



全面阐释Java 7在语法、JVM、API类库等方面的所有重要新功能和新特性，可帮助开发者大幅度提升编码效率和代码质量

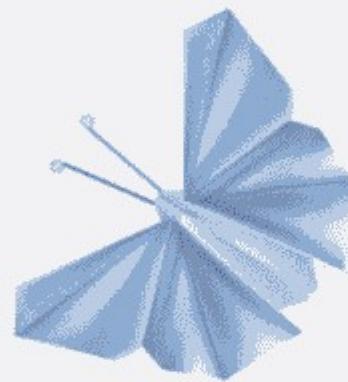
对JVM、源代码和字节代码操作、类加载器、对象生命周期、多线程、并发编程、泛型、安全等Java平台的核心技术进行深入解析，包含大量最佳实践



深入理解 Java 7

核心技术与最佳实践

Understanding the Java 7
the Core Techniques and Best Practice



NLIC2970800422

成富 著



机械工业出版社
China Machine Press

Understanding the Java 7

the Core Techniques and Best Practice



Java 7六年磨一剑，它在前一个版本上发生了很大的变化，引入了非常多激动人心的新特性和新功能，在语法、Java虚拟机、I/O、安全性、并发和国际化等方面都有重要的更新。作为国内第一本Java 7方面的专著，本书全面介绍了这些重要的更新，对有一定开发经验的Java程序员来说非常宝贵，能极大地降低他们的学习成本。此外，本书还探讨了各项Java技术的底层原理，对于想深入学习Java的读者尤为有价值。强烈推荐！

——51CTO (www.51cto.com) 中国领先的IT技术网站

长期以来，Java一直雄踞TIOBE编程语言排行榜的首位，它拥有人数最庞大的开发者社区。Java的每一次版本更新都会对语言、虚拟机和API类库等进行重要更新，Java 7这个版本尤为明显。Java 7的一个显著改变就是它提供的新特性能让开发者通过简化代码来提升开发效率和编码质量。本书以“深入理解”为宗旨，对Java 7中新增的内容和之前版本中已经非常成熟的核心技术都进行了非常深入的讨论，是系统学习Java 7和Java技术进阶的必备参考书。书中还给出了非常多的代码示例，可以帮助读者更好地在实际中应用这些知识。

——秦小波 资深Java技术专家
著有畅销书《设计模式之禅》和《编写高质量代码：改善Java程序的151个建议》

学习一门技术，要想做到“知其然”并不难，难的是要“知其所以然”，而且这更重要，也是考查一个技术人员真实实力的标杆。想跨越这个标杆，除了自己付出时间和精力之外，高人的点拨和指导更能起到事半功倍之效。成富是InfoQ中文站的资深专栏作者，从读者对他的“Java深度历险”系列技术文章的反馈可以看出：他不仅能帮你“知其然”，而且还能通过深入浅出的讲解让你“知其所以然”。他的这本书一定能让你的Java技能更上一层楼。

——郑柯 InfoQ中文站总编辑

作者简介

成富资深Java软件工程师，有多年Java企业级应用开发经验，对Java 7和Java平台的各项技术的底层原理有深入透彻的研究。曾就职于IBM中国研发中心，先后在IBM新技术学院和Lotus部门参与了多个重要产品的开发工作，现就职于新西兰PropellerHead公司。他是非常受欢迎的技术作家，在IBM developerWorks上发表中英文技术文章近30篇，获得了其颁发的“极具人气作者奖”；他还是知名技术网站InfoQ的专栏作家，撰写了“Java深度历险”专栏，共发表技术文章10余篇。此外，他还非常精通HTML 5、CSS 3、JavaScript等Web 2.0核心技术，实战经验丰富。

客服热线：(010) 88378991, 88361066
购书热线：(010) 68326294, 88379649, 68995259
投稿热线：(010) 88379604
读者信箱：hzjsj@hzbook.com



华章网站 <http://www.hzbook.com>

网上购书：www.china-pub.com

上架指导：计算机 程序设计 Java

ISBN 978-7-111-38039-9



9 787111 380399

定价：79.00元



深入理解 Java 7

核心技术与最佳实践

Understanding the Java 7

the Core Techniques and Best Practice



成富 著



NLIC2970800422



机械工业出版社
China Machine Press

本书是学习 Java 7 新功能和新特性以及深入理解 Java 核心技术的最佳选择之一。经过近 6 年的等待，Java 迎来了它的又一个历史性的版本——Java 7。Java 7 在提高开发人员的生产效率、平台性能和模块方向上又迈进了一步，变得更加强大和灵活。本书不仅对 Java 7 的所有重要更新进行了全面的解读，而且还对 Java 平台的核心技术的底层实现进行了深入探讨，包含大量最佳实践。

全书的主要内容可分为三大部分：第一部分是 1～6 章，全面阐释 Java 7 在语法、JVM、类库和 API 等方面的所有重要新功能和新特性，掌握这部分内容有助于大幅度提升编码效率和提高代码质量；第二部分是 7～13 章，对 JVM、Java 源代码和字节代码操作、类加载器、对象生命周期、多线程、并发编程、泛型、安全等 Java 平台的核心技术进行了深入解析，掌握这部分内容有助于深入理解 Java 的底层原理；第三部分为第 14 章，是对 Java 8 的展望，简要介绍了 Java 8 中将要增加的新特性。

封底无防伪标均为盗版

版权所有，侵权必究

本书法律顾问 北京市展达律师事务所

图书在版编目（CIP）数据

深入理解 Java 7：核心技术与最佳实践 / 成富著. —北京：机械工业出版社，2012.5

ISBN 978-7-111-38039-9

I. 深… II. 成… III. JAVA 语言－程序设计 IV. TP312

中国版本图书馆 CIP 数据核字（2012）第 068801 号

机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码 100037）

责任编辑：朱秀英

北京京师印务有限公司印刷

2012 年 5 月第 1 版第 1 次印刷

186mm×240mm • 29.25 印张

标准书号：ISBN 978-7-111-38039-9

定价：79.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：(010) 88378991；88361066

购书热线：(010) 68326294；88379649；68995259

投稿热线：(010) 88379604

读者信箱：hzjsj@hzbook.com

前　　言

为什么要写这本书

我最早开始接触 Java 语言是在大学的时候。当时除了用 Java 开发一些小程序之外，就是用 Struts 框架开发 Web 应用。在后来的实习和工作中，我对 Java 的使用和理解更加深入，逐渐涉及 Java 相关的各种不同技术。使用 Java 语言的一个深刻体会是：Java 语言虽然上手容易，但是要真正掌握并不容易。

Java 语言对开发人员屏蔽了一些与底层实现相关的细节，但是仍然有很多内容对开发人员来说是很复杂的，这些内容恰好是容易出现错误的地方。我在工作中就经常遇到与类加载器和垃圾回收相关的问题。在解决这些问题的过程中，我积累了一些经验，遇到类似的问题可以很快地找到问题的根源。同时，在解决这些实际问题的过程中，我意识到虽然可以解决某些具体的问题，但是并没有真正理解这些问题的解决办法背后所蕴含的基本原理，仍然还只是处于一个“知其然，不知其所以然”的状态。于是我开始阅读 Java 相关的基础资料，包括 Java 语言规范、Java 虚拟机规范、Java 类库的源代码和其他在线资料等。在阅读的基础上，编写小程序进行测试和试验。通过阅读和实践，我对 Java 平台中的一些基本概念有了更加深入的理解。从 2010 年开始，我对积累的相关知识进行了整理，在 InfoQ 中文站的“Java 深度历险”专栏上发表出来，受到了一定的关注。

2011 年 7 月，在时隔数年之后，Java 的一个重大版本 Java SE 7 发布了。在这个新的版本中，Java 平台增加了很多新的特性。在 Java 虚拟机方面，invokedynamic 指令的加入使虚拟机上的动态语言的性能得到很大的提升。这使得开发人员可以享受到动态语言带来的在提高生产效率方面的好处。在 Java 语言方面，语言本身的进步进一步简化，使开

发人员编写代码的效率更高。在 Java 类库方面，新的 IO 库和同步实用工具类为开发人员提供了更多实用的功能。从另外一个角度来说，Java SE 7 是 Oracle 公司收购 Sun 公司之后发布的第一个 Java 版本，从侧面反映出了 Oracle 公司对 Java 社区的领导力，可以继续推动 Java 平台向前发展。这可以打消企业和社区对于 Oracle 公司领导力的顾虑。Java SE 7 的发布也证明了基于 JCP 和 OpenJDK 的社区驱动模式可以很好地推动 Java 向前发展。

随着新版本的发布，肯定会有越来越多的开发人员想尝试使用 Java SE 7 中的新特性，毕竟开发者社区对这个新版本期待了太长的时间。在 Java 程序中使用这些新特性，可以提高代码质量，提升工作效率。Java 平台的每个版本都致力于提高 Java 程序的运行性能。随着新版本的发布，企业都应该考虑把 Java 程序的运行平台升级到最新的 Java SE 7，这样可以享受到性能提升所带来的好处。对于新的 Java 程序开发，推荐使用 Java SE 7 作为标准的运行平台。本书将 Java SE 7 中的新特性介绍和对 Java 平台的深入探讨结合起来，让读者既可以了解最新版本的 Java 平台的新特性，又可以对 Java 平台的底层细节有更加深入的理解。

读者对象及如何阅读本书

本书面向的主要读者是具备一定 Java 基础的开发人员和在校学生。本书中不涉及 Java 的基本语法，因此不适合 Java 初学者阅读。如果只对 Java SE 7 中的新特性感兴趣，可以阅读第 1 章到第 6 章；如果对 Java 中的特定主题感兴趣，可以根据目录有选择地阅读。另外，第 1 章到第 6 章虽然以 Java SE 7 的新特性介绍为主，但是其中也穿插了对相关内容的深入探讨。

本书可分为三大部分：

第一部分为 Java SE 7 新特性介绍，从第 1 章到第 6 章。这部分详细地介绍了 Java SE 7 中新增的重要特性。在对新特性的介绍中，也包含了对 Java 平台相关内容的详细介绍。

第二部分为 Java SE 7 的深入探讨，从第 7 章到第 13 章。这部分着重讲解了 Java 平台上的底层实现，并对一些重要的特性进行了深入探讨。这个部分所涉及的内容包括 Java 虚拟机、Java 源代码和字节代码操作、Java 类加载器、对象生命周期、多线程与并发编程实践、Java 泛型和 Java 安全。

第三部分为 Java SE 8 的内容展望，即第 14 章。这部分简要介绍了 Java SE 8 中将

要增加的新特性。

本书还通过两个附录对 OpenJDK（附录 A）和 Java 语言的历史（附录 B）进行了简要的介绍。

勘误和支持

由于作者的水平有限，编写时间仓促，书中难免会出现一些错误或者不准确的地方，恳请读者批评指正。如果您有更多的宝贵意见，欢迎发送邮件至邮箱 alexcheng1982@gmail.com，也可以通过微博（<http://weibo.com/alexcheng1982>）与我取得联系。期待能够得到您的真挚反馈。

本书官方微博群：<http://q.weibo.com/943166>。

本书中的源代码请登录华章公司的网站（<http://www.hzbook.com>）本书页面进行下载。

致谢

感谢 InfoQ 中文站和 InfoQ 编辑张凯峰先生。这本书能够面世，得益于我在 InfoQ 中文站的“Java 深度历险”专栏上发表的文章。

感谢机械工业出版社华章公司的编辑杨福川和姜影的辛勤工作，使得这本书能够最终顺利完成。

感谢家人和朋友对我的支持与帮助！

Java 的挑战与展望

从 Java 语言出现到现在的 16 年间，在 Java 语言本身发展演化的同时，整个软件开发行业也在发生着巨大的变化。新的软件开发思想和程序设计语言层出不穷。虽然 Java 语言一直是最流行的程序设计语言之一，但它也面临着来自其他编程语言的冲击。这其中主要是互联网应用发展所带来的动态语言的影响。

Java 是静态强类型语言。这种特性使 Java 编译器在编译时就可以发现非常多的类型错误，而不会让这些错误在运行时才暴露出来。对于构建一个稳定而安全的应用来说，这是一个很大的优势，但是这种静态的类型检查也限制了开发人员编写代码时的创造性和灵活性。

Web 2.0 概念的出现和互联网应用的发展，为新语言的流行创造了契机。Ruby 语言凭借着杀手级应用 Ruby on Rails 一举蹿红，而 Google 的 Web 应用开发平台 Google App Engine 最初也只支持 Python 一种语言，甚至流行的 JavaScript 语言也借助于 node.js 和 Aptana Jaxer 等平台在服务器端开发中占据了一席之地。这些语言的共同特征是动态类型与灵活自由的语法。开发人员一旦掌握了这些语言，开发效率会非常高。在这一点上，Java 语言繁琐的语法就显得缺乏吸引力。Java 语言也受到来自同样运行在 Java 虚拟机上的其他语言的挑战。这些语言包括 Groovy、Scala、JRuby 和 Jython 等。任何语言，只要它生成的字节代码符合 Java 字节代码规范，就可以在 Java 虚拟机上运行。前面提到的这些 Java 虚拟机上的语言既具有简洁优雅的语法，又能充分利用已有的 Java 虚拟机资源，相对于 Java 语言本身来说，非常具有竞争力。

基于前面的这些现状，在社区中有人悲观地预言：Java 已死，COBOL 式的死亡。

COBOL 这门诞生于 20 世纪 50 年代末的编程语言，已经被诸多机构和个人论证为已经死亡的语言。实际上，COBOL 语言仍然在银行、金融和会计等商业应用领域占据着主导地位。只要这些应用存在，COBOL 语言就不会消亡。Java 语言也是如此。只要运行在 Java 平台上的应用还存在，Java 语言就能一直生存下去。事实上，现在仍然有许多公司和个人在向 Java 平台投资。这些投资既包括投入资金和人力来开发基于 Java 平台的应用，也包括投入时间来学习 Java 平台的相关技术。

当然，Java 平台也有不足之处，其中最明显的是整个 Java 平台的复杂性。最早在 JDK 1.0 发布的时候，只有几百个 Java 类，而现在的 Java 6 已经包括 Java SE、Java EE 和 Java ME 等多个版本，所包含的 Java 类多达数千个。对于普通开发者来说，完全理解和熟悉如此庞大的类库的难度非常大。在日常的开发过程中，经常可以看到开发者在重复实现某些功能，而这些功能在 Java 类库中已经存在，只是不被人知道而已。除了庞大的类库之外，Java 语言的语法本身也缺乏足够的灵活性，实现某些功能所需的代码量可能是其他语言的几倍。另外一个复杂性体现在 Web 应用开发方面。一个完整的 Java EE 应用程序要求程序员掌握和理解的概念太多，要使用的库也非常多。这点从市面上到处可见的以 Java Web 应用开发和 Struts、Spring 及 Hibernate 等框架为内容的图书上就可以看出来。虽然新出现的 Grails 和 Play 框架等都试图降低这个复杂度，但是这些新的框架的流行仍然需要足够长的时间。

对于 Java 语言的未来，我们有理由相信 Java 平台会一直发展下去。其中很重要的依据是 Java 平台的开放性。依托 JCP 和 OpenJDK 项目，Java 平台不仅在语言规范这个层次上有健康的开放管理流程，也有与之对应的参考实现。Java 语言有着人数众多的开发者社区，每年有非常多的开发者学习和使用 Java。大量的开发者使用 Java 语言开发各种不同类型的应用。在社区中可以看到很多提供不同功能的类库和框架。Java 虚拟机已经被安装到数以十亿计的不同类型的设备上，包括服务器、个人计算机、移动设备和智能卡等。依托庞大的社区和数量众多的运行平台，Java 语言的发展前景是非常乐观的。

对于 Java 平台来说，未来的发展将侧重于以下几个重要的方面。**第一个方面是提高开发人员的生产效率。**由于 Java 语言的静态强类型特性，使用 Java 语言编写的程序代码一般比较繁琐，包含了过多不必要的语法元素，这在一定程度上降低了开发人员的生产效率。大量的时间被浪费在语言本身上，而不是真正需要的业务逻辑上。从另外一个角度来说，Java 语言的这种严谨性，对于复杂应用的团队开发是大有好处的，有利于构建健壮的应用。Java 语言需要在这两者之间达到一个平衡。Java 语言的一个发展趋势是在可能的范围内降低语言本身的语法复杂度。从 J2SE 5.0 中增强的 for 循环，到 Java

SE 7 中的 try-with-resources 语句和 `<>` 操作符，再到 Java SE 8 中引入的 lambda 表达式，Java 正在不断地简化自身的语法。

第二个方面是提高性能。Java 平台的性能一直为开发人员所诟病，这主要是因为 Java 虚拟机这个中间层次的存在。随着硬件技术的发展，越来越多的硬件平台采用了多核 CPU 和多 CPU 的架构。应用程序应该充分利用这些资源来提高程序的运行性能。Java 平台需要帮助开发人员更好地实现这个目标。Java SE 7 中的 fork/join 框架是一个高效的任务执行框架。Java SE 8 对集合类框架和相关 API 做了增强，以支持对批量数据进行自动的并行处理。

第三个方面是模块化。一直以来，Java 平台所包含的各种功能不同的类库是一个统一的整体。在一个程序的运行过程中，很多类库其实是不需要的。比如对于一个服务器端运行的程序来说，Swing 用户界面组件库通常是不需要的。模块化的含义是把 Java 平台提供的类库划分成不同的相互依赖的模块，程序可以根据需要选择运行时所依赖的模块，只有被选择的模块才会在运行时被加载。模块化的实现不仅可以应用到 Java 平台本身，也可以应用到 Java 应用程序的开发中，OpenJDK 中的 Jigsaw 项目提供了这种模块化的支持。

目 录

前 言

Java 的挑战与展望

第 1 章 Java 7 语法新特性 / 1

- 1.1 Coin 项目介绍 / 1
- 1.2 在 switch 语句中使用字符串 / 2
 - 1.2.1 基本用法 / 2
 - 1.2.2 实现原理 / 3
 - 1.2.3 枚举类型 / 5
- 1.3 数值字面量的改进 / 5
 - 1.3.1 二进制整数字面量 / 6
 - 1.3.2 在数值字面量中使用下划线 / 6
- 1.4 优化的异常处理 / 7
 - 1.4.1 异常的基础知识 / 7
 - 1.4.2 创建自己的异常 / 8
 - 1.4.3 处理异常 / 12
 - 1.4.4 Java 7 的异常处理新特性 / 14
- 1.5 try-with-resources 语句 / 17
- 1.6 优化变长参数的方法调用 / 19
- 1.7 小结 / 21

第2章 Java语言的动态性 / 22

- 2.1 脚本语言支持 API / 22
 - 2.1.1 脚本引擎 / 23
 - 2.1.2 语言绑定 / 24
 - 2.1.3 脚本执行上下文 / 25
 - 2.1.4 脚本的编译 / 27
 - 2.1.5 方法调用 / 28
 - 2.1.6 使用案例 / 29
- 2.2 反射 API / 31
 - 2.2.1 获取构造方法 / 32
 - 2.2.2 获取域 / 34
 - 2.2.3 获取方法 / 34
 - 2.2.4 操作数组 / 35
 - 2.2.5 访问权限与异常处理 / 36
- 2.3 动态代理 / 36
 - 2.3.1 基本使用方式 / 36
 - 2.3.2 使用案例 / 40
- 2.4 动态语言支持 / 42
 - 2.4.1 Java语言与Java虚拟机 / 43
 - 2.4.2 方法句柄 / 44
 - 2.4.3 invokedynamic指令 / 66
- 2.5 小结 / 73

第3章 Java I/O / 75

- 3.1 流 / 75
 - 3.1.1 基本输入流 / 76
 - 3.1.2 基本输出流 / 77
 - 3.1.3 输入流的复用 / 78
 - 3.1.4 过滤输入输出流 / 80
 - 3.1.5 其他输入输出流 / 81
 - 3.1.6 字符流 / 81
- 3.2 缓冲区 / 82
 - 3.2.1 基本用法 / 83
 - 3.2.2 字节缓冲区 / 84

3.2.3 缓冲区视图 / 86
3.3 通道 / 87
3.3.1 文件通道 / 88
3.3.2 套接字通道 / 93
3.4 NIO.2 / 98
3.4.1 文件系统访问 / 98
3.4.2 zip/jar 文件系统 / 106
3.4.3 异步 I/O 通道 / 108
3.4.4 套接字通道绑定与配置 / 111
3.4.5 IP 组播通道 / 111
3.5 使用案例 / 113
3.6 小结 / 115

第 4 章 国际化与本地化 / 117

4.1 国际化概述 / 117
4.2 Unicode / 118
4.2.1 Unicode 编码格式 / 119
4.2.2 其他字符集 / 124
4.2.3 Java 与 Unicode / 124
4.3 Java 中的编码实践 / 125
4.3.1 Java NIO 中的编码器和解码器 / 126
4.3.2 乱码问题详解 / 130
4.4 区域设置 / 133
4.4.1 IETF BCP 47 / 134
4.4.2 资源包 / 135
4.4.3 日期和时间 / 143
4.4.4 数字和货币 / 144
4.4.5 消息文本 / 146
4.4.6 默认区域设置的类别 / 148
4.4.7 字符串比较 / 148
4.5 国际化与本地化基本实践 / 149
4.6 小结 / 152

第 5 章 图形用户界面 / 153

5.1 Java 图形用户界面概述 / 153

5.2 AWT / 156
5.2.1 重要组件类 / 156
5.2.2 任意形状的窗口 / 157
5.2.3 半透明窗口 / 158
5.2.4 组件混合 / 159
5.3 Swing / 159
5.3.1 重要组件类 / 159
5.3.2 JLayer 组件和 LayerUI 类 / 161
5.4 事件处理与线程安全性 / 163
5.4.1 事件处理 / 163
5.4.2 事件分发线程 / 165
5.4.3 SwingWorker 类 / 167
5.4.4 SecondaryLoop 接口 / 169
5.5 界面绘制 / 170
5.5.1 AWT 中的界面绘制 / 170
5.5.2 Swing 中的绘制 / 171
5.6 可插拔式外观样式 / 172
5.7 JavaFX / 175
5.7.1 场景图 / 175
5.7.2 变换 / 177
5.7.3 动画效果 / 177
5.7.4 FXML / 179
5.7.5 CSS 外观描述 / 181
5.7.6 Web 引擎与网页显示 / 182
5.8 使用案例 / 183
5.9 小结 / 185

第 6 章 Java 7 其他重要更新 / 186

6.1 关系数据库访问 / 186
6.1.1 使用 try-with-resources 语句 / 186
6.1.2 数据库查询的默认模式 / 187
6.1.3 数据库连接超时时间与终止 / 188
6.1.4 语句自动关闭 / 189
6.1.5 RowSet 实现提供者 / 190
6.2 java.lang 包的更新 / 191

6.2.1	基本类型的包装类 / 191
6.2.2	进程使用 / 192
6.2.3	Thread 类的更新 / 194
6.3	Java 实用工具类 / 195
6.3.1	对象操作 / 195
6.3.2	正则表达式 / 197
6.3.3	压缩文件处理 / 200
6.4	JavaBeans 组件 / 201
6.4.1	获取组件信息 / 201
6.4.2	执行语句和表达式 / 202
6.4.3	持久化 / 202
6.5	小结 / 203

第 7 章 Java 虚拟机 / 205

7.1	虚拟机基本概念 / 205
7.2	内存管理 / 206
7.3	引用类型 / 208
7.3.1	强引用 / 209
7.3.2	引用类型基本概念 / 211
7.3.3	软引用 / 213
7.3.4	弱引用 / 215
7.3.5	幽灵引用 / 217
7.3.6	引用队列 / 220
7.4	Java 本地接口 / 221
7.4.1	JNI 基本用法 / 221
7.4.2	Java 程序中集成 C/C++ 代码 / 225
7.4.3	在 C/C++ 程序中启动 Java 虚拟机 / 227
7.5	HotSpot 虚拟机 / 228
7.5.1	字节代码执行 / 229
7.5.2	垃圾回收 / 229
7.5.3	启动参数 / 235
7.5.4	分析工具 / 236
7.5.5	Java 虚拟机工具接口 / 241
7.6	小结 / 244

第8章 Java 源代码和字节代码操作 / 245

- 8.1 Java 字节代码格式 / 245
 - 8.1.1 基本格式 / 246
 - 8.1.2 常量池的结构 / 248
 - 8.1.3 属性 / 249
- 8.2 动态编译 Java 源代码 / 249
 - 8.2.1 使用 javac 工具 / 250
 - 8.2.2 Java 编译器 API / 251
 - 8.2.3 使用 Eclipse JDT 编译器 / 254
- 8.3 字节代码增强 / 257
 - 8.3.1 使用 ASM / 258
 - 8.3.2 增强代理 / 267
- 8.4 注解 / 271
 - 8.4.1 注解类型 / 271
 - 8.4.2 创建注解类型 / 273
 - 8.4.3 使用注解类型 / 274
 - 8.4.4 处理注解 / 275
- 8.5 使用案例 / 284
- 8.6 小结 / 286

第9章 Java 类加载器 / 287

- 9.1 类加载器概述 / 287
- 9.2 类加载器的层次结构与代理模式 / 288
- 9.3 创建类加载器 / 290
- 9.4 类加载器的隔离作用 / 294
- 9.5 线程上下文类加载器 / 296
- 9.6 Class.forName 方法 / 298
- 9.7 加载资源 / 299
- 9.8 Web 应用中的类加载器 / 301
- 9.9 OSGi 中的类加载器 / 303
 - 9.9.1 OSGi 基本的类加载器机制 / 303
 - 9.9.2 Equinox 框架的类加载实现机制 / 303
 - 9.9.3 Equinox 框架嵌入到 Web 容器中 / 306
- 9.10 小结 / 308

第 10 章 对象生命周期 / 309

- 10.1 Java 类的链接 / 309
- 10.2 Java 类的初始化 / 311
- 10.3 对象的创建与初始化 / 312
- 10.4 对象终止 / 314
- 10.5 对象复制 / 318
- 10.6 对象序列化 / 322
 - 10.6.1 默认的对象序列化 / 324
 - 10.6.2 自定义对象序列化 / 326
 - 10.6.3 对象替换 / 329
 - 10.6.4 版本更新 / 330
 - 10.6.5 安全性 / 331
 - 10.6.6 使用 Externalizable 接口 / 332
- 10.7 小结 / 334

第 11 章 多线程与并发编程实践 / 335

- 11.1 多线程 / 335
 - 11.1.1 可见性 / 336
 - 11.1.2 Java 内存模型 / 339
 - 11.1.3 volatile 关键词 / 340
 - 11.1.4 final 关键词 / 341
 - 11.1.5 原子操作 / 342
- 11.2 基本线程同步方式 / 343
 - 11.2.1 synchronized 关键词 / 343
 - 11.2.2 Object 类的 wait、notify 和 notifyAll 方法 / 344
- 11.3 使用 Thread 类 / 346
 - 11.3.1 线程状态 / 346
 - 11.3.2 线程中断 / 347
 - 11.3.3 线程等待、睡眠和让步 / 348
- 11.4 非阻塞方式 / 349
- 11.5 高级实用工具 / 352
 - 11.5.1 高级同步机制 / 352
 - 11.5.2 底层同步器 / 355
 - 11.5.3 高级同步对象 / 357

11.5.4	数据结构 / 363
11.5.5	任务执行 / 365
11.6	Java SE 7 新特性 / 368
11.6.1	轻量级任务执行框架 fork/join / 368
11.6.2	多阶段线程同步工具 / 370
11.7	ThreadLocal 类 / 373
11.8	小结 / 374

第 12 章 Java 泛型 / 375

12.1	泛型基本概念 / 375
12.2	类型擦除 / 378
12.3	上界和下界 / 382
12.4	通配符 / 384
12.5	泛型与数组 / 385
12.6	类型系统 / 388
12.7	覆盖与重载 / 391
12.7.1	覆盖对方法类型签名的要求 / 391
12.7.2	覆盖对返回值类型的要求 / 395
12.7.3	覆盖对异常声明的要求 / 396
12.7.4	重载 / 396
12.8	类型推断和 <操作符 / 397
12.9	泛型与反射 API / 400
12.10	使用案例 / 402
12.11	小结 / 403

第 13 章 Java 安全 / 405

13.1	Java 安全概述 / 405
13.2	用户认证 / 406
13.2.1	主体、身份标识与凭证 / 406
13.2.2	登录 / 407
13.3	权限控制 / 415
13.3.1	权限、策略与保护域 / 416
13.3.2	访问控制权限 / 418
13.3.3	特权动作 / 420
13.3.4	访问控制上下文 / 421

第 1 章 Java 7 语法规新特性

前面介绍 Java 所面临的挑战时就提到了 Java 语言的语法过于复杂的问题。与其他动态语言相比，利用 Java 语言所编写出来的代码不够简洁和直接。Java 语言一直在不断改进自身的语法，以满足开发人员的需求。最大的改动发生在 J2SE 5.0 版本中。泛型、增强的 for 循环、基本类型的自动装箱和拆箱机制、枚举类型、参数长度可变的方法、静态引入（import static）和注解等都是在这个版本中添加的。随后的 Java SE 6 并没有增加新的语法特性，而 Java SE 7 又增加了一些语法新特性。本章将会着重介绍这些新特性。

OpenJDK 中的 Coin 项目（Project Coin）的目的就是为了收集对 Java 语言的语法进行增强的建议。最终有 6 个语法新特性被加入到了 Java 7 中。这些语法新特性涉及 switch 语句、整数字面量、异常处理、泛型、资源处理和参数长度可变方法的调用等。

下面将对新特性进行具体的介绍。每节是独立的，读者可以有选择地阅读自己感兴趣的特性的相关章节。需要注意的是，Java 7 中与泛型相关的语法新特性将在专门介绍泛型的第 12 章中介绍。

1.1 Coin 项目介绍

在介绍具体的新特性之前，有必要介绍一下 Coin 项目。OpenJDK 中的 Coin 项目的目的是维护对 Java 语言所做的语法增强。在 Coin 项目开始之初，曾经广泛地向社区征求提议。在短短的一个月时间内就收到了近 70 条提议。最后有 9 条提议被列入考虑之中。在这 9 条提议中，有 6 条成为 Java 7 的一部分，剩下的 2 条提议会在 Java 8 中重新考虑，还有 1 条提议被移到其他项目中实现。这 6 条被接纳的提议除了本章会介绍的在 switch 语句中使用字符串、数值字面量的改进、优化的异常处理、try-with-resources 语句和优化变长参数的方法调用之外，还包括第 12 章中会介绍的简化泛型类创建的“`<>`”操作符。在 Java 8 中考虑的 2 条提议则分别是集合类字面量和为 List 和 Map 提供类似数组的按序号的访问方式。

和其他对 Java 平台所做的修改一样，Coin 项目所建议的修改也需要通过 JCP 来完成。这些改动以 JSR 334（Small Enhancements to the Java™ Programming Language）的形式提交到 JCP。

1.2 在 switch 语句中使用字符串

对于 switch 语句，开发人员应该都不陌生。大部分编程语言中都有类似的语法结构，用来根据某个表达式的值选择要执行的语句块。对于 switch 语句中的条件表达式类型，不同编程语言所提供的支持是不一样的。对于 Java 语言来说，在 Java 7 之前，switch 语句中的条件表达式的类型只能是与整数类型兼容的类型，包括基本类型 char、byte、short 和 int，与这些基本类型对应的封装类 Character、Byte、Short 和 Integer，还有枚举类型。这样的限制降低了语言的灵活性，使开发人员在需要根据其他类型的表达式来进行条件选择时，不得不增加额外的代码来绕过这个限制。为此，Java 7 放宽了这个限制，额外增加了一种可以在 switch 语句中使用的表达式类型，那就是很常见的字符串，即 String 类型。

1.2.1 基本用法

在基于 Java 7 的代码中使用这个新特性非常简单，因为这个新特性并没有改变 switch 的语法含义，只是多了一种开发人员可以选择的条件判断的数据类型。但是这个简单的新特性却带来了重大的影响，因为根据字符串进行条件判断在开发中是很常见的。

考虑这样一个应用情景，在程序中需要根据用户的性别来生成合适的称谓，比如男性就使用“×××先生”，女性就使用“×××女士”。判断条件的类型可以是字符串，如“男”表示男性，“女”表示女性。不过这在 Java 7 之前的 switch 语句中是行不通的，之前只能添加额外的代码先将字符串转换成整数类型。而在 Java 7 中就可以根据字符串进行条件判断，如下面的代码清单 1-1 所示。

代码清单 1-1 在 switch 语句中使用字符串的示例

```
public class Title {
    public String generate(String name, String gender) {
        String title = "";
        switch (gender) {
            case "男":
                title = name + " 先生";
                break;
            case "女":
                title = name + " 女士";
                break;
            default:
                title = name;
        }
        return title;
    }
}
```

在上面的代码中，Title 类的 generate 方法中的 switch 语句以传入的字符串参数 gender 作为判断条件，在对应的 case 子句中使用的是字符串常量。

注意 在 switch 语句中，表达式的值不能是 null，否则会在运行时抛出 NullPointerException。在 case 子句中也不能使用 null，否则会出现编译错误。

根据 switch 语句的语法要求，其 case 子句的值是不能重复的。这个要求对字符串类型的条件表达式同样适用。不过对于字符串来说，这种重复值的检查还有一个特殊之处，那就是 Java 代码中的字符串可以包含 Unicode 转义字符。重复值的检查是在 Java 编译器对 Java 源代码进行相关的词法转换之后才进行的。这个词法转换过程中包括了对 Unicode 转义字符的处理。也就是说，有些 case 子句的值虽然在源代码中看起来是不同的，但是经词法转换后是一样的，这就会造成编译错误。代码清单 1-2 给出了一个例子。

代码清单 1-2 switch 语句的 case 子句包含重复值的示例

```
public class TitleDuplicate {
    public String generate(String name, String gender) {
        String title = "";
        switch (gender) {
            case "男" :
                break;
            case "\u7537":
                break;
        }
        return title;
    }
}
```

在上面的代码中，类 TitleDuplicate 是无法通过编译的。这是因为其中的 switch 语句中的两个 case 子句所使用的值“男”和“\u7537”在经过词法转换之后变成一样的。“\u7537”是“男”的 Unicode 转义字符形式。

1.2.2 实现原理

在讨论了 switch 语句中字符串表达式的用法之后，下面来看看这个新特性是怎么实现的。实际上，这个新特性是在编译器这个层次上实现的。而在 Java 虚拟机和字节代码这个层次上，还是只支持在 switch 语句中使用与整数类型兼容的类型。这么做的目的是为了减少这个特性所影响的范围，以降低实现的代价。在编译器层次实现的含义是，虽然开发人员在 Java 源代码的 switch 语句中使用了字符串类型，但是在编译的过程中，编译器会根据源代码的含义来进行转换，将字符串类型转换成与整数类型兼容的格式。不同的 Java 编译器可能采用不同的方式来完成这个转换，并采用不同的优化策略。举

例来说，如果 switch 语句中只包含一个 case 子句，那么可以简单地将其转换成一个 if 语句。如果 switch 语句中包含一个 case 子句和一个 default 子句，那么可以将其转换成 if-else 语句。而对于最复杂的情况，即 switch 语句中包含多个 case 子句的情况，也可以转换成 Java 7 之前的 switch 语句，只不过使用字符串的哈希值作为 switch 语句的表达式的值。

为了探究 OpenJDK 中的 Java 编译器使用的是什么样的转换方式，需要一个名为 JAD 的工具。这个工具可以把 Java 的类文件反编译成 Java 源代码。在对编译生成 Title 类的 class 文件使用了 JAD 之后，所得到的内容如代码清单 1-3 所示。

代码清单 1-3 包含 switch 语句的 Java 类文件反编译之后的结果

```

public class Title
{
    public String generate(String name, String gender)
    {
        String title = "";
        String s = gender;
        byte byte0 = -1;
        switch(s.hashCode())
        {
            case 30007:
                if(s.equals("\u7537"))
                    byte0 = 0;
                break;
            case 22899:
                if(s.equals("\u5973"))
                    byte0 = 1;
                break;
        }
        switch(byte0)
        {
            case 0: // '\0'
                title = (new StringBuilder()).append(name).append(" \u5148\u751F").
                    toString();
                break;
            case 1: // '\001'
                title = (new StringBuilder()).append(name).append(" \u5973\u58EB").
                    toString();
                break;
            default:
                title = name;
                break;
        }
        return title;
    }
}

```

从上面的代码中可以看出，原来用在 switch 语句中的字符串被替换成了对应的哈希

值，而 case 子句的值也被换成了原来字符串常量的哈希值。经过这样的转换，Java 虚拟机所看到的仍然是与整数类型兼容的类型。在这里值得注意的是，在 case 子句对应的语句块中仍然需要使用 String 的 equals 方法来进行字符串比较。这是因为哈希函数在映射的时候可能存在冲突，多个字符串的哈希值可能是一样的。进行字符串比较是为了保证转换之后的代码逻辑与之前完全一样。

1.2.3 枚举类型

以笔者的个人观点来看，Java 7 引入的这个新特性虽然为开发人员提供了方便，但是比较容易被误用，造成代码的可维护性问题。提到这一点就必须要说一下 Java SE 5.0 中引入的枚举类型。switch 语句的一个典型的应用就是在多个枚举值之间进行选择。比如代码清单 1-1 中的性别枚举值“男”和“女”，或者是一个星期中的每一天。在 Java SE 5.0 之前，一般的做法是使用一个整数来为这些枚举值编号，比如 0 表示“男”，1 表示“女”。在 switch 语句中使用这个整数编码来进行判断。这种做法的弊端有很多，比如不是类型安全的、没有名称空间、可维护性差和不够直观等。Joshua Bloch 最早在他的《Effective Java》一书中提出了一种类型安全的枚举类型的实现方式。这种方式在 J2SE 5.0 中被引入到标准库，就是现在的 enum 关键字。

Java 语言中的枚举类型的最大优势在于它是一个完整的 Java 类，除了定义其中包含的枚举值之外，还可以包含任意的方法和域，以及实现任意的接口。这使得枚举类型可以很好地与其他 Java 类进行交互。在涉及多个枚举值的情况下，都应该优先使用枚举类型。

在 Java 7 之前，也就是 switch 语句还不支持使用字符串表达式类型时，如果要枚举的值本身都是字符串，使用枚举类型是唯一的选择。而在 Java 7 中，由于 switch 语句增加了对字符串条件表达式的支持，一些开发人员会选择放弃枚举类型而直接在 case 子句中用字符串常量来列出各个枚举值。这种方式虽然简单和直接，但是会带来维护上的麻烦，尤其是这样的 switch 语句在程序的多个地方出现的时候。在程序中多次出现字符串常量总是一个不好的现象，而使用枚举类型就可以避免这种情况。

对此，笔者的建议是，如果代码中有多个地方使用 switch 语句来枚举字符串，就考虑用枚举类型进行替换。

1.3 数值字面量的改进

在编程语言中，字面量 (literal) 指的是在源代码中直接表示的一个固定的值。绝大部分编程语言都支持在源代码中使用基本类型字面量，包括整数、浮点数、字符串和布尔值等。少数编程语言支持复杂类型的字面量，如数组和对象等。Java 语言只支持基本类型的字面量。Java 7 中对数值类型字面量进行了增强，包括对整数和浮点数字面量的增强。

1.3.1 二进制整数字面量

在 Java 源代码中使用整数字面量的时候，可以指定所使用的进制。在 Java 7 之前，所支持的进制包括十进制、八进制和十六进制。十进制是默认使用的进制。八进制是用在整数字面量之前添加“0”来表示的，而十六进制则是用在整数字面量之前添加“0x”或“0X”来表示的。Java 7 中增加了一种可以在字面量中使用的进制，即二进制。二进制整数字面量是通过在数字前面添加“0b”或“0B”来表示的，如代码清单 1-4 所示。

代码清单 1-4 二进制整数字面量的示例

```
import static java.lang.System.out;
public class BinaryIntegralLiteral {
    public void display() {
        out.println(0b001001); // 输出 9
        out.println(0B0001110); // 输出 14
    }
}
```

这种新的二进制字面量的表示方式使得在源代码中使用二进制数据变得更加简单，不再需要先手动将数据转换成对应的八 / 十 / 十六进制的数值。

1.3.2 在数值字面量中使用下划线

如果 Java 源代码中有一个很长的数值字面量，开发人员在阅读这段代码时需要很费力地去分辨数字的位数，以知道其所代表的数值大小。在现实生活中，当遇到很长的数字的时候，我们采取的是分段分隔的方式。比如数字 500000，我们通常会写成 500,000，即每三位数字用逗号分隔。利用这种方式就可以很快知道数值的大小。这种做法的理念被加入到了 Java 7 中，不过用的不是逗号，而是下划线“_”。

在 Java 7 中，数值字面量，不管是整数还是浮点数，都允许在数字之间插入任意多个下划线。这些下划线不会对字面量的数值产生影响，其目的主要是方便阅读。一些典型的用法包括每三位数字插入一个下划线来分隔，以及多行数值的对齐，如代码清单 1-5 所示。

代码清单 1-5 在数值字面量中使用下划线的示例

```
import static java.lang.System.out;
public class Underscore {
    public void display() {
        out.println(1_500_000); // 输出 1500000
        double value1 = 5_6.3_4;
        int value2 = 89_3__1;
        out.println(value1); // 输出 56.34
        out.println(value2); // 输出 8931
    }
}
```

虽然下划线在数值字面量中的应用非常灵活，但有些情况是不允许出现的。最基本的原则是下划线只能出现在数字中间，也就是说前后都必须是数字。所以“_100”、“120_”、“0b_101”、“0x_da0”这样的使用方式都是非法的，无法通过编译。这样限制的动机在于降低实现的复杂度。有了这个限制之后，Java 编译器只需要在扫描源代码的时候，将所发现的数字中间的下划线直接删除就可以了。这样就和没有使用下划线的形式是相同的。如果不添加这个限制，那么编译器需要进行语法分析才能做出判断。比如“_100”可能是一个整数字面量 100，也可能一个变量名称。这就要求编译器的实现做出更加复杂的改动。

1.4 优化的异常处理

这一节将要介绍的是 Java 语言中的异常处理。相信大部分开发人员对于 Java 语言中使用 try-catch-finally 语句块进行异常处理的基本方式都有所了解。异常处理以一种简洁的方式表示了程序中可能出现的错误，以及应对这些错误的处理方式。适当地使用异常处理技术，可以提高代码的可靠性、可维护性和可读性。但是如果使用不当，就会产生相反的效果。比如虽然一个方法声明了会抛出某个异常，但是使用这个方法的代码在异常发生的时候，却只能捕获完异常之后就直接忽略它，无法做其他的处理。而为了能够通过编译，又不得不加上 catch 语句。这势必会造成冗余无用的代码，同时给出不适当的异常设计的一个信号。类似这种错误使用异常的例子在日常开发中还有很多。在 Java 标准库中同样也有设计失败的异常处理的例子。

Java 7 对异常处理做了两个重要的改动：一个是支持在一个 catch 子句中同时捕获多个异常，另外一个是在捕获并重新抛出异常时的异常类型更加精确。本节的内容并不限于介绍 Java 7 中关于异常处理的这两个新特性，还会围绕整个异常处理进行展开。这样安排的目的是帮助读者深入理解与 Java 的异常处理相关的内容。

1.4.1 异常的基础知识

Java 语言中基本的异常处理是围绕 try-catch-finally、throws 和 throw 这几个关键词展开的。具体来说，throws 用来声明一个方法可能抛出的异常，对方法体中可能抛出的异常都要进行声明；throw 用来在遇到错误的时候抛出一个具体的异常；try-catch-finally 则用来捕获异常并进行处理。Java 中的异常有受检异常和非受检异常两类。

1. 受检异常和非受检异常

在异常处理的时候，都会接触到受检异常（checked exception）和非受检异常（unchecked exception）这两种异常类型。非受检异常指的是 `java.lang.RuntimeException` 和 `java.lang.Error` 类及其子类，所有其他的异常类都称为受检异常。两种类型的异常在作用上并没有差别，唯一的差别就在于使用受检异常时的合法性要在编译时刻由编译器

来检查。正因为如此，受检异常在使用的时候需要比非受检异常更多的代码来避免编译错误。

一直以来，关于在程序中到底是该使用受检异常还是非受检异常，开发者之间一直存在着争议，毕竟两类异常都各有优缺点。受检异常的特点在于它强制要求开发人员在代码中进行显式的声明和捕获，否则就会产生编译错误。这种限制从好的方面来说，可以防止开发人员意外地忽略某些出错的情况，因为编译器不允许出现未被处理的受检异常；从不好的方面来说，受检异常对程序中的设计提出了更高的要求。不恰当地使用受检异常，会使代码中充斥着大量没有实际作用、只是为了通过编译而添加的代码。而非受检异常的特点是，如果不捕获异常，不会产生编译错误，异常会在运行时刻才被抛出。非受检异常的好处是可以去掉一些不需要的异常处理代码，而不好之处是开发人员可能忽略某些应该处理的异常。一个典型的例子是把字符串转换成数字时会发生 `java.lang.NumberFormatException` 异常，忽略该异常可能导致一个错误的输入就造成整个程序退出。

目前的主流意见是，最好优先使用非受检异常。

2. 异常声明是 API 的一部分

这一条提示主要是针对受检异常的。在一个公开方法的声明中使用 `throws` 关键词来声明其可能抛出的异常的时候，这些异常就成为这个公开方法的一部分，属于开放 API。在维护这个公开 API 的时候，这些异常有可能会对 API 的演化造成阻碍，使得编写代码时不得不考虑向后兼容性的问题。

如果公开方法声明了会抛出一个受检异常，那么这个 API 的使用者肯定已经使用了 `try-catch-finally` 来处理这个异常。如果在后面的版本更新中，发现该 API 抛出这个异常是不合适的，也不能直接把这个异常的声明删除。因为这样会造成之前的 API 使用者的代码无法通过编译。

因此，对于 API 的设计者来说，谨慎考虑每个公开方法所声明的异常是很有必要的。因为一旦加了异常声明，在很长的一段时间内都无法甩掉它。这也是为什么推荐使用非受检异常的一个重要原因，非受检异常不需要声明就可以直接抛出。但是对于一个方法会抛出的非受检异常，也需要在文档中进行说明。

1.4.2 创建自己的异常

和程序中的其他部分一样，异常部分也需要经过仔细的考虑和设计。开发人员一般会花费大量的精力对程序的主要功能部分进行设计，而忽略对于异常的设计。这会对程序的整体架构造成影响。在对异常部分进行设计的时候，考虑下面几个建议。

1. 精心设计异常的层次结构

一般来说，一个程序中应该要有自己的异常类的层次结构。如果只打算使用非受检异常，至少需要一个继承自 `RuntimeException` 的异常类。如果还需要使用受检异常，还要有另外一个继承自 `Exception` 的异常类。如果程序中可能出现的异常情况比较多，应

该在不同的抽象层次上定义相关的异常，并形成一个完整的层次结构。这个异常的层次结构与程序本身的类层次结构是相对应的。不同抽象层次上的代码应该只声明抛出同一层次上的相关异常。

比如一个典型的 Web 应用按照自顶向下的顺序一般分成展现层、服务层和数据访问层。与之对应的异常也应该按照这个层次结构来进行划分。数据访问层的代码应该只声明抛出与访问数据相关的异常，如数据库连接和操作相关的异常。这么做的好处是工作于某个抽象层次上的开发人员不需要去了解其他层次上的细节。比如服务层开发人员会调用数据访问层的代码，他只需要关心数据访问可能出现异常即可，而并不需要去关心这是一个数据库访问异常，还是一个文件系统访问异常。这种抽象层次的划分对系统的演化是比较重要的。假如系统以后不再使用数据库作为数据访问的实现，服务层的异常处理逻辑也不会受到影响。

一般来说，对于程序中可能出现的各种错误，都需要声明一个异常类与之对应。有些开发人员会选择一个大而全的异常类来表示各种不同类型错误，利用这个异常的消息来区分不同的错误。比如声明一个异常类 `BaseException`，不管是数据访问错误还是用户输入的数据格式不对，都会抛出同一个异常，只是使用的消息内容不同。当采用这种异常设计方式的时候，异常的处理者只能根据异常消息字符串的内容来判断具体的错误类型。如果异常的处理者只是简单地进行日志记录或重新抛出此异常，这种方式并没有太大的问题。如果异常的处理者需要解析异常的消息格式来判断具体类型，那么这种方式就是不可取的，应该换成不同的异常类。

采用这种异常层次结构会遇到的一个常见的异常处理模式是包装异常。包装异常的目的在于使异常只出现在其所对应的抽象层次上。当一个异常抛出的时候，如果没有被捕获到，就会一直沿着调用栈往上传递，直到被上层方法捕获或是最终由 Java 虚拟机来处理。这种传递方式会使这个异常跨越多个抽象层次的边界，使得上层代码看到不需要关注的底层异常。为此，在一个异常要跨越抽象层次边界的时候，需要进行包装。包装之后的异常才是上层代码需要关注的。

对一个异常进行包装是一件非常简单的事情。从 JDK 1.4 开始，所有异常的基类 `java.lang.Throwable` 就支持在构造方法中传入另外一个异常作为参数。而这个参数所表示的异常被包装在新的异常中，可以通过 `getCause` 方法来获取。代码清单 1-6 给出了一个异常包装的示例，即把底层的 `IOException` 包装成更为抽象的 `DataAccessException`。使用 `DataAccessGateway` 类的上层代码只需要知道 `DataAccessException` 即可，并不需要知道 `IOException` 的存在。

代码清单 1-6 使用异常包装技术的示例

```
public class DataAccessGateway {
    public void load() throws DataAccessException {
        try {
            FileInputStream input = new FileInputStream("data.txt");
        }
    }
}
```

```

        }
        catch (IOException e) {
            throw new DataAccessException(e);
        }
    }
}

```

在使用异常包装的时候，一个典型的做法就是为每个层次定义一个基本的异常类。这个层次的所有公开方法在声明异常的时候都使用这个异常类。所有这个层次中出现的底层异常都被包装成这个异常。

2. 异常类中包含足够的信息

异常存在的一个很重要的意义在于，当错误发生的时候，调用者可以对错误进行处理，从产生的错误中恢复。为了方便调用者处理这些异常，每个异常中都需要包含尽量丰富的信息。异常不应该只说明某个错误发生了，还应该给出相关的信息。异常类是完整的 Java 类，因此在其中添加所需的域和方法是一件很简单的事情。

考虑下面一个场景，当用户进行支付的时候，如果他的当前余额不足以完成支付，那么在所抛出的异常信息中，可以包含当前所需的金额、余额和其中的差额等信息。这样异常处理器就可以提供给用户更加具体的出错信息以及更加明确的解决方案。

3. 异常与错误提示

对于与用户进行交互的程序来说，需要适当区分异常与展示给用户的错误提示。通常来说，异常指的是程序的内部错误。与异常相关的信息，主要是供开发人员调试时使用的。这些信息对于最终用户来说是没有意义的。一般来说，普通用户除了重新执行出错的操作之外，没有其他应对办法。因此，程序需要保证在直接与用户交互的代码层次上，捕获所有的异常，并生成相应的错误提示。比如在一个 servlet 中，要确保在产生 HTTP 响应的时候捕获全部的异常，以避免用户看到一个包含异常堆栈信息的错误页面。

有些开发人员会直接将异常自带的消息作为给用户的错误提示。这个时候需要注意异常消息的国际化问题。只需要把异常与 Java 中的 `java.util.ResourceBundle` 结合起来，就可以很容易地实现异常消息的国际化。代码清单 1-7 给出了一个支持国际化异常消息的异常类的基类 `LocalizedMessage`。

代码清单 1-7 支持国际化异常消息的异常类的基类

```

public abstract class LocalizedException extends Exception {
    private static final String DEFAULT_BASE_NAME = "com/java7book/chapter1/
        exception/java7/messages";
    private String baseName = DEFAULT_BASE_NAME;
    protected ResourceBundle resourceBundle;
    private String messageKey;
    public LocalizedException(String messageKey) {

```

```

        this.messageKey = messageKey;
        initResourceBundle();
    }
    public LocalizedException(String messageKey, String baseName) {
        this.messageKey = messageKey;
        this.baseName = baseName;
        initResourceBundle();
    }
    private void initResourceBundle() {
        resourceBundle = ResourceBundle.getBundle(baseName);
    }
    protected void setBaseName(String baseName) {
        this.baseName = baseName;
    }
    protected void setMessageKey(String key) {
        messageKey = key;
    }
    public abstract String getLocalizedMessage();
    public String getMessage() {
        return getLocalizedMessage();
    }
    protected String format(Object ... args) {
        String message = resourceBundle.getString(messageKey);
        return MessageFormat.format(message, args);
    }
}

```

在使用的时候，每个需要国际化的异常类只需要继承 LocalizedException，并实现 getLocalizedMessage 方法即可。代码清单 1-8 是之前提到的支付余额不足时抛出的异常类。在子类的构造方法中指定异常消息在消息资源文件中对应的名称。使用 format 方法可以对消息进行格式化。

代码清单 1-8 继承自支持国际化异常消息的异常类的子类

```

public class InsufficientBalanceException extends LocalizedException {
    private BigDecimal requested;
    private BigDecimal balance;
    private BigDecimal shortage;
    public InsufficientBalanceException(BigDecimal requested, BigDecimal balance) {
        super("INSUFFICIENT_BALANCE_EXCEPTION");
        this.requested = requested;
        this.balance = balance;
        this.shortage = requested.subtract(balance);
    }
    public String getLocalizedMessage() {
        return format(balance, requested, shortage);
    }
}

```

1.4.3 处理异常

处理异常的基本思路也比较简单。一般来说就两种选择：处理或是不处理。如果某个异常在当前的调用栈层次上是可以处理和应该处理的，那么就应该直接处理掉；如果不能处理，或者不适合在这个层次上处理，就可以选择不理会该异常，而让它自行往更上层的调用栈上传递。如果当前的代码位于抽象层次的边界，就需要首先捕获该异常，重新包装之后，再往上传递。

决定是否在某个方法中处理一个异常需要判断从异常中恢复的方式是否合理。比如一个方法要从文件中读取配置信息，进行文件操作时可能抛出 IOException。当出现异常的时候，如果可以采取的恢复措施是使用默认值，那么在这个方法中处理 IOException 就是合理的。而在同样的场景中，如果某些配置项没有合法的默认值，必须要手工设置一个值，那么读取文件时出现的 IOException 就不应该在这个方法中处理。

在确定了需要对异常进行处理之后，按照程序本身的逻辑来处理即可。下面将要介绍的是一个处理异常时容易忽略的问题——消失的异常。

开发人员对异常处理的 try-catch-finally 语句块都比较熟悉。如果在 try 语句块中抛出了异常，在控制权转移到调用栈上一层代码之前，finally 语句块中的语句也会执行。但是 finally 语句块在执行的过程中，也可能会抛出异常。如果 finally 语句块也抛出了异常，那么这个异常会往上传递，而之前 try 语句块中的那个异常就丢失了。代码清单 1-9 给出了一个示例，try 语句块会抛出 NumberFormatException，而在 finally 语句块中会抛出 ArithmeticException。对这个方法的使用者来说，他最终看到的只是 finally 语句块中抛出的 ArithmeticException，而 try 语句中抛出的 NumberFormatException 消失不见了。

代码清单 1-9 异常消失的示例

```
public class DisappearedException {
    public void show() throws BaseException{
        try {
            Integer.parseInt("Hello");
        }
        catch (NumberFormatException nfe) {
            throw new BaseException(nfe);
        } finally {
            try {
                int result = 2 / 0;
            } catch (ArithmeticException ae) {
                throw new BaseException(ae);
            }
        }
    }
}
```

其实这样的例子在日常开发中也是比较常见的。比如在打开一个文件进行读取的时

候，肯定需要用 try-catch 语句块来捕获其中的 IOException，并且在 finally 语句块中关闭文件输入流。在关闭输入流的时候可能会抛出异常，造成之前在读取文件时产生的异常丢失。还有一个典型的情况发生在数据库操作的时候，在 finally 语句块中关闭数据库连接。由于之前产生的异常丢失，开发人员可能无法准确定位异常的发生位置，造成错误的判断。

对这种问题的解决办法一般有两种，一种是抛出 try 语句块中产生的原始异常，忽略在 finally 语句块中产生的异常。这么做的出发点是 try 语句块中的异常才是问题的根源。另外一种是把产生的异常都记录下来。这么做好处是不会丢失任何异常。在 Java 7 之前，这种做法需要实现自己的异常类，而在 Java 7 中，已经对 Throwable 类进行了修改以支持这种情况。

第一种做法的实现方式如代码清单 1-10 所示。

代码清单 1-10 抛出 try 语句块中产生的原始异常的示例

```
public class ReadFile {
    public void read(String filename) throws BaseException {
        FileInputStream input = null;
        IOException readException = null;
        try {
            input = new FileInputStream(filename);
        } catch (IOException ex) {
            readException = ex;
        } finally {
            if (input != null) {
                try {
                    input.close();
                } catch (IOException ex) {
                    if (readException == null) {
                        readException = ex;
                    }
                }
            }
            if (readException != null) {
                throw new BaseException(readException);
            }
        }
    }
}
```

第二种做法需要利用 Java 7 中为 Throwable 类增加的 addSuppressed 方法。当一个异常被抛出的时候，可能有其他异常因为该异常而被抑制住，从而无法正常抛出。这时可以通过 addSuppressed 方法把这些被抑制的方法记录下来。被抑制的异常会出现在抛出的异常的堆栈信息中，也可以通过 getSuppressed 方法来获取这些异常。这样做的好处是不会丢失任何异常，方便开发人员进行调试。代码清单 1-11 给出了使用

addSuppressed 方法记录异常的示例。

代码清单 1-11 使用 addSuppressed 方法记录异常的示例

```

public class ReadFile {
    public void read(String filename) throws IOException {
        FileInputStream input = null;
        IOException readException = null;
        try {
            input = new FileInputStream(filename);
        } catch (IOException ex) {
            readException = ex;
        } finally {
            if (input != null) {
                try {
                    input.close();
                } catch (IOException ex) {
                    if (readException != null) {
                        readException.addSuppressed(ex);
                    }
                    else {
                        readException = ex;
                    }
                }
            }
            if (readException != null) {
                throw readException;
            }
        }
    }
}

```

这种做法的关键在于把 finally 语句中产生的异常通过 addSuppressed 方法加到 try 语句产生的异常中。

1.4.4 Java 7 的异常处理新特性

下面详细介绍 Java 7 中引入的与异常处理相关的新特性。

1. 一个 catch 子句捕获多个异常

在 Java 7 之前的异常处理语法中，一个 catch 子句只能捕获一类异常。在要处理的异常种类很多时这种限制会很麻烦。每一种异常都需要添加一个 catch 子句，而且这些 catch 子句中的处理逻辑可能都是相同的，从而会造成代码重复。虽然可以在 catch 子句中通过这些异常的基类来捕获所有的异常，比如使用 Exception 作为捕获的类型，但是这要求对这些不同的异常所做的处理是相同的。另外也可能会捕获到某些不应该被捕获的非受检异常。而在某些情况下，代码重复是不可避免的。比如某个方法可能抛出 4 种不同的异常，其中有 2 种异常使用相同的处理方式，另外 2 种异常的处理方式也相同，但是不同于前面的 2 种异常。这势必会在 catch 子句中包含重复的代码。

对于这种情况，Java 7改进了catch子句的语法，允许在其中指定多种异常，每个异常类型之间使用“|”来分隔，如代码清单1-12所示。ExceptionThrower类的manyExceptions方法会抛出ExceptionA、ExceptionB和ExceptionC三种异常，其中对ExceptionA和ExceptionB采用一种处理方式，对ExceptionC采用另外一种处理方式。

代码清单1-12 在catch子句中指定多种异常

```
public class ExceptionHandler {
    public void handle() {
        ExceptionThrower thrower = new ExceptionThrower();
        try {
            thrower.manyExceptions();
        } catch (ExceptionA | ExceptionB ab) {
        } catch (ExceptionC c) {
        }
    }
}
```

这种新的处理方式使上面提出的问题得到了很好的解决。需要注意的是，在catch子句中声明捕获的这些异常类中，不能出现重复的类型，也不允许其中的某个异常是另外一个异常的子类，否则会出现编译错误。如果在catch子句中声明了多个异常类，那么异常参数的具体类型是所有这些异常类型的最小上界。

关于一个catch子句中的异常类型不能出现其中一个是一个的子类的情况，实际上涉及捕获多个异常的内部实现方式。比如在代码清单1-13中，虽然NumberFormatException是RuntimeException的子类，但是这段代码是可以通过编译的。

代码清单1-13 catch子句中声明异常的顺序的正确示例

```
public void testSequence() {
    try {
        Integer.parseInt("Hello");
    }
    catch (NumberFormatException | RuntimeException e) {}
}
```

但是如果把catch子句中两个异常的声明位置调换一下，就会出现编译错误。代码清单1-14会产生编译错误。

代码清单1-14 catch子句中声明异常的顺序的错误示例

```
public void testSequenceError() {
    try {
        Integer.parseInt("Hello");
    }
    catch (RuntimeException | NumberFormatException e) {}
}
```

原因在于，编译器的做法其实是把捕获多个异常的 catch 子句转换成了多个 catch 子句，在每个 catch 子句中捕获一个异常。代码清单 1-14 中的 testSequenceError 方法实际上相当于代码清单 1-15。这段代码显然是不能通过编译的，因为在上一个 catch 子句中已经捕获了 RuntimeException，在下一个 catch 子句中无法再捕获其子类异常。

代码清单 1-15 代码清单 1-14 中异常捕获的等价形式

```
public void testSequenceError() {
    try {
        Integer.parseInt("Hello");
    }
    catch (RuntimeException e) {}
    catch (NumberFormatException e) {}
}
```

关于 catch 子句中异常参数的具体类型，可以参看代码清单 1-16。这里 catch 子句的异常类型包括 ExceptionASub1 和 ExceptionASub2，因此参数“e”的具体类型是 ExceptionASub1 和 ExceptionASub2 在类继承层次结构上的最小祖先类，即 ExceptionA，在 catch 子句中可以调用 ExceptionA 中的方法。因为所有的异常都是 Exception 类的后代，所以这样一个最小的上界总是会存在的。

代码清单 1-16 catch 子句中异常参数的具体类型

```
public void testCatchType() {
    try {
        throwException();
    }
    catch (ExceptionASub1 | ExceptionASub2 e) {
        e.methodInExceptionA();
    }
}
```

2. 更加精确的异常抛出

在进行异常处理的时候，如果遇到当前代码无法处理的异常，应该把异常重新抛出，交由调用栈的上层代码来处理。在重新抛出异常的时候，需要判断异常的类型。Java 7 对重新抛出异常时的异常类型做了更加精确的判断，以保证抛出的异常的确是可以被抛出的。这个改进初看起来会让人有点费解，因为从语义上来说，不能被抛出来的异常是不会被重新抛出的。但是在 Java 7 之前，Java 编译器并不能做出精确的判断，因此会存在一些隐含的不正确的情况。在 Java 7 中，如果一个 catch 子句的异常类型参数在 catch 代码块中没有被修改，而这个异常又被重新抛出，编译器会知道这个重新被抛出的异常肯定是 try 语句块中可以抛出的异常，同时也是没有被之前的 catch 子句捕获的异常。代码清单 1-17 给出了一个精确的异常抛出的例子来说明 Java 7 之前的编译器和 Java 7 编译器不一样的行为。

代码清单 1-17 精确的异常抛出的示例

```

public class PreciseThrowUse {
    public void testThrow() throws ExceptionA {
        try {
            throw new ExceptionASub2();
        }
        catch(ExceptionA e) {
            try {
                throw e;
            }
            catch (ExceptionASub1 e2) { // 编译错误
            }
        }
    }
}

```

在上面的代码中，异常类 ExceptionASub1 和 ExceptionASub2 都是 ExceptionA 的子类，而且这两者之间并没有继承关系。方法 testThrow 中首先抛出了 ExceptionASub2 异常，通过第一个 catch 子句捕获之后重新抛出。在这里，Java 编译器可以准确知道变量 e 表示的异常类型是 ExceptionASub2，接下来的第二个 catch 子句试图捕获 ExceptionASub1 类型的异常，这显然是不可能的，因此会产生编译错误。上面的代码在 Java 6 编译器上是可以通过编译的。对于 Java 6 编译器来说，第二个 try 子句中抛出的异常类型是前一个 catch 子句中声明的 ExceptionA 类型，因此在第二个 catch 子句中尝试捕获 ExceptionA 的子类型 ExceptionASub1 是合法的。

1.5 try-with-resources 语句

这一节将要介绍的是 Java 7 中引入的使用 try 语句进行资源管理的新用法。这一节的内容与上一节介绍的异常处理的关系比较密切。比如 1.4.3 节中介绍的 Throwable 中的新方法 addSuppressed 就是为 try-with-resources 语句添加的。对于资源管理，大多数开发人员都知道的一条原则是：谁申请，谁释放。这些资源涉及操作系统中的主存、磁盘文件、网络连接和数据库连接等。凡是数量有限的、需要申请和释放的实体，都应该纳入到资源管理的范围中来。

对于 C++ 程序员来说，程序的内存管理是他们的一项职责。他们需要保证每一块申请的内存都在正确的时候得到了释放。要么在构造函数中申请，在析构函数中释放；要么使用类似智能指针一样的结构来实现资源管理。Java 语言把内存管理的任务交给了 Java 虚拟机，通过自动垃圾回收机制减少了开发人员的很多工作。但是像输入输出流和数据库连接这样的资源，还是需要开发人员手动释放。

在使用资源的时候，有可能会抛出各种异常，比如读取磁盘文件和访问数据库时都

可能出现各种不同的异常。而资源管理的一个要求就是不管操作是否成功，所申请的资源都要被正确释放。1.4.3 节的代码清单 1-10 就是资源管理的经典案例，即通过 try-catch-finally 语句块的 finally 语句进行资源释放操作。这种方式虽然比较易懂，但是其中包含的冗余代码比较多。

为了简化这种典型的应用，Java 7 对 try 语句进行了增强，使它可以支持对资源进行管理，保证资源总是被正确释放。代码清单 1-18 给出了一个读取磁盘文件内容的示例。

代码清单 1-18 读取磁盘文件内容的示例

```
public class ResourceBasicUsage {
    public String readFile(String path) throws IOException {
        try (BufferedReader reader = new BufferedReader(new FileReader(path))) {
            StringBuilder builder = new StringBuilder();
            String line = null;
            while ((line = reader.readLine()) != null) {
                builder.append(line);
                builder.append(String.format("%n"));
            }
            return builder.toString();
        }
    }
}
```

上面的代码并不需要使用 finally 语句来保证打开的流被正确关闭，这是自动完成的。相对于传统的使用 finally 语句的做法，这种方式要简单得多。开发人员只需要关心使用资源的业务逻辑即可。资源的申请是在 try 子句中进行的，而资源的释放则是自动完成的。在使用 try-with-resources 语句的时候，异常可能发生在 try 语句中，也可能发生在释放资源时。如果资源初始化时或 try 语句中出现异常，而释放资源的操作正常执行，try 语句中的异常会被抛出；如果 try 语句和释放资源都出现了异常，那么最终抛出的异常是 try 语句中出现的异常，在释放资源时出现的异常会作为被抑制的异常添加进去，即通过 Throwable.addSuppressed 方法来实现。

能够被 try 语句所管理的资源需要满足一个条件，那就是其 Java 类要实现 java.lang.AutoCloseable 接口，否则会出现编译错误。当需要释放资源的时候，该接口的 close 方法会被自动调用。Java 类库中已有不少接口或类继承或实现了这个接口，使得它们可以用在 try 语句中。在这些已有的常见接口或类中，最常用的就是与 I/O 操作和数据库相关的接口。与 I/O 相关的 java.io.Closeable 继承了 AutoCloseable，而与数据库相关的 java.sql.Connection、java.sql.ResultSet 和 java.sql.Statement 也继承了该接口。如果希望自己开发的类也能利用 try 语句的自动化资源管理，只需要实现 AutoCloseable 接口即可。代码清单 1-19 给出了一个自定义资源的使用示例，在 close 方法中可以添加所需要的资源释放逻辑。

代码清单 1-19 自定义资源使用 AutoCloseable 接口的示例

```
public class CustomResource implements AutoCloseable {
    public void close() throws Exception {
        System.out.println("进行资源释放。");
    }

    public void useCustomResource() throws Exception {
        try (CustomResource resource = new CustomResource()) {
            System.out.println("使用资源。");
        }
    }
}
```

除了对单个资源进行管理之外，try-with-resources 还可以对多个资源进行管理。代码清单 1-20 给出了 try-with-resources 语句同时管理两个资源的例子，即经典的文件内容复制操作。

代码清单 1-20 使用 try-with-resources 语句管理两个资源的示例

```
public class MultipleResourcesUsage {
    public void copyFile(String fromPath, String toPath) throws IOException {
        try (InputStream input = new FileInputStream(fromPath);
             OutputStream output = new FileOutputStream(toPath)) {
            byte[] buffer = new byte[8192];
            int len = -1;
            while ((len = input.read(buffer)) != -1) {
                output.write(buffer, 0, len);
            }
        }
    }
}
```

当对多个资源进行管理的时候，在释放每个资源时都可能会产生异常。所有这些异常都会被加到资源初始化异常或 try 语句块中抛出的异常的被抑制异常列表中。

在 try-with-resource 语句中也可以使用 catch 和 finally 子句。在 catch 子句中可以捕获 try 语句块和释放资源时可能发生的各种异常。

1.6 优化变长参数的方法调用

J2SE 5.0 中引入的一个新特性就是允许在方法声明中使用可变长度的参数。一个方法的最后一个形式参数可以被指定为代表任意多个相同类型的参数。在调用的时候，这些参数是以数组的形式来传递的。在方法体中也可以按照数组的方式来引用这些参数。代码清单 1-21 给出了一个简单的示例，对多个整数进行求和。可以用类似 sum(1, 2, 3) 这样的形式来调用此方法。

代码清单 1-21 变长参数方法的示例

```
public int sum(int... args) {
    int result = 0;
    for (int value : args) {
        result += value;
    }
    return result;
}
```

可变长度的参数在实际开发中可以简化方法的调用方式。但是在 Java 7 之前，如果可变长度的参数与泛型一起使用会遇到一个麻烦，就是编译器产生的警告过多。比如代码清单 1-22 中给出的方法。

代码清单 1-22 使用泛型的变长参数方法产生编译器警告的示例

```
public static <T> T useVarargs(T... args) {
    return args.length > 0 ? args[0] : null;
}
```

如果参数传递的是不可具体化（non-reifiable）的类型，如 `List<String>` 这样的泛型类型，会产生警告信息。每一次调用该方法，都会产生警告信息。比如在 Java 7 之前的编译器上编译代码清单 1-23 中的代码，编译器会给出警告信息。如果希望禁止这个警告信息，需要使用 `@SuppressWarnings("unchecked")` 注解来声明。

代码清单 1-23 调用使用泛型的变长参数方法的示例

```
VarargsWarning.useVarargs(new ArrayList<String>());
```

这其中的原因是可变长度的方法参数的实际值是通过数组来传递的，而数组中存储的是不可具体化的泛型类对象，自身存在类型安全问题。因此编译器会给出相应的警告信息。关于泛型的内容，本书的第 12 章会详细介绍，这里就不再赘述。

这样的警告信息在使用 Java 标准类库中的 `java.util.Arrays` 类的 `asList` 和 `java.util.Collections` 类的 `addAll` 方法中也会遇到。建议开发人员每次使用方法时都抑制编译器的警告信息，并不是一个好主意。为了解决这个问题，Java 7 引入了一个新的注解 `@SafeVarargs`。如果开发人员确信某个使用了可变长度参数的方法，在与泛型类一起使用时不会出现类型安全问题，就可以用这个注解进行声明。在使用了这个注解之后，编译器遇到类似的情况，就不会再给出相关的警告信息，如代码清单 1-24 所示。

代码清单 1-24 使用 `@SafeVarargs` 注解抑制编译器警告的示例

```
@SafeVarargs
public static <T> T useVarargs(T... args) {
    return args.length > 0 ? args[0] : null;
}
```

@SafeVarargs注解只能用在参数长度可变的方法或构造方法上，且方法必须声明为static或final，否则会出现编译错误。一个方法使用@SafeVarargs注解的前提是，开发人员必须确保这个方法的实现中对泛型类型参数的处理不会引发类型安全问题。

1.7 小结

本章内容主要围绕Java 7中通过Coin项目添加的语法规新特性展开。这次在Java 7中对语法所做的改进，是自J2SE 5.0版本以来比较大的一次。在这几个新特性中，最值得在日常开发中使用的是在switch语句中使用字符串、在一个catch子句中捕获多个异常，以及利用try-with-resources语句管理资源。数值字面量和参数长度可变方法的调用方式的改进，也可以为开发人员带来便利。需要提醒的是，在switch语句中使用字符串时要谨慎。在多数时候，使用枚举类型是一个更好的做法。善用这些新特性，可以减少程序中的冗余代码，提高开发效率。

第 2 章 Java 语言的动态性

Java 语言是一种静态类型的编程语言。静态类型的含义是指在编译时进行类型检查。Java 源代码中的每个变量的类型都需要显式地进行声明。所有变量、方法的参数和返回值的类型在程序运行之前就必须是已知的。Java 语言的这种静态类型特性使编译器可以在编译时执行大量的检查来发现代码中明显的类型错误，不过这样的话，代码中会包含很多不必要的类型声明，使代码不够简洁和灵活。与静态类型语言相对应的是动态类型语言，如 JavaScript 和 Ruby 等。动态类型语言的类型检查在运行时进行。源代码中不需要显式地声明类型。去掉了类型声明之后，使用动态类型语言编写的代码更加简洁。近年来，动态类型语言的流行也反映了语言中动态性的最重要。适当的动态性对于提高开发的效率是有帮助的，可以减少开发人员需要编写的代码量。

对于使用 Java 的开发人员来说，学习一门新的动态类型语言的代价可能比较高，因为从一门新语言的入门到将其真正运用到实践中的时间可能比较长。熟悉 Java 的开发人员还是都希望用 Java 来解决问题。实际上，Java 语言本身对动态性的支持也有很多。这里的动态性指的不是类型上的，而是使用方式上的。这些动态性可以在一些对灵活性要求比较高的场合发挥作用。反射 API 就是一个很好的例子，它提供了在运行时根据方法名称查找并调用方法的能力。随着版本的更新，Java 语言本身也在不断地提高对动态性和灵活性的支持。

本章将围绕 Java 语言的动态性来展开，所涉及的内容既有 Java 7 中的新特性，又有之前版本中就有的功能。集中在这一章进行介绍的目的是使读者对相关知识有一个全面的了解。本章所介绍的内容都属于 Java 的标准 API，不需要了解字节代码等底层细节。这一章的内容分成 4 个部分：首先介绍 Java 6 中引入的脚本语言支持 API，接着介绍可以在运行时检查程序内部结构和直接调用方法的反射 API，然后对可以在运行时实现接口的动态代理进行讲解，最后是本章的重点，即 Java 7 中引入的在 Java 虚拟机级别实现的动态语言支持和方法句柄。

2.1 脚本语言支持 API

随着 Java 平台的流行，很多脚本语言（scripting language）都可以运行在 Java 虚拟机上，其中比较流行的有 JavaScript、JRuby、Jython 和 Groovy 等。相对 Java 语言来说，脚本语言由于其灵活性很强，非常适合在某些情况下使用，比如描述应用中复杂多变的业务逻辑，并在应用运行过程中进行动态修改；为应用提供一种领域特定语言（Domain-specific Language，DSL），供没有技术背景的普通用户使用；作为应用中各个组件之间的“胶水”，快速进行组件之间的整合；快速开发出应用的原型系统，从而迅速获取用户

反馈，并进行改进；帮助开发人员快速编写测试用例等。对于这些场景，如果使用Java来开发，会事倍功半。

对于这些运行在Java虚拟机平台上的脚本语言来说，并不需要为它们准备额外的运行环境，直接复用已有的Java虚拟机环境即可。这就节省了在运行环境上所需的成本投入。在应用开发中使用脚本语言，实际上是“多语言开发”的一种很好的实践，即根据应用的需求和语言本身的特性来选择最合适的编程语言，以快速高效地解决应用中的某一部分问题。多种不同语言实现的组件结合起来，就形成了最终的完整应用程序。比如一个应用，可以用Groovy来编写用户界面，用Java编写核心业务逻辑，用Ruby来进行数据处理。不同语言编写的代码可以同时运行在同一个Java虚拟机之上。这些脚本语言与Java语言之间的交互，是由脚本语言支持API来完成的。

JSR 223 (Scripting for the Java™ Platform) 中规范了在Java虚拟机上运行的脚本语言与Java程序之间的交互方式。JSR 223是Java SE 6的一部分，在Java标准API中的包是javax.script。下面将详细介绍与脚本语言支持API相关的内容。

2.1.1 脚本引擎

一段脚本的执行需要由该脚本语言对应的脚本引擎来完成。一个Java程序可以选择同时包含多种脚本语言的执行引擎，这完全由程序的需求来决定。程序中所用到的脚本语言，都需要有相应的脚本引擎。JSR 223中定义了脚本引擎的注册和查找机制。这对于脚本引擎的实现者来说是需要了解的。而一般的开发人员只需要了解如何通过脚本引擎管理器来获取对应语言的脚本引擎即可，并不需要了解脚本引擎的注册机制。Java SE 6中自带了JavaScript语言的脚本引擎，是基于Mozilla的Rhino来实现的。对于其他的脚本语言，则需要下载对应的脚本引擎的库并放到程序的类路径中。一般只要放在类路径中，脚本引擎就可以被应用程序发现并使用。

首先介绍脚本引擎的一般用法。在代码清单2-1中，首先创建一个脚本引擎管理器javax.script.ScriptEngineManager对象，再通过管理器来查找所需的JavaScript脚本引擎，最后通过脚本引擎来执行JavaScript代码。示例中的JavaScript代码做的事情很简单，只输出了字符串“Hello!”。JavaScript代码中的println是Rhino引擎额外提供的方法，相当于Java中的System.out.println方法。

代码清单2-1 脚本引擎的一般用法

```
public void greet() throws ScriptException {
    ScriptEngineManager manager = new ScriptEngineManager();
    ScriptEngine engine = manager.getEngineByName("JavaScript");
    if (engine == null) {
        throw new RuntimeException("找不到JavaScript语言执行引擎。");
    }
    engine.eval("println('Hello!');");
}
```

上面的代码中是通过脚本引擎的名称进行查找的。实际上，脚本引擎管理器共支持三种查找脚本引擎的方式，分别通过名称、文件扩展名和 MIME 类型来完成。比如对于同样的 JavaScript 语言引擎，还可以通过 `getEngineByExtension("js")` 和 `getEngineByMimeType("text/javascript")` 来查找到。得到脚本引擎 `ScriptEngine` 的对象之后，通过其 `eval` 方法可以执行一段代码，并返回这段代码的执行结果。这是最基本的通过脚本引擎来解释执行一段脚本的实现方式。

2.1.2 语言绑定

脚本语言支持 API 的一个很大优势在于它规范了 Java 语言与脚本语言之间的交互方式，使 Java 语言编写的程序可以与脚本之间进行双向的方法调用和数据传递。方法调用的方式会在 2.1.5 小节中介绍。数据传递是通过语言绑定对象来完成的。所谓的语言绑定对象就是一个简单的哈希表，用来存放和获取需要共享的数据。所有数据都对应这个哈希表中的一个条目，是简单的名值对。接口 `javax.script.Bindings` 定义了语言绑定对象的接口，它继承自 `java.util.Map` 接口。一个脚本引擎在执行过程中可能会使用多个语言绑定对象。不同语言绑定对象的作用域不同。在默认情况下，脚本引擎会提供多个语言绑定对象，用来存放在执行过程中产生的全局对象等。`ScriptEngine` 类提供了 `put` 和 `get` 方法对脚本引擎中特定作用域的默认语言绑定对象进行操作。程序可以直接使用这个默认的语言绑定对象，也可以使用自己的语言绑定对象。在脚本语言的执行过程中，可以将语言绑定对象看成是一个额外的变量映射表。在解析变量值的时候，语言绑定对象中的名称也会被考虑在内。脚本执行过程中产生的全局变量等内容，会出现在语言绑定对象中。通过这种方式，就完成了 Java 与脚本语言之间的双向数据传递。

在代码清单 2-2 中，首先通过 `ScriptEngine` 的 `put` 方法向脚本引擎默认的语言绑定对象中添加了一个名为“name”的字符串，接着在脚本中直接根据名称来引用这个对象。同样，在脚本中创建的全局变量“message”也可以通过 `ScriptEngine` 的 `get` 方法来获取。这样就实现了 Java 程序与脚本之间的双向数据传递。数据传递过程中的类型转换是由脚本引擎来完成的，转换规则取决于具体的脚本语言的语法。

代码清单 2-2 脚本引擎默认的语言绑定对象的示例

```
public void useDefaultBinding() throws ScriptException {
    ScriptEngine engine = getJavaScriptEngine();
    engine.put("name", "Alex");
    engine.eval("var message = 'Hello, ' + name;");
    engine.eval("println(message);");
    Object obj = engine.get("message");
    System.out.println(obj);
}
```

在大多数情况下，使用 `ScriptEngine` 的 `put` 和 `get` 方法就足够了。如果仅使用 `put` 和 `get` 方法，语言绑定对象本身对于开发人员来说是透明的。在某些情况下，需要使用

程序自己的语言绑定对象，比如语言绑定对象中包含了程序自己独有的数据。如果希望使用自己的语言绑定对象，可以调用脚本引擎的 `createBindings` 方法或创建一个 `javax.script.SimpleBindings` 对象，并传递给脚本引擎的 `eval` 方法，如代码清单 2-3 所示。

代码清单 2-3 自定义语言绑定对象的示例

```
public void useCustomBinding() throws ScriptException {
    ScriptEngine engine = getJavaScriptEngine();
    Bindings bindings = new SimpleBindings();
    bindings.put("hobby", "playing games");
    engine.eval("println('I like ' + hobby);", bindings);
}
```

通过 `eval` 方法传递的语言绑定对象，仅在当前 `eval` 调用中生效，并不会改变引擎默认的语言绑定对象。

2.1.3 脚本执行上下文

与脚本引擎执行相关的另外一个重要接口是 `javax.script.ScriptContext`，其中包含脚本引擎执行过程中的相关上下文信息，可以与 Java EE 中 `servlet` 规范中的 `javax.servlet.ServletContext` 接口来进行类比。脚本引擎通过此上下文对象来获取与脚本执行相关的信息，也允许开发人员通过此对象来配置脚本引擎的行为。该上下文对象中主要包含以下 3 类信息。

1. 输入与输出

首先介绍与脚本输入和输出相关的配置信息，其中包括脚本在执行中用来读取数据输入的 `java.io.Reader` 对象以及输出正确内容和出错信息的 `java.io.Writer` 对象。在默认情况下，脚本的输入输出都发生在标准控制台中。如果希望把脚本的输出写入到文件中，可以使用代码清单 2-4 中的代码。通过 `setWriter` 方法把脚本的输出重定向到一个文件中。通过 `ScriptContext` 的 `setReader` 和 `setErrorWriter` 方法可以分别设置脚本执行时的数据输入来源和产生错误时出错信息的输出目的。

代码清单 2-4 把脚本运行时的输出写入到文件中的示例

```
public void scriptToFile() throws IOException, ScriptException {
    ScriptEngine engine = getJavaScriptEngine();
    ScriptContext context = engine.getContext();
    context.setWriter(new FileWriter("output.txt"));
    engine.eval("println('Hello World!');");
}
```

2. 自定义属性

下面介绍执行上下文中包含的自定义属性。`ScriptContext` 中也有与 `ServletContext` 中类似的获取和设置属性的方法，即 `setAttribute` 和 `getAttribute`。所不同的是，

ScriptContext 中的属性是有作用域之分的。不同作用域的区别在于查找属性时的顺序不同。每个作用域都以一个对应的整数表示其查找顺序。该整数值越小，说明查找时的顺序越优先。优先级高的作用域中的属性会隐藏优先级低的作用域中的同名属性。因此，设置属性时需要显式地指定所在的作用域。在获取属性的时候，既可以选择在指定的作用域中查找，也可以选择根据作用域优先级自动进行查找。

不过脚本执行上下文实现中包含的作用域是固定的，开发人员不能随意定义自己的作用域。通过 ScriptContext 的 getScopes 方法可以得到所有可用的作用域列表。ScriptContext 中预先定义了两个作用域：常量 ScriptContext.ENGINE_SCOPE 表示的作用域对应的是当前的脚本引擎，而 ScriptContext.GLOBAL_SCOPE 表示的作用域对应的是从同一引擎工厂中创建出来的所有脚本引擎对象。前者的优先级较高。代码清单 2-5 给出了作用域影响同名属性查找的一个示例。ENGINE_SCOPE 中的属性“name”隐藏了 GLOBAL_SCOPE 中的同名属性。

代码清单 2-5 作用域影响同名属性查找的示例

```
public void scriptContextAttribute() {
    ScriptEngine engine = getJavaScriptEngine();
    ScriptContext context = engine.getContext();
    context.setAttribute("name", "Alex", ScriptContext.GLOBAL_SCOPE);
    context.setAttribute("name", "Bob", ScriptContext.ENGINE_SCOPE);
    context.getAttribute("name"); // 值为 Bob
}
```

3. 语言绑定对象

脚本执行上下文中的最后一类信息是语言绑定对象。语言绑定对象也是与作用域相对应的。同样的作用域优先级顺序对语言绑定对象也适用。这样的优先级顺序会对脚本执行时的变量解析产生影响。比如在代码清单 2-6 中，两个不同的语言绑定对象中都有名称为“name”的对象，而在脚本的执行过程中，作用域 ENGINE_SCOPE 的语言绑定对象的优先级较高，因此变量“name”的值是“Bob”。

代码清单 2-6 语言绑定对象的优先级顺序的示例

```
public void scriptContextBindings() throws ScriptException {
    ScriptEngine engine = getJavaScriptEngine();
    ScriptContext context = engine.getContext();
    Bindings bindings1 = engine.createBindings();
    bindings1.put("name", "Alex");
    context.setBindings(bindings1, ScriptContext.GLOBAL_SCOPE);
    Bindings bindings2 = engine.createBindings();
    bindings2.put("name", "Bob");
    context.setBindings(bindings2, ScriptContext.ENGINE_SCOPE);
    engine.eval("println(name);");
}
```

通过 ScriptContext 的 setBindings 方法设置的语言绑定对象会影响到 ScriptEngine 在执行脚本时的变量解析。ScriptEngine 的 put 和 get 方法所操作的实际上就是 ScriptContext 中作用域为 ENGINE_SCOPE 的语言绑定对象。在代码清单 2-7 中，从 ScriptContext 中得到语言绑定对象之后，可以直接对这个对象进行操作。如果在 ScriptEngine 的 eval 方法中没有指明所使用的语言绑定对象，实际上起作用的是 ScriptContext 中作用域为 ENGINE_SCOPE 的语言绑定对象。

代码清单 2-7 通过脚本执行上下文获取语言绑定对象的示例

```
public void useScriptContextValues() throws ScriptException {
    ScriptEngine engine = getJavaScriptEngine();
    ScriptContext context = engine.getContext();
    Bindings bindings = context.getBindings(ScriptContext.ENGINE_SCOPE);
    bindings.put("name", "Alex");
    engine.eval("println(name);"); // 输出 Alex
}
```

上一小节介绍的自定义属性实际上也保存在语言绑定对象中。在代码清单 2-8 中，不直接操作语言绑定对象本身，而是通过 ScriptContext 的 setAttribute 来向语言绑定对象中添加数据。所添加的数据在脚本执行时也同样是可见的。

代码清单 2-8 自定义属性保存在语言绑定对象中的示例

```
public void attributeInBindings() throws ScriptException {
    ScriptEngine engine = getJavaScriptEngine();
    ScriptContext context = engine.getContext();
    context.setAttribute("name", "Alex", ScriptContext.GLOBAL_SCOPE);
    engine.eval("println(name);"); // 输出为 Alex
}
```

2.1.4 脚本的编译

脚本语言一般是解释执行的。脚本引擎在运行时需要先解析脚本之后再执行。一般来说，通过解释执行的方式来运行脚本的速度比编译之后再运行会慢一些。当一段脚本需要被多次重复执行时，可以先对脚本进行编译。编译之后的脚本在执行时不需要重复解析，可以提高执行效率。不是所有的脚本引擎都支持对脚本进行编译。如果脚本引擎支持这一特性，它会实现 javax.script.Compilable 接口来声明这一点。脚本引擎的使用者可以利用这个能力来提高需要多次执行的脚本的运行效率。Java SE 中自带的 JavaScript 脚本引擎是支持对脚本进行编译的。

在代码清单 2-9 中，Compilable 接口的 compile 方法用来对脚本代码进行编译，编译的结果用 javax.script.CompiledScript 来表示。由于不是所有的脚本引擎都支持 Compilable 接口，因此这里需要用 instanceof 进行判断。在 run 方法中，通过

CompiledScript 的 eval 方法就可以执行脚本。代码中把一段脚本重复执行了 100 次，以此说明编译完的脚本在重复执行时的性能优势。

代码清单 2-9 进行脚本编译的示例

```
public CompiledScript compile(String scriptText) throws ScriptException {
    ScriptEngine engine = getJavaScriptEngine();
    if (engine instanceof Compilable) {
        CompiledScript script = ((Compilable) engine).compile(scriptText);
        return script;
    }
    return null;
}

public void run(String scriptText) throws ScriptException {
    CompiledScript script = compile(scriptText);
    if (script == null) {
        return;
    }
    for (int i = 0; i < 100; i++) {
        script.eval();
    }
}
```

CompiledScript 的 eval 方法所接受的参数与 ScriptEngine 的 eval 方法是相同的。

2.1.5 方法调用

在脚本中，最常见和最实用的就是方法。有些脚本引擎允许使用者单独调用脚本中的某个方法。支持这种方法调用方式的脚本引擎可以实现 javax.script.Invocable 接口。通过 Invocable 接口可以调用脚本中的顶层方法，也可以调用对象中的成员方法。如果脚本中顶层方法或对象中的成员方法实现了 Java 中的接口，可以通过 Invocable 接口中的方法来获取脚本中相应的 Java 接口的实现对象。这样就可以在 Java 语言中定义接口，在脚本中实现接口。程序中使用该接口的其他部分代码并不知道接口是由脚本来实现的。与 Compilable 接口一样，ScriptEngine 对于 Invocable 接口的实现也是可选的。

代码清单 2-10 通过 Invocable 接口的 invokeFunction 来调用脚本中的顶层方法，调用时的参数会被传递给脚本中的方法。因为 Java SE 自带的 JavaScript 脚本引擎实现了 Invocable 接口，所以这里省去了对引擎是否实现了 Invocable 接口的判断。

代码清单 2-10 在 Java 中调用脚本中顶层方法的示例

```
public void invokeFunction() throws ScriptException, NoSuchMethodException {
    ScriptEngine engine = getJavaScriptEngine();
    String scriptText = "function greet(name) { println('Hello, ' + name); } ";
    engine.eval(scriptText);
    Invocable invocable = (Invocable) engine;
```

```

    invocable.invokeFunction("greet", "Alex");
}

```

如果被调用的方法是脚本中对象的成员方法，就需要使用 invokeMethod 方法，如代码清单 2-11 所示。代码中的 getGreeting 方法是属于对象 obj 的，在调用的时候需要把这个对象作为参数传递进去。

代码清单 2-11 在 Java 中调用脚本中对象的成员方法的示例

```

public void invokeMethod() throws ScriptException, NoSuchMethodException {
    ScriptEngine engine = getJavaScriptEngine();
    String scriptText = "var obj = { getGreeting : function(name) { return 'Hello,
        ' + name; } }; ";
    engine.eval(scriptText);
    Invocable invocable = (Invocable) engine;
    Object scope = engine.get("obj");
    Object result = invocable.invokeMethod(scope, "getGreeting", "Alex");
    System.out.println(result);
}

```

方法 invokeMethod 与方法 invokeFunction 的用法差不多，区别在于 invokeMethod 要指定包含待调用方法的对象。

在有些脚本引擎中，可以在 Java 语言中定义接口，并在脚本中编写接口的实现。这样程序中的其他部分可以只同 Java 接口交互，并不需要关心接口是由什么方式来实现的。在代码清单 2-12 中，Greet 是用 Java 定义的接口，其中包含一个 getGreeting 方法。在脚本中实现这个接口。通过 getInterface 方法可以得到由脚本实现的这个接口的对象，并调用其中的方法。

代码清单 2-12 在脚本中实现 Java 接口的示例

```

public void useInterface() throws ScriptException {
    ScriptEngine engine = getJavaScriptEngine();
    String scriptText = "function getGreeting(name) { return 'Hello, ' + name; } ";
    engine.eval(scriptText);
    Invocable invocable = (Invocable) engine;
    Greet greet = invocable.getInterface(Greet.class);
    System.out.println(greet.getGreeting("Alex"));
}

```

代码清单 2-12 中的接口的实现是由脚本中的顶层方法来完成的。同样的，也可以由脚本中对象的成员方法来实现。对于这种情况，getInterface 方法的另外一种重载形式可以接受一个额外的参数来指定接口实现所在的对象。

2.1.6 使用案例

由于脚本语言的语法简单和灵活，非常适于没有或只有少量编程背景的用户来使

用。这些用户可以通过脚本语言来定制程序的业务逻辑和用户界面等。通过脚本语言，可以在程序的易用性和灵活性之间达到一个比较好的平衡。比如脚本语言 Lua 就被广泛应用在游戏开发中，用来对游戏的内部行为和用户界面进行定制。

下面通过一个具体的案例来说明如何使用脚本语言。这个案例也和游戏有关。一般的游戏都会自带一个控制台，允许用户输入命令来对游戏本身进行修改。这里展示的是一个通过 JavaScript 来进行游戏配置的案例。这个游戏控制台的实现被大大简化了。运行时在一个线程中等待用户输入命令。对输入的命令进行适当处理之后就交给 JavaScript 脚本引擎来执行。这里的 GameConfig 表示的是与游戏有关的配置信息，可以通过 JavaScript 语言在控制台中进行修改。代码清单 2-13 给出了这个简易的游戏控制台的基本实现。GameConfig 的 getScriptBindings 方法返回的语言绑定对象中包含的是对于游戏控制台可见的配置项。比如，用户在控制台输入 “config.screenWidth = 300” 就可以直接修改 GameConfig 中的 screenWidth 域的值。

代码清单 2-13 使用脚本引擎实现的游戏控制台

```
public class GameConsole extends JsScriptRunner implements Runnable {
    private GameConfig config;

    public GameConsole(GameConfig config) {
        this.config = config;
    }

    public void executeCommand(String command) throws ScriptException {
        ScriptEngine engine = getJavaScriptEngine();
        if (command.indexOf("println") == -1) {
            command = "println(" + command + ");";
        }
        engine.eval(command, config.getScriptBindings());
    }

    public void run() {
        Scanner scanner = new Scanner(System.in);
        while (true) {
            String line = scanner.nextLine();
            if ("quit".equals(line)) {
                break;
            }
            try {
                executeCommand(line);
            } catch (ScriptException ex) {
                System.err.println("错误的命令！");
            }
        }
    }
}
```

通过这个简易的控制台，用户对脚本语言所做的配置修改，对于通过Java语言来实现的其他组件来说是立即生效的。

2.2 反射 API

2.1节介绍的Java脚本语言支持API是通过引入其他脚本语言来增强Java平台的动态性，而这一节将要介绍的反射API则是Java语言本身提供的动态支持。通过反射API可以获取Java程序在运行时刻的内部结构，比如Java类中包含的构造方法、域和方法等元素，并可以与这些元素进行交互。通过反射API，Java语言也可以实现很多动态语言所支持的实用而又简洁的功能。下面先通过一个示例来为读者提供一个反射API的直观印象。

按照一般的面向对象的设计思路，一个对象的内部状态都应该通过相应的方法来改变，而不是直接去修改属性的值。一般Java类中的属性设置和获取方法的命名都遵循JavaBeans规范中的要求，即利用setXxx和getXxx这样的方法声明。因此可以实现一个实用工具类来完成对任意对象的属性设置和获取的操作，只要设置和获取属性的方法满足JavaBeans规范。具体的实现方式可以通过与动态语言进行对比来分别介绍。用JavaScript语言来实现这样功能，如代码清单2-14所示。限于篇幅，代码中省略了应有的类型检查和错误处理。

代码清单2-14 设置任意对象的属性值的JavaScript实现

```
function invokeSetter(obj, property, value) {
    var funcName = "set" + property.substring(0,1).toUpperCase() + property.
        substring(1);
    obj[funcName](value);
}
var obj = {
    value : 0,
    setValue : function(val) { this.value = val; }
};
invokeSetter(obj, "value", 5);
```

上面的代码只是属性设置方法的示例。代码的逻辑也并不复杂，首先把要设置的属性名称按照JavaBeans规范转换成对应的方法名称，如设置属性“value”的方法名称为“setValue”。由于JavaScript语言本身的特性，方法也是对象的属性，因此可以直接获取到方法后再进行调用。

代码清单2-15给出了使用反射API的Java实现。从代码量上来说，与JavaScript的实现差别并不算大。基本的实现思路也比较直接，先从对象的类中查找到方法，再用传入的参数调用此方法。这个静态方法可以作为一个实用工具方法在程序中使用。

代码清单 2-15 使用反射 API 设置对象的属性值的示例

```
public class ReflectSetter {  
    public static void invokeSetter(Object obj, String field, Object value) throws  
        NoSuchMethodException, InvocationTargetException, IllegalAccessException {  
        String methodName = "set" + field.substring(0, 1).toUpperCase() + field.  
            substring(1);  
        Class<?> clazz = obj.getClass();  
        Method method = clazz.getMethod(methodName, value.getClass());  
        method.invoke(obj, value);  
    }  
}
```

从上面的示例可以看出，通过反射 API，Java 语言也可以应用在很多对灵活性要求很高的场景中。从根本上来说，反射 API 实际上定义了一种功能提供者和使用者之间的松散契约。以方法调用为例，按照 Java 语言的一般做法，在调用方法的时候，在代码中首先需要一个对象的变量作为调用的接收者，再把方法的名称直接写在代码中。方法的名称不可能是变量。编译器会检查这个对象中是否确实有待调用的方法，如果没有就会出现编译错误。这种一般的做法，是提供者和使用者之间的一种紧密的契约，由编译器来保证其合法性。而使用反射 API，两者的契约只需要建立在名称和参数类型这个层次上就足够了。方法名称可以是变量，参数值也可以动态生成。调用的合法性由开发人员自己保证。如果方法调用不是合法的，相关的异常会在运行时抛出。

反射 API 的一个重要的使用场合是要调用的方法或者要操作的域的名称按照某种规律变化的时候。一个典型的场景就是在 Servlet 中用 HTTP 请求的参数值来填充领域对象。比如在用户注册的时候，包含在 HTTP 请求中的用户所填写的相关信息，需要被填充到程序中定义好的领域对象中。只需要利用 Servlet 提供的 API 遍历请求中的所有参数，然后用代码清单 2-15 中给出的 invokeSetter 方法设置领域对象中与参数名称相对应的属性的值即可。另外一个场景是在数据库操作中，从 SQL 查询结果集中创建并填充领域对象。数据库的列名和领域对象的属性名称也存在着类似的对应关系。

反射 API 在为 Java 程序带来灵活性的同时，也产生了额外的性能代价。由于反射 API 的实现机制，对于相同的操作，比如调用同一个方法，用反射 API 来动态实现比直接在源代码中编写的方式大概慢一到两个数量级。随着 Java 虚拟机实现的改进，反射 API 的性能已经有了非常大的提升。但是这种性能的差距是客观存在的。因此，在某些对性能要求比较高的应用中，要慎用反射 API。

2.2.1 获取构造方法

通过反射 API 可以获取到 Java 类中的构造方法。通过构造方法可以在运行时动态地创建 Java 对象，而不只是通过 new 操作符来进行创建。在得到 Class 类的对象之后，可以通过其中的方法来获取构造方法。相关的方法有 4 个，其中 getConstructors 用来获取

所有的公开构造方法的列表，`getConstructor`则根据参数类型来获取公开的构造方法。另外两个对应方法`getDeclaredConstructors`和`getDeclaredConstructor`的作用类似，只不过它们会获取类中真正声明的构造方法，而忽略从父类中继承下来的构造方法。得到了表示构造方法的`java.lang.reflect.Constructor`对象之后，就可以获取关于构造方法的更多信息，以及通过`newInstance`方法创建出新的对象。

一般的构造方法的获取和使用并没有什么特殊之处，需要特别说明的是对参数长度可变的构造方法和嵌套类（nested class）的构造方法的使用。

如果构造方法声明了长度可变的参数，在获取构造方法的时候，要使用对应的数组类型的`Class`对象。这是因为长度可变的参数实际上是通过数组来实现的。如代码清单2-16所示，类`VarargsConstructor`的构造方法包含`String`类型的可变长度参数，在调用`getDeclaredConstructor`方法的时候，需要使用`String[].class`，否则会找不到该构造方法。在调用`newInstance`的时候，要把作为实际参数的字符串数组先转换成为`Object`类型，这是为了避免方法调用时的歧义。这样编译器就知道把这个字符串数组作为一个可变长度的参数来传递。

代码清单2-16 使用反射API获取参数长度可变的构造方法

```
public class VarargsConstructor {
    public VarargsConstructor(String... names) {}
}

public void useVarargsConstructor() throws Exception {
    Constructor<VarargsConstructor> constructor = VarargsConstructor.class.
        getDeclaredConstructor(String[].class);
    constructor.newInstance((Object) new String[]{"A", "B", "C"});
}
```

对嵌套类的构造方法的获取，需要区分静态和非静态两种情况，即是否在声明嵌套类的时候使用`static`关键词。静态的嵌套类并没有特别之处，按照一般的方式来使用即可。而对于非静态嵌套类来说，其特殊之处在于它的对象实例中都有一个隐含的对象引用，指向包含它的外部类对象。也正是这个隐含的对象引用的存在，使非静态嵌套类中的代码可以直接引用外部类中包含的私有域和方法。因此，在获取非静态嵌套类的构造方法的时候，类型参数列表的第一个值必须是外部类的`Class`对象。如代码清单2-17所示，静态嵌套类`StaticNestedClass`的使用并没有特殊之处。在获取到非静态嵌套类`NestedClass`的构造方法之后，用`newInstance`创建新对象，此时第一个参数就是其外部对象的引用`this`，与调用`getDeclaredConstructor`方法时的第一个参数相对应。

代码清单2-17 使用反射API获取嵌套类的构造方法

```
static class StaticNestedClass {
    public StaticNestedClass(String name) {}
```

```

    }

    class NestedClass {
        public NestedClass(int count) {}
    }

    public void useNestedClassConstructor() throws Exception {
        Constructor<StaticNestedClass> sncc = StaticNestedClass.class.
            getDeclaredConstructor(String.class);
        sncc.newInstance("Alex");
        Constructor<NestedClass> ncc = NestedClass.class.getDeclaredConstructor(Const
            ructorUsage.class, int.class);
        NestedClass ic = ncc.newInstance(this, 3);
    }
}

```

2.2.2 获取域

除了可以获取 2.2.1 节提到的构造方法之外，还可以通过反射 API 获取类中的域（field）。通过反射 API 可以获取到类中公开的静态域和对象中的实例域。得到表示域的 `java.lang.reflect.Field` 类的对象之后，就可以获取和设置域的值。与上面的构造方法类似，`Class` 类中也有 4 个方法用来获取域，分别是 `getFields`、`getField`、`getDeclaredFields` 和 `getDeclaredField`，其含义与获取构造方法的 4 个方法类似。代码清单 2-18 给出了获取和使用静态域和实例域的示例，两者的区别在于使用静态域时不需要提供具体的对象实例，使用 `null` 即可。`Field` 类中除了操作 `Object` 的 `get` 和 `set` 方法之外，还有操作基本类型的对应方法，包括 `getBoolean` / `setBoolean`、`getByte` / `setByte`、`getChar` / `setChar`、`getDouble` / `setDouble`、`getFloat` / `setFloat`、`getInt` / `setInt` 和 `getLong` / `setLong` 等。

代码清单 2-18 使用反射 API 获取和使用静态域和实例域

```

public void useField() throws Exception {
    Field fieldCount = FieldContainer.class.getDeclaredField("count");
    fieldCount.set(null, 3);
    Field fieldName = FieldContainer.class.getDeclaredField("name");
    FieldContainer fieldContainer = new FieldContainer();
    fieldName.set(fieldContainer, "Bob");
}

```

总的来说，对域的获取和设置都比较简单。但是只能对类中的公开域进行操作。私有域没有办法通过反射 API 获取到，也无法进行操作。

2.2.3 获取方法

最后一个可以通过反射 API 获取的元素是方法，这也是最常使用反射 API 的场景，即获取到一个对象中的方法，并在运行时调用该方法。与之前提到的构造方法

和域类似，Class类中也有4个方法用来获取方法，分别是getMethods、getMethod、getDeclaredMethods和getDeclaredMethod。这4个方法的含义类似于获取构造方法和域的对应方法。在得到了表示方法的java.lang.reflect.Method类的对象之后，就可以查询该方法的详细信息，比如方法的参数和返回值的类型等。最重要的是可以通过invoke方法来传入实际参数并调用该方法。代码清单2-19中分别给出了获取和调用对象中的公开和私有方法的示例。需要注意的是，在调用私有方法之前，需要先调用Method类的setAccessible方法来设置可以访问的权限。

代码清单2-19 使用反射API获取和使用公开和私有方法

```
public void useMethod() throws Exception {
    MethodContainer mc = new MethodContainer();
    Method publicMethod = MethodContainer.class.getDeclaredMethod("publicMethod");
    publicMethod.invoke(mc);
    Method privateMethod = MethodContainer.class.getDeclaredMethod("privateMethod");
    privateMethod.setAccessible(true);
    privateMethod.invoke(mc);
}
```

与构造方法和域不同的是，通过反射API可以获取到类中的私有方法。

2.2.4 操作数组

使用反射API对数组进行操作的方式不同于一般的Java对象，是通过专门的java.lang.reflect.Array这个实用工具类来实现的。Array类中提供的方法包括创建数组和操作数组中的元素。如代码清单2-20所示，newInstance方法用来创建新的数组，第一个参数是数组中元素的类型，后面的参数是数组的维度信息。比如names是一个长度为10的一维String数组。matrix1是一个 $3 \times 3 \times 3$ 的三维数组。由于matrix2的元素类型是int[]，虽然在创建时只声明了两个维度，但是它实际上也是一个三维数组。

代码清单2-20 使用反射API操作数组

```
public void useArray() {
    String[] names = (String[]) Array.newInstance(String.class, 10);
    names[0] = "Hello";
    Array.set(names, 1, "World");
    String str = (String) Array.get(names, 0);
    int[][][] matrix1 = (int[][][]) Array.newInstance(int.class, 3, 3, 3);
    matrix1[0][0][0] = 1;
    int[][][] matrix2 = (int[][][]) Array.newInstance(int[].class, 3, 4);
    matrix2[0][0] = new int[10];
    matrix2[0][1] = new int[3];
    matrix2[0][0][1] = 1;
}
```

2.2.5 访问权限与异常处理

使用反射 API 的一个重要好处是可以绕过 Java 语言中默认的访问控制权限。比如在正常的代码中，一个类的对象是不能访问在另外一个类中声明的私有方法的，但是通过反射 API 可以做到这一点，具体的做法如代码清单 2-19 所示。Constructor、Field 和 Method 都继承自 `java.lang.reflect.AccessibleObject`，其中的方法 `setAccessible` 可以用来设置是否绕开默认的权限检查。

在利用 `invoke` 方法来调用方法时，如果方法本身抛出了异常，`invoke` 方法会抛出 `InvocationTargetException` 异常来表示这种情况。在捕获到 `InvocationTargetException` 异常的时候，通过 `InvocationTargetException` 异常的 `getCause` 方法可以获取到真正的异常信息，帮助进行调试。

值得一提的是，Java 7 为所有与反射操作相关的异常类添加了一个新的父类 `java.lang.ReflectiveOperationException`。在处理与反射相关的异常的时候，可以直接捕获这个新的异常。而在 Java 7 之前，这些异常是需要分别捕获的。

2.3 动态代理

这一节要介绍 Java 语言支持动态性的另外一个方面，即动态代理（dynamic proxy）机制。这个名称中的“代理”会让人很容易联想到设计模式中的代理模式。实际上，使用动态代理机制，不但可以实现代理模式，还可以实现装饰器和适配器模式。通过使用动态代理，可以在运行时动态创建出同时实现多个 Java 接口的代理类及其对象实例。当客户代码通过这些被代理的接口来访问其中的方法时，相关的调用信息会被传递给代理中的一个特殊对象进行处理，处理的结果作为方法调用的结果返回。动态代理的这种实现机制，属于代理模式的基本用法。客户代码看到的只是接口，具体的逻辑被封装在代理的实现中。

动态代理机制的强大之处在于可以在运行时动态实现多个接口，而不需要在源代码中通过 `implements` 关键词来声明。同时，动态代理把对接口中方法调用的处理逻辑交给开发人员，让开发人员可以灵活处理。通过动态代理可以实现面向方面编程（AOP）中常见的方法拦截功能。

2.3.1 基本使用方式

使用动态代理时只需要理解两个要素即可：第一个是要代理的接口，另外一个是处理接口方法调用的 `java.lang.reflect.InvocationHandler` 接口。动态代理只支持对接口提供代理，一般的 Java 类是不行的。如果要代理的接口不是公开的，那么被代理的接口和创建动态代理的代码必须在同一个包中。在创建动态代理的时候，需要提供 `InvocationHandler` 接口的实现，以处理实际的调用。在进行处理的时候可以得到表示

实际调用方法的 Method 对象和调用的实际参数列表。代码清单 2-21 给出了一个简单的 InvocationHandler 接口的实现类。InvocationHandler 接口只有一个需要实现的方法 invoke。当客户代码调用被代理的接口中的方法时，invoke 方法就会被调用，而代理对象、所调用方法的 Method 对象和实际参数列表都会作为 invoke 方法的参数。在下面 invoke 方法的实现代码中，只是简单地通过 Java 的日志 API 记录下方法调用的相关信息，再调用原始的方法，并返回结果。

代码清单 2-21 InvocationHandler 接口的实现类的示例

```
public class LoggingInvocationHandler implements InvocationHandler {
    private static final Logger LOGGER = Logger.getLogger(LoggingInvocationHandler.class);
    private Object receiverObject;
    public LoggingInvocationHandler(Object object) {
        this.receiverObject = object;
    }
    public Object invoke(Object proxy, Method method, Object[] args) throws
        Throwable {
        LOGGER.log(Level.INFO, "调用方法 " + method.getName() + "；参数为 " +
            Arrays.deepToString(args));
        return method.invoke(receiverObject, args);
    }
}
```

在有了 InvocationHandler 接口的实现之后，就可以创建和使用动态代理，代码清单 2-22 给出了一个示例。创建动态代理时需要一个 InvocationHandler 接口的实现，这里用到的是上面的 LoggingInvocationHandler 类的实例。动态代理的创建是由 java.lang.reflect.Proxy 类的静态方法 newProxyInstance 来完成的。创建时需要提供类加载器实例、被代理的接口列表以及 InvocationHandler 接口的实现。在创建完成之后，需要通过类型转换把代理对象转换成被代理的某个接口来使用。

代码清单 2-22 创建和使用动态代理的示例

```
public static void useProxy() {
    String str = "Hello World";
    LoggingInvocationHandler handler = new LoggingInvocationHandler(str);
    ClassLoader cl = SimpleProxy.class.getClassLoader();
    Comparable obj = (Comparable) Proxy.newProxyInstance(cl, new Class[]
        {Comparable.class}, handler);
    obj.compareTo("Good");
}
```

在上面的代码中，当通过代理对象的 Comparable 接口来调用其中的方法时，这个调用会被传递给 LoggingInvocationHandler 中的 invoke 方法。代理对象本身 obj、所调用的方法 compareTo 对应的 Method 对象，以及实际参数字符串“Good”都会作为参数传

递过去。在输出相关的日志信息之后，原始的 `compareTo` 方法会被执行。而 `invoke` 方法的执行结果则被作为方法调用“`obj.compareTo("Good")`”的返回结果。

虽然 `LoggingInvocationHandler` 类只是简单地记录了日志，并没有改变方法的实际执行，但是实际上，在 `InvocationHandler` 接口的 `invoke` 方法中可以实现各种各样复杂的逻辑。比如对实际调用的参数进行变换，或是改变实际调用的方法，还可以对调用的返回结果进行修改。开发人员可以根据自己的需要，添加感兴趣的业务逻辑。这实际上就是 AOP 中常用的方法拦截，即拦截一个方法调用，以在其上附加所需的业务逻辑。`InvocationHandler` 很适合于封装一些横切（cross-cutting）的代码逻辑，包括日志、参数检查与校验、异常处理和返回值归一化等。

一般来说，在创建一个动态代理的 `InvocationHandler` 实例的时候，需要把原始的方法调用的接收者对象也传入进去，以方便执行原始的方法调用。这可以在创建 `InvocationHandler` 的时候，通过构造方法来传递。在大多数情况下，代理对象只会实现一个 Java 接口。对于这种情况，可以结合泛型来开发一个通用的工厂方法，以创建代理对象。在代码清单 2-23 中，工厂方法 `makeProxy` 为任何接口及其实现类创建代理。

代码清单 2-23 为任何接口及其实现类创建代理的工厂方法

```
public static <T> T makeProxy(Class<T> intf, final T object) {
    LoggingInvocationHandler handler = new LoggingInvocationHandler(object);
    ClassLoader cl = object.getClass().getClassLoader();
    return (T) Proxy.newProxyInstance(cl, new Class<?>[] { intf }, handler);
}
```

上面的通用工厂方法的使用方式如代码清单 2-24 所示。

代码清单 2-24 创建代理对象的工厂方法的使用示例

```
public static void useGenericProxy() {
    String str = "Hello World";
    Comparable proxy = makeProxy(Comparable.class, str);
    proxy.compareTo("Good");
    List<String> list = new ArrayList<String>();
    list = makeProxy(List.class, list);
    list.add("Hello");
}
```

在这里需要注意的是，通过 `Proxy.newProxyInstance` 创建出来的代理对象只能转换成它所实现的接口类型，而不能转换成接口的具体实现类。这是因为动态代理只对接口起作用。

上面的示例代码都只代理了一个接口，如果希望代理多个接口，只需要传入多个接口类即可。所得到的代理对象可以被类型转换成这些接口中的任何一个。如果希望直接代理某个类所实现的所有接口，可以参考代码清单 2-25 中的做法。代码清单 2-25 中的 `proxyAll` 方法并没有对创建的代理对象进行类型转换，而是直接返回给调用者。这是

为了让调用者可以灵活操作，允许它们根据需要转换成不同的接口。比如，如果传入的是 String 类的对象实例，则调用者可以将其转换成 String 类所实现的 Comparable 或是 CharSequence 接口。

代码清单 2-25 代理某个类所实现的所有接口

```
public static Object proxyAll(final Object object) {
    LoggingInvocationHandler handler = new LoggingInvocationHandler(object);
    ClassLoader cl = object.getClass().getClassLoader();
    Class<?>[] interfaces = object.getClass().getInterfaces();
    return Proxy.newProxyInstance(cl, interfaces, handler);
}
```

上面介绍的是通过 `Proxy.newProxyInstance` 方法来直接创建动态代理对象，实际上这是一个快速创建代理对象的捷径。还可以通过 `Proxy.getProxyClass` 方法来首先获取到代理类。得到的代理类实现了被代理的接口。通过 `Proxy.newProxyInstance` 方法得到的代理对象实际上是通过反射 API 调用代理类的构造方法来得到的。代理类的构造方法只有一个参数，即前面提到的 `InvocationHandler` 接口的实现。对于一个 Java 类，可以通过 `Proxy.isProxy` 方法来判断是否为代理类。

当同时代理多个接口时，这些接口在代理类创建时的排列顺序就显得尤为重要。即便是同样的接口，不同的排列顺序所产生的代理类也是不同的。实际上，对于相同排列的接口类型，其对应的代理类只会被创建一次。创建完成之后就会被缓存起来。之后的创建请求得到的是缓存的代理类。强调接口的排列顺序的一个重要原因是，这个顺序会对接口中声明类型相同的方法的选择产生影响。如果多个接口中都存在声明类型相同的方法，那么在调用方法时，排列顺序中最先出现的接口中的方法会被选择。代码清单 2-26 中给出了被代理的接口中包含声明类型相同的方法的情况。在这里并没有使用 `Proxy.newProxyInstance` 方法来直接创建代理对象，而是先通过 `Proxy.getProxyClass` 来创建代理类，再使用反射 API 来创建代理类的对象。

代码清单 2-26 被代理的接口中包含声明类型相同的方法的示例

```
public void proxyMultipleInterfaces() throws Throwable {
    List<String> receiverObj = new ArrayList<String>();
    ClassLoader cl = MultipleInterfacesProxy.class.getClassLoader();
    LoggingInvocationHandler handler = new LoggingInvocationHandler(receiverObj);
    Class<?> proxyClass = Proxy.getProxyClass(cl, new Class<?>[]{List.class, Set.class});
    Object proxy = proxyClass.getConstructor(new Class[]{InvocationHandler.class})
        .newInstance(new Object[]{handler});
    List list = (List) proxy;
    list.add("Hello");
    Set set = (Set) proxy;
    set.add("World");
}
```

在上面代码中，代理类代理了 `java.util.List` 和 `java.util.Set` 两个接口，所以下面两个对代理对象进行类型转换的操作都会成功。而 `LoggingInvocationHandler` 中实际调用的接收者 `receiverObj` 其实是一个 `java.util.ArrayList` 对象，并没有实现 `Set` 接口。但是上面代码中的 `set.add("World")` 语句并不会出现错误。这是因为创建代理类时，`List` 接口出现在 `Set` 接口的前面。当调用 `add` 方法的时候，实际上调用的是 `List` 接口中的方法，而与转换之后的接口类型 `Set` 无关。如果把 `List` 和 `Set` 接口在创建代理类时的顺序调换一下，再运行代码就会出现错误。因为调换之后实际调用的是 `Set` 接口中的 `add` 方法，而实际的调用接收者并没有实现 `Set` 接口，所以会出现类型错误。

注意 在通过动态代理对象来调用 `Object` 类中声明的 `equals`、`hashCode` 和 `toString` 等方法的时候，这个调用也会被传递给 `InvocationHandler` 中的 `invoke` 方法。

动态代理的关键就是上面提到的 `InvocationHandler` 接口，通过它可以添加动态的方法调用逻辑。从 `InvocationHandler` 的 `invoke` 方法可以看出，动态代理在方法调用上额外添加一个新的抽象层次，使开发人员有机会在方法调用发生时和实际的调用执行之间，添加自己的代码逻辑。如果希望根据调用方法的名称和参数的不同，实现不同的逻辑，可以考虑使用动态代理，以减少代码重复。

2.3.2 使用案例

下面用一个完整的案例来说明动态代理在实际开发中的作用。在实际开发中，我们会遇到的一个具体问题就是程序的版本更新。在开发新版本的时候，一方面要考虑与旧版本的兼容，另一方面又希望能够修复旧版本中存在的一些设计上的问题。动态代理可以帮助平衡这两方面的需求。

比如在程序中有一个接口用来生成显示给用户的问候语。最开始设计的时候，这个接口 `GreetV1` 就一个方法 `greet`，它接收 2 个参数，分别是用户的姓名和性别，如代码清单 2-27 所示。

代码清单 2-27 早期版本的 `GreetV1` 接口的定义

```
public interface GreetV1 {
    String greet(String name, String gender) throws GreetException;
}
```

在开发新版本的时候，发现这个接口的设计不太合理，希望把方法的参数改为一个，表示用户名即可，姓名和性别可以通过进一步的查找来完成。另外，也不希望方法抛出受检异常，希望使用更为方便的非受检异常。基于上面的考虑，就有了新的接口定义 `GreetV2`，如代码清单 2-28 所示。

代码清单 2-28 新版本的 GreetV2 接口的定义

```
public interface GreetV2 {
    String greet(String username);
}
```

这个新接口比旧接口更加简单和实用，同时新定义了相关的非受检异常 GreetRuntimeException。以后的代码中都应该使用这个新的接口，同时使用旧接口的代码也要能够继续使用。这就是动态代理可以发挥作用的地方。通过动态代理，可以把实现旧接口 GreetV1 的对象实例转换成可以通过新接口 GreetV2 来调用。这样既保证了对新接口的使用，又使旧接口的实现可以继续存在。这实际上是通过动态代理来实现适配器设计模式的。实现这样的动态代理的关键就在于适配两个接口的 InvocationHandler 的实现，完整的实现如代码清单 2-29 所示。

代码清单 2-29 完成接口适配的 InvocationHandler 接口的实现

```
public class GreetAdapter implements InvocationHandler {
    private GreetV1 greetV1;

    public GreetAdapter(GreetV1 greetV1) {
        this.greetV1 = greetV1;
    }

    public Object invoke(Object proxy, Method method, Object[] args) throws
        Throwable {
        String methodName = method.getName();
        if ("greet".equals(methodName)) {
            String username = (String) args[0];
            String name = findName(username);
            String gender = findGender(username);
            try {
                Method greetMethodV1 = GreetV1.class.getMethod(methodName, new
                    Class<?>[]{String.class, String.class});
                return greetMethodV1.invoke(greetV1, new Object[]{name, gender});
            } catch (InvocationTargetException e) {
                Throwable cause = e.getCause();
                if (cause != null && cause instanceof GreetException) {
                    throw new GreetRuntimeException(cause);
                }
                throw e;
            }
        } else {
            return method.invoke(greetV1, args);
        }
    }

    private String findGender(String username) {
        return Math.random() > 0.5 ? username : null;
    }
}
```

```

    }

    private String findName(String username) {
        return username;
    }
}

```

由于动态代理把 GreetV1 接口的实现对象适配到 GreetV2 接口上，因此需要有一个已有的 GreetV1 接口的对象作为调用的接收者，通过构造方法传递即可实现。在 invoke 方法中，首先通过检查调用的方法的名称来判断是否为 GreetV1 中已有的 greet 方法。这是因为其他方法的调用也会被传入到 invoke 方法中，而这些方法是不需要被处理的。如果调用的是 greet 方法，则说明这次调用需要代理给 GreetV1 接口的实现对象来完成。因为 GreetV1 和 GreetV2 接口中的 greet 方法的参数不匹配，需要先进行转换。在 GreetAdapter 中使用了两个方法来通过传入的用户名查找对应的姓名和性别。接着通过反射 API 获取 GreetV1 接口中的 greet 方法，再把转换之后的参数传入以进行方法调用，最后返回调用的结果。在调用的时候，GreetV1 接口中的 greet 方法可能会抛出受检异常 GreetException，因此需要捕获这个异常，并重新包装成非受检异常 RuntimeGreetException 之后再次抛出。因为 GreetV1 的接口实现在参数 gender 的值为 null 时会抛出 GreetException。这里就通过生成随机数的方式来模拟出错的情况。

对于每一个 GreetV1 接口的实现，都可以通过一个工厂方法转换成可以通过 GreetV2 接口来使用的新对象。工厂方法如代码清单 2-30 所示。

代码清单 2-30 进行对象转换的工厂方法

```

public class GreetFactory {
    public static GreetV2 adaptGreet(GreetV1 greet) {
        GreetAdapter adapter = new GreetAdapter(greet);
        ClassLoader cl = greet.getClass().getClassLoader();
        return (GreetV2) Proxy.newProxyInstance(cl, new Class<?>[] {GreetV2.class},
            adapter);
    }
}

```

在实际的使用中，如果遇到 GreetV1 接口的实现，只需要将调用 GreetFactory 的 adaptGreet 方法转换成 GreetV2 接口，再按照 GreetV2 接口的方式来使用即可。GreetV1 接口可以继续在遗留代码中使用。

2.4 动态语言支持

这节将要介绍的是 Java 7 中的一个重要的新特性。这个新特性的特殊之处在于它是对 Java 虚拟机规范的修改，而不是对 Java 语言规范的修改。从这个角度来说，这个改动会比之前介绍的 Java 7 新特性更加复杂，对 Java 平台的影响也更加深远。这个新特性

增强了Java虚拟机中对方法调用的支持。虽然这个特性的直接受益者是Java平台上的动态语言的编译器，但是它对一般应用程序也有重大的影响，最直接的就是提供了比反射API更加强大的动态方法调用能力。本节将会详细介绍这个Java 7的重要新特性，所涉及的内容包括Java虚拟机中的新的方法调用指令invokedynamic，以及Java SE 7核心库中的java.lang.invoke包。这一个新特性对应的修改内容包含在JSR 292（Supporting Dynamically Typed Languages on the Java™ Platform）中。

2.4.1 Java语言与Java虚拟机

在介绍新特性之前，首先需要简单介绍一下Java虚拟机。Java虚拟机本身并不知道Java语言的存在，它只理解Java字节代码格式，即class文件。一个class文件包含了Java虚拟机规范中所定义的指令和符号表。Java虚拟机只是负责执行class文件中包含的指令。而这些class文件可以由Java语言的编译器生成，也可以由其他编程语言的编译器生成，还可以通过工具来手动生成。只要class文件的格式是符合规范的，Java虚拟机就能正确执行它。

Java虚拟机的存在实际上是在底层操作系统和应用程序之间添加了一个新的抽象层次。对于一种编程语言来说，可以选择直接把源代码编译成目标平台上的机器代码。这种做法无疑是效率最高的。但是所带来的问题是生成的二进制内容无法兼容不同平台，且实现的复杂度也很高。如果存在某种虚拟机，事情就会变得简单很多。首先虚拟机提供了一个抽象层次，屏蔽了底层系统的差别，其所暴露的接口是规范而统一的，可以真正实现“编写一次，到处运行”的目标。另外，虚拟机会提供很多编程语言所需要的运行时支持能力，包括内存管理、安全机制、并发控制、标准库和工具等。最后，使用一个已有的虚拟机作为运行平台，使编程语言的使用者可以复用与这个虚拟机平台相关的已有资产，包括相关的工具、集成开发环境和开发经验等。这有利于编程语言本身的推广和普及。

正因为如此，已经有非常多的编程语言支持把Java虚拟机作为目标运行平台。也就是说，这些语言的编译器支持把源代码编译成Java字节代码，其中比较主流的语言包括Java、Scala、JRuby、Groovy、Jython、PHP、C#、JavaScript、Tcl和Lisp等。这其中最主流的还是Java语言本身。

前面虽然说到Java虚拟机并不关心字节代码是由哪种编程语言产生的，但是Java语言作为Java虚拟机上的第一个也是最重要的一种语言，它对Java虚拟机规范本身所产生的影响是最大的。事实上，Java虚拟机上的很多特性，是为了配合Java语言而产生的。Java语言作为一门静态类型的编程语言，也影响了Java虚拟机本身动态性。随着越来越多的动态类型编程语言将Java虚拟机作为运行平台，而Java虚拟机本身又缺乏对动态性的支持，所以会对这些动态类型语言的实现产生比较大的阻碍。当然，动态类型语言的实现者总是能找到方法绕开Java虚拟机中的各种限制，这样做所带来的后果就

是复杂度比较高，性能也会受到影响。Java 7 中的动态语言支持，就是在 Java 虚拟机规范这个层次上进行修改，使 Java 虚拟机对于动态类型编程语言来说更加友好，性能也更好。

JSR 292 中包含的相关改动涉及应用程序运行中最常见的方法调用。具体来说主要是两个部分，一个是 Java 标准库中新的方法调用 API，另外一个是 Java 虚拟机规范中新的 `invokedynamic` 指令。在下面的内容中，先介绍相关的 Java API，因为对一般的开发者来说，这个是用得最多的。随后也会对 `invokedynamic` 指令进行具体的介绍。首先从方法句柄开始介绍。

2.4.2 方法句柄

方法句柄（method handle）是 JSR 292 中引入的一个重要概念，它是对 Java 中方法、构造方法和域的一个强类型的可执行的引用。这也是句柄这个词的含义所在。通过方法句柄可以直接调用该句柄所引用的底层方法。从作用上来说，方法句柄的作用类似于 2.2 节中提到的反射 API 中的 `Method` 类，但是方法句柄的功能更强大、使用更灵活、性能也更好。实际上，方法句柄和反射 API 也是可以协同使用的，下面会具体介绍。在 Java 标准库中，方法句柄是由 `java.lang.invoke.MethodHandle` 类来表示的。

1. 方法句柄的类型

对于一个方法句柄来说，它的类型完全由它的参数类型和返回值类型来确定，而与它所引用的底层方法的名称和所在的类没有关系。比如引用 `String` 类的 `length` 方法和 `Integer` 类的 `intValue` 方法的方法句柄的类型就是一样的，因为这两个方法都没有参数，而且返回值类型都是 `int`。

在得到一个方法句柄，即 `MethodHandle` 类的对象之后，可以通过其 `type` 方法来查看其类型。该方法的返回值是一个 `java.lang.invoke.MethodType` 类的对象。`MethodType` 类的所有对象实例都是不可变的，类似于 `String` 类。所有对 `MethodType` 类对象的修改，都会产生一个新的 `MethodType` 类对象。两个 `MethodType` 类对象是否相等，只取决于它们所包含的参数类型和返回值类型是否完全一致。

`MethodType` 类的对象实例只能通过 `MethodType` 类中的静态工厂方法来创建。这样的工厂方法有三类。第一类是通过指定参数和返回值的类型来创建 `MethodType`，这主要是使用 `methodType` 方法的多种重载形式。使用这些方法的时候，至少需要指定返回值类型，而参数类型则可以是 0 到多个。返回值类型总是出现在 `methodType` 方法参数列表的第一个，后面紧接着的是 0 到多个参数的类型。类型都是由 `Class` 类的对象来指定的。如果返回值类型是 `void`，可以用 `void.class` 或 `java.lang.Void.class` 来声明。代码清单 2-31 中给出了使用 `methodType` 方法的几个示例。每个 `MethodType` 声明上以注释的方式给出了与之相匹配的 `String` 类中的一个方法。这里值得一提的是，最后一个 `methodType` 方法调用中使用了另外一个 `MethodType` 的参数类型作为当前 `MethodType` 类对象的参数类型。

代码清单 2-31 MethodType 类中的 methodType 方法的使用示例

```

public void generateMethodTypes() {
    //String.length()
    MethodType mt1 = MethodType.methodType(int.class);
    //String.concat(String str)
    MethodType mt2 = MethodType.methodType(String.class, String.class);
    //String.getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)
    MethodType mt3 = MethodType.methodType(void.class, int.class, int.class,
        char[].class, int.class);
    //String.startsWith(String prefix)
    MethodType mt4 = MethodType.methodType(boolean.class, mt2);
}

```

除了显式地指定返回值和参数的类型之外，还可以生成通用的 MethodType 类型，即返回值和所有参数的类型都是 Object 类。这是通过静态工厂方法 genericMethodType 来创建的。方法 genericMethodType 有两种重载形式：第一种形式只需要指明方法类型中包含的 Object 类型的参数个数即可，而第二种形式可以提供一个额外的参数来说明是否在参数列表的后面添加一个 Object[] 类型的参数。在代码清单 2-32 中，mt1 有 3 个类型为 Object 的参数，而 mt2 有 2 个类型为 Object 的参数和后面的 Object[] 类型参数。

代码清单 2-32 生成通用 MethodType 类型的示例

```

public void generateGenericMethodTypes() {
    MethodType mt1 = MethodType.genericMethodType(3);
    MethodType mt2 = MethodType.genericMethodType(2, true);
}

```

最后介绍的一个工厂方法是比较复杂的 fromMethodDescriptorString。这个方法允许开发人员指定方法类型在字节代码中的表示形式作为创建 MethodType 时的参数。这个方法的复杂之处在于字节代码中的方法类型格式不是很好理解。比如代码清单 2-31 中的 String.getChars 方法的类型在字节代码中的表示形式是 “(II[CI)V”。不过这种格式比逐个声明返回值和参数类型的做法更加简洁，适合于对 Java 字节代码格式比较熟悉的开发人员。在代码清单 2-33 中，“(Ljava/lang/String;)Ljava/lang/String;” 所表示的方法类型是返回值和参数类型都是 java.lang.String，相当于使用 MethodType.methodType(String.class, String.class)。

代码清单 2-33 使用方法类型在字节代码中的表示形式来创建 MethodType

```

public void generateMethodTypesFromDescriptor() {
    ClassLoader cl = this.getClass().getClassLoader();
    String descriptor = "(Ljava/lang/String;)Ljava/lang/String;";
    MethodType mt1 = MethodType.fromMethodDescriptorString(descriptor, cl);
}

```

在使用 fromMethodDescriptorString 方法的时候，需要指定一个类加载器。该类加载器用来加载方法类型表达式中出现的 Java 类。如果不指定，默认使用系统类加载器。

在通过工厂方法创建出 MethodType 类的对象实例之后，可以对其进行进一步修改。这些修改都围绕返回值和参数类型展开。所有这些修改方法都返回另外一个新的 MethodType 对象。代码清单 2-34 给出了对 MethodType 中的返回值和参数类型进行修改的示例代码。基本的修改操作包括改变返回值类型、添加和插入新参数、删除已有参数和修改已有参数的类型等。在每个修改方法上以注释形式给出修改之后的类型，括号里面是参数类型列表，外面是返回值类型。

代码清单 2-34 对 MethodType 中的返回值和参数类型进行修改的示例

```
public void changeMethodType() {
    // (int, int) String
    MethodType mt = MethodType.methodType(String.class, int.class, int.class);
    // (int, int, float) String
    mt = mt.appendParameterTypes(float.class);
    // (int, double, long, int, float) String
    mt = mt.insertParameterTypes(1, double.class, long.class);
    // (int, double, int, float) String
    mt = mt.dropParameterTypes(2, 3);
    // (int, double, String, float) String
    mt = mt.changeParameterType(2, String.class);
    // (int, double, String, float) void
    mt = mt.changeReturnType(void.class);
}
```

除了上面这几个精确修改返回值和参数的类型的方法之外，MethodType 还有几个可以一次性对返回值和所有参数的类型进行处理的方法。代码清单 2-35 给出了这几个方法的使用示例，其中 wrap 和 unwrap 用来在基本类型及其包装类型之间进行转换，generic 方法把所有返回值和参数类型都变成 Object 类型，而 erase 只把引用类型变成 Object，并不处理基本类型。修改之后的方法类型同样以注释的形式给出。

代码清单 2-35 一次性修改 MethodType 中的返回值和所有参数的类型的示例

```
public void wrapAndGeneric() {
    // (int, double) Integer
    MethodType mt = MethodType.methodType(Integer.class, int.class, double.class);
    // (Integer, Double) Integer
    MethodType wrapped = mt.wrap();
    // (int, double) int
    MethodType unwrapped = mt.unwrap();
    // (Object, Object) Object
    MethodType generic = mt.generic();
    // (int, double) Object
    MethodType erased = mt.erase();
}
```

由于每个对 MethodType 对象进行修改的方法的返回值都是一个新的 MethodType 对象，可以很容易地通过方法级联来简化代码。

2. 方法句柄的调用

在获取到了一个方法句柄之后，最直接的使用方法就是调用它所引用的底层方法。在这点上，方法句柄的使用类似于反射 API 中的 Method 类。但是方法句柄在调用时所提供的灵活性是 Method 类中的 invoke 方法所不能比的。

最直接的调用一个方法句柄的做法是通过 invokeExact 方法实现的。这个方法与直接调用底层方法是完全一样的。invokeExact 方法的参数依次是作为方法接收者的对象和调用时候的实际参数列表。比如在代码清单 2-36 中，先获取 String 类中 substring 的方法句柄，再通过 invokeExact 来进行调用。这种调用方式就相当于直接调用 "Hello World".substring(1, 3)。关于方法句柄的获取，下一节会具体介绍。

代码清单 2-36 使用 invokeExact 方法调用方法句柄

```
public void invokeExact() throws Throwable {
    MethodHandles.Lookup lookup = MethodHandles.lookup();
    MethodType type = MethodType.methodType(String.class, int.class, int.class);
    MethodHandle mh = lookup.findVirtual(String.class, "substring", type);
    String str = (String) mh.invokeExact("Hello World", 1, 3);
    System.out.println(str);
}
```

在这里强调一下静态方法和一般方法之间的区别。静态方法在调用时是不需要指定方法的接收对象的，而一般的方法则是需要的。如果方法句柄 mh 所引用的是 java.lang.Math 类中的静态方法 min，那么直接通过 mh.invokeExact(3, 4) 就可以调用该方法。

注意 invokeExact 方法在调用的时候要求严格的类型匹配，方法的返回值类型也是在考虑范围之内的。代码清单 2-36 中的方法句柄所引用的 substring 方法的返回值类型是 String，因此在使用 invokeExact 方法进行调用时，需要在前面加上强制类型转换，以声明返回值的类型。如果去掉这个类型转换，而直接赋值给一个 Object 类型的变量，在调用的时候会抛出异常，因为 invokeExact 会认为方法的返回值类型是 Object。去掉类型转换但是不进行赋值操作也是错误的，因为 invokeExact 会认为方法的返回值类型是 void，也不同于方法句柄要求的 String 类型的返回值。

与 invokeExact 所要求的类型精确匹配不同的是，invoke 方法允许更加松散的调用方式。它会尝试在调用的时候进行返回值和参数类型的转换工作。这是通过 MethodHandle 类的 asType 方法来完成的。asType 方法的作用是把当前的方法句柄适配到新的 MethodType 上，并产生一个新的方法句柄。当方法句柄在调用时的类型与其声明的类型完全一致的时候，调用 invoke 等同于调用 invokeExact；否则，invoke 会先调用 asType 方法来尝试适配到调用时的类型。如果适配成功，调用可以继续；否则会抛出

相关的异常。这种灵活的适配机制，使 `invoke` 方法成为在绝大多数情况下都应该使用的方法句柄调用方式。

进行类型适配的基本规则是比对返回值类型和每个参数的类型是否都可以相互匹配。只要返回值类型或某个参数的类型无法完成匹配，那么整个适配过程就是失败的。从待转换的源类型 S 到目标类型 T 匹配成功的基本原则如下：

- 1) 可以通过 Java 的类型转换来完成，一般是从子类转换成父类，接口的实现类转换成接口，比如从 `String` 类转换到 `Object` 类。
- 2) 可以通过基本类型的转换来完成，只能进行类型范围的扩大，比如从 `int` 类型转换到 `long` 类型。
- 3) 可以通过基本类型的自动装箱和拆箱机制来完成，比如从 `int` 类型到 `Integer` 类型。
- 4) 如果 S 有返回值类型，而 T 的返回值是 `void`，S 的返回值会被丢弃。
- 5) 如果 S 的返回值是 `void`，而 T 的返回值是引用类型，T 的返回值会是 `null`。
- 6) 如果 S 的返回值是 `void`，而 T 的返回值是基本类型，T 的返回值会是 0。

满足上面规则时进行两个方法类型之间的转换是会成功的。对于 `invoke` 方法的具体使用，只需要把代码清单 2-36 中的 `invokeExact` 方法换成 `invoke` 即可，并不需要做太多的介绍。

最后一种调用方式是使用 `invokeWithArguments`。该方法在调用时可以指定任意多个 `Object` 类型的参数。完整的调用方式是首先根据传入的实际参数的个数，通过 `MethodType` 的 `genericMethodType` 方法得到一个返回值和参数类型都是 `Object` 的新方法类型。再把原始的方法句柄通过 `asType` 转换后得到一个新的方法句柄。最后通过新方法句柄的 `invokeExact` 方法来完成调用。这个方法相对于 `invokeExact` 和 `invoke` 的优势在于，它可以通过 Java 反射 API 被正常获取和调用，而 `invokeExact` 和 `invoke` 不可以这样。它可以作为反射 API 和方法句柄之间的桥梁。

3. 参数长度可变的方法句柄

在方法句柄中，所引用的底层方法中包含长度可变的参数是一种比较特殊的情况。虽然最后一个长度可变的参数实际上是一个数组，但是仍然可以简化方法调用时的语法。对于这种特殊的情况，方法句柄也提供了相关的处理能力，主要是一些转换的方法，允许在可变长度的参数和数组类型的参数之间互相转换，以方便开发人员根据需求选择最适合的调用语法。

`MethodHandle` 中第一个与长度可变参数相关的方法是 `asVarargsCollector`。它的作用是把原始的方法句柄中的最后一个数组类型的参数转换成对应类型的可变长度参数。如代码清单 2-37 所示，方法 `normalMethod` 的最后一个参数是 `int` 类型的数组，引用它的方法句柄在通过 `asVarargsCollector` 方法转换之后，得到的新方法句柄在调用时就可以使用长度可变参数的语法格式，而不需要使用原始的数组形式。在实际的调用中，`int` 类型的参数 3、4 和 5 组成的数组被传入到了 `normalMethod` 的参数 `args` 中。

代码清单 2-37 asVarargsCollector 方法的使用示例

```
public void normalMethod(String arg1, int arg2, int[] arg3) {
}

public void asVarargsCollector() throws Throwable {
    MethodHandles.Lookup lookup = MethodHandles.lookup();
    MethodHandle mh = lookup.findVirtual(Varargs.class, "normalMethod",
        MethodType.methodType(void.class, String.class, int.class, int[].class));
    mh = mh.asVarargsCollector(int[].class);
    mh.invoke(this, "Hello", 2, 3, 4, 5);
}
```

第二个方法 asCollector 的作用与 asVarargsCollector 类似，不同的是该方法只会把指定数量的参数收集到原始方法句柄所对应的底层方法的数组类型参数中，而不像 asVarargsCollector 那样可以收集任意数量的参数。如代码清单 2-38 所示，还是以引用 normalMethod 的方法句柄为例，asCollector 方法调用时的指定参数为 2，即只有 2 个参数会被收集到整数类型数组中。在实际的调用中，int 类型的参数 3 和 4 组成的数组被传入到了 normalMethod 的参数 args 中。

代码清单 2-38 asCollector 方法的使用示例

```
public void asCollector() throws Throwable {
    MethodHandles.Lookup lookup = MethodHandles.lookup();
    MethodHandle mh = lookup.findVirtual(Varargs.class, "normalMethod",
        MethodType.methodType(void.class, String.class, int.class, int[].class));
    mh = mh.asCollector(int[].class, 2);
    mh.invoke(this, "Hello", 2, 3, 4);
}
```

上面的两个方法把数组类型参数转换为长度可变的参数，自然还有与之对应的执行反方向转换的方法。代码清单 2-39 给出的 asSpreader 方法就把长度可变的参数转换成数组类型的参数。转换之后的新方法句柄在调用时使用数组作为参数，而数组中的元素会被按顺序分配给原始方法句柄中的各个参数。在实际的调用中，toBeSpread 方法所接收到的参数 arg2、arg3 和 arg4 的值分别是 3、4 和 5。

代码清单 2-39 asSpreader 方法的使用示例

```
public void toBeSpreaded(String arg1, int arg2, int arg3, int arg4) {
}

public void asSpreader() throws Throwable {
    MethodHandles.Lookup lookup = MethodHandles.lookup();
    MethodHandle mh = lookup.findVirtual(Varargs.class, "toBeSpreaded",
        MethodType.methodType(void.class, String.class, int.class, int.class, int.class));
    mh = mh.asSpreader(int[].class, 3);
```

```

        mh.invoke(this, "Hello", new int[]{3, 4, 5});
    }

```

最后一个方法 `asFixedArity` 是把参数长度可变的方法转换成参数长度不变的方法。经过这样的转换之后，最后一个长度可变的参数实际上就变成了对应的数组类型。在调用方法句柄的时候，就只能使用数组来进行参数传递。如代码清单 2-40 所示，`asFixedArity` 会把引用参数长度可变方法 `varargsMethod` 的原始方法句柄转换成固定长度参数的方法句柄。

代码清单 2-40 `asFixedArity` 方法的使用示例

```

public void varargsMethod(String arg1, int... args) {
}

public void asFixedArity() throws Throwable {
    MethodHandles.Lookup lookup = MethodHandles.lookup();
    MethodHandle mh = lookup.findVirtual(Varargs.class, "varargsMethod",
        MethodType.methodType(void.class, String.class, int[].class));
    mh = mh.asFixedArity();
    mh.invoke(this, "Hello", new int[]{2, 4});
}

```

4. 参数绑定

在前面介绍过，如果方法句柄在调用时引用的底层方法不是静态的，调用的第一个参数应该是该方法调用的接收者。这个参数的值一般在调用时指定，也可以事先进行绑定。通过 `MethodHandle` 的 `bindTo` 方法可以预先绑定底层方法的调用接收者，而在实际调用的时候，只需要传入实际参数即可，不需要再指定方法的接收者。代码清单 2-41 给出了对引用 `String` 类的 `length` 方法的方法句柄的两种调用方式：第一种没有进行绑定，调用时需要传入 `length` 方法的接收者；第二种方法预先绑定了一个 `String` 类的对象，因此调用时不需要再指定。

代码清单 2-41 参数绑定的基本用法

```

public void bindTo() throws Throwable {
    MethodHandles.Lookup lookup = MethodHandles.lookup();
    MethodHandle mh = lookup.findVirtual(String.class, "length", MethodType.
        methodType(int.class));
    int len = (int) mh.invoke("Hello"); // 值为 5
    mh = mh.bindTo("Hello World");
    len = (int) mh.invoke(); // 值为 11
}

```

这种预先绑定参数的方式的灵活性在于它允许开发人员只公开某个方法，而不公开该方法所在的对象。开发人员只需要找到对应的方法句柄，并把适合的对象绑定到方法

句柄上，客户代码就可以只获取到方法本身，而不会知道包含此方法的对象。绑定之后的方法句柄本身就可以在任何地方直接运行。

实际上，MethodHandle 的 bindTo 方法只是绑定方法句柄的第一个参数而已，并不要求这个参数一定表示方法调用的接收者。对于一个 MethodHandle，可以多次使用 bindTo 方法来为其中的多个参数绑定值。代码清单 2-42 给出了多次绑定的一个示例。方法句柄所引用的底层方法是 String 类中的 indexOf 方法，同时为方法句柄的前两个参数分别绑定了具体的值。

代码清单 2-42 多次参数绑定的示例

```
public void multipleBindTo() throws Throwable {
    MethodHandles.Lookup lookup = MethodHandles.lookup();
    MethodHandle mh = lookup.findVirtual(String.class, "indexOf",
        MethodType.methodType(int.class, String.class, int.class));
    mh = mh.bindTo("Hello").bindTo("l");
    System.out.println(mh.invoke(2)); // 值为 2
}
```

需要注意的是，在进行参数绑定的时候，只能对引用类型的参数进行绑定。无法为 int 和 float 这样的基本类型绑定值。对于包含基本类型参数的方法句柄，可以先使用 wrap 方法把方法类型中的基本类型转换成对应的包装类，再通过方法句柄的 asType 将其转换成新的句柄。转换之后的新句柄就可以通过 bindTo 来进行绑定，如代码清单 2-43 所示。

代码清单 2-43 基本类型参数的绑定方式

```
MethodHandle mh = lookup.findVirtual(String.class, "substring", MethodType.
    methodType(String.class, int.class, int.class));
mh = mh.asType(mh.type().wrap());
mh = mh.bindTo("Hello World").bindTo(3);
System.out.println(mh.invoke(5)); // 值为 "lo"
```

5. 获取方法句柄

获取方法句柄最直接的做法是从一个类中已有的方法中转换而来，得到的方法句柄直接引用这个底层方法。在之前的示例中都是通过这种方式来获取方法句柄的。方法句柄可以按照与反射 API 类似的方法，从已有的类中根据一定的条件进行查找。与反射 API 不同的是，方法句柄并不区分构造方法、方法和域，而是统一转换成 MethodHandle 对象。对于域来说，获取到的是用来获取和设置该域的值的方法句柄。

方法句柄的查找是通过 java.lang.invoke.MethodHandles.Lookup 类来完成的。在查找之前，需要通过调用 MethodHandles.lookup 方法获取到一个 MethodHandles.Lookup 类的对象。MethodHandles.Lookup 类提供了一些方法以根据不同的条件进行查找。代码清单 2-44 以 String 类为例说明了查找构造方法和一般方法的示例。方法 findConstructor

用来查找类中的构造方法，需要指定返回值和参数类型，即 MethodType 对象。而 findVirtual 和 findStatic 则用来查找一般方法和静态方法，除了表示方法的返回值和参数类型的 MethodType 对象之外，还需要指定方法的名称。

代码清单 2-44 查找构造方法、一般方法和静态方法的方法句柄的示例

```
public void lookupMethod() throws NoSuchMethodException, IllegalAccessException {
    MethodHandles.Lookup lookup = MethodHandles.lookup();
    // 构造方法
    lookup.findConstructor(String.class, MethodType.methodType(void.class, byte[].class));
    //String.substring
    lookup.findVirtual(String.class, "substring", MethodType.methodType(String.class, int.class, int.class));
    //String.format
    lookup.findStatic(String.class, "format", MethodType.methodType(String.class, String.class, Object[].class));
}
```

除了上面 3 种类型的方法之外，还有一个 findSpecial 方法用来查找类中的特殊方法，主要是类中的私有方法。代码清单 2-45 给出了 findSpecial 的使用示例，MethodHandleLookup 是 lookupSpecial 方法所在的类，而 privateMethod 是该类中的一个私有方法。由于访问的是类的私有方法，从访问控制的角度出发，进行方法查找的类需要具备访问私有方法的权限。

代码清单 2-45 查找类中特殊方法的方法句柄的示例

```
public MethodHandle lookupSpecial() throws NoSuchMethodException,
    IllegalAccessException, Throwable {
    MethodHandles.Lookup lookup = MethodHandles.lookup();
    MethodHandle mh = lookup.findSpecial(MethodHandleLookup.class,
        "privateMethod", MethodType.methodType(void.class), MethodHandleLookup.class);
    return mh;
}
```

从上面的代码中可以看到，findSpecial 方法比之前的 findVirtual 和 findStatic 等方法多了一个参数。这个额外的参数用来指定私有方法被调用时所使用的类。提供这个类的原因是为了满足对私有方法的访问控制的要求。当方法句柄被调用时，指定的调用类必须具备访问私有方法的权限，否则会出现无法访问的错误。

除了类中本来就存在的方法之外，对域的处理也是通过相应的获取和设置域的值的方法句柄来完成的。代码清单 2-46 说明了如何查找到类中的静态域和一般域所对应的获取和设置的方法句柄。在查找的时候只需要提供域所在的类的 Class 对象、域的名称和类型即可。

代码清单 2-46 查找类中的静态域和一般域对应的获取和设置的方法句柄的示例

```
public void lookupFieldAccessor() throws NoSuchFieldException, IllegalAccessException{
    MethodHandles.Lookup lookup = MethodHandles.lookup();
    lookup.findGetter(Sample.class, "name", String.class);
    lookup.findSetter(Sample.class, "name", String.class);
    lookup.findStaticGetter(Sample.class, "value", int.class);
    lookup.findStaticSetter(Sample.class, "value", int.class);
}
```

对于静态域来说，调用其对应的获取和设置值的方法句柄时，并不需要提供调用的接收者对象作为参数。而对于一般域来说，该对象在调用时是必需的。

除了直接在某个类中进行查找之外，还可以从通过反射 API 得到的 Constructor、Field 和 Method 等对象中获得方法句柄。如代码清单 2-47 所示，首先通过反射 API 得到表示构造方法的 Constructor 对象，再通过 unreflectConstructor 方法就可以得到其对应的一个方法句柄；而通过 unreflect 方法可以将 Method 类对象转换成方法句柄。对于私有方法，则需要使用 unreflectSpecial 来进行转换，同样也需要提供一个作用与 findSpecial 中参数相同的额外参数；对于 Field 类的对象来说，通过 unreflectGetter 和 unreflectSetter 就可以得到获取和设置其值的方法句柄。

代码清单 2-47 通过反射 API 获取方法句柄的示例

```
public void unreflect() throws Throwable {
    MethodHandles.Lookup lookup = MethodHandles.lookup();
    Constructor constructor = String.class.getConstructor(byte[].class);
    lookup.unreflectConstructor(constructor);
    Method method = String.class.getMethod("substring", int.class, int.class);
    lookup.unreflect(method);

    Method privateMethod = ReflectMethodHandle.class.getDeclaredMethod("privateMethod");
    lookup.unreflectSpecial(privateMethod, ReflectMethodHandle.class);

    Field field = ReflectMethodHandle.class.getField("name");
    lookup.unreflectGetter(field);
    lookup.unreflectSetter(field);
}
```

除了通过在 Java 类中进行查找来获取方法句柄外，还可以通过 java.lang.invoke.MethodHandles 中提供的一些静态工厂方法来创建一些通用的方法句柄。

第一个方法是用来对数组进行操作的，即得到可以用来获取和设置数组中元素的值的方法句柄。这些工厂方法的作用等价于 2.2.4 节介绍的反射 API 中的 java.lang.reflect.Array 类中的静态方法。如代码清单 2-48 所示，MethodHandles 的 arrayElementGetter 和 arrayElementSetter 方法分别用来得到获取和设置数组元素的值的方法句柄。调用这些方

法句柄就可以对数组进行操作。

代码清单 2-48 获取和设置数组中元素的值的方法句柄的使用示例

```
public void arrayHandles() throws Throwable {
    int[] array = new int[] {1, 2, 3, 4, 5};
    MethodHandle setter = MethodHandles.arrayElementSetter(int[].class);
    setter.invoke(array, 3, 6);
    MethodHandle getter = MethodHandles.arrayElementGetter(int[].class);
    int value = (int) getter.invoke(array, 3); // 值为 6
}
```

MethodHandles 中的静态方法 identity 的作用是通过它所生成的方法句柄，在每次调用的时候，总是返回其输入参数的值。如代码清单 2-49 所示，在使用 identity 方法的时候只需要传入方法句柄的唯一参数的类型即可，该方法句柄的返回值类型和参数类型是相同的。

代码清单 2-49 MethodHandles 类的 identity 方法的使用示例

```
public void identity() throws Throwable {
    MethodHandle mh = MethodHandles.identity(String.class);
    String value = (String) mh.invoke("Hello"); // 值为 "Hello"
}
```

而方法 constant 的作用则更加简单，在生成的时候指定一个常量值，以后这个方法句柄被调用的时候，总是返回这个常量值，在调用时也不需要提供任何参数。这个方法提供了一种把一个常量值转换成方法句柄的方式，如下面的代码所示。在调用 constant 方法的时候，只需要提供常量的类型和值即可。

代码清单 2-50 MethodHandles 类的 constant 方法的使用示例

```
public void constant() throws Throwable {
    MethodHandle mh = MethodHandles.constant(String.class, "Hello");
    String value = (String) mh.invoke(); // 值为 "Hello"
}
```

MethodHandles 类中的 identity 方法和 constant 方法的作用类似于在开发中用到的“空对象（Null object）”模式的应用。在使用方法句柄的某些场合中，如果没有合适的方法句柄对象，可能不允许直接用 null 来替换，这个时候可以通过这两个方法来生成简单无害的方法句柄对象作为替代。

6. 方法句柄变换

方法句柄的强大之处在于可以对它进行各种不同的变换操作。这些变换操作包括对方法句柄的返回值和参数的处理等，同时这些单个的变换操作可以组合起来，形成复杂的变换过程。所有的这些变换方法都是 MethodHandles 类中的静态方法。这些方法一般

接受一个已有的方法句柄对象作为变换的来源，而方法的返回值则是变换操作之后得到的新方法句柄。下面的内容中经常出现的“原始方法句柄”表示的是变换之前的方法句柄，而“新方法句柄”则表示变换之后的方法句柄。

首先介绍对参数进行处理的变换方法。在调用变换之后的新方法句柄时，调用时的参数值会经过一定的变换操作之后，再传递给原始的方法句柄来完成具体的执行。

第一个方法 dropArguments 可以在一个方法句柄的参数中添加一些无用的参数。这些参数虽然在实际调用时不会被使用，但是它们可以使变换之后的方法句柄的参数类型格式符合某些所需的特定模式。这也是这种变换方式的主要应用场景。

如代码清单 2-51 所示，原始的方法句柄 mhOld 引用的是 String 类中的 substring 方法，其类型是 String 类的返回值加上两个 int 类型的参数。在调用 dropArguments 方法的时候，第一个参数表示待变换的方法句柄，第二个参数指定的是要添加的新参数类型在原始参数列表中的起始位置，其后的多个参数类型将被添加到参数列表中。新的方法句柄 mhNew 的参数类型变为 float、String、String、int 和 int，而在实际调用时，前面两个参数的值会被忽略掉。可以把这些多余的参数理解成特殊调用模式所需要的占位符。

代码清单 2-51 dropArguments 方法的使用示例

```
public void dropArguments() throws Throwable {
    MethodHandles.Lookup lookup = MethodHandles.lookup();
    MethodType type = MethodType.methodType(String.class, int.class, int.class);
    MethodHandle mhOld = lookup.findVirtual(String.class, "substring", type);
    String value = (String) mhOld.invoke("Hello", 2, 3);
    MethodHandle mhNew = MethodHandles.dropArguments(mhOld, 0, float.class,
        String.class);
    value = (String) mhNew.invoke(0.5f, "Ignore", "Hello", 2, 3);
}
```

第二个方法 insertArguments 的作用与本小节前面提到的 MethodHandle 的 bindTo 方法类似，但是此方法的功能更加强大。这个方法可以同时为方法句柄中的多个参数预先绑定具体的值。在得到的新方法句柄中，已经绑定了具体值的参数不再需要提供，也不会出现在参数列表中。

在代码清单 2-52 中，方法句柄 mhOld 所表示的底层方法是 String 类中的 concat 方法。在调用 insertArguments 方法的时候，与上面的 dropArguments 方法类似，从第二个参数所指定的参数列表中的位置开始，用其后的可变长度的参数的值作为预设值，分别绑定到对应的参数上。在这里把 mhOld 的第二个参数的值预设成了固定值“--”，其作用是在调用新方法句柄时，只需要传入一个参数即可，相当于总是与“--”进行字符串连接操作，即使用“--”作为后缀。由于有一个参数被预先设置了值，因此 mhNew 在调用时只需要一个参数即可。如果预先绑定的是方法句柄 mhOld 的第一个参数，那就相当于用字符串“--”来连接各种不同的字符串，即为字符串添加“--”作为前缀。如果

`insertArguments` 方法调用时指定了多个绑定值，会按照第二个参数指定的起始位置，依次进行绑定。

代码清单 2-52 `insertArguments` 方法的使用示例

```
public void insertArguments() throws Throwable {
    MethodHandles.Lookup lookup = MethodHandles.lookup();
    MethodType type = MethodType.methodType(String.class, String.class);
    MethodHandle mhOld = lookup.findVirtual(String.class, "concat", type);
    String value = (String) mhOld.invoke("Hello", "World");
    MethodHandle mhNew = MethodHandles.insertArguments(mhOld, 1, "--");
    value = (String) mhNew.invoke("Hello"); // 值为 "Hello--"
}
```

第三个方法 `filterArguments` 的作用是可以对方法句柄调用时的参数进行预处理，再把预处理的结果作为实际调用时的参数。预处理的过程是通过其他的方法句柄来完成的。可以对一个或多个参数指定用来进行处理的方法句柄。代码清单 2-53 给出了 `filterArguments` 方法的使用示例。要执行的原始方法句柄所引用的是 `Math` 类中的 `max` 方法，而在实际调用时传入的却是两个字符串类型的参数。中间的参数预处理是通过方法句柄 `mhGetLength` 来完成的，该方法句柄的作用是获得字符串的长度。这样就可以把字符串类型的参数转换成原始方法句柄所需要的整数类型。完成预处理之后，将处理的结果交给原始方法句柄来完成调用。

代码清单 2-53 `filterArguments` 方法的使用示例

```
public void filterArguments() throws Throwable {
    MethodHandles.Lookup lookup = MethodHandles.lookup();
    MethodType type = MethodType.methodType(int.class, int.class, int.class);
    MethodHandle mhGetLength = lookup.findVirtual(String.class, "length",
        MethodType.methodType(int.class));
    MethodHandle mhTarget = lookup.findStatic(Math.class, "max", type);
    MethodHandle mhNew = MethodHandles.filterArguments(mhTarget, 0, mhGetLength,
        mhGetLength);
    int value = (int) mhNew.invoke("Hello", "New World"); // 值为 9
}
```

在使用 `filterArguments` 的时候，第二个参数和后面的方法句柄参数是配合起来使用的。第二个参数指定的是进行预处理的方法句柄需要处理的参数在参数列表中的起始位置。紧跟在后面的是一系列对应的完成参数预处理的方法句柄。方法句柄与它要处理的参数是一一对应的。如果希望跳过某些参数不进行处理，可以使用 `null` 作为方法句柄的值。在进行预处理的时候，要注意预处理方法句柄和原始方法句柄之间的类型匹配。如果预处理方法句柄用于对某个参数进行处理，那么该方法句柄只能有一个参数，而且参数的类型必须匹配所要处理的参数的类型；其返回值类型需要匹配原始方法句柄中对应的参数类型。只有类型匹配，才能用方法句柄对实际传入的参数进行预处

理，再把预处理的结果作为原始方法句柄调用时的参数来使用。

第四个方法 foldArguments 的作用与 filterArguments 很类似，都是用来对参数进行预处理的。不同之处在于，foldArguments 对参数进行预处理之后的结果，不是替换掉原始的参数值，而是添加到原始参数列表的前面，作为一个新的参数。当然，如果参数预处理的返回值是 void，则不会添加新的参数。另外，参数预处理是由一个方法句柄完成的，而不是像 filterArguments 那样可以由多个方法句柄来完成。这个方法句柄会负责处理根据它的类型确定的所有可用参数。下面先看一下具体的使用示例。代码清单 2-54 中原始的方法句柄引用的是静态方法 targetMethod，而用来对参数进行预处理的方法句柄 mhCombiner 引用的是 Math 类中的 max 方法。变换之后的新方法句柄 mhResult 在被调用时，两个参数 3 和 4 首先被传递给句柄 mhCombiner 所引用的 Math.max 方法，返回值是 4。这个返回值被添加到原始调用参数列表的前面，即得到新的参数列表 4、3、4。这个新的参数列表会在调用时被传递给原始方法句柄 mhTarget 所引用的 targetMethod 方法。

代码清单 2-54 foldArguments 方法的使用示例

```
public static int targetMethod(int arg1, int arg2, int arg3) {
    return arg1;
}

public void foldArguments() throws Throwable {
    MethodHandles.Lookup lookup = MethodHandles.lookup();
    MethodType typeCombiner = MethodType.methodType(int.class, int.class, int.class);
    MethodHandle mhCombiner = lookup.findStatic(Math.class, "max", typeCombiner);
    MethodType typeTarget = MethodType.methodType(int.class, int.class, int.class, int.class);
    MethodHandle mhTarget = lookup.findStatic(Transform.class, "targetMethod", typeTarget);
    MethodHandle mhResult = MethodHandles.foldArguments(mhTarget, mhCombiner);
    int value = (int) mhResult.invoke(3, 4); // 输出为 4
}
```

进行参数预处理的方法句柄会根据其类型中参数的个数 N ，从实际调用的参数列表中获取前面 N 个参数作为它需要处理的参数。如果预处理的方法句柄有返回值，返回值的类型需要与原始方法句柄的第一个参数的类型匹配。这是因为返回值会被作为调用原始方法句柄时的第一个参数来使用。

第五个方法 permuteArguments 的作用是对调用时的参数顺序进行重新排列，再传递给原始的方法句柄来完成调用。这种排列既可以是真正意义上的全排列，即所有的参数都在重新排列之后的顺序中出现；也可以是仅出现部分参数，没有出现的参数将被忽略；还可以重复某些参数，让这些参数在实际调用中出现多次。代码清单 2-55 给出了一个对参数进行完全排列的示例。代码中的原始方法句柄 mhCompare 所引用的是 Integer 类中的 compare 方法。当使用参数 3 和 4 进行调用的时候，返回值

是 -1。通过 `permuteArguments` 方法把参数的排列顺序进行颠倒，得到了新的方法句柄 `mhNew`。再用同样的参数调用方法句柄 `mhNew` 时，返回结果就变成了 1，因为传递给底层 `compare` 方法的实际调用参数变成了 4 和 3。新方法句柄 `mhDuplicateArgs` 在通过 `permuteArguments` 方法进行变换的时候，重复了第二个参数，因此传递给底层 `compare` 方法的实际调用参数是 4 和 4，返回的结果是 0。

代码清单 2-55 `permuteArguments` 方法的使用示例

```
public void permuteArguments() throws Throwable {
    MethodHandles.Lookup lookup = MethodHandles.lookup();
    MethodType type = MethodType.methodType(int.class, int.class, int.class);
    MethodHandle mhCompare = lookup.findStatic(Integer.class, "compare", type);
    int value = (int) mhCompare.invoke(3, 4); // 值为 -1
    MethodHandle mhNew = MethodHandles.permuteArguments(mhCompare, type, 1, 0);
    value = (int) mhNew.invoke(3, 4); // 值为 1
    MethodHandle mhDuplicateArgs = MethodHandles.permuteArguments(mhCompare, type,
        1, 1);
    value = (int) mhDuplicateArgs.invoke(3, 4); // 值为 0
}
```

在这里还要着重介绍一下 `permuteArguments` 方法的参数。第二个参数表示的是重新排列完成之后的新方法句柄的类型。紧接着的是多个用来表示新的排列顺序的整数。这些整数的个数必须与原始句柄的参数个数相同。整数出现的位置及其值就表示了在排列顺序上的对应关系。比如在上面的代码中，创建方法句柄 `mhNew` 的第一个整数参数是 1，这就表示调用原始方法句柄时的第一个参数的值实际上是调用新方法句柄时的第二个参数（编号从 0 开始，1 表示第二个）。

第六个方法 `catchException` 与原始方法句柄调用时的异常处理有关。可以通过该方法为原始方法句柄指定处理特定异常的方法句柄。如果原始方法句柄的调用正常完成，则返回其结果；如果出现了特定的异常，则处理异常的方法句柄会被调用。通过该方法可以实现通用的异常处理逻辑。可以对程序中可能出现的异常都提供一个进行处理的方法句柄，再通过 `catchException` 方法来封装原始的方法句柄。

如代码清单 2-56 所示，原始的方法句柄 `mhParseInt` 所引用的是 `Integer` 类中的 `parseInt` 方法，这个方法在字符串无法被解析成数字时会抛出 `java.lang.NumberFormatException`。来进行异常处理的方法句柄是 `mhHandler`，它引用了当前类中的 `handleException` 方法。通过 `catchException` 得到的新方法句柄 `mh` 在被调用时，如果抛出了 `NumberFormatException`，则会调用 `handleException` 方法。

代码清单 2-56 `catchException` 方法的使用示例

```
public int handleException(Exception e, String str) {
    System.out.println(e.getMessage());
    return 0;
}
```

```

public void catchExceptions() throws Throwable {
    MethodHandles.Lookup lookup = MethodHandles.lookup();
    MethodType typeTarget = MethodType.methodType(int.class, String.class);
    MethodHandle mhParseInt = lookup.findStatic(Integer.class, "parseInt",
        typeTarget);
    MethodType typeHandler = MethodType.methodType(int.class, Exception.class,
        String.class);
    MethodHandle mhHandler = lookup.findVirtual(Transform.class,
        "handleException", typeHandler).bindTo(this);
    MethodHandle mh = MethodHandles.catchException(mhParseInt,
        NumberFormatException.class, mhHandler);
    mh.invoke("Hello");
}

```

在这里需要注意几个细节：原始方法句柄和异常处理方法句柄的返回值类型必须是相同的，这是因为当产生异常的时候，异常处理方法句柄的返回值会作为调用的结果；而在两个方法句柄的参数方面，异常处理方法句柄的第一个参数是它所处理的异常类型，其他参数与原始方法句柄的参数相同。在异常处理方法句柄被调用的时候，其对应的底层方法可以得到原始方法句柄调用时的实际参数值。在上面的例子中，当 handleException 方法被调用的时候，参数 e 的值是 NumberFormatException 类的对象，参数 str 的值是原始的调用值“Hello”；在获得异常处理方法句柄的时候，使用了 bindTo 方法。这是因为通过 findVirtual 找到的方法句柄的第一个参数类型表示的是方法调用的接收者，这与 catchException 要求的第一个参数必须是异常类型的约束不相符，因此通过 bindTo 方法来为第一个参数预先绑定值。这样就可以得到所需的确切的方法句柄。当然，如果异常处理方法句柄所引用的是静态方法，就不存在这个问题。

最后一个对方法句柄进行变换时与参数相关的方法是 guardWithTest。这个方法可以实现在方法句柄这个层次上的条件判断的语义，相当于 if-else 语句。使用 guardWithTest 时需要提供 3 个不同的方法句柄：第一个方法句柄用来进行条件判断，而剩下的两个方法句柄则分别在条件成立和不成立的时候被调用。用来进行条件判断的方法句柄的返回值类型必须是布尔型，而另外两个方法句柄的类型则必须一致，同时也是生成的新方法句柄的类型。

如代码清单 2-57 所示，进行条件判断的方法句柄 mhTest 引用的是静态 guardTest 方法，在条件成立和不成立的时候调用的方法句柄则分别引用了 Math 类中的 max 方法和 min 方法。由于 guardTest 方法的返回值是随机为 true 或 false 的，所以两个方法句柄的调用也是随机选择的。

代码清单 2-57 guardWithTest 方法的使用示例

```

public static boolean guardTest() {
    return Math.random() > 0.5;
}

```

```

public void guardWithTest() throws Throwable {
    MethodHandles.Lookup lookup = MethodHandles.lookup();
    MethodHandle mhTest = lookup.findStatic(Transform.class, "guardTest",
        MethodType.methodType(boolean.class));
    MethodType type = MethodType.methodType(int.class, int.class, int.class);
    MethodHandle mhTarget = lookup.findStatic(Math.class, "max", type);
    MethodHandle mhFallback = lookup.findStatic(Math.class, "min", type);
    MethodHandle mh = MethodHandles.guardWithTest(mhTest, mhTarget, mhFallback);
    int value = (int) mh.invoke(3, 5); // 值随机为 3 或 5
}

```

除了可以在变换的时候对方法句柄的参数进行处理之外，还可以对方法句柄被调用后的返回值进行修改。对返回值进行处理是通过 filterReturnValue 方法来实现的。原始的方法句柄被调用之后的结果会被传递给另外一个方法句柄进行再次处理，处理之后的结果被返回给调用者。代码清单 2-58 展示了 filterReturnValue 的用法。原始的方法句柄 mhSubstring 所引用的是 String 类的 substring 方法，对返回值进行处理的方法句柄 mhUpperCase 所引用的是 String 类的 toUpperCase 方法。通过 filterReturnValue 方法得到的新方法句柄的运行效果是将调用 substring 得到的子字符串转换成大写的形式。

代码清单 2-58 filterReturnValue 方法的使用示例

```

public void filterReturnValue() throws Throwable {
    MethodHandles.Lookup lookup = MethodHandles.lookup();
    MethodHandle mhSubstring = lookup.findVirtual(String.class, "substring",
        MethodType.methodType(String.class, int.class));
    MethodHandle mhUpperCase = lookup.findVirtual(String.class, "toUpperCase",
        MethodType.methodType(String.class));
    MethodHandle mh = MethodHandles.filterReturnValue(mhSubstring, mhUpperCase);
    String str = (String) mh.invoke("Hello World", 5); // 输出 WORLD
}

```

7. 特殊方法句柄

在有些情况下，可能会需要对一组类型相同的方法句柄进行同样的变换操作。这个时候与其对所有的方法句柄都进行重复变换，不如创建出一个可以用来调用其他方法句柄的方法句柄。这种特殊的方法句柄的 invoke 方法或 invokeExact 方法被调用的时候，可以指定另外一个类型匹配的方法句柄作为实际调用的方法句柄。因为调用方法句柄时可以使用 invoke 和 invokeExact 两种方法，对应有两种创建这种特殊的方法句柄的方式，分别通过 MethodHandles 类的 invoker 和 exactInvoker 实现。两个方法都接受一个 MethodType 对象作为被调用的方法句柄的类型参数，两者的区别只在于调用时候的行为是类似于 invoke 还是 invokeExact。

代码清单 2-59 给出了 invoker 方法的使用示例。首先 invoker 方法句柄可以调用的方法句柄类型的返回值类型为 String，加上 3 个类型分别为 Object、int 和 int 的参数。

两个被调用的方法句柄，其中一个引用的是 String 类中的 substring 方法，另外一个引用的是当前类中的 testMethod 方法。这两个方法都可以通过 invoke 方法来正确调用。

代码清单 2-59 invoker 方法的使用示例

```
public void invoker() throws Throwable {
    MethodType typeInvoker = MethodType.methodType(String.class, Object.class,
        int.class, int.class);
    MethodHandle invoker = MethodHandles.invoker(typeInvoker);
    MethodType typeFind = MethodType.methodType(String.class, int.class, int.
        class);
    MethodHandles.Lookup lookup = MethodHandles.lookup();
    MethodHandle mh1 = lookup.findVirtual(String.class, "substring", typeFind);
    MethodHandle mh2 = lookup.findVirtual(InvokerUsage.class, "testMethod",
        typeFind);
    String result = (String) invoker.invoke(mh1, "Hello", 2, 3);
    result = (String) invoker.invoke(mh2, this, 2, 3);
}
```

而 exactInvoker 的使用与 invoker 非常类似，这里就不举例说明了。

上面提到了使用 invoker 和 exactInvoker 的一个重要好处就是在对这个方法句柄进行变换之后，所得到的新方法句柄在调用其他方法句柄的时候，这些变换操作都会被自动地引用，而不需要对每个所调用的方法句柄再单独应用。如代码清单 2-60 所示，通过 filterReturnValue 为通过 exactInvoker 得到的方法句柄添加变换操作，当调用方法句柄 mh1 的时候，这个变换会被自动应用，使作为调用结果的字符串自动变成大写形式。

代码清单 2-60 invoker 和 exactInvoker 对方法句柄变换的影响

```
public void invokerTransform() throws Throwable {
    MethodType typeInvoker = MethodType.methodType(String.class, String.class,
        int.class, int.class);
    MethodHandle invoker = MethodHandles.exactInvoker(typeInvoker);
    MethodHandles.Lookup lookup = MethodHandles.lookup();
    MethodHandle mhUpperCase = lookup.findVirtual(String.class, "toUpperCase",
        MethodType.methodType(String.class));
    invoker = MethodHandles.filterReturnValue(invoker, mhUpperCase);
    MethodType typeFind = MethodType.methodType(String.class, int.class, int.
        class);
    MethodHandle mh1 = lookup.findVirtual(String.class, "substring", typeFind);
    String result = (String) invoker.invoke(mh1, "Hello", 1, 4); // 值为“ELL”
}
```

通过 invoker 方法和 exactInvoker 方法得到的方法句柄被称为“元方法句柄”，具有调用其他方法句柄的能力。

8. 使用方法句柄实现接口

2.3 节介绍的动态代理机制可以在运行时为多个接口动态创建实现类，并拦截通过

接口进行的方法调用。方法句柄也具备动态实现一个接口的能力。这是通过 `java.lang.invoke.MethodHandleProxies` 类中的静态方法 `asInterfaceInstance` 来实现的。不过通过方法句柄来实现接口所受的限制比较多。首先该接口必须是公开的，其次该接口只能包含一个名称唯一的方法。这样限制是因为只有一个方法句柄用来处理方法调用。调用 `asInterfaceInstance` 方法时需要两个参数，第一个参数是要实现的接口类，第二个参数是处理方法调用逻辑的方法句柄对象。方法的返回值是一个实现了该接口的对象。当调用接口的方法时，这个调用会被代理给方法句柄来完成。方法句柄的返回值作为接口调用的返回值。接口的方法类型与方法句柄的类型必须是兼容的，否则会出现异常。

代码清单 2-61 是使用方法句柄实现接口的示例。被代理的接口是 `java.lang.Runnable`，其中仅包含一个 `run` 方法。实现接口的方法句柄引用的是当前类中的 `doSomething` 方法。在调用 `asInterfaceInstance` 之后得到的 `Runnable` 接口的实现对象被用来创建一个新的线程。该线程运行之后发现 `doSomething` 方法会被调用。这是由于当 `Runnable` 接口的 `run` 方法被调用的时候，方法句柄 `mh` 也会被调用。

代码清单 2-61 使用方法句柄实现接口的示例

```
public void doSomething() {
    System.out.println("WORK");
}

public void useMethodHandleProxy() throws Throwable {
    MethodHandles.Lookup lookup = MethodHandles.lookup();
    MethodHandle mh = lookup.findVirtual(UseMethodHandleProxies.class,
        "doSomething", MethodType.methodType(void.class));
    mh = mh.bindTo(this);
    Runnable runnable = MethodHandleProxies.asInterfaceInstance(Runnable.class, mh);
    new Thread(runnable).start();
}
```

通过方法句柄来实现接口的优势在于不需要新建额外的 Java 类，只需要复用已有的方法即可。在上面的示例中，任何已有的不带参数和返回值的方法都可以用来实现 `Runnable` 接口。需要注意的是，要求接口所包含的方法的名称唯一，不考虑 `Object` 类中的方法。实际的方法个数可能不止一个，可能包含同一方法的不同重载形式。

9. 访问控制权限

在通过查找已有类中的方法得到方法句柄时，要受限于 Java 语言中已有的访问控制权限。方法句柄与反射 API 在访问控制权限上的一个重要区别在于，在每次调用反射 API 的 `Method` 类的 `invoke` 方法的时候都需要检查访问控制权限，而方法句柄只在查找的时候需要进行检查。只要在查找过程中不出现问题，方法句柄在使用中就不会出现与访问控制权限相关的问题。这种实现方式也使方法句柄在调用时的性能要优于 `Method` 类。

之前介绍过，通过 `MethodHandles.Lookup` 类的方法可以查找类中已有的方法以得到

MethodHandle 对象。而 MethodHandles.Lookup 类的对象本身则是通过 MethodHandles 类的静态方法 lookup 得到的。在 Lookup 对象被创建的时候，会记录下当前所在的类（称为查找类）。只要查找类能够访问某个方法或域，就可以通过 Lookup 的方法来查找到对应的方法句柄。代码清单 2-62 给出了一个访问控制权限相关的示例。AccessControl 类中的 accessControl 方法返回了引用其中私有方法 privateMethod 的方法句柄。由于当前查找类可以访问该私有方法，因此查找过程是成功的。其他类通过调用 accessControl 得到的方法句柄就可以调用这个私有方法。虽然其他类不能直接访问 AccessControl 类中的私有方法，但是在调用方法句柄的时候不会进行访问控制权限检查，因此对方法句柄的调用可以成功进行。

代码清单 2-62 方法句柄查找时的访问控制权限

```
public class AccessControl {
    private void privateMethod() {
        System.out.println("PRIVATE");
    }

    public MethodHandle accessControl() throws Throwable {
        MethodHandles.Lookup lookup = MethodHandles.lookup();
        MethodHandle mh = lookup.findSpecial(AccessControl.class, "privateMethod",
            MethodType.methodType(void.class), AccessControl.class);
        mh = mh.bindTo(this);
        return mh;
    }
}
```

10. 交换点

交换点是在多线程环境下控制方法句柄的一个开关。这个开关只有两个状态：有效和无效。交换点初始时处于有效状态，一旦从有效状态变到无效状态，就无法再继续改变状态。也就是说，只允许发生一次状态改变。这种状态变化是全局和即时生效的。使用同一个交换点的多个线程会即时观察到状态变化。交换点用 java.lang.invoke.SwitchPoint 类来表示。通过 SwitchPoint 对象的 guardWithTest 方法可以设置在交换点的不同状态下调用不同的方法句柄。这个方法的作用类似于 MethodHandles 类中的 guardWithTest 方法，只不过少了用来进行条件判断的方法句柄，只有条件成立和不成立时分别调用的方法句柄。这是因为选择哪个方法句柄来执行是由交换点的有效状态来决定的，不需要额外的条件判断。

在代码清单 2-63 中，在调用 guardWithTest 方法的时候指定在交换点有效的时候调用方法句柄 mhMin，而在无效的时候则调用 mhMax。guardWithTest 方法的返回值是一个新的方法句柄 mhNew。交换点在初始时处于有效状态，因此 mhNew 在第一次调用时使用的是 mhMin，结果为 3。在通过 invalidateAll 方法把交换点设成无效状态之后，再

次调用 mhNew 时实际调用的方法句柄就变成了 mhMax，结果为 4。

代码清单 2-63 交换点的使用示例

```
public void useSwitchPoint() throws Throwable {
    MethodHandles.Lookup lookup = MethodHandles.lookup();
    MethodType type = MethodType.methodType(int.class, int.class, int.class);
    MethodHandle mhMax = lookup.findStatic(Math.class, "max", type);
    MethodHandle mhMin = lookup.findStatic(Math.class, "min", type);
    SwitchPoint sp = new SwitchPoint();
    MethodHandle mhNew = sp.guardWithTest(mhMin, mhMax);
    mhNew.invoke(3, 4); // 值为 3
    SwitchPoint.invalidateAll(new SwitchPoint[] {sp});
    mhNew.invoke(3, 4); // 值为 4
}
```

交换点的一个重要作用是在多线程环境下使用，可以在多个线程中共享同一个交换点对象。当某个线程的交换点状态改变之后，其他线程所使用的 guardWithTest 方法返回的方法句柄的调用行为就会发生变化。

11. 使用方法句柄进行函数式编程

通过上面章节对方法句柄的详细介绍可以看出，方法句柄是一个非常灵活的对方法进行操作的轻量级结构。方法句柄的作用类似于在某些语言中出现的函数指针（function pointer）。在程序中，方法句柄可以在对象之间自由传递，不受访问控制权限的限制。方法句柄的这种特性，使得在 Java 语言中也可以进行函数式编程。下面通过几个具体的示例来进行说明。

第一个示例是对数组进行操作。数组作为一个常见的数据结构，有的编程语言提供了对它进行复杂操作的功能。这些功能中比较常见的是 forEach、map 和 reduce 操作等。这些操作的语义并不复杂，forEach 是对数组中的每个元素都依次执行某个操作，而 map 则是把原始数组按照一定的转换过程变成一个新的数组，reduce 是把一个数组按照某种规则变成单个元素。这些操作在其他语言中可能比较好实现，而在 Java 语言中，则需要引入一些接口，由此带来的是繁琐的实现和冗余的代码。有了方法句柄之后，这个实现就变得简单多了。代码清单 2-64 给出了使用方法句柄的 forEach、map 和 reduce 方法的实现。对数组中元素的处理是由一个方法句柄来完成的。对这个方法句柄只有类型的要求，并不限制它所引用的底层方法所在的类或名称。

代码清单 2-64 使用方法句柄实现数组操作的示例

```
private static final MethodType typeCallback = MethodType.methodType(Object.class,
    Object.class, int.class);

public static void forEach(Object[] array, MethodHandle handle) throws Throwable {
    for (int i = 0, len = array.length; i < len; i++) {
        handle.invoke(array[i], i);
    }
}
```

```

    }
}

public static Object[] map(Object[] array, MethodHandle handle) throws Throwable {
    Object[] result = new Object[array.length];
    for (int i = 0, len = array.length; i < len; i++) {
        result[i] = handle.invoke(array[i], i);
    }
    return result;
}

public static Object reduce(Object[] array, Object initialValue, MethodHandle
handle) throws Throwable {
    Object result = initialValue;
    for (int i = 0, len = array.length; i < len; i++) {
        result = handle.invoke(result, array[i]);
    }
    return result;
}

```

第二个例子是方法的柯里化 (currying)。柯里化的含义是对一个方法的参数值进行预先设置之后，得到一个新的方法。比如一个做加法运算的方法，本来有两个参数，通过柯里化把其中一个参数的值设为 5 之后，得到的新方法就只有一个参数。新方法的运行结果是用 5 加上这个唯一的参数的值。通过 MethodHandles 类中的 insertArguments 方法可以很容易地实现方法句柄的柯里化。代码清单 2-65 给出了相关的实现。方法 curry 负责把一个方法句柄的第一个参数的值设为指定值；add 方法就是一般的加法操作；add5 方法对引用 add 的方法句柄进行柯里化，得到新的方法句柄，再调用此方法句柄。

代码清单 2-65 使用方法句柄实现的柯里化

```

public static MethodHandle curry(MethodHandle handle, int value) {
    return MethodHandles.insertArguments(handle, 0, value);
}

public static int add(int a, int b) {
    return a + b;
}

public static int add5(int a) throws Throwable {
    MethodHandles.Lookup lookup = MethodHandles.lookup();
    MethodType type = MethodType.methodType(int.class, int.class, int.class);
    MethodHandle mhAdd = lookup.findStatic(Curry.class, "add", type);
    MethodHandle mh = curry(mhAdd, 5);
    return (int) mh.invoke(a);
}

```

上面给出的这两个示例所实现的功能虽然比较简单，但是反映出了方法句柄在使用

时的极大灵活性。配合方法句柄支持的变换操作，可以实现很多有趣和实用的功能。

2.4.3 invokedynamic 指令

在详细介绍了 `java.lang.invoke` 包中与方法句柄相关的 API 之后，现在要深入到 Java 虚拟机的指令层次。本节将要介绍的是 JSR 292 中引入的新的方法调用指令 `invokedynamic`。这个新指令的引入，为 Java 虚拟机平台上动态语言的开发带来了福音。动态语言的实现者终于可以在灵活性、复杂性和性能这几个要素之间找到一个很好的平衡。利用 `invokedynamic` 指令，可以通过简单高效的方式实现灵活性很强的方法调用。

方法调用是 Java 虚拟机执行过程中最常见也是最重要的指令，控制着程序的具体执行流程。在 Java 7 之前，Java 虚拟机中包含了 4 种方法调用指令，分别是 `invokestatic`、`invokespecial`、`invokevirtual` 和 `invokeinterface`。这 4 种指令分别适用于不同的方法调用场景，都可以在 Java 语言的源代码中找到相应的产生模式。下面先介绍这 4 种方法调用指令。

1. 普通方法调用指令

在本章的开始部分中提到了一个现象，那就是 Java 字节代码规范受 Java 语言的影响很深。因此虽然本节介绍的主体是 Java 字节代码规范中的方法调用指令，但是仍然免不了用 Java 语言的语法来做类比。代码清单 2-66 给出了 Java 程序中可能会出现的几种方法调用形式。

代码清单 2-66 Java 程序中的方法调用形式

```
public interface SampleInterface {  
    void sampleMethodInInterface();  
}  
  
public class Sample implements SampleInterface {  
    public void sampleMethodInInterface() {}  
    public void normalMethod() {}  
    public static void staticSampleMethod() {}  
}  
  
public class MethodInvokeTypes {  
    public void invoke() {  
        SampleInterface sample = new Sample();  
        sample.sampleMethodInInterface();  
        Sample newSample = new Sample();  
        newSample.normalMethod();  
        Sample.staticSampleMethod();  
    }  
}
```

上面代码中包含一个接口 `SampleInterface`，以及这个接口的实现类 `Sample`。类

Sample 中除了实现 SampleInterface 接口所需要的 sampleMethodInInterface 方法之外，还包含一个一般的公开方法 normalMethod 和一个静态方法 staticSampleMethod。类 MethodInvokeTypes 的 invoke 方法中既有通过构造方法创建新的对象，又有对接口中方法和类中的一般方法和静态方法的调用。这实际上就代表了 Java 语言中所提供的 4 种方法调用形式。而在 Java 字节代码规范中，同样有 4 种方法调用指令与这 4 种调用方式相对应。

为了探究这 4 种方法调用指令，需要查看包含 Java 字节代码的 class 文件的内容。在这里需要使用 JDK 中自带的 javap 工具来完成。比如代码清单 2-66 中的 MethodInvokeTypes 类编译出来的 class 文件，可以通过在类文件所在目录下使用 javap -verbose MethodInvokeTypes 命令来显示 class 文件的内容。一个 class 文件中包含的内容很多，javap 工具也会输出很多内容。本书第 8 章会详细介绍 Java 字节代码的格式，这里只介绍与方法调用相关的指令。图 2-1 给出了在 javap 输出中与方法调用相关的部分。

```

public void invoke();
Code:
Stack=2, Locals=3, Args_size=1
  0: new      #2; //class com/java7book/chapter2/invoke/Sample
  3: dup
  4: invokespecial #3; //Method com/java7book/chapter2/invoke/Sample."<init>":()V
  7: astore_1
  8: aload_1
  9: invokeinterface #4,  1; //InterfaceMethod com/java7book/chapter2/invoke/
SampleInterface.sampleMethodInInterface:()V
 14: new      #2; //class com/java7book/chapter2/invoke/Sample
 17: dup
 18: invokespecial #3; //Method com/java7book/chapter2/invoke/Sample."<init>":()V
 21: astore_2
 22: aload_2
 23: invokevirtual #5; //Method com/java7book/chapter2/invoke/Sample.normal
Method:()V
 26: invokestatic #6; //Method com/java7book/chapter2/invoke/Sample.static
SampleMethod:()V
 29: return

```

图 2-1 通过 javap 工具查看方法调用相关的字节代码内容

按照 Java 源代码中的顺序来看，第一个方法调用指令是 invokespecial，这个指令是用来调用类的构造方法、父类的方法（通过 super）和私有方法的。这里的 invokespecial 指令调用的是 Sample 类的构造方法，对应源代码中的“new Sample()”。第二个方法调用指令是 invokeinterface，这个指令是通过接口来调用方法的。这里的 invokeinterface 指令调用的是 SampleInterface 接口中的 sampleMethodInInterface 方法。第三个方法调用指令是 invokevirtual，这个指令是用来调用类中的一般方法的。所调用的实际方法取决于调用接收者对象的运行时类型。这里的 invokevirtual 指令调用的是 Sample 类中的 normalMethod 方法。最后一个方法调用指令是 invokestatic，这个指令用来调用类中的静态方法。这里调用的是 Sample 类的 staticSampleMethod 方法。

在这 4 个方法调用指令中，除了 invokestatic 之外，都需要一个调用的接收者。因为

静态方法是在类中定义的，并不需要一个具体的对象实例作为接收者。其他 3 个调用指令的接收者就是方法调用表达式的“.”之前的对象。这些接收者有一个静态的类型，也就是在编译时刻确定的调用对象的类型。对于 invokespecial 和 invokevirtual 来说，这个静态类型就是接收者对象的类型；而对于 invokeinterface 来说，这个静态类型就是接口的类型。而在运行时刻，接收者会有一个动态类型。这个动态类型可能和编译时刻确定的静态类型一样，也可能不一样。这其中的原因是存在运行时的方法派发。如果这个动态类型和静态类型不一样，那么该动态类型肯定是静态类型的子类型，可能是静态类型所表示的 Java 类的子类或所表示的接口的实现类。在代码清单 2-67 中，hashCode 方法的接收者的静态类型是 Object 类，但是其在运行时刻的动态类型是 String 类，所调用的是 String 类中定义的 hashCode 方法。

代码清单 2-67 静态类型与动态类型的区别

```
String str = "Hello World";
Object obj = str;
System.out.println(obj.hashCode());
```

在这里需要说明的是，这 4 种普通的方法调用指令只支持方法调用时的单派发 (single dispatch)，也就是说实际调用时对方法的选择只会根据调用的接收者不同而有所不同，不受其他因素的影响。这种单派发只对 invokevirtual 和 invokeinterface 两种指令有效。由于类继承和接口实现机制的存在，实际的方法调用接收者可能是声明时类型的子类是接口的实现类，取决于运行时刻的动态类型。而另外的 invokestatic 和 invokespecial 指令的调用接收者都是固定的，是无法在运行时改变的。如果按照方法句柄的方式，把方法的调用者也抽象成方法调用时的一个参数，就可以知道单派发方式实际上只根据方法调用时的第一个参数来进行方法的分配。有些编程语言需要支持多派发，也就是说在方法调用时会根据多个参数的值来选择要具体执行的方法。多派发在某些情况下会使代码编写起来更加简单。

2. invokedynamic 指令简介

上一节说明了 Java 字节代码规范中的 4 种普通的方法调用指令。这 4 种方法调用指令的特点是在 Java 语言中有相应的语法形式与其对应，同时灵活性比较低。下面从典型的方法调用流程来说明灵活性不足体现在什么地方。

当 Java 虚拟机执行方法调用的时候，需要确定下面 4 个要素。

- 1) 名称：要调用的方法的名称一般是由开发人员在源代码中指定的符号名称。这个名称同样会出现在编译之后的字节代码中。
 - 2) 链接：链接包含了要调用方法的类。这一步有可能会涉及类的加载。
 - 3) 选择：选择要调用的方法。在类中根据方法名称和参数选择要调用的方法。
 - 4) 适配：调用者和接收者对调用的方式达成一致，即对方法的类型声明达成共识。
- 确定了上面 4 个要素之后，Java 虚拟机会把控制权转移到被调用的方法中，并把调

用时的实际参数传递过去。

再结合图2-1给出的4种调用指令来看这4个要素是如何体现的：所有这4种调用指令中的方法名称都是直接在字节代码中固定下来的，从Java源代码中直接映射过来。而链接的过程也由Java虚拟机来统一处理。唯一可能变化的就是方法的选择和适配。对于`invokestatic`来说，方法的选择是固定的，总是调用声明了此静态方法的类中的方法。另外方法的声明也必须是完全匹配的。对于`invokespecial`来说，由于它所调用的方法只有当前类的对象才有权限调用，因此它的方法选择也是固定的。而对于`invokevirtual`和`invokeinterface`来说，由于类继承和接口实现的存在，它们的方法选择是不固定的，但是仅限于根据调用的接收者类型来进行选择，即上面提到的单派发机制。

比如代码清单2-67中的`hashCode`方法的调用，实际方法的选择取决于调用的接收者。如果接收者是一个`Object`类型的对象，那么调用的是`Object`类中的方法；如果接收者是一个`String`类型的对象，那么调用的是`String`类中的方法；如果另外一个类继承自`Object`类，但是没有覆写`hashCode`方法，那么调用的还是`Object`类中的方法。

从方法适配的角度来说，只有接收者一方能进行适配，而且只能缩小类型的范围。比如在调用一个类中的方法的时候，接收者可以换成该类的子类的对象，但是不能换成父类的对象。

Java 7中新引入`invokedynamic`指令的目的就是弥补已有的4个方法调用指令的不足，提供更加强大的灵活性。既可以方便Java虚拟机上动态语言的编译器开发人员，也适应于对方法调用灵活性要求较高的一般应用。

新指令`invokedynamic`在多个方面解放了方法调用，还是通过上面给出的方法调用4个要素来说明：

1) 在方法的名称方面，不一定是符合Java命名规范的字符串，可以任意指定。方法的调用者和提供者也不需要在方法名称上达成一致。实际上，上一节介绍的方法句柄就已经把方法的名称剥离出去了。

2) 提供了更加灵活的链接方式。一个方法调用所实际调用的方法可以在运行时再确定。这就相当于把链接操作推迟到了运行时，而不是必须在编译时就确定下来。对于一个已经链接好的方法调用，也可以重新进行链接，让它指向另外的方法。

3) 在方法选择方面，不再是只能在方法调用的接收者上进行派发，而是可以考虑所有调用时的参数，即支持方法的多派发。

4) 在调用之前，可以对参数进行各种不同的处理，包括类型转换、添加和删除参数、收集和分发可变长度参数等。在2.4.2节中已经介绍过这些变换操作了。

新的`invokedynamic`指令需要与2.4.2节中介绍的方法句柄结合起来使用。该指令的灵活性在很大程度上取决于方法句柄的灵活性。对于`invokedynamic`指令来说，在Java源代码中是没有直接的对应产生方式的。这也是`invokedynamic`指令的新颖之处。它是一个完全的Java字节代码规范中的指令。传统的Java编译器并不会帮开发人员生成`invokedynamic`指令。为了利用`invokedynamic`指令，需要开发人员自己来生成包含这

个指令的 Java 字节代码。因为这个指令本来就是设计给动态语言的编译器使用的，所以这种限制也是合理的。对于一般的程序来说，如果希望使用这个指令，就需要使用操作 Java 字节代码的工具来完成。本书第 8 章会详细介绍如何对字节代码进行操作。这里不再详细介绍工具的用法，读者只需要理解最终生成的字节代码中所包含的内容就可以了。

在字节代码中每个出现的 `invokedynamic` 指令都成为一个动态调用点（dynamic call site）。每个动态调用点在初始化的时候，都处于未链接的状态。在这个时候，这个动态调用点并没有被指定要调用的实际方法。

当 Java 虚拟机要执行 `invokedynamic` 指令时，首先需要链接其对应的动态调用点。在链接的时候，Java 虚拟机会先调用一个启动方法（bootstrap method）。这个启动方法的返回值是 `java.lang.invoke.CallSite` 类的对象。在通过启动方法得到了 `CallSite` 之后，通过这个 `CallSite` 对象的 `getTarget` 方法可以获取到实际要调用的目标方法句柄。有了方法句柄之后，对这个动态调用点的调用，实际上是代理给方法句柄来完成的。也就是说，对 `invokedynamic` 指令的调用实际上就等价于对方法句柄的调用，具体来说是被转换成对方法句柄的 `invoke` 方法的调用。

3. 动态调用点

Java 7 中提供了三种类型的动态调用点 `CallSite` 的实现，分别是 `java.lang.invoke.ConstantCallSite`、`java.lang.invoke.MutableCallSite` 和 `java.lang.invoke.VolatileCallSite`。这些 `CallSite` 实现的不同之处在于所对应的目标方法句柄的特性不同。`ConstantCallSite` 所表示的调用点绑定的是一个固定的方法句柄，一旦链接之后，就无法修改；`MutableCallSite` 所表示的调用点则允许在运行时动态修改其目标方法句柄，即可以重新链接到新的方法句柄上；而 `VolatileCallSite` 的作用与 `MutableCallSite` 类似，不同的是它适用于多线程情况，用来保证对于目标方法句柄所做的修改能够被其他线程看到。这也是名称中 `volatile` 的含义所在，类似于 Java 中的 `volatile` 关键词的作用。

虽然 `CallSite` 一般同 `invokedynamic` 指令结合起来使用，但是在 Java 代码中也可以通过调用 `CallSite` 的 `dynamicInvoker` 方法来获取一个方法句柄。调用这个方法句柄就相当于执行 `invokedynamic` 指令。通过此方法可以预先对 `CallSite` 进行测试，以保证字节代码中的 `invokedynamic` 指令的行为是正确的，毕竟在生成的字节代码中进行调试是一件很麻烦的事情。下面介绍 `CallSite` 时会先通过 `dynamicInvoker` 方法在 Java 程序中直接试验 `CallSite` 的使用。

首先介绍 `ConstantCallSite` 的使用。`ConstantCallSite` 要求在创建的时候就指定其链接到的目标方法句柄。每次该调用点被调用的时候，总是会执行对应的目标方法句柄。在代码清单 2-68 中，创建了一个 `ConstantCallSite` 并指定目标方法句柄为引用 `String` 类中的 `substring` 方法。

代码清单 2-68 `ConstantCallSite` 的使用示例

```
public void useConstantCallSite() throws Throwable {
```

```

MethodHandles.Lookup lookup = MethodHandles.lookup();
MethodType type = MethodType.methodType(String.class, int.class, int.class);
MethodHandle mh = lookup.findVirtual(String.class, "substring", type);
ConstantCallSite callSite = new ConstantCallSite(mh);
MethodHandle invoker = callSite.dynamicInvoker();
String result = (String) invoker.invoke("Hello", 2, 3);
}

```

接下来的 MutableCallSite 则允许对其所关联的目标方法句柄进行修改。修改操作是通过 setTarget 方法来完成的。在创建 MutableCallSite 的时候，既可以指定一个方法类型 MethodType，又可以指定一个初始的方法句柄。如果像下面代码中那样指定方法类型，则通过 setTarget 设置的方法句柄都必须有同样的方法类型。如果创建时指定的是初始的方法句柄，则之后设置的其他方法句柄的类型也必须与初始的方法句柄相同。MutableCallSite 对象中的目标方法句柄的类型总是固定的。下面的代码通过 setTarget 方法把目标方法句柄分别设置为 Math 类中的 max 和 min 方法，在调用 MutableCallSite 时可以得到不同的结果。

代码清单 2-69 MutableCallSite 的使用示例

```

public void useMutableCallSite() throws Throwable {
    MethodType type = MethodType.methodType(int.class, int.class, int.class);
    MutableCallSite callSite = new MutableCallSite(type);
    MethodHandle invoker = callSite.dynamicInvoker();
    MethodHandles.Lookup lookup = MethodHandles.lookup();
    MethodHandle mhMax = lookup.findStatic(Math.class, "max", type);
    MethodHandle mhMin = lookup.findStatic(Math.class, "min", type);
    callSite.setTarget(mhMax);
    int result = (int) invoker.invoke(3, 5); // 值为 5
    callSite.setTarget(mhMin);
    result = (int) invoker.invoke(3, 5); // 值为 3
}

```

需要考虑的是多线程情况下的可见性问题。有可能在一个线程中对 MutableCallSite 的目标方法句柄做了修改，而在另外一个线程中不能及时看到这个变化。对于这种情况，MutableCallSite 提供了一个静态方法 syncAll 来强制要求各个线程中 MutableCallSite 的使用者立即获取最新的目标方法句柄。该方法接收一个 MutableCallSite 类型的数组作为参数。

如果一个目标方法句柄可变的调用点被设计为在多线程的情况下使用，可以直接使用 VolatileCallSite，而不使用 MutableCallSite。当使用 VolatileCallSite 的时候，每当目标方法句柄发生变化的时候，其他线程会自动看到这个变化。这与 Java 中 volatile 关键词的语义是一样的。这比使用 MutableCallSite 再加上 syncAll 方法要简单得多。除了这一点之外，VolatileCallSite 的作用与 MutableCallSite 完全相同。

4. invokedynamic 指令实战

下面将要介绍 invokedynamic 指令在 Java 字节代码中的具体使用方式。由于涉及字节代码的生成，这里使用了 ASM 工具^Θ。暂时不会对 ASM 工具的使用做过多的介绍，在第 8 章中会进行详细介绍。首先需要提供 invokedynamic 指令所需的启动方法，如代码清单 2-70 所示。

代码清单 2-70 invokedynamic 指令的启动方法

```
public class ToUpperCase {
    public static CallSite bootstrap(Lookup lookup, String name, MethodType type,
        String value) throws Exception {
        MethodHandle mh = lookup.findVirtual(String.class, "toUpperCase",
            MethodType.methodType(String.class)).bindTo(value);
        return new ConstantCallSite(mh);
    }
}
```

该启动方法是一个普通的 Java 类中的方法。该方法的类型声明可以是多种格式。返回值必须是 CallSite，而参数则允许多种形式。在典型情况下，前面的 3 个参数分别是进行方法查找的 MethodHandles.Lookup 对象、方法的名称和方法的类型 MethodType。这 3 个参数之后的其他参数都会被传递给 CallSite 对应的方法句柄。在上面的代码中，使用了一个 ConstantCallSite，而该调用点所绑定的方法句柄引用的底层方法是 String 类中的 toUpperCase 方法。启动方法 bootstrap 接收一个额外的参数 value。这个参数被预先绑定给方法句柄。因此当该方法句柄被调用的时候，不需要额外的参数，而返回结果是对参数 value 表示的字符串调用 toUpperCase 方法的结果。

有了启动方法之后，就需要在字节代码中生成 invokedynamic 指令。代码清单 2-71 给出的程序会产生一个新的 Java 类文件 ToUpperCaseMain.class。通过 java 命令可以运行该类文件，输出结果是“HELLO”。

代码清单 2-71 生成使用 invokedynamic 指令的字节代码

```
public class ToUpperCaseGenerator {
    private static final MethodHandle BSM =
        new MethodHandle(MH_INVOKESTATIC,
            ToUpperCase.class.getName().replace('.', '/'),
            "bootstrap",
            MethodType.methodType(
                CallSite.class, Lookup.class, String.class, MethodType.class, String.
                class).toMethodDescriptorString());
    public static void main(String[] args) throws IOException {
```

^Θ ASM 工具的官方网站是 <http://asm.ow2.org/>。

```

ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES);
cw.visit(V1_7, ACC_PUBLIC | ACC_SUPER, "ToUpperCaseMain", null, "java/
    lang/Object", null);
MethodVisitor mv = cw.visitMethod(ACC_PUBLIC | ACC_STATIC, "main",
    "([Ljava/lang/String;)V", null, null);
mv.visitCode();
mv.visitFieldInsn(GETSTATIC, "java/lang/System", "out", "Ljava/io/
    PrintStream;");
mv.visitInvokeDynamicInsn("toUpperCase", "()Ljava/lang/String;", BSM,
    "Hello");
mv.visitMethodInsn(INVOKEVIRTUAL, "java/io/PrintStream", "println",
    "(Ljava/lang/String;)V");
mv.visitInsn(RETURN);
mv.visitMaxs(0, 0);
mv.visitEnd();
cw.visitEnd();

Files.write(Paths.get("ToUpperCaseMain.class"), cw.toByteArray());
}
}

```

上面的代码中包含了大量使用 ASM 工具的代码，这里只需要关心的是“mv.visitInvokeDynamicInsn("toUpperCase", "()Ljava/lang/String;", BSM, "Hello");”这行代码。这行代码是用来在字节代码中生成 invokedynamic 指令的。在调用的时候传入了方法的名称、方法句柄的类型、对应的启动方法和额外的参数 “Hello”。在 invokedynamic 指令被执行的时候，会先调用对应的启动方法，即代码清单 2-70 中的 bootstrap 方法。bootstrap 方法的返回值是一个 ConstantCallSite 的对象。接着从该 ConstantCallSite 对象中通过 getTarget 方法获取目标方法句柄，最后再调用此方法句柄。在调用 visitInvokeDynamicInsn 方法时提供了一个额外的参数 “Hello”。这个参数会被传递给 bootstrap 方法的最后一个参数 value，用来创建目标方法句柄。当目标方法句柄被调用的时候，返回的结果是把参数 “Hello” 转换成大写形式之后的值 “HELLO”。

从上面这个简单的示例可以看出，invokedynamic 指令是如何与方法句柄结合起来使用的。上面的示例只使用了最简单的 ConstantCallSite。复杂的示例包括根据参数的值确定需要返回的 CallSite 对象，或是对已有的 MutableCallSite 对象的目标方法句柄进行修改等。

2.5 小结

单纯从编程语言的角度来说，静态类型语言和动态类型语言都有各自的优劣势。静态类型语言所提供的强大的编译时刻类型检查能力，可以尽可能早地发现程序中存在的类型错误。其严谨的语法对初学者来说更加容易理解。而动态类型语言所能提供的是更加灵活和简洁的语法。这通常意味着更少的代码量和更易理解的代码逻辑，同时也对开

发人员提出了更高的要求。考虑到这些优劣势，开发人员可以根据应用开发的需要来灵活选择。编程语言不应该对开发人员的这种选择造成阻碍。开发人员也可以在一个应用的开发中使用多种不同的编程语言来开发不同的组件。

Java 语言虽然是静态类型语言，但是它也提供了足够多的动态性来应对灵活性要求较高的场景。这些动态性体现在本章介绍的脚本语言支持 API、反射 API、动态代理和 Java 7 中通过 JSR 292 引入的动态语言支持上。基于 Java 平台的开发人员可以选择各种不同的方式来应对灵活性的要求。可以选择把灵活性和动态性完全交给脚本语言去解决，再通过脚本语言支持 API 集成到主 Java 程序中；也可以通过反射 API 在运行时刻动态地调用方法；当需要对接口中的方法调用进行拦截时，动态代理是一个很好的选择；JSR 292 引入的方法句柄在灵活性上要远胜于反射 API 中的 Method 类。在方法句柄上的各种变换操作足以应付多种常见的需求。

第 3 章 Java I/O

绝大多数应用程序在运行过程中都会进行两种类型的计算：一种是占用 CPU 时间的计算，另外一种是与数据输入 / 输出（I/O）相关的计算。在这两种计算中，一般是与 I/O 相关的计算所花费的时间占较大的比重。这其中的主要原因是在进行 I/O 操作时，一般需要竞争操作系统中有限的资源，或是需要等待速度较慢的外部设备完成其操作，从而造成 I/O 相关的计算所等待的时间较长。从性能优化的角度出发，提升 I/O 相关操作的性能会对应用程序的整体性能产生比较大的帮助。

Java 平台提供了丰富的标准类库来满足应用程序中可能出现的与 I/O 操作相关的需求。这些标准类库本身也在不断发展，从最初的 `java.io` 包，到 JDK 1.4 中的新 IO 支持（JSR 51: New I/O APIs for the Java™ Platform, NIO），再到 Java 7 中的 NIO 的补充类库 NIO.2（JSR 203: More New I/O APIs for the Java™ Platform）。Java 平台对 I/O 的支持功能越发强大。很多之前需要开发人员来编写的功能，在现在的标准库中都有了实现。不过开发人员需要额外的时间来学习这些新 API 的使用。

对于 I/O 操作来说，其根本的作用在于传输数据。输入和输出指的仅是数据的流向，实际传输是通过某些具体的媒介来完成的。这其中最主要的媒介是文件系统和网络连接。这两种传输方式也在 Java I/O 中受到了良好的支持。从不同的抽象层次来看 I/O 操作，所得到的 API 是不同的。最早的 `java.io` 包把 I/O 操作抽象成数据的流动，进而有了流（stream）的概念。在 Java NIO 中，则把 I/O 操作抽象成端到端的一个数据连接，这就有了通道（channel）的概念。不同的抽象层次对开发人员所暴露的复杂度是不同的。推荐开发人员使用 Java NIO 中新的通道的概念。在下面的内容中会详细介绍流和通道相关的内容。

3.1 流

流是 Java 中最早提供的对 I/O 操作的抽象，从 JDK 1.0 就存在了。流把 I/O 操作抽象成数据的流动。流所代表的是流动中的数据。对于传输的数据来说，除了最底层的字节表示外，还支持不同的抽象表示方式。对一个计算机程序来说，其数据的最终表现形式都是 0 或 1 的比特值。程序一般不直接处理单个的比特值，而是处理由 8 个比特组成的字节。不管是内存中的数据、磁盘上的数据，还是通过网络传输的数据，其基本格式都是一系列的字节。所不同的是，不同的程序对这一系列的字节有不同的解释方式。将一组字节按照特定的方式进行解释，就形成了编程语言中的不同的基本类型。比如在 Java 语言中，4 个字节可以表示一个整数（`int`）或是单精度浮点数（`float`），而 8 个字节

可以表示一个长整数 (long) 或是双精度浮点数 (double)。单纯的一个字节序列可以有多种不同的解释方式。比如一个 16 个字节的数组，既可以解释成 4 个整数，也可以解释成 2 个双精度浮点数。不同的解释所表示的语义是完全不同的。这种解释工作是由应用程序自己来完成的，编程语言的类库一般会提供相关的支持。

Java 中最基本的流是在字节这个层次上进行操作的。也就是说基本的流只负责在来源和目的之间传输字节，并不负责对字节的含义进行解释。在基本的字节流基础上，Java 也提供了一些过滤流 (filter stream) 的实现。这些过滤流实际上是基本字节流上的一个封装，在其上增加了不同的处理能力，如基本类型与字节序列之间的转换等。这些过滤流对开发人员的接口更加友好，可以自动完成很多转换工作。

3.1.1 基本输入流

最基本的 I/O 流是 `java.io` 包中的抽象类 `java.io.InputStream` 和 `java.io.OutputStream`。由于流的相关 API 设计得比较早，因此并没有采用现在流行的面向接口编程的思路，而是采用了抽象类。新的 I/O 相关的 API 则大量使用了接口。如果流的实现只对使用者暴露字节这个层次的细节，则可以直接继承 `InputStream` 或 `OutputStream` 类，并提供自己额外的能力。

输入流 `InputStream` 类中包含两类功能：一类是与读取流中字节数据相关的功能，另一类则是流的控制功能。读取流中的字节通过 `read` 方法来完成。该方法有 3 种重载形式：第一种形式不带任何参数，每次读取一个字节并返回；第二种形式使用字节数组作为缓冲区，用读取到的字节数据填充缓冲区；最后一种形式需要提供作为缓冲区使用的字节数组和数组中的起始位置和长度，读取到的字节数据被填充到缓冲区的指定位置上。这 3 种形式中，第一种是声明为 `abstract` 的，必须由子类来实现。而对于另外两种，`InputStream` 类有自己的默认实现，通过循环调用第一种形式的 `read` 方法来填充缓冲区。

最常见的读取 `InputStream` 类的对象中的数据的方式是创建一个字节数组作为缓冲区，然后循环读取，直到 `read` 方法返回 -1 或抛出 `java.io.IOException` 异常。`read` 方法的返回值是每次调用中成功读取的字节数。在读取数据的过程中，对 `read` 方法的调用是阻塞的。当流中没有数据可用时，对 `read` 方法的调用需要等待。这种阻塞式的特性可能会成为应用中的性能瓶颈。如果不使用字节数组作为缓冲区，`read` 方法一次只能读入一个字节。在提供缓冲区的情况下，虽然 `InputStream` 类也只是以循环的方式每次读取一个字节来填充缓冲区，但是 `InputStream` 类的子类一般会为接受缓冲区作为参数的 `read` 方法提供更加高效的实现。这也是为什么使用缓冲区的重要原因。

从流本身所代表的抽象层次出发，它表示的是一个流动的字节流，如流水一样。正因为如此，流中所包含的字节一旦流过去，就无法再重新使用。从这个角度出发，对一般的输入流所能做的操作就只是顺序地读取，直到流的末尾或中间出现读取错误。当然，不同的输入流可能也支持额外的控制操作，因此 `InputStream` 类中也包含了相应的

方法来允许其子类进行选择性地覆写。这也是采用抽象类的设计方式所带来的弊端。所有可能会用到的方法都需要在抽象类中进行声明。

第一个最直接的功能是关闭流，通过 `close` 方法来完成。在 Java 7 中，应该尽量通过 1.5 节介绍的 `try-with-resources` 语句来使用流，可以避免显式调用 `close` 方法。

第二个流控制功能是跳过指定数目的字节，相当于把流中的当前读取位置往后移动若干个字节。这个功能是通过 `skip` 方法来实现的。由于跳过若干个字节后，可能就已经到达了流的末尾，因此 `skip` 方法并不总能正确跳过指定数目的字节。调用者应该检查 `skip` 方法的返回值来获取实际跳过的字节数。并不是所有 `InputStream` 类的子类都支持 `skip` 方法。

第三个流控制功能是流的标记（`mark`）与重置（`reset`）。标记与重置配合起来使用，可以实现流中部分内容的重复读取，而不会像一般的读取操作那样，数据流过去之后就无法再次读取。简单来说，标记操作负责在流的当前读取位置做一个记号。当进行重置操作时，流的当前读取位置会被移动到上次标记的位置，这样就可以从上次标记位置开始再次进行读取操作。不是所有的流都支持标记功能，因此在使用 `mark` 方法来标记当前位置之前，需要通过 `markSupported` 方法来判断当前流的实现是否支持标记功能。在使用 `mark` 方法进行标记时，需要指定一个整数来表示允许重复读取的字节数。例如，标记时使用的是“`mark(1024)`”，那么在调用了 `reset` 之后，就只能从之前标记的位置开始再次重复读取最多 1024 字节。一般的内部实现方式是在标记之后把读取到的字节先保存起来。当重置之后，再调用 `read` 方法读取的就是之前保存的数据。

除了上面介绍的流控制方法之外，`InputStream` 类的最后一个方法是 `available`。这个方法与前面提到的 `InputStream` 类的对象进行读取操作时的阻塞特性相关。当 `read` 方法被调用，且当前流中并没有立即可用的数据时，这个调用操作会被阻塞，直到当前流成功地完成数据的准备为止。而 `available` 方法的作用在于告诉流的使用者，在不产生阻塞的情况下，当前流中还有多少字节可供读取。如果每次只读取调用 `available` 方法获取到的字节数，那么读取操作肯定不会被阻塞。这种非阻塞的特性在某些场合可能是很有作用的，比如在读取一个大文件的同时对文件的内容进行处理，如果每次读取时都不发生阻塞，就可以比较好地平衡数据读取和处理的时间。

3.1.2 基本输出流

与 `InputStream` 类相对应的 `OutputStream` 类表示的是基本的输出流，用来把数据从程序中输出到其他地方。基本的 `OutputStream` 类的对象也是在字节这个层次上进行操作的。其中最主要的是写入数据的 `write` 方法。同 `InputStream` 类中的 `read` 方法一样，`write` 方法也有 3 种类似的重载形式，可以每次写入一个字节，也可以写入一个字节数组中的全部或部分内容。

而在流的控制方面，`OutputStream` 类除了关闭流的 `close` 方法之外，还有一个 `flush`

方法用来强制要求 OutputStream 类的对象对暂时保存在内部缓冲区中的内容立即进行实际的写入操作。有些 OutputStream 类的子类会在内部维护一个缓冲区，通过 write 方法写入的数据会被首先存放在这个缓冲区中，然后在某个合适的时机再一次性地执行已缓冲的内容的实际写入操作。这种实现方式的出发点是为了性能考虑，减少实际的写入操作次数。在通常的使用场景中，OutputStream 类的对象的使用者一般不需要直接调用 flush 方法来保证内部缓冲区的数据被成功写入。这是因为当 OutputStream 类的对象的内部的缓冲区满了之后，会自动执行实际的写入操作。同时在 OutputStream 类的对象被关闭时，flush 方法一般也会被自动调用。

3.1.3 输入流的复用

输入流的复用其实有些自我矛盾的应用场景。一方面，在实际应用中，很多需要提供输入数据的 API 都使用 InputStream 类作为其参数的类型，比如 XML 文档的解析 API 就是一个典型的例子。同时很多数据的提供者允许使用者通过 InputStream 类的对象的方式来读取其数据，比如通过 java.net.HttpURLConnection 类的对象打开一个网络连接之后，可以得到用来读取其中数据的 InputStream 类的对象。如果每个这样的数据源仅有一个接收者，处理起来比较简单；如果有多个接收者，那么就有些复杂。主要的原因在于，从另一个方面出发，按照流本身所代表的抽象含义，数据一旦流过去，就无法被再次使用。如果直接把一个 InputStream 类的对象传递给一个接收者使用之后再传递给另外一个接收者，后者不能读取到流中的任何数据，因为流的当前读取位置已经到了末尾。这其实可以理解成数据的使用方式和数据本身的区别。InputStream 类表示的是数据的使用方式，而并不是数据本身。两者的根本区别在于每个 InputStream 类的对象作为 Java 虚拟机中的对象，是有其内部状态的，无法被简单地复用；而纯粹的数据本身是无状态的。在实际开发中，需要复用一个输入流的场景是比较的。比如，通过 HTTP 连接获取到的 XML 文档的输入流，可能既要进行合法性检验，又要解析文档内容，还有可能要保存到磁盘中。这些操作都需要直接接收同一个输入流。

对于现实应用中存在的对输入流复用的需求，基本上来说有两种方式可以解决：第一种是利用输入流提供的标记和重置的控制能力，第二种则是把输入流转换成数据来使用。

对于第一种解决方案来说，需要用到 java.io 包中的 InputStream 类的子类 java.io.BufferedInputStream。正如这个类名的含义一样，BufferedInputStream 类在 InputStream 类的基础上使用内部的缓冲区来提升性能，同时提供了对标记和重置的支持。BufferedInputStream 类属于过滤流的一种，在创建时需要传入一个已有的 InputStream 类的对象作为参数。BufferedInputStream 类的对象则在这个已有的 InputStream 类的对象的基础上提供额外的增强能力。

使用 BufferedInputStream 类之后，对流进行复用的过程就变得简单清楚。只需要

在流开始的地方进行标记，当一个接收者读取完流中的内容之后，再进行重置即可。重置完成之后，流的当前读取位置又回到了流的开始，就可以再次使用。代码清单 3-1 中给出了一个示例。对于一个 InputStream 类的子类的对象来说，如果它本来就支持标记，那么不再需要用 BufferedInputStream 类进行包装。在使用流之前，首先调用 mark 方法来进行标记。这里设置的标记在重置之后允许读取的字节数是整数的最大值，即 Integer.MAX_VALUE，这是为了能够复用整个流的全部内容。当流的接收者使用完流之后，需要显式地调用 markUsed 方法来发出通知，以完成对流的重置。

代码清单 3-1 使用 BufferedInputStream 类进行流复用的示例

```
public class StreamReuse {
    private InputStream input;
    public StreamReuse(InputStream input) {
        if (!input.markSupported()) {
            this.input = new BufferedInputStream(input);
        } else {
            this.input = input;
        }
    }

    public InputStream getInputStream() {
        input.mark(Integer.MAX_VALUE);
        return input;
    }

    public void markUsed() throws IOException {
        input.reset();
    }
}
```

第二种复用流的方案是直接把流中的全部数据读取到一个字节数组中。在不同的流的接收者之间的数据传递都是通过这个字节数组来完成的，而不再使用原始的 InputStream 类的对象。从一个字节数组得到一个 InputStream 类的对象是很容易的事情，只需要从该字节数组上创建一个 java.io.ByteArrayInputStream 类的对象就可以了。完整的实现如代码清单 3-2 所示。在创建 SavedStream 类的对象时，作为参数传递的 InputStream 类的对象中的数据首先被写入到一个 java.io.ByteArrayOutputStream 类的对象中，再把得到的字节数组保存下来。

代码清单 3-2 通过保存流的数据进行流复用的示例

```
public class SavedStream {
    private InputStream input;
    private byte[] data = new byte[0];

    public SavedStream(InputStream input) throws IOException {
        this.input = input;
    }
```

```

        save();
    }

    private void save() throws IOException {
        ByteArrayOutputStream output = new ByteArrayOutputStream();
        byte[] buffer = new byte[1024];
        int len = -1;
        while ((len = input.read(buffer)) != -1) {
            output.write(buffer, 0, len);
        }
        data = output.toByteArray();
    }

    public InputStream getInputStream() {
        return new ByteArrayInputStream(data);
    }
}

```

实际上，这两种复用流的做法在实现上的思路是一样的，都是预先把要复用的数据保存起来。`BufferedInputStream` 类在内部有一个自己的字节数组来维护标记位置之后可供读取的内容，与第二种做法中的字节数组的作用是一样的。

3.1.4 过滤输入输出流

在基本的输入输出流之上，`java.io` 包还提供了多种功能更强的过滤输入输出流。这些过滤流所提供的增强能力各不相同，比如前面提到的 `BufferedInputStream` 类和 `BufferedOutputStream` 类使用了内部的缓冲区来提高读写操作时的性能。另外一组过滤流 `DataInputStream` 类和 `DataOutputStream` 类在基本的字节流基础上提供了对读取和写入 Java 基本类型的支持。如果使用基本的字节流来操作 Java 中基本的整数、浮点数和字符串等类型的数据，需要开发人员自己完成这些数据类型与字节数组之间的转换工作。这个转换工作是平台相关的，并不是非常简单就能完成的。比如在读取和写入时需要考虑字节顺序，大端表示（big-endian）和小端表示（little-endian）的差别是很大的。在使用 `DataInputStream` 类时，可以通过 `readInt`、`readFloat` 和 `readUTF` 等方法来读取基本数据类型；在使用 `DataOutputStream` 类时，可以通过 `writeInt`、`writeFloat` 和 `writeUTF` 等方法来进行相应的写入操作。为了保证数据的正确性，对同样类型数据的写入和读取操作需要配对完成，这也是数据的提供者和消费者之间的契约。

除了读写基本数据类型的 `DataInputStream` 类和 `DataOutputStream` 类之外，`ObjectInputStream` 类和 `ObjectOutputStream` 类在基本数据类型的基础上增加了读写 Java 对象的支持。可以把一个 Java 对象的内部状态写入到输出流中，还可以从输入流中直接创建 Java 对象。这是一种实用的对象持久化的实现方式。在第 10 章介绍 Java 对象序列化时，会深入讨论 `ObjectInputStream` 类和 `ObjectOutputStream` 类的使用。

在有些时候，当从输入流中读取了某些数据之后，希望把这些数据又放回输入流中，以便下次可以重新读取。这种重复读取的动机与 3.1.3 节中提到的流的复用并不相同。将数据放回输入流是为了实现流的前瞻功能。有些情况下，在处理流的时候需要查看流中当前剩下的内容以确定是否继续读取。如果不符条件，就不能继续读取。为了查看这些内容，需要先读取到这些内容。但是一旦读取过了，再次读取的时候就无法获取到这些内容了，相当于有些内容丢失了。过滤流 `java.io.PushbackInputStream` 类可以解决这个问题，其中最关键的方法是 `unread`，这个方法可以把一个或多个字节放回输入流中，下次再读取时，会首先读取被放回去的内容。

3.1.5 其他输入输出流

在 `java.io` 包中，还有一些实用的输入输出流的实现，比如进行文件读写操作的 `FileInputStream` 类和 `FileOutputStream` 类，作为字节数组和流之间的桥梁的 `ByteArrayInputStream` 类和 `ByteArrayOutputStream` 类。这两对输入输出流的使用比较简单，这里不再赘述。

使用过 UNIX 和 Linux 操作系统的开发人员对使用命令行工具时可用的管道操作符（“|”）可能都不陌生。每个命令行工具都接收一定的输入数据，完成处理之后再产生相应的输出结果。通过管道操作符可以把一个命令行工具的输出作为另一个工具的输入，从而使它们级联起来，可以简洁地实现复杂的功能。Java 中的 `java.io.PipedInputStream` 类和 `java.io.PipedOutputStream` 类就是这样一对通过管道方式连接在一起的输入和输出流。一个 `PipedInputStream` 类的对象和一个 `PipedOutputStream` 类的对象连接在一起之后，通过 `PipedOutputStream` 类的对象所写入的数据可以在 `PipedInputStream` 类的对象中读取到。两者的连接既可以通过构造方法来完成，也可以通过 `connect` 方法来实现。从设计的角度来说，这实际上实现了典型的数据生产者 - 消费者模式。不过需要注意的是，使用 `PipedInputStream` 类和 `PipedOutputStream` 类的对象要在不同的线程之中，否则容易出现死锁的问题。

另外一个特殊的输入流是 `java.io.SequenceInputStream` 类。它可以把多个输入流按顺序连接起来，形成一个完整的输入流。调用 `SequenceInputStream` 类的对象的 `read` 方法会依次读取底层输入流中的内容。当一个输入流中的内容读取完毕之后，再换下一个输入流进行读取，直到所有底层输入流都读取完毕。`SequenceInputStream` 类的作用相当于多个 `InputStream` 类的对象的连接操作。

3.1.6 字符流

前面介绍的基本 `InputStream` 类和 `OutputStream` 类以及包装它们的各种过滤流实现都是在字节层次上操作的。字节流主要由机器来处理。而对于程序的用户来说，他们更愿意看到直接可读的字符。在 `java.io` 包中还有一组类用来处理字符流，即 `java.`

io.Reader 类和 java.io.Writer 类及其子类。这些字符流处理的是字符类型，而不是字节类型。字符流适合用于处理程序中包含的文本类型的内容。

创建一个字符流的最常见做法是通过一个字节流 InputStream 类或 OutputStream 类的对象进行创建，对应的是 InputStreamReader 类和 OutputStreamWriter 类。在从字节流转换成字符流时，需要指定字符的编码格式。关于字符的编码和解码，在第 4 章会有详细的介绍。如果编码格式错误，会产生包含乱码的字符串。在创建 InputStreamReader 类和 OutputStreamWriter 类的对象时，总是应该显式地指定一个字符编码格式。如果不指定，使用的是底层 Java 平台的默认编码格式。这可能造成程序在不同运行平台上的兼容性问题。字符流也可以从 String 类的对象中创建，即使用 java.io.StringReader 类和 java.io.StringWriter 类进行创建；还可以从字符数组中创建，即使用 java.ioCharArrayReader 类和 java.io.CharArrayWriter 类进行创建。java.io.BufferedReader 类和 java.io.BufferedWriter 类用来为已有的 Reader 类和 Writer 类的对象提供内部缓冲区的支持，以提高读写操作的性能。

在使用字符流处理文本内容时，要注意那些在内容中声明了编码格式的文本格式。这其中最典型的例子是 XML 文档。XML 文档一般通过处理指令来声明其内容的编码格式，如处理指令 “`<?xml version="1.0" encoding="UTF-8" ?>`” 声明了文档使用的是 UTF-8 编码格式。当处理指令中不包含编码格式声明时，XML 处理器有自己的一套编码格式识别算法。对于这种类型的文档，不应该使用字符流来处理，而是应该使用字节流。使用字节流时不会对原始数据造成影响，能够保证文档处理时的正确性。

在开发中经常会遇到的一个场景是把 InputStream 类的对象中的内容转换成一个 String 类的对象。对于这种情况，典型的做法是从 InputStream 类的对象中创建一个对应的 InputStreamReader 类的对象，并循环读取到一个 char 数组中，再把 char 数组的内容添加到一个 java.lang.StringBuilder 类的对象中，最后把 StringBuilder 类的对象转换成一个 String 类的对象即可。如果希望提高性能，可以使用一个 BufferedReader 类的对象来包装 InputStreamReader 类的对象，再调用 readLine 方法每次读取一行数据。使用 java.util.Scanner 类也可以完成类似的转换。还可以使用 Apache Commons IO 库中的 org.apache.commons.io.IOUtils 类的 `toString` 方法。

3.2 缓冲区

从前面对 Java I/O 中流的概念和实现的介绍可以看出，Java 中流的实现采用了一种简单而朴素的做法，即以字节流为基础，在字节流上再通过过滤流来满足不同的需要。对于开发人员来说，流加上字节数组的使用方式的抽象层次较低，使用起来比较繁琐。这其中比较麻烦的一点在于字节数组的长度是不可变的。一旦创建了某个长度的字节数组，当数据过多以至于超出数组长度的限制时，需要开发人员自己来进行管理，比如重新创建一个新的长度更大的数组，然后再把之前的数据复制进去。一种可行的做法是利

用 `ByteArrayOutputStream` 类，不断地向 `ByteArrayOutputStream` 类的对象中写入数据，写入完成之后，用它的 `toByteArray` 方法可以得到一个字节数组。但这种做法缺乏足够的灵活性，性能也比较差。更好的做法是使用 Java NIO 中新的缓冲区实现。

3.2.1 基本用法

Java NIO 中的缓冲区在某些特性上类似于 Java 中的基本类型的数组（如字节数组或整型数组等），比如缓冲区中的数据排列是线性的，缓冲区的空间也是有限的。不过两者的差别也是显著的，最主要的区别在于缓冲区所提供的功能远比数组丰富得多，而且也支持存储类型异构的数据。要理解缓冲区的使用，就需要理解缓冲区的 3 个状态变量，分别是容量（capacity）、读写限制（limit）和读写位置（position）。使用缓冲区时产生的错误，绝大多数都源自错误地理解了这 3 个变量的含义。

首先最容易理解的状态变量是缓冲区的容量。容量指的是缓冲区的额定大小。容量是在创建缓冲区时指定的，无法在创建后更改。在任何时候缓冲区中的数据总数都不可能超过容量。第二个变量是读写限制，表示的是在缓冲区中进行读写操作时的最大允许位置。比如对于一个容量为 32 的缓冲区来说，如果设置其读写限制的值是 16，那么就只有前半个缓冲区在读写时是可用的。如果希望后半个缓冲区也能进行读写操作，就必须把读写限制设置为 32。最后一个变量是读写位置，表示的是当前进行读写操作时的位置。当在缓冲区中进行相对读写操作时，在这个位置上进行。对于这 3 个变量，除了只能获取容量外，读写限制和读写位置都有相应的获取和设置的方法，分别是 `limit` 和 `position`，其中不带参数的重载形式用来获取值，而带参数的形式用来设置值。对于一个缓冲区来说，它的当前可以使用的范围是在读取位置和读取限制之间的这一段区域。通过 `remaining` 方法可以获取到这段可用范围的长度。

缓冲区同样也支持标记和重置的特性，类似于前面提到的流的标记和重置。当调用 `mark` 方法时，会在当前的读写位置上设置一个标记。在调用 `reset` 方法之后，会使得读写位置回到上一次 `mark` 方法设置的位置上。进行标记时的位置不能超过当前的读写位置。如果通过 `position` 方法重新设置了读写位置，而使之设置的标记的位置超出了新的读写位置的范围，那么该标记就会失效。在任何时候，缓冲区中的各个状态变量之间满足关系“ $0 \leq \text{标记位置} \leq \text{读写位置} \leq \text{读写限制} \leq \text{容量}$ ”。

Java NIO 中的 `java.nio.Buffer` 类是所有不同数据类型的缓冲区的父类。一般来说，通过调用缓冲区对象的 `limit` 和 `position` 方法就可以满足大部分的需求。不过对于某些常见的场景，`Buffer` 类也提供了快捷的方法。当复用一个已有的缓冲区时，如果希望向缓冲区中写入数据，可以调用 `clear` 方法，该方法会把读写限制设为缓冲区的容量，同时把读写位置设为 0；当需要读取一个缓冲区中的数据时，可以调用 `flip` 方法，该方法会把读写限制设为当前的读写位置，再把读写位置设为 0，这样可以保证缓冲区中的全部数据都可以被读取；当希望重新读取缓冲区中的数据的时候，可以调用 `rewind` 方法，该

方法不会改变读写限制，但是会把读写位置设为 0。在后面的示例代码中，可以看到这几个方法在缓冲区读写操作时的使用模式。熟悉这些模式，就会避免出现一些常见的错误。

缓冲区进行的读写操作分成两类：一类是根据当前读写位置进行的相对读写操作，另外一类是根据在缓冲区中的绝对位置进行的读写操作。两者的差别在于相对读写会改变当前读写位置，而绝对读写则不会。

3.2.2 字节缓冲区

对于 Java 中的基本类型，除了布尔类型之外，都有对应的缓冲区实现，用来存储此类型的数据。布尔类型的缓冲区可以很容易地用字节类型来替代。这些缓冲区类型中最重要的实现类是 `java.nio.ByteBuffer` 类。在 `ByteBuffer` 类中，除了可以对基本的字节进行操作之外，还可以操作其他基本类型的数据。对于字节数据，除了每次操作单个字节之外，还支持批量式处理一个字节数组。代码清单 3-3 中给出了使用 `ByteBuffer` 类的示例。在创建 `ByteBuffer` 类的对象时，只能通过其静态工厂方法 `allocate` 来分配新空间，或者通过 `wrap` 方法来包装一个已有的字节数组。在创建 `ByteBuffer` 类的对象时需要指定缓冲区的容量。在创建完成之后，可以通过 `put` 方法向缓冲区中添加数据，而 `get` 方法则从其中读取数据。进行绝对读写操作的 `put` 和 `get` 方法的第一个参数都是读写位置的序号。需要注意的是，这个序号是根据字节数来进行计算的。如果在写入数据时使用的是类似 `putChar` 或 `putLong` 这样的操作基本数据类型的方法，在通过相应的 `getChar` 或 `getLong` 方法来读取的时候，需要开发人员自己来计算起始字节的位置。如果计算错误，那么得到的就不是当时写入的数据。如果 `ByteBuffer` 类的对象中存放的是不同类型的数据，那么这种计算是无法避免的。如果 `ByteBuffer` 类的对象中存放的是相同的类型，可以考虑使用对应类型的缓冲区实现类，或是从 `ByteBuffer` 类的对象中创建相应的视图。

代码清单 3-3 `ByteBuffer` 类的使用示例

```
public void useByteBuffer() {
    ByteBuffer buffer = ByteBuffer.allocate(32);
    buffer.put((byte)1);
    buffer.put(new byte[3]);
    buffer.putChar('A');
    buffer.putFloat(0.0f);
    buffer.putLong(10, 100L);
    buffer.getChar(4); // 值为 'A'
}
```

由于 `ByteBuffer` 类支持对基本数据类型的处理，因此必须要考虑字节顺序。同样的字节序列按照不同的顺序去解释，所得到的结果是不同的。`java.nio.ByteOrder` 类中定义了两种最基本的字节顺序：`BIG_ENDIAN` 对应的大端表示和 `LITTLE_ENDIAN` 对应的小端表示。大端表示的含义是字节序列中高位在前，而小端表示则正好相反。`ByteOrder`

类中的静态方法 nativeOrder 可以获取到底层操作系统平台采用的字节顺序。ByteBuffer 类的对象默认使用的是大端表示。代码清单 3-4 中给出了一个通过修改字节顺序来改变基本类型的值的示例。

代码清单 3-4 字节缓冲区的字节顺序

```
public void byteOrder() {
    ByteBuffer buffer = ByteBuffer.allocate(4);
    buffer.putInt(1);
    buffer.order(ByteOrder.LITTLE_ENDIAN);
    buffer.getInt(0); // 值为 16777216
}
```

ByteBuffer 类支持的另外一个操作是压缩。压缩操作的一个典型的应用场景是把 ByteBuffer 类的对象作为数据传输时的缓冲区来使用。例如，有一个发送者不断地向 ByteBuffer 类的对象中填充数据，而另外一个接收者从相同的 ByteBuffer 类的对象中不断地获取数据。发送者可能用数据填充满了 ByteBuffer 类的对象中读写限制范围之内的全部可用空间，而接收者却暂时只读取了 ByteBuffer 类的对象中的一部分数据。接收者在完成读取之后要使用 compact 方法进行压缩操作。压缩操作就是把 ByteBuffer 类的对象中当前读写位置到读写限制范围内的数据复制到内部存储空间中的最前面，然后再把读写位置移动到紧接着复制完成的数据的下一个位置，读写限制也被设置成 ByteBuffer 类的对象的容量。经过压缩之后，发送者的下次写入操作就不会覆盖接收者还没读取的内容，而接收者每次总是可以从 ByteBuffer 类的对象的起始位置进行读取。代码清单 3-5 中给出了压缩操作的使用示例。

代码清单 3-5 字节缓冲区的压缩操作的示例

```
public void compact() {
    ByteBuffer buffer = ByteBuffer.allocate(32);
    buffer.put(new byte[16]);
    buffer.flip();
    buffer.getInt(); // 当前读取位置为 4
    buffer.compact();
    int pos = buffer.position(); // 值为 12
}
```

在代码清单 3-5 中，经过 put、flip 和 getInt 等方法调用之后，ByteBuffer 类的对象的当前读取位置是 4，而读取限制是 16，所以在压缩的时候，没有被读取的 12 个字节会被复制到 ByteBuffer 类的对象的内部存储空间的开头位置，同时当前读取位置变为被复制的字节数，即 12。

ByteBuffer 类的实现分为直接缓冲区和非直接缓冲区两种。在直接缓冲区中进行 I/O 操作时，会尽可能避免多余的数据复制，而直接使用操作系统底层的 I/O 操作来完成。这样可以提升读写操作时的性能，不过也带来了额外的创建和销毁时的代价。直接缓冲

区一般是常驻内存的，会增加程序的内存开销，所以直接缓冲区一般只用在对性能要求很高的情况下。通过 ByteBuffer 类的静态方法 allocateDirect 可以创建新的直接缓冲区，这类似于 allocate 方法的使用。

3.2.3 缓冲区视图

ByteBuffer 类的另外一个常见的使用方式是在一个已有的 ByteBuffer 类的对象上创建出各种不同的视图。这些视图和它所基于的 ByteBuffer 类的对象共享同样的存储空间，但是提供额外的实用功能。在功能上，ByteBuffer 类的视图与它所基于的 ByteBuffer 类的对象之间的关系类似于 3.1.4 节介绍的过滤流和它所包装的流的关系。正因为这种共享存储空间的特性，在视图中对数据所做的修改会反映在原始的 ByteBuffer 类的对象中。

最常见的 ByteBuffer 类的视图是转换成对基本数据类型进行操作的缓冲区对象。这些缓冲区包括 CharBuffer、ShortBuffer、IntBuffer、LongBuffer、FloatBuffer 和 DoubleBuffer 等 Java 类。从这些缓冲区的类名就可以知道所操作的数据类型。ByteBuffer 类提供了对应的方法来完成相应的转换，如 asIntBuffer 方法在当前 ByteBuffer 类的对象的基础上创建一个新的 IntBuffer 类的视图。新创建的视图和原始的 ByteBuffer 类的对象所共享的不一定是全部的空间，而只是从 ByteBuffer 类的对象中的当前读写位置到读写限制之间的可用空间。在这个空间范围内，不论是在 ByteBuffer 类的对象中还是在作为视图的新缓冲区中，对数据所做的修改，对另一个来说都是可见的。除了数据本身之外，两者的读写位置、读写限制和标记位置等都是相互独立的。在代码清单 3-6 中，创建视图的时候，两者所共享的是序号 4~32 的空间。在 IntBuffer 类的对象中所做的修改，对于原始的 ByteBuffer 类的对象也是可见的。ByteBuffer 类的基本数据类型视图在开发中的使用场景比较多，这主要是因为很多 I/O 相关的 API 都使用 ByteBuffer 类作为参数类型，而 ByteBuffer 类的视图可以很方便地对内容进行操作。

代码清单 3-6 字节缓冲区的视图

```
public void viewBuffer() {
    ByteBuffer buffer = ByteBuffer.allocate(32);
    buffer.putInt(1); // 读取位置为 4
    IntBuffer intBuffer = buffer.asIntBuffer();
    intBuffer.put(2);
    int value = buffer.getInt(); // 值为 2
}
```

除了基本类型的缓冲区视图之外，另外一类视图是类型相同的 ByteBuffer 类的对象。通过 slice 方法可以创建一个当前 ByteBuffer 类的对象的视图，其行为类似于通过 asIntBuffer 方法创建的视图，只不过得到的是 ByteBuffer 类的对象。而 duplicate 方法则用来完全复制一个 ByteBuffer 类的对象。这个方法与 slice 方法的区别在于，duplicate 方

法并不考虑 ByteBuffer 类的对象的当前读写位置和读写限制，只是简单地全部复制。方法 asReadOnlyBuffer 的行为类似于 duplicate 方法，只不过得到的 ByteBuffer 类的对象是只读的，不能执行写入操作。

对于其他基本类型的缓冲区来说，除了通过 ByteBuffer 类的视图来创建之外，也可以通过对称类的 allocate 方法来直接创建。这些缓冲区类与 ByteBuffer 类的最显著区别在于，其中的容量、读写位置和读写限制都是根据基本类型的个数来计算的，而不是根据字节数计算的。如通过 “IntBuffer.allocate(32)” 创建的整型缓冲区的容量是 32 个整数，而不是 32 个字节。另外，不能直接创建出除 ByteBuffer 类之外的其他类型的直接缓冲区，只能先创建 ByteBuffer 类型的直接缓冲区，再创建相应的基本类型视图。

3.3 通道

通道是 Java NIO 对 I/O 操作提供的另外一种新的抽象方式。通道不是从 I/O 操作所处理的数据这个层次上去抽象，而是表示为一个已经建立好的到支持 I/O 操作的实体的连接。这个连接一旦建立，就可以在这个连接上进行各种 I/O 操作。在通道上所进行的 I/O 操作的类型取决于通道的特性，一般的操作包括数据的读取和写入等。在 Java NIO 中，不同的实体有不同的通道实现，比如文件通道和网络通道等。通道在进行读写操作时使用的都是 3.2 节介绍的新的缓冲区的实现，而不是字节数组。

Java NIO 中的通道都实现了 java.nio.channels.Channel 接口。Channel 接口本身很简单，只有关闭通道的 close 方法和判断通道是否被打开的 isOpen 方法。由于 Channel 接口继承了 java.lang.AutoCloseable 接口，通道的所有实现对象都可以用在 try-with-resources 语句中，方便了对通道的使用。从 API 设计的角度来说，Java NIO 更多地采用了面向接口的设计思路，很多功能都被抽象到不同的接口中。

对于一个支持读取操作的通道来说，应该实现 java.nio.channels.ReadableByteChannel 接口。这个接口只有一个 read 方法来读取数据。读取的时候把数据读取到一个 ByteBuffer 类的对象中。在读取的时候，数据的填充从 ByteBuffer 类的对象的当前读写位置开始，直到写入到读写限制所指定的位置为止。类似的 java.nio.channels.WritableByteChannel 接口用来进行数据的写入。该接口的 write 方法也使用 ByteBuffer 类的对象作为参数。写入时的数据来源也与 ReadableByteChannel 接口中 read 方法类似，取决于 ByteBuffer 类的对象中的可用字节。

有些通道除了支持读写操作之外，还支持移动读写操作的位置。这种通道一般实现 java.nio.channels.SeekableByteChannel 接口，该接口中的 position 方法用来获取和设置当前的读写位置，而 truncate 方法则将通道对应的实体截断。如果调用 truncate 方法时指定的新的长度值小于实体的当前长度，那么实体被截断成指定的新长度。另外的一个方法 size 可以获取实体的当前长度。

另外一个实用的通道接口是 java.nio.channels.ScatteringByteChannel。这个接口的

read 方法不同于 ReadableByteChannel 接口中的 read 方法，支持使用一个 ByteBuffer 类的对象的数组作为参数。在进行读取操作时，从通道对应的实体中得到的数据被依次写入这些 ByteBuffer 类的对象中。向每个 ByteBuffer 类的对象中写入的字节数是该 ByteBuffer 类的对象中当前可用的字节数。与 ScatteringByteBuffer 接口对应的是 java.nio.channels.GatheringByteBuffer 接口，这个接口用来将多个 ByteBuffer 类的对象包含的数据同时写入到通道中。

3.3.1 文件通道

文件是 I/O 操作的一个常见实体。与文件实体对应的表示文件通道的 java.nio.channels.FileChannel 类也是一个功能强大的通道实现。FileChannel 类除了可以进行读写操作之外，还实现了前面介绍的大部分接口，最主要的是实现了 SeekableByteChannel 接口。除了这些接口之外，FileChannel 类还提供了一些与文件操作相关的特有功能。这些功能会在后面进行介绍。

1. 基本用法

在使用文件通道之前，首先需要获取到一个 FileChannel 类的对象。FileChannel 类的对象既可以通过直接打开文件的方式来创建，也可以从已有的流中得到。通过直接打开文件来创建文件通道的方式是 Java 7 中新增的。代码清单 3-7 给出了一个简单的打开文件通道并写入数据的示例。FileChannel 类的 open 方法用来打开一个新的文件通道。调用时的第一个参数是要打开的文件的路径，第二个参数是打开文件时的选项。不同的选项会对通道的能力产生影响。比如，当一个文件通道以只读的方式打开时，就不能通过 write 方法来写入数据。

代码清单 3-7 打开文件通道并写入数据的示例

```
public void openAndWrite() throws IOException {
    FileChannel channel = FileChannel.open(Paths.get("my.txt"),
        StandardOpenOption.CREATE, StandardOpenOption.WRITE);
    ByteBuffer buffer = ByteBuffer.allocate(64);
    buffer.putChar('A').flip();
    channel.write(buffer);
}
```

在打开文件通道时可以选择的选项有很多，其中最常见的是读取和写入模式的选择，分别通过 java.nio.file.StandardOpenOption 枚举类型中的 READ 和 WRITE 来声明。CREATE 表示当目标文件不存在时，需要创建一个新文件；而 CREATE_NEW 同样会创建新文件，区别在于如果文件已经存在，则会产生错误；APPEND 表示对文件的写入操作总是发生在文件的末尾处，即在文件的末尾添加新内容；当声明了 TRUNCATE_EXISTING 选项时，如果文件已经存在，那么它的内容将被清空；DELETE_ON_CLOSE 用在需要创建临时文件的时候。声明了这个选项之后，当文件通道关闭时，Java 虚拟机

会尽力尝试去删除这个文件。

另外一种创建文件通道的方式是从已有的 FileInputStream 类、 FileOutputStream 类和 RandomAccessFile 类的对象中得到。这 3 个类都有一个 getChannel 方法来获取对应的 FileChannel 类的对象，所得到的 FileChannel 类的对象的能力取决于其来源流的特征。对 InputStream 类的对象来说，它所得到的 FileChannel 类的对象是只读的，而 FileOutputStream 类的对象所得到的通道是可写的，RandomAccessFile 类的对象所得到的通道的能力则取决于文件打开时的选项。

对于 FileChannel 类所实现的来自 SeekableByteChannel、ScatteringByteChannel 和 GatheringByteChannel 接口的方法，这里不再赘述。下面要介绍的是 FileChannel 类中独有的方法。首先介绍在文件通道的任意位置进行读写的能力。调用 ReadableByteChannel 接口中的 read 方法和 WritableByteChannel 接口中的 write 方法都只能进行相对读写操作。而对于 FileChannel 类来说，得益于文件本身的特性，可以在任意绝对位置进行读写操作，只需额外传入一个参数来指定读写的位置即可。在代码清单 3-8 中，对于一个新创建的文件，同样可以指定任意的写入位置。文件的大小会根据写入的位置自动变化。

代码清单 3-8 对文件通道的绝对位置进行读写操作的示例

```
public void readWriteAbsolute() throws IOException {
    FileChannel channel = FileChannel.open(Paths.get("absolute.txt"),
        StandardOpenOption.READ, StandardOpenOption.CREATE, StandardOpenOption.
        WRITE);
    ByteBuffer writeBuffer = ByteBuffer.allocate(4).putChar('A').putChar('B');
    writeBuffer.flip();
    channel.write(writeBuffer, 1024);
    ByteBuffer readBuffer = ByteBuffer.allocate(2);
    channel.read(readBuffer, 1026);
    readBuffer.flip();
    char result = readBuffer.getChar(); // 值为 'B'
}
```

2. 文件数据传输

在使用文件进行 I/O 操作时的一些典型场景包括把来自其他实体的数据写入文件中，以及把文件中的内容读取到其他实体中，按照通道的概念来说，就是文件通道和其他通道之间的数据传输。对于这种常见的需求，FileChannel 类提供了 transferFrom 和 transferTo 方法用来快速地传输数据，其中 transferFrom 方法把来自一个实现了 ReadableByteChannel 接口的通道中的数据写入文件通道中，而 transferTo 方法则把当前文件通道的数据传输到一个实现了 WritableByteChannel 接口的通道中。在进行这两种方式的数据传输时都可以指定当前文件通道中的传输的起始位置和数据长度。

使用 FileChannel 类中的这两个数据传输方法比传统的使用缓冲区进行循环读取的

做法要简单，性能也更好。这主要是因为这两个方法在实现中尽可能地使用了底层操作系统的支持。比如，当需要通过 HTTP 协议来获取一个网页的内容并保存在文件中时，可以使用代码清单 3-9 中的代码实现。

代码清单 3-9 使用文件通道保存网页内容的示例

```
public void loadWebPage(String url) throws IOException {
    try (FileChannel destChannel = FileChannel.open(Paths.get("content.txt"),
        StandardOpenOption.WRITE, StandardOpenOption.CREATE)) {
        InputStream input = new URL(url).openStream();
        ReadableByteChannel srcChannel = Channels.newChannel(input);
        destChannel.transferFrom(srcChannel, 0, Integer.MAX_VALUE);
    }
}
```

在代码清单 3-9 中，打开一个 HTTP 连接并获取到其对应的数据输入流之后，可以将其转换成一个通道，最后通过 transferFrom 方法来把通道中的内容写入文件中。这里使用了 try-with-resources 语句来简化对通道的使用。

文件通道中的这两个传输方法的另一个重要的好处是可以简洁和高效地实现文件的复制。在前面的章节中介绍过使用字节数组作为缓冲区的文件复制操作的基本实现方式。如果采用传统的循环读取的方式，使用新的 ByteBuffer 类会比字节数组简单一些，如代码清单 3-10 所示。使用 ByteBuffer 类的时候并不需要记录每次实际读取的字节数，但是要注意 flip 和 compact 方法的使用。

代码清单 3-10 使用字节缓冲区进行文件复制操作的示例

```
public void copyUseByteBuffer() throws IOException {
    ByteBuffer buffer = ByteBuffer.allocateDirect(32 * 1024);
    try (FileChannel src = FileChannel.open(Paths.get(srcFilename),
        StandardOpenOption.READ);
        FileChannel dest = FileChannel.open(Paths.get(destFilename),
        StandardOpenOption.WRITE, StandardOpenOption.CREATE)) {
        while (src.read(buffer) > 0 || buffer.position() != 0) {
            buffer.flip();
            dest.write(buffer);
            buffer.compact();
        }
    }
}
```

如果使用 FileChannel 类中的传输方法来实现，代码就更加简单了，如代码清单 3-11 所示，进行复制的逻辑只需要一行代码即可。

代码清单 3-11 使用文件通道进行文件复制的示例

```
public void copyUseChannelTransfer() throws IOException {
    try (FileChannel src = FileChannel.open(Paths.get(srcFilename),
        StandardOpenOption.READ);
        FileChannel dest = FileChannel.open(Paths.get(destFilename),
        StandardOpenOption.WRITE, StandardOpenOption.CREATE)) {
        src.transferTo(0, src.size(), dest);
    }
}
```

```

        StandardOpenOption.READ);
        FileChannel dest = FileChannel.open(Paths.get(destFilename),
            StandardOpenOption.WRITE, StandardOpenOption.CREATE)) {
    src.transferTo(0, src.size(), dest);
}
}

```

3. 内存映射文件

在对大文件进行操作时，性能问题一直比较难处理。通过操作系统的内存映射文件支持，可以比较快速地对大文件进行操作。内存映射文件的原理在于把系统的内存地址映射到要操作的文件上。读取这些内存地址就相当于读取文件的内容，而改变这些内存地址的值就相当于修改文件中的内容。被映射到内存地址上的文件在使用上类似于操作系统中使用的虚拟内存文件。通过内存映射的方式对文件进行操作时，不再需要通过 I/O 操作来完成，而是直接通过内存地址访问操作来完成，这就大大提高了操作文件的性能，因为 I/O 操作比访问内存地址要慢得多。

FileChannel 类的 map 方法可以把一个文件的全部或部分内容映射到内存中，所得到的是一个 ByteBuffer 类的子类 MappedByteBuffer 的对象，程序只需要对这个 MappedByteBuffer 类的对象进行操作即可。对这个 MappedByteBuffer 类的对象所做的修改会自动同步到文件内容中。代码清单 3-12 给出了使用文件通道的内存映射功能的一个示例。在进行内存映射时需要指定映射的模式，一共有 3 种可用的模式，由 FileChannel.MapMode 这个枚举类型来表示：READ_ONLY 表示只能对映射之后的 MappedByteBuffer 类的对象进行读取操作；READ_WRITE 表示是可读可写的；而 PRIVATE 的含义是通过 MappedByteBuffer 类的对象所做的修改不会被同步到文件中，而是被同步到一个私有的副本中。这些修改对其他同样映射了该文件的程序是不可见的。如果希望对 MappedByteBuffer 类的对象所做的修改被立即同步到文件中，可以使用 force 方法。

代码清单 3-12 内存映射文件的使用示例

```

public void mapFile() throws IOException {
    try (FileChannel channel = FileChannel.open(Paths.get("src.data"),
        StandardOpenOption.READ, StandardOpenOption.WRITE)) {
        MappedByteBuffer buffer = channel.map(FileChannel.MapMode.READ_WRITE, 0,
            channel.size());
        byte b = buffer.get(1024 * 1024);
        buffer.put(5 * 1024 * 1024, b);
        buffer.force();
    }
}

```

如果希望更加高效地处理映射到内存中的文件，把文件的内容加载到物理内存中是

一个好办法。通过 `MappedByteBuffer` 类的 `load` 方法可以把该缓冲区所对应的文件内容加载到物理内存中，以提高文件操作时的性能。由于物理内存的容量受限，不太可能直接把一个大文件的全部内容一次性地加载到物理内存中。可以每次只映射文件的部分内容，把这部分内容完全加载到物理内存中进行处理。完成处理之后，再映射其他部分的内容。

由于 I/O 操作一般比较耗时，出于性能考虑，很多操作在操作系统内部都是使用缓存的。在程序中通过文件通道 API 所做的修改不一定会立即同步到文件系统中。如果在没有同步之前发生了程序错误，可能导致所做的修改丢失。因此，在执行完某些重要文件内容的更新操作之后，应该调用 `FileChannel` 类的 `force` 方法来强制要求把这些更新同步到底层文件中。可以强制同步的更新有两类，一类是文件的数据本身的更新，另一类是文件的元数据的更新。在使用 `force` 方法时，可以通过参数来声明是否在同步数据的更新时也同步元数据的更新。

4. 锁定文件

当需要在多个程序之间进行数据交换时，文件通常是一种很好的选择。一个程序把产生的输出保存在指定的文件中，另外一个程序进行读取即可。双方只需要在文件的格式上达成一致就可以了，内部逻辑的实现都是独立的。但是在这种情况下，对这个文件的访问操作容易产生冲突，而且对两个独立的应用程序来说，也没有什么比较好的方式来实现操作的同步。对于这种情况，最好的办法是对文件进行加锁。在一个程序完成操作之前，阻止另外一个程序对该文件的访问。通过 `FileChannel` 类的 `lock` 和 `tryLock` 方法可以对当前文件通道所对应的文件进行加锁。加锁时既可以选择锁定文件的全部内容，也可以锁定指定的范围区间中的部分内容。`lock` 和 `tryLock` 两个方法的区别在于 `lock` 方法是阻塞式的，而 `tryLock` 方法则不是。当成功加锁之后，会得到一个 `FileLock` 类对象。在完成对锁定文件的操作之后，通过 `FileLock` 类的 `release` 方法可以解除锁定状态，允许其他程序来访问。`FileLock` 类表示的锁分共享锁和排它锁两类。共享锁不允许其他程序获取到与当前锁定范围相重叠的排它锁，而获取共享锁是允许的；排它锁不允许其他程序获取到与锁定范围相重叠的共享锁和排它锁。如果调用 `FileLock` 类的对象的 `isShared` 方法的返回值为 `true`，则表明是一个共享锁，否则是排它锁。

注意 对 `FileLock` 类表示的共享锁和排它锁的限制只发生在待锁定的文件范围与当前已有锁的范围发生重叠的时候。不同程序可以同时在一个文件上加上自己的排它锁，只要这些锁的锁定范围不互相重叠即可。

一个系统可能由 C++ 和 Java 等不同编程语言所编写的不同组件组成，这些组件可能共享一个配置文件。当 Java 程序要更新配置文件的时候，可以先锁定该文件，再进行更新。这样可以保证更新时内容的完整性。代码清单 3-13 给出了一个示例的使用模板。

代码清单 3-13 锁定文件的示例

```

public void updateWithLock() throws IOException {
    try (FileChannel channel = FileChannel.open(Paths.get("settings.config"),
        StandardOpenOption.READ, StandardOpenOption.WRITE);
        FileLock lock = channel.lock()) {
        // 更新文件内容
    }
}

```

对文件进行加锁操作的主体是当前的 Java 虚拟机，也就是说这个加锁的功能用来协同当前 Java 虚拟机上运行的 Java 程序和操作系统上运行的其他程序。对于 Java 虚拟机上运行的多线程程序，不能用这种机制来协同不同线程对文件的访问。

3.3.2 套接字通道

除了文件之外，另外一个在应用开发中经常需要处理的 I/O 实体是网络连接。Java 从 JDK 1.0 开始就有了处理网络连接的相关 API，包含在 `java.net` 包中。这其中比较重要的是套接字连接的相关 API。通过套接字 API 可以很容易地实现自己的网络服务器和客户端程序。

如果需要实现一个网络客户端程序，只需要先创建一个 `java.net.Socket` 类的对象，再连接到远程服务器。当连接成功之后，可以从 `Socket` 类的对象中获取到套接字连接对应的输入流和输出流。通过输入流可以得到服务器端发送的数据，而通过输出流可以向服务器端发送数据。对服务器端程序来说，则可以创建一个 `java.net.ServerSocket` 类的对象并使用其 `accept` 方法在指定的端口进行监听。调用方法 `accept` 时会处于阻塞状态，等待客户端程序的连接请求。当有新的连接建立时，`accept` 方法会返回一个与连接发起者进行通信时所使用的 `Socket` 类的对象。

从上面的简要描述可以看出，`java.net` 包中的套接字的相关实现现在使用时是比较简单的，所包含的概念也并不多，且结合了已有的 I/O 操作中流的概念。开发人员通过类比文件操作的方式，可以很容易理解套接字的使用。实际上，`java.net` 包中的套接字实现的最大问题不是来自于 API 本身，而是出于性能方面的考虑。`Socket` 类和 `ServerSocket` 类中提供的与建立连接和数据传输相关的方法都是阻塞式的，也就是说，如果操作没有完成，当前线程会处于等待状态。比如，通过调用 `Socket` 类的 `connect` 方法连接远程服务器时，如果由于网络的原因，连接一直没有办法成功建立，那么 `connect` 方法会一直阻塞下去，直到连接建立成功或出现超时错误。在这段等待时间内，其他代码是无法继续执行的。相对于同样是阻塞式的文件操作来说，网络操作的这个特性所带来的问题更为严重，这是因为网络操作的延迟远比文件操作中的延迟要长得多，而且影响网络操作速度的因素也更多。

为了提高网络服务器和客户端的性能和吞吐量，采用多线程的方式就成了解决这个问题的“银弹”。以服务器的实现为例，在一般情况下，会有一个线程专门用来调用

ServerSocket 类的 accept 方法来监听连接请求。一旦有新的连接建立，就会创建一个新的线程来专门处理这个请求。这种一个请求对应一个线程的方式，显然并不适合服务器负载压力比较大的情况，因为每个线程都要占用资源，创建线程也是有代价的。对此，一般又会引入线程池的实现，以能够复用已有的线程，从而减少每次都要新创建线程所带来的代价。采用多线程的方式确实能解决问题。当某个线程由于等待网络操作而阻塞时，其他线程还可以继续执行，整体的性能和吞吐量得到了提高。不过多线程方式的问题在于它太复杂了，而且容易出现非常多的隐含错误，多线程的相关内容会在第 11 章中进行讨论。

为了解决这些与网络操作相关的问题，Java NIO 提供了非阻塞式和多路复用的套接字连接，并在 Java 7 中又进行了改进。与文件操作一样，网络操作也被抽象成通道的概念，接口 `java.nio.channels.NetworkChannel` 表示的是一个套接字所对应的通道。

1. 阻塞式套接字通道

与 `Socket` 类和 `ServerSocket` 类相对应，Java NIO 中也提供了 `SocketChannel` 类和 `ServerSocketChannel` 类两种不同的套接字通道实现。这两种通道都支持阻塞和非阻塞两种模式。阻塞模式的使用简单，但是性能不是很好；非阻塞模式则正好相反。开发人员可以根据自己的需要来选择合适的模式。一般来说，低负载的程序可以选择阻塞模式，实现起来简单且性能足够好。代码清单 3-9 中给出的保存网页内容的示例程序，也可以通过 `SocketChannel` 类来实现，如代码清单 3-14 所示。先通过 `SocketChannel` 类的 `open` 方法来打开对远程服务器的连接。如果在调用 `open` 方法时提供了远程服务器的地址作为参数，那么 `open` 方法会直接调用 `connect` 方法进行连接；否则还需要显式地调用 `connect` 方法进行连接。在默认情况下，`open` 方法的调用是阻塞式的。当连接成功之后，就可以向套接字通道中写入数据。这里写入的是 HTTP 请求信息。写入完成之后可以进行读取操作，读取服务器端返回的 HTTP 响应的内容。这里把通道的内容传输到文件中。从这里可以看出通道相对于流的灵活性，是它不再需要显式地去获取输入流或输出流的对象，而是可以直接进行读写操作。

代码清单 3-14 阻塞式客户端套接字的使用示例

```
public void loadWebPageUseSocket() throws IOException {
    try (FileChannel destChannel = FileChannel.open(Paths.get("content.txt"),
        StandardOpenOption.WRITE, StandardOpenOption.CREATE);
        SocketChannel sc = SocketChannel.open(new InetSocketAddress("www.
            baidu.com", 80))) {
        String request = "GET / HTTP/1.1\r\n\r\nHost: www.baidu.com\r\n\r\n";
        ByteBuffer header = ByteBuffer.wrap(request.getBytes("UTF-8"));
        sc.write(header);
        destChannel.transferFrom(sc, 0, Integer.MAX_VALUE);
    }
}
```

对于 ServerSocketChannel 类的阻塞模式的使用也比较直接。代码清单 3-15 给出了一个简单的使用 ServerSocketChannel 类的服务器端程序的示例。通过 open 方法打开一个新的套接字通道。当通道打开之后，需要通过调用 bind 方法将其绑定到某个地址上。这里绑定到了本机的 10800 端口上。绑定成功之后，就可以在这个端口上监听客户端的连接请求。ServerSocketChannel 类的 accept 方法会阻塞直到有新的连接发生。当有新的连接建立时，可以通过从 accept 方法得到的 SocketChannel 类的对象来与发起连接的客户端进行数据传输。这里只简单地发送一个字符串给客户端之后就关闭连接。

代码清单 3-15 阻塞式服务器端套接字的使用示例

```
public void startSimpleServer() throws IOException {
    ServerSocketChannel channel = ServerSocketChannel.open();
    channel.bind(new InetSocketAddress("localhost", 10800));
    while (true) {
        try (SocketChannel sc = channel.accept()) {
            sc.write(ByteBuffer.wrap("Hello".getBytes("UTF-8")));
        }
    }
}
```

套接字通道的阻塞模式总体来说与 java.net 包中的 Socket 类和 ServerSocket 类的使用方式非常类似，区别在于使用了 NIO 中新的通道的概念。

2. 多路复用套接字通道

如果程序对网络操作的并发性和吞吐量的要求比较高，那么阻塞式的套接字通道就不能比较简单地满足程序的需求。这时比较好的办法是通过非阻塞式的套接字通道实现多路复用或者使用 NIO.2 中的异步套接字通道。

套接字通道的多路复用的思想比较简单，通过一个专门的选择器（selector）来同时对多个套接字通道进行监听。当其中的某些套接字通道上有它感兴趣的事件发生时，这些通道会变为可用的状态，可以在选择器的选择操作中被选中。选择器通过一次选择操作可以获取这些被选中的通道的列表，然后根据所发生的事件类型分别进行处理。这种基于选择器的做法的优势在于可以同时管理多个套接字通道，而且可用通道的选择一般是通过操作系统提供的底层系统调用来实现的，性能也比较高。

多路复用的实现方式的核心是选择器，即 java.nio.channels.Selector 类的对象。非阻塞式的套接字通道可以通过 register 方法注册到某个 Selector 类的对象上，以声明由该 Selector 类的对象来管理当前这个套接字通道。在进行注册时，需要提供一个套接字通道感兴趣的事件的列表。这些事件包括连接完成、接收到新连接请求、有数据可读和可以写入数据等。这些事件定义在 java.nio.channels.SelectionKey 类中。在完成注册之后，可以调用 Selector 类的对象的 select 方法来进行选择。选择操作完成之后，可以从 Selector 类的对象中得到一个可用的套接字通道的列表。对于这个列表中的套接字通道来说，至少有一个它注册时声明的感兴趣的事件发生了。接着就可以根据事件的类型来

进行相应的处理。一个套接字通道只有在通过 `configureBlocking` 方法设置为非阻塞模式之后，才能被注册到选择器上。套接字通道在非阻塞模式下的读取和写入操作与阻塞模式下差别很大，在使用时需要格外注意。比如在进行读取操作时，非阻塞式套接字通道的 `read` 方法只会读取当时立即可以获取的数据，而不会等待数据的到来。因此，有可能在一次 `read` 方法调用中没有读取到任何数据。

下面用一个完整的示例来说明 `Selector` 类和非阻塞式 `SocketChannel` 类如何结合起来使用。示例的场景仍然是代码清单 3-9 中给出的通过套接字连接来下载一个网页的内容，只不过需求变成同时下载多个网页的内容。如果用传统的阻塞式套接字通道的方式，那么可以启动多个线程来完成。这里要介绍的是在一个线程中使用 `Selector` 类的做法。完整的实现如代码清单 3-16 所示。通过代码中 `LoadWebPageUseSelector` 类的 `load` 方法就可以下载多个网页的内容到本地。

代码清单 3-16 选择器的使用示例

```
public class LoadWebPageUseSelector {

    public void load(Set<URL> urls) throws IOException {
        Map<SocketAddress, String> mapping = urlToSocketAddress(urls);
        Selector selector = Selector.open();
        for (SocketAddress address : mapping.keySet()) {
            register(selector, address);
        }
        int finished = 0, total = mapping.size();
        ByteBuffer buffer = ByteBuffer.allocate(32 * 1024);
        int len = -1;
        while (finished < total) {
            selector.select();
            Iterator<SelectionKey> iterator = selector.selectedKeys().iterator();
            while (iterator.hasNext()) {
                SelectionKey key = iterator.next();
                iterator.remove();
                if (key.isValid() && key.isReadable()) {
                    SocketChannel channel = (SocketChannel) key.channel();
                    InetSocketAddress address = (InetSocketAddress) channel.
                        getRemoteAddress();
                    String filename = address.getHostName() + ".txt";
                    FileChannel destChannel = FileChannel.open(Paths.get(filename),
                        StandardOpenOption.APPEND, StandardOpenOption.CREATE);
                    buffer.clear();
                    while ((len = channel.read(buffer)) > 0 || buffer.position()
                        != 0) {
                        buffer.flip();
                        destChannel.write(buffer);
                        buffer.compact();
                    }
                    if (len == -1) {

```

```

        finished++;
        key.cancel();
    }
} else if (key.isValid() && key.isConnectable()) {
    SocketChannel channel = (SocketChannel) key.channel();
    boolean success = channel.finishConnect();
    if (!success) {
        finished++;
        key.cancel();
    } else {
        InetSocketAddress address = (InetSocketAddress) channel.
            getRemoteAddress();
        String path = mapping.get(address);
        String request = "GET " + path + " HTTP/1.0\r\n\r\nHost:
            " + address.getHostString() + "\r\n\r\n";
        ByteBuffer header = ByteBuffer.wrap(request.
            getBytes("UTF-8"));
        channel.write(header);
    }
}
}

private void register(Selector selector, SocketAddress address) throws
IOException {
    SocketChannel channel = SocketChannel.open();
    channel.configureBlocking(false);
    channel.connect(address);
    channel.register(selector, SelectionKey.OP_CONNECT | SelectionKey.OP_
        READ);
}

private Map<SocketAddress, String> urlToSocketAddress(Set<URL> urls) {
    Map<SocketAddress, String> mapping = new HashMap<>();
    for (URL url : urls) {
        int port = url.getPort() != -1 ? url.getPort() : url.getDefaultPort();
        SocketAddress address = new InetSocketAddress(url.getHost(), port);
        String path = url.getPath();
        if (url.getQuery() != null) {
            path = path + "?" + url.getQuery();
        }
        mapping.put(address, path);
    }
    return mapping;
}
}

```

在使用选择器之前，首先需要创建它。通过 Selector 类的 open 方法可以创建一个新的选择器。有了选择器之后，下一步是把套接字通道注册到选择器上，这一步在私有

方法 register 中完成。在 register 方法中，会首先创建连接 HTTP 服务器的套接字通道，并通过 configureBlocking 方法将通道设置成非阻塞模式，最后再注册到选择器上。在注册时要指定套接字通道感兴趣的事件。由于程序只需要连接远程服务器并进行读取操作，这里指定了套接字通道只对连接完成 (OP_CONNECT) 和通道有数据可读 (OP_READ) 两种事件感兴趣。套接字通道注册完成之后，下一步就是调用 Selector 类的对象的 select 方法来进行通道选择操作。直接调用 select 方法是会阻塞的，直到所监听的套接字通道中至少有一个它们所感兴趣的事件发生为止。在执行完 select 方法之后，通过调用 selectedKeys 方法可以获取到表示被选中的通道的 SelectionKey 类的对象的集合。每个 SelectionKey 类的对象与一个被监听的通道相对应。接下来的操作就是对选中的每一个 SelectionKey 类的对象所发生的是什么类型的事件进行判断，再进行相应的处理。

以连接完成的事件来说，这次连接可能成功也可能失败。通过 SocketChannel 类的对象的 finishConnect 方法可以完成连接，同时判断连接是否成功建立。如果连接建立失败，那么通过 SelectionKey 类的对象的 cancel 方法可以取消选择器对此通道的管理；如果连接建立成功，那么应该向通道中写入 HTTP 的请求头信息，相当于向 HTTP 服务器发送 HTTP 请求。当 HTTP 服务器返回网页内容时，套接字通道会变成可读的状态。这个状态会在下一次调用 select 方法时被选中。在通道处于可读时的处理逻辑是读取通道中的数据，并写入到文件中。对于一个通道来说，由于数据是持续不断地传输的，所以通道可能多次处于可读的状态。如果 read 方法的返回值为 0，说明本次没有数据可读，不需要额外的操作；如果 read 方法返回值为 -1，说明该通道的所有数据已经读取完毕，应该通过对 SelectionKey 类的对象的 cancel 方法取消对此通道的监听。

代码清单 3-16 的示例虽然没有使用多线程，但是如果在运行时输出相关调试信息，会发现来自不同服务器的数据的读取操作是交错进行的。这是因为当某个通道的数据暂时还在传输中时，可能另外一个通道的数据已经准备就绪，可以进行读取了。通过这种多路复用的特性，使程序尽可能地利用网络操作本身的特性来提高性能和吞吐量，而不是依靠多线程带来的并发性。

3.4 NIO.2

Java NIO 在 Java I/O 库的基础上增加了通道的概念，提供了 I/O 操作的性能，使用起来也更加简单。Java 7 中的 I/O 库得到了进一步增强，称为 NIO.2。NIO.2 中包含的主要内容包括文件系统访问和异步 I/O 通道。

3.4.1 文件系统访问

3.3.1 节介绍了文件通道相关的内容。相对于传统的基于 java.io.File 类的文件操作来说，文件通道在某些方面更加简单和高效，但还是有一些与文件相关的操作需要依靠 File 类来完成，比如列出某个目录下的所有文件的操作。File 类中的操作存在一些不方

便开发人员使用的地方，对此，Java 7 加强了文件操作相关的功能，即新的 `java.nio.file` 包，其所提供的新功能包括文件路径的抽象、文件目录列表流、文件目录树遍历、文件属性和文件变化监视服务等。

1. 文件路径的抽象

在使用 `java.io.File` 类来操作文件时，需要以字符串的方式指定文件的路径。虽然文件路径本身最终的表现形式是字符串，但是直接利用字符串来进行与路径相关操作时，丢失了路径本身的语义。比如，一个路径中可能包含多个子部分，每个部分表示一个目录或是文件，如果希望把两个路径连接起来得到一个新的路径，传统的做法是进行字符串相加。这种做法不是类型安全的，要正确无误地实现也不是那么容易。NIO.2 中引入的 `java.nio.file.Path` 接口作为文件系统中路径的一个抽象，很好地解决了这个问题。`Path` 接口除了带来了类型安全的好处之外，还提供了操作路径的很多实用方法，使开发人员不再需要编写很多重复的代码。类型安全的重要性是很明显的。在有了 `Path` 接口之后，就不会在一个需要使用文件路径的地方将一个随意的字符串作为参数传递进去。在使用 `String` 类的对象来表示文件路径时，这种错误是很可能发生的。

`Path` 接口相当于将一个字符串表示的文件路径重新细分，使之赋有语义。以一个典型的文件路径 “`C:\foo\bar\myfile.txt`” 为例，如果得到了它对应的 `Path` 接口的实现，那么 `Path` 接口实现对象中的“根”指的是“`C:\`”，可以通过 `getRoot` 方法来获取，路径中通过路径分隔符隔开的每一个部分都成为其中的名称元素。该路径有 3 个名称元素，分别是表示目录名或文件名的“`foo`”、“`bar`” 和 “`myfile.txt`”，可以通过 `getNameCount` 获取到名称元素的总数，通过 `getName` 来获取单个名称元素；在路径中距离根最远的名称元素，称为该路径的文件名，可以通过 `getFileName` 方法来获取，这里的值是 “`myfile.txt`”；通过 `getParent` 可以获取到当前路径的父路径，这里的值是 “`C:\foo\bar`”。

有了 `Path` 接口之后，对文件路径进行的很多操作变得很简单，不再需要依靠复杂的字符串操作。代码清单 3-17 给出了通过 `Path` 接口操作文件路径的示例，每个方法调用的后面用注释的形式给出了运行结果。`Path` 接口中 `resolve` 方法的作用相当于把当前路径当成父目录，而把参数中的路径当成子目录或是其中的文件，进行解析之后得到一个新路径；`resolveSibling` 方法的作用与 `resolve` 方法类似，只不过把当前路径的父目录当成解析时的父目录；`relativize` 方法的作用与 `resolve` 方法正好相反，用来计算当前路径相对于参数中给出的路径的相对路径；`subpath` 方法用来获取当前路径的子路径，参数中的序号表示的是路径中名称元素的序号；`startsWith` 和 `endsWith` 方法用来判断当前路径是否以参数中的路径开始或结尾。在一般的路径中，“`.`” 和 “`..`” 分别用来表示当前目录和上一级目录。通过 `normalize` 方法可以去掉路径中的“`.`” 和 “`..`”。所有这些方法的返回值都是 `Path` 接口的实现对象，因此这些方法可以很容易地级联起来。

代码清单 3-17 `Path` 接口的使用示例

```
public void usePath() {
```

```

Path path1 = Paths.get("folder1", "sub1");
Path path2 = Paths.get("folder2", "sub2");
path1.resolve(path2); //folder1\sub1\folder2\sub2
path1.resolveSibling(path2); //folder1\folder2\sub2
path1.relativize(path2); //..\..\folder2\sub2
path1.subpath(0, 1); //folder1
path1.startsWith(path2); //false
path1.endsWith(path2); //false
Paths.get("folder1/.../folder2/my.text").normalize(); //folder2\my.text
}

```

2. 文件目录列表流

当需要列出一个目录下的子目录和文件时，传统的做法是使用 `File` 类中的 `list` 或 `listFiles` 方法。不过这两个方法在目录中包含的文件数量很多的时候，性能比较差。NIO.2 中引入了一个新的接口 `java.nio.file.DirectoryStream` 来支持这种遍历操作。`DirectoryStream` 接口继承了 `java.lang.Iterable` 接口，使 `DirectoryStream` 接口的实现对象可以直接在增强的 `for` 循环中使用。`DirectoryStream` 接口的优势在于它渐进式地遍历文件，每次只读取一定数量的内容，从而可以降低遍历时的开销。在实际的使用中，`DirectoryStream` 接口的实现对象是通过 `java.nio.file.Files` 类的工厂方法来创建的。在创建时，可以指定遍历时的过滤条件，即满足何种条件的目录和文件才会被包括进来。指定遍历条件时既可以使用 `DirectoryStream.Filter` 接口实现类的对象，也可以使用字符串来表示简单的模式。代码清单 3-18 中给出了遍历当前目录下所有的“.java”文件的示例。

代码清单 3-18 目录列表流的使用示例

```

public void listFiles() throws IOException {
    Path path = Paths.get("");
    try (DirectoryStream<Path> stream = Files.newDirectoryStream(path, "*.java")) {
        for (Path entry: stream) {
            // 使用 entry
        }
    }
}

```

如果希望程序自己来遍历 `DirectoryStream` 接口实现对象中的条目，可以通过 `iterator` 方法获取到 `DirectoryStream` 接口实现对象的迭代器对象。不过 `iterator` 方法只能调用一次，得到唯一的一个迭代器对象。如果在遍历过程中，目录中的文件发生了变化，这种变化可能会被迭代器捕获到，也可能不会。程序不应该依赖迭代器来发现这些变化，更好的做法是使用目录监视服务。

3. 文件目录树遍历

`DirectoryStream` 接口只能遍历当前目录下的直接子目录或文件，并不会递归地遍历子目录下的子目录。如果希望对整个目录树进行遍历，需要使用 `java.nio.file.FileVisitor`

接口。FileVisitor 接口是典型的访问者模式的实现。在这个接口中定义了 4 个方法，分别表示对目录树的不同访问动作。首先是 visitFile 方法，它表示正在访问某个文件；其次是 visitFileFailed 方法，它表示访问某个文件时出现了错误；接着是 preVisitDirectory 方法，它表示在访问一个目录中包含的子目录和文件之前被调用；最后是 postVisitDirectory 方法，它表示在访问一个目录的全部子目录中的内容之后被调用。这 4 个方法都返回 java.nio.file.FileVisitResult 枚举类型，用来声明整个遍历过程的下一步动作。在这些枚举值中：CONTINUE 表示继续进行正常的遍历过程；SKIP_SIBLINGS 表示跳过当前目录或文件的兄弟节点；SKIP_SUBTREE 表示不再遍历当前目录中的内容；TERMINATE 表示立即结束整个遍历过程。通过实现这 4 个方法并根据情况返回相应的遍历动作，程序可以很容易地控制整个遍历过程，并在遍历中对整个目录树进行修改。

下面通过一个具体的示例来进行说明。在开发过程中我们有时候会使用 Subversion 作为源代码配置管理的工具。Subversion 会在其管理的目录下面添加 “.svn” 子目录来保存其所需的元数据。如果直接把整个目录复制给其他人，会发现 “.svn” 目录也包含在其中。这个目录是没有必要存在的。代码清单 3-19 给出了一个遍历某个目录并清除其中包含的 “.svn” 目录的 FileVisitor 接口的实现。SvnInfoCleanVisitor 类并没有直接实现 FileVisitor 接口，而是继承自 java.nio.file.SimpleFileVisitor 类。SimpleFileVisitor 类是一个简单的 FileVisitor 接口的适配器。通过继承 SimpleFileVisitor 类的方式可以不必实现 FileVisitor 接口中的全部方法。在进行遍历的过程中，如果遇到一个名称为 “.svn” 的目录，则说明需要将此目录下的所有子目录和文件删除。由于 “.svn” 目录中的文件是只读的，因此在删除文件时，需要先取消对只读属性的设置，再进行删除。在进行删除操作时，需要先删除目录中包含的文件，再删除该目录。

代码清单 3-19 删除 Subversion 元数据的目录遍历方式

```
public class SvnInfoCleanVisitor extends SimpleFileVisitor<Path> {

    private boolean cleanMark = false;

    private boolean isSvnFolder(Path dir) {
        return ".svn".equals(dir.getFileName().toString());
    }

    public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs)
        throws IOException {
        if (isSvnFolder(dir)) {
            cleanMark = true;
        }
        return FileVisitResult.CONTINUE;
    }

    public FileVisitResult postVisitDirectory(Path dir, IOException e) throws
        IOException {
```

```

        if (e == null && cleanMark) {
            Files.delete(dir);
            if (isSvnFolder(dir)) {
                cleanMark = false;
            }
        }
        return FileVisitResult.CONTINUE;
    }

    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws
        IOException {
        if (cleanMark) {
            Files.setAttribute(file, "dos:readonly", false);
            Files.delete(file);
        }
        return FileVisitResult.CONTINUE;
    }
}

```

4. 文件属性

文件属性是文件除了本身的数据之外的元数据。常见的属性包括是否只读、是否为隐藏文件、上次访问时间和所有者信息等。在 Java 7 之前，Java 标准库只有少量与文件属性相关的方法，主要在 `File` 类中。这些方法的功能比较弱，而且也不够系统。在 Java 7 中，NIO.2 提供了专门的 `java.nio.file.attribute` 包来对文件属性进行处理。由于不同操作系统上的文件系统对文件属性的支持是不同的，NIO.2 对文件属性进行了抽象，采用了文件属性视图的概念。每个属性视图中包含了可以从这个视图中获取和设置的各种属性。不同的视图所包含的属性是不一样的。每个属性视图都有自己的名称。

接口 `java.nio.file.attribute.AttributeView` 是所有属性视图的父接口。`AttributeView` 的子接口 `java.nio.file.attribute.FileAttributeView` 表示的是文件的属性视图。`FileAttributeView` 接口表示的属性视图分为两类，一类是由 `java.nio.file.attribute.BasicFileAttributeView` 接口表示的包含基本文件属性的视图，另外一类是由 `java.nio.file.attribute.FileOwnerAttributeView` 接口表示的包含文件所有者信息的属性视图。调用 `BasicFileAttributeView` 接口的 `readAttributes` 方法可以获取到表示文件基本属性的 `java.nio.file.attribute.BasicFileAttributes` 接口的实现对象。`BasicFileAttributes` 接口中包含了类型安全的方法，这些方法用来获取不同文件系统中的通用属性，包括创建时间 (`creationTime`)、上次访问时间 (`lastAccessTime`)、上次修改时间 (`lastModifiedTime`)、是否是目录 (`isDirectory`)、是否为普通文件 (`isRegularFile`)、是否为符号链接 (`isSymbolicLink`) 和文件大小 (`size`) 等。`FileOwnerAttributeView` 接口的 `getOwner` 和 `setOwner` 方法可以用来获取和设置文件的所有者信息。

Windows 操作系统中的文件系统一般使用遗留的“DOS”文件属性视图，由 `java.nio.file.attribute.DosFileAttributeView` 接口来表示。`DosFileAttributeView` 接口中包含的属

性有是否为只读文件（readonly）、是否为隐藏文件（hidden）、是否为系统文件（system）和是否为归档文件（archive）等。DosFileAttributeView 接口中包含了设置这些属性的方法。如果需要获取属性的值，先通过 readAttributes 方法获取到 java.nio.file.attribute.DosFileAttributes 接口的实现对象，再调用该实现对象中的对应方法来获取属性值。与 DosFileAttributeView 接口相对应的是 java.nio.file.attribute.PosixFileAttributeView 接口，表示 UNIX 和 Linux 系统使用的 POSIX 文件属性视图。PosixFileAttributeView 接口中包含的属性有所有者信息（group）和权限信息（permissions）。PosixFileAttributeView 接口的使用方式类似于 DosFileAttributeView 接口，使用 PosixFileAttributeView 接口本身提供的方法来进行属性设置；通过 readAttributes 方法获取 java.nio.file.attribute.PosixFileAttributes 接口的实现对象来读取属性的值。

上面介绍的属性视图相关的表示方式都是接口。实际的获取和设置文件属性的操作是通过 Files 类中的静态方法来完成的。Files 类提供的与文件属性相关的方法比较多，分成获取和设置属性两类。获取属性的方式有两种：一种是获取 FileAttributeView 接口或 BasicFileAttributes 接口的实现对象后，再调用相应的方法来获取属性的值；另外一种是直接指定属性的名称来获取相应的值。Files 类中的 getFileAttributeView 方法用来获取 FileAttributeView 接口的实现对象，在调用时需要指定所要获取的属性视图的类名。代码清单 3-20 通过 getFileAttributeView 方法获取 DosFileAttributeView 接口中包含的文件的“是否只读”属性的值。

代码清单 3-20 文件属性视图的使用示例

```
public void useFileAttributeView() throws IOException {
    Path path = Paths.get("content.txt");
    DosFileAttributeView view = Files.getFileAttributeView(path,
        DosFileAttributeView.class);
    if (view != null) {
        DosFileAttributes attrs = view.readAttributes();
        System.out.println(attrs.isReadOnly());
    }
}
```

在代码清单 3-20 中，获取到 DosFileAttributeView 接口的实现对象之后，还需要调用其 readAttributes 方法来获取具体的包含属性的对象。从简化使用的角度出发，Files 类中提供了 readAttributes 方法来直接获取特定类型的 BasicFileAttributes 接口的实现对象。

Files 类中的 readAttributes 和 getAttribute 方法可以根据属性名称来获取对应的值。不同之处在于 readAttributes 方法可以批量获取一组属性的值，而 getAttribute 方法只能获取一个属性的值。使用这两个方法时需要指定文件的路径及属性的名称。属性的名称是带名称空间的，其前缀是属性所在的属性视图的名称，比如“DOS”文件属性视图中的“是否为隐藏文件”属性的完整名称是“dos:hidden”。在不带前缀的情况下，默认属性来自基本属性视图。在调用 readAttributes 方法时，多个属性名称之间用

逗号分隔即可。代码清单 3-21 给出了一个通过检查文件的上次修改时间来判断文件是否需要更新的示例，文件的上次修改时间对应的属性名称是“lastModifiedTime”。由于“lastModifiedTime”属性在基本属性视图中，因此使用时不需要添加视图名称作为前缀。文件属性中的创建时间、上次修改时间和上次访问时间都是由 `java.nio.file.attribute.FileTime` 类的对象来表示的。

代码清单 3-21 获取文件的上次修改时间的示例

```
public boolean checkUpdatesRequired(Path path, int intervalInMillis) throws
    IOException {
    FileTime lastModifiedTime = (FileTime) Files.getAttribute(path,
        "lastModifiedTime");
    long now = System.currentTimeMillis();
    return now - lastModifiedTime.toMillis() > intervalInMillis;
}
```

在设置文件属性值时，也有两种对应的方式：一种是使用 `Files` 类的 `getFileTypeAttributeView` 方法获取到 `FileAttributeView` 接口的实现对象之后，通过该对象提供的方法来进行设置；另外一种是调用 `Files` 的 `setAttribute` 方法，设置时使用的是属性名称。`Files` 类中也提供了一些快捷的方法来获取和设置常见文件属性的值，比如 `getOwner/setOwner` 和 `getLastModifiedTime/setLastModifiedTime` 等实用方法。

5. 监视目录变化

在实际开发中可能会需要监视某个目录下的文件所发生的变化，比如支持热部署的 Web 容器需要监视某个特定目录下是否出现新的待部署的 Web 应用的归档文件。另外一个场景是程序的输入来自某个特定目录下面的文本文件，要求每出现一个文件就立即进行处理。在 Java 7 之前，这种目录监视功能需要开发人员自己来实现。一般的做法是：在一个独立的线程中使用 `File` 类的 `listFiles` 方法来定时检查目录中的内容，并与之前的内容进行比较，从而判断是否有新的文件出现，文件内容是否被修改或被删除。NIO.2 中提供了新的目录监视服务，使用该服务可以在指定目录中的子目录或文件被创建、更新或删除时得到事件通知。基于这些通知，程序可以进行相应的处理。

目录监视服务的使用方式类似于 3.3.2 节介绍的非阻塞式套接字通道与选择器的使用方式，被监视的对象要实现 `java.nio.file.Watchable` 接口，并通过 `register` 方法注册到表示监视服务的 `java.nio.file.WatchService` 接口的实现对象上，注册时需要指定被监视对象感兴趣的事件类型。注册成功之后，调用者可以得到一个表示这次注册行为的 `java.nio.file.WatchKey` 接口的实现对象，其作用类似于 `SelectionKey` 类。通过 `WatchKey` 接口可以获取在对应的被监视对象上所产生的事件。每个事件用 `java.nio.file.WatchEvent` 接口来表示。与 `Selector` 类中的 `select` 方法一样，`WatchService` 接口也提供了类似的方法来获取当前所有被监视的对象上的可用事件。查询的方式也分成阻塞式和非阻塞式两种：阻塞式方式使用的是 `take` 方法，而非阻塞式方式使用的是 `poll` 方法。查询结果的返

回值是 WatchKey 接口的实现对象。调用 WatchKey 接口的 pollEvents 方法可以得到对应被监视对象上所发生的所有事件。

代码清单 3-22 中的代码会监视当前的工作目录，当有新的文件被创建时，输出该文件的大小。WatchService 接口的实现对象是由工厂方法创建的，需要从表示文件系统的 java.nio.file.FileSystem 类的对象中得到。目前，唯一可以被监视的对象只有 Path 接口的实现对象。可以被监视的事件包括创建或重命名 (ENTRY_CREATE)、更新 (ENTRY_MODIFY) 和删除 (ENTRY_DELETE)。这些事件定义在 java.nio.file.StandardWatchEventKinds 类中。当有事件发生时，通过对 WatchKey 接口的实现对象的 pollEvents 方法获取所有的事件。WatchEvent 接口的 context 方法的返回值表示的是事件的上下文信息。在与目录内容变化相关的事件中，上下文信息是一个 Path 接口的实现对象，表示的是产生事件的文件路径相对于被监视路径的相对路径，因此需要使用 Path 接口的 resolve 方法来得到完整的路径。在处理完一个 WatchKey 接口实现对象中的全部事件之后，需要通过 reset 方法来进行重置。只有在重置之后，新产生的同类事件才有可能从 WatchService 接口实现对象中再次获取。

代码清单 3-22 目录监视服务的使用示例

```
public void calculate() throws IOException, InterruptedException {
    WatchService service = FileSystems.getDefault().newWatchService();
    Path path = Paths.get("").toAbsolutePath();
    path.register(service, StandardWatchEventKinds.ENTRY_CREATE);
    while (true) {
        WatchKey key = service.take();
        for (WatchEvent<?> event : key.pollEvents()) {
            Path createdPath = (Path) event.context();
            createdPath = path.resolve(createdPath);
            long size = Files.size(createdPath);
            System.out.println(createdPath + " ==> " + size);
        }
        key.reset();
    }
}
```

如果希望取消对一个目录的监视，只需要调用对应 WatchKey 接口实现对象的 cancel 方法即可。

6. 文件操作的实用方法

在程序中进行文件操作时，经常会使用一些通用操作。Files 类中提供了一系列的静态方法，可以满足很多常见的需求。在前面给出的示例代码中，大量用到了 Files 类。

Files 类中提供了创建目录和文件的功能。Files 类中提供的方法既可以创建目录和一般文件，也可以创建符号连接，还可以创建临时目录和临时文件。在创建时可以指定新目录和文件的属性。Files 类还提供了复制文件的功能，既支持从一个 InputStream 类的

对象中读入数据到一个文件，也支持从一个文件中读取数据并写入到一个 OutputStream 类的对象中，还支持两个文件之间的数据传递。这个功能类似于 3.3.1 节介绍的文件通道的数据传输功能。在文件读写方面，Files 类支持一次性读入文件的所有字节或所有行，也支持把一个字节数组和一组字符串写入到文件中。除了这些之外，Files 类对文件的删除和移动也提供了支持。所有这些操作在指定目录或文件时都是使用 Path 接口来表示的。代码清单 3-23 中给出了 Files 类中部分实用方法的使用示例。

代码清单 3-23 文件操作的实用方法的使用示例

```
public void manipulateFiles() throws IOException {
    Path newFile = Files.createFile(Paths.get("new.txt").toAbsolutePath());
    List<String> content = new ArrayList<String>();
    content.add("Hello");
    content.add("World");
    Files.write(newFile, content, Charset.forName("UTF-8"));
    Files.size(newFile);
    byte[] bytes = Files.readAllBytes(newFile);
    ByteArrayOutputStream output = new ByteArrayOutputStream();
    Files.copy(newFile, output);
    Files.delete(newFile);
}
```

在 Files 中，除了有直接操作文件的方法之外，还有把文件转换成各种不同形式的方法，比如，newInputStream 和 newOutputStream 方法可以分别得到一个文件对应的 InputStream 类的对象和 OutputStream 类的对象；newBufferedReader 和 newBufferedWriter 方法可以得到文件对应的 BufferedReader 类的对象和 BufferedWriter 类的对象；newByteChannel 可以得到一个实现 SeekableByteChannel 接口的通道对象。

3.4.2 zip/jar 文件系统

NIO.2 的一个重要新特性是允许开发人员创建自定义的文件系统实现。在 Java 7 之前，对文件系统的操作只能使用由 Java 标准库提供的基于底层操作系统支持的默认实现。Java 标准库中的与文件相关的抽象，如 File 类，都是基于此默认实现的。在有些情况下，默认的文件系统实现不能满足要求，在这种情况下虽然可以开发出自己的文件系统，但是在使用时并没有已有的文件系统那么直接和自然。

NIO.2 把对文件系统的表示抽象出来，形成 java.nio.file.FileSystem 接口。如果默认的文件系统实现不能满足要求，可以通过实现此接口来添加自定义的实现，如创建基于内存的文件系统，或者创建分布式的文件系统。在 FileSystem 接口被引入之后，使用文件系统的代码不需要关心文件系统的底层实现细节，只需要通过 Java 标准库的相关 API 来操作即可。

除了实现 FileSystem 接口之外，自定义文件系统的实现还需要实现 java.nio.file.spi.FileSystemProvider 接口，把自定义的文件系统实现注册到 Java 平台中。每个文

件系统都有一个对应的 URI 模式作为该文件系统的标识符，比如，默认的文件系统的 URI 模式是“file”。FileSystemProvider 接口的 getScheme 方法返回的是该模式的值。FileSystemProvider 接口的 newFileSystem 方法用来创建新的文件系统实现，即 FileSystem 接口的实现对象。在 newFileSystem 方法的实现中，需要根据作为参数传入的 URI 或路径创建出对应的 FileSystem 接口的实现对象。FileSystemProvider 接口的实现类以标准的服务提供者接口方式进行注册，所对应的服务名称是“java.nio.file.spi.FileSystemProvider”。通过 FileSystemProvider 接口的 installedProviders 方法可以获取程序中当前可用的 FileSystemProvider 接口实现类的列表。

对于使用者来说，可以通过 java.nio.file.FileSystems 类中的静态工厂方法来获取或创建 FileSystem 接口的实现对象，其中 getDefault 方法用来获取默认的文件系统实现。通常的默认文件系统实现是基于底层操作系统上的文件系统的，可以通过系统参数“java.nio.file.spi.DefaultFileSystemProvider”来设置默认的文件系统实现的 Java 类名。FileSystems 类中的 getFileSystem 方法根据 URI 来获取对应的 FileSystem 类的对象，而 newFileSystem 方法用来创建新的 FileSystem 类的对象。

Java 标准库中包含了两种文件系统的实现：一种是默认的基于底层操作系统的文件系统的实现，另外一种是 NIO.2 中新增的操作 zip 和 jar 文件的文件系统。Java 7 之前处理 zip 和 jar 等压缩文件时使用的是 java.util.zip 包和 java.util.jar 包中的 Java 类。这两个包中的 Java 类使用起来并不灵活。API 的用法不同于一般的文件操作，比如向一个已经存在的 zip 文件中添加一个新文件的需求，通过 java.util.zip 包中的 API 来实现的代码如代码清单 3-24 所示。基本的实现思路是先创建一个临时文件作为中转，把 zip 文件中已有的内容重新复制，再添加新的文件。

代码清单 3-24 向已有的 zip 文件中添加新文件的传统做法

```
public void addFileToZip(File zipFile, File fileToAdd) throws IOException {
    File tempFile = File.createTempFile(zipFile.getName(), null);
    tempFile.delete();
    zipFile.renameTo(tempFile);
    try (ZipInputStream input = new ZipInputStream(new FileInputStream(tempFile));
         ZipOutputStream output = new ZipOutputStream(new FileOutputStream(zipFile))) {
        ZipEntry entry = input.getNextEntry();
        byte[] buf = new byte[8192];
        while (entry != null) {
            String name = entry.getName();
            if (!name.equals(fileToAdd.getName())) {
                output.putNextEntry(new ZipEntry(name));
                int len = 0;
                while ((len = input.read(buf)) > 0) {
                    output.write(buf, 0, len);
                }
            }
        }
    }
}
```

```

        entry = input.getNextEntry();
    }
    try(InputStream newFileInput = new FileInputStream(fileToAdd)) {
        output.putNextEntry(new ZipEntry(fileToAdd.getName()));
        int len = 0;
        while ((len = newFileInput.read(buf)) > 0) {
            output.write(buf, 0, len);
        }
        output.closeEntry();
    }
    tempFile.delete();
}

```

如果使用 NIO.2 中新增的 zip/jar 文件系统，同样的需求可以通过更加简洁的方式来实现。这种实现方式是把一个 zip/jar 文件看成一个独立的文件系统，进而使用 Java 提供的与各种文件操作相关的 API。创建基于 zip 和 jar 文件的文件系统的方式有两种：一种是使用模式为“jar”的 URI 来调用 FileSystems 类的 newFileSystem 方法；另一种是使用 Path 接口的实现对象来调用 newFileSystem 方法。如果文件路径的后缀是“.zip”或“.jar”，会自动创建对应的 zip/jar 文件系统实现。得到对应的 FileSystem 类的对象之后，可以使用 FileSystem 类和 Files 类中的方法来对文件进行操作。代码清单 3-25 使用 zip/jar 文件系统来实现与代码清单 3-24 同样的需求。相比较而言，代码清单 3-25 中的逻辑非常简洁清晰，核心的代码只有一行，即调用 Files 类的 copy 方法来完成文件的复制操作。同样，可以使用 Files 类中的 createDirectory 方法在 zip/jar 文件中创建新的目录，还可以使用 delete 方法来删除 zip/jar 文件中已有的目录或文件。

代码清单 3-25 基于 zip/jar 文件系统实现的添加新文件到已有 zip 文件的做法

```

public void addFileToZip2(File zipFile, File fileToAdd) throws IOException {
    Map<String, String> env = new HashMap<>();
    env.put("create", "true");
    try (FileSystem fs = FileSystems.newFileSystem(URI.create("jar:" + zipFile.
        toURI()), env)) {
        Path pathToAddFile = fileToAdd.toPath();
        Path pathInZipfile = fs.getPath="/" + fileToAdd.getName();
        Files.copy(pathToAddFile, pathInZipfile, StandardCopyOption.REPLACE_
        EXISTING);
    }
}

```

3.4.3 异步 I/O 通道

NIO.2 中引入了新的异步通道的概念，并提供了异步文件通道和异步套接字通道的实现。这两种异步通道在功能上类似于前面介绍过的一般通道，只是对应功能的使用方

式不相同。对于文件通道来说，一般的操作，如读取和写入，都是同步进行的，调用者会处于阻塞状态，以等待相应操作的完成。而对于套接字通道来说，阻塞式套接字通道的使用方式与文件通道相同，而非阻塞式套接字通道的使用方式则依靠选择器来完成。异步通道一般提供两种使用方式：一种是通过 Java 同步工具包中的 `java.util.concurrent.Future` 类的对象来表示异步操作的结果；另外一种是在执行操作时传入一个 `java.nio.channels.CompletionHandler` 接口的实现对象作为操作完成时的回调方法。这两种使用方式的区别只在于调用者通过何种方式来使用异步操作的结果。在使用 `Future` 类的对象时，要求调用者在合适的时机显式地通过 `Future` 类的对象的 `get` 方法来得到实际的操作结果；而在使用 `CompletionHandler` 接口时，实际的调用结果作为回调方法的参数来给出。

下面先通过一个异步文件通道来介绍使用 `Future` 类的对象的做法。异步文件通道由 `java.nio.channels.AsynchronousFileChannel` 类来表示。如代码清单 3-26 所示，打开一个异步文件通道的方式与使用 `FileChannel` 类的做法相似，也是通过 `open` 方法来完成的。对文件通道的读取和写入也是通过对读和 `write` 方法来完成的。所不同的是 `read` 和 `write` 方法要么返回一个 `Future` 类的对象，要么要求传入一个 `CompletionHandler` 接口的实现对象作为回调方法。这里用的是 `write` 方法返回的 `Future` 类的对象。在调用 `write` 方法之后，程序可以执行其他的操作，然后再调用 `Future` 类的对象的 `get` 方法来获取 `write` 操作的执行结果。如果操作执行成功，`get` 方法会返回实际写入的字符数；如果执行失败，会抛出 `java.util.concurrent.ExecutionException` 异常。

代码清单 3-26 向异步文件通道中写入数据的示例

```
public void asyncWrite() throws IOException, ExecutionException,
    InterruptedException {
    AsynchronousFileChannel channel = AsynchronousFileChannel.open(Paths.
        get("large.bin"), StandardOpenOption.CREATE, StandardOpenOption.WRITE);
    ByteBuffer buffer = ByteBuffer.allocate(32 * 1024 * 1024);
    Future<Integer> result = channel.write(buffer, 0);
    // 其他操作
    Integer len = result.get();
}
```

这里需要注意的是，异步文件通道并不支持 `FileChannel` 类所提供的相对读写操作。在异步文件通道中并没有当前读写位置的概念，因此所有的 `read` 和 `write` 方法在调用时都必须显式地指定读写操作的位置。

异步套接字通道 `AsynchronousSocketChannel` 和 `AsynchronousServerSocketChannel` 类分别对应一般的 `SocketChannel` 和 `ServerSocketChannel` 类。代码清单 3-27 给出了使用 `AsynchronousServerSocketChannel` 类和 `CompletionHandler` 接口的示例。与 `ServerSocketChannel` 类相同的是，`accept` 方法用来接受来自客户端的连接。不过，当有新连接建立时会调用 `CompletionHandler` 接口实现对象中的 `completed` 方法；当出现错误

时，会调用 failed 方法。值得一提的是 accept 方法的第一个参数，该参数可以是一个任意类型的对象，称为调用时的“附件对象”。附件对象在 accept 方法调用时传入，可以在 CompletionHandler 接口的实现对象中从 completed 和 failed 方法的参数中获取，这样就可以进行数据的传递。使用 CompletionHandler 接口的方法都支持使用附件对象来传递数据。

代码清单 3-27 异步套接字通道的使用示例

```

public void startAsyncSimpleServer() throws IOException {
    AsynchronousChannelGroup group = AsynchronousChannelGroup.
        withFixedThreadPool(10, Executors.defaultThreadFactory());
    final AsynchronousServerSocketChannel serverChannel =
        AsynchronousServerSocketChannel.open(group).bind(new
            InetSocketAddress(10080));
    serverChannel.accept(null, new CompletionHandler<AsynchronousSocketChannel,
        Void>() {
        public void completed(AsynchronousSocketChannel clientChannel, Void
            attachement) {
            serverChannel.accept(null, this);
            // 使用 clientChannel
        }
        public void failed(Throwable throwable, Void attachement) {
            // 错误处理
        }
    });
}

```

异步通道在处理 I/O 请求时，需要使用一个 `java.nio.channels.AsynchronousChannelGroup` 类的对象。`AsynchronousChannelGroup` 类的对象表示的是一个异步通道的分组，每一个分组都有一个线程池与之关联，这个线程池中的线程用来处理 I/O 事件。多个异步通道可以共享一个分组的线程池资源。调用 `AsynchronousSocketChannel` 和 `AsynchronousServerSocketChannel` 类的 open 方法打开异步套接字通道时，可以传入一个 `AsynchronousChannelGroup` 类的对象作为所使用的分组。如果调用 open 方法时没有传入 `AsynchronousChannelGroup` 类的对象，默认使用系统提供的分组。需要注意的是，系统分组对应的线程池中的线程是守护线程。如果代码清单 3-27 中没有显式使用 `AsynchronousChannelGroup` 类的对象，程序启动之后会很快退出，因为系统分组使用的守护线程不会阻止虚拟机退出。

创建 `AsynchronousChannelGroup` 类的对象需要使用 `AsynchronousChannelGroup` 类中的静态工厂方法 `withFixedThreadPool`、`withCachedThreadPool` 或 `withThreadPool`。创建出来的 `AsynchronousChannelGroup` 类对象需要被显式关闭，否则虚拟机不会退出。关闭的方式类似于停止 `java.util.concurrent.ExecutorService` 接口表示的任务执行服务的做法。第 11 章将对 `ExecutorService` 接口进行详细介绍，可以参考相应的关闭方式来关闭

AsynchronousChannelGroup 类的对象。

3.4.4 套接字通道绑定与配置

NIO.2 中新增了一个接口 `java.nio.channels.NetworkChannel`。所有与套接字相关的通道的接口和实现类都继承或实现了 `NetworkChannel` 接口。`NetworkChannel` 接口是连接网络套接字的通道的抽象表示。`NetworkChannel` 接口提供了套接字通道的绑定与配置功能。

套接字通道的绑定是把套接字绑定到本机的一个地址上。在 Java 7 之前，通过 `ServerSocketChannel` 类的 `open` 方法打开一个套接字通道之后，新创建的通道处于未绑定的状态。需要调用 `ServerSocketChannel` 类的对象的 `socket` 方法先得到该通道对应的底层 `ServerSocket` 类的对象，再调用该对象的 `bind` 方法进行绑定。利用 `NetworkChannel` 接口中 `bind` 方法可以直接进行套接字通道的绑定，使用起来简便了很多。调用 `bind` 方法时需要提供表示套接字地址的 `java.net.SocketAddress` 类的对象。如果传入值为 `null`，套接字通道会绑定在一个自动分配的地址上。通过 `NetworkChannel` 接口的 `getLocalAddress` 方法可以获取当前套接字通道的实际绑定地址。

`NetworkChannel` 接口的另外一组方法用来对套接字通道进行配置。不同的套接字通道可能提供一些配置项来允许使用者配置其行为。通过 `NetworkChannel` 接口的 `supportedOptions` 方法可以获取套接字通道对象所支持的配置项集合。配置项是 `java.net.SocketOption` 接口的实现对象。`SocketOption` 接口的 `name` 和 `type` 方法分别用来获取配置项的名称和值类型。在 `java.net.StandardSocketOptions` 类中定义了一些标准的配置项，如 `SO_REUSEADDR` 用来配置是否允许重用已有的套接字地址。`NetworkChannel` 接口的 `setOption` 和 `getOption` 方法分别用来设置或获取配置项的值。

3.4.5 IP 组播通道

通过 IP 协议的组播 (multicasting) 支持可以将数据报文传输给属于同一分组的多个主机。当不同主机上的程序都需要接收相同的数据报文时，使用 IP 组播是最自然的方式。每一条组播消息都会被属于特定分组的所有主机接收到。每个组播分组都有一个对应的标识符，该标识符是一个 D 类 IP 地址，范围在 “224.0.0.1” 和 “239.255.255.255” 之间。进行组播的程序可以选择这个范围之内的任意一个 IP 地址作为分组的标识符。主机可以选择加入某个组播分组来接收所有发送给该分组的消息。

NIO.2 中新增了 `java.nio.channels.MulticastChannel` 接口来表示支持 IP 组播的网络通道。已有的 `java.nio.channels.DatagramChannel` 类实现了 `MulticastChannel` 接口。调用 `MulticastChannel` 接口的 `join` 方法可以使当前通道加入到由参数指定的组播分组中，可以接收发送给该分组的消息。调用 `join` 方法的返回值是 `java.nio.channels.MembershipKey` 类的对象，表示该通道在组播分组中的成员身份。`MembershipKey` 类的作用类似于前面介

绍的 SelectionKey 类，可以通过 MembershipKey 类的对象来进行管理。当不再需要接收分组中的消息时，使用 MembershipKey 类的对象的 drop 方法可以将对应的通道移出分组；当底层操作系统提供的 IP 协议的实现支持对组播消息的发送来源进行过滤时，可以使用 block 方法来阻止接收来自特定地址的消息，而 unblock 方法用来解除阻止。

下面通过一个具体示例来说明 IP 组播通道的使用方式。在网络中有一个主机定时向特定组播分组广播当前的时间，相关的实现如代码清单 3-28 所示。通过 DatagramChannel 类的 open 方法创建新的数据报文通道并绑定之后，调用 send 方法向标识符为“224.0.0.2”的分组中发送消息。

代码清单 3-28 进行组播的服务器端实现

```
public class TimeServer {
    public void start() throws IOException {
        DatagramChannel dc = DatagramChannel.open(StandardProtocolFamily.INET).
            bind(null);
        InetAddress group = InetAddress.getByName("224.0.0.2");
        int port = 5000;
        while (true) {
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                break;
            }
            String str = (new Date()).toString();
            dc.send(ByteBuffer.wrap(str.getBytes()), new InetSocketAddress(group,
                port));
        }
    }
}
```

相应的接收消息的实现如代码清单 3-29 所示。在使用 DatagramChannel 类的 open 方法创建新的数据报文通道时，需要显式指定与所使用的组播分组地址相对应的 IP 协议的版本。另外，需要使用 setOption 方法把配置项 StandardSocketOptions.SO_REUSEADDR 的值设为 true，允许组播分组的不同成员绑定到相同的地址上。调用 DatagramChannel 类的 join 方法加入分组之后，可以用 receive 方法接收数据并进行处理。

代码清单 3-29 接收组播消息的客户端实现

```
public class TimeClient {
    public void start() throws IOException {
        NetworkInterface ni = NetworkInterface.getByName("eth1");
        int port = 5000;
        try (DatagramChannel dc = DatagramChannel.open(StandardProtocolFamily.
            INET)
            .setOption(StandardSocketOptions.SO_REUSEADDR, true)
            .bind(new InetSocketAddress(port))
            .join(InetAddress.getByName("224.0.0.2"))
            .register(SelectionKey.OP_READ)) {
            ByteBuffer buffer = ByteBuffer.allocate(1024);
            dc.receive(buffer);
            System.out.println(new String(buffer.array()));
        }
    }
}
```

```
        .setOption(StandardSocketOptions.IP_MULTICAST_IF, ni)) {  
  
    InetAddress group = InetAddress.getByName("224.0.0.1");  
    MembershipKey key = dc.join(group, ni);  
    ByteBuffer buffer = ByteBuffer.allocate(1024);  
    dc.receive(buffer);  
    buffer.flip();  
    byte[] data = new byte[buffer.limit()];  
    buffer.get(data);  
    String str = new String(data);  
    System.out.println(str); // 输出时间  
    key.drop();  
}  
}  
}
```

3.5 使用案例

下面通过一个具体的案例来说明如何使用 Java 中的 I/O 相关的功能。这个案例开发的是一个简单的基于文件系统中静态文件的 HTTP 服务器。在实现中需要用到文件和网络 I/O 操作相关的 API。基本的实现方式是把 HTTP 请求中的路径映射为文件系统中对应文件的路径，再把文件的内容作为 HTTP 请求的响应。在对 HTTP 请求的处理中，需要对一些常见的出错情况进行处理，如文件找不到的情况。HTTP 响应中也需要包含正确的 HTTP 头信息来指明文件的内容类型。

完整的服务器实现如代码清单 3-30 所示。处理 HTTP 请求时采用了异步套接字通道，并用一个包含 10 个线程的线程池来处理请求。对于每个 HTTP 请求，先读取客户端发送的请求的具体信息，从中提取出请求对应的路径，再把 HTTP 请求中的路径与服务器所管理的文件的根目录合并在一起，得到完整的文件路径。如果找不到对应的文件，服务器返回 404 错误；如果找到对应的文件，在 HTTP 响应中先输出 HTTP 头信息，再通过 Files 类的 copy 方法把文件输入流中包含的数据直接复制到 AsynchronousSocketChannel 类的对象所对应的输出流中。这一步相当于把 HTTP 请求头和内容都返回给客户端。如果在读取文件内容时出错，服务器返回 500 错误。

代码清单 3-30 基于文件系统中静态文件的 HTTP 服务器

```
public class StaticFileHttpServer {  
    private static final Logger LOGGER = Logger.getLogger(StaticFileHttpServer.  
        class.getName());  
    private static final Pattern PATH_EXTRACTOR = Pattern.compile("GET (.*)?  
        HTTP");  
    private static final String INDEX_PAGE = "index.html";  
  
    public void start(final Path root) throws IOException {
```

```

AsynchronousChannelGroup group = AsynchronousChannelGroup.
    withFixedThreadPool(10, Executors.defaultThreadFactory());
final AsynchronousServerSocketChannel serverChannel =
    AsynchronousServerSocketChannel.open(group).bind(new
    InetSocketAddress(10080));
serverChannel.accept(null, new CompletionHandler<AsynchronousSocketChann
el, Void>() {
    public void completed(AsynchronousSocketChannel clientChannel, Void
        attachement) {
        serverChannel.accept(null, this);
        try {
            ByteBuffer buffer = ByteBuffer.allocate(1024);
            clientChannel.read(buffer).get();
            buffer.flip();
            String request = new String(buffer.array());
            String requestPath = extractPath(request);
            Path filePath = getFilePath(root, requestPath);
            if (!Files.exists(filePath)) {
                String error404 = generateErrorResponse(404, "Not Found");
                clientChannel.write(ByteBuffer.wrap(error404.getBytes()));
                return;
            }
            LOGGER.log(Level.INFO, "处理请求: {0}", requestPath);
            String header = generateFileContentResponseHeader(filePath);
            clientChannel.write(ByteBuffer.wrap(header.getBytes())).get();
            Files.copy(filePath, Channels.newOutputStream(clientChannel));
        } catch (Exception e) {
            String error = generateErrorResponse(500, "Internal Server
                Error");
            clientChannel.write(ByteBuffer.wrap(error.getBytes()));
            LOGGER.log(Level.SEVERE, e.getMessage(), e);
        } finally {
            try {
                clientChannel.close();
            } catch (IOException e) {
                LOGGER.log(Level.WARNING, e.getMessage(), e);
            }
        }
    }
    public void failed(Throwable throwable, Void attachement) {
        LOGGER.log(Level.SEVERE, throwable.getMessage(), throwable);
    }
});
LOGGER.log(Level.INFO, "服务器已经启动, 文件根目录为: " + root);
}

private String extractPath(String request) {
    Matcher matcher = PATH_EXTRACTOR.matcher(request);
    if (matcher.find()) {
        return matcher.group(1);
    }
}

```

```
    }
    return null;
}

private Path getFilePath(Path root, String requestPath) {
    if (requestPath == null || "/".equals(requestPath)) {
        requestPath = INDEX_PAGE;
    }
    if (requestPath.startsWith("/")) {
        requestPath = requestPath.substring(1);
    }
    int pos = requestPath.indexOf("?");
    if (pos != -1) {
        requestPath = requestPath.substring(0, pos);
    }
    return root.resolve(requestPath);
}

private String getContentType(Path filePath) throws IOException {
    return Files.probeContentType(filePath);
}

private String generateFileContentResponseHeader(Path filePath) throws
    IOException {
    StringBuilder builder = new StringBuilder();
    builder.append("HTTP/1.1 200 OK\r\n");
    builder.append("Content-Type: ");
    builder.append(getContentType(filePath));
    builder.append("\r\n");
    builder.append("Content-Length: " + Files.size(filePath) + "\r\n");
    builder.append("\r\n");
    return builder.toString();
}

private String generateErrorResponse(int statusCode, String message) {
    StringBuilder builder = new StringBuilder();
    builder.append("HTTP/1.1 " + statusCode + " " + message + "\r\n");
    builder.append("Content-Type: text/plain\r\n");
    builder.append("Content-Length: " + message.length() + "\r\n");
    builder.append("\r\n");
    builder.append(message);
    return builder.toString();
}
}
```

3.6 小结

I/O 操作一直是程序开发中的重要组成部分。高效的 I/O 操作实现也是很多开发人员

所追求的目标。从概念层次来说，I/O 操作所表示的抽象含义并不复杂，只是把数据从一个地方传输到另外一个地方。但是，不同的传输实体本身的特征会使在其上进行的 I/O 操作有各自不同的特点，I/O 操作也需要根据这些实体的特征来做出相应的调整。本章主要侧重于介绍 Java I/O 操作中的底层抽象和重要 API 的使用。如果程序是基于 Java 7 来构建的，从通道开始着手是一个很好的选择，对于流则尽量少使用通道。了解 Java 7 增加的异步套接字通道和文件操作方面的功能，可以避免在开发中重复地发明一些实际上用不到的“轮子”，使用标准库通常总是一个更好的选择。

在开发高性能网络应用方面，Java 提供的标准库所支持的抽象层次过低，并不适合一般的开发人员直接使用通道。过多的底层细节和性能调优会耗费开发人员大量的精力，选用一个已有的网络应用开发库是一种更好的选择。Apache MINA[⊖] 和 JBoss Netty[⊖] 都是不错的库，可以作为开发的基础。

⊖ Apache MINA 库的网址是 <http://mina.apache.org/>。

⊖ JBoss Netty 库的网址是 <http://www.jboss.org/netty>。

第4章 国际化与本地化

提到应用程序的国际化和本地化，可能大多数开发人员不是特别熟悉，或者觉得与自己的日常开发关系并不大。实际上，很多开发人员一直对国际化和本地化存在着不少误解，其中最常见的一种观点是，如果一个程序只面向某种特定的区域设置（locale）中的用户，那就不需要提供国际化的支持。比如，某个程序只是针对中国用户开发的，那么在源程序中直接使用中文字符串也有可取之处。这种观点似乎有它的道理，但是实际上国际化更多地表示的是一种程序应该具有的能力。程序都应该具有国际化的能力，只不过可以根据自己的需要选择对哪些区域设置提供本地化支持，而不是因为仅打算支持一种本地化区域设置，就放弃国际化的能力。另外的一种观点是国际化只需要把程序中出现的用户会看到的字符串都提取出来，然后加上翻译就可以了。虽然展示给用户的消息的国际化是实现国际化的重要部分，但并不是全部。除了消息之外，日期和时间、数字和货币等内容也是需要提供国际化支持的。任何不是随便写着玩的程序，都应该具备相关的国际化的能力。

本章将要对 Java 提供的国际化和本地化支持进行详细介绍。从实际开发的角度来说，为 Java 程序添加国际化的支持并不是一件很困难的事情。Java 平台已经为开发人员提供了很多的抽象和简化。本章会介绍使用 Java 标准库的基本实践方式。除了与实际开发相关的部分之外，本章还会占用一些篇幅来介绍国际化底层的相关知识，尤其在 Unicode 和字符编码方面会着重介绍。Java 7 中新增的对区域设置的支持也是重点内容。

4.1 国际化概述

关于国际化的动机，大家应该都比较清楚。世界上很多国家或地区都有自己的语言和文字。一个程序要与普通用户进行交互，就应该提供目标用户所能理解的语言文本。同样，每个国家或地区也有自己惯用的日期、时间、数字以及货币的格式，程序应该使用用户惯用的格式来显示这些内容。

每种语言都是由一系列的字符按照某种规则组合而成的。这些字符是语言在发展过程中形成的。这些字符之间可能有内在的排列顺序关系，如英文的 26 个字母的顺序；也可能只是一个无序的集合，如中文的汉字。在计算机程序中处理输入数据和输出结果的时候，总是免不了与这些字符打交道。如何在计算机程序中表示这些字符，就成了一个需要解决的问题。由于计算机内部对数据的表示都是 0 和 1 的序列，表示这些字符的问题实际上就等价于如何在字符与 0 和 1 的序列之间建立一个映射关系。

通常的做法是先对由语言中所有字符组成的字符集（character set）按照一定的规则

进行编码。最直接的编码方式是为每个字符在给定的范围之内分配一个整数序号。这个整数序号被称为该字符的代码点（code point）。比如一个字符集中包含 1000 个字符，就可以把这 1000 个字符分别编号为从 1 到 1000。对于一个字符集来说，其中每个字符的代码点的分配方式是固定的，一般由相关的标准化组织来制定。在经过这种编码之后，之前的字符集就变成了编码字符集（coded character set）。目前存在非常多的编码字符集，开发人员熟悉的包括 ASCII、Unicode、GBK 和 GB2312 等。以 ASCII 为例，英文字母、数字以及常见的符号都在 ASCII 中有相应的编码，如字母“A”的编码是 65，而“a”的编码则是 97。当计算机程序处理以 ASCII 来编码的文本的时候，如果识别出来的编码是 65，则认为这是一个字母“A”。在进行输出的时候，用户就会看到相应形状的字符出现。

有了编码字符集之后，下一步要做的就是把字符的代码点映射到计算机中。最直接的映射方式是一一映射，即把字符的代码点直接映射到对应的数字。比如 ASCII 中的字母“A”就用整数 65 来表示。这种映射方式直接而易懂，实现起来也很简单，但是并没有考虑字符集中的字符总数和字符的使用频率。不同编码字符集中所包含的字符个数是大不相同的，如 ASCII 就只有 128 个字符，而 Unicode 中可使用的字符数多达 1 114 112 个。如果使用一一映射的方式，对于 ASCII 来说，只需要 7 位就足够了；而对于 Unicode 来说，则最少需要 21 位来表示。对于包含字符较多的编码字符集，一一映射的方式带来的问题是造成存储空间上的浪费。以 Unicode 为例，Unicode 字符集包含 ASCII 字符集中的所有字符。如果使用一一映射，每个 Unicode 字符都需要使用 21 位来表示。如果一段 Unicode 文本只包含 ASCII 字符，使用 ASCII 编码就可以完全表示；而使用一一映射方式的 Unicode 编码所需的空间是使用 ASCII 编码的 3 倍。在实际的使用中，ASCII 字符的使用频率要远高于其他 Unicode 字符。所以这种一一映射的方式对于 Unicode 这样庞大的编码字符集来说并不可取。

从节省空间和提高使用效率的角度出发，对于 Unicode 这样的编码字符集，更好的办法是采用多个代码单元（code unit）来表示一个字符。一个代码单元是编码时使用的最小单元。每个字符所使用的代码单元个数是不确定的。如果采用 8 位作为代码单元，那么 Unicode 中的 ASCII 字符就可以直接用 8 位来表示；其他语言中的常用字符可以用 16 位来表示；另外一些比较少见的字符，可以使用更多的位来表示。通过这种划分方式，既可以保证常用字符所占用的存储空间尽可能少，又不会对表达能力产生影响。

4.2 Unicode

不同的国家和地区通常会有自己特定的编码字符集，而且以包含某种语言相关的字符为主。这使得支持国际化的工作变得复杂，因为需要同时考虑非常多的编码字符集。这些编码字符集的字符的代码点各不相同，为了提高效率也需要使用不同的代码单元来进行映射，有可能不同的编码字符集用相同的编码来表示不同的字符，或是用不同的编

码来表示相同的字符。对于平台和应用开发人员来说，理解这些编码字符集并在程序中正确地使用，所要花费的代价太高。

Unicode 就是为了解决这个问题而出现的。Unicode 的目标是能够包括世界上所有语言的文字。以目前最新的 Unicode 6.0 来说，它已经包含了来自 93 种语言的超过 109 000 个字符。现在 Unicode 的开发和维护工作由 1991 年成立的 Unicode 联盟负责。该联盟的成员包括了大部分重量级的信息技术公司。Unicode 除了在各种主流操作系统和应用平台上得到应用之外，同时也与国际标准化组织（ISO）的 ISO/IEC 10646 标准保持完全同步。从这两个方面来说，Unicode 应该是开发国际化应用程序时的编码字符集的最佳选择。随着 Unicode 本身的发展，越来越多的语言将被包括进来。除了语言中的字符之外，Unicode 还包含了各种常见的符号。

4.2.1 Unicode 编码格式

由于 Unicode 中包含的字符非常多，另外为了尽可能地保持与已有编码字符集的兼容性，Unicode 中对字符的编码格式比较复杂。Unicode 一共定义了 1 114 112 个代码点，值的范围从 0 到 0x10FFFF。在下面对代码点的说明中，都是以 16 进制的方式来表示的，并在前面加上“U+”。这一百多万个代码点中目前使用的只有 10% 左右，大部分代码点目前还没有被使用，可以在以后的版本中被分配给新添加的语言的字符。Unicode 中的代码点被分成 17 个区域，每个区域为一个平面（plane）。每个平面中包含 65 536 个字符。第 1 个平面的代码点的范围是 0 到 U+FFFF，第 2 个平面的代码点的范围是 U+10000 到 U+1FFFF，依次类推，最后一个平面的代码点的范围是 U+F0000 到 U+10FFFF。也就是说代码点的后四位（65 536 个值）总是从 0 变化到 0xFFFF，而前两位则在 0 到 0x10 之间变化，即总计 17 个平面。代码点的前两位表明了代码点所在的平面的编号。在这 17 个平面中，目前只有前 4 个和后 3 个已经被使用了或被保留，中间的 10 个平面尚处于未分配的状态。

在 Unicode 的这 17 个平面中，最常见的是第 1 个平面，即代码点范围是 0 到 U+FFFF。这个平面被称为基本多语言平面（Basic Multilingual Plane, BMP），其中包含了最常见的语言中的字符和大量的符号。第 2 个平面称为补充多语言平面（Supplementary Multilingual Plane, SMP），其中主要包含一些不常见的语言中的字符以及音乐与数学符号。第 3 个平面称为补充表意文字平面（Supplementary Ideographic Plane, SIP），其中包含的是 Unicode 所定义的统一的汉字的表意符号。第 4 个平面称为第三表意文字平面（Tertiary Ideographic Plane, TIP），被保留用来包含其他的表意文字，目前其中还没有定义任何字符。第 5 到第 14 个平面前目前处于未分配的状态，也没有被保留使用。第 15 个平面称为补充特殊用途平面（Supplementary Special-purpose Plane, SSP），其中主要包含非图形化的字符。第 16 和第 17 个平面称为私有使用区域平面（Private Use Area planes, PUA）。这两个平面中的代码点所对应的字符不由 Unicode

联盟来规定，而允许第三方自行定义。相当于允许第三方在 Unicode 的框架之内开发自己的字符编码格式。

在说明了 Unicode 编码字符集中代码点的定义方式之后，下面来看看如何把这些代码点映射到计算机程序中。Unicode 没有采用一一映射的方式，而是把字符的代码点分别映射到多个代码单元上。Unicode 规范中定义的映射方式分成 Unicode 传输格式（Unicode Transformation Format, UTF）编码和统一字符集（Universal Character Set, UCS）编码两种。这两种映射方式根据代码单元的位数的不同，又有不同的变体，比如 UTF-8、UTF-16 和 UTF-32 就分别使用 8 位、16 位和 32 位来表示单个代码单元；而 UCS-2 和 UCS-4 则分别使用 2 个字节和 4 个字节来表示单个代码单元。这些映射方式中最常见的是 UTF 编码格式的两种变体 UTF-8 和 UTF-16，而 UCS 编码格式用得比较少。

1. UTF-8

UTF-8 编码格式应该是最为开发人员所熟悉的 Unicode 编码格式。有很多图书和教程都告诉开发人员在编写网页的时候应该使用 UTF-8 编码，在创建数据库的时候也需要使用 UTF-8 编码。很多开发人员都认为使用 UTF-8 编码是解决程序中各种乱码问题的终极解决方案，不过经过本章的详细介绍之后，开发人员应该会对乱码问题有更加深刻的认识，而不是仅了解这样一个结论。UTF-8 编码格式能够流行是有原因的。UTF-8 是一种变长的编码格式，其中每个代码单元是 8 位。一个 Unicode 字符的代码点会被映射到 1 到 6 个代码单元。这种变长编码格式的一个重要的好处是可以减少所需要的存储空间。对于 Unicode 中的常见字符，仅用一个代码单元就可以表示。另外，UTF-8 编码的前 128 个字符与 ASCII 编码是完全一致的。也就是说一段用 ASCII 编码的文本也是一段合法的 UTF-8 编码的文本。考虑到 ASCII 编码的流行程度，保持 UTF-8 编码与 ASCII 编码的这种兼容性，有利于 UTF-8 的推广。

由于 UTF-8 是变长编码的，它对不同范围内的 Unicode 代码点的编码方式是不同的，所以 UTF-8 的编码和解码方式有点复杂。不过 UTF-8 编码格式本身设计得非常精巧[⊖]。表 4-1 给出了具体的编码分配方式，其中的“x”表示的是可以用来编码的位。

表 4-1 UTF-8 编码方式

代码点位数	代码点范围	字节 1	字节 2	字节 3	字节 4	字节 5	字节 6
7	0~U+007F	0xxxxxxx					
11	U+0080~U+07FF	110xxxxx	10xxxxxx				
16	U+0800~U+FFFF	1110xxxx	10xxxxxx	10xxxxxx			
21	U+10000~U+1FFFFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx		
26	U+2000000~U+3FFFFFFF	111110xx	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx	
31	U+4000000~U+7FFFFFFF	1111110x	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx

[⊖] UTF-8 格式的主要设计者之一是 1973 年图灵奖获得者之一的 Ken Thompson，他因设计 UNIX 而闻名。

如表 4-1 所示，首先是用 1 个字节来表示的 Unicode 字符。为了保持与 ASCII 的兼容性，这个字节中的高位始终是 0，仅用剩下的 7 位来表示字符，因此可以表示的 Unicode 中的字符的代码点范围是 0 ~ U+007F，对应的编码之后的代码值是 0 ~ 0x007F。接着是用 2 个字节表示的 Unicode 字符。在这两个字节中，前一个字节的前 3 位固定为 110，后一个字节的前 2 位固定为 10，剩下的 11 位用来编码。这两个字节可以表示的代码点范围是 U+0080 ~ U+07FF。进行编码就是把代码点的 11 位按照从高到低的顺序分成 5 位和 6 位两个部分，分别放在第一个和第二个字节中的剩余部分，这样就得到了对应的 UTF-8 编码。对于由 3 个字节表示的 Unicode 代码点来说，第一个字节的前 4 位固定为 1110，后面两个字节的前 2 位固定为 10。这样 3 个字节总共剩下 16 位用来编码，可以表示的代码点范围是 U+0800 ~ U+FFFF。利用 3 个字节进行编码的方式与利用 2 个字节的做法相似，即把 16 位按照从高到低的顺序分配到各个字节的剩余位中。对于剩下的分别用 4、5 和 6 个字节来表示的 Unicode 代码点来说，基本的编码方式是相似的。从上面对不同字节数的编码方式的分析中可以看出编码时的规律。如果使用的字节数多于一个，那么第一个字节中的高位固定为多个 1 后面加上一个 0，其中 1 的个数等于使用的字节数。而除了第一个字节之外的其他字节的高位都固定以二进制的“10”开头。利用这种编码方式可以很容易地对一个 UTF-8 编码的字节序列进行解码。只需要查看第一个字节的高位中第一个 0 之前的 1 的个数，就可以知道整个编码序列由几个字节组成。同样的，当遇到一个字节的时候，只需要查看其前 2 位就可以知道该字节在 UTF-8 编码之后的字节序列中的位置：如果第一位是 0，说明这是一个与 ASCII 兼容的编码；如果前两位都是 1，说明这是一个多字节编码的起始字节；如果前两位是 10，说明这是一个多字节编码的后续字节，只需要往前查找就可以找到整个编码序列的起始字节。

2. UTF-16

除了 UTF-8 之外，另一个常用的 Unicode 编码格式是 UTF-16。与 UTF-8 不同的是，UTF-16 中的每个代码单元是 16 位。每个 Unicode 代码点会被映射到 1 或 2 个 16 位的代码单元上。在使用 UTF-16 的时候，最多只需要 4 字节就可以表示所有的字符；而在使用 UTF-8 的时候，则最多需要 6 字节。这是因为 UTF-8 中有很多位的值是固定的，不能用在编码中。就 UTF-8 和 UTF-16 的使用场景来说，UTF-8 更多是作为字符传输时的编码格式，而 UTF-16 则更多是作为字符在系统中的内部表示方式。这是因为 UTF-8 编码格式可以减少传输时所需的字节数，而 UTF-16 使用起来相对简单。使用 UTF-16 对 Unicode 中的代码点进行编码的过程也比较复杂。

对于 Unicode 的 BMP 中的 65 536 个字符，这些字符是直接将代码点一一映射到 16 位整数值上的。BMP 中的代码点只需要一个代码单元就可以表示。对于不在 BMP 中的代码点，由于其数值范围已经超过了 16 位所能表示的范围，所以需要两个 16 位的代码单元才能表示。这两个代码单元被称为一个代理项对（surrogate pair）。由于不

在 BMP 中的代码点的范围是 U+10000 ~ U+10FFFF，对这些代码点的具体的映射规则是：用代码点的数值减去 0x10000，这样就得到了 0 ~ 0xFFFF 范围内的 20 位的数值。用这 20 位中的前 10 位加上 0xD800 之后作为代理项对的高位代理，得到的值的范围是 0xD800 ~ 0xDBFF；再把这 20 位中剩下的 10 位加上 0xDC00 之后作为代理项对的低位代理，得到的值的范围是 0xDC00 ~ 0xDFFF。把高位代理和低位代理拼接起来，就得到了由两个 16 位值组成的代码点的 UTF-16 的表示形式。代码清单 4-1 给出了一个 UTF-16 的示意编码过程。

代码清单 4-1 UTF-16 的示意编码过程

```
public char[] encode(int codePoint) {
    if ((codePoint >= 0 && codePoint <= 0xD7FF)
        || (codePoint >= 0xE000 && codePoint <= 0xFFFF)) {
        return new char[] {(char) codePoint};
    } else {
        codePoint = codePoint - 0x10000;
        int high = (codePoint >> 10) + 0xD800;
        int low = (codePoint & 0x3FF) + 0xDC00;
        return new char[] {(char) high, (char) low};
    }
}
```

值得一提的是，在 Unicode 编码字符集中，U+D800 ~ U+DFFF 中间的代码点是没有定义字符的。这么做就是为 UTF-16 编码考虑的。经过这样的设计，对于编码之后的字节序列，以 16 位为一个单元来看，仅使用一个 16 位的 BMP 中字符的编码和使用两个 16 位的字符编码中的高位和低位代理项，这三者的数值范围是互相不重叠的。这种不重叠的设计，使得在解码的时候可以很容易地判断一个字节序列中不同代码点所对应的字节范围。比如对一个 16 位值来说，如果其数值在 0 ~ 0xD7FF 或 0xE000 ~ 0xFFFF 之间，就说明这是一个 BMP 中的字符的编码，使用这 16 位来解码就足够了；如果数值范围在 0xD800 ~ 0xDBFF 之间，就说明这是一个代理项对的高位代理，应该再往后查看 16 位；如果数值范围在 0xDC00 ~ 0xDFFF 之间，就说明这是一个代理项对的低位代理，应该再往前查看 16 位。

UTF-16 和 UTF-8 编码格式都具有“自我同步”的特性。这种特性的含义是，一个代码点对应的编码后的字节序列的起点和终点位置只需要查看当前代码点就可以得出来。也就是说，在一段文本被编码之后的字节序列中，任意指定一个字节位置，最多只需要查看一个代码点对应的字节序列长度的字节数，就可以找到这个字节所属的代码点所对应的完整字节序列。这种特性的优势使解码过程变得比较简单，可以很容易地把一个完整的字节序列切分成不同的小段，每个小段对应一个代码点。

由于 UTF-16 通过 2 个或 4 个字节来编码一个代码点，因此这些字节之间的顺序就变得有意义了。这就是第 3 章介绍过的字节顺序的含义。同样的 UTF-16 编码，按

照大端表示 (big-endian) 和小端表示 (little-endian) 所解释出来的字符是不一样的。对于这种情况，UTF-16 允许在一段编码之后的字节序列的最前面使用字节顺序标记 (byte order mark, BOM) 来说明解码时应该使用的字节顺序。字节顺序标记是一个特殊的 Unicode 代码点 U+FEFF，所表示的字符含义是宽度为零的不断行的空格 (ZERO-WIDTH NON-BREAKING SPACE)。当解码器进行解码时，它会根据底层平台的字节顺序去尝试解码。如果解码之后的结果是 0xFEFF，则说明不需要改变字节顺序，直接使用底层平台的字节顺序即可；如果解码之后的结果是 0xFFFF，即两个字节的顺序发生了调换，由于 U+FFFF 在 Unicode 规范中不对应任何字符，因此实际上产生了解码错误。出现这个错误就说明解码时的字节顺序不对，应该把字节调换之后再进行解码。解码器就会在每次读取两个字节的时候，先把顺序颠倒一下，再进行解码。除了使用 BOM 之外，另外一种指定字节顺序的做法是在编码格式上给出具体的声明：UTF-16BE 说明采用的是大端表示的 UTF-16 编码，而 UTF-16LE 说明采用的是小端表示的 UTF-16 编码。

UTF-16、UTF-16BE 和 UTF-16LE 在处理字节顺序标记时的行为是不同的。对于 UTF-16BE 和 UTF-16LE 来说，在编码的时候，不会输出字节顺序标记，因为编码格式的名称就指明了字节顺序；在解码的时候，会把字节顺序标记当成其对应的普通 Unicode 字符，即代码点为 U+FEFF 的字符。而对于 UTF-16 来说，在编码的时候总是使用大端表示并输出字节顺序标记；在解码的时候则根据字节顺序标记进行判断，如果没有就默认为大端表示。

3. UTF-32 与 UCS-2

另外一个使用得比较少的编码格式是 UTF-32。UTF-32 总是使用 32 位来编码一个 Unicode。由于 32 位对于 Unicode 的代码点范围来说完全足够了，因此 UTF-32 是一种定长编码格式，不同于 UTF-8 和 UTF-16 的变长编码格式。Unicode 代码点与 UTF-32 的 32 位之间是一一映射的关系。UTF-32 的优势在于支持对代码点的随机查找，这也是其定长编码带来的好处。比如一段文本中包含了 128 个 Unicode 字符，那么在经 UTF-32 编码之后的字节序列中，只需要定位到第 13 个字节，就找到了第 4 个字符的起始位置。而对于 UTF-8 和 UTF-16 来说，则只能进行顺序查找，因为每个代码点所对应的字节数是不相同的。UTF-32 的最大缺点在于对存储空间的浪费，这也是前面说过的一一映射的编码方式的共同缺点。因为这个缺点，在实际的开发中很难见到 UTF-32 的影子。一般的开发人员也不需要特别关注 UTF-32 编码格式。

还有一个可能会遇到的 Unicode 编码格式是 UCS-2。UCS-2 是在国际标准 ISO/IEC 10646 中定义的，是 UTF-16 的一个子集。UCS-2 总是使用 2 字节来表示一个 Unicode 编码。因此它只能对 BMP 中的代码点进行编码。在 UCS-2 的基础上进行扩充，增加了对非 BMP 中代码点的支持之后，就得到了 UTF-16 编码。UCS-2 编码会出现在一些规范和应用平台的早期版本中，现在基本都改为使用 UTF-16。

4.2.2 其他字符集

除了 Unicode 之外，还存在很多其他的编码字符集。如前面提到过的使用 7 位的包含 128 个字符的 ASCII 编码。ASCII 编码主要是为英语环境而开发的。另外一个常见的字符集是 ISO 8859-1，这个字符集中主要包含了绝大部分的西欧语言中的字符。ISO 8859 其实是一个包含了 16 个部分的字符编码规范，其中以第一部分，即 ISO 8859-1 最为常用。字符集 ISO 8859-1 也是通过 HTTP 协议获取到的文本类型的文档的默认编码格式。在中文编码方面，最常见的是 GB2312、GBK 和 GB18030 这三种字符集，其中 GB2312 是最早的国家规范，而 GBK 在 GB2312 的基础上进行了扩展，GB18030 则是最新的国家规范。

4.2.3 Java 与 Unicode

Java 平台从诞生之初就对 Unicode 提供了支持。Java 7 支持最新的 Unicode 6.0 规范。Java 语言中的字符最早用的是 UCS-2 编码格式。因此在设计之初，字符类型 char 是由 2 字节来表示的，只能表示 Unicode 中 BMP 中的字符。后来从 J2SE 5.0 开始支持 UTF-16，从而可以表示 Unicode 中 BMP 之外的其他字符。也就是说，如果是 BMP 中的字符，一个 char 类型单元就足够了；如果是不在 BMP 中的字符，则需要由两个 char 类型单元通过代理项对的方式来实现，而且这两个 char 类型单元要连续出现。这样带来的问题是，Java 语言中的 char 类型与 Unicode 中的字符不再是一一对应的关系。如果得到了一个长度为 3 的 char 类型的数组，那么其中可能包含 2 个或 3 个 Unicode 字符。如果在 Java 语言的设计之初就采用长度为 4 字节的 char 类型，就不会出现这种不一致的情况。当然，这些问题在设计之初也是不可能全部预计到的，否则也不会有类似“千年虫”这样的问题出现。

从 J2SE 5.0 开始，Java 中也可以用 int 类型来表示 Unicode 中的代码点。使用 int 类型的时候只会用到对应的 32 位中的后 21 位，前 11 位值全部为 0。这实际上相当于使用了 UTF-32 编码格式。同时，java.lang.Character 类也增加了与 Unicode 代码点和 UTF-16 编码相关的实用方法。代码清单 4-2 给出了 Character 类的使用示例，其中 codePointAt 方法可以获取到一个 java.lang.CharSequence 接口的实现对象或一个 char 类型数组中从给定位置开始的字符的 Unicode 代码点。这个方法可以处理代理项对。如果当前位置的 char 类型的值是在 UTF-16 的代理项对的高位的范围之内，同时接下来的 char 类型的值也在代理项对的低位的范围之内，会把这两个 char 按照 UTF-16 格式解码之后再得到其代码点的值。而 isBmpCodePoint 方法可以判断一个代码点是否在 BMP 之内。对于非 BMP 中的字符，可以通过 highSurrogate 和 lowSurrogate 方法来分别得到其对应的代理项对的高位和低位的 char 值。

代码清单 4-2 Character 类的使用示例

```

public void useCharacter() {
    String str = "你好";
    int codePoint = Character.codePointAt(str, 0); // 值为 20320
    Character.isBmpCodePoint(codePoint); // 值为 true
    int smpCodePoint = 0x12367;
    Character.isSupplementaryCodePoint(smpCodePoint); // 值为 true
    Character.charCount(smpCodePoint); // 值为 2
    char high = Character.highSurrogate(smpCodePoint);
    char low = Character.lowSurrogate(smpCodePoint);
}

```

4.3 Java 中的编码实践

对 Java 程序来说，由于 Java 平台内部统一使用 UTF-16 的编码格式，因此在单个程序内部并不会存在编码问题。真正的编码问题发生在程序与外界发生字符数据传递的时候。可能的交互情况包括用户输入数据到程序中，其他程序产生的输入文件被当前程序所使用，程序传递参数给操作系统底层方法调用，以及程序产生输出文件等。也就是说，当一个字节序列跨越当前程序的边界，而且这个字节序列需要当成字符来处理的时候，就可能产生编码的问题，因为这些字节所表示的字符会因编码格式的不同而千差万别。而在程序内部，总是可以使用统一的编码格式来处理字符，这也是 Java 程序中会产生各种乱码问题的原因。基本的解决思路也很简单，那就是在程序与外界进行输入输出的边界严格控制字符的编码格式，确保这些字符以正确的 UTF-16 格式进入到程序内部。

以乱码问题较多的 Web 应用为例进行说明。一个 Web 应用通常都允许用户输入一些字符来与应用进行交互。这些字符通过 HTTP 协议来传输，在发送的时候就会有一个自己的编码格式。如果希望把用户输入的字符保存到数据库之中，那么还需要考虑数据库本身存储时使用的编码格式。最后把 HTML 页面通过 HTTP 协议传输给浏览器的时候，也需要指定一个合适的编码。浏览器会根据此编码来对页面进行处理。在上面这些环节中，如果有对编码处理不当的地方，都可能造成用户看到的不是他想要的内容，而是一堆无法识别的字符。

由于存在非常多的编码字符集，每个字符集主要面向的国家和地区也不尽相同。不同的 Java 虚拟机实现可以选择支持不同的字符集编码方式。但是，在这些编码方式中，ASCII、ISO 8859-1、UTF-8、UTF-16、UTF-16BE 和 UTF-16LE 是虚拟机必须支持的，也就是说虚拟机肯定支持这 6 种编码格式的编码和解码。由于 Java 平台原生使用的是 UTF-16 编码，因此这里的编码和解码的概念，指的是 UTF-16 编码的 Unicode 字符与该字符在不同字符集中表示方式之间的转换。如果 Java 虚拟机的实现支持 GB18030，而需要处理的字符串是“你好”，那么使用 GB18030 进行编码的含义是把“你好”对应的 UTF-16 编码格式的字节序列 0x4F60597D 转换成 GB18030 中同样表示“你好”的字节

序列 0xC4E3BAC3；而解码的过程正好相反。如果需要进行编码转换的两种格式都不是 UTF-16，就需要使用 UTF-16 作为中间格式。

4.3.1 Java NIO 中的编码器和解码器

在 J2SE 1.4 之前，Java 提供的对不同字符集的编码和解码的支持比较少，主要是通过 String 类的 getBytes 方法和构造方法来完成相应的转换。比如希望把字符串从 UTF-16 编码变成 GB18030 编码，可以执行代码 “str.getBytes("GB18030")”，返回的结果是一个字节数组；同样，如果已有一个字节数组和它的编码格式，可以通过 String 类的构造方法来创建一个 String 类的对象，如代码 “new String(data, "GB18030")” 可以把 GB18030 编码的字节数组 data 转换成 UTF-16 格式的内部表示。J2SE 1.4 在引入 NIO 的时候，提供了对字符集的更多支持能力，即新的 java.nio.charset 包。这其中最重要的是表示字符集的 java.nio.charset.Charset 类及进行编码和解码的 java.nio.charset.CharsetEncoder 和 java.nio.charset.CharsetDecoder 类。Java 中的字符集 (charset) 实际上由上面介绍的编码字符集和相关的映射方式两部分所组成。

开始使用 Charset 类之前需要得到 Charset 类的对象，一般来说有 4 种方法可以使用。如果希望使用 Java 虚拟机都支持的字符集，可以从 java.nio.charset.StandardCharsets 类的公开静态变量中得到，如 StandardCharsets.UTF_16 表示 UTF-16 字符集；如果希望使用 Java 虚拟机平台默认的字符集，可以使用 Charset 类的 defaultCharset 方法。这个默认字符集是在 Java 虚拟机启动时得到的，一般取决于底层操作系统的区域设置；还可以通过 Charset 类的 availableCharsets 方法得到当前 Java 虚拟机上支持的所有字符集，再从中进行选择；最后可以根据字符集的名称来通过 Charset 类的 forName 方法进行创建。

得到了 Charset 类的对象之后的典型用法是创建出对应的编码器和解码器对象。通过 newEncoder 方法可以创建一个 CharsetEncoder 类的对象，而通过 newDecoder 方法可以创建出一个 CharsetDecoder 类的对象。CharsetEncoder 和 CharsetDecoder 类在进行编码和解码的时候都是有内部状态的，由一系列相关的动作来完成整个编码或解码过程。下面以 CharsetEncoder 类的使用为例进行说明。

在使用 CharsetEncoder 类的对象之前，除非这个 CharsetEncoder 类的对象是新创建的，否则一般需要调用 reset 方法来重置其内部状态。接着就需要多次调用 encode 方法来进行编码。encode 方法有 3 个参数：第一个参数是包含待编码字符的 CharBuffer 类的对象，第二个参数是存放编码之后的结果的 ByteBuffer 类的对象，最后一个参数用来表示是否还有更多的输入内容。在多次的 encode 方法调用之后，需要调用 flush 方法来清空 CharsetEncoder 类的对象的内部缓冲区。在进行编码时，encode 方法的返回值是 java.nio.charset.CoderResult 类的对象，用来表示本次编码过程的执行结果。一般来说有 4 种不同的结果：第一种用 CoderResult.UNDERFLOW 来表示，说明编码的输入缓冲区中的内容已经被完全耗尽，或是需要额外的输入才能继续当前的编码过程；第二种用

CoderResult.OVERFLOW 来表示，说明编码的输出缓冲区中没有足够的空间来存放编码之后的结果；第三种结果是输入缓冲区中的字符不是合法的 UTF-16 格式，可以通过 CoderResult 的 isMalformed 方法来判断；第四种结果是输入缓冲区中的某些字符在当前字符集中无法表示，可以通过 CoderResult 的 isUnmappable 方法来判断。

CharsetEncoder 类的对象的调用者应该根据 encode 方法的执行结果来采取不同的处理方式。对于返回值为 CoderResult.UNDERFLOW 的情况，应该在向输入缓冲区中添加更多的内容之后，再次调用 encode 方法；对于返回值为 CoderResult.OVERFLOW 的情况，应该读取输出缓冲区中的内容以腾出更多的空间供下次 encode 方法调用来使用。对于后面两种返回值结果来说，一般有 3 种处理方式，定义在 java.nio.charset.CodingErrorAction 类中。这 3 种处理方式分别是报告（CodingErrorAction.REPORT）、替换（CodingErrorAction.REPLACE）和忽略（CodingErrorAction.IGNORE）。报告的含义是指通过 encode 方法的返回值来说明错误的信息；替换的含义是用特定的字节数组来替换无效或无法映射的字符；忽略的含义则是指直接跳过输入中引起错误的字符。通过 CharsetEncoder 类的 onMalformedInput 和 onUnmappableCharacter 方法可以设置当 CharsetEncoder 类的对象遇到无效或无法映射的字符时的处理行为，参数就是上面说到的 CodingErrorAction 类的对象。如果采用替换的做法，来进行替换的字节数组的值可以通过 CharsetEncoder 类的对象的 replaceWith 方法来改变，而通过 replacement 方法也可以得到当前的替换值。

先用一个简单的例子来说明 CharsetEncoder 的用法。代码清单 4-3 中将字符“你好”编码成 UTF-8 格式，并输出相应的字节数组。这段代码的作用与“"你好".getBytes("UTF-8")”是相同的。

代码清单 4-3 CharsetEncoder 类的使用示例

```
public void simpleEncode() {
    Charset charset = StandardCharsets.UTF_8;
    CharsetEncoder encoder = charset.newEncoder();
    CharBuffer inputBuffer = CharBuffer.allocate(256);
    inputBuffer.put("你好").flip();
    ByteBuffer outputBuffer = ByteBuffer.allocate(256);
    encoder.encode(inputBuffer, outputBuffer, true);
    encoder.flush(outputBuffer);
    outputBytes(outputBuffer); // 值为 0xE4BDA0E5A5BD
}
```

代码清单 4-3 只是简单地用 encode 方法来编码一小段文本，所以并没有考虑前面提到的输入和输出缓冲区的数据的问题。下面考虑一个更加复杂的例子，即对一个文件进行完整的编码过程。示例用的文件是一个保存的 UTF-8 编码的网页，通过编码之后转换成用 GB18030 来编码。编码之后的网页，如果直接用浏览器打开就会出现乱码。这个时候手动切换浏览器采用的编码格式为 GB18030 就可以正确地显示了。这也验证了编码的

过程是正确无误的。如代码清单 4-4 所示，在创建出 GB18030 编码格式的编码器之后，首先把编码器遇到无效和无法映射的字符的处理行为设成“忽略”，避免在 encode 方法中产生这两种错误，从而不需要在代码中进行相应的处理。接着对于输入文件中的每一行，用一个 CharBuffer 类的对象封装之后作为 encode 方法的输入。每次调用 encode 方法之后检查调用结果，如果是 CoderResult.OVERFLOW，说明输出缓冲区已满，应该把结果写入到输出文件中；如果是 CoderResult.UNDERFLOW，说明输入缓冲区中的字符已经被编码完毕，可以开始下一行的编码工作。所有行都遍历完成之后，还需要调用一次 encode 方法并指示已经没有其他的输入。最后调用 flush 方法来清空内部缓冲区。在这里需要注意的是 flush 方法的返回值有可能是 CoderResult.UNDERFLOW 或 CoderResult.OVERFLOW。如果是 CoderResult.UNDERFLOW，则说明清空操作成功完成；如果是 CoderResult.OVERFLOW，则说明调用 flush 方法时传入的缓冲区不够大，应该创建一个更大的 ByteBuffer 类的对象并重新调用 flush 方法。

代码清单 4-4 对整个文件进行编码的示例

```

public void encodeFile() throws IOException {
    Charset charset = Charset.forName("GB18030");
    CharsetEncoder encoder = charset.newEncoder();
    encoder.onMalformedInput(CodingErrorAction.IGNORE);
    encoder.onUnmappableCharacter(CodingErrorAction.IGNORE);
    ByteBuffer outputBuffer = ByteBuffer.allocate(128);
    List<String> lines = Files.readAllLines(Paths.get("test.htm"),
        StandardCharsets.UTF_8);
    try (FileChannel destChannel = FileChannel.open(Paths.get("result.htm"),
        StandardOpenOption.CREATE, StandardOpenOption.WRITE)) {
        for (String line : lines) {
            CharBuffer charBuffer = CharBuffer.wrap(line);
            while (true) {
                CoderResult result = encoder.encode(charBuffer, outputBuffer,
                    false);
                if (result.isOverflow()) {
                    writeToChannel(destChannel, outputBuffer);
                } else if (result.isUnderflow()) {
                    break;
                }
            }
        }
        writeToChannel(destChannel, outputBuffer);
        encoder.encode(CharBuffer.allocate(0), outputBuffer, true);
        CoderResult result = encoder.flush(outputBuffer);
        if (result.isOverflow()) {
            ByteBuffer newBuffer = ByteBuffer.allocate(1024);
            encoder.flush(newBuffer);
            writeToChannel(destChannel, newBuffer);
        }
    } else {

```

```

        writeToChannel(destChannel, outputBuffer);
    }
}

private void writeToChannel(WritableByteChannel channel, ByteBuffer buffer) throws
    IOException {
    buffer.flip();
    channel.write(buffer);
    buffer.compact();
}

```

如果不需要像代码清单 4-4 那样考虑完整的编码过程，可以直接使用 encode 方法的另外一个重载形式，即使用一个 CharBuffer 类的对象作为参数表示输入字符，而返回一个 ByteBuffer 类的对象作为编码之后的结果。这个方法在内部实现了一套完整的编码过程，开发人员不需要考虑过多的底层细节。编码过程中如果出现错误，会抛出 java.nio.charset.CharacterCodingException 异常。

解码器 CharsetDecoder 类的使用可以完全将编码的过程反过来。解码是通过 decode 方法来实现的，其中的流程类似编码，分成重置、循环解码和清空内部缓冲区这 3 个步骤。唯一的区别在于 decode 方法调用时的输入变成了 ByteBuffer 类的对象，而输出则变成了 CharBuffer 类的对象。CharsetDecoder 类的具体使用可以类比上面介绍的 CharsetEncoder 类，这里不再赘述。关于 CharsetDecoder 类需要额外提到的一点是，有些解码器支持编码的自动检测，也就是说这些编码器并不是固定地只能支持一种编码格式的解码。通过 isAutoDetecting 方法可以判断一个 CharsetDecoder 类的对象是否支持编码的自动检测。当解码器完成对某些字节的处理之后，就有可能已经检测出字符集。通过 isCharsetDetected 可以判断是否已经检测出字符集。一旦检测出来之后，可以通过 detectedCharset 方法来获取检测出来的字符集。

现在的程序一般都使用 Unicode 作为内部的字符集，很多应用开发平台都直接支持 Unicode。但是还是存在某些遗留应用只支持某种特定的字符集，或有些应用在设计上就只使用某个固定的字符集。比如只面向英语用户的程序，可以选择使用 ASCII 作为其字符集。当为这样的程序提供输入的时候，需要先对 Unicode 字符串进行过滤，去掉其中不能被目标程序识别的字符。通过编码器和解码器可以实现对字符的过滤，具体的做法是，指示 CharsetDecoder 类的对象在解码过程中遇到无法映射的字符时直接忽略即可。代码清单 4-5 给出了一个示例，对字符串按照 ISO 8859-1 字符集进行过滤。经过过滤后的字符串不会包含无法通过 ISO 8859-1 字符集表示的字符。

代码清单 4-5 字符串过滤的示例

```

public String filter(String str) throws CharacterCodingException {
    Charset charset = StandardCharsets.ISO_8859_1;
    CharsetDecoder decoder = charset.newDecoder();

```

```

        CharsetEncoder encoder = charset.newEncoder();
        encoder.onUnmappableCharacter(CodingErrorAction.IGNORE);
        CharBuffer buffer = CharBuffer.wrap(str);
        ByteBuffer byteBuffer = encoder.encode(buffer);
        CharBuffer result = decoder.decode(byteBuffer);
        return result.toString();
    }

    public void useFilter() throws CharacterCodingException {
        String result = filter("你好，123 世界！"); // 值为 123
    }
}

```

4.3.2 乱码问题详解

在 Java 程序中，一个常见的与编码相关的问题就是乱码问题，尤其是与中文相关的乱码问题。中文乱码问题和 Java 类路径（CLASSPATH）的问题一样，是很多刚接触 Java 的开发人员都会遇到的问题。经过前面的介绍，读者应该对 Java 中的编码格式有了一定的了解。所谓的乱码，指的是某种编码格式产生的字节序列被错误地用另外一种不兼容的编码格式进行解码，得到的结果通常是一些奇怪的字符。乱码的问题通常发生在字符串在 Java 程序的边界之间传递的时候，尤其是在字符串输入的时候。当一个字符串传入 Java 程序的时候，如果没有正确地指定其使用的编码格式，Java 程序可能采用错误的编码格式进行解码，因此得到了错误的 UTF-16 字节序列。而在 Java 程序产生输出的时候，问题则相对较少。Java 程序总是可以使用 UTF-16 或 UTF-8 作为其输出内容的编码格式。从应用的类别来说，桌面应用的乱码情况比较少。这主要是因为接收数据输入的一般是 Java 平台提供的用户界面组件。对于 Web 应用来说，出现乱码问题的情况就比较多。这主要是因为不同平台上的不同浏览器在发送字符串数据给服务器的时候，所采用的编码格式可能各不相同，不同的 Web 应用开发平台所提供的编码支持能力也不尽相同，这使得问题变得更加复杂。

下面着重对 Web 应用中的乱码问题进行比较深入的讨论。通常来说，用户的输入会出现在 Web 应用的两个地方：一个是作为 Web 应用的 URL 的一部分，比如对一个搜索引擎来说，用户提交的搜索关键词通常是直接出现在进行搜索的 URL 中的；另一个是出现在 HTML 表单提交的数据中，比如在一个新用户注册页面中，用户输入的相关信息会被以表单提交的方式传输到 Web 应用中。从 HTTP 请求的角度来说，前者对应的是 GET 请求，而后者对应的是 POST 请求。用户提供的这些信息都是以字符的形式输入的，通过 HTTP 协议传输之后被 Web 应用的后台接收到。

首先讨论 HTTP GET 请求。在 GET 请求中，相关的信息都是作为 URL 的一部分而出现的。在目前的 Web 应用开发实践中，使用有意义的 URL 被认为是一个很好的实践。比如对一个博客网站来说，一个常见的做法是把每篇文章的标题作为该文章的 URL 的一部分。这种做法带来的最直接的好处是对搜索引擎更加友好，在搜索结果中的排名

比较靠前。因为搜索引擎通常会对 URL 中出现的内容赋予比较高的权重。对于只包含 ASCII 字符的内容来说，作为 URL 的一部分并不是一件困难的事情。不过对于 ASCII 之外的字符，如中文字符来说，则只有经过正确的编码之后才能出现在 URL 之中。

在与 URL 相关的规范 RFC 3986[⊖]中对 URL 中允许出现的字符做了详细的规定。允许出现在 URL 中的字符分成两大类：保留字符和非保留字符。保留字符是指在某些情况下有特殊含义的字符，包括“!”、“*”、“!”、“(”、“)”、“;”、“:”、“@”、“&”、“=”、“+”、“\$”、“,”、“/”、“?”、“#”、“[”和“]”。这些保留字符的含义各不相同，比如“/”用来分隔 URL 中的路径，“?”表示 URL 中查询字符串的开始，“&”用来分隔查询字符串中的不同参数等。非保留字符则是不具备特殊含义的字符，包括大小写英文字母、0 到 9 的数字、“-”、“_”、“.” 和 “~”。对于保留字符来说，如果只希望表示这个字符本身，而忽略其特殊含义，需要对这个字符进行编码。URL 中使用的编码格式是“百分号编码格式”，也就是将保留字符对应的 ASCII 编码值的二进制形式转换成相应的十六进制方式之后，再加上“%”作为前缀。例如，当希望在查询字符串中把“?”作为某个参数的值的一部分的时候，就需要对它进行百分号编码，因为这个时候没有用到“?”的特殊含义。“?”的 ASCII 编码是 63，对应的十六进制形式是 3F，因此百分号编码的结果是“%3F”。除了非保留字符之外的其他字符，要出现在 URL 中，都需要经过百分号编码。一个特例是空格，由于空格出现得比较频繁，除了可以使用其百分号编码格式“%20”之外，也可以用“+”字符来代替。这样的好处是可以减少 URL 的长度。

对于 ASCII 字符集之外的其他字符来说，RFC 3986 规范的要求是这些字符应该先通过 UTF-8 编码格式得到其对应的字节序列，再对这个字节序列中的每个字节都使用百分号编码格式。比如中文字符“你好”，经过 UTF-8 编码之后的字节序列是“0xE4BDA0E5A5BD”。如果出现在 URL 中，则应该使用“%E4%BD%A0%E5%A5%BD”的形式。使用 UTF-8 是一种常见的做法，却不是唯一的做法。不同的 Web 应用有可能采用不同的编码格式，这取决于 Web 应用本身，因为这些 URL 最终是由应用本身来处理的。Java 提供了 `java.net.URLEncoder` 类，根据不同的编码方式对字符串进行百分号编码，以在构造 URL 时使用。以百度搜索引擎为例，在百度的搜索 URL 中对输入的关键词用 GB2312 进行编码，所以当需要构造 URL 时，可以使用代码清单 4-6 中的代码实现。

代码清单 4-6 URL 编码方式

```
String url = "http://www.baidu.com/s?wd=" + URLEncoder.encode(keyword, "GB18030");
```

如果采用了错误的编码方式，百度搜索引擎的服务器在按照 GB2312 进行解码的时候，就会变成无法识别的字符。不过这种错误情况一般只发生在直接在外部程序中构造 URL 的时候，如一些网页抓取程序。一般用户都是直接单击页面内的超链接来访问新页

[⊖] 规范的具体内容见：<http://www.ietf.org/rfc/rfc3986.txt>。

面的。这些 URL 都是 Web 应用自己生成的，不存在这个问题。

在服务器端处理 GET 请求的时候，一般底层应用服务器会提供相关的 URL 解码功能，只需要对应用服务器进行配置即可。比如 Tomcat 是通过 URIEncoding 来进行配置的。如果希望自己来处理 URL 的解码，可以使用 `java.net.URLDecoder` 类的 `decode` 方法。

在通过页面中的 HTML 表单来提交数据的时候容易产生问题。从浏览器提交来的字符可能会经过多个不同层次的编码转换，每个环节都可能产生错误。表单提交的数据一般是用“`application/x-www-form-urlencoded`”作为其内容类型。正如类型名称中的“`urlencoded`”所表示的含义一样，表单中的内容也是以类似 GET 请求中的 URL 的编码方式来进行编码的。对于其中包含的字符，也使用百分号编码的方式。而在编码的时候使用的字符集应该在 POST 请求的内容类型中显式地声明。比如在用 GB18030 进行编码的时候，就应该使用“`application/x-www-form-urlencoded; charset=gb18030`”作为 POST 请求的 HTTP 头“Content-Type”的值。

浏览器在对表单内容进行编码的时候使用的字符集由当前页面的字符集来确定。当前页面的字符集的确定取决于下面几个因素：返回页面的 HTTP 响应中的“Content-Type”头中给出的字符集，如“`text/html; charset=utf-8`”；页面中通过 `<meta>` 标签声明的字符集；用户通过浏览器的界面手动选择的字符集。如果确定的字符集是 UTF-8，那么在表单提交的时候，非 ASCII 字符会在以 UTF-8 编码之后转换成百分数编码形式发送给服务器。通过 HTML 中 `<form>` 元素的“`accept-charset`”属性可以设置与页面编码不同的专供表单提交使用的编码格式。不过需要注意浏览器兼容性问题，不同的浏览器对于这个属性的支持是不同的。但是浏览器在完成编码之后，一般不会把所用的字符集声明出来。大部分浏览器都只是用“`application/x-www-form-urlencoded`”作为 HTTP 头“Content-Type”的值，而不会显式地使用“`application/x-www-form-urlencoded; charset=utf8`”这样的格式。这也是造成乱码问题的一个很重要的原因。

将编码之后的数据发送到服务器之后，服务器端的 Java 程序一般不直接处理 HTTP 请求，而是依靠底层的框架来完成。一般的 Java Web 应用都基于 servlet 规范进行开发。HTTP 请求的入口一般是某个 servlet 实现类中的 `doGet`、`doPost` 或 `doPut` 方法。在这几个方法中通过以参数方式传入的 `HttpServletRequest` 类的对象就可以获取到浏览器端发送过来的数据。对于 GET 和内容类型为“`application/x-www-form-urlencoded`”的 POST 请求，都可以通过 `HttpServletRequest` 类的对象的 `getParameter` 方法来得到参数的值。对于 POST 或 PUT 请求，还可以通过 `getInputStream` 来得到用来读取请求中的内容的 `java.io.InputStream` 类的对象。如果读取到的是 `InputStream` 类的对象，处理起来相对容易一些，因为是字节流。而通过 `getParameter` 方法得到的是 `String` 类的对象，这其中就涉及编码的问题。如果没有在“Content-Type”头中显式指定，根据相关的规范，ISO 8859-1 是默认的编码格式。也就是说，虽然包含 HTML 表单的网页使用了 UTF-8 作为编码格式，浏览器也按照 UTF-8 的格式提交了数据，如果服务器或代码中没有经过正确的设置，底层实现将使用默认的 ISO 8859-1 进行处理，这样就会产生乱码问题。

对于新开发的 Web 应用来说，推荐在整个应用的各个层次上都使用 UTF-8 作为统一的编码格式。即便不使用 UTF-8，编码格式也应该统一为一种。所有的 HTML 页面都通过 <meta> 标签来声明使用 UTF-8 作为编码格式。也需要将文件本身的编码设置为 UTF-8。同时对服务器进行配置以发送正确的 HTTP 响应头信息。JSP 页面则通过 “<%@page pageEncoding="UTF-8"%>” 来声明页面的编码格式。而在服务器端，通过 HttpServletRequest 类的 setCharacterEncoding 方法来显式地设置解析时使用的编码格式。常用的做法是通过一个过滤器 javax.servlet.Filter 接口的实现来对所有请求的 HttpServletRequest 类的对象进行设置。代码清单 4-7 给出了一个简单的实现，基本的思路是，如果请求中没有指定编码格式，就设置为默认的 UTF-8 格式。

代码清单 4-7 设置编码格式的过滤器

```
public void doFilter(ServletRequest request, ServletResponse response,
                     FilterChain chain)
    throws IOException, ServletException {
    if (request.getCharacterEncoding() == null) {
        request.setCharacterEncoding("UTF-8");
    }
    chain.doFilter(request, response);
}
```

在某些情况下，并不能要求对所有的页面都使用同样的 UTF-8 编码格式。这种情况在处理遗留系统的时候比较常见，比如需要把一个遗留系统的 Web 前端对接到新的服务器端。这就会造成新旧两种 Web 前端使用不同编码格式的情况，而同一个后台要能够对它们的请求做出正确处理。为了能够在这两种情况下都正确地进行编码，需要在请求中显式地指明所用的编码格式。由于浏览器不会直接在表单提交中添加编码格式的标识，最直接的做法就是自己加上一个额外的查询参数，如 “encoding=GB18030”。在过滤器中就可以通过检查这个参数的值来设置相应的编码格式。不过要注意的是，不能直接通过 HttpServletRequest 类的 getParameter 方法来获取这个参数的值，因为 getParameter 方法在执行的过程中就已经对请求中的内容进行了解码处理，之后再通过 HttpServletRequest 类的 setCharacterEncoding 方法进行设置就没有意义了，得到的仍然是乱码。一种可行的解决办法就是由程序自己来解析 URL 中的查询字符串，从中得到编码格式的参数值，从而可以解决这个问题。如果请求不是由浏览器本身来发送，而是通过浏览器中的 XMLHttpRequest 对象或是 Java 程序来发出的，就具备了直接修改 HTTP 头的能力。这个时候，就可以使用正确的“Content-Type”头来指明编码格式了。

4.4 区域设置

如果程序需要提供国际化的支持，就要求程序对用户的区域设置信息是敏感的。也就是说，程序在输出相关信息给用户的时候，应该考虑当前用户所在的地理位置区域。

比如当用户所在地区为中国大陆的时候，程序应该采用简体中文作为信息的显示语言，并在日期和时间、数字和货币等显示的格式上也符合中国大陆的用户的使用习惯。通常来说，操作系统会提供相关的功能来允许用户设置其所在的区域。Java 虚拟机也会以底层操作系统的区域设置作为其默认的设置。Java 中与区域设置相关的 API 都可以根据 Java 虚拟机默认的或是程序指定的区域设置来产生对应的输出。通过操作系统或其他方式识别出来的用户的区域设置信息不一定是准确的，程序一般都应该提供允许用户自由切换区域设置的功能。

Java 中的区域设置是通过 `java.util.Locale` 类的对象来表示的。由于区域设置本身的复杂性，`Locale` 类的对象中所包含的信息是比较多多的。这些信息的目的在于准确地区分不同的区域，同时又以便于使用者理解的方式组织起来。在 Java 7 之前，`Locale` 只包含了语言名称、国家或地区名称以及变体信息。从 Java 7 开始，Java 中的区域设置支持了 IETF BCP 47（Tags for Identifying Languages）[⊖]这一最佳实践，相关的 API 也做了修订和增强。

4.4.1 IETF BCP 47

用户的区域设置可以用一个标识符来表示。程序通过这个标识符来确定应该采用什么样的方式来把信息呈现给用户。一般来说，一个区域设置标识符至少应该包括语言标识符和国家或地区的标识符两部分，还可能包含一个变体信息。比如英文的语言标识符是“en”，而美国的标识符是“US”。所以对于生活在美国的英语使用者来说，他们的区域设置就同时包含“en”和“US”。这也是 Java 中的 `Locale` 类的对象最开始的时候包含这 3 种信息的原因。不同的操作系统平台实现可能采用不同的标识符格式，但是这些基本信息都是存在的，只是表示的方式不同。为了增强互操作性，IETF BCP 47 定义了一个语言标签的格式规范，这个标签可以作为区域设置的标识符的格式。从 Java 7 开始，Java 平台开始使用 BCP 47 的格式作为其区域设置标识符的基本格式，之前的格式也予以保留以保证与早期版本的兼容性。新的 Java 程序应该使用 BCP 47 的格式来表示其区域设置，Java 也提供了相关的 API 可以很方便地创建出这种格式的 `Locale` 类的对象。

IETF BCP 47 中定义的语言标签只是一个简单的字符串，可以附加在一段信息上，用来声明这段信息所使用的语言。这个字符串由几个部分组成，每个部分之间用“-”分隔。每个部分被称为一个子标签，其格式和含义是不相同的，可以很容易地进行解析。即便某个部分中包含的内容本身可能是无意义的，也可以正确地进行解析。比如包含国家或地区标识符的那部分内容可能并不对应于任何实际存在的国家或地区，但是仍然可以被正确地分析出来。这么设计的好处在于可以很好地应对以后的变化。

第一部分是语言的名称，如“zh”、“en”和“fr”等。这个部分可以是 2 到 8 个字母。最常使用的是 2 或 3 个字母的 ISO 639 规范定义的语言名称。在主要语言名称的后

[⊖] IETF BCP47 规范的网址是：<http://tools.ietf.org/html/bcp47>。

面可以加上扩展的子语言名称。这通常是因为某些语言存在一些使用广泛的子语言。如中文中的粤语就可以用“zh-yue”来表示。第二部分是语言的书写格式。这个部分是在 ISO 15924 中定义的 4 个字符长度的符号。第三部分是国家或地区的名称，这个部分可以是 ISO 3166-1 定义的 2 个字母的编号或是 UN M.49 定义的 3 个数字的编号。第四部分是变体信息，用来描述语言或其方言的变体。以字母开头的变体信息的长度至少是 5，而以数字开头的变体信息的长度至少是 4。第五部分是语言的扩展。这些扩展用来描述语言的一些附加信息。每个扩展由两个部分组成：第一部分是单个字母的键，第二部分则是 2 到 8 个字母或数字组成的值。最后一部分是私有标记。这个部分以字母“x”开始，后面跟着 1 到 8 个字母或数字。

从上面对语言标签的描述中可以看出，语言标签以及基于它的 Locale 类的对象更多的时候只是一个标识符，它本身并不包含具体的本地化的能力。支持本地化的方法应该接受一个 Locale 类的对象作为参数，根据这个 Locale 类的对象所表示的区域来正确地产生相应的输出。

在 Java 7 之前，只有两种创建 Locale 类的对象的方式，一种是使用它的构造方法，另外一种是使用 Locale 类中预先定义的常用区域设置。Java 7 新增了两种与 BCP 47 相关的构造方式。一种是用静态方法 forLanguageTag 来把一个符合 BCP 47 的语言标签字符串转换成 Locale 类的对象。另外一种则是使用新的 Locale.Builder 类来设置语言标签的各个部分，并最终产生一个完整的 Locale 类的对象。代码清单 4-8 给出了一个 Locale.Builder 类的使用示例，注意其中方法级联的使用。

代码清单 4-8 Locale.Builder 类的使用示例

```
public void useLocaleBuilder() {
    Locale locale = new Locale.Builder().setLanguage("zh").setRegion("CN").
        setExtension('m', "myext").build();
    String tag = locale.toLanguageTag(); // 值为 "zh-CN-m-myext"
}
```

4.4.2 资源包

资源包（resource bundle）应该是开发人员最熟悉的对 Java 程序进行国际化的方式。通常的做法是把程序中要显示给用户的消息文本都抽取出来，保存在属性文件中。这样的属性文件通常有一组，包含的内容是相似的，只是每个属性文件分别对应于一个特定的区域设置。资源包在使用方式上类似于一个 java.util.Map 接口，即其中所包含的是键值对的列表。从属性文件中创建出来的资源包自动地包含文件中声明的键值对作为其内容。使用者只需要根据当前的区域设置获取到对应的资源包对象，再从其中根据当前要显示的消息的键来获取消息的内容并显示出来即可。

Java 中的 java.util.ResourceBundle 类用来表示一个资源包。每个资源包都有两个重

要的属性：一个是它的基本名称，用来区分不同的资源包；另外一个是资源包对应的 `Locale` 类的对象。基本名称相同的所有资源包所包含的键是相同的，区别在于键的值是否经过本地化处理，以对应于不同的 `Locale` 类的对象所表示的区域设置。前面提到的基于属性文件的资源包只是资源包的一种形式，由 `java.util.PropertyResourceBundle` 类来表示，其中只能包含字符串作为键的值；另外一种形式是直接继承自 `ResourceBundle` 类的 Java 类，其中可以包含任意 Java 对象作为键的值。这两种形式其实是统一的，对于属性文件，在获取的时候会自动从文件中创建 `PropertyResourceBundle` 类的对象。在程序中使用资源包的时候，总是使用 `ResourceBundle` 类及其子类的对象。

如果希望创建自己的 `ResourceBundle` 类的实现，可以直接继承 `ResourceBundle` 类，或使用 `java.util.ListResourceBundle` 类。在直接继承 `ResourceBundle` 类时，只需要实现用来遍历其中包含的所有键的 `getKeys` 方法，以及根据键来获取对应值的 `handleGetObject` 方法即可。`ListResourceBundle` 类则提供了一种基于二维数组的快速创建 `ResourceBundle` 类的对象的实现。这个二维数组的每一行表示一条记录，而对应的第一列和第二列分别是键和值。只需要继承此类，实现其中的 `getContents` 方法来返回一个包含了全部数据的二维数组即可。`ListResourceBundle` 类非常适合在内存中创建和维护资源包或是作为其他 `ResourceBundle` 类格式的包装容器。

在创建了程序中所需的属性文件或 `ResourceBundle` 类的子类之后，下一步是找到适合于当前用户的区域设置的 `ResourceBundle` 类的对象。这是通过 `ResourceBundle` 的 `getBundle` 方法的各种不同的重载方式来实现的。这些不同的实现方式分成两类：第一类是 Java 6 之前的基于资源包的基本名称、`Locale` 类的对象和类加载器对象的查找方式；第二类是 Java 6 中新增的通过 `ResourceBundle.Control` 类的对象来控制查找过程的查找方式。

第一种查找方式的关键在于根据资源包的基本名称和 `Locale` 类的对象来生成一个资源包对应的 Java 类和属性文件的名称查找序列。这个名称查找序列会考虑到 `Locale` 类的对象中的语言、书写方式、国家或地区，以及变体等信息，其基本的思想是优先查找最具体的资源包，如果找不到，就尝试查找通用一些的资源包。如果在尝试了指定的 `Locale` 类的对象之后，仍然无法找到对应的资源包，这个过程会以当前 Java 虚拟机的默认的 `Locale` 类的对象为目标，再重复执行一次。如果仍然无法找到，就会抛出 `java.util.MissingResourceException` 异常。例如，假设基本名称是“Messages”，所用的 `Locale` 类的对象的语言和国家或地区分别是“en”和“US”，那么对应的名称查找序列是：`Messages_en_US`、`Messages_en` 和 `Messages`。对于这些候选名称，首先尝试通过指定的类加载器来查找并加载对应名称的 Java 类，如果这个过程成功并且该 Java 类可以转换成 `ResourceBundle` 类，就创建该类的一个实例，作为查找到的结果。如果查找 Java 类的过程失败，接着会尝试查找属性文件。由于基本名称中可能带有名称空间，对应的属性文件名称是把候选名称中的“.”替换成“/”之后的名称，再通过类加载器的 `getResource` 方法来进行查找。如果找到属性文件，会以此文件作为输入来创建一个

PropertyResourceBundle 类的对象作为结果。

ResourceBundle 类的对象本身也是存在一定层次结构的。一个 ResourceBundle 类的对象有可能存在一个父 ResourceBundle 类的对象。子 ResourceBundle 类的对象中会包含父 ResourceBundle 类的对象中定义的键值对。这种层次关系只有在资源包的查找过程中才会建立。根据上面提到的查找时的候选名称序列，出现在后面的查找到的 ResourceBundle 类的对象是之前的 ResourceBundle 类的对象的父亲。在查找过程中，会对所有能够成功创建的 ResourceBundle 类的对象建立这种层次结构关系。代码清单 4-9 给出了资源包的查找过程和基本的使用方式。对于同一基本名称的资源包，代码中分别使用 Java 类和属性文件来提供对应不同区域设置的本地化内容。查找到 ResourceBundle 类的对象之后，通过其 getString 方法来获取本地化的内容。

代码清单 4-9 资源包的查找过程和基本的使用方式

```
//Java 类定义的 ResourceBundle
public class Messages_en_US extends ListResourceBundle {
    public Object[][] getContents() {
        return new Object[][]{
            {"GREETING", "Hello!"},
            {"THANK_YOU", "Thank you!"}
        };
    }
}

// 属性文件
GREETING = 你好!
THANK_YOU = 谢谢!

// 使用 ResourceBundle 的代码
public void useResourceBundle() {
    String baseName = "com.java7book.chapter4.resourcebundle.Messages";
    ResourceBundle bundleEn = ResourceBundle.getBundle(baseName, Locale.US);
    bundleEn.getString("GREETING"); // 值为“Hello!”
    ResourceBundle bundleCn = ResourceBundle.getBundle(baseName, Locale.CHINA);
    bundleCn.getString("THANK_YOU"); // 值为“谢谢!”
}
```

在第一种查找过程中，开发人员所能控制的地方很少。为了满足开发人员的需求，Java 6 中引入了 ResourceBundle.Control 类来允许开发人员对 ResourceBundle 的查找过程进行复杂的定制，同时也对查找过程进行了增强。开发人员可以通过继承 ResourceBundle.Control 类的方式来定制自己所需的行为。实际上，从 Java 6 之后，第一种查找方式的内部实现也用了 ResourceBundle.Control 类，只不过用的是该类的默认实现，以符合 Java 6 之前的查找过程。在 ResourceBundle.Control 类中可以定制的部分包括以下几个方面：

首先是 ResourceBundle 类的对象的类型。通过 ResourceBundle.Control 类的 getFormats 方法可以返回对于给定的基本名称应该要查找的 ResourceBundle 类的对象的格式名称。前面提到的 Java 类和属性文件等两种类型的名称分别是“java.class”和“java.properties”。开发人员可以选择只查找这两种基本格式中的一种，或者使用自定义的格式名称。

第二个可以定制的是在对指定的 Locale 类的对象进行查找时要搜索的 Locale 类的对象的列表。这个 Locale 类的对象的列表的作用等价于第一种查找方式中的候选名称列表。通过覆写 getCandidateLocales 方法来为每个基本名称和指定的 Locale 类的对象提供一组 Locale 类的对象作为候选名称。

第三个可以定制的是当通过给定的 Locale 类的对象找不到资源包时应该要尝试使用的 Locale 类的对象。第一种查找方式在这种情况下使用的是 Java 虚拟机的默认 Locale 类的对象。通过覆写 getFallbackLocale 方法可以改变这种行为，以合适的 Locale 类的对象来替代。

第四个可以定制的是 ResourceBundle 类的对象的缓存行为。在默认情况下，ResourceBundle 类的对象在查找完成并设置好父 ResourceBundle 类的对象之后会被缓存起来。当下次再查找的时候，会直接使用缓存中的对象。通过第一种查找方式无法对缓存进行控制，而 ResourceBundle.Control 类则提供了相关的缓存控制机制。可以在缓存中的 ResourceBundle 类的对象指定一个存活时间（time to live）。通过覆写 getTimeToLive 方法可指定这个时间。如果在调用 getBundle 方法的时候发现缓存中的 ResourceBundle 类的对象已经超过了其存活时间，会调用 ResourceBundle.Control 类的 needsReload 方法来判断是否需要重新创建。如果需要，getBundle 方法会重新查找并创建新的 ResourceBundle 类的对象；否则还是会使用缓存中的对象并更新其存活时间。这个缓存机制与 HTTP 协议中的缓存机制是类似的。如果 ResourceBundle 类的对象还有存活时间，是不会通过 needsReload 方法来进行检查的。使用 ResourceBundle 类的 clearCache 方法可以清空缓存的 ResourceBundle 类的对象。

最后是通过 newBundle 方法来实际创建 ResourceBundle 类的对象。默认的实现只是提供了对 Java 类和属性文件的处理。如果采用的是自己的资源包格式名称，需要在这个方法中添加相应的创建逻辑。

在使用了 ResourceBundle.Control 类的对象的情况下，资源包的查找过程与采用第一种查找方式相比，整体的流程是相似的。只不过在某些关键步骤上 getBundle 方法会调用 ResourceBundle.Control 类中的特定方法来确定下一步的处理方式。在最开始的时候，getBundle 方法会首先检查缓存。接着是通过 getFormats 方法来确定要查找的资源包的格式。然后通过 getCandidateLocales 方法来得到要搜索的 Locale 类的对象列表。列表中的 Locale 类的对象都会通过 newBundle 方法来尝试创建新的 ResourceBundle 类的对象。如果对上述 Locale 类的对象的尝试都失败，会通过 getFallbackLocale 方法来得到一个新的替代 Locale 类的对象，并再次尝试上面的查找过程。

下面通过一个具体的示例来说明 ResourceBundle.Control 类的用法和如何创建自己的资源包类型。前面提到过 Java 平台本身就支持 Java 类文件和属性文件两种资源包类型，而且 Java 类需要继承自 ResourceBundle 类。这里要创建的是一种基于任意 Java 类的资源包类型。这个 Java 类中公开的静态方法的名称作为资源包中的键，而方法调用的返回结果则作为键对应的值。使用第 2 章中介绍的反射 API 实现起来并不复杂。首先代码清单 4-10 中给出的 ReflectiveResourceBundle 类是自定义的资源包实现，其中的 getKeys 方法的实现是通过反射 API 得到当前类中所包含的所有不带参数的公开静态方法，而 handleGetObject 方法则根据键的值找到对应的方法，在执行方法调用之后返回结果。之所以使用静态方法，是因为在调用的时候不需要提供额外的接收者对象作为参数。

代码清单 4-10 自定义的资源包实现

```
public class ReflectiveResourceBundle extends ResourceBundle {
    private Class clazz;

    public ReflectiveResourceBundle(Class clazz) {
        this.clazz = clazz;
    }

    public Object handleGetObject(String key) {
        if (key == null) {
            throw new NullPointerException();
        }
        try {
            Method method = clazz.getMethod(key);
            if (method == null) {
                return null;
            }
            return method.invoke(null);
        } catch (Exception ex) {
            return null;
        }
    }

    public Enumeration<String> getKeys() {
        Vector<String> result = new Vector<String>();
        Method[] methods = clazz.getMethods();
        for (Method method : methods) {
            int mod = method.getModifiers();
            if (Modifier.isStatic(mod) && Modifier.isPublic(mod) && method.
                getParameterTypes().length == 0) {
                result.add(method.getName());
            }
        }
        return result.elements();
    }
}
```

接下来就是提供对应的 ResourceBundle.Control 类，如代码清单 4-11 所示。这里使用了“reflection”作为新资源包类型的名称。而在 newBundle 方法的实现中，只需要根据基本名称加载对应的 Java 类，再把该类对应的 Class 类的对象封装在 ReflectiveResourceBundle 类的对象中即可。

代码清单 4-11 ResourceBundle.Control 类的自定义子类

```
public class ReflectiveResourceBundleControl extends ResourceBundle.Control {
    public List<String> getFormats(String baseName) {
        if (baseName == null) {
            throw new NullPointerException();
        }
        return Arrays.asList("reflection");
    }

    public ResourceBundle newBundle(String baseName, Locale locale, String format,
        ClassLoader loader, boolean reload) throws IllegalAccessException,
        InstantiationException, IOException {
        if (baseName == null || locale == null
            || format == null || loader == null) {
            throw new NullPointerException();
        }
        ResourceBundle bundle = null;
        if (format.equals("reflection")) {
            String bundleName = toBundleName(baseName, locale);
            try {
                Class<?> clazz = loader.loadClass(bundleName);
                return new ReflectiveResourceBundle(clazz);
            } catch (ClassNotFoundException ex) {
                return bundle;
            }
        }
        return bundle;
    }
}
```

接着是以 Java 类的方式声明不同区域设置对应的资源包所包含的内容。每条记录对应一个类中的公开静态方法。代码清单 4-12 是一个简单的示例，其中只包含一个键“greet”，而该键的值是随机变化的。

代码清单 4-12 具体的资源包中所包含的内容

```
public class ReflectiveMessages_zh_CN {
    public static String greet() {
        return "你好，" + (Math.random() > 0.5 ? "先生" : "女士");
    }
}
```

最后是在实际中的应用，在代码清单 4-13 中可以看到，使用方式并没有很大的区别，只是在调用 `getBundle` 方法时多了一个 `ReflectiveResourceBundleControl` 类的对象而已。

代码清单 4-13 自定义资源包实现的使用

```
public void useReflectiveResourceBundle() {
    String baseName = "com.java7book.chapter4.resourcebundle.ReflectiveMessages";
    ReflectiveResourceBundleControl control = new ReflectiveResourceBundleControl();
    ResourceBundle bundle = ResourceBundle.getBundle(baseName, Locale.CHINA,
        control);
    Object value = bundle.getObject("greet");
}
```

得到了 `ResourceBundle` 类的对象之后，对它的使用主要是通过 `getString` 和 `getObject` 两个方法。对基于属性文件的 `ResourceBundle` 类的对象来说，只能通过 `getString` 方法来获取字符串格式的内容；而对于基于 Java 类的 `ResourceBundle` 类的对象来说，`getObject` 方法也是可以使用的。在根据键来查找的时候，会考虑当前 `ResourceBundle` 类的对象在层次结构上的所有父 `ResourceBundle` 类的对象。如果从 `ResourceBundle` 类的对象中得到的字符串中包含占位符，可以通过下面介绍的消息格式化来进行处理。

在日常的开发中，基于属性文件的 `ResourceBundle` 类的对象的使用是最多的。在 Java 6 之前，属性文件对应的 `PropertyResourceBundle` 类的对象只能从 `InputStream` 类的对象中创建，而且对应的属性文件只能采用 ISO 8859-1 的编码格式。如果使用其他编码，会导致其中的内容无法识别。因此，无法在属性文件中直接使用非 ISO 8859-1 字符集中的字符。JDK 自带的 `native2ascii` 工具可以把其他编码格式的属性文件转换成 ISO 8859-1 的格式。一般的做法是原始的属性文件本身使用 UTF-8 编码，由开发人员直接来修改。在构建过程中通过脚本的方式（如 Apache Ant）调用 `native2ascii` 工具完成编码的转换。从 Java 6 开始，`PropertyResourceBundle` 类也可以从 `java.io.Reader` 类的对象中创建，这就为其他编码格式的属性文件提供了便利。基于 Java 6 及其以后版本的程序利用从 `Reader` 类的对象创建的这种方式，可以简化构建的过程。代码清单 4-14 是一个使用了这种思想的 `ResourceBundle.Control` 类的子类。具体做法是，如果资源包的类型是“`java.properties`”，即表示属性文件，就从得到的 `InputStream` 类的对象中利用指定的编码格式创建一个 `Reader` 类的对象，再从该 `Reader` 类的对象中得到 `PropertyResourceBundle` 类的对象。

代码清单 4-14 使用 `Reader` 类读取属性文件的 `ResourceBundle.Control` 类的子类

```
public class BetterResourceControl extends ResourceBundle.Control {
    private String encoding = "UTF-8";
    public BetterResourceControl(String encoding) {
        if (encoding != null) {
```

```

        this.encoding = encoding;
    }
}

public ResourceBundle newBundle(String baseName, Locale locale, String format,
    ClassLoader loader, boolean reload) throws IllegalAccessException,
    InstantiationException, IOException {
    if ("java.properties".equals(format)) {
        String bundleName = toBundleName(baseName, locale);
        String resourceName = toResourceName(bundleName, "properties");
        InputStream stream = null;
        if (reload) {
            URL url = loader.getResource(resourceName);
            if (url != null) {
                URLConnection connection = url.openConnection();
                if (connection != null) {
                    connection.setUseCaches(false);
                    stream = connection.getInputStream();
                }
            }
        } else {
            stream = loader.getResourceAsStream(resourceName);
        }
        BufferedReader reader = new BufferedReader(new InputStreamReader(
            stream, encoding));
        return new PropertyResourceBundle(reader);
    }
    return super.newBundle(baseName, locale, format, loader, reload);
}
}

```

在程序中，可以把 ResourceBundle.Control 类的使用隐藏起来，提供一个封装好的工厂方法，如代码清单 4-15 所示。这样程序的其他部分只需要用 UTF-8 格式来编写属性文件，并用这个工厂方法来加载即可。

代码清单 4-15 使用工厂方法封装对 ResourceBundle.Control 类的使用

```

public class ResourceBundleLoader {
    public static ResourceBundle load(String baseName, Locale locale) {
        BetterResourceControl control = new BetterResourceControl(null);
        return ResourceBundle.getBundle(baseName, locale, control);
    }
}

```

通过这种方式，就省去了对 native2ascii 工具的使用，简化了构建过程。下面要介绍的是 Java 提供的对日期和时间、货币和数字以及消息的格式化和解析的支持。类 java.text.Format 及其子类用来完成相应的格式化和解析操作。格式化是通过 format 方法来实现，即把一个对象转换成字符串类型；而解析则通过 parseObject 方法来实现，即把一个

字符串转换成对象。除了标准库中提供的格式化支持之外，程序也可以通过继承 Format 类来开发针对特定对象的格式化实现。

4.4.3 日期和时间

除了用户界面中显示的消息文本之外，日期和时间也是需要根据区域设置进行处理的重要内容。不同国家和地区所惯用的日期和时间的格式都不尽相同。日期和时间中除了通用的阿拉伯数字之外，一般还包含对应语言中的相关字符。这些字符也需要进行本地化。Java 程序内部一般使用 `java.util.Date` 和 `java.util.Calendar` 类来处理日期和时间。当把这些对象的值输出到用户界面上的时候，需要考虑与区域设置相关的格式化操作。这一般是通过 `java.text.DateFormat` 类来完成的。`DateFormat` 类对日期和时间的处理能力有格式化和解析两种，用来在 `Date` 类的对象和字符串形式之间进行转换。转换的时候都会考虑区域设置。

`DateFormat` 类的对象是通过工厂方法来创建的，可以选择只处理日期、只处理时间以及两者都处理的 `DateFormat` 类的对象，分别通过 `DateFormat` 类的静态方法 `getDateInstance`、`getTimeInstance` 和 `getDateTimeInstance` 来创建。在创建的时候，可以指定所用的格式和对应的 `Locale` 类的对象。格式共有 `SHORT`、`MEDIUM`、`LONG` 和 `FULL` 四种，按照包含的信息量从少到多进行排列。

得到 `DateFormat` 类的对象之后最直接的做法是通过 `format` 方法把一个 `Date` 类的对象转换成字符串，以及通过 `parse` 方法把一个字符串转换成 `Date` 类的对象。在通过 `format` 方法进行格式化的时候，有时候需要单独提取出格式化之后的内容中的一部分，比如只希望得到格式化之后的日期中关于星期的部分，这时可以使用 `parse` 方法的另外一种重载形式，需要用到 `java.text.FieldPosition` 类。简单地说，日期和时间格式化之后的结果字符串由多个部分组成，通过 `FieldPosition` 类可以跟踪感兴趣的部分在最终生成的字符串中的起始和结束位置。代码清单 4-16 给出了一个示例，在创建 `FieldPosition` 类的对象的时候就指定了要记录的是日期中的星期信息，使用了 `DateFormat.DAY_OF_WEEK_FIELD` 来进行声明。在格式化完成之后，就可以从保存结果的 `StringBuffer` 类的对象中根据 `FieldPosition` 类的对象中保存的位置得到所需要的信息。在 `DateFormat` 类中，日期和时间格式中的不同类型的组成部分都有相关的常量定义。使用这些常量定义可以选择跟踪日期和时间中的其他组成部分，如使用 `DateFormat.YEAR_FIELD` 可以跟踪年份信息。

代码清单 4-16 跟踪格式化结果中不同部分字符串的位置的示例

```
public void trackFormatPosition() {
    DateFormat format = DateFormat.getDateInstance(DateFormat.FULL);
    Date date = new Date();
    StringBuffer result = new StringBuffer();
    FieldPosition dayField = new FieldPosition(DateFormat.DAY_OF_WEEK_FIELD);
    format.format(date, result, dayField);
    String day = result.substring(dayField.getBeginIndex(), dayField.
```

```

    getEndIndex()); // 值为星期几的本地化形式
}

```

同样的，在通过 `parse` 方法进行解析的时候，也可以传入一个 `java.text.ParsePosition` 类的对象来表示在字符串中的解析位置。在解析之前，`ParsePosition` 类的对象可以用来表示解析的起始位置；而在解析之后，该 `ParsePosition` 类的对象中可以得到解析成功之后的下一个位置。代码清单 4-17 给出了一个示例，如果要解析的字符串中包含日期和时间的部分不是从起始位置开始的，就需要通过 `ParsePosition` 类的对象来指定起始的位置，否则无法解析。在解析完成之后，可以通过当前 `ParsePosition` 类对象的 `getIndex` 方法来获取输入字符串中的当前位置。

代码清单 4-17 设置解析时的起始位置的示例

```

public void parseWithPosition() {
    DateFormat format = DateFormat.getDateInstance(DateFormat.FULL);
    Date date = new Date();
    String dateStr = format.format(date);
    String prefix = "== START ==";
    String toParse = prefix + dateStr + "== END ==";
    ParsePosition position = new ParsePosition(prefix.length());
    Date d = format.parse(toParse, position);
    int index = position.getIndex();
}

```

如果希望对日期和时间的格式化进行更加精细的定制，可以使用 `DateFormat` 类的子类 `SimpleDateFormat`。`SimpleDateFormat` 类允许使用一个自定义的模式来指定格式化的输出结果。在模式中，不同的字符表示日期和时间的不同组成部分。常见的字符有表示年份的“y”，表示月份的“M”，表示月份中日子的“d”，表示小时数的“H”，表示分钟数的“m”和表示秒数的“s”。例如，模式“yyyy-MM-dd”的输出结果可能是“2011-07-12”，“HH:mm:ss”的输出结果可能是“13:23:03”。

4.4.4 数字和货币

数字和货币的格式化也是国际化中的重要组成部分。不同国家和地区在数字中小数点的位置、是否使用数字间的分隔符，以及使用的数字符号等方面都有所不同。而在货币方面，货币的符号也有所不同。Java 也提供了与区域设置相关的数字和货币的格式化和解析功能，这个功能通过 `java.text.NumberFormat` 类来实现。

`NumberFormat` 类的使用与上一节介绍的 `DateFormat` 类的用法很类似，也要通过工厂方法来创建。可以创建的 `NumberFormat` 类的对象总共有 4 种：`getNumberInstance` 方法得到一个通用的数字格式化对象，`getIntegerInstance` 方法得到一个处理整数的格式化对象，`getPercentInstance` 方法得到一个处理百分数形式的数字的格式化对象，`getCurrencyInstance` 方法得到一个处理货币的格式化对象。每个 `NumberFormat` 类的对

象同样有 format 和 parse 方法来分别格式化和解析数字，FieldPosition 和 ParsePosition 类也同样可以使用。除了常规的选项之外，NumberFormat 类还支持对格式化和解析时的行为进行定制，包括定制格式化之后的数字中整数和小数的位数、是否对数字进行分组，以及对数字进行四舍五入的方式等。代码清单 4-18 给出了 NumberFormat 类的使用示例，其中设置了数字在格式化时的整数和小数部分的最少位数。

代码清单 4-18 格式化和解析数字的示例

```
public void formatAndParseNumber() throws ParseException {
    NumberFormat format = NumberFormat.getNumberInstance();
    double num = 100.5;
    format.setMinimumFractionDigits(3);
    format.setMinimumIntegerDigits(5);
    format.format(num); // 值为 00,100.500
    String numStr = "523.34";
    format.setParseIntegerOnly(true);
    format.parse(numStr); // 值为 523
}
```

NumberFormat 类本身对数字所做的格式化和解析功能比较有限。如果希望更多地定制数字的格式，可以使用 NumberFormat 类的子类 java.text.DecimalFormat。实际上，通过 NumberFormat 类的工厂方法所得到的通常都是 DecimalFormat 类的对象。可以进行强制类型转换之后再使用 DecimalFormat 类中的方法。除此之外，还可以直接创建 DecimalFormat 类的对象。DecimalFormat 类的好处之一在于可以用字符串的方式来声明格式化的模式。代码清单 4-19 给出了使用 DecimalFormat 类的示例，通过 applyPattern 方法可以直接设置格式化的模式。

代码清单 4-19 DecimalFormat 类的使用示例

```
public void useDecimalFormat() {
    NumberFormat format = NumberFormat.getNumberInstance();
    DecimalFormat df = null;
    if (format instanceof DecimalFormat) {
        df = (DecimalFormat) format;
    }
    else {
        df = new DecimalFormat();
    }
    df.applyPattern("000.###");
    df.format(3.14); // 输出为 003.14
}
```

另外一个可用的类是 java.text.ChoiceFormat，它可以用来实现根据数字值的大小来应用不同的格式化模式。一个典型的应用场景是某些语言中的单复数所用的形式不同，需要根据数字值的大小来使用不同的文本。例如，英语中的单词“apple”的复数

形式是“apples”，在使用时要使用正确的形式，如“one apple”和“three apples”。在创建 ChoiceFormat 类的对象时需要指定一个 double 类型的数组作为进行数值比较时的各个区间的端点值，以及与这些区间对应的格式化字符串。当格式化的数字的值落在某个区间上的时候，就应用这个区间对应的格式化模式。代码清单 4-20 给出了一个示例，其中的 ChoiceFormat 类的对象在创建的时候指定了 3 个数字值，实际上创建了 4 个区间。每个区间都是左闭右开的，也就是说，如果值等于这些数字，属于左边的数字值。ChoiceFormat 类的 nextDouble 和 previousDouble 方法分别用来得到大于给定参数的最小的和小于给定参数的最大的双精度浮点数。这两个方法的返回结果通常分别作为最右边和最左边的区间的端点值。如果数字值落在最左边或最右边的区间中，则分别使用第一个和最后一个格式化字符串。在代码清单 4-20 中，只有值为 1 的情况会对应第二个格式化字符串，任何大于 1 的值都会对应第三个格式化字符串。

代码清单 4-20 ChoiceFormat 类的使用示例

```
public void useChoiceFormat() {
    ChoiceFormat format = new ChoiceFormat(new double[] {0, 1, ChoiceFormat.
        nextDouble(1)},
        new String[] {"no people", "person", "people"});
    int count = 2;
    String msg = count + " " + format.format(count); // 值为 2 people
}
```

在创建 ChoiceFormat 类的对象时并没有利用 Locale 类的对象，而是直接使用了普通的字符串。实际上，ChoiceFormat 类本身并不处理与区域设置相关的内容，通常需要与资源包配合起来使用。利用从资源包中得到的字符串作为创建时的格式化模式来使用。

4.4.5 消息文本

这里要介绍的是对字符串进行格式化和解析的 java.text.MessageFormat 类。之所以到此才介绍，是因为 MessageFormat 类在本身的模式之中允许使用上面介绍的 DateFormat、NumberFormat 和 ChoiceFormat 类的对象作为其内部的子模式。当一个字符串中包含日期、时间和数字的时候，就可以利用 MessageFormat 类的这个特性。MessageFormat 类的重点在于其使用的模式，可以在构造方法中提供，也可以通过 applyPattern 方法来改变。如果使用某个模式的 MessageFormat 类的对象只打算使用一次，可以用 MessageFormat 类中的静态方法 format 来快速根据某个模式进行格式化，不需要单独创建新的对象。在每个模式中除了直接显示的字符串之外，一般都包含在运行时刻进行替换的实际参数的占位符。在进行消息国际化时的一个基本原则是避免在代码中进行一段消息的连接操作，要把这整段消息都放在资源包中，由 MessageFormat 类来处理。例如，要生成类似“你好，张三，今天是星期五”这样的消息内容，其中的姓名和日期是可变的。不推荐用字符串拼接的做法把消息中的几个部分连接起来，而推荐把

这一段消息都放在资源包中，写成“你好，{0}，今天是{1}”这样的带参数占位符的形式。这其中的重要原因是不同语言的行文顺序是不同的。如果按照字符串相加的做法，某些语言就无法翻译成通顺的句子。

根据上面的示例，在MessageFormat类的模式中，参数占位符是通过在“{}”中包含参数的序号的方式来表示的。这些序号对应的是调用format方法进行格式化时提供的实际参数的数组中的位置，以及调用parse方法所得到的返回值的数组中的位置。对于每个参数，还可以进一步声明格式化的具体类型和模式。格式化类型的可选值有number、date、time、choice，而对应的模式则既可以是标准模式，又可以是自定义模式。例如，对number类型来说，既可以使用NumberFormat类支持的标准模式integer、currency和percent，又可以使用自定义的模式；对于date和time类型来说，可以使用的标准模式是DateFormat类支持的short、medium、long和full；而对于choice类型来说，则只支持自定义的模式。代码清单4-21给出了MessageFormat类的使用示例，一共定义了3个数字类型的参数，前两个使用的是标准模式integer和currency，而第三个使用的是自定义的模式。

代码清单4-21 MessageFormat类的使用示例

```
public void formatWithNumber() {
    String pattern = "购买了{0,number,integer}件商品，单价为{1,number,currency}，合
        计：{2,number,\u00A4#,###.##}";
    MessageFormat format = new MessageFormat(pattern);
    int count = 3;
    double price = 1599.3;
    double total = price * count;
    format.format(new Object[] {count, price, total});
}
```

除了通过MessageFormat类的模式字符串来声明内部所使用的子模式之外，也可以通过MessageFormat类的方法来进行设置。如果程序中提供了自己的Format类的子类实现，则可以通过这种方式来进行设置。与设置相关的一共有4个方法，区别在于是设置一个还是多个Format类的对象，以及是根据Format类的对象在格式化模式中的出现顺序还是参数顺序。考虑格式化模式“共有{1}人参加会议，其中{0}来自本部门”，其中定义了两个参数，不过第二个参数出现在第一个的前面。如果希望按照在模式中实际参数值的顺序来设置两个参数对应的Format类的对象，可以使用setFormatsByArgumentIndex方法；如果只希望设置单个参数的Format类的对象，可以使用setFormatByArgumentIndex方法；如果希望按照在模式中的出现顺序来设置，可以相应地使用setFormats和setFormat方法。由于表达同样语义的字符串在经过翻译之后，其中参数的出现位置可能发生变化，因此推荐的做法是基于参数值的顺序进行设置。

把程序中的文本都抽取出来放到资源文件中，这对于程序本身的可维护性也是很有帮助的。有些文本可能会在程序中出现多次，如果每次都是直接在源代码中使用，当文

本发生变化的时候，需要在多个地方进行修改。如果文本存放在资源文件中，引用时使用的都是抽象的固定的键，对应的内容可以随意修改，而且只需要修改属性文件这个地方即可。

4.4.6 默认区域设置的类别

在程序的国际化实现中，一般都使用当前 Java 虚拟机默认的区域设置信息。这个默认的 Locale 类的对象对用户来说通常是最合适的选择。Java 7 对默认的 Locale 类的对象进行了更进一步地细分，划分成不同的类别。这些类别定义在 Locale.Category 这个枚举类型中，目前包括 DISPLAY 和 FORMAT 两种。类别为 DISPLAY 的 Locale 类的对象在显示用户界面时会作为默认的区域设置，而类别为 FORMAT 的 Locale 类的对象则在格式化日期和时间、数字和货币的时候被作为默认的区域设置。同时 Locale 类中的 getDefault 和 setDefault 方法也支持使用 Locale.Category 枚举类型中的值作为参数来获取和设置对应类别的默认 Locale 类的对象。

提供不同类别的默认 Locale 类的对象的动机在于为程序提供更多的灵活性。因为在某些情况下，显示用户界面需要的区域设置与格式化日期和时间等用的区域设置可能是不相同的。由于默认区域设置的类别在 Java 7 中才引入，对于习惯了 Java 7 之前的只有一种默认区域设置的开发人员来说，容易产生错误。

以笔者的 Windows XP 系统为例，它本身的系统语言是英语，但是区域设置是中国。当通过 getDefault 方法来获取默认区域设置的时候，会发现 DISPLAY 类别的区域设置是 en_US，而 FORMAT 类别的区域设置则是 zh_CN。如果在默认的区域设置下使用 MessageFormat 类，会发现一个有趣的现象，如下面的代码清单 4-22 所示，通过 MessageFormat 类的对象格式化之后的文本中同时包含了英文和中文，类似“Hello, 张三。Today is 11-8-6 下午 5:17.”。这是因为 ResourceBundle 类在查找资源包时找到的是 DISPLAY 类别的默认区域设置对应的属性文件，而在格式化日期的时候，使用的则是 FORMAT 类别的默认区域设置，两者是不同的。

代码清单 4-22 区域设置类别的使用示例

```
public void useLocaleCategory() {
    ResourceBundle bundle = ResourceBundle.getBundle("messages");
    String str = bundle.getString("GREETING");
    String msg = MessageFormat.format(str, new Object[] {"张三", new Date()});
}
```

如果希望统一不同类别的区域设置，可以在程序启动之初通过 setDefault 方法把两个类别的默认区域设置设置为相同的值。

4.4.7 字符串比较

另外一个开发人员了解得比较少的需要进行本地化处理的是字符串的比较操作。对

于英语来说，字符串的比较一般是按照字典顺序进行的。而在其他语言中，则有自己的字符串比较规则。当需要与区域设置相关的字符串进行比较操作时，应该使用 `java.text.Collator` 类来完成。

`Collator` 的类和前面提到的 `DateFormat` 和 `NumberFormat` 类比较类似，也是通过一个工厂方法 `getInstance` 来得到一个具体的对象。通过 `Collator` 类的 `compare` 方法可以比较大小，返回值的含义与 Java 中常见的基于 `java.util.Comparable` 接口的比较操作是相同的。另外也提供 `equals` 方法来判断两个字符串在当前的区域设置下是否相等。

对 `Collator` 类来说，在进行排序时需要考虑的一个因素是如何处理不同语言中的字符的各种变体。在有些语言中，某些字符虽然表现形式不同，但是在排序的语义上却是相同的。通过设置 `Collator` 类的对象的分解模式可以定制对变体的处理行为。这些不同的行为是由 `Collator` 类中的常量来定义的：`NO_DECOMPOSITION` 表示不做任何处理，`CANONICAL_DECOMPOSITION` 表示按照 Unicode 中定义的规范变体来进行处理，`FULL_DECOMPOSITION` 表示不仅考虑 Unicode 中的规范变体，也考虑其中的兼容性变体。规范变体在 Unicode 中指的是互为变体的字符。最常见的变体形式是某些语言中的同一发音的不同声调由不同的字符来表示。这些不同的声调在比较的时候是没意义的。如果使用的是 `NO_DECOMPOSITION` 模式，会认为不同声调的字符是不同的，这会造成错误的结果。对于包含声调的语言，采用 `CANONICAL_DECOMPOSITION` 或 `FULL_DECOMPOSITION` 才会得到正确的结果。通过 `setDecomposition` 方法可以改变 `Collator` 类的对象所使用的分解模式。

另外一个需要在比较时考虑的因素是差异的不同粒度。比如，从不同角度来说，可以认为大写字母“A”和小写字母“a”是不同的，也可以认为是相同的。`Collator` 类中定义了 4 种不同的差异粒度，按照从粗到细的顺序分别由常量 `PRIMARY`、`SECONDARY`、`TERTIARY` 和 `IDENTICAL` 表示。这 4 种粒度的具体含义与具体的区域设置相关的。当通过 `setStrength` 方法把粒度设置在某个值上的时候，比较时只会考虑该粒度及其之上的差异。代码清单 4-23 给出了粒度对比较结果的影响。

代码清单 4-23 字符串比较时的粒度选择对结果的影响

```
public void useCollator() {
    Collator collator = Collator.getInstance(Locale.US);
    collator.setStrength(Collator.PRIMARY);
    collator.compare("abc", "ABC"); // 值为 0, 认为是相等
    collator.setStrength(Collator.IDENTICAL);
    collator.compare("abc", "ABC"); // 值为 -1
}
```

4.5 国际化与本地化基本实践

为 Java 程序添加国际化的支持，只需要遵循一定的模式即可，更多的是执行按部就

班的过程。下面对这个过程进行具体介绍，并说明其中一些需要注意的地方。

第一步是提取出程序中需要本地化的内容。这些内容包括前面介绍的用户界面上的消息文本、日期和时间、数字和货币，以及字符串比较操作等。这一步可以在程序开发的早期开始进行，即在代码编写过程中就进行提取；也可以在程序开发的后期进行。比较推荐的做法是在程序开发的后期进行。这主要是因为在程序开发过程中，本地化的内容可能还在不断变化，过早进行相关内容的提取，可能造成工作量的浪费。

由于消息文本是最常见的需要本地化的内容，很多集成开发环境（IDE），如 Eclipse 和 NetBeans，都提供了相应的功能来实现快速提取。利用 IDE 的支持，可以提高提取工作的效率。在源代码中，某些字符串字面量并不需要进行本地化。应该把这些字符串字面量重构为 Java 类中的字符串常量，以区别于其他需要本地化的字符串字面量。对于除消息文本之外的其他需要本地化的内容，需要开发人员手动识别。

第二步是添加国际化的能力。这一步需要对程序的代码进行修改。对于消息文本，把直接使用字符串字面量的方式替换成从资源包中根据指定的键来读取对应的值。从源代码中提取出来的文本内容被保存到一个属性文件中。一般来说，一个程序使用一个属性文件即可。对于复杂的程序，其中的每个组件可以有属于自己的独立的属性文件。开发人员需要负责为每条文本指定一个有意义的键的名称。在源代码中有可能使用了字符串拼接的方式来生成一段长文本，对于这种情况，需要把多个字符串字面量提取成资源包中的单条记录。代码清单 4-24 给出了一个未经本地化处理的 Java 类。

代码清单 4-24 未经本地化处理的 Java 类

```
public class NormalGreetings {
    public String greet(String name) {
        return "Hello, " + name + ". Today is " + new Date().toString() + ".";
    }
}
```

由于代码清单 4-24 中实际上只包含一条消息，在进行提取时，需要对多个字符串字面量进行合并。在属性文件中只包含一条记录，即“GREETINGS = Hello, {0}. Today is {1}.”。下一步需要创建相关的辅助 Java 类来负责从资源包中读取消息文本。代码清单 4-25 中给出了辅助 Java 类的示例。在 Messages 类的静态代码块中，调用 ResourceBundle 类的 getBundle 方法来加载指定基本名称的资源包。如果找不到，则用一个空的 ListResourceBundle 类的对象来代替。Messages 类的 get 方法用来根据消息文本的键和进行格式化的实际参数值来得到最终显示的文本。如果在资源包中找不到给定的键，则返回一个特殊形式的文本内容来进行声明。

代码清单 4-25 从资源包中读取消息文本的辅助 Java 类

```
public class Messages {
    private static ResourceBundle bundle;
    static {
```

```

try {
    bundle = ResourceBundle.getBundle("com.java7book.chapter4.demo.
        Messages", LocaleHolder.get());
} catch (MissingResourceException e) {
    e.printStackTrace();
    bundle = new ListResourceBundle() {
        protected Object[][] getContents() {
            return new Object[0][0];
        }
    };
}
}

public static String get(String key, Object... args) {
    try {
        String value = bundle.getString(key);
        return MessageFormat.format(value, args);
    }
    catch (MissingResourceException e) {
        return "!!!!" + key;
    }
}
}

```

在调用 ResourceBundle 类的 getBundle 方法时，总是应该显式地指定一个 Locale 类的对象，而不是依靠 Java 平台的默认区域设置。这是因为默认区域设置可能随着底层操作系统的变化而发生改变，可能会对程序的行为产生影响。一个比较好的做法是把应该使用的 Locale 类的对象保存在 ThreadLocal 类的对象中，与当前的运行线程绑定在一起。这种方式尤其适合于 Web 应用。

对于除消息文本之外的其他需要本地化的内容，在代码中使用对应的 Format 类的子类对象来进行格式化。具体的使用方式参考前面对 DateFormat 和 NumberFormat 类的介绍。

在选择程序使用的区域设置时，通常依次考虑三种情况。第一种情况是用户显式指定的区域设置。程序应该提供相关的功能供用户手动设置使用的区域设置。如果用户进行了选择，那么应该使用用户所选择的区域设置。第二种情况是自动检测区域设置。这种情况通常发生在 Web 应用中。在 HTTP 请求中，可以通过“Accept-Language”头来指定所要求使用的区域设置。Web 服务器需要处理“Accept-Language”头，并返回正确的响应。第三种情况是使用虚拟机的默认设置。使用 Locale 类的 getDefault 方法可以获取默认设置。

第三步是进行内容的本地化。这主要是针对消息文本的。这一步通常需要由专门的翻译人员对包含消息文本的属性文件的内容进行翻译。翻译之后的属性文件的名称需要进行修改，以表明所对应的区域设置，如“Messages_en_US.properties”文件名表示的

是针对区域设置“en_US”的属性文件。

最后一步是进行相关的测试。首先把程序的区域设置修改成待测试的值，再查看用户界面上的内容。检查用户界面上的消息文本、日期和时间，以及数字和货币的显示方式是否正确。对于 Web 应用来说，通过修改浏览器的设置可以改变所发送的“Accept-Language”HTTP 头的值。通过这种方式来测试 Web 页面的展示结果是否正确。如果程序在选择区域设置时使用了虚拟机的默认区域设置，可以通过虚拟机的启动参数“user.language”、“user.country”、“user.script”和“user.variant”来修改默认的区域设置，比如，使用“-Duser.country=US -Duser.language=en”可以把默认的区域设置修改为“en_US”。

4.6 小结

从实际开发的角度来说，Java 程序的国际化并不是一件很麻烦的事情。大多数时候要做的只是利用集成开发环境（IDE）提供的向导功能，把源代码中的硬编码字符串提取出来，放到某个属性文件中，再由专门的人员进行翻译即可。IDE 的功能已经完善到让开发人员不用理会过多的底层细节。而除了消息文本之外，对日期和时间以及数字和货币等类型来说，也多半只需要利用提供的工厂方法进行格式化即可。更多的时候要认识到国际化是软件开发中的重要一环，而不是忽略它。

本章虽然主要介绍了 Java 7 中提供的与区域设置以及格式化相关的标准 API，但是本章开始时介绍的字符的编码和解码等相关的背景知识也很重要。了解 Unicode 字符集和 UTF-8 等编码格式可以帮助开发人员在遇到一些棘手问题时知道从何处着手进行解决。

第 5 章 图形用户界面

与 Java 在服务器端，尤其是 Web 应用开发领域的成功相比，Java 在桌面应用开发中的地位一直比较尴尬，没有发展起来。很少有面向普通用户的桌面应用是基于 Java 平台开发的。Java 在桌面应用上比较成功的案例是集成开发环境（IDE）的实现，包括 Eclipse 和 NetBeans 等。Java 在桌面应用开发中没能流行的原因有很多，其中一个重要的原因在于 Java 桌面应用的运行离不开 Java 运行环境（Java Runtime Environment, JRE）的支持，JRE 在应用本身之外，需要安装在用户的操作系统之上，这就增加了用户安装和使用的难度。有些 Java 桌面应用选择把 JRE 打包在一起，不过这样又会造成安装包过大的问题。还有一个原因是性能方面的影响，由于存在 Java 虚拟机这样一个中间层，Java 桌面应用的性能通常要比原生代码编写的应用差一些，这会对普通用户的使用体验带来不好的影响。

当然，Java 桌面应用也并非一无是处。Java 平台通过 Java 虚拟机实现的“编写一次，到处运行”的特性在很多场合都是很有价值的。只需要维护一份代码，就可以在不同的操作系统平台上运行功能相同的桌面应用。进行 Java 开发也可以充分利用 Java 平台上丰富的社区资源，包括文档、示例和开源类库等。局域网内部使用的桌面应用很适合于用 Java 来开发，比如公司内部的管理系统等。桌面应用可以提供比 Web 应用更强的交互能力。而在内部网中，部署相关的问题也比较好解决。如果既要开发跨平台的桌面应用程序，又能比较好的解决部署的问题，那么 Java 平台是一个不错的选择。

Java 的用户界面组件库在使用上比较简单，只需要熟悉相关组件的 API 即可。限于篇幅，本章内容不会深入到桌面应用开发的具体细节，不会介绍具体的用户界面组件的用法，只是侧重在 Java 用户界面库中比较复杂和难以理解的部分，以及 Java 7 中增加的与图形用户界面相关的新特性。最后着重介绍了 Java 平台的下一代桌面应用开发技术 JavaFX。

5.1 Java 图形用户界面概述

Java 平台的图形用户界面库与其他编程语言平台所提供的图形用户界面库相比，总体上来说大同小异。图形用户界面库通常包含 3 个要素：组件、布局和事件。图形用户界面库可以看成是一些用户界面组件的集合。不同的组件提供不同的交互能力。开发人员通过某种布局方式把这些组件排列起来以构建用户界面，与用户进行交互。交互的基本模式是典型的事件驱动方式。用户的各种动作触发相应的事件，而事件的处理器则实现对应的业务逻辑。Java 平台上的图形用户界面库也由这 3 个要素组成。不同图形用户

界面库的区别主要在所使用的编程语言、组件的丰富性和布局的灵活性上。在 Java 平台上，实际上存在几种可选的图形用户界面库。

□ AWT

在 Java 语言刚诞生的时候，它提供的图形用户界面库叫做抽象窗口工具箱（Abstract Window Toolkit, AWT）。AWT 在底层操作系统提供的原生图形用户界面的基础上，提供了一个新的抽象。这样实现的目的是为了让用户界面也能完全跨越不同的平台，为开发人员屏蔽底层实现的细节。这个抽象层所做的只是根据当前的操作系统平台创建组件对应的原生控件，再把组件上的方法调用直接代理给原生控件。对于每一个 AWT 组件，在后台都有一个原生控件与其对应。这个原生控件被称为 AWT 组件的对等体（peer）。这种对应关系使 AWT 组件的运行时开销比较大；另外在资源管理方面也比较麻烦，要处理原生控件的资源释放问题。AWT 组件的外观渲染实际上是由其对等体来完成的，因此同一个 AWT 组件在不同操作系统平台上的外观是不同的，这取决于当前操作系统的外观样式。同样，同一个基于 AWT 的桌面应用在不同的操作系统平台上的外观是不同的，符合当前操作系统的外观样式。

AWT 作为 Java 平台上最早的图形用户界面库，它所包含的内容是比较完备的。首先是一组常用的用户界面组件，包括按钮、文本框、菜单和窗口等。这些构成了应用的用户界面的基础。其次是事件处理系统，用来处理系统中发生的事件及响应用户的动作。最后是一些辅助功能，包括布局管理、鼠标和键盘支持、剪贴板访问、拖放的支持、访问系统托盘等。有了这些功能，就可以开发出跨平台的桌面应用。总的来说，AWT 以一种相对简单办法实现了跨平台的用户界面。

□ Swing

AWT 虽然为 Java 的桌面应用开发创造了一个良好的开端，但是它所提供的功能还是比较有限的，难以应付一些复杂的应用需求，也缺少一些常用的复杂组件。另外 AWT 开发的应用只能采用与底层操作系统相似的外观风格，无法方便地进行外观看制。JDK 1.2 中引入的 Swing 用户界面库从不同方面解决了 AWT 中的问题。

与 AWT 直接利用底层操作系统的原生控件的做法不同，Swing 的用户界面组件是由 Java 平台自己绘制出来的，也就是说，用户看到的组件界面其实是通过 Java 提供的二维图形绘制功能（Java 2D）画出来的。不过 Swing 用户界面仍然是基于 AWT 的。一个完整的 Swing 用户界面是在一个空白的 AWT 组件上绘制的。从这点来看，Swing 所消耗的系统资源要低于 AWT，因为一个 Swing 用户界面只需要使用一个操作系统提供的原生控件。一般把 Swing 的用户界面组件称为轻量级组件，而将 AWT 的用户界面组件称为重量级组件。另外，由于 Swing 组件是 Java 平台自己进行绘制，可以对组件的外观进行完善的自定义。除了 Swing 自带的几个可选的样式风格之外，应用也可以完全定制自己的样式风格。而在用户界面组件方面，Swing 也提供了更加丰富的组件，包括一些常见的复杂组件，如表格和树形控件等。在设计方面，Swing 也更加成熟，很多组件都采用了模型 – 视图 – 控制器（Model-Viewer-Controller, MVC）设计模式，层次更清

晰，使用起来更容易。新开发的桌面应用都推荐使用 Swing，而不是 AWT。

□ SWT

标准小部件工具箱（Standard Widget Toolkit, SWT）是开发 Java 桌面应用时可以使用的另外一个图形用户界面库，也是 Swing 的有力竞争对手之一。SWT 最突出的使用是在 Eclipse 中，它是 Eclipse 使用的底层图形用户界面框架。SWT 的实现方式与 AWT 类似，也采用创建底层操作系统中的原生控件的做法。不过 SWT 比 AWT 更进一步，提供了与 Swing 相匹敌的丰富的组件库。JFace 在 SWT 的基础上又提供了更多实用的组件。

从功能和性能等方面来说，并不存在 Swing 和 SWT 中的一个一定强于另外一个的说法。两者各有各的优势和不足，也都可以满足日常的开发需求。不过两者的 API 风格有比较大的区别，组件的使用方式也有较大不同。对于开发人员来说，两者唯一的区别仅在于已经熟悉或打算熟悉哪种 API 风格而已。熟悉 SWT 和 JFace 的一个额外好处是在开发 Eclipse 插件时会更加得心应手。

SWT 不是 Java 平台默认支持的用户界面库，需要下载额外的第三方库，因此本章没有对 SWT 进行过多的介绍。

□ JavaFX

AWT 和 Swing 一直以来是基于 Java 平台的桌面应用的基本开发框架。AWT 和 Swing 适合开发传统的数据驱动的桌面应用，如信息管理系统等。这类应用大多只使用 AWT 和 Swing 中已有的组件，在数据展现和交互模式上也比较简单。现代的桌面应用对交互性提出了更高的要求，这些新的应用一般大量使用图片、音频和视频等多媒体内容。界面组件库中的通用组件不能满足应用的需求，因此，应用一般会大量使用自定义的组件。这类应用中比较典型的有多媒体应用和游戏等。AWT 和 Swing 不适于这类应用的开发。JavaFX 的作用是推动 Java 桌面开发继续向前发展，以满足现代桌面应用的开发需求。

JavaFX 的第一个版本在 2007 年发布。它的最初目标是开发富互联网应用程序（Rich Internet Application, RIA）。JavaFX 在早期是与 Adobe 的 Flex 及微软的 Silverlight 相互竞争的。受限于 JRE 的部署状况，JavaFX 的表现一直差强人意。Oracle 收购了 Sun 之后，投入了大量的精力对 JavaFX 进行推广和更新。JavaFX 2.0 在 2011 年 10 月正式发布，它调整了 JavaFX 中的很多概念，并且重新设计和实现了很多重要组件。目前，在运行基于 JavaFX 开发的程序时，需要在 JRE 之外安装额外的 JavaFX 运行时（JavaFX runtime）环境支持。不过，JavaFX 运行时环境支持计划在 Java SE 7 Update 2 中与 JRE 共同安装。安装 JRE 7 Update 2 的用户将可以直接运行 JavaFX 程序。JavaFX 3.0 也正在开发中，将作为 Java SE 8 的一个组成部分。在以后的开发中，AWT 和 Swing 将会逐渐淡出桌面应用开发人员的视野，JavaFX 将成为 Java 平台上主流的图形用户界面开发库。

总的来说，使用 Java 平台开发桌面应用并不是一件复杂的事情。大多数时候，你会发现开发的过程比较程式化，基本上就是创建组件、设置其属性和绑定事件处理器这几

步。不过也确实有一些容易出错的地方，后面会进行介绍。

5.2 AWT

AWT 作为 Java 中最早的图形用户界面库，具有了与底层操作系统进行交互的能力。即便后来的 Swing 用 Java 自己的方式来绘制组件，也还要依赖 AWT 与底层操作系统进行交互。从设计的角度来说，AWT 的组件库采用的是基于类继承的层次结构。每一个组件都是 `java.awt.Component` 类的对象。开发人员既可以使用 AWT 中的标准组件，也可以开发自己的组件。`Component` 类中定义了操作组件的各种方法，包括常见属性（如组件的大小、位置、字体和颜色等）的获取和设置，事件监听器的处理，以及组件状态的改变等。关于这些方法的使用，通过方法名称和 API 文档就可以了解到具体的细节，这里不再赘述。

5.2.1 重要组件类

`Component` 类的一个重要子类是 `java.awt.Container`。顾名思义，`Container` 类的对象用来包含其他 AWT 中的 `Component` 类的对象，是一个通用的容器。通过 `Container` 类的 `add` 和 `remove` 方法可以向容器中添加和删除组件。`Component` 类的对象（尤其是 `Container` 类）的一个重要内部属性是该组件的有效状态。一个组件上发生与布局相关的变化之后，这个组件就会被置为无效状态。这些与布局相关的变化包括：组件的大小发生变化，`Container` 类的对象中添加或删除了子组件。当一个组件被置为无效状态的时候，它的层次结构树上的所有祖先组件都会被置为无效状态。通过 `Component` 类的 `isValid` 方法可以检查一个组件当前是否处于有效状态。对于处于无效状态的组件，需要调用其 `validate` 方法来对包含的所有子组件重新进行布局，以恢复到有效的状态。由于 `validate` 方法会处理所有的子组件，因此一般会比较耗时。从性能的角度出发，最好对组件进行批量修改之后，再调用一次 `validate` 方法来重新布局。如果希望直接把某个组件设成无效的状态，可以使用 `invalidate` 方法。

考虑到 `validate` 方法的性能问题，Java 7 把 Swing 中原有的有效性验证根组件（validate root）的概念扩展到了 AWT 中。在之前的实现中，如果一个组件被置为无效状态，那么从该组件的层次结构树向上，直到界面的顶层组件，这条路径上的所有组件都会被置为无效状态。在使用 `validate` 方法的时候要考虑所有无效的组件及其子组件，这无疑对性能影响较大。有效性验证根组件指的是某些特定类型的组件。AWT 中 `Container` 类及其子类所表示的组件都可以作为有效性验证根组件。这些组件的子组件的无效状态不会影响到有效性验证根组件的父组件。也就是说，在沿着层次结构树往上设置父组件的无效状态的时候，如果发现父组件是有效性验证根组件，就不需要再把有效性验证根组件的父组件也置为无效状态了，这样就可以降低无效状态的影响范围。一个组件是否为有效性验证根组件可以通过 `isValidateRoot` 方法来判断。AWT 中最常见的

有效性验证根组件是 `java.awt.Window` 类。Swing 中的 `javax.swing.JScrollPane` 类也是有效性验证根组件。在使用了有效性验证根组件的情况下，可以通过 `revalidate` 方法来使一个组件树快速地重新恢复到有效状态，因为 `revalidate` 会把有效性验证根组件考虑在内。不过为了与早期版本保持兼容性，有效性验证根组件并不是默认启用的，需要通过将 Java 系统参数 “`java.awt.smartInvalidate`” 设为 “`true`” 来启用它。在需要恢复一个组件树到有效状态的时候，开发人员应该使用 `revalidate` 方法而不是 `validate` 方法，因为 `revalidate` 方法的性能要优于 `validate` 方法。

在使用 AWT 的程序界面中，通常都要一个 `java.awt.Frame` 类的对象作为顶层的窗口。`Frame` 类的作用等同于在其他程序中所看到的窗口，可以包含标题栏、边框和菜单。类 `java.awt.Dialog` 表示的是一般程序中常见的对话框，可以是模态或非模态的。`Window` 类是 `Frame` 类和 `Dialog` 类的父类。一个 `Window` 类所表示的组件不能包含边框和菜单。在创建 `Window` 类的对象时，需要指定另外一个已有的 `Frame` 或 `Window` 类的对象作为该组件的所有者。`Window` 类适合于创建程序中的各种不同的自定义窗口。

5.2.2 任意形状的窗口

在桌面应用中，窗口的形状一般以矩形为主。这也是大多数用户所需要的窗口形状。在某些特定的情况下，可能会需要使用其他形状的窗口，其目的也是为了提高用户体验。对于一个音乐播放器应用来说，如果播放窗口的形状是连在一起的多个圆形，类似多个圆形按钮拼接在一起，就可以更加地贴近用户的使用习惯。在 Java 7 中，AWT 的 `Window` 类中新增了用来获取和设置其窗口形状的方法，分别是 `getShape` 和 `setShape`。设置窗口形状时使用的参数是 AWT 中表示几何形状的 `java.awt.Shape` 接口的实现对象。AWT 中本身提供了很多 `Shape` 接口的实现类，用来表示常见的图形，包括弧线、椭圆、多边形、矩形和圆角矩形等。开发人员还可以通过实现 `Shape` 接口来创建独特的形状，这为在桌面应用中使用任意形状的窗口提供了极大的便利。

把一个窗口的形状设置为给定的 `Shape` 接口的实现对象之后，该窗口只有在该形状范围之内的区域才是可见的，在其他区域都是不可见的。设置任意形状的窗口需要底层操作系统的支持，最主要的是支持将界面上的每个像素都设置为完全透明或完全不透明。另外，要求窗口不能包含装饰元素，如标题栏和边框。窗口也不能处于最大化的状态。满足这三个条件之后，就可以设置窗口的形状了。如果想恢复原来的默认形状，只需要传入 `null` 作为调用 `setShape` 方法的参数即可。

代码清单 5-1 给出了任意形状窗口的一个示例，其中把窗口的形状设置成椭圆。代码中使用 `Window` 类的子类 `Frame` 来创建一个顶层窗口，方法 `setUndecorated` 用来去掉窗口包含的装饰元素，`Ellipse2D.Float` 类则表示一个椭圆。

代码清单 5-1 任意形状窗口的示例

```
public void createShapedWindow() {
    Frame frame = new Frame();
```

```

        frame.setUndecorated(true);
        Shape shape = new Ellipse2D.Float(0, 0, 400, 300);
        frame.setShape(shape);
        Label label = new Label("Hello World!");
        frame.add(label);
        frame.setSize(400, 300);
        frame.setVisible(true);
    }

```

5.2.3 半透明窗口

除了任意形状的窗口之外，Java 7 支持的另外一个窗口特殊效果是窗口的半透明化。窗口的半透明化也是很多桌面应用会采用的做法。这种做法既可以带来不错的视觉效果，又可以在特定的情况下帮助用户更好地使用窗口。比如，当用户在一个文档编辑器的当前文档中进行查找的时候，可以把查找窗口设成半透明的。这样用户可以同时看到文档中的内容和查找窗口，方便进行比对和再次查找。与设置窗口形状类似的是，Window 类中有一组方法用来获取和设置窗口的透明度，分别是 getOpacity 和 setOpacity。透明度的区间是 0 到 1，0 表示完全透明，而 1 则表示完全不透明。

实现半透明窗口要求底层操作系统支持为窗口中包含的像素设置透明度值，还要求窗口不包含装饰元素以及不能处于最大化状态。后两点要求与设置窗口形状时一样。而第一点要求的差别在于，设置窗口形状时只要求支持每个像素的透明度值能够被设置为 0 或 1 即可，而设置半透明窗口时要求每个像素的透明度值都能够被设置为 0 到 1 区间内的任意值。代码清单 5-2 给出了半透明窗口的一个示例。在示例中，创建了一个 Swing 的滑动条组件。当滑动条的值发生变化的时候，整个窗口的透明度也会发生改变。

代码清单 5-2 半透明窗口的示例

```

public void createTranslucentWindow() {
    final Frame frame = new Frame();
    frame.setUndecorated(true);
    frame.setSize(400, 300);
    final JSlider slider = new JSlider(0, 100, 80);
    slider.addChangeListener(new ChangeListener() {
        public void stateChanged(ChangeEvent e) {
            frame.setOpacity(slider.getValue() / 100.0f);
        }
    });
    frame.add(slider);
    frame.setOpacity(0.8f);
    frame.setVisible(true);
}

```

5.2.4 组件混合

前面提到过，每个 AWT 组件都存在一个与之对应的底层操作系统上的原生控件，一般称之为重量级组件；而 Swing 是自己绘制用户界面的内容，一般称之为轻量级组件。在 Java 6 Update 12 之前，在同一个用户界面中同时混用重量级和轻量级组件，会产生显示上的问题。这主要是由组件在 Z 轴上的覆盖顺序造成的，所产生的结果是 AWT 组件始终出现在 Swing 组件的前面。这个问题在 Java 6 Update 12 中得到了修正。从而使混用 AWT 和 Swing 组件不再存在显示上的问题。

从实现的角度上来说，Swing 组件也不能完全脱离 AWT 而存在。在 Swing 中，所有组件的父类 javax.swing.JComponent 继承自 AWT 中的 Container 类。虽然 Swing 组件的界面是自行绘制的，但是也需要一个包围它的 AWT 组件，因为 Swing 组件依赖 AWT 与底层的操作系统进行交互。Swing 的做法相当于：AWT 提供一个完全空白的窗口，Swing 的组件直接在这个窗口上进行绘制。用户界面要与底层操作系统进行交互时，通过这个 AWT 窗口来完成即可。所以对一个完全使用 Swing 组件来创建的用户界面来说，它的顶层窗口一定会通过 AWT 与一个原生控件相对应。

虽然混用 AWT 和 Swing 组件不再存在显示上的问题，但仍然推荐应用程序只选用 Swing 或 AWT 中的一种来作为图形界面组件库，尤其推荐使用 Swing。新开发的桌面程序应该考虑使用 JavaFX。

5.3 Swing

Swing 组件库中的核心类是 JComponent。Swing 中所有的非顶层窗口的组件类都继承自 JComponent 类，而顶层窗口组件则继承自 AWT 中的对应组件。Swing 中组件的类名都以“J”作为前缀，以区别于 AWT 中同样功能的组件。相对于 AWT 来说，Swing 组件库的内容更加丰富，性能也更好。

5.3.1 重要组件类

Swing 中的顶层窗口组件需要继承自 AWT 中的相关组件以支持与底层操作系统的交互。Swing 中共有 4 种顶层窗口组件，分别是 javax.swing 包中的 JFrame、JWindow、JDialog 和 JApplet。前面提到过，这些顶层窗口组件实际上是由 Swing 的用户界面提供绘制的区域，这个绘制区域本身又是用 javax.swing.JRootPane 类的对象来表示的。每个顶层窗口组件都只有唯一的一个直接子组件，是一个 JRootPane 类的对象。对窗口中内容的操作也在这个 JRootPane 类的对象上进行。一个 JRootPane 类的对象所表示的组件由两个部分组成。第一部分是作为内容区域之上的遮罩的一个组件（glass pane）。这个组件覆盖在其他组件之上，大小与当前 JRootPane 组件一样，可以用来拦截鼠标和键盘事件。在默认情况下，这个组件是不可见的。第二部分是一个 javax.swing.

JLayeredPane 类的对象。在这个 JLayeredPane 组件中又包含了两个部分：第一部分是作为菜单的 javax.swing.JMenuBar 类的对象。菜单不是必须存在的。第二部分是作为内容区域的 Container 类的对象。内容区域是必须存在的。对于一个 JRootPane 组件来说，最重要的是其中包含的内容区域。内容区域中应该包含一个用户界面中所有非菜单的组件。当通过一个顶层窗口组件的 add 方法来添加组件的时候，默认这个组件实际上被添加到了内容区域中。同样，在设置一个顶层窗口组件的布局格式的时候，实际上设置的是内容区域的布局格式。顶层窗口组件类都实现了 javax.swing.RootPaneContainer 接口。通过 RootPaneContainer 接口中的方法可以对 JRootPane 组件进行操作，如调用 getContentPane 方法可以获取内容区域组件，并对这个组件进行操作。

JRootPane 的遮罩组件可以用来拦截顶层窗口组件中包含内容区域的 JLayeredPane 组件上的鼠标和键盘事件，也可以绘制其他内容来盖住 JLayeredPane 中的组件。当需要暂时屏蔽用户对内容区域的操作时，可以使用这个遮罩组件。JRootPane 的遮罩组件可以是任何继承自 JComponent 类的组件。比如，当程序启动的时候，某些组件可能还没有完全初始化，这时可以把遮罩组件显示出来，给出相关的提示并屏蔽用户的使用。代码清单 5-3 给出了一个示例。LoadingPane 类是一个普通的 JComponent 的子类，它在界面的正中显示字符，同时捕获所有的鼠标事件但不做任何处理。

代码清单 5-3 作为遮罩组件的 JComponent 的子类

```
public class LoadingPane extends JComponent {
    public LoadingPane() {
        addMouseListener(new MouseAdapter() {});
    }

    public void paintComponent(Graphics g) {
        g.drawString(" 加载中 ... ", getWidth() / 2, getHeight() / 2);
    }
}
```

下一步是把 LoadingPane 类的对象作为遮罩组件添加到某个 JRootPane 组件上。代码清单 5-4 给出了示例的做法。其中关键的是要把遮罩组件设置为可见，使它可以发挥作用。程序运行起来之后，会发现内容区域上的组件都无法接收到鼠标事件，因为这些事件已经被 LoadingPane 拦截并处理了。

代码清单 5-4 把遮罩组件添加到 JRootPane 组件

```
LoadingPane pane = new LoadingPane();
frame.setGlassPane(pane);
pane.setVisible(true)
```

代码清单 5-3 中的 LoadingPane 类只是一个简单的遮罩组件的实现。可以根据需要开发出更加复杂的实现，比如只拦截某些组件上的事件，或者实现复杂的图形绘制功能。

5.3.2 JLayer 组件和 LayerUI 类

Java 7 引入了一个新的用户界面组件 javax.swing.JLayer，可以把它看成覆盖在一个已有组件上的图层。JLayer 组件可以对这个已有的组件进行装饰，也可以监听这个组件上发生的事件。JLayer 组件可以把对已有组件所做的处理抽象出来，由专门的类来管理，并应用到其他组件上去。比如，希望监听任何组件上的鼠标单击事件，可以把监听事件的这部分逻辑抽象到一个具体的类中。当需要这种行为的时候，只需要对已有的组件应用这个类的行为即可。JLayer 组件本身只负责与已有的组件进行绑定，真正的处理工作交由 javax.swing.plaf.LayerUI 类来完成。LayerUI 类中封装了对组件所做的处理行为。在得到一个 LayerUI 类的对象之后，可以通过 JLayer 类的对象将其应用到不同的组件上。

LayerUI 类主要可以完成两类操作：一类是对已有组件的界面进行修改，另外一类是对已有组件上产生的事件进行处理。在 LayerUI 类中可以通过覆写 paint 方法来对已有组件的外观进行修改。代码清单 5-5 给出了一个 LayerUI 类的实现。这个 LayerUI 类所封装是一种通用的高亮显示一个组件的行为。在这个实现中，首先调用父类的 paint 方法来进行正常的界面绘制，再在组件的边界上绘制一个红色的边框。

代码清单 5-5 高亮显示组件的 LayerUI 类的实现

```
public class HighlightLayerUI extends LayerUI {
    public void paint(Graphics g, JComponent c) {
        super.paint(g, c);
        g.setColor(Color.red);
        g.drawRect(0, 0, c.getWidth() - 1, c.getHeight() - 1);
    }
}
```

有了 LayerUI 类之后，下一步是创建一个 JLayer 类对象并把 LayerUI 类对象与一个已有的组件关联起来。这一步并不复杂，只需要用一个已有的组件和 LayerUI 类的对象作为参数来创建一个新的 JLayer 类的对象，再把该 JLayer 类的对象像其他组件一样添加到界面中即可。这种使用方式相当于用 JLayer 类的对象对已有的组件进行包装。代码清单 5-6 给出了 JLayer 类的一个使用示例。这个例子展示了 JLayer 类的一种实用的应用场景，可以在运行时动态地改变已有组件的外观。当用户的鼠标移动到某个按钮上时，通过 JLayer 类的对象的 setUI 方法为这个按钮添加代码清单 5-5 中的 HighlightLayerUI 类所封装的行为，完成后会在按钮上加上一个红色的边框；而在鼠标移开的时候，则为按钮设置了默认的 LayerUI 类的实现，按钮就恢复了默认的外观。这种动态改变外观的能力可以使程序在某些情况下变得更加灵活。

代码清单 5-6 高亮显示组件的 LayerUI 类的使用示例

```
public void useHighlight() {
```

```

final JFrame frame = new JFrame();
frame.setSize(400, 300);
frame.setLayout(new GridLayout(2, 1));
frame.add(new JLabel("标签"));
JButton button = new JButton("按钮");
final JLayer<? extends Component> layer = new JLayer<>(button);
final LayerUI layerUI = new HighlightLayerUI();
final LayerUI defaultUI = new LayerUI();
frame.add(layer);
button.addMouseListener(new MouseAdapter() {
    public void mouseEntered(MouseEvent e) {
        layer.setUI(layerUI);
    }

    public void mouseExited(MouseEvent e) {
        layer.setUI(defaultUI);
    }
});
frame.setVisible(true);
}

```

除了改变外观之外，JLayer 类还可以处理它所包装的组件上的事件。当组件上发生事件的时候，LayerUI 类的对象可以得到通知，可以获取到该事件的相关信息。通过这个功能，可以监听组件上发生的事件。代码清单 5-7 给出了一个示例，LayerUI 类的 installUI 方法在 LayerUI 类的对象被配置到一个 JLayer 类的对象上时会被调用。在这个方法中，通过 JLayer 类的 setLayerEventMask 方法设置了所感兴趣的事件类型，这里设置了只对鼠标相关的事件感兴趣。而 uninstallUI 方法则在取消 JLayer 类的对象与对应的 LayerUI 类的对象的关联时被调用。在这个方法中，要取消在 installUI 方法中设置的感兴趣的事件类型，并恢复为默认类型。对于相关的事件，在 LayerUI 类中有相应的方法来进行处理，比如 processMouseEvent 方法用来处理鼠标事件，processKeyEvent 方法用来处理键盘事件。只要在 installUI 方法中设置了对某类事件感兴趣，当事件发生的时候，LayerUI 类中的相关事件处理方法就会被调用。

代码清单 5-7 使用 LayerUI 类来监听组件上发生的事件

```

public class MouseMonitorLayerUI extends LayerUI {

    public void installUI(JComponent c) {
        super.installUI(c);
        JLayer layer = (JLayer) c;
        layer.setLayerEventMask(AWTEvent.MOUSE_EVENT_MASK);
    }

    public void uninstallUI(JComponent c) {
        super.uninstallUI(c);
        JLayer layer = (JLayer) c;
    }
}

```

```

        layer.setLayerEventMask(0);
    }

    public void processMouseEvent(MouseEvent e, JLayer l) {
        System.out.println(e paramString());
    }
}

```

需要注意的是，LayerUI 类的对象只能在对应 JLayer 类的对象所包装的组件上的事件发生时得到通知，并不能拦截事件的发生，所以 LayerUI 类一般用来实现监视相关功能。

5.4 事件处理与线程安全性

AWT 和 Swing 都是采用经典的事件处理模型。系统中的内部动作和用户的操作都会被抽象成为对应的事件，放入到一个事件队列中。这个队列中的事件会被依次进行处理。

5.4.1 事件处理

在 AWT 中的事件用 java.awt.AWTEvent 类来表示，而事件队列则由 java.awt.EventQueue 类来表示。EventQueue 类中包含了对 AWTEvent 类的对象的处理方法，包括发布、获取、查看和分发事件等。事件队列的基本处理思路是：对于队列中当前存在的事件，按照顺序逐个调用 dispatchEvent 方法来分发这些事件。在分发的过程中就包括了对事件的处理，只不过实际的处理可能是由其他对象来完成的。

事件队列的一个实用功能是可以往其中发布自定义的事件，由事件队列来处理。代码清单 5-8 中给出了一个自定义事件的使用示例。当需要创建一个任意类型的自定义事件的时候，可以实现 java.awt.ActiveEvent 接口。当一个 ActiveEvent 接口的实现对象被分发时，其 dispatch 方法会被调用，相当于对该事件进行处理。通过 EventQueue 类的 postEvent 方法可以向事件队列中添加事件。由于 EventQueue 类要求队列中的事件对象都继承自 AWTEvent 类，所以自定义的事件类也需要继承自 AWTEvent 类。在一个 AWTEvent 类中需要包含两个要素：第一个是产生事件的源对象，可以是任意的 Java 对象；第二个是事件类型的标识符，它的值需要大于 AWT 为内部事件保留的最大标识符 AWTEvent.RESERVED_ID_MAX，以避免与 AWT 定义的内部事件产生冲突。

代码清单 5-8 自定义事件的使用示例

```

public class UseEventQueue {

    private static class MyEvent extends AWTEvent implements ActiveEvent {
        private String content;
        public MyEvent(String content) {

```

```

        super(content, AWTEvent.RESERVED_ID_MAX + 1);
        this.content = content;
    }

    public void dispatch() {
        System.out.println("字符串长度为：" + content.length());
    }
}

public void useEventQueue() {
    Frame frame = new Frame();
    frame.setVisible(true);
    Toolkit toolkit = Toolkit.getDefaultToolkit();
    EventQueue queue = toolkit.getSystemEventQueue();
    queue.postEvent(new MyEvent("Hello"));
}
}

```

除了自定义的事件之外，在 AWT 和 Swing 中，使用更多的是与组件相关的事情。EventQueue 类在通过 dispatchEvent 方法来分发事件时，会根据事件的源对象类型和事件本身的类型来确定所采取的动作。比如前面提到的 ActiveEvent 类型的事件，处理的方式是直接调用该类型对象的 dispatch 方法；而对于源对象类型为 AWT 中的 Component 类和 MenuComponent 类的对象的事件来说，处理的方式是直接调用源对象的 dispatchEvent 方法，也就是说，对这类事件的处理由组件自己来完成；对于其他类型的事件，EventQueue 类会直接忽略。除了使用系统提供的 EventQueue 类的对象之外，也可以自己创建 EventQueue 类的对象并使用。通过 java.awt.Toolkit 类可以与系统提供的事件队列进行交互。

当事件发生之后，AWT 和 Swing 中对事件的处理采用了经典的监听者设计模式。在 Component 类及其子类中都有类似“add×××Listener(×××Listener)”这样的方法，用来对组件上相关事件添加处理器。比如，对组件上的鼠标事件感兴趣，可以通过 addMouseListener 方法来添加一个 java.awt.event.MouseListener 接口的实现对象。这样当组件上与鼠标相关的事件发生的时候，预先声明的处理器的对应方法就会被调用。这种事件处理模式在其他编程语言的用户界面组件库以及 Web 开发中也经常用到，开发人员应该都比较熟悉。×××Listener 接口中通常包含了与某个主题相关的一系列需要实现的方法，比如 MouseListener 接口包括了鼠标单击、按下、释放、进入和离开等具体事件的处理方法。如果只对接口中的某几个方法感兴趣，那么可以继承自与该接口对应的 ×××Adapter 类，并覆写其中的部分相关方法即可。结合前面提到的 EventQueue 类的实现，不难得出 AWT 中事件处理的基本流程。

当事件发生的时候，事件被添加到系统的事件队列中。该事件中包含了产生事件的源对象和相关的元数据。事件队列按照顺序依次对其中包含的事件进行处理。在处理的时候，如果发现这是一个 Component 类的对象上发生的事件，直接调用 Component 类

的对象的 dispatchEvent 方法。而 dispatchEvent 方法会调用当前 Component 类的对象上已经注册的事件监听器中的方法，从而完成对事件的处理。对于一个 Component 类的对象，可以添加同一类事件的多个监听器。这些监听器中的方法被调用的先后顺序是不确定的。程序的处理逻辑不应该依赖特定的监听器调用顺序。各个监听器之间应该互相独立。

5.4.2 事件分发线程

AWT 和 Swing 中对用户界面组件的事件处理都是单线程的。对于每个事件队列，都有一个专门的线程负责进行事件的分发和处理。在分发事件的时候，事件对应的处理方法也是在这个线程中被调用的。这就要求在一个事件的处理方法中包含的逻辑应该尽可能地少，可以在较短的时间内完成，否则可能造成队列中的其他事件长时间处于等待状态，给用户的感觉就是程序的用户界面在较长的时间内失去了响应。比如，当用户界面的窗口从最小化状态恢复的时候，需要重新绘制界面，系统会在事件队列中添加一个新的重新绘制的事件来指示各组件重新绘制自身的界面。如果当前事件队列中已经有一个事件正在处理，而这个事件的处理又很耗时，那么这个新的重新绘制的事件在较长的时间内不会被分发，会导致用户界面成为一片空白，没有任何内容。

从线程的安全性方面来说，AWT 和 Swing 的大部分 API 都不是线程安全的，这包括绝大部分操作用户界面的 API。如果多个线程同时对界面进行更新，可能会造成界面的显示问题。实际上，要实现一个线程安全的用户界面组件库是一件很困难的事情。AWT 和 Swing 没有选择花费大量的精力去实现线程的安全性，而是采取了一种简单做法，即要求所有与界面相关的操作都在单一的线程中进行。这种单线程的界面更新方式自然可以避免线程安全性的问题，这个线程就是前面提到的事件队列对应的事件分发线程（Event Dispatch Thread，EDT）。

在大多数情况下，事件分发线程都在后台工作，开发人员也不需要了解它的存在。与事件分发线程产生联系的一个典型场景是需要执行长时间任务的时候。如果一个任务的执行时间很长，那么由事件分发线程来处理就不合适，因为这样会导致其无法及时处理其他事件。合理的做法是由另外一个工作线程来处理。不过要注意的是，如果在处理中需要更新用户界面，应该交由事件分发线程来处理，而不是由当前工作线程处理。这么做的目的是为了保证与界面相关的操作都在同一个事件分发线程中执行，避免线程安全性的问题。

AWT 和 Swing 都提供了相关的方法来在事件分发线程中执行更新界面的操作。代码清单 5-9 给出了一个 AWT 中事件分发线程的使用示例。示例中使用一个后台线程来计算 π 的值，计算出结果后显示在一个标签组件中。因为需要在事件分发线程中执行界面相关的操作，对 java.awt.Label 类的 setText 方法的调用需要封装在 EventQueue 类的 invokeLater 方法中。方法 invokeLater 参数中的 java.lang.Runnable 接口的实现对象

会在系统默认事件队列对应的分发线程中执行。与 invokeLater 方法作用相似的方法是 invokeAndWait，两者的区别在于： invokeLater 方法只是把 Runnable 接口实现对象放入事件队列中就立刻返回，实际的执行需要等待队列中的其他事件处理完毕，所以是一个异步操作；而 invokeAndWait 方法会等待队列中的所有其他事件及 Runnable 接口实现对象的 run 方法执行完成之后才返回，所以是一个同步操作。开发人员可以根据需要选择异步方式还是同步方式。

代码清单 5-9 AWT 中事件分发线程的使用示例

```

public class CalculatePi {
    public void calculate() {
        Frame frame = new Frame();
        Label label = new Label();
        frame.add(label);
        frame.setSize(400, 300);
        frame.setVisible(true);
        new CalculateThread(label).start();
    }

    private static class CalculateThread extends Thread {
        double sum = 0.0, term, sign = 1.0;
        int N = 30 * 1000 * 1000;
        private Label label;

        public CalculateThread(Label label) {
            this.label = label;
        }

        public void run() {
            for (int k = 0; k < N; k++) {
                term = 1.0 / (2.0 * k + 1.0);
                sum = sum + sign * term;
                if (k % (N / 100) == 0) {
                    EventQueue.invokeLater(new Runnable() {
                        public void run() {
                            label.setText(Double.toString(4 * sum));
                        }
                    });
                }
                sign = -sign;
            }
        }
    }
}

```

Swing 中也有与 AWT 中的 EventQueue 类的 invokeLater 和 invokeAndWait 方法功能相同的方法，这些方法在 javax.swing.SwingUtilities 类中。 SwingUtilities 类中对应的

方法的名称也是 invokeLater 和 invokeAndWait，调用时的行为也是相同的。对于一个使用 Swing 的桌面应用来说，应该把创建用户界面相关的逻辑都封装在 SwingUtilities 的 invokeLater 方法中。如代码清单 5-10 所示，程序的 main 方法应该对创建用户界面的 createUI 的方法的调用进行封装，以确保该方法在事件分发线程中执行。

代码清单 5-10 SwingUtilities 类的 invokeLater 方法的使用示例

```
public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            createUI();
        }
    });
}
```

5.4.3 SwingWorker 类

在 Swing 中，如果要实现后台运行的工作线程，更好的做法是使用 Java 6 中引入的 javax.swing.SwingWorker 类。SwingWorker 类的作用是把工作线程的任务和用户界面的更新这两者做了一个良好的切分。对于 SwingWorker 类的对象中的方法，一部分在工作线程中被调用，而另外一部分则在事件分发线程中被调用。开发人员只需要按照 SwingWorker 类的约定实现这些方法即可，不需要考虑这些方法所对应的线程的含义。SwingWorker 类会负责保证这些方法在正确的线程中被调用。

SwingWorker 类中唯一需要实现的方法是 doInBackground。这个方法用来执行具体的工作，是在工作线程中运行的。方法 doInBackground 的返回值是工作线程的执行结果。如果希望得到 doInBackground 方法的执行结果，可以调用 SwingWorker 类的 get 方法。这个方法会阻塞直到 doInBackground 方法执行结束。一般来说，get 方法会在事件分发线程中被调用。在这种情况下，会无法处理事件队列中的其他事件，使程序的界面暂时处于没有响应的状态。因此，如果要在事件分发线程中等待工作线程的完成，比较好的做法是在等待过程中弹出一个模态对话框来提示用户。

在任务的执行过程中，可能会需要在用户界面上进行更新来显示任务的执行进度，比如更新进度条的指示值。一般通过 3 种做法来实现进度信息在工作线程和事件分发线程之间的传递。第一种做法是通过 SwingWorker 类的 setProgress 方法来更新任务的执行进度。进度的值从 0 到 100，非常适合直接供进度条组件来使用。第二种做法是使用 publish 和 process 方法，其中 publish 方法是在工作线程中被调用的，用来发布任务执行过程中产生的中间结果；process 方法是在事件分发线程中被调用的，利用 publish 方法产生的中间结果来更新用户界面。通过 publish 方法发布的数据会在 process 方法调用时的实际参数中得到。第三种做法是使用自定义的属性变化事件。在 SwingWorker 类中可以通过 firePropertyChange 方法来发布属性变化的事件。可以在事件分发线程中通过

SwingWorker 类的对象的 addPropertyChangeListener 方法来添加属性变化事件的处理方法。实际上，第一种做法中提到的 setProgress 方法也是以属性变化的方式来实现的，只不过用的是预定义的属性名称“progress”。

完成 doInBackground 方法之后，done 方法会在事件分发线程中被调用，用来在任务完成之后进行界面的更新。自定义的 SwingWorker 类如果需要在任务完成之后更新界面，应该直接覆写 done 方法。如果希望取消一个任务的执行，可以使用 SwingWorker 类的 cancel 方法，不过 cancel 方法的成功完成，要求 doInBackground 方法在执行的过程中不时地通过 isCancelled 方法来检测是否发出了取消的请求，以取消任务。如果在 doInBackground 方法的实现中没有处理取消请求的相关逻辑，调用 cancel 方法实际上是不起作用的。一个设计良好的 doInBackground 方法的实现，应该允许用户随时取消任务的执行。在创建了 SwingWorker 类的对象之后，通过它的 execute 方法就可以启动其任务的执行。

代码清单 5-11 给出了 SwingWorker 类的一个示例，用来模拟网络上远程文件的下载过程。在 doInBackground 方法中通过 setProgress 方法来更新下载的进度，同时通过 publish 方法把当前的下载速度发布出去。而在事件分发线程中，通过 SwingWorker 类的 addPropertyChangeListener 方法来检查“process”属性的变化，并更新进度条组件的显示。在 SwingWorker 类的 process 方法中，则把 publish 方法发布出来的下载速度的值显示在标签上。在 process 方法被调用的时候，可能会收到多次 publish 方法调用的结果，程序应该根据需要从结果列表中选择合适的数据来显示。

代码清单 5-11 SwingWorker 类的使用示例

```

public class UseSwingWorker {
    public void downloadFile() {
        JFrame frame = new JFrame();
        final JProgressBar progressBar = new JProgressBar();
        frame.add(progressBar, BorderLayout.NORTH);
        final JLabel label = new JLabel();
        frame.add(label, BorderLayout.CENTER);
        DownloadWorker worker = new DownloadWorker(label);
        worker.addPropertyChangeListener(new PropertyChangeListener() {
            public void propertyChange(PropertyChangeEvent evt) {
                if ("progress".equals(evt.getPropertyName())) {
                    progressBar.setValue((Integer) evt.getNewValue());
                }
            }
        });
        worker.execute();
        frame.setSize(400, 300);
        frame.setVisible(true);
    }

    private static class DownloadWorker extends SwingWorker<String, Double> {

```

```

private JLabel label;
public DownloadWorker(JLabel label) {
    this.label = label;
}

public String doInBackground() throws Exception {
    Random random = new Random();
    for (int i = 0; i < 100; i++) {
        Thread.sleep(random.nextInt(1000));
        setProgress(i + 1);
        publish(random.nextDouble() * 30);
    }
    return "<Path>";
}

protected void process(List<Double> chunks) {
    Double speed = chunks.get(chunks.size() - 1);
    label.setText(MessageFormat.format("下载速度: {0,number,#.##} kb/s",
        speed));
}
}

```

5.4.4 SecondaryLoop 接口

在介绍 SwingWorker 类的时候曾经提到，如果希望当前线程等待 SwingWorker 类的对象的工作线程运行完成，可以调用 `get` 方法，该方法会阻塞当前线程直到 SwingWorker 类的对象中的任务完成或被取消。如果调用 `get` 方法的线程是事件分发线程，那么会造成用户界面失去响应，有时候确实需要等待某个任务完成才能进行下一步操作。为了解决这种同步调用和事件分发线程处理之间的矛盾，Java 7 对 AWT 中的事件处理进行了增强，添加一个额外的事件队列。这个新的事件队列既可以阻塞当前线程以等待操作完成，又可以继续处理事件队列中的事件，并且不会造成界面失去响应。这个新的事件队列由 `java.awt.SecondaryLoop` 接口来表示，通过 `EventQueue` 类的对象的 `createSecondaryLoop` 方法来创建具体的实例。在创建出 `SecondaryLoop` 接口的实现对象之后，可以调用该对象的 `enter` 方法来阻塞当前线程并进入新的事件处理循环中；当要等待的操作完成之后，可以调用 `exit` 方法来退出这个新的事件处理循环，同时恢复被 `enter` 方法阻塞的线程的执行。

在需要等待操作完成的时候，都应该使用 `SecondaryLoop` 接口，它可以避免用户界面失去响应的问题。代码清单 5-12 给出了一个示例，通过一个线程来模拟需要等待完成的操作。在线程启动之后，通过 `SecondaryLoop` 接口的 `enter` 方法来阻塞事件分发线程；而在工作线程完成任务之后，通过 `SecondaryLoop` 接口的 `exit` 方法来恢复事件分发线程的执行。在工作线程的运行过程中，用户界面产生的其他事件仍然可以得到处理。

代码清单 5-12 SecondaryLoop 接口的使用示例

```

private static class WorkerThread extends Thread {
    private SecondaryLoop loop;
    public WorkerThread(SecondaryLoop loop) {
        this.loop = loop;
    }

    public void run() {
        try {
            Thread.sleep(5000);
        } catch (InterruptedException ex) {
        }
        loop.exit();
    }
}

public void useLoop() {
    JFrame frame = new JFrame();
    frame.setSize(400, 300);
    frame.addMouseListener(new MouseAdapter() {
        public void mouseClicked(MouseEvent e) {
            EventQueue queue = Toolkit.getDefaultToolkit().getSystemEventQueue();
            SecondaryLoop loop = queue.createSecondaryLoop();
            WorkerThread thread = new WorkerThread(loop);
            thread.start();
            loop.enter();
        }
    });
    frame.setVisible(true);
}

```

5.5 界面绘制

用户界面组件库中的组件都要在屏幕上绘制之后显示给用户。Java 平台上的 AWT 和 Swing 也不例外。在 Swing 还没出现之前，AWT 并不需要负责管理组件的界面绘制工作，因为这些是由操作系统底层的原生控件负责完成的。在 Swing 出现之后，由于 Swing 组件的界面是自己绘制组件的，因此需要对绘制的过程进行管理，而作为 Swing 基础的 AWT 也需要有相应的实现。

5.5.1 AWT 中的界面绘制

AWT 中的界面绘制围绕着 Component 类中的 paint 方法展开。这个方法用来绘制 AWT 组件的界面。当系统认为某个组件需要绘制的时候，就会调用该组件的 paint 方法，并传入预先配置好的 java.awt.Graphics 类的对象作为参数。在 paint 方法的实现中，

需要利用这个 `Graphics` 类的对象所提供的方法来完成绘制。

有两类情况会造成 `paint` 方法被调用。第一类是由系统根据组件的状态来决定的，可能造成 `paint` 方法被调用的情况包括：组件第一次通过 `setVisible` 方法设置为在屏幕上可见的时候，组件的大小发生改变的时候，以及组件的部分区域需要重新绘制的时候。第二类情况是程序本身通过 `repaint` 方法来要求组件重新进行绘制，这通常是因为程序的内部状态发生了改变，组件需要重新绘制以反映这些变化。调用 `repaint` 方法也会使得 `paint` 方法被调用。程序不应该直接调用 `paint` 方法，而是通过调用 `repaint` 方法来发出通知，由系统来调用 `paint` 方法。

在 `paint` 方法中进行界面绘制时需要考虑的一个重要属性是本次绘制时的剪辑区域 (`clip bounds`)。通过作为 `paint` 方法实际参数的 `Graphics` 类的对象的 `getClipBounds` 方法可以获取这个剪辑区域。剪辑区域的含义是只有这个区域内的部分界面需要重新绘制，其余部分则不需要。如果是系统产生的 `paint` 方法调用，剪辑区域的范围由系统来确定；如果是程序通过 `repaint` 方法产生的 `paint` 方法调用，剪辑区域可以通过调用 `repaint` 方法时的参数来指定。在 `paint` 方法的实现中，应该考虑到剪辑区域的存在，只重新绘制剪辑区域即可，而不是在任何情况下都把组件的全部区域重新绘制。考虑剪辑区域的重要好处是可以提高界面绘制时的性能。在程序中调用 `repaint` 方法的时候，应该首先计算出需要重新绘制的区域的范围，再把该区域作为调用时的参数传入。如果使用不带参数的 `repaint` 方法，则说明组件的全部区域都需要重新绘制。

5.5.2 Swing 中的绘制

Swing 中的界面绘制继承了 AWT 中界面绘制的特性，又有自己的增强功能。在 Swing 组件中同样可以利用 `paint` 方法绘制组件的界面。不过 Swing 对 `paint` 方法重新进行了细分，分成了 3 个具体的方法，分别是 `paintComponent`、`paintBorder` 和 `paintChildren`。从这 3 个方法的名称可以看出，它们分别用来绘制组件自身的区域、边框和子组件。在一般情况下，Swing 组件只需要覆写 `paintComponent` 方法来添加与绘制自身界面相关的逻辑。另外两个方法使用默认实现即可。

为了提高界面绘制时的性能，Swing 默认启用了双缓冲的技术。在使用双缓冲的时候，界面的绘制是在一个屏幕显示范围之外的图形上下文中进行的。等全部绘制工作完成之后，再把该图形上下文中的内容一次性复制到当前可见的显示区域中。这种技术的好处在于，界面的更新是平滑进行的，可以一次性完成整个界面的更新，用户看不到更新的中间结果。

Swing 中组件绘制的一个额外属性是组件的透明性。有些组件在绘制的过程中可能会不占满组件的全部可用空间，比如可能只绘制了一个组件的左半部分，而空着右半部分不用。对于这空着的右半部分，系统需要把它变为透明的，而让该组件下方的组件的部分内容显示出来。这就要求系统进行计算来找到底层的组件，然后先绘制底层的组

件，再绘制位于其上方的当前组件，这会对系统的绘制性能产生比较大的影响。为此，`JComponent` 类定义了一个“`opaque`”的属性来声明此组件在透明性上的特征。如果通过 `setOpaque` 方法设置了属性“`opaque`”的值为“`true`”，就说明该组件在绘制的时候会占满全部可用的空间，系统不需要额外的工作来使底层组件可见，否则系统需要按照自底向上的方式逐个重新绘制相关组件。对于组件的实现者来说，尽量把“`opaque`”的属性设为“`true`”，同时在绘制时使用组件的全部可用空间。

在绘制的时候，另一个棘手的问题是处理组件之间互相重叠的情况。当互相重叠的组件中有一个需要重新绘制时，与它重叠的其他组件的部分区域也可能需要被重新绘制。组件之间的重叠关系有可能很复杂。系统在确定哪些重叠组件需要重新绘制时，需要花费大量的时间进行计算。如果组件能够额外提供一些与重叠相关的信息，计算的时间就可以大大减少。如果组件的直接子组件之间不会互相重叠，在计算时就可以更加高效。如果组件的实现者确实能够保证该组件的直接子组件都不会互相重叠，可以通过覆盖 `isOptimizedDrawingEnabled` 方法并返回“`true`”来表明这一点。系统可以利用这一特征来提高重叠计算时的速度。

前面提到的属性“`opaque`”和方法“`isOptimizedDrawingEnabled`”都可以看成组件与 Swing 系统之间的契约。Swing 系统可以利用这两个属性来更快地完成界面的绘制工作。而对于组件的实现者来说，要保证所声明的属性的值与其内部的实现是一致的，否则可能出现显示问题。

5.6 可插拔式外观样式

Swing 相对于 AWT 的一个重要优势是 Swing 允许开发人员定制其组件的外观样式，而 AWT 只能使用底层操作系统提供的组件外观样式。Swing 的这个特性为开发人员在美化程序外观样式时提供了足够的灵活性。Swing 所提供的这个特性被称为可插拔式外观样式 (pluggable look-and-feel, PLAF)。Swing 也默认提供了一些外观样式供开发人员选择，这其中包括默认使用的“metal”样式和 Java 7 中新增的“nimbus”样式。除了这些在不同平台上相同的外观样式之外，还包括不同平台上特有的外观样式。程序外观样式的管理由 `javax.swing.UIManager` 类来完成。通过 `UIManager` 类可以为程序切换不同的外观样式。

代码清单 5-13 给出了一个切换外观样式的示例。在示例中，通过 `UIManager` 类的 `getInstalledLookAndFeels` 方法可以获取当前 Java 平台上所有可用的外观样式的信息。每种信息由 `UIManager.LookAndFeelInfo` 类的对象来表示。通过 `UIManager` 类的 `setLookAndFeel` 方法可以设置所要使用的外观样式。设置时的参数既可以是一个已有的 `javax.swing.LookAndFeel` 类的对象，也可以是外观样式实现类的名称。这里使用的是类的名称。在完成设置之后，需要通过 `SwingUtilities` 类的 `updateComponentTreeUI` 方法来通知界面上的所有组件更新自己的外观样式。如果不使用这个方法，可能会出现界面上

组件的外观样式不一致的情况。在示例中，每当通过下拉列表选择一个外观样式之后，整个程序的外观样式会随之发生变化。

代码清单 5-13 切换 Swing 界面的外观样式的示例

```

public void selectPlaf() {
    final JFrame frame = new JFrame();
    UIManager.LookAndFeelInfo[] lafs = UIManager.getInstalledLookAndFeels();
    JComboBox combo = new JComboBox(lafs);
    combo.addItemListener(new ItemListener() {
        public void itemStateChanged(ItemEvent e) {
            if (ItemEvent.SELECTED == e.getStateChange()) {
                UIManager.LookAndFeelInfo info = (UIManager.LookAndFeelInfo)
                    e.getItem();
                try {
                    UIManager.setLookAndFeel(info.getClassName());
                    SwingUtilities.updateComponentTreeUI(frame);
                } catch (Exception ex) {
                }
            }
        }
    });
    frame.add(combo, BorderLayout.NORTH);
    frame.add(new JButton("按钮"), BorderLayout.SOUTH);
    frame.setSize(400, 300);
    frame.setVisible(true);
}

```

如果希望基于 Swing 开发的程序的界面像 AWT 一样在不同的操作系统平台上使用当前平台的默认外观样式，可以通过 UIManager 类的 getSystemLookAndFeelClassName 方法来得到当前平台的默认外观样式的名称，再通过 setLookAndFeel 方法进行设置即可。

Swing 为了实现可插拔的外观样式，采用了一种用户界面代理（UI delegate）的设计思路。当一个 Swing 用户界面组件在进行绘制时，实际的绘制工作是代理给另外的一个对象来完成的。通过这种职责分离方式，使得同一个组件的外观样式可以根据用户界面代理的不同而发生变化。Swing 中所有的用户界面代理类都继承自 javax.swing.plaf.ComponentUI 类。每个组件都有其对应的用户界面代理类来负责绘制其外观，比如与 javax.swing.JButton 类对应的代理类是 javax.swing.plaf.ButtonUI 类。有两种做法可以把一个组件对象和一个用户界面代理对象关联起来：一种是通过组件对象的 setUI 方法，另外一种是通过用户界面代理对象的 installUI 方法。前面介绍 JLayer 和 LayerUI 类时就提到了这两种做法。从这里可以看出，每个样式外观其实是一组用户界面代理类的集合。这些用户界面代理类分别为 Swing 中的不同组件提供了对应的外观样式。

如果使用 setUI 方法来设置组件对应的代理对象，就要求对每个组件的对象实例进行设置。如果需要对某一类型组件的所有对象实例都应用某种外观样式，更好的

选择是使用自定义的外观样式实现。实现自定义的外观样式很简单，只需要继承自 LookAndFeel 类并实现相应的方法，然后再通过 UIManager 类来进行设置即可。如果只是修改少数组件的外观，那么更好的选择是继承已有的 javax.swing.plaf.basic.BasicLookAndFeel 类。BasicLookAndFeel 类可以作为实现自定义的外观样式的基础。比如，希望把界面上所有的标签都改成黑底白字的显示方式，可以继承自 javax.swing.plaf.basic.BasicLabelUI 类，并提供自己的绘制逻辑，如代码清单 5-14 所示。

代码清单 5-14 自定义标签的外观样式的示例

```
public class MyLabelUI extends BasicLabelUI {
    public static ComponentUI createUI(JComponent c) {
        return new MyLabelUI();
    }

    public void paint(Graphics g, JComponent c) {
        g.setColor(Color.BLACK);
        g.fillRect(0, 0, c.getWidth(), c.getHeight());
        g.setColor(Color.WHITE);
        JLabel label = (JLabel) c;
        g.drawString(label.getText(), 0, label.getHeight() / 2);
    }
}
```

通过覆写用户界面代理类的 paint 方法可以控制组件的外观样式。从 paint 方法的 JComponent 类型的参数可以获取与用户界面代理对象相关联的组件，从而获得所需的信息。代码清单 5-14 获取了标签的文本。需要注意的是，自定义的界面代理类都需要覆写 createUI 这个静态方法以创建出正确的界面代理对象。下一步是创建程序自己的 LookAndFeel 类，如代码清单 5-15 所示。程序自己的 MyLookAndFeel 类选择继承 BasicLookAndFeel 类以减少实现的工作量。受限于篇幅，代码清单 5-15 中省略了 MyLookAndFeel 类的一些方法，这些方法都只是用来提供自定义外观样式的元数据，作用并不大。这里只列出了比较重要的 initClassDefaults 方法。在这个方法中，通过修改参数中的 javax.swing.UIDefaults 类的对象把标签组件对应的用户界面代理的 Java 类名设成了 MyLabelUI 类。经过这样的设置，在创建标签组件时，MyLabelUI 类的对象会作为默认的界面代理对象，从而成功地应用自定义的外观样式。如果还需要修改其他类型组件的外观样式，只要按照相同的方式进行设置即可。

代码清单 5-15 启用自定义外观样式的示例

```
public class MyLookAndFeel extends BasicLookAndFeel {
    public void initClassDefaults(UIDefaults table) {
        super.initClassDefaults(table);
        table.putDefaults(new Object[] {"LabelUI", "com.java7book.chapter5.plaf.
            mylaf.MyLabelUI"});
    }
}
```

在程序中，只需要在最开始的时候通过 UIManager 类设置使用 MyLookAndFeel 类作为外观样式即可。

5.7 JavaFX

JavaFX 是利用 Java 开发桌面应用的发展方向。前面提到，JavaFX 将与 Java 平台本身进行更加深度的集成，JavaFX 把 Java 平台变成了一个开发富客户端应用（Rich Client Platform, RCP）的良好平台。使用 JavaFX 可以开发出功能强大和交互性强的桌面应用。JavaFX 应用也可以通过类似 Java Applet 的方式运行在浏览器中。本节将对 JavaFX 2.0 进行比较具体的介绍。

5.7.1 场景图

JavaFX 应用程序的编写方式与 AWT 及 Swing 有很大不同。JavaFX 用了更加形象的方式来描述用户界面及其变化。JavaFX 的这种方式类似于戏剧表演，戏剧表演在一个舞台上进行，一部剧可由多幕组成，每一幕的内容各不相同，完成当前一幕的表演之后，会切换到下一幕。在 JavaFX 中，`javafx.stage.Stage` 类的作用类似于戏剧表演时的舞台，是一个顶层容器，用来包含其他的界面组件。而 `javafx.scene.Scene` 类的作用类似于戏剧表演中的不同幕，表示程序运行时的不同场景。通过 Stage 类的 `setScene` 方法可以切换显示不同的场景。每个场景中可以包含以树形结构组织的多个节点，称之为场景图（scene graph）。JavaFX 把用户界面上的基本图形元素和用户界面组件两类元素进行了统一，统称为节点，用 `javafx.scene.Node` 类来表示。矩形、椭圆、按钮或表格，都是用户界面上的节点，可以用相似的方式来处理。

下面通过一个简单的示例来说明 JavaFX 程序的基本结构。代码清单 5-16 中的 JavaFX 程序实现在界面上显示一个按钮和标签。当单击按钮时，标签上的文本会变为“Hello World!”。JavaFX 程序的主 Java 类需要继承自 `javafx.application.Application` 类。`Application` 类负责管理 JavaFX 程序的生命周期。在 `Application` 类中定义了与生命周期相关的方法。在 `Application` 类的对象被创建出来之后，会调用 `init` 方法进行 JavaFX 程序的初始化工作；接着 `start` 方法会被调用。运行时环境负责创建一个 `Stage` 类的对象，并作为 `start` 方法调用时的实际参数。在 `start` 方法中创建界面所需的 `Scene` 类的对象，并设置到 `Stage` 类的对象上，使该 `Scene` 类的对象作为程序的当前界面场景。`Scene` 类的对象包含了界面所需的其他组件。程序继续运行，直到程序的最后一个窗口被关闭或程序调用 `javafx.application.Platform` 类的 `exit` 方法显式地结束运行。最后调用 `stop` 方法来进行适当的资源释放和清理工作。程序只需要覆写 `Application` 类中的对应方法即可。在主 Java 类的 `main` 方法中调用 `Application` 类的静态方法 `launch` 来启动程序运行。

代码清单 5-16 JavaFX 应用程序的基本结构

```
public class JavaFXHelloWorld extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    public void start(Stage primaryStage) {
        primaryStage.setTitle("JavaFX Sample");
        Button button = new Button();
        button.setText("Button");
        final Label label = new Label();
        button.setOnAction(new EventHandler<ActionEvent>() {
            public void handle(ActionEvent event) {
                label.setText("Hello World!");
            }
        });
        BorderPane pane = new BorderPane();
        pane.setTop(button);
        pane.setBottom(label);
        Group root = new Group(pane);
        primaryStage.setScene(new Scene(root, 400, 300));
        primaryStage.show();
    }
}
```

JavaFX 也使用了与 AWT 和 Swing 相似的线程处理方式。在 JavaFX 程序启动时，运行时环境创建一个新的线程来执行 Application 类的对象的 start 方法。创建 Stage 和 Scene 类的对象的操作，以及对界面上节点的修改，都需要在这个线程中进行。这个线程被称为 JavaFX 的应用线程。通过 Platform 类的 isFxApplicationThread 方法可以判断当前线程是否为 JavaFX 的应用线程。在 JavaFX 程序中，同样可以使用工作线程在后台执行任务。调用 Platform 类的 runLater 方法可以在应用线程中进行界面的更新操作。这个方法的作用类似于前面提到的 EventQueue 和 SwingUtilities 类中的 invokeLater 方法。

在场景图中可以使用的节点都是 Node 类的对象。节点大致可分为三类：第一类是常见的用户界面控件，继承自 javafx.scene.control.Control 类；第二类是几何图形形状，继承自 javafx.scene.shape.Shape 类；第三类是与多媒体相关的类，包括显示图片的 javafx.scene.image.ImageView 类以及播放音频和视频文件的 javafx.scene.media.MediaView 类。JavaFX 还提供了对图表绘制的支持，包括 javafx.scene.chart.Chart 类及其子类。

JavaFX 2.0 中的图形渲染引擎 Prism 可以借助底层操作系统上的 DirectX 和 OpenGL 支持来进行界面渲染，并利用硬件平台提供的加速能力来提升界面绘制时的性能。这种方式要优于 Swing 在绘制界面时使用的基于 Java 2D 的软件渲染方式。因此 JavaFX 程序的性能相对于 Swing 有大幅度的提升。

5.7.2 变换

JavaFX 提供了对用户界面组件进行变换的支持。通过变换能力，可以改变组件的大小和位置。所有 Node 类的子类对象都可以进行变换。JavaFX 提供了四种不同的变换方式，定义在 javafx.scene.transform 包中。第一种变换方式是平移，即沿着 X、Y 和 Z 轴平行移动。平移变换由 javafx.scene.transform.Translate 类完成。在创建 Translate 类的对象时，要指定在 X、Y 和 Z 轴上的平移量。第二种变换方式是旋转，即以某个点作为中心，旋转指定的角度。旋转变换由 javafx.scene.transform.Rotate 类完成。在创建 Rotate 类的对象时，要指定旋转的角度、旋转中心点的坐标和旋转使用的坐标轴。第三种变换是缩放，即以某个点为中心，沿着 X、Y 和 Z 轴放大或缩小。缩放变换由 javafx.scene.transform.Scale 类表示。在创建 Scale 类的对象时，要指定缩放的中心点，以及在 X、Y 和 Z 轴上的缩放比例。第四种变换是切变，即对 X 和 Y 轴进行旋转，使 X 轴和 Y 轴不再互相垂直。切变变换由 javafx.scene.transform.Shear 类表示。在创建 Shear 类的对象时，要指定变换的起始点和在 X 和 Y 轴上的切变系数。

当需要进行变换时，可以通过 Node 类的 getTransforms 方法获取该对象上的已有的变换的集合，再向该集合中添加新的变换对象即可。代码清单 5-17 给出了一个对矩形进行四种变换的示例。

代码清单 5-17 对矩阵进行变换的示例

```
Rectangle rect = new Rectangle(200, 100);
rect.getTransforms().add(new Translate(100, 50));
rect.getTransforms().add(new Rotate(30, 100, 50));
rect.getTransforms().add(new Scale(2, 1.5));
rect.getTransforms().add(new Shear(0.5, 0));
```

5.7.3 动画效果

JavaFX 提供了对动画效果的支持，可以实现各种不同的动画效果。每个动画效果由多个帧组成。每一帧按照设定的频率进行播放。javafx.animation.Animation 类是所有动画效果的父类。通过 Animation 类的 play、pause 和 stop 方法可以开始、暂停和停止动画效果的运行。JavaFX 提供了两种类型的动画效果，一种是基于值的自动变化的，另外一种是基于时间线和关键帧的，这两种方式的区别在于动画效果中包含的帧的定义方式不同。

第一种方式需要继承 javafx.animation.Transition 类并覆写 interpolate 方法。在创建 Transition 类的子类对象之后，系统会根据动画效果的持续时间自动计算所需的帧数。在播放每一帧时，interpolate 方法会被调用，调用时的参数是当前的动画播放进度，范围在 0.0 到 1.0 之间。具体的进度由系统通过不同的插值算法来计算。javafx.animation.Interpolator 类的子类表示不同的插值计算算法。通过 Transition 类的 setInterpolator 方法

可以改变使用的插值算法。比较常用的是由 Interpolator.LINEAR 表示的线性插值方式。通过 Transition 类可以实现各种不同的动画效果，比如，实现一个矩形在界面上沿斜线运动，可以使用代码清单 5-18 中的实现方式。通过 setCycleDuration 方法把动画效果的周期设为 3 秒。在 interpolate 方法的实现中，根据当前的播放进度，通过修改 X 轴和 Y 轴坐标的方式把矩形移动到相应的位置上。方法 setCycleCount 用来设置动画周期的播放次数，Animation.INDEFINITE 的含义是无限执行。如果调用 setAutoReverse 方法并使用“true”作为参数，动画在下一个周期会改变播放的方向。

代码清单 5-18 基于值的自动变化的动画效果的示例

```
final Rectangle rect = new Rectangle(200, 100);
final double distanceX = 400;
final double distanceY = 300;
final Animation animation = new Transition() {
{
    setCycleDuration(Duration.millis(3000));
}
protected void interpolate(double frac) {
    rect.setX(distanceX * frac);
    rect.setY(distanceY * frac);
}
};
animation.setCycleCount(Animation.INDEFINITE);
animation.setAutoReverse(true);
animation.play();
```

JavaFX 中提供了一些实现常见动画效果的 Transition 类的子类。为了实现代码清单 5-18 中的动画效果，更好的做法是使用 javafx.animation.TranslateTransition 类。

相对于使用 Transition 类的做法，第二种做法允许开发人员使用更加灵活的方式为 JavaFX 中组件的任何属性添加动画效果，这种做法基于 javafx.animation.Timeline 类来实现。Timeline 类表示的是一条时间线，其中包含多个关键帧，即 javafx.animation.KeyFrame 类的对象。每个 KeyFrame 类的对象都对应时间线上的某个时间点，以及在这个时间点上属性应该具有的值。属性的值由 javafx.animation.KeyValue 类的对象来表示。每个 KeyFrame 类的对象可以包含多个 KeyValue 类的对象，这些对象表示该关键帧所定义的多个属性的值。在指明了关键帧之后，JavaFX 会自动使用插值算法来计算出实现整个动画效果所需的其他帧。这种实现方式的核心在于由开发人员定义整个动画效果中的关键点，由系统负责完成整个动画效果。

代码清单 5-19 给出了使用 Timeline 类实现动画效果的示例。在创建 Timeline 类的对象时，指定了两个关键帧。这两个关键帧对矩形的 X 轴坐标和填充颜色两个属性添加了动画效果。第一个关键帧对应的时间点是第 2 秒，此时对属性值的要求是，当动画效果运行到这一帧时，矩形的 X 轴坐标是 100，而填充颜色的值是 Color.RED。第二个关键帧对应的时间点是第 4 秒，此时对属性值的要求是，矩形的 X 轴坐标是 200，而填充

颜色的值是 Color.WHITE。由于没有指定时间点为第 0 秒时的关键帧，当动画开始播放时，会以矩形中这两个属性的当前值作为起始值。在整个动画播放过程中，其他帧上的这两个属性的值由系统自动计算。动画的播放效果是，界面上矩形的位置会沿着 X 轴从坐标 0 移动到 200，同时矩形的填充颜色先从白色逐渐变为红色，再逐渐变回白色。

代码清单 5-19 基于时间线和关键帧的动画效果的示例

```
final Rectangle rect = new Rectangle(200, 100, Color.WHITE);
final Animation animation = new Timeline(
    new KeyFrame(Duration.seconds(2), new KeyValue(rect.xProperty(), 100), new
        KeyValue(rect.fillProperty(), Color.RED)),
    new KeyFrame(Duration.seconds(4), new KeyValue(rect.xProperty(), 200), new
        KeyValue(rect.fillProperty(), Color.WHITE))
);
animation.setCycleCount(Animation.INDEFINITE);
animation.play();
```

5.7.4 FXML

有些 JavaFX 程序的界面比较复杂，包含非常多的元素。创建这类界面的代码非常繁琐，可读性差，也很难进行维护。JavaFX 允许开发人员使用 FXML 语言来描述程序的界面。FXML 使用的是 XML 的语法。可以从 FXML 文档中创建出用户界面。使用 FXML 的好处是界面上组件之间的层次结构非常清晰，易于更新和维护。不熟悉 Java 的界面设计人员也可以用 FXML 来描述界面。

在 JavaFX 程序中，使用 FXML 有三个步骤。第一步是使用 FXML 的语法格式来描述程序的用户界面；第二步是通过 javafx.fxml.FXMLLoader 类加载一个 FXML 文档，并得到相应的用户界面的对象；第三步是在 JavaFX 程序中直接使用得到的组件对象。

FXML 的基本语法比较简单。FXML 中的元素可以表示对象实例、属性或代码块。对于对象实例，元素的名称是对应的 Java 类的名称，而元素的属性用来设置对象的属性值。对象的某些属性比较复杂，不能用简单的 XML 属性来描述，只能用子元素来表示。代码清单 5-20 给出了一个用 FXML 描述 JavaFX 程序界面的示例。处理指令“import”类似 Java 中的 import 语句。FXML 文档中的元素“<VBox>”、“<HBox>”、“<Button>”和“<Label>”等用来创建对应的对象。简单的属性，如 VBox 类的 spacing，可以直接在 XML 元素的属性上设置；复杂的属性，如 VBox 类的对象所包含的子组件 children，可以通过子元素来设置。

代码清单 5-20 使用 FXML 描述 JavaFX 程序界面的示例

```
<?xml version="1.0" encoding="UTF-8"?>

<?import java.lang.*?>
<?import javafx.scene.*?>
```

```

<?import javafx.collections.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<VBox id="main" spacing="20" prefHeight="300" prefWidth="400" xmlns:fx="http://
javafx.com/fxml" fx:controller="javafx.fxml.Form">
<children>
    <HBox spacing="10">
        <children>
            <Label text="Name:" />
            <TextField fx:id="name" promptText="Enter your name" />
        </children>
    </HBox>
    <HBox spacing="10">
        <children>
            <Label text="Gender:" />
            <ListView fx:id="gender">
                <items>
                    <FXCollections fx:factory="observableArrayList">
                        <String fx:value="Male"/>
                        <String fx:value="Female"/>
                    </FXCollections>
                </items>
            </ListView>
        </children>
    </HBox>
    <HBox>
        <children>
            <Button id="button" text="Say" onAction="#say" fx:id="button" />
            <Label id="message" fx:id="message" />
        </children>
    </HBox>
</children>
</VBox>

```

每个 FXML 文档可以与一个 Java 对象关联起来，这个 Java 对象被称为 FXML 文档的控制器。通过 FXML 文档根元素的“`fx:controller`”属性来声明控制器的 Java 类名。在 FXML 文档中可以引用控制器对象中的方法。具体的格式是在方法名称上加上“#”作为前缀。代码清单 5-20 中的“`<Button>`”元素的“`onAction`”属性引用了控制器对象中的“`say`”方法。

创建出 FXML 文档之后，可以在 JavaFX 程序中通过 `FXMLLoader` 类来使用它。代码清单 5-21 给出了相关的示例。`FXMLLoader` 类的 `load` 方法可以从不同的来源加载 FXML 文档，并创建出 FXML 文档中声明的组件对象。这个创建出来的组件对象可以在 JavaFX 程序中自由使用。

代码清单 5-21 在 JavaFX 程序中加载 FXML 文档的示例

```

public class JavaFxFXML extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    public void start(Stage stage) throws Exception {
        Parent root = FXMLLoader.load(getClass().getResource("Form.fxml"));
        stage.setScene(new Scene(root));
        stage.show();
    }
}

```

5.7.5 CSS 外观描述

用户对桌面应用外观的要求越来越高，很多应用都提供了更换界面皮肤的功能来满足用户的需求。从前面对 AWT 和 Swing 的介绍可以知道，AWT 无法改变组件的外观，而 Swing 则需要通过编写 Java 代码的方式来实现。JavaFX 采用了一种更加先进的做法，即允许使用 CSS 来改变用户界面的外观。引入 CSS 支持所带来的影响是巨大的：首先，对用户界面的修改变得很容易，只需要修改 CSS 代码即可，不需要重新进行编译；其次，熟悉 CSS 用户界面的设计人员可以直接修改程序的外观，而不需要依靠 Java 开发人员来编写 Java 代码；最后，JavaFX 和 Web 应用的外观样式声明可以混杂在一起使用。

JavaFX 中使用的 CSS 语法基于 W3C 的 CSS 2.1 规范，并添加了部分 CSS 3 的内容。鉴于 JavaFX 程序本身的特性，所使用的 CSS 语法中某些属性的含义不同于在 Web 应用中的含义。JavaFX 程序使用的 CSS 样式表中的所有属性都以“-fx-”作为前缀，以区别于一般的 CSS 属性。

有两种方式来使用 CSS 设置程序的外观样式。第一种方式是使用 Node 类的 `setStyle` 方法直接设置节点的 CSS 样式声明，这种做法适合于修改单个具体的节点的样式，比如，当需要把某个矩形对象 `rect` 的填充颜色设为红色，可以使用 “`rect.setStyle("-fx-fill: #FF0000");`”。第二种方式是把样式声明提取到单独的 CSS 文件中，并通过 Scene 类的对象来应用 CSS 文件。如代码清单 5-22 所示，在 Scene 类的对象中添加了新的 CSS 文件 “main.css”，在 “main.css” 中定义了相关的样式规则。为了在 CSS 文件中能够引用界面上的组件，一般需要使用 `setId` 方法为组件指定一个标识符。“main.css” 中的样式声明是 “`#rect { -fx-fill: #ff0000; }`”，把标识符为 “rect”的节点的填充颜色设置为红色。

代码清单 5-22 使用 CSS 描述 JavaFX 程序外观的示例

```

Group root = new Group();
Rectangle rect = new Rectangle(100, 50);
rect.setId("rect");

```

```
root.getChildren().add(rect);
Scene scene = new Scene(root, 300, 250);
scene.getStylesheets().add("main.css");
```

如果在 CSS 文件中定义了 CSS 类，可以在代码中为组件添加 CSS 类，例如，“rect.getStyleClass().add("myRect")”为 rect 对象添加了名称为“myRect”的 CSS 类。

5.7.6 Web 引擎与网页显示

JavaFX 提供了用户界面组件来显示 Web 页面及对页面的内容进行操纵。JavaFX 使用的网页显示引擎基于 Webkit 内核，支持 HTML5 的新特性。在 JavaFX 程序中可以访问 Web 引擎组件中显示的网页的 DOM 结构和执行 JavaScript 代码。这个组件相当于一个内嵌的浏览器，它使一种新的结合 Java 和 HTML 的部署方式成为可能。具体的做法是：程序的主体内容是使用 HTML、JavaScript 和 CSS 来编写的 Web 页面，并且可以利用 HTML5 的新特性。程序的外层由 JavaFX 来编写，通过一个 Web 引擎组件来显示作为主体的 Web 页面。用户在使用时运行的是 JavaFX 程序。这样的好处是在编写 Web 应用时限制了用户使用的浏览器类型，不需要过多考虑浏览器兼容性问题，同时可以利用 Webkit 内核的新特性。越来越多的 Web 应用使用了 HTML5 的新特性。如果用户直接通过浏览器来访问，此时的浏览器可能并不支持这些新特性。这一方面会影响用户体验，另外一方面也使程序的实现变得更加复杂。采用这种 JavaFX 与 HTML 集成的部署方式之后，用户只需要运行 JavaFX 程序即可，不需要考虑浏览器相关的问题。

Web 引擎组件由 javafx.scene.web.WebEngine 和 javafx.scene.web.WebView 类来表示。WebEngine 类负责管理网页以及与网页中的内容进行交互；WebView 类则负责显示网页的内容，可以被当做场景上的普通节点来使用。代码清单 5-23 给出了使用 Web 引擎组件进行网页自动化测试的一个示例。待测试的网页是一个用户登录页面。用户输入用户名和密码之后，如果验证成功，会跳转到相应的页面。通过 Web 引擎组件可以对这个测试过程进行自动化。先创建一个 WebView 类的对象，并将其添加到界面场景中。然后通过 WebView 类的对象的 getEngine 方法可以获取所关联的 WebEngine 类的对象。WebEngine 类的 load 方法可以用来加载一个网页并显示。网页的加载过程是异步执行的。通过 WebEngine 类的 getLoadWorker 方法可以获取到一个用来监视加载进度的 javafx.concurrent.Worker 接口的实现对象。Worker 接口的 workDone 属性表示当前的加载进度。在该属性上添加一个变化监听器，可以监视当前的网页加载进度。网页加载完成之后，通过 WebEngine 类的 getDocument 方法可以获取表示当前页面 DOM 结构的 org.w3c.dom.Document 接口的实现对象。可以使用该 Document 接口的实现对象对网页内容进行操作。在代码清单 5-23 中设置了登录页面上两个 <input> 元素的值，用来模拟输入用户名和密码。WebEngine 类的 executeScript 方法可以用来执行一段 JavaScript 代码。代码清单 5-23 通过 JavaScript 代码的方式来提交表单。如果用户验证正确，提交表

单后，应该跳转到一个新的页面。WebEngine 类的 location 属性可以用来监视内嵌浏览器中网页地址的变化。可以通过这种方式来验证网页是否发生了跳转。

代码清单 5-23 使用 Web 引擎组件进行网页自动化测试的示例

```
public class JavaFxWeb extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    public void start(Stage primaryStage) {
        primaryStage.setTitle("Web Test");
        WebView view = new WebView();
        primaryStage.setScene(new Scene(view, 800, 600));
        primaryStage.show();
        final WebEngine engine = view.getEngine();
        String url = "http://localhost/test.html";
        engine.load(url);
        final Worker worker = engine.getLoadWorker();
        worker.workDoneProperty().addListener(new ChangeListener<Number>() {
            public void changed(ObservableValue<? extends Number> observable,
                Number oldValue, Number newValue) {
                if (newValue.intValue() == 100) {
                    worker.workDoneProperty().removeListener(this);
                    engine.locationProperty().addListener(new ChangeListener-
                        <String>() {
                        public void changed(ObservableValue<? extends String>
                            observable, String oldValue, String newValue) {
                            System.out.println(newValue); // 检查地址是否正确
                        }
                    });
                }
                Document doc = engine.getDocument();
                Element nameElem = doc.getElementById("name");
                nameElem.setAttribute("value", "alex");
                Element passwordElem = doc.getElementById("password");
                passwordElem.setAttribute("value", "password");
                String script = "document.getElementById('form').submit();";
                engine.executeScript(script);
            }
        });
    }
}
```

5.8 使用案例

本章的案例主要介绍如何把 Swing 和 JavaFX 集成起来使用。在目前的 Java 桌面应用中，除了使用 SWT 之外，绝大部分是利用 Swing 来开发的。在使用 JavaFX 开发新

的桌面应用时，免不了要与遗留的 Swing 应用进行整合。在一个已有的 Swing 程序中嵌入 JavaFX 组件是一件比较容易的事情。JavaFX 提供了 `javafx.embed.swing.JFXPanel` 类，可以显示 JavaFX 中的场景。JFXPanel 类继承自 `JComponent` 类，因此可以在 Swing 程序中使用，同时可以通过 `setScene` 方法来设置 JavaFX 中表示场景的 `Scene` 类的对象。

在集成 Swing 和 JavaFX 时，要注意界面更新线程的使用。Swing 中的界面更新操作都需要在事件分发线程中进行，而 JavaFX 中的界面更新操作需要在 JavaFX 应用线程中进行。这两个线程是不一样的。需要使用 `SwingUtilities` 类的 `invokeLater` 方法和 `Platform` 类的 `runLater` 方法来在正确的线程中执行界面更新操作。

本案例开发的是一个简单的 MP3 播放器，实现根据 MP3 文件的 URL 来进行播放的功能。由于 JavaFX 提供了良好的对音频文件播放的支持，因此音乐播放这部分工作由 JavaFX 来实现，剩下的界面显示由 Swing 来实现。代码清单 5-24 给出了案例的完整实现。方法 `initAndShowUI` 用来创建整个 Swing 界面，对该方法的调用需要封装在 `SwingUtilities` 类的 `invokeLater` 方法的调用中。方法 `initPanel` 用来创建 `JFXPanel` 类的对象中包含的界面组件，对该方法的调用需要封装在 `Platform` 类的 `runLater` 方法的调用中。

代码清单 5-24 集成 JavaFX 和 Swing 的 MP3 播放器

```

public class JavafxPlayer {
    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                initAndShowUI();
            }
        });
    }

    private static void initAndShowUI() {
        JFrame frame = new JFrame("Music Player");
        final JFXPanel fxPanel = new JFXPanel();
        frame.add(fxPanel, BorderLayout.NORTH);
        final JLabel label = new JLabel();
        frame.add(label, BorderLayout.CENTER);
        frame.setSize(400, 200);
        frame.setVisible(true);
        Platform.runLater(new Runnable() {
            public void run() {
                initPanel(fxPanel, label);
            }
        });
    }

    private static void initPanel(JFXPanel fxPanel, final JLabel label) {
        HBox box = new HBox(10);
        Button play = new Button("Play");
        play.setMinWidth(100);
    }
}

```

```
Media media = new Media("http://localhost/test.mp3");
final MediaPlayer player = new MediaPlayer(media);
play.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent t) {
        player.play();
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                label.setText(player.getMedia().getSource());
            }
        });
    }
});
box.getChildren().addAll(play);
fxPanel.setScene(new Scene(box, 400, 50));
```

5.9 小结

使用 Java 开发桌面应用并不是一件很困难的事情。Java 语言本身的特性及用户界面库的设计方式，使开发人员上手更容易，要精通也并不困难。本章的内容主要围绕 Java 7 中 AWT 和 Swing 的新特性展开，同时介绍了 AWT 和 Swing 中一些比较复杂的概念，包括事件分发线程、界面绘制和可插拔式外观样式等。理解这些概念可以避免一些开发中会出现的错误。而对于 AWT 和 Swing 中组件的具体使用，并没有进行详细介绍，相关的组件说明可以从 API 文档中找到。本章着重介绍了 JavaFX 2.0。作为 Java 平台上桌面应用的未来发展方向，了解 JavaFX 是很有必要的。

对于基于 Java 平台的桌面应用开发，开发人员可能会希望借助框架来简化具体的工作。实际上，早在 2006 年，相关框架的设计与实现工作就已经开始了。这个被称为 Swing 应用框架（Swing application framework）的框架由 JSR 296 规范来定义。原来的计划是 JSR 296 会成为 Java 7 的一部分，后来由于时间原因而被迫放弃。JSR 296 有可能会最终出现在 Java 8 中。因此，在 Java 7 中并不存在标准的 Swing 应用开发框架。开发人员只能从已有的一些框架中做出选择。一些不错的候选框架包括：Better Swing Application Framework (BSAF)[⊖]、Guice Utilities & Tools Set (GUTS)[⊖] 和 Spring Rich Client[⊖]。

⊖ <http://kenai.com/projects/bsaf>.

≡ <http://kenai.com/projects/guts>.

④ <http://spring-rich-c.sourceforge.net/>.

第 6 章 Java 7 其他重要更新

本书前面几章从不同的方面介绍了 Java 7 中比较重要的更新，这些更新的介绍不是独立进行，而是围绕 Java 平台上的某个特定主题展开。在对这些主题进行介绍的时候，既介绍了 Java 7 之前已有的概念，又介绍了 Java 7 的新特性，目的在于让读者全面了解一个主题各个方面相关知识。本章则以 Java 7 中的其他重要更新为主要内容，同时也包括了其他相对较小却值得关注的改动。

在内容组织方式上，本章相对比较松散，每一节围绕一个小的主题展开。对于某些小改动，合并在一个小节里面进行介绍。在具体的内容上，以 Java 7 的新特性为主，并适当介绍相关的背景知识。

6.1 关系数据库访问

关系数据库访问一直是 Java 平台的重要功能，尤其对 Web 应用开发来说，大多数 Web 应用都是由关系数据库来驱动的。Java 平台的数据库访问方式由 JDBC（Java Data Base Connectivity）规范来定义。JDBC 规范本身也在不断更新，以适应数据库技术的发展，同时满足开发人员的使用需求。Java 7 在数据库访问方面的重要更新是增加了对 JDBC 4.1 规范的支持，Java 6 支持的仅是 JDBC 4.0 规范。相对于 JDBC 4.0 来说，JDBC 4.1 规范又引入了一些新的特性，这些特性都可以在 Java 7 中使用。需要注意的是，不同数据库实现的 JDBC 驱动对 JDBC 4.1 规范的支持程度是不相同的。特定的数据库实现有可能暂时还不支持某些新特性。JDBC 4.1 规范所更新的特性在 `java.sql` 和 `javax.sql` 包中都有，下面会分别进行介绍。

6.1.1 使用 try-with-resources 语句

在第 1 章介绍了 Java 7 通过新增的 `try-with-resources` 语句来管理 Java 平台上的各种资源。与数据库访问相关的各种资源，包括数据库连接、查询语句和结果集等，都可以用 `try-with-resources` 语句来进行管理。通过 `try-with-resources` 语句可以去掉数据库操作时对 `close` 方法的调用，大大降低了代码的复杂性。在 Java 7 中，`java.sql` 包中的 `java.sql.Connection`、`java.sql.Statement` 和 `java.sql.ResultSet` 接口都继承了 `java.lang.AutoCloseable` 接口，以支持由 `try-with-resources` 语句来管理。代码清单 6-1 给出了使用 `try-with-resources` 语句进行数据库操作的示例。可以把对 `Connection`、`Statement` 和 `ResultSet` 的创建都封装在 `try-with-resources` 语句中，这样就不用考虑这些对象的关闭操作。

代码清单 6-1 用 try-with-resources 语句进行数据库操作的示例

```
public void dbOperation() throws SQLException {
    try (Connection connection = DriverManager
        .getConnection("jdbc:derby://localhost/java7book");
        Statement stmt = connection.createStatement();
        ResultSet rs = stmt.executeQuery("SELECT * FROM book")) {
        while (rs.next()) {
            System.out.println(rs.getString("name"));
        }
    }
}
```

6.1.2 数据库查询的默认模式

大部分关系数据库系统都支持为数据库中包含的表和其他对象创建一个额外的名称空间，即模式（schema）。一个模式中可以包含表、视图以及其他对象。不同模式中可以有名称相同的表或视图，而同一模式中不允许有名称相同的对象存在。通过 SQL 语句“CREATE SCHEMA”可以创建新的模式。当访问某个模式中包含的表或视图等对象时，需要使用模式名称作为前缀来访问，否则无法找到相应的对象。比如，对于一个名为“DEMO_SCHEMA”的模式中的“author”表，在 SQL 语句中应该使用“DEMO_SCHEMA.author”来引用。如果每次都要加上模式名称作为前缀，那么使用起来会比较麻烦。JDBC 4.1 为 Connection 接口添加了一对新的方法 getSchema 和 setSchema，用来获取和设置数据库操作时使用的默认模式名称。当通过 setSchema 进行设置之后，在 SQL 语句中就不再需要使用模式名称作为前缀了。代码清单 6-2 给出了使用 setSchema 方法的示例。一般来说，如果希望使用 setSchema 方法，那么应该在从 Connection 接口中创建出来的 Statement 对象被使用之前进行设置，否则可能会造成默认的模式无法生效。

代码清单 6-2 setSchema 方法的使用示例

```
public void setSchema() throws SQLException {
    try (Connection connection = DriverManager
        .getConnection("jdbc:derby://localhost/java7book")) {
        connection.setSchema("DEMO_SCHEMA");
        try (Statement stmt = connection.createStatement();
            ResultSet rs = stmt.executeQuery("SELECT * FROM author")) {
            while (rs.next()) {
                System.out.println(rs.getString("name"));
            }
        }
    }
}
```

6.1.3 数据库连接超时时间与终止

在使用数据库连接时会遇到的一个问题是，网络连接出现问题造成数据库连接中断。当数据库连接中断时，可能会造成发出数据库操作命令的线程在较长时间内处于等待状态。具体的等待时间取决于 TCP 连接的超时时间设置。这个超时时间一般会长达几分钟。也就是说，在数据库操作命令发出之后，可能需要等待几分钟才能得知数据库连接已经中断了。对于一个数据库应用来说，几分钟的等待时间过长，为此，JDBC 4.1 添加了对数据库连接超时的处理方式，主要是在 Connection 接口中新增了 setNetworkTimeout 和 abort 两个方法。

Connection 接口的 setNetworkTimeout 方法用来设置通过此数据库连接进行数据库操作时的超时等待时间。如果远程数据库没有在给定的时间内返回操作结果，那么会认为该连接已经关闭，无法再继续使用，处于等待状态的数据库操作方法则会抛出 SQLException 异常来表示这种超时的情况。对一个 Connection 接口的实现对象设置超时等待时间之后，会影响到从该 Connection 接口的实现对象中创建的其他对象，主要包括 Statement 和 java.sql.PreparedStatement 接口的实现对象。通过 Statement 和 PreparedStatement 接口的实现对象执行的查询操作也适用于 Connection 接口的实现对象上设置的超时等待时间。可以多次调用 setNetworkTimeout 方法，以对不同的情况设置不同的超时时间。对于同一个 Connection 接口的实现对象，如果预计某些查询操作比较耗时，可以把超时等待时间设置为一个较大的值，等操作完成之后，再恢复为一个对大多数操作都适用的较小值。

与 setNetworkTimeout 相关的方法 abort 用来强制关闭一个数据库连接。当调用一个 Connection 接口的实现对象的 abort 方法时，这个数据库连接会被标记为关闭状态，同时与该连接相关的资源都会被释放，另外，当前正在使用该连接的方法会抛出 SQLException 异常。从作用上讲，abort 方法类似于 Connection 接口中已有的 close 方法，但是两者的使用场景不同：close 方法一般是由 Connection 接口的使用者来调用的，当一个使用者完成数据库操作之后，可以通过 close 方法来关闭此数据库连接；而 abort 方法一般由数据库连接的管理者来调用，如果发生了前面提到的由于网络问题造成数据库连接中断的情况，连接的管理者在检测到问题发生之后，可以调用 abort 方法来强制终止此连接，该连接的使用者也不需要等待数据库操作的完成即可进入到 SQLException 异常的处理阶段。

setNetworkTimeout 和 abort 方法都接收一个 java.util.concurrent.Executor 接口的实现对象作为参数。这个 Executor 接口的实现对象用来执行方法调用中可能出现的相关任务。对于 abort 方法来说，在终止数据库连接的过程中需要释放相关的资源。释放资源的相关任务由该 Executor 接口的实现对象来执行。当 abort 方法返回之后，Executor 接口的实现对象可能仍在继续执行相关的任务。代码清单 6-3 给出了 abort 方法的使用示例。为了查看在 abort 方法调用过程中执行的相关任务，这里使用了一个自定义的 java.

util.concurrent.ThreadPoolExecutor 类的实现。在任务被执行之前，会在控制台输出相关的提示信息。在运行过程中可以发现，在调用 abort 方法之后，有新的任务被添加到 Executor 接口的实现对象中执行。

代码清单 6-3 abort 方法的使用示例

```

public class AbortConnection {
    public void abortConnection() throws SQLException {
        Connection connection = DriverManager
            .getConnection("jdbc:derby://localhost/java7book");
        ThreadPoolExecutor executor = new DebugExecutorService(2, 10, 60,
            TimeUnit.SECONDS, new LinkedBlockingQueue<Runnable>());
        connection.abort(executor);
        executor.shutdown();
        try {
            executor.awaitTermination(5, TimeUnit.MINUTES);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    private static class DebugExecutorService extends ThreadPoolExecutor {
        public DebugExecutorService(int corePoolSize, int maximumPoolSize,
            long keepAliveTime, TimeUnit unit,
            BlockingQueue<Runnable> workQueue) {
            super(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue);
        }

        public void beforeExecute(Thread t, Runnable r) {
            System.out.println("清理任务：" + r.getClass());
            super.beforeExecute(t, r);
        }
    }
}

```

6.1.4 语句自动关闭

在 JDBC 4.1 之前，当使用一个 Statement 接口的实现对象进行查询，得到表示结果集的 ResultSet 接口的实现对象并完成处理之后，需要显式地通过 close 方法来关闭 ResultSet 和 Statement 接口的实现对象以释放资源。使用 Java 7 的 try-with-resources 语句可以简化这个操作，而更好的办法是使用 JDBC 4.1 为 Statement 接口添加的自动关闭功能。在调用了 Statement 接口的 closeOnCompletion 方法之后，表明当依赖此 Statement 接口的实现对象的所有结果集都被关闭之后，该 Statement 接口的实现对象也被自动关闭。通过这种方式，开发人员只需显式地关闭结果集即可，不需要考虑 Statement 接口的实现对象本身的关闭问题。这个新特性在 Statement 接口的实现对象作

为参数在多个方法中传递时尤为实用。当代码中的多个地方都使用了相同的 Statement 接口的实现对象来打开结果集时，以前需要设计好由哪个方法来关闭该 Statement 接口的实现对象，而使用这个自动关闭功能之后，每个方法只需要关闭该方法内部打开的结果集即可，不需要考虑 Statement 接口的实现对象自身的关闭问题。

6.1.5 RowSet 实现提供者

javax.sql.RowSet 接口是 JDBC 2.0 引入的表格式数据的抽象表示形式。RowSet 接口继承自 ResultSet 接口，并在 ResultSet 接口的基础上添加了属性设置和变化通知相关的功能。RowSet 接口是符合 JavaBeans 组件规范的，可以与其他 JavaBeans 组件协同使用。RowSet 接口的使用者并不需要显式地管理数据库连接，只需要把数据库相关的连接信息以属性的方式设置到 RowSet 接口的实现对象中即可。RowSet 接口的实现会负责处理数据库连接的相关工作。RowSet 接口有 5 个不同的子接口，每个子接口实现所适应的场景不同。每个子接口对应的具体实现类也有不少，分别来自不同的提供者。在 Java 7 之前并没有一个规范的方式来管理 RowSet 接口的实现对象的创建，开发人员需要直接根据实现类的类名来创建新的 RowSet 接口的实现对象。这种直接依赖具体类而不是接口的做法，不利于代码的移植和维护。

Java 7 对 RowSet 接口的实现对象的创建做了更新，采用了 Java 标准的服务提供者接口（Service Provider Interface，SPI）机制。使用者通过工厂方法来创建具体的 RowSet 接口的实现对象，而工厂对象本身由 RowSet 实现的提供者注册到 Java 平台上。具体的工厂对象的查找和创建是由 javax.sql.rowset.RowSetProvider 类的静态方法来完成的，具体的工厂对象则实现 javax.sql.rowset.RowSetFactory 接口。RowSetFactory 接口提供了 5 种不同 RowSet 接口实现的创建方法。代码清单 6-4 给出了使用 javax.sql.rowset.JdbcRowSet 实现的示例，从中可以看到 RowSet 接口的基本用法，数据库连接字符串和 SQL 语句都是以属性的方式来进行设置的。

代码清单 6-4 使用工厂方法创建 RowSet 接口的实现对象的示例

```
public void useRowSet() throws SQLException {
    RowSetFactory rsFactory = RowSetProvider.newFactory();
    try (JdbcRowSet jrs = rsFactory.createJdbcRowSet()) {
        jrs.setUrl("jdbc:derby://localhost/java7book");
        jrs.setCommand("SELECT * FROM book");
        jrs.execute();
        jrs.absolute(1);
        jrs.updateString("name", "New book");
        jrs.updateRow();
    }
}
```

除了上面介绍的几个比较重要的更新之外，JDBC 4.1 还有一些比较小的更新。这些

小更新包括：在使用 ResultSet 中的 getObject 方法时，可以直接把结果类型的 Java 类作为参数传递进去，而不需要在调用之后再进行强制类型转换。比如，之前的调用方式是 “(String) rs.getObject(1)”，新的调用方式是 “rs.getObject(1, String.class)”；表示数据库中元数据的 java.sql.DatabaseMetaData 接口中新增了 getPseudoColumns 方法来获取数据库表中包含的伪列（pseudo column）和隐藏列（hidden column）的元数据。伪列指的是不在数据库表结构中定义但可以在查询中获取的列。不同数据库实现所支持的伪列是不同的，比如 Oracle 数据库允许在查询中使用伪列 “ROWNUM” 来获取当前行的行号，类似的伪列还有获取当前日期和时间戳的 “SYSDATE” 和 “SYSTIMESTAMP”。隐藏列通常用来保存一些与数据库表相关的内部信息。getPseudoColumns 方法的返回值是 ResultSet 接口的实现对象，其中的每一行都包含了查找到的伪列和隐藏列的名称、数据类型、大小等信息。如果通过 Statement 接口的 setQueryTimeout 方法设置了数据库查询的超时等待时间，那么当操作超时的时候，会抛出更加具体的 java.sql.SQLTimeoutException 异常，而不是通用的 SQLException 异常。

在使用 JDBC 4.1 时，需要检测所增加的新特性是否已经被关系数据库实现支持。不同的关系数据库实现在对 JDBC 规范的支持上可能有所不同。在使用 JDBC 4.1 的新特性之前，先查看一下所使用的数据库和驱动的相关信息，以确定对 JDBC 4.1 规范的支持程度。

6.2 java.lang 包的更新

Java 7 对 java.lang 包的更新也比较多，主要涉及基本类型的包装类、进程使用和线程类。下面分别具体介绍。

6.2.1 基本类型的包装类

Java 7 对基本类型的包装类做了一些更新，以更好地满足日常的开发需求。第一个更新是在基本类型的比较方面， Boolean、 Byte、 Short、 Integer、 Long 和 Character 类都添加了一个比较两个基本类型值的静态 compare 方法，比如 Long 类的 compare 方法可以用来比较两个 long 类型的值。这个 compare 方法更多是作为“语法糖衣”而存在的，可以简化进行基本类型数值比较时的代码。在 Java 7 之前，如果需要对两个 int 数值 x 和 y 进行比较，一般的做法是使用代码 “Integer.value(x).compareTo(Integer.value(y))”，而在 Java 7 中可以直接使用 “Integer.compare(x, y)”。

对于字符串内部化（string interning）技术，开发人员可能并不陌生，采用这种技术是常见的优化策略，可以提高字符串比较时的性能，是一种典型的空间换时间的做法。Java 也采用了这种技术。在 Java 中，包含相同字符的字符串字面量引用的是相同的内部对象。String 类也提供了 intern 方法来返回与当前字符串内容相同但已经被包含在内部缓存中的对象引用。在对被内部缓存的字符串进行比较时，可以直接使用 “==” 操作

符，而不需要用更加耗时的 equals 方法。比如，在代码清单 6-5 中，第一个字符串比较的结果为 true，第二个字符串比较的结果为 false，第三个字符串比较的结果为 true。第二个字符串比较如果使用 equals 方法进行，那么结果也是 true。

代码清单 6-5 字符串内部化的示例

```
public void stringIntern() {
    boolean value1 = "Hello" == "Hello";
    boolean value2 = (new String("Hello") == "Hello");
    boolean value3 = (new String("Hello").intern() == "Hello");
}
```

Java 7 把这种内部化机制扩大到了 -128 到 127 之间的数字。根据 Java 语言规范，对于 -128 到 127 范围内的 short 类型和 int 类型，以及 \u0000 到 \u007f 范围内的 char 类型，它们对应的包装类对象始终指向相同的对象，即通过 “==” 进行判断时的结果为 true。为了满足这个要求，Byte、Short、Integer 类的 valueOf 方法对于 -128 到 127 范围内的值，以及 Character 类的 valueOf 方法对于 0 到 127 范围内的值，都会返回内部缓存的对象。如代码清单 6-6 所示，在使用了内部缓存的情况下，由于第一个比较操作的数值是在 -128 到 127 之间，因此 Integer 类的 valueOf 方法返回的是同一个缓存对象，value1 的值为 true；而第二个比较操作的数值超出了默认的缓存范围，因此 valueOf 方法返回的是两个不同的对象，value2 的值为 false。

代码清单 6-6 Integer 类的内部化的示例

```
public void numberCache() {
    boolean value1 = Integer.valueOf(3) == Integer.valueOf(3);
    boolean value2 = Integer.valueOf(129) == Integer.valueOf(129);
}
```

如果希望缓存更多的值，可以通过 Java 虚拟机启动参数 “java.lang.Integer.IntegerCache.high” 来进行设置。例如，使用 “-Djava.lang.Integer.IntegerCache.high=256” 之后，数值缓存的范围就变成了 -128 到 256，再次运行代码清单 6-6 会发现，value2 的值变成 true，因为 129 处于修改之后的缓存范围之内。

6.2.2 进程使用

在 Java 程序中可能需要调用底层操作系统上的其他程序，Java 标准 API 提供了创建底层操作系统上运行的进程的能力，只需要传入正确的命令和相关的参数，就可以启动一个进程。在进程启动之后，可以从 Java 程序向进程提供输入数据，以及读取进程运行过程中产生的输出数据。对于在 Java 程序中启动其他进程这个任务来说，最重要的是输入和输出的处理。通常的做法是把 Java 程序的内部运行结果作为输入传递给一个新创建的进程，然后等待进程执行完成。在得到进程输出的运行结果之后，再继续下面的处

理。通过这种方式，底层操作系统上的其他进程可以很好地与 Java 程序集成起来。

在 Java 7 之前，对进程的输入和输出进行处理的方式比较有限，只支持管道式的方式。进程的输入对 Java 程序来说是一个输出流，程序向这个输出流中写入的数据会通过管道传递给进程。同样的，进程的输出对于 Java 程序来说是一个输入流，通过读取此输入流的内容获得进程的输出。标准的创建新进程的过程是使用 `java.lang.ProcessBuilder` 类来设置新进程的属性，然后通过 `start` 方法来启动进程的执行。`ProcessBuilder` 类的 `start` 方法的返回值是一个表示进程的 `java.lang.Process` 类的对象。通过 `Process` 类的 `getOutputStream` 方法可以得到向进程写入数据的输出流，而通过 `getInputStream` 和 `getErrorStream` 方法可以分别得到包含进程正常执行和出错时输出内容的输入流。代码清单 6-7 给出了创建进程的示例。示例中启动了 Windows 上的命令行工具来执行“`netstat -a`”命令，并把结果保存到一个文件中。

代码清单 6-7 创建进程的示例

```
public void startProcessNormal() throws IOException {
    ProcessBuilder pb =
        new ProcessBuilder("cmd.exe", "/c", "netstat", "-a");
    Process process = pb.start();
    InputStream input = process.getInputStream();
    Files.copy(input, Paths.get("netstat.txt"), StandardCopyOption.REPLACE_
        EXISTING);
}
```

使用管道的方式在某些情况下显得不够灵活，因此 Java 7 对进程的输入和输出处理进行了更新，增加了另外的两种处理方式。第一种是继承式，即新创建进程的输入和输出与当前的 Java 进程相同。第二种是基于文件式，即把文件作为进程输入的来源和输出的目的地。代码清单 6-8 给出了继承式的一个示例，其中启动的进程通过 Windows 上的命令行工具来执行 `dir` 命令，通过 `ProcessBuilder` 类的 `redirectOutput` 方法把进程的输出设置为继承自父进程，运行的结果就是 `dir` 命令的输出内容，会显示在 Java 程序默认的输出控制台中。

代码清单 6-8 进程的输入和输出的继承式处理方式的示例

```
public void dir() throws IOException {
    ProcessBuilder pb =
        new ProcessBuilder("cmd.exe", "/c", "dir");
    pb.redirectOutput(Redirect.INHERIT);
    pb.start();
}
```

如果希望把进程的输入或输出改为文件，那么可以使用 `ProcessBuilder` 类中的 `redirectInput` 和 `redirectOut` 方法的其他重载形式。代码清单 6-9 给出了基于文件式处理方式的示例。示例中通过一个文件来保存进程的输出内容，只需要把一个 `java.io.File` 类

的对象作为 redirectOutput 方法的参数即可。这种做法在实现上相当于通过管道方式读取输入流来获取进程的输出，然后将其写入一个文件中。不过以标准 API 的方式给出的实现，显然比程序自己来实现要好。

代码清单 6-9 进程的输入和输出的文件式处理方式的示例

```
public void listProcesses() throws IOException {
    ProcessBuilder pb =
        new ProcessBuilder("wmic", "process");
    File output = Paths.get("tasks.txt").toFile();
    pb.redirectOutput(output);
    pb.start();
}
```

从 API 的角度来说，Java 7 通过新增的 ProcessBuilder.Redirect 类对进程的输入和输出重定向方式进行了统一。ProcessBuilder.Redirect 类提供了两种直接使用的重定向类型，一种是 Java 7 之前就有的管道式，用 ProcessBuilder.Redirect.PIPE 来表示；另一种是前面介绍的继承式，用 ProcessBuilder.Redirect.INHERIT 来表示。其余 3 种方式都是与文件相关的，在使用时都需要一个 File 类的对象作为参数。ProcessBuilder.Redirect.from 表示从一个文件中读取内容作为输入，ProcessBuilder.Redirect.to 表示把输出写入一个文件中，ProcessBuilder.Redirect.appendTo 表示把输出的内容添加到一个已有的文件中。

在通过 ProcessBuilder 类的 redirectInput 和 redirectOutput 方法将输入和输出重定向之后，如果新的输入源和输出目标不是默认的管道方式，那么就无法访问所创建进程的 Process 类的对象中的对应流。例如，通过 redirectInput 方法把进程的输入重定向到某个文件之后，Process 类的对象的 getOutputStream 方法返回的是一个空的输出流，调用该输出流的 write 方法总是会抛出 IOException 异常；通过 redirectOutput 和 redirectError 方法把进程的正常和错误的输出重定向到某个文件之后，Process 类的对象的 getInputStream 和 getErrorStream 方法返回的是一个空的输入流，调用该输入流的 read 方法总是会返回 -1。因此，如果在 ProcessBuilder 类的对象中对进程的输入或输出进行了重定向，那么相应的 Process 类的对象的使用者一定要了解这一点，以免造成使用错误。

6.2.3 Thread 类的更新

Java 7 对于表示线程的类 java.lang.Thread 也做了一些更新，这些更新主要明确了 Thread 类的对象在某些情况下的行为，并且去掉了之前使用中比较模糊的和设计不合理的一部分。首先将 Thread 类的 clone 方法改为总是抛出 CloneNotSupportedException 异常，这是因为对一个 Thread 类的对象进行克隆是没有意义的。Java 7 显式地禁止了对 Thread 类对象的克隆操作。其次，在 Java 7 之前，Thread 类的 join 方法和 sleep 方法可以接收一个 long 类型的参数表示等待的时间，但是并没有定义当这个参数值为负数

时的处理方式。Java 7 中规定：如果这两个方法的等待时间参数的值为负数，则会抛出 `IllegalArgumentException` 异常。

另外一个明确了参数处理行为的是 `Thread` 类的构造方法。在创建 `Thread` 类的对象时可以使用的参数包括：表示 `Thread` 类的对象所在线程组的 `java.lang.ThreadGroup` 类的对象，表示需要运行的任务的 `java.lang.Runnable` 接口的实现对象，以及表示线程名称的 `String` 类的对象。如果传入的 `ThreadGroup` 类的对象为 `null`，那么会先尝试调用当前配置好的安全管理器（`java.lang.SecurityManager` 类的对象）的 `getThreadGroup` 方法来获取 `ThreadGroup` 类的对象；如果没有配置安全管理器或 `getThreadGroup` 方法也返回 `null`，那么会使用当前线程所在线程组的 `ThreadGroup` 类的对象；如果传入的 `Runnable` 接口的实现对象为 `null`，那么会调用 `Thread` 类的对象本身的 `run` 方法；如果传入的线程名称是 `null`，会抛出 `NullPointerException` 异常。

在调用 `Thread` 类的 `setContextClassLoader` 方法来设置线程上下文类加载器时，如果传入的参数为 `null`，则表明使用的是系统类加载器。如果无法使用系统类加载器，就使用启动类加载器。同样的，如果当前线程上下文类加载器是系统类加载器或启动类加载器，那么 `getContextClassLoader` 方法的返回值是 `null`。关于类加载器的更多介绍，请见第 9 章。

6.3 Java 实用工具类

Java 中包含实用工具类的包 `java.util` 在日常开发中使用得非常频繁，Java 7 对这个包中的不少内容进行了更新。

6.3.1 对象操作

在 `java.util` 包中新增了一个用来操作对象的工具类 `java.util.Objects`。`Objects` 类中包含的都是静态方法，通过这些方法可以快速对对象进行操作。

在进行两个对象的比较操作时，可以使用 `Objects` 类的 `compare` 方法。一般来说，进行对象比较是先由 Java 类实现 `java.lang.Comparable` 接口，再通过 `compareTo` 方法来进行比较。如果对集合中的元素进行排序，那么还会用到 `java.util.Comparator` 接口的实现。`Objects` 类中的 `compare` 方法可以将两个对象通过特定的 `Comparator` 接口的实现对象来进行比较。代码清单 6-10 中给出了一个简单的对 `Long` 对象进行比较的 `Comparator` 接口的实现，以及 `Objects` 类的 `compare` 方法的使用示例。

代码清单 6-10 `Objects` 类的 `compare` 方法的使用示例

```
private static class ReverseComparator implements Comparator<Long> {
    public int compare(Long num1, Long num2) {
        return num2.compareTo(num1);
    }
}
```

```
public void compare() {
    int value1 = Objects.compare(1L, 2L, new ReverseComparator());
}
```

判断对象相等的方式一般是调用 Object 类的 equals 方法，如判断两个对象 a 和 b 是否相等，可以使用代码“a.equals(b)”。Objects 类的 equals 方法可以直接判断两个对象是否相等，如“Objects.equals(a, b)”。此方法的一个好处是会对 null 值进行处理。如果直接调用一个对象的 equals 方法，需要先判断这个对象是否为 null，而使用 Objects 类的 equals 方法则不需要。如果 Objects 类的 equals 方法调用时的两个参数的值都是 null，则判断结果是 true；而如果只有一个参数为 null，则判断结果是 false；如果两个参数都不为 null，则调用第一个参数的 equals 方法来进行判断。Objects 类中与 equals 方法作用相似的是 deepEquals 方法，利用该方法也可以对两个对象进行相等性判断。与 equals 方法不同的是，如果 deepEquals 方法的两个参数都是数组，则会调用 java.util.Arrays 类的 deepEquals 来进行比较。Arrays 类的 deepEquals 方法在进行数组比较时，会考虑数组中的所有元素的相等性。在其他情况下，deepEquals 方法和 equals 方法的作用是相同的。代码清单 6-11 给出了示例，其中 value1 和 value2 的值都为 false。

代码清单 6-11 Objects 类的 equals 方法的使用示例

```
public void equals() {
    boolean value1 = Objects.equals(new Object(), new Object());
    Object[] array1 = new Object[] {"Hello", 1, 1.0};
    Object[] array2 = new Object[] {"Hello", 1, 1.5};
    boolean value2 = Objects.deepEquals(array1, array2);
}
```

Objects 类中的 hashCode 方法可以用来获取对象的哈希值。如果参数为 null，那么返回值是 0；否则返回值是参数对象的 hashCode 方法的返回结果。如果需要计算一组对象的哈希值，那么可以使用 Objects 类的 hash 方法。Objects 类的 hash 方法的实现使用的是 Arrays 类中 hashCode 方法。需要注意的是，在调用 hash 方法时传入单个对象作为参数的返回结果，与使用同样的参数调用 hashCode 方法的结果并不相同。如代码清单 6-12 所示，hashCode1 和 hashCode3 的值是不相同的，Objects 类的 hash 方法有自己的计算方式，不同于 hashCode 方法。

代码清单 6-12 Objects 类的 hash 和 hashCode 方法的使用示例

```
public void hash() {
    int hashCode1 = Objects.hashCode("Hello");
    int hashCode2 = Objects.hash("Hello", "World");
    int hashCode3 = Objects.hash("Hello");
}
```

Objects 中还有一组用于获取对象的字符串表示的 `toString` 方法的不同重载形式。Objects 的 `toString` 方法在参数为 `null` 时返回值是“`null`”，而在其他情况下相当于调用参数对象的 `toString` 方法。如果希望在参数为 `null` 时返回给定的内容作为提示信息，那么可以使用 `toString` 方法的另外一个重载形式，即通过一个额外的参数来指定参数值为 `null` 时的返回结果。如代码清单 6-13 所示，`str1` 和 `str2` 的值分别是“Hello”和“空对象”。

代码清单 6-13 Objects 类的 `toString` 方法的使用示例

```
public void useToString() {
    String str1 = Objects.toString("Hello");
    String str2 = Objects.toString(null, "空对象");
}
```

6.3.2 正则表达式

在日常开发中处理文本内容时经常会用到正则表达式。通过正则表达式可以简洁地解决一些常见的问题。Java 7 对 `java.util.regex` 包中的内容进行了更新，主要包括以下几个方面。

1. 支持命名捕获分组

捕获分组（capturing group）在使用正则表达式从文本中提取满足某种模式的部分字符时非常有用。通过把感兴趣的字符串的模式封装在捕获分组中，可以在匹配之后很容易地获取这些内容。另外捕获分组也可以在正则表达式中以后向引用（back reference）的方式来直接使用，以表示相同的模式。在 Java 7 之前，对捕获分组的引用只支持使用表示出现顺序的数字形式。这体现在 `java.util.Matcher` 类的 `group` 方法只接受 `int` 类型作为参数，后向引用的语法也仅支持类似“`\1`”这样的形式。如果一个正则表达式中包含很多捕获分组，那么开发人员需要清楚每个数字所代表的捕获分组的含义，这对于代码的编写者和阅读者来说都是一件很麻烦的事情。Java 7 引入的命名捕获分组可以很好地解决这个问题。通过为每个捕获分组添加一个有意义的名字，使开发人员可以很容易地明白每个分组所表示的含义，这比使用无意义的数字要方便得多。

代码清单 6-14 给出了通过命名捕获分组来匹配字符串并提取内容时的用法。待匹配的字符串是一个 URL，其中通过路径的不同部分来表示查询参数的名称和值。这种采用路径而不是查询字符串来指明参数的方式，在目前的 Web 开发中比较常见。在正则表达式的模式中，为提取每个参数内容的捕获分组都指定了一个有意义的名字。当匹配完成之后，可以通过 `Matcher` 类的 `group` 方法来获取每个捕获分组的内容，参数是在模式中指定的名字。采用“`?<>`”的格式为一个捕获分组命名，“`<>`”中的内容是名称。名称必须由大小写英文字母和数字组成，同时第一个字符必须是字母。

代码清单 6-14 通过命名捕获分组来匹配字符串并提取内容的示例

```

public void namedCapturingGroup() {
    String url = "http://www.example.org/uid/alex/docid/1/title/MyFirstBlog";
    Pattern pattern = Pattern.compile("^.*/uid/(?<uid>.*)/docid/(?<docid>.*)/
        title/(?<title>.*)");
    Matcher matcher = pattern.matcher(url);
    if (matcher.matches()) {
        String uid = matcher.group("uid"); // 值为 alex
        String docId = matcher.group("docid"); // 值为 1
        String title = matcher.group("title"); // 值为 MyFirstBlog
    }
}

```

捕获分组的名称也可以用在正则表达式之中，用来替换使用数字来进行后向引用的做法。代码清单 6-15 给出了一个示例，用正则表达式来查找重复出现的数字。在正则表达式中引用命名捕获分组使用的语法是“`\k<>`”，“`<>`”中是之前定义的捕获分组的名称。

代码清单 6-15 捕获分组的名称作为后向引用的示例

```

public void repeatPattern() {
    String str = "123-123-12-456-456";
    Pattern pattern = Pattern.compile("(?<num>\\d+)-\\k<num>");
    Matcher matcher = pattern.matcher(str);
    while (matcher.find()) {
        String repeat = matcher.group("num");
    }
}

```

2. 使用 “\x” 表示 Unicode 中的代码点

在正则表达式中，要引用 Unicode 字符时，可以通过“`\u×××`”的形式，如“`\u0030`”表示数字“0”。通过这种方式可以表示某些无法在源代码中直接出现的字符，比如无法输入汉字的开发人员可以通过这种方式来表示中文字符。第 4 章介绍 Unicode 时提到过，如果一个 Unicode 字符不在基本多语言平面（BMP）中，那么要在 Java 中以代理项对的方式出现，在转换成正则表达式中使用“`\u`”形式的时候，则需要两个相邻的字符。这样既不够直观，使用起来也不方便。Java 7 对正则表达式新增了“`\x`”来直接表示 Unicode 中的代码点。“`\x`”的使用方式与“`\u`”类似，只不过允许表示的范围更广，如 Unicode 代码点 U+1011F 可以直接表示成“`\x1011F`”。

3. 新标记 Pattern.UNICODE_CHARACTER_CLASS

在通过 Pattern 类的 compile 方法对正则表达式进行编译时，可以指定多个标记。这些标记可以控制匹配时的行为。常见的标记包括设置匹配时不区分大小写的 Pattern.CASE_INSENSITIVE，以及设置“.” 匹配包括换行符在内的所有字符的 Pattern.DOTALL。Java 7 中添加了一个额外的标记 Pattern.UNICODE_CHARACTER_CLASS 来

设置使用 Unicode 版本的预定义字符类和 POSIX 字符类。以 “\d” 这个预定义字符类为例，在默认情况下，只会匹配 0 到 9 的字符，如果启用了 UNICODE_CHARACTER_CLASS 标记，“\d” 会匹配 Unicode 规范中所定义的所有属于数字类别的字符，不只是 0 到 9 的数字，也会包括其他语言中的数字字符。代码清单 6-16 给出了一个示例，使用 “(\d+)" 模式来匹配字符串中的数字。示例中输入的字符串是 “100 1 0 0”，其中包含了一般的数字 “100” 和全角数字 “1 0 0”。在两个 Pattern 类的对象中，第一个 Pattern 类的对象在编译时没有使用 UNICODE_CHARACTER_CLASS 标记，只能匹配一般的数字 “100”，而第二个 Pattern 类对象可以匹配整个字符串。

代码清单 6-16 UNICODE_CHARACTER_CLASS 标记的使用示例

```
public void useUnicodeCharacterClass() {
    String str = "100 1 0 0";
    Pattern pattern = Pattern.compile("(\\d+)");
    Matcher matcher = pattern.matcher(str);
    if (matcher.find()) {
        String digit = matcher.group(1); // 值为 100
    }
    pattern = Pattern.compile("(\\d+)", Pattern.UNICODE_CHARACTER_CLASS);
    matcher = pattern.matcher(str);
    if (matcher.find()) {
        String digit = matcher.group(1); // 值为 100 1 0 0
    }
}
```

受到 UNICODE_CHARACTER_CLASS 标记影响的字符类除了 “\d” 之外，还包括 “\s”、“\w”、“\p{Lower}”、“\p{Upper}” 和 “\p{Punct}” 等。

4. 指定 Unicode 字符使用的书写格式

Java 7 中另外一个与正则表达式相关的更新也与 Unicode 相关。在 Java 7 之前，正则表达式在匹配 Unicode 字符串时允许指定 Unicode 字符所在的区块和类别，而在 Java 7 中还允许指定 Unicode 字符使用的书写格式 (script)。在指定书写格式时使用的是 “\p”，如代码清单 6-17 所示。“\p{script=Han}” 的含义是匹配字符串中书写格式为汉字的 Unicode 字符，因此正则表达式的执行结果是原始字符串中的汉字 “你好”。

代码清单 6-17 指定 Unicode 字符使用的书写格式的示例

```
public void matchScript() {
    String str = "abc 你好 123";
    Pattern pattern = Pattern.compile("(\\p{script=Han}+)");
    Matcher matcher = pattern.matcher(str);
    if (matcher.find()) {
        String hans = matcher.group(1); // 值为“你好”
    }
}
```

在匹配时可以使用的合法书写格式名称都定义在枚举类型 Character.UnicodeScript 中。

6.3.3 压缩文件处理

Java 标准库中的 `java.util.zip` 包用于对压缩文件进行处理。Java 7 对这一部分功能的更新也比较多。在对文件进行压缩时，允许选择压缩时缓存的中间结果的输出方式，这体现在 `java.util.zip.Deflater` 类的 `deflate` 方法增加了一个参数来表示不同的输出方式。默认的方式是 `Deflater.NO_FLUSH`，该方式由压缩者来确定输出缓存的中间结果的具体时机。在这种方式下，压缩者可以自由决定缓存中数据的大小，因此通常可以获得最佳的性能。第二种输出方式是 `Deflater.SYNC_FLUSH`，这种方式在每次调用 `deflate` 方法时自动清空内部缓冲区，把压缩的中间结果输出。这种方式的好处在于，如果有解压缩程序正在等待压缩之后的输出结果，及时地清空缓冲区可以让解压缩程序更早地开始工作，有利于提高解压缩程序的工作效率。不过这种方式会对压缩性能产生影响。最后一种输出方式是 `Deflater.FULL_FLUSH`，这种方式除了清空缓冲区之外，同时还重置压缩者的内部状态。如果解压缩的程序发现压缩的结果不正确，可以使用此方式来调用 `deflate` 方法，要求压缩者重新进行压缩操作。这种方式对性能的影响最大，只在必要时才使用。

与 `Deflater` 类对应的 `java.util.zip.DeflaterOutputStream` 类的对象在创建时增加了一个参数 `syncFlush`，用来表示对 `Deflater` 类的对象所缓存的中间结果的处理方式。如果 `syncFlush` 的值为 `true`，那么在调用 `DeflaterOutputStream` 类的对象的 `flush` 方法来清空其本身的内部缓冲区之前，会先按照 `SYNC_FLUSH` 的方式清空对应的 `Deflater` 类的对象所缓存的中间结果。

在 Java 7 之前，压缩文件中的文件名和注释都使用默认编码格式 UTF-8。这种 UTF-8 格式可能造成通过 Java 压缩的文件无法被其他工具打开。为了解决这个问题，Java 7 允许在创建压缩文件时显式地指定文件名和注释所用的字符集。这体现在 `java.util.zip.ZipFile`、`java.util.zip.ZipInputStream` 和 `java.util.zip.ZipOutputStream` 类的构造方法中都增加了一个 `java.nio.charset.Charset` 类的对象作为参数。这个 `Charset` 类的对象就表明了压缩文件中文件名和注释所用的编码字符集。通过 Java 产生的压缩文件中也包含了相关的元数据，用来表示压缩时的文件名和注释所使用的字符集。

除了上面这些比较大的改动之外，还有一些相关的小改动，包括：Java 7 支持大于 4 GB 的压缩文件的处理；`ZipFile` 类实现了 `java.io.Closeable` 接口，从而可以在 `try-with-resources` 语句中使用；`ZipFile` 类多了一个方法 `getComment`，用来获取在创建压缩文件时通过 `ZipOutputStream` 类的 `setComment` 方法所添加的文件注释；如果 `java.util.zip.GZIPInputStream` 类在处理压缩文件时遇到了格式不正确的压缩文件，会抛出更加具体的 `java.util.zip.ZipException` 异常。

在集合类方面，`java.util.Collections` 类增加了两个新的方法 `emptyIterator` 和 `emptyEnumeration`，用来返回空的迭代器对象和列举对象。

6.4 JavaBeans 组件

JavaBeans 是 Java 平台上的组件模型。要在 Java 平台上创建可复用的组件，应该遵循 JavaBeans 的规范。对于 JavaBeans 组件，开发人员比较熟悉的是 Java EE 中的 EJB (Enterprise JavaBeans)，以及 Java 类中遵循 JavaBeans 命名规范的属性获取和设置的方法。JavaBeans 的强大之处在于以规范的组件模型作为基础，可以通过工具很方便地进行单个组件的自定义和多个组件的组装等操作。对于所有遵循 JavaBeans 规范的组件，都可以通过工具以统一的方式来进行操作。

符合 JavaBeans 规范的每个组件都包含 3 类信息，分别是属性、方法和事件。属性指的是一个组件暴露出来的外观或行为上的特征。可以通过改变属性的值来定制组件的外观或行为。JavaBeans 组件的方法与一般的 Java 方法并没有区别，可以在其他组件中调用。事件是组件之间进行交互的方式。某个组件可以发布事件，而另外的组件可以在该事件上注册监听器。

6.4.1 获取组件信息

一个 JavaBeans 组件可以通过 `java.beans.Introspector` 类来获取组件中的属性、方法和事件的信息，使用的是 `Introspector` 类中的静态方法 `getBeanInfo`。该方法的返回值是包含了组件相关信息的 `java.beans.BeanInfo` 接口的实现对象。获取组件信息的方式可能有两种，一种是开发人员自己提供的 `BeanInfo` 接口的实现类，另外一种是由系统通过反射 API 来自动发现组件中的信息。对于第一种方式，系统会根据固定的名称模式来查找组件对应的 `BeanInfo` 接口的实现类。比如，对于类名为“`com.java7book.My`”的组件，查找类名为“`com.java7book.MyBeanInfo`”的 `BeanInfo` 接口的实现类作为组件的信息来源。如果没有找到相关实现类，会使用第二种方式，即通过反射 API 来发现相关信息。这两种方式的一个重要区别在于，在找到 `BeanInfo` 接口的实现类之后，不会再继续查找该组件的父类来获取信息；而通过反射 API 的方式则会沿着继承层次结构树一直向上查找父类中的相关信息。

对于具体的组件信息的获取过程，`getBeanInfo` 方法也提供了不同的重载方式供开发人员进行配置。可以配置的内容主要有两个，一个是在获取过程中包含哪些类中的信息。前面提到过，组件的父类中的信息有可能被包含进来，如果不希望包含组件的某些父类中的信息，那么可以指明终止组件信息获取过程的类名。当沿着继承层次结构树向上获取时，如果遇到指定的终止类，就停止继续获取。另外一个可配置的内容是对找到的 `BeanInfo` 接口实现类的处理方式。在 `getBeanInfo` 方法中允许设置 3 种不同的处理方式，对应的参数值分别是使用所有 `BeanInfo` 接口实现类的 `USE_ALL_BEANINFO`、忽略组件类对应的 `BeanInfo` 接口实现类的 `IGNORE_IMMEDIATE_BEANINFO` 和忽略包括组件类的父类在内的所有 `BeanInfo` 接口实现类的 `IGNORE_ALL_BEANINFO`。

通过这两种配置方式，可以很好地控制组件信息的获取过程。不过在 Java 7 之前，

这两种配置方式不能同时使用，只能使用其中一种。Java 7 添加了额外的 getBeanInfo 的重载方式。代码清单 6-18 给出了 getBeanInfo 方法的使用示例。MyBean 是要获取信息的组件类，ParentBean 是 MyBean 的父类。getBeanInfo 方法调用表明获取信息时不考虑 ParentBean 类的信息，同时忽略所有的 BeanInfo 接口的实现类。

代码清单 6-18 getBeanInfo 方法的使用示例

```
public void introspect() throws IntrospectionException {
    BeanInfo beanInfo = Introspector.getBeanInfo(MyBean.class, ParentBean.class,
        Introspector.IGNORE_ALL_BEANINFO);
    outputBeanInfo(beanInfo);
}
```

6.4.2 执行语句和表达式

JavaBeans 组件也提供了动态执行语句和表达式的能力，主要是为了方便工具的使用者以类似脚本语言的方式来对组件进行操作，比如调用一个组件对象 myBean 的 open 方法的语句可以直接写成“myBean.open()”。语句和表达式的区别在于，语句不关心具体的执行结果，而表达式则会把执行结果记录下来。语句和表达式分别用 java.beans.Statement 和 java.beans.Expression 类来表示。Expression 类继承自 Statement 类，并添加了获取和设置执行结果的方法。Statement 类的 execute 方法用来执行语句。在 Java 7 中，Expression 类增加了缓存执行结果的功能。当通过 Expression 类的 getValue 来获取执行结果时，如果 execute 方法之前没有被调用过，则会先调用 execute 方法，再返回结果，同时也会把执行结果记录下来。代码清单 6-19 给出了 Expression 类的使用示例。在创建 Expression 类的对象时，需要提供目标对象、方法名称和方法的调用参数。如果再次调用 getValue 方法，只会得到缓存中的执行结果，execute 方法不会被再次调用。

代码清单 6-19 Expression 类的使用示例

```
public void executeExpression() throws Exception {
    Expression expr = new Expression(new MyObject(), "greet", new Object[]
        {"alex"});
    expr.execute();
    Object result = expr.getValue();
}
```

6.4.3 持久化

JavaBeans 组件在其属性发生变化之后，可以被持久化，以保存组件的内部状态。之后在需要时可以把保存的内容再次读取出来，并恢复组件的内部状态。JavaBeans 组件的持久化依赖的是 Java 标准的对象序列化机制。一般来说，可以把组件的内部状态以流的二进制形式保存，或者保存成 XML 文件。以流的形式进行持久化时使用的是 Java

中的 `java.io.ObjectInputStream` 和 `java.io.ObjectOutputStream` 类，而在以 XML 文件作为持久化形式时，使用的是 `java.beans.XMLEncoder` 和 `java.beans.XMLDecoder` 类。

Java 7 对将 JavaBeans 组件保存成 XML 文档的功能做了更新，在 `XMLEncoder` 类中增加了构造方法，可以更加精细地控制保存时的行为。Java 7 之前的 `XMLEncoder` 构造方法只接受一个 `java.io.OutputStream` 类的对象作为参数，表示保存内容的输出流。而 Java 7 新增的构造方法中添加了额外的 3 个参数，分别是输出时使用的字符集、是否输出 XML 处理指令声明和整个 XML 文档的缩进空格数。允许指定输出时使用的字符集主要是为了满足不同的编码格式需求，而另外两个参数是为了使输出的 XML 文档内容可以被嵌入到其他 XML 文档中。代码清单 6-20 给出了 `XMLEncoder` 类的新构造方法的使用示例。如果 `XMLEncoder` 类的对象的输出要作为其他 XML 文档的一部分，第 3 个参数的值应该为 `false`，另外第 4 个参数应该是正确的缩进空格数，以保持整个 XML 文档的良好缩进格式。

代码清单 6-20 XMLEncoder 类的新的构造方法的使用示例

```
public void xmlEncode() throws IOException {
    OutputStream output = Files.newOutputStream(Paths.get("result.xml"),
        StandardOpenOption.CREATE_NEVER);
    try (XMLEncoder encoder = new XMLEncoder(output, StandardCharsets.UTF_8.
        name(), true, 0)) {
        encoder.writeObject(new MyBean());
    }
}
```

用于读取 `XMLEncoder` 类的输出文档的 `XMLDecoder` 类也有一些更新，主要是提供了更好的对 SAX 解析方式的支持。`XMLDecoder` 类新增了一个接受 `org.xml.sax.InputSource` 类型参数的构造方法。在 Java 7 之前，创建 `XMLDecoder` 类的对象时，只能使用 `InputStream` 类的对象来表示解析时的输入数据。而 `InputSource` 类则提供了更加丰富的方式来表示输入数据，除了 `InputStream` 类的对象之外，还支持使用标识符和 `java.io.Reader` 类的对象。

6.5 小结

本章主要围绕 Java 7 对标准库 API 所做的更新来展开，涉及数据库操作、`java.lang` 包、`java.util` 包和 JavaBeans 组件等方面。在这些更新中，数据库操作相关的对象都可以用在 `try-with-resources` 语句中，有利于提高代码的简洁性，新增的 `setNetworkTimeout` 和 `abort` 方法可以解决由于网络连接问题造成的查询操作等待时间过长的问题；而在 `java.lang` 包中，`ProcessBuilder` 类新增的对进程输入和输出进行重定向的能力，方便开发人员处理进程的输入和输出；在 `java.util` 包中，`Objects` 是很实用的工具类，对正则表达式在命名捕获分组和 Unicode 支持上的更新，方便了开发人员的使用；在 JavaBeans 组

件方面则增强了组件信息的获取过程、Expression 类的使用方式和 XMLEncoder 类在输出用于内嵌的 XML 文档时的能力。

Java 7 还有其他一些小的更新，限于篇幅，不可能逐一进行介绍。总的来说，Java 7 除了添加新的功能方便开发人员之外，还更新了大量文档以澄清之前版本中容易产生误解的地方，另外还对之前比较模糊的行为添加了明确的定义，前面提到的 Thread 类的更新就是很好的例子。

第 7 章 Java 虚拟机

Java 虚拟机是 Java 语言能够取得成功的重要因素之一。Java 语言在推出之时的一个口号是“编写一次，到处运行（write once, run anywhere）”。这个特性使 Java 语言相对同时代的其他编程语言更具吸引力，而正是 Java 虚拟机使这个特性成为可能。Java 源代码经过编译器编译之后，得到以类文件（.class 文件）形式出现的字节代码。只要字节代码的版本与当前 Java 虚拟机的版本是兼容的，这些字节代码可以在任何平台上的 Java 虚拟机上执行。Java 虚拟机的作用是为 Java 程序屏蔽底层操作系统的细节，提供一个统一的运行平台。第 2 章已经对 Java 虚拟机进行了简单的介绍，本章将进行更加深入的讨论。

对于 Java 虚拟机本身，不同的开发人员所关心的角度可能并不相同。有的开发人员从事的是 Java 虚拟机本身的开发或虚拟机级别的性能优化工作，从这个角度出发要求对 Java 虚拟机的实现有比较深入的了解，不过这类开发人员的数量相对较少；较大一部分开发人员对 Java 虚拟机的了解仅限于知道它的存在而已，在绝大多数时候，这些开发人员都在 Java 语言这个层次上工作。本书要介绍的主要内容并不是 Java 虚拟机，而是 Java 语言，因此并不会涉及 Java 虚拟机本身的实现相关的底层细节等内容。之所以仍然安排一章来介绍 Java 虚拟机，是为了帮助开发人员更好地使用 Java 虚拟机。Java 语言也提供了一些与 Java 虚拟机进行交互的能力。这些能力也是本章中的内容之一。本章的内容可以看成是 Java 开发人员所应该了解的 Java 虚拟机的相关内容的一个汇总。

7.1 虚拟机基本概念

“编写一次，到处运行”是 Java 语言吸引开发人员的重要原因之一。用 Java 语言编写的程序，可以在任何平台上运行，只需要在操作系统之上安装 Java 运行环境即可。某些编程语言的开发平台中并没有虚拟机的概念，而是通过直接从源代码生成目标操作系统上的二进制文件来运行。不同操作系统上的二进制文件是无法兼容的。以 C/C++ 语言为例，在 Windows 平台上编译 C/C++ 源代码所生成的可执行文件，无法在 Linux 平台上运行。当需要使用程序的时候，用户要么直接下载其操作系统对应的可执行文件，要么根据源代码自行编译和链接来得到可执行文件。对程序的开发人员来说，如果程序需要支持不同的操作系统平台，则需要做很多工作来确保程序在不同的操作系统平台上都可以正常工作，其中包括对程序进行修改以适应不同的操作系统平台。虽然有一些类库可以帮助解决这个问题，但是开发人员的任务量仍然比较大。

Java 平台早在开发时就引入了虚拟机的概念。Java 程序不是由操作系统以可执行文件的形式直接运行的，而是运行在 Java 虚拟机中的。在运行 Java 程序的时候，需要指

定一个主 Java 类。Java 虚拟机在启动之后，会从主 Java 类的 main 方法开始执行。当 main 方法执行结束之后，Java 虚拟机会自动终止。每个 Java 虚拟机在运行时是底层操作系统上的一个独立的进程。比如，当在 Windows 操作系统上运行 Java 程序的时候，可以从任务管理器的进程列表中看到名为“java.exe”或“javaw.exe”的进程，这些就是 Java 虚拟机的进程。

虚拟机的作用主要有两个：一个是为应用程序屏蔽底层操作系统的细节，另外一个则是为应用程序提供必要的运行时的支持能力。不同的操作系统在实现上存在很多差异。跨平台的应用程序需要自己来考虑这些不同，并在代码中进行处理。这通常意味着更长的开发时间和更高的维护成本，也意味着需要更多熟悉不同平台的开发人员。很多程序都采用多线程的方式来提高性能。在创建线程时，Windows 平台上的 API 与 Linux 上的相关 API 就存在很大不同，在程序中需要通过不同的分支代码来进行处理。虚拟机的作用就在于处理这些细节，为程序提供统一的接口。同样的线程操作，在 Java 语言中可以通过抽象的 java.lang.Thread 类来完成。而 Thread 类在不同平台上实现的不同，则由虚拟机来负责处理。除此之外，虚拟机为在其上运行的应用程序提供了必要的支持。以 Java 虚拟机为例，这些支持包括基本类型和操作符、对象模型、Unicode 支持、动态链接、垃圾回收器、内存模型和访问控制等。虚拟机所提供的这些功能，是 Java 程序运行时的基础。在 Java 语言中可以与这些功能进行交互，比如，第 4 章中介绍的 Java 语言中的 Unicode 相关的内容。其他的相关内容也会在之后的章节中进行介绍。简而言之，虚拟机的作用相当于一个简化后的操作系统，它所提供的功能不仅丰富，而且规范、统一。

随着 Java 语言的流行，在编程语言开发平台中使用虚拟机也成为了一种被广泛认可的做法。微软的 .NET 框架也采用了类似的架构。.NET 架构中的虚拟机被称为通用语言运行环境（Common Language Runtime, CLR）。.NET 平台支持的各种编程语言的代码，如 C# 和 VB.NET，会先被编译成 CLR 上的中间语言（Common Intermediate Language, CIL）的形式，类似于 Java 中的字节代码。CIL 再由 CLR 来运行。这种使用模式与 Java 的做法是非常类似的。

7.2 内存管理

运行的程序总要与内存进行交互。内存作为操作系统中的重要资源，对它的分配和释放进行管理是一项非常重要的工作。对于某些编程语言，内存管理的工作由开发人员来处理，C 和 C++ 语言是这类编程语言的典型代表。以 C++ 为例，当程序通过 new 操作符创建新的对象之后，就会分配相应的内存资源。当程序不再需要这些对象的时候，需要在代码中将其显式释放，一般通过 delete 操作符来完成。内存分配和释放操作的正确性，由开发人员来保证，这在很大程度上取决于开发人员的经验和技能。实际上，在使用这些编程语言开发的程序中，与内存分配和释放相关的问题是比较的多的。

第一个常见的问题是产生悬挂引用 (dangling reference)。悬挂引用指的是对某个对象的引用实际上指向一个错误的内存地址。比如程序中某部分代码引用了由另外一部分代码创建的对象，在代码的运行过程中，这个被引用的对象被删除，它所占用的内存也被释放。随后这部分内存被重新分配给另外的对象使用，而之前的代码可能仍然保存着对原始对象的引用。当代码仍然通过这个旧的引用尝试访问对象的时候，就会出现错误，因为这个旧的引用所对应的内存地址中的内容已经发生了变化。如果这个新的内存地址被分配给另外的进程，这次访问请求就可能造成程序退出。

第二个常见的问题是产生内存泄露。造成这个问题的原因是某些对象所占用的内存没有被释放，又没有对象引用指向这些内存。这样就会导致这部分内存对程序来说既不可用，又无法被释放，因为在缺乏对象引用的情况下，程序无法对这部分内存进行任何操作。

对于需要进行显式内存管理的编程语言来说，开发人员通常需要遵循某些最佳实践或利用相关工具来进行正确的内存管理。比如 C++ 语言中的最佳实践一般在构造方法中分配内存，在析构方法中释放内存。利用 C++ 中的智能指针也可以进行自动内存释放。不管采用何种方式，都对开发人员提出了比较高的要求，也会造成程序中存在很多与内存管理相关的潜在问题。

因为显式内存管理比较容易出现错误，所以不少编程语言采用了自动内存管理机制。自动内存管理的含义是运行平台会提供专门的组件来进行内存的管理工作。这个组件通常被称为垃圾回收器 (garbage collector, GC)。垃圾回收器不仅负责内存的回收，还负责内存的分配。这些编程语言通常也不提供直接的 API 来进行内存的分配和释放，内存的分配是隐式进行的，在创建新对象时自动分配所需的内存，而对象所占用的内存不需要在程序中显式释放，由垃圾回收器自动进行处理。Java 语言是使用自动内存管理机制的典型代表。在 Java 中有用来创建对象的 new 操作符，但是没有对应的 delete 操作符。

Java 平台上的内存管理由垃圾回收器负责完成。垃圾回收器属于 Java 虚拟机的一部分。当 Java 程序在虚拟机中运行时，垃圾回收器也在运行。垃圾回收器负责管理虚拟机中的内存。它的职责很简单，即负责内存的分配和释放。在程序运行过程中创建新对象时，垃圾回收器要在虚拟机可用的内存中找到一块合适的内存区域供此对象来使用。当不再需要某一个对象时，垃圾回收器会负责回收该对象占用的内存空间，以供之后的内存分配所用。虽然从描述上来说，垃圾回收器的功能并不复杂，但是要实现一个实用的垃圾回收器并非易事，最重要的是处理好垃圾回收器与运行的程序之间的关系。

当 Java 虚拟机启动并运行某个程序之后，它所能使用的内存总量的上限通常是固定的。在程序刚开始运行的时候，虚拟机中的大部分内存都处于空闲可用的状态。随着程序的运行，不断有空闲的内存区域被分配给程序运行所需的对象来使用。经过一段时间之后，虚拟机的内存大概就可以分成三类：第一类是当前仍然处于空闲状态的可用内存；第二类是正在被程序使用的内存；第三类是程序已经不再使用的内存，已经不再需要其中包含的数据内容。第二类和第三类内存的区别在于其所属的对象是否处于活动状

态。一个对象处于活动状态的含义是指在当前运行的程序中还存在指向该对象的引用。如果没有引用指向一个对象，那么说明该对象无法被运行的程序所使用，它所占用的内存会被当成垃圾来回收。

随着程序的不断运行，虚拟机的内存中可用的空闲空间越来越少，垃圾越来越多。这时就需要运行垃圾回收器来回收内存中的垃圾区域，将其转换成可用的区域，以供下次内存分配时使用。垃圾回收器运行一次之后，程序中的部分垃圾区域会被正确回收。Java 虚拟机中的垃圾回收器是运行在一个独立的线程中的，它会根据当前虚拟机中的内存状态，决定在什么时候进行垃圾回收工作。每次垃圾回收时所处理的内存区域的范围也是不同的。垃圾回收器的具体运行时间和频率无法事先预计，取决于垃圾回收器的实现算法。不同的虚拟机实现中的垃圾回收算法也有所不同。

由于垃圾回收线程和当前应用程序同时在 Java 虚拟机中运行，因此当前运行程序会受到垃圾回收器的影响。不同的垃圾回收器实现算法对程序的影响也不相同。应该根据程序的特性来选择与之相匹配的垃圾回收器实现算法。

在垃圾回收器的实现方式中，通常有很多因素需要考虑和权衡，其中与当前运行程序相关的是垃圾回收器的运行方式。一般来说有并发运行和暂停执行两种。并发运行的含义是指垃圾回收器与程序同时运行，垃圾回收器在绝大部分时候不会影响程序的运行。而暂停执行的方式是垃圾回收器在运行时，程序的运行被暂停。并发运行的方式对程序影响较小，但是对垃圾回收器的实现要求较高，实现起来也更复杂。在回收的过程中，由于程序仍然在运行，因此内存的状态也在不断发生变化。垃圾回收器需要花费更多的运行时间和内存空间来处理这种情况。暂停执行的做法实现起来比较简单，因为在回收的过程中，内存的状态是固定不变的。但是暂停会对程序产生比较大的影响，用户也可能感觉到程序运行时的停顿。

虽然大多数时候垃圾回收器的运行时间和频率是无法预计的，但是程序仍然可以在特定的时机建议垃圾回收器进行回收工作，通过 `System.gc` 方法就可以建议垃圾回收器立即运行。不过在这种情况下，垃圾回收器也可能选择不运行。

7.3 引用类型

在 Java 程序中可以通过 `System.gc` 方法来建议垃圾回收器立即进行回收工作。除此之外，Java 程序本身与垃圾回收器能够进行的直接交互比较有限。垃圾回收器在进行回收时并不了解运行程序的具体特征。因此，在某些情况下，垃圾回收器可能并不能够按照程序所期望的方式工作。比如，一个图像处理程序可能同时打开多个图像文件进行编辑，而同一时刻只有一张图片处于编辑状态。当同时打开的图片过多时，程序所占用的内存空间会变大，垃圾回收器无法回收这些处于活动状态的对象所占用的内存，而使虚拟机中的可用内存不断减少。当垃圾回收器无法找到可用的空闲内存时，创建新对象的操作会抛出 `java.lang.OutOfMemoryError` 错误，导致虚拟机退出。而就这个图像处理程

序的特征来说，当可用内存不足的时候，可以把那些当前不处于编辑状态的图像所占用的内存释放掉，这样的垃圾回收操作对这个程序来说是合理的。同样，其他的程序也可能有类似的情况。

对于以上情况，Java程序需要通过一种方式把其中的对象在内存需求方面的特征传达给垃圾回收器。垃圾回收器根据对象的特征可以更好地进行回收。比如，在虚拟机可用内存不足的时候，释放程序中更多的内存。这种传达方式是通过Java中对象的引用类型来实现的。在程序的运行过程中，对于同一个对象，可能存在多个指向它的引用。如果不再有引用指向一个对象，那么这个对象会成为垃圾回收的候选目标。Java语言中存在不同的对象引用类型。不同类型的引用对垃圾回收器的含义是不同的。

7.3.1 强引用

在Java程序中，最常见的引用类型是强引用（strong reference），它也是默认的引用类型。当在Java语言中使用new操作符创建一个新的对象，并将其赋值给一个变量的时候，这个变量就成为指向该对象的一个强引用。在前面提到过，判断一个对象是否存活的标准为是否存在指向这个对象的引用。垃圾回收器可能采取不同的算法来判断对象的引用是否存在。一个常见的做法是使用引用计数器。当有新的引用指向某个对象时，把该计数器的值加1；当一个引用失效时，就把该计数器的值减1。例如，显式地把一个引用某个对象的变量的值设为null，该对象的引用计数器的值会减1。当一个对象的引用计数器的值变为0的时候，说明不存在任何指向该对象的引用，该对象可以被垃圾回收器回收。引用计数器的原理比较简单，但是实现起来需要编译器的支持，另外使用引用计数器不能解决循环引用孤岛的回收问题。比如，3个对象之间互相存在引用关系，但是并不存在指向这3个对象的其他引用，这三个对象实际上就成为了内存区域中的一个孤岛。这3个对象的引用计数器的值都不为0，因此无法通过引用计数器的方式来回收。

由于引用计数器存在无法处理“孤岛”的问题，Java虚拟机的垃圾回收器没有采用这种做法，而是采取跟踪对象引用的做法。这种做法会从虚拟机内存中的某些存活对象开始，递归检查这些对象所引用的其他对象，直到找到不引用其他对象的对象为止。在这个过程中所发现的所有对象都会被标记为存活的，而其他对象则是可以被回收的。这个遍历过程的起始对象是一个集合，称之为根集合。根集合中一般包含系统类、程序寄存器、JNI全局引用、静态变量和线程的当前活动栈中的变量所指向的对象等。可以将这个跟踪过程看成是基于引用关系的树的遍历。在跟踪过程中发现的存活对象被称为可达的（reachable）。将从遍历的根节点到当前存活对象的路径称为可达路径。这条路径的边对应的是对象之间的引用。如果一个对象的可达路径中只包含强引用，则把这个对象称为强引用可达的（strongly reachable）。程序中的大多数存活对象都是强引用可达的。

对于垃圾回收器来说，强引用的存在会阻止一个对象被回收。在垃圾回收器遍历对象引用并进行标记之后，如果一个对象是强引用可达的，那么这个对象不会作为垃圾回

收的候选。因为该对象仍然被程序所使用，回收其内存显然是一个错误的做法。虽然由于垃圾回收器的存在，Java 虚拟机中并不存在真正意义上的内存泄露，但是某些错误的用法会对程序中所能使用的内存空间造成影响。这些情况可以看成是另外一种意义上的内存泄露，这些内存泄露的发生也都和强引用的使用有关。

通常来说，这种意义上的内存泄露有两种情况：一种是虚拟机中存在程序无法使用的内存区域。这些内存区域被程序中一些无法使用的存活对象占用。这些对象由于存在隐式的强引用，无法对其进行垃圾回收。但是在程序的正常运行过程中，这些对象也无法被使用。造成这种问题的原因通常是程序编写时的逻辑错误。另一种情况是程序中存在大量存活时间过长的对象。这些对象的存活时间长于使用它们的对象。在正常情况下，这些对象在引用它们的对象被回收之后，也应该被回收，但是由于某些程序中的错误而没有被回收。这些对象无法被回收，仍然占据着虚拟机中的内存资源。时间长了，虚拟机会因为没有足够的内存分配给新的对象而抛出 OutOfMemoryError 错误。下面会通过示例对这两种情况进行具体的说明。

对于第一种情况，代码清单 7-1 中给出了一个简单的先进先出队列的实现。它在内部使用了一个 java.util.List 接口的实现对象来保存队列中的对象。向队列中添加的新对象会被直接放在 List 接口实现对象的末尾。队列的队首位置则由内部变量 topIndex 来维护。每次有对象被移出队列时，topIndex 的值会增加 1。这个队列实现的问题在于出队列的方法只简单地改变了 topIndex 的值，并没有把对象从队列中删除。在经过若干次队列操作之后，topIndex 的值会逐渐变大。变量 backendList 所指向的对象中包含的序号小于 topIndex 的对象无法被队列的使用者通过正常的方式来访问。由于 backendList 所指向的对象仍然包含指向这些对象的强引用，因此这些对象也无法被垃圾回收，这些对象占用的内存就成为虚拟机中无法使用的区域。

代码清单 7-1 产生内存泄露的先进先出队列的实现

```
public class LeakingQueue<T> {
    private List<T> backendList = new ArrayList<T>();
    private int topIndex = 0;
    public void enqueue(T value) {
        backendList.add(value);
    }

    public T dequeue() {
        T result = backendList.get(topIndex);
        topIndex++;
        return result;
    }
}
```

第二种情况的典型情景发生在使用基于内存实现的缓存的时候。代码清单 7-2 中给出了一个示例。利用 calculate 方法进行实际运算所需的时间可能比较长，因此使用了一

个 `java.util.HashMap` 类的对象来保存之前计算的结果。在 `calculate` 方法被调用时，会先检查缓存中是否已经存在之前计算出来的结果，这样可以避免重复的计算，进而提高性能。不过这种做法延长了计算结果对象的存活时间。在不使用缓存的情况下，在 `calculate` 方法的调用者获得计算结果对象，并完成对该对象的使用之后，就可以对该对象进行垃圾回收。当使用了缓存之后，计算结果对象的存活时间就变得至少和用来进行缓存的 `HashMap` 类的对象一样长。因为 `HashMap` 类的对象有其所包含的所有计算结果对象的引用，所以，只要 `HashMap` 类的对象无法被回收，其中所包含的计算结果对象也无法被回收。在 `calculate` 方法被多次调用之后，缓存中包含的对象会越来越多，导致占用的内存越来越大，而程序中其他部分可用的内存则越来越少。

代码清单 7-2 使用缓存造成对象存活时间过长的示例

```
public class Calculator {
    private Map<String, Object> cache = new HashMap<String, Object>();
    public Object calculate(String expr) {
        if (cache.containsKey(expr)) {
            return cache.get(expr);
        }
        Object result = doCalculate(expr);
        cache.put(expr, result);
        return result;
    }

    private Object doCalculate(String expr) {
        return new Object();
    }
}
```

从前面两个示例中可以看出，强引用所提供的与垃圾回收器的交互功能非常有限。当强引用存在的时候，所指向的对象无法被垃圾回收。为了增强程序与垃圾回收器的交互能力，JDK 1.2 引入了 `java.lang.ref` 包，提供了 3 种新的引用类型，分别是软引用、弱引用和幽灵引用。这些引用类型除了可以引用对象之外，还可以在不同程度上影响垃圾回收器对被引用对象的处理行为。

7.3.2 引用类型基本概念

所有的引用类型都是 `java.lang.ref.Reference` 类的子类。`Reference` 类中包含了所有引用类型公用的方法。对一个 `Reference` 类的对象来说，在创建的时候都需要提供一个它所指向的对象。引用一个对象的通常做法如代码清单 7-3 所示，变量 `obj` 引用了一个新创建的 `Object` 类的对象，这相当于在该对象上添加了一个强引用。

代码清单 7-3 引用对象的通常做法

```
Object obj = new Object();
```

如果使用引用类型，相关的实现如代码清单 7-4 所示。通过创建一个 Reference 子类 `java.lang.ref.SoftReference` 的对象 `ref`，就可以在变量 `obj` 所引用的对象上添加一个软引用。在创建 `SoftReference` 类的对象时，要把所指向的对象作为构造方法的参数传递进去。需要注意的是，在创建了新的软引用之后，要显式地把之前创建的对象上的强引用清除，这也是代码清单 7-4 中“`obj = null;`”语句的作用。在清除了强引用之后，指向新创建的 `Object` 类的对象的引用就只有 `SoftReference` 类的对象 `ref`。如果在程序中没有清除之前的强引用，添加其他引用类型实际上是毫无作用的。因为只要强引用存在，该对象就无法被垃圾回收器处理。

代码清单 7-4 引用类型使用示例

```
Object obj = new Object();
SoftReference<Object> ref = new SoftReference<Object>(obj);
obj = null;
```

使用了引用类型之后，对象之间的引用关系也发生了变化。在代码清单 7-3 中，通过 `obj` 持有对 `Object` 类的对象的强引用；而在代码清单 7-4 中，程序持有的是对 `SoftReference` 类的对象 `ref` 的强引用，`ref` 再通过软引用来指向 `Object` 类的对象。这种引用关系的改变，使对 `Object` 类的对象进行垃圾回收变为可能。

在创建引用对象时的一种错误做法如代码清单 7-5 所示。这种做法的问题在于没有使用强引用先指向待引用的对象。垃圾回收器可能随时进行内存的回收工作。可能出现的一种情况是：在 `SoftReference` 类的对象创建出来之后，垃圾回收器正好回收了 `SoftReference` 类的对象所指向的对象，这会使该引用对象实际上毫无作用。

代码清单 7-5 使用引用类型的错误用法

```
SoftReference<Object> ref = new SoftReference<Object>(new Object());
```

`Reference` 类的 `get` 方法用来获取所引用的实际对象。如果引用关系已经被清除，该方法的返回值为 `null`。因此在使用 `get` 方法时，需要检查返回值是否为 `null`，并分别使用不同的代码逻辑。`Reference` 类的 `clear` 方法用来清除引用关系。当在某个时间点上不再需要继续引用 `Reference` 类的对象所指向的对象时，可以使用 `clear` 方法来清除这个引用。当 `clear` 方法被调用之后，无法通过 `get` 方法获取所引用的对象，一般由垃圾回收器或程序本身在合适的时机调用 `clear` 方法来清除引用。`Reference` 类中的另外两个方法 `enqueue` 和 `isEnqueued` 都与引用队列相关，分别用来把当前引用对象放入对应的引用队列中，以及判断当前引用对象是否已经被放入队列中。

在使用 `get` 方法的时候，需要确保有强引用指向 `get` 方法的返回值，否则可能会出现错误。代码清单 7-6 给出了错误的用法，虽然先检查了 `get` 方法的返回值是否为 `null`，但是 `obj` 变量所指向的对象仍有可能为 `null`。这是因为垃圾回收器可能在 `if` 判断语句之后运行，回收了引用对象所指向的对象，使下一次的 `get` 方法调用返回 `null`。

代码清单 7-6 使用 get 方法的错误示例

```
if (ref.get() != null) {
    Object obj = ref.get();
}
```

引用队列是一个包含了引用类型对象的队列，按照先进先出的方式来操作。引用队列由类 `java.lang.ref.ReferenceQueue` 来表示。`ReferenceQueue` 类中提供了从队列中获取引用对象的方法，但是并没有向队列中添加对象的方法。添加引用对象到队列中的操作由垃圾回收器自动完成或通过 `Reference` 类的 `enqueue` 方法来完成。从引用队列中获取对象有两种方式，分别是阻塞式和非阻塞式的。阻塞式的方式是通过 `remove` 方法来实现的，该方法会阻塞当前线程直到从队列中获取到一个引用对象为止；非阻塞式的方式则通过 `poll` 方法来实现。如果当前队列中有引用对象，`poll` 方法会返回队列中的第一个对象，否则直接返回 `null`。在创建 `Reference` 类对象时，可以提供一个额外的引用队列的对象作为参数，这样可以把引用对象和该引用队列关联起来。垃圾回收器会在合适的时间点上把引用对象放入到对应的队列中。代码清单 7-7 给出了引用队列的使用示例。

代码清单 7-7 引用队列的使用示例

```
Object obj = new Object();
ReferenceQueue queue = new ReferenceQueue();
SoftReference<Object> ref = new SoftReference<Object>(obj, queue);
obj = null;
```

7.3.3 软引用

软引用（soft reference）在强度上弱于强引用，用 `java.lang.ref.SoftReference` 类来表示。如果一个对象不是强引用可达的，同时可以通过软引用访问，那么将这个对象称为软引用可达（softly reachable）。软引用所要传递给垃圾回收器的信息是：软引用所指向的对象是在需要的时候被回收的。垃圾回收器会保证在抛出 `OutOfMemoryError` 错误之前，回收掉所有软引用可达的对象。通过软引用，垃圾回收器就可以在内存不足时释放软引用可达的对象所占的内存空间。程序所要做的是保证软引用可达的对象被垃圾回收器回收之后，程序也能正常工作。在之前介绍的图像编辑器打开文件的示例中，可以用强引用指向当前正在编辑的图片，而用软引用指向不处于编辑状态的其他已经打开的图片。这样当图像编辑器程序的内存不足时，垃圾回收器可以释放不处于编辑状态的图片所占的内存空间。当程序中需要引用占用内存比较大的对象时，可以考虑使用软引用指向该对象。

下面通过一个示例来具体说明软引用的用法。代码清单 7-8 中的 `FileEditor` 类用来对多个文件进行编辑。出于性能方面的考虑，`FileEditor` 类的对象会在内部缓存之前已经打开过的文件的数据内容，以方便用户在同时打开的多个文件之间进行快速切换。同时

打开的文件过多会占用比较多的内存资源。因此，表示文件数据的 `FileData` 类使用软引用用来指向包含文件数据的 `byte[]` 对象。当虚拟机的内存不足时，这些 `byte[]` 对象可以被垃圾回收器释放。这里需要注意 `FileData` 类的 `getData` 方法的实现中对软引用的使用方式。通过 `get` 方法获取软引用所指向的对象之后，需要判断这个对象是否还存活。如果 `get` 方法的返回值为 `null`，那么说明对该对象的引用已经被清空，应该重新创建出相关的对象。

代码清单 7-8 使用软引用的文件编辑器

```
public class FileEditor {
    private static class FileData {
        private Path filePath;
        private SoftReference<byte[]> dataRef;

        public FileData(Path filePath) {
            this.filePath = filePath;
            this.dataRef = new SoftReference<byte[]>(new byte[0]);
        }

        public Path getPath() {
            return filePath;
        }

        public byte[] getData() throws IOException {
            byte[] dataArray = dataRef.get();
            if (dataArray == null || dataArray.length == 0) {
                dataArray = readFile();
                dataRef = new SoftReference<byte[]>(dataArray);
                dataArray = null;
            }
            return dataRef.get();
        }

        private byte[] readFile() throws IOException {
            return Files.readAllBytes(filePath);
        }
    }

    private FileData currentFileData;
    private Map<Path, FileData> openedFiles = new HashMap<>();

    public void switchTo(String filePath) {
        Path path = Paths.get(filePath).toAbsolutePath();
        if (openedFiles.containsKey(path)) {
            currentFileData = openedFiles.get(path);
        } else {
            currentFileData = new FileData(path);
            openedFiles.put(path, currentFileData);
        }
    }
}
```

```

    }

    public void useFile() throws IOException {
        if (currentFileDialog != null) {
            System.out.println(String.format("当前文件 %1$s 的大小为 %2$d",
                currentFileDialog.getPath(), currentFileDialog.getData().length));
        }
    }
}

```

为了测试代码清单 7-8 中软引用在实际运行中的效果，使用 FileEditor 类的对象依次打开某个目录下包含的大小各异的多个文件，同时通过虚拟机的启动参数“-Xmx”把虚拟机所用的堆内存的最大值设置为一个相对较小的值。在运行时会发现，虽然虚拟机可用堆内存的最大值远小于所处理的所有文件的大小的总和，但是程序在运行中也不会抛出 OutOfMemoryError 错误。这是因为当虚拟机中内存不足时，软引用指向的 byte[] 对象会被释放，从而可以腾出内存空间供之后的文件操作使用。如果使用强引用指向 byte[] 对象，在打开目录下的部分文件之后，就会出现 OutOfMemoryError 错误，这是因为虚拟机的堆内存不足以容纳全部文件的内容，而垃圾回收器又无法释放强引用指向的 byte[] 对象。

7.3.4 弱引用

弱引用 (weak reference) 在强度上弱于软引用，用 java.lang.ref.WeakReference 类来表示。弱引用传递给垃圾回收器的信息是：在判断一个对象是否存活时，可以不考虑弱引用的存在。比如，指向一个对象的既有一个强引用，又有多个弱引用，当该强引用被移除之后，这个对象就可以被垃圾回收器回收，可以忽略指向该对象的弱引用。弱引用的存在不会影响垃圾回收器回收一个对象，还提供了一种方式可以引用到这个对象。如果一个对象既不是强引用可达，也不是软引用可达，同时可以通过弱引用来访问，则将该对象称为弱引用可达 (weakly reachable)。

弱引用的重要作用是解决对象的存活时间过长的问题。在程序中，一个对象的实际存活时间应该与它的逻辑存活时间一样。从逻辑上来说，一个对象应该在某个方法调用完成之后就不再需要，可以对其进行垃圾回收。但是，如果仍然有其他的强引用存在，该对象的实际存活时间会长于逻辑存活时间，直到其他的强引用不再存在。这样的对象在程序中过多出现会导致虚拟机的内存占用率上升，最后产生 OutOfMemoryError 错误。要解决这样的问题，需要小心注意管理对象上的强引用。当不再需要引用一个对象时，显式地清除这些强引用。不过这会对开发人员提出更高的要求，代码编写起来也更加复杂。更好的办法是使用弱引用来代替强引用来引用这些对象。这样既可以引用对象，又可以避免强引用带来的问题。

比较典型的例子是在哈希表中使用弱引用。在代码清单 7-9 中，通过 BookKeeper 类来对图书及其借阅者进行管理。在内部实现中使用了一个 HashMap 类的对象来保存图书及其借阅者之间的对应关系。由于 HashMap 类的对象具有对所包含的键和值的对象的强引用，这会使 Book 类和 User 类对象的存活时间变得至少和 HashMap 类的对象本身一样长。当 HashMap 类的对象本身还存活时，其中所包含的 Book 类和 User 类的对象都无法被垃圾回收器回收。

代码清单 7-9 HashMap 类造成对象存活时间过长的示例

```

public class BookKeeper {
    private Map<Book, Set<User>> books = new HashMap<>();
    public void borrowBook(Book book, User user) {
        Set<User> users = null;
        if (books.containsKey(book)) {
            users = books.get(book);
        }
        else {
            users = new HashSet<User>();
            books.put(book, users);
        }
        users.add(user);
    }

    public void returnBook(Book book, User user) {
        if (books.containsKey(book)) {
            Set<User> users = books.get(book);
            users.remove(user);
        }
    }
}

```

解决这个问题的做法是使用弱引用来指向这些对象，而不是使用默认的强引用。因为这样使用 Map 接口的情况很多，Java 标准库提供了 `java.util.WeakHashMap` 类来满足这种常见的需求。WeakHashMap 类使用弱引用来指向其中所包含的键，而键对应的值对象仍然由强引用来指向。使用弱引用的好处是 WeakHashMap 类的对象本身对其中包含的键的弱引用不会影响键对象的垃圾回收。当键对象不存在其他类型更强的引用时，键对象会被从 WeakHashMap 类的对象中删除。对于代码清单 7-9 中存在的问题，只需要把 books 对应的 Map 接口的实现类换成 WeakHashMap 类即可。不过 WeakHashMap 类的对象中包含的值对象仍然由强引用来指向，因此不能在值对象中包含键对象的引用。这种循环引用会导致键对象无法被垃圾回收器回收。如果觉得对于值对象使用强引用不合适，可以在添加到 WeakHashMap 类的对象之前用一个 WeakReference 类的对象来包装它。

7.3.5 幽灵引用

幽灵引用 (phantom reference) 是强度最弱的一种引用类型，用 `java.lang.ref.PhantomReference` 类来表示。幽灵引用的主要目的是在一个对象所占的内存被实际回收之前得到通知，从而可以进行一些相关的清理工作。幽灵引用在使用方式上与之前介绍的两种引用类型有很大不同：首先幽灵引用在创建时必须提供一个引用队列作为参数；其次幽灵引用对象的 `get` 方法总是返回 `null`，因此无法通过幽灵引用来获取被引用的对象。

幽灵引用在使用的时候只能通过引用队列来操作。幽灵引用的最大优势在于引用对象被添加到队列中的时机。Java 语言提供了对象终止 (finalization) 机制来允许开发人员提供对象被销毁之前的自定义处理逻辑。`Object` 类提供了 `finalize` 方法来添加自定义的销毁逻辑。如果一个类有特殊的销毁逻辑，可以覆写 `finalize` 方法。从功能上来说，`finalize` 方法与 C++ 中的析构函数比较相似，但是 Java 采用的是基于垃圾回收器的自动内存管理机制，所以 `finalize` 方法在本质上不同于 C++ 中的析构函数。当垃圾回收器发现没有引用指向一个对象时，会调用这个对象的 `finalize` 方法。通常在这个方法中进行一些资源释放和清理的工作，比如关闭文件、套接字和数据库连接等。由于 `finalize` 方法的存在，虚拟机中的对象一般处于三种可能的状态。第一种是可达状态，当有引用指向该对象时，该对象处于可达状态。根据引用类型的不同，有可能处于强引用可达、软引用可达或弱引用可达状态。第二种是可复活状态，如果对象的类覆写了 `finalize` 方法，则对象有可能处于该状态。虽然垃圾回收器是在对象没有引用的情况下才调用其 `finalize` 方法，但是在 `finalize` 方法的实现中可能为当前对象添加新的引用。因此在 `finalize` 方法运行完成之后，垃圾回收器需要重新检查该对象的引用。如果发现新的引用，那么对象会回到可达状态，相当于该对象被复活；否则对象会变成不可达状态。当对象从可复活状态变成可达状态之后，对象会再次出现没有引用存在的情况。在这个情况下，`finalize` 方法不会被再次调用，对象会直接变成不可达状态，也就是说，一个对象的 `finalize` 方法只会被调用一次。第三种是不可达状态，在这个状态下，垃圾回收器可以自由地释放对象所占用的内存空间。对象终止机制在第 10 章进行详细介绍。

软引用和弱引用在其可达状态达到时就可能被添加到对应的引用队列中。也就是说，当一个对象变成软引用可达或弱引用可达的时候，指向这个对象的引用对象就可能被添加到引用队列中。在添加到队列之前，垃圾回收器会清除掉这个引用对象的引用关系。当软引用和弱引用进入队列之后，对象的 `finalize` 方法可能还没有被调用。在 `finalize` 方法执行之后，该对象有可能重新回到可达状态。如果该对象回到了可达状态，而指向该对象的软引用或弱引用对象的引用关系已经被清除，那么就无法再通过引用对象来查找这个对象。而幽灵引用则不同，只有在对象的 `finalize` 方法被运行之后，幽灵引用才会被添加到队列中。与软引用和弱引用不同的是，幽灵引用在被添加到队列之前，垃圾回收器不会自动清除其引用关系，需要通过 `clear` 方法来显式地清除。当幽灵引用被清除之后，对象就进入了不可达状态，垃圾回收器可以回收其内存。当幽灵引用

被添加到队列之后，由于 PhantomReference 类的 get 方法总是返回 null，程序也不能对幽灵引用所指向的对象进行任何操作。这就避免了 finalize 方法可能会出现的对象复活的问题。幽灵引用是作为一个通知机制而存在的。程序应该在得到通知之后进行与当前对象相关的清理工作。

代码清单 7-10 给出了幽灵引用对象的使用示例。在类 ReferencedObject 中覆写 finalize 方法来提供自定义的销毁逻辑。这里只是简单地在控制台输出提示信息。在使用幽灵引用队列时，通过队列的 poll 方法来进行轮询。如果队列为空，就通过 System.gc 方法来建议垃圾回收器运行。运行示例之后会发现 finalize 方法中输出的消息总是最早出现，这说明当幽灵引用进入队列之后， finalize 方法已经被运行过了。如果改用软引用或弱引用来进行相同试验，会发现多次运行的结果并不一致，这是因为软引用和弱引用进入队列的时机和 finalize 方法的调用之间并没有必然的先后关系。

代码清单 7-10 幽灵引用对象的使用示例

```

public class UseReferenceQueue {
    private static class ReferencedObject {
        protected void finalize() throws Throwable {
            System.out.println("finalize 方法被调用。");
            super.finalize();
        }
    }

    public void phantomReferenceQueue() {
        ReferenceQueue<ReferencedObject> queue = new ReferenceQueue<>();
        ReferencedObject obj = new ReferencedObject();
        PhantomReference<ReferencedObject> phantomRef = new PhantomReference<ReferencedObject>(
            obj, queue);
        obj = null;
        Reference<? extends ReferencedObject> ref = null;
        while ((ref = queue.poll()) == null) {
            System.gc();
        }
        phantomRef.clear();
        System.out.println(ref == phantomRef); // 值为 true
        System.out.println(" 幽灵引用被清除。");
    }
}

```

如果希望在一个对象的内存被回收之前进行某些清理工作，那么相对于使用 finalize 方法来说，使用幽灵引用是更好的选择。幽灵引用避免了 finalize 方法可能造成对象复活的问题，减少了开发时可能出现的错误。不过幽灵引用的使用比 finalize 方法要复杂得多。最主要的问题是从引用队列中获取幽灵引用之后，无法获取其指向的对象，也就无法对这个对象进行操作。幽灵引用本身只作为一个通知机制存在，必须存在其他指向

此对象的引用。因此，相对于使用幽灵引用，开发人员更倾向于谨慎使用 finalize 方法。只要 finalize 方法的实现避免了对象复活的问题，就不失为一个不错的选择。

一个比较实际的幽灵引用的应用是在虚拟机内存总量受限的情况下。当内存总量受限时，可能需要等待一个占用内存空间较大的对象被回收之后再申请新的内存空间。通过这种方式，可以把程序中某部分占用的内存控制在一定的范围之内。代码清单 7-11 给出了一个示例。类 PhantomAllocator 用来分配一个字节数组供调用者使用。当每次分配新的字节数组时，会先确保之前分配的内存空间被成功释放。在每次分配新的字节数组之前，引用队列的 remove 方法会处于阻塞状态，直到有新的引用对象被添加到队列中。remove 方法返回之后，之前的字节数组的内存已经可以被释放，通过调用 System.gc 方法要求垃圾回收器马上回收这些内存。等内存回收之后再创建新的字节数组，创建一个幽灵引用指向新的字节数组，并与引用队列关联起来。

代码清单 7-11 使用幽灵引用的内存分配方式

```

public class PhantomAllocator {
    private byte[] data = null;
    private ReferenceQueue<byte[]> queue = new ReferenceQueue<byte[]>();
    private Reference<? extends byte[]> ref = null;
    public byte[] get(int size) {
        if (ref == null) {
            data = new byte[size];
            ref = new PhantomReference<byte[]>(data, queue);
        }
        else {
            data = null;
            System.gc();
            try {
                ref = queue.remove();
                ref.clear();
                ref = null;
                System.gc();
                data = new byte[size];
                ref = new PhantomReference<byte[]>(data, queue);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        return data;
    }
}

```

在上面的代码中，通过 “`data = null`” 来清除 PhantomAllocator 类对象本身对字节数组的强引用。在进行测试的时候，可以进行多次字节数组分配操作，同时使用工具来观察程序所占用的堆内存的情况。实际的运行结果是，程序的堆内存的占用量的峰值会维持在一个相对稳定的值。

7.3.6 引用队列

在前面的小节中对引用队列做了一些简要的说明。引用队列的主要作用是作为一个通知机制。当对象的可达状态发生变化时，如果程序希望得到通知，可以使用引用队列。当从引用队列中获取了引用对象之后，不可能再获取所指向的具体对象。对于软引用和弱引用来说，在被放入队列之前，它们的引用关系就已经被清除了；而幽灵引用的 get 方法总是返回 null。下面通过分析 Java 标准库中的 WeakHashMap 类来说明引用队列在实际中的应用。

WeakHashMap 类的实现中的 Entry 类表示哈希表中包含的条目。Entry 类继承自 WeakReference 类，这样就可以比较方便地处理从引用队列中获取的引用对象，因为引用对象本身代表了哈希表中的条目。Entry 类中也包含了与条目相关的基本信息，包括键对象、值对象和引用队列的引用等。Entry 类作为一个弱引用，指向的是条目的键对象，而值对象仍然由强引用来指向。在程序的运行过程中，WeakHashMap 类的对象中的某些条目的键对象可能变成了弱引用可达的状态。垃圾回收器会清除这些弱引用并将其放入 WeakHashMap 类的对象的引用队列中。WeakHashMap 类的对象需要在合适的时机检查这个队列中包含的引用对象。由于引用对象本身就是 Entry 类的对象，因此可以直接把引用对象从 WeakHashMap 类的对象中删除。

删除引用队列中的条目是通过 WeakHashMap 类中的私有方法 expungeStaleEntries 来完成的。代码清单 7-12 给出了核心的方法实现，即通过 poll 方法检查队列中的引用对象，并将该引用对象转换成 Entry 类的对象来处理。在 WeakHashMap 类中，大部分涉及条目的方法的实现都会直接或间接地调用 expungeStaleEntries 方法来处理 WeakHashMap 类的对象中引用队列所包含的条目。这也是 expungeStaleEntries 方法使用 poll 来检查引用队列的原因。由于对 expungeStaleEntries 方法的调用会比较频繁，会对 WeakHashMap 类中正常操作的性能产生影响，因此使用非阻塞式的 poll 方法是个更好的选择。

代码清单 7-12 在 WeakHashMap 类中删除可被回收的条目的示例

```
private void expungeStaleEntries() {
    for (Object x; (x = queue.poll()) != null; ) {
        synchronized (queue) {
            Entry<K,V> e = (Entry<K,V>) x;
        }
    }
}
```

鉴于 WeakHashMap 类的实现机制，当其中包含的条目的键对象变成弱引用可达之后，在下一次对 WeakHashMap 类的对象进行操作时，这些键对象对应的条目才会被删除，所以必须注意这种情况，如果对 WeakHashMap 类的对象的操作比较少，那么使用 WeakHashMap 类的对象也会出现某些键对象的存活时间过长的情况。

7.4 Java本地接口

Java虚拟机为Java开发人员屏蔽了底层的实现细节，使得开发人员不用考虑底层操作系统的差异性。不过在某些应用程序中，免不了要直接与底层操作系统上的原生代码进行交互。Java本地接口（Java Native Interface，JNI）的作用是提供一个标准的方式让Java程序通过虚拟机与原生代码进行交互。与原生代码进行交互的动机主要有下面几个。

第一个动机是从性能的角度出发。Java语言从其运行速度上来说，在大多数方面是慢于底层操作系统上原生的C和C++等语言的。这主要是由于Java虚拟机这个中间层次的存在。如果完全用Java语言实现的性能无法达到程序的预期要求，可以选择把部分重要且耗时的代码用C或C++来实现。

第二个动机是满足某些特殊的需求。Java平台提供的标准类库的功能很强大，包括了在开发中可能遇到的大部分功能。但是仍然有一些功能无法用标准API来实现，主要是一些与底层硬件平台直接交互的功能。Java虚拟机没有把这一部分功能暴露给运行在其上的程序。如果需要这方面的功能，那么只能使用原生代码来进行开发。

最后一个动机是与已有的使用原生代码编写的程序之间进行集成。如果Java程序需要与底层操作系统上由C和C++语言开发的程序进行交互，那么可以使用JNI。

7.4.1 JNI基本用法

JNI所包含的内容比较复杂，下面通过几个具体的示例来介绍JNI的使用。JNI的一个重要使用场景是提高程序的性能相关。当对程序中关键部分的性能要求比较高的时候，可以使用C和C++代码来实现。使用原生代码实现的方法被称为原生方法，由native关键词来声明。如代码清单7-13所示，sqrt方法由原生代码来实现。在NativeMath类的静态初始化块中通过System.loadLibrary方法加载名为NativeMath的原生代码库。这个原生代码库中包含了sqrt方法的实现。原生方法与Java接口中的方法或抽象类中的抽象方法一样，只包含方法声明，没有相关的实现。程序中的其他部分可以用正常的方法调用原生方法，比如参数传递和返回值使用等都与正常的方法相同。当虚拟机在执行原生方法时，会尝试在已经加载的原生代码库中查找原生方法的对应实现。在查找到对应的实现方法之后，虚拟机会负责进行参数传递、实际方法调用和返回值传递等工作。

代码清单7-13 包含原生方法的Java类

```
public class NativeMath {
    static {
        System.loadLibrary("NativeMath");
    }

    public native double sqrt(double value);
}
```

在编写了 Java 源代码之后，下一步要编写实现原生方法的 C/C++ 代码。Java 提供的命令行工具 javah 根据 Java 源代码生成 C/C++ 代码所需的头文件。对于原生方法，头文件中会包含相关的方法声明与其对应。代码清单 7-14 给出了生成的头文件的内容。在这个头文件中，先引用 JDK 中 include 目录下的 jni.h 文件。这个由 JDK 提供的头文件中包含了实现原生方法的 C/C++ 代码中所需的类型和方法声明。原生方法 sqrt 对应的 C/C++ 方法是 Java_com_java7book_chapter7_jni_NativeMath_sqrt。这个方法的名称是在“Java_”的前缀后加上类名和方法名而得到的。在方法的参数方面，原生方法中声明的参数会被自动映射到用来实现的 C/C++ 方法中。C/C++ 方法中最前面两个参数是标准的。第一个参数是指向 JNIEnv 接口的指针，通过这个指针可以访问包含一系列 JNI 方法的方法表。在原生方法的实现中需要利用这些 JNI 方法来访问和操作虚拟机中的数据结构。第二个参数取决于原生方法的类型。对于类中的实例方法，这个参数表示的是方法调用时的当前对象，相当于 this 关键词的含义；对于类中的静态方法来说，这个参数表示的是方法所在的 Java 类的对象。这两个标准参数之后是原生方法中映射过来的参数。

代码清单 7-14 通过 javah 工具生成的 JNI 头文件

```
#include <jni.h>
#ifndef _Included_com_java7book_chapter7_jni_NativeMath
#define _Included_com_java7book_chapter7_jni_NativeMath
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:      com_java7book_chapter7_jni_NativeMath
 * Method:     sqrt
 * Signature:  (D)D
 */
JNIEXPORT jdouble JNICALL Java_com_java7book_chapter7_jni_NativeMath_sqrt
(JNIEnv *, jobject, jdouble);

#ifdef __cplusplus
}
#endif
#endif
```

Java 与 C/C++ 的语法不同，在参数的类型上需要进行一定的映射。对于 Java 中的基本类型，如 int 和 float 等，在原生方法的实现中，是直接映射到对应的类型上的。例如，Java 中 double 类型的对应类型是 jdouble。而 Java 中的引用类型则映射到一个 C/C++ 中的指针上。这个指针所指向的内容是虚拟机内部的结构，其具体的内容对使用者来说是不透明的。原生方法的实现代码利用 JNI 提供的方法来对这个指针所指向的结构进行操作。

代码清单 7-13 中的 Java 源程序只用到了基本类型 double，其对应的 C/C++ 方法

的实现比较简单，只需要调用 C/C++ 中的 sqrt 方法即可。代码清单 7-15 给出了相应的 C++ 实现。

代码清单 7-15 NativeMath 类的 sqrt 方法的 C++ 实现

```
#include "com_java7book_chapter7_jni_NativeMath.h"
#include <math.h>

JNIEXPORT jdouble JNICALL Java_com_java7book_chapter7_jni_NativeMath_sqrt (JNIEnv
    *env, jobject obj, jdouble value) {
    return sqrt(value);
}
```

在编写了相应的 C++ 程序之后，可以使用 C++ 编译器来进行编译和链接，得到所需的原生代码库文件。在编译时需要把 JDK 中的 include 目录添加到编译器的搜索路径中，否则无法找到对应的类型声明。在 Windows 平台上，编译和链接的结果是动态链接库 DLL 文件。这个 DLL 文件的名称要与 System.loadLibrary 方法中使用的名称相同。在运行 Java 程序时，需要通过启动参数 “-Djava.library.path” 来指定 DLL 文件所在的目录，使 System.loadLibrary 方法可以找到所需的 DLL 文件。

在使用 JNI 时最容易遇到的问题是出现 java.lang.UnsatisfiedLinkError 错误。造成这个错误的原因通常有两个：第一个原因是找不到包含原生方法实现的代码库，比如找不到 DLL 文件。在出现这个错误时，通常只需要使用虚拟机启动参数 “-Djava.library.path” 显式指定代码库所在的目录即可。如果不显式指定，那么虚拟机会在某些预设目录中进行搜索。第二个原因是无法在代码库中找到原生方法对应的实现方法。一般来说，使用 javah 工具从 Java 源代码中生成的头文件中包含了对应方法的正确声明。只需要在 C/C++ 源代码中复制该方法的声明即可。如果发生了无法找到对应方法的情况，那么很可能是 C/C++ 编译器在编译和链接时改变了所生成的方法的名称。C/C++ 编译器的这个特性被称为名称装饰 (name decoration)，其主要目的是解决相同名称的编程实体的解析问题。不同的名称空间中可能包含相同名称的实体，如名称相同的方法。在进行链接的时候，链接器需要足够的信息来区分这些名称相同的实体。典型的做法是在生成的实体名称上添加一些额外的信息作为装饰来进行区分。比如，在 Windows 平台上使用 MinGW 的 GCC 编译器对代码清单 7-15 中的 C++ 代码进行编译和链接之后，所得的 DLL 文件中的 Java_com_java7book_chapter7_jni_NativeMath_sqrt 方法的实际名称是 Java_com_java7book_chapter7_jni_NativeMath_sqrt@8。方法名称的后缀 “@8” 表示的是方法的所有参数所占用的字节数。方法名称的不一致造成虚拟机在运行 Java 程序时找不到原生方法的对应实现，因此出现 UnsatisfiedLinkError 错误。对于方法名称后多余的 @ 后缀，只要在使用 GCC 编译器时加上参数 “-Wl,--kill-at” 就可以将其去掉。在产生 UnsatisfiedLinkError 错误时，可以从错误的详细信息中得到具体的说明。如果是由于第二个原因造成的，可以通过工具来查看代码库中方法的名称来定位问题的原因。比如在

Windows 平台上，可以使用 DLL Export Viewer[⊖]来查看 DLL 中导出的方法的详细信息。

代码清单 7-15 只说明了 Java 中的基本类型在 JNI 中的使用方式。当原生方法的声明中包含了 Java 标准库的类或自定义的类时，利用 C/C++ 来实现的方式会有所不同。比如，在 NativeMath 类中添加一个新的原生方法 size，其声明如代码清单 7-16 所示。方法 size 的参数的类型是自定义的表示矩形的 Rectangle 类，该类中包含了用来获取矩形区域的宽度和高度的 getWidth 和 getHeight 方法。

代码清单 7-16 NativeMath 类中 size 方法的声明

```
public native double size(Rectangle rectangle);
```

再次使用 javah 工具来更新头文件之后，原生方法 size 的 C++ 实现如代码清单 7-17 所示。先使用 JNIEnv 中的 GetObjectClass 方法查找到参数 rectangle 对象所对应的 Java 类，再通过 GetMethodID 方法找到 Rectangle 类中包含的 getWidth 和 getHeight 方法。最后通过 CallDoubleMethod 方法来调用这两个方法，可以得到进行计算所需的矩形的宽度和高度的值。

代码清单 7-17 NativeMath 类中 size 方法的实现

```
JNIEXPORT jdouble JNICALL Java_com_java7book_chapter7_jni_NativeMath_size (JNIEnv
    *env, jobject obj, jobject rectangle) {
    jclass cls = env->GetObjectClass(rectangle);
    jmethodID getWidthMid = env->GetMethodID(cls, "getWidth", "()D");
    double width = env->CallDoubleMethod(rectangle, getWidthMid);
    jmethodID getHeightMid = env->GetMethodID(cls, "getHeight", "()D");
    double height = env->CallDoubleMethod(rectangle, getHeightMid);
    return width * height;
}
```

从代码清单 7-17 中可以看出，对参数对象的使用方式类似于在 Java 中使用反射 API 来操作对象。由于 jobject 表示的只是一个不透明的结构，因此所有的操作都需要以反射的方式来进行。在 JNIEnv 中，GetObjectClass 方法的作用相当于 Java 中 Object 类中的 getClass 方法，GetMethodID 方法的作用相当于 Java 中 Class 类的 getMethod 方法，只不过在查找时使用的方式不同。GetMethodID 方法要求提供方法类型在字节代码中的表现形式作为查找的条件，例如，“()D” 表示不包含参数，返回值类型为 double。CallDoubleMethod 方法的作用相当于 Java 中 Method 类的 invoke 方法。在 JNIEnv 中，对于 Java 中返回值类型、方法类型和调用方式不同的方法，都有与之对应的调用方法，比如调用 Java 中返回值类型为 boolean 的静态方法应该使用 CallStaticBooleanMethod 方法。

[⊖] DLL Export Viewer 工具的网址是：http://www.nirsoft.net/utils/dll_export_viewer.html。

7.4.2 Java程序中集成C/C++代码

使用JNI的另外一个常见的情景是与已有的C/C++程序进行集成。在编写Java程序之前，就已经有了可以使用的原生代码库。这个原生代码库可能是程序的一部分，也可能是底层操作系统自带的。这些原生代码库的特点是在实现的时候并没有考虑与Java虚拟机的集成，因此也没有使用与JNI相关的内容。在使用这样的原生代码库时，可能会需要一个中间的原生代码库作为桥梁。这个原生代码库作为Java程序中原生方法的实现，负责实际调用时的参数类型转换和返回值传递等工作。考虑实现一个Java程序，希望使用底层操作系统上原生的消息提示对话框作为给用户提示信息的方式。使用第5章介绍的AWT或Swing用户界面库做到这一点并不难。不过下面介绍的是如何使用JNI来实现这个功能。以Windows平台为例，系统动态链接库user32.dll中包含的MessageBox方法可以提供所需的功能。

要使用user32.dll中的方法，一种做法是使用另外一个原生代码库作为桥梁。这种使用方式类似于上面介绍的第一种使用JNI的方式，即先对原生方法进行声明，再通过javah工具生成头文件，最后利用C++代码编写相关的实现。包含原生方法的Java类的基本声明如代码清单7-18所示。

代码清单7-18 消息提示Java类的基本声明

```
public class MessageBox {
    static {
        System.loadLibrary("MessageBox");
    }

    public native int show(String text, String caption);
}
```

相关的C++代码实现如代码清单7-19所示，其中也显示了JNI的原生方法实现中字符串的使用方式。Java中原生方法声明中的String类型会被转换成JNI中的jstring类型。在C++代码中，需要先通过JNIEnv中的GetStringUTFChars方法把jstring类型转换成C++中可以使用的char类型。转换之后可以直接调用user32.dll中的MessageBox方法。需要格外注意的是，在使用完从jstring类型转换后的char类型的字符串之后，需要通过ReleaseStringUTFChars方法来释放相关的内存，否则会出现内存泄露。这是因为C++中并没有Java语言中的自动内存管理机制。

代码清单7-19 消息提示的C++实现

```
JNIEXPORT jint JNICALL Java_com_java7book_chapter7_jni_MessageBox_show (JNIEnv
    *env, jobject obj, jstring text, jstring caption) {
    const char *text_str = env->GetStringUTFChars(text, NULL);
    const char *caption_str = env->GetStringUTFChars(caption, NULL);
    int result = MessageBox(0, text_str, caption_str, MB_OK | MB_ICONINFORMATION);
    env->ReleaseStringUTFChars(text, text_str);
```

```

    env->ReleaseStringUTFChars(caption, caption_str);
    return result;
}

```

这种方式的不足之处在于开发人员不但需要了解 C/C++ 语言，还需要了解 JNI 实现中的类型转换等细节。对于纯 Java 背景的开发人员来说，要调用一个已有代码库中的方法，可以使用 JNA（Java Native Access）库[⊖]。JNA 库简化了对原生代码库的调用方式，使用纯 Java 就可以实现对代码库的调用。JNA 在实现方式上采用了代理设计模式。对于一个原生代码库，JNA 可以创建出相应的代理对象。该代理对象中包含了原生代码库中已有方法所对应的 Java 方法，在程序中使用此代理对象即可。对代理对象中方法的调用会在进行自动类型转换之后传递给原生代码库中的相应方法，调用的返回结果也在类型转换后被正确返回。在对原生代码库中的方法进行调用时，方法查找和类型转换等操作都是由 JNA 负责完成的，对程序代码是透明的。

在 JNA 中，可以用继承自 com.sun.jna.Library 的接口来表示原生代码库。接口中的每个方法都对应原生代码库中的方法。接口中的方法声明需要与原生方法库中的方法相匹配。代码清单 7-20 中的 User32Library 接口表示 Windows 平台上的 user32.dll。在接口中只声明了程序中所需的 MessageBoxA 方法，此方法对应于 user32.dll 中的 MessageBoxA 方法。在 user32.dll 中，MessageBoxA 方法的声明是 int MessageBoxA(HWND hWnd, LPCTSTR lpText, LPCTSTR lpCaption, UINT uType)，在映射到 Java 接口中的方法之后，HWND 类型映射到 com.sun.jna.Pointer 类，其他的参数类型直接映射到 Java 中的基本类型。在得到表示原生代码库的 Java 接口之后，可以使用 JNA 中的 com.sun.jna.Native 类的 loadLibrary 方法来得到一个实现此接口的代理对象。在调用原生方法代码库的时候直接使用代理对象即可。

代码清单 7-20 表示 user32.dll 的 User32Library 接口

```

public interface User32Library extends Library {
    User32Library INSTANCE = (User32Library) Native.loadLibrary("user32",
        User32Library.class);

    int MessageBoxA(Pointer handle, String text, String caption, int type);
}

```

代码清单 7-21 给出了 JNA 代理对象的使用方式。对于与代码清单 7-18 中类型相同的 show 方法，使用 JNA 提供的代理对象的实现如代码清单 7-21 所示。调用的方式很简单，直接使用 User32Library 接口中的 MessageBoxA 方法即可。最后一个参数 0x40 的含义是代码清单 7-19 中的“MB_OK | MB_ICONINFORMATION”表达式的实际值。这里的 show 方法不需要声明为原生方法，只是一个普通的 Java 方法。

[⊖] JNA 项目的网址是 <https://github.com/twall/jna>。

代码清单 7-21 使用 JNA 的消息提示

```
public class MessageBoxJna {
    public int show(String text, String caption) {
        return User32Library.INSTANCE.MessageBoxA(null, text, caption, 0x40);
    }
}
```

在使用 JNA 时，开发人员不需要直接编写 C/C++ 代码，也不需要生成额外的原生代码库来调用已有的原生代码库中的方法。JNA 所提供的代理对象可以负责完成相关的工作。JNA 的不足之处在于调用原生代码库中的方法时的性能会受到一定的影响。另外在传递参数时，也不能使用在 C/C++ 头文件中定义的常量。

注意 原生代码库中的某些方法可能是宏定义，如 user32.dll 中的 MessageBox 方法实际上是一个宏定义。实际存在的方法是 MessageBoxA 和 MessageBoxW，分别对应使用 ASCII 和 Unicode 编码的情况。JNA 无法直接翻译宏定义，需要手动选择正确的方法。如果在调用的时候出现找不到方法的错误，可以查询原生代码库的头文件或文档说明，以检查该方法名称是否为宏定义。

如果可以找到原生代码库对应的头文件，那么可以使用 JNAerator 工具[⊖]从头文件中生成 JNA 中 Library 的子接口的代码，其中包含了头文件中的全部方法。使用 JNAerator 工具的好处是可以避免手动映射过程中的错误，更加高效。

7.4.3 在 C/C++ 程序中启动 Java 虚拟机

上面介绍的这两个使用 JNI 的场景都是以 Java 程序为主体的。用户使用的是 Java 程序，只是程序的部分组件由原生代码来实现。下面介绍的第三个场景以 C/C++ 程序为主体，把 Java 虚拟机嵌入到原生代码中。通过这种方式，可以在 C/C++ 程序中调用由 Java 编写的组件。实际上，Java 虚拟机本身是通过原生代码来实现的，要在 C/C++ 程序中调用并非难事。在下面的示例中，C++ 程序需要获取一个网页的内容之后再进行处理。由于 Java 程序中包含的网络相关的类库比较丰富，获取网页的功能由 Java 程序来实现，而对网页内容的处理则由 C++ 代码来完成。对此，创建了一个 Java 类 com.java7book.chapter7.jni.WebPageDownloader，其中的静态方法 getContent 用来根据网页的 URL 返回其中的内容。在 C++ 程序中启动 Java 虚拟机如代码清单 7-22 所示，方法 JNI_CreateJavaVM 用来创建并启动一个 Java 虚拟机。在创建时需要提供虚拟机的启动参数。这里只提供了 Java 类文件所在的路径，用于查找所需的 Java 类。Java 虚拟机创建成功之后，JavaVM 表示虚拟机对象，可以利用 JNIEnv 中的方法来查找 Java 类并调用其中的方法。在使用完虚拟机之后，需要通过 DestroyJavaVM 方法来销毁虚拟机。在链接 C++ 代码时需要把 JDK 的 lib 目录添加到搜索路径中。

[⊖] JNAerator 工具的网址是 <http://code.google.com/p/jnaerator/>。

代码清单 7-22 在 C++ 程序中启动 Java 虚拟机

```
#include <jni.h>

int main()
{
    JNIEnv *env;
    JavaVM *jvm;
    JavaVMInitArgs vm_args;
    JavaVMOption options[1];
    options[0].optionString =
        "-Djava.class.path=C:\\\\java7\\\\code\\\\chapter7\\\\bin";
    vm_args.version = JNI_VERSION_1_6;
    vm_args.options = options;
    vm_args.nOptions = 1;
    vm_args.ignoreUnrecognized = 0;
    jint res;
    res = JNI_CreateJavaVM(&jvm, (void**)&env, &vm_args);
    if (res < 0)
    {
        return res;
    }
    jclass clsDownloader = env->FindClass("com/java7book/chapter7/jni/
        WebPageDownloader");
    jmethodID midGetContent = env->GetStaticMethodID(clsDownloader, "getContent",
        "(Ljava/lang/String;)Ljava/lang/String;");
    jstring content = (jstring) env->CallStaticObjectMethod(clsDownloader,
        midGetContent, env->NewStringUTF("http://www.baidu.com"));
    if (env->ExceptionOccurred())
    {
        printf("Error occurs when downloading content.");
        jvm->DestroyJavaVM();
        return -1;
    }
    const char *text_str = env->GetStringUTFChars(content, NULL);
    printf(text_str);
    jvm->DestroyJavaVM();
}
```

上面的代码也说明了 C++ 代码对 Java 代码中产生的异常的处理方式。在 C++ 代码中，可以通过 JNIEnv 中的 ExceptionOccurred 方法来检查 Java 代码中是否产生了异常，如果产生了异常，那么会执行相应的异常处理逻辑。

7.5 HotSpot 虚拟机

目前有不同厂商或机构开发的 Java 虚拟机实现。所有这些虚拟机实现都遵守 Java 虚拟机规范，但是所适用的情况有所不同。Java SE 7 的 OpenJDK 实现使用的是 Oracle 的 HotSpot 虚拟机。HotSpot 虚拟机从 JDK 1.3 开始是 Sun 提供的默认虚拟机实现。大

部分开发人员在使用 Java SE 7 时都使用默认虚拟机。本节会介绍 HotSpot 虚拟机的相关细节，主要目的是帮助开发人员更好地了解和使用 HotSpot 虚拟机。

7.5.1 字节代码执行

Java 语言的源代码在经过 Java 编译器的编译之后，被转换成 Java 字节代码。虚拟机在执行字节代码时一般采用的是即时编译的方式，即所谓的 Just-in-time (JIT) 编译方式。虚拟机会在运行过程中把字节代码中的指令直接转换成底层操作系统平台上的原生指令。由于虚拟机所理解的只是 Java 的字节代码格式，因此这样的转换是必需的。不过 JIT 编译方式有一些性能方面的问题，会降低程序的执行效率。HotSpot 虚拟机采用了自适应的优化技术来解决 JIT 编译方式的性能问题。这项优化技术的关键是利用程序运行中的热点 (hot spot)，这也是 HotSpot 虚拟机名称的由来。

程序运行过程中的一个重要特征是程序局部性，即在程序的运行过程中，小部分代码会占据比较多的运行时间。这小部分的代码被称为程序运行中的热点。这也是“80-20 原则”的体现，程序中 20% 的代码会占据 80% 的运行时间。如果把这重要的 20% 的代码的优化工作做好，就可以节省大量的时间。在程序刚开始运行的时候，HotSpot 虚拟机会分析程序的字节代码，以找出其中的热点，并对这些热点进行复杂的优化工作。随着程序的运行，其中的热点可能会发生变化。虚拟机会随时监控程序的运行状态，以追踪其中的热点。利用热点的好处在于不需要对程序中的所有代码都进行复杂的优化，这样可以把时间用在对重要代码的优化上，减少代码优化的时间开销。

HotSpot 虚拟机的另外一个优化措施是方法内联。虚拟机在运行过程中的大部分时间都花费在方法调用上。方法内联的作用是把被调用的方法中的代码直接内联到调用的地方。通过这种方式可以减少方法调用，同时为虚拟机提供更多可以优化的代码。

7.5.2 垃圾回收

垃圾回收器对虚拟机上运行的 Java 程序有着非常重要的作用，也会对程序的性能产生不同的影响。垃圾回收器在实现上要考虑的因素非常多，并不存在一个完美的算法能够适合不同 Java 程序的运行情况。垃圾回收器的实现算法更多的是对各种不同因素的权衡和取舍，而权衡的依据是程序本身的特性和需求。为了适合不同的程序的运行情况，HotSpot 虚拟机提供了多种不同的垃圾回收算法。这些算法的具体细节虽然各不相同，但是都采用了分代回收的方式。

1. 分代回收方式

分代回收是垃圾回收中的一种常见算法。这种算法的特点是把内存划分成不同的世代 (generation)，分别对应虚拟机中存活时间不同的对象。进行这种划分的依据是从对象存活时间得出的统计规律。在一般程序的运行过程中，大部分对象的存活时间比较短。比如，在一个方法内部创建的局部变量，方法执行完并退出之后，这些变量所引用

的对象的内存就不再需要，可以被回收。只有少量对象存活的时间会比较长，还有极少数对象的存活时间与程序本身一样长。对于存活时间不同的对象，可以采用不同的回收策略。对于包含存活时间较短的对象的内存空间，其中所包含的存活对象较少，可被回收的内存区域较多，而且状态变化比较快，因此，对这个世代的内存进行回收的频率比较高，速度比较快。而对于存活时间较长的对象，回收的频率可以比较低。

在 HotSpot 虚拟机中，一般把内存划分成 3 个世代：年轻、年老和永久世代。大部分对象所需内存的分配是在年轻世代区域进行的。当垃圾回收器运行时，年轻世代中的很多对象可能已经不再存活，可以直接被回收。而有些对象可能仍然处于存活状态。某些对象可能在经过若干个垃圾回收操作之后，仍然处于存活状态。对于这些仍处于存活状态的对象，垃圾回收器会把这些对象移动到年老世代的内存区域。永久世代中包含的是 Java 虚拟机自身运行所需的对象。年轻世代被进一步划分成伊甸园（eden）和两个存活区（survivor space）。大部分对象的内存分配都是在伊甸园中进行的。由于伊甸园的内存空间较小，因此某些所需内存较大的对象无法直接在伊甸园中进行分配，而直接在年老世代中进行。两个存活区中总有一个是空白的。在对年轻世代进行垃圾回收时，先把伊甸园中的存活对象复制到当前空白的存活区中，接着对另外一个非空白存活区中的存活对象进行处理。如果对象的存活时间较短，那么同样将其复制到空白的存活区中；如果对象存活时间已经较长，那么将其复制到年老世代区域。在复制到空白的存活区的过程中，如果发现该存活区已满，就把这些存活对象直接复制到年老世代区域。经过这两次复制之后，就可以把伊甸园和非空白存活区中的内容直接全部清空。因为这两个区域中的对象要么不再存活，要么已经被复制到了其他内存区域中。在完成垃圾回收之后，下次的内存分配可以继续从空白的伊甸园开始进行，两个存活区的作用也发生了交换。

对于年老和永久世代内存区域，通常采用另外一种回收算法。这种算法分 3 个具体的步骤，其名称也来源于这 3 个步骤，称为标记–清除–压缩（mark-sweep-compact）算法。第一个步骤的作用是扫描整个内存区域，把当前仍然存活的对象标记出来；第二个步骤的作用则是清理内存区域，清除垃圾；第三个步骤的作用是压缩整个内存区域，把存活对象所占的内存都移动到内存区域的起始位置，使内存中可用区域是连续的。经过压缩之后，在年老和永久世代中进行内存分配就变得很容易，只需从可用区域的开头位置进行分配即可。

2. 解决永久世代内存不足

垃圾回收器在每次回收操作时所处理的内存世代区域并不相同。一次较小的回收操作只会对年轻世代进行回收处理，一次较大的回收操作会处理年老世代，而完全的回收操作会对整个内存区域进行处理。这些回收操作的运行频率也并不相同。在垃圾回收器的运行过程中，最常回收的是年轻世代的内存区域；对于年老世代的内存区域的回收操作要少得多；而对于永久世代来说，回收的操作就更少。永久世代中存放的一般是虚拟机运行所需的元数据，包括加载的 Java 类等。如果程序中加载的类比较多，可能会造成

永久世代的空间不够，而出现 OutOfMemoryError 错误。错误的提示信息一般是“java.lang.OutOfMemoryError: PermGen space”。如果出现这样的错误，可以通过启动参数“-XX:MaxPermSize”为永久世代指定一个更大的内存容量。不过需要注意的是，虚拟机对字符串的内部化处理，有可能会造成永久世代的内存不足。

由于 Java 中的 String 类的对象是不可变的，为了提高字符串比较时的性能，Java 提供了一种字符串内部化（intern）的机制。这种机制的实现方式是在虚拟机中缓存 String 类的对象，当需要使用包含相同字符串的 String 类对象的时候，可以直接使用缓存中的对象。这样只需要简单地使用“==”操作符就可以比较两个 String 对象是否相等，而不需要使用更加耗时的 equals 方法。虚拟机会对 Java 源代码中的字符串字面量进行内部化处理，同时也可以使用 String 类的 intern 方法来得到一个缓存的 String 类的对象。这种内部化机制在 Java 7 中被用到了数值型的基本类型上，具体的细节可以参见第 6 章。

不过需要注意的一点是，不同虚拟机在实现字符串内部化机制时有很大不同。在一些虚拟机实现中，所缓存的 String 对象是保存在永久世代中的。如果使用了太多的内部化 String 对象，对象又都处于被引用的状态，就会导致永久世代的内存不足，出现 OutOfMemoryError 错误。永久世代的内存容量一般都比较小，比较容易出现内存不足的问题。在代码清单 7-23 中，通过循环来不断地生成包含随机内容的字符串，并调用 String 类的 intern 方法来缓存这些字符串。使用 List 接口的作用是保持对创建出来的 String 对象的引用，防止被垃圾回收器处理。

代码清单 7-23 由于字符串内部化机制造成内存不足的示例

```
public class StringIntern {
    private List<String> list = new ArrayList<String>();
    public void useInternString() {
        Random random = new Random();
        for (int i = 0; i < 200; i++) {
            char[] data = new char[128 * 1024];
            for (int j = 0; j < data.length; j++) {
                data[j] = (char) random.nextInt(32768);
            }
            list.add((new String(data)).intern());
        }
    }
}
```

上面的代码在不同的虚拟机上的运行结果不同。在 OpenJDK 的 Java 7 虚拟机上不会出现 OutOfMemoryError 错误。通过检查垃圾回收器的输出信息可以发现，大部分的内存占用发生在年老世代中，而不是永久世代中。在 JDK 6 更新 21 的虚拟机中运行时，会出现由于永久世代内存不足而造成的 OutOfMemoryError 错误。因此，如果程序中使用了大量的字符串字面量或是 String 类的 intern 方法，有可能会产生兼容性的问题。程序在某个虚拟机上可以正确运行，换到另外一个虚拟机上可能会出现

OutOfMemoryError 错误。如果字符串是由用户或其他程序提供的，那么一定不要调用这些字符串对象的 intern 方法，因为有可能会使程序被恶意攻击，比如一个 Web 应用接受用户提供的字符串作为输入。在通过 servlet 请求获取表示参数值的字符串对象之后，不应该调用该字符串对象的 intern 方法。如果调用了 intern 方法，攻击者可以通过发送一些内容很长的字符串的方式进行攻击，当虚拟机尝试缓存这个对象时，可能会因为 OutOfMemoryError 错误而退出。

另外一个会造成永久世代内存不足的原因是加载的 Java 类过多。虚拟机中当前加载的类的元数据是保存在永久世代中的。如果加载的类过多，会导致永久世代内存不足而引发 OutOfMemoryError 错误。代码清单 7-24 给出了一个示例，通过 ASM 工具创建出一个简单的 Java 类的字节代码，将其保存在一个字节数组中。由于 LoadClass 类继承自 Java 标准库中的 java.lang.ClassLoader 类，可以直接调用 defineClass 方法从包含字节代码的数组中得到表示 Java 类的 Class 类的对象。在调用了 defineClass 方法之后，Java 类的元数据被保存在永久世代中。运行之后会发现，当加载的类的数量达到一定值时，虚拟机会抛出 OutOfMemoryError 错误来说明永久世代区域内存不足。

代码清单 7-24 由于加载的 Java 类过多造成内存不足的示例

```
public class LoadClass extends ClassLoader {

    public void loadManyClasses() {
        int num = 50000;
        String classNamePrefix = "ManyClass";
        for (int i = 0; i < num; i++) {
            String className = classNamePrefix + i;
            createAndLoadClass(className);
        }
    }

    private void createAndLoadClass(String className) {
        ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES);
        cw.visit(V1_7, ACC_PUBLIC | ACC_SUPER, className, null,
                "java/lang/Object", null);
        cw.visitEnd();
        byte[] classData = cw.toByteArray();
        this.defineClass(className, classData, 0, classData.length);
    }
}
```

3. 选择垃圾回收算法

以上介绍的是分代回收的方式，HotSpot 虚拟机提供的垃圾回收算法都使用了这种方式，不过在其他方面仍然有很多不同。在为程序选择合适的垃圾回收算法时，需要综合考虑多个因素。

第一个需要考虑的因素是使用串行或并行的回收方式。如果虚拟机运行的硬件平台只有一个CPU，那么垃圾回收工作只能由这个CPU按照串行的方式来完成。如果存在多个CPU的情况，那么可以利用这些CPU并行地进行回收工作。

第二个需要考虑的因素是对回收过程中发现的存活对象的处理方式。第一种做法是进行压缩，即把存活对象移动到内存区域的一端，使内存中的空闲区域变成连续的。这样的好处是分配内存时的速度会非常快，只需要从空闲区域的端点开始计算，判断是否有足够的空闲区域满足分配请求也会很容易。不足之处是需要花费额外的时间来进行存活对象的移动操作。第二种做法是不进行压缩，即存活对象仍然被保留在原始位置。这样的好处是垃圾回收操作可以很快完成。不足之处是分配内存的过程会比较慢，这是因为内存中的可用空间不是连续的，需要逐个查找可用内存块来寻找满足当前分配请求的空闲块。第三种做法是进行复制，即把存活对象复制到另外一个区域之中。这样做的好处是在复制操作完成之后，之前的内存区域就变成了全部可用的内存空间，内存分配速度会很快。不足之处是需要花费额外的时间来进行复制操作。总的来说，这三种方式都要求在垃圾回收操作所花费的时间及后续的内存分配操作所花费的时间这两者之间进行权衡。

理解这些不同的因素及如何在这些因素之间进行取舍，对大多数用户和开发人员来说是一件很困难的事情，因此，对虚拟机的默认垃圾回收机制的选择就显得尤为重要。绝大多数程序在运行时都不会对默认的垃圾回收机制进行修改。

HotSpot虚拟机会根据硬件平台的能力来选择最适合的垃圾回收方式。硬件平台根据其性能被划分成服务器级别和非服务器级别两类。服务器级别指的是有两个或两个以上的CPU以及2GB以上物理内存的硬件平台。对于服务器级别的机器，默认使用并行回收方式，堆内存大小的初始值和最大值分别是物理内存的1/64和1/4；对于非服务器级别的机器，默认使用串行回收方式，堆内存大小的初始值和默认值分别是4MB和64MB。

如果虚拟机默认的垃圾回收的相关设置不能满足要求，可以通过修改启动参数的方式进行调整。可以使用的启动参数比较多，不过对于一般用户来说，最简单的启动参数是指明垃圾回收机制所要达到的目标。这种目标驱动的方式对于用户来说更加直接。垃圾回收机制的目标包括降低回收时造成的程序的停顿时间、提高吞吐量和降低程序的内存占用量等。第一个目标是降低由于垃圾回收造成的程序的停顿时间。在进行垃圾回收时，一个不可避免的问题是当前运行程序的停顿。过长的停顿时间会对程序造成比较大的影响。某些对实时性要求很高的程序更加不允许出现较长时间的停顿。如果对停顿时间有严格的要求，那么可以通过虚拟机的启动参数“-XX:MaxGCPauseMillis”来指定最长的停顿时间。虚拟机会调整垃圾回收器的其他参数来保证这个目标得以满足。第二个目标是提高程序运行的吞吐量。吞吐量指的是虚拟机花费在垃圾回收操作上的时间和程序实际运行时间的比例。虚拟机应该保证把尽可能多的时间花费在程序运行上，同时保证正常的垃圾回收操作不受影响。通过启动参数“-XX:GCTimeRatio”可以指定垃圾回

收时间所占的比例，如使用“-XX:GCTimeRatio=49”就指明垃圾回收时间所占的比例是 $1/(1+49)=2\%$ 。最后一个目标是降低虚拟机占用的内存总量。当前面两个目标满足时，垃圾回收器会降低堆内存的大小，直到出现了前面两个目标不满足的情况。这样的作用是在满足前两个目标的前提下，尽可能地减少程序所占用的内存。

如果默认的垃圾回收算法和目标驱动的配置方式都不能满足需求，可以通过启动参数直接指定所要使用的垃圾回收算法。这种配置方式要求对垃圾回收算法有比较深入的了解，一般只建议有经验的开发人员使用。

第一种可用的垃圾回收方式是串行回收，这也是非服务器级别的硬件平台上的默认回收方式。可以通过启动参数“-XX:+UseSerialGC”来显式指定。

第二种可用的垃圾回收方式是并行回收，这也是服务器级别的硬件平台上的默认回收方式。可以通过启动参数“-XX:+UseParallelGC”来显式地指定。并行回收方式相对于串行回收方式的改进体现在对年轻世代进行回收时会利用多个CPU来并行完成，可以减少垃圾回收操作造成的停顿时间和整体的回收操作所占用的时间，提高吞吐量。而对于年老世代，并行回收的方式与串行回收的方式是一样的。

由于并行回收的方式对于年老世代并没有采用并行的做法，因此性能会受到一定的影响。并行压缩回收方式改进了这一点，对于年老世代的回收也采用并行的处理方式，可以通过启动参数“-XX:+UseParallelOldGC”来使用这种回收方式。这种回收方式会把年老世代的内存区域划分成若干个固定大小的子区域。对每个子区域以并行的方式同时进行存活对象的标记工作。由于标记工作是并行进行的，执行的效率会比较高。在完成标记工作之后，下一步是找出这些子区域中需要进行压缩操作的部分。这是因为某些子区域中存活对象所占的比例可能比较大，不需要进行压缩。不需要压缩的子区域一般都出现在年老世代的某一端。找到了需要进行压缩的子区域之后，就可以通过复制和移动存活对象等操作来压缩子区域，使存活对象密度较高的子区域都出现在年老世代内存区域的某一端，方便后续的内存分配。

如果希望程序运行时因垃圾回收操作而造成的停顿时间比较短，那么可以使用并发标记清除的回收方式。这种回收方式对年轻世代的做法和并行回收方式一样，而对于年老世代的处理方式则比较复杂，分成几个具体的阶段。第一个阶段是初始的标记阶段。这个阶段会先暂停程序的运行，再标记出从程序的代码中直接可达的存活对象。完成这个阶段之后，程序可以继续运行。与此同时，垃圾回收器会从上一阶段标记出的存活对象出发，递归地标记可达的其他存活对象。这个标记过程与程序的运行并发进行。完成这个并发的标记阶段之后，下一个阶段会再次暂停程序的运行。这是因为在上一个标记阶段，由于程序仍然在运行，对象的存活状态可能已经发生了变化。这一个阶段会再次暂停程序，并对这些发生变化的对象重新进行标记。重新标记完成之后，就可以继续程序的运行，同时并发地对标记过程中发现的垃圾进行回收处理。这种回收方式所造成的程序停顿时间比较短，但是会要求比较大的堆内存。另外，这种回收方式不会进行内存区域的压缩操作，使得后续的内存分配操作比较耗时。通过启动参数

“-XX:+UseConcMarkSweepGC”可以指定使用这种回收方式。

Java 7 中添加了一个新的垃圾回收方式，即垃圾优先方式 (garbage first)，简称为 G1。G1 所采用的回收方式不同于之前已有的其他回收算法。G1 也采用了把内存区域划分成不同世代的做法，对年轻世代进行更加频繁的回收操作。但是 G1 并没有把年轻世代和年老世代所占用的内存区域从物理上分隔开来，而是把整个堆内存划分成大小相同的子区域，每个子区域的内容可以属于年轻世代或年老世代。垃圾回收的过程是先找出需要进行回收的子区域和接收待回收子区域中存活对象的其他子区域。回收的过程是并行进行的，把待回收子区域中的存活对象移动到用来接收的其他子区域中，再把原始的子区域清空。回收操作比较多地发生在年轻世代的子区域中，对于年老世代的子区域也会顺带进行处理。启用 G1 回收器需要使用参数 “-XX:+UnlockExperimentalVMOptions -XX:+UseG1GC” 来完成。G1 的主要目标是替换前面介绍的并发标记清除回收方式。

7.5.3 启动参数

HotSpot 虚拟机在启动时可以指定很多不同的参数。这些参数中有些是标准参数，是所有平台上的虚拟机都支持的。另外一些则是非标准的或试验性质的参数，不是所有平台上的虚拟机都支持。这些试验性质的参数的功能可能并不稳定，会在后续的版本中发生变化。

HotSpot 虚拟机支持的标准参数并不是很多：“-classpath” 和 “-cp” 都可以用来指定程序运行时的类路径 (classpath)；“-D” 用来设置系统属性的值；“-ea” 和 “-da” 分别用来启用和禁用程序中的断言；“-esa” 和 “-dsa” 分别用来启用和禁用系统类中的断言；“-verbose” 用来要求虚拟机输出一些详细的信息，如 “-verbose:class” 用来输出加载 Java 类时的信息，“-verbose:gc” 用来在垃圾回收器运行时输出相关信息，“-verbose:jni” 用来在使用原生方法时输出相关信息。

在启动参数中以 “-X” 开头的参数是非标准参数，不是所有平台上的虚拟机都支持。这些参数中比较常见的是设置虚拟机堆内存大小的参数，其中 “-Xms” 用来设置堆内存的初始值，“-Xmx” 用来设置堆内存的最大值。

以 “-XX” 开头的参数的实现不是很稳定，一般不推荐使用。在前面介绍虚拟机的垃圾回收机制时，启用特定垃圾回收方式的参数都是以 “-XX” 开头的，说明对这些参数的使用需要特别注意。这些参数根据数据类型的不同可以分成三类：第一类是布尔型的参数，可以通过 “-XX:+<option>” 的形式来打开，通过 “-XX:-<option>” 的形式来关闭。第二类是数值型的参数，可以通过 “-XX:<option>=<number>” 的形式来指定。在指定数值大小时可以使用 “k”、“m” 和 “g” 等单位，分别表示千、兆和千兆。最后一类是字符串类型的参数，可以通过 “-XX:<option>=<string>” 的形式来指定。

根据作用的不同大致可以将这些参数分成三类。第一类是与虚拟机行为相关的。前面介绍的指定垃圾回收方式的参数都属于这一类。除此之外，还包括一些有用

的参数，如“-XX:+DisableExplicitGC”用来禁止通过 `System.gc` 方法来显式要求运行垃圾回收器。如果不希望程序直接影响垃圾回收器的运行，那么可以打开此功能。“-XX:+ScavengeBeforeFullGC”用来要求垃圾回收器在对整个内存空间进行回收之前，先回收年轻世代的内存空间。这个功能默认是打开的。“-XX:+UseGCOverheadLimit”用来限制虚拟机花费在垃圾回收上的时间。如果虚拟机把大量的时间花费在垃圾回收上，就说明虚拟机的内存过小，不足以支持程序的运行。如果这样的情况持续发生一段时间，虚拟机会抛出 `OutOfMemoryError` 错误。这个功能默认是打开的，如果不希望存在这样的限制，可以关闭此功能。

第二类参数是与性能优化相关的。这些参数主要用来处理字符串的优化，以及调整虚拟机堆内存各个部分的大小。下面介绍一下这些参数中比较重要的几个。“-XX:+AggressiveOpts”用来启用在当前虚拟机版本中处于试验性质的优化策略，这些优化策略有可能在以后的虚拟机版本中被默认启用。启用这些优化策略通常会提升程序的运行性能，但是有可能造成一些程序运行时的错误。“-XX:+UseStringCache”用来启用对经常使用的字符串进行缓存的功能，可以提升性能。“-XX:+UseCompressedStrings”用来启用对字符串的压缩处理。如果字符串中仅包含 ASCII 字符，会使用 `byte[]` 而不是 `char[]` 来表示字符串，这样可以节省占用的内存空间。“-XX:+OptimizeStringConcat”用来启用对字符串连接操作的优化功能。“-XX:NewRatio”用来指定年轻世代和年老世代所占的内存的比例。“-XX:NewSize”用来指定年轻世代所占的内存大小。“-XX:MaxPermSize”用来指定永久世代所占内存的最大值。

第三类参数是与程序调试相关的，可以控制虚拟机在运行时输出一些调试信息。这些参数中比较重要的包括：“-XX:+CITime”用来控制输出 JIT 编译器所花费的时间；“-XX:+PrintGC”用来控制当垃圾回收器运行时输出与回收操作相关的信息；“-XX:+PrintGCDetails”的作用类似于“-XX:+PrintGC”，只是输出的信息更加详细；“-XX:+PrintGCTimeStamps”也用来输出与垃圾回收相关的信息，只不过会在信息中包含时间戳；“-XX:+TraceClassLoading”用来控制 Java 类被加载时输出相关的调试信息；“-XX:+TraceClassUnloading”用来控制 Java 类被卸载时输出相关的调试信息。

7.5.4 分析工具

在大多数时候，Java 程序本身并不需要对 Java 虚拟机有太多的了解。但是程序在运行过程中可能出现一些与虚拟机相关的错误，比如内存不足或线程死锁等问题。当出现这样的问题时，需要通过一些相关的工具来对虚拟机进行分析，找出问题的原因。对某些程序来说，可能需要对虚拟机的一些性能指标进行监视，以避免一些潜在的问题。Java 虚拟机所提供的分析工具比较丰富，下面逐一进行介绍。

1. 命令行和图形化工具

首先可以使用的工具是虚拟机在运行过程中输出的相关调试信息。在上一节中介绍

了可以通过启动参数来控制虚拟机输出与垃圾回收和类加载相关的调试信息。

其次可以使用的是 JDK 中自带的命令行工具。第一组工具 jmap 和 jhat 用来对共享对象映射关系和堆内存进行分析。工具 jmap 可用于查看正在运行的 Java 程序、虚拟机的核心转储（core dump）文件，以及远程调试服务器上的共享对象的映射关系和堆内存的占用情况。使用 jmap 工具时，对于正在运行的 Java 程序，需要指定 Java 进程的标识符；对于虚拟机的核心转储文件，需要指定文件的路径；对于远程调试服务器，需要指定服务器的主机名或 IP 地址。在使用 jmap 时，可以指定一些选项来声明需要查看的内容。下面对 jmap 工具的介绍都以查看正在运行的 Java 程序为例来进行。

如果不指定任何选项，那么默认的行为是在控制台输出程序中共享对象的映射关系。对于每个虚拟机加载的共享对象，输出它在内存中的起始地址、映射空间的大小和共享对象所在的文件路径。在 Windows 操作系统中，被共享的对象通常是虚拟机所加载的操作系统中的 DLL 文件。如果添加 “-heap” 选项，那么会在控制台输出 Java 程序当前的堆内存占用情况的详细信息，其中包括所用的垃圾回收方式、堆内存的参数配置信息，以及当前堆内存中各个世代的不同区域的内存占用情况。如果添加 “-finalizerinfo” 选项，那么会在控制台输出 Java 程序中当前正在等待终止的对象的数量。添加 “-histo” 选项会在控制台输出堆内存占用情况的统计数据，对于每个 Java 类，输出其实例对象的个数和占用的内存。当使用 “-histo:live” 选项时，只会统计当前存活对象的信息。选项 “-permstat” 用来在控制台输出永久世代中包含的数据的统计信息。该信息主要由两部分组成，第一部分是被内部化的字符串的个数和占用的内存空间大小，第二部分是与类加载器相关的内容，包括每个类加载器已经加载的类的个数和占用的空间。选项 “-dump” 可以把当前程序的堆内存信息转储到文件中，再用 jhat 工具进行查看。

工具 jhat 用来对虚拟机堆内存的转储文件进行分析。其独特之处在于它会启动一个 Web 服务器。开发人员可以通过浏览器来查看转储文件中的各种内容。这种网页形式的查看方式既方便又直观。在使用 jhat 时需要指定转储文件的路径。在 jhat 工具运行起来后，可以通过默认的 7000 端口来访问 Web 界面。有多种方式可以得到堆内存的转储文件，如使用 jmap、jconsole 和 hprof 等工具，也可以在启动虚拟机时指定参数 “-XX:+HeapDumpOnOutOfMemoryError”。当虚拟机出现 OutOfMemoryError 错误时，会自动生成堆内存的转储文件。

比如，使用 “jmap -dump:format=b,file=heap.bin 2456” 命令可以把进程标识符为 2456 的虚拟机的堆内存转储到 heap.bin 文件中，再使用 “jhat heap.bin” 命令就可以启动 jhat 工具。通过浏览器就可以查看相关的信息，其中包括所加载的所有 Java 类、每个 Java 类的实例对象、堆中内存分布的统计数据和对象终止的情况等。对于每个对象实例，可以列出其中包含的数据成员。在对象的可达性方面，可以列出从一个对象实例可达的其他对象，还可以列出引用了一个对象实例的其他对象。前面介绍过，虚拟机在判断对象是否存活时从一些根对象开始遍历。使用 jhat 可以查看从根对象到当前对象的引用关系路径。通过这个功能可以发现程序中存在的内存泄露问题。如果发现某个应该被

回收的对象仍然出现在内存中，可以检查它的引用关系路径，从而可以发现是由于哪个对象的引用关系仍然存在而造成当前对象无法被回收。

在 jhat 的 Web 界面中可以使用一种对象查询语言（Object Query Language, OQL）来查询堆内存的转储文件中的对象。OQL 语言的语法类似 SQL，按照“select…from…where”的结构来组织，其中表达式使用的是 JavaScript 语法。select 子句中的内容是生成查询结果的 JavaScript 表达式，而 from 子句中的内容是待查询对象所在的 Java 类的全名，where 子句中的内容则是用来进行过滤的条件表达式。例如，OQL 语句“select s from java.lang.String s where s.count >= 200”用来查询所有长度大于等于 200 的字符串对象。

除了堆内存的分析工具之外，还有其他一些工具可供使用，包括用来列出所有 Java 虚拟机进程标识符的 jps 命令行工具，查看虚拟机中线程堆栈信息的 jstack 工具，查看虚拟机中系统属性值和启动参数的 jinfo 工具，以及显示虚拟机运行性能相关指数的 jstat 工具。

以上介绍的这些工具都是命令行工具，通常把输出结果直接显示在控制台。JDK 也提供了图形化界面的工具来监控虚拟机的状态。第一个可用的工具是 jconsole。通过 jconsole 可以连接到正在运行的本地或远程 Java 程序上，获取程序所在虚拟机的内存、Java 类和线程的相关信息。这些信息以图形化的方式显示出来，并且实时更新。

从前面的介绍中可以看到，JDK 所包含的工具种类繁多，所提供的功能也不同。这么多的工具分开使用也不方便。从 JDK 6 更新 7 开始，新的图形化工具 Java VisualVM 被加入进来，通过 jvisualvm 命令可以启动这个工具。VisualVM 的作用在于把各种不同工具的功能整合到了一起，用一个统一的图形化的方式展现出来。除此之外，VisualVM 本身是一个可扩展的平台，可以利用社区贡献的插件来增强 VisualVM 本身对虚拟机的监控能力。VisualVM 除了可以查看虚拟机的各种信息之外，还可以进行程序的性能取样和剖析，找出影响性能的瓶颈所在。

2. JMX

上面介绍的这些工具都是独立运行的，并没有提供相关的 API 接口供程序使用。从 Java SE 5.0 开始，Java 平台提供了一套完整的 API 来对运行的 Java 程序及虚拟机本身进行监控和管理。这一套 API 由多个部分组成，其中最重要的组成部分是 Java 管理扩展（Java Management Extensions, JMX）API。JMX API 是 Java 平台上进行资源监控和管理的标准 API，可以用来对应用程序、设备、服务和 Java 虚拟机本身进行监控和管理。JMX API 在很多情况下都可以发挥作用，包括在运行时动态获取和更新程序的配置信息、收集程序运行过程中的统计数据以及当程序内部状态发生变化或出现错误时发出相关通知等。JMX API 的基础概念是 MBean。一个 MBean 表示的是可以被管理的命名资源。每个 MBean 都提供一个管理接口允许第三方来使用。这个管理接口的内容包括可以获取和修改值的命名属性、可以调用的命名方法，以及可以发出的事件通知。

MBean 本身只是一个接口，需要由被监控和管理的资源提供相关的实现。所有的 MBean 实现被注册到 MBean 服务器上。使用者通过名称在 MBean 服务器上查找所需 MBean 的实现。在得到了实现对象之后，可以通过 MBean 的管理接口来调用其中的方法。Java 虚拟机本身提供了一个 MBean 服务器，可以在其上注册相关 MBean。与 Java 虚拟机本身的监控和管理相关的 MBean 都注册在该 MBean 服务器上。相关 MBean 的接口都在 `java.lang.management` 包中定义，可以监控和管理的资源包括虚拟机中的缓冲区、类加载系统、代码编译、垃圾回收器、内存、底层操作系统、虚拟机运行时、线程和日志记录器等。通过 `java.lang.management.ManagementFactory` 类中的工厂方法可以得到所需的管理不同资源的 MBean 接口的实现，比如对线程的监控和管理是由接口 `java.lang.management.ThreadMXBean` 来表示的。如果需要获取当前虚拟机上的活动线程的数目，可以通过 `ManagementFactory` 类中的 `getThreadMXBean` 方法先得到 `ThreadMXBean` 接口的实现对象，再调用该对象的 `getThreadCount` 方法。对监控和管理其他资源的 MBean 的使用也是类似的。

利用对虚拟机平台的监控和管理的能力，可以根据虚拟机的运行情况来动态调整程序本身的运行逻辑。比如，某个程序中提供了后台运行的定时备份能力，程序中有一个线程在后台运行，并定期把程序中的重要数据备份到磁盘上，整个备份过程需要占用一定的内存空间，当虚拟机的内存空间不足时，备份过程应该暂停运行以免出现 `OutOfMemoryError` 错误而导致虚拟机退出，通过使用监控和管理虚拟机内存的 `java.lang.management.MemoryPoolMXBean` 接口就可以实现这样的功能。代码清单 7-25 给出了后台备份线程的实现示例。虚拟机通常把内存划分成多个区域，典型的是划分成多个世代。`ManagementFactory` 类的 `getMemoryPoolMXBeans` 方法返回的是所有内存区域的列表。每个区域都有对应的进行监控和管理的 `MemoryPoolMXBean` 接口的实现对象。这里所关心的是年老世代所在的内存区域，因此使用的是名为“Tenured Bean”的 `MemoryPoolMXBean` 接口的实现对象。通过 `setUsageThreshold` 方法可以设置内存使用量的阈值。当超过这个阈值时，`isUsageThresholdExceeded` 方法会返回 `true`，说明剩余内存量已经不足。

代码清单 7-25 考虑内存剩余量的备份任务的示例

```
public class BackupTaskRunnable implements Runnable {
    private MemoryPoolMXBean poolBean;

    public BackupTaskRunnable() {
        init();
    }

    private void init() {
        List<MemoryPoolMXBean> beans = ManagementFactory
            .getMemoryPoolMXBeans();
        for (MemoryPoolMXBean bean : beans) {
            if (bean.getName().contains("Tenured")) {
                poolBean = bean;
                break;
            }
        }
    }

    public void run() {
        while (true) {
            if (poolBean.getUsageThresholdExceeded()) {
                // 备份操作
            }
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
            }
        }
    }
}
```

```

        if ("Tenured Gen".equals(bean.getName())) {
            poolBean = bean;
            break;
        }
    }
    poolBean.setUsageThreshold(10 * 1024 * 1024);
}

public void run() {
    while (true) {
        if (poolBean.isUsageThresholdExceeded()) {
            System.out.println(" 内存不足，暂停备份任务。 ");
        } else {
            System.out.println(" 执行备份任务。 ");
        }
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
}

```

上面的代码使用的是轮询的方式来判断内存使用量是否超过阈值。还有一种做法是使用 MBean 的事件监听机制。某些 MBean 在内部状态发生变化或出现错误的时候，会产生相应的事件通知。MBean 的使用者可以在事件通知上注册监听器。当事件发生的时候，所注册的监听器方法会被调用。这种方式类似于 Web 和桌面应用开发中的用户界面组件上的事件处理方式。

与虚拟机内存相关的 `java.lang.management.MemoryMXBean` 可以在内存区域的占用空间大小超过指定阈值时发出事件通知。对此事件感兴趣的程序只需要实现 `javax.management.NotificationListener` 接口并注册到 `MemoryMXBean` 对象上即可。代码清单 7-26 给出了相关的代码实现。

代码清单 7-26 使用事件监听机制的内存监控示例

```

private static class MemoryListener implements NotificationListener {
    public void handleNotification(Notification notification,
                                   Object handback) {
        String type = notification.getType();
        if (type.equals(MemoryNotificationInfo.MEMORY_THRESHOLD_EXCEEDED)) {
            System.out.println(" 内存占用量超过阈值。 ");
        }
    }
}

public void addListener() {

```

```
MemoryMXBean mbean = ManagementFactory.getMemoryMXBean();
NotificationEmitter emitter = (NotificationEmitter) mbean;
MemoryListener listener = new MemoryListener();
emitter.addNotificationListener(listener, null, null);
}
```

7.5.5 Java虚拟机工具接口

前面介绍了使用虚拟机中 JMX API 中的 MBean 来对虚拟机进行监控和管理。不过可以通过 MBean 来进行监控和管理的内容只是虚拟机所提供的功能中的很小一部分，其中大部分功能是无法通过 Java API 来访问的。这主要是因为对于运行在虚拟机上的 Java 程序来说，其他的监控和管理功能可以使用的场景非常少。虚拟机提供的所有监控和管理功能，虽然不能通过 Java API 来直接使用，但是可以通过 C/C++ 原生代码来使用。J2SE 5.0 对虚拟机中与监控和管理相关的功能进行整合，形成了标准的 Java 虚拟机工具接口（Java Virtual Machine Tools Interface，JVM TI）。

1. 基本使用

与 JMX API 不同，JVM TI 主要是供开发、调试和监控工具使用的，并不是针对普通的应用程序而设计的。使用 JVM TI 可以查看和控制当前在虚拟机上运行的程序的状态。通过使用 JVM TI，可以开发出各种面向虚拟机上运行程序的工具，实现包括程序调试、性能分析、运行状态监控、线程分析和代码覆盖率分析等功能。不过这些工具只能使用原生代码来实现。

使用 JVM TI 开发的工具被称为虚拟机上的代理程序（agent）。在虚拟机启动时，代理程序也会被加载和运行。当 Java 程序在虚拟机上运行时，代理程序可以通过 JVM TI 查看和控制 Java 程序中的相关状态。当虚拟机退出时，代理程序也会相应退出。代理程序和虚拟机在同一个进程中运行。代理程序本身是底层操作系统平台上的一个原生代码库，例如在 Windows 平台上是一个 DLL 文件。在虚拟机启动时，通过参数“-agentlib”或“-agentpath”来指定原生代码库的名称或绝对路径。如果指定的是代码库的名称，则由虚拟机根据名称自动进行搜索。在指定代理程序名称或路径的时候，可以指定额外的配置参数。例如，“-agentlib:myAgent=option1,option2”指明了代理程序的名称是 myAgent，同时传入两个额外的参数 option1 和 option2。

2. 使用案例

下面通过一个具体的案例来说明 JVM TI 的用法。这个案例中的工具的作用是统计 Java 程序在运行过程中不同方法的调用次数。方法调用次数的信息对于分析程序中的性能瓶颈是很重要的。代理程序可以通过两种方式来使用 JVM TI 提供的功能。第一种是方法调用，来进行直接的状态查询和更新；第二种是注册事件监听器，用来在虚拟机中特定的事件发生时执行相关的逻辑。对于这个案例来说，主要采用的是注册事件监听

器的方式。当虚拟机开始执行某个方法时，会产生相关的事件，只需要在这个事件的处理方法中根据当前方法的名称进行统计即可。

在代理程序的 C++ 代码中需要添加对 JDK 中 include 目录下的 jvmti.h 头文件的引用。这个头文件中包含 JVM TI 中的类型声明和方法定义。代理程序被加载之后，其中的 Agent_OnLoad 方法会被调用。Agent_OnLoad 方法也是代理程序运行的起点。Agent_OnLoad 方法的声明是固定的，如代码清单 7-27 所示。JavaVM 类型的参数表示的是当前的虚拟机对象，而参数 options 则是之前提到的通过虚拟机启动参数传递给代理程序的额外参数。在 Agent_OnLoad 方法被调用时，虚拟机本身的初始化还没有完成，因此在 Agent_OnLoad 方法中所能执行的操作是有限的。通常在 Agent_OnLoad 方法中注册事件监听器。

虚拟机上的所有事件默认是禁用的。要启用某个事件，通常要三步：首先启用某些事件所依赖的虚拟机本身的支持能力。JVM TI 中所规范的监控和管理的功能并不是在所有虚拟机实现上都可用的。这些功能分为必需和可选两类。所有虚拟机都应该实现必需的功能，而可选功能在某些虚拟机上可能并不支持。本案例所需的在进入方法调用时产生的事件 JVMTI_EVENT_METHOD_ENTRY 是一个可选功能。对于可选功能，需要通过 JVM TI 的 AddCapabilities 方法显式地启用。在启用了事件相关的功能之后，接着需要启用该事件。只需要使用 SetEventNotificationMode 方法把事件的状态设为 JVMTI_ENABLE 即可。最后一步是注册事件的处理方法。事件发生时的回调方法由 JVM TI 中的 jvmtiEventCallbacks 结构来表示，在其中可以设置所监听的各种事件对应的回调方法。本案例只对 JVMTI_EVENT_METHOD_ENTRY 事件感兴趣，因此只通过 MethodEntry 属性设置了该事件的回调方法，其值是一个函数指针。在填充完 jvmtiEventCallbacks 结构之后，使用 SetEventCallbacks 方法来注册事件监听器。

代码清单 7-27 代理程序的 Agent_OnLoad 方法

```
JNIEXPORT jint JNICALL Agent_OnLoad(JavaVM *jvm, char *options, void *reserved)
{
    jvmtiEnv *jvmti;
    jvm->GetEnv((void **)&jvmti, JVMTI_VERSION_1_0);
    jvmtiCapabilities capa;
    memset(&capa, 0, sizeof(jvmtiCapabilities));
    capa.can_generate_method_entry_events = 1;
    jvmti->AddCapabilities(&capa);
    jvmtiEventCallbacks callbacks;
    callbacks.MethodEntry = &Method_Entry;
    jvmti->SetEventNotificationMode(JVMTI_ENABLE, JVMTI_EVENT_METHOD_ENTRY, NULL);
    jvmti->SetEventCallbacks(&callbacks, (jint)sizeof(callbacks));
    return JNI_OK;
}
```

代码清单 7-27 中的 jvmtiEnv 类型的变量表示的是 JVM TI 的运行环境。代理程序通

过此变量与JVM TI进行交互。对于方法进入事件JVMTI_EVENT_METHOD_ENTRY的处理函数如代码清单7-28所示。当虚拟机调用某个方法的时候，代理程序中的Method_Entry函数会被调用。通过函数的参数可以获取表示当前调用方法的jmethodID对象。Method_Entry函数中的逻辑比较简单，通过jvmtiEnv提供的功能来获取方法所在的类的名称和方法的名称，并把统计数据保存在一个哈希表中。这里需要注意的一点是，通过调用jvmtiEnv中方法所得到的字符串，都需要在使用完毕之后调用Deallocate方法来释放。

代码清单7-28 方法进入事件处理函数

```

map<string, int> methodCountMap;

void JNICALL Method_Entry(jvmtiEnv *jvmti_env, JNIEnv* jni_env, jthread thread,
    jmethodID method)
{
    char *method_name;
    jclass cls;
    char *class_signature;
    jvmti_env->GetMethodName(method, &method_name, NULL, NULL);
    jvmti_env->GetMethodDeclaringClass(method, &cls);
    jvmti_env->GetClassSignature(cls, &class_signature, NULL);
    char name[256] = {0};
    strcpy(name, class_signature);
    strcat(name, method_name);
    methodCountMap[name]++;
    jvmti_env->Deallocate((unsigned char*)method_name);
    jvmti_env->Deallocate((unsigned char*)class_signature);
}

```

与Agent_OnLoad方法相对应，当虚拟机完成程序的执行并准备退出之前，会调用代理程序中的Agent_OnUnload方法。代码清单7-29给出了案例中的Agent_OnUnload方法的实现，其作用是把统计出来的方法调用次数的数据输出到一个文件中。

代码清单7-29 代理程序的Agent_OnUnload方法

```

JNIEXPORT void JNICALL Agent_OnUnload(JavaVM *vm)
{
    ofstream file;
    file.open ("C:\\method_trace.txt", ios::out);
    for( map<string, int>::iterator iter = methodCountMap.begin(); iter != methodCountMap.end(); ++iter)
    {
        file << (*iter).first << "\t" << (*iter).second << endl;
    }
    file.close();
}

```

在编译和链接 C++ 程序之后，可以得到代理程序的原生代码库。在启动虚拟机执行任何 Java 程序的时候都可以加载该代理程序。当程序运行完毕之后，可以从输出的文件中看到程序中所有方法的调用次数的统计信息。

7.6 小结

Java 虚拟机作为一个运行平台，为 Java 程序提供了一个简洁和统一的运行环境。一般开发人员需要对虚拟机有适度的了解。适度的含义是既不被虚拟机的底层实现细节所干扰，又可以有效地利用虚拟机本身的能力。本章所介绍的内容围绕这样的主题展开，即介绍 Java 开发人员应该知道的虚拟机的相关内容。Java 中的引用类型可以让程序与垃圾回收器进行交互，把程序在对象内存管理中的需求传递给垃圾回收器。JNI 可以让 Java 程序和用 C/C++ 语言编写的原生代码之间进行交互，不仅可以提高性能，还可以进行应用集成。HotSpot 虚拟机作为 Java SE 7 的 OpenJDK 实现中的默认虚拟机，其特征值得开发人员进行必要的了解。本章对 HotSpot 虚拟机的垃圾回收机制、启动参数和分析工具等重要内容进行了介绍。

第 8 章 Java 源代码和字节代码操作

经过前面章节的介绍，相信读者对 Java 源代码、Java 字节代码和 Java 虚拟机之间的关系有了一定的了解。一般的流程是这样的：由开发人员编写 Java 源代码，再通过 Java 编译器编译成字节代码，最后由虚拟机来运行。通过 JDK 中的命令行工具 `javac` 可以启动编译器来编译 Java 源代码，以生成字节代码。使用命令行工具 `java` 或 `javaw` 可以启动虚拟机来运行字节代码。如果使用的是集成开发环境（IDE），那么相关的步骤更加简单，只需要通过 IDE 的用户界面来操作即可。字节代码作为一个中间层次，为 Java 平台增加了很多的灵活性。字节代码的格式是公开的。通过工具可以绕开 Java 编译器直接生成字节代码，也可以对已有的字节代码进行修改。通过操作字节代码，可以实现很多强大的功能，并用简洁的方式解决复杂的问题。

前面提到的从 Java 源代码到字节代码再到虚拟机运行的过程中，其中的每一步都有不同的实现方式。比如，可以不使用命令行工具 `javac`，而直接在运行时动态编译 Java 源代码；字节代码可以不通过编译器来生成，而是使用工具来动态创建；在字节代码被虚拟机执行之前，可以通过修改字节代码的内容来改变程序的行为。Java 开发人员对 Java 源代码的语法应该都很熟悉，下面先从开发人员可能比较陌生的字节代码格式开始介绍。

8.1 Java 字节代码格式

大多数开发人员对 Java 字节代码的格式可能会有点陌生。字节代码一般出现在 Java 源代码编译之后生成的 `class` 文件中。每个 `class` 文件中包含了单个类或接口的定义。Java 源文件中的内部类会被编译到单独的 `class` 文件中。实际上，字节代码并不是只存在于 `class` 文件中，还可以通过网络从远程服务器下载，或者由程序在运行时动态生成。所以，字节代码更加准确的说法是包含单个 Java 类或接口定义的字节流，通常用 `byte[]` 来表示。

Java 字节代码是一种二进制格式，其具体的格式在 Java 虚拟机规范中定义。使用二进制编辑器打开一个 `class` 文件，可以看到字节代码的内容。要理解字节代码格式，可以参考对应的 Java 源代码的组织结构。一个 Java 类从源代码的角度来说，包含类本身的信息及类中包含的域和方法的信息。字节代码中也包含同样的信息，并且以松散的结构进行组织。为了节省空间，字节代码对 Java 类中常量的存储进行了优化。了解字节代码的格式，是对字节代码进行操作的基础。工具无法为开发人员屏蔽与字节代码相关的所有细节。为了避免涉及过多的细节，本节只对重要的内容进行介绍。在介绍字节代码

格式的时候，采用的是 Java 虚拟机规范中的描述方式。

在介绍字节代码的格式之前，先说明一下 Java 中的类或接口、域和方法等在字节代码中的表现形式。在 Java 源代码中引用一个类或接口时，使用的是类似“com.java7book.chapter8.Sample”这种形式的包含名称空间的全名。通过使用 import 语句，可以省略类或接口所在的包的名称。而在字节代码中，始终使用全名，并且把全名中的“.”替换成“/”，即类似“com/java7book/chapter8/Sample”这样的形式。对于域和方法来说，在字节代码中使用描述符来说明其类型。对于域来说，其类型可能是 Java 的基本类型、对象类型或数组类型。基本类型在字节代码中用一个字符来表示：byte、char、double、float、int、long、short 和 boolean 类型对应的字符分别是 B、C、D、F、I、J、S 和 Z。对象类型的表示方式是在全名上加“L”前缀和“;”后缀。例如，一个 String 类型的域的描述符是“Ljava/lang/String;”。数组类型的表示形式是在其元素类型之前加上“[”作为前缀。“[”的个数表示数组的维度。例如，一个 double 类型的二维数组的描述符是“[[D”。“[[”的个数表示数组的维度。对于一个方法来说，它的描述符取决于参数和返回值的类型，基本形式是“(参数类型)返回值类型”，参数和返回值类型的表示方式与域相同。如果返回值是 void，则用“V”表示。例如，方法声明“int calculate(String str)”的类型描述符是“(Ljava/lang/String;)I”。除了类型描述符之外，类、域和方法还可能包含类型签名信息。类型签名是在 Java SE 5.0 中随着泛型的加入而被添加到字节代码中的，其目的是在运行时也能通过反射 API 获取泛型相关的信息。第 12 章会对泛型进行详细介绍。从前面的介绍中可以看出，字节代码中对各种名称的表示方式有些奇怪，这主要是历史原因造成的。

8.1.1 基本格式

字节代码是一个连续的字节流，其中每个部分所表示的含义是不同的。在进行解析时，需要识别出表示不同内容的字节数据之间的边界。这些数据内容可以分成定长和不定长两类。对于定长的内容来说，只需要根据长度依次读取即可；对于不定长的内容来说，会在数据的最前面给出其长度，以进行读取。为了方便介绍，定长数据的类型用 u1、u2 和 u4 等来表示，分别表示 1 字节、2 字节和 4 字节。多字节的字节顺序采用大端表示。字节代码中数据的整体分布如代码清单 8-1 所示，其中 cp_info、field_info、method_info 和 attribute_info 是表示常量池中常量、域、方法和属性的子结构，有自己内部的格式。

代码清单 8-1 字节代码的基本格式

u4 魔法数
u2 小版本号
u2 大版本号
u2 常量池中常量的个数再加 1
cp_info 常量池内容的数组
u2 访问控制标记和属性修饰符
u2 当前类或接口信息的常量池序号

u2 父类或父接口信息的常量池序号
u2 实现接口的个数
u2 实现接口名称的常量池序号
u2 域的个数
field_info 包含域信息的数组
u2 方法的个数
method_info 包含方法信息的数组
u2 属性的个数
attribute_info 包含属性信息的数组

下面对代码清单 8-1 中的内容进行详细介绍。字节代码的前 4 字节是魔法数 (magic number)，用作字节代码格式的标识符。魔法数的值固定为 0xCAFEBAE，英文含义为“咖啡宝贝”，正好与 Java 名称的来源相对应。不少二进制格式在起始位置都使用类似这样的标识符。[⊖]魔法数的作用是可以快速判断一个字节序列是否为合法的字节代码。

紧接着的 4 字节表示的是字节代码的版本号。前 2 字节表示小版本号，后 2 字节表示大版本号。由 JDK 7 编译器生成的字节代码的版本号是 51.0，对应的 4 字节的值是 0x00000033。Java 6 和 Java SE 5.0 对应的字节代码的版本号分别是 50.0 和 49.0。每个虚拟机有固定的所支持的字节代码的版本范围。当虚拟机运行它所不支持的版本的字节代码时，会抛出 `java.lang.UnsupportedClassVersionError` 错误。如果不能从其他途径得到字节代码的版本，可以通过查看字节代码的第 5 到第 8 字节的值来确定。

接下来的 2 字节表示常量池 (constant pool) 中常量的个数再加 1。常量池中所包含的是 Java 中基本类型和字符串常量值、类和接口的名称及域的名称等。每个常量的类型和所占用的字节数是不同的。这些常量被字节代码中的其他部分所引用。相同的常量在常量池中只会出现一次。可以将常量池看成是常量的一个查找表。在引用的时候，只需要指定常量在常量池中的序号即可。在常量池的个数之后，紧接着是由 `cp_info` 结构表示的每个常量的具体定义。

接下来的 2 字节表示的是类或接口的访问控制标记和属性修饰符。每个标记或修饰符对应一个比特位。只需要检查这 2 字节中特定的比特位是否为 1，就可以判断标记和修饰符是否生效。对这两个字节使用比特位与操作可以进行快速判断。其中常见的标记与修饰符包括：ACC_PUBLIC 对应于 `public` 声明，值为 0x0001；ACC_FINAL 对应于 `final` 声明，值为 0x0010；ACC_INTERFACE 表明是接口而不是一般类，值为 0x0200；ACC_ABSTRACT 对应于 `abstract` 声明，值为 0x4000；ACC_SYNTHETIC 声明由编译器生成而不在源代码中出现，值为 0x1000；ACC_ANNOTATION 声明是一个注解类型，值为 0x2000；ACC_ENUM 声明是一个枚举类型，值为 0x4000。这些标记和修饰符的设置需要遵循 Java 语言的规范。比如，在 Java 中，不能将一个接口声明为 `final`。因此在字节代码中，ACC_INTERFACE 和 ACC_FINAL 这两个标记或修饰符不能同时被设置。

[⊖] ZIP 格式文件的起始两个字节的值为“PK”，是该格式的发明者 Phil Katz 姓名的首字母缩写。

接下来的 2 字节表示的是当前 Java 接口或类的信息在常量池中的序号。同样，接下来的 2 字节表示的是当前 Java 接口或类的父类或父接口信息在常量池中的序号。如果当前类是 `java.lang.Object`，则这两个字节的值为 0，因为 `Object` 类是唯一没有父类的 Java 类。如果字节代码表示的是接口，那么对应的父类只能是 `Object` 类。

接下来的字节代码的格式就比较简单了，依次表示的是当前接口或类所继承或实现的接口的信息，以及包含的域、方法和属性的信息。由于实现的接口、域、方法和属性都可能存在多个，因此，在字节代码中表示时，先用 2 字节表示元素个数，紧接着是包含所有元素信息的数组。对于实现的接口的数组，其中的每个元素都是常量池中的序号；而对于域、方法和属性来说，每个元素都有自己独特的结构，即代码清单 8-1 中的 `field_info`、`method_info` 和 `attribute_info` 结构。

8.1.2 常量池的结构

下面介绍定义常量池中的常量的 `cp_info` 结构。每个常量定义的起始字节的值标明了常量的类型。我们将该字节称为标签。在这个字节之后是包含常量内容的若干个字节。先介绍 Java 基本类型常量的定义方式。字节代码中只包含基本类型 `int`、`long`、`float` 和 `double` 的对应表示，其他基本类型都可以用 `int` 来表示。在这 4 种类型中，`int` 和 `float` 类型的常量的内容是在标签后紧跟包含数据的 4 个字节；`long` 和 `double` 类型的常量则在标签后紧跟 8 个字节。值得注意的是，`long` 和 `double` 类型的常量会占用常量池中的两个位置。在进行计算时需要考虑这一点。

常量 `CONSTANT_Utf8_info` 表示的是一个使用修改后的 UTF-8 格式表示的字符串序列。在标签之后的两个字节表示序列的长度，紧接着是序列的内容。这种类型的常量在字节代码中的出现次数比较多。表示 `String` 类型的常量 `CONSTANT_String_info` 直接引用 `CONSTANT_Utf8_info` 常量，只包含一个对应的常量池中的序号。常量 `CONSTANT_Class_info` 表示类或接口，在标签后接着的是类或接口的全名对应的 `CONSTANT_Utf8_info` 常量的序号。对于类或接口中包含的域和方法，由两类常量来共同表示：第一类常量 `CONSTANT_NameAndType_info` 表示域和方法的名称和类型，分别由两个 `CONSTANT_Utf8_info` 常量来表示；第二类常量表示域和方法与类或接口的对应关系。常量 `CONSTANT_Fieldref_info`、`CONSTANT_Methodref_info` 和 `CONSTANT_InterfaceMethodref_info` 分别表示域、类中的方法和接口中的方法。这三种常量的结构是相似的，在标签之后分别是表示所在类或接口的 `CONSTANT_Class_info` 常量和表示名称与类型的 `CONSTANT_NameAndType_info` 常量的序号。从上面对常量池中常量的说明可以看出，常量之间存在复杂的引用关系。通过这种引用关系，可以最大限度地复用已有的内容，避免重复。

表示域的 `field_info` 结构的格式如代码清单 8-2 所示。

代码清单 8-2 表示域的 field_info 结构的格式

u2 访问控制标记和属性修饰符
u2 名称的常量的序号
u2 类型描述符的常量的序号
u2 属性的个数
attribute_info 包含属性信息的数组

域的访问控制标记和属性修饰符的表示方式与类或接口中的表示方式很相似，只是所能使用的访问控制标记和修饰符的种类不同。域的名称和类型描述符的表示方式与常量 CONSTANT_NameAndType_info 是相同的。域的属性是在个数后接着一个包含 attribute_info 结构的数组。表示方法的 method_info 结构的格式与代码清单 8-2 中的 field_info 的格式是相同的。

8.1.3 属性

前面介绍的结构只能表示类、域和方法的最基本信息。Java 语法结构中的其他信息都由属性来表示。属性从本质上来说只是一个简单的名值对。这一点从 attribute_info 结构的格式可以看出来。在 attribute_info 结构中，开始的两个字节表示属性名称对应的 CONSTANT_Utf8_info 常量的序号，接下来的 4 字节表示属性值的字节数组的长度，随后是属性值的字节数组。这种简单的结构使得 attribute_info 可以表示任何类型的属性。Java 字节代码规范预先定义了一些属性的名称及其内部结构。随着 Java 语言的发展，不断有新的预定义属性被加入进来。例如，J2SE 5.0 中引入了表示泛型信息的 Signature 属性，Java 7 中引入了表示 invokedynamic 指令对应的启动方法的 BootstrapMethods 属性。除了这些预定义的标准属性之外，不同的虚拟机实现可以提供自己独有的属性。虚拟机会自动忽略所有无法识别的属性。

8.2 动态编译 Java 源代码

把 Java 源代码编译成字节代码的过程通常是由 Java 编译器来完成的。编译和运行是两个独立的过程。编译所得到的字节代码将作为程序的分发方式。字节代码的运行则可能在不同平台上的虚拟机上进行。通过动态编译 Java 源代码，可以把编译和运行两个过程统一起来，都在运行时完成。这为程序增加了在运行时动态修改其行为的能力。第 9 章介绍的类加载器也可以实现类似的功能，但类加载器所操纵的对象是字节代码，而动态编译所操纵的对象是 Java 源代码。对于开发人员来说，使用源代码比字节代码要简单很多。

动态编译 Java 源代码的使用场景并不复杂。对于一个 Java 源文件，在运行时使用 API 编译出字节代码，再使用类加载器加载到虚拟机中运行。运行时一般使用 Java 反射 API。Java 源文件的内容可以来自磁盘文件，也可以来自远程服务器。任何满足 Java 语法要求的字符流都可以作为动态编译的输入。编译之后的字节代码可以保存在磁盘上，也可以保存在内存中。Java 平台提供了多种方式来动态编译源代码。这些方式的优缺点

各不相同，可以在不同的场景中使用。下面对这些方式进行具体的介绍。

8.2.1 使用 javac 工具

使用 javac 工具是最直接、最简单的动态编译 Java 源代码的方式。虽然 javac 大多数时候是作为命令行工具来使用的，但是通过 Java 标准库提供的创建底层操作系统上进程的能力，可以在运行时动态调用 javac 工具来编译源代码。第 6 章对创建并启动进程做了详细的说明。代码清单 8-3 给出了相关实现的示例。类 JavacCompiler 中 compile 方法的参数 src 和 output 分别表示作为输入的 Java 源代码的路径和编译之后产生的 class 文件的输出路径。

代码清单 8-3 在程序中使用 javac 工具编译 Java 源代码

```
public class JavacCompiler {
    public void compile(Path src, Path output) throws CompileException {
        ProcessBuilder pb =
            new ProcessBuilder("javac.exe", src.toString(), "-d", output.
                toString());
        try {
            pb.start();
        } catch (IOException e) {
            throw new CompileException(e);
        }
    }
}
```

使用 javac 工具的不足之处在于输入和输出都只能以文件形式存在。这也是 javac 工具本身的使用方式。对于字节流形式的输入内容，需要保存在磁盘上再进行编译。对于输出的字节代码，类加载器也需要支持从磁盘文件加载的方式。除此之外，使用外部进程方式的性能也比较低。

除了以外部进程方式来调用之外，javac 工具也提供了编程接口供程序直接调用。这种方式相对于外部进程方式，可移植性更好、性能更优。对源代码进行编译操作由类 com.sun.tools.javac.Main 中的静态方法 compile 完成。调用 javac 工具时的参数以 String 数组的形式传递给 compile 方法。除此之外，还可以使用一个 java.io.PrintWriter 类型的参数来指定编译器的输出位置。代码清单 8-4 给出了使用 javac 工具 API 的示例。编译器的输出信息保存在一个文件中，方便查看。

代码清单 8-4 使用 javac 工具 API 编译 Java 源代码

```
public class JavacAPICompiler {
    public void compile(Path src, Path output) throws CompileException {
        String[] args = new String[] {src.toString(), "-d", output.toString()};
        try {
            PrintWriter out = new PrintWriter(Paths.get("output.txt").toFile());
            com.sun.tools.javac.Main.compile(args, out);
        }
```

```
        } catch (FileNotFoundException e) {
            throw new CompileException(e);
        }
    }
}
```

8.2.2 Java 编译器 API

使用 javac 工具的一个问题是其 API 是 Oracle 的私有实现，这点从 API 的包名 com.sun.* 可以看出来。私有 API 的接口和实现可能发生变化，从而造成后期维护上的问题。从 Java SE 6 开始，Java 编译器相关的 API 以 JSR 199 (Java™ Compiler API) 的形式规范下来，Java 编译器 API 采用了 Java 平台标准的服务提供者接口的定义方式。编译器 API 中仅包含接口声明，对应的实现由平台实现者提供。使用者通过工厂方法或服务加载器来查找具体的实现。相关的方法调用都通过接口来完成。Java 平台当前的编译器提供了编译器 API 的默认实现，可以直接使用。通过编译器 API 可以对编译过程进行更加精细的控制。

先通过与代码清单 8-3 和代码清单 8-4 两个示例相同的编译源文件的方式来说明编译器 API 的基本用法，如代码清单 8-5 所示。编译器 API 中的编译器由 javax.tools.JavaCompiler 接口表示。首先要得到 JavaCompiler 接口的实现。通过 javax.tools.ToolProvider 类的 getSystemJavaCompiler 方法可以得到当前 Java 平台上默认的编译器实现。一般来说，使用这个默认实现就足够了。在编译器 API 中对所操作的对象来源进行了抽象，用 javax.tools.FileObject 接口表示。虽然从名称上看，FileObject 接口表示的是文件对象，但是实际上它是一个数据来源的抽象表示，不仅包括磁盘文件，也包括内存中的对象和数据库中的数据等。FileObject 接口中的方法主要用来获取数据来源的信息和进行读写操作等。接口 javax.tools.JavaFileManager 继承自 FileObject 接口，用来表示 Java 的源代码和字节代码文件。在编译过程中对 Java 源代码和字节代码文件的管理是由 javax.tools.JavaFileManager 接口的实现对象来完成的。JavaFileManager 接口提供的方法所操作的目标是抽象的路径。通过 JavaFileManager 接口不仅可以得到某个路径对应的 FileObject 接口和 JavaFileManager 接口的实现对象，还可以根据条件列出某个路径下包含的文件对应的 JavaFileManager 接口的实现对象。如果 Java 源代码保存在磁盘上，那么可以使用基于 java.io.File 接口实现的 javax.tools.StandardJavaFileManager 接口。通过 StandardJavaFileManager 接口可以从文件名或 File 类的对象中创建 JavaFileManager 接口的实现对象。在一般情况下，使用 StandardJavaFileManager 接口进行文件管理就足够了。通过 JavaCompiler 类的对象的 getStandardFileManager 方法可以得到一个 StandardJavaFileManager 接口的实现对象。

具体的编译过程是首先调用 JavaCompiler 类的对象的 getTask 方法得到一个表示编译任务的 JavaCompiler.CompilationTask 类的对象，再调用此对象的 call 方法来执行此

任务。创建编译任务的 getTask 方法的参数比较多，首先是用来输出编译器信息的 java.io.Writer 类的对象；其次是管理源代码和字节代码文件的 JavaFileManager 接口的实现对象；接着是处理编译过程中诊断信息的 javax.tools.DiagnosticListener 接口的实现对象；最后 3 个参数都是 java.lang.Iterable 类型的对象，分别表示用来遍历编译时的选项、字节代码文件路径和源文件路径的迭代器。

代码清单 8-5 Java 编译器 API 的使用示例

```
public class JavaCompilerAPICompiler {
    public void compile(Path src, Path output) throws IOException {
        JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
        try (StandardJavaFileManager fileManager = compiler.
            getStandardFileManager(null, null, null)) {
            Iterable<? extends JavaFileObject> compilationUnits = fileManager.
                getJavaFileObjects(src.toFile());
            Iterable<String> options = Arrays.asList("-d", output.toString());
            CompilationTask task = compiler.getTask(null, fileManager, null,
                options, null, compilationUnits);
            boolean result = task.call();
        }
    }
}
```

编译器 API 相对于 javac 工具的一个重要优势在于 Java 源代码的存在不限于文件形式。JavaFileObject 接口的实现对象可以作为源代码的来源。在实现 JavaFileObject 接口时，比较好的做法是继承已有的 javax.tools.SimpleJavaFileObject 类，从而减少实现的代价。最常见的需求是允许编译器使用字符串作为源代码的表现形式，免去了使用文件作为中间形式的麻烦。代码清单 8-6 给出的 StringSourceJavaFileObject 类表示从字符串创建出的 JavaFileObject 接口的实现对象，在创建时传入类名和内容即可。

代码清单 8-6 字符串形式的 Java 源代码表示方式

```
public class StringSourceJavaFileObject extends SimpleJavaFileObject {
    private String content;

    public StringSourceJavaFileObject(String name, String content) {
        super(URI.create("string:/// " + name.replace('..', '/') + Kind.SOURCE.
            extension), Kind.SOURCE);
        this.content = content;
    }

    public CharSequence getCharContent(boolean ignoreEncodingErrors)
        throws IOException {
        return content;
    }
}
```

使用 StringSourceJavaFileObject 类之后，可以对任意包含 Java 源代码的字符串进行编译。这为编译带来了很多灵活性。举例来说，编写一个 Java 程序来计算带括号的四则运算表达式的值，实现的方式可以有很多。比较传统的做法是对表达式进行语法解析，模拟计算过程来得到结果。如果考虑到括号的存在和操作符之间的优先级顺序等问题，那么采用这种做法的实现并不简单，而且容易出错。另外一种做法是使用第 2 章介绍的脚本语言支持 API，把表达式当成一个 JavaScript 语句，通过 eval 方法得到结果。还有一种做法是使用 Java 编译器 API，实现起来也会比较简单。基本的思路是把表达式作为一个 Java 方法的内容，得到一个 Java 源文件，编译此源文件之后得到字节代码，再使用类加载器加载字节代码到虚拟机中，最后通过反射 API 调用其中的方法，得到计算结果。代码清单 8-7 给出了这种计算方式的完整实现。

代码清单 8-7 使用 Java 编译器 API 的表达式求值方式

```

public class Calculator extends ClassLoader {
    public double calculate(String expr) throws Exception {
        String className = "CalculatorMain";
        String methodName = "calculate";
        String source = "public class " + className + " { public static double "
            + methodName + "() { return " + expr + "; } }";
        JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
        StandardJavaFileManager fileManager = compiler.getStandardFileManager(null,
            null, null);
        JavaFileObject sourceObject = new StringSourceJavaFileObject(className,
            source);
        Iterable<? extends JavaFileObject> fileObjects = Arrays.
            asList(sourceObject);
        Path output = Files.createTempDirectory("calculator");
        Iterable<String> options = Arrays.asList("-d", output.toString());
        CompilationTask task = compiler.getTask(null, fileManager, null, options,
            null, fileObjects);
        boolean result = task.call();
        if (result) {
            byte[] classData = Files.readAllBytes(Paths.get(output.toString(),
                className + ".class"));
            Class<?> clazz = defineClass(className, classData, 0, classData.
                length);
            Method method = clazz.getMethod(methodName);
            Object value = method.invoke(null);
            return (Double) value;
        }
        else {
            throw new Exception("无法识别的表达式。");
        }
    }
}

```

在代码清单 8-7 中，先根据传入的表达式的内容创建 Java 源代码。比如，表达式的内容是“(3+2)*5”，所得到的 Java 源代码的内容如代码清单 8-8 所示。只需要编译其中的 Java 类，并通过反射 API 调用其 calculate 方法，就可以得到表达式的计算结果。

代码清单 8-8 动态生成的 Java 源代码的内容示例

```
public class CalculatorMain {  
    public static double calculate() {  
        return (3+2)*5;  
    }  
}
```

调用编译器 API 的方式与代码清单 8-5 很相似，区别在于不是通过 StandardJavaFileManager 接口的实现对象从文件路径中得到 JavaFileObject 接口的实现对象，而是使用 StringSourceJavaFileObject 类的对象作为编译器的源文件来源。编译之后的字节代码保存在磁盘上的临时目录中。编译成功之后，从 class 文件中得到字节代码的内容，再使用类加载器从字节代码中定义出 Java 类。得到 Java 类之后，通过反射 API 来调用类中包含的 calculate 方法，得到表达式的计算结果。

8.2.3 使用 Eclipse JDT 编译器

Java 编译器 API 虽然为编译 Java 源代码提供了规范的 API，但是它的能力很有限。使用编译器 API 除了在源代码方面可以使用不同的来源之外，与编译相关的其他设置仍需要由各种选项来表示。编译器 API 所提供的对开发人员的接口也并不友好。比如，当编译过程出现错误的时候，只能依靠 DiagnosticListener 接口获得一些简单的错误信息。

Eclipse IDE 中使用的是 Eclipse 自己开发的 Java 编译器。该 Java 编译器是 Eclipse Java 开发工具 (Java development tools, JDT) 的一部分。JDT 的 Java 编译器相对于 Java 编译器 API 来说，所提供的编程接口更加丰富，使用起来也相对复杂。通过 JDT 编译器完成一个基本的编译过程，需要提供多个接口的实现。当要求对编译过程进行更加复杂的定制时，JDT 编译器是个不错的选择。在使用 JDT 编译器之前，需要下载 JDT 的二进制分发包。只需要使用分发包中的 org.eclipse.jdt.core 库。JDT 编译器相关的 Java 类都在 org.eclipse.jdt.internal.compiler 包中。包名中的“internal”说明此包是 Eclipse 内部使用的，一般不推荐直接使用。这里使用内部包的原因是 JDT 编译器并没有直接的公开 API，而是与 Eclipse 平台的项目构建机制结合在一起，由构建机制自动触发。为了可以在程序中直接使用，必须利用内部 API。下面从基本编译过程所需的接口开始介绍。

要介绍的第一个接口是 org.eclipse.jdt.internal.compiler.env.ICompilationUnit，表示编译器所操作的编译单元，指的是 Java 源代码。ICompilationUnit 接口的作用类似于之前介绍的 JavaFileObject 接口，其实现类需要提供与源代码相关的信息，包括源代码对应的 Java 文件的名称、类型的名称、所在的 Java 包的名称以及源文件的内容。代码清

单 8-9 中给出了与代码清单 8-6 中 StringSourceJavaFileObject 类相似的基于字符串的编译单元的实现。创建 StringCompilationUnit 类的对象时需要提供 Java 类的全名和源代码的内容。

代码清单 8-9 基于字符串的编译单元的实现类

```
public class StringCompilationUnit implements ICompilationUnit {

    private String fileName;
    private char[] typeName;
    private char[][] packageName;
    private String content;

    public StringCompilationUnit(String className, String content) {
        this.content = content;
        if (className.contains("$")) {
            className = className.substring(0, className.indexOf("$"));
        }
        fileName = className.replace('.', '/') + ".java";
        int pos = className.lastIndexOf('.');
        if (pos > 0) {
            typeName = className.substring(pos + 1).toCharArray();
        } else {
            typeName = className.toCharArray();
        }
        String[] names = className.split("\\.");
        packageName = new char[names.length - 1][];
        for (int i = 0; i < packageName.length; i++) {
            packageName[i] = names[i].toCharArray();
        }
    }

    public char[] getFileName() {
        return fileName.toCharArray();
    }

    public char[] getContents() {
        return content.toCharArray();
    }

    public char[] getMainTypeName() {
        return typeName;
    }

    public char[][] getPackageName() {
        return packageName;
    }
}
```

与 JDT 编译器相关的第二个接口是 org.eclipse.jdt.internal.compiler.env.INameEnvironment，表示编译过程中使用的名称环境。编译器在编译过程中使用该接口的方法来查找所遇到的类型、编译单元和包的定义。比如，编译器在遇到 java.lang.Object 类型的时候，会调用 INameEnvironment 接口中的 findType 方法来查找该类型的定义。如果找不到相关的定义，编译过程会出现错误。INameEnvironment 接口的 findType 方法接受类型的全名作为参数，返回表示查找结果的 org.eclipse.jdt.internal.compiler.env.NameEnvironmentAnswer 类的对象。NameEnvironmentAnswer 类的对象表示的结果可以是类型的字节代码、对应的编译单元或未解析的源代码。这三种不同形式的结果为编译时查找类型的定义提供了足够的灵活性。比如类 A 在编译时依赖于类 B，而类 B 的源代码可能是运行时动态生成的字符串。对于这种情况，在 findType 方法的实现中，当需要查找的类名为 B 时返回一个表示代码清单 8-9 中编译单元的 NameEnvironmentAnswer 类的对象即可。代码清单 8-10 给出了 INameEnvironment 接口实现类中的一个 findType 方法。如果可以获取 Java 类型对应的字节代码，可以从字节代码数据中创建出 org.eclipse.jdt.internal.compiler.classfmt.ClassFileReader 类的对象，并作为结果返回。

代码清单 8-10 INameEnvironment 接口实现类中的 findType 方法

```

private NameEnvironmentAnswer findType(String name) {
    byte[] bytes = getClassDefinition(name);
    if (bytes == null) {
        return null;
    }
    try {
        ClassFileReader classFileReader = new ClassFileReader(bytes, name.
            toCharArray(), true);
        return new NameEnvironmentAnswer(classFileReader, null);
    } catch (ClassFormatException e) {
        e.printStackTrace();
    }
    return null;
}

```

最后一个重要的接口是 org.eclipse.jdt.internal.compiler.ICompilerRequestor，用来对编译的结果进行处理。该接口只有一个方法 acceptResult，参数是一个 org.eclipse.jdt.internal.compiler.CompilationResult 类的对象。CompilationResult 类中包含了与编译结果相关的各种具体信息，包括编译之后的字节代码和出现的错误等。代码清单 8-11 中给出了完整的编译过程的示例。首先创建一个 ICompilerRequestor 接口的实现对象来处理编译的结果。出于演示的目的，代码清单 8-11 中并没有对编译生成的字节代码进行额外的操作，对错误的处理也比较简单。类 org.eclipse.jdt.internal.compiler.ClassFile 表示包含字节代码的 class 文件。JDT 中的编译器由类 org.eclipse.jdt.internal.compiler.Compiler 表示。为了创建出 Compiler 类的对象，除了 INameEnvironment 接口和 ICompilerRequestor 接口的实现类外，还需要另外几个接口的实现类。不过这

几个接口可以使用 JDT 提供的默认实现类。类 org.eclipse.jdt.internal.compiler.impl.CompilerOptions 表示编译时的选项，其中包含一系列可供配置的公开域。接口 org.eclipse.jdt.internal.compiler.IErrorHandlingPolicy 表示的是在编译中遇到错误时的处理策略。处理策略包括两个选择：一个选择为是否在遇到第一个错误时就终止整个编译过程，另外一个选择为是否带着发现的错误继续进行编译。接口 org.eclipse.jdt.internal.compiler.IProblemFactory 用来在编译出现错误时生成具体的错误信息。由 IProblemFactory 接口的实现对象创建的错误可以从 CompilationResult 类的对象中得到。创建出 Compiler 类的对象之后，可以调用其 compile 方法对一组 ICompilationUnit 接口的实现对象进行处理。代码清单 8-11 中编译的是以字符串形式出现的源代码。

代码清单 8-11 使用 JDT 编译器进行编译的示例

```

public void compile(String code) {
    ICompilerRequestor compilerRequestor = new ICompilerRequestor() {
        public void acceptResult(CompilationResult result) {
            if (result.hasErrors()) {
                System.out.println("出现编译错误。");
            }
            ClassFile[] clazzFiles = result.getClassFiles();
            ClassFile clazzFile = clazzFiles[0];
            // 使用编译之后的字节代码
        }
    };

    IErrorHandlerPolicy policy = DefaultErrorHandlerPolicies.exitOnFirstError();
    IProblemFactory problemFactory = new DefaultProblemFactory();
    INameEnvironment nameEnvironment = new BasicNameEnvironment();
    CompilerOptions options = new CompilerOptions();
    Compiler jdtCompiler = new Compiler(nameEnvironment, policy, options,
        compilerRequestor, problemFactory);

    ICompilationUnit[] compilationUnits = new StringCompilationUnit[] {new
        StringCompilationUnit("Main", code)};
    jdtCompiler.compile(compilationUnits);
}

```

JDT 编译器由于其功能强大、使用复杂，一般只用在开发工具或框架中。Java Web 开发框架 Play[⊖]在内部使用 JDT 编译器来对 Web 应用的 Java 源代码进行编译，同时结合类加载器来实现对 Java 源代码的修改即时生效，不需要重启服务器。

8.3 字节代码增强

上一节介绍的动态编译 Java 源代码方式的目的在于产生新的字节代码，其输入是

[⊖] Play 框架的网址是 <http://www.playframework.org/>。

Java 源代码。在某些情况下，可能无法获得相关的源代码，而只得到二进制的字节代码。如果希望对字节代码进行处理，需要用到字节代码增强技术。字节代码增强的含义是对已有的 Java 字节代码进行修改，从而改变其运行时的行为。Java 字节代码格式的开放性使这种增强方式成为可能。增强的工作可以在程序运行之前或运行时完成。在程序运行之前的做法是先对程序的字节代码进行处理，得到修改后的字节代码，再由虚拟机来运行。通常是在基本的编译过程完成之后，再使用工具对编译之后的字节代码进行处理。在程序运行时的做法是在字节代码被虚拟机加载之前对字节代码进行修改。通常由增强代理或类加载器来完成修改工作。一般来说，自己开发的程序适合使用第一种做法，原因是可以避免运行时修改所带来的开销。而框架一般使用第二种做法，原因是框架本身无法在运行之前进行任何处理，只能在运行时进行处理。有些框架则两种做法都支持，如 Apache OpenJPA 既提供了 Ant 任务对使用 OpenJPA 的程序在运行之前进行处理，又可以通过增强代理在运行时进行。

字节代码增强技术在很多场景中都有应用。使用它可以灵活高效地解决某些问题，比如典型的 AOP 编程中的方法拦截功能。还可以利用字节代码增强技术进行代码生成。由于 Java 字节代码的格式是公开和规范的，对字节代码进行操作并不是一件复杂的事。不过字节代码格式本身比较复杂，最好借助工具的支持来进行修改。有不少工具可以用于操作字节代码，如 OpenJPA 使用的 serp[⊖] 和 Play 框架使用的 Javassist[⊖]。下面要介绍的操作字节代码的工具是 ASM，它在 AspectJ、JRuby 和 Jython 等框架中都有使用。

8.3.1 使用 ASM

ASM 是一个轻量级的 Java 字节代码操作工具。ASM 为字节代码中的各种结构和数据提供了一种面向对象的表示方式，使开发人员不需要了解常量池等具体细节，就可以很方便地创建新的字节代码或对已有的字节代码进行修改。ASM 所提供的 API 在设计上比较贴近于字节代码的原始格式，这使得 ASM 操作字节代码的性能比较高，同时也要求使用者对字节代码格式有比较深入的了解。尤其在用 ASM 生成方法体的字节代码时，需要使用者对 Java 虚拟机的指令有比较详细的了解。不过，ASM 也提供了一些工具帮助开发人员查看字节代码的内容，以及生成相关的使用 ASM 的代码。

从之前对字节代码格式的介绍中可以看出，字节代码实际上采用了一种松散的树形组织结构。类中包含域、方法和属性，而域和方法又有各自具体的结构。对字节代码的处理，可以与同样是树形结构的 XML 文档的处理方式进行类比。XML 文档通常有 SAX 和 DOM 两种处理方式。SAX 是基于事件的，而 DOM 是基于文档的树形结构的。ASM 的 API 可以与 XML 文档的两种处理 API 相对应。ASM 中基本的字节代码处理 API 是基于事件的，类似于 SAX。当在处理过程中遇到特定的结构时，会产生对应的事件。通

[⊖] serp 工具的网址是 <http://serp.sourceforge.net/>。

[⊖] Javassist 工具的网址是 <http://www.javassist.org/>。

通过对事件的处理来操作字节代码。在具体的实现上，采用了访问者模式对树形结构进行遍历。在事件 API 的基础上，ASM 也提供了类似于 DOM 的树形 API。这两种 API 的优缺点可以同样类比 SAX 和 DOM。

ASM 对于字节代码的操作有 3 个典型的场景，分别是字节代码的读取、生成和修改。读取的含义是指查看一个已有的字节代码中的内容，适合进行相关的代码分析；生成的含义是指从零开始生成一个 Java 类或接口的字节代码，由虚拟机来运行，适合代码生成；修改的含义则是指对已有的字节代码进行修改，适合功能增强。

1. 读取字节代码

在介绍对字节代码的读取之前，先介绍 ASM 中的核心接口 `org.objectweb.asm.ClassVisitor`。正如 `ClassVisitor` 接口的名称所表示的含义一样，这个接口是对字节代码中的类或接口信息进行访问的访问者。该接口中所包含的方法用于对类中包含的不同部分进行访问，比如 `visitField` 方法可以访问一个域的信息，`visitMethod` 方法访问一个方法的信息。读取字节代码的操作由 `ClassVisitor` 接口的实现对象和 `org.objectweb.asm.ClassReader` 类的对象结合起来完成。`ClassReader` 类可以读取包含字节代码的字节流，也可以根据类名来查找并读取。`ClassReader` 类的 `accept` 方法可以接受一个 `ClassVisitor` 接口的实现对象作为参数。当 `ClassReader` 类的对象在读取过程中遇到类中的不同结构时，会调用 `ClassVisitor` 接口的实现对象中的相关方法。比如读取一个方法的时候，`ClassVisitor` 接口的实现对象中的 `visitMethod` 方法会被调用，调用时的实际参数中包含了当前方法的相关信息。如果从事件的角度来看，`ClassReader` 类的对象是事件的生产者，`ClassVisitor` 接口的实现对象是事件的消费者。事件的具体来源是已有的字节代码，对事件的处理逻辑则由开发人员提供。

下面通过一个具体的示例来介绍 `ClassVisitor` 接口和 `ClassReader` 类的使用方式。这个示例要实现的功能很简单，统计一个类中包含的方法的个数。完整的实现如代码清单 8-12 所示。先介绍一下 `ClassVisitor` 接口中的方法。`visit` 方法表示的是访问类的基本信息，包括版本号、访问控制标记和修饰符、名称、类型签名、父类名称和实现的接口名称等。这些信息是与字节代码格式中的内容直接对应的。`visitInnerClass` 方法表示的是访问类中包含的内部类的信息。`visitOuterClass` 方法表示的是访问当前类的外部类的信息。`visitSource` 方法表示的是访问类对应的源文件的信息。`visitAttribute` 方法表示的是访问属性的信息。`visitField` 方法表示的是访问类中的一个域，参数中包含了域的基本信息。如果需要继续访问域中包含的详细信息，那么需要返回一个 `org.objectweb.asm.FieldVisitor` 接口的实现对象。`visitAnnotation` 方法表示的是访问类中的注解的信息。如果需要继续访问注解的详细信息，那么需要返回一个 `org.objectweb.asm.AnnotationVisitor` 接口的实现对象。`visitMethod` 方法表示的是访问类中包含的方法的信息。同样，可以返回一个 `org.objectweb.asm.MethodVisitor` 接口的实现对象来访问方法的具体信息。从这些方法的名称和参数可以看出，ASM 的 `ClassVisitor` 接口采用了与字

节代码进行直接对应的方式。熟悉字节代码格式的开发人员很容易进行对应。使用访问者模式的实现也很清晰。对域、注解和方法的处理比较特殊，这是因为它们包含的信息比较复杂，需要另外的访问者接口来表示。

代码清单 8-12 统计方法个数的 ClassVisitor 接口的实现

```
public class MethodCounter implements ClassVisitor {
    private int count = 0;
    public void visit(int version, int access, String name, String signature,
                      String superName, String[] interfaces) {
    }

    public AnnotationVisitor visitAnnotation(String desc, boolean visible) {
        return null;
    }

    public void visitAttribute(Attribute attr) {
    }

    public void visitEnd() {
    }

    public FieldVisitor visitField(int access, String name, String desc, String
                                  signature, Object value) {
        return null;
    }

    public void visitInnerClass(String name, String outerName, String innerName,
                               int access) {
    }

    public MethodVisitor visitMethod(int access, String name, String desc, String
                                    signature, String[] exceptions) {
        count++;
        return null;
    }

    public void visitOuterClass(String owner, String name, String desc) {
    }

    public void visitSource(String source, String debug) {
    }

    public int getCount() {
        return count;
    }

    public static void main(String[] args) throws IOException {
        ClassReader reader = new ClassReader("java.lang.String");
        MethodCounter counter = new MethodCounter();
    }
}
```

```

        reader.accept(counter, 0);
        System.out.println(counter.getCount());
    }
}

```

代码清单 8-12 中的示例只对类中包含的方法感兴趣，因此只在 visitMethod 方法中添加了具体实现。在 visitMethod 方法的实现中直接修改计数器的值。由于不需要获取方法的详细信息，因此不需要使用自定义的 MethodVisitor 接口的实现对象，直接返回 null 即可。在使用时，由 ClassReader 类的对象负责读取 String 类的字节代码，通过 accept 方法交由 MethodCounter 类的对象来处理。

2. 生成字节代码

在有些情况下，可能需要从零开始生成一个类或接口对应的字节代码，比如生成程序运行所需的辅助代码。ASM 中与 ClassReader 类对应的 org.objectweb.asm.ClassWriter 类可以用来创建 Java 类或接口的字节代码。ClassWriter 类实现了 ClassVisitor 接口。这也是 ASM 设计中比较巧妙的一个地方。ClassVisitor 接口不仅可以访问字节代码中的内容，还可以用来生成相关的内容。在进行访问时，ClassVisitor 接口中的 visit、visitField 和 visitMethod 等方法是被动调用的，用来通知在字节代码中发现了对应的结构，而相关的信息作为方法调用时的实际参数来传递。在生成字节代码时，ClassVisitor 接口中的方法由使用者来调用。调用者提供的参数作为字节代码中相关结构的值。从事件的角度来看，ClassVisitor 接口不仅可以作为事件的消费者，还可以作为事件的生产者。

在第 2 章介绍 Java 语言的动态性时提到，很多动态类型语言可以生成 Java 字节代码，并且在 Java 虚拟机上执行这些字节代码。这些语言直接生成所需的字节代码，而不需要通过 Java 编译器。下面通过设计一门小的语言来说明 ASM 工具生成字节代码的用处。这门语言是 DRAW，来进行简单的图形绘制，只包含 MOVETO 和 LINETO 两条指令，分别表示移动到某个位置，以及从当前位置画线到另外一个位置。DRAW 语言可以看成是领域特定语言（Domain Specific Language，DSL）的一个简单示例。代码清单 8-13 给出了 DRAW 语言的示例代码，其功能是绘制一个三角形。

代码清单 8-13 DRAW 语言的示例代码

```

MOVETO 30 30
LINETO 30 100
LINETO 70 100
LINETO 30 30

```

使用 DRAW 语言来编写绘制图形时的指令，这些指令对于普通用户来说是容易理解的。要把 DRAW 语言的代码转换成可以在虚拟机上运行的字节代码才能让用户看到绘制的结果。利用 Java 平台中的 Java 2D API 可以进行图形绘制。与代码清单 8-13 中的 DRAW 语言代码对应的 Java 代码如代码清单 8-14 所示。

代码清单 8-14 与 DRAW 语言对应的 Java 代码

```
public class DrawingComponent extends Component {
    public void paint(Graphics g) {
        g.drawLine(30, 30, 30, 100);
        g.drawLine(30, 100, 70, 100);
        g.drawLine(70, 100, 30, 30);
    }
}
```

从实现的角度来说，一种做法是把 DRAW 语言的代码转换成 Java 源代码，再使用之前介绍的动态编译技术来进行编译。这种做法的不足之处在于不够直接，性能会受到一定程度的影响。更好的做法是直接从 DRAW 语言的代码中生成 Java 字节代码。生成一个 Java 类对应的字节代码并不是一件容易的事情。在字节代码中所要考虑的细节比源代码要多不少。一般的做法是对所需要生成的字节代码进行逆向处理。ASM 提供了两组工具类，可以对生成字节代码的操作提供辅助。第一组类用来以直观的方式输出字节代码中的内容，方便开发人员查看字节代码中各部分的细节。这组类中比较重要的是 org.objectweb.asm.util.TraceClassVisitor 类，用来输出 Java 类的信息。第二组类用来产生生成字节代码所需的使用 ASM 的源代码。这组类中比较重要的是 org.objectweb.asm.util.ASMifierClassVisitor 类。开发人员可以先编写与所要生成的字节代码相对应的 Java 代码，再编译此 Java 代码得到字节代码。对于字节代码使用 ASMifierClassVisitor 类可以得到使用 ASM 进行字节代码生成时应该编写的 Java 代码。这些 Java 代码可以作为具体实现的基础。

先使用 ASMifierClassVisitor 类对代码清单 8-14 中的 DrawingComponent 类产生的字节代码进行处理，得到基本的使用 ASM 的 Java 代码。再以此 Java 代码为基础，得到完整的字节代码生成的实现，如代码清单 8-15 所示。在 DrawingCodeGenerator 类中，创建了一个 ClassWriter 类的对象。在创建时使用了标记 ClassWriter.COMPUTE_MAXS，表明由 ClassWriter 类的对象自动计算栈大小和局部变量个数的最大值。如果不使用此标记，在调用 MethodVisitor 接口的实现对象的 visitMaxs 方法时，需要手动计算这两个值。如果计算出错，字节代码无法正确运行。字节代码生成的基本逻辑是先生成字节代码中类的基本信息，再生成 paint 方法的内容，最后根据 DRAW 语言代码的内容生成 paint 方法中调用 drawLine 方法的相关指令代码。

代码清单 8-15 生成 DRAW 语言对应的字节代码

```
public class DrawingCodeGenerator implements Opcodes {
    private ClassWriter writer = new ClassWriter(ClassWriter.COMPUTE_MAXS);
    private MethodVisitor mv = null;
    private int currentX = 0;
    private int currentY = 0;

    public byte[] generate(String sourceCode) throws IOException {
```

```
generateClassInfo();
generatePaintMethod(sourceCode);
writer.visitEnd();
return writer.toByteArray();
}

private void generateClassInfo() {
    writer.visit(V1_7, ACC_PUBLIC + ACC_SUPER, "com/java7book/chapter8/asm/
        DrawingComponent", null, "java.awt.Component", null);
    mv = writer.visitMethod(ACC_PUBLIC, "<init>", "()V", null, null);
    mv.visitCode();
    mv.visitVarInsn(ALOAD, 0);
    mv.visitMethodInsn(INVOKEPECIAL, "java.awt.Component", "<init>", "()V");
    mv.visitInsn(RETURN);
    mv.visitMaxs(1, 1);
    mv.visitEnd();
}

private void generatePaintMethod(String sourceCode) throws IOException {
    mv = writer.visitMethod(ACC_PUBLIC, "paint", "(Ljava.awt.Graphics;)V",
        null, null);
    mv.visitCode();
    BufferedReader reader = new BufferedReader(new StringReader(sourceCode));
    String line = null;
    while ((line = reader.readLine()) != null) {
        if (line.startsWith("MOVETO")) {
            handleMoveTo(line);
        }
        else if (line.startsWith("LINETO")) {
            handleLineTo(line);
        }
    }
    mv.visitInsn(RETURN);
    mv.visitMaxs(0, 0);
    mv.visitEnd();
}

private void handleMoveTo(String line) {
    String pos = line.substring("MOVETO ".length());
    String[] parts = pos.split(" ");
    currentX = Integer.parseInt(parts[0]);
    currentY = Integer.parseInt(parts[1]);
}

private void handleLineTo(String line) {
    String pos = line.substring("LINETO ".length());
    String[] parts = pos.split(" ");
    int x = Integer.parseInt(parts[0]);
    int y = Integer.parseInt(parts[1]);
    mv.visitVarInsn(ALOAD, 1);
```

```

        mv.visitIntInsn(BIPUSH, currentX);
        mv.visitIntInsn(BIPUSH, currentY);
        mv.visitIntInsn(BIPUSH, x);
        mv.visitIntInsn(BIPUSH, y);
        mv.visitMethodInsn(INVOKEVIRTUAL, "java.awt/Graphics", "drawLine", "(II)V");
        currentX = x;
        currentY = y;
    }
}

```

3. 修改字节代码

使用 ASM 的另外一个场景是对已有的字节代码进行修改。通常由 ClassReader 类、ClassWriter 类和 org.objectweb.asm.ClassAdapter 类来共同完成。ClassAdapter 类实现了 ClassVisitor 接口，可以作为 ClassReader 类和 ClassWriter 类之间的桥梁。ClassReader 类的对象在读取字节代码时，把 ClassAdapter 类的对象作为它所产生的事件的消费者。当事件产生时，ClassAdapter 类的对象中的相关方法会被调用。ClassAdapter 类的对象在创建时，需要指定另外一个 ClassVisitor 接口的实现对象作为参数。这个 ClassVisitor 接口的实现作为事件的消费者。当 ClassAdapter 类的对象中的方法被调用时，默认的行为是直接调用所封装的 ClassVisitor 接口的实现对象中的对应方法。这相当于把 ClassReader 类的对象所产生的事件原封不动地传递到作为消费者的 ClassVisitor 接口的实现对象中。如果需要进行修改操作，可以覆写 ClassAdapter 类中的对应方法来改变相关的逻辑。例如，如果希望删除原始字节代码中的名为“test”的方法，可以在 ClassAdapter 类的 visitMethod 方法的实现中检查当前方法的名称。如果名称为“test”，则直接返回 null，不把该事件传递给 ClassAdapter 类的对象封装的 ClassVisitor 接口的实现对象。在这种情况下，被封装的 ClassVisitor 接口的实现对象就看不到这个方法，相当于这个方法被删除。ClassAdapter 类中的所有方法都可以通过被覆写的方式来实现对原始字节代码的修改。

下面通过一个具体的示例来进行说明。该示例的场景是在原始的 Java 类中添加静态域来跟踪该 Java 类被创建出来的对象实例的个数。在原始的 Java 类实现中并没有这样的能力。出于调试的目的，可以修改原始类的字节代码，添加这样的功能。从实现的角度来说，只需要在类中添加一个公开的静态变量作为计数器，并在构造方法中修改计数器的值即可。每次构造方法被调用时，计数器的值会加 1。从源代码的角度来讲，进行这样的修改并不困难，但修改已有的字节代码需要比较复杂的实现。主要的复杂性体现在如何找到正确的添加方式，否则会造成字节代码无法运行的后果。

比较合适的做法是先编写修改之后的字节代码对应的 Java 源代码，再将其编译成字节代码。使用 ASM 提供的工具来比较修改前后的字节代码的差异。通过这些差异，可以知道要做出的修改。代码清单 8-16 中给出了完整的示例。在覆写 ClassAdapter 类的 visit 方法的实现中，通过 visitField 方法创建出保存对象实例数量的 instanceCount

域。在 visitMethod 方法的实现中，如果当前方法的名称是“<init>”，即构造方法，则返回一个继承自 MethodAdapter 类的 UpdateInstanceCounterAdapter 类的对象。在 UpdateInstanceCounterAdapter 类中通过覆写 visitInsn 方法来添加相关的指令。如果当前指令代码是与方法返回或异常抛出相关的，说明当前指令是方法返回前的最后一条指令。在此指令之前添加修改类静态域 instanceCount 值的指令。

代码清单 8-16 记录 Java 类的对象实例个数的字节代码修改方式

```

public class InstanceCounter extends ClassAdapter implements Opcodes{
    private static class UpdateInstanceCounterAdapter extends MethodAdapter
        implements Opcodes {
        private String className;

        public UpdateInstanceCounterAdapter(String className, MethodVisitor mv) {
            super(mv);
            this.className = className;
        }

        public void visitInsn(int opcode) {
            if ((opcode >= IRETURN && opcode <= RETURN) || opcode == ATHROW) {
                mv.visitFieldInsn(GETSTATIC, className, "instanceCount", "I");
                mv.visitInsn(ICONST_1);
                mv.visitInsn(IADD);
                mv.visitFieldInsn(PUTSTATIC, className, "instanceCount", "I");
            }
            mv.visitInsn(opcode);
        }

        public void visitMaxs(int maxStack, int maxLocals) {
            mv.visitMaxs(maxStack + 2, maxLocals);
        }
    }

    private String className;

    public InstanceCounter(ClassVisitor cv) {
        super(cv);
    }

    public void visit(int version, int access, String name, String signature,
        String superName, String[] interfaces) {
        cv.visit(version, access, name, signature, superName, interfaces);
        className = name;
        FieldVisitor fv = cv.visitField(ACC_PUBLIC + ACC_STATIC, "instanceCount",
            "I", null, null);
        fv.visitEnd();
    }

    public MethodVisitor visitMethod(int access, String name, String desc, String

```

```

        signature, String[] exceptions) {
    MethodVisitor mv = cv.visitMethod(access, name, desc, signature,
        exceptions);
    if ("<init>".equals(name)) {
        mv = new UpdateInstanceCounterAdapter(className, mv);
    }
    return mv;
}

public static void main(String[] args) throws IOException {
    ClassReader reader = new ClassReader("com.java7book.chapter8.asm.
        CreatedObject");
    ClassWriter writer = new ClassWriter(0);
    InstanceCounter counter = new InstanceCounter(writer);
    reader.accept(counter, 0);
    byte[] byteCode = writer.toByteArray();
    Files.write(Paths.get("bin", "com", "java7book", "chapter8", "asm",
        "CreatedObject.class"), byteCode);
}
}

```

在运行时，创建一个 ClassReader 类的对象来读取原始的字节代码，同时创建一个 ClassWriter 类的对象来输出修改之后的字节代码。进行修改操作的 InstanceCounter 类使用 ClassWriter 类的对象作为参数。ClassReader 类的对象产生的事件在经过 InstanceCounter 类的对象处理之后，被传递给 ClassWriter 类的对象。由 ClassWriter 类的对象负责修改后的字节代码的生成。

4. 树形 API

除了基本的事件 API 之外，ASM 中也有树形 API。树形 API 把字节代码的信息读取到内存中，用不同的对象来表示。树形 API 相对于事件 API 来说，所需的内存更多，运行速度也更慢。不过在某些情况下，树形 API 有自己的优势。对字节代码进行的某些操作可能需要获取一些字节代码相关的全局信息。这些信息通过事件 API 是不能在一次操作中获取的，因为这些信息只有在处理全部结束之后才能统计出来，所以，事件 API 至少需要两次处理才能完成。第一次收集信息，第二次进行实际的处理。而树形 API 由于已经把全部信息读入内存，因此获取全局信息是很容易的。

同样是计算类中包含的方法的个数，相对于代码清单 8-12 中给出的使用事件 API 的实现来说，使用树形 API 则简单很多，如代码清单 8-17 所示。

代码清单 8-17 使用树形 API 计算方法个数

```

public class TreeMethodCounter {
    public int count(String className) throws IOException {
        ClassReader reader = new ClassReader(className);
        ClassNode cn = new ClassNode();
        reader.accept(cn, 0);

```

```
        return cn.methods != null ? cn.methods.size() : 0;
    }

    public static void main(String[] args) throws IOException {
        TreeMethodCounter counter = new TreeMethodCounter();
        int count = counter.count("java.lang.String");
        System.out.println(count);
    }
}
```

ASM中树形API的Java类都在org.objectweb.asm.tree包中。对于字节代码中出现的各种结构，都有Java类与之对应。如ClassNode类表示一个Java类或接口，FieldNode类和MethodNode类分别表示域和方法。ClassNode类实现了ClassVisitor接口。使用ClassNode类的对象访问一个ClassReader类的对象读取的字节代码，可以得到字节代码在内存中的完整表示形式。通过ClassNode类的对象可以访问字节代码中包含的各种结构，如ClassNode类的对象的methods域表示的是包含所有方法的MethodNode类的对象的列表。

8.3.2 增强代理

ASM工具的一种典型用法是在字节代码被使用之前进行处理。一般的做法是在程序的构建过程中，在源代码被编译之后，再运行相关的程序来对字节代码进行处理。如果需要在运行时进行处理，可以使用类加载器来完成。不过类加载器的方式相对比较复杂。一种更简单的做法是使用J2SE 5.0中引入的java.lang.instrument包提供的API。java.lang.instrument.Instrumentation接口中实现了一种在运行时对类的字节代码进行转换的能力。

1. 增强代理的基本用法

对字节代码的转换操作由特殊的代理程序来完成。代理程序是一个jar包。在该jar包的清单文件中定义了启动代理的Java类名称。不同的虚拟机实现提供的启动代理的方式不尽相同，一般有两种实现方法。第一种做法是通过虚拟机的启动参数指定代理程序jar包的路径。所用的参数是“-javaagent”，如“-javaagent:myAgent.jar”。对于这种方式，代理程序的jar包的清单文件中要包含Premain-Class的属性，用来指定一个Java类。该类中必须包含一个premain方法。在虚拟机启动之后，代理程序类中的premain方法会被调用，然后才是程序的主Java类的main方法被调用。在premain方法的参数中可以得到Instrumentation接口的实现对象。第二种做法是在虚拟机运行主程序之后，再启动代理程序。这类代理程序的jar包的清单文件中要由Agent-Class属性指明代理类的名称。当代理类被加载之后，虚拟机会尝试调用类中的agentmain方法。该方法的参数类型与premain相同。下面的介绍都是针对第一种方式进行的。

在代理类的premain或agentmain方法中可以获取作为参数传递的Instrumentation

接口的实现。该接口中的重要方法是 addTransformer，用来添加 java.lang.instrument.ClassFileTransformer 接口的实现对象。ClassFileTransformer 接口只有一个方法 transform，用来对字节代码进行处理，返回处理的结果。代码清单 8-18 中的 TraceTransformer 类的作用是对字节代码中的方法进行处理，在方法的开始处插入额外的指令，用 System.out 方法输出当前方法的名称。经过 TraceTransformer 类的转换之后，类中的方法在被调用时都会先输出方法的名称。对字节代码修改时使用了 ASM 工具。

代码清单 8-18 追踪方法调用的字节代码转换代理

```

public class TraceTransformer implements ClassFileTransformer {
    public byte[] transform(ClassLoader loader, String className,
                           Class<?> classBeingRedefined, ProtectionDomain protectionDomain,
                           byte[] classfileBuffer) throws IllegalClassFormatException {
        ClassReader reader = new ClassReader(classfileBuffer);
        ClassWriter writer = new ClassWriter(0);
        ClassAdapter adapter = new ClassAdapter(writer) {
            public MethodVisitor visitMethod(int access, final String name, String
                                             desc, String signature, String[] exceptions) {
                MethodVisitor mv = cv.visitMethod(access, name, desc, signature,
                                                 exceptions);
                return new MethodAdapter(mv) {
                    public void visitCode() {
                        mv.visitCode();
                        mv.visitFieldInsn(Opcodes.GETSTATIC, "java/lang/System",
                                         "out", "Ljava/io/PrintStream;");
                        mv.visitLdcInsn("进入方法：" + name);
                        mv.visitMethodInsn(Opcodes.INVOKEVIRTUAL, "java/io/
                                         PrintStream", "println", "(Ljava/lang/String;)V");
                    }
                };
            }
            public void visitMaxs(int maxStack, int maxLocals) {
                mv.visitMaxs(maxStack + 2, maxLocals);
            }
        };
        reader.accept(adapter, 0);
        return writer.toByteArray();
    }
}

```

接下来需要把 TraceTransformer 类添加到代理程序中。代理程序可以是任何 Java 类。对于在虚拟机启动时运行的代理程序，只要包含 premain 方法即可。代码清单 8-19 给出了代理程序的实现。在 premain 方法中调用 addTransformer 方法添加了代码清单 8-18 中的 TraceTransformer 类的对象，通过该对象来进行相应的字节代码转换操作。把 TraceAgent 类和相应的清单文件打包在一个 jar 文件中。在虚拟机启动时添加相关的启

动参数来使用此 jar 文件作为代理程序。

代码清单 8-19 追踪方法调用的代理程序

```
public class TraceAgent {
    public static void premain(String args, Instrumentation inst) {
        inst.addTransformer(new TraceTransformer());
    }
}
```

2. 字节代码的重新转换

有两种类型的字节代码转换器，一种是允许重新进行转换的，另外一种是不允许重新进行转换的。这两种类型通过 `Instrumentation` 接口的 `addTransformer` 方法的调用方式来区分。在调用 `addTransformer` 方法时可以指定一个额外的参数来声明是否允许进行重新转换。对于使用 `addTransformer` 方法添加的转换器，在通过类加载器的 `defineClass` 方法进行定义或使用 `Instrumentation` 接口的 `redefineClasses` 方法重新进行定义时，转换器的 `transform` 方法都会被调用。当类已经被加载后，可以通过 `Instrumentation` 接口的 `retransformClasses` 方法来重新进行转换。在重新转换的过程中，只有允许进行重新转换的转换器类的 `transform` 方法才会被调用。通过 `addTransformer` 方法可以添加多个转换器。在转换过程中，这些转换器按照添加时的顺序级联起来。前一个转换器的输出是下一个转换器的输入。如果转换器不需要对某个类的字节代码进行处理，那么 `transform` 方法直接返回 `null` 即可。

不是所有虚拟机的实现都支持对类的字节代码进行重定义和重新转换操作，也就是说 `Instrumentation` 接口的 `redefineClasses` 和 `retransformClasses` 方法不一定起作用。需要通过 `Instrumentation` 接口的 `isRedefineClassesSupported` 和 `isRetransformClassesSupported` 方法来分别检查虚拟机对于重定义和重新转换的支持能力。除了虚拟机支持之外，代理程序也需要在其 jar 包的清单文件中声明是否需要启用重定义和重新转换类的功能。这是通过清单文件中的 `Can-Redefine-Classes` 和 `Can-Retransform-Classes` 属性来表示的。这两个属性的默认值都为 `false`，需要显式地启用。这两个条件需要同时满足才能完成重定义和重新转换的操作。

由于重定义和重新转换的支持，可以在运行时动态改变类的行为。代码清单 8-19 中的 `TraceAgent` 在所有类被加载之前都会进行转换操作。如果希望在运行时进行转换操作，可以使用代码清单 8-20 中的代理程序。在 `RetransformClassesAgent` 的 `premain` 方法中把 `Instrumentation` 接口的实现对象保存下来。当程序调用 `enableTrace` 方法时，会添加代码清单 8-18 中的 `TraceTransformer` 转换器，再对 `ToBeTraced` 类进行转换操作。转换操作完成之后，`ToBeTraced` 类的行为会即时发生改变。在 `disableTrace` 方法的实现中，先把 `TraceTransformer` 转换器从列表中删除，再重新进行转换。其结果是 `ToBeTraced` 类恢复到了最初的状态。重新转换是必须进行的，否则 `ToBeTraced` 类的定义不会更新。

代码清单 8-20 使用重新转换操作的代理程序

```

public class RetransformClassesAgent {
    static Instrumentation instrumentation;
    static TraceTransformer traceTransformer = new TraceTransformer();
    public static void premain(String args, Instrumentation inst) {
        instrumentation = inst;
    }

    public static void enableTrace() {
        if (instrumentation.isRerunClassesSupported()) {
            instrumentation.addTransformer(traceTransformer, true);
            try {
                instrumentation.retransformClasses(ToBeTraced.class);
            } catch (UnmodifiableClassException e) {
                e.printStackTrace();
            }
        }
    }

    public static void disableTrace() {
        if (instrumentation.isRerunClassesSupported()) {
            instrumentation.removeTransformer(traceTransformer);
            try {
                instrumentation.retransformClasses(ToBeTraced.class);
            } catch (UnmodifiableClassException e) {
                e.printStackTrace();
            }
        }
    }
}

```

通过代码清单 8-21 中给出的方式来使用 ToBeTraced 类。在 method1 方法被调用时，ToBeTraced 类还没有被转换，不会有调试信息输出。随后调用 RetransformClassesAgent 类的 enableTrace 方法进行转换。接下来调用 method2 方法时会输出相关的调试信息。在 disableTrace 方法被调用之后，再次对 ToBeTraced 类进行转换。再次转换完成后，调用 method3 方法时不再输出调试信息。

代码清单 8-21 使用重新转换的代理程序的测试示例

```

ToBeTraced traced = new ToBeTraced();
traced.method1();
RetransformClassesAgent.enableTrace();
traced.method2();
RetransformClassesAgent.disableTrace();
traced.method3();

```

每次转换操作都从类最初的字节代码开始，也就是说，在没有改变转换器列表的情

况下，多次调用 `retransformClasses` 方法的结果是一样的，并不会出现同一转换操作被应用多次的情况。这也是 `disableTrace` 方法实现的基础。

8.4 注解

注解（annotation）是 J2SE 5.0 引入的对 Java 所做的重大修改之一。注解的含义可以理解为 Java 源代码中的元数据。提到源代码中的元数据，注释（comment）是最为开发人员所熟悉的一种形式。注释用来描述源代码中类、域和方法的作用等。注解与注释的最大不同在于注解会影响源代码的行为，注释则不会。在编译器对源代码进行处理时，注释会被直接删除，而注解则可能保留在字节代码中。在程序中，另外一种常见的元数据形式是以文件形式存在的 XML、JSON 或 YAML 文档等。这些元数据中有些是供用户使用的，这类元数据无法用注解来替代。还有些元数据是供开发人员使用的，用来配置第三方库的行为，这类元数据可以用注解来替代，而且使用注解会更加方便。

注解在第三方库中得到广泛的应用，主要是相对于其他配置方式来说，注解有着突出的优点。注解与源代码紧密结合在一起，而其他配置方式则依赖与源代码分开的外部文件。以 Java EE 开发中常见的对象关系映射为例。当需要声明一个 Java 类为可被持久化的实体时，如果使用外部配置文件，那么通常需要在该配置文件中指明该 Java 类的全名，以及一些持久化相关的属性。当 Java 类发生变化时，配置文件中的相应部分也需要更新。Java 类和配置文件之间的同步，需要由开发人员自己来维护。在实际开发中，很容易出现不同步的情况，造成错误。例如，在重构过程中修改了 Java 类的名称，但是配置文件并没有进行相应的更新。而在使用注解时，只需在 Java 类中添加相应的注解来包含配置信息即可。所有的修改都在一个 Java 类的源代码中完成。这种统一性使开发人员更容易管理代码中的变化。

8.4.1 注解类型

在创建和使用注解之前，要先对注解类型有一定的了解。注解类型是一种特殊的接口，在声明时使用的是“`@interface`”而不是“`interface`”。注解类型不能添加泛型声明，也不能使用 `extends` 关键词来继承自另外的接口。所有注解类型都继承自 `java.lang.annotation.Annotation` 接口，但是这个继承关系是隐式的。`Annotation` 接口的作用是为注解类型提供一些公用方法，包括常用的 `equals`、`hashCode` 和 `toString` 方法。除此之外，`Annotation` 接口的 `annotationType` 方法可以返回当前的注解类型。需要注意的是 `Annotation` 接口本身并不是注解类型。直接继承自 `Annotation` 接口并不能创建新的注解类型。已有注解类型的子类或子接口也不是注解类型。创建注解类型必须通过“`@interface`”声明。除了在继承关系上的限制之外，注解类型与普通的接口没有其他区别。

一个注解类型中可以包含多个元素。每个元素都可以看成是注解中包含的配置项，通过方法声明的形式来定义。这些方法的声明中不能有任何形式参数或类型参数，也不

能有抛出受检异常的 throws 声明。方法的名称是元素的名称，方法的返回值类型决定了元素的类型。方法的返回值类型只能是基本类型、String 类型、Class 类型、枚举类型、注解类型和数组类型。数组类型中元素的类型只能是非数组类型，不支持多维数组。注解类型中可以没有任何元素，这种注解类型被称为标记注解类型，是作标记用的。如果注解类型中只有一个元素，那么这个元素的名称应该根据惯例使用 value。使用 value 的好处是可以简化注解的使用方式，可以为注解类型中的元素设定默认值，通过在方法声明之后的 default 关键词来指定。

Java 标准库中提供了一些可以直接使用的注解类型。这些注解类型有两类：一类是配置注解类型本身行为的元注解，另外一类是与 Java 语言某些特性相关的一般注解。先从元注解说起。

元注解 `java.lang.annotation.Target` 的作用是配置注解类型可以应用的程序元素。Java 源程序中的很多元素都可以添加注解。某些注解类型有其适用的程序元素的类型。比如某些注解类型只对方法有意义，可以通过 Target 元注解来声明这一点。如果开发人员错误地把该注解应用在域上，那么编译器会产生相关的错误。如果不通过 Target 元注解来声明，则注解类型可以应用在任何程序元素上。Target 注解只有一个可配置元素，是枚举类型 `java.lang.annotation.ElementType` 的数组。ElementType 中的值包括 TYPE、ANNOTATION_TYPE、PACKAGE、CONSTRUCTOR、FIELD、METHOD、PARAMETER 和 LOCAL_VARIABLE，分别表示类型声明、注解类型声明、包声明、构造方法声明、域声明、方法声明、方法参数声明和方法中局部变量声明。值相同的 ElementType 枚举类型不能在同一个 Target 注解的声明中出现多次。

另一个元注解是 `java.lang.annotation.Retention`，表示注解的保留策略。源代码中声明的注解可以有不同的保留策略。这些不同的策略由枚举类型 `java.lang.annotation.RetentionPolicy` 来表示。总共有三种不同的策略，分别是 SOURCE、CLASS 和 RUNTIME。如果保留策略是 SOURCE，则注解声明不会出现在字节代码中；如果保留策略是 CLASS 或 RUNTIME，注解声明会出现在字节代码中，注解是声明在方法中的局部变量上的情况除外。局部变量上的注解在任何情况下都不会保留在字节代码中。如果保留策略是 RUNTIME，那么注解声明在运行时是可用的，可以通过反射 API 来获取注解的相关信息。注解类型应该选择合适的保留策略。默认的策略是 CLASS。

第三个元注解是 `java.lang.annotation.Inherited`，表示注解类型对应的注解声明是被继承的。对于可以被继承的注解类型的注解声明，在进行查询时，如果当前类没有直接声明此注解，会继续查询其父类，直到找到该注解声明或查询到 Object 类。Inherited 是一个标记注解，不包含配置元素。Inherited 注解类型对除类型声明之外的其他程序元素不起作用。

除了元注解之外，Java 标准库还提供了几个一般的注解，主要与 Java 语法相关。第一个是 `java.lang.Override`，用来表示一个方法声明覆写其父类型中的对应方法。Override 注解的作用是避免由于开发人员没有正确区分方法重载和覆写而带来的错误。当需要覆

写一个方法的时候，可以在方法声明上添加 `Override` 注解。如果这个方法实际上并没有覆写父类型中的方法，而是进行了重载，那么编译器会产生相应的错误信息。在代码清单 8-22 中，`User` 类的 `equals` 方法的本意是覆写 `Object` 类的 `equals` 方法，提供自己的对象比较方式的实现，而实际上 `User` 类中的 `equals` 方法并没有进行覆写，而是提供了 `equals` 方法的另外一种重载形式，这是在编程过程中很难发现的错误。

代码清单 8-22 错误的方法覆写方式

```
public class User {
    public boolean equals(User user) {
        // 比较操作
    }
}
```

如果代码清单 8-22 中的 `equals` 方法加上了 `Override` 注解，编译器会给出错误信息，开发人员会意识到这个错误，并根据需要做出相应的修改。

另外一个一般注解是 `java.lang.Deprecated`，用来声明一个程序元素已经被废弃，不应该继续使用。当使用已经被废弃的程序元素时，编译器会产生相关的错误信息。注解 `java.lang.SuppressWarnings` 用来阻止编译器产生某些编译错误。

8.4.2 创建注解类型

除了标准库提供的注解类型外，开发人员可以根据需要创建程序中要用的注解类型。创建注解类型一般有两个要素：第一个是确定注解中包含的元素，即可能的配置选项；另外一个是确定注解类型的元注解。代码清单 8-23 给出了一个名为 `Author` 的注解类型，用来表示作者信息。该注解类型中的配置元素包括作者的姓名、电子邮件地址和是否启用电子邮件通知，其中 `enableEmailNotification` 通过 `default` 关键词声明默认值为 `true`。在该注解类型上，通过 `@Target` 声明了该注解类型只适用于源代码中的类型声明，通过 `@Retention` 声明了该注解只保留在源代码中，不会在字节代码中出现。

代码清单 8-23 注解类型示例

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.SOURCE)
public @interface Author {
    String name();
    String email();
    boolean enableEmailNotification() default true;
}
```

在注解类型的声明中，除了表示配置元素的方法之外，还可以包含常量声明、类和接口声明、枚举类型声明及注解类型声明。代码清单 8-24 给出了在注解类型声明中使用枚举类型的示例。由于注解类型 `Employee` 只包含一个元素，使用 `value` 作为该元素的名称。

代码清单 8-24 注解类型声明中使用枚举类型的示例

```

@Target(ElementType.TYPE)
public @interface Employee {
    enum EMPLOYEE_TYPE {REGULAR, CONTRACT};
    EMPLOYEE_TYPE value();
}

```

8.4.3 使用注解类型

注解类型的使用并不复杂，只需要在源代码的程序元素上以“@”加注解类型名称的方式进行声明即可。如果注解类型中包含了元素，则需要对所有不包含默认值的元素都指定一个值。比如，要把代码清单 8-23 中的注解类型 Author 应用到一个 Java 类上，可以使用代码清单 8-25 中给出的方式。Author 中的元素 name 和 email 没有默认值，需要在使用注解类型时显式指定。对于包含默认值的配置元素也可以指定值。

代码清单 8-25 注解类型的使用示例

```

@Author(name = "Alex", email = "alex@example.org")
public class MyClass {
}

```

如果注解类型中只包含一个元素，且这个元素的名称是 value，那么在使用注解类型时可以省略元素的名称。比如，代码清单 8-24 中的 Employee 注解类型可以按照代码清单 8-26 中的方式来使用。代码清单 8-26 中的用法“@Employee(EMPLOYEE_TYPE.REGULAR)”等价于“@Employee(value = EMPLOYEE_TYPE.REGULAR)”，只是使用上更加简洁。只有当注解类型中使用 value 作为唯一的配置元素的名称时，才能使用这种简洁的写法。

代码清单 8-26 注解类型的简洁使用示例

```

@Employee(EMPLOYEE_TYPE.REGULAR)
public class AnotherClass {
}

```

如果注解类型中包含类型为数组的配置元素，那么在使用时有一些特殊的地方要注意。比如，代码清单 8-27 的注解类型 WithArrayValue 中包含一个类型为 Class<?>[] 的元素。

代码清单 8-27 包含数组类型的注解类型声明

```

@Target(ElementType.TYPE)
public @interface WithArrayValue {
    String name();
    Class<?>[] appliedTo();
}

```

使用该注解类型的方式如代码清单 8-28 所示。对于数组类型的配置元素，使用“{}”来包含数组中具体的值。如果数组中的值只有一个，那么可以省略掉“{}”。注解声明“@WithArrayValue(name = "Test", appliedTo = {String.class})”和“@WithArrayValue(name = "Test", appliedTo = String.class)”在作用上是等价的。

代码清单 8-28 包含数组类型的注解类型的使用示例

```
@WithArrayValue(name = "Test", appliedTo = {String.class, Integer.class})
public class ArrayClass {
}
```

对于标记注解，只需要声明注解类型的名称即可。在使用注解时，善用这些简化的方式可以减少代码量。

8.4.4 处理注解

当注解被添加到 Java 源代码中之后，并不会自动产生作用。某些注解甚至不会在字节代码中出现。创建和使用注解只完成了第一步，更重要的是如何对注解进行处理。在一般情况下，创建和处理注解是 Java 标准库和第三方库应该做的事情，开发人员只需要使用注解即可。以前面提到的 `Override` 注解为例，`Override` 注解类型是 Java 标准库提供的，对它的处理由编译器负责。开发人员只需要在方法声明上添加 `@Override` 的声明来使用它即可。编译器会负责在编译时检查该方法是否的确覆写了父类型中的方法。如果程序中创建了特有的注解，就需要对它进行处理。了解处理注解的方式，对于框架和类库的开发人员来说尤其重要。在一般应用程序开发中也可能需要处理注解。

处理注解通常有两种方式：一种是在编译时，另外一种是在运行时。对于保留策略为 SOURCE 的注解类型来说，编译时是唯一的处理方式。如果注解声明在运行时仍然保留，那么可以通过反射 API 动态地处理。

在编译时处理注解的工作由编译器来完成。对于 Java 标准库中的 `Override`、`SuppressWarnings` 和 `Deprecated` 等注解，编译器知道如何进行处理。而对于程序中自定义的注解类型，则需要程序提供相应的处理器，由编译器在编译时调用。从注解类型被引入的 J2SE 5.0 到 Java 7，自定义注解类型的处理方式发生了很多变化。J2SE 5.0 使用的是 `apt` 工具和 Oracle 私有的 `Mirror API` 来进行处理。Java SE 6 引入了可插拔式注解处理机制及相应的 `javax.annotation.processing` 和 `javax.lang.model` 包，另外 `javac` 工具也提供了对注解处理的支持，不再需要使用 `apt` 工具。Java SE 7 则把 J2SE 5.0 中的 `apt` 工具和对应的 `Mirror API` 声明为被废弃的，不推荐使用。本节主要介绍的是 Java SE 6 中引入的新的注解处理机制，也是推荐使用的注解处理方式。

1. 可插拔式注解处理机制

Java SE 6 中通过 JSR 269 (Pluggable Annotation Processing API) 对注解处理机制进行了标准化。JSR 269 主要包括两个部分，一部分是进行注解处理的 `javax.annotation.`

processing 包，另一部分是对程序的静态结构进行建模的 javax.lang.model 包。这两部分 API 是相辅相成的：javax.annotation.processing 包用来完成实际的注解处理，在处理过程中需要了解程序中被注解的元素的信息。这部分信息由 javax.lang.model 包来表示。对 javax.lang.model 包的介绍会在说明 javax.annotation.processing 包时穿插进行。

JSR 269 的注解处理机制的重要特征是采用可插拔的设计方式。由底层的工具提供基本的框架和运行环境，开发人员编写的注解处理功能作为插件嵌入到此框架之中，并利用框架提供的功能完成所需的处理。对于注解的处理分多轮来进行。在每轮处理中，注解处理器会对源代码和字节代码文件中发现的部分注解进行处理，并可能产生新的源代码和字节代码文件。上一轮处理的输出作为下一轮处理的输入，按照顺序来完成。第一轮处理的输入由运行编译器时的参数来确定。

所有的注解处理器都需要实现 javax.annotation.processing.Processor 接口。注解处理器一般选择继承自 javax.annotation.processing.AbstractProcessor 类。所有实现 Processor 接口的类需要提供一个公开的不带参数的构造方法。注解处理框架会使用此构造方法来创建处理器的对象实例。实例创建完成后，Processor 接口的 init 方法会被调用，以完成处理器的初始化。init 方法声明中包含类型为 javax.annotation.processing.ProcessingEnvironment 接口的参数。ProcessingEnvironment 接口表示注解处理时的运行环境，提供了一些方法来创建新文件、报告错误以及获取其他实用工具类。注解处理器的实现对象通过 ProcessingEnvironment 接口与框架进行交互。在调用 init 方法时，ProcessingEnvironment 接口的实现对象由框架提供。一般把此对象保留下，供后续操作使用。接着框架调用 Processor 接口中的方法来获取一些与注解处理器相关的信息，这些方法包括获取处理器支持的注解类型的 getSupportedAnnotationTypes 方法、处理器支持的最高源代码版本号的 getSupportedSourceVersion 方法和处理器所能识别的选项名称的 getSupportedOptions 方法。这些方法对一个处理器实例只会调用一次。在每轮注解处理过程中，框架会根据当前源代码中具有的注解声明和每个处理器所能处理的注解类型来确定由哪些处理器来处理。被选中的处理器的 process 方法会被调用。在调用时，process 方法的第一个参数表示的是待处理的注解类型，第二个参数表示的是可以用来获取本轮处理的相关环境信息的 javax.annotation.processing.RoundEnvironment 接口的实现对象。当没有可供处理的注解声明或是其他匹配的处理器时，本轮处理结束。当所有轮处理都完成时，整个注解处理过程结束。

下面通过一个简单的示例开始对注解处理器的介绍。需要处理的注解是代码清单 8-23 中的 Author。Author 注解被添加在 Java 源代码中的类或接口上，提供类或接口的作者的相关信息。对 Author 注解进行处理的目的统计每个作者所创建的类或接口的数量。代码清单 8-29 中给出了完整的处理器实现。AuthorProcessor 类继承自 AbstractProcessor 类。AbstractProcessor 类对 Processor 接口的大部分方法都提供了默认实现，继承者只需要实现 process 方法即可。在 AbstractProcessor 类的 init 方法中，把之后的处理中需要用到的 ProcessingEnvironment 接口的实现对象保存在受保护的域 processingEnv 中，

AbstractProcessor 类的子类可以直接使用该域来引用这个对象。在 AuthorProcessor 类上添加了注解 javax.annotation.processing.SupportedSourceVersion 和 javax.annotation.processing.SupportedAnnotationTypes。AbstractProcessor 类可以处理这些注解，并作为 Processor 接口中 getSupportedSourceVersion 和 getSupportedAnnotationTypes 方法的实现。同样 AbstractProcessor 类还支持 javax.annotation.processing.SupportedOptions 注解作为 getSupportedOptions 方法的返回值。AbstractProcessor 类的这个能力简化了对注解处理器所支持的最高源代码版本号、所支持的注解类型和所能识别的额外选项这三个配置的处理。对 AuthorProcessor 类来说，支持的最高源代码版本是 7，而支持的注解类型只有一种，即 com.java7book.chapter8.annotation.Author。在声明所支持的注解类型时，可以使用通配符 “*”。单个 “*” 字符表示可以处理所有的注解类型。

在 process 方法的实现中，通过 RoundEnvironment 接口可以获取与本轮处理相关的信息。其中，通过 getRootElement 方法可以获取前一轮处理完成之后的可供处理的程序元素集合。程序元素由 javax.lang.model.element.Element 接口表示。Java 源代码中包含的各种静态结构都由此接口或其子接口来表示。在 RoundEnvironment 接口提供的方法中，比较常用的是 getElementsAnnotatedWith 方法，用来获取包含指定注解类型声明的元素的集合。对于每个包含注解类型声明的元素，使用 getAnnotationMirrors 方法可以得到它所包含的所有注解。每个注解用 javax.lang.model.element.AnnotationMirror 接口表示。遍历所有这些注解，根据名称找到其中的 Author 注解。通过 AnnotationMirror 接口的 getElementValues 方法可以获取注解中包含的所有配置元素的值。遍历这些配置元素，可以找到元素 name 的值，即作者的姓名，之后即可对作者姓名进行统计。

由于注解处理的过程可能要经过多轮来完成，因此一个处理器的 process 方法会被多次调用。如果在某轮处理中，一个处理器被调用了，那么在后续的每轮处理中，即便没有注解可供该处理器来处理，也会调用该处理器。在 process 方法中可以通过 RoundEnvironment 接口的 processingOver 方法进行判断。在一轮处理中，如果 processingOver 方法返回 true，则说明这是处理的最后一轮。如果本轮的处理过程依赖前一轮的执行结果，那么通过 RoundEnvironment 接口的 errorRaised 方法可以判断前一轮的处理是否出现错误。需要注意 process 方法的返回值的使用。如果返回值为 true，说明对这些注解类型的处理由当前处理器独占完成，其他的处理器不会再进行处理；如果返回值为 false，则其他处理器仍然有机会进行处理。如果某个处理器声明的注解处理类型是 “*”，同时又在 process 方法中返回 true，则相当于独占处理了所有的注解类型。其他处理器都不会得到处理的机会。

代码清单 8-29 Author 注解的处理器

```

@SupportedSourceVersion(SourceVersion.RELEASE_7)
@SupportedAnnotationTypes("com.java7book.chapter8.annotation.Author")
public class AuthorProcessor extends AbstractProcessor {
    private Map<String, Integer> countMap = new HashMap<>();

```

```

private TypeElement authorElement;
public synchronized void init(ProcessingEnvironment processingEnv) {
    super.init(processingEnv);
    ElementUtils elementUtils = processingEnv.getElementUtils();
    authorElement = elementUtils.getTypeElement("com.java7book.chapter8.
        annotation.Author");
}

public boolean process(Set<? extends TypeElement> annotations,
    RoundEnvironment roundEnv) {
    Set<? extends Element> elements = roundEnv.getElementsAnnotatedWith(autho
        rElement);
    for (Element element : elements) {
        processAuthor(element);
    }
    if (roundEnv.processingOver()) {
        for (Map.Entry<String, Integer> entry : countMap.entrySet()) {
            System.out.println(entry.getKey() + " ==> " + entry.getValue());
        }
    }
    return true;
}

private void processAuthor(Element element) {
    List<? extends AnnotationMirror> annotations = element.
        getAnnotationMirrors();
    for (AnnotationMirror mirror : annotations) {
        if (mirror.getAnnotationType().asElement().equals(authorElement)) {
            String name = (String) getAnnotationValue(mirror, "name");
            if (!countMap.containsKey(name)) {
                countMap.put(name, 1);
            }
            else {
                countMap.put(name, countMap.get(name) + 1);
            }
        }
    }
}

private Object getAnnotationValue(AnnotationMirror mirror, String name) {
    Map<? extends ExecutableElement, ? extends AnnotationValue> values =
        mirror.getElementValues();
    for (Map.Entry<? extends ExecutableElement, ? extends AnnotationValue>
        entry : values.entrySet()) {
        ExecutableElement elem = entry.getKey();
        AnnotationValue value = entry.getValue();
        String elemName = elem.getSimpleName().toString();
        if (name.equals(elemName)) {
            return value.getValue();
        }
    }
}

```

```
    }
    return null;
}
}
```

在得到包含注解的 Element 接口的实现对象之后，有两种方式可以获取该对象上包含的注解。一种方式是使用代码清单 8-29 中给出的 getAnnotationMirrors 方法。该方法返回的是所有注解的列表，需要遍历该列表进一步找到所需的注解。该方法返回注解列表中包含直接出现在元素中的注解，并不包含继承下来的注解。另一种方式是使用 Element 接口的 getAnnotation 方法。该方法接受一个注解类型的 Class 类的对象作为参数，返回元素上对应此类型的注解。通过这个方法得到的注解可能是直接出现的，也可能是继承而来的。这两个方法的显著区别在于注解信息的来源不同：getAnnotationMirrors 方法使用的是源代码中出现的程序静态结构中的信息，getAnnotation 方法使用的则是运行时通过反射得到的信息。尤其要注意这两个方法在处理注解中 Class 和 Class[] 类型的配置元素上的区别。如果注解中元素的类型是 Class 或 Class[]，那么通过 getAnnotation 方法得到的注解中元素的值不能直接使用。任何尝试使用得到的 Class 和 Class[] 类型的对象的操作，都会抛出异常。这是因为缺乏加载对应 Java 类所需的信息，无法定义出对应的 Class 类的对象。而通过 getAnnotationMirrors 方法可以得到相关的信息。每个 Java 类由 javax.lang.model.type.TypeMirror 接口来表示。

注解处理器的运行由 Java 编译器来完成。可以通过两种方式来声明在编译时要运行的注解处理器。一种方式是通过 javac 命令行工具的“-processor”参数来指定注解处理器的类名。另外一种方式是使用注解处理器的自动发现机制。编译器在编译时的类路径（CLASSPATH）中查找路径名为“META-INF/services/javax.annotation.processing.Processor”的文件。这个文件中的每一行都包含要运行的注解处理器的类名。除了类路径外，还可以通过编译器的“-processorpath”参数来显式指定查找路径。如果不希望进行注解处理，那么可以使用“-proc:none”参数来禁用。

2. 创建和修改源代码

之前介绍的注解处理器只是从注解中提取出所需的相关信息，并没有对 Java 源代码本身做出修改。实际上，在注解处理器中既可以创建新的 Java 源文件、字节代码文件和资源文件，又可以对已有的源代码进行修改。创建新文件的操作由从 ProcessingEnvironment 接口得到的 javax.annotation.processing.Filer 接口的实现对象来完成。对已有源代码的修改则由 Java 编译器提供的内部 API 来完成。

下面先介绍如何生成一个新的 Java 源代码文件。代码清单 8-30 给出了在注解处理器中生成新 Java 源代码的示例。其中，filer 是在 init 方法中通过 ProcessingEnvironment 接口的 getFiler 方法得到的 Filer 接口的实现对象。通过 Filer 接口的 createSourceFile 方法创建一个新的 JavaFileObject 接口的实现对象，表示一个新的 Java 源文件。不过创建

完成后，JavaFileObject 接口的实现对象所表示的文件内容是空的。通过 JavaFileObject 接口的 openOutputStream 方法得到输出文件内容所需的 OutputStream 类的对象，再把源文件的内容写入即可。这里只写入了一个简单的 Java 类的源代码内容。

代码清单 8-30 注解处理器中生成源代码的示例

```

private void generateSource() throws IOException {
    String packageName = "com.java7book.chapter8.annotation";
    String className = "HelloWorld";
    String fullName = packageName + "." + className;
    JavaFileObject javaFile = filer.createSourceFile(fullName, (Element[]) null);
    Writer writer = new OutputStreamWriter(javaFile.openOutputStream(), "UTF-8");
    String source = getSource(packageName, className);
    writer.write(source);
    writer.close();
}

private String getSource(String packageName, String className) {
    StringBuilder builder = new StringBuilder();
    builder.append("package " + packageName + ";");
    builder.append("public class " + className + "{}");
    return builder.toString();
}

```

新的 Java 源文件被创建出来之后，Java 编译器会对新文件进行编译。由于注解处理发生在编译之前，创建新的 Java 源文件的做法适合于自动代码生成的场景。在已有的代码中通过注解的方式来添加元数据，确定代码生成的逻辑。在注解处理过程中完成代码的生成。这种自动的方式解决了手工生成可能带来的变化不一致的问题。不过，由于生成的代码在编译之前对于程序中的其他部分来说是未知的，因此需要某些机制来保证这些代码可以被已有的代码使用。一般是在生成的代码中调用已有代码中的逻辑。

下面介绍如何对已有的源代码进行修改。对源代码进行修改需要分析源代码的结构，得到对应的抽象语法树，再对其中的元素进行修改。OpenJDK 中使用的 Java 编译器的源代码是开放的，可以在 Java 程序中使用 Java 编译器所提供的 API 来对 Java 源代码进行分析。相关的 API 在“com.sun.tools.javac”包中。由于编译器的实现不是 Java 标准的一部分，它的 API 可能在不同版本之间发生变化，因此使用时要考虑这一点。也可以不对源代码进行语法分析，而是把源文件当成字符串来进行处理。这种做法的优势是简单易用，不需要附加库的支持，但是有可能出现语法错误。下面的介绍中使用的是 OpenJDK 中 Java 编译器的 API。

先介绍一下示例的背景。Java 语言提供了不同的访问控制级别，包括 public、private 和 protected 等。一般 Java 类内部的域和方法使用 private 来修饰，外部的 Java 类无法进行访问。这种做法的好处是提高了 Java 类的封装性，但是在进行测试的时候，无法对这些私有方法进行直接测试，只能通过 Java 类的公开方法进行间接测试。从方便测

试的角度出发，可以把原始代码中的私有域和方法修改成公开的。修改之后的代码只为测试所用。

为了允许把代码中的私有域和方法的声明修改为公开的，需要增加一个新的注解类型 `Visible`。该注解可以添加在需要进行转换的 Java 类上。完整的注解处理器的实现如代码清单 8-31 所示。通过编译器的 API，可以得到表示程序元素的 `Element` 接口的实现对象所对应的 `com.sun.tools.javac.tree.JCTree` 类的对象。`JCTree` 类的对象是一个树形结构，可以在上面添加访问者来进行处理。而 `com.sun.tools.javac.tree.TreeTranslator` 类是一个用来进行源代码转换的访问者实现。`VisibilityTranslator` 类所实现的是在 Java 源代码中遇到域声明时，把它的访问控制修饰符的值直接设置为 1，即 `public` 声明对应的标记值。经过这样的处理之后，源代码中的域都变成可以公开访问的。

代码清单 8-31 把域的声明修改为 public 的注解处理器

```

@SupportedSourceVersion(SourceVersion.RELEASE_7)
@SupportedAnnotationTypes("com.java7book.chapter8.annotation.Visible")
public class VisibilityProcessor extends AbstractProcessor {
    private TypeElement visibleElement;
    private Trees trees;
    private TreeMaker treeMaker;

    public synchronized void init(ProcessingEnvironment processingEnv) {
        super.init(processingEnv);
        trees = Trees.instance(processingEnv);
        Context context = ((JavacProcessingEnvironment) processingEnv).
            getContext();
        treeMaker = TreeMaker.instance(context);
        Elements elementUtils = processingEnv.getElementUtils();
        visibleElement = elementUtils.getTypeElement("com.java7book.chapter8.
            annotation.Visible");
    }

    public boolean process(Set<? extends TypeElement> annotations,
        RoundEnvironment roundEnv) {
        if (!roundEnv.processingOver()) {
            Set<? extends Element> elements = roundEnv.getElementsAnnotatedWith(v
                isibleElement);
            for (Element element : elements) {
                JCTree tree = (JCTree) trees.getTree(element);
                TreeTranslator visitor = new VisibilityTranslator();
                tree.accept(visitor);
            }
        }
        return true;
    }

    private class VisibilityTranslator extends TreeTranslator {
        public void visitVarDef(JCVariableDecl def) {

```

```
super.visitVarDef(def);
JCVariableDecl pub = treeMaker.VarDef(treeMaker.Modifiers(1), def.name,
    def.vartype, def.init);
result = pub;
}
}
```

3. 使用反射 API 处理注解

如果不在编译之前对注解进行处理，那么另外一种做法是可以在运行时通过反射 API 来利用注解中提供的信息。在反射 API 中，`java.lang.reflect.AnnotatedElement` 接口表示包含注解的元素。`java.lang` 包中的 `Class` 和 `Package`，以及 `java.lang.reflect` 包中的 `Constructor`、`Field` 和 `Method` 都实现了 `AnnotatedElement` 接口。`AnnotatedElement` 接口中的 `getAnnotations` 方法用来获取当前元素上的所有注解，包括直接出现的和继承而来的，而 `getDeclaredAnnotations` 方法只返回直接出现的注解。使用 `getAnnotation` 方法可以根据注解类型来查找相应的注解声明。在得到注解之后，可以调用其中的方法来获得包含的配置元素的值。

一个典型的使用场景是把注解、反射 API 和动态代理结合起来使用。注解用来设置程序在运行时的行为，反射 API 用来解析注解，而动态代理负责应用具体的行为。相对于动态修改 Java 源代码或字节代码的方式来说，这种做法的实现都包括在程序的源代码中，开发和调试比较简单。

例如，在一个企业的员工管理系统中，对访问控制权限有比较严格的要求，某些操作只能由特定角色的用户来完成。如给员工加薪的操作，只能由具有“经理”角色的用户来完成。这种访问控制的限制一般是分级进行的，除了前端界面显示和后台服务接口需要有相应的权限检查逻辑之外，进行实际操作的业务逻辑实现代码也要有相应的检查逻辑。

从实现的角度来说，访问控制属于程序中横切的非功能性需求，可以由专门的人员负责开发。业务逻辑的实现代码只需要以声明的方式来描述访问控制的需求即可。使用代码清单 8-32 中的 Role 注解类型可以指定调用一个方法时，当前用户要具备的角色。Role 类型的保留策略要设置为 RUNTIME，这样可以在运行时通过反射 API 来查找到该注解的信息。

代码清单 8-32 声明方法调用时所需用户角色的注解类型

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Role {
    String[] value();
}
```

代码清单 8-33 在对员工信息进行操作的 EmployeeInfoManager 接口的 updateSalary 方法中添加了 Role 注解，声明调用此方法时，当前用户需要具有“manager”角色。

代码清单 8-33 修改员工薪酬的接口

```
public interface EmployeeInfoManager {
    @Role("manager")
    public void updateSalary();
}
```

在 EmployeeInfoManager 接口的实现类中并不需要考虑访问控制的问题，具体的访问控制是由动态代理来负责的。代码清单 8-34 中的 AccessInvocationHandler 类负责进行调用时的检查。如果方法中添加了 Role 注解，则获取此注解中包含的值，即调用此方法所应具备的角色。通过与程序中保存的当前用户的角色进行比较，可以判断该调用是否应该进行。使用者通过工厂方法得到 EmployeeInfoManager 接口的实现对象，该对象是一个封装了访问控制权限检查逻辑的动态代理对象。

代码清单 8-34 用来获取修改员工薪酬的代理对象的工厂类

```
public class EmployeeInfoManagerFactory {
    private static class AccessInvocationHandler<T> implements InvocationHandler {
        private final T targetObject;

        public AccessInvocationHandler(T targetObject) {
            this.targetObject = targetObject;
        }

        public Object invoke(Object proxy, Method method, Object[] args)
                throws Throwable {
            Role annotation = method.getAnnotation(Role.class);
            if (annotation != null) {
                String[] roles = annotation.value();
                String currentUserRole = AccessManager.getCurrentUserRole();
                if (!Arrays.asList(roles).contains(currentUserRole)) {
                    throw new RuntimeException("没有调用此方法的权限。");
                }
            }
            return method.invoke(targetObject, args);
        }
    }

    public static EmployeeInfoManager getManager() {
        EmployeeInfoManager instance = new DefaultEmployeeInfoManager();
        return (EmployeeInfoManager) Proxy.newProxyInstance(instance.getClass()
                .getClassLoader(), new Class<?>[] { EmployeeInfoManager.class }, new
                AccessInvocationHandler<EmployeeInfoManager>(instance));
    }
}
```

8.5 使用案例

下面通过一个具体的案例把字节代码操作、源代码生成和注解处理等技术结合起来，形成完整的解决方案。这个案例涉及 Java 程序的国际化问题，有关 Java 程序国际化的知识在第 4 章中已经做了介绍。国际化实现最主要的问题是需要在代码中将 ResourceBundle 类的使用与属性文件内容保持同步。如果添加了新的字符串，那么需要同时更新 Java 代码和属性文件。如果对属性文件中字符串的键做了修改，也需要对 Java 代码进行相应的修改。一个改动会涉及程序中的多个部分。通过本章介绍的技术，可以把这些变化统一到一个地方，即 Java 源代码本身中。基本的实现思路是在 Java 源代码中通过注解的形式声明所有需要国际化的字符串在属性文件中的键和值，由注解处理器负责收集所有的字符串，动态生成属性文件和相应的使用 ResourceBundle 类的 Java 代码，再通过字节代码修改，在程序中添加调用 ResourceBundle 类的字节代码。开发人员只需要使用注解在源代码中声明需要国际化的字符串即可，其他的工作都是自动完成的。

代码清单 8-35 给出了一个包含需要国际化的字符串的 Java 类的示例。由于注解无法添加在源代码中方法内部的字符串常量上，需要把这些字符串提取出来，包装在一个方法中，然后在方法中添加注解。方法 getTestMessage 的作用只是为了封装字符串，它的返回值并不重要，但是该方法必须包含一个可变长度的 Object 类型的参数，表示构建字符串时的参数。注解类型 Message 的作用是声明方法所封装的字符串在属性文件中的键和值，注解类型 MessageBundle 的作用是声明类使用的属性文件对应的资源包的名称。

代码清单 8-35 使用注解进行国际化字符串声明的代码示例

```

@MessageBundle("Messages")
public class DemoClass {
    public void output() {
        System.out.println(getTestMessage("Hello"));
    }

    @Message(key = "TEST_MESSAGE", value = "This is a test message. %1$s")
    public String getTestMessage(Object... args) {
        return "";
    }
}

```

在相应的注解处理器实现中，扫描所有的 MessageBundle 和 Message 注解，把相关的信息收集起来。接着生成属性文件和使用 ResourceBundle 类的 Java 文件。创建属性文件使用的是 Filer 接口中的 createResource 方法。与代码清单 8-30 中创建 Java 文件的做法相似，得到 FileObject 接口的实现对象之后，再向对应的输出流中写入数据。使用 ResourceBundle 类的 Java 文件也通过自动的方式生成，所生成的 Java 文件如代码清单 8-36 所示。在 Messages 类中，加载了对应的属性文件，并提供 getMessage 方法从属性

文件中获取字符串。

代码清单 8-36 使用 ResourceBundle 类的 Java 源代码

```
public class Messages {
    private static ResourceBundle bundle;

    static {
        bundle = ResourceBundle.getBundle("com.java7book.chapter8.annotation.i18n.
            Messages");
    }

    public static String getMessage(String key, Object... args) {
        String message = bundle.getString(key);
        return String.format(message, args);
    }
}
```

最后一步是对包含 Message 注解的方法的字节代码进行修改，使这些方法不再简单地返回空字符串，而是调用代码清单 8-36 中 Messages 类的 getMessage 方法来返回实际的字符串。这一步操作在编译之后由 ASM 工具来完成。代码清单 8-37 给出了修改方法的字节代码的实现。这里使用的是 ASM 的树形 API。先把当前方法对应的 MethodNode 类的对象中包含的指令全部删除，再添加调用 Messages 类中方法的指令。

代码清单 8-37 修改返回国际化字符串的方法的字节代码

```
private static void updateMethodNode(MethodNode mn, String key) {
    InsnList instructions = mn.instructions;
    Iterator iterator = instructions.iterator();
    while (iterator.hasNext()) {
        iterator.next();
        iterator.remove();
    }
    instructions.add(new LdcInsnNode(key));
    instructions.add(new VarInsnNode(ALOAD, 1));
    instructions.add(new MethodInsnNode(INVOKESTATIC, "com/java7book/chapter8/
        annotation/i18n/Messages", "getMessage", "(Ljava/lang/String;[Ljava/lang/
        Object;)Ljava/lang/String;"));
    instructions.add(new InsnNode(ARETURN));
    mn.maxStack = 2;
    mn.maxLocals = 2;
}
```

在经过这些处理之后，当代码清单 8-35 中的 getTestMessage 方法被调用时，所执行的代码指令是调用代码清单 8-36 中的 Messages 类的 getMessage 方法。而 getMessage 方法会根据国际化字符串在属性文件中的键的名称来查找对应的值，并作为 getTestMessage 方法的返回值。

8.6 小结

Java 源代码和字节代码是 Java 开发中的重要资源。相对于源代码，字节代码的格式对一般开发人员来说更加陌生。本章先从字节代码的格式开始介绍。在编译 Java 源代码时，除了使用 JDK 自带的 javac 命行工具之外，还可以使用 Java 编译器 API 和 Eclipse JDT 编译器。在对字节代码进行处理时，介绍了如何使用 ASM 工具来读取、生成和修改字节代码。通过使用增强代理机制，可以在运行时动态修改字节代码。注解是一种在源代码中添加元数据的方式。本章对如何处理注解进行了详细介绍。最后通过一个使用案例介绍了字节代码操作、源代码生成和注解处理等技术的综合运用。

第 9 章 Java 类加载器

在得到了 Java 程序的字节代码之后，需要通过一种方式把字节代码加载到 Java 虚拟机中运行。Java 选择了一种更加灵活和开放的方式来实现这个加载过程，即类加载器（class loader）。Java 是随着互联网的发展而流行起来的编程语言。在 Java 产生的早期，Applet 可以说是 Java 平台的“杀手级应用”。Applet 的特点是将字节代码存放在远程服务器上，浏览器在运行 Applet 时需要从远程服务器下载字节代码后再运行，因此需要一种新的机制来允许从远程服务器加载字节代码。这种新的机制就是类加载器。类加载器机制是 Java 平台的一个重要创新，它的出现带来了 Java 平台的很多新特性。

类加载器最根本的作用只有一个，即从包含字节代码的字节流中定义出虚拟机中的 Class 类的对象。得到 Class 类的对象之后，一个 Java 类就可以在虚拟机中自由使用，包括创建新的对象或调用类中的静态方法。除了定义类这个最根本的功能之外，类加载器的其他功能都是围绕如何找到类的字节代码展开的。类加载器也是 Java 平台中比较复杂难懂的一部分，尤其被使用在 Web 容器和 OSGi 环境中的时候。

9.1 类加载器概述

Java 中的类加载器并不是以特殊的方式实现的。类加载器本身也是 Java 类。Java 标准库中的 `java.lang.ClassLoader` 类是所有由 Java 代码创建的类加载器的父类。通过调用类加载器中的 `loadClass` 方法可以加载 Java 类。由于类加载器本身也是 Java 类，因此类加载器自身的 Java 类需要由另外的类加载器来加载。注意，类加载器只有自身被加载到虚拟机中之后才能加载其他的 Java 类。这似乎是一个无法解决的循环问题。实际上，Java 平台提供了一个启动类加载器（bootstrap class loader），由原生代码来实现。启动类加载器负责加载 Java 自身的核心类到虚拟机中。在启动类加载器完成初始的加载工作之后，其他继承自 `ClassLoader` 类的类加载器可以正常工作。

一个 Java 类被加载之后，可以通过其对应的 Class 类的对象的 `getClassLoader` 方法来获取加载它的类加载器的对象。如果通过继承 `ClassLoader` 类实现自己的类加载器，那么可以直接创建出类加载器类的对象。`ClassLoader` 类提供的方法比较多，除了加载 Java 类之外，还能加载相关的资源文件，比如图片和属性文件等。`ClassLoader` 类中的一部分方法是声明为受保护的，只能在子类中使用。这些受保护的方法主要是为创建自定义的类加载器而提供的。在使用 `ClassLoader` 类的对象来加载 Java 类时，使用 `loadClass` 方法即可。该方法的参数是 Java 类的二进制名称，返回值是表示该 Java 类的 Class 对象。代码清单 9-1 给出了 `ClassLoader` 类的 `loadClass` 方法的使用示例。在代码中使用

当前 Java 类的类加载器来加载 `java.lang.String` 类。通过这种方式加载的 Class 类的对象与直接使用 `java.lang.String.class` 是一样的。

代码清单 9-1 ClassLoader 类的 `loadClass` 方法的使用示例

```
public void loadClass() throws Exception {
    ClassLoader current = getClass().getClassLoader();
    Class<?> clazz = current.loadClass("java.lang.String");
    Object str = clazz.newInstance();
    System.out.println(str.getClass()); // 输出 java.lang.String
}
```

Java 平台上的类加载器大概可以分成两类：启动类加载器和用户自定义的类加载器。两者的区别在于启动类加载器是由原生代码实现的，而用户自定义的类加载器继承自 `ClassLoader` 类。在用户自定义的类加载器中，一部分类加载器由 Java 平台默认提供，另外一部分由程序自己创建。Java 平台默认提供的用户自定义类加载器有两个：第一个是扩展类加载器（extension class loader），用来从特定的路径加载 Java 平台的扩展库；第二个是系统类加载器（system class loader），又称应用类加载器（application class loader），它的作用是根据应用程序运行时的类路径（CLASSPATH）来加载 Java 类。如果程序中没有使用其他自定义的类加载器，则程序本身的 Java 类都由系统类加载器负责加载。通过 `ClassLoader` 类的静态方法 `getSystemClassLoader` 可以得到系统类加载器对象。通过系统类加载器对象的 `getParent` 方法可以得到扩展类加载器对象。

前面提到过，类加载器的根本作用是从字节代码中定义出表示 Java 类的 `Class` 类的对象。这个定义过程由 `ClassLoader` 类中的 `defineClass` 方法来实现。如果一个 Java 类是由某个类加载器对象的 `defineClass` 方法定义的，则称这个类加载器对象是该 Java 类的定义类加载器（defining class loader）。当使用类加载器对象的 `loadClass` 方法来加载一个 Java 类时，称这个类加载器对象为该 Java 类的初始类加载器（initiating class loader）。一个 Java 类的初始类加载器和定义类加载器不一定相同。初始类加载器可能把实际的加载工作代理给其他类加载器，由后者完成定义 Java 类的工作。初始类加载器和定义类加载器之间的关系是：一个 Java 类的定义类加载器是该类所引用的其他 Java 类的初始类加载器。在通过调用类加载器对象的 `loadClass` 方法进行类的加载，并成功得到一个 `Class` 类的对象之后，虚拟机会把类加载器对象和它加载的 `Class` 类的对象之间的关联记录下来。在加载 Java 类时，虚拟机会先检查是否存在与当前类加载器对象关联在一起的名称相同的 `Class` 类的对象。如果存在，那么直接使用该 `Class` 类的对象，而不会重新进行加载。

9.2 类加载器的层次结构与代理模式

类加载器的一个重要特征是所有类加载器对象都可以有一个作为其双亲的类加载器对象。通过 `ClassLoader` 类的 `getParent` 方法可以获取双亲类加载器对象。`ClassLoader` 类

提供的构造方法允许在创建时指定类加载器对象的双亲类加载器对象。不过 ClassLoader 类本身是抽象的，无法直接创建出 ClassLoader 类本身的对象。自定义类加载器的 Java 类继承自 ClassLoader 类。在自定义类加载器的构造方法中应该调用父类 ClassLoader 类的构造方法，并指定双亲类加载器对象的值。如果在创建时不指定双亲类加载器，则默认双亲类加载器是系统类加载器。如果当前类加载器对象的双亲类加载器是启动类加载器，那么有些虚拟机实现会选择让 getParent 方法返回 null，从而无法获取启动类加载器的对象引用。虚拟机中的类加载器对象按照这种方式组织起来，形成一个树状层次结构。如果程序中大量使用自定义类加载器，这个层次结构会比较复杂。类加载器也可以选择没有双亲类加载器。

如果不使用自定义类加载器，则一个 Java 程序运行时的类加载器通常有 3 个层次，由之前介绍过的 Java 平台上默认提供的类加载器形成，从根节点开始依次是启动类加载器、扩展类加载器和系统类加载器。代码清单 9-2 给出了查看这些类加载器的层次结构的示例。

代码清单 9-2 查看类加载器的层次结构的示例

```
public void displayParents() {
    ClassLoader current = getClass().getClassLoader();
    while (current != null) {
        System.out.println(current.toString());
        current = current.getParent();
    }
}
```

在 Java SE 7 的 OpenJDK 实现中运行此代码，输出的信息如代码清单 9-3 所示。输出的第一个类加载器对象是加载应用程序的系统类加载器，由 sun.misc.Launcher\$AppClassLoader 类表示。输出的第二个类加载器对象是扩展类加载器，由 sun.misc.Launcher\$ExtClassLoader 类表示，它是系统类加载器的双亲类加载器。由于扩展类加载器的双亲类加载器是启动类加载器，getParent 方法返回为 null，因此在代码清单 9-3 中没有输出。

代码清单 9-3 查看类加载器的层次结构的输出结果

```
sun.misc.Launcher$AppClassLoader@177b3cd
sun.misc.Launcher$ExtClassLoader@1bd7848
```

类加载器在加载 Java 类时通常使用代理模式。代理模式指的是一个类加载器对象既可以自己完成 Java 类的定义工作，也可以代理给其他的类加载器对象来完成。在 ClassLoader 类的默认实现中，当类加载器对象需要加载一个 Java 类或资源时，会先把加载请求代理给双亲类加载器对象来完成。只有在双亲类加载器对象无法找到 Java 类或资源时，才由当前类加载器对象进行处理。这种代理关系会沿着类加载器对象的层次结

构树一直向上传递，直到成功加载一个类。在加载类的过程中，依靠双亲类加载器对象的原因是有些类的加载只有双亲类加载器对象才能完成。在程序运行过程中，从类加载器层次结构树的根节点开始，不断有新的类加载器对象被添加进来。对于之后添加的类加载器对象来说，加载类时所需的一些信息对当前类加载器对象来说是不可见的。对于这样的类的加载，只能代理给双亲类加载器对象来完成。不过当前类加载器对象可以选择在不同的时机把加载请求代理给双亲类加载器对象。ClassLoader 类的默认实现使用的是双亲优先的策略，即先由双亲类加载器对象尝试加载，找不到的情况下再由当前类加载器对象来尝试加载。程序可以根据需要采用当前类加载器优先的策略，即先由当前类加载器尝试加载，找不到的情况下再代理给双亲类加载器对象尝试加载；或者根据要加载的 Java 类的名称采取不同的查找策略。

在代码清单 9-4 中，NoParentClassLoader 类在构造方法中设置双亲类加载器为 null，在 testLoad 方法中尝试加载当前包中的一个 Java 类，由于没有双亲类加载器对象可以代理加载类的请求，因此加载过程失败。

代码清单 9-4 没有设置双亲类加载器对象的类加载器

```
public class NoParentClassLoader extends ClassLoader {
    public NoParentClassLoader() {
        super(null);
    }

    public void testLoad() throws ClassNotFoundException {
        Class<?> clazz = loadClass("com.java7book.chapter9.ToLoad");
    }
}
```

9.3 创建类加载器

大部分 Java 程序在运行时并不需要使用自己的类加载器，依靠 Java 平台提供的 3 个类加载器就足够了。在绝大多数时候，也只有系统类加载器发挥作用。如果程序对加载类的方式有特殊的要求，就要创建自己的类加载器。这样的实际应用场景有很多，主要可以分成两类：一类是对 Java 类的字节代码进行特殊的查找和处理，如 Java 类的字节代码存放在磁盘上特定的位置或远程服务器上，或者字节代码的数据经过了加密处理；另一类是利用类加载器产生的隔离特性来满足特殊的需求。

创建自定义类加载器只需继承 ClassLoader 类即可，可以选择覆写其中的某些方法来实现自定义的类加载逻辑。ClassLoader 类中的 defineClass 方法用来从字节代码中定义出表示 Java 类的 Class 类的对象。由于定义 Java 类涉及虚拟机的核心功能，从安全的角度出发，defineClass 方法被声明为 final，不能由 ClassLoader 类的子类来覆写。一般来说，defineClass 方法是由原生代码实现的。在 ClassLoader 类中包含了不少声明为

`protected` 的方法，这些方法是创建自定义类加载器的基础。自定义类加载器通过覆写这些方法来实现特殊的功能。

第一个方法是 `loadClass`。这个方法与 `ClassLoader` 类公开的 `loadClass` 方法同名，但是多一个表示是否对加载的类进行链接操作的参数。在这个声明为 `protected` 的 `loadClass` 方法中封装了默认的双亲类加载器优先的代理模式的实现。默认的实现流程是进行下面的查找过程：先通过 `findLoadedClass` 方法来检查该 Java 类是否已经被加载过，如果已经被加载过了，就直接返回之前加载过的 `Class` 类的对象；接着通过 `getParent` 方法得到双亲类加载器对象，再调用双亲类加载器对象的 `loadClass` 方法，这一步是代理模式生效的地方，如果 `getParent` 方法返回为 `null`，则使用启动类加载器来进行加载；最后调用 `findClass` 方法由当前类加载器对象进行查找。这 3 步依次进行，如果在其中某一步的查找过程中找到了 Java 类的定义，就返回定义的 `Class` 类的对象作为 `loadClass` 方法的返回值。如果尝试所有的步骤后仍然找不到 Java 类的定义，`loadClass` 方法就会抛出 `java.lang.ClassNotFoundException` 异常。如果调用 `loadClass` 方法的第二个参数值为 `true`，即需要对找到的类进行链接操作，则 `loadClass` 方法会调用 `resolveClass` 方法进行链接。

第二个方法是 `findLoadedClass`。在类加载的过程中，虚拟机会记录下已经加载的 Java 类的初始类加载器。在 `findLoadedClass` 方法的实现中，会查找已经加载的 Java 类，比较这些 Java 类的初始类加载器对象和要加载的 Java 类的名称。如果某个已经加载的 Java 类的初始类加载器是当前类加载器对象，同时类的名称也与要加载的类的名称相同，就把该 Java 类对应的 `Class` 类的对象作为结果返回。

第三个方法是 `findClass`。默认情况下，当通过代理模式无法使用双亲类加载器对象成功加载 Java 类时，`findClass` 方法被调用。这个方法主要用来封装当前类加载器对象自己的类加载逻辑。在一般的自定义类加载器中只需要覆写此方法即可。只有在需要改变默认的双亲优先代理模式的类加载器中才需要覆写 `loadClass` 方法。`ClassLoader` 类中的 `findClass` 方法只是简单抛出 `ClassNotFoundException` 异常。

最后一个方法是 `resolveClass`。这个方法的作用是链接一个定义好的 `Class` 类的对象。链接的具体过程将在第 10 章进行介绍。

下面通过几个具体的示例来说明如何创建自定义类加载器。第一个示例是从磁盘上的特定目录加载 Java 类的字节代码的类加载器。代码清单 9-5 中给出了完整的实现代码。在创建 `FileSystemClassLoader` 类的对象时，需要指定一个路径作为 Java 类字节代码所在的目录。在 `findClass` 方法的实现中，先把要加载的类名转换成对应的 `class` 文件的路径，再读取 `class` 文件以得到字节代码的内容，最后通过 `defineClass` 方法来定义 `Class` 类的对象。

代码清单 9-5 从文件系统加载字节代码的类加载器

```
public class FileSystemClassLoader extends ClassLoader {
```

```

private Path path;

public FileSystemClassLoader(Path path) {
    this.path = path;
}

protected Class<?> findClass(String name) throws ClassNotFoundException {
    try {
        byte[] classData = getClassData(name);
        return defineClass(name, classData, 0, classData.length);
    } catch (IOException e) {
        throw new ClassNotFoundException();
    }
}

private byte[] getClassData(String className) throws IOException {
    Path classFilePath = classNameToPath(className);
    return Files.readAllBytes(classFilePath);
}

private Path classNameToPath(String className) {
    return path.resolve(className.replace('.', File.separatorChar) +
        ".class");
}
}

```

FileSystemClassLoader 类只是简单地读取了字节代码的内容，实际上可以在读取字节代码之后，调用 `defineClass` 方法之前进行很多操作。例如，如果字节代码的内容是经过加密的，就需要在调用 `defineClass` 方法之前进行解密操作。另外，可以对字节代码应用第 8 章介绍的字节代码增强技术来处理。

除了通过磁盘文件或网络方式加载已有的字节代码之外，还可以在类加载器中即时生成所需的字节代码。使用第 8 章介绍的 ASM 工具可以很容易地实现这个功能。代码清单 9-6 中给出了一个动态生成字节代码的类加载器的实现。在 `GreetingClassLoader` 类的 `findClass` 方法中，使用 ASM 工具生成了类的字节代码。在生成的 Java 类的构造方法中，添加了对 `System.out` 方法的调用来输出提示信息。

代码清单 9-6 动态生成字节代码的类加载器

```

public class GreetingClassLoader extends ClassLoader implements Opcodes {
    private String message;

    public GreetingClassLoader(String message) {
        this.message = message;
    }

    protected Class<?> findClass(String name) throws ClassNotFoundException {
        byte[] classData = generateClassData(name);

```

```

        return defineClass(name, classData, 0, classData.length);
    }

    private byte[] generateClassData(String className) {
        className = className.replaceAll("\\.", "/");
        ClassWriter writer = new ClassWriter(0);
        writer.visit(V1_7, ACC_PUBLIC + ACC_SUPER, className, null, "java/lang/Object", null);
        MethodVisitor mv = writer.visitMethod(Opcodes.ACC_PUBLIC, "<init>", "()V", null, null);
        mv.visitCode();
        mv.visitVarInsn(ALOAD, 0);
        mv.visitMethodInsn(INVOKESTATIC, "java/lang/Object", "<init>", "()V");
        mv.visitFieldInsn(Opcodes.GETSTATIC, "java/lang/System", "out", "Ljava/io/PrintStream;");
        mv.visitLdcInsn(message);
        mv.visitMethodInsn(Opcodes.INVOKEVIRTUAL, "java/io/PrintStream", "println", "(Ljava/lang/String;)V");
        mv.visitInsn(RETURN);
        mv.visitMaxs(2, 1);
        mv.visitEnd();
        writer.visitEnd();
        return writer.toByteArray();
    }
}

```

前面介绍的两个示例并没有覆写 ClassLoader 类中的 loadClass 方法中的逻辑，而是覆写了 findClass 方法。在这两个示例中，ClassLoader 类默认使用的双亲优先的代理模式是有效的。如果希望改变这种默认的代理模式，那么可以覆写 loadClass 方法。在某些情况下，可能需要优先使用当前类加载器对象进行查找，再考虑使用双亲类加载器对象进行查找。代码清单 9-7 给出了这种当前类加载器对象优先的代理模式的实现。在 ParentLastClassLoader 类的 loadClass 方法中仍然先通过 findLoadedClass 方法来查找已经加载的 Java 类。这一步通常是必需的。接着调用 findClass 方法优先由当前类加载器对象进行查找。最后再代理给双亲类加载器对象来进行查找。

代码清单 9-7 当前类加载器对象优先的类加载器

```

public class ParentLastClassLoader extends ClassLoader {
    protected Class<?> loadClass(String name, boolean resolve)
        throws ClassNotFoundException {
        Class<?> clazz = findLoadedClass(name);
        if (clazz != null) {
            return clazz;
        }
        clazz = findClass(name);
        if (clazz != null) {
            return clazz;
        }
        // ...
    }
}

```

```

    }
    ClassLoader parent = getParent();
    if (parent != null) {
        return parent.loadClass(name);
    }
    else {
        return super.loadClass(name, resolve);
    }
}

```

9.4 类加载器的隔离作用

类加载器的一个重要特性是为它所加载的 Java 类创建了隔离空间，相当于添加一个新的名称空间。要理解这一点，先来说明 Java 虚拟机如何判断两个 Java 类是否相等，即两个 Class 类的对象是否相等。如果两个对象的类不同，并且两个类之间不存在父类型与子类型的关系，那么在它们之间进行赋值等操作会抛出 `java.lang.ClassCastException` 异常。Java 虚拟机需要根据两个条件进行判断：一个是 Class 类的对象表示的 Java 类的全名是否相同，另一个是 Class 类的对象的定义类加载器对象是否相同。这两个条件缺一不可。第一个条件很容易理解，如果类的名称不相同，那么它们不可能表示同一个类。第二个条件则比较难理解。在进行 Class 类的对象的相等性判断中，它们的定义类加载器是非常重要的。相同的字节代码如果由不同的类加载器对象来加载并定义，所得到的 Class 类的对象是不相等的。

下面通过一个具体的示例来说明类加载器对 Class 类的对象的相等性的影响。代码清单 9-8 给出了一个简单的 Java 类 `Sample`。`Sample` 类中包含了一个 `setSample` 方法。在这个方法中，把作为参数传入的 `Object` 类的对象强制类型转换成 `Sample` 类的对象。如果类型转换时出现 `ClassCastException` 异常，就说明参数对象的 Java 类与当前的 `Sample` 类不相等。

代码清单 9-8 用来说明 Class 类的对象相等性判断方式的示例 Java 类

```

public class Sample {
    private Sample obj;

    public void setSample(Object obj) {
        this.obj = obj;
    }
}

```

对 `Sample` 类进行编译之后，将得到的 `class` 文件放在某个目录中，用代码清单 9-5 中的 `FileSystemClassLoader` 类的对象进行加载，如代码清单 9-9 所示。测试方式是使用

两个不同的 FileSystemClassLoader 类的对象来分别加载 Sample 类的字节代码，并定义出对应的 Java 类。然后从 Java 类中创建出新的对象，用一个对象作为参数调用另外一个对象上的 setSample 方法。实际运行的结果是抛出 ClassCastException 异常。虽然是从同样的字节代码中创建出来的同名 Java 类，但是由于定义它们的类加载器对象不同，这两个 Class 类的对象仍然是不相等的。

代码清单 9-9 Class 类的对象的相等性测试

```
public class ClassIdentity {
    public void test() throws Exception {
        Path path = Paths.get("classData");
        FileSystemClassLoader fscl1 = new FileSystemClassLoader(path);
        FileSystemClassLoader fscl2 = new FileSystemClassLoader(path);
        String className = "com.java7book.chapter9.Sample";
        Class<?> class1 = fscl1.loadClass(className);
        Object obj1 = class1.newInstance();
        Class<?> class2 = fscl2.loadClass(className);
        Object obj2 = class2.newInstance();
        Method setSampleMethod = class1.getMethod("setSample", java.lang.Object.
            class);
        setSampleMethod.invoke(obj1, obj2); // 抛出 ClassCastException 异常
    }
}
```

在很多场合都可以利用类加载器的这个特性为虚拟机中同名的 Java 类创建一个隔离的名称空间，使同名的 Java 类可以在虚拟机中共存。同名 Java 类需要共存的一个典型场景是程序的版本更新。一个程序可能存在多个不同的版本。用户既希望使用新版本的程序，又希望基于旧版本的代码可以继续运行。这就要求两个版本的 Java 类在虚拟机中同时存在。如果不使用自定义类加载器来划分名称空间，就只能让新旧版本的 Java 类使用不同的名称，比如在正常的类名后添加类似“V1”和“V2”等后缀进行标识。这种做法使用起来很不方便。使用自定义的类加载器就可以仍然使用相同名称的 Java 类，在实现时使用不同的类加载器对象来进行加载不同版本的 Java 类。

在一般的程序版本更新中，会保持接口不变，只修改接口的后台实现。如果接口发生变化，那么客户端代码要做出比较大的修改。这里以接口不变的版本更新为例进行说明。代码清单 9-10 中给出了一个简单的接口 Versionized。

代码清单 9-10 用来说明版本更新方式的接口示例

```
public interface Versionized {
    String getVersion();
}
```

该接口的实现可能随着版本更新而发生变化。客户端代码通过一个工厂方法来获取该接口的具体实现。代码清单 9-11 给出了这个工厂方法的实现。在工厂方法中不是简单

地创建出所需的具体实现的对象，而是创建一个新的类加载器对象。由类加载器对象先加载 Java 类，再创建出相应的接口实现对象。在具体的实现中，不同版本的 Java 类的字节代码存放在不同的路径中。使用代码清单 9-5 中的 FileSystemClassLoader 类的对象来加载所需版本的 Java 类。

代码清单 9-11 获取接口实现对象的工厂方法

```
public class ServiceFactory {
    public static Versionized getService(String className, String version) throws
        Exception {
        Path path = Paths.get("service", version);
        FileSystemClassLoader loader = new FileSystemClassLoader(path);
        Class<?> clazz = loader.loadClass(className);
        return (Versionized) clazz.newInstance();
    }
}
```

代码清单 9-12 中给出了代码清单 9-11 中 getService 方法的使用方式。Versionized 接口的不同版本的实现使用相同的 Java 类名。由于类加载器带来的隔离作用，这些同名的 Java 类可以共存在虚拟机中。用户可以根据所需的版本号找到对应的 Java 类。

代码清单 9-12 不同版本的接口实现对象的使用示例

```
public class ServiceConsumer {
    public void consume() throws Exception {
        String serviceName = "com.java7book.chapter9.SampleService";
        Versionized v1 = ServiceFactory.getService(serviceName, "v1");
        Versionized v2 = ServiceFactory.getService(serviceName, "v2");
    }
}
```

9.5 线程上下文类加载器

线程上下文类加载器 (context class loader) 是在 J2SE 1.2 中引入的概念。Java 中表示线程的 `java.lang.Thread` 类中有两个方法用来获取和设置当前线程的上下文类加载器。这两个方法分别是 `getContextClassLoader` 和 `setContextClassLoader`，其中 `setContextClassLoader` 方法接受一个 `ClassLoader` 类的对象作为参数。当前线程中运行的代码可以使用该线程的上下文类加载器来加载 Java 类和资源文件。如果一个线程在创建之后没有显式地设置其上下文类加载器的值，则使用其父线程的上下文类加载器对象作为自身的上下文类加载器对象。程序启动时的第一个线程的上下文类加载器默认是 Java 平台的系统类加载器对象。因此，在默认情况下，通过当前线程的 `getContextClassLoader` 方法获取的类加载器对象和使用当前类的 `getClassLoader` 方法得到的类加载器对象是相同的，二者均为系统类加载器。

线程上下文类加载器提供了一种直接的方式在程序的各部分之间共享 ClassLoader 类的对象。当在程序中需要使用类加载器来加载类或资源时，有几种做法可以获取需要使用的 ClassLoader 类的对象。第一种做法是创建新的 ClassLoader 类的对象，这种做法使用的场合比较少。第二种做法是使用加载当前 Java 类的 ClassLoader 类的对象。第三种做法是使用 Java 平台提供的系统类加载器或扩展类加载器。这三种做法都无法简单地满足某些场景的需要。比如，存在两个互相关联的 Java 类 A 和 B，这两个类必须由同一个类加载器对象来加载，否则会出现内部错误。满足这个需求的做法是用加载类 A 的 ClassLoader 类的对象去加载类 B。如果使用前面提到的做法，就需要提供额外的方式在加载类 A 和类 B 的代码之间传递 ClassLoader 类的对象。这会带来附加的复杂性。只要加载类 A 和类 B 的代码在同一个线程中运行，使用线程上下文类加载器是最简单的做法。在加载类 A 时，获取当前使用的 ClassLoader 类的对象，调用当前线程对象的 setContextClassLoader 方法把线程上下文类加载器设置为该 ClassLoader 类的对象。在加载类 B 的时候，通过当前线程对象的 getContextClassLoader 方法来得到之前保存的 ClassLoader 类的对象，再进行加载即可。

线程上下文类加载器的重要作用是解决 Java 平台的服务提供者接口 (service provider interface, SPI) 带来的类加载相关的问题。Java 平台上的很多规范都是以 SPI 的形式出现的，比如数据库访问规范 JDBC 和 XML 处理规范 JAXP 等。这些 SPI 的特点是 Java 平台只提供接口和部分辅助 Java 类，接口的具体实现由规范的实现者提供。以 JDBC 规范为例来说，相关的接口声明在 java.sql 和 javax.sql 包中，例如 java.sql.Connection 接口表示一个数据库连接。而 Connection 接口的具体实现类由数据库驱动来提供。SQL Server、MySQL 和 Apache Derby 等数据库都有自己对应的 JDBC 接口的实现类。SPI 接口通常会提供一些工厂方法来创建接口的具体实现对象。在第 2 章介绍 Java 的脚本语言支持 API 时提到的 javax.script.ScriptEngineManager 类就是一个典型的例子。ScriptEngineManager 类用来管理当前程序中可用的脚本执行引擎。脚本语言开发者可以实现 javax.script.ScriptEngine 接口，使开发人员可以通过脚本语言支持 API 来使用这种脚本语言。当开发人员下载了该脚本语言对应的 ScriptEngine 接口的实现，并将其添加到程序的类路径 (CLASSPATH) 中之后，新的脚本执行引擎对 ScriptEngineManager 类来说必须是可见的，从而允许开发人员通过工厂方法得到对应的 ScriptEngine 接口的实现对象。脚本引擎在注册时只提供了类名。ScriptEngineManager 类的对象为了能够提供 ScriptEngine 接口的具体实现对象，需要加载对应的 Java 类并创建出新的对象。

这里存在的问题是使用代理模式无法完成 SPI 实现类的加载。SPI 接口相关的类本身作为 Java 标准库的一部分，是由启动类加载器来加载的。也就是说，加载 ScriptEngineManager 类的是启动类加载器。在 ScriptEngineManager 类的实现中需要查找程序的 CLASSPATH 来找到 SPI 的实现类，并创建出相应的对象。程序的 CLASSPATH 中的类一般是由系统类加载器负责加载的，启动类加载器无法完成相关的

加载工作。而启动类加载器又无法把加载的工作代理给系统类加载器来完成，因为启动类加载器是系统类加载器的祖先。在理论上，可以在启动类加载器内部保存一个系统类加载器对象的引用，在加载 SPI 实现类时代理给系统类加载器来完成。但是在某些情况下，SPI 实现类可能并不出现在 CLASSPATH 中，而是需要由自定义的类加载器对象来完成加载。启动类加载器是无法处理这种情况的。

为了解决这个问题，ScriptEngineManager 类的对象在加载 SPI 实现类时，使用的是线程上下文类加载器。在默认情况下，程序运行时的线程上下文类加载器是系统类加载器，这样可以加载 CLASSPATH 中出现的 SPI 实现类。如果需要使用自定义类加载器来加载 SPI 实现类，可以把当前线程的上下文类加载器设置成能够加载到 SPI 实现类的类加载器对象。如果在程序运行中改变了线程上下文类加载器的值，可能会造成 SPI 的实现类无法加载。

9.6 Class.forName 方法

熟悉 Java EE 开发的开发人员对于 Class.forName 方法应该并不陌生。在 Java EE 开发中，Class.forName 方法的一个典型应用是加载使用 JDBC 操作数据库时的数据库驱动。比如，当需要加载 Apache Derby 数据库的嵌入式驱动时，可以使用代码 Class.forName("org.apache.derby.jdbc.EmbeddedDriver")。不过这种做法从 JDBC 4.0 开始就不再需要了，因为 java.sql.DriverManager 类支持了使用服务发现机制来自动查找可用的数据库驱动。

Class.forName 方法的作用是根据 Java 类的名称得到对应的 Class 类的对象。该方法有两种重载形式。第一种是使用 3 个参数的复杂形式。3 个参数依次表示 Java 类的名称、是否初始化 Java 类，以及用于加载 Java 类的类加载器对象。如果第 3 个参数的值为 null，则使用启动类加载器来进行加载。第二种是只使用一个参数的简单形式，相当于第一种重载形式中的第 2 个和第 3 个参数的值分别是 true 和 this.getClass().getClassLoader()。

Class.forName 方法与 ClassLoader 类的重要区别在于 Class.forName 方法可以初始化 Java 类，而 ClassLoader 类的对象是不行的。这也是 Class.forName 方法的优势所在。初始化 Java 类意味着 Java 类中的静态变量会被初始化，同时静态代码块也会被执行。代码清单 9-13 给出了一个简单的包含静态代码块的 Java 类。在该类被初始化时，会在控制台输出提示信息。

代码清单 9-13 包含静态代码块的示例 Java 类

```
public class ClassForNameTest {
    static {
        System.out.println(" 初始化 ");
    }
}
```

代码清单 9-14 展示了分别使用 Class.forName 方法和 ClassLoader 类来加载代码清单 9-13 中的 ClassForNameTest 类时的不同之处。在调用 Class.forName 方法时会输出与初始化类相关的提示信息，而调用 ClassLoader 类的 loadClass 方法则不会。

代码清单 9-14 Class.forName 方法与 ClassLoader 类在加载类时的不同之处

```
public void classForNameVsLoader() throws ClassNotFoundException {
    String className = "com.java7book.chapter9.ClassForNameTest";
    Class<?> clazz1 = Class.forName(className);
    ClassLoader loader = this.getClass().getClassLoader();
    Class<?> clazz2 = loader.loadClass(className);
}
```

这也是为什么在 JDBC 4.0 之前需要使用 Class.forName 方法来加载数据库驱动的 Java 类的原因。在不支持驱动的自动发现之前，在数据库驱动类的静态代码块中可以添加必要的驱动注册和初始化的逻辑，在驱动的 Java 类被初始化时，完成相关的处理。

9.7 加载资源

类加载器除了可以加载 Java 类之外，还可以加载与 Java 类相关的资源文件，如文本文件、图片文件和音频文件等。Java 程序在运行过程中需要访问这些资源文件的内容。每个资源文件通过名称来标识。资源名称可以由多个部分组成，每个部分之间用“/”分隔，如一个图片文件的资源名称可以是“resources/images/logo.gif”。资源名称的表示形式是平台无关的，由具体的实现负责把平台无关的抽象资源名称映射到实际的资源保存方式上。如果资源是保存在文件系统上的，通常是映射到资源文件的路径，而资源名称中每个用“/”分隔的部分代表一个目录。这些资源文件通常与 class 文件保存在同一个目录下，或者同一个 jar 包中。

使用类加载器来加载资源的好处是可以解决资源文件存放时路径不固定的问题。比如，一个 Java 类在运行时需要读取保存配置信息的属性文件的内容，如果使用 Java 的文件操作 API 来读取，就要知道属性文件的绝对路径。Java 类的 class 文件与属性文件的相对位置虽然固定，但是它们所在的绝对路径是不确定的。通过文件操作 API 无法保证总是能正确地找到文件，通过类加载器提供的加载资源的方法则可以正确地加载到相关的资源。

ClassLoader 类中负责加载资源的方法是 getResource 和 getResourceAsStream。根据资源名称，前者得到表示资源路径的 java.net.URL 类的对象，后者得到用来读取资源内容的 java.io.InputStream 类的对象。从实现上来说，getResourceAsStream 方法在内部先调用 getResource 方法得到 URL 类的对象，再调用该对象的 openStream 方法获取 InputStream 类的对象。一般对资源进行读取操作时，getResourceAsStream 方法的使用频率较高。在进行查找时，getResource 方法会先检查当前类加载器的双亲类加载器是

否为 null。如果不为 null，则调用双亲类加载器的 `getResource` 方法来进行查找；如果为 null，则通过启动类加载器来查找。如果找不到对应的资源，则调用 `ClassLoader` 类的 `findResource` 方法进行查找。这种实现方式类似于 `ClassLoader` 类在加载 Java 类时默认的双亲优先的代理模式。在 `ClassLoader` 类中声明为 `protected` 的 `findResource` 方法的作用类似于 `findClass` 方法。如果类加载器有自定义的资源查找机制，那么需要覆写此方法。

除了 `getResource` 方法之外，`ClassLoader` 类中还有一个相关的 `getResources` 方法。这个方法会根据资源名称返回所有具有该名称的资源文件。该方法的返回值是一个可以遍历所有查找结果的 `java.util.Enumeration` 类的对象。在实现上，`getResources` 方法的机制与 `getResource` 方法是一样的，都是先通过双亲类加载器进行查找，只不过 `getResources` 方法会完成整个查找过程来搜索所有满足条件的资源，而不是像 `getResource` 方法一样查找到第一个满足条件的结果就返回。`ClassLoader` 类中也有与 `findResource` 方法作用相似的 `findResources` 方法，用来与 `getResources` 方法配合使用。在某些情况下，可能会需要查找在不同位置上的所有同名资源，此时可以使用 `getResources` 方法。

当需要使用系统类加载器来加载资源时，可以直接使用 `ClassLoader` 类中的静态方法 `getSystemResource`、`getSystemResourceAsStream` 和 `getSystemResources`。这 3 个方法在实现上是先得到系统类加载器，再调用系统类加载器的对应方法来进行加载。如果当前系统类加载器为 null，则通过启动类加载器来进行加载。代码清单 9-15 给出了使用类加载器来加载属性文件的示例。

代码清单 9-15 使用类加载器来加载属性文件的示例

```
public class LoadResource {
    public Properties loadConfig() throws IOException {
        ClassLoader loader = this.getClass().getClassLoader();
        InputStream input = loader.getResourceAsStream("com/java7book/chapter9/
            config.properties");
        if (input == null) {
            throw new IOException("找不到配置文件。");
        }
        Properties props = new Properties();
        props.load(input);
        return props;
    }
}
```

在使用类加载器的 `getResource` 和 `getResourceAsStream` 方法时，需要注意的是使用正确的资源名称。代码清单 9-15 给出了资源名称的通常表示形式，即根据资源文件所在的 Java 包名来确定。配置文件 `config.properties` 在 `com.java7book.chapter9` 包中，把包名中的“.”变成“/”之后再加上文件的文件名就得到了资源名称。资源名称需要根据资

源文件所在的包名进行调整。

除了使用 ClassLoader 类中的方法来加载资源之外，Class 类中也有相关的方法来加载资源。Class 类中的方法与 ClassLoader 类中的方法在名称上是相同的，分别是 getResource 和 getResourceAsStream。Class 类中的这两个方法的实现是这样的，先通过 getClassLoader 方法得到加载当前类的 ClassLoader 类的对象，再通过 ClassLoader 类的对象的对应方法来进行加载。如果 getClassLoader 方法的返回值为 null，则使用 ClassLoader 类中的 getSystemResource 和 getSystemResourceAsStream 方法来进行加载，相当于使用系统类加载器来进行加载。不过 Class 类中的方法在调用 ClassLoader 类的对应方法之前，会进行资源名称的转换。如果资源名称以“/”开头，则会去掉开头的“/”；否则自动在资源名称前加上 Class 类的对象所在的包的名称。对于代码清单 9-15 的示例，如果使用代码“this.getClass().getResourceAsStream”来进行加载，那么可以直接使用资源名称“config.properties”，而不需要加上前面的包名，包名的添加工作由 Class 类来完成。使用 Class 类中的加载资源的方法比使用 ClassLoader 类中的相关方法要实用一些。在重构的过程中，可能对包含资源文件的包名进行了修改。如果使用 ClassLoader 类中的方法，则需要手动修改加载时使用的资源名称，而使用 Class 类中的方法则不需要进行修改。

9.8 Web 应用中的类加载器

类加载器在基于 Java EE 技术实现的 Web 容器中有着非常重要的作用。在一个 Web 容器中通常运行着很多个 Web 应用。这些应用之间是互相隔离的，互不影响，但又都依赖于 Web 容器提供的功能，运行在同一个 Java 虚拟机之上。这种受管理的隔离方式是通过类加载器来实现的。典型的做法是每个 Web 应用使用自己的类加载器对象来加载应用中包含的 Java 类和资源。不同的 Web 应用中相同名称的 Java 类可以共存于虚拟机中。比较典型的场景是对 Web 应用中使用的第三方库的处理。不同 Web 应用在开发中可能使用同样的第三方库，但是使用的库的版本可能不同。如果没有进行隔离，那么所有 Web 应用都会引用某一个版本的库中的 Java 类。某些应用可能会因为使用了错误版本的库而出现错误。

在 Java EE 中的 Servlet 规范中给出了 Web 应用的类加载器的实现方式的推荐做法，即对默认的双亲优先的代理模式进行修改，改为使用当前类加载器优先的方式。这种做法的出发点是解决 Web 应用中的第三方库和容器本身使用的第三方库的冲突问题。如果采用双亲优先的方式，那么容器中提供的第三方库的 Java 类会被优先加载。Web 应用可能使用了同样的库，但是版本与容器提供的并不兼容，这会造成 Web 应用出现错误。通过提高 Web 应用本身的 Java 类和第三方库的优先级，可以避免这个问题。一般的 Web 容器都遵循 Servlet 规范中的推荐做法。有些应用服务器允许 Web 应用在双亲优先和当前类加载器优先这两种策略中进行选择。

下面通过具体的 Apache Tomcat 7.0 示例分析来说明 Web 容器中类加载器是如何工作的。Tomcat 是一个使用很广泛的 Web 容器，也是开放源代码的。Tomcat 中的 Web 应用对应的类加载器是 org.apache.catalina.loader.WebappClassLoader 类的对象。WebappClassLoader 类继承自标准库中的 java.net.URLClassLoader 类。在类加载器中，最重要的是 loadClass 方法的实现。WebappClassLoader 类中 loadClass 方法的实现按照下面几个步骤尝试加载 Java 类。

1) 调用 findLoadedClass 来查找该 Java 类是否已经被加载过。一般的类加载器都会进行这样的检查，可以避免不必要的查找过程。

2) 调用系统类加载器的 loadClass 方法来尝试加载类。这是为了避免 Web 应用覆盖 Java 标准库中的类。

3) WebappClassLoader 类不一定会把加载类的请求代理给双亲类加载器。在两种情况下会进行代理。第一种情况是使用 setDelegate 方法把代理模式打开之后。在默认情况下代理模式是关闭的。第二种情况是要加载的 Java 类的名称满足一定的条件，比如名称以“javax.servlet”开头的 Servlet API 相关的 Java 类是由双亲类加载器来加载的。

4) 调用 findClass 方法来查找 Web 应用本身的 Java 类。

5) 如果在第 3) 步中没有把加载类的请求代理给双亲类加载器，则在这一步中进行。从第 4) 和第 5) 步的顺序可以看出，WebappClassLoader 类使用的是当前类加载器优先的策略。

对于 Web 应用本身的 Java 类，Tomcat 是按照划分成不同仓库的方式来进行管理的。每个仓库表示一个 Java 类的存放位置。仓库分成外部仓库和内部仓库两种。每个外部仓库对应一个 URL 类的对象，表示一个加载类时的查找路径。由于 WebappClassLoader 类继承自 URLClassLoader 类，通过 URLClassLoader 类中的 addURL 方法可以添加新的外部仓库。内部仓库则指的是 Web 应用本身的 WEB-INF 目录下的 classes 和 lib 目录。这两个标准目录分别用来存放 Web 应用的 class 文件和使用的第三方库的 jar 包。WebappClassLoader 类的对象可以配置在查找时是否优先外部仓库。在 WebappClassLoader 类的 findClass 方法中的查找过程如下：

1) 如果存在外部仓库并且配置了优先查找外部仓库，则先调用双亲类加载器对象的 findClass 方法进行查找。

2) 在内部仓库中进行查找。

3) 如果存在外部仓库并且配置了不优先查找外部仓库，则说明第 1) 步没有执行。此时调用双亲类加载器对象的 findClass 方法来进行查找。

在 Web 应用开发中，每个 Web 应用自己的 Java 类文件和使用的库的 jar 包，分别放在 WEB-INF 目录下的 classes 和 lib 目录下。多个应用共享的 Java 类文件和 jar 包，则放在 Web 容器指定的由所有 Web 应用共享的目录下面。通过这种标准的方式，可以避免一些常见的类加载相关的错误。

9.9 OSGi 中的类加载器

OSGi 是 Java 平台上的动态模块系统，它为开发人员提供了面向服务和基于组件的运行环境，并提供标准的方式来管理软件的生命周期。OSGi 已经被部署在很多产品上，在开源社区也得到了广泛的支持。Eclipse 平台是基于 OSGi 技术来构建的。在 OSGi 技术的实现中，类加载器扮演了非常重要的作用。

9.9.1 OSGi 基本的类加载器机制

OSGi 中最基本的组成部分是模块（bundle）。每个模块作为一个独立的组件，完成特定的功能，并对外提供服务。每个模块由 Java 类和所需的资源文件组成，以 jar 包的形式出现。每个模块既可以是服务的提供者，又可以是服务的消费者。从服务提供者的角度来说，一个模块中的 Java 类可以被其他模块使用；从服务消费者的角度来说，一个模块为了完成其功能，可能需要使用其他模块提供的 Java 类。对于一个模块中包含的 Java 类来说，一部分 Java 类作为对外提供的服务，对其他模块是可见的；而另外一部分 Java 类则作为模块的内部实现，对其他模块是不可见的。每个模块的 Java 类相当于存在于一个受管理的隔离空间中。对这个隔离空间的管理由 OSGi 实现中的类加载器来完成。

下面通过 3 个相互依赖的模块来说明 OSGi 中类加载器机制的作用。这 3 个模块的作用是实现并使用一个简单的计算器程序。其中 calculator.common 模块中包含实现通用功能的 Java 类，calculator.impl 模块中包含计算器程序的具体实现类，calculator.user 模块中包含使用计算器程序的 Java 类。从依赖关系上说，calculator.impl 模块依赖于 calculator.common，calculator.user 模块依赖于 calculator.impl。OSGi 中每个模块使用清单文件声明自己所依赖的需要导入的来自其他模块的 Java 包的名称，也可以声明自己提供的可供其他模块使用的 Java 包的名称。代码清单 9-16 给出了 calculator.impl 模块的清单文件中与依赖关系相关的部分。

代码清单 9-16 OSGi 模块的清单文件中与依赖关系相关的部分

```
Export-Package: com.java7book.calculator.impl
Import-Package: com.java7book.calculator.common
```

在清单文件中通过 Import-Package 属性声明了 calculator.impl 模块需要使用 com.java7book.calculator.common 包，通过 Export-Package 属性声明了 calculator.impl 模块内部的 com.java7book.calculator.impl 包对其他模块是可见的。在声明了依赖关系之后，calculator.impl 模块中的代码可以使用 com.java7book.calculator.common 包中的 Java 类。

9.9.2 Equinox 框架的类加载实现机制

目前存在不少 OSGi 规范的实现，这里使用 Eclipse Equinox 作为介绍时的 OSGi 实现，运行环境在 Eclipse IDE 中。每个模块在运行时都会有一个对应的类加载器对象。由

这个类加载器对象负责加载模块本身包含的 Java 类和资源。Eclipse Equinox 在启动模块时使用 `org.eclipse.osgi.internal.loader.BundleLoader` 类的对象来加载一个模块。每个模块都有自己对应的 `BundleLoader` 类的对象。`BundleLoader` 类的对象中都封装了一个类加载器对象来加载模块中的 Java 类。这个类加载器对象需要实现 `org.eclipse.osgi.framework.adaptor.BundleClassLoader` 接口，并且继承自 `ClassLoader` 类。默认的类加载器实现类是 `org.eclipse.osgi.internal.baseadaptor.DefaultClassLoader`。`DefaultClassLoader` 类的对象在创建时需要提供一个 `org.eclipse.osgi.framework.adaptor.ClassLoaderDelegate` 接口的实现对象。`ClassLoaderDelegate` 接口表示的是实际完成类加载工作的代理对象，其中声明了用来加载 Java 类的 `findClass` 方法。`DefaultClassLoader` 类在其 `loadClass` 方法的实现中，只是简单地把加载请求代理给其中包含的 `ClassLoaderDelegate` 接口的实现对象的 `findClass` 方法。`DefaultClassLoader` 类的对象本身并不会尝试去加载类，也不像其他类加载器一样会把加载请求代理给双亲类加载器对象。这是另外一种形式的代理模式。`BundleLoader` 类本身实现了 `ClassLoaderDelegate` 接口，所以 `BundleLoader` 类既负责创建加载 Java 类时使用的 `BundleClassLoader` 接口的实现，又负责完成具体的类加载任务。当模块中的代码需要加载 Java 类时，可以通过代码“`this.getClass().getClassLoader()`”来得到加载当前类的类加载器对象。

`BundleLoader` 类中负责加载 Java 类的 `findClass` 方法中封装了比较复杂的 Java 类的查找机制。对于以“`java.`”开头的 Java 类，会直接代理给双亲类加载器对象来完成。对于其他的 Java 类，则在模块内部和所导入的包中进行查找。基本的类查找步骤如下：

1) 检查要加载的 Java 类是否出现在被配置为由双亲类加载器对象负责加载的包名列表中。如果是，则直接代理给双亲类加载器对象来完成。通过 OSGi 框架的属性“`org.osgi.framework.bootdelegation`”可以配置这个列表中包含的包名。这个步骤的作用是允许某些 Java 类被双亲类加载器对象来加载。

2) 搜索该模块中通过 `Import-Package` 属性声明导入的 Java 包的列表。如果找到，由提供该 Java 包的模块对应的 `BundleLoader` 类的对象负责该 Java 类的加载。

3) 搜索该模块中通过 `Require-Bundle` 属性声明所依赖的其他模块的列表。如果找到可以提供该 Java 类的模块，则由该模块对应的 `BundleLoader` 类的对象来负责加载该 Java 类。

4) 在模块内部包含的 Java 类中进行查找。

5) 搜索模块中通过 `DynamicImport-Package` 属性声明动态导入的 Java 包列表。如果找到，则由提供该 Java 包的模块对应的 `BundleLoader` 类的对象负责加载该 Java 类。

6) 如果仍然找不到 Java 类，在某些情况下代理给双亲类加载器对象进行加载。

从上面的类加载流程可以看出，`BundleLoader` 类在尝试加载 Java 类时主要依赖两个外部来源来代理类加载的请求，一个是 `BundleClassLoader` 接口实现对象的双亲类加载器，另外一个是其他模块对应的 `BundleLoader` 类的对象。`BundleClassLoader` 接口的实现类可以选择不同的双亲类加载器。可以通过 OSGi 框架的属性配置使用不

同的双亲类加载器选择方式。默认的双亲类加载器是 Java 平台的启动类加载器。还可以选择使用加载 OSGi 框架本身的类加载器对象、系统类加载器或扩展类加载器作为 BundleClassLoader 接口的实现对象的双亲类加载器。模块 calculator.impl 导入了 calculator.common 导出的 com.java7book.calculator.common 包。当在 calculator.impl 模块中使用 com.java7book.calculator.common 包中的 Java 类时，实际的类加载工作是由 calculator.common 模块对应的 BundleLoader 类的对象来完成的。通过这种代理模式，在一个模块中只有通过 Export-Package 属性声明的 Java 包才对其他模块可见。如果一个模块 A 中的某个 Java 包没有包含在 Export-Package 属性声明的列表中，而另外一个模块 B 又试图引用该 Java 包中的类，模块 B 会找不到对应的类。因为该 Java 类没有出现在模块 A 的 Export-Package 属性声明的 Java 包列表中，不会考虑代理给模块 A 来进行查找，从而无法找到该 Java 类。

在 OSGi 运行环境中，如果在模块中使用 SPI 的实现类，可能会出现问题，因为 SPI 的代码中通常使用线程上下文类加载器来加载 SPI 接口的实现类。在一般的运行环境中，SPI 的实现类是可以从程序运行时的 CLASSPATH 中找到的。但是在 OSGi 运行环境中，SPI 的实现类一般作为模块所依赖的库出现在模块本身的 CLASSPATH 中。如果在程序的其他部分对当前线程的上下文类加载器进行了修改，有可能造成 SPI 的实现类无法被加载。这个时候可以使用当前线程的 setContextClassLoader 方法把线程上下文类加载器设置成加载模块 Java 类的类加载器对象，从而保证可以加载模块的 CLASSPATH 中的 SPI 实现的 Java 类。代码清单 9-17 给出了通过修改线程上下文类加载器来加载 SPI 实现类的一般做法。在完成加载之后，要把线程上下文类加载器的值恢复为之前的值，以免对程序的其他部分造成影响。

代码清单 9-17 通过修改线程上下文类加载器来加载 SPI 实现类的一般做法

```
ClassLoader oldContextLoader = Thread.currentThread().getContextClassLoader();
Thread.currentThread().setContextClassLoader(this.getClass().getClassLoader());
// 加载 SPI 实现类
Thread.currentThread().setContextClassLoader(oldContextLoader);
```

在 OSGi 中使用这种类加载器的实现方式，可以对一个模块中的 Java 类按照所在的包设置其可见性，从而带来了访问控制上的灵活性。不过这种方式也可能在实际开发中带来一些麻烦，尤其在使用第三方库或嵌入到其他容器中时。某些第三方库或容器可能同样使用了复杂的类加载器实现，会与 OSGi 框架本身的类加载器实现交织在一起，造成难以解决的问题。在 OSGi 模块中使用第三方库时，可以考虑下面的建议：

- 1) 如果一个库只被一个模块使用，把该库的 jar 包放在模块中，并在模块清单文件中使用 Bundle-ClassPath 属性进行声明。
- 2) 如果一个库被多个模块共用，则可以为这个库创建一个新的模块。对于库中的 Java 包，如果其他模块会用到，那么在清单文件中通过 Export-Package 属性进行声明。

其他模块只需要通过 Import-Package 属性声明所需要的包即可。

3) 如果出现了找不到 Java 类的情况，则检查当前线程的上下文类加载器是否正确。一般可以通过把当前线程的上下文类加载器设置为模块的类加载器对象来解决这个问题。

9.9.3 Equinox 框架嵌入到 Web 容器中

Eclipse Equinox 框架支持被嵌入到 Web 容器中来使用。可以在 servlet 容器中启动一个 Equinox 框架，用来处理 servlet 请求。通过这种方式，既可以把 OSGi 技术应用到复杂的 Web 应用开发中，又可以复用已有的 servlet 容器的相关资产。这种实现方式是通过一个特殊的 servlet 来处理请求，并转发给 Equinox 框架来处理。当在 servlet 容器中嵌入 Equinox 框架时，会出现一些与类加载器相关的问题。这是因为 servlet 容器和 OSGi 框架内部都采用了比较复杂的类加载器实现，当它们两个在一起共同使用时，会出现无法加载 Java 类的情况。

以之前提到的计算器程序中的 OSGi 模块为例，把这些模块嵌入到 servlet 容器中来运行。在计算器程序的内部实现中，使用脚本语言支持 API 来进行运算表达式的求值。使用的脚本语言不是 Java 平台默认支持的 JavaScript，而是 Ruby。为了能够使用 Ruby 语言，在下载了 JRuby 的库之后，将其放在合适的位置以被脚本语言支持 API 所识别。脚本语言支持 API 使用线程上下文类加载器来查找不同脚本引擎的实现类。因此 JRuby 库的 jar 包需要放在可以被线程上下文类加载器查找到的位置。在一般情况下，只需要把 jar 包放在 OSGi 模块的某个目录下，并在清单文件中通过 Bundle-ClassPath 属性声明即可。不过在这个示例中，除了 OSGi 框架之外，Web 应用的其他部分也需要使用 JRuby 库，这就要求在 Web 应用的 WEB-INF 下的 lib 目录中也要有 JRuby 的 jar 包。在同一个 Web 应用中包含两个相同的 jar 包会在后期的维护中带来麻烦。更好的做法是把 JRuby 的 jar 包放在 Web 应用的 WEB-INF 下的 lib 目录中，在 OSGi 模块中引用这个 jar 包，这要求在 OSGi 模块运行时的线程上下文类加载器能够加载到 servlet 容器中 lib 目录中的 jar 包。可以在脚本语言支持 API 的 ScriptEngineManager 类的对象查找到可用的脚本引擎之前，把线程上下文类加载器设置成加载 Web 应用的类加载器对象，这样就可以查找到 WEB-INF 下的 lib 目录中的 jar 包。

首先需要获取 Web 应用的当前类加载器。Web 应用的类加载器对象用来加载 servlet 实现类，只需要通过 servlet 实现类的 getClassLoader 方法就可以获取。Equinox 框架使用 org.eclipse.equinox.servletbridge.BridgeServlet 类的对象来接受 HTTP 请求并转发给 OSGi 框架中的 servlet 来处理。为了能够在 OSGi 模块中使用 Web 应用的类加载器对象，需要继承 BridgeServlet 类并提供保存 ClassLoader 类的对象的功能。代码清单 9-18 给出了对应实现的代码。在 CustomBridgeServlet 类的 service 方法实现中，在处理 servlet 请求之前，把 Web 应用的类加载器对象保存在表示 HTTP 请求的 HttpServletRequest 类的

对象中，以便可以在后续的代码实现中使用。

代码清单 9-18 自定义的 BridgeServlet 类的子类

```
public class CustomBridgeServlet extends BridgeServlet {
    protected void service(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        req.setAttribute("WebappClassLoader", this.getClass().getClassLoader());
        super.service(req, resp);
    }
}
```

代码清单 9-19 给出了处理表达式计算请求的实际 servlet 类的代码。在实现中，先从 HttpServletRequest 类的对象中得到之前保存的 Web 应用的类加载器对象，接着把线程的上下文类加载器设置为该类加载器对象。经过这样的设置之后，ScriptEngineManager 类的对象就可以正确地查找到 Ruby 语言的脚本执行引擎的实现类。在完成查找之后，需要把当前线程的上下文类加载器恢复为之前的值，以确保不影响后面代码的使用。

代码清单 9-19 处理表达式计算请求的 servlet 的代码

```
public class CalculatorServlet extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        Thread currentThread = Thread.currentThread();
        ClassLoader oldContextLoader = currentThread.getContextClassLoader();
        ClassLoader webappLoader = (ClassLoader) req.getAttribute("WebappClassLo
            ader");
        currentThread.setContextClassLoader(webappLoader);
        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = manager.getEngineByExtension("rb");
        currentThread.setContextClassLoader(oldContextLoader);
        if (engine == null) {
            resp.sendError(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
            return;
        }
        String expr = req.getParameter("expr");
        try {
            Object result = engine.eval(expr);
            resp.getWriter().write(result.toString());
        } catch (ScriptException e) {
            throw new ServletException(e);
        }
    }
}
```

9.10 小结

类加载器是 Java 语言中的一个重要概念，为 Java 带来了很多灵活性。在对 Java 字节代码的处理中，类加载器是重要的一环。通过自定义类加载器可以实现 Java 字节代码的增强、加密和生成等功能。通过类加载器可以在虚拟机中划分同名 Java 类的隔离空间，这一特性在很多框架中都得到了实际的应用。线程上下文类加载器可以把类加载器和当前线程结合起来，一般用来加载 SPI 接口的实现类。本章对 Web 容器和 OSGi 框架中类加载器的使用方式做了说明，使读者可以了解相关的知识。

第 10 章 对象生命周期

Java 语言是一门面向对象的编程语言。除了基本类型之外，在运行的 Java 程序中出现的都是对象。从 J2SE 5.0 开始，通过基本类型的自动装箱 / 拆箱机制，基本类型也可以通过自动的方式与程序中的其他对象进行交互。Java 中的所有类都继承自 `java.lang.Object` 类，所有的对象都是 `Object` 类的实例。给定一个 Java 类，如果满足其构造方法的访问控制要求，那么使用 `new` 操作符可以创建出该类的对象。对象创建完成后，可以与其他对象进行交互来完成程序要求的功能。当不再使用对象时，可以由 Java 平台的垃圾回收器来回收对象所占用的内存空间。一般对象都会经历从创建到使用再到销毁的过程。

一个对象的完整生命周期涉及 Java 平台的很多相关技术。在创建一个 Java 类的对象之前，需要先由虚拟机加载该 Java 类（类加载的过程在第 9 章进行了介绍）。在 Java 类被加载之后，还需要对该 Java 类进行链接和初始化。初始化完成之后，才能创建出该 Java 类的新的对象实例。对象也有自己的初始化过程，主要通过调用对应 Java 类的特定构造方法来完成。当不再有引用指向一个对象时，这个对象成为垃圾回收器的候选。对象所占用的内存空间会在合适的时机被垃圾回收器回收。对象终止机制提供了一种方式在对象被回收之前进行清理工作。当需要复制一个对象时，可以使用 `Object` 类的 `clone` 方法。如果需要将对象的状态持久化，可以使用对象序列化机制来得到一个方便存储的字节流。本章的内容围绕这些对象生命周期中可能涉及的相关技术展开。这些技术虽然并不复杂，但是存在很多容易误解和出错的地方。

10.1 Java 类的链接

Java 虚拟机运行时会在内部维护所有可用 Java 类的相关信息。虚拟机刚启动时，内部只包含 Java 核心类的相关信息。随着程序的运行，不断有新的 Java 类被加载到虚拟机中，变为可用状态。Java 类被加载之后，经过链接和初始化就可以在虚拟机中使用了。链接的过程是把加载的 Java 类的字节代码中包含的信息与虚拟机的内部信息进行合并，使 Java 类的代码可以被执行。链接的过程由 3 个子步骤组成，分别是验证、准备和解析。在链接过程中，会对 Java 类的直接父类或父接口进行验证和准备，但是对类中形式引用的解析是可选的。

验证是用来确保 Java 类的字节代码表示在结构上是完全正确的。验证过程有可能会导致其他 Java 类或接口被加载。如果验证过程中发现字节代码的格式不正确，会抛出 `java.lang.VerifyError` 错误。通过 Java 编译器生成的字节代码通常不会出现验证错误。使

用 ASM 等工具生成的字节代码可能会出现格式不正确的情况。

准备过程会创建 Java 类中的静态域，并将这些域的值设为默认值。在准备过程中并不会执行代码。准备过程中的一个重要环节是保证类加载时的类型安全。在链接过程中，可能有两个不同的类加载器对象同时开始加载一个 Java 类。在加载过程中，这两个类加载器对象也会分别加载 Java 类中的域和方法的参数和返回值引用的其他 Java 类。从类型安全的角度来说，不应该出现一个方法的参数类型对应的 Java 类，以及返回值类型对应的 Java 类由不同的类加载器对象来定义的情况。在准备过程中，当虚拟机中的某个类加载器对象开始加载某个类时，虚拟机会把该类加载器对象记录为该 Java 类的初始类加载器。记录完成后，虚拟机会马上进行一次检查。如果发现刚才的加载操作导致类型安全约束被破坏，则类加载过程不能进行。虚拟机会抛出 `java.lang.LinkageError` 错误。

解析过程是处理所加载的 Java 类中包含的形式引用。在一个 Java 类中会包含对其他类或接口的形式引用，包括它的父类、所实现的接口、方法的形式参数和返回值的 Java 类等。这些形式引用对应的 Java 类都被正确加载之后，当前 Java 类才能正常工作。在 Java 类中可能包含了对其他类中方法的调用，对于这些方法调用，在解析过程中需要检查所调用的方法确实存在。

在解析过程中会遇到的一个问题是处理复杂的引用关系图。Java 类之间的引用关系可能非常复杂。在解析一个 Java 类的过程中，可能导致其他的 Java 类被加载和解析，从而导致更多的 Java 类被加载和解析。通常可以利用两种策略来处理这种情况：一种是提前解析，即在链接时递归地对依赖的所有形式引用都进行解析，这种做法的缺点是性能比较差；另外一种策略是延迟解析，即只在真正需要一个形式引用时才进行解析，也就是说，如果一个 Java 类只是被引用，没有在程序运行中被真正用到，这个类就不会被解析。这种做法解决了提前解析方式性能较差的问题。不同的虚拟机实现可能采取不同的策略，不管采用哪种策略，都不会对程序的正确性造成影响。

代码清单 10-1 中给出了一个测试虚拟机的类解析策略的 Java 程序。类 `LazyLink` 中引用了类 `ToBeLinked`，但是没有创建 `ToBeLinked` 类的对象或引用类的静态域或方法。在 Java SE 7 的 OpenJDK 实现中，先把 `ToBeLinked` 类的字节代码删掉，再运行 `LazyLink` 类，程序并没有抛出错误，而是可以正确运行。这说明 OpenJDK 采用了延迟解析的策略。虽然 `LazyLink` 类引用了 `ToBeLinked` 类，但是在最开始的时候，`ToBeLinked` 类只是作为一个形式引用存在。在 `LazyLink` 类的运行过程中并没有实际用到 `ToBeLinked` 类，因此 `ToBeLinked` 类不会被加载和解析。所以即便 `ToBeLinked` 类的字节代码不存在，程序运行也不会出现错误。

代码清单 10-1 测试虚拟机的类解析策略的示例程序

```
public class LazyLink {
    public static void main(String[] args) {
        ToBeLinked toBeLinked = null;
```

```

        System.out.println(" 使用延迟解析。");
    }
}

```

如果把代码清单 10-1 中的 “ToBeLinked toBeLinked = null;” 改成 “ToBeLinked toBeLinked = new ToBeLinked();”，再按照相同的方式运行，则会抛出异常。这是由于程序运行中需要创建 ToBeLinked 类的对象，因此需要把 ToBeLinked 类加载到虚拟机中并进行链接。

10.2 Java 类的初始化

当一个 Java 类第一次被真正使用时，虚拟机会对该 Java 类进行初始化。初始化的主要工作是执行 Java 类中的静态代码块和初始化静态域。在初始化过程中，Java 类中的静态代码块和静态域会按照在代码中出现的顺序依次执行初始化。在当前 Java 类被初始化之前，它的直接父类也会被初始化，但是该 Java 类所实现的接口并不会被初始化。对一个接口来说，当其被初始化时，它的父接口不会被初始化。

在初始化的过程中，静态代码块和静态域的出现顺序很重要。虚拟机会严格按照在源代码中的出现顺序来执行初始化操作。代码清单 10-2 展示了静态域的初始化顺序可能带来的问题。静态域 X 的值最初是 20，所以 Y 的值是 40。虽然随后通过静态代码块把 X 的值设成了 30，但是 Y 的值不会发生变化，仍然是 40。

代码清单 10-2 静态域的初始化顺序的示例

```

public class StaticOrder {
    public static int X = 20;
    public static int Y = 2 * X;
    static {
        X = 30;
    }
    public static void main(String[] args) {
        System.out.println(Y); // 输出 40
    }
}

```

需要注意的是，当访问一个 Java 类或接口中的静态域时，只有真正声明这个域的类或接口才会被初始化。代码清单 10-3 给出了一个类继承时的静态域的初始化示例。在类 A 中声明了静态域 value，类 B 继承自类 A。通过 B.value 可以直接访问类 A 中声明的静态域 value。虽然引用时使用的是类 B，但是由于 value 是在类 A 中声明的，因此访问 B.value 只会使类 A 被初始化，类 B 不会被初始化。

代码清单 10-3 类继承时的静态域的初始化示例

```

class A {
    static int value = 100;
    static {

```

```

        System.out.println("类 A 初始化。");
    }
}

class B extends A {
    static {
        System.out.println("类 B 初始化。");
    }
}

public class StaticFieldInit {
    public static void main(String[] args) {
        System.out.println(B.value);
    }
}

```

有很多不同的原因会使一个 Java 类被初始化。下面列出了可能造成类被初始化的操作。

- 1) 创建一个 Java 类的实例对象。比如调用 “MyClass obj = new MyClass();” 语句时，类 MyClass 会被初始化。
- 2) 调用一个 Java 类中的静态方法。比如 myMethod 是 MyClass 类中的静态方法，调用 myMethod 方法会使 MyClass 类被初始化。
- 3) 为类或接口中的静态域赋值。比如 myField 是 MyClass 类中的静态域，调用 “MyClass.myField = 10” 会使 MyClass 类被初始化。
- 4) 访问类或接口中声明的静态域，并且该域的值不是常值变量。常值变量是声明为 final 的 Java 基本类型或 String 类型的变量，使用编译时常量来初始化。比如，代码 “private static final String name = "Alex";” 中的域 name 的值是常值变量。如果类 MyClass 中的静态域 myField 的值不是常值变量，访问 myField 域的操作会使类 MyClass 被初始化。
- 5) 在一个顶层 Java 类中执行 assert 语句也会使该 Java 类被初始化。
- 6) 调用 Class 类和反射 API 中进行反射操作的方法也会初始化 Java 类。

10.3 对象的创建与初始化

在 Java 语言中通过 new 操作符可以创建出一个 Java 类的实例对象。除了 Object 类之外，其他 Java 类都至少有一个父类。在没有通过 extends 显式声明时，Object 类是默认的父类。在 Java 类中可以通过构造方法添加对象初始化时的逻辑。在创建对象时，父类和祖先类的初始化逻辑会被依次执行。实际的初始化流程是先沿着继承层次结构树往上传递，完成部分初始化工作。到达 Object 类之后，再沿着层次结构树向下，完成其余的初始化工作，最后回到最初的 Java 类。任何一个步骤出现错误，都会导致对象创建失败。

在进行实际的对象创建之前，需要为要创建的对象分配内存空间。所需要的内存空间大小取决于 Java 类及其父类和祖先类包含的所有实例域的数量和类型。如果内存空间不足，则创建过程会抛出 OutOfMemoryError 错误。如果内存分配成功，则把新创建的对象的所有实例域都设为默认值，包括 Java 类本身声明的及父类声明的。这个新创建的对象并不能直接使用，因为类的构造方法还没有被调用。

类的构造方法的调用过程分成三步。第一步是调用父类的构造方法。对父类构造方法的调用分成显式和隐式两种。显式调用是通过 super 关键词来完成的。如果没有显式地添加对 super 关键词的调用，则由编译器自动生成相关的代码。第二步是初始化类中实例域的值，按照实例域的出现顺序依次初始化。第三步是执行类的构造方法中的其他代码，完成最终的初始化工作。由于在第一步中会调用父类的构造方法，实际的执行流程会先跳转到父类的构造方法，再沿着继承层次结构树依次往上跳转，直到到达 Object 类。由于 Object 类没有父类，不再继续向上传递，而是进行后两步操作。整个过程是一个典型的递归调用过程。

通过代码清单 10-4 的示例来说明构造方法的调用过程。Animal 类是 Dog 类的父类。在创建 Dog 类的对象时，Dog 类的构造方法先被调用。在 Dog 类的构造方法中使用参数值“4”调用父类 Animal 的构造方法。在 Animal 类的构造方法的最开始，会再调用 Animal 类的父类 Object 类的构造方法。由于 Object 类没有父类，不再需要继续调用父类的构造方法，而是执行自身的构造方法的逻辑。Object 类的构造方法执行完成之后，调用流程回到 Animal 类。会先对 Animal 类中实例域进行初始化，使 legs 变量被初始化为 0。接着 Animal 类的构造方法中的其他代码被调用，legs 变量的值被设置为 4。接着调用流程转入到 Dog 类的构造方法中，先把实例变量 name 的值初始化为“<default>”，然后调用 Dog 类的构造方法中的其他代码。此时就完成了 Dog 类的对象的创建过程。

代码清单 10-4 说明构造方法调用顺序的示例

```

class Animal {
    int legs = 0;
    Animal(int legs) {
        this.legs = legs;
    }
}

class Dog extends Animal {
    String name = "<default>";
    Dog() {
        super(4);
    }
}

public class NewObject {
    public static void main(String[] args) {
        Dog dog = new Dog();
    }
}

```

```

        System.out.println(dog.legs); // 输出值为 4
    }
}

```

在编写构造方法时，要注意不要在构造方法中调用可以被子类覆写的方法。这是因为如果子类覆写了该方法，那么在初始化过程中进行到调用父类的构造方法时，父类的构造方法所调用的是子类所覆写的方法，而此时子类的构造方法中的代码并没有被执行，对象仍处于初始化的过程中。这时调用子类的方法，很容易出现错误。代码清单 10-5 中给出了一个示例，在父类 Parent 的构造方法中调用 getCount 方法，在子类 Child 中覆写了此方法并返回在 Child 类的构造方法中传入的值。这两个类初看起来似乎并没有问题，但是当试图创建 Child 类的实例时，会出现除零错误。这是因为在执行 Parent 类的构造方法时使用的是 Child 类中的 getCount 方法，而此时 Child 类的实例域 count 还没有被初始化，它的值是 0。

代码清单 10-5 在父类构造方法中调用子类方法的错误示例

```

class Parent {
    public Parent() {
        int average = 30 / getCount();
    }
    protected int getCount() {
        return 4;
    }
}

class Child extends Parent {
    private int count;
    public Child(int count) {
        this.count = count;
    }
    public int getCount() {
        return count;
    }
}

public class BadConstructor {
    public void test() {
        Child child = new Child(5);
    }
}

```

10.4 对象终止

对象创建完成后，使用一段时间就可能不再需要了。如果没有引用指向一个对象，说明该对象可以被销毁。在创建和使用对象的过程中，可能申请了相关的资源，在对象

被销毁之前，这些资源要被正确地释放。资源分成内存资源和非内存资源两类。内存资源指的是对象中实例域所占据的内存空间。由于 Java 采用了自动内存管理机制，对象占用的内存资源的回收由垃圾回收器自动完成，不需要开发人员显式地进行内存释放。非内存资源指的是程序运行时申请的其他系统资源，包括打开的文件、打开的套接字连接和数据库连接等。这些非内存资源需要由程序显式地进行释放。

与 C++ 语言进行对比可以发现，在 Java 中，对这两种资源的释放操作是分开处理的，而在 C++ 语言中，对这两种资源的释放方式是统一的，都在析构方法中进行。在 Java 中，没有析构方法的概念，同时，对非内存资源的释放又无法以自动的方式来进行，因此，Java 引入了对象终止机制（finalization）来解决非内存资源的释放问题。但是由于设计上的各种问题和功能上的局限性，对象终止机制并没有发挥应有的作用。

1. finalize 方法的基本用法

如果一个 Java 类的对象有自定义的销毁逻辑，那么可以覆写 Object 类的 finalize 方法并在 finalize 方法中添加相关的逻辑。在一个对象的内存空间被垃圾回收器回收之前，该对象的 finalize 方法会被调用。finalize 方法中的这一段处理逻辑称为对象的终止器（finalizer）。Java 的对象终止机制看起来比较有用，但是在实际的程序中并不实用。最主要的原因是 Java 语言规范并没有对 finalize 方法的调用时间进行明确的规定，只是规定 finalize 方法一定在对象的内存空间被垃圾回收器回收之前运行。第 7 章介绍垃圾回收器时提到过，垃圾回收器的运行时间是不固定的，因此一个对象被回收的时间也是不确定的。这双重的不确定性导致无从得知 finalize 方法的具体运行时间。这就意味着如果把非内存资源的释放操作放在 finalize 方法中，那么该资源的实际释放时间是不固定的，从而可能产生与时间相关的错误。如果在某个时间点上 finalize 方法碰巧被执行了，那么程序的行为是正确的；如果 finalize 方法没有被执行，则可能资源没被正确释放。这种随机错误显然是不能出现在程序中的。

代码清单 10-6 中给出了运行 finalize 方法的示例。RunFinalize 类通过覆写 finalize 方法提供了自定义的对象终止逻辑。如果 finalize 方法被运行了，那么会在控制台输出提示信息。在代码运行时首先创建了一个 RunFinalize 类的对象，然后显式地把对象引用设为 null，使该对象可以被垃圾回收。如果程序运行到此处就结束，那么会发现 finalize 方法并没有被运行。这是因为垃圾回收器并没有运行，也就不可能调用对象的 finalize 方法。在后面的代码中通过 System.gc 方法来建议垃圾回收器运行并等待一段时间。通过这种方式可以增加垃圾回收器运行的几率，也使创建的 RunFinalize 类的对象有机会被回收。添加这样的逻辑之后，finalize 方法被运行的几率大大增加。

代码清单 10-6 运行 finalize 方法的示例

```
public class RunFinalize {
    protected void finalize() throws Throwable {
        System.out.println("运行 finalize 方法。");
        super.finalize();
    }
}
```

```

    }
    public static void main(String[] args) throws InterruptedException {
        RunFinalize runFinalize = new RunFinalize();
        runFinalize = null;

        for (int i = 0; i < 10; i++) {
            System.gc();
            Thread.sleep(100);
        }
    }
}

```

需要注意的是，虽然通过 `System.gc` 方法可以增加垃圾回收器运行的几率，进而增加 `finalize` 方法被运行的几率，但是 `finalize` 方法的实际运行时间仍然是不能保证的。代码清单 10-6 只是为了说明 `finalize` 方法的运行时机与垃圾回收器的关系，不应该作为实际程序中的处理方式。

2. `finalize` 方法与资源释放

在实际的程序中，不应该仅依靠对象的 `finalize` 方法来进行非内存资源的释放。例如，在一个对象的使用过程中打开了多个文件，如果把文件的关闭操作放在 `finalize` 方法中进行，则可能会出现问题。代码清单 10-7 中给出了一个错误使用 `finalize` 方法的示例。在类 `FileHolder` 的构造方法中传入一个要打开的文件的路径，在 `open` 方法中打开该文件得到一个 `InputStream` 类的对象。需要对打开的文件执行正确的关闭操作。`FileHolder` 类把 `InputStream` 类的对象的关闭操作放在了 `finalize` 方法中。这样做时 `finalize` 方法的运行时间是不确定的，有可能出现的情况是，程序中存在大量 `FileHolder` 类的对象，而这些对象的 `finalize` 方法都没有被调用，导致大量处于打开状态的文件没有被关闭。操作系统对同时打开的文件数量是有限制的，大量打开文件会造成程序运行出现严重问题。

代码清单 10-7 错误使用 `finalize` 方法来关闭文件的示例

```

// 错误的 finalize 使用
public class FileHolder {
    private Path path;
    private InputStream inputStream;
    public FileHolder(Path path) {
        this.path = path;
    }

    public void open() throws IOException {
        this.inputStream = Files.newInputStream(path, StandardOpenOption.WRITE);
    }

    protected void finalize() throws Throwable {
        if (inputStream != null) {

```

```

        inputStream.close();
    }
    super.finalize();
}
}

```

正确释放非内存资源的做法应该是在类中添加显式释放资源的方法，由对象的使用者负责调用。Java 类可以实现 Java 7 中新增的 `java.lang.AutoCloseable` 接口，进而可以通过调用 `close` 方法或者使用更加简便的 `try-with-resources` 语句来进行资源释放。对于代码清单 10-7 中的 `FileHolder` 类，正确的做法是实现 `AutoCloseable` 接口，并在 `close` 方法中关闭 `InputStream` 类的对象。提供显式的资源释放方法相当于把资源释放的职责交给了对象的使用者。这要求使用者在编写代码时注意在合适的时机释放所申请的非内存资源。在添加了显式的资源释放方法之后，也可以在 `finalize` 方法中添加对这个方法的调用。这样做的好处是，即使对象的使用者忘记调用释放资源的方法，也可能有机会释放这个资源。

3. 实现正确的 `finalize` 方法

在 `finalize` 方法的声明中，`finalize` 方法可能抛出任何类型的异常。如果 `finalize` 方法在执行时出现异常，则抛出的异常会直接被忽略，对 `finalize` 方法的调用也马上终止。由于 `finalize` 方法是由虚拟机来直接调用的，因此无法在代码中捕获 `finalize` 方法中抛出的异常，也不会有异常的堆栈信息被输出。

在自定义的 `finalize` 方法的实现中，总是应该调用父类的 `finalize` 方法。这是因为在对象创建时，会依次调用父类的构造方法来完成对象的初始化。与之相对应的是，在对象被回收之前，父类的终止逻辑也要被调用。但是与构造方法不同的是，父类的 `finalize` 方法不会被自动调用。因此，在 `finalize` 方法的实现中，先编写当前类的终止逻辑，再通过 `super.finalize()` 来调用父类的 `finalize` 方法。由于在 `Object` 类中定义了 `finalize` 方法，因此 `super.finalize()` 的调用总是合法的。为了保证父类的 `finalize` 方法总是被调用，需要把当前类的终止逻辑封装在一个 `try-finally` 结构中。不管当前类的终止逻辑是否成功完成，父类的 `finalize` 方法始终会被调用。

如果当前 Java 类通过覆写 `finalize` 方法添加了相关的对象终止逻辑，同时该类的子类也覆写了 `finalize` 方法，则子类的 `finalize` 方法应该调用父类的 `finalize` 方法。但是由于子类的实现不由当前 Java 类控制，因此可能会因为开发人员的错误而造成当前类的 `finalize` 方法没有被调用。为了避免这种情况，可以使用一种被称为“终止器守卫者（finalizer guardian）”的模式。如代码清单 10-8 所示，`WithFinalizer` 类有自定义的对象终止逻辑，但是这些代码没有添加在 `WithFinalizer` 类的 `finalize` 方法中，而添加在 `WithFinalizer` 类的一个实例域 `guardian` 的 `finalize` 方法中。当 `WithFinalizer` 类的对象可以被回收时，`guardian` 对象也同样可以被回收。此时 `guardian` 对象的 `finalize` 方法会被调用，完成 `WithFinalizer` 类的对象的终止逻辑。当使用这种模式时，即便 `WithFinalizer`

类的子类没有调用 `super.finalize` 方法，`WithFinalizer` 类的对象也能被正确终止。

代码清单 10-8 “终止器守卫者 (finalizer guardian)” 模式的示例

```
public class WithFinalizer {
    private final Object guardian = new Object() {
        protected void finalize() throws Throwable {
            //WithFinalizer 类的对象终止实现
            super.finalize();
        }
    };
}
```

在 `finalize` 方法的实现中要避免创建对当前对象的新的引用，不论是直接引用还是间接引用，都要避免。创建新的引用会造成当前对象从没有引用的状态又回到有引用的状态。不过当对象再次变成没有引用的状态时，该对象的 `finalize` 方法不会被再次调用。一个对象的 `finalize` 方法只会被调用一次。

10.5 对象复制

在程序运行过程中，可能会需要复制一个对象。例如，根据防御式编程 (defensive programming) 的实践，当一个方法将一个内部使用的对象返回给调用者时，最好先把该对象复制一份，将复制的对象返回给调用者。如果不进行复制，则调用者和当前对象使用的是同一个对象。如果调用者对这个对象进行了修改，那么会影响当前对象的内部状态。对象复制功能还可以在其他场景下发挥作用。`Object` 类中的 `clone` 方法和 `java.lang.Cloneable` 接口用来提供标准的对象复制功能。

要实现对象复制功能，需要 `Object` 类的 `clone` 方法和 `Cloneable` 接口配合使用。`Cloneable` 是一个不包含任何方法的标记接口，它的作用是作为复制功能相关的标记。如果一个类实现了 `Cloneable` 接口，就说明可以通过 `Object` 类的 `clone` 方法提供的默认实现来对该类的实例对象包含的域进行复制。如果调用 `clone` 方法的对象的 Java 类没有实现 `Cloneable` 接口，那么 `clone` 方法会直接抛出 `java.lang.CloneNotSupportedException` 异常。代码清单 10-9 给出了一个简单的可以进行复制操作的 Java 类。如果删去 `CloneableObject` 类对 `Cloneable` 接口的实现，则代码运行时会抛出 `CloneNotSupportedException` 异常。按照惯例，实现了 `Cloneable` 接口的 Java 类需要提供一个公开的 `clone` 方法来覆写 `Object` 类中的 `clone` 方法，这是因为 `Object` 类中的 `clone` 方法被声明为受保护的，如果不进行覆写，外部的对象无法访问到 `clone` 方法。

代码清单 10-9 可以进行复制操作的 Java 类的示例

```
public class CloneableObject implements Cloneable {
    public Object clone() {
        try {
```

```

        return super.clone();
    } catch (CloneNotSupportedException e) {
        throw new Error(e); // 不会发生该异常
    }
}

public static void main(String[] args) {
    CloneableObject obj = new CloneableObject();
    obj.clone();
}
}

```

Java 中的对象复制操作的这种设计，不太符合一般的习惯做法。一般认为在 Cloneable 接口中有一个 clone 方法，如果 Java 类需要提供复制功能，就实现 Cloneable 接口并编写对应的 clone 方法的实现。但是 Java 的实现并不是这样的。即便一个 Java 类实现了 Cloneable 接口，也不表示可以调用该类的 clone 方法进行复制操作，一种可能是该 Java 类并没有提供公开的 clone 方法，因此无法调用 clone 方法。

Object 类的 clone 方法复制对象的做法是对当前对象中所有的实例域进行逐一复制。先创建一个新的对象，再把新对象中所有的实例域的值初始化成原始对象中对应域的当前值。该方法一般是使用原生代码实现的。代码清单 10-10 给出了使用 Object 类的 clone 方法的示例。ToBeCloned 类的 clone 方法直接通过 Object 类的 clone 方法来实现。ToBeCloned 类包含一个 int 类型的实例域 value。在 cloneObject 方法中先创建了一个 ToBeCloned 类的对象 obj。在调用对象 obj 的 clone 方法时，先创建一个 ToBeCloned 类的对象，再把该对象中实例域 value 的值初始化为对象 obj 中 value 的当前值。这个新创建的 ToBeCloned 类的对象 clonedObj，就是复制对象 obj 之后的结果。

代码清单 10-10 Object 类的 clone 方法的使用示例

```

class ToBeCloned implements Cloneable {
    private int value = 0;

    public void setValue(int value) {
        this.value = value;
    }

    public int getValue() {
        return this.value;
    }

    public Object clone() {
        try {
            return super.clone();
        } catch (CloneNotSupportedException e) {
            throw new Error(e);
        }
    }
}

```

```

    }
}

public class SimpleClone {
    public void cloneObject() {
        ToBeCloned obj = new ToBeCloned();
        obj.setValue(1);
        ToBeCloned clonedObj = (ToBeCloned) obj.clone();
        System.out.println(clonedObj.getValue());
    }
}

```

可以将 Object 类的 clone 方法的实现看成是为原始对象创建了一个浅拷贝。如果对象中只包含值为基本类型或不可变对象的域，浅拷贝就足够了。如果对象中某些域的值为可变对象，浅拷贝就不能满足需求。因为所复制出来的对象的域与原始对象的域使用相同的对象引用，指向的是同一个对象，相当于在两个对象中对同一个对象进行处理，会产生潜在的问题。代码清单 10-11 给出了一个浅拷贝可能带来的问题的示例。类 MutableObject 中包含一个实例域 counter，是 Counter 类的对象。Counter 类的对象是可变的，有自己不同的内部状态值 value。类 MutableObject 的 clone 方法只是简单地复用了 Object 类中的 clone 方法。在进行复制操作之后，原始对象 obj 和复制出来的新对象 clonedObj 内部的 counter 域都指向对象 obj 中的 Counter 类的对象，所以虽然只有一次对 clonedObj 对象的 increase 方法的调用，但是 clonedObj 对象的 getValue 方法的返回值却为 3，这是因为通过对象 obj 的 increase 方法所做的修改同样影响了 clonedObj 对象。

代码清单 10-11 浅拷贝可能带来的问题的示例

```

class Counter {
    private int value = 0;
    public void increase() {
        value++;
    }
    public int getValue() {
        return value;
    }
}

class MutableObject implements Cloneable {
    private Counter counter = new Counter();
    public void increase() {
        counter.increase();
    }

    public int getValue() {
        return counter.getValue();
    }
}

```

```

public Object clone() {
    try {
        return super.clone();
    } catch (CloneNotSupportedException e) {
        throw new Error(e);
    }
}

public class MutableObjectClone {
    public void cloneObject() {
        MutableObject obj = new MutableObject();
        obj.increase();
        MutableObject clonedObj = (MutableObject) obj.clone();
        clonedObj.increase();
        obj.increase();
        System.out.println(clonedObj.getValue()); // 值为 3
    }
}

```

要解决这个浅拷贝的问题，就要提供自己的深拷贝的实现。虽然 Object 类的 clone 方法已经不能满足需求，但是仍然可以作为实现的基础。Object 类的 clone 方法已经对类中的基本类型和不可变对象的域进行了处理，只要在这基础上添加对可变对象的域的处理即可。要对代码清单 10-11 中的类进行修改，先要让 Counter 类实现 Cloneable 接口，并提供对应的公开的 clone 方法。由于 Counter 类中只包含一个 int 类型的域，因此可以直接调用 Object 类中的 clone 方法。而对于 MutableObject 类中的 clone 方法，将其修改成代码清单 10-12 中的实现。在这个实现中，先调用 Object 类的 clone 方法得到复制之后的对象 obj 作为基础，再对 MutableObject 类中的可变对象的域 counter 进行处理。使用 clone 方法对原始的 counter 对象进行复制，再修改对象 obj 中 counter 域的值，使之指向复制出来的 counter 对象。经过这样的修改，obj 对象中的 counter 域引用的是一个新的 Counter 类的对象。

代码清单 10-12 深拷贝的实现方式的示例

```

public Object clone() {
    MutableObject obj;
    try {
        obj = (MutableObject) super.clone();
        obj.counter = (Counter) counter.clone();
        return obj;
    } catch (CloneNotSupportedException e) {
        throw new Error(e);
    }
}

```

这种深拷贝操作要求对当前对象的实例域所引用的可变对象都以递归的方式进行复

制。其中所涉及的每个对象的类都应该实现 Cloneable 接口，并提供正确的 clone 方法的实现。

进行对象复制的另外一个做法是使用复制构造方法，即用一个已有的对象去构造另外—个对象。复制构造方法相对于 clone 方法来说更加容易使用，也不容易出错。如果一个 Java 类需要提供公开 API 来进行复制操作，使用复制构造方法是更好的选择。复制构造方法的一个好处是可以在功能相似而类型不同的对象之间传递数据。可以在 Java 的集合类框架中看到相关的示例。比如 java.util.ArrayList 的复制构造方法可以接受任何 java.util.Collection 接口的实现对象作为参数值。可以很容易地把一个 java.util.HashSet 类的对象转换成 ArrayList 类的对象。使用 clone 方法是无法做到这一点的。

代码清单 10-13 中的 User 类除了一般的构造方法之外，还提供了一个复制构造方法。当需要复制一个已有的 User 类的对象时，可以使用已有的对象作为参数来调用复制构造方法。创建出来的新对象就是已有对象的副本。

代码清单 10-13 包含复制构造方法的 Java 类的示例

```
public class User {
    private String name;
    private String email;

    public User(String name, String email) {
        this.name = name;
        this.email = email;
    }

    public User(User user) {
        this.name = user.getName();
        this.email = user.getEmail();
    }

    public String getName() {
        return this.name;
    }

    public String getEmail() {
        return this.email;
    }
}
```

10.6 对象序列化

在程序的运行过程中，活动对象的状态保存在内存中。内存中的数据是非持久化的。当虚拟机终止时，内存中对象的信息就会丢失。能够持久化保存对象数据的方式有

很多。典型的方式是使用文件系统或数据库。持久化对象时通常涉及自定义存储格式。使用文件存储时可以基于标准的文件格式，如 XML、JSON 和 CSV 等，也可以使用自定义的文本或二进制格式。使用数据库存储时需要定义数据库的表结构。定义了存储格式之后，在保存和读取操作时，需要在活动对象的内部状态和存储格式之间互相转换。

代码清单 10-14 给出了一个简单的 Java 类 User，表示程序中的用户。User 类中只有两个 String 类型的域 name 和 email。在持久化 User 类的对象时，可以选择使用基于 XML 的文件格式，即定义两个子元素来分别包含 name 和 email 的值；也可以创建一个数据库表，包含对应于两个域的列。在保存时，使用 User 类的对象的公开方法获取两个域的值，再根据存储格式使用文件操作 API 或数据库操作 API 来写入数据。在读取时，先获取保存的两个域的值，再使用 User 类的构造方法得到对应的对象。

代码清单 10-14 说明对象持久化机制的示例 Java 类

```
public class User {  
    private String name;  
    private String email;  
    public User(String name, String email) {  
        this.name = name;  
        this.email = email;  
    }  
    public String getName() {  
        return this.name;  
    }  
    public String getEmail() {  
        return this.email;  
    }  
}
```

在持久化实现中，自定义存储格式的工作量相对较小，比较复杂的是保存和读取操作时的数据转换。数据转换这一步通常涉及与持久化实现方式相关的外部 API，如 XML、JSON 处理 API 和数据库操作 API 等。代码清单 10-14 中的 User 类的内部结构相对简单，实现的工作量比较小。某些 Java 类的内部结构比较复杂，包含对其他 Java 类的对象的引用，对象之间通过引用关系形成一个复杂的对象图。在保存和读取过程中都需要遍历整个对象图，对其中包含的所有对象进行处理。如果使用数据库作为存储方式，那么可以利用对象关系映射（Object-relational mapping，ORM）技术来减少实现的工作量。

从简化实现的角度出发，可以使用 Java 语言内建的对象持久化方式，即对象序列化（object serialization）机制。对象序列化用来在虚拟机中的活动对象和字节流之间进行转换。序列化机制通常包括两个过程：第一个是序列化（serialization），把活动对象的内部状态转换成一个字节流；第二个是反序列化（deserialization），从一个字节流中得到可以直接使用的 Java 对象。这两个过程对应于保存和读取两个不同的操作。序列化

操作可以作为持久化实现的基础。通过序列化得到的字节流可以保存在文件中，也可以作为二进制数据保存在数据库中。序列化的好处是为开发人员省去了繁琐的数据转换操作，所有的数据转换都由 Java 平台来完成，并提供相应的扩展方式允许开发人员进行自定义。

10.6.1 默认的对象序列化

Java 平台提供了良好的默认序列化实现。在此基础上实现基本的对象序列化是比较简单的。与 10.5 节介绍的对象复制功能一样，在使用 Java 平台的默认序列化实现之前，需要先通过接口声明启用对象序列化功能。待序列化的 Java 类只需实现 `java.io.Serializable` 接口即可启用这个功能。`Serializable` 仅是一个标记接口，并不包含任何需要实现的具体方法。实现该接口的目的是声明该 Java 类的对象是可以被序列化的。对于支持序列化的 Java 类的对象，可以使用 `java.io.ObjectOutputStream` 类和 `java.io.ObjectInputStream` 类的对象来完成 Java 对象与字节流之间的相互转换。

`ObjectOutputStream` 类是一个 Java I/O 库中标准的过滤输出流的实现。在创建 `ObjectOutputStream` 类的对象时，需要提供另外一个 `OutputStream` 类的对象作为实际的输出目的。在 `ObjectOutputStream` 类中包含一系列以“`write`”作为名称前缀的方法用于写入基本类型的值和对象到输出流中，其中 `writeObject` 方法用于把一个 Java 对象写入到输出流中。

代码清单 10-15 给出了把 `User` 类的对象写入到文件中的示例代码。只需要从文件输出流中创建一个 `ObjectOutputStream` 类的对象，再使用 `writeObject` 方法进行写入即可。得到的“`user.bin`”文件采用了 Java 平台定义的二进制格式。开发人员并不需要关心格式的具体细节。任何 `OutputStream` 类的对象都可以作为序列化时的输出目标。除了写入到文件中之外，还可以通过套接字连接进行网络传输。

代码清单 10-15 写入 Java 对象的内容到文件中

```
public class WriteUser {
    public void write(User user) throws IOException {
        Path path = Paths.get("user.bin");
        try (ObjectOutputStream output = new ObjectOutputStream(Files.
            newOutputStream(path))) {
            output.writeObject(user);
        }
    }

    public static void main(String[] args) throws IOException {
        WriteUser writeUser = new WriteUser();
        User user = new User("Alex", "alex@example.org");
        writeUser.write(user);
    }
}
```

对象序列化之后得到的字节流可以通过不同的方式进行分发，比如文件复制或网络传输。在得到字节流之后，可以通过反序列化方式从字节流中得到 Java 对象。反序列化时使用的是与 ObjectOutputStream 类相对应的 ObjectInputStream 类。在创建 ObjectInputStream 类的对象时需要提供一个 InputStream 类的对象作为参数。该 InputStream 类的对象用于读取包含序列化操作结果的字节流。ObjectInputStream 类中包含一系列以“read”作为名称前缀的方法来从输入流中读取其中包含的基本类型数据和对象，其中 readObject 方法用于读取一个 Java 对象。读取到的 Java 对象中实例域的值由字节流中保存的值来确定。代码清单 10-16 给出了读取代码清单 10-15 中产生的“user.bin”文件包含的 Java 对象的示例。

代码清单 10-16 从文件中读取 Java 对象

```

public class ReadUser {
    public User readUser() throws IOException, ClassNotFoundException {
        Path path = Paths.get("user.bin");
        try (ObjectInputStream input = new ObjectInputStream(Files.
            newInputStream(path))) {
            User user = (User) input.readObject();
            return user;
        }
    }

    public static void main(String[] args) throws ClassNotFoundException,
        IOException {
        ReadUser readUser = new ReadUser();
        User user = readUser.readUser();
        System.out.println(user.getName());
    }
}

```

在写入和读取时，虽然提供的参数或得到的返回值是单个 Java 对象，但是实际上操纵的是一个对象图。该对象图中包括当前对象所引用的其他对象，以及这些对象所引用的另外的对象。Java 的序列化机制会自动遍历整个对象图并依次进行处理。在写入过程中，如果传递给 ObjectOutputStream 类的对象的 writeObject 方法的 Java 对象并没有实现 Serializable 接口，那么 writeObject 方法会抛出 java.io.NotSerializableException 异常。而对于 Java 对象中的域所引用的其他对象，如果这些对象本身并没有实现 Serializable 接口，那么这个域不会出现在序列化之后的字节流中。

在默认的序列化实现中，Java 对象中的非静态和非瞬时域都会被自动包括进来，而与域的可见性声明没有关系。这可能会导致某些不应该出现的域被包含在序列化之后的字节流中，比如密码等隐私信息对应的域，进而造成隐私信息的泄露，成为程序中的安全隐患。针对这种情况，一种解决办法是把不希望被序列化的域声明为瞬时的，即使用 transient 关键词。另外一种做法是添加一个 serialPersistentFields 域来声明序列化时

要包含的域。比如对于代码清单 10-14 中的 User 类，如果不希望 email 域出现在序列化之后的字节流中，可以使用代码清单 10-17 中的声明方式，只包含 name 域。在使用 serialPersistentFields 时，它的名称和类型声明需要严格按照代码清单 10-17 给出的形式。如果名称或类型声明不对，则无法生效。具体来说，serialPersistentFields 域的值是一个 java.io.ObjectStreamField 类型的数组，数组中的每个 ObjectStreamField 类的对象表示一个包含在序列化过程中的域。

代码清单 10-17 声明序列化中需要包含的域

```
private static final ObjectStreamField[] serialPersistentFields = { new
ObjectStreamField("name", String.class) };
```

10.6.2 自定义对象序列化

默认的对象序列化机制虽然使用简单，但是存在的问题也比较多。其中最重要的问题是默认的序列化机制依赖于 Java 类的内部实现，而不是公开接口。随着程序的版本更新，公开接口基本上不会发生变化，而内部的实现可能发生很多变化。内部实现的变化会导致旧版本的 Java 对象序列化之后的字节流无法被重新转换成新版本的 Java 对象。这与通常的“面向接口编程”的实践方式是相背离的。

比如，代码清单 10-14 中的 User 类需要增加获取用户年龄的功能，从公开 API 的角度来说，需要增加一个 getAge 方法获取用户的年龄。早期版本的实现是用一个 int 类型的域 age 来直接保存用户的年龄数据，在构造方法中设置该域的值。在后来的版本更新中，发现直接保存年龄的做法并不合适，改为根据出生日期自动计算出年龄。于是把原来实现中的 age 域删除，增加一个 java.util.Date 类型的域 birthDate，并修改 getAge 方法的内部实现。在这个版本更新过程中，类的公开接口并没有变化，但内部实现发生了改变。如果某个文件中保存的是旧版本的对象序列化之后的字节流，那么在通过 ObjectInputStream 类的 readObject 方法读取之后得到的新版本的对象中，域 birthDate 的值是 null。这是因为旧版本的序列化结果中只有 age 域的值，域 birthDate 的值被设为默认值 null。默认的序列化机制并不理解 age 域和 birthDate 域之间的关系，只会根据域的名称和类型来进行赋值。

为了解决版本更新带来的序列化格式不兼容的问题，需要为 Java 类定义自己的序列化格式，而不是简单地使用默认格式。这就要求对 Java 类的职责及可能会发生变化的地方进行缜密的思考与设计，定义好序列化后的格式中要包含的数据。这个格式在之后的版本更新中应该是相对稳定的。完成设计之后，通过序列化机制提供的扩展方式来编写相关的逻辑。

通过在 Java 类中添加特定的方法来编写自定义的序列化实现逻辑。自定义的序列化逻辑由两个配对的方法来实现，分别是 writeObject 和 readObject 方法。当使用 ObjectOutputStream 类的对象进行序列化时，如果 Java 类中定义了 writeObject 方法，那

么会调用该方法来完成实际的写入操作；当使用 `ObjectInputStream` 类的对象进行反序列化时，如果 Java 类中定义了 `readObject` 方法，那么会调用该方法进行实际的读取操作。这两个方法一般同时出现，但是所包含的逻辑正好相反。对 `writeObject` 和 `readObject` 方法的类型声明有严格的要求。不满足要求的方法不会在序列化时被调用。

以之前提到的在 `User` 类中添加表示年龄的域的需求为例来说明自定义序列化格式的用法。经过设计，在序列化后的格式中，只需要包含年龄的值即可。代码清单 10-18 给出了新的 `NewUser` 类的代码。`NewUser` 类中的 `writeObject` 方法中包含的是序列化时的逻辑。在 `writeObject` 方法被调用时，当前进行对象写入操作的 `ObjectOutputStream` 类的对象会作为参数传入。使用该 `ObjectOutputStream` 类的对象中的方法可以写入任何所需的内容。在 `writeObject` 方法中一般先使用 `defaultWriteObject` 方法来执行默认的写入操作，即写入非静态和非瞬时的域。这样可以提高代码的灵活性。接着把 `age` 域的值通过 `.writeInt` 方法写入流中。与 `writeObject` 方法相对应的 `readObject` 方法中包含的是反序列化时的逻辑。在 `readObject` 方法被调用时，当前进行读取操作的 `ObjectInputStream` 类的对象会作为参数传入。使用该 `ObjectInputStream` 类的对象来读取流中的内容，并初始化对象中的对应实例域。在 `readObject` 方法中一般先使用 `defaultReadObject` 方法来读取非静态和非瞬时域。在 `writeObject` 方法和 `readObject` 方法中的操作是相对应的，按照写入时的顺序来进行读取。

代码清单 10-18 自定义序列化机制

```
public class NewUser implements Serializable {
    private static final long serialVersionUID = 1L;
    private transient int age;
    public NewUser(int age) {
        this.age = age;
    }

    public int getAge() {
        return this.age;
    }

    private void writeObject(ObjectOutputStream output) throws IOException {
        output.defaultWriteObject();
        output.writeInt(getAge());
    }
    private void readObject(ObjectInputStream input) throws IOException,
        ClassNotFoundException {
        input.defaultReadObject();
        int age = input.readInt();
        this.age = age;
    }
}
```

代码清单 10-18 中的 writeObject 方法和 readObject 方法的实现都比较简单。实际上，可以在 writeObject 方法和 readObject 方法中添加比较复杂的逻辑，包括修改域的值或添加额外的数据等。如果在自定义的 writeObject 方法和 readObject 方法中对某个域的数据进行了处理，一般把该域声明为 transient，这样 defaultWriteObject 方法和 defaultReadObject 方法就不会处理这个域，避免默认实现带来的不兼容性问题。

使用自定义序列化格式可以解决之前提到的由于版本更新造成的序列化内容不兼容的问题。代码清单 10-19 给出了新版本的 NewUser 类。新版本的 NewUser 类使用了 Date 类型表示出生日期。为了保持序列化格式的兼容性，在序列化之前需要把 birthDate 域转换成年龄，在反序列化之后要进行相反的操作。

代码清单 10-19 版本更新后的序列化机制的实现

```
public class NewUser implements Serializable {
    private static final long serialVersionUID = 1L;
    private transient Date birthDate;
    public NewUser(Date birthDate) {
        this.birthDate = birthDate;
    }

    public int getAge() {
        return dateToAge(birthDate);
    }

    private Date ageToDate(int age) {
        // 年龄转换成日期
    }

    private int dateToAge(Date date) {
        // 日期转换成年龄
    }

    private void writeObject(ObjectOutputStream output) throws IOException {
        output.defaultWriteObject();
        int age = dateToAge(birthDate);
        output.writeInt(age);
    }

    private void readObject(ObjectInputStream input) throws IOException,
        ClassNotFoundException {
        input.defaultReadObject();
        int age = input.readInt();
        this.birthDate = ageToDate(age);
    }
}
```

通过 ObjectInputStream 类的 readObject 方法可以从字节流中得到一个 Java 对象。

不过对象的创建并不是通过一般的方式来完成的，对应的 Java 类的构造方法没有被调用，而不少 Java 类在其构造方法中有相关的对象初始化的逻辑。通过反序列化得到的对象由于没有调用构造方法，其内部的完整性约束可能被破坏，因此在 `readObject` 方法中需要确保对象在返回之前经过了完整的初始化。

10.6.3 对象替换

在进行反序列化操作时，会创建出新的 Java 对象。从某种意义上来说，反序列化的作用相当于一个构造方法。有些 Java 类对于其对象实例有自己的管理逻辑，例如使用单例模式时，要求某个类在虚拟机中只有一个实例。对于这样的 Java 类，在反序列化时，需要对得到的对象进行处理，以保证序列化机制不会破坏 Java 类本身的约束，否则，每进行一次反序列化操作，都会创建一个新的对象，就破坏了单例模式要求的约束条件。在其他情况下，可能需要在序列化时使用另外一个对象来代替当前对象，原因可能是当前对象中包含了一些不需要被序列化的域，比如这些域是从另外一个域派生而来的。对于这种情况，也可以通过把域声明为 `transient` 或使用自定义的 `writeObject` 方法和 `readObject` 方法来实现。但是如果这样的逻辑比较复杂，可以考虑封装在一个 Java 类中。另外还可以隐藏实际的类层次结构。比如类 A 中的某个域引用了类 B，在正常的序列化过程中，类 A 和类 B 都会出现在序列化之后的字节流中，如果希望在字节流中隐藏类 B，可以用另外一个类的对象来代替类 B 的对象。

在序列化时进行的对象替换操作由一组对应的 `writeReplace` 方法和 `readResolve` 方法来完成。在 `writeReplace` 方法中可以根据当前对象创建一个新的对象作为替代。这个替代对象被写入到 `ObjectOutputStream` 类的对象中。同样的，在 `readResolve` 方法中，可以对读取出来的对象进行转换，把转换的结果作为反序列化的结果返回。

通过一个示例来说明，在一个订单管理系统中，使用 `Order` 类的对象来表示实际的订单。在 `Order` 类中引用了 `User` 类，用来表示该订单的客户信息。如果使用默认的序列化机制，在 `Order` 类的对象被序列化时，其引用的 `User` 类的对象也会被序列化。为了满足使用的需求，需要把 `Order` 类的对象序列化后的字节流通过网络的方式来传输。通过网络传输可能会带来一些安全隐患，比如序列化后的内容被第三方窃取，造成客户信息的泄露。比较好的做法是为 `Order` 类定义一个专门的传输格式，只包含尽可能少的信息。进行序列化时，实际上使用的是专门的传输对象。代码清单 10-20 给出了 `Order` 类的代码。在 `Order` 类的 `writeReplace` 方法中，从当前对象中创建出一个传输时使用的 `OrderTO` 类的对象。

代码清单 10-20 序列化时进行对象替换的示例

```
public class Order implements Serializable {
    private User user;
    private String id;
    public Order(String id, User user) {
```

```

        this.id = id;
        this.user = user;
    }
    public String getId() {
        return this.id;
    }
    private Object writeReplace() throws ObjectOutputStreamException {
        return new OrderTO(this);
    }
}

```

代码清单 10-21 给出了 OrderTO 类的实现。OrderTO 类本身使用了默认的序列化格式，只包含 OrderTO 类中表示订单编号的 orderId 域的内容。在调用 readResolve 方法时，基本的反序列化操作已经完成，orderId 域已经被初始化为正确的值。在 readResolve 方法中通过相关的查找逻辑根据域 orderId 的值得到对应的 Order 类的对象。把 Order 类的对象作为 readResolve 方法的返回值，即反序列化的最终结果。对调用者来说，替换对象的存在是透明的。

代码清单 10-21 替换对象的 Java 类

```

public class OrderTO implements Serializable {
    private String orderId;
    public OrderTO(Order order) {
        this.orderId = order.getId();
    }
    private Object readResolve() throws ObjectStreamException {
        return OrderGateway.getOrder(orderId);
    }
}

```

经过对象替换之后，序列化之后的字节流中只包含订单的编号，并没有其他额外的信息。因此，即便被第三方窃取，也不会出现信息泄露。

10.6.4 版本更新

在介绍自定义序列化格式时提到了由于版本更新带来的序列化格式不兼容的问题。版本更新所带来的不兼容问题是程序开发中的常见问题。版本升级之后，可能造成之前的程序无法正确运行。不过，如果程序中使用了序列化机制，兼容性的问题就会更多。旧版本的 Java 对象序列化后的字节流通常会保存在磁盘文件或数据库之中。这些持久存储中的数据的保存时间一般都很长，而且都记录了程序运行的历史数据。当版本更新之后，要求这些历史数据仍然是可用的。程序在更新过程中应该尽可能保证序列化格式的兼容性。但是在有些情况下，可能无法做到完全兼容，可以选择从某些版本开始不再与之前的版本保持兼容。一般来说，在新的版本中添加域不会产生什么问题，而删去一些域或改变已有域的行为则不行。

在使用 `ObjectInputStream` 类的对象读取一个旧版本的 Java 对象的序列化结果时，会尝试在当前虚拟机中查找其 Java 类的定义。如果找不到类的定义，就尝试加载该 Java 类，被加载的有可能是新版本的 Java 类。需要一种方式来判断旧版本的序列化内容是否与当前的 Java 类兼容。如果不兼容，那么读取操作会直接失败。这种兼容性的判断是通过 Java 类中的全局唯一标识符 `serialVersionUID` 来实现的。当 Java 类实现了 `Serializable` 接口时，需要声明该 Java 类唯一的序列化版本号。这个版本号会被包括在序列化后的字节流中。在进行读取时，会比较从字节流中得到的版本号与对应 Java 类中声明的版本号是否一致，以确定两者是否兼容。只有在版本一致时，序列化操作才能完成。代码清单 10-18 和代码清单 10-19 中的两个 `NewUser` 类的 `serialVersionUID` 域的值均为 1，说明这两个类在序列化格式上是兼容的。如果 `NewUser` 类的后续更新造成了序列化格式不再兼容，那么需要把 `serialVersionUID` 域设置成另外一个不同的值。

如果 Java 类实现了 `Serializable` 接口，但是没有显式地声明 `serialVersionUID` 域，那么虚拟机会根据 Java 类的各种元素的特征计算出一个散列值，作为 `serialVersionUID` 域的值。默认生成的 `serialVersionUID` 域的值会随着 Java 类的更新而发生变化。如果使用默认的序列化格式，那么默认生成的 `serialVersionUID` 域是合理的。如果使用自定义的序列化格式，则通常要显式地声明 `serialVersionUID` 域的值。可以通过 Java 提供的 `serialver` 命令行工具生成一个 Java 类的 `serialVersionUID` 域的值。在 Eclipse 中，如果 Java 类实现了 `Serializable` 接口，那么 Eclipse 会提示并辅助生成这个 `serialVersionUID` 域。

10.6.5 安全性

在使用了对象序列化机制之后，在虚拟机中运行的活动对象的内部状态被持久化，可以通过网络及其他方式进行传输。这会使一些内部数据存在暴露的风险。对象序列化后的字节流的格式是固定的，可以很容易地从中分析出相关的数据。为了避免产生安全方面的问题，应该从多个方面着手解决。

第一个方面是根据信息隐藏的原则，尽可能地减少序列化结果中包含的信息。所包含的信息越少，泄露之后造成的影响就越小。通过自定义的序列化格式和对象替换可以实现这一点。

第二个方面是对包含序列化结果的字节流进行保护。措施主要是加密和解密。可以对序列化后的字节流的整体使用各种加密算法进行处理。在反序列化之前再使用相应的解密算法进行处理即可。加密和解密也可以在序列化的过程中进行。在 Java 类的 `writeObject` 方法中，可以在写入域的值之前，先进行加密操作。在 `readObject` 方法中，在读取域的值之后，先进行解密操作，再赋值给对象中的域。这两种方式可以结合起来形成复杂的加密方案。

第三个方面是从字节流中进行反序列化操作时进行数据完整性验证。在使用 `ObjectInputStream` 类的对象进行读取操作时，输入字节流的内容有可能已经被篡改，因此在 Java 类的 `readObject` 方法中需要添加相应的验证代码来检查完整性是否被破坏，比

如验证域的值是否为 null，以及域的值是否在合理的范围之内等。在 `readObject` 方法中的处理适合于对单个 Java 类的对象进行验证。要对一个完整的对象图进行验证，可以通过 `ObjectInputStream` 类的 `registerValidation` 方法添加 `java.io.ObjectInputValidation` 接口的实现对象，进而添加完整的对象验证逻辑。通常的做法是在 `readObject` 方法中处理完所有域之后，再添加一个 `ObjectInputValidation` 接口的实现对象来进行完整的验证。验证通常在引用关系根节点的 Java 类的 `readObject` 方法中进行。代码清单 10-22 给出了 `NewUser` 类的对象的验证方法的示例。类 `UserValidator` 完成具体的验证工作，检查 `age` 域的值是否合法。如果验证中发现了错误，则直接抛出 `java.io.InvalidObjectException` 异常。在 `readObject` 方法中，对作为参数传入的 `ObjectInputStream` 类的对象添加 `UserValidator` 类的对象来执行验证逻辑。

代码清单 10-22 对象完整性验证的示例

```

private void readObject(ObjectInputStream input) throws IOException,
    ClassNotFoundException {
    input.defaultReadObject();
    int age = input.readInt();
    this.birthDate = ageToDate(age);

    input.registerValidation(new UserValidator(this), 0);
}

private static class UserValidator implements ObjectInputValidation {
    private NewUser user;
    public UserValidator(NewUser user) {
        this.user = user;
    }
    public void validateObject() throws InvalidObjectException {
        if (user.getAge() < 0) {
            throw new InvalidObjectException("非法的年龄数值。");
        }
    }
}

```

10.6.6 使用 Externalizable 接口

Java 类只需要简单地实现标记接口 `Serializable` 就可以使用 Java 平台提供的默认序列化机制。如果希望进行自定义的序列化处理，那么可以在 Java 类中添加 `writeObject` 方法和 `readObject` 方法。虽然可以进行自定义，但是完整的序列化过程仍然是由 Java 平台来控制的，无法修改序列化之后的二进制格式。如果希望对序列化的过程进行完全的控制，那么可以实现 `java.io.Externalizable` 接口。`Externalizable` 接口继承自 `Serializable` 接口，包含 `writeExternal` 和 `readExternal` 两个方法。在使用 `ObjectOutputStream` 类的对象写入 Java 对象时，如果该对象的 Java 类实现了 `Externalizable` 接口，则 `writeExternal`

方法会被调用；在使用 `ObjectInputStream` 类的对象读取 Java 对象时，如果实现了 `Externalizable` 接口，则 `readExternal` 方法会被调用。方法 `writeExternal` 和 `readExternal` 的作用类似于自定义序列化操作时使用的 `writeObject` 和 `readObject` 方法，只不过这两个方法是在 `Externalizable` 接口中显式声明的，更容易被开发人员所理解。代码清单 10-23 给出了一个实现了 `Externalizable` 接口的 Java 类的示例。在 `writeExternal` 方法中，使用作为参数传入的 `java.io.ObjectOutput` 类的对象来写入数据；在 `readExternal` 方法中，则使用作为参数传入的 `java.io.ObjectInput` 类的对象来读取数据并进行赋值。

代码清单 10-23 `Externalizable` 接口的使用示例

```
public class ExternalizableUser implements Externalizable {
    private String name;
    private String email;

    public ExternalizableUser() {
    }

    public ExternalizableUser(String name, String email) {
        this.name = name;
        this.email = email;
    }

    public String getName() {
        return this.name;
    }

    public String getEmail() {
        return this.email;
    }

    public void writeExternal(ObjectOutput out) throws IOException {
        out.writeUTF(getName());
        out.writeUTF(getEmail());
    }

    public void readExternal(ObjectInput in) throws IOException,
        ClassNotFoundException {
        name = in.readUTF();
        email = in.readUTF();
    }
}
```

实现 `Externalizable` 接口的 Java 类必须具备一个不带参数的公开构造方法。在反序列化的过程中，如果对象的 Java 类实现了 `Externalizable` 接口，则先调用不带参数的公开构造方法得到一个新的对象，再在此对象上调用 `readExternal` 方法。

10.7 小结

本章的主要内容是关于 Java 对象生命周期的各个不同阶段的技术细节的。在对象被创建之前，对象的 Java 类需要被正确地加载、链接和初始化。在对象的创建过程中，会调用当前类和父类的构造方法来完成对象本身的初始化工作。当不再需要对象时，可以将其销毁。可以通过对象复制机制来复制一个 Java 对象。当需要保存对象的内部状态时，可以将对象序列化机制和持久化技术结合起来使用。

在使用这些技术时会遇到的一个问题是已有的技术不太符合一般的惯例，容易造成开发人员的误解。比如 Cloneable 接口中并没有声明 clone 方法，而是通过声明 Cloneable 接口来改变 Object 类中的 clone 方法的行为。在自定义序列化操作时，很多方法的使用都是隐式的，如 writeObject、readObject、writeReplace 和 readResolve 方法等。为了正确使用序列化机制，开发人员需要了解这些隐含方法的使用。

第 11 章 多线程与并发编程实践

提到使用 Java 开发多线程程序，很多开发人员可能都认为是一个非常艰巨的任务。虽然有非常多的书籍或在线资料来介绍 Java 多线程开发相关的知识，但是开发出正确的多线程程序对开发人员来说仍然充满挑战。Java 多线程开发中主要有三个困难。首先是编写正确的程序很难。不论是 Java 中的基本原语，还是 `java.util.concurrent` 包中的辅助工具类，要正确使用都不容易。其次是很难测试程序是否正确。在实际运行时，多个线程中的代码的执行顺序是无法确定的。程序在很多次运行时都不出现错误，并不表示该程序就是完全正确的。最后是在程序出现问题时很难调试。进行调试的一个先决条件是找到确定的重现错误的步骤。一个多线程程序可能在连续正常运行很长时间后突然出现与多线程相关的错误。在修改代码之后，即便程序运行了很长时间都没出现问题，也不能保证问题已经被修复。

对于 Java 多线程开发中的困难也并非“无计可施”。两个重要的武器是基础知识和实用工具。善用实用工具可以提高多线程程序的开发效率，避免常见的错误；而掌握基础知识是解决特定问题和正确使用工具的基础。本章围绕这两方面展开，大致上可以划分成三个部分：第一个部分是与多线程相关的基础知识，主要从 Java 虚拟机的层次进行介绍；第二个部分是基本的线程同步方式，主要介绍锁机制和 `Object` 类中的 `wait`、`notify` 和 `notifyAll` 等方法；第三个部分是 `java.util.concurrent` 包中提供的抽象层次更高的实用工具，以及 Java SE 7 中新增的内容。

11.1 多线程

在操作系统中，两个比较容易混淆的概念是进程（process）和线程（thread）。操作系统中的进程是一个计算机程序的运行实例。计算机程序中包含了需要执行的指令，而进程则表示正在执行的指令。对同一个计算机程序可以创建多个进程。这些进程的运行状态各不相同。进程一般作为资源的组织单位。进程有自己独立的地址空间，包含程序内容和数据。不同进程的地址空间是互相隔离的。进程拥有各种资源和状态信息，包括打开的文件、子进程和信号处理器等。线程表示的是程序的执行流程，是 CPU 调度执行的基本单位。线程有自己的程序计数器、寄存器、堆栈和帧等。同一进程中的线程共用相同的地址空间，同时共享进程所拥有的内存和其他资源。

引入线程的主要动机在于提高程序的运行性能。在一个程序中主要存在使用 CPU 和 I/O 操作的两类计算。I/O 操作相对 CPU 运算来说比较耗时，而且很多都是阻塞式的。当一个线程所执行的 I/O 操作被阻塞时，同一进程中的其他线程可以使用 CPU 来进行计

算。在资源允许时，多个线程可以同时进行 I/O 操作。这种方式提高了操作系统中资源的使用效率，进而提高了程序的运行性能。线程的概念在主流操作系统和编程语言中都得到了支持。不同操作系统和编程语言中的线程的使用方式有很大的区别。这对于开发跨平台的多线程程序来说是一个不小的挑战。Java 平台通过 Java 虚拟机解决了跨平台的问题，使由相同 API 开发的多线程程序在不同平台上都能够正确运行。

Java 标准库提供了与进程和线程相关的 API。第 6 章具体介绍了表示进程的 `java.lang.Process` 类和创建进程的 `java.lang.ProcessBuilder` 类的使用方式。表示线程的是 `java.lang.Thread` 类。在虚拟机启动之后，通常只有一个普通线程来运行程序的代码。这个线程用来启动主 Java 类的 `main` 方法的运行。程序在运行时可以根据需要创建新的线程并启动线程的运行。除了普通线程之外，还有一类线程是守护线程（daemon thread）。守护线程一般在后台运行，提供程序运行时所需的服务。当虚拟机中运行的所有线程都是守护线程时，虚拟机终止运行。

线程表示的是一段程序的执行过程。线程中最重要的部分是所要执行的代码逻辑。有两种方式可以创建线程。第一种方式是继承 `Thread` 类并覆写 `run` 方法，在 `run` 方法中包含线程的执行逻辑。第二种方式是实现 `java.lang.Runnable` 接口，并在 `Thread` 类的构造方法中传入 `Runnable` 接口的实现对象。使用这种方式创建的 `Thread` 类的对象的 `run` 方法中，调用的实际是 `Runnable` 接口中的 `run` 方法。在创建出 `Thread` 类的对象之后，通过调用 `start` 方法启动线程的运行。当线程的代码逻辑执行完毕之后，线程会自动结束。

Java 线程 API 的具体实现由底层的 Java 虚拟机来负责提供。为了更好地理解线程 API 的使用及多线程开发，需要对虚拟机内部的相关机制有一定的了解。下面的内容围绕 Java 语言规范中定义的 Java 内存模型展开，涉及可见性问题的基本概念以及相关的 `volatile` 和 `final` 关键词。

11.1.1 可见性

在一个多线程程序中，多个线程通过共同协作来完成指定的任务。在协作的过程中，线程之间需要进行数据交换以协调各自的状态。同一个进程中的各个线程通过共享内存的方式来进行通信。由于这些线程共享所在进程的地址空间，因此都可以自由访问所需的内存位置。互相协作的线程之间对共享的内存位置达成一致。一个线程在适当的时候修改该内存位置的值，另外一个线程在后续的操作中通过读取相同内存位置来得到修改后的值。Java 中的代码无法直接操作内存，而是通过不同类型的变量来间接访问。比如线程 A 和线程 B 共同协作完成某项任务，线程 B 需要等待线程 A 完成其任务之后才能继续运行，两个线程可以使用一个 `boolean` 类型的变量来协调状态。当线程 A 完成任务之后，修改该变量的值为 `true`，以通知线程 B。线程 B 在运行时，如果读取到该变量的值为 `true`，就开始执行自身的操作。使用共享内存方式在多线程程序中可能造成可见性相关的问题，即一个线程所做的修改对于其他的线程并不可见，导致其他的线程仍

然使用错误的值。比如线程 B 看不到线程 A 对该 boolean 类型变量所做的修改，造成线程 B 一直等待下去。

每个线程在运行过程中所做的事情是按照代码中编写的逻辑来执行对应的虚拟机字节代码指令。Thread 类或 Runnable 接口实现类中的 run 方法所包含的代码就是线程要执行的指令序列的来源。对于一个单线程程序来说，可见性的含义是很容易理解和验证的。在代码执行过程中，如果先改变一个变量的值，再读取该变量的值，那么所读取的值是上次写入操作所写入的值。也就是说前面的写入操作的结果对后面的读取操作是肯定可见的。这个在单线程程序中显而易见的特征，在多线程程序中则不一定成立。如果不使用某些互斥或同步机制，则不能保证一个线程所写入的值对另外一个线程是可见的。如果可见性条件不能成立，那么程序在运行中就会出现问题。

代码清单 11-1 给出了一个简单的 Java 类 IdGenerator，用来生成唯一的标识符。在单线程程序中，每次对 getNext 方法的调用都可以保证得到一个不重复的值。对于一个 IdGenerator 类的对象来说，域 value 的初始值为 0。每调用一次 getNext 方法，value 的值会加 1。每次调用后的 value 的新值，对后续的调用都是可见的。

代码清单 11-1 生成唯一标识符的 Java 类

```
public class IdGenerator {
    private int value = 0;
    public int getNext() {
        return value++;
    }
}
```

如果在多线程程序中使用 IdGenerator 类的对象，则不能保证每次调用 getNext 方法所返回的值是不重复的。在多线程程序中，可以运行的各个线程之间相互竞争 CPU 时间来获取运行的机会。一般 CPU 采用时间片轮转等不同算法来对线程进行调度。当调度发生时，当前正在运行的线程被暂停运行。CPU 进行上下文切换，开始运行其他线程的代码。在上下文切换时，之前运行线程的当前状态会被记录下来，以便在下次被调度时从上次被中断的地方继续运行。CPU 进行调度的时机是不可预知的，可能发生在当前运行线程的任何两条指令的执行间隙。代码清单 11-1 中的 getNext 方法虽然只有一行代码，但是对应于 7 条字节代码指令，具体的指令序列如代码清单 11-2 所示。

代码清单 11-2 IdGenerator 类的 getNext 方法的指令序列

```
aload_0
dup
getfield #12
dup_x1
iconst_1
iadd
putfield #12
```

这里对代码清单 11-2 中的指令序列进行简单说明：`aload_0` 指令的含义是把第一个局部变量加载到操作数堆栈中，这里的第一个局部变量是 `this`；`dup` 指令的含义是把操作数堆栈中的栈顶元素进行复制；`getfield` 指令的含义是获取操作数堆栈中栈顶元素表示的对象中的指定域的值，并将其压入堆栈中；`dup_x1` 指令的含义是复制操作数堆栈中的栈顶元素，并将其插入到距离栈顶两个元素的位置上；`iconst_1` 指令的含义是把常量 1 压入栈中；`iadd` 指令的含义是从操作数堆栈中弹出两个元素，进行整数加法操作，再把结果压入栈中；`putfield` 指令的含义是从操作数堆栈中依次弹出要设置的域的值和所在的对象的引用，进行域的赋值操作。这个指令序列的作用是先读取域 `value` 的值，再把该值加上 1，最后把所得的新值赋值给域 `value`。

当多个线程共享同一个 `IdGenerator` 类的对象时，可能某个线程在执行 `getNext` 方法对应的 7 条指令的过程中 CPU 进行了线程切换，该线程的运行被暂停，而另外一个共享了相同 `IdGenerator` 类的对象的线程获得了运行的机会。多个线程执行 `getNext` 方法的实际指令序列是交织在一起的。考虑下面的线程运行情况：该 `IdGenerator` 类的对象的域 `value` 的值当前为 1。线程 A 执行到了 `getfield` 指令，把获取的域 `value` 的值 1 压入操作数堆栈中。接着线程 A 暂停运行，线程 B 获得了运行的机会。当线程 B 执行 `getfield` 指令时，所看到的域 `value` 的值也是 1。线程 B 继续执行，把域 `value` 的值更新为 2 之后，返回域 `value` 之前的值 1 作为结果。然后线程 A 再次获得运行的机会，继续从上次暂停的指令开始运行。由于当前操作数堆栈中的值为 1，线程 A 仍然使用这个值来继续运行，把域 `value` 的值更新为 2，而返回的结果同样为 1。在这个情况下，线程 A 和线程 B 使用的是值相同的标识符，出现了错误。这是因为线程 B 对 `IdGenerator` 类的对象所做的修改并没有被线程 A 所看到。线程 A 仍然使用的是它上次读取操作时获取的旧值。

除了多个线程的实际执行顺序之外，还有其他一些原因会造成与可见性相关的问题。目前 CPU 一般采用层次结构的多级缓存的架构。有的 CPU 提供了 L1、L2 和 L3 三级缓存。现在硬件平台多采用多核 CPU 或多个 CPU。当 CPU 需要读取主存中某个位置的数据时，会依次检查各级缓存中是否存在对应的数据。如果有，直接从缓存中读取。这比从主存中读取速度快很多。在进行写入时，数据先被写入缓存中，之后在某个特定的时间被写回主存中。不同的 CPU 可能采用不同的写入策略，如写穿透或写返回等。由于缓存的存在，在某些时间点上，缓存中的数据与主存中的数据可能是不一致的。某个线程所执行的写入操作的新值当前可能在 CPU 的缓存中，还没有被写回到主存中，这时另外一个线程的读取操作所读取的可能还是主存中的旧值。这样的问题主要出现在包含多核 CPU 或多个 CPU 的操作系统中。另外一个造成与可见性相关的问题的原因是代码重排。出于性能的考虑，编译器在编译时可能会对生成的字节代码中的指令顺序进行重新排列，以优化指令的执行顺序。CPU 也可能改变指令的执行顺序。指令顺序的重新排列在单线程程序中通常不会产生问题，但是在多线程程序中可能产生与可见性相关的问题。

这些可见性相关的问题与底层硬件平台和操作系统密切相关。Java 平台的做法是提供一个抽象的模型来描述多线程程序中变量的访问语义，提供相关的 API 来允许开发人

员定义线程之间的交互方式，最后由 Java 虚拟机和编译器来保证该抽象模型的语义在不同平台上都有一致的行为。开发人员只需要正确使用 Java 平台提供的 API，就可以开发出线程安全的多线程程序。

11.1.2 Java 内存模型

Java 内存模型（Java Memory Model）描述了程序中共享变量的关系以及在主存中写入和读取这些变量值的底层细节。Java 内存模型作为一个抽象模型，只关注主存中的共享变量。只关注主存可以对开发人员屏蔽 CPU 缓存等细节，简化内存模型自身的定义。对象的实例域、静态域和数组元素存储在堆内存中，可以被多个线程共享。这些变量是内存模型需要关注的内容。局部变量、方法的形式参数和异常处理时的异常参数都是不被共享的，不会受到内存模型的影响。当对共享变量的两个访问中至少有一个是写入操作时，这两个访问存在冲突。两个互相冲突的访问的执行结果由内存模型来确定。

在一个单线程程序的运行过程中，指令的执行结果是可预期的。编译器有可能会对某些指令进行重排，但这些操作不会影响程序的行为。在一个多线程程序中，线程所执行的动作可以分成两类：一类是线程内部的动作，比如操作方法中的局部变量等；另外一类是线程之间的动作。线程之间的动作是由一个线程产生的动作，同时可以被另外一个线程检测到或受到另外一个线程的直接影响。线程之间的动作包括读取和写入共享变量以及加锁和解锁等同步操作。线程内部的动作不会产生可见性相关的问题，因此内存模型只考虑线程之间的动作。

在一个线程的运行过程中，如果假设该线程处于隔离状态，即其他线程都不存在时，那么该线程所产生的所有线程之间的动作会按照某个顺序来执行。这个执行顺序由程序内部的代码逻辑来确定，称为程序顺序（program order）。在多个线程同时运行时，所产生的线程间的动作会交织在一起，形成实际的执行顺序。如果这个实际的执行顺序与每个线程隔离运行时的程序顺序保持一致，同时每个读取操作所看到的都是最近一次写入操作的执行结果，则称这个执行顺序是具备顺序一致性的。顺序一致性是一个非常严格的约束。如果 Java 内存模型要求动作的执行顺序满足顺序一致性的要求，则很多 CPU 和编译器进行的优化措施都是不允许的。

线程间的动作中的很大一部分是同步动作。如果只考虑同步动作，则将这些动作的执行顺序称为同步顺序（synchronization order）。同步动作的出现在动作之间形成了同步关系。同步关系定义了同步动作之间的先后顺序。这种顺序是强制的。以加锁动作为例，在成功加锁之前，对应的锁要被释放，否则加锁动作无法成功。所以成功的加锁动作必然在某个解锁动作之后。常见的同步关系如下所示，其中“**A 与 B 保持同步**”的含义是 A 必然发生在 B 之前。

- 1) 在一个监视器对象上的解锁动作与相同对象上后续成功的加锁动作保持同步。
- 2) 对一个声明为 volatile 的变量的写入动作与同一变量的后续读取动作保持同步。
- 3) 启动一个线程的动作与该线程执行的第一个动作保持同步。

4) 向线程中共享变量写入默认值的动作与该线程执行的第一个动作保持同步。这种同步关系的含义是在线程运行之前，该线程所使用的全部对象从概念上说已经被创建出来，并且对象中的变量被赋值为默认值。这种同步关系的含义是保证线程所看到的变量的值是确定的。变量的值可能是根据变量类型确定的默认值，也可能是其他线程所设置的值，但不可能是其他值。

5) 线程 A 运行时的最后一个动作与另外一个线程中任何可以检测到线程 A 终止的动作保持同步。

6) 如果线程 A 中断线程 B，那么线程 A 的中断动作与任何其他线程检测到线程 B 处于被中断的状态的动作保持同步。

除了程序顺序和同步顺序之外，还有一种更加实用的顺序，即“在之前发生 (happens-before)”顺序。如果一个动作按照“在之前发生”的顺序发生在另外一个动作之前，那么前一个动作的执行结果在多线程程序中对后一个动作是肯定可见的。“在之前发生”顺序的情况包括：

1) 如果两个动作 A 和 B 在一个线程中执行，同时在程序顺序中 A 出现在 B 之前，则 A 在 B 之前发生。

2) 一个对象的构造方法的结束在该对象的 finalize 方法运行之前发生。

3) 如果动作 A 和动作 B 保持同步，则 A 在 B 之前发生。

4) 如果动作 A 在动作 B 之前发生，同时动作 B 在动作 C 之前发生，则 A 在 C 之前发生。也就是说，“在之前发生”顺序是传递性的。

如果程序中存在对共享变量的互相冲突的访问，且这些访问没有通过“在之前发生”顺序来正确排列时，那么程序中存在数据竞争 (data race)。数据竞争的存在是多线程运行时产生错误的根源。编写多线程程序时的首要任务是找出并解决程序中存在的数据竞争。代码清单 11-1 中的 getNext 方法在调用时存在数据竞争。不同线程都需要读写共享变量 value 的值，但是这些访问动作没有定义“在之前发生”顺序，因此程序运行可能出现错误。如果程序中不存在数据竞争，则程序的任何执行方式都具备顺序一致性。开发人员需要做的是利用 Java 平台提供的支持来消除程序中的数据竞争，这样就可以保证程序的正确性。不需要考虑 CPU 和编译器可能进行的指令重排，因为 Java 虚拟机和编译器会确保这些指令重排不会影响程序的正确性。

11.1.3 volatile 关键词

关键词 volatile 用来对共享变量的访问进行同步。对一个 volatile 变量的上一次写入操作和下一次读取操作之间存在“在之前发生”的顺序。也就是说，上一次写入操作的结果对下一次读取操作是肯定可见的。在写入 volatile 变量值之后，CPU 缓存中的内容会被写回主存；在读取 volatile 变量时，CPU 缓存中的对应内容被置为失效状态，重新从主存中进行读取。将变量声明为 volatile 相当于为单个变量的读取和写入添加了同步

操作。但是 volatile 在使用时不需要利用锁机制，因此性能要优于 synchronized 关键词。关键词 volatile 的主要作用是确保对一个变量的修改被正确地传播到其他线程中。最常使用 volatile 变量的一个场景是把作为循环结束的判断条件的变量声明为 volatile。比如有两个线程 A 和 B，线程 A 在一个循环中不断进行处理，线程 B 负责向线程 A 发送停止处理的信号。代码清单 11-3 中给出了这样的示例。线程 A 调用了 Worker 类的对象的 work 方法，开始执行具体的任务。在适当的时候，线程 B 会调用同一 Worker 类的对象的 setDone 方法来声明终止任务的执行。把 done 变量声明为 volatile 是很重要的。只有这样才能保证线程 B 对 done 变量所做的修改对于线程 A 的后续读取操作是可见的。否则，线程 A 可能由于无法看到 done 变量值的变化而一直运行下去。

代码清单 11-3 使用 volatile 变量作为循环结束的判断条件

```
public class Worker {
    private volatile boolean done;
    public void setDone(boolean done) {
        this.done = done;
    }
    public void work() {
        while (!done) {
            // 执行任务
        }
    }
}
```

虽然 volatile 关键词使用简单，但是由于在实现时没有锁机制的存在，volatile 关键词的适用场景是受限的。比如，对于代码清单 11-1 中的 IdGenerator 类，如果只是把域 value 声明为 volatile，这样是不够的，仍然会出现问题。这是因为写入的 value 的正确值依赖于 value 的当前值，而当前值有可能是不正确的。代码清单 11-3 中要写入变量 done 的新值与该变量的当前值没有关系，使用 volatile 就足够了。

11.1.4 final 关键词

在 Java 语言中，final 关键词有很多语义，比如声明为 final 的类无法被继承，声明为 final 的方法无法在子类中被覆写。从内存模型的角度来说，final 关键词最重要的语义是声明一个域的值只能被初始化一次。在初始化之后，该域的值无法被修改。在多线程程序开发中，final 域通常用来实现不可变对象（immutable object）。当对象中的共享变量的值不可能发生变化时，在多线程访问时不会出现问题，也就不需要使用线程之间的同步机制来进行处理。在 Java 标准库中最常用的不可变对象是 String 类的对象。在多线程程序中，应该尽可能地使用不可变对象，以避免使用同步机制。代码清单 11-4 给出了不可变对象的类的声明方式。把类中所有域声明为 final，并在构造方法中进行初始化。

代码清单 11-4 不可变对象的类的声明方式

```
public class User {
    private final String name;
    private final String email;
    public User(String name, String email) {
        this.name = name;
        this.email = email;
    }
}
```

在构造方法成功完成之前，要确保正在创建的对象的引用不会被其他线程访问到，否则，其他线程可能看到部分创建完成的对象。代码清单 11-5 给出了一个错误的示例。在 WrongUser 类的构造方法中，把当前对象的引用赋值给 UserHolder 类的静态变量 user，会导致使用 UserHolder 类的线程看到尚未创建完成的 WrongUser 类的对象。该对象中包含的变量可能没有被初始化成正确的值。

代码清单 11-5 在构造方法中添加当前对象的引用的错误示例

```
public class WrongUser {
    private final String name;
    public WrongUser(String name) {
        UserHolder.user = this;
        this.name = name;
    }
}
```

如果一个线程是在对象的构造方法成功完成之后才通过该对象的引用来进行访问的，那么该线程肯定可以看到对象中的 final 域被初始化之后的值。如果域没有被声明为 final，则构造方法完成之后，其他线程不一定可以看到这个域被初始化之后的值，而有可能看到域的默认值。由于 final 域具有这些特征，编译器对 final 域的处理是很灵活的。这些域可能被随意地与其他代码进行重新排列。在代码执行时，final 域的值可以被保存在寄存器中，而不用从主存中频繁重新读取。

11.1.5 原子操作

在介绍代码清单 11-1 中的 IdGenerator 类的 getNext 方法时提到过，Java 代码中的一条语句，可能实际上对应的是多条字节代码指令。CPU 在一条指令的执行过程中不会进行线程调度和上下文切换，但是在两条指令的执行间隙，可能发生线程切换。如果代码清单 11-1 中的 “value++” 语句由一条 CPU 指令来完成，就不存在多线程访问时可能出现的问题。

在 Java 中，对于非 long 型和 double 型的域的读取和写入操作是原子操作。对象引用的读取和写入操作也是原子操作。比如读取一个 int 类型的域时，该域对应的内存地

址中的 32 位的内容会被完整地读取，在读取过程中不会被其他线程所打断。在进行写入操作时也不会被打断。在写入非 volatile 的 long 型和 double 型的域的值时，分成两次操作来完成。一个 long 型或 double 型的域的长度是 64 位，每次写入 32 位。在一个线程写入了 long 型或 double 型的域的前 32 位之后，在写入后 32 位之前，另外一个线程有可能访问这个域的值，从而读取只完成部分写入操作的错误值。因此在多线程程序中使用 long 型和 double 型的共享变量时，需要把变量声明为 volatile，以保证读取和写入操作的完整性。

11.2 基本线程同步方式

对于多线程程序中存在的数据竞争，需要利用 Java 平台提供的同步机制来确保对共享变量的访问存在合适的“在之前发生”的顺序。Java 平台提供的基本同步方式包括 synchronized 关键词和 Object 类中提供的 wait、notify 和 notifyAll 方法。

11.2.1 synchronized 关键词

synchronized 关键词应该是最为开发人员所熟悉的多线程开发时可以使用的结构，是基本的线程同步方式之一。synchronized 关键词可以添加在方法或代码块之上。声明为 synchronized 的方法或代码块在同一时刻只能有一个线程允许访问。如果当前已经有线程正在访问 synchronized 方法或代码块，那么其他试图访问该方法或代码块的线程会处于等待状态。这种互斥性使该方法或代码块中的代码逻辑实际上成为一个原子操作。从“在之前发生”顺序的角度更容易理解 synchronized 关键词的含义。所有的 Java 对象都有一个与之关联的监视器对象（monitor），允许线程在该监视器对象上进行加锁和解锁操作。每个 synchronized 关键词在使用时都与一个监视器对象相对应。对于声明为 synchronized 的方法，静态方法对应的监视器对象是所在 Java 类对应的 Class 类的对象所关联的监视器对象，而实例方法使用的是当前对象实例所关联的监视器对象。对于 synchronized 代码块，对应的监视器对象是 synchronized 代码块声明中的对象所关联的监视器对象。

在一个线程允许执行方法或代码块之前，需要先获取对应的监视器对象上的锁。在执行完成之后，该线程所持有的锁会被自动释放。根据“在之前发生”顺序，上一次的解锁操作肯定在下一次的成功加锁操作之前发生。由于解锁操作是上一次线程在 synchronized 方法或代码块中执行的最后一个动作，而加锁操作是下一次线程执行 synchronized 方法或代码块时的第一个动作，所以上一次线程运行时对共享变量所做的修改对下一次线程中的动作是肯定可见的。这是开发人员使用 synchronized 关键词时可以得到的保证。Java 虚拟机和编译器会负责完成实际的工作。当锁被释放时，对共享变量的修改会从 CPU 缓存中直接写回到主存中；当锁被获取时，CPU 的缓存中的内容被置为无效的状态，从主存中重新读取共享变量的值。当有线程在执行 synchronized 方法

或代码块时，其他线程由于无法获取锁而处于等待状态，不会影响当前线程的运行。编译器在处理 synchronized 方法或代码块时，不会把其中包含的代码移动到 synchronized 方法或代码块之外，从而避免了由于代码重排而造成的问题。

synchronized 关键词的使用比较简单。代码清单 11-6 给出了对代码清单 11-1 中 IdGenerator 类的修改方式。类 SynchronizedIdGenerator 中的 getNext 方法使用了 synchronized 来声明，而 getNextV2 方法则包含了一个 synchronized 代码块。两个方法的作用是相同的。由于 getNext 方法是一个对象的实例方法，因此在同步时使用的监视器对象是当前对象实例所关联的监视器对象。而 getNextV2 方法中的 synchronized 代码块声明中也同样使用了当前对象实例 this。

代码清单 11-6 使用 synchronized 生成唯一标识符的 Java 类

```
public class SynchronizedIdGenerator {
    private int value = 0;
    public synchronized int getNext() {
        return value++;
    }
    public int getNextV2() {
        synchronized(this) {
            return value++;
        }
    }
}
```

在使用 synchronized 时有一个错误倾向，那就是被 synchronized 所保护的代码过多，比如一个方法中只有少数几行代码访问共享变量，却把整个方法声明为 synchronized。这么做虽然不会对程序的正确性造成影响，但是会影响程序的性能。正确的做法是把方法中需要同步的代码用 synchronized 代码块包围即可。

11.2.2 Object 类的 wait、notify 和 notifyAll 方法

使用 synchronized 关键词主要用来实现线程之间的互斥，即同一时刻只有一个线程允许执行特定的代码。通过互斥的方式来保证多个线程访问共享变量时的正确性。除了互斥访问之外，线程之间也需要通过协作的方式来完成某些任务。比如在典型的生产者－消费者场景中，生产者线程在缓冲区已满时需要等待，等消费者线程从缓冲区中取走数据之后，再进行生产；消费者线程在缓冲区已空时需要等待，等生产者线程向缓冲区中放入数据之后，再进行消费。这种协作方式可以抽象为等待－通知机制。在线程运行时，可能需要满足某些逻辑条件才能继续进行。当线程所要求的条件不满足时，线程进入等待状态，等待由于其他线程的运行而使条件得到满足；其他线程则负责在合适的时机发出通知来唤醒处于等待状态的线程，表明等待线程所要求的条件已经满足。这是一种在多线程程序中经常会遇到的典型场景，即判断是否满足条件之后再选择继续运

行。可以用代码清单 11-3 中的 while 循环和 volatile 变量来处理这个场景。但是这种做法的本质是让线程处于忙等待的状态，并通过轮询的方式来判断条件是否满足。处于忙等待状态的线程仍然需要占用 CPU 时间，会对性能造成影响。更好的做法是使用 Object 类提供的 wait、notify 和 notifyAll 方法。

由于 wait 方法是在 Object 类中定义的，因此可以调用任何 Java 对象的 wait 方法。使用 wait 方法的关键在于理解调用 wait 方法的含义。Java 中的每个对象除了有与之关联的监视器对象之外，还有一个与之关联的包含线程的等待集合。在调用 wait 方法时，该方法调用的接收者所关联的监视器对象是所使用的监视器对象，同时 wait 方法所影响的是执行 wait 方法调用的当前线程。成功调用 wait 方法的先决条件是当前线程获取到监视器对象上的锁。如果没有锁，则直接抛出 java.lang.IllegalMonitorStateException 异常，wait 方法调用失败；如果有锁，那么当前线程被添加到对象所关联的等待集合中，并释放其持有的监视器对象上的锁。当前线程被阻塞，无法继续执行，直到被从对象所关联的等待集合中移除。

由于 wait 方法的成功调用需要当前线程持有监视器对象上的锁，因此 wait 方法的调用需要放在使用 synchronized 关键词声明的方法或代码块中。当执行 wait 方法时，当前线程已经进入了 synchronized 关键词所声明的互斥块中，已经持有所需的锁。在 synchronized 方法或代码块中使用的监视器对象必须是 wait 方法调用的接收者所关联的监视器对象。

通过调用 wait 方法进入的等待状态分成无超时和有超时两种。如果线程处于有超时的等待状态，那么线程除了可以被主动唤醒而离开等待状态之外，设定的超时时间过去后也会自动离开等待状态。在设定超时时间时可以指定毫秒数和纳秒数。

wait 方法的作用是使当前线程进入等待状态，对应的 notify 和 notifyAll 方法用来通知线程离开等待状态。调用一个对象的 notify 方法会从该对象关联的等待集合中选择一个线程来唤醒。被唤醒的线程可以和其他线程竞争运行的机会。与 notify 方法相对应的 notifyAll 方法会唤醒对象关联的等待集合中的所有线程。而 notify 方法所唤醒的线程的选择由虚拟机实现来决定，不能保证一个对象所关联的等待集合中的线程按照所期望的顺序被唤醒。很可能一个线程被唤醒之后，发现它所要求的条件并没有满足，而重新进入等待状态，而真正需要被唤醒的线程却仍然处于等待集合中。因此，当等待集合中可能包含多个线程时，一般使用 notifyAll 方法。不过 notifyAll 方法会导致线程在没有必要的情况下被唤醒，之后又马上进入等待状态，因此会造成一定的性能影响，不过可以保证程序的正确性。与 wait 方法相同，notify 和 notifyAll 方法在调用时都要求当前线程拥有方法调用接收者所关联的监视器对象上的锁。当线程被唤醒之后，由于在调用 wait 方法时已经释放了之前所持有的监视器对象上的锁，线程需要重新竞争锁来获得继续运行 wait 方法调用完成之后的代码的机会。

需要注意的一个情况是，处于某个对象所关联的等待集合中的线程可能被意外唤醒。这种唤醒不是以开发人员可以预期的方式发生的，而是由底层操作系统和虚拟机内

部实现所产生的非正常行为。这种意外的唤醒无法避免，需要开发人员来处理。以生产者–消费者场景为例，当缓冲区已满时，生产者线程处于等待状态。消费者线程在从缓冲区中取走数据之后，唤醒生产者线程。但是，当生产者线程被唤醒时，并不意味着缓冲区肯定是不满的，因为生产者线程有可能在缓冲区已满时被意外唤醒。因此，通常要把 `wait` 方法的调用包含在一个循环中。循环的条件是线程可以继续执行需要满足的逻辑条件。如果线程继续执行的逻辑条件不满足，那么线程应该再次调用 `wait` 方法来重新进入等待状态。代码清单 11-7 给出了 `wait` 方法的一般使用方式。

代码清单 11-7 把 `wait` 方法调用置于循环之中

```
synchronized (obj) {
    while (/* 逻辑条件不满足 */) {
        obj.wait();
    }
    // 条件满足
}
```

11.3 使用 Thread 类

作为线程的跨平台抽象，`Thread` 类中提供的方法并不多。这主要是因为不同平台上的线程实现差别很大，很难提供一个完整的抽象。在 JDK 最早的版本中，`Thread` 类提供了对线程进行控制的一些方法，包括 `stop`、`suspend`、`resume` 和 `destroy` 等，分别用来停止、暂停和继续线程的执行及销毁一个线程。这些方法在随后的版本中被声明为废弃的，因为这些方法在实现上是不安全的，使用时可能会造成对象损坏和出现线程死锁的问题。这也从一个侧面说明了跨平台线程实现的复杂性。通过 `start` 方法启动一个线程的运行之后，应该让线程运行完其所包含的全部代码之后自动结束，而不需要通过 `stop` 方法强制终止一个线程。要暂停或继续一个线程的执行，可以使用 `Object` 类的 `wait` 和 `notify` 方法提供的线程等待和唤醒机制。

11.3.1 线程状态

在一个 `Thread` 类的对象被创建出来之后，它可能处于不同的状态中。进行与线程相关的不同操作可能导致该 `Thread` 类的对象所处的状态发生变化。不同的线程状态由枚举类型 `Thread.State` 来表示，可以通过 `Thread` 类的 `getState` 方法来得到。`Thread.State` 只表示虚拟机中线程的状态，并不表示对应的操作系统上的线程的状态。`Thread.State` 中包含的线程状态有以下几种。

- 1) NEW：线程刚被创建出来。一个新创建的 `Thread` 类的对象处于此状态中。
- 2) RUNNABLE：线程处于可运行的状态。该线程有可能正在运行，也有可能在等待其他操作系统中的资源。

3) BLOCKED：线程在等待一个监视器对象上的锁时，处于此状态。当一个线程尝试执行声明为 synchronized 的方法或代码块，又无法获取对应的锁时，处于 BLOCKED 状态。

4) WAITING：调用某些方法会使当前线程进入等待状态。这个等待没有超时时间。处于这个状态的线程等待其他线程执行特定的操作来使当前线程退出等待状态。

5) TIMED_WAITING：该状态类似于 WAITING，但是增加了指定的超时时间。当超时时间过去，如果线程等待的条件仍然没有发生，那么线程也会退出等待状态。

6) TERMINATED：线程的运行已经终止。

线程在同一时刻只能处于上述六种状态中的一种。了解线程的状态可以为调试提供帮助。

11.3.2 线程中断

线程中断是线程之间的一种通信方式。通过一个线程对应的 Thread 类的对象的 interrupt 方法可以向该线程发出中断请求。根据线程当前所处的状态不同，中断一个线程会产生不同的效果。在一般的情况下，中断一个线程会在线程对应的 Thread 类的对象上设置一个标记。该标记用来记录当前的中断状态。通过 Thread 类的 isInterrupted 方法可以查询此标记来判断是否有中断请求发生。当线程 A 通过调用线程 B 的 interrupt 方法来发出中断请求时，线程 A 再向线程 B 发出一个信号。线程 B 应该在方便的时候来处理这个中断请求，当然这不是必须的。一个线程可以选择忽略所有或部分中断请求。

Object 类的 wait 方法及 Thread 类的 join 方法和 sleep 方法都会抛出受检异常 java.lang.InterruptedException。在调用这 3 个方法及其重载形式时，必须捕获 InterruptedException 异常并进行处理。当线程由于调用这 3 个方法而进入等待状态时，通过 interrupt 方法中断该线程会导致该线程离开等待状态。对于 wait 方法调用来说，线程需要在重新获取到监视器对象上的锁之后才能抛出 InterruptedException 异常，并执行对 InterruptedException 异常的处理逻辑。

线程中断的一个典型应用场景是实现可取消的任务。有些线程在执行任务时会使用一个无限循环来重复执行。这时需要一种方式来结束线程的运行。一种做法是使用之前介绍的 volatile 变量作为结束标记；另外一种做法是向线程发出中断请求。仍然以生产者 - 消费者场景为例，对于生产者线程来说，只要缓冲区不为空，就会不断运行，产生新的数据。可以在循环条件中加上对当前线程对应的 Thread 类的对象的 isInterrupted 方法的调用，用来检查是否收到了中断请求。如果收到了中断请求，可以终止执行。

对 InterruptedException 异常需要谨慎地进行处理。在大多数时候，开发人员只是为了能够通过编译，简单地捕获 InterruptedException 异常，而不进行任何处理。这种做法通常是不合适的。当线程由于调用 wait、join 和 sleep 方法进入等待状态时，如果收到中断请求，线程就会进入 InterruptedException 异常的处理逻辑。如果在当前层次上处理

InterruptedException 异常是合理的，就直接进行处理；否则应该把 InterruptedException 异常重新抛出，由调用者进行处理。在 InterruptedException 异常发生时，当前线程对应的 Thread 类的对象内部的中断标记会被清空，相当于该中断请求已经被 InterruptedException 异常的处理逻辑处理了。如果在当前 InterruptedException 异常的处理代码中不适合处理该异常，又无法把该异常重新抛出，则需要通过 interrupt 方法来重新中断该线程。这样就保存了当前线程曾经被中断过的状态信息，可以让后续代码来处理该中断请求。

Thread 类中还有一个与线程中断相关的方法 interrupted。该方法不但可以判断当前线程是否被中断，还可以清除线程内部的中断标记。如果调用 interrupted 方法的返回值为 true，说明该线程曾经被中断过。在 interrupted 方法调用完成之后，线程内部的中断标记已经被清空。

如果一个线程当前处于某个对象所关联的等待集合中，那么中断该线程或发出唤醒通知都可以使该线程从等待集合中被移除。如果两者同时发生，那么具体的运行结果取决于两者实际发生顺序，由虚拟机实现来确定。通过 notify 方法发出的唤醒请求不会被线程中断所影响。当调用某个对象上的 notify 方法时，至少有一个线程被唤醒，或者所有的线程都被中断，唤醒请求不会丢失。如果一个线程被选为唤醒的对象，而该线程同时又被中断，并且虚拟机选择让该线程中断，那么等待集合中的另外一个线程必须被唤醒。

11.3.3 线程等待、睡眠和让步

Thread 类的 join 方法提供了一种简单的同步方式，允许当前线程等待另外一个线程运行结束。根据之前对同步关系的介绍，如果线程 A 通过调用线程 B 的 join 方法等待线程 B 运行结束，那么在线程 B 中对共享变量所做的修改对于线程 A 是肯定可见的。一般的做法是，在线程 A 中创建并启动线程 B 之后，线程 A 执行另外的一些操作，接着调用 join 方法等待线程 B 完成。线程 B 和线程 A 通过修改共享变量的方式来进行交互。代码清单 11-8 中给出了 join 方法的一般使用方式。

代码清单 11-8 通过 join 方法等待线程运行结束

```
public void useJoin() {
    Thread thread = new Thread() {
        public void run() {
            try {
                Thread.sleep(5000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    };
    thread.start(); // 启动线程
}
```

```

// 执行其他操作
try {
    thread.join(); // 等待线程运行结束
} catch (InterruptedException e) {
    e.printStackTrace();
}
}

```

Thread 类的静态方法 sleep 可以让当前线程进入睡眠状态一段时间。在睡眠状态下，线程的代码执行会暂停，但是线程不会释放所持有的锁。因此不要把对 sleep 方法的调用放在 synchronized 方法或代码块中，否则会造成其他等待获取锁的线程长时间处于等待状态。

如果当前线程因为某些原因无法继续执行，那么可以使用 yield 方法来尝试让出所占用的 CPU 资源，让其他线程获得运行的机会。调用 yield 方法对操作系统上的调度器来说是一个信号，但是调度器不一定会立即进行线程切换。调用 yield 方法可以使线程切换更加频繁，从而让某些与多线程相关的错误更容易暴露出来。在实际开发中调用 yield 方法可以作为进行测试的一个辅助手段。

11.4 非阻塞方式

线程之间的同步机制的核心是监视器对象上的锁。不同线程之间通过竞争锁的所有权来获得执行某些特定代码的机会。锁机制的目的是保证多线程程序运行时的正确性，但是会对性能造成很大的影响。锁机制在运行时的代价很高，涉及 CPU 和内存方面的很多处理。对于一个声明为 synchronized 的方法，有多个线程竞争对应监视器对象上的锁来获取执行该方法的机会。如果其中某一个线程成功获取了对象上的锁，则其他尝试获取锁的线程会处于等待状态。基于锁机制的实现方式在很大程度上限制了多线程程序的吞吐量和性能，而且会带来死锁和优先级倒置等问题。

死锁问题指的是两个线程分别持有另外一个线程所需要获取的锁，同时又希望获取另外一个线程已经持有的锁。每个线程都由于等待对方释放锁而处于等待状态，因此无法释放自己持有的锁来让对方运行。这样造成的结果是两个线程都无法运行。优先级倒置问题指的是线程的优先级由于锁机制而无法成功应用。有可能因为低优先级的线程持有锁的时间过长，导致高优先级的线程长时间处于等待状态。

锁机制对于多线程程序性能的影响主要来自于其所带来的线程阻塞问题。如果能够不阻塞线程，又能保证多线程程序的正确性，那无疑是更好的一种机制。在程序中，对共享变量的使用一般遵循一定的模式，即由读取、修改和写入三步组成。在读取步骤中读取共享变量的当前值，在修改步骤中根据变量的当前值进行某些修改操作，最后在写入步骤中把修改之后的结果作为共享变量的新值写入。比如代码清单 11-1 中的 getNext 方法对共享变量 value 的使用就遵循这样的模式。先读取 value 的当前值，把该值加上 1

之后作为新值，最后把新值写入 value 中。类似的使用方式还有很多。在这三步执行过程中，随时可能发生线程切换，造成不同线程看到变量 value 的不同值，从而产生错误。锁机制的做法是把这三步变成一个原子操作，从而解决了这个问题。如果读取、修改和写入这三步合起来在 CPU 上形成一个原子操作，那么 CPU 在执行这三步时不会发生线程切换，也就不会造成数据不一致的问题。

目前 CPU 基本都提供了相关的指令来实现读取、修改和写入这三步的原子操作。比较常见的指令名称是“比较和替换（compare and swap，CAS）”。这个指令会先比较某个内存地址的当前值是不是指定的旧值。如果是，就用指定的新值来替换它；否则就什么也不做。指令的返回结果是内存地址的当前值。CAS 指令的这种执行方式的含义在于：如果这个内存地址的当前值还是上次访问时的旧值，就说明从上次访问到这次访问的时间间隔内，没有其他线程对该内存地址进行过修改，可以安全地进行修改操作；否则说明有其他线程进行了修改，当前线程所持有的值已经不再有效，需要使用内存地址的当前值重新进行计算，并在适当的时机再次使用 CAS 指令进行更新。通过 CAS 指令可以实现不依赖锁机制的非阻塞算法。一般的做法是把对 CAS 指令的调用放在一个无限循环中。如果当前循环没能完成修改操作，就不断进行尝试，总会在某个时机上，CAS 指令成功完成修改。

Java 平台利用了 CPU 提供的 CAS 指令来实现非阻塞操作。相关的 API 在 `java.util.concurrent.atomic` 包中。需要注意的是，不是所有的 CPU 都支持 CAS 或类似功能的指令。在某些平台上，`java.util.concurrent.atomic` 包的实现仍然是通过内部的锁机制来实现的。`java.util.concurrent.atomic` 包中提供的 Java 类分成三类，每一类 Java 类提供不同的功能。

第一类是支持以原子操作来进行更新的数据类型的 Java 类，包括 `AtomicBoolean`、`AtomicInteger`、`AtomicLong` 和 `AtomicReference` 类，分别用于对布尔类型、整数、长整数和对象引用进行更新。在内存模型相关的语义上，这 4 个类的对象类似于 `volatile` 变量。这 4 个类所包含的方法不尽相同，但是都与值的获取和设置相关。其中最重要的方法是 `compareAndSet`，用来实现 CAS 指令的语义。调用 `compareAndSet` 方法时需要提供两个参数，第一个是所期望的对象的旧值，第二个是希望设置成的对象的新值。如果对象的当前值与所期望的旧值相同，则把当前值设置为新值。从 `compareAndSet` 方法的返回值可以判断设置是否成功。在内存模型语义上，`compareAndSet` 方法相当于读取 `volatile` 变量的当前值后再写入新值，不过是作为一个原子操作而完成的。与 `compareAndSet` 方法功能相同的是 `weakCompareAndSet` 方法。与 `compareAndSet` 方法相比，`weakCompareAndSet` 方法的性能要好一些，但是所提供的语义比较弱。使用 `weakCompareAndSet` 方法只能保证对当前变量的修改对于其他线程是可见的，但不能保证在调用 `weakCompareAndSet` 方法之前对其他变量的修改也是对于其他线程可见的。这点与 `volatile` 关键词的语义是不同的。所以 `weakCompareAndSet` 方法只适合于用来对值不依赖于其他变量的变量进行修改，如计数器。这 4 个类中的 `get` 和 `set` 方法分别用来直

接获取和设置变量的值，相当于读取和写入 volatile 变量的值。这 4 个类中最后一个相同的方法是 lazySet。这个方法与 set 方法类似，但是允许编译器把对 lazySet 方法的调用与后面的指令进行重排，因此对值的设置操作有可能被推迟。代码清单 11-9 给出了一个使用 AtomicInteger 类实现的线程安全的标识符生成器的示例。AtomicInteger 类提供了 getAndIncrement 方法，相当于代码清单 11-1 中的 value++，不过这个方法是线程安全和非阻塞的。

代码清单 11-9 使用 AtomicInteger 类实现的生成唯一标识符的 Java 类

```
public class AtomicIdGenerator {
    private final AtomicInteger counter = new AtomicInteger(0);
    public int getNext() {
        return counter.getAndIncrement();
    }
}
```

代码清单 11-10 给出了 getAndIncrement 方法的内部实现方式，这也是 compareAndSet 方法的一般使用模式。由于 compareAndSet 方法不一定能够成功完成，因此需要把对 compareAndSet 方法的调用包装在一个无限循环中，不断进行调用，直到 compareAndSet 方法调用成功为止。

代码清单 11-10 AtomicInteger 类的 getAndIncrement 的内部实现

```
public final int getAndIncrement() {
    for (;;) {
        int current = get();
        int next = current + 1;
        if (compareAndSet(current, next))
            return current;
    }
}
```

第二类是提供对数组类型的变量进行处理的 Java 类。把数组对象的引用变量声明为 volatile 只能保证对这个引用变量本身的修改对其他线程是可见的，但是不涉及数组中所包含的元素。AtomicIntegerArray、AtomicLongArray 和 AtomicReferenceArray 类把 volatile 的语义扩展到了数组元素的访问中。这三个类的使用方式类似于对应的操作单个变量的类，只不过在方法调用中多了一个参数来指定要操作的数组元素的序号。

最后一类 Java 类通过反射的方式对任何对象中包含的 volatile 变量使用 compareAndSet 方法进行修改。AtomicIntegerFieldUpdater、AtomicLongFieldUpdater 和 AtomicReferenceFieldUpdater 类分别用来对声明为 volatile 的 int 型、long 型和对象引用类型的变量进行修改。这三个类提供了一种方式把 compareAndSet 方法的功能扩展到在任何 Java 类中声明为 volatile 的域上。这三个类在使用上虽然灵活，但是在原子操作上所提供的语义较弱，因为除了这三个类的对象之外，还可能有其他对象以其他方式

对声明为 volatile 的域进行修改。代码清单 11-11 给出了 AtomicReferenceFieldUpdater 类的使用示例。在对 TreeNode 类中的 volatile 域 parent 进行更新时，通过一个固定的 AtomicReferenceFieldUpdater 类的对象的 compareAndSet 方法来完成。AtomicReferenceFieldUpdater 类提供了一个静态工厂方法 newUpdater，这个静态工厂方法根据包含 volatile 域的 Java 类的类型、域本身的类型和域的名称这 3 个参数来创建用来更新域的值的 AtomicReferenceFieldUpdater 类的对象。AtomicReferenceFieldUpdater 类的对象中包含的方法与对应的 AtomicReference 类是相同的。如果程序中的 Java 类需要用到 compareAndSet 方法提供的语义，那么可以利用这三个类。

代码清单 11-11 更新 volatile 对象引用 AtomicReferenceFieldUpdater 类的使用示例

```
public class TreeNode {
    private volatile TreeNode parent;
    private static final AtomicReferenceFieldUpdater<TreeNode, TreeNode>
        parentUpdater = AtomicReferenceFieldUpdater.newUpdater(TreeNode.class,
        TreeNode.class, "parent");
    public boolean compareAndSetParent(TreeNode expect, TreeNode update) {
        return parentUpdater.compareAndSet(this, expect, update);
    }
}
```

从整体的角度来说，java.util.concurrent.atomic 包中的 Java 类属于比较底层的实现，一般作为 java.util.concurrent 包中很多非阻塞的数据结构的实现基础。实际上使用比较多的主要有 AtomicBoolean、AtomicInteger、AtomicLong 和 AtomicReference 这 4 个类。在实现线程安全的计数器时，AtomicInteger 和 AtomicLong 类是最佳的选择。

11.5 高级实用工具

在 Java 平台出现之后的很长一段时间内，开发多线程程序只能使用 Java 平台提供的 synchronized 和 volatile 关键词，以及 Object 类中的 wait、notify 和 notifyAll 等方法。这些关键词和方法的抽象层次比较低，在程序开发中使用起来比较繁琐，而且容易产生错误。在多线程开发中，线程之间的交互方式存在某些固定的模式，比如常见的生产者 - 消费者和读者 - 写者模式。如果可以把这些模式抽象成高层的 API，那么使用起来会非常方便。J2SE 5.0 引入的 java.util.concurrent 包为多线程程序开发提供了高层的 API，可以满足日常开发中常见的需求。

11.5.1 高级同步机制

虽然非阻塞方式的性能优于阻塞式方式，但是并非所有场景都采用非阻塞式的实现方式，在很多情况下仍然需要使用基于锁机制的阻塞式实现方式。Java 中基本的阻塞式实现方式是基于 synchronized 关键词和 Object 类的 wait、notify 和 notifyAll 方法。通过

`synchronized` 关键词可以获取监视器对象上的锁，不过这种锁的获取和释放都是隐式进行的，由线程在执行 `synchronized` 方法或代码块时自动完成。使用 `synchronized` 关键词的问题在于加锁的范围是固定的，无法把锁在对象之间进行传递，使用起来并不灵活。不过它的好处是使用起来简单，不容易出现错误。如果需要使用更加灵活的锁机制，可以使用 `java.util.concurrent.locks` 包中提供的 API。

`java.util.concurrent.locks` 包中的重要接口之一是 `Lock`。`Lock` 接口表示的是一个锁，可以通过其中的 `lock` 方法来获取锁，而 `unlock` 方法用来释放锁。使用 `Lock` 接口的代码需要保证锁总是被释放。一般把 `unlock` 方法的调用放在 `finally` 代码块中。`Lock` 接口提供了几种不同的获取锁的方式。使用 `lock` 方法获取锁的方式类似于 `synchronized` 关键词。如果调用 `lock` 方法时无法获取锁，那么当前线程会处于等待状态，直到成功获取锁。与 `lock` 方法相似的 `lockInterruptibly` 方法允许当前线程在等待获取锁的过程中被中断。所以调用 `lockInterruptibly` 方法时要处理 `InterruptedException` 异常。除了通过阻塞式的获取锁外，`Lock` 接口也提供了 `tryLock` 方法以非阻塞的方式获取锁。如果在调用 `tryLock` 方法时无法获取锁，那么 `tryLock` 方法只返回 `false`，不会阻塞当前线程。利用 `tryLock` 方法的另外一种重载形式可以指定超时时间。如果指定了超时时间，当无法获取锁时，当前线程会处于阻塞状态，但是等待的时间不会超过指定的超时时间，同时线程也是可以被中断的。

另外一个与锁相关的接口是 `ReadWriteLock`。`ReadWriteLock` 接口实际上表示的是两个锁，一个是读取操作使用的共享锁，另外一个是写入操作使用的排他锁。可以通过 `ReadWriteLock` 接口的 `readLock` 和 `writeLock` 方法来获取表示对应的锁的 `Lock` 接口的实现对象。`ReadWriteLock` 接口适合于解决与常见的读者 - 写者问题类似的场景。在没有线程进行写入操作时，进行读取操作的多个线程都可以获取读取锁；而进行写入操作的线程只有在获取写入锁之后才能进行写入操作。多个线程可以同时进行读取操作，但是同一时刻只允许一个线程进行写入操作。`ReadWriteLock` 接口可以在很多情况下提高多线程程序的性能。在大多数情况下，对一个数据结构的读取操作的次数要远多于写入操作的次数。`ReadWriteLock` 接口允许多个线程同时进行读取操作，这样可以提高使用该数据结构时的吞吐量。如果对数据结构的访问模式不满足这种特性，比如有较多的写入操作，则使用 `ReadWriteLock` 接口会降低性能。

在 `java.util.concurrent.locks` 包中提供了 `Lock` 接口和 `ReadWriteLock` 接口的基本实现，分别是 `ReentrantLock` 类和 `ReentrantReadWriteLock` 类。这两个实现类的共同特征是可重入性，即允许一个线程多次获取同一个锁。`ReentrantLock` 类的对象可以有一个所有者线程，表示上一次成功获取该锁，但还没有释放锁的线程。`ReentrantLock` 类的对象同时保存了所有者线程在该对象上的加锁次数。通过 `getHoldCount` 方法可以获取当前的加锁次数。如果 `ReentrantLock` 类的对象当前没有所有者线程，则当前线程获取锁的操作会成功，加锁次数为 1。在随后的操作中，当前线程可以再次获取该锁，这也是可重入的含义所在。每次加锁操作会使加锁次数增加 1，而每一次调用 `unlock` 方法释放锁

会使加锁次数减 1。当加锁次数变为 0 时，该锁会被释放，可以被其他线程获取。代码清单 11-12 给出了使用 ReentrantLock 类实现的标识符生成器的代码示例。代码中的 getNext 方法也给出了 Lock 接口的基本使用方式。开始时使用 lock 方法来加锁，接着把所要执行的操作放在 try-finally 代码块中，在 finally 代码块中通过 unlock 方法来释放锁。

代码清单 11-12 使用 ReentrantLock 类实现的生成唯一标识符的 Java 类

```
public class LockIdGenerator {
    private final ReentrantLock lock = new ReentrantLock();
    private int value = 0;
    public int getNext() {
        lock.lock();
        try {
            return value++;
        } finally {
            lock.unlock();
        }
    }
}
```

在创建 ReentrantLock 类的对象时，可以通过一个额外的 boolean 类型的参数来声明使用更加公平的加锁机制。在使用锁机制时会遇到的一个问题是线程的饥饿问题。当多个线程同时竞争某个锁时，可能有的线程一直无法成功获取锁，一直处于无法运行的状态。线程饥饿是有些程序应该避免的问题。如果在创建 ReentrantLock 类的对象时添加了额外的参数 true，则 ReentrantLock 类的对象会使用相对公平的锁分配策略。当锁处于可被获取状态时，在由于尝试获取该锁而处于等待状态的线程中，等待时间最长的线程会成功获取这个锁。这就避免了线程的饥饿问题。不过需要注意的是不带参数的 tryLock 方法会忽略公平模式的设置。ReentrantReadWriteLock 类是 ReadWriteLock 接口的可重入的实现类。

可重入锁的优势在于减少了锁在各个线程之间的传递次数，可以提高程序的吞吐量。这里以线程安全的散列表的实现为例来进行说明。在散列表实现中，每个对内部状态进行修改的公开方法都由锁来保护。在方法执行之前都要求线程获取锁，在方法调用完成之后，锁被释放。在一段程序的执行过程中，通常会多次使用同一个散列表对象上的方法。如果在散列表对象中使用的是可重入的锁，则当前线程只有在调用第一个方法时需要与其他线程竞争锁。在成功加锁之后，后续调用的方法可以通过重入的方式来快速获取锁，这就降低了加锁的代价，锁的所有者也没有发生改变。如果锁的实现不是可重入的，那么线程在每次调用方法时都需要与其他线程竞争锁的所有权。如果没能获取锁，则当前线程需要等待，直到再次获取锁。在这种情况下，整段代码执行中的很大一部分时间都花费在加锁与解锁操作上。为了提高程序整体的吞吐量，应该尽可能使用可重入的锁。

Lock 接口替代 synchronized 关键词，相对应的 Condition 接口替代 Object 类的 wait、notify 和 notifyAll 方法。就如同使用 wait、notify 和 notifyAll 方法时不能脱离 synchronized 关键词一样，使用 Condition 接口时也需要与一个对应的 Lock 接口的实现对象关联起来。通过 Lock 接口的 newCondition 方法可以创建新的 Condition 接口的实现对象。在调用 Condition 接口的方法之前，也需要使用 Lock 接口的方法来获取锁。

Condition 接口提供了多个类似 Object 类的 wait 方法的方法，最基本的是 await 方法，调用该方法会使当前线程进入等待状态，直到被唤醒或被中断。另外一种 await 方法的重载形式可以指定超时时间。方法 awaitNanos 以纳秒数为单位指定超时时间，该方法的返回值是剩余等待时间的估计值。类似的 awaitUntil 方法也可以指定超时时间，只不过指定的不是要经过的时间，而是超时发生的时间点，参数是一个 java.util.Date 类的对象。前面几种等待方法都会响应其他线程发出的中断请求，而 awaitUninterruptibly 方法则不会处理中断请求。如果线程通过调用 awaitUninterruptibly 方法进入等待状态，那么，当收到中断请求时，线程仍然会继续处于等待状态，直到被唤醒。当线程从 awaitUninterruptibly 方法返回时，其内部的中断标记会被设置，以表明曾经有中断请求发生。与 Object 类的 wait 方法相同，当线程由于调用 await 等方法进入等待状态时，会释放其持有的锁。

与 Condition 接口中的等待方法相对应的是 signal 和 signalAll 方法，相当于 Object 类中的 notify 和 notifyAll 方法。这两个方法的含义与 notify 和 notifyAll 方法是相同的。代码清单 11-13 给出了 Lock 接口和 Condition 接口的一般使用方式，类似于代码清单 11-7 中对 Object 类的 wait 方法的使用方式。

代码清单 11-13 Lock 接口和 Condition 接口的一般使用方式

```

Lock lock = new ReentrantLock();
Condition condition = lock.newCondition();

lock.lock();
try {
    while /* 逻辑条件不满足 */ {
        condition.await();
    }
} finally {
    lock.unlock();
}

```

11.5.2 底层同步器

在一个多线程程序中，线程之间可能存在多种不同的同步方式。除了 Java 标准库提供的同步方式之外，程序中特有的同步方式需要由开发人员自己来实现。在这些同步方式中，一类比较常见的需求是对有限个共享资源的同步访问，多线程程序中的很多场景

都可以抽象成这类同步方式。比如对某个监视器对象的互斥访问，实际上是多个线程在竞争唯一的一个资源。如果系统中安装了两个打印机，那么需要进行打印操作的多个线程相互竞争这两个资源。当多个线程在等待同一个资源时，从公平的角度出发，这些线程会被放入到一个先入先出（FIFO）的队列中。当资源变成可用时，处于队首的线程会获取该资源。

如果程序中的同步方式可以抽象成对有限个资源的同步访问，那么可以使用 `java.util.concurrent.locks` 包 中 的 `AbstractQueuedSynchronizer` 类 和 `AbstractQueuedLongSynchronizer` 类作为实现的基础。这两个类的作用是相同的，只不过 `AbstractQueuedSynchronizer` 类在内部使用一个 `int` 类型的变量来维护内部状态，而 `AbstractQueuedLongSynchronizer` 类使用 `long` 类型的变量。可以将这个内部变量理解成共享资源的个数。通过 `getState`、`setState` 和 `compareAndSetState` 这 3 个方法来更新这个内部变量的值。在下面的介绍中使用 `AbstractQueuedSynchronizer` 类来说明。

`AbstractQueuedSynchronizer` 类是声明为 `abstract` 的，因此需要继承并覆写其中包含的部分方法后才能使用。通常的做法是把 `AbstractQueuedSynchronizer` 类的子类作为一个 Java 类的内部类。外部的 Java 类提供具体的同步方式，而 `AbstractQueuedSynchronizer` 类的对象则作为实现的基础。实际使用 `AbstractQueuedSynchronizer` 类需要经过几个步骤。首先确定如何把需要同步访问的资源映射到 `AbstractQueuedSynchronizer` 类的内部变量上。一般用内部变量的值表示当前可用资源的数量。接着选择使用排他模式或共享模式。共享模式允许多个线程同时获取所管理的资源，而排他模式在同一时刻只允许一个线程获取资源。选择好模式之后，继承自 `AbstractQueuedSynchronizer` 类的子类需要实现对应的资源获取和释放方法。使用排他模式时需要实现的方法是 `tryAcquire` 和 `tryRelease`，而使用共享模式时对应的方法是 `tryAcquireShared` 和 `tryReleaseShared`。在这些方法的实现中，使用 `getState`、`setState` 和 `compareAndSetState` 这 3 个方法来修改内部变量的值，以此来反映资源的状态。在对资源进行同步访问的 Java 类中需要创建一个 `AbstractQueuedSynchronizer` 类的子类的对象，并通过该对象来完成实际的资源获取和释放的同步操作。

代码清单 11-14 给出了一个基于 `AbstractQueuedSynchronizer` 类实现的对资源进行管理的 `SimpleResourceManager` 类。内部类 `InnerSynchronizer` 继承自 `AbstractQueuedSynchronizer` 类并覆写了 `tryAcquireShared` 和 `tryReleaseShared` 方法，说明对资源的访问使用的是共享模式，直接用内部变量的值表示资源的当前可用数量。在这两个方法的实现中，通过 `getState` 方法获取当前的状态值，并根据需要使用 `compareAndSetState` 方法来更新获取和释放操作之后的新状态值。在外部类中对资源进行访问的方法只是简单地调用 `InnerSynchronizer` 类的对象上的适当方法来完成的。

代码清单 11-14 基于 `AbstractQueuedSynchronizer` 类的资源管理实现

```
public class SimpleResourceManager {
```

```

private final InnerSynchronizer synchronizer;
private static class InnerSynchronizer extends AbstractQueuedSynchronizer {
    InnerSynchronizer(int numOfResources) {
        setState(numOfResources);
    }
    protected int tryAcquireShared(int acquires) {
        for (;;) {
            int available = getState();
            int remaining = available - acquires;
            if (remaining < 0 ||
                compareAndSetState(available, remaining)) {
                return remaining;
            }
        }
    }
    protected boolean tryReleaseShared(int releases) {
        for (;;) {
            int available = getState();
            int next = available + releases;
            if (compareAndSetState(available, next)) {
                return true;
            }
        }
    }
    public SimpleResourceManager(int numOfResources) {
        synchronizer = new InnerSynchronizer(numOfResources);
    }
    public void acquire() throws InterruptedException {
        synchronizer.acquireSharedInterruptibly(1);
    }
    public void release() {
        synchronizer.releaseShared(1);
    }
}

```

11.5.3 高级同步对象

使用 `java.util.concurrent.atomic` 包和 `java.util.concurrent.locks` 包提供的 Java 类可以满足基本的互斥和同步访问的需求，但是这些 Java 类的抽象层次较低，使用起来比较复杂。对于一般的程序来说，更简单的做法是使用 `java.util.concurrent` 包中的高级同步对象。

本节主要介绍的是 `java.util.concurrent` 包提供的满足常见需求的高级同步对象，包括进行资源管理的信号量、协调不同线程运行顺序的倒数闸门和循环屏障，以及进行数据交换的对象交换器等。如果程序中的多个线程之间的交互方式满足这些高级同步对象所适用的场景，那么使用这些同步对象可以大大提高开发效率。

1. 信号量

对于信号量的概念，开发人员可能并不陌生。信号量在操作系统中一般用来管理数量有限的资源。每类资源有一个对应的信号量。信号量的值表示资源的可用数量。在使用资源时，要先从该信号量上获取一个使用许可。成功获取许可之后，资源的可用数减 1。在持有许可期间，使用者可以对获取的资源进行操作。完成对资源的使用之后，需要在信号量上释放一个使用许可，资源的可用数加 1，允许其他使用者获取资源。当资源可用数为 0 的时候，需要获取资源的线程以阻塞的方式来等待资源变为可用，或者过段时间之后再检查资源是否变为可用。代码清单 11-14 中的 SimpleResourceManager 类实际上是信号量的一个简单实现。如果需要在程序中使用信号量，更好的方式是直接使用 `java.util.concurrent.Semaphore` 类。在创建 `Semaphore` 类的对象时指定资源的可用数，通过 `acquire` 方法以阻塞式的方式获取许可，而 `tryAcquire` 方法以非阻塞式的方式来获取许可。当需要释放许可时，使用 `release` 方法。`Semaphore` 类也支持同时获取和释放多个资源的许可。通过 `acquire` 方法获取许可的过程是可以被中断的。如果不希望被中断，那么可以使用 `acquireUninterruptibly` 方法。

代码清单 11-15 给出了使用 `Semaphore` 类管理程序中可用的打印机的示例。在创建 `PrinterManager` 类的对象时，需要指定程序中所有可用 `Printer` 类的对象的集合。根据可用的 `Printer` 类的对象的数目创建出 `Semaphore` 类的对象。`Semaphore` 类也支持在分配许可时使用公平模式，通过把构造方法的第二个参数值设为 `true` 来使用该模式。在公平模式下，当资源可用时，等待线程按照调用 `acquire` 方法申请资源的顺序依次获取许可。在进行资源管理时，一般使用公平模式，以避免造成线程饥饿问题。需要注意的是访问内部变量 `printers` 的方法都通过 `synchronized` 关键词声明为同步的。这是因为 `Semaphore` 类只是一个资源数量的抽象表示，并不负责管理资源对象本身，可能有多个线程同时获取到资源使用许可，因此需要使用同步机制避免数据竞争。

代码清单 11-15 使用信号量管理资源的示例

```

public class PrinterManager {
    private final Semaphore semaphore;
    private final List<Printer> printers = new ArrayList<>();
    public PrinterManager(Collection<? extends Printer> printers) {
        this.printers.addAll(printers);
        this.semaphore = new Semaphore(this.printers.size(), true);
    }
    public Printer acquirePrinter() throws InterruptedException {
        semaphore.acquire();
        return getAvailablePrinter();
    }
    public void releasePrinter(Printer printer) {
        putBackPrinter(printer);
        semaphore.release();
    }
}

```

```

private synchronized Printer getAvailablePrinter() {
    Printer result = printers.get(0);
    printers.remove(0);
    return result;
}
private synchronized void putBackPrinter(Printer printer) {
    printers.add(printer);
}
}

```

2. 倒数闸门

在多个线程进行协作时，一个常见的情景是一个线程需要等待另外的线程完成某些任务之后才能继续进行。在这种情况下，可以使用 `java.util.concurrent.CountDownLatch` 类。`CountDownLatch` 类相当于多个线程等待开启的一个闸门。只有在其他线程完成任务之后，闸门才会打开，等待的线程才能运行。在创建 `CountDownLatch` 类的对象时需要指定等待完成的任务数目。一个 `CountDownLatch` 类的对象被执行任务的线程和等待任务完成的线程所共享。当执行任务的线程完成其任务时，调用 `countDown` 方法来使得待完成的任务数量减 1。等待任务完成的线程通过调用 `await` 方法进入阻塞状态直到待完成的任务数量变为 0。当所有任务都完成时，等待任务完成的线程会从 `await` 方法返回，可以继续执行后续的代码。`CountDownLatch` 类的对象的使用是一次性的。一旦待完成的任务数量变为 0，再调用 `await` 方法就不再阻塞当前线程，而是立即返回。

`CountDownLatch` 类在很多场景下都可以得到应用。比如在使用多线程方式下载一个文件时，可以有一个主线程和若干个下载线程。在某个下载线程完成部分内容的下载任务时，调用 `countDown` 方法来进行声明，主线程则调用 `await` 方法等待所有部分下载完成。代码清单 11-16 给出了使用 `CountDownLatch` 类的一个示例。每个线程负责获取一个网页的内容并计算其内容的大小。当所有网页的大小都计算完成之后，由主线程对网页内容大小进行排序。在 `GetSizeWorker` 类的 `run` 方法中，当任务完成之后，通过 `countDown` 方法发出通知；在主线程中通过 `await` 方法来等待所有使用 `GetSizeWorker` 类的线程完成运行。

代码清单 11-16 倒数闸门的使用示例

```

public class PageSizeSorter {
    private static final ConcurrentHashMap<String, Integer> sizeMap = new
        ConcurrentHashMap<>();
    private static class GetSizeWorker implements Runnable {
        private final String urlString;
        private final CountDownLatch signal;
        public GetSizeWorker(String urlString, CountDownLatch signal) {
            this.urlString = urlString;
            this.signal = signal;
        }
    }
}

```

```

public void run() {
    try {
        InputStream is = new URL(urlString).openStream();
        int size = IOUtils.toByteArray(is).length;
        sizeMap.put(urlString, size);
    } catch (IOException e) {
        sizeMap.put(urlString, -1);
    } finally {
        signal.countDown();
    }
}
private void sort() {
    List<Entry<String, Integer>> list = new ArrayList<>(sizeMap.entrySet());
    Collections.sort(list, new Comparator<Entry<String, Integer>>() {
        public int compare(Entry<String, Integer> o1,
                           Entry<String, Integer> o2) {
            return Integer.compare(o2.getValue(), o1.getValue());
        }
    });
    System.out.println(Arrays.deepToString(list.toArray()));
}
public void sortPageSize(Collection<String> urls) throws InterruptedException {
    CountDownLatch sortSignal = new CountDownLatch(urls.size());
    for (String url : urls) {
        new Thread(new GetSizeWorker(url, sortSignal)).start();
    }
    sortSignal.await();
    sort();
}
}

```

3. 循环屏障

循环屏障在作用上类似于倒数闸门，不过有几个显著的不同点。首先循环屏障不像倒数闸门一样是一次性的，可以循环使用。其次使用循环屏障的线程之间是互相平等的，彼此都需要等待对方完成。当一个线程完成自己的任务之后，等待其他线程完成。当所有线程都完成任务之后，所有线程才可以继续运行。当线程之间需要再次进行互相等待时，可以复用同一个循环屏障。

类 `java.util.concurrent.CyclicBarrier` 用来表示循环屏障。`CyclicBarrier` 类的对象在创建时需要指定使用该对象的线程数目，还可以指定一个 `Runnable` 接口的实现对象作为每次所有线程之间完成相互等待之后执行的动作。当最后一个线程完成任务之后，在所有的线程继续执行之前，这个 `Runnable` 接口的实现对象会被运行。如果这些线程之间需要更新一些共享的内部状态，可以利用这个 `Runnable` 接口的实现对象来进行处理。每个线程在完成任务之后，通过调用 `await` 方法进入等待状态。等所有线程都调用了 `await` 方法

之后，处于等待状态的线程都可以继续执行。在所有参与线程中，只要有一个在等待过程中被中断、出现超时或是其他错误，整个循环屏障会失效。所有处于等待状态的其他线程会抛出 `java.util.concurrent.BrokenBarrierException` 异常而结束。

代码清单 11-17 中给出了使用多个线程来查找质数的示例。每个线程负责查找给定数字区间范围内的质数。线程之间使用 `CyclicBarrier` 类的对象进行同步。当所有线程都完成一轮计算之后，会调用创建 `CyclicBarrier` 类的对象时提供的 `Runnable` 接口的实现。在这个 `Runnable` 接口实现中，检查当前已经找到的质数数目是否足够。如果已经足够，则设置标记变量 `done` 的值为 `true`；否则，所有线程继续运行，在下一个区间中进行查找。这里需要注意的是，如果在调用 `await` 方法中出现异常，所有的线程都会结束。因此需要正确设置 `done` 的值，使得主线程在计算线程出错时也能正确处理。

代码清单 11-17 使用多个线程来查找质数的示例

```

public class PrimeNumber {
    private static final int TOTAL_COUNT = 5000;
    private static final int RANGE_LENGTH = 200;
    private static final int WORKER_NUMBER = 5;
    private static volatile boolean done = false;
    private static int rangeCount = 0;
    private static final List<Long> results = new ArrayList<Long>();
    private static final CyclicBarrier barrier = new CyclicBarrier(WORKER_NUMBER,
        new Runnable() {
            public void run() {
                if (results.size() >= TOTAL_COUNT) {
                    done = true;
                }
            }
        });
    private static class PrimeFinder implements Runnable {
        public void run() {
            while (!done) {
                int range = getNextRange();
                long start = range * RANGE_LENGTH;
                long end = (range + 1) * RANGE_LENGTH;
                for (long i = start; i < end; i++) {
                    if (isPrime(i)) {
                        updateResult(i);
                    }
                }
                try {
                    barrier.await();
                } catch (InterruptedException | BrokenBarrierException e) {
                    done = true;
                }
            }
        }
    }
}

```

```

private synchronized static void updateResult(long value) {
    results.add(value);
}
private synchronized static int getNextRange() {
    return rangeCount++;
}
private static boolean isPrime(long number) {
    // 省略判断是否为质数的代码
}
public void calculate() {
    for (int i = 0; i < WORKER_NUMBER; i++) {
        new Thread(new PrimeFinder()).start();
    }
    while (!done) {
    }
    // 计算完成
}
}

```

4. 对象交换器

对象交换器适合于两个线程需要进行数据交换的场景。在某些情况下，两个线程对于共享的对象有不同的处理逻辑。在两个线程都完成处理之后，处理的结果对象被交换给另外一个线程，由另外一个线程继续进行处理。类 `java.util.concurrent.Exchanger` 的作用是提供对这种对象交换能力的支持。两个线程共享一个 `Exchanger` 类的对象。一个线程完成对数据的处理之后，调用 `Exchanger` 类的 `exchange` 方法把处理之后的数据作为参数发送给另外一个线程。而 `exchange` 方法的返回结果是另外一个线程所提供的相同类型的对象。如果另外一个线程尚未完成对数据的处理，那么 `exchange` 方法会使当前线程进入等待状态，直到另外一个线程也调用了 `exchange` 方法来进行数据交换。

代码清单 11-18 给出了 `Exchanger` 类的使用示例。`Sender` 和 `Receiver` 类分别负责准备各自的数据。任何一方完成准备之后，调用 `exchange` 方法来启动交换。等两者都完成任务之后，`exchange` 方法会返回，两个线程得到了对方提供的对象。

代码清单 11-18 对象交换器的使用示例

```

public class SendAndReceiver {
    private final Exchanger<StringBuilder> exchanger = new Exchanger<StringBuilder>();
    private class Sender implements Runnable {
        public void run() {
            try {
                StringBuilder content = new StringBuilder("Hello");
                content = exchanger.exchange(content);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }
}

```

```

        }
    }

    private class Receiver implements Runnable {
        public void run() {
            try {
                StringBuilder content = new StringBuilder("World");
                content = exchanger.exchange(content);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }

    public void exchange() {
        new Thread(new Sender()).start();
        new Thread(new Receiver()).start();
    }
}

```

11.5.4 数据结构

java.util.concurrent 包中也提供了一些适合多线程程序使用的高性能数据结构，包括队列和集合类对象等。

1. 队列

队列是多线程程序中比较常用的一种数据结构。多个线程对同一个队列进行操作，通过队列来进行数据传递。java.util.concurrent 包中的 BlockingQueue 接口表示的是线程安全的阻塞式队列。BlockingQueue 接口本身封装了生产者 - 消费者场景所需的语义。当队列已满时，向队列中添加数据的方法会阻塞当前线程；当队列已空时，从队列中获取数据的方法会阻塞当前线程。使用 BlockingQueue 接口可以非常容易地实现生产者 - 消费者场景，只需要让生产者和消费者线程共享一个 BlockingQueue 接口的实现对象，并分别调用不同的方法即可。线程之间的具体同步机制由 BlockingQueue 接口的实现类负责处理。BlockingQueue 接口中的方法支持阻塞式和非阻塞式两种方式。阻塞式的方式通过 put 方法向队列添加数据，通过 take 方法从队列中获取数据。非阻塞方式则分别通过 offer 和 poll 方法来完成。Java 标准库中提供的 BlockingQueue 接口的实现包括基于数组的固定元素个数的 ArrayBlockingQueue 类和基于链表结构的不固定元素个数的 LinkedBlockingQueue 类。开发人员可以根据需要选择合适的实现类。

与 BlockingQueue 接口功能相似的是 BlockingDeque 接口，不过 BlockingDeque 接口表示的是可以对头尾都进行添加和删除操作的双向队列。BlockingDeque 接口中的方法分成两类，分别在队首和队尾进行操作。标准库中只提供了 BlockingDeque 接口的一种基于链表的实现——LinkedBlockingDeque 类。

BlockingQueue 和 BlockingDeque 接口的实现类都是阻塞式队列。如果队列中所包含的元素数量没有限制，可以使用 ConcurrentLinkedQueue 和 ConcurrentLinkedDeque 类。这两个类在实现中使用了非阻塞式算法，避免了使用锁机制，性能比较好。

2. 集合类

另外一部分实用的 Java 类是线程安全的集合类。在多线程程序中，如果共享变量是集合类的对象，则不适合直接使用 java.util 包中已有的集合类。这些集合类有些不是线程安全的，有些在多线程环境下性能比较差。更好的选择是使用 java.util.concurrent 包中的集合类。

当需要使用散列表时，可以使用 ConcurrentHashMap 接口的实现类。ConcurrentMap 接口继承自 java.util.Map 接口，增添了一个方法。这几个方法虽然包含了条件判断，但都是原子操作。调用者不需要考虑同步的问题。其中 putIfAbsent 方法只有在散列表中不包含给定的键时，才会把给定的值放入散列表中； remove 方法在散列表中包含给定键和值的条目时，删除该条目； replace 方法有两种重载形式，第一种形式是在散列表中包含给定的键时，用指定的新值替换原来的值，另外一种形式是在调用时需要指定键、期望的旧值和要设置的新值。当散列表中给定键对应的值与期望的旧值相等时，用给定的新值进行替换。这种形式的 replace 方法的含义相当于 compareAndSet 方法。

ConcurrentMap 接口的基本实现是 ConcurrentHashMap 类。在创建 ConcurrentHashMap 类的对象时，一般使用默认的不带任何参数的构造方法。不过考虑到 ConcurrentHashMap 在多线程程序中的性能问题，如果可以预先估算一些数值，那么可以提高 ConcurrentHashMap 的使用性能。第一个要考虑的因素是 ConcurrentHashMap 类的对象中可能包含的条目个数。因为在 ConcurrentHashMap 类的实现中动态调整所能包含的条目数量的操作比较耗时，如果可以事先指定一个相对合理的初始大小，那么可以避免不必要的调整大小的操作。第二个要考虑的因素是同时进行更新操作的线程数。ConcurrentHashMap 类在实现中会根据这个估计的线程数把内部的空间划分成对应数量的部分。从理论上来说，这些线程可以分别在不同的部分同时进行更新操作，而不会互相冲突。如果设置的线程数过大或过小，会对性能造成一定的影响。如果不显式指定，那么使用的默认值是 16。这个默认值对某些场景来说是不合适的。比如，在很多情况下，只有一个线程进行更新操作，其他线程只进行读取操作，这时把值设为 1 可以提高性能。

需要注意的是，ConcurrentHashMap 类的对象在使用时，读取操作和更新操作可能会重叠在一起。比如，putAll 方法会把一组条目添加到散列表中。在完成对某一个条目的添加之后，并发进行的读取操作就可以获取新添加的条目，此时 putAll 方法可能还在进行其他条目的添加操作。通过 ConcurrentHashMap 接口的方法可以获取散列表中包含的键和值的集合。当从这些集合中创建出迭代器对象之后，迭代器对象只与集合保持弱一致性。通过迭代器可以访问在其创建时集合中包含的元素，但是不一定可以

反映出迭代器创建之后散列表所发生的变化。在使用迭代器的方法时不会抛出 `java.util.ConcurrentModificationException` 异常。

如果需要在多线程程序中使用 `java.util.List` 接口的实现类，可以使用 `CopyOnWriteArrayList` 类。在 `CopyOnWriteArrayList` 类的实现中，所有对列表的更新操作都会重新创建一个底层数组的副本，并使用这个副本存储数据。对列表的更新操作是加锁的，而读取操作是不加锁的。通过复制的方式避免了可能产生的数据竞争，但是带来了额外的开销。一般对 `List` 接口的读取操作要多于更新操作，因此采用这种方式是比较合理的。如果在使用中发现对 `List` 接口实现对象的更新操作相对较多，则不适合使用 `CopyOnWriteArrayList` 类。通过 `CopyOnWriteArrayList` 类的 `iterator` 方法得到的迭代器对象只反映迭代器创建时列表的状态。如果在创建迭代器之后，对 `CopyOnWriteArrayList` 类的对象进行了更新操作，在更新之后，`CopyOnWriteArrayList` 类的对象使用了新的底层数组，而之前创建的迭代器仍然引用的是旧的底层数组。

11.5.5 任务执行

线程最基本的用法是在程序的主线程之外执行其他的任务。最简单的做法是创建一个表示线程的 `Thread` 类的对象，再调用 `start` 方法启动该线程的运行。这种做法虽然简单，但要求开发人员对创建出来的线程进行维护，当线程较多时，会带来不小的维护成本。在线程较多时，一般的做法是创建一个线程池来进行统一管理，同时降低重复创建线程的开销。在 Java 早期版本中，开发人员需要自己开发线程池的实现，或者利用第三方库。在 J2SE 5.0 中，Java 标准库的 `java.util.concurrent` 包提供了丰富的用来管理线程和执行任务的实现。

首先介绍在执行任务时会用到的几个比较重要的接口。在 J2SE 5.0 之前，`Runnable` 接口用来表示一个可执行的任务。不过 `Runnable` 接口有一些局限性，主要是受限于 `run` 方法的类型签名。这些局限性通过 `Callable` 接口来解决。`Callable` 接口只有一个方法 `call`。这个 `call` 方法可以有返回值，同时可以抛出受检异常。这两点是 `Callable` 接口相对于 `Runnable` 接口的优势。不过 `Callable` 接口的实现对象不能通过直接创建 `Thread` 类的对象的方式来执行，而需要使用 `java.util.concurrent` 包提供的任务执行 API。

当需要异步执行一个任务时，一般的做法是把任务的执行封装在一个线程中。如果需要获取任务的执行结果，则要求在主线程和任务执行线程之间进行同步和数据传递。`Future` 接口简化了任务的异步执行。`Future` 接口可以作为异步操作的一个抽象。调用 `Future` 接口的 `get` 方法可以获取异步任务的执行结果。如果任务没有执行完，那么调用 `get` 方法的线程会处于等待状态，直到任务完成或被取消。如果希望取消一个任务的执行，那么可以调用 `cancel` 方法。

有些任务对执行时的调度方式有一定的要求，这些任务不在提交之后就立即执行，而是需要等待一段时间。`Delayed` 接口用来声明任务在调度方式上的这种需求。`Delayed` 接口中的 `getDelay` 方法用来返回当前剩余的延迟时间。当 `getDelay` 方法的返回值不大于

0 时，说明所延迟的时间已经过去，应该调度并执行该任务。

把 Runnable、Future 和 Delayed 接口组合起来，可以形成具备组合功能的新接口。RunnableFuture 接口继承自 Runnable 接口和 Future 接口。当来自 Runnable 接口中的 run 方法成功执行之后，相当于 Future 接口表示的异步任务已经成功完成，可以通过 get 方法来获取运行的结果。ScheduledFuture 接口继承自 Delayed 接口和 Future 接口，表示一个可以进行调度的异步操作。RunnableScheduledFuture 接口则同时继承自 Runnable、Delayed 和 Future 接口。RunnableScheduledFuture 接口的实现对象是可调度的，同时通过 run 方法的成功运行来表示异步操作的完成。RunnableScheduledFuture 接口中包含的方法 isPeriodic 用来表明该异步操作是否可以被重复调度执行。

上面介绍的接口都是用来描述任务本身的，与任务的执行没有关系。在执行任务时，可以手动创建 Thread 类的对象，不过更好的做法是使用 java.util.concurrent 包中提供的与任务执行相关的接口和实现。最基本的接口是 Executor，其中的 execute 方法用来执行一个 Runnable 接口的实现对象。不同的 Executor 接口实现可能采取不同的任务执行策略。Executor 接口所提供的任务执行功能比较弱，只能处理 Runnable 接口。ExecutorService 接口继承自 Executor 接口，并提供了更多实用的功能，其中，第一项重要功能是对任务的管理。使用 ExecutorService 接口的 submit 方法可以把 Callable 接口和 Runnable 接口的实现对象作为任务来提交，得到一个 Future 接口的实现对象作为返回值。通过该 Future 接口的实现对象可以获取任务的执行结果或取消任务。第二项功能是任务的批量执行。通过 invokeAll 和 invokeAny 方法可以同时提交多个 Callable 接口的实现对象。调用 invokeAll 方法之后，会等待所有的任务都执行完成，返回值是一个包含每个任务对应的 Future 接口实现对象的列表，从中可以获取每个任务的运行结果。调用 invokeAny 方法之后，任何一个任务成功完成后，都会把该任务的执行结果返回给调用者。最后一项功能是任务执行服务的关闭。当不再需要使用 ExecutorService 接口的实现对象时，可以调用 shutdown 或 shutdownNow 方法来关闭服务。两者的区别在于，shutdown 方法只是不再允许新的任务被提交，在 shutdown 方法被调用之前提交的任务仍然可以继续运行；而 shutdownNow 方法会试图终止正在运行的和等待运行的任务，并返回已经提交但没有被运行的任务的列表。这两个方法都不会等待服务真正关闭，只是发出关闭请求。通过调用 ExecutorService 接口的 awaitTermination 方法可以使当前线程在一定时间内等待服务完成关闭。在 shutdownNow 方法中强制终止任务时，通常的做法是向线程发出中断请求，因此确保提交的任务实现了正确的中断处理逻辑，能够在收到中断请求时进行必要的清理工作并结束任务。

如果需要对任务的执行方式进行调度，可以使用继承自 ExecutorService 接口的 ScheduledExecutorService 接口。ScheduledExecutorService 接口支持任务的延迟执行和定期执行。可以执行的任务由 Callable 或 Runnable 接口来表示。通过 schedule 方法可以调度一个任务在延迟若干时间之后再执行；而 scheduleAtFixedRate 方法则调度一个任务在初始的延迟时间后，每隔一段时间重复执行，在下一次执行开始时，上一次执行可能

还没有结束；`scheduleWithFixedDelay` 方法与 `scheduleAtFixedRate` 方法的作用类似，只不过 `scheduleWithFixedDelay` 方法是在上一次任务执行完成之后，经过给定的间隔时间再开始下一次的执行。所以 `scheduleAtFixedRate` 方法可能造成任务执行时的重叠，而 `scheduleWithFixedDelay` 方法则不会。这三个方法的返回值都是 `ScheduledFuture` 接口的实现对象。

在 `java.util.concurrent` 包中提供了这些任务执行接口的默认实现。在大多数情况下，使用默认实现已经足够好。这些默认实现也提供了比较多的配置项，允许开发人员进行自定义。通过 `Executors` 类的静态工厂方法可以创建出 `ExecutorService` 和 `ScheduledExecutorService` 接口的实现对象，比如调用 `newFixedThreadPool` 方法可以创建出一个使用固定数量的线程池来执行任务的 `ExecutorService` 接口的实现对象。

代码清单 11-19 给出了一个利用 `ExecutorService` 接口的使用多线程方式下载文件的 Java 类 `FileDownloader`。`ExecutorService` 接口的实现对象使用了 10 个线程来处理下载请求。在进行文件下载时，通过 `submit` 方法提交一个新的 `Callable` 接口的实现对象到 `ExecutorService` 中。当不再使用 `FileDownloader` 类的对象时，应该使用 `close` 方法来关闭其中包含的 `ExecutorService` 接口的实现对象，否则，虚拟机不会退出，所占用的内存也不会释放。在 `close` 方法的实现中，首先使用 `shutdown` 方法来发出关闭请求，此时新的任务不会被接受。接着使用 `awaitTermination` 方法来等待一段时间，使正在执行和等待执行的任务有机会可以完成。如果这些任务超过给定的时间仍没有完成，那么使用 `shutdownNow` 方法来强制结束。在调用 `shutdownNow` 方法之后，再使用 `awaitTermination` 方法来等待另外一段时间，使被强制结束的任务可以完成必要的清理工作。

代码清单 11-19 `ExecutorService` 接口的使用示例

```
public class FileDownloader {
    private final ExecutorService executor = Executors.newFixedThreadPool(10);
    public boolean download(final URL url, final Path path) {
        Future<Path> future = executor.submit(new Callable<Path>() {
            public Path call() {
                try {
                    InputStream is = url.openStream();
                    Files.copy(is, path, StandardCopyOption.REPLACE_EXISTING);
                    return path;
                } catch (IOException e) {
                    return null;
                }
            }
        });
        try {
            return future.get() != null ? true : false;
        } catch (InterruptedException | ExecutionException e) {
            return false;
        }
    }
}
```

```

        }
    }

    public void close() {
        executor.shutdown();
        try {
            if (!executor.awaitTermination(3, TimeUnit.MINUTES)) {
                executor.shutdownNow();
                executor.awaitTermination(1, TimeUnit.MINUTES);
            }
        } catch (InterruptedException e) {
            executor.shutdownNow();
            Thread.currentThread().interrupt();
        }
    }
}

```

在使用 ExecutorService 接口时，通过 submit 方法提交任务，并得到一个 Future 接口的实现对象来获取任务的执行结果。在有些情况下，任务的提交者和任务执行结果的使用者是程序中的不同部分。如果使用 ExecutorService 接口，就需要把得到的 Future 接口的对象在不同部分之间进行传递。更好的做法是使用 CompletionService 接口。程序中的不同部分可以共享 CompletionService 接口的实现对象。任务的提交者通过 submit 方法提交表示任务的 Runnable 和 Callable 接口的实现对象，而任务执行结果的使用者可以通过 take 或 poll 方法获取表示执行结果的 Future 接口的实现对象。两个方法的区别在于 take 是阻塞式的，而 poll 是非阻塞式的。ExecutorCompletionService 类是 Java 标准库提供的 CompletionService 接口的实现类。在创建 ExecutorCompletionService 类的对象时需要提供一个 Executor 接口的实现对象作为参数，用来进行实际的任务执行。

11.6 Java SE 7 新特性

Java SE 7 对 java.util.concurrent 包进行了更新，增加了新的轻量级任务执行框架 fork/join 和多阶段线程同步工具。

11.6.1 轻量级任务执行框架 fork/join

Java SE 7 对 java.util.concurrent 包的一个重要更新是增加了一个新的轻量级任务执行框架，一般称为 fork/join 框架。这个框架的目的主要是更好地利用底层平台上的多核 CPU 和多处理器来进行并行处理，解决问题时通常使用分治（divide and conquer）算法或 map/reduce 算法来进行。这个框架的名称来源于使用时的两个基本操作 fork 和 join，可以类比于 map/reduce 中的 map 和 reduce 操作。fork 操作的作用是把一个大的问题划分成若干个较小的问题。这个划分过程一般是递归进行的，直到得到可以直接进行计算的粒度合适的子问题。在划分时，需要恰当地选取子问题的大小。太大的子问题不利于通过并行方式来提高性能，而太小的子问题则会带来较大的额外开销。每个子问题在计

算完成之后，可以得到关于整个问题的部分解。join 操作的作用是把这些部分解收集并组织起来，得到最终的完整解。调用 join 操作的过程也可能是递归进行的，与 fork 操作相对应。

相对于一般的线程池实现，fork/join 框架的优势体现在对其中包含的任务的处理方式上。在一般的线程池中，如果一个线程正在执行的任务由于某些原因无法继续运行，那么该线程会处于等待状态。而在 fork/join 框架实现中，如果某个子问题由于等待另外一个子问题的完成而无法继续运行，那么处理该子问题的线程会主动寻找其他尚未运行的子问题来执行。这种方式减少了线程的等待时间，提高了性能。为了 fork/join 框架能够高效运行，在每个子问题的实现中应该避免使用 synchronized 关键词或其他方式进行同步，也不应该使用阻塞式 I/O 操作或过多地访问共享变量。在理想情况下，每个子问题的实现中都应该只进行 CPU 相关的计算，并且只使用每个问题的内部对象。唯一的同步应该只发生在子问题和创建它的父问题之间。

一个由 fork/join 框架执行的任务由 ForkJoinTask 类表示。ForkJoinTask 类实现了 Future 接口，可以按照 Future 接口的方式来使用。在 ForkJoinTask 类中最重要的两个方法是 fork 和 join，其中 fork 方法用来以异步方式启动任务的执行，而 join 方法则等待任务完成并返回执行的结果。在创建自己的任务时，最好不要直接继承自 ForkJoinTask 类，而要继承自 ForkJoinTask 类的子类 RecursiveTask 或 RecursiveAction 类。两者的区别在于 RecursiveTask 类表示的任务可以返回结果，而 RecursiveAction 类不行。

在 fork/join 框架中，任务的执行由 ForkJoinPool 类的对象来完成。ForkJoinPool 类实现了 ExecutorService 接口，除了执行 ForkJoinTask 类的对象之外，还可以使用一般的 Callable 和 Runnable 接口来表示任务。在 ForkJoinPool 类的对象中执行的任务大致可以分成两类：一类是通过 execute、invoke 或 submit 方法直接提交的任务；另外一类是 ForkJoinTask 类的对象在执行过程中产生的子任务，并通过 fork 方法来运行。一般的做法是表示整个问题的 ForkJoinTask 类的对象用第一类形式提交，而在执行过程中产生的子任务并不需要进行处理，ForkJoinPool 类的对象会负责子任务的执行。

代码清单 11-20 给出了使用 fork/join 框架查找数组中的最大值的示例。类 MaxValueTask 表示的是进行查找的任务，继承自 RecursiveTask 类。每个查找任务都在一定的数组序号区间范围内进行。如果当前区间范围较大，则将其划分成两个子区间，对每个子区间创建子任务来进行查找，递归进行这个过程，直到范围区间足够小，再在这个区间中进行顺序比较来查找最大值。通过 fork 方法来启动子任务，join 方法用来获取子任务的执行结果。在子任务执行完成之后，再把得到的部分结果合并到最终的结果中。这是一种典型的使用分治算法的实现方式。

代码清单 11-20 fork/join 框架的使用示例

```
public class MaxValue {
    private static final int RANGE_LENGTH = 2000;
    private final ForkJoinPool forkJoinPool = new ForkJoinPool();
```

```

private static class MaxValueTask extends RecursiveTask<Long> {
    private final long[] array;
    private final int start;
    private final int end;
    MaxValueTask(long[] array, int start, int end) {
        this.array = array;
        this.start = start;
        this.end = end;
    }
    protected Long compute() {
        long max = Long.MIN_VALUE;
        if (end - start <= RANGE_LENGTH) {
            for (int i = start; i < end; i++) {
                if (array[i] > max) {
                    max = array[i];
                }
            }
        } else {
            int mid = (start + end) / 2;
            MaxValueTask lowTask = new MaxValueTask(array, start, mid);
            MaxValueTask highTask = new MaxValueTask(array, mid, end);
            lowTask.fork();
            highTask.fork();
            max = Math.max(max, lowTask.join());
            max = Math.max(max, highTask.join());
        }
        return max;
    }
}
public void calculate(long[] array) {
    MaxValueTask task = new MaxValueTask(array, 0, array.length);
    Long result = forkJoinPool.invoke(task);
    System.out.println(result);
}
}

```

代码清单 11-20 的目的是给出 fork/join 框架的使用示例。单从性能方面来说，代码清单 11-20 中的实现方式的效率要低于直接对整个数组进行顺序比较的实现方式。这主要是因为多线程所带来的额外开销过大。在实际中，fork/join 框架发挥作用的场合是很多的，比如在一个目录包含的所有文本文件中搜索某个关键词时，可以为每个文件创建一个子任务。这种实现方式的性能要优于单线程的查找方式。如果相关的功能可以用递归和分治算法来解决，就适合使用 fork/join 框架。

11.6.2 多阶段线程同步工具

Phaser 类是 Java SE 7 中新增的一个实用同步工具，所提供的功能和灵活性比之前

介绍的倒数闸门和循环屏障要强很多。在 fork/join 框架中的子任务之间要进行同步时，应该优先使用 Phaser 类的对象。Phaser 类的特点是把多个线程协作执行的任务划分成多个阶段 (phase)，在每个阶段上都可以有任意个参与者参与。线程可以随时注册并参与到某个阶段的执行中来。当一个阶段中所有的线程都成功完成之后，Phaser 类的对象会自动进入下一个阶段。如此循环下去，直到 Phaser 类的对象中不再包含任何参与者。此时，Phaser 类的对象的运行自动结束。

在 Phaser 类的对象被创建出来之后，其初始的阶段编号为 0。在 Phaser 类的构造方法中可以指定初始的参与者的个数，也可以在创建出来之后，使用 register 方法或 bulkRegister 方法来动态添加一个或多个参与者。当某个参与者完成其任务之后，调用 arrive 方法来进行声明。有的参与者完成一次执行之后就不再继续参与，可以调用 arriveAndDeregister 方法在声明完成之后取消自己的注册。如果参与者在完成之后需要等待其他参与者完成，那么可以使用 arriveAndAwaitAdvance 方法。调用 arriveAndAwaitAdvance 方法的线程会阻塞，直到 Phaser 类的对象成功进入下一阶段。当需要等待 Phaser 类的对象进入下一个阶段时，可以使用 awaitAdvance 和 awaitAdvanceInterruptibly 方法。两个方法的参数是当前的阶段编号。使用 awaitAdvanceInterruptibly 方法时可以设置超时时间和处理中断请求。

当某个阶段中的所有参与者都完成任务之后，Phaser 类的对象的 onAdvance 方法会被调用，可以通过覆写此方法来添加自定义的处理逻辑。该方法的作用类似于在创建表示循环屏障的 CyclicBarrier 类的对象时使用的 Runnable 接口的实现对象。如果 onAdvance 方法的返回值为 true，则该 Phaser 类的对象会被终止。默认的实现是在参与者的数量为 0 时终止该 Phaser 类的对象。可以通过覆写 onAdvance 方法来使用不同的终止逻辑。要强制终止一个 Phaser 类的对象，可以使用 forceTermination 方法。

Phaser 类的一个重要特征是多个 Phaser 类的对象可以组织成树形结构。这种树形结构正好与 fork/join 框架中子任务可能形成的树形结构相对应。Phaser 类提供了一个构造方法来指定当前对象的父对象。当一个子 Phaser 类的对象中的参与者个数大于 0 时，它会被自动注册到父对象中；当参与者个数为 0 时，它会被自动从父对象中解除注册。

Phaser 类的功能很强大，在实际中可以替代 CountDownLatch 类和 CyclicBarrier 类。代码清单 11-21 给出了 Phaser 类的使用示例。该示例的 Java 类 WebPageImageDownloader 用来下载一个网页中包含的图片文件。整个下载过程由多个线程共同参与，并分成几个阶段。第一个阶段是由主线程负责下载页面的内容，并分析其中包含的图片文件的链接地址。在创建 Phaser 类的对象时，声明初始的参与者个数为 1，即只有主线程参与。对每个图片文件使用单独的线程来下载。在创建图片下载线程时，通过 register 方法把每个下载线程注册到 Phaser 类的对象中。在每个图片下载线程的 run 方法中先通过 arriveAndAwaitAdvance 方法等待其他线程创建完成。由于主线程也是参与者，同样需要通过调用 arriveAndAwaitAdvance 方法来进行等待。等所有线程都完成创建之后，Phaser 类的对象进入下一个阶段。下载图片文件的线程开始进行下载，

而主线程则通过第二个 `arriveAndAwaitAdvance` 方法等待所有下载线程完成运行。每个下载线程在完成任务之后，通过 `arriveAndDeregister` 方法进行声明，同时解除在 `Phaser` 类的对象上的注册。当所有下载线程都完成并解除注册之后，主线程成为 `Phaser` 类的对象中唯一的参与者。主线程最后通过 `arriveAndDeregister` 方法来解除注册，`Phaser` 类的对象由于不存在任何参与者而自动关闭。

代码清单 11-21 Phaser 类的使用示例

```
public class WebPageImageDownloader {
    private final Phaser phaser = new Phaser(1);
    private final Pattern imageUrlPattern = Pattern.compile("src=['\"]?(.*)?(\\.jpg|\\.gif|\\.png)['\"]?[\\s>]+", Pattern.CASE_INSENSITIVE);
    public void download(URL url, final Path path, Charset charset) throws IOException {
        if (charset == null) {
            charset = StandardCharsets.UTF_8;
        }
        String content = getContent(url, charset);
        List<URL> imageUrls = extractImageUrls(content);
        for (final URL imageUrl : imageUrls) {
            phaser.register();
            new Thread() {
                public void run() {
                    phaser.arriveAndAwaitAdvance();
                    try {
                        InputStream is = imageUrl.openStream();
                        Files.copy(is, getSavedPath(path, imageUrl), StandardCopyOption.REPLACE_EXISTING);
                    } catch (IOException e) {
                        e.printStackTrace();
                    } finally {
                        phaser.arriveAndDeregister();
                    }
                }
            }.start();
        }
        phaser.arriveAndAwaitAdvance();
        phaser.arriveAndAwaitAdvance();
        phaser.arriveAndDeregister();
    }
    private String getContent(URL url, Charset charset) throws IOException {
        InputStream is = url.openStream();
        return IOUtils.toString(new InputStreamReader(is, charset.name()));
    }
    private List<URL> extractImageUrls(String content) {
        List<URL> result = new ArrayList<URL>();
        Matcher matcher = imageUrlPattern.matcher(content);
        while (matcher.find()) {
            try {

```

```

        result.add(new URL(matcher.group(1)));
    } catch (MalformedURLException e) {
        // 忽略
    }
}
return result;
}
private Path getSavedPath(Path parent, URL url) {
    // 省略获取图片保存路径的代码
}
}

```

11.7 ThreadLocal 类

当多个线程需要同时访问一个共享变量时，不可避免地产生数据竞争。通常的做法是利用之前介绍的同步机制来解决这个问题。另外一种做法是使用线程局部变量，即 `java.lang.ThreadLocal` 类。在不同的线程访问一个 `ThreadLocal` 类的对象时，所访问和修改的是每个线程各自独立的对象，相当于这个对象是线程的一个私有对象。一个线程无法访问其他线程内部的私有对象，因此也不存在数据竞争的问题。通过使用 `ThreadLocal` 类，可以快速地把一个非线程安全的对象转换成线程安全的对象。

代码清单 11-22 中给出了 `ThreadLocal` 类的使用示例。在 `ThreadLocal` 类的对象中保存的是代码清单 11-1 中 `IdGenerator` 类的对象。`ThreadLocal` 类提供了对其中包含的对象进行处理的方法，`get` 和 `set` 方法分别用来获取和设置当前线程中包含的对象的值，而 `remove` 方法用来删除对象的值。一般的用法是覆写 `ThreadLocal` 类中的 `initialValue` 方法来提供对象的初始值。如果没有通过 `set` 方法来设置对象的值，那么在第一次调用 `get` 方法时会通过 `initialValue` 方法来获取对象的初始值。代码清单 11-22 也给出了 `ThreadLocal` 类的一般使用方式。创建一个 `ThreadLocal` 类的匿名子类并覆写 `initialValue` 方法，并把对 `ThreadLocal` 类的对象的使用封装在另外一个类中。程序的其他部分使用这个外部类的方法来获取被 `ThreadLocal` 类的对象所管理的对象。不同的线程调用 `ThreadLocalIdGenerator` 类的 `getNext` 方法时，所使用的是这个线程所对应的私有的 `IdGenerator` 类的对象，不同线程使用的是不同的对象。

代码清单 11-22 `ThreadLocal` 类的使用示例

```

public class ThreadLocalIdGenerator {
    private static final ThreadLocal<IdGenerator> idGenerator = new
        ThreadLocal<IdGenerator>() {
            protected IdGenerator initialValue() {
                return new IdGenerator();
            }
        };
    public static int getNext() {

```

```
        return idGenerator.get().getNext();
    }
}
```

ThreadLocal 类的另外一个作用是创建线程唯一的对象。某些对象的创建比较耗时，可以把对象的创建逻辑封装在 ThreadLocal 类的 initialValue 方法中。当通过 get 方法获取时，所有线程都可以得到唯一的一个对象。在同一个线程中运行的代码访问的都是同一个对象。在有些情况下，一个对象在代码中的各个部分都需要用到，传统的做法是把这个对象作为参数在代码之间进行传递。这种方式需要改变方法的类型声明。如果所有使用这个对象的代码都在同一个线程中运行，可以把该对象封装在一个 ThreadLocal 类的对象中，这样使用起来会非常方便。ThreadLocal 类适合用在 Web 应用的 servlet 中。一个 servlet 请求一般由单一的线程来处理。可以在开始处理请求时把某些全局对象保存到 ThreadLocal 类的对象中，并在后续的处理中使用，这些全局对象包括 servlet 请求对象、数据库连接和配置信息等。

在一个多线程程序中，如果需要生成随机数，应该使用 Java SE 7 中新增的 java.util.concurrent.ThreadLocalRandom 类。ThreadLocalRandom 类中的随机数生成器是使用 ThreadLocal 类来实现的，避免了使用 java.util.Random 对象可能带来的竞争问题，可以获得更佳的性能。

11.8 小结

在程序开发中使用多线程技术是一项复杂的任务，但是在很多情况下，多线程技术是必须使用的。以桌面程序来说，如果所有耗时的操作都在事件分发线程中完成，程序的界面会出现失去响应的情况。在实际开发中，对于多线程协作的场景，可以先对线程之间的交互模式进行抽象，并在 java.util.concurrent 包中寻找合适的同步方式与之对应。比如经典的生产者 - 消费者和读者 - 写者场景，使用 java.util.concurrent 包中提供的高层 API 可以很容易地实现。优先使用高层 API，应该是 Java 多线程开发中的首要原则。如果已有的高层 API 无法满足需求，那么应该使用 java.util.concurrent.atomic 和 java.util.concurrent.locks 包提供的中层 API 来创建程序自己的同步 API。而 synchronized 和 volatile 关键词，以及 Object 类中的 wait、notify 和 notifyAll 等低层 API 应该是最后才考虑的。

从理解和掌握多线程开发的角度出发，应该是按照从 Java 内存模型、低层 API、中层 API 到高层 API 的顺序来进行学习，这也是本章对应内容的排列顺序；而在实际的开发中，应该按照相反的顺序来考虑这些 API 的使用。

第 12 章 Java 泛型

在编程语言中，泛型是一个常见的特性。在主流编程语言（如 Java、C++ 和 C#）中都可以看到类似的语言特性。在程序中通常只对固定类型的对象进行操作。有些代码可以对多种不同类型的对象进行操作。实际使用的类型在代码中只是以参数形式出现的占位符，在具体实例化时，用实际类型替代其中的类型占位符，这种方式被称为泛型编程（generic programming）。泛型适用于对抽象类型进行处理，可以有效地减少代码重复，通过一份代码即可对不同类型的对象进行操作。典型的例子是处理集合相关的数据结构。对于这些数据结构可以进行一些抽象的操作，如排序和反转等。这些操作只与数据结构的特征相关，与其中包含的对象的类型无关。在实现这些操作时，对象的类型用占位符来表示即可，并不影响操作的实现逻辑。使用者可以根据需要指定实际的类型。

Java 语言发展到 J2SE 5.0 才引入了泛型的特性。泛型的主要动机是为了实现类型安全的集合类。除此之外，泛型还可以用来创建处理抽象类型的新类型。本章的内容围绕 Java 中的泛型展开，涉及的内容包括泛型的基本概念、底层实现和使用方式等。

12.1 泛型基本概念

引入泛型的主要动机是让开发人员更安全地使用 Java 标准库中的集合类，尽早地发现一些代码中包含的潜在错误。Java 的集合类框架在 JDK 1.2 中被添加到 Java 标准库中，其中包含了常用的 `java.util.List`、`java.util.Map` 和 `java.util.Set` 等接口及其实现类。在 J2SE 5.0 引入泛型之前，Java 中的集合类对象实际上是异构类型对象的集合。为了能够存放任何类型的对象，集合类中的元素的类型统一为 `Object` 类。在存放元素时，不论对象的实际类型如何，都可以将其保存到集合中。在读取元素时，需要对得到的对象进行强制类型转换，转换成对象的实际类型。使用强制类型转换的问题在于，运行时才可能发现类型不兼容的情况，由 `java.lang.ClassCastException` 异常来表示。运行时才发现的错误的处理代价比编译时发现的错误的处理代价要高得多，应该尽可能早地发现这些错误。

比如，在创建了一个 `List` 接口的实现类 `ArrayList` 的对象之后，可以向该对象中添加任何类型的对象。该 `ArrayList` 类的对象中可以同时包含 `String` 类和 `Number` 类的对象，这通常不是程序中所需要的行为。集合中通常包含的是同构类型的对象，但是 Java 语言并没有提供相应的机制来阻止向一个集合类的对象中添加不正确类型的对象，开发人员也不能在代码中表明集合类的对象中应该包含的对象类型。向一个集合类的对象中添加不兼容类型的对象，在编译时并不会出错。在运行时，当程序使用不正确类型的对

象时，会在进行强制类型转换时抛出 ClassCastException 异常。

为了实现类型安全的集合类，J2SE 5.0 引入了泛型的语言特性。泛型中包含的具体内容比较多，主要包括泛型类型和泛型方法的声明和实例化。泛型的引入也对 Java 标准库中的很多 API 造成了影响。泛型类型与一般类型的区别在于，泛型类型有形式类型参数 (type parameter)，可以在泛型类型被实例化时替换成实际的具体类型 (type argument)。先从一个具体的示例说起。代码清单 12-1 中的 ObjectHolder 类是一个简单的泛型类，用来保存特定类型的对象引用。与一般的类型不同，泛型类型声明中的“<T>”用来表示形式类型参数 T。形式类型参数 T 可以用在泛型类型实现的代码中。

代码清单 12-1 声明泛型类型的示例

```
public class ObjectHolder<T> {
    private T obj;
    public T getObject() {
        return obj;
    }
    public void setObject(T obj) {
        this.obj = obj;
    }
}
```

泛型类的使用与一般的 Java 类并没有太大的区别，只是需要为其中声明的形式类型参数指定实际的类型。代码清单 12-2 给出了泛型类 ObjectHolder 的基本使用方式。形式类型参数 T 在实例化泛型类型时被替换成实际类型 String。

代码清单 12-2 使用泛型类型的示例

```
ObjectHolder<String> holder = new ObjectHolder<String>();
holder.setObject("Hello");
String str = holder.getObject();
```

在创建出泛型类的对象之后，该对象在使用时的类型是受限的，比如代码清单 12-2 中的 holder 对象，在调用 setObject 方法时，参数的类型只能是 String 类型；getObject 方法的返回值也是 String 类型。如果不使用泛型来实现类似的功能，那么 setObject 方法的参数声明只能是 Object 类。同一对象的某个使用者可能在调用 setObject 方法时传入一个 String 类的对象，而另外一个使用者可能错误地认为其中包含的是 Number 类的对象，在调用 getObject 方法之后进行强制类型转换。这样在运行时会出现 ClassCastException 异常。如果希望只保存 String 类的对象，在不使用泛型的情况下，需要把 setObject 方法的参数和 getObject 方法的返回值都声明为 String 类型。这样在保存其他类型的对象时，需要创建对应类型的新的 Java 类。这些 Java 类的内部逻辑是相同的，造成了不必要的代码重复。通过使用泛型，只需要一个 Java 类就可以表示不同类型的对象。

结合代码清单 12-1 和代码清单 12-2 中对泛型类的声明和使用方式，对泛型相关的基本概念进行具体说明。如果在一个类型中使用了形式类型参数，则称该类型为泛型类型 (generic type)。代码清单 12-1 中的 ObjectHolder 类是一个泛型类。泛型类型可以被实例化。在实例化之后，泛型类型声明中的形式类型参数被替换成实际的类型。实例化之后的泛型类型被称为参数化类型 (parameterized type)。代码清单 12-2 中的 ObjectHolder<String> 是一种参数化类型。对于同一个泛型类型来说，可能的参数化类的数量非常多。根据使用的实际类型，参数化类型分为两类：一类是不带通配符的类型，另外一类是带通配符的类型。通配符 (wildcard) “?” 的作用是表示一组类型的集合，可以匹配特定范围内的类型。在使用通配符时可以指定其上界或下界。通过添加上界或下界可以限制通配符表示的具体类型的范围。不包含上界或下界的通配符被称为无界通配符 (unbounded wildcard)。例如，参数化类型 ObjectHolder<?> 表示其中包含的对象的具体类型是不确定的，可以是任何类型。在声明形式类型参数时也可以指定上界，用来限制实例化时可用的实际类型的范围。

在 Java 中，除了枚举类型、匿名内部类型和异常类型之外，其他类型都可以添加形式类型参数，成为泛型类型。形式类型参数的名称可以是 Java 中任何合法的标识符。一般使用单个大写字母作为形式类型参数的名称，以区别于一般的标识符。形式类型参数可以有多个，如 “MyClass<S, U, V>” 中声明了 3 个形式类型参数。不同的形式类型参数在代码中表示不同的含义。以集合类框架为例，List 接口只包含一个形式类型参数，表示列表中包含的元素的类型；而 Map 接口则包含两个形式类型参数，分别表示映射表中条目的键和值的类型。

形式类型参数类似于一般的类型，但是两者存在一些差别。两者的共同之处在于都可以作为类型使用在某些场合，包括作为方法的参数和返回值类型、作为域和局部变量的类型声明、进行强制类型转换及作为泛型类型和泛型方法的实际类型参数。但是形式类型参数在某些情况下是不能使用的，包括不能用来创建对象和数组、不能作为父类型、不能使用在 instanceof 表达式中、不能使用其类型字面量、不能出现在异常处理中，以及不能出现在静态上下文中。这就意味着，如果 T 是形式类型参数，类似 “new T()”、“new T[]”、“class MyClass extends T”、“instanceof T”、“T.class”、“catch (T)” 和 “static T” 等都是无法通过编译的错误用法。这些限制源于 Java 中泛型类型的实现机制，即类型擦除 (type erasure)。

为了兼容 J2SE 5.0 之前的遗留代码，泛型类型在使用时可以不指定实际类型。如果不指定实际类型而直接使用类型声明，所得到的类型被称为原始类型 (raw type)。如果在代码中直接使用 ObjectHolder 进行声明，则使用的是泛型类 ObjectHolder 的原始类型。原始类型的作用是与无法使用泛型的遗留代码进行交互。除此之外，原始类型不应该用在其他地方，否则引入泛型就变得毫无意义。使用原始类型是不安全的操作，编译器会给出相关的警告信息。

在构造方法或一般方法的声明中也可以使用形式类型参数。包含形式类型参数的方

法被称为泛型方法。泛型方法与泛型类型并没有直接的关系。在一个非泛型类型中同样可以包含泛型方法。泛型类型中的泛型方法可以使用在类型中定义的形式类型参数，也可以使用自己的形式类型参数。在调用泛型方法时，通常不需要显式指定所用的实际类型。编译器可以根据方法调用时的实际参数类型和上下文信息进行类型推断。代码清单 12-1 中的 ObjectHolder 类中的 setObject 和 getObject 都是泛型方法。

在使用泛型的情况下，编译器会对代码进行详细的类型检查。对于可以确定为错误的类型使用的地方，编译器会给出错误信息；对于无法判断是否正确的情况，编译器会给出警告信息，这意味着代码中存在可能的类型安全问题，开发人员需要谨慎处理这些警告。如果确定没有问题，那么可以使用“`@SuppressWarnings("unchecked")`”注解来抑制警告信息的输出。通常只有使用原始类型与遗留代码交互时所产生的警告信息才是可以忽略的。如果代码在编译过程中没有出现任何与类型安全相关的警告信息，那么代码在运行时肯定不会出现未预期的 ClassCastException 异常。

12.2 类型擦除

类型擦除是 Java 中泛型的实现方式。泛型是在编译器这个层次来实现的。在 Java 源代码中声明的泛型类型信息，在编译过程中会被擦除，只保留不带类型参数的形式。被擦除的类型信息包括泛型类型和泛型方法声明时的形式类型参数，以及参数化类型中的实际类型信息。经过类型擦除之后，包含泛型类型的代码被转换成不包含泛型类型的代码，相当于回到了泛型被引入之前的形式。Java 虚拟机在运行字节代码时并不知道泛型类型的存在。虽然为了反射 API 的需要，在 Java 字节代码中包含了与泛型类型相关的信息，但这些信息在字节代码执行时是不被使用的。与泛型相关的类型检查由编译器在编译时进行。

1. 基本概念

由于某些类型信息在编译过程中被擦除，因此并不是所有类型在运行时都是可用的。编译器和虚拟机所能区分的类型是不同的：对于编译器来说，`List<String>` 和 `List<Integer>` 是不同的类型；而对于虚拟机来说，这两者的类型都是 `List`。在运行时可用的类型被称为可具体化类型（reifiable type）。Java 中的可具体化类型包括非泛型类型、所有实际类型都是无界通配符的参数化类型、原始类型、基本类型、元素类型为可具体化类型的数组类型，以及父类型和自身都是可具体化类型的嵌套类型。举例来说，`String`、`List<?>`、`List`、`int`、`String[]` 和 `MyClass<?>.Inner` 都是可具体化类型。

除了实际类型都是无界通配符的参数化类型外，Java 泛型实现中的最重要的特点是几乎所有参数化类型都是不可具体化的。如 `List<String>` 和 `List<? extends Number>` 等类型都是不可具体化的。虚拟机在执行字节代码时只能使用运行时可用的可具体化类型。这使 Java 中与虚拟机相关的语法规特性对于不可具体化的参数化类型是不可用的。以异常处理为例，Java 代码运行时的异常捕获和处理是由虚拟机来完成的。因此异常类型必须

是可具体化的。任何泛型类型都不能直接或间接继承自 `Throwable` 类。Java 采用这种做法实现泛型的根本出发点是保持 Java 平台的兼容性，保证泛型引入之前的字节代码在不经过任何修改的情况下就可以在新版本的虚拟机上运行。因此，Java 选择在编译器这个层次来实现泛型，而保持虚拟机不变。对于 Java 这样一个使用广泛的语言来说，这种兼容性很重要，但也带来了泛型在设计上的不自然和使用方式上的局限性。这是各方面利弊因素权衡的结果。

在类型擦除过程中需要处理形式类型参数和参数化类型中的实际类型。对于形式类型参数，在泛型类型声明中的部分会被直接删除，如 `ObjectHolder<T>` 被替换成 `ObjectHolder`；在泛型类型代码中出现的则根据上界替换成具体的类型。如果形式类型参数声明了上界，则声明中最左边的上界作为进行替换的类型；如果没有上界，则使用 `Object` 类进行替换。而对于参数化类型的实际类型，它们在代码中的出现会被直接删除。进行这些替换之后，可能会出现代码逻辑不合法的情况，编译器会通过插入适当的强制类型转换代码和生成桥接方法（bridge method）来解决。

以代码清单 12-1 中的 `ObjectHolder` 类为例，经过类型擦除后的形式如代码清单 12-3 所示，由于形式类型参数 `T` 没有上界，`T` 的所有出现被替换成 `Object` 类。

代码清单 12-3 泛型类型经过类型擦除之后的代码示例

```
public class ObjectHolder {
    private Object obj;
    public Object getObject() {
        return obj;
    }
    public void setObject(Object obj) {
        this.obj = obj;
    }
}
```

代码清单 12-2 中使用 `ObjectHolder` 类的代码，经过类型擦除后的形式如代码清单 12-4 所示。在类型擦除后，`ObjectHolder` 类中的 `getObject` 方法的返回值类型实际上是 `Object` 类型，因此需要添加强制类型转换把 `getObject` 方法的返回值转换成 `String` 类型。这些类型转换操作由编译器自动添加。由于编译器已经确保不允许使用除 `String` 类的对象之外的其他对象调用 `setObject` 方法，因此这个强制类型转换操作始终是合法的。

代码清单 12-4 使用泛型类型的代码在类型擦除之后的示例

```
ObjectHolder holder = new ObjectHolder();
holder.setObject("Hello");
String str = (String) holder.getObject();
```

2. 桥接方法

当一个类继承某个参数化类或实现参数化接口时，在经过类型擦除之后，可能造成

所继承的方法的类型签名发生改变。典型的示例是 `java.lang.Comparable` 接口的实现类。代码清单 12-5 给出的 `Sequence` 类实现了 `Comparable` 接口并定义了 `compareTo` 方法的实现。在 `Comparable` 接口中添加了实际类型 `Sequence`，说明 `Sequence` 类的对象只有与类型相同的对象进行比较才有意义。因此 `compareTo` 方法的参数类型是 `Sequence` 类。如果试图在调用 `compareTo` 方法时使用除 `Sequence` 类及其子类之外的其他类型的对象作为参数，那么会出现编译错误。

代码清单 12-5 Comparable 接口的实现类

```
public class Sequence implements Comparable<Sequence> {
    private final int sequenceNumber;
    public Sequence(int sequenceNumber) {
        this.sequenceNumber = sequenceNumber;
    }
    public int compareTo(Sequence sequence) {
        return Integer.compare(sequenceNumber, sequence.sequenceNumber);
    }
}
```

在经过类型擦除之后，`Comparable` 接口的实际类型“`<Sequence>`”被删除。`Sequence` 类的声明变成了实现原始的 `Comparable` 接口。从接口实现的角度来说，这要求 `Sequence` 类中包含一个类型签名为“`int compareTo(Object)`”的方法，否则 `Sequence` 类的实现是不正确的。这是由类型擦除造成的，编译器需要添加相应的方法来确保代码实现的正确性。这些由编译器自动添加的方法被称为桥接方法。

对 `Sequence` 类来说，编译器会自动添加一个类型签名为“`int compareTo(Object)`”的方法。在桥接方法的实现中，只是在进行必要的类型转换之后直接调用 `Sequence` 类中定义的类型签名为“`int compareTo(Sequence)`”的方法。代码清单 12-6 中给出了桥接方法的内部实现。

代码清单 12-6 桥接方法的内部实现

```
public int compareTo(Object obj) {
    return this.compareTo((Sequence) obj);
}
```

虽然自动添加的桥接方法 `compareTo` 接受 `Object` 类型的参数，但是代码中不能直接使用这个方法。使用除 `Sequence` 类及其子类之外的其他类型的对象调用 `compareTo` 方法会产生编译错误。由于桥接方法在运行时是可见的，可以通过反射 API 来查找并调用桥接方法。代码清单 12-7 给出了使用反射 API 来调用 `Sequence` 类中的桥接方法的示例。通过反射 API 可以查找出表示桥接方法的 `java.lang.reflect.Method` 类的对象。通过 `Method` 类的 `isBridge` 方法可以判断一个方法是否为桥接方法。当尝试使用错误类型的参数调用该方法时，会抛出 `ClassCastException` 异常。

代码清单 12-7 使用反射 API 调用桥接方法

```

public void invoke() {
    try {
        Method method = Sequence.class.getMethod("compareTo", new Class<?>[]
            {Object.class});
        method.isBridge(); // 值为 true
        Sequence seq1 = new Sequence(1);
        Sequence seq2 = new Sequence(2);
        method.invoke(seq1, seq2);
        method.invoke(seq1, "Hello"); // 抛出 ClassCastException 异常
    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

3. 类型擦除对泛型特性的影响

类型擦除机制的存在影响了很多泛型的特性。同一泛型类型的所有实例化形式在运行时的表示形式是相同的。每个泛型类型只对应一份字节代码。虚拟机并不区分同一泛型类型的不同实例化形式。List<String> 和 List<Integer> 类型对于虚拟机来说是相同的，表示的都是 List 接口。所以无法通过类似“List<String>.class”的形式来获取参数化类型的类对象字面量，而只能使用 List.class。在运行时并不存在 List<String> 类型，只有 List 类型。

除了实际类型都是无上界通配符外的泛型类型的其他实例化形式，都不能用在 instanceof 操作符中。例如，除了类似“obj instanceof List<?>”之外的其他使用方式，如“obj instanceof List<String>”和“obj instanceof List<? extends Serializable>”等，都是非法的。这是因为 instanceof 操作符是根据对象的运行时类型来进行判断的，只对具体化类型有意义。对于参数化类型来说，只能比较类型擦除之后的类型。在 instanceof 操作符看来，一个 ArrayList<String> 类的对象也是 ArrayList<Integer> 类型的实例。因为这两种参数化类型在类型擦除之后的类型都是 ArrayList。如果允许这种行为，开发人员容易产生误解，可能破坏代码中的类型安全，因此， instanceof 操作符不允许这样的使用方式。

在泛型类型中定义的静态方法和域是被所有的实例化形式的对象所共享的。代码清单 12-8 中用不同的实际类型实例化了泛型类 StaticField。对于虚拟机来说，静态变量 count 与类型擦除之后的 StaticField 类型相对应，与具体的参数化类型无关。在引用泛型类型中定义的静态变量和方法时，直接使用原始类型，不能使用参数化类型。“StaticField<String>.count”是非法的引用方式。

代码清单 12-8 泛型类型中的静态域

```

public class StaticField <T> {
    public static int count = 0;
}

```

```

public StaticField() {
    count++;
}
public static void main(String[] args) {
    new StaticField<String>();
    new StaticField<Integer>();
    System.out.println(StaticField.count); // 输出为 2
}
}

```

泛型类型声明中的形式类型参数不能出现在任何静态上下文中，包括不能出现在静态域的类型声明中、不能出现在静态方法的声明和实现中、不能出现在静态初始化代码块中，以及不能出现在静态嵌套类型中。静态嵌套类型包括静态嵌套类、嵌套接口和嵌套枚举类型等。类型中的静态上下文是与类型关联在一起的，与类型的实例对象无关。由于泛型类型的不同实例化形式在运行时对应的是同一个类型，在静态上下文中使用形式类型参数并没有意义，反而容易造成开发人员的误解，因此，编译器直接禁止这样的用法。

12.3 上界和下界

泛型类型和泛型方法与一般类型和方法的区别在于可以声明形式类型参数。在实际的类型声明中，形式类型参数会被替换成具体的类型。虽然形式类型参数在源代码中可以使用，但其使用方式是受限的。编译器把形式类型参数当成实际类型的占位符。编译器不能对实际声明时可能使用的类型做出任何假定，只能使用最严格的检查方式，以避免出现类型安全错误。如代码清单 12-1 中的 ObjectHolder 类的形式类型参数 T，在 setObject 方法中作为参数的类型。在 setObject 方法中，可以对类型为 T 的参数对象进行的操作并不多。编译器所允许的是把该参数对象当成 Object 类的对象来处理，只能调用 Object 类中的方法。除此之外的任何操作都可能带来类型安全问题。

如果希望限制在实际类型声明时可以使用的类型，那么可以为形式类型参数添加上界。在添加了上界之后，泛型类型在实例化时只能使用由上界表示的类型及其子类型。代码清单 12-9 通过 extends 关键词为形式类型参数 T 添加了上界 Comparable<T> 接口。声明泛型类型时的实际类型必须实现 Comparable 接口，否则会出现编译错误。这样可以确保实际类型实现了 Comparable 接口，可以在代码中调用参数对象的 compareTo 方法。如果没有显式指定上界，那么默认的上界是 Object 类。形式类型参数虽然有上界，但是没有下界，这是因为下界在实际中几乎没有作用。

代码清单 12-9 形式类型参数的上界

```

public class ComparableObjectHolder<T extends Comparable<T>> {
    private T obj;
    public int compareTo(T anotherObj) {

```

```

        return obj.compareTo(anotherObj);
    }
}

```

形式类型参数上界的作用是限制泛型类型实例化时可以使用的类型，可以在代码中使用上界类型中提供的公开成员，包括公开的方法、域和嵌套类型，但是不包括构造方法。这是因为构造方法是不被继承的，子类的构造方法的参数可以与父类构造方法不同。所以使用类似“new T()”这样的表达式并不能保证可以正确创建出所需的对象。

除了基本类型和数组类型之外的其他类型都可以作为形式类型参数的上界，其中除了一般的类和接口之外，还包括参数化类型和形式类型参数。代码清单 12-9 中的上界用的是参数化类型。对于 int 和 float 等基本类型，可以使用其对应的包装类来作为上界；对于数组类型，可以使用集合类框架中的类或接口来替代。代码清单 12-10 中的 SampleClass 类使用了两个形式类型参数 S 和 T，且 T 是 S 的上界。这就要求实例化时第一个类型与第二个类型相同，或者第一个类型是第二个类型的子类型。类似“SampleClass<String, Comparable<String>>”的实例化形式是合法的。

代码清单 12-10 使用形式类型参数作为上界

```

public class SampleClass<S extends T, T> {
    public void test() {
        SampleClass<String, Comparable<String>> obj = new SampleClass<>();
    }
}

```

一个形式类型参数可以包含多个上界，不同上界之间使用“&”来分隔。在实例化时所能使用的类型是所有上界的子类型。代码清单 12-11 中类的形式类型参数使用 Cloneable 接口和 java.io.Serializable 接口作为共同的上界。

代码清单 12-11 形式类型参数包含多个上界的示例

```

public class CloneableSerializable<T extends Cloneable & Serializable> {
    public void serialize(T obj) {
    }
}

```

这些上界之间的顺序虽然不会对实例化时所能使用的具体类型的范围产生影响，但是会影响类型擦除之后使用的类型。在类型擦除过程中，形式类型参数会被最左边的上界所替代。例如代码清单 12-11 中的类，在类型擦除之后，serialize 方法的参数的类型是 Cloneable 接口。如果在声明时使用的形式是“<T extends Serializable & Cloneable>”，那么在类型擦除之后，serialize 方法的参数的类型是 Serializable 接口。不过由于实际类型是所有上界的子类型，具体使用哪个上界类型来表示实际上是没有差别的。

除了形式类型参数之外，泛型类型实例化时的实际类型也可以包含上界或下界。参

数化类型中的上界或下界都是与通配符一块来使用的。

12.4 通配符

在实例化一个泛型类型时，需要为该泛型类型中的形式类型参数指定具体的类型。在某些情况下，所要使用的具体类型是确定的。比如声明一个只包含 String 类型对象的 List 接口，直接使用 List<String> 即可。在另外一些情况下，并不需要对具体的类型做出明确的限定，只需要这些类型满足一定的条件即可。比如对一组对象进行排序的方法，并不要求对象的类型是固定的某个类型，只要这些类型实现 Comparable 接口，就可以使用 Comparable 接口的 compareTo 方法来确定顺序。这时要表示的不是单个的类型，而是一组类型的集合。通配符的作用是描述一组类型，而其中包含的类型要满足的条件由通配符的上界或下界来声明。通配符用“?”表示，分成无界和有界两种。无界通配符只用“?”来表示，代表的是包含所有类型的集合；有界通配符可以有上界或下界，分别用“? extends”和“? super”来表示，但不能同时有上界和下界。上界通配符代表的是包含上界类型及其子类型的集合，而下界通配符代表的是包含下界类型及其父类型的集合。

通配符的含义是表示一组类型的集合，不同于单个具体的类型。为了表示这种类型，编译器在内部使用通配符捕获（wildcard capture）类型的方式。通配符捕获类型是一种特殊的类型，可以表示通配符所代表的类型集合中的任意类型。编译器在涉及通配符的类型比较中使用通配符捕获类型来进行处理。通配符捕获类型与所有的具体类型都是不兼容的，因为一个类型的集合无法与单个具体类型兼容。通配符捕获类型只能与其他的通配符捕获类型相兼容。例如，对于一个类型声明为“List<? extends Number>”的变量，编译器在内部实际使用的通配符捕获类型是“List<capture#1-of ? extends Number>”。由于类型不兼容，代码清单 12-12 中 use 方法的赋值操作会出现编译错误。编译器给出的错误信息是“Type mismatch: cannot convert from List<capture#1-of ? extends Number> to List<String>”，其含义是赋值操作试图把“List<capture#1-of ? extends Number>”类型的对象转换成“List<String>”类型，由于这两个类型不兼容，类型转换操作无法进行。

代码清单 12-12 说明通配符捕获类型的示例

```
public class Wildcard {
    public List<? extends Number> createList() {
        return new ArrayList<>();
    }

    public void use() {
        List<String> list = createList(); // 编译错误
    }
}
```

获取一个包含通配符的参数化类型的对象引用之后，对这个对象引用所能做的操作是非常有限的。例如，对于一个类型为 `List<?>` 的对象引用，当尝试调用 `add` 方法向列表中添加任何类型的对象时，都会出现编译错误，因为编译器无法确定 `List<?>` 类型对象中实际的类型，这个 `add` 方法的调用有可能是不安全的，因此编译器禁止了这样的行为。但是在很多情况下，只能使用含通配符的参数化类型作为引用的类型。例如，`Class` 类中的很多方法的返回值都是 `Class<?>` 类型的。以 `Class` 类中的 `getInterfaces` 方法为例，该方法用来介绍获取当前类所实现的接口或当前接口所继承的接口。该方法的返回值类型是 `Class<?>[]`。使用其他更加具体的类型作为返回值类型都是不正确的，因为该方法的返回值的实际类型可能是任何类型。在使用包含通配符的参数化类型的对象引用时，通常需要提供额外的类型信息来方便对该对象的使用。

下面通过一个示例来说明如何提供额外的类型信息。在一个程序中包含了某个接口的多个不同实现类，程序根据相关的配置信息来选择使用的具体实现类。代码清单 12-13 给出了一个创建该接口实现对象的示例。在调用 `create` 方法时，需要提供表示接口类型的 `Class` 类的对象作为额外的参数。找到具体实现类的类名之后，使用 `Class` 类的 `forName` 方法来加载该类。由于 `forName` 方法的返回值类型是 `Class<?>`，因此通过由参数给出的接口类型的 `cast` 方法把创建出来的对象转换成正确的类型。

代码清单 12-13 为包含通配符的参数化类型提供类型信息

```
public class ObjectFactory {
    public static <T> T create(Class<T> interfaceType) throws Exception {
        String className = searchForClassName();
        Class<?> clazz = Class.forName(className);
        return interfaceType.cast(clazz.newInstance());
    }

    private static String searchForClassName() {
        return ""; // 省略查找逻辑
    }
}
```

在上面的示例中，具体的类型信息由 `create` 方法的调用者来提供，这就保证了 `create` 方法的返回值是调用者所期望的类型。

12.5 泛型与数组

数组是 Java 语言中的基本数据结构，有其自身的特殊性。数组对象是由 Java 虚拟机根据元素类型创建出来的。数组不同于集合类对象的一个重要特征是数组是协变的 (covariant)。如果一个数组类型的元素类型是另外一个数组类型的元素类型的子类型，那么这个数组类型同时也是对应的数组类型的子类型。因此，`String[]` 类型是 `Object[]` 的子类型。这种协变关系对于集合类对象来说是不存在的。比如，`List<String>` 并不是

List<Object> 的子类型。其中的原因是数组的元素类型信息在运行时仍然是被保留的，而泛型类型的类型信息因类型擦除机制而被去掉。在运行时，虚拟机可以利用数组的元素类型信息来判断对数组的操作是否合法。如果尝试向数组中添加类型不兼容的对象，那么在运行时会产生 java.lang.ArrayStoreException 异常。在进行类型检查时，使用的是元素的运行时类型。

1. 数组声明时的可用类型

在代码清单 12-14 中创建了一个 Integer 类型的数组，但是用 Object[] 类型来引用。由于 Integer[] 是 Object[] 的子类型，因此把 Integer[] 类型的对象赋值给 Object[] 类型的变量是合法的。当尝试向数组中添加 String 类型的元素时，虽然编译时不会出现错误，但运行时会抛出 ArrayStoreException 异常。因为 array 变量所指向的实际是 Integer 类型的数组，所以虚拟机会根据运行时类型来进行判断。在使用泛型集合类对象时并不会出现这种问题。如果把代码改写成使用泛型集合类型的形式，如类似“List<Object> list = new ArrayList<Integer>”的方式，会发现这条语句无法通过编译。使用泛型可以避免出现类似的潜在问题。

代码清单 12-14 使用数组时可能产生的运行时类型安全问题

```
public void storeInArray() {
    Object[] array = new Integer[10];
    array[0] = "Hello"; // 抛出异常
}
```

由于数组的这种特殊性，除了只包含无界通配符的参数化类型和原始类型之外，其他泛型类型的实例化形式都不允许用来创建数组。也就是说，只有可具体化类型才可以用来创建数组。如果不对泛型的实例化形式做出限制，就可能造成操作数组元素时的动态类型检查失效，造成错误类型的对象被添加到数组中。在代码清单 12-15 的方法中，如果创建 ArrayList<String> 类型的数组的操作是合法的，那么尝试向该数组中添加 ArrayList<Integer> 类型的对象的操作不但在编译时是合法的，在运行时也不会抛出 ArrayStoreException 异常。原因是经过类型擦除之后，ArrayList<String> 和 ArrayList<Integer> 的实际类型都是 ArrayList，虚拟机认为这是合法的数组保存操作。但是，数组的使用者在使用数组的元素时，会遇到 ClassCastException 异常。为了避免出现这种问题，编译器不允许创建 ArrayList<String> 类型的数组，因此代码清单 12-15 中方法的第一行代码会出现编译错误。

代码清单 12-15 创建参数化类型数组的错误示例

```
public void storeInGenericArray() {
    Object[] array = new ArrayList<String>[10]; // 编译错误
    array[0] = new ArrayList<Integer>();
}
```

唯一允许用来创建数组的是泛型类型的只包含无界通配符的实例化形式，例如，“new List<?>[]”是合法的。这是因为无界通配符表示的是所有类型的集合，不管数组中存放的元素的具体类型是什么，使用无界通配符进行引用总是合法的。不过由于无界通配符在作为对象引用时的使用限制，创建这种形式的数组的意义并不大。需要注意的是，虽然不允许创建数组，但是元素为参数化类型的数组引用是合法的，例如，“List<String>[] list = null;”是一个合法的语句。允许使用这种引用方式的原因是一个非泛型类型可以继承自某个参数化类型，而用这个非泛型类型创建数组是合法的。代码清单 12-16 给出了一个示例，非泛型类型 StringArrayList 继承自参数化类型 ArrayList<String>，在 createArray 方法中使用 StringArrayList 类型来创建数组，同时用 ArrayList<String>[] 类型来引用这个数组。

代码清单 12-16 继承自参数化类型的非泛型类型

```
public class StringArrayList extends ArrayList<String> {
    public void createArray() {
        ArrayList<String>[] array = new StringArrayList[10];
    }
}
```

2. 调用参数长度可变方法时的隐式数组的使用

另外一个与泛型和数组相关的内容是调用参数长度可变的方法。在调用参数长度可变的方法时，实际参数是通过数组来传递的，比如方法声明“void method(String... values)”实际上等价于“void method(String[] values)”。在实际调用方法时，编译器根据实际参数的个数创建一个数组对象，并把实际参数保存到数组中，再把数组传递给方法。数组的创建工作由编译器自动完成。当长度可变参数的类型是可具体化类型时，这种数组创建工作并没有问题。如果参数的类型是不可具体化类型，则可能会出现类型安全问题，这也是不允许创建元素类型为不可具体化类型的数组的原因。但是，编译器并没有禁止使用不可具体化类型作为长度可变参数的类型，只是给出相关的警告信息。忽视这些警告有可能产生运行时的异常。代码清单 12-17 中的 varargsMethod 方法的参数类型是不可具体化的 List<String>。在方法实现中，可以使用 Object[] 类型引用作为中间变量把 List<String> 类型转换成 List<Integer> 类型，并添加 Integer 类型的对象到列表中。

代码清单 12-17 长度可变参数的类型为不可具体化类型时存在的类型安全问题

```
public class Varargs {
    public void varargsMethod(List<String>... values) {
        Object[] array = values;
        List<Integer> list = (List<Integer>) array[0];
        list.add(1);
    }

    public void useVarargsMethod() {
```

```

        List<String> list = new ArrayList<>();
        list.add("Hello");
        varargsMethod(list);
        String str = list.get(1); // 抛出 ClassCastException 异常
    }
}

```

当长度可变参数的类型是不可具体化类型时，如果在方法的实现中只读取参数中的内容，而不修改参数中的对象，那么不会出现类型安全问题。如果可以确定一个方法不会产生类型安全问题，那么可以使用 Java SE 7 中的“@SafeVarargs”注解来进行声明。编译器对添加了“@SafeVarargs”注解的方法的调用不会给出警告信息。

12.6 类型系统

在 Java 语言中，类型系统描述了不同类型之间的转换关系。如果一个类型是另外一个类型的子类型，那么从子类型到父类型的转换是自动进行的，而从父类型到子类型的转换需要在代码中显式进行，这是因为从父类型到子类型的转换可能是不安全的，需要由开发人员显式进行。在泛型被引入之前，Java 中的父类型和子类型的关系主要通过类继承和接口实现机制来声明。如果类或接口 A 继承自类或接口 B，则 A 是 B 的子类型；如果类 C 实现了接口 D，则 C 是 D 的子类型。泛型的引入对 Java 语言中的类型系统产生了比较大的影响。这是因为泛型类型的实例化形式中包含了所使用的实际类型，这些类型之间也可以有父子类型关系。这相当于把类型系统从之前的一维结构扩展为二维结构：一个维度是泛型类型本身，另外一个维度是参数化类型中的实际类型。比如 ArrayList 类实现了 List 接口，因此 ArrayList 类是 List 接口的子类型。把泛型考虑在内，对于 ArrayList<Number> 和 List<Number>、ArrayList<Integer> 和 ArrayList<Number>，以及 ArrayList<Integer> 和 List<Number> 这三组参数化类型，虽然 ArrayList 类是 List 接口的子类型，Integer 类也是 Number 类的子类型，但是这三组类型之间并不一定存在子类型与父类型的关系。

先看两种简单的情况。当两个参数化类型的实际类型完全相同时，两个类型的父子类型关系取决于泛型类型本身的父子关系。对于 ArrayList<Number> 和 List<Number> 这两个参数化类型，它们的实际类型都是 Number，因此两者的父子类型关系取决于 ArrayList 类和 List 接口之间的父子类型关系，从而可以得出 ArrayList<Number> 是 List<Number> 的子类型。这条规则对于包含通配符的情况也是适用的，所以 ArrayList<? extends Number> 是 List<? extends Number> 的子类型。

如果泛型类型相同，在实例化时使用的是不包含通配符的两个不同的具体类型，则这两个类型之间不存在任何父子类型的关系。例如，ArrayList<Integer> 和 ArrayList<Number> 两个类型之间不存在父子类型的关系。这一点是比较容易理解错误的地方。一般的理解是，既然 Integer 类是 Number 类的子类型，那么包含一组 Integer

类的对象的列表，同时也应该是包含一组 Number 类的对象的列表。实际上，如果允许这种父子类型关系的存在，那么会产生类型安全问题。

代码清单 12-18 给出了具体的示例。类 ModifyList 中的 modify 方法的参数类型是 ArrayList<Number>，在 changeList 方法中试图把一个 ArrayList<Integer> 类型的对象传递给 modify 方法，但是编译器并不允许这么做。假设这种传递方式是合法的，在 modify 方法中，由于参数类型是 ArrayList<Number>，因此该方法可以向参数对象中添加任何 Number 类及其子类型的对象，包括与 Integer 类型不同的 Float 或 Double 类的对象。而变量 list 的使用者所期望的是 ArrayList 类的对象中只包含 Integer 类型的对象，这样就出现了类型安全问题，所以编译器禁止这种使用方式。

代码清单 12-18 ArrayList<Number> 和 ArrayList<Integer> 无法兼容的说明示例

```
public class ModifyList {
    public void modify(ArrayList<Number> list) {
        list.add(1.0f); // 添加一个 Float 类型的对象
    }
    public void changeList() {
        ArrayList<Integer> list = new ArrayList<Integer>();
        list.add(3);
        modify(list); // 编译错误
        Integer value = list.get(1);
    }
}
```

上面只介绍了在对泛型类型实例化时使用的类型完全相同的情况，如果考虑到类型之间的转换关系和通配符的使用，则情况更加复杂。在参数化类型的实际类型这个维度上，父子类型的关系只存在于包含通配符的情况。通配符代表的是一组类型，而不是单一的具体类型。当两个类型都是具体类型时，会遇到之前介绍的类似 ArrayList<Integer> 和 ArrayList<Number> 类型的问题，所以至少要有一个类型是包含通配符的类型，才可能存在父子类型的关系。实际类型维度上的父子类型关系可以看成是所表示的类型集合之间的包含关系。如果一个类型集合是另外一个类型集合的子集，则在对应的相同泛型类型的实例化形式中，前者是后者的子类型。实际上，单一类型也可以看成是包含单个类型元素的集合。在两个类型不相同的情况下，这两个单一元素集合中不可能有其中一个是另外一个的子集合，因此也不可能存在父子类型的关系。

在包含通配符时，无界和有界通配符的情况有所不同。对于无界通配符来说，使用无界通配符的参数化类型是所有该泛型类型的其他实例化形式的父类型，因为无界通配符表示的是所有可能的类型的集合。从集合论的角度来说，无界通配符表示的是全集，包含所有其他的集合。例如，List<?> 是所有 List 泛型类的实例化形式的父类型，List<String>、List<? extends Number> 和 List<? super Integer> 等都是 List<?> 的子类型。

使用有界通配符的情况取决于具体使用的上界或下界的类型。上界通配符表示的是

上界类型及其子类型的集合。两个包含上界通配符的类型对应的集合之间的关系取决于上界类型本身是否存在父子类型关系。例如，“? extends A” 和 “? extends B” 之间的关系取决于类型 A 和 B 之间的关系，这是因为如果 A 是 B 的子类型，那么包含在 “? extends A” 对应的集合中的类型肯定包含在 “? extends B” 对应的集合中。由于 Integer 类是 Number 类的子类型，因此 “? extends Integer” 是 “? extends Number”的子类型，进而可知 List<? extends Integer> 是 List<? extends Number> 的子类型。上界通配符的情况对于下界通配符也是适用的，只不过父子类型关系要颠倒过来。两个包含下界通配符的类型对应的集合之间的关系同样取决于下界类型本身是否存在父子类型关系。对于 “? super A” 和 “? super B” 来说，当 B 是 A 的子类型时，“? super A” 是 “? super B”的子类型，因此 “? super Number” 是 “? super Integer”的子类型，进而可知 List<? super Number> 是 List<? super Integer> 的子类型。

有些泛型类型包含多个形式类型参数，在实例化时可以使用对应的多个通配符。对于每个实际类型都包含通配符的情况，对每个类型都应用上面提到的规则来进行判断。如果一个参数化类型声明中的所有类型都是另外一个参数化类型的对应类型的子类型，则前者是后者的子类型。比如，类型 Map<Integer, ? extends Number> 是 Map<? extends Number, ?> 的子类型，这是因为 Map<Integer, ? extends Number> 的第一个实际类型 Integer 包含在 “? extends Number” 所表示的类型集合中，第二个类型参数 “? extends Number” 所表示的集合是 “?” 所表示的集合的子集。

在参数化类型的单个类型中也可以多次使用通配符，如在 List<? extends List<? extends Number>> 类型声明中使用了两个上界通配符作为实际的类型。对于这种情况，对父类型与子类型的判断比较复杂。基本的思路是按照递归的方式依次进行判断，从最内层的类型开始进行判断。比如，对于 List<? extends List<Integer>> 和 List<? extends List<? extends Number>> 这两个类型，具体的判断过程是：Integer 类型是 “? extends Number”的子类型，进而可知 List<Integer> 是 List<? extends Number> 的子类型。而 List<Integer> 和 List<? extends Number> 作为通配符的上界出现，因此 List<? extends List<Integer>> 是 List<? extends List<? extends Number>> 的子类型。

最后一种特殊的父子类型的关系与原始类型有关。一个泛型类型的所有实例化形式是其对应的原始类型的子类型，如 List<String> 和 List<? extends Number> 都是 List 的子类型。这么设计的目的是为了与不使用泛型的遗留代码进行交互。

因为父子类型关系是传递的，所以在进行复杂类型的判断时，可以根据上面的规则从简单的情况开始考虑。比如，在判断 ArrayList<Integer> 和 Collection<? extends Number> 之间的关系时，先考虑到 ArrayList<Integer> 是 Collection<Integer> 的子类型，而 Collection<Integer> 是 Collection<? extends Number> 的子类型，所以 ArrayList<Integer> 是 Collection<? extends Number> 的子类型。

12.7 覆写与重载

覆写 (override) 与重载 (overload) 是 Java 语言中两个比较容易混淆的概念。覆写是指在子类型中重新定义从父类型中继承的方法的实现。覆写是面向对象中的重要概念，通过与继承关系结合来实现基于运行时类型的动态方法分派。当使用一个子类型的对象引用调用一个方法时，实际执行的是子类型中覆写的方法。重载指的是类中可以包含名称相同但类型签名 (signature) 不同的方法。重载的作用是允许方法的使用者采用不同的参数来调用相同名称的方法。重载的方法可以看成是一组功能相同的方法，只是对于不同的参数类型进行操作。覆写只发生在子类型和父类型之间。由于子类型会继承父类型中的方法，因此子类型中包含的方法既可以覆写父类型中的方法，也可能重载从父类型继承下的方法。覆写和重载的语义不同，需要正确地区分这两种情况。常见的问题是在需要覆写父类型方法时错误地进行了重载，导致父类型中的对应方法没有被覆写，当使用子类的对象调用方法时，仍然调用的是父类的方法，这会造成难以发现的错误。由于泛型的引入，覆写和重载的区分条件变得更加复杂。

覆写和重载的共同前提条件是方法名称相同。名称不同的方法不可能存在覆写或重载的关系。方法的覆写需要满足的条件包括方法类型签名、返回值类型和抛出的异常类型三个方面。因为代码可能使用父类的对象引用来调用方法，并把方法的调用结果赋值给变量，同时还需要处理方法调用中可能产生的异常，所以如果子类型覆写了父类型中的方法，要保证覆写的方法在使用父类型的对象引用进行调用时也是合法的。这一点是基于运行时类型的动态方法分派的基础。调用者使用父类型的引用来进行操作，而运行时的实际类型可能是该父类型的子类型。这一点对于调用者来说是透明的。

12.7.1 覆写对方法类型签名的要求

方法类型签名包括方法名称和参数类型两个部分。子类型中的方法覆写父类型中方法的条件是两个方法的类型签名是相同的，或者父类型中的方法在类型擦除之后的类型签名与子类型中方法的类型签名相同。这两种情况对于一般类型和参数化类型都适用。代码清单 12-19 给出了用来测试方法覆写的代码示例。类 SuperClass 和其子类 SubClass 中包含名称相同的方法 method，通过查看调用子类对象的 method 方法的执行结果，可以判断实际调用的是 SuperClass 类中的对应方法还是 SubClass 类中的对应方法，从而可以知道方法覆写是否生效。代码清单 12-19 中子类的 method 方法的类型签名与父类的 method 方法在类型擦除之后的类型签名相同，因此这是一个正确的覆写操作。

代码清单 12-19 方法覆写是否生效的测试代码

```
class SuperClass {
    public void method(List<?> param) {
        System.out.println("调用父类中的方法。");
    }
}
```

```

    }

    class SubClass extends SuperClass {
        public void method(List param) {
            System.out.println("调用子类中的方法。");
        }
    }

    public class OverrideTest {
        public static void main(String[] args) {
            SubClass subClass = new SubClass();
            subClass.method(new ArrayList<String>()); // 调用子类的方法
        }
    }
}

```

方法的参数类型相同的条件对于一般类型和参数化类型都适用。不管 SuperClass 类和 SubClass 类中的 method 方法的参数是相同的 String 类型，还是 List<? extends Number> 类型，都是正确的覆盖方式。需要注意子类型和父类型的方法在类型擦除之后类型签名相同但不存在覆盖关系的情况。比如，SuperClass 类中的 method 方法的参数类型是 List<? extends Number>，而 SubClass 类中对应方法的参数类型是 List<? extends Integer>，会出现编译错误。这是因为子类和父类中的同名方法的类型签名不同，而且父类型的方法在类型擦除之后的类型签名不同于子类型的方法的原始类型签名，因此不存在覆盖关系，但是两个方法在类型擦除之后的类型签名都是“method(List)”，编译器无法判断哪个方法应该被调用。

在父类型和子类型中有一个或两个都是泛型类型时，参数类型是否相同的判断变得更加复杂。这是因为在方法类型签名中可能使用泛型类型定义中的形式类型参数。下面按照父类型和子类型是否为泛型类型的 4 种情况进行讨论。

1. 父类型和子类型均为非泛型类型

第一种情况是父类型和子类型都是非泛型类型，即不包含形式类型参数。

对于父类型中的非泛型方法，子类型中类型签名相同的非泛型方法可以进行覆盖。但是子类型中的泛型方法无法覆盖父类型中的非泛型方法。这是因为这种方式不可能满足前面提到的覆盖方法时关于类型签名的两个条件中的任何一个：非泛型方法和泛型方法的类型签名不可能相同，因为泛型方法的类型签名中包含形式类型参数；进行类型擦除也不会使父类型中的非泛型方法的类型签名发生改变。

对于父类型中的泛型方法，子类型中的非泛型方法可以对其进行覆盖，只要子类型中的非泛型方法的类型签名与父类方法在类型擦除之后的类型签名相同即可。代码清单 12-20 给出了子类型中的非泛型方法覆盖父类型中的泛型方法的示例。父类型中的 method 方法的形式类型参数没有上界，因此在类型擦除后的类型为 Object，与子类型中方法的参数类型相同。如果把父类型中的方法声明改为“<T extends Number> void

method(T obj)”, 那么子类型的方法就不再覆盖父类型中的方法。这是因为父类型中方法在类型擦除之后的参数类型变为 Number, 不同于子类型对应方法的参数类型 Object。

代码清单 12-20 子类型中的非泛型方法覆盖父类型中的泛型方法的示例

```
class SuperClass {
    public <T> void method(T obj) {
    }
}

class SubClass extends SuperClass {
    public void method(Object obj) {
    }
}
```

对于父类型中的泛型方法, 子类型中的泛型方法也可以进行对其进行覆盖, 要求子类型中的方法与父类型中的方法的类型签名相同。由于形式类型参数只是实例化时使用的真实类型的占位符, 在考虑方法类型签名时, 要根据形式类型参数可能的取值范围来进行判断, 判断的依据是不会破坏类型安全性。代码清单 12-21 给出了子类型中的泛型方法覆盖父类型中的泛型方法的示例。两个方法的类型签名中的形式类型参数都表示的是继承自 Object 类的所有类型, 因此两个类型签名是相同的。包含通配符的形式也可以用类似的方式来进行判断。

代码清单 12-21 子类型中的泛型方法覆盖父类型中的泛型方法的示例

```
class SuperClass {
    public <T> void method(T obj) {
    }
}

class SubClass extends SuperClass {
    public <S> void method(S obj) {
    }
}
```

2. 父类型为非泛型类型, 子类型为泛型类型

第二种情况是父类型为非泛型类型, 子类型为泛型类型。这种情况与第一种情况的差别在于子类型中的方法可以使用在类型定义时声明的形式类型参数。如果子类型中的方法不使用在类型定义时声明的形式类型参数, 则与第一种情况相同。如果子类型中的方法使用了在类型定义时声明的形式类型参数, 则无法覆盖父类型中的对应方法。代码清单 12-22 中的代码是无法通过编译的, 原因在于子类中的 method 方法并没有覆盖父类中的对应方法, 但是在类型擦除之后两者的类型签名是相同的。没有进行覆盖的原因是子类方法的类型签名是不确定的, 并不能保证与父类型中的方法签名保持一致, 比如使

用 GenericeSubClass<String> 声明的子类中的 method 方法的参数类型是 String，与父类型中的参数类型 Object 是不相同的。

代码清单 12-22 在泛型类型的方法声明中使用形式类型参数造成无法覆写的示例

```
class SuperClass {
    public void method(Object obj) {
    }
}

class GenericeSubClass<S> extends SuperClass {
    public void method(S obj) {
    }
}
```

3. 父类型为泛型类型，子类型为非泛型类型

第三种情况是父类型为泛型类型，子类型为非泛型类型。这种情况与第一种情况的不同在于父类型中的泛型方法可能使用在泛型类型定义时声明的形式类型参数。子类型中的非泛型方法可以覆写父类型中的泛型方法，这是因为子类型在继承父类型时需要声明父类型中形式类型参数的实际类型。因此父类型中的方法的类型签名实际上是固定的。代码清单 12-23 给出了相关的示例。父类型的方法 method 使用了形式类型参数 T，在子类型继承父类型时声明了父类型使用的实际类型是 Number，所以在父类型中定义的 method 方法的实际参数类型是 Number。在子类型中定义了参数类型为 Number 的 method 方法，因此子类型中的方法覆写了父类型中的对应方法。子类型中的泛型方法无法覆写父类型中使用了泛型类型定义的形式类型参数的方法，这是因为不能保证类型签名总是相同的。

代码清单 12-23 子类型中的非泛型方法覆写父类型中的泛型方法的示例

```
class GenericeSuperClass<T> {
    public void method(T obj) {
    }
}

class SubClass extends GenericeSuperClass<Number> {
    public void method(Number obj) {
    }
}
```

4. 父类型和子类型均为泛型类型

最后一种情况是父类型和子类型都是泛型类型。这种情况的复杂性在于父类型和子类型中的泛型方法都可以使用泛型类型的形式类型参数。由于在实例化时使用的实际类型可能有多种选择，这使对于是否覆写的判断变得复杂。这种情况的解决办法是判断实

际类型是否兼容，并且是否会带来类型安全问题。代码清单 12-24 给出了简单的示例。父类型和子类型中的泛型方法 method 都使用了类型定义中的形式类型参数。在对子类型实例化时，形式类型参数 S 的值同时也是父类型中形式类型参数 T 的值，因此两个类型的 method 方法的类型签名实际上是相同的。

代码清单 12-24 父子类型都是泛型类型时的方法覆写的示例

```
class GenericeSuperClass<T> {
    public void method(T obj) {
    }
}

class GenericeSubClass<S> extends GenericeSuperClass<S> {
    public void method(S obj) {
    }
}
```

由于泛型类型的存在，在有些情况下，子类型和父类型中方法的覆写关系并不是非常清晰的，需要仔细判断。一个典型的情况是泛型类型中的形式类型参数包含上界时。代码清单 12-25 给出了一个示例。子类型中的 method 方法的类型签名与父类型中对应方法的签名是不同的，子类型中方法的参数类型固定为 Number，而父类型中的参数类型可能是 Number 类及其子类型，但是子类型中的方法仍然覆写了父类型中的方法，这是因为这种覆写关系不会引起类型安全问题，当使用者通过父类型对象的引用调用 method 方法时，所有可能的参数对象值对于子类型的方法来说都是合法的。

代码清单 12-25 与泛型相关的复杂的方法覆写关系的示例

```
class GenericeSuperClass<T> {
    public void method(T obj) {
    }
}

class GenericeSubClass<S extends Number> extends GenericeSuperClass<S> {
    public void method(Number obj) {
    }
}
```

12.7.2 覆写对返回值类型的要求

在判断是否进行覆写的三个条件中，最复杂的是之前介绍过的方法的类型签名，下面介绍返回值类型需要满足的条件。子类型中方法的返回值类型必须可以替代父类型中对应方法的返回值类型。当使用者通过父类型的对象引用来调用一个被覆写的方法时，所得到的实际是子类型方法的返回结果。而使用者是通过父类型中方法的返回值类型来引用方法的返回结果，这就要求子类型中方法的实际返回结果的类型可以替代父类型中

方法声明的返回值类型。比如，父类型中方法的返回值类型是 Object，子类所覆写的方法的返回值类型可以是 String，这是因为 String 类型可以替代 Object 类型，反之则不行。

子类型中方法的返回值类型可以替代父类型中对应方法的返回值类型的情况比较多，典型的情况包括两个方法的返回值类型完全相同，以及前者是后者的子类型。在进行判断时可以使用在介绍类型系统中提到的泛型类型之间的父子类型的关系。代码清单 12-26 给出了使用包含通配符的参数化类型作为返回值类型的示例。父类型中 method 方法的返回值类型是 List<? extends Serializable>，而子类型中方法的返回值类型实际上是 List<? extends Number>。由于 List<? extends Number> 是 List<? extends Serializable> 的子类型，因此这是一个正确的方法覆写。

代码清单 12-26 使用包含通配符的参数化类型作为返回值类型的示例

```
class GenericeSuperClass<T> {
    public List<? extends Serializable> method() {
        return null;
    }
}

class GenericeSubClass<S extends Number> extends GenericeSuperClass<S> {
    public List<S> method() {
        return null;
    }
}
```

12.7.3 覆写对异常声明的要求

最后一个要求是子类型的方法声明中不能抛出父类型中对应方法没有声明的受检异常。对于子类型中的方法所声明的每个受检异常，父类型中的方法都需要声明抛出该受检异常或其父类型异常。当使用者通过 try-catch 语句来捕获父类型方法中抛出的异常时，所能捕获的只是父类型的方法所声明的异常。如果该方法被子类型所覆写，而子类型又抛出了新的受检异常，会导致编译错误。所以方法覆写要求抛出的受检异常类型也是兼容的。

12.7.4 重载

当方法类型签名、返回值类型和声明的受检异常这三方面的条件都满足之后，子类型中的方法可以覆写父类型中的对应方法。当两个方法的名称相同，但是方法类型签名不满足方法覆写的条件时，这两个方法是不同的重载形式。与覆写不同的是，重载可以发生在同一类型中。在判断是否为重载形式时，只会考虑方法的类型签名，返回值类型和声明的受检异常不在考虑的范围内。如果两个方法的类型签名满足覆写的要求，那么

两个方法不可能是重载的关系。如果这两个方法由于返回值类型或声明的异常类型的不兼容造成无法满足覆写的要求，则会产生编译错误。因为在这种情况下，这两个方法既不存在覆写关系，又不存在重载关系。编译器无法确定应该调用哪个方法。

在一个类型中可能包含同一方法的不同重载形式，包括类型中直接定义的及从父类型中继承下来的。对于一个方法调用，可能有多种重载形式都满足要求。比如，对两个重载形式“void method(Object)”和“void method(String)”来说，如果调用的方式是“method("Hello")”，则两种重载形式都是合法的，编译器会选择最合适的形式，即“void method(String)”；如果调用的方式是“method(0)”，则只有重载形式 void method(Object) 满足要求，编译器直接选择此重载形式。

在某些调用方式下，不同重载形式之间的差别并不明显，不存在一种形式优于另外一种的情况。编译器无法确定该使用哪种形式，会产生编译错误。如果子类型中的方法的目的是覆写父类型中的方法，则应该在子类型中的方法上加上“@Override”注解。添加了“@Override”注解之后，如果该方法并没有正确地覆写父类型中的方法，则会出现编译错误。

12.8 类型推断和 <> 操作符

在调用泛型方法时，对于其中包含的形式类型参数，通常不需要显式指定其实际类型，这是因为编译器会根据方法调用时的上下文信息自动推断对应的实际类型。如果在泛型方法中使用了泛型类型声明中的形式类型参数，方法中的实际类型可以根据参数化类型的实际类型来推断。比如创建了一个类型为 ArrayList<String> 的对象，在调用该对象的 add 方法时，编译器自动推断出参数类型为 String。编译器通过推断出的类型来拒绝不合法的方法调用。

如果泛型方法包含在非泛型类型中，或者方法声明中的形式类型参数与包含该方法的泛型类型的形式类型参数无关，那么可以通过两种方式来判断所使用的实际类型：一种方式是在方法调用时显式指定类型，另一种方式是由编译器根据方法调用上下文信息进行推断。代码清单 12-27 给出了一个非泛型类 TypeInference 中的泛型方法 method 的声明。

代码清单 12-27 用来说明类型推断的非泛型类中的泛型方法

```
public class TypeInference {
    public <T> T method(T obj) {
        return obj;
    }
}
```

1. 显式指定类型

在方法调用时可以显式指定类型。在代码清单 12-28 中，虽然方法调用时的参数的

类型是 String，但是通过显式的类型声明“<Serializable>”，方法调用时形式类型参数所表示的实际类型是 Serializable，而不是 String。

代码清单 12-28 方法调用时显式指定类型的示例

```
TypeInference typeInference = new TypeInference();
typeInference.<Serializable>method("Hello");
```

在显式指定类型时，方法实际参数的静态类型不能与指定的类型发生冲突。在代码清单 12-29 中，在调用方法时显式指定了形式类型参数的类型为 String，但是实际参数的静态类型为 Object，无法通过自动类型转换与 String 类型兼容，因此代码出现编译错误。虽然变量 str 的运行时类型是 String，但是编译器在处理时只考虑静态类型。

代码清单 12-29 形式类型参数的指定类型与实际参数的静态类型发生冲突的示例

```
TypeInference typeInference = new TypeInference();
Object str = "Hello";
typeInference.<String>method(str); // 编译错误
```

2. 类型推断

在大多数情况下并不需要显式地指定类型，编译器可以根据调用的上下文信息进行有效的推断。类型的自动推断有两种方式：第一种是根据方法调用时的实际参数的静态类型来进行推断，另一种是当方法调用的结果被赋值给另外一个变量时，可以根据该变量的静态类型进行推断，其中第一种方式的优先级较高。在代码清单 12-30 的方法调用中，实际的类型由调用时参数的静态类型来确定，为 String 类型。

代码清单 12-30 基于实际参数的静态类型的类型推断的示例

```
TypeInference typeInference = new TypeInference();
typeInference.method("Hello");
```

在有些方法中，形式类型参数不出现在参数列表中，而是出现在返回值类型中。如果方法调用的结果被赋值给另外一个变量，则根据此变量的静态类型来推断实际类型。代码清单 12-31 中给出了 TypeInference 类中的另外一个泛型方法 createList 及其调用方式。在调用 createList 方法时，无法从参数推断出类型信息。由于方法调用的结果被赋值给一个 List<Integer> 类型的变量，因此实际的类型可以从该变量的类型进行推断，为 Integer 类型。

代码清单 12-31 基于赋值操作的变量类型的类型推断的示例

```
public <T> List<T> createList() {
    return new ArrayList<T>();
}

TypeInference typeInference = new TypeInference();
List<Integer> list = typeInference.createList();
```

在进行类型推断时只会考虑参数类型和接受赋值操作结果的变量类型这两种信息。代码清单 12-32 中给出了两种不同的调用方式。两种方式的作用都是把 `createList` 方法的调用结果作为参数传递给 `method` 方法。两种方式的区别在于最后得到的对象的类型不同。通过第一种方式调用 `createList` 方法时，编译器无法推断出实际使用的类型，因此只能使用 `Object` 作为实际类型；第二种方式通过一个临时变量来保存 `createList` 方法的调用结果，由于赋值操作的存在，编译器可以推断出实际类型为 `Integer`。

代码清单 12-32 通过临时变量来允许编译器进行类型推断的示例

```
List<Object> list1 = typeInference.method(typeInference.createList());  
  
List<Integer> list2 = typeInference.createList();  
List<Integer> list3 = typeInference.method(list2);
```

3. <> 操作符

Java SE 7 把类型推断从方法调用扩展到了对象创建中，即增加了“<> 操作符”(diamond operator)。在 Java SE 7 之前，创建一个泛型类型的对象总是需要显式指定实际的类型，在对象创建完成后，通常有一个类型兼容的对象引用指向它。一般的做法如代码清单 12-33 所示，泛型类型声明在构造方法的类型声明和创建的对象的引用类型声明中同时出现。这两者在很多情况下是相同的，产生了不必要的代码重复。

代码清单 12-33 创建泛型类型的一般做法

```
List<String> list = new ArrayList<String>();  
Map<List<? extends Number>, Map<String, Long>> map = new HashMap<List<? extends  
Number>, Map<String, Long>>();
```

Java SE 7 对这个对象创建形式进行了简化，在调用构造方法时不再需要显式声明类型，直接使用“<>”来代替，具体的类型通过对对象引用的类型来进行推断。这种做法与上面提到的调用方法时的类型推断机制在本质上是相同的。简化之后的创建方式如代码清单 12-34 所示。

代码清单 12-34 使用 <> 操作符简化泛型类型的对象创建

```
List<String> list = new ArrayList<>();  
Map<List<? extends Number>, Map<String, Long>> map = new HashMap<>();
```

创建对象可以看成是一种通过构造方法完成的特殊的方法调用形式，因此，之前提到的调用方法时的类型推断机制对构造方法也是适用的。在进行类型推断时，优先考虑调用构造方法的实际参数的静态类型，再考虑对象引用的静态类型。当可能出现错误时，显式指定类型来避免类型推断可能出现的问题。

12.9 泛型与反射 API

在第 2 章中对反射 API 进行过详细的介绍。通过反射 API 所获取的是程序运行时的信息。泛型类型和泛型方法相关的信息在源代码中虽然存在，但是在编译时已经被擦除。在实际开发中，仍然有在运行时获取泛型相关信息的需求。为了满足这种需求，随着泛型的引入，J2SE 5.0 对 Java 字节代码的格式进行了修改，把泛型相关的信息添加到字节代码中。不过字节代码中的泛型相关的内容只是提供相关的信息，不会对字节代码的运行造成影响。在程序中可以使用反射 API 来获取这些信息。

结合第 8 章对 Java 字节代码格式的介绍，泛型相关的信息是作为类、方法和域的可选属性来保存的。对应的属性名称是“Signature”。这个属性的值是一个表示类或接口、方法或域的泛型类型签名的字符串。类或接口的签名由形式类型参数、父类型的签名和所实现的接口的签名组成；方法的签名由形式类型参数、参数类型签名、返回值类型签名和声明的受检异常类型签名组成；域的签名是其引用的一般类型、数组类型或形式类型参数的签名。使用 javap 工具分析代码清单 12-35 中的泛型类对应的字节代码，可以得出 GenericSignature 类的类型签名是“<S:Ljava/lang/Number;>Ljava/lang/Object;”，由形式类型参数 S、上界类型“Ljava/lang/Number;”和父类“Ljava/lang/Object;”组成；域 obj 的签名是“TS;”，其中“T”和“;”分别是形式类型参数的固定前缀和后缀，中间是形式类型参数“S”；方法 get 的签名是“(TS;)TS;”，其中参数类型和返回值类型都是表示形式类型参数“S”的“TS;”。

代码清单 12-35 用来说明泛型类型、方法和域的签名的示例

```
public class GenericSignature <S extends Number> {
    public S obj;
    public void set(S obj) {
        this.obj = obj;
    }
    public S get(S obj) {
        return obj;
    }
}
```

在不了解字节代码格式的情况下，可以通过反射 API 获取相同的信息。在使用反射 API 之前，先要获取泛型类型对应的 Class 类的对象，通过泛型类型的原始类型的 class 字面量可以得到这个对象，使用 Class 类的对象的 getField 和 getMethod 方法来获取表示域和方法的 Field 类和 Method 类的对象。

在 Field 类中，与泛型相关的方法是 getGenericType。方法 getGenericType 的返回值是 java.lang.reflect.Type 接口的实现对象。Type 接口是 Java 语言中所有类型都需要实现的接口。Type 接口只是一个标记接口，其中并没有任何方法。Class 类实现了 Type 接口，其他与泛型相关的类型接口 java.lang.reflect.GenericArrayType、java.lang.reflect.

ParameterizedType、java.lang.reflect.TypeVariable 和 java.lang.reflect.WildcardType 都继承自 Type 接口。在得到一个 Type 接口的实现对象时，需要先判断具体的类型，再使用具体类型中对应的方法。

GenericSignature 类中的域 obj 是一个类型参数。通过 getGenericType 方法返回的是 Type 接口的子接口 TypeVariable 的实现对象，表明这是一个类型参数。TypeVariable 接口提供了 3 个方法来获取与类型参数相关的信息：getName 方法返回的是类型参数在源代码中的名称；getGenericDeclaration 方法返回的是声明该类型参数的类或方法；getBounds 方法返回的是包含类型参数的所有上界的数组。

由于方法的参数、返回值和异常声明中都可以包含类型参数，Method 类中的 getGenericParameterTypes、getGenericReturnType 和 getGenericExceptionTypes 方法分别用来获取表示这些类型的 Type 接口的实现对象。如果一个方法的参数类型是参数化类型，那么所得到的类型实际上是 ParameterizedType 接口的实现对象。ParameterizedType 接口提供了 3 个方法来获取与参数化类型相关的信息：getActualTypeArguments 方法用来获取参数化类型的实际类型；getOwnerType 方法获取包含当前类型的父类型；getRawType 方法获取对应的原始类型。当在参数化类型中使用了通配符时，实际的类型是 WildcardType 接口的实现对象。WildcardType 接口中的 getLowerBounds 和 getUpperBounds 方法分别用来获取通配符的上界和下界。

代码清单 12-36 给出了使用反射 API 获取泛型信息的示例，围绕对类 Target 中的 create 方法的处理而展开。在得到表示 create 方法的 Method 类的对象之后，通过 getGenericParameterTypes 方法得到该方法的参数的类型是一个 TypeVariable 接口的实现对象。接着通过 getGenericReturnType 方法得到返回值类型是一个 ParameterizedType 接口的实现对象。该 ParameterizedType 接口的实现对象对应的参数化类型的实际类型是一个包含通配符的类型，由 WildcardType 接口的实现对象来表示。

代码清单 12-36 使用反射 API 获取泛型信息的示例

```

class Target <T> {
    public List<? extends Comparable<T>> create(T obj) {
        return null;
    }
}

public void reflect() throws Exception {
    Class<?> clazz = Target.class;
    Method method = clazz.getMethod("create", new Class<?>[] {Object.class});
    Type paramType = method.getGenericParameterTypes()[0];
    TypeVariable<?> typeVariable = (TypeVariable<?>) paramType;
    typeVariable.getName(); // 值为 T
    Type returnType = method.getGenericReturnType();
    ParameterizedType pType = (ParameterizedType) returnType;
    Type actualType = pType.getActualTypeArguments()[0];
}

```

```
Type[] bounds = ((WildcardType) actualType).getUpperBounds();
ParameterizedType boundType = (ParameterizedType) bounds[0];
boundType.getRawType(); //Comparable 接口的 Class 类对象
}
```

12.10 使用案例

使用泛型可以处理程序中会遇到的并行继承层次结构 (parallel inheritance hierarchy) 的问题。在并行继承层次结构中包含两组互相对应的类，其中一组类中两个类之间的继承关系也意味着另外一组类中的对应类之间也存在继承关系。比如，在一个用户界面组件库中，组件的抽象表示和实际的渲染逻辑通常是分开的，由不同的类来表示。每个组件类都有对应的进行渲染的类。如果一个组件类是另外一个组件类的子类，那么该组件对应的渲染类同时也是另外的组件对应的渲染类的子类。

代码清单 12-37 给出了一般的实现方式。Component 类是所有组件的父类，ComponentRenderer 类是所有进行组件渲染的类的父类。ComponentRenderer 类中的 render 方法用来对组件进行渲染操作。表示按钮的 Button 类继承自 Component 类，同时 Button 类对应的进行渲染的类 ButtonRenderer 也继承自 ComponentRenderer 类。在 ButtonRenderer 类的 render 方法实现中，由于参数类型是 Component 类，因此需要对参数对象的实际类型进行判断，以避免对错误类型的对象进行操作。在调用 ButtonRenderer 类的方法时，可以传递同样继承自 Component 类的 Label 类的对象作为参数。这在编译时是正确的，而运行时会出现错误。

代码清单 12-37 不使用泛型的并行继承层次结构的实现示例

```
abstract class Component {
}

abstract class ComponentRenderer {
    abstract void render(Component window);
}

class Button extends Component {
}

class Label extends Component {
}

class ButtonRenderer extends ComponentRenderer {
    void render(Component window) {
        if (window instanceof Button) {
            // 界面渲染
        }
        else {
    }}
```

```

        throw new IllegalArgumentException();
    }
}

public class NormalComponent {
    public void render() {
        ButtonRenderer renderer = new ButtonRenderer();
        renderer.render(new Button());
        renderer.render(new Label()); // 编译正确，运行时错误
    }
}

```

使用泛型可以限制方法调用时所接收的参数的类型，从而避免这个问题。代码清单 12-38 给出了使用泛型之后的实现方式。新的 ButtonRenderer 类的 render 方法只能使用 Button 类的对象作为参数，这就避免了使用错误类型的对象来调用此方法。类型检查在编译时完成，可以及早地发现问题。

代码清单 12-38 使用泛型的并行继承层次结构的实现示例

```

abstract class Component {
}

abstract class ComponentRenderer <C extends Component> {
    abstract void render(C component);
}

class Button extends Component {
}

class ButtonRenderer extends ComponentRenderer<Button> {
    void render(Button button) {
        // 界面渲染
    }
}

public class GenericComponent {
    public void render() {
        ButtonRenderer renderer = new ButtonRenderer();
        renderer.render(new Button());
    }
}

```

12.11 小结

虽然泛型直到 J2SE 5.0 才被添加到 Java 语言中，但是它仍然是 Java 语言中非常重要的特性。恰当地使用泛型，可以编写出类型安全的代码。编译器在编译时可以尽可能

早地发现潜在的类型安全问题，避免在运行时才发现问题。为了保持与遗留代码的兼容性，泛型由编译器在编译时通过类型擦除机制来实现，这种方式会对泛型的使用形成一些限制。

在使用泛型特性时的一条重要原则是不要忽视任何编译器给出的与类型安全相关的警告信息。这些警告信息说明了代码中可能存在类型安全问题，应该采用类型安全的做法来替代可能产生问题的做法，而不是简单地使用“`@SuppressWarnings`”注解来抑制这些警告信息的输出。

本章对泛型的基本概念、使用类型擦除的实现机制、上界和下界、通配符、类型系统、覆写和重载、类型推断和反射 API 等都做了详细的介绍。理解这些相关的概念可以更好地使用泛型。

第 13 章 Java 安全

安全性对于程序的重要性是不言而喻的。程序应该保证只有具备正确身份标识的用户才可以访问其所管理的数据，同时用户在访问数据时要满足由业务逻辑定义的访问控制权限的要求。不过，在程序的开发过程中，安全性与性能和可访问性等其他非功能性需求一样，在实现时的优先级通常低于程序的功能性需求。在程序开发过程中通常在比较靠后的阶段才考虑安全性相关的需求，这是因为开发人员通常更加专注于实现比较具体的功能模块。安全性等横切功能在每个具体功能模块的开发中都会遇到，一般的做法是把安全性相关的功能当成一个专门的模块，在程序的中后期专门实现。这种做法看似合理，但由于安全功能的重要性和复杂性，如果在程序开发的后期遇到了难以解决的安全性相关的问题，有可能需要对程序的架构做出比较大的调整，可能造成对前期已经完成的功能模块进行比较大的修改。正确的做法应该是在程序开发的早期就把安全性作为一个重要的模块进行考虑，并贯彻到整个程序的设计、开发和测试过程中。

在程序开发中经常会遇到两个与安全性相关的问题。第一个问题是忽视安全性。这个问题在桌面程序中的影响相对较小，因为使用桌面程序的用户通常是授权用户。如果在 Web 应用中忽视安全性，则可能带来非常严重的后果。第二个问题是忽视 Java 平台自身提供的与安全性相关的支持。Java 平台提供了良好的安全性相关的功能，不仅可以用来保护 Java 平台本身，还可以作为程序自身安全功能的实现基础。本章的主要内容是介绍 Java 平台的安全性相关的功能，以及如何在程序开发中使用这些功能。

13.1 Java 安全概述

构建安全的程序是一项复杂的系统工程，涉及操作系统、应用开发平台和程序自身等各个不同层次上的多个组件。安全性相关的约束在各个层次上都需要正确地应用。底层出现的安全问题会对上面层次的组件造成影响。如果由于底层操作系统上的漏洞导致系统管理员账号被窃取，那么在应用开发平台和程序中添加的安全性约束就形同虚设。程序自身也可以被划分成多个不同的层次，在每个层次的实现中，不应该盲目依靠其他层次的安全性约束。在一个 Web 应用中，在浏览器端和服务器端都需要进行安全性相关的约束检验，只依靠浏览器端的检验是不够的。正因为如此，在程序中的各个部分都会看到与安全性相关的代码，一方面要保证安全性的约束检验在每个需要的地方都被正确添加，另外一方面要避免安全性约束的检查代码的出现对阅读和理解程序中的业务逻辑实现代码造成影响。对于这种横切的需求，使用面向方面编程的思想是一个不错的选择。

安全性相关的内容大概可以分成 3 个部分：认证（authentication）、授权（authorization）和审计（audit）。认证的作用是验证系统的使用者具有合法的身份标识。这个身份标识可以由系统进行分配，也可以由用户自行注册。每个身份标识通常与安全凭证关联在一起，如用户注册时填写的密码。如果在认证过程中提供的身份标识和安全凭证是正确的，系统认为当前用户是认证时所声称的用户。通过认证并不意味着当前用户所执行的任何操作都是合法的，很多系统对用户的权限有自己的分配方式，即对用户进行授权。普通用户和系统管理员所能执行的操作肯定是不同的，因此在执行操作之前，需要根据系统当前的权限控制设置来进行判断，系统会拒绝不合法的操作。最后一个功能是审计。对于与安全相关的操作，如认证和授权等操作，都应该保留详细的日志来记录。这么做的好处是在发现安全问题时可以查找相关历史记录来进行追踪，还可以从用户的访问模式中找到潜在的安全漏洞。对于审计相关的功能，本章并没有进行详细的介绍。一般来说，只需要使用 Java 提供的日志 API 在合适的地方进行记录即可。

Java 平台所提供的对构建安全程序的支持是很丰富的。由于版本更新的历史原因，Java 平台的安全相关 API 分散在不同的组件中，包括进行认证和授权的 Java 认证和授权服务（Java Authentication and Authorization Service，JAAS）、提供加密和解密相关功能的 Java 密码框架（Java Cryptography Architecture，JCA）、处理公钥和数字证书验证的公钥基础设施（Public Key Infrastructure，PKI）及使用安全套接字连接进行通信的 Java 安全套接字扩展（Java Secure Socket Extension，JSSE）。这些不同的 API 都使用类似的服务提供者架构，可以很方便地进行扩展。开发人员既可以使用 Java 平台提供的默认实现，也可以使用由第三方提供或自己开发的实现。

13.2 用户认证

用户认证是用户向系统表明身份并由系统进行验证的过程。系统中的某些功能是认证用户才可以使用的。为了使用这些功能，用户需要先向系统表明身份，然后系统进行必要的身份认证。验证成功之后，当前用户变为认证用户，具备对应的基本权限。

13.2.1 主体、身份标识与凭证

在 Java 安全中用术语“主体（subject）”来表示访问请求的来源。主体是一个抽象的概念，可以表示任何实体。主体的具体含义由系统本身来确定，通常的理解是把主体看成系统的用户。这些用户可以是具体的人，也可以是其他程序。比如，对一个电子商务网站来说，用户可能是购买商品的消费者，也可能是合作商家，还可能是搜索引擎的抓取爬虫程序。

一个主体可以有多个不同的身份标识（principal）。身份标识是主体的具体表示。最常见的身份标识是系统分配或用户自己注册的用户名。在有的系统中，一个用户可以有多个身份标识。比如，用户注册的用户名和手机号码等都可以作为用户认证时的身份标识。

除了身份标识之外，一个主体还可以有公开或私有的安全相关的凭证（credential），包括密码和密钥等。最典型的凭证是用户注册时设置的密码。凭证是用户拥有一个身份标识的依据。

Java API 中表示主体的是 javax.security.auth.Subject 类，表示用户身份标识的是 java.security.Principal 接口。一个主体可以与多个身份标识关联。通过 Subject 类的 getPrincipals 方法可以得到包含当前主体关联的所有 Principal 接口的实现对象的集合。主体所关联的凭证信息分成公开和私有两类。公开凭证包括公钥等，通过 Subject 类的 getPublicCredentials 方法获取所包含的公开凭证对象的集合；私有凭证包括密码和私钥等，通过 Subject 类的对应方法 getPrivateCredentials 来获取。对于身份标识和凭证，Subject 类都是使用 java.util.Set 接口来管理，通过使用返回的 Set 接口的实现对象中的方法来完成对身份标识及凭证的添加和删除等操作。Principal 接口中的重要方法是 getName，用来返回 String 类型的身份标识的名称。Principal 接口的具体实现类可以为该身份标识设置不同的含义，比如代码清单 13-1 中的 UserPrincipal 类中使用的身份标识是用户名。Subject 类对于凭证的具体表示方式并没有做出限制，任何对象都可以作为凭证。

代码清单 13-1 Principal 接口实现类的示例

```
public class UserPrincipal implements Principal {
    private final String username;
    public UserPrincipal(String username) {
        this.username = username;
    }
    public String getName() {
        return username;
    }
}
```

为了防止程序中的其他代码对 Subject 类的对象进行错误的修改，可以在添加必要的身份标识和凭证信息之后，调用 setReadOnly 方法把 Subject 类的对象设为只读状态。一旦被设为只读状态，该 Subject 类的对象无法被重新设置为可修改状态。

13.2.2 登录

典型的用户认证过程通过登录操作来完成。用户把所持有的身份标识和凭证信息提供给系统，由系统进行相关的身份验证操作。登录成功之后，一个主体中就具备了相应身份标识。Java 提供了一个可扩展的登录框架，使应用开发人员可以很容易地定制和扩展与登录相关的逻辑。Java 提供的可扩展的登录框架由登录上下文、登录模块和登录配置等几个部分组成。这几个部分之间相互协作，完成整个登录过程。下面将对这几个部分进行详细的介绍。

1. 登录上下文

登录相关的 API 在 `javax.security.auth.login` 包中，整个登录过程由 `javax.security.auth.login.LoginContext` 类的对象来负责启动和管理。每个 `LoginContext` 类的对象一般只用来对一个 `Subject` 类的对象进行认证。当认证成功之后，通过 `LoginContext` 类的对象的 `getSubject` 方法可以获取包含正确身份标识和凭证信息的 `Subject` 类的对象。程序的其他部分可以通过查询该 `Subject` 类的对象是否存在来判断当前用户是否通过了认证。对于认证之后的 `Subject` 类的对象，桌面程序通常把它作为全局对象，Web 应用通常使用 `ThreadLocal` 类来对它进行封装。

`LoginContext` 类实现的是一个抽象的用户登录过程。`LoginContext` 类本身并不负责实际登录方式的实现，而只是负责协调整个登录过程。具体的登录方式由 `javax.security.auth.spi.LoginModule` 接口的实现类来表示，比如某个 `LoginModule` 接口的实现类会负责把当前用户给出的身份标识和凭证信息与数据库中的记录进行比对。`LoginModule` 接口的实现类被称为登录模块。在一个登录过程中，可能使用多种不同的登录方式。这些登录方式之间可以相互配合。比如，在电子商务网站中，可以使用基本的用户名和密码的登录方式，也可以使用动态手机验证码作为登录方式，还可以两种方式配合起来使用。开发人员通过提供自己的 `LoginModule` 接口的实现类来封装不同的登录逻辑。一个程序所使用的登录方式可能随着版本的更新而不断变化，比如，最早只提供通过用户名和密码来进行登录的功能，后来加上了手机短信验证码以及数字证书等附加方式。为了满足这种需求，对于程序中使用的登录方式，可以在不修改程序代码的情况下，通过修改配置的方式来进行更新。登录方式的配置由 `javax.security.auth.login.Configuration` 类的对象来表示。在一个 `Configuration` 类的对象中可以包含多种不同名称的配置。在登录过程中，登录模块可能需要与用户或其他组件进行交互来获取相关的信息，比如要求用户输入进行认证的用户名和密码等。这种交互方式通过 `javax.security.auth.callback.CallbackHandler` 接口的实现类来表示。

在创建 `LoginContext` 类的对象时可以使用多种不同的构造方法。必需的参数是登录时使用的 `Configuration` 类的对象中包含的配置的名称。除配置名称之外，在调用构造方法时还可以提供其他可选的参数。调用者可以提供 `Subject` 类的对象，如果不提供该对象，那么 `LoginContext` 类的对象会创建一个新的 `Subject` 类的对象，并在登录过程中使用。该 `Subject` 类的对象也是调用 `getSubject` 方法时的返回值。如果调用构造方法时提供了 `CallbackHandler` 接口的实现对象，那么 `LoginContext` 类的对象使用这个对象来处理与用户或其他组件的交互；否则 `LoginContext` 类的对象通过系统属性来查找默认的 `CallbackHandler` 接口的实现类，并创建其对象来使用。如果调用构造方法时提供了 `Configuration` 类的对象，那么 `LoginContext` 类的对象直接使用该对象来查找配置信息；否则 `LoginContext` 类的对象使用 `Configuration` 类的静态方法 `getConfiguration` 来获取当前的默认配置对象。

LoginContext 类中的 login 和 logout 方法分别用来启动登录过程和注销登录。在 login 方法的实现中，先从 Configuration 类的对象中获取当前配置的 LoginModule 接口的实现对象，以及相关的配置参数信息。接着进行两个阶段的登录过程。在第一个阶段中，每个 LoginModule 接口的实现对象的 login 方法都会被调用，依次进行身份验证。等所有登录模块都完成之后，由当前 LoginContext 类的对象来确定整个登录过程是否成功，进入第二个阶段的处理。在第二个阶段中，如果整个登录过程是成功的，那么每个 LoginModule 接口的实现对象的 commit 方法会被调用；否则 abort 方法会被调用。在注销登录过程中，所有 LoginModule 接口的实现对象的 logout 方法都会被调用。

2. 登录模块

LoginModule 接口的实现类用来提供程序自身的登录逻辑。具体的实现类的对象由当前登录过程对应的 LoginContext 类的对象负责创建。LoginContext 类的对象会在合适的时机调用 LoginModule 接口中的不同方法来实现完整的登录流程。每个 LoginModule 接口的实现类都应该提供一个不带任何参数的构造方法，使 LoginContext 类的对象可以创建出所需的 LoginModule 接口的实现对象。

在 LoginModule 接口的实现对象被创建出来之后，LoginModule 接口的 initialize 方法会先被调用。调用方法时的参数中包含了由当前 LoginContext 类的对象所提供的各种上下文信息，包括等待认证的 Subject 类的对象、用来进行用户交互的 CallbackHandler 接口的实现对象、不同 LoginModule 接口的实现对象之间的共享数据，以及额外的配置参数值。后面两个参数都是 java.util.Map 接口的实现对象，可以看成数据的简单存储方式。有些登录模块在登录过程中依赖其他模块所提供的数据，可以通过 initialize 方法的第三个参数对象来进行传递。在一个登录模块中进行修改，在另外一个登录模块中获取。有些登录模块提供的是比较通用的实现方式，在使用之前需要进行适当的配置，这些配置由 Configuration 类的对象来负责读取。LoginModule 接口实现类可以通过 initialize 方法的第四个参数来获取配置参数的实际值。比如一个通用的基于数据库的登录模块，在使用时需要配置数据库连接和表的信息。一般来说，在 initialize 方法的实现中会把通过调用参数传递的相关信息保存下来，提供给 LoginModule 接口实现类中的其他方法来使用。

登录过程的第一个阶段，调用的是 LoginModule 接口的 login 方法，在这个方法的实现中进行必要的身份认证。身份认证的实现方式与具体的登录功能相关，比如先获取用户输入的用户名和密码，再通过数据库、网络操作或其他方式来完成认证。如果认证成功，login 方法会把认证相关的信息保存起来；如果认证失败，则抛出 javax.security.auth.login.LoginException 异常或其子类型异常。登录过程的第二个阶段调用的是 LoginModule 接口的 commit 或 abort 方法。如果当前 LoginContext 类的对象认为整个登录过程是成功的，LoginModule 接口的 commit 方法会被调用，用来把身份标识和凭证信息关联到主体上；如果整个登录过程是失败的，LoginModule 接口的 abort 方法会被调用，

用来清除之前保存的认证相关信息。在用户注销登录时，`LoginModule` 接口的 `logout` 方法会被调用。由当前登录模块在 `commit` 方法实现中关联到 `Subject` 类的对象中的身份标识和凭证信息，需要在 `logout` 方法实现中清除。

在很多情况下，`LoginModule` 接口的实现中都需要通过与用户或其他组件进行交互来获取认证所需的相关信息。这种交互方式是通过 `CallbackHandler` 接口及其关联的 `javax.security.auth.callback.Callback` 接口来完成的。在 `CallbackHandler` 接口中只有一个方法 `handle`，其参数是 `Callback` 接口的数组。`Callback` 接口本身只是一个标记接口，并没有提供任何需要实现的方法。在 `CallbackHandler` 接口的 `handle` 方法的实现中，需要根据每个 `Callback` 接口实现对象的具体类型来采取不同的处理方式。`javax.security.auth.callback` 包提供了一些可以直接使用的 `Callback` 接口的实现类，比如，`TextInputCallback` 类和 `TextOutputCallback` 类分别用来进行文本信息的输入和输出；`NameCallback` 类和 `PasswordCallback` 类分别用来进行用户名和密码的输入；`LanguageCallback` 类来进行语言的选择；`ChoiceCallback` 类用来提供一组可供选择的选项；`ConfirmationCallback` 类用来要求用户对操作进行确认。这些 `Callback` 接口的实现类只是信息的简单的保存方式，具体的处理由 `CallbackHandler` 接口的实现类来完成。

`CallbackHandler` 接口的实现类选择用适合的方式来处理不同类型的 `Callback` 接口实现对象。比如在处理文本输入时，可以选择由用户在控制台输入，或提供一个图形用户界面给用户；在处理文本输出时，把信息显示在控制台或以对话框的方式来显示。

代码清单 13-2 给出了 `CallbackHandler` 接口的一个实现类 `TextCallbackHandler`。在 `CallbackHandler` 接口的 `handle` 方法中，只对 `TextInputCallback` 类和 `TextOutputCallback` 类的对象进行处理。对于其他类型的 `Callback` 接口的实现对象，抛出 `javax.security.auth.callback.UnsupportedCallbackException` 异常。对于 `TextInputCallback` 类的对象的处理方式是在控制台输出提示信息之后，接收用户的输入作为文本的内容；对于 `TextOutputCallback` 类的对象的处理方式是根据消息的类别创建文本内容，并在控制台输出。

代码清单 13-2 `CallbackHandler` 接口的实现类的示例

```
public class TextCallbackHandler implements CallbackHandler {
    public void handle(Callback[] callbacks) throws IOException,
        UnsupportedCallbackException {
        for (Callback callback : callbacks) {
            if (callback instanceof TextInputCallback) {
                TextInputCallback textInputCallback = (TextInputCallback)
                    callback;
                System.out.print(textInputCallback.getPrompt());
                textInputCallback.setText((new BufferedReader(new
                    InputStreamReader(System.in))).readLine());
            }
            else if (callback instanceof TextOutputCallback) {
                TextOutputCallback textOutputCallback = (TextOutputCallback)
                    callback;
            }
        }
    }
}
```

```
String messageType = "";
switch (textOutputCallback.getMessageType()) {
    case TextOutputCallback.INFORMATION:
        messageType = "信息：";
        break;
    case TextOutputCallback.WARNING:
        messageType = "警告：";
        break;
    case TextOutputCallback.ERROR:
        messageType = "错误：";
        break;
}
System.out.println(messageType + textOutputCallback.getMessage());
}
else {
    throw new UnsupportedCallbackException(callback);
}
}
```

下面通过一个具体的 `LoginModule` 接口的实现类来说明登录模块的实现。该登录模块使用属性文件来保存用户的信息。属性文件中的每一个条目表示一条用户记录，条目的键表示的是用户名，而对应的值是密码。代码清单 13-3 给出了该 `LoginModule` 接口的实现类 `PropertiesFileBasedLoginModule` 的代码。在 `initialize` 方法中把调用参数中的 `Subject` 类的对象和 `CallbackHandler` 接口的实现对象都保存下来，供其他方法使用。在使用该登录模块时需要配置存储用户信息的属性文件的路径。在 `initialize` 方法中，从第 4 个调用参数中获取文件路径，再通过 `java.util.Properties` 类的对象来读取此文件的内容，将读取完成之后的 `Properties` 类的对象作为用户信息的内部存储方式。在 `login` 方法的实现中，先创建两个 `TextInputCallback` 类的对象与用户进行交互，获取用户输入的用户名和密码，再与属性文件中保存的用户信息进行比较。如果用户提供的信息无法通过认证，那么抛出 `LoginException` 异常或其子类型异常来声明认证失败；如果认证成功，则把用户名保存下来，并设置认证成功的内部标记。

在登录的第二阶段的 commit 方法实现中，根据调用 login 方法时记录的认证结果进行不同的处理。如果在 login 方法中的身份认证是成功的，则修改 Subject 类的对象，添加相关身份标识信息。添加的身份标识是代码清单 13-1 中的 UserPrincipal 类的对象。在 abort 方法中，清除在 login 方法中保存的相关信息。

在进行注销的 logout 方法中，需要在 Subject 类的对象中查找由当前登录模块在 commit 方法中添加的身份标识和凭证信息，并删除这些信息。LoginModule 接口的实现类只能删除由自己所添加的相关信息。

代码清单 13-3 基于属性文件的登录模块的实现

```

public class PropertiesFileBasedLoginModule implements LoginModule {
    private CallbackHandler callbackHandler;
    private Subject subject;
    private Properties props = new Properties();
    private boolean authSucceeded = false;
    private String authUsername = null;
    public void initialize(Subject subject, CallbackHandler callbackHandler,
        Map<String, ?> sharedState, Map<String, ?> options) {
        this.callbackHandler = callbackHandler;
        this.subject = subject;
        String propsFilePath = (String) options.get("properties.file.path");
        File propsFile = new File(propsFilePath);
        try {
            props.load(new FileInputStream(propsFile));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public boolean login() throws LoginException {
        TextInputCallback usernameInputCallback = new TextInputCallback("用户名：");
        TextInputCallback passwordInputCallback = new TextInputCallback("密码：");
        try {
            callbackHandler.handle(new Callback[] {usernameInputCallback,
                passwordInputCallback});
        } catch (Exception e) {
            throw new LoginException(e.getMessage());
        }
        String username = usernameInputCallback.getText();
        if (username == null || "".equals(username.trim())) {
            throw new AccountException("用户名为空！");
        }
        if (!props.containsKey(username)) {
            throw new AccountNotFoundException("该用户不存在！");
        }
        String password = passwordInputCallback.getText();
        if (password == null || "".equals(password.trim())) {
            throw new CredentialException("密码为空！");
        }
        if (!password.equals(props.get(username))) {
            throw new FailedLoginException("用户名和密码不匹配！");
        }
        authSucceeded = true;
        authUsername = username;
        return true;
    }

    public boolean commit() throws LoginException {
        if (authSucceeded) {
    
```

```

        this.subject.getPrincipals().add(new UserPrincipal(authUsername));
        authUsername = null;
        authSucceeded = false;
        return true;
    }
    return false;
}

public boolean abort() throws LoginException {
    authUsername = null;
    if (authSucceeded) {
        authSucceeded = false;
        return true;
    }
    return false;
}

public boolean logout() throws LoginException {
    Set<Principal> principals = subject.getPrincipals();
    Set<UserPrincipal> userPrincipals = subject.getPrincipals(UserPrincipal.
        class);
    for (UserPrincipal principal : userPrincipals) {
        if (principal.getName().equals(authUsername)) {
            principals.remove(principal);
            break;
        }
    }
    return true;
}
}

```

在 LoginModule 接口中，与登录和注销相关的 4 个方法 login、commit、abort 和 logout 有着相似的类型签名。这 4 个方法的返回值都是 boolean 类型，声明都抛出受检异常 LoginException。这 4 个方法实际上有 3 种不同的执行结果，分别是返回值为 true 或 false，以及抛出 LoginException 异常。当前的 LoginContext 类的对象会根据这些方法的返回值及是否抛出 LoginException 异常来确定整个登录过程的结果，因此在这 4 个方法的实现中需要根据不同的情况进行正确的处理。

对于 login 方法来说，返回 true 表明该登录模块所执行的身份认证操作是成功的，抛出 LoginException 异常或其子类型异常说明认证失败。在整个登录过程中，某些登录模块的使用可能是可选的，比如使用手机短信验证码进行认证的方式，如果用户没有提供手机号码，那么这个登录模块应该被忽略，只有在用户提供了手机号码时，该模块才生效。login 方法返回 false，说明该模块在当前登录过程中应该被忽略。在 login 方法中需要保存认证成功或失败的信息，但是不能修改 Subject 类的对象。

在 commit 方法实现中，认证成功后会把相关的身份标识和凭证信息关联到 Subject

类的对象上。在这个过程中也可能出现错误。如果在 login 方法中的认证操作成功，而 commit 方法执行中出现了错误，那么应该抛出 LoginException 异常；如果在 commit 方法的执行中没有出现错误，则返回值为 true。如果认证操作是失败的，则不管在 commit 方法的执行中是否出现错误，都直接返回 false。在 commit 方法的实现中需要把该过程是否成功的结果记录下来。

在 abort 方法中采用的处理方式取决于 login 方法中的身份认证是否成功，以及 commit 方法是否成功完成。如果认证成功，则不管 commit 方法是否成功完成，只要 abort 方法本身执行时没有出现错误，abort 方法总是返回 true；如果 abort 方法本身出现错误，那么 abort 方法直接抛出 LoginException 异常。如果认证失败，则不管 commit 和 abort 方法是否成功完成，abort 方法都直接返回 false。

在 logout 方法中，如果注销成功，那么返回值为 true；否则抛出 LoginException 异常。

3. 登录配置

创建出 LoginModule 接口的实现类之后，需要对登录模块的使用进行配置。登录模块的配置由 Configuration 类的对象来负责管理。在一个 Configuration 类的对象中，一般同时维护多个配置。每个配置都有自己的名称，由一组 javax.security.auth.login.AppConfigurationEntry 类的对象组成。每个 AppConfigurationEntry 类的对象表示一个登录模块配置项。每个配置项由 3 部分组成，分别是登录模块的 Java 类、登录模块的控制标记和额外的配置参数。配置中的每个登录模块会按照对应配置项出现的顺序依次执行。控制标记用以说明每个登录模块在整个登录过程中的作用，有必须、必要、充分和可选 4 种标记。对于声明为“必须”的登录模块，不论认证成功与否，在其之后出现的登录模块仍然会被执行。对于声明为“必要”的模块，如果认证成功，那么在其之后出现的登录模块会被执行；如果认证失败，则整个登录过程直接失败。对于声明为“充分”的登录模块，如果认证成功，那么整个登录过程直接变为成功状态，后面的登录模块不会被执行；如果登录失败，则继续执行后面的登录模块。对于声明为“可选”的模块，不管登录成功与否，后面的登录模块仍然会被执行。登录模块的额外配置参数是一个名值对列表，其中包含的内容会作为最后一个参数传递给 LoginModule 接口的 initialize 方法。

只有当所有声明为“必须”和“必要”的登录模块都认证成功时，整个登录过程才是成功的。如果配置了声明为“充分”的登录模块，并且该模块认证成功，那么只要求出现在该模块之前的声明为“必须”和“必要”的登录模块认证成功即可。如果没有配置声明为“必须”或“必要”的登录模块，则至少要有一个声明为“充分”或“可选”的登录模块认证成功，整个登录过程才是成功的。

开发人员可以使用 Configuration 类的默认实现，也可以继承 Configuration 类开发自己的配置管理类。默认的 Configuration 类的实现是基于文件的。配置文件的格式如代码清单 13-14 所示。“MyApp”是登录配置的名称。在一个文件中可以包含多个不同名

称的登录配置。文件中的每一行表示一个配置项。配置项的不同部分之间用空格分开。第一个部分是登录模块的 Java 类的全名。第二个部分是登录模块的控制标记。对于“必须”、“必要”、“充分”和“可选”等控制标记，分别用 required、requisite、sufficient 和 optional 来表示。第三个部分是登录模块的配置参数。每个配置参数是一个名值对，使用“=”分隔。多个配置参数之间使用空格分隔。

代码清单 13-4 默认登录模块的配置文件的示例

```
MyApp {
    com.java7book.chapter13.auth.PropertiesFileBasedLoginModule required
        properties.file.path="C:\\\\java7\\\\code\\\\chapter13\\\\user.properties";
}
```

在运行时只有一个 Configuration 类的对象起作用。通过 Configuration 类的静态方法 getConfiguration 可以获取这个 Configuration 类的对象。如果使用默认的配置方式，那么需要添加虚拟机启动参数 “java.security.auth.login.config” 来指定配置文件的路径。

把 LoginContext 类、LoginModule 接口和 Configuration 类结合起来，可以实现完整的登录过程。代码清单 13-5 给出了启动登录过程和进行注销操作的代码示例。

代码清单 13-5 完整的登录过程的示例

```
public class MyApp {
    private LoginContext loginContext;
    public MyApp() throws LoginException {
        TextCallbackHandler callbackHandler = new TextCallbackHandler();
        loginContext = new LoginContext("MyApp", callbackHandler);
    }

    public Subject login() throws LoginException {
        loginContext.login();
        return loginContext.getSubject();
    }

    public void logout() throws LoginException {
        loginContext.logout();
    }
}
```

13.3 权限控制

在一个主体通过了身份认证之后，程序中就具有了表示该主体的 Subject 类的对象。该 Subject 类的对象中包含了该主体的身份标识和凭证等信息。如果程序的实现中并不区分不同的用户，那么只进行身份认证就足够了。不过大多数程序会把用户划分成不同的组，每个组的用户可以执行的操作是不同的。在用户能够执行某些操作之前，需要先

检查当前用户是否具有相应的权限。在访问控制中两个重要的概念是权限和策略。权限用来表示允许执行某些操作的能力，而策略用来说明权限的分配方式。如果在策略中为某个主体分配了某个权限，那么该主体可以执行由该权限对应的操作。

13.3.1 权限、策略与保护域

下面介绍与访问控制权限相关的权限、策略和保护域的概念。

1. 权限

Java 安全中的权限由 `java.security.Permission` 类及其子类来表示。每个权限都有一个名称，该名称的具体含义与权限类型相关。在创建 `Permission` 类的对象时，需要提供这个名称。通过 `getName` 方法可以获取权限的名称。某些权限有与之关联的动作列表。这些权限通常用名称来表示所操作的对象，用动作列表来表示在该对象上所能执行的操作的集合。比如文件操作权限 `java.io.FilePermission` 类，它的名称是文件的路径，而它的动作列表包括读取、写入和执行等。通过 `getActions` 方法可以获取权限的动作列表。只包含名称的权限适合于描述简单的“是”或“否”类型的访问控制要求。如果分配了只包含名称的权限，就说明可以执行对应的操作，而不会对操作的类型进行进一步的细分。

`Permission` 类中最重要的是 `implies` 方法，它定义了权限之间的包含关系，是验证权限是否具备的基础。在分配权限时，不可能穷举所有可能的情况，只能从逻辑上进行判断。同一类型的权限的不同声明或不同类型的权限之间可能存在包含关系。以文件系统访问权限 `FilePermission` 类为例，如果一个主体拥有某个目录上的读取权限，那么该主体同样拥有对该目录下某个子目录或文件的读取权限。这种包含关系通过 `implies` 方法的实现来声明。`implies` 方法的参数是另外一个 `Permission` 类的对象。如果当前 `Permission` 类的对象在逻辑上包含了作为参数的 `Permission` 类的对象，则 `implies` 方法返回值为 `true`。如果“`A.implies(B)`”的返回值为 `true`，则拥有权限 A 的主体就自动拥有权限 B。对于 `FilePermission` 类来说，分配权限时通常只分配某个目录的权限，不可能为目录中的所有文件逐一分配权限。在 `FilePermission` 类的 `implies` 方法的实现中，如果当前 `FilePermission` 类的对象所对应的路径中包含了作为参数的 `FilePermission` 类的对象所对应的路径，则 `implies` 方法返回值为 `true`。

在实际开发中，通常需要对一组权限进行处理。Java 安全 API 中提供了两种不同的权限集合，一种是 `java.security.PermissionCollection` 类的对象，表示相同类别的权限；另外一种是 `java.security.Permissions` 类的对象，表示不同类别的权限。`Permissions` 类是 `PermissionCollection` 类的子类。在需要使用权限集合时，应该先调用 `Permission` 类的 `newPermissionCollection` 方法来获取一个 `PermissionCollection` 类的对象。有些 `Permission` 类的子类会通过覆写 `newPermissionCollection` 方法来提供自定义的权限集合的实现，这是为了保证 `PermissionCollection` 类的 `implies` 方法提供正确的权限包含语

义。如果 newPermissionCollection 方法的返回值为 null，则调用者可以使用自己的权限集合实现方式；否则，应该直接使用 newPermissionCollection 方法的返回值。通过 PermissionCollection 类的 add 方法可以向集合中添加新的 Permission 类的对象，而通过 implies 方法可以判断该权限集合是否包含另外一个 Permission 类的对象。Permissions 类的对象可以看成由多个 PermissionCollection 类的对象组成的不同类别的权限的集合。

Java 标准库提供了一些常用的 Permission 类的子类。类 java.security.AllPermission 表示的是所有的权限。AllPermission 类的 implies 方法的返回值总是 true，分配此权限相当于禁用了访问权限控制，因此需要谨慎使用 AllPermission 类。类 java.security.BasicPermission 可以作为自定义的 Permission 类的实现基础。BasicPermission 类的权限名称使用的是类似名称空间的形式，可以在名称的最后部分使用通配符，如“myapp.login”和“my.app.ui.*”等都是合法的权限名称。BasicPermission 类的 implies 方法会负责处理通配符。标准库提供的很多权限实现类都是 BasicPermission 类的子类。

2. 策略

策略是根据程序当前的访问控制权限设置来判断主体是否具有某个权限。判断的方式是基于显式声明的权限和权限之间的包含关系。类 java.security.Policy 是 Java 安全 API 中策略的表示。Policy 类的对象的使用方式类似于登录过程中使用的 Configuration 类的对象。在同一时间内，只能有一个 Policy 类的对象处于活动状态。与权限相关的决策都通过当前活动的 Policy 类的对象来处理。通过 Policy 类的静态方法 getPolicy 和 setPolicy 可以获取和设置该活动对象。

Java 平台提供的默认策略实现是基于文件来存储的。在 JRE 安装目录下的“lib/security/java.policy”文件是 Java 运行环境自身使用的策略文件。程序也可以提供自己的策略文件，通过虚拟机启动参数“java.security.policy”来指定文件的路径。策略文件的格式比较复杂，包含密钥、签名、主体身份标识和授权条目等内容，其中最重要的内容是授权条目。每一个授权条目表示的是显式声明的主体所具有的一个权限。在条目中包含权限的类名、名称和动作列表。代码清单 13-6 给出了策略文件的示例，以“permission”开始的每一行表示一个权限。

代码清单 13-6 默认策略文件的示例

```
grant {
    permission java.io.FilePermission "c:\\tmp", "write";
    permission java.util.PropertyPermission "user.*", "read, write";
};
```

由于策略文件的格式比较复杂，因此 JDK 中提供了可视化 policytool 工具对策略文件进行编辑。如果需要使用数据库或其他方式的存储策略，那么可以继承 Policy 类来提供自己的实现。Policy 类的对象的使用方式是调用其 implies 方法来判断参数中的权限是否符合策略的要求。不过，在进行访问控制权限检查时，一般不需要显式调用当前活动

的 Policy 类的对象的 implies 方法，而是使用更加方便的方法。

3. 保护域

保护域是 Java 安全机制中的基本概念。保护域是一个主体所能访问的对象的集合。程序通过策略中的声明把不同的权限分配给不同的主体，这些权限定义了主体所能访问的对象的集合。一般将 Java 平台分成两类保护域：一类是系统域，用来保护系统中的资源，如文件系统、网络连接、屏幕显示和键盘鼠标等；另一类是应用域，由应用根据需要来划分。

每个保护域中包含一组 Java 类、主体的身份标识和权限列表。当访问请求的来源是这些身份标识所对应的主体时，这一组 Java 类的对象实例自动拥有给定权限列表中的所有权限。保护域的权限既可以是固定的，也可以根据策略动态变化。类 java.security.ProtectionDomain 表示保护域，它的两个构造方法分别用来支持静态和动态的权限。在创建 ProtectionDomain 类的对象时需要提供 java.security.CodeSource 类的对象。CodeSource 类表示的是代码的来源，包含表示来源地址的 java.net.URL 类的对象及对该来源的代码进行验证的数字证书链或代码签名工具。比如从远程服务器下载到浏览器中执行的 Java Applet，它的代码来源是 Applet 中 jar 包的 URL。对于 Applet 的 jar 包，通常需要进行数字签名以保证来源是真实有效的。在创建 ProtectionDomain 类的对象时，除了 CodeSource 类的对象之外，还需要提供表示该保护域所具有的权限的 PermissionCollection 类的对象。如果只提供这两个参数，那么 ProtectionDomain 类的对象所表示的保护域中的权限是静态的，在创建后不会改变，在进行权限检查时也不会考虑当前活动的 Policy 类的对象。如果要求权限根据策略动态变化，那么需要提供额外的两个参数：一个参数是 ClassLoader 类的对象，另外一个参数是包含主体身份标识的 Principal 接口的实现对象的数组。使用这个构造方法创建出来的 ProtectionDomain 类的对象在进行权限检查时，不但要考虑构造方法的参数中给出的静态权限，还要考虑通过当前活动的 Policy 类的对象所赋予的权限。在具体使用中，ProtectionDomain 类的对象使用 implies 方法来检查一个 Permission 类的对象所对应的权限是否被该保护域所允许。

13.3.2 访问控制权限

在声明了程序在运行中应该满足的访问控制权限之后，需要在代码中显式地应用这些要求。比如某个方法只有具有了特定权限的主体才能调用，在该方法被执行之前，需要先检查当前调用上下文，判断主体是否具备这样的权限。如果没有权限，方法应该直接抛出相关的 java.security.AccessControlException 异常。这些访问控制权限与具体的程序相关，需要由开发人员在代码中显式地进行处理。

访问控制权限的检查由 java.lang.SecurityManager 类和 java.security.AccessController 类来共同完成。SecurityManager 类是 Java 早期版本中的实现方式，其中包含一些以“check”作为前缀的方法，通过这些方法可以进行相应的权限检查。比如 checkRead 方

法可以检查是否具有读取给定路径对应的文件的权限。对于程序本身的权限，可以通过继承 SecurityManager 类的方式来添加相应的检查方法。Java 标准库中的很多方法中都添加了使用 SecurityManager 类进行访问控制权限检查的代码。

对于 Java Applet 来说，SecurityManager 类所做的检查总是启用的，这是由 Applet 本身的不安全特性所决定的。对于在本地虚拟机上直接运行的程序，SecurityManager 类所做的检查默认是不启用的，这是因为这些程序被认为是安全的。如果希望对程序启用 SecurityManager 类的检查，那么可以通过虚拟机启动参数 “-Djava.security.manager” 来启用默认的安全管理器实现。如果程序使用了自定义的 SecurityManager 类的实现，那么可以使用类似 “-Djava.security.manager=com.java7book.chapter13.MySecurityManager” 的方式来指定自定义安全管理器的类名。使用 System 类的 setSecurityManager 方法可以在运行时设置需要使用的安全管理器的实现。在启用了安全管理器之后，会发现很多原先正常执行的代码都会抛出 AccessControlException 异常，这是因为 Java 标准库中的权限检查代码开始生效，这时程序要提供自己的策略文件来添加所需的权限。代码清单 13-7 给出了使用 SecurityManager 类来进行权限检查的示例，在进行文件写入操作之前先检查是否具有相应的权限。

代码清单 13-7 使用 SecurityManager 类进行权限检查

```
public void writeFile(Path path, byte[] content) throws IOException {
    SecurityManager securityManager = System.getSecurityManager();
    if (securityManager != null) {
        securityManager.checkWrite(path.toString());
    }
    Files.write(path, content);
}
```

权限检查也可以由 java.security.AccessController 类来完成。AccessController 类的 checkPermission 方法用来检查参数中 Permission 类的对象所表示的权限在当前的访问控制上下文和策略中是否合法。如果合法，那么 checkPermission 方法直接返回；否则抛出 AccessControlException 异常。代码清单 13-8 给出了使用 AccessController 类实现与代码清单 13-7 中相同的权限检查功能的示例。

代码清单 13-8 使用 AccessController 类进行权限检查

```
public void writeFile(Path path, byte[] content) throws IOException {
    FilePermission permission = new FilePermission(path.toString(), "write");
    AccessController.checkPermission(permission);

    Files.write(path, content);
}
```

AccessController 类是在 SecurityManager 类之后被引入到 Java 标准库中的，可以用来替代 SecurityManager 类的功能。不过在 Java 标准库早期版本中使用 Security-

Manager 类所做的权限检查，在引入 AccessController 类之后并没有被替换成使用 AccessController 类的实现，只是对 SecurityManager 类的实现进行了修改，改为通过调用 AccessController 类中的方法完成相关的权限检查。在程序中应该优先使用 AccessController 类进行权限检查。

13.3.3 特权动作

在一个方法调用的执行过程中，实际的执行流程通常会跨越多个保护域的边界。比如在程序中输出一行文本到控制台，在执行过程中最终需要调用系统域中所提供的访问系统资源的方法，相当于对该方法的调用同时跨越了应用域和系统域。在进行权限检查时，不仅需要查看对该方法的调用在当前调用上下文中是否合法，还需要沿着方法调用栈逐个向上检查，确保对其中的每个方法的调用都是合法的。这些方法的调用过程形成一个完整的链条。如果在调用链上的任何一个方法不具有所需的权限，那么整个调用过程是非法的，会抛出 AccessControlException 异常。唯一例外的情况是，如果方法的调用执行的是特权动作（privileged action），那么就不受这个限制。在进行权限检查时，Java 平台采用的是延迟判断的机制。在每次需要进行权限检查时，会根据当前调用上下文的即时状态来进行判断。

特权动作只关心是否具备动作本身所要求的权限，而并不关心调用者是谁。在沿着调用链向上进行权限检查的过程中，如果遇到了特权动作，则只检查该特权动作所要求的权限是否满足，而不再继续沿着调用链向上检查。对于一个写入文件的特权动作，它只要求对该文件具有写入权限即可，并不关心是谁要求它执行这样的动作。特权动作由 java.security.PrivilegedAction 接口和 java.security.PrivilegedExceptionAction 接口的实现类来表示。两个接口中都只有一个 run 方法用来执行具体的操作。不同之处在于 PrivilegedAction 接口的 run 方法不能抛出受检异常，而 PrivilegedExceptionAction 类的 run 方法是可以的。PrivilegedAction 接口和 PrivilegedExceptionAction 接口实现类的对象由 AccessController 类的静态方法 doPrivileged 负责执行。

如果程序中需要读取系统属性，可以使用 System 类的 getProperty 方法。在 getProperty 方法的调用中会检查调用者是否具有相应的“java.util.PropertyPermission”权限。在启用了安全管理器之后，代码清单 13-9 中的 get 方法在调用时会抛出 AccessControlException 异常。

代码清单 13-9 获取系统属性的方法示例

```
public static String get(String property) {
    return System.getProperty(property);
}
```

一种简单的做法是修改程序所用的策略，为代码清单 13-9 中的方法调用增加所需的权限。但是程序中类似的调用可能很多，不可能为所有这些方法的调用都逐一

添加所需的权限。特权动作可以解决这个问题。在代码清单 13-10 中，把对 System 类的 getProperty 方法的调用封装在一个 PrivilegedAction 接口的实现中，并使用 AccessController 类的 doPrivileged 方法来执行。这样做好处是，只要为代码清单 13-10 中的 getWithPrivilege 方法添加所需的“java.util.PropertyPermission”权限声明，程序中的其他部分即可直接调用该方法，不需要分配额外的权限。

代码清单 13-10 使用特权动作获取系统属性

```
public static String getWithPrivilege(final String property) {
    return AccessController.doPrivileged(new PrivilegedAction<String>() {
        public String run() {
            return System.getProperty(property);
        }
    });
}
```

如果特权动作的执行抛出受检异常，那么需要使用 PrivilegedExceptionAction 接口的实现类。代码清单 13-11 给出了一个使用特权动作写入文件的示例。在进行文件写入操作的过程中可能会抛出受检异常 IOException，因此在 PrivilegedExceptionAction 接口的实现类的 run 方法中声明了抛出 IOException 异常。AccessController 类的 doPrivileged 方法在执行 PrivilegedExceptionAction 接口的实现对象时会抛出受检异常 java.security.PrivilegedActionException，因此在代码中需要捕获该异常，并通过 getCause 方法得到在特权动作执行中产生的实际的 IOException 异常，并把该异常重新抛出。通过这样的方式，writeFileWithPrivilege 方法的声明与代码清单 13-8 中的 writeFile 方法保持一致，对方法的使用者屏蔽了使用特权动作的实现细节。

代码清单 13-11 使用特权动作写入文件的示例

```
public void writeFileWithPrivilege(final Path path, final byte[] content) throws
IOException {
    try {
        AccessController.doPrivileged(new PrivilegedExceptionAction<String>() {
            public String run() throws IOException {
                Files.write(path, content);
                return path.toString();
            }
        });
    } catch (PrivilegedActionException e) {
        throw (IOException) e.getCause();
    }
}
```

13.3.4 访问控制上下文

在一个线程的执行过程中，该线程维护了与它所具有的访问控制权限相关的上下文

信息。当前线程在新线程创建时的访问控制上下文信息会被保存下来，并作为被继承的上下文信息与新创建的线程关联起来。在调用方法时的访问控制权限检查会先按照正常的方式检查当前线程中方法的调用栈，再使用继承的上下文信息来进行检查。

访问控制上下文由 `java.security.AccessControlContext` 类表示。通过 `AccessController` 类的 `getContext` 方法可以得到当前线程所使用的访问控制上下文信息的一个快照，将其封装在 `AccessControlContext` 类的对象中。如果希望在其他线程中使用当前线程的访问控制上下文来进行权限检查，那么可以把该 `AccessControlContext` 类的对象传递给另外一个线程，并通过 `checkPermission` 方法来进行检查。`AccessController` 类的 `doPrivileged` 方法在执行特权动作时，也可以指定一个 `AccessControlContext` 类的对象。在特权动作的执行过程中，不仅会考虑当前线程的访问控制上下文，还会考虑 `AccessControlContext` 类中封装的上下文信息。实际上，在 `AccessController` 类的 `checkPermission` 方法的实现中，也是先得到当前线程对应的 `AccessControlContext` 类的对象，再使用该对象的 `checkPermission` 方法来进行权限检查。

每个 `AccessControlContext` 类的对象是与一组 `ProtectionDomain` 类的对象关联在一起的。在 `AccessControlContext` 类的 `checkPermission` 方法的实现中，作为参数传递的 `Permission` 类的对象会被传递给所有关联的 `ProtectionDomain` 类的对象的 `implies` 方法。只有当所有的 `ProtectionDomain` 类的对象调用 `implies` 方法的返回值都为 `true` 时，需要检查的权限才是具备的。如果 `AccessControlContext` 类的对象所关联的 `ProtectionDomain` 类的对象需要被动态修改，那么可以在创建时传入一个 `java.security.DomainCombiner` 接口的实现对象。在获取 `AccessControlContext` 类的对象时，`DomainCombiner` 接口的 `combine` 方法会被调用，用来对所关联的 `ProtectionDomain` 类的对象进行修改。

在有些情况下，需要以特定的主体身份标识来执行某些操作。如果在策略声明中为某个主体分配了相应的权限，那么只有以该主体作为身份标识时才能运行。使用 `Subject` 类的静态方法 `doAs` 和 `doAsPrivileged` 可以将调用参数中给出的 `Subject` 类的对象作为主体来执行特权动作。两个方法的区别在于 `doAsPrivileged` 方法不像 `doAs` 方法一样使用的是当前线程的访问控制上下文信息，而是使用作为参数传递的 `AccessControlContext` 类的对象。代码清单 13-12 给出了 `Subject` 类的 `doAsPrivileged` 方法的使用示例。在实际的程序中，`Subject` 类中的身份标识信息应该通过登录过程来添加。

代码清单 13-12 `Subject` 类的 `doAsPrivileged` 方法的使用示例

```
public void doAs() {
    Subject subject = new Subject();
    UserPrincipal principal = new UserPrincipal("Alex");
    subject.getPrincipals().add(principal);
    subject.setReadOnly();
    String userHome = Subject.doAsPrivileged(subject, new PrivilegedAction<String>() {
```

```

        public String run() {
            return System.getProperty("user.home");
        }
    }, null);
}

```

与 doAsPrivileged 方法调用对应的策略文件如代码清单 13-13 所示。

代码清单 13-13 与 doAsPrivileged 方法调用对应的策略文件

```

grant principal com.java7book.chapter13.auth.UserPrincipal "Alex" {
    permission java.util.PropertyPermission "user.*", "read";
};

grant {
    permission javax.security.auth.AuthPermission "*";
};

```

在 doAsPrivileged 方法的实现中使用了 DomainCombiner 接口的实现类 javax.security.auth.SubjectDomainCombiner，在获取 AccessControlContext 类的对象时，SubjectDomainCombiner 类的对象会找到当前 Subject 类的对象中的身份标识所关联的 ProtectionDomain 类的对象，并把这些 ProtectionDomain 类的对象合并到 AccessControlContext 类的对象中。

13.3.5 守卫对象

通过传递 AccessControlContext 类的对象可以在其他线程中通过当前线程的访问控制上下文信息进行权限检查。在有些情况下，可能传递 AccessControlContext 类的对象不是一个可以接受的选择，这时可以使用 java.security.GuardedObject 类和 java.security.Guard 接口。在创建 GuardedObject 类的对象时需要提供另外一个被保护的对象，以及保护该对象的 Guard 接口的实现对象。GuardedObject 类的对象的使用者可以通过 getObject 方法来获取这个被保护的对象，前提是对应的 Guard 接口的实现对象的 checkGuard 方法可以成功完成。Permission 类实现了 Guard 接口，可以很方便地作为判断条件来使用。代码清单 13-14 给出了 GuardedObject 类和 Guard 接口的使用示例——把一个 FilePermission 类的对象作为 GuardedObject 类的对象使用的判断条件。如果 getObject 方法的调用者具有相应的权限，则 getObject 方法可以正确返回被保护的对象。

代码清单 13-14 GuardedObject 类和 Guard 接口的使用示例

```

public GuardedObject readFile(Path path) throws IOException {
    FilePermission permission = new FilePermission(path.toString(), "read");
    byte[] data = Files.readAllBytes(path);
    GuardedObject guardedObj = new GuardedObject(data, permission);
    return guardedObj;
}

```

```
public void useFile(Path path) throws IOException {
    GuardedObject guardedObj = readFile(path);
    byte[] data = (byte[]) guardedObj.getObject();
}
```

13.4 加密与解密、报文摘要和数字签名

构建安全的应用离不开加密和解密操作。当一个程序中的数据被保存到存储介质上或通过网络方式传输时，为了保证数据的安全性，需要对数据进行一定的处理。如果不希望数据被恶意的第三方所窃取，那么可以对数据进行加密。经过加密处理之后，即便数据被窃取，窃取者也无法获取数据的真实内容。如果希望确保数据没有被第三方篡改，那么可以从数据中创建报文摘要或添加消息验证码。Java 密码框架中提供了加密、解密、报文摘要和数字签名等相关的 API。

13.4.1 Java 密码框架

Java 密码框架采用了常见的服务提供者架构，以提供所需的可扩展性和互操作性。Java 密码框架的设计原则是实现的独立性和互操作性，以及算法的独立性和可扩展性。密码框架可以看成一些服务的集合，加密、解密、报文摘要和数字签名等都是它所提供的服务。每个服务可能有不同的实现算法，比如报文摘要可以使用 MD5 算法或 SHA-1 算法。程序本身作为服务的消费者，并不需要了解服务本身的实现细节，只需要根据程序的需求，选择合适的服务和算法即可。服务的具体实现由 Java 平台或第三方提供。Java 密码框架所使用的服务提供者架构使添加新的服务和算法实现变得很容易。Java 标准库为不同的服务提供了各自抽象的接口。服务的不同实现之间可以互操作。

Java 平台或第三方都提供了很多密码框架中的服务的实现。一个服务提供者通常会实现多个服务，并为每个服务提供多种不同的算法实现。每个服务提供者都继承自 `java.security.Provider` 类。Provider 类的对象有自己的名称及它所实现的服务和算法的列表。程序如果要使用第三方的服务提供者所提供的实现，需要先把相应的 jar 包放在程序的类路径或 JDK 的标准扩展目录下，再通过修改配置文件或调用 `java.security.Security` 类的 `addProvider` 方法来进行添加。通过 `Security` 类的 `getProviders` 方法可以得到包含当前所有可用的 Provider 类的对象的数组。

代码清单 13-15 给出了 Provider 类的使用示例。通过 `getServices` 方法可以得到 Provider 类的对象表示的服务提供者所能提供的服务的列表。在表示服务的 `Provider.Service` 类的对象中，可以使用 `getAlgorithm`、`getClassName` 和 `getType` 方法分别获取服务所实现的算法、对应的 Java 类名和服务的类别。

代码清单 13-15 Provider 类的使用示例

```

Provider[] providers = Security.getProviders();
for (Provider provider : providers) {
    Set<Provider.Service> services = provider.getServices();
    for (Provider.Service service : services) {
        System.out.println(service.getAlgorithm() + " <==> " + service.
            getClassName());
    }
}

```

当系统中存在一个服务的多个实现时，如果服务的消费者没有显式指定该服务的提供者的名称，则 Java 平台会根据服务提供者的优先级来选择最合适的实现。服务的消费者在使用一个服务时只需要确定服务的类别、所用的算法及可选的提供者的名称。对于不同类别的服务，Java 标准库提供了对应的 Java 类作为该服务的抽象表示。以报文摘要服务为例来说，java.security.MessageDigest 类是该服务的抽象表示。这些抽象表示类一般都提供一个 getInstance 方法根据算法名称和提供者的名称来创建出具体的实现对象。对于每个服务，Java 标准库还提供了对应的服务提供者接口，是一些抽象的 Java 类（SPI 类）。这些 SPI 类的命名规则是在服务的抽象表示类名之后加上“Spi”，如 MessageDigest 类对应的 SPI 类是 java.security.MessageDigestSpi。服务的具体实现者需要继承自对应的 SPI 类，并实现自己的逻辑。这些 SPI 类的存在，使得在不同服务提供者的实现之间可以进行互操作。

13.4.2 加密与解密

加密的目的是保证数据被窃取之后，数据中所包含的信息仍然不会泄露。加密的过程是把原始数据（明文）转换成第三方无法识别的内容（密文），解密的过程则是把密文重新转换成明文。在这个过程中必不可少的关键部分是密钥。密钥是加密和解密算法实现的基础。加密算法一般分为对称和非对称两种。对称加密算法使用同一个密钥进行加密和解密，而非对称加密算法使用一对密钥分别进行加密和解密，两个密钥分别称为公钥和私钥。使用其中一个密钥进行加密的数据，需要使用另外一个密钥来解密。非对称算法的公钥是公开的，而私钥则由程序妥善保存。需要与程序进行通信的其他代码使用公钥对数据进行加密，程序使用保存的私钥对数据进行解密。

1. 密钥

密钥在 Java 密码框架中有两种表示方式，一种是基于 java.security.Key 接口的不透明表示方式，另一种是基于 java.security.spec.KeySpec 接口的透明表示方式。在程序中使用密钥时，可以直接从相关工厂方法中得到所需的 Key 接口的实现对象。Key 接口所提供的操作比较有限，不能通过该接口获取密钥具体实现的内部细节。通过 Key 接口的 getAlgorithm 方法可以得到该密钥所使用的算法。当密钥在不同组件之间传输时，

需要使用一些标准的格式进行编码。使用 `getFormat` 方法可以得到编码格式的名称，而使用 `getEncoded` 方法可以获得编码之后的字节数组。对于对称算法来说，`javax.crypto.SecretKey` 接口表示唯一的私钥；对于非对称算法来说，`java.security.PublicKey` 接口和 `java.security.PrivateKey` 接口分别表示公钥和私钥。`KeySpec` 只是一个标记接口，并没有具体的方法。从 `KeySpec` 接口的具体实现类中可以获取密钥底层实现的具体细节。通过 `java.security.KeyFactory` 和 `javax.crypto.SecretKeyFactory` 类中的方法可以在 `Key` 接口和 `KeySpec` 接口的对象之间互相转换，前者和后者分别适合非对称算法和对称算法使用的密钥。

密钥的获取通过标准的服务提供者来完成，对称算法使用 `javax.crypto.KeyGenerator` 类，而非对称算法使用 `java.security.KeyPairGenerator` 类。通过静态方法 `getInstance` 获得服务提供者的具体实现对象。在使用这两个类的对象之前，需要调用 `init` 方法进行初始化。如果没有调用 `init` 方法进行初始化，那么两个类使用默认设置来生成密钥。

2. 加密与解密的过程

加密和解密功能由服务 `javax.crypto.Cipher` 类提供。创建 `Cipher` 类的实现对象也是通过 `getInstance` 方法来完成。在调用 `getInstance` 方法时需要指定加密时的转换方式。转换方式包括必要的算法名称及可选的反馈模式和填充方式。接着调用 `init` 方法通过合适的密钥把 `Cipher` 类的对象初始化成所需的加密或解密模式。如果需要加密或解密的数据过多，那么可以分成多次来进行处理。在每次处理中调用 `update` 方法来提供数据。在最后一次处理中调用 `doFinal` 方法来完成。如果数据较少，可以直接使用 `doFinal` 方法来提供数据并同时完成处理。调用 `doFinal` 方法的返回值是加密或解密之后的结果数据。

代码清单 13-16 给出了对称加密和解密操作的一个示例。在加密方法 `encrypt` 中，先使用 `KeyGenerator` 类的对象来生成所需的密钥，并把密钥编码之后的格式保存下来。在得到 `Cipher` 类的对象并完成初始化之后，对一段文本进行加密处理，把加密之后的结果数据保存下来。使用对称加密方式的双方需要通过可靠的方式来传递密钥。在示例中是通过文件的方式来传递的。在解密方法 `decrypt` 中，先读取保存的密钥数据，从中创建出 `SecretKeySpec` 类的对象，再使用 `SecretKeySpec` 类的对象把 `Cipher` 类的对象初始化为解密模式，最后对加密的数据进行处理即可得到原始的明文数据。

代码清单 13-16 对称加密和解密操作的示例

```
public class SymmetricEncryption {
    public void encrypt() throws Exception {
        KeyGenerator generator = KeyGenerator.getInstance("DES");
        SecretKey key = generator.generateKey();
        Files.write(Paths.get("key.data"), key.getEncoded());
        Cipher cipher = Cipher.getInstance("DES");
        cipher.init(Cipher.ENCRYPT_MODE, key);
        String text = "Hello World";
        byte[] encrypted = cipher.doFinal(text.getBytes());
```

```

        Files.write(Paths.get("encrypted.bin"), encrypted);
    }

    public void decrypt() throws Exception {
        byte[] keyData = Files.readAllBytes(Paths.get("key.data"));
        SecretKeySpec keySpec = new SecretKeySpec(keyData, "DES");
        Cipher cipher = Cipher.getInstance("DES");
        cipher.init(Cipher.DECRYPT_MODE, keySpec);
        byte[] data = Files.readAllBytes(Paths.get("encrypted.bin"));
        byte[] result = cipher.doFinal(data);
        System.out.println(new String(result));
    }
}

```

非对称加密的实现方式类似于对称加密的方式，只不过在加密和解密时使用的密钥不同。

要对一个数据流进行加密和解密操作，可以使用 javax.crypto.CipherInputStream 类和 javax.crypto.CipherOutputStream 类这两个过滤流实现。在创建这两个流的对象时都需要使用一个 Cipher 类的对象。在 CipherOutputStream 类的对象写入数据时，会先通过 Cipher 类的对象对数据进行加密处理，再把加密的结果写入所包装的底层输出流中；CipherInputStream 类的对象进行数据读取时，会先对从底层输入流中读取的数据进行解密处理，再把解密的结果返回给调用者。代码清单 13-17 给出了使用加密方式把对象序列化之后的内容安全保存在磁盘上的示例。

代码清单 13-17 使用加密方式保存对象序列化之后的内容

```

public void storeSafely(Serializable obj, Cipher cipher, Path path) throws
IOException {
    try (ObjectOutputStream oos = new ObjectOutputStream(new CipherOutputStream
        (Files.newOutputStream(path), cipher))) {
        oos.writeObject(obj);
    }
}

```

Cipher 类、CipherInputStream 类和 CipherOutputStream 类所处理的对象都是字节流。如果需要加密 Java 对象，可以使用 javax.crypto.SealedObject 类。SealedObject 类可以用来封装一个实现了 Serializable 接口的对象。在创建 SealedObject 类的对象时需要提供进行加密操作的 Cipher 类的对象。使用者通过 getObject 方法来获取所封装的对象，不过在调用 getObject 方法时需要提供进行解密操作的 Cipher 类的对象或密钥。

13.4.3 报文摘要

使用报文摘要的目的是防止数据的内容被恶意篡改。当数据通过网络或其他方式传输时，有可能有第三方对数据进行拦截并篡改其中的内容。数据的接收者会在不知情的

情况下使用错误的数据。报文摘要的做法是使用某种算法对原始数据进行处理，得到一个固定长度的摘要，这个摘要会随着数据一同公开出来。接收者对接收到的数据使用相同的算法计算出摘要，再与正确的摘要进行比较。如果不一致，则说明数据已经被篡改。摘要算法可以对任意长度的数据进行处理，得到的都是固定长度的摘要。报文摘要算法具有一个显著的特点，那就是要找到两段内容不同而摘要相同的数据，在计算上几乎是不可能的。这一特点保证如果数据被篡改，总是可以通过摘要的变化来检查出来。另外，摘要本身也不会暴露原始数据的任何信息。不过随着技术的进步，一些之前安全的摘要算法，也被发现有安全漏洞。

进行报文摘要的 `MessageDigest` 类的使用方式类似于之前提到的 `Cipher` 类，使用 `getInstance` 方法获取不同算法的实现。如果数据较多，那么使用 `update` 方法来分成多次进行处理。最后一次处理或数据较少时使用 `digest` 方法。代码清单 13-18 给出了使用常见的 MD5 算法计算摘要的示例。

代码清单 13-18 使用 MD5 算法计算摘要的示例

```
MessageDigest md = MessageDigest.getInstance("MD5");
byte[] digest = md.digest("Hello World".getBytes());
```

与报文摘要类似的机制是消息验证码（Message Authentication Code，MAC）。与报文摘要的不同在于消息验证码在计算过程中使用了密钥，只有掌握了密钥的接收者才能验证数据的完整性。消息验证码可以解决摘要本身也可能被篡改的问题。在实现中使用了散列函数的消息验证码被称为 HMAC。一般的做法是对数据本身使用 HMAC 计算出消息验证码，再把消息验证码附加在原始数据之后，最后使用密钥对整个数据进行加密之后进行传输。消息验证码使用 `javax.crypto.Mac` 类进行计算。代码清单 13-19 中给出了计算消息验证码的示例。

代码清单 13-19 计算消息验证码的示例

```
KeyGenerator keyGenerator = KeyGenerator.getInstance("HmacMD5");
SecretKey key = keyGenerator.generateKey();
Mac mac = Mac.getInstance("HmacMD5");
mac.init(key);
byte[] result = mac.doFinal("Hello World".getBytes());
```

13.4.4 数字签名

数字签名可以用来实现身份验证功能。在进行数字签名时需要使用一对公钥和私钥。例如，对于进行通信的两个对等体 A 和 B，如果 A 需要验证 B 的身份，那么 B 需要使用私钥对消息进行加密，并把加密结果发送给 A，A 使用公钥进行解密。由于私钥只有 B 知道，当 A 使用公钥成功对数据进行解密之后，可以判定消息的来源肯定是该公钥对应的持有者 B，这就相当于 B 对消息进行了签名。

数字签名的服务由 `java.security.Signature` 类来提供。`Signature` 类的对象有签名和验证两种不同的工作模式。代码清单 13-20 给出了 `Signature` 类的使用示例。在 `init` 方法中通过 `getInstance` 方法得到对应算法的 `Signature` 类的对象，并通过 `KeyPairGenerator` 类的对象生成分别用于进行签名和验证操作的公钥和私钥。在进行签名时，使用私钥调用 `initSign` 方法进行初始化。初始化完成之后，使用 `update` 方法提供原始数据。所有数据提供完毕之后，调用 `sign` 方法得到签名。在进行验证时，使用公钥调用 `initVerify` 方法进行初始化，同时使用 `update` 方法提供原始数据。最后调用 `verify` 方法对签名进行验证。

代码清单 13-20 数字签名的使用示例

```
public class DigitalSignature {  
    private Signature signature;  
    private PublicKey publicKey;  
    private PrivateKey privateKey;  
    private byte[] data = "Hello World".getBytes();  
  
    public DigitalSignature() {  
        init();  
    }  
  
    private void init() {  
        try {  
            signature = Signature.getInstance("SHA1withDSA");  
            KeyPairGenerator keyGenerator = KeyPairGenerator.getInstance("DSA");  
            KeyPair keyPair = keyGenerator.generateKeyPair();  
            publicKey = keyPair.getPublic();  
            privateKey = keyPair.getPrivate();  
        }  
        catch (GeneralSecurityException e) {  
            e.printStackTrace();  
        }  
    }  
  
    public byte[] sign() throws GeneralSecurityException {  
        signature.initSign(privateKey);  
        signature.update(data);  
        return signature.sign();  
    }  
  
    public boolean verify(byte[] signatureData) throws GeneralSecurityException {  
        signature.initVerify(publicKey);  
        signature.update(data);  
        return signature.verify(signatureData);  
    }  
  
    public void testSignature() throws GeneralSecurityException {  
        boolean result = verify(sign());  
    }  
}
```

```

        System.out.println(result); // 输出为 true
    }
}

```

Signature 类所操作的对象和输出的结果都是字节数组。这种方式使用起来比较麻烦，尤其在处理 Java 对象时。当需要对某个 Java 对象进行签名时，可以使用 java.security.SignedObject 类。SignedObject 类的对象可以用来封装任何实现了 Serializable 接口的类的对象。在创建 SignedObject 类的对象时，需要提供进行数字签名的 PrivateKey 接口的实现对象和 Signature 类的对象。SignedObject 类的对象是不可变的，同时它所封装的是基于序列化机制实现的对象的深拷贝。SignedObject 类的对象一旦创建之后，对原始对象的修改不会影响到它。

SignedObject 类的对象适合于在不同组件之间进行传递，而不用担心对象的内容会被其他组件修改。接收到 SignedObject 类的对象的组件可以使用公钥来调用 verify 方法对该对象进行验证。SignedObject 类本身也实现了 Serializable 接口，因此当需要把对象的序列化形式安全地保存下来时，可以用一个 SignedObject 类的对象先进行封装，再直接序列化该 SignedObject 类的对象。代码清单 13-21 给出了使用 SignedObject 类来安全保存对象的示例。

代码清单 13-21 使用 SignedObject 类来安全保存对象的示例

```

public void saveObject(Serializable obj, Path path) throws GeneralSecurity-
Exception, IOException {
    Signature signature = Signature.getInstance("SHA1withDSA");
    KeyPairGenerator keyGenerator = KeyPairGenerator.getInstance("DSA");
    KeyPair keyPair = keyGenerator.generateKeyPair();
    SignedObject signedObj = new SignedObject(obj, keyPair.getPrivate(),
        signature);
    try (ObjectOutputStream oos = new ObjectOutputStream(Files.newOutputStream(
        path))) {
        oos.writeObject(signedObj);
    }
}

```

SignedObject 类的使用类似于之前介绍的 SealedObject 类，不同在于 SignedObject 类用来防止 Java 对象被篡改，而 SealedObject 类用来防止对象的信息泄露。

13.5 安全套接字连接

在各种数据传输方式中，网络传输目前使用较广，存在的安全隐患也较多。数据在网络传输过程中可能被第三方窃取或篡改。为了保护数据的安全，在涉及网络传输的程序中，可以使用安全套接字层（Secure Sockets Layer，SSL）协议。SSL 协议在 TCP/IP 协议栈中位于传输层协议（TCP）和应用层协议（包括 HTTP、Telnet 和 FTP 等）之间。SSL 比较常见的使用方式是与 HTTP 一起使用，用来构建安全的 Web 应用。SSL 协议最

早由 Netscape 公司开发，现在是由 IETF 组织维护的国际标准。标准化之后的 SSL 协议改名为传输层安全协议（Transport Layer Security，TLS）。Java SE 7 中默认的实现支持 TLS 1.1 和 TLS 1.2 协议。

13.5.1 SSL 协议

SSL 协议的目的在于解决网络传输中存在的 3 个安全问题。第一个问题是身份认证，即确保当前正在通信的对等体的身份是合法的。这点在 Web 应用中尤其重要。如果不解决身份认证的问题，那么用户会将攻击者创建的钓鱼网站误认为是正确的网站。攻击者通过这种方式可以盗取用户的信息。SSL 协议允许通信的双方在建立数据连接之前，先进行身份验证。在身份验证中使用经过数字签名的证书。第二个问题是数据被窃取。SSL 协议在传输数据的过程中会对数据进行加密，这样可以保证即便在数据泄露的情况下，其中所包含的信息不会被窃取。第三个问题是数据可能被篡改。SSL 协议对传输的数据添加了消息验证码，接收者可以对数据的完整性进行校验。

在安全套接字连接建立之前，客户端和服务器端之间需要通过握手机制就连接的细节达成一致。当握手结束之后，双方可以按照约定的方式自由发送数据。SSL 协议的握手过程比较复杂，大致分成如下几步。

1) 客户端发出连接请求。请求中包含客户端所能支持的 SSL 协议的最高版本，以及所能使用的加密算法的信息。

2) 服务器端接收到连接请求之后，根据客户端给出的信息，选择双方都能支持的最高版本的 SSL 协议，并确定双方都能使用的加密算法。服务器端把选择的结果发送给客户端。如果客户端要求认证服务器端的身份，那么服务器端把它所持有的数字证书发送给客户端。如果服务器端也需要认证客户端的身份，那么服务器端向客户端发出认证请求。如果服务器端发送的数字证书中包含的信息不足以在双方之间进行密钥交换，那么服务器端会发送额外的密钥信息。

3) 接着由客户端来处理服务器端返回的响应内容。如果客户端收到了服务器端进行身份验证的请求，那么客户端使用自己的私钥对所持有的数字证书进行加密之后发送给服务器端。客户端生成在数据传输时进行加密操作所使用的密钥，并使用服务器端给出的公钥进行加密之后发送给服务器端。所有这些操作完成之后，客户端发出通知，切换到加密的数据传输方式。

4) 最后由服务器端来完成整个握手过程。如果服务器端在第 2) 步中选择验证客户端身份，会先验证客户端发送过来的信息是否正确。接着同样切换到加密的数据传输方式。握手过程结束后双方进行数据传输，直到连接关闭。

上述握手过程中的一个重要组成部分是数字证书。数字证书中包含证书持有者的身份信息和公钥。在非对称加密算法中，公钥是公开的。使用公钥可以解决如何安全共享密钥的问题。但是还有一个问题是如何确保接收者所得到的公钥来自所声明的真实发送者，而不是伪造的。攻击者可以随意创建一个新的公钥私钥对，并声称该公钥属于

另外一个公司或组织。数字证书的目的就是解决这个问题。证书由用户所信任的机构（Certification Authority）签发，并通过该机构的私钥来加密。数字证书持有者的真实性由信任机构来保证。在有些情况下，某个证书签发机构的真实性需要由另外一个机构的证书来证明。通过这种证明关系形成一个证书的链条，而链条的根是公认的值得信任的机构。只有当证书链条上的所有证书都被信任时，证书中所给出的公钥才能得到信任。支持 SSL 协议的应用，如浏览器，通常会把一些重要的信任机构的公钥保存起来，在验证时使用。对于一个证书，先使用这些保存的公钥来检验证书本身的合法性。证书检验合法之后，证明其中所包含的身份信息和公钥是真实的，可以使用这个证书中包含的公钥进行身份认证。

在 SSL 握手过程中的一个重要步骤是双方对数据传输时使用的密钥达成一致。数据传输过程中使用的是对称加密算法。服务器端和客户端持有相同的密钥，客户端把生成的密钥经过服务器端的公钥加密之后，发送给服务器端，服务器端用自己的私钥解密，就得到了双方共同使用的密钥。

对于开发人员来说，安全套接字的使用与普通的套接字连接并没有太大的区别。建立连接和数据传输时使用的不是 `java.net.Socket` 类和 `java.net.ServerSocket` 类，而是它们的子类 `javax.net.ssl.SSLSocket` 类和 `javax.net.ssl.SSLServerSocket` 类。相应的工厂类 `javax.net.ssl.SSLSocketFactory` 和类 `javax.net.ssl.SSLServerSocketFactory` 分别用来创建 `SSLSocket` 类和 `SSLServerSocket` 类的对象。在使用 `SSLSocket` 类和 `SSLServerSocket` 类的对象建立连接的过程中，相关的 SSL 握手机制是自动完成的。在使用过程中，先要创建 `javax.net.ssl.SSLContext` 类的对象，通过 `SSLContext` 类的对象的 `getSocketFactory` 和 `getServerSocketFactory` 方法可以得到用来创建套接字对象的 `SSLSocketFactory` 类和 `SSLServerSocketFactory` 类的对象。

`SSLContext` 类也采用了标准的服务提供者机制，使用 `getInstance` 方法并指定 SSL 协议的版本可以创建出所需的对象。创建了 `SSLContext` 类的对象之后，一般需要调用 `init` 方法进行初始化。在初始化时需要提供 3 个参数：第一个参数是 `javax.net.ssl.KeyManager` 接口的实现对象，用于对 SSL 握手中所需的密钥进行管理，提供当前对等体所持有的公钥；第二个参数是 `javax.net.ssl.TrustManager` 接口的实现对象，用于根据当前对等体的证书判断是否信任与之通信的另一个对等体；第三个参数是 `java.security.SecureRandom` 类的对象，用于生成加密时所需的安全的随机数。

13.5.2 HTTPS

HTTPS 是一种把 HTTP 和 SSL/TLS 协议结合起来的通信方式。一些对安全性和隐私保护要求较高的网站要求通过 HTTPS 的方式进行数据传输。典型的网站包括电子商务、网上银行和电子邮件服务等网站。有些网站的所有通信都是使用 HTTPS 进行的，而有些网站只在访问重要页面时使用 HTTPS，如用户登录或进行网上支付时。在使用 HTTPS 的情况下，通过 HTTP 传输的数据都会被加密，保证了数据的安全。

配置 Web 服务器支持 HTTPS 方式并不是一件复杂的事情。很多 Web 服务器都提供了所需的功能，只要进行简单的配置即可。在通过 HTTPS 方式访问网站时，身份验证变得非常重要。进行身份验证的前提是服务器端可以提供合法有效的证书。网站的提供者通常需要向证书签发机构交纳一定的费用，才能得到有效的证书。在开发中经常会遇到自签发的证书，这些证书是由网站提供者自行创建的。在完整的证书链条中，并没有信任机构签发的证书。当通过 HTTPS 方式访问使用自签发证书的网站时，浏览器会给出相关的警告信息，要求用户对访问请求进行确认。如果是以程序的方式进行访问的，则访问请求会失败。

代码清单 13-22 中的 Java 类在读取 HTTPS 连接内容时会忽略所有与证书和主机名称验证相关的错误，总是能够读出内容。不过这种方式会带来很大的安全风险，应该谨慎使用。

代码清单 13-22 忽略证书和主机名称验证的 HTTPS 连接

```
public class ReadAllHttpsClient {  
    public byte[] read(String urlString) throws IOException, GeneralSecurityException {  
        URL url = new URL(urlString);  
        SSLContext context = SSLContext.getInstance("TLS");  
        context.init(new KeyManager[] {}, new TrustManager[] { new MyTrustManager() },  
                    new SecureRandom());  
        HttpsURLConnection connection = (HttpsURLConnection) url.openConnection();  
        connection.setSSLSocketFactory(context.getSocketFactory());  
        connection.setHostnameVerifier(new MyHostnameVerifier());  
        return IOUtils.toByteArray(connection.getInputStream());  
    }  
  
    private static class MyTrustManager implements X509TrustManager {  
        public void checkClientTrusted(X509Certificate[] chain, String authType)  
            throws CertificateException {  
        }  
  
        public void checkServerTrusted(X509Certificate[] chain, String authType)  
            throws CertificateException {  
        }  
  
        public X509Certificate[] getAcceptedIssuers() {  
            return null;  
        }  
    }  
  
    private static class MyHostnameVerifier implements HostnameVerifier {  
        public boolean verify(String hostname, SSLSession session) {  
            return true;  
        }  
    }  
}
```

13.6 使用案例

在 Web 应用中需要考虑的安全性相关的问题比较多，因为 Web 应用本身面对的使用环境中存在较多的不安全因素，开发人员对 Web 应用的安全性也比较重视。Web 应用的安全性一般包括身份认证和权限控制两个方面。Web 应用一般允许用户注册自己的用户名，但是某些特定的页面只有认证用户才可以访问。不同的用户一般会被分配不同的角色。每个角色所对应的权限不同。下面以一个 Web 应用为例来介绍身份认证和权限控制的使用方式。该 Web 应用是一个简单的员工信息管理系统，在访问时要先进行登录，只有具有“manager”角色的用户才可以查看员工信息。

首先是登录过程。这个案例使用的登录模块是代码清单 13-3 中基于属性文件的 PropertiesFileBasedLoginModule 类的扩展。除了用户名和密码之外，在属性文件中增加了用户的角色信息。由于用户通过 HTTP 请求来提交用户名和密码等信息，因此需要对登录模块中使用的与用户交互的 CallbackHandler 接口的实现进行相应的修改。代码清单 13-23 中的 RequestCallbackHandler 类在处理 Callback 接口实现对象时从 HttpServletRequest 对象中获取参数的值。

代码清单 13-23 从 HttpServletRequest 对象中获取参数值的 CallbackHandler 接口的实现

```
public class RequestCallbackHandler implements CallbackHandler {
    private final HttpServletRequest request;

    public RequestCallbackHandler(HttpServletRequest request) {
        this.request = request;
    }

    public void handle(Callback[] callbacks) throws IOException,
        UnsupportedCallbackException {
        for (Callback callback : callbacks) {
            if (callback instanceof RequestParameterCallback) {
                RequestParameterCallback requestCallback =
                    (RequestParameter-
                        Callback) callback;
                String parameter = requestCallback.getParameter();
                String value = request.getParameter(parameter);
                requestCallback.setValue(value);
            } else {
                throw new UnsupportedCallbackException(callback);
            }
        }
    }
}
```

案例在登录过程中使用了自定义的 Configuration 类的子类来配置所使用的登录模块。代码清单 13-24 给出了所使用的 SecureConfiguration 类的代码。SecureConfiguration 类中只包含一个 AppConfigurationEntry 类的对象，表示的是案例中使用的唯一的登录模块。

代码清单 13-24 登录配置的实现类

```

public class SecureConfiguration extends Configuration {
    private Path rootPath;
    public SecureConfiguration(Path rootPath) {
        this.rootPath = rootPath;
    }

    public AppConfigurationEntry[] getAppConfigurationEntry(String name) {
        Map<String, String> properties = new HashMap<String, String>();
        Path path = rootPath.resolve("WEB-INF/classes/user.properties");
        properties.put("properties.file.path", path.toAbsolutePath().toString());
        AppConfigurationEntry entry = new AppConfigurationEntry("com.
            java7book.securewebapp.login.PropertiesFileBasedLoginModule",
            LoginModuleControlFlag.REQUIRED, properties);
        return new AppConfigurationEntry[] { entry };
    }
}

```

在案例中创建了一种新的主体身份标识 RolePrincipal 类，用来包含每个用户的角色信息。在登录认证成功之后，RolePrincipal 类的对象会被关联到表示当前主体的 Subject 类的对象中。当登录成功之后，从 LoginContext 类的对象中得到的 Subject 类的对象会被保存在会话中。用户登录的处理由一个 servlet 来完成，代码清单 13-25 中给出了该 servlet 中的 doPost 方法的实现代码。

代码清单 13-25 处理用户登录的 servlet 中的 doPost 方法的实现代码

```

protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    Path rootPath = Paths.get(request.getServletContext().getRealPath("/"));
    SecureConfiguration config = new SecureConfiguration(rootPath);
    RequestCallbackHandler callbackHandler = new RequestCallbackHandler(request);
    try {
        LoginContext loginContext = new LoginContext("SecureWebApp", null,
            callbackHandler, config);
        loginContext.login();
        Subject subject = loginContext.getSubject();
        request.getSession(true).setAttribute("subject", subject);
    } catch (LoginException e) {
        throw new ServletException(e);
    }
}

```

在权限控制方面，需要先声明操作执行时的权限。案例中使用的是基于 BasicPermission 类的包含名称和动作列表的权限。不过由于 BasicPermission 类被声明为 abstract，因此创建了一个新的子类 BasePermission 在程序中使用。声明一个权限最自然的方式是使用注解，代码清单 13-26 给出了案例中使用的声明权限的注解。

代码清单 13-26 声明权限的注解

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface DeclaredPermission {
    String name();
    String actions();
}
```

该注解可以添加到方法声明上，表明该方法调用时的权限要求。在代码清单 13-27 中，使用该注解声明了在调用 EmployeeService 接口的 read 方法时要具有“employee.info”对象上的“read”权限。

代码清单 13-27 添加了权限声明注解的接口

```
public interface EmployeeService {
    @DeclaredPermission(name="employee.info", actions="read")
    Employee read(String employeeId);
}
```

在声明了权限之后，需要在程序中的适当位置添加检查访问控制权限的逻辑，这就要求在调用某些方法之前和调用过程中添加适当代码。对于这类横切的功能，比较好的做法是使用动态代理机制进行处理。代码清单 13-28 给出了创建 EmployeeService 接口实现对象的工厂类的代码。在 EmployeeService 接口的每个方法被调用之前，会根据方法上声明的注解来获取所需的权限，并调用 AccessController 类的 checkPermission 方法进行权限检查。整个方法的执行是封装在 Subject 类调用的 doAsPrivileged 方法中的，其作用是以当前登录用户的身份标识来执行这个方法。SubjectHolder 类通过 ThreadLocal 类的对象来保存当前登录用户的 Subject 类的对象，由专门的过滤器（servlet filter）从会话中获取 Subject 类的对象，并将其保存在对应的 ThreadLocal 类的对象中。

代码清单 13-28 使用动态代理的工厂方法

```
public class EmployeeServiceFactory {
    private static class AccessControlInvocationHandler implements InvocationHandler {
        private Object targetObj;

        public AccessControlInvocationHandler(Object targetObj) {
            this.targetObj = targetObj;
        }

        public Object invoke(Object proxy, final Method method, final Object[] args)
            throws Throwable {
            Subject subject = SubjectHolder.get();
            return Subject.doAsPrivileged(subject,
                new PrivilegedExceptionAction<Object>() {
                    public Object run() throws Exception {
                        try {

```

```

        DeclaredPermission annotation = method.getAnnotation(
            DeclaredPermission.class);
        if (annotation != null) {
            BasePermission permission = new BasePermission
                (annotation.name(), annotation.actions());
            AccessController.checkPermission(permission);
        }
        return method.invoke(targetObj, args);
    } catch (ReflectiveOperationException e) {
        throw e;
    }
}
}, null);
}

public static EmployeeService getEmployeeService() {
    EmployeeService service = new EmployeeServiceImpl();
    ClassLoader cl = service.getClass().getClassLoader();
    return (EmployeeService) Proxy.newProxyInstance(cl, new Class<?>[] {
        EmployeeService.class }, new AccessControlInvocationHandler(service));
}
}

```

该案例应用使用的策略文件如代码清单 13-29 所示。在该策略文件中，为角色为“manager”的主体添加了“employee.info”对象上的“read”权限。

代码清单 13-29 案例使用的策略文件

```

grant principal com.java7book.securewebapp.RolePrincipal "manager" {
    permission com.java7book.securewebapp.BasePermission "employee.info", "read";
};

```

13.7 小结

构建安全的程序并不是一件容易的事情，需要多个方面互相配合。一个环节中出现错误，会导致严重的安全漏洞。Java 平台本身在安全性方面提供了丰富的支持，包括身份认证、权限控制、加密和解密、报文摘要、数字签名等。对于 SSL 协议也提供了良好的支持。这些支持可以作为程序本身安全性功能实现的基础。本章就 Java 的安全性相关的内容进行了介绍，掌握这些内容可以避免在开发中出现安全问题。

第 14 章 超越 Java 7

本书的前面章节都是围绕 Java 7 展开的。Java SE 7 在 2011 年 7 月发布之后，Java SE 8 的开发正在进行中。根据 Oracle 在 JavaOne 2011 会议上公布的计划，Java SE 8 预计于 2013 年夏天发布。虽然 Java SE 8 将包含的内容大部分已经确定，但是可能会随着具体的开发过程而发生改变。相对于 Java SE 7 而言，Java SE 8 将会对 Java 平台做出革命性的修改，其中包含的修改涉及 Java 语言、Java 标准库和 Java 虚拟机。Java SE 8 的目标是支持在不同的计算环境下创建出可维护的、可伸缩的和高性能的 Java 应用。Java SE 8 主要侧重三个方面，分别是生产率、性能和模块化。在生产率方面，Java SE 8 通过在 Java 语言和标准库中的改进，提升 API 的抽象层次，减少不必要的冗余代码；在性能方面，Java SE 8 对集合类框架和相关 API 做了增强，以支持对批量数据进行自动的并行处理，通过 lambda 表达式和接口的默认方法可以很方便地使用这些增强功能；在模块化方面，Java SE 8 对 Java 平台本身提供了模块化支持，把 Java 平台划分成不同的模块和组件，这种模块化技术也可以用在应用的开发中。

Java SE 8 中包含的更新内容由 JSR 337 (Java™ SE 8 Release Contents) 来确定，目前已经确定的具体内容包括：增加 lambda 表达式的 JSR 335 (Lambda Expressions for the Java™ Programming Language)、Java 平台模块化、对 Java 语言进行的一些小的改动、Java 类型的注解及日期与时间 API 等。除此之外，Java SE 8 的更新还包括 JavaFX 3.0、脚本语言支持 API 中新的 JavaScript 引擎以及虚拟机的整合等。本章的内容围绕 Java SE 8 的新特性展开。由于 Java SE 8 还在持续的开发中，很多内容都没有最终确定，因此本章内容以写作时（2012 年 2 月）为准。

14.1 lambda 表达式

lambda 表达式是 Java 开发人员长期以来一直期待 Java 语言增加的一个语法特性。使用 lambda 表达式可以通过简洁的代码来进行计算，减少了不必要的冗余代码。lambda 表达式主要用在两个地方：一个是操作新的集合类框架实现，另外一个是更好地使用已有的函数式接口。lambda 表达式实际上是创建函数式接口的一种简化的方式。lambda 表达式相关内容由 JSR 335 来规范。

在接下来的内容中，先介绍可以用 lambda 表达式来实现的函数式接口，再对 lambda 表达式的语法进行介绍。通过目标类型，编译器可以推断出 lambda 表达式所实现的接口类型，并进行必要的合法性检验。lambda 表达式中的代码与包围它的代码块在同一个词法作用域中，这会对代码中的名称解析产生影响。已有类中的方法也可以通过

方法引用的方式作为 lambda 表达式来使用。接口的默认方法则提供了一种与已有实现相兼容的方式来更新接口的定义。

14.1.1 函数式接口

Java 是一种面向对象的编程语言。在开发程序时，程序中的代码逻辑被封装在不同的类中。Java 类通常既包括用来维护内部状态的域，又包括用来执行业务逻辑和更新内部状态的方法。在有些情况下，程序所要做的只是简单地执行一段代码。比如在进行排序操作时，最核心的逻辑是判断任意两个对象之间的先后顺序。从函数的角度来说，这段代码接受两个对象作为输入，返回表示两个对象的先后顺序的值作为输出。在 Java SE 7 中，Java 语言并没有提供直接的语法结构来表示一个函数，与函数相近的结构是方法，但是方法不能脱离类而单独存在。因此，对于这样的场景，程序的一般做法是先定义一个接口，再定义接口的实现类。在需要使用方法时，先创建该实现类的对象，再调用对象中的方法。这个完整的过程比较繁琐。通过 Java 提供的匿名内部类可以简化一些操作。以 `java.lang.Runnable` 接口为例，当需要创建一个线程并启动该线程时，一般的做法如代码清单 14-1 所示。

代码清单 14-1 `Runnable` 接口的一般使用示例

```
new Thread(new Runnable() {
    public void run() {
        System.out.println("Hello World!");
    }
}).start();
```

从上面的代码可以看到，真正执行程序逻辑的有意义的代码其实只有 1 行，剩下的 4 行代码都是为了满足 Java 的语法要求而加上的，这就造成了一定的代码冗余。类似这样的代码在程序中会经常见到。由于相关的逻辑通常只在一个地方出现，没有必要把这些代码封装在另外一个单独的类中，因此，使用匿名内部类就成了开发人员最常见的选择。

Java 标准库中还存在其他与 `Runnable` 接口类似的接口，这些接口的特征是只声明了一个方法。具备这样特征的接口被称为函数式接口（functional interface）。对于函数式接口，通过匿名内部类实现的代码形式可以进一步简化。在代码清单 14-1 中，对 `Runnable` 接口中的 `run` 方法的声明是可以被去掉的。`Runnable` 接口中只包含一个方法，因此所包含的代码肯定是对这个唯一的方法的实现。同时对 `Runnable` 接口本身的声明也是可以被去掉的，可以从使用该匿名内部类的上下文环境中推断出使用的接口的具体类型。Java SE 8 中的 lambda 表达式利用了函数式接口的特点，提供了一种比匿名内部类更加简洁的方式来实现函数式接口。

Java 标准库中存在的函数式接口比较多，常见的包括 `Runnable`、`java.util.Comparator`、`java.io.FileFilter`、`java.util.concurrent.Callable`、`java.lang.reflect.InvocationHandler`、

`java.beans.PropertyChangeListener`、`java.awt.event.ActionListener` 和 `javax.swing.event.ChangeListener` 等。程序也可以定义自己的函数式接口。

14.1.2 lambda 表达式的语法

Java SE 8 中的 lambda 表达式提供了一种匿名方法的表示形式。它针对函数式接口的特征，提供了一种简洁的方式来替代匿名内部类的使用。代码清单 14-2 与代码清单 14-1 功能相同，但是代码清单 14-2 是使用 lambda 表达式来实现的代码。

代码清单 14-2 使用 lambda 表达式实现 Runnable 接口

```
new Thread(() -> {System.out.println("Hello World!");}).start();
```

可以看到，使用 lambda 表达式把代码清单 14-1 中的 5 行代码缩减成 1 行。lambda 表达式的基本形式是“(参数列表) -> 代码块”。lambda 表达式的表示形式类似于一般的方法，由形式参数和方法体两部分组成。形式参数包含在“()”中，“->”用来分隔形式参数和方法体。方法体可以是简单的表达式，也可以是复杂的代码块。复杂的代码块需要用“{}”包围起来。代码清单 14-3 给出了一些 lambda 表达式的示例。

代码清单 14-3 lambda 表达式的示例

```
(int x, int y) -> x + y
(int x) -> 100
() -> "Hello World"
(int x) -> {
    int num = 10;
    return num * x;
}
```

14.1.3 目标类型

lambda 表达式虽然在形式上类似于方法声明，但实际上是一种创建对象的方式。比如，代码清单 14-2 中的 lambda 表达式“() -> {System.out.println("Hello World!");}”实际上创建的是一个 `Runnable` 接口的实现对象。不过 lambda 表达式在声明时省略了所实现的函数式接口的具体类型，因此，同样的 lambda 表达式可能表示不同类型的对象。如果两个不同的函数式接口中的唯一方法的类型签名相同，那么可以用相同的 lambda 表达式来创建这两个接口的不同实现对象。lambda 表达式所表示的对象类型由 lambda 表达式所出现的上下文环境来确定，由编译器负责进行类型推断。编译器所进行的这种类型推断方式，与第 12 章中介绍的在泛型方法调用和 `<>` 操作符中使用的类型推断方式是相似的。

为了能够进行类型推断，lambda 表达式所出现的上下文环境中需要能够推断出一个具体类型。这个类型是 lambda 表达式的目标类型。可以推断出具体类型的上下文环

境包括变量声明、赋值操作、return 语句、数组初始化、一般方法或构造方法的参数、lambda 表达式的方法体、“?:”表达式和强制类型转换表达式等。如果从当前上下文环境中无法推断出具体的类型，那么 lambda 表达式无法出现在该上下文环境中。在确定了目标类型之后，编译器需要检查 lambda 表达式是否与目标类型兼容。兼容的条件是该目标类型是一个函数式接口，同时接口中的唯一方法的参数类型、返回值类型和抛出的受检异常都与 lambda 表达式的对应部分保持兼容。

在代码清单 14-2 中，lambda 表达式出现在 Thread 类的构造方法中，对应的上下文环境中的目标类型是 Runnable 接口。任何与 Runnable 接口中的 run 方法的参数类型、返回值类型和抛出的受检异常保持兼容的 lambda 表达式都可以使用。代码清单 14-4 给出了 lambda 表达式可以使用的其他上下文环境。

代码清单 14-4 lambda 表达式可以使用的上下文环境

```
Runnable r = () -> { System.out.println("Run"); }; // 赋值操作

public Comparator<String> createComparator() { //return 语句
    return (String s1, String s2) -> s1.compareTo(s2);
}

runTasks(new Runnable[] { () -> {}, () -> {} }); // 方法参数
```

由于目标类型的存在，通常可以省略 lambda 表达式中形式参数的类型声明，而由编译器根据目标类型进行自动推断，这样可以进一步减少冗余代码。如果 lambda 表达式中只有一个形式参数，那么参数列表外围的“()”也可以省略。

14.1.4 词法作用域

在编写 lambda 表达式的方法体时，需要注意的是方法体中声明的变量与包含该 lambda 表达式的上下文环境之间的关系。在这方面，lambda 表达式采用了一种简单的实现方式：lambda 表达式的方法体并没有引入新的作用域。方法体中的名称与表达式外部的代码处于同一个词法作用域中。在进行解析时，相当于方法体中的名称出现在表达式外部的代码中。而 lambda 表达式的形式参数也是属于表达式外部代码的名称。

代码清单 14-5 中的代码会出现编译错误。这是因为变量 str 在 lambda 表达式外部的代码中已经被定义了，在 lambda 表达式的方法体中又被重新定义了一次。由于这两个同名变量存在于同一个词法作用域中，因此编译器不允许多次声明该变量。

代码清单 14-5 lambda 表达式方法体与外部代码的名称重复的示例

```
String str = "Hello";
new Thread(() -> {
    String str = "World";
    System.out.println(str);
}).start();
```

lambda 表达式的这个特性对 this 变量也适用。对于 lambda 表达式方法体中出现的 this 变量，其值相当于 this 出现在表达式外部的代码中时应该具有的值。这点不同于匿名内部类。由于方法体中 this 变量的这个特性，方法体中的 this 变量无法用来引用 lambda 表达式所表示的对象本身。在某些递归方式的实现中，需要在方法体的代码中引用当前 lambda 表达式的对象。代码清单 14-6 给出了一个示例，其中的 lambda 表达式所表示的对象被赋值给变量 task，并在方法体中引用了这个变量。

代码清单 14-6 在 lambda 表达式的方法体中引用表达式所表示的对象

```
boolean done = false;
List<Runnable> taskQueue = new ArrayList<>();
final Runnable task = () -> {
    if (!done) {
        taskQueue.add(task);
    }
    else {
        // 任务完成
    }
};
```

14.1.5 方法引用

lambda 表达式提供了一种简洁的方式来创建函数式接口的实现对象。Java 类中的已有方法也可以被当成 lambda 表达式来使用，只需要直接引用已有的方法即可。方法引用相当于复用了 Java 类中已有方法的方法体。方法引用的使用条件与 lambda 表达式是相同的。代码清单 14-7 给出了方法引用的示例。在调用 Arrays 类的静态方法 sort 时，第二个参数是 Comparator 接口的实现对象。代码清单 14-7 中使用了已有类 Comparators 中的静态方法 compareString 作为 Comparator 接口的实现对象的内部实现逻辑。

代码清单 14-7 方法引用的使用示例

```
// 已有方法的声明
public class Comparators {
    public static int compareString(String s1, String s2) {
        return s1.compareToIgnoreCase(s2);
    }
}

// 方法引用的使用
String[] array = new String[] {"c", "b", "a"};
Arrays.sort(array, Comparators::compareString);
System.out.println(Arrays.deepToString(array));
```

在使用方法引用时，除了可以引用静态类中的方法外，还可以引用特定对象中的实例方法、特定类型的任意对象中的实例方法及构造方法。在引用对象的实例方法时，使

用对象的引用作为方法引用时的前缀，如类似“myObj::myMethod”的形式。如果引用的是任意对象中的实例方法，那么在引用时，使用方法所在的类型名称作为前缀，如类似“ MyClass::myMethod”的形式。在方法被调用时，实际的调用接收者对象会作为方法的第一个参数。例如，代码清单 14-7 中的字符串比较的逻辑可以简化成“ Arrays.sort(array, String::compareToIgnoreCase);”。在 String 类的 compareToIgnoreCase 方法被实际调用时，当前的 String 类的对象会作为 compareToIgnoreCase 方法的第一个参数。引用构造方法的形式类似于引用类中的静态方法，只不过方法的名称固定为“new”。例如，“ MyClass::new”用来引用类 MyClass 中的构造方法。如果一个类中定义了多个构造方法，那么方法引用被使用时的目标类型用于选择适用的构造方法。

14.1.6 接口的默认方法

引入默认方法的动机在于解决 Java 中接口所固有的更新困难的问题。在 Java 中，一旦接口被公开，对该接口进行修改是一件非常困难的事情。如果在新版本中为一个已有的公开接口添加一个新的方法，那么所有实现了该接口的已有类都会出现错误，因为这些类中没有添加这个新方法的实现。接口的默认方法提供了一种简洁的方式来解决这个问题。在接口中，除了方法的声明之外，还可以有方法对应的默认实现。如果该接口的实现类没有实现某个方法，而这个方法由接口提供了默认实现，那么该类会继承接口中的默认实现。在版本更新时，如果向接口中添加了新的方法，可以同时添加对应的默认实现。该接口的已有实现类会继承默认实现，而新的实现类则可以提供新的实现。通过这种方式，既不会对已有的接口实现类造成影响，又可以不断地进行更新。

下面通过一个示例来说明默认方法的作用。代码清单 14-8 是接口 Cursor 的已有声明，其中定义了两个方法 previous 和 next，分别用来向前和向后移动指针位置。

代码清单 14-8 Cursor 接口的已有声明

```
public interface Cursor {
    boolean previous();
    boolean next();
}
```

代码清单 14-9 是 Cursor 接口的新版本的声明，其中添加了两个新的方法 first 和 last，用来快速把指针移动到起始或末尾。在 first 和 last 方法的声明中， default 关键词之后是这两个方法的默认实现。默认实现通过调用已有的 previous 和 next 方法来完成。

代码清单 14-9 Cursor 接口的新版本的声明

```
public interface Cursor {
    boolean previous();
    boolean next();

    void first() default {

```

```

        while(previous()) {}
    }

    void last() default {
        while(next()) {}
    }
}

```

Cursor 接口的新的实现类可以提供 first 和 last 方法更加高效的实现。

随着 lambda 表达式的引入，Java 中的集合类框架会做出相应的调整，以方便开发人员在进行集合类的相关操作时使用 lambda 表达式。这就意味着需要对集合类框架中的很多接口进行更新。由于集合类使用广泛，这样的更新势必会造成已有的代码无法运行。通过接口的默认方法，可以很好地解决这个问题。

14.2 Java 平台模块化

模块化是目前软件开发中经常会用到的设计理念。把系统划分成不同的模块之后，开发团队可以分工合作，快速高效地完成开发工作。采用模块化设计的系统也更容易复用、维护和更新。不过 Java 平台本身并没有对模块化的系统的实现提供很多的支持。一般的做法是把 Java 类组织在不同的名称空间中，多个 Java 类组织在一个 jar 包中，多个 jar 包同时出现在程序的类路径（CLASSPATH）中。由于不同 jar 包中可能存在同名的 Java 类，加上复杂的类加载器实现机制，使 Java 程序经常出现与类加载相关的问题。由于 Java 平台本身存在限制，为 Java 平台提供模块化支持的 OSGi 技术得到了广泛的流行，这也从侧面反应了开发者社区对于模块化的需求。OpenJDK 中的 Jigsaw 项目的目标在于为 Java 平台增加模块化的支持，这也是 Java SE 8 的重要组成部分。

Jigsaw 项目为 Java 语言增加了模块的概念。模块是 Java 类型的集合。每个模块有自己的名称、一个可选的版本号，以及当前模块和其他模块之间关系的描述。模块之间最重要的关系是依赖关系。一个模块可以依赖其他的模块，并利用所依赖的模块提供的 Java 类。对于一个模块来说，模块本身的信息由名为 module-info.java 的 Java 文件来提供。代码清单 14-10 给出了 module-info.java 文件的内容示例。模块“com.my.base”的版本号是 1.0，依赖于版本号为 2.0 的“com.example”模块。通过“exports”声明了模块“com.my.base”中的“com.my.base.utils”包对依赖它的模块是可见的。通过“class”声明了模块“com.my.base”的主 Java 类是“com.my.base.MainApp”。

代码清单 14-10 module-info.java 文件的内容示例

```

module com.my.base @ 1.0 {
    requires com.example @ 2.0;
    exports com.my.base.utils;
    class com.my.base.MainApp;
}

```

```
module com.my.ui @ 1.0 {  
    requires com.my.base;  
}
```

在模块中，除了 module-info.java 文件之外，其他的内容与一般的 Java 程序中可以包含的内容相同，可以包括 Java 源代码、资源文件、配置文件和本地代码等。在编译过程中，模块中包含的源代码会被编译，然后进行打包和发布。打包好的模块可以安装到某个模块仓库中。这个仓库中包含了所有可用的模块。在运行时，虚拟机会负责从模块仓库中加载模块，并与它所依赖的其他模块进行链接，之后就可以运行该模块了。

14.3 Java SE 8 的其他更新

除了 lambda 表达式和模块化支持之外，Java SE 8 中的其他更新内容包括：

- 把 JRockit 虚拟机中的部分特性整合到 HotSpot 虚拟机中，提供一个统一的虚拟机实现。
 - 集成 JavaFX 3.0。在 Java SE 8 中会直接集成 JavaFX 3.0。
 - 在虚拟机上可以直接使用的新的 JavaScript 引擎，以及更好的 JavaScript 和 Java 代码之间的互操作性。新的 JavaScript 引擎被称为 Nashorn，是一个基于 JSR 292 的实现。
 - 在移动设备上，增加对多点触控、摄像头、地理位置信息、罗盘和重力加速器的支持。
 - 对 Java 安全、网络、国际化和可访问性 API 的更新。
 - 允许 Java 中的注解出现在类型的任意使用方式上，而不仅限于类型、方法、域和变量的声明中。注解可以使用的位置包括方法调用的接收者、泛型类型参数、数组、强制类型转换、instanceof 操作符、对象创建、泛型中类型参数的上界或下界、类继承关系和 throws 子句。具体的细节由 JSR 308 (Annotations on Java Types) 规范来描述。
 - 新的日期和时间 API，用来解决当前使用 java.util.Date 类和 java.util.Calendar 类处理日期和时间中产生的问题。具体的细节由 JSR 310 (Date and Time API) 规范来描述。
- 对于上述更新，由于相关的内容正在开发中，还没有最终确定，本章不进行具体的介绍。

14.4 小结

在 Java SE 7 发布之后，Java SE 8 的开发正在进行中。相对于 Java SE 7 而言，Java SE 8 将是一个包含很多重要更新的版本。由于 Java SE 8 中的很多新的内容还没有最终确定。本章只对 Java SE 8 中两个最重要的更新——lambda 表达式和模块化支持进行了介绍。希望通过这些介绍，使读者对 Java SE 8 中将要发生的变化有一定的了解。

附录 A OpenJDK

在 Java 语言产生之后的很长一段时间内，Java 语言的规范制定以及类库和运行环境的开发工作，都是由 Sun 公司独自完成的，都是 Sun 公司的私有实现。在 Java 发展的初期，这种方式避免了繁琐的流程，降低了对外的沟通成本，是有利于 Java 语言的快速发展的。但是随着 Java 语言的日益流行，这种 Sun 公司的私有实现模式已经不能适应发展的要求了。首先是广大 Java 开发者对 Java 的需求越来越多，单凭 Sun 公司一己之力难以快速应对。另外对那些采用了 Java 平台的企业来说，一个很现实的担忧是 Java 平台的供应商锁定（vendor lock-in）问题。如果自己公司的核心业务系统都基于 Java 平台来构建，而这个平台本身却是另外一家公司的私有技术，有这样的顾虑是合情合理的。Sun 公司也意识到了这一点，于是开始了 Java 平台的开放化进程。

首先开放的是 Java 平台的规范。Sun 公司公开了 Java 语言规范和 Java 虚拟机规范。Java 语言规范详细描述了 Java 语言的语法和重要特性。Java 虚拟机规范则规定了 Java 字节代码格式和执行 Java 字节代码的虚拟机的相关细节。前面介绍了 Java 程序社区 JCP，Sun 公司正是依托这个社区的流程来规范对 Java 平台所做的各种修改，同时依靠社区的力量来完善 Java 平台本身。社区对 Java 平台更新的贡献的一个典型例子是 J2SE 5.0 中引入的同步实用类库 `java.util.concurrent` 包。在这之前，使用 Java 进行多线程编程只能使用最基本的几个原语，不但复杂而且容易出错。Doug Lea 领导开发了方便多线程开发的实用工具包。这个工具包最初是独立分发的，后来正式成为 Java 平台的一部分。

另外一部分需要开放的是 Sun 自己的 Java 虚拟机实现、Java 类库实现和编译器等实用工具。Sun 公司于 2006 年宣布 Java 将成为开放源代码的软件。为了实践这个开放策略，Sun 开发的 HotSpot 虚拟机和编译器最先成为使用 GPL 协议的自由软件。接着 Java 类库中的绝大部分都按照 GPL 协议开放了源代码。而对于剩下的小部分由于版权原因无法开放源代码的类库，也在社区的努力下找到了替代的开源实现。至此，Java 语言终于有了一个自由的开放源代码的实现，即 OpenJDK。

OpenJDK 的出现正在对 Java 语言产生着积极而深远的影响。依托于社区的开放源代码模式，使 Java 平台可以依靠社区的力量快速发展。而这种模式也避免了供应商锁定的问题。值得一提的是，OpenJDK 的出现并不阻碍其他私有的 JDK 的发展。其他公司仍然可以针对不同的应用环境开发出更加适合的 JDK，这些私有的 JDK 也可以自由地使用 OpenJDK 作为其实现的基础。

目前的 OpenJDK 主要有 JDK 7 和 JDK 6 两个版本。OpenJDK 7 是目前主要开发的版本，也是 Java SE 7 的参考实现。OpenJDK 6 则是 Java SE 6 的一个开放源代码的实现，主要被用在 Fedora 等 Linux 分发平台上。

在 Oracle 公司收购了 Sun 公司之后，Oracle 公司并没有改变在 Java 平台上的开放策略，而是继续支持 OpenJDK 的发展。随后，Oracle 同 IBM 和苹果公司都建立了合作关系，共同推进 OpenJDK 的发展。

由于 OpenJDK 是 Java SE 7 的参考实现，所以本书中的所有示例代码都是基于 OpenJDK 7 进行开发和调试的。

附录 B Java 简史

相信本书的读者对于 Java 语言肯定都有一定的了解。从 1995 年至今，Java 语言经过了 16 年的发展和完善，目前已经成为软件开发中的主流编程语言。数以千万计的开发人员使用 Java 语言在不同平台上开发出各种不同的应用，包括 Web 应用、桌面应用和移动设备上的应用等。在 TIOBE 给出的编程语言流行程度统计排名中，Java 语言常年占据首位。这个排名直观地反映了 Java 语言在开发者社区中的强大影响力。

对于一种有着悠久历史的编程语言来说，其历史本身既是一种财富，也是一种负担。庞大的开发者社区为这门语言的演化提供了坚实的用户基础，也为这门语言贡献了非常多的可复用的资产。从另外一方面来说，必须保持的后向兼容性也使得 Java 语言在应对变化时显得力不从心。当 Ruby 程序员可以把基本类型，如整数和浮点数，当做功能完备的对象来使用时，在 Java 世界中，基本类型和对象还是分开的。Java 程序员享受不到 Ruby 中 `3.next` 和 `5.times` 那样简洁易用的语法。虽然 Java SE 5.0 引入了基本类型的自动装箱和拆箱机制，缓解了这个问题，但是依然无法实现 Ruby 那样的灵活性。当然，Java 语言在最初设计的时候，肯定权衡了各种选择的优缺点，做出了当时最合理的设计选择。通过了解一门编程语言的历史，可以深刻理解其发展过程中各种变化背后的动机。下面简要地介绍一下 Java 语言的发展历史。

1991 年，James Gosling、Mike Sheridan 和 Patrick Naughton 在 Sun 公司开始了一种新的编程语言的设计和开发工作。这种新的语言换过好几个名字，从最初的 Oak 到后来的 Green，直到最后被定名为 Java。提到 Java，就会想到印度尼西亚群岛中盛产咖啡的爪哇（Jawa）岛。正因如此，Java 语言的标志是一杯咖啡的形状。在最初的时候，这门语言是以与计算机相连接的智能消费电子产品作为目标平台而设计的。Java 语言最早运行在数字有线电视的控制器上。不过，这项技术对当时的产业来说，过于超前了。幸运的是，Java 赶上了从 1995 年开始的互联网发展的上升期。Java 语言乘着互联网的东风，以极快的速度发展起来。

1995 年底，Sun 公司正式发布 Java 语言。在随后不久的 1996 年初，JDK 1.0 发布。当时 Java 语言的杀手级应用（killer app）就是 Java applet。当时主流的浏览器都支持 Java applet。在当时，HTML 语言所能提供的交互性能力还比较弱，而用户对于互联网应用的热情却非常高涨。这就在 Web 应用的交互能力和用户的需求之间产生了一个缺口。Java applet 的出现正好填补了这个空白。Java applet 提供了丰富的用户界面组件，以及 2D 图形绘制能力，其所提供的交互性体验远优于当时的 HTML，因此 Java applet 得到了广泛流行。Java applet 的代码是从远程服务器上下载到用户的本地浏览器中来运行的。这种需求催生了 Java 语言的一个重要特性，那就是类加载器。动

态类加载的概念被认为是 Java 语言唯一的重要创新。本书的第 9 章专门对类加载器进行介绍。

在 JDK 1.0 之后, JDK 1.1 于 1997 年发布。在这个版本中, 一些新特性被加入进来, 包括内部类、JavaBeans、JDBC、RMI 和反射等。JDK 1.1 发布三个星期之后, 就达到了 22 万的下载量, 其关注程度可见一斑。

从 JDK 1.1 之后的版本开始, Java 为不同的目标平台提供了不同的配置, 这就是现在为开发人员所熟知的标准版、企业版和移动版。Sun 公司也启用了 Java 2 这样一个新名称。这些不同的配置也被对应地称为 Java 2 标准版 (J2SE)、Java 2 企业版 (J2EE) 和 Java 2 移动版 (J2ME)。J2SE 1.2 于 1998 年发布, 其中最重要的更新是添加了 Swing 用户界面库和集合类框架。Swing 用户界面的引入, 使 Java 在桌面应用开发中占据了一席之地。而 Java 的集合类框架则一直被认为是架构和 API 设计的良好典范, 其主要设计和开发者是大名鼎鼎的《Effective Java》的作者 Joshua Bloch。

Java 2 的版本更新以较快的速度进行。2000 年发布的 J2SE 1.3 中最重要的改进是使用 HotSpot 作为默认的 Java 虚拟机, 极大地提升了 Java 程序的运行性能, 在一定程度上缓解了一直以来为开发者所诟病的性能问题。动态代理机制也在这个版本中被加入, 允许以简单的方式来实现面向方面编程 (AOP)。

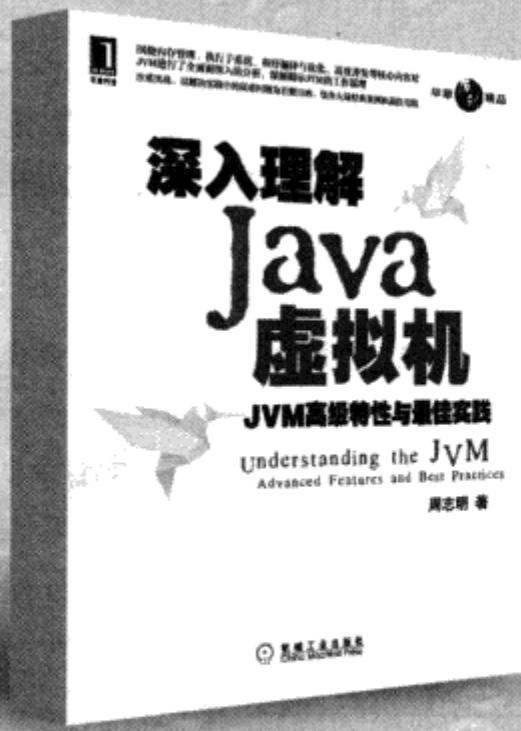
随着 Java 语言的不断流行, 对 Java 平台的需求也日益增多。如何让 Java 语言健康有序地发展, 成为 Sun 公司亟待解决的一个问题。Sun 公司尝试向国际标准化组织靠拢, 以寻求 Java 语言的标准化。后来 Sun 公司放弃了这种努力, 转而拥抱广大的开发者社区, 采用社区驱动的方式来促进 Java 语言的发展。1998 年, Java 程序社区 (Java Community Process, JCP) 成立, 其中包含了对 Java 感兴趣的公司、研究机构和个人。对 Java 平台的改进都通过 JCP 来运作。对 Java 平台所做的每一个修改都以 Java 规范请求 (Java Specification Request, JSR) 的形式来提交给 JCP 进行审阅和批准, 最终成为 Java 平台的一部分。

2002 年, 首个采用 JCP 方式开发的 Java 平台 J2SE 1.4 发布。在这个版本中, 更多的类库被加入进来, 包括正则表达式、非阻塞 IO (NIO)、日志记录 API、XML 和 XSLT 支持以及安全和加密功能等。

2004 年发布的 J2SE 5.0 是 Java 语言发展历史上的一个重要的版本。在这个版本中, Java 增加了很多语法上的新特性, 包括泛型支持、注解、基本类型的自动装箱和拆箱、枚举类型、参数长度可变的方法、增强的 for 循环和静态引入等。而在类库方面, 并发实用类库 `java.util.concurrent` 的引入, 极大地降低了并发应用开发的复杂度。

从 J2SE 5.0 的下一个版本开始, Sun 又对不同目标平台的版本改换了新的名称, 分别是 Java SE、Java EE 和 Java ME。Java SE 6 于 2006 年发布。这个版本的新特性主要体现在对脚本语言的支持、Java 编译器 API 和可插拔式注解等方面。Java SE 6 另一个重要提升是在性能方面, 包括核心平台和 Swing 用户界面库的性能都有了很大的提升。在正式版本发布之后, 每隔一段时间就会有小的更新修订版本发布。

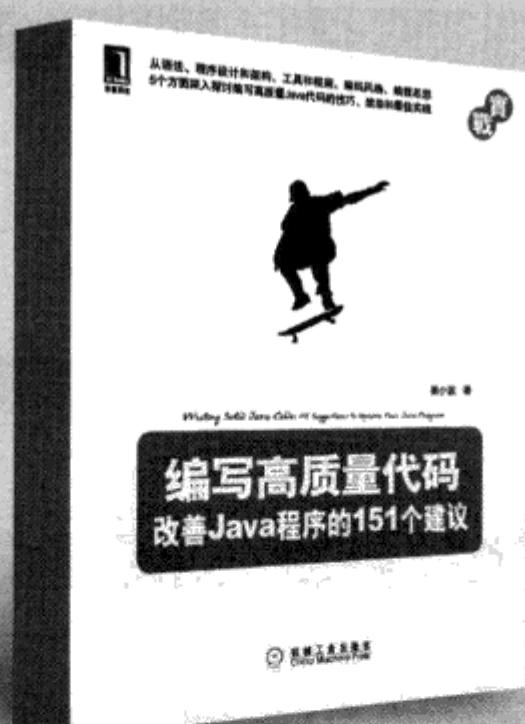
Java SE 6 正式发布之后相当长的一段时间内，Java 平台都没有大版本的更新。而 Java 7 的发布时间是在 2011 年的 7 月底。从 Java 6 到 Java 7 的时间跨度来说，开发者社区已经对 Java 平台的这个新版本期待了很长的时间。Java 7 及其后续的版本所包含的改动，会对 Java 语言本身产生深远而重大的影响。Java 7 中所包含的新特性和变化正是本书的重要组成部分。



Java领域超级畅销书，9个月7次印刷，繁体版即将在中国台湾发行
围绕内存管理、执行子系统、编程编译与优化、高效并发等核心内容
对JVM进行全面而深入的分析，深刻揭示JVM的工作原理
注重实践，以解决实践中的疑难问题为首要目的，包含大量经典案例
和最佳实践



Java领域畅销书，繁体版在中国台湾同步发行
Java安全领域的百科全书和权威经典，开发企业
级Java应用的必备参考手册



Java领域畅销书，繁体版即将在中国台湾发行
从语法、程序设计和架构、工具和框架、编码风格、
编程思想5个方面深入探讨编写高质量Java代码的技
巧、禁忌和最佳实践



专业成就人生
立体服务大众
www.hzbook.com

填写读者调查表 加入华章书友会
获赠精彩技术书 参与活动和抽奖

尊敬的读者：

感谢您选择华章图书。为了聆听您的意见，以便我们能够为您提供更优秀的图书产品，敬请您抽出宝贵的时间填写本表，并按底部的地址邮寄给我们（您也可通过www.hzbook.com填写本表）。您将加入我们的“华章书友会”，及时获得新书资讯，免费参加书友会活动。我们将定期选出若干名热心读者，免费赠送我们出版的图书。请一定填写书名书号并留全您的联系信息，以便我们联络您，谢谢！

书名：

书号：7-111-()

姓名：	性别： <input type="checkbox"/> 男 <input type="checkbox"/> 女	年龄：	职业：
通信地址：		E-mail：	
电话：	手机：	邮编：	

1. 您是如何获知本书的：

朋友推荐 书店 图书目录 杂志、报纸、网络等 其他

2. 您从哪里购买本书：

新华书店 计算机专业书店 网上书店 其他

3. 您对本书的评价是：

技术内容	<input type="checkbox"/> 很好	<input type="checkbox"/> 一般	<input type="checkbox"/> 较差	<input type="checkbox"/> 理由_____
文字质量	<input type="checkbox"/> 很好	<input type="checkbox"/> 一般	<input type="checkbox"/> 较差	<input type="checkbox"/> 理由_____
版式封面	<input type="checkbox"/> 很好	<input type="checkbox"/> 一般	<input type="checkbox"/> 较差	<input type="checkbox"/> 理由_____
印装质量	<input type="checkbox"/> 很好	<input type="checkbox"/> 一般	<input type="checkbox"/> 较差	<input type="checkbox"/> 理由_____
图书定价	<input type="checkbox"/> 太高	<input type="checkbox"/> 合适	<input type="checkbox"/> 较低	<input type="checkbox"/> 理由_____

4. 您希望我们的图书在哪些方面进行改进？

5. 您最希望我们出版哪方面的图书？如果有英文版请写出书名。

6. 您有没有写作或翻译技术图书的想法？

是，我的计划是_____ 否

7. 您希望获取图书信息的形式：

邮件 信函 短信 其他_____

请寄：北京市西城区百万庄南街1号 机械工业出版社 华章公司 计算机图书策划部收

邮编：100037 电话：(010) 88379512 传真：(010) 68311602 E-mail: hzjsj@hzbook.com

[General Information]

书名 = 深入理解 Java 7 核心技术与最佳实践

作者 = 成富著

页数 = 452

出版社 = 北京市 : 机械工业出版社

出版日期 = 2012.05

SS号 = 13010858

DX号 = 000008284444

URL = http://book.szdnet.org.cn/bookDetail.jsp

?dxNumber=000008284444&d=3B276294774CDC9A8

471DB7A613CB9BD