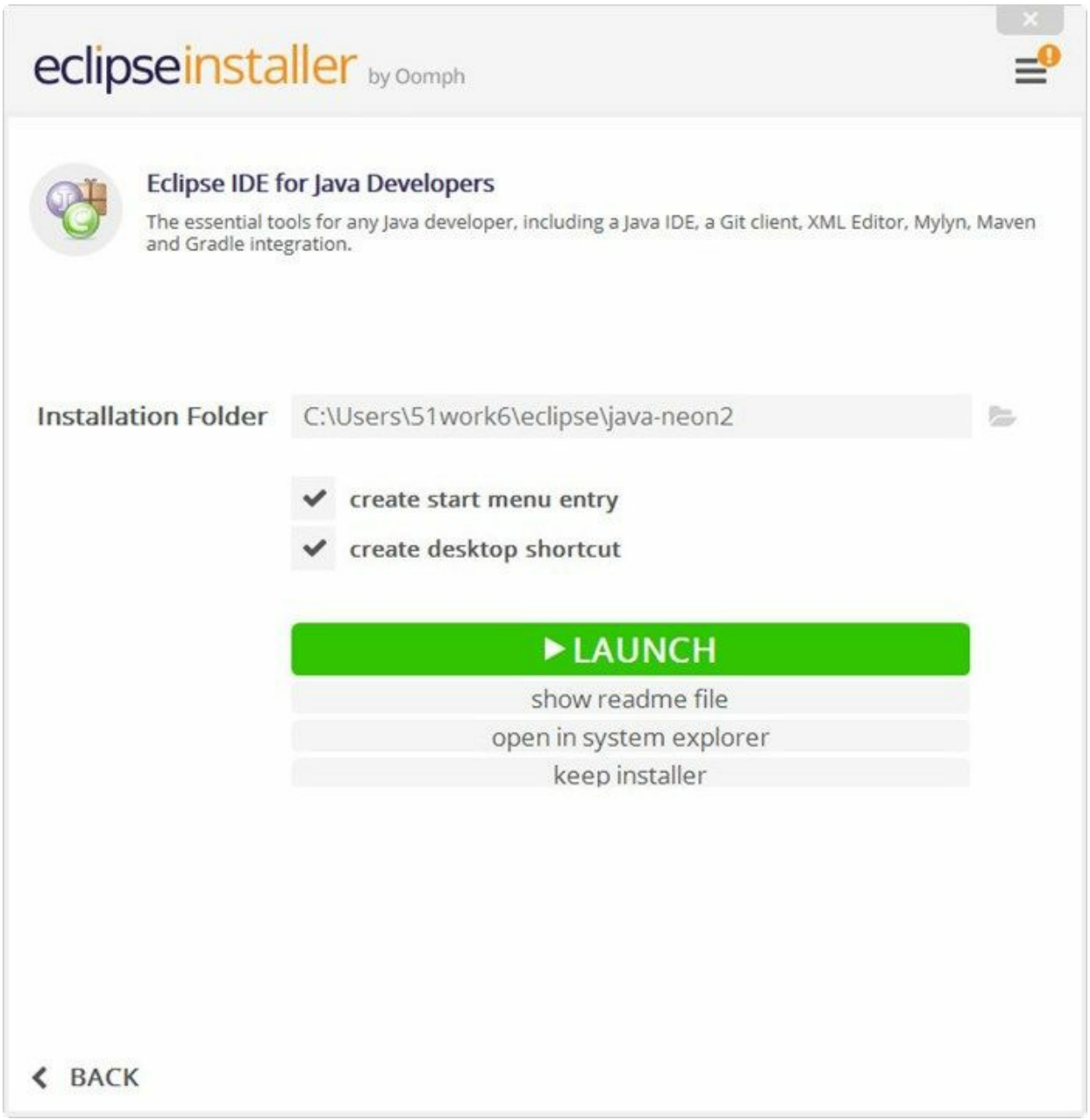


图**2-14　Eclipse**安装完成

在Eclipse启动过程中，会弹出如图2-15所示，选择工作空间（workspace）对话框，工作空间是用来保存工程的目录。默认情况下每次Eclipse启动时候都需要选择工作空间，如果你觉得每次启动时都选择工作空间比较麻烦，可以选中Use this as the default and to not ask again选项，设置工作空间默认目录。初次启动Eclipse成功后，会进入如图2-16所示的欢迎界面。

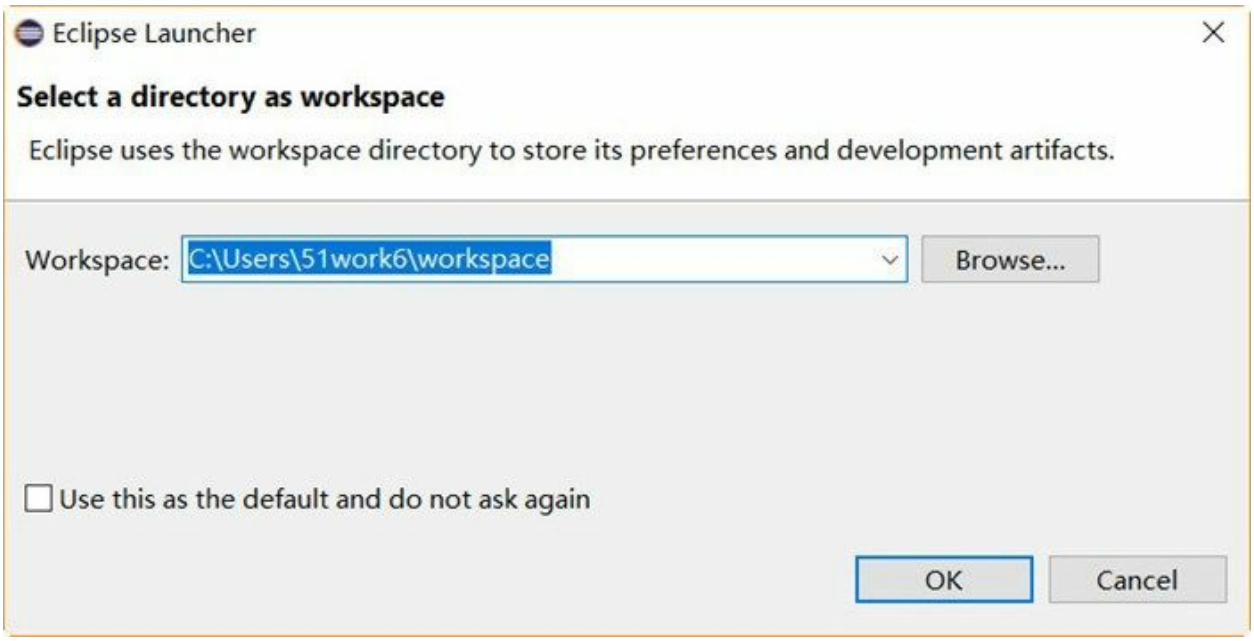


图**2-15**　选择工作空间

图**2-16**　**Eclipse**欢迎界面

2.2.2 安装中文语言包

Eclipse界面默认是英文，对于一些初学者英文界面使用起来还是有一定困难的。Eclipse平台提供了一个语言包项目——Eclipse Babel Project（http://www.eclipse.org/babel/），Babel是一个插件，安装Babel插件可以通过离线或在线安装，Babel 插件下载地址是http://www.eclipse.org/babel/downloads.php，如图2-17所示，单击Zipped p2 repository for Neon超链接下载离线包，注意离线包所支持的Eclipse版本。笔者推荐在线安装，从图2-17所示页面中可见在线安装网址是http://download.eclipse.org/technology/babel/update-site/R0.14.1/neon。

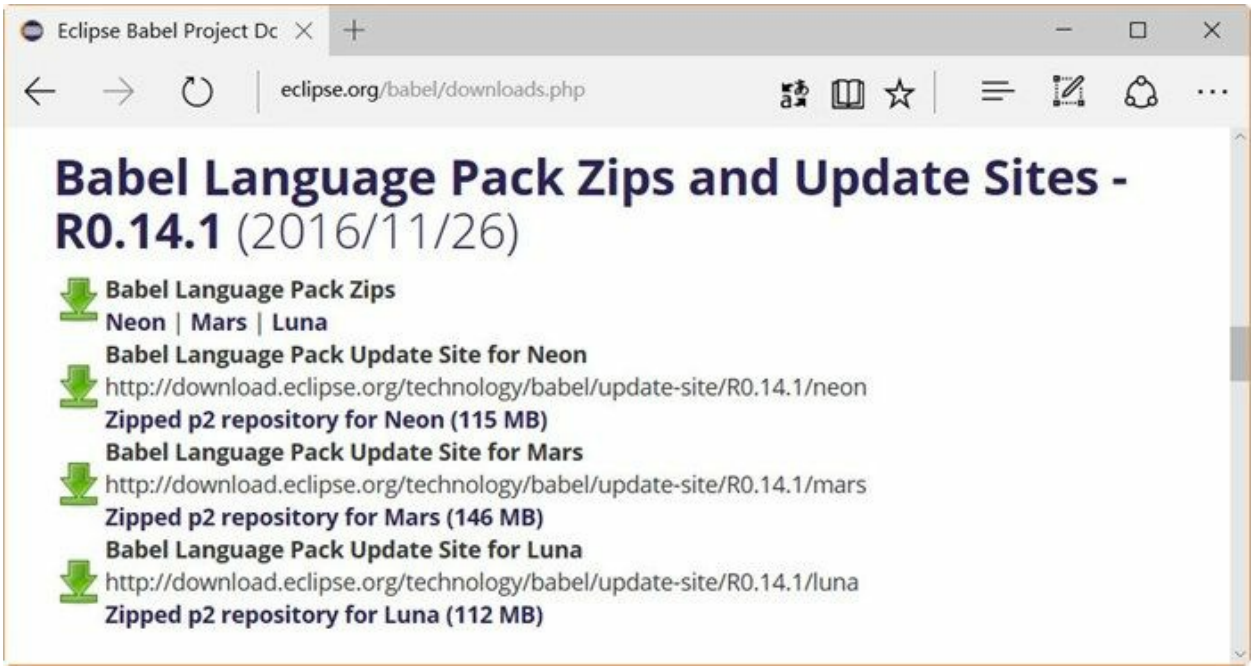


图**2-17** 下载**Eclipse**语言包

安装插件过程如下，首先启动Eclipse，选择菜单Help→Install New Software弹出如图2-18所示的对话框。单击Add按钮弹出如图2-19所示对话框，在Location中输入插件在线地址http://download.eclipse.org/technology/babel/update-site/R0.14.1/neon，如图2-20所示。

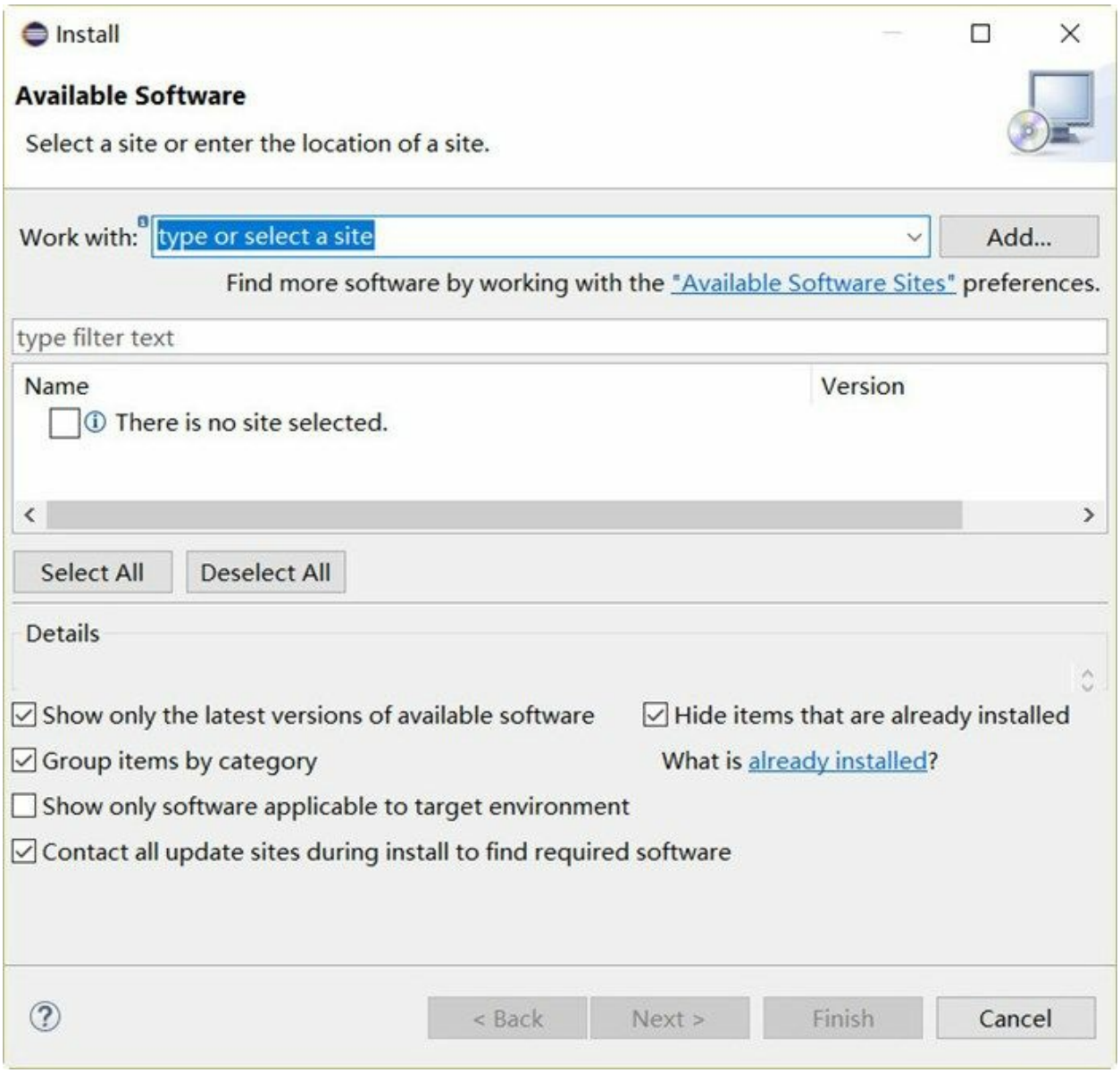

Install
Available Software
Select a site or enter the location of a site.
Work with:
type or select a site
Add...
Find more software by working with the "Available Software Sites" preferences.
type filter text
Name
Version
There is no site selected.
Select All
Deselect All
Details
Show only the latest versions of available software
Hide items that are already installed
Group items by category
What is already installed?
Show only software applicable to target environment
Contact all update sites during install to find required software
< Back
Next >
Finish
Cancel

图**2-18**　安装插件

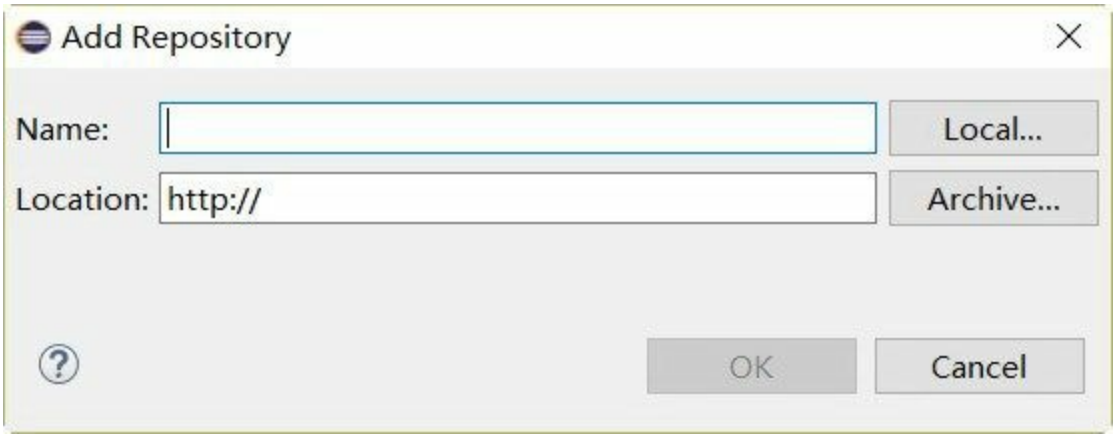

Add Repository
Name:
Local...
Location:
http://
Archive...
OK
Cancel

图**2-19**　插件地址

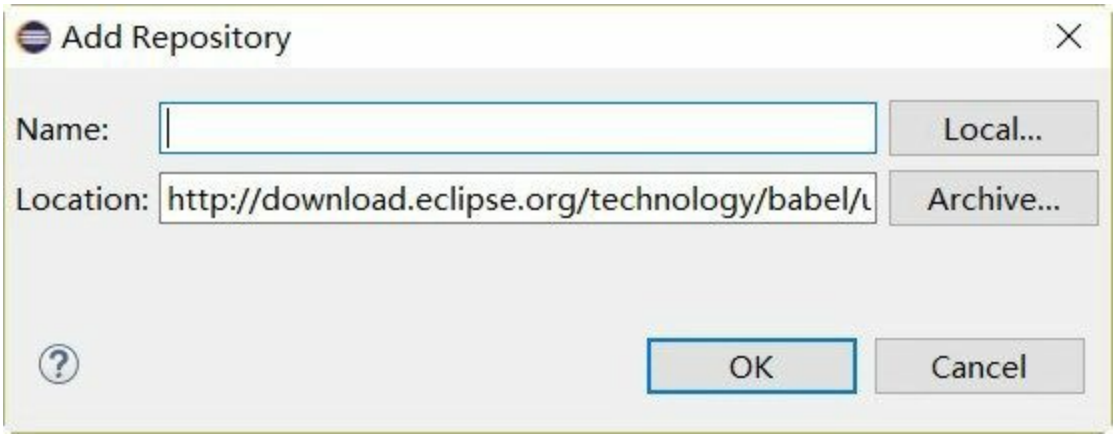


图**2-20**　输入插件地址

确定输入内容后单击OK按钮关闭对话，Eclipse通过刚刚输入的网址查找插件，如果能够找到插件，则出现如图2-21所示对话框，从中选择简体中文语言包。选择完成后单击Next按钮进行安装，安装过程需要从网上下载插件，这个过程需要等一段时间。

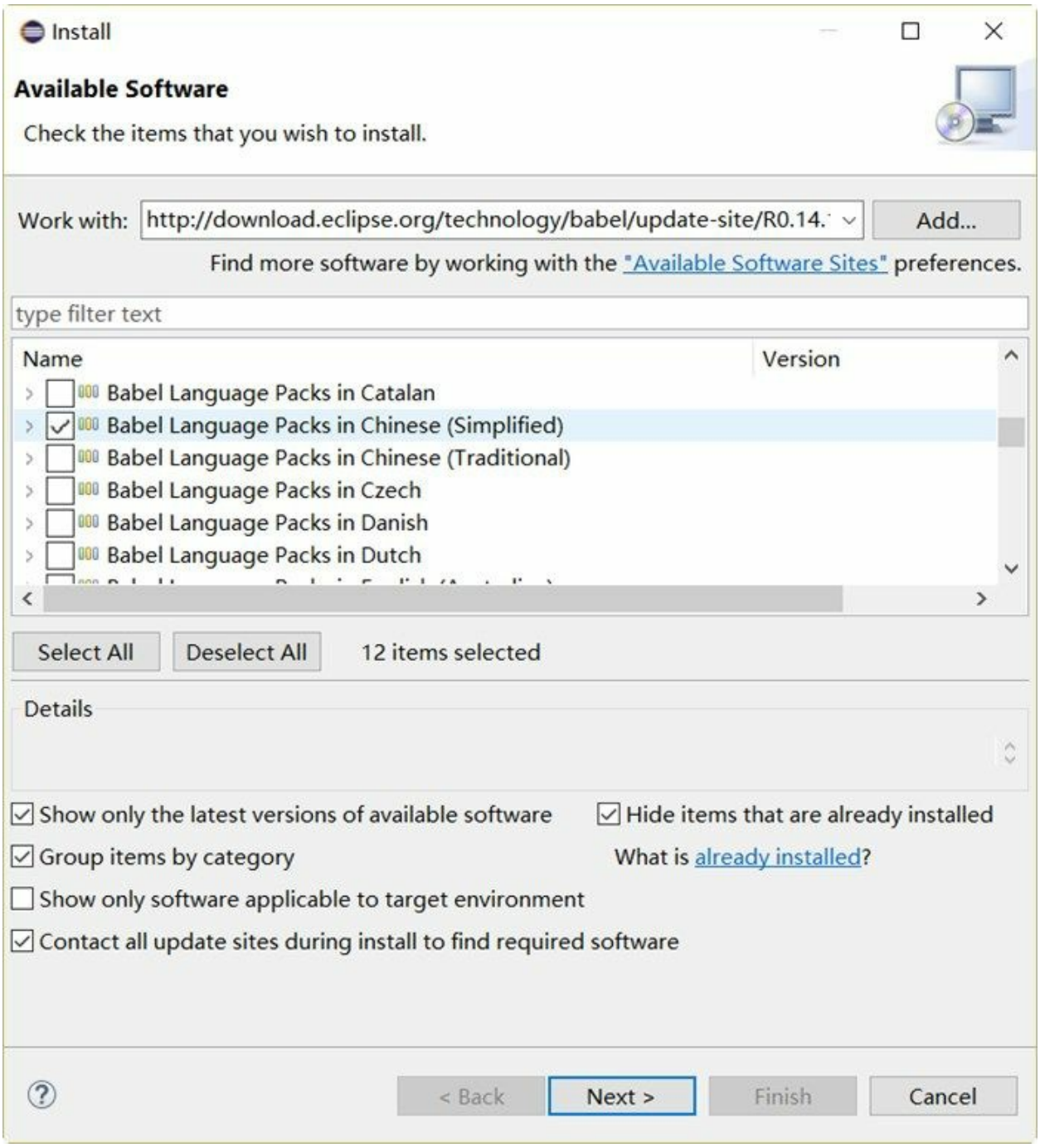


图**2-21**　选择简体中文语言包

安装简体中文语言包插件后重新启动Eclipse，界面如图2-22所示。

图**2-22**　安装简体中文语言包后**Eclipse**

2.2.3　**Eclipse**界面

关闭Eclipse的“欢迎”界面，并创建一个Java工程后（如何创建Java工程将在第3章介绍），可以看到如图2-23所示的主界面。该界面主要分成4个区域：

①号区域是包资源管理器视图，以包形式管理Java源文件，包是一种命名空间将在后面再详细介绍。

②号区域是代码编辑视图，编码工作就是在这里完成的。

③号区域是显示大纲等辅助视图，大纲视图中列出了当前Java类中方法和成员变量，并且单击可以快速导航到指定代码。

④号区域是显示问题、控制台等辅助视图，问题可以列出当前工程的编译错误和警告等问题。

事实上，这4个区域视图都可以互换，只要拖曳视图标题到相应的区域。Eclipse视图标题如图2-24所示，标题的右端有两个按钮：最小化按钮和最大化按钮，单击可以实现视图的最小化和最大化显示。

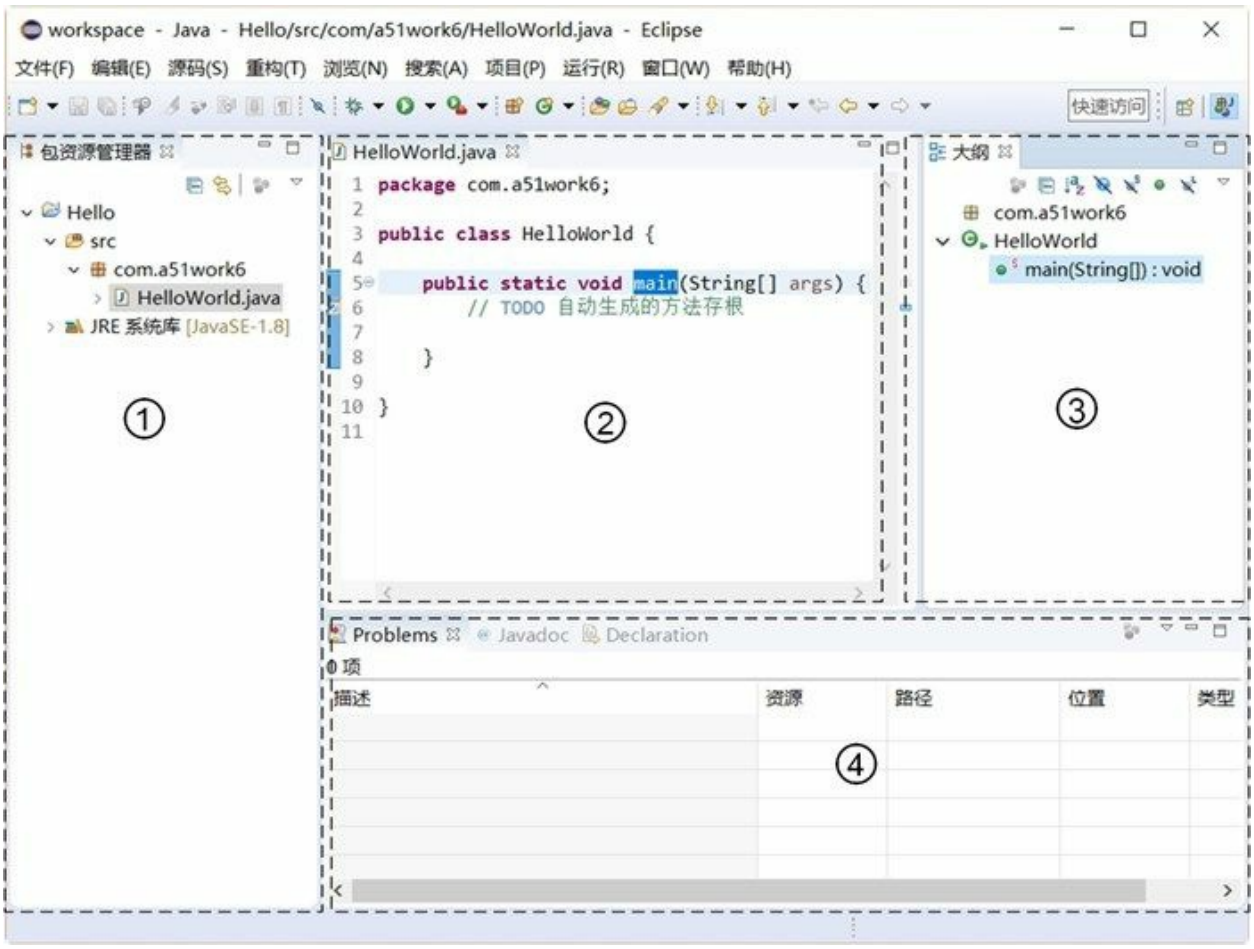


图**2-23** **Eclipse**主界面

图**2-24** **Eclipse**视图

此外，Eclipse提供了丰富的菜单和工具栏，随着学习的深入本书会有重点地介绍，这里不再赘述。

2.2.4 Windows系统中常用快捷键

一个优秀的IDE开发工具应该提供丰富的快捷键，快捷键虽然不能完全替代鼠标操作，但却可以锦上添花。由于Eclipse工具提供很多快捷键，本书不打算介绍全部的快捷键，笔者总结了一些Eclipse工具在Windows系统常用的快捷键，如表2-1所示。

表 2-1 Eclipse在Windows系统常用快捷键

作用域	快捷键	功能
全局	Ctrl+M	最大化/最小化当前视图
全局	Ctrl+=	放大视图
全局	Ctrl+-	缩小视图
文本编辑器	Ctrl+F	查找并替换
文本编辑器	Ctrl+L	转至某行
Java 编辑器	Ctrl+Shift+F	代码格式化
Java 编辑器	Ctrl+/	注释/取消注释当前行
Java 编辑器	Ctrl+Shift+M	添加导入包
Java 编辑器	Ctrl+Shift+O	组织导入包
Java 编辑器	Ctrl+Shift+↑	转至上一个成员
Java 编辑器	Ctrl+Shift+↓	转至下一个成员
Java 编辑器	Ctrl+B	重新编译 Java 程序代码
Java 编辑器	Ctrl+F11	运行上次程序

这些快捷键只是冰山一角，想了解更多Eclipse在Windows系统常用快捷键，读者可以参考http://baike.baidu.com/item/Eclipse快捷键指南。

2.3 其他开发工具

Java IDE开发工具除了Eclipse当然还有很多，其中被广泛认可的还有IntelliJ IDEA和NetBeans，令人惊奇的是它们都源自捷克人之手。

2.3.1 IntelliJ IDEA

虽然IntelliJ IDEA市场份额不如Eclipse，但是被很多Java专家认为是最优秀的Java IDE开发工具。IntelliJ IDEA是Jetbrains公司（www.jetbrains.com）研发的一款Java IDE开发工具，Jetbrains是一家捷克公司，该公司开发的很多工具都好评如潮，如图2-25所示Jetbrains开发的工具，这些工具可以编写C/C++、C#、DSL、Go、Groovy、Java、JavaScript、Kotlin、Objective-C、PHP、Python、Ruby、Scala、SQL和Swift语言。

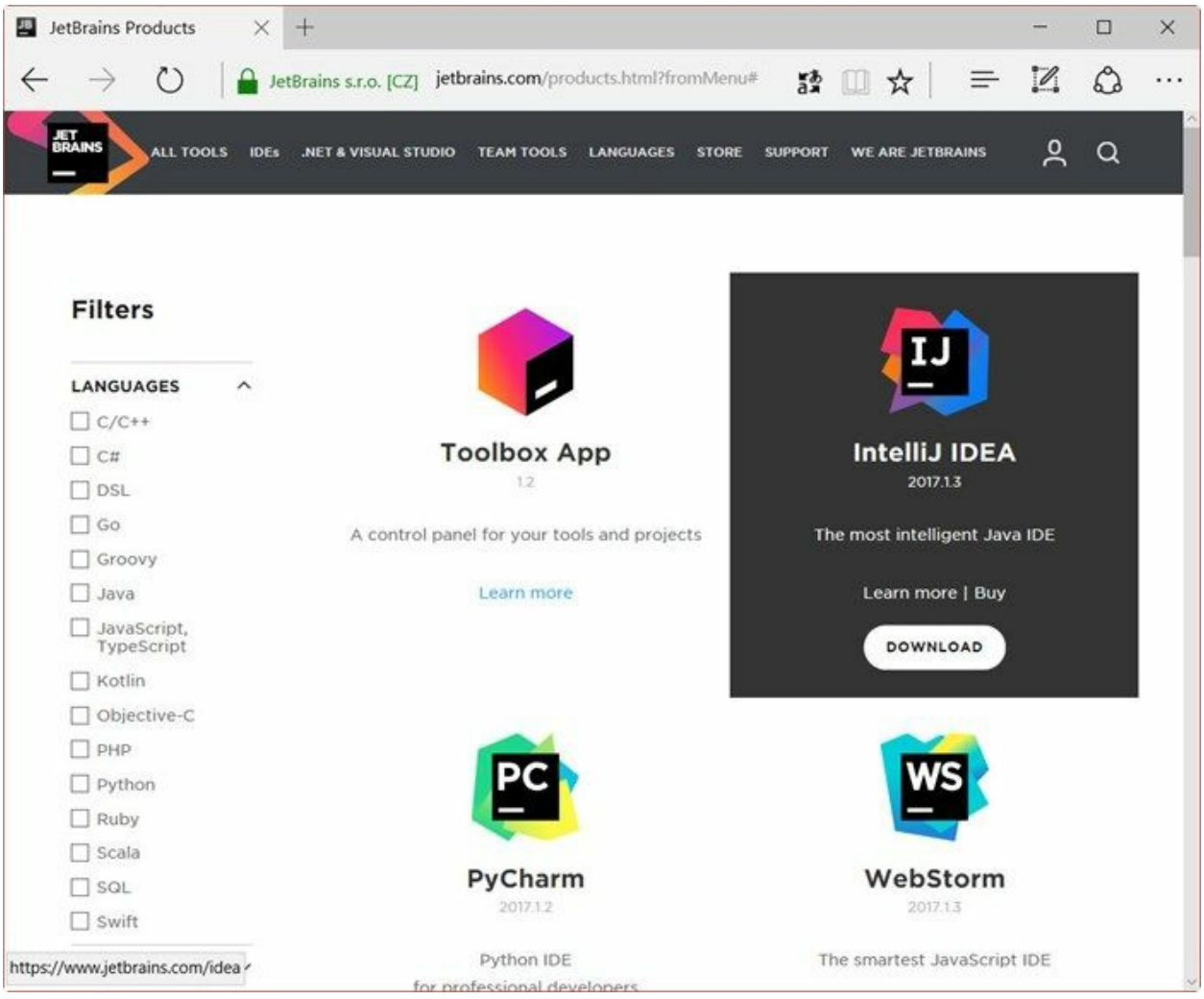


图**2-25** **Jetbrains**公司工具

IntelliJ IDEA下载地址是https://www.jetbrains.com/idea/download/，如图2-26所示页面可以见，IntelliJ IDEA有两个版本：Ultimate（旗舰版）和Community（社区版）。旗舰版是收费的，可以免费试用30天，如果超过30天，则需要购买软件许可(License key)。社区版是完全免费的，对于学习Java语言社区版已经足够了。在图2-26页面下载IntelliJ IDEA工具，完成之后即可安装了。

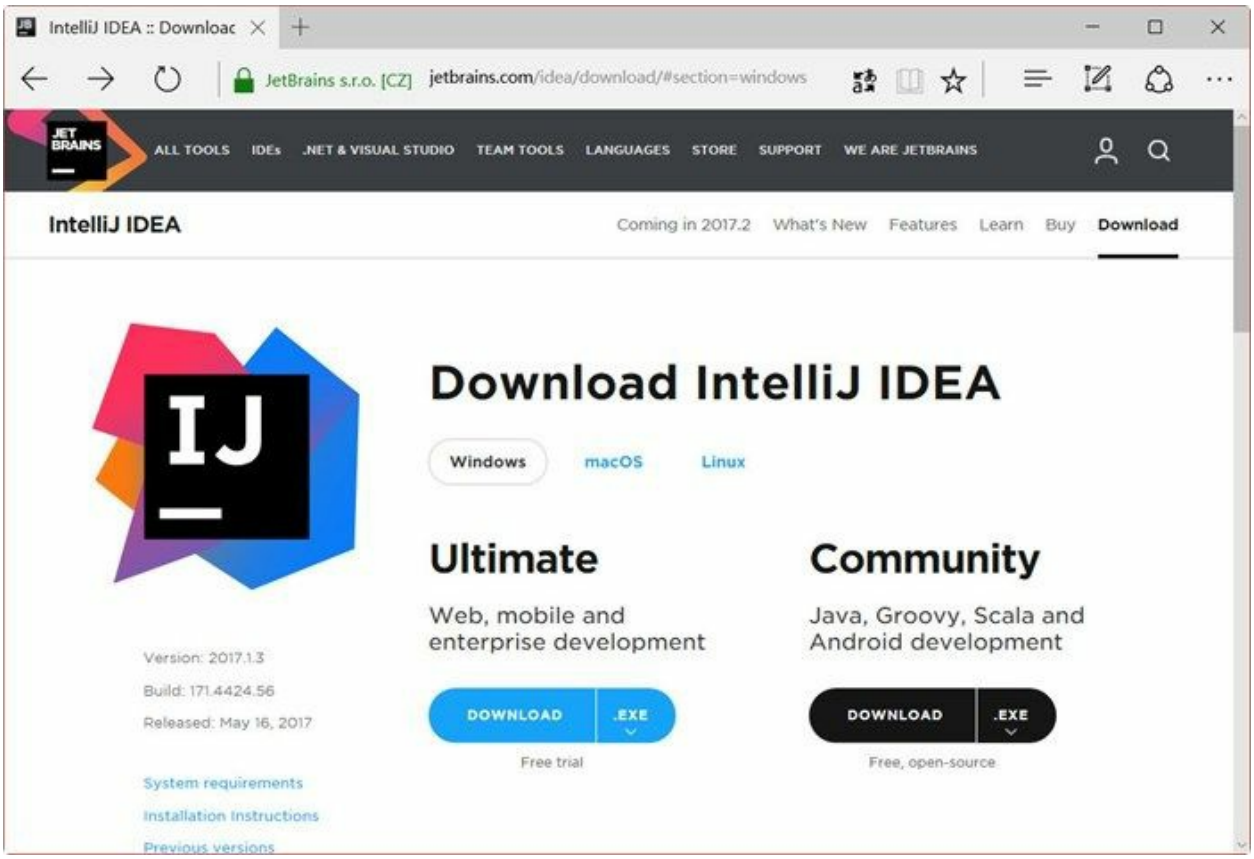


图**2-26**　下载**IntelliJ IDEA**

IntelliJ IDEA工具使用起来比较复杂，而且用户群少，因此IntelliJ IDEA具体使用细节，本书不再介绍。

2.3.2　NetBeans IDE

NetBeans是一个始于捷克布拉格查理大学的一个学生项目（Xelfi计划），Xelfi计划延伸发展成为NetBeans IDE工具，1999年被Sun公司收购，后来随着Oracle公司收购Sun公司NetBeans IDE成为了Oracle工具产品。

被Oracle收购后NetBeans IDE仍然是免费工具，下载网址https://netbeans.org/downloads/，打开页面如图2-27所示，可以NetBeans IDE支持的平台有Windows、Mac OS X和Linux等，除完全支持所有 Java 平台（Java SE、Java EE、Java ME 和 JavaFX）之外，还支持PHP、HTML5、JavaScript、Groovy和C/C++等语言。在图2-27页面选择适合自己的版本下载NetBeans IDE工具，完成之后需要安装了。

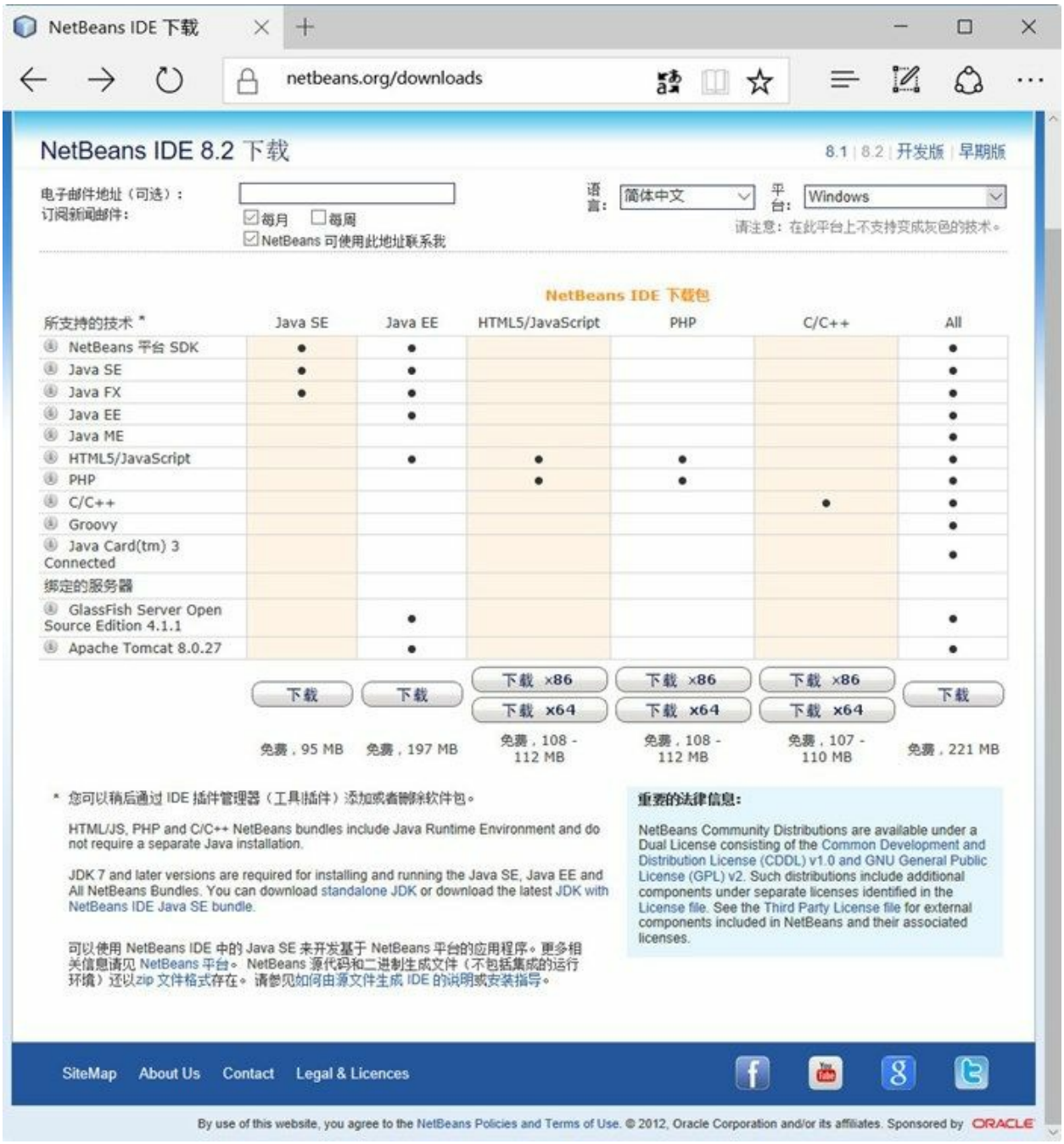


图**2-27　NetBeans IDE**下载页面

NetBeans IDE工具用户群比较少，因此NetBeans IDE具体使用细节，本书不再介绍。

2.3.3　文本编辑工具

IDE开发工具提供了强大开发能力，提供了语法提示功能，但对于学习Java的学员而言语法提示并不是件好事，笔者建议初学者采用文本编辑工具+JDK学习。开发过程就使用文本编辑工具编写Java源程序，然后使用JDK提供的javac指令编译Java源程序，再使用JDK和JRE提供的java指令运行。

提示　javac和java等指令需要在命令提示行中执行，打开命令行参考2.1.2节。

Windows平台下的文本编辑工具有很多，常用如下：

- 记事本：Windows平台自带的文本编辑工具，关键字不能高亮显示。
- UltraEdit：历史悠久强大的文本编辑工具，可支持文本列模式等很多有用的功能，官网www.ultraedit.com。
- EditPlus：历史悠久强大的文本编辑工具，小巧、轻便、灵活，官网www.editplus.com。
- Sublime Text：近年来发展和壮大的文本编辑工具，所有的设置没有图形界面，在JSON格式[4]的文件中进行的，初学者入门比较难，官网www.sublimetext.com。

> [4]JSON(JavaScript Object Notation, JS对象标记) 是一种轻量级的数据交换格式，采用键值对形式，如：{"firstName": "John"}。

除了记事本工具外，其他的UltraEdit、EditPlus和Sublime Text等工具都可以与JDK集成起来，能够在这些工具中直接，执行JDK指令。

下面重点介绍一下EditPlus与JDK集成过程。首先，打开启动EditPlus打开菜单“工具”→“首选项”，弹出首选项对话框，如图2-28所示，选择“工具”→“自定义工具”，在“自定义工具组及项目”中选择Group1组。然后通过下面的步骤添加编译和运行菜单。

01. 添加编译菜单

    在图2-28所示界面单击“添加工具”→“程序”按钮，添加一个命令菜单。如图2-29所示输入并选择相关项目，其中“菜单文本”中输入是出现在“工具”菜单中菜单名，这里可以根据需要的喜好取名字；“命令”是菜单要执行的JDK指令，这里指定JDK中javac.exe文件路径；“参数”是指，命令后面的参数，这里需要指定要编译的文件名，$(FileName)是EditPlus获得文件名的系统变量，$(FileName)是带有扩展名的文件名；“起始目录”是命令执行的目录，$(FileDir) 是EditPlus获得文件当前文件目录的系统变量；最后还需要在“动作”中选择“捕获控制台输出”，可以将命令执行结果输出到EditPlus控制台。

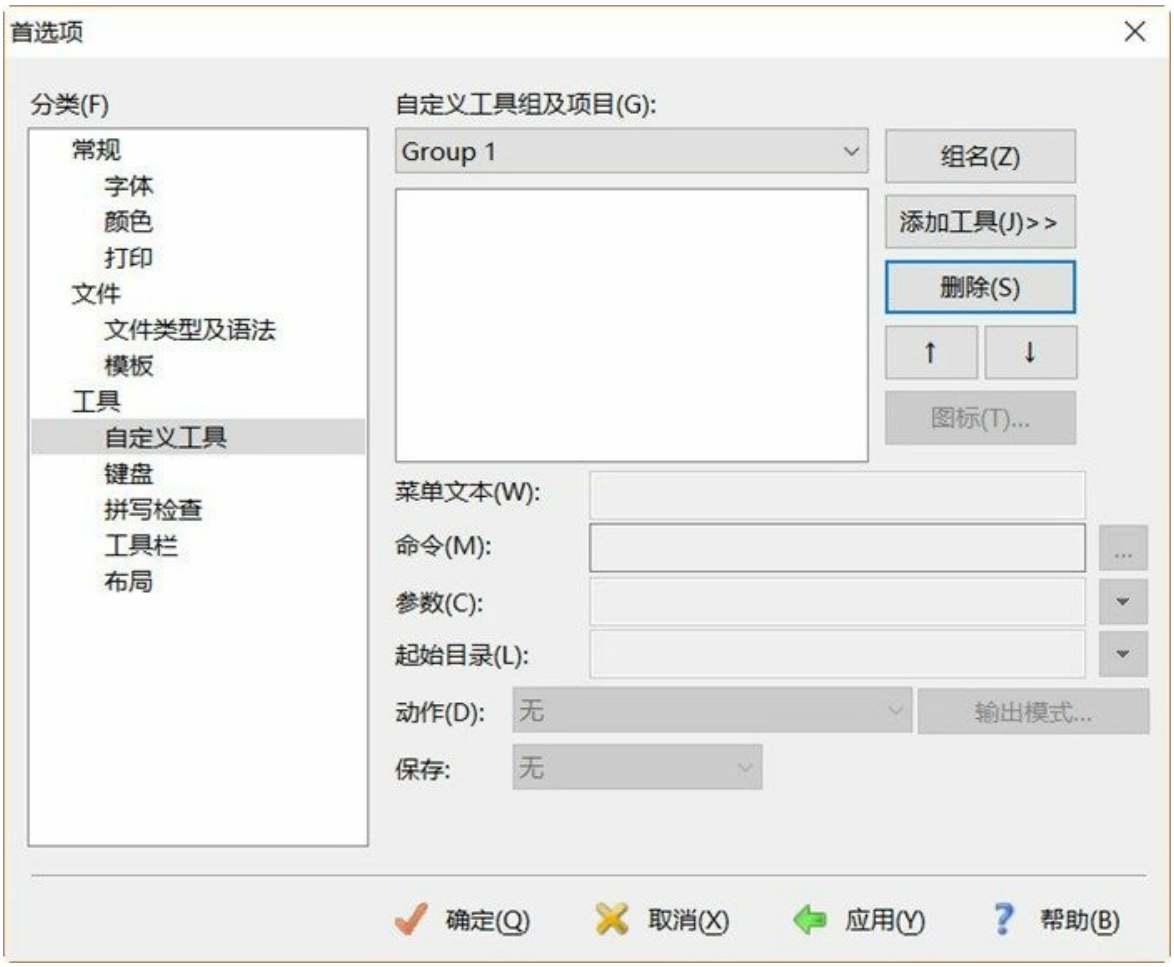


图**2-28　EditPlus**设置参数

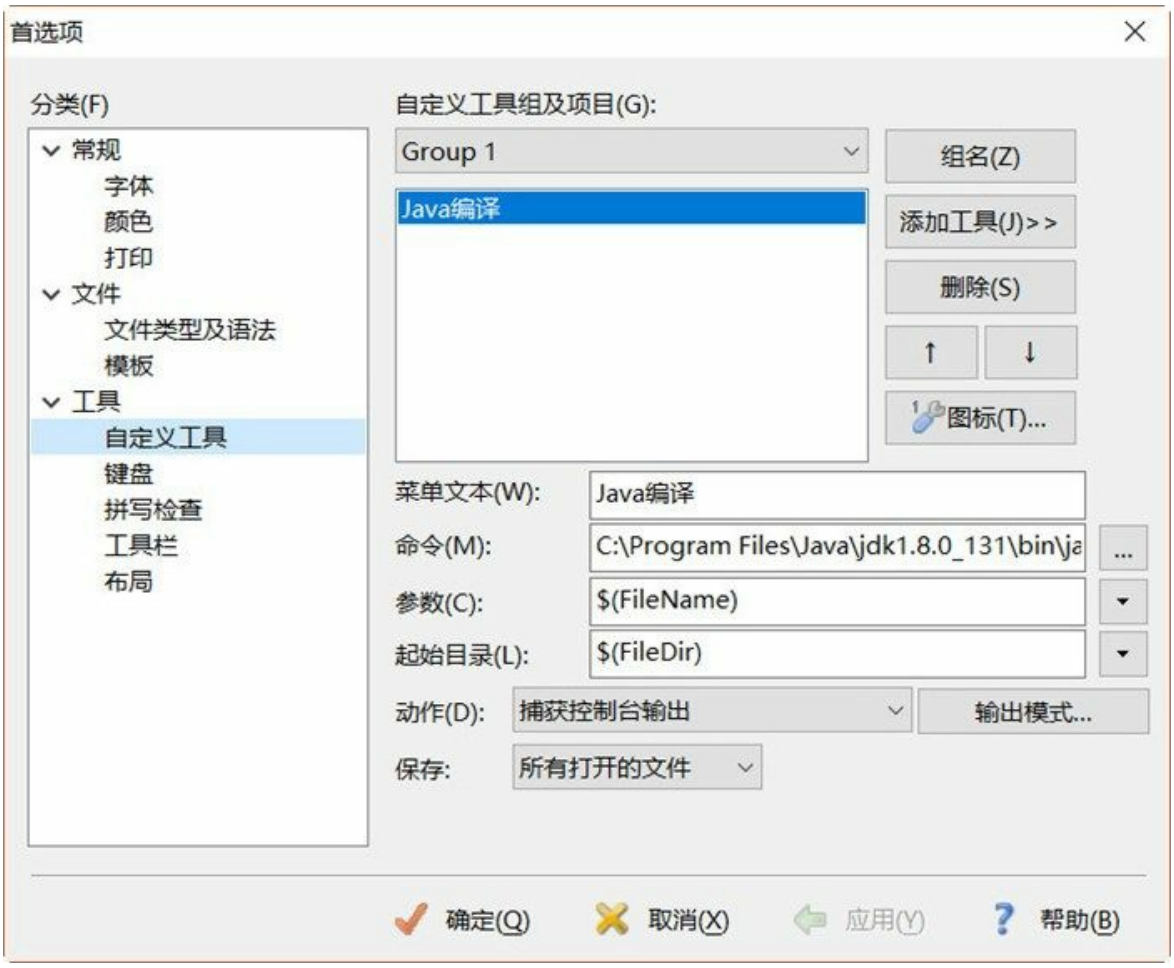


图**2-29**　添加编译菜单

02. 添加运行菜单

参考“添加编译菜单”添加过程，添加一个命令菜单。如图2-30所示，在“命令”中指定JDK中java.exe文件路径；“参数”是$(FileNameNoExt)，表示不带扩展名的文件名。

注意：编译时指定的Java源代码文件，要带有扩展名，指令类似于javac HelloWorld.java。而运行时不需要指定字节码文件的扩展名，指令类似于java HelloWorld。

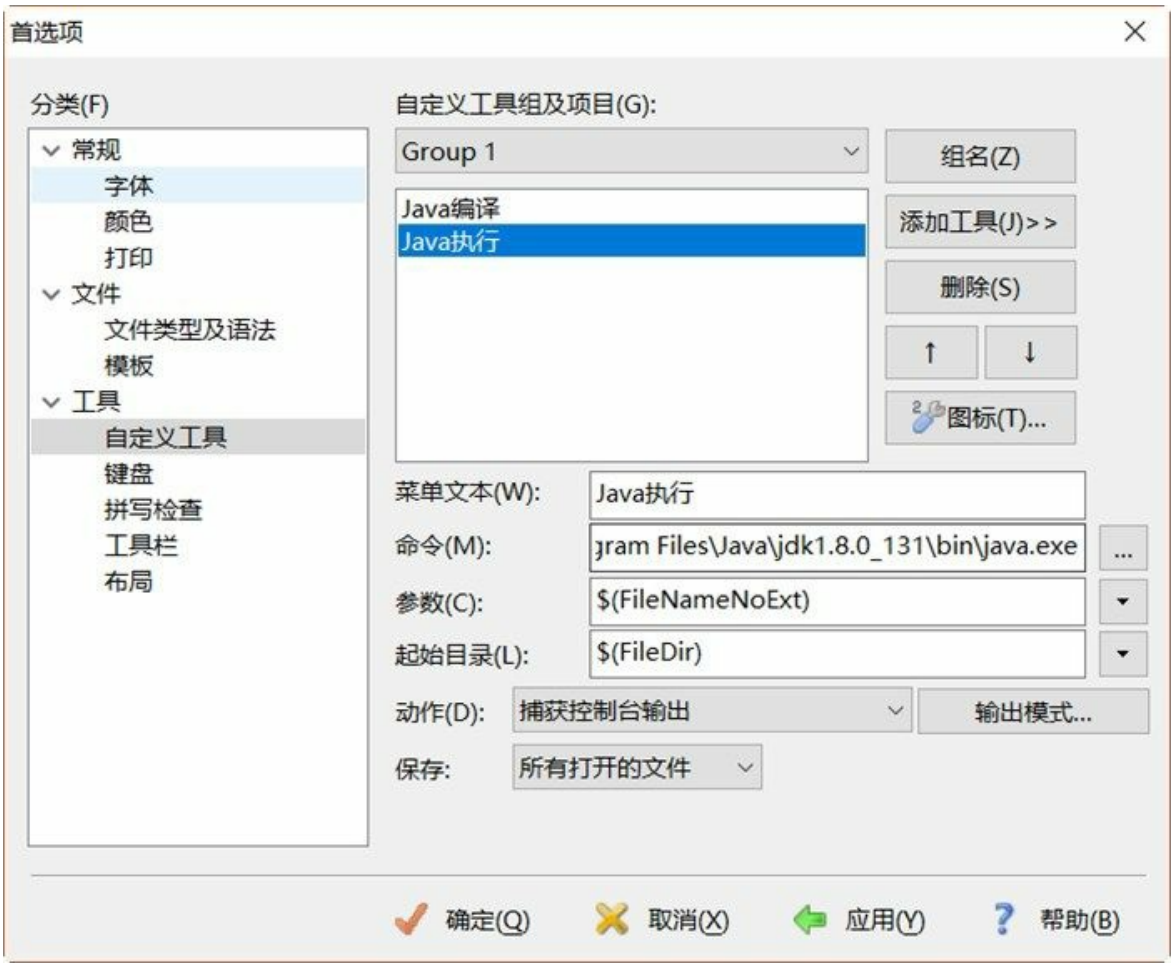


图**2-30**　添加执行菜单

添加成功后会发现EditPlus的工具菜单中多出了两个子菜单，如图2-31所示，Java编译和Java执行。当打开一个源程序HelloWorld.java，可通过单击Java编译菜单（或Ctrl+1快捷键）编写HelloWorld.java，如图2-32所示，编译结果输出到EditPlus控制台；然后通过单击Java执行菜单（或Ctrl+2快捷键）执行编译完成的字节码文件HelloWorld.class，如图2-33所示，运行结果输出到EditPlus控制台。

图**2-31**　添加后的工具菜单

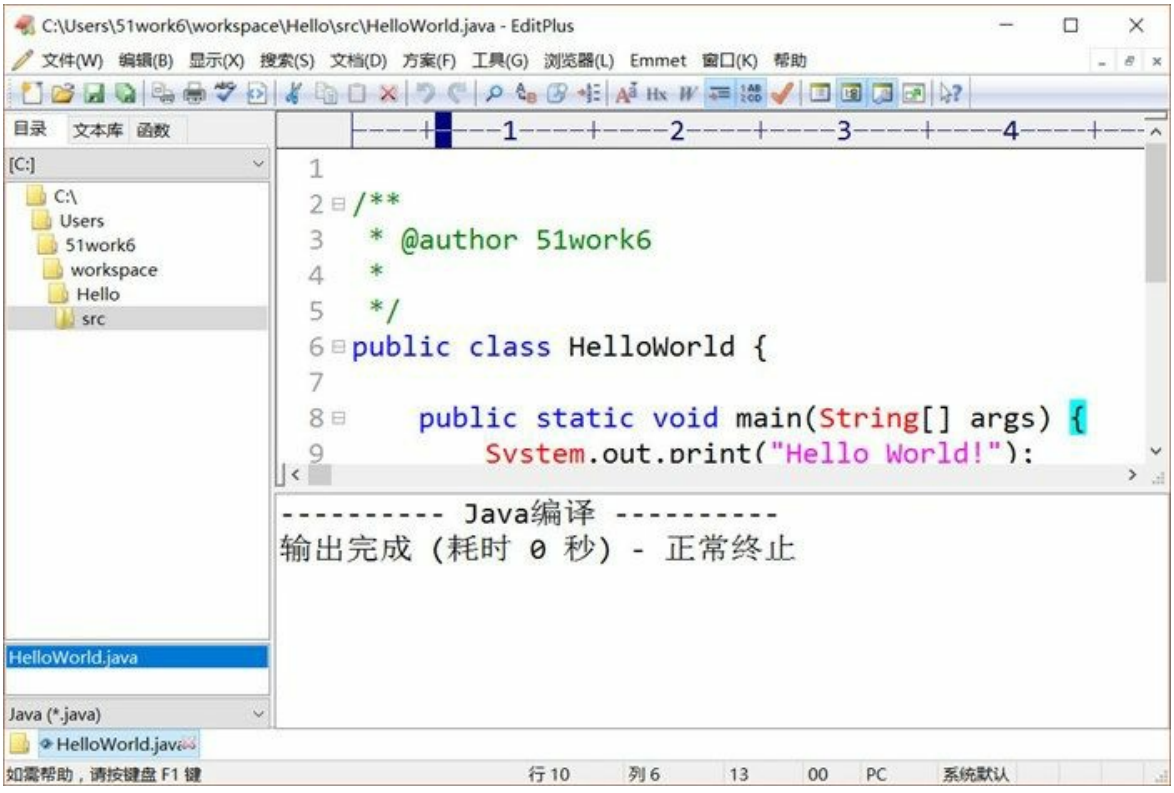


图**2-32**　执行**Java**编译菜单

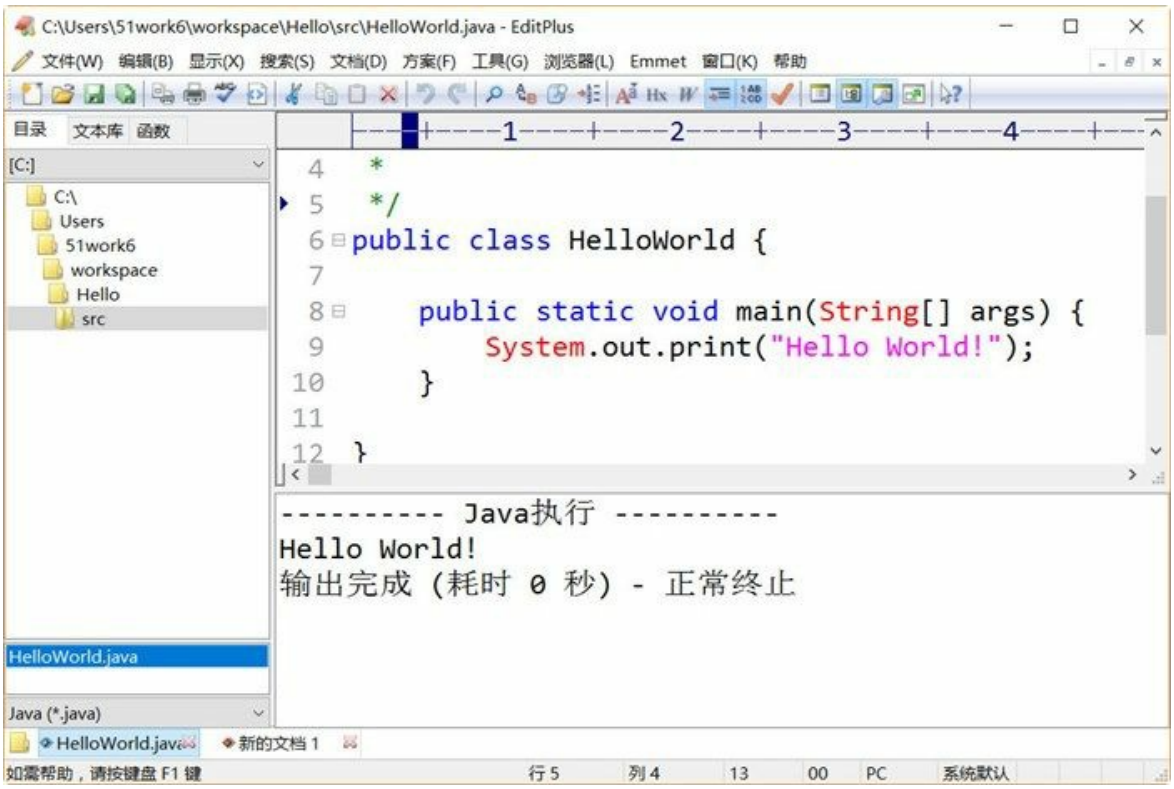


图**2-33**　执行**Java**执行菜单

每一种文本编辑工具的配置方式都有很大差别，这里笔者不能一一穷尽，其他工具的配置过程读者可以参考工具的官方资料。

本章小结

通过对本章的学习，读者可以了解Java开发工具，其中重点是Eclipse工具的下载、安装和使用。此外，还介绍了其他的一些工具：IntelliJ IDEA和NetBeans，以及文本编辑工具EditPlus+JDK的配置过程。

第 3 章　第一个Java程序

本书第一个Java程序是通过控制台输出HelloWorld，以这个示例为切入点，向大家系统介绍Java程序的编写、Java源代码结构以及一些基础知识。

在Java中，程序都是以类的方式组织的，Java源文件都保存为.java文件当中。每个可运行的程序都是一个类文件，或者称之为字节码文件，保存为.class文件。要实现在控制台中输出HelloWorld示例，则需要编写一个Java类。

3.1 使用**Eclipse**实现

HelloWorld示例可通过多种工具实现，这一节首先介绍如何通过Eclipse实现。

3.1.1 创建项目

在Eclipse中通过项目（Project）管理Java类，因此需要先创建一个Java项目，然后在项目中创建一个Java类。

Eclipse创建项目步骤是：打开Eclipse，选择菜单“文件”→“新建”→“Java项目”，打开新建Java项目对话框，如图3-1所示。

下面简要说明图3-1所示各个选项：

- 项目名：是要创建的项目名称。
- 使用缺省位置：选中该选项，创建的项目会保存到工作空间中。
- JRE：开发人员可以在这里指定项目运行所需要的JRE，默认是使用系统Path环境变量所指定的JRE。
- 项目布局：是设置项目中源文件和类文件的存放目录，默认情况下选中“为源文件和类文件创建单独的文件夹”，这个选项选中后，源文件和类文件会在两个不同的文件夹下，即源文件被放置在当前项目的文件夹中，类文件被放置在当前项目的bin文件夹中；如果选中“使用项目文件夹作为源文件和类文件的根目录”，则源文件和类文件都被放置在当前项目根目录下，而且混合在一起。
- 工作集：可以将多个相关的项目集中在一个工作集中管理。

图3-1所示对话框中看起来有很多项目需要设置，其实除了项目名称必须输入外，其他的完全可以采用默认值。选项设置完成后，单击“下一步”按钮，进入如图3-2所示的Java设置对话框，在这里可以对源文件和类文件的保存文件夹进行进一步设置。确认无误后，单击“完成”按钮创建项目。项目创建完成后，回到如图3-3所示的Eclipse主界面。

图**3-1**　新建**Java**项目对话框

图**3-2** **Java**设置对话框

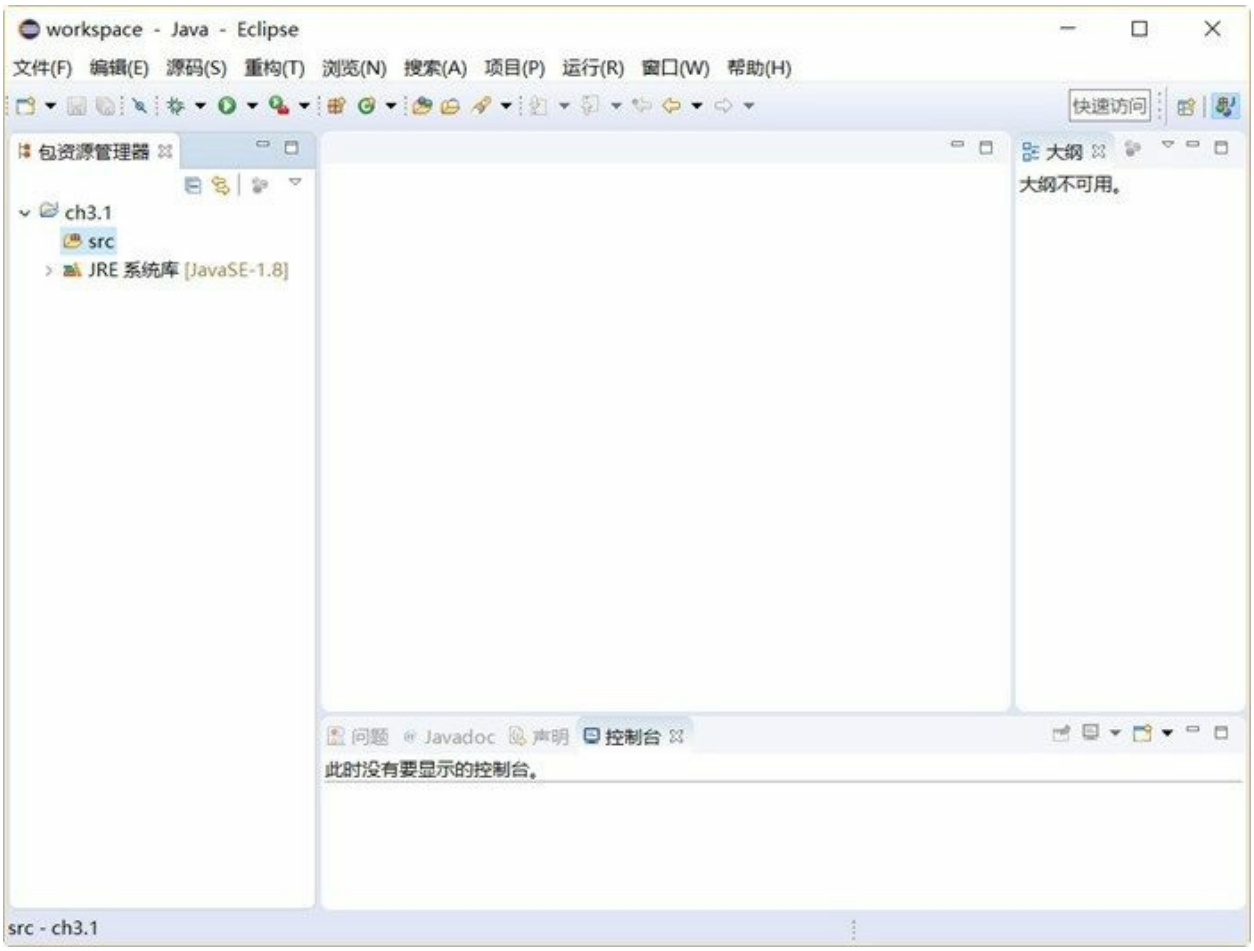


图**3-3**　项目创建完成

3.1.2　创建类

项目创建完成后，需要创建一个类执行控制台输出操作。选择刚刚创建的项目，然后选择菜单“文件”→“新建”→“类”，打开新建类对话框，在对话框中输入如图3-4所示内容。

下面简要说明图3-4所示各个选项：

- 源文件夹：由于创建项目时候指定了源文件夹，这里使用默认值即可。
- 包：是类所在的包，包名一般是公司域名的倒置，可以没有。
- 名称：是类的名称。
- 修饰符：是类前面的修饰符，这些修饰符含义，目前先不解释，选择公有就可以了。
- 超类：即父类，这里可以指定该类的父类。
- 接口：指定该类实现哪些接口。
- 创建方法存根：就是在代码创建这些方法，本例中需要选中第一个方法（main方法），这个main方法是程序的入口。

- 添加注释：这里可以设置代码是否生成注释，也可以修改注释模板。

图**3-4**　创建类对话框

在图3-4所示对话框中输入完成，单击“完成”按钮就创建了一个Java类，如图3-5所示，在包资源管理器中可以看到刚才创建的源文件。

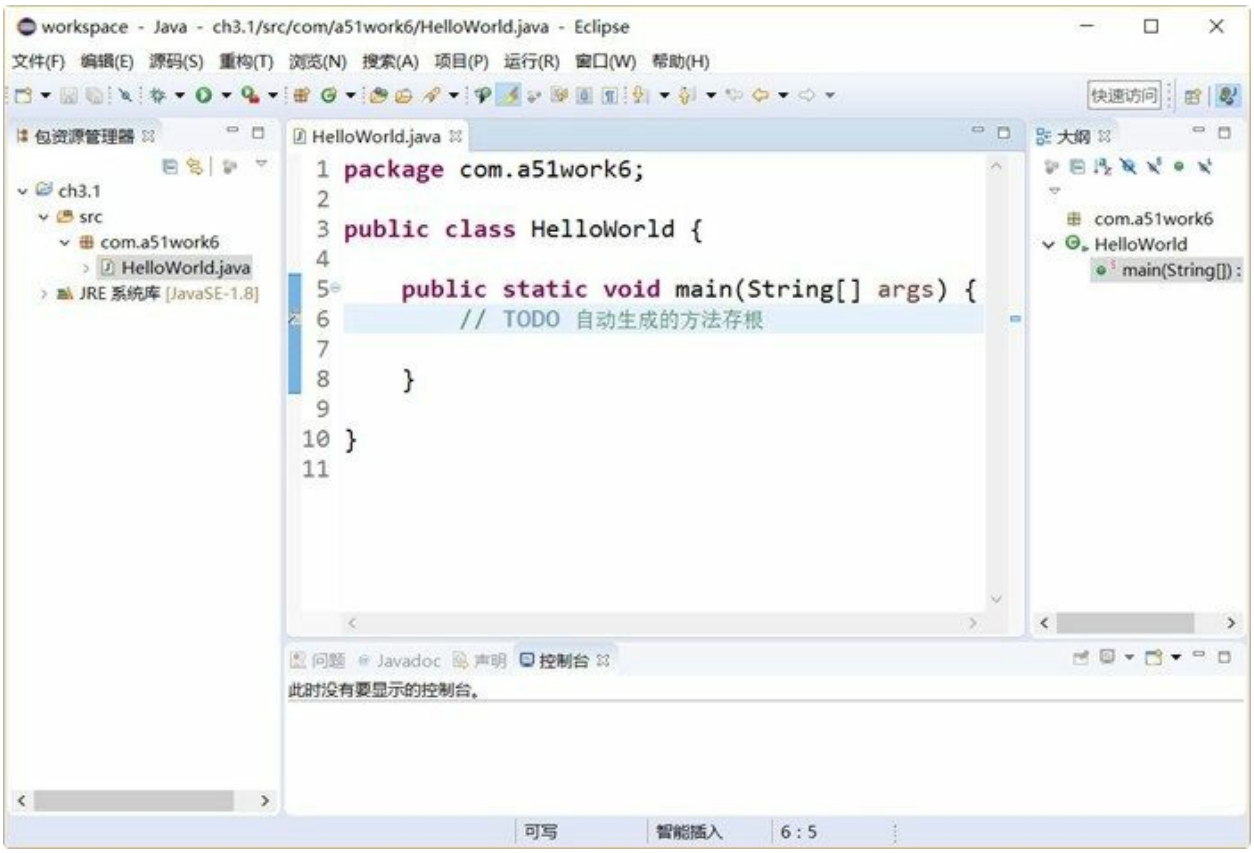


图**3-5**　创建类完成

3.1.3　运行程序

修改刚刚生成的HelloWorld.java源文件，在main方法中添加输出语句，修该完成后代码如下：

```
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {    ①
        System.out.print("Hello World."); ②
    }

}
```

代码第①行中的public static void main(String[] args)方法是一个应用程序的入口，也表明了HelloWorld是一个Java应用程序（Java Application），可以独立运行。代码第②行的System.out.print("Hello World.")语句是输出Hello World.字符串到控制台。

> 提示　在Java SE平台有两种可以独立运行的程序：Java Application（Java应用程序）和Java Applet（Java小应用程序）两种。Java应用程序具有public static void main(String[] args)，上述HelloWorld就是这种类型。Java小应用程序是主要是嵌入到网页中运行的，Java小应用程序是一种淘汰的技术，不再介绍Java小应用程序。

程序编写完成可以运行了。如果是第一次运行，则需要选择运行方法，具体步骤是：选中文件，选择菜单“运行”→“运行方法”→“Java应用程序”，这样就会运行HelloWorld程序了。如果已经运行过程一

次，就不需要这么麻烦了，直接单击工具栏中的“运行”  按钮，或选择菜单“运行”→“运行”，或使用快捷键Ctrl+F11，都可以就运行上次的程序了。运行结果如图3-6所示，Hello World.字符串到下面的控制台。

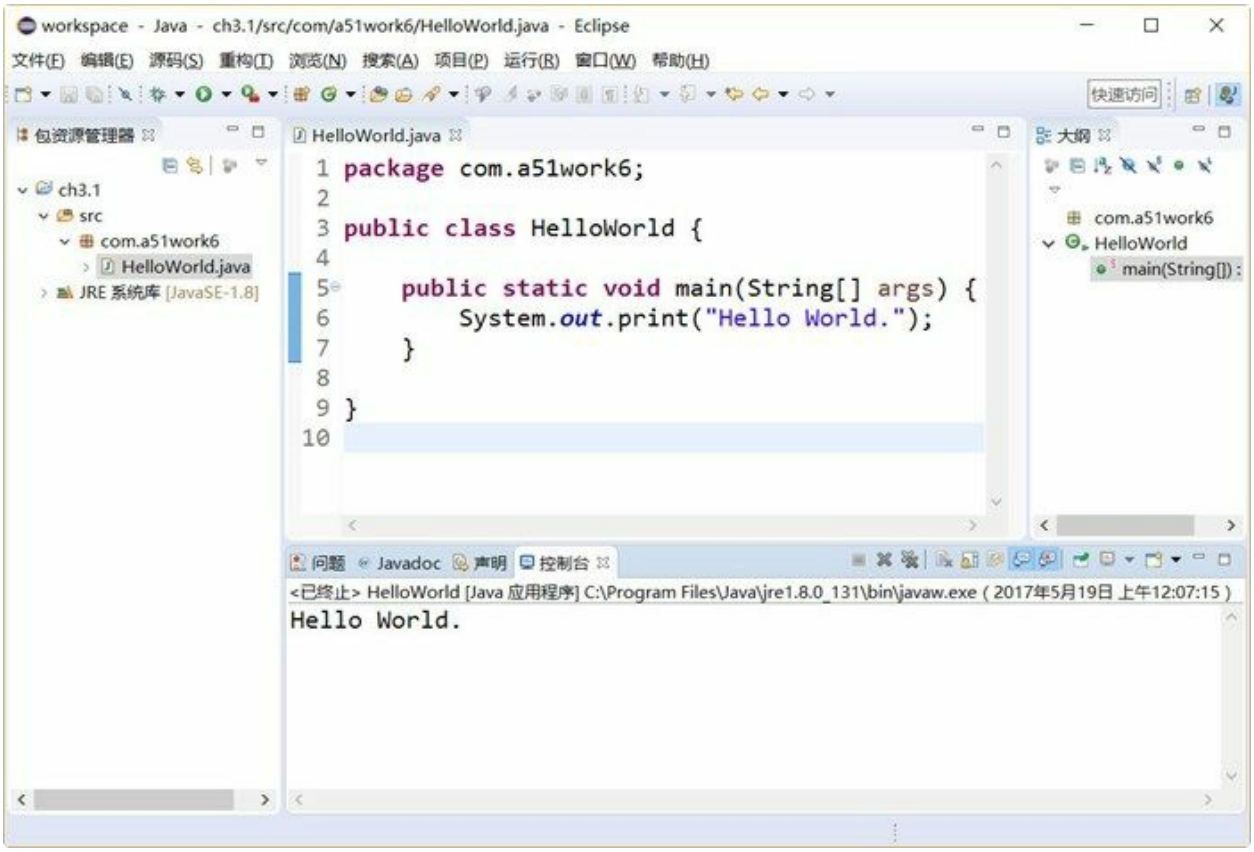


图**3-6**　运行结果

3.2 文本编辑工具+JDK实现

如果不想使用IDE工具（笔者建议出学者通过这种方式学习Java），那么文本编辑工具+JDK对于初学者而言是一个不错的选择，这种方式可以使初学者了解到Java程序的编译和运行过程，通过自己在编辑器中敲入所有代码，可以帮助熟悉常用类和方法。

注意　在2.3.3节介绍过EditPlus与JDK集成过程，2.3.3节集成方式有一个弊端是：不能执行带有包的Java应用程序。

3.2.1 编写源代码文件

首先使用任何文本编辑工具创建一个文件，然后将文件保存为HelloWorld.java。接着在HelloWorld.java文件中编写如下代码：

```
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {
        System.out.print("Hello World.");
    }

}
```

在Java中一个源程序文件中可以定义多个类，如下代码定义了三个类HelloWorld、A和B。

```
//HelloWorld.java源文件
package com.a51work6;

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

class A {

}

class B {

}
```

注意　一个源程序文件包含多个类时，需要注意如下问题：

01. 只能有一个类声明为公有（public）的。

02. 文件命名必须与公有类名完全一致，包括字母大小写。

03. public static void main(String[] args)只能定义在公有类中。

3.2.2 编译程序

编译程序需要在命令行中使用JDK的javac指令编写，参考2.1.2节打开命令行，如图3-7所示，通过cd命令进入到源文件所在的目录，然后执行javac指令。如果没有错误提示，说明编译成功，编译成功则在

当前目录下面生成类文件，如图3-8所示生成了三个类文件，这是因为HelloWorld.java源文件中定义了三个类。

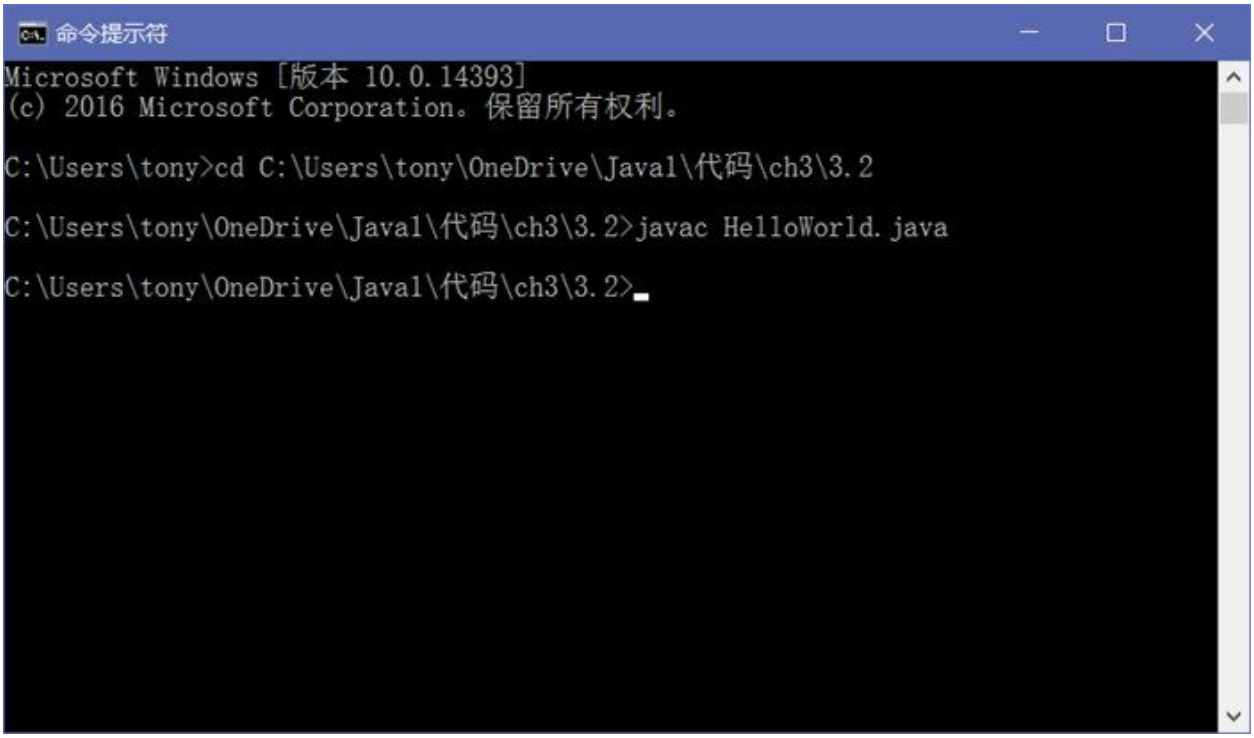


图**3-7**　编译源文件

图**3-8**　编译成功

上述编译过程虽然成功了，但是运行时会有以下问题，这是由于HelloWorld.java源文件中定义了包com.a51work6，编译应该使用-d参数，编译指令如图3-9所示。

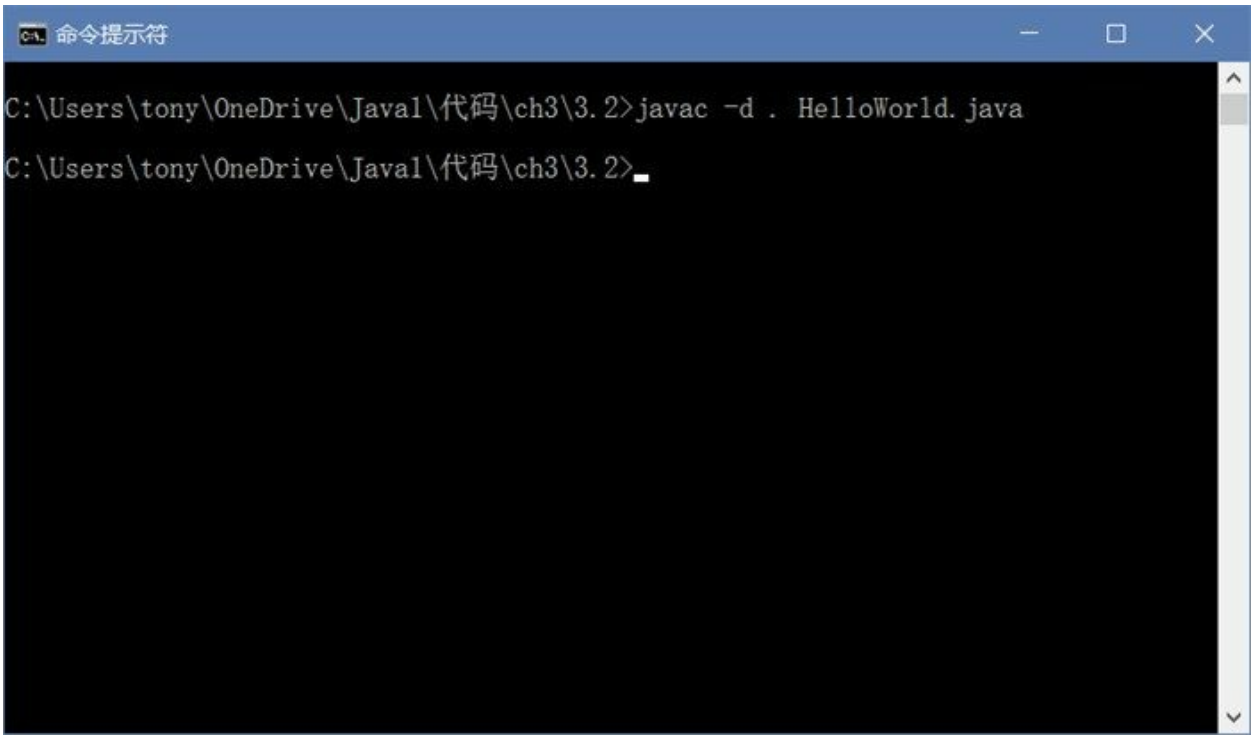


图**3-9** 编译有包的源文件

编译指令javac中的-d参数是指定类文件生成位置，-d后面跟的是一个目录的路径，本例中使用“.”点表示当前目录，编译成功之后的目录结果如下：

```
当前目录
 HelloWorld.java
└com
    └a51work6
            A.class
            B.class
            HelloWorld.class
```

其中的com是目录，它在当前目录的子目录，a51work6也是目录，它是com的子目录，可以包com.a51work6会生成com\a51work6的目录结构。

3.2.3 运行程序

编译成功之后就可以运行了。执行类文件需要在命令行中使用JDK的java指令，参考2.1.2节打开命令行，如图3-10所示，通过cd命令进入到源文件所在的目录，然后执行java -classpath .;c:\com.a51work6.HelloWorld指令，执行成功在命令行窗口输出Hello World!字符串。

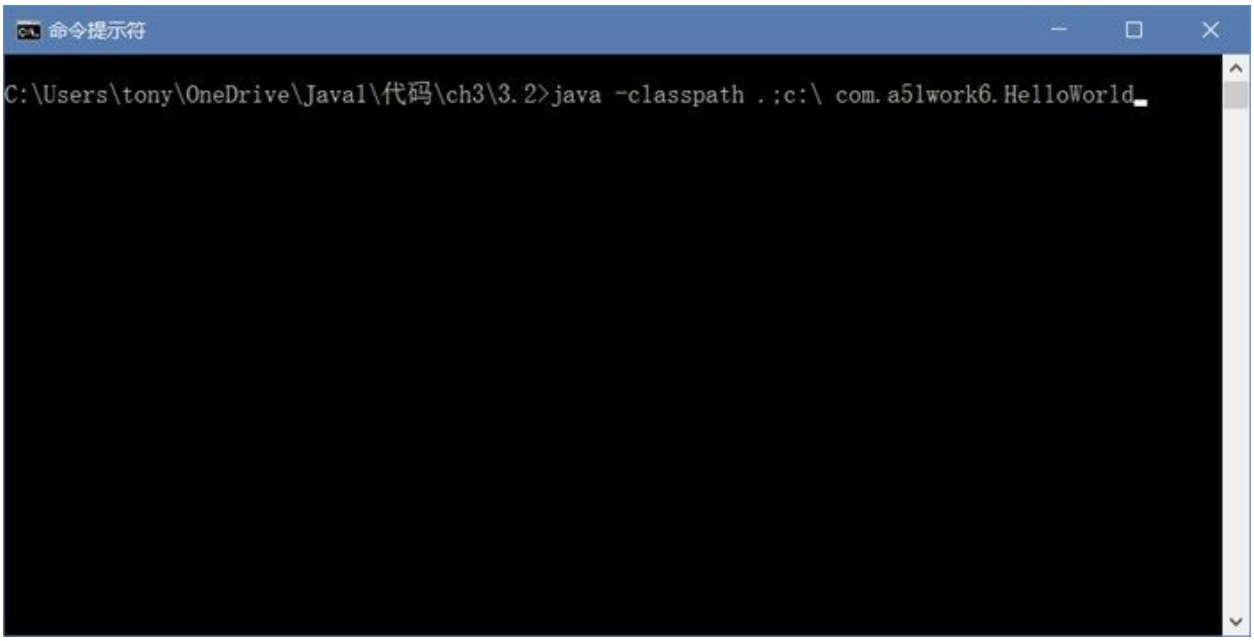


图**3-10**　运行类文件

注意　java和javac指令都可以带有-classpath（缩写-cp），它用来指定类路径，即搜索类的路径，类似于操作系统中的path，路径之间用分号分隔，其中点（.）表示当前路径。就本例而言运行java程序HelloWorld所需要的全部类都在当前路径下，因此只需要设置-classpath .就可以了，或者省略（当前路径不用指定）。

3.3 代码解释

经过前文的介绍，读者应该能够照猫画虎，自己动手做一个Java应用程序了。但还是对其中的一些代码不甚了解，下面来详细解释一下HelloWorld示例中的代码。

```
//包定义
package com.a51work6; ①

//类定义
public class HelloWorld { ②

    //定义静态main方法
    public static void main(String[] args) { ③
        System.out.print("Hello World."); ④
    }

}
```

代码第①行是定义类所在的包，package是关键字，com.a51work6是包名，包是一个命名空间，可以防止命名冲突问题，关于包的概念将在后面章节详细介绍。

代码第②行是定义类，public修饰符是声明类是公有的，class是定义类关键字，HelloWorld是自定义的类名了，后面跟有“{…}”是类体，类体中会有成员变量和方法，也会有一些静态变量和方法。

代码第③行是定义静态main方法，而作为一个Java应用程序，类中必须包含静态main方法，程序执行是从main方法开始的。main方法中除参数名args可以自定义外，其他必须严格遵守如下来两种格式：

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

这两种格式本质上就是一种，String args[]和String[] args都是声明String数组。另外，args参数是程序运行时，通过控制台向应用程序传递字符串参数。

代码第④行System.out.print("Hello World.");语句是通过Java输出流（PrintStream）对象System.out打印Hello World.字符串，System.out是标准输出流对象，它默认输出到控制台。输出流（PrintStream）中常用打印方法：

- print(String s)：打印字符串不换行，有多个重载方法，可以打印任何类型数据。
- println(String x)：打印字符串换行，有多个重载方法，可以打印任何类型数据。
- printf(String format, Object... args)：使用指定输出格式，打印任何长度的数据，但不换行。

修改HelloWorld.java示例代码如下：

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

        //通过print打印第一个控制台参数
        System.out.print(args[0]); ①
        //通过println打印第二个控制台参数
        System.out.println(args[1]); ②
        //通过printf打印第三个控制台参数，%s表示格式化字符串
        System.out.printf("%s", args[2]); ③
        System.out.println();
```

```
        int i = 123;
        //%d表示格式化整数
        System.out.printf("%d\n", i);  ④

        double d = 123.456;
        //%f表示格式化浮点数
        System.out.printf("%f%n", d);  ⑤
        System.out.printf("%5.2f", d);  ⑥

    }
}
```

编译HelloWorld.java源代码后，通过如图3-11所示，其中的java命令行后面的HelloWorld是要运行的类文件，Tony Hello World.是参数，多个参数用空格分隔。

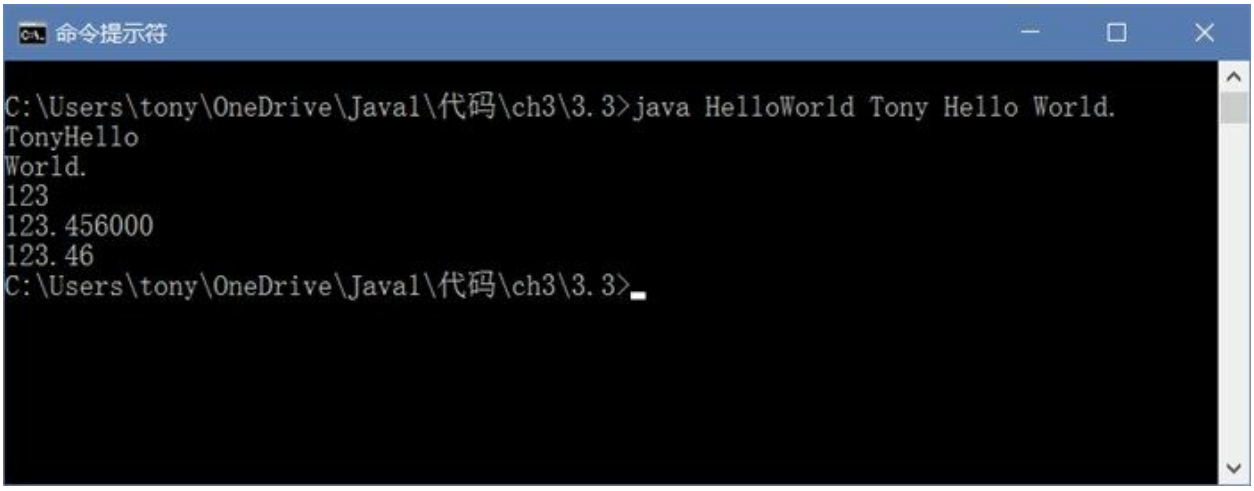


图**3-11**　在命令行中运行程序

上述代码第①行使用print方法打印第一个参数args[0]，注意该方法是打印完成后面不换行，从输出结果中可见第一个参数Tony和第二个参数Hello连在一起了。代码第②行使用println方法打印第二个参数args[1]，从输出结果中可见第二个参数Hello后面是有换行的。

代码第③、④、⑤、⑥行都是使用printf方法打印，注意printf方法后面是没有换行的，想在后面换行可以通过System.out.println()语句实现，或在打印第字符串后面添加换行符号（\n或%n），见代码第④行和第⑤行。代码第⑥行中%5.2f也表示格式化浮点数，5表示总输出的长度，2表示保留的小数位。

本章小结

本章通过一个HelloWorld示例入手，介绍使用Eclipse和使用文本工具+JDK实现该示例具体过程。掌握Eclipse使用非常重要，但是使用文本工具+JDK对于初学者也很有帮助。最后详细解释了HelloWorld示例。

第 4 章　Java语法基础

本章主要为大家介绍Java的一些基本语法，其中包括标识符、关键字、保留字、常量、变量、表达式等内容。

4.1 标识符、关键字和保留字

任何一种计算机语言都离不开标识符和关键字，因此下面将详细介绍Java标识符、关键字和保留字。

4.1.1 标识符

标识符就是变量、常量、方法、枚举、类、接口等由程序员指定的名字。构成标识符的字母均有一定的规范，Java语言中标识符的命名规则如下：

01. 区分大小写：Myname与myname是两个不同的标识符。

02. 首字符，可以是下划线（_）或美元符或字母，但不能是数字。

03. 除首字符外其他字符，可以是下划线（_）、美元符、字母和数字。

04. 关键字不能作为标识符。

例如，身高、identifier、userName、User_Name、$Name、_sys_val等为合法的标识符，注意中文“身高”命名的变量是合法的；而2mail、room#和class为非法的标识符，注意#是非法字符，而class是关键字。

注意　Java语言中字母采用的是双字节Unicode编码1。Unicode叫作统一编码制，它包含了亚洲文字编码，如中文、日文、韩文等字符。

4.1.2 关键字

关键字是类似于标识符的保留字符序列，由语言本身定义好的，不能挪作他用，Java语言中有50个关键字，如表4-1所示。

表 **4-1　Java**关键字

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

关键字很多这里不再一一介绍了，但是读者需要记住一点的是Java中的关键字全部是小写字母。

4.1.3 保留字

Java中有一些字符序列既不能当作标识符使用，也不是关键字，也不能在程序中使用，这些字符序列称为保留字。Java语言中的保留字只有两个goto和const：

01. goto：在其他语言中叫做“无限跳转”语句，在Java语言中不再使用goto语句，因为“无限跳转”语句会破坏程序结构。在Java语言中goto的替换语句可以通过break、continue和return实现“有限跳转”。

02. const：在其他语言中是声明常量关键字，在Java语言中声明常量使用public static final 方式声明。

4.2 Java分隔符

在Java源代码中，有一些字符被用作分隔，称为分隔符。分隔符主要有：分号（;）、左右大括号（{}）和空白。

01. 分号

    分号是Java语言中最常用的分隔符，它表示一条语句的结束。下面代码：

```
int totals = 1 + 2 + 3 + 4;
```

    等价于

```
int totals = 1 + 2
    + 3 + 4;
```

02. 大括号

    在Java语言中，以左右大括号（{}）括起来语句集合称为语句块（block）或复合语句，语句块中可以有0~n条语句。在定义类或方法时，语句块也被用做分隔类体或方法体。语句块也可以嵌套，且嵌套层次没有限制。示例代码如下：

```
public class HelloWorld {

    public static void main(String args[]) {

        int m = 5;
        if (m < 10) {
            System.out.println("<10");
        }

    }
}
```

03. 空白

    在Java源代码中元素之间允许有空白，空白的数量不限。空白包括空格、制表符（Tab键输入）和换行符（Enter键输入），适当的空白可以改善对源代码可读性。下列几段代码是等价。

```
if (m < 10) {
    System.out.println("<10"); }
```

    等价于

```
if (m < 10)
    {
    System.out.println("<10");
}
```

    等价于

```
if (m < 10) {
    System.out.println("<10");
```

```
}
```

4.3 变量

变量和常量是构成表达式的重要部分，变量所代表的内部是可以被修改的。变量包括变量名和变量值，变量的声明格式为：

```
数据类型  变量名  [=初始值];
```

变量名要遵守用标识符命名规范，却在相关的作用域中不能有重复的变量名。变量作用域是变量的使用范围，在此范围内变量可以使用，超过作用域，变量内容则被释放，根据作用域不同分为：成员变量和局部变量，示例代码如下：

```
public class HelloWorld {

    // 声明int型成员变量
    int y; ①

    public static void main(String[] args) {

        // 声明int型局部变量
        int x; ②
        // 声明float型变量并赋值
        float f = 4.5f; ③

        // x = 10;
        System.out.println("x = " + x);// 编译错误，局部变量 x未初始化 ④
        System.out.println("f = " + f);

        if (f < 10) {
            // 声明型局部变量
            int m = 5; ⑤

        }
        System.out.println(m); // 编译错误 ⑥
    }
}
```

上述代码中代码第①行是声明的成员变量y，成员变量是在类体中，而在方法之外，作用域是整个类，如果没有初始赋值，系统会为它分配一个默认值，每一种数据类型都有默认值，int类型默认值是0。

代码第②、③、⑤行都是声明局部变量，局部变量是在方法或if、for和while等代码块中声明的变量，第②和③行声明局部变量作用域是整个方法，第⑤行声明的m变量作用域是当前的if语句。

另外，代码第④行和第⑥行会有编译错误方法，这是因为第④行是因为x使用之前没有被初始化，与成员变量不同，局部变量在使用之前必须显示地初始化。代码第③行是在声明的同时初始化了。代码第⑥行的错误是因为m变量超过了作用域。

4.4 常量

常量事实上是那些内容不能被修改的变量，常量与变量类似也需要初始化，即在声明常量的同时要赋予一个初始值。常量一旦初始化就不可以被修改。它的声明格式为：

```
final 数据类型 变量名 = 初始值;
```

final关键字表示最终的，它可以修改很多元素，修饰变量就变成了常量。示例代码如下：

```
public class HelloWorld {

    // 静态常量，替代保留字const
    public static final double PI = 3.14; ①

    // 声明成员常量
    final int y = 10;    ②

    public static void main(String[] args) {
        // 声明局部常量
        final double x = 3.3; ③
    }
}
```

事实上常量有三种类型：静态常量、成员常量和局部常量。代码第①行的是声明静态常量，使用在final 之前使用public static修饰，用来替代保留字const。public static修饰的常量作用域是全局的，不需要创建对象就可以访问它，在类外部访问形式：HelloWorld. PI，这种常量在编程中使用很多。

代码第②行声明成员常量，作用域类似于成员变量，但不能修改。代码第③行声明局部常量，作用域类似于局部变量，但不能修改。

本章小结

本章主要介绍了Java语言中最基本的语法，首先介绍了标识符、关键字和保留字，读者需要掌握标识符构成，了解Java关键字和保留字。接着介绍了Java中的分隔符，最后介绍了变量和常量，读者需要掌握变量种类和作用域，以及常量的声明。

第 5 章　Java编码规范

俗话说：“没有规矩不成方圆”。编程工作往往都是一个团队协同进行，因而一致的编码规范非常有必要，这样写成的代码便于团队中的其他人员阅读，也便于编写者自己以后阅读。

5.1 命名规范

程序代码中到处都是标识符，因此取一个一致并且符合规范的名字非常重要。

命名方法很多，但是比较有名的且被广泛接受的命名法包括下面两种。

- 匈牙利命名，一般只是命名变量，原则是：变量名 = 类型前缀 + 描述，如bFoo表示布尔类型变量，pFoo表示指针类型变量。匈牙利命名还是有一定争议的，在Java编码规范中基本不被采用。
- 驼峰命名（Camel-Case），又称“骆驼命名法”，是指混合使用大小写字母来命名。驼峰命名又分为小驼峰法和大驼峰法。小驼峰法就是第一个单词是全部小写，后面的单词首字母大写，如myRoomCount；大驼峰法是第一个单词的首字母也大写，如ClassRoom。

除了包和常量外，Java编码规范命名方法采用驼峰法，下面分类说明一下。

- 包名：包名是全小写字母，中间可以由点分隔开。作为命名空间，包名应该具有唯一性，推荐采用公司或组织域名的倒置，如com.apple.quicktime.v2。但Java核心库包名不采用域名的倒置命名，如java.awt.event。
- 类和接口名：采用大驼峰法，如SplitViewController。
- 文件名：采用大驼峰法，如BlockOperation.java。
- 变量：采用小驼峰法，如studentNumber。
- 常量名：全大写，如果是由多个单词构成，可以用下划线隔开，如YEAR和WEEK_OF_MONTH。
- 方法名：采用小驼峰法，如balanceAccount、isButtonPressed等。

命名规范示例如下：

```
package com.a51work6;

public class Date extends java.util.Date {

    private static final int DEFAULT_CAPACITY = 10;

    private int size;

    public static Date valueOf(String s) {

        final int YEAR_LENGTH = 4;
        final int MONTH_LENGTH = 2;

        int firstDash;
        int secondDash;
        ...
    }

    public String toString () {
        int year = super.getYear() + 1900;
        int month = super.getMonth() + 1;
        int day = super.getDate();
        ...
    }
}
```

5.2 注释规范

Java中注释的语法有三种：单行注释（//）、多行注释（/*...*/）和文档注释（/**...*/）。本节介绍如何规范使用这些注释。

5.2.1 文件注释

文件注释就是在每一个文件开头添加注释。文件注释通常包括如下信息：版权信息、文件名、所在模块、作者信息、历史版本信息、文件内容和作用等。

下面看一个文件注释的示例：

```
/*
* 版权所有 2015 北京智捷东方科技有限公司
* 许可信息查看LICENSE.txt文件
* 描述:
*    实现日期基本功能
* 历史版本:
*    2015-7-22: 创建 关东升
*    2015-8-20: 添加socket库
*    2015-8-22: 添加math库
*/
```

上述注释只是提供了版权信息、文件内容和历史版本信息等，文件注释要根据本身的实际情况包括内容。

5.2.2 文档注释

文档注释就是指这种注释内容能够生成API帮助文档，JDK中javadoc命令能够提取这些注释信息并生成HTML文件。文档注释主要对类（或接口）、实例变量、静态变量、实例方法和静态方法等进行注释。

> 提示　文档是要给别人看的帮助文档，一般注释的实例变量、静态变量、实例方法和静态方法都应该是非私有的，那些只给自己看的内容可以不用文档注释。

文档注释示例：

```
package com.a51work6;

/**
 * 自定义的日期类，具有日期基本功能，继承java.util.Date
 * <p>实现日期对象和字符串之间的转换</p>
 * @author 关东升
 */
public class Date extends java.util.Date {

    private static final int DEFAULT_CAPACITY = 10;

    /**
     * 容量
     */
    public int size;

    /**
     * 将字符串转换为Date日期对象
     * @param s 要转换的字符串
     * @return Date日期对象
```

```
     */
    public static Date valueOf(String s) {

        final int YEAR_LENGTH = 4;
        final int MONTH_LENGTH = 2;

        int firstDash;
        int secondDash;

      ...
    }

    /**
     * 将日期转换为yyyy-mm-dd格式的字符串
     * @return yyyy-mm-dd格式的字符串
     */
    public String toString () {
        int year = super.getYear() + 1900;
        int month = super.getMonth() + 1;
        int day = super.getDate();
      ...
    }
}
```

由于文档注释最终会生成HTML文档，所以可以在文档注释中使用HTML标签，上述注释中的 是HTML段落标签。

另外，上述的文档注释中还用到了@author、@return和@param等文档注释标签，这些标签能够方便生成API帮助文档，表5-1所示是常用的文档注释标签。

表 **5-1**　文档注释标签

标签	描述
@author	说明类或接口的作者
@deprecated	说明类、接口或成员已经废弃
@param	说明方法参数
@return	说明返回值
@see	参考另一个主题的链接
@exception	说明方法所抛出的异常类
@throws	同@exception 标签
@version	类或接口的版本

如果你想生成API帮助文档，可以使用javadoc指令，如图5-1所示，在命令行中输入javadoc -d apidoc Data.java指令，-d参数指明要生成文档的目录，apidoc是当前目录下面的apidoc目录，如果不存在javadoc会创建一个apidoc目录；Data.java是当前目录下的Java源文件。

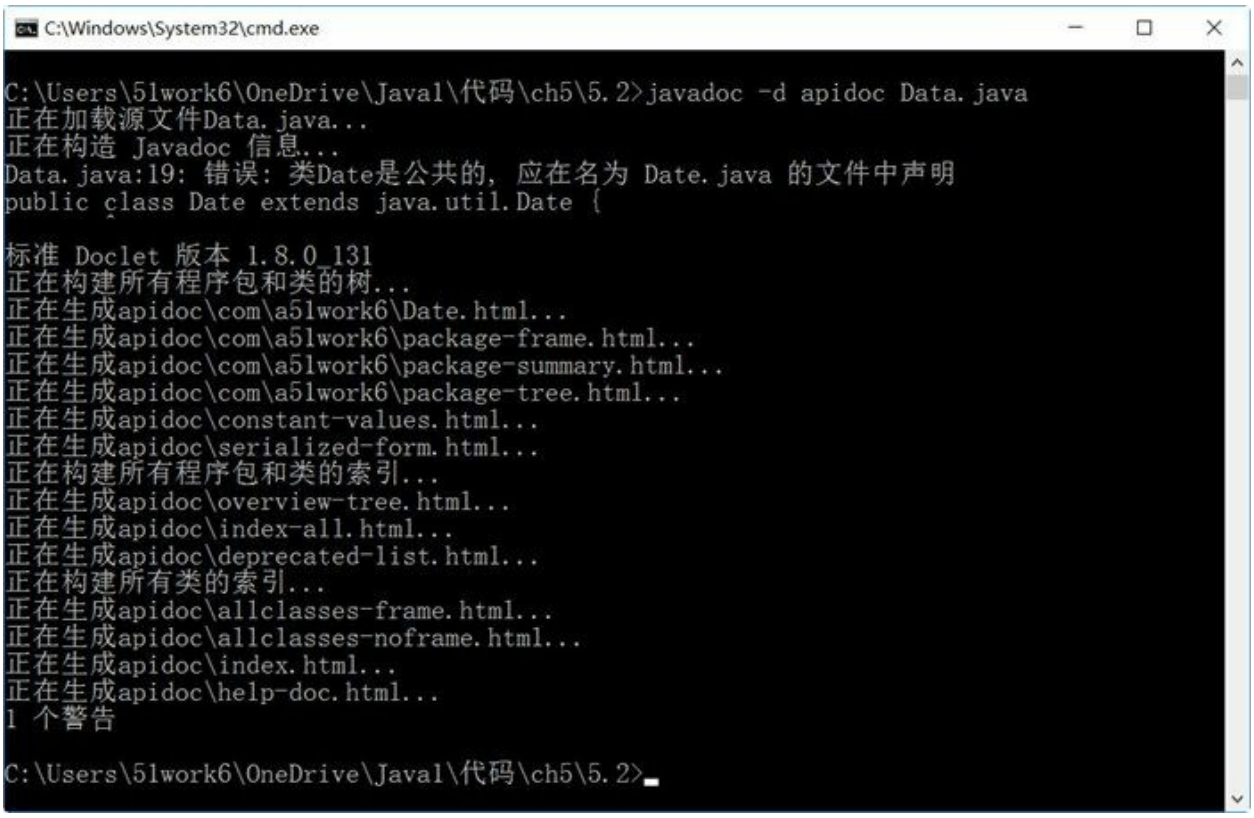


图**5-1**　生成**API**帮助文档

如果生成成功则在当前apidoc目录下生成很多HTML文件，其中的index.html文件是文档的入口，双击此文件，打开如图5-2所示的页面，从页面中可见在Date.java中注释的内容会出现在HTML页面中。


Date
file:///C:/Users/51work6/OneDrive/Java1/代码/ch5/5.2
所有类
Date
程序包 类 树 已过时 索引 帮助
上一个类 下一个类 框架 无框架
概要: 嵌套 | 字段 | 构造器 | 方法 详细资料: 字段 | 构造器 | 方法
com.a51work6
类 Date
java.lang.Object
java.util.Date
com.a51work6.Date
所有已实现的接口:
java.io.Serializable, java.lang.Cloneable, java.lang.Comparable<java.util.Date>
public class Date
extends java.util.Date
自定义的日期类,具有日期基本功能，继承java.util.Date
实现日期对象和字符串之间的转换
另请参阅:
序列化表格
字段概要
字段
限定符和类型
字段和说明
int
size
容量
构造器概要
构造器
构造器和说明
Date()
方法概要
所有方法 静态方法 实例方法 具体方法
限定符和类型
方法和说明
java.lang.String
toString()
将日期转换为yyyy-mm-dd格式的字符串
static Date
valueOf(java.lang.String s)
将字符串转换为Date日期对象

图**5-2** **API**帮助文档

注意　javadoc生成文档中的一些中文翻译与本书不一致，如“构造器”是本书中“构造方法”，文档中的“字段”包括了本书中“实例变量”和“静态变量”。

5.2.3 代码注释

程序代码中处理文档注释还需要在一些关键的地方添加代码注释，文档注释一般是给一些看不到源代码的人看的帮助文档，而代码注释是给阅读源代码的人参考的。代码注释一般是采用单行注释（//）和多行注释（/*...*/）。

示例代码如下：

```
public class Date extends java.util.Date {

    // 默认的容量，是一个常量  ①
    private static final int DEFAULT_CAPACITY = 10;

    /**
     * 容量
     */
    public int size;

    /**
     * 将字符串转换为Date日期对象
     * @param s 要转换的字符串
     * @return Date日期对象
     */
    public static Date valueOf(String s) {

        final int YEAR_LENGTH = 4;
        final int MONTH_LENGTH = 2;

        int firstDash;
        int secondDash;

        Date d = null;
        ...

        /*                                  ②
        * 判断d是否为空，
        * 如果为空抛出异常IllegalArgumentException，否则返回d。
        */
        if (d == null) {
            throw new java.lang.IllegalArgumentException();
        }

        return d;
    }

    /**
     * 将日期转换为yyyy-mm-dd格式的字符串
     * @return yyyy-mm-dd格式的字符串
     */
    public String toString () {
        int year = super.getYear() + 1900; //计算年份    ③
        int month = super.getMonth() + 1; /*计算月份*/  ④
        int day = super.getDate();
        ...
    }
}
```

上述代码第①行采用了单行注释，要求与其后的代码具有一样的缩进层级。如果注释的文字很多，可以采用多行注释，见代码第②行。多行注释也要求与其后的代码具有一样的缩进层级。有时也会在代

码的尾端进行注释，这要求注释内容极短，应该再有足够的空白来分开代码和注释，见代码第③行和第④行。

5.2.4 使用地标注释

Eclipse等IDE工具都为Java源代码提供了一些特殊的注释，就是在代码中加一些标识，便于IDE工具快速定位代码，称为“地标注释”。这种注释虽然不是Java官方所提供的，但是主流语言和主流的IDE工具也都支持“地标注释”。

Eclipse工具支持如下三种地标注释：

- TODO：说明此处有待处理的任务，或代码没有编写完成。
- FIXME：说明此处代码是错误的，需要修正。
- XXX：说明此处代码虽然实现了功能，但是实现的方法有待商榷，希望将来能改进。

示例代码如下：

```
@Override
public Order findById(int orderid) {
    // TODO 自动生成的方法存根
    return null;
}

@Override
public int modify(Order order) {
    // FIXME 方法无返回值
    return 0;
}

@Override
public int remove(Order order) {
    // XXX 可采用数据连接池
    return 0;
}
```

这些注释在Eclipse工具的任务视图查看，如果没有打开任务视图，可以通过菜单“窗口”→“显示视图”→“任务”，打开如图5-3所示的视图，双击其中的任务可以跳转到注释处。

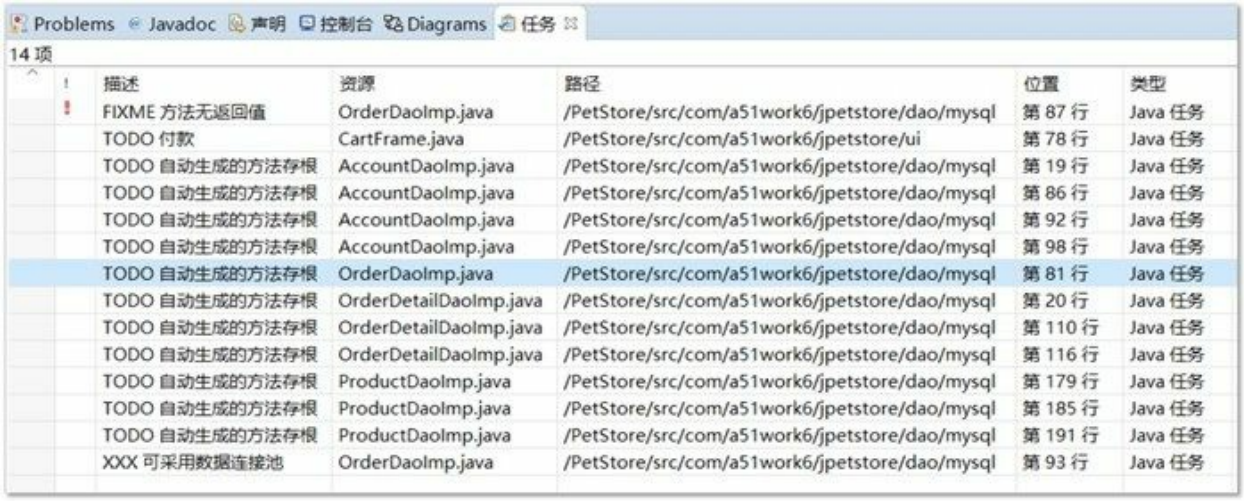


图**5-3　Eclipse**任务视图

5.3 代码排版

代码排版包括空行、空格、断行和缩进等内容。代码排版内容比较多，工作量很大，也非常重要。

5.3.1 空行

空行用以将逻辑相关的代码段分隔开，以提高可读性。空行使用规范：

01. 类声明和接口声明之间保留两个空行。见示例Date.java代码第⑧行和第⑨行。
02. 两个方法之间保留一个空行。见示例Date.java代码第⑦行。
03. 方法的第一条语句之前保留一个空行。见示例Date.java代码第⑤行。
04. 代码注释（尾端注释外）之前保留一个空行。见示例Date.java代码第①、②、③、④行。
05. 一个方法内的两个逻辑段之间。见示例Date.java代码第⑥行。

示例Date.java代码如下：

```
/*
* 版权所有 2015 北京智捷东方科技有限公司
* 许可信息查看LICENSE.txt文件
* 描述:
*    实现日期基本功能
* 历史版本:
*    2015-7-22: 创建 关东升
*    2015-8-20: 添加socket库
*    2015-8-22: 添加math库
*/
package com.a51work6;
    ①
/**
 * 自定义的日期类，具有日期基本功能，继承java.util.Date
 * <p>实现日期对象和字符串之间的转换</p>
 * @author 关东升
 */
public class Date extends java.util.Date {
        ②
    // 默认的容量，是一个常量
    private static final int DEFAULT_CAPACITY = 10;
                ③
    /**
     * 容量
     */
    public int size;
            ④
    /**
     * 将字符串转换为Date日期对象
     * @param s 要转换的字符串
     * @return Date日期对象
     */
    public static Date valueOf(String s) {
            ⑤
        final int YEAR_LENGTH = 4;
        final int MONTH_LENGTH = 2;

        int firstDash;
        int secondDash;
                ⑥
        Date d = null;
```

```
        ...

        /*
        * 判断d是否为空，
        * 如果为空抛出异常IllegalArgumentException，否则返回d。
        */
        if (d == null) {
            throw new java.lang.IllegalArgumentException();
        }

        return d;
    }
                ⑦
    /**
     * 将日期转换为yyyy-mm-dd格式的字符串
     * @return yyyy-mm-dd格式的字符串
     */
    public String toString () {

        int year = super.getYear() + 1900; //计算年份
        int month = super.getMonth() + 1; /*计算月份*/
        int day = super.getDate();
        ...
    }
}

                ⑧
class A {

}

                ⑨
class B {

}
```

5.3.2 空格

代码中的有些位置是需要有空格的，这个工作量也很大。下面是使用空格的规范：

01. 赋值符号“=”前后各有一个空格。示例如下：

```
int YEAR_LENGTH = 4;
int day = super.getDate();
```

02. 所有的二元运算符都应该使用空格与操作数分开。示例如下：

```
a += c + d;
prints("size is " + foo + "\n");
```

03. 一元操作符：负号“-”、自增“++”和自减“--”等，它们与操作数之间没有空格。示例如下：

```
int a = -b;
a++;
--b;
```

04. 小左括号“(”之后，小右括号“)”之前不应有空格。示例如下：

```
a = (a + b) / (c * d)
```

05. 大左括号“{”之前有一个空格。示例如下：

```
while (a == d) {
    n += 1
}
```

06. 方法参数列表小左括号“(”之前没有空格，小右括号“)”之后有一个空格，参数列表中参数逗号“,”之后也有一个空格。示例如下：

```
String format(Object obj, StringBuffer toAppendTo, FieldPosition fieldPosition) {
        ...
}
```

07. 关键字之后紧跟着小左括号“(”，关键字之后应该有一个空格。如下示例中while之后有一个空格。

```
while (a == d) {
    ...
}
```

5.3.3 缩进

4个空格常被作为缩进排版的一个单位。虽然在开发时程序员使用制表符进行缩进，而默认情况下一个制表符等于8个空格，但是不同的IDE工具中一个制表符与空格对应个数会有不同。Eclipse中默认是一个制表符对应4个空格。

缩进可以依据一般规范，如下。

01. 在方法、Lambda、控制语句等包含大括号“{}”的代码块中，代码块的内容相对于首行缩进一个级别（4个空格）。

02. 如果是if语句中条件表达式的断行，那么新的一行应该相对于上一行缩进两个级别（8个空格），再往后的断行要与第一次的断行对齐。

示例如下：

```
public class Date extends java.util.Date {

    …

    public String getString() {

        int year = super.getYear() + 1900; // 计算年份
        int month = super.getMonth() + 1; /* 计算月份 */
        int day = super.getDate();

        if ((longName1 == longName2)
                || (longName3 == longName4) && (longName3 > longName4)     ①
                && (longName2 > longName5)) {  ②

        }

        return null;
    }
}
```

上述代码第①行和第②行是if语句条件表达式的断行，代码第①行和第②行要对齐。

5.3.4 断行

一行代码的长度应尽量不要超过80个字符，如果超过则需断行，可以依据下面的一般规范断开：

01. 在一个逗号后面断开。

02. 在一个操作符前面断开，要选择较高级别的运算符（而非较低级别的运算符）断开。

03. 新的一行应该相对于上一行缩进两个级别（8个空格）。

下面通过一些示例说明：

```
longName1 = longName2 * (longName3 + longName4 - longName5)
        + 4 * longName6     ①
longName1 = longName2 * (longName3 + longName4
        - longName5) + 4 * longName6  ②

private static DateFormat get(int timeStyle, int dateStyle,
                                  int flags, Locale loc) {  ③
    ...
}

if ((longName1 == longName2)
    || (longName3 == longName4) && (longName3 > longName4)
    && (longName2 > longName5)) {    ④

}

 boolName1 = (longName3 == longName4)
     ? (longName3 > longName4)
     : (longName2 > longName5);     ⑤
```

上述代码第①行和第②行是带有小括号运算的表示式，其中代码第①行的断开位置要比第②行的断开位置要好。因为代码第①行断开处位于括号表达式的外边，这是个较高级别的断开。

代码第③行是方法名断开是在参数逗号之后。

代码第④行是if判断语句，由于可能有很多长的条件表达式，断开的位置应在逻辑运算符处。

代码第⑤行是三元运算符的断开。

5.4 其他规范

除了上述规范外，还有很多零散的规范，下面补充一些重要的规范。

01. 在声明变量或常量时推荐一行一个声明。示例如下：

    推荐使用：

```
int longName1 = 0 ;
int longName2 = 0 ;
```

    不推荐使用：

```
int longName1 = 0, longName2 = 0 ;
```

02. 左大括号“{”位于声明语句同行的末尾。右大括号“}”另起一行，与相应的声明语句对齐，除非是一个空语句，右大括号“}”应紧跟在左大括号“{”之后。示例如下：

```
public class Date extends java.util.Date {

    int longName1 = 0;
    int longName2 = 0;

    boolean boolName1 = true;

    public String getString() {

        int year = super.getYear() + 1900; // 计算年份
        int month = super.getMonth() + 1; /* 计算月份 */
        int day = super.getDate();

        return null;
    }

    public void setString() {}
}
```

03. 每行至多包含一条语句。示例如下：

    推荐使用：

```
argv++;
argc--;
```

    不推荐使用：

```
argv++; argc--;
```

04. 虽然Java语言允许if、for等控制语句只有一行代码情况下，省略左右两个大括号，但是编码规范并不推荐这样使用。示例如下：

    推荐使用：

```
if (1 == 3) {
```

```
    x = 2.3;
}
```

不推荐使用：

```
if (1 == 3)
    x = 2.3;
```

关于规范事实上还有很多，不能穷尽，这里不再赘述。

本章小结

通过对本章内容的学习，读者可以了解Java到编码规范，包括命名规范、注释规范、声明规范和代码排版等内容。

第 **6** 章　数据类型

在声明变量或常量时会用到数据类型，在前面已经用到一些数据类型，例如int、double和String等。Java语言的数据类型分为：基本类型和引用类型。

6.1 基本数据类型

基本类型表示简单的数据，基本类型分为4大类，共8种数据类型。

- 整数类型：byte、short、int和long
- 浮点类型：float和double
- 字符类型：char
- 布尔类型：boolean

基本数据类型如图6-1所示，其中整数类型、浮点类型和字符类型都属于数值类型，它们之间可以互相转换。

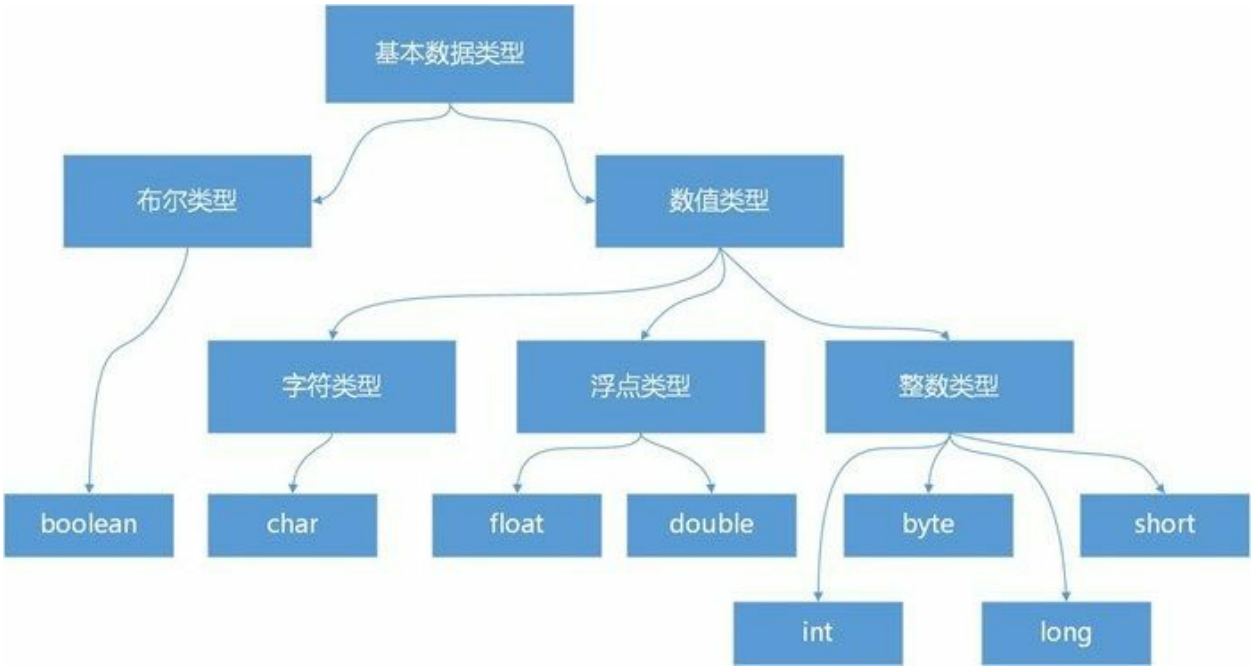


图**6-1** 基本数据类型

6.2 整型类型

从图6-1可见Java中整数类型包括：byte、short、int和long，它们之间的区别仅仅是宽度和范围的不同。Java中整数都是有符号，与C不同没有无符号的整数类型。

Java的数据类型是跨平台的（与平台无关），无论你计算机是32位的还是64位的，byte类型整数都是一个字节（8位）。这些整数类型的宽度和范围如表6-1所示。

表 **6-1** 整数类型

整数类型	宽度	取值范围
byte	1 个字节（8 位）	-128~127
short	2 个字节（16 位）	-2^{15}~$2^{15}-1$
int	4 个字节（32 位）	-2^{31}~$2^{31}-1$
long	8 个字节（64 位）	-2^{63}~$2^{63}-1$

Java语言的整型类型默认是int类型，例如16表示为int类型常量，而不是short或byte，更不是long，long类型需要在数值后面加l（小写英文字母）或L（大写英文字母），示例代码如下：

```
public class HelloWorld {

    public static void main(String[] args) {
        // 声明整数变量
        // 输出一个默认整数常量
        System.out.println("默认整数常量    =  " + 16);        ①
        byte a = 16;                                          ②
        short b = 16;                                         ③
        int c = 16;                                           ④
        long d = 16L;                                         ⑤
        long e = 16l;                                         ⑥

        System.out.println("byte整数        =  " + a);
        System.out.println("short整数       =  " + b);
        System.out.println("int整数         =  " + c);
        System.out.println("long整数        =  " + d);
        System.out.println("long整数        =  " + e);

    }
}
```

上述代码多次用到了16整数，但它们是有所区别的。其中代码①行的16是默认整数类型，即int类型常量。代码②行16是byte整数类型。代码③行的16是short类型。代码第④行的16是int类型。代码第⑤行的16后加了L，这是说明long类型整数。代码第⑥行的16后加了l（小写英文字母l），这是也long类型整数。

提示　在程序代码中，尽量不用小写英文字母l，因为它容易与数字1混淆，特别是在Java中表示

long类型整数时候很少使用小写英文字母l，而是使用大写的英文字母L。例如：16L要比16l可读性更好。

6.3 浮点类型

浮点类型主要用来储存小数数值，也可以用来储存范围较大的整数。它分为浮点数（float）和双精度浮点数（double）两种，双精度浮点数所使用的内存空间比浮点数多，可表示的数值范围与精确度也比较大。浮点类型说明如表6-2所示。

表 **6-2** 浮点类型

浮点类型	宽度
float	4 个字节（32 位）
double	8 个字节（64 位）

Java语言的浮点类型默认是double类型，例如0.0表示double类型常量，而不是float类型。如果想要表示float类型，则需要在数值后面加f或F，示例代码如下：

```
public class HelloWorld {

    public static void main(String[] args) {
        // 声明浮点数
        // 输出一个默认浮点常量
        System.out.println("默认浮点常量    =  " + 360.66);          ①
        float myMoney = 360.66f;                                     ②
        double yourMoney = 360.66;                                   ③
        final double PI = 3.14159d;                                  ④

        System.out.println("float整数 =  " + myMoney);
        System.out.println("double整数    =  " + yourMoney);
        System.out.println("PI        =  " + PI);

    }
}
```

上述代码①行的360.66是默认浮点类型double。代码②行360.66f是float浮点类型，float浮点类型常量表示时，数值后面需要加f或F。代码第③行的360.66表示是double浮点类型。事实上double浮点数值后面也可以加字母d或D，以表示是double浮点数，代码第④行是声明一个double类型常量，数值后面加了d字母。

6.4 数字表示方式

整数类型和浮点类型都表示数字类型，那么在给这些类型的变量或常量赋值时，应该如何表示这些数字的值呢？下面介绍一下数字和指数等的表示方式。

6.4.1 进制数字表示

如果为一个整数变量赋值，使用二进制数、八进制数和十六进制数表示，它们的表示方式分别如下：

- 二进制数：以 0b 或0B为前缀，注意0是阿拉伯数字，不要误认为是英文字母o。
- 八进制数：以0为前缀，注意0是阿拉伯数字。
- 十六进制数：以 0x 或0X为前缀，注意0是阿拉伯数字。

例如下面几条语句都是表示int整数28。

```
int decimalInt = 28;
int binaryInt1 = 0b11100;
int binaryInt2 = 0B11100;
int octalInt = 034;
int hexadecimalInt1 = 0x1C;
int hexadecimalInt2 = 0X1C;
```

6.4.2 指数表示

进行数学计算时往往会用到指数表示的数值。如果采用十进制表示指数，需要使用大写或小写的e表示幂，e2表示10^2。

采用十进制指数表示的浮点数示例如下：

```
double myMoney = 3.36e2;
double interestRate = 1.56e-2;
```

其中3.36e2表示的是3.36×10^2，1.56e-2表示的是1.56×10^{-2}。

6.5 字符类型

字符类型表示单个字符，Java中char声明字符类型，Java中的字符常量必须用单引号括起来的单个字符，如下所示：

char c = 'A';

Java字符采用双字节Unicode编码，占两个字节（16位），因而可用十六进制（无符号的）编码形式表示，它们的表现形式是\un，其中n为16位十六进制数，所以'A'字符也可以用Unicode编码'\u0041'表示，如果对字符编码感兴趣可以到维基百科（https://zh.wikipedia.org/wiki/Unicode字符列表）查询。

示例代码如下：

```
public class HelloWorld {

    public static void main(String[] args) {

        char c1 = 'A';
        char c2 = '\u0041';
        char c3 = '花';

        System.out.println(c1);
        System.out.println(c2);
        System.out.println(c3);
    }
}
```

上述代码变量c1和c2都是保存的'A'，所以输出结果如下：

```
A
A
花
```

提示　字符类型也属于是数值类型，可以与int等数值类型进行数学计算或进行转换。这是因为字符类型在计算机中保存的是Unicode编码，双字节Unicode的存储范围在\u0000~\uFFFF，所以char类型取值范围0~2^{16}-1。

在Java中，为了表示一些特殊字符，前面要加上反斜杠（\），这称为字符转义。常见的转义符的含义参见表6-3。

表 **6-3**　转义符

字符表示	Unicode 编码	说　　明
\t	\u0009	水平制表符 tab
\n	\u000a	换行
\r	\u000d	回车
\"	\u0022	双引号
\'	\u0027	单引号
\\	\u005c	反斜线

示例如下：

```
//在Hello和World插入制表符
String specialCharTab1 = "Hello\tWorld.";
//在Hello和World插入制表符，制表符采用Unicode编码\u0009表示
String specialCharTab2 = "Hello\u0009World.";
//在Hello和World插入换行符
String specialCharNewLine = "Hello\nWorld.";
//在Hello和World插入回车符
String specialCharReturn = "Hello\rWorld.";
//在Hello和World插入双引号
String specialCharQuotationMark = "Hello\"World\".";
//在Hello和World插入单引号
String specialCharApostrophe = "Hello\'World\'.";
//在Hello和World插入反斜杠
String specialCharReverseSolidus = "Hello\\World.";

System.out.println("水平制表符tab1: " + specialCharTab1);
System.out.println("水平制表符tab2: " + specialCharTab2);
System.out.println("换行: " + specialCharNewLine);
System.out.println("回车: " + specialCharReturn);
System.out.println("双引号: " + specialCharQuotationMark);
System.out.println("单引号: " + specialCharApostrophe);
System.out.println("反斜杠: " + specialCharReverseSolidus);
```

输出结果如下：

```
水平制表符tab1: Hello    World.
水平制表符tab2: Hello    World.
换行: Hello
World.
回车: Hello
World.
双引号: Hello"World".
单引号: Hello'World'.
反斜杠: Hello\World.
```

6.6 布尔类型

在Java语言中声明布尔类型的关键字是boolean，它只有两个值：true和false。

提示　在C语言中布尔类型是数值类型，它有两个取值：1和0。而在Java中的布尔类型取值不能用1和0替代，也不属于数值类型，不能与int等数值类型之间进行数学计算或类型转化。

示例代码如下：

```
boolean isMan = true;
boolean isWoman = false;
```

如果试图给它们赋值true和false之外的常量，如下所示。

```
boolean isMan = 1;
boolean isWoman = 'A';
```

则发生类型不匹配编译错误。

6.7 数值类型相互转换

学习了前面的数据类型后，大家会思考一个问题，数据类型之间是否可以转换呢？数据类型的转换情况比较复杂。基本数据类型中数值类型之间可以互相转换，布尔类型不能与它们之间进行转换。但有些不兼容类型之间，如String（字符串）转换为int整数等，可以借助于一些类的方法实现。本节只讨论数值类型的互相转换。

从图6-1可见数值类型包括了byte、short、char、int、long、float和double，这些数值类型之间的转换有两个方向：自动类型转换和强制类型转换。

6.7.1 自动类型转换

自动类型转换就是需要类型之间转换是自动的，不需要采取其他手段，总的原则是小范围数据类型可以自动转换为大范围数据类型，列类型转换顺序如图6-2所示，从左到右是自动。

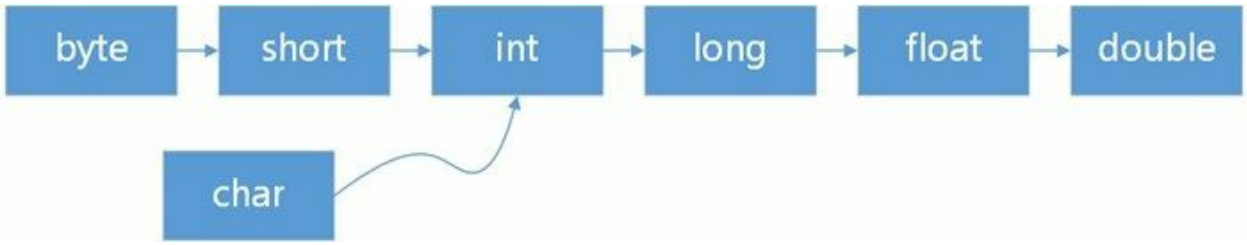


图**6-2** 数据类型转换顺序

> 注意 如图6-2所示，char类型比较特殊，char自动转换为int、long、float和double，但byte和short不能自动转换为char，而且char也不能自动转换为byte或short。

自动类型转换不仅发生在赋值过程中，在进行数学计算时也会发生自动类型转换，在运算中往往是先将数据类型转换为同一类型，然后再进行计算。计算规则如表6-4所示。

表 **6-4** 计算过程中自动类型转换规则

操作数 1 类型	操作数 2 类型	转换后的类型
byte、short、char	int	int
byte、short、char、int	long	long
byte、short、char、int、long	float	float
byte、short、char、int、long、float	double	double

示例如下：

```
// 声明整数变量
byte byteNum = 16;
short shortNum = 16;
int intNum = 16;
long longNum = 16L;

// byte类型转换为int类型
intNum = byteNum;
// 声明char变量
```

```
char charNum = '花';
// char类型转换为int类型
intNum = charNum;

// 声明浮点变量
// long类型转换为float类型
float floatNum = longNum;
// float类型转换为double类型
double doubleNum = floatNum;

//表达式计算后类型是double
double result = floatNum * intNum + doubleNum / shortNum;    ①
```

上述代码第①行中表达式loatNum * intNum + doubleNum / shortNum进行数学计算，该表达式是由4个完全不同的数据类型组成，范围最大的是double，所以在计算过程中它们先转换成double，所以最后的结果是double。

6.7.2 强制类型转换

在数值类型转换过程中，除了需要自动类型转换外，有时还需要强制类型转换，强制类型转换是在变量或常量之前加上“(目标类型)”实现，示例代码如下：

```
//int型变量
int i = 10;
//把int变量i强制转换为byte
byte b = (byte) i;
```

上述代码(byte) i表达式实现强制类型转换。强制类型转换主要用于大宽度类型转换为小宽度类型情况，如把int转换为byte。示例代码如下：

```
//int型变量
int i = 10;
//把int变量i强制转换为byte
byte b = (byte) i;
int i2 = (int)i;        ①
int i3 = (int)b;        ②
```

上述代码第①行是将int类型的i变量“强制类型转换”为int类型，这显然是没有必要，但是语法也是允许的。代码第②行是将byte类型的b变量强制转换为int类型，从图6-2可见这个转换是自动，不需要强制转换，本例中这个转换没有实际意义，但有时为了提高精度需要种转换。示例代码如下：

```
//int型变量
int i = 10;
float c1 = i / 3;                   ①
System.out.println(c1);             ②
//把int变量i强制转换为float
float c2 = (float)i / 3;            ③
System.out.println(c2);             ④
```

输出结果：

```
3.0
3.3333333
```

上述代码比较输出结果发现c1和c2变量小数部分差别比较大的，这种差别在一些金融系统中是不允许的。在代码第①行i除以3从结果是小数，但由于两个操作数都是整数int类型，小数部分被截掉了，结果是3，然后再赋值给float 类型的c1变量，最后c1保存的是3.0。为了防止两个整数进行除法等运算导致小数位被截掉问题，可以将其中一个操作数强制类型转换为float，见代码第③行，这样计算过程中操作数是float类型，结果也是float不会截掉小数部分。

再看一个强制类型转换与精度丢失的示例。

```
long yourNumber = 6666666666L;
System.out.println(yourNumber);
int myNuber = (int)yourNumber;
System.out.println(myNuber);
```

输出结果：

```
6666666666
-1923267926
```

上述代码输出结果可见，经过强制类型转换后，原本的6666666666L变成了负数。当大宽度数值转换为小宽度数值时，大宽度数值的高位被截掉，这样就会导致数据精度丢失。除非大宽度数值的高位没有数据，就是这个数比较小的情况，例如将6666666666L换为6L就不会丢失精度。

6.8 引用数据类型

在Java中除了8种基本数据类型外，其他数据类型全部都是引用（reference）数据类型，引用数据类型用了表示复杂数据类型，如图6-3所示，包含：类、接口和数组声明的数据类型。

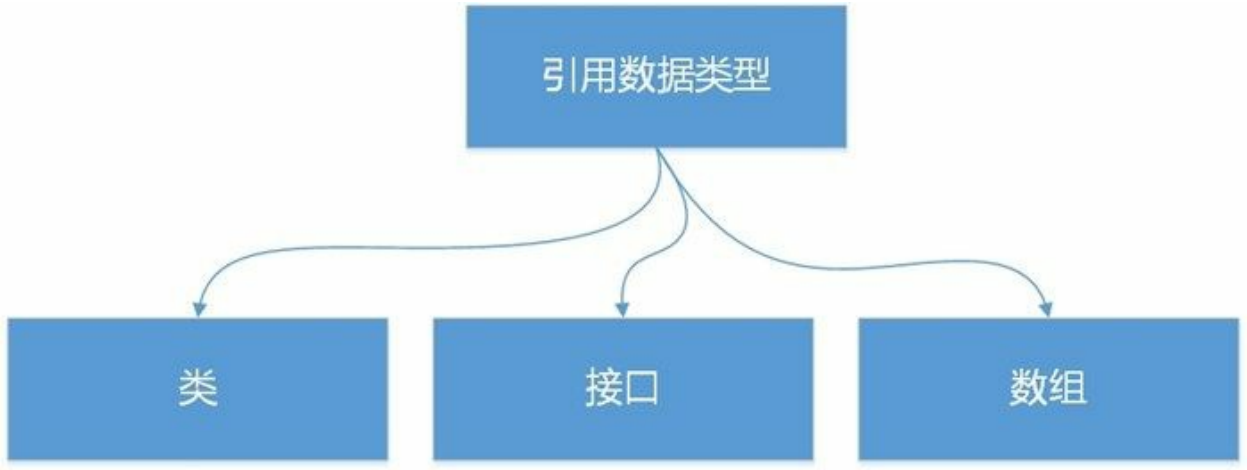


图**6-3** 引用数据类型

提示 Java中的引用数据类型，相当于C等语言中指针（pointer）类型，引用事实上就是指针，是指向一个对象的内存地址。引用数据类型变量中保持的是指向对象的内存地址。很多资料上提到Java不支持指针，事实上是不支持指针计算，而指针类型还是保留了下来，只是在Java中称为引用数据类型。

引用数据类型示例如下：

```
int x = 7;                    ①
int y = x;                    ②

String str1 = "Hello";        ③
String str2 = str1;           ④
str2 = "World";               ⑤
```

上述代码声明了两个基本数据类型（int）和两个引用数据类型（String）。当程序执行完第②行代码后，x值为7，x赋值给y，这时y的值也是7，它们的保持方式如图6-4所示，x和y两个变量值都是7，但是它们之间是独立的，任何一个变化都不会影响另一个。

当程序执行完第③行时，字符串“Hello”对象被创建，保持到内存地址0x12345678中，str1是引用类型变量，它保存的是内存地址0x12345678，这个地址指向“Hello”对象。

当程序执行完第④行时，str1变量内容（0x12345678）被赋值给str2是引用类型变量，这样一来str1和str2保存了相同的内存地址，都指向“Hello”对象。见图6-4所示，此时str1和str2本质上是引用一个对象，通过任何一个引用都可以修改对象本身。

当程序执行完第⑤行时，字符串“World”对象被创建，保持到内存地址0x23455678中，地址保存到str2变量中，此时，str1和str2不再指向相同内存地址，见图6-5所示。

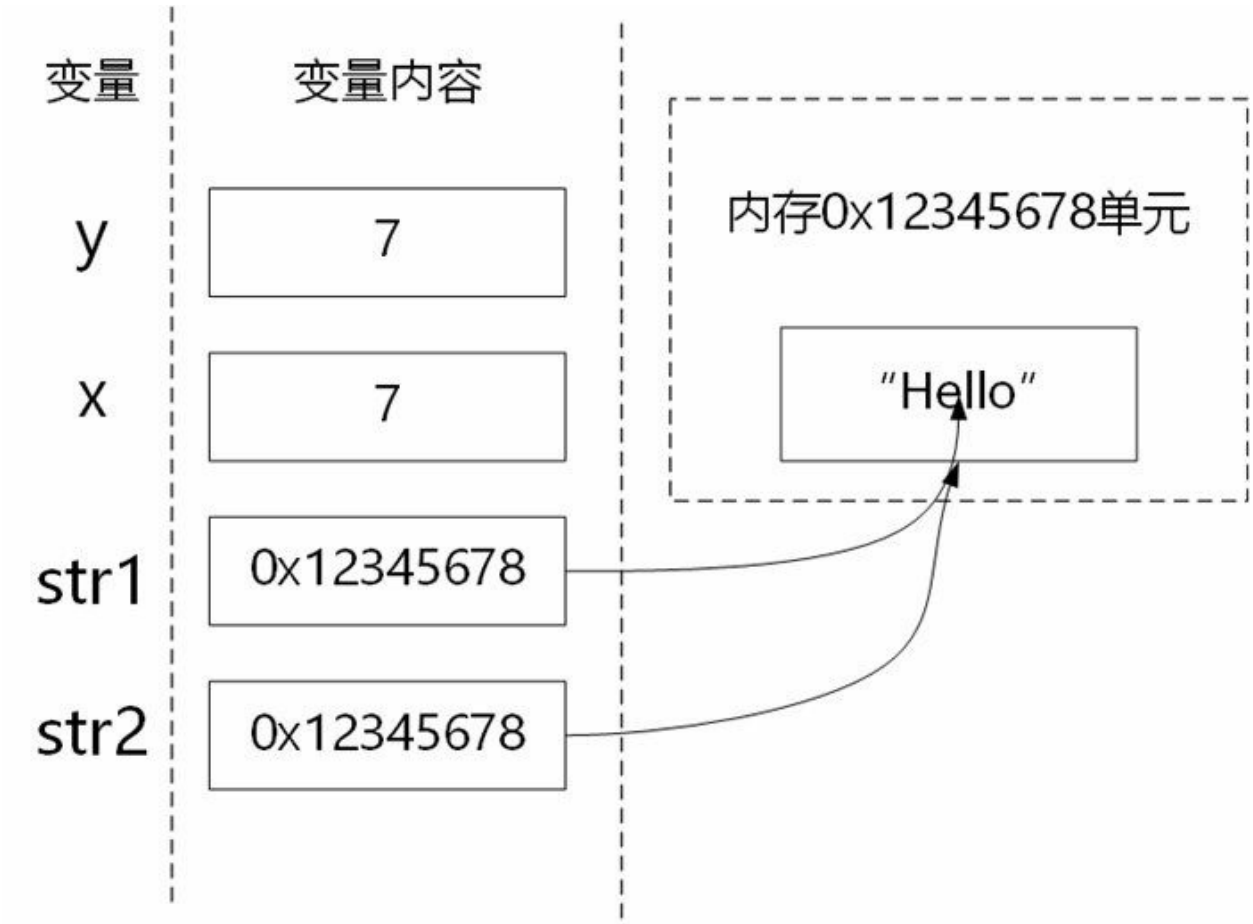


图**6-4**　引用数据类型赋值过程**1**

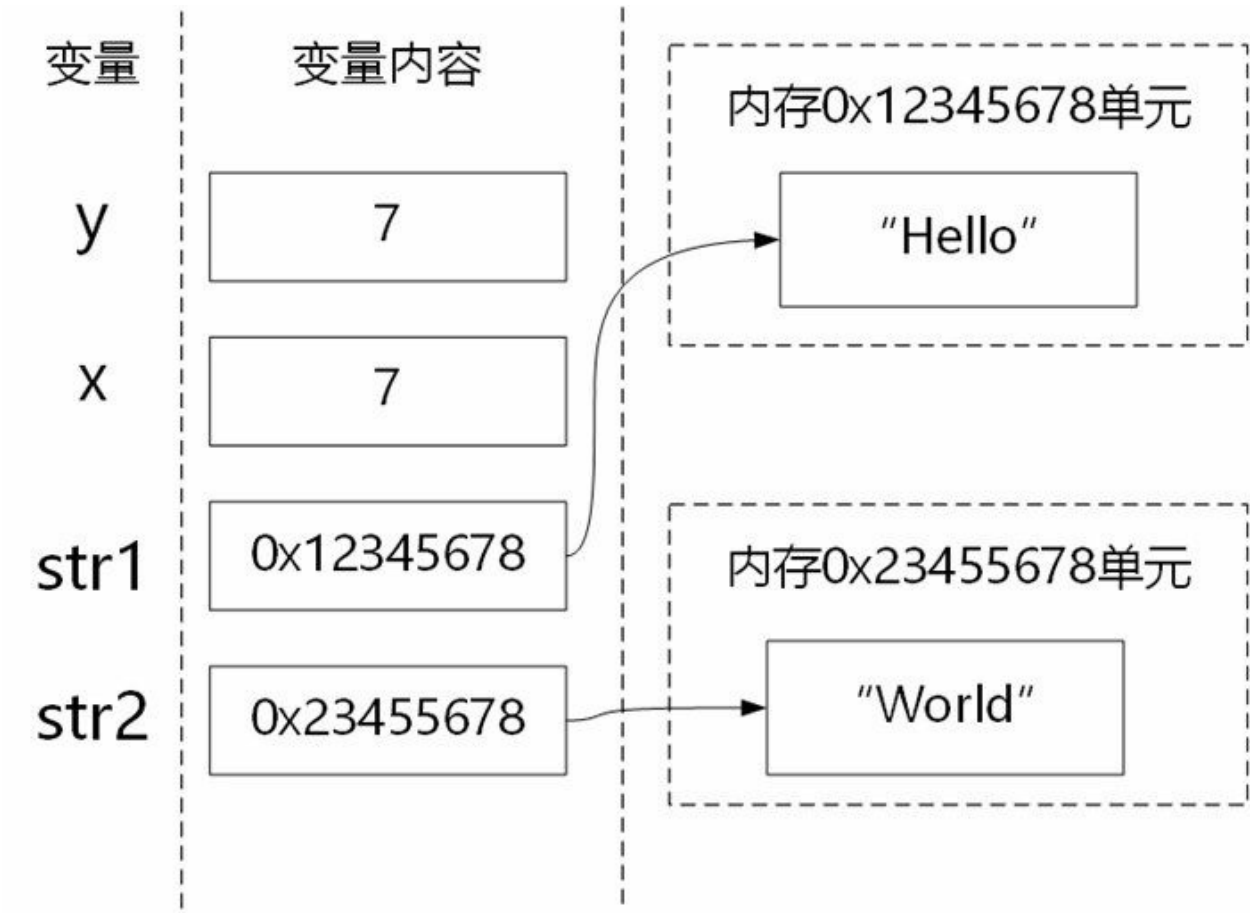


图**6-5**　引用数据类型赋值过程**2**

本章小结

本章主要介绍了Java中的数据类型，读者需要重点掌握基本数据类型，理解基本数据类型与引用数据类型的区别，熟悉数值类型如何互相转换。

第 7 章　运算符

Java语言中的运算符（也称操作符）在风格和功能上都与C 和C++极为相似。本章为大家介绍Java语言中一些主要的运算符，包括算术运算符、关系运算符、逻辑运算符、位运算符和其他运算符。

7.1 算术运算符

Java中的算术运算符主要用来组织数值类型数据的算术运算，按照参加运算的操作数的不同可以分为一元运算符和二元运算符。

7.1.1 一元运算符

算术一元运算符一共有3个，分别是-、++和--。具体说明参见表7-1。

表 **7-1** 一元算术运算符

运 算 符	名 称	说 明	例 子
-	取反符号	取反运算	b = -a
++	自加一	先取值再加一，或先加一再取值	a++或++a
--	自减一	先取值再减一，或先减一再取值	a--或--a

表7-1中，-a是对a取反运算，a++或a--是在表达式运算完后，再给a加一或减一。而++a或--a是先给a加一或减一，然后再进行表达式运算。

示例代码如下：

```
int a = 12;
System.out.println(-a);             ①
int b = a++;                        ②
System.out.println(b);
b = ++a;                            ③
System.out.println(b);
```

输出结果如下：

```
-12
12
14
```

上述代码第①行是-a，是把a变量取反，结果输出是-12。第②行代码是先把a赋值给b变量再加一，即先赋值后++，因此输出结果是12。第③行代码是把a加一，然后把a赋值给b变量，即先++后赋值，因此输出结果是14。

7.1.2 二元运算符

二元运算符包括：+、-、*、/和%，这些运算符对数值类型数据都有效，具体说明参见表7-2。

表 **7-2** 二元算术运算符

运 算 符	名 称	说 明	例 子
+	加	求 a 加 b 的和，还可用于 String 类型，进行字符串连接操作	a + b
-	减	求 a 减 b 的差	a - b
*	乘	求 a 乘以 b 的积	a * b
/	除	求 a 除以 b 的商	a / b
%	取余	求 a 除以 b 的余数	a % b

示例代码如下：

```
//声明一个字符类型变量
char charNum = 'A';
// 声明一个整数类型变量
int intResult = charNum + 1;            ①
System.out.println(intResult);

intResult = intResult - 1;
System.out.println(intResult);

intResult = intResult * 2;
System.out.println(intResult);

intResult = intResult / 2;
System.out.println(intResult);

intResult = intResult + 8;
intResult = intResult % 7;
System.out.println(intResult);

System.out.println("-------");

// 声明一个浮点型变量
double doubleResult = 10.0;
System.out.println(doubleResult);

doubleResult = doubleResult - 1;
System.out.println(doubleResult);

doubleResult = doubleResult * 2;
System.out.println(doubleResult);

doubleResult = doubleResult / 2;
System.out.println(doubleResult);

doubleResult = doubleResult + 8;
doubleResult = doubleResult % 7;
System.out.println(doubleResult);
```

输出结果如下：

```
66
65
130
65
3
```

```
-------
10.0
9.0
18.0
9.0
3.0
```

上述例子中分别对数值类型数据进行了二元运算，其中代码第①行将字符类型变量charNum与整数类型进行加法运算，参与运算的该字符（'A'）的Unicode编码为65。其他代码比较简单不再赘述。

7.1.3 算术赋值运算符

算术赋值运算符只是一种简写，一般用于变量自身的变化，具体说明参见表7-3。

表 **7-3** 算术赋值运算符

运 算 符	名 称	例 子
+=	加赋值	a += b、a += b+3
-=	减赋值	a -= b
*=	乘赋值	a *= b
/=	除赋值	a /= b
%=	取余赋值	a %= b

示例代码如下：

```
int a = 1;
int b = 2;
a += b;     // 相当于 a = a + b
System.out.println(a);

a += b + 3; // 相当于 a = a + b + 3
System.out.println(a);
a -= b;     // 相当于 a = a - b
System.out.println(a);

a *= b;     // 相当于 a=a*b
System.out.println(a);

a /= b;        // 相当于 a=a/b
System.out.println(a);

a %= b;        // 相当于 a=a%b
System.out.println(a);
```

输出结果如下：

```
3
8
6
12
```

```
6
0
```

上述例子分别对整型进行了+=、-=、*=、/=和%=运算，具体语句不再赘述。

7.2 关系运算符

关系运算是比较两个表达式大小关系的运算，它的结果是布尔类型数据，即true或false。关系运算符有6种：==、!=、>、<、>=和<=，具体说明参见表7-4。

表 **7-4** 关系运算符

运算符	名　称	说　明	例　子
==	等于	a 等于 b 时返回 true，否则返回 false。可以应用于基本数据类型和引用数据类型	a == b
!=	不等于	与==相反	a != b
>	大于	a 大于 b 时返回 true，否则返回 false，只应用于基本数据类型	a > b
<	小于	a 小于 b 时返回 true，否则返回 false，只应用于基本数据类型	a < b
>=	大于等于	a 大于等于 b 时返回 true，否则返回 false，只应用于基本数据类型	a >= b
<=	小于等于	a 小于等于 b 时返回 true，否则返回 false，只应用于基本数据类型	a <= b

提示　==和!=可以应用于基本数据类型和引用数据类型。当用于引用数据类型比较时，比较的是两个引用是否指向同一个对象，但在实际开发过程中多数情况下，只是比较对象的内容是否相当，不需要比较是否为同一个对象。

示例代码如下：

```
int value1 = 1;
int value2 = 2;

if (value1 == value2) {
    System.out.println("value1 == value2");
}

if (value1 != value2) {
    System.out.println("value1 != value2");
}

if (value1 > value2) {
    System.out.println("value1 > value2");
}

if (value1 < value2) {
    System.out.println("value1 < value2");
}

if (value1 <= value2) {
    System.out.println("value1 <= value2");
```

```
}
```

运行程序输出结果如下：

```
value1 != value2
value1 < value2
value1 <= value2
```

7.3 逻辑运算符

逻辑运算符是对布尔型变量进行运算，其结果也是布尔型，具体说明参见表7-5。

表 **7-5** 逻辑运算符

运算符	名 称	说 明	例 子
!	逻辑非	a 为 true 时，值为 false，a 为 false 时，值为 true	!a
&	逻辑与	ab 全为 true 时，计算结果为 true，否则为 false	a & b
|	逻辑或	ab 全为 false 时，计算结果为 false，否则为 true	a | b
&&	短路与	ab 全为 true 时，计算结果为 true，否则为 false。&&与&区别：如果 a 为 false，则不计算 b（因为不论 b 为何值，结果都为 false）	a && b
||	短路或	ab 全为 false 时，计算结果为 false，否则为 true。||与|区别：如果 a 为 true，则不计算 b（因为不论 b 为何值，结果都为 true）	a || b

提示　短路与（&&）和短路或（||）能够采用最优化的计算方式，从而提高效率。在实际编程时，应该优先考虑使用短路与和短路或。

示例代码如下：

```
int i = 0;
int a = 10;
int b = 9;

if ((a > b) || (i == 1)) {                    ①
    System.out.println("或运算为 真");
} else {
    System.out.println("或运算为 假");
}

if ((a < b) && (i == 1)) {                    ②
    System.out.println("与运算为 真");
} else {
    System.out.println("与运算为 假");
}

if ((a > b) || (a++ == --b)) {                ③
    System.out.println("a = " + a);
    System.out.println("b = " + b);
}
```

上述代码运行输出结果如下：

```
或运算为 真
与运算为 假
a = 10, b = 9
```

其中，第①行代码进行短路计算，由于(a > b)是true，后面的表达式(i == 1)不再计算，输出的结果为真。类似地，第②行代码也进行短路计算，由于(a < b)是false，后面的表达式(i == 1)不再计算，输出的结果为假。

代码第③行中在条件表达中掺杂了++和—运算，由于(a > b)是true，后面的表达式(a++ == --b)不再计算，所以最后是a = 10, b = 9。如果把短路或（||）改为逻辑或（|），那么输出的结果就是a = 11, b = 8了。

7.4 位运算符

位运算是以二进位（bit）为单位进行运算的，操作数和结果都是整型数据。位运算符有如下几个运算符：&、|、^、~、>>、<<和>>>，以及相应的赋值运算符，具体说明参见表7-6。

表 **7-6** 位运算符

运算符	名　称	例　子	说　明
~	位反	~x	将 x 的值按位取反
&	位与	x&y	x 与 y 位进行位与运算
|	位或	x|y	x 与 y 位进行位或运算
^	位异或	x^y	x 与 y 位进行位异或运算
>>	有符号右移	x>>a	x 右移 a 位，高位采用符号位补位
<<	左移	x<<a	x 左移 a 位，低位用 0 补位
>>>	无符号右移	x >>> a	x 右移 a 位，高位用 0 补位
&=	位与等于	a &= b	等价于 a = a&b
|=	位或等于	a |= b	等价于 a = a|b
^=	位异或等于	a ^= b	等价于 a = a^b
<<=	左移等于	a <<= b	等价于 a = a<<b
>>=	右移等于	a >>= b	等价于 a = a>>b
>>>=	右移等于	a >>>= b	等价于 a = a>>>b

注意　无符号右移>>>运算符仅被允许用在int和long整数类型, 如果用于short或byte数据, 则数据在位移之前，转换为int类型后再进行位移计算。

位运算示例代码：

```
byte a = 0B00110010;    //十进制50                              ①
byte b = 0B01011110;    //十进制94                              ②

System.out.println("a | b = " + (a | b));     // 0B01111110    ③
System.out.println("a & b = " + (a & b));     // 0B00010010    ④
System.out.println("a ^ b = " + (a ^ b));     // 0B01101100    ⑤
System.out.println("~b = " + (~b));         // 0B10100001      ⑥

System.out.println("a >> 2 = " + (a >> 2));     // 0B00001100    ⑦
System.out.println("a >> 1 = " + (a >> 1));     // 0B00011001    ⑧
System.out.println("a >>> 2 = " + (a >>> 2));   // 0B00001100    ⑨
System.out.println("a << 2 = " + (a << 2));     // 0B11001000    ⑩
System.out.println("a << 1 = " + (a << 1));     // 0B01100100    ⑪
```

```
int c = -12;                                          ⑫
System.out.println("c >>> 2 = " + (c >>> 2));     ⑬
System.out.println("c >> 2 = " + (c >> 2));       ⑭
```

输出结果如下：

```
a | b = 126
a & b = 18
a ^ b = 108
~b = -95
a >> 2 = 12
a >> 1 = 25
a >>> 2 = 12
a << 2 = 200
a << 1 = 100
c >>> 2 = 1073741821
c >> 2 = -3
```

上述代码第①行和第②行分别定义了byte变量a和b，为了便于查看代码采用二进制整数表示。

代码第③行中表达式(a | b)进行位或运算，结果是二进制的0B01111110。a和b按位进行或计算，只要有一个为1，这一位就为1，否则为0。

代码第④行(a & b)是进行位与运算，结果是二进制的0B00010010。a和b按位进行与计算，只有两位全部为1，这一位才为1，否则为0。

代码第⑤行(a ^ b)是进行位异或运算，结果是二进制的0B01101100。a和b按位进行异或计算，只有两位相反时这一位才为1，否则为0。

代码第⑦行(a >> 2)是进行有符号右位移2位运算，结果是二进制的0B00001100。a的低位被移除掉，由于是正数符号位是0，高位空位用0补。类似代码第⑧行(a >> 1)是进行右位移1位运算，结果是二进制的0B00011001。

代码第⑨行(a >>> 2)是进行无符号右位移2位运算，与代码第⑦行不同的是，无论是否有数符号位，高位空位用0补，所以在正数情况下>>和>>>运算结果是一样的。

代码第⑩行(a << 2)是进行左位移2位运算，结果是二进制的0B11001000。a的高位被移除掉，低位用0补位。类似代码第⑪行(a << 1)是进行左位移1位运算，结果是二进制的0B01100100。

代码第⑫声明int类型负数。右位移（>>>和>>）在负数情况下差别比较大。代码第⑬行的(c >>> 2)表达式输出结果是1073741821，这是一个如此大的正数，从一个负数变成一个正数，这说明无符号右位移对于负数计算会导致精度的丢失。而有符号右位移对于负数的计算是正确的，见代码第⑭行。

提示　有符号右移n位，相当于操作数除以2^n，例如代码第⑦行(a >> 2)表达式相当于(a / 2^2)，a = 50所以结果等于12，类似的还有代码第⑧行和第⑭行。另外，左位移n位，相当于操作数乘以2^n，例如代码第⑩行(a << 2)表达式相当于(a * 2^2)，a = 50所以结果等于200，类似的还有代码第⑪行。

7.5 其他运算符

除了前面介绍的主要运算符，Java还有一些其他运算符。

- 三元运算符（?:）。例如x?y:z;，其中x、y和z都为表达式。
- 小括号。起到改变表达式运算顺序的作用，它的优先级最高。
- 中括号。数组下标。
- 引用号（.）。对象调用实例变量或实例方法的操作符，也是类调用静态变量或静态方法的操作符。
- 赋值号（=）。赋值是用等号运算符（=）进行的。
- instanceof。判断某个对象是否为属于某个类。
- new。对象内存分配运算符。
- 箭头（->）。Java 8新增加的，用来声明Lambda表达式。
- 双冒号（::）。Java 8新增加的，用于Lambda表达式中方法的引用。

示例代码如下：

```
import java.util.Date;

public class HelloWorld {

    public static void main(String[] args) {

        int score = 80;
        String result = score > 60 ? "及格" : "不及格"; // 三元运算符（? : ）
        System.out.println(result);

        Date date = new Date();                 // new运算符可以创建Date对象
        System.out.println(date.toString());    //通过.运算符调用方法

    }
}
```

此外，还有一些鲜为人知的运算符，随着学习的深入用到后再为大家介绍，这里就不再赘述了。

7.6 运算符优先级

在一个表达式计算过程中，运算符的优先级非常重要。表7-7中从上到小，运算符的优先级从高到低，同一行具有相同的优先级。二元运算符计算顺序从左向右，但是先级15的赋值运算符的计算顺序从右向左的。

表 **7-7** **Java**运算符优先级

优先级	运算符
1	. (引用号)　小括号　　中括号
2	++　　--　　-(数值取反)　　~(位反)　　!(逻辑非)　类型转换小括号
3	*　　/　　%
4	+　　-
5	<<　　>>　　>>>
6	<　　>　　<=　　>=　　instanceof
7	==　　!=
8	&(逻辑与、位与)
9	^(位异或)
10	|(逻辑或、位或)
11	&&
12	||
13	?:
14	->
15	=　　*=　　/=　　%=　　+=　　-=　　<<=　　>>=　　>>>=　　&=　　^=　　|=

总结　运算符优先级大体顺序，从高到低是：算术运算符→位运算符→关系运算符→逻辑运算符→赋值运算符。

本章小结

通过对本章内容的学习，读者可以了解到Java语言的基本运算符，这些运算符包括算术运算符、关系运算符、逻辑运算符、位运算符和其他运算符。

第 **8** 章　控制语句

程序设计中的控制语句有三种，即顺序、分支和循环语句。Java程序通过控制语句来管理程序流，完成一定的任务。程序流是由若干个语句组成的，语句可以是一条单一的语句，也可以是一个用大括号（{}）括起来的复合语句。Java中的控制语句有以下几类：

- 分支语句：if和switch
- 循环语句：while、do-while和for
- 跳转语句：break、continue、return和throw

8.1 分支语句

分支语句提供了一种控制机制，使得程序具有了“判断能力”，能够像人类的大脑一样分析问题。分支语句又称条件语句，条件语句使部分程序可根据某些表达式的值被有选择地执行。Java编程语言提供了if和switch两种分支语句。

8.1.1 if语句

由if语句引导的选择结构有if结构、if-else结构和else-if结构三种。

01. if结构

    如果条件表达式为true就执行语句组，否则就执行if结构后面的语句。如果语句组只有一条语句，可以省略大括号，当从编程规范角度不要省略大括号，省略大括号会是程序的可读性变差。语法结构如下：

```
if (条件表达式) {
    语句组
}
```

    if结构示例代码如下：

```
int score = 95;
if (score >= 85) {
    System.out.println("您真优秀！");
}
if (score < 60) {
    System.out.println("您需要加倍努力！");
}
if ((score >= 60) && (score < 85)) {
    System.out.println("您的成绩还可以，仍需继续努力！");
}
```

    程序运行结果如下：

```
您真优秀！
```

02. if-else结构

    所有的语言都有这个结构，而且结构的格式基本相同，语句如下：

```
if (条件表达式) {
   语句组1
} else {
   语句组2
}
```

    当程序执行到if语句时，先判断条件表达式，如果值为true，则执行语句组1，然后跳过else语句及语句组2，继续执行后面的语句。如果条件表达式的值为false，则忽略语句组1而直接执行语句组2，然后继续执行后面的语句。

    if-else结构示例代码如下：

```
int score = 95;
if (score < 60) {
    System.out.println("不及格");
} else {
    System.out.println("及格");
}
```

程序运行结果如下：

```
及格
```

03. else-if结构

else-if结构如下：

```
if (条件表达式1) {
    语句组1
} else if (条件表达式2) {
    语句组2
} else if (条件表达式3) {
    语句组3
...
} else if (条件表达式n) {
    语句组n
} else {
    语句组n＋1
}
```

可以看出，else-if结构实际上是if-else结构的多层嵌套，它明显的特点就是在多个分支中只执行一个语句组，而其他分支都不执行，所以这种结构可以用于有多种判断结果的分支中。

else-if结构示例代码如下：

```
int testScore = 76;
char grade;
if (testScore >= 90) {
    grade = 'A';
} else if (testScore >= 80) {
    grade = 'B';
} else if (testScore >= 70) {
    grade = 'C';
} else if (testScore >= 60) {
    grade = 'D';
} else {
    grade = 'F';
}
System.out.println("Grade = " + grade);
```

输出结果如下：

```
Grade = C
```

其中char grade是声明字符变量，然后经过判断最后结果是C。

8.1.2 **switch**语句

switch提供多分支程序结构语句。下面先介绍一下switch语句基本形式的语法结构，如下所示：

```
switch (表达式) {
    case 值1:
        语句组1
    case 值2:
        语句组2
    case 值3:
        语句组3
            ...
    case 判断值n:
        语句组n
    default:
        语句组n＋1
}
```

switch语句中“表达式”计算结果只能是int、byte、short和char类型，不能是long更不能其他的类型。每个case后面只能跟一个int、byte、short和char类型的常量，default语句可以省略。

当程序执行到switch语句时，先计算条件表达式的值，假设值为A，然后拿A与第1个case语句中的值1进行匹配，如果匹配则执行语句组1，语句组执行完成后不跳出switch，只有遇到break才跳出switch。如果A没有与第1个case语句匹配，则与第2个case语句进行匹配，如果匹配则执行语句组2，以此类推，直到执行语句组n。如果所有的case语句都没有执行，就执行default的语句组n＋1，这时才跳出switch。

示例代码如下：

```
int testScore = 75;

char grade;
switch (testScore / 10) {            ①
case 9:
    grade = '优';
    break;
case 8:
    grade = '良';
    break;
case 7:              // 7是贯通的      ②
case 6:
    grade = '中';
    break;
default:
    grade = '差';
}
System.out.println("Grade = " + grade);
```

输出结果如下：

```
Grade = 中
```

上述代码将100分制转换为：“优”、“良”、“中”、“差”评分制，其中7分和6分都是“中”成绩，把case 7和case 6当成一种情况考虑。代码第①行计算表达式获得0~9分数值。代码第②行的case 7是贯通的，就它的后面不加break，程序流执行完当前case后，则会进入到下一个case，因此本例中case 7和case 6都执行相同的代码。

8.2 循环语句

循环语句能够使程序代码重复执行。Java支持三种循环构造类型：while、do-while、和for。for和while循环是在执行循环体之前测试循环条件，而do-while是在执行循环体之后测试循环条件。这就意味着for和while循环可能连一次循环体都未执行，而do-while将至少执行一次循环体。另外Java 5之后推出for-each循环语句，for-each循环是for循环的变形，它是专门为集合遍历而设计的，注意for-each并不是一个关键字。

8.2.1 **while**语句

while语句是一种先判断的循环结构，格式如下：

```
while (循环条件) {
    语句组
}
```

while循环没有初始化语句，循环次数是不可知的，只要循环条件满足，循环就会一直进行下去。

下面看一个简单的示例，代码如下：

```
int i = 0;
while (i * i < 100000) {
    i++;
}

System.out.println("i = " + i);
System.out.println("i * i = " + (i * i));
```

输出结果如下：

```
i = 317
i * i = 100489
```

上述程序代码的目的是找到平方数小于100000的最大整数。使用while循环需要注意几点，while循环条件语句中只能写一个表达式，而且是一个布尔型表达式，那么如果循环体中需要循环变量，就必须在while语句之前对循环变量进行初始化。本例中先给i赋值为0，然后在循环体内部必须通过语句更改循环变量的值，否则将会发生死循环。

8.2.2 **do-while**语句

do-while语句的使用与while语句相似，不过do-while语句是事后判断循环条件结构，语句格式如下：

```
do {
    语句组
} while (循环条件)
```

do-while循环没有初始化语句，循环次数是不可知的，不管循环条件是否满足，都会先执行一次循环体，然后再判断循环条件。如果条件满足则执行循环体，不满足则停止循环。

下面看一个示例代码：

```
int i = 0;
do {
    i++;
} while (i * i < 100000);

System.out.println("i = " + i);
System.out.println("i * i = " + (i * i));
```

输出结果如下：

```
i = 317
i * i = 100489
```

该示例与上一节的示例是一样的，都是找到平方数小于100000的最大整数。输出结果也是一样的。

8.2.3 for语句

for语句是应用最广泛、功能最强的一种循环语句。一般格式如下：

```
for (初始化; 循环条件; 迭代) {
    语句组
}
```

for语句执行流程如图8-1所示，首先会先执行初始化语句，它的作用是初始化循环变量和其他变量，然后程序会判断循环条件是否满足，如果满足，则继续执行循环体并计算迭代语句，之后再判断循环条件，如此反复，直到判断循环条件不满足时跳出循环。

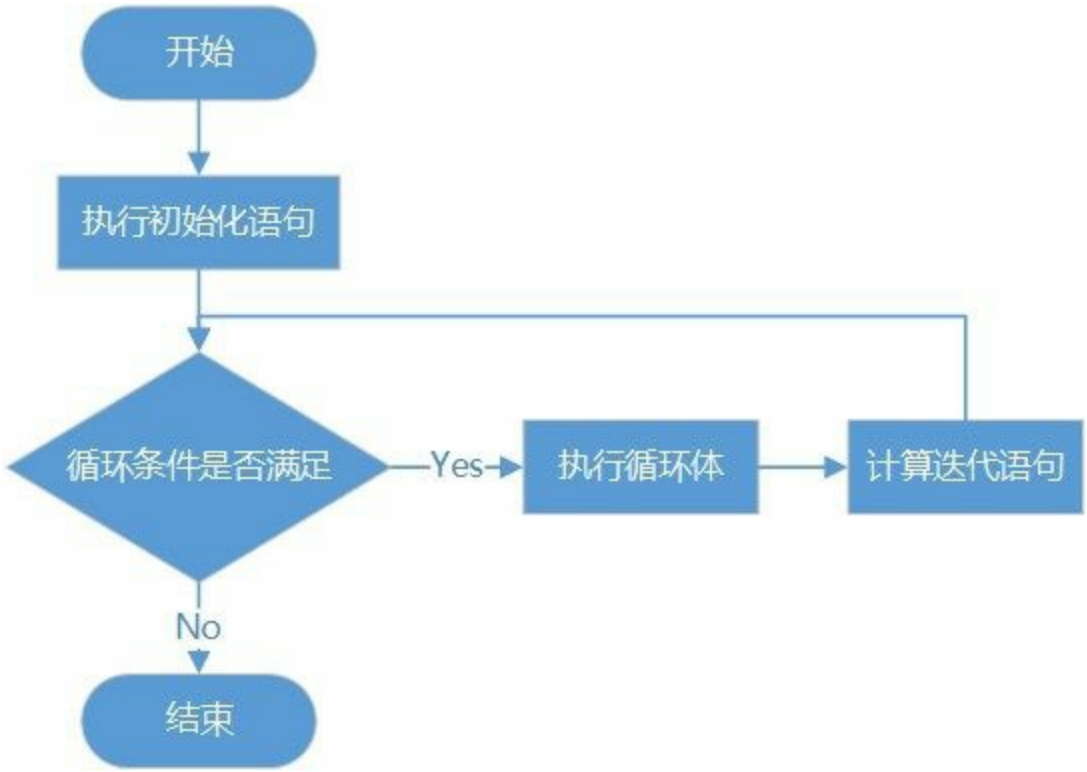


图**8-1 for**循环执行流程图

以下示例代码是计算1~9的平方表程序：

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

for (int i = 1; i < 10; i++)  {
    System.out.printf("%d x %d = %d", i, i, i * i);
    //打印一个换行符，实现换行
    System.out.println();
}
```

输出结果如下：

```
---------
1 x 1 = 1
2 x 2 = 4
3 x 3 = 9
4 x 4 = 16
5 x 5 = 25
6 x 6 = 36
7 x 7 = 49
8 x 8 = 64
9 x 9 = 81
```

在这个程序的循环部分初始时，给循环变量i赋值为1，每次循环都要判断i的值是否小于10，如果为true，则执行循环体，然后给i加1。因此，最后的结果是打印出1~9的平方，不包括10。

提示　初始化、循环条件以及迭代部分都可以为空语句（但分号不能省略），三者均为空的时候，相当于一个无限循环。代码如下：

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

另外，在初始化部分和迭代部分，可以使用逗号语句来进行多个操作。逗号语句是用逗号分隔的语句序列，如下程序代码所示：

```
int x;
int y;

for (x = 0, y = 10; x < y; x++, y--) {
    System.out.printf("(x,y) = (%d, %d)", x, y);
    // 打印一个换行符，实现换行
    System.out.println();
}
```

输出结果如下：

```
(x,y) = (0,10)
(x,y) = (1,9)
(x,y) = (2,8)
(x,y) = (3,7)
(x,y) = (4,6)
```

8.2.4　for-each语句

Java 5之后提供了一种专门用于遍历集合的for循环——for-each循环。使用for-each循环不必按照for的标准套路编写代码，只需要提供一个集合就可以遍历。

假设有一个数组，采用for语句遍历数组的方式如下：

```
// 声明并初始化int数组
int[] numbers = { 43, 32, 53, 54, 75, 7, 10 };

System.out.println("----for-------");
// for语句
for (int i = 0; i < numbers.length; i++) {
    System.out.println("Count is:" + numbers[i]);
}
```

上述语句int[] numbers = { 43, 32, 53, 54, 75, 7, 10 }声明并初始化了10个元素数组集合，目前大家只需要知道当初始化数组时，要把相同类型的元素放到{…}中并且用逗号分隔（,）即可，关于数组集合会在后面第9章详细介绍。numbers.length是获得数组的长度，length是数组的属性，numbers[i]是通过数组下标访问数组元素。

那么采用for-each循环语句遍历数组的方式如下：

```
// 声明并初始化int数组
int[] numbers = { 43, 32, 53, 54, 75, 7, 10 };

System.out.println("----for each----");
// for-each语句
for (int item : numbers) {
    System.out.println("Count is:" + item);
}
```

从示例中可以发现，item不是循环变量，它保存了集合中的元素，for-each语句将集合中的元素一一取出来，并保存到item中，这个过程中不需要使用循环变量，通过数组下标访问数组中的元素。可见for-each语句在遍历集合的时候要简单方便得多。

8.3 跳转语句

跳转语句能够改变程序的执行顺序，可以实现程序的跳转。Java有4种跳转语句：break、continue、throw和return。本节重点介绍break和continue语句的使用。throw和return将后面章节介绍。

8.3.1 break语句

break语句可用于上一节介绍的while、repeat-while和for循环结构，它的作用是强行退出循环体，不再执行循环体中剩余的语句。

在循环体中使用break语句有两种方式：带有标签和不带标签。语法格式如下：

```
break;           //不带标签
break label;     //带标签，label是标签名
```

不带标签的break语句使程序跳出所在层的循环体，而带标签的break语句使程序跳出标签指示的循环体。

下面看一个示例，代码如下：

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

for (int i = 0; i < numbers.length; i++) {
    if (i == 3) {
        //跳出循环
        break;
    }
    System.out.println("Count is: " + i);
}
```

在上述程序代码中，当条件i==3的时候执行break语句，break语句会终止循环，程序运行的结果如下：

```
Count is: 0
Count is: 1
Count is: 2
```

break还可以配合标签使用，示例代码如下：

```
label1: for (int x = 0; x < 5; x++) {          ①
    for (int y = 5; y > 0; y--) {              ②
        if (y == x) {
            //跳转到label1指向的循环
            break label1;                      ③
        }
        System.out.printf("(x,y) = (%d,%d)", x, y);
        // 打印一个换行符，实现换行
        System.out.println();
    }
}
System.out.println("Game Over!");
```

默认情况下，break只会跳出最近的内循环（代码第②行for循环）。如果要跳出代码第①行的外循环，可以为外循环添加一个标签label1，注意在定义标签的时候后面跟一个冒号。代码第③行的break语句

后面指定了label1标签，这样当条件满足执行break语句时，程序就会跳转出label1标签所指定的循环。

程序运行结果如下：

```
(x,y) = (0,5)
(x,y) = (0,4)
(x,y) = (0,3)
(x,y) = (0,2)
(x,y) = (0,1)
(x,y) = (1,5)
(x,y) = (1,4)
(x,y) = (1,3)
(x,y) = (1,2)
Game Over!
```

如果break后面没有指定外循环标签，则运行结果如下：

```
(x,y) = (0,5)
(x,y) = (0,4)
(x,y) = (0,3)
(x,y) = (0,2)
(x,y) = (0,1)
(x,y) = (1,5)
(x,y) = (1,4)
(x,y) = (1,3)
(x,y) = (1,2)
(x,y) = (2,5)
(x,y) = (2,4)
(x,y) = (2,3)
(x,y) = (3,5)
(x,y) = (3,4)
(x,y) = (4,5)
Game Over!
```

比较两种运行结果，就会发现给break添加标签的意义，添加标签对于多层嵌套循环是很有必要的，适当使用可以提高程序的执行效率。

8.3.2 continue语句

continue语句用来结束本次循环，跳过循环体中尚未执行的语句，接着进行终止条件的判断，以决定是否继续循环。对于for语句，在进行终止条件的判断前，还要先执行迭代语句。

在循环体中使用continue语句有两种方式可以带有标签，也可以不带标签。语法格式如下：

```
continue          //不带标签
continue label    //带标签，label是标签名
```

下面看一个示例，代码如下：

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

for (int i = 0; i < numbers.length; i++) {
    if (i == 3) {
        continue;
    }
    System.out.println("Count is: " + i);
}
```

在上述程序代码中，当条件i==3的时候执行continue语句，continue语句会终止本次循环，循环体中continue之后的语句将不再执行，接着进行下次循环，所以输出结果中没有3。程序运行结果如下：

```
Count is: 0
Count is: 1
Count is: 2
Count is: 4
Count is: 5
Count is: 6
Count is: 7
Count is: 8
Count is: 9
```

带标签的continue语句示例代码如下：

```
label1: for (int x = 0; x < 5; x++)     ①
    for (int y = 5; y > 0; y--) {       ②
        if (y == x) {
            continue label1;            ③
        }
        System.out.printf("(x,y) = (%d,%d)", x, y);
        System.out.println();
    }
}
System.out.println("Game Over!");
```

默认情况下，continue只会跳出最近的内循环（代码第②行for循环），如果要跳出代码第①行的外循环，可以为外循环添加一个标签label1，然后在第③行的continue语句后面指定这个标签label1，这样当条件满足执行continue语句时，程序就会跳转出外循环。

程序运行结果如下：

```
(x,y) = (0,5)
(x,y) = (0,4)
(x,y) = (0,3)
(x,y) = (0,2)
(x,y) = (0,1)
(x,y) = (1,5)
(x,y) = (1,4)
(x,y) = (1,3)
(x,y) = (1,2)
(x,y) = (2,5)
(x,y) = (2,4)
(x,y) = (2,3)
(x,y) = (3,5)
(x,y) = (3,4)
(x,y) = (4,5)
Game Over!
```

由于跳过了x == y，因此下面的内容没有输出。

```
(x,y) = (1,1)
(x,y) = (2,2)
(x,y) = (3,3)
(x,y) = (4,4)
```

本章小结

通过对本章内容的学习，读者可以了解到Java语言的控制语句，其中包括分支语句（if和switch）、循环语句（while、do-while、for和for-each）和跳转语句（break和continue）等。

第 **9** 章　数组

在计算机语言中数组是非常重要的集合类型，大部分计算机语言中数组具有如下三个基本特性：

01. 一致性：数组只能保存相同数据类型元素，元素的数据类型可以是任何相同的数据类型。

02. 有序性：数组中的元素是有序的，通过下标访问。

03. 不可变性：数组一旦初始化，则长度（数组中元素的个数）不可变。

在Java中数组的下标是从零开始的，事实上很多计算机语言的数组下标从零开始的。Java数组下标访问运算符是中括号，如intArray[0]，表示访问intArray数组的第一个元素，0是第一个元素的下标。

另外，Java中的数组本身是引用数据类型，它的长度属性是length。数组可以分为：一维数组和多维数组，下面先介绍一维数组。

9.1 一维数组

当数组中每个元素都只带有一个下标时，这种数组就是“一维数组”。数组是引用数据类型，引用数据类型在使用之前一定要做两件事情：声明和初始化。

9.1.1 数组声明

数组的声明就宣告这个数组中元素类型，数组的变量名。

注意　数组声明完成后，数组的长度还不能确定，JVM（Java虚拟机）还没有给元素分配内存空间。

数组声明语法如下：

```
元素数据类型[] 数组变量名;
元素数据类型 数组变量名[];
```

可见数组的声明有两种形式：一种是两个中括号（[]）跟在元素数据类型之后；另一种是两个中括号（[]）跟在变量名之后。

提示　从面向对象角度看，Java更推荐采用第一种声明方式，因为它把“元素数据类型[]”看成是一个整体类型，即数组类型。而第二种是C语言数组声明方式。

数组声明示例如下：

```
int intArray[];
float[] floatArray;
String strArray[];
Date[] dateArray;
```

9.1.2 数组初始化

声明完成就要对数组进行初始化，数组初始化的过程就是为数组每一个元素分配内存空间，并为每一个元素提供初始值。初始化之后数组的长度就确定下来不能再变化了。

提示　有些计算机语言虽然提供了可变类型数组，它的长度是可变的，这种数组本质上是创建了一个新的数组对象，并非是原始数组的长度发生了变化。

数组初始化可以分为静态初始化和动态初始化。

01. 静态初始化

静态初始化就是将数组的元素放到大括号中，元素之间用逗号（,）分隔。示例代码如下：

```
int[] intArray;
//静态初始化int数组
intArray = {21,32,43,45};

String[] strArray;
//静态初始化Stirng数组
strArray = {"张三","李四","王五","董六"};

//声明同时初始化数组
int intArray[] = {21,32,43,45};
```

```
String strArray[] = {"张三","李四","王五","董六"};
```

静态初始化是在已知数组的每一个元素内容情况下使用的。很多情况下数据是从数据库或网络中获得的，在编程时不知道元素有多少，更不知道元素的内容，此时可采用动态初始化。

02. 动态初始化

动态初始化使用new运算符分配指定长度的内存空间，语法如下：

```
new 元素数据类型[数组长度] ;
```

示例代码如下：

```
int intArray[];
// 动态初始化int数组
intArray = new int[4];                          ①
intArray[0] = 21;
intArray[1] = 32;
intArray[2] = 43;
intArray[3] = 45;                               ②

// 动态初始化String数组
String[] stringArray = new String[4];           ③
// 初始化数组中元素
stringArray[0] = "张三";
stringArray[1] = "李四";
stringArray[2] = "王五";
stringArray[3] = "董六";                        ④
```

上述代码第①行和第③行通过new运算符分配了4个元素的内存空间。

提示　new分配数组内存空间后，数组中的元素内容是什么呢？答案是数组类型的默认值，不同类型默认值是不同的，如表9-1所示。

表 **9-1**　数据类型默认值

基本类型	默认值
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
boolean	false
引用	null

当代码第①行执行完成，intArray数组内容如图9-1(a)所示，intArray数组中的所有元素都是0，根据需要会动态添加元素内容，代码第②行执行完成，intArray数组内容如图9-1(b)所示。

图**9-1　intArray**数组

当代码第③行执行完成，stringArray数组内容如图9-2(a)所示，stringArray数组中所有元素都是null，随着每一个元素被初始化和赋值，代码第④行执行完之后每个元素都有不同内容，这里需要注意的是引

用类型数组，每一个元素保存都是指向实际对象的内存地址，如图9-2(b)所示，每个对象还需要创建和初始化过程，有关对象创建和初始化内容，将在后面章节详细介绍。

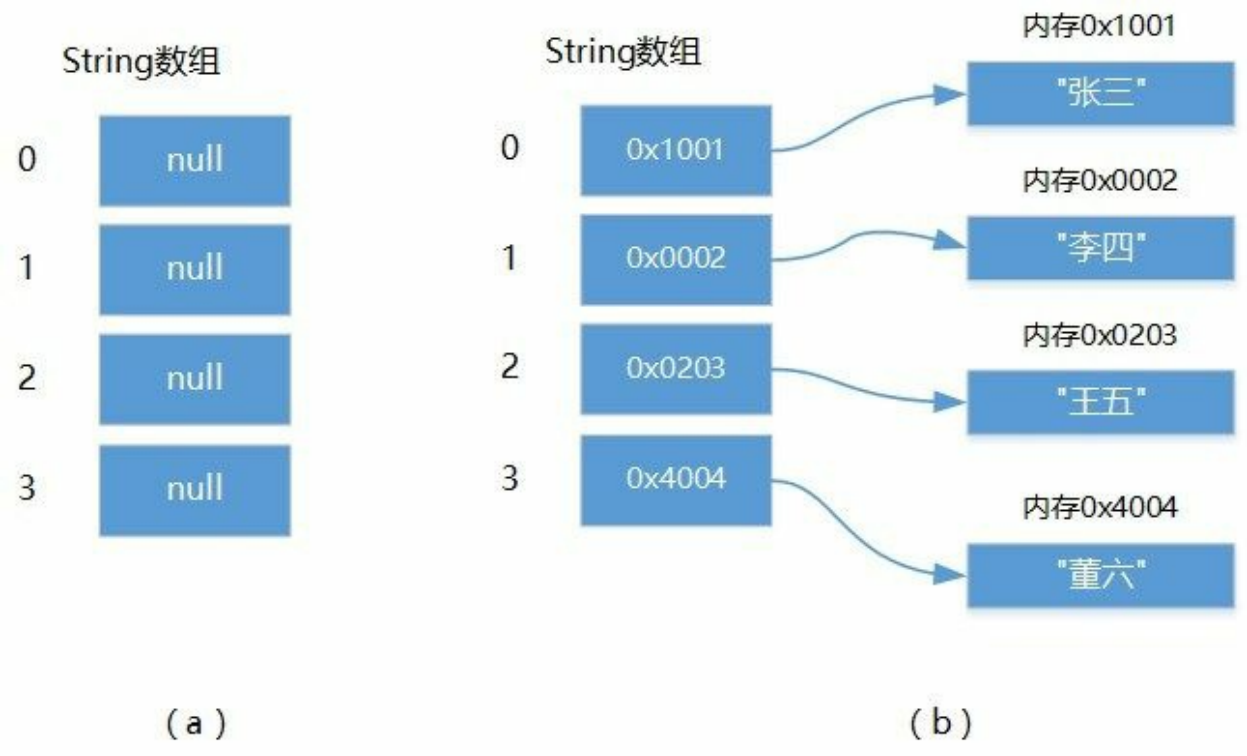


图**9-2　stringArray**数组

9.1.3　案例：数组合并

数组长度是不可变，要想合并两个不同的数组，不能通过在一个数组的基础上追加另一个数组实现。需要创建一个新的数组，新数组长度是两个数组长度之和。然后再将两个数组的内容导入到新数组中。

下面具体看看实现代码：

```
public class HelloWorld {

    public static void main(String[] args) {

        // 两个待合并数组
        int array1[] = { 20, 10, 50, 40, 30 };
        int array2[] = { 1, 2, 3 };

        // 动态初始化数组，设置数组的长度是array1和array2长度之和
        int array[] = new int[array1.length + array2.length];

        // 循环添加数组内容
        for (int i = 0; i < array.length; i++) {

            if (i < array1.length) {                    ①
                array[i] = array1[i];                   ②
            } else {
                array[i] = array2[i - array1.length];   ③
            }
        }
```

```
        System.out.println("合并后:");
        for (int element : array) {
            System.out.printf("%d ", element);
        }

    }
}
```

上述代码第①行是判断当前循环变量i是否小于array1.length，在此条件下将array1数组内容导入新数组，见代码第②行。当array1数组内容导入完成后，再通过代码第③行将另一个数组array2导入到新数组，其中array2下标应该是i - array1.length。

9.2 多维数组

当数组中每个元素又可以带有多个下标时，这种数组就是“多维数组”。本节重点介绍二维数组。

9.2.1 二维数组声明

Java中声明二维数组需要有两个中括号，具体有三种语法如下：

```
元素数据类型[][] 数组变量名;
元素数据类型 数组变量名[][];
元素数据类型[] 数组变量名[];
```

三种形式中前两种比较好理解，最后一种形式看起来有些古怪。数组声明示例如下：

```
int[][] array1;
int array1[][];
int[] array1[];
```

9.2.2 二维数组的初始化

二维数组的初始化也可以分为静态初始化和动态初始化。

01. 静态初始化

    静态初始化示例如下：

    ```
    int intArray[][] = { { 1, 2, 3 }, { 11, 12, 13 }, { 21, 22, 23 }, { 31, 32, 33 } };
    ```

    上述代码创建并初始化了一个4×3二维数组，理解Java中的多维数组应该从数组的数组的角度出发。首先将intArray看成是一个一维数组，它有4个元素，如图9-3所示，其中第1个元素是{ 1, 2, 3 }，第2个元素是{ 11, 12, 13 }，第3个元素是{ 21, 22, 23 }，第4个元素是{ 31, 32, 33 }。然后再分别考虑每一个元素，{ 1, 2, 3 }表示形式说明它是一个int类型的一维数组，其他3个元素也是一维int类型数组。

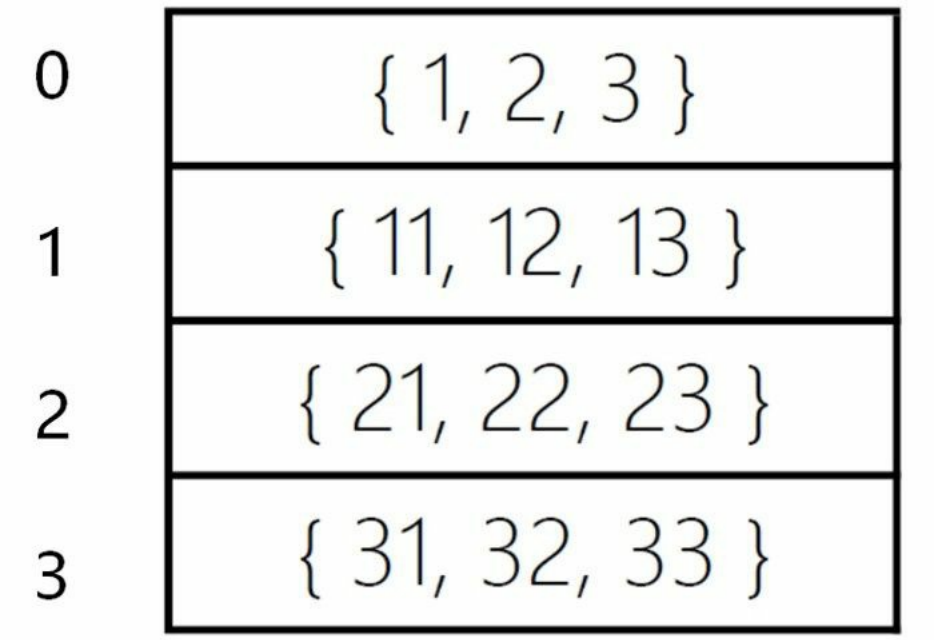


图**9-3　intArray**二维数组

提示　严格意义上说Java中并不存在真正意义上的多维数组，只是一维数组，不过数组中的元素也是数组，以此类推三维数组就是数组的数组的数组了，例如{ { {1, 2}, {3} }, { {21}, {22, 23} } }表示一个三维数组。

02. 动态初始化

动态初始化二维数组语法如下：

```
new 元素数据类型[高维数组长度] [低维数组长度] ;
```

高维数组就是最外面的数组，低维数组是每一个元素的数组。动态创建并初始化一个4×3二维数组示例代码如下：

```
int[][] intArray = new int[4][3];
```

二维数组的下标[4][3]有两个，前面的[4]是高维数组下标索引，后面的[3]是低维数组下标索引。4×3二维数组的每一个元素的下标索引，如图9-4(a)所示。由于低维数组是int类型，所以初始化完后所有元素全部是0，如图9-4(b)所示。

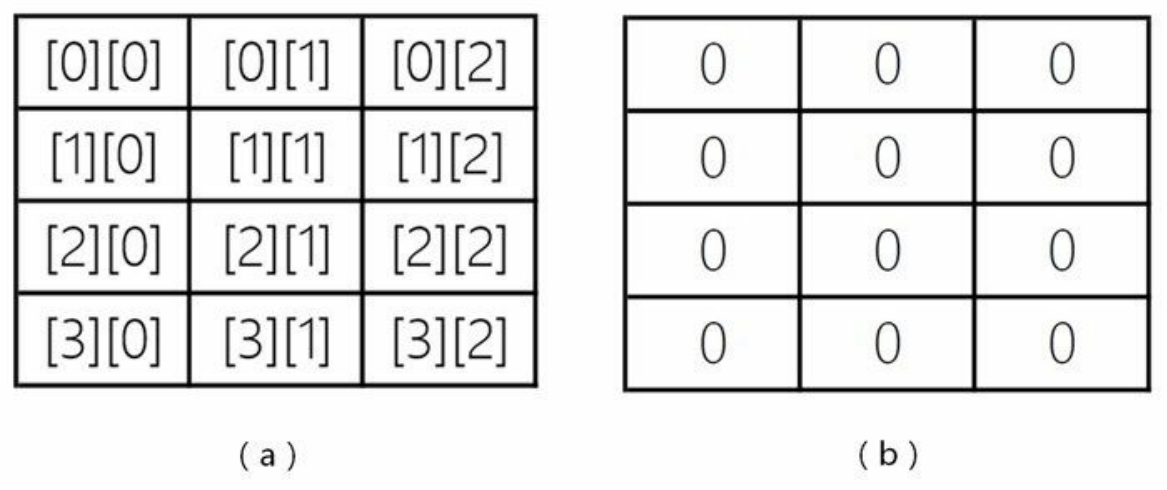


图**9-4**　二维数组动态初始化

二维数组示例如下：

```
public class HelloWorld {

    public static void main(String[] args) {

        // 静态初始化二维数组
        int[][] intArray = {
                { 1, 2, 3 },
                { 11, 12, 13 },
                { 21, 22, 23 },
                { 31, 32, 33 } };

        // 动态初始化二维数组
        double[][] doubleArray = new double[4][3];

        // 计算数组intArray元素的平方根，结果保存到doubleArray
        for (int i = 0; i < intArray.length; i++) {
            for (int j = 0; j < intArray[i].length; j++) {
                // 计算平方根
                doubleArray[i][j] = Math.sqrt(intArray[i][j]);            ①
            }
        }

        // 打印数组doubleArray
        for (int i = 0; i < doubleArray.length; i++) {
            for (int j = 0; j < doubleArray[i].length; j++) {
                System.out.printf("[%d][%d] = %f", i, j, doubleArray[i][j]);
                System.out.print('\t');
            }
            System.out.println();
        }
    }
}
```

代码第①行是中Math.sqrt(intArray[i][j])表达式是计算平方根，Math是java.lang包中提供的用于数学计算类，它提供很多常用的数学计算方法，sqrt是计算平方根，如取绝对值的abs、幂运算的pow等。

9.2.3　不规则数组

由于Java多维数组是数组的数组，因此会衍生出一种不规则数组，规则的4×3二维数组有12个元素，而不规则数组就不一定了。如下代码静态初始化了一个不规则数组。

```
int intArray[][] = { { 1, 2 }, { 11 }, { 21, 22, 23 }, { 31, 32, 33 } };
```

高维数组是4个元素，但是低维数组元素个数不同，如图9-5所示，其中第1个数组有两个元素，第2个数组有1个元素，第3个数组有3个元素，第4个数组有3个元素。这就是不规则数组。

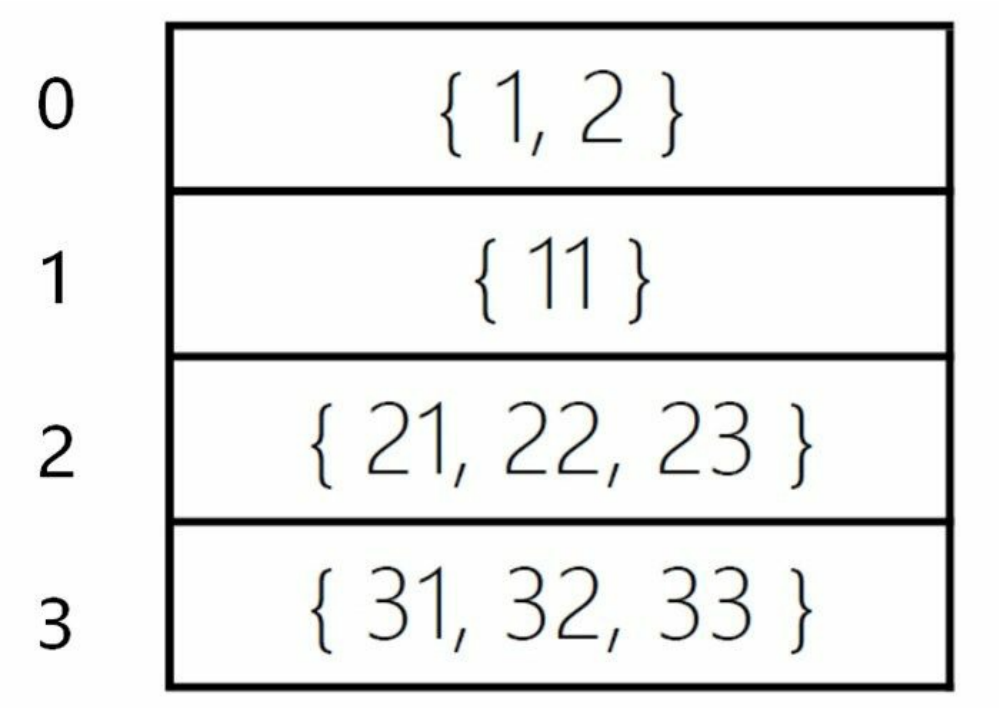


图**9-5**　不规则数组

动态初始化不规则数组比较麻烦，不能使用new int[4][3]语句，而是先初始化高维数组，然后再分别逐个初始化低维数组。代码如下：

```
int intArray[][] = new int[4][]; //先初始化高维数组为4
//逐一初始化低维数组
intArray[0] = new int[2];
intArray[1] = new int[1];
intArray[2] = new int[3];
intArray[3] = new int[3];
```

从上述代码初始化数组完成之后，不是有12个元素而是9个元素，它们的下标索引如图9-6所示，可见其中下标[0][2]、[1][1]和[1][2]是不存在的，如果试图访问它们则会抛出下标越界异常。

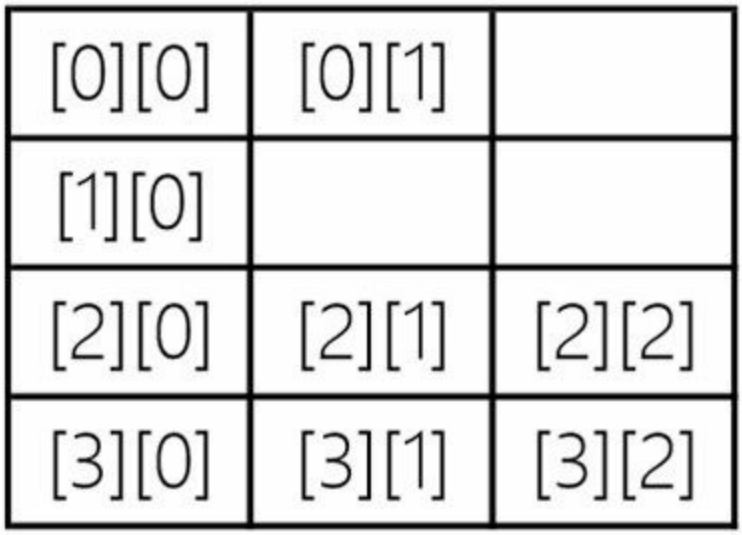


图**9-6**　不规则数组访问

提示　下标越界异常（ArrayIndexOutOfBoundsException）是试图访问不存在的下标时引发的。例如一个一维array数组如果有10个元素，那么表达式array[10]就会发生下标越界异常，这是因为数组下标是从0开始的，最后一个元素下标是数组长度减1，所以array[10]访问的元素是不存在的。

下面介绍一个不规则数组的示例：

```
public class HelloWorld {

    public static void main(String[] args) {

        int intArray[][] = new int[4][]; //先初始化高维数组为4
        //逐一初始化低维数组
        intArray[0] = new int[2];
        intArray[1] = new int[1];
        intArray[2] = new int[3];
        intArray[3] = new int[3];

        //for循环遍历
        for (int i = 0; i < intArray.length; i++) {
            for (int j = 0; j < intArray[i].length; j++) {
                intArray[i][j] = i + j;
            }
        }
        //for-each循环遍历
        for (int[] row : intArray) {        ①
            for (int column : row) {        ②
                System.out.print(column);
                //在元素之间添加制表符,
                System.out.print('\t');
            }
            //一行元素打印完成后换行
            System.out.println();
        }

        //System.out.println(intArray[0][2]); //发生运行期错误     ③
    }
}
```

不规则数组访问和遍历可以使用for和for-each循环，但要注意下标越界异常发生。上述代码第①行和第②行采用for-each循环遍历不规则数组，其中代码第①行for-each循环取出的数据是int数组，所以row类型是int[]。代码第②行for-each循环取出的数据是int数据，所以column的类型int。

另外，注意代码第③行试图访问intArray[0][2]元素，由于[0][2]不存在所以会发生下标越界异常。

本章小结

本章介绍了Java的数组，包括一维数组和多维数组，读者要重点掌握一维数组的声明、初始化和使用，了解二维数组的声明、初始化和使用。另外，还需要了解不规则数组。

第 **10** 章　字符串

由字符组成的一串字符序列，称为“字符串”，在前面的章节中也多次用到了字符串，本章将重点介绍。

10.1 Java中的字符串

Java中的字符串是由双引号括起来的多个字符，下面示例都是表示字符串常量：

```
"Hello World"                                                                    ①
"\u0048\u0065\u006c\u006c\u006f\u0020\u0057\u006f\u0072\u006c\u0064"             ②
"世界你好"                                                                       ③
"A"                                                                              ④
""                                                                               ⑤
```

Java中的字符采用Unicode编码，所以Java字符串可以包含中文等亚洲字符，见代码第③行的"世界你好"字符串。代码第②行的字符串是用Unicode编码表示的字符串，事实上它表示的也是"Hello World"字符串，可通过System.out.print方法将Unicode编码表示的字符串输出到控制台，则会看到Hello World字符串。

另外，单个字符如果用双引号括起来，那它表示的是字符串，而不是字符了，见代码第④行的"A"是表示字符串A，而不是字符A。

> 注意　字符串还有一个极端情况，就代码第⑤行的""表示空字符串，双引号中没有任何内容，空字符串不是null，空字符串是分配内存空间，而null是没有分配内存空间。

Java SE提供了三个字符串类：String、StringBuffer和StringBuilder。String是不可变字符串，StringBuffer和StringBuilder是可变字符串。

10.2　使用API文档

Java中很多类，每一个类又有很多方法和变量，通过查看Java API文档能够知道这些类、方法和变量如何使用。在5.2.2节介绍过使用javadoc指令生成API文档，Java官方为Java SE提供了已经生成HTML的API文档。作为Java程序员应该熟悉如何使用API文档。

本节介绍一下如何使用Java SE的API文档。Java官方提供了Java 8在线API文档，网址是http://docs.oracle.com/javase/8/docs/api/，页面如图10-1所示。

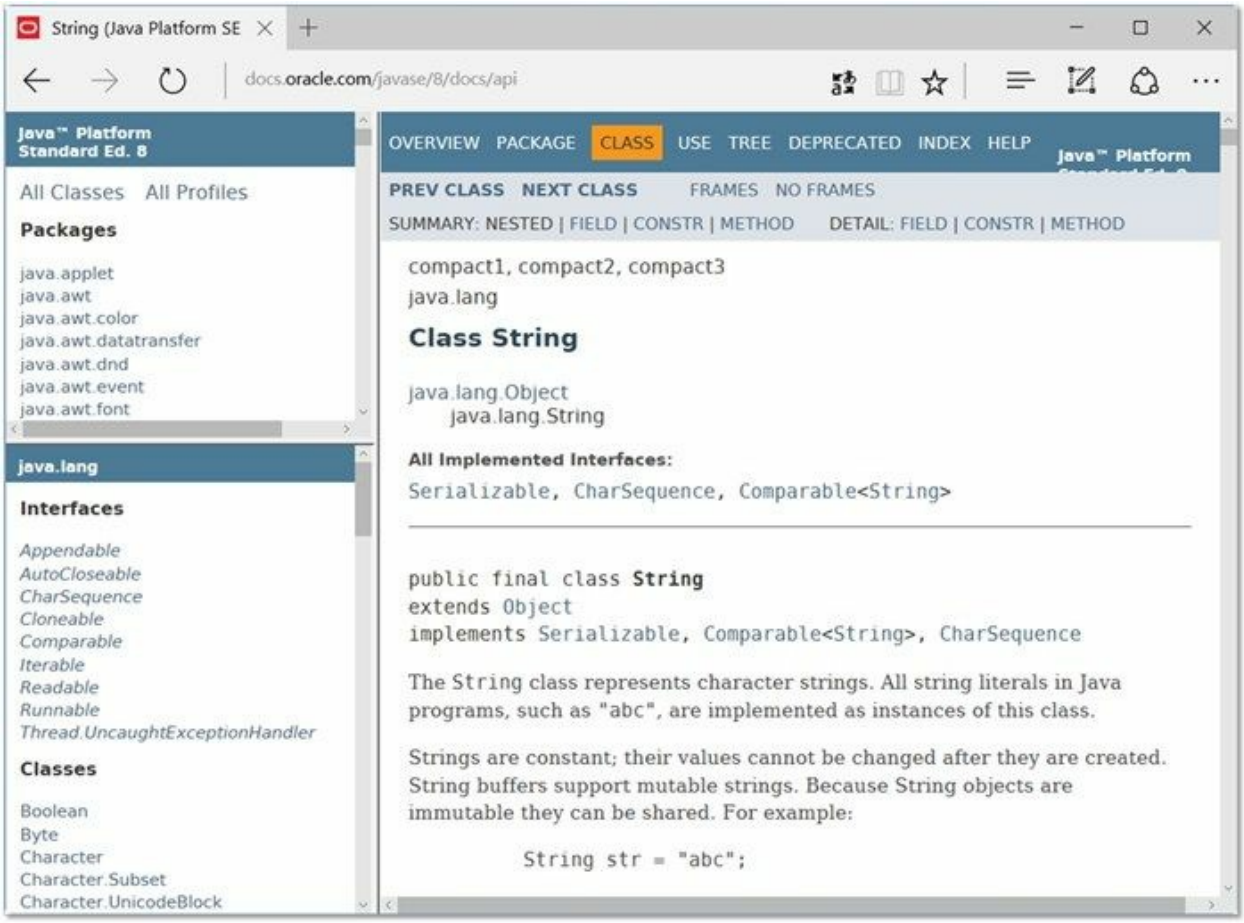


图**10-1**　**Java 8**在线**API**文档

> 提示　很多读者希望能够有离线的中文Java API文档，但Java官方只提供了Java 6的中文API文档，该文件下载地址是http://download.oracle.com/technetwork/java/javase/6/docs/zh/api.zip，下载完成后解压api.zip文件，找到其中的index.html文件，双击就会在浏览器中打开API文档了。

下面介绍一下如何使用API文档，熟悉一下API文档页面中的各个部分含义，如图10-2所示，类和接口中，斜文字体显示是接口，正常字体才是类。

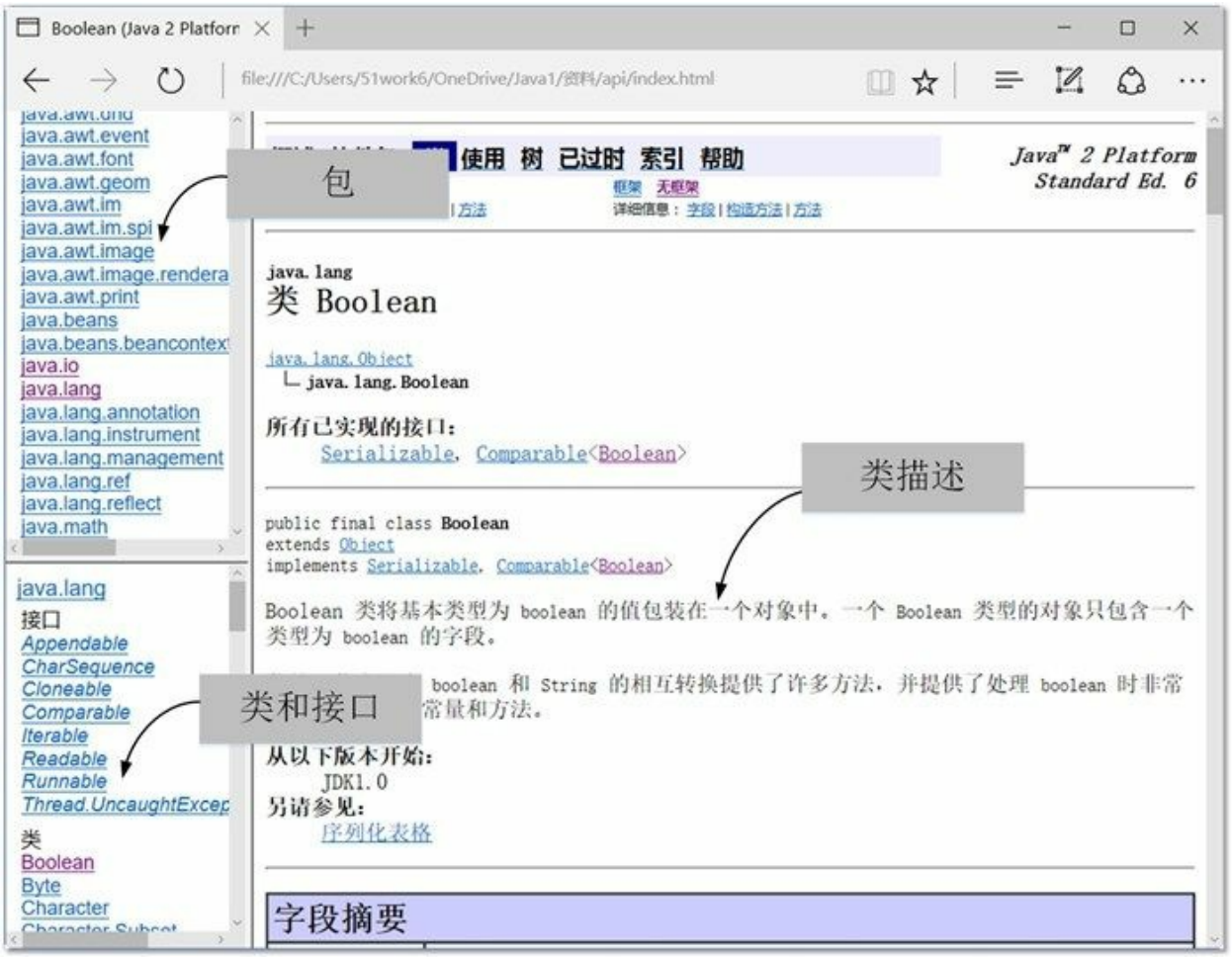


图**10-2　API**文档页面各个部分

在类窗口还有很多内容，向下拖曳滚动条会看到如图10-3所示页面，其中“字段摘要”描述了类中的实例变量和静态变量；“构造方法摘要”描述了类中所有构造方法；“方法摘要”描述了类中所有方法。这些“摘要”只是一个概要说明，单击链接可以进入到该主题更加详细的描述，如图10-4所示单击了compareTo方法看到的详细信息。

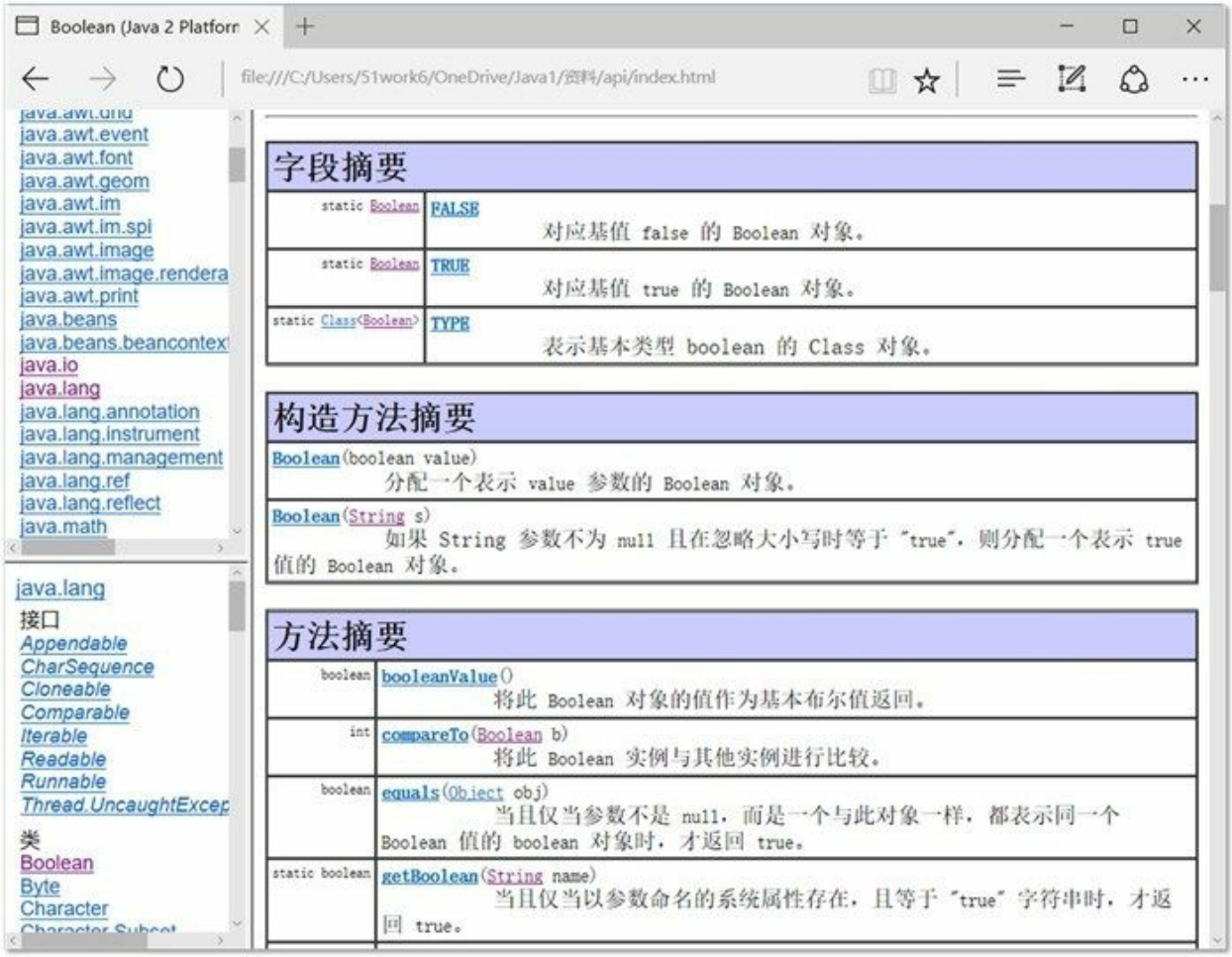


图**10-3**　类窗口页面其他内容

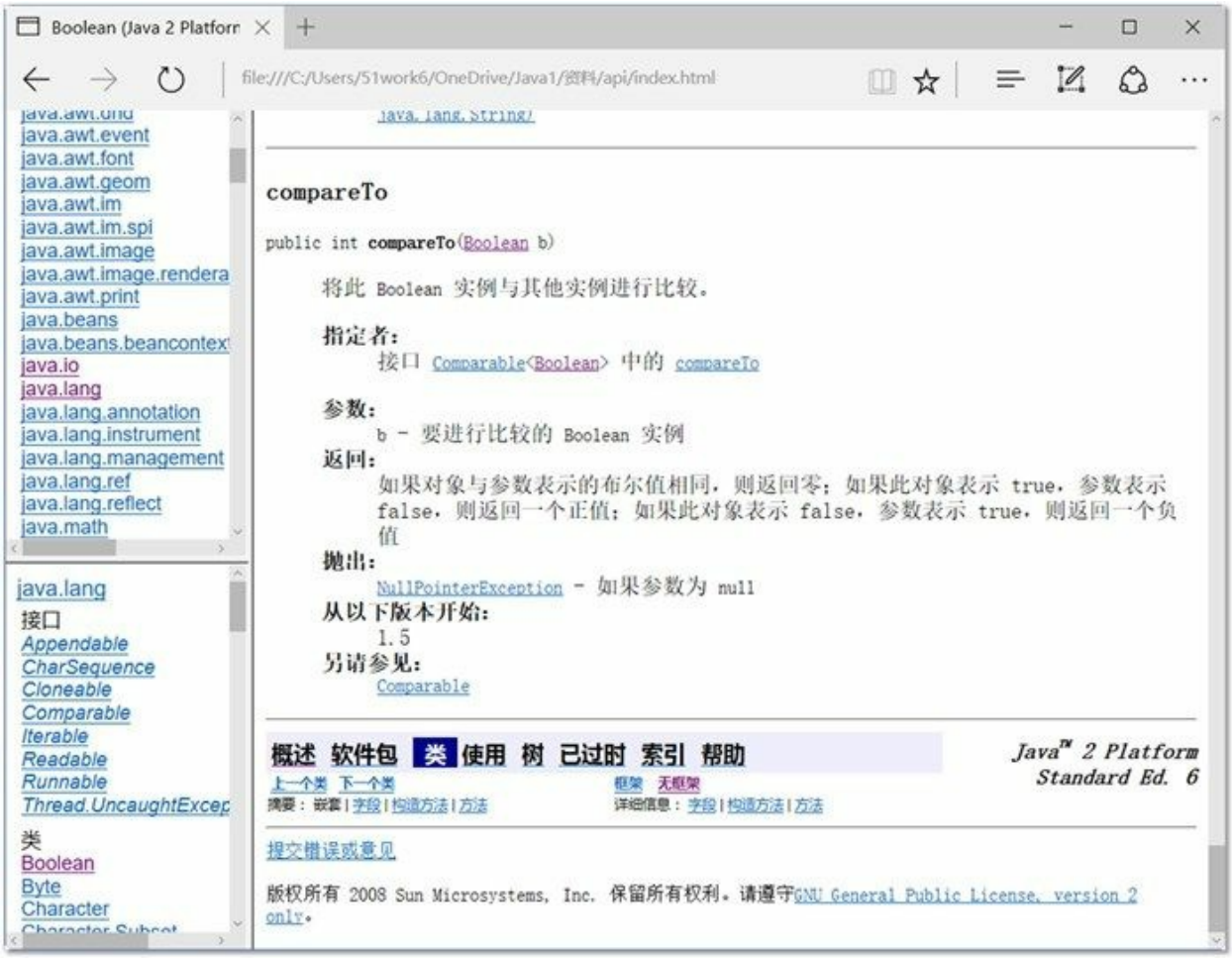


图**10-4　compareTo**方法详细描述

查询API的一般流程是：找包→找类或接口→查看类或接口→找方法或变量。读者可以尝试查找一下String、StringBuffer和StringBuilder这些字符串类的API文档，熟悉一下这些类的用法。

10.3 不可变字符串

很多计算机语言都提供了两种字符串，即不可变字符串和可变字符串，它们区别在于当字符串进行拼接等修改操作时，不可变字符串会创建新的字符串对象，而可变字符串不会创建新对象。

10.3.1 String

Java中不可变字符串类是String，属于java.lang包，它也是Java非常重要的类。

> 提示　java.lang包中提供了很多Java基础类，包括Object、Class、String和Math等基本类。在使用java.lang包中的类时不需要引入（import）该包，因为它是由解释器自动引入的。当然引入java.lang包程序也不会有编译错误。

创建String对象可以通过构造方法实现，常用的构造方法：

- String()：使用空字符串创建并初始化一个新的String对象。
- String(String original)：使用另外一个字符串创建并初始化一个新的 String 对象。
- String(StringBuffer buffer)：使用可变字符串对象（StringBuffer）创建并初始化一个新的 String 对象。
- String(StringBuilder builder)：使用可变字符串对象（StringBuilder）创建并初始化一个新的 String 对象。
- String(byte[] bytes)：使用平台的默认字符集解码指定的byte数组，通过byte数组创建并初始化一个新的 String 对象。
- String(char[] value)：通过字符数组创建并初始化一个新的 String 对象。
- String(char[] value, int offset, int count)：通过字符数组的子数组创建并初始化一个新的 String 对象；offset参数是子数组第一个字符的索引，count参数指定子数组的长度。

创建字符串对象示例代码如下：

```
// 创建字符串对象
String s1 = new String();
String s2 = new String("Hello World");
String s3 = new String("\u0048\u0065\u006c\u006c\u006f\u0020\u0057\u006f\u0072\u006c\u0064");
System.out.println("s2 = " + s2);
System.out.println("s3 = " + s3);

char chars[] = { 'a', 'b', 'c', 'd', 'e' };
// 通过字符数组创建字符串对象
String s4 = new String(chars);
// 通过子字符数组创建字符串对象
String s5 = new String(chars, 1, 4);
System.out.println("s4 = " + s4);
System.out.println("s5 = " + s5);

byte bytes[] = { 97, 98, 99 };
// 通过byte数组创建字符串对象
String s6 = new String(bytes);
System.out.println("s6 = " + s6);
System.out.println("s6字符串长度 = " + s6.length());
```

输出结果：

```
s2 = Hello World
s3 = Hello World
s4 = abcde
s5 = bcde
s6 = abc
s6字符串长度 = 3
```

上述代码中s2和s3都是表示Hello World字符串，获得字符串长度方法是length()，其他代码比较简单，这里不再赘述。

10.3.2 字符串池

在前面的学习过程中细心的读者可能会发现，前面的示例代码中获得字符串对象时都是直接使用字符串常量，但Java中对象是使用new关键字创建，字符串对象也可以使用new关键字创建，代码如下：

```
String s9 = "Hello";                //字符串常量
String s7 = new String("Hello");    //使用new关键字创建
```

使用new关键字与字符串常量都能获得字符串对象，但它们之间有一些区别。先看下面代码运行结果：

```
String s7 = new String("Hello");        ①
String s8 = new String("Hello");        ②

String s9 = "Hello";                    ③
String s10 = "Hello";                   ④

System.out.printf("s7 == s8 : %b%n", s7 == s8);
System.out.printf("s9 == s10: %b%n", s9 == s10);
System.out.printf("s7 == s9 : %b%n", s7 == s9);
System.out.printf("s8 == s9 : %b%n", s8 == s9);
```

输出结果：

```
s7 == s8 : false
s9 == s10: true
s7 == s9 : false
s8 == s9 : false
```

==运算符比较的是两个引用是否指向相同的对象，从上面的运行结果可见，s7和s8指的是不同对象，s9和s10指向的是相同对象。

这是为什么？Java中的不可变字符串String常量，采用字符串池（String Pool）管理技术，字符串池是一种字符串驻留技术。采用字符串常量赋值时（见代码第③行），如图10-5所示，会字符串池中查找"Hello"字符串常量，如果已经存在把引用赋值给s9，否则创建"Hello"字符串对象，并放到池中。根据此原理，可以推定s10与s9是相同的引用，指向同一个对象。但此原理并不适用于new所创建的字符串对象，代码运行到第①行后，会创建"Hello"字符串对象，而它并没有放到字符串池中。代码第②行又创建了一个新的"Hello"字符串对象，s7和s8是不同的引用，指向不同的对象。

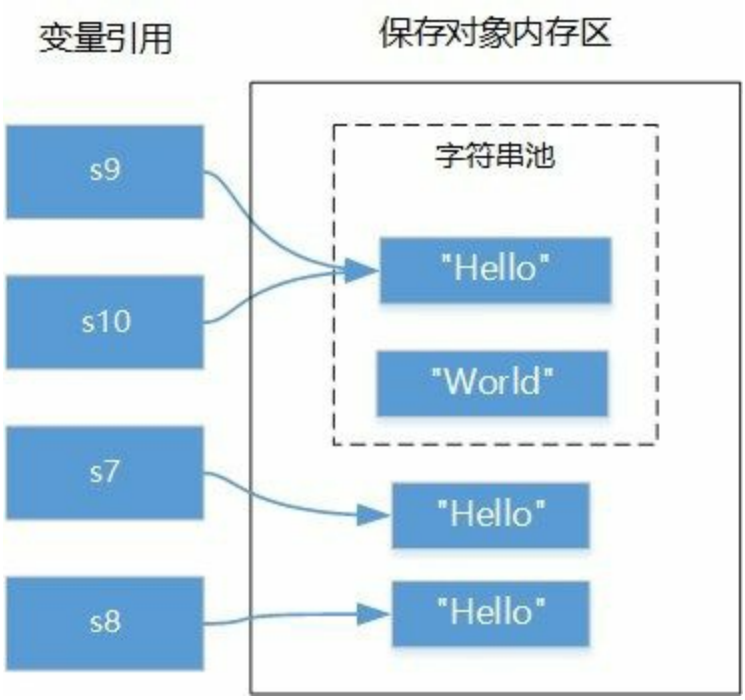


图**10-5**　字符串池

10.3.3　字符串拼接

String字符串虽然是不可变字符串，但也可以进行拼接只是会产生一个新的对象。String字符串拼接可以使用+运算符或String的concat(String str)方法。+运算符优势是可以连接任何类型数据拼接成为字符串，而concat方法只能拼接String类型字符串。

字符串拼接示例如下：

```
String s1 = "Hello";
// 使用+运算符连接
String s2 = s1 + " ";                          ①
String s3 = s2 + "World";                      ②
System.out.println(s3);

String s4 = "Hello";
// 使用+运算符连接，支持+=赋值运算符
s4 += " ";                                     ③
s4 += "World";                                 ④
System.out.println(s4);

String s5 = "Hello";
// 使用concat方法连接
s5 = s5.concat(" ").concat("World");           ⑤
System.out.println(s5);

int age = 18;
String s6= "她的年龄是" + age + "岁。";        ⑥
System.out.println(s6);

char score = 'A';
String s7= "她的英语成绩是" + score;          ⑦
System.out.println(s7);

java.util.Date now = new java.util.Date();     ⑧
```

```
//对象拼接自动调用toString()方法
String s8= "今天是：" + now;                    ⑨
System.out.println(s8);
```

输出结果：

```
Hello World
Hello World
Hello World
她的年龄是18岁。
她的英语成绩是A
今天是：Thu May 25 16:25:40 CST 2017
```

上述代码第①~②行使用+运算符进行字符串的拼接，其中产生了三个对象。代码第③~④行是使用+=赋值运算符，本质上也是+运算符进行拼接。

代码第⑤行采用concat方法进行拼接，该方法的完整定义如下：

```
public String concat(String str)
```

它的参数和返回值都是String，因此代码第⑤行可以连续调用该方法进行多个字符串的拼接。

代码第⑥和第⑦行是使用+运算符，将字符串与其他类型数据进行的拼接。代码第⑨行是与对象可以进行拼接，Java中所有对象都有一个toString()方法，该方法可以将对象转换为字符串，拼接过程会调用该对象的toString()方法，将该对象转换为字符串后再进行拼接。代码第⑧行的java.util.Date类是Java SE提供的日期类。

10.3.4 字符串查找

在给定的字符串中查找字符或字符串是比较常见的操作。在String类中提供了indexOf和lastIndexOf方法用于查找字符或字符串，返回值是查找的字符或字符串所在的位置，-1表示没有找到。这两个方法有多个重载版本：

- int indexOf(int ch)：从前往后搜索字符ch，返回第一次找到字符ch所在处的索引。
- int indexOf(int ch, int fromIndex)：从指定的索引开始从前往后搜索字符ch，返回第一次找到字符ch所在处的索引。
- int indexOf(String str)：从前往后搜索字符串str，返回第一次找到字符串所在处的索引。
- int indexOf(String str, int fromIndex)：从指定的索引开始从前往后搜索字符串str，返回第一次找到字符串所在处的索引。
- int lastIndexOf(int ch)：从后往前搜索字符ch，返回第一次找到字符ch所在处的索引。
- int lastIndexOf(int ch, int fromIndex)：从指定的索引开始从后往前搜索字符ch，返回第一次找到字符ch所在处的索引。
- int lastIndexOf(String str)：从后往前搜索字符串str，返回第一次找到字符串所在处的索引。
- int lastIndexOf(String str, int fromIndex)：从指定的索引开始从后往前搜索字符串str，返回第一次找到字符串所在处的索引。

提示　字符串本质上是字符数组，因此它也有索引，索引从零开始。String的charAt(int index)方法

可以返回索引index所在位置的字符。

字符串查找示例代码如下：

```
String sourceStr = "There is a string accessing example.";

//获得字符串长度
int len = sourceStr.length();
//获得索引位置16的字符
char ch = sourceStr.charAt(16);

//查找字符和子字符串
int firstChar1 = sourceStr.indexOf('r');
int lastChar1 = sourceStr.lastIndexOf('r');
int firstStr1 = sourceStr.indexOf("ing");
int lastStr1 = sourceStr.lastIndexOf("ing");
int firstChar2 = sourceStr.indexOf('e', 15);
int lastChar2 = sourceStr.lastIndexOf('e', 15);
int firstStr2 = sourceStr.indexOf("ing", 5);
int lastStr2 = sourceStr.lastIndexOf("ing", 5);

System.out.println("原始字符串:" + sourceStr);
System.out.println("字符串长度:" + len);
System.out.println("索引16的字符:" + ch);
System.out.println("从前往后搜索r字符，第一次找到它所在索引:" + firstChar1);
System.out.println("从后往前搜索r字符，第一次找到它所在索引:" + lastChar1);
System.out.println("从前往后搜索ing字符串，第一次找到它所在索引:" + firstStr1);
System.out.println("从后往前搜索ing字符串，第一次找到它所在索引:" + lastStr1);
System.out.println("从索引为15位置开始，从前往后搜索e字符，第一次找到它所在索引:" + firstChar2);
System.out.println("从索引为15位置开始，从后往前搜索e字符，第一次找到它所在索引:" + lastChar2);
System.out.println("从索引为5位置开始，从前往后搜索ing字符串，第一次找到它所在索引:" + firstStr2);
System.out.println("从索引为5位置开始，从后往前搜索ing字符串，第一次找到它所在索引:" + lastStr2);
```

输出结果：

```
原始字符串:There is a string accessing example.
字符串长度:36
索引16的字符:g
从前往后搜索r字符，第一次找到它所在索引:3
从后往前搜索r字符，第一次找到它所在索引:13
从前往后搜索ing字符串，第一次找到它所在索引:14
从后往前搜索ing字符串，第一次找到它所在索引:24
从索引为15位置开始，从前往后搜索e字符，第一次找到它所在索引:21
从索引为15位置开始，从后往前搜索e字符，第一次找到它所在索引:4
从索引为5位置开始，从前往后搜索ing字符串，第一次找到它所在索引:14
从索引为5位置开始，从后往前搜索ing字符串，第一次找到它所在索引:-1
```

sourceStr字符串索引如图10-6所示。上述字符串查找方法比较类似，这里重点解释一下sourceStr.indexOf("ing", 5)和sourceStr.lastIndexOf("ing", 5)表达式。从图10-6可见ing字符串出现过两次，索引分别是14和24。sourceStr.indexOf("ing", 5)表达式从索引为5的字符（" "）开始从前往后搜索，结果是找到第一个ing（索引为14），返回值为14。sourceStr.lastIndexOf("ing", 5)表达式从索引为5的字符（" "）开始从后往前搜索，没有找到，返回值为-1。

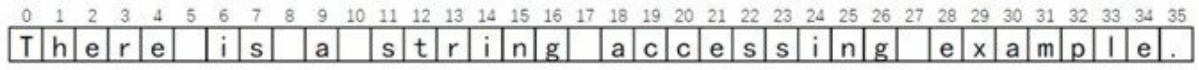


图**10-6　sourceStr**字符串索引

10.3.5 字符串比较

字符串比较是常见的操作，包括比较相等、比较大小、比较前缀和后缀等。

01. 比较相等

String提供的比较字符串相等的方法：

- boolean equals(Object anObject)：比较两个字符串中内容是否相等。
- boolean equalsIgnoreCase(String anotherString)：类似equals方法，只是忽略大小写。

02. 比较大小

有时不仅需要知道是否相等，还要知道大小，String提供的比较大小的方法：

- int compareTo(String anotherString)：按字典顺序比较两个字符串。如果参数字符串等于此字符串，则返回值 0；如果此字符串小于字符串参数，则返回一个小于 0 的值；如果此字符串大于字符串参数，则返回一个大于 0 的值。
- int compareToIgnoreCase(String str)：类似compareTo，只是忽略大小写。

03. 比较前缀和后缀

- boolean endsWith(String suffix)：测试此字符串是否以指定的后缀结束。
- boolean startsWith(String prefix)：测试此字符串是否以指定的前缀开始。

字符串比较示例代码如下：

```
String s1 = new String("Hello");
String s2 = new String("Hello");
// 比较字符串是否是相同的引用
System.out.println("s1 == s2 : " + (s1 == s2));
// 比较字符串内容是否相等
System.out.println("s1.equals(s2) : " + (s1.equals(s2)));

String s3 = "HELlo";
// 忽略大小写比较字符串内容是否相等
System.out.println("s1.equalsIgnoreCase(s3) : " + (s1.equalsIgnoreCase(s3)));

// 比较大小
String s4 = "java";
String s5 = "Swift";
// 比较字符串大小 s4 > s5
System.out.println("s4.compareTo(s5) : " + (s4.compareTo(s5)));                          ①
// 忽略大小写比较字符串大小 s4 < s5
System.out.println("s4.compareToIgnoreCase(s5) : " + (s4.compareToIgnoreCase(s5)));    ②

// 判断文件夹中文件名
String[] docFolder = { "java.docx", " JavaBean.docx", "Objecitve-C.xlsx", "Swift.docx " };
int wordDocCount = 0;
// 查找文件夹中Word文档个数
for (String doc : docFolder) {
    // 去的前后空格
    doc = doc.trim();                          ③
    // 比较后缀是否有.docx字符串
    if (doc.endsWith(".docx")) {
        wordDocCount++;
    }
}
```

```
System.out.println("文件夹中Word文档个数是：  " + wordDocCount);

int javaDocCount = 0;
// 查找文件夹中Java相关文档个数
for (String doc : docFolder) {
    // 去的前后空格
    doc = doc.trim();
    // 全部字符转成小写
    doc = doc.toLowerCase();                    ④
    // 比较前缀是否有java字符串
    if (doc.startsWith("java")) {
        javaDocCount++;
    }
}
System.out.println("文件夹中Java相关文档个数是：" + javaDocCount);
```

输出结果：

```
s1 == s2 : false
s1.equals(s2) : true
s1.equalsIgnoreCase(s3) : true
s4.compareTo(s5) : 23
s4.compareToIgnoreCase(s5) : -9
文件夹中Word文档个数是：  3
文件夹中Java相关文档个数是：2
```

上述代码第①行的compareTo方法按字典顺序比较两个字符串，s4.compareTo(s5)表达式返回结果大于0，说明s4大于s5，字符在字典中顺序事实上就它的Unicode编码，先比较两个字符串的第一个字符j和S，j的Unicode编码是106，S的Unicode编码是83，所以可以得出结论s4 > s5。代码第②行是忽略大小写时，要么全部当成小写字母进行比较，要么当前成全部大写字母进行比较，无论哪种比较结果都是一样的s4 < s5。

代码第③行trim()方法可以去除字符串前后空白。代码第④行toLowerCase()方法可以将此字符串全部转化为小写字符串，类似的方法还有toLowerCase()方法，可将字符串全部转化为小写字符串。

10.3.6 字符串截取

Java中字符串String截取方法主要的方法如下：

- String substring(int beginIndex)：从指定索引beginIndex开始截取一直到字符串结束的子字符串。
- String substring(int beginIndex, int endIndex)：从指定索引beginIndex开始截取直到索引endIndex - 1处的字符，注意包括索引为beginIndex处的字符，但不包括索引为endIndex处的字符。

字符串截取方法示例代码如下：

```
String sourceStr = "There is a string accessing example.";
// 截取example.子字符串
String subStr1 = sourceStr.substring(28);           ①
// 截取string子字符串
String subStr2 = sourceStr.substring(11, 17);       ②
System.out.printf("subStr1 = %s%n", subStr1);
System.out.printf("subStr2 = %s%n",subStr2);

// 使用split方法分割字符串
System.out.println("-----使用split方法-----");
String[] array = sourceStr.split(" ");              ③
for (String str : array) {
```

```
    System.out.println(str);
}
```

输出结果：

```
subStr1 = example.
subStr2 = string
-----使用split方法-----
There
is
a
string
accessing
example.
```

上述sourceStr字符串索引参考图10-6所示。代码第①行是截取example.子字符串，从图10-6可见e字符索引是28, 从索引28字符截取直到sourceStr结尾。代码第②行是截取string子字符串，从图10-6可见，s字符索引是11，g字符索引是16，endIndex参数应该17。

另外，String还提供了字符串分割方法，见代码第③行split(" ")方法，参数是分割字符串，返回值String[]。

10.4 可变字符串

可变字符串在追加、删除、修改、插入和拼接等操作不会产生新的对象。

10.4.1 StringBuffer和StringBuilder

Java提供了两个可变字符串类StringBuffer和StringBuilder，中文翻译为“字符串缓冲区”。

StringBuffer是线程安全的，它的方法是支持线程同步[1]，线程同步会操作串行顺序执行，在单线程环境下会影响效率。StringBuilder是StringBuffer单线程版本，Java 5之后发布的，它不是线程安全的，但它的执行效率很高。

> [1]线程同步是一个多线程概念，就是当多个线程访问一个方法时，只能由一个优先级别高的线程先访问，在访问期间会锁定该方法，其他线程只能等到它访问完成释放锁，才能访问。有关多线程问题将在后面章节详细介绍。

StringBuffer和StringBuilder具有完全相同的API，即构造方法和普通方法等内容一样。StringBuilder的中构造方法有4个：

- StringBuilder()：创建字符串内容是空的StringBuilder对象，初始容量默认为16个字符。
- StringBuilder(CharSequence seq)：指定CharSequence字符串创建StringBuilder对象。CharSequence接口类型，它的实现类有：String、StringBuffer和StringBuilder等，所以参数seq可以是String、StringBuffer和StringBuilder等类型。
- StringBuilder(int capacity)：创建字符串内容是空的StringBuilder对象，初始容量由参数capacity指定的。
- StringBuilder(String str)：指定String字符串创建StringBuilder对象。

上述构造方法同样适合于StringBuffer类，这里不再赘述。

> 提示　字符串长度和字符串缓冲区容量区别。字符串长度是指在字符串缓冲区中目前所包含字符串长度，通过length()获得；字符串缓冲区容量是缓冲区中所能容纳的最大字符数，通过capacity()获得。当所容纳的字符超过这个长度时，字符串缓冲区自动扩充容量，但这是以牺牲性能为代价的扩容。

字符串长度和字符串缓冲区容量示例代码如下：

```
// 字符串长度length和字符串缓冲区容量capacity
StringBuilder sbuilder1 = new StringBuilder();
System.out.println("包含的字符串长度：" + sbuilder1.length());
System.out.println("字符串缓冲区容量：" + sbuilder1.capacity());

StringBuilder sbuilder2 = new StringBuilder("Hello");
System.out.println("包含的字符串长度：" + sbuilder2.length());
System.out.println("字符串缓冲区容量：" + sbuilder2.capacity());

// 字符串缓冲区初始容量是16，超过之后会扩容
StringBuilder sbuilder3 = new StringBuilder();
for (int i = 0; i < 17; i++) {
    sbuilder3.append(8);
}
System.out.println("包含的字符串长度：" + sbuilder3.length());
System.out.println("字符串缓冲区容量：" + sbuilder3.capacity());
```

输出结果：

```
包含的字符串长度：0
字符串缓冲区容量：16
包含的字符串长度：5
字符串缓冲区容量：21
包含的字符串长度：17
字符串缓冲区容量：34
```

10.4.2 字符串追加

StringBuilder在提供了很多修改字符串缓冲区的方法，追加、插入、删除和替换等，这一节先介绍字符串追加方法。字符串追加方法是append，append有很多重载方法，可以追加任何类型数据，它的返回值还是StringBuilder。StringBuilder的追加法与StringBuffer完全一样，这里不再赘述。

字符串追加示例代码如下：

```
//添加字符串、字符
StringBuilder sbuilder1 = new StringBuilder("Hello");  ①
sbuilder1.append(" ").append("World");                 ②
sbuilder1.append('.');                                 ③
System.out.println(sbuilder1);

StringBuilder sbuilder2 = new StringBuilder();
Object obj = null;
//添加布尔值、转义符和空对象
sbuilder2.append(false).append('\t').append(obj);      ④
System.out.println(sbuilder2);

//添加数值
StringBuilder sbuilder3 = new StringBuilder();
for (int i = 0; i < 10; i++) {
    sbuilder3.append(i);
}
System.out.println(sbuilder3);
```

运行结果：

```
Hello World.
false  null
0123456789
```

上述代码第①行是创建一个包含Hello字符串StringBuilder对象。代码第②行是两次连续调用append方法，由于所有的append方法都返回StringBuilder对象，所有可以连续调用该方法，这种写法比较简洁。如果连续调用append方法不行喜欢，可以append方法占一行，见代码第③行。

代码第④行连续追加了布尔值、转义符和空对象，需要注意的是布尔值false转换为false字符串，空对象null也转换为"null"字符串。

10.4.3 字符串插入、删除和替换

StringBuilder中实现插入、删除和替换等操作的常用方法说明如下：

- StringBuilder insert(int offset, String str)：在字符串缓冲区中索引为offset的字符位置之前插入str，insert有很多重载方法，可以插入任何类型数据。

- StringBuffer delete(int start, int end)：在字符串缓冲区中删除子字符串，要删除的子字符串从指定索引start开始直到索引end - 1处的字符。start和end两个参数与substring(int beginIndex, int endIndex)方法中的两个参数含义一样。

- StringBuffer replace(int start, int end, String str)字符串缓冲区中用str替换子字符串，子字符串从指定索引start开始直到索引end - 1处的字符。start和end同delete(int start, int end)方法。

以上介绍的方法虽然是StringBuilder方法，但StringBuffer也完全一样，这里不再赘述。

示例代码如下：

```
// 原始不可变字符串
String str1 = "Java C";
// 从不可变的字符创建可变字符串对象
StringBuilder mstr = new StringBuilder(str1);

// 插入字符串
mstr.insert(4, " C++");                          ①
System.out.println(mstr);

// 具有追加效果的插入字符串
mstr.insert(mstr.length(), " Objective-C");  ②
System.out.println(mstr);

// 追加字符串
mstr.append(" and Swift");
System.out.println(mstr);

// 删除字符串
mstr.delete(11, 23);                             ③
System.out.println(mstr);
```

输出结果：

```
Java C++ C
Java C++ C Objective-C
Java C++ C Objective-C and Swift
Java C++ C and Swift
```

上述代码第①行mstr.insert(4, " C++")是在索引4，插入字符串，原始字符串索引如图10-7所示，索引4位置是一个空格，在它之前插入字符串。代码第②行mstr.insert(mstr.length(), " Objective-C")是按照字符串的长的插入，也就是在尾部追加字符串。在执行代码第③行删除字符串之前的字符串如图10-8所示，mstr.delete(11, 23)语句是要删除"Objective-C "子字符串，第一个参数是子字符串开始索引11；第二个参数是23，结束字符的索引是22（end - 1），所以参数end是23。

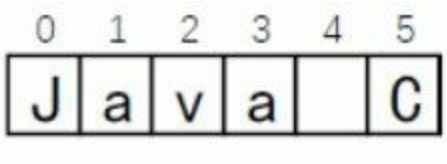


图**10-7**　原始字符串索引

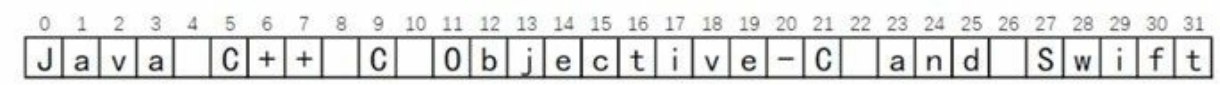


图**10-8**　删除之前字符串索引

本章小结

本章介绍了Java中的字符串，Java字符串类分为：可变字符串类（String）和不可变字符串类（StringBuilder和StringBuffer）。然后分别介绍了这些字符串类的用法。

第 **11** 章　面向对象基础

面向对象是Java最重要的特性。Java是彻底的、纯粹的面向对象语言，在Java中“一切都是对象”。本章将介绍面向对象基础知识。

11.1 面向对象概述

面向对象的编程思想：按照真实世界客观事物的自然规律进行分析，客观世界中存在什么样的实体，构建的软件系统就存在什么样的实体。

例如：在真实世界的学校里，会有学生和老师等实体，学生有学号、姓名、所在班级等属性（数据），学生还有学习、提问、吃饭和走路等操作。学生只是抽象的描述，这个抽象的描述称为“类”。在学校里活动是学生个体，即：张同学、李同学等，这些具体的个体称为“对象”，“对象”也称为“实例”。

在现实世界有类和对象，面向对象软件世界也会有，只不过它们会以某种计算机语言编写的程序代码形式存在，这就是面向对象编程（Object Oriented Programming，OOP）。作为面向对象的计算机语言——Java，具有定义类和创建对象等面向对象能力。

11.2 面向对象三个基本特性

面向对象思想有三个基本特性：封装性、继承性和多态性。

11.2.1 封装性

在现实世界中封装的例子到处都是。例如：一台计算机内部极其复杂，有主板、CPU、硬盘和内存，而一般用户不需要了解它的内部细节，不需要知道主板的型号、CPU主频、硬盘和内存的大小，于是计算机制造商将用机箱把计算机封装起来，对外提供了一些接口，如鼠标、键盘和显示器等，这样当用户使用计算机就变非常方便。

那么，面向对象的封装与真实世界的目的是一样的。封装能够使外部访问者不能随意存取对象的内部数据，隐藏了对象的内部细节，只保留有限的对外接口。外部访问者不用关心对象的内部细节，使得操作对象变得简单。

11.2.2 继承性

在现实世界中继承也是无处不在。例如：轮船与客轮之间的关系，客轮是一种特殊轮船，拥有轮船的全部特征和行为，即数据和操作。在面向对象中轮船是一般类，客轮是特殊类，特殊类拥有一般类的全部数据和操作，称为特殊类继承一般类。在Java语言中一般类称为“父类”，特殊类称为“子类”。

> 提示　在有些语言如C++支持多继承，多继承就是一个子类可有多个父类，例如，客轮是轮船也是交通工具，客轮的父类是轮船和交通工具。多继承会引起很多冲突问题，因此现在很多面向对象的语言都不支持多继承。Java语言是单继承的，即只能有一个父类，但Java可以实现多个接口，可以防止多继承所引起的冲突问题。

11.2.3 多态性

多态性是指在父类中成员变量和成员方法被子类继承之后，可以具有不同的状态或表现行为。有关多态性详细解释，请参考13.4节，这里不再赘述。

11.3 类

类是Java中的一种重要的引用数据类型，是组成Java程序的基本要素。它封装了一类对象的数据和操作。

11.3.1 类声明

Java语言中一个类的实现包括：类声明和类体。类声明语法格式如下。

```
[public][abstract|final] class className [extends superclassName] [implements interfaceNameList] {
    //类体
}
```

其中，class是声明类的关键字，className是自定义的类名；class前面的修饰符public、abstract、final用来声明类，它们可以省略，它们的具体用法后面章节会详细介绍；superclassName为父类名，可以省略，如果省略则该类继承Object类，Object类所有类的根类，所有类都直接或间接继承Object；interfaceNameList是该类实现的接口列表，可以省略，接口列表中的多个接口之间用逗号分隔。

提示　本书语法表示符号约定，在语法说明中，括号（[]）部分表示可以省略；竖线（|）表示“或关系”，例如abstract|final，说明可以使用abstract或final关键字，两个关键字不能同时出现。

声明动物（Animal）类代码如下：

```
    // Animal.java
    public class Animal extends Object {

        //类体
    }
```

上述代码声明了动物（Animal）类，它继承了Object类。继承Object类extends Object代码可以省略。

类体是类的主体，包括数据和操作，即成员变量和成员方法。下面就来展开介绍一下。

11.3.2 成员变量

声明类体中成员变量语法格式如下：

```
    class className {
        [public | protected | private ] [static] [final] type variableName;    //成员变量
    }
```

其中type是成员变量数据类型，variableName是成员变量名。type前的关键字都是成员变量修饰符，它们说明如下：

01. public、protected和private修饰符用于封装成员变量。

02. static修饰符用于声明静态变量，所以静态变量也称为“类变量”。

03. final修饰符用于声明变量，该变量不能被修改。

下面看一个声明成员变量示例：

```
// Animal.java
public class Animal extends Object {

    //动物年龄
    int age = 1;
    //动物性别
    public boolean sex = false;
    //动物体重
    private double weight = 0.0;

}
```

上述代码中没有展示静态变量声明，有关静态变量稍后会详细介绍。

11.3.3 成员方法

声明类体中成员方法语法格式如下：

```
class className {

        [public | protected | private ] [static] [final | abstract] [native] [synchronized]
                type methodName([paramList]) [throws exceptionList] {
                    //方法体
            }
}
```

其中type是方法返回值数据类型，methodName是方法名。type前的关键字都是方法修饰符，它们说明如下：

01. public、protected和private修饰符用于封装方法。
02. static修饰符用于声明静态方法，所以静态方法也称为“类方法”。
03. final | abstract不能同时修饰方法，final修饰的方法不能在子类中被覆盖；abstract用来修饰抽象方法，抽象方法必须在子类中被实现。
04. native修饰的方法，称为“本地方法”，本地方法调用平台本地代码（如：C或C++编写的代码），不能实现跨平台。
05. synchronized修饰的方法是同步的，当多线程方式同步方法时，只能串行地执行，保证是线程安全的。

方法声明中还有([paramList])部分，它是方法的参数列表。throws exceptionList是声明抛出异常列表。

下面看一个声明方法示例：

```
public class Animal {// extends Object {

    //动物年龄
    int age = 1;
    //动物性别
    public boolean sex = false;
    //动物体重
    private double weight = 0.0;

    public void eat() {                ①
        // 方法体
        return;                        ②
```

```
        }

        int run() {                          ③
            // 方法体
            return 10;                       ④
        }

        protected int getMaxNumber(int number1, int number2) {        ⑤
            // 方法体
            if (number1 > number2) {
                return number1;        ⑥
            }
            return number2;
        }
    }
```

上述代码第①、③、⑤行声明了三个方法。方法在执行完毕后把结果返还给它的调用者，方法体包含“return 返回结果值;”语句，见代码第④行的“return 10;”，“返回结果值”数据类型与方法的返回值类型要匹配。如果方法返回值类型为void时，方法体包含“return;”语句，见代码第②行，如果“return;”语句是最后一行则可以省略。

提示　通常return语句通常用在一个方法体的最后，否则会产生编译错误，除非用在if-else语句中，见代码第⑥行。

11.4 包

在程序代码中给类起一个名字是非常重要的，但是有时候会出现非常尴尬的事情，名字会发生冲突，例如：项目中自定义了一个日期类，我为它取名为Date，但是会发现Java SE核心库中还有两个Date，它们分别位于java.util包和java.sql包中。

11.4.1 包作用

在Java中为了防止类、接口、枚举和注释等命名冲突引用了包（package）概念，包本质上命名空间（namespace）[1]。在包中可以定义一组相关的类型（类、接口、枚举和注释），并为它们提供访问保护和命名空间管理。

> [1]命名空间，也称名字空间、名称空间等，它表示着一个标识符（identifier）的可见范围。一个标识符可在多个命名空间中定义，它在不同命名空间中的含义是互不相干的。这样，在一个新的命名空间中可定义任何标识符，它们不会与任何已有的标识符发生冲突，因为已有的定义都处于其他命名空间中。 ——引自于 维基百科 https://zh.wikipedia.org/wiki/命名空间

在前面提到的Date类名称冲突问题，很好解决，将不同Date类放到不同的包中，我们自定义Date，可以放到自己定义的包com.a51work6中，这样就不会与java.util包和java.sql包中Date发生冲突问题了。

11.4.2 包定义

Java中使用package语句定义包，package语句应该放在源文件的第一行，在每个源文件中只能有一个包定义语句，并且package语句适用于所有类型（类、接口、枚举和注释）的文件。定义包语法格式如下：

```
package pkg1[.pkg2[.pkg3…]];
```

pkg1~ pkg3都是组成包名一部分，之间用点（.）连接，它们命名应该是合法的标识符，其次应该遵守Java包命名规范，即全部小写字母。

定义包示例代码如下：

```
// Date.java文件
package com.a51work6;

public class Date {

}
```

com.a51work6是自定义的包名，包名一般是公司域名的倒置。

> 提示　我们公司的域名是51work6.com，倒置后是com.51work6，其中51work6是非法标识符（不能用数字开头），所以com.51work6包名是非法的，于是将包名改为com.a51work6。

如果在源文件中没有定义包，那么类、接口、枚举和注释类型文件将会被放进一个无名的包中，也称为默认包。

定义好包后，包采用层次结构管理这些类型（类、接口、枚举和注释），如图11-1所示是在Eclipse包资源视图中查看包，可见有默认包和com.a51work6包。如果文件系统中查看这些包，会发现如图11-2所示的层次结构，源文件目录是根目录，也是默认包目录，可见其中有一个HelloWorld.java文件。com是文件夹，a51work6子文件夹，在a51work6中包含：Animal.java和Date.java两个文件。Java编译器把包对应于文件系统的目录管理，不仅是源文件，编译之后的字节码文件也采用文件系统的目录管理的。

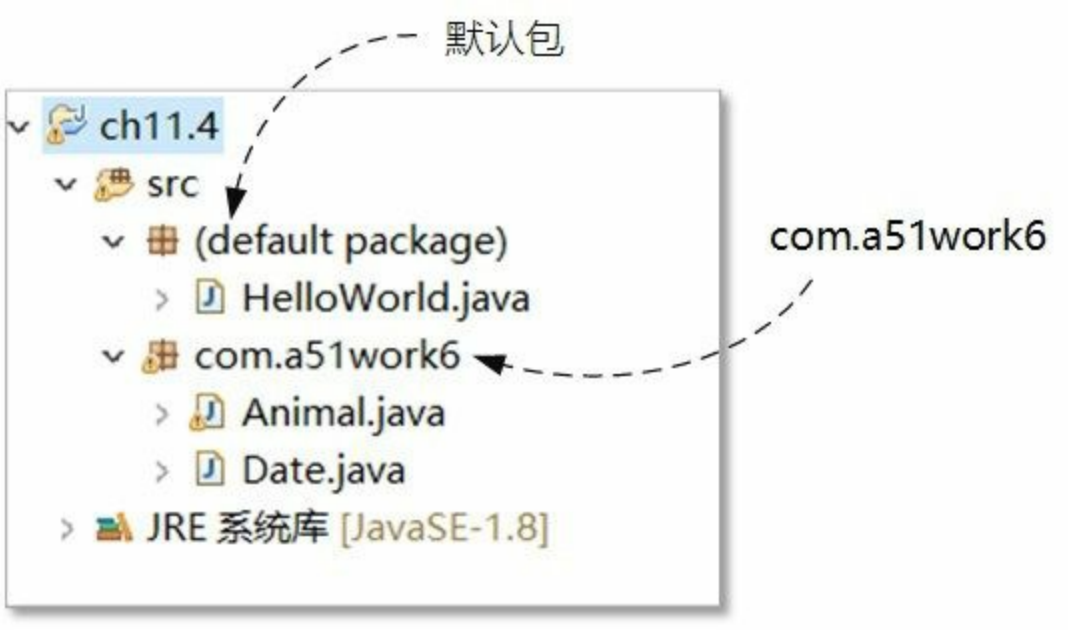


图**11-1**　**Eclipse**包资源视图中查看包

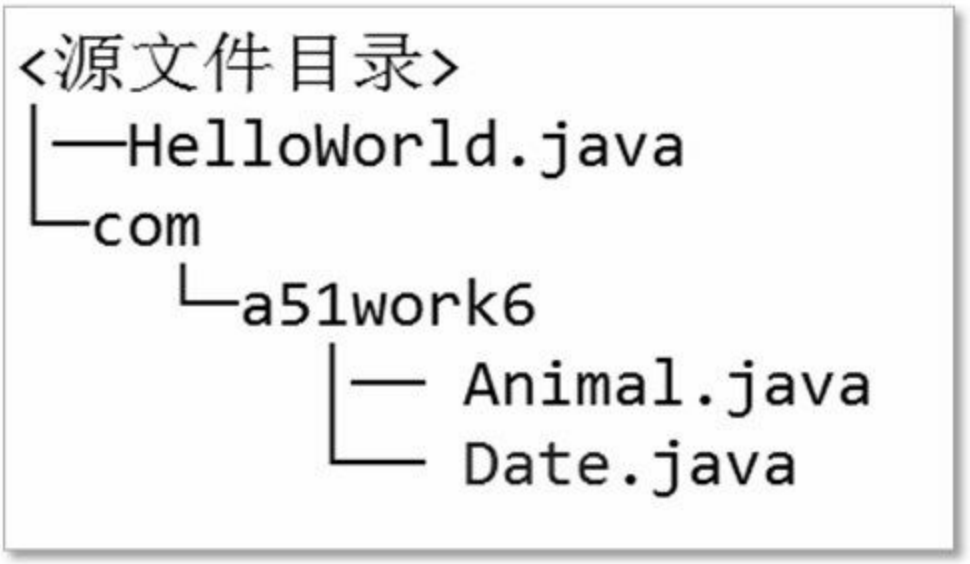


图**11-2**　文件系统目录与包

11.4.3　包引入

为了能够使用一个包中类型（类、接口、枚举和注释），需要在Java程序中明确引入该包。使用import语句实现引入包，import语句应位于package语句之后，所有类的定义之前，可以有0~n条import语句，其语法格式为：

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

“包名.类型名”形式只引入具体类型，“包名.*”采用通配符，表示引入这个包下所有的类型。但从编程

规范的角度提倡明确引入类型名，即“包名.类型名”形式可以提高程序的可读性。

如果需要在程序代码中使用com.a51work6包中Date类。示例代码如下：

```
// HelloWorld.java文件
import com.a51work6.Date;           ①

public class HelloWorld {

    public static void main(String[] args) {

        Date date = new Date();           ②
        System.out.println(date);
    }
}
```

上述代码第②行使用了Date类，需要引入Date所在的包，见代码第①行，import是关键字，代码第①行的import语句采用“包名.类型名”形式。

提示　如果在一个源文件中引入两个相同包名+类型名，见如下代码，代码第②行会发生编译错误。为避免这个编译错误，可以在没有引入包的类型名前加上包名，详见如下代码第②行中的java.util.Date。

```
// HelloWorld.java文件
import com.a51work6.Date;
//import java.util.Date;                                    ①

public class HelloWorld {

    public static void main(String[] args) {

        Date date = new Date();
        System.out.println(date);
        java.util.Date now = new java.util.Date();        ②
        System.out.println(now);

    }
}
```

注意　当前源文件与要使用的类型（类、接口、枚举和注释）在同一个包中，可以不用引入包。

11.4.4 常用包

Java SE提供一些常用包，其中包含了Java开发中常用的基础类。这些包有：java.lang、java.io、java.net、java.util、java.text、java.awt和javax.swing。

01. java.lang包

    java.lang包含中包含了Java语言的核心类，如Object、Class、String、包装类和Math等，还有包装类Boolean、Character、Integer、Long、Float和Double。使用java.lang包中的类型，不需要显示使用import语句引入，它是由解释器自动引入。

02. java.io包

    java.io包含中提供多种输入/输出流类，如InputStream、OutputStream、Reader和Writer。还有文件管理相关类和接口，如File和FileDescriptor类以及FileFilter接口。

03. java.net包

    java.net包含进行网络相关的操作的类，如URL、Socket和ServerSocket等。

04. java.util包

    java.util包含一些实用工具类和接口，如集合、日期和日历相关类和接口。

05. java.text包

    java.text包中提供文本处理、日期式化和数字格式化等相关类和接口。

06. java.awt和javax.swing包

    java.awt和javax.swing包提供了Java图形用户界面开发所需要的各种类和接口。java.awt提供是一些基础类和接口，javax.swing提供了一些高级组件。

11.5 方法重载（Overload）

在第10章介绍字符串时就已经用到过方法重载，这一节详细介绍一下重载。出于使用方便等原因，在设计一个类时将具有相似功能的方法起相同的名字。例如String字符串查找方法indexOf有很多不同版本，如图11-3所示：

int	indexOf(int ch) 返回指定字符在此字符串中第一次出现处的索引。
int	indexOf(int ch, int fromIndex) 返回在此字符串中第一次出现指定字符处的索引，从指定的索引开始搜索。
int	indexOf(String str) 返回指定子字符串在此字符串中第一次出现处的索引。
int	indexOf(String str, int fromIndex) 返回指定子字符串在此字符串中第一次出现处的索引，从指定的索引开始。

图**11-3 indexOf**方法重载

这些相同名字的方法之所以能够在一个类中同时存在，是因为它们的方法参数列表，调用时根据参数列表调用相应重载方法。

> 提示 方法重载中参数列表不同的含义是：参数的个数不同或者是参数类型不同。另外，返回类型不能用来区分方法重载。

方法重载示例MethodOverloading.java代码如下：

```
// MethodOverloading.java文件
package com.a51work6;

class MethodOverloading {

    void receive(int i) {                              ①
        System.out.println("接收一个int参数");
        System.out.println("i = " + i);
    }

    void receive(int x, int y) {                       ②
        System.out.println("接收两个int参数");
        System.out.printf("x = %d, y = %d \r", x, y);
    }

    int receive(double x, double y) {                  ③
        System.out.println("接收两个double参数");
        System.out.printf("x = %f, y = %f \r", x, y);
        return 0;
    }
}

// HelloWorld.java文件调用MethodOverloading
package com.a51work6;

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

        MethodOverloading mo = new MethodOverloading();
```

```
        //调用void receive(int i)
        mo.receive(1);                          ④

        //调用void receive(int x, int y)
        mo.receive(2, 3);                       ⑤

        //调用void receive(double x, double y)
        mo.receive(2.0, 3.3);                   ⑥
    }
}
```

MethodOverloading类中有三个相同名字的receive方法，在HelloWorld的main方法中调用MethodOverloading的receive方法。运行结果如下：

```
接收一个int参数
i = 1
接收两个int参数
x = 2, y = 3
接收两个double参数
x = 2.000000, y = 3.300000
```

调用哪一个receive方法是根据参数列表决定的。如果参数类型不一致，编译器会进行自动类型转换寻找适合版本的方法，如果没有适合方法，则会发生编译错误。假设删除代码第②行的void receive(int x, int y)方法，代码第⑤行的mo.receive(2, 3)语句调用的是void receive(double x, double y)方法，其中int类型参数（2和3）自动会转换为double类型（2.0和3.0）再调用。

11.6 封装性与访问控制

Java面向对象的封装性是通过对成员变量和方法进行访问控制实现的，访问控制分为4个等级：私有、默认、保护和公有，具体规则如表11-1所示。

表 **11-1** **Java**类成员的访问控制

可否直接访问 / 控制等级	同一个类	同一个包	不同包的子类	不同包非子类
私有	Yes			
默认	Yes	Yes		
保护	Yes	Yes	Yes	
公有	Yes	Yes	Yes	Yes

下面详细解释一下这4种访问级别。

11.6.1 私有级别

私有级别的关键字是private，私有级别的成员变量和方法只能在其所在类的内部自由使用，在其他的类中则不允许直接访问。私有级别限制性最高。私有级别示例代码如下：

```
// PrivateClass.java文件
package com.a51work6;

public class PrivateClass {            ①

    private int x;                     ②

    public PrivateClass() {            ③
        x = 100;
    }

    private void printX() {            ④
        System.out.println("Value Of x is" + x);
    }

}

// HelloWorld.java文件调用PrivateClass
package com.a51work6;

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

        PrivateClass p;
        p = new PrivateClass();

        //编译错误，PrivateClass中的方法 printX()不可见
        p.printX();                    ⑤
```

```
    }
}
```

上述代码第①行声明PrivateClass类，其中的代码第②行是声明私有实例变量x，代码第③行是声明公有的构造方法，构造方法将在第12章详细介绍。代码第④行声明私有实例方法。

HelloWorld类中代码第⑤行会有编译错误，因为PrivateClass中printX()的方法是私有方法。

11.6.2 默认级别

默认级别没有关键字，也就是没有访问修饰符，默认级别的成员变量和方法，可以在其所在类内部和同一个包的其他类中被直接访问，但在不同包的类中则不允许直接访问。

默认级别示例代码如下：

```
// DefaultClass.java文件
package com.a51work6;

public class DefaultClass {

    int x;                          ①

    public DefaultClass() {
        x = 100;
    }

    void printX() {                 ②
        System.out.println("Value Of x is" + x);
    }

}
```

上述代码第①行的x变量前没有访问限制修饰符，代码第②行的方法也是没有访问限制修饰符。它们的访问级别都有默认访问级别。

在相同包（com.a51work6）中调用DefaultClass类代码如下：

```
// com.a51work6包中HelloWorld.java文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        DefaultClass p;
        p = new DefaultClass();
        p.printX();
    }
}
```

默认访问级别可以在同一包中访问，上述代码可以编译通过。

在不同的包中调用DefaultClass类代码如下：

```
// 默认包中HelloWorld.java文件
import com.a51work6.DefaultClass;
```

```
public class HelloWorld {

    public static void main(String[] args) {

        DefaultClass p;
        p = new DefaultClass();
        // 编译错误，DefaultClass中的方法 printX()不可见
        p.printX();
    }
}
```

该HelloWorld.java文件与DefaultClass类不在同一个包中，printX()是默认访问级别，所以p.printX()方法无法编译通过。

11.6.3 公有级别

公有级别的关键字是public，公有级别的成员变量和方法可以在任何场合被直接访问，是最宽松的一种访问控制等级。

公有级别示例代码如下：

```
// PublicClass.java文件
package com.a51work6;

public class PublicClass {

    public int x;                ①

    public PublicClass() {
        x = 100;
    }

    public void printX() {       ②
        System.out.println("Value Of x is" + x);
    }

}
```

上述代码第①行的x变量是公有级别，代码第②行的方法也是公有级别。调用PublicClass类代码如下：

```
// 默认包中HelloWorld.java文件
import com.a51work6.PublicClass;

public class HelloWorld {

    public static void main(String[] args) {

        PublicClass p;
        p = new PublicClass();
        p.printX();
    }
}
```

该HelloWorld.java文件与PublicClass类不在同一个包中，可以直接访问公有的printX()方法。

11.6.4 保护级别

保护级别的关键字是protected，保护级别在同一包中完全与默认访问级别一样，但是不同包中子类能

够继承父类中的protected变量和方法，这就是所谓的保护级别，“保护”就是保护某个类的子类都能继承该类的变量和方法。

保护级别示例代码如下：

```
// ProtectedClass.java文件
package com.a51work6;

public class ProtectedClass {

    protected int x;                    ①

    public ProtectedClass() {
        x = 100;
    }

    protected void printX() {           ②
        System.out.println("Value Of x is " + x);
    }

}
```

上述代码第①行的x变量是保护级别，代码第②行的方法也是保护级别。

在相同包（com.a51work6）中调用ProtectedClass类代码如下：

```
// 默认包中HelloWorld.java文件
package com.a51work6;

import com.a51work6.ProtectedClass;

public class HelloWorld {

    public static void main(String[] args) {

        ProtectedClass p;
        p = new ProtectedClass();
        // 同一包中可以直接访问ProtectedClass中的方法 printX()
        p.printX();

    }
}
```

同一包中保护访问级别与默认访问级别一样，可以直接访问ProtectedClass的printX()方法，上述代码可以编译通过。

在不同的包中调用ProtectedClass类代码如下：

```
// 默认包中HelloWorld.java文件
import com.a51work6.ProtectedClass;

public class HelloWorld {

    public static void main(String[] args) {

        ProtectedClass p;
        p = new ProtectedClass();
        // 编译错误，不同包中不能直接访问保护方法printX()
        p.printX();
```

```
    }
}
```

该HelloWorld.java文件与ProtectedClass类不在同一个包中，不同包中不能直接访问保护方法printX()，所以p.printX()方法无法编译通过。

在不同的包中继承ProtectedClass类代码如下：

```
// 默认包中SubClass.java文件
import com.a51work6.ProtectedClass;

public class SubClass extends ProtectedClass {

    void display() {
        //printX()方法是从父类继承过来
        printX();                          ①
        //x实例变量是从父类继承过来
        System.out.println(x);             ②
    }
}
```

不同包中SubClass从ProtectedClass类继承了printX()方法和x实例变量。代码第①行是调用从父类继承下来的方法，代码第②行是调用从父类继承下来的实例变量。

提示　访问成员有两种方式：一种是调用，即通过类或对象调用它的成员，如p.printX()语句；另一种是继承，即子类继承父类的成员变量和方法。

- 公有访问级别任何情况下两种方式都可以；
- 默认访问级别在同一包中两种访问方式都可以，不能在包之外访问；
- 保护访问级别在同一包中与默认访问级别一样，两种访问方式都可以。但是在不同包之外只能继承访问；
- 私有访问级别只能在本类中通过调用方法访问，不能继承访问。

提示　访问类成员时，在能满足使用的前提下，应尽量限制类中成员的可见性，访问级别顺序是：私有级别→默认级别→保护级别→公有级别。

11.7 静态变量和静态方法

有一个Account（银行账户）类，假设它有三个成员变量：amount（账户金额）、interestRate（利率）和owner（账户名）。在这三个成员变量中，amount和owner会因人而异，对于不同的账户这些内容是不同的，而所有账户的interestRate都是相同的。

amount和owner成员变量与账户个体有关，称为“实例变量”，interestRate成员变量与个体无关，或者说是所有账户个体共享的，这种变量称为“静态变量”或“类变量”。

静态变量和静态方法示例代码如下：

```
// Account.java文件
package com.a51work6;

public class Account {

    // 实例变量账户金额
    double amount = 0.0;                    ①
    // 实例变量账户名
    String owner;                           ②

    // 静态变量利率
    static double interestRate = 0.0668;    ③

    // 静态方法
    public static double interestBy(double amt) {        ④
        //静态方法可以访问静态变量和其他静态方法
        return interestRate * amt;                       ⑤
    }

    // 实例方法
    public String messageWith(double amt) {              ⑥
        //实例方法可以访问实例变量、实例方法、静态变量和静态方法
        double interest = Account.interestBy(amt);       ⑦
        StringBuilder sb = new StringBuilder();
        // 拼接字符串
        sb.append(owner).append("的利息是").append(interest);
        // 返回字符串
        return sb.toString();
    }
}
```

static修饰的成员变量是静态变量，见代码第③行。staitc修饰的方法是静态方法，见代码第④行。相反，没有static修饰的成员变量是实例变量，见代码第①行和第②行；没有staitc修饰的方法是实例方法，见代码第⑥行。

注意　静态方法可以访问静态变量和其他静态方法，例如访问代码第⑤行中的interestRate静态变量。实例方法可以访问实例变量、其他实例方法、静态变量和静态方法，例如访问代码第⑦行interestBy静态方法。

调用Account代码如下：

```
// HelloWorld.java文件
package com.a51work6;

public class HelloWorld {

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

```
        // 访问静态变量
        System.out.println(Account.interestRate);           ①
        // 访问静态方法
        System.out.println(Account.interestBy(1000));       ②

        Account myAccount = new Account();
        // 访问实例变量
        myAccount.amount = 1000000;                         ③
        myAccount.owner = "Tony";                           ④
        // 访问实例方法
        System.out.println(myAccount.messageWith(1000));    ⑤

        // 通过实例访问静态变量
        System.out.println(myAccount.interestRate);         ⑥
    }
}
```

调用静态变量或静态方法时，可以通过类名或实例名调用，代码第①行Account.interestRate通过类名调用静态变量，代码第②行Account.interestBy(1000)是通过类名调用静态方法。代码第⑥行是通过实例调用静态变量。

11.8 静态代码块

前面介绍的静态变量interestRate，可以在声明同时初始化，如下代码所示。

```
public class Account {

    // 静态变量利率
    static double interestRate = 0.0668;
    ...
}
```

如果初始化静态变量不是简单常量，需要进行计算才能初始化，可以使用静态（static）代码块，静态代码块在类第一次加载时执行，并只执行一次。示例代码如下：

```
// Account.java文件
package com.a51work6;

public class Account {

    // 实例变量账户金额
    double amount = 0.0;
    // 实例变量账户名
    String owner;

    // 静态变量利率
    static double interestRate;

    // 静态方法
    public static double interestBy(double amt) {
        // 静态方法可以访问静态变量和其他静态方法
        return interestRate * amt;
    }

    // 静态代码块
    static {                                ①
        System.out.println("静态代码块被调用...");
        // 初始化静态变量
        interestRate = 0.0668;          ②
    }
}
```

上述代码第①行是静态代码块，在静态代码块中可以初始化静态变量，见代码第②行，也可以调用静态方法。

调用Account代码如下：

```
// HelloWorld.java文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        Account myAccount = new Account();              ①
        // 访问静态变量
        System.out.println(Account.interestRate);       ②
        // 访问静态方法
        System.out.println(Account.interestBy(1000));
```

```
    }
}
```

Account静态代码块是在第一次加载Account类时调用。上述代码第①行是第一次使用Account类，此时会调用静态代码块。

本章小结

本章主要介绍了面向对象基础知识。首先介绍了面向对象一些基本概念，面向对象三个基本特性。然后介绍了类、包、方法重载和访问控制。最后介绍了静态变量、静态方法和静态代码块。

第 12 章　对象

类实例化可生成对象，实例方法就是对象方法，实例变量就是对象属性。一个对象的生命周期包括三个阶段：创建、使用和销毁。前面章节已经多次 用到了对象，本章详细介绍一下对象的创建和销毁等相关知识。

12.1 创建对象

创建对象包括两个步骤：声明和实例化。

01. 声明

    声明对象与声明普通变量没有区别，语法格式如下：

```
type objectName;
```

    其中type是引用类型，即类、接口和数组。示例代码如下：

```
String name;
```

    该语句声明了字符串类型对象name。可以声明并不为对象分配内存空间，而只是分配一个引用。

02. 实例化

    实例化过程分为两个阶段：为对象分配内存空间和初始化对象，首先使用new运算符为对象分配内存空间，然后再调用构造方法初始化对象。示例代码如下：

```
String name;
name = new String("Hello World");
```

    代码中String("Hello World")表达式就是调用String的构造方法。初始化完成之后如图12-1所示。

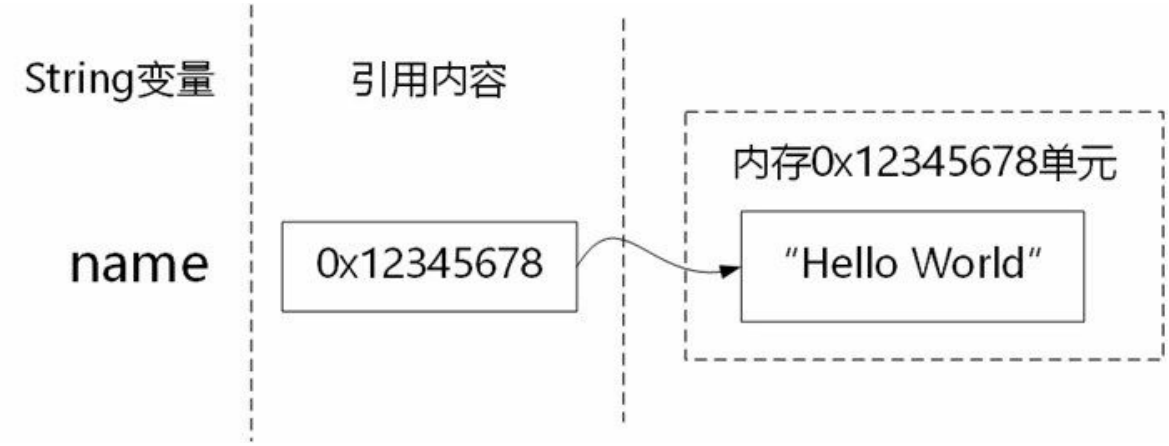


图**12-1** 对象实例化

12.2 空对象

一个引用变量没有通过new分配内存空间，这个对象就是空对象，Java使用关键字null表示空对象。示例代码如下：

```
String name = null;
name = "Hello World";
```

引用变量默认值是null。当试图调用一个空对象的实例变量或实例方法时，会抛出空指针异常NullPointerException，如下代码所示：

```
String name = null;
//输出null字符串
System.out.println(name);
//调用length()方法
int len = name.length();          ①
```

但是代码运行到第①行时，系统会抛出异常。这是因为调用length()方法时，name是空对象。程序员应该避免调用空对象的成员变量和方法，代码如下：

```
//判断对象是否为null
if (name != null) {
    int len = name.length();
}
```

提示　产生空对象有两种可能性：第一是程序员自己忘记了实例化，第二是空对象是别人传递过来的。程序员必须防止第一种情况的发生，应该仔细检查自己的代码，为自己创建的所有对象进行实例化并初始化。第二种情况需要通过判断对象非null进行避免。

12.3 构造方法

在12.1节使用了表达式new String("Hello World")，其中String("Hello World")是调用构造方法。构造方法是类中特殊方法，用来初始化类的实例变量，这个就是构造方法，它在创建对象（new运算符）之后自动调用。

Java构造方法的特点：

01. 构造方法名必须与类名相同。
02. 构造方法没有任何返回值，包括void。
03. 构造方法只能与new运算符结合使用。

构造方法示例代码如下：

```
//Rectangle.java文件
package com.a51work6;

// 矩形类
public class Rectangle {

    // 矩形宽度
    int width;
    // 矩形高度
    int height;
    // 矩形面积
    int area;

    // 构造方法
    public Rectangle(int w, int h) {        ①
        width = w;
        height = h;
        area = getArea(w, h);
    }
    ...
}
```

代码第①行是声明了一个构造方法，其中有两个参数w和h，用来初始化Rectangle对象的两个成员变量width和height，注意前面没有任何的返回值。

12.3.1 默认构造方法

有时在类中根本看不到任何的构造方法。例如本节中User类代码如下：

```
//User.java文件
package com.a51work6;

public class User {

    // 用户名
    private String username;

    // 用户密码
    private String password;

}
```

从上述User类代码，只有两个成员变量，看不到任何的构造方法，但是还是可以调用无参数的构造方法创建User对象，见如下代码。

```
//HelloWorld.java文件
...
User user = new User();
```

Java虚拟机为没有构造方法的类，提供一个无参数的默认构造方法，默认构造方法其方法体内无任何语句，默认构造方法相当于如下代码：

```
//默认构造方法
public User() {
}
```

默认构造方法的方法体内无任何语句，也就不能够初始化成员变量了，那么这些成员变量就会使用默认值，成员变量默认值是与数据类型有关，具体内容可以参考9.1.2节中的表9-1所示。这里不再赘述。

12.3.2 构造方法重载

在一个类中可以有多个构造方法，它们具体有相同的名字（与类名相同），参数列表不同，所以它们之间一定是重载关系。

构造方法重载示例代码如下：

```
//Person.java文件
package com.a51work6;

import java.util.Date;

public class Person {

    // 名字
    private String name;
    // 年龄
    private int age;
    // 出生日期
    private Date birthDate;

    public Person(String n, int a, Date d) {     ①
        name = n;
        age = a;
        birthDate = d;
    }

    public Person(String n, int a) {             ②
        name = n;
        age = a;
    }

    public Person(String n, Date d) {            ③
        name = n;
        age = 30;
        birthDate = d;
    }

    public Person(String n) {                     ④
        name = n;
        age = 30;
    }
```

```
    public String getInfo() {
        StringBuilder sb = new StringBuilder();
        sb.append("名字: ").append(name).append('\n');
        sb.append("年龄: ").append(age).append('\n');
        sb.append("出生日期: ").append(birthDate).append('\n');
        return  sb.toString();
    }
}
```

上述代码Person类代码提供了4个重载的构造方法，如果有准确的姓名、年龄和出生日期信息，则可以选用代码第①行的构造方法创建Person对象；如果只有姓名和年龄信息则可选用代码第②行的构造方法创建Person对象；如果只有姓名和出生日期信息则可选用代码第③行的构造方法创建Person对象；如果只有姓名信息则可选用代码第④行的构造方法创建Person对象。

12.3.3 构造方法封装

构造方法也可以进行封装，访问级别与普通方法一样，构造方法的访问级别参考表11-1所示。示例代码如下：

```
//Person.java文件
package com.a51work6;

import java.util.Date;

public class Person {

    // 名字
    private String name;
    // 年龄
    private int age;
    // 出生日期
    private Date birthDate;

    // 公有级别限制
    public Person(String n, int a, Date d) {    ①
        name = n;
        age = a;
        birthDate = d;
    }

    // 默认级别限制
    Person(String n, int a) {                   ②
        name = n;
        age = a;
    }

    // 保护级别限制
    protected Person(String n, Date d) {        ③
        name = n;
        age = 30;
        birthDate = d;
    }

    // 私有级别限制
    private Person(String n) {                  ④
        name = n;
        age = 30;
    }

    ...
}
```

上述代码第①行是声明公有级别的构造方法。代码第②行是声明默认级别，默认级别只能在同一个包中访问。代码第③行是保护级别的构造方法，该构造方法在同一包中与默认级别相同，在不同包中可以被子类继承。代码第④行是私有级别构造方法，该构造方法只能在当前类中使用，不允许在外边访问，私有构造方法可以应用于单例设计模式[1]等设计。

[1]单例模式是一种常用的软件设计模式，单例模式可以保证系统中一个类只有一个实例。

12.4 this关键字

前面章节中使用过this关键字，this指向对象本身，一个类可以通过this来获得一个代表它自身的对象变量。this使用在如下三种情况中：

- 调用实例变量。
- 调用实例方法。
- 调用其他构造方法。

使用this变量的示例代码：

```
//Person.java文件
package com.a51work6;

import java.util.Date;

public class Person {

    // 名字
    private String name;
    // 年龄
    private int age;
    // 出生日期
    private Date birthDate;

    // 三个参数构造方法
    public Person(String name, int age, Date d) {       ①
        this.name = name;                               ②
        this.age = age;                                 ③
        birthDate = d;
        System.out.println(this.toString());            ④
    }

    public Person(String name, int age) {
        // 调用三个参数构造方法
        this(name, age, null);                          ⑤
    }

    public Person(String name, Date d) {
        // 调用三个参数构造方法
        this(name, 30, d);                              ⑥
    }

    public Person(String name) {
        // System.out.println(this.toString());
        // 调用Person(String name, Date d)构造方法
        this(name, null);                               ⑦
    }

    @Override
    public String toString() {
        return "Person [name=" + name                   ⑧
                + ", age=" + age                        ⑨
                + ", birthDate=" + birthDate + "]";
    }
}
```

上述代码中多次用到了this关键字，下面详细分析一下。代码第①行声明三个参数构造方法，其中参数

name和age与实例变量name和age命名冲突，参数是作用域为整个方法的局部变量，为了防止局部变量与成员变量命名发生冲突，可以使用this调用成员变量，见代码第②行和第③行。注意代码第⑧行和第⑨行的name和age变量没有冲突，所以可以不使用this调用。

this也可以调用本对象的方法，见代码第④行的this.toString()语句，在本例中this可以省略。

在多个构造方法重载时，一个构造方法可以调用其他的构造方法，这样可以减少代码量，上述代码第⑤行this(name, age, null)使用this调用其他构造方法。类似调用还有代码第⑥行的this(name, 30, d)和第⑦行的this(name, null)。

注意　使用this调用其他构造方法时，this语句一定是该构造方法的第一条语句。例如在代码第⑦行之前调用toString()方法则会发生错误。

12.5 对象销毁

对象不再使用时应该销毁。C++语言对象是通过delete语句手动释放，Java语言对象是由垃圾回收器（Garbage Collection）收集然后释放，程序员不用关心释放的细节。自动内存管理是现代计算机语言发展趋势，例如：C#语言的垃圾回收，Objective-C和Swift语言的ARC（内存自动引用计数管理）。

垃圾回收器（Garbage Collection）的工作原理是：当一个对象的引用不存在时，认为该对象不再需要，垃圾回收器自动扫描对象的动态内存区，把没有引用的对象作为垃圾收集起来并释放。

本章小结

通过对本章的学习，可以了解如何创建Java对象，理解构造方法的作用。此外，还介绍了this关键字的使用，以及如何销毁对象。

第 13 章　继承与多态

类的继承性是面向对象语言的基本特性，多态性的前提是继承性。Java支持继承性和多态性。本章讨论Java继承性和多态性。

13.1 Java中的继承

为了了解继承性，先看这样一个场景：一位面向对象的程序员小赵，在编程过程中需要描述和处理个人信息，于是定义了类Person，如下所示：

```
//Person.java文件
package com.a51work6;

import java.util.Date;

public class Person {

    // 名字
    private String name;
    // 年龄
    private int age;
    // 出生日期
    private Date birthDate;

    public String getInfo() {
        return "Person [name=" + name
                + ", age=" + age
                + ", birthDate=" + birthDate + "]";
    }

}
```

一周以后，小赵又遇到了新的需求，需要描述和处理学生信息，于是他又定义了一个新的类Student，如下所示：

```
//Student.java文件
package com.a51work6;

import java.util.Date;

public class Student {

    // 所在学校
    public String school;
    // 名字
    private String name;
    // 年龄
    private int age;
    // 出生日期
    private Date birthDate;

    public String getInfo() {
        return "Person [name=" + name
                + ", age=" + age
                + ", birthDate=" + birthDate + "]";
    }
}
```

很多人会认为小赵的做法能够理解并相信这是可行的，但问题在于Student和Person两个类的结构太接近了，后者只比前者多了一个属性school，却要重复定义其他所有的内容，实在让人“不甘心”。Java提供了解决类似问题的机制，那就是类的继承，代码如下所示：

```
//Student.java文件
```

```
package com.a51work6;

import java.util.Date;

public class Student extends Person {
    // 所在学校
    private String school;
}
```

Student类继承了Person类中的所有成员变量和方法，从上述代码可以见继承使用的关键字是extends，extends后面的Person是父类。

如果在类的声明中没有使用extends关键字指明其父类，则默认父类为Object类，java.lang.Object类是Java的根类，所有Java类包括数组都直接或间接继承了Object类，在Object类中定义了一些有关面向对象机制的基本方法，如equals()、toString()和finalize()等方法。

提示　一般情况下，一个子类只能继承一个父类，这称为“单继承”，但有的情况下一个子类可以有多个不同的父类，这称为“多重继承”。在Java中，类的继承只能是单继承，而多重继承可以通过实现多个接口实现。也就是说，在Java中，一个类只能继承一个父类，但是可以实现多个接口。

提示　面向对象分析与设计（OOAD）时，会用到UML图[1]，其中类图非常重要，用来描述系统静态结构。Student继承Person的类图如图13-1所示。类图中的各个元素说明如图13-2所示，类用矩形表示，一般分为上、中、下三个部分，上部分是类名，中部分是成员变量，下部分是成员方法。实线+空心箭头表示继承关系，箭头指向父类，箭头末端是子类。UML类图中还有很多关系，如图13-3所示，如图虚线＋空心箭头表示实线关系，箭头指向接口，箭头末端是实线类。

[1]UML是Unified Modeling Language的缩写，即统一标准建模语言。它集成了各种优秀的建模方法学发展而来的。UML图常用的有例图、协作图、活动图、序列图、部署图、构件图、类图、状态图。

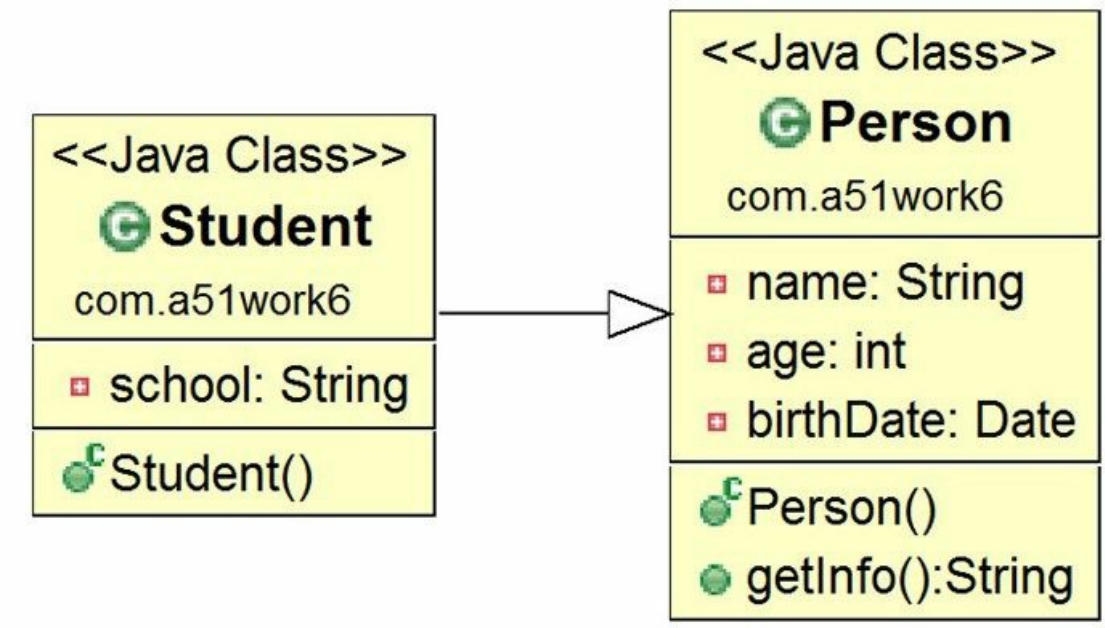


图**13-1　Student**继承**Person**的类图

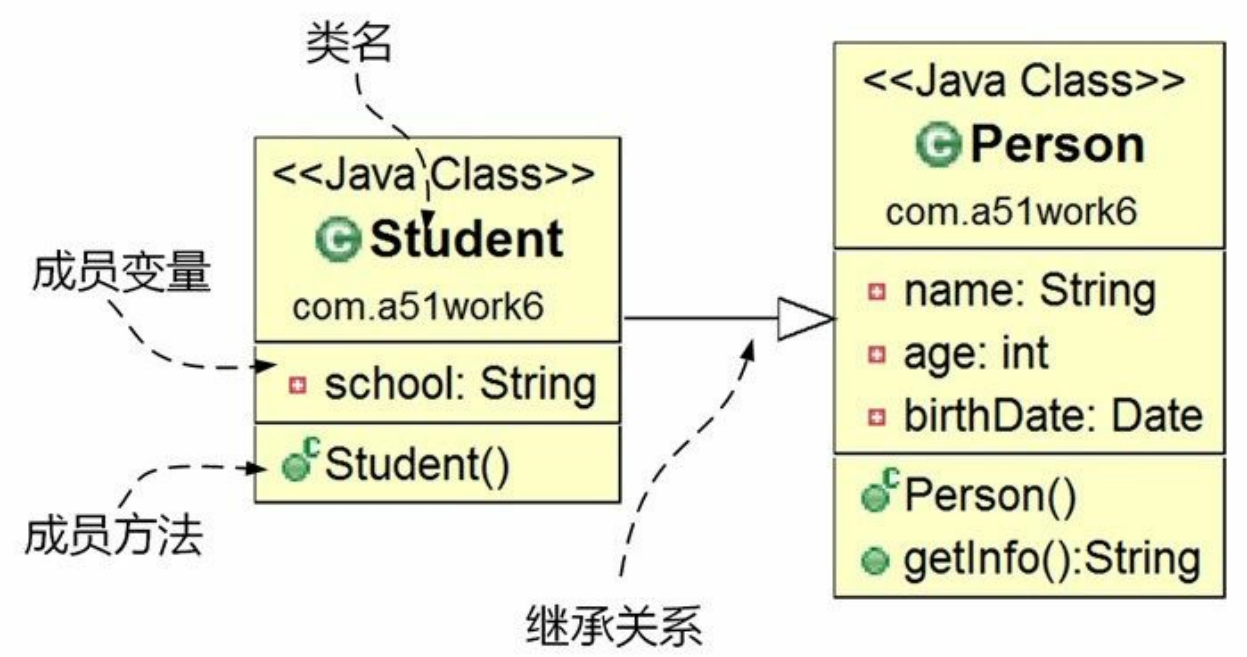


图**13-2**　类图中元素

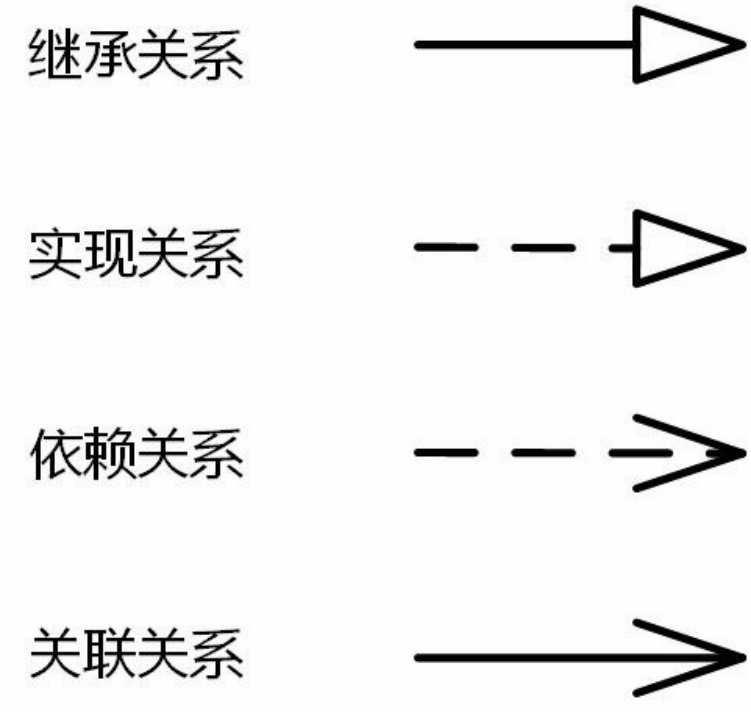


图**13-3**　元素之间关系

13.2 调用父类构造方法

当子类实例化时，不仅需要初始化子类成员变量，也需要初始化父类成员变量，初始化父类成员变量需要调用父类构造方法，子类使用super关键字调用父类构造方法。

下面看一个示例，现有父类Person和子类Student，它们类图如图13-4所示。

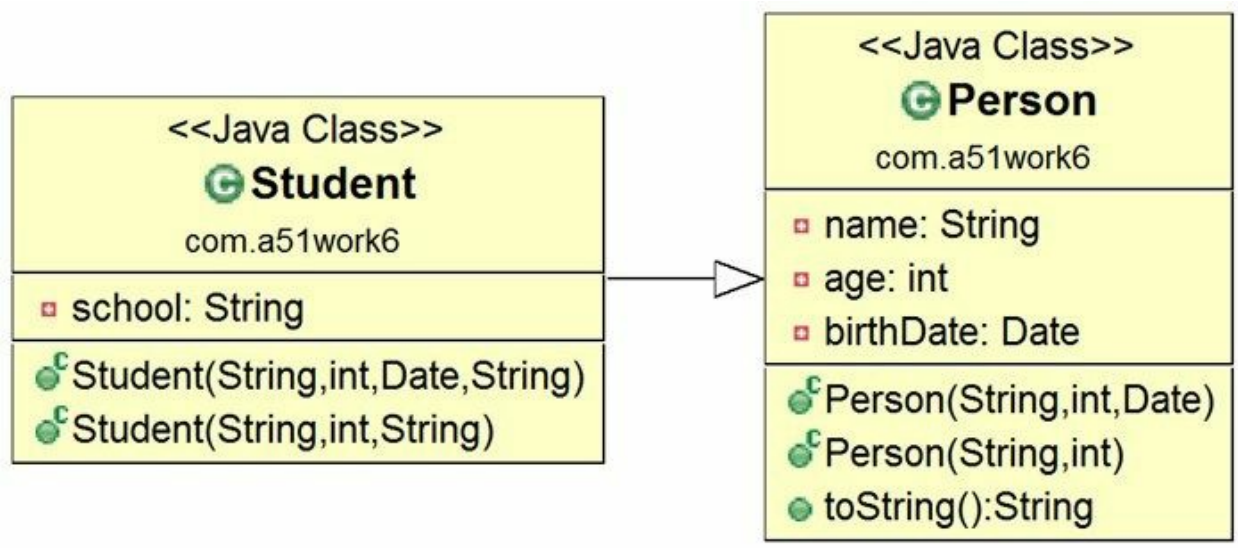


图**13-4** **Person**和**Student**类图

父类Person代码如下：

```
//Person.java文件
package com.a51work6;

import java.util.Date;

public class Person {

    // 名字
    private String name;
    // 年龄
    private int age;
    // 出生日期
    private Date birthDate;

    // 三个参数构造方法
    public Person(String name, int age, Date d) {
        this.name = name;
        this.age = age;
        birthDate = d;
    }

    public Person(String name, int age) {
        // 调用三个参数构造方法
        this(name, age, new Date());
    }
    ...
}
```

子类Student代码如下：

```
//Student.java文件
package com.a51work6;

import java.util.Date;

public class Student extends Person {

    // 所在学校
    private String school;

    public Student(String name, int age, Date d, String school) {
        super(name, age, d);                                          ①
        this.school = school;
    }

    public Student(String name, int age, String school) {
        // this.school = school;//编译错误
        super(name, age);                                             ②
        this.school = school;
    }

    public Student(String name, String school) { // 编译错误          ③
        // super(name, 30);
        this.school = school;
    }
}
```

在Student子类代码第①行和第②行是调用父类构造方法，代码第①行super(name, age, d)语句是调用父类的Person(String name, int age, Date d)构造方法，代码第②行super(name, age)语句是调用父类的Person(String name, int age)构造方法。

提示　super语句必须位于子类构造方法的第一行。

代码第③行构造方法由于没有super语句，编译器会试图调用父类默认构造方法（无参数构造方法），但是父类Person并没有默认构造方法，因此会发生编译错误。解决这个编译错误有三种办法：

01. 在父类Person中添加默认构造方法，子类Student会隐式调用父类的默认构造方法。
02. 在子类Studen构造方法添加super语句，显式调用父类构造方法，super语句必须是第一条语句。
03. 在子类Studen构造方法添加this语句，显式调用当前对象其他构造方法，this语句必须是第一条语句。

13.3 成员变量隐藏和方法覆盖

子类继承父类后，在子类中有可能声明了与父类一样的成员变量或方法，那么会出现什么情况呢？

13.3.1 成员变量隐藏

子类成员变量与父类一样，会屏蔽父类中的成员变量，称为“成员变量隐藏”。示例代码如下：

```
//ParentClass.java文件
package com.a51work6;

class ParentClass {
    // x成员变量
    int x = 10;                                          ①
}

class SubClass extends ParentClass {
    // 屏蔽父类x成员变量
    int x = 20;                                          ②

    public void print() {
        // 访问子类对象x成员变量
        System.out.println("x = " + x);                  ③
        // 访问父类x成员变量
        System.out.println("super.x = " + super.x);      ④
    }
}
```

调用代码如下：

```
//HelloWorld.java文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {
        //实例化子类SubClass
        SubClass pObj = new SubClass();
        //调用子类print方法
        pObj.print();
    }
}
```

运行结果如下：

```
x = 20
super.x = 10
```

上述代码第①行是在ParentClass类声明x成员变量，那么在它的子类SubClass代码第②行也声明了x成员变量，它会屏蔽父类中的x成员变量。那么代码第③行的x是子类中的x成员变量。如果要调用父类中的x成员变量，则需要super关键字，见代码第④行的super.x。

13.3.2 方法的覆盖（**Override**）

如果子类方法完全与父类方法相同，即：相同的方法名、相同的参数列表和相同的返回值，只是方法

体不同，这称为子类覆盖（Override）父类方法。

示例代码如下：

```
//ParentClass.java文件
package com.a51work6;

class ParentClass {
    // x成员变量
    int x;

    protected void setValue() {                    ①
        x = 10;
    }
}

class SubClass extends ParentClass {
    // 屏蔽父类x成员变量
    int x;

    @Override
    public void setValue() { // 覆盖父类方法       ②
        // 访问子类对象x成员变量
        x = 20;
        // 调用父类setValue()方法
        super.setValue();
    }

    public void print() {
        // 访问子类对象x成员变量
        System.out.println("x = " + x);
        // 访问父类x成员变量
        System.out.println("super.x = " + super.x);
    }
}
```

调用代码如下：

```
//HelloWorld.java文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {
        //实例化子类SubClass
        SubClass pObj = new SubClass();
        //调用setValue方法
        pObj.setValue();
        //调用子类print方法
        pObj.print();
    }
}
```

运行结果如下：

```
x = 20
super.x = 10
```

上述代码第①行是在ParentClass类声明setValue方法，那么在它的子类SubClass代码第②行覆盖父类中

的setValue方法，在声明方法时添加@Override注解，@Override注解不是方法覆盖必须的，它只是锦上添花，但添加@Override注解有两个好处：

01. 提高程序的可读性。

02. 编译器检查@Override注解的方法在父类中是否存在，如果不存在则报错。

注意　方法覆盖时应遵循的原则：

01. 覆盖后的方法不能比原方法有更严格的访问控制（可以相同）。例如将代码第②行访问控制public修改private，那么会发生编译错误，因为父类原方法是protected。

02. 覆盖后的方法不能比原方法产生更多的异常。

13.4 多态

在面向对象程序设计中多态是一个非常重要的特性，理解多态有利于进行面向对象的分析与设计。

13.4.1 多态概念

发生多态要有三个前提条件：

01. 继承。多态发生一定要子类和父类之间。

02. 覆盖。子类覆盖了父类的方法。

03. 声明的变量类型是父类类型，但实例则指向子类实例。

下面通过一个示例理解什么多态。如图13-5所示，父类Figure（几何图形）类有一个onDraw（绘图）方法，Figure（几何图形）它有两个子类Ellipse（椭圆形）和Triangle（三角形），Ellipse和Triangle覆盖onDraw方法。Ellipse和Triangle都有onDraw方法，但具体实现的方式不同。

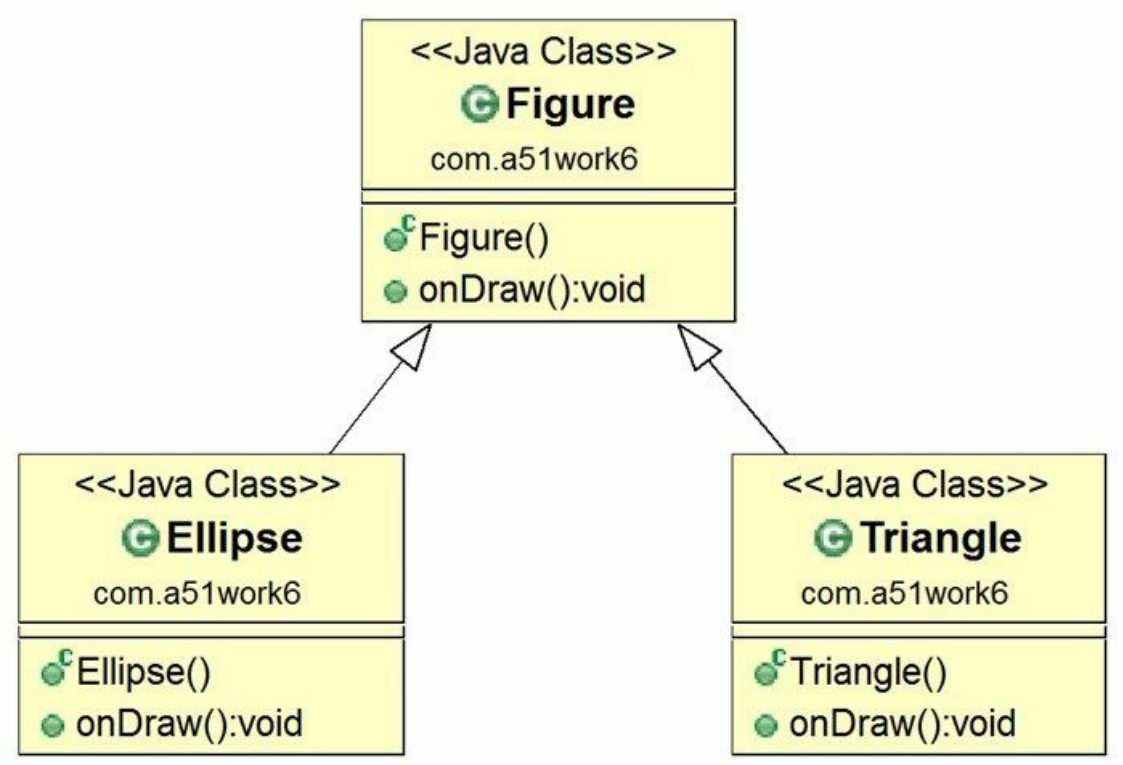


图**13-5** 几何图形类图

具体代码如下：

```
//Figure.java文件
package com.a51work6;

public class Figure {

    //绘制几何图形方法
    public void onDraw() {
```

```
        System.out.println("绘制Figure...");
    }
}

//Ellipse.java文件
package com.a51work6;

//几何图形椭圆形
public class Ellipse extends Figure {

    //绘制几何图形方法
    @Override
    public void onDraw() {
        System.out.println("绘制椭圆形...");
    }

}

//Triangle.java文件
package com.a51work6;

//几何图形三角形
public class Triangle extends Figure {

    // 绘制几何图形方法
    @Override
    public void onDraw() {
        System.out.println("绘制三角形...");
    }
}
```

调用代码如下：

```
//HelloWorld.java文件
package com.a51work6;
public class HelloWorld {
    public static void main(String[] args) {

        // f1变量是父类类型，指向父类实例
        Figure f1 = new Figure();                          ①
        f1.onDraw();

        //f2变量是父类类型，指向子类实例，发生多态
        Figure f2 = new Triangle();                        ②
        f2.onDraw();

        //f3变量是父类类型，指向子类实例，发生多态
        Figure f3 = new Ellipse();                         ③
        f3.onDraw();

        //f4变量是子类类型，指向子类实例
        Triangle f4 = new Triangle();                      ④
        f4.onDraw();

    }
}
```

上述带代码第②行和第③行是符合多态的三个前提，因此会发生多态。而代码第①行和第④行都不符合，没有发生多态。

运行结果如下：

```
绘制Figure...
绘制三角形...
绘制椭圆形...
绘制三角形...
```

从运行结果可知，多态发生时，Java虚拟机运行时根据引用变量指向的实例调用它的方法，而不是根据引用变量的类型调用。

13.4.2 引用类型检查

有时候需要在运行时判断一个对象是否属于某个引用类型，这时可以使用instanceof运算符，instanceof运算符语法格式如下：

```
obj  instanceof  type
```

其中obj是一个对象，type是引用类型，如果obj对象是type引用类型实例则返回true，否则false。

为了介绍引用类型检查，先看一个示例，如图13-6所示的类图，展示了继承层次树，Person类是根类，Student是Person的直接子类，Worker是Person的直接子类。

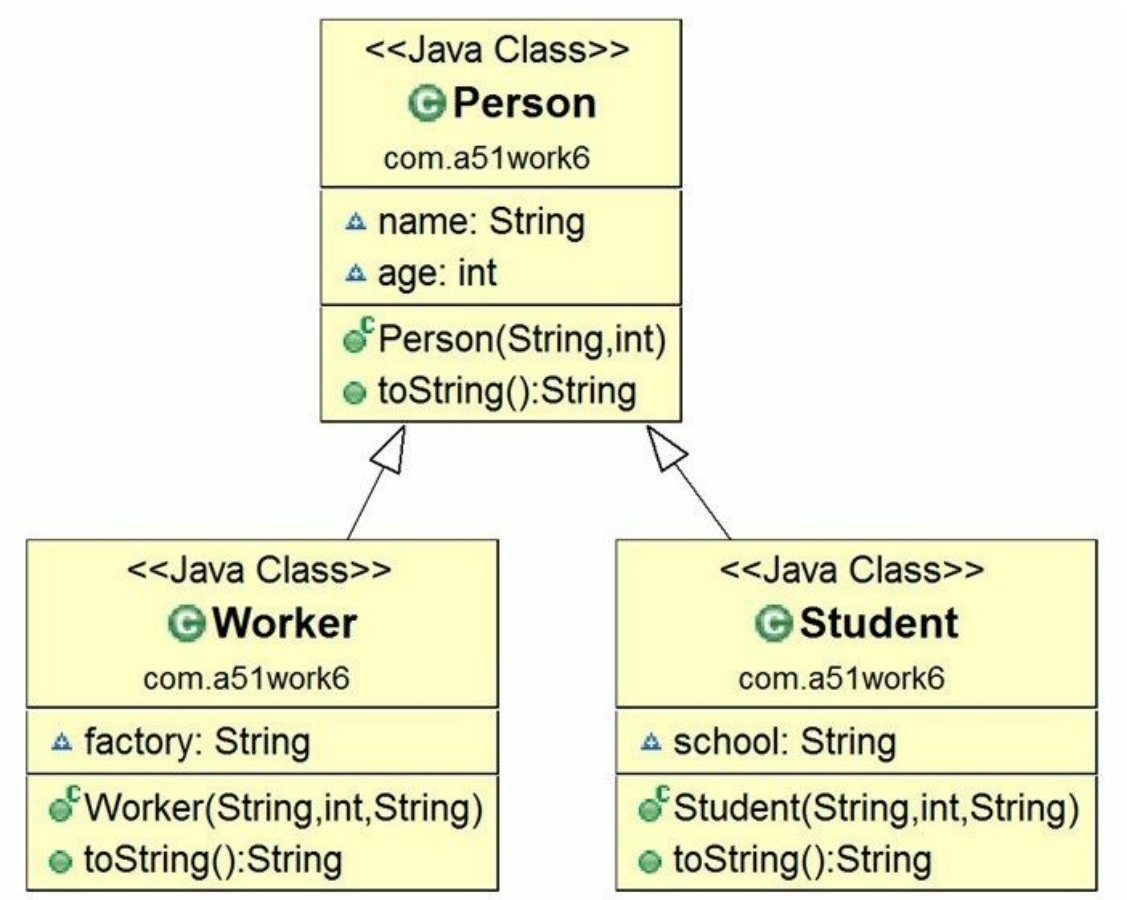


图**13-6** 继承关系类图

继承层次树中具体实现代码如下：

```
//Person.java文件
package com.a51work6;
public class Person {

    String name;
    int age;

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

    @Override
    public String toString() {
        return "Person [name=" + name
                + ", age=" + age + "]";
    }
}

//Worker.java文件
package com.a51work6;
public class Worker extends Person {

    String factory;

    public Worker(String name, int age, String factory) {
        super(name, age);
        this.factory = factory;
    }

    @Override
    public String toString() {
        return "Worker [factory=" + factory
                + ", name=" + name
                + ", age=" + age + "]";
    }
}

//Student.java文件
package com.a51work6;
public class Student extends Person {

    String school;

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

    @Override
    public String toString() {
        return "Student [school=" + school
                + ", name=" + name
                + ", age=" + age + "]";
    }

}
```

调用代码如下：

```
//HelloWorld.java文件
package com.a51work6;

public class HelloWorld {
```

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

        Student student1 = new Student("Tom", 18, "清华大学");          ①
        Student student2 = new Student("Ben", 28, "北京大学");
        Student student3 = new Student("Tony", 38, "香港大学");         ②

        Worker worker1 = new Worker("Tom", 18, "钢厂");                 ③
        Worker worker2 = new Worker("Ben", 20, "电厂");                 ④

        Person[] people = { student1, student2, student3, worker1, worker2 };    ⑤

        int studentCount = 0;
        int workerCount = 0;

        for (Person item : people) {                ⑥
            if (item instanceof Worker) {           ⑦
                workerCount++;
            } else if (item instanceof Student) {   ⑧
                studentCount++;
            }
        }
        System.out.printf("工人人数：%d，学生人数：%d", workerCount, studentCount);
    }
}
```

上述代码第①行和第②行创建了3个Student实例，代码第③行和第④行创建了两个Worker实例，然后程序把这5个实例放入people数组中。

代码第⑥行使用for-each遍历people数组集合，当从people数组中取出元素时，元素类型是People类型，但是实例不知道是哪个子类（Student和Worker）实例。代码第⑦行item instanceof Worker表达式是判断数组中的元素是否是Worker实例；类似地，第⑧行item instanceof Student表达式是判断数组中的元素是否是Student实例。

输出结果如下：

```
工人人数：2，学生人数：3
```

13.4.3 引用类型转换

在6.7节介绍过数值类型相互转换，引用类型可以进行转换，但并不是所有的引用类型都能互相转换，只有属于同一棵继承层次树中的引用类型才可以转换。

在上一节示例上修改HelloWorld.java代码如下：

```
//HelloWorld.java文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        Person p1 = new Student("Tom", 18, "清华大学");
        Person p2 = new Worker("Tom", 18, "钢厂");

        Person p3 = new Person("Tom", 28);
        Student p4 = new Student("Ben", 40, "清华大学");
        Worker p5 = new Worker("Tony", 28, "钢厂");
        …
```

```
    }
}
```

上述代码创建了5个实例p1、p2、p3、p4和p5，它们的类型都是Person继承层次树中的引用类型，p1和p4是Student实例，p2和p5是Worker实例，p3是Person实例。首先，对象类型转换一定发生在继承的前提下，p1和p2都声明为Person类型，而实例是由Person子类型实例化的。

表13-1归纳了p1、p2、p3、p4和p5这5个实例与Worker、Student和Person这3种类型之间的转换关系。

表 **13-1**　类型转换

对 象	Person 类型	Worker 类型	Student 类型	说　明
p1	支持	不支持	支持（向下转型）	类型：Person 实例：Student
p2	支持	支持（向下转型）	不支持	类型：Person 实例：Worker
p3	支持	不支持	不支持	类型：Person 实例：Person
p4	支持（向上转型）	不支持	支持	类型：Student 实例：Student
p5	支持（向上转型）	支持	不支持	类型：Worker 实例：Worker

作为这段程序的编写者是知道p1本质上是Student实例，但是表面上看是Person类型，编译器也无法推断p1的实例是Person、Student还是Worker。此时可以使用instanceof操作符来判断它是哪一类的实例。

引用类型转换也是通过小括号运算符实现，类型转换有两个方向：将父类引用类型变量转换为子类类型，这种转换称为向下转型（downcast）；将子类引用类型变量转换为父类类型，这种转换称为向上转型（upcast）。向下转型需要强制转换，而向上转型是自动的。

下面通过示例详细说明一下向下转型和向上转型，在HelloWorld.java的main方法中添加如下代码：

```
// 向上转型
Person p = (Person) p4;            ①

// 向下转型
Student p11 = (Student) p1;        ②
Worker p12 = (Worker) p2;          ③

// Student p111 = (Student) p2;    //运行时异常     ④
if (p2 instanceof Student) {
    Student p111 = (Student) p2;
}
// Worker p121 = (Worker) p1;    //运行时异常       ⑤
if (p1 instanceof Worker) {
    Worker p121 = (Worker) p1;
}
// Student p131 = (Student) p3;    //运行时异常     ⑥
if (p3 instanceof Student) {
    Student p131 = (Student) p3;
}
```

上述代码第①行将p4对象转换为Person类型，p4本质上是Student实例，这是向上转型，这种转换是自动的，其实不需要小括号(Person)进行强制类型转换。

代码第②行和第③行是向下类型转换，它们的转型都能成功。而代码第④、⑤、⑥行都会发生运行时异常ClassCastException，如果不能确定实例是哪一种类型，可以在转型之前使用instanceof运算符判断一下。

13.5 再谈final关键字

在前面的学习过程中，为了声明常量使用过final关键字，在Java中final关键字的作用还有很多，final关键字能修饰变量、方法和类。下面详细说明。

13.5.1 final修饰变量

final修饰的变量即成为常量，只能赋值一次，但是final所修饰局部变量和成员变量有所不同。

01. final修饰的局部变量必须使用之前被赋值一次才能使用。
02. final修饰的成员变量在声明时没有赋值的叫“空白final变量”。空白final变量必须在构造方法或静态代码块中初始化。

final修饰变量示例代码如下：

```
//FinalDemo.java文件
package com.a51work6;

class FinalDemo {

    void doSomething() {
        // 没有在声明的同时赋值
        final int e;                          ①
        // 只能赋值一次
        e = 100;                              ②
        System.out.print(e);
        // 声明的同时赋值
        final int f = 200;                    ③
    }

    //实例常量
    final int a = 5; // 直接赋值              ④
    final int b; // 空白final变量             ⑤

    //静态常量
    final static int c = 12;// 直接赋值       ⑥
    final static int d; // 空白final变量      ⑦

    // 静态代码块
    static {
        // 初始化静态变量
        d = 32;                               ⑧
    }

    // 构造方法
    FinalDemo() {
        // 初始化实例变量
        b = 3;                                ⑨
        // 第二次赋值，会发生编译错误
        // b = 4;                             ⑩
    }
}
```

上述代码第①行和第③行是声明局部常量，其中第①行只是声明没有赋值，但必须在使用之前赋值（见代码第②行），其实局部常量最好在声明的同时初始化。

代码第④、⑤、⑥和⑦行都声明成员常量。代码第④和⑤行是实例常量，如果是空白final变量（见代码第⑤行），则需要在构造方法中初始化（见代码第⑨行）。代码第⑥和⑦行是静态常量，如果是空

白final变量（见代码第⑦行），则需要在静态代码块中初始化（见代码第⑧行）。

另外，无论是那种常量只能赋值一次，见代码第⑩行为b常量赋值，因为之前b已经赋值过一次，因此这里会发生编译错误。

13.5.2 final修饰类

final修饰的类不能被继承。有时出于设计安全的目的，不想让自己编写的类被别人继承，这时可以使用final关键字修饰父类。

示例代码如下：

```
//SuperClass.java文件
package com.a51work6;

final class SuperClass {
}

class SubClass extends SuperClass { //编译错误
}
```

在声明SubClass类时会发生编译错误。

13.5.3 final修饰方法

final修饰的方法不能被子类覆盖。有时也是出于设计安全的目的，父类中的方法不想被别人覆盖，这是可以使用final关键字修饰父类中方法。

示例代码如下：

```
//SuperClass.java文件
package com.a51work6;

class SuperClass {
    final void doSomething() {
        System.out.println("in SuperClass.doSomething()");
    }
}

class SubClass extends SuperClass {
    @Override
    void doSomething() { //编译错误
        System.out.println("in SubClass.doSomething()");
    }
}
```

子类中的void doSomething()方法试图覆盖父类中void doSomething()方法，父类中的void doSomething()方法是final的，因此会发生编译错误。

本章小结

通过对本章的学习，首先介绍了Java中的继承概念，在继承时会发生方法的覆盖、变量的隐藏。然后介绍了Java中的多态概念，广大读者需要熟悉多态发生的条件，掌握引用类型检查和类型转换。最后还介绍了final关键字。

第 **14** 章　抽象类与接口

设计良好的软件系统应该具备“可复用性”和“可扩展性”，能够满足用户需求的不断变更。使用抽象类和接口是实现“可复用性”和“可扩展性”重要的设计手段。

14.1 抽象类

Java语言提供了两种类：一种是具体类；另一种是抽象子类。前面章节接触的类都是具体类。这一节介绍一下抽象类。

14.1.1 抽象类概念

在13.4.1节介绍多态时，使用过几何图形类示例，其中Figure（几何图形）类中有一个onDraw（绘图）方法，Figure有两个子类Ellipse（椭圆形）和Triangle（三角形），Ellipse和Triangle覆盖onDraw方法。

作为父类Figure（几何图形）并不知道在实际使用时有多少个子类，目前有椭圆形和三角形，那么不同的用户需求可能会有矩形或圆形等其他几何图形，而onDraw方法只有确定是哪一个子类后才能具体实现。Figure中的onDraw方法不能具体实现，所以只能是一个抽象方法。在Java中具有抽象方法的类称为“抽象类”，Figure是抽象类，其中的onDraw方法是抽象方法。如图14-1所示类图中Figure是抽象类，Ellipse和Triangle是Figure子类实现Figure的抽象方法onDraw。

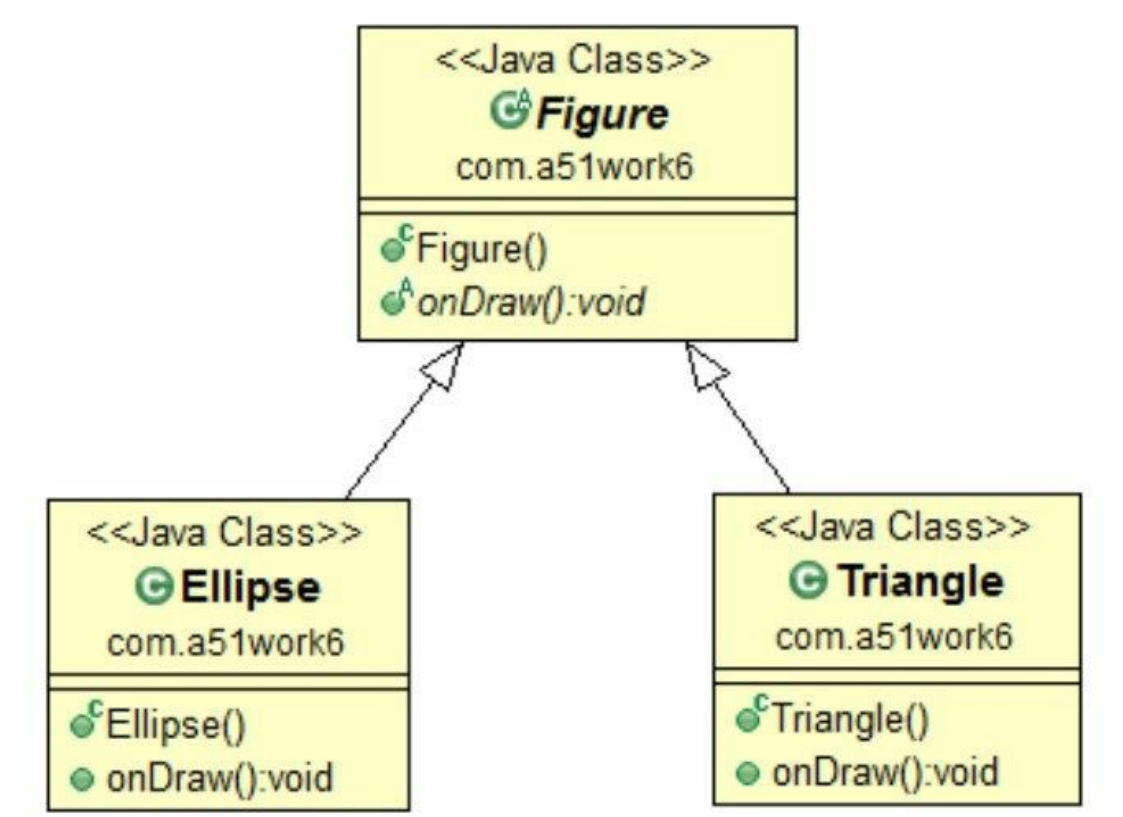


图**14-1** 抽象类几何图形类图

提示 在UML类图抽象类和抽象方法字体是斜体的，见图14-1所示中的Figure类和onDraw方法都是斜体的。

14.1.2 抽象类声明和实现

在Java中抽象类和抽象方法的修饰符是abstract，声明抽象类Figure示例代码如下：

```
//Figure.java文件
```

```
package com.a51work6;

public abstract class Figure {            ①
    // 绘制几何图形方法
    public abstract void onDraw();        ②
}
```

代码第①行是声明抽象类，在类前面加上abstract修饰符。代码第②行声明抽象方法，方法前面的修饰符也是abstract，注意抽象方法中只有方法的声明，没有方法的实现，即没有大括号（{}）部分。

注意　如果一个方法被声明为抽象的，那么这个类也必须声明为抽象的。而一个抽象类中，可以有0~n个抽象方法，以及0~n具体方法。

设计抽象方法目的就是让子类来实现的，否则抽象方法就没有任何意义，实现抽象类示例代码如下：

```
//Ellipse.java文件
package com.a51work6;

//几何图形椭圆形
public class Ellipse extends Figure {

    //绘制几何图形方法
    @Override
    public void onDraw() {
        System.out.println("绘制椭圆形...");
    }
}

//Triangle.java文件
package com.a51work6;

//几何图形三角形
public class Triangle extends Figure {

    // 绘制几何图形方法
    @Override
    public void onDraw() {
        System.out.println("绘制三角形...");
    }
}
```

上述代码声明了两个具体类Ellipse和Triangle，它们实现（覆盖）了抽象类Figure的抽象方法onDraw。

调用代码如下：

```
//HelloWorld.java文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        // f1变量是父类类型，指向子类实例，发生多态
        Figure f1 = new Triangle();
        f1.onDraw();

        // f2变量是父类类型，指向子类实例，发生多态
        Figure f2 = new Ellipse();
        f2.onDraw();
    }
```

```
}
```

上述代码中实例化两个具体类Triangle和Ellipse，对象f1和f2是Figure引用类型。

注意　抽象类不能被实例化,只有具体类才能被实例化。

14.2 使用接口

比抽象类更加抽象的是接口，在接口中所有的方法都是抽象的。

提示　Java 8之后接口中新增加了默认方法，因此“接口中所有的方法都是抽象的”这个提法在Java 8之后是有待商榷。

14.2.1 接口概念

其实14.1.1节抽象类Figure可以更加彻底，即Figure接口，接口中所有方法都是抽象的，而且接口可以有成员变量。将14.1.1节几何图形类改成接口后，类图如图14.2所示。

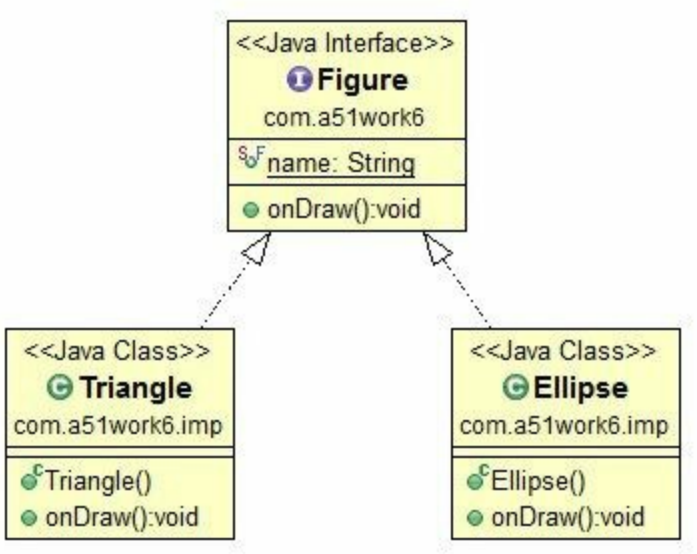


图**14-2**　接口几何图形类图

提示　在UML类图中接口的图标是“I”，见图14-2所示中的Figure接口。类的图标是“C”，见图14-2所示中的Triangle接口。

14.2.2 接口声明和实现

在Java中接口的声明使用的关键字是interface，声明接口Figure示例代码如下：

```
//Figure.java文件
package com.a51work6;

public interface Figure {                                    ①
    //接口中静态成员变量
    String name = "几何图形";//省略public static final     ②

    // 绘制几何图形方法
    void onDraw();            //省略public                 ③
}
```

代码第①行是声明Figure接口，声明接口使用interface关键字，interface前面的修饰符是public或省略。public是公有访问级别，可以在任何地方访问。省略是默认访问级别，只能在当前包中访问。

代码第②行声明接口中的成员变量，在接口中成员变量都静态成员变量，即省略了public static final修饰符。代码第③行是声明抽象方法，即省略了public关键字。

某个类实现接口时，要在声明时使用implements关键字，当实现多个接口之间用逗号（,）分隔。实现接口时要实现接口中声明的所有方法。

实现接口Figure示例代码如下：

```
//Ellipse.java文件
package com.a51work6.imp;

import com.a51work6.Figure;

//几何图形椭圆形
public class Ellipse implements Figure {

    //绘制几何图形方法
    @Override
    public void onDraw() {
        System.out.println("绘制椭圆形...");
    }
}

//Triangle.java文件
package com.a51work6.imp;

import com.a51work6.Figure;

//几何图形三角形
public class Triangle implements Figure {

    // 绘制几何图形方法
    @Override
    public void onDraw() {
        System.out.println("绘制三角形...");
    }
}
```

上述代码声明了两个具体类Ellipse和Triangle，它们实现了接口Figure中的抽象方法onDraw。

调用代码如下：

```
//HelloWorld.java文件
import com.a51work6.imp.Ellipse;
import com.a51work6.imp.Triangle;

public class HelloWorld {

    public static void main(String[] args) {

        // f1变量是父类类型，指向子类实例，发生多态
        Figure f1 = new Triangle();
        f1.onDraw();
        System.out.println(f1.name);         ①
        System.out.println(Figure.name);     ②

        // f2变量是父类类型，指向子类实例，发生多态
        Figure f2 = new Ellipse();
        f2.onDraw();
    }
}
```

上述代码中实例化两个具体类Triangle和Ellipse，对象f1和f2是Figure接口引用类型。接口Figure中声明了成员变量，它是静态成员变量，代码第①行和第②行是访问name静态变量。

注意　接口与抽象类一样都不能被实例化。

14.2.3　接口与多继承

在C++语言中一个类可以继承多个父类，但这会有潜在的风险，如果两个父类有相同的方法，那么子类将继承哪一个父类方法呢？这就是C++多继承所导致的冲突问题。

在Java中只允许继承一个类，但可实现多个接口。通过实现多个接口方式满足多继承的设计需求。如果多个接口中即便有相同方法，它们也都是抽象的，子类实现它们不会有冲突。

图14-3所示是多继承类图，其中的有两个接口InterfaceA和InterfaceB，从类图中可以见两个接口中都有一个相同的方法void methodB()。AB实现了这两个接口，继承了Object父类。

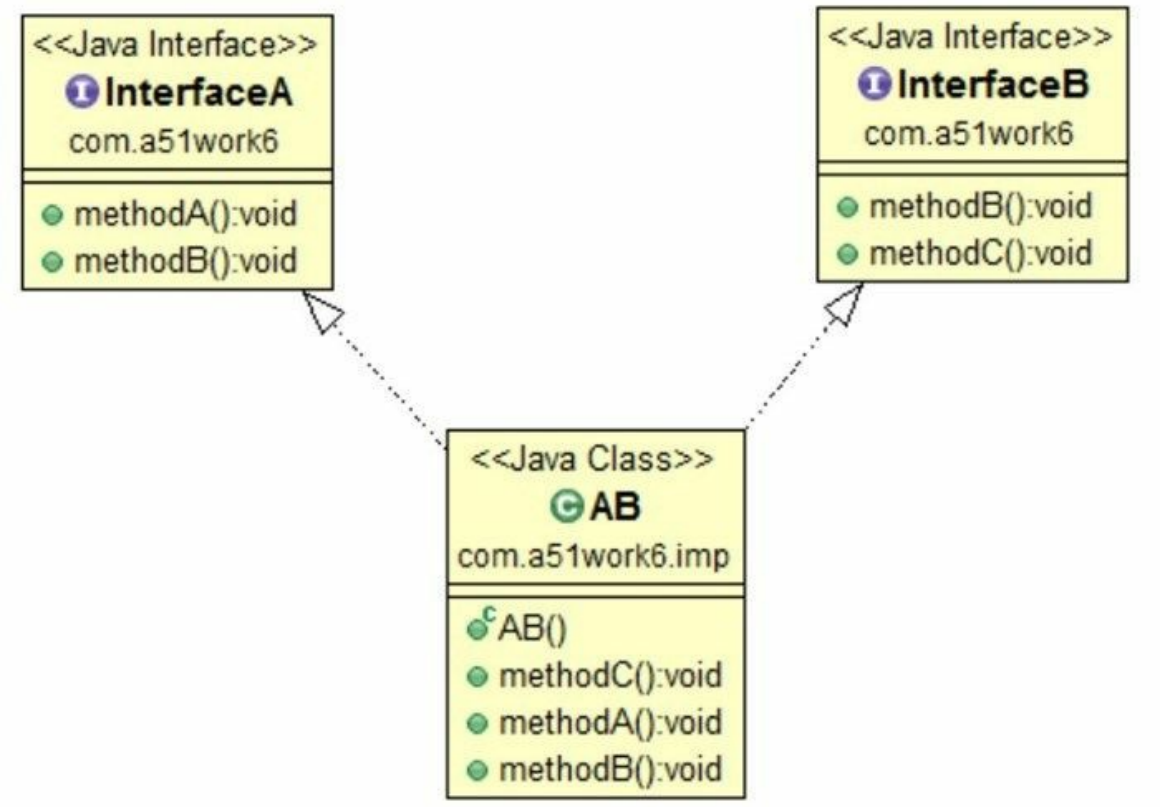


图**14-3**　多继承类图

接口InterfaceA和InterfaceB代码如下：

```
//InterfaceA.java文件
package com.a51work6;

public interface InterfaceA {

    void methodA();

    void methodB();
```

```
}

//InterfaceB.java文件
package com.a51work6;

public interface InterfaceB {

    void methodB();

    void methodC();
}
```

从代码中可见两个接口都有两个方法，其中方法methodB()完全相同。

实现接口InterfaceA和InterfaceB的AB类代码如下：

```
//AB.java文件
package com.a51work6.imp;

import com.a51work6.InterfaceA;
import com.a51work6.InterfaceB;

public class AB extends Object implements InterfaceA, InterfaceB {    ①

    @Override
    public void methodC() {
    }

    @Override
    public void methodA() {
    }

    @Override
    public void methodB() {            ②
    }
}
```

在AB类中的代码第②行实现methodB()方法。注意在AB类声明时，实现两个接口，接口之间使用逗号（，）分隔，见代码第①行。

14.2.4 接口继承

Java语言中允许接口和接口之间继承。由于接口中的方法都是抽象方法，所以继承之后也不需要做什么，因此接口之间的继承要比类之间的继承简单的多。如图4-4所示，其中InterfaceB继承了InterfaceA，在InterfaceB中还覆盖了InterfaceA中的methodB()方法。ABC是InterfaceB接口的实现类，从图4-4中可见ABC需要实现InterfaceA和InterfaceB接口中的所有方法。

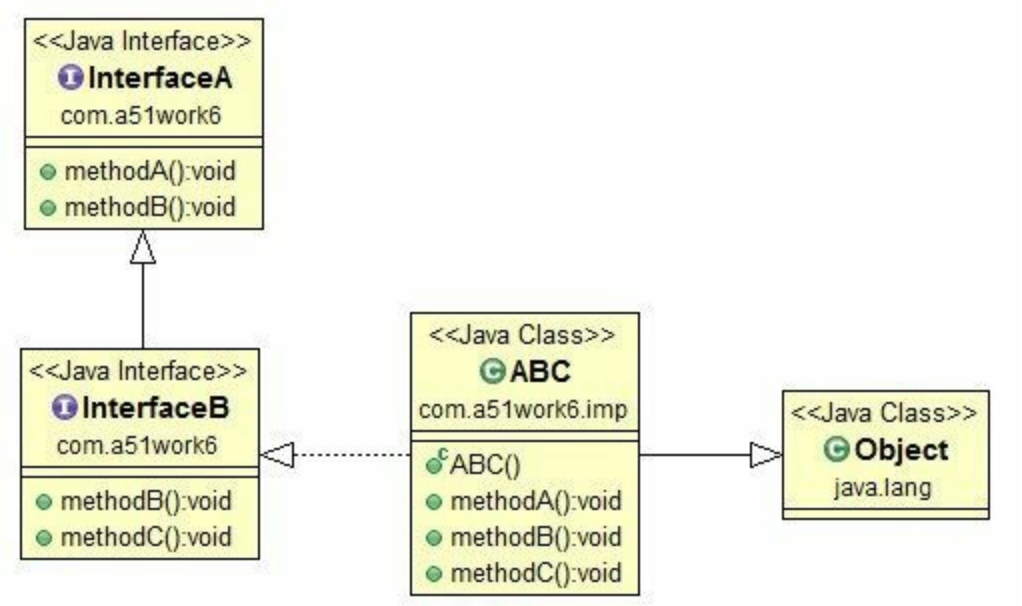


图**14-4**　接口继承类图

接口InterfaceA和InterfaceB代码如下：

```
//InterfaceA.java文件
package com.a51work6;

public interface InterfaceA {

    void methodA();

    void methodB();
}

//InterfaceB.java文件
package com.a51work6;

public interface InterfaceB extends InterfaceA {

    @Override
    void methodB();

    void methodC();
}
```

InterfaceB继承了InterfaceA，声明时也使用extends关键字。InterfaceB 中的methodB()覆盖了InterfaceA，事实上在接口中覆盖方法，并没有实际意义，因为它们都是抽象的，都是留给子类实现的。

实现接口InterfaceB的ABC类代码如下：

```
//ABC.java文件
package com.a51work6.imp;

import com.a51work6.InterfaceB;

public class ABC implements InterfaceB {
```

```
    @Override
    public void methodA() {
    }

    @Override
    public void methodB() {
    }

    @Override
    public void methodC() {
    }
}
```

ABC类实现了接口InterfaceB，事实上是实现InterfaceA和InterfaceB中所有方法，相当于同时实现InterfaceA和InterfaceB接口。

14.2.5 Java 8新特性默认方法和静态方法

在Java 8之前，尽管Java语言中接口已经非常优秀了，但相比其他面向对象的语言而言Java接口存在如下不足之处：

01. 不能可选实现方法，接口的方法全部是抽象的，实现接口时必须全部实现接口中方法，哪怕是有些方法并不需要，也必须实现。

02. 没有静态方法。

针对这些问题，Java 8在接口中提供了声明默认方法和静态方法的能力。接口示例代码如下：

```
//InterfaceA.java文件
package com.a51work6;

public interface InterfaceA {

    void methodA();

    String methodB();

    // 默认方法
    default int methodC() {
        return 0;
    }

    // 默认方法
    default String methodD() {
        return "这是默认方法...";
    }

    // 静态方法
    static double methodE() {
        return 0.0;
    }
}
```

在接口InterfaceA中声明了两个抽象方法methodA和methodB，两个默认方法methodC和methodD，还有声明了静态方法methodE。接口中的默认方法类似于类中具体方法，给出了具体实现，只是方法修饰符是default。接口中静态方法类似于类中静态方法。

实现接口示例代码如下：

```
//ABC.java文件
package com.a51work6.imp;

import com.a51work6.InterfaceA;

public class ABC implements InterfaceA {

    @Override
    public void methodA() {
    }

    @Override
    public String methodB() {
        return "实现methodB方法...";
    }

    @Override
    public int methodC() {
        return 500;
    }
}
```

实现接口时接口中原有的抽象方法在实现类中必须实现。默认方法可以根据需要有选择实现（覆盖）。静态方法不需要实现，实现类中不能拥有接口中的静态方法。

上述代码中ABC类实现了InterfaceA接口，InterfaceA接口中的两个默认方法ABC只是实现（覆盖）了methodB。

调用代码如下：

```
//HelloWorld.java文件
package com.a51work6.imp;

import com.a51work6.InterfaceA;

public class HelloWorld {

    public static void main(String[] args) {

        //声明接口类型，对象是实现类，发生多态
        InterfaceA abc = new ABC();

        // 访问实现类methodB方法
        System.out.println(abc.methodB());

        // 访问默认方法methodC
        System.out.println(abc.methodC());              ①

        // 访问默认方法methodD
        System.out.println(abc.methodD());              ②

        // 访问InterfaceA静态方法methodE
        System.out.println(InterfaceA.methodE());       ③
    }
}
```

运行结果：

```
实现methodB方法...
500
```

```
这是默认方法...
0.0
```

从运行结果可见，代码第①行调用默认方法methodC，是调用类AB中的实现。代码第②行调用默认方法methodD，是调用接口InterfaceA中的实现。代码第③行调用接口静态方法，只能通过接口名（InterfaceA）调用，不能通过实现类ABC调用，可以这样理解接口中声明的静态方法与其他实现类没有任何关系。

14.3 抽象类与接口区别

经过前面的学习，广大读者应该对于抽象类和接口有所了解，但可能会有这样的疑问抽象类和接口有什么区别？本节就回答这个问题。

归纳抽象类与接口区别如下：

01. 接口支持多继承，而抽象类（包括具体类）只能继承一个父类。

02. 接口中不能有实例成员变量，接口所声明的成员变量全部是静态常量，即便是变量不加public static final修饰符也是静态常量。抽象类与普通类一样各种形式的成员变量都可以声明。

03. 接口中没有包含构造方法，由于没有实例成员变量，也就不需要构造方法了。抽象类中可以有实例成员变量，也需要构造方法。

04. 抽象类中可以声明抽象方法和具体方法。Java 8之前接口中只有抽象方法，而Java 8之后接口中也可以声明具体方法，具体方法通过声明默认方法实现。

提示　学习了接口默认方法后，有些读者还会有这样的疑问，Java 8之后接口可以声明抽象方法和具体方法，这就相当于抽象类一样了吗？在多数情况下接口不能替代抽象类，例如当需要维护一个对象的信息和状态时只能使用抽象类，而接口不行，因为维护一个对象的信息和状态需要存储在实例成员变量中，而接口中不能声明实例成员变量。

本章小结

通过对本章的学习，读者可以了解抽象类和接口的概念，掌握如何声明抽象类和接口，如何实现抽象类和接口。了解Java 8之后的接口的新变化。熟悉抽象类和接口的区别。

第 **15** 章　枚举类

Java 5之前没有提供枚举类型，尽管可以通过声明静态常量（final static变量）替代枚举，但是仍然很多Java程序员期待能有类似其他语言中的枚举类型。Java 5之后提供了枚举类型，Java枚举类型本质上是一种继承java.lang.Enum类，是引用数据类型，因此也称为“枚举类”。本章介绍Java枚举类。

15.1 枚举概述

在C和Objective-C等其他语言中，枚举用来管理一组相关常量的集合，使用枚举可以提高程序的可读性，使代码更清晰且更易于维护。

在Java 5之前没有提供枚举类型，可以通过声明静态常量（final static变量）替代枚举常量，例如想声明一组常量表示一周中的5个工作日，那么Java 5之前实现代码如下：

```
//WeekDays.java文件
package com.a51work6;

public interface WeekDays {
    // 枚举常量列表
    int MONDAY          = 0;    //星期一
    int TUESDAY         = 1;    //星期二
    int WEDNESDAY       = 2;    //星期三
    int THURSDAY        = 3;    //星期四
    int FRIDAY          = 4;    //星期五
}
```

通常在接口中声明一组静态常量，当然也可以在一般类中声明一组静态常量。这些常量往往都是int类型，这是为了以后方便使用switch语句进行判断。

调用代码如下：

```
//HelloWorld.java文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        //day工作日变量
        int day = WeekDays.FRIDAY;            ①

        switch (day) {
        case WeekDays.MONDAY:
            System.out.println("星期一");
            break;
        case WeekDays.TUESDAY:
            System.out.println("星期二");
            break;
        case WeekDays.WEDNESDAY:
            System.out.println("星期三");
            break;
        case WeekDays.THURSDAY:
            System.out.println("星期四");
            break;
        case WeekDays.FRIDAY:
            System.out.println("星期五");
            break;
        }
    }
}
```

从上述代码中直接使用这些常量。但这种方式还存在一些问题：

01. 类型不安全。代码第①行是声明工作日变量day，day是整数类型，程序执行过程中很有可能给

day变量传入一个任意的整数值，可能导致程序出现错误。

02. 程序不方便调试。在程序调试时，如果通过日志输出day值，那么只能看到0~4之间的数值，程序员需要比较这些数值代表的含义，才能知道输出的结果是什么。

枚举类型可以避免直接使用常量所导致的问题。Java 5之后可以使用枚举类型了，Java中枚举类型的作用已经不仅仅是定义一组常量提高程序的可读性了，还具有如下特性：

01. Java枚举类型是一种类，是引用类型，具有了面向对象特性，可以添加方法和成员变量等。

02. Java枚举类型父类是java.lang.Enum，不需要显式声明。

03. Java枚举类型可以实现接口，与类实现接口类似。

04. Java枚举类型不能被继承，不存在子类。

15.2 枚举类声明

先来看Java中的枚举类声明。Java中是使用enum关键词声明枚举类，具体定义放在一对大括号内，枚举的语法格式如下：

```
[public] enum 枚举名 {
     枚举常量列表
}
```

enum前面的修饰符是[public]表示public或省略。public是公有访问级别，可以在任何地方访问。省略是默认访问级别，只能在当前包中访问。

“枚举名”是该枚举类的名称。它首先应该是有效的标识符，其次应该遵守Java命名规范。它应该是一个名称，如果采用英文单词命名，首字母应该大写，且应尽量用一个英文单词。“枚举常量列表”是枚举的核心，它由一组相关常量组成。

15.2.1 最简单形式的枚举类

如果采用枚举类来表示工作日，最简单枚举类WeekDays具体代码如下：

```
//WeekDays.java文件
package com.a51work6;

public enum WeekDays {
    // 枚举常量列表
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY
}
```

在枚举类WeekDays中定义了5个常量，使用枚举类WeekDays代码如下：

```
//HelloWorld.java文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {
        // day工作日变量
        WeekDays day = WeekDays.FRIDAY;          ①
        System.out.println(day);                 ②

        switch (day) {                           ③
        case MONDAY:                             ④
            System.out.println("星期一");
            break;
        case TUESDAY:
            System.out.println("星期二");
            break;
        case WEDNESDAY:
            System.out.println("星期三");
            break;
        case THURSDAY:
            System.out.println("星期四");
            break;
        default: //case FRIDAY:                   ⑤
            System.out.println("星期五");
        }
    }
```

```
}
```

输出结果：

```
FRIDAY
星期五
```

上述代码第①行是声明工作日变量day，day是WeekDays枚举类型，取值是WeekDays.FRIDAY，是枚举类中定义的枚举常量。day = WeekDays.FRIDAY赋值过程中实例化WeekDays枚举类对象，并初始化为WeekDays.FRIDAY。注意赋值表达式是“枚举类型名.枚举常量”的形式。

代码第②行day对象日志输出结果不是整数，而是FRIDAY。

枚举类与switch语句能够很好地配合使用，代码第③行switch表达式直接使用day枚举对象，case常量直接使用枚举常量，见代码第④行，而且不需要枚举类名作为前缀，使用起来比较简洁。

提示　在8.1.2节介绍过switch语句时，提到过switch表达式类型和case常量类型只能是int、byte、short和char类型，而Java 5之后还可以是枚举类型。另外，在switch中使用枚举类型时，switch语句中的case分支语句个数应该对应枚举常量个数，不要多也不要少，当使用default时，default应该只表示等于最后一个枚举常量情况。上述示例代码第⑤行是switch语句中使用default，default表示的是FRIDAY情况。

15.2.2 枚举类中成员变量和成员方法

枚举类可以像类一样包含成员变量和成员方法，成员变量可以是实例变量也可以是静态变量，成员方法可以是实例方法，也可以是静态方法，但不能是抽象方法。

示例代码如下：

```
//WeekDays.java文件
package com.a51work6;

public enum WeekDays {
    // 枚举常量列表
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY;        ①

    // 实例变量
    private String name;
    private int index;

    // 静态变量
    private static int staticVar = 100;

    // 覆盖父类中的toString()方法
    @Override
    public String toString() {                           ②
        StringBuilder sb = new StringBuilder();
        sb.append(name);
        sb.append('-');
        sb.append(index);
        return sb.toString();
    }

    // 实例方法
    public String getInfo() {
        // 调用父类中toString()方法
        return super.toString();
    }
```

```
    // 静态方法
    public static int getStaticVar() {
        return staticVar;
    }
}
```

上述代码第①行在枚举类WeekDays中添加了一些成员变量和成员方法，这些方法还可以覆盖枚举父类（java.lang.Enum）中的方法，见代码第②行的toString()方法。

添加的其他成员的枚举类需要注意，“枚举常量列表”语句必须是枚举类中的第一行代码。而且“枚举常量列表”语句后面要加分号（;）表示语句的结束，见代码第①行所示。

使用枚举类WeekDays代码如下：

```
//HelloWorld.java文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {
        // day工作日变量
        WeekDays day = WeekDays.FRIDAY;

        //打印day默认调用枚举toString()方法
        System.out.println(day);
        //调用枚举实例方法
        System.out.println(day.getInfo());
        //调用枚举静态方法
        System.out.println(WeekDays.getStaticVar());

    }
}
```

上述代码比较简单这里不再赘述。

15.2.3 枚举类构造方法

在15.2.2节示例中实例变量name和index，都是没有初始化，在类中成员变量的初始化是通过构造方法实现的，而在枚举类中也是通过构造方法初始化成员变量的。

为15.2.2节示例添加构造方法，代码如下：

```
//WeekDays.java文件
package com.a51work6;

public enum WeekDays {
    // 枚举常量列表
    MONDAY("星期一", 0),  TUESDAY("星期二", 1),  WEDNESDAY("星期三", 2),
            THURSDAY("星期四", 3),  FRIDAY("星期五", 4);            ①

    // 实例变量
    private String name;
    private int index;

    // 静态变量
    private static int staticVar = 100;

    private WeekDays(String name, int index) {                      ②
```

```
        this.name = name;
        this.index = index;
    }

    // 覆盖父类中的toString()方法
    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append(name);
        sb.append('-');
        sb.append(index);
        return sb.toString();
    }

    // 实例方法
    public String getInfo() {
        // 调用父类中toString()方法
        return super.toString();
    }

    // 静态方法
    public static int getStaticVar() {
        return staticVar;
    }
}
```

上述代码第②行是添加的构造方法，注意枚举类的中的构造方法只能是私有访问级别，构造方法可以省略private关键字，但它仍然是私有的构造方法。这也说明了枚举类不允许在外部创建对象。

提示　私有构造方法经常用于单例设计模式和工厂设计模式，使得不允许在类的外边直接调用构造方法创建对象。枚举类实现类似于工厂设计模式。

一旦添加了有参数的构造方法，那么“枚举常量列表”也需要修改，见代码第②行，每一个枚举常量都是一个实例，都会调用构造方法进行初始化成员变量，("星期一", 0)是调用构造方法。

15.3 枚举常用方法

所有枚举类都继承java.lang.Enum类，Enum中定义了一些枚举中常用的方法：

- int ordinal()：返回枚举常量的顺序。这个顺序根据枚举常量声明的顺序而定，顺序从零开始。
- 枚举类型[] values()：静态方法，返回一个包含全部枚举常量的数组。
- 枚举类型 valueOf(String str)：静态方法，str是枚举常量对应的字符串，返回一个包含枚举类型实例。

WeekDays枚举类代码如下：

```
//WeekDays.java文件
package com.a51work6;

public enum WeekDays {
    // 枚举常量列表
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY
}
```

使用枚举常用方法示例代码如下：

```
//HelloWorld.java文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        // 返回一个包含全部枚举常量的数组
        WeekDays[] allValues = WeekDays.values();                ①
        // 遍历枚举常量数值
        for (WeekDays value : allValues) {
            System.out.printf("%d - %s\n", value.ordinal(), value);    ②
        }

        // 创建WeekDays对象
        WeekDays day1 = WeekDays.FRIDAY;
        WeekDays day2 = WeekDays.valueOf("FRIDAY");              ③

        System.out.println(day1 == WeekDays.FRIDAY);             ④
        System.out.println(day1.equals(WeekDays.FRIDAY));        ⑤
        System.out.println(day1 == day2);                        ⑥

    }
}
```

上述代码第①行是通过values方法获得所有枚举常量的数组，代码第②行是获得枚举常量value，其中value.ordinal()获得当前枚举常量的顺序。

代码第③行是通过valueOf方法获得枚举对象WeekDays.FRIDAY，参数是枚举常量对应的字符串。

代码第④行~第⑥行是比较枚举对象，它们比较的结果都是true。

提示　在Java类引用类型进行比较时，有两种比较方法==和equals，==比较的是两个引用是否指

向同一个对象，equals是比较对象内容是否相同。但是，枚举引用类型中==和equals都是一样的，都是比较两个引用是否指向同一个实例，枚举类中每个枚举常量无论何时都只有一个实例。

本章小结

通过对本章的学习，读者可以了解到Java中枚举的作用、特点和常用方法。重点是声明和使用枚举类。

第 16 章　Java常用类

在Java SE中提供了众多丰富类和接口，其中很多类前面已经使用过了，如String、StringBuiler和StringBuffer等。由于数量众多，不书不可能一一介绍，也没有这个必要。本章归纳了Java中一些在日常开发过程中常用的类介绍一下，至于其他的不常用类读者可以自己查询Java SE API文档。

16.1 Java根类——Object

第一个应该介绍的常用类就是java.lang.Object类，它是Java所有类的根类，Java所有类都直接或间接继承自Object类，它是所有类的“祖先”。Object类属于java.lang包中的类型，不需要显示使用import语句引入，它是由解释器自动引入。

Object类有很多方法，常用的几个方法：

- String toString()：返回该对象的字符串表示。
- boolean equals(Object obj)：指示其他某个对象是否与此对象“相等”。

这些方法都是需要在子类中用来覆盖的，下面就详细解释一下它们的用法。

16.1.1 toString()方法

为了日志输出等处理方便，所有的对象都可以以文本方式表示，需要在该对象所在类中覆盖toString()方法。如果没有覆盖toString()方法，默认的字符串是“类名@对象的十六进制哈希码[1]”。

> [1]哈希码（hashCode），每个Java对象都有哈希码（hashCode）属性，哈希码可以用来标识对象，提高对象在集合操作中的执行效率。

下面看一个示例，在前面章节介绍过Person类，它的代码如下：

```
 //Person.java文件
package com.a51work6;

public class Person {

    String name;
    int age;

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

    @Override
    public String toString() {            ①
        return "Person [name=" + name + ", age=" + age + "]";
    }

}
```

上述代码第①行覆盖toString()方法，返回什么样的字符串完全是自定义的，只要是能够表示当前类和当前对象即可，本例是将Person成员变量拼接成为一个字符串。

调用代码如下：

```
//HelloWorld.java文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        Person person = new Person("Tony", 18);
```

```
        //打印过程自动调用person的 toString()方法
        System.out.println(person);
    }
}
```

运行输出结果如下；

```
Person [name=Tony, age=18]
```

使用System.out.println等输出语句可以自动调用对象的toString()方法将对象转换为字符串输出。读者可以测试一下，如果Person中没有覆盖toString()方法会是什么样子？它会输出类似如下的字符串：

```
com.a51work6.Person@15db9742
```

16.1.2 对象比较方法

在前面学习字符串比较的时，曾经介绍过有两种比较方法：==运算符和equals()方法，==运算符是比较两个引用变量是否指向同一个实例，equals()方法是比较两个对象的内容是否相等，通常字符串的比较，只是关心的内容是否相等。

事实上equals()方法是继承自Object的，所有对象都可以通过equals()方法比较，问题是比较的规则是什么，例如两个人（Person对象）相等是指什么？是名字？是年龄？问题的关键是需要指定相等的规则，就是要指定比较的是哪些属性相等，所以为了比较两个Person对象相等，则需要覆盖equals()方法，在该方法中指定比较规则。

修改Person代码如下：

```
 //Person.java文件
package com.a51work6;
public class Person {

    String name;
    int age;

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

    @Override
    public String toString() {
        return "Person [name=" + name + ", age=" + age + "]";
    }

    @Override
    public boolean equals(Object otherObject) {            ①

        //判断比较的参数也是Person类型
        if (otherObject instanceof Person) {               ②
            Person otherPerson = (Person) otherObject;     ③
            // 年龄作为比较规则
            if (this.age == otherPerson.age) {             ④
                return true;
            }
        }
        return false;
    }
```

```
}
```

上述代码第①行覆盖了equals()，为了防止传入的参数对象不是Person类型，则需要使用instanceof运算符判断一下，见代码第②行。如果是Person类型，通过代码第③行强制类型转换为Person。代码第④行进行比较，把年龄作为比较是否相等的规则，不管其他属性只要是年龄相等，则认为两个Person对象相等。

调用代码如下：

```
//HelloWorld.java文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        Person person1 = new Person("Tony", 20);
        Person person2 = new Person("Tom", 20);

        System.out.println(person1 == person2);         // false
        System.out.println(person1.equals(person2));    // true
    }
}
```

从上述代码中创建了两个Person对象，它们具有相关的年龄，这两个Person对象使用==比较结果是false，因为它们是两个不同的对象。使用equals()方法比较结果是true。

16.2 包装类

在Java中8种基本数据类型不属于类，不具备“对象”的特征，没有成员变量和方法，不方便进行面向对象的操作。为此，Java提供包装类（Wrapper Class）来将基本数据类型包装成类，每个Java基本数据类型在java.lang包中都有一个相应的包装类，每个包装类对象封装一个基本数据类型数值。对应关系如表16-1所示，除int和char类型外，其他的类型对应规则就是第一个字母大写。

表 **16-1** 基本数据类型与包装类对应关系

基本数据类型	包装类
boolean	Boolean
byte	Byte
char	Character
short	Short
int	Integer
long	Long
float	Float
double	Double

包装类都是final的，不能被继承。包装类都是不可变类，类似于String类，一旦创建了对象，其内容就不可以修改。包装类还可以分成三种不同类别：数值包装类（Byte、Short、Integer、Long、Float和Double）、Character和Boolean。下面分别详细介绍一下。

16.2.1 数值包装类

这些数值包装类（Byte、Short、Integer、Long、Float和Double）都有一些相同特点。

01. 构造方法类似

    每一个数值包装类都有两个构造方法，以Integer为例，Integer构造方法如下：

    - Integer(int value)：通过指定一个数值构造Integer对象。
    - Integer(String s)：通过指定一个字符串s构造对象，s是十进制字符串表示的数值。

02. 共同的父类

这6个数值包装类有一个共同的父类——Number，Number是一个抽象类，除了这6个子类还有：AtomicInteger、AtomicLong、BigDecimal和BigInteger，其中BigDecimal和BigInteger后面还会详细介绍。Number是抽象类，要求它的子类必须实现如下6个方法：

- byte byteValue()：将当前包装的对象转换为byte类型的数值。
- double doubleValue()：将当前包装的对象转换为double类型的数值。
- float floatValue()：将当前包装的对象转换为float类型的数值。
- int intValue()：将当前包装的对象转换为int类型的数值。
- long longValue()：将当前包装的对象转换为long类型的数值。
- short shortValue()：将当前包装的对象转换为short类型的数值。

通过这6个方法数值包装类可以互相转换这6种数值，但是需要注意的是大范围数值转换为小范围的数值，如果数值本身很大，可以会导致精度的丢失。

03. compareTo()方法

    每一个数值包装类都有int compareTo(数值包装类对象)方法，可以进行包装对象的比较。方法返回值是int，如果返回值是0，则相等；如果返回值小于0，则此对象小于参数对象；如果返回值大于0，则此对象大于参数对象。

04. 字符串转换为基本数据类型

    每一个数值包装类都提供一些静态parseXXX()方法将字符串转换为对应的基本数据类型，以Integer为例，方法定义如下：

    - static int parseInt(String s)：将字符串s转换有符号的十进制整数。
    - static int parseInt(String s, int radix)：将字符串s转换有符号的整数，radix是指定基数，基数用来指定进制。注意这种指定基数的方法在浮点数包装类（Double和Float）中没有的。

05. 基本数据类型转换为字符串

    每一个数值包装类都提供一些静态toString()方法实现将基本数据类型数值转换为字符串，以Integer为例，方法定义如下：

    - static String toString(int i)：将该整数i转换为有符号的十进制表示的字符串。
    - static String toString(int i, int radix)：将该整数i转换为有符号的特定进制表示的字符串，radix是基数可以指定进制。注意这种指定基数的方法在浮点数包装类（Double和Float）中没有的。

示例代码如下：

```
//HelloWorld.java文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        // 1.构造方法
```

```
        //创建数值为80的Integer对象
        Integer objInt = new Integer(80);
        //创建数值为80.0的Double对象
        Double objDouble = new Double(80.0);
        //通过"80.0"字符串创建数值为80.0的Float对象
        Float objFloat = new Float("80.0");
        //通过"80"字符串创建数值为80的Long对象
        Long objLong = new Long("80");

        // 2.Number类方法
        //Integer对象转换为long数值
        long longVar = objInt.longValue();
        //Double对象转换为int数值
        int intVar = objDouble.intValue();
        System.out.println("intVar = " + intVar);
        System.out.println("longVar = " + longVar);

        // 3.compareTo()方法
        Float objFloat2 = new Float(100);
        int result = objFloat.compareTo(objFloat2);
        // result = -1，表示objFloat小于objFloat2
        System.out.println(result);

        // 4.字符串转换为基本数据类型
        // 10进制"100"字符串转换为10进制数为100
        int intVar2 = Integer.parseInt("100");
        // 16进制"ABC"字符串转换为10进制数为2748
        int intVar3 = Integer.parseInt("ABC", 16);
        System.out.println("intVar2 = " + intVar2);
        System.out.println("intVar3 = " + intVar3);

        // 5.基本数据类型转换为字符串
        // 100转换为10进制字符串
        String str1 = Integer.toString(100);
        // 100转换为16进制字符串结果是64
        String str2 = Integer.toString(100, 16);
        System.out.println("str1 = " + str1);
        System.out.println("str2 = " + str2);

    }
}
```

代码中注释比较清楚，这里不再解释了。

16.2.2 Character类

Character类是char类型的包装类。Character类常用方法如下：

- Character(char value)：构造方法，通过char值创建一个新的Character对象。
- char charValue()：返回此Character对象的值。
- int compareTo(Character anotherCharacter)：方法返回值是int，如果返回值是0，则相等；如果返回值小于0，则此对象小于参数对象；如果返回值大于0，则此对象大于参数对象。

示例代码如下：

```
//HelloWorld.java文件
package com.a51work6;

public class HelloWorld {
```

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

        // 创建数值为'A'的Character对象
        Character objChar1 = new Character('A');
        // 从Character对象返回char值
        char ch = objChar1.charValue();

        // 字符比较
        Character objChar2 = new Character('C');
        int result = objChar1.compareTo(objChar2);
        // result = -2，表示objChar1小于objChar2
        if (result < 0) {
            System.out.println("objChar1小于objChar2");
        }
    }
}
```

16.2.3 Boolean类

Boolean类是boolean类型的包装类。

01. 构造方法

    Boolean类有两个构造方法，构造方法定义如下：

    - Boolean(boolean value)：通过一个boolean值创建Boolean对象。
    - Boolean(String s)：通过字符串创建Boolean对象。s不能为null，s如果是忽略大小写"true"则转换为true对象，其他字符串都转换为false对象。

02. compareTo()方法

    Boolean类有int compareTo(Boolean包装类对象)方法，可以进行包装对象的比较。方法返回值是int，如果返回值是0，则相等；如果返回值小于0，则此对象小于参数对象；如果返回值大于0，则此对象大于参数对象。

03. 字符串转换为boolean类型

    Boolean包装类都提供静态parseBoolean()方法实现将字符串转换为对应的boolean类型，方法定义如下：

    static boolean parseBoolean(String s)：将字符串转换为对应的boolean类。s不能为null，s如果是忽略大小写"true"则转换为true，其他字符串都转换为false。

示例代码如下：

```
//HelloWorld.java文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        // 创建数值为true的Character对象true
        Boolean obj1 = new Boolean(true);
        // 通过字符串"true"创建Character对象true
        Boolean obj2 = new Boolean("true");
        // 通过字符串"True"创建Character对象true
        Boolean obj3 = new Boolean("True");
```

```
        // 通过字符串"TRUE"创建Character对象true
        Boolean obj4 = new Boolean("TRUE");
        // 通过字符串"false"创建Character对象false
        Boolean obj5 = new Boolean("false");
        // 通过字符串"Yes"创建Character对象false
        Boolean obj6 = new Boolean("Yes");
        // 通过字符串"abc"创建Character对象false
        Boolean obj7 = new Boolean("abc");


        boolean b1 = Boolean.parseBoolean("true");
        boolean b2 = Boolean.parseBoolean("True");
        boolean b3 = Boolean.parseBoolean("TRUE");
        boolean b4 = Boolean.parseBoolean("false");
        boolean b5 = Boolean.parseBoolean("Yes");
        boolean b6 = Boolean.parseBoolean("abc");
        ...
    }
}
```

16.2.4 自动装箱/拆箱

有了包装类丰富了Java语言面向对象，提供了原来基本数据类型没有方法。但是另外也带来使用的不便。例如如下代码试图对包装类对象进行算数运算，在Java 5之前代码第①行会发生编译错误，想想可以理解，这些对象不能使用简单使用算数运算符连接起来。

```
//创建数值为80的Integer对象
Integer objInt = new Integer(80);
//创建数值为80.0的Double对象
Double objDouble = new Double(80.0);
//算数运算
double sum = objInt + objDouble; //Java 5之前有编译错误        ①
```

但是代码第①行在Java 5之后可以编译通过了，并能计算出正确的结果。这是因为Java 5之后提供了拆箱(unboxing)功能，拆箱能够将包装类对象自动转换为基本数据类型的数值，而不需要使用intValue()或doubleValue()等方法。类似Java 5还提供了相反功能，自动装箱(autoboxing)，装箱能够自动地将基本数据类型的数值自动转换为包装类对象，而不需要使用构造方法。

示例代码如下：

```
//HelloWorld.java文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        Integer objInt = new Integer(80);
        Double objDouble = new Double(80.0);
        //自动拆箱
        double sum = objInt + objDouble;

        //自动装箱
        //自动装箱'C'转换为Character对象
        Character objChar = 'C';
        //自动装箱true转换为Boolean对象
        Boolean objBoolean = true;
        //自动装箱80.0f转换为Float对象
        Float objFloat = 80.0f;
```

```
        //自动装箱100转换为Integer对象
        display(100);

        //避免出现下面的情况
        Integer obj = null;                                          ①
        int intVar = obj;    //运行期异常NullPointerException        ②

    }

    /**
     * @param objInt Integer对象
     * @return int数值
     */
    public static int display(Integer objInt) {

        System.out.println(objInt);

        //return objInt.intValue();
        //自动拆箱Integer对象转换为int
        return objInt;
    }
}
```

在自动装箱和拆箱时，要避免空对象，代码第①行obj是null，则代码第②行会发生运行期NullPointerException异常，这是因为拆箱的过程本质上是调用intValue()方法实现的，试图访问空对象的方法和成员变量，就会抛出运行期NullPointerException异常。

16.3 Math类

Java语言是彻底地面向对象语言，哪怕是进行数学运算也封装到一个类中的，这个类是java.lang.Math，Math类是final的不能被继承。Math类中包含用于进行基本数学运算的方法，如指数、对数、平方根和三角函数等。这些方法分类如下：

01. 舍入方法

    - static double ceil(double a)：返回大于或等于a最小整数。
    - static double floor(double a)：返回小于或等于a最大整数。
    - static int round(float a)：四舍五入方法。

02. 最大值和最小值

    - static int min(int a, int b)：取两个int整数中较小的一个整数。
    - static int min(long a, long b)：取两个long整数中较小的一个整数。
    - static int min(float a, float b)：取两个float浮点数中较小的一个浮点数。
    - static int min(double a, double b)：取两个double浮点数中较小的一个浮点数。

    max方法取两个数中较大的一个数，max方法与min方法参数类似也有4个版本，这里不再赘述。

03. 绝对值

    - static int abs(int a)：取int整数a的绝对值。
    - static long abs(long a)：取long整数a的绝对值。
    - static float abs(float a)：取float浮点数a的绝对值。
    - static double abs(double a)：取double浮点数a的绝对值。

04. 三角函数：

    - static double sin(double a)：返回角的三角正弦。
    - static double cos(double a)：返回角的三角余弦。
    - static double tan(double a)：返回角的三角正切。
    - static double asin(double a)：返回一个值的反正弦。
    - static double acos(double a)：返回一个值的反余弦。
    - static double atan(double a)：返回一个值的反正切。
    - static double toDegrees(double angrad)：将弧度转换为角度。
    - static double toRadians(double angdeg)：将角度转换为弧度。

05. 对数运算：static double log(double a)，返回a的自然对数。

06. 平方根：static double sqrt(double a)，返回a的正平方根。

07. 幂运算：static double pow(double a, double b)，返回第一个参数的第二个参数次幂的值。

08. 计算随机值：static double random()，返回大于等于 0.0 且小于 1.0随机数。

09. 常量

    - 圆周率PI
    - 自然对数的底数E。

示例代码如下：

```
//HelloWorld.java文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        double[] nums = { 1.4, 1.5, 1.6 };

        // 测试最大值和最小值
        System.out.printf("min(%.1f, %.1f) = %.1f\n", nums[1], nums[2], Math.min(nums[1], nums[2]));
        System.out.printf("max(%.1f, %.1f) = %.1f\n", nums[1], nums[2], Math.max(nums[1], nums[2]));
        System.out.println();

        // 测试三角函数
        // 1π弧度 = 180°
        System.out.printf("toDegrees(0.5π)    = %f\n", Math.toDegrees(0.5 * Math.PI));
        System.out.printf("toRadians(180/π) = %f\n", Math.toRadians(180 / Math.PI));
        System.out.println();

        // 测试平方根
        System.out.printf("sqrt(%.1f) = %f\n", nums[2], Math.sqrt(nums[2]));
        System.out.println();

        // 测试幂运算
        System.out.printf("pow(8, 3) = %f\n", Math.pow(8, 3));
        System.out.println();

        // 测试计算随机值
        System.out.printf("0.0~1.0之间的随机数 = %f\n", Math.random());
        System.out.println();

        // 测试舍入
        for (double num : nums) {
            display(num);
        }

    }

    // 测试舍入方法
    public static void display(double n) {
        System.out.printf("ceil(%.1f)    = %.1f\n", n, Math.ceil(n));
        System.out.printf("floor(%.1f)     = %.1f\n", n, Math.floor(n));
        System.out.printf("round(%.1f)     = %d\n", n, Math.round(n));
        System.out.println();
    }
}
```

运行结果如下：

```
min(1.5, 1.6) = 1.5
max(1.5, 1.6) = 1.6

toDegrees(0.5π)    = 90.000000
toRadians(180/π) = 1.000000

sqrt(1.6) = 1.264911

pow(8, 3) = 512.000000

0.0~1.0之间的随机数 = 0.881115

ceil(1.4)    = 2.0
floor(1.4)     = 1.0
round(1.4)     = 1

ceil(1.5)    = 2.0
floor(1.5)     = 1.0
round(1.5)     = 2

ceil(1.6)    = 2.0
floor(1.6)     = 1.0
round(1.6)     = 2
```

上述代码比较简单，这里不再赘述。

16.4 大数值

对货币等大值数据进行计算时，int、long、float和double等基本数据类型已经在精度方面不能满足需求了。为此Java提高了两个大数值类：BigInteger和BigDecimal，这里两个类都继承自Number抽象类。

16.4.1 BigInteger

java.math.BigInteger是不可变的任意精度的大整数。BigInteger构造方法有很多，其中字符串参数的构造方法有两个：

- BigInteger(String val)：将十进制字符串val转换为BigInteger对象。
- BigInteger(String val, int radix)：按照指定基数radix将字符串val转换为BigInteger对象。

BigInteger提供多种方法，下面列举几个常用的方法：

- int compareTo(BigInteger val)：将当前对象与参数val进行比较，方法返回值是int，如果返回值是0，则相等；如果返回值小于0，则此对象小于参数对象；如果返回值大于0，则此对象大于参数对象。
- BigInteger add(BigInteger val)：加运算，当前对象数值加参数val。
- BigInteger subtract(BigInteger val)：减运算，当前对象数值减参数val。
- BigInteger multiply(BigInteger val)：乘运算，当前对象数值乘参数val。
- BigInteger divide(BigInteger val)：除运算，当前对象数值除以参数val。

另外，BigInteger继承了抽象类Number，那么它还有实现抽象类Number的6个方法，具体方法参考6.2.1节。

示例代码如下：

```
//HelloWorld.java文件
package com.a51work6;

import java.math.BigInteger;

public class HelloWorld {

    public static void main(String[] args) {

        //创建BigInteger，字符串表示10进制数值
        BigInteger number1 = new BigInteger("999999999999");
        //创建BigInteger，字符串表示16进制数值
        BigInteger number2 = new BigInteger("567800000", 16);

        // 加法操作
        System.out.println("加法操作：" + number1.add(number2));
        // 减法操作
        System.out.println("减法操作：" + number1.subtract(number2));
        // 乘法操作
        System.out.println("乘法操作：" + number1.multiply(number2));
        // 除法操作
        System.out.println("除法操作：" + number1.divide(number2));
    }
}
```

运行结果如下：

```
加法操作：1023211278335
减法操作：976788721663
乘法操作：23211278335976788721664
除法操作：43
```

上述代码比较简单，这里不再赘述。

16.4.2 BigDecimal

java.math.BigDecimal是不可变的任意精度的有符号十进制数。BigDecimal构造方法有很多：

- BigDecimal(BigInteger val)：将BigInteger对象val转换为BigDecimal对象。
- BigDecimal(double val)：将double转换为BigDecimal对象，参数val是double类型的二进制浮点值准确的十进制表示形式。
- BigDecimal(int val)：将int转换为BigDecimal对象。
- BigDecimal(long val)：将long转换为BigDecimal对象。
- BigDecimal(String val)：将字符串表示数值形式转换为BigDecimal对象。

BigDecimal提供多种方法，下面列举几个常用的方法：

- int compareTo(BigDecimal val)：将当前对象与参数val进行比较，方法返回值是int，如果返回值是0，则相等；如果返回值小于0，则此对象小于参数对象；如果返回值大于0，则此对象大于参数对象。
- BigDecimal add(BigDecimal val)：加运算，当前对象数值加参数val。
- BigDecimal subtract(BigDecimal val)：减运算，当前对象数值减参数val。
- BigDecimal multiply(BigDecimal val)：乘运算，当前对象数值乘参数val。
- BigDecimal divide(BigDecimal val)：除运算，当前对象数值除以参数val。
- BigDecimal divide(BigDecimal val, int roundingMode)：除运算，当前对象数值除以参数val。roundingMode要应用的舍入模式。

另外，BigDecimal继承了抽象类Number，那么它还实现抽象类Number的6个方法，具体方法参考16.2.1节。

示例代码如下：

```
//HelloWorld.java文件
package com.a51work6;

import java.math.BigDecimal;

public class HelloWorld {

    public static void main(String[] args) {

        // 创建BigDecimal，通过字符参数串创建
        BigDecimal number1 = new BigDecimal("999999999.99988888");
```

```
        // 创建BigDecimal，通过double参数创建
        BigDecimal number2 = new BigDecimal(567800000.888888);

        // 加法操作
        System.out.println("加法操作：" + number1.add(number2));
        // 减法操作
        System.out.println("减法操作：" + number1.subtract(number2));
        // 乘法操作
        System.out.println("乘法操作：" + number1.multiply(number2));
        // 除法操作
        System.out.println("除法操作："
                    + number1.divide(number2, BigDecimal.ROUND_HALF_UP));        ①
    }
}
```

运行结果如下：

```
加法操作：1567800000.88877688144195556640625
减法操作：432199999.11100087855804443359375
乘法操作：567800000888824907.505856793171529769897460937500
除法操作：1.76118351
```

上述代码第①行是进行除法运算，该方法需要指定舍入模式，如果不指定舍入模式那么会发生运行期异常ArithmeticException，舍入模式BigDecimal.ROUND_HALF_UP是四舍五入。

16.5 日期时间相关类

Java 8之前日期类是java.util.Date，Date类比较古老，其中的很多方法现在已经废弃了，但是目前仍然有很多程序还在使用Date类，考虑到还有很多程序使用Date类，因此本节还是介绍一下Date类及日期时间相关类的使用。

此外，Java 8之前与日期时间相关类还有DateFormat、Calendar和TimeZone，DateFormat用于日期格式化，Calendar日历类，TimeZone是时区类。

提示　在Java SE核心类中有两个Date，分别是java.util.Date和java.sql.Date。java.util.Date就是本节要介绍的日期时间类，而java.sql.Date是JDBC中日期字段类型。

16.5.1 Date类

Date类中有很多构造方法和普通方法。首先看看Date类构造方法：

- Date()：用当前时间创建Date对象，精确到毫秒。
- Date(long date)：指定标准基准时间以来的毫秒数创建Date对象。标准基准时间是格林威治时间1970年1月1日00:00:00。

Date类的普通方法：

- boolean after(Date when)：测试此日期是否在when日期之后。
- boolean before(Date when)：测试此日期是否在when日期之前。
- int compareTo(Date anotherDate)：比较两个日期的顺序。如果参数日期等于此日期，则返回值0；如果此日期在参数日期之前，则返回小于0的值；如果此日期在参数日期之后，则返回大于0的值。
- long getTime()：返回自1970年1月1日00:00:00以来此Date对象表示的毫秒数。
- void setTime(long time)：用毫秒数time设置日期对象，time是自1970年1月1日00:00:00以来此Date对象表示的毫秒数。

示例代码如下：

```
//HelloWorld.java文件
package com.a51work6;

import java.util.Date;

public class HelloWorld {

    public static void main(String[] args) {

        Date now = new Date();                              ①
        System.out.println("now = " + now);                 ②
        System.out.println("now.getTime() = " + now.getTime());
        System.out.println();

        Date date = new Date(1234567890123L);               ③
        System.out.println("date = " + date);

        // 测试now和date日期
        display(now, date);                                 ④
```

```
        // 重新设置日期time
        date.setTime(9999999999999L);                    ⑤

        System.out.println("修改之后的date = " + date);

        // 重新测试now和date日期
        display(now, date);                              ⑥

    }

    // 测试after、before和compareTo方法
    public static void display(Date now, Date date) {
        System.out.println();
        System.out.println("now.after(date)     = " + now.after(date));
        System.out.println("now.before(date)    = " + now.before(date));
        System.out.println("now.compareTo(date)    = " + now.compareTo(date));
        System.out.println();
    }
}
```

运行结果如下：

```
now = Sun Jun 04 10:03:09 CST 2017
now.getTime() = 1496541789730

date = Sat Feb 14 07:31:30 CST 2009

now.after(date)     = true
now.before(date)    = false
now.compareTo(date)    = 1

修改之后的date = Sun Nov 21 01:46:39 CST 2286

now.after(date)     = false
now.before(date)    = true
now.compareTo(date)    = -1
```

上述代码①行是创建当前日期对象，代码第②行是打印输出当前日期对象，从输出结果可见是Sun Jun 04 10:03:09 CST 2017，其中CST是美国中部标准时间。

代码第③行通过long整数1234567890123L创建日期对象，打印输出date日期是Sat Feb 14 07:31:30 CST 2009。代码第⑤行又重新设置了time，之后打印输出date日期是Sun Nov 21 01:46:39 CST 2286。

代码第④行和第⑥行两次调用display方法测试after、before和compareTo方法。

16.5.2 日期格式化和解析

上一节示例日期输出结果，如Sun Jun 04 10:03:09 CST 2017，这个时间不符合中国人的习惯，这需要对日期进行格式化输出。日期格式化类是java.text.DateFormat，DateFormat是抽象类，它的常用具体类是java.text.SimpleDateFormat。

DateFormat中提供日期格式化和日期解析方法，具体方法说明如下：

- String format(Date date)：将一个Date格式化为日期/时间字符串。
- Date parse(String source)：从给定字符串的开始解析文本，以生成一个日期对象。如果解析失败则抛出ParseException。

另外，具体类是SimpleDateFormat构造方法如下：

- SimpleDateFormat()：用默认的模式和默认语言环境的日期格式符号构造SimpleDateFormat。
- SimpleDateFormat(String pattern)：用给定的模式和默认语言环境的日期格式符号构造SimpleDateFormat。pattern参数是日期和时间格式模式，表16-2所示是常用的日期和时间格式模式。

表 **16-2**　常用的日期和时间格式模式

字母	日期或时间元素
y	年
M	年中的月份
D	年中的天数
d	月份中的天数
H	一天中的小时数（0-23）
h	AM/PM 中的小时数（1-12）
a	AM/PM 标记
m	小时中的分钟数
s	分钟中的秒数
S	毫秒数
Z	时区

示例代码如下：

```
//HelloWorld.java文件
package com.a51work6;

import java.text.DateFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

public class HelloWorld {

    public static void main(String[] args) throws ParseException {  ①
```

```
        Date date = new Date(1234567890123L);                          ②
        System.out.println("格式化前date = " + date);

        DateFormat df = new SimpleDateFormat();                        ③
        System.out.println("格式化后date = " + df.format(date));     ④
        df = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");              ⑤
        System.out.println("格式化后date = " + df.format(date));     ⑥

        String dateString = "2018-08-18 08:18:58";
        Date date1 = df.parse(dateString);                             ⑦
        System.out.println("从字符串获得日期对象 = " + date1);
    }
}
```

运行结果如下：

```
格式化前date = Sat Feb 14 07:31:30 CST 2009
格式化后date = 09-2-14 上午7:31
格式化后date = 2009-02-14 07:31:30
从字符串获得日期对象 = Sat Aug 18 08:18:58 CST 2018
```

上述代码第②行创建日期对象，代码第③行采用默认构造方法创建日期格式化SimpleDateFormat对象，代码第④行进行格式化输出，结果是“09-2-14 上午7:31”，这个格式化采用的当前操作系统默认格式，在实际开发时用的不多。代码第⑤行重新创建SimpleDateFormat对象，这里指定它的日期时间格式模式"yyyy-MM-dd HH:mm:ss"，代码第⑥行是格式化输出，结果是“2009-02-14 07:31:30”，开发人员还可以根据自己的需要指定设置其他格式。

日期格式化，一方面可以将日期对象转换为特定格式的字符串；另一方面可以将特定格式的字符串转换为日期对象。代码第⑦行是将字符串"2018-08-18 08:18:58"转换为日期对象。

注意　并不是所有的字符串都能够转换为日期，如果转换失败parse方法会抛出异常ParseException。由于ParseException异常是受检查类型异常，这种异常必须处理，本例是抛出处理见代码第①行main方法后的throws ParseException语句。目前读者只需了解异常这样处理就可以了，异常将在第19章详细说明。

16.5.3 Calendar类

有时为了取得更多的日期时间信息，或对日期时间进行操作，可以使用java.util.Calendar类，Calendar是一个抽象类，不能实例化，但是通过静态工厂方法getInstance()获得Calendar实例。

Calendar类的主要方法：

- static Calendar getInstance()：使用默认时区和语言环境获得一个日历。
- void set(int field, int value)：将给定的日历字段设置为给定值。
- void set(int year,int month,int date)：设置日历字段YEAR、MONTH和DAY_OF_MONTH的值。
- Date getTime()：返回一个表示此Calendar时间值（从1970年1月1日00:00:00至现在的毫秒数）的Date对象。
- boolean after(Object when)：判断此Calendar表示的时间是否在指定时间之后，返回判断结果。
- boolean before(Object when)：判断此Calendar表示的时间是否在指定时间之前，返回判断结果。

- int compareTo(Calendar anotherCalendar)：比较两个Calendar对象表示的时间值。

日历示例代码如下：

```
//HelloWorld.java文件
package com.a51work6;

import java.text.DateFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.Date;

public class HelloWorld {

    public static void main(String[] args) throws ParseException {

        // 获得默认的日历对象
        Calendar calendar = Calendar.getInstance();
        // 设置日期2018年8月18日
        calendar.set(2018, 7, 18);                          ①

        // 通过日历获得Date对象
        Date date = calendar.getTime();                     ②
        System.out.println("格式化前date = " + date);
        DateFormat df = new SimpleDateFormat("yyyy-MM-dd");
        System.out.println("格式化后date = " + df.format(date));
        System.out.println();

        calendar.clear();                                   ③
        // 设置日期2018年8月28日
        calendar.set(Calendar.YEAR, 2018);                  ④
        calendar.set(Calendar.MONTH, 7);                    ⑤
        calendar.set(Calendar.DATE, 28);                    ⑥

        // 通过日历获得Date对象
        date = calendar.getTime();
        System.out.println("格式化前date = " + date);
        System.out.println("格式化后date = " + df.format(date));
    }
}
```

运行结果如下：

```
格式化前date = Sat Aug 18 14:47:22 CST 2018
格式化后date = 2018-08-18

格式化前date = Tue Aug 28 00:00:00 CST 2018
格式化后date = 2018-08-28
```

上述代码第①行是设置日历的年、月和日字段，注意在设置“月”时，应该是“月份-1”，因为日历中的月份中第一个月是0，第二个月是1，依次类推那么本例中设置8月份，则实际参数应该为7。代码第②行是通过日历获得日期对象。

代码第③行calendar.clear()语句是重新初始化日历对象。代码第④~⑥行分别设置日历的年、月和日字段。

16.6 Java 8新日期时间相关类

Java 8之后提供了新的日期时间相关类、接口和枚举，这些类型内容非常多，令人生畏。但是使用起来非常方便。

16.6.1 时间和日期

Java 8之后提供了新的日期时间类有三个：LocalDate、LocalTime和LocalDateTime，它们都位于java.time包中，LocalDate表示一个不可变的日期对象；LocalTime表示一个不可变的时间对象；LocalDateTime表示一个不可变的日期和时间。

这三个类有类似的方法，首先先看看创建日期时间对象相关方法，这三个类并没有提供公有的构造方法，创建它们对象可以使用静态工厂方法，主要有now()和of()方法。

now()方法说明如下：

- static LocalDate now()：LocalDate静态工厂方法，该方法使用默认时区获得当前日期，返回LocalDate对象。
- static LocalTime now()：LocalTime静态工厂方法，该方法使用默认时区获得当前时间，返回LocalTime对象。
- static LocalDateTime now()：LocalDateTime静态工厂方法，该方法使用默认时区获得当前日期时间，返回LocalDateTime对象。

of()方法有很多重载方法，说明如下：

- static LocalDateTime of(int year, int month, int dayOfMonth, int hour, int minute, int second)：按照指定的年、月、日、小时、分钟和秒获得LocalDateTime实例，将纳秒设置为零。
- static LocalTime of(int hour, int minute, int second)：按照指定的小时、分钟和秒获取一个LocalTime实例。
- static LocalDate of(int year, int month, int dayOfMonth)：按照指定的年、月和日获得一个LocalDate实例，日期中年、月和日必须有效，否则将抛出异常。

上述方法中的参数取值范围如表16-3所示。

表 **16-3** 参数取值范围

参数	说明
year	从-999,999,999 到 999,999,999 的年份。
month	一年中的月份，从 1 至 12。
dayOfMonth	月中的天，从 1 到 31。
hour	从 0 到 23 表示的小时。
minute	从 0 到 59 表示的分钟。
second	从 0 到 59 表示的秒。

示例代码如下：

```
//HelloWorld.java文件
package com.a51work6;

import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;

public class HelloWorld {

    public static void main(String[] args) {

        // 使用now方法获得LocalDate对象
        LocalDate date1 = LocalDate.now();
        System.out.println("date1 = " + date1);

        // 使用of方法获得LocalDate对象2018-08-18
        LocalDate date2 = LocalDate.of(2018, 8, 18);
        System.out.println("date2 = " + date2);

        // 使用now方法获得LocalTime对象
        LocalTime time1 = LocalTime.now();
        System.out.println("time1 = " + time1);

        // 使用of方法获得LocalTime对象08:58:18
        LocalTime time2 = LocalTime.of(8, 58, 18);
        System.out.println("time2 = " + time2);

        // 使用now方法获得LocalDateTime对象
        LocalDateTime dateTime1 = LocalDateTime.now();
        System.out.println("dateTime1 = " + dateTime1);

        // 使用of方法获得LocalDateTime对象2018-08-18T08:58:18
        LocalDateTime dateTime2 = LocalDateTime.of(2018, 8, 18, 8, 58, 18);
        System.out.println("dateTime2 = " + dateTime2);
    }

}
```

运行结果如下：

```
date1 = 2017-06-04
date2 = 2018-08-18
time1 = 17:41:15.073
time2 = 08:58:18
dateTime1 = 2017-06-04T17:41:15.073
dateTime2 = 2018-08-18T08:58:18
```

从运行结果可见，日期时间输出都是本地格式。上述代码比较简单，这里不再赘述。

16.6.2 日期格式化和解析

Java 8提供的日期格式化类是java.time.format.DateTimeFormatter，DateTimeFormatter中本身没有提供日期格式化和日期解析方法，这些方法还是由LocalDate、LocalTime和LocalDateTime提供的。

01. 日期格式化

    日期格式化方法是format，这三个类每一个都有String format(DateTimeFormatter formatter)，参数formatter是DateTimeFormatter类型。

02. 日期解析

日期解析方法是parse，这三个类每一个都有两个版本的parse方法，具体说明如下：

- static LocalDateTime parse(CharSequence text)：使用默认格式，从一个文本字符串获取一个LocalDateTime实例，如2007-12-03T10:15:30。
- static LocalDateTime parse(CharSequence text, DateTimeFormatter formatter)：使用指定格式化，从文本字符串获取LocalDateTime实例。
- static LocalDate parse(CharSequence text)：使用默认格式，从一个文本字符串获取一个LocalDate实例，如2007-12-03。
- static LocalDate parse(CharSequence text, DateTimeFormatter formatter)：使用指定格式化，从文本字符串获取LocalDate实例。
- static LocalTime parse(CharSequence text)：使用默认格式，从一个文本字符串获取一个LocalTime实例。
- static LocalTime parse(CharSequence text, DateTimeFormatter formatter)：使用指定的格式化，从文本字符串获取LocalTime实例。

示例代码如下：

```
//HelloWorld.java文件
package com.a51work6;

import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.format.DateTimeFormatter;

public class HelloWorld {

    public static void main(String[] args) {

        //// 创建LocalDateTime对象
        LocalDateTime dateTime = LocalDateTime.now();
        System.out.println("dateTime格式化之前：" + dateTime);

        // 设置格式化类
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
        String text = dateTime.format(formatter);
        System.out.println("dateTime格式化之后：" + text);

        // 格式化字符串"2018-08-18 08:58:18"，返回LocalDateTime对象
        LocalDateTime parsedDateTime = LocalDateTime.parse("2018-08-18 08:58:18", formatter);
        System.out.println("LocalDateTime解析之后：" + parsedDateTime);

        //// 创建LocalDate对象
        LocalDate date = LocalDate.now();
        System.out.println("date格式化之前：" + date);

        // 重新设置格式化类
        formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd");
        text = date.format(formatter);
        System.out.println("date格式化之后：" + text);

        // 格式化字符串"2018-08-18"，返回LocalDate对象
        LocalDate parsedDate = LocalDate.parse("2018-08-18", formatter);
        System.out.println("LocalDate解析之后：" + parsedDate);
```

```
        //// 创建LocalTime对象
        LocalTime time = LocalTime.now();
        System.out.println("time格式化之前：" + time);

        // 重新设置格式化类
        formatter = DateTimeFormatter.ofPattern("HH:mm:ss");              ①
        text = time.format(formatter);
        System.out.println("time格式化之后：" + text);

        // 格式化字符串"08:58:18"，返回LocalTime对象
        LocalTime parsedTime = LocalTime.parse("08:58:18", formatter);    ②
        System.out.println("LocalTime解析之后：" + parsedTime);
    }
}
```

运行结果如下：

```
dateTime格式化之前：2017-06-04T18:22:25.874
dateTime格式化之后：2017-06-04 18:22:25
LocalDateTime解析之后：2018-08-18T08:58:18
date格式化之前：2017-06-04
date格式化之后：2017-06-04
LocalDate解析之后：2018-08-18
time格式化之前：18:22:25.891
time格式化之后：18:22:25
LocalTime解析之后：08:58:18
```

上述代码中格式化类DateTimeFormatter对象是通过ofPattern(String pattern)获得，其中pattern是日期和时间格式模式，具体说明参考表16-2。

注意　解析时间日期字符串时候需要注意两方面的问题：第一、要解析的字符串格式一定要与格式模式匹配，假设将代码第②的时间字符串"08:58:18"改为"08 58 18"，那么程序运行会抛出异常DateTimeParseException；第二、要解析的字符串格式一定是有效的时间或日期，假设将代码第②的时间字符串"08:58:18"改为"08:58:68"，那么程序运行页会抛出异常DateTimeParseException。

本章小结

通过对本章的学习，读者可以学习到Object类、包装类、Math类、BigInteger类和BigDecimal类。最后，还介绍了旧版本日期时间类和Java 8之后的日期时间类。

第 17 章　内部类

Java中还有一种内部类技术，简单说就是在一个类的内部定义一个类。内部类看起来很简单，但是当你深入其中，你会发现它是极其复杂的。事实上Java应用程序开发过程中内部类使用的地方不是很多，一般在图形用户界面开发中用于事件处理。

提示　内部类技术虽然使程序结构变得紧凑，但是却在一定程度上破坏了Java面向对象思想。

17.1 内部类概述

Java语言中允许在一个类（或方法、代码块）的内部定义另一个类，后者称为“内部类”（Inner Classes），也称为“嵌套类”（Nested Classes），封装它的类称为“外部类”。内部类与外部类之间存在逻辑上的隶属关系，内部类一般只用在封装它的外部类或代码块中使用。

17.1.1 内部类的作用

内部类的作用如下：

01. 封装。将不想公开的实现细节封装到一个内部类中，内部类可以声明为私有的，只能在所在外部类中访问。

02. 提供命名空间。静态内部类和外部类能够提供有别于包的命名空间。

03. 便于访问外部类成员。内部类能够很方便访问所在外部类的成员，包括私有成员也能访问。

17.1.2 内部类的分类

内部类的分类如图17-1所示，按照内部类在定义的时候是否给它一个类名，可以分为：有名内部类和匿名内部类。有名内部类又按照作用域不同可以分为：局部内部类和成员内部类，成员内部类又分为：实例内部类和静态内部类。

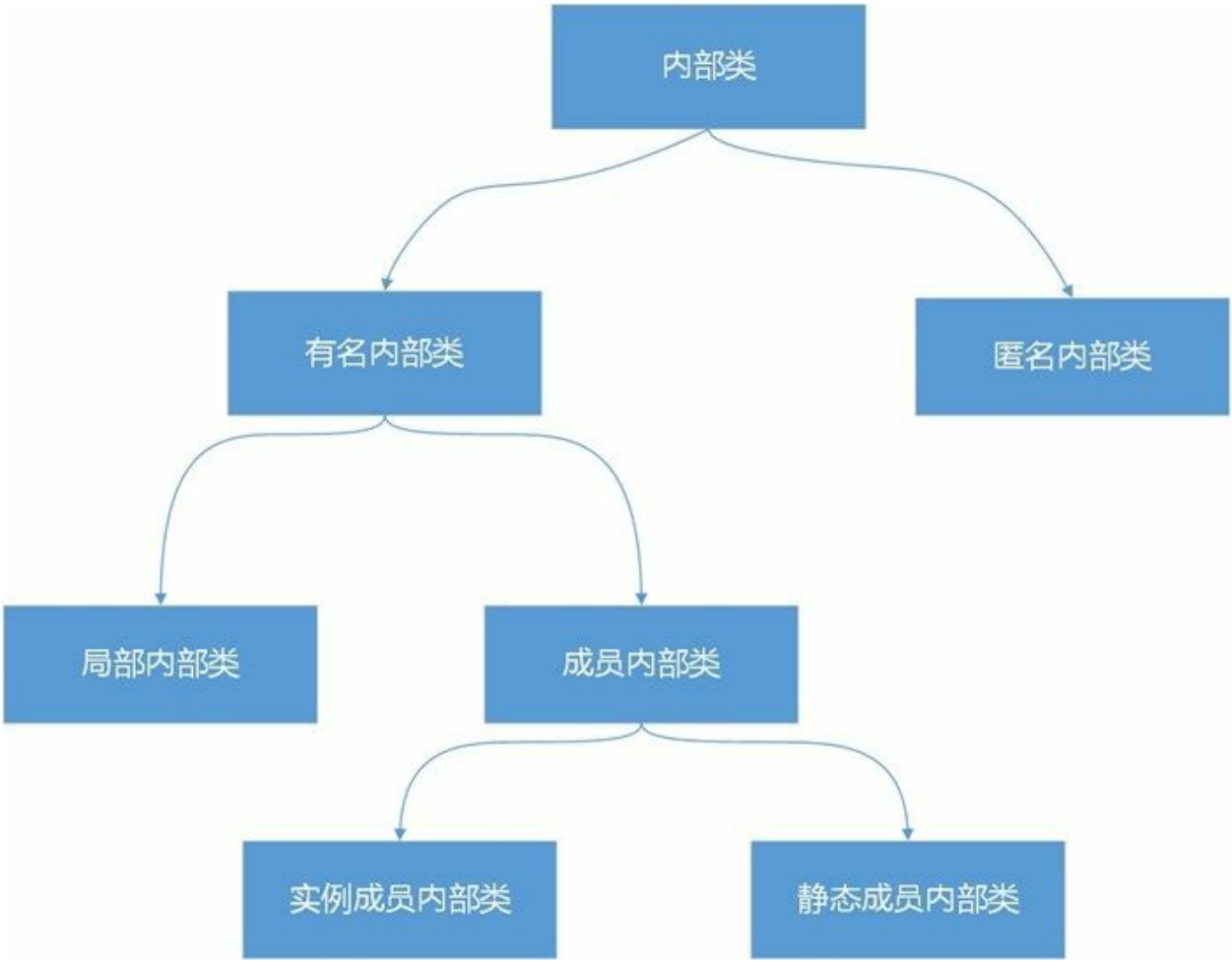


图**17-1**　内部类的分类

17.2 成员内部类

成员内部类类似于外部类的成员变量，在外边类的内部，且方法体和代码块之外定义的内部类。

17.2.1 实例内部类

实例内部类与实例变量类似，可以声明为公有级别、私有级别、默认级别或保护级别，即4种访问级别都可以，而外部类只能声明为公有或默认级别。

实例内部类示例代码如下：

```
//Outer.java文件
package com.a51work6;

//外部类
public class Outer {

    // 外部类成员变量
    private int x = 10;

    // 外部类方法
    private void print() {
        System.out.println("调用外部方法...");
    }

    // 测试调用内部类
    public void test() {
        Inner inner = new Inner();
        inner.display();
    }

    // 内部类
    class Inner {                          ①

        // 内部类成员变量
        private int x = 5;                 ②

        // 内部类方法
        void display() {                   ③

            // 访问外部类的成员变量x
            System.out.println("外部类成员变量 x = " + Outer.this.x);      ④
            // 访问内部类的成员变量x
            System.out.println("内部类成员变量 x = " + this.x);            ⑤
            System.out.println("内部类成员变量 x = " + x);                 ⑥

            // 调用外部类的成员方法
            Outer.this.print();            ⑦
            print();                       ⑧
        }
    }
}
```

上述代码第①行声明了内部类Inner，它的访问级别是默认，这里还可以是public、private和protected。内部类Inner有一个成员变量x和成员方法display()。在display()方法中代码第④行是访问外部类的x成员变量，代码第⑤行和第⑥行一样都是访问内部类的x成员变量。代码第⑦行和第⑧行都是访问外部类的print()成员方法。

提示　在内部类中this是引用当前内部类对象，见代码第⑤行。而要引用外部类对象需要使用“外

部类名.this”，见代码第④行。另外，如果内部类和外部类它们的成员命名没有冲突情况下，在引用外部类成员时可以不用加“外部类名.this”，如代码第⑧行的print()方法只有外部类中定义，所以可以省略Outer.this。

测试内部HelloWorld代码如下：

```
//HelloWorld.java文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        // 通过外部类访问内部类
        Outer outer = new Outer();
        outer.test();                                ①

        System.out.println("-------直接访问内部类------");
        // 直接访问内部类
        Outer.Inner inner =  outer.new Inner();     ②
        inner.display();                             ③
    }
}
```

运行结果如下：

```
外部类成员变量 x = 10
内部类成员变量 x = 5
内部类成员变量 x = 5
调用外部方法...
调用外部方法...
-------直接访问内部类------
外部类成员变量 x = 10
内部类成员变量 x = 5
内部类成员变量 x = 5
调用外部方法...
调用外部方法...
```

通常情况下，使用实例成员内部类不是给外部类之外调用使用的，而就是给外部类自己使用的。但是一定要在外部类的之外访问内部类，Java语言也是支持的，见代码第②行内部类的类型表示“外部类.内部类”，实例化过程是先实例化外部类，再实例化内部类，outer对象是外部类实例，outer.new Inner()表达式实例化内部类对象。 另外，HelloWorld与内部类Inner在同一个包中，内部类Inner和它的方法display()访问级别都是默认的，它们对于在同一包中HelloWorld是可见的。

提示　内部类编译成功后生成的字节码文件是“外部类$内部类.class”。

17.2.2 静态内部类

静态内部类与静态变量类似，在声明的时候使用关键字static修饰，静态内部类只能访问外部类静态成员，所以静态内部类使用的场景不多。但可以提供有别于包的命名空间。

示例代码如下：

```
//View.java文件
package com.a51work6;

//外部类
public class View {
```

```
    // 外部类实例变量
    private int x = 20;                 ①
    // 外部类静态变量
    private static int staticX = 10;    ②

    // 静态内部类
    static class Button {               ③

        // 内部类方法
        void onClick() {                ④
            //访问外部类的静态成员
            System.out.println(staticX);            ⑤
            //不能访问外部类的非静态成员
            // System.out.println(x); //编译错误     ⑥
        }
    }
}
```

上述代码第③行定义了静态内部类Button，在静态内部类中可以访问外部类的静态成员，见代码第⑤行。但是不能访问非静态成员，见代码第⑥行试图访问外部类的x实例变量，会发生编译错误。

测试内部HelloWorld代码如下：

```
//HelloWorld.java文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        // 直接访问内部类
        View.Button button = new View.Button();
        button.onClick();
    }
}
```

从代码View.Button button = new View.Button()可见，在声明静态内部时采用“内部类.静态内部类”形式，实例化也是如此形式。

提示　如果不看代码或文档，View.Button形式看起来像是View包中的Button类，事实上它是View类中静态内部类Button。View.Button形式客观上能够提供有别于包的命名空间，View相关的类集中管理起来，View.Button可以防止命名冲突。

17.3 局部内部类

局部内部类就是在方法体或代码块中定义的内部类，局部内部类的作用域仅限于方法体或代码块中。局部内部类访问级别只能是默认的，不能是公有的、私有的和保护的访问级别，即不能使用public、private和protected修饰。局部内部类也不能是静态，即不能使用static修饰。局部内部类可以访问外部类所有成员。

示例代码如下：

```
//Outer.java文件
package com.a51work6;

//外部类
public class Outer {

    // 外部类成员变量
    private int value = 10;

    // 外部类方法
    public void add(final int x, int y) {     ①
        //局部变量
        int z = 100;

        // 定义内部类
        class Inner {                         ②
            // 内部类方法
            void display() {
                int sum = x + z + value;      ③
                System.out.println("sum = " + sum);
            }
        }

        // Inner inner = new Inner();
        // inner.display();
        //声明匿名对象
        new Inner().display();                ④
    }
}
```

上述代码在add方法中定义了局部内部类，见代码第②行，在内部类中代码第③行访问了外部类成员变量value、方法参数x和方法局部变量z，其中方法参数应该声明为final的，见代码第①行x参数是final的。

> 提示　代码第④行new Inner().display()是实例化Inner对象后马上调用它的方法，没有为Inner对象分配一个引用变量名，这种写法称为“匿名对象”。匿名对象适合只运行一次情况下。匿名对象写法使代码变得简洁，但是给初学者阅读代码带来了难度。

测试内部HelloWorld代码如下：

```
//HelloWorld.java文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        Outer outer = new Outer();
        outer.add(100, 300);
    }
```

```
}
```

17.4 匿名内部类

匿名内部类是没有名字的内部类，本质上是没有名的局部内部类，具有局部内部类所有特征。例如：可以访问外部类所有成员。如果匿名内部类在方法中定义，它所访问的参数需要声明为final的。

下面通过示例介绍一下匿名内部类。有如下一个View类：

```
//View.java文件
package com.a51work6;

//外部类
public class View {

    public void handler(OnClickListener listener) {        ①
        listener.onClick();
    }
}
```

代码第①行中handler方法需要一个实现OnClickListener接口的参数。OnClickListener接口代码如下：

```
//OnClickListener.java文件
package com.a51work6;

public interface OnClickListener {
    void onClick();
}
```

接口中只有一个onClick()方法。使用匿名内部类的示例代码如下：

```
//HelloWorld.java文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        View v = new View();                              ①
        // 方法参数是匿名内部类
        v.handler(new OnClickListener() {                 ②

            @Override
            public void onClick() {
                System.out.println("实现接口的匿名内部类...");
            }

        });

        //继承类的匿名内部类
        Figure f = new Figure() {                         ③
            @Override
            public void onDraw() {
                System.out.println("继承类的匿名内部类...");
            }
        };

        //具体类作为内部类
        Person person = new Person("Tony", 18) {          ④
            @Override
```

```
            public String toString() {
                return "匿名内部类.实现 "
                        + " Person[name=" + name
                        + ", age=" + age + "]";
            }
        };

        //打印过程自动调用person的 toString()方法
        System.out.println(person);

    }
}
```

在HelloWorld的main方法中，代码第①行是实例化View对象，代码第②行是调用它的handler方法，该方法需要一个实现OnClickListener接口的参数，new OnClickListener() {… }表达式是实际参数，它就是匿名内部类。表达式中OnClickListener是要实现的接口或要继承的类，new是为匿名内部类创建对象，()是调用构造方法，{… }是类体部分。

代码第③行是在赋值时候使用匿名内部类，其中Figure是14.1.2节使用过的抽象类。匿名内部类实现了它的抽象方法onDraw()。

代码第④行也是在赋值时候使用匿名内部类，其中Person是13.4.2节使用过的具体类，匿名内部类覆盖了toString()方法。它有个两个参数的构造方法，匿名内部类使用了这个构造方法。

匿名内部类通常用来实现接口或抽象类的，很少覆盖具体类。

本章小结

通过对本章的学习，读者了解了内部类的概念，熟悉了内部类非划分，如何编写内部类。

第 18 章　Java 8函数式编程基础——Lambda表达式

Java 8之后推出的Lambda表达式开启了Java语言支持函数式编程[1]（Functional Programming）新时代。Lambda表达式，也称为闭包（Closure），现在很多语言都支持Lambda表达式，如C++、C#、Swift、Objective-C和JavaScript等。为什么Lambda表达式这怎么受欢迎，这是因为Lambda表达式是实现支持函数式编程技术基础。

> [1]函数式编程是种编程范式，它将计算机运算视为函数的计算。函数编程语言最重要的基础是λ演算（lambda calculus）。而且λ演算的函数可以接受函数当作输入（参数）和输出（返回值）。和指令式编程相比，函数式编程强调函数的计算比指令的执行重要。和过程化编程相比，函数式编程里，函数的计算可随时调用。——引自于百度百科 http://baike.baidu.com/item/函数式编程

提示　函数式编程与面向对象编程有很大的差别，函数式编程将程序代码看作数学中的函数，函数本身作为另一个函数的参数或返回值，即高阶函数。而面向对象编程是按照真实世界客观事物的自然规律进行分析，客观世界中存在什么样的实体，构建的软件系统就存在什么样的实体。即便Java 8之后提供了对函数式编程的支持，但是Java还是以面向对象为主的语言，函数式编程只是对Java语言的补充。

18.1 Lambda表达式概述

与其他语言相比Java语言的Lambda表达式有着明显的区别。本节介绍Java语言中的Lambda表达式概念和具体实现方法。

18.1.1 从一个示例开始

为了理解Lambda表达式的概念，下面先从一个示例开始。

假设有这样的一个需求：设计一个通用方法，能够实现两个数值的加法和减法运算。Java中方法不能单独存在，必须定义在类或接口中，考虑是一个通用方法，可以设计一个数值计算接口，其中定义该通用方法，代码如下：

```
//Calculable.java文件
package com.a51work6;

//可计算接口
public interface Calculable {
    // 计算两个int数值
    int calculateInt(int a, int b);
}
```

Calculable接口只有一个方法calculateInt，参数是两个int类型，返回值也是int类型。通过方法如下：

```
//HelloWorld.java文件
…
 /**
  * 通过操作符，进行计算
  * @param opr 操作符
  * @return 实现Calculable接口对象
  */
public static Calculable calculate(char opr) {

    Calculable result;

    if (opr == '+') {
        // 匿名内部类实现Calculable接口
        result = new Calculable() {                              ①
            // 实现加法运算
            @Override
            public int calculateInt(int a, int b) {              ②
                return a + b;
            }
        };
    } else {
        // 匿名内部类实现Calculable接口
        result = new Calculable() {                              ③
            // 实现减法运算
            @Override
            public int calculateInt(int a, int b) {              ④
                return a - b;
            }
        };
    }

    return result;
}
```

通用方法calculate中的参数opr是运算符，返回值是实现Calculable接口对象。代码第①行和第③行都采用匿名内部类实现Calculable接口。代码第②行实现加法运算。代码第④行实现减法运算。

调用通用方法代码如下：

```
//HelloWorld.java文件
…
public static void main(String[] args) {

    int n1 = 10;
    int n2 = 5;

    // 实现加法计算Calculable对象
    Calculable f1 = calculate('+');                  ①
    // 实现减法计算Calculable对象
    Calculable f2 = calculate('-');                  ②

    // 调用calculateInt方法进行加法计算
System.out.printf("%d + %d = %d \n", n1, n2,
            f1.calculateInt(n1, n2));                ③
    // 调用calculateInt方法进行减法计算
System.out.printf("%d - %d = %d \n", n1, n2,
            f2.calculateInt(n1, n2));                ④
}
```

代码第①行中f1是实现加法计算Calculable对象，代码第②行中f2是实现减法计算Calculable对象。代码第③行和第④行才进行方法调用。

18.1.2 Lambda表达式实现

18.1.1节通过匿名内部类实现通用方法calculate代码很臃肿，Java 8采用Lambda表达式可以替代匿名内部类。修改之后的通用方法calculate代码如下：

```
// HelloWorld.java文件
…
/**
 * 通过操作符，进行计算
 * @param opr 操作符
 * @return 实现Calculable接口对象
 */
public static Calculable calculate(char opr) {

    Calculable result;

    if (opr == '+') {
        // Lambda表达式实现Calculable接口
        result = (int a, int b) -> {                 ①
            return a + b;
        };
    } else {
        // Lambda表达式实现Calculable接口
        result = (int a, int b) -> {                 ②
            return a - b;
        };
    }

    return result;
}
```

代码第①行和第②行用Lambda表达式替代匿名内部类，可见代码变得简洁。

通过以上示例的演变，可以给Lambda表达式一个定义：Lambda表达式是一个匿名函数（方法）代码块，可以作为表达式、方法参数和方法返回值。

Lambda表达式标准语法形式如下：

```
(参数列表) -> {
    //Lambda表达式体
}
```

其中，Lambda表达式参数列表与接口中方法参数列表形式一样，Lambda表达式体实现接口方法。

18.1.3 函数式接口

Lambda表达式实现的接口不是普通的接口，称为是函数式接口，这种接口只能有一个方法。如果接口中声明多个抽象方法，那么Lambda表达式会发生编译错误：

```
The target type of this expression must be a functional interface
```

这说明该接口不是函数式接口，为了防止在函数式接口中声明多个抽象方法，Java 8提供了一个声明函数式接口注解@FunctionalInterface，示例代码如下。

```
//Calculable.java文件
package com.a51work6;

//可计算接口
@FunctionalInterface
public interface Calculable {
    // 计算两个int数值
    int calculateInt(int a, int b);
}
```

在接口之前使用@FunctionalInterface注解修饰，那么试图增加一个抽象方法时会发生编译错误。但可以添加默认方法和静态方法。

提示　Lambda表达式是一个匿名方法代码，Java中的方法必须声明在类或接口中，那么Lambda表达式所实现的匿名方法是在函数式接口中声明的。

18.2 Lambda表达式简化形式

使用Lambda表达式是为了简化程序代码，Lambda表达式本身也提供了多种简化形式，这些简化形式虽然简化了代码，但客观上使得代码可读性变差。本节介绍Lambda表达式本几种简化形式。

18.2.1 省略参数类型

Lambda表达式可以根据上下文环境推断出参数类型。calculate方法中Lambda表达式能推断出参数a和b是int类型，简化形式如下：

```
public static Calculable calculate(char opr) {

    Calculable result;

    if (opr == '+') {
        // Lambda表达式实现Calculable接口
        result = (a, b) -> {                    ①
            return a + b;
        };
    } else {
        // Lambda表达式实现Calculable接口
        result = (a, b) -> {                    ②
            return a - b;
        };
    }

    return result;
}
```

上述代码第①行和第②行的Lambda表达式是上一节示例的简化写法，其中a和b是参数。怎么样？很简单吧？

18.2.2 省略参数小括号

Lambda表达式中参数只有一个时，可以省略参数小括号。修改Calculable接口，代码如下。

```
//Calculable.java文件
package com.a51work6;

//可计算接口
@FunctionalInterface
public interface Calculable {
    // 计算一个int数值
    int calculateInt(int a);
}
```

其中calculateInt方法只有一个int类型参数，返回值也是int类型。调用代码如下：

```
//HelloWorld.java文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        int n1 = 10;
```

```
        // 实现二次方计算Calculable对象
        Calculable f1 = calculate(2);
        // 实现三次方计算Calculable对象
        Calculable f2 = calculate(3);

        // 调用calculateInt方法进行加法计算
        System.out.printf("%d二次方 = %d \n", n1, f1.calculateInt(n1));
        // 调用calculateInt方法进行减法计算
        System.out.printf("%d三次方 = %d \n", n1, f2.calculateInt(n1));
    }

    /**
     * 通过幂计算
     * @param power 幂
     * @return 实现Calculable接口对象
     */
    public static Calculable calculate(int power) {

        Calculable result;

        if (power == 2) {
            // Lambda表达式实现Calculable接口
            result = (int a) -> {    //标准形式            ①
                return a * a;
            };
        } else {
            // Lambda表达式实现Calculable接口
            result = a -> {        //省略形式              ②
                return  a * a * a;
            };
        }
        return result;
    }
}
```

上述代码第①行和第②行都是实现Calculable接口的Lambda表达式。代码第①行是标准形式没有任何的简化。代码第②行省略了参数类型和小括号。

18.2.3 省略**return**和大括号

如果Lambda表达式体中只有一条语句，那么可以省略return和大括号，代码如下：

```
public static Calculable calculate(int power) {

    Calculable result;

    if (power == 2) {
        // Lambda表达式实现Calculable接口
        result = (int a) -> {    //标准形式
            return a * a;
        };
    } else {
        // Lambda表达式实现Calculable接口
        result = a -> a * a * a; //省略形式            ①
    }
    return result;
}
```

上述代码第①行是省略了return和大括号，这是最简化形式的Lambda表达式了，代码太简洁了，但是对于初学者而言很难理解这个表达式。

18.3 作为参数使用Lambda表达式

Lambda表达式一种常见的用途是作为参数传递给方法。这需要声明参数的类型声明为函数式接口类型。

示例代码如下：

```
//HelloWorld.java文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        int n1 = 10;
        int n2 = 5;

        // 打印计算结果加法计算结果
        display((a, b) -> {
            return a + b;
        }, n1, n2);                                  ①

        // 打印计算结果减法计算结果
        display((a, b) -> a - b, n1, n2);            ②

    }

    /**
     * 打印计算结果
     *
     * @param calc Lambda表达式
     * @param n1 操作数1
     * @param n2 操作数2
     */
    public static void display(Calculable calc, int n1, int n2) {        ③
        System.out.println(calc.calculateInt(n1, n2));
    }

}
```

上述代码第③行定义display打印计算结果方法，其中参数calc类型是Calculable，这个参数即可以接收实现Calculable接口的对象，也可以接收Lambda表达式，因为Calculable是函数式接口。

代码第①行和第②行两次调用display方法，它们第一个参数都是Lambda表达式。

18.4 访问变量

Lambda表达式可以访问所在外层作用域内定义的变量，包括：成员变量和局部变量。

18.4.1 访问成员变量

成员变量包括：实例成员变量和静态成员变量。在Lambda表达式中可以访问这些成员变量，此时的Lambda表达式与普通方法一样，可以读取成员变量，也可以修改成员变量。

示例代码如下：

```
//LambdaDemo.java文件
package com.a51work6;

public class LambdaDemo {
    // 实例成员变量
    private int value = 10;
    // 静态成员变量
    private static int staticValue = 5;

    // 静态方法，进行加法运算
    public static Calculable add() {                          ①

        Calculable result = (int a, int b) -> {               ②
            // 访问静态成员变量，不能访问实例成员变量
            staticValue++;
            int c = a + b + staticValue; // this.value;
            return c;
        };

        return result;
    }

    // 实例方法，进行减法运算
    public Calculable sub() {                                 ③

        Calculable result = (int a, int b) -> {               ④
            // 访问静态成员变量和实例成员变量
            staticValue++;
            this.value++;
            int c = a - b - staticValue - this.value;
            return c;
        };
        return result;
    }
}
```

LambdaDemo类中声明一个实例成员变量value和一个静态成员变量staticValue。此外，还声明了静态方法add（见代码第①行）和实例方法sub（见代码第③行）。add方法是静态方法，静态方法中不能访问实例成员变量，所以代码第②行的Lambda表达式中也不能访问实例成员变量，也不能访问实例成员方法。sub方法是实例方法，实例方法中能够访问静态成员变量和实例成员变量，所以代码第④行的Lambda表达式中可以访问这些变量，当然实例方法和静态方法也可以访问，当访问实例成员变量或实例方法时可以使用this，如果不与局部变量发生冲突情况下可以省略this。

18.4.2 捕获局部变量

对于成员变量的访问Lambda表达式与普通方法没有区别，但是对于访问外层局部变量时，会发生“捕获变量”情况。Lambda表达式中捕获变量时，会将变量当成final的，在Lambda表达式中不能修改那些

捕获的变量。

示例代码如下：

```
//LambdaDemo.java文件
package com.a51work6;

public class LambdaDemo {
    // 实例成员变量
    private int value = 10;
    // 静态成员变量
    private static int staticValue = 5;

    // 静态方法，进行加法运算
    public static Calculable add() {
        //局部变量
        int localValue = 20;                                ①

        Calculable result = (int a, int b) -> {
            // localValue++; //编译错误                     ②
            int c = a + b + localValue;                     ③
            return c;
        };
        return result;
    }

    // 实例方法，进行减法运算
    public Calculable sub() {
        //final局部变量
        final int localValue = 20;                          ④

        Calculable result = (int a, int b) -> {
            int c = a - b - staticValue - this.value;       ⑤
            // localValue = c; //编译错误                   ⑥
            return c;
        };
        return result;
    }
}
```

上述代码第①行和第④行都声明一个局部变量localValue，Lambda表达式中捕获这个变量，见代码第③行和第⑤行。不管这个变量是否显式地使用final修饰，它都不能在Lambda表达式中修改变量，所以代码第②行和第⑥行如果去掉注释会发生编译错误。

18.5 方法引用

Java 8之后增加了双冒号“::”运算符，该运算符用于“方法引用”，注意不是调用方法。“方法引用”虽然没有直接使用Lambda表达式，但也与Lambda表达式有关，与函数式接口有关。

方法引用分为：静态方法的方法引用和实例方法的方法引用。它们的语法形式如下：

```
类型名::静态方法          // 静态方法的方法引用
实例名::实例方法          // 实例方法的方法引用
```

注意　被引用方法的参数列表和返回值类型，必须与函数式接口方法参数列表和方法返回值类型一致。

示例代码如下：

```
//LambdaDemo.java文件
package com.a51work6;

public class LambdaDemo {

    // 静态方法，进行加法运算
    // 参数列表要与函数式接口方法calculateInt(int a, int b)兼容
    public static int add(int a, int b) {
        return a + b;
    }

    // 实例方法，进行减法运算
    // 参数列表要与函数式接口方法calculateInt(int a, int b)兼容
    public int sub(int a, int b) {
        return a - b;
    }
}
```

LambdaDemo类中提供了一个静态方法add，一个实例方法sub。这两个方法必须与函数式接口方法参数列表一致，方法返回值类型也要保持一致。

调用代码如下：

```
//HelloWorld.java文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        int n1 = 10;
        int n2 = 5;

        // 打印计算结果加法计算结果
        display(LambdaDemo::add, n1, n2);                    ①

        LambdaDemo d = new LambdaDemo();
        // 打印计算结果减法计算结果
        display(d::sub, n1, n2);                             ②

    }
```

```
    /**
     * 打印计算结果
     * @param calc Lambda表达式
     * @param n1  操作数1
     * @param n2 操作数2
     */
    public static void display(Calculable calc, int n1, int n2) {        ③
        System.out.println(calc.calculateInt(n1, n2));
    }
}
```

代码第③行声明display方法，第一个参数calc是Calculable类型，它可以接受三种对象：Calculable实现对象、Lambda表达式和方法引用。代码第①行中第一个实际参数LambdaDemo::add是静态方法的方法引用。代码第②行中第一个实际参数d::sub,是实例方法的方法引用，d是LambdaDemo实例。

提示　代码第①行的LambdaDemo::add和第②行的d::sub中Lambda是方法引用，此时并没有调用方法，只是将引用传递给display方法，在display方法中才真正调用方法。

本章小结

本章介绍了Lambda表达式，读者需要了解为什么使用Lambda表达式，Lambda表达式的优点是什么。掌握Lambda表达式标准语法，了解Lambda表达式的几个简写方式。熟悉Lambda表达式作为参数使用的场景，了解方法引用。

第 19 章　异常处理

很多事件并非总是按照人们自己设计意愿顺利发展的，而是有能够出现这样那样的异常情况。例如：你计划周末郊游，你的计划会安排满满的，你计划可能是这样的：从家里出发→到达目的→游泳→烧烤→回家。但天有不测风云，当前你准备烧烤时候天降大雨，你只能终止郊游提前回家。“天降大雨”是一种异常情况，你的计划应该考虑到这样情况，并且应该有处理这种异常的预案。

为增强程序的健壮性，计算机程序的编写也需要考虑处理这些异常情况，Java语言提供了异常处理功能，本章介绍Java异常处理机制。

19.1 从一个问题开始

为了学习Java异常处理机制，首先看看下面程序。

```
//HelloWorld.java文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {
        int a = 0;
        System.out.println(5 / a);
    }
}
```

这个程序没有编译错误，但会发生如下的运行时错误：

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at com.a51work6.HelloWorld.main(HelloWorld.java:9)
```

在数学上除数不能为0，所以程序运行时表达式（5 / a）会抛出ArithmeticException异常，ArithmeticException是数学计算异常，凡是发生数学计算错误都会抛出该异常。

程序运行过程中难免会发生异常，发生异常并不可怕，程序员应该考虑到有可能发生这些异常，编程时应该捕获并进行处理异常，不能让程序发生终止，这就是健壮的程序。

19.2 异常类继承层次

异常封装成为类Exception，此外，还有Throwable和Error类，异常类继承层次如图19-1所示。

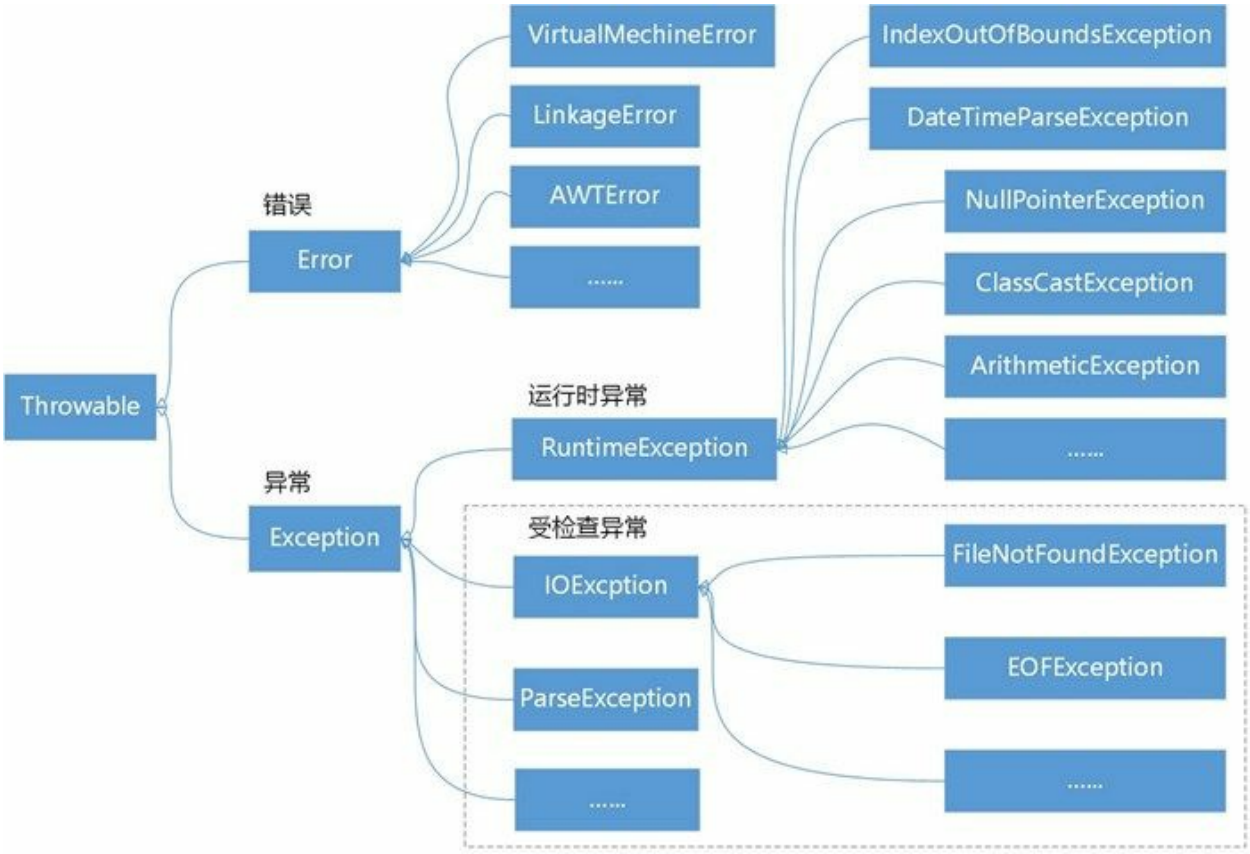


图**19-1　Java**异常类继承层次

19.2.1　Throwable类

从图19-1可见，所有的异常类都直接或间接地继承于java.lang.Throwable类，在Throwable类有几个非常重要的方法：

- String getMessage()：获得发生异常的详细消息。
- void printStackTrace()：打印异常堆栈跟踪信息。
- String toString()：获得异常对象的描述。

提示　堆栈跟踪是方法调用过程的轨迹，它包含了程序执行过程中方法调用的顺序和所在源代码行号。

为了介绍Throwable类的使用，下面修改19.1节的示例代码如下：

```
//HelloWorld.java文件
package com.a51work6;

public class HelloWorld {
```

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

        int a = 0;
        int result = divide(5, a);
        System.out.printf("divide(%d, %d) = %d", 5, a, result);
    }

    public static int divide(int number, int divisor) {

        try {
            return number / divisor;
        } catch (Throwable throwable) {                                          ①

            System.out.println("getMessage() : " + throwable.getMessage());    ②

            System.out.println("toString() : " + throwable.toString());        ③

            System.out.println("printStackTrace()输出信息如下：");
            throwable.printStackTrace();                                         ④
        }

        return 0;
    }
}
```

运行结果如下：

```
getMessage() : / by zero
toString() : java.lang.ArithmeticException: / by zero
printStackTrace()输出信息如下：
java.lang.ArithmeticException: / by zero
    at com.a51work6.HelloWorld.divide(HelloWorld.java:17)
    at com.a51work6.HelloWorld.main(HelloWorld.java:10)
divide(5, 0) = 0
```

将可以发生异常的语句System.out.println(5 / a)放到try-catch代码块中，称为捕获异常，有关捕获异常的相关知识会在下一节详细介绍。在catch中有一个Throwable对象throwable，throwable对象是系统在程序发生异常时创建，通过throwable对象可以调用Throwable中定义的方法。

代码第②行是调用getMessage()方法获得异常消息，输出结果是“/ by zero”。代码第③行是调用toString()方法获得异常对象的描述，输出结果是java.lang.ArithmeticException: / by zero。代码第④行是调用printStackTrace()方法打印异常堆栈跟踪信息。

提示　堆栈跟踪信息从下往上，是方法调用的顺序。首先JVM调用是com.a51work6.HelloWorld类的main方法，接着在HelloWorld.java源代码第10行调用com.a51work6.HelloWorld类的divide方法，在HelloWorld.java源代码第17行发生了异常，最后输出的是异常信息。

19.2.2　Error和Exception

从图19-1可见，Throwable有两个直接子类：Error和Exception。

01. Error

    Error是程序无法恢复的严重错误，程序员根本无能为力，只能让程序终止。例如：JVM内部错误、内存溢出和资源耗尽等严重情况。

02. Exception

Exception是程序可以恢复的异常，它是程序员所能掌控的。例如：除零异常、空指针访问、网络连接中断和读取不存在的文件等。本章所讨论的异常处理就是对Exception及其子类的异常处理。

19.2.3 受检查异常和运行时异常

从图19-1可见，Exception类可以分为：受检查异常和运行时异常。

01. 受检查异常

    如图19-1所示，受检查异常是除RuntimeException以外的异常类。它们的共同特点是：编译器会检查这类异常是否进行了处理，即要么捕获（try-catch语句），要么不抛出（通过在方法后声明throws），否则会发生编译错误。它们种类很多，前面遇到过的日期解析异常ParseException。

02. 运行时异常

    运行时异常是继承RuntimeException类的直接或间接子类。运行时异常往往是程序员所犯错误导致的，健壮的程序不应该发生运行时异常。它们的共同特点是：编译器不检查这类异常是否进行了处理，也就是对于这类异常不捕获也不抛出，程序也可以编译通过。由于没有进行异常处理，一旦运行时异常发生就会导致程序的终止，这是用户不希望看到的。由于19.2.1节除零示例的ArithmeticException异常属于RuntimeException异常，见图19-1所示，可以不用加try-catch语句捕获异常。

提示　对于运行时异常通常不采用抛出或捕获处理方式，而是应该提前预判，防止这种发生异常，做到未雨绸缪。例如19.2.1节除零示例，在进行除法运算之前应该判断除数是非零的，修改示例代码如下，从代码可见提前预判这样处理要比通过try-catch捕获异常要友好的多。

```
//HelloWorld.java文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        int a = 0;
        int result = divide(5, a);
        System.out.printf("divide(%d, %d) = %d", 5, a, result);
    }

    public static int divide(int number, int divisor) {

        //判断除数divisor非零，防止运行时异常
        if (divisor != 0) {
            return number / divisor;
        }
        return 0;
    }

}
```

除了图19-1所示异常，还有很多异常，本书不能一一穷尽，随着学习的深入会介绍一些常用的异常，其他异常读者可以自己查询API文档。

19.3 捕获异常

在学习本内容之前，你先考虑一下，在现实生活中是如何对待领导交给你的任务呢？当然无非是两种：自己有能解决的自己处理；自己无力解决的反馈给领导，让领导自己处理。

那么对待受检查异常亦是如此。当前方法有能力解决，则捕获异常进行处理；没有能力解决，则抛出给上层调用方法处理。如果上层调用方法还无力解决，则继续抛给它的上层调用方法，异常就是这样向上传递直到有方法处理它，如果所有的方法都没有处理该异常，那么JVM会终止程序运行。

这一节先介绍一下捕获异常。

19.3.1 try-catch语句

捕获异常是通过try-catch语句实现的，最基本try-catch语句语法如下：

```
try{
    //可能会发生异常的语句
} catch(Throwable e){
    //处理异常e
}
```

01. try代码块

    try代码块中应该包含执行过程中可能会发生异常的语句。一条语句是否有可能发生异常，这要看语句中调用的方法。例如日期格式化类DateFormat的日期解析方法parse()，该方法的完整定义如下：

    ```
    public Date parse(String source) throws ParseException
    ```

    方法后面的throws ParseException说明：当调用parse()方法时有可以能产生ParseException异常。

    提示　静态方法、实例方法和构造方法都可以声明抛出异常，凡是抛出异常的方法都可以通过try-catch进行捕获，当然运行时异常可以不捕获。一个方法声明抛出什么样的异常需要查询API文档。

02. catch代码块

    每个try代码块可以伴随一个或多个catch代码块，用于处理try代码块中所可能发生的多种异常。catch(Throwable e)语句中的e是捕获异常对象，e必须是Throwable的子类，异常对象e的作用域在该catch代码块中。

    下面看看一个try-catch示例：

    ```
    //HelloWorld.java文件
    package com.a51work6;

    import java.text.DateFormat;
    import java.text.ParseException;
    import java.text.SimpleDateFormat;
    import java.util.Date;

    public class HelloWorld {

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

```
        Date date = readDate();
        System.out.println("日期  = " + date);
    }

    // 解析日期
    public static Date readDate() {                 ①

        try {
            String str = "2018-8-18";     //"201A-18-18"
            DateFormat df = new SimpleDateFormat("yyyy-MM-dd");
            // 从字符串中解析日期
            Date date = df.parse(str);              ②
            return date;
        } catch (ParseException e) {                ③
            System.out.println("处理ParseException…");
            e.printStackTrace();                    ④
        }
        return null;
    }
}
```

上述代码第①行定义了一个静态方法用来将字符串解析成日期，但并非所有的字符串都是有效的日期字符串，因此调用代码第②行的解析方法parse()有可能发生ParseException异常，ParseException是受检查异常，在本例中使用try-catch捕获。代码第③行的e就是ParseException对象。代码第④行e.printStackTrace()是打印异常堆栈跟踪信息，本例中的"2018-8-18"字符串是有个有效的日期字符串，因此不会发生异常。如果将字符串改为无效的日期字符串，如"201A-18-18"，则会打印信息。

```
处理ParseException
java.text.ParseException: Unparseable date: "201A-18-18"
日期  = null
    at java.text.DateFormat.parse(Unknown Source)
    at com.a51work6.HelloWorld.readDate(HelloWorld.java:24)
    at com.a51work6.HelloWorld.main(HelloWorld.java:13)
```

提示　在捕获到异常之后，通过e.printStackTrace()语句打印异常堆栈跟踪信息，往往只是用于调试，给程序员提示信息。堆栈跟踪信息对最终用户是没有意义的，本例中如果出现异常很有可能是用户输入的日期无效，捕获到异常之后给用户弹出一个对话框，提示用户输入日期无效，请用户重新输入，用户重新输入后再重新调用上述方法。这才是捕获异常之后的正确处理方案。

19.3.2　多**catch**代码块

如果try代码块中有很多语句会发生异常，而且发生的异常种类又很多。那么可以在try后面跟有多个catch代码块。多catch代码块语法如下：

```
try{
    //可能会发生异常的语句
} catch(Throwable e){
    //处理异常e
} catch(Throwable e){
    //处理异常e
} catch(Throwable e){
    //处理异常e
}
```

在多个catch代码情况下，当一个catch代码块捕获到一个异常时，其他的catch代码块就不再进行匹配。

注意　当捕获的多个异常类之间存在父子关系时，捕获异常顺序与catch代码块的顺序有关。一般先捕获子类，后捕获父类，否则子类捕获不到。

示例代码如下：

```
//HelloWorld.java文件
package com.a51work6;

……

public class HelloWorld {

    public static void main(String[] args) {
        Date date = readDate();
        System.out.println("读取的日期  = " + date);
    }

    public static Date readDate() {

        FileInputStream readfile = null;
        InputStreamReader ir = null;
        BufferedReader in = null;
        try {
            readfile = new FileInputStream("readme.txt");     ①
            ir = new InputStreamReader(readfile);
            in = new BufferedReader(ir);
            // 读取文件中的一行数据
            String str = in.readLine();                       ②
            if (str == null) {
                return null;
            }

            DateFormat df = new SimpleDateFormat("yyyy-MM-dd");
            Date date = df.parse(str);                        ③
            return date;

        } catch (FileNotFoundException e) {                   ④
            System.out.println("处理FileNotFoundException...");
            e.printStackTrace();
        } catch (IOException e) {                             ⑤
            System.out.println("处理IOException...");
            e.printStackTrace();
        } catch (ParseException e) {                          ⑥
            System.out.println("处理ParseException...");
            e.printStackTrace();
        }
        return null;
    }

}
```

上述代码通过Java I/O（输入输出）流技术从文件readme.txt中读取字符串，然后解析成为日期。由于Java I/O技术还没有介绍，读者先不要关注I/O技术细节，这考虑调用它们的方法会发生异常就可以了。

在try代码块中第①行代码调用FileInputStream构造方法可以会发生FileNotFoundException异常。第②行代码调用BufferedReader输入流的readLine()方法可以会发生IOException异常。从图19-1可见FileNotFoundException异常是IOException异常的子类，应该先FileNotFoundException捕获，见代码第④行；后捕获IOException，见代码第⑤行。

如果将FileNotFoundException和IOException捕获顺序调换，代码如下：

```
try{
    //可能会发生异常的语句
} catch (IOException e) {
    // IOException异常处理
} catch (FileNotFoundException e) {
    // FileNotFoundException异常处理
}
```

那么第二个catch代码块永远不会进入，FileNotFoundException异常处理永远不会执行。

由于上述代码第⑥行ParseException异常与IOException和FileNotFoundException异常没有父子关系，捕获ParseException异常位置可以随意放置。

19.3.3 try-catch语句嵌套

Java提供的try-catch语句嵌套是可以任意嵌套，修改19.3.2节示例代码如下：

```
//HelloWorld.java文件
package com.a51work6;
… …
public class HelloWorld {

    public static void main(String[] args) {
        Date date = readDate();
        System.out.println("读取的日期  = " + date);
    }

    public static Date readDate() {

        FileInputStream readfile = null;
        InputStreamReader ir = null;
        BufferedReader in = null;
        try {
            readfile = new FileInputStream("readme.txt");
            ir = new InputStreamReader(readfile);
            in = new BufferedReader(ir);

            try {                                                    ①
                String str = in.readLine();                          ②
                if (str == null) {
                    return null;
                }

                DateFormat df = new SimpleDateFormat("yyyy-MM-dd");
                Date date = df.parse(str);                           ③
                return date;

            } catch (ParseException e) {
                System.out.println("处理ParseException...");
                e.printStackTrace();
            }                                                        ④

        } catch (FileNotFoundException e) {                          ⑤
            System.out.println("处理FileNotFoundException...");
            e.printStackTrace();
        } catch (IOException e) {                                    ⑥
            System.out.println("处理IOException...");
            e.printStackTrace();
        }
        return null;
    }
}
```

上述代码第①~④行是捕获ParseException异常try-catch语句，可见这个try-catch语句就是嵌套在捕获IOException和FileNotFoundException异常的try-catch语句中。

程序执行时内层如果会发生异常，首先由内层catch进行捕获，如果捕获不到，则由外层catch捕获。例如：代码第②行的readLine()方法可能发生IOException异常，该异常无法被内层catch捕获，最后被代码第⑥行的外层catch捕获。

注意　try-catch不仅可以嵌套在try代码块中，还可以嵌套在catch代码块或finally代码块，finally代码块后面会详细介绍。try-catch嵌套会使程序流程变的复杂，如果能用多catch捕获的异常，尽量不要使用try-catch嵌套。特别对于初学者不要简单地使用Eclipse的语法提示不加区分地添加try-catch嵌套，要梳理好程序的流程再考虑try-catch嵌套的必要性。

19.3.4　多重捕获

多catch代码块客观上提高了程序的健壮性，但是程序代码量大大增加。如果有些异常虽然种类不同，但捕获之后的处理是相同的，看如下代码。

```
try{
    //可能会发生异常的语句
} catch (FileNotFoundException e) {
    //调用方法methodA处理
} catch (IOException e) {
    //调用方法methodA处理
} catch (ParseException e) {
    //调用方法methodA处理
}
```

三个不同类型的异常，要求捕获之后的处理都是调用methodA方法。是否可以把这些异常合并处理，Java 7推出了多重捕获（multi-catch）技术，可以帮助解决此类问题，上述代码修改如下：

```
try{
    //可能会发生异常的语句
} catch (IOException | ParseException e) {
    //调用方法methodA处理
}
```

在catch中多重捕获异常用“|”运算符连接起来。

注意　有的读者会问什么不写成FileNotFoundException | IOException | ParseException 呢？这是因为由于FileNotFoundException属于IOException异常，IOException异常可以捕获它的所有子类异常了。

19.4 释放资源

有时在try-catch语句中会占用一些非Java资源，如：打开文件、网络连接、打开数据库连接和使用数据结果集等，这些资源并非Java资源，不能通过JVM的垃圾收集器回收，需要程序员释放。为了确保这些资源能够被释放可以使用finally代码块或Java 7之后提供自动资源管理（Automatic Resource Management）技术。

19.4.1 finally代码块

try-catch语句后面还可以跟有一个finally代码块，try-catch-finally语句语法如下：

```
try{
    //可能会生成异常语句
} catch(Throwable e1){
    //处理异常e1
} catch(Throwable e2){
    //处理异常e2
} catch(Throwable eN){
    //处理异常eN
} finally{
    //释放资源
}
```

无论try正常结束还是catch异常结束都会执行finally代码块，如图19-2所示。

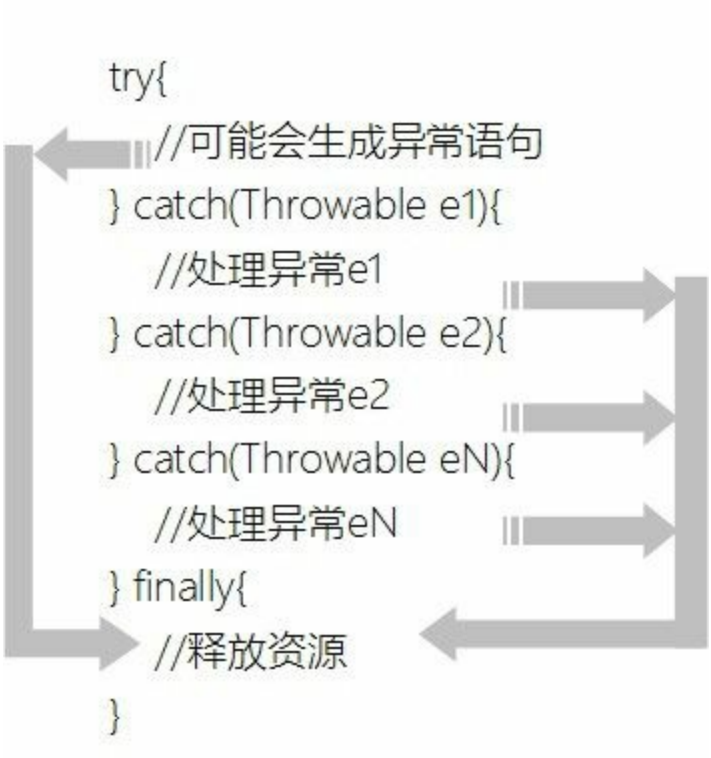


图19-2 finally代码块流程

使用finally代码块示例代码如下：

```
//HelloWorld.java文件
package com.a51work6;

… …

public class HelloWorld {

    public static void main(String[] args) {
        Date date = readDate();
        System.out.println("读取的日期  = " + date);
    }

    public static Date readDate() {

        FileInputStream readfile = null;
        InputStreamReader ir = null;
        BufferedReader in = null;
        try {
            readfile = new FileInputStream("readme.txt");
            ir = new InputStreamReader(readfile);
            in = new BufferedReader(ir);
            // 读取文件中的一行数据
            String str = in.readLine();
            if (str == null) {
                return null;
            }

            DateFormat df = new SimpleDateFormat("yyyy-MM-dd");
            Date date = df.parse(str);
            return date;

        } catch (FileNotFoundException e) {
            System.out.println("处理FileNotFoundException...");
            e.printStackTrace();
        } catch (IOException e) {
            System.out.println("处理IOException...");
            e.printStackTrace();
        } catch (ParseException e) {
            System.out.println("处理ParseException...");
            e.printStackTrace();
        } finally {                                 ①
            try {
                if (readfile != null) {
                    readfile.close();               ②
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
            try {
                if (ir != null) {
                    ir.close();                     ③
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
            try {
                if (in != null) {
                    in.close();                     ④
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }                                           ⑤

        return null;
    }
```

```
}
```

上述代码第①行~第⑤行是finally语句，在这里通过关闭流释放资源，FileInputStream、InputStreamReader和BufferedReader是三个输入流，它们都需要关闭，见代码第②行~第④行通过流的close()关闭流，但是流的close()方法还有可以能发生IOException异常，所以这里又针对每一个close()语句还需要进行捕获处理。

注意　为了代码简洁等目的，可能有的人会将finally代码中的多个嵌套的try-catch语句合并，例如将上述代码改成如下形式，将三个有可以发生异常的close()方法放到一个try-catch。读者自己考虑一下这处理是否稳妥呢？每一个close()方法对应关闭一个资源，如果第一个close()方法关闭时发生了异常，那么后面的两个也不会关闭，因此如下的程序代码是有缺陷的。

```
try {
    ... ...
} catch (FileNotFoundException e) {
    ... ...
} catch (IOException e) {
    ... ...
} catch (ParseException e) {
    ... ...
} finally {
    try {
        if (readfile != null) {
            readfile.close();
        }
        if (ir != null) {
            ir.close();
        }
        if (in != null) {
            in.close();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

19.4.2 自动资源管理

19.4.1节使用finally代码块释放资源会导致程序代码大量增加，一个finally代码块往往比正常执行的程序还要多。在Java 7之后提供自动资源管理（Automatic Resource Management）技术，可以替代finally代码块，优化代码结构，提高程序可读性。

自动资源管理是在try语句上的扩展，语法如下：

```
try (声明或初始化资源语句) {
    //可能会生成异常语句
} catch(Throwable e1){
    //处理异常e1
} catch(Throwable e2){
    //处理异常e1
} catch(Throwable eN){
    //处理异常eN
}
```

在try语句后面添加一对小括号“()”，其中是声明或初始化资源语句，可以有多条语句语句之间用分号“;”分隔。

示例代码如下：

```
//HelloWorld.java文件
package com.a51work6;
… …
public class HelloWorld {

    public static void main(String[] args) {
        Date date = readDate();
        System.out.println("读取的日期  = " + date);
    }

    public static Date readDate() {

        // 自动资源管理
        try (FileInputStream readfile = new FileInputStream("readme.txt");   ①
                InputStreamReader ir = new InputStreamReader(readfile);      ②
                BufferedReader in = new BufferedReader(ir)) {                ③

            // 读取文件中的一行数据
            String str = in.readLine();
            if (str == null) {
                return null;
            }

            DateFormat df = new SimpleDateFormat("yyyy-MM-dd");
            Date date = df.parse(str);
            return date;

        } catch (FileNotFoundException e) {
            System.out.println("处理FileNotFoundException...");
            e.printStackTrace();
        } catch (IOException e) {
            System.out.println("处理IOException...");
            e.printStackTrace();
        } catch (ParseException e) {
            System.out.println("处理ParseException...");
            e.printStackTrace();
        }

        return null;
    }

}
```

上述代码第①行~第③行是声明或初始化三个输入流，三条语句放到在try语句后面小括号中，语句之间用分号“;”分隔，这就是自动资源管理技术了，采用了自动资源管理后不再需要finally代码块，不需要自己close这些资源，释放过程交给了JVM。

注意　所有可以自动管理的资源需要实现AutoCloseable接口，上述代码中三个输入流FileInputStream、InputStreamReader和BufferedReader从Java 7之后实现AutoCloseable接口，具体哪些资源实现AutoCloseable接口需要查询API文档。

19.5 throws与声明方法抛出异常

在一个方法中如果能够处理异常，则需要捕获并处理。但是本方法没有能力处理该异常，捕获它没有任何意义，则需要在方法后面声明抛出该异常，通知上层调用者该方法有可以发生异常。

方法后面声明抛出使用throws关键字，回顾一下11.3.3节成员方法语法格式如下：

```
class className {

        [public | protected | private ] [static] [final | abstract] [native] [synchronized]
                type methodName([paramList]) [throws exceptionList] {
                    //方法体
        }
}
```

其中参数列表之后的[throws exceptionList]语句是声明抛出异常。方法中可能抛出的异常（除了Error和RuntimeException及其子类外）都必须通过throws语句列出，多个异常之间采用逗号（,）分隔。

注意　如果声明抛出的多个异常类之间有父子关系，可以只声明抛出父类。但如果没有父子关系情况下，最好明确声明抛出每一个异常，因为上层调用者会根据这些异常信息进行相应的处理。假如一个方法中有可能抛出IOException和ParseException两个异常，那么声明抛出IOException和ParseException呢？还是只声明抛出Exception呢？因为Exception是IOException和ParseException的父类，只声明抛出Exception从语法是允许的，但是声明抛出IOException和ParseException更好一些。

如果将19.3节示例进行修改，在readDate()方法后声明抛出异常，代码如下：

```
//HelloWorld.java文件
package com.a51work6;

… …
public class HelloWorld {

    public static void main(String[] args) {            ①

        try {
            Date date = readDate();                      ②
            System.out.println("读取的日期  = " + date);
        } catch (IOException e) {                        ③
            System.out.println("处理IOException...");
            e.printStackTrace();
        } catch (ParseException e) {                     ④
            System.out.println("处理ParseException...");
            e.printStackTrace();
        }

    }

    public static Date readDate() throws IOException, ParseException {    ⑤

        // 自动资源管理
        FileInputStream readfile = new FileInputStream("readme.txt");     ⑥
        InputStreamReader ir = new InputStreamReader(readfile);
        BufferedReader in = new BufferedReader(ir);

        // 读取文件中的一行数据
        String str = in.readLine();                                  ⑦
        if (str == null) {
            return null;
```

```
            }

            DateFormat df = new SimpleDateFormat("yyyy-MM-dd");
            Date date = df.parse(str);                              ⑧
            return date;
        }

    }
```

由于readDate()方法中代码第⑥、⑦、⑧行都有可能引发异常。在readDate()方法内又没有捕获处理，所以需要在代码第⑤行方法后声明抛出异常，事实上有三个异常FileNotFoundException、IOException和ParseException，由于FileNotFoundException属于IOException异常，所以只声明IOException和ParseException就可以了。

一旦readDate()方法声明抛出了异常，那么它的调用者main()方法，也会面临同样的问题：要么捕获自己处理，要么抛出给上层调用者。如果一旦发生异常main()方法也选择抛出那么程序运行就会终止。本例中main()方法是捕获异常进行处理，捕获异常过程前面已经介绍过了，这里不再赘述。

19.6 自定义异常类

有些公司为了提高代码的可重用性，自己开发了一些Java类库或框架，其中少不了自己编写了一些异常类。实现自定义异常类需要继承Exception类或其子类，如果自定义运行时异常类需继承RuntimeException类或其子类。

实现自定义异常类示例代码如下：

```
package com.a51work6;

public class MyException extends Exception {    ①

    public MyException() {                      ②

    }

    public MyException(String message) {        ③
        super(message);
    }

}
```

上述代码实现了自定义异常，自定义异常类一般需要提供两个构造方法，一个是代码第②行的无参数的默认构造方法，异常描述信息是空的；另一个是代码第③行的字符串参数的构造方法，message是异常描述信息，getMessage()方法可以获得这些信息。

自定义异常就这样简单，主要是提供两个构造方法就可以了。

19.7 throw与显式抛出异常

Java异常相关的关键字中有两个非常相似，它们是throws和throw，其中throws关键字前面19.5节已经介绍了，throws用于方法后声明抛出异常，而throw关键字用来人工引发异常。本节之前读者接触到的异常都是由于系统生成的，当异常发生时，系统会生成一个异常对象，并将其抛出。但也可以通过throw语句显式抛出异常，语法格式如下：

```
throw Throwable或其子类的实例
```

所有Throwable或其子类的实例都可以通过throw语句抛出。

显式抛出异常目的有很多，例如不想某些异常传给上层调用者，可以捕获之后重新显式抛出另外一种异常给调用者。

修改19.4节示例代码如下：

```
//HelloWorld.java文件
package com.a51work6;
… …
public class HelloWorld {

    public static void main(String[] args) {
        try {
            Date date = readDate();
            System.out.println("读取的日期  = " + date);
        } catch (MyException e) {
            System.out.println("处理MyException...");
            e.printStackTrace();
        }
    }

    public static Date readDate() throws MyException {

        // 自动资源管理
        try (FileInputStream readfile = new FileInputStream("readme.txt");
                InputStreamReader ir = new InputStreamReader(readfile);
                BufferedReader in = new BufferedReader(ir)) {

            // 读取文件中的一行数据
            String str = in.readLine();
            if (str == null) {
                return null;
            }

            DateFormat df = new SimpleDateFormat("yyyy-MM-dd");
            Date date = df.parse(str);
            return date;

        } catch (FileNotFoundException e) {                ①
            throw new MyException(e.getMessage());         ②
        } catch (IOException e) {                          ③
            throw new MyException(e.getMessage());         ④
        } catch (ParseException e) {
            System.out.println("处理ParseException...");
            e.printStackTrace();
        }
        return null;
    }
}
```

如果软件设计者不希望readDate()方法中捕获的FileNotFoundException和IOException异常出现在main()方法（上层调用者）中，那么可以在捕获到FileNotFoundException和IOException异常时，通过throw语句显式抛出一个异常，见代码第②行和第④行throw new MyException(e.getMessage())语句，MyException是自定义的异常。

注意　throw显式抛出的异常与系统生成并抛出的异常，在处理方式上没有区别，就是两种方法：要么捕获自己处理，要么抛出给上层调用者。在本例中是声明抛出，所以在readDate()方法后面要声明抛出MyException异常。

本章小结

本章介绍了Java异常处理机制，其中包括Java异常类继承层次、捕获异常、释放资源、throws、throw和自定义异常类。读者需要重点掌握捕获异常处理，熟悉throws和throw的区分和用法。

第 **20** 章　对象容器——集合

当你有很多书时，你会考虑买一个书柜，将你的书分门别类摆放进入。使用了书柜不仅仅使房间变得整洁，也便于以后使用书时方便查找。在计算机中管理对象亦是如此，当获得多个对象后，也需要一个容器将它们管理起来，这个容器就是集合。

集合本质是基于某种数据结构数据容器。常见的数据结构：数组（Array）、集（Set）、队列（Queue）、链表（Linkedlist）、树（Tree）、堆（Heap）、栈（Stack）和映射（Map）等结构。

本章介绍Java中的集合。

20.1 集合概述

Java中提供了丰富的集合接口和类，它们来自于java.util包。如图20-1所示是Java主要的集合接口和类，从图中可见Java集合类型分为：Collection和Map，Collection子接口有：Set、Queue和List等接口。每一种集合接口描述了一种数据结构。

本章重点介绍List、Set和Map接口，因此图20-1中只列出了这三个接口的具体实现类，事实上Queue也有具体实现类，由于很少使用，这里不再赘述，读者感兴趣可以自己查询API文档。

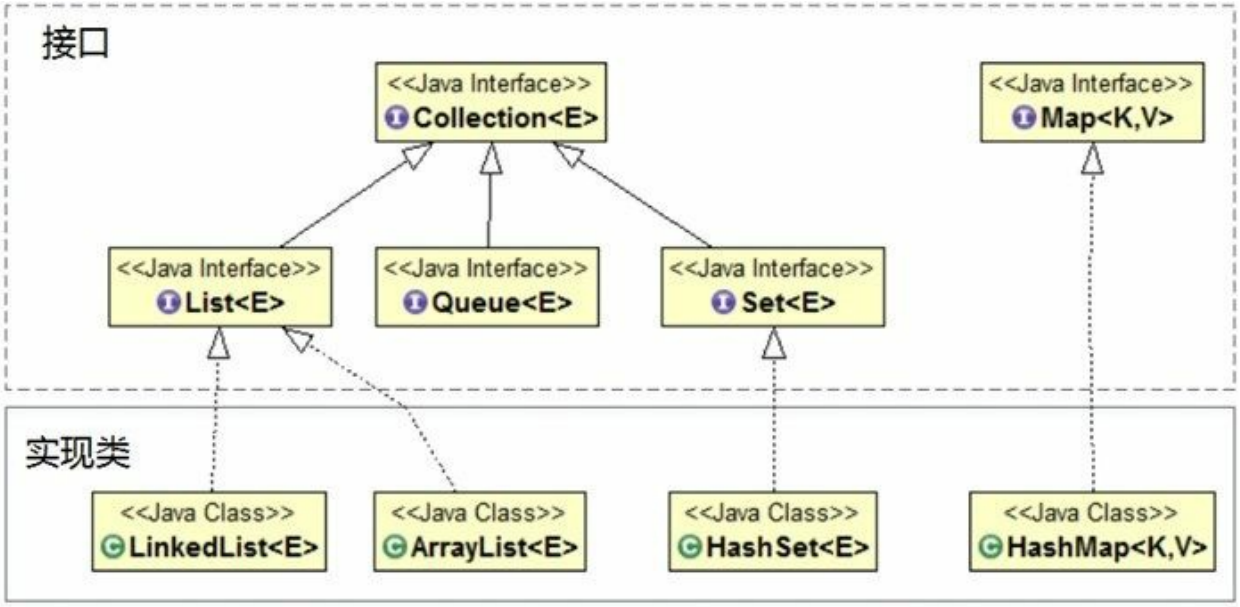


图**20-1　Java**主要集合接口和类

提示　在Java SE中List名称的类型有两个，一个是java.util.List，另外一个是java.awt.List。java.util.List是一个接口，这本章介绍的List集合。而java.awt.List是一个类，用于图形用户界面开发，它是一个图形界面中的组件。

提示　学习Java中的集合，首先从两大接口入手，重点掌握List、Set和Map三个接口，熟悉这些接口中提供的方法。然后再熟悉这些接口的实现类，并了解不同实现类之间的区别。

20.2 List集合

List集合中的元素是有序的，可以重复出现。图20-2是一个班级集合数组，这个集合中有一些学生，这些学生是有序的，顺序是他们被放到集合中的顺序，可以通过序号访问他们。这就像老师给进入班级的人分配学号，第一个报到的是“张三”，老师给他分配的是0，第二个报到的是“李四”，老师给他分配的是1，以此类推，最后一个序号应该是“学生人数-1”。

图**20-2** 数组集合

> 提示 List集合关心的元素是否有序，而不关心是否重复，请大家记住这个原则。例如，图20-2所示的班级集合中就有两个“张三”。

List接口的实现类有：ArrayList 和 LinkedList。ArrayList是基于动态数组数据结构的实现，LinkedList是基于链表数据结构的实现。ArrayList访问元素速度优于LinkedList，LinkedList占用的内存空间比较大，但LinkedList在批量插入或删除数据时优于ArrayList。

不同的结构对应于不同的算法，有的考虑节省占用空间，有的考虑提高运行效率，对于程序员而言，

它们就像是“熊掌”和“鱼肉”，不可兼得！提高运行速度往往是以牺牲空间为代价的，而节省占用空间往往是以牺牲运行速度为代价的。

20.2.1 常用方法

List接口继承自Collection接口，List接口中的很多方法都继承自Collection接口的。List接口中常用方法如下。

01. 操作元素

- get(int index)：返回List集合中指定位置的元素。
- set(int index, Object element)：用指定元素替换List集合中指定位置的元素。
- add(Object element)：在List集合的尾部添加指定的元素。该方法是从Collection集合继承过来的。
- add(int index, Object element)：在List集合的指定位置插入指定元素。
- remove(int index)：移除List集合中指定位置的元素。
- remove(Object element)：如果List集合中存在指定元素，则从List集合中移除第一次出现的指定元素。该方法是从Collection集合继承过来的。
- clear()：从List集合中移除所有元素。该方法是从Collection集合继承过来的。

02. 判断元素

- isEmpty()：判断List集合中是否有元素，没有返回true，有返回false。该方法是从Collection集合继承过来的。
- contains(Object element)：判断List集合中是否包含指定元素，包含返回true，不包含返回false。该方法是从Collection集合继承过来的。

03. 查询元素

- indexOf(Object o)：从前往后查找List集合元素，返回第一次出现指定元素的索引，如果此列表不包含该元素，则返回-1。
- lastIndexOf(Object o)：从后往前查找List集合元素，返回第一次出现指定元素的索引，如果此列表不包含该元素，则返回-1。

04. 其他

- iterator()：返回迭代器（Iterator）对象，迭代器对象用于遍历集合。该方法是从Collection集合继承过来的。
- size()：返回List集合中的元素数，返回值是int类型。该方法是从Collection集合继承过来的。
- subList(int fromIndex, int toIndex)：返回List集合中指定的 fromIndex（包括 ）和 toIndex（不包括）之间的元素集合，返回值为List集合。

示例代码如下：

```
//HelloWorld.java文件
```

```
package com.a51work6;

import java.util.ArrayList;
import java.util.List;

public class HelloWorld {

    public static void main(String[] args) {

        List list = new ArrayList();                    ①

        String b = "B";

        //向集合中添加元素
        list.add("A");
        list.add(b);                                    ②
        list.add("C");
        list.add(b);                                    ③
        list.add("D");
        list.add("E");

        //打印集合元素个数
        System.out.println("集合size = " + list.size());
        //打印集合
        System.out.println(list);

        //从前往后查找集合中的"B"元素
        System.out.println("indexOf(\"B\") = " + list.indexOf(b));
        //从后往前查找集合中的"B"元素
        System.out.println("lastIndexOf(\"B\") = " + list.lastIndexOf(b));

        //删除集合中第一个"B"元素
        list.remove(b);
        System.out.println("remove(3)前：  " + list);
        //判断集合中是否包含"B"元素
        System.out.println("是否包含\"B\"：" + list.contains(b));

        //删除集合第4个元素
        list.remove(3);
        System.out.println("remove(3)后：  " + list);
        //判断集合是否为空
        System.out.println("list集合是空的：" + list.isEmpty());

        System.out.println("替换前：" + list);
        //替换集合第2个元素
        list.set(1, "F");
        System.out.println("替换后：" + list);

        //清空集合
        list.clear();                                   ④
        System.out.println(list);


        // 重新添加元素
        list.add(1);//发生自动装箱                       ⑤
        list.add(3);

        int item = (Integer)list.get(0);//发生自动拆箱    ⑥
    }
}
```

运行结果如下：

```
集合size = 6
```

```
[A, B, C, B, D, E]
indexOf("B") = 1
lastIndexOf("B") = 3
remove(3)前： [A, C, B, D, E]
是否包含"B"：true
remove(3)后： [A, C, B, E]
list集合是空的：false
替换前：[A, C, B, E]
替换后：[A, F, B, E]
[]
```

代码第①行声明List类型集合变量list，使用ArrayList类实例化list，List是接口不能实例化。添加集合元素过程中可以添加重复的元素，见代码第②行和第③行。代码第④行list.clear()是清空集合，但需要注意的是变量list所引用的对象还是存在的，不是null，只是集合中没有了元素。

提示　在Java中任何集合中存放的都是对象，即引用数据类型，基本数据类型不能放到集合中。但上述代码第⑤行却将整数1放到集合中，这是因为这个过程中发生了自动装箱，整数1被封装成Integer对象1，然后再放入到集合中。相反从集合中取出的也是对象，代码第⑥行从集合中取出的是Integer对象，之所以能够赋值给int类型，是因为这个过程发生了自动拆箱。

20.2.2 遍历集合

集合最常用的操作之一是遍历，遍历就是将集合中的每一个元素取出来，进行操作或计算。List集合遍历有三种方法：

01. 使用for循环遍历：List集合可以使用for循环进行遍历，for循环中有循环变量，通过循环变量可以访问List集合中的元素。

02. 使用for-each循环遍历：for-each循环是针对遍历各种类型集合而推出的，笔者推荐使用这种遍历方法。

03. 使用迭代器遍历：Java提供了多种迭代器，List集合可以使用Iterator和ListIterator迭代器。

示例代码如下：

```
//HelloWorld.java文件
package com.a51work6;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class HelloWorld {

    public static void main(String[] args) {

        List list = new ArrayList();

        String b = "B";
        // 向集合中添加元素
        list.add("A");
        list.add(b);
        list.add("C");
        list.add(b);
        list.add("D");
        list.add("E");

        // 1.使用for循环遍历
        System.out.println("--1.使用for循环遍历--");
        for (int i = 0; i < list.size(); i++) {
```

```
            System.out.printf("读取集合元素(%d): %s \n", i, list.get(i));      ①
        }

        // 2.使用for-each循环遍历
        System.out.println("--2.使用for-each循环遍历--");
        for (Object item : list) {                                    ②
            String s = (String) item;                                 ③
            System.out.println("读取集合元素: " + s);
        }

        // 3.使用迭代器遍历
        System.out.println("--3.使用迭代器遍历--");
        Iterator it = list.iterator();                                ④
        while (it.hasNext()) {                                        ⑤
            Object item = it.next();                                  ⑥
            String s = (String) item;                                 ⑦
            System.out.println("读取集合元素: " + s);
        }
    }
}
```

上述代码采用三种方法遍历List集合，采用for循环遍历需要通过List集合的get方法获得元素，代码第①行的list.get(i)。代码第②行采用for-each循环遍历list集合，从集合中取出的元素都是Object类型，代码第③行是强制转换为String类型。使用迭代器遍历，首先需要获得迭代器对象，代码第④行list.iterator()方法可以返回迭代器对象。代码第⑤行调用迭代器hasNext()方法可以判断集合中是否还有元素可以迭代，有返回true，没有返回false。代码第⑥行调用迭代器的next()返回迭代的下一个元素，该方法返回的Object类型需要强制转换为String类型，见代码第⑦行。

20.3 Set集合

Set集合是由一串无序的，不能重复的相同类型元素构成的集合。图20-3是一个班级的Set集合。这个Set集合中有一些学生，这些学生是无序的，不能通过类似于List集合的序号访问，而且不能有重复的同学。

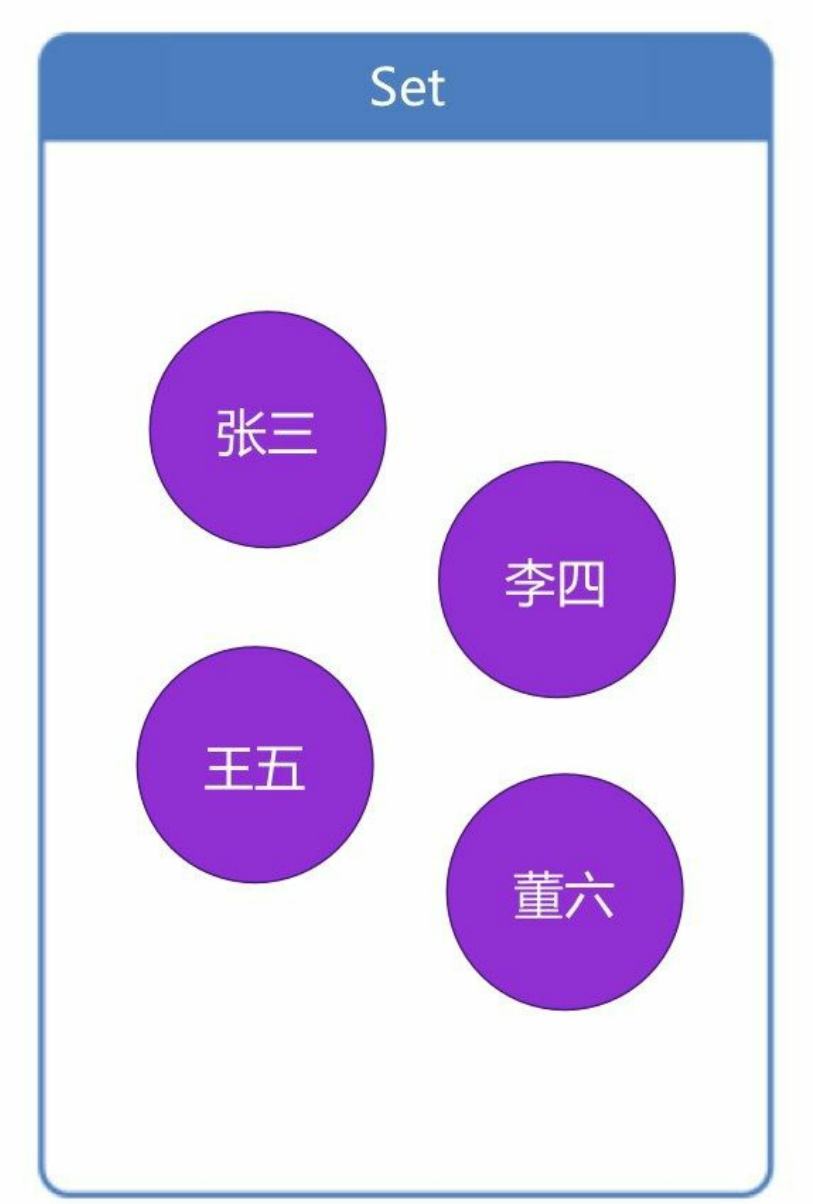


图**20-3 Set**集合

提示 List集合中的元素是有序的、可重复的，而Set集合中的元素是无序的、不能重复的。List集合强调的是有序，Set集合强调的是不重复。当不考虑顺序，且没有重复元素时，Set集合和List集合可以互相替换的。

Set接口直接实现类主要是HashSet，HashSet是基于散列表数据结构的实现。

20.3.1 常用方法

Set接口也继承自Collection接口，Set接口中大部分都是继承自Collection接口，这些方法如下。

01. 操作元素

- add(Object element)：在Set集合的尾部添加指定的元素。该方法是从Collection集合继承过来的。
- remove(Object element)：如果Set集合中存在指定元素，则从Set集合中移除该元素。该方法是从Collection集合继承过来的。
- clear()：从Set集合中移除所有元素。该方法是从Collection集合继承过来的。

02. 判断元素

- isEmpty()：判断Set集合中是否有元素，没有返回true，有返回false。该方法是从Collection集合继承过来的。
- contains(Object element)：判断Set集合中是否包含指定元素，包含返回true，不包含返回false。该方法是从Collection集合继承过来的。

03. 其他

- iterator()：返回迭代器（Iterator）对象，迭代器对象用于遍历集合。该方法是从Collection集合继承过来的。
- size()：返回Set集合中的元素数，返回值是int类型。该方法是从Collection集合继承过来的。

示例代码如下：

```
//HelloWorld.java文件
package com.a51work6;

import java.util.HashSet;
import java.util.Set;

public class HelloWorld {

    public static void main(String[] args) {

        Set set = new HashSet();                          ①

        String b = "B";

        // 向集合中添加元素
        set.add("A");
        set.add(b);                                       ②
        set.add("C");
        set.add(b);                                       ③
        set.add("D");
        set.add("E");

        // 打印集合元素个数
        System.out.println("集合size = " + set.size());        ④
        // 打印集合
        System.out.println(set);

        // 删除集合中第一个"B"元素
```

```
        set.remove(b);
        // 判断集合中是否包含"B"元素
        System.out.println("是否包含\"B\"：" + set.contains(b));
        // 判断集合是否为空
        System.out.println("set集合是空的：" + set.isEmpty());

        // 清空集合
        set.clear();
        System.out.println(set);
    }
}
```

运行结果：

```
集合size = 5
[A, B, C, D, E]
是否包含"B"：false
set集合是空的：false
[]
```

代码第①行声明Set类型集合变量set，使用HashSet类实例化set，Set是接口不能实例化。添加集合元素时试图添加重复的元素，见代码第②行和第③行，但是Set集合不能添加重复元素，所有代码第④行打印集合元素个数是5。

20.3.2 遍历集合

Set集合中的元素由于没有序号，所以不能使用for循环进行遍历，但可以使用for-each循环和迭代器进行遍历。事实上这两种遍历方法也是继承自Collection集合，也就是说所有的Collection集合类型都有这两种遍历方式。

示例代码如下：

```
//HelloWorld.java文件
package com.a51work6;

import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

public class HelloWorld {

    public static void main(String[] args) {

        Set set = new HashSet();

        String b = "B";
        // 向集合中添加元素
        set.add("A");
        set.add(b);
        set.add("C");
        set.add(b);
        set.add("D");
        set.add("E");

        // 1.使用for-each循环遍历
        System.out.println("--1.使用for-each循环遍历--");
        for (Object item : set) {
            String s = (String) item;
            System.out.println("读取集合元素: " + s);
        }
```

```
        // 2.使用迭代器遍历
        System.out.println("--2.使用迭代器遍历--");
        Iterator it = set.iterator();
        while (it.hasNext()) {
            Object item = it.next();
            String s = (String) item;
            System.out.println("读取集合元素: " + s);
        }
    }
}
```

上述代码采用两种方法遍历Set集合，具体实现与List集合完全一样，这里不再赘述。

20.4 Map集合

Map（映射）集合表示一种非常复杂的集合，允许按照某个键来访问元素。Map集合是由两个集合构成的，一个是键（key）集合，一个是值（value）集合。键集合是Set类型，因此不能有重复的元素。而值集合是Collection类型，可以有重复的元素。Map集合中的键和值是成对出现的。

图20-4所示是Map类型的“国家代号”集合。键是国家代号集合，不能重复。值是国家集合，可以重复。

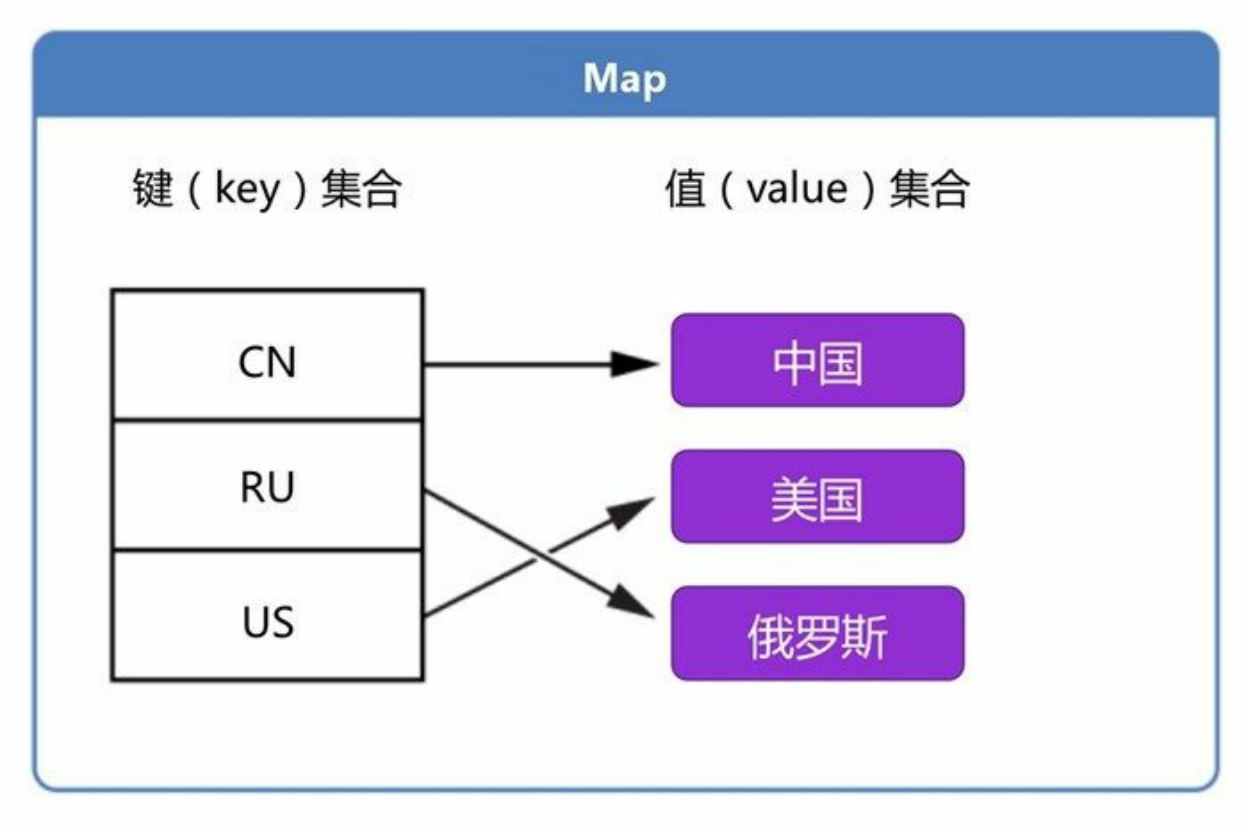


图**20-4 Map**集合

> 提示 Map集合更适合通过键快速访问值，就像查英文字典一样，键就是要查的英文单词，而值是英文单词的翻译和解释等。有的时候，一个英文单词会对应多个翻译和解释，这是与Map集合特性对应的。

Map接口直接实现类主要是HashMap，HashMap是基于散列表数据结构的实现。

20.4.1 常用方法

Map集合中包含两个集合（键和值），所以操作起来比较麻烦，Map接口提供很多方法用来管理和操作集合。主要的方法如下。

01. 操作元素

- get(Object key)：返回指定键所对应的值；如果Map集合中不包含该键值对，则返回null。
- put(Object key, Object value)：指定键值对添加到集合中。

- remove(Object key)：移除键值对。
- clear()：移除Map集合中所有键值对。

02. 判断元素

- isEmpty()：判断Map集合中是否有键值对，没有返回true，有返回false。
- containsKey(Object key)：判断键集合中是否包含指定元素，包含返回true，不包含返回false。
- containsValue(Object value)：判断值集合中是否包含指定元素，包含返回true，不包含返回false。

03. 查看集合

- keySet()：返回Map中的所有键集合，返回值是Set类型。
- values()：返回Map中的所有值集合，返回值是Collection类型。
- size()：返回Map集合中键值对数。

示例代码如下：

```
//HelloWorld.java文件
package com.a51work6;

import java.util.HashMap;
import java.util.Map;

public class HelloWorld {

    public static void main(String[] args) {

        Map map = new HashMap();                ①

        map.put(102, "张三");
        map.put(105, "李四");                    ②
        map.put(109, "王五");
        map.put(110, "董六");
        //"李四"值重复
        map.put(111, "李四");                    ③
        //109键已经存在，替换原来值"王五"
        map.put(109, "刘备");                    ④


        // 打印集合元素个数
        System.out.println("集合size = " + map.size());
        // 打印集合
        System.out.println(map);

        // 通过键取值
        System.out.println("109 - " + map.get(109));    ⑤
        System.out.println("108 - " + map.get(108));    ⑥

        // 删除键值对
        map.remove(109);
        // 判断键集合中是否包含109
        System.out.println("键集合中是否包含109：" + map.containsKey(109));
        // 判断值集合中是否包含 "李四"
        System.out.println("值集合中是否包含：" + map.containsValue("李四"));
```

```
        // 判断集合是否为空
        System.out.println("集合是空的：" + map.isEmpty());

        // 清空集合
        map.clear();
        System.out.println(map);
    }
}
```

运行结果如下：

```
集合size = 5
{102=张三, 105=李四, 109=王五, 110=董六, 111=刘备}
109 - 王五
108 - null
是否包含"B"：false
值集合中是否包含：true
集合是空的：false
{}
```

代码第①行声明Map类型集合变量map，使用HashMap类实例化map，Map是接口不能实例化。Map集合添加键值对时候需要注意两个问题：第一，如果键已经存在，则会替换原有值，见代码第④行是109键原来对应的是"王五"，该语句会替换为"刘备"；第二，如果这个值已经存在，则不会替换，见代码第②行和第③行，添加了两个相同的值"李四"。

代码第⑤行和第⑥行是通过键取对应的值，如果不存在键值对，则返回null，代码第⑥行的108键对应的值不存在，所以这里打印的是null。

20.4.2 遍历集合

Map集合遍历与List和Set集合不同，Map有两个集合，因此遍历时可以只遍历值的集合，也可以只遍历键的集合，也可以同时遍历。这些遍历过程都可以使用for-each循环和迭代器进行遍历。

示例代码如下：

```
//HelloWorld.java文件
package com.a51work6;

import java.util.Collection;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;

public class HelloWorld {

    public static void main(String[] args) {

        Map map = new HashMap();

        map.put(102, "张三");
        map.put(105, "李四");
        map.put(109, "王五");
        map.put(110, "董六");
        map.put(111, "李四");

        // 1.使用for-each循环遍历
        System.out.println("--1.使用for-each循环遍历--");
        // 获得键集合
```

```
        Set keys = map.keySet();                                    ①
        for (Object key : keys) {
            int ikey = (Integer) key; // 自动拆箱                    ②
            String value = (String) map.get(ikey); // 自动装箱       ③
            System.out.printf("key=%d - value=%s \n", ikey, value);
        }

        // 2.使用迭代器遍历
        System.out.println("--2.使用迭代器遍历--");
        // 获得值集合
        Collection values = map.values();                           ④
        // 遍历值集合
        Iterator it = values.iterator();
        while (it.hasNext()) {
            Object item = it.next();
            String s = (String) item;
            System.out.println("值集合元素: " + s);
        }

    }
}
```

上述代码第①行是获得键集合，返回值是Set类型。在遍历键时，从集合里取出的元素类型都是Object，代码第②行是将key强制类型转换为Integer，然后又赋值给int整数，这个过程发生了自动拆箱。代码第③行是通过键获得对应的值。

代码第④行是获得值集合，它是Collection类型。遍历Collection集合与遍历Set集合一样，这里不再赘述。

本章小结

本章介绍了Java中的集合，其中包括常用接口Collection、Set、List和Map，重点掌握Set、List和Map三个接口，熟悉具体实现类。熟练几种集合的遍历操作。

第 **21** 章　泛型

Java 5之后提供泛型（Generics）支持，使用泛型可以最大限度地重用代码、保护类型的安全以及提高性能。泛型特性对Java影响最大是集合框架的使用。本章详细介绍使用泛型。

21.1 一个问题的思考

为了理解什么是泛型，请大家先看一个使用集合的示例：

```
//HelloWorld.java文件
package com.a51work6;

import java.util.ArrayList;
import java.util.List;

public class HelloWorld {

    public static void main(String[] args) {

        List list = new ArrayList();

        // 向集合中添加元素
        list.add("1");
        list.add("2");
        list.add("3");
        list.add("4");
        list.add("5");

        // 遍历集合
        for (Object item : list) {                          ①
            Integer element = (Integer) item;               ②
            System.out.println("读取集合元素: " + element);
        }
    }
}
```

上述代码实现的功能很简单，就将一些数据保存到集合中，然后再取出。但对于Java 5之前程序员而言，使用集合经常会面临一个很尴尬的问题：放入一个种特定类型，但是取出时候全部是Object类型，于是在具体使用时候需要将元素转换为特定类型。上述代码第①行取出的元素是Object类型，在代码第②行需要强制类型转换。强制类型转换是有风险的，如果不进行判断就臆断进行类型转换会发生ClassCastException异常。本例代码第②行就会发生了这个异常，JVM会抛出异常，打印出如下的异常堆栈跟踪信息。

```
Exception in thread "main" java.lang.ClassCastException: java.lang.String cannot be cast to java.lang.I
```

从异常堆栈跟踪信息可知，在源代码代码第23行试图将java.lang.String对象java.lang.Integer对象。

在Java 5之前没有好的解决办法，在类型转换之前要通过instanceof运算符判断一下，该对象是否是目标类型。而泛型的引入可以将这些运行时异常提前到编译期暴露出来，这增强了类型安全检查。

修改程序代码如下：

```
//HelloWorldGen.java文件
package com.a51work6;

import java.util.ArrayList;
import java.util.List;

public class HelloWorldGen {

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

```
        List<String> list = new ArrayList<String>();           ①

        // 向集合中添加元素
        list.add("1");
        list.add("2");
        list.add("3");
        list.add("4");
        list.add("5");
        //list.add(new Date); //发生编译错误                    ②

        // 遍历集合
        // 使用for-each循环遍历
        for (String item : list) {                             ③
            //Integer element = (Integer) item;                ④
            System.out.println("读取集合元素: " + item);
        }

    }
}
```

上述代码第①行添加了List和ArrayList 后面添加了<String>，这就是List和ArrayList的泛型表示方式，尖括号中可以任何的引用类型，它限定了集合中是能存放该种类型的对象，所以代码第②行试图添加非String类型元素时，会发生编译错误。

代码第③行从集合取出的元素就是String类型，所以如果在代码第④行试图转换为Integer会发生编译错误。可见原本在运行时发生的异常，提早暴露到编译期，使程序员早发现问题，避免程序发布上线之后发生系统崩溃。

提示　在集合中如果没有使用泛型，Eclipse等IDE工具都会警告，如图21-1所示。读者可以单击这些警告，根据Eclipse的修订功能，修订这些警告，如图21-2所示，在弹出对话框中选择“推断通用类型参数”就可以添加泛型了。

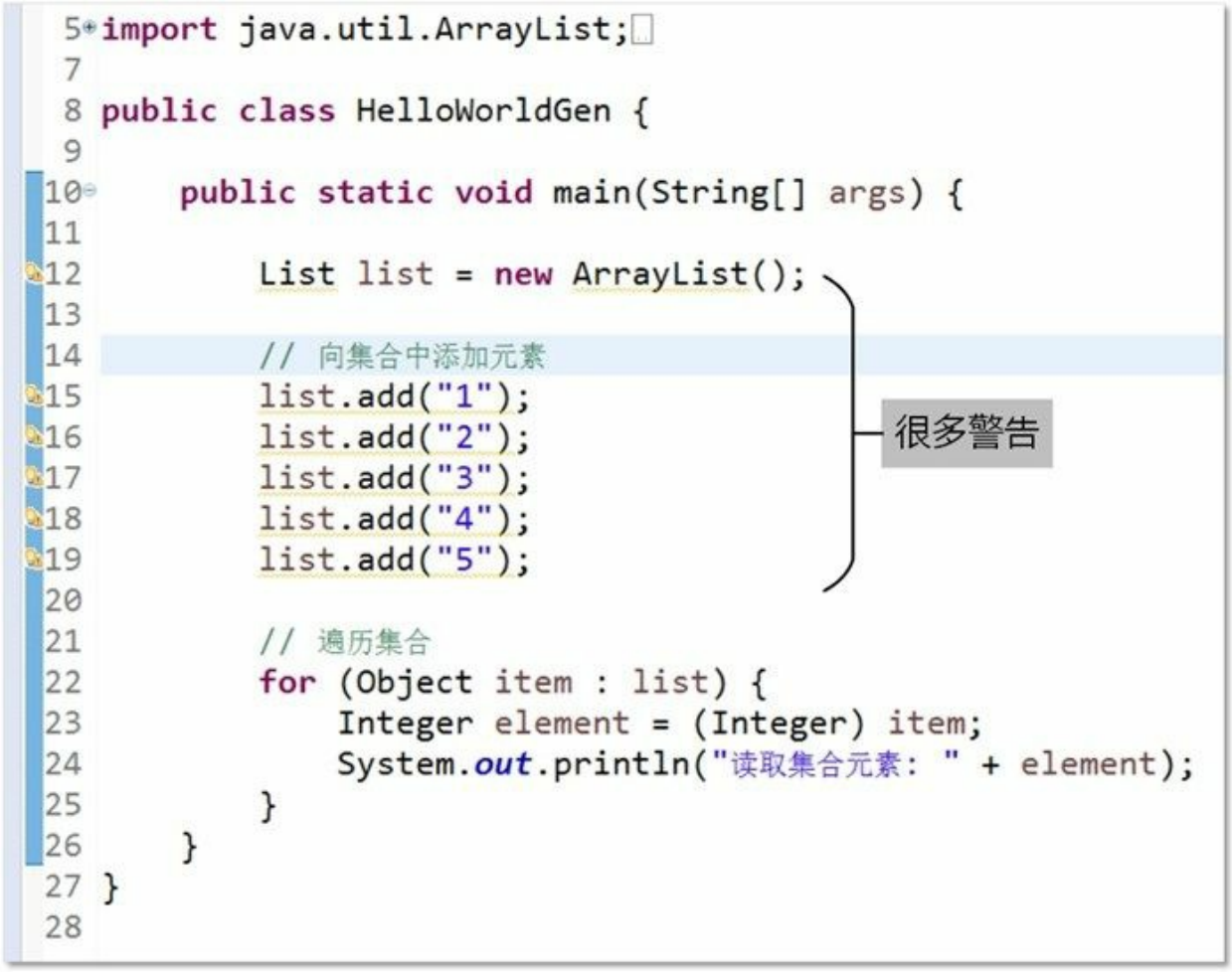


图**21-1　Eclipse**中的警告

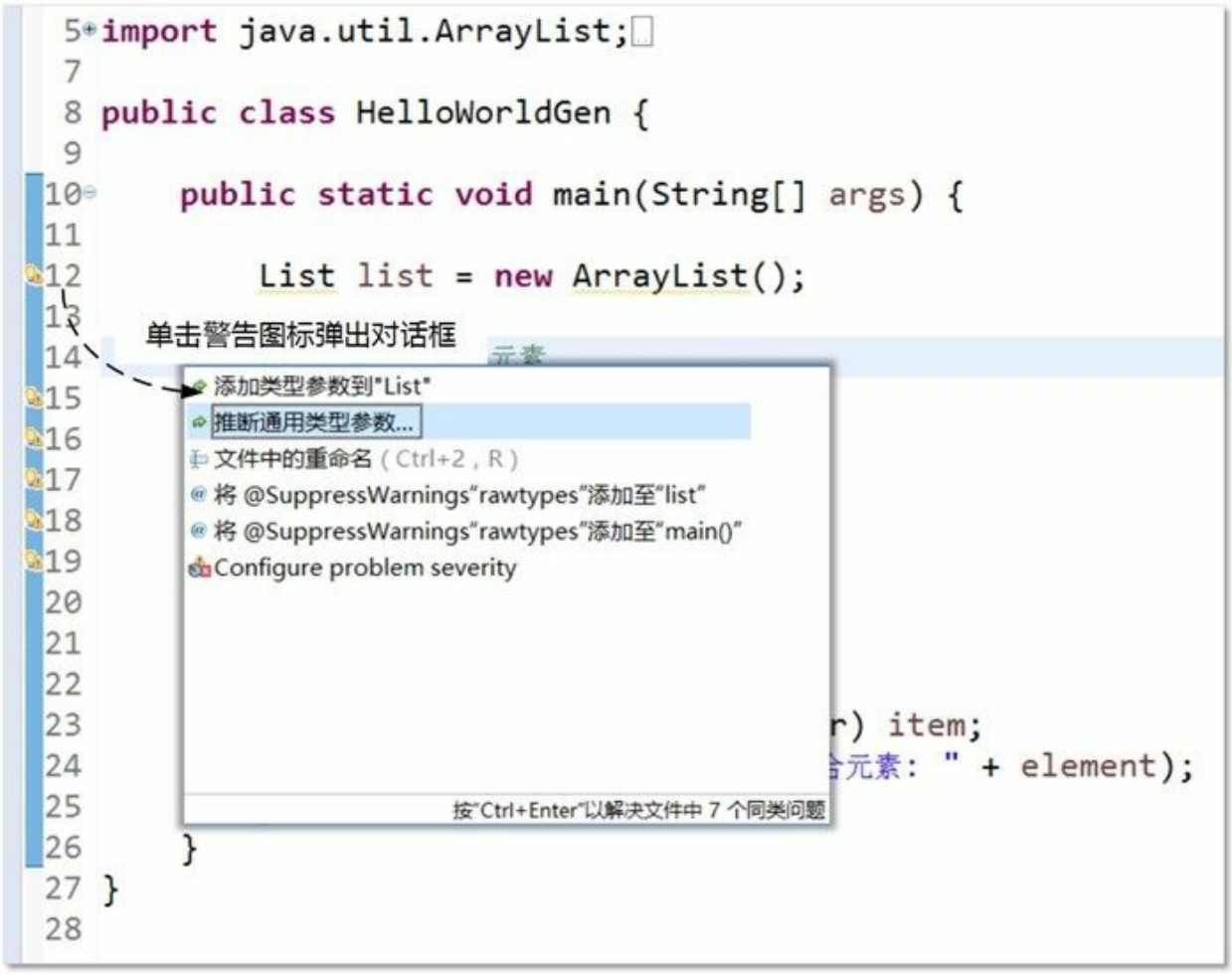


图**21-2**　使用**Eclipse**修订功能

21.2 使用泛型

泛型对于Java影响最大就是集合了，Java 5之后所有的集合类型都可以有泛型类型，可以限定存放到集合中的类型。打开图20-1类图或打开API文档，会发现集合类型后面会有<E>，如：Collection<E>、List<E>、ArrayList<E>、Set<E>和Map<K,V>，这说明这些类型是支持泛型的。尖括号中的E、K和V等是类型参数名称，它们是实际类型的占位符。

事实上第20章中所有集合示例都可以添加泛型支持，下面修改几个示例体会一下泛型的好处。下面看一个Set泛型集合示例：

```
//HelloWorldGen.java文件
... ...
// 测试Set泛型集合方法
private static void testSet() {

    Set<String> set = new HashSet<String>();              ①
    // 向集合中添加元素
    set.add("A");
    set.add("D");
    set.add("E");

    // 1.使用for-each循环遍历
    System.out.println("--1.使用for-each循环遍历--");
    for (String item : set) {                             ②
        System.out.println("读取集合元素: " + item);
    }

    // 2.使用迭代器遍历
    System.out.println("--2.使用迭代器遍历--");
    Iterator<String> it = set.iterator();                 ③
    while (it.hasNext()) {
        String item = it.next();                          ④
        System.out.println("读取集合元素: " + item);
    }
}
```

上述代码第①行Set和HashSet类型后面都指定了泛型，<String>说明实际类型是String。因为有了泛型可以保证从集合中取出元素一定是String类型，所以代码第②行声明元素类型是String。

在采用Iterator迭代器遍历集合，也需要为迭代器指定泛型，限定它的实际类型是String，见代码第③行。那么指定泛型的迭代器，在取出元素时不需要强制类型转换，见代码第④行。

再看一个Map泛型集合示例：

```
//HelloWorldGen.java文件
... ...
// 测试Map泛型集合方法
private static void testMap() {

    Map<Integer, String> map = new HashMap<Integer, String>();    ①

    map.put(102, "张三");
    map.put(105, "李四");
    map.put(109, "王五");
    map.put(110, "董六");

    // 1.使用for-each循环遍历
    System.out.println("--1.使用for-each循环遍历--");
    // 获得键集合
```

```
    Set<Integer> keys = map.keySet();                                ②
    for (Integer key : keys) {                                       ③
        String value = map.get(key);                                 ④
        System.out.printf("key=%d - value=%s \n", key, value);
    }

    // 2.使用迭代器遍历
    System.out.println("--2.使用迭代器遍历--");
    // 获得值集合
    Collection<String> values = map.values();                        ⑤
    // 遍历值集合
    Iterator<String> it = values.iterator();
    while (it.hasNext()) {
        String item = it.next();
        System.out.println("值集合元素: " + item);
    }
}
```

上述代码第①行中Map<Integer, String>是指定Map泛型集合类型，其中键集合限定Integer类型，值集合限定String类型，HashMap<Integer, String>也需要同样的泛型。代码第②行是取出Map中键集合，需要指定它的类型是Set<Integer>。代码第③行遍历键集合，其中取出的元素是Integer类型，代码第④行是从Map集合中取出值，它是String类型。这里都不需要强制类型转换，使用起来非常方便。

代码第⑤行是取出Map中的值集合，它是Collection<String>类型。迭代器的遍历过程与Set泛型集合示例类似，这里不再赘述。

21.3 自定义泛型类

根据自己的需要也可以自定义泛型类、泛型接口和带有泛型参数的方法。下面通过一个示例介绍一下泛型类。数据结构中有一种“队列”（queue）数据结构（如图21-3所示），它的特点是遵守“先入先出”（FIFO）规则。

图**21-3** 队列数据结构

虽然Java SE已经提供了支持泛型的队列java.util.Queue<E>类型，但是为了学习泛型的目的，本节中还是要介绍一个自己实现的支持泛型的队列集合。

具体实现代码如下：

```
//Queue.java文件
package com.a51work6;

import java.util.ArrayList;
import java.util.List;

/**
 * 自定义的泛型队列集合
 */
public class Queue<T> {                          ①

    // 声明保存队列元素集合items
    private List<T> items;                       ②

    // 构造方法初始化是集合items
    public Queue() {
        this.items = new ArrayList<T>();         ③
    }

    /**
     * 入队方法
     * @param item 参数需要入队的元素
     */
    public void queue(T item) {                  ④
        this.items.add(item);
    }

    /**
     * 出队方法
     * @return 返回出队元素
     */
    public T dequeue(){                          ⑤
        if (items.isEmpty()) {
            return null;
        } else {
            return this.items.remove(0);         ⑥
        }
    }

    @Override
    public String toString() {
        return items.toString();
```

```
    }

}
```

上述代码第①行定义了Queue<T>泛型类型的队列，T是参数类型占位符。代码第②行是声明一个List泛型集合成员变量items，用来保存队列中的元素。代码第③行是构造方法，初始化items成员变量。

代码第④行的queue()方法是队列入队方法，其中参数item是要入队的元素，参数类型使用占位符T表示，注意要与Queue<T>中的占位符保持一致。

代码第⑤行的dequeue()是出队方法，返回出队的那个元素，返回值类型用占位符T表示，注意要与Queue<T>中的占位符保持一致。在dequeue()方法中首先判断集合是否有元素，如果没有元素返回null；如果有元素则通过第⑥行this.items.remove(0)方法删除队列的第一个元素，并把删除的元素返回，以达到出队的目的。

提示　泛型中参数类型占位符，可以是任何大写或小写的英文字母，一般情况下人们习惯于使用字母T、E、K和U等大写英文字母，但也可以使用其他的字母。

调用队列示例代码如下：

```
//HelloWorld.java文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        Queue<String> genericQueue = new Queue<String>();          ①
        genericQueue.queue("A");
        genericQueue.queue("C");
        genericQueue.queue("B");
        genericQueue.queue("D");
        //genericQueue.queue(1);//编译错误                          ②

        System.out.println(genericQueue);
        genericQueue.dequeue();                                    ③

        System.out.println(genericQueue);

    }
}
```

输出结果如下：

```
[A, C, B, D]
[C, B, D]
```

上述代码在使用了刚刚自定义的支持泛型的队列Queue集合，使用它与使用Java SE提供的泛型集合没有什么区别。首先在代码第①行实例化Queue对象，通过尖括号指定限定的类型是String，这个队列中只能存放String类型数据。代码第②行试图向队列中添加1整数数据，则会发生编译错误。

代码第③行出队后操作，通过运行的结果可见，出队后第一个元素"A"，会从中队列中删除。

自定义泛型类时可以能会用到多个类型参数，可以使用多个不同的字母作为占位符，类似于Map<K,V>。这些需要注意程序代码中哪些地方是用K表示，哪些地方用V表示。

21.4 自定义泛型接口

自定义泛型接口与自定义泛型类类似，定义的方式完全一样。下面将21.3节的示例修改称为队列接口，代码如下：

```
//IQueue.java文件
package com.a51work6;

/**
 * 自定义的泛型队列集合
 */
public interface IQueue<T> {                    ①

    /**
     * 入队方法
     * @param item 参数需要入队的元素
     */
    public void queue(T item);                  ②

    /**
     * 出队方法
     * @return 返回出队元素
     */
    public T dequeue();                         ③

}
```

上述代码定义了支持泛型的接口。代码第①行定义了IQueue<T>泛型接口，T是参数类型占位符。该接口中声明两个方法，代码第②行的queue()方法是入队方法，参数类型使用占位符T表示的类型。代码第③行的dequeue()方法是出队方法，返回值类型是占位符T表示的类型。

实现这接口IQueue<T>具体方式有很多，可以是List、Set或Hash等多种不同方式，下面笔者给出一个基于List实现方式，代码如下：

```
//ListQueue.java文件
package com.a51work6;

import java.util.ArrayList;
import java.util.List;

/**
 * 自定义的泛型队列集合
 */
public class ListQueue<T> implements IQueue<T> {

    // 声明保存队列元素集合items
    private List<T> items;

    // 构造方法初始化是集合items
    public ListQueue() {
        this.items = new ArrayList<T>();
    }

    /**
     * 入队方法
     *
     * @param item
     *            参数需要入队的元素
     */
    @Override
```

```
    public void queue(T item) {
        this.items.add(item);
    }

    /**
     * 出队方法
     *
     * @return 返回出队元素
     */
    @Override
    public T dequeue() {
        if (items.isEmpty()) {
            return null;
        } else {
            return this.items.remove(0);
        }
    }

    @Override
    public String toString() {
        return items.toString();
    }

}
```

上述实现代码与上一节Queue<T>类很现实，只是实现了IQueue<T>接口。读者需要注意的实现泛型接口的具体类也应该支持泛型，所以Queue<T>中类型参数名要与IQueue<T>接口中的类型参数名一致，占位符所用字母相同。

21.5 泛型方法

在方法中也可以使用泛型，即方法的参数类型或返回值类型，可以用类型参数表示。假设笔者想编写一个能够比较对象大小的方法，实现代码如下：

```
//HelloWorld.java文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        System.out.println(isEquals(new Integer(1), new Integer(5)));       ①
        System.out.println(isEquals(1, 5));// 发生了自动装箱                ②
        System.out.println(isEquals(new Double(1.0), new Double(1.0)));     ③
        System.out.println(isEquals(1.0, 1.0));// 发生了自动装箱             ④
        System.out.println(isEquals("A", "A"));                             ⑤
    }

    // 限定类型参数为Number
    public static <T> boolean isEquals(T a, T b) {                          ⑥
        return a.equals(b);
    }
}
```

上述代码第⑥行是定义了比较方法isEquals()，该方法可以接收两个参数，它们是任何引用类型，返回值是<T>指定占位符为T，方法中的参数类型用T表示。

在main方法中代码第①行~第⑤行都是能够正常执行。其中代码第②行和第④行参数都是基本数据类型，它们在调用过程中发生了自动装箱，被自动转换成为对象了。

另外，泛型的类型参数也可以限定一个边界，例如比较方法isEquals()只想用于数值对象大小的比较，实现代码如下：

```
//HelloWorldLimit.java文件
package com.a51work6;

public class HelloWorldLimit {

    public static void main(String[] args) {

        System.out.println(isEquals(new Integer(1), new Integer(5)));
        System.out.println(isEquals(1, 5));// 发生了自动装箱
        System.out.println(isEquals(new Double(1.0), new Double(1.0)));
        System.out.println(isEquals(1.0, 1.0));// 发生了自动装箱
        // System.out.println(isEquals("A", "A")); //编译错误            ①

    }

    // 限定类型参数为Number
    public static <T extends Number> boolean isEquals(T a, T b) {        ②
        return a.equals(b);
    }

}
```

上述代码第②行定义泛型使用<T extends Number>语句，该语句是限定类型参数只能是Number类型。所以代码第①行的试图传递String类型的参数，则会发生编译错误。

本章小结

本章介绍了Java中的泛型技术，包括泛型概念、在集合中使用泛型、自定义泛型类、自定义泛型接口和泛型方法等。广大读者通过本章的学习应该使用泛型的优势，并且从本章之后使用集合时，尽量使用泛型集合。

第 22 章　文件管理与I/O流

程序经常需要访问文件和目录，读取文件信息或写入信息到文件，在Java语言中对文件的读写是通过I/O流技术实现的。本章先介绍文件管理，然后再介绍I/O流。

22.1 文件管理

Java语言使用File类对文件和目录进行操作，查找文件时需要实现FilenameFilter或FileFilter接口。另外，读写文件内容可以通过FileInputStream、FileOutputStream、FileReader和FileWriter类实现，它们属于I/O流，下一节会详细介绍I/O流。这些类和接口全部来源于java.io包。

22.1.1 File类

File类表示一个与平台无关的文件或目录。File类名很有欺骗性，初学者会误认为是File对象只是一个文件，但它也可能是一个目录。

File类中常用的方法如下。

01. 构造方法

    - File(String path)：如果path是实际存在的路径，则该File对象表示的是目录；如果path是文件名，则该File对象表示的是文件。
    - File(String path, String name)：path是路径名，name是文件名。
    - File(File dir, String name)：dir是路径对象，name是文件名。

02. 获得文件名

    - String getName( )：获得文件的名称，不包括路径。
    - String getPath( )：获得文件的路径。
    - String getAbsolutePath( )：获得文件的绝对路径。
    - String getParent( )：获得文件的上一级目录名。

03. 文件属性测试

    - boolean exists( )：测试当前File对象所表示的文件是否存在。
    - boolean canWrite( )：测试当前文件是否可写。
    - boolean canRead( )：测试当前文件是否可读。
    - boolean isFile( )：测试当前文件是否是文件。
    - boolean isDirectory( )：测试当前文件是否是目录。

04. 文件操作

    - long lastModified( )：获得文件最近一次修改的时间。
    - long length( )：获得文件的长度，以字节为单位。
    - boolean delete( )：删除当前文件。成功返回 true，否则返回false。
    - boolean renameTo(File dest)：将重新命名当前File对象所表示的文件。成功返回 true，否则返回false。

05. 目录操作

- boolean mkdir( )：创建当前File对象指定的目录。
- String[] list()：返回当前目录下的文件和目录，返回值是字符串数组。
- String[] list(FilenameFilter filter)：返回当前目录下满足指定过滤器的文件和目录，参数是实现FilenameFilter接口对象，返回值是字符串数组。
- File[] listFiles()：返回当前目录下的文件和目录，返回值是File数组。
- File[] listFiles(FilenameFilter filter)：返回当前目录下满足指定过滤器的文件和目录，参数是实现FilenameFilter接口对象，返回值是File数组。
- File[] listFiles(FileFilter filter)：返回当前目录下满足指定过滤器的文件和目录，参数是实现FileFilter接口对象，返回值是File数组。

对目录操作有两个过滤器接口：FilenameFilter和FileFilter。它们都只有一个抽象方法accept，FilenameFilter接口中的accept方法如下：

- boolean accept(File dir, String name)：测试指定dir目录中是否包含文件名为name的文件。

FileFilter接口中的accept方法如下：

- boolean accept(File pathname)：测试指定路径名是否应该包含在某个路径名列表中。

注意　路径中会用到路径分隔符，路径分隔符在不同平台上是有区别的，UNIX、Linux和macOS中使用正斜杠“/”，而Windows下使用反斜杠“\”。Java是支持两种写法，但是反斜杠“\”属于特殊字符，前面需要加转义符。例如C:\Users\a.java在程序代码中应该使用C:\\Users\\a.java表示，或表示为C:/Users/a.java也可以。

22.1.2　案例：文件过滤

为熟悉文件操作，本节介绍一个案例，该案例从指定的目录中列出文件信息。代码如下：

```
//HelloWorld.java文件
package com.a51work6;

import java.io.File;
import java.io.FilenameFilter;

public class HelloWorld {

    public static void main(String[] args) {

        // 用File对象表示一个目录,.表示当前目录
        File dir = new File("./TestDir");                                    ①
        // 创建HTML文件过滤器
        Filter filter = new Filter("html");                                  ②

        System.out.println("HTML文件目录：" + dir);
        // 列出目录TestDir下，文件后缀名为HTML的所有文件
        String files[] = dir.list(filter); //dir.list();
        // 遍历文件列表
        for (String fileName : files) {
            // 为目录TestDir下的文件或目录创建File对象
            File f = new File(dir, fileName);
            // 如果该f对象是文件，则打印文件名
            if (f.isFile()) {
```

```
                System.out.println("文件名：" + f.getName());
                System.out.println("文件绝对路径：" + f.getAbsolutePath());
                System.out.println("文件路径：" + f.getPath());
            } else {
                System.out.println("子目录：" + f);
            }
        }

    }
}

// 自定义基于文件扩展名的文件过滤器
class Filter implements FilenameFilter {                                    ③

    // 文件扩展名
    String extent;

    // 构造方法
    Filter(String extent) {
        this.extent = extent;
    }

    @Override
    public boolean accept(File dir, String name) {                          ④
        // 测试文件扩展名是否为extent所指定的
        return name.endsWith("." + extent);
    }
}
```

上述代码第①行创建TestDir目录对象，"./TestDir"表示当前目录下的TestDir目录，还可以表示为".\\TestDir"和"TestDir"。

提示　在编程时尽量使用相对路径，尽量不要使用绝对路径。"./TestDir"就是相对路径，相对路径中会用到点“.”，在目录中一个点“.”表示当前目录，两个点表示“..”表示父目录。

注意　在Eclipse工具中运行的Java程序，那么当前目录在哪里呢？例如"./TestDir"表示当前目录下的TestDir子目录，那么应该在哪里创建TestDir目录呢？在Eclipse中当前目录就是工程的根目录，如图22-1所示，当前目录是Eclipse工程根目录，子目录TestDir位于工程根目录下。

图**22-1　Eclipse**中的当前目录

上述代码第②行创建针对HTML文件过滤器Filter，Filter类要求实现FilenameFilter接口，见代码第⑤行。FilenameFilter接口要求实现抽象方法accept，见代码第④行，在该方法中通过判断文件名是否指定的扩展名结尾则返回true，否则返回false。

22.2 I/O流概述

Java将数据的输入输出（I/O）操作当作“流”来处理，“流”是一组有序的数据序列。“流”分为两种形式：输入流和输出流，从数据源中读取数据是输入流，将数据写入到目的地是输出流。

提示　以CPU为中心，从外部设备读取数据到内存，进而再读入到CPU，这是输入（Input，缩写I）过程；将内存中的数据写入到外部设备，这是输出（Output，缩写O）过程。所以输入输出简称为I/O。

22.2.1 Java流设计理念

如图22-2所示，数据输入的数据源有多种形式，如文件、网络和键盘等，键盘是默认的标准输入设备。而数据输出的目的地也有多种形式，如文件、网络和控制台，控制台是默认的标准输出设备。

图22-2 I/O流

所有的输入形式都抽象为输入流，所有的输出形式都抽象为输出流，它们与设备无关。

22.2.2 流类继承层次

以字节为单位的流称为字节流，以字符为单位的流称为字符流。Java SE提供4个顶级抽象类，两个字节流抽象类：InputStream和OutputStream；两个字符流抽象类：Reader和Writer。

01. 字节输入流

字节输入流根类是InputStream，如图22-3所示它有很多子类，这些类的说明如表22-1所示。

表 22-1　主要的字节输入流

类	描述
FileInputStream	文件输入流
ByteArrayInputStream	面向字节数组的输入流
PipedInputStream	管道输入流，用于两个线程之间的数据传递
FilterInputStream	过滤输入流，它是一个装饰器扩展其他输入流
BufferedInputStream	缓冲区输入流，它是 FilterInputStream 的子类
DataInputStream	面向基本数据类型的输入流

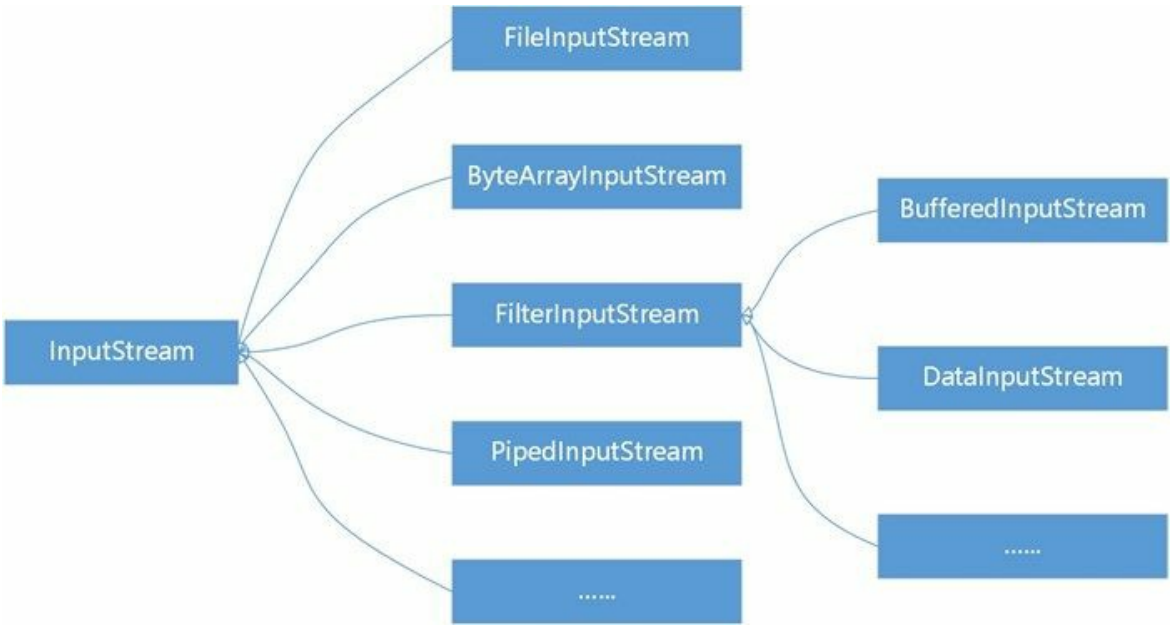


图**22-3**　字节输入流类继承层次

02. 字节输出流

字节输出流根类是OutputStream，如图22-4所示它有很多子类，这些类的说明如表22-2所示。

表 **22-2**　主要的字节输出流

类	描述
FileOutputStream	文件输出流
ByteArrayOutputStream	面向字节数组的输出流
PipedOutputStream	管道输出流，用于两个线程之间的数据传递
FilterOutputStream	过滤输出流，它是一个装饰器扩展其他输出流
BufferedOutputStream	缓冲区输出流，它是 FilterOutputStream 的子类
DataOutputStream	面向基本数据类型的输出流

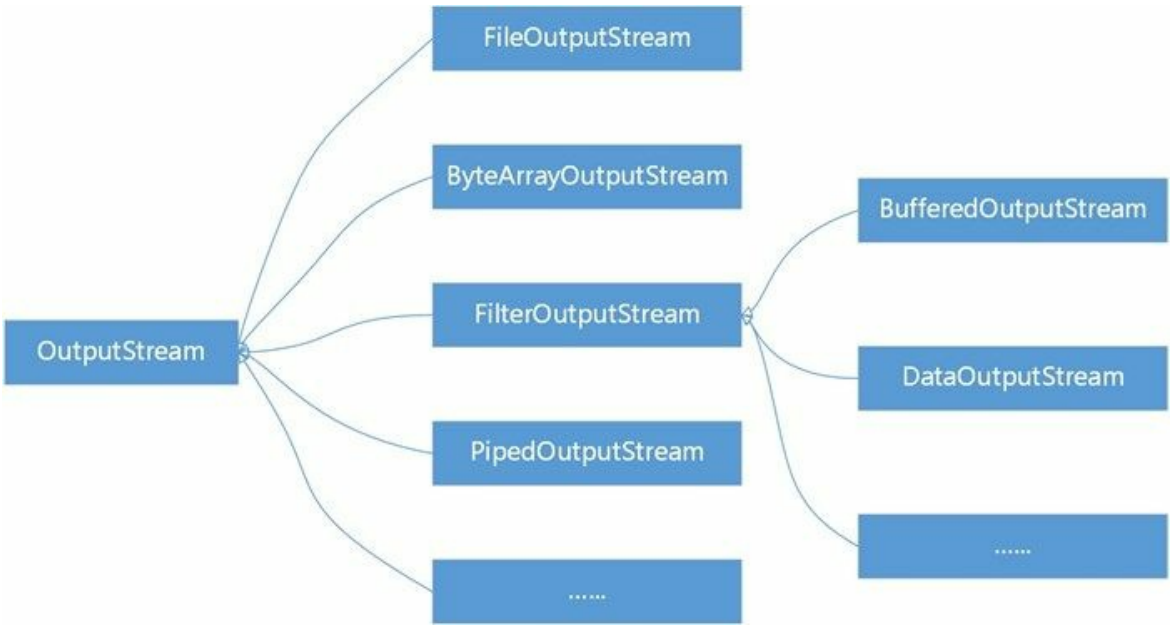


图**22-4**　字节输出流类继承层次

03. 字符输入流

字符输入流根类是Reader，这类流以16位的Unicode编码表示的字符为基本处理单位。如图22-5所示它有很多子类，这些类的说明如表22-3所示。

表 **22-3**　主要的字符输入流

类	描述
FileReader	文件输入流
CharArrayReader	面向字符数组的输入流
PipedReader	管道输入流，用于两个线程之间的数据传递
FilterReader	过滤输入流，它是一个装饰器扩展其他输入流
BufferedReader	缓冲区输入流，它是也是装饰器，它不是 FilterReader 的子类
InputStreamReader	把字节流转换为字符流，它是也一个装饰器，是 FileReader 的父类

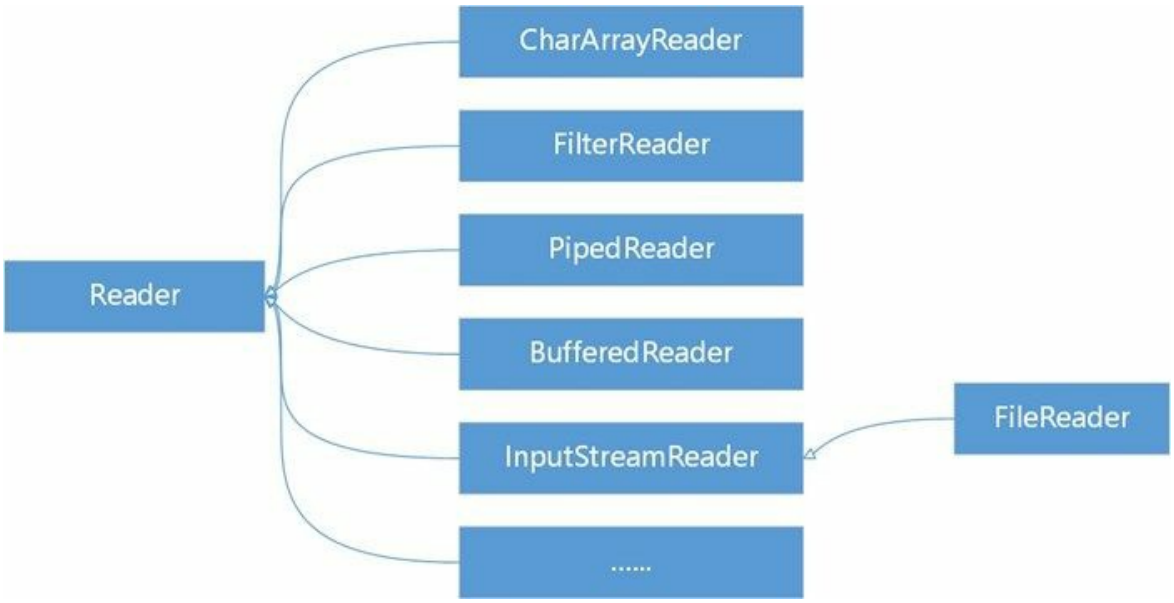


图**22-5**　字符输入流类继承层次

04. 字符输出流

字符输出流根类是Writer，这类流以16位的Unicode编码表示的字符为基本处理单位。如图22-6所示它有很多子类，这些类的说明如表22-4所示。

表 **22-4**　主要的字符输出流

类	描述
FileWriter	文件输出流
CharArrayWriter	面向字符数组的输出流
PipedWriter	管道输出流，用于两个线程之间的数据传递
FilterWriter	过滤输出流，它是一个装饰器扩展其他输出流
BufferedWriter	缓冲区输出流，它是也是装饰器，它不是 FilterWriter的子类
OutputStreamWriter	把字节流转换为字符流，它是也一个装饰器，是 FileWriter 的父类

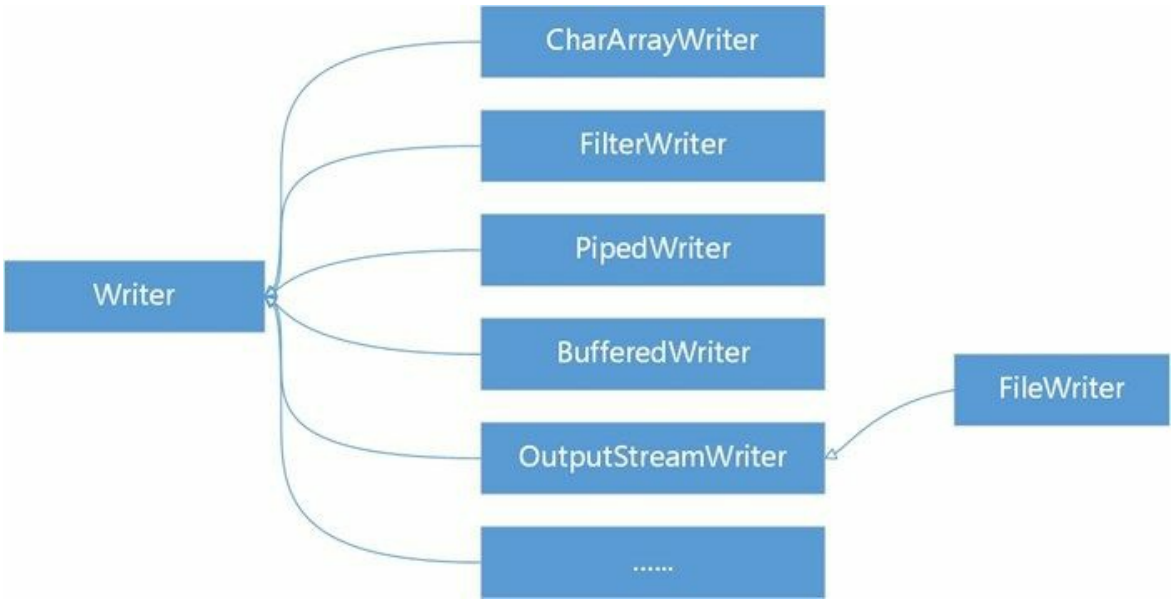


图**22-6**　字符输出流类继承层次

22.3 字节流

上一节总体概述了Java中I/O流层次结构技术，本节详细介绍一下字节流的API。掌握字节流的API先要熟悉它的两个抽象类：InputStream 和OutputStream，了解它们有哪些主要的方法。

22.3.1 InputStream抽象类

InputStream是字节输入流的根类，它定义了很多方法，影响着字节输入流的行为。下面详细介绍一下。

InputStream主要方法如下：

- int read()：读取一个字节，返回0到255范围内的int字节值。如果已经到达流末尾，而且没有可用的字节，则返回值-1。
- int read(byte b[])：读取多个字节，数据放到字节数组b中，返回值为实际读取的字节的数量，如果已经到达流末尾，而且没有可用的字节，则返回值-1。
- int read(byte b[ ], int off, int len)：最多读取len个字节，数据放到以下标off开始字节数组b中，将读取的第一个字节存储在元素b[off]中，下一个存储在b[off+1]中，依次类推。返回值为实际读取的字节的数量，如果已经到达流末尾，而且没有可用的字节，则返回值-1。
- void close()：流操作完毕后必须关闭。

上述所有方法都可能会抛出IOException，因此使用时要注意处理异常。

22.3.2 OutputStream抽象类

OutputStream是字节输出流的根类，它定义了很多方法，影响着字节输出流的行为。下面详细介绍一下。

OutputStream主要方法如下：

- void write(int b)：将b写入到输出流，b是int类型占有32位，写入过程是写入b 的8个低位，b的24个高位将被忽略。
- void write(byte b[ ])：将b.length个字节从指定字节数组b写入到输出流。
- void write(byte b[ ], int off, int len)：把字节数组b中从下标off开始，长度为len的字节写入到输出流。
- void flush()：刷空输出流，并输出所有被缓存的字节。由于某些流支持缓存功能，该方法将把缓存中所有内容强制输出到流中。
- void close( )：流操作完毕后必须关闭。

上述所有方法都声明抛出IOException，因此使用时要注意处理异常。

注意　流（包括输入流和输出流）所占用的资源，不能通过JVM的垃圾收集器回收，需要程序员自己释放。一种方法是可以在finally代码块调用close()方法关闭流，释放流所占用的资源。另一种方法通过自动资源管理技术管理这些流，流（包括输入流和输出流）都实现了AutoCloseable接口，可以使用自动资源管理技术，具体内容参考19.4.2节。

22.3.3 案例：文件复制

前面介绍了两种字节流常用的方法，下面通过一个案例熟悉一下它们的使用，该案例实现了文件复制，数据源是文件，所以会用到文件输入流FileInputStream，数据目的地也是文件，所以会用到文件输出流FileOutputStream。

FileInputStream和FileOutputStream中主要方法都是继承自InputStream和OutputStream前面两个节已经详细介绍了，这里不再赘述。下面介绍一下它们的构造方法，FileInputStream构造方法主要有：

- FileInputStream(String name)：创建FileInputStream对象，name是文件名。如果文件不存在则抛出FileNotFoundException异常。
- FileInputStream(File file)：通过File对象创建FileInputStream对象。如果文件不存在则抛出FileNotFoundException异常。

FileOutputStream构造方法主要有：

- FileOutputStream(String name)：通过指定name文件名创建FileOutputStream对象。如果name文件存在，但如果是一个目录或文件无法打开则抛出FileNotFoundException异常。
- FileOutputStream(String name, boolean append)：通过指定name文件名创建FileOutputStream对象，append参数如果为 true，则将字节写入文件末尾处，而不是写入文件开始处。如果name文件存在，但如果是一个目录或文件无法打开则抛出FileNotFoundException异常。
- FileOutputStream(File file)：通过File对象创建FileOutputStream对象。如果file文件存在，但如果是一个目录或文件无法打开则抛出FileNotFoundException异常。
- FileOutputStream(File file, boolean append)：通过File对象创建FileOutputStream对象，append参数如果为 true，则将字节写入文件末尾处，而不是写入文件开始处。如果file文件存在，但如果是一个目录或文件无法打开则抛出FileNotFoundException异常。

下面介绍如果将./TestDir/build.txt文件内容复制到./TestDir/subDir/build.txt。./TestDir/build.txt文件内容是AI-162.3764568，实现代码如下：

```
//FileCopy.java文件
package com.a51work6;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

public class FileCopy {

    public static void main(String[] args) {

        try (FileInputStream in = new FileInputStream("./TestDir/build.txt");
                FileOutputStream out = new FileOutputStream("./TestDir/subDir/build.txt")) {    ①

            // 准备一个缓冲区
            byte[] buffer = new byte[10];                    ②
            // 首先读取一次
            int len = in.read(buffer);                       ③

            while (len != -1) {                              ④
                String copyStr = new String(buffer);         ⑤
                // 打印复制的字符串
                System.out.println(copyStr);
                // 开始写入数据
                out.write(buffer, 0, len);                   ⑥
                // 再读取一次
                len = in.read(buffer);                       ⑦
```

```
                }

            } catch (FileNotFoundException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
```

控制台输出结果：

```
AI-162.376
456862.376
```

上述代码第①行创建FileInputStream和FileOutputStream对象，这是自动资源管理的写法，不需要自己关闭流。

第②行代码是准备一个缓冲区，它是字节数组，读取输入流的数据保存到缓冲区中，然后将缓冲区中的数据再写入到输出流中。

> 提示　缓冲区大小（字节数组长度）多少合适？缓冲区大小决定了一次读写操作的最多字节数，缓冲区设置的很小，会进行多次读写操作才能完成。所以如果当前计算机内存足够大，而不影响其它应用运行情况下，当然缓冲区是越大越好。本例中缓冲区大小设置的10，源文件中内容是AI-162.3764568，共有14个字符，由于这些字符都属于ASCII字符，因此14个字符需要14字节描述，需要读写两次才能完成复制。

代码第③行是第一次从输入流中读取数据，数据保存到buffer中，len是实际读取的字节数。代码第⑦行也从输入流中读取数据。由于本例中缓冲区大小设置为10，因此这两次读取数据会把数据读完，第一次读了10个字节，第二次读了4个字节。

代码第④行是判断读取的字节数len是否等于-1，代码第⑦行的len = in.read(buffer)事实上执行了两次，第一次执行时len为4，第二次执行时len为-1。

代码第⑤行是使用字节数组构造字符串，然后通过System.out.println(copyStr)语句将字符串输出到控制台。从输出的结果看输出了两次，每次10个字节，第一次输出结果AI-162.376容易理解，它是AI-162.3764568的前10个字符；那么第二次输出的结果456862.376令人匪夷所思，事实上前4个字符（4568）是第二次读取的，后面的6个字符（62.376）是上一次读取的。两次读取内容如图22-7所示。

图**22-7**　文件读取示意图

代码第⑥行out.write(buffer, 0, len)是向输出流写入数据，与读取数据对应，数据写入也调用了两次，第一次len为10，将缓冲区buffer所有元素全部写入输出流；第二次len为4，将缓冲区buffer所有前4个元素写入输出流。注意这里不要使用void write(byte b[ ])方法，因为它没法控制第二次写入的字节数。

上面的案例由于使用了字节输入输出流，所以不仅可以复制文本文件，还可以复制二进制文件。

22.3.4　使用字节缓冲流

BufferedInputStream和BufferedOutputStream称为字节缓冲流，使用字节缓冲流内置了一个缓冲区，第一次调用read方法时尽可能多地从数据源读取数据到缓冲区，后续再到用read方法时先看看缓冲区中是否有数据，如果有则读缓冲区中的数据，如果没有再将数据源中的数据读入到缓冲区，这样可以减少直接读数据源的次数。通过输出流调用write方法写入数据时，也先将数据写入到缓冲区，缓冲区满了之后再写入数据目的地，这样可以减少直接对数据目的地写入次数。使用了缓冲字节流可以减少I/O操作次数，提高效率。

从图22-3和图22-4可见，BufferedInputStream的父类是FilterInputStream，BufferedOutputStream的父类是FilterOutputStream，FilterInputStream和FilterOutputStream称为过滤流。过滤流的作用是扩展其他流，增强其功能。那么BufferedInputStream和BufferedOutputStream增强了缓冲能力。

> 提示　过滤流实现了装饰器（Decorator）设计模式，这种设计模式能够在运行时扩充一个类的功能。而继承在编译时扩充一个类的功能。

BufferedInputStream和BufferedOutputStream中主要方法都是继承自InputStream和OutputStream前面两个节已经详细介绍了，这里不再赘述。下面介绍一下它们的构造方法，BufferedInputStream构造方法主要有：

- BufferedInputStream(InputStream in)：通过一个底层输入流in对象创建缓冲流对象，缓冲区大小是默认的，默认值8192。
- BufferedInputStream(InputStream in, int size)：通过一个底层输入流in对象创建缓冲流对象，size指定的缓冲区大小，缓冲区大小应该是2的n次幂，这样可提供缓冲区的利用率。

BufferedOutputStream构造方法主要有：

- BufferedOutputStream(OutputStream out)：通过一个底层输出流out 对象创建缓冲流对象，缓冲区大小是默认的，默认值8192。

- BufferedOutputStream(OutputStream out, int size)：通过一个底层输出流out对象创建缓冲流对象，size指定的缓冲区大小，缓冲区大小应该是2的n次幂，这样可提高缓冲区的利用率。

下面将22.3.3节的文件复制的案例改造成缓冲流实现，代码如下：

```
//FileCopyWithBuffer.java文件
package com.a51work6;

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

public class FileCopyWithBuffer {

    public static void main(String[] args) {

        try (FileInputStream fis = new FileInputStream("./TestDir/src.zip");              ①
             BufferedInputStream bis = new BufferedInputStream(fis);                      ②
             FileOutputStream fos = new FileOutputStream("./TestDir/subDir/src.zip");     ③
             BufferedOutputStream bos = new BufferedOutputStream(fos)) {                  ④

            //开始时间
            long startTime = System.nanoTime();                                           ⑤
            // 准备一个缓冲区
            byte[] buffer = new byte[1024];                                               ⑥
            // 首先读取一次
            int len = bis.read(buffer);

            while (len != -1) {
                // 开始写入数据
                bos.write(buffer, 0, len);
                // 再读取一次
                len = bis.read(buffer);
            }

            //结束时间
            long elapsedTime = System.nanoTime() - startTime;                             ⑦
            System.out.println("耗时：" + (elapsedTime / 1000000.0) + " 毫秒");

        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

上述代码第①行是创建文件输入流，它是一个底层流，通过它构造缓冲输入流，见代码第②行。同理，代码第④行是构造缓冲输出流。

为了记录复制过程所耗费的时间，在复制之前获取当前系统时间，见代码第⑤行，System.nanoTime()是获得当前系统时间，单位是纳秒。在复制结束之后同样获取系统时间，代码第⑦行用结束时的系统时间减去复制之前的系统时间，elapsedTime就是耗时了，但是它的单位是纳秒，需要除以10^6才是毫秒。

提示　在程序代码第⑥行也指定了缓冲区buffer，这个缓冲区与缓冲流内置缓冲区不同，决定是否进行I/O操作次数的是缓冲流内置缓冲区，不是这个缓冲区。

为了比较，可以将22.3.3节的案例也添加耗时输出功能，代码如下：

```
//FileCopy.java文件
package com.a51work6;
...
public class FileCopy {

    public static void main(String[] args) {

        try (FileInputStream in = new FileInputStream("./TestDir/src.zip");
                FileOutputStream out = new FileOutputStream("./TestDir/subDir/src.zip")) {

            //开始时间，当前系统纳秒时间
            long startTime = System.nanoTime();
            // 准备一个缓冲区
            byte[] buffer = new byte[1024];
            // 首先读取一次
            int len = in.read(buffer);

            while (len != -1) {
                // 开始写入数据
                out.write(buffer, 0, len);
                // 再读取一次
                len = in.read(buffer);
            }

            //结束时间，当前系统纳秒时间
            long elapsedTime = System.nanoTime() - startTime;
            System.out.println("耗时：" + (elapsedTime / 1000000.0) + " 毫秒");

        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

FileCopy与FileCopyWithBuffer复制相同文件src.zip，缓冲区buffer都设置1024，那么运行的结果：

```
FileCopyWithBuffer耗时：94.927181 毫秒
FileCopy耗时：206.087523 毫秒
```

可能每次运行稍有不同，但是可以看出它们的差别了，使用缓冲流的FileCopyWithBuffer明显要比不使用缓冲流的FileCopy速度快。

22.4 字符流

上一节介绍了字节流，本节详细介绍一下字符流的API。掌握字符流的API先要熟悉它的两个抽象类：Reader和Writer，了解它们有哪些主要的方法。

22.4.1 Reader抽象类

Reader是字符输入流的根类，它定义了很多方法，影响着字符输入流的行为。下面详细介绍一下。

Reader主要方法如下：

- int read()：读取一个字符，返回值范围在0~65535(0x00~0xffff)之间。如果因为已经到达流末尾，则返回值-1。
- int read(char[] cbuf)：将字符读入到数组cbuf中，返回值为实际读取的字符的数量，如果因为已经到达流末尾，则返回值-1。
- int read(char[] cbuf, int off, int len)：最多读取len个字符，数据放到以下标off开始字符数组cbuf中，将读取的第一个字符存储在元素cbuf[off]中，下一个存储在cbuf[off+1]中，依次类推。返回值为实际读取的字符的数量，如果因为已经到达流末尾，则返回值-1。
- void close()：流操作完毕后必须关闭。

上述所有方法都声明了抛出IOException，因此使用时要注意处理异常。

22.4.2 Writer抽象类

Writer是字符输出流的根类，它定义了很多方法，影响着字符输出流的行为。下面详细介绍一下。

Writer主要方法如下：

- void write(int c)：将整数值为c的字符写入到输出流，c是int类型占有32位，写入过程是写入c的16个低位，c的16个高位将被忽略。
- void write(char[] cbuf)：将字符数组cbuf写入到输出流。
- void write(char[] cbuf, int off, int len)：把字符数组cbuf中从下标off开始，长度为len的字符写入到输出流。
- void write(String str)：将字符串str中的字符写入输出流。
- void write(String str,int off,int len)：将字符串str 中从索引off开始处的len个字符写入输出流。
- void flush()：刷空输出流，并输出所有被缓存的字符。由于某些流支持缓存功能，该方法将把缓存中所有内容强制输出到流中。
- void close( )：流操作完毕后必须关闭。

上述所有方法都可以会抛出IOException，因此使用时要注意处理异常。

注意　Reader和Writer都实现了AutoCloseable接口，可以使用自动资源管理技术自动关闭它们。

22.4.3 案例：文件复制

前面两节介绍了字符流常用的方法，下面通过一个案例熟悉一下它们的使用，该案例实现了文件复

制，数据源是文件，所以会用到文件输入流FileReader，数据目的地也是文件，所以会用到文件输出流FileWriter。

FileReader和FileWriter中主要方法都是继承自Reader和Writer前面两个节已经详细介绍了，这里不再赘述。下面介绍一下它们的构造方法，FileReader构造方法主要有：

- FileReader(String fileName)：创建FileReader对象，fileName是文件名。如果文件不存在则抛出FileNotFoundException异常。
- FileReader(File file)：通过File对象创建FileReader对象。如果文件不存在则抛出FileNotFoundException异常。

FileWriter构造方法主要有：

- FileWriter(String fileName)：通过指定fileName文件名创建FileWriter对象。如果fileName文件存在，但如果是一个目录或文件无法打开则抛出FileNotFoundException异常。
- FileWriter(String fileName, boolean append)：通过指定fileName文件名创建FileWriter对象，append参数如果为 true，则将字符写入文件末尾处，而不是写入文件开始处。如果fileName文件存在，但如果是一个目录或文件无法打开则抛出FileNotFoundException异常。
- FileWriter(File file)：通过File对象创建FileWriter对象。如果file文件存在，但如果是一个目录或文件无法打开则抛出FileNotFoundException异常。
- FileWriter(File file, boolean append)：通过File对象创建FileWriter对象，append参数如果为 true，则将字符写入文件末尾处，而不是写入文件开始处。如果file文件存在，但如果是一个目录或文件无法打开则抛出FileNotFoundException异常。

注意　字符文件流只能复制文本文件，不能是二进制文件。

下面采用文件字符流重新实现22.3.3节文件复制案例，代码如下：

```
//FileCopy.java文件
package com.a51work6;

import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class FileCopy {

    public static void main(String[] args) {

        try (FileReader in = new FileReader("./TestDir/build.txt");
                FileWriter out = new FileWriter("./TestDir/subDir/build.txt")) {

            // 准备一个缓冲区
            char[] buffer = new char[10];
            // 首先读取一次
            int len = in.read(buffer);

            while (len != -1) {
                String copyStr = new String(buffer);
                // 打印复制的字符串
                System.out.println(copyStr);
                // 开始写入数据
                out.write(buffer, 0, len);
                // 再读取一次
                len = in.read(buffer);
```

```
                }

            } catch (FileNotFoundException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
```

控制台输出结果：

```
AI-162.376
456862.376
```

上述代码与22.3.3节非常相似，只是将文件输入流改为FileReader，文件输出流改为FileWriter，缓冲区使用的是字符数组。

22.4.4 使用字符缓冲流

BufferedReader和BufferedWriter称为字符缓冲流。BufferedReader特有方法和构造方法有：

- String readLine()：读取一个文本行，如果因为已经到达流末尾，则返回值null。
- BufferedReader(Reader in)：构造方法，通过一个底层输入流in对象创建缓冲流对象，缓冲区大小是默认的，默认值8192。
- BufferedReader(Reader in, int size)：构造方法，通过一个底层输入流in对象创建缓冲流对象，size指定的缓冲区大小，缓冲区大小应该是2的n次幂，这样可提高缓冲区的利用率。

BufferedWriter特有方法和构造方法主要有：

- void newLine()：写入一个换行符。
- BufferedWriter(Writerout)：构造方法，通过一个底层输出流out 对象创建缓冲流对象，缓冲区大小是默认的，默认值8192。
- BufferedWriter(Writerout, int size)：构造方法，通过一个底层输出流out对象创建缓冲流对象，size指定的缓冲区大小，缓冲区大小应该是2的n次幂，这样可提高缓冲区的利用率。

下面将22.4.3节的文件复制的案例改造成缓冲流实现，代码如下：

```
//FileCopyWithBuffer.java文件
package com.a51work6;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class FileCopyWithBuffer {

    public static void main(String[] args) {

        try (FileReader fis = new FileReader("./TestDir/JButton.html");
```

```
                BufferedReader bis = new BufferedReader(fis);
                FileWriter fos = new FileWriter("./TestDir/subDir/JButton.html");
                BufferedWriter bos = new BufferedWriter(fos)) {

            // 首先读取一行文本
            String line = bis.readLine();                ①

            while (line != null) {
                // 开始写入数据
                bos.write(line);                    ②
                //写一个换行符
                bos.newLine();                    ③
                // 再读取一行文本
                line = bis.readLine();
            }
            System.out.println("复制完成");
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

上述代码第①行是通过字节缓冲流readLine方法读取一行文本，当读取是文本为null时说明流已经读完了。代码第②行是写入文本到输出流，由于在输入流的readLine方法会丢掉一个换行符或回车符，为了保持复制结果完全一样，因此需要在写完一个文本后，调用输出流的newLine方法写入一个换行符。

22.4.5 字节流转换字符流

有时需要将字节流转换为字符流，InputStreamReader和OutputStreamWriter是为实现这种转换而设计的。

InputStreamReader构造方法如下：

- InputStreamReader(InputStream in)：将字节流in转换为字符流对象，字符流使用默认字符集。
- InputStreamReader(InputStream in, String charsetName)：将字节流in转换为字符流对象，charsetName指定字符流的字符集，字符集主要有：US-ASCII、ISO-8859-1、UTF-8和UTF-16。如果指定的字符集不支持会抛出UnsupportedEncodingException异常。

OutputStreamWriter构造方法如下：

- OutputStreamWriter(OutputStream out)：将字节流out转换为字符流对象，字符流使用默认字符集。
- OutputStreamWriter(OutputStream out,String charsetName)：将字节流out转换为字符流对象，charsetName指定字符流的字符集，如果指定的字符集不支持会抛出UnsupportedEncodingException异常。

下面将22.4.3节的文件复制的案例改造成缓冲流实现，代码如下：

```
//FileCopyWithBuffer.java文件
package com.a51work6;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
```

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;

public class FileCopyWithBuffer {

    public static void main(String[] args) {

        try ( // 创建字节文件输入流对象
                FileInputStream fis = new FileInputStream("./TestDir/JButton.html");    ①
                // 创建转换流对象
                InputStreamReader isr = new InputStreamReader(fis);
                // 创建字符缓冲输入流对象
                BufferedReader bis = new BufferedReader(isr);

                // 创建字节文件输出流对象
                FileOutputStream fos = new FileOutputStream("./TestDir/subDir/JButton.html");
                // 创建转换流对象
                OutputStreamWriter osw = new OutputStreamWriter(fos);
                // 创建字符缓冲输出流对象
                BufferedWriter bos = new BufferedWriter(osw)) {                          ②

            // 首先读取一行文本
            String line = bis.readLine();

            while (line != null) {
                // 开始写入数据
                bos.write(line);
                // 写一个换行符
                bos.newLine();
                // 再读取一行文本
                line = bis.readLine();
            }
            System.out.println("复制完成");
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

上述代码第①行~第②行只是一条语句，将这6个流放到try (…)，由JVM自动管理关闭。上述流从一个文件字节流，构建转换流，再构建缓冲流，这个过程比较麻烦，在I/O流开发过程中经常遇到这种流的“链条”。

本章小结

本章主要介绍了Java文件管理和I/O流技术。读者需要熟悉File类使用。读者还需要掌握字节流两个根类：InputStream和OutputStream，还有字符流的两个根类：Reader和Writer。了解一个常用的装饰器流，如：InputStreamReader、OutputStreamWriter、BufferedReader、BufferedWriter、BufferedInputStream和BufferedOutputStream等。

第 **23** 章　多线程编程

无论PC（个人计算机）还是智能手机现在都支持多任务，都能够编写并发访问程序。多线程编程可以编写并发访问程序。本章介绍多线程编程。

23.1 基础知识

那么线程究竟是什么？在Windows操作系统出现之前，PC上的操作系统都是单任务系统，只有在大型计算机上才具有多任务和分时设计。随着Windows、Linux等操作系统出现，把原本只在大型计算机才具有的优点，带到了PC系统中。

23.1.1 进程

一般可以在同一时间内执行多个程序的操作系统都有进程的概念。一个进程就是一个执行中的程序，而每一个进程都有自己独立的一块内存空间、一组系统资源。在进程的概念中，每一个进程的内部数据和状态都是完全独立的。在Windows操作系统下可以通过Ctrl+Alt+Del组合键查看进程，在UNIX和Linux操作系统下是通过ps命令查看进程的。打开Windows当前运行的进程，如图23-1所示。

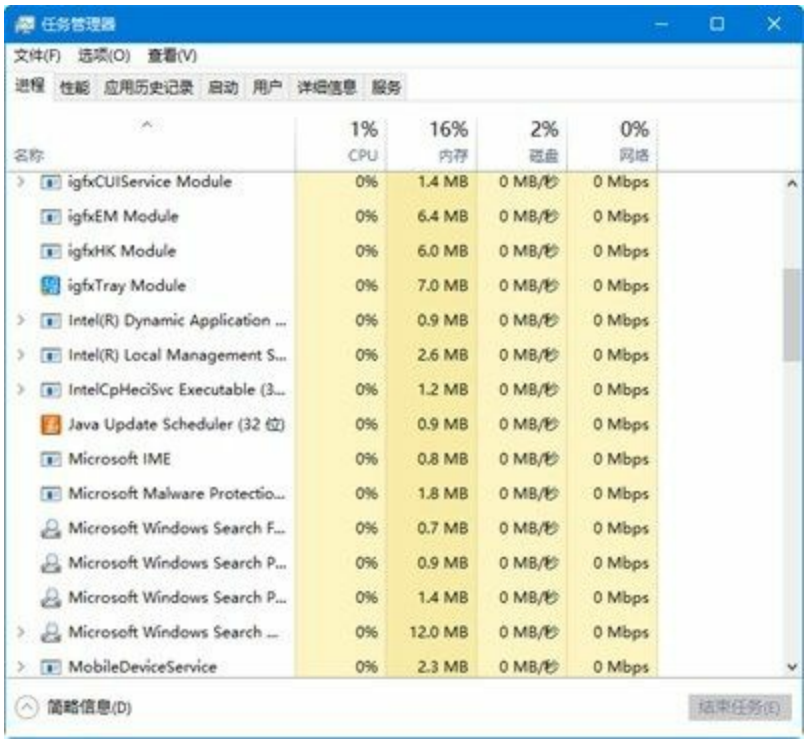


图**23-1** **Windows**操作系统进程

在Windows操作系统中一个进程就是一个exe或者dll程序，它们相互独立，互相也可以通信，在Android操作系统中进程间的通信应用也是很多的。

23.1.2 线程

线程与进程相似，是一段完成某个特定功能的代码，是程序中单个顺序控制的流程，但与进程不同的是，同类的多个线程是共享一块内存空间和一组系统资源。所以系统在各个线程之间切换时，开销要比进程小的多，正因如此，线程被称为轻量级进程。一个进程中可以包含多个线程。

23.1.3 主线程

Java程序至少会有一个线程，这就是主线程，程序启动后是由JVM创建主线程，程序结束时由JVM停止主线程。主线程它负责管理子线程，即子线程的启动、挂起、停止等等操作。图23-2所示是进程、主线程和子线程的关系，其中主线程负责管理子线程，即子线程的启动、挂起、停止等操作。

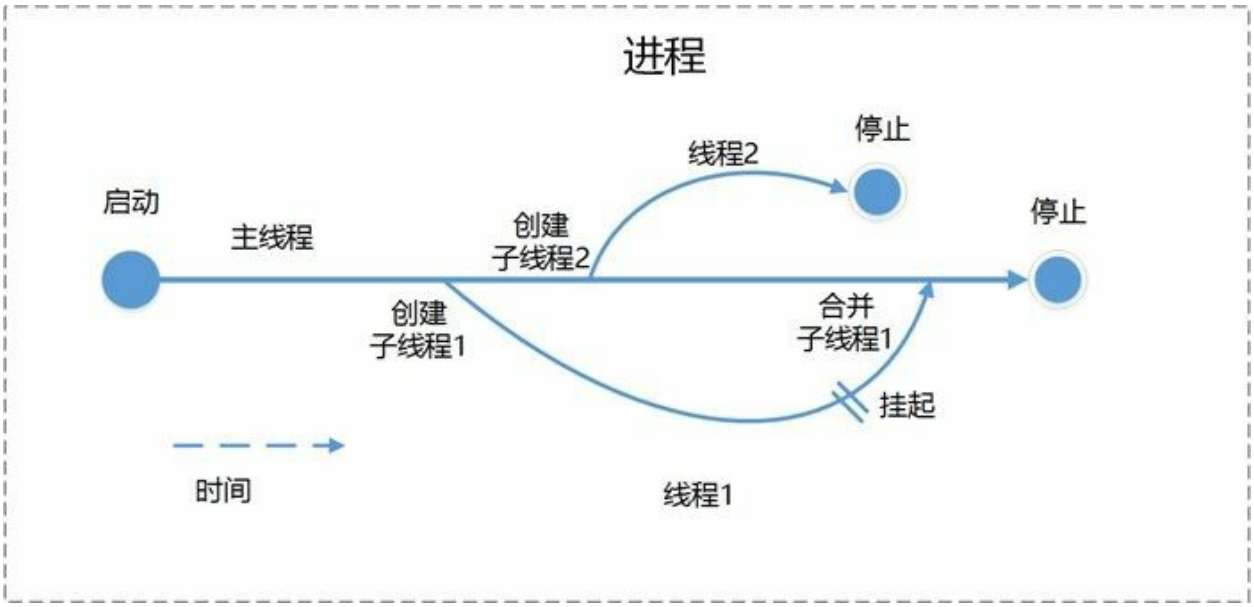


图**23-2**　进程、主线程和子线程关系

获取主线程示例代码如下：

```
//HelloWorld.java文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {
        //获取主线程
        Thread mainThread = Thread.currentThread();                          ①

        System.out.println("主线程名：" + mainThread.getName());            ②
    }
}
```

上述代码第①行是Thread.currentThread()获得当前线程，由于在main()方法中当前线程就是主线程，Thread是Java线程类，位于java.lang包中。代码第②行的getName()方法获得线程的名字，主线程名是main，由JVM分配。

23.2 创建子线程

Java中创建一个子线程涉及到：java.lang.Thread类和java.lang.Runnable接口。Thread是线程类，创建一个Thread对象就会产生一个新的线程。而线程执行的程序代码是在实现Runnable接口对象的run()方法中编写的，实现Runnable接口对象是线程执行对象。

线程执行对象实现Runnable接口的run()方法，run()方法是线程执行的入口，该线程要执行程序代码都在此编写的，run()方法称为线程体。

> 提示　主线程中执行入口是main(String[] args)方法，这里可以控制程序的流程，管理其他的子线程等。子线程执行入口是线程执行对象（实现Runnable接口对象）的run()方法，在这个方法可以编写子线程相关处理代码。

23.2.1 实现**Runnable**接口

创建线程Thread对象时，可以将线程执行对象传递给它，这需要是使用Thread类如下两个构造方法：

- Thread(Runnable target, String name)：target是线程执行对象，实现Runnable接口。name为线程指定一个名字。
- Thread(Runnable target)：target是线程执行对象，实现Runnable接口。线程名字是由JVM分配的。

下面看一个具体示例，实现Runnable接口的线程执行对象Runner代码如下：

```
//Runner.java文件
package com.a51work6;

//线程执行对象
public class Runner implements Runnable {                              ①

    // 编写执行线程代码
    @Override
    public void run() {                                                ②
        for (int i = 0; i < 10; i++) {
            // 打印次数和线程的名字
            System.out.printf("第 %d次执行 - %s\n", i,
                            Thread.currentThread().getName());         ③

            try {
                // 随机生成休眠时间
                long sleepTime = (long) (1000 * Math.random());
                // 线程休眠
                Thread.sleep(sleepTime);                               ④
            } catch (InterruptedException e) {
            }
        }
        // 线程执行结束
        System.out.println("执行完成! " + Thread.currentThread().getName());
    }
}
```

上述代码第①行声明实现Runnable接口，这要覆盖代码第②行的run()方法，run()方法是线程体，在该方法中编写你自己的线程处理代码。

本例中线程体中进行了十次循环，每次让当前线程休眠一段时间。其中代码第③行是打印次数和线程的名字，Thread.currentThread()可以获得当前线程对象，getName()是Thread类的实例方法，可以获得线

程的名。代码第④行Thread.sleep(sleepTime)是休眠当前线程, sleep是静态方法它有两个版本：

- static void sleep(long millis)：在指定的毫秒数内让当前正在执行的线程休眠。
- static void sleep(long millis, int nanos) 在指定的毫秒数加指定的纳秒数内让当前正在执行的线程休眠。

测试程序HelloWorld代码如下：

```
//HelloWorld.java文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        // 创建线程t1，参数是一个线程执行对象Runner
        Thread t1 = new Thread(new Runner());                ①
        // 开始线程t1
        t1.start();                                          ②

        // 创建线程t2，参数是一个线程执行对象Runner
        Thread t2 = new Thread(new Runner(), "MyThread");    ③
        // 开始线程t2
        t2.start();                                          ④
    }
}
```

上述代码创建了两个子线程，见代码第①行和第③行，构造方法参数是线程执行对象Runner，其中代码第①行的构造方法没有指定线程的名字，代码第③行的构造方法指定了线程名字。线程创建完成还需要调用start()方法才能执行，见代码第②行和第④行，start()方法一旦调用线程进入可以执行状态，可以执行状态下的线程等待CPU调度执行，CPU调用后线程进行执行状态，运行run()方法。

运行结果如下：

```
第 0次执行 - MyThread
第 0次执行 - Thread-0
第 1次执行 - Thread-0
第 1次执行 - MyThread
第 2次执行 - MyThread
第 2次执行 - Thread-0
第 3次执行 - MyThread
第 3次执行 - Thread-0
第 4次执行 - Thread-0
第 5次执行 - Thread-0
第 6次执行 - Thread-0
第 4次执行 - MyThread
第 7次执行 - Thread-0
第 5次执行 - MyThread
第 8次执行 - Thread-0
第 6次执行 - MyThread
第 9次执行 - Thread-0
第 7次执行 - MyThread
执行完成! Thread-0
第 8次执行 - MyThread
第 9次执行 - MyThread
执行完成! MyThread
```

提示　仔细分析一下运行结果，会发现两个线程是交错运行的，感觉就像是两个线程在同时运

行。但是实际上一台PC通常就只有一颗CPU，在某个时刻只能是一个线程在运行，而Java语言在设计时就充分考虑到线程的并发调度执行。对于程序员来说，在编程时要注意给每个线程执行的时间和机会，主要是通过让线程休眠的办法（调用sleep()方法）来让当前线程暂停执行，然后由其他线程来争夺执行的机会。如果上面的程序中没有用到sleep()方法，则就是第一个线程先执行完毕，然后第二个线程再执行完毕。所以用活sleep()方法是多线程编程的关键。

23.2.2 继承**Thread**线程类

事实上Thread类也实现了Runnable接口，所以Thread类也可以作为线程执行对象，这需要继承Thread类，覆盖run()方法。

采用继承Thread类重新实现23.2.1节示例，自定义线程类MyThread代码如下：

```
//MyThread.java文件
package com.a51work6;

//线程执行对象
public class MyThread extends Thread {

    public MyThread() {                          ①
        super();                                 ②
    }

    public MyThread(String name) {               ③
        super(name);                             ④
    }

    // 编写执行线程代码
    @Override
    public void run() {                          ⑤
        for (int i = 0; i < 10; i++) {
            // 打印次数和线程的名字
            System.out.printf("第 %d次执行 - %s\n", i, getName());

            try {
                // 随机生成休眠时间
                long sleepTime = (long) (1000 * Math.random());
                // 线程休眠
                sleep(sleepTime);
            } catch (InterruptedException e) {
                }
        }
        // 线程执行结束
        System.out.println("执行完成! " + getName());
    }
}
```

上述代码第①行和第③行定义了一个构造方法，通过super调用父类Thread构造方法，这两个Thread类构造方法：

- Thread(String name)：name为线程指定一个名字。代码第④行调用都就是此构造方法。
- Thread()：线程名字是JVM分配的。代码第②行调用都就是此构造方法。

代码第⑤行覆盖Thread类的run()方法，run()方法是线程体，需要线程执行的代码编写在这里。

测试程序HelloWorld代码如下：

```
//HelloWorld.java文件
package com.a51work6;
```

```
public class HelloWorld {

    public static void main(String[] args) {

        // 创建线程t1
        Thread t1 = new MyThread();                  ①
        // 开始线程t1
        t1.start();

        // 创建线程t2
        Thread t2 = new MyThread("MyThread");        ②
        // 开始线程t2
        t2.start();
    }
}
```

代码第①行调用无参数构造方法创建线程对象t1，代码第②行是调用有一个字符串参数的构造方法创建线程对象t2。

提示　由于Java只支持单重继承，继承Thread类的方式不能再继承其他父类。当开发一些图形界面的应用时，需要一个类既是一个窗口（继承JFrame）又是一个线程体，那么只能采用实现Runnable接口方式。

23.2.3　使用匿名内部类和Lambda表达式实现线程体

如果线程体使用的地方不是很多，可以不用单独定义一个类。可以使用匿名内部类或Lambda表达式直接实现Runnable接口。Runnable中只有一个方法是函数式接口，可以使用Lambda表达式。

重新实现23.2.1节示例，代码如下：

```
//HelloWorld.java文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        // 创建线程t1，参数是实现Runnable接口的匿名内部类
        Thread t1 = new Thread(new Runnable() {                ①
            // 编写执行线程代码
            @Override
            public void run() {
                for (int i = 0; i < 10; i++) {
                    // 打印次数和线程的名字
                    System.out.printf("第 %d次执行 - %s\n", i, Thread.currentThread().getName());
                    try {
                        // 随机生成休眠时间
                        long sleepTime = (long) (1000 * Math.random());
                        // 线程休眠
                        Thread.sleep(sleepTime);
                    } catch (InterruptedException e) {
                    }
                }
                // 线程执行结束
                System.out.println("执行完成! " + Thread.currentThread().getName());
            }

        });
        // 开始线程t1
        t1.start();
```

```
        // 创建线程t2，参数是实现Runnable接口的Lambda表达式
        Thread t2 = new Thread(() -> {                          ②
            for (int i = 0; i < 10; i++) {
                // 打印次数和线程的名字
                System.out.printf("第 %d次执行 - %s\n", i, Thread.currentThread().getName());
                try {
                    // 随机生成休眠时间
                    long sleepTime = (long) (1000 * Math.random());
                    // 线程休眠
                    Thread.sleep(sleepTime);
                } catch (InterruptedException e) {
                }
            }
            // 线程执行结束
            System.out.println("执行完成! " + Thread.currentThread().getName());
        }, "MyThread");
        // 开始线程t2
        t2.start();
    }
}
```

上述代码第①行采用匿名内部类实现Runnable接口，覆盖run()方法。这里使用的是Thread(Runnable target)构造方法。代码第②行采用Lambda表达式实现Runnable接口，覆盖run()方法。这里使用的是Thread(Runnable target, String name)构造方法，Lambda表达式是它的第一个参数。匿名内部类和Lambda表达式代码虽然很多，但是它只是一个参数，实现了Runnable接口线程执行对象，如图23-3所示深颜色部分是匿名内部类，如图23-4所示深颜色部分是Lambda表达式。

```
// 创建线程t1，参数是实现Runnable接口的匿名内部类
Thread t1 = new Thread(new Runnable() {
    // 编写执行线程代码
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            // 打印次数和线程的名字
            System.out.printf("第%d次执行 - %s\n", i, Thread.currentThread().getName());
            try {
                // 随机生成休眠时间
                long sleepTime = (long) (1000 * Math.random());
                // 线程休眠
                Thread.sleep(sleepTime);
            } catch (InterruptedException e) {
            }
        }
        // 线程执行结束
        System.out.println("执行完成! " + Thread.currentThread().getName());
    }
});
```

图**23-3**　匿名内部类

```
// 创建线程t2，参数是实现Runnable接口的Lambda不表达式
Thread t2 = new Thread(() -> {
    for (int i = 0; i < 10; i++) {
        // 打印次数和线程的名字
        System.out.printf("第%d次执行 - %s\n", i, Thread.currentThread().getName());
        try {
            // 随机生成休眠时间
            long sleepTime = (long) (1000 * Math.random());
            // 线程休眠
            Thread.sleep(sleepTime);
        } catch (InterruptedException e) {
        }
    }
    // 线程执行结束
    System.out.println("执行完成！ " + Thread.currentThread().getName());
}, "MyThread");
```

图**23-4　Lambda**表达式

提示　匿名内部类和Lambda表达式不需要定义一个线程类文件，使用起来很方便。特别是Lambda表达式使代码变得非常简洁。但是客观上匿名内部类和Lambda表达式会使代码可读性变差，对于初学者不容易理解。

23.3 线程的状态

在线程的生命周期中，线程会有几种状态，如图23-5所示，线程有5种状态。下面分别介绍一下。

01. 新建状态

    新建状态（New）是通过new等方式创建线程对象，它仅仅是一个空的线程对象。

02. 就绪状态

    当主线程调用新建线程的start()方法后，它就进入就绪状态（Runnable）。此时的线程尚未真正开始执行run()方法，它必须等待CPU的调度。

03. 运行状态

    CPU的调度就绪状态的线程，线程进入运行状态（Running），处于运行状态的线程独占CPU，执行run()方法。

04. 阻塞状态

    因为某种原因运行状态的线程会进入不可运行状态，即阻塞状态（Blocked），处于阻塞状态的线程JVM系统不能执行该线程，即使CPU空闲，也不能执行该线程。如下几个原因会导致线程进入阻塞状态：

    - 当前线程调用sleep()方法，进入休眠状态。
    - 被其他线程调用了join()方法，等待其他线程结束。
    - 发出I/O请求，等待I/O操作完成期间。
    - 当前线程调用wait()方法。

    处于阻塞状态可以重新回到就绪状态，如：休眠结束、其他线程加入、I/O操作完成和调用notify或notifyAll唤醒wait线程。

05. 死亡状态

    线程退出run()方法后，就会进入死亡状态（Dead），线程进入死亡状态有可以是正常实现完成run()方法进入，也可能是由于发生异常而进入的。

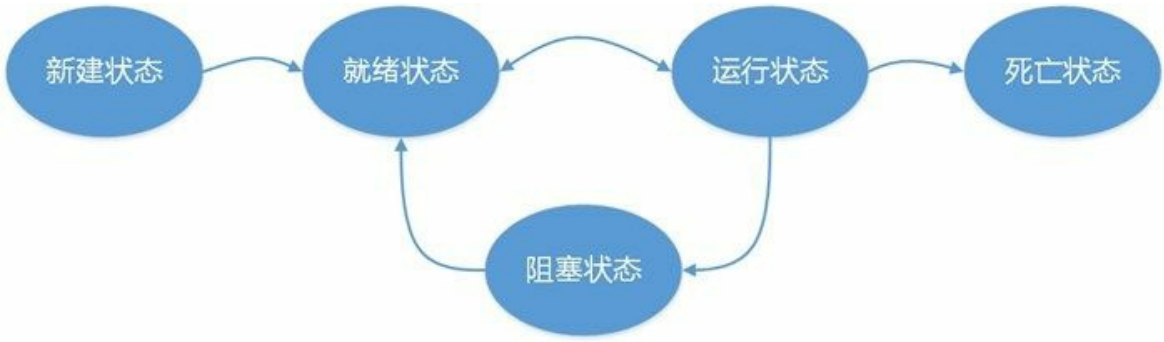


图**23-5** 线程状态

23.4 线程管理

线程管理是比较头痛的事情，这是学习线程的难点。下面分别介绍一下。

23.4.1 线程优先级

线程的调度程序根据线程决定每次线程应当何时运行，Java提供了10种优先级，分别用1~10整数表示，最高优先级是10用常量MAX_PRIORITY表示；最低优先级是1用常量MIN_PRIORITY；默认优先级是5用常量NORM_PRIORITY表示。

Thread类提供了setPriority(int newPriority)方法可以设置线程优先级，通过getPriority()方法获得线程优先级。

设置线程优先级示例代码如下：

```
//HelloWorld.java文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        // 创建线程t1，参数是一个线程执行对象Runner
        Thread t1 = new Thread(new Runner());
        t1.setPriority(Thread.MAX_PRIORITY);                    ①
        // 开始线程t1
        t1.start();

        // 创建线程t2，参数是一个线程执行对象Runner
        Thread t2 = new Thread(new Runner(), "MyThread");
        t2.setPriority(Thread.MIN_PRIORITY);                    ②
        // 开始线程t2
        t2.start();
    }
}
```

在代码第①行设置线程t1优先级最高，代码第②行设置线程t2优先级最低。

提示　多次运行上面的示例会发现，t1线程经常先运行，但是偶尔t2线程也会先运行。这些现象说明了：影响线程获得CPU时间的因素，除了受到的线程优先级外，还与操作系统有关。

23.4.2 等待线程结束

在介绍现在状态时提到过join()方法，当前线程调用t1线程的join()方法，则阻塞当前线程，等待t1线程结束，如果t1线程结束或等待超时，则当前线程回到就绪状态。

Thread类提供了多个版本的join()，它们定义如下：

- void join()：等待该线程结束。
- void join(long millis)：等待该线程结束的时间最长为millis毫秒。如果超时为0意味着要一直等下去。
- void join(long millis, int nanos)：等待该线程结束的时间最长为millis毫秒加nanos纳秒。

使用join()方法示例代码如下：

```
//HelloWorld.java文件
package com.a51work6;

public class HelloWorld {

    //共享变量
    static int value = 0;                                    ①

    public static void main(String[] args) throws InterruptedException {

        System.out.println("主线程 开始...");

        // 创建线程t1，参数是一个线程执行对象Runner
        Thread t1 = new Thread(() -> {                       ②
            System.out.println("ThreadA 开始...");
            for (int i = 0; i < 2; i++) {
                System.out.println("ThreadA 执行...");
                value++;                                     ③
            }
            System.out.println("ThreadA 结束...");

        }, "ThreadA");
        // 开始线程t1
        t1.start();
        // 主线程被阻塞，等待t1线程结束
        t1.join();                                           ④
        System.out.println("value = " + value);              ⑤
        System.out.println("主线程 结束...");
    }
}
```

运行结果如下：

```
主线程 开始...
ThreadA 开始...
ThreadA 执行...
ThreadA 执行...
ThreadA 结束...
value = 2
主线程 结束...
```

上述代码第①行是声明了一个共享变量value，这个变量在子线程中修改，然后主线程访问它。代码第②行是采用Lambda表达式创建线程，指定线程名为ThreadA。代码第③行是在子线程ThreadA中修改共享变量value。

代码第④行是在当前线程（主线程）中调用t1的join()方法，因此会导致主线程阻塞，等待t1线程结束，从运行结果可以看出主线程被阻塞了。代码第⑤行是打印共享变量value，从运行结果可见value = 2。

如果尝试将t1.join()语句注释掉，输出结果如下：

```
主线程 开始...
value = 0
主线程 结束...
ThreadA 开始...
ThreadA 执行...
ThreadA 执行...
ThreadA 结束...
```

提示　使用join()方法的场景是，一个线程依赖于另外一个线程的运行结果，所以调用另一个线程的join()方法等它运行完成。

23.4.3　线程让步

线程类Thread还提供一个静态方法yield()，调用yield()方法能够使当前线程给其他线程让步。它类似于sleep()方法，能够使运行状态的线程放弃CPU使用权，暂停片刻，然后重新回到就绪状态。与sleep()方法不同的是，sleep()方法是线程进行休眠，能够给其他线程运行的机会，无论线程优先级高低都有机会运行。而yield()方法只给相同优先级或更高优先级线程机会。

示例代码如下：

```
//Runner.java文件
package com.a51work6;

//线程执行对象
public class Runner implements Runnable {

    // 编写执行线程代码
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            // 打印次数和线程的名字
            System.out.printf("第 %d次执行 - %s\n", i,
                    Thread.currentThread().getName());
            Thread.yield();                                ①
        }
        // 线程执行结束
        System.out.println("执行完成! " + Thread.currentThread().getName());
    }
}
```

代码第①行Thread.yield()能够使当前线程让步。

提示　yield()方法只能给相同优先级或更高优先级的线程让步，yield()方法在实际开发中很少使用，大多都使用sleep()方法，sleep()方法可以控制时间，而yield()方法不能。

23.4.4　线程停止

线程体中的run()方法结束，线程进入死亡状态，线程就停止了。但是有些业务比较复杂，例如想开发一个下载程序，每隔一段执行一次下载任务，下载任务一般会在由子线程执行的，休眠一段时间再执行。这个下载子线程中会有一个死循环，但是为了能够停止子线程，设置一个结束变量。

示例下面如下：

```
//HelloWorld.java文件
package com.a51work6;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class HelloWorld {

    private static String command = "";                          ①

    public static void main(String[] args) {

        // 创建线程t1，参数是一个线程执行对象Runner
```

```
        Thread t1 = new Thread(() -> {

            // 一直循环，直到满足条件在停止线程
            while (!command.equalsIgnoreCase("exit")) {        ②
                // 线程开始工作
                // TODO
                System.out.println("下载中...");
                try {
                    // 线程休眠
                    Thread.sleep(10000);
                } catch (InterruptedException e) {
                }
            }
            // 线程执行结束
            System.out.println("执行完成!");
        });
        // 开始线程t1
        t1.start();

        try (InputStreamReader ir = new InputStreamReader(System.in);     ③
                BufferedReader in = new BufferedReader(ir)) {
            // 从键盘接收了一个字符串的输入
            command = in.readLine();                                      ④
        } catch (IOException e) {
        }

    }
}
```

上述代码第①行是设置一个结束变量。代码第②行是在子线程的线程体中判断，用户输入的是否为exit字符串，如果不是则进行循环，否则结束循环，结束循环就结束了run()方法，线程就停止了。

代码第③行中的System.in是一个很特殊的输入流，能够从控制台（键盘）读取字符。代码第④行是通过流System.in读取键盘输入的字符串。测试是需要注意：在控制台输入exit，然后敲Enter键，如图23-6所示。

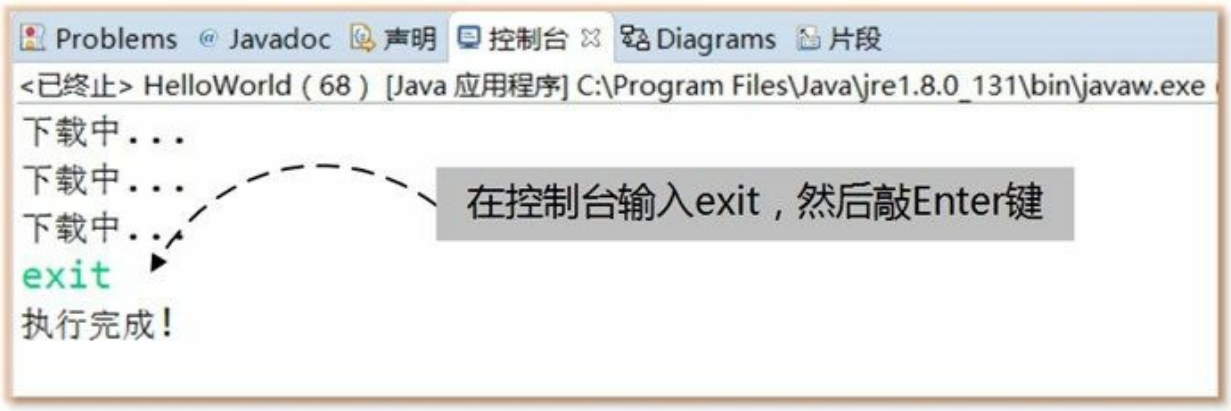


图**23-6**　在控制台输入字符

提示　控制线程的停止有人会想到使用Thread提供的stop()方法，这个方法已经不推荐使用了，这个方法有时会引发严重的系统故障，类似还是有suspend()和resume()挂起方法。Java现在推荐的做法就是采用本例的结束变量方式。

23.5 线程安全

在多线程环境下，访问相同的资源，有可以会引发线程不安全问题。本节讨论引发这些问题的根源和解决方法。

23.5.1 临界资源问题

多一个线程同时运行，有时线程之间需要共享数据，一个线程需要其他线程的数据，否则就不能保证程序运行结果的正确性。

例如有一个航空公司的机票销售，每一天机票数量是有限的，很多售票点同时销售这些机票。下面是一个模拟销售机票系统，示例代码如下：

```
//TicketDB.java文件
package com.a51work6;

//机票数据库
public class TicketDB {

    // 机票的数量
    private int ticketCount = 5;                                ①

    // 获得当前机票数量
    public int getTicketCount() {                               ②
        return ticketCount;
    }

    // 销售机票
    public void sellTicket() {                                  ③
        try {
            // 等于用户付款
            // 线程休眠，阻塞当前线程，模拟等待用户付款
            Thread.sleep(1000);                                 ④
        } catch (InterruptedException e) {
        }
        System.out.printf("第%d号票,已经售出\n", ticketCount);
        ticketCount--;                                          ⑤
    }
}
```

上述代码模拟机票销售过程，代码第①行是声明机票数量成员变量ticketCount，这是模拟当天可供销售的机票数，为了测试方便初始值设置为5。代码第②行是定义了获取当前机票数的getTicketCount()方法。代码第③行是销售机票方法，售票网点查询出有没有票可以销售，那么会调用sellTicket()方法销售机票，这个过程中需要等待用户付款，付款成功后，会将机票数减一，见代码第⑤行。为模拟等待用户付款，在代码第④行使用了sleep()方法让当前线程阻塞。

调用代码如下：

```
//HelloWorld.java文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        TicketDB db = new TicketDB();

        // 创建线程t1
```

```
        Thread t1 = new Thread(() -> {
            while (true) {
                int currTicketCount = db.getTicketCount();        ①
                // 查询是否有票
                if (currTicketCount > 0) {                        ②
                    db.sellTicket();                              ③
                } else {
                    // 无票退出
                    break;
                }
            }
        });
        // 开始线程t1
        t1.start();

        // 创建线程t2
        Thread t2 = new Thread(() -> {
            while (true) {
                int currTicketCount = db.getTicketCount();
                // 查询是否有票
                if (currTicketCount > 0) {
                    db.sellTicket();
                } else {
                    // 无票退出
                    break;
                }
            }
        });
        // 开始线程t2
        t2.start();
    }
}
```

在HelloWorld中创建了两个线程，模拟两个售票网点，没有线程所做的事情类似。首先获得当前机票数量（见代码第①行），然后判断机票数量是否大于零（见代码第②行），如果有票则出票（见代码第②行），否则退出循环，结束线程。

一次运行结果如下：

```
第5号票,已经售出
第5号票,已经售出
第3号票,已经售出
第3号票,已经售出
第1号票,已经售出
第0号票,已经售出
```

虽然可以能每次运行的结果都不一样，但是从结果看还是能发现一些问题：同一张票重复销售、出现第0号票和5张票卖了6次。这些问题的根本原因是多个线程间共享的数据导致数据的不一致性。

> 提示　多个线程间共享的数据称为共享资源或临界资源，由于是CPU负责线程的调度，程序员无法精确控制多线程的交替顺序。这种情况下，多线程对临界资源的访问有时会导致数据的不一致性。

23.5.2 多线程同步

为了防止多线程对临界资源的访问有时会导致数据的不一致性，Java提供了“互斥”机制，可以为这些资源对象加上一把“互斥锁”，在任一时刻只能由一个线程访问，即使该线程出现阻塞，该对象的被锁定状态也不会解除，其他线程仍不能访问该对象，这就多线程同步。线程同步保证线程安全的重要手段，但是线程同步客观上会导致性能下降。

可以通过两种方式实现线程同步，两种方式都涉及到使用synchronized关键字，一种是synchronized方法，使用synchronized关键字修饰方法，对方法进行同步；另一种是synchronized语句，使用synchronized关键字放在对象前面限制一段代码的执行。

01. synchronized方法

synchronized关键字修饰方法实现线程同步，方法所在的对象被锁定，修改23.5.1节售票系统示例， TicketDB.java文件代码如下：

```
//TicketDB.java文件
package com.a51work6.method;

//机票数据库
public class TicketDB {

    // 机票的数量
    private int ticketCount = 5;

    // 获得当前机票数量
    public synchronized int getTicketCount() {                    ①
        return ticketCount;
    }

    // 销售机票
    public synchronized void sellTicket() {                        ②
        try {
            // 等于用户付款
            // 线程休眠，阻塞当前线程，模拟等待用户付款
            Thread.sleep(1000);
        } catch (InterruptedException e) {
        }
        System.out.printf("第%d号票,已经售出\n", ticketCount);
        ticketCount--;
    }
}
```

上述代码第①行和第②行的方法前都使用了synchronized关键字，表明这两个方法是同步的，被锁定的，每一个时刻只能由一个线程访问。并不是每一个方法都有必要加锁的，要仔细研究加上的必要性，上述代码第①行加锁可以防止出现第0号票情况和5张票卖出6次的情况；代码第②行加锁是防止出现销售两种一样的票，读者可以自己测试一下。

采用synchronized方法修改示例，调用代码HelloWorld.java不需要任何修改。

02. synchronized语句

synchronized语句方式主要用于第三方类，不方便修改它的代码情况。同样是23.5.1节售票系统示例，可以不用修改TicketDB.java类，只修改调用代码HelloWorld.java实现同步。

HelloWorld.java代码如下：

```
//HelloWorld.java文件
package com.a51work6.statement;

public class HelloWorld {

    public static void main(String[] args) {

        TicketDB db = new TicketDB();

        // 创建线程t1
```

```
        Thread t1 = new Thread(() -> {
            while (true) {
                synchronized (db) {                    ①
                    int currTicketCount = db.getTicketCount();
                    // 查询是否有票
                    if (currTicketCount > 0) {
                        db.sellTicket();
                    } else {
                        // 无票退出
                        break;
                    }
                }
            }
        });
        // 开始线程t1
        t1.start();

        // 创建线程t2
        Thread t2 = new Thread(() -> {
            while (true) {
                synchronized (db) {                    ②
                    int currTicketCount = db.getTicketCount();
                    // 查询是否有票
                    if (currTicketCount > 0) {
                        db.sellTicket();
                    } else {
                        // 无票退出
                        break;
                    }
                }
            }
        });
        // 开始线程t2
        t2.start();
    }
}
```

代码第①行和第②行是使用synchronized语句，将需要同步的代码用大括号括起来。synchronized后有小括号，将需要同步的对象括起来。

23.6 线程间通信

第23.5节的示例只是简单地为特定对象或方法加锁，但有时情况会更加复杂，如果两个线程之间有依赖关系，线程之间必须进行通信，互相协调才能完成工作。

例如有一个经典的堆栈问题，一个线程生成了一些数据，将数据压栈；另一个线程消费了这些数据，将数据出栈。这两个线程互相依赖，当堆栈为空时，消费线程无法取出数据时，应该通知生成线程添加数据；当堆栈已满时，生产线程无法添加数据时，应该通知消费线程取出数据。

为了实现线程间通信，需要使用Object类中声明的5个方法：

- void wait()：使当前线程释放对象锁，然后当前线程处于对象等待队列中阻塞状态，如图23-7所示，等待其他线程唤醒。
- void wait(long timeout)：同wait()方法，等待timeout毫秒时间。
- void wait(long timeout, int nanos)：同wait()方法，等待timeout毫秒加nanos纳秒时间。
- void notify()：当前线程唤醒此对象等待队列中的一个线程，如图23-7所示该线程将进入就绪状态。
- void notifyAll()：当前线程唤醒此对象等待队列中的所有线程，如图23-7所示这些线程将进入就绪状态。

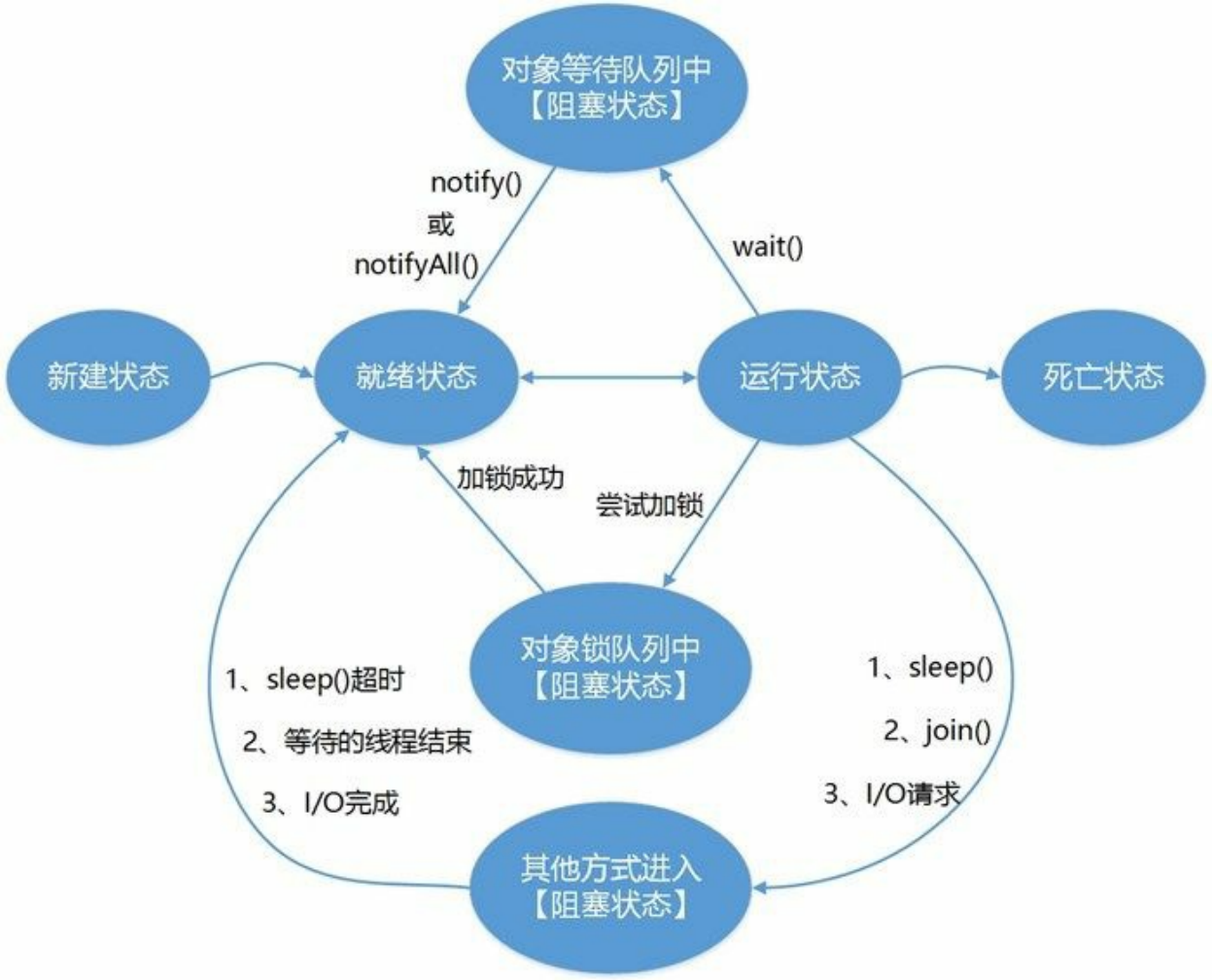


图**23-7**　线程间通信

提示　图23-7是图23-5补充，从图23-7可见，线程有多种方式进入阻塞状态，除了通过wait()外还有，加锁的方式和其他方式，加锁方式是23.5节介绍的使用synchronized加互斥锁；其他方式事实上是23.3节线程状态时介绍的方式，这里不再赘述。

下面看看消费和生产示例中堆栈类代码：

```
//Stack.java文件
package com.a51work6;

//堆栈类
class Stack {
    // 堆栈指针初始值为0
    private int pointer = 0;
    // 堆栈有5个字符的空间
    private char[] data = new char[5];

    // 压栈方法，加上互斥锁
    public synchronized void push(char c) {                    ①
        // 堆栈已满，不能压栈
        while (pointer == data.length) {
            try {
                // 等待，直到有数据出栈
```

```
                this.wait();
            } catch (InterruptedException e) {
            }
        }
        // 通知其他线程把数据出栈
        this.notify();
        // 数据压栈
        data[pointer] = c;
        // 指针向上移动
        pointer++;
    }

    // 出栈方法，加上互斥锁
    public synchronized char pop() {                        ②
        // 堆栈无数据，不能出栈
        while (pointer == 0) {
            try {
                // 等待其他线程把数据压栈
                this.wait();
            } catch (InterruptedException e) {
            }
        }
        // 通知其他线程压栈
        this.notify();
        // 指针向下移动
        pointer--;
        // 数据出栈
        return data[pointer];
    }
}
```

上述代码实现了同步堆栈类，该堆栈有最多5个元素的空间，代码第①行声明了压栈方法push()，该方法是一个同步方法，在该方法中首先判断是否堆栈已满，如果已满不能压栈，调用this.wait()让当前线程进入对象等待状态中。如果堆栈未满，程序会往下运行调用this.notify()唤醒对象等待队列中的一个线程。代码第②行声明了出栈方法pop()方法，与push()方法类似，这里不再赘述。

调用代码如下：

```
//HelloWorld.java文件
package com.a51work6;

public class HelloWorld {

    public static void main(String args[]) {

        Stack stack = new Stack();                                  ①

        // 下面的消费者和生产者所操作的是同一个堆栈对象stack
        // 生产者线程
        Thread producer = new Thread(() -> {                        ②
            char c;
            for (int i = 0; i < 10; i++) {
                // 随机产生10个字符
                c = (char) (Math.random() * 26 + 'A');
                // 把字符压栈
                stack.push(c);
                // 打印字符
                System.out.println("生产: " + c);
                try {
                    // 每产生一个字符线程就睡眠
                    Thread.sleep((int) (Math.random() * 1000));
                } catch (InterruptedException e) {
                }
```

```
            }

        });

        // 消费者线程
        Thread consumer = new Thread(() -> {                    ③
            char c;
            for (int i = 0; i < 10; i++) {
                // 从堆栈中读取字符
                c = stack.pop();
                // 打印字符
                System.out.println("消费: " + c);
                try {
                    // 每读取一个字符线程就睡眠
                    Thread.sleep((int) (Math.random() * 1000));
                } catch (InterruptedException e) {
                }
            }

        });

        producer.start(); // 启动生产者线程
        consumer.start(); // 启动消费者线程
    }
}
```

上述代码第①行创建堆栈对象。代码第②行创建生产者线程，代码第③行创建消费者线程。

本章小结

本章介绍了Java线程技术，首先是介绍了线程相关的一些概念，然后介绍了如何创建子线程、线程状态、线程管理、线程安全和线程间通信等内容，其中创建线程和线程管理是学习的重点，掌握线程状态和线程安全，了解线程间通信。

第 **24** 章　网络编程

现代的应用程序都离不开网络，网络编程是非常重要的技术。Java SE提供java.net包，其中包含了网络编程所需要的最基础一些类和接口。这些类和接口面向两个不同的层次：基于Socket的低层次网络编程和基于URL的高层次网络编程，所谓高低层次就是通信协议的高低层次，Socket采用TCP、UDP等协议，这些协议属于低层次的通信协议；URL采用HTTP和HTTPS这些属于高层次的通信协议。低层次网络编程，因为它面向底层，比较复杂，但是“低层次网络编程”并不等于它功能不强大。恰恰相反，正因为层次低，Socket编程与基于URL的高层次网络编程比较，能够提供更强大的功能和更灵活的控制，但是要更复杂一些。

本章会介绍基于Socket的低层次网络编程和基于URL的高层次网络编程，以及数据交换格式。

24.1 网络基础

网络编程需要程序员掌握一下基础的网络知识，这一节先介绍一些网络基础知识。

24.1.1 网络结构

首先了解一下网络结构，网络结构是网络的构建方式，目前流行的有客户端服务器结构网络和对等结构网络。

01. 客户端服务器结构网络

    客户端服务器（Client Server，缩写C/S）结构网络，是一种主从结构网络。如图24-1所示，服务器一般处于等待状态，如果有客户端请求，服务器响应请求建立连接提供服务。服务器是被动的，有点像在餐厅吃饭时候的服务员。而客户端是主动的，像在餐厅吃饭的顾客。

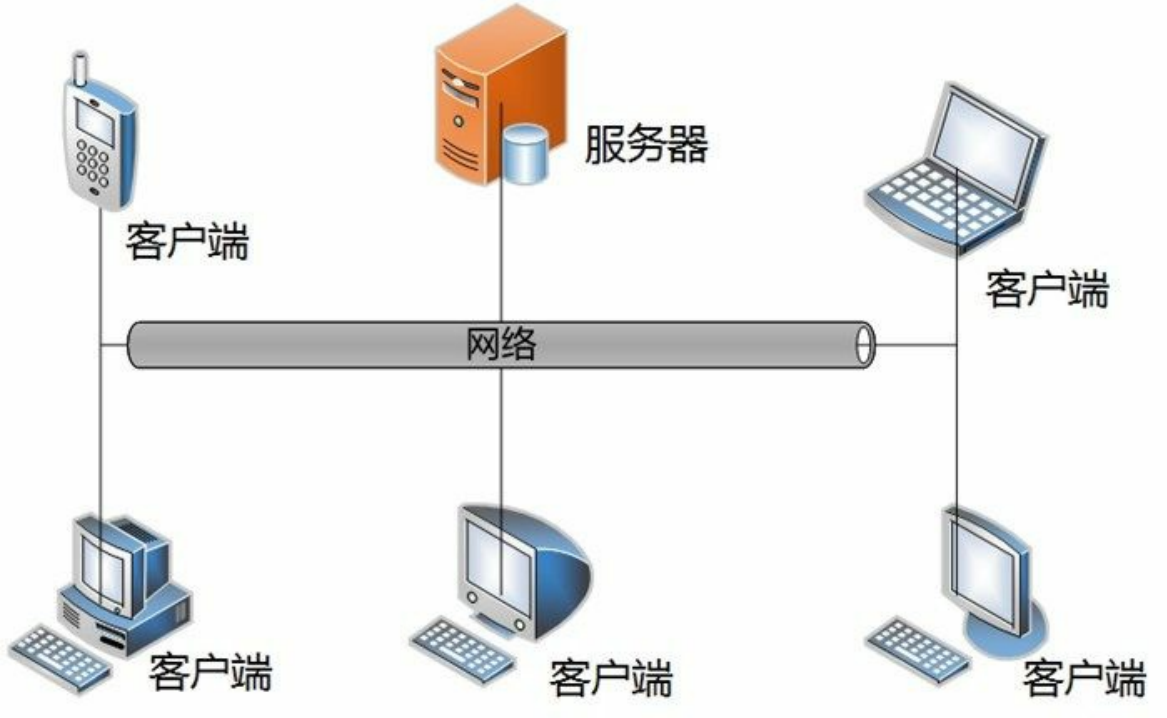


图**24-1** 客户端服务器结构网络

事实上，生活中很多网络服务都采用这种结构。例如：Web服务、文件传输服务和邮件服务等。虽然它们存在的目的不一样，但基本结构是一样的。这种网络结构与设备类型无关，服务器不一定是电脑，也可能是手机等移动设备。

02. 对等结构网络

    对等结构网络也叫点对点网络（Peer to Peer，缩写P2P），每个节点之间是对等的。它们如图24-2所示，每个节点既是服务器又是客户端，这种结构有点像吃自助餐。

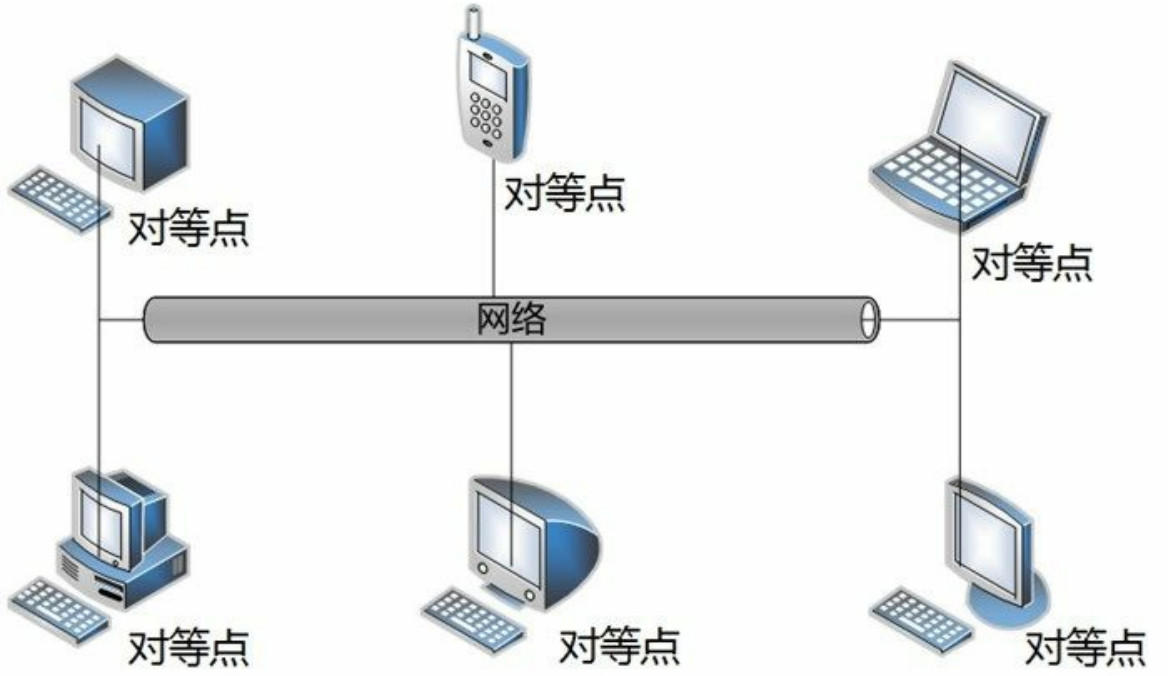


图**24-2**　对等结构网络

对等结构网络分布范围比较小。通常在一间办公室或一个家庭内，因此它非常适合于移动设备间的网络通讯，网络链路层是由蓝牙和WiFi实现。

24.1.2　TCP/IP协议

网络通信会用到协议，其中TCP/IP协议是非常重要的。TCP/IP协议是由IP和TCP两个协议构成的，IP（Internet Protocol）协议是一种低级的路由协议，它将数据拆分成许多小的数据包中，并通过网络将它们发送到某一特定地址，但无法保证都所有包都抵达目的地，也不能保证包的顺序。

由于IP协议传输数据的不安全性，网络通信时还需要TCP协议，传输控制协议（Transmission Control Protocol，TCP）是一种高层次的协议，面向连接的可靠数据传输协议，如果有些数据包没有收到会重发，并对数据包内容准确性检查并保证数据包顺序，所以该协议保证数据包能够安全地按照发送时顺序送达目的地。

24.1.3　IP地址

为实现网络中不同计算机之间的通信，每台计算机都必须有一个与众不同的标识，这就是IP地址，TCP/IP使用IP地址来标识源地址和目的地址。最初所有的IP地址都是32位数字构成，由4个8位的二进制数组成，每8位之间用圆点隔开，如：192.168.1.1，这种类型的地址通过IPv4指定。而现在有一种新的地址模式称为IPv6，IPv6使用128位数字表示一个地址，分为8个16位块。尽管IPv6比IPv4有很多优势，但是由于习惯的问题，很多设备还是采用IPv4。不过Java语言同时指出IPv4和IPv6。

在IPv4地址模式中IP地址分为A、B、C、D和E等5类。

- A类地址用于大型网络，地址范围：1.0.0.1~126.155.255.254。
- B类地址用于中型网络，地址范围：128.0.0.1~191.255.255.254。
- C类地址用于小规模网络，192.0.0.1~223.255.255.254。

- D类地址用于多目的地信息的传输和作为备用。
- E类地址保留仅作实验和开发用。

另外，有时还会用到一个特殊的IP地址127.0.0.1，127.0.0.1称为回送地址，指本机。主要用于网络软件测试以及本地机进程间通信，使用回送地址发送数据，不进行任何网络传输，只在本机进程间通信。

24.1.4 端口

一个IP地址标识这一台计算机，每一台计算机又有很多网络通信程序在运行，提供网络服务或进行通信，这就需要不同的端口进行通信。如果把IP地址比作电话号码，那么端口就是分机号码，进行网络通信时不仅要指定IP地址，还要指定端口号。

TCP/IP系统中的端口号是一个16位的数字，它的范围是0~65535。小于1024的端口号保留给预定义的服务，如HTTP是80，FTP是21，Telnet是23，Email是25等，除非要和那些服务进行通信，否则不应该使用小于1024的端口。

24.2 TCP Socket低层次网络编程

TCP/IP协议的传输层有两种传输协议：TCP（传输控制协议）和 UDP（用户数据报协议）。TCP是面向连接的可靠数据传输协议。TCP就像好比电话，电话接通后双方才能通话，在挂断电话之前，电话一直占线。TCP连接一旦建立起来，一直占用，直到关闭连接。另外，TCP为了保证数据的正确性，会重发一切没有收到的数据，还会对进行数据内容进行验证，并保证数据传输的正确顺序。因此TCP协议对系统资源的要求较多。

基于TCP Socket编程很有代表性，先介绍TCP Socket编程。

24.2.1 TCP Socket通信概述

Socket是网络上的两个程序，通过一个双向的通信连接，实现数据的交换。这个双向链路的一端称为一个Socket。Socket通常用来实现客户端和服务端的连接。Socket是TCP/IP协议的一个十分流行的编程接口，一个Socket由一个IP地址和一个端口号唯一确定，一旦建立连接Socket还会包含本机和远程主机的IP地址和远端口号，如图24-3所示，Socket是成对出现的。

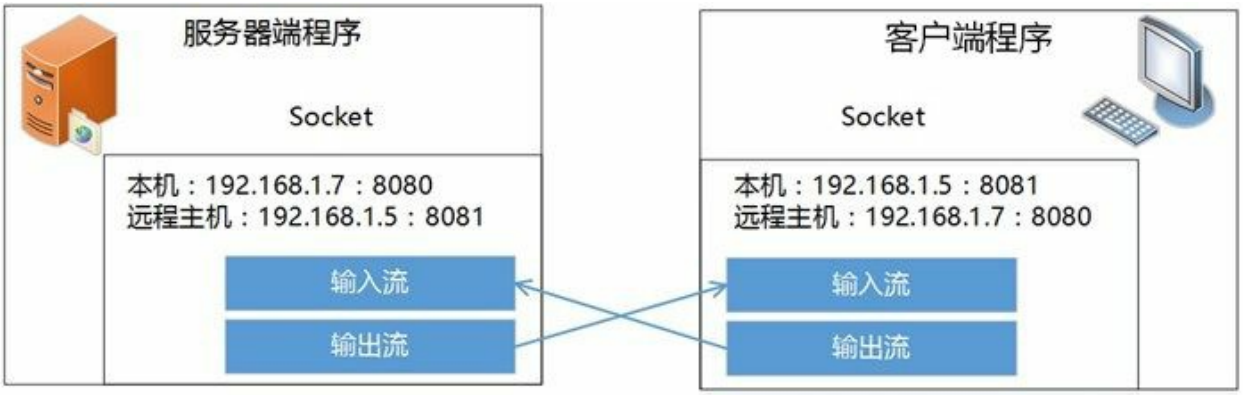


图**24-3 TCP Socket**通信

24.2.2 TCP Socket通信过程

使用Socket进行C/S结构编程，通信过程如图24-4所示。

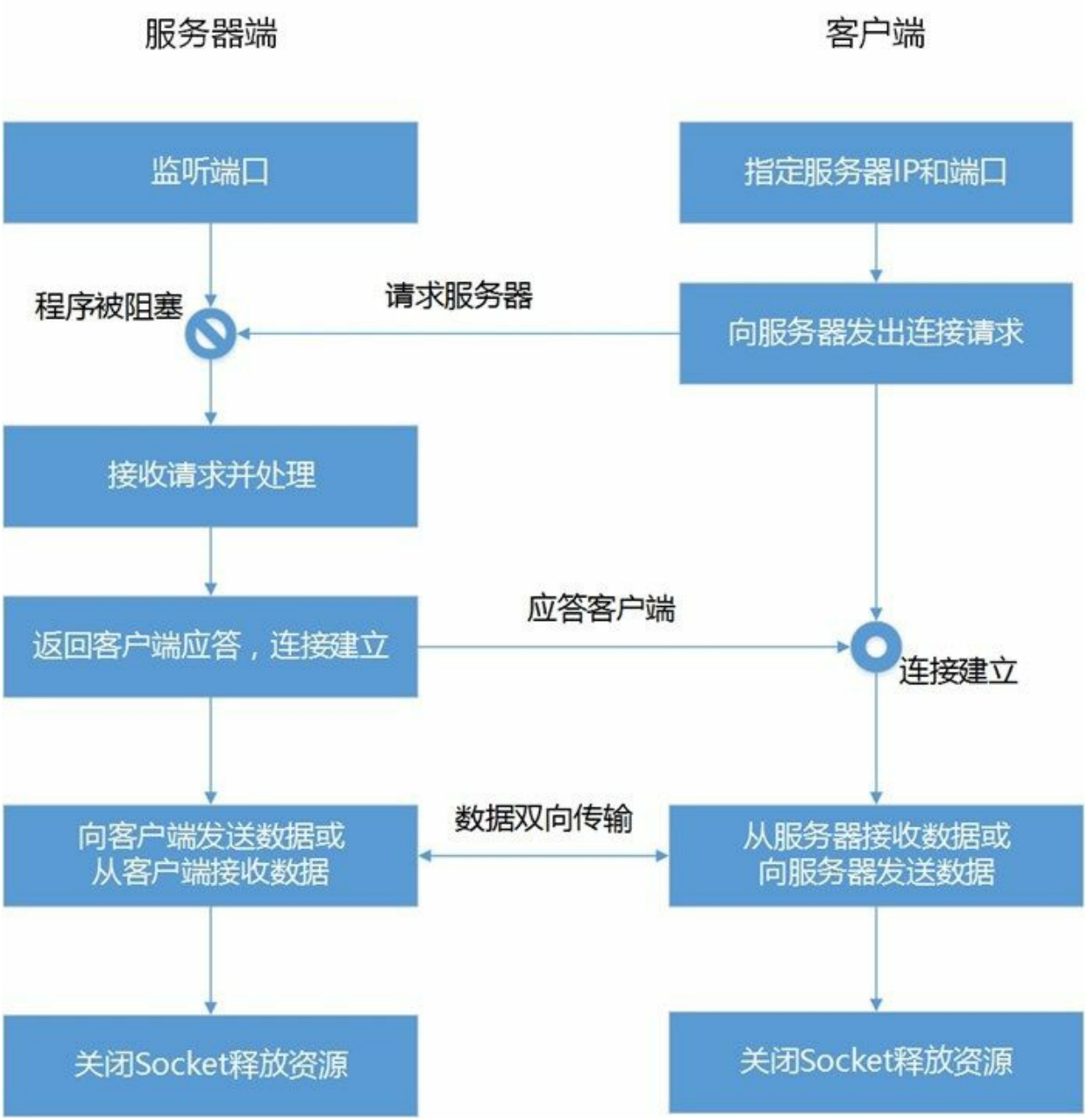


图**24-4　TCP Socket**通信过程

服务器端监听某个端口是否有连接请求，服务器端程序处于阻塞状态，直到客户端向服务器端发出连接请求，服务器端接收客户端请求，服务器会响应请求，处理请求，然后将结果应答给客户端，这样就会建立连接。一旦连接建立起来，通过Socket可以获得输入输出流对象。借助于输入输出流对象就可以实现服务器与客户端的通信，最后不要忘记关闭Socket和释放一些资源（包括：关闭输入输出流）。

24.2.3　Socket类

java.net包为TCP Socket编程提供了两个核心类：Socket和ServerSocket，分别用来表示双向连接的客户端和服务器端。

本节先介绍一下Socket类，Socket常用的构造方法有：

- Socket(InetAddress address, int port) ：创建Socket对象，并指定远程主机IP地址和端口号。
- Socket(InetAddress address, int port, InetAddress localAddr, int localPort)：创建Socket对象，并指定远程主机IP地址和端口号，以及本机的IP地址（localAddr）和端口号（localPort）。
- Socket(String host, int port)：创建Socket对象，并指定远程主机名和端口号，IP地址为null，null表示回送地址，即127.0.0.1。
- Socket(String host, int port, InetAddress localAddr, int localPort)：创建Socket对象，并指定远程主机和端口号，以及本机的IP地址（localAddr）和端口号（localPort）。host主机名，IP地址为null，null表示回送地址，即127.0.0.1。

Socket其他的常用方法有：

- InputStream getInputStream()：通过此Socket返回输入流对象。
- OutputStream getOutputStream()：通过此Socket返回输出流对象。
- int getPort()：返回Socket连接到的远程端口。
- int getLocalPort()：返回Socket绑定到的本地端口。
- InetAddress getInetAddress()：返回Socket连接的地址。
- InetAddress getLocalAddress()：返回Socket绑定的本地地址。
- boolean isClosed()：返回Socket是否处于关闭状态。
- boolean isConnected()：返回Socket是否处于连接状态。
- void close()：关闭Socket。

注意　Socket与流类似所占用的资源，不能通过JVM的垃圾收集器回收，需要程序员释放。一种方法是可以在finally代码块调用close()方法关闭Socket，释放流所占用的资源。另一种方法通过自动资源管理技术释放资源，Socket和ServerSocket都实现了AutoCloseable接口。

24.2.4 ServerSocket类

ServerSocket类常用的构造方法有：

- ServerSocket(int port, int maxQueue)：创建绑定到特定端口的服务器Socket。maxQueue设置连接的请求最大队列长度，如果队列满时，则拒绝该连接。默认值是50。
- ServerSocket(int port)：创建绑定到特定端口的服务器Socket。最大队列长度是50。

ServerSocket其他的常用方法有：

- InputStream getInputStream()：通过此Socket返回输入流对象。
- OutputStream getOutputStream()：通过此Socket返回输出流对象。
- boolean isClosed()：返回Socket是否处于关闭状态。
- Socket accept()：侦听并接收到Socket的连接。此方法在建立连接之前一直阻塞。

- void close()：关闭Socket。

ServerSocket类本身不能直接获得I/O流对象，而是通过accept()方法返回Socket对象，通过Socket对象取得I/O流对象，进行网络通信。此外，ServerSocket也实现了AutoCloseable接口，通过自动资源管理技术关闭ServerSocket。

24.2.5 案例：文件上传工具

基于TCP Socket编程比较复杂，先从一个简单的文件上传工具案例介绍TCP Socket编程基本流程。上传过程是一个单向Socket通信过程，如图24-5所示，客户端通过文件输入流读取文件，然后从Socket获得输出流写入数据，写入数据完成上传成功，客户端任务完成。服务器端从Socket获得输入流，然后写入文件输出流，写入数据完成上传成功，服务器端任务完成。

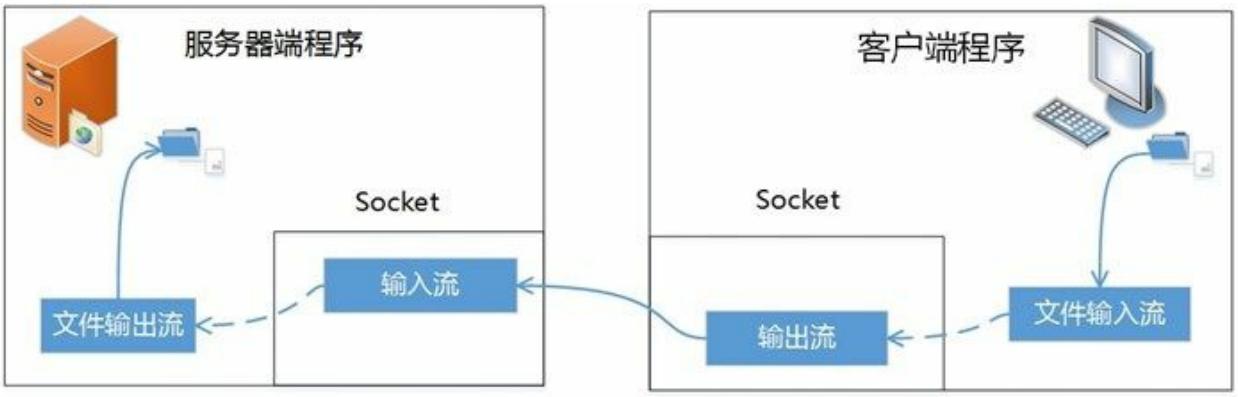


图**24-5** 单向**Socket**通信

下面看看案例服务器端UploadServer代码如下：

```
//UploadServer.java文件
package com.a51work6;
… …
public class UploadServer {

    public static void main(String[] args) {

        System.out.println("服务器端运行...");

        try ( // 创建一个ServerSocket监听8080端口的客户端请求
                ServerSocket server = new ServerSocket(8080);   ①
                // 使用accept()阻塞当前线程，等待客户端请求
                Socket socket = server.accept();                ②
                // 由Socket获得输入流，并创建缓冲输入流
                BufferedInputStream in = new BufferedInputStream(socket.getInputStream());  ③
                // 由文件输出流创建缓冲输出流
                FileOutputStream out = new FileOutputStream("./TestDir/subDir/coco2dxcplus.jpg")) {④

            // 准备一个缓冲区
            byte[] buffer = new byte[1024];
            // 首次从Socket读取数据
            int len = in.read(buffer);
            while (len != -1) {
                // 写入数据到文件
                out.write(buffer, 0, len);
                // 再次从Socket读取数据
                len = in.read(buffer);
            }
```

```
            System.out.println("接收完成! ");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

}
```

上述代码第①行创建ServerSocket对象监听本机的8080端口，这是当前线程还没有阻塞，调用代码第②行的server.accept()才会阻塞当前线程，等待客户端请求。

提示　由于当前线程是主线程，所以server.accept()会阻塞主线程，阻塞主线程是不明智的，如果是在一个图形界面的应用程序，阻塞主线程会导致无法进行任何的界面操作，就是常见的“卡”现象，所以最好是把server.accept()语句放到子线程中。

代码第③行是从socket对象中获得输入流对象，代码第④行是文件输出流。下面输入输出代码读者可以参考第22章，这里不再赘述。

再看看案例客户端UploadClient代码如下：

```
//UploadClient.java文件
package com.a51work6;
… …
public class UploadClient {

    public static void main(String[] args) {

        System.out.println("客户端运行...");

        try ( // 向本机的8080端口发出请求
                Socket socket = new Socket("127.0.0.1", 8080);          ①
                // 由Socket获得输出流，并创建缓冲输出流
                BufferedOutputStream out = new BufferedOutputStream(socket.getOutputStream());②
                // 创建文件输入流
                FileInputStream fin = new FileInputStream("./TestDir/coco2dxcplus.jpg");
                // 由文件输入流创建缓冲输入流
                BufferedInputStream in = new BufferedInputStream(fin)) {

            // 准备一个缓冲区
            byte[] buffer = new byte[1024];
            // 首次读取文件
            int len = in.read(buffer);
            while (len != -1) {
                // 数据写入Socket
                out.write(buffer, 0, len);
                // 再次读取文件
                len = in.read(buffer);
            }

            System.out.println("上传成功! ");

        } catch (ConnectException e) {                                  ③
            System.out.println("服务器未启动! ");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

上述代码第①行创建Socket，指定远程主机的IP地址和端口号。代码第②行是从socket对象获得输出

流。代码第③行是捕获ConnectException异常，这个异常引起的原因是在代码第①行向服务器发出请求时，服务器拒绝了客户端请求，这有两种可能性：一是服务器没有启动，服务器的8080端口没有打开；二是服务器请求队列已满（默认是50个）。

提示　案例测试时，先运行服务器，再运行客户端。测试Socket程序最好打开两个命令行窗口，通过java指令分别运行服务器程序和客户端程序，如图24-6和24-7所示，需要注意当前运行的路径是Eclipse工程根目录，需要指定类路径，命令的-cp .;./bin就是指定类路径，包括两个当前路径：其中点（.）表示当前路径，./bin表示bin目录，也可以写成.\bin。为什么要指定bin目录呢？是因为编译之后的字节码文件放在此目录中。另外，如果想在Eclipse中查看多个控制台信息，如图24-8所示，在控制台上面的工具栏中，单击“选择控制台”按钮实现切换。

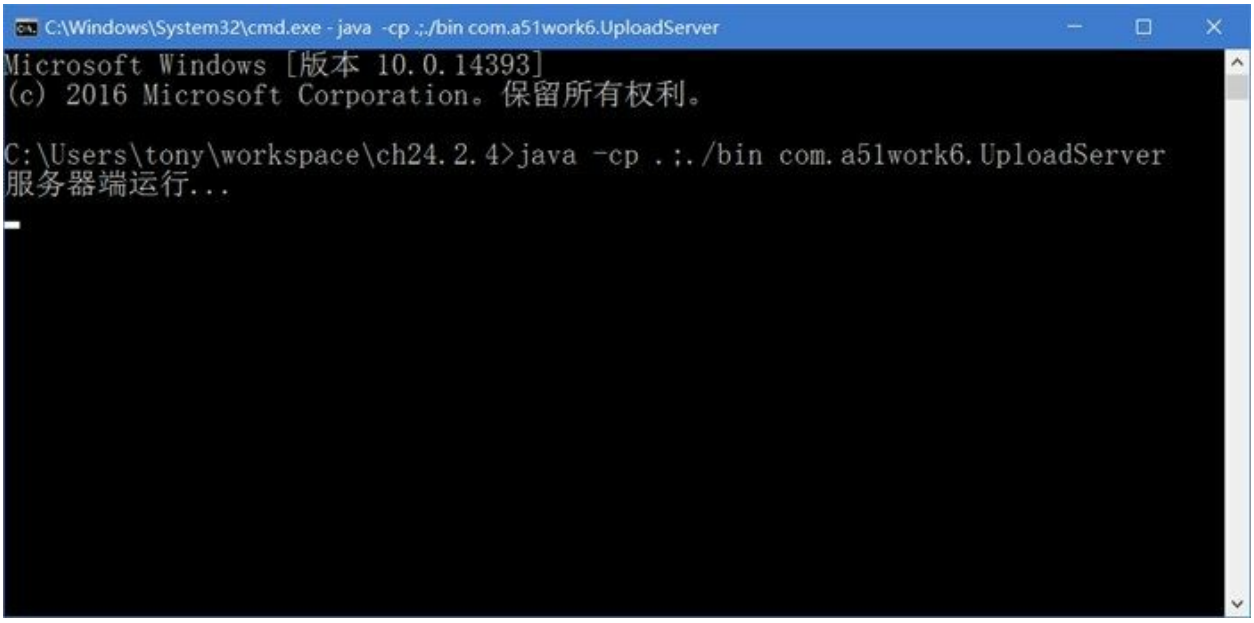


图**24-6**　命令行窗口运行服务器程序

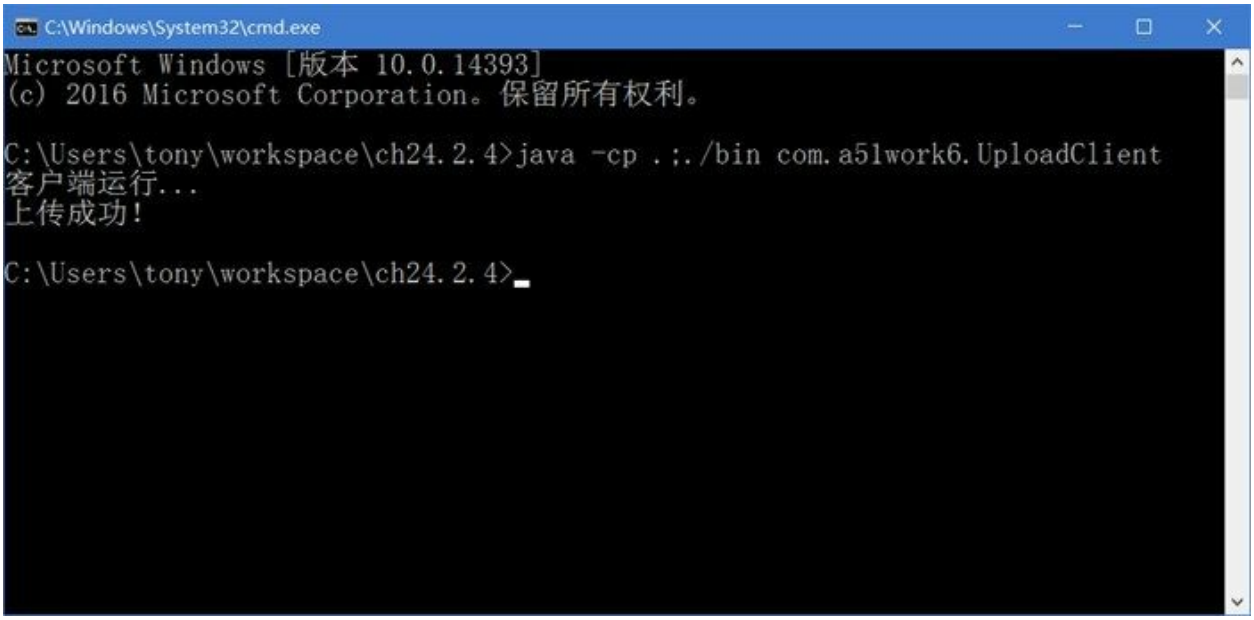


图**24-7**　命令行窗口运行客户端程序

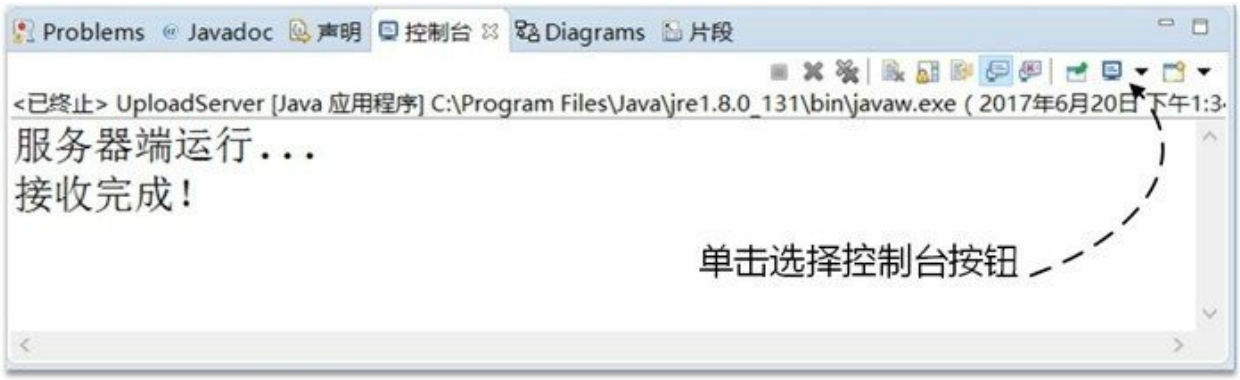


图**24-8**　**Eclipse**中切换控制台

24.2.6　案例：聊天工具

第24.2.5节介绍的案例只是单向传输的Socket，Socket可以双向数据传输，但是这就有些复杂了，比较有代表性的案例就是聊天工具。

如图24-9所示是基于TCP Socket聊天工具案例，其中的标准输入是键盘，标准输出是显示器的控制台。首先客户端通过键盘输入字符串，通过标准输入流读取字符串，然后通过Socket获得输出流，将字符串写入输出流。接着服务器通过Socket获得输入流，从输入流中读取来自客户端发送过来的字符串，然后通过标准输入流输出到显示器的控制台。服务器向客户端字符串过程类似。

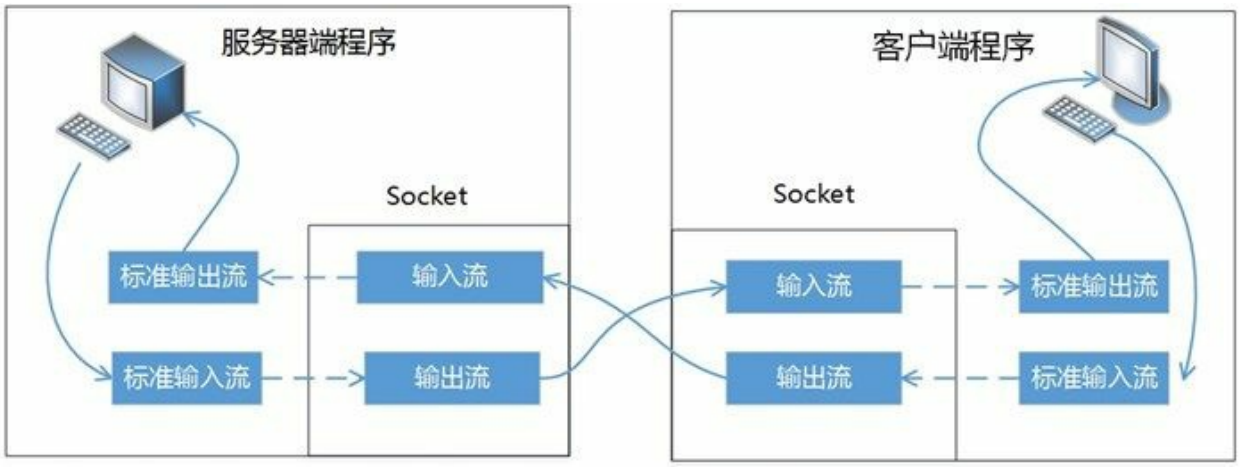


图**24-9**　基于**TCP Socket**聊天工具案例

下面看看案例服务器端ChatServer代码如下：

```
// ChatServer.java文件
package com.a51work6;
… …
public class ChatServer {

    public static void main(String[] args) {

        System.out.println("服务器运行...");

        Thread t = new Thread(() -> {                                                    ①

            try ( // 创建一个ServerSocket监听端口8080客户请求
```

```
                    ServerSocket server = new ServerSocket(8080);
                    // 使用accept()阻塞等待客户端请求
                    Socket socket = server.accept();
                    DataInputStream in = new DataInputStream(socket.getInputStream());          ②
                    DataOutputStream out = new DataOutputStream(socket.getOutputStream());     ③
                    BufferedReader keyboardIn
                            = new BufferedReader(new InputStreamReader(System.in))) {          ④

                while (true) {
                    /* 接收数据 */
                    String str = in.readUTF();                                                  ⑤
                    // 打印接收的数据
                    System.out.printf("从客户端接收的数据：【%s】\n", str);

                    /* 发送数据 */
                    // 读取键盘输入的字符串
                    String keyboardInputString = keyboardIn.readLine();
                    // 结束聊天
                    if (keyboardInputString.equals("bye")) {
                        break;
                    }
                    // 发送
                    out.writeUTF(keyboardInputString);                                          ⑥
                    out.flush();
                }
            } catch (Exception e) {
            }
            System.out.println("服务器停止...");
        });

        t.start();
    }
}
```

上述代码第①行是创建一个子线程，将网络通信放到子线程中处理是一种很好的做法，因为网络通信往往有线程阻塞过程，放到子线程中处理就不会阻塞主线程了。

代码第②行是从socket中获得数据输入流，代码第③行是从socket中获得数据输出流，数据流主要面向基本数据类型，本例中使用它们主要用来输入输出UTF编码的字符串，代码第⑤行readUTF()是数据输入流读取字符串。代码第⑥行writeUTF()是数据输出流写入字符串。代码第④行中的System.in是标准输入流，然后使用标准输入流创建缓冲输入流。

下面看看案例客户端ChatClient代码如下：

```
//ChatClient.java文件
package com.a51work6;
…
public class ChatClient {

    public static void main(String[] args) {

        System.out.println("客户端运行...");

        Thread t = new Thread(() -> {

            try ( // 向127.0.0.1主机8080端口发出连接请求
                    Socket socket = new Socket("127.0.0.1", 8080);
                    DataInputStream in = new DataInputStream(socket.getInputStream());
                    DataOutputStream out = new DataOutputStream(socket.getOutputStream());
                    BufferedReader keyboardIn
                        = new BufferedReader(new InputStreamReader(System.in))) {
```

```
                while (true) {
                    /* 发送数据 */
                    // 读取键盘输入的字符串
                    String keyboardInputString = keyboardIn.readLine();
                    // 结束聊天
                    if (keyboardInputString.equals("bye")) {
                        break;
                    }
                    // 发送
                    out.writeUTF(keyboardInputString);
                    out.flush();

                    /* 接收数据 */
                    String str = in.readUTF();
                    // 打印接收的数据
                    System.out.printf("从服务器接收的数据：【%s】\n", str);
                }
            } catch (ConnectException e) {
                System.out.println("服务器未启动！");
            } catch (Exception e) {
            }
            System.out.println("客户端停止！");
        });

        t.start();

    }
}
```

客户端ChatClient代码与服务器端ChatServer代码类似，这里不再赘述。

24.3 UDP Socket低层次网络编程

UDP（用户数据报协议）就像日常生活中的邮件投递，是不能保证可靠地寄到目的地。UDP是无连接的，对系统资源的要求较少，UDP可能丢包不保证数据顺序。但是对于网络游戏和在线视频等要求传输快、实时性高、质量可稍差一点的数据传输，UDP还是非常不错的。

UDP Socket网络编程比TCP Socket编程简单多，UDP是无连接协议，不需要像TCP一样监听端口，建立连接，然后才能进行通信。

24.3.1 DatagramSocket类

java.net包中提供了两个类：DatagramSocket和DatagramPacket用来支持UDP通信。这一节先介绍一下DatagramSocket类，DatagramSocket用于在程序之间建立传送数据报的通信连接。

先来看一下DatagramSocket常用的构造方法：

- DatagramSocket()：创建数据报DatagramSocket对象，并将其绑定到本地主机上任何可用的端口。
- DatagramSocket(int port)：创建数据报DatagramSocket对象，并将其绑定到本地主机上的指定端口。
- DatagramSocket(int port, InetAddress laddr)：创建数据报DatagramSocket对象，并将其绑定到指定的本地地址。

DatagramSocket其他的常用方法有：

- void send(DatagramPacket p)：从发送数据报包。
- void receive(DatagramPacket p)：接收数据报包。
- int getPort()：返回DatagramSocket连接到的远程端口。
- int getLocalPort()：返回DatagramSocket绑定到的本地端口。
- InetAddress getInetAddress()：返回DatagramSocket连接的地址。
- InetAddress getLocalAddress()：返回DatagramSocket绑定的本地地址。
- boolean isClosed()：返回DatagramSocket是否处于关闭状态。
- boolean isConnected()：返回DatagramSocket是否处于连接状态。
- void close()：关闭DatagramSocket。

DatagramSocket也实现了AutoCloseable接口，通过自动资源管理技术关闭DatagramSocket。

24.3.2 DatagramPacket类

DatagramPacket用来表示数据报包，是数据传输的载体。DatagramPacket实现无连接数据包投递服务，每投递数据包仅根据该包中信息从一台机器路由到另一台机器。从一台机器发送到另一台机器的多个包可能选择不同的路由，也可能按不同的顺序到达，不保证包都能到达目的。

下面看一下DatagramPacket的构造方法：

- DatagramPacket(byte[] buf, int length)：构造数据报包，buf包数据，length是接收包数据的长度。
- DatagramPacket(byte[] buf, int length, InetAddress address, int port)：构造数据报包，包发送到指定主机上的指定端口号。
- DatagramPacket(byte[] buf, int offset, int length)：构造数据报包，offset是buf字节数组的偏移量。
- DatagramPacket(byte[] buf, int offset, int length, InetAddress address, int port)：构造数据报包，包发送到指定主机上的指定端口号。

DatagramPacket常用的方法：

- InetAddress getAddress()：返回发往或接收该数据报包相关的主机的IP地址。
- byte[] getData()：返回数据报包中的数据。
- int getLength()：返回发送或接收到的数据（byte[]）的长度。
- int getOffset()：返回发送或接收到的数据（byte[]）的偏移量。
- int getPort()：返回发往或接收该数据报包相关的主机的端口号。

24.3.3 案例：文件上传工具

使用UDP Socket将24.1.5节文件上传工具重新实现一下。

下面看看案例服务器端UploadServer代码如下：

```
//UploadServer.java文件
package com.a51work6;
…
public class UploadServer {
    public static void main(String args[]) {

        System.out.println("服务器端运行...");

        // 创建一个子线程
        Thread t = new Thread(() -> {                                          ①

            try ( // 创建DatagramSocket对象，指定端口8080
                    DatagramSocket socket = new DatagramSocket(8080);          ②
                    FileOutputStream fout = new FileOutputStream("./TestDir/subDir/coco2dxcplus.jpg");
                    BufferedOutputStream out = new BufferedOutputStream(fout)) {

                // 准备一个缓冲区
                byte[] buffer = new byte[1024];

                //循环接收数据报包
                while (true) {

                    // 创建数据报包对象，用来接收数据
                    DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
                    // 接收数据报包
                    socket.receive(packet);
                    // 接收数据长度
                    int len = packet.getLength();

                    if (len == 3) {                                            ③
                        // 获得结束标志
                        String flag = new String(buffer, 0, 3);
                        // 判断结束标志，如果是bye结束接收
```

```
                    if (flag.equals("bye")) {                                ④
                        break;
                    }
                }
                // 写入数据到文件输出流
                out.write(buffer, 0, len);
            }
            System.out.println("接收完成！");
        } catch (IOException e) {
            e.printStackTrace();
        }
    });
    // 启动线程
    t.start();
  }
}
```

上述代码第①行是创建一个子线程，由于客户端上传的数据分为很多数据包，因此需要一个循环接收数据包，另外，调用后receive()方法会导致线程阻塞，因此需要将接收数据的处理放到一个子线程中。

代码第②行是创建DatagramSocket对象，并指定端口8080，作为服务器一般应该明确指定绑定的端口。

与TCP Socket不同UDP Socket无法知道哪些数据包已经是最后一个了，因此需要发送方发出一个特殊的数据包，包中包含了一些特殊标志。代码第③行~第④行是取出并判断这个标志。

再看看案例客户端UploadClient代码如下：

```
//UploadClient.java文件
package com.a51work6;
…
public class UploadClient {

    public static void main(String[] args) {

        System.out.println("客户端运行...");

        try (   // 创建DatagramSocket对象，由系统分配可以使用的端口
                DatagramSocket socket = new DatagramSocket();                    ①
                FileInputStream fin = new FileInputStream("./TestDir/coco2dxcplus.jpg");
                BufferedInputStream in = new BufferedInputStream(fin)) {

            // 创建远程主机IP地址对象
            InetAddress address = InetAddress.getByName("localhost");

            // 准备一个缓冲区
            byte[] buffer = new byte[1024];
            // 首次从文件流中读取数据
            int len = in.read(buffer);

            while (len != -1) {
                // 创建数据报包对象
                DatagramPacket packet = new DatagramPacket(buffer, len, address, 8080);
                // 发送数据报包
                socket.send(packet);
                // 再次从文件流中读取数据
                len = in.read(buffer);
            }

            // 创建数据报对象
            DatagramPacket packet = new DatagramPacket("bye".getBytes(), 3, address, 8080);
            // 发送结束标志
```

```
            socket.send(packet);                                                    ②
            System.out.println("上传完成！");

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

上述是上传文件客户端，发送数据不会堵塞线程，因此没有使用子线程。代码第①行是创建DatagramSocket对象，由系统分配可以使用的端口，作为客户端DatagramSocket对象经常自己不指定了，而是有系统分配。

代码第②行是发送结束标志，这个结束标志是字符串bye，服务器端接收到这个字符串则结束接收数据包。

24.3.4 案例：聊天工具

使用UDP Socket将24.1.6节文件聊天工具重新实现一下。

下面看看案例服务器端ChatServer代码如下：

```
// ChatServer.java文件
package com.a51work6;
…
public class ChatServer {

    public static void main(String args[]) {

        System.out.println("服务器运行...");
        // 创建一个子线程
        Thread t = new Thread(() -> {                                               ①
            try ( // 创建DatagramSocket对象，指定端口8080
                    DatagramSocket socket = new DatagramSocket(8080);
                    BufferedReader keyboardIn
                        = new BufferedReader(new InputStreamReader(System.in))) {

                while (true) {
                    /* 接收数据报 */
                    // 准备一个缓冲区
                    byte[] buffer = new byte[128];
                    DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
                    socket.receive(packet);
                    // 接收数据长度
                    int len = packet.getLength();

                    String str = new String(buffer, 0, len);
                    // 打印接收的数据
                    System.out.printf("从客户端接收的数据：【%s】\n", str);

                    /* 发送数据 */
                    // 从客户端传来的数据包中得到客户端地址
                    InetAddress address = packet.getAddress();                      ②
                    // 从客户端传来的数据包中得到客户端端口号
                    int port = packet.getPort();                                    ③

                    // 读取键盘输入的字符串
                    String keyboardInputString = keyboardIn.readLine();
                    // 读取键盘输入的字节数组
                    byte[] b = keyboardInputString.getBytes();
                    // 创建DatagramPacket对象，用于向客户端发送数据
                    packet = new DatagramPacket(b, b.length, address, port);
```

```
                    // 向客户端发送数据
                    socket.send(packet);
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        });
        // 启动线程
        t.start();
    }
}
```

上述代码第①行是创建一个子线程，因为socket.receive(packet)方法会阻塞主线程了。服务器给客户端发数据包，也需要知道它的IP地址和端口号，代码第②行根据接收的数据包获得客户端的地址，代码第③行类似根据接收的数据包获得客户端的端口号。

下面看看案例客户端ChatClient代码如下：

```
//ChatClient.java文件
package com.a51work6;
…
public class ChatClient {

    public static void main(String[] args) {

        System.out.println("客户端运行...");
        // 创建一个子线程
        Thread t = new Thread(() -> {

            try ( // 创建DatagramSocket对象，由系统分配可以使用的端口
                    DatagramSocket socket = new DatagramSocket();
                    BufferedReader keyboardIn
                         = new BufferedReader(new InputStreamReader(System.in))) {

                while (true) {

                    /* 发送数据 */
                    // 准备一个缓冲区
                    byte[] buffer = new byte[128];
                    // 服务器IP地址
                    InetAddress address = InetAddress.getByName("localhost");
                    // 服务器端口号
                    int port = 8080;
                    // 读取键盘输入的字符串
                    String keyboardInputString = keyboardIn.readLine();
                    // 退出循环,结束线程
                    if (keyboardInputString.equals("bye")) {
                        break;
                    }
                    // 读取键盘输入的字节数组
                    byte[] b = keyboardInputString.getBytes();
                    // 创建DatagramPacket对象
                    DatagramPacket packet = new DatagramPacket(b, b.length, address, port);
                    // 发送
                    socket.send(packet);

                    /* 接收数据报 */
                    packet = new DatagramPacket(buffer, buffer.length);
                    socket.receive(packet);

                    // 接收数据长度
                    int len = packet.getLength();
                    String str = new String(buffer, 0, len);
```

```
                    // 打印接收的数据
                    System.out.printf("从服务器接收的数据：【%s】\n", str);
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        });
        // 启动线程
        t.start();
    }
}
```

客户端ChatClient代码与服务器端ChatServer代码类似，这里不再赘述。这里需要注意的是ChatClient可以通过键盘输入bye，退出循环结束线程。

24.4 数据交换格式

数据交换格式就像两个人在聊天一样，采用彼此都能听得懂的语言，你来我往，其中的语言就相当于通信中的数据交换格式。有时候，为了防止聊天被人偷听，可以采用暗语。同理，计算机程序之间也可以通过数据加密技术防止“偷听”。

数据交换格式主要分为纯文本格式、XML格式和JSON格式，其中纯文本格式是一种简单的、无格式的数据交换方式。

例如，为了告诉别人一些事情，我会写下如图24-10所示的留言条。

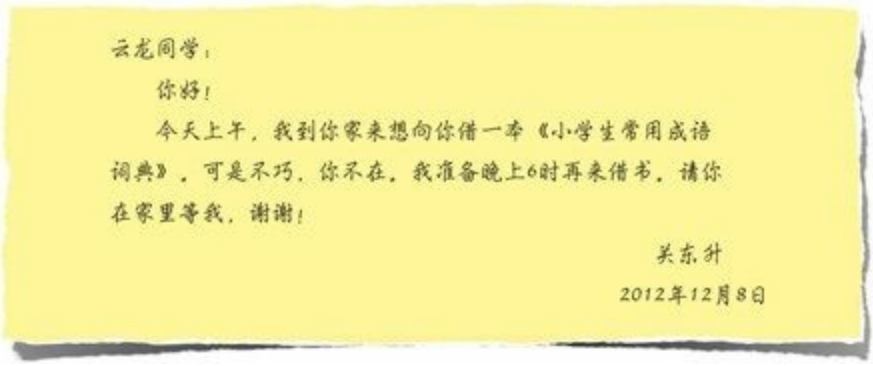


图**24-10** 留言条

留言条有一定的格式，共有4部分：称谓、内容、落款和时间，如图24-11所示。

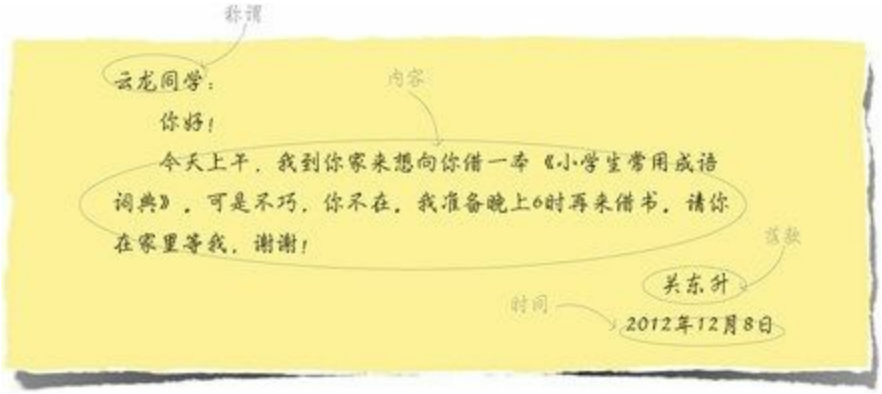


图**24-11** 留言条格式

如果用纯文本格式描述留言条，可以按照如下的形式：

```
"云龙同学","你好！\n今天上午，我到你家来想向你借一本《小学生常用成语词典》。可是不巧，你不在。我准备晚上6时再来借
```

留言条中的4部分数据按照顺序存放，各个部分之间用逗号分隔。数据量小的时候，可以采用这种格式。但是随着数据量的增加，问题也会暴露出来，可能会搞乱它们的顺序，如果各个数据部分能有描述信息就好了。而XML格式和JSON格式可以带有描述信息，它们叫做“自描述的”结构化文档。

将上面的留言条写成XML格式，具体如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<note>
    <to>云龙同学</to>
    <conent>你好！\n今天上午，我到你家来想向你借一本《小学生常用成语词典》。
        可是不巧，你不在。我准备晚上6时再来借书。请你在家里等我，谢谢！</conent>
    <from>关东升</from>
    <date>2012年12月08日</date>
</note>
```

上述代码中位于尖括号中的内容（<to>…</to>等）就是描述数据的标识，在XML中称为“标签”。

将上面的留言条写成JSON格式，具体如下：

```
{to:"云龙同学",conent:"你好！\n今天上午，我到你家来想向你借一本《小学生常用成语词典》。可是不巧，你不在。我准备
```

数据放置在大括号{}之中，每个数据项目之前都有一个描述名字（如to等），描述名字和数据项目之间用冒号（:）分开。

可以发现，一般来讲，JSON所用的字节数要比XML少，这也是很多人喜欢采用JSON格式的主要原因，因此JSON也被称为“轻量级”的数据交换格式。接下来，重点介绍JSON数据交换格式。

24.4.1 JSON文档结构

JSON（JavaScript Object Notation）是一种轻量级的数据交换格式。所谓轻量级，是与XML文档结构相比而言的，描述项目的字符少，所以描述相同数据所需的字符个数要少，那么传输速度就会提高，而流量却会减少。

如果留言条采用JSON描述，可以设计成下面的样子：

```
{"to":"云龙同学",
    "conent": "你好！\n今天上午，我到你家来想向你借一本《小学生常用成语词典》。可是
        不巧，你不在。我准备晚上6时再来借书。请你在家里等我，谢谢！",
    "from": "关东升",
    "date": "2012年12月08日"}
```

由于Web和移动平台开发对流量的要求是要尽可能少，对速度的要求是要尽可能快，而轻量级的数据交换格式JSON就成为理想的数据交换格式。

构成JSON文档的两种结构为对象和数组。对象是“名称-值”对集合，它类似于Java中Map类型，而数组是一连串元素的集合。

对象是一个无序的“名称/值”对集合，一个对象以{（左括号）开始，}（右括号）结束。每个“名称”后跟一个:（冒号），“名称-值”对之间使用,（逗号）分隔。JSON对象的语法表如图24-12所示。

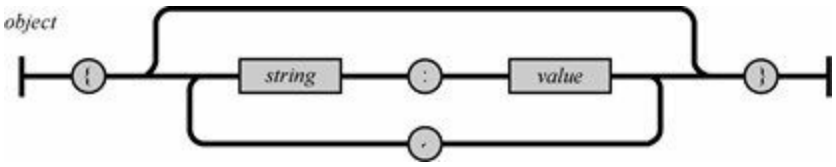


图**24-12 JSON**对象的语法表

下面是一个JSON对象的例子：

```
{
    "name":"a.htm",
    "size":345,
    "saved":true
}
```

数组是值的有序集合，以[（左中括号）开始，]（右中括号）结束，值之间使用,（逗号）分隔。JSON数组的语法表如图24-13所示。

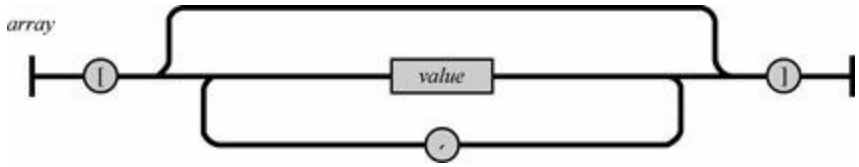


图**24-13　JSON**数组的语法表

下面是一个JSON数组的例子：

```
["text","html","css"]
```

在数组中，值可以是双引号括起来的字符串、数值、true、false、null、对象或者数组，而且这些结构可以嵌套。数组中值的JSON语法结构如图24-14所示。

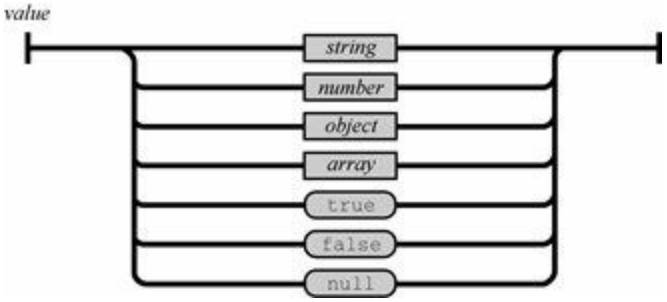


图**24-14　JSON**值的语法结构图

24.4.2　使用第三方**JSON**库

由于目前Java官方没有提供JSON编码和解码所需要的类库，所以需要使用第三方JSON库，笔者推荐JSON-java库，JSON-java库提供源代码，最重要的是不依赖于其他第三方库，需要再起找其他的库了。读者可以在https://github.com/stleary/JSON-java下载源代码。API在线文档http://stleary.github.io/JSON-java/index.html。

下载JSON-java获得源代码文件，解压后文件如图24-15所示。

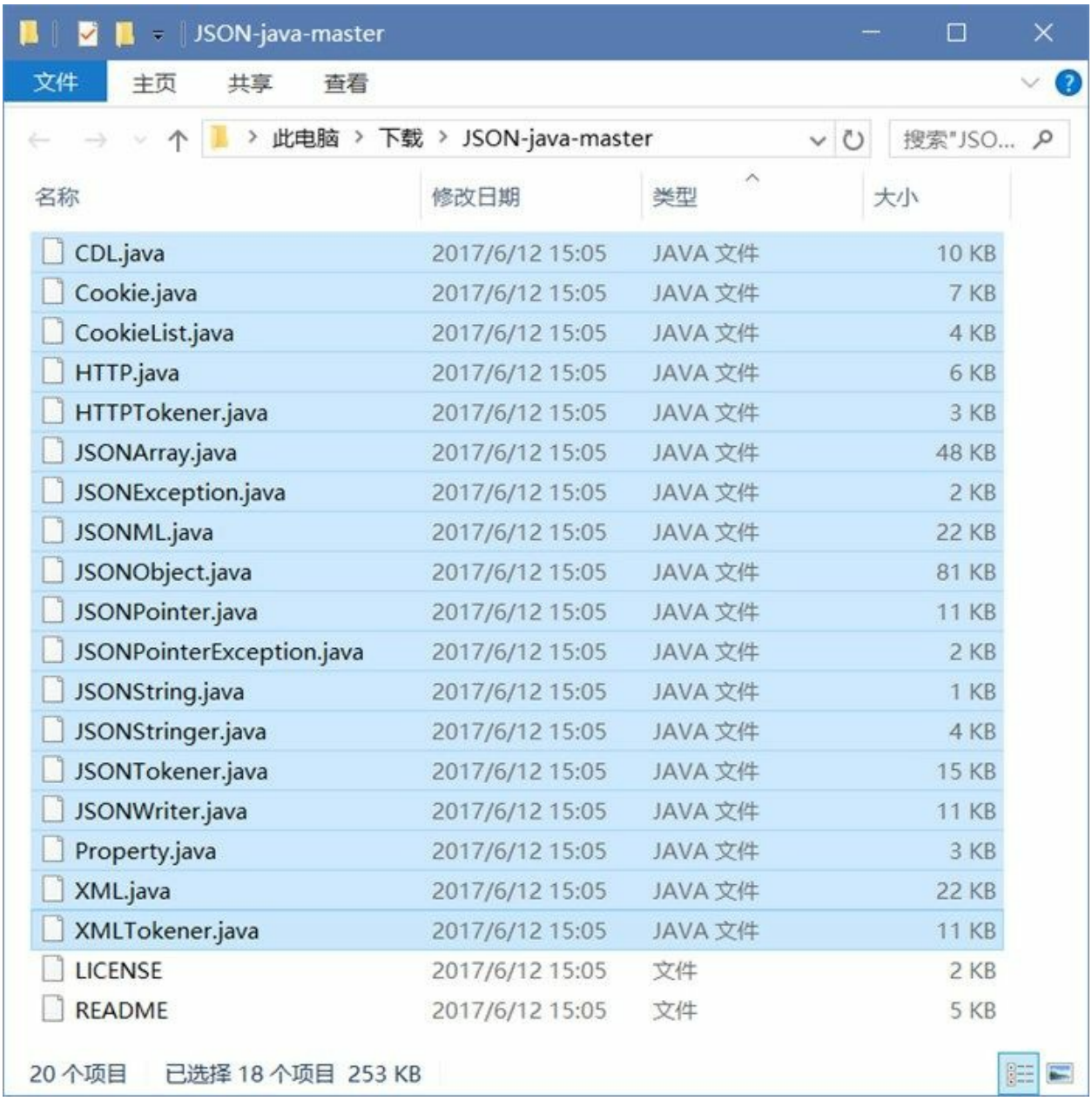


图**24-15 JSON-java**源代码文件

将JSON-java库源代码文件添加到工程中，需要两个步骤：

01. 创建org.json包

    JSON-java库中的源代码文件都隶属于org.json包，从图24-15可见源文件夹下没有与包对应的目录结构，为此需要在Eclipse的项目中创建org.json包。选择Eclipse项目的src源代码文件夹，右击菜单中选择“新建”→“包”，弹出新建包对话框，如图24-16所示在名称的中输入org.json，然后单击完成，就成功创建org.json包。

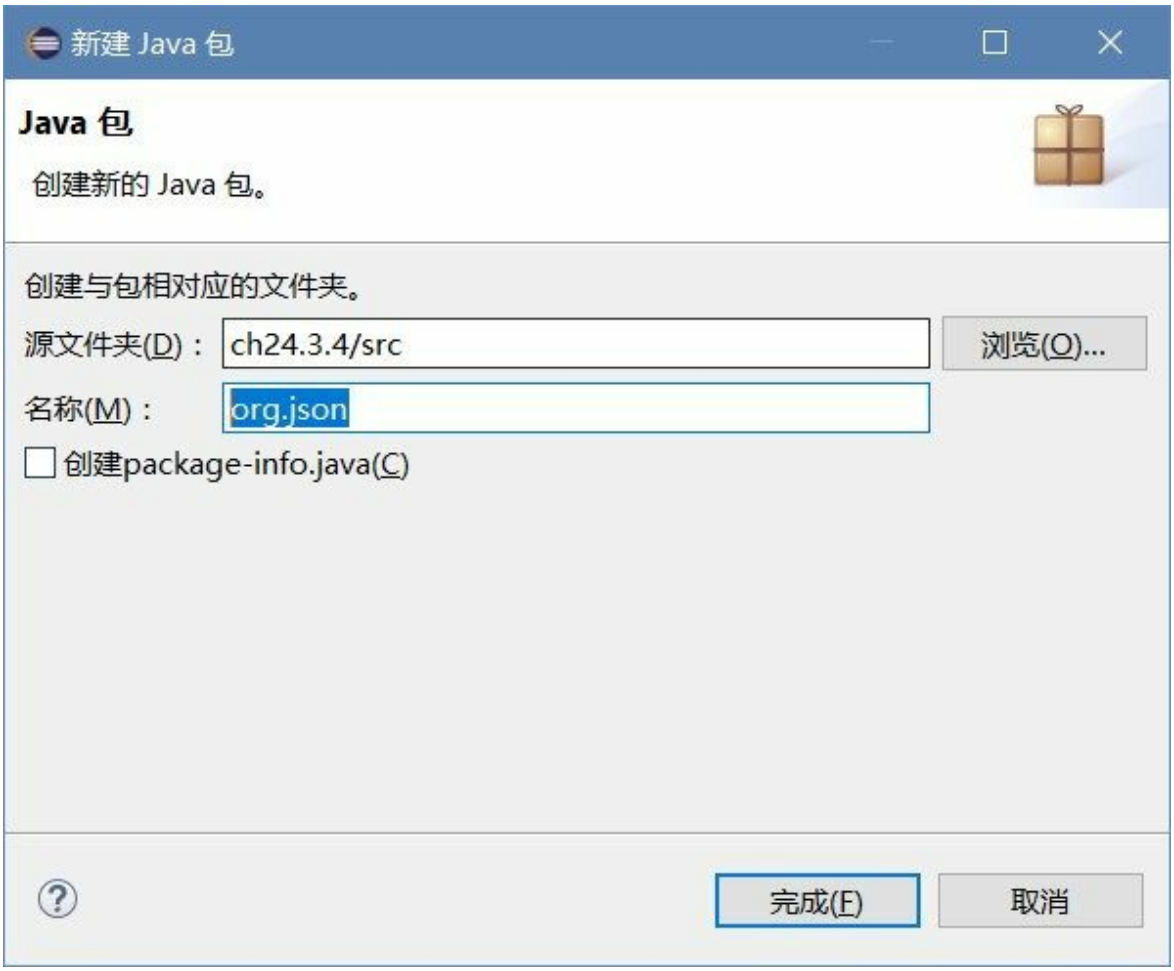


图**24-16**　在**Eclipse**中创建包

02. 复制源代码文件

org.json包创建好后，需要将JSON-java库文件夹中的源代码文件复制到Eclipse工程的org.json包中。由于操作系统的资源管理器与Eclipse工具之间互相复制粘贴，Eclipse中复制和粘贴操作的快捷键和右键菜单与操作系统下完全一样。如图24-17所示，将源代码文件复制到Eclipse中。

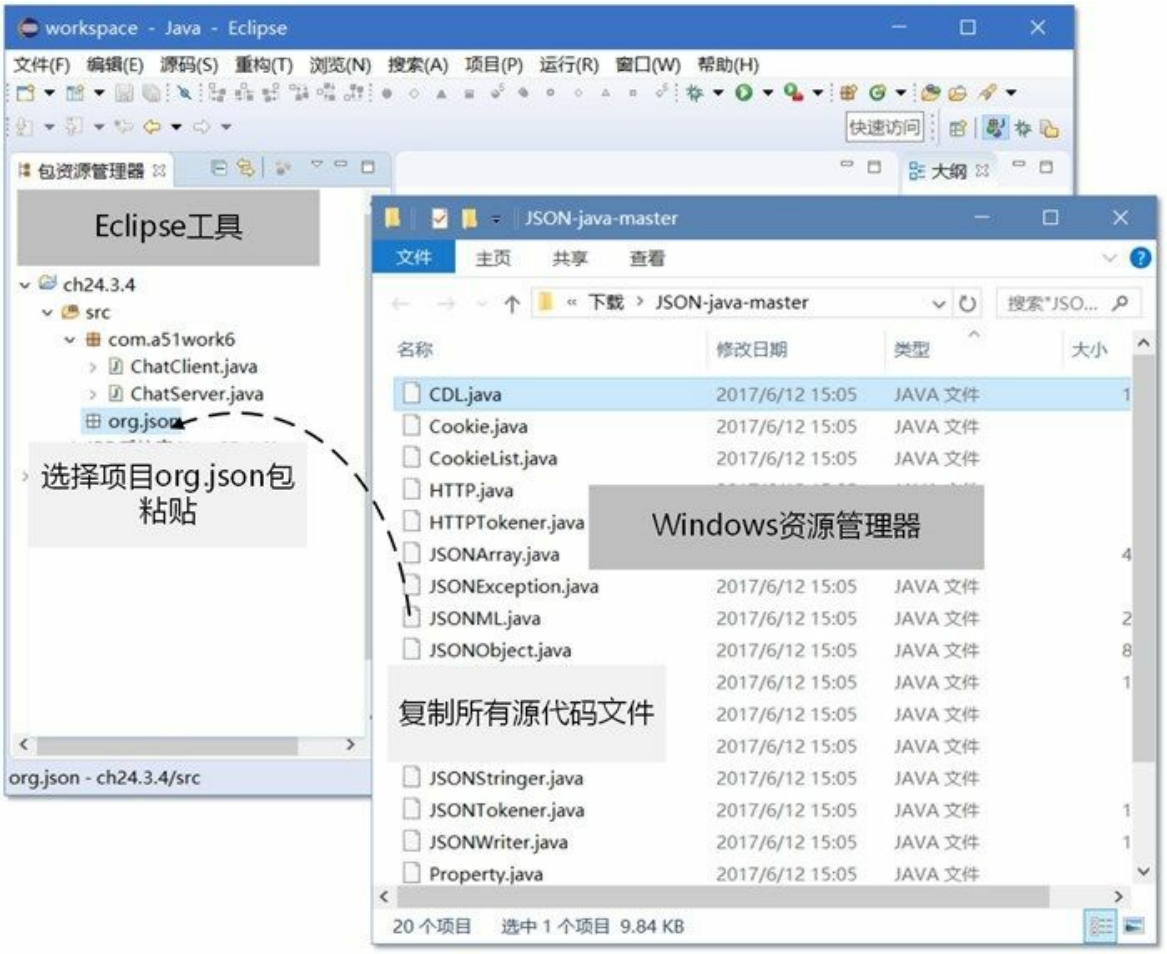


图**24-17** 复制源代码文件到**Eclipse**工程

24.4.3 JSON数据编码和解码

JSON和XML真正在进行数据交换时候，它们存在的形式就是一个很长的字符串，这个字符串在网络中传输或者存储于磁盘等介质中。在传输和存储之前需要把JSON对象转换成为字符串才能传输和存储，这个过程称之为“编码”过程。接收方需要将接收到的字符串转换成为JSON对象，这个过程称之为“解码”过程。编码和解码过程就像发电报时发送方把语言变成能够传输的符号，而接收时要将符号转换成为能够看懂的语言。

下面具体介绍一下JSON数据编码和解码过程。

01. 编码

如果想获得如下这样JSON字符串：

```
{"name": "tony", "age": 30, "a": [1, 3]}
```

应该如何实现编码过程，参考代码如下：

```
try {
    JSONObject jsonObject = new JSONObject();     ①
    jsonObject.put("name", "tony");               ②
```

```
    jsonObject.put("age", 30);                    ③

    JSONArray jsonArray = new JSONArray();        ④
    jsonArray.put(1).put(3);                      ⑤
    jsonObject.put("a", jsonArray);               ⑥
    //编码完成
    System.out.println(jsonObject.toString())     ⑦
} catch (JSONException e) {
    e.printStackTrace();
}
```

上述代码第①行是创建JSONObject（JSON对象），代码第②行和第③行是把JSON数据项添加到JSON对象jsonObject中，代码第④行创建JSONArray（JSON数组），代码第⑤行是向JSON数组中添加1和3两个元素。代码第⑥是将JSON数组jsonArray作为JSON对象jsonObject的数据项添加到JSON对象。

代码第⑦行jsonObject.toString()是将JSON对象转换为字符串，真正完成了JSON编码过程。

02. 解码

解码过程是编码反向操作，如果有如下JSON字符串：

```
{"name":"tony", "age":30, "a":[1, 3]}
```

那么如何把这个JSON字符串解码成JSON对象或数组，参考代码如下：

```
String jsonString = "{\"name\":\"tony\", \"age\":30, \"a\":[1, 3]}";    ①
try {
    JSONObject jsonObject = new JSONObject(jsonString);    ②
    String name = jsonObject.getString("name");            ③
    System.out.println("name : " + name);
    int age = jsonObject.getInt("age");
    System.out.println("age : " + age);
    JSONArray jsonArray = jsonObject.getJSONArray("a");    ④
    int n1 = jsonArray.getInt(0);                          ⑤
    System.out.println("数组a第一个元素 : " + n1);
    int n2 = jsonArray.getInt(1);
    System.out.println("数组a第二个元素 : " + n2);
} catch (JSONException e) {
    e.printStackTrace();
}
```

上述代码第①行是声明一个JSON字符串，网络通信过程中JSON字符串是从服务器返回的。代码第②行通过JSON字符串创建JSON对象，这个过程事实上就是JSON字符串解析过程，如果能够成功地创建JSON对象，说明解析成功，如果发生异常则说明解析失败。

代码第③行从JSON对象中按照名称取出JSON中对应的数据。代码第④行是取出一个JSON数组对象，代码第⑤行取出JSON数组第一个元素。

注意　如果按照规范的JSON文档要求，每个JSON数据项目的“名称”必须使用双引号括起来，不能使用单引号或没有引号。在下面的代码文档中，“名称”省略了双引号，该文档在其他平台解析时会出现异常，而在Java平台则可以通过，这得益于Java解析类库的强大，但这并不是规范的做法。如果与其他平台进行数据交换时，采用这种不规范的JSON文档进行数据交换，那么很有可能会导致严重的问题发生。

```
{ResultCode:0,Record:[
```

```
    {ID:'1',CDate:'2012-12-23',Content:'发布iOSBook0',UserID:'tony'},
    {ID:'2',CDate:'2012-12-24',Content:'发布iOSBook1',UserID:'tony'}]}。
```

24.4.4 案例：聊天工具

为了进一步熟悉JSON数据交换格式，将24.2.6节的聊天工具修改为使用JSON进行数据交换。

客户端与服务器之间采用JSON数据交换格式，JSON格式内部结构是自定义的，代码如下：

```
{"message":"Hello","userid":"javaee","username":"关东升"}
```

下面看看案例服务器端ChatServer代码如下：

```
//ChatServer.java文件
package com.a51work6;
…
import org.json.JSONObject;

public class ChatServer {

    public static void main(String[] args) {

        System.out.println("服务器运行...");

        Thread t = new Thread(() -> {

            try ( // 创建一个ServerSocket监听端口8080客户请求
                    ServerSocket server = new ServerSocket(8080);
                    // 使用accept()阻塞等待客户端请求
                    Socket socket = server.accept();
                    DataInputStream in = new DataInputStream(socket.getInputStream());
                    DataOutputStream out = new DataOutputStream(socket.getOutputStream());
                    BufferedReader keyboardIn
                            = new BufferedReader(new InputStreamReader(System.in))) {

                while (true) {
                    /* 接收数据 */
                    String str = in.readUTF();
                    // JSON解码
                    JSONObject jsonObject = new JSONObject(str);                    ①
                    // 打印接收的数据
                    System.out.printf("从客户端接收的数据：%s\n", jsonObject);     ②

                    /* 发送数据 */
                    // 读取键盘输入的字符串
                    String keyboardInputString = keyboardIn.readLine();
                    // 结束聊天
                    if (keyboardInputString.equals("bye")) {
                        break;
                    }
                    // 编码
                    jsonObject = new JSONObject();                                  ③
                    jsonObject.put("message", keyboardInputString);                 ④
                    jsonObject.put("userid", "acid");                               ⑤
                    jsonObject.put("username", "赵1");                              ⑥
                    // 发送
                    out.writeUTF(jsonObject.toString());                            ⑦
                    out.flush();
                }
            } catch (Exception e) {
            }
```

```
            System.out.println("服务器停止...");
        });

        t.start();
    }
}
```

上述代码第①行是对，从服务器返回的字符串进行解码，并返回JSON对象，注意要解码的字符串应该是有效的JSON字符串。代码第②行是打印JSON对象。

代码第③行是创建JSON对象，代码第④行~第⑥行是添加JSON对象。代码第⑦行jsonObject.toString()语句是将JSON对象转换为JSON字符串。

下面看看案例客户端ChatClient代码如下：

```
//ChatClient.java文件
package com.a51work6;
…
import org.json.JSONObject;

public class ChatClient {

    public static void main(String[] args) {

        System.out.println("客户端运行...");

        Thread t = new Thread(() -> {

            try ( // 向127.0.0.1主机8080端口发出连接请求
                    Socket socket = new Socket("127.0.0.1", 8080);
                    DataInputStream in = new DataInputStream(socket.getInputStream());
                    DataOutputStream out = new DataOutputStream(socket.getOutputStream());
                    BufferedReader keyboardIn
                            = new BufferedReader(new InputStreamReader(System.in))) {

                while (true) {
                    /* 发送数据 */
                    // 读取键盘输入的字符串
                    String keyboardInputString = keyboardIn.readLine();
                    // 结束聊天
                    if (keyboardInputString.equals("bye")) {
                        break;
                    }
                    JSONObject jsonObject = new JSONObject();
                    jsonObject.put("message", keyboardInputString);
                    jsonObject.put("userid", "javaee");
                    jsonObject.put("username", "关东升");

                    // 发送
                    out.writeUTF(jsonObject.toString());
                    out.flush();

                    /* 接收数据 */
                    String str = in.readUTF();
                    jsonObject = new JSONObject(str);
                    // 打印接收的数据
                    System.out.printf("从服务器接收的数据：%s \n", str);
                }
            } catch (ConnectException e) {
                System.out.println("服务器未启动！");
            } catch (Exception e) {
            }
            System.out.println("客户端停止！");
```

```
        });
        t.start();
    }
}
```

客户端ChatClient代码与服务器端ChatServer代码类似，这里不再赘述。

24.5 访问互联网资源

Java的java.net包中还提供了高层次网络编程类——URL，通过URL类访问互联网资源。使用URL进行网络编程，不需要对协议本身有太多的了解，相对而言是比较简单的。

24.5.1 URL概念

互联网资源是通过URL指定的，URL是Uniform Resource Locator简称，翻译过来是“一致资源定位器”，但人们都习惯URL简称。

URL组成格式如下：

```
协议名://资源名
```

“协议名”指明获取资源所使用的传输协议，如http、ftp、gopher和file等，“资源名”则应该是资源的完整地址，包括主机名、端口号、文件名或文件内部的一个引用。例如：

```
http://www.sina.com/
http://home.sohu.com/home/welcome.html
http://www.51work6.com:8800/Gamelan/network.html#BOTTOM
```

24.5.2 HTTP/HTTPS协议

访问互联网大多都基于HTTP/HTTPS协议。下面介绍一下HTTP/HTTPS协议。

01. HTTP协议

    HTTP是Hypertext Transfer Protocol的缩写，即超文本传输协议。HTTP是一个属于应用层的面向对象的协议，其简捷、快速的方式适用于分布式超文本信息的传输。它于1990年提出，经过多年的使用与发展，得到不断完善和扩展。HTTP协议支持C/S网络结构，是无连接协议，即每一次请求时建立连接，服务器处理完客户端的请求后，应答给客户端然后断开连接，不会一直占用网络资源。

    HTTP/1.1协议共定义了8种请求方法：OPTIONS、HEAD、GET、POST、PUT、DELETE、TRACE和CONNECT。在HTTP访问中，一般使用GET和HEAD方法。

    - GET方法：是向指定的资源发出请求，发送的信息“显式”地跟在URL后面。GET方法应该只用在读取数据，例如静态图片等。GET方法有点像使用明信片给别人写信，“信内容”写在外面，接触到的人都可以看到，因此是不安全的。
    - POST方法：是向指定资源提交数据，请求服务器进行处理，例如提交表单或者上传文件等。数据被包含在请求体中。POST方法像是把“信内容”装入信封中，接触到的人都看不到，因此是安全的。

02. HTTPS协议

    HTTPS是Hypertext Transfer Protocol Secure，即超文本传输安全协议，是超文本传输协议和SSL的组合，用以提供加密通信及对网络服务器身份的鉴定。

    简单地说，HTTPS是HTTP的升级版，HTTPS与HTTP的区别是：HTTPS使用https://代替http://，HTTPS使用端口443，而HTTP使用端口80来与TCP/IP进行通信。SSL使用40位关键字作为RC4流加密算法，这对于商业信息的加密是合适的。HTTPS和SSL支持使用X.509数字认证，如果需要

的话，用户可以确认发送者是谁。

24.5.3 使用URL类

Java 的java.net.URL类用于请求互联网上的资源，采用HTTP/HTTPS协议，请求方法是GET方法，一般是请求静态的、少量的服务器端数据。

URL类常用构造方法：

- URL(String spec)：根据字符串表示形式创建URL对象。
- URL(String protocol, String host, String file)：根据指定的协议名、主机名和文件名称创建URL对象。
- URL(String protocol, String host, int port, String file)：根据指定的协议名、主机名、端口号和文件名称创建URL对象。

URL类常用方法：

- InputStream openStream()：打开到此URL的连接，并返回一个输入流。
- URLConnection openConnection()：打开到此URL的新连接，返回一个URLConnection对象。

下面通过一个示例介绍一下如何使用java.net.URL类，示例代码如下：

```
//HelloWorld.java文件
package com.a51work6;
...
public class HelloWorld {

    public static void main(String[] args) {
        // Web网址
        String url = "http://www.sina.com.cn/";

        URL reqURL;
        try {
            reqURL = new URL(url);                                    ①
        } catch (MalformedURLException e1) {
            return;
        }

        try ( // 打开网络通信输入流
                InputStream is = reqURL.openStream();                 ②
                InputStreamReader isr = new InputStreamReader(is, "utf-8");
                BufferedReader br = new BufferedReader(isr)) {

            StringBuilder sb = new StringBuilder();
            String line = br.readLine();
            while (line != null) {
                sb.append(line);
                sb.append('\n');
                line = br.readLine();
            }
            // 日志输出
            System.out.println(sb);

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

上述代码第①行创建URL对象，参数是一个HTTP网址。代码第②行通过URL对象的openStream()方法打开输入流。

24.5.4 使用HttpURLConnection发送GET请求

由于URL类只能发送HTTP/HTTPS的GET方法请求，如果要想发送其他的情况或者对网络请求有更深入的控制时，可以使用HttpURLConnection类型。

示例代码如下：

```
//HelloWorld.java文件
package com.a51work6;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;

public class HelloWorld {

    // Web服务网址
    static String urlString = "http://www.51work6.com/service/mynotes/WebService.php?"
            + "email=<换成你在51work6.com注册时填写的邮箱>&type=JSON&action=query"; ①

    public static void main(String[] args) {

        BufferedReader br = null;
        HttpURLConnection conn = null;

        try {
            URL reqURL = new URL(urlString);
            conn = (HttpURLConnection) reqURL.openConnection();            ②
            conn.setRequestMethod("GET");                                  ③

            // 打开网络通信输入流
            InputStream is = conn.getInputStream();                        ④
            // 通过is创建InputStreamReader对象
            InputStreamReader isr = new InputStreamReader(is, "utf-8");
            // 通过isr创建BufferedReader对象
            br = new BufferedReader(isr);

            StringBuilder sb = new StringBuilder();
            String line = br.readLine();
            while (line != null) {
                sb.append(line);
                line = br.readLine();
            }
            // 日志输出
            System.out.println(sb);

        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if (conn != null) {
                conn.disconnect();                                         ⑤
            }
            if (br != null) {
                try {
                    br.close();
```

```
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

上述代码第①行是一个Web服务网址字符串。

提示　发送GET请求时发送给服务器的参数是放在URL的“?”之后，参数采用键值对形式，例如：第①行的URL中type=JSON是一个参数，type是参数名，JSON是参数名，服务器端会根据参数名获得参数值。多个参数之间用“&”分隔，例如type=JSON&action=query就是两个参数。

代码第②行是用reqURL.openConnection()方法打开一个连接，返回URLConnection对象，由于本次连接是HTTP连接，所以返回的是HttpURLConnection对象。URLConnection是抽象类，HttpURLConnection是URLConnection的子类。

代码第③行conn.setRequestMethod("GET")是设置请求方法为GET方法。代码第④行是通过conn.getInputStream()打开输入流，上一节实例使用的URL的openStream()方法获得输入流。代码第⑤行conn.disconnect()是断开连接，这可以释放资源。

从服务器端返回的数据是JSON字符串，格式化后内容如下：

```
{
    "ResultCode": 0,
    "Record": [
        {
            "ID": 5238,
            "CDate": "2017-05-18",
            "Content": "欢迎来到智捷课堂。"
        },
        {
            "ID": 5239,
            "CDate": "2018-10-18",
            "Content": "Welcome to zhijieketang."
        }
    ]
}
```

提示　上述示例中URL所指向的Web服务是由作者所在的智捷课堂提供的，读者要想使用这个Web服务需要在www.51work6.com进行注册，注册时需要提供自己有效的邮箱，这个邮箱用来激活用户。在网络请求时需要提交email参数，这个参数是注册时填写的邮箱。

24.5.5　使用HttpURLConnection发送POST请求

HttpURLConnection也可以发送HTTP/HTTPS的POST请求，下面介绍如何使用HttpURLConnection发送POST请求。

示例代码如下：

```
//HelloWorld.java文件
package com.a51work6;

import java.io.BufferedReader;
import java.io.DataOutputStream;
import java.io.IOException;
```

```
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;

public class HelloWorld {

    // Web服务网址
    static String urlString = "http://www.51work6.com/service/mynotes/WebService.php";    ①

    public static void main(String[] args) {

        BufferedReader br = null;
        HttpURLConnection conn = null;
        try {
            URL reqURL = new URL(urlString);
            conn = (HttpURLConnection) reqURL.openConnection();            ②
            conn.setRequestMethod("POST");                                 ③
            conn.setDoOutput(true);                                        ④

            String param = String.format("email=%s&type=%s&action=%s",
                              "<换成你在51work6.com注册时填写的邮箱>", "JSON", "query");  ⑤
            // 设置参数
            DataOutputStream dStream = new DataOutputStream(conn.getOutputStream());   ⑥
            dStream.writeBytes(param);            ⑦
            dStream.close();                    ⑧

            // 打开网络通信输入流
            InputStream is = conn.getInputStream();
            // 通过is创建InputStreamReader对象
            InputStreamReader isr = new InputStreamReader(is, "utf-8");
            // 通过isr创建BufferedReader对象
            br = new BufferedReader(isr);

            StringBuilder sb = new StringBuilder();
            String line = br.readLine();
            while (line != null) {
                sb.append(line);
                line = br.readLine();
            }
            // 日志输出
            System.out.println(sb);

        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if (conn != null) {
                conn.disconnect();
            }
            if (br != null) {
                try {
                    br.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

上述代码第①行URL后面不带参数，这是因为要发送的是POST请求，POST请求参数是放在请求体中。代码第②行是通过reqURL.openConnection()是建立HTTP连接，代码第③行是设置HTTP请求方法为POST，代码第④行conn.setDoOutput(true)是设置请求过程可以传递参数给服务器。

代码第⑤是设置请求参数格式化字符串"email=%s&type=%s&action=%s"，其中%s是占位符。

代码第⑥行～第⑧行是将请求参数发送给服务器，代码第⑥行中conn.getOutputStream()是打开输出流，new DataOutputStream(conn.getOutputStream())是创建基于数据输出流。代码第⑦行dStream.writeBytes(param)是向输出流中写入数据，第⑧行dStream.close()是关闭流，并将数据写入到服务器端。

24.5.6 实例：Downloader

为了进一步熟悉URL类，这一节介绍一个下载程序Downloader。Downloader.java代码如下：

```
//Downloader.java文件
package com.a51work6;

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.HttpURLConnection;
import java.net.URL;

public class Downloader {

    // Web服务网址
    private static String urlString = "https://ss0.bdstatic.com/5aV1bjqh_Q23odCf/"
            + "static/superman/img/logo/bd_logo1_31bdc765.png";

    public static void main(String[] args) {
        download();
    }

    // 下载方法
    private static void download() {

        HttpURLConnection conn = null;

        try {
            // 创建URL对象
            URL reqURL = new URL(urlString);
            // 打开连接
            conn = (HttpURLConnection) reqURL.openConnection();                    ①

            try (// 从连接对象获得输入流
                    InputStream is = conn.getInputStream();                        ②
                    BufferedInputStream bin = new BufferedInputStream(is);         ③
                    // 创建文件输出流
                    OutputStream os = new FileOutputStream("./download.png");      ④
                    BufferedOutputStream bout = new BufferedOutputStream(os);) {   ⑤

                byte[] buffer = new byte[1024];
                int bytesRead = bin.read(buffer);
                while (bytesRead != -1) {
                    bout.write(buffer, 0, bytesRead);
                    bytesRead = bin.read(buffer);
                }
            } catch (IOException e) {
            }
            System.out.println("下载完成。");
        } catch (IOException e) {
        } finally {
            if (conn != null) {
                conn.disconnect();
```

```
            }
        }
    }
}
```

上述代码第①行打开连接获得HttpURLConnection对象。代码第②行是从连接对象获得输入流。代码第③行创建缓冲流输入流，使用缓冲流可以提高读写效率。

代码第④行是创建文件输出流，代码第⑤行是创建缓冲流输出流。

运行Downloader程序，如果成功会在当前目录获得一张图片。

本章小结

本章主要介绍了Java网络编程，首先介绍了一些网络方面的基本知识。然后重点介绍了TCP Socket编程和UDP Socket编程，其中TCP Socket编程很有代表性希望读者重点掌握这部分知识。接着介绍了数据交换格式，重点介绍了JSON数据交换格式，由于Java官方没有提供JSON解码和编码库，需要是使用第三方库。最后介绍了使用URL类访问互联网资源。

第 25 章　Swing图形用户界面编程

图形用户界面（Graphical User Interface，简称 GUI）编程对于某种语言来说非常重要。Java的应用主要方向是基于Web浏览器的应用，用户界面主要是HTML、CSS和JavaScript等基于Web的技术，这些介绍要到Java EE阶段才能学习到。

而本章介绍的Java图形用户界面技术是基于Java SE的Swing，事实上它们在实际应用中使用不多，因此本章的内容只做了解。

25.1 Java图形用户界面技术

Java图形用户界面技术主要有：AWT、Applet、Swing和JavaFX。

01. AWT

    AWT（Abstract Window Toolkit）是抽象窗口工具包，AWT是Java 程序提供的建立图形用户界面最基础的工具集。AWT支持图形用户界面编程的功能包括：用户界面组件（控件）、事件处理模型、图形图像处理（形状和颜色）、字体、布局管理器和本地平台的剪贴板来进行剪切和粘贴等。AWT是Applet和Swing技术的基础。

    AWT在实际的运行过程中是调用所在平台的图形系统，因此同样一段AWT程序在不同的操作系统平台下运行所看到的样式不同的。例如在Windows下运行，则显示的窗口是Windows风格的窗口，如图25-1所示，而在UNIX下运行时，则显示的是UNIX风格的窗口，如图25-2所示的macOS[1]是AWT窗口。

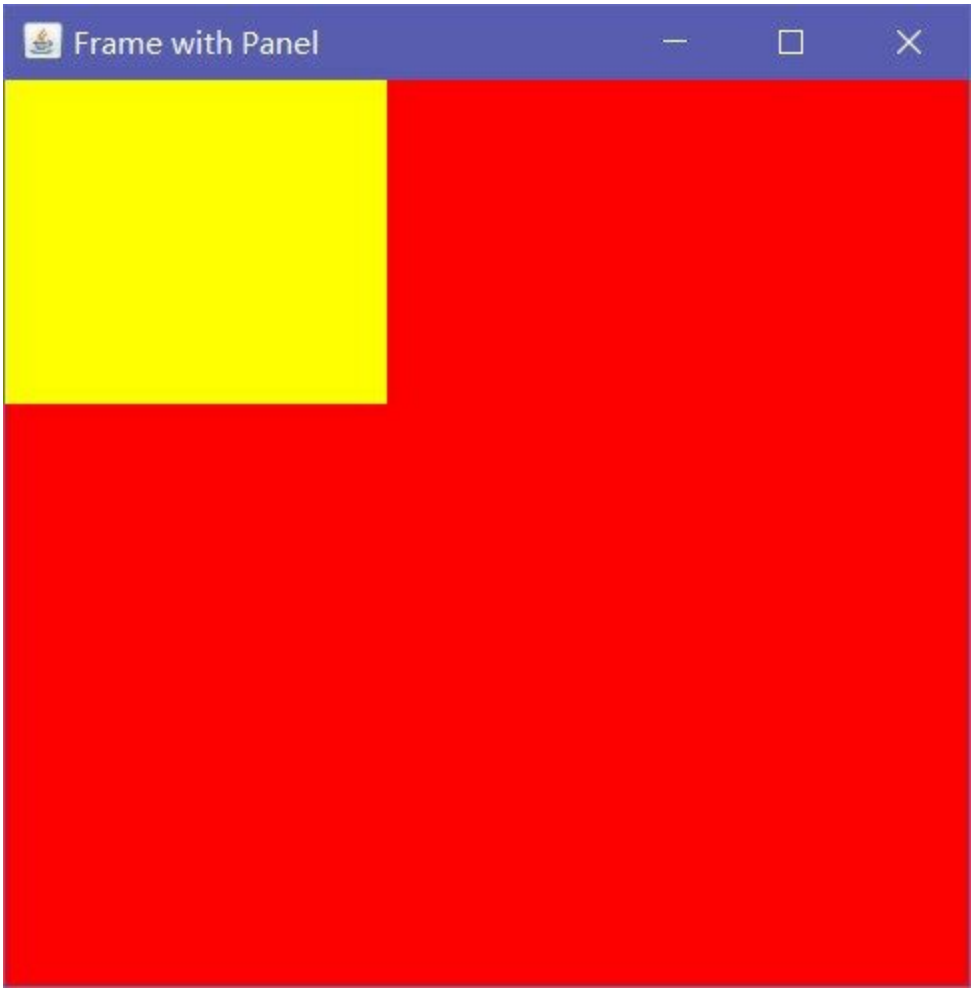


图**25-1　Windows**风格的**AWT**窗口

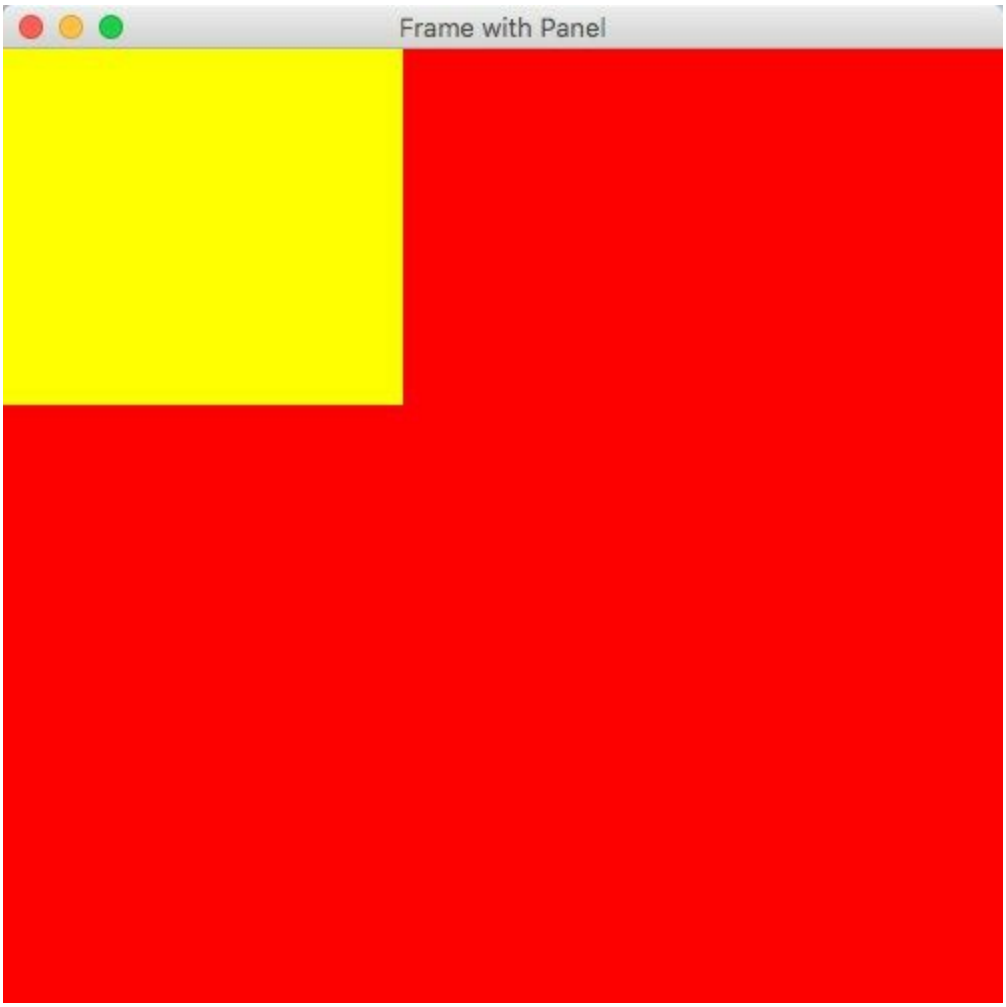


图**25-2　macOS**风格的**AWT**窗口

02. Applet

    Applet称为Java小应用程序，Applet基础是AWT，但它主要嵌入到HTML代码中，由浏览器加载和运行，由于存在安全隐患和运行速度慢等问题，已经很少使用了。

03. Swing

    Swing是Java主要的图形用户界面技术，Swing提供跨平台的界面风格，用户可以自定义Swing的界面风格。Swing提供了比AWT更完整的组件，引入了许多新的特性。Swing API是围绕着实现AWT各个部分的API构筑的。Swing是由100%纯Java实现的，Swing组件没有本地代码，不依赖操作系统的支持，这是它与AWT组件的最大区别。本章重点介绍Swing技术。

04. JavaFX

    JavaFX是开发丰富互联网应用程序（Rich Internet Application，缩写RIA）的图形用户界面技术，JavaFX期望能够在桌面应用的开发领域与Adobe公司的AIR、微软公司的Silverlight相竞争。传统的互联网应用程序基于Web的，客户端是浏览器。而丰富互联网应用程序试图打造自己的客户端，替代浏览器。

[1]macOS是苹果计算机操作系统，它也是UNIX内核。

25.2 Swing技术基础

AWT是Swing的基础，Swing事件处理和布局管理都是依赖于AWT，AWT内容来自java.awt包，Swing内容来自javax.swing包。AWT和Swing作为图形用户界面技术包括了4个主要的概念：组件（Component）、容器（Container）、事件处理和布局管理器（LayoutManager），下面将围绕这些概念展开。

25.2.1 Swing类层次结构

容器和组件构成了Swing的主要内容，下面分别介绍一下Swing中容器和组件类层次结构。

图25-3所示是Swing容器类层次结构，Swing容器类主要有：JWindow、JFrame和JDialog，其他的不带“J”开头都是AWT提供的类，在Swing中大部分类都是以“J”开头。

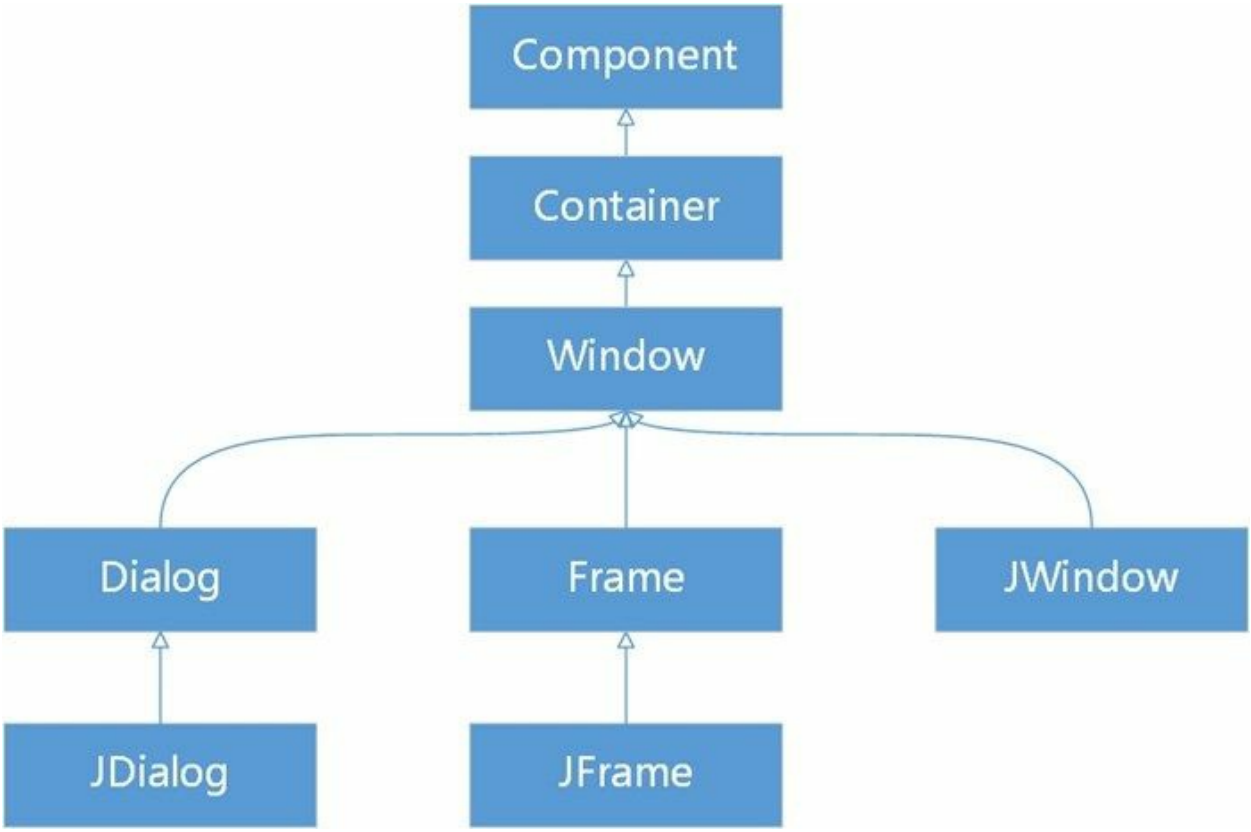


图**25-3 Swing**容器类层次结构

图25-4所示是Swing组件类层次结构，Swing所有组件继承自JComponent，JComponent又间接继承自AWT的java.awt.Component类。Swing组件很多，这里不一一解释了，在后面的学习过程中会重点介绍一下组件。

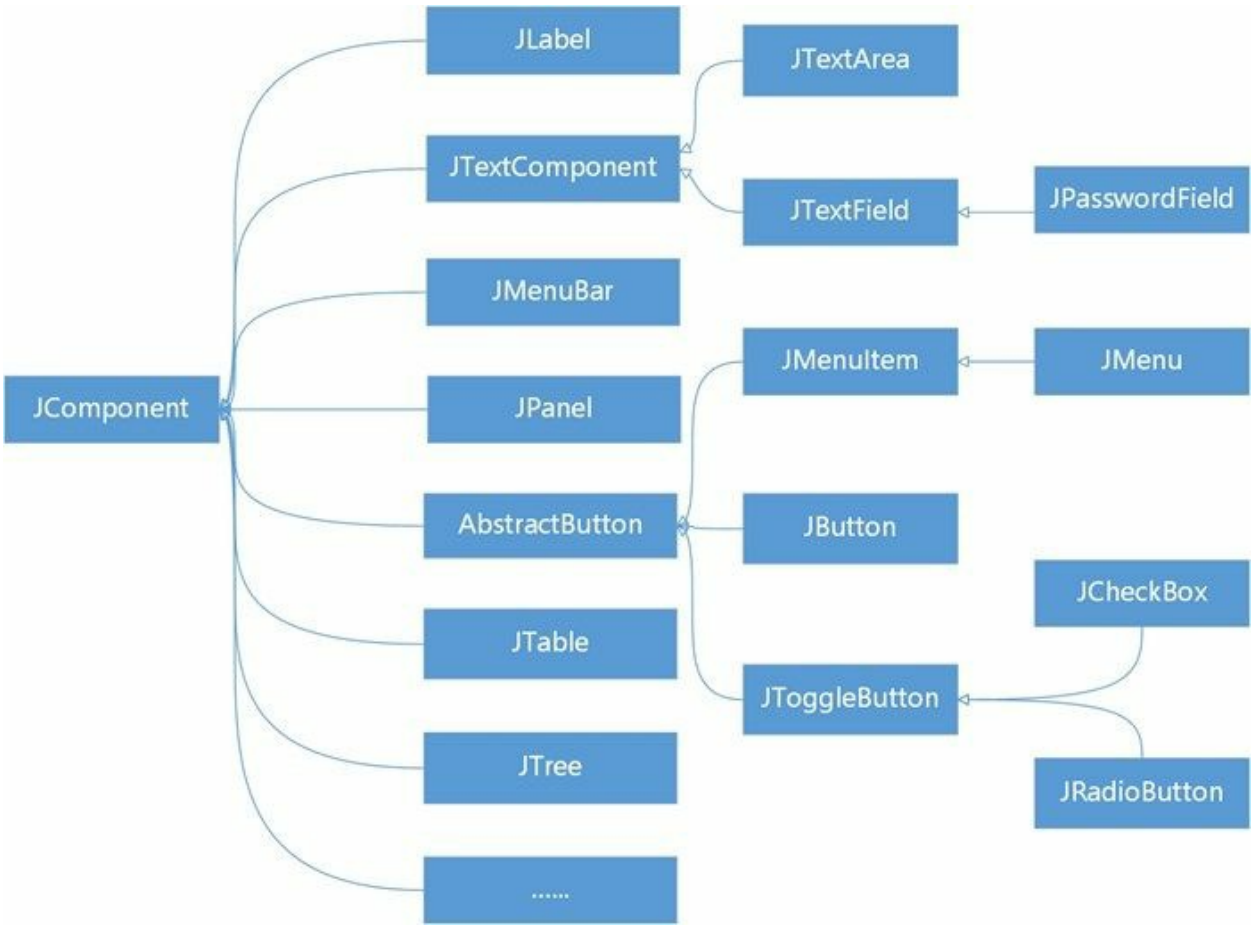


图**25-4　Swing**组件类层次结构

25.2.2　Swing程序结构

图形用户界面主要是由窗口以及窗口中的组件构成的，编写Swing程序主要就是创建窗口和添加组件过程。Swing中的窗口主要是使用JFrame，很少使用JWindow。JFrame有标题、边框、菜单、大小和窗口管理按钮等窗口要素，而JWindow没有标题栏和窗口管理按钮。

构建Swing程序主要有两种方式：创建JFrame或继承JFrame。下面通过一个示例介绍一下这两种方式如何实现，该示例运行效果如图25-5所示，窗口标题是MyFrame，窗口中有显示字符串“Hello Swing！”。

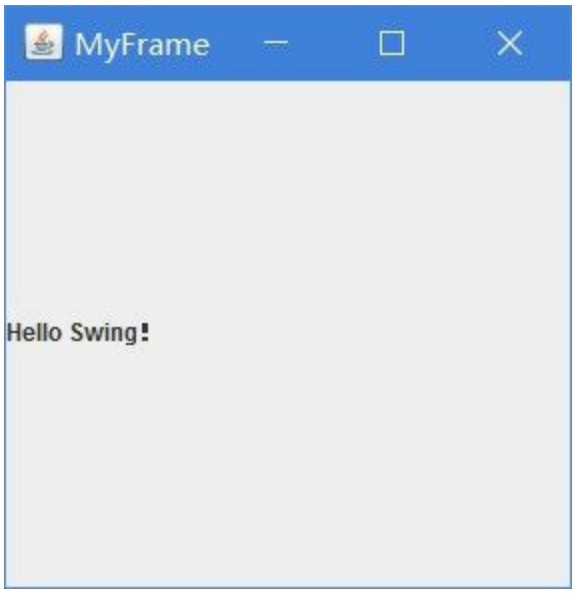


图**25-5　Swing**示例运行效果

01. 创建JFrame方式

创建JFrame方式就是直接实例化JFrame对象，然后设置JFrame属性，添加窗口所需要的组件。

示例代码如下：

```
// SwingDemo1.java文件
package com.a51work6;

import java.awt.Container;

import javax.swing.JFrame;
import javax.swing.JLabel;

public class SwingDemo1 {

    public static void main(String[] args) {
        //创建窗口对象
        JFrame frame = new JFrame("MyFrame");                    ①

        // 创建标签
        JLabel label = new JLabel("Hello Swing! ");              ②
        // 获得窗口的内容面板
        Container contentPane = frame.getContentPane();          ③
        // 添加标签到内容面板
        contentPane.add(label);                                  ④

        // 设置窗口大小
        frame.setSize(300, 300);                                 ⑤
        // 设置窗口可见
        frame.setVisible(true);                                  ⑥
    }
}
```

上述代码第①行使用JFrame的JFrame(String title)构造方法创建JFrame对象，title是设置创建的标题。默认情况下JFrame是没有大小且不可见的，因此创建JFrame对象后还需要设置大小和可见，代码第⑤行是设置窗口大小，代码第⑥行是设置窗口的可见。

注意　设置JFrame窗口大小和可见这两条语句，应该在添加完成所有组件之后调用。否则在多个组件情况下，会导致有些组件没有显示。

创建好窗口后，就需要将其中的组件添加进来，代码第②行是创建标签对象，构造方法中字符串参数是标签要显示的文本。创建好组件之后需要把它添加到窗口的内容面板上，代码第③行是获得窗口的内容面板，它是Container容器类型。代码第④行调用容器的add()方法将组件添加到窗口上。

注意　在Swing中添加到JFrame上的所有可见组件，除菜单栏外，全部添加到内容面板上，而不要直接添加到JFrame上，这是Swing绘制系统所要求的。内容面板如图25-6所示，内容面板是JFrame中包含的一个子容器。

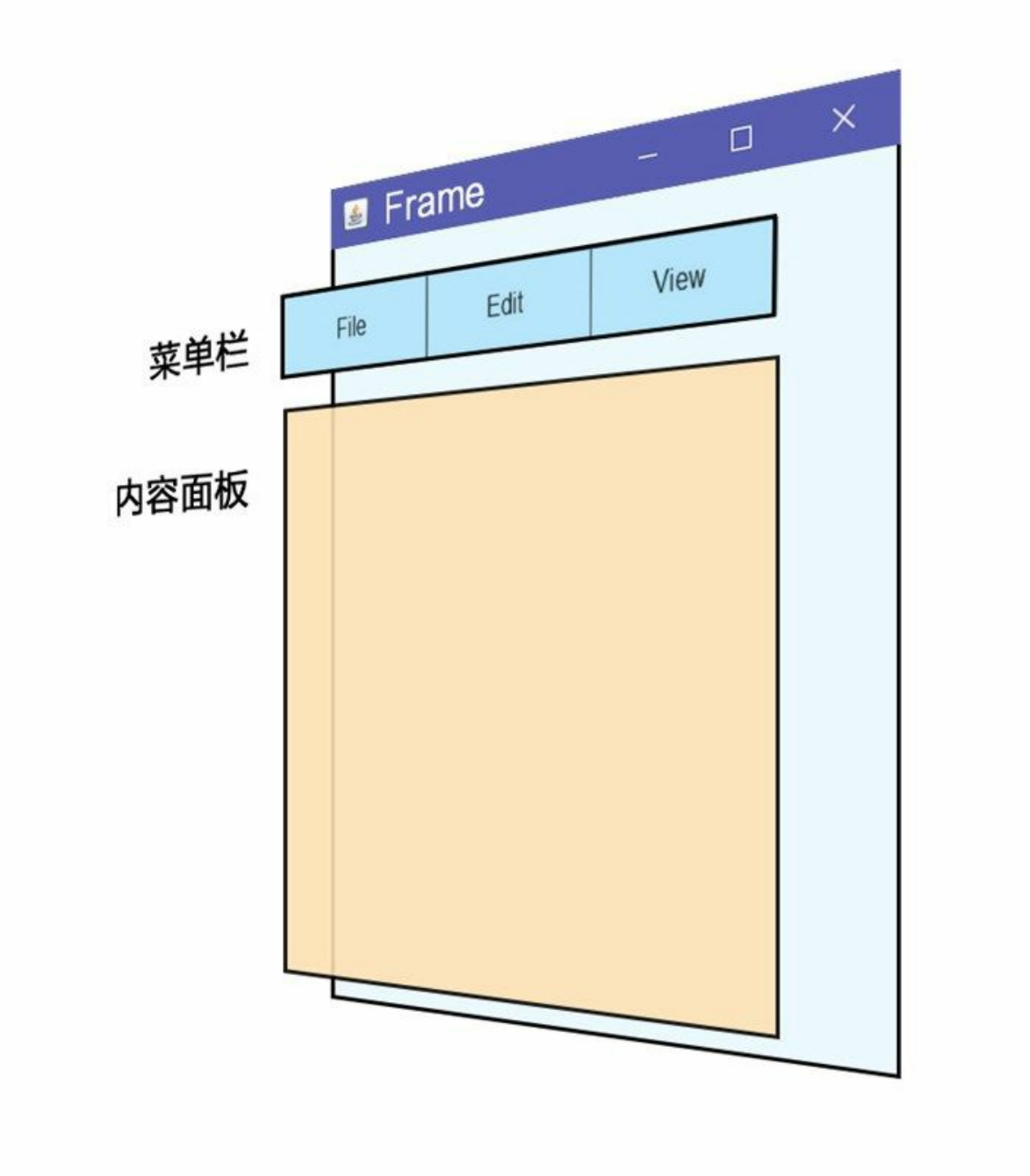


图**25-6 JFame**的内容面板

提示　几乎所有的图形用户界面技术，在构建界面时都采用层级结构（树形结构），如图25-7所示，根是顶级容器（只能包含其他容器的容器），子容器有内容面板和菜单栏（本例中没有菜单），然后其他的组件添加到内容面板容器中。所有的组件都有add()方法通过，通过调用add()方法将其他组件添加到容器中，作为当前容器的子组件。

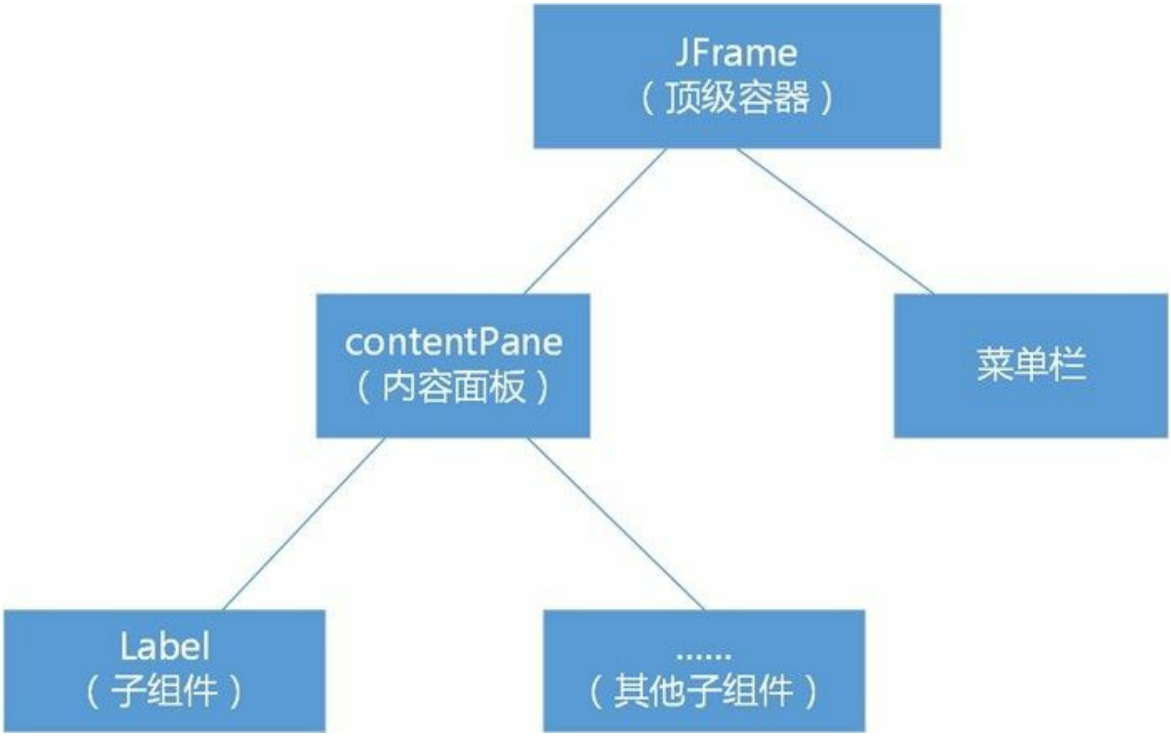


图**25-7**　界面构建层次

02. 继承JFrame方式

继承JFrame方式就是编写一个继承JFrame的子类，在构造方法中初始化窗口，添加窗口所需要的组件。

自定义窗口代码如下：

```
//MyFrame.java文件
package com.a51work6;

import java.awt.Container;

import javax.swing.JFrame;
import javax.swing.JLabel;

public class MyFrame extends JFrame {                ①

    public MyFrame(String title) {                    ②
        super(title);

        // 创建标签
        JLabel label = new JLabel("Hello Swing! ");
        // 获得窗口的内容面板
        Container contentPane = getContentPane();
        // 添加标签到内容面板
```

```
            contentPane.add(label);

            // 设置窗口大小
            setSize(300, 300);
            // 设置窗口可见
            setVisible(true);
        }
    }
```

上述代码第①行是声明MyFrame继承JFrame，代码第②行定义构造方法，参数是窗口标题。

调用代码如下：

```
    //SwingDemo2.java文件
    package com.a51work6;

    public class SwingDemo2{

        public static void main(String[] args) {
            //创建窗口对象
            new MyFrame("MyFrame");
        }
    }
```

运行上述代码可见继承JFrame方式和创建JFrame方式效果完全一样。

提示　创建JFrame方式适合于小项目，代码量少、窗口不多、组件少的情况。继承JFrame方式，适合于大项目，可以针对不同界面自定义一个Frame类，属性可以在构造方法中进行设置；缺点是有很多类文件需要有效地管理。

25.3 事件处理模型

图形界面的组件要响应用户操作，就必须添加事件处理机制。Swing采用AWT的事件处理模型进行事件处理。在事件处理的过程中涉及三个要素：

01. 事件：是用户对界面的操作，在Java中事件被封装称为事件类java.awt.AWTEvent及其子类，例如按钮单击事件类是java.awt.event.ActionEvent。

02. 事件源：是事件发生的场所，就是各个组件，例如按钮单击事件的事件源是按钮（Button）。

03. 事件处理者：是事件处理程序，在Java中事件处理者是实现特定接口的事件对象。

在事件处理模型中最重要的是事件处理者，它根据事件（假设XXXEvent事件）的不同会实现不同的接口，这些接口命名为XXXListener，所以事件处理者也称为事件监听器。最后事件源通过addXXXListener()方法添加事件监听，监听XXXEvent事件。各种事件和对应的监听器接口，如表25-1所示。

事件处理者可以是实现了XXXListener接口任何形式，即：外部类、内部类、匿名内部类和Lambda表达式；如果XXXListener接口只有一个抽象方法，事件处理者还可以是Lambda表达式。为了访问窗口中的组件方便，往往使用内部类、匿名内部类和Lambda表达式情况很多。

表 **25-1** 事件类型和事件监听器接口

事件类型	相应监听器接口	监听器接口中的方法
Action	ActionListener	actionPerformed(ActionEvent)
Item	ItemListener	itemStateChanged(ItemEvent)
Mouse	MouseListener	mousePressed(MouseEvent)
		mouseReleased(MouseEvent)
		mouseEntered(MouseEvent)
		mouseExited(MouseEvent)
		mouseClicked(MouseEvent)
Mouse Motion	MouseMotionListener	mouseDragged(MouseEvent)
		mouseMoved(MouseEvent)
Key	KeyListener	keyPressed(KeyEvent)
		keyReleased(KeyEvent)
		keyTyped(KeyEvent)
Focus	FocusListener	focusGained(FocusEvent)
		focusLost(FocusEvent)
Adjustment	AdjustmentListener	adjustmentValueChanged(AdjustmentEvent)
Component	ComponentListener	componentMoved(ComponentEvent)
		componentHidden (ComponentEvent)
		componentResized(ComponentEvent)
		componentShown(ComponentEvent)
Window	WindowListener	windowClosing(WindowEvent)
		windowOpened(WindowEvent)
		windowIconified(WindowEvent)
		windowDeiconified(WindowEvent)
		windowClosed(WindowEvent)
		windowActivated(WindowEvent)
		windowDeactivated(WindowEvent)
Container	ContainerListener	componentAdded(ContainerEvent)
		componentRemoved(ContainerEvent)
Text	TextListener	textValueChanged(TextEvent)

25.3.1 采用内部类处理事件

内部类和匿名内部类能够方便访问窗口中的组件，所有本书重点介绍内部类和匿名内部类实现的事件监听器。

下面通过一个示例介绍采用内部类和匿名内部类实现的事件处理模型，如图25-8所示的示例，界面中有两个按钮和一个标签，当单击Button1或Button2时会改变标签显示的内容。

图**25-8** 事件处理模型示例

示例代码如下：

```
//MyFrame.java文件
package com.a51work6;

import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class MyFrame extends JFrame {

    // 声明标签
    JLabel label;                                                    ①

    public MyFrame(String title) {
        super(title);

        // 创建标签
        label = new JLabel("Label");
        // 添加标签到内容面板
        getContentPane().add(label, BorderLayout.NORTH);             ②

        // 创建Button1
        JButton button1 = new JButton("Button1");
        // 添加Button1到内容面板
        getContentPane().add(button1, BorderLayout.CENTER);          ③

        // 创建Button2
        JButton button2 = new JButton("Button2");
        // 添加Button2到内容面板
        getContentPane().add(button2, BorderLayout.SOUTH);           ④

        // 设置窗口大小
        setSize(350, 120);
        // 设置窗口可见
```

```
            setVisible(true);

            // 注册事件监听器，监听Button2单击事件
            button2.addActionListener(new ActionEventHandler());        ⑤

            // 注册事件监听器，监听Button1单击事件
            button1.addActionListener(new ActionListener() {            ⑥
                @Override
                public void actionPerformed(ActionEvent event) {
                    label.setText("Hello Swing!");
                }
            });
        }

        // Button2事件处理者
        class ActionEventHandler implements ActionListener {           ⑦
            @Override
            public void actionPerformed(ActionEvent e) {
                label.setText("Hello World!");
            }
        }
    }
```

上述代码第②行通过add(button1, BorderLayout.NORTH)方法将标签添加到内容面板，这个add()方法与前面介绍的有所不同，它的第二个参数是指定组件的位置，有关布局管理的内容，将在25.4节详细介绍。类似的添加还有第③行和第④行。

代码第⑤行和第⑥行都是注册事件监听器监听Button的单击事件。但是第⑤行的事件监听器是一个内部类ActionEventHandler，它的定义是在代码第⑦行。代码第⑥行的事件监听器是一个匿名内部类。

提示　在事件处理模型中，内部类实现的模型，内部类会定义为成员内部类，因此不能访问其他方法中的局部变量组件，只能访问成员变量组件，所以代码第①行将标签组件声明为成员变量，否则ActionEventHandler内部类无法访问该组件。而匿名内部类既可以访问所在方法的局部变量组件，也可以访问成员变量组件。

25.3.2　采用**Lambda**表达式处理事件

如果一个事件监听器接口只有一个抽象方法，则可以使用Lambda表达式实现事件处理，这些接口主要有：ActionListener、AdjustmentListener、ItemListener、MouseWheelListener、TextListener和WindowStateListener等。

下面将25.3.2节的示例修改一下：

```
    //MyFrame.java文件
    package com.a51work6;

    import java.awt.BorderLayout;
    import java.awt.event.ActionEvent;
    import java.awt.event.ActionListener;

    import javax.swing.JButton;
    import javax.swing.JFrame;
    import javax.swing.JLabel;

    public class MyFrame extends JFrame implements ActionListener {        ①

        // 声明标签
        JLabel label;

        public MyFrame(String title) {
```

```
        super(title);

        // 创建标签
        label = new JLabel("Label");
        // 添加标签到内容面板
        getContentPane().add(label, BorderLayout.NORTH);

        // 创建Button1
        JButton button1 = new JButton("Button1");
        // 添加Button1到内容面板
        getContentPane().add(button1, BorderLayout.CENTER);

        // 创建Button2
        JButton button2 = new JButton("Button2");
        // 添加Button2到内容面板
        getContentPane().add(button2, BorderLayout.SOUTH);

        // 设置窗口大小
        setSize(350, 120);
        // 设置窗口可见
        setVisible(true);

        // 注册事件监听器，监听Button2单击事件
        button2.addActionListener(this);                ②

        // 注册事件监听器，监听Button1单击事件
        button1.addActionListener((event) -> {          ③
            label.setText("Hello World!");
        });
    }

    @Override
    public void actionPerformed(ActionEvent event) {    ④
        label.setText("Hello Swing!");
    }
}
```

上述代码第③行采用Lambda表达式实现的事件监听器，可见代码非常简单。另外，当前窗口本身也可以是事件处理者，代码第①行声明窗口实现ActionListener接口，代码第④行是实现抽象方法，那么注册事件监听器参数就是this了，见代码第②行。

25.3.3 使用适配器

事件监听器都是接口，在Java中接口中定义的抽象方法必须全部是实现，哪怕你对某些方法并不关心，你也要给一对空的大括号表示实现。例如WindowListener是窗口事件（WindowEvent）监听器接口，为了在窗口中接收到窗口事件，需要在窗口中注册WindowListener事件监听器，示例代码如下：

```
this.addWindowListener(new WindowListener() {

    @Override
    public void windowActivated(WindowEvent e) {
    }

    @Override
    public void windowClosed(WindowEvent e) {
    }

    @Override
    public void windowClosing(WindowEvent e) {          ①
        // 退出系统
        System.exit(0);
    }
```

```
        @Override
        public void windowDeactivated(WindowEvent e) {
        }

        @Override
        public void windowDeiconified(WindowEvent e) {
        }

        @Override
        public void windowIconified(WindowEvent e) {
        }

        @Override
        public void windowOpened(WindowEvent e) {
        }
    });
```

实现WindowListener接口需要提供它的7个方法的实现，很多情况下只是想在关闭窗口是释放一下资源，只需要实现代码第①行的windowClosing(WindowEvent e)，其他的方法并不关心，但是也必须给出空的实现。这样的代码看起来很臃肿，为此Java还提供了一些与监听器相配套的适配器。监听器是接口，命名采用XXXListener，而适配器是类，命名采用XXX Adapter。在使用时通过继承事件所对应的适配器类，覆盖所需要的方法，无关方法不用实现。

采用适配器注册接收窗口事件代码如下：

```
    this.addWindowListener(new WindowAdapter(){
        @Override
        public void windowClosing(WindowEvent e) {
            // 退出系统
            System.exit(0);
        }
    });
```

可见代码非常的简洁。事件适配器提供了一种简单的实现监听器的手段，可以缩短程序代码。但是，由于Java的单一继承机制，当需要多种监听器或此类已有父类时，就无法采用事件适配器了。

并非所有的监听器接口都有对应的适配器类，一般定义了多个方法的监听器接口，例如WindowListener有多个方法对应多种不同的窗口事件时，才需要配套的适配器，主要的适配器如下：

- ComponentAdapter：组件适配器。
- ContainerAdapter：容器适配器。
- FocusAdapter：焦点适配器。
- KeyAdapter：键盘适配器。
- MouseAdapter：鼠标适配器。
- MouseMotionAdapter：鼠标运动适配器。
- WindowAdapter：窗口适配器。

25.4 布局管理

Java为了实现图形用户界面的跨平台，并实现动态布局等效果，Java将容器内的所有组件布局交给布局管理器管理。布局管理器负责，如组件的排列顺序、大小、位置，当窗口移动或调整大小后组件如何变化等。

Java SE提供了7种布局管理器包括：FlowLayout、BorderLayout、GridLayout、BoxLayout、CardLayout、SpringLayout和GridBagLayout，其中最基础的是FlowLayout、BorderLayout和GridLayout布局管理器。下面重点介绍这三种布局。

25.4.1 FlowLayout布局

FlowLayout布局摆放组件的规律是：从上到下、从左到右进行摆放，如果容器足够宽，第一个组件先添加到容器中第一行的最左边，后续的组件依次添加到上一个组件的右边，如果当前行已摆放不下该组件，则摆放到下一行的最左边。

FlowLayout主要的构造方法如下：

- FlowLayout(int align, int hgap, int vgap)：创建一个FlowLayout对象，它具有指定的对齐方式以及指定的水平和垂直间隙，hgap参数是组件之间的水平间隙，vgap参数是组件之间的垂直间隙，单位是像素。
- FlowLayout(int align)：创建一个FlowLayout对象，指定的对齐方式，默认的水平和垂直间隙是5个单位。
- FlowLayout()：创建一个FlowLayout对象，它是居中对齐的，默认的水平和垂直间隙是5个单位。

上述参数align是对齐方式，它是通过FlowLayout的常量指定的，这些常量说明如下：

- FlowLayout.CENTER：指示每一行组件都应该是居中的。
- FlowLayout.LEADING：指示每一行组件都应该与容器方向的开始边对齐，例如，对于从左到右的方向，则与左边对齐。
- FlowLayout.LEFT：指示每一行组件都应该是左对齐的。
- FlowLayout.RIGHT：指示每一行组件都应该是右对齐的。
- FlowLayout.TRAILING：指示每行组件都应该与容器方向的结束边对齐，例如，对于从左到右的方向，则与右边对齐。

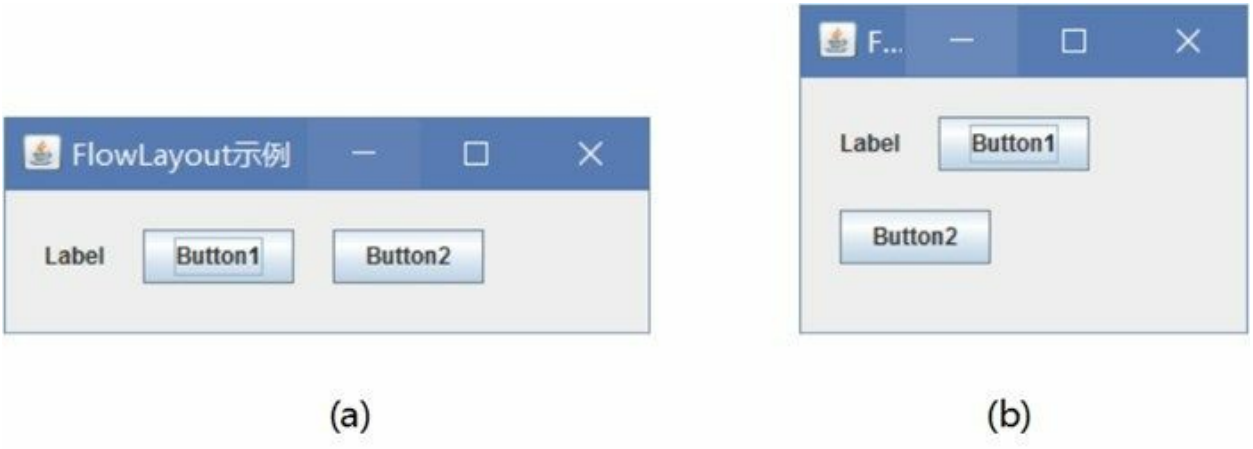
示例代码如下：

```
//MyFrame.java文件
package com.a51work6;

import java.awt.FlowLayout;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class MyFrame extends JFrame {

    // 声明标签
```

```
        JLabel label;

        public MyFrame(String title) {
            super(title);

            setLayout(new FlowLayout(FlowLayout.LEFT, 20, 20));     ①
            // 创建标签
            label = new JLabel("Label");
            // 添加标签到内容面板
            getContentPane().add(label);                            ②

            // 创建Button1
            JButton button1 = new JButton("Button1");
            // 添加Button1到内容面板
            getContentPane().add(button1);                          ③

            // 创建Button2
            JButton button2 = new JButton("Button2");
            // 添加Button2到内容面板
            getContentPane().add(button2);                          ④

            // 设置窗口大小
            setSize(350, 120);
            // 设置窗口可见
            setVisible(true);

            // 注册事件监听器，监听Button2单击事件
            button2.addActionListener((event) -> {
                label.setText("Hello Swing!");
            });

            // 注册事件监听器，监听Button1单击事件
            button1.addActionListener((event) -> {
                label.setText("Hello World!");
            });
        }
    }
```

上述代码第①行是设置当前窗口的布局是FlowLayout布局，采用FlowLayout(int align, int hgap, int vgap)构造方法。一旦设置了FlowLayout布局，就可以通过add(Component comp)方法添加组件到窗口的内容面板，见代码第②行、第③行和第④行。

运行结果如图25-9(a)所示。采用FlowLayout布局如果水平空间比较小，组件会垂直摆放，拖曳窗口的边缘使窗口变窄，如图25-9(b)所示，最后一个组件换行了。



(a) (b)

图**25-9　FlowLayout**示例运行结果

25.4.2　BorderLayout布局

BorderLayout布局是窗口的默认布局管理器，前面25.3节的示例就是采用BorderLayout布局实现。

BorderLayout 是JWindow、JFrame和JDialog的默认布局管理器。BorderLayout布局管理器把容器分成5个区域：North、South、East、West和Center，如图25-10所示每个区域只能放置一个组件。

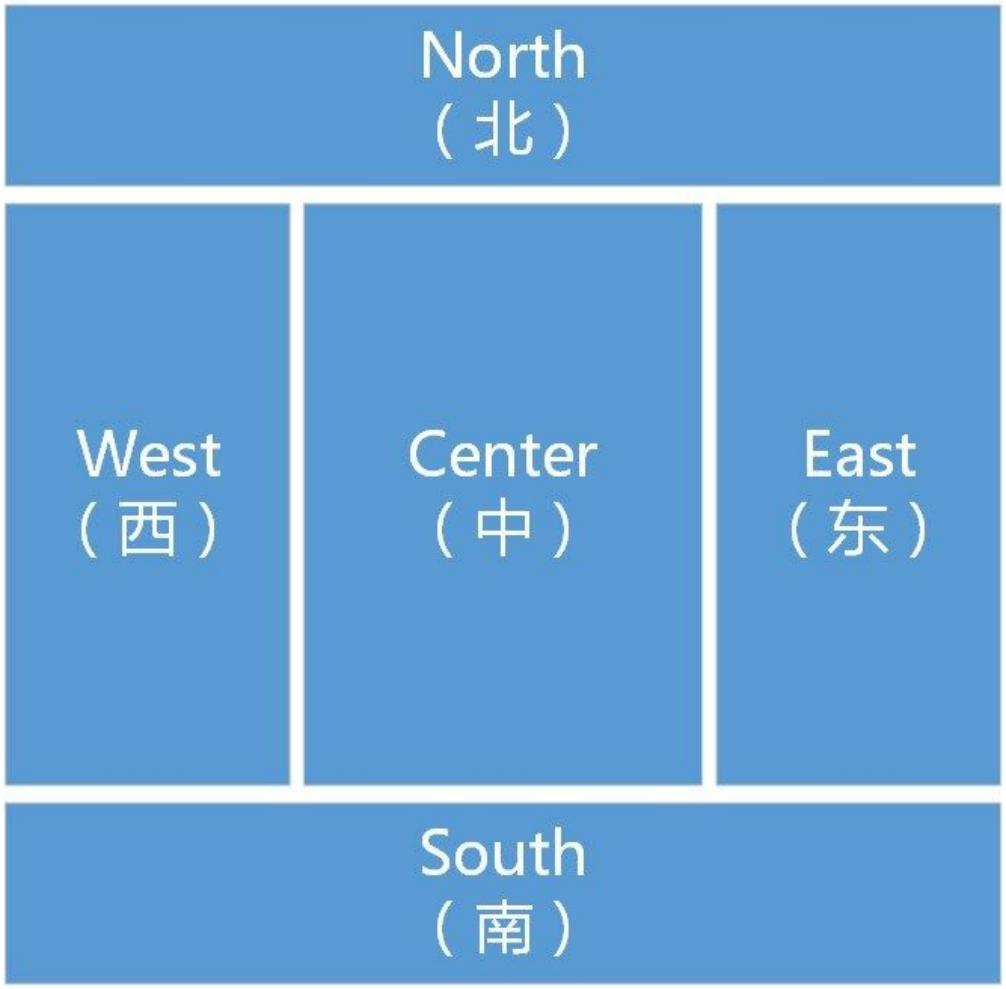


图**25-10　BorderLayout**布局

BorderLayout主要的构造方法如下：

- BorderLayout(int hgap, int vgap)：创建一个BorderLayout对象，指定水平和垂直间隙，hgap参数是组件之间的水平间隙，vgap参数是组件之间的垂直间隙，单位是像素。
- BorderLayout()：创建一个BorderLayout对象，组件之间没有间隙。

BorderLayout布局有5个区域，为此BorderLayout中定义了5个约束常量，说明如下：

- BorderLayout.CENTER：中间区域的布局约束（容器中央）。
- BorderLayout.EAST：东区域的布局约束（容器右边）。

- BorderLayout.NORTH：北区域的布局约束（容器顶部）。
- BorderLayout.SOUTH：南区域的布局约束（容器底部）。
- BorderLayout.WEST：西区域的布局约束（容器左边）。

示例代码如下：

```
//MyFrame.java文件
package com.a51work6;

import java.awt.BorderLayout;
import java.awt.Button;

import javax.swing.JFrame;

public class MyFrame extends JFrame {

    public MyFrame(String title) {
        super(title);

        // 设置BorderLayout布局
        setLayout(new BorderLayout(10, 10));                    ①

        // 添加按钮到容器的North区域
        getContentPane().add(new Button("北"), BorderLayout.NORTH);    ②
        // 添加按钮到容器的South区域
        getContentPane().add(new Button("南"), BorderLayout.SOUTH);    ③
        // 添加按钮到容器的East区域
        getContentPane().add(new Button("东"), BorderLayout.EAST);    ④
        // 添加按钮到容器的West区域
        getContentPane().add(new Button("西"), BorderLayout.WEST);    ⑤
        // 添加按钮到容器的Center区域
        getContentPane().add(new Button("中"), BorderLayout.CENTER);    ⑥

        setSize(300, 300);
        setVisible(true);
    }
}
```

上述代码第①行设置窗口布局为BorderLayout布局，组件之间间隙是10个像素，事实上窗口默认布局就是BorderLayout，只是组件之间没有间隙，如图25-11所示。代码第②行~第⑥行分别添加了5个按钮，使用的添加方法是add(Component comp, Object constraints)，第二个参数constraints是指定约束。

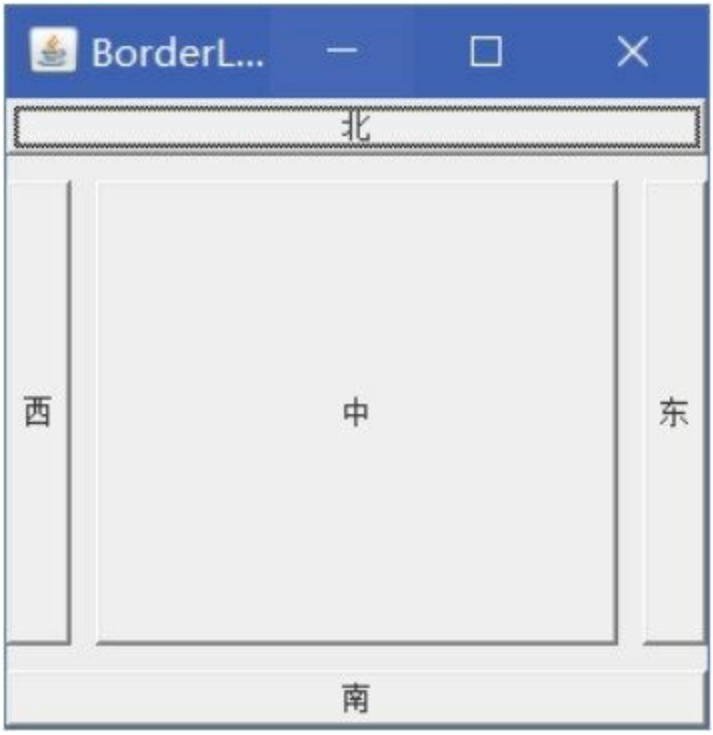


图**25-11　BorderLayout**布局示例运行结果

当使用BorderLayout时，如果容器的大小发生变化，其变化规律为：组件的相对位置不变，大小发生变化。如图25-12所示，如果容器变高或矮，则North和South不变，West、Center和East变高或矮；如果容器变宽或窄，West和East区域不变，North、Center和South变宽或窄。

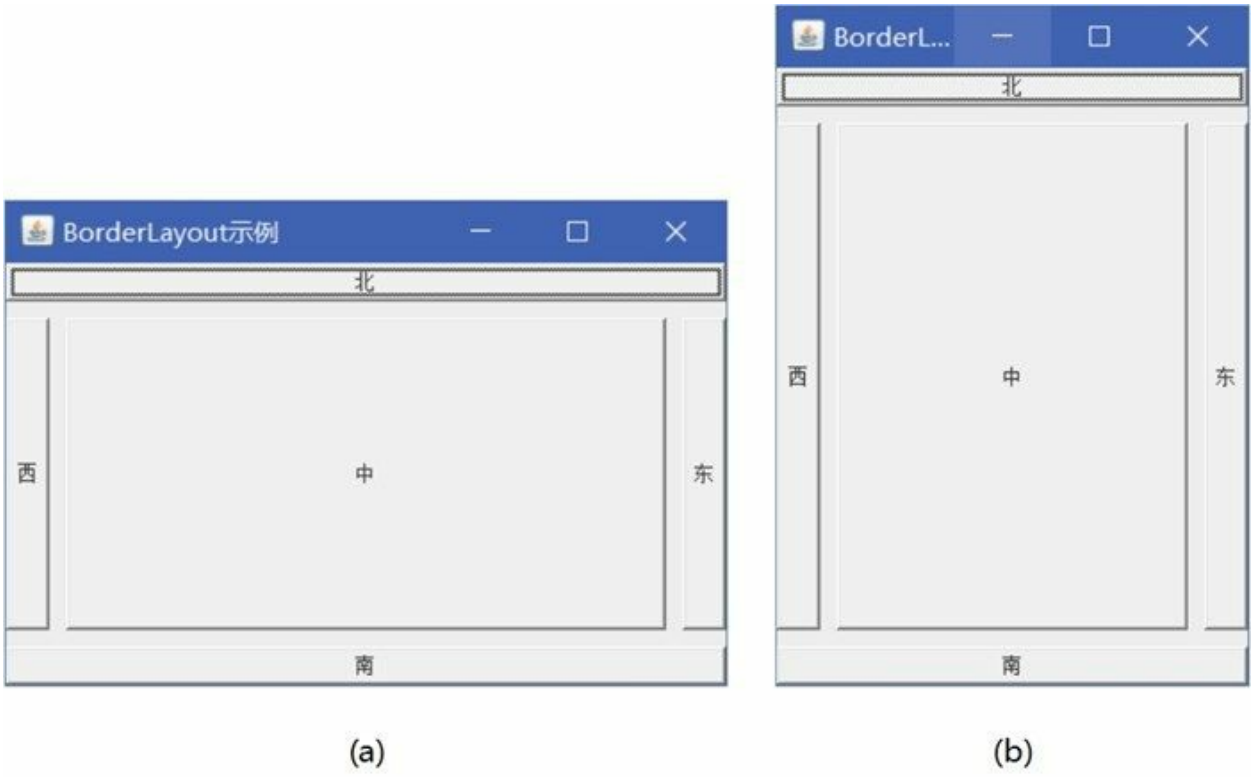


图**25-12**　**BorderLayout**布局与容器大小变化

另外，在5个区域中不一定都放置了组件，如果某个区域缺少组件，界面布局会有比较大的影响，具体影响如图25-13所示，图中列出了主要的一些情况。

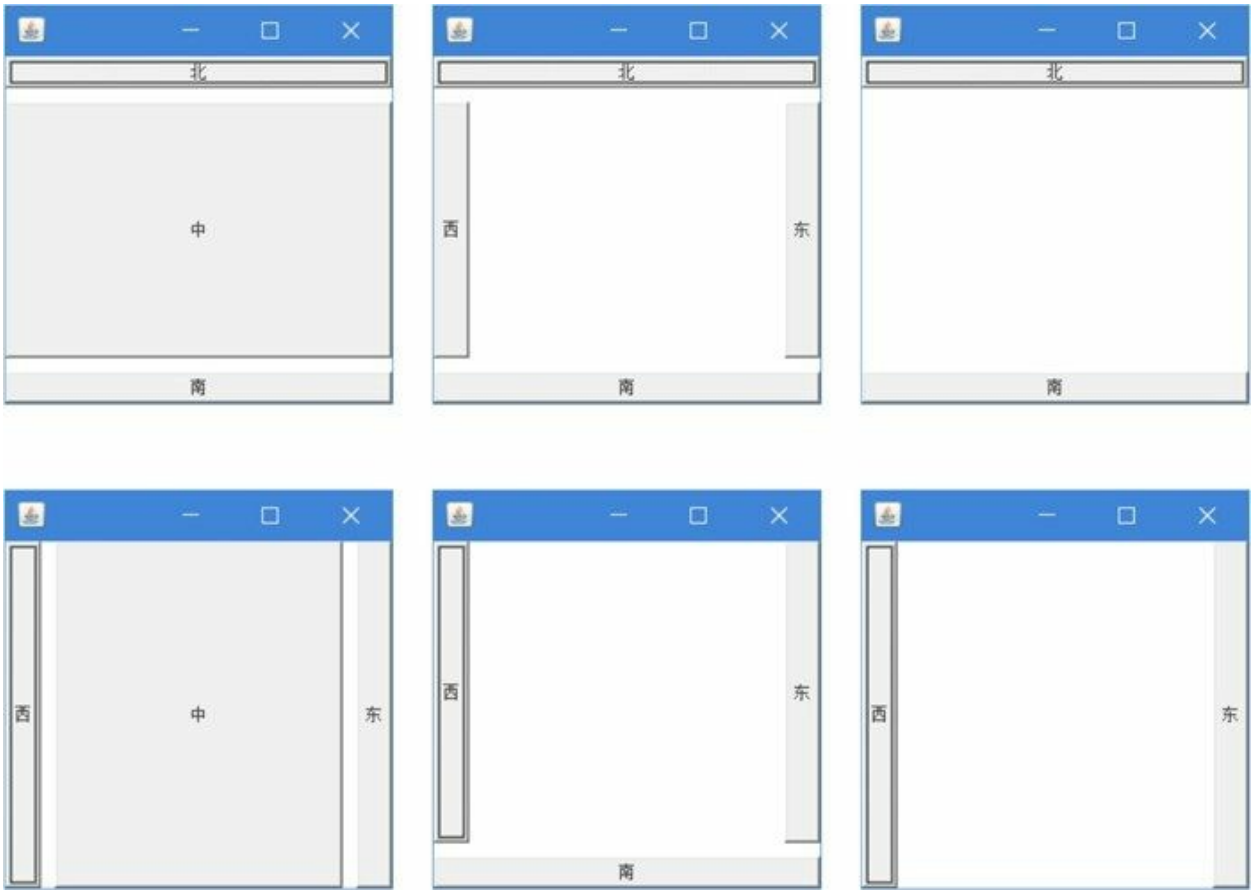


图**25-13**　某个区域缺少组件

25.4.3　GridLayout布局

GridLayout布局以网格形式对组件进行摆放，容器被分成大小相等的矩形，一个矩形中放置一个组件。

GridLayout布局主要的构造方法如下：

- GridLayout()：创建具有默认值的GridLayout对象，即每个组件占据一行一列。
- GridLayout(int rows, int cols)：创建具有指定行数和列数的GridLayout对象。
- GridLayout(int rows, int cols, int hgap, int vgap)：创建具有指定行数和列数的GridLayout对象，并指定水平和垂直间隙。

示例代码如下：

```
//MyFrame.java文件
package com.a51work6;

import java.awt.Button;
```

```
import java.awt.GridLayout;

import javax.swing.JFrame;

public class MyFrame extends JFrame {

    public MyFrame(String title) {
        super(title);

        // 设置3行3列的GridLayout布局管理器
        setLayout(new GridLayout(3, 3));              ①

        // 添加按钮到第一行的第一格
        getContentPane().add(new Button("1"));        ②
        // 添加按钮到第一行的第二格
        getContentPane().add(new Button("2"));
        // 添加按钮到第一行的第三格
        getContentPane().add(new Button("3"));
        // 添加按钮到第二行的第一格
        getContentPane().add(new Button("4"));
        // 添加按钮到第二行的第二格
        getContentPane().add(new Button("5"));
        // 添加按钮到第二行的第三格
        getContentPane().add(new Button("6"));
        // 添加按钮到第三行的第一格
        getContentPane().add(new Button("7"));
        // 添加按钮到第三行的第二格
        getContentPane().add(new Button("8"));
        // 添加按钮到第三行的第三格
        getContentPane().add(new Button("9"));        ③

        setSize(400, 400);
        setVisible(true);
    }
}
```

上述代码第①行是设置当前窗口布局采用3行3列的GridLayout布局，它有一个9个区域，分别从左到右，从上到下摆放。代码第②行~第③行添加了9个Button。运行结果如图25-14所示。

图**25-14　GridLayout**布局示例运行结果

GridLayout布局将容器分成几个区域，也会出现某个区域缺少组件情况，GridLayout布局会根据行列划分的不同，会平均占据容器的空间，实际情况比较复杂。图25-15中列出了一些主要情况。

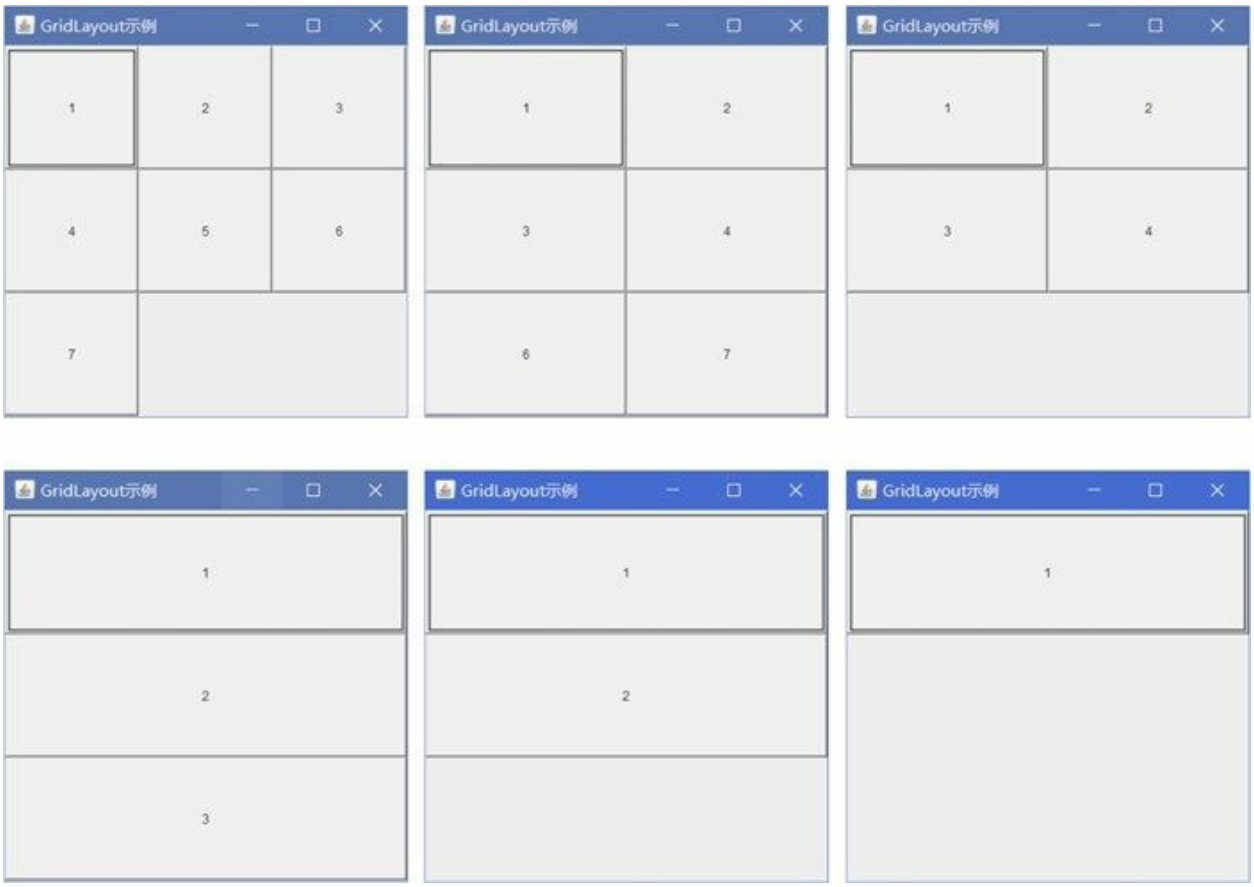


图**25-15**　某个区域缺少组件

25.4.4　不使用布局管理器

如果要开发的图形用户界面应用不考虑跨平台，不考虑动态布局，窗口大小不变的，那么布局管理器就失去使用的意义。容器也可以不设置布局管理器，那么此时的布局是由开发人员自己管理的。

组件有三个与布局有关的方法setLocation()、setSize()和setBounds()，在设置了布局管理的容器中组件的这几个方法不起作用的，不设置布局管理时它们才起作用的。

这三个方法的说明如下：

- void setLocation(int x, int y)：方法是设置组件的位置。
- void setSize(int width, int height)：方法是设置组件的大小。
- void setBounds(int x, int y, int width, int height)：方法是设置组件的大小和位置。

下面通过示例介绍一下不使用布局管理器情况，如图25-16所示的界面。

图**25-16**　不使用布局管理器示例

示例代码如下：

```
//MyFrame.java文件
package com.a51work6;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.SwingConstants;

public class MyFrame extends JFrame {

    public MyFrame(String title) {
        super(title);

        //设置窗口大小不变的
        setResizable(false);                                    ①

        // 不设置布局管理器
        getContentPane().setLayout(null);                       ②

        // 创建标签
        JLabel label = new JLabel("Label");
        // 设置标签的位置和大小
        label.setBounds(89, 13, 100, 30);                       ③
        // 设置标签文本水平居中
        label.setHorizontalAlignment(SwingConstants.CENTER);    ④
        // 添加标签到内容面板
        getContentPane().add(label);

        // 创建Button1
        JButton button1 = new JButton("Button1");
        // 设置Button1的位置和大小
        button1.setBounds(89, 59, 100, 30);                     ⑤
        // 添加Button1到内容面板
        getContentPane().add(button1);

        // 创建Button2
        JButton button2 = new JButton("Button2");
        // 设置Button2的位置
        button2.setLocation(89, 102);                           ⑥
        // 设置Button2的大小
        button2.setSize(100, 30);                               ⑦
        // 添加Button2到内容面板
        getContentPane().add(button2);

        // 设置窗口大小
```

```
            setSize(300, 200);
            // 设置窗口可见
            setVisible(true);

            // 注册事件监听器，监听Button2单击事件
            button2.addActionListener((event) -> {
                label.setText("Hello Swing!");
            });

            // 注册事件监听器，监听Button1单击事件
            button1.addActionListener((event) -> {
                label.setText("Hello World!");
            });
        }
    }
```

上述代码第①行是设置不能调整窗口大小，没有设置布局管理器后，容器中的组件都绝对布局，当容器大小如果变化，那么其中的组件大小和位置都不会变化，如图25-17所示，将窗口拉大后，组件还是在原来的位置。

图**25-17**　不使用布局管理器后调整窗口大小

代码第②行setLayout(null)方法是不设置布局管理器，参数是null。

代码第③行和第⑤行是通过调用setBounds()方法设置组件的大小和位置。也可以分别调用setLocation()和setSize()方法设置组件的大小和位置，实现与setBounds()方法相同的效果，见代码第⑥行和第⑦行。

另外，代码第④行setHorizontalAlignment(SwingConstants.CENTER)方法设置了标签的文本水平居中。

25.4.5　使用可视化设计工具

通过前面的学习，读者应该已经感受到了，通过代码实现界面布局工作量非常大。是否有可视化设计工具呢？各个主流的Java IDE工具都提供了可视化设计工具，IntelliJ IDEA和NetBeans IDE都内置了可视化设计工具，见图25-18和25-19所示。

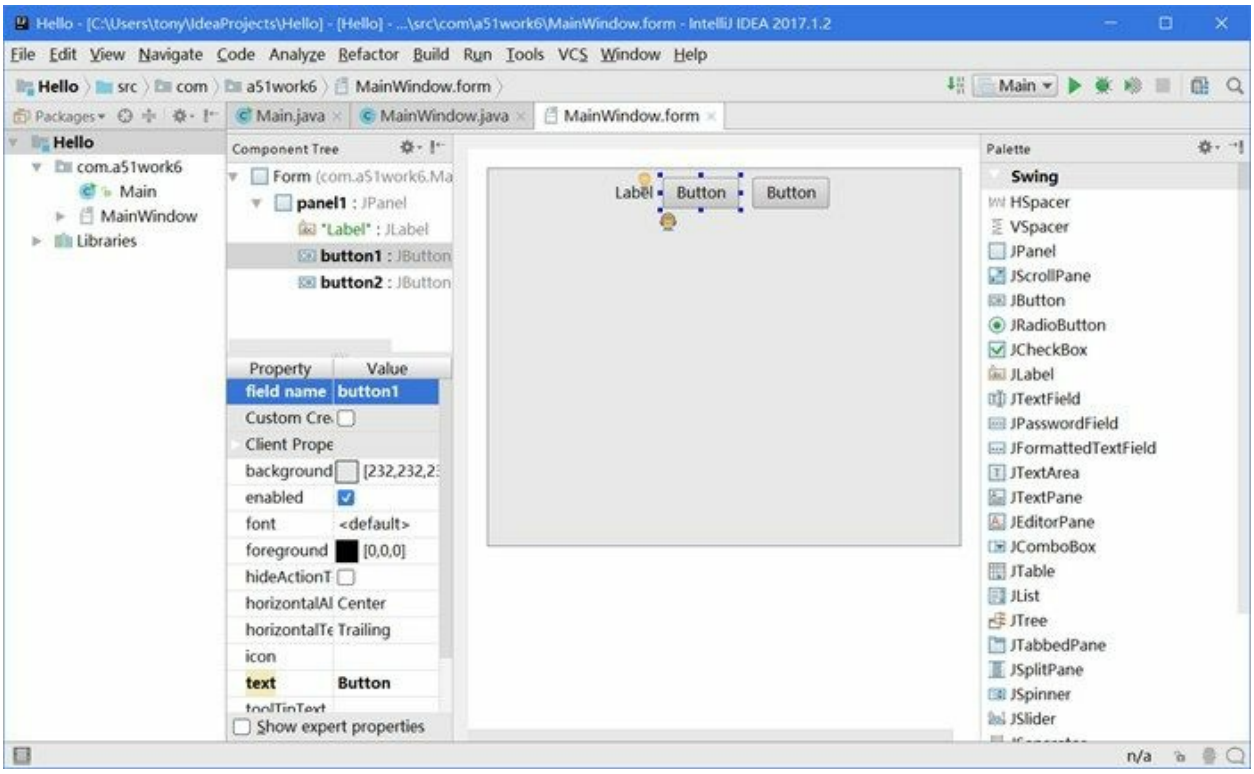


图**25-18　IntelliJ IDEA**可视化设计工具

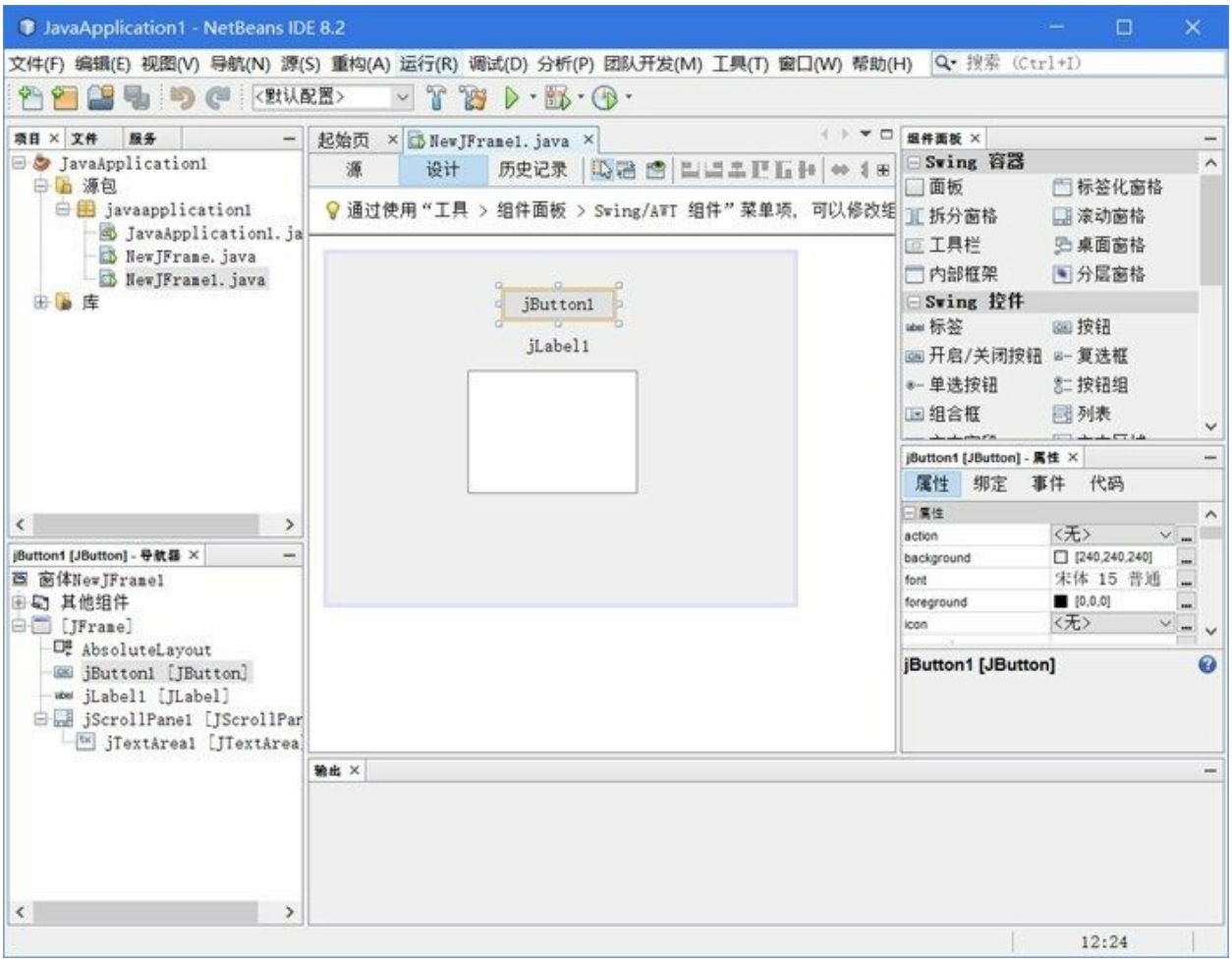


图**25-19　NetBeans IDE**可视化设计工具

Eclipse本身不提供可视化工具，但是可以安装其他可视化设计工具插件实现，目前流行的是WindowBuilder（http://www.eclipse.org/windowbuilder/），安装插件WindowBuilder网址是http://www.eclipse.org/windowbuilder/download.php，找到适合你自己的Eclipse版本的在线安装地址，读者可以参考2.2.2节在线安装插件WindowBuilder，安装过程不再赘述。WindowBuilder插件界面如图25-20所示。

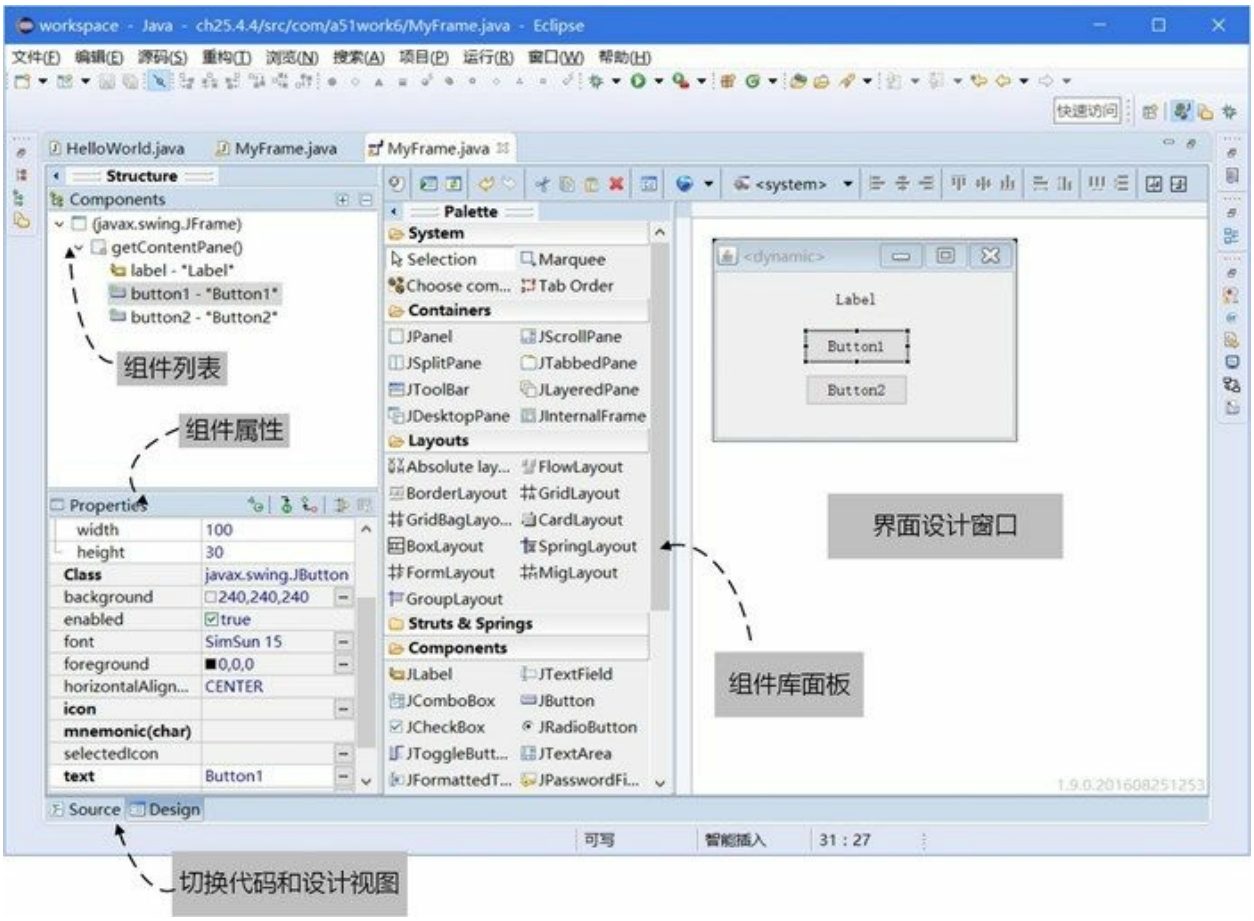


图**25-20　WindowBuilder**可视化设计工具

下面重点介绍一下WindowBuilder工具的使用。使用WindowBuilder工具可以创建一个新的窗口容器，也可以打开已经存在的窗口容器。

使用WindowBuilder工具创建新的窗口容器操作过程：选择要放置窗口源代码的包，然后选择菜单“文件”→“新建”→“其他”，打开新建类对话框，在对话框中找文件夹WindowBuilder→Swing Designer，如图25-21所示，选择其中的JFrame，然后创建窗口类。

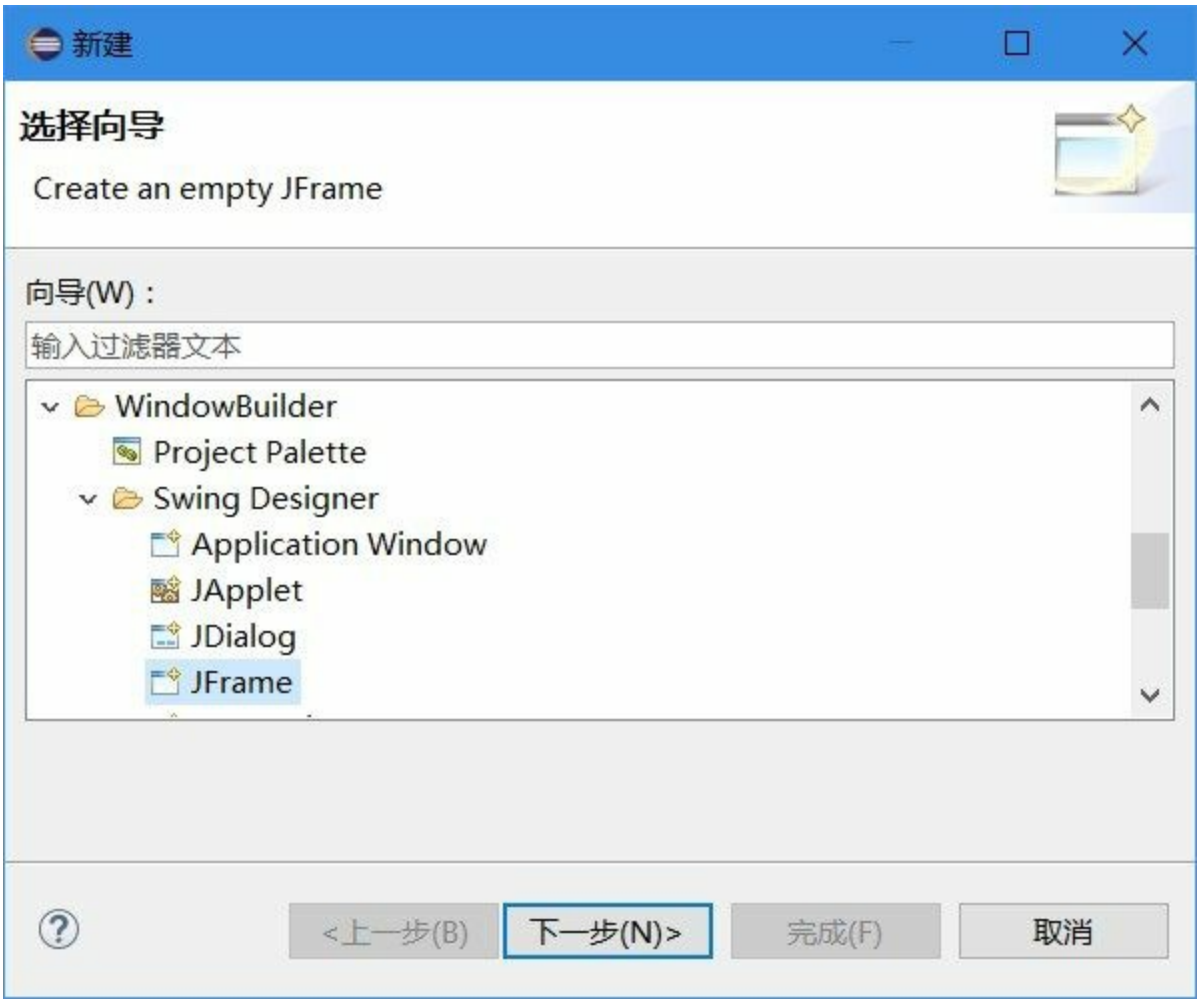


图**25-21　WindowBuilder**中创建窗口

如果你自己已经编写了一个窗口类，则可以通过WindowBuilder打开，右击窗口源代码文件，在弹出菜单中选择“打开方式”→“WindowBuilder Editor”，就可以打开了。

这些可视化设计工具都是所见即所得的，使用起来比较简单这里，不再赘述。

25.5 Swing组件

Swing所有组件都继承自JComponent，主要有文本处理、按钮、标签、列表、面板、组合框、滚动条、滚动面板、菜单、表格和树等组件。下面介绍一下常用的组件。

25.5.1 标签和按钮

标签和按钮在前面示例中已经用到了，本节再深入地介绍一下它们。

Swing中标签类是JLabel，它不仅可以显示文本还可以显示图标，JLabel的构造方法如下：

- JLabel()：创建一个无图标无标题标签对象。
- JLabel(Icon image)：创建一个具有图标的标签对象。
- JLabel(Icon image, int horizontalAlignment)：通过指定图标和水平对齐方式创建标签对象。
- JLabel(String text)：创建一个标签对象，并指定显示的文本。
- JLabel(String text, Icon icon, int horizontalAlignment)：通过指定显示的文本、图标和水平对齐方式创建标签对象。
- JLabel(String text, int horizontalAlignment)：通过指定显示的文本和水平对齐方式创建标签对象。

上述构造方法horizontalAlignment参数是水平对齐方式，它的取值是SwingConstants 中定义的以下常量之一：LEFT、CENTER、RIGHT、LEADING 或 TRAILING。

Swing中的按钮类是JButton，JButton不仅可以显示文本还可以显示图标，JButton常用的构造方法有：

- JButton()：创建不带文本或图标的按钮对象。
- JButton(Icon icon)：创建一个带图标的按钮对象。
- JButton(String text)：创建一个带文本的按钮对象。
- JButton(String text, Icon icon)：创建一个带初始文本和图标的按钮对象。

下面通过示例介绍一下标签和按钮中使用图标，示例如图25-22所示的界面，界面中上面图标是标签，下面两个图标是按钮，当单击按钮时标签可以切换图标。

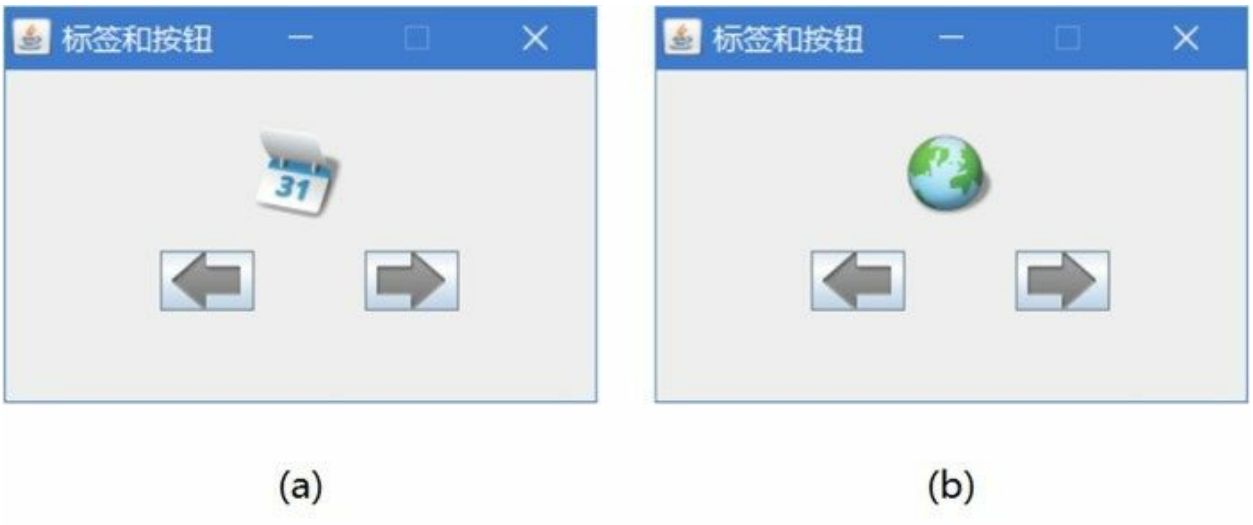


图**25-22**　标签和按钮示例

示例代码如下：

```
//MyFrame.java文件
package com.a51work6;

import javax.swing.Icon;
import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.SwingConstants;

public class MyFrame extends JFrame {
    // 用于标签切换的图标
    private static Icon images[] = { new ImageIcon("./icon/0.png"),
            new ImageIcon("./icon/1.png"),
            new ImageIcon("./icon/2.png"),
            new ImageIcon("./icon/3.png"),
            new ImageIcon("./icon/4.png"),
            new ImageIcon("./icon/5.png") };                ①

    // 当前页索引
    private static int currentPage = 0;                     ②

    public MyFrame(String title) {
        super(title);

        // 设置窗口大小不变的
        setResizable(false);

        // 不设置布局管理器
        getContentPane().setLayout(null);                   ③

        // 创建标签
        JLabel label = new JLabel(images[0]);
        // 设置标签的位置和大小
        label.setBounds(94, 27, 100, 50);
        // 设置标签文本水平居中
        label.setHorizontalAlignment(SwingConstants.CENTER);
        // 添加标签到内容面板
        getContentPane().add(label);
```

```
            // 创建向后翻页按钮
            JButton backButton = new JButton(new ImageIcon("./icon/ic_menu_back.png"));   ④
            // 设置按钮的位置和大小
            backButton.setBounds(77, 90, 47, 30);
            // 添加按钮到内容面板
            getContentPane().add(backButton);

            // 创建向前翻页按钮
            JButton forwardButton = new JButton(new ImageIcon("./icon/ic_menu_forward.png"));    ⑤
            // 设置按钮的位置和大小
            forwardButton.setBounds(179, 90, 47, 30);
            // 添加按钮到内容面板
            getContentPane().add(forwardButton);

            // 设置窗口大小
            setSize(300, 200);
            // 设置窗口可见
            setVisible(true);

            // 注册事件监听器，监听向后翻页按钮单击事件
            backButton.addActionListener((event) -> {
                if (currentPage < images.length - 1) {
                    currentPage++;
                }
                label.setIcon(images[currentPage]);
            });

            // 注册事件监听器，监听向前翻页按钮单击事件
            forwardButton.addActionListener((event) -> {
                if (currentPage > 0) {
                    currentPage--;
                }
                label.setIcon(images[currentPage]);
            });

        }
    }
```

上述代码第①行定义ImageIcon数组，用于标签切换图标，注意Icon是接口，ImageIcon是实现Icon接口。代码第②行currentPage变量记录了当前页索引，前后翻页按钮会改变前页索引。

代码第③行是不设置布局管理器。代码第④行和第⑤行是创建向后翻页按钮，构造方法参数是ImageIcon对象。

25.5.2 文本输入组件

文本输入组件主要有：文本框（JTextField）、密码框（JPasswordField）和文本区(JTextArea)。文本框和密码框都只能输入和显示单行文本。当按下Enter键时，可以触发ActionEvent事件。而文本区可以输入和显示多行多列文本。

文本框（JTextField）常用的构造方法有：

- JTextField()：创建一个空的文本框对象。
- JTextField(int columns)：指定列数，创建一个空的文本框对象，列数是文本框显示的宽度，列数主要用于FlowLayout布局。
- JTextField(String text)：创建文本框对象，并指定初始化文本。
- JTextField(String text, int columns)：创建文本框对象，并指定初始化文本和列数。

JPasswordField继承自JTextField构造方法类似，这里不再赘述。

文本区(JTextArea)常用的构造方法有：

- JTextArea()：创建一个空的文本区对象。
- JTextArea(int rows, int columns)：创建文本区对象，并指定行数和列数。
- JTextArea(String text)：创建文本区对象，并指定初始化文本。
- JTextArea(String text, int rows, int columns)：创建文本区对象，并指定初始化文本、行数和列数。

下面通过示例介绍一下文本输入组件，示例如图25-23所示的界面，界面中有三个标签（TextField:、Password:和TextArea:），一个文本框、一个密码框和一个文本区。这个布局有点复杂，可以采用布局嵌套，如图25-24所示，将TextField:标签、Password:标签、文本框和密码框都放到一个面板（panel1）中；将TextArea:和文本区放到一个面板（panel2）中。两个面板panel1和panel2放到内容视图中，内容视图采用BorderLayout布局，每个面板内部采用FlowLayout布局。

图**25-23**　文本输入组件示例

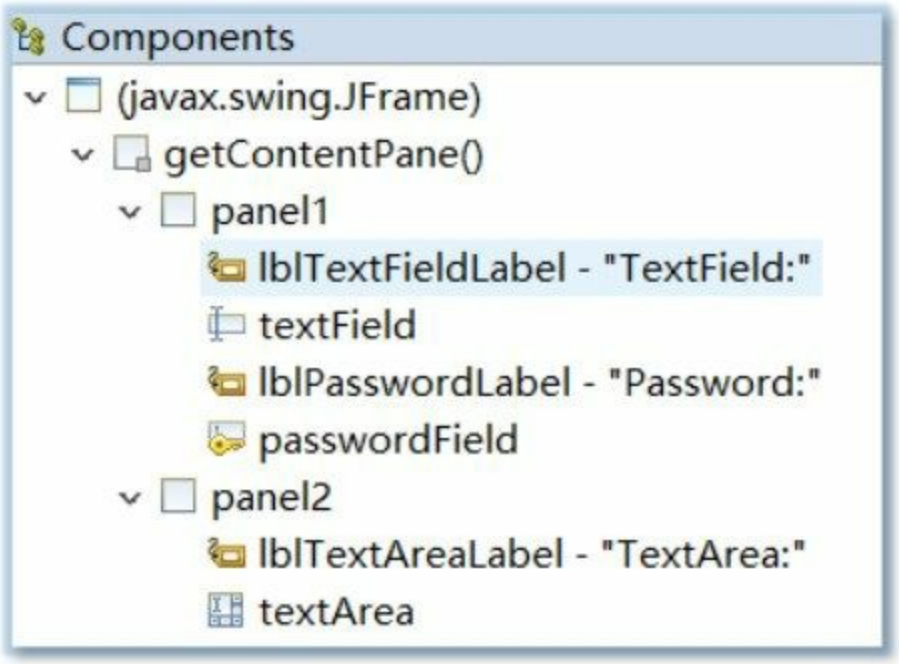


图**25-24**　布局嵌套

示例代码如下：

```
//MyFrame.java文件
package com.a51work6;

import java.awt.BorderLayout;

import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JPasswordField;
import javax.swing.JTextArea;
import javax.swing.JTextField;

public class MyFrame extends JFrame {
    private JTextField textField;
    private JPasswordField passwordField;

    public MyFrame(String title) {
        super(title);

        // 设置布局管理BorderLayout
        getContentPane().setLayout(new BorderLayout());

        // 创建一个面板panel1放置TextField和Password
        JPanel panel1 = new JPanel();                              ①
        // 将面板panel1添加到内容视图
        getContentPane().add(panel1, BorderLayout.NORTH);          ②

        // 创建标签
        JLabel lblTextFieldLabel = new JLabel("TextField:");
        // 添加标签到面板panel1
        panel1.add(lblTextFieldLabel);

        // 创建文本框
        textField = new JTextField(12);                            ③
        // 添加文本框到面板panel1
        panel1.add(textField);


        // 创建标签
        JLabel lblPasswordLabel = new JLabel("Password:");
        // 添加标签到面板panel1
        panel1.add(lblPasswordLabel);

        // 创建密码框
        passwordField = new JPasswordField(12);                    ④
        // 添加密码框到面板panel1
        panel1.add(passwordField);

        // 创建一个面板panel2放置TextArea
        JPanel panel2 = new JPanel();                              ⑤
        getContentPane().add(panel2, BorderLayout.SOUTH);          ⑥

        // 创建标签
        JLabel lblTextAreaLabel = new JLabel("TextArea:");
        // 添加标签面板panel2
        panel2.add(lblTextAreaLabel);

        // 创建文本区
        JTextArea textArea = new JTextArea(3, 20);                 ⑦
        // 添加文本区到面板panel2
        panel2.add(textArea);

        // 设置窗口大小
```

```
            pack();    // 紧凑排列，其作用相当于setSize()              ⑧

            // 设置窗口可见
            setVisible(true);

            textField.addActionListener((event)->{                      ⑨
                textArea.setText("在文本框上按下Enter键");
            });
        }
    }
```

上述代码第①行和第⑤行是创建面板容器，面板（JPanel）是一种没有标题栏和边框的容器，经常用于嵌套布局。然后再将这两个面板，添加到内容视图中，见代码第②行和代码第⑥行。

代码第③行创建文本框对象，指定列数是12。代码第④行是创建密码框，指定列数是12。它们都添加到面板panel1中。

代码第⑦行创建文本区对象，指定行数为3，列数为20，并将其添加到面板panel2中。

代码第⑧行pack()是设置窗口的大小，它设置的大小是将容器中所有组件刚好保护进去。

代码第⑨行是文本框textField注册ActionEvent事件，当用户在文本框中按下Enter键时触发。

25.5.3 复选框和单选按钮

Swing中提供了用于多选和单选功能的组件。

多选组件是复选框（JCheckBox），复选框（JCheckBox）有时也单独使用，能提供两种状态的开和关。

单选组件是单选按钮（JRadioButton），同一组的多个单选按钮应该具有互斥特性，这也是为什么单选按钮也叫做收音机按钮（RadioButton），就是当一个按钮按下时，其他按钮一定抬起。同一组多个单选按钮应该放到同一个ButtonGroup对象，ButtonGroup对象不属于容器，它会创建一个互斥作用范围。

JCheckBox主要构造方法如下：

- JCheckBox()：创建一个没有文本、没有图标并且最初未被选定的复选框对象。
- JCheckBox(Icon icon)：创建有一个图标、最初未被选定的复选框对象。
- JCheckBox(Icon icon, boolean selected)：创建一个带图标的复选框对象，并指定其最初是否处于选定状态。
- JCheckBox(String text)：创建一个带文本的、最初未被选定的复选框对象。
- JCheckBox(String text, boolean selected)：创建一个带文本的复选框对象，并指定其最初是否处于选定状态。
- JCheckBox(String text, Icon icon)：创建带有指定文本和图标的、最初未被选定的复选框对象。
- JCheckBox(String text, Icon icon, boolean selected)：创建一个带文本和图标的复选框对象，并指定其最初是否处于选定状态。

JCheckBox和JRadioButton它们有着相同的父类JToggleButton，有着相同方法和类似的构造方法，因此JRadioButton构造方法这里不再赘述。

下面通过示例介绍一下复选框和单选按钮，示例如图25-25所示的界面，界面中有一组复选框和一组单选按钮。

图**25-25**　复选框和单选按钮示例

示例代码如下：

```
//MyFrame.java文件
package com.a51work6;

...

public class MyFrame extends JFrame implements ItemListener {          ①

    //声明并创建RadioButton对象
    private JRadioButton radioButton1 = new JRadioButton("男");        ②
    private JRadioButton radioButton2 = new JRadioButton("女");        ③

    public MyFrame(String title) {
        super(title);

        // 设置布局管理BorderLayout
        getContentPane().setLayout(new BorderLayout());

        // 创建一个面板panel1放置TextField和Password
        JPanel panel1 = new JPanel();
        FlowLayout flowLayout_1 = (FlowLayout) panel1.getLayout();
        flowLayout_1.setAlignment(FlowLayout.LEFT);
        // 将面板panel1添加到内容视图
        getContentPane().add(panel1, BorderLayout.NORTH);

        // 创建标签
        JLabel lblTextFieldLabel = new JLabel("选择你喜欢的编程语言：");
        // 添加标签到面板panel1
        panel1.add(lblTextFieldLabel);

        JCheckBox checkBox1 = new JCheckBox("Java");                   ④
        panel1.add(checkBox1);

        JCheckBox checkBox2 = new JCheckBox("C++");
        panel1.add(checkBox2);

        JCheckBox checkBox3 = new JCheckBox("Objective-C");
        //注册checkBox3对ActionLEvent事件监听
        checkBox3.addActionListener((event) -> {                       ⑤
            // 打印checkBox3状态
            System.out.println(checkBox3.isSelected());
        });
        panel1.add(checkBox3);

        // 创建一个面板panel2放置TextArea
        JPanel panel2 = new JPanel();
        FlowLayout flowLayout = (FlowLayout) panel2.getLayout();
        flowLayout.setAlignment(FlowLayout.LEFT);
        getContentPane().add(panel2, BorderLayout.SOUTH);
```

```
            // 创建标签
            JLabel lblTextAreaLabel = new JLabel("选择性别：");
            // 添加标签到面板panel2
            panel2.add(lblTextAreaLabel);

            //创建ButtonGroup对象
            ButtonGroup buttonGroup = new ButtonGroup();                    ⑥
            //添加RadioButton到ButtonGroup对象
            buttonGroup.add(radioButton1);
            buttonGroup.add(radioButton2);

            //添加RadioButton到面板panel2                                    ⑦
            panel2.add(radioButton1);
            panel2.add(radioButton2);

            //注册ItemEvent事件监听器
            radioButton1.addItemListener(this);                             ⑧
            radioButton2.addItemListener(this);

            // 设置窗口大小
            pack(); // 紧凑排列，其作用相当于setSize()

            // 设置窗口可见
            setVisible(true);
        }

        //实现ItemListener接口方法
        @Override
        public void itemStateChanged(ItemEvent) {                   ⑨

            if (e.getStateChange() == ItemEvent.SELECTED) {         ⑩
               JRadioButton button = (JRadioButton) e.getItem();
               System.out.println(button.getText());
            }

        }
    }
```

上述代码第②行和第③行创建了两个单选按钮对象，为了能让这两单选按钮互斥，则需要把它们添加到一个ButtonGroup对象，见代码第⑥行创建ButtonGroup对象，并把它们添加进来。为了监听两单选按钮的选择状态，注册ItemEvent事件监听器，见代码第⑧行，为了一起处理两个单选按钮事件，它们需要使用同一个事件处理者，本例是this，这说明当前窗口是事件处理者，它实现了ItemListener接口，见代码第①行代码第⑨行实现了ItemListener接口的抽象方法。两个单选按钮使用同一个事件处理者，那么如何判断是哪一个按钮触发的事件呢？代码第⑩行是判断按钮是否被选中，如果选中通过e.getItem()方法获得按钮引用，然后再通过getText()方法获得按钮的文本标签。

代码第④行是创建了一个复选框对象，并且应该把它添加到面板panel1中。复选框和单选按钮都属于按钮，也能响应ActionEvent事件，代码第⑤行是注册checkBox3对ActionLEvent事件监听。

25.5.4 下拉列表

Swing中提供了下拉列表（JComboBox）组件，每次只能选择其中的一项。

JComboBox常用的构造方法有：

- JComboBox()：创建一个下拉列表对象。
- JComboBox(Object [] items)：创建一个下拉列表对象，items设置下拉列表中选项。下拉列表中选项内容可以是任意类，而不再局限于String。

下面通过示例介绍一下拉列表组件，示例如图25-26所示的界面，界面中有；两个下拉列表组件。

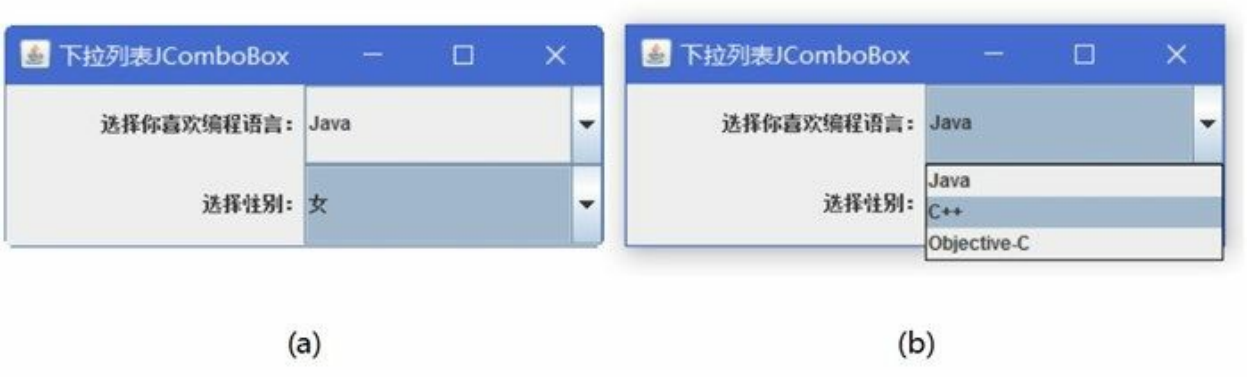


图**25-26**　下拉列表示例

示例代码如下：

```
    //MyFrame.java文件
    package com.a51work6;
    ...
    public class MyFrame extends JFrame {

        // 声明下拉列表JComboBox
        private JComboBox choice1;
        private JComboBox choice2;

        private String[] s1 = { "Java", "C++", "Objective-C" };
        private String[] s2 = { "男", "女" };

        public MyFrame(String title) {
            super(title);

            getContentPane().setLayout(new GridLayout(2, 2, 0, 0));

            // 创建标签
            JLabel lblTextFieldLabel = new JLabel("选择你喜欢的编程语言：");
            lblTextFieldLabel.setHorizontalAlignment(SwingConstants.RIGHT);
            getContentPane().add(lblTextFieldLabel);

            // 实例化JComboBox对象
            choice1 = new JComboBox(s1);                                    ①
            // 注册Action事件侦听器，采用Lambda表达式
            choice1.addActionListener(e -> {                                ②
                JComboBox cb = (JComboBox) e.getSource();                   ③
                // 获得选择的项目
                String itemString = (String) cb.getSelectedItem();          ④
                System.out.println(itemString);
            });

            getContentPane().add(choice1);

            // 创建标签
            JLabel lblTextAreaLabel = new JLabel("选择性别：");
            lblTextAreaLabel.setHorizontalAlignment(SwingConstants.RIGHT);
            getContentPane().add(lblTextAreaLabel);

            // 实例化JComboBox对象，采用Lambda表达式
            choice2 = new JComboBox(s2);                                    ⑤
            // 注册项目选择事件侦听器
            choice2.addItemListener(e -> {                                  ⑥
```

```
            // 项目选择
            if (e.getStateChange() == ItemEvent.SELECTED) {          ⑦
                // 获得选择的项目
                String itemString = (String) e.getItem();            ⑧
                System.out.println(itemString);
            }
        });
        getContentPane().add(choice2);

        // 设置窗口大小
        setSize(400, 150);

        // 设置窗口可见
        setVisible(true);
    }
}
```

上述代码第①行和第⑤行是创建下拉列表组件对象，其中构造方法参数是字符串数组。下拉列表组件在进行事件处理时，可以注册两种事件监听器：ActionListener和ItemListener，这两个监听器都只有一个抽象方法需要实现，因此可以采用Lambda表达式作为事件处理者，代码第②行和第⑥行分别注册这两个事件监听器。

代码第③行通过e事件参数获得事件源，代码第④行是获得选中的项目。代码第⑦行是判断当前的项目是否被选中，代码第⑧行是从e事件参数中取出项目对象。

25.5.5 列表

Swing中提供了列表（JList）组件，可以单选或多选。

JList常用的构造方法有：

- JList()：创建一个列表对象。
- JList(Object [] listData)：创建一个列表对象，listData设置列表中选项。列表中选项内容可以是任意类，而不再局限于String。

下面通过示例介绍列表组件，示例如图25-27所示的界面，界面中有一个列表组件。

图**25-27** 列表示例

示例代码如下：

```
//MyFrame.java文件
package com.a51work6;
...
public class MyFrame extends JFrame {
```

```
    private String[] s1 = { "Java", "C++", "Objective-C" };

    public MyFrame(String title) {
        super(title);
        // 创建标签
        JLabel lblTextFieldLabel = new JLabel("选择你喜欢的编程语言：");
        getContentPane().add(lblTextFieldLabel, BorderLayout.NORTH);

        // 列表组件JList
        JList list1 = new JList(s1);                                          ①
        list1.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);          ②
        // 注册项目选择事件侦听器，采用Lambda表达式。
        list1.addListSelectionListener(e -> {                                 ③
            if (e.getValueIsAdjusting() == false) {                           ④
                // 获得选择的内容
                String itemString = (String) list1.getSelectedValue();        ⑤
                System.out.println(itemString);
            }
        });
        getContentPane().add(list1);

        // 设置窗口大小
        setSize(300, 200);
        // 设置窗口可见
        setVisible(true);
    }

}
```

上述代码第①行创建列表组件对象，代码第②行是设置列表为单选，代码第③行是选择列表事件，代码第④行e.getValueIsAdjusting() == false可判断鼠标释放，e.getValueIsAdjusting() == true可以判断鼠标按下。

代码第⑤行是取出getSelectedValue()获得选中的项目值，如果是多选时可以通过getSelectedValues()选中的项目值。

25.5.6 分隔面板

Swing中提供了一种分隔面板（JSplitPane）组件，可以将屏幕分成左右或上下两部分。JSplitPane常用的构造方法有：

- JSplitPane(int newOrientation)：创建一个分隔面板，参数newOrientation指定布局方向，newOrientation取值是JSplitPane.HORIZONTAL_SPLIT水平或JSplitPane.VERTICAL_SPLIT垂直。

- JSplitPane(int newOrientation, Component newLeftComponent, Component newRightComponent)：创建一个分隔面板，参数newOrientation指定布局方向，newLeftComponent左侧面板组件，newRightComponent右侧面板组件。

下面通过示例介绍分隔面板组件，示例如图25-28所示的界面，界面分左右两部分，左边有列表组件，选中列表项目时右边会显示相应的图片。

图**25-28**　分隔面板示例

示例代码如下：

```
//MyFrame.java文件
package com.a51work6;
...
public class MyFrame extends JFrame {

    private String[] data = { "bird1.gif", "bird2.gif", "bird3.gif",
            "bird4.gif", "bird5.gif", "bird6.gif" };

    public MyFrame(String title) {
        super(title);

        // 右边面板
        JPanel rightPane = new JPanel();
        rightPane.setLayout(new BorderLayout(0, 0));
        JLabel lblImage = new JLabel();
        lblImage.setHorizontalAlignment(SwingConstants.CENTER);
        rightPane.add(lblImage, BorderLayout.CENTER);

        // 左边面板
        JPanel leftPane = new JPanel();
        leftPane.setLayout(new BorderLayout(0, 0));
        JLabel lblTextFieldLabel = new JLabel("选择鸟儿: ");
        leftPane.add(lblTextFieldLabel, BorderLayout.NORTH);

        // 列表组件JList
        JList list1 = new JList(data);
        list1.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
        // 注册项目选择事件侦听器，采用Lambda表达式。
        list1.addListSelectionListener(e -> {
            if (e.getValueIsAdjusting() == false) {
                // 获得选择的内容
                String itemString = (String) list1.getSelectedValue();
                String petImage = String.format("/images/%s", itemString);          ①
                Icon icon = new ImageIcon(MyFrame.class.getResource(petImage));     ②
                lblImage.setIcon(icon);
            }
        });
        leftPane.add(list1, BorderLayout.CENTER);

        // 分隔面板
        JSplitPane splitPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
                                              leftPane, rightPane);                 ③
        splitPane.setDividerLocation(100);                                          ④

        getContentPane().add(splitPane, BorderLayout.CENTER);                       ⑤
```

```
            // 设置窗口大小
            setSize(300, 200);
            // 设置窗口可见
            setVisible(true);
        }

    }
```

上述代码分别创建两个面板。然后在代码第③行创建分隔面板，设置布局是水平方向和左右面板。代码第④行splitPane.setDividerLocation(100)是设置分隔条的位置。代码第⑤行是将分隔面板添加到内容面板中。

代码第①行是获得图片的相对路径，代码第②行是创建图片ImageIcon对象，MyFrame.class.getResource(petImage)语句是获取资源图片的绝对路径。

提示　资源文件是放在字节码文件夹中文件，可通过XXX.class.getResource()方法获得它的运行时绝对路径。

25.5.7　使用表格

当有大量数据需要展示时，可以使用二维表格，有时也可以使用表格修改数据。表格是非常重要的组件。Swing提供了表格组件JTable类，但是表格组件比较复杂，它的表现形式与数据分离的，Swing的很多组件都是按照MVC[2]设计模式进行设计的，JTable最有代表性，按照MVC设计理念JTable属于视图，对应的模型是javax.swing.table.TableModel接口，根据自己的业务逻辑和数据实现TableModel接口实现类。TableModel接口要求实现所有抽象方法，使用起来比较麻烦，有时只是使用很简单的表格，此时可以使用AbstractTableModel抽象类。实际开发时需要继承AbstractTableModel抽象类。

[2]MVC是一种设计理念，将一个应用分为：模型（Model）、视图（View）和控制器（Controller），它将业务逻辑、数据、界面表示进行分离的方法组织代码，界面表示的变化不会影响到业务逻辑组件，不需要重新编写业务逻辑。

JTable类常用的构造方法有：

- JTable(TableModel dm)：通过模型创建表格，dm是模型对象，其中包含了表格要显示的数据。
- JTable(Object[][] rowData, Object[] columnNames)：通过二维数组和指定列名，创建一个表格对象，rowData是表格中的数据，columnNames是列名。
- JTable(int numRows, int numColumns)：指定行和列数创建一个空的表格对象。

如图25-29所示的一个使用JTable表格示例，该表格放置在一个窗口中，由于数据比较多，还有滚动条。下面具体介绍一下如果通过JTable实现该示例。

书籍编号	书籍名称	作者	出版社	出版日期	库存数量
0036	高等数学	李放	人民邮电出版社	20000812	1
0004	FLASH精选	刘扬	中国纺织出版社	19990312	2
0026	软件工程	牛田	经济科学出版社	20000328	4
0015	人工智能	周末	机械工业出版社	19991223	3
0037	南方周末	邓光明	南方出版社	20000923	3
0008	新概念3	余智	外语出版社	19990723	2
0019	通讯与网络	欧阳杰	机械工业出版社	20000517	1
0014	期货分析	孙宝	飞鸟出版社	19991122	3
0023	经济概论	思佳	北京大学出版社	20000819	3
0017	计算机理论基础	戴家	机械工业出版社	20000218	4
0002	汇编语言	李利光	北京大学出版社	19980318	2
0033	模拟电路	邓英才	电子工业出版社	20000527	2
0011	南方旅游	王爱国	南方出版社	19990930	2
0039	黑幕	李仪	华光出版社	20000508	24

图**25-29**　表格示例

这一节先介绍通过二维数组和列名实现表格。这种方式创建表格不需要模型，实现起来比较简单。但是表格只能接受二维数组作为数据。

具体代码如下：

```
//MyFrameTable.java文件
package com.a51work6.array;
...
public class MyFrameTable extends JFrame {

    // 获得当前屏幕的宽高
    private double screenWidth = Toolkit.getDefaultToolkit().getScreenSize().getWidth();      ①
    private double screenHeight = Toolkit.getDefaultToolkit().getScreenSize().getHeight();    ②

    private JTable table;

    public MyFrameTable(String title) {
        super(title);

        table = new JTable(rowData, columnNames);                                  ③
        // 设置表中内容字体
        table.setFont(new Font("微软雅黑", Font.PLAIN, 16));
        // 设置表列标题字体
        table.getTableHeader().setFont(new Font("微软雅黑", Font.BOLD, 16));
        // 设置表行高
        table.setRowHeight(40);
        // 设置为单行选中模式
        table.setSelectionMode(javax.swing.ListSelectionModel.SINGLE_SELECTION);
        // 返回当前行的状态模型
```

```
        ListSelectionModel rowSM = table.getSelectionModel();
        // 注册侦听器，选中行发生更改时触发
        rowSM.addListSelectionListener(new ListSelectionListener() {        ④

            public void valueChanged(ListSelectionEvent e) {
                //只处理鼠标按下
                if (e.getValueIsAdjusting() == false) {
                    return;
                }
                ListSelectionModel lsm = (ListSelectionModel) e.getSource();
                if (lsm.isSelectionEmpty()) {
                    System.out.println("没有选中行");
                } else {
                    int selectedRow = lsm.getMinSelectionIndex();
                    System.out.println("第" + selectedRow + "行被选中");
                }
            }
        });                                                                 ⑤

        JScrollPane scrollPane = new JScrollPane();                         ⑥
        scrollPane.setViewportView(table);                                  ⑦
        getContentPane().add(scrollPane, BorderLayout.CENTER);

        // 设置窗口大小
        setSize(960, 640);
        // 计算窗口位于屏幕中心的坐标
        int x = (int) (screenWidth - 960) / 2;                              ⑧
        int y = (int) (screenHeight - 640) / 2;                             ⑨
        // 设置窗口位于屏幕中心
        setLocation(x, y);

        // 设置窗口可见
        setVisible(true);
    }

    // 表格列标题
    String[] columnNames = { "书籍编号", "书籍名称", "作者", "出版社", "出版日期", "库存数量" };
    // 表格数据
    Object[][] rowData = { { "0036", "高等数学", "李放", "人民邮电出版社", "20000812", 1 },
            { "0004", "FLASH精选", "刘扬", "中国纺织出版社", "19990312", 2 },
            { "0026", "软件工程", "牛田", "经济科学出版社", "20000328", 4 },
            { "0015", "人工智能", "周未", "机械工业出版社", "19991223", 3 },
            { "0037", "南方周末", "邓光明", "南方出版社", "20000923", 3 },
            ...
            { "0032", "SOL使用手册", "贺民", "电子工业出版社", "19990425", 2 } };

}
```

上述代码第①行和第②行是获得当前机器屏幕的高和宽，通过屏幕高和宽可以计算出当前窗口屏幕的居中时的坐标，代码第⑧行和第⑨行是计算这个坐标，由于坐标原点在屏幕的左上角，所以窗口居中坐标公式：

```
x = (屏幕宽度-窗口宽度) / 2
y = (屏幕高度-窗口高度) / 2
```

代码第③行是创建JTable表格对象，采用了二维数组和字符串一维数组创建表格对象。代码第④行~第⑤行是，注册事件监听器，监听器当行选择变化时触发。由于ListSelectionListener接口虽然不是函数式接口，但只有一个方法，所以可以使用Lambda表达式实现该接口，修改代码如下：

```
// 也可换成Lambda表达式
rowSM.addListSelectionListener(e -> {
```

```
        ListSelectionModel lsm = (ListSelectionModel) e.getSource();
        if (lsm.isSelectionEmpty()) {
            System.out.println("没有选中行");
        } else {
            int selectedRow = lsm.getMinSelectionIndex();
            System.out.println("第" + selectedRow + "行被选中");
        }
    });
```

表格一般都会放到一个滚动面板（JScrollPane）中，这可以保证数据很多超出屏幕时，能够出现滚动条。把表格添加到滚动面板并不是使用add()方法，而是使用代码第⑦行的scrollPane.setViewportView(table)语句。滚动面板是非常特殊的面板，它管理这一个视口或窗口，当里面的内容超出视口会出现滚动条，setViewportView()方法可以设置一个容器或组件作为滚动面板的视口。

25.6 案例：图书库存

在实际项目开发中往往数据是从数据库中查询返回的，数据结构有多种形式，采用自定义模型可以接收任何形式的数据。本节将上一节的图书表格示例采用自定义模型重构一下。

在进行数据库设计时，数据库中每一个表对应Java一个实体类，实体类是系统的“人”、“事”、“物”等一些名词，例如图书（Book）就是一个实体类了，实体类Book代码如下：

```
//Book.java
package com.a51work6.entity;

//图书实体类
public class Book {

    // 图书编号
    private String bookid;
    // 图书名称
    private String bookname;
    // 图书作者
    private String author;
    // 出版社
    private String publisher;
    // 出版日期
    private String pubtime;
    // 库存数量
    private int inventory;

    public String getBookid() {
        return bookid;
    }
    public void setBookid(String bookid) {
        this.bookid = bookid;
    }
    public String getBookname() {
        return bookname;
    }
    public void setBookname(String bookname) {
        this.bookname = bookname;
    }
    public String getAuthor() {
        return author;
    }
    public void setAuthor(String author) {
        this.author = author;
    }
    ...
    //省略Getter和Setter方法
}
```

从代码可见实体类有很多私有属性（成员变量），为了在类外部能够访问它们，一般都会提供公有的Getter和Setter方法。

提示　主流的Java IDE编程工具都提供了，通过属性生成对应的Getter和Setter方法功能。所以一般程序员只编写属性，然后通过IDE工具生成。Eclipse中生成Getter和Setter方法是，代码源代码文件，选择菜单“源码”→“生成Getter和Setter”，弹出如图25-30所示的对话框，根据需要选中相应的属性，也可以只生成Getter或Setter方法。选择完成之后，单击确定按钮就会生成了。

图**25-30　Eclipse**中生成**Getter**和**Setter**方法

由于目前没有介绍数据库编程，本例表格中的数据是从JSON文件Books.json中读取的，Books.json位于项目的db目录中，JSON文件Books.json的内容如下：

```
[{"bookid":"0036","bookname":"高等数学","author":"李放","publisher":"人民邮电出版社","pubtime":"2000
{"bookid":"0004","bookname":"FLASH精选","author":"刘扬","publisher":"中国纺织出版社","pubtime":"199
...
{"bookid":"0005","bookname":"java基础","author":"王一","publisher":"电子工业出版社","pubtime":"19990
{"bookid":"0032","bookname":"SQL使用手册","author":"贺民","publisher":"电子工业出版社","pubtime":"19
```

从文件Books.json可见整个文档结构是JSON数组，因为JSON字符串的开始和结尾被中括号括起来，这说明是JSON数组。JSON数组的每一个元素是JSON对象，因为JSON对象是用大括号括起来的，代码如下。

```
{"bookid":"0032","bookname":"SOL使用手册","author":"贺民","publisher":"电子工业出版社","pubtime":"19
```

清楚这个JSON文档结构非常必要，当编程时候会根据这个文档结构，解析JSON文档代码参考HelloWorld，代码如下：

```
//HelloWorld.java文件
package com.a51work6;
...

public class HelloWorld {

    public static void main(String[] args) {
        List<Book> data = readData();
        new MyFrameTable("图书库存", data);
    }

    // 从文件中读取数据
    private static List<Book> readData() {
        // 返回的数据列表
        List<Book> list = new ArrayList<Book>();
        // 数据文件
        String dbFile = "./db/Books.json";

        try (FileInputStream fis = new FileInputStream(dbFile);
                InputStreamReader ir = new InputStreamReader(fis);
                BufferedReader in = new BufferedReader(ir)) {

            // 1.读取文件
            StringBuilder sbuilder = new StringBuilder();
            String line = in.readLine();

            while (line != null) {
                sbuilder.append(line);
                line = in.readLine();
            }

            // 2.JSON解码
            // 读取JSON字符完成
            System.out.println("读取JSON字符完成...");
            // JSON解码，解码成功返回JSON数组
            JSONArray jsonArray = new JSONArray(sbuilder.toString());
            System.out.println("JSON解码成功完成...");

            // 3.将JSON数组放到List<Book>集合中
            // 遍历集合
            for (Object item : jsonArray) {

                JSONObject row = (JSONObject) item;

                Book book = new Book();
                book.setBookid((String) row.get("bookid"));
                book.setBookname((String) row.get("bookname"));
                book.setAuthor((String) row.get("author"));
                book.setPublisher((String) row.get("publisher"));
                book.setPubtime((String) row.get("pubtime"));
                book.setInventory((Integer) row.get("inventory"));
```

```
                list.add(book);
            }

        } catch (Exception e) {
        }

        return list;
    }
}
```

上述代码处理过程，经历了三个步骤：主要的过程：

01. 读取文件：通过Java IO取得文件./db/Books.json，每次读取的字符串保存到StringBuilder 的sbuilder对象中。文件读完sbuilder中就是全部的JSON字符串。

02. JSON解码：读取JSON字符完成后，需要对其进行解码。由于JSON字符串是数组结构，因此解码时候使用JSONArray，创建JSONArray对象过程就是对字符串进行解码的过程，如果没有发生异常，说明成功解码。

03. 将JSON数组放到List<Book>集合中：本例表格使用的数据格式不是JSON数组形式，而是List<Book>，这种结构就是List集合中每一个元素都是Book类型。这个过程需要遍历JSON数组，把数据重新组装到Book对象中。

下面看看模型BookTableModel代码。

```
//BookTableModel.java文件
package com.a51work6;

import java.util.List;

import javax.swing.table.AbstractTableModel;

public class BookTableModel extends AbstractTableModel {        ①

    // 列名数组
    private String[] columnNames = { "书籍编号", "书籍名称", "作者", "出版社", "出版日期", "库存数量"

    // data保存了表格中数据，data类型是List集合
    private List<Book> data = null;

    public BookTableModel(List<Book> data) {
        this.data = data;
    }

    // 获得列数
    @Override
    public int getColumnCount() {
        return columnNames.length;
    }

    // 获得行数
    @Override
    public int getRowCount() {
        return data.size();
    }

    // 获得某行某列的数据
    @Override
    public Object getValueAt(int row, int col) {        ②
```

```
            Book book = (Book) data.get(row);
            switch (col) {
            case 0:
                return book.getBookid();
            case 1:
                return book.getBookname();
            case 2:
                return book.getAuthor();
            case 3:
                return book.getPublisher();
            case 4:
                return book.getPubtime();
            case 5:
                return new Integer(book.getInventory());
            }
            return null;
        }

        // 获得某列的名字
        @Override
        public String getColumnName(int col) {            ③
            return columnNames[col];
        }
    }
```

上述代码是自定义的模型，它继承了抽象类AbstractTableModel，见代码第①行。抽象类AbstractTableModel要求必须实现getColumnCount()、getRowCount()和getValueAt()三个抽象方法，getColumnCount()方法提供表格列数，getRowCount()方法提供表格的行数，getValueAt()方法提供了指定行和列时单元格内容。代码第③行的getColumnName()方法不是抽象类要求实现的方法，重写该方法能够给表格提供有意义的列名。

窗口代码如下：

```
    //MyFrameTable.java文件
    package com.a51work6;
    ...
    public class MyFrameTable extends JFrame {

        // 获得当前屏幕的宽高
        private double screenWidth = Toolkit.getDefaultToolkit().getScreenSize().getWidth();
        private double screenHeight = Toolkit.getDefaultToolkit().getScreenSize().getHeight();

        private JTable table;
        //图书列表
        private List<Book> data;

        public MyFrameTable(String title, List<Book> data) {
            super(title);

            this.data = data;
            TableModel model = new BookTableModel(data);

            table = new JTable(model);
            // 设置表中内容字体
            table.setFont(new Font("微软雅黑", Font.PLAIN, 16));
            // 设置表列标题字体
            table.getTableHeader().setFont(new Font("微软雅黑", Font.BOLD, 16));
            // 设置表行高
            table.setRowHeight(40);
            // 设置为单行选中模式
            table.setSelectionMode(javax.swing.ListSelectionModel.SINGLE_SELECTION);
            // 返回当前行的状态模型
            ListSelectionModel rowSM = table.getSelectionModel();
```

```
            // 注册侦听器，选中行发生更改时触发
            rowSM.addListSelectionListener(e -> {
                //只处理鼠标按下
                if (e.getValueIsAdjusting() == false) {
                    return;
                }
                ListSelectionModel lsm = (ListSelectionModel) e.getSource();
                if (lsm.isSelectionEmpty()) {
                    System.out.println("没有选中行");
                } else {
                    int selectedRow = lsm.getMinSelectionIndex();
                    System.out.println("第" + selectedRow + "行被选中");
                }
            });

            JScrollPane scrollPane = new JScrollPane();
            scrollPane.setViewportView(table);
            getContentPane().add(scrollPane, BorderLayout.CENTER);

            // 设置窗口大小
            setSize(960, 640);
            // 计算窗口位于屏幕中心的坐标
            int x = (int) (screenWidth - 960) / 2;
            int y = (int) (screenHeight - 640) / 2;
            // 设置窗口位于屏幕中心
            setLocation(x, y);

            // 设置窗口可见
            setVisible(true);
        }

    }
```

窗口代码与25.5.4节的类似，这里不再赘述。

本章小结

本章介绍了Java中图形用户界面编程技术Swing，Swing的基础是AWT，Swing的事件处理和布局管理都是依赖于AWT，但是AWT提供的组件在使用开发中很少使用，因此重点学习Swing提供的组件。

第 **26** 章　反射

反射（Reflection）是程序的自我分析能力，通过反射可以确定类有哪些方法、有哪些构造方法以及有哪些成员变量。Java语言提供了反射机制，通过反射机制能够动态读取一个类的信息；能够在运行时动态加载类，而不是在编译期。反射可以应用于框架开发，它能够从配置文件中读取配置信息动态加载类、创建对象，以及调用方法和成员变量。

提示　Java反射机制在一般的Java应用开发中很少使用，即便是Java EE阶段也很少使用。除非你为了开发一个框架或出于兴趣对反射机制感兴趣，否则可以跳过本章内容。

26.1 Java反射机制API

Java反射机制API主要是 java.lang.Class类和java.lang.reflect包。

26.1.1 java.lang.Class类

java.lang.Class类是实现反射的关键所在，Class类的一个实例表示Java的一种数据类型，包括类、接口、枚举、注解（Annotation）、数组、基本数据类型和void，void是“无类型”，主要用于方法返回值类型声明，表示不需要返回值。Class没有公有的构造方法，Class实例是由JVM在类加载时自动创建的。

在程序代码中获得Class实例可以通过如下代码实现；

```
//1.通过类型class静态变量
Class clz1 = String.class;
String str = "Hello";
//2.通过对象的getClass()方法
Class clz2 = str.getClass();
```

每一种类型包括类和接口等，都有一个class静态变量可以获得Class实例。另外，每一个对象都有getClass()方法可以获得Class实例，该方法是由Object类提供的实例方法。

Class类提供了很多方法可以获得运行时对象的相关信息，下面的程序代码展示了其中一些方法。

```
//HelloWorld.java文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        // 获得Class实例
        // 1.通过类型class静态变量
        Class clz1 = String.class;

        String str = "Hello";
        // 2.通过对象的getClass()方法
        Class clz2 = str.getClass();
        //获得int类型Class实例
        Class clz3 = int.class;                          ①
        //获得Integer类型Class实例
        Class clz4 = Integer.class;                      ②

        System.out.println("clz2类名称：" + clz2.getName());
        System.out.println("clz2是否为接口：" + clz2.isInterface());
        System.out.println("clz2是否为数组对象：" + clz2.isArray());
        System.out.println("clz2父类名称：" + clz2.getSuperclass().getName());

        System.out.println("clz2是否为基本类型：" + clz2.isPrimitive());
        System.out.println("clz3是否为基本类型：" + clz3.isPrimitive());
        System.out.println("clz4是否为基本类型：" + clz4.isPrimitive());

    }
}
```

运行结果如下：

```
clz2类名称：java.lang.String
clz2是否为接口：false
clz2是否为数组对象：false
clz2父类名称：java.lang.Object
clz2是否为基本类型：false
clz3是否为基本类型：true
clz4是否为基本类型：false
```

注意上述代码第①行和第②行的区别，int和Integer的区别在于int是基本数据类型，所以输出结果为true，Integer是类，是引用类型。可见Class可以描述int等基本数据类型运行时实例。

26.1.2 java.lang.reflect包

java.lang.reflect包提供了反射中用到类，主要的类说明如下：

- Constructor类：提供类的构造方法信息。
- Field类：提供类或接口中成员变量信息。
- Method类：提供类或接口成员方法信息。
- Array类：提供了动态创建和访问Java数组的方法。
- Modifier类：提供类和成员访问修饰符信息。

示例代码如下：

```
//HelloWorld.java文件
package com.a51work6;

import java.lang.reflect.Method;
import java.lang.reflect.Modifier;

public class HelloWorld {

    public static void main(String[] args) {

        try {
            // 动态加载xx类的运行时对象
            Class c = Class.forName("java.lang.String");      ①
            // 获取成员方法集合
            Method[] methods = c.getDeclaredMethods();        ②
            // 遍历成员方法集合
            for (Method method : methods) {                   ③
                // 打印权限修饰符，如public、protected、private
                System.out.print(Modifier.toString(method.getModifiers()));        ④
                // 打印返回值类型名称
                System.out.print(" " + method.getReturnType().getName() + " ");    ⑤
                // 打印方法名称
                System.out.println(method.getName() + "();");                      ⑥
            }
        } catch (ClassNotFoundException e) {                                       ⑦
            System.out.println("找不到指定类");
        }
    }
}
```

上述代码第①行是通过Class的静态方法forName(String)创建某个类的运行时对象，其中的参数是类全名字符串，如果在类路径中找不到这个类则抛出ClassNotFoundException异常，见代码第⑦行。

代码第②行是通过Class的实例方法getDeclaredMethods()返回某个类的成员方法对象数组。代码第③行是遍历成员方法集合，其中的元素是Method类型。

代码第④行的method.getModifiers()方法返回访问权限修饰符常量代码，是int类型，例如1代表public，这些数字代表的含义可以通过Modifier.toString(int)方法转换为字符串。代码第⑤行通过Method的getReturnType()方法获得方法返回值类型，然后再调用getName()方法返回该类型的名称。代码第⑥行method.getName()返回方法名称。

26.2 创建对象

Java反射机制提供了另外一种创建对象方法，Class类提供了一个实例方法newInstance()，通过该方法可以创建对象，使用起来比较简单，下面两条语句实现了创建字符串String对象。

```
Class clz = Class.forName("java.lang.String");
String str = (String) clz.newInstance();
```

这两条语句相当于String str = new String()语句。另外，需要注意newInstance()方法有可以会抛出InstantiationException和IllegalAccessException异常，InstantiationException不能实例化异常，IllegalAccessException是不能访问构造方法异常。

26.2.1 调用构造方法

调用方法newInstance()创建对象，这个过程中需要调用构造方法，上面的代码只是调用了String的默认构造方法。如果想要调用非默认构造方法，需要使用Constructor对象，它对应着一个构造方法，获得Constructor对象需要使用Class类的如下方法：

- Constructor[] getConstructors()：返回所有公有构造方法Constructor对象数组。
- Constructor[] getDeclaredConstructors()：返回所有构造方法Constructor对象数组。
- Constructor getConstructor(Class... parameterTypes)：根据参数列表返回一个共有Constructor对象。参数parameterTypes是Class数组，指定构造方法的参数列表。
- Constructor getDeclaredConstructor(Class... parameterTypes)：根据参数列表返回一个Constructor对象。参数parameterTypes同上。

示例代码如下：

```
//HelloWorld.java文件
package com.a51work6;

import java.lang.reflect.Constructor;

public class HelloWorld {

    public static void main(String[] args) {

        try {
            Class clz = Class.forName("java.lang.String");
            // 调用默认构造方法
            String str1 = (String) clz.newInstance();                    ①

            // 设置构造方法参数类型
            Class[] params = new Class[1];                               ②
            // 第一个参数是String
            params[0] = String.class;                                    ③

            // 获取与参数对应的构造方法
            Constructor constructor = clz.getConstructor(params);        ④
            // 为构造方法传递参数
            Object[] argObjs = new Object[1];                            ⑤
            // 第一个参数传递"Hello"
            argObjs[0] = "Hello";                                        ⑥

            // 调用非默认构造方法，构造方法第一个参数是String类型
```

```
            String str2 = (String) constructor.newInstance(argObjs);     ⑦
            System.out.println(str2);

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

上述代码第①行是通过调用Class的newInstance()方法创建String对象，这个过程中使用String的默认构造方法public String()。

代码第②行~第⑦行通过反射机制调用String的public String(String original)构造方法。代码第②行和第③行是设置构造方法参数类型，参数有可能有多个需要Class数组类型。代码第④行是构造方法Constructor对象。代码第⑤行和第⑥行是为构造方法准备参数值，参数值放到Object数组中，与第②行的参数类型是一一对应的。

代码第⑦行是通过调用Constructor对象的newInstance(Object... initargs)方法创建String对象。

26.2.2 案例：依赖注入实现

Java反射机制能够在运行时动态加载类，而不是在编译期。在一些框架开发中经常将要实例化的类名保存到配置文件中，在运行时从配置文件中读取类名字符串，然后动态创建对象，建立依赖关系[1]。采用new创建对象依赖关系是在编译期建立的，反射机制能够将依赖关系推迟到运行时建立，这种依赖关系动态注入进来称为依赖注入。

[1]依赖关系是一种非常普遍的关系，如果在A中使用了B，B变化会引起A的变化，A依赖于B。

例如：如同26-1所示有三个类，Student和Worker继承自Person，在HelloWorld类的main()方法中会创建Person子类实例，至于依赖哪一个类，Student还是Worker，可以在运行时从配置文件中读取，然后创建对象。

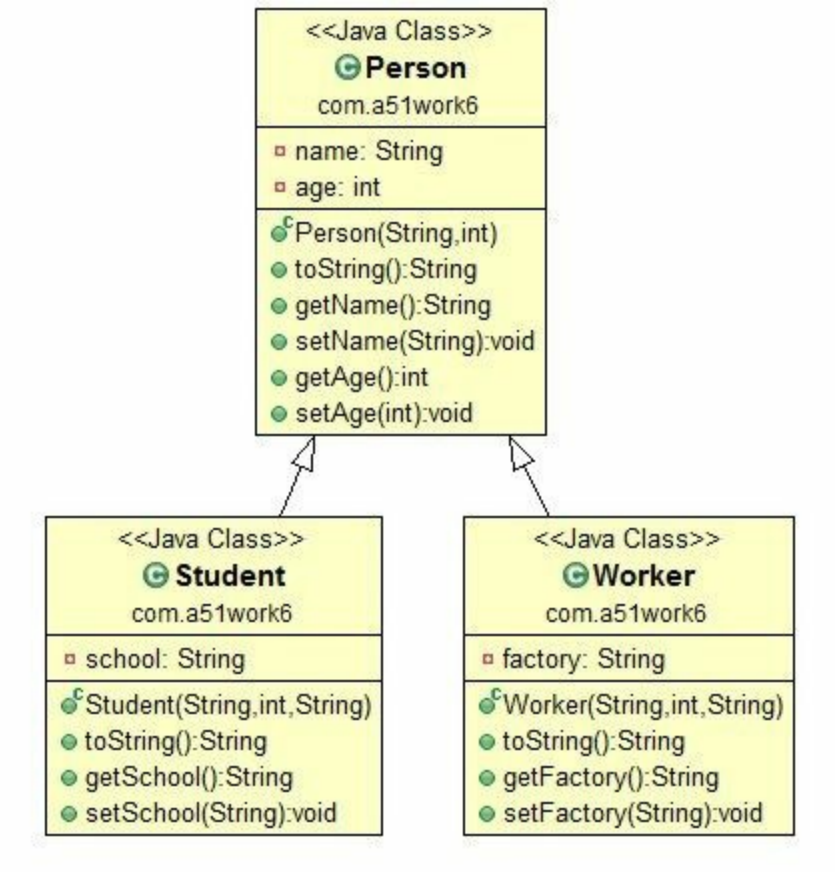


图**26-1　Person**继承类图

下面介绍一下这个案例实现过程，在Eclipse项目的根目录下创建一个文本文件Configuration.ini，文件内容如下：

```
com.a51work6.Student
```

文件中只有一行字符串，前后没有空格，它是要配置的类全名，根据自己情况修改配置信息。

HelloWorld类代码如下：

```
//HelloWorld.java文件
package com.a51work6;

… …

public class HelloWorld {

    public static void main(String[] args) {

        try {
            String className = readClassName();                ①
            Class clz = Class.forName(className);              ②
            // 指定参数类型
            Class[] params = new Class[3];
            // 第一个参数是String
            params[0] = String.class;
            // 第二个参数是int
```

```
            params[1] = int.class;
            // 第三个参数是String
            params[2] = String.class;

            // 获取对应参数的构造方法
            Constructor constructor = clz.getConstructor(params);
            // 设置传递参数
            Object[] argObjs = new Object[3];
            // 第一个参数传递"Tony"
            argObjs[0] = "Tony";
            // 第二个参数传递18
            argObjs[1] = 18;
            // 第三个参数传递"清华大学"
            argObjs[2] = "清华大学";

            // 调用非默认构造方法
            Object p = constructor.newInstance(argObjs);        ③
            System.out.println(p);

        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    // 从Configuration.ini文件中读取类名
    public static String readClassName() {

        FileInputStream readfile = null;
        InputStreamReader ir = null;
        BufferedReader in = null;
        try {
            readfile = new FileInputStream("Configuration.ini");
            ir = new InputStreamReader(readfile);
            in = new BufferedReader(ir);
            // 读取文件中的一行数据
            String str = in.readLine();
            return str;
        } catch (FileNotFoundException e) {
            System.out.println("处理FileNotFoundException...");
            e.printStackTrace();
        } catch (IOException e) {
            System.out.println("处理IOException...");
            e.printStackTrace();
        }
        return null;
    }

}
```

上述代码第①行通过调用readClassName()方法从Configuration.ini文件中读取类名，读取Configuration.ini文件内容，采用Java I/O技术，关于I/O流这里不再赘述。

代码第②行通过从配置文件Configuration.ini中读取的字符串创建Class对象。代码第③行是调用三个参数构造方法创建对象，这个对象是哪个类的实例，与你的Configuration.ini文件中配置字符串有关。

26.3 调用方法

通过反射机制还可以调用方法，这与调用构造方法类似。调用方法需要使用Method对象，它对应着一个方法，获得Method对象需要使用Class类的如下方法：

- Method[] getMethods()：返回所有公有方法Method对象数组。
- Method[] getDeclaredMethods()：返回所有方法Method对象数组。
- Method getMethod(String name, Class... parameterTypes)：通过方法名和参数类型返回公有方法Method对象。参数parameterTypes是Class数组，指定方法的参数列表。
- Method getDeclaredMethod(String name, Class... parameterTypes)：通过方法名和参数类型返回方法Method对象。参数parameterTypes同上。

现有一个Person类，它的代码如下：

```
//Person.java文件
package com.a51work6;

public class Person {

    private String name;
    private int age;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public void setNameAndAge(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString() {
        return "Person [name=" + name + ", age=" + age + "]";
    }
}
```

现在编写一个程序通过反射机制调用Person类setNameAndAge和setName方法，具体代码如下：

```
//HelloWorld.java文件
package com.a51work6;

import java.lang.reflect.Method;
```

```
public class HelloWorld {

    public static void main(String[] args) {

        try {
            Class clz = Class.forName("com.a51work6.Person");
            // 调用默认构造方法
            Person person = (Person) clz.newInstance();
            System.out.println(person);

            // 指定参数类型
            Class[] params = new Class[2];
            // 第一个参数是String
            params[0] = String.class;
            // 第二个参数是int
            params[1] = int.class;

            // 获取setNameAndAge方法对象
            Method method = clz.getMethod("setNameAndAge", params);        ①
            // 设置传递参数
            Object[] argObjs = new Object[2];
            // 第一个参数传递"Tony"
            argObjs[0] = "Tony";
            // 第二个参数传递18
            argObjs[1] = 18;
            //调用setNameAndAge方法
            method.invoke(person, argObjs);                                ②
            System.out.println(person);

            // 获取getName方法对象
            method = clz.getMethod("getName");                             ③
            // 调用getName方法
            Object result = method.invoke(person);                         ④
            System.out.println(result);

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

输出结果如下：

```
Person [name=null, age=0]
Person [name=Tony, age=18]
Tony
```

上述代码实现了调用Person类setNameAndAge和setName方法。代码第①行是获取setNameAndAge方法对象，params参数是指定参数类型，这个过程与构造方法类似。代码第②行method.invoke(person, argObjs)语句是调用setNameAndAge方法，person是要调用的对象，argObjs是设置要传递的参数值。

代码第③行是获取getName方法对象，该方法没有参数。代码第④行method.invoke(person)语句是调用person的getName方法，invoke方法会返回一个Object对象，它是调用目标方法的返回数据，本例中相当于调用getName方法返回的String类型数据。

26.4 调用成员变量

通过反射机制还可以调用成员变量，调用方法需要使用Field对象，它对应着一个方法，获得Field对象需要使用Class类的如下方法：

- Field[] getFields()：返回所有公有成员变量Field对象数组。
- Field[] getDeclaredFields()：返回所有成员变量Field对象数组。
- Field getField(String name)：通过指定公共成员变量名返回Field对象。
- Field getDeclaredField(String name)：通过指定成员变量名返回Field对象。

现有一个Person类，它的代码如下：

```
//Person.java文件
package com.a51work6;

public class Person {

    private String name = "";
    private int age;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public String toString() {
        return "Person [name=" + name + ", age=" + age + "]";
    }
}
```

Person类有两个私有的成员变量name和age。

提示　Java的反射机制非常强大，可在类外部调用类的私有成员变量和成员方法。这种功能看似强大，事实上却破坏了面向对象封装性。

现在编写一个程序通过反射机制调用Person类的私有成员变量name和age，具体代码如下：

```
//HelloWorld.java文件
package com.a51work6;

import java.lang.reflect.Field;

public class HelloWorld {
```

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

        try {
            Class clz = Class.forName("com.a51work6.Person");
            // 调用默认构造方法
            Person person = (Person) clz.newInstance();

            // 返回成员变量名为name的Field对象
            Field name = clz.getDeclaredField("name");           ①
            //设置成员变量accessible标志为true
            name.setAccessible(true);                            ②
            //为成员变量name赋值
            name.set(person, "Tony");                            ③

            // 返回成员变量名为age的Field对象
            Field age = clz.getDeclaredField("age");             ④
            //设置成员变量accessible标志为true
            age.setAccessible(true);                             ⑤
            //为成员变量age赋值
            age.set(person, 18);                                 ⑥

            // 获取成员变量保存的数据
            System.out.printf("[name:%s, age:%d]",
                    name.get(person), age.get(person));          ⑦

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

运行结果如下：

```
Person [name=, age=0]
Person [name=Tony, age=18]
```

上述代码第①行和第④行是通过名字返回成员变量Field对象，注意这里是有的getDeclaredField方法，而不是getField方法，因为这两个成员变量都是私有的。

代码第②行和第⑤行是设置成员变量accessible标志为true，accessible是可访问性标志，值为 true 则指示反射的对象在使用时应该取消Java语言访问检查。值为false则指示反射的对象应该实施Java语言访问检查。不仅是成员变量，方法和构造方法也可以通过setAccessible(true)设置，实现对私有方法和构造方法的访问。

代码第③行和第⑥行是调用Field的void set(Object obj, Object value)方法为成员变量赋值，其中obj要访问的目标对象，value是要赋给成员变量的数据。

代码第⑦行通过调用Field的Object get(Object obj)方法获取成员变量保存的数据，其中obj要访问的目标对象，方法返回值是成员变量的保存的数据。

本章小结

本章介绍了Java的反射机制，读者应该清楚什么时候使用反射。本章还详细介绍了通过反射机制创建对象、访问构造方法、访问方法和访问成员变量，读者需要了解这些API的使用。

第 27 章　注解（Annotation）

Java 5之后可以在源代码中嵌入一些补充信息，这种补充信息称为注解（Annotation），例如在方法覆盖中使用过的@Override注解，注解都是@符号开头的。

提示　Annotation可以翻译为“注解”或“注释”，笔者推荐翻译为“注解”，因为“注释”一词已经用于说明//、/**...*/和/*...*/等符号，这里的“注释”是英文Comment翻译。

注解并不能改变程序运行的结果，不会影响程序运行的性能。有些注解可以在编译时给用户提示或警告，有的注解可以在运行时读写字节码文件信息。

提示　使用注解对于代码实现功能没有任何的影响。程序员即便是不知道注解，也完全可以编写Java程序代码。对此不兴趣读者可以跳过本章内容。

27.1 基本注解

无论是哪一种注解，本质上都一种数据类型，是一种接口类型。到Java 8为止Java SE提供11种内置注解。其中有5是基本注解，它们来自于java.lang包。有6个是元注解[1]（Meta?Annotation），它们来自于java.lang.annotation包，自定义注解会用到元注解。

[1]元注解就是负责注解其他的注解。

基本注解包括：@Override、@Deprecated、@SuppressWarnings、@SafeVarargs和@FunctionalInterface。下面逐一介绍一下。

27.1.1 @Override

@Override只能用于方法，子类覆盖父类方法（或者实现接口的方法）时可以@Override注解。编译器会检查被@Override注解的方法，确保该方法父类中存在的方法，否则会有编译错误。

使用@Override注解示例代码如下：

```
//Person.java文件
package com.a51work6;

public class Person {

    private String name = "";
    private int age;

    ... ...

    @Override
    public String t0String() { //toString()        ①
        return "Person [name=" + name
                + ", age=" + age + "]";
    }
}
```

在代码第①行的方法toString()是覆盖Object类的方法，该方法使用@Override注解。如果toString()被错误成了t0String()，那么程序会发生编译错误。会有如下的代码提示：

```
类型为 Person 的方法t0String()必须覆盖或实现超类型方法
```

注意　当然如果该方法前面不加@Override注解，即便是方法写错误了，也不会有编译错误，但是Object父类的toString()方法并没有被覆盖。这会引起程序出现Bug（缺陷）。

27.1.2 @Deprecated

@Deprecated用来指示API已经过时了，@Deprecated可以用来注解类、接口、成员方法和成员变量。

使用@Deprecated注解示例代码如下：

```
//Person.java文件
package com.a51work6;

@Deprecated
public class Person {                                    ①
```

```
    @Deprecated
    protected String name;                                    ②

    private int age;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Deprecated
    public void setNameAndAge(String name, int age) {        ③
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString() {
        return "Person [name=" + name + ", age=" + age + "]";
    }

}
```

上述代码第①行类Person、第②行的成员变量name和第③行的setNameAndAge方法都被@Deprecated注解。在Eclipse中这些被注解的API都画上删除线。调用这些API代码也会有删除线，示例代码如下。

```
//HelloWorld.java文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        Person p = new Person();
        p.setNameAndAge("Tony", 20);
        p.name = "Tom";
    }
}
```

不仅代码中有删除线，而且还有编译警告。

27.1.3 @SuppressWarnings

@SuppressWarnings注解用来抑制编译器警告，如果你确认程序中的警告没有问题，可以不用理会。但是就是不想看到这些警告，可以使用@SuppressWarnings注解消除这些警告。

使用@SuppressWarnings注解示例代码如下：

```
//HelloWorld.java文件
package com.a51work6;
```

```
import java.util.ArrayList;
import java.util.List;

public class HelloWorld {

    @SuppressWarnings({ "deprecation" })            ①
    public static void main(String[] args) {

        Person p = new Person();                    ②
        p.setNameAndAge("Tony", 20);                ③
        p.name = "Tom";                             ④

    }
}
```

上述代码第①行使用@SuppressWarnings({ "deprecation" })注解了main方法，这是由于代码第②行~第④行是编译警告，因为这些API已经过时，见上一节Person代码。@SuppressWarnings注解中的deprecation表示要抑制API已经过时。使用了@SuppressWarnings注解后会发现程序代码的警告没有了。

27.1.4 @SafeVarargs

在介绍@SafeVarargs注解用法之前，先来看看如下代码：

```
//HelloWorld.java文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        // 传递可变参数，参数是泛型集合
        display(10, 20, 30);                        ①
        // 传递可变参数，参数是非泛型集合
        display("10", 20, 30);// 会有编译警告        ②

    }

    public static <T> void display(T... array) {    ③
        for (T arg : array) {
            System.out.println(arg.getClass().getName() + ":" + arg);
        }
    }
}
```

代码第③行声明了一种可变参数方法display，display方法参数个数可以变化，它可以接受不确定数量的相同类型的参数。可以通过在参数类型名后面加入...的方式来表示这是可变参数。可变参数方法中的参数类型相同，为此声明参数是需要指定泛型。

但是调用可变参数方法时，应该提供相同类型的参数，代码第①行调用时没有警告，而代码第②行调用时则会发生警告，这个警告是unchecked“未检查不安全代码”，就是由于将非泛型变量赋值给泛型变量所发生的。

那么如何抑制编译器警告可用使用@SafeVarargs注解，修改代码如下：

```
//HelloWorld.java文件
package com.a51work6;

public class HelloWorld {
```

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

            // 传递可变参数，参数是泛型集合
            display(10, 20, 30);
            // 传递可变参数，参数是非泛型集合
            display("10", 20, 30);// 没有@SafeVarargs会有编译警告

        }

        @SafeVarargs
        public static <T> void display(T... array) {
            for (T arg : array) {
                System.out.println(arg.getClass().getName() + ":" + arg);
            }
        }
    }
```

在可变参数display前添加@SafeVarargs注解。当然可用也可以使用@SuppressWarnings("unchecked")注解，但@SafeVarargs注解更适合。

27.1.5 @FunctionalInterface

@FunctionalInterface注解是Java 8增加的，用于接口的注解，声明接口是函数式接口，在前面讲解Lambda表达式时介绍过函数式接口，有关@FunctionalInterface注解不再赘述。

27.2 元注解

元注解包括：@Documented、@Target、@Retention、@Inherited、@Repeatable和@Native。元注解是为其他注解进行说明的注解，当自定义一个新的注解类型时，其中可以使用元注解。这一节先介绍一下这几种元注解含义，下一节在自定义注解中详细介绍它们的使用的。

01. @Documented

    如果在一个自定义注解中引用@Documented注解，那么该注解可以修饰代码元素（类、接口、成员变量和成员方法等），javadoc等工具可以提取这些注解信息。

02. @Target

    @Target注解用来指定一个新注解的适用目标。@Target注解有一个成员（value）用来设置适用目标，value是java.lang.annotation.ElementType枚举类型的数组，ElementType描述Java程序元素类型，它有10个枚举常量，如表27-1所示。

    表 **27-1　ElementType**枚举类型中的枚举常量

枚举常量	说明
ANNOTATION_TYPE	其他注解类型声明
CONSTRUCTOR	构造方法声明
FIELD	成员变量或常量声明
LOCAL_VARIABLE	局部变量声明
METHOD	方法声明
PACKAGE	包声明
PARAMETER	参数声明
TYPE	类、接口声明
TYPE_PARAMETER	用于泛型中类型参数声明，Java 8 推出
TYPE_USE	用于任何类型的声明，Java 8 推出

03. @Retention

    @Retention注解用来指定一个新注解的有效范围，@Retention注解有一个成员（value）用来设置保留策略，value是java.lang.annotation.RetentionPolicy枚举类型，RetentionPolicy描述注解保留策略，它有3个枚举常量，如表27-2所示。

    表 **27-2　RetentionPolicy**枚举类型中的枚举常量

枚举常量	说明
SOURCE	只适用于 Java 源代码文件中，此范围最小
CLASS	编译器把注解信息记录在字节码文件中，此范围居中
RUNTIME	编译器把注解信息记录在字节码文件中，并在运行时可以读取这些信息，此范围最大

04. @Inherited

    @Inherited注解用来指定一个新注解可以被继承。假定一个类A被该新注解修饰，那么这个A类的子类会继承该新注解。

05. @Repeatable

    @Repeatable注解是Java 8新增加的，它允许在相同的程序元素中重复注释，可重复的注释必须使用@Repeatable进行注释。

06. @Native

    @Native注解一个成员变量，指示这个变量可以被本地代码引用。常常被代码生成工具使用。

27.3 自定义注解

如果前面的Java SE提供的11内置注解不能满足你的需求，可以自定义注解，注解本质是一种接口，它是java.lang.annotation.Annotation接口的子接口，是引用数据类型。

27.3.1 声明注解

声明自定义注解可以使用@interface关键字实现，最简单形式的注解示例代码如下：

```
// Marker.java文件
package com.a51work6;

public @interface Marker{

}
```

上述代码声明一个Marker注解，@interface声明一个注解类型，它前面的访问限定修饰符与类一样有两种：公有访问权限和默认访问权限。

注意　关于注解源程序文件与类一样，一个源程序文件中可以声明多个注解，但只能有一个是公有访问权限的，源程序文件命名与公有访问权限的注解名一致。

Marker注解中不包含任何的成员，这种注解称为标记注解（Marked Annotation），基本注解中的@Override就属于标记注解。根据需要注解中可以包含一些成员，示例代码如下：

```
//Marker.java文件
package com.a51work6;
//单值注解
@interface MyAnnotation {
    String value();
}
```

代码中声明MyAnnotation 注解，它有一个成员value，注意value后面是有一对小括号，value前面的是数据类型。成员也可以有访问权限修饰符，但是只能是公有权限和默认权限。

注解中的成员也可以有默认值，示例代码如下：

```
//Marker.java文件
package com.a51work6;

//带有默认值注解
@interface MyAnnotation1 {

    String value() default "注解信息";

    int count() default 0;
}
```

通过关键字default指定默认值。使用这些注解示例代码如下：

```
//HelloWorld.java文件
package com.a51work6;

@Marker                                          ①
```

```
public class HelloWorld {

    @MyAnnotation(value = "Annotation")        ②
    private String info = "";

    @MyAnnotation1(count = 10)                 ③
    public static void main(String[] args) {

    }
}
```

默认情况下注解可以修饰任意的程序元素（类、接口、成员变量、成员方法和数据类型等）。代码第①行使用@Marker注解修饰类。代码第②行是@MyAnnotation(value = "Annotation")注解修饰成员变量，其中value = "Annotation"是为value成员提供数值。代码第③行是@MyAnnotation1(count = 10) 注解修饰成员方法，@MyAnnotation1有两个成员，但是只为count成员赋值，另外一个成员value使用默认值。

27.3.2 案例：使用元注解

上一节声明注解只是最基本形式的注解，对于复杂的注解可以在声明注解时使用元注解。下面通过一个案例介绍一下在自定义注解中使用元注解，在本案例中定义了两个注解。

首先看看第一个注解MyAnnotation，它用来修饰类或接口，MyAnnotation代码如下：

```
//MyAnnotation.java文件
package com.a51work6;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Documented;
import java.lang.annotation.Target;

@Documented                                    ①
@Target({ ElementType.TYPE })                  ②
@Retention(RetentionPolicy.RUNTIME)            ③
public @interface MyAnnotation {               ④
    String description();                      ⑤
}
```

上述代码第⑤行是声明注解类型MyAnnotation，其中使用了三个元注解修饰MyAnnotation注解，代码第①行使用@Documented指定MyAnnotation注解信息可以被javadoc工具读取。代码第②行使用@Target({ ElementType.TYPE })指定MyAnnotation注解用于修饰类和接口等类型。代码第③行@Retention(RetentionPolicy.RUNTIME)指定MyAnnotation注解信息可以在运行时被读取。代码第⑤行的description是MyAnnotation注解的成员。

提示　Eclipse工具不仅可以创建类和接口，还可以创建注解，在Eclipse中选择菜单“文件”→“新建”→“注释”，打开如图27-1所示对话框，可以根据自己的需要添加@Retention、@Target或@Documented注解。注意Eclipse工具汉化包将Annotation翻译为“注释”。

图**27-1　Eclipse**工具创建注解对话框

第二个注解MemberAnnotation，它用来类中成员变量和成员方法，MemberAnnotation代码如下：

```
//MemberAnnotation.java文件
package com.a51work6;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
```

```
@Documented
@Retention(RetentionPolicy.RUNTIME)                    ①
@Target({ ElementType.FIELD, ElementType.METHOD })    ②
public @interface MemberAnnotation {                   ③
    Class<?> type() default void.class;                ④
    String description();                              ⑤
}
```

上述代码第③行是声明注解类型MemberAnnotation，其中也使用了三个元注解修饰MemberAnnotation注解，代码第①行的@Retention(RetentionPolicy.RUNTIME)指定MemberAnnotation注解信息可以在运行时被读取。代码第②行@Target({ ElementType.FIELD, ElementType.METHOD })指定MemberAnnotation注解用于修饰类中成员。

代码第④行和第⑤行是声明两个成员，type类型是Class<?>，默认值是void.class，void.class是void类型表示方式。description类型是String，没有设置默认值。

提示　代码第④行中Class<?>类型，表示Class的泛型，?是泛型通配符，可以是任何类型。泛型多数情况下尖括号中指定的都某个具体类型，泛型也是为此而设计的。但是有时确实不需要知道具体类型，或者说什么类型都可以，此时可以使用?作为占位符。

使用了MyAnnotation和MemberAnnotation注解是Person类，Person类代码如下：

```
//Person.java文件
package com.a51work6;

@MyAnnotation(description = "这是一个测试类")                              ①
public class Person {

    @MemberAnnotation(type = String.class, description = "名字")          ②
    private String name;

    @MemberAnnotation(type = int.class, description = "年龄")             ③
    private int age;

    @MemberAnnotation(type = String.class, description = "获得名字")      ④
    public String getName() {
        return name;
    }

    @MemberAnnotation(type = int.class, description = "获得年龄")         ⑤
    public int getAge() {
        return age;
    }

    @MemberAnnotation(description = "设置姓名和年龄")                     ⑥
    public void setNameAndAge(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString() {
        return "Person [name=" + name + ", age=" + age + "]";
    }

}
```

使用注解时如果当前类与注解不在同一个包中，则需要将注解引入。代码第①行@MyAnnotation注解只能Person之前，修饰Person类型。代码第②行和第③行是使用@MemberAnnotation注解修饰成员变

量。代码第④行、第⑤行和第⑥行是使用@MemberAnnotation注解修饰成员方法。

27.3.3 案例：读取运行时注解信息

注解是为工具读取信息而准备的。有些工具可以读取源代码文件中的注解信息；有的可以读取字节码文件中的注解信息；有的可以在运行时读取注解信息。但是读取这些注解信息的代码都是一样的，区别只在于自定义注解中@Retention的保留策略不同。

读取注解信息需要反射相关API，Class类如下方法：

- <A extends Annotation> A getAnnotation(Class<A> annotationClass)：如果此元素存在annotationClass类型的注解，则返回注解，否则返回null。
- Annotation[] getAnnotations()：返回此元素上存在的所有注解。
- Annotation[] getDeclaredAnnotations():返回直接存在于此元素上的所有注解。与getAnnotations()区别在于该方法将不返回继承的注释。
- boolean isAnnotationPresent(Class<? extends Annotation> annotationClass)：如果此元素上存在annotationClass类型的注解，则返回true，否则返回false。
- boolean isAnnotation()：如果此Class对象表示一个注解类型则返回true。

读者运行时Person类中注解信息代码如下：

```
//HelloWorld.java文件
package com.a51work6;

import java.lang.reflect.Field;
import java.lang.reflect.Method;

public class HelloWorld {

    public static void main(String[] args) {
        try {
            Class<?> clz = Class.forName("com.a51work6.Person");            ①

            // 读取类注解
            if (clz.isAnnotationPresent(MyAnnotation.class)) {              ②
                MyAnnotation ann = (MyAnnotation) clz.getAnnotation(MyAnnotation.class);    ③
                System.out.printf("类%s，读取注解描述： %s \n",
                                  clz.getName(), ann.description());                        ④
            }

            // 读取成员方法的注解信息
            Method[] methods = clz.getDeclaredMethods();                            ⑤
            for (Method method : methods) {
                if (method.isAnnotationPresent(MemberAnnotation.class)) {           ⑥
                    MemberAnnotation ann = method.getAnnotation(MemberAnnotation.class);    ⑦
                    System.out.printf("方法%s，读取注解描述： %s \n",
                                      method.getName(), ann.description());         ⑧
                }
            }

            // 读取成员变量的注解信息
            Field[] fields = clz.getDeclaredFields();                               ⑨
            for (Field field : fields) {
                if (field.isAnnotationPresent(MemberAnnotation.class)) {            ⑩
                    MemberAnnotation ann = field.getAnnotation(MemberAnnotation.class);
                    System.out.printf("成员变量%s，读取注解描述： %s \n",
                                      field.getName(), ann.description());
```

```
                    }
                }

            } catch (Exception e) {
                e.printStackTrace();
            }
        }

    }
```

运行结果如下：

```
类com.a51work6.Person，读取注解描述： 这是一个测试类
方法getName，读取注解描述： 获得名字
方法getAge，读取注解描述： 获得年龄
方法setNameAndAge，读取注解描述： 设置姓名和年龄
成员变量name，读取注解描述： 名字
成员变量age，读取注解描述： 年龄
```

上述代码第①行是创建Person类对应的Class对象，代码第②行是判断Person类是否存在MyAnnotation注解，如果存在则通过代码第③行的getAnnotation方法将MyAnnotation注解实例返回。代码第④行中ann.description()表达式读取MyAnnotation注解中description成员内容。

代码第⑤行是获得所有成员方法对象数组，通过遍历方法对象数组，在代码第⑥行判断方法中是否存在MemberAnnotation注解，如果存在则通过代码第⑦行的getAnnotation方法将MemberAnnotation注解实例返回。代码第⑧行中ann.description()表达式读取MemberAnnotation注解中description成员内容。

代码第⑨行是获得所有成员变量对象数组，代码第⑩行是判断成员变量中是否存在MemberAnnotation注解。其他的处理与成员方法类似，这里不再赘述。

本章小结

本章介绍了Java的注解，首先介绍了基本注解，接着介绍了元注解，最后介绍了自定义注解。读者需要掌握基本注解有哪些它们的用途，了解元注解、自定义注解，了解读取自定义注解信息的方法。另外，读者不要把注解与注释混淆了。

第 **28** 章　数据库编程

数据必须以某种方式来存储才可以有用，数据库实际上是一组相关数据的集合。例如，某个医疗机构中所有信息的集合可以被称为一个“医疗机构数据库”，这个数据库中的所有数据都与医疗机构的相关。

数据库编程相关的技术很多，涉及具体的数据库安装、配置和管理，还要掌握SQL语句，最后才能编写程序访问数据库。本章重点介绍MySQL数据库的安装和配置，以及JDBC数据库编程。

28.1 数据持久技术概述

把数据保存到数据库中只是一种数据持久化方式。凡是将数据保存到存储介质中，需要的时候能够找到它们，并能够对数据进行修改，这些就属于数据持久化。

Java中数据持久化技术有很多：

01. 文本文件

    通过Java I/O流技术将数据保存到文本文件中，然后进行读写操作，这些文件一般是结构化的文档，如XML、JSON和CSV等文件。结构化文档就是文件内部采取某种方式将数据组织起来。

02. 对象序列化

    序列化用于将某个对象以及它的状态写到文件中，它保证了被写入的对象之间的关系，当需要这个对象时，可以完整地从文件重新构造出来，并保持原来的状态。在Java中实现java.io.Serilizable接口的对象才能被序列化和反序列化。Java还提供了两个流：ObjectInputStream和ObjectOutputStream。但序列化不支持事务处理、查询或者向不同的用户共享数据。序列化只适用于最简单的应用，或者在某些无法有效地支持数据库的嵌入式系统中。

03. 数据库

    将数据保存数据库中是不错的选择，数据库的后面是一个数据库管理系统，它支持事务处理、并发访问、高级查询和SQL语言。Java对象保存到数据库中主要的技术有：JDBC[1]、EJB[2]和ORM[3]框架等。JDBC是本书重点介绍的技术。

[1] Java数据库连接（Java Database Connectivity，简称JDBC）。

[2] 企业级JavaBean（Enterprise JavaBean，简称EJB）是一个用来构筑企业级应用的服务器端组件。

[3] 对象关系映射（Object-Relational mapping，简称ORM），它能将对象保存到数据库表中，对象与数据库表结构之间是有某种对应关系的。

28.2 MySQL数据库管理系统

介绍JDBC技术一定会依托某个数据库管理系统（(Database Management System，缩写DBMS），还会使用SQL语句，所以本节先介绍一下数据库管理系统。

数据库管理系统负责对数据进行管理、维护和使用。现在主流数据库管理系统有Oracle、SQL Server、DB 2、Sysbase和MySQL等，本节介绍MySQL数据库管理系统使用和管理。

MySQL（https://www.mysql.com）是流行的开放源码SQL数据库管理系统，它是由MySQL AB公司开发，先被Sun公司收购，后来又被Oracle公司收购，现在MySQL数据库是Oracle旗下的数据库产品，Oracle负责提供技术支持和维护。

28.2.1 数据库安装与配置

目前Oracle提供了多个MySQL版本，其中社区版MySQL Community Edition是免费的，社区版本比较适合中小企业数据库，本书也采用这个版本介绍。

社区版下载地址是https://dev.mysql.com/downloads/windows/installer/5.7.html，如图28-1所示，可以选择不同的平台版本，MySQL可运行在Windows、Linux和UNIX等操作系统上安装和运行。本书选择的是Windows 版中的mysql-installer-community-5.7.18.1.msi安装文件。

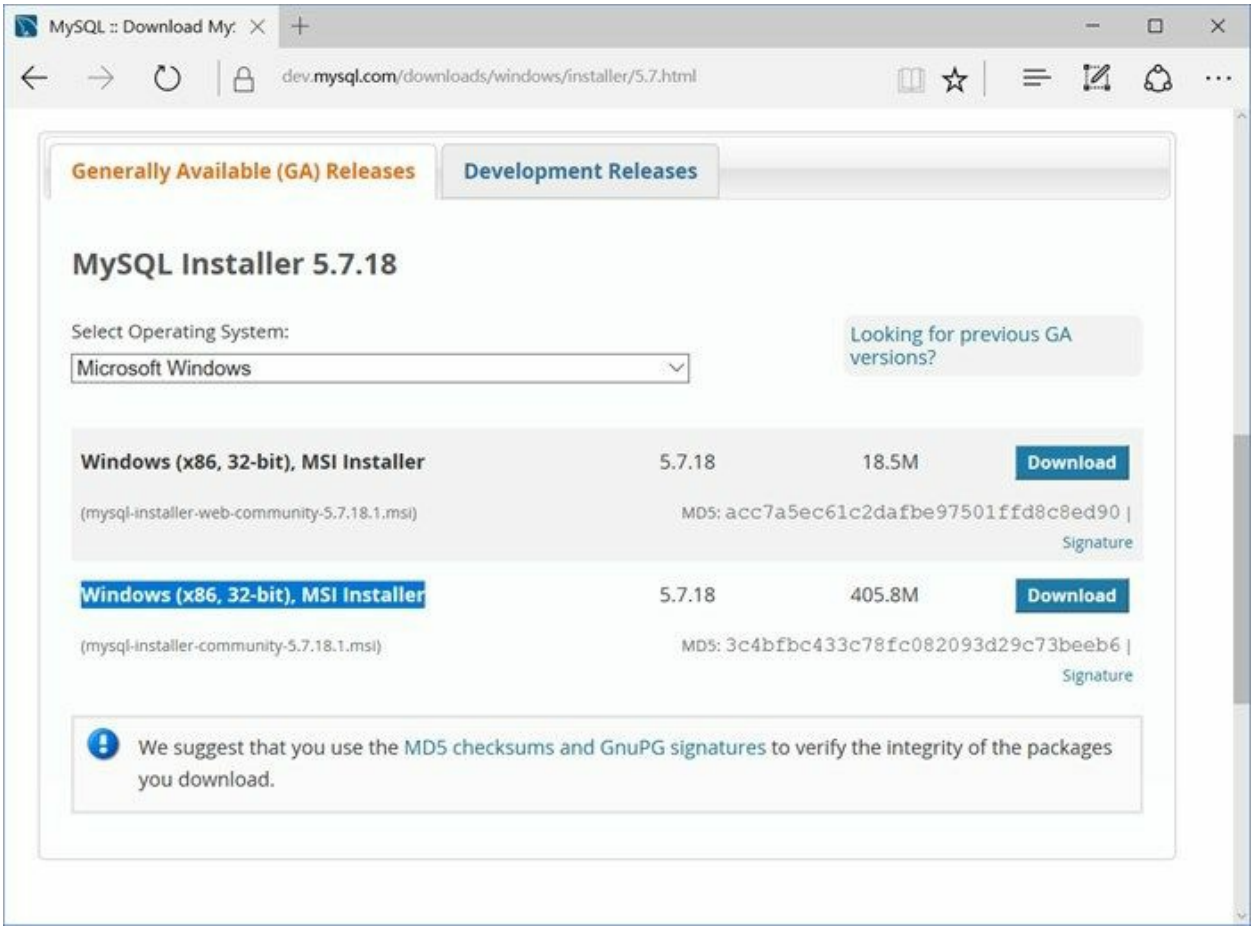


图28-1 MySQL数据库社区版下载

下载成功后，可以双击.msi文件启动安装过程，安装过程比较简单，这里介绍一个关键步骤。

01. 安装类型选择

如图28-2所示是安装类型选择对话框。在这个页面中可以选择安装类型，有5种安装类型：Developer Default（开发者安装）、Server only（只安装服务器）、Client only（只安装客户端）、Full（全部安装）和Custom（自定义安装）。对于学习和开发可以选择Developer Default安装。

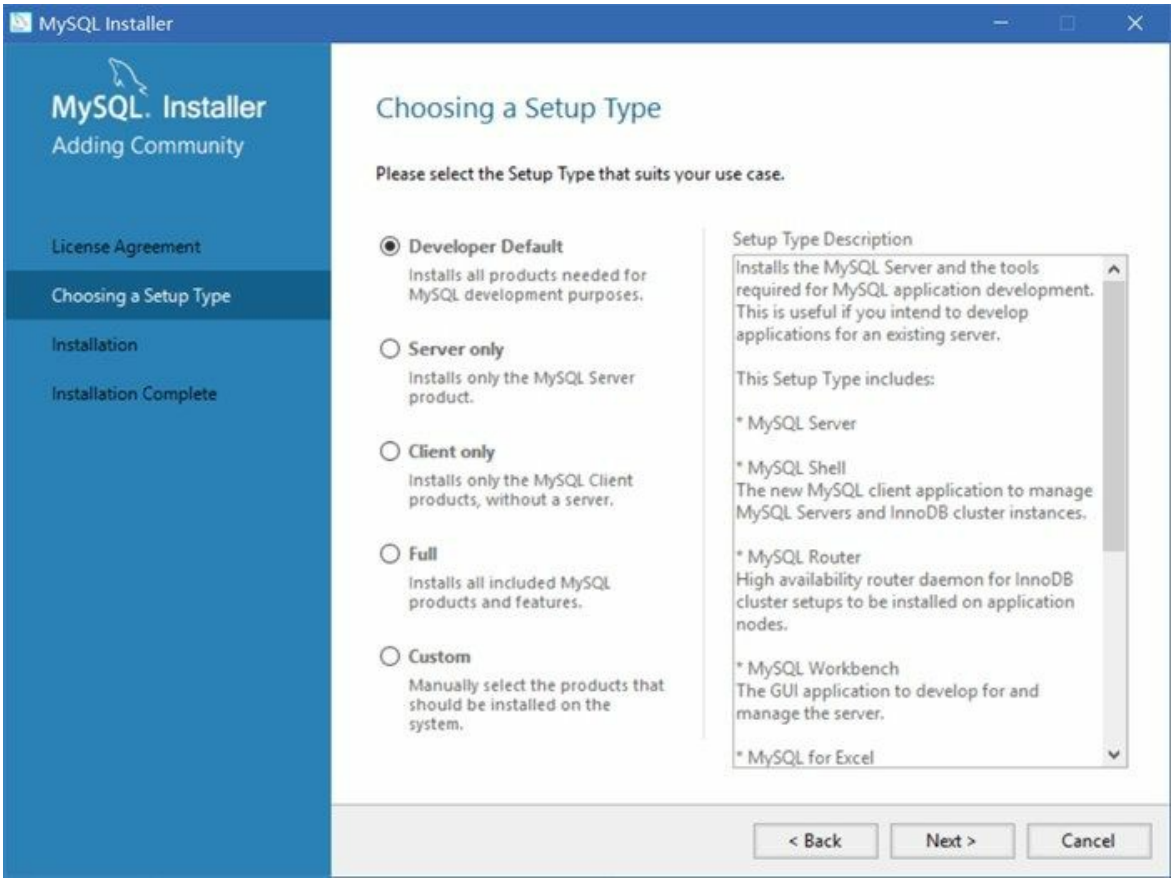


图**28-2**　安装类型选择

02. 安装环境检查

在Windows下安装时，由于Windows版本多样性。安装过程会检查你的需要，缺少哪些Windows安装包，安装过程会给出提示，如图28-3所示，安装MySQL Server需要Microsoft Visual C++ 2013 Runtime，则需要到微软网站下载Microsoft Visual C++ 2013 Runtime安装包，安装好Microsoft Visual C++ 2013 Runtime后，再重新安装MySQL。

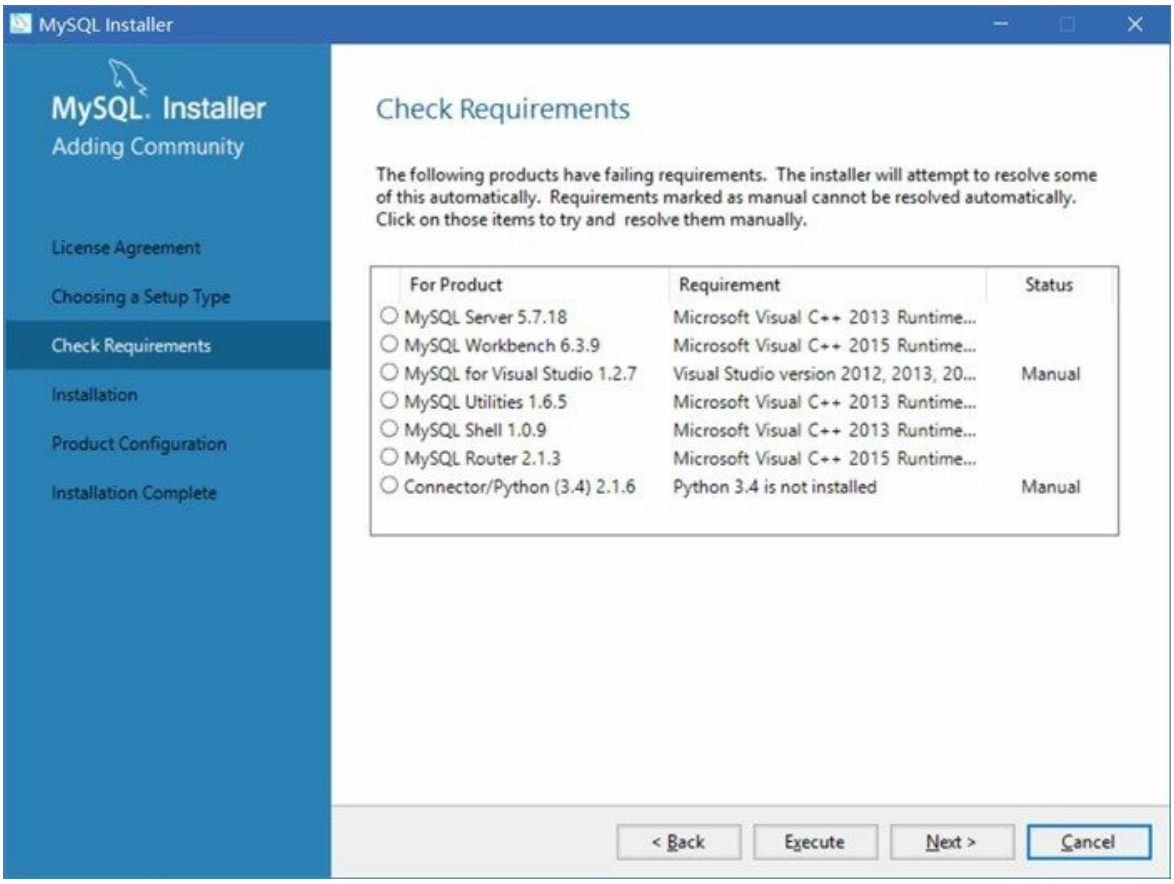


图**28-3**　安装环境检查

03. 配置过程

所需要的文件安装完成后，就会进入MySQL的配置过程。首先如图28-4所示是数据库类型选择对话框，Standalone是单个服务器，innoDB Cluster是数据库集群。

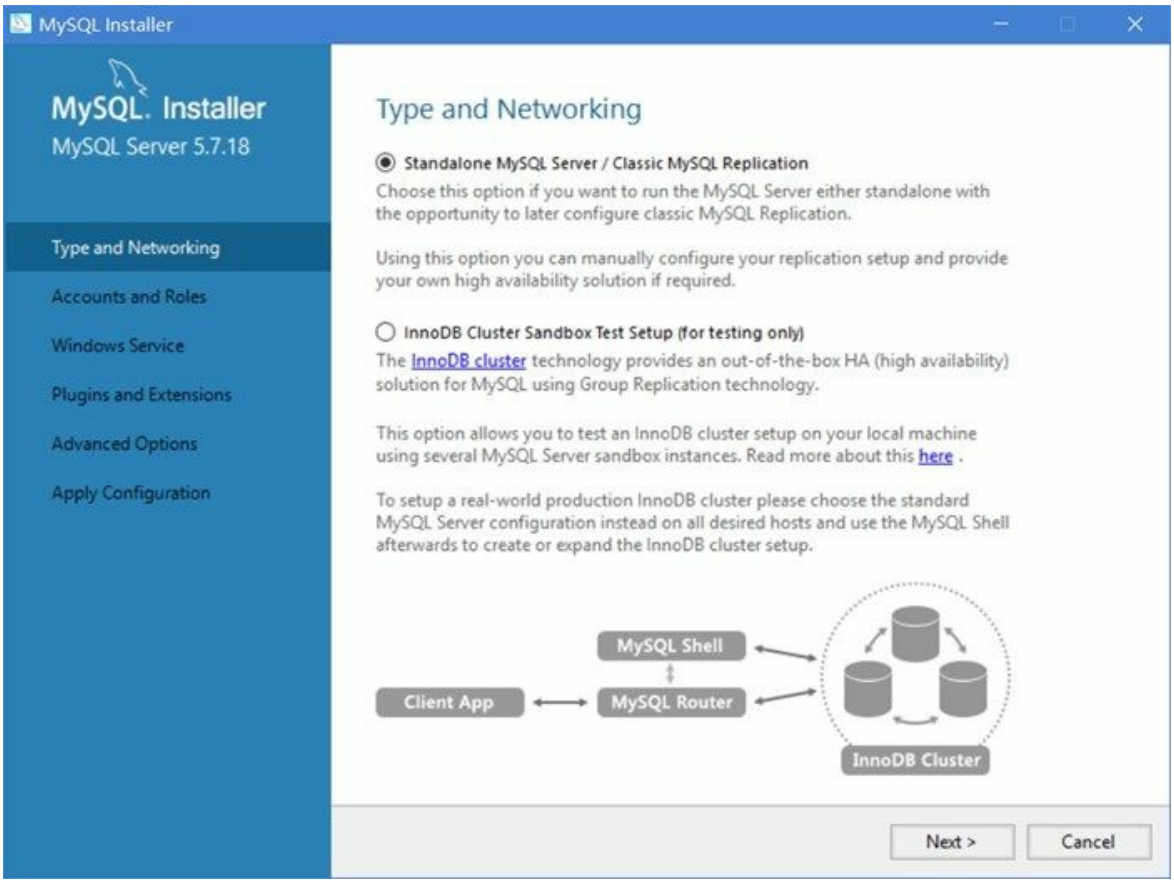


图**28-4**　数据库类型选择对话框

在图28-4所示的对话框中选择Standalone，单击Next按钮进入如图28-5所示的服务器配置类型选择对话框。在这里可以选择配置类型、通信协议和端口等，单击Config Type下拉列表可以选择如下的配置类型：

- Development Machine（开发机器）：该选项代表典型个人用桌面工作站，假定机器上运行着多个桌面应用程序。将MySQL服务器配置成使用最少的系统资源。
- Server Machine（服务器）：该选项代表服务器，MySQL服务器可以同其他应用程序一起运行，例如FTP、email和web服务器。MySQL服务器配置成使用适当比例的系统资源。
- Dedicated Machine（专用MySQL服务器）：该选项代表只运行MySQL服务的服务器。假定没有运行其它应用程序。MySQL服务器配置成使用所有可用系统资源。

根据自己的需要选择配置类型，其他的配置项目保持默认值，单击Next按钮进入如图28-6所示的账号和用户角色设置对话框。

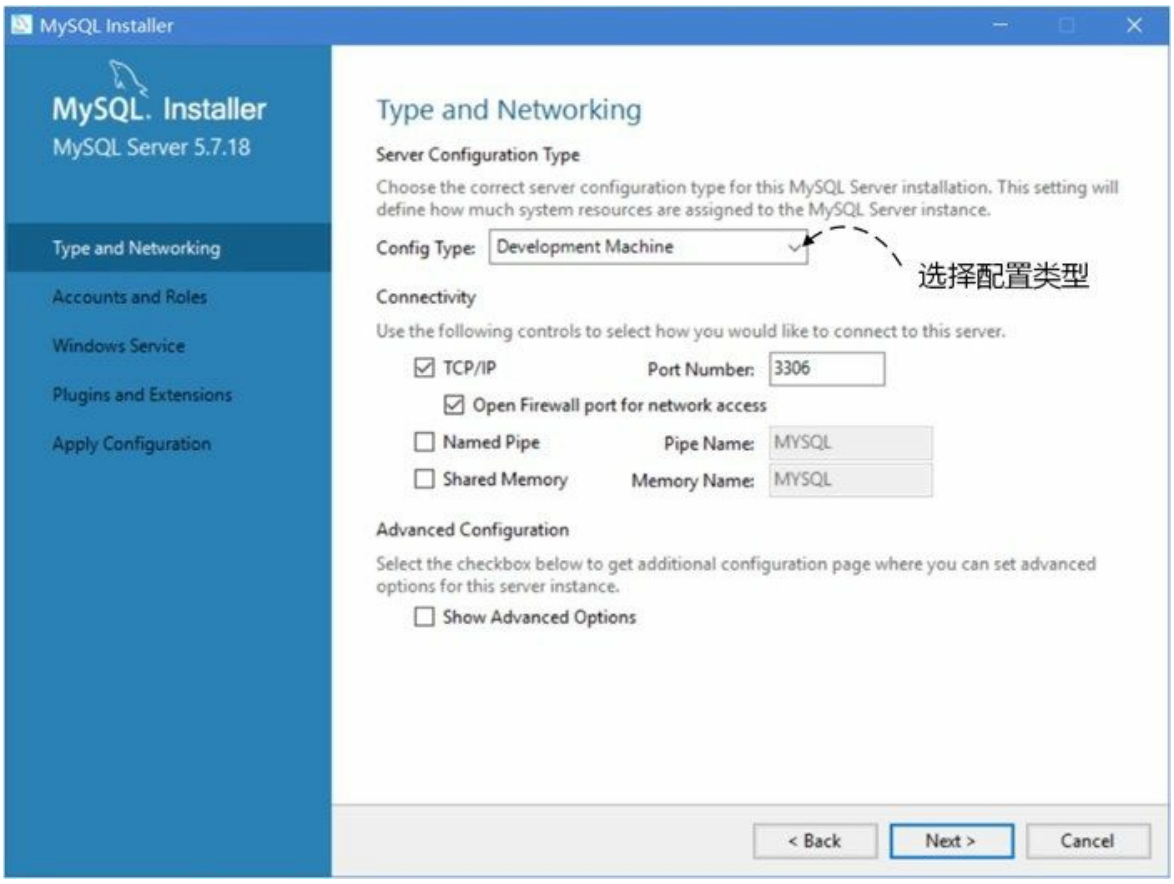


图**28-5**　服务器配置类型对话框

在图28-6所示的对话框中可以进行设置root密码，以及添加其他账号等操作。root密码必须是4位以上，根据需要设置root密码。此外，还可以单击Add User按钮添加其他的账号。

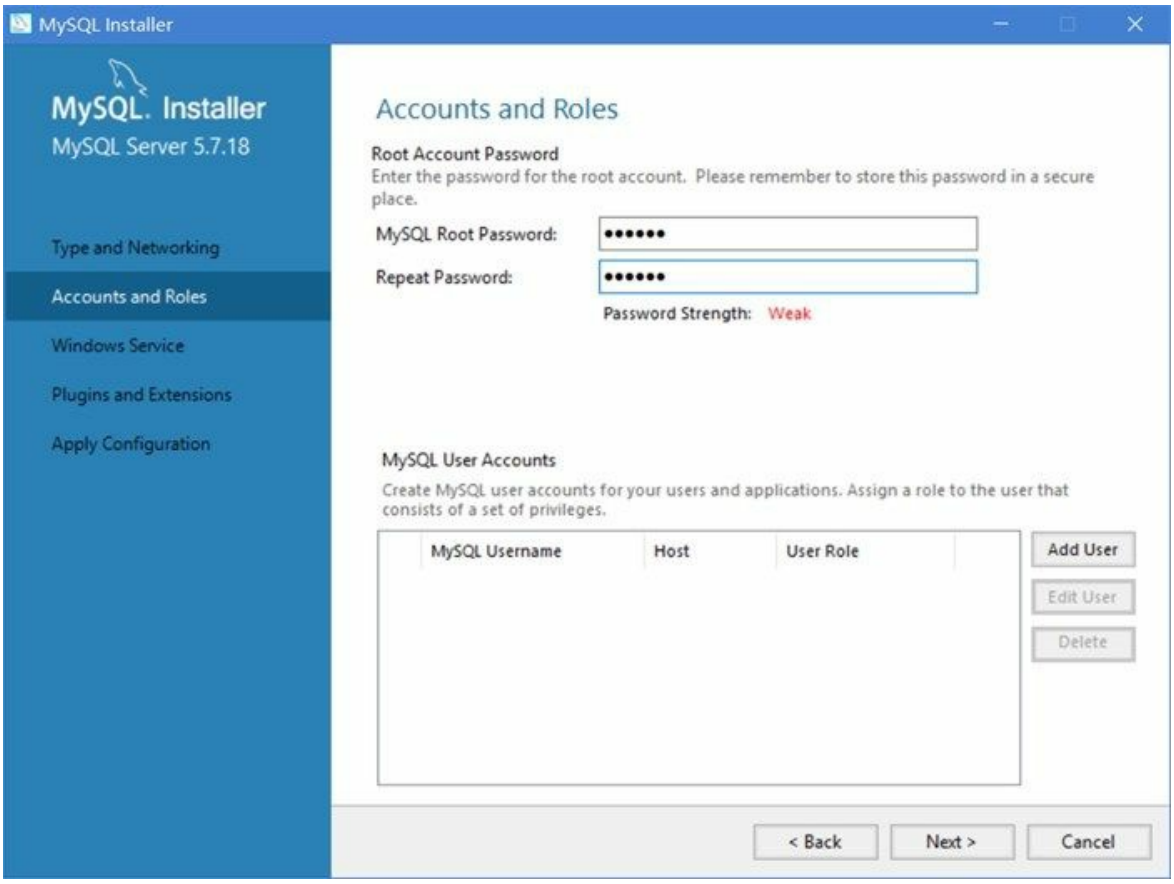


图**28-6**　账号和用户角色设置对话框

在图28-6对话框设置完成后，单击Next按钮进入图28-7所示的配置Windows服务对话框，在这里可以将MySQL数据库配置成为一个Windows服务，Windows服务可以在后台随着Windows已启动而启动，不需要认为干预。其实默认的服务名是MySQL57。

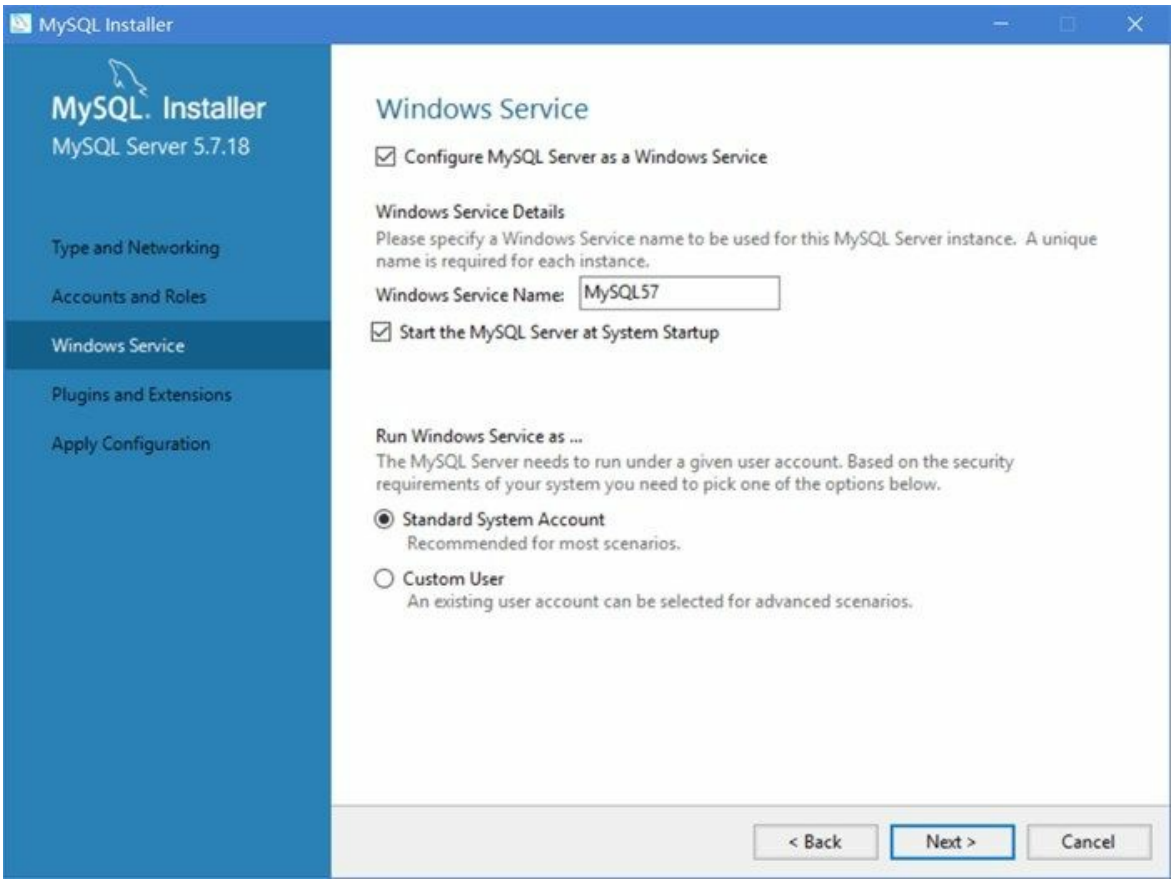


图**28-7**　配置**Windows**服务对话框

在图28-7所示配置界面之后，不需要再进行配置了，只需要单击Next按钮，这里不再赘述。

28.2.2　连接**MySQL**服务器

由于MySQL是C/S（客户端/服务器）结构的，所以应用程序包括它的客户端必须连接到服务器才能使用其服务功能。下面主要介绍MySQL本身客户端如何连接到服务器。

01. 快速连接服务器方式

MySQL for Windows版本提供一个菜单项目可以快速连接服务器，打开过程右击屏幕左下角的Windows图标，在“最近添加”中找到MySQL 5.7 Command Line Client，则会在打开一个终端窗口如图28-8所示对话框。

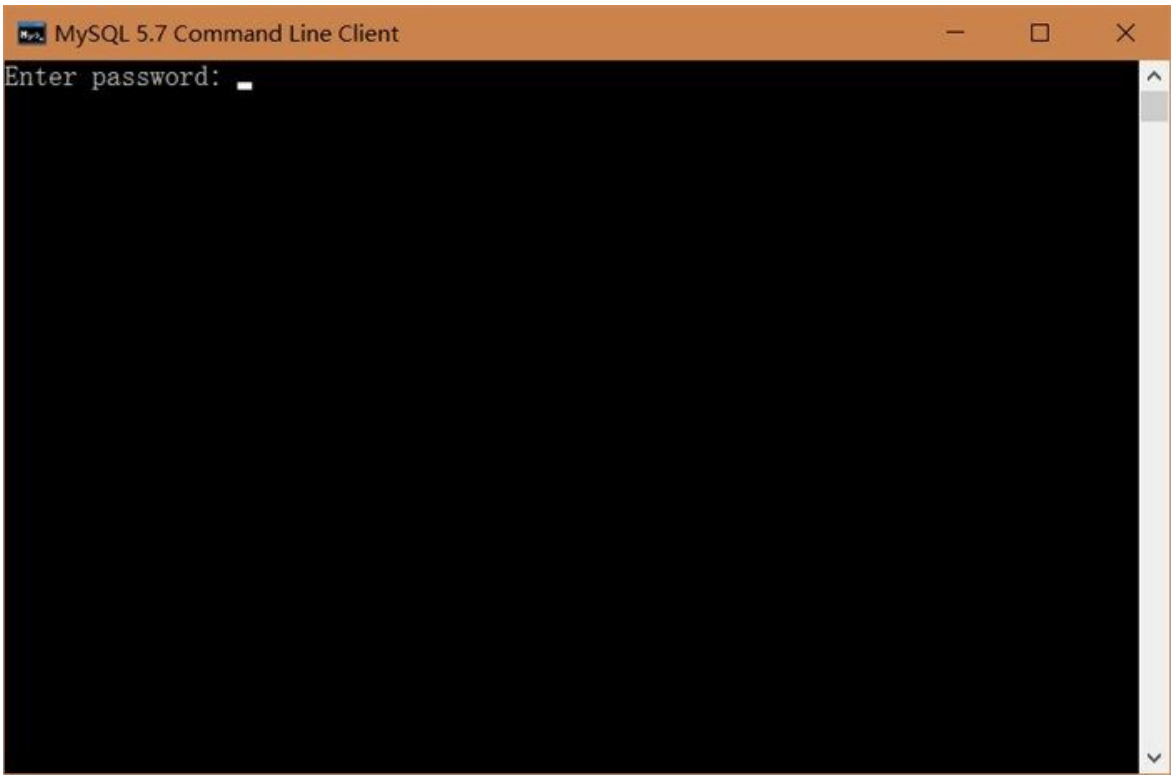


图**28-8　MySQL**命令行客户端

这个工具就是MySQL命令行客户端工具，可以使用MySQL命令行客户端工具连接到MySQL服务器，要求输入root密码。输入root密码按Enter键，如果密码正确则连接到MySQL服务器，如图28-9所示。

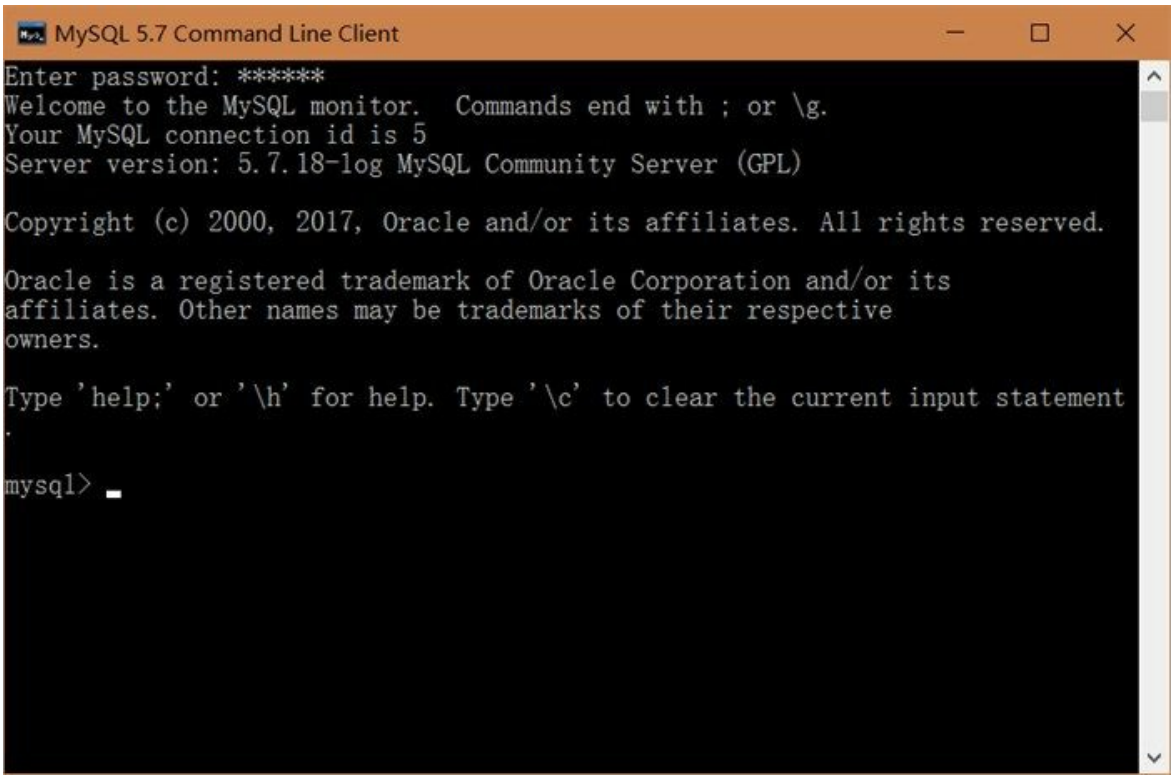


图**28-9**　使用命令行客户端连接到服务器

02. 通用的连接方式

快速连接服务器方式连接的是本地数据库，如果服务器不在本地，而是在一个远程主机上，那么需要可以使用通用的连接方式。

首先在操作系统下打开一个终端窗口，Windows下是命令行工具，在次输入mysql -h localhost -u root –p命令。如图28-10所示，如果出现“'MySQL' 不是内部或外部命令，也不是可运行的程序或批处理文件。” 的错误，则说明在环境变量的Path没有配置MySQL的Path。参考2.1.2追加C:\Program Files\MySQL\MySQL Server 5.7\bin到环境变量Path之后。

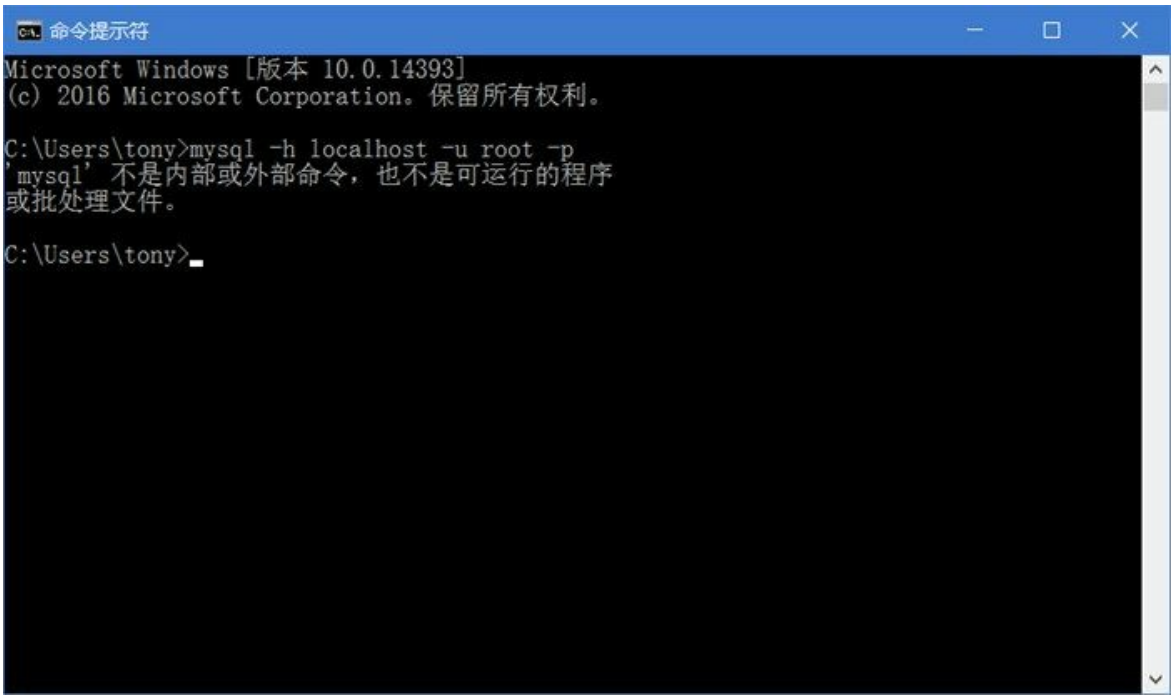


图**28-10**　环境变量中没有**MySQL**的**Path**

如果Path环境变量添加成功，重新打开命令行，再次输入mysql -h localhost -u root -p命令，然后系统会提示你输入root密码，输入密码按下Enter键，如果密码正确成功连接到服务器，会看到如图28-9所示的界面。

提示： mysql -h localhost -u root -p命令,参数说明：

-h：要连接的服务器主机名或IP地址，可以是远程的一个服务器主机，也可以是-hlocalhost方式没有空格。

-u：是服务器要验证的用户名，这个用户一定是数据库中存在的，并且具有连接服务器的权限，也可以是-uroot方式没有空格。

-p：是与上面用户对应的密码，也可以直接输入密码-p123456，123456是root密码。

所以mysql -h localhost -u root -p命令也可以替换为mysql -hlocalhost -uroot -p123456。

28.2.3　常见的管理命令

通过命令行客户端管理MySQL数据库，需要了解一些常用的命令。

01. help

第一个应该熟悉的就是help命令，help命令能够列出MySQL其他命令的帮助，在命令行客户端中输入help，不需要分号结尾，直接按下Enter键，如图28-11所示，这里都是MySQL的管理命令，这些命令大部分不需要分号结尾。

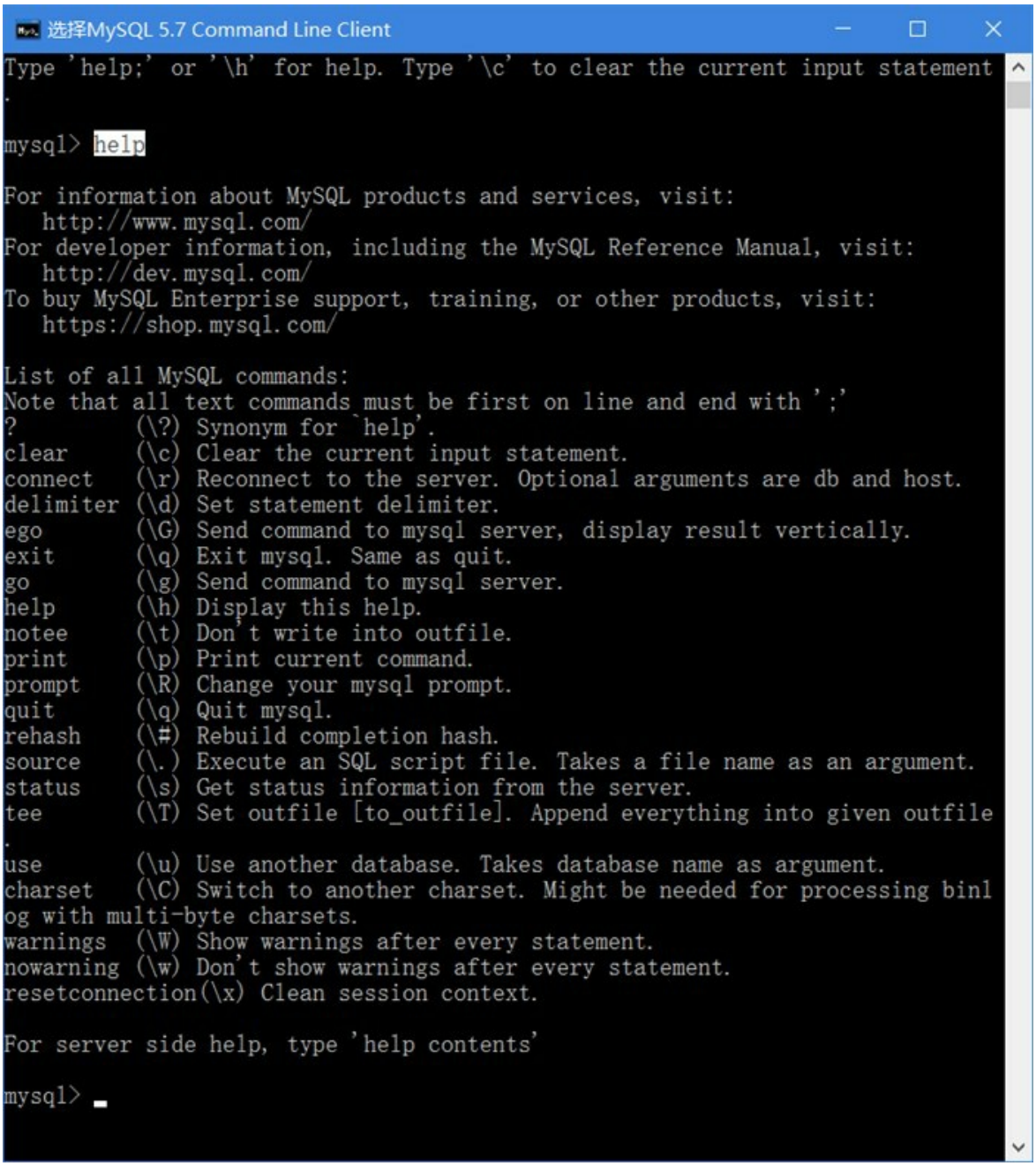


图**28-11**　使用**help**命令

02. 退出命令

如果命令行客户端中退出，可以命令行客户端中使用quit或exit命令，如图28-12所示。注意这两个命令也不需要分号结尾。

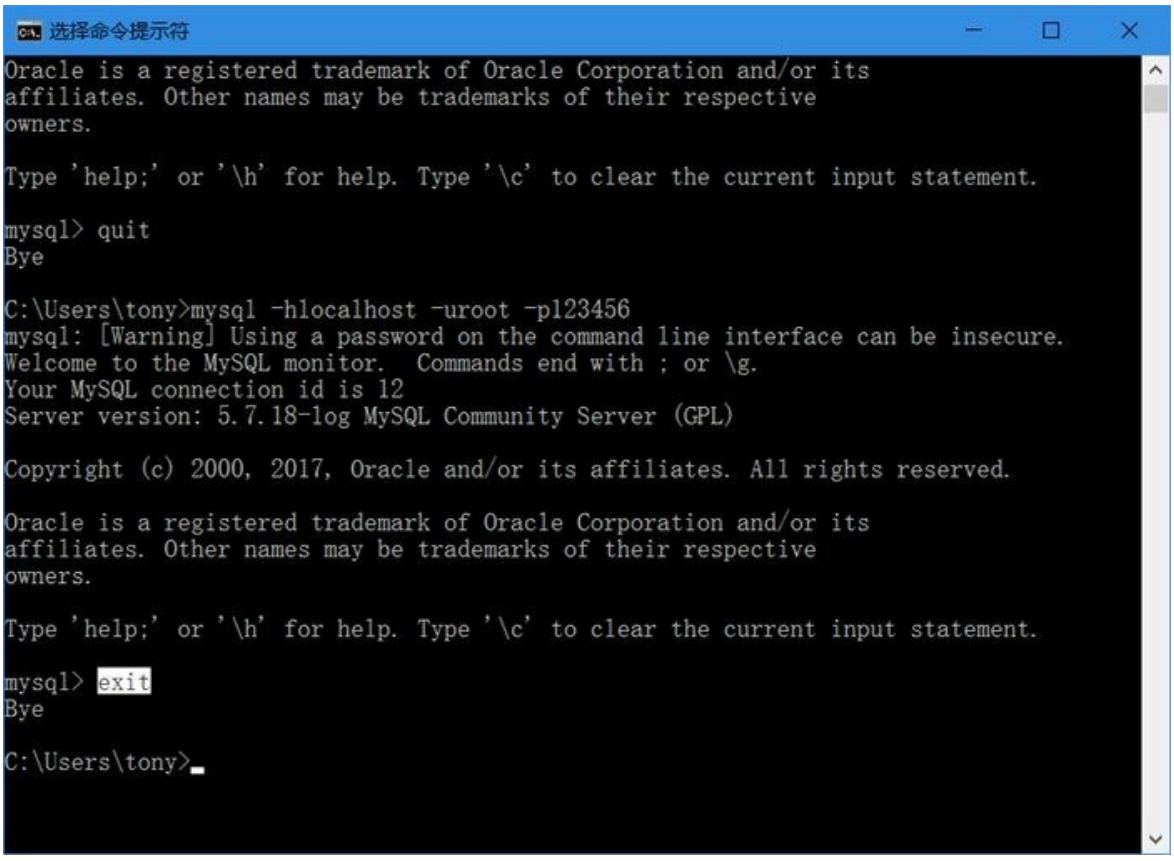


图**28-12**　使用退出命令

03. 数据库管理

在使用数据库的过程中，有时需要知道服务器中哪些数据库或自己创建和删除数据库。查看数据库的命令是show databases;，如图28-13所示，注意该命令后面是有分号结尾的。

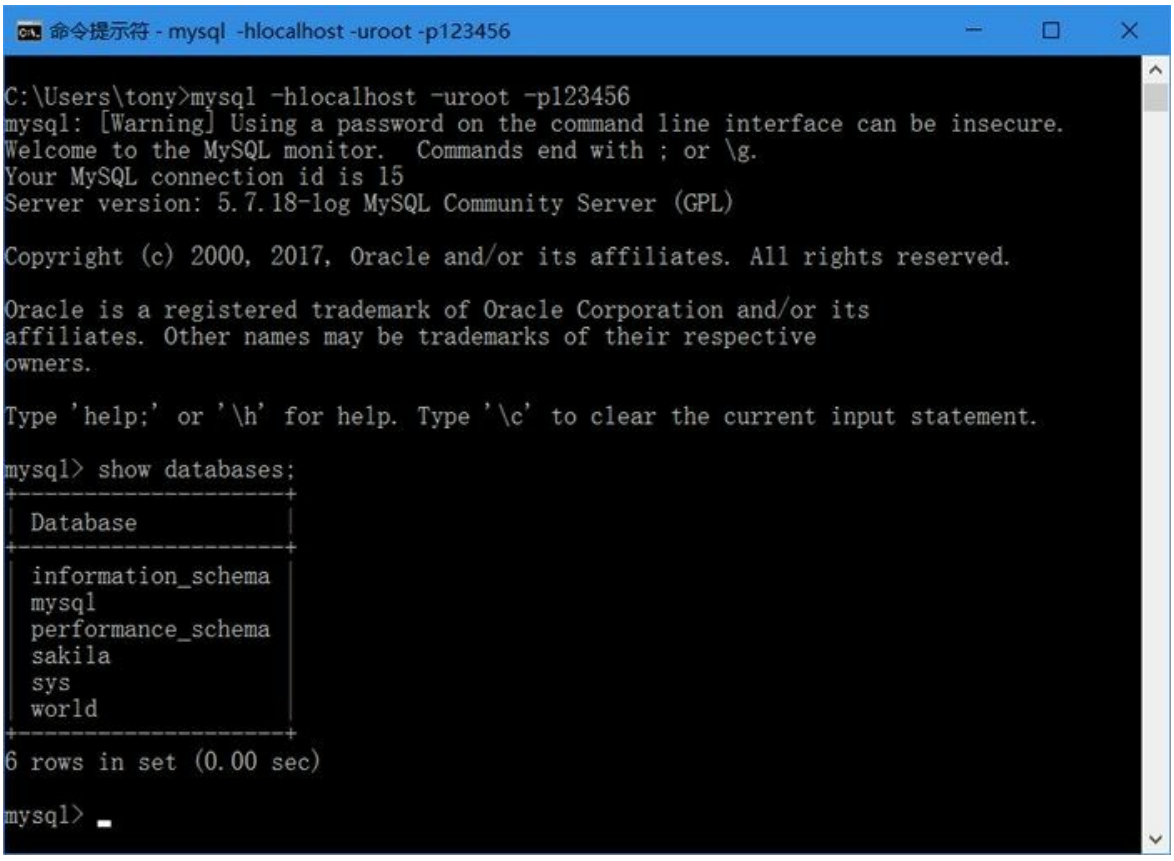


图**28-13** 查看数据库信息

创建数据库可以使用create database testdb;命令，如图28-14所示，testdb是自定义数据库名，注意该命令后面是有分号结尾的。

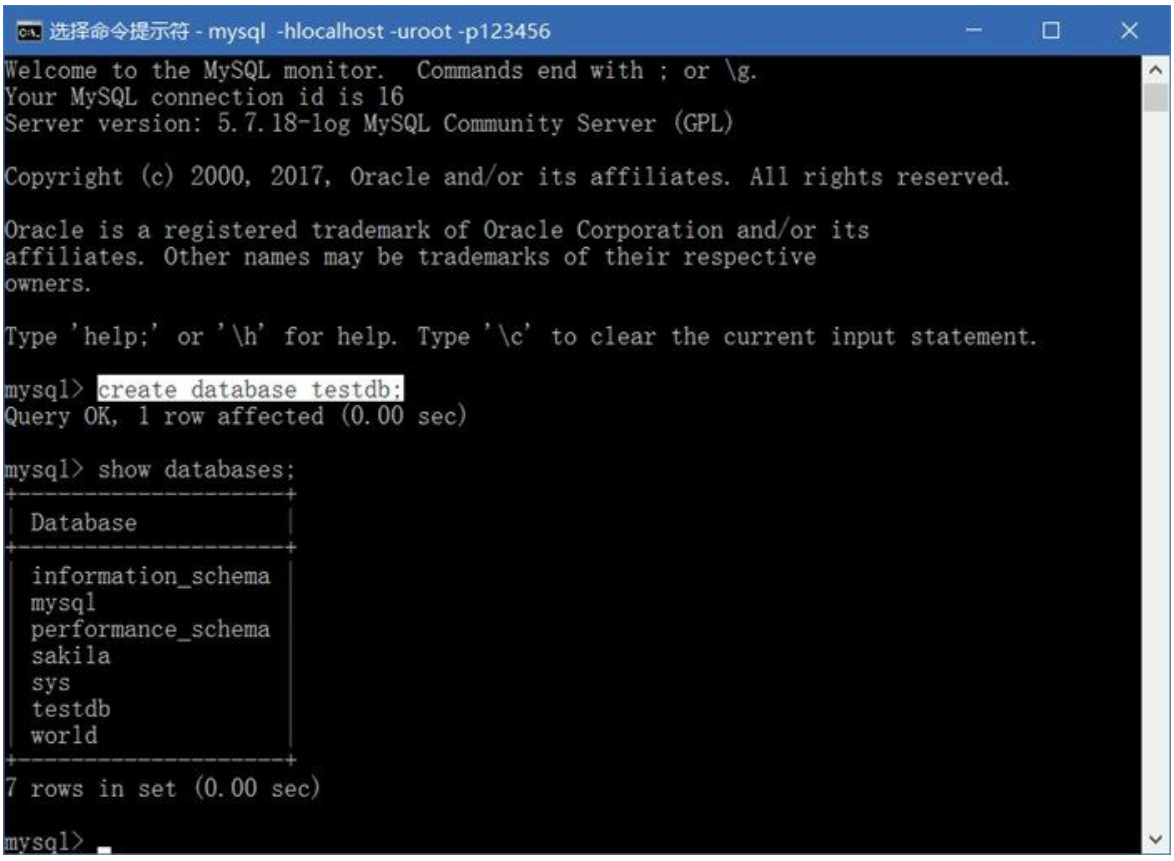


图**28-14** 创建数据库

想要删除数据库可以使用database testdb;命令，如图28-15所示，testdb是自定义数据库名，注意该命令后面是有分号结尾的。

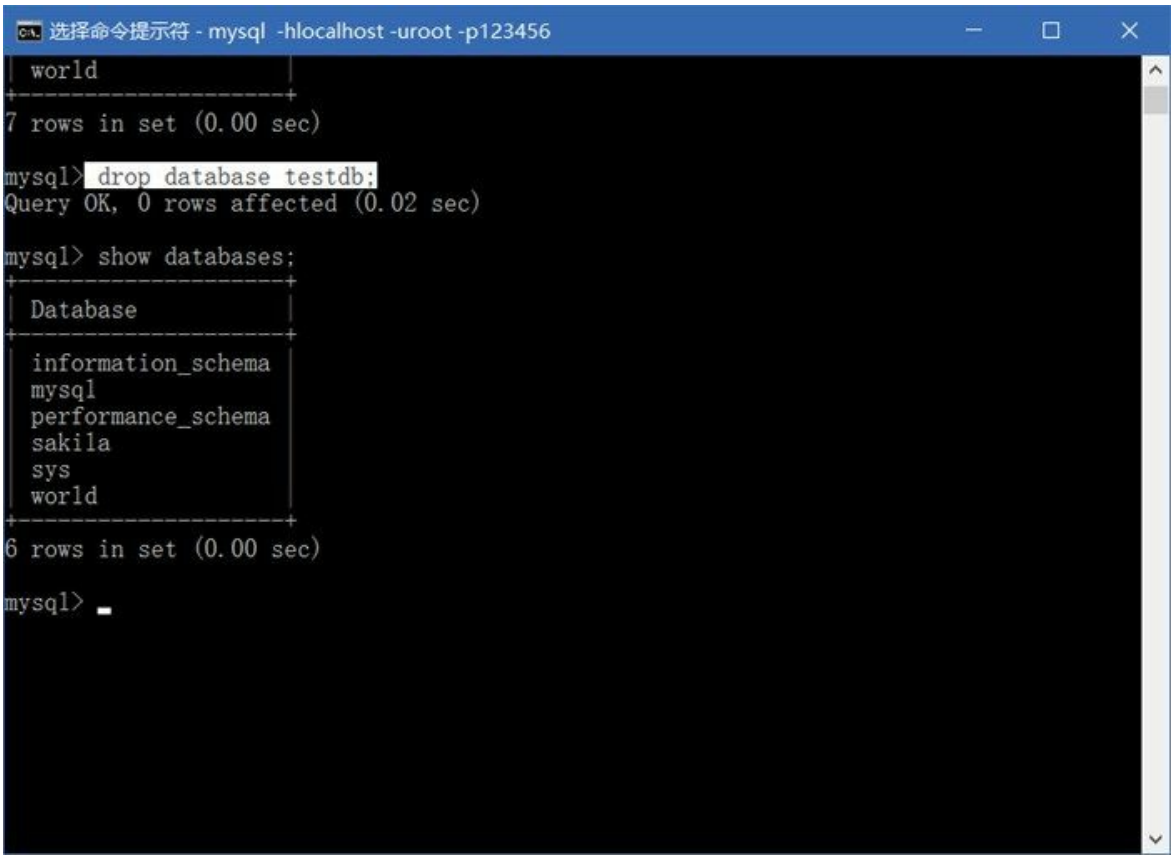


图**28-15**　删除数据库

04. 数据表管理

在使用数据库的过程中，有时需要知道某个数据库下有多少个数据表，并想查看表结构等信息。

查看有多少个数据表的命令是show tables;，如图28-16所示，注意该命令后面是有分号结尾的。因为一个服务器中有很多数据库，应该先使用use 选择数据库，如图28-16所示，use world命令结尾没有分号。如果没有选择数据库，会发生错误，如图28-16所示。

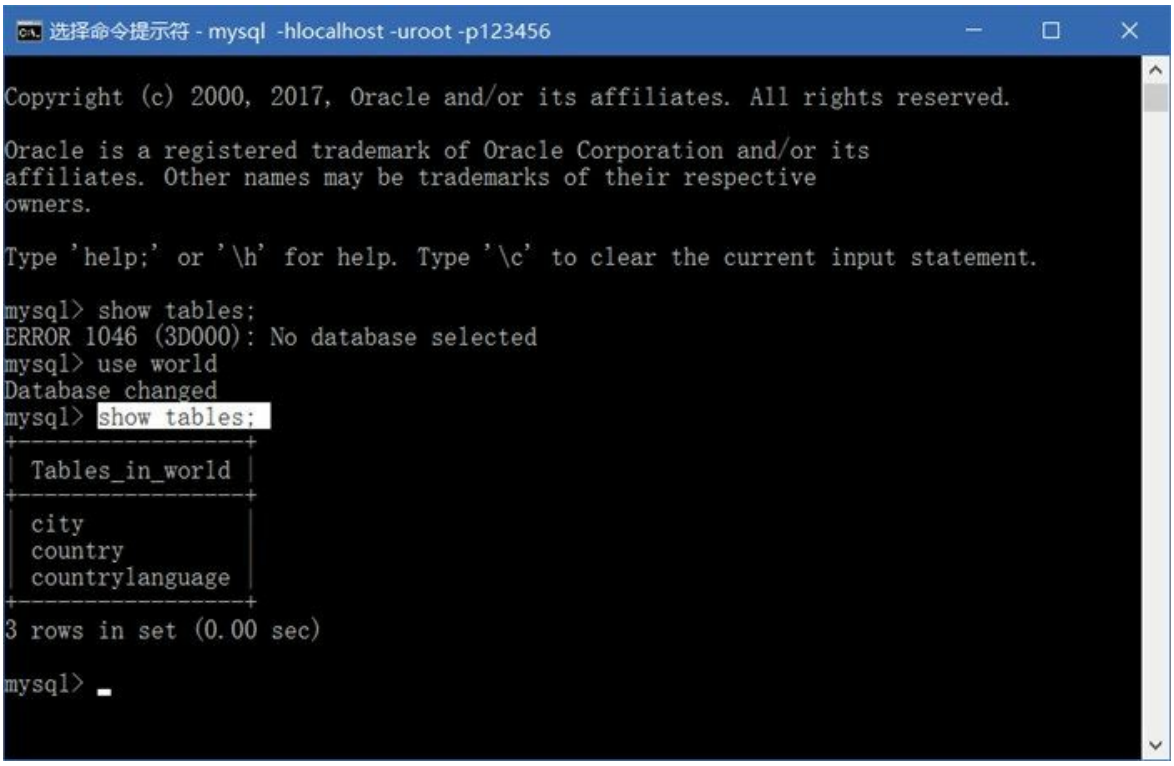


图**28-16**　查看数据库中表信息

知道了有哪些表后，还需要知道表结构，可以使用desc命令，例如像知道city表结构可以使用命令desc city;命令，如图28-17所示，注意该命令后面是有分号结尾的。

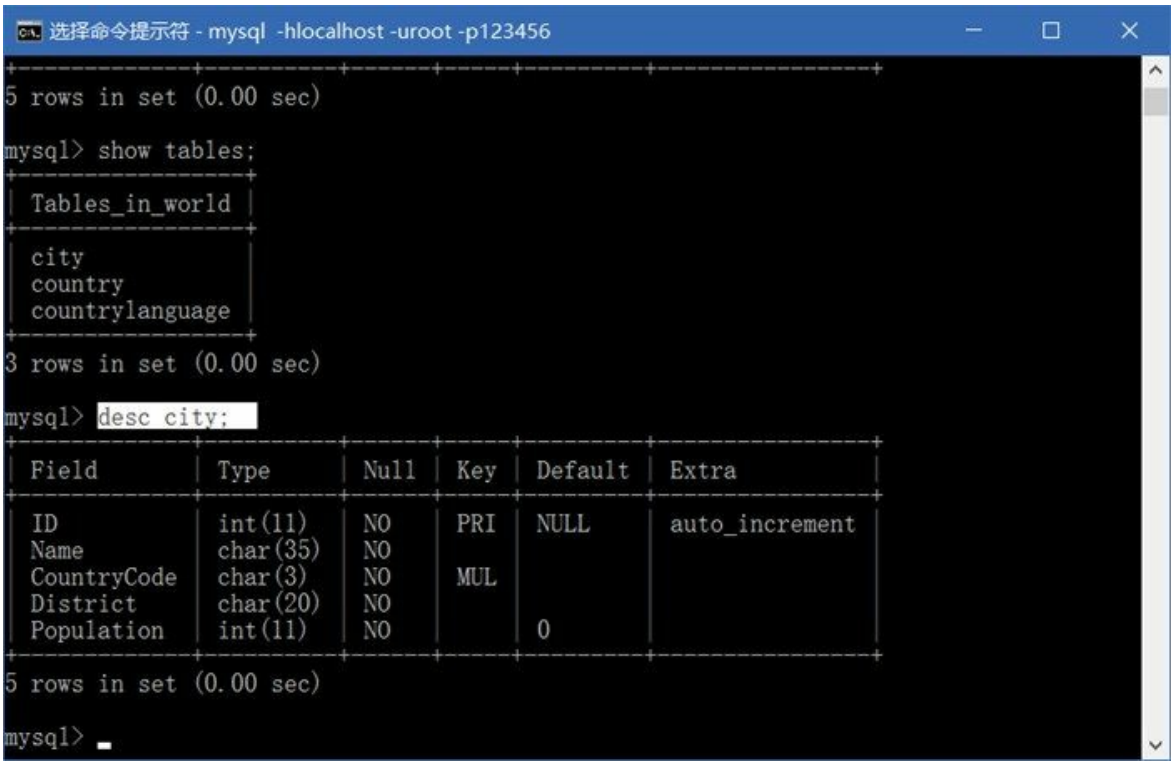


图**28-17**　查看表结构

28.3 JDBC技术

Java中数据库编程是通过JDBC（Java Database Connectivity）实现的。使用JDBC技术涉及到三种不同的角色：Java官方、开发人员和数据库厂商。

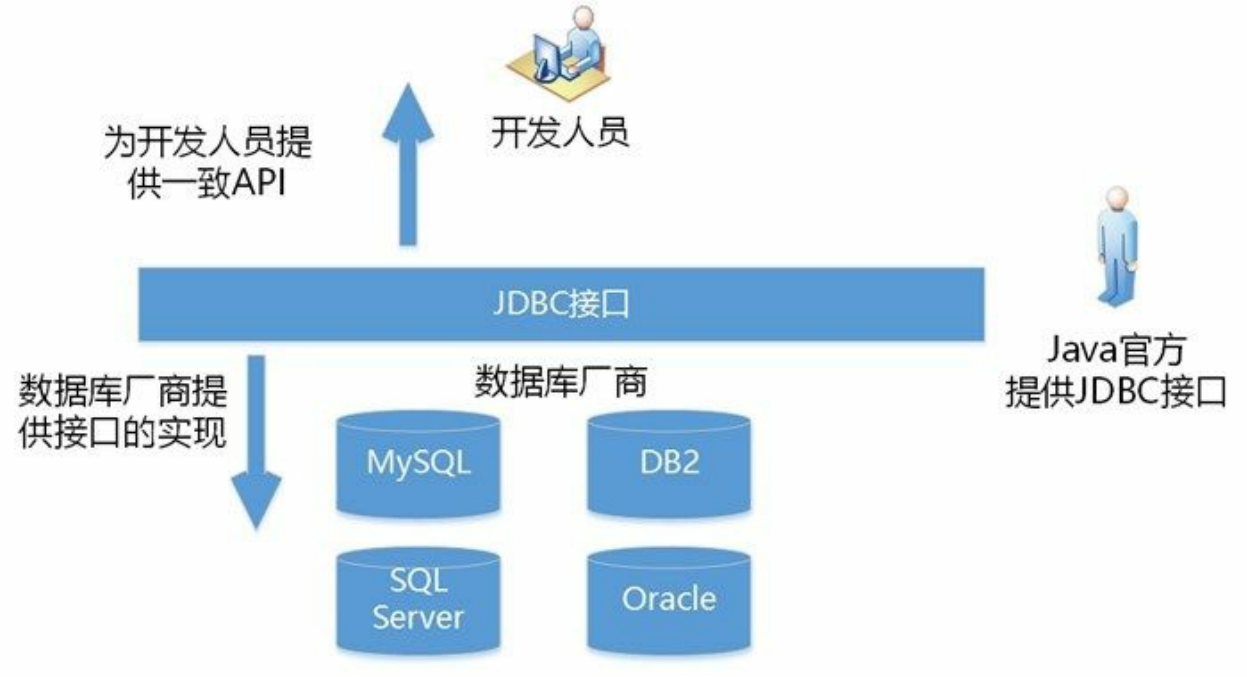


图**28-18 JDBC**技术涉及到三种不同的角色

- Java官方提供JDBC接口，如Connection、Statement和ResultSet等。
- 数据库厂商为了支持Java语言使用自己的数据库，他们根据这些接口提供了具体的实现类，这些具体实现类称为JDBC Driver（JDBC驱动程序），例如Connection是数据库连接接口，如何能够高效地连接数据库或许只有数据库厂商自己清楚，因此他们提供的JDBC驱动程序当然是最高效的，当然针对某种数据库也可能有其他第三方JDBC驱动程序。
- 对于开发人员而言，JDBC提供了一致的API，开发人员不用关心实现接口的细节。

28.3.1 JDBC API

JDBC API为Java开发者使用数据库提供了统一的编程接口，它由一组Java类和接口组成。这种类和接口来自于java.sql和javax.sql两个包。

- java.sql：这个包中的类和接口主要针对基本的数据库编程服务，如创建连接、执行语句、语句预编译和批处理查询等。同时也有一些高级的处理，如批处理更新、事务隔离和可滚动结果集等。
- javax.sql：它主要为数据库方面的高级操作提供了接口和类，提供分布式事务、连接池和行集等。

28.3.2 加载驱动程序

在编程实现数据库连接时，JVM必须先加载特定厂商提供的数据库驱动程序。使用Class.forName()方

法实现驱动程序加载过程，该方法在26章介绍过。

不同驱动程序的装载方法如下：

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");     //JDBC-ODBC桥接，Java自带
Class.forName("特定的JDBC驱动程序类名");           //数据库厂商提供
```

例如加载MySQL驱动程序代码如下：

```
Class.forName("com.mysql.jdbc.Driver");
```

如果直接这样运行程序会抛出如下的ClassNotFoundException异常。

```
java.lang.ClassNotFoundException: com.mysql.jdbc.Driver
```

这是因为程序无法找到MySQL驱动程序com.mysql.jdbc.Driver类，这需要配置当前项目的类路径（Classpath），在类路径中包含MySQL驱动程序。MySQL驱动程序是在MySQL安装目录中Connector.J 5.1中的mysql-connector-java-xxx-bin.jar文件。笔者默认安装驱动程序文件路径如下：

```
"C:\Program Files (x86)\MySQL\Connector.J 5.1\mysql-connector-java-5.1.41-bin.jar"
```

数据库厂商提供驱动程序一般都是.jar文件。

提示　一般在发布java文件时，会把字节码文件（class文件）打包成.jar文件，.jar文件是一种基于.zip结构的压缩文件。

在Eclipse中将.jar文件添加到项目步骤，首先在Eclipse中选择项目，右击菜单中选择“构建路径”→“配置构建路径”，在弹出对话框中选择“Java构建路径”→“库”如图28-19所示，其中“添加JAR”是在当前项目中找.jar文件，“添加外部JAR”是在项目之外找.jar文件。

提示　笔者推荐前一种方式（添加JAR），它的好处在于当你把Eclipse项目复制给别人时，这些.jar文件也一起复制，项目重新编译和运行时，不会发生找不到.jar文件情况。而添加外部JAR方法，只是添加了一个基于本机的绝对路径，当你的项目复制给别人时，会发生找不到.jar文件情况。

图**28-19**　构建路径对话框

在图28-19对话框中如果单击“添加JAR”按钮，则弹出如图28-20所示的对话框。该对话框只能选择该项目内.jar文件，所以要保证所添加的.jar文件应该在Eclipse项目中,如果文件存在，选择.jar文件单击“确定”按钮添加。

图**28-20**　添加**jar**文件到项目类路径

如果文件在Eclipse中没有，需要从操作系统的资源管理器中复制到Eclipse中，参考图28-21，将文件复制到Eclipse中。

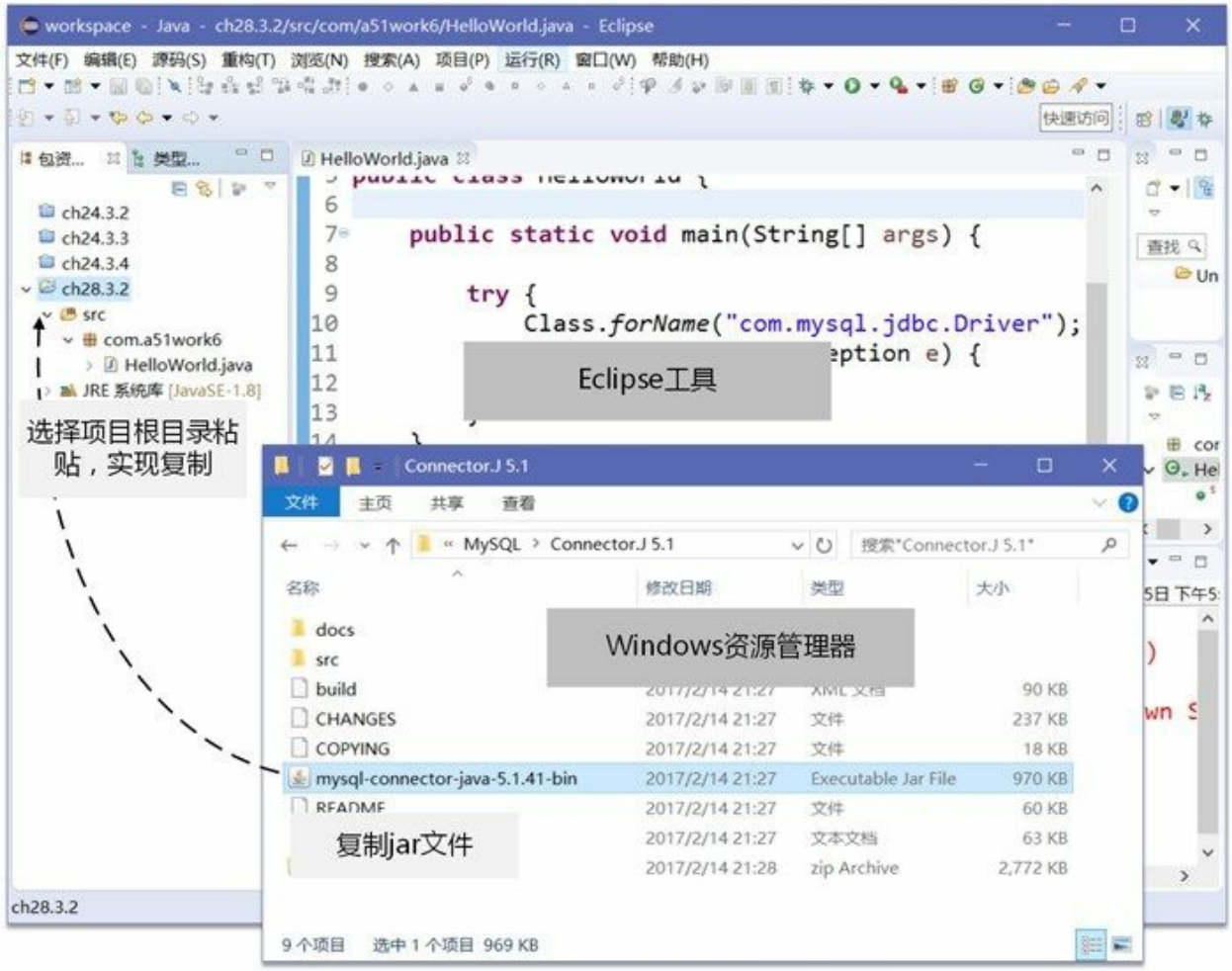


图**28-21**　从资源管理器复制文件到**Eclipse**

将驱动程序.jar文件添加类路径中后，再运行上面的程序看看是否还有ClassNotFoundException异常。

28.3.3　建立数据连接

驱动程序加载成功就可以进行数据库连接了。建立数据库连接可以通过调用DriverManager类的getConnection()方法实现，该方法有几个重载版本，如下所示。

- static Connection getConnection(String url)：尝试通过一个URL建立数据库连接，调用此方法时，DriverManager会试图从已注册的驱动中选择恰当的驱动来建立连接。
- static Connection getConnection(String url，Properties info)：尝试通过一个URL建立数据库连接，一些连接参数（如user和password）可以按照键值对的形式放置到info中，Properties是Hashtable的子类，它是一种Map结构。
- static Connection getConnection(String url, String user, String password)：尝试通过一个URL建立数据库连接，指定数据库用户名和密码。

上面的几个getConnection()方法都会抛出受检查的SQLException异常，注意处理这个异常。

JDBC的URL类似于其他场合的URL，它的语法如下：

```
jdbc:<subprotocol>:<subname>
```

这里有三个部分，它们用冒号隔离。

01. 协议：jdbc表示协议，它是唯一的，JDBC只有这一种协议。

02. 子协议：主要用于识别数据库驱动程序，也就是说，不同的数据库驱动程序的子协议不同。

03. 子名：它属于专门的驱动程序，不同的专有驱动程序可以采用不同的实现。

对于不同的数据库，厂商提供的驱动程序和连接的URL都不同，在这里总结后如表28-1所示。

表 **28-1**　数据库厂商提供的驱动程序和连接的**URL**

数据库名	驱动程序	URL
MS SQLServer	com.microsoft.jdbc.sqlserver.SQLServerDriver	jdbc:microsoft:sqlserver://[ip]:[port];user=[user];password=[password]
JDBC-ODBC	sun.jdbc.odbc.JdbcOdbcDriver	jdbc:odbc:[odbcsource]
Oracle thin Driver	oracle.jdbc.driver.OracleDriver	jdbc:oracle:thin:@[ip]:[port]:[sid]
MySQL	com.mysql.jdbc.Driver	jdbc:mysql://ip/database

建立数据连接示例代码如下：

```
    //HelloWorld.java文件
    package com.a51work6;

    import java.sql.Connection;
    import java.sql.DriverManager;
    import java.sql.SQLException;

    public class HelloWorld {

        public static void main(String[] args) {

            try {
                Class.forName("com.mysql.jdbc.Driver");
                System.out.println("驱动程序加载成功...");

            } catch (ClassNotFoundException e) {
                System.out.println("驱动程序加载失败...");
                // 退出
                return;
            }

            String url = "jdbc:mysql://localhost:3306/MyDB";
            String user = "root";
            String password = "123456";

            try (Connection conn = DriverManager.getConnection(url, user, password)) {  ①

                System.out.println("数据库连接成功：" + conn);

            } catch (SQLException e) {
                e.printStackTrace();
            }

        }
```

```
    }
```

上述代码第①行使用DriverManager的getConnection(String url, String user, String password)方法建立数据连接，在url中3306是数据库端口号，MyDB是MySQL服务器中的数据库。

另外Connection接对象是通过自动资源管理技术释放资源的。

> 注意　Connection对象代表的数据连接不能被JVM的垃圾收集器回收，在使用完连接后必须关闭（调用close()方法），否则连接会保持一段比较长的时间，直到超时。Java 7之前都在finally模块中关闭数据库连接。Java 7之后可以Connection接口继承了AutoCloseable接口，可以通过自动资源管理技术释放资源。

事实上上面代码虽然可以成功建立连接，但是控制台会有警告，警告如下：

```
    WARN: Establishing SSL connection without server's identity verification is not recommended. Accor
```

这是由于现在的网络通信为了提高网络完全，都要求使用SSL[4](Secure Sockets Layer 安全套接层) 安全协议。但是由于各种原因，很多服务器并未使用SSL安全协议，特别是对于学习和测试阶段可以不使用SSL安全协议。为此，需要修改url连接字符串：

[4]SSL为网络通信提供安全及数据完整性的一种安全协议，SSL在传输层对网络连接进行加密。

```
    "jdbc:mysql://localhost:3306/MyDB?verifyServerCertificate=false&useSSL=false"
```

其中verifyServerCertificate设置为false表示不进行安全认证，useSSL设置为false表示不使用SSL进行网络通信。

数据连接的url字符串可以有很多参数对，包括数据库用户名和密码也都可以参数对形式放到url字符串中，有的url字符串会很长维护起来不方便，可以把这些参数对放置到Properties对象中，示例代码如下：

```
    //HelloWorldWithProp.java文件
    package com.a51work6;

    import java.sql.Connection;
    import java.sql.DriverManager;
    import java.sql.SQLException;
    import java.util.Properties;

    public class HelloWorldWithProp {

        public static void main(String[] args) {

            try {
                Class.forName("com.mysql.jdbc.Driver");
                System.out.println("驱动程序加载成功...");

            } catch (ClassNotFoundException e) {
                System.out.println("驱动程序加载失败...");
                // 退出
                return;
            }

            String url = "jdbc:mysql://localhost:3306/MyDB";
            //创建Properties对象
```

```
        Properties info = new Properties();                     ①
        info.setProperty("user", "root");                       ②
        info.setProperty("password", "123456");                 ③
        info.setProperty("verifyServerCertificate", "false");   ④
        info.setProperty("useSSL", "false");                    ⑤

        try (Connection conn = DriverManager.getConnection(url, info)) {    ⑥

            System.out.println("数据库连接成功：" + conn);

        } catch (SQLException e) {
            e.printStackTrace();
        }

    }
}
```

上述代码第①行是创建Properties对象，代码第②行~第⑤行是设置参数，setProperty()方法键和值都是字符串类型。代码第⑥行是DriverManager的getConnection(String url，Properties info)方法建立数据连接。

但是上述代码还是有不尽人意的地方，就是这些参数都是“硬编码[5]”在程序代码中的，程序编译之后不能修改。但是数据库用户名、密码、服务器主机名、端口等等这一切，在开发阶段和部署阶段可能完全不同，这些参数信息应该可以配置的，可以放到一个属性文件中，借助于输入流，可以在运行时读取属性文件内容到Properties对象中。具体示例代码如下：

[5]硬编码俗称“写死”，指将可变变量用一个固定值来代替的方法，用这种方法编译后，如果以后需要更改此变量就非常困难了。

```
//HelloWorldWithPropFile.java文件
package com.a51work6;

...

public class HelloWorldWithPropFile {

    public static void main(String[] args) {

        //加载驱动程序
        ...

        Properties info = new Properties();        ①
        try {
            InputStream input = HelloWorldWithPropFile.class.getClassLoader()
                    .getResourceAsStream("config.properties");        ②

            info.load(input);                      ③

        } catch (IOException e) {
            // 退出
            return;
        }

        String url = "jdbc:mysql://localhost:3306/MyDB";

        try (Connection conn = DriverManager.getConnection(url, info)) {

            System.out.println("数据库连接成功：" + conn);

        } catch (SQLException e) {
            e.printStackTrace();
        }
```

```
        }
    }
```

上述代码第①行创建一个Properties对象。代码第②行获得config.properties属性文件输入流对象，属性文件一般放到src目录与源代码文件放置在一起，但是编译时，这些文件会复制到字节码文件所在的目录中，这种目录称为资源目录，获得资源目录要通过Java反射机制，HelloWorldWithPropFile.class.getClassLoader().getResourceAsStream("config.properties")语句能够获得运行时config.properties的文件输入流对象。

代码第③行是从流中加载信息到Properties对象中。

config.properties文件内容如下：

```
#Tony
user=root
password=123456
useSSL=false
verifyServerCertificate=false
```

在开发和部署阶段使用文本编辑器修改该文件，不需要修改程序代码。

28.3.4 三个重要接口

下面重点介绍一下JDBC API中最重要的三个接口：Connection、Statement和ResultSet。

01. Connection接口

java.sql.Connection接口的实现对象代表与数据库的连接，也就是在Java程序和数据库之间建立连接。Connection接口中常用的方法：

- Statement createStatement()：创建一个语句对象，语句对象用来将SQL语句发送到数据库。
- PreparedStatement prepareStatement(String sql)：创建一个预编译的语句对象，用来将参数化的SQL语句发送到数据库，参数包含一个或者多个问号“?”占位符。
- CallableStatement prepareCall(String sql)：创建一个调用存储过程的语句对象，参数是调用的存储过程，参数包含一个或者多个问号“?”为占位符。
- close()：关闭到数据库的连接，在使用完连接后必须关闭，否则连接会保持一段比较长的时间，直到超时。
- isClosed()：判断连接是否已经关闭。

02. Statement接口

java.sql.Statement称为语句对象，它提供用于向数据库发出SQL语句，并且访问结果。Connection接口提供了生成Statement的方法，一般情况下可以通过connection.createStatement()方法就可以得到Statement对象。

有三种Statement接口：java.sql.Statement、java.sql.PreparedStatement和java.sql.CallableStatement，其中PreparedStatement继承Statement接口，CallableStatement继承PreparedStatement接口。Statement实现对象用于执行基本的SQL语句，PreparedStatement实现对象用于执行预编译的SQL语句，CallableStatement实现对象用于用来调用数据库中的存储过程。

> 注意　预编译SQL语句是在程序编译的时一起进行编译，这样的语句在数据库中执行时候，不需要编译过程，直接执行SQL语句，所以速度很快。在预编译SQL语句会有一些程序执行时才能确定的参数，这些参数采用“?”占位符，直到运行时再用实际参数替换。

Statement提供了许多方法，最常用的方法如下：

- executeQuery()：运行查询语句，返回ResultSet对象。
- executeUpdate()：运行更新操作，返回更新的行数。
- close()：关闭语句对象。
- isClosed()：判断语句对象是否已经关闭。

Statement对象用于执行不带参数的简单SQL语句，它的典型使用如下。

```
Connection conn = DriverManager.getConnection("jdbc:odbc:accessdb",  "admin",  "admin");
Statement stmt = conn.createStatement();
ResultSet rst = stmt.executeQuery("select userid, name from user");
```

PreparedStatement对象用于执行带参数的预编译SQL语句，它的典型使用如下。

```
Connection conn = DriverManager.getConnection("jdbc:odbc:accessdb",  "admin",  "admin");
PreparedStatement pstmt = conn.prepareStatement("insert into user values(?，?)");
pstmt.setInt(1,10);           //绑定第一个参数
pstmt.setString(2,"guan");    //绑定第二个参数
pstmt.executeUpdate();        //执行SQL语句
```

上述SQL语句"insert into user values(?，?)"在Java源程序编译时一起编译，两个问号占位符所代表的参数，在运行时绑定。

注意　绑定参数时需要注意两个问题：绑定参数顺序和绑定参数的类型，绑定参数索引是从1开始的，而不是从0开始的。根据绑定参数的类型不同选择对应的set方法。

CallableStatement对象用于执行对数据库已存储过程的调用，它的典型使用如下。

```
Connection conn = DriverManager.getConnection("jdbc:odbc:accessdb",  "admin",  "admin");
strSQL = "{call proc_userinfo(?，?)}";
java.sql.CallableStatement sqlStmt = conn.prepaleCall(strSQL);
sqlStmt.setString(1，"tony");
sqIStmt.setString(2，"tom");
//执行存储过程
int i = sqlStmt.exeCuteUpdate();
```

03. ResultSet接口

在Statement执行SQL语句时，如果是SELET语句会返回结果集，结果集通过接口java.sql.ResultSet描述的，它提供了逐行访问结果集的方法，通过该方法能够访问结果集中不同字段的内容。

ResultSet提供了检索不同类型字段的方法，最常用的方法介绍如下：

- close()：关闭结果集对象。
- isClosed()：判断结果集对象是否已经关闭。
- next()：将结果集的光标从当前位置向后移一行。
- getString()：获得在数据库里是CHAR 或 VARCHAR等字符串类型的数据，返回值类型是String。
- getFloat()：获得在数据库里是浮点类型的数据，返回值类型是float。
- getDouble()：获得在数据库里是浮点类型的数据，返回值类型是double。
- getDate()：获得在数据库里是日期类型的数据，返回值类型是java.sql.Date。
- getBoolean()：获得在数据库里是布尔数据的类型，返回值类型是boolean。
- getBlob()：获得在数据库里是Blob(二进制大型对象)类型的数据，返回值类型是Blob类型。
- getClob()：获得在数据库里是Clob(字符串大型对象)类型的数据，返回值类型是Clob。

这些方法要求有列名或者列索引，如getString()方法的两种情况：

public String getString(int columnlndex) throws SQLException

public String getString(String columnName) throws SQLException

方法getXXX提供了获取当前行中某列值的途径，在每一行内，可按任何次序获取列值。使用列索引有时会比较麻烦，这个顺序是select语句中的顺序：

```
select * from user
select userid, name from user
select name,userid from user
```

注意　columnlndex列索引是从1开始的，而不是从0开始的。这个顺序与select语句有关，如果select使用*返回所有字段，如select * from user语句，那么列索引是数据表中字段的顺序；如果select指定具体字段，如select userid, name from user或select name,userid from user，那么列索引是select指定字段的顺序。

ResultSet示例代码如下：

```
//HelloWorldWithPropFile.java文件
...
String url = "jdbc:mysql://localhost:3306/MyDB";

try ( // 自动资源管理技术释放资源
        Connection conn = DriverManager.getConnection(url, info);
        Statement stmt = conn.createStatement();
        ResultSet rst = stmt.executeQuery("select name,userid from user")) {

    while (rst.next()) {
        System.out.printf("name:%s    id:%d\n", rst.getString("name"), rst.getInt(2));
    }

} catch (SQLException e) {
    e.printStackTrace();
}
```

上述代码可见Connection对象、Statement对象和ResultSet对象的释放采用自动资源管理。

在遍历结果集时使用了rst.next()方法，next()是将结果集光标从当前位置向后移一行，结果集光标最初位于第一行之前；第一次调用 next 方法使第一行成为当前行；第二次调用使第二行成为当前行，依此类推。如果新的当前行有效，则返回 true；如果不存在下一行，则返回false。

注意　Connection对象、Statement对象和ResultSet对象都不能被JVM的垃圾收集器回收，在使用完后都必须关闭（调用它们的close()方法）。Java 7之前都在finally模块中关闭释放资源。Java 7之后它们都继承了AutoCloseable接口，可以通过自动资源管理技术释放资源。

28.4 案例：数据CRUD操作

对数据库表中数据可以进行4类操作：数据插入（Create）、数据查询（Read）、数据更新（Update）和数据删除（Delete），也是俗称的“增、删、改、查”。

本节通过一个案例介绍如何通过JDBC技术实现Java对数据的CRUD操作。

28.4.1 数据库编程一般过程

在讲解案例之前，有必要先介绍一下通过JDBC进行数据库编程的一般过程。

如图28-22所示是数据库编程的一般过程，其中查询（Read）过程最多需要7个步骤，修改（C插入、U更新、D删除）过程最多需要5个步骤。这个过程采用了预编译语句对象进行数据操作，所以有可能进行绑定参数，见第4步骤。

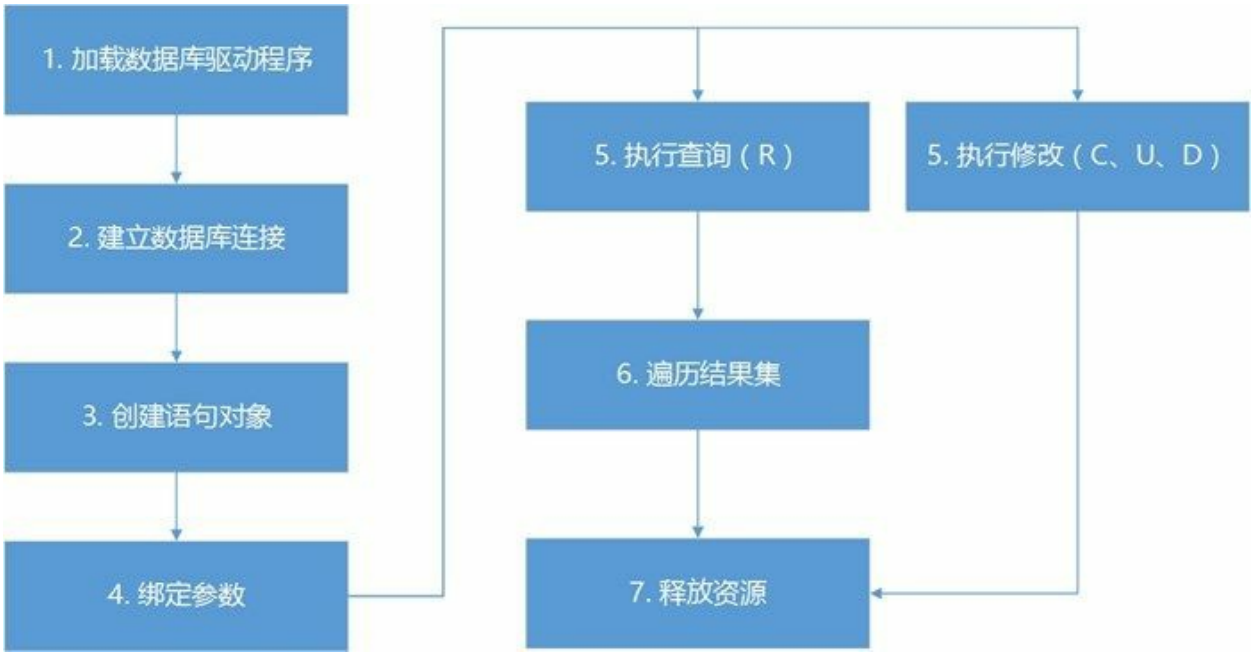


图**28-22** 数据库编程的一般过程

上述步骤是基本的一般步骤，实际情况会有所变化，例如没有参数需要绑定，则第4步骤就省略了。还有，如果Connection对象、Statement对象和ResultSet对象都采用自动资源管理技术释放资源，那么第7步骤也可以省略。

28.4.2 数据查询操作

为了介绍数据查询操作案例，这里准备了一个User表，它有两个字段name和userid，如表28-2所示。

表 **28-2** **User**表结构

字段名	类型	是否可以为 Null	主键
name	varchar(20)	是	否
userid	int	否	是

下面介绍实现如下两条SQL语句查询功能：

```
select name,userid from user where userid > ? order by userid    //有条件查询
select max(userid) from user                                     //使用max等函数，无条件查询
```

01. 有条件查询实现代码如下：

```
//CRUDSample.java文件
package com.a51work6;

import java.io.IOException;
import java.io.InputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Properties;

public class CRUDSample {

    // 连接数据库url
    static String url;
    // 创建Properties对象
    static Properties info = new Properties();

    // 1.驱动程序加载
    static {                                                    ①
        // 获得属性文件输入流
        InputStream input
                = CRUDSample.class.getClassLoader().getResourceAsStream("config.properties");

        try {
            // 加载属性文件内容到Properties对象
            info.load(input);
            // 从属性文件中取出url
            url = info.getProperty("url");
            // Class.forName("com.mysql.jdbc.Driver");
            // 从属性文件中取出driver
            String driverClassName = info.getProperty("driver");
            Class.forName(driverClassName);
            System.out.println("驱动程序加载成功...");
        } catch (ClassNotFoundException e) {
            System.out.println("驱动程序加载失败...");
        } catch (IOException e) {
            System.out.println("加载属性文件失败...");
        }
    }                                                           ②

    public static void main(String[] args) {

        // 查询数据
        read();
```

```
    }

    // 数据查询操作
    public static void read() {                ③

        Connection conn = null;
        PreparedStatement pstmt = null;
        ResultSet rs = null;

        try {
            // 2.创建数据库连接
            conn = DriverManager.getConnection(url, info);
            // 3. 创建语句对象
            pstmt = conn.prepareStatement("select name,userid from "
                    + "user where userid > ? order by userid");
            // 4. 绑定参数
            pstmt.setInt(1, 0);
            // 5. 执行查询（R）
            rs = pstmt.executeQuery();
            // 6. 遍历结果集
            while (rs.next()) {
                System.out.printf("id: %d    name: %s\n", rs.getInt(2), rs.getString("name"));
            }

        } catch (SQLException e) {
            e.printStackTrace();
        } finally {                    // 7.释放资源     ④
            if (rs != null) {
                try {
                    rs.close();
                } catch (SQLException e) {
                }
            }
            if (pstmt != null) {
                try {
                    pstmt.close();
                } catch (SQLException e) {
                }
            }
            if (conn != null) {
                try {
                    conn.close();
                } catch (SQLException e) {
                }
            }
        }                                          ⑤
    }

}
```

上述代码第①行~第②行是静态代码块，在静态代码块中读取属性文件内容到Properties对象和加载驱动程序，这两个操作只需执行一次，所以它们最好放到静态代码块中。另外，需要注意本例中将驱动程序类名和数据库连接的url字符串都放到属性文件中config.properties，这样一来更加方便配置，config.properties内容如下：

```
driver=com.mysql.jdbc.Driver
url=jdbc:mysql://localhost:3306/MyDB
user=root
password=123456
useSSL=false
verifyServerCertificate=false
```

上述代码第③行read()方法是数据查询方法，查询完成之后采用finally代码块释放资源，见代码第④行~第⑤行。本例也可以使用自动资源管理技术，但会引起try语句发生嵌套，反而会有些麻烦。

02. 无条件查询实现代码如下：

```
// 1.驱动程序加载
static {
    ...
}
...

// 查询最大的用户Id
public static int readMaxUserId() {

    int maxId = 0;
    try (
            // 2.创建数据库连接
            Connection conn = DriverManager.getConnection(url, info);
            // 3. 创建语句对象
            PreparedStatement pstmt = conn.prepareStatement("select max(userid) from user");
            // 4. 绑定参数
            // pstmt.setInt(1, 0);
            // 5. 执行查询（R）
            ResultSet rs = pstmt.executeQuery()) {
        // 6. 遍历结果集
        if (rs.next()) {
            maxId = rs.getInt(1);
        }

    } catch (SQLException e) {
        e.printStackTrace();
    }

    return maxId;
}
```

上述代码使用了自动资源管理技术，由于没有参数需要绑定，所以ResultSet对象可以与Connection对象和PreparedStatement对象放在一个try代码块中进行管理。而前面的有条件查询read()方法则不行。

28.4.3 数据修改操作

数据修改操作包括了：数据插入、数据更新和数据删除。

01. 数据插入

数据插入代码如下：

```
// 数据插入操作
public static void create() {

    try ( // 2.创建数据库连接
            Connection conn = DriverManager.getConnection(url, info);
            // 3. 创建语句对象
            PreparedStatement pstmt
                = conn.prepareStatement("insert into user (userid, name) values (?,?)")) { ①

        // 查询最大值
        int maxId = readMaxUserId();
```

```
        // 4. 绑定参数
        pstmt.setInt(1, ++maxId);                    ②
        pstmt.setString(2, "Tony" + maxId);          ③
        // 5. 执行修改（C、U、D）
        int affectedRows = pstmt.executeUpdate();    ④

        System.out.printf("成功插入%d条数据。\n", affectedRows);

    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

代码第①行是创建插入语句对象，其中有两个占位符。因此需要绑定参数，代码第②行是绑定第一个参数，代码第③行是绑定第二个参数。代码第④行executeUpdate()方法执行SQL语句，该方法与查询方法executeQuery()不同。executeUpdate()方法返回的是整数，成功影响的记录数，即成功插入记录数。

02. 数据更新

数据更新代码如下：

```
// 数据更新操作
public static void update() {

    try ( // 2.创建数据库连接
            Connection conn = DriverManager.getConnection(url, info);
            // 3. 创建语句对象
            PreparedStatement pstmt
                = conn.prepareStatement("update user set name = ? where userid > ?")) {

        // 4. 绑定参数
        pstmt.setString(1, "Tom");
        pstmt.setInt(2, 30);
        // 5. 执行修改（C、U、D）
        int affectedRows = pstmt.executeUpdate();

        System.out.printf("成功更新%d条数据。\n", affectedRows);

    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

03. 数据删除

数据删除代码如下：

```
// 数据删除操作
public static void delete() {

    try ( // 2.创建数据库连接
            Connection conn = DriverManager.getConnection(url, info);
            // 3. 创建语句对象
            PreparedStatement pstmt = conn.prepareStatement("delete from user where userid = ?")

        // 查询最大值
        int maxId = readMaxUserId();

        // 4. 绑定参数
        pstmt.setInt(1, maxId);
```

```
        // 5. 执行修改（C、U、D）
        int affectedRows = pstmt.executeUpdate();

        System.out.printf("成功删除%d条数据。\n", affectedRows);

    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

数据更新、数据删除与数据插入程序结构上非常类似，差别主要在于SQL语句的不同，绑定参数的不同。具体代码不再解释。

本章小结

本章首先介绍MySQL数据库的安装、配置和日常的管理命令，然后重点讲解了JDBC数据库编程技术。读者需要重点掌握三个主要接口，熟悉一般数据库编程过程。

第 29 章　项目实战1：开发PetStore宠物商店项目

前面学习的Java知识只有通过项目贯穿起来，才能将书本中知识变成自己的。通过项目实战读者能够了解软件开发流程，了解所学知识在实际项目中使用的情况，哪些是重点的，哪些是了解的。

本章介绍Java SE技术实现的PetStore宠物商店项目，所涉及的知识点：Java面向对象、Lambda表达式、Java Swing技术、JDBC技术和数据库相关等知识，其中还会用到方方面面的Java基础知识。

29.1 系统分析与设计

本节对PetStore宠物商店项目进行分析和设计，其中设计过程包括原型设计、数据库设计、架构设计和系统设计。

29.1.1 项目概述

PetStore是Sun（现在Oracle）公司为了演示自己的Java EE技术，而编写的一个基于Web宠物店项目，如图29-1所示为项目启动页面，项目介绍网站是http://www.oracle.com/technetwork/java/index-136650.html。

PetStore是典型的电子商务项目，是现在很多电商平台的雏形。技术方面主要是Java EE技术，用户界面采用Java Web介绍实现。但本书介绍Java SE技术，不介绍Java Web，所以本章的PetStore项目用户界面采用Java Swing技术实现。

图**29-1 PetStore**项目启动页面

29.1.2 需求分析

PetStore宠物商店项目主要功能如下：

- 用户登录
- 查询商品
- 添加商品到购物车

- 查看购物车
- 下订单
- 查看订单

采用用例分析方法描述用例图如图29-2所示。

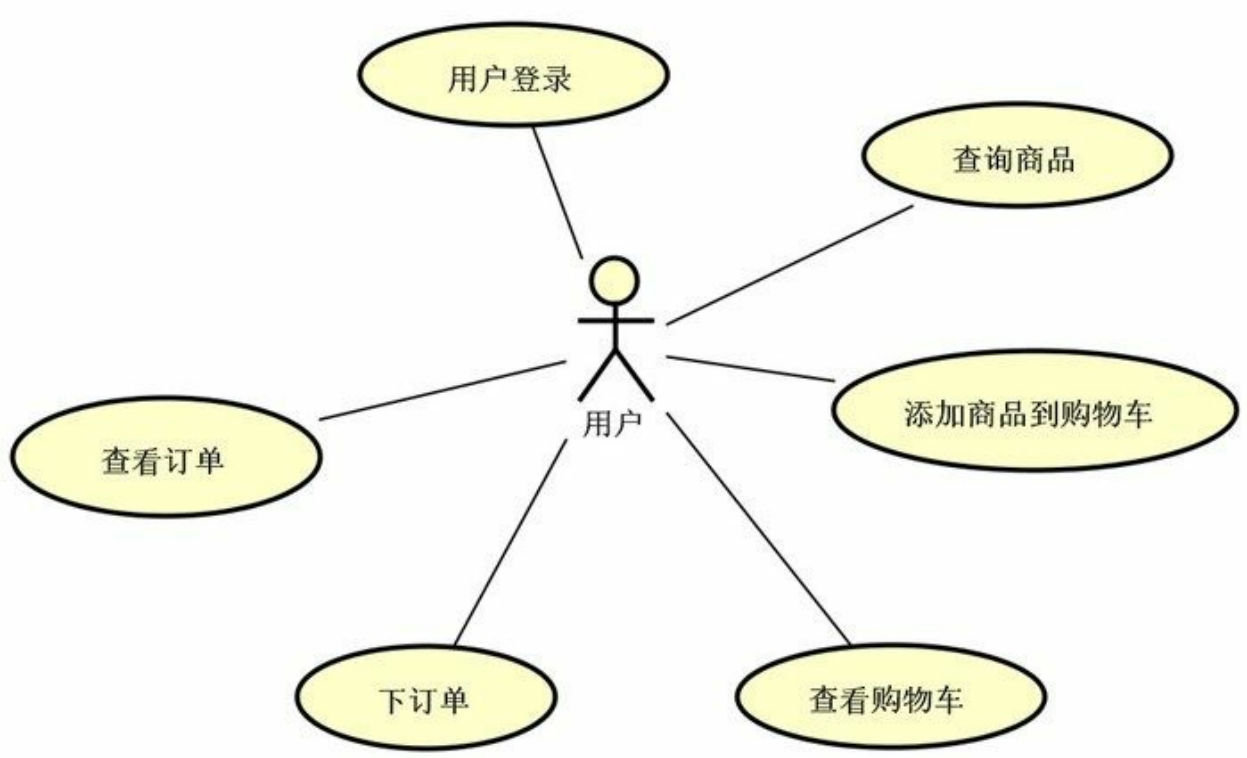


图**29-2　PetStore**宠物商店用例图

29.1.3　原型设计

原型设计草图对于开发人员、设计人员、测试人员、UI设计人员以及用户都是非常重要的。PetStore宠物商店项目原型设计图如图29-3所示。

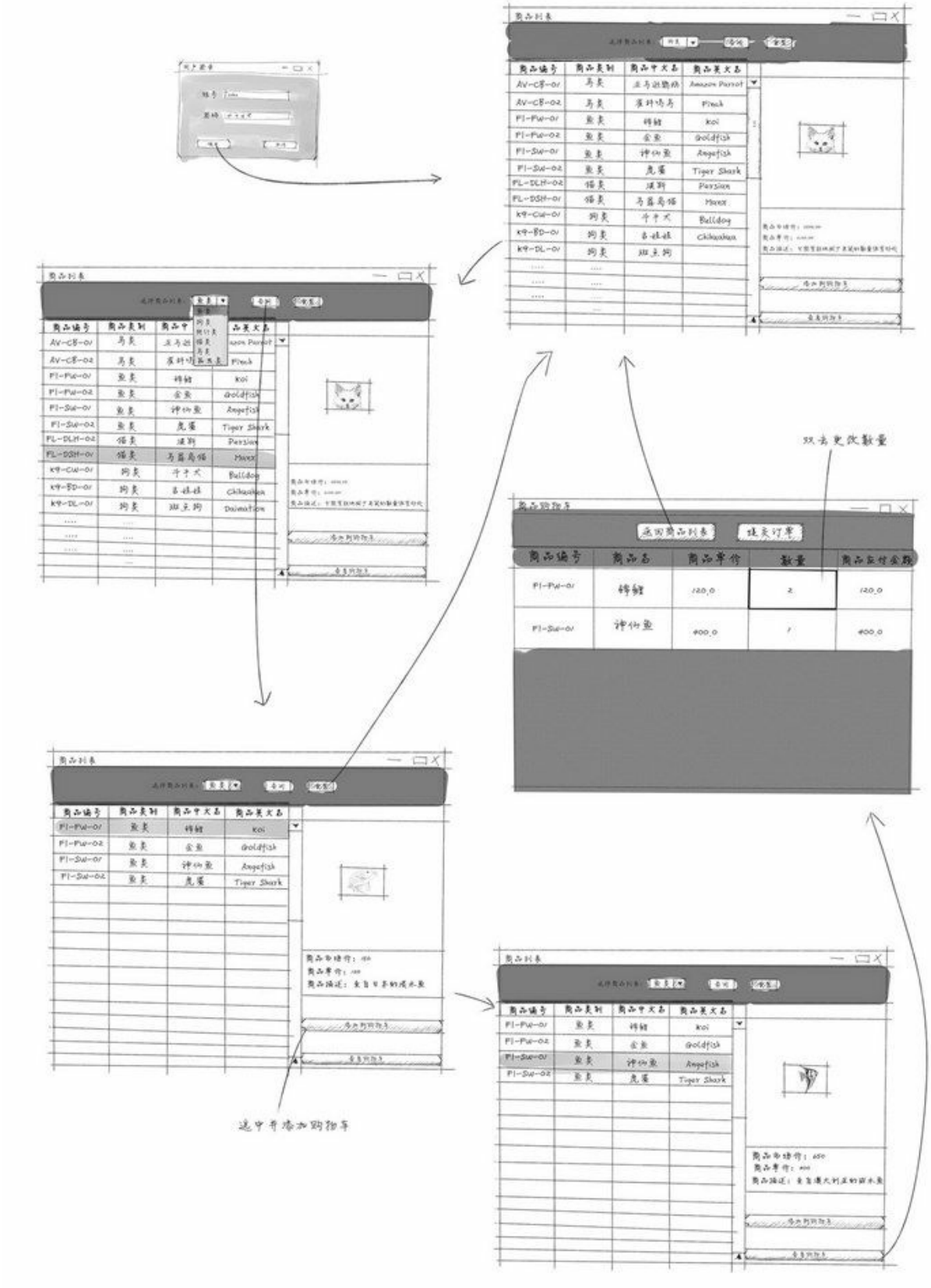

双击更改数量
选中并添加购物车

图**29-3**　**PetStore**宠物商店项目原型设计图

29.1.4　数据库设计

Java官方提供的PetStore宠物商店项目数据库设计比较复杂，根据如图29-2的用例图重新设计数据库，数据库设计模型如图29-4所示。

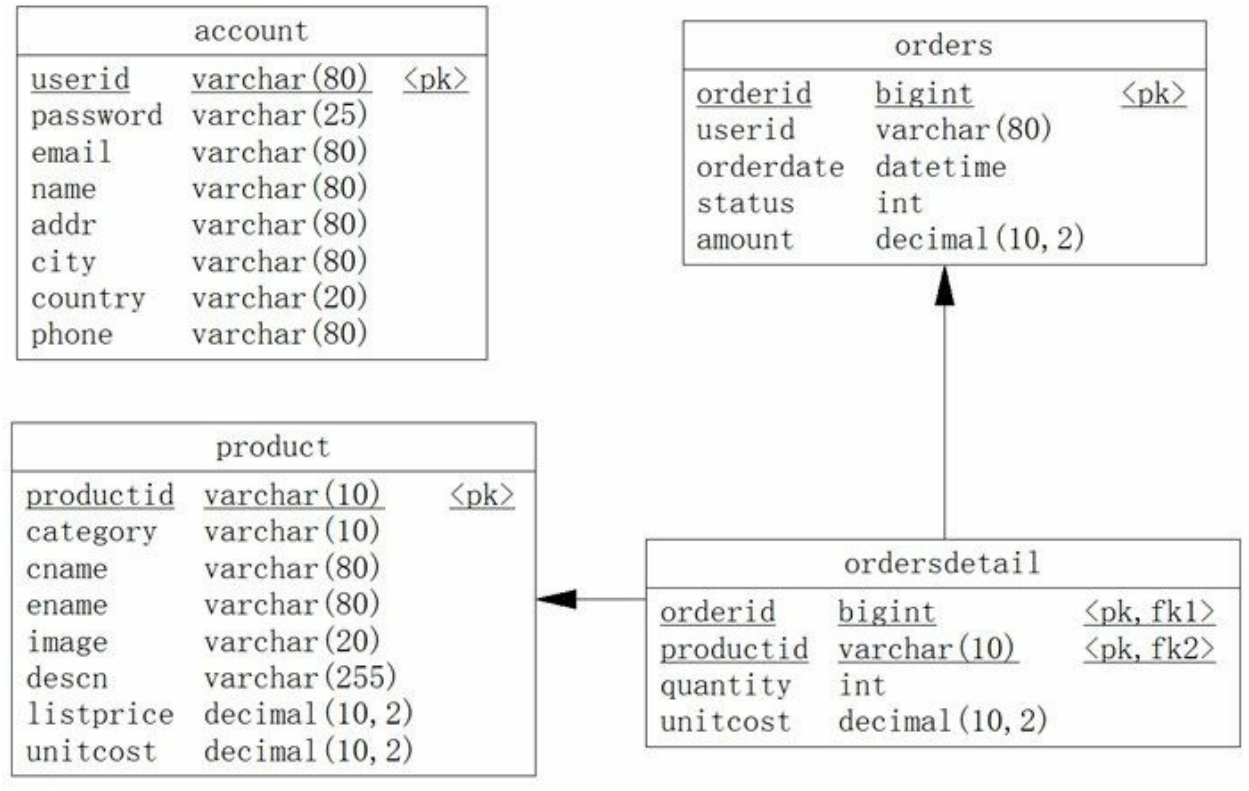


图**29-4**　数据库设计模型

数据库设计模型中各个表说明如下：

01. 用户表

    用户表（英文名account）是PetStore宠物商店的注册用户，用户Id（英文名userid）是主键，用户表结构如表29-1所示。

    表 **29-1**　用户表

字段名	数据类型	长度	精度	主键	外键	备注
userid	varchar(80)	80	-	是	否	用户 Id
password	varchar(25)	25	-	否	否	用户密码
email	varchar(80)	80	-	否	否	用户 Email
name	varchar(80)	80	-	否	否	用户名
addr	varchar(80)	80	-	否	否	地址
city	varchar(80)	80	-	否	否	所在城市
country	varchar(20)	20	-	否	否	国家
phone	varchar(80)	80	-	否	否	电话号码

02. 商品表

商品表（英文名product）是PetStore宠物商店所销售的商品（宠物），商品Id（英文名productid）是主键，商品表结构如表29-2所示。

表 **29-2**　商品表

字段名	数据类型	长度	精度	主键	外键	备注
productid	varchar(10)	10	-	是	否	商品 Id
category	varchar(10)	10	-	否	否	商品类别
cname	varchar(80)	80	-	否	否	商品中文名
ename	varchar(80)	80	-	否	否	商品英文名
image	varchar(20)	20	-	否	否	商品图片
descn	varchar(255)	255	-	否	否	商品描述
listprice	decimal(10,2)	10	2	否	否	商品市场价
unitcost	decimal(10,2)	10	2	否	否	商品单价

03. 订单表

订单表（英文名orders）记录了用户每次购买商品所生成的订单信息，订单Id（英文名orderid）是主键，订单表结构如表29-3所示。

表 **29-3**　订单表

字段名	数据类型	长度	精度	主键	外键	备注
orderid	bigint		-	是	否	订单 Id
userid	varchar(80)	80	-	否	否	下订单的用户 Id
orderdate	datetime		-	否	否	下订单时间
status	int		-	否	否	订单付款状态　0 待付款 1 已付款
amount	decimal(10,2)	10	2	否	否	订单应付金额

04. 订单明细表

订单表中不能描述其中有哪些商品，购买商品的数量，购买时商品的单价等信息，这些信息被记录在订单详细表中。订单明细表（英文名ordersdetail），记录了某个订单更加详细的详细，订单明细表主键是由orderid和productid两个字段联合而成，是一种联合主键，订单明细表结构如表29-4所示。

表 **29-4**　订单明细表

字段名	数据类型	长度	精度	主键	外键	备注
orderid	bigint		-	是	是	订单 Id
productid	varchar(10)	10	-	是	是	商品 Id
quantity	int		-	否	否	商品数量
unitcost	decimal(10,2)	10	2	否	否	商品单价

从图29-4所示的数据库设计模型中可以编写DDL（数据定义）语句[1]，使用这些语句可以很方便地创建和维护数据库中的表结构。

[1]数据定义语句，用于创建、删除和修改数据库对象，包括DROP、CREATE、ALTER、GRANT、REVOKE和TRUNCATE等语句。

29.1.5　架构设计

无论是庞大企业级系统，还是手机上的应用，都应该有效地组织程序代码。这就需要设计。而架构设计就是系统的“骨架”，它是源自于前人经验的总结和提炼，形式一种模式推而广之。但是遗憾的是本书的定位是初学者，并不是介绍架构设计方面的书。为了开发PetStore宠物商店项目需要，这里笔者给出最简单的架构设计结果。

世界著名软件设计大师Martin Fowler在他《企业应用架构模式》（英文名Patterns of Enterprise Application Architecture）一书中提到，为了有效地组织代码，一个系统应该分为三个基本层，如图29-5所示。“层”（Layer）是相似功能的类和接口的集合，“层”之间是松耦合的，“层”的内部是高内聚的。

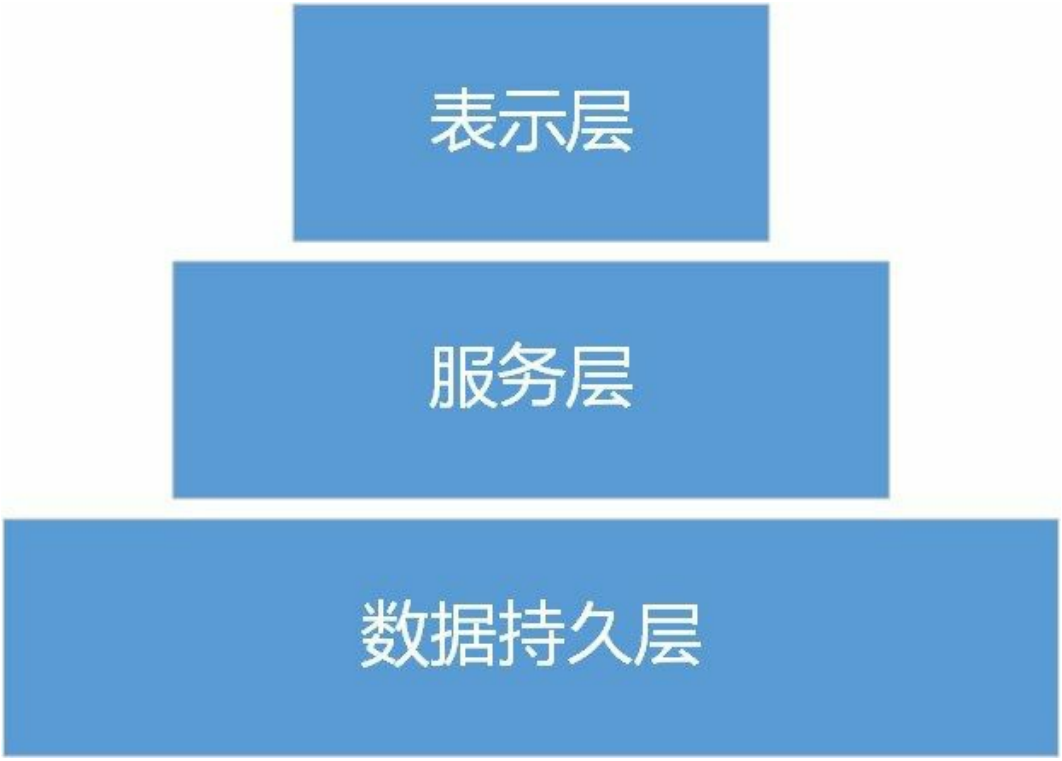


图**29-5　Martin Fowler**分层架构设计

- 表示层：用户与系统交互的组件集合。用户通过这一层向系统提交请求或发出指令，系统通过这一层接收用户请求或指令，待指令消化吸收后再调用下一层，接着将调用结果展现到这一层。表示层应该是轻薄的，不应该具有业务逻辑。

- 服务层。系统的核心业务处理层。负责接收表示层的指令和数据，待指令和数据消化吸收后，再进行组织业务逻辑的处理，并将结果返回给表示层。

- 数据持久层。数据持久层用于访问持久化数据，持久化数据可以是保存在数据库、文件、其他系统或者网络数据。根据不同的数据来源，数据持久层会采用不同的技术，例如：如果数据保存到数据库中，则使用JDBC技术；如果数据保存JSON文件在，则需要I/O流和JSON解码技术实现。

Martin Fowler分层架构设计看起来像一个多层“蛋糕”，蛋糕师们在制作多层“蛋糕”的时候先做下层再做上层，最后做顶层。没有下层就没有上层，这叫作“上层依赖于下层”。为了降低松耦度，层之间还需要定义接口，通过接口隔离实现细节，上层调用者用只关心接口，不关心下一层的实现细节。

Martin Fowler分层架构是基本形式，在具体实现项目设计时，可能有所增加，也可能有所减少。本章实现的PetStore宠物商店项目，由于简化了需求，逻辑比较简单，可以不需要服务层，表示层可以直接访问数据持久层，如图29-6所示，表示层采用Swing技术实现，数据持久层采用JDBC技术实现。

图**29-6　PetStore**宠物商店项目架构设计

29.1.6　系统设计

系统设计是在具体架构下的设计实现，PetStore宠物商店项目主要分为表示层和数据数据持久层。下面分别介绍一下它们的具体实现。

01. 数据持久层设计

    数据持久层在具体实现时，会采用DAO（数据访问对象）设计模式，数据库中每一个数据表，对应一个DAO对象，每一个DAO对象中有访问数据表的CRUD[2]四类操作。

    如图29-7所示PetStore宠物商店项目的数据持久层类图，首先定义了4个DAO接口，这4个接口对应数据中4个表，接口定义的方法是对数据库表的CRUD操作。

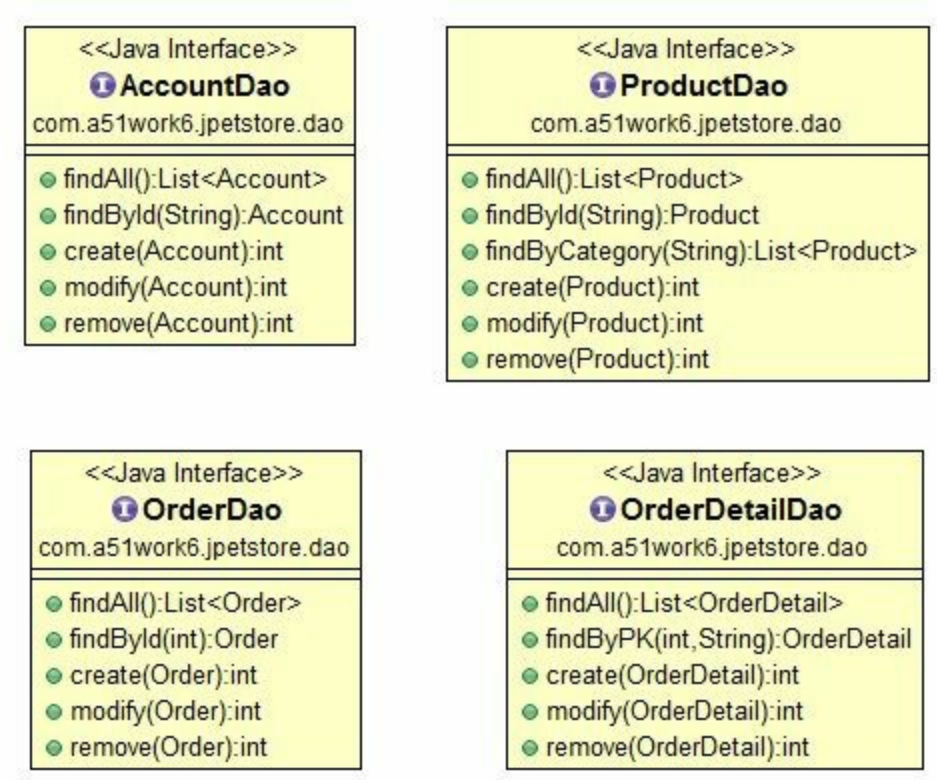


图**29-7　PetStore**宠物商店项目数据持久层类图

02. 表示层

主要使用Swing技术，每一个界面就是一个窗口对象。在表示层中各个窗口是依据原型设计而来的。PetStore宠物商店项目表示层类如图29-8所示，其中有三个窗口类，LoginFrame用户登录窗口、CartFrame购物车窗口和ProductListFrame商品列表窗口，它们有共同的父类MyFrame，MyFrame类是根据自己的项目情况进行的封装，从类图中可见CartFrame与ProductListFrame具有关联关系，CartFrame包含一个对ProductListFrame的引用。

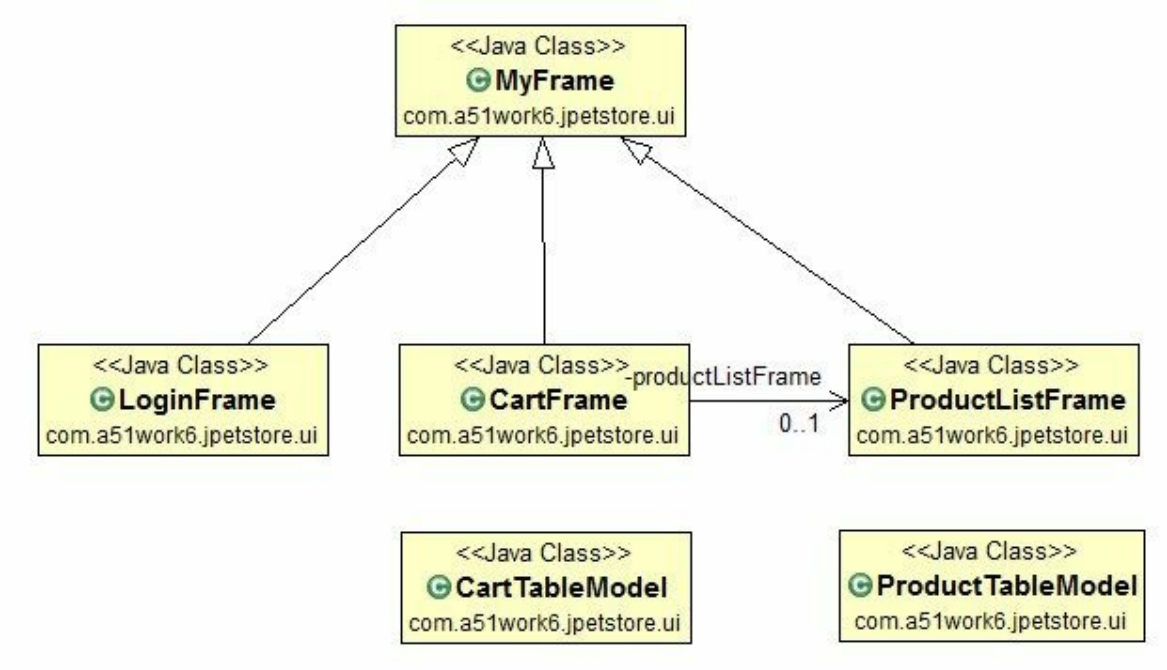


图**29-8　PetStore**宠物商店项目表示层类图

另外，CartFrame与ProductListFrame会使用到表格，所以自定义了两个表模型CartTableModel和ProductTableModel。

[2]CRUD方法是访问数据的4个方法，即增加、删除、修改和查询：C为Create，表示增加数据；R是Read，表示查询数据；U是Update，表示修改数据；D是Delete，表示删除数据。

29.2 任务1：创建数据库

在设计完成之后，在编写Java代码之前，应该创建数据库。

29.2.1 迭代1.1：安装和配置MySQL数据库

首先应该为开发该项目，准备好数据库。本书推荐使用MySQL数据库，如果没有安装MySQL数据库，可以参考28.2.1节安装MySQL数据库。

29.2.2 迭代1.2：编写数据库DDL脚本

按照图29-4所示的数据库设计模型编写数据库DDL脚本。当然，也可以通过一些工具生成DDL脚本，然后把这个脚本放在数据库中执行就可以了。下面是编写的DDL脚本：

```
/* 创建数据库 */
CREATE DATABASE  IF NOT EXISTS  petstore;

use petstore;

/* 用户表 */
CREATE TABLE IF NOT EXISTS account (
    userid varchar(80) not null,        /* 用户Id  */
    password varchar(25)  not null,     /* 用户密码 */
    email varchar(80) not null,         /* 用户Email */
    name varchar(80) not null,          /* 用户名 */
    addr varchar(80) not null,          /* 地址 */
    city varchar(80) not  null,         /*  所在城市 */
    country varchar(20) not null,       /*  国家 */
    phone varchar(80) not null,         /*  电话号码 */
PRIMARY KEY (userid));

/* 商品表 */
CREATE TABLE IF NOT EXISTS product (
    productid varchar(10) not null,    /* 商品Id */
    category varchar(10) not null,     /* 商品类别 */
    cname varchar(80) null,            /* 商品中文名 */
    ename varchar(80) null,            /* 商品英文名 */
    image varchar(20) null,            /* 商品图片 */
    descn varchar(255) null,           /* 商品描述 */
    listprice decimal(10,2) null,      /* 商品市场价 */
    unitcost decimal(10,2) null,       /* 商品单价 */
PRIMARY KEY (productid));

/* 订单表 */
CREATE TABLE IF NOT EXISTS orders (
    orderid bigint not null,           /* 订单Id */
    userid varchar(80) not null,       /* 下订单的用户Id */
    orderdate datetime not null,       /* 下订单时间 */
    status int not null default 0,     /* 订单付款状态  0待付款  1已付款 */
    amount decimal(10,2) not null,     /* 订单应付金额 */
PRIMARY KEY (orderid));

/* 订单明细表 */
CREATE TABLE IF NOT EXISTS ordersdetail (
    orderid bigint not null,           /* 订单Id */
    productid varchar(10) not null,    /* 商品Id */
    quantity int not null,             /* 商品数量 */
    unitcost decimal(10,2) null,       /* 商品单价 */
PRIMARY KEY (orderid, productid));
```

如果读者对于编写DDL脚本不熟悉，可以直接使用笔者编写好的jpetstore-mysql-schema-gbk.sql脚本文件，文件位于PetStore项目下db目录中。

29.2.3 迭代1.3：插入初始数据到数据库

PetStore宠物商店项目有一些初始的数据，这些初始数据在创建数据库之后插入。这些插入数据的语句如下：

```
use petstore;

/* 用户表数据 */
INSERT INTO account VALUES('j2ee','j2ee','yourname@yourdomain.com','关东升', '北京丰台区', '北京',
INSERT INTO account VALUES('ACID','ACID','acid@yourdomain.com','Tony', '901 San Antonio Road', 'Pa

/* 商品表数据 */
INSERT INTO product VALUES ('FI-SW-01','鱼类','神仙鱼', 'Angelfish', 'fish1.jpg', '来自澳大利亚的咸水
INSERT INTO product VALUES ('FI-SW-02','鱼类','虎鲨', 'Tiger Shark','fish4.gif', '来自澳大利亚的咸水
...
INSERT INTO product VALUES ('AV-CB-01','鸟类','亚马逊鹦鹉', 'Amazon Parrot','bird4.gif', '寿命长达75
INSERT INTO product VALUES ('AV-SB-02','鸟类','雀科鸣鸟', 'Finch','bird1.gif', '会唱歌的鸟儿', 150,
```

如果读者不愿意自己编写插入数据的脚本文件，可以直接使用笔者编写好的jpetstore-mysql-dataload-gbk.sql脚本文件，文件位于PetStore项目下db目录中。

29.3 任务2：初始化项目

本项目推荐使用Eclipse工具，所以首先参考3.1节创建一个Eclipse项目，项目名称PetStore。

29.3.1 任务2.1：配置项目构建路径

PetStore项目创建完成后，需要参考如图29-9，在PetStore项目根目录下面创建普通文件夹db。然后将MySQL数据库JDBC驱动程序mysql-connector-java-5.xxx-bin.jar拷贝到db目录，参考28.3.1节将驱动程序文件添加到项目的构建路径中。images文件夹中内容是项目使用的图片。

图29-9 PetStore项目目录结构

29.3.2 任务2.2：添加资源图片

项目中会用到很多资源图片，为了打包发布项目方便，这些图片最好放到src源文件夹下，Eclipse会将该文件夹下有文件一起复制到字节码文件夹中。参考图29-9在src文件夹下创建images文件夹，然后将本书配套资源中找到images中的图片，并复制到Eclipse项目的images文件夹中。

29.3.3 任务2.3：添加包

参考图29-9在src文件夹中创建如下4个包：

- com.a51work6.jpetstore.ui。放置表示层组件。
- com.a51work6.jpetstore.domain。放置实体类。
- com.a51work6.jpetstore.dao。放置数据持久层组件中DAO接口。
- com.a51work6.jpetstore.dao.mysql。放置数据持久层组件中DAO接口具体实现类，mysql说明是MySQL数据库DAO对象。该包中还放置了访问MySQL数据库一些辅助类和配置文件。

29.4 任务3：编写数据持久层代码

Eclipse项目创建并初始化完成后，可以先编写数据持久层代码。

29.4.1 任务3.1：编写实体类

无论是数据库设计还是面向对象的架构设计都会“实体”，“实体”是系统中的“人”、“事”、“物”等名词，如用户、商品、订单和订单明细等。在数据库设计时它将演变为表，如用户表（account）、商品表（product）、订单表（orders）和订单明细表（ordersdetail），在面向对象的架构设计时，实体将演变为“实体类”，如图29-10所示是PetStore宠物商店项目中的实体类，实体类属性与数据库表字段在是相似的，事实上它们描述的同一个事物，当然具有相同的属性，只是它们分别采用不同设计理念，实体类采用对象模型，表采用关系模式。

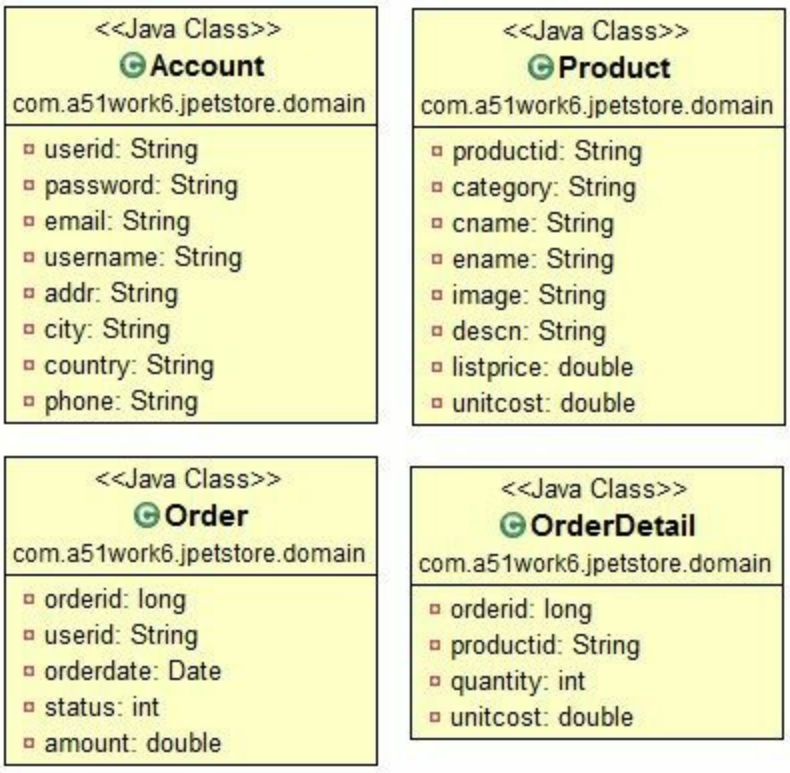


图29-10 PetStore宠物商店项目实体类图

订单明细实体类OrderDetail的代码如下：

```
//OrderDetail.java文件
package com.a51work6.jpetstore.domain;

//订单明细
public class OrderDetail {

    private long orderid;          // 订单Id
    private String productid;      // 商品Id
    private int quantity;          // 商品数量
    private double unitcost;       // 单价

    public long getOrderid() {
```

```
        return orderid;
    }

    public void setOrderid(long orderid) {
        this.orderid = orderid;
    }

    public double getUnitcost() {
        return unitcost;
    }

    public void setUnitcost(double unitcost) {
        this.unitcost = unitcost;
    }

    public String getProductid() {
        return productid;
    }

    public void setProductid(String productid) {
        this.productid = productid;
    }

    public int getQuantity() {
        return quantity;
    }

    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }

}
```

从上述代码中可见实体类结构很简单，主要是一个私有属性，以及对这些属性方法的公有Getter和Setter方法。在使用Eclipse编程时只需要编写那些私有属性即可，然后通过Eclipse工具生成Getter和Setter方法，具体步骤参考25.6节。

订单实体类Order的代码如下：

```
//Account.java文件
package com.a51work6.jpetstore.domain;
//Order.java文件
package com.a51work6.jpetstore.domain;

import java.util.Date;

public class Order {

    private long orderid;       // 订单Id
    private String userid;      // 下订单的用户Id
    private Date orderdate;     // 下订单时间
    private int status;         // 订单付款状态 0待付款 1已付款
    private double amount;      // 订单应付金额

    //省略Setter和Getter方法

}
```

用户实体类Account的代码如下：

```
//Account.java文件
```

```
package com.a51work6.jpetstore.domain;

public class Account {

    /* 私有成员变量 */
    private String userid;      // 用户Id
    private String password;    // 用户密码
    private String email;       // 用户Email
    private String username;    // 用户名
    private String addr;        // 地址
    private String city;        // 所在城市
    private String country;     // 国家
    private String phone;       // 电话号码

    //省略Setter和Getter方法

}
```

商品实体类Product的代码如下：

```
//Product.java文件
package com.a51work6.jpetstore.domain;

public class Product {

    private String productid;    // 商品Id
    private String category;     // 商品类别
    private String cname;        // 商品中文名
    private String ename;        // 商品英文名
    private String image;        // 商品图片
    private String descn;        // 商品描述
    private double listprice;    // 商品市场价
    private double unitcost;     // 商品单价

    //省略Setter和Getter方法

}
```

29.4.2 迭代3.2：编写DAO类

编写DAO类就没有实体类那么简单了，数据持久层开发的主要工作量主要是DAO类。图29-11是DAO实现类图。

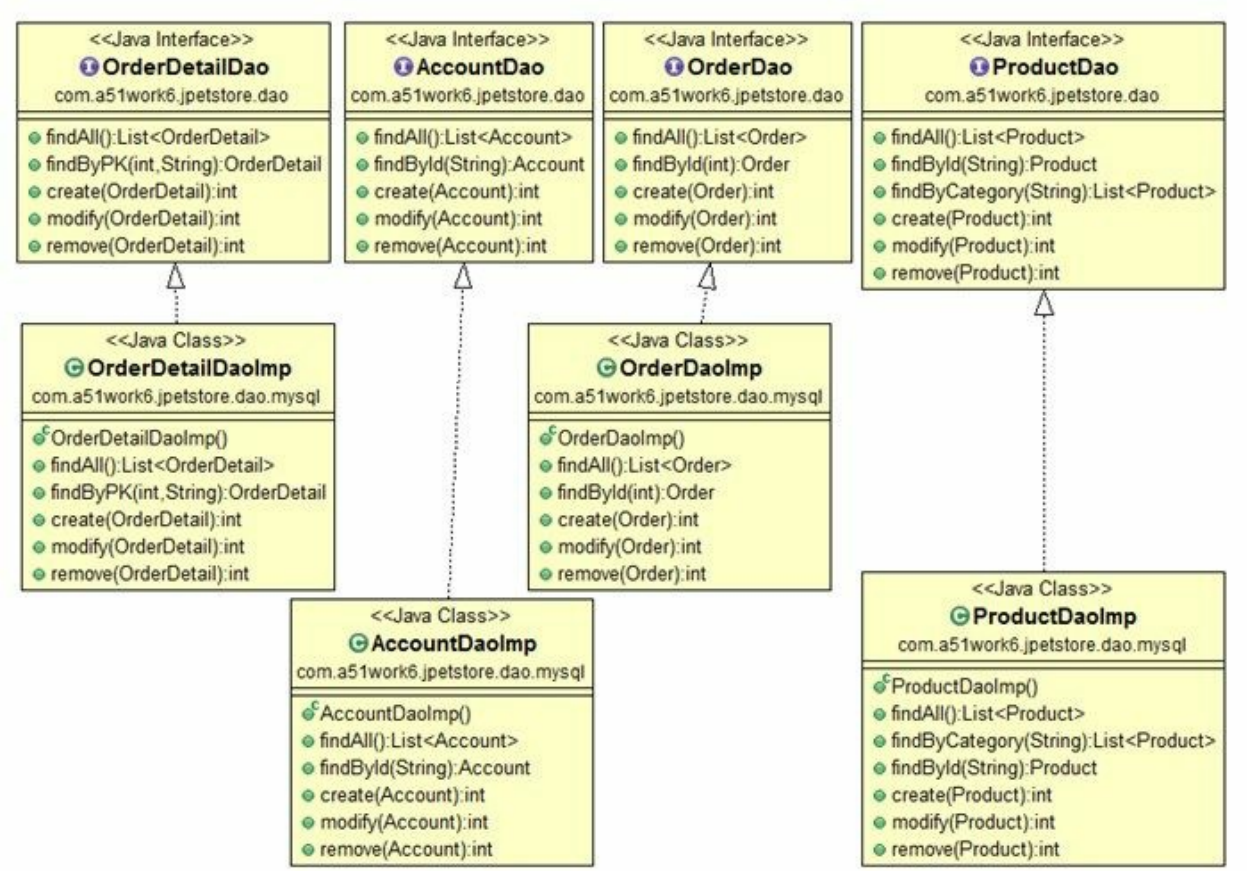


图**29-11　DAO**实现类图

01. 用户管理DAO

用户管理AccountDao实现类AccountDaoImp代码如下：

```
//AccountDaoImp.java文件
package com.a51work6.jpetstore.dao.mysql;
...
public class AccountDaoImp implements AccountDao {

    @Override
    public List<Account> findAll() {
        // TODO 自动生成的方法存根
        return null;
    }

    @Override
    public Account findById(String userid) {

        Connection conn = null;
        PreparedStatement pstmt = null;
        ResultSet rs = null;
        Account account = null;
        try {
            // 2.创建数据库连接
            conn = DBHelper.getConnection();
            // 3. 创建语句对象
            String sql = "select userid,password,email,name,addr,city,country,phone"
```

```
                    + " from account where userid = ?";

            pstmt = conn.prepareStatement(sql);
            // 4. 绑定参数
            pstmt.setString(1, userid);
            // 5. 执行查询（R）
            rs = pstmt.executeQuery();
            // 6. 遍历结果集
            if (rs.next()) {
                account = new Account();

                account.setUserid(rs.getString("userid"));
                account.setPassword(rs.getString("password"));
                account.setEmail(rs.getString("email"));
                account.setUsername(rs.getString("name"));
                account.setAddr(rs.getString("addr"));
                account.setUserid(rs.getString("userid"));
                account.setCity(rs.getString("city"));
                account.setCountry(rs.getString("country"));
                account.setPhone(rs.getString("phone"));

                return account;
            }

        } catch (SQLException e) {
            e.printStackTrace();
        } finally { // 释放资源
            if (rs != null) {
                try {
                    rs.close();
                } catch (SQLException e) {
                }
            }
            if (pstmt != null) {
                try {
                    pstmt.close();
                } catch (SQLException e) {
                }
            }
            if (conn != null) {
                try {
                    conn.close();
                } catch (SQLException e) {
                }
            }
        }

        return null;
    }

    @Override
    public int create(Account account) {
        // TODO 自动生成的方法存根
        return 0;
    }

    @Override
    public int modify(Account account) {
        // TODO 自动生成的方法存根
        return 0;
    }

    @Override
    public int remove(Account account) {
        // TODO 自动生成的方法存根
        return 0;
```

```
        }

    }
```

AccountDao接口中定义了5个抽象方法。但这些方法，在本项目中只需要实现findById()方法。具体代码不再赘述。

02. 商品管理DAO

商品管理ProductDao实现类ProductDaoImp代码如下：

```
    //ProductDaoImp.java文件
    package com.a51work6.jpetstore.dao.mysql;
    ...
    //商品管理DAO
    public class ProductDaoImp implements ProductDao {

        @Override
        public List<Product> findAll() {

            String sql = "select productid,category,cname,ename,image,"
                    + "listprice,unitcost,descn from product";

            List<Product> products = new ArrayList<Product>();

            try (
                    // 2.创建数据库连接
                    Connection conn = DBHelper.getConnection();
                    // 3. 创建语句对象
                    PreparedStatement pstmt = conn.prepareStatement(sql);
                    // 4. 绑定参数
                    // 5. 执行查询（R）
                    ResultSet rs = pstmt.executeQuery()) {

                // 6. 遍历结果集
                while (rs.next()) {
                    Product p = new Product();
                    p.setProductid(rs.getString("productid"));
                    p.setCategory(rs.getString("category"));
                    p.setCname(rs.getString("cname"));
                    p.setEname(rs.getString("ename"));
                    p.setImage(rs.getString("image"));
                    p.setListprice(rs.getDouble("listprice"));
                    p.setUnitcost(rs.getDouble("unitcost"));
                    p.setDescn(rs.getString("descn"));

                    products.add(p);
                }

            } catch (SQLException e) {
                e.printStackTrace();
            }
            return products;
        }

        @Override
        public List<Product> findByCategory(String category) {

            Connection conn = null;
            PreparedStatement pstmt = null;
            ResultSet rs = null;
            List<Product> products = new ArrayList<Product>();
```

```
        try {
            // 2.创建数据库连接
            conn = DBHelper.getConnection();
            // 3. 创建语句对象
            String sql = "select productid,category,cname,ename,image,listprice,unitcost,de
                    + "from product where category=?";

            pstmt = conn.prepareStatement(sql);
            // 4. 绑定参数
            pstmt.setString(1, category);
            // 5. 执行查询（R）
            rs = pstmt.executeQuery();
            // 6. 遍历结果集
            while (rs.next()) {
                Product p = new Product();
                p.setProductid(rs.getString("productid"));
                p.setCategory(rs.getString("category"));
                p.setCname(rs.getString("cname"));
                p.setEname(rs.getString("ename"));
                p.setImage(rs.getString("image"));
                p.setListprice(rs.getDouble("listprice"));
                p.setUnitcost(rs.getDouble("unitcost"));
                p.setDescn(rs.getString("descn"));

                products.add(p);
            }

        } catch (SQLException e) {
            e.printStackTrace();
        } finally { // 释放资源
            if (rs != null) {
                try {
                    rs.close();
                } catch (SQLException e) {
                }
            }
            if (pstmt != null) {
                try {
                    pstmt.close();
                } catch (SQLException e) {
                }
            }
            if (conn != null) {
                try {
                    conn.close();
                } catch (SQLException e) {
                }
            }
        }

        return products;
    }

    @Override
    public Product findById(String productid) {

        Connection conn = null;
        PreparedStatement pstmt = null;
        ResultSet rs = null;

        try {
            // 2.创建数据库连接
            conn = DBHelper.getConnection();
            // 3. 创建语句对象
            String sql = "select productid,category,cname,ename,image,listprice,unitcost,de
                    + "from product where productid=?";
```

```
            pstmt = conn.prepareStatement(sql);
            // 4. 绑定参数
            pstmt.setString(1, productid);
            // 5. 执行查询（R）
            rs = pstmt.executeQuery();
            // 6. 遍历结果集
            if (rs.next()) {

                Product p = new Product();
                p.setProductid(rs.getString("productid"));
                p.setCategory(rs.getString("category"));
                p.setCname(rs.getString("cname"));
                p.setEname(rs.getString("ename"));
                p.setImage(rs.getString("image"));
                p.setListprice(rs.getDouble("listprice"));
                p.setUnitcost(rs.getDouble("unitcost"));
                p.setDescn(rs.getString("descn"));

                return p;
            }

        } catch (SQLException e) {
            e.printStackTrace();
        } finally { // 释放资源
            if (rs != null) {
                try {
                    rs.close();
                } catch (SQLException e) {
                }
            }
            if (pstmt != null) {
                try {
                    pstmt.close();
                } catch (SQLException e) {
                }
            }
            if (conn != null) {
                try {
                    conn.close();
                } catch (SQLException e) {
                }
            }
        }

        return null;
    }

    @Override
    public int create(Product product) {
        // TODO 自动生成的方法存根
        return 0;
    }

    @Override
    public int modify(Product product) {
        // TODO 自动生成的方法存根
        return 0;
    }

    @Override
    public int remove(Product product) {
        // TODO 自动生成的方法存根
        return 0;
    }
```

```
}
```

ProductDao接口中定义了6个抽象方法。但这些方法，在本项目中只需要实现findById()、findAll()、findByCategory()和findById()方法。

03. 订单管理DAO

订单管理OrderDao实现类OrderDaoImp代码如下：

```
//OrderDaoImp.java文件
package com.a51work6.jpetstore.dao.mysql;
...
//订单管理DAO
public class OrderDaoImp implements OrderDao {

    @Override
    public List<Order> findAll() {

        String sql = "select orderid,userid,orderdate from product";
        List<Order> orderList = new ArrayList<Order>();

        try (
                // 2.创建数据库连接
                Connection conn = DBHelper.getConnection();
                // 3. 创建语句对象
                PreparedStatement pstmt = conn.prepareStatement(sql);
                // 4. 绑定参数
                // 5. 执行查询（R）
                ResultSet rs = pstmt.executeQuery()) {

            // 6. 遍历结果集
            while (rs.next()) {
                Order order = new Order();
                order.setOrderid(rs.getInt("orderid"));
                order.setUserid(rs.getString("userid"));
                order.setOrderdate(rs.getDate("orderdate"));

                orderList.add(order);
            }

        } catch (SQLException e) {
            e.printStackTrace();
        }
        return orderList;
    }

    @Override
    public Order findById(int orderid) {
        // TODO 自动生成的方法存根
        return null;
    }

    @Override
    public int create(Order order) {

        try ( // 2.创建数据库连接
                Connection conn = DBHelper.getConnection();
                // 3. 创建语句对象
                PreparedStatement pstmt = conn.prepareStatement(
                        "insert into orders (orderid,userid,orderdate,status,amount)"
                        + "values (?,?,?,?,?)")) {

            // 4. 绑定参数
```

```
                pstmt.setLong(1, order.getOrderid());
                pstmt.setString(2, order.getUserid());
                java.util.Date date = order.getOrderdate();
                //pstmt.setDate(3, new java.sql.Date(date.getTime()));
                pstmt.setTimestamp(3, new java.sql.Timestamp(date.getTime()));
                pstmt.setInt(4, order.getStatus());
                pstmt.setDouble(5, order.getAmount());

                // 5. 执行修改（C、U、D）
                int affectedRows = pstmt.executeUpdate();
                System.out.printf("成功插入%d条数据。\n", affectedRows);

            } catch (SQLException e) {
                return -1;
            }

            return 0;
        }

        @Override
        public int modify(Order order) {
            // TODO 自动生成的方法存根
            return 0;
        }

        @Override
        public int remove(Order order) {
            // TODO 自动生成的方法存根
            return 0;
        }

    }
```

OrderDao接口中定义了5个抽象方法。但这些方法，在本项目中只需要实现findAll()和create()方法。

04. 订单明细管理DAO

订单明细管理OrderDetailDao实现类OrderDetailDaoImp代码如下：

```
    //OrderDetailDaoImp.java文件
    package com.a51work6.jpetstore.dao.mysql;
    ...
    //订单明细管理DAO
    public class OrderDetailDaoImp implements OrderDetailDao {

        @Override
        public List<OrderDetail> findAll() {
            // TODO 自动生成的方法存根
            return null;
        }

        @Override
        public OrderDetail findByPK(int orderid, String productid) {

            Connection conn = null;
            PreparedStatement pstmt = null;
            ResultSet rs = null;
            OrderDetail orderDetail = null;

            try {
                // 2.创建数据库连接
                conn = DBHelper.getConnection();
```

```
            // 3. 创建语句对象
            String sql = "select orderid,productid,quantity,unitprice "
                    + "from ordersdetail where orderid = ? and productid = ?";

            pstmt = conn.prepareStatement(sql);
            // 4. 绑定参数
            pstmt.setInt(1, orderid);
            pstmt.setString(2, productid);

            // 5. 执行查询（R）
            rs = pstmt.executeQuery();
            // 6. 遍历结果集
            if (rs.next()) {

                orderDetail = new OrderDetail();
                orderDetail.setOrderid(rs.getInt("orderid"));
                orderDetail.setProductid(rs.getString("productid"));
                orderDetail.setQuantity(rs.getInt("quantity"));
                orderDetail.setUnitcost(rs.getDouble("unitcost"));

                return orderDetail;
            }

        } catch (SQLException e) {
            e.printStackTrace();
        } finally { // 释放资源
            if (rs != null) {
                try {
                    rs.close();
                } catch (SQLException e) {
                }
            }
            if (pstmt != null) {
                try {
                    pstmt.close();
                } catch (SQLException e) {
                }
            }
            if (conn != null) {
                try {
                    conn.close();
                } catch (SQLException e) {
                }
            }
        }

        return null;
    }

    @Override
    public int create(OrderDetail orderDetail) {
        try ( // 2.创建数据库连接
                Connection conn = DBHelper.getConnection();
                // 3. 创建语句对象

                PreparedStatement pstmt = conn
                        .prepareStatement("insert into ordersdetail "
                                + "(orderid, productid,quantity,unitcost) values (?,?,?,?)"

            // 4. 绑定参数
            pstmt.setLong(1, orderDetail.getOrderid());
            pstmt.setString(2, orderDetail.getProductid());
            pstmt.setInt(3, orderDetail.getQuantity());
            pstmt.setDouble(4, orderDetail.getUnitcost());

            // 5. 执行修改（C、U、D）
```

```
                int affectedRows = pstmt.executeUpdate();
                System.out.printf("成功插入%d条数据。\n", affectedRows);

            } catch (SQLException e) {
                return -1;
            }
            return 0;
        }

        @Override
        public int modify(OrderDetail orderDetail) {
            // TODO 自动生成的方法存根
            return 0;
        }

        @Override
        public int remove(OrderDetail orderDetail) {
            // TODO 自动生成的方法存根
            return 0;
        }

    }
```

OrderDetailDao接口中定义了5个抽象方法。但这些方法，在本项目中只需要实现findByPK()和create()方法。

29.4.3 迭代3.3：数据库帮助类DBHelper

数据库帮助类DBHelper可以进行JDBC驱动程序加载以及获得数据库连接。具体实现代码如下：

```
//DBHelper.java文件
package com.a51work6.jpetstore.dao.mysql;

import java.io.IOException;
import java.io.InputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.Properties;

//数据库辅助类
public class DBHelper {

    // 连接数据库url
    static String url;
    // 创建Properties对象
    static Properties info = new Properties();

    // 1.驱动程序加载
    static {                                                    ①
        // 获得属性文件输入流
        InputStream input = DBHelper.class.getClassLoader()
                .getResourceAsStream("com/a51work6/jpetstore/dao/mysql/config.properties");

        try {
            // 加载属性文件内容到Properties对象
            info.load(input);
            // 从属性文件中取出url
            url = info.getProperty("url");
            // 从属性文件中取出driver
            String driverClassName = info.getProperty("driver");
            Class.forName(driverClassName);
            System.out.println("驱动程序加载成功...");
```

```
            } catch (ClassNotFoundException e) {
                System.out.println("驱动程序加载失败...");
            } catch (IOException e) {
                System.out.println("加载属性文件失败...");
            }
        }                                                                    ②

        // 获得数据库连接
        public static Connection getConnection() throws SQLException {    ③
            // 创建数据库连接
            Connection conn = DriverManager.getConnection(url, info);
            return conn;
        }

    }
```

上述代码第①行~第②行通过静态代码库加载数据库驱动程序，并且在静态代码块中读取配置文件config.properties信息，该配置文件位于com.a51work6.jpetstore.dao.mysql包中，内容如下：

```
    driver=com.mysql.jdbc.Driver
    url=jdbc:mysql://localhost:3306/petstore
    user=root
    password=123456
    useSSL=false
    verifyServerCertificate=false
```

代码第③行提供了获得数据库连接方法，这是一个静态方法使用起来比较方便。

29.5 任务4：编写表示层代码

从客观上讲，表示层开发的工作量是很大的，不仅有很多细节工作。

29.5.1 迭代4.1：编写启动类

Java SE应用程序需要有一个具有main主方法的类，它是项目启动类，代码如下：

```
//MainApp.java文件
package com.a51work6.jpetstore.ui;

import com.a51work6.jpetstore.domain.Account;

//启动类
public class MainApp {

    // 用户登录成功后，保存当前用户信息
    public static Account accout;            ①

    public static void main(String[] args) {

        LoginFrame frame = new LoginFrame();
        frame.setVisible(true);
    }

}
```

在main主方法中实例化用户登录窗口——LoginFrame类。另外，代码第①行声明静态变量accout，当用户登录成功后accout用来保存后用户信息。静态变量accout可以在其他类中方便访问。这是为了模拟Web应用开发中的会话（Session）对象，等用户打开浏览器，登录Web系统后，服务器端会将用户信息保存到会话对象中。

29.5.2 迭代4.2：编写自定义窗口类——MyFrame

由于Swing提供的JFrame类启动窗口后默认位于在屏幕的左上角，而本项目中所有的窗口都屏幕居中的，因此自定义了窗口类MyFrame，MyFrame代码如下：

```
//MyFrame.java文件
package com.a51work6.jpetstore.ui;

import java.awt.Toolkit;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

import javax.swing.JFrame;

//这是一个屏幕居中的自定义窗口
public class MyFrame extends JFrame {

    // 获得当前屏幕的宽
    private double screenWidth = Toolkit.getDefaultToolkit().getScreenSize().getWidth();    ①
    // 获得当前屏幕的高
    private double screenHeight = Toolkit.getDefaultToolkit().getScreenSize().getHeight();  ②

    public MyFrame(String title, int width, int height) {
        super(title);

        // 设置窗口大小
        setSize(width, height);
```

```
            // 计算窗口位于屏幕中心的坐标
            int x = (int) (screenWidth - width) / 2;       ③
            int y = (int) (screenHeight - height) / 2;     ④
            // 设置窗口位于屏幕中心
            setLocation(x, y);

            // 注册窗口事件
            addWindowListener(new WindowAdapter() {        ⑤
                // 单击窗口关闭按钮时调用
                public void windowClosing(WindowEvent e) {
                    // 退出系统
                    System.exit(0);
                }
            });
        }
    }
```

上述代码第①行和第②行是获取当前屏幕的宽和高，具体的计算过程，见代码第③行和第④行，具体的原理在25.5.7节已经介绍过程，这里不再赘述。

另外，代码第⑤行注册窗口事件，当用户单击窗口的关闭按钮时调用System.exit(0)语句退出系统，继承MyFrame类的所有窗口都可以单击关闭按钮时退出系统。

29.5.3 迭代4.3：用户登录窗口

MainApp类运行时会启动用户登录窗口，界面如图29-12所示，界面中有一个文本框、一个密码框和两个按钮。用户输入账号和密码，单击“确定”按钮，如果输入的账号和密码正确，则登录成功进入商品列表窗口；如果输入的不正确，则弹出如图29-13所示的对话框。

图**29-12** 用户登录窗口

图**29-13** 用户登录失败提示

用户登录窗口LoginFrame代码如下：

```
    //LoginFrame.java文件
```

```
package com.a51work6.jpetstore.ui;
...
//用户登录窗口
public class LoginFrame extends MyFrame {

    private JTextField txtAccountId;
    private JPasswordField txtPassword;

    public LoginFrame() {
        super("用户登录", 400, 230);
        // 设置布局管理为绝对布局
        getContentPane().setLayout(null);

        JLabel label1 = new JLabel();
        label1.setHorizontalAlignment(SwingConstants.RIGHT);
        label1.setBounds(51, 33, 83, 30);
        getContentPane().add(label1);
        label1.setText("账号：");
        label1.setFont(new Font("微软雅黑", Font.PLAIN, 15));

        txtAccountId = new JTextField(10);
        txtAccountId.setText();
        txtAccountId.setBounds(158, 33, 157, 30);
        txtAccountId.setFont(new Font("微软雅黑", Font.PLAIN, 15));
        getContentPane().add(txtAccountId);

        JLabel label2 = new JLabel();
        label2.setText("密码：");
        label2.setFont(new Font("微软雅黑", Font.PLAIN, 15));
        label2.setHorizontalAlignment(SwingConstants.RIGHT);
        label2.setBounds(51, 85, 83, 30);
        getContentPane().add(label2);

        txtPassword = new JPasswordField(10);
        txtPassword.setText();
        txtPassword.setBounds(158, 85, 157, 30);
        getContentPane().add(txtPassword);

        JButton btnOk = new JButton();
        btnOk.setText("确定");
        btnOk.setFont(new Font("微软雅黑", Font.PLAIN, 15));
        btnOk.setBounds(61, 140, 100, 30);
        getContentPane().add(btnOk);

        JButton btnCancel = new JButton();
        btnCancel.setText("取消");
        btnCancel.setFont(new Font("微软雅黑", Font.PLAIN, 15));
        btnCancel.setBounds(225, 140, 100, 30);
        getContentPane().add(btnCancel);

        // 注册btnOk的ActionEvent事件监听器
        btnOk.addActionListener(e -> {                                          ①
            AccountDao accountDao = new AccountDaoImp();                        ②
            Account account = accountDao.findById(txtAccountId.getText());      ③

            String passwordText = new String(txtPassword.getPassword());        ④
            if (account != null && passwordText.equals(account.getPassword())) {    ⑤
                System.out.println("登录成功。");
                ProductListFrame form = new ProductListFrame();
                form.setVisible(true);
                setVisible(false);
                //用户登录成功后，将用户信息保存到MainApp.accout静态变量中
                MainApp.accout = account;
            } else {
                JLabel label = new JLabel("您输入的账号或密码有误，请重新输入。");
                label.setFont(new Font("微软雅黑", Font.PLAIN, 15));
```

```
                JOptionPane.showMessageDialog(null, label, "登录失败",
                                JOptionPane.ERROR_MESSAGE);                    ⑥
            }

        });

        // 注册btnCancel的ActionEvent事件监听器
        btnCancel.addActionListener(e -> {                    ⑦
            // 退出系统
            System.exit(0);
        });
    }
}
```

上述代码第①行是用户单击“确定”按钮调用代码块。代码第②行是创建AccountDaoImp对象，代码第③行是通过DAO对象调用findById()方法，该方法是通过用户账号查询用户信息。代码第④行从密码框中取出密码。代码第⑤行比较窗口界面中的密码与从数据库中查询的密码是否一致，如果一致则登录成功；否则登录失败，失败时弹出对话框，代码第⑥行JOptionPane.showMessageDialog方法可以弹出对话框，JOptionPane类还有类似的静态方法：

- showConfirmDialog：弹出确认框，可以Yes、No、Ok和Cancel等按钮。
- showInputDialog：弹出提示输入对话框。
- showMessageDialog：弹出消息提示对话框。

代码第⑥行showMessageDialog方法第一个参数是设置对话框所在的窗口，如果是当前窗口则设置为null，第二个参数要显示的消息，第三个参数是对话框标题，第四个参数要显示的消息类型，这些类型有：ERROR_MESSAGE、INFORMATION_MESSAGE、WARNING_MESSAGE、QUESTION_MESSAGE或PLAIN_MESSAGE，其中ERROR_MESSAGE是错误消息类型。

29.5.4 迭代**4.4**：商品列表窗口

登录成功后会进行商品列表窗口，如图29-14所示。在商品列表窗口是分栏显示的，左栏是商品列表，右栏是商品明细信息。商品列表窗口是PetStore项目的最核心窗口，在该窗口可进行操作如下：

- 查看商品信息：当左栏的表格中选择某一个商品时，右栏会显示该商品的详细信息。
- 选择商品类型进行查询：用户可以选择商品类型，单击“查询”按钮根据商品类型进行查询，如图29-15所示，选中“鱼类”商品类型时查询结果。
- 重置查询：根据商品类型查询后，如果想返回查询之前的状态，可以单击“重置”按钮重置商品列表，回到如图29-14所示界面。
- 添加商品到购物车：用户在商品列表中选中商品后，可以单击“添加到购物车”按钮，将选中的商品添加到购物车中，注意用户每单击一次增加一次该商品的数量到购物车。
- 查看购物车：用户单击“查看购物车”按钮后窗口会跳转到购物车窗口。

图**29-14**　商品列表窗口

图**29-15**　查询商品列表

商品列表窗口ProductListFrame代码如下：

```
//ProductListFrame.java文件
package com.a51work6.jpetstore.ui;
...

//商品列表窗口
public class ProductListFrame extends MyFrame {

    private JTable table;
    private JLabel lblImage;
    private JLabel lblListprice;
    private JLabel lblDescn;
    private JLabel lblUnitcost;

    // 商品列表集合
    private List<Product> products = null;
    // 创建商品Dao对象
    private ProductDao dao = new ProductDaoImp();

    // 购物车，键是选择的商品Id，值是商品的数量
    private Map<String, Integer> cart = new HashMap<String, Integer>();
    // 选择的商品索引
    private int selectedRow = -1;

    public ProductListFrame() {

        super("商品列表", 1000, 700);
```

```
        // 查询所有商品
        products = dao.findAll();                          ①

        // 添加顶部搜索面板
        getContentPane().add(getSearchPanel(), BorderLayout.NORTH);

        // 创建分栏面板
        JSplitPane splitPane = new JSplitPane();
        // 设置指定分隔条位置，从窗格的左边到分隔条的左边
        splitPane.setDividerLocation(600);
        // 设置左侧面板
        splitPane.setLeftComponent(getLeftPanel());
        // 设置右侧面板
        splitPane.setRightComponent(getRightPanel());
        // 把分栏面板添加到内容面板
        getContentPane().add(splitPane, BorderLayout.CENTER);
    }

    // 初始化搜索面板
    private JPanel getSearchPanel() {

        JPanel searchPanel = new JPanel();
        FlowLayout flowLayout = (FlowLayout) searchPanel.getLayout();
        flowLayout.setVgap(20);
        flowLayout.setHgap(40);

        JLabel lbl = new JLabel("选择商品类别: ");
        lbl.setFont(new Font("微软雅黑", Font.PLAIN, 15));
        searchPanel.add(lbl);

        String[] categorys = { "鱼类", "狗类", "爬行类", "猫类", "鸟类" };
        JComboBox comboBox = new JComboBox(categorys);
        comboBox.setFont(new Font("微软雅黑", Font.PLAIN, 15));
        searchPanel.add(comboBox);

        JButton btnGo = new JButton("查询");
        btnGo.setFont(new Font("微软雅黑", Font.PLAIN, 15));
        searchPanel.add(btnGo);

        JButton btnReset = new JButton("重置");
        btnReset.setFont(new Font("微软雅黑", Font.PLAIN, 15));
        searchPanel.add(btnReset);

        // 注册查询按钮的ActionEvent事件监听器
        btnGo.addActionListener(e -> {                     ②
            // 所选择的类别
            String category = (String) comboBox.getSelectedItem();
            // 按照类别进行查询
            products = dao.findByCategory(category);
            TableModel model = new ProductTableModel(products);
            table.setModel(model);
        });

        // 注册重置按钮的ActionEvent事件监听器
        btnReset.addActionListener(e -> {
            products = dao.findAll();                      ③
            TableModel model = new ProductTableModel(products);
            table.setModel(model);
        });

        return searchPanel;
    }

    // 初始化右侧面板
    private JPanel getRightPanel() {
```

```
JPanel rightPanel = new JPanel();
rightPanel.setBackground(Color.WHITE);

rightPanel.setLayout(new GridLayout(2, 1, 0, 0));

lblImage = new JLabel();
rightPanel.add(lblImage);
lblImage.setHorizontalAlignment(SwingConstants.CENTER);

JPanel detailPanel = new JPanel();
detailPanel.setBackground(Color.WHITE);
rightPanel.add(detailPanel);
detailPanel.setLayout(new GridLayout(8, 1, 0, 5));

JSeparator separator_1 = new JSeparator();
detailPanel.add(separator_1);

lblListprice = new JLabel();
detailPanel.add(lblListprice);
// 设置字体
lblListprice.setFont(new Font("微软雅黑", Font.PLAIN, 16));

lblUnitcost = new JLabel();
detailPanel.add(lblUnitcost);
// 设置字体
lblUnitcost.setFont(new Font("微软雅黑", Font.PLAIN, 16));

lblDescn = new JLabel();
detailPanel.add(lblDescn);
// 设置字体
lblDescn.setFont(new Font("微软雅黑", Font.PLAIN, 16));

JSeparator separator_2 = new JSeparator();
detailPanel.add(separator_2);

JButton btnAdd = new JButton("添加到购物车");
btnAdd.setFont(new Font("微软雅黑", Font.PLAIN, 15));
detailPanel.add(btnAdd);

// 布局占位使用
JLabel lb1 = new JLabel("");
detailPanel.add(lb1);

JButton btnCheck = new JButton("查看购物车");
btnCheck.setFont(new Font("微软雅黑", Font.PLAIN, 15));
detailPanel.add(btnCheck);

// 注册【添加到购物车】按钮的ActionEvent事件监听器
btnAdd.addActionListener(e -> {                                    ④

    if (selectedRow < 0) {
        return;
    }
    // 添加商品到购物车处理
    Product selectProduct = products.get(selectedRow);
    String productid = selectProduct.getProductid();

    if (cart.containsKey(productid)) {// 购物车中已经有该商品      ⑤
        // 获得商品数量
        Integer quantity = cart.get(productid);                    ⑥
        cart.put(productid, ++quantity);                           ⑦
    } else {// 购物车中还没有该商品
        cart.put(productid, 1);                                    ⑧
    }

    System.out.println(cart);
```

```
        });

        // 注册【查看购物车】按钮的ActionEvent事件监听器
        btnCheck.addActionListener(e -> {                                    ⑨
            CartFrame cartFrame = new CartFrame(cart, this);
            cartFrame.setVisible(true);
            setVisible(false);
        });

        return rightPanel;
    }

    // 初始化左侧面板
    private JScrollPane getLeftPanel() {

        JScrollPane leftScrollPane = new JScrollPane();
        // 将表格作为滚动面板的各个视口视图
        leftScrollPane.setViewportView(getTable());
        return leftScrollPane;
    }

    // 初始化左侧面板中的表格控件
    private JTable getTable() {

        TableModel model = new ProductTableModel(this.products);

        if (table == null) {
            table = new JTable(model);
            // 设置表中内容字体
            table.setFont(new Font("微软雅黑", Font.PLAIN, 16));
            // 设置表列标题字体
            table.getTableHeader().setFont(new Font("微软雅黑", Font.BOLD, 16));
            // 设置表行高
            table.setRowHeight(51);
            table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
            ListSelectionModel rowSelectionModel = table.getSelectionModel();

            rowSelectionModel.addListSelectionListener(e -> {               ⑩

                //只处理鼠标释放
                if (e.getValueIsAdjusting()) {
                    return;
                }

                ListSelectionModel lsm = (ListSelectionModel) e.getSource();
                selectedRow = lsm.getMinSelectionIndex();
                if (selectedRow < 0) {
                    return;
                }
                // 更新右侧面板内容
                Product p = products.get(selectedRow);
                String petImage = String.format("/images/%s", p.getImage());    ⑪
                ImageIcon icon = new ImageIcon(ProductListFrame.class.getResource(petImage));    ⑫
                lblImage.setIcon(icon);

                String descn = p.getDescn();
                lblDescn.setText("商品描述：" + descn);

                double listprice = p.getListprice();
                String slistprice = String.format("商品市场价：%.2f", listprice);
                lblListprice.setText(slistprice);

                double unitcost = p.getUnitcost();
                String slblUnitcost = String.format("商品单价：%.2f", unitcost);
                lblUnitcost.setText(slblUnitcost);
            });
```

```
            } else {
                table.setModel(model);
            }
            return table;
        }

    }
```

上述代码第①行的roducts = dao.findAll()语句是查询所有数据，单击“重置”按钮也调用dao.findAll()语句查询所有数据，见代码第③行。代码第②行代码块是用户单击“查询”按钮调用的。

代码第④行的代码块是用户单击“添加到购物车”按钮调用的，其中代码第⑤行判断购物车中是否已经有了选中的商品，如果有则通过代码第⑥行取出商品数量，代码第⑦行是将商品数量加一后，再重新放回到购物车中。如果没有商品则将该商品添加到购物车中，商品数量为1。

代码第⑨行是单击“查看购物车”按钮时调用的代码块。此时当前界面会调转到购物车窗口。

代码第⑩行是用户选中表格中某一行时调用的代码块，在这里根据用户选中的商品更新右边的详细商品信息。

代码第⑪行是获得图片相对路径，它们属于资源目录（即src源文件夹）。代码第⑫行的ProductListFrame.class.getResource(petImage)语句可以获得图片文件运行时的绝对路径。

商品列表窗口中使用了自定义表格模型ProductTableModel，ProductTableModel代码如下：

```
    //ProductTableModel.java文件
    package com.a51work6.jpetstore.ui;
    ...
    //商品列表表格模型
    public class ProductTableModel extends AbstractTableModel {

        // 表格列名columnNames
        private String[] columnNames = { "商品编号", "商品类别", "商品中文名", "商品英文名" };

        // 表格中的内容保存在List<Product>集合中
        private List<Product> data = null;

        public ProductTableModel(List<Product> data) {
            this.datas = data;
        }

        // 返回列数
        @Override
        public int getColumnCount() {
            return columnNames.length;
        }

        // 返回行数
        @Override
        public int getRowCount() {
            return data.size();
        }

        // 获得某行某列的数据，而数据保存在对象数组data中
        @Override
        public Object getValueAt(int rowIndex, int columnIndex) {

            // 每一行就是一个Product商品对象
            Product p = data.get(rowIndex);

            switch (columnIndex) {
```

```
            case 0:
                return p.getProductid(); // 第一列商品编号
            case 1:
                return p.getCategory(); // 第二列商品类别
            case 2:
                return p.getCname(); // 第三列商品中文名
            default:
                return p.getEname(); // 第四列商品英文名
            }
        }

        @Override
        public String getColumnName(int columnIndex) {
            return columnNames[columnIndex];
        }
    }
```

上述表格模型代码继承了AbstractTableModel抽象类，表格中的数据保存在List<Product>集合中。类似的表格模型在25.6节介绍过，这里不再赘述。

29.5.5 迭代**4.5**：商品购物车窗口

当用户在商品列表窗口，单击了“查看购物车”按钮，则会跳转到商品购物车窗口,如图29-16所示。在该窗口可进行操作如下：

- 返回商品列表：当用户单击“返回商品列表”按钮时，界面跳转回上一级窗口（商品列表窗口），用户还可以重新添加新的到购物车。
- 修改商品数量：用户如果想修改商品数量，可以在购物车表格中双击某一商品数量单元格，使其进入编辑状态。用户只能输入大于0的数值，不能输入负数或非数值字符。
- 提交订单：如果商品选择完成，用户想提交订单，可以单击“提交订单”按钮生成订单，订单生成会在数据库中插入订单信息和订单明细信息。然后会弹出如图29-17所示订单等待付款确认对话框，如果用户单击“是”按钮则进入付款流程，由于付款需要实际的支付接口，因此付款功能未实现。如果用户单击“否”则退出系统。

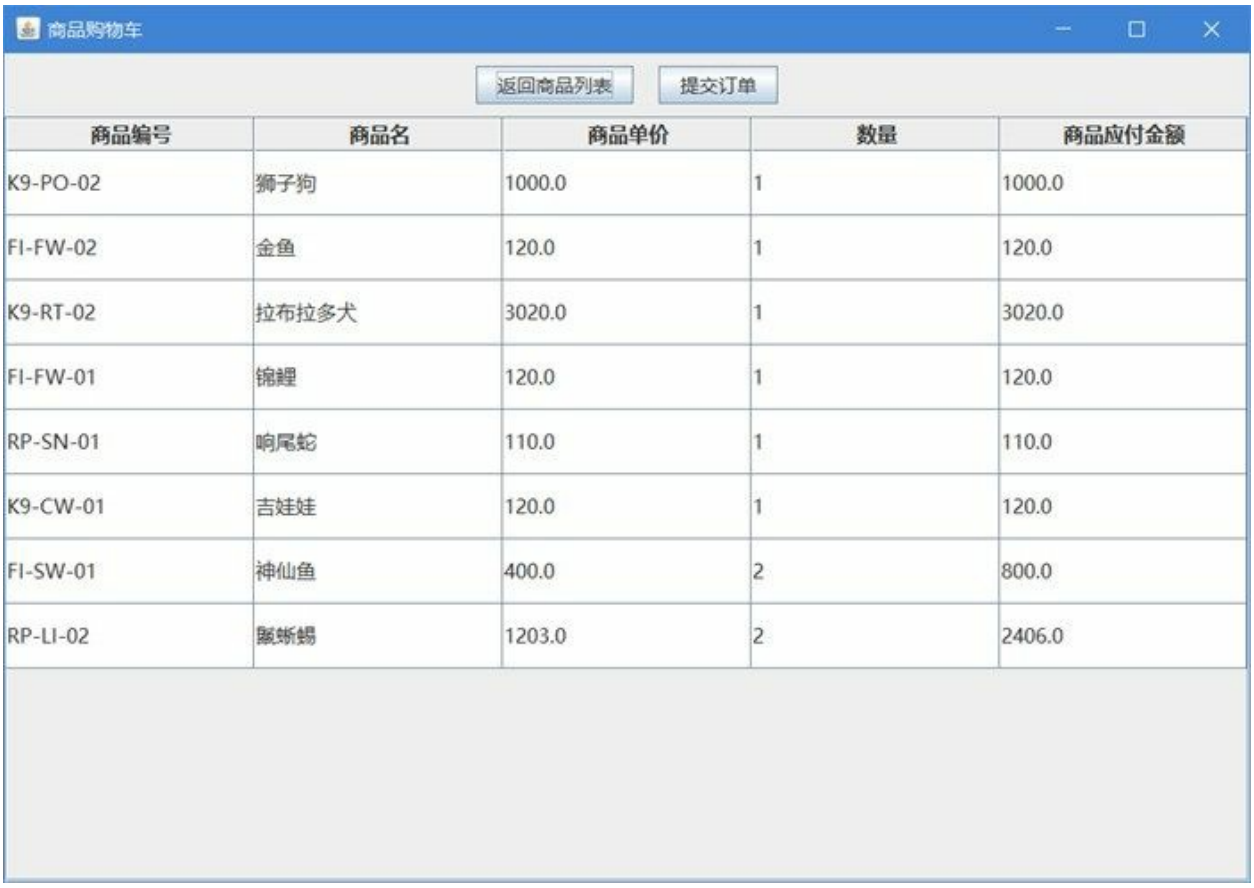


图**29-16**　商品购物车窗口

图**29-17**　订单生成确认对话框

商品购物车窗口CartFrame代码如下：

```
    //CartFrame.java文件
    package com.a51work6.jpetstore.ui;
    ...
    //商品购物车窗口
    public class CartFrame extends MyFrame {

        private JTable table;

        // 购物车数据
        private Object[][] data = null;

        // 创建商品Dao对象
        private ProductDao dao = new ProductDaoImp();
```

```
// 购物车，键是选择的商品Id，值是商品的数量
private Map<String, Integer> cart;
// 引用到上级Frame（ProductListFrame）
private ProductListFrame productListFrame;

public CartFrame(Map<String, Integer> cart, ProductListFrame productListFrame) {

    super("商品购物车", 1000, 700);
    this.cart = cart;
    this.productListFrame = productListFrame;

    JPanel topPanel = new JPanel();
    FlowLayout fl_topPanel = (FlowLayout) topPanel.getLayout();
    fl_topPanel.setVgap(10);
    fl_topPanel.setHgap(20);
    getContentPane().add(topPanel, BorderLayout.NORTH);

    JButton btnReturn = new JButton("返回商品列表");
    btnReturn.setFont(new Font("微软雅黑", Font.PLAIN, 15));
    topPanel.add(btnReturn);

    JButton btuSubmit = new JButton("提交订单");
    topPanel.add(btuSubmit);
    btuSubmit.setFont(new Font("微软雅黑", Font.PLAIN, 15));

    JScrollPane scrollPane = new JScrollPane();
    getContentPane().add(scrollPane, BorderLayout.CENTER);
    scrollPane.setViewportView(getTable());

    // 注册【提交订单】按钮的ActionEvent事件监听器
    btuSubmit.addActionListener(e -> {                          ①
        // 生成订单
        generateOrders();

        JLabel label = new JLabel("订单已经生成，等待付款。");
        label.setFont(new Font("微软雅黑", Font.PLAIN, 15));
        if (JOptionPane.showConfirmDialog(this, label, "信息",
                JOptionPane.YES_NO_OPTION) == JOptionPane.YES_OPTION) {
            // TODO 付款
            System.exit(0);
        } else {
            System.exit(0);
        }
    });

    // 注册【返回商品列表】按钮的ActionEvent事件监听器
    btnReturn.addActionListener(e -> {
        // 更新购物车
        for (int i = 0; i < data.length; i++) {
            // 商品编号
            String productid = (String) data[i][0];
            // 数量
            Integer quantity = (Integer) data[i][3];
            cart.put(productid, quantity);
        }
        this.productListFrame.setVisible(true);
        setVisible(false);
    });
}

// 初始化左侧面板中的表格控件
private JTable getTable() {

    // 准备表中数据
    data = new Object[cart.size()][5];
```

```
        Set<String> keys = this.cart.keySet();
        int indx = 0;
        for (String productid : keys) {
            Product p = dao.findById(productid);
            data[indx][0] = p.getProductid();// 商品编号
            data[indx][1] = p.getCname();// 商品名
            data[indx][2] = new Double(p.getUnitcost());// 商品单价
            data[indx][3] = new Integer(cart.get(productid));// 数量
            // 计算商品应付金额
            double amount = (double) data[indx][2] * (int) data[indx][3];
            data[indx][4] = new Double(amount);
            indx++;
        }

        // 创建表数据模型
        TableModel model = new CartTableModel(data);

        if (table == null) {
            // 创建表
            table = new JTable(model);
            // 设置表中内容字体
            table.setFont(new Font("微软雅黑", Font.PLAIN, 16));
            // 设置表列标题字体
            table.getTableHeader().setFont(new Font("微软雅黑", Font.BOLD, 16));
            // 设置表行高
            table.setRowHeight(51);
            table.setRowSelectionAllowed(false);

        } else {
            table.setModel(model);
        }
        return table;
    }

    // 生成订单
    private void generateOrders() {                                    ②

        OrderDao orderDao = new OrderDaoImp();
        OrderDetailDao orderDetailDao = new OrderDetailDaoImp();

        Order order = new Order();
        order.setUserid(MainApp.accout.getUserid());                   ③
        // 0待付款
        order.setStatus(0);
        // 订单Id是当前时间
        Date now = new Date();
        long orderId = now.getTime();
        order.setOrderid(orderId);                                     ④
        order.setOrderdate(now);
        order.setAmount(getOrderTotalAmount());                        ⑤

        // 下订单时间是数据库自动生成不用设置
        // 创建订单
        orderDao.create(order);                                        ⑥

        for (int i = 0; i < data.length; i++) {

            OrderDetail orderDetail = new OrderDetail();
            orderDetail.setOrderid(orderId);
            orderDetail.setProductid((String) data[i][0]);
            orderDetail.setQuantity((int) data[i][3]);
            orderDetail.setUnitcost((double) data[i][2]);
            // 创建订单详细
            orderDetailDao.create(orderDetail);                        ⑦
        }
    }
```

```
        // 计算订单应付总金额
        private double getOrderTotalAmount() {

            double totalAmount = 0.0;
            for (int i = 0; i < data.length; i++) {
                // 计算商品应付金额
                totalAmount += (Double) data[i][4];
            }
            return totalAmount;
        }
    }
```

当用户单击“提交订单”按钮时调用代码第①行的代码块，在该代码块中首先调用generateOrders()方法生成订单，然后通过调用JOptionPane.showConfirmDialog方法弹出付款确认对话框。

代码第②行是生成订单generateOrders()方法定义，在该方法中将订单信息插入到数据库订单表和订单明细表中。其中代码第③行是设置订单中用户Id属性，这个属性是在登录时候保存在MainApp.accout静态变量中的。代码第④行是设置订单Id属性，订单Id生成规则是当前系统时间毫秒数，这种生成规则在用户访问量少情况下可以满足要求。代码第⑤行是设置该订单应付金额，该金额的计算是通过getOrderTotalAmount()方法实现的，就是将订单中所有商品价格乘以数量，然后累加起来。

代码第⑥行是将订单数据插入到数据库中，由于订单中有可能有多个商品，所有代码第⑦行循环插入订单明细数据。

订单生成后可以在数据中查看生成的结果，如图29-18所示。

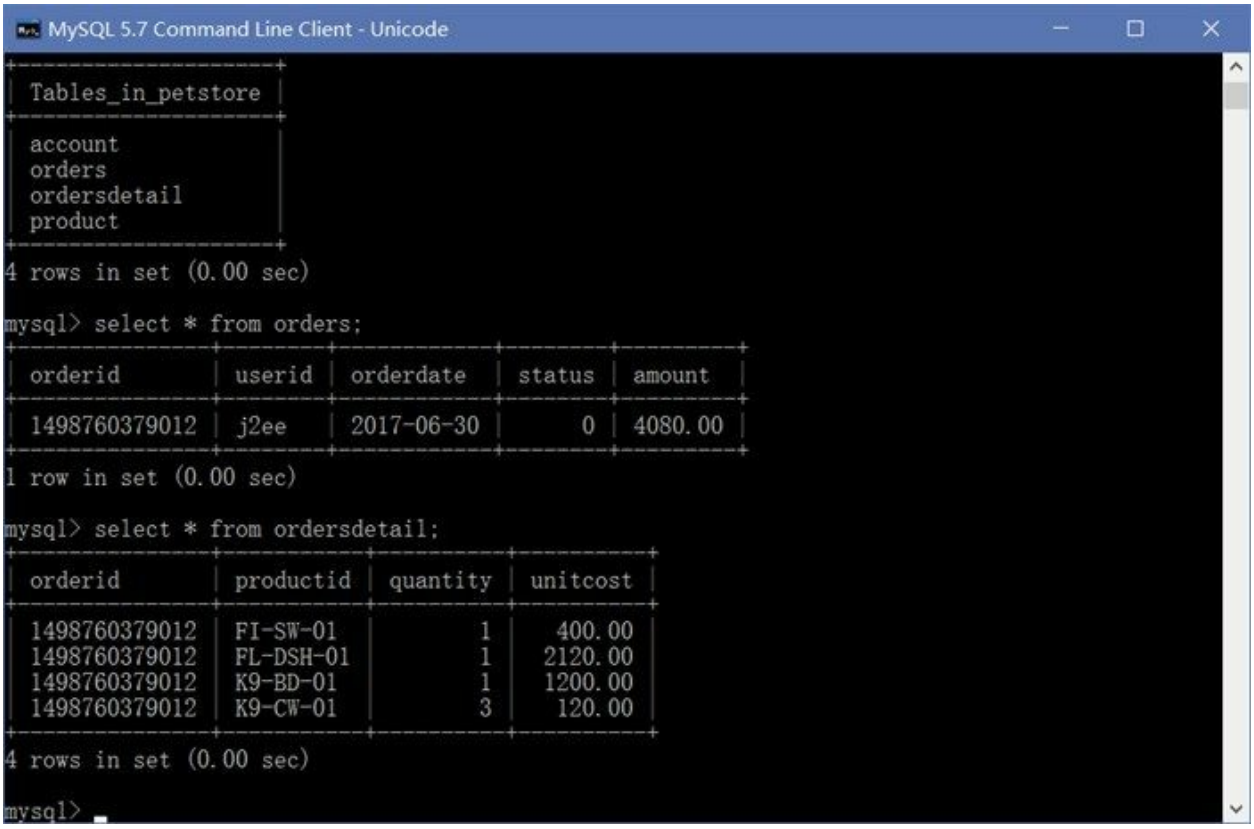


图**29-18**　订单生成数据

购物车窗口中会用到购物车表格，购物车表格比较复杂，用户可以修改数量这一列，其他的列不能修改，还有修改的数量是要验证的，不能小于0，更不能输入非数值字符。这些需求的解决是通过自定义表格模型实现的，表格模型CartTableModel代码如下：

```
//CartTableModel.java文件
package com.a51work6.jpetstore.ui;

import javax.swing.table.AbstractTableModel;

//购物车表格模型
public class CartTableModel extends AbstractTableModel {

    // 表格列名columnNames
    private String[] columnNames = { "商品编号", "商品名", "商品单价", "数量", "商品应付金额" };

    // 表格中数据保存在data二维数组中
    private Object[][] data = null;

    public CartTableModel(Object[][] data) {
        this.data = data;
    }

    // 返回列数
    @Override
    public int getColumnCount() {
        return columnNames.length;
    }

    // 返回行数
    @Override
    public int getRowCount() {
        return data.length;
    }

    // 获得某行某列的数据，而数据保存在对象数组data中
    @Override
    public Object getValueAt(int rowIndex, int columnIndex) {
        return data[rowIndex][columnIndex];
    }

    @Override
    public String getColumnName(int columnIndex) {
        return columnNames[columnIndex];
    }

    @Override
    public boolean isCellEditable(int rowIndex, int columnIndex) {            ①
        // 数量列可以修改
        if (columnIndex == 3) {
            return true;
        }
        return false;
    }

    @Override
    public void setValueAt(Object aValue, int rowIndex, int columnIndex) {    ②
        // 只允许修改数量列
        if (columnIndex != 3) {                    ③
            return;
        }
        try {
            // 从表中获得修改之后的商品数量，从表而来的数据都String类型
            int quantity = new Integer((String) aValue);    ④
            // 商品数量不能小于0
            if (quantity < 0) {                    ⑤
```

```
                    return;
                }
                // 更新数量列
                data[rowIndex][3] = quantity;            ⑥
                // 计算商品应付金额
                double unitcost = (double) data[rowIndex][2];    ⑦
                double totalPrice = unitcost * quantity;        ⑧
                // 更新商品应付金额列
                data[rowIndex][4] = new Double(totalPrice);    ⑨

            } catch (Exception e) {
            }
        }
    }
```

为了能让表格可以被编辑需要覆盖代码第①行的isCellEditable方法，在该方法中判断当前列索引是3（就是数量列）则返回ture，表示这一列可以修改。

在修改数量时需要进行验证，则需要覆盖代码第②行的setValueAt方法，其中aValue参数是当前单元格（rowIndex，columnIndex）的输入值。代码第③行判断数量列才进行处理。代码第④行将输入值aValue转换为整数，如果是非数值字符会发生异常，结束setValueAt方法。代码第⑤行是判断小于0时结束setValueAt方法。代码第⑥行将的是用输入值aValue替换二维data中对应的元素。代码第⑦行data[rowIndex][2]是取出二维数组商品单价。代码第⑧行是计算商品应付金额，然后通过代码第⑨行将商品应付金额更新二维数组data中的商品应付金额元素。

29.6 任务5：应用程序打包发布

编写的Java程序最后要被人使用，大多数人他们不会用JDK指令或Eclipse工具运行java程序，而且一个java项目可能有很多字节码文件，使用起来也不好管理。因此，最后需要发布时需要给应用程序打包。

29.6.1 迭代5.1：处理TODO、FIXME和XXX任务

在最后发布打包之前，还需要处理一些任务，其中首先应该处理代码中的：TODO、FIXME和XXX注释任务，有关这三种注释详细解释请参考5.2.4节。PetStore项目待处理任务如图29-19所示。

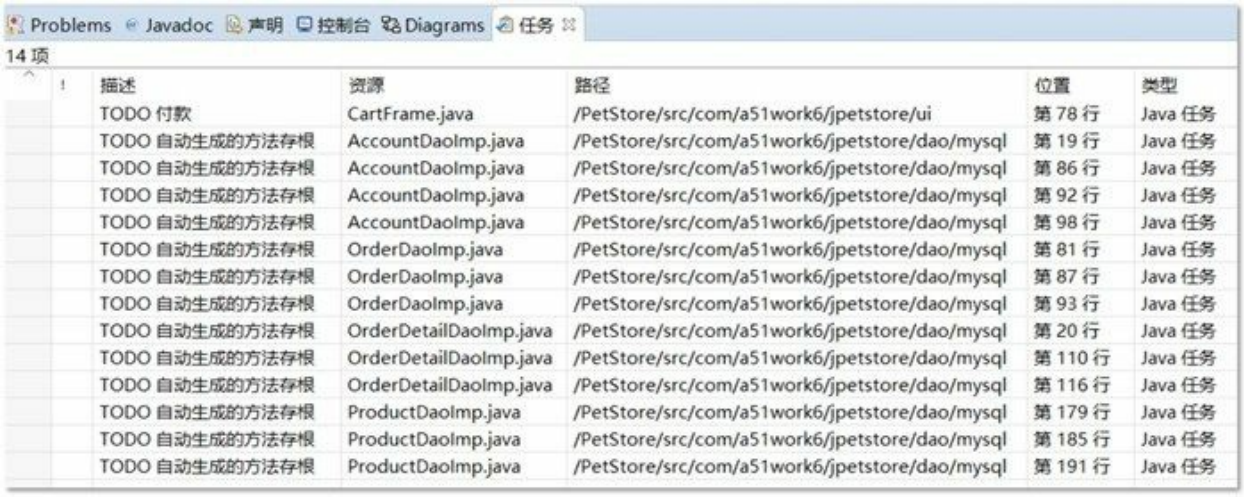


图29-19 处理TODO、FIXME和XXX任务

请根据需要处理这些任务。

29.6.2 迭代5.2：处理警告

Java的程序代码往往还有很多警告，这些警告也不可忽视，应该检查一下那些是程序本身的问题，然后加以修正。这些警告可以在，如图29-20所示问题视图中查看到。

如果问题视图没有打开，可以通过菜单“窗口”→“显示视图”→“问题”，打开如图29-0所示的视图，双击其次的问题可以跳转到代码处。

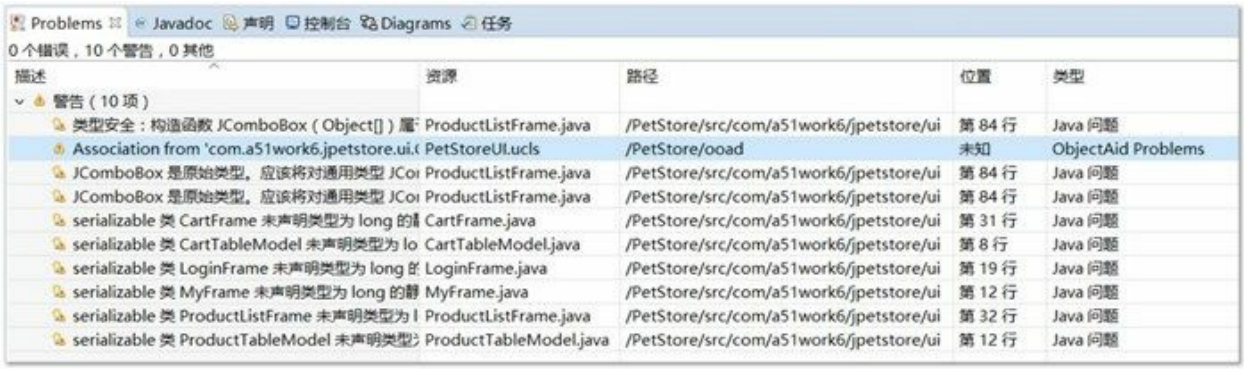


图29-20 处理警告任务

29.6.3 迭代5.3：打包

任务和问题都已经检查并修正后，可以打包了，在JDK中有一个jar命令，它可以为Java字节码文件进行打包，打包之后的文件一般是.jar文件，该文件是zip压缩格式。使用jar文件有很多好处，首先文件是经过压缩占用空间小，其次是文件多个字节码、资源和配置文件被打包成一个文件方便管理。

要发布的Java项目的类型不同，打包的内容也会有所不同。主要是注意Java应用程序（Java Application）[3]的项目打包，与其他的Java项目所有不同。Java应用程序在打包是需要指定包含main主方法的类是哪一个。PetStore项目属于Java应用程序项目，下面来介绍一下PetStore项目打包过程。

[3]Java应用程序是指包含有main主方法的类，通过该类能够其中Java程序。

可以使用JDK中的jar命令或者使用Eclipse等IDE工具进行打包，下面先来介绍jar命令打包。

01. 使用jar命令打包

    jar命令模拟UNIX和Linux中的tar命令，参数也类似，常用的有：

    - -c：创建新文档。
    - -x：从文档案中解压文件。
    - -v：输出压缩或解压信息。
    - -f：指定文档文件名。
    - -m：指定一个自定义清单文件。
    - -C：更改默认目录为指定的目录，默认目录是当前目录，有时是需要打包其他目录下的文件，则需要使用此参数。

    首先需要创建一个打包目录，将需要准备打包文件复制到这里，如图29-21所示，将Eclipse编译之后的bin目录复制打包目录，lib目录是放置第三方类库文件，PetStroe项目需要MySQL的JDBC驱动程序mysql-connector-java-5.1.41-bin.jar，该文件需要复制到lib目录。打包目录下还有一个mymanifest.txt文件，它自己编写的清单（Manifest）文件，所有的jar文件都包含一个清单文件，如果不提供清单文件jar命令会生成该文件，mymanifest.txt文件内容如下：

```
Manifest-Version: 1.0
Class-Path: . ./lib/mysql-connector-java-5.1.41-bin.jar
Main-Class: com.a51work6.jpetstore.ui.MainApp
```

    其中Manifest-Version指定清单文件版本号。Class-Path指定类路径，多个类路径之间用空格分隔，mymanifest.txt中有两个类路径，“.”表示当前路径，./lib/mysql-connector-java-5.1.41-bin.jar表示当前lib目录下的mysql-connector-java-5.1.41-bin.jar文件。Main-Class指定主类文件，就是包含main主方法的类。

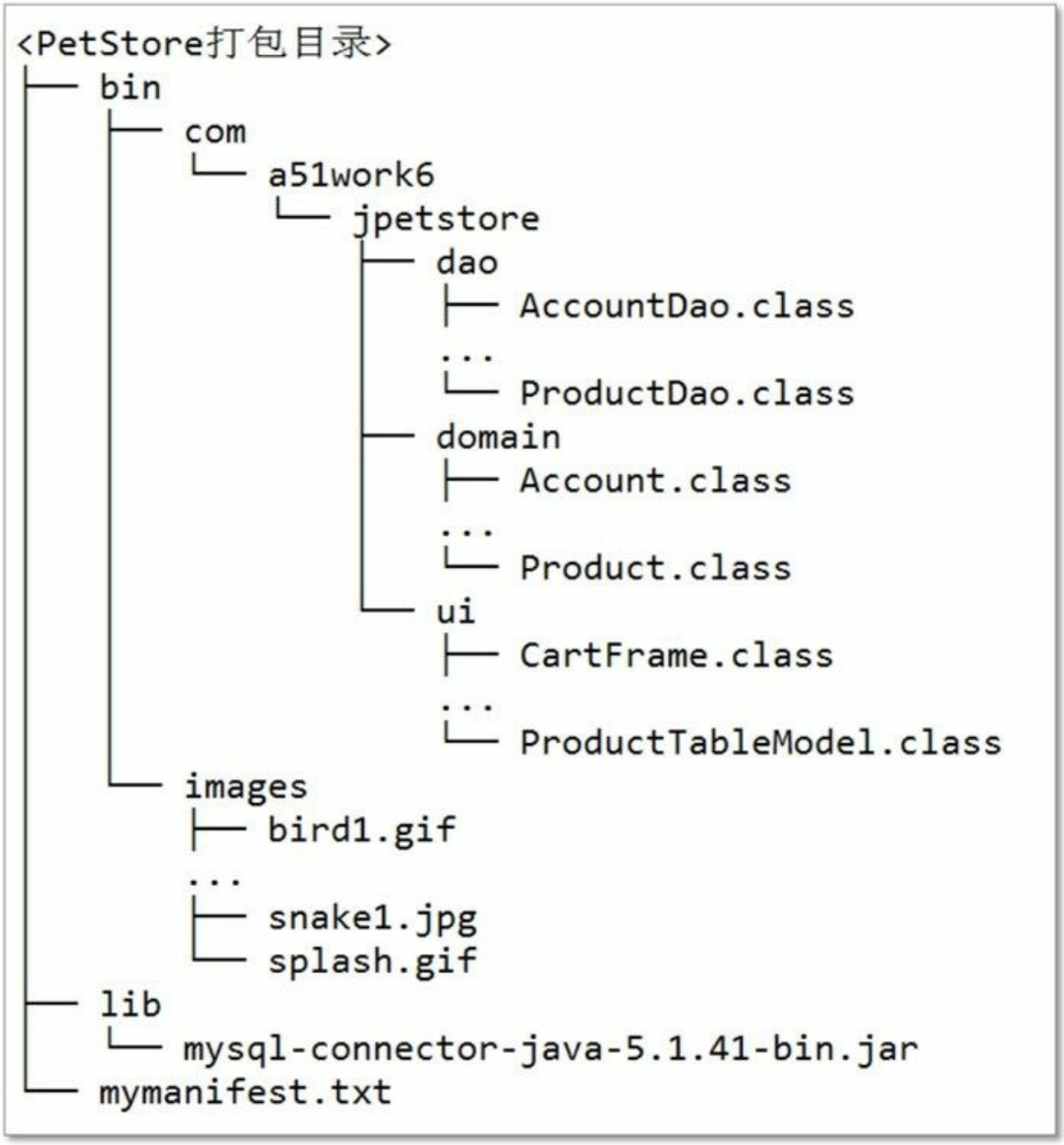


图**29-21**　打包目录结构

启动命令行工具，进入到PetStore目录，输入如下指令：

```
jar -cvfm PetStore.jar mymanifest.txt -C bin . ./lib
```

-cvfm参数是c、v、f、m四个参数的叠加，PetStore.jar打包生成的文件，mymanifest.txt指定的清单文件，-C参数指定默认目录是bin目录，“.”代表bin目录所有文件都打包到PetStore.jar中，./lib表示把该目录也打包到PetStore.jar中。

输入指令会按Enter键，终端窗口会输出打包信息。如果成功会在当前目录下生成一个PetStore.jar文件。PetStore.jar文件可以使用jar解压，由于PetStore.jar是Java应用程序所以可以运行，最简单的运行方式是双击该文件就可以运行了，也可以使用Java命令，在命令行中输入如

下命令如下：

```
java -jar PetStore.jar
```

02. 使用Eclipse IDE工具进行打包

使用Eclipse IDE工具进行打包非常简单，在Eclipse中选择PetStore项目，右键在菜单中选择“导出”，弹出如图29-22所示对话框，“可运行的JAR文件”选项是打包Java应用程序时使用，普通程序打包选择“JAR文件”。

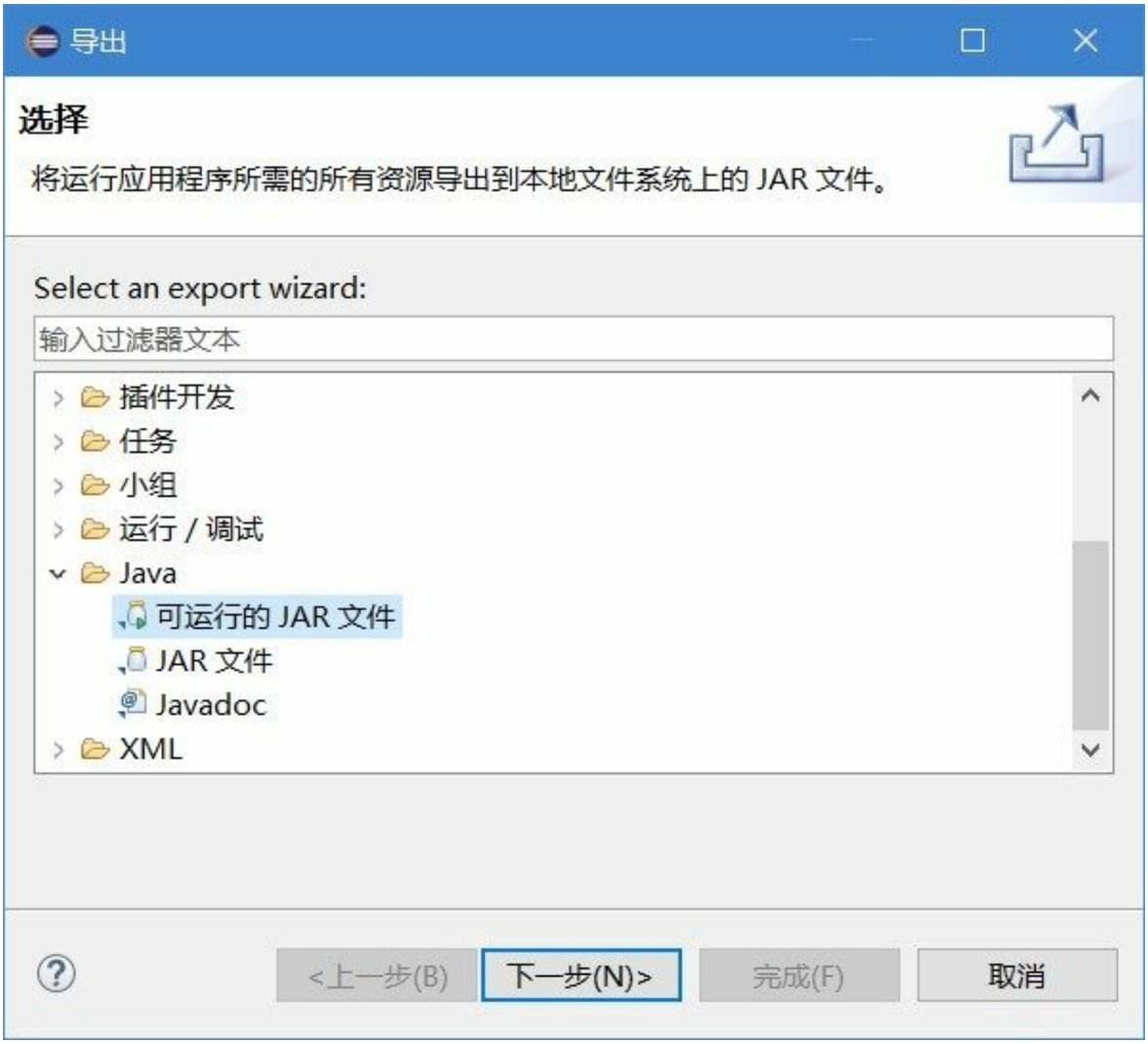


图**29-22**　导出文件对话框

在图29-22中选择“可运行的JAR文件” 后，单击“下一步”按钮，进入如图29-23所示对话框，这里可以选择第三方库的处理方式。

①处理方式是将库中所需要的字节码文件提取出来，打到到jar文件中，这种方式是笔者推荐的做法。

②处理方式是将库直接打包到jar文件中，然后在清单文件中指定类路径，类似于前文介绍的jar打包方式。

③处理方法是将库不打包到jar文件中，而放置在与jar文件所在目录下的XXX_lib目录中。这种方式发布的时候需要把jar文件和XXX_lib目录一同发布。

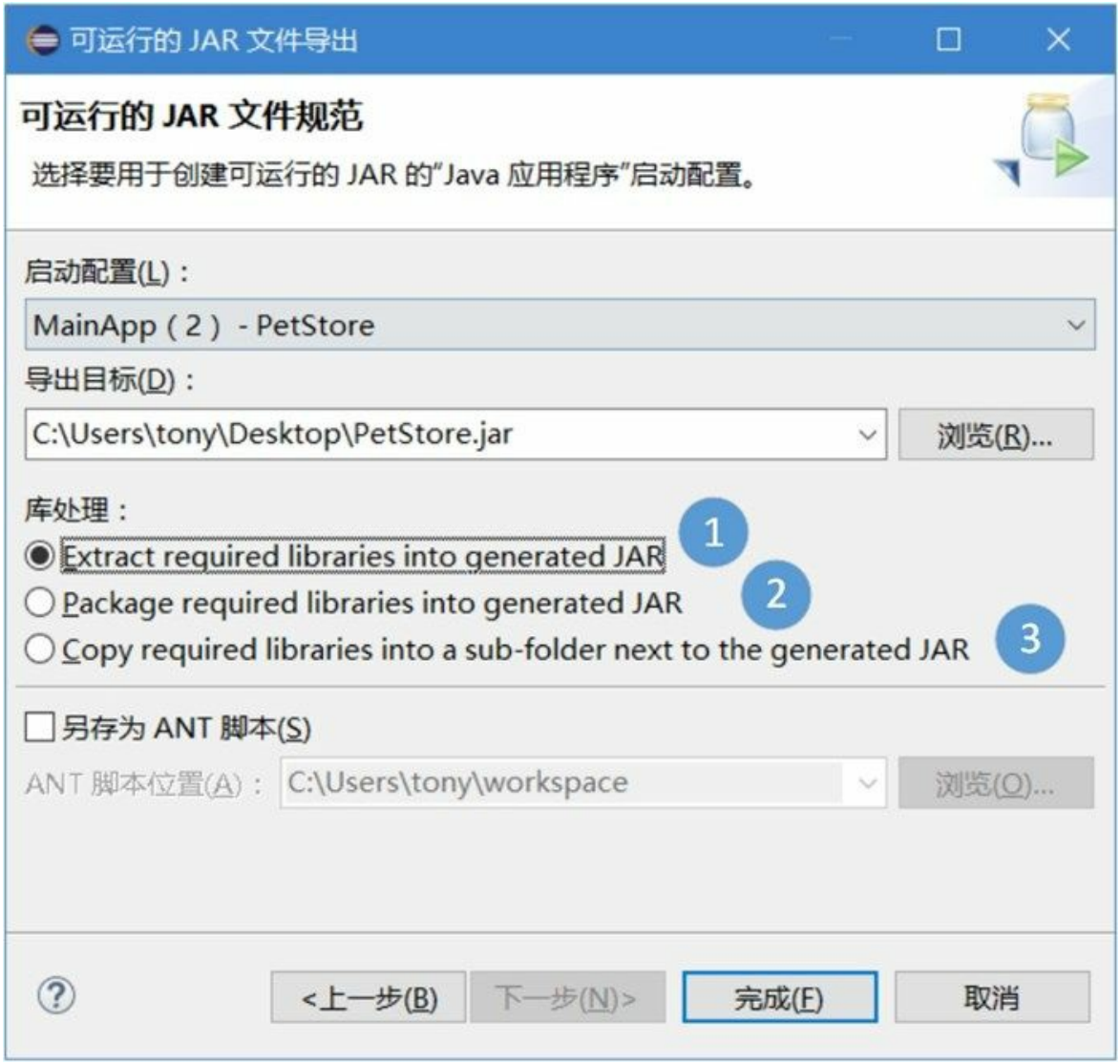


图**29-23**　第三方库处理方式

打包文件生成之后就可以双击运行了。

第 30 章　项目实战2：开发Java版QQ2006聊天工具

上一章开发的PetStore宠物商店项目没涉及到多线程和网络通信，本章介绍的QQ2006聊天工具会涉及到这方面的技术。

本章介绍Java SE技术实现的QQ2006聊天工具项目，所涉及到的知识点：Java面向对象、Lambda表达式、Java Swing技术、多线程技术和网络通信等知识，其中还会用到方方面面的Java基础知识。

30.1 系统分析与设计

本节对QQ2006聊天工具项目分析和设计，其中设计过程包括原型设计、数据库设计和系统设计。

30.1.1 项目概述

QQ2006是一个网络即时聊天工具，即时聊天工具可以在两名或多名用户之间传递即时消息的网络软件，大部分的即时聊天软件都可以显示联络人名单，并能显示联络人是否在线，聊天者发出的每一句话都会显示在双方的屏幕上。

即时聊天工具主要有：

- ICQ：最早的网络即时通讯工具。1996年，三个以色列人维斯格、瓦迪和高德芬格一起开发了ICQ工具。ICQ支持在Internet上聊天、发送消息和文件等。
- QQ：国内最流行的即时通讯工具。
- MSN Messenger：是微软所开发，曾经在公司中使用广泛。
- 百度HI：百度公司推出的一款集文字消息、音视频通话、文件传输等功能的即时通讯软件。
- 阿里旺旺：是阿里巴巴公司为自己旗下产品用户定制的商务沟通软件。
- Gtalk：Google的即时通讯工具。
- Skype：网络即时语音沟通工具。
- 微信：基于移动平台的即时通讯工具。

30.1.2 需求分析

QQ2006项目工具分为有客户端和服务器端，客户端和服务器端都提供了很多工作线程，这些线程帮助进行后台通信等处理。

客户端有聊天用户和工作线程完成工作，客户端主要功能如下：

- 用户登录：用户打开登录窗口，单击登录按钮登录。客户端工作线程向服务器发送用户登录请求消息；客户端工作线程接收到服务器返回信息，如果成功界面跳转，是否弹出提示框，提示用户登录失败。
- 打开聊天对话框：用户双击好友列表中的好友，打开聊天对话框。
- 显示好友列表：当用户登录成功后，客户端工作线程接收服务器端数据，根据数据显示好友列表。
- 刷新好友列表：每一个用户上线（登录成功），服务器会广播用户上线消息，客户端工作线程接收到用户上线消息，则将好友列表中好友在线状态更新。
- 向好友发送消息：用户在聊天对话框中发送消息给好友，服务器端工作线程接收到这个消息后，转发给用户好友。
- 接收好友消息：客户端工作线程接收好友消息，这个消息是服务器转发的。
- 用户注销：单击好友列表的关闭窗口，则用户下线。客户端工作线程向服务器发送用户下线消息。

采用用例分析方法描述客户端用例图，如图30-1所示。

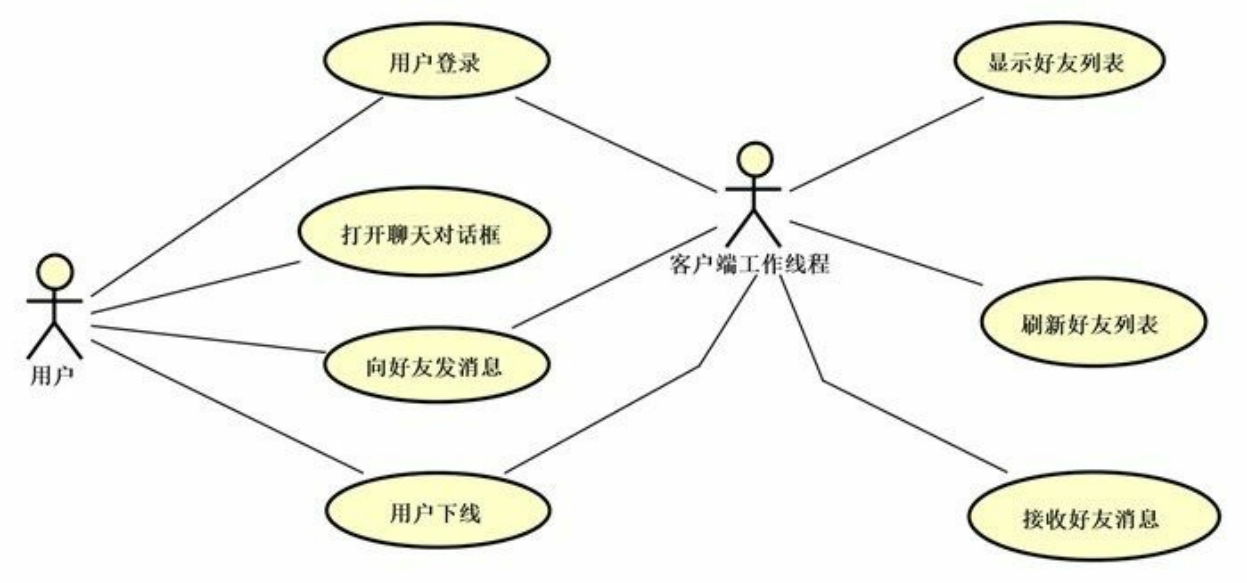


图**30-1　QQ2006**项目客户端用例图

服务器端所有功能都是通过服务线程工作线程完成的，没有人为操作，服务器端主要功能如下：

- 客户用户登录：客户端用户发生登录请求，服务器端工作线程查询数据库用户信息，验证用户登录。用户登录成功后服务器端工作线将好友信息发送个客户端。
- 广播用户上线消息：当用户登录成功后，服务器向所有在线用户发送消息，即广播。
- 接收用户消息：用户在聊天时发送消息给服务器，服务器端工作线程一直不断地接收用户消息。
- 转发消息给好友：服务器端工作线程接收到用户发送的聊天信息，然后再将消息转发给好友。

采用用例分析方法描述服务器端用例图，如图30-2所示。

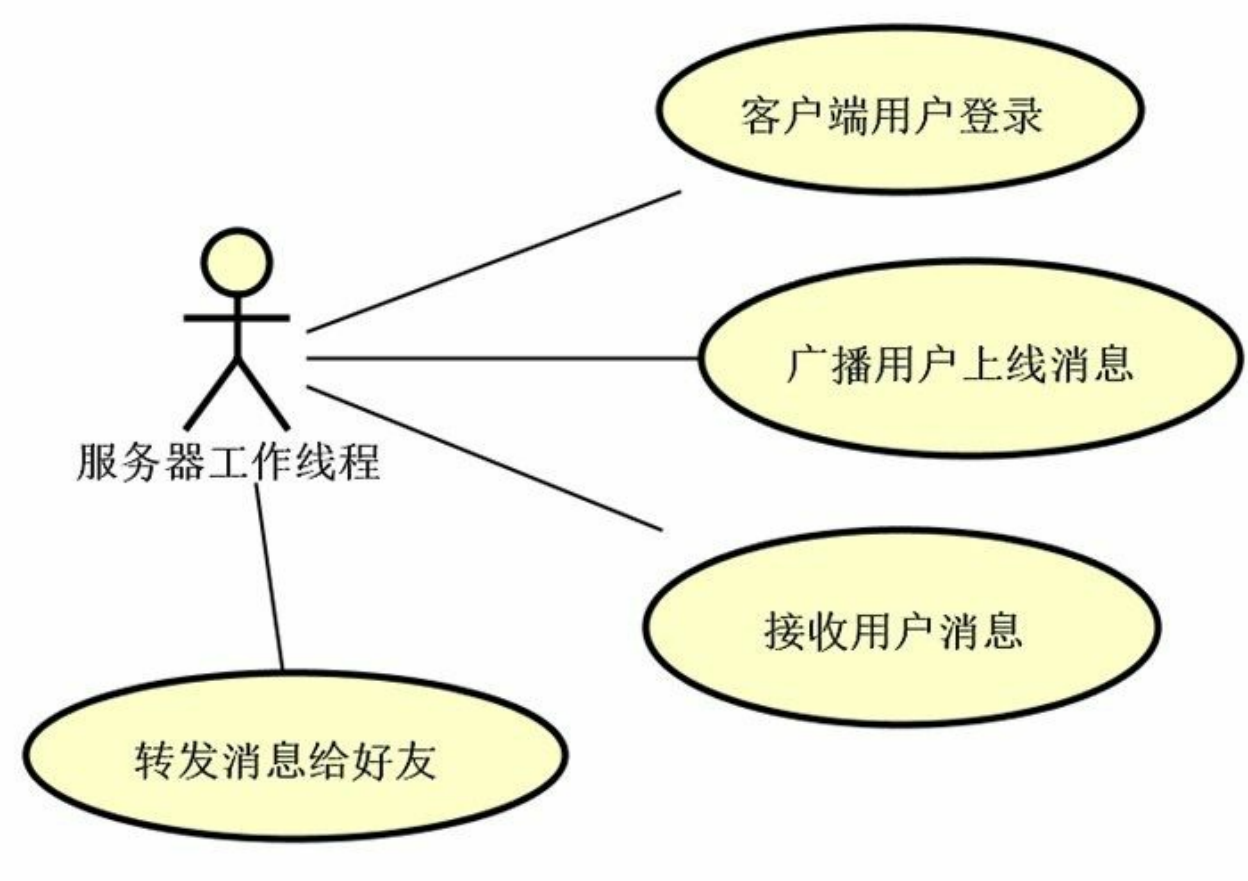


图**30-2　QQ2006**项目服务器端用例图

30.1.3　原型设计

服务器端没有界面，没有原型设计。客户端有原型设计，原型设计主要应用于图形界面应用程序，原型设计对于设计人员、开发人员、测试人员、UI设计人员以及用户都是非常重要的。QQ2006项目客户端原型设计图如图30-3所示。

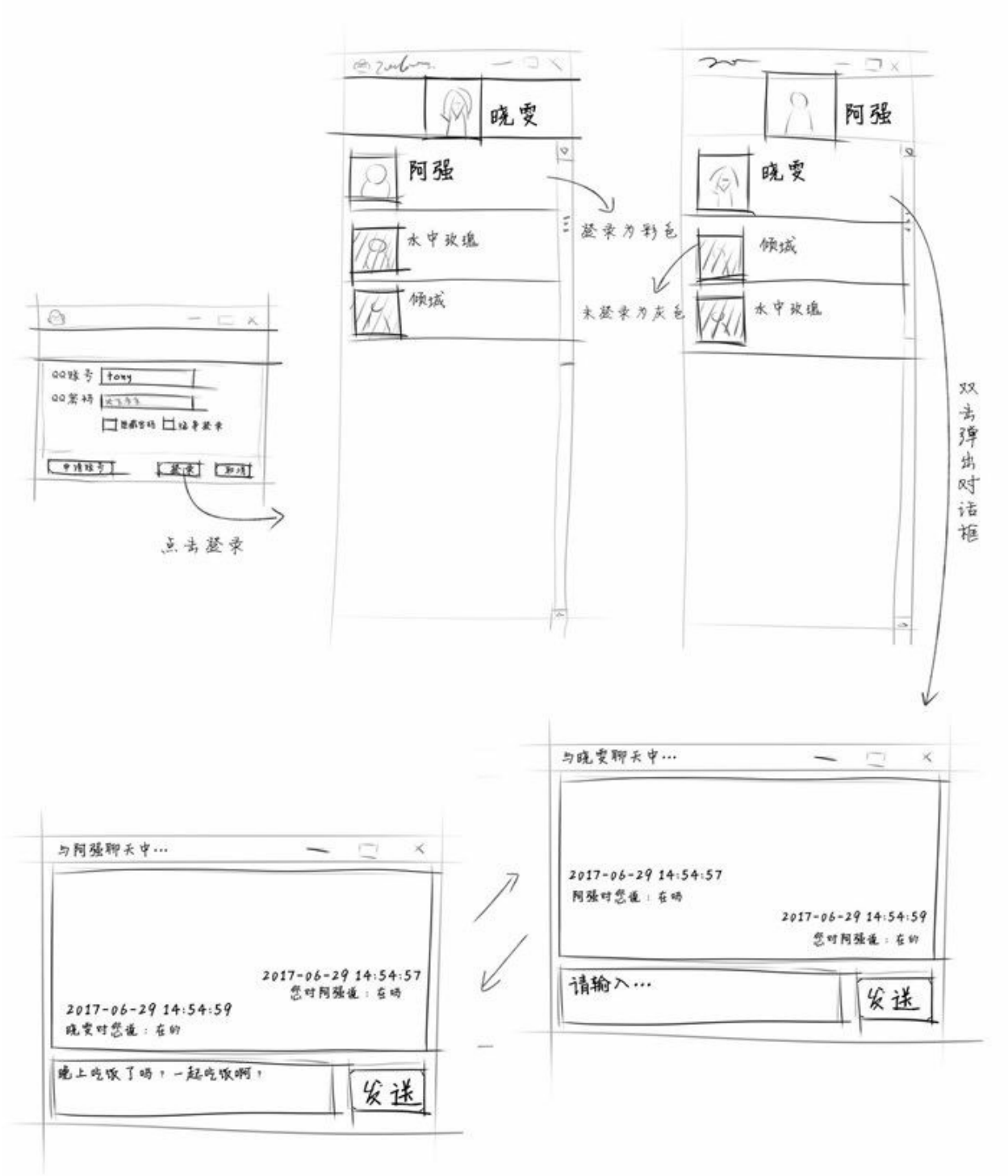


图**30-3　QQ2006**项目客户端原型设计图

30.1.4　数据库设计

QQ2006项目中客户端没有数据库，只有服务器端有数据库，服务器数据库设计如图30-4所示。

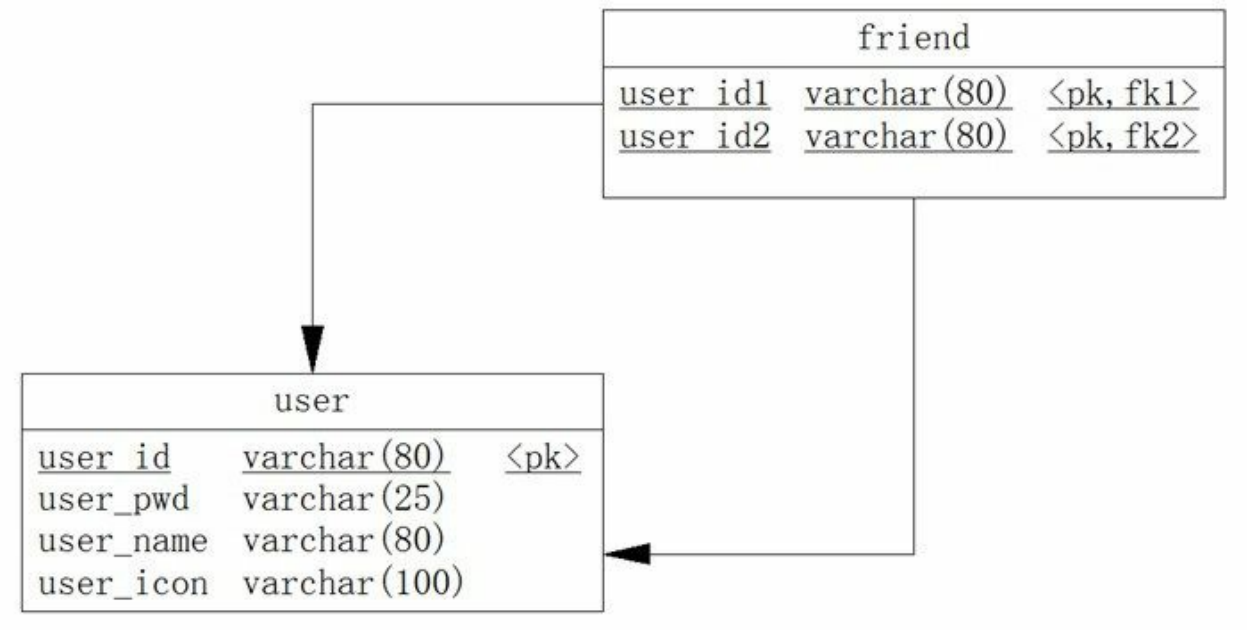


图**30-4**　数据库设计模型

数据库设计模型中各个表说明如下：

01. 用户表

用户表（英文名user）是QQ2006的注册用户，用户Id（英文名user_id）是主键，用户表结构如表30-1所示。

表 **30-1**　用户表

字段名	数据类型	长度	精度	主键	外键	备注
user_id	varchar(80)	80	-	是	否	用户 Id
user_pwd	varchar(25)	25	-	否	否	用户密码
user_name	varchar(80)	80	-	否	否	用户名
user_icon	varchar(100)	100	-	否	否	用户头像

02. 用户好友表

用户好友表（英文名friend）只有两个字段用户Id1和用户Id2，它们是用户好友的联合主键，给定一个用户Id1和用户Id2可以确定用户好友表中唯一一条数据，这是“主键约束”。用户好友表与用户表关系比较复杂，用户好友表的两个字段都引用到用户表用户Id字段，用户好友表中的用户Id1和用户Id2都是必须是用户表中存在的用户Id，这是“外键约束”，用户好友表结构如表30-2所示。

表 **30-2**　用户好友表

字段名	数据类型	长度	精度	主键	外键	备注
user_id1	varchar(80)	80	-	是	是	用户 Id1
user_id2	varchar(80)	80	-	是	是	用户 Id2

对于初学者理解用户好友表与用户表的关系有一定的困难，下面通过如图30-5所示进一步理解它们之间的关系。从图中可见用户好友表中的user_id1和user_id2数据都是用户表user_id存在的。

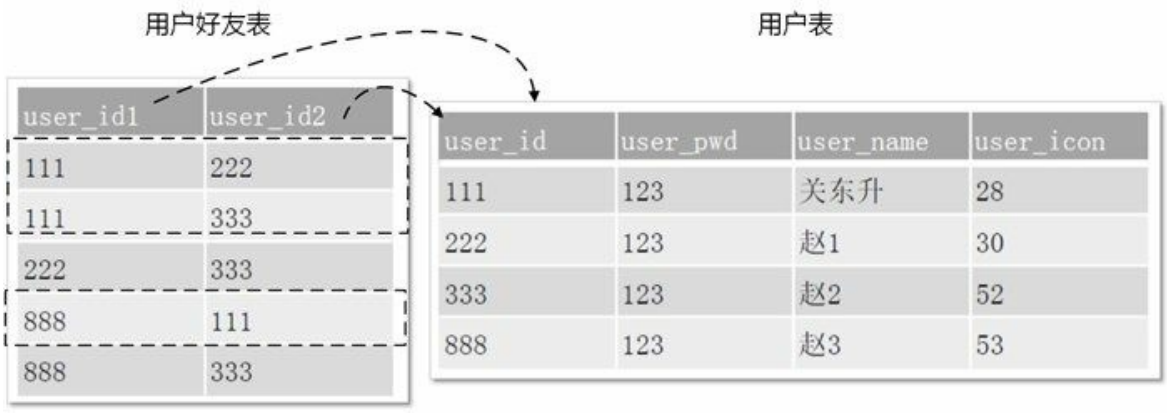


图**30-5**　用户好友表与用户表数据

那么用户111的好友应该有222、333和888，凡是好友表中user_id1或user_id2等于111的数据都是其好友。要想通过一条SQL语句查询出用户111的好友信息，可以多种写法，主要使用表连接或子查询实现，如下代码是笔者通过子查询实现SQL语句：

```
select user_id,user_pwd,user_name,user_icon FROM user
   WHERE user_id IN (select user_id2 as user_id  from friend where user_id1 = 111)
      OR user_id IN (select user_id1 as user_id  from friend where user_id2 = 111)
```

其中select user_id2 as user_id from friend where user_id1 = 111和select user_id1 as user_id from friend where user_id2 = 111是两个子查询，分别查询出好友表中user_id1 = 111的user_id2的数据和user_id2 = 111的user_id1的数据。

在MySQL数据库执行SQL语句，结果如图30-6所示。

图**30-6**　子查询实现**SQL**语句

30.1.5　网络拓扑图

QQ2006项目分为客户端和服务器，采用C/S（客户端/服务器）网络结构，如图30-7所示，服务器只有一个，客户端可以有多个。

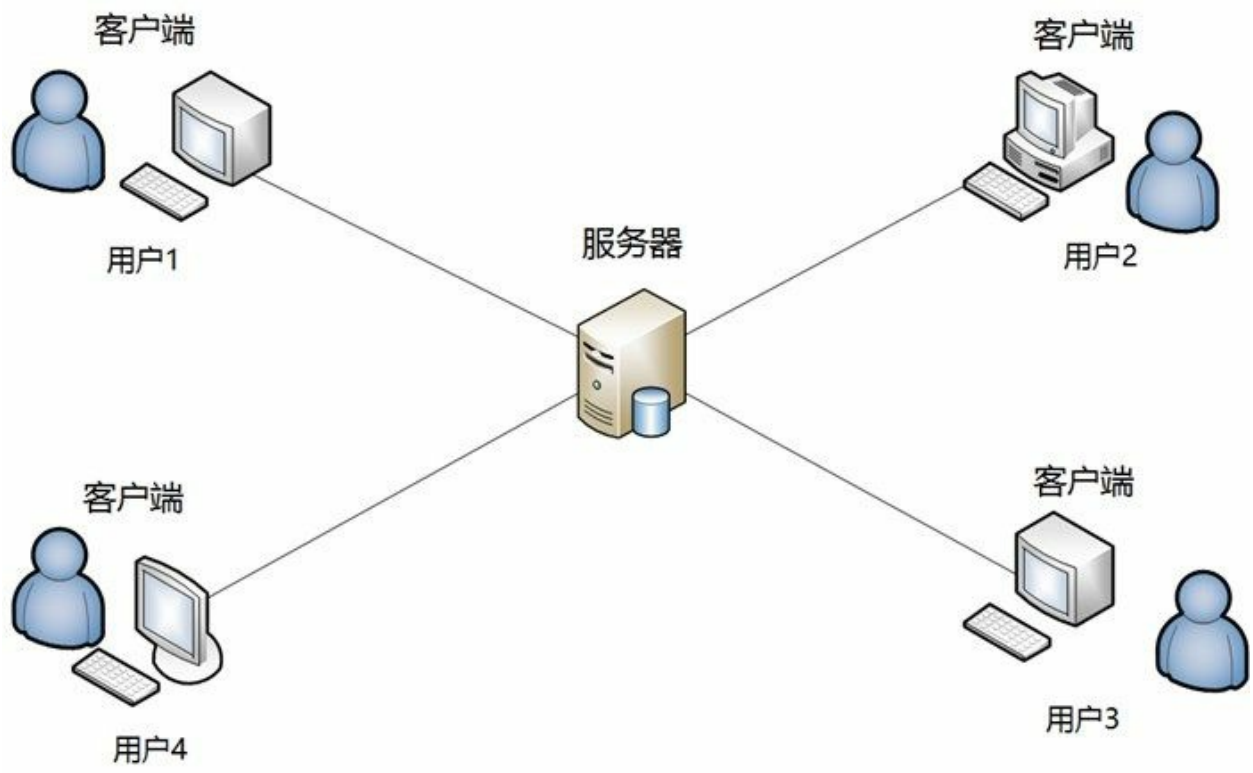


图**30-7**　**QQ2006**项目网络结构

30.1.6　系统设计

系统设计也分为客户端和服务器端。

01. 客户端系统设计

客户端系统设计如图30-8所示，客户端有很多三个窗口：用户登录窗口LoginFrame、好友列表窗口FriendsFrame和聊天窗口ChatFrame，其中CartFrame与FriendsFrame关联关系。Client是客户端启动类。

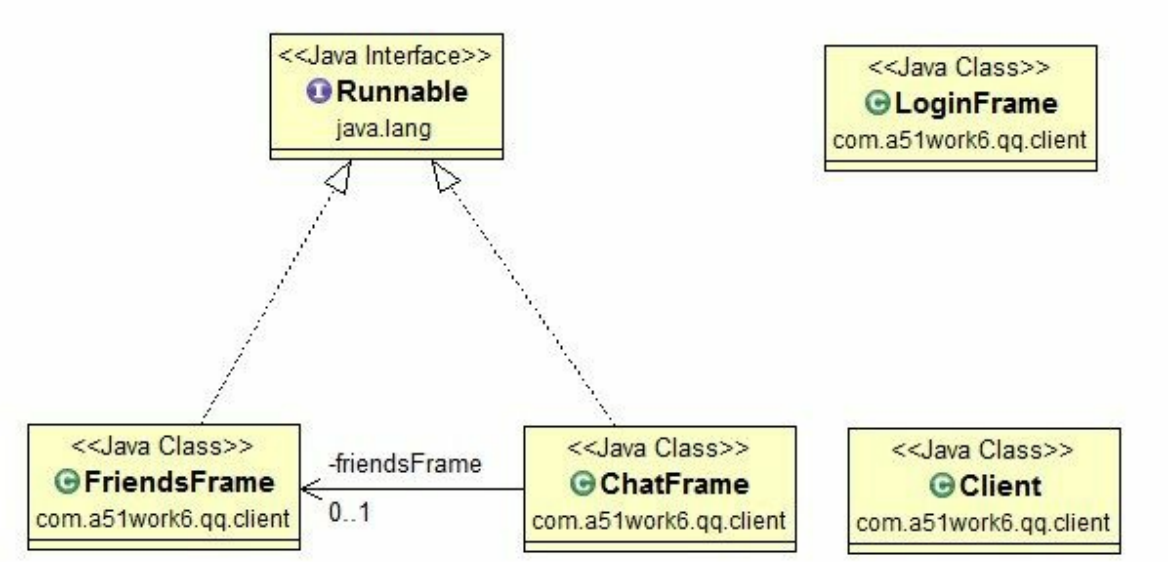


图**30-8**　客户端系统设计类图

02. 服务器端系统设计

服务器系统设计如图30-9所示，服务器端没有图像用户界面，服务器端4个类说明如下：

- Server：服务器端启动类。
- UserDAO：服务器端用户信息DAO类，用来操作数据库用户表。
- DBHelper：连接数据库辅助类。
- ClientInfo：服务器端保存客户端信息类，userId属性是客户Id、address属性客户端地址、port是客户端端口号。

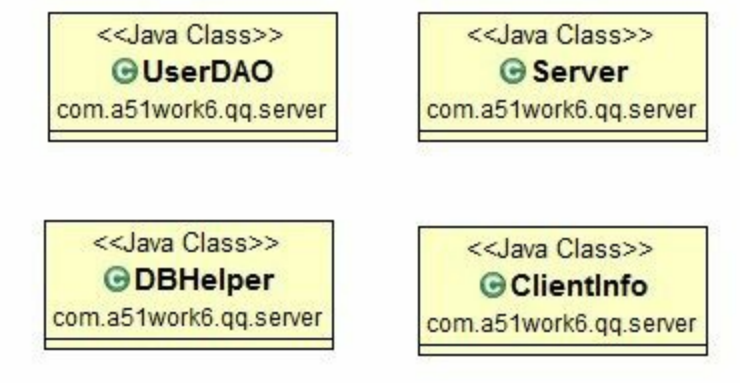


图**30-9**　服务器端系统设计类图

30.2 任务1：创建服务器端数据库

在设计完成之后，在编写Java代码之前，应该先创建服务器端数据库。

30.2.1 迭代1.1：安装和配置MySQL数据库

首先应该为开发该项目，准备好数据库。本书推荐使用MySQL数据库，如果没有安装MySQL数据库，可以参考28.2.1节安装MySQL数据库。

30.2.2 迭代1.2：编写数据库DDL脚本

按照图30-4所示的数据库设计模型编写数据库DDL脚本。当然，也可以通过一些工具生成DDL脚本，然后把这个脚本放在数据库中执行就可以了。下面是编写的DDL脚本：

```
/* 创建数据库 */
CREATE DATABASE  IF NOT EXISTS  qq;

use qq;

/* 用户表 */
CREATE TABLE IF NOT EXISTS user (
    user_id varchar(80) not null,        /* 用户Id  */
    user_pwd varchar(25)  not null,      /* 用户密码 */
    user_name varchar(80) not null,       /* 用户名 */
    user_icon varchar(100) not null,      /* 用户头像 */
PRIMARY KEY (user_id));


/* 用户好友表Id1和Id2互为好友 */
CREATE TABLE IF NOT EXISTS friend (
    user_id1 varchar(80) not null,        /* 用户Id1  */
    user_id2 varchar(80) not null,        /* 用户Id2  */
PRIMARY KEY (user_id1, user_id2));
```

如果读者对于编写DDL脚本不熟悉，可以直接使用笔者编写好的qq-mysql-schema-gbk.sql脚本文件，文件位于QQ2006项目下db目录中。

30.2.3 迭代1.3：插入初始数据到数据库

QQ2006项目服务器端有一些初始的数据，这些初始数据在创建数据库之后插入。这些插入数据的语句如下：

```
use qq;

/* 用户表数据 */
INSERT INTO user VALUES('111','123', '关东升', '28');
INSERT INTO user VALUES('222','123', '赵1', '30');
INSERT INTO user VALUES('333','123', '赵2', '52');
INSERT INTO user VALUES('888','123', '赵3', '53');

/* 用户好友表Id1和Id2互为好友 */
INSERT INTO friend VALUES('111','222');
INSERT INTO friend VALUES('111','333');
INSERT INTO friend VALUES('888','111');
INSERT INTO friend VALUES('222','333');
```

如果读者不愿意自己编写插入数据的脚本文件，可以直接使用笔者编写好的qq-mysql-dataload-gbk.sql

脚本文件，文件位于QQ2006项目下db目录中。

30.3 任务2：应用并初始化项目

本项目推荐使用Eclipse工具，所以首先参考3.1节创建一个Eclipse项目，项目名称QQ2006。

30.3.1 任务2.1：配置项目构建路径

QQ2006项目创建完成后，需要参考如图30-10所示，在QQ2006项目根目录下面创建普通文件夹db。然后将MySQL数据库JDBC驱动程序mysql-connector-java-5.xxx-bin.jar拷贝到db目录，参考28.3.2节将驱动程序文件添加到项目的构建路径中。resource/img文件夹中内容是项目使用的图片。

图**30-10 QQ2006**项目目录结构

30.3.2 任务2.2：添加资源图片

项目中会用到很多资源图片，为了打包发布项目方便，这些图片最好放到src源文件夹下，Eclipse会将该文件夹下有文件一起复制到字节码文件夹中。参考图30-10在src文件夹下创建resource/img文件夹，然后将本书配套资源中找到images中的图片，并复制到Eclipse项目的resource/img文件夹中。

30.3.3 任务2.3：添加**JSON-java**库

客户端与服务器之间数据交换采用JSON格式，JSON解码和编码库采用第三方的JSON-java库，参考24.4.2节添加JSON-java库到项目。

30.3.4 任务2.4：添加包

参考图30-10在src文件夹中创建如下3个包：

- com.a51work6.qq.client。放置客户端组件。
- com.a51work6.qq.server。放置服务器端组件。
- org.json。放置JSON-java库类。

30.4 任务3：编写服务器端外围代码

服务器端外围代码主要是涉及到UserDAO、DBHelper和ClientInfo等几个非通信类。

30.4.1 任务3.1：编写UserDAO类

UserDAO是操作数据库用户表的DAO对象，如图30-11所示是UserDAO详细类图。它有三个公有查询方法：

```
public List<Map<String, String>> findAll()
public Map<String, String> findById(String id)
public List<Map<String, String>> findFriends(String id)
```

findById通过用户ID查询用户信息，查询返回的数据在Map中，没有放到实体类中。findFriends是通过用户Id查询他的所有好友，返回的是List集合，其中的每一个元素都是Map。

图30-11 DAO实现类图

UserDAO代码如下：

```
package com.a51work6.qq.server;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class UserDAO {

    // 查询所有用户信息
    public List<Map<String, String>> findAll() {

        List<Map<String, String>> list = new ArrayList<Map<String, String>>();

        // SQL语句
        String sql = "select user_id,user_pwd,user_name, user_icon from user where user_id";
        try (// 2.创建数据库连接
                Connection conn = DBHelper.getConnection();
                // 3. 创建语句对象
                PreparedStatement pstmt = conn.prepareStatement(sql);
                // 5. 执行查询
```

```
            ResultSet rs = pstmt.executeQuery();) {

        // 6. 遍历结果集
        while (rs.next()) {

            Map<String, String> row = new HashMap<String, String>();
            row.put("user_id", rs.getString("user_id"));                ①
            row.put("user_name", rs.getString("user_name"));
            row.put("user_pwd", rs.getString("user_pwd"));
            row.put("user_icon", rs.getString("user_icon"));            ②

            list.add(row);
        }

    } catch (SQLException e) {
    }

    return list;
}


// 按照主键查询
public Map<String, String> findById(String id) {

    Connection conn = null;
    PreparedStatement pstmt = null;
    ResultSet rs = null;
    // SQL语句
    String sql = "select user_id,user_pwd,user_name, user_icon from user where user_id = ?";
    try {
        // 2.创建数据库连接
        conn = DBHelper.getConnection();
        // 3. 创建语句对象

        pstmt = conn.prepareStatement(sql);
        // 4. 绑定参数
        pstmt.setString(1, id);
        // 5. 执行查询（R）
        rs = pstmt.executeQuery();
        // 6. 遍历结果集
        if (rs.next()) {

            Map<String, String> row = new HashMap<String, String>();
            row.put("user_id", rs.getString("user_id"));                ③
            row.put("user_name", rs.getString("user_name"));
            row.put("user_pwd", rs.getString("user_pwd"));
            row.put("user_icon", rs.getString("user_icon"));            ④

            return row;
        }

    } catch (SQLException e) {
        e.printStackTrace();
    } finally { // 释放资源
        if (rs != null) {
            try {
                rs.close();
            } catch (SQLException e) {
            }
        }
        if (pstmt != null) {
            try {
                pstmt.close();
            } catch (SQLException e) {
            }
        }
```

```
        if (conn != null) {
            try {
                conn.close();
            } catch (SQLException e) {
            }
        }
    }

    return null;
}

// 查询好友 列表
public List<Map<String, String>> findFriends(String id) {

    Connection conn = null;
    PreparedStatement pstmt = null;
    ResultSet rs = null;
    List<Map<String, String>> friends = new ArrayList<Map<String, String>>();
    // SQL语句
    String sql = "select user_id,user_pwd,user_name,user_icon FROM user " + " WHERE "
            + "    user_id IN (select user_id2 as user_id  from friend where user_id1 = ?)"
            + " OR user_id IN (select user_id1 as user_id  from friend where user_id2 = ?)";
    try {
        // 2.创建数据库连接
        conn = DBHelper.getConnection();
        // 3. 创建语句对象

        pstmt = conn.prepareStatement(sql);
        // 4. 绑定参数
        pstmt.setString(1, id);
        pstmt.setString(2, id);
        // 5. 执行查询（R）
        rs = pstmt.executeQuery();
        // 6. 遍历结果集
        while (rs.next()) {

            Map<String, String> row = new HashMap<String, String>();
            row.put("user_id", rs.getString("user_id"));                ⑥
            row.put("user_name", rs.getString("user_name"));
            row.put("user_pwd", rs.getString("user_pwd"));
            row.put("user_icon", rs.getString("user_icon"));            ⑦

            friends.add(row);
        }

    } catch (SQLException e) {
        e.printStackTrace();
    } finally { // 释放资源
        if (rs != null) {
            try {
                rs.close();
            } catch (SQLException e) {
            }
        }
        if (pstmt != null) {
            try {
                pstmt.close();
            } catch (SQLException e) {
            }
        }
        if (conn != null) {
            try {
                conn.close();
            } catch (SQLException e) {
            }
        }
```

```
            }

            return friends;
        }

    }
```

在findAll()方法中代码第①行~第②行遍历结果集，并将数据从字段中取出，然后按照键值对放入到Map对象中。findById方法中代码第③行~第④行，findFriends方法中代码第⑥行~第⑦行的处理也是类似的。

代码第⑤行SQL语句使用了子查询，在前面数据库设计时已经介绍了这里不再赘述。

30.4.2 迭代3.2：数据库帮助类DBHelper

数据库帮助类DBHelper可以进行JDBC驱动程序加载以及获得数据库连接。如图30-12所示是DBHelper详细类图。它有两个静态变量和一个静态方法。

图**30-12** 数据库帮助类**DBHelper**类图

DBHelper具体实现代码如下：

```
package com.a51work6.qq.server;

import java.io.IOException;
import java.io.InputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.Properties;

public class DBHelper {

    // 连接数据库url
    static String url;
    // 创建Properties对象
    static Properties info = new Properties();

    // 1.驱动程序加载
    static {                                                    ①
        // 获得属性文件输入流
        InputStream input = DBHelper.class.getClassLoader()
                .getResourceAsStream("com/a51work6/c/server/config.properties");

        try {
            // 加载属性文件内容到Properties对象
            info.load(input);
```

```
                // 从属性文件中取出url
                url = info.getProperty("url");
                // Class.forName("com.mysql.jdbc.Driver");
                // 从属性文件中取出driver
                String driverClassName = info.getProperty("driver");
                Class.forName(driverClassName);
                System.out.println("驱动程序加载成功...");
            } catch (ClassNotFoundException e) {
                System.out.println("驱动程序加载失败...");
            } catch (IOException e) {
                System.out.println("加载属性文件失败...");
            }
        }                                                          ②

        public static Connection getConnection() throws SQLException {    ③
            // 创建数据库连接
            Connection conn = DriverManager.getConnection(url, info);
            return conn;
        }

    }
```

上述代码第①行~第②行通过静态代码库加载数据库驱动程序，并且在静态代码块中读取配置文件config.properties信息，该配置文件位于com.a51work6.qq.server包中，内容如下：

```
driver=com.mysql.jdbc.Driver
url=jdbc:mysql://localhost:3306/petstore
user=root
password=123456
useSSL=false
verifyServerCertificate=false
```

代码第③行提供了获得数据库连接方法，这是一个静态方法使用起来比较方便。

30.4.3 任务3.3：编写ClientInfo类

一个用户可以在任何一个客户端主机上登录，因此登录的客户端主机IP和端口号都是动态的。为了在服务器端保存所有登录的用户Id，以及登录的客户端主机地址和端口号信息，所以设计了ClientInfo类。如图30-13所示是ClientInfo详细类图。

图**30-13　ClientInfo**类图

ClientInfo代码如下：

```
package com.a51work6.qq.server;

import java.net.InetAddress;

public class ClientInfo {
    // 用户Id
    private String userId;
    // 客户端IP地址
    private InetAddress address;
    // 客户端端口号
    private int port;

    public String getUserId() {
        return userId;
    }

    public void setUserId(String userId) {
        this.userId = userId;
    }

    public InetAddress getAddress() {
        return address;
    }

    public void setAddress(InetAddress address) {
        this.address = address;
    }

    public int getPort() {
        return port;
    }

    public void setPort(int port) {
        this.port = port;
    }
}
```

注意为了方便客户端IP地址属性的类型是InetAddress，不是字符串。

30.5 任务4：客户端UI实现

从客观上讲，客户端UI实现开发的工作量是很大的，有很多细节工作需要完成。

30.5.1 迭代4.1：登录窗口实现

客户端启动马上显示用户登录窗口，界面如图30-14所示，界面中有很多组件，但是本例中主要使用文本框、密码框和两个按钮。等用户输入QQ号码和QQ密码，单击“登录”按钮，如果输入的账号和密码正确，则登录成功进入好友列表窗口；如果输入的不正确，则弹出如图30-15所示的对话框。

图**30-14** 登录窗口

图**30-15** 登录失败提示

登录窗口类是LoginFrame，它的类图如图30-16所示。

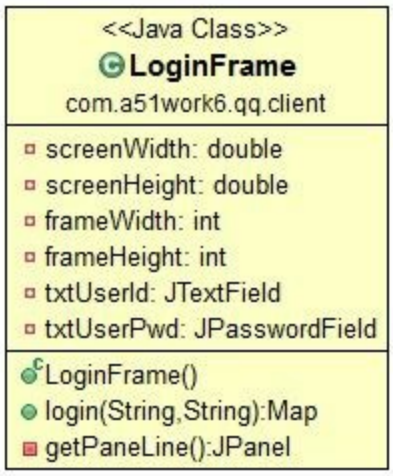


图**30-16**　登录窗口类图

用户登录窗口LoginFrame代码如下：

```
package com.a51work6.qq.client;

...

public class LoginFrame extends JFrame {

    // private Client client;
    // 获得当前屏幕的宽和高
    private double screenWidth = Toolkit.getDefaultToolkit().getScreenSize().getWidth();      ①
    private double screenHeight = Toolkit.getDefaultToolkit().getScreenSize().getHeight();    ②

    // 登录窗口宽和高
    private int frameWidth = 329;
    private int frameHeight = 250;

    // QQ号码文本框
    private JTextField txtUserId = null;
    // QQ密码框
    private JPasswordField txtUserPwd = null;

    public LoginFrame() {

        JLabel lblImage = new JLabel();
        lblImage.setIcon(new ImageIcon(LoginFrame.class.getResource("/resource/img/QQll.JPG")));
        lblImage.setBounds(0, 0, 325, 48);
        getContentPane().add(lblImage);

        // 添加蓝线面板
        getContentPane().add(getPaneLine());

        // 初始化登录按钮
        JButton btnLogin = new JButton();
        btnLogin.setBounds(152, 181, 63, 19);
        btnLogin.setFont(new Font("Dialog", Font.PLAIN, 12));
        btnLogin.setText("登录");
        getContentPane().add(btnLogin);
        // 注册登录按钮事件监听器
        btnLogin.addActionListener(e -> {                    ③

            //TODO 登录处理
```

```
        });

        // 初始化取消按钮
        JButton btnCancel = new JButton();
        btnCancel.setBounds(233, 181, 63, 19);
        btnCancel.setFont(new Font("Dialog", Font.PLAIN, 12));
        btnCancel.setText("取消");
        getContentPane().add(btnCancel);
        btnCancel.addActionListener(e -> {
            // 退出系统
            System.exit(0);
        });

        // 初始化【申请号码↓】按钮
        JButton btnSetup = new JButton();
        btnSetup.setBounds(14, 179, 99, 22);
        btnSetup.setFont(new Font("Dialog", Font.PLAIN, 12));
        btnSetup.setText("申请号码↓");
        getContentPane().add(btnSetup);

        /// 初始化当前窗口
        setIconImage(Toolkit.getDefaultToolkit()                ④
                .getImage(Client.class.getResource("/resource/img/QQ.png")));
        setTitle("QQ登录");
        setResizable(false);
        getContentPane().setLayout(null);
        // 设置窗口大小
        setSize(frameWidth, frameHeight);
        // 计算窗口位于屏幕中心的坐标
        int x = (int) (screenWidth - frameWidth) / 2;
        int y = (int) (screenHeight - frameHeight) / 2;
        // 设置窗口位于屏幕中心
        setLocation(x, y);                                      ⑤

        // 注册窗口事件
        addWindowListener(new WindowAdapter() {                 ⑥
            // 单击窗口关闭按钮时调用
            public void windowClosing(WindowEvent e) {
                // 退出系统
                System.exit(0);
            }
        });
    }


    // 蓝线面板
    private JPanel getPaneLine() {                              ⑦

        JPanel paneLine = new JPanel();
        paneLine.setLayout(null);
        paneLine.setBounds(7, 54, 308, 118);
        // 边框颜色设置为蓝色
        paneLine.setBorder(BorderFactory.createLineBorder(new Color(102, 153, 255), 1));

        // 初始化【忘记密码？】标签
        JLabel lblHelp = new JLabel();
        lblHelp.setBounds(227, 47, 67, 21);
        lblHelp.setFont(new Font("Dialog", Font.PLAIN, 12));
        lblHelp.setForeground(new Color(51, 51, 255));
        lblHelp.setText("忘记密码？");
        paneLine.add(lblHelp);

        // 初始化【QQ密码】标签
        JLabel lblUserPwd = new JLabel();
        lblUserPwd.setText("QQ密码");
        lblUserPwd.setFont(new Font("Dialog", Font.PLAIN, 12));
```

```
            lblUserPwd.setBounds(21, 48, 54, 18);
            paneLine.add(lblUserPwd);

            // 初始化【QQ号码↓】标签
            JLabel lblUserId = new JLabel();
            lblUserId.setText("QQ号码↓");
            lblUserId.setFont(new Font("Dialog", Font.PLAIN, 12));
            lblUserId.setBounds(21, 14, 55, 18);
            paneLine.add(lblUserId);

            // 初始化【QQ号码】文本框
            this.txtUserId = new JTextField();
            this.txtUserId.setBounds(84, 14, 132, 18);
            paneLine.add(this.txtUserId);

            // 初始化【QQ密码】密码框
            this.txtUserPwd = new JPasswordField();
            this.txtUserPwd.setBounds(84, 48, 132, 18);
            paneLine.add(this.txtUserPwd);

            // 初始化【自动登录】复选框
            JCheckBox chbAutoLogin = new JCheckBox();
            chbAutoLogin.setText("自动登录");
            chbAutoLogin.setFont(new Font("Dialog", Font.PLAIN, 12));
            chbAutoLogin.setBounds(79, 77, 73, 19);
            paneLine.add(chbAutoLogin);

            // 初始化【隐身登录】复选框
            JCheckBox chbHideLogin = new JCheckBox();
            chbHideLogin.setText("隐身登录");
            chbHideLogin.setFont(new Font("Dialog", Font.PLAIN, 12));
            chbHideLogin.setBounds(155, 77, 73, 19);
            paneLine.add(chbHideLogin);

            return paneLine;
        }

    }
```

上述代码第①行和第②行是获取当前屏幕的宽和高，用来介绍当前窗口屏幕居中的坐标。具体的原理在25.5.7节已经介绍过程，这里不再赘述。

代码第③行是用户单击登录按钮之后的处理，本节暂时不介绍具体实现过程，后面介绍登录处理时在详细说明。

代码第④行~第⑤行的初始化登录窗口，包括设置窗口图标，窗口大小和位置等内容。代码第⑥行注册窗口事件，当用户单击窗口的关闭按钮时调用System.exit(0)语句退出系统。

代码第⑦行的getPaneLine()方法用来初始化“蓝线面板”，蓝线面板如图30-17所示的虚线部分，其中包括：一个文本框、一个密码框、两个复选框和三个标签。

图**30-17**　登录窗口中的蓝线面板

30.5.2　迭代**4.2**：好友列表窗口实现

在客户端用户登录成功之后，界面会跳转到好友列表窗口，界面如图30-18所示。

图**30-18**　好友列表窗口

好友列表窗口类是FriendsFrame，它的类图如图30-19所示。

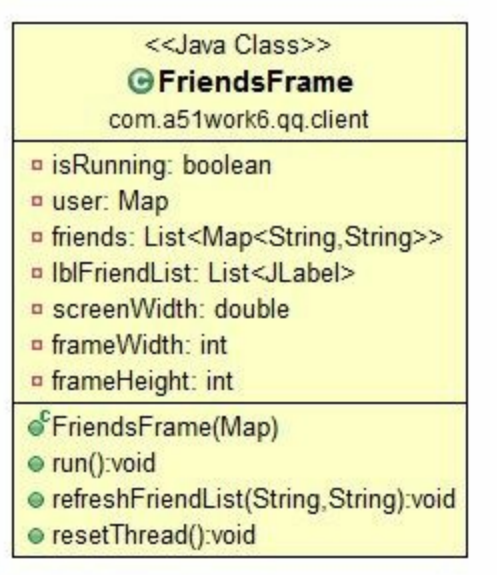


图**30-19**　好友列表窗口类图

好友列表窗口类FriendsFrame代码如下：

```
package com.a51work6.qq.client;

...

public class FriendsFrame extends JFrame implements Runnable {

    // 线程运行状态
    private boolean isRunning = true;
    // 用户信息
    private Map user;
    // 好友列表
    private List<Map<String, String>> friends;
    // 好友标签控件列表
    private List<JLabel> lblFriendList;

    // 获得当前屏幕的宽
    private double screenWidth = Toolkit.getDefaultToolkit().getScreenSize().getWidth();

    // 登录窗口宽高
    private int frameWidth = 260;
    private int frameHeight = 600;

    public FriendsFrame(Map user) {
        setTitle("QQ2006");

        // 初始化成员变量
        this.user = user;
        /// 初始化用户列表
        this.friends = (List<Map<String, String>>) user.get("friends");

        // 设置布局
        BorderLayout borderLayout = (BorderLayout) getContentPane().getLayout();
        borderLayout.setVgap(5);

        String userId = (String) user.get("user_id");
        String userName = (String) user.get("user_name");
        String userIcon = (String) user.get("user_icon");
```

```
JLabel lblLabel = new JLabel(userName);
lblLabel.setHorizontalAlignment(SwingConstants.CENTER);
String iconFile = String.format("/resource/img/%s.jpg", userIcon);
lblLabel.setIcon(new ImageIcon(FriendsFrame.class.getResource(iconFile)));
getContentPane().add(lblLabel, BorderLayout.NORTH);

JScrollPane scrollPane = new JScrollPane();
scrollPane.setBorder(BorderFactory.createLineBorder(Color.blue, 1));

getContentPane().add(scrollPane, BorderLayout.CENTER);

JPanel panel1 = new JPanel();
scrollPane.setViewportView(panel1);
panel1.setLayout(new BorderLayout(0, 0));

JLabel label = new JLabel("我的好友");
label.setHorizontalAlignment(SwingConstants.CENTER);
panel1.add(label, BorderLayout.NORTH);

// 好友列表面板
JPanel friendListPanel = new JPanel();
panel1.add(friendListPanel);
friendListPanel.setLayout(new GridLayout(50, 0, 0, 5));

lblFriendList = new ArrayList<JLabel>();                    ①
// 初始化好友列表
for (int i = 0; i < friends.size(); i++) {                  ②
    Map<String, String> friend = this.friends.get(i);
    String friendUserId = friend.get("user_id");
    String friendUserName = friend.get("user_name");
    String friendUserIcon = friend.get("user_icon");
    // 获得好友在线状态
    String friendUserOnline = friend.get("online");

    JLabel lblFriend = new JLabel(friendUserName);          ③
    lblFriend.setToolTipText(friendUserId);                 ④
    String friendIconFile = String.format("/resource/img/%s.jpg", friendUserIcon);
    lblFriend.setIcon(new ImageIcon(FriendsFrame.class
                                .getClass().getResource(friendIconFile)));
    // 在线设置可用，离线设置不可用
    if (friendUserOnline.equals("0")) {                     ⑤
        lblFriend.setEnabled(false);
    } else {
        lblFriend.setEnabled(true);
    }

    lblFriend.addMouseListener(new MouseAdapter() {         ⑥
        @Override
        public void mouseClicked(MouseEvent e) {
            // 用户图标双击鼠标时显示对话框
            if (e.getClickCount() == 2) {
                ChatFrame chatFrame = new ChatFrame(FriendsFrame.this, user, friend);
                chatFrame.setVisible(true);
                isRunning = false;
            }
        }
    });
    // 添加到列表集合
    lblFriendList.add(lblFriend);
    // 添加到面板
    friendListPanel.add(lblFriend);
}

/// 初始化当前Frame
setBounds((int) screenWidth - 300, 10, frameWidth, frameHeight);
setIconImage(Toolkit.getDefaultToolkit().getImage(FriendsFrame.class
```

```
                                        .getResource("/resource/img/QQ.png")));

        // 注册窗口事件
        addWindowListener(new WindowAdapter() {
            // 单击窗口关闭按钮时调用
            public void windowClosing(WindowEvent e) {            ⑦

                //TODO 用户下线

                // 退出系统
                System.exit(0);
            }
        });

    }

    //TODO 启动接收消息子线程
    //TODO 刷新好友列表
}
```

上述代码第①行实例化lblFriendList对象，它保存了好友标签组件集合。代码第②行通过循环初始化好友列表，显示的好友名和图标事实上是一个标签组件（JLabel），代码第③行是创建标签组，显示的内容是好友名。代码第④行lblFriend.setToolTipText(friendUserId)是设置好友的Id，标签的setToolTipText方法设置ToolTipText属性，该属性是当鼠标放到标签上时弹出的气泡。代码第⑤行是设置好友标签是否可用，好友在线可用，好友离线不可用。代码第⑥行是为好友标签注册鼠标双击事件。

代码第⑦行是窗口关闭时调用，在该方法中进行用户下线处理。

另外，有关用户下线、启动接收消息子线程和刷新好友列表，这些处理会在后面再详细介绍。

30.5.3 迭代4.3：聊天窗口实现

在客户端用户双击好友列表中的好友，会弹出好友聊天窗口，界面如图30-20(a)所示，在这里可以给好友发送聊天信息，可以接收好友回复的信息，如图30-20(b)所示。

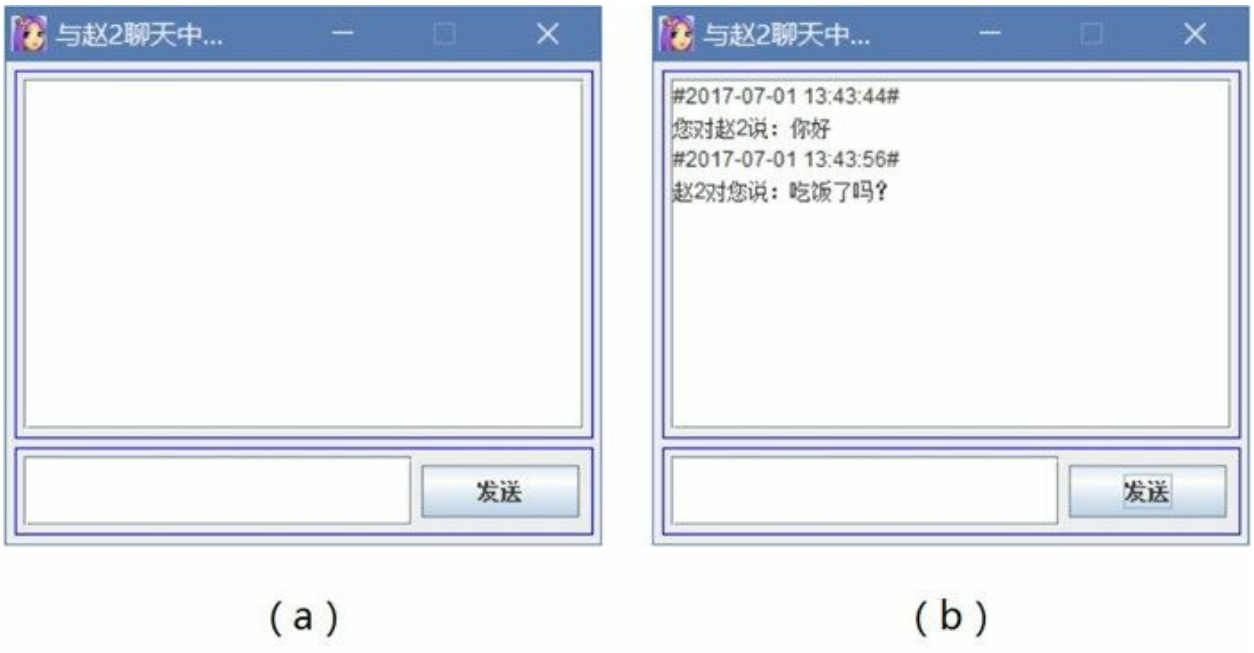


图**30-20**　聊天窗口

聊天窗口类是ChatFrame，它的类图如图30-21所示。

图**30-21**　聊天窗口类图

聊天窗口类ChatFrame代码如下：

```
package com.a51work6.qq.client;
...
public class ChatFrame extends JFrame implements Runnable {

    private boolean isRunning = true;

    // 当前用户Id
    private String userId;
    // 聊天好友用户Id
    private String friendUserId;
    // 聊天好友用户Id
    private String friendUserName;

    // 获得当前屏幕的高和宽
    private double screenHeight = Toolkit.getDefaultToolkit().getScreenSize().getHeight();
    private double screenWidth = Toolkit.getDefaultToolkit().getScreenSize().getWidth();

    // 登录窗口宽和高
    private int frameWidth = 345;
    private int frameHeight = 310;
```

```
// 查看消息文本区
private JTextArea txtMainInfo;
// 发送消息文本区
private JTextArea txtInfo;
// 消息日志
private StringBuffer infoLog;

// 接收消息子线程
private Thread receiveMessageThread;
// 日期格式化
private DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
// 好友列表Frame
private FriendsFrame friendsFrame;

public ChatFrame(FriendsFrame friendsFrame,
                 Map<String, String> user, Map<String, String> friend) {

    // 初始化成员变量
    this.friendsFrame = friendsFrame;

    this.userId = user.get("user_id");
    String userIcon = user.get("user_icon");

    this.friendUserId = friend.get("user_id");
    this.friendUserName = friend.get("user_name");

    this.infoLog = new StringBuffer();

    // 初始化查看消息面板
    getContentPane().add(getPanLine1());
    // 初始化发送消息面板
    getContentPane().add(getPanLine2());

    /// 初始化当前Frame
    String iconFile = String.format("/resource/img/%s.jpg", userIcon);
    setIconImage(Toolkit.getDefaultToolkit().getImage(Client.class.getResource(iconFile)));
    String title = String.format("与%s聊天中...", friendUserName);
    setTitle(title);
    setResizable(false);
    getContentPane().setLayout(null);

    // 设置Frame大小
    setSize(frameWidth, frameHeight);
    // 计算Frame位于屏幕中心的坐标
    int x = (int) (screenWidth - frameWidth) / 2;
    int y = (int) (screenHeight - frameHeight) / 2;
    // 设置Frame位于屏幕中心
    setLocation(x, y);

    receiveMessageThread = new Thread(this);
    receiveMessageThread.start();

    // 注册窗口事件
    addWindowListener(new WindowAdapter() {
        // 单击窗口关闭按钮时调用
        public void windowClosing(WindowEvent e) {            ①
            isRunning = false;
            setVisible(false);
            // 重启好友列表线程
            friendsFrame.resetThread();
        }
    });
}

// 查看消息面板
```

```
        private JPanel getPanLine1() {

            txtMainInfo = new JTextArea();
            txtMainInfo.setEditable(false);

            JScrollPane scrollPane = new JScrollPane();
            scrollPane.setBounds(5, 5, 320, 200);
            scrollPane.setViewportView(txtMainInfo);

            JPanel panLine1 = new JPanel();
            panLine1.setLayout(null);
            panLine1.setBounds(new Rectangle(5, 5, 330, 210));
            panLine1.setBorder(BorderFactory.createLineBorder(Color.blue, 1));
            panLine1.add(scrollPane);

            return panLine1;
        }

        // 发送消息面板
        private JPanel getPanLine2() {

            JPanel panLine2 = new JPanel();
            panLine2.setLayout(null);
            panLine2.setBounds(5, 220, 330, 50);
            panLine2.setBorder(BorderFactory.createLineBorder(Color.blue, 1));
            panLine2.add(getSendButton());

            JScrollPane scrollPane = new JScrollPane();
            scrollPane.setBounds(5, 5, 222, 40);
            panLine2.add(scrollPane);

            txtInfo = new JTextArea();
            scrollPane.setViewportView(txtInfo);

            return panLine2;
        }

        private JButton getSendButton() {

            JButton button = new JButton("发送");
            button.setBounds(232, 10, 90, 30);
            button.addActionListener(e -> {
                sendMessage();
                txtInfo.setText("");
            });
            return button;
        }

        private void sendMessage() {

            if (!txtInfo.getText().equals("")) {

                // TODO 发送消息
            }
        }

        // TODO 接收消息
    }
```

用户关闭聊天窗口并不退出系统，见代码第①行，只是停止当前窗口中的线程，隐藏当前窗口，回到好友列表界面，并重启好友列表线程。

提示　线程的使用原则，当前窗口中启动的线程，在窗口退出时、隐藏时是一定停止线程。

另外，发送消息和接收消息后面会详细介绍。

30.6 任务5：用户登录过程实现

用户登录时客户端和服务器互相交互，客户端和服务器端代码比较复杂，涉及到多线程编程。用户登录过程如图30-22所示，当用户1打开登录对话框，输入QQ号码和QQ密码，单击登录按钮，用户登录过程开始：

第①步：用户1登录。客户端将QQ号码和QQ密码数据封装发给服务器。

第②步：服务器接收用户1请求，验证用户1的QQ号码和QQ密码，是否与数据库的QQ号码和QQ密码一致。

第③步：返回给用户1登录结果。服务器端将登录结果发给客户端。

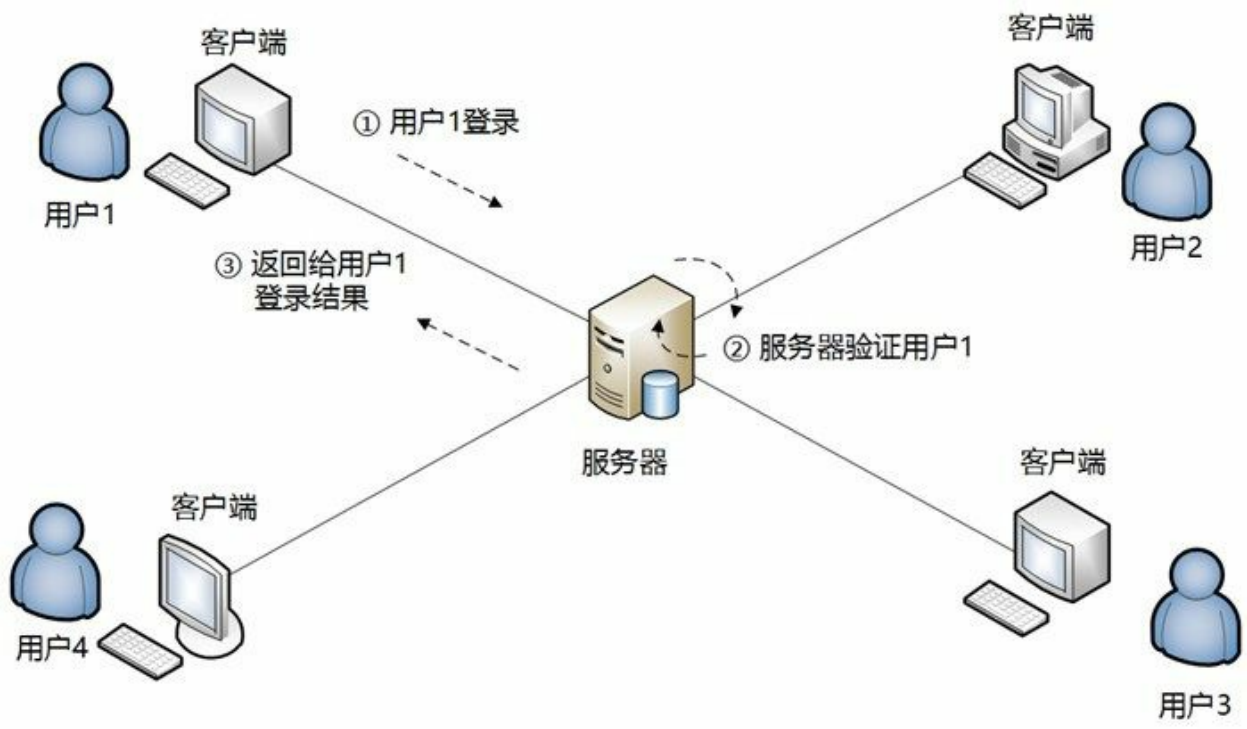


图30-22 用户登录过程

30.6.1 迭代5.1：客户端启动

在介绍客户端登录编程之前，首先介绍客户端启动程序Client。Client.java代码如下：

```
package com.a51work6.qq.client;

import java.io.IOException;
import java.net.DatagramSocket;

public class Client {

    // 命令代码
    public static final int COMMAND_LOGIN = 1;      // 登录命令      ①
    public static final int COMMAND_LOGOUT = 2;     // 注销命令
    public static final int COMMAND_SENDMSG = 3;    // 发消息命令    ②
```

```
    public static DatagramSocket socket;                    ③
    // 服务器端IP
    public static String SERVER_IP = "192.168.1.113";       ④
    // 服务器端端口号
    public static int SERVER_PORT = 7788;                   ⑤

    public static void main(String[] args) {

        if (args.length == 2) {
            SERVER_IP = args[0];                            ⑥
            SERVER_PORT = Integer.parseInt(args[1]);        ⑦
        }

        try {// 创建DatagramSocket对象，由系统分配可以使用的端口
            socket = new DatagramSocket();                  ⑧
            // 设置超时5秒，不再等待接收数据
            socket.setSoTimeout(5000);                      ⑨
            System.out.println("客户端运行...");
            LoginFrame frame = new LoginFrame();            ⑩
            frame.setVisible(true);

        } catch (IOException e) {
            e.printStackTrace();
        }

    }
}
```

上述代码第①行~第②行定义了三个命令代码常量，客户端与服务器端都定义了这三个命令代码常量，服务器端根据客户端的命令代码，获知客户端请求的意图，然后再进一步处理。

代码第③行声明了一个公有的静态数据报Socket变量socket，代码第⑧行实例化数据报Socket对象socket。

注意　socket对象一直没有关闭，这是因为socket对象的生命周期是整个Client应用程序。在这些Client应用程序中有很多线程，一直是使用socket对象发送和接收数据，因此不能关闭socket对象。只有Client应用程序停止socket对象才关闭。

代码第④行是声明服务器端IP地址，由于IP地址硬编码并不是好的做法，可以在运行Client时，通过main主方法的args参数传递进来，见代码第⑥行。

代码第⑤行是声明服务器端口号，端口号也可以通过main主方法的args参数传递进来，见代码第⑦行。

代码第⑨行是设置socket对象超时时间，数据报Socket的receive方法会导致线程阻塞，客户端有一个子线程一直在调用receive方法接收来自于服务器的数据，有时服务器会没有数据返回，如果不设置超时，那么客户端接收线程一直会被阻塞，设置了超时后，接收线程只对待5秒钟。

代码第⑩行调用LoginFrame启动登录窗口。

30.6.2　迭代5.2：客户端登录编程

LoginFrame在客户端登录编程代码如下；

```
package com.a51work6.qq.client;
...
public class LoginFrame extends JFrame {
    ...
    public LoginFrame() {
```

```
    ...

    // 注册登录按钮事件监听器
    btnLogin.addActionListener(e -> {

        // 先进行用户输入验证，验证通过再登录
        String userId = txtUserId.getText();
        String password = new String(txtUserPwd.getPassword());

        Map user = login(userId, password);        ①

        if (user != null) {                        ②
            // 登录成功调转界面
            System.out.println("登录成功调转界面");
            // 设置登录窗口可见
            this.setVisible(false);
            FriendsFrame frame = new FriendsFrame(user);
            frame.setVisible(true);
        } else {
            JOptionPane.showMessageDialog(null, "您QQ号码或密码不正确");
        }

    });
    ...
}

// 客户端向服务器发送登录请求
public Map login(String userId, String password) {

    // 准备一个缓冲区
    byte[] buffer = new byte[1024];
    InetAddress address;

    try {
        address = InetAddress.getByName(Client.SERVER_IP);

        JSONObject jsonObj = new JSONObject();           ③
        jsonObj.put("command", Client.COMMAND_LOGIN);    ④
        jsonObj.put("user_id", userId);
        jsonObj.put("user_pwd", password);
        // 字节数组
        byte[] b = jsonObj.toString().getBytes();
        // 创建DatagramPacket对象
        DatagramPacket packet = new DatagramPacket(b,
                        b.length, address, Client.SERVER_PORT);    ⑤
        // 发送
        Client.socket.send(packet);                      ⑥

        /* 接收数据报 */
        packet = new DatagramPacket(buffer, buffer.length,
                    address, Client.SERVER_PORT);
        Client.socket.receive(packet);                   ⑦
        // 接收数据长度
        int len = packet.getLength();
        String str = new String(buffer, 0, len);
        System.out.println("receivedjsonObj = " + str);
        JSONObject receivedjsonObj = new JSONObject(str);

        if ((Integer) receivedjsonObj.get("result") == 0) {        ⑧
            Map user = receivedjsonObj.toMap();                    ⑨
            return user;
        }

    } catch (IOException e) {
        e.printStackTrace();
    }
```

```
        return null;
    }
}
```

上述代码第③行~第⑥行是客户端向服务器发送登录请求。代码第③行创建JSON对象，它保存了发送给服务器端的数据。代码第④行将登录命令放入JSON对象，另外还将QQ号码和QQ密码放入JSON对象中，客户端发给服务器JSON对象内容如下：

```
{
    "user_id": "111",             //QQ号码
    "user_pwd": "123",            //QQ密码
    "command": 1                  //命令 1为登录
}
```

代码第⑤行是创建数据包对象，JSON对象编码后将数据包中。代码第⑥行是发送数据给指定服务器。

到处为止用户发送登录请求给服务器，完了图30-22中所示的第①步操作。

代码第⑦行客户端调用socket对象的receive()方法等待服务器端应答。服务器端返回数据给客户端，代码第⑧行判断JSON对象中的result键是否等于0。代码第⑨行将JSON对象转换为Map到对象。

从服务器端返回的JSON对象示例如下：

```
{
    "result": 0,                          // 登录结果 0登录成功 -1登录失败
    "user_icon": "52",
    "user_pwd": "123",
    "user_id": "333",
    "user_name": "赵2",
    "friends": [                          //该用户的好友列表
        {
            "online": "1",                //好友在线状态 1为在线 0为离线
            "user_icon": "28",
            "user_pwd": "123",
            "user_id": "111",
            "user_name": "关东升"
        },
        {
            "online": "1",
            "user_icon": "30",
            "user_pwd": "123",
            "user_id": "222",
            "user_name": "赵1"
        },
        {
            "online": "0",
            "user_icon": "53",
            "user_pwd": "123",
            "user_id": "888",
            "user_name": "赵3"
        }
    ]
}
```

到此为止完了图30-22中所示的第③步操作。如果用户登录成功login方法会返回非空数据，登录失败login方法返回空。

上述代码第②行判断login方法返回值是否为空，如果为非空登录成功则显示FriendsFrame窗口。

30.6.3 迭代5.3：服务器启动

在介绍服务器端编程之前，首先介绍服务器端启动程序Server。Server.java代码如下：

```
package com.a51work6.qq.server;
...
public class Server {

    // 命令代码
    public static final int COMMAND_LOGIN = 1; // 登录命令          ①
    public static final int COMMAND_LOGOUT = 2; // 注销命令
    public static final int COMMAND_SENDMSG = 3; // 发消息命令     ②

    public static int SERVER_PORT = 7788;                       ③
    // 所有已经登录的客户端信息
    private static List<ClientInfo> clientList
                = new CopyOnWriteArrayList<ClientInfo>();     ④

    // 创建数据访问对象
    static UserDAO dao = new UserDAO();                         ⑤

    public static void main(String[] args) {

        if (args.length == 1) {
            SERVER_PORT = Integer.parseInt(args[0]);          ⑥
        }

        System.out.printf("服务器启动，监听自己的端口%d...\n", SERVER_PORT);

        byte[] buffer = new byte[2048];

        try ( // 创建DatagramSocket对象，监听自己的端口7788
                DatagramSocket socket = new DatagramSocket(SERVER_PORT)) {     ⑦

                while (true) {                                  ⑧

                    //TODO 服务器端处理
                }

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

上述代码第①行~第②行定义了三个命令代码常量，与客户端都定义的三个命令代码保持一致。

代码第③行声明了一个公有的静态端口号，端口号也可以通过main主方法的args参数传递进来，见代码第⑥行。

代码第④行创建List集合对象clientList，用来保存所有登录的客户端信息。

> 注意　List的实例是CopyOnWriteArrayList对象，CopyOnWriteArrayList是线程安全的List对象，ArrayList是非线程安全。CopyOnWriteArrayList使用了一种叫写入时复制（英语：Copy-on-write，简称COW）的方法写，COW是指在并发访问的集合修改集合元素时，不直接修改该集合，而是先复制一份副本，在副本上进行修改。修改完成之后，将指向原来集合的引用指向集合副本。

代码第⑤行是创建UserDAO数据访问对象。代码第⑦行实例化DatagramSocket对象。代码第⑧行是服务器端循环，服务器端一直循环接收客户端数据和发送数据给客户端。

30.6.4 迭代5.4：服务器验证编程

迭代5.4任务实现图30-22中所示的第②步操作。服务器实现代码如下；

```
package com.a51work6.qq.server;
...
while (true) {

    DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
    socket.receive(packet);
    // 接收数据长度
    int len = packet.getLength();
    String str = new String(buffer, 0, len);
    // 从客户端传来的数据包中得到客户端地址
    InetAddress address = packet.getAddress();
    // 从客户端传来的数据包中得到客户端端口号
    int port = packet.getPort();

    JSONObject jsonObj = new JSONObject(str);
    System.out.println(jsonObj);

    int cmd = (int) jsonObj.get("command");                              ①

    if (cmd == COMMAND_LOGIN) {// 用户登录过程                            ②
        // 通过用户Id查询用户信息
        String userId = (String) jsonObj.get("user_id");                 ③
        Map<String, String> user = dao.findById(userId);

        // 判断客户端发送过来的密码与数据库的密码是否一致
        if (user != null && jsonObj.get("user_pwd").equals(user.get("user_pwd"))) {    ④

            JSONObject sendJsonObj = new JSONObject(user);
            // 添加result:0键值对，0表示成功，-1表示失败
            sendJsonObj.put("result", 0);                                ⑤

            ClientInfo cInfo = new ClientInfo();
            cInfo.setUserId(userId);
            cInfo.setAddress(address);
            cInfo.setPort(port);
            clientList.add(cInfo);                                       ⑥

            // 取出好友用户列表
            List<Map<String, String>> friends = dao.findFriends(userId);     ⑦

            // 设置好友状态，更新friends集合，添加online字段
            for (Map<String, String> friend : friends) {                 ⑧
                // 添加好友状态 1在线 0离线
                friend.put("online", "0");                               ⑨
                String fid = friend.get("user_id");
                // 好友在clientList集合中存在，则在线
                for (ClientInfo c : clientList) {
                    String uid = c.getUserId();
                    // 好友在线
                    if (uid.equals(fid)) {
                        // 更新好友状态 1在线 0离线
                        friend.put("online", "1");                       ⑩
                        break;
                    }
                }
            }
            sendJsonObj.put("friends", friends);                         ⑪

            // 创建DatagramPacket对象，用于向客户端发送数据
            byte[] b = sendJsonObj.toString().getBytes();
            packet = new DatagramPacket(b, b.length, address, port);
```

```
            socket.send(packet);                                        ⑫

            // TODO 广播当前用户上线了

        } else {
            // 送失败消息
            JSONObject sendJsonObj = new JSONObject();
            sendJsonObj.put("result", -1);                              ⑬
            byte[] b = sendJsonObj.toString().getBytes();
            packet = new DatagramPacket(b, b.length, address, port);
            // 向请求登录的客户端发送数据
            socket.send(packet);                                        ⑭
        }
    } else if (cmd == COMMAND_SENDMSG) {// 用户发送消息命令
        //TODO用户发送消息
    } else if (cmd == COMMAND_LOGOUT) {// 用户发送注销命令
        //TODO用户注销
    }
}
```

上述代码第①行是从客户端传递过来的命令。代码第②行是判断命令是否为用户登录命令。代码第③行从客户端传递过来的用户Id。代码第④行判断客户端传递过来密码与数据库查询出来的密码是否一致。如果密码一致登录成功，代码第⑤行将result:0键值对放入JSON对象。代码第⑥行将用户所在客户端信息添加到clientList集合中。

代码第⑦行根据用户Id获取好友用户列表。

代码第⑧行是循环获得好友信息，代码第⑨行friend.put("online", "0")是设置好友登录状态为离线。代码第⑩行是设置好友登录状态为离线friend.put("online", "1")。代码第⑪行将好友信息放置到sendJsonObj对象中。

在登录失败时，代码第⑬行是将result:-1键值对放入JSON对象。

代码第⑫行和第⑭行发送数据给客户端。至此服务器验证结束。

30.7　任务6：用户登录刷新好友列表

用户登录成功后，需要给其他客户端发送通知，通知该用户上线，客户端需要刷新好友列表。这个过程如图30-23所示，其中第①步和第②步在任务5已经介绍了，这里不再赘述。

第③步：广播用户1登录成功。

第④步：客户端刷新好友列表。

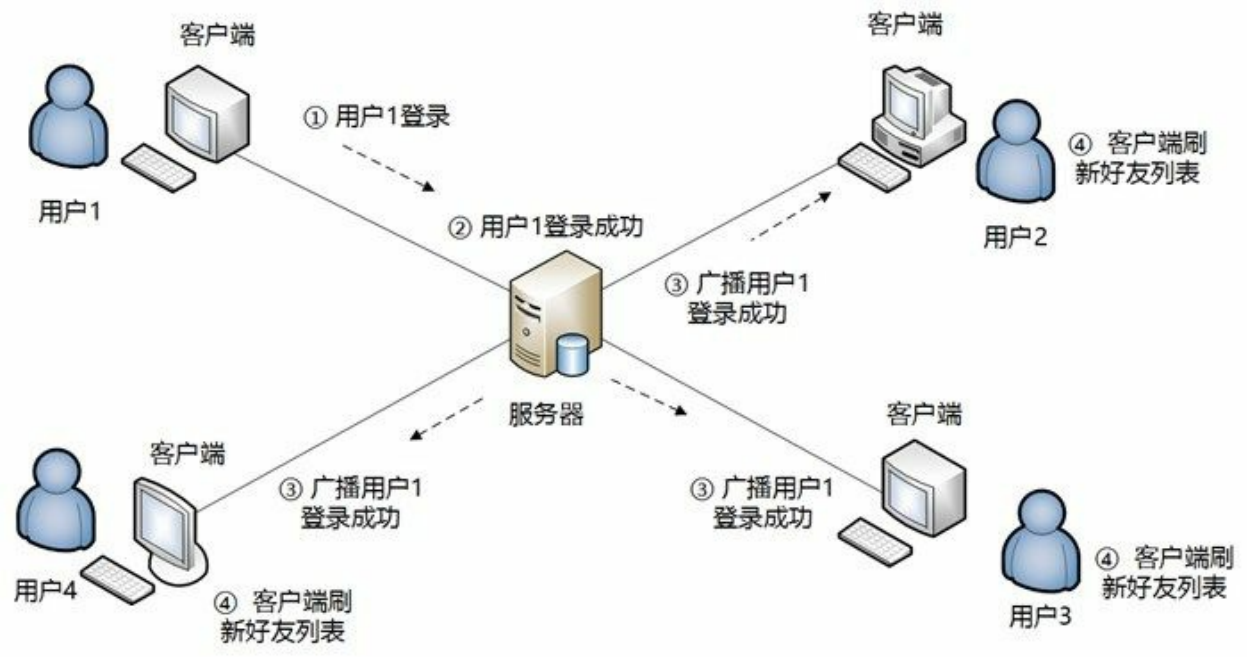


图**30-23**　用户登录刷新好友列表过程

30.7.1　迭代**6.1**：用户登录刷新好友列表服务器端编程

用户登录之后，服务器端接收用户等成功的消息，广播给其他用户的客户端。

Server.java代码如下：

```
if (cmd == COMMAND_LOGIN) {// 用户登录过程
    // 通过用户Id查询用户信息
    String userId = (String) jsonObj.get("user_id");
    Map<String, String> user = dao.findById(userId);

    // 判断客户端发送过来的密码与数据库的密码是否一致
    if (user != null && jsonObj.get("user_pwd").equals(user.get("user_pwd"))) {
        ...
        // 广播当前用户上线了
        for (ClientInfo info : clientList) {                ①
            //  给其他好友发送，当前用户上线消息
            if (!info.getUserId().equals(userId)) {         ②
                jsonObj = new JSONObject();
                jsonObj.put("user_id", userId);
                jsonObj.put("online", "1");                 ③

                byte[] b2 = jsonObj.toString().getBytes();
                packet = new DatagramPacket(b2, b2.length, info.getAddress(), info.getPort());
```

```
                // 发送给其他用户
                socket.send(packet);                          ④
            }
        }

    } else {
        // 送失败消息
        ...
    }
} else if (cmd == COMMAND_SENDMSG) {// 用户发送消息
    //TODO用户发送消息
} else if (cmd == COMMAND_LOGOUT) {// 用户发送注销命令
    //TODO用户注销
}
```

上述代码第①行广播当前用户上线，代码第②行判断如果不是当前用户，则发送消息给其他用户，通过他们当前用户已经上线。代码第③行设当前用户在线状态为1（上线）。代码第④行是发送消息给其他用户。如果当前用户是222，那么其他用户会收如下JSON消息。

```
{
    "user_id": "222",            //当前用户Id
    "online": "1"                //在线
}
```

30.7.2 迭代6.2：用户登录刷新好友列表客户端编程

某个用户登录成功后面，服务器端会广播某用户登录成功，其他用户会更新自己的好友列表。

FriendsFrame.java相关代码如下：

```
package com.a51work6.qq.client;
...
public class FriendsFrame extends JFrame implements Runnable {

    // 线程运行状态
    private boolean isRunning = true;                 ①
    ...
    public FriendsFrame(Map user) {
        setTitle("QQ2006");
        ...
        // 启动接收消息子线程
        resetThread();                                ②
    }

    @Override
    public void run() {                               ③
        // 准备一个缓冲区
        byte[] buffer = new byte[1024];
        while (isRunning) {

            try {
                InetAddress address = InetAddress.getByName(Client.SERVER_IP);
                /* 接收数据报 */
                DatagramPacket packet = new DatagramPacket(buffer,
                        buffer.length, address, Client.SERVER_PORT);
                // 开始接收
                Client.socket.receive(packet);        ④
                // 接收数据长度
                int len = packet.getLength();
                String str = new String(buffer, 0, len);
```

```
                System.out.println("客户端：   " + str);

                JSONObject jsonObj = new JSONObject(str);
                String userId = (String) jsonObj.get("user_id");
                String online = (String) jsonObj.get("online");

                // 刷新好友列表
                refreshFriendList(userId, online);    ⑤

            } catch (Exception e) {
            }
        }
    }

    // 刷新好友列表
    public void refreshFriendList(String userId, String online) {
        // 初始化好友列表
        for (JLabel lblFriend : lblFriendList) {
            // 判断用户Id是否一致
            if (userId.equals(lblFriend.getToolTipText())) {      ⑥
                if (online.equals("1")) {
                    lblFriend.setEnabled(true);
                } else {
                    lblFriend.setEnabled(false);
                }
            }
        }
    }

    // 重新启动接收消息子线程
    public void resetThread() {                              ⑦
        isRunning = true;
        // 接收消息子线程
        Thread receiveMessageThread = new Thread(this);    ⑧
        // 启动接收消息线程
        receiveMessageThread.start();                        ⑨
    }
}
```

上述代码第①行定义线程运行状态，默认为true。代码第②行调用resetThread()方法启动线程。代码第③行是实现线程体，一直接收服务器端返回的消息，代码第④行是接收方法。接收的JSON消息格式如下：

```
{
    "user_id": "222",            //上线好友Id
    "online": "1"                //上线
}
```

如果接收到来自于服务器端的消息，则调用代码第⑤行refreshFriendList(userId, online)方法刷新用户好友列表。

在refreshFriendList方法中，代码第⑥行判断用户Id是否与标签的ToolTipText属性一致，如果一致则设置标签可用，表明用户上线了；否则不可用，说明用户下线了。注意：标签ToolTipText属性保存的用户Id，Text属性保存的用户名。

代码第⑦行定义重新启动接收消息子线程方法resetThread()，该方法用来启动一个接收消息子线程。当用户登录成功时进入好友列表窗口时，或关闭聊天窗口回到好友列表窗口时调用该方法。代码第⑧行是创建接收消息的子线程。代码第⑨行是启动接收消息子线程。

30.8 任务7：聊天过程实现

聊天过程如图30-24所示，客户端用户1向用户3发送消息，这个过程实现有三个步骤：

第①步：客户端用户1向用户3发送消息。

第②步：服务器接收用户1消息与转发给用户3消息。

第③步：客户端用户3接收用户1消息。

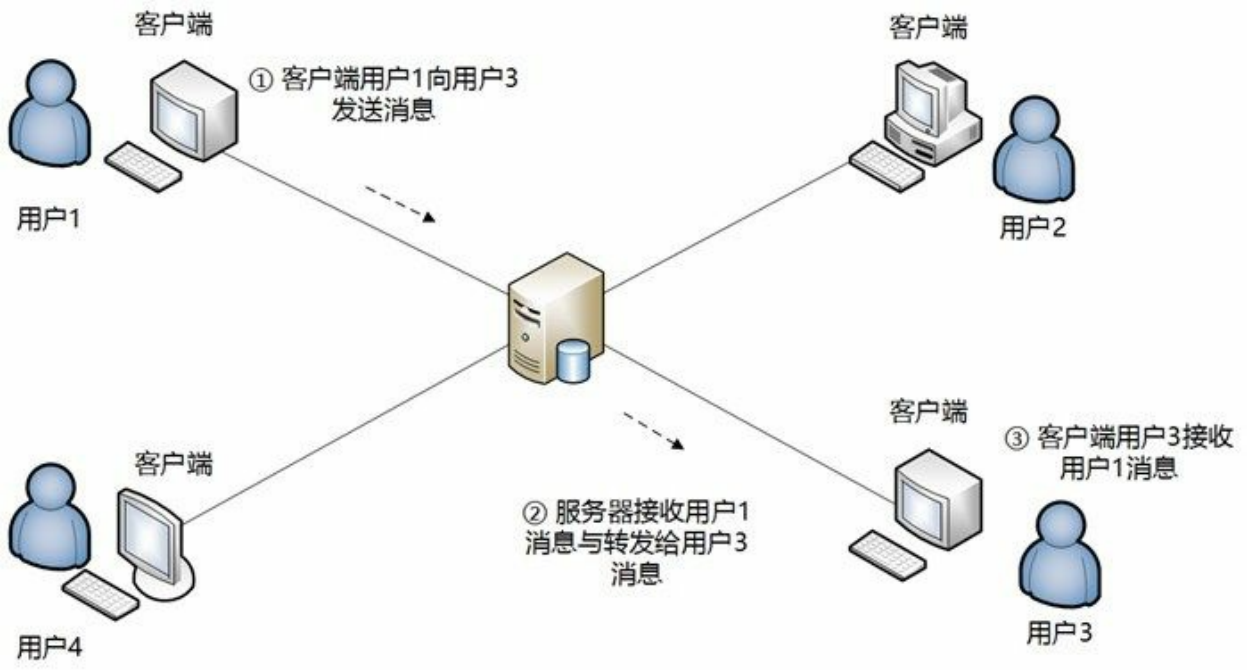


图**30-24** 聊天过程过程

30.8.1 迭代7.1：客户端用户1向用户3发送消息

客户端用户1向用户3发送消息实现是在聊天窗口ChatFrame中实现的。ChatFrame.java相关代码如下：

```
package com.a51work6.qq.client;
...
public class ChatFrame extends JFrame implements Runnable {
    ...
    private JButton getSendButton() {

        JButton button = new JButton("发送");
        button.setBounds(232, 10, 90, 30);
        button.addActionListener(e -> {
            sendMessage();                                  ①
            txtInfo.setText("");
        });
        return button;
    }

    private void sendMessage() {                            ②

        if (!txtInfo.getText().equals("")) {
```

```
            // 获得当前时间，并格式化
            String date = dateFormat.format(new Date());

            String info = String.format("#%s#" + "\n" + "您对%s说：%s",
                                date, friendUserName, txtInfo.getText());
            infoLog.append(info).append('\n');
            txtMainInfo.setText(infoLog.toString());             ③

            Map<String, String> message = new HashMap<String, String>();
            message.put("receive_user_id", friendUserId);
            message.put("user_id", userId);
            message.put("message", txtInfo.getText());

            JSONObject jsonObj = new JSONObject(message);      ④
            jsonObj.put("command", Client.COMMAND_SENDMSG);    ⑤

            try {
                InetAddress address = InetAddress.getByName(Client.SERVER_IP);
                /* 发送数据报 */
                byte[] b = jsonObj.toString().getBytes();
                DatagramPacket packet = new DatagramPacket(b, b.length,
                                address, Client.SERVER_PORT);
                Client.socket.send(packet);                    ⑥

            } catch (IOException e) {
            }
        }
    }
}
```

上述代码第①行是当用户单击发送按钮时调用sendMessage()方法。代码第②行定义sendMessage()方法。代码第③行更新txtMainInfo（显示聊天记录）组件内容。

代码第④行是创建JSON对象，message是客户端发送给服务器的消息。代码第⑤行是添加命令为发聊天消息命令。代码第⑥行是发送数据给服务器端。

```
{
    "receive_user_id": "222",     //接收消息的用户Id（即用户3）
    "message": "你好吗？",        //发送的消息
    "user_id": "111",             //发送消息的用户Id（即用户1）
    "command": 3                  //命令 3是发送聊天消息
}
```

30.8.2 迭代7.2：服务器接收用户1消息与转发给用户3消息

服务器接收用户1消息与转发给用户3消息是在Server中完成，相关代码如下：

```
package com.a51work6.qq.server;
...
public class Server {
    ...
    public static void main(String[] args) {
        ...
        try ( // 创建DatagramSocket对象，监听自己的端口7788
                DatagramSocket socket = new DatagramSocket(SERVER_PORT)) {

            while (true) {
                ...
                if (cmd == COMMAND_LOGIN) {// 用户登录过程
                    ...
```

```
                } else if (cmd == COMMAND_SENDMSG) {// 用户发送聊天消息                    ①
                    // 获得好友Id
                    String friendUserId = (String) jsonObj.get("receive_user_id");       ②

                    // 向客户端发送数据
                    for (ClientInfo info : clientList) {
                        // 找到好友的IP地址和端口号
                        if (info.getUserId().equals(friendUserId)) {                     ③

                            jsonObj.put("OnlineUserList", getUserOnlineStateList());     ④

                            // 创建DatagramPacket对象，用于向客户端发送数据
                            byte[] b = jsonObj.toString().getBytes();
                            packet = new DatagramPacket(b, b.length,
                                        info.getAddress(), info.getPort());
                            // 转发给好友
                            socket.send(packet);
                            break;
                        }
                    }
                } else if (cmd == COMMAND_LOGOUT) {// 用户发送注销命令
                    ...
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    //获得用户在线状态
    private static List<Map<String, String>> getUserOnlineStateList() {                 ⑤
        //从数据库查询所有用户信息
        List<Map<String, String>> userList = dao.findAll();
        //保存用户在线状态集合
        List<Map<String, String>> list = new ArrayList<Map<String, String>>();

        for (Map<String, String> user : userList) {                                      ⑥

            String userId = user.get("user_id");
            Map<String, String> map = new HashMap<String, String>();
            map.put("user_id", userId);
            // 默认离线
            map.put("online", "0");

            for (ClientInfo info : clientList) {                                         ⑦
                //如果clientList（已经登录的客户端信息）中有该用户，则该用户在线
                if (info.getUserId().equals(userId)) {                                   ⑧
                    // 设置为在线
                    map.put("online", "1");
                    break;
                }
            }
            list.add(map);
        }
        return list;
    }
}
```

上述代码第①行是判断客户端命名是否为“用户发送聊天消息”。代码第②行获得接收消息的用户Id。要想给用户3发消息，需要在clientList集合中查找该用户，代码第③行是判断是否找到该用户，如果找该用户那么可以从clientList返回，该用户的客户端主机IP地址和端口号码，有了这些信息就可以给他发消息了。

代码第④行是向发给用户3的消息中添加OnlineUserList消息，OnlineUserList是所有用户的登录状态，这是为了让客户端在聊天的同时也能刷新好友列表。添加OnlineUserList消息后，发给用户3的消息如下：

```
{
    "user_id": "111",                   //发送消息的用户Id（即用户1）
    "receive_user_id": "222",           //接收消息的用户Id（即用户3）
    "message": "你好吗？",
    "OnlineUserList": [                 //所有用户状态
        {
            "user_id": "111",
            "online": "1"
        },
        {
            "user_id": "222",
            "online": "1"
        },
        {
            "user_id": "333",
            "online": "0"
        },
        {
            "user_id": "888",
            "online": "0"
        }
    ],                                  //所有用户状态
    "command": 3
}
```

通过getUserOnlineStateList()方法可以获得所有用户状态，该方法定义见代码第⑤行。在该方法中首先从数据中查询所有的用户信息，然后进行遍历，见代码第⑥行。遍历获得一个用户信息，那么如何判断这个用户是否在线呢？这需要在clientList集合中查找，如果clientList中有该用户，则该用户在线，见代码第⑦行和第⑧行。

30.8.3 迭代7.3：客户端用户3接收用户1消息

客户端用户3接收用户1消息是在聊天窗口类ChatFrame中的，接收消息子线程体中完成。相关代码如下：

```
package com.a51work6.qq.client;
...
public class ChatFrame extends JFrame implements Runnable {

    private boolean isRunning = true;

    // 接收消息子线程
    private Thread receiveMessageThread;
    ...
    @Override
    public void run() {                                                    ①
        // 准备一个缓冲区
        byte[] buffer = new byte[1024];

        while (isRunning) {
            try {

                InetAddress address = InetAddress.getByName(Client.SERVER_IP);
                /* 接收数据报 */
                DatagramPacket packet = new DatagramPacket(buffer,
                        buffer.length, address, Client.SERVER_PORT);
```

```
                // 开始接收
                Client.socket.receive(packet);                    ②
                // 接收数据长度
                int len = packet.getLength();
                String str = new String(buffer, 0, len);

                // 打印接收的数据
                System.out.printf("从服务器接收的数据：【%s】\n", str);
                JSONObject jsonObj = new JSONObject(str);

                // 获得当前时间，并格式化
                String date = dateFormat.format(new Date());
                String message = (String) jsonObj.get("message");

                String info = String.format("#%s#" + "\n" + "%s对您说：%s",
                            date, friendUserName, message);
                infoLog.append(info).append('\n');

                txtMainInfo.setText(infoLog.toString());          ③
                txtMainInfo.setCaretPosition(txtMainInfo.getDocument().getLength());    ④

                Thread.sleep(100);
                // 刷新好友列表
                JSONArray userList = (JSONArray) jsonObj.get("OnlineUserList");         ⑤

                for (Object item : userList) {                                          ⑥
                    JSONObject onlineUser = (JSONObject) item;
                    String userId = (String) onlineUser.get("user_id");
                    String online = (String) onlineUser.get("online");
                    friendsFrame.refreshFriendList(userId, online);                     ⑦
                }

            } catch (Exception e) {
            }
        }
    }
}
```

上述代码第①行是接收消息子线程体，代码第②行是接收消息，代码第③行是将接收的消息显示在文本区中。代码第④行是文本区文本滚动显示到最后一行。

代码第⑤行从接收的消息中获得好友状态。然后通过循环刷新好友列表状态，见代码第6行和第⑦行。

30.9 任务8：用户下线刷新好友列表过程

一个用户单击关闭好友列表窗口，就会下线，在服务器端注销用户登录信息，其他客户端也会收到，该用户下线消息。这个过程如图30-25所示：

第①步：用户1下线。

第②步：广播用户1下线。

第③步：客户端刷新好友列表。

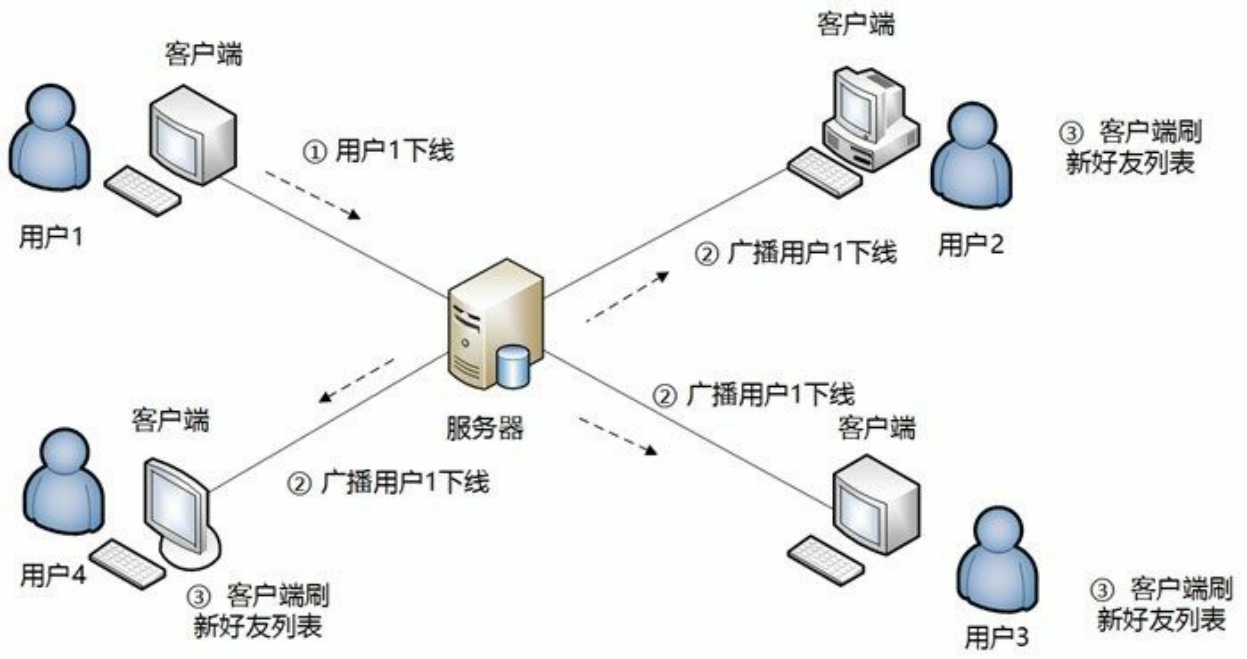


图**30-25** 用户下线刷新好友列表

30.9.1 迭代8.1：客户端编程

用户关闭好友列表窗口触发下线处理，FriendsFrame.java相关代码如下：

```
package com.a51work6.qq.client;
...
public class FriendsFrame extends JFrame implements Runnable {
    ...
    public FriendsFrame(Map user) {
        setTitle("QQ2006");
        ...
        // 注册窗口事件
        addWindowListener(new WindowAdapter() {
            // 单击窗口关闭按钮时调用
            public void windowClosing(WindowEvent e) {                ①

                // 当前用户下线
                JSONObject jsonObj = new JSONObject();                ②
                jsonObj.put("command", Client.COMMAND_LOGOUT);        ③
                jsonObj.put("user_id", userId);
                byte[] b = jsonObj.toString().getBytes();
```

```
                InetAddress address;
                try {
                    address = InetAddress.getByName(Client.SERVER_IP);
                    // 创建DatagramPacket对象
                    DatagramPacket packet = new DatagramPacket(b,
                                b.length, address, Client.SERVER_PORT);
                    // 发送
                    Client.socket.send(packet);                          ④
                } catch (IOException e1) {
                }

                // 退出系统
                System.exit(0);
            }
        });
        ...
    }
    ...
}
```

上述代码第①行是用户关闭窗口时候调用。代码第②行创建JSON对象。代码第③行设置命令。代码第④行发送下线消息。发送的JSON消息格式如下：

```
{
    "user_id": "111",           //发送消息的用户Id（即用户1）
    "command": 2                //命令 2是用户下线
}
```

30.9.2 迭代8.2：服务器端编程

服务器端接收用户下线消息，广播给其他用户的客户端。

Server.java代码如下：

```
package com.a51work6.qq.server;
...
public class Server {
    ...
    public static void main(String[] args) {
        ...

        try ( // 创建DatagramSocket对象，监听自己的端口7788
                DatagramSocket socket = new DatagramSocket(SERVER_PORT)) {

            while (true) {
                ...
                int cmd = (int) jsonObj.get("command");

                if (cmd == COMMAND_LOGIN) {// 用户登录过程
                    ...
                } else if (cmd == COMMAND_SENDMSG) {// 用户发送消息
                    ...

                } else if (cmd == COMMAND_LOGOUT) {// 用户发送注销命令          ①

                    // 获得用户Id
                    String userId = (String) jsonObj.get("user_id");           ②

                    // 从clientList集合中删除用户
                    for (ClientInfo info : clientList) {                       ③
```

```
                        if (info.getUserId().equals(userId)) {
                            clientList.remove(info);                          ④
                            break;
                        }
                    }

                    //向其他客户端广播该用户下线
                    for (ClientInfo info : clientList) {                      ⑤

                        jsonObj = new JSONObject();
                        jsonObj.put("user_id", userId);
                        jsonObj.put("online", "0");                           ⑥

                        byte[] b2 = jsonObj.toString().getBytes();
                        packet = new DatagramPacket(b2, b2.length,
                                    info.getAddress(), info.getPort());
                        socket.send(packet);                                  ⑦
                    }
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    ...
}
```

上述代码第①行判断命令是用户注销命令。代码第②行是获得当前用户Id，代码第③行遍历集合clientList，代码第④行从集合中删除用户信息。

代码第⑤行遍历集合clientList，分别给其它客户端发送当前用户下线。JSON消息格式如下：

```
{
    "user_id": "111",               //发送消息的用户Id（即用户1）
    "online": "0"                   //用户下线
}
```

客户端接收下线信息，并更新好友列表。这个过程与登录成功，刷新好友列表代码共用，具体代码请参考30.7.2节的迭代6.2任务，这里不再赘述。

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们：

- 微博 @图灵教育 : 好书、活动每日播报
- 微博 @图灵社区 : 电子书和好文章的消息
- 微博 @图灵新知 : 图灵教育的科普小组
- 微信 图灵访谈 : ituring_interview，讲述码农精彩人生
- 微信 图灵教育 : turingbooks
