HZ BOOKS
华章科技

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

华章原创精品

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

机械工业出版社
China Machine Press

深入理解 Java 7：
核心技术与最佳实践

成 富 著

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。

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
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，每年有非常多新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

第 1 章　Java 7 语法新特性

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(0B001110); // 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的错误，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。这样异常处理者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 LocalizedException。

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。这么做的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另外一个的子类的情况，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重要性。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的接口是 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 3×3×3 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(LoggingInvocationHandl
        er.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 虚拟机本身的动态性。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，toBeSpreaded 方法所接受到的参数 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 类似的做法，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, Illegal-
    AccessException{
    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("privateMe
        thod");
    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的方法句柄。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可变长度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。用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正确的方法句柄。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。

最后一个在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 initalValue, MethodHandle
    handle) throws Throwable {
    Object result = initalValue;
    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[⊖]。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 {
```

⊖ 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多的。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。如果不符合条件，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.io.CharArrayReader 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 <= 标记位置 <= 读写位置 <= 读写限制 <= 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中。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，也可以通过对应类的 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的视图。

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);
        }
    }
```

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复本中。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，应该通过对应的 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的操作时，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 getFileAttributeView 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，通过对应的 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应的 read 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))
```

```
                .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新功能，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。

Java程序员参考书！

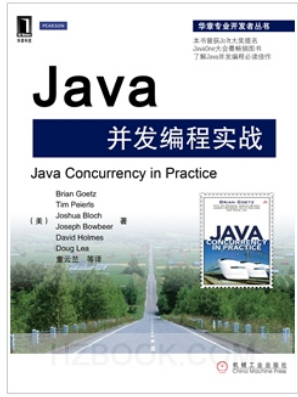

JAVA
核心技术 卷Ⅰ：基础知识
Core Java , Volume Ⅰ : Fundamentals
Eighth Edition
Cay S. Horstmann
Gary Cornell
第6版，畅销30000册
第7版，畅销50000册
第8版，让经典无限延伸……


计算机科学丛书
原书第8版
Java语言程序设计
基础篇
Introduction to Java Programming
Eighth Edition
Y. Daniel Liang


计算机科学丛书
原书第8版
Java语言程序设计
进阶篇
Introduction to Java Programming
Eighth Edition
Y. Daniel Liang

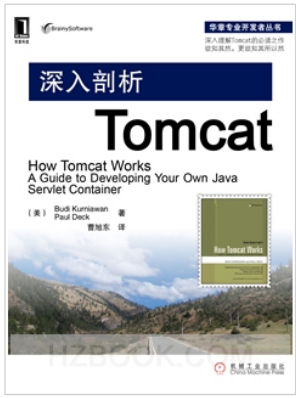

计算机科学丛书
Java编程思想
Thinking in Java
THINKING IN JAVA
第4版


Sun公司核心技术丛书
Effective Java中文版第2版
Effective Java Second Edition
(美) Joshua Bloch 著
Java之父James Gosling


Java
并发编程实战
Java Concurrency in Practice
Brian Goetz
Tim Peierls
Joshua Bloch
Joseph Bowbeer
David Holmes
Doug Lea


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


深入理解
Java
虚拟机
JVM高级特性与最佳实践
Understanding the JVM
Advanced Features and Best Practices
周志明 著


深入剖析
Tomcat
How Tomcat Works
A Guide to Developing Your Own Java Servlet Container
Budi Kurniawan
Paul Deck


Spring Internals
Spring 技术内幕
深入解析Spring架构与设计原理
（第2版）
计文柯 著


Struts2 Internals
Struts2 技术内幕
深入解析Struts2架构设计与实现原理
陆舟 著


Maven实战

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。

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

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