华章科技

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

InfoQ 软件开发丛书

免费在线版本

（非印刷免费在线版）

了解本书更多信息请登录本书的官方网站

InfoQ 中文站出品

本书由 InfoQ 中文站免费发放，如果您从其他渠道获取本书，请注册 InfoQ 中文站以支持作者和出版商，并免费下载更多 InfoQ 软件开发系列图书。

本迷你书主页为

http://www.infoq.com/cn/minibooks/know-more-about-java7

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！

目录

第 1 章　Java 7 语法新特性 5
1.1　Coin 项目介绍 5
1.2　在 switch 语句中使用字符串 2
1.3　数值字面量的改进 5
1.4　优化的异常处理 7
1.5　try-with-resources 语句 17
1.6　优化变长参数的方法调用 19
1.7　小结 21
第 2 章　Java 语言的动态性 22
2.1　脚本语言支持 API 22
2.2　反射 API 31
2.3　动态代理 36
2.4　动态语言支持 42
2.5　小结 73
第 6 章　Java 7 其他重要更新 75
6.1　关系数据库访问 75
6.2　java.lang 包的更新 191
6.3　Java 实用工具类 195
6.4　JavaBeans 组件 201
6.5　小结 203
第 14 章　超越 Java 7 205

14.1 lambda 表达式 205

14.2 Java 平台模块化 444

14.3 Java SE 8 的其他更新 445

14.4 小结 445

InfoQueue

促进软件开发领域知识与创新的传播

中文 | 英文 | 日文 | 葡文 | ……

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。

QClub

我们影响有影响力的人

北京 上海 广州 大连 西安 太原 成都 杭州 武汉 南京 深圳...

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，

Powered by InfoQ

QCon 2012

全球软件开发大会 [杭州站]

International Software Development Conference

百余位讲师，
千人规模……

7折 2240元

立减 1080 OFF 元

知名网站案例分享
Architectures you always wonder about

高效运维
Effective Operation

PaaS引擎已经启动
PaaS Engine has started

可编程的Web
Programmable Web

Java 20年
Java: the past 20 years, and the future

安全，从一点一滴做起
Security Ground-up

构建高效能团队
How to build highly productive team

让移动用户High起来
Get your APP's user high

优秀测试实践分析
The Best of The Test

10月25-27 | 杭州之江饭店

主办方：InfoQ 战略合作伙伴：Alibaba Group 阿里巴巴集团 www.infoq.com/cn www.qconhangzhou.com

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

ArchSummit

全 球 架 构 师 峰 会

www.archsummit.com

中国 · 深圳 · 大梅沙

2012.08.10~08.12

国际讲师

来自Facebook、Siemens、VMware、LinkedIn等公司

国内讲师

来自腾讯、新浪、淘宝、百度、Yahoo!中国、搜狗、土豆、诺基亚西门子、IBM等公司

精选演讲

- 淘宝技术大学校长 赵超：淘宝网的架构演进和变化
- 搜狗搜索事业部总经理 茹立云：深层网络搜索核心技术研讨
- 土豆网架构师 李明杰：视频播放编码与协议交付架构
- 前Facebook数据基础设施团队主管 Ashish Thusoo：Facebook的海量数据处理架构
- 腾讯手机浏览器后台开发组总监 张凯：手机QQ浏览器X架构剖析
- 南京韩海源科技CEO、知名黑客FLASHSKY 方兴：基于安全漏洞的攻防对抗技术
- 腾讯云平台部总经理 陈磊：构建高可用与持续运营的云平台
- 虎扑网研发经理 洪涛：从林书豪到全明星 - 虎扑网技术架构如何化解流量高峰
- IBM中国全球服务中心高级构架师 陈良国：击败人类万事通的Watson AI系统架构剖析

ArchSummit

现已开始售票

TICKET

7月15日前，8折优惠！

另剩余少量特别票*

*特别票包含

ArchSummit8折票一张

万科蓝色会所公寓酒店（四星级）

豪华套房一间（3晚住宿及双早）

主办方：InfoQ　战略合作伙伴：Tencent 腾讯　特别支持：

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.visit InvokeDynamicInsn("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 6 章　Java 7 其他重要更新

本书前面几章从不同的方面介绍了 Java 7 中比较重要的更新，这些更新的介绍不是独立进行，而是围绕 Java 平台上的某个特定主题展开。在对这些主题进行介绍的时候，既介绍了 Java 7 之前已有的概念，又介绍了 Java 7 的新特性，目的在于让读者全面了解一个主题各个方面的相关知识。本章则以 Java 7 中的其他重要更新为主要内容，同时也包括了其他相对较小却值得关注的改动。

在内容组织方式上，本章相对比较松散，每一节围绕一个小的主题展开。对于某些小改动，合并在一个小节里面进行介绍。在具体的内容上，以 Java 7 的新特性为主，并适当介绍相关的背景知识。

6.1　关系数据库访问

关系数据库访问一直是 Java 平台的重要功能，尤其对 Web 应用开发来说，大多数 Web 应用都是由关系数据库来驱动的。Java 平台的数据库访问方式由 JDBC（Java Data Base Connectivity）规范来定义。JDBC 规范本身也在不断更新，以适应数据库技术的发展，同时满足开发人员的使用需求。Java 7 在数据库访问方面的重要更新是增加了对 JDBC 4.1 规范的支持，Java 6 支持的仅是 JDBC 4.0 规范。相对于 JDBC 4.0 来说，JDBC 4.1 规范又引入了一些新的特性，这些特性都可以在 Java 7 中使用。需要注意的是，不同数据库实现的 JDBC 驱动对 JDBC 4.1 规范的支持程度是不相同的。特定的数据库实现有可能暂时还不支持某些新特性。JDBC 4.1 规范所更新的特性在 java.sql 和 javax.sql 包中都有，下面会分别进行介绍。

6.1.1　使用 try-with-resources 语句

在第 1 章介绍了 Java 7 通过新增的 try-with-resources 语句来管理 Java 平台上的各种资源。与数据库访问相关的各种资源，包括数据库连接、查询语句和结果集等，都可以用 try-with-resources 语句来进行管理。通过 try-with-resources 语句可以去掉数据库操作时对 close 方法的调用，大大降低了代码的复杂性。在 Java 7 中，java.sql 包中的 java.sql.Connection、java.sql.Statement 和 java.sql.ResultSet 接口都继承了 java.lang.AutoCloseable 接口，以支持由 try-with-resources 语句来管理。代码清单 6-1 给出了使用 try-with-resources 语句进行数据库操作的示例。可以把对 Connection、Statement 和 ResultSet 的创建都封装在 try-with-resources 语句中，这样就不用考虑这

些对象的关闭操作。

代码清单 6-1　用 try-with-resources 语句进行数据库操作的示例

```
public void dbOperation() throws SQLException {
    try (Connection connection = DriverManager
            .getConnection("jdbc:derby://localhost/java7book");
            Statement stmt = connection.createStatement();
            ResultSet rs = stmt.executeQuery("SELECT * FROM book")) {
        while (rs.next())
            { System.out.println(rs.getString("name")
            );
        }
    }
}
```

6.1.2　数据库查询的默认模式

大部分关系数据库系统都支持为数据库中包含的表和其他对象创建一个额外的名称空间，即模式（schema）。一个模式中可以包含表、视图以及其他对象。不同模式中可以有名称相同的表或视图，而同一模式中不允许有名称相同的对象存在。通过 SQL 语句“CREATE SCHEMA”可以创建新的模式。当访问某个模式中包含的表或视图等对象时，需要使用模式名称作为前缀来访问，否则无法找到相应的对象。比如，对于一个名为“DEMO_SCHEMA”的模式中的“author”表，在 SQL 语句中应该使用“DEMO_SCHEMA.author”来引用。如果每次都要加上模式名称作为前缀，那么使用起来会比较麻烦。JDBC 4.1 为 Connection 接口添加了一对新的方法 getSchema 和 setSchema，用来获取和设置数据库操作时使用的默认模式名称。当通过 setSchema 进行设置之后，在 SQL 语句中就不再需要使用模式名称作为前缀了。代码清单 6-2 给出了使用 setSchema 方法的示例。一般来说，如果希望使用 setSchema 方法，那么应该在从 Connection 接口中创建出来的 Statement 对象被使用之前进行设置，否则可能会造成默认的模式无法生效。

代码清单 6-2　setSchema 方法的使用示例

```
public void setSchema() throws SQLException {
    try (Connection connection = DriverManager
            .getConnection("jdbc:derby://localhost/java7book")) {
        connection.setSchema("DEMO_SCHEMA");
        try (Statement stmt = connection.createStatement();
                ResultSet rs = stmt.executeQuery("SELECT * FROM author")) {
            while (rs.next())
                { System.out.println(rs.getString("name")
                );
            }
        }
    }
}
```

6.1.3 数据库连接超时时间与终止

在使用数据库连接时会遇到的一个问题是，网络连接出现问题造成数据库连接中断。当数据库连接中断时，可能会造成发出数据库操作命令的线程在较长时间内处于等待状态。具体的等待时间取决于 TCP 连接的超时时间设置。这个超时时间一般会长达几分钟。也就是说，在数据库操作命令发出之后，可能需要等待几分钟才能得知数据库连接已经中断了。对于一个数据库应用来说，几分钟的等待时间过长，为此，JDBC 4.1 添加了对数据库连接超时的处理方式，主要是在 Connection 接口中新增了 setNetworkTimeout 和 abort 两个方法。

Connection 接口的 setNetworkTimeout 方法用来设置通过此数据库连接进行数据库操作时的超时等待时间。如果远程数据库没有在给定的时间内返回操作结果，那么会认为该连接已经关闭，无法再继续使用，处于等待状态的数据库操作方法则会抛出 SQLException 异常来表示这种超时的情况。对一个 Connection 接口的实现对象设置超时等待时间之后，会影响到从该 Connection 接口的实现对象中创建的其他对象，主要包括 Statement 和 java.sql.PreparedStatement 接口的实现对象。通过 Statement 和 PreparedStatement 接口的实现对象执行的查询操作也适用于 Connection 接口的实现对象上设置的超时等待时间。可以多次调用 setNetworkTimeout 方法，以对不同的情况设置不同的超时时间。对于同一个 Connection 接口的实现对象，如果预计某些查询操作比较耗时，可以把超时等待时间设置为一个较大的值，等操作完成之后，再恢复为一个对大多数操作都适用的较小值。

与 setNetworkTimeout 相关的方法 abort 用来强制关闭一个数据库连接。当调用一个 Connection 接口的实现对象的 abort 方法时，这个数据库连接会被标记为关闭状态，同时与该连接相关的资源都会被释放，另外，当前正在使用该连接的方法会抛出 SQLException 异常。从作用上讲，abort 方法类似于 Connection 接口中已有的 close 方法，但是两者的使用场景不同：close 方法一般是由 Connection 接口的使用者来调用的，当一个使用者完成数据库操作之后，可以通过 close 方法来关闭此数据库连接；而 abort 方法一般由数据库连接的管理者来调用，如果发生了前面提到的由于网络问题造成数据库连接中断的情况，连接的管理者在检测到问题发生之后，可以调用 abort 方法来强制终止此连接，该连接的使用者也不需要等待数据库操作的完成即可进入到 SQLException 异常的处理阶段。

setNetworkTimeout 和 abort 方法都接收一个 java.util.concurrent.Executor 接口的实现对象作为参数。这个 Executor 接口的实现对象用来执行方法调用中可能会出现的相关任务。对于 abort 方法来说，在终止数据库连接的过程中需要释放相关的资源。释放资源的相关任务由该 Executor 接口的实现对象来执行。当 abort 方法返回之后，Executor 接口的实现对象可能仍在继续执行相关的任务。代码清单 6-3 给出了 abort 方法的使用示例。为了查看在 abort 方法调用过程中执行的相关任务，这里使用了一个自定义的 java.

util.concurrent.ThreadPoolExecutor 类的实现。在任务被执行之前，会在控制台输出相关的提示信息。在运行过程中可以发现，在调用 abort 方法之后，有新的任务被添加到 Executor 接口的实现对象中执行。

代码清单 6-3　abort 方法的使用示例

```
public class AbortConnection {
    public void abortConnection() throws SQLException
        { Connection connection = DriverManager
            .getConnection("jdbc:derby://localhost/java7book");
        ThreadPoolExecutor executor = new DebugExecutorService(2, 10, 60,
            TimeUnit.SECONDS, new LinkedBlockingQueue<Runnable>());
        connection.abort(executor);
        executor.shutdown();
        try {
            executor.awaitTermination(5, TimeUnit.MINUTES);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    private static class DebugExecutorService extends ThreadPoolExecutor {
        public DebugExecutorService(int corePoolSize, int maximumPoolSize,
                long keepAliveTime, TimeUnit unit,
                BlockingQueue<Runnable> workQueue) {
            super(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue);
        }

        public void beforeExecute(Thread t, Runnable r) {
            System.out.println("清理任务 :" + r.getClass());
            super.beforeExecute(t, r);
        }
    }
}
```

6.1.4　语句自动关闭

在 JDBC 4.1 之前，当使用一个 Statement 接口的实现对象进行查询，得到表示结果集的 ResultSet 接口的实现对象并完成处理之后，需要显式地通过 close 方法来关闭 ResultSet 和 Statement 接口的实现对象以释放资源。使用 Java 7 的 try-with-resources 语句可以简化这个操作，而更好的办法是使用 JDBC 4.1 为 Statement 接口添加的自动关闭功能。在调用了 Statement 接口的 closeOnCompletion 方法之后，表明当依赖此 Statement 接口的实现对象的所有结果集都被关闭之后，该 Statement 接口的实现对象也被自动关闭。通过这种方式，开发人员只需显式地关闭结果集即可，不需要考虑 Statement 接口的实现对象本身的关闭问题。这个新特性在 Statement 接口的实现对象作

为参数在多个方法中传递时尤为实用。当代码中的多个地方都使用了相同的 Statement 接口的实现对象来打开结果集时，以前需要设计好由哪个方法来关闭该 Statement 接口的实现对象，而使用这个自动关闭功能之后，每个方法只需要关闭该方法内部打开的结果集即可，不需要考虑 Statement 接口的实现对象自身的关闭问题。

6.1.5 RowSet 实现提供者

javax.sql.RowSet 接口是 JDBC 2.0 引入的表格式数据的抽象表示形式。RowSet 接口继承自 ResultSet 接口，并在 ResultSet 接口的基础上添加了属性设置和变化通知相关的功能。RowSet 接口是符合 JavaBeans 组件规范的，可以与其他 JavaBeans 组件协同使用。RowSet 接口的使用者并不需要显式地管理数据库连接，只需要把数据库相关的连接信息以属性的方式设置到 RowSet 接口的实现对象中即可。RowSet 接口的实现会负责处理数据库连接的相关工作。RowSet 接口有 5 个不同的子接口，每个子接口实现所适应的场景不同。每个子接口对应的具体实现类也有不少，分别来自不同的提供者。在 Java 7 之前并没有一个规范的方式来管理 RowSet 接口的实现对象的创建，开发人员需要直接根据实现类的类名来创建新的 RowSet 接口的实现对象。这种直接依赖具体类而不是接口的做法，不利于代码的移植和维护。

Java 7 对 RowSet 接口的实现对象的创建做了更新，采用了 Java 标准的服务提供者接口（Service Provider Interface，SPI）机制。使用者通过工厂方法来创建具体的 RowSet 接口的实现对象，而工厂对象本身由 RowSet 实现的提供者注册到 Java 平台上。具体的工厂对象的查找和创建是由 javax.sql.rowset.RowSetProvider 类的静态方法来完成的，具体的工厂对象则实现 javax.sql.rowset.RowSetFactory 接口。RowSetFactory 接口提供了 5 种不同 RowSet 接口实现的创建方法。代码清单 6-4 给出了使用 javax.sql.rowset.JdbcRowSet 实现的示例，从中可以看到 RowSet 接口的基本用法，数据库连接字符串和 SQL 语句都是以属性的方式来进行设置的。

代码清单 6-4　使用工厂方法创建 RowSet 接口的实现对象的示例

```
public void useRowSet() throws SQLException {
    RowSetFactory rsFactory = RowSetProvider.newFactory();
    try (JdbcRowSet jrs = rsFactory.createJdbcRowSet())
        { jrs.setUrl("jdbc:derby://localhost/java7book");
        jrs.setCommand("SELECT * FROM book");
        jrs.execute();
        jrs.absolute(1);
        jrs.updateString("name", "New book");
        jrs.updateRow();
    }
}
```

除了上面介绍的几个比较重要的更新之外，JDBC 4.1 还有一些比较小的更新。这些

小更新包括：在使用 ResultSet 中的 getObject 方法时，可以直接把结果类型的 Java 类作为参数传递进去，而不需要在调用之后再进行强制类型转换。比如，之前的调用方式是“(String) rs.getObject(1)”，新的调用方式是“rs.getObject(1, String.class)”；表示数据库中元数据的 java.sql.DatabaseMetaData 接口中新增了 getPseudoColumns 方法来获取数据库表中包含的伪列（pseudo column）和隐藏列（hidden column）的元数据。伪列指的是不在数据库表结构中定义但可以在查询中获取的列。不同数据库实现所支持的伪列是不同的，比如 Oracle 数据库允许在查询中使用伪列“ROWNUM”来获取当前行的行号，类似的伪列还有获取当前日期和时间戳的“SYSDATE”和“SYSTIMESTAMP”。隐藏列通常用来保存一些与数据库表相关的内部信息。getPseudoColumns 方法的返回值是 ResultSet 接口的实现对象，其中的每一行都包含了查找到的伪列和隐藏列的名称、数据类型、大小等信息。如果通过 Statement 接口的 setQueryTimeout 方法设置了数据库查询的超时等待时间，那么当操作超时的时候，会抛出更加具体的 java.sql.SQLTimeoutException 异常，而不是通用的 SQLException 异常。

在使用 JDBC 4.1 时，需要检测所增加的新特性是否已经被关系数据库实现支持。不同的关系数据库实现在对 JDBC 规范的支持上可能有所不同。在使用 JDBC 4.1 的新特性之前，先查看一下所使用的数据库和驱动的相关信息，以确定对 JDBC 4.1 规范的支持程度。

6.2 java.lang 包的更新

Java 7 对 java.lang 包的更新也比较多，主要涉及基本类型的包装类、进程使用和线程类。下面分别具体介绍。

6.2.1 基本类型的包装类

Java 7 对基本类型的包装类做了一些更新，以更好地满足日常的开发需求。第一个更新是在基本类型的比较方面，Boolean、Byte、Short、Integer、Long 和 Character 类都添加了一个比较两个基本类型值的静态 compare 方法，比如 Long 类的 compare 方法可以用来比较两个 long 类型的值。这个 compare 方法更多是作为“语法糖衣”而存在的，可以简化进行基本类型数值比较时的代码。在 Java 7 之前，如果需要对两个 int 数值 x 和 y 进行比较，一般的做法是使用代码“Integer.value(x).compareTo(Integer.value(y))”，而在 Java 7 中可以直接使用“Integer.compare(x, y)”。

对于字符串内部化（string interning）技术，开发人员可能并不陌生，采用这种技术是常见的优化策略，可以提高字符串比较时的性能，是一种典型的空间换时间的做法。Java 也采用了这种技术。在 Java 中，包含相同字符的字符串字面量引用的是相同的内部对象。String 类也提供了 intern 方法来返回与当前字符串内容相同但已经被包含在内部缓存中的对象引用。在对被内部缓存的字符串进行比较时，可以直接使用“==”操作

符，而不需要用更加耗时的 equals 方法。比如，在代码清单 6-5 中，第一个字符串比较的结果为 true，第二个字符串比较的结果为 false，第三个字符串比较的结果为 true。第二个字符串比较如果使用 equals 方法进行，那么结果也是 true。

代码清单 6-5　字符串内部化的示例

```
public void stringIntern() {
    boolean value1 = "Hello" == "Hello";
    boolean value2 = (new String("Hello") == "Hello");
    boolean value3 = (new String("Hello").intern() == "Hello");
}
```

Java 7 把这种内部化机制扩大到了 –128 到 127 之间的数字。根据 Java 语言规范，对于 –128 到 127 范围内的 short 类型和 int 类型，以及 \u0000 到 \u007f 范围内的 char 类型，它们对应的包装类对象始终指向相同的对象，即通过“==”进行判断时的结果为 true。为了满足这个要求，Byte、Short、Integer 类的 valueOf 方法对于 –128 到 127 范围内的值，以及 Character 类的 valueOf 方法对于 0 到 127 范围内的值，都会返回内部缓存的对象。如代码清单 6-6 所示，在使用了内部缓存的情况下，由于第一个比较操作的数值是在 –128 到 127 之间，因此 Integer 类的 valueOf 方法返回的是同一个缓存对象，value1 的值为 true ；而第二个比较操作的数值超出了默认的缓存范围，因此 valueOf 方法返回的是两个不同的对象，value2 的值为 false。

代码清单 6-6　Integer 类的内部化的示例

```
public void numberCache() {
    boolean value1 = Integer.valueOf(3) == Integer.valueOf(3);
    boolean value2 = Integer.valueOf(129) == Integer.valueOf(129);
}
```

如果希望缓存更多的值，可以通过 Java 虚拟机启动参数“java.lang.Integer.IntegerCache.high”来进行设置。例如，使用“-Djava.lang.Integer.IntegerCache.high=256”之后，数值缓存的范围就变成了 –128 到 256，再次运行代码清单 6-6 会发现，value2 的值变成 true，因为 129 处于修改之后的缓存范围之内。

6.2.2　进程使用

在 Java 程序中可能需要调用底层操作系统上的其他程序，Java 标准 API 提供了创建底层操作系统上运行的进程的能力，只需要传入正确的命令和相关的参数，就可以启动一个进程。在进程启动之后，可以从 Java 程序向进程提供输入数据，以及读取进程运行过程中产生的输出数据。对于在 Java 程序中启动其他进程这个任务来说，最重要的是输入和输出的处理。通常的做法是把 Java 程序的内部运行结果作为输入传递给一个新创建的进程，然后等待进程执行完成。在得到进程输出的运行结果之后，再继续下面的处

理。通过这种方式，底层操作系统上的其他进程可以很好地与 Java 程序集成起来。

在 Java 7 之前，对进程的输入和输出进行处理的方式比较有限，只支持管道式的方式。进程的输入对 Java 程序来说是一个输出流，程序向这个输出流中写入的数据会通过管道传递给进程。同样的，进程的输出对于 Java 程序来说是一个输入流，通过读取此输入流的内容获得进程的输出。标准的创建新进程的过程是使用 java.lang.ProcessBuilder 类来设置新进程的属性，然后通过 start 方法来启动进程的执行。ProcessBuilder 类的 start 方法的返回值是一个表示进程的 java.lang.Process 类的对象。通过 Process 类的 getOutputStream 方法可以得到向进程写入数据的输出流，而通过 getInputStream 和 getErrorStream 方法可以分别得到包含进程正常执行和出错时输出内容的输入流。代码清单 6-7 给出了创建进程的示例。示例中启动了 Windows 上的命令行工具来执行“netstat -a”命令，并把结果保存到一个文件中。

代码清单 6-7　创建进程的示例

```
public void startProcessNormal() throws IOException
    { ProcessBuilder pb =
        new ProcessBuilder("cmd.exe", "/c", "netstat", "-a");
    Process process = pb.start();
    InputStream input = process.getInputStream();
    Files.copy(input, Paths.get("netstat.txt"), StandardCopyOption.REPLACE_
        EXISTING);
}
```

使用管道的方式在某些情况下显得不够灵活，因此 Java 7 对进程的输入和输出处理进行了更新，增加了另外的两种处理方式。第一种是继承式，即新创建进程的输入和输出与当前的 Java 进程相同。第二种是基于文件式，即把文件作为进程输入的来源和输出的目的地。代码清单 6-8 给出了继承式的一个示例，其中启动的进程通过 Windows 上的命令行工具来执行 dir 命令，通过 ProcessBuilder 类的 redirectOutput 方法把进程的输出设置为继承自父进程，运行的结果就是 dir 命令的输出内容，会显示在 Java 程序默认的输出控制台中。

代码清单 6-8　进程的输入和输出的继承式处理方式的示例

```
public void dir() throws IOException
    { ProcessBuilder pb =
        new ProcessBuilder("cmd.exe", "/c", "dir");
    pb.redirectOutput(Redirect.INHERIT);
    pb.start();
}
```

如果希望把进程的输入或输出改为文件，那么可以使用 ProcessBuilder 类中的 redirectInput 和 redirectOut 方法的其他重载形式。代码清单 6-9 给出了基于文件式处理方式的示例。示例中通过一个文件来保存进程的输出内容，只需要把一个 java.io.File 类

的对象作为 redirectOutput 方法的参数即可。这种做法在实现上相当于通过管道方式读取输入流来获取进程的输出，然后将其写入一个文件中。不过以标准 API 的方式给出的实现，显然比程序自己来实现要好。

代码清单 6-9　进程的输入和输出的文件式处理方式的示例

```
public void listProcesses() throws IOException
    { ProcessBuilder pb =
        new ProcessBuilder("wmic", "process");
    File output = Paths.get("tasks.txt").toFile();
    pb.redirectOutput(output);
    pb.start();
}
```

从 API 的角度来说，Java 7 通过新增的 ProcessBuilder.Redirect 类对进程的输入和输出重定向方式进行了统一。ProcessBuilder.Redirect 类提供了两种直接使用的重定向类型，一种是 Java 7 之前就有的管道式，用 ProcessBuilder.Redirect.PIPE 来表示；另一种是前面介绍的继承式，用 ProcessBuilder.Redirect.INHERIT 来表示。其余 3 种方式都是与文件相关的，在使用时都需要一个 File 类的对象作为参数。ProcessBuilder.Redirect.from 表示从一个文件中读取内容作为输入，ProcessBuilder.Redirect.to 表示把输出写入一个文件中，ProcessBuilder.Redirect.appendTo 表示把输出的内容添加到一个已有的文件中。

在通过 ProcessBuilder 类的 redirectInput 和 redirectOutput 方法将输入和输出重定向之后，如果新的输入源和输出目标不是默认的管道方式，那么就无法访问所创建进程的 Process 类的对象中的对应流。例如，通过 redirectInput 方法把进程的输入重定向到某个文件之后，Process 类的对象的 getOutputStream 方法返回的是一个空的输出流，调用该输出流的 write 方法总是会抛出 IOException 异常；通过 redirectOutput 和 redirectError 方法把进程的正常和错误的输出重定向到某个文件之后，Process 类的对象的 getInputStream 和 getErrorStream 方法返回的是一个空的输入流，调用该输入流的 read 方法总是会返回 –1。因此，如果在 ProcessBuilder 类的对象中对进程的输入或输出进行了重定向，那么相应的 Process 类的对象的使用者一定要了解这一点，以免造成使用错误。

6.2.3　Thread 类的更新

Java 7 对于表示线程的类 java.lang.Thread 也做了一些更新，这些更新主要明确了 Thread 类的对象在某些情况下的行为，并且去掉了之前使用中比较模糊的和设计不合理的部分。首先将 Thread 类的 clone 方法改为总是抛出 CloneNotSupportedException 异常，这是因为对一个 Thread 类的对象进行克隆是没有意义的。Java 7 显式地禁止了对 Thread 类对象的克隆操作。其次，在 Java 7 之前，Thread 类的 join 方法和 sleep 方法可以接收一个 long 类型的参数表示等待的时间，但是并没有定义当这个参数值为负数

时的处理方式。Java 7 中规定：如果这两个方法的等待时间参数的值为负数，则会抛出 IllegalArgumentException 异常。

另外一个明确了参数处理行为的是 Thread 类的构造方法。在创建 Thread 类的对象时可以使用的参数包括：表示 Thread 类的对象所在线程组的 java.lang.ThreadGroup 类的对象，表示需要运行的任务的 java.lang.Runnable 接口的实现对象，以及表示线程名称的 String 类的对象。如果传入的 ThreadGroup 类的对象为 null，那么会先尝试调用当前配置好的安全管理器（java.lang.SecurityManager 类的对象）的 getThreadGroup 方法来获取 ThreadGroup 类的对象；如果没有配置安全管理器或 getThreadGroup 方法也返回 null，那么会使用当前线程所在线程组的 ThreadGroup 类的对象；如果传入的 Runnable 接口的实现对象为 null，那么会调用 Thread 类的对象本身的 run 方法；如果传入的线程名称是 null，会抛出 NullPointerException 异常。

在调用 Thread 类的 setContextClassLoader 方法来设置线程上下文类加载器时，如果传入的参数为 null，则表明使用的是系统类加载器。如果无法使用系统类加载器，就使用启动类加载器。同样的，如果当前线程上下文类加载器是系统类加载器或启动类加载器，那么 getContextClassLoader 方法的返回值是 null。关于类加载器的更多介绍，请见第 9 章。

6.3 Java 实用工具类

Java 中包含实用工具类的包 java.util 在日常开发中使用得非常频繁，Java 7 对这个包中的不少内容进行了更新。

6.3.1 对象操作

在 java.util 包中新增了一个用来操作对象的工具类 java.util.Objects。Objects 类中包含的都是静态方法，通过这些方法可以快速对对象进行操作。

在进行两个对象的比较操作时，可以使用 Objects 类的 compare 方法。一般来说，进行对象比较是先由 Java 类实现 java.lang.Comparable 接口，再通过 compareTo 方法来进行比较。如果对集合中的元素进行排序，那么还会用到 java.util.Comparator 接口的实现。Objects 类中的 compare 方法可以将两个对象通过特定的 Comparator 接口的实现对象来进行比较。代码清单 6-10 中给出了一个简单的对 Long 对象进行比较的 Comparator 接口的实现，以及 Objects 类的 compare 方法的使用示例。

代码清单 6-10　Objects 类的 compare 方法的使用示例

```
private static class ReverseComparator implements Comparator<Long> {
    public int compare(Long num1, Long num2) {
        return num2.compareTo(num1);
    }
}
```

```
public void compare() {
    int value1 = Objects.compare(1L, 2L, new ReverseComparator());
}
```

判断对象相等的方式一般是调用 Object 类的 equals 方法，如判断两个对象 a 和 b 是否相等，可以使用代码“a.equals(b)”。Objects 类的 equals 方法可以直接判断两个对象是否相等，如“Objects.equals(a, b)”。此方法的一个好处是会对 null 值进行处理。如果直接调用一个对象的 equals 方法，需要先判断这个对象是否为 null，而使用 Objects 类的 equals 方法则不需要。如果 Objects 类的 equals 方法调用时的两个参数的值都是 null，则判断结果是 true；而如果只有一个参数为 null，则判断结果是 false；如果两个参数都不为 null，则调用第一个参数的 equals 方法来进行判断。Objects 类中与 equals 方法作用相似的是 deepEquals 方法，利用该方法也可以对两个对象进行相等性判断。与 equals 方法不同的是，如果 deepEquals 方法的两个参数都是数组，则会调用 java.util.Arrays 类的 deepEquals 来进行比较。Arrays 类的 deepEquals 方法在进行数组比较时，会考虑数组中的所有元素的相等性。在其他情况下，deepEquals 方法和 equals 方法的作用是相同的。代码清单 6-11 给出了示例，其中 value1 和 value2 的值都为 false。

代码清单 6-11　Objects 类的 equals 方法的使用示例

```
public void equals() {
    boolean value1 = Objects.equals(new Object(), new Object());
    Object[] array1 = new Object[] {"Hello", 1, 1.0};
    Object[] array2 = new Object[] {"Hello", 1, 1.5};
    boolean value2 = Objects.deepEquals(array1, array2);
}
```

Objects 类中的 hashCode 方法可以用来获取对象的哈希值。如果参数为 null，那么返回值是 0；否则返回值是参数对象的 hashCode 方法的返回结果。如果需要计算一组对象的哈希值，那么可以使用 Objects 类的 hash 方法。Objects 类的 hash 方法的实现使用的是 Arrays 类中 hashCode 方法。需要注意的是，在调用 hash 方法时传入单个对象作为参数的返回结果，与使用同样的参数调用 hashCode 方法的结果并不相同。如代码清单 6-12 所示，hashCode1 和 hashCode3 的值是不相同的，Objects 类的 hash 方法有自己的计算方式，不同于 hashCode 方法。

代码清单 6-12　Objects 类的 hash 和 hashCode 方法的使用示例

```
public void hash() {
    int hashCode1 = Objects.hashCode("Hello");
    int hashCode2 = Objects.hash("Hello", "World");
    int hashCode3 = Objects.hash("Hello");
}
```

Objects 中还有一组用于获取对象的字符串表示的 toString 方法的不同重载形式。Objects 的 toString 方法在参数为 null 时返回值是“null”，而在其他情况下相当于调用参数对象的 toString 方法。如果希望在参数为 null 时返回给定的内容作为提示信息，那么可以使用 toString 方法的另外一个重载形式，即通过一个额外的参数来指定参数值为 null 时的返回结果。如代码清单 6-13 所示，str1 和 str2 的值分别是“Hello”和“空对象”。

代码清单 6-13　Objects 类的 toString 方法的使用示例

```
public void useToString() {
    String str1 = Objects.toString("Hello");
    String str2 = Objects.toString(null, "空对象");
}
```

6.3.2　正则表达式

在日常开发中处理文本内容时经常会用到正则表达式。通过正则表达式可以简洁地解决一些常见的问题。Java 7 对 java.util.regex 包中的内容进行了更新，主要包括以下几个方面。

1. 支持命名捕获分组

捕获分组（capturing group）在使用正则表达式从文本中提取满足某种模式的部分字符时非常有用。通过把感兴趣的字符串的模式封装在捕获分组中，可以在匹配之后很容易地获取这些内容。另外捕获分组也可以在正则表达式中以后向引用（back reference）的方式来直接使用，以表示相同的模式。在 Java 7 之前，对捕获分组的引用只支持使用表示出现顺序的数字形式。这体现在 java.util.Matcher 类的 group 方法只接受 int 类型作为参数，后向引用的语法也仅支持类似“\1”这样的形式。如果一个正则表达式中包含很多捕获分组，那么开发人员需要清楚每个数字所代表的捕获分组的含义，这对于代码的编写者和阅读者来说都是一件很麻烦的事情。Java 7 引入的命名捕获分组可以很好地解决这个问题。通过为每个捕获分组添加一个有意义的名字，使开发人员可以很容易地明白每个分组所表示的含义，这比使用无意义的数字要方便得多。

代码清单 6-14 给出了通过命名捕获分组来匹配字符串并提取内容时的用法。待匹配的字符串是一个 URL，其中通过路径的不同部分来表示查询参数的名称和值。这种采用路径而不是查询字符串来指明参数的方式，在目前的 Web 开发中比较常见。在正则表达式的模式中，为提取每个参数内容的捕获分组都指定了一个有意义的名字。当匹配完成之后，可以通过 Matcher 类的 group 方法来获取每个捕获分组的内容，参数是在模式中指定的名字。采用“?<>”的格式为一个捕获分组命名，“<>”中的内容是名称。名称必须由大小写英文字母和数字组成，同时第一个字符必须是字母。

代码清单 6-14　通过命名捕获分组来匹配字符串并提取内容的示例

```
public void namedCapturingGroup() {
    String url = "http://www.example.org/uid/alex/docid/1/title/MyFirstBlog";
    Pattern pattern = Pattern.compile("^.*/uid/(?<uid>.*)/docid/(?<docid>.*)/
        title/(?<title>.*)");
    Matcher matcher = pattern.matcher(url);
    if (matcher.matches()) {
        String uid = matcher.group("uid"); // 值为alex
        String docId = matcher.group("docid"); // 值为1
        String title = matcher.group("title"); // 值为MyFirstBlog
    }
}
```

捕获分组的名称也可以用在正则表达式之中，用来替换使用数字来进行后向引用的做法。代码清单 6-15 给出了一个示例，用正则表达式来查找重复出现的数字。在正则表达式中引用命名捕获分组使用的语法是“\k<>”,“<>”中是之前定义的捕获分组的名称。

代码清单 6-15　捕获分组的名称作为后向引用的示例

```
public void repeatPattern() {
    String str = "123-123-12-456-456";
    Pattern pattern = Pattern.compile("(?<num>\\d+)-\\k<num>");
    Matcher matcher = pattern.matcher(str);
    while (matcher.find()) {
        String repeat = matcher.group("num");
    }
}
```

2. 使用“\x”表示 Unicode 中的代码点

在正则表达式中，要引用 Unicode 字符时，可以通过“\u×××”的形式，如“\u0030”表示数字“0”。通过这种方式可以表示某些无法在源代码中直接出现的字符，比如无法输入汉字的开发人员可以通过这种方式来表示中文字符。第 4 章介绍 Unicode 时提到过，如果一个 Unicode 字符不在基本多语言平面（BMP）中，那么要在 Java 中以代理项对的方式出现，在转换成正则表达式中使用“\u”形式的时候，则需要两个相邻的字符。这样既不够直观，使用起来也不方便。Java 7 对正则表达式新增了“\x”来直接表示 Unicode 中的代码点。“\x”的使用方式与“\u”类似，只不过允许表示的范围更广，如 Unicode 代码点 U+1011F 可以直接表示成“\x1011F”。

3. 新标记 Pattern.UNICODE_CHARACTER_CLASS

在通过 Pattern 类的 compile 方法对正则表达式进行编译时，可以指定多个标记。这些标记可以控制匹配时的行为。常见的标记包括设置匹配时不区分大小写的 Pattern.CASE_INSENSITIVE，以及设置“.”匹配包括换行符在内的所有字符的 Pattern.DOTALL。Java 7 中添加了一个额外的标记 Pattern.UNICODE_CHARACTER_CLASS 来

设置使用 Unicode 版本的预定义字符类和 POSIX 字符类。以“\d”这个预定义字符类为例，在默认情况下，只会匹配 0 到 9 的字符，如果启用了 UNICODE_CHARACTER_CLASS 标记，“\d”会匹配 Unicode 规范中所定义的所有属于数字类别的字符，不只是 0 到 9 的数字，也会包括其他语言中的数字字符。代码清单 6-16 给出了一个示例，使用“(\d+)”模式来匹配字符串中的数字。示例中输入的字符串是“100 １００”，其中包含了一般的数字“100”和全角数字“１００”。在两个 Pattern 类的对象中，第一个 Pattern 类的对象在编译时没有使用 UNICODE_CHARACTER_CLASS 标记，只能匹配一般的数字“100”，而第二个 Pattern 类对象可以匹配整个字符串。

代码清单 6-16　UNICODE_CHARACTER_CLASS 标记的使用示例

```
public void useUnicodeCharacterClass()
    { String str = "100 １００";
    Pattern pattern = Pattern.compile("(\\d+)");
    Matcher matcher = pattern.matcher(str);
    if (matcher.find()) {
        String digit = matcher.group(1); // 值为 100
    }
    pattern = Pattern.compile("(\\d+)", Pattern.UNICODE_CHARACTER_CLASS);
    matcher = pattern.matcher(str);
    if (matcher.find()) {
        String digit = matcher.group(1); // 值为 100 １００
    }
}
```

受到 UNICODE_CHARACTER_CLASS 标记影响的字符类除了“\d”之外，还包括“\s”、“\w”、“\p{Lower}”、“\p{Upper}”和“\p{Punct}”等。

4. 指定 Unicode 字符使用的书写格式

Java 7 中另外一个与正则表达式相关的更新也与 Unicode 相关。在 Java 7 之前，正则表达式在匹配 Unicode 字符串时允许指定 Unicode 字符所在的区块和类别，而在 Java 7 中还允许指定 Unicode 字符使用的书写格式（script）。在指定书写格式时使用的是“\p”，如代码清单 6-17 所示。“\p{script=Han}”的含义是匹配字符串中书写格式为汉字的 Unicode 字符，因此正则表达式的执行结果是原始字符串中的汉字“你好”。

代码清单 6-17　指定 Unicode 字符使用的书写格式的示例

```
public void matchScript() {
    String str = "abc 你好 123";
    Pattern pattern = Pattern.compile("(\\p{script=Han}+)");
    Matcher matcher = pattern.matcher(str);
    if (matcher.find()) {
        String hans = matcher.group(1); // 值为“你好”
    }
}
```

在匹配时可以使用的合法书写格式名称都定义在枚举类型 Character.UnicodeScript 中。

6.3.3 压缩文件处理

Java 标准库中的 java.util.zip 包用于对压缩文件进行处理。Java 7 对这一部分功能的更新也比较多。在对文件进行压缩时，允许选择压缩时缓存的中间结果的输出方式，这体现在 java.util.zip.Deflater 类的 deflate 方法增加了一个参数来表示不同的输出方式。默认的方式是 Deflater.NO_FLUSH，该方式由压缩者来确定输出缓存的中间结果的具体时机。在这种方式下，压缩者可以自由决定缓存中数据的大小，因此通常可以获得最佳的性能。第二种输出方式是 Deflater.SYNC_FLUSH，这种方式在每次调用 deflate 方法时自动清空内部缓冲区，把压缩的中间结果输出。这种方式的好处在于，如果有解压缩程序正在等待压缩之后的输出结果，及时地清空缓冲区可以让解压缩程序更早地开始工作，有利于提高解压缩程序的工作效率。不过这种方式会对压缩性能产生影响。最后一种输出方式是 Deflater.FULL_FLUSH，这种方式除了清空缓冲区之外，同时还重置压缩者的内部状态。如果解压缩的程序发现压缩的结果不正确，可以使用此方式来调用 deflate 方法，要求压缩者重新进行压缩操作。这种方式对性能的影响最大，只在必要时才使用。

与 Deflater 类对应的 java.util.zip.DeflaterOutputStream 类的对象在创建时增加了一个参数 syncFlush，用来表示对 Deflater 类的对象所缓存的中间结果的处理方式。如果 syncFlush 的值为 true，那么在调用 DeflaterOutputStream 类的对象的 flush 方法来清空其本身的内部缓冲区之前，会先按照 SYNC_FLUSH 的方式清空对应的 Deflater 类的对象所缓存的中间结果。

在 Java 7 之前，压缩文件中的文件名和注释都使用默认编码格式 UTF-8。这种 UTF-8 格式可能造成通过 Java 压缩的文件无法被其他工具打开。为了解决这个问题，Java 7 允许在创建压缩文件时显式地指定文件名和注释所用的字符集。这体现在 java.util.zip.ZipFile、java.util.zip.ZipInputStream 和 java.util.zip.ZipOutputStream 类的构造方法中都增加了一个 java.nio.charset.Charset 类的对象作为参数。这个 Charset 类的对象就表明了压缩文件中文件名和注释所用的编码字符集。通过 Java 产生的压缩文件中也包含了相关的元数据，用来表示压缩时的文件名和注释所使用的字符集。

除了上面这些比较大的改动之外，还有一些相关的小改动，包括：Java 7 支持大于 4 GB 的压缩文件的处理；ZipFile 类实现了 java.io.Closeable 接口，从而可以在 try-with-resources 语句中使用；ZipFile 类多了一个方法 getComment，用来获取在创建压缩文件时通过 ZipOutputStream 类的 setComment 方法所添加的文件注释；如果 java.util.zip.GZIPInputStream 类在处理压缩文件时遇到了格式不正确的压缩文件，会抛出更加具体的 java.util.zip.ZipException 异常。

在集合类方面，java.util.Collections 类增加了两个新的方法 emptyIterator 和 emtyEnumeration，用来返回空的迭代器对象和列举对象。

6.4 JavaBeans 组件

JavaBeans 是 Java 平台上的组件模型。要在 Java 平台上创建可复用的组件，应该遵循 JavaBeans 的规范。对于 JavaBeans 组件，开发人员比较熟悉的是 Java EE 中的 EJB（Enterprise JavaBeans），以及 Java 类中遵循 JavaBeans 命名规范的属性获取和设置的方法。JavaBeans 的强大之处在于以规范的组件模型作为基础，可以通过工具很方便地进行单个组件的自定义和多个组件的组装等操作。对于所有遵循 JavaBeans 规范的组件，都可以通过工具以统一的方式来进行操作。

符合 JavaBeans 规范的每个组件都包含 3 类信息，分别是属性、方法和事件。属性指的是一个组件暴露出来的外观或行为上的特征。可以通过改变属性的值来定制组件的外观或行为。JavaBeans 组件的方法与一般的 Java 方法并没有区别，可以在其他组件中调用。事件是组件之间进行交互的方式。某个组件可以发布事件，而另外的组件可以在这个事件上注册监听器。

6.4.1 获取组件信息

一个 JavaBeans 组件可以通过 java.beans.Introspector 类来获取组件中的属性、方法和事件的信息，使用的是 Introspector 类中的静态方法 getBeanInfo。该方法的返回值是包含了组件相关信息的 java.beans.BeanInfo 接口的实现对象。获取组件信息的方式可能有两种，一种是开发人员自己提供的 BeanInfo 接口的实现类，另外一种是由系统通过反射 API 来自动发现组件中的信息。对于第一种方式，系统会根据固定的名称模式来查找组件对应的 BeanInfo 接口的实现类。比如，对于类名为“com.java7book.My”的组件，查找类名为“com.java7book.MyBeanInfo”的 BeanInfo 接口的实现类作为组件的信息来源。如果没有找到相关实现类，会使用第二种方式，即通过反射 API 来发现相关信息。这两种方式的一个重要区别在于，在找到 BeanInfo 接口的实现类之后，不会再继续查找该组件的父类来获取信息；而通过反射 API 的方式则会沿着继承层次结构树一直向上查找父类中的相关信息。

对于具体的组件信息的获取过程，getBeanInfo 方法也提供了不同的重载方式供开发人员进行配置。可以配置的内容主要有两个，一个是在获取过程中包含哪些类中的信息。前面提到过，组件的父类中的信息有可能被包含进来，如果不希望包含组件的某些父类中的信息，那么可以指明终止组件信息获取过程的类名。当沿着继承层次结构树向上获取时，如果遇到指定的终止类，就停止继续获取。另外一个可配置的内容是对找到的 BeanInfo 接口实现类的处理方式。在 getBeanInfo 方法中允许设置 3 种不同的处理方式，对应的参数值分别是使用所有 BeanInfo 接口实现类的 USE_ALL_BEANINFO、忽略组件类对应的 BeanInfo 接口实现类的 IGNORE_IMMEDIATE_BEANINFO 和忽略包括组件类的父类在内的所有 BeanInfo 接口实现类的 IGNORE_ALL_BEANINFO。

通过这两种配置方式，可以很好地控制组件信息的获取过程。不过在 Java 7 之前，

这两种配置方式不能同时使用，只能使用其中一种。Java 7 添加了额外的 getBeanInfo 的重载方式。代码清单 6-18 给出了 getBeanInfo 方法的使用示例。MyBean 是要获取信息的组件类，ParentBean 是 MyBean 的父类。getBeanInfo 方法调用表明获取信息时不考虑 ParentBean 类的信息，同时忽略所有的 BeanInfo 接口的实现类。

代码清单 6-18　getBeanInfo 方法的使用示例

```
public void introspect() throws IntrospectionException {
    BeanInfo beanInfo = Introspector.getBeanInfo(MyBean.class, ParentBean.class,
        Introspector.IGNORE_ALL_BEANINFO);
    outputBeanInfo(beanInfo);
}
```

6.4.2　执行语句和表达式

JavaBeans 组件也提供了动态执行语句和表达式的能力，主要是为了方便工具的使用者以类似脚本语言的方式来对组件进行操作，比如调用一个组件对象 myBean 的 open 方法的语句可以直接写成“myBean.open()”。语句和表达式的区别在于，语句不关心具体的执行结果，而表达式则会把执行结果记录下来。语句和表达式分别用 java.beans.Statement 和 java.beans.Expression 类来表示。Expression 类继承自 Statement 类，并添加了获取和设置执行结果的方法。Statement 类的 execute 方法用来执行语句。在 Java 7 中，Expression 类增加了缓存执行结果的功能。当通过 Expression 类的 getValue 来获取执行结果时，如果 execute 方法之前没有被调用过，则会先调用 execute 方法，再返回结果，同时也会把执行结果记录下来。代码清单 6-19 给出了 Expression 类的使用示例。在创建 Expression 类的对象时，需要提供目标对象、方法名称和方法的调用参数。如果再次调用 getValue 方法，只会得到缓存中的执行结果，execute 方法不会被再次调用。

代码清单 6-19　Expression 类的使用示例

```
public void executeExpression() throws Exception {
    Expression expr = new Expression(new MyObject(), "greet", new Object[]
        {"alex"});
    expr.execute();
    Object result = expr.getValue();
}
```

6.4.3　持久化

JavaBeans 组件在其属性发生变化之后，可以被持久化，以保存组件的内部状态。之后在需要时可以把保存的内容再次读取出来，并恢复组件的内部状态。JavaBeans 组件的持久化依赖的是 Java 标准的对象序列化机制。一般来说，可以把组件的内部状态以流的二进制形式保存，或者保存成 XML 文件。以流的形式进行持久化时使用的是 Java

中的 java.io.ObjectInputStream 和 java.io.ObjectOutputStream 类，而在以 XML 文件作为持久化形式时，使用的是 java.beans.XMLEncoder 和 java.beans.XMLDecoder 类。

Java 7 对将 JavaBeans 组件保存成 XML 文档的功能做了更新，在 XMLEncoder 类中增加了构造方法，可以更加精细地控制保存时的行为。Java 7 之前的 XMLEncoder 构造方法只接受一个 java.io.OutputStream 类的对象作为参数，表示保存内容的输出流。而 Java 7 新增的构造方法中添加了额外的 3 个参数，分别是输出时使用的字符集、是否输出 XML 处理指令声明和整个 XML 文档的缩进空格数。允许指定输出时使用的字符集主要是为了满足不同的编码格式需求，而另外两个参数是为了使输出的 XML 文档内容可以被嵌入到其他 XML 文档中。代码清单 6-20 给出了 XMLEncoder 类的新构造方法的使用示例。如果 XMLEncoder 类的对象的输出要作为其他 XML 文档的一部分，第 3 个参数的值应该为 false，另外第 4 个参数应该是正确的缩进空格数，以保持整个 XML 文档的良好缩进格式。

代码清单 6-20　XMLEncoder 类的新的构造方法的使用示例

```
public void xmlEncode() throws IOException {
    OutputStream output = Files.newOutputStream(Paths.get("result.xml"),
        StandardOpenOption.CREATE_NEW);
    try (XMLEncoder encoder = new XMLEncoder(output, StandardCharsets.UTF_8.
       name(), true, 0)) {
        encoder.writeObject(new MyBean());
    }
}
```

用于读取 XMLEncoder 类的输出文档的 XMLDecoder 类也有一些更新，主要是提供了更好的对 SAX 解析方式的支持。XMLDecoder 类新增了一个接受 org.xml.sax.InputSource 类型参数的构造方法。在 Java 7 之前，创建 XMLDecoder 类的对象时，只能使用 InputStream 类的对象来表示解析时的输入数据。而 InputSource 类则提供了更加丰富的方式来表示输入数据，除了 InputStream 类的对象之外，还支持使用标识符和 java.io.Reader 类的对象。

6.5　小结

本章主要围绕 Java 7 对标准库 API 所做的更新来展开，涉及数据库操作、java.lang 包、java.util 包和 JavaBeans 组件等方面。在这些更新中，数据库操作相关的对象都可以用在 try-with-resources 语句中，有利于提高代码的简洁性，新增的 setNetworkTimeout 和 abort 方法可以解决由于网络连接问题造成的查询操作等待时间过长的问题；而在 java.lang 包中，ProcessBuilder 类新增的对进程输入和输出进行重定向的能力，方便开发人员处理进程的输入和输出；在 java.util 包中，Objects 是很实用的工具类，对正则表达式在命名捕获分组和 Unicode 支持上的更新，方便了开发人员的使用；在 JavaBeans 组

件方面则增强了组件信息的获取过程、Expression 类的使用方式和 XMLEncoder 类在输出用于内嵌的 XML 文档时的能力。

Java 7 还有其他一些小的更新，限于篇幅，不可能逐一进行介绍。总的来说，Java 7 除了添加新的功能方便开发人员之外，还更新了大量文档以澄清之前版本中容易产生误解的地方，另外还对之前比较模糊的行为添加了明确的定义，前面提到的 Thread 类的更新就是很好的例子。

第 14 章　超越 Java 7

本书的前面章节都是围绕 Java 7 展开的。Java SE 7 在 2011 年 7 月发布之后，Java SE 8 的开发正在进行中。根据 Oracle 在 JavaOne 2011 会议上公布的计划，Java SE 8 预计于 2013 年夏天发布。虽然 Java SE 8 将包含的内容大部分已经确定，但是可能会随着具体的开发过程而发生改变。相对于 Java SE 7 而言，Java SE 8 将会对 Java 平台做出革命性的修改，其中包含的修改涉及 Java 语言、Java 标准库和 Java 虚拟机。Java SE 8 的目标是支持在不同的计算环境下创建出可维护的、可伸缩的和高性能的 Java 应用。Java SE 8 主要侧重三个方面，分别是生产率、性能和模块化。在生产率方面，Java SE 8 通过在 Java 语言和标准库中的改进，提升 API 的抽象层次，减少不必要的冗余代码；在性能方面，Java SE 8 对集合类框架和相关 API 做了增强，以支持对批量数据进行自动的并行处理，通过 lambda 表达式和接口的默认方法可以很方便地使用这些增强功能；在模块化方面，Java SE 8 对 Java 平台本身提供了模块化支持，把 Java 平台划分成不同的模块和组件，这种模块化技术也可以用在应用的开发中。

Java SE 8 中包含的更新内容由 JSR 337（Java™ SE 8 Release Contents）来确定，目前已经确定的具体内容包括：增加 lambda 表达式的 JSR 335（Lambda Expressions for the Java™ Programming Language）、Java 平台模块化、对 Java 语言进行的一些小的改动、Java 类型的注解及日期与时间 API 等。除此之外，Java SE 8 的更新还包括 JavaFX 3.0、脚本语言支持 API 中新的 JavaScript 引擎以及虚拟机的整合等。本章的内容围绕 Java SE 8 的新特性展开。由于 Java SE 8 还在持续的开发中，很多内容都没有最终确定，因此本章内容以写作时（2012 年 2 月）为准。

14.1　lambda 表达式

lambda 表达式是 Java 开发人员长期以来一直期待 Java 语言增加的一个语法特性。使用 lambda 表达式可以通过简洁的代码来进行计算，减少了不必要的冗余代码。lambda 表达式主要用在两个地方：一个是操作新的集合类框架实现，另外一个是更好地使用已有的函数式接口。lambda 表达式实际上是创建函数式接口的一种简化的方式。lambda 表达式的相关内容由 JSR 335 来规范。

在接下来的内容中，先介绍可以用 lambda 表达式来实现的函数式接口，再对 lambda 表达式的语法进行介绍。通过目标类型，编译器可以推断出 lambda 表达式所实

现的接口类型，并进行必要的合法性检验。lambda 表达式中的代码与包围它的代码块在同一个词法作用域中，这会对代码中的名称解析产生影响。已有类中的方法也可以通过

方法引用的方式作为 lambda 表达式来使用。接口的默认方法则提供了一种与已有实现相兼容的方式来更新接口的定义。

14.1.1 函数式接口

Java 是一种面向对象的编程语言。在开发程序时，程序中的代码逻辑被封装在不同的类中。Java 类通常既包括用来维护内部状态的域，又包括用来执行业务逻辑和更新内部状态的方法。在有些情况下，程序所要做的只是简单地执行一段代码。比如在进行排序操作时，最核心的逻辑是判断任意两个对象之间的先后顺序。从函数的角度来说，这段代码接受两个对象作为输入，返回表示两个对象的先后顺序的值作为输出。在 Java SE 7 中，Java 语言并没有提供直接的语法结构来表示一个函数，与函数相近的结构是方法，但是方法不能脱离类而单独存在。因此，对于这样的场景，程序的一般做法是先定义一个接口，再定义接口的实现类。在需要使用方法时，先创建该实现类的对象，再调用对象中的方法。这个完整的过程比较繁琐。通过 Java 提供的匿名内部类可以简化一些操作。以 java.lang.Runnable 接口为例，当需要创建一个线程并启动该线程时，一般的做法如代码清单 14-1 所示。

代码清单 14-1　Runnable 接口的一般使用示例

```
new Thread(new Runnable() {
    public void run()
        { System.out.println("Hello
        World!");
    }
}).start();
```

从上面的代码可以看到，真正执行程序逻辑的有意义的代码其实只有 1 行，剩下的 4 行代码都是为了满足 Java 的语法要求而加上的，这就造成了一定的代码冗余。类似这样的代码在程序中会经常见到。由于相关的逻辑通常只在一个地方出现，没有必要把这些代码封装在另外一个单独的类中，因此，使用匿名内部类就成了开发人员最常见的选择。

Java 标准库中还存在其他与 Runnable 接口类似的接口，这些接口的特征是只声明了一个方法。具备这样特征的接口被称为函数式接口（functional interface）。对于函数式接口，通过匿名内部类实现的代码形式可以进一步简化。在代码清单 14-1 中，对 Runnable 接口中的 run 方法的声明是可以被去掉的。Runnable 接口中只包含一个方法，因此所包含的代码肯定是对这个唯一的方法的实现。同时对 Runnable 接口本身的声明也是可以被去掉的，可以从使用该匿名内部类的上下文环境中推断出使用的接口的具体类型。Java SE 8 中的 lambda 表达式利用了函数式接口的特点，提供了一种比匿名内部类更加简洁的方式来实现函数式接口。

Java 标准库中存在的函数式接口比较多，常见的包括 Runnable、java.util.Comparator、java.io.FileFilter、java.util.concurrent.Callable、java.lang.reflect.InvocationHandler、

java.beans.PropertyChangeListener、java.awt.event.ActionListener 和 javax.swing.event.ChangeListener 等。程序也可以定义自己的函数式接口。

14.1.2 lambda 表达式的语法

Java SE 8 中的 lambda 表达式提供了一种匿名方法的表示形式。它针对函数式接口的特征，提供了一种简洁的方式来替代匿名内部类的使用。代码清单 14-2 与代码清单 14-1 功能相同，但是代码清单 14-2 是使用 lambda 表达式来实现的代码。

代码清单 14-2　使用 lambda 表达式实现 Runnable 接口

```
new Thread(() -> {System.out.println("Hello World!");}).start();
```

可以看到，使用 lambda 表达式把代码清单 14-1 中的 5 行代码缩减成 1 行。lambda 表达式的基本形式是“(参数列表) -> 代码块”。lambda 表达式的表示形式类似于一般的方法，由形式参数和方法体两部分组成。形式参数包含在“()”中，“->”用来分隔形式参数和方法体。方法体可以是简单的表达式，也可以是复杂的代码块。复杂的代码块需要用“{}”包围起来。代码清单 14-3 给出了一些 lambda 表达式的示例。

代码清单 14-3　lambda 表达式的示例

```
(int x, int y) -> x + y
(int x) -> 100
() -> "Hello World"
(int x) -> {
    int num = 10;
    return num * x;
}
```

14.1.3 目标类型

lambda 表达式虽然在形式上类似于方法声明，但实际上是一种创建对象的方式。比如，代码清单 14-2 中的 lambda 表达式“() -> {System.out.println("Hello World!");}”实际上创建的是一个 Runnable 接口的实现对象。不过 lambda 表达式在声明时省略了所实现的函数式接口的具体类型，因此，同样的 lambda 表达式可能表示不同类型的对象。如果两个不同的函数式接口中的唯一方法的类型签名相同，那么可以用相同的 lambda 表达式来创建这两个接口的不同实现对象。lambda 表达式所表示的对象类型由 lambda 表达式所出现的上下文环境来确定，由编译器负责进行类型推断。编译器所进行的这种类型推断方式，与第 12 章中介绍的在泛型方法调用和 <> 操作符中使用的类型推断方式是相似的。

为了能够进行类型推断，lambda 表达式所出现的上下文环境中需要能够推断出一个具体类型。这个类型是 lambda 表达式的目标类型。可以推断出具体类型的上下文环

境包括变量声明、赋值操作、return 语句、数组初始化、一般方法或构造方法的参数、lambda 表达式的方法体、“?:”表达式和强制类型转换表达式等。如果从当前上下文环境中无法推断出具体的类型，那么 lambda 表达式无法出现在该上下文环境中。在确定了目标类型之后，编译器需要检查 lambda 表达式是否与目标类型兼容。兼容的条件是该目标类型是一个函数式接口，同时接口中的唯一方法的参数类型、返回值类型和抛出的受检异常都与 lambda 表达式的对应部分保持兼容。

在代码清单 14-2 中，lambda 表达式出现在 Thread 类的构造方法中，对应的上下文环境中的目标类型是 Runnable 接口。任何与 Runnable 接口中的 run 方法的参数类型、返回值类型和抛出的受检异常保持兼容的 lambda 表达式都可以使用。代码清单 14-4 给出了 lambda 表达式可以使用的其他上下文环境。

代码清单 14-4　lambda 表达式可以使用的上下文环境

```
Runnable r = () -> { System.out.println("Run"); }; // 赋值操作
public Comparator<String> createComparator() {  //return 语句
    return (String s1, String s2) -> s1.compareTo(s2);
}
runTasks(new Runnable[] {() -> {}, () -> {}}); // 方法参数
```

由于目标类型的存在，通常可以省略 lambda 表达式中形式参数的类型声明，而由编译器根据目标类型进行自动推断，这样可以进一步减少冗余代码。如果 lambda 表达式中只有一个形式参数，那么参数列表外围的“()”也可以省略。

14.1.4　词法作用域

在编写 lambda 表达式的方法体时，需要注意的是方法体中声明的变量与包含该 lambda 表达式的上下文环境之间的关系。在这方面，lambda 表达式采用了一种简单的实现方式：lambda 表达式的方法体并没有引入新的作用域。方法体中的名称与表达式外部的代码处于同一个词法作用域中。在进行解析时，相当于方法体中的名称出现在表达式外部的代码中。而 lambda 表达式的形式参数也是属于表达式外部代码的名称。

代码清单 14-5 中的代码会出现编译错误。这是因为变量 str 在 lambda 表达式外部的代码中已经被定义了，在 lambda 表达式的方法体中又被重新定义了一次。由于这两个同名变量存在于同一个词法作用域中，因此编译器不允许多次声明该变量。

代码清单 14-5　lambda 表达式方法体与外部代码的名称重复的示例

```
String str = "Hello";
new Thread(() -> {
    String str = "World";
    System.out.println(str);
}).start();
```

lambda 表达式的这个特性对 this 变量也适用。对于 lambda 表达式方法体中出现的 this 变量，其值相当于 this 出现在表达式外部的代码中时应该具有的值。这点不同于匿名内部类。由于方法体中 this 变量的这个特性，方法体中的 this 变量无法用来引用 lambda 表达式所表示的对象本身。在某些递归方式的实现中，需要在方法体的代码中引用当前 lambda 表达式的对象。代码清单 14-6 给出了一个示例，其中的 lambda 表达式所表示的对象被赋值给变量 task，并在方法体中引用了这个变量。

代码清单 14-6　在 lambda 表达式的方法体中引用表达式所表示的对象

```
boolean done = false;
List<Runnable> taskQueue = new ArrayList<>();
final Runnable task = () -> {
    if (!done) {
        taskQueue.add(task);
    }
    else {
        // 任务完成
    }
};
```

14.1.5　方法引用

lambda 表达式提供了一种简洁的方式来创建函数式接口的实现对象。Java 类中的已有方法也可以被当成 lambda 表达式来使用，只需要直接引用已有的方法即可。方法引用相当于复用了 Java 类中已有方法的方法体。方法引用的使用条件与 lambda 表达式是相同的。代码清单 14-7 给出了方法引用的示例。在调用 Arrays 类的静态方法 sort 时，第二个参数是 Comparator 接口的实现对象。代码清单 14-7 中使用了已有类 Comparators 中的静态方法 compareString 作为 Comparator 接口的实现对象的内部实现逻辑。

代码清单 14-7　方法引用的使用示例

```
// 已有方法的声明
public class Comparators {
    public static int compareString(String s1, String s2) {
        return s1.compareToIgnoreCase(s2);
    }
}

// 方法引用的使用
String[] array = new String[] {"c", "b", "a"};
Arrays.sort(array, Comparators::compareString);
System.out.println(Arrays.deepToString(array));
```

在使用方法引用时，除了可以引用静态类中的方法外，还可以引用特定对象中的实例方法、特定类型的任意对象中的实例方法及构造方法。在引用对象的实例方法时，使

用对象的引用作为方法引用时的前缀，如类似“myObj::myMethod”的形式。如果引用的是任意对象中的实例方法，那么在引用时，使用方法所在的类型名称作为前缀，如类似“MyClass:myMethod”的形式。在方法被调用时，实际的调用接收者对象会作为方法的第一个参数。例如，代码清单 14-7 中的字符串比较的逻辑可以简化成“Arrays.sort(array, String::compareToIgnoreCase);”。在 String 类的 compareToIgnoreCase 方法被实际调用时，当前的 String 类的对象会作为 compareToIgnoreCase 方法的第一个参数。引用构造方法的形式类似于引用类中的静态方法，只不过方法的名称固定为“new”。例如，“MyClass:new”用来引用类 MyClass 中的构造方法。如果一个类中定义了多个构造方法，那么方法引用被使用时的目标类型用于选择适用的构造方法。

14.1.6 接口的默认方法

引入默认方法的动机在于解决 Java 中接口所固有的更新困难的问题。在 Java 中，一旦接口被公开，对该接口进行修改是一件非常困难的事情。如果在新版本中为一个已有的公开接口添加一个新的方法，那么所有实现了该接口的已有类都会出现错误，因为这些类中没有添加这个新方法的实现。接口的默认方法提供了一种简洁的方式来解决这个问题。在接口中，除了方法的声明之外，还可以有方法对应的默认实现。如果该接口的实现类没有实现某个方法，而这个方法由接口提供了默认实现，那么该类会继承接口中的默认实现。在版本更新时，如果向接口中添加了新的方法，可以同时添加对应的默认实现。该接口的已有实现类会继承默认实现，而新的实现类则可以提供新的实现。通过这种方式，既不会对已有的接口实现类造成影响，又可以不断地进行更新。

下面通过一个示例来说明默认方法的作用。代码清单 14-8 是接口 Cursor 的已有声明，其中定义了两个方法 previous 和 next，分别用来向前和向后移动指针位置。

代码清单 14-8　Cursor 接口的已有声明

```
public interface Cursor
    { boolean previous();
    boolean next();
}
```

代码清单 14-9 是 Cursor 接口的新版本的声明，其中添加了两个新的方法 first 和 last，用来快速把指针移动到起始或末尾。在 first 和 last 方法的声明中，default 关键词之后是这两个方法的默认实现。默认实现通过调用已有的 previous 和 next 方法来完成。

代码清单 14-9　Cursor 接口的新版本的声明

```
public interface Cursor
    { boolean previous();
    boolean next();

    void first() default {
```

```
            while(previous()) {}
        }

        void last() default {
            while(next()) {}
        }
    }
```

Cursor 接口的新的实现类可以提供 first 和 last 方法更加高效的实现。

随着 lambda 表达式的引入，Java 中的集合类框架会做出相应的调整，以方便开发人员在进行集合类的相关操作时使用 lambda 表达式。这就意味着需要对集合类框架中的很多接口进行更新。由于集合类使用广泛，这样的更新势必会造成已有的代码无法运行。通过接口的默认方法，可以很好地解决这个问题。

14.2 Java 平台模块化

模块化是目前软件开发中经常会用到的设计理念。把系统划分成不同的模块之后，开发团队可以分工合作，快速高效地完成开发工作。采用模块化设计的系统也更容易复用、维护和更新。不过 Java 平台本身并没有对模块化的系统的实现提供很多的支持。一般的做法是把 Java 类组织在不同的名称空间中，多个 Java 类组织在一个 jar 包中，多个 jar 包同时出现在程序的类路径（CLASSPATH）中。由于不同 jar 包中可能存在同名的 Java 类，加上复杂的类加载器实现机制，使 Java 程序经常出现与类加载相关的问题。由于 Java 平台本身存在限制，为 Java 平台提供模块化支持的 OSGi 技术得到了广泛的流行，这也从侧面反应了开发者社区对于模块化的需求。OpenJDK 中的 Jigsaw 项目的目标在于为 Java 平台增加模块化的支持，这也是 Java SE 8 的重要组成部分。

Jigsaw 项目为 Java 语言增加了模块的概念。模块是 Java 类型的集合。每个模块有自己的名称、一个可选的版本号，以及当前模块和其他模块之间关系的描述。模块之间最重要的关系是依赖关系。一个模块可以依赖其他的模块，并利用所依赖的模块提供出的 Java 类。对于一个模块来说，模块本身的信息由名为 module-info.java 的 Java 文件来提供。代码清单 14-10 给出了 module-info.java 文件的内容示例。模块“com.my.base”的版本号是 1.0，依赖于版本号为 2.0 的“com.example”模块。通过“exports”声明了模块“com.my.base”中的“com.my.base.utils”包对依赖它的模块是可见的。通过“class”声明了模块“com.my.base”的主 Java 类是“com.my.base.MainApp”。

代码清单 14-10　module-info.java 文件的内容示例

```
module com.my.base @ 1.0
    { requires com.example @
    2.0; exports
    com.my.base.utils; class
    com.my.base.MainApp;
}
```

```
module com.my.ui @ 1.0 {
    requires com.my.base;
}
```

在模块中，除了 module-info.java 文件之外，其他的内容与一般的 Java 程序中可以包含的内容相同，可以包括 Java 源代码、资源文件、配置文件和本地代码等。在编译过程中，模块中包含的源代码会被编译，然后进行打包和发布。打包好的模块可以安装到某个模块仓库中。这个仓库中包含了所有可用的模块。在运行时，虚拟机会负责从模块仓库中加载模块，并与它所依赖的其他模块进行链接，之后就可以运行该模块了。

14.3 Java SE 8 的其他更新

除了 lambda 表达式和模块化支持之外，Java SE 8 中的其他更新内容包括：

- 把 JRockit 虚拟机中的部分特性整合到 HotSpot 虚拟机中，提供一个统一的虚拟机实现。
- 集成 JavaFX 3.0。在 Java SE 8 中会直接集成 JavaFX 3.0。
- 在虚拟机上可以直接使用的新的 JavaScript 引擎，以及更好的 JavaScript 和 Java 代码之间的互操作性。新的 JavaScript 引擎被称为 Nashorn，是一个基于 JSR 292 的实现。
- 在移动设备上，增加对多点触控、摄像头、地理位置信息、罗盘和重力加速器的支持。
- 对 Java 安全、网络、国际化和可访问性 API 的更新。
- 允许 Java 中的注解出现在类型的任意使用方式上，而不仅限于类型、方法、域和变量的声明中。注解可以使用的位置包括方法调用的接收者、泛型类型参数、数组、强制类型转换、instanceof 操作符、对象创建、泛型中类型参数的上界或下界、类继承关系和 throws 子句。具体的细节由 JSR 308（Annotations on Java Types）规范来描述。
- 新的日期和时间 API，用来解决当前使用 java.util.Date 类和 java.util.Calendar 类处理日期和时间中产生的问题。具体的细节由 JSR 310（Date and Time API）规范来描述。

对于上述更新，由于相关的内容正在开发中，还没有最终确定，本章不进行具体的介绍。

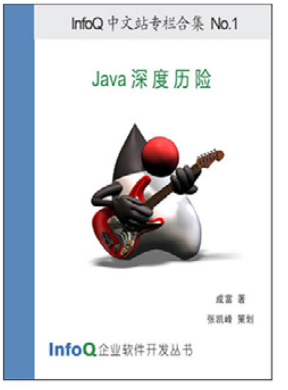
14.4 小结

在 Java SE 7 发布之后，Java SE 8 的开发正在进行中。相对于 Java SE 7 而言，Java SE 8 将是一个包含很多重要更新的版本。由于 Java SE 8 中的很多新的内容还没有最终确定。本章只对 Java SE 8 中两个最重要的更新——lambda 表达式和模块化支持进行了

介绍。希望通过这些介绍，使读者对 Java SE 8 中将要发生的变化有一定的了解。

InfoQ软件开发丛书

欢迎免费下载



商务合作：sales@cn.infoq.com　　读者反馈/内容提供：editors@cn.infoq.com