HI，小伙伴你好~

我们在维护者**全网最大的计算机相关编程书籍分享仓库**，目前已有超过 **1000本** 的计算机经典书籍了。

其中涉及**C/C++、Java、Python、Go语言等各种编程语言，还有数据结构与算法、操作系统、后端架构、计算机系统知识、数据库、计算机网络、设计模式、前端、汇编以及校招社招各种面经**等~

只有你想不到，没有我们没分享到的计算机学习书籍，如果真的有我们没能分享到的书籍或者是你所需要的，欢迎添加下方联系方式来告诉我们，期待你的到来。

在此承诺本仓库**永不收费**，永远免费分享给有需要的人，希望自己的**幸苦结晶**能够帮助到曾经那些像我一样的小白、新手、在校生们，为那些曾经像我一样迷茫的人指明一条路。

告诉他们，你是可以的！

本仓库无偿分享各种计算机书籍、各种专业**PDF**资料以及个人笔记资料等，所有权归仓库作者**阿秀**（公众号【**拓跋阿秀**】）所有，如有疑问提请**issue**或者联系本人forthespada@foxmail.com，感谢~

衷心希望我以前踩过的坑，你们不要再踩，我走过的路，你们可以照着走下来。

因为从双非二本爬到字节跳动这种大厂来，太**TMD**难了。

QQ群：①群:1002342950、②群:826980895

从小白 Java 到大牛

免费精简版

关东升 著

免费
For Free

版权声明

联系方式电子邮箱：eorient@sina.com

作者简介

作者关东升，一个在 IT 领域摸爬滚打 20 多年的老程序员、培训师、作者。精通多种 IT 技术，如 Java、iOS、Android、游戏开发、数据库开发与设计、软件架构设计等。

参与设计和开发北京市公交一卡通等大型项目，开发国家农产品追溯系统、金融系统微博等移动客户端项目，并在 App Store 发布多款游戏和应用软件。近期为中国移动、中国联通、南方航空、中石油、工商银行、平安银行和天津港务局等企事业单位授课。

著有 20 多本计算机书籍，其中畅销书有：

- 《Java 从小白到大牛》
- 《iOS 开发指南——从零基础到 App Store 上架》1-3 版
- 《iOS 开发指南——从 HelloWorld 到 App Store 上架》4-5 版
- 《从零开始学 Swift》1-2 版
- 《Swift 开发指南》1-修订版
- 《iOS 实战：入门与提高卷》Swift 版
- 《iOS 实战：苹果“生态圈”编程卷》Swift 版
- 《iOS 实战：传感器卷》Swift 版
- 《iOS 实战：图形图像、动画和多媒体卷》Swift 版
- 《iOS 网络编程与云端应用最佳实践》Objective-C 版
- 《iOS 传感器应用开发最佳实践》Objective-C 版
- 《iOS 图形图像、动画和多媒体编程技术最佳实践》Objective-C 版
- 《iPhone 与 iPad 开发实战——iOS 经典应用剖析》
- 《品味移动设计》
- 《交互设计的艺术》
- 《Android 开发案例驱动教程》
- 《Android 网络游戏开发实战》
- 《Cocos2d-x 实战：C++卷》1-2 版
- 《Cocos2d-x 实战：JS 卷——Cocos2d-JS 开发》1-2 版
- 《Cocos2d-x 实战：Lua 卷》1-2 版

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。

本书收费版本：

另外，与本书免费版对应的还有一个收费版本。

百度阅读地址：

https://yuedu.baidu.com/ebook/7c1499987e192279168884868762caaedd33ba00

图灵社区地址：

http://www.ituring.com.cn/book/2480

免费版和收费版在内容方面的差别如下：

内容	免费版本	收费版本
Java 编码规范	无	有
枚举类	无	有
Java 常用类	无	有
内部类	无	有
Lambda 表达式	无	有
泛型	无	有
多线程	无	有
Swing 图形用户界面编程	无	有
反射	无	有
注解（Annotation）	无	有
数据库编程	无	有

项目实战 1：开发 PetStore 宠物商店项目	无	有
项目实战 2：开发 Java 版 QQ2006 聊天工具	无	有

免费版和收费版服务方面的差别如下：

服务	免费版本	收费版本
智捷课堂提供的图书配套视频	无	有
图书配套课件 PPT	无	有
智捷课堂提供的答疑服务	无	有
图书配套源代码	无	有

本书服务网址：

为了更好地为广大读者提供服务，我们专门为本书建立了一个服务网址 www.51work6.com/book/java1.php，希望读者对书中内容发表评论，提出宝贵意见。

源代码：

书中包括了 100 多个完整的案例项目源代码，大家可以到本书网站 www.51work6.com/book/java1.php 下载免费下载。

我们的联系方式：

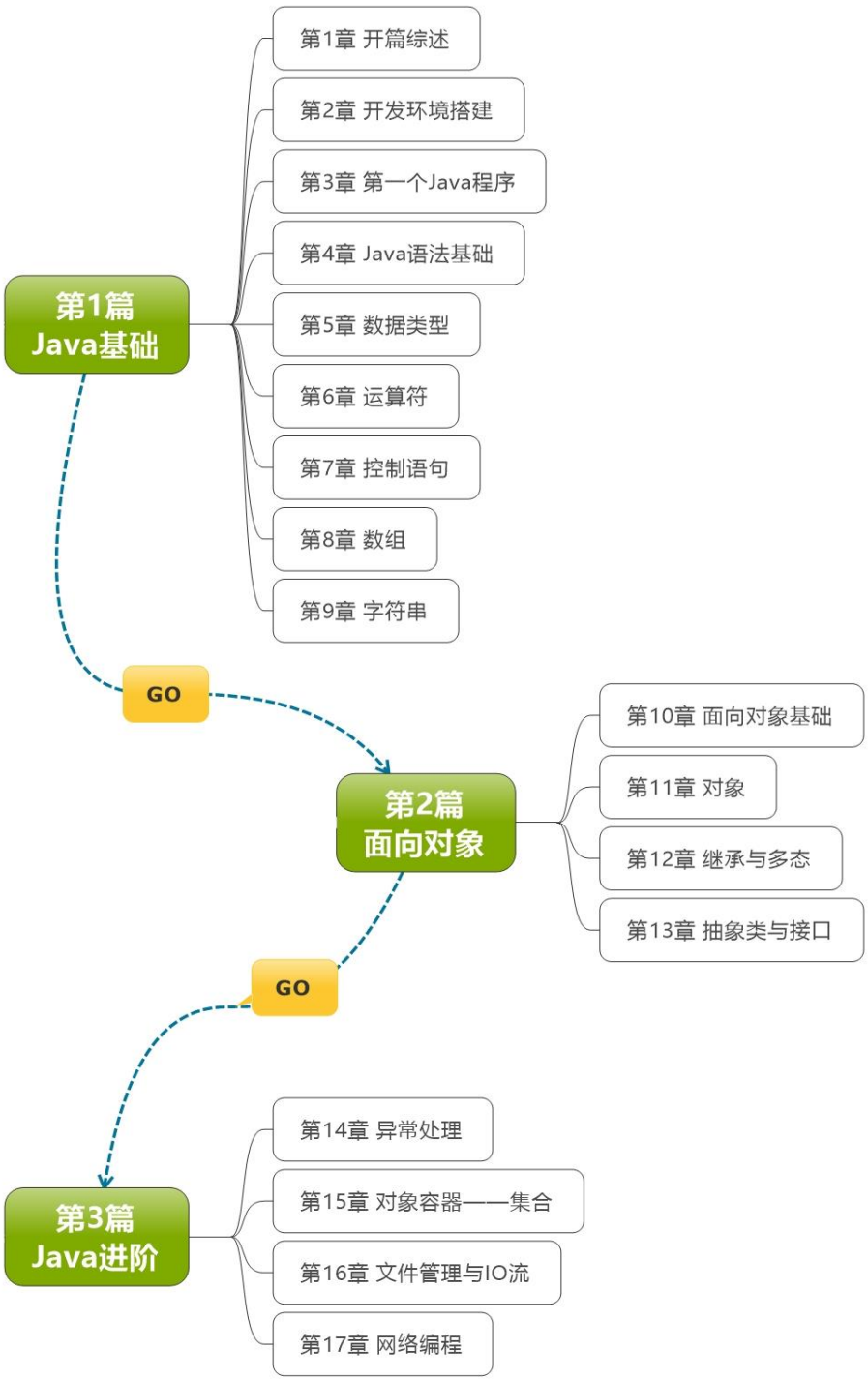
作者微博：@tony_关东升
邮箱：eorient@sina.com
QQ 群：547370999
智捷课堂在线课堂：www.zhijieketang.com
智捷课堂微信公共号：zhijieketang

关东升　2017 年 8 月 15 日 于北京

内容简介

本书是一本 Java 语言学习教程，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、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、集合框架、异常处理、输入输出和网络编程等技术。

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 数据类型
第6章 运算符
第7章 控制语句
第8章 数组
第9章 字符串
GO
第2篇
面向对象
第10章 面向对象基础
第11章 对象
第12章 继承与多态
第13章 抽象类与接口
GO
第3篇
Java进阶
第14章 异常处理
第15章 对象容器——集合
第16章 文件管理与IO流
第17章 网络编程

目　　录

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 TIOBE 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	↓	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	↓	PHP	2.693%	-0.300%
10	9	↓	Perl	2.602%	0.280%
11	8	↓	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	↓	Objective-C	2.101%	0.500%
16	42	⇈	Go	2.080%	1.830%
17	18	↑	MATLAB	2.063%	0.780%
18	11	⇊	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

[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。

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。

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：

1. 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。

2. 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。

3. 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。

4. 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。

5. 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。

6. 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。

7. 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。

8. 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

[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。

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。

9. 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。

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。

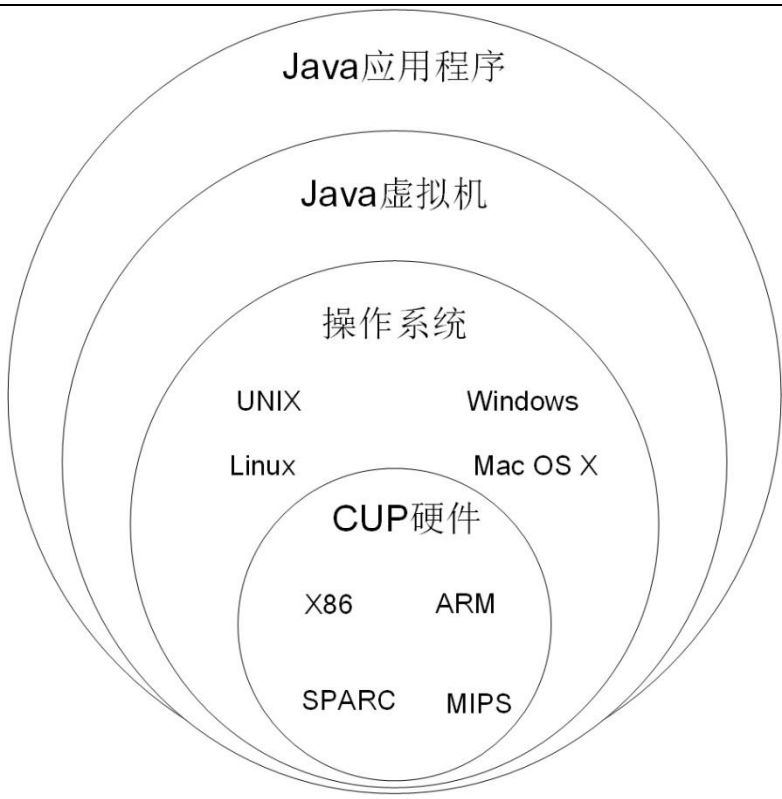


图 1-1　Java 虚拟机

Java 虚拟机是中包含了 Java 解释器，Java 程序在运行过程如图 1-2 所示，首先由编译器将加 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。

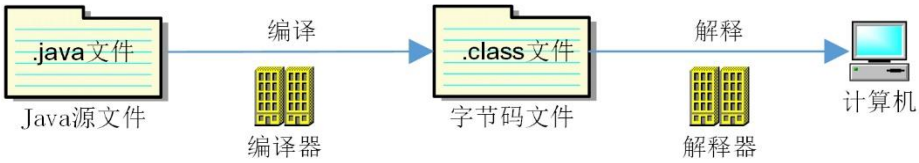


图 1-2　Java 程序运行过程

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件。

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[3]、Solaris[4] 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。

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。

[3] 苹果桌面操作系统，基于 UNIX 操作系统，现在改名为 macOS。
[4] 原 Sun 公司 UNIX 操作系统，现在被 Oracle 公司收购。

Overview | Downloads | Documentation | Community | Technologies | Training

Java SE Development Kit 8 Downloads

Thank you for downloading this release of the Java™ Platform, Standard Edition Development Kit (JDK™). The JDK is a development environment for building applications, applets, and components using the Java programming language.

The JDK includes tools useful for developing and testing programs written in the Java programming language and running on the Java platform.

See also:

- Java Developer Newsletter: From your Oracle account, select **Subscriptions**, expand **Technology**, and subscribe to **Java**.
- Java Developer Day hands-on workshops (free) and other events
- Java Magazine

JDK 8u131 checksum

Java SE Development Kit 8u131

You must accept the Oracle Binary Code License Agreement for Java SE to download this software.

○ Accept License Agreement ◉ Decline License Agreement

Product / File Description	File Size	Download
Linux ARM 32 Hard Float ABI	77.87 MB	jdk-8u131-linux-arm32-vfp-hflt.tar.gz
Linux ARM 64 Hard Float ABI	74.81 MB	jdk-8u131-linux-arm64-vfp-hflt.tar.gz
Linux x86	164.66 MB	jdk-8u131-linux-i586.rpm
Linux x86	179.39 MB	jdk-8u131-linux-i586.tar.gz
Linux x64	162.11 MB	jdk-8u131-linux-x64.rpm
Linux x64	176.95 MB	jdk-8u131-linux-x64.tar.gz
Mac OS X	226.57 MB	jdk-8u131-macosx-x64.dmg
Solaris SPARC 64-bit	139.79 MB	jdk-8u131-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	99.13 MB	jdk-8u131-solaris-sparcv9.tar.gz
Solaris x64	140.51 MB	jdk-8u131-solaris-x64.tar.Z
Solaris x64	96.96 MB	jdk-8u131-solaris-x64.tar.gz
Windows x86	191.22 MB	jdk-8u131-windows-i586.exe
Windows x64	198.03 MB	jdk-8u131-windows-x64.exe

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

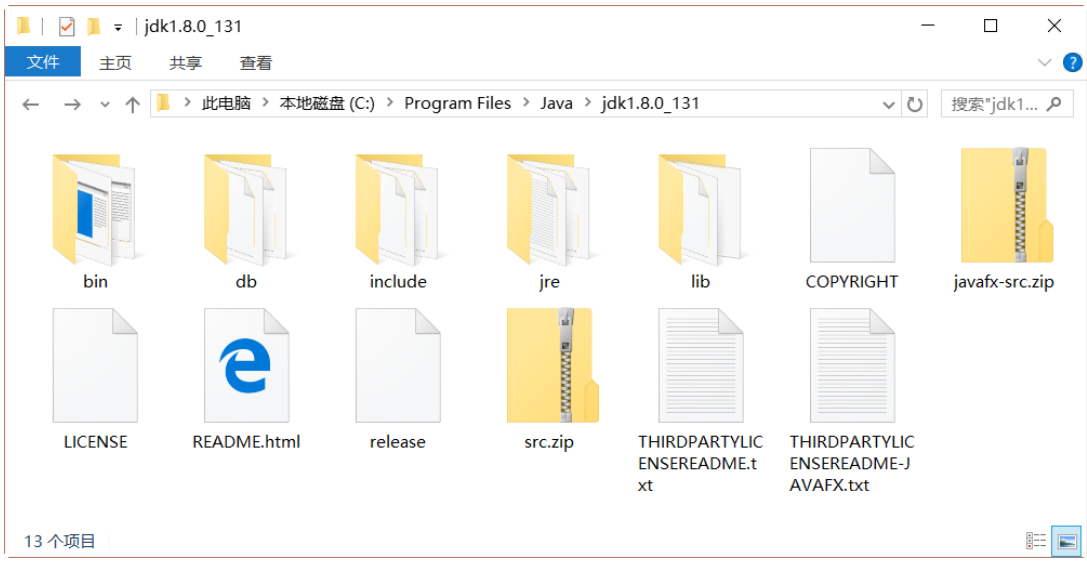


图 2-3 JDK 安装后的内容

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：

1. 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。
2. 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，单击右边的“高级系统设置”超连接，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

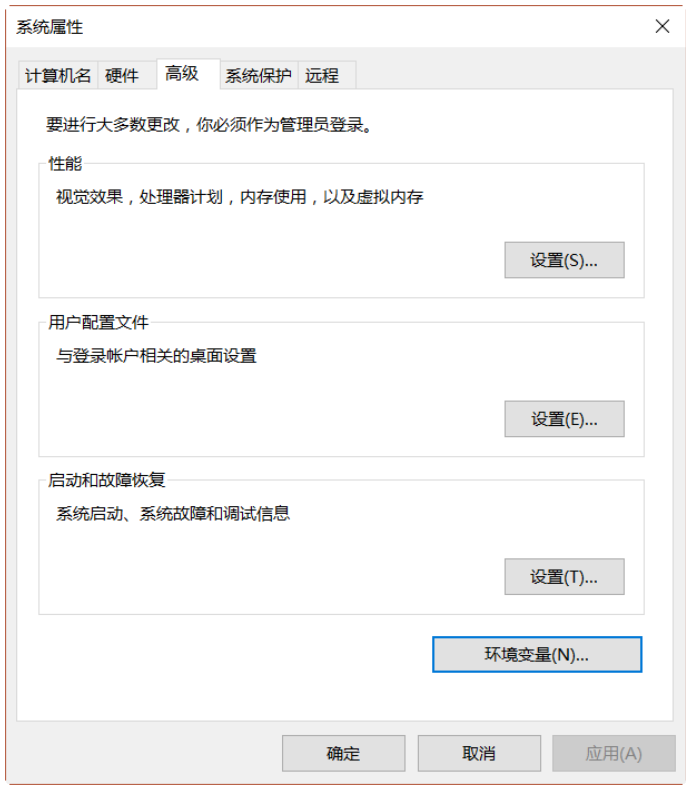


图 2-5 高级系统设置对话框

在如图 2-5 所示的高级系统设置对话框中，点击“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。

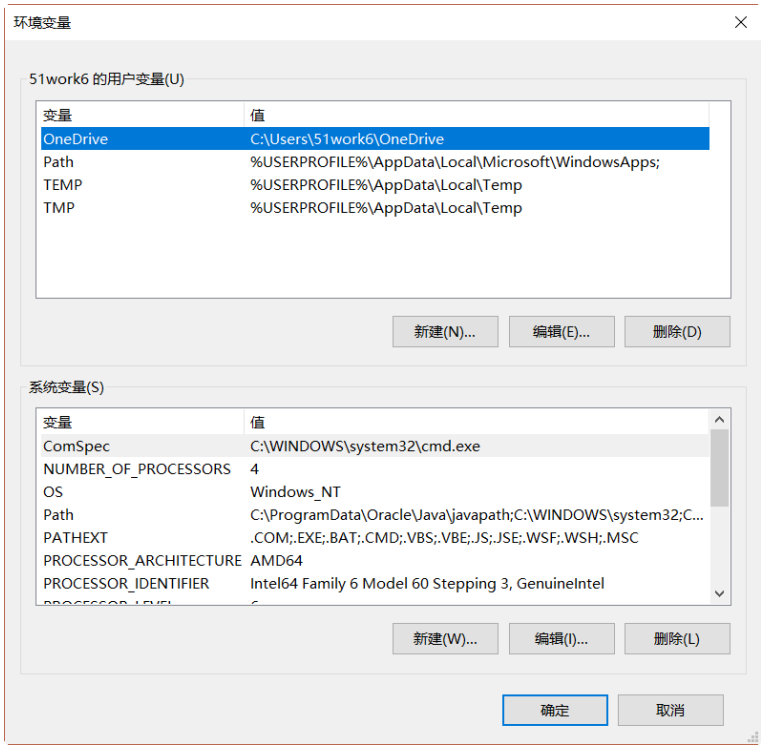


图 2-6 环境变量设置对话框

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。

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）分割。最后单击“确定”按钮完成设置。

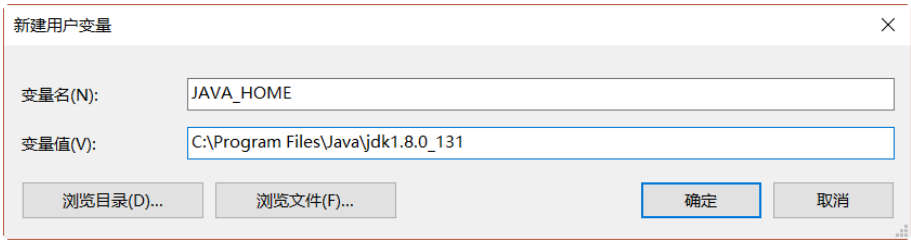


图 2-7 设置 JAVA_HOME

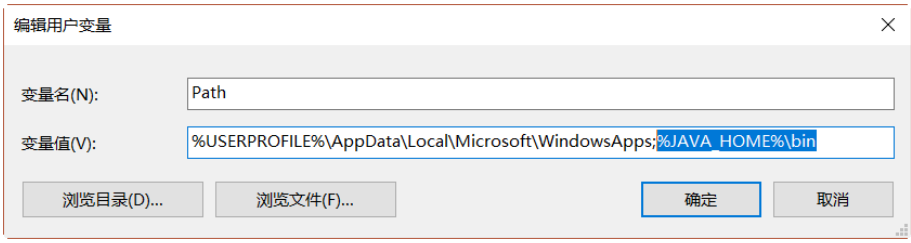


图 2-8 添加 Path 变量对话框

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[5] 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

[5] 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）。

开始下载。

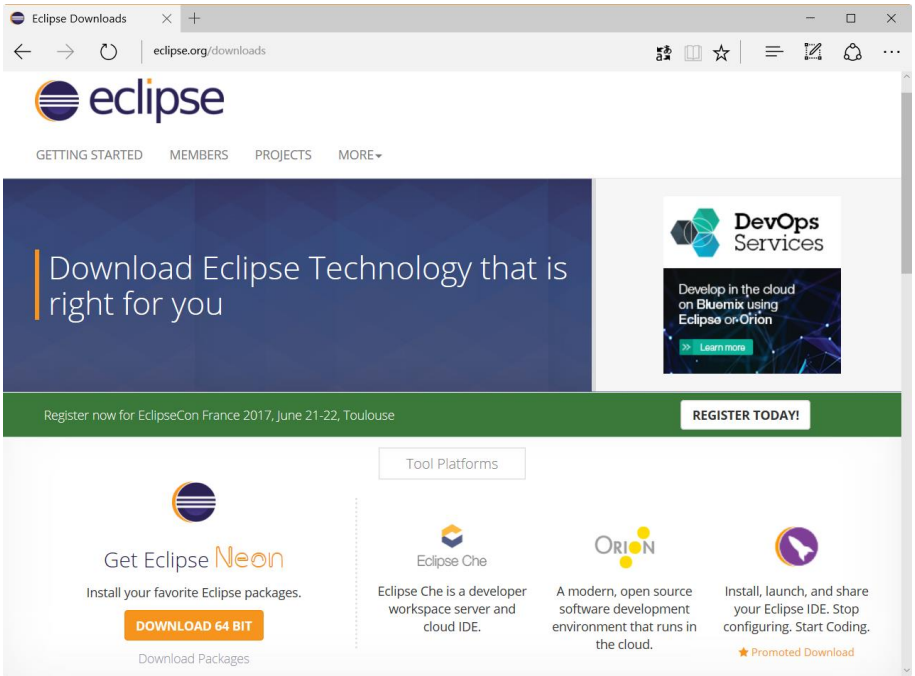


图 2-10 Eclipse 4.6 下载页面

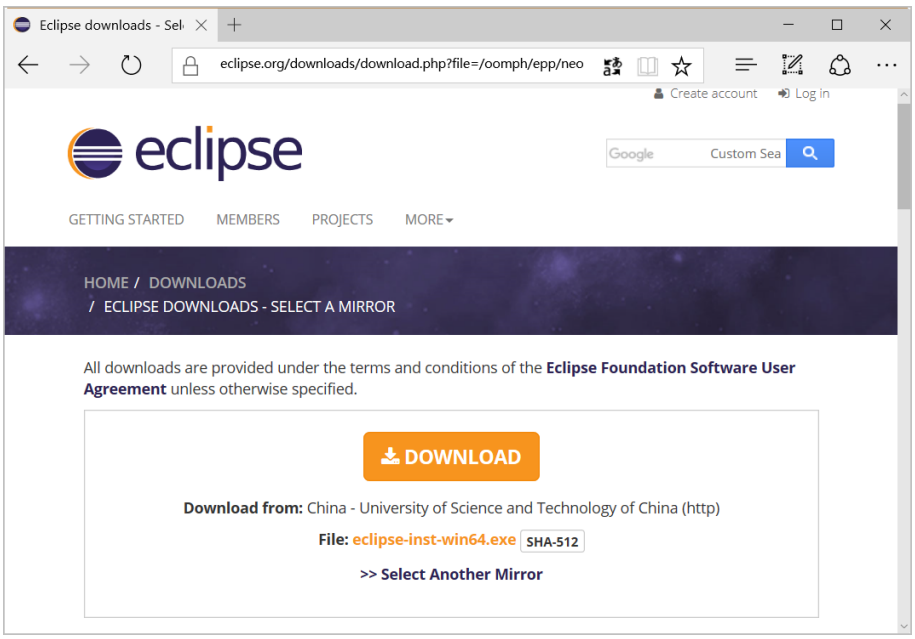


图 2-11 选择下载镜像地址

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，安装完成后单击 LAUNCH 按钮启动 Eclipse。

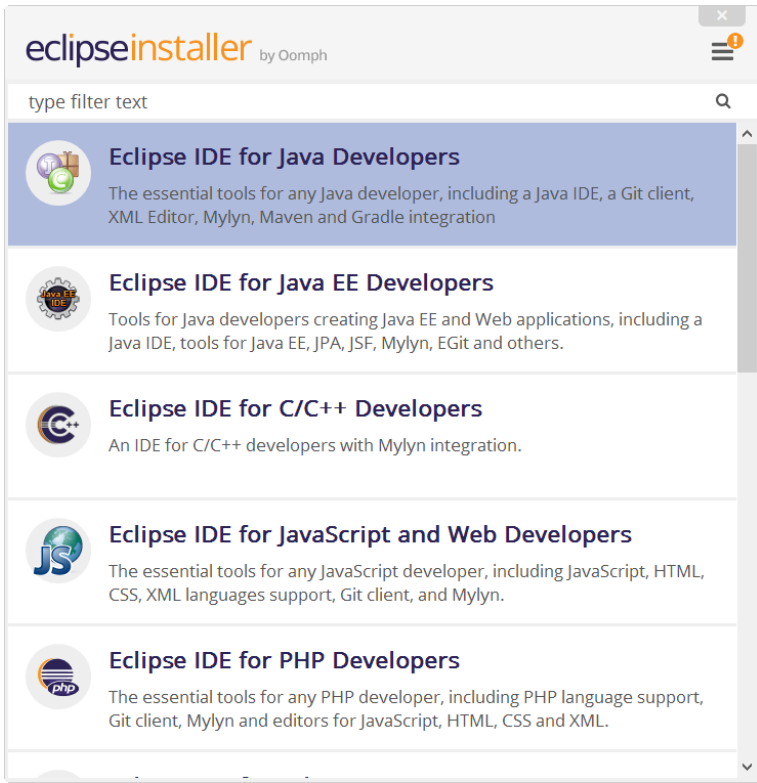


图 2-12 安装各种 Eclipse 版本客户端

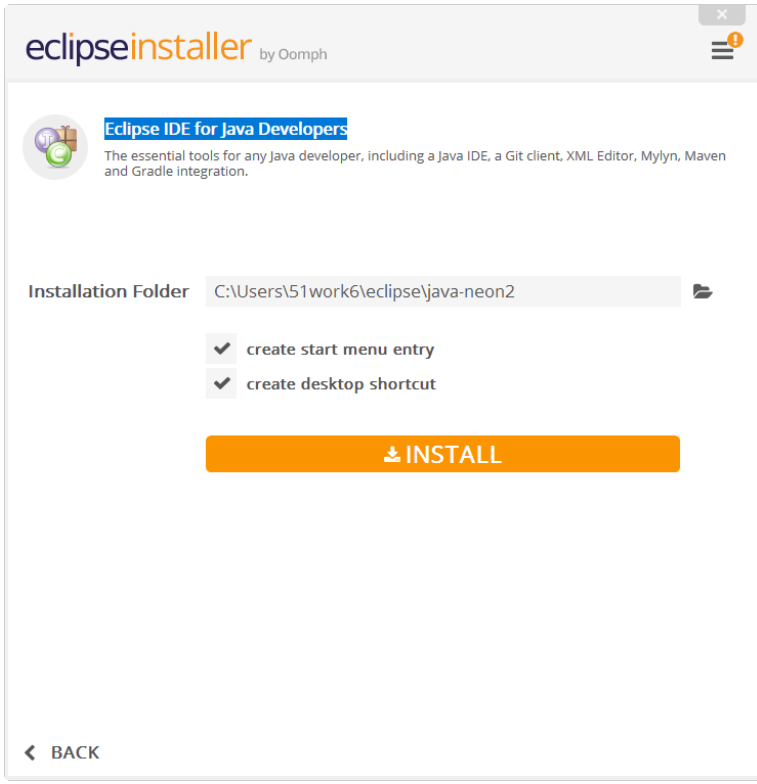


图 2-13 Eclipse 安装

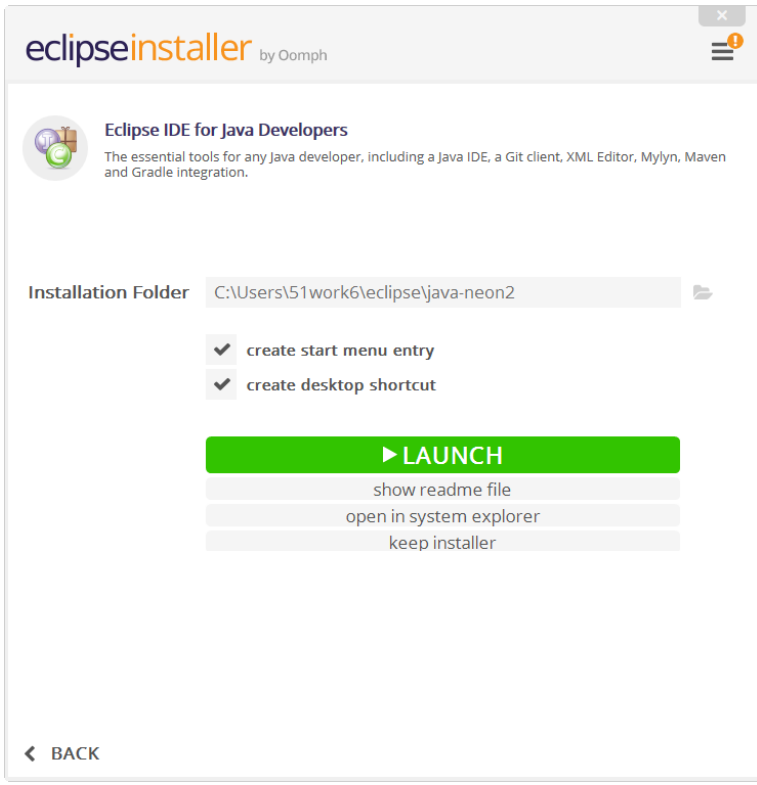


图 2-14 Eclipse 安装完成

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。

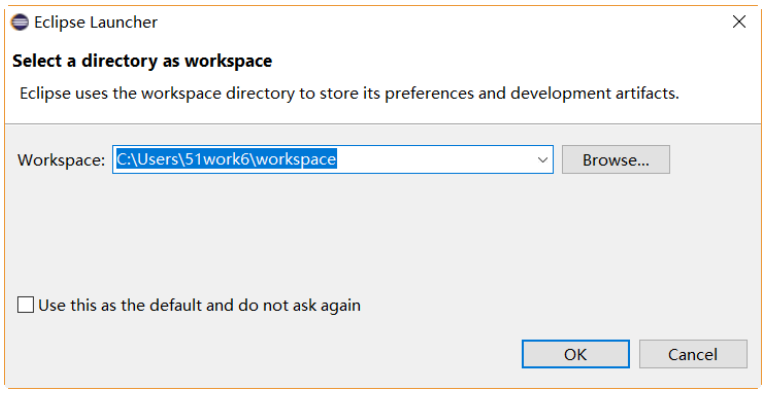


图 2-15 选择工作空间

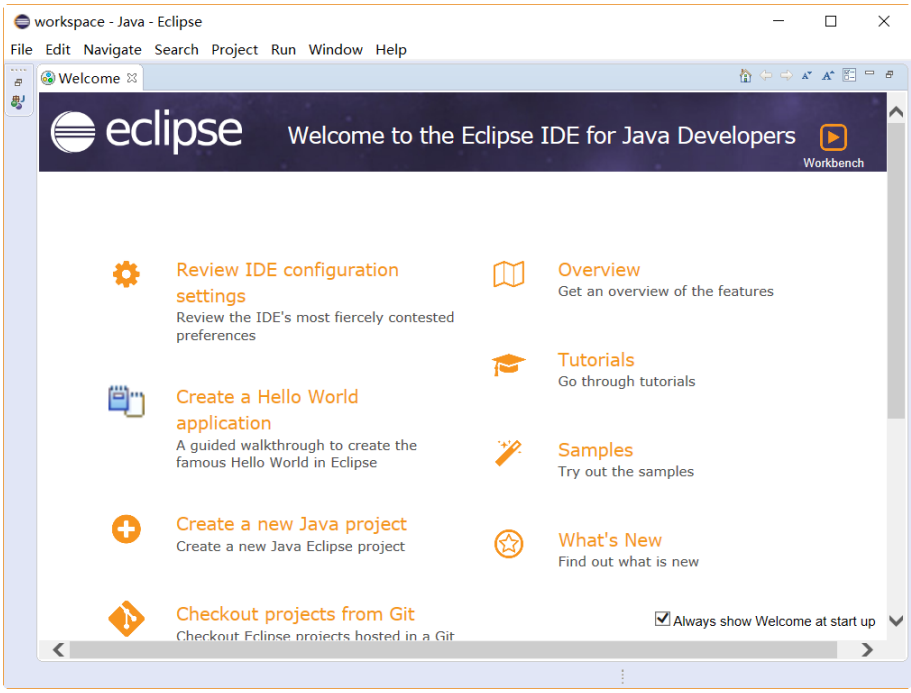


图 2-16 Eclipse 欢迎界面

2.2.2 安装中文语言包

Eclipse 界面默认是英文，对于一些初学者英语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 超连接下载离线包，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。

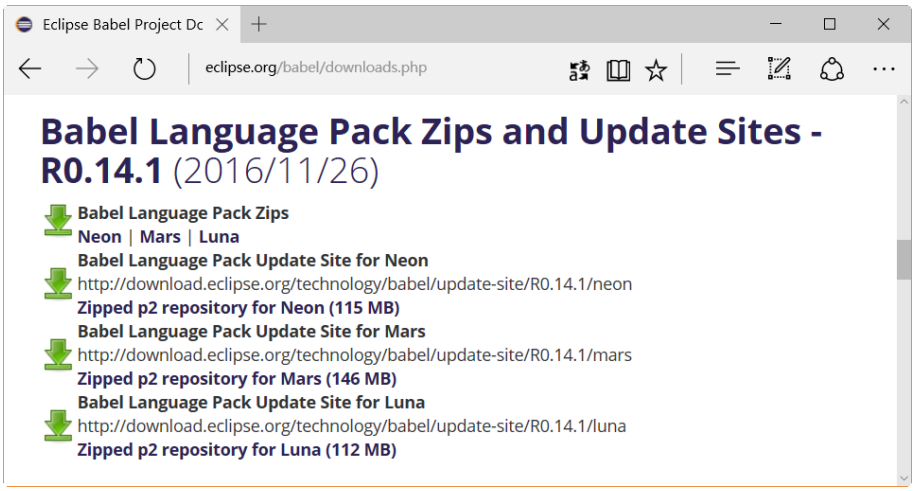


图 2-17 下载 Eclipse 语言包

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。

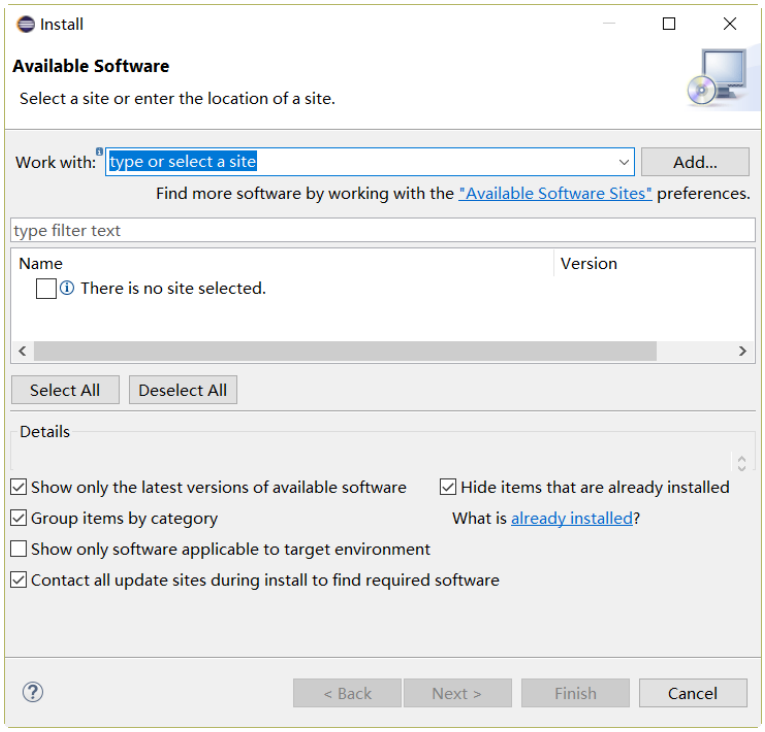


图 2-18 安装插件

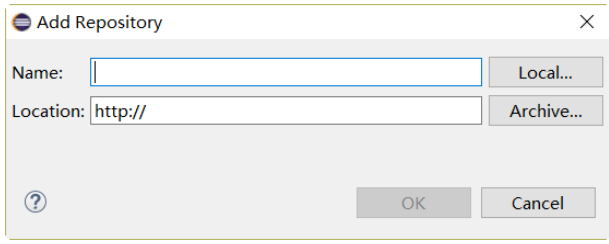


图 2-19 插件地址

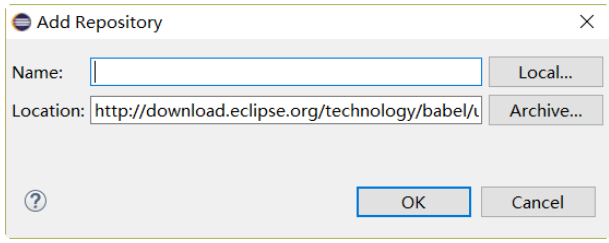


图 2-20 输入插件地址

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。

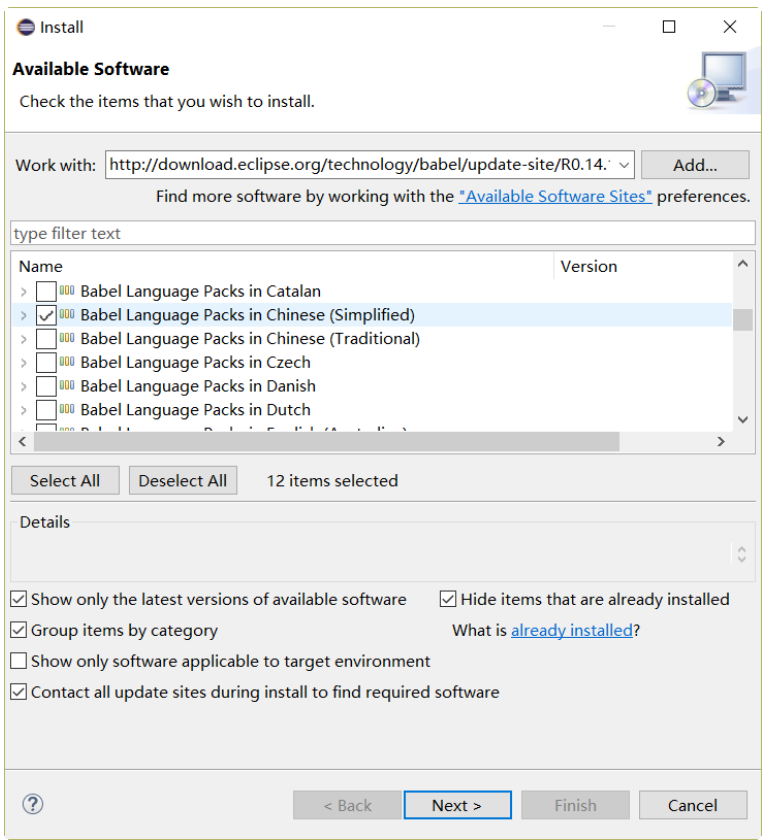


图 2-21 选择简体中文语言包

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。

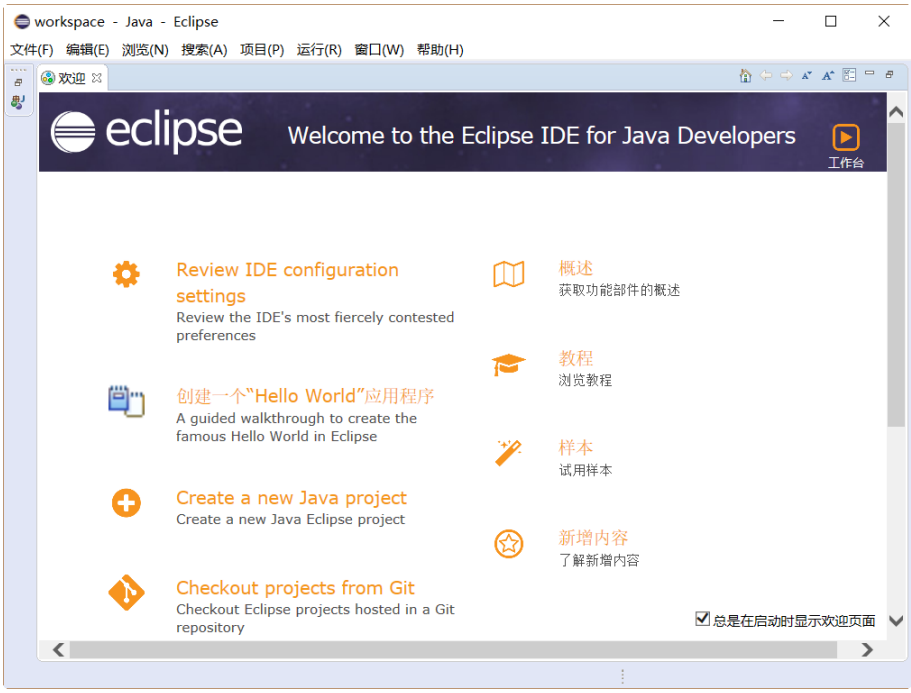


图 2-22 安装简体中文语言包后 Eclipse

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。

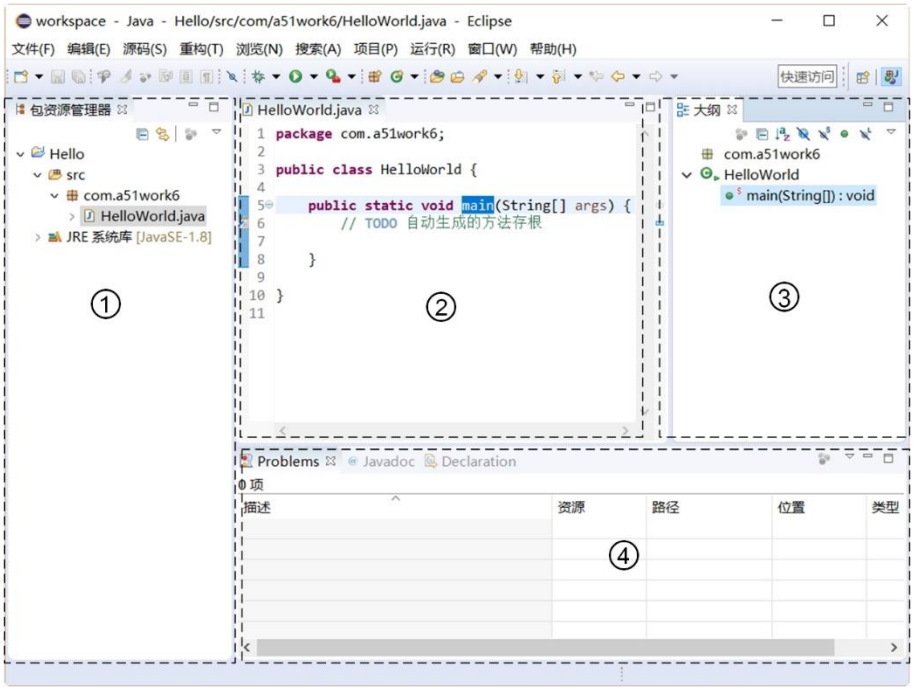


图 2-23 Eclipse 主界面

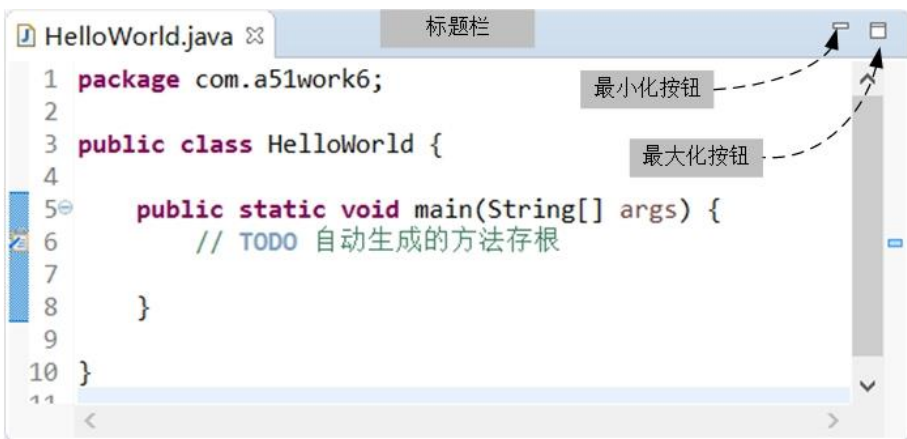


图 2-24 Eclipse 视图

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，但有可以锦上添花。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 卡发工具。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。

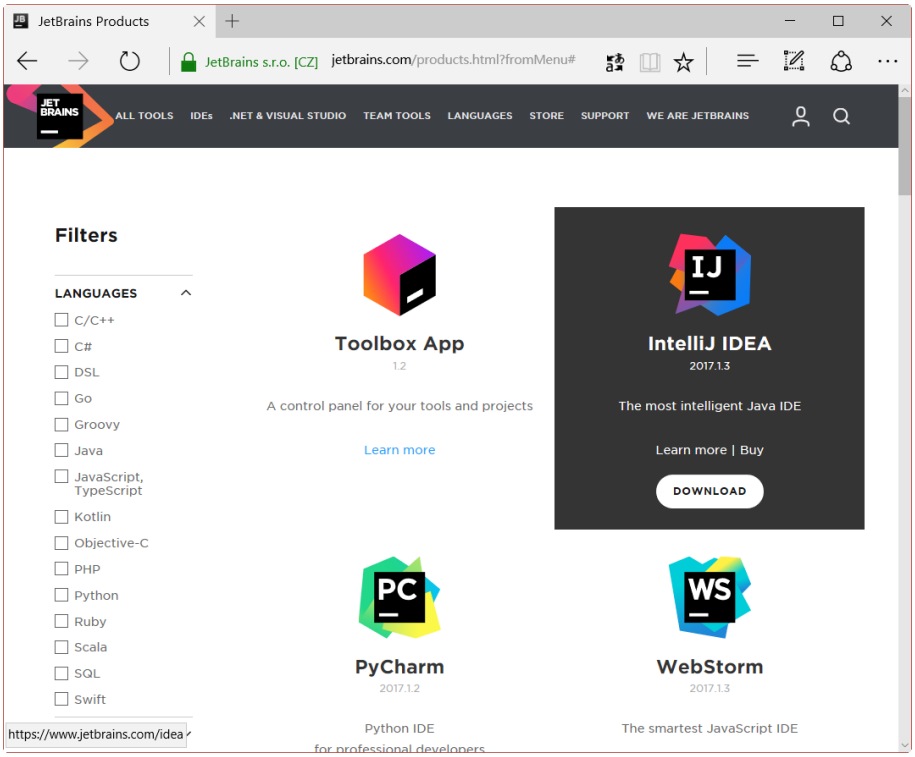


图 2-25 Jetbrains 公司工具

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，完成之后需要安装了。

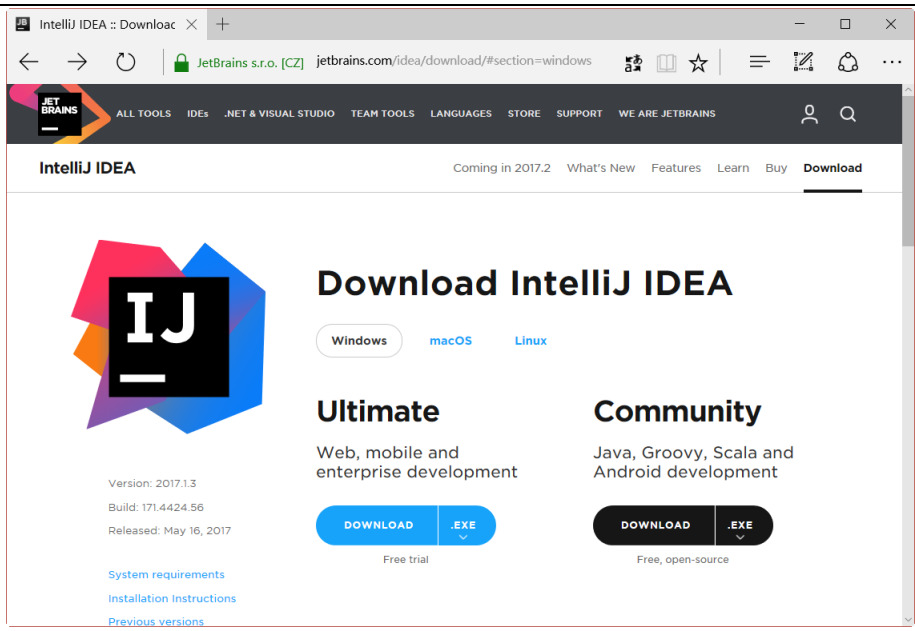


图 2-26 下载 IntelliJ IDEA

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。

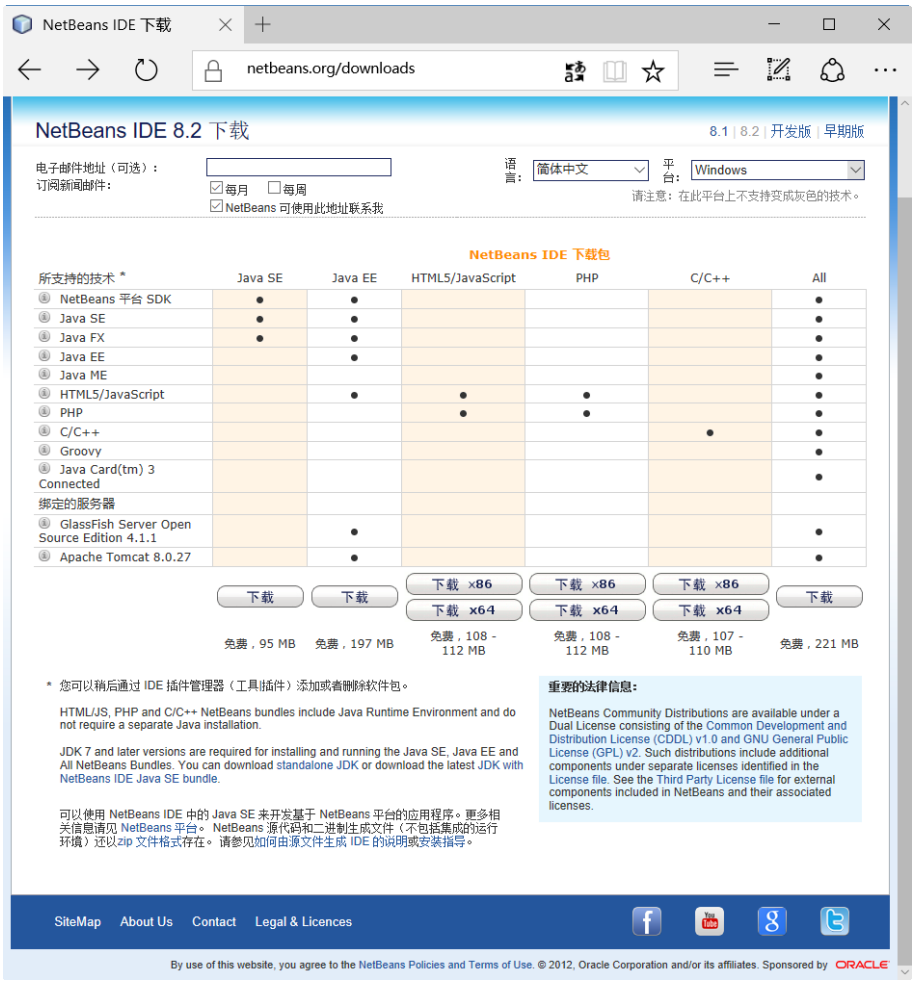


图 2-27 NetBeans IDE 下载页面

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[6]的文件中进行的，初学者入门比较难，官网 www.sublimetext.com。

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。

1. 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。

2. 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。

[6] 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"}。

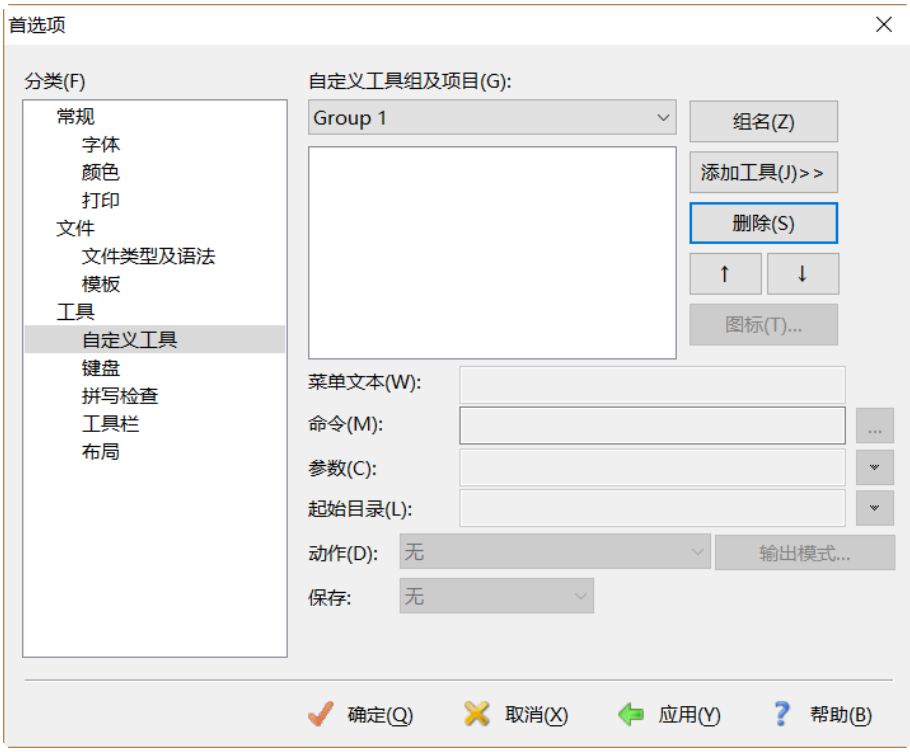


图 2-28 EditPlus 设置参数

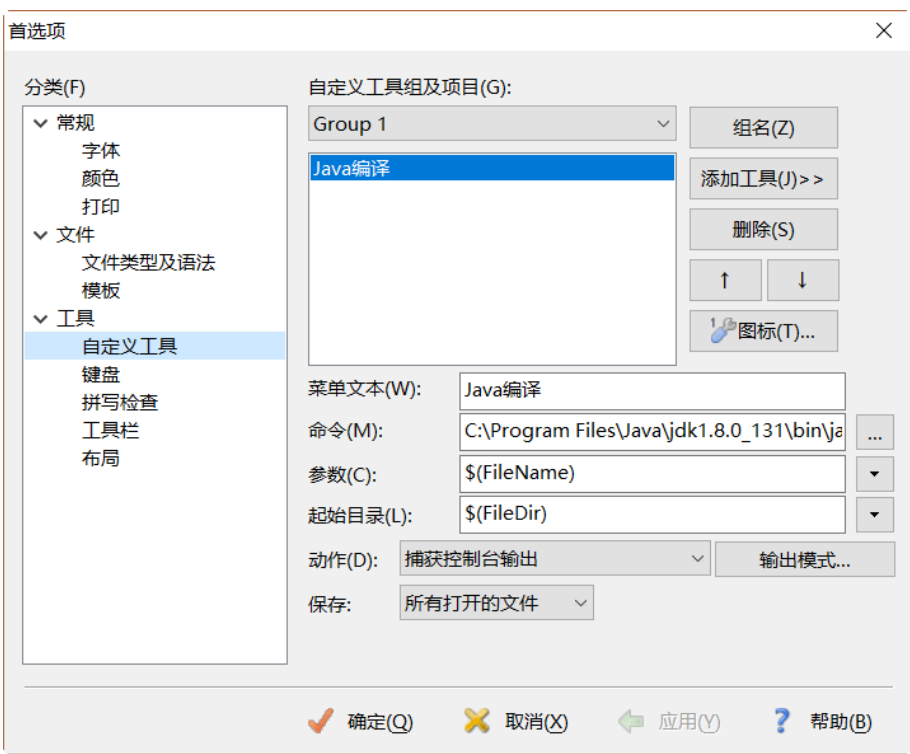


图 2-29 添加编译菜单

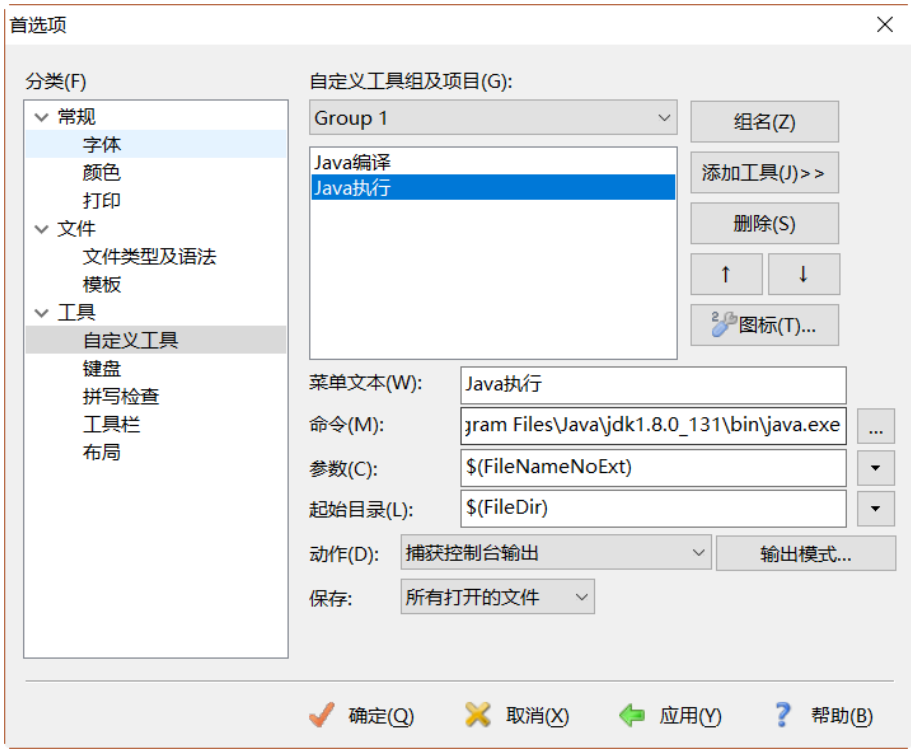


图 2-30 添加执行菜单

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

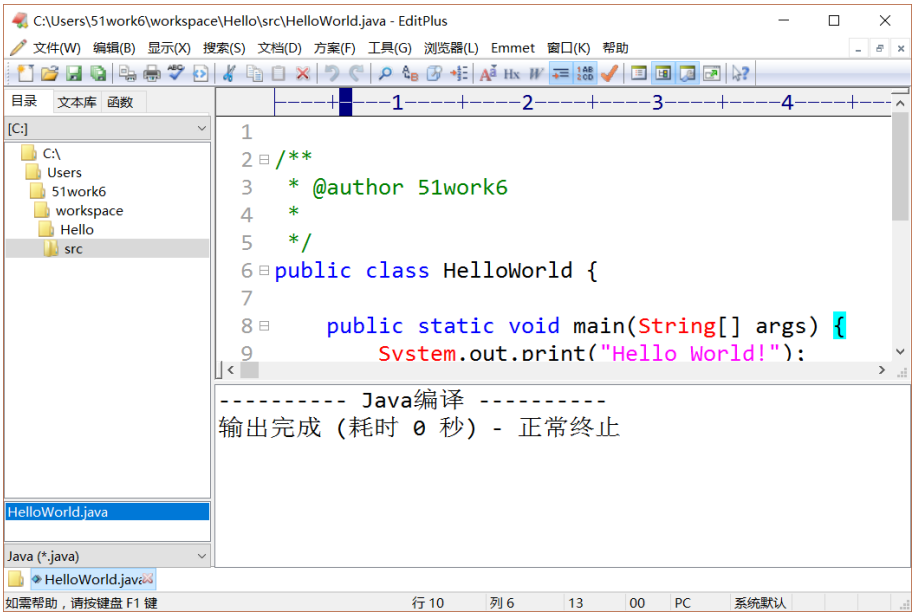


图 2-32 执行 Java 编译菜单

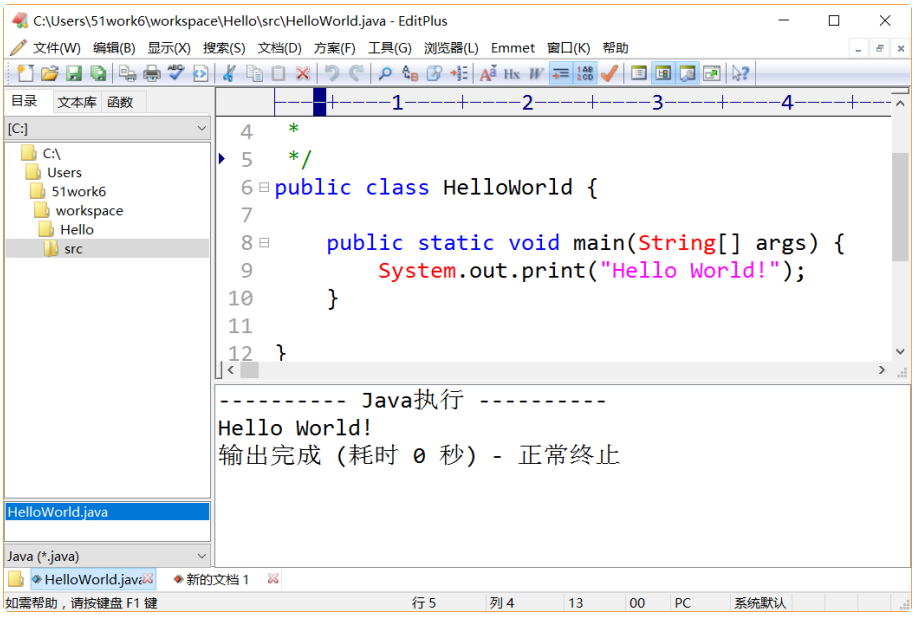


图 2-33 执行 Java 运行菜单

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放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。

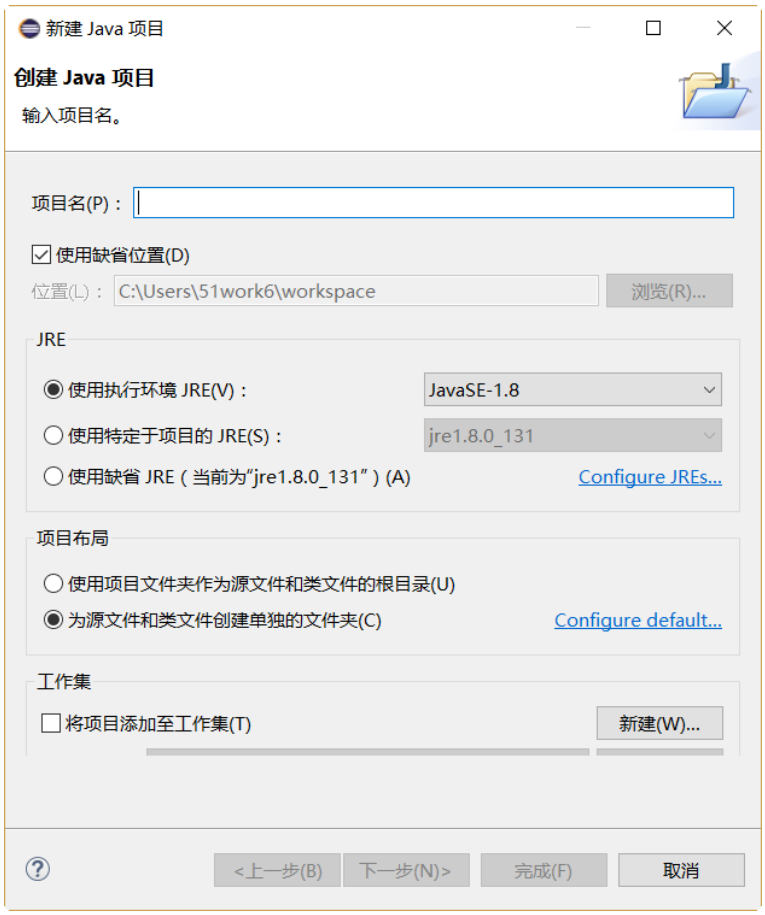


图 3-1　新建 Java 项目对话框

图 3-2　Java 设置对话框

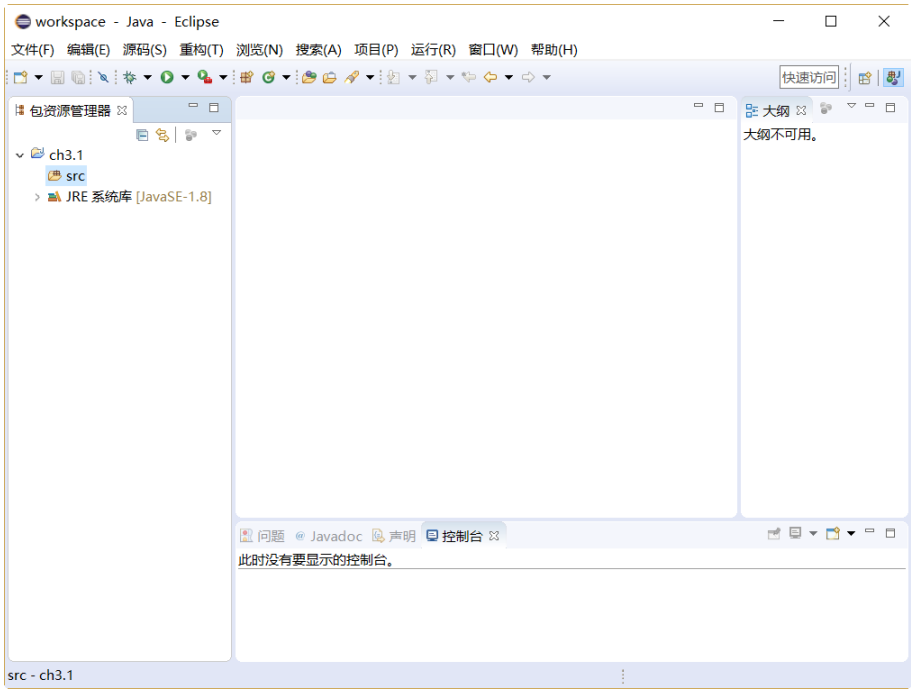


图 3-3　项目创建完成

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，也可以修改注释模版。

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。

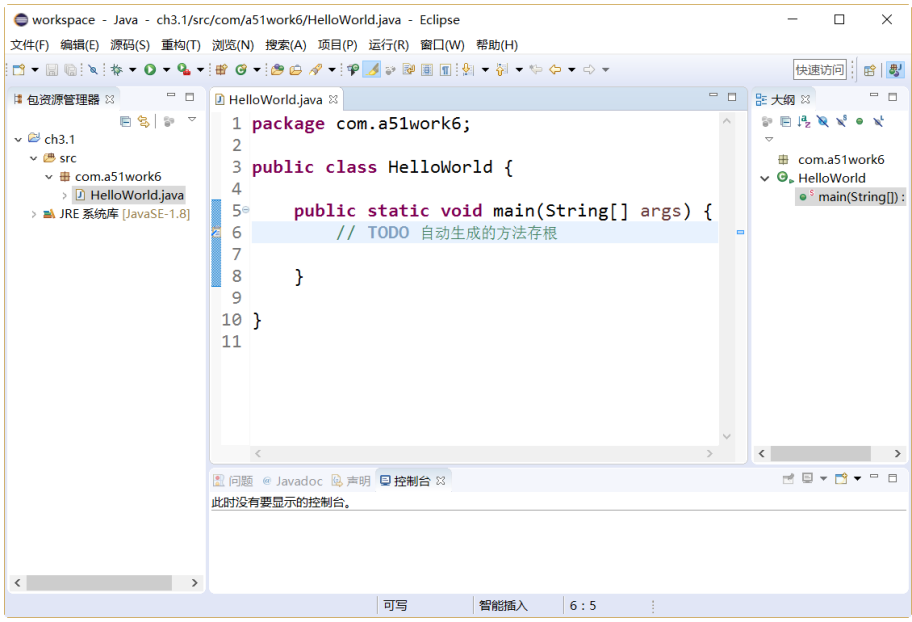


图 3-5　创建类完成

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。

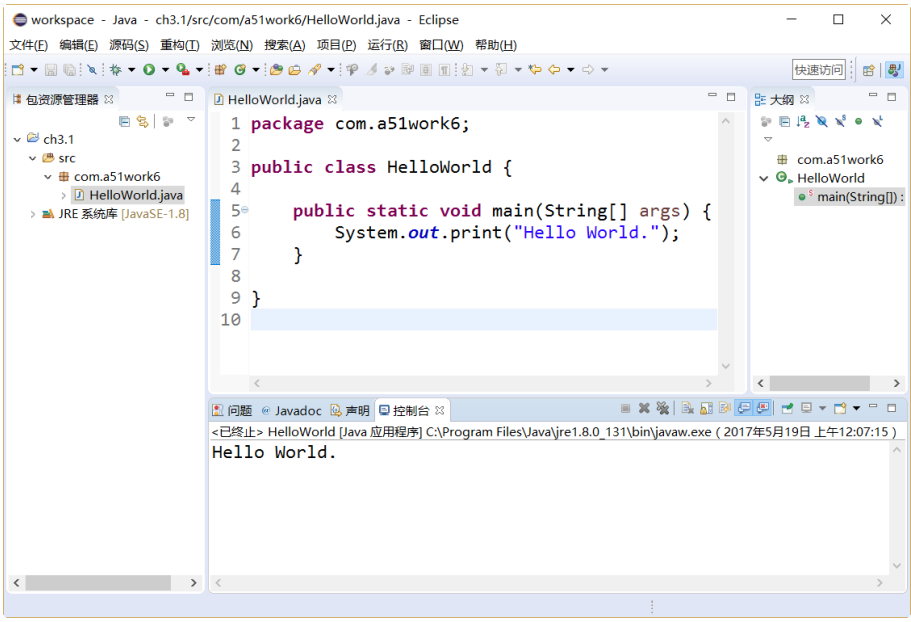


图 3-6 运行结果

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：

1. 只能有一个类声明为公有（public）的。

2. 文件命名必须与公有类名完全一致，包括字母大小写。

3. 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。

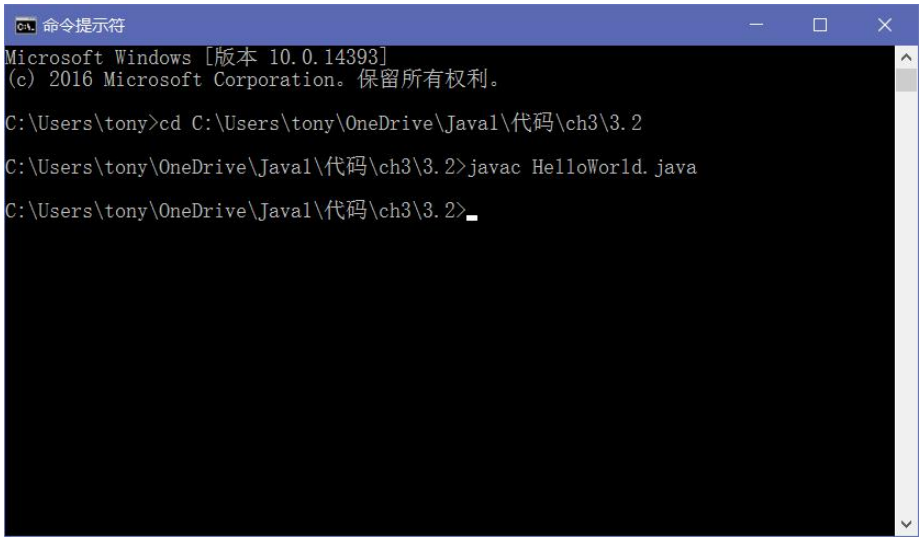


图 3-7 编译源文件

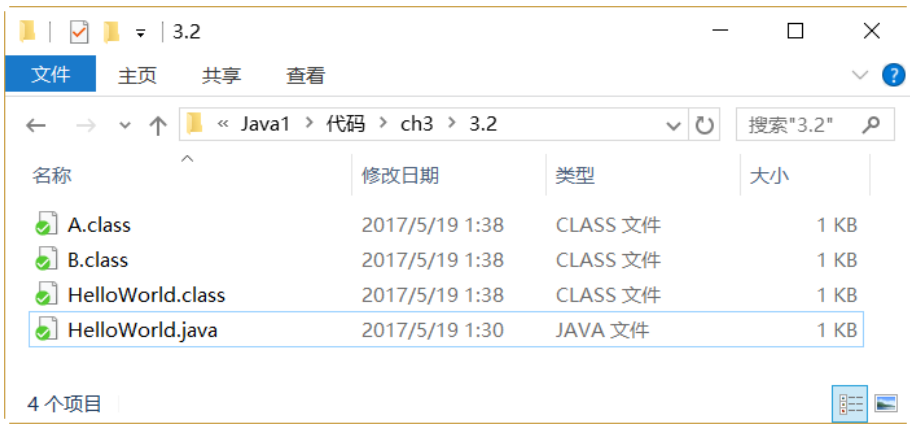


图 3-8 编译成功

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。

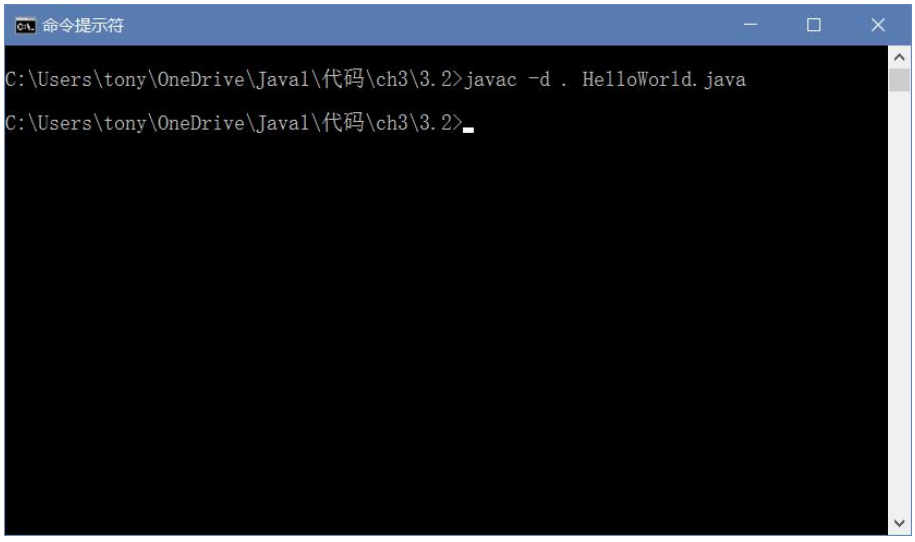


图 3-9 编译有包的源文件

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。

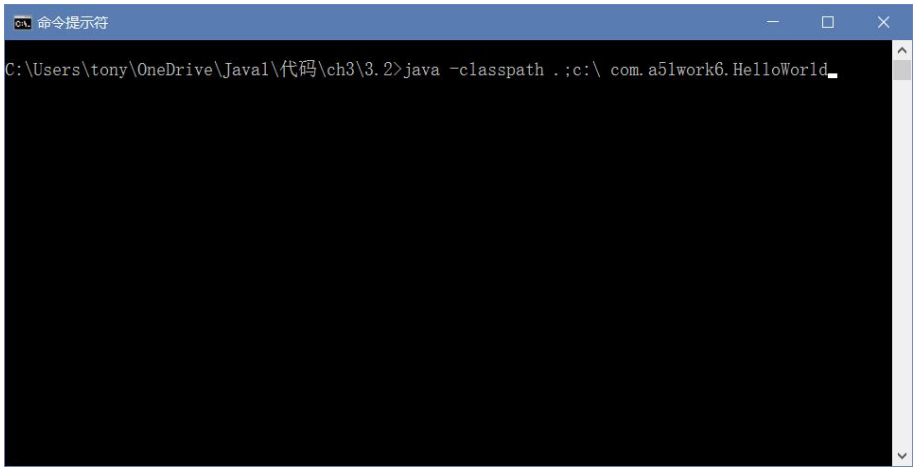


图 3-10 运行类文件

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，多个参数用空格分割。

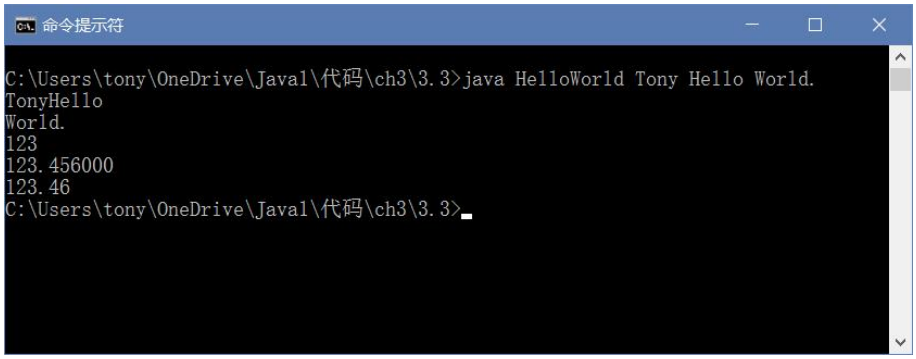


图 3-11 在命令行中运行程序

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%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：

1. 区分大小写：Myname 与 myname 是两个不同的标识符。
2. 首字符，可以是下划线（_）或美元符或字母，但不能是数字；
3. 除首字符外其他字符，可以是下划线（_）、美元符、字母和数字。
4. 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[7]。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

[7] Unicode 是国际组织制定的可以容纳世界上所有文字和符号的字符编码方案。

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 中有一些字符序列即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：

1. 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”。
2. 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。

1. 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;
```

2. 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");
            }

        }
}
```

3. 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 public static 修饰，用来保留字 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 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。

5.1 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 5-1 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。

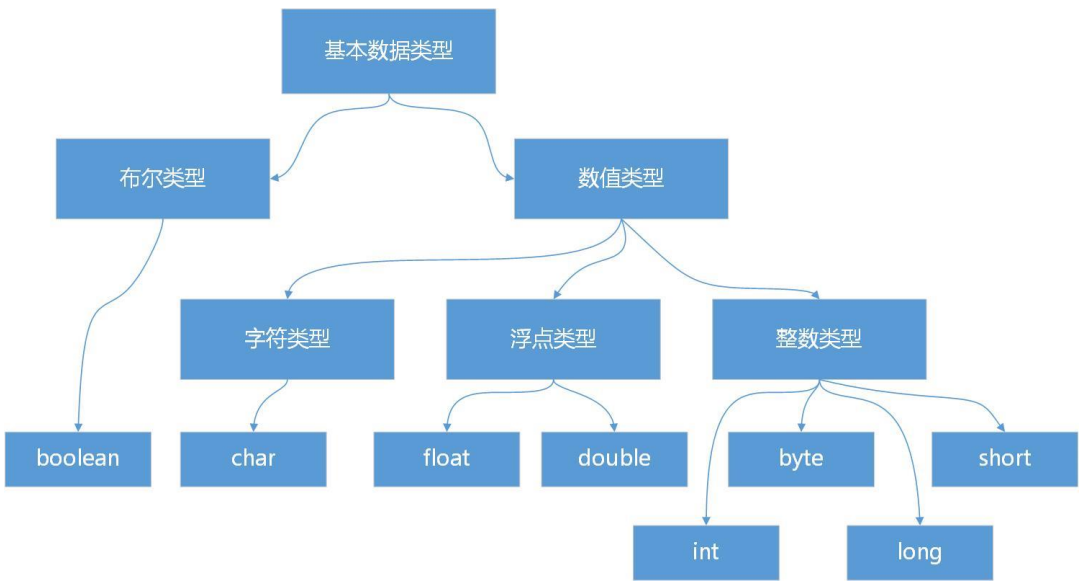


图 5-1　基本数据类型

5.2 整型类型

从图 5-1 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 为的，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 5-1 所示。

表 5-1 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。

5.3 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 5-2 所示。

表 5-2 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。

5.4 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。

5.4.1 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;
```

5.4.2 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}。

5.5 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 5-3。

表 5-3 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.
```

5.6 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。

5.7 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 5-1 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。

5.7.1 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 5-2 所示，从左到右是自动。

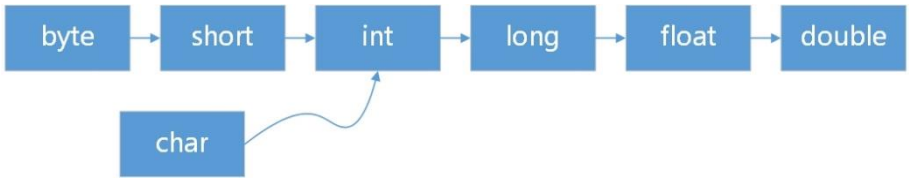


图 5-2　数据类型转换顺序

注意　如图5-2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，然后在进行计算。计算规则如表 5-4 所示。

表 5-4 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。

5.7.2 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 5-2 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 保持的是 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。

5.8 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 5-3 所示，包含：类、接口和数组声明的数据类型。

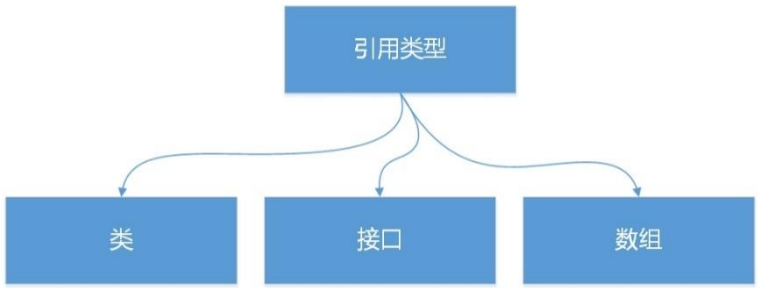


图 5-3　引用数据类型

提示　Java中的引用类型，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 5-4 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 5-4 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 5-5 所示。

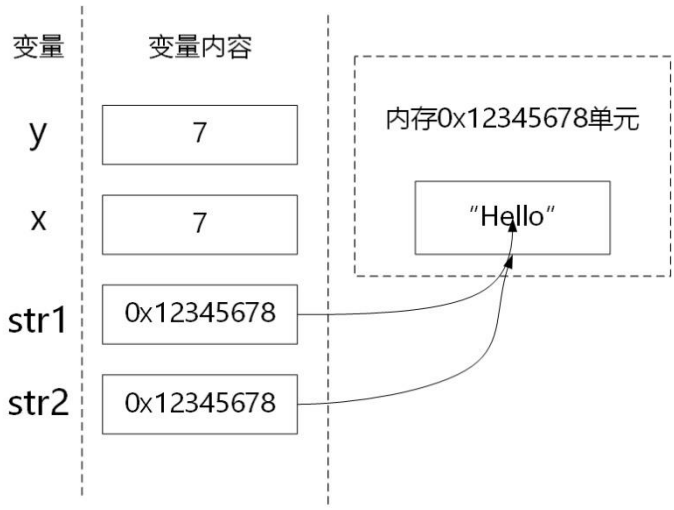


图 5-4　引用数据类型赋值过程 1

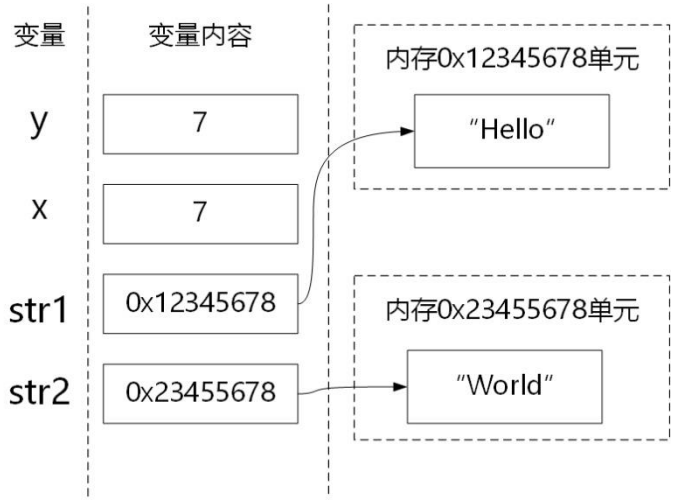


图 5-5　引用数据类型赋值过程 2

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6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。

6.1 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。

6.1.1 一元运算符

算术一元运算一共有 3 个，分别是-、++和--。具体说明参见表 6-1。

表 6-1 一元算术运算

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 6-1 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。

6.1.2 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 6-2。

表 6-2 二元算术运算

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 65。其他代码比较简单不再赘述。

6.1.3 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 6-3。

表 6-3 算术赋值符

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。

6.2 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 6-4。

表 6-4 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类型。当用于引用类型比较时，比较的是两个引用是否指向同一个对象，但当时实际开发过程多数情况下，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
```

6.3 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 6-5。

表 6-5 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。

6.4 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 6-6。

表 6-6 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。

6.5 其他运算符

除了前面介绍的主要运算符，Java 还有一些其他一些运算符。

- 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。

6.6 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 6-7 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 6-7 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7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

7.1 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。

7.1.1 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。

1. 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！
```

2. 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
```

3. 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。

7.1.2 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，因此本来中 case 7 和 case 6 都执行相同的代码。

7.2 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。

7.2.1 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。

7.2.2 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。

7.2.3 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 7-1 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。

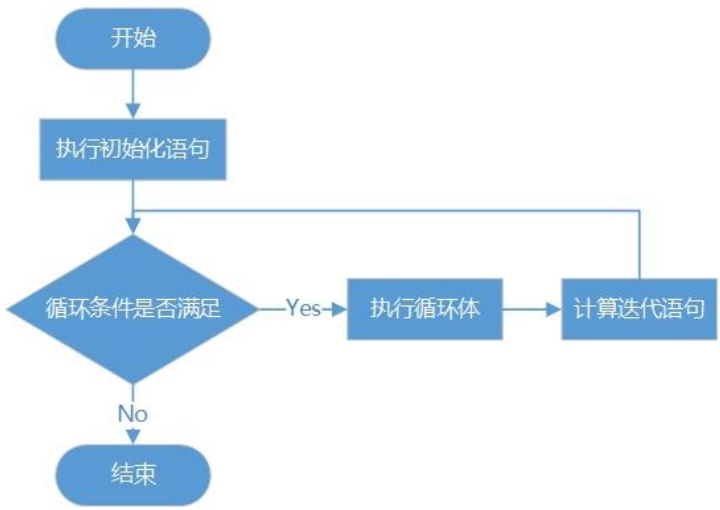


图 7-1　for 循环执行流程图

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)
```

7.2.4 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 8 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，的 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。

7.3 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。本章重点介绍 break 和 continue 语句的使用。throw 和 return 将后面章节介绍。

7.3.1 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。

7.3.2 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8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：

1. 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。
2. 有序性：数组中的元素是有序的，通过下标访问。
3. 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。

8.1 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。

8.1.1 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;
```

8.1.2 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。

1. 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。

2. 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8-1所示。

表 8-1 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 8-1(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 8-1(b)所示。

当代码第③行执行完成，stringArray 数组内容如图 8-2(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 8-2(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。

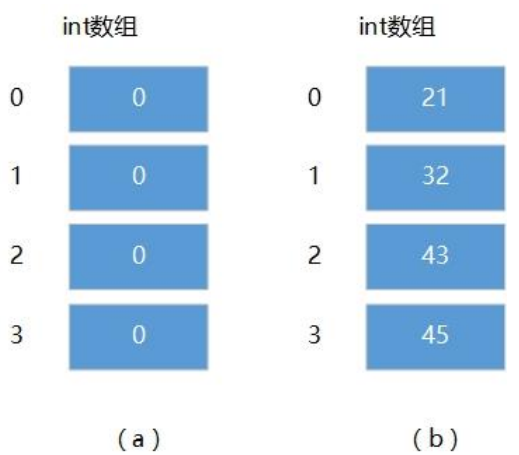


图 8-1 intArray 数组

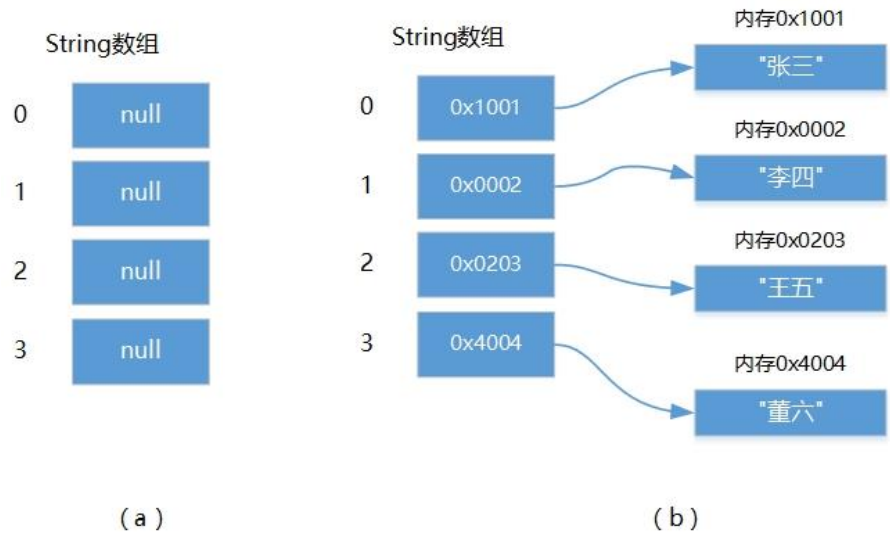


图 8-2 stringArray 数组

8.1.3 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。

8.2 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。

8.2.1 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[];
```

8.2.2 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。

1. 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 8-3 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。

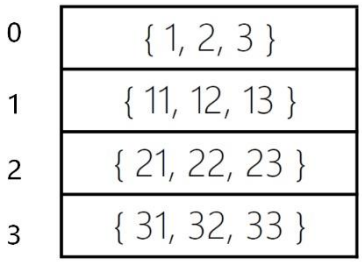


图 8-3　intArray 二维数组

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。

2. 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 8-4(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 8-4(b)所示。

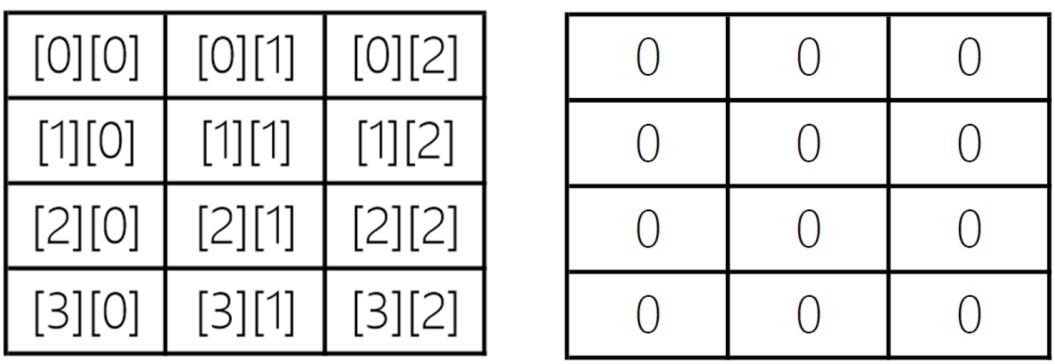


图 8-4　二维数组动态初始化

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。

8.2.3 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 8-5 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。

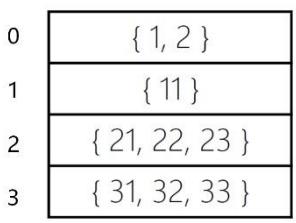


图 8-5　不规则数组

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 8-6 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。

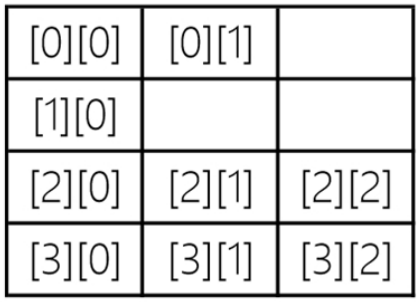


图 8-6　不规则数组访问

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9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。

9.1 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。

9.2 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。Java 官方为 Java SE 提供了基于 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 9-1 所示。

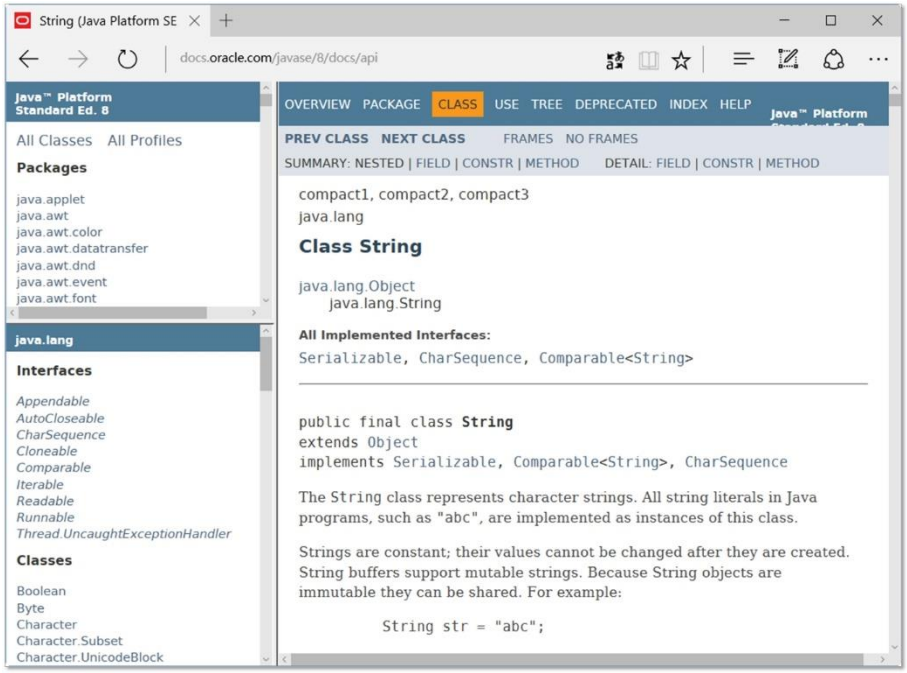


图 9-1　Java 8 在线 API 文档

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 9-2 所示，类和接口中，斜文字体显示是接口，正常字体才是类。

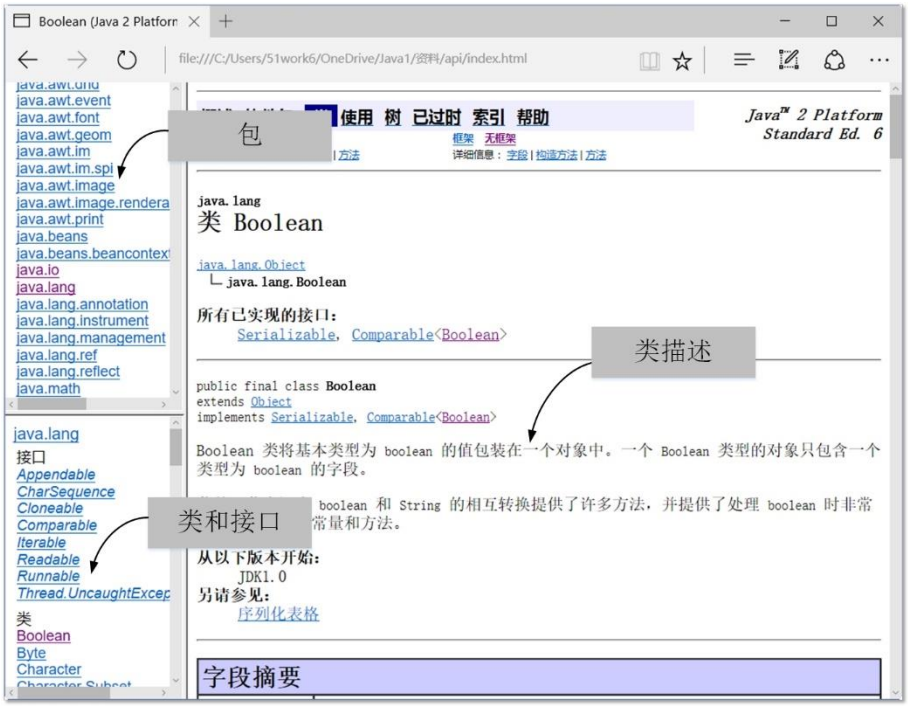


图 9-2　API 文档页面各个部分

在类窗口还有很多内容，向下拖曳滚动条会看到如图 9-3 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 9-4 所示单击了 compareTo 方法看到的详细信息。

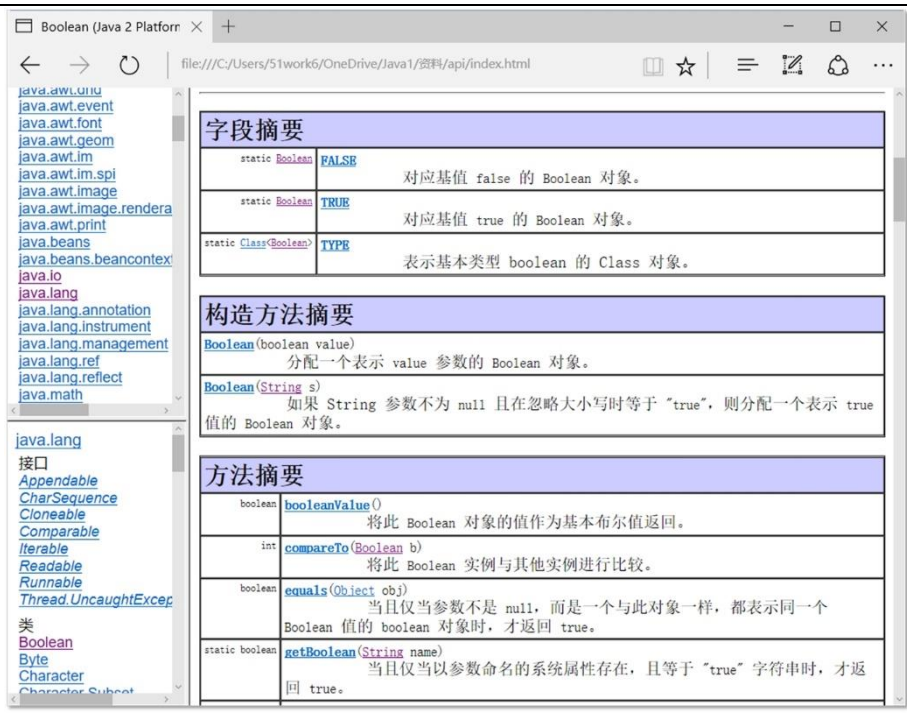


图 9-3　类窗口页面其他内容

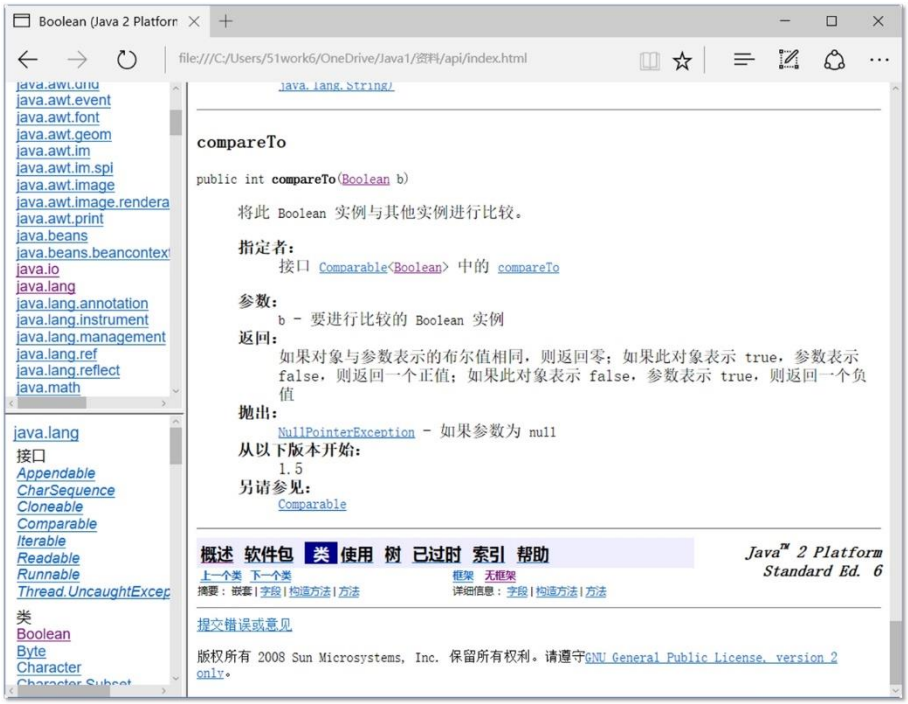


图 9-4　compareTo 方法详细描述

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。

9.3 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。

9.3.1 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。

9.3.2 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

9-5 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。

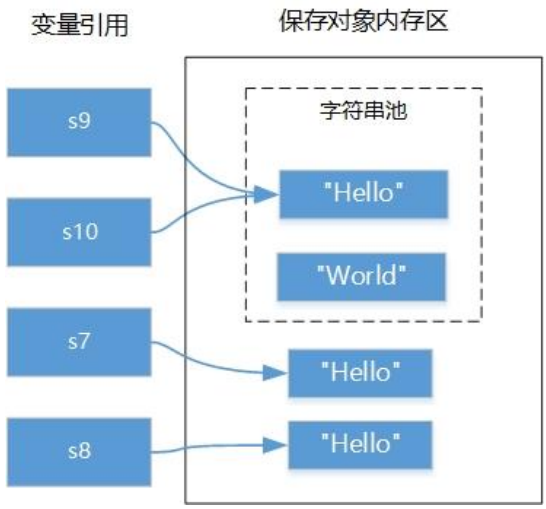


图 9-5　字符串池

9.3.3 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行业是使用+=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。

9.3.4 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，第一次找到它所在的索引:" + 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，第一次找到它所在的索引: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 9-6 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 9-6 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。

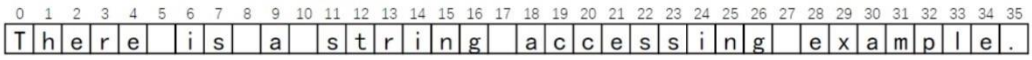


图 9-6　sourceStr 字符串索引

9.3.5 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、比较前缀和后缀串等。

1. 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。

2. 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。

3. 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。

9.3.6 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 9-6 所示。代码第①行是截取 example.子字符串，从图 9-6 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 9-6 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[]。

9.4 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。

9.4.1 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[8]，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。

StringBuffer 和 StringBuilder 具有完全相同的 API，即构造方法和方法等内容一样。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长度时，字符串缓冲区自动扩充容量，但这是以牺牲新能为代价的扩容。

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
```

[8] 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。

字符串缓冲区容量：34

9.4.2 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。StringBuffer 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。

9.4.3 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);
```

运行输出结果：

```
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，原始字符串索引如同 9-7 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 9-8 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。

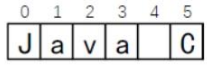


图 9-7　原始字符串索引

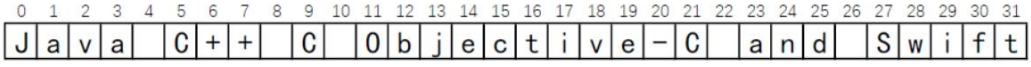


图 9-8　删除之前字符串索引

本章小结

本周介绍了 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10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。

10.1 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。

10.2 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。

10.2.1　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。

10.2.2　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。

10.2.3 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 12.4 节，这里不再赘述。

10.3 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。

10.3.1 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。

10.3.2 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：

1. public、protected 和 private 修饰符用于封装成员变量。
2. static 修饰符用于声明静态变量，所以静态变量也称为“类变量”。
3. 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。

10.3.3 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：

1. public、protected 和 private 修饰符用于封装方法。
2. static 修饰符用于声明静态方法，所以静态方法也称为“类方法”。
3. 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。

4. 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。
5. 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。

10.4 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。

10.4.1 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）[9]。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。

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。

10.4.2 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 10-1 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，会发现如同 10-2 所示的层次结构，源文件目录是根目录，也是默认包目录，

[9] 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

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。

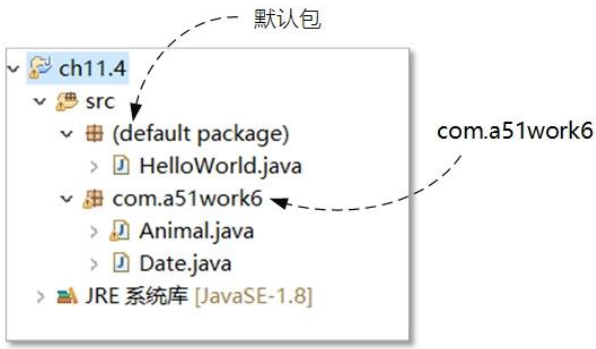


图 10-1　Eclipse 包资源视图中查看包

```
<源文件目录>
├─HelloWorld.java
└─com
    └─a51work6
        ├─ Animal.java
        └─ Date.java
```

图 10-2　文件系统目录与包

10.4.3　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。

10.4.4 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。

1. 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。

2. 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。

3. java.net 包

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

4. 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。

5. 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。

6. 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。

10.5 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 10-3 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 10-3 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。

10.6 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 10-1 所示。

表 10-1 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。

10.6.1 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。

10.6.2 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。

10.6.3 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，可以公有的 printX()方法。

10.6.4 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。

10.7 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。

10.8 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11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。前面章节已经多少用到了对象，这一章详细介绍一下对象的创建和销毁等相关知识。

11.1 创建对象

创建对象包括两个步骤：声明和实例化。

1. 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。

2. 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 11-1 所示。

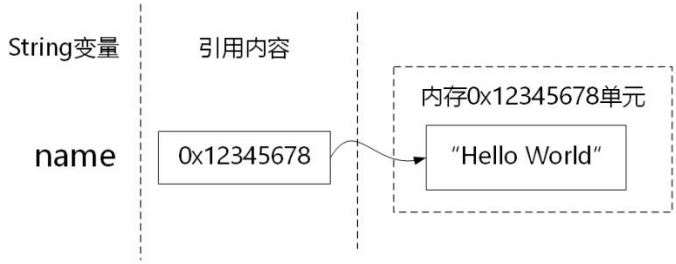


图 11-1　对象实例化

11.2 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给我们的。第一种程序员必须防止这种情况发生，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。**

11.3 构造方法

在 11.1 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：

1. 构造方法名必须与类名相同。
2. 构造方法没有任何返回值，包括 void。
3. 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。

11.3.1 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。

11.3.2 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。

11.3.3 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[10]等设计。

11.4 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。

[10] 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。

- 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 调用局部变量，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，这本例中 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。

11.5 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，程序员不用关系释放的细节。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 关键的使用。

第12章 继承与多态

类的继承性是面向对象语言的基本特性，多态性前提是继承性。Java 支持继承性和多态性。这一章讨论 Java 继承性和多态性。

12.1 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[11]，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12-1所示。类图中的各个元素说明如图12-2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12-3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。

[11] UML 是 Unified Modeling Language 的缩写，既统一标准建模语言。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。

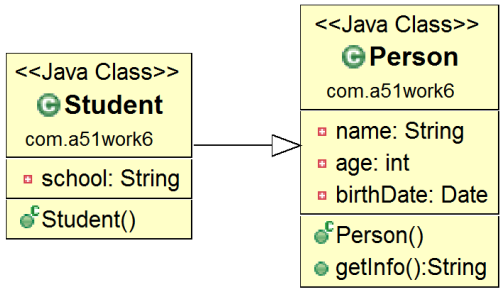


图 12-1　Student 继承 Person 的类图

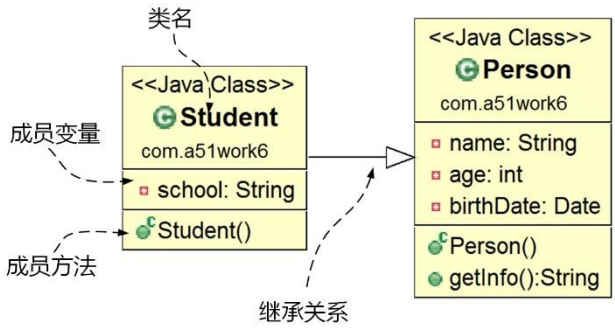


图 12-2　类图中元素

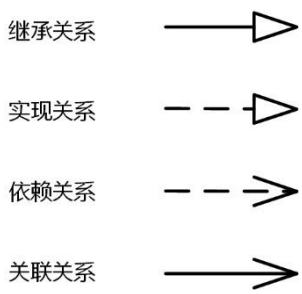


图 12-3　元素之间关系

12.2 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 12-4 所示。

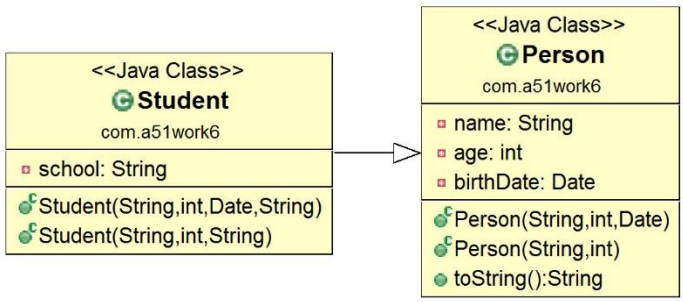


图 12-4　Person 和 Student 类图

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：

1. 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。
2. 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。
3. 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。

12.3 成员变量隐藏和方法覆盖

子类继承父类后，有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？

12.3.1 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。

12.3.2 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：

1. 提高程序的可读性。
2. 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　方法重写时应遵循的原则：

1. **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。**
2. **覆盖后的方法不能比原方法产生更多的异常。**

12.4 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。

12.4.1　多态概念

发生多态要有三个前提条件：

1. 继承。多态发生一定要子类和父类之间。

2. 覆盖。子类覆盖了父类的方法。
3. 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 12-5 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。

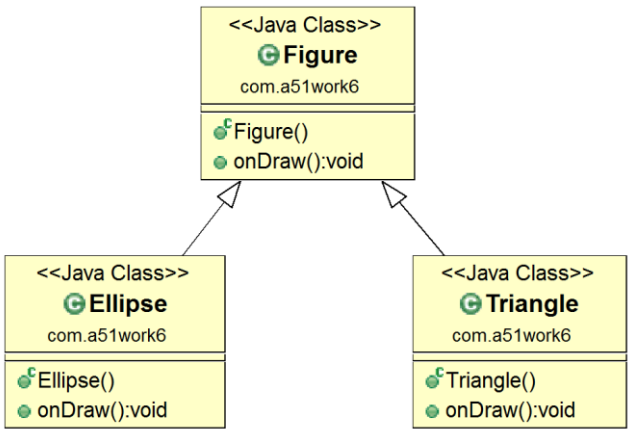


图 12-5　几何图形类图

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。

12.4.2 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，如同 12-6 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。

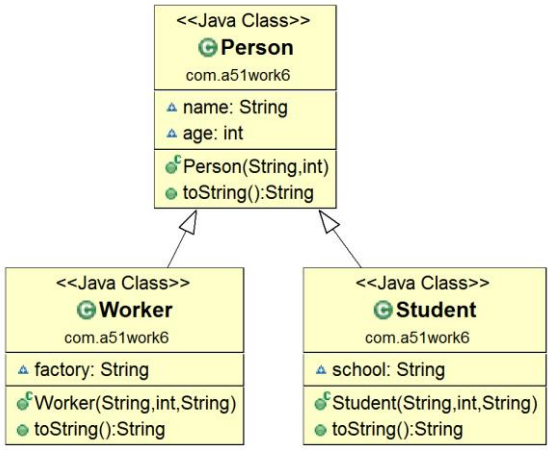


图 12-6　继承关系类图

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

12.4.3 引用类型转换

在 5.7 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，只有属于同一颗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 3 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 12-1 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 12-1 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。

12.5 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。

12.5.1 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。

1. final 修饰的局部变量必须使用之前被赋值一次才能使用。
2. 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。

12.5.2 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，这是可以使用 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。

12.5.3 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13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。

13.1 抽象类

Java 语言提供了两种类：一种是具体类；另一种是抽象了。前面章节接触的类都是具体类。这一节介绍一下抽象类。

13.1.1　抽象类概念

在 13.4.1 节介绍多态时候，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 13-1 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。

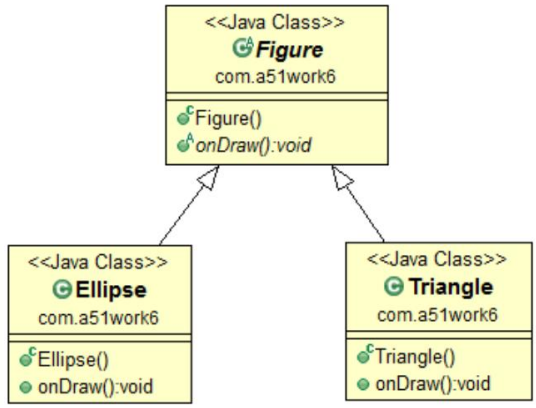


图 13-1　抽象类几何图形类图

提示　在UML类图抽象类和抽象方法字体是斜体的，见图13-1所示中的Figure类和onDraw方法都是斜体的。

13.1.2 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。

13.2 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。

13.2.1　接口概念

其实 13.1.1 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 13.1.1 节几何图形类改成接口后，类图如图 13.2 所示。

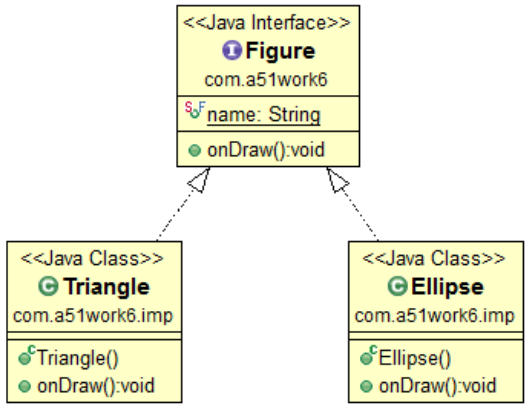


图 13-2　接口几何图形类图

提示　在UML类图中接口的图标是“I”，见图13-2所示中的Figure接口。类的图标是“C”，见图13-2所示中的Triangle接口。

13.2.2　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。

13.2.3 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，如果两个父类在有相同的方法，那么子类如何继承哪一个方法呢？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 13-3 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。

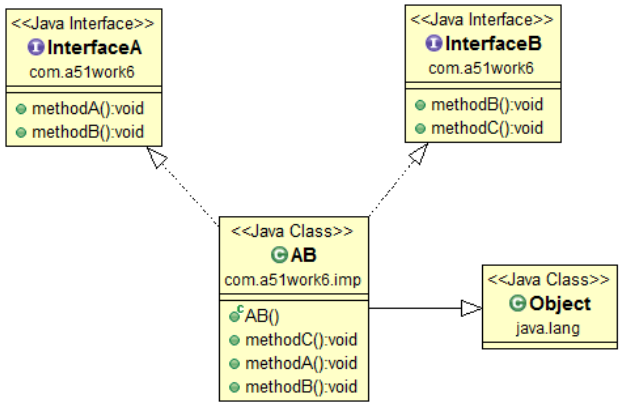


图 13-3　多继承类图

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。

13.2.4 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。如同 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，从图可见 ABC 需要实现 InterfaceA 和 InterfaceB 接口中的所有方法。

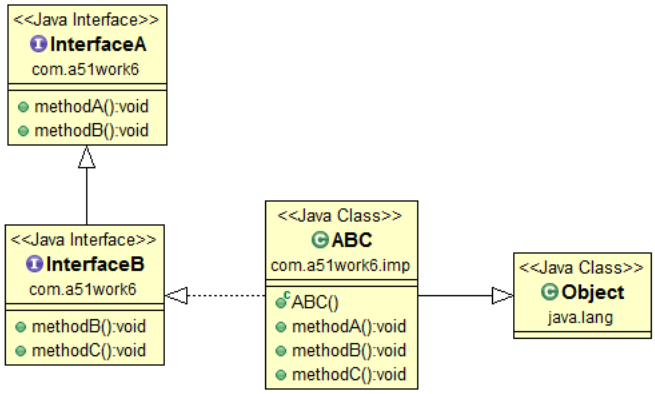


图 13-4 接口继承类图

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。

13.2.5 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：

1. 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。
2. 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。

13.3 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所了解，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：

1. 接口支持多继承，而抽象类（包括具体类）只能继承一个父类。
2. 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。
3. 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。
4. 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14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。

14.1 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。

14.2 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 14-1 所示。

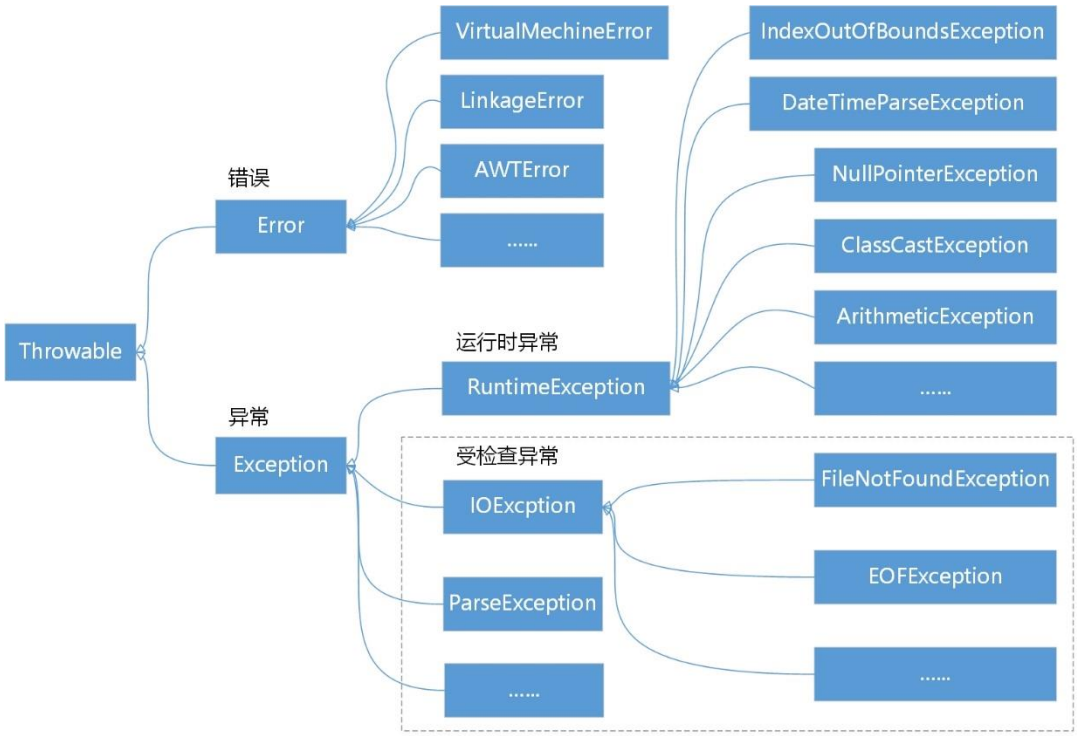


图 14-1　Java 异常类继承层次

14.2.1　Throwable 类

从图 14-1 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 14.1 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。

14.2.2 Error 和 Exception

从图 14-1 可见，Throwable 有两个直接子类：Error 和 Exception。

1. 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。

2. 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。

14.2.3 受检查异常和运行时异常

从图 14-1 可见，Exception 类可以分为：受检查异常和运行时异常。

1. 受检查异常

如图 14-1 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。

2. 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 14.2.1 节除零示例的 ArithmeticException 异常属于 RuntimeException 异常，见图 14-1 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14.2.1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 14-1 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。

14.3 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。

14.3.1 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
}
```

1. 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。**

---

2. 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。

14.3.2 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 14-1 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。

14.3.3 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 14.3.2 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。

14.3.4　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。

14.4 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。

14.4.1 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e1
} 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，如同 14-2 所示。

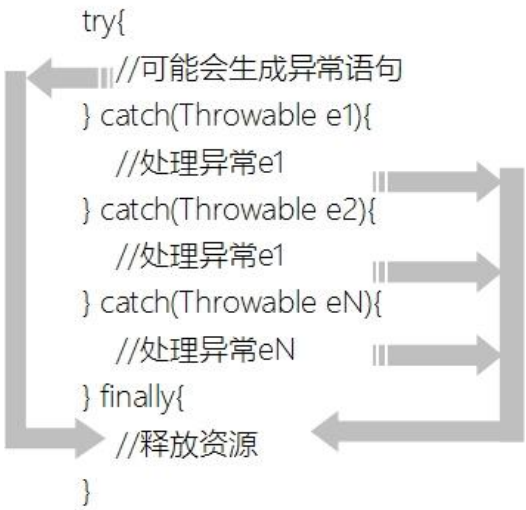


图 14-2　finally 代码块流程

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();
        }
    }
```

14.4.2 自动资源管理

14.4.1 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。

14.5 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 10.3.3 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 14.3 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，所有需要在代码第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。

14.6 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，

14.7 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 14.5 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 14.4 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15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。

15.1 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。如同 15-1 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 15-1 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。

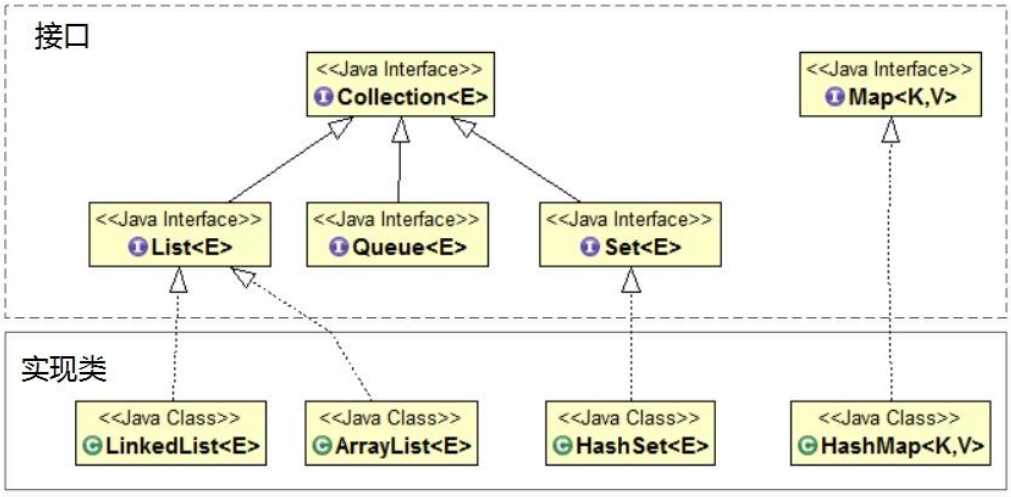


图 15-1　Java 主要集合接口和类

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。

15.2 List 集合

List 集合中的元素是有序的，可以重复出现。图 15-2 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”。

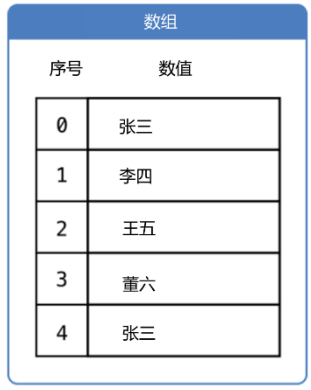


图 15-2 数组集合

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15-2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。

15.2.1 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。

1. 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 集合继承过了的。
- 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 集合继承过了的。
- clear()：从 List 集合中移除所有元素。该方法是从 Collection 集合继承过了的。

2. 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 集合继承过了的。
- 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 集合继承过了的。

3. 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。

4. 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 集合继承下来的。
- size()：返回 List 集合中的元素数，返回值是 int 类型。该方法是从 Collection 集合继承下来的。
- 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 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。

15.2.2 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：

1. 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。
2. 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。
3. 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。

15.3 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 15-3 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。

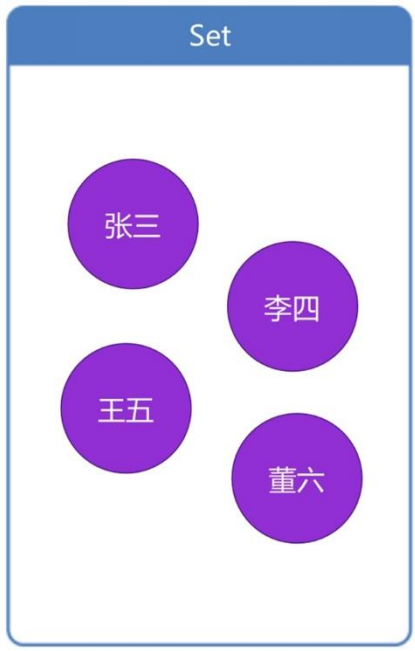


图 15-3　Set 集合

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 是基散列表数据结构的实现。

15.3.1　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。

1. 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 集合继承过了的。
- remove(Object element)：如果 Set 集合中存在指定元素，该方法是从 Collection 集合继承过了的。
- clear()：从 Set 集合中移除所有元素。该方法是从 Collection 集合继承过了的。

2. 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 集合继承过了的。
- 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 集合继承过了的。

3. 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 集合继承下来的。
- size()：返回 Set 集合中的元素数，返回值是 int 类型。该方法是从 Collection 集合继承下来的。

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。

15.3.2 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。

15.4 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 15-4 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。

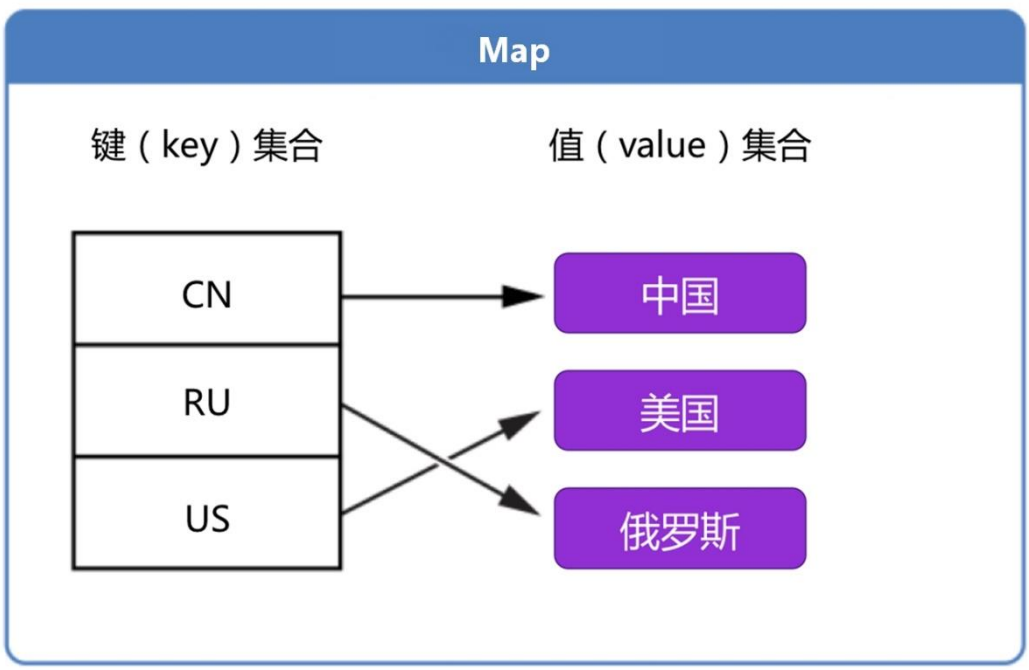


图 15-4 Map 集合

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 是基散列表数据结构的实现。

15.4.1 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。

1. 操作元素

- get(Object key)：返回指定键所对于的值；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。

2. 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。

3. 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，所有这里打印的 null。

15.4.2 遍历集合

Map 集合遍历与 List 和 Set 集合不同，Map 有两个集合，因此遍历过程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 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16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。

16.1 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。

16.1.1 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。

1. 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。

2. 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。

3. 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。

4. 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。

5. 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。

16.1.2　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，如同16-1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。

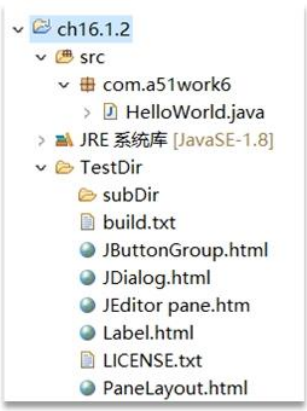


图 16-1　Eclipse 中的当前目录

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。

16.2 I/O 流概述

Java 将数据的输入输出（I/O）操作当做“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。

16.2.1　Java 流设计理念

如同 16-2 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 16-2　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。

16.2.2　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。

1. 字节输入流

字节输入流根类是 InputStream，如图 16-3 所示它有很多子类，这些类的说明如表 16-1 所示。

表 16-1 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

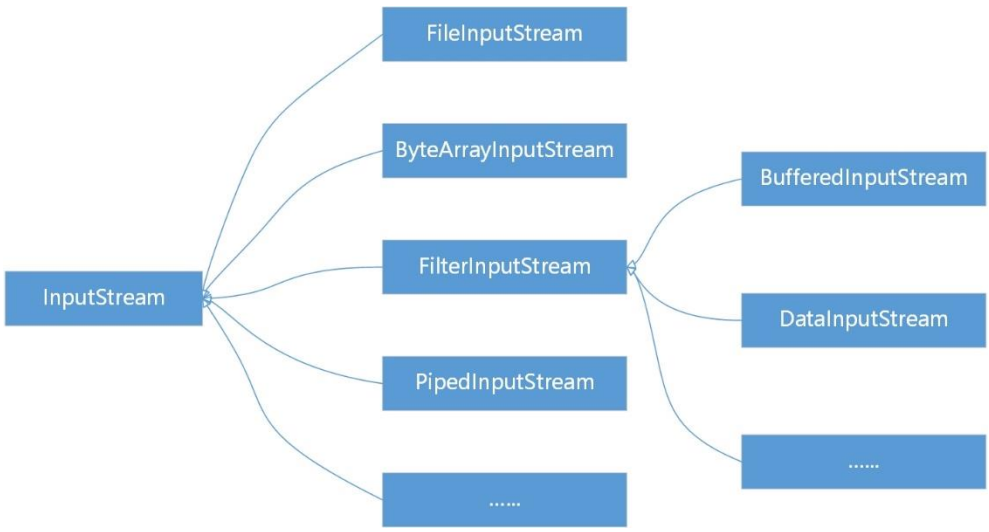


图 16-3　字节输入流类继承层次

2.　字节输出流

字节输出流根类是 OutputStream，如图 16-4 所示它有很多子类，这些类的说明如表 16-2 所示。

表 16-2 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

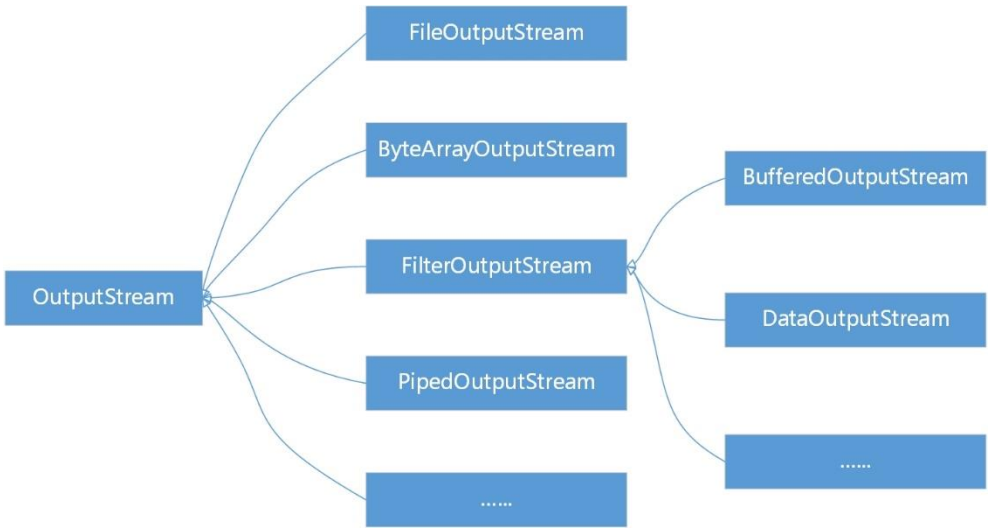


图 16-4　字节输出流类继承层次

3. 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 16-5 所示它有很多子类，这些类的说明如表 16-3 所示。

表 16-3 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

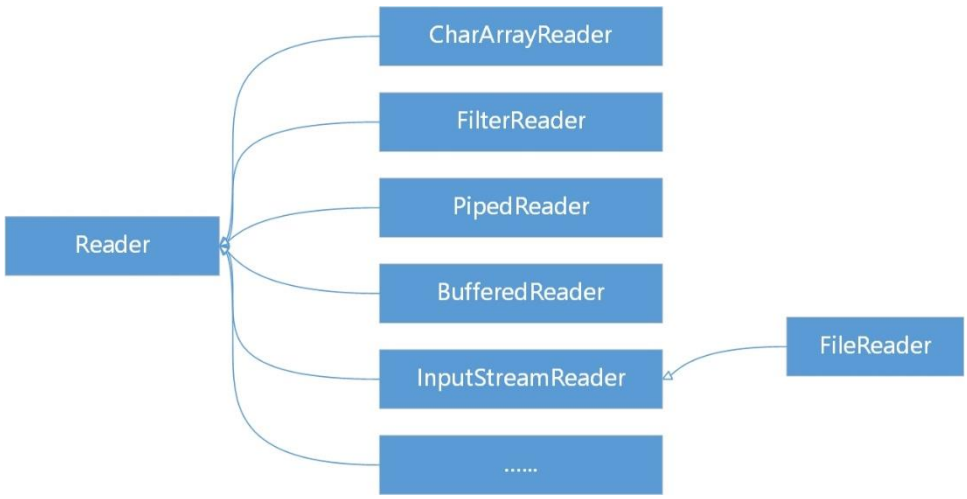


图 16-5　字符输入流类继承层次

4. 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 16-6 所示它有很多子类，这些类的说明如表 16-4 所示。

表 16-4 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 FilterReader 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

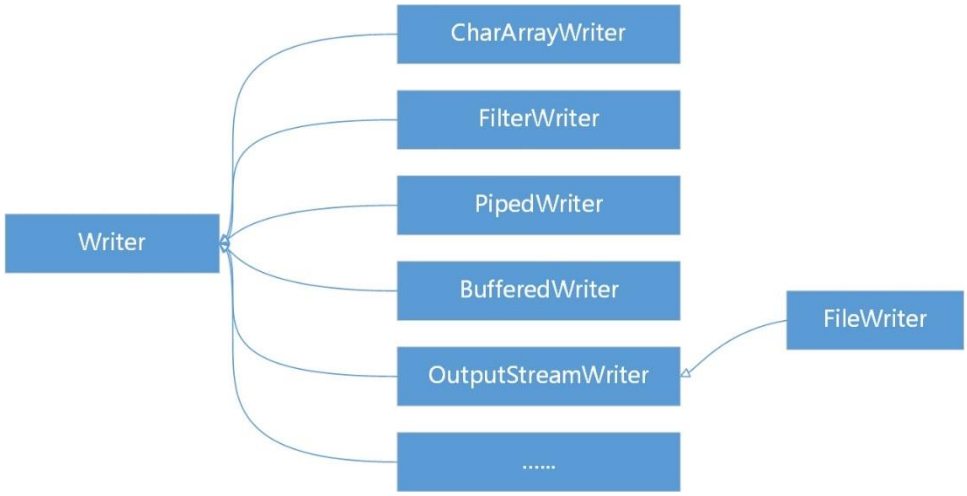


图 16-6　字符输出流类继承层次

16.3 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。

16.3.1 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。如果因为已经到达流末尾，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，如果因为已经到达流末尾，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，如果因为已经到达流末尾，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。

上述所有方法都可以会抛出 IOException，因此使用时要注意处理异常。

16.3.2 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14.4.2节。

16.3.3 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行也进行从输入流中读取数据。由于本例中缓冲区大小设置的 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。两次读取内容图 16-7 所示。

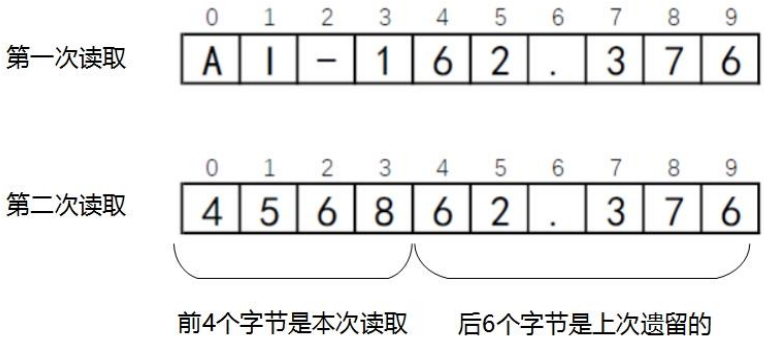


图 16-7　文件读取示意图

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的字节输入输出流，所以不仅可以复制文本文件，还有复制二进制文件。

16.3.4 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 16-3 和图 16-4 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，这样可提供缓冲区的利用率。

下面将 16.3.3 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 16.3.3 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。

16.4 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。

16.4.1 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。

上述所有方法都可以会抛出 IOException，因此使用时要注意处理异常。

16.4.2 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。

16.4.3 案例：文件复制

前面两个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 16.3.3 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 16.3.3 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。

16.4.4 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，这样可提供缓冲区的利用率。

BufferedWriter 特有方法和构造方法主要有：

- void newLine()：写入一个换行符。
- BufferedWriter(Writer out)：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(Writer out, 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，这样可提供缓冲区的利用率。

下面将 16.4.3 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。

16.4.5 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 16.4.3 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、InputStreamReader、BufferedReader、BufferedWriter、BufferedInputStream 和 BufferedOutputStream 等。

第17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底层次，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 URL 的高层次网络编程，以及数据交换格式。

17.1 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。

17.1.1　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。

1. 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 17-1 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。

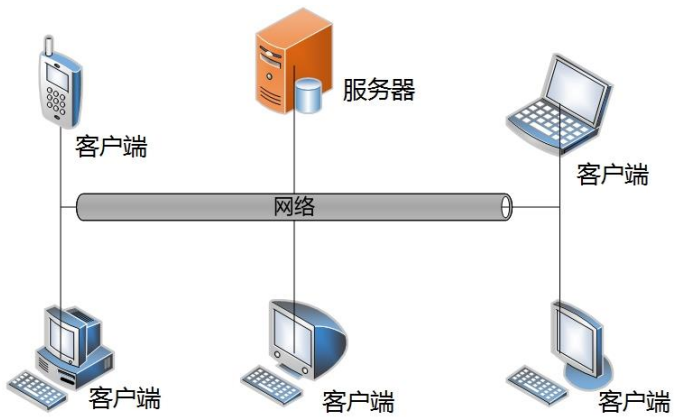


图 17-1　客户端服务器结构网络

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。

2. 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 17-2 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。

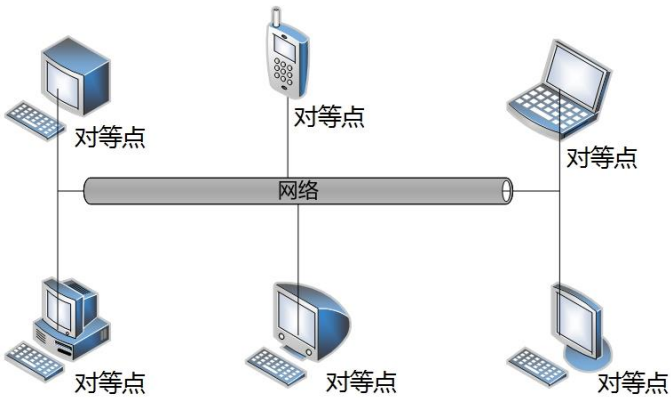


图 17-2　对等结构网络

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。

17.1.2　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。

17.1.3 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。

17.1.4 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。

17.2 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 17-3 所示的留言条。

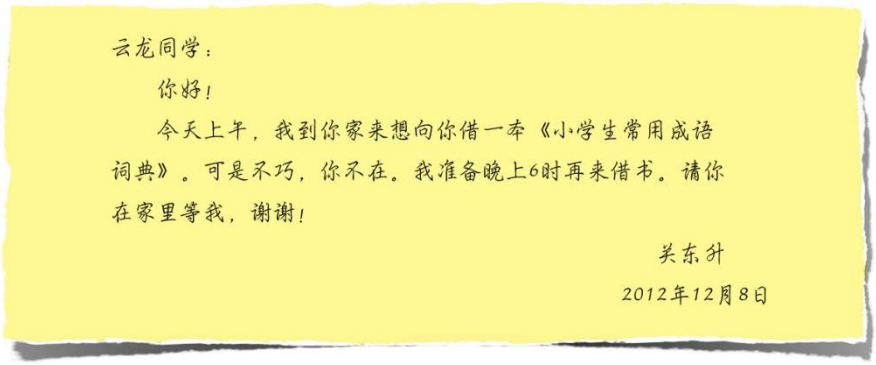


图 17-3　留言条

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 17-4 所示。

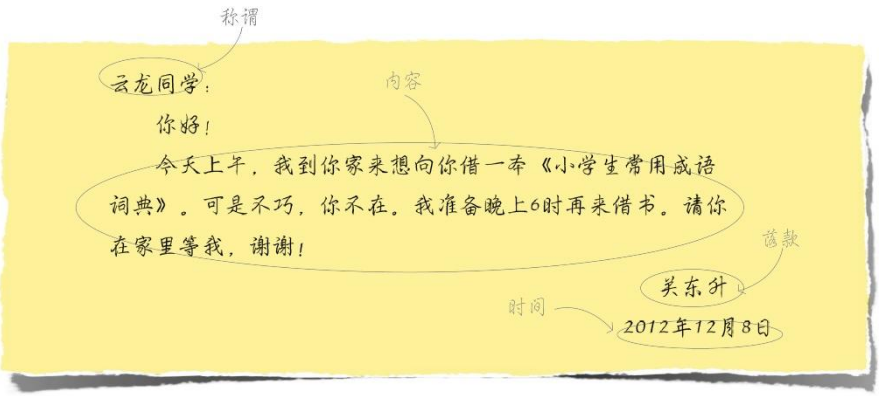


图 17-4　留言条格式

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时再来借书。请你在家里等我，谢谢！","关东升","2012年12月08日"
```

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割。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"}

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。

17.2.1 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 17-5 所示。

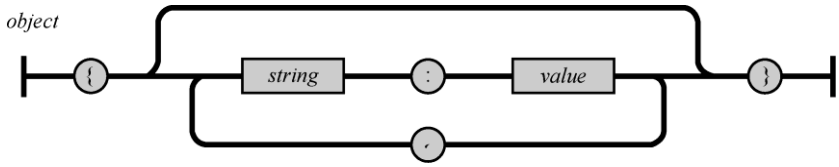


图 17-5　JSON 对象的语法表

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 17-6 所示。

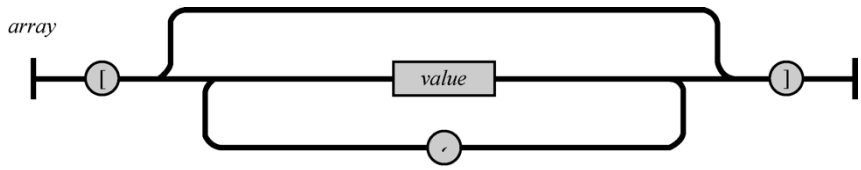


图 17-6　JSON 数组的语法表

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 17-7 所示。

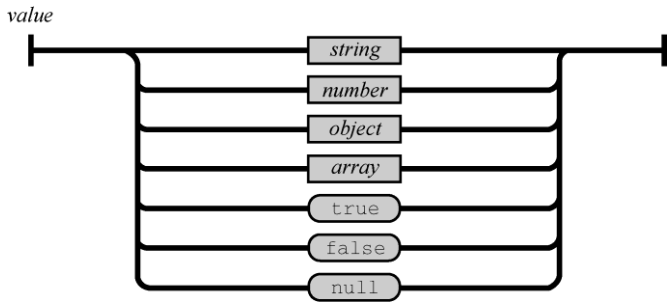


图 17-7　JSON 值的语法结构图

17.2.2　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 在线问档 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 17-8 所示。

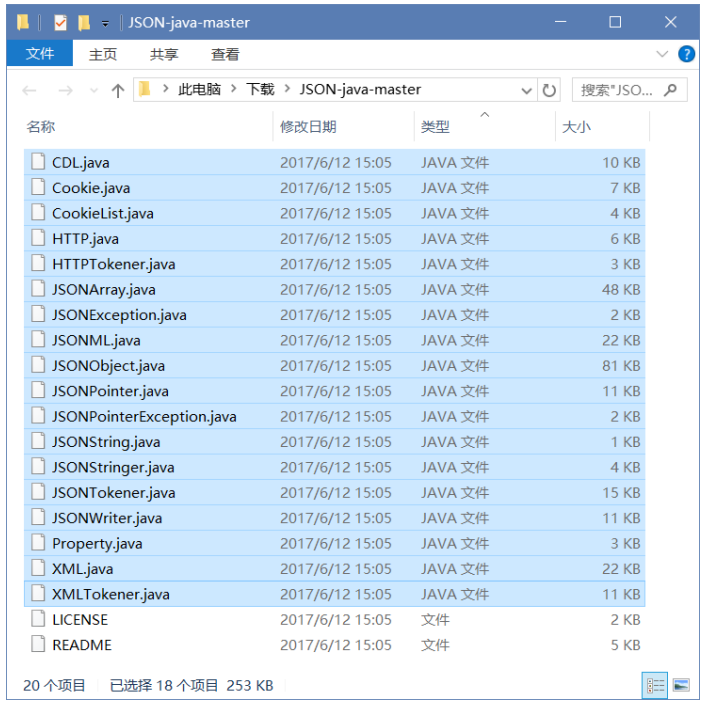


图 17-8　JSON-java 源代码文件

将 JSON-java 库源代码文件添加到工程中，需要两个步骤：

1. 创建 org.json 包

JSON-java 库中的源代码文件都隶属于 org.json 包，从图 17-8 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 17-9 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。

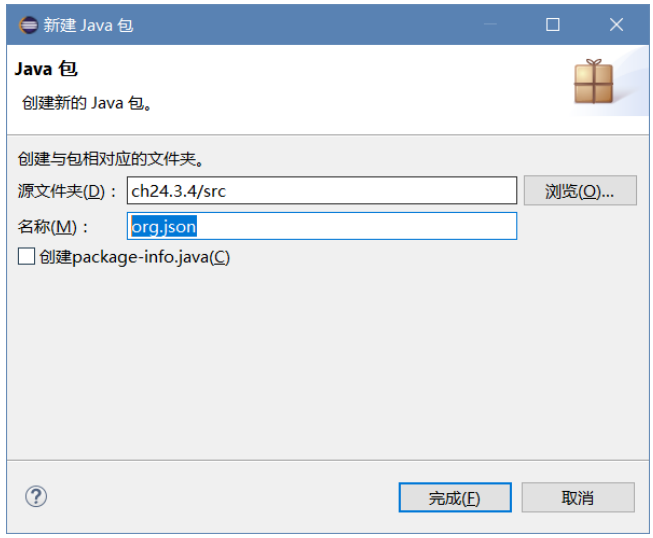


图 17-9　在 Eclipse 中创建包

2. 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 17-10 所示，将源代码文件复制到 Eclipse 中。

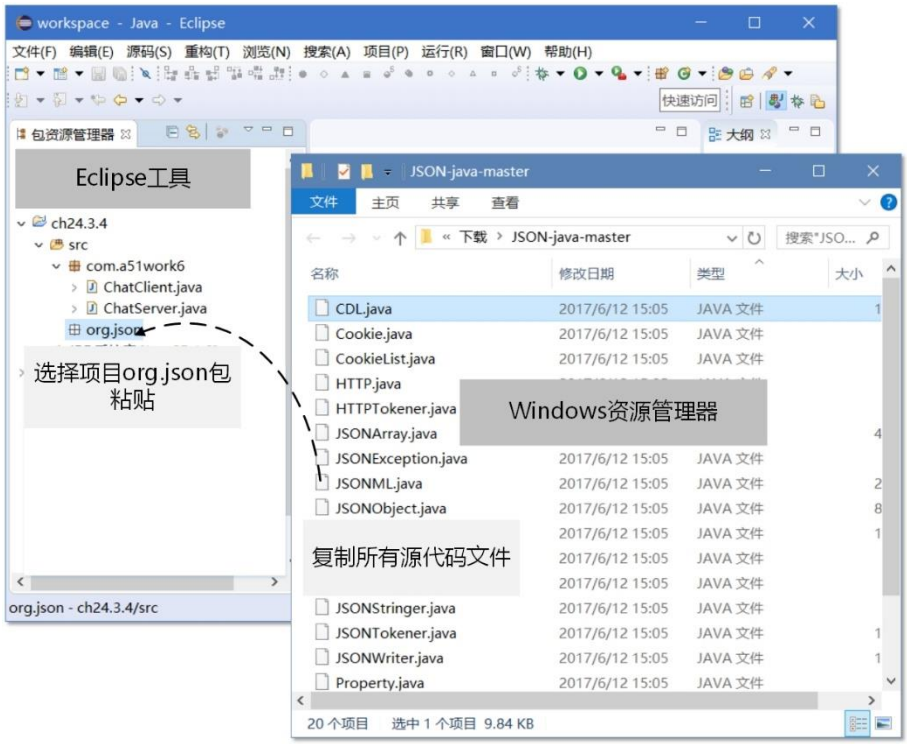


图 17-10　复制源代码文件到 Eclipse 工程

17.2.3　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。

1. 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。

2. 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'}]}。

17.3 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。

17.3.1 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
```

17.3.2 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。

1. 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。作为 Web 服务器，必须实现 GET 和 HEAD 方法，其他方法都是可选的。

- 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。

2. 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。

17.3.3 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。

17.3.4 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 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 是抽象了，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服务是有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www.51work6.com进行注册，注册时需要提供你自己有效的邮箱，这个邮箱用来激活用户。在网络请求时需要提交email参数，这个参数是你在注册时填写

的邮箱。

17.3.5 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(paramString)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。

17.3.6 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。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。

后　　记

如果我的图书能够给您带来帮助，希望您能慷慨捐助，捐助方法可以通过手机扫描下面的二维码实现。

微信支付

支付宝支付

另外，您想购买收费版本或享受更多的服务可通过，购买我们的电子书，我们会提供电子书、课件、配套视频和答疑服务。

配套视频观看地址：

http://www.zhijieketang.com/classroom/6/introduction

电子书百度阅读地址：

https://yuedu.baidu.com/ebook/7c1499987e192279168884868762caaedd33ba00

电子书图灵社区地址：

http://www.ituring.com.cn/book/2480