HZ BOOKS

Sun 公司核心技术丛书

Mc Graw Hill Education

深入 Java 虚拟机（原书第2版）

Inside the Java Virtual Machine, Second Edition

(美) Bill Venners 著

曹晓钢 蒋靖 译

机械工业出版社
China Machine Press

Inside the Java Virtual Machine, Second Edition

深入Java虚拟机（原书第2版）

想写出更漂亮的Java程序吗？那么打开看看精密的Java引擎是如何运转的吧！

本书深入详细地介绍了Java体系结构及其内部细节，了解这些内容才能更快速地编写更高效的程序！理解了Java虚拟机，深入细致地了解了Java技术的底层，才能使自己的程序充分发挥Java技术的优势！

本书详细解释了JVM的体系结构，包括Java栈、堆、方法区和执行引擎。还深入讨论了各种技术实现，比如解释、即时编译及自适应优化。对Java线程和监视器的行为也有精彩讲解。

本书还讨论了垃圾收集，包括引用对象、火车算法以及对象终结。最后，还讨论了错综复杂的Java安全模型，包括类型安全性、类装载器体系、类校验器、安全管理器、访问控制器和代码签名。

通过本书，读者可以充分理解Java的连接模型和动态扩展机制，学习如何编写类装载器，了解编写平台独立的Java程序的7个步骤。

本书包含下列内容：

- Java世界和JVM体系结构的完整描述
- class文件、字节码及其在类装载期间的转换和验证
- 算术、逻辑和数组操作以及流程控制
- 方法调用及返回、异常、垃圾收集和线程
- JVM的即时编译器实现

作 者 介 绍：

Bill Venners有14年编写软件专著的经验。他在硅谷的Artima软件公司提供软件咨询和培训服务。自1996年以来，他已经编写了40多篇有关Java技术的文章。他在《Java World杂志》辟有热门专栏，介绍Java内部细节、面向对象设计技术和Jini技术。他还是artima.com的作者和网站管理员，这是Java和Jini开发者的一个在线资源站点。他在全世界范围举办内部培训和公开的研究报告，有时也在软件会议上进行Java技术讲演。

适用水平：中、高级

附赠CD-ROM

ISBN 7-111-12805-2

封面设计 陈子平

HZ BOOKS 华章图书

网上购书：www.china-pub.com

北京市西城区百万庄南街1号 100037

读者服务热线：(010)68995259, 68995264

读者服务信箱：hzedu@hzbook.com

http://www.hzbook.com

ISBN 7-111-12805-2/TP · 2868

定价：58.00元（附光盘）

McGraw-Hill

Sun公司核心技术丛书

深入Java虚拟机

（原书第2版）

（美） Bill Venners 著

曹晓钢 蒋 靖 译

机械工业出版社
China Machine Press

本书作者曾因本书荣获专业技术杂志《Java Report》评选的优秀作者奖，细心的读者可以从网上找到许多对本书第1版的赞誉。

作者以易于理解的方式深入揭示了Java虚拟机的内部工作原理，深入理解这些内容，将对读者更快速地编写更高效的程序大有裨益！

本书共分20章，第1～4章解释了Java虚拟机的体系结构，包括Java栈、堆、方法区、执行引擎等；第5～20章深入描述了Java技术的内部细节，包括垃圾收集、Java安全模型、Java的连接模型和动态扩展机制、class文件、运算及流程控制等等，其中第6章和附录A～C完全可以作为class文件和指令集的参考手册。本书还附带光盘，光盘中包含用以辅助说明正文内容的交互式例示applet及示例源代码。

本书版权登记号：图字：01-2003-4928

图书在版编目（CIP）数据

深入Java虚拟机（原书第2版）/（美）文纳斯（Venners, B.）著；曹晓钢，蒋靖译. -北京：机械工业出版社，2003.9

（Sun公司核心技术丛书）

书名原文：Inside the Java Virtual Machine, Second Edition

ISBN 7-111-12805-2

Ⅰ.深… Ⅱ.①文… ②曹… ③蒋… Ⅲ.Java语言-程序设计 Ⅳ.TP312

中国版本图书馆CIP数据核字（2003）第068436号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）

责任编辑：刘立卿

北京牛山世兴印刷厂印刷 · 新华书店北京发行所发行

2003年9月第1版第1次印刷

787mm × 1092mm 1/16 · 30.25印张

印数：0 001- 4 000册

定价：58.00元（附光盘）

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线电话：（010）68326294

对本书第1版的赞誉

作者卓有成效地深入解释了Java虚拟机（JVM）的内部工作原理，对这个错综复杂的软件中的许多部分都给出了可能的实现，这是对Sun的官方规范的精彩补充。每一个概念都很清晰，一般都有例子作辅助说明。随书光盘中还包含了许多富有启发的示例，它们演示了虚拟机内部工作的情况。这本书得到虚拟机实现者的极高评价，相信任何有兴趣了解虚拟机核心部分的人都会获益匪浅。

——Antoine Trux，芬兰赫尔辛基话基亚研究中心项目经理，《Java Report》杂志，1998年12月。《深入Java虚拟机》一书的作者因本书获《Java Report》杂志1998年优秀作者奖。

在我钻研本书的结构和内容之前，我很高兴提到Venners的书给我印象最深刻的一点：对细节的全心关注和对内容的精确协调。

从第5章到第20章都包含花很多心思编写的动态交互式applet，它们为每章的主题带来了活力。比如说垃圾收集这一章，不仅介绍了许多现代垃圾收集算法，还附带了一个“鱼堆”applet，让读者真正理解垃圾收集中的设计问题及可能的解决方案。

简单地说，Venners的书是卓越的，是一本我必须推荐的书。

——Laurence Vanhelsuwé，《Java World杂志》，1998年3月

感谢你写出这么优秀的书。我已经编写Java程序很多年了，这本书真的帮助我洞察了这门语言的内脏。再次为了美妙的阅读体验感谢你！

——Noah S. Friedland 博士

最近购买了你的书，它比JVM规范易读、易懂多了！我还喜欢你的applet，它们让事情变得简明易懂。

——Paul Bathen

《深入Java虚拟机》这本书，是我所有Java图书收藏中编写得最好和最有帮助的书之一。

——Louis Barton

我刚刚读完你的《深入Java虚拟机》，感谢你富有帮助的工作！

——Antoine Trux

一本关于Java虚拟机的详尽而系统的书。假如你准备开始编写自己的JVM，或者你对“在执行.class文件的时候究竟发生了什么事情”感兴趣的话，就必须拥有这本书。对所有读过Java虚拟机规范后还想寻找更多资料的人来说，这本书是受欢迎的、减轻痛苦的良药。

——Gopal Ananthraman

我真的在阅读你的书的时候感到愉悦。它有很多很好的内容，我觉得它们会使我成为更好的Java程序员。

——Joel Nylund，美国管理系统公司

我购买了一本《深入Java虚拟机》。虽然我只阅读了第7章和第8章，但我感到非常愉快，并且对中间的细节印象深刻。你回答了我所遇到的很多问题，包括“在调用ClassLoader.findSystemClass()的时候，在动态类装载器委派责任中，对于已装载的类，虚拟机会解释哪一个类装载器？”

我以前在Lotus开发公司工作的时候，我与别人合作为Prentice-Hall写过一本叫做《深入Lotus Add-in 工具包》的书。我们讨论的技术和Java很相似——一个平台中立的、拥有部分复杂性的语言（其字节码需要一个运行时虚拟机来执行）。

作为作者，我们的目标是在描述整个技术的时候保持精确性和幽默感。我们在技术上花费了大量劳动，对精确性和技术细节特别关注——如同开发者一样，我们希望文章是有用的、正确的；如同读者一样，我们精通英语的用法，因为大部分流行的技术文章都不敢恭维。

这些方面都是我对你的工作表示激赏的。当一个作者花时间来写完整的句子，采用通俗的语气，保持专业术语的一致性，并且提供真正有价值的内容，而不仅仅是重复公开的规范（通常还是不精确的），我向这样的作者致敬。

——David McCall

如果你真的希望揭开Java的面纱，这是最好的Java书。如果你真的希望了解JVM的输入输出，《深入Java虚拟机》是一本值得敬畏的著作。我被作为技术作家的Bill先生的能力打动了，对任何认真的Java开发者，如果想深入理解Java，我强烈推荐这本书。

——Rashid Jilani，发表于AMAZON.COM

一本伟大的书。

这是我到目前为止读过的最好的Java书。Bill是一个伟大的软件工程师，也是作家。如果你希望了解JVM的内幕，这本书是必不可少的。

——Michael Young，发表于AMAZON.COM

译 者 序

计算机艺术的魅力在于其严谨性和复杂性。无论是股票交易还是人机国际象棋大战，所有的程序运行所需要的底层机器指令都只是有限的若干条。从大型的UNIX机器到桌面个人计算机，无不基于那些设计精良而优美的指令集。但是这些指令集之间互不兼容，这就使得程序的移植变得非常困难，所需时间甚至超过了重新编写一遍的时间。于是，虚拟机的概念出现了。Java虚拟机（JVM）在多个平台上实现统一语言；.NET的虚拟机（目前）在单一平台上实现多种语言。但无论如何，它们都是抽象的计算机。尽管它们都有自己的指令集，自己的内存体系。但它们却往往比实际的硬件机器简单明了。分析这样的一个虚拟机，对提高读者对底层硬件和虚拟机平台的理解大有裨益。

Java之所以得以大行其道，除了它是一门面向对象、构造精美的语言之外，更重要的原因在于：它摆脱了具体机器的束缚，使跨越不同平台编写程序成为可能。Java语言丰富的开放式类库大量使用设计模式，成功地改变了很多程序员的编程思想和习惯。但是，诸如class文件是如何被调入内存执行的、类的静态方法和静态变量的初始化是按照什么步骤进行的、对象的垃圾收集是如何以及在什么时候发生的，这些具体问题却不是每个人都清楚的。很多人对此只是有一个模糊的印象，稍微深入就难以回答。假若能够有一本书简明扼要地解释这些JVM运行的细节，而又不需要去钻研艰深的JVM规范，那该有多好！读者现在看到的这本书就可以满足这些要求。

本书作者是一位JVM领域的顶尖高手，他写的这本书却一点儿都不难懂，任何人都可以从书中领会到Java的真正精髓——无需别人告诉你，从作者娓娓道来的分析中，就可以自己得出那些高手们才能理解的结论。

本书在Java专业书籍中的地位是无可替代的，简单地说，读者不容易找到第二本像本书这样如此细致地讨论Java虚拟机的书。假如你真的想深入了解Java的精髓，理解在别的书中花费很长时间讲解的“道理”或者“难点”，那么读读这本书吧！你可能比那些所谓的高手理解得更加深刻。

本书由曹晓钢和蒋靖翻译。其中蒋靖翻译了第6章、第10～19章以及附录，曹晓钢翻译了前言、第2～5章、第7～9章和第20章，第1章由两人合译。

由于时间仓促，加上译者水平有限，书中难免有翻译不妥之处，希望广大读者和同行批评指正。

译 者

2003.5

前　言

我写本书的主要目的是向Java程序员解释Java虚拟机——包括几个和虚拟机紧密相关的核心Java API。虽然Java虚拟机使用了许多有效技术——这些技术已在Java语言之前的其他语言中被尝试和证明过，但其中采用的很多技术还没有被普遍使用。因此很多程序员在开始使用Java编程的时候，都感觉是第一次接触这些技术。垃圾收集、多线程、异常处理、动态扩展，甚至使用虚拟机本身，这些对于很多程序员来说都是全新的。本书的目的是为了帮助程序员理解这些东西的工作方式，并在这个过程中帮助他们更加适应用Java编程。

编写本书的另外一个目的是为了试验改变文本的意义。网页有三个有趣的特性，这些特性使得它们和纸质文本有所区别：它们是动态的（可以随时间变化），它们是交互式的（特别是在上面嵌入Java applet后），还有，它们是相互链接的（可以很容易地在它们之间漫游）。除了传统的文本和图表，本书还包括几个Java applet（在随书光盘给出的迷你Web站点中），用它们作为交互式例示以补充文中所述概念。除此之外，我还在Internet上维护一个Web网站 artima.com，读者可以以此为起点找到与本书主题有关的更多、更新的参考资料。本书的构成包括文本、图表、交互式例示，还有网上链接，这样做的目的是为了方便读者深入阅读。

本书介绍

本书讲述了Java虚拟机——运行所有Java程序的抽象计算机，还讲了几种与虚拟机密切相关的核心Java API。本书通过分析讲解、可运行的示例、参考资料和applet（它作为文中所述概念的交互式例示），提供了Java技术的深入概览。

Java编程语言似乎将要成为继C和C++之后的下一门流行的主流商业软件开发语言，之所以这样的一个基本原因是，Java的体系结构能帮助程序员适应发展的硬件环境，Java具有在硬件环境中按照要求切换的特性，这都是由Java虚拟机提供的能力。

编程语言革命由硬件的发展所推动（当然还有更多推动力）。硬件在飞速发展，变得更加廉价且功能更加强大，软件变得越来越庞大、越来越复杂。从汇编语言到结构化语言的转变（比如C），以及到面向对象语言的转变（比如C++），在很大程度上是为了满足管理更高复杂度软件的需要——不断强大的硬件使得复杂度可能更高。

今天，获得更廉价、更快速、更强大硬件的势头仍在继续，软件复杂度不断增长的势头也在继续。在C和C++基础上，Java帮助程序员解决了一些复杂性，因为一些在C和C++中常见的固定类型的bug不再存在了。Java与生俱来的内存安全性——垃圾收集、取消了指针算法、在使用引用的时候进行运行时检查，避免了可能曾出现在Java程序中的大多数内存bug。Java的内存安全性使程序员生产效率更高，并在复杂度管理方面给他们提供了帮助。

除了持续增长的硬件能力之外，另外一个基础的硬件环境变化就是网络。网络把越来越多的计算机和设备连接起来，对软件提出了新的要求。随着网络的兴起，平台无关性和安全性也

变得更加重要了。

Java虚拟机负责Java程序设计语言的内存安全、平台无关和安全特性。虽然虚拟机在Java之前已经出现一段时间了，但是没有进入主流。然而，在今天不断变化的硬件环境现实面前，软件开发者需要一种使用虚拟机的编程语言。Sun用Java打开了这个市场的窗口。

也就是说，Java虚拟机为未来数年装备了正确的软件特性。本书会帮助读者理解Java虚拟机以及密切相关的几种Java API。有了这些知识，再通过自己的努力，就能使Java独一无二的体系结构发挥出更大的效能。

本书读者对象

本书主要是针对想了解Java技术的专业软件开发者和学生编写的。我假设读者对Java语言已经比较熟悉（但不需要精通），阅读本书会帮助读者深入理解Java编程知识。如果你是编写Java编译器或者编写Java虚拟机实现的少数精英之一，本书可以看作是对Java虚拟机规范的补充，书中对规范做出了解释。

如何使用本书

本书由五个部分组成：

1)对Java体系结构的介绍（第1～4章）。

2) Java内部细节的深入技术教程（第5～20章）。

3) class文件和指令集的索引参考（第6章和附录A～C)。

4) 交互式例示和示例源代码（在随书光盘中）。

5) 资源页（http://www.artima.com/insidejvm/resources/)。

对Java体系结构的介绍

第1～4章（本书的第一部分）给出了Java体系结构的总览，包括隐藏在Java体系结构设计背后的动机。这几章展示了Java虚拟机是如何与Java体系结构的其他组成部分（class文件、API和编程语言）相互关联的。如果想对Java技术有一个基础的了解，请阅读这些章节。下面是这部分的提要。

第1章“Java体系结构介绍”，在Java体系结构的概览和内部细节讨论上做了合理取舍。

第2章“平台无关”，讨论了平台无关的确切含义，Java体系结构是如何支持这个特性的，以及创建平台无关的Java程序的步骤。

第3章“安全”，描述了Java核心体系内置的安全模型，包含一个经精心制作的、可运行的例子，该例子示范了1.2版Java安全框架中的细粒访问控制的好处。

第4章“网络移动性”，讨论了网络移动软件的新范型。

Java内部技术教程

第5～20章(本书的第二部分)给出了Java虚拟机和相关核心Java API内部工作的深入技术描述，这些章节会帮助读者理解Java程序的实际运作情况。第二部分内容按照教程的方式组织，有很多示例。下面是这部分的提要。

第5章“Java虚拟机”，给出了对Java虚拟机内部工作的全面概览。

第6章“Java class文件”，是一份关于class文件格式的完整的教程和参考。如果你正在解析、生成或者比较关注Java class文件，那么这一章非看不可。

第7章“类型的生命周期”，讨论了类在Java虚拟机中的完整的生命周期，包含类被卸载的环境。

第8章“连接模型”，完整解释了Java的连接模型，包括使用forName（）和类装载器的例子，以便在运行时用新类型对Java应用程序进行动态扩展。

第9章“垃圾收集”，讨论了垃圾收集和终结（finalization），解释了什么是软、弱和影子引用，也提出了如何使用终结方法。

第10～19章是关于Java虚拟机指令集的教程。

第20章“线程同步”，解释了什么叫做监视器，以及如何使用它们编写线程安全的Java代码。

class文件和指令集参考

第6章除了作为Java class文件的教程之外，同时也是class文件格式的完整参考。同样，第10～20章构成了Java虚拟机指令集的教程，而附录A～C是指令集的完整参考。如果读者需要查阅有关指令的内容，请参见这些章节和附录。

交互式例示和示例源代码

本书的大多数章节在随书光盘上都找得到相关的材料——比如示例代码或者模拟applet。

随书光盘的applets目录中包含了一个叫做“Interactive Illustrations Web Site”的迷你Web网站，其中包含了15个Java applet，它们描绘了文中叙述的概念。这些交互式例示是本书的整体组成部分，其中11个applet模拟了Java虚拟机执行字节码，其他的演示了垃圾收集、二进制补码和IEEE 754标准的浮点数以及装载class文件的过程。这些applet可以在任何平台上、使用任何具有Java能力的浏览器浏览。这些模拟applet的源代码也包含在随书光盘上。

在“Interactive Illustrations Web Site”目录中的HTML，.java和.class文件根据版权声明允许读者把它们张贴到网络上（包括Internet）——只是必须遵守一些简单的规则。比如，必须完整地张贴整个站点（不能做任何修改），并且不能向浏览这个站点的人收费。版权声明的全文会在下面给出。

随书光盘中所有的示例源代码都包含源代码形式和编译过的形式(class文件)。如果对文本中的某个例子有兴趣或感到好奇，可以自己试验一下。

大部分示例代码都是用于解释目的的，除了帮助读者理解Java之外没有什么实际价值。不管怎样，读者可以从示范代码中随意拷贝、粘贴，用在自己的程序中，或者用二进制形式（比如Java class文件）发布。关于示例源代码的版权声明的全文会在下面给出。

Java虚拟机资源页面

为了让读者了解更多的信息，跟上时代的变化，我在artima.com维护一些页面，其中包含一些链接，指向与本书内容相关的阅读材料。这些链接页面的主页面是“Java虚拟机资源页面”，URL是 http://www.artima.com/insidejvm/resources/。

每章综述

第1章　对Java技术做了介绍，给出了Java体系结构的纵览，讨论了为什么Java很重要以及Java的优缺点。

第2章　展示了Java体系结构是如何让程序在任何平台上运行的，讨论了决定Java程序实际可移植性的要素，还考察了如何在可移植性及性能方面保持相应的平衡。

第3章　对内置于Java核心体系中的安全模型进行了深入概述，追踪了Java安全模型的演变过程——从1.0版本的沙箱到1.1版本的代码签名和验证，再到1.2版本的细粒度访问控制。

第4章　考察了Java带来的网络移动软件的新范型，并且展示了Java体系结构是如何让这项功能得以实现的。

第5章　给出了Java虚拟机内部体系的详细概述。随书光盘上与该章对应的applet叫做“Eternal Math”，它模拟了Java虚拟机执行一小段Java字节码的情况。

第6章　讲述了class文件的内容，包括常量池的结构和格式。这一章既可以作为Java class文件格式的教程，也可以作为class文件的完整参考。随书光盘上与该章对应的applet叫做“Getting Loaded”，它模拟了Java虚拟机装载一个Java class文件的过程。

第7章　对一个类型（类或者接口）的生命周期（从类型进入虚拟机到它最终退出）进行跟踪。该章还讨论了装载、连接和初始化的过程；还有如何创建对象示例，垃圾收集和终结；以及类型卸载。

第8章　深入考察了Java的连接模型，描述了类装载器的双亲委派模型、常量池解析、命名空间和装载约束。这一章还揭示了如何使用forName（）和类装载器，以便可以在运行时动态扩展Java应用程序。

第9章　讲述了垃圾收集的几种不同技术，解释了虚拟机中垃圾收集的工作原理——包含对火车算法以及对软引用、弱引用和影子引用的讨论。随书光盘上与该章对应的applet叫做“Heap of Fish”，它模拟了一个压缩的、“标记并清除”的垃圾收集堆。

第10章　讲述了用于操作数栈的Java虚拟机指令——把常数压入栈、进行通常的栈操作、在局部变量和栈之间互相传递数值等等。随书光盘上对应该章的applet是“Fibonacci Forever”，它模拟了Java虚拟机执行一个方法（该方法产生斐波那契序列）的过程。

第11章　讲述了在主要类型之间互相转换数值的指令。随书光盘上对应该章的applet是“Conversion Diversion”，它模拟了Java虚拟机执行一个方法（它进行类型转换）的过程。

第12章　讲述了Java虚拟机中的整数算法，解释了二进制补码算法，列出了用于整数计算的指令集。随书光盘上对应该章的有两个applet，它们以交互式例示形式描绘了该章的内容：其中一个applet叫做“Inner Int”，它可以让读者查看并操作二进制补码；另一个叫做“Prime Time”，它模拟了Java虚拟机执行一个方法（它生成质数）的过程。

第13章　讲述了Java虚拟机内部进行逐位运算、逻辑运算的指令，这些指令包括对整数进行小数点移位和Boolean(布尔)操作的操作码。随书光盘上对应该章的applet叫做“Logical Results”，它模拟了Java虚拟机执行一个方法（该方法使用一些逻辑操作码）的过程。

第14章　介绍了浮点数和Java虚拟机中执行浮点运算的指令，包括对strictfp关键字的讨论和出现在第2版Java虚拟机规范中的修改过后的浮点规则。随书光盘上有两个使用交互式例示来阐述该章内容的applet，一个名为“Inner Float”的applet能用来对组成浮点数的各个部分进行观察和操作，另一个名为“Circle of Squares”的applet能对Java虚拟机进行模拟，模拟它执行一个方法（它使用浮点操作码）的情况。

第15章　讲述了创建和操作对象和数组的Java虚拟机指令。随书光盘中为该章准备了一个名为“Three Dimensional Array”的applet，它模拟了Java虚拟机执行一个方法（它分配和初始化

三维数组）的过程。

第16章　介绍了控制Java虚拟机在同一个方法中进行条件或者无条件分支操作的指令。随书光盘中为该章准备了一个名为“Saying Tomato”的applet，它模拟了Java虚拟机执行一个方法的过程，该方法包含完成表跳转的字节码（Java源代码中switch语句编译后的版本）。

第17章　对字节码实现异常的方式、显式抛出异常的指令、异常表以及catch子句的工作方式进行了描述。随书光盘中为该章准备了一个名为“Play Ball!”的applet，它模拟了Java虚拟机执行一个方法（它抛出及捕获异常）的过程。

第18章　介绍finally子句在字节码中实现的方式，并举例介绍了相关指令。该章还对Java源代码中finally子句所展现的一些令人惊讶的特性进行了描述，并在字节码层对此特性做出了解释。随书光盘中为该章准备了一个名为“Hop Around”的applet，它模拟了Java虚拟机执行一个方法（它包含finally子句）的过程。

第19章　对Java虚拟机用来调用方法的四条指令以及使用这四条指令的环境进行了介绍。

第20章　讲述了监视器（Java用来支持同步的机制），并阐述了Java虚拟机使用它们的方式。该章还描述了指令集在数据的锁定和解锁方面对监视器的支持。

附录A　按照助记符的字母顺序列出操作码。对于每个操作码，该附录都给出了助记符、操作码字节值、指令格式（操作数，如果有的话）、指令执行前后堆栈快照，还描述了指令执行过程。附录A可作为指令集参考手册。

附录B　按照功能分组操作码。该附录中所使用的组织方式与指令在第10～20章中出现的顺序相对应。

附录C　按照操作码字节值的数字顺序排列操作码。对于每一个数值，该附录都给出了对应的助记符。

附录D　描述了最后一个applet，名为“Slices of Pi”，此applet模拟了Java虚拟机计算π的过程。

版权声明

下面是每个示例源代码文件（光盘中除了applets和jdk两个子目录之外的所有内容）都适用的版权声明：

LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR PARTICULAR PURPOSE, OR NON-INFRINGEMENT. BILL VENNERS SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY A LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES.

Interactive Illustrations网站的HTML页（包括applet）和Java源文件（存储在光盘的applets目录中）都遵循下列版权声明：

All the web pages and Java applets delivered in the applets directory of the CD-ROM, consisting of ".html," ".gif," ".class," and ".java" files, are copyrighted © 1996, 1997 by Bill Venners, and all rights are reserved. This material may be copied and placed on any commercial or non-commercial web server on any network (including the internet) provided that the following guidelines are followed:

a. All the web pages and Java Applets (".html," ".gif," ".class," and ".java" files), including the source code, that are delivered in the applets directory of the CD-ROM that accompanies the book must be published together on the same web site.
b. All the web pages and Java Applets (".html," ".gif," ".class," and ".java" files) must be published "as is" and may not be altered in any way.
c. All use and access to this web site must be free, and no fees can be charged to view these materials, unless express written permission is obtained from Bill Venners.
d. The web pages and Java Applets may not be distributed on any media, other than a web server on a network, and may not accompany any book or publication.

BILL VENNERS MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF THE SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR PARTICULAR PURPOSE, OR NON-INFRINGEMENT. BILL VENNERS SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY A LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES.

一些术语

在本书中，我尝试使用与Java语言和Java虚拟机规范相一致的术语。考虑到有些读者可能不熟悉这套术语，因此先简单阐明一些术语。

首先，在本书中我尝试细分使用术语，"类型"和"类"。在Java的术语中，变量和表达式都有类型，对象和数组都有类。Java程序中的每一个变量和表达式都有编译时可以确认的类型——或者为基本类型（int、long、float、double等等），或者为引用类型（类、接口或者数组）。变量或表达式的类型决定了它所拥有的值的范围和种类、所支持的操作以及这些操作的含义。

在运行时，每一个对象和数组都有一个类。虽然对象是它的类和所有超类的实例，但是它只拥有一个类。一个对象的类可以是以下情形之一：

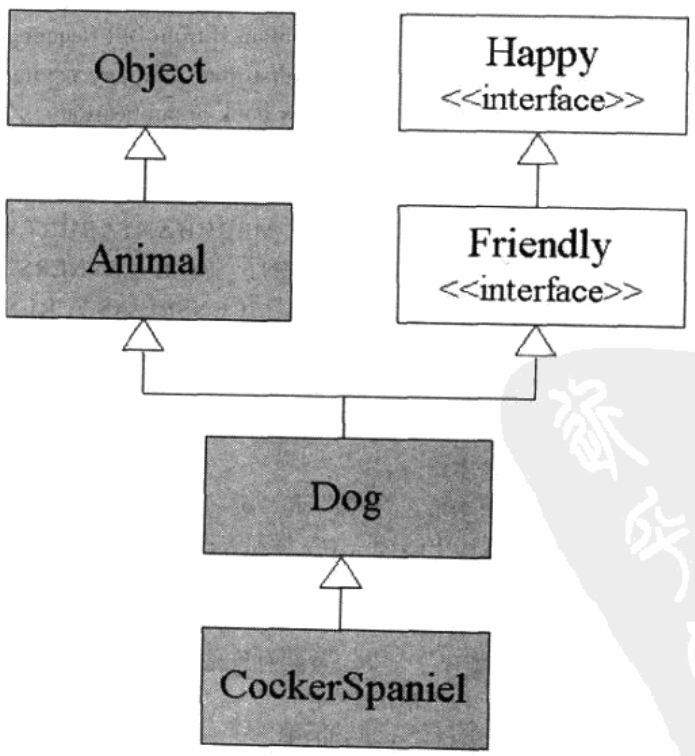
- 创建对象时所使用的创建实例表达式中提到的类。
- 使用newInstance（）方法创建对象时所使用的Class对象所代表的类。

- 使用clone（）方法创建对象时被克隆对象的类。
- 对以前序列化的对象解除序列化时，创建的对象具有与原对象相同的类。

数组类的名字是如[D或者[[[[I样的形式，这在Java语言中是不合法的（数组类的名字在第6章描述）。如果在运行时变量有一个非null的引用类型，那个变量就会指向一个对象，该对象的类和变量的类型必须是兼容的。

更复杂一点来说，在规范中包含对“类型”这个术语的另一个不同用法。因为变量可以声明自己的类型是某个类或者接口，所以类和接口可以定义程序使用的新类型（当然，创造新类型的能力是面向对象编程的基本概念之一）。在全书中，我只针对类使用“类”这个术语（而不是包含类和接口二者）。同样，我只针对接口使用“接口”这个术语。当我想同时表达二者的时候，我有时候说“类和接口”，但更经常使用“类型”。比如说，当我说“当类装载器装载了一个新的类型……”的时候，我的意思是“当类装载器装载了一个新的类或者接口”。在这句话中，类型不是指编译时一个变量声明的类型，而是指每个类和接口定义都代表的新的类型。

另一些我想事先澄清的术语是，在Java规范中描述两个类型（类或者接口）之间在继承图中的关系时所用的术语。考虑在下图中展示的继承图，在这个图中，类CockerSpaniel扩展自类Dog，Dog又是扩展自类Animal，而后者从类Object扩展而来。除此之外，接口Friendly扩展自接口Happy，类Dog实现了接口Friendly。



继承图

在Java术语中，在继承图中位于更高位置的类叫做“超类”，在较低位置的叫做“子类”。在上图中，Dog的超类是Animal和Object，Dog是Animal和Object二者的子类。在继承图上直接和类相连、紧邻上方的超类叫做“直接超类”。同理，直接连接在下方的叫做“直接子类”。例如，Animal是Dog的直接超类，CockerSpaniel是Dog的直接子类。

术语“子”和“超”的概念也可应用于接口，例如，接口Happy和Friendly是Dog和CockerSpaniel的超接口，接口Friendly是Dog的直接超接口及Happy的直接子接口。使用“子”和“超”术语概念的最后一种情况是表示“类”和“接口”的组合概念时，即“类型”概念中，在上图中，Friendly、Dog和CockerSpaniel是Happy的“子类型”，而Object、Animal、Happy、Friendly和Dog都是CockerSpanied的“超类型”。

Java版本和规范版本

本书的文本针对Java 2 SDK 版本1.2和第2版Java虚拟机规范。虽然本书包含的内容在版本1.0和1.1之间变动不多，但是在版本1.1和1.2之间有很大变化。第2版Java虚拟机规范要比第1版规范澄清了很多问题，也做了一些修订。

JDK版本1.0.2引进的一个变化是invokespecial指令的语义，这在第19章和附录A中描述。版本1.1中在class文件格式中增加了两个属性，用来支持内部类，还增加了一个属性用来支持@deprecated javadoc标签，它们都在第6章描述。版本1.1中还扩展了ClassLoader类的API，第8章演示了这个API的1.1版本。

本书第1版中讲述的Java API 1.1版本，和本书第2版中讲述的1.2版本在某些方面发生了很大变动。对本书影响最大的变动可能要算1.2版本的安全模型了。关于1.2版本的安全模型组件——基本沙箱、代码签名和验证、策略和策略文件、权限、代码源、保护域以及访问控制器的栈检查算法，这些都在第3章详细讲述。在Java语言1.2版中加入的strictfp关键字以及在class文件格式中加入的相应的访问标志，这些在第6章解释。1.2版本中引入的类装载器的双亲委派模型，还有1.2版本中引入的Java.lang.Class类和java.lang.ClassLoader类的几个新方法，在第8章讲述。软、弱和影子引用是在1.2版本的Java API中加入到java.lang.ref包中的，它们在第9章讲述。

除了对Java版本1.2中新加入的几种API进行介绍外，本书还解释了第2版Java虚拟机规范对原规范所做的很多澄清和修正。比如，第2版Java虚拟机规范中记录了新的装载约束，用于在存在多个类装载器时保证类型安全连接，这些装载约束在第8章进行了描述，并且有代码示例。第2版Java虚拟机规范中指出的对Java虚拟机浮点数规则的修订，在第14章中做了解释。本书的第2版还对class文件的版本号、方法调用、装载、连接和初始化类型方面的很多修正和澄清进行解释。

本书全书所使用的字节码示例都是用Sun公司的JDK版本1.1中附带的javac编译器生成的。记住，编译类可以有很多种方法。不同的编译器，或者是同一个编译器的不同版本都可能产生不同的结果。

本书中使用的模拟applet（交互式例示）的源代码遵守Java版本1.0。就像将在第2章中讨论的那样，Java平台无关的承诺所面对的现实之一就是，在使用一个特定的目标Java平台版本时，必须自行判断何时这个版本已经被广泛安装，何时才是值得使用这个平台的时机。虽然我1997年就有针对1.1版本Java虚拟机的模拟applet了，但在为本书第1版配CD-ROM时，我还是决定回

目　录

第1章 Java体系结构介绍

Java技术的核心就是Java虚拟机——所有的Java程序都在其上运行。尽管Java这个名字一般用于描述Java程序设计语言，但是除了语言本身以外还有很多其他含义。需要Java虚拟机、Java API 和Java class文件的配合，Java程序才能够运行。

本书的前4章展示了Java虚拟机在这个大结构中的地位。它们揭示虚拟机是如何和Java体系中的其他部分（class文件、API和语言）互动的，还描述了Java技术的整体设计背后的动机和含义。

本章讲述作为一门技术的Java，文中给出了Java的概览，讨论为什么Java重要的理由，以及Java的优缺点。

1.1 为什么使用Java

多年以来，人们使用工具来帮助完成任务，直到最近我们的工具才开始变得越来越聪明，并且互相连接起来。微处理器已经出现在很多日常使用的物件中，并且越来越多地和网络有了联系。举个例子，作为个人计算机和工作站的心脏，微处理器普遍连接到了网络。微处理器也在除了PC和工作站之外的专用设备上出现了，电视机、录像机、组合音响、传真机、扫描仪、打印机、蜂窝电话、电子记事簿、传呼机，甚至手表都装上了微处理器，并且大多数都可以上网。再加上信息处理能力的不断提高和费用的不断下降，网络正在迅速扩张它的触角。

逐渐通过网络连接起来的智能设备和计算机组成的基础结构开创了软件的新环境，它为软件开发者带来新的挑战和机遇。Java可以很好地帮助软件开发者面对挑战，抓住机会，因为Java是为网络而设计的。Java的这种适合网络环境的能力是由其体系结构决定的，它可以保证安全的、健壮的且和平台无关的程序通过网络传播，在很多不同的计算机和设备上运行。

1.2 网络带来的挑战和机遇

软件开发者面对的挑战之一是这种逐渐增长的以网络为核心的硬件环境，其包含的设备越来越广泛。一般网络都有很多不同的设备、不同的硬件体系、不同的操作系统，用于不同的用途。Java通过创建与平台无关的程序来解决这个问题。一个Java程序可以不需要修改就在很大范围内的计算机和设备上运行。和为一个特定的系统以及操作环境编译的程序比起来，用Java编写的平台无关的程序会更容易编写、管理和维护，代价也更低。

网络为软件开发者带来的另外一个挑战是安全性。除了潜在的好处，网络也为恶意的程序员打开了渠道，他们可以窃取或者破坏信息，偷盗计算资源，或者令人生厌地搞些恶作剧。举例来说，病毒编写者可以把他们的程序放在网络上，让没有防护的用户下载。Java解决这个挑战的方法是，提供一个受保护的环境，从网络上下载的程序可以以不同的定制安全级别运行。

安全性的一个方面是程序的健壮性。如同恶意程序员写的狡猾的代码一样，善意的程序员编写的有缺陷（bug）的程序也有可能破坏信息，把计算机带入死循环，或者导致系统崩溃。

Java体系结构对程序健壮性有一定的保证，一些有害代码不会在Java代码中出现，比如内存泄漏。这种体系结构可以保障从网络上下载的代码不会无意（或者有意）地崩溃。它还带来和网络无关的另外一个好处：提高程序员的生产力。因为Java先天上防止了很多bug的出现，Java程序员不需要在发现和修正它们上浪费时间。

无所不在的网络带来了一个机会就是在线程序发布。Java利用这个优点在网络上传送小段的二进制代码。通过这个能力，Java程序的发布会比原有的程序发布更加方便廉价。这也为版本控制带来了好处。因为最新版本的Java程序可以在用户需要的时候从网络上得到，所以不需要担心最终用户运行程序的版本，他们每一次总会得到最新的版本。

可移动的代码带来了另一个机会：运动的对象——它同时在网络上传递代码和状态。Java实现了对象移动的诺言——通过它的对象序列化API和RMI（远程方法调用）。在Java的底层结构之上，对象序列化和RMI为分布式系统中的各个部分共享对象提供了基础。对象在网络上的运动性为分布式系统编程创造了新模型，有效地把面向对象编程的优点带到了网络上。

平台无关性、安全性和网络移动性，Java体系的这三个方面共同使得Java和发展中的网络计算环境相得益彰。因为Java程序是平台无关的，可以在网络上移动的代码和对象就更加有效可行了。同样的代码可以被送到网络所连接的所有计算机和设备上。不同硬件平台上运行的分布式系统的各个不同组件可以互相交换对象。Java的内置安全性框架也可以帮助网络上的软件传送更加有效。因为降低了风险，安全性框架可以使新范型的、可通过网络移动的软件是可信任的。

1.3 体系结构

Java 体系结构包括四个独立但相关的技术：

- Java程序设计语言。
- Java class文件格式。
- Java应用编程接口（API）。
- Java虚拟机。

当编写并运行一个Java程序时，就同时体验了这四种技术。用Java编程语言编写源代码，把它编译成Java class文件，然后再在Java虚拟机中运行class文件。当编写程序时，通过调用类（这些类实现了Java API）中的方法来访问系统资源（比如I/O）。当程序运行的时候，它通过调用class文件中实现了Java API的方法来满足程序的Java API调用。可以在图1-1中看到这四者之间的联系。

Java虚拟机和Java API一起组成了一个“平台”，所有Java程序都在这上面编译。Java虚拟机和Java API的组合除了被称为Java运行时系统之外，还被称为Java平台（从版本1.2开始，称为Java 2平台）。Java程序可以在不同的计算机上运行，这是因为Java平台自己可以用软件实现。从图1-2可以看出，Java程序可以在有Java平台的任何地方运行。

1.3.1 Java虚拟机

Java的面向网络的核心就是Java虚拟机，它支持Java面向网络体系结构三大支柱的所有方面：平台无关性、安全性和网络移动性。

Java虚拟机是一台抽象的计算机，其规范定义了每个Java虚拟机都必须实现的特性，但是为

每个特定实现都留下了很多选择。举个例子，虽然每个Java虚拟机都必须能够执行Java字节码，但是用何种技术来执行是可以选择的。而且，它的规范也很灵活，它允许虚拟机用纯粹软件方式来实现，也可以很大部分由硬件实现。Java规范本质上的灵活性保证了它能在很广泛的计算机和设备上得到实现。

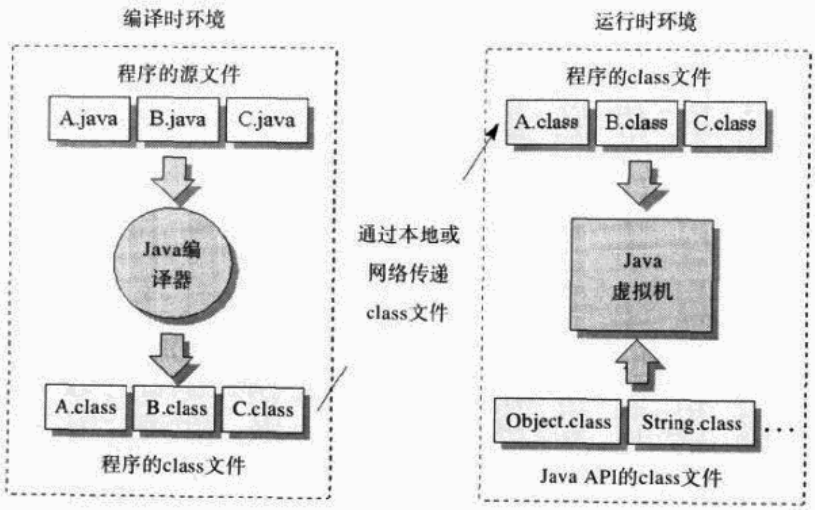


图1-1 Java编程环境

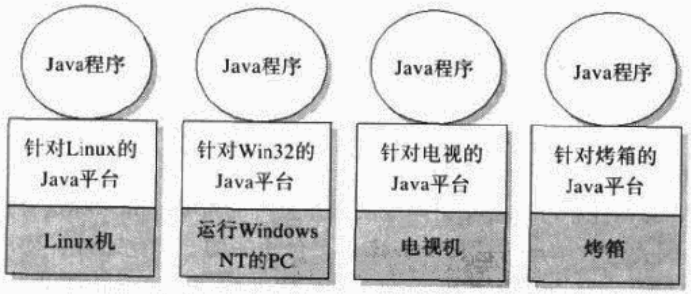


图1-2 在Java平台上运行的Java程序

Java虚拟机的主要任务是装载class文件并且执行其中的字节码。在图1-3中可以看到，Java虚拟机包含一个类装载器（class loader），它可以从程序和API中装载class文件。Java API中只有程序执行时需要的那些类才会被装载。字节码由执行引擎来执行。

不同的Java虚拟机中，执行引擎可能实现得非常不同。在由软件实现的虚拟机中，最简单的执行引擎就是一次性解释字节码。另一种执行引擎更快，但是也更消耗内存，叫做“即时编译器”（just-in-time compiler）。在这种情况下，第一次被执行的字节码会被编译成本地机器代码。编译出的本地机器代码会被缓存，当方法以后被调用的时候可以重用。第三种执行引擎是自适应优化器。在这种方法里，虚拟机开始的时候解释字节码，但是会监视运行中程序的活动，并且记录下使用最频繁的代码段。程序运行的时候，虚拟机只把那些活动最频繁的代码编译成本

地代码，其他的代码由于使用得并不很频繁，继续保留为字节码——由虚拟机继续解释它们。一个自适应的优化器可以使得Java虚拟机在80%～90%的时间里执行被优化过的本地代码，而只需要编译10%～20%对性能有影响的代码。最后一种虚拟机由硬件芯片构成，它用本地方法执行Java字节码，这种执行引擎实际上是内嵌在芯片里的。

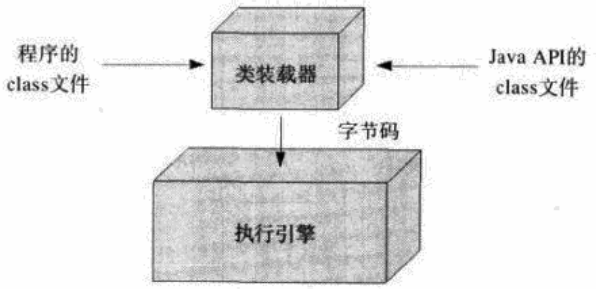


图1-3 Java虚拟机的基本结构图

有时候Java虚拟机被称为Java解释器。然而，考虑到执行字节码的方式可能是不同的，这个称谓可能造成误导。虽然对于直接解释字节码的Java虚拟机来说“Java解释器”是一个合理的名字。但是当讨论执行技术时，“解释”是一种我们所知道的易于实现而执行缓慢的特殊技术。因此，“Java解释器”只表示是“Java虚拟机”，但并没有任何执行技术的含义。

当Java虚拟机是由主机操作系统上的软件实现的时候，Java程序通过调用本地方法（native method）和主机交互。Java中有两种方法：Java方法和本地方法。Java方法是由Java语言编写，编译成字节码，存储在class文件中的。本地方法是由其他语言（比如C，C++，或者汇编语言）编写的，编译成和处理器相关的机器代码。本地方法保存在动态连接库中，格式是各个平台专有的。Java方法是与平台无关的，但是本地方法却不是。运行中的Java程序调用本地方法时，虚拟机装载包含这个本地方法的动态库，并调用这个方法。在图1-4中可以看到，本地方法是联系Java程序和底层主机操作系统的连接方法。

通过本地方法，Java程序可以直接访问底层操作系统的资源。如果你这样用，你的程序就变成了平台相关的，因为包含本地方法的动态库是平台相关的。除此之外，使用本地方法还可能把程序变得和特定的Java平台实现相关。一个本地方法接口——Java本地接口（Java Native Interface，JNI）——使得本地方法可以在特定主机系统的任何一个Java平台实现上运行。然而Java平台供应商并不一定必须支持JNI。除了JNI之外，他们还可以提供自己的本地方法接口（或者按照合同要求来取代JNI）。

Java给人们提供了选择的机会。如果希望使用特定主机上的资源，它们又无法从Java API访问，那么可以写一个平台相关的Java程序来调用本地方法。如果希望保证程序的平台无关性，那么只能通过Java API来访问底层系统资源。

1.3.2 类装载器的体系结构

类装载器的体系结构是Java虚拟机在安全性和网络移动性上发挥重要作用的一个方面。在图1-3和1-4中，仅有一个标示为“类装载器”的神秘方块，但实际上在Java虚拟机中，存在着多个类装载器，因而结构图中的类装载器方块实际上表示的是一个可能包含多个类装载器的子系统。Java

虚拟机拥有灵活的类装载器体系结构，从而使Java应用程序得以用自定义的方式来实现类的装载。

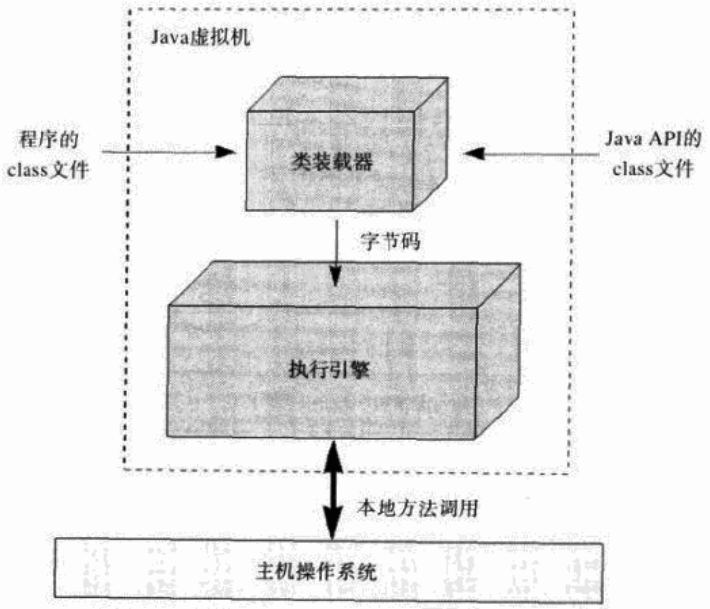


图1-4 在主机操作系统上由软件实现的Java虚拟机

一个Java应用程序可以使用两种类装载器："启动"（bootstrap）类装载器和用户定义的类装载器。启动类装载器（这是系统中惟一的）是Java虚拟机实现的一部分。例如，如果Java虚拟机在已有操作系统上实现为C程序，那么启动类装载器就会是此C程序的一个部分。启动类装载器通常使用某种默认方式从本地磁盘中装载类，包括Java API的类（启动类装载器也被称为原始类装载器、系统类装载器或者默认类装载器。在1.2版本中，"系统类装载器"这个名称被赋予新的含义，此含义将在第3章讨论）。

Java应用程序能够在运行时安装用户定义的类装载器，这种类装载器能够使用自定义的方式来装载类，例如，从网络下载class文件。尽管启动类装载器是虚拟机实现的本质部分，而用户定义的类装载器不是；但用户定义的类装载器能够用Java编写，能够被编译为class文件，能够被虚拟机装载，还能够像其他对象一样实例化。它们实际上只是运行中的 Java应用程序可执行代码的一部分。图1-5描述了这种体系结构。

由于有用户定义类装载器，所以不必在编译的时候就知道运行中的Java应用程序中最终会加入的所有的类。用户定义的类装载器使得在运行时扩展Java应用程序成为可能。当它运行时，应用程序能够决定它需要哪些额外的类，能够决定是使用一个或是更多的用户定义的类装载器来装载。由于类装载器是使用Java编写的，所以能用任何在Java代码中可以表述的风格来进行类的装载。这些类可以通过网络下载，可以从某些数据库中获取，甚至可以动态生成。

每一个类被装载的时候，Java虚拟机都监视这个类，看它到底是被启动类装载器还是被用户定义类装载器装载。当被装载的类引用了另外一个类时，虚拟机就会使用装载第一个类的类装载器装载被引用的类。例如，如果虚拟机使用一个特定的类装载器装载Volcano这个类，它就会

使用这个类装载器装载Volcano类使用的所有类。如果Volcano使用了一个叫做Lava的类，比方说，可能是调用了Lava类的一个方法，那么虚拟机将使用装载Volcano的同一个类装载器装载Lava。这样，被这个类装载器返回的Lava类就动态地与Volcano类建立起了联系。

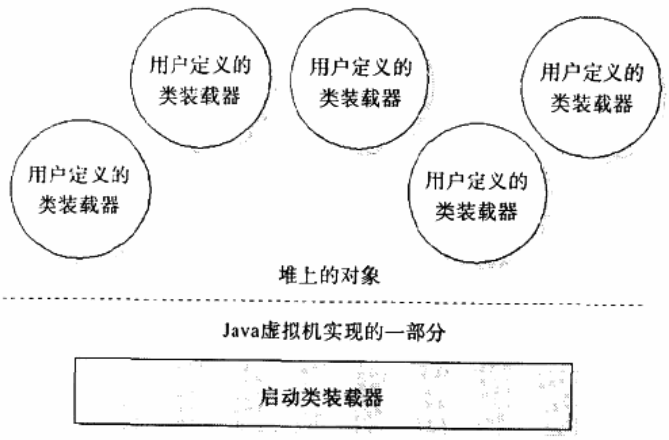


图1-5 Java类装载器的体系结构

由于Java虚拟机采取这种方式进行类的装载，所以被装载的类默认情况下只能看到被同一个类装载器装载的别的类。通过这种方法，Java的体系结构允许在一个Java应用程序中建立多个命名空间。运行时的Java程序中的每一个类装载器都有它自己的命名空间。

一个Java应用程序能够从同一个类或者多个类中实例化多个用户定义的类装载器。因此，需要多少个（或多少种）用户自定义的类装载器，Java应用程序就可以创建多少个（或多少种）。被不同的类装载器装载的类存放在不同的命名空间中，它们不能相互访问，除非应用程序显式地允许这样做。当编写一个Java应用程序的时候，从不同源文件装载的类可以分隔在不同的命名空间中。通过这种方法，就能够使用Java类装载器的体系结构来控制任何从不同源文件中装载的代码之间的相互影响，特别是能够阻止恶意代码获取访问和破坏善意代码的权限。

Web浏览器是一个动态扩展的例子，Web浏览器使用用户定义的类装载器从网络下载用于Java applet的class文件。Web浏览器使用一个用来安装用户定义类装载器的Java应用程序，这个用户定义类装载器通常被称为Java applet类装载器，它知道如何向HTTP服务器请求class文件。Java applet可以作为动态扩展的例子，因为Java应用程序并不知道它什么时候会开始从网络下载浏览器请求的class文件。只有当浏览器遇到有Java applet的页面的时候，才决定是否需要下载class文件。

Web浏览器启动的Java应用程序通常为每个提供class文件的网络地址分别创建不同的用户定义类装载器，因此，不同的用户定义类装载器装载不同来源的class文件。这就可以把它们分别放置在Java主机应用程序的不同命名空间之中。由于不同来源的Java applet的class文件放置在不同的命名空间中，恶意的Java applet代码就不会直接妨碍到从别的地方下载的class文件。

通过允许实例化用户定义的类装载器（该类装载器）知道如何从网络下载class文件，Java类装载器的体系结构提供了对网络移动性的支持；通过允许使用不同的用户定义的类装载器装载不同来源的class文件，Java类装载器的体系结构提供了对安全性的支持。它把不同来源的class文

件放置在不同的命名空间中，这就能够限制或阻止不同来源的代码之间的相互访问。

1.3.3 Java class文件

Java class文件主要在平台无关性和网络移动性方面使Java更适应于网络。它在平台无关性方面的任务是：为Java程序提供独立于底层主机平台的二进制形式的服务，这正是Java虚拟机所期望实现的。这种途径打破了C或者C++等语言所遵循的传统，使用这些传统语言写的程序通常首先被编译，然后被连接成为单独的、专门支持特定硬件平台和操作系统的二进制文件。通常情况下，一个平台上的二进制可执行文件不能在其他平台上工作。而Java class文件是可以运行在任何支持Java虚拟机的硬件平台和操作系统上的二进制文件。

当编译和连接一个C++程序的时候，所获得的可执行二进制文件只能在指定的硬件平台和操作系统上运行，因为这个二进制文件包含了目标处理器的机器语言。而Java编译器把Java源文件的指令翻译成字节码，这种字节码就是Java虚拟机的“机器语言”。

除了特定处理器的机器语言之外，传统二进制可执行文件的另一个依赖于具体平台的属性是整数的字节顺序。比方说，在支持X86系列处理器的二进制可执行文件中，字节顺序是低位在前，而对于PowerPC处理器，则是高位在前。在Java class文件中字节顺序是高位在前，这与使用何种平台产生这个文件和在何种平台上使用这个文件都没有关系。

除了对于平台无关性的支持，Java class文件还在支持网络移动性的Java体系结构中担当了至关重要的角色。首先，class文件设计得紧凑，因此它们可以快速地在网络上传送。其次，由于Java程序是动态连接和动态扩展的，class文件可以在需要的时候才下载。这个特点使得Java应用程序能够安排从网络上下载class文件的时间，从而可以最大限度地减少终端用户的等待时间。

1.3.4 Java API

Java API通过支持平台无关性和安全性，使得Java适应于网络应用。Java API是运行库的集合，它提供一套访问主机系统资源的标准方法。编写Java程序时，可以假设在任何可运行程序的Java虚拟机上都能够获取Java API class文件。这是一个相对安全的假设，因为Java虚拟机和Java API class文件是任何Java平台都要实现的必要部分。运行Java程序时，虚拟机装载程序的class文件所使用的Java API class文件。所有被装载的class文件（包括从应用程序中和从Java API中提取的）和所有已经装载的动态库（包含本地方法）共同组成了在Java虚拟机上运行的整个程序。

Java API的class文件天生就与主机平台密切相关。在一个平台能够支持Java程序以前，必须在这个特定平台上明确地实现API的功能。为访问主机上的本地资源，Java API调用了本地方法。如图1-6所示，由于Java API class文件调用了本地方法，Java程序就不需要再调用它们了。通过这种方法，Java API class文件为底层主机提供了具有平台无关性的、标准接口的Java程序。对Java程序而言，无论平台内部如何，Java API都会有同样的表现和可预测的行为。正是由于在每个特定的主机平台上都明确地实现了Java虚拟机和Java API，因此，Java程序自身就能够成为具有平台无关性的程序。

Java API的内部设计也和平台无关性相关。例如，Java API的用户图形界面库——即AWT（Abstract Windows Toolkit，抽象窗口工具箱）和Swing的设计目的是使用户设计的界面能够在所有平台上运行。由于不同平台上用户界面的外观风格具有相当大的差异，创建平台无关的用户界面是一件很困难的事情。AWT类库体系结构并没有强制Java API把所有平台上的Java程序用

户界面都做成一个模式。恰恰相反，AWT类库体系结构鼓励在外观风格上适应于相应的底层平台。Swing类库甚至提供了更加灵活的方式：允许程序员自己选择外观风格。而且，由于平台与平台之间字体大小、按钮和其他用户界面控件存在相当大的差异，AWT和Swing都包含了在运行时定位窗体或对话框成员的布局管理器。当显示对话框时，布局管理器会安排对话框上控件的位置，而并非强迫为对话框中的各个控件都标明准确的X、Y坐标。为了使对话框的外观在每个平台上都能达到最佳效果，布局管理器在不同的平台上采用了些许不同的方式来定位对话框控件。除此之外，还有其他一些方面，Java API的内部系统结构把推动Java程序平台无关性的发展作为一个设计目的。

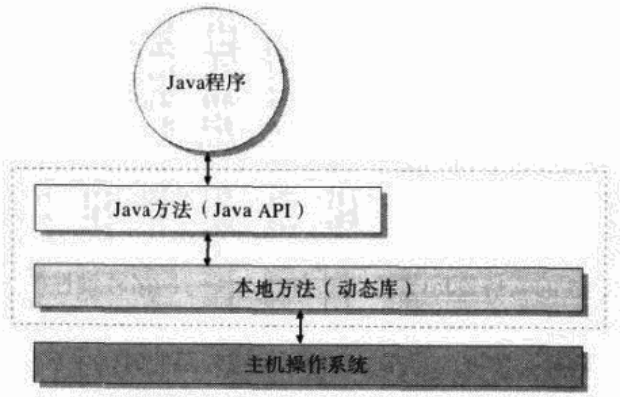


图1-6 一个具有平台无关性的Java程序

除了对平台无关性的推动之外，Java API在Java安全性模型方面也做出了贡献。当Java API的方法进行任何有潜在危险的操作（比如进行本地磁盘写操作）之前，都会通过查询安全管理器来检验是否得到了授权。安全管理器是一个为应用程序提供自定义安全策略的特殊对象。例如，安全管理器能够禁止对本地磁盘的访问。Java 1.2版本以前，如果应用程序通过调用Java API的方法来请求对本地磁盘进行写操作，那个API方法会首先请求安全管理器进行验证。当从安全管理器得知磁盘访问被禁止后，Java API会拒绝执行写操作。在Java 1.2版本中，安全管理器的工作被访问控制器所取代。访问控制器是一个类，该类用来执行栈检验，以决定是否准许某种操作（考虑到向下兼容问题，安全管理器在Java 1.2版本中仍然存在）。通过强制执行安全管理器和访问控制器建立的安全策略，Java API促进了安全环境的建立，在这种安全环境中，可以运行具有潜在危险的代码。

1.3.5 Java程序设计语言

尽管Java是为网络设计的，但Java的应用并不局限于网络。平台无关性、网络移动性和安全性是网络计算环境最重要的因素，但是并不是每个人都会遇到面向网络的问题。因此，也并不是任何时候都需要编写具备平台无关的程序。有时可能并不需要从网上传递程序或者通过安全约束限制程序的能力。有时使用Java技术的首要原因可能只是因为想要利用Java程序设计语言的优势。

总的来说，Java技术非常适用于网络，但Java程序设计语言是相当通用的。使用Java语言编

写程序，能够充分利用如下的许多软件技术：

- 面向对象。
- 多线程。
- 结构化错误处理。
- 垃圾收集。
- 动态连接。
- 动态扩展。

Java语言并不是作为新的、实验性技术的测试平台；相反，Java语言使用一种新的方法把已经在其他语言中得到验证的概念和技术组合到了一起。这些概念和技术使得Java程序设计语言成为强有力的通用的工具。它可以在多种领域使用，而并不仅仅局限于与网络相关的领域。

在一个新项目开始的时候，可能会面对这样一个问题："在下一个项目中，我应该用C++（或者别的什么语言）还是用Java呢？"作为一种程序语言，Java和别的语言相比，有其自身的优缺点。使用Java语言最引人注目的原因之一是，它能够提高开发者的效率；而它最主要的缺点在于执行速度较慢。

首先，Java是一门面向对象的语言。面向对象技术的承诺之一是提升代码的重用率，提高开发者的效率。这个因素可能会使Java较之过程语言（例如C语言）更具吸引力，但这并不意味着Java比C++更有价值。不过，和C++相比较，Java还是有一些可以提高开发者效率的十分重要的差别。这种效率的提升主要来自于Java对直接内存操作的约束。

与C++不同，在Java中，没有通过使用强制转换指针类型或者通过进行指针运算直接访问内存的方法。在Java中使用对象时，需要严格地遵守类型规则。如果存在一个Mountain类对象的引用（类似于C++中的指针），它只能作为Mountain使用。这个引用不能强制转换为Lava类型，更不能把它所指向的内存假设为一个Lava对象来操作。像在C++中使用指针运算那样，简单地给引用加上偏移量也是被禁止的。在Java中，将一个引用转换成别的类型是可以的，但只能在此对象的确为这种新类型的前提下才能进行。例如，如果一个Mountain的引用实际指向一个Volcano类（一种特殊的Mountain类型）的实例，这个Mountain的引用可以转换为Volcano的引用。由于Java在运行时强制执行严格的类型规则，根本无法以可能导致内存冲突的方式直接管理内存。因此，Java程序中不会出现那些常使C++程序员降低效率的特定的bug。

Java避免无意间破坏内存的另一个办法是自动垃圾收集。Java和C++一样，有一个new操作符，可以通过它来为新对象在堆中分配内存。但是和C++不一样的是，Java并没有与new相对应的delete操作符，C++程序员们常常使用delete来清除程序中不再需要的对象；而在Java中，只需要停止对一个对象的引用，一段时间后，垃圾收集器会自动回收这个对象所占据的内存。

垃圾收集器禁止Java程序员显式指明哪个对象应该被释放。当C++项目的大小和复杂性逐渐上升时，程序员决定哪一个对象应该被释放，或者判断一个对象是否已经被释放的难度也随之增加。如果不再使用的对象没有被释放，会导致内存泄漏；多次释放一个对象会导致内存冲突。这两种内存问题都会导致C++程序崩溃，但C++使用的这种内存管理方式使得确定问题所在相当困难。Java比C++更有效率的原因在于Java中不用再在内存冲突的问题上纠缠不清。而且，当不用再为显式释放内存担心的时候，效率会更高；程序设计也会更加容易。

Java在运行时保护内存完整性的第三个办法是数组边界检查。在C++中，数组操作实际上就是指针运算，这会带来潜在的内存冲突。在C++中能够声明一个有10个成员的数组，然后再向第11个成员写入（尽管这是错误的用法，但C++并不会限制这样做）。在Java中，数组是发展完备的对象，在每次使用数组的时候，Java都会检查数组的边界。当在Java中创建一个10个成员的数组，然后尝试向第11个成员写入的时候，Java会抛出一个异常。Java绝对不允许数组操作超出边界，从而导致内存冲突。

最后一个关于Java确保程序健壮性的例子是对对象引用的检查，每次使用引用的时候，Java都会确保这些引用不为空值。在C++中，使用一个空指针通常会导致程序崩溃。在Java中，使用一个空的引用只会导致一个异常被抛出。

使用Java语言能够提高生产效率，效率提升就带来了开发周期的缩短和开发费用的降低。如果利用到Java程序潜在的平台无关性，就会使费用进一步降低。即使你不关心网络应用，很可能也会希望在多个平台上发布程序。而Java可以更加容易地提供多平台支持，因此，成本会更低。

1.4 Java体系结构的代价

尽管Java面向网络的特征是如此吸引人，特别是在网络环境中，但是它们要以牺牲别的一些有价值的特性为代价。每当需要在两个理想的特性之间做出取舍的时候，Java的设计者只会选择在网络环境中表现得更好的那个。因此，Java并不是对所有工作都恰当的工具。Java适合解决网络相关的问题，而且也能解决一些和网络无关的问题，但是Java体系结构为侧重网络而付出的代价将会使Java丧失解决一些特定问题的资格。

和其他技术（诸如C++）相比，Java程序的执行速度可能比较低，这是Java在面向网络特性上所付出的最主要的代价之一。事实上，在Java诞生后的几年中，为了获取令人满意的性能，Java的开发者们进行了最艰苦的奋斗。尽管早期使用Java的经验使得开发者群体断言：Java很慢，但这并不是必然正确的结论。Java或许很慢，但它并不会永远这样慢。随着虚拟机技术的不断发展，虚拟机性能将会得到大幅度提高，到那时，Java的效率甚至会和本地编译的C程序不相上下。

第一个Java虚拟机诞生于1995年，它可以在一个解释器上执行字节码，这只是一项较为简单的技术，它的性能不高。不久，即时编译器诞生了。和解释器相比，它极大提高了Java的性能，但是仍然使Java在性能上大大落后于本地编译的C++程序。随着最近虚拟机技术的发展，即使不能说Java速度上的缺陷完全消失了，但也可以说有了显著的改善。适应性优化等先进技术使Java程序能够以和本地编译的C程序相媲美的速度运行。

尽管近期在Java性能方面的进展捷报频传，但这并不表示开发者在Java性能上的麻烦结束了。开发者所面对的困难是：即使特定的Java虚拟机能够提供相当好的性能，但一般情况下，开发者还是无法选择他们的程序将会在哪一种Java虚拟机上运行。Java体系结构的承诺之一是，Java程序能够在“任何地方”运行，这也就意味着能在任何Java虚拟机上运行。如果使用Java写一个内部使用的服务器应用程序，可能可以选择应用程序所运行的虚拟机。但是一旦Java程序拥有多个客户，它必须要在多种虚拟机上都有令人满意的性能。而且在一个为Java体系结构所鼓励的多种分布式系统所组成的世界里，随着代码和对象通过网络在虚拟机之间移动，开发者也无从知晓他们的程序究竟会运行在哪些虚拟机上。

最后，性能是否是一个问题，是否需要去解决这个问题，这些都取决于要做的究竟是什么。幸运的是，Java是一种非常灵活的工具，它提供许多方法去解决潜在的性能问题。例如，如果需要提供的仅仅是一个单独的可执行文件（就像字处理程序或者服务器进程），就可以：

- 随程序附带一种虚拟机。
- 把程序中对时间要求严格的部分作为本地方法实现。
- 把整个程序编译成一个的单独的可执行文件，就像C和C++一样。
- 在安装时，把程序编译成一个单独的可执行程序。

管理单独应用程序性能的最有效途径，可能就是自主选择虚拟机，但在某些情况下，把可执行部分或者整个程序本地化也许是最佳的解决途径。

把Java程序编译成单独的可执行程序的方法（有时也被称为“预编译”）能够改善性能，但这通常是以牺牲Java的动态扩展能力为代价的。预编译是进行静态而不是动态的连接，它产生出完全连接的、单独的本地可执行文件，但是这种可执行文件通常失去了运行时动态创建和连接新类型的能力。然而，对于无论怎样都不会使用动态扩展的Java程序，预编译至少应该产生与在传统虚拟机上运行的程序同样表现的程序。由于一些嵌入式系统并不需要动态扩展，而且通常都有资源限制，预编译经常被用来把一个Java程序编译为可烧入嵌入式系统的ROM里的本地可执行程序的映像文件。只要没有用到动态扩展，预编译也可以用于桌面应用。如果想解决的是相对独立的Java程序的性能问题而又不需要动态扩展，那么预编译可能是个好办法。

然而，当开发的不是一个单独的应用程序，而是一个分布式系统，特别是代码和对象需要在虚拟机之间移动的时候，性能管理就会变得十分困难。但毕竟这种面向对象的网络程序设计是Java结构的首要承诺之一。在这种情况下，性能管理的最佳途径就在于设计系统的方式。在这里必须凭借传统的技巧来提高性能，例如，减少网络流量，选择最佳的运算法则，以及任何语言中都会用的其他能够提升性能的标准方法。

尽管使用Java时，程序的速度是令人担忧的事情，但还是有很多方法可以弥补这一点。通过适当使用开发、发布和执行Java程序的各种技巧，大多数情况下还是能够取得令终端用户满意的速度。只要能够成功地解决速度问题，就能够使用Java语言并认识到它的益处——提高开发者的效率，为终端用户增强程序健壮性。

除了性能以外，Java的面向网络的体系结构所付出的另一个代价是，在内存管理和线程调度上的缺陷。垃圾收集器可以使得许多程序更加健壮，这也是网络环境中很有价值的安全性保障措施。但是垃圾收集器也给程序运行时的性能加入了一些不确定性，你无法确认垃圾收集器什么时候开始收集垃圾，无法确认垃圾收集器是否开始收集垃圾，也无法确认垃圾收集到底要持续多长时间。而且，Java虚拟机规范讨论线程调度的地方非常笼统，这种对线程行为的松散规范有利于将Java虚拟机移植到许多不同类型的硬件上去。尽管虚拟机的可移植性对于网络环境非常重要，但这种对线程管理含糊不清的说明使程序员无法了解应该如何调度线程，无法控制线程的调度。内存管理和线程调度上的缺陷使Java很难成为某些特定系统的可选方案，比如需要实时响应事件的软件。

Java为了实现平台无关性，也要付出代价，即最小公分母问题，这是在任何尝试提供跨平台功能的API上都会出现的固有困难。尽管在各个操作系统之间有许多共性，但每个操作系统通常

都有一些自己的特性。想要给程序提供访问任何操作系统的系统服务功能的API，就不得不决定究竟支持哪一种特性。如果某个特性只在一种操作系统上存在，API的设计者很可能会决定不支持这个特性。如果某个特性存在于绝大多数的操作系统中，但并不是所有的操作系统都有这个特性，设计者可能会决定无论如何都要添加对这个特性的支持。这时就需要在不具备这项特性的操作平台上由API模拟实现它。这些最小公分母的选择都将会在某种程度上得罪相关操作系统上的开发者和终端用户。

看上去太糟糕了，最小公分母问题不仅仅折磨平台无关性API的设计者，而且还影响到了使用这种API的程序的开发者。我们以用户界面为例，AWT努力尝试在每个平台上都给程序提供一个符合本地视觉习惯的界面。然而，如果需要设计在多平台上运行的用户界面，而控件在每个平台上都能像在本地系统上那样相互作用，这是很难做到的。AWT设计的时候已经经过了最小公分母选择，而用户在使用它的时候，也面临着自己的最小公分母选择问题。虽然Swing库能够给出更多的选择，但最终设计一个跨平台用户界面的时候，可能还是会被各平台上用户期望之间的差异困扰。

当把Java class文件与Java编程语言之间的紧密联系 和Java天生的动态连接特性联系到一起的时候，还要付出一个代价。因为Java程序是动态连接的，从一个类到另一个类的引用是符号化的。在静态连接的可执行程序中，类之间的引用只是直接的指针或者偏移量。相反，在Java class 文件中，指向另一个类的引用通过字符串清楚地标明了所指向的这个类的名字。如果引用是指向一个字段的话，这个字段的名字和描述符（字段的类别）会被详细说明。如果引用指向一个成员方法，那么这个成员方法的名称和描述符（方法的返回类型、方法参数的数量和类型）也会被详细说明。而且，Java的class文件不仅仅包含对其他类的字段和成员方法的符号引用，它们还包含对自己的字段和成员方法的符号引用。Java class文件还包含了可选的调试信息，这些调试信息包含局部变量的名称和类型。一个class文件的符号信息，以及字节码指令集和Java语言之间的密切关系，这些方面都使得把Java class文件逆向编译为Java源码文件相当容易。这就会使得你的竞争者能够很轻松地窃取到你辛勤工作的成果。

尽管竞争者总是可以逆向编译静态连接的二进制文件，并从你的程序中窃取思路，但比较起来，如果有了中间二进制文件（如Java的class文件）的话，逆向编译就会容易多了。逆向编译静态连接的二进制可执行文件相当困难，这并不仅仅因为缺失了符号信息（最初的类、字段、方法和局部变量名），还因为通常静态连接库的优化程度非常高。而静态连接二进制文件优化程度越高，它与初始源码的符合程度就越小。尽管如此，但如果你的竞争对手认为隐藏在二进制可执行文件中的算法值得他们历尽千辛万苦去获取的话，那么就没有办法阻止他们从二进制可执行文件中窃取这个算法。

然而，有一种办法可以防止这种知识产权被轻易窃取的问题：可以使用混淆器来混淆你的class 文件。混淆器通过更改类、字段、方法和局部变量名字的方法来修改类，但是它并不更改程序的工作流程。程序仍然能够被逆向编译，但是所有的类、字段、方法和局部变量的名称都不再是最初设定的有意义的名称。对于大型程序来说，混淆器可以使从逆向编译中得到的代码含义很模糊，所以，如果想要从这些代码中得到什么的话，还要花费几乎与分析静态连接的可执行文件相同的时间去分析。

1.5 结论

那么，什么是Java体系结构的要点呢？就像在这一章中所讨论的那样，Java程序设计语言是一种十分通用的语言，它和其他技术相比具有明显的优势。特别是，Java能在极大程度上提高程序员的效率，增强程序的健壮性，与老的程序设计技术（诸如C和C++）相比，具有过得去的性能。Java系统结构设计的核心并不仅仅在于使程序员更有效率，使程序更健壮，而在于为新兴的以网络为中心的计算环境提供了一种工具。Java的系统结构为新的面向网络的软件结构铺出了一条道路，这种软件架构充分利用了Java对于代码和对象的网络移动性的支持。

1.6 资源页

如果想获取有关本章的更多相关信息，请访问资源页：http://www.artima.com/insidejvm/resources。

第2章 平台无关

上一章展示了Java体系结构是多么适宜于开发网络环境下的软件，接下来的三章会进一步深入了解Java体系结构是如何做到这一点的。本章将详细地讨论Java的平台无关性，即Java编写的软件是如何运行于任何平台的，以及影响Java程序可移植性的各种因素是什么。同时，也讨论了为了达到这种平台无关性，Java付出了什么代价。

2.1 为什么要平台无关

Java技术在网络环境下非常有用，其中一个关键的理由就是，用Java创建的可执行二进制程序，能够不加改变地运行于多个平台。这一点在网络化环境中尤为重要，因为大多数网络通常都是由各种各样不同种类的计算机和设备互联而成。比如，网络上可能连接了艺术创作部门的Macintosh计算机、工程部门的UNIX工作站以及随处可见的运行Windows的PC。尽管这种情形下公司内部的各种计算机和设备可以共享彼此的数据，但是它仍然需要大量的管理工作。像这样一个网络，要求系统管理员必须随时维持运行于不同种类计算机上的同一个程序，在更新的时候，要根据特定于它所运行的不同平台进行版本同步更新。如果程序能够不加修改地运行于网络上的任何计算机，而不管该计算机是什么种类，那么这将极大地减轻系统管理员的工作。特别是当这样的程序是通过网络交付的时候，效果更加显著。

此外，新兴的网络化嵌入式设备则展示了Java又一擅长的领域，因为它的平台无关性在这种环境下非常有用。例如，在工作场所有各种各样的嵌入式设备，如打印机、扫描仪、传真机等，它们通常都连接到了内部网络中，像这样连接到网络的嵌入式设备，也可以出现在消费品领域，像家庭网络和汽车等等。在这个嵌入式的世界，Java的平台无关性也有助于简化系统管理任务。Jini 技术——专用于给网络带来即插即用功能的技术，就极大地减少了在网络互联的嵌入式设备环境下的管理任务，不管是对在家里的消费者还是在工作场所的系统管理员都一样。一旦某个设备加入到这样的网络中，它就能立即访问网络上的其他设备，同样其他的设备也可以访问它。为了达到如此简单易用的连接能力，采用了Jini技术的设备将通过网络彼此交换对象，要是没有Java对平台无关性的支持，这简直不可能做到。

从开发者的观点看，Java能够减少开发和在多个平台上部署应用程序的成本和时间。尽管从传统看来，大多数程序都只能支持一个平台，因为要支持多个平台的话，相对于得到的回报而言，成本过于高昂。而Java则能够戏剧性地减少支持多个平台所花的代价，对于很多种程序，这点代价是合算的。

从另一方面看，对于软件开发者来说，Java的平台无关性既有好的一面，也有不利的一面。如果正在开发和销售某个软件产品，Java的平台无关性有助于商家进入到更多的市场，而不是开发一个仅能运行于Windows的程序。比如，可以开发一个能同时运行于Windows、OS/2、Solaris以及Linux的程序。使用Java，有助于商家拥有更多的潜在客户。不利之处就在于，别人同样也

能这么做。想像一下，比如你正在集中精力为Solaris开发一个非常棒的软件，Java却让其他人更容易也开发出类似的软件，并和你在同一个目标市场中竞争。使用Java，不能只看到它带来更多潜在客户的好处，同样它也带来了更多潜在的竞争对手。

然而对于开发者来说，也许最值得注意的就是，Java程序可以不加修改地运行于多个平台的能力，这给予了网络一个同构的运行环境。这就使得新的分布式系统可以围绕着“网络移动”对象来构建。像对象序列化、RMI（远程方法调用）以及Jini这样的API就利用了这样的能力，把面向对象编程从虚拟机中带到了网络上（关于Jini的更多信息请参阅第4章）。

2.2 Java的体系结构对平台无关的支持

对平台无关性的支持，像对安全性和网络移动性的支持一样，是分布在整个Java体系结构中的，所有的组成部分——语言、class文件、API以及虚拟机，都在对平台无关性的支持方面扮演着重要角色。

2.2.1 Java平台

Java的体系结构通过几种途径支持Java程序的平台无关性，其中主要是通过Java平台自己。Java平台扮演一个运行时Java程序与其下的硬件和操作系统之间的缓冲角色。Java程序被编译为可运行于Java虚拟机中的二进制程序，并且假定Java API的class文件在运行时都是可用的。接着虚拟机运行程序，那些API则给予程序访问底层计算机资源的能力。无论Java程序被部署到何处，它只需要与Java平台交互，而不需要担心底层的硬件和操作系统。因此，它就能够运行于任何拥有Java平台的计算机。

2.2.2 Java 语言

Java 编程语言主要通过以下方式支持Java的平台无关性：它的基本数据类型的值域和行为都是由语言自己定义的。在像C或者C++这样的语言中，基本整数类型 int的值域是由它的占位宽度决定的，而它的占位宽度则由目标平台决定。一般来说，C或C++中int的占位宽度是由编译器根据目标平台的字长来决定。这就意味着针对不同平台编译的同一个C++程序在运行时可能会有不同的行为，这仅仅是因为基本数据类型在不同的平台上值域不同。然而，对于Java程序而言，不管其运行的平台是什么，Java中的 int 都是32位二进制补码表示的有符号整数，而float则总是遵守IEEE 754 浮点标准的32位浮点数。同样，这一点在Java虚拟机内部以及class文件中都是一致的。通过确保基本数据类型在所有平台上的一致性，Java语言本身为Java程序的平台无关性提供了强有力的支持。

2.3.3 Java class文件

前一章曾提到，class文件定义了一个特定于Java虚拟机的二进制格式。Java class文件可以在任何平台上创建，也可以被任何平台的Java虚拟机装入并运行。它的格式，包括多字节值的高位优先存放约定，都有严格的定义，并且是与Java虚拟机所在平台无关的。

2.2.4 可伸缩性

Java支持平台无关性，一个方面就是它的可伸缩性。Java平台可以在各种各样不同类型的（从嵌入式设备到大型主机）计算机上实现。

尽管Java目前在Web领域和桌面领域都声名卓著，但它最初却是被期望用于嵌入式设备和消

费电器领域的，而不是桌面计算机。这样的设计目标，部分原因是由于，尽管Microsoft公司和Intel公司在桌面市场占有统治地位，但是它们在嵌入式设备和消费电器市场并不具备这种优势。微处理器开始越来越多地出现音频视频装置、蜂窝电话、打印机、传真机、复印机等设备中，而这些微处理器将可以连接到网络。因而，Java最初的设计目标之一就是提供某种方式，让软件可以通过网络交付到任意种类的嵌入式设备中——不管它的微处理器和操作系统是什么。

为了达到这个目标，Java运行时系统（Java平台）不得不设计得尽量紧凑，以便它可以使用嵌入式系统中有限的资源以软件的方式来实现。嵌入式微处理器通常有一些特殊的限制，比如很少的内存，没有磁盘，没有图形化显示，甚至根本就没有显示功能。这样的限制，也就意味着嵌入式设备和消费品系统通常没有必要，或者（由于内存太少等原因）不可能支持所有的Java API。

针对嵌入式和消费性电器设备的特殊需求，Sun创建了几个具体的Java平台，它们包含更少的API。

- Java个人版平台，用于消费性电器设备。
- Java嵌入式平台，用于嵌入式系统。
- Java卡平台，用于智能卡。

这些Java平台是由Java虚拟机和比标准的Java平台更小的运行时库组成。因此，个人版平台和标准版平台的区别就在于，前者比后者提供的Java API运行时库内容更少，而嵌入式平台则比个人版平台还要少，最少的就是Java卡平台。虽然这些Java平台依次面向更小的执行环境，在资源利用上有更严格的限制，但是它们所提供的API之间并非是简单的子集关系。每一个平台提供的API子集都是面向一个特定的目标领域，因此，也包含专门针对该目标领域的API。

Java卡平台是专门面向智能卡的。除了提供的API集最少之外，它也只使用了整个Java虚拟机指令集的一部分。因而，只有那些仅仅使用了Java卡平台所支持特性的Java程序，才能在智能卡上运行。

Sun试图用这三个子集来匹配在嵌入式设备和消费性电器市场对特殊API的需求，或者说这个市场中对API的不同需求产生了这三种不同的API子集。因为嵌入式系统有特殊的限制，特别是内存很小，磁盘存储容量也很有限，嵌入式系统的生产商常常面对选择最经济的API的压力。因为嵌入式设备价低的特点，生产商常常不能容忍在设备中包含那些不直接需要的API。就算Sun定义了这三个不同的子集，生产商仍然觉得需要定义自己的API子集。

最终Sun认识到这三个子集仍然不敷所需，于是改变了它们定义嵌入式系统和消费者产品市场的API标准。Sun不再希望用一个API子集满足所有的需要（比如说用个人和嵌入式Java），而是定义了一个非常小的API指令集，称为Java 2 平台的微型版本（J2ME）。在J2ME之上，Sun计划针对不同的行业定义它们自己的子集，以适合各自的市场需要（比如，汽车电话、电视置顶盒、可视电话、无线寻呼机、手机、PDA等等）。Sun把这些API子集称为“profile”。原来的个人和嵌入式平台变成这种新方式中的一些“子集”。

因为Java平台很紧凑，它可以在很多嵌入式系统和消费性电器中实现。Java平台蕴含的紧凑性并没有把实现限制在很小的范围。Java平台可以在个人计算机、工作站和大型机上保持伸缩性。虽然在Java开始的几年里，Java虚拟机在服务器端遇到过伸缩性的困难，但是虚拟机现在已经针对服务器做过优化，很多实现可以在服务器端得到非常好的性能。在这个方面，Sun定义了一个

API超集：Java 2企业版（J2EE）。除了标准的Java API之外，J2EE包含了在企业服务环境中非常有用的一些技术，比如servlet和Enterprise JavaBean。

最终，Sun改变过的API定义方式得到了三个基础API集合，它们表现Java平台不同的伸缩性：

- 企业版（J2EE）。
- 标准版（J2SE）。
- 微型版（J2ME）。

在高端，企业版的存在表明了Java平台在高端服务的可用性。在中端，标准版提供了在浏览器中启动传统applet的功能和桌面环境下的Java平台。在低端，微型版通过不同的行业子集，显示了Java平台可以向下伸缩，并改变自己以适应完全不同的消费性电器市场和嵌入式系统需求的能力。

2.3 影响平台无关性的因素

Java的体系结构不仅促进了平台无关性软件的创建，而且还可以让人们创建平台相关的软件。当编写Java程序时，平台无关性只是一个可选的性能。

Java程序的平台无关程度依赖于多种因素，其中有些因素不在开发人员的控制范围之内，但是大多数是由开发人员来控制的。从根本上说，任何Java程序的平台无关程度都依赖于作者怎样编写它。

2.3.1 Java平台的部署

决定Java程序其平台无关性的最主要因素就是Java平台在不同的平台上被部署的程度。只有在拥有Java平台的计算机或设备上，才能运行Java程序。因此，当想要在一个特定的计算机，比如你朋友的计算机上，运行自己编写的Java程序时，必须要做两件事。首先，必须将Java平台移植到你朋友的特定类型的硬件和操作系统之上，假若某些Java平台的开发商已经完成了移植的实现，这个实现接口还必须用某种方法安装到你朋友的计算机上。因此，决定Java程序平台无关性真正程度的一个重要因素——这个因素一般不是由开发人员控制的——就是已有了可用的Java平台实现和发布版本。

然而对Java开发人员来说幸运的是，Java平台的部署因其广泛的需求性已得到推广，从Web浏览器，到桌面计算机、工作站、网络操作系统，再到许多种消费性和嵌入式设备，它们都需要Java平台。因此，你的朋友非常有可能已经在她的计算机或设备上拥有了Java平台的实现。

2.3.2 Java平台的版本

Java平台的部署有一些复杂，因为并非所有的标准运行时库在每个Java平台上都是可用的。Java平台中保证可用的基本库集合被称为标准API。Sun把Java虚拟机1.2版以及组成标准API的那些class文件称为Java 2平台标准版。这个版本的Java平台是Java API库的最小集合，例如可以在桌上型电脑和工作站上使用。但是就像前面所说的，Sun同时也定义了Java 2平台的微型版和企业版，并鼓励在各种消费电器和嵌入式设备行业开发API子集以加强微型版。另外，Sun定义了一些标准运行时库，它把这些库作为标准版的可选项，把它们称为标准扩展API。这些库包括一些如电话、商业性的服务，以及如音频、视频或3D类的多媒体服务。如果程序使用了标准扩展API中的库，它将可以在任何支持标准扩展API的地方运行，但是不能在一个只安装了最基本的

标准版平台的计算机上运行。另一方面，一些标准扩展API在企业版的任何实现中都保证可用。有了这么多种API的版本以及协议，Java 2平台就不能只代表一个同构的执行环境了——同构的执行环境可以使一段代码一次编写多处运行。

另一个复杂因素在于，在某种意义上，Java平台是一个变化的目标——它随着时间而发展。虽然Java虚拟机的发展可能非常缓慢，但Java API可能改动得非常频繁。随着时间的推移，不论是标准版还是标准扩展API，都将加入或删减某些特性，标准扩展API中的一部分可能会移植到标准版中。对于Java平台的改动，大部分都必须是向上兼容的，这就意味着它们不能破坏已有的Java程序，但是，有些改动也未必一定要这样。有些过时的特性已在Java平台的新版本中删除了，已有的、依赖那些特性的程序将不能在新版本中运行。而且，改动也可能不是向下兼容的，这意味着针对Java平台新版本而编写的程序不一定要能在老版本上运行。Java平台的动态特性在一定程度上使事情变得复杂，因为开发人员希望能够只编写一个程序而可以在任何计算机上运行。在理论上，只要程序仅仅依赖于标准API的运行时库，那么程序就应该可以在有Java 2平台标准版的所有计算机上运行。但在实际上，标准API的新版本要过一段时间才能在任何地方都适用。当程序依赖于标准API最新版本的一些新特性时，有些主机可能不能运行这个程序，因为它们只有比较老的版本。这对于软件开发人员来说不是一个新问题——例如，为Windows 95编写的程序，不能在以前的操作系统Windows 3.1中工作，但是因为Java实现了软件的网络分发，这个问题就更为尖锐了。Java的好处不仅在于它可以方便地使程序从一个平台移植到另一个平台，而且可以使同一段二进制Java代码通过网络发送，并且可以在任何计算机或设备上运行。

作为一个开发人员，你不可能控制Java平台版本的发布周期或者部署进度表，但是你可以选择自己的程序所依赖的Java平台版本。因此，在实际情况中，当发布一个新的Java平台版本时，你必须自己确定针对那个版本编写程序（时机）是否合适。

2.3.3 本地方法

除了程序所依赖的Java平台的版本之外，决定Java程序的平台无关程度的另一个主要因素就是你是否调用了本地方法。当编写一个平台独立的Java程序时，必须遵守的一条最重要的原则就是：不要直接或间接调用不属于Java API的本地方法。就像在图2-1中所示，调用Java API以外的本地方法将使程序平台相关。

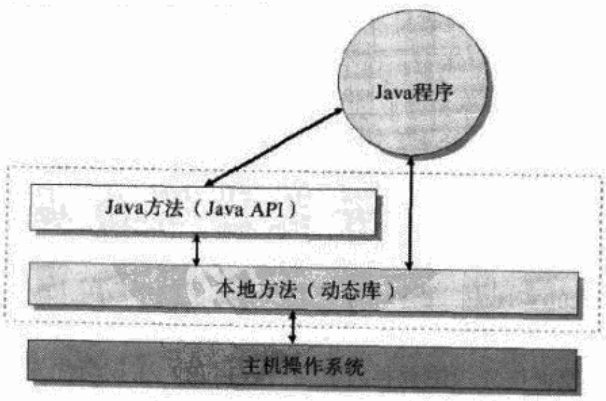


图2-1 一个平台相关的Java程序

在不需要平台无关性的情况下，直接调用本地方法是适用的。一般来说，本地方法在三种情况下适用：为了使用底层的主机平台的某个特性，而这个特性不能通过Java API访问；为了访问一个老的系统或者使用一个已有的库，而这个系统或库不是用Java编写的；为了加快程序的性能，而将一段时间敏感的代码作为本地方法实现。

如果必需使用本地方法，而且要使程序可以在多种平台上运行，那么，就必须将本地方法移植到所有需要的平台上。这种移植必须用传统的方法实现，而一旦完成了移植，必须说明怎样将这个平台相关的本地方法库分发到合适的主机。因为Java的体系结构的设计目的是简化多平台的支持，所以，编写平台独立的Java程序的最初目的是要完全禁止本地方法，并且应该仅通过Java API和主机交互。

2.3.4 非标准运行时库

本地方法本质上就和平台无关性不相容。重要的是你要调用的方法是否在“任何地方”都已实现。Java API在如Windows或Solaris等操作系统上的实现使用了本地方法来访问主机。当调用了Java API中的一个方法时，你就可以肯定它在任何地方都是可用的。而这个方法在某些地方是否是用本地方法实现的这个问题是无关紧要的。

Java平台可以由许多开发商来实现，虽然每个开发商必须提供Java API的标准运行时库，但是个别开发商还可能提供了另外的库。如果开发者侧重于平台无关性，那么就必须清楚地知道所使用的那些非标准运行时库是否调用了本地方法。没有调用本地方法的非标准库不会降低程序的平台无关性。然而如果使用了调用本地方法的运行时库，那么就会产生和直接调用本地方法一样的结果——使得程序和平台相关了。

2.3.5 对虚拟机的依赖

在编写平台独立的Java程序时，还必须遵从两条原则，这两条原则和Java虚拟机中的某些部分有关，Java虚拟机中的某些部分可以由不同的开发商用不同的方法实现。这两条原则是：

1) 不要依赖及时终结（finalization）来达到程序的正确性。

2) 不要依赖线程的优先级（thread prioritization）来达到程序的正确性。

这两条原则可以防止Java虚拟机规范中允许的垃圾收集和线程在不同实现中的变化所带来的不利影响。

所有的Java虚拟机都必须有垃圾收集堆，但是不同的实现可能使用不同的垃圾收集技术。在Java虚拟机规范中的这个灵活性意味着，在不同的虚拟机中，一个特定的Java程序中的对象可能在不同的时间被垃圾收集。这也就意味着那些在对象被释放以前由垃圾收集器运行的终结方法（finalizer），在不同的虚拟机中可能是在不同的时间运行的。如果使用了一个终结方法来释放有限的内存资源，例如文件句柄，程序就可能可以在一些虚拟机的实现上运行，而在其他实现上却不能。在一些实现上，程序可能在垃圾收集器得到机会调用释放资源的终结方法之前，就已经将有限的资源耗尽了。

在不同的Java虚拟机的实现中，另一个变化和线程的优先级有关。Java虚拟机规范只保证了，程序中所有拥有最高优先级的可运行线程将会得到一些CPU时间。这个规范也保证了在较高优先级的线程被阻塞时，较低优先级的线程将会运行。但是，在较高优先级的线程没有被阻塞的情况下，并没有禁止较低优先级的线程的运行。在某些虚拟机的实现中，即使较高优先级的线

程并未被阻塞，那些较低优先级的线程也可能先得到CPU时间。如果你的程序依赖于这个行为的正确性，它将在某些虚拟机的实现上可以正常运行，而在某些实现上却不能。为了保证多线程Java程序的平台独立，必须依赖同步（sychronization）而不是优先级来在线程之间协调相互间的动作。

2.3.6 对用户界面的依赖

在不同的Java平台的实现之间，另一个主要的变化就是用户的接口。在编写平台独立的Java程序时，用户界面是一个更为困难的问题。AWT用户界面库向用户提供了一个基本的用户界面组件集，这些组件被映射成每个平台上的本地组件。Swing库为用户提供了更高级的组件，但它们并没有映射成本地组件。用户必须利用这些基础类库建立一个接口，以使许多不同平台上的用户使用起来比较舒适，这往往不是一项简单的工作。

在不同平台上的用户已经习惯了用各自不同的方式和计算机进行交互，外部的表示不同，组件不同，在不同组件之间的交互也不同。虽然AWT和Swing库使得创建运行在不同平台上的用户界面变得比较容易，但是它们并不一定使界面设计变得方便，界面必须使不同平台上的用户使用起来都感到愉快。

2.3.7 Java平台实现中的bug

Java平台的不同实现之间还有一个变化就是其bug。虽然Sun已经开发了一套全面的测试标准，Java平台的实现必须通过这套测试，但是，可能其中某些实现在发布的时候仍然包含bug。你只能通过测试来防止这种可能性。如果有bug，可以通过测试来确定这个bug是否影响你的程序，如果确实影响，那么就必须试图找到一个绕开的途径。

2.3.8 测试

因为Java平台的实现之间可能存在差异，依赖某些特定平台写的Java程序，以及在任何特定的Java平台的实现中都可能存在bug，所以应该尽可能在所有希望运行的平台上对Java程序进行测试。Java程序的平台无关性并没有达到只需在一个平台上测试它们即可的程度。仍然需要在多个平台上测试Java程序，而且应该在程序运行的主机上尽可能找到所有的Java平台的不同实现，在这些实现上都对程序进行测试。因此，在实际情况中，在程序要运行的不同主机和不同Java平台实现上测试你的Java程序，是程序平台无关性的一个关键因素。

2.4 平台无关的七个步骤

Java的体系结构允许开发者在平台无关性和其他考虑之间进行选择。编写程序时，通过选择所使用的方法来进行你的选择。如果目的是使用那些平台相关特性（这些特性通过Java API不能访问到），来和一个老的系统进行交互，或者使用一个不是用Java编写的现有的库，或者要得到程序的最快执行速度，那么可以使用本地方法来帮助达到此目的。在这种情况下，程序的平台无关性将会降低，而这往往是可以接受的。相反，如果目的是考虑平台无关性，那么在编写程序时，应该遵从一定的规则。下面的七个步骤概述了为保证程序的最佳可移植性而可以采取的途径：

1) 选择程序要运行的主机和设备的集合（你的“目标宿主机”）。

2) 在目标宿主机中选择自认为足够好的Java平台版本，在该版本Java平台上编写、运行程序。

3) 对于每个目标宿主机，选择一些程序将要运行的Java平台实现（你的“目标运行时环境”）。

4) 编写程序，使它只通过Java API的标准运行库来访问计算机（不要调用本地方法，或者开发商专有的那些调用本地方法的库）。

5) 编写程序，使它不依赖垃圾收集器及时终结的正确性，也不依赖线程的优先级。

6) 努力设计一个用户界面，使它在你所有的目标宿主机上都能正常工作。

7) 在所有的目标运行时环境和所有的目标宿主机上测试程序。

如果遵从了以上列出的七个步骤，那么Java程序将肯定可以在你所有的目标宿主机上运行。如果目标宿主机涉及到大多数主要的Java平台的开发商和大多数主要类型的计算机，那么很有可能你的程序也能在许多其他地方运行。

如果愿意的话，也可以使程序被鉴定为“100%纯Java”的。如果正在编写一个希望具有平台无关性的程序，那么就有许多理由去这么做。例如，如果程序被鉴定为100%纯Java的，那么就可以在它上面打上100%纯Java的咖啡杯图标，还可以加入Sun的捆绑销售程序。或者，可能仅仅是把通过鉴定的过程简单地看成是对程序独立性的一个检查。在这种情况下，可以选择仅仅运行100%纯Java验证工具，这些工具可以免费下载。这些工具将报告程序的“纯度”的问题，而并不要求你通过所有的鉴定过程。

对平台无关性而言，100%纯Java鉴定不是一个充分的衡量标准。平台无关性在一定程度上就是用户的期望可以在多个平台上实现。100%纯Java测试过程并不是试图对用户的实现进行评价，它仅仅是检查确定程序是不是仅仅依赖于标准API。可以写一个Java程序，它能通过100%纯Java测试，但是仍然不能在用户期望的所有平台上正常运行。虽然如此，使代码通过100%纯Java测试过程，还是创建一个平台独立的Java程序过程中值得一试的一步。

2.5 平台无关性的策略

如图2-2所示，Java平台开发商都能以非标准的、平台相关的方式来扩展Java平台的标准组件，但是它们必须支持标准组件。今后，Sun 公司试图防止Java平台的标准组件被分裂成多个相互竞争的、轻微不兼容的系统，就像UNIX中所发生的一样。所有Java平台的开发商必须签署许可证，要求保证在Java虚拟机和Java API层的相容性，但是允许在性能和扩展等领域存在差异。就像上面提到的一样，开发商在实现线程、垃圾收集以及用户界面的外观等方面可以有一定的灵活性。如果事情像Sun计划的那样发展，Java平台的核心组件将成为所有开发商必须诚实遵守的标准，而且标准Java平台到处存在的特性将使你可以写出真正平台独立的程序。

用户可以依赖于Java平台的标准组件，因为所有Java平台的开发商都支持它们。如果编写的程序只依赖于这些组件，它就可以“到处运行”，但是可能会遇到在一定程度上的“最小公分母问题”。因为开发商可以扩展Java平台，它们可以向你提供编写平台相关程序的方法，这种程序充分利用底层的主机操作系统的特性。由于在任何Java平台的实现中既要求必需的标准组件存在，又允许开发商进行扩展，开发者就需要做出选择。这种安排允许开发人员在平台无关性和其他考虑之间进行权衡。

现在，存在着一场市场竞争，它关系着开发人员该怎样编写程序的观念——具体来说就是

该选择编写平台独立的程序还是选择编写平台相关的程序。Java带给开发人员选择的机会，进行选择的同时可能潜在地动摇整个软件工业。

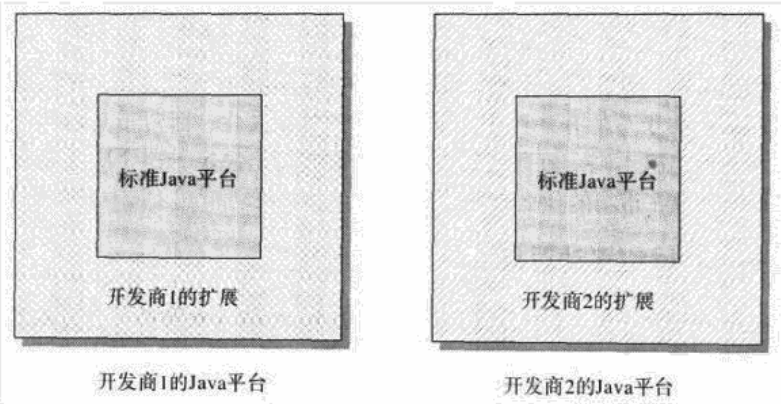


图2-2 不同开发商的Java平台的实现

Java对平台无关性的支持威胁并削弱了操作系统开发商所享有的“垄断性”。如果所有的程序都是运行在一个操作系统之上，那么，下一个计算机上很可能也会运行相同的操作系统，这样就被“垄断”在一个操作系统开发商身上，因为在软件上的投资是依赖于那个开发商的API属性的。也很可能被垄断在一个硬件体系结构之上，因为程序的二进制格式要求一种特定的微处理器。相反，如果许多程序都是用Java API编写的，并且以字节码形式存储在class文件中，那么下一次你买计算机后，就能方便地把它移植到不同的操作系统开发商了。因为Java平台可以用现有操作系统之上的软件实现，你可以变换操作系统，但不必更换所有的、老的、平台独立并基于Java的软件。

Microsoft主宰了桌面操作系统的市场，这很大程度上是因为大部分可用的软件只能运行在Microsoft操作系统之上。这是Microsoft为维持现状而采取的战略，所以他们鼓励开发人员使用Java作为一种编程语言，但编写的程序只能运行在Microsoft平台之上（译者注：这里指的是Microsoft J++，其编写的程序很难在其他平台上使用）。而其他操作系统开发商的战略目标则是削弱Microsoft对操作系统市场的垄断，所以这些开发商鼓励开发人员编写平台独立的Java程序。例如，Sun、Netscape、IBM以及许多其他开发商都联合起来支持Sun的“100%纯Java行动”，他们希望通过这个行动来培养和鼓励开发人员走平台独立的路线。

Microsoft对Java的态度是，想使Windows成为开发和运行Java程序的最好的平台。他们想不管开发人员是否选择平台独立，都让他们使用Microsoft的工具和类库。Microsoft利用为开发人员提供Java的额外特性而刮起了Java旋风，他们仍然强烈支持平台相关的Windows路线。他们强调了用Java编写可充分利用Windows平台的程序的优点。（译者注：最新的行动结果是J#，一种Java语言的Windows特定变种。）

Sun和其他支持100%纯Java行动的操作系统开发商正在试图用一些他们自己的产品抵抗Microsoft的旋风。这些公司捐献的代码和产品强调了编写平台独立的Java程序的益处。（译者注：Sun和IBM为Apache的Jarkarta项目捐献了为数不少的研究成果和代码。）

从某个层面上说，这是一场两个图标之间的战争。如果你用Microsoft的方法编写Java程序，那么将在你的产品上打上Windows图标，显示为著名的四面板样式的Windows标识。如果你走的是100%纯Java路线，那么将在你的产品上打上一个100%纯Java图标，也就是著名的冒着热气的咖啡杯标识。

作为一个开发人员，当思考怎样编写某个特定的Java程序时，软件工业的策略和宣传不一定是主要的考虑因素。对于写的有些程序，可能适合平台无关性；而对于其他的一些，则平台相关的程序可能会更有意义一些。在每种情况下，都需要做出决定，这个决定要基于用户的需要什么，以及要把自己放在市场的什么位置。

2.6 平台无关性和网络移动对象

就像本章前面提到的一样，Java技术设计的最初目的是嵌入式设备，这样做的部分原因是在现有情况下，桌上型电脑已经被Microsoft和Intel控制，而嵌入式设备则提供了最开放的市场。还有一个原因是，以嵌入式设备为目标，是因为它们将会在未来的硬件革命中——接入到宽带（通常是无线的）网络的无盘、嵌入式设备，扮演重要角色。

在Sun推出了Java第一个版本的三年后，Sun又发布了Jini。Jini试图定义一个“计算机”的体系结构，这个计算机是以新兴的嵌入式设备和消费性产品环境为代表的，这些设备联入一个到处存在的网络中。Jini的体系结构很大程度上依赖于网络移动对象。在支持Jini的设备中，对象通过网络在Java平台的实现之间传输，这些Java平台是在嵌入式设备或消费性产品、桌上型电脑以及服务器中的。那些网络移动对象所属的Java平台的实现，可能存在于各种设备和计算机硬件中，它们可能是由不同的开发商生产的。这个体系结构显著提高了平台无关性的呼声。

为了使Jini在现实世界中工作，由一个设备开发商编写的对象必须能在由其他设备开发商提供的Java平台执行环境中正常工作。在所有最终会运行的平台上测试网络移动对象几乎是不可能的，虽然在本章前面提到的平台无关性的七个步骤中推荐大家这样做。因为有这么多开发商正在生产不同种类的设备，新设备正在以不断增加的速度出现，因此要预测网络移动代码将嵌入并运行的所有地方几乎没有可能。这样，就必须使用其他的测试方法，例如对网络移动代码的一套兼容性测试。另外，为了Jini能在现实世界中工作，必须尽可能最大程度地实现执行环境的同构性。最后，当程序员编写网络移动代码及程序时，很可能需要考虑执行环境不同的可能性。

2.7 资源页

要了解更多关于Java和平台无关性的信息，可以访问本章的资源页：http://www.artima.com/insidejvm/resources 。

第3章 安　　全

除了前一章讨论的平台独立，面向网络的软件技术还必须解决的另一个技术难题就是安全。因为网络允许多台计算机共享数据和分布式处理，所以它提供了一条侵入计算机系统的潜在途径，使得其他人可能窃取信息，改变或破坏信息，盗取计算资源等等。因此，将计算机连入网络产生了很多安全问题。

为了解决由网络引起的安全问题，Java体系结构采用了一个扩展的内置安全模型，这个模型随着Java平台的主要版本而不断发展。本章对Java核心体系结构中建立的安全模型作了概述，并且介绍了它的发展历程。

3.1 为什么需要安全性

Java的安全模型是其多个重要结构特点之一，它使Java成为适于网络环境的技术。因为网络提供了一条攻击连入的计算机的潜在途径，因此安全性是非常重要的。如果在一个环境中，软件可以通过网络下载并在本地运行，这个问题尤其严重。例如Java applet和Jini服务对象就是这样的例子。因为当用户在浏览器中打开网页时，applet的class文件被自动下载，很有可能用户会遇到来自不可靠来源的applet。同样，当一个Jini服务对象用Jini查找服务进行服务注册时，它的class文件将从服务供应商指定的代码库进行下载。Jini实现了一个自发的网络互联，在这个网络中，客户机进入一个新的环境查找并访问本地可用服务，因此，Jini服务的客户机可能会遇到来自不可靠来源的服务对象。如果没有任何安全机制，这些代码自动下载的模式为恶意代码的发布提供了便捷的途径。Java的安全机制使得Java适合于网络，因为它们建立了对网络移动代码安全执行的必要的可信机制。

Java安全模型侧重于保护终端用户免受从网络下载的、来自不可靠来源的、恶意程序（以及善意程序中的bug）的侵犯。为了达到这个目的，Java提供了一个用户可配置的“沙箱”，在沙箱中可以放置不可靠的Java程序。沙箱对不可靠程序的活动进行了限制，程序可以在沙箱的安全边界内做任何事，但是不能进行任何跨越这些边界的举动。例如，原来在版本1.0中的沙箱对很多不可靠Java applet的活动做了限制，包括：

- 对本地硬盘的读写操作。
- 进行任何网络连接，但不能连接到提供这个applet的源主机。
- 创建新的进程。
- 装载新的动态连接库。

由于下载的代码不可能进行这些特定的操作，这就使Java安全模型可以保护终端用户免受恶意或有漏洞的代码的威胁。因为沙箱安全模型对不可靠代码能做什么、不能做什么进行了严格的控制，所以用户可以相对安全地运行不可靠代码。但是，对于1.0系统的程序员和用户来说，这个最初的沙箱限制太过严格，善意（但不可靠）的代码常常无法进行有效的工作。在版本1.1

中，最初的沙箱模型得到了改进，引入了基于代码签名和认证的信任模式。签名和认证使得接收端系统可以确认一系列class文件（在一个JAR文件中）已经由某一实体进行了数字签名（有效，可被信赖），并且在签名过后，这些class文件没有改动。这就使得终端用户和系统管理员减少了对某些代码在沙箱中的限制，但这些代码必须已由可信任团体进行数字签名。

虽然版本1.1中发布的安全API包含了对认证的支持，但实际上，除了提供完全信任和完全不信任策略（换句话说，代码要么被完全信任，要么完全不被信任）以外，它们没有提供许多实际的帮助。Java的下一个主要版本——版本1.2中，提供的API可以帮助建立细粒度的安全策略，这种策略是建立在数字签名代码的认证基础上的。本章后面以Java安全模型的发展为主线，介绍1.0版本的基本沙箱，然后是1.1版本的代码签名和认证，最后是1.2版本的细粒度访问控制。

3.2 基本沙箱

在个人电脑世界中，一般来说，在运行一个软件前你必须信任它，用户只能通过小心地使用来自可信任来源的软件来达到安全性，并且定期扫描、检查病毒来确保安全性。一旦某个软件有权使用你的系统，它将拥有完全权限。如果这个软件是恶意的，它就可以大肆进行破坏，因为你的计算机中的运行环境没有对它们实施任何限制。所以，在传统的安全模式中，甚至必须想办法防止恶意代码有权使用你的计算机。

沙箱安全模型使得工作变得容易，即使某个软件来自你不能完全信任的地方。沙箱模型使你可以接收来自任何来源的代码，而不是要求用户避免将来自不信认站点的代码下载到机器上。但是当来自不可靠来源的代码运行时，沙箱限制它进行可能破坏系统的任何动作。不必指出哪些代码可以信任，哪些代码不可以信任，也不必扫描查找病毒，沙箱本身限制了下载的任何病毒或其他恶意的、有漏洞的代码，使得它们不能对计算机进行破坏。

如果你还有疑问，在确信它能保护你之前，用户必需确认沙箱没有任何漏洞。为了保证沙箱没有漏洞，Java安全模型对其体系结构的各方面都进行了考虑。如果在Java体系结构中有任何没有考虑到安全的区域，恶意的程序员（“黑客”）很可能会利用这些区域来绕开沙箱。因此，为了对沙箱有一个了解，我们必须先看一下Java体系结构的几个不同部分，并且理解它们是怎样一起工作的。

组成Java沙箱的基本组件如下：

- 类装载器结构。
- class文件检验器。
- 内置于Java虚拟机（及语言）的安全特性。
- 安全管理器及Java API。

Java的沙箱安全模型，最重要的优点之一就是这些组件中的类装载器和安全管理器是可以由用户定制的。通过定制这些组件，可以为Java程序创建个性化的安全策略。但是，这种可定制性是需要代价的，因为这种体系结构的灵活性也对它本身产生一定的风险。类装载器和安全管理器非常复杂，因此，单纯的定制操作也可能潜在地产生错误，从而开启安全漏洞。

在Java API的每一个主要版本中，都进行了一些改进，使得创建定制的安全策略时出错机会更少。最重要的改变是在版本1.2中，引入了一个新的且更为精细的访问控制体系结构。在版本

1.0和1.1中，访问控制包括安全策略规范和运行时安全策略的实施，它是由称作安全管理器的对象负责的。要在版本1.0和1.1中建立定制的策略，必须编写自己的定制安全管理器。在版本1.2中，可以利用Java 2平台提供的安全管理器，这个预先制作好的安全管理器，允许用户在一个和程序分离的ASCII策略文件中说明安全策略。在运行时，这个预先制作好的安全管理器获得一个类（称为访问控制器）的帮助，来执行在策略文件中说明的安全策略。在版本1.2中引入的访问控制基础架构提供了灵活易定制的安全管理器的默认实现，这个默认实现可以满足大多数的安全需要。为了向后兼容，也为了使一些有特殊安全需要的团体可以对预先制作好的安全管理器中的默认功能进行改写，版本1.2的应用程序还可以安装它们自己的安全管理器。是使用预先制作的安全管理器，还是使用在它之上进行扩展的访问控制基础架构，用户是可以选择的。

3.3 类装载器体系结构

在Java沙箱中，类装载器体系结构是第一道防线。毕竟，是由类装载器将代码——这个代码可能是恶意的或是有漏洞的——装入Java虚拟机中。类装载器体系结构在三个方面对Java的沙箱起作用：

1) 它防止恶意代码去干涉善意的代码。

2) 它守护了被信任的类库的边界。

3) 它将代码归入某类（称为保护域），该类确定了代码可以进行哪些操作。

类装载器体系结构可以防止恶意的代码去干涉善意的代码，这是通过为由不同的类装载器装入的类提供不同的命名空间来实现的。命名空间由一系列惟一的名称组成，每一个被装载的类有一个名字，这个命名空间是由Java虚拟机为每一个类装载器维护的。例如，一旦Java虚拟机将一个名为Volcano的类装入一个特定的命名空间，它就不能再装载名为Volcano的其他类到相同的命名空间了。可以把多个Volcano类装入一个Java虚拟机中，因为可以通过创建多个类装载器在一个Java应用程序中创建多个命名空间。如果在一个运行的Java应用程序中创建了三个独立的命名空间（为三个类装载器中的每一个都创建一个命名空间），就可以通过为每个命名空间装载一个Volcano类，从而满足将三个不同的Volcano类装载到应用程序中的要求。

命名空间有助于安全的实现，因为你可以有效地在装入了不同命名空间的类之间设置一个防护罩。在Java虚拟机中，在同一个命名空间内的类可以直接进行交互，而不同的命名空间中的类甚至不能察觉彼此的存在，除非显式地提供了允许它们进行交互的机制。一旦加载后，如果一个恶意的类被赋予权限访问其他虚拟机加载的当前类，它就可以潜在地知道一些它不应该知道的信息，或者干扰程序的正常运行。

图3-1显示了和两个类装载器有关的命名空间，它们都装载了一个名为Volcano的类。命名空间的每一个命名都被关联到方法区中的一个类型数据，这个类型数据用那个名字定义了一个类型。图3-1中画出了从命名空间中的名字指向方法区中的类型的箭头，方法区定义了类型。左边深灰色的类装载器装载了两个深灰色的类型，名字为Climber和Volcano；右边浅灰色的类装载器装载了两个浅灰色的类型，名字为BakingSoda和Volcano。基于命名空间的特性，当类Climber引用类Volcano时，它指向的是深灰色的Volcano，这个Volcano和它装载在同一个命名空间内。Climber甚至并不知道另一个Volcano的存在，即使它们位于同一个虚拟机中。如果读者想要了解

类装载器体系结构是怎样实现命名空间的独立性的，请参见第8章。

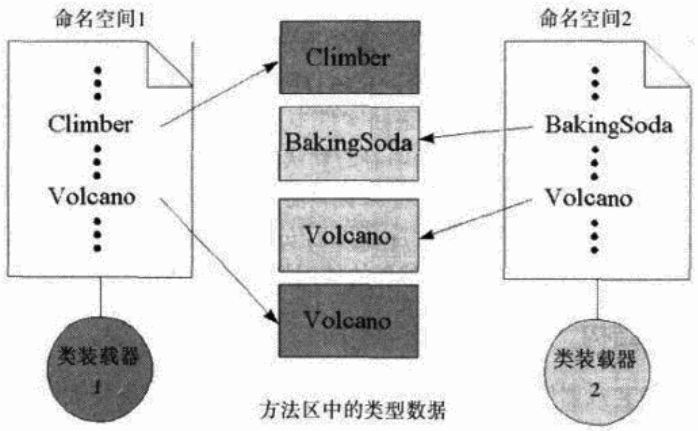


图3-1 类装载器和命名空间

类装载器体系结构守护了被信任的类库的边界，这是通过分别使用不同的类装载器装载可靠的包和不可靠的包来实现的。虽然通过赋给成员受保护（或包访问）的访问限制，可以在同一个包中的类型间授予彼此访问的特殊权限，但这种特殊的权限只能授给在同一个包中的运行时成员，而且它们必须是由同一个类装载器装载的。

用户自定义类装载器经常依赖其他类装载器——至少依赖于虚拟机启动时创建的启动类装载器——来帮助它实现一些类装载请求。在版本1.2以前，非启动类装载器必须显式地求助于其他类装载器，类装载器可以请求另一个用户自定义的类装载器来装载一个类，这个请求是通过对被请求的用户自定义类装载器调用loadClass（）来实现的。除此以外，类装载器也可以通过调用findSystemClass（）来请求启动类装载器来装载类，这是类ClassLoader中的一个静态方法。在版本1.2中，类装载器请求另一个类装载器来装载类型的过程被形式化，称为双亲委派模式。从版本1.2开始，除启动类装载器以外的每一个类装载器，都有一个“双亲”类装载器，在某个特定的类装载器试图以常用方式装载类型以前，它会先默认地将这个任务“委派”给它的双亲——请求它的双亲来装载这个类型。这个双亲再依次请求它自己的双亲来装载这个类型。这个委派的过程一直向上继续，直到达到启动类装载器，通常启动类装载器是委派链中的最后一个类装载器。如果一个类装载器的双亲类装载器有能力来装载这个类型，则这个类装载器返回这个类型。否则，这个类装载器试图自己来装载这个类型。

在版本1.2以前的大多数虚拟机的实现中，内置的类装载器（以后将称为原始类装载器）负责在本地装载可用的class文件。这些class文件通常包括那些要运行的Java应用程序的class文件，以及这个应用程序所需要的任何类库，这些类库中包含Java API的基本class文件。虽然如何找到这些被请求类型的class文件属于实现细节，但是许多实现都是按照类路径（class path）指明的顺序查找目录和JAR文件的。

在版本1.2中，装载本地可用的class文件的工作被分配到多个类装载器中。刚才称为原始类装载器的内置的类装载器被重新命名为启动类装载器，表示它现在只负责装载那些核心Java API

的class文件。因为核心Java API的class文件是用于“启动”Java虚拟机的class文件，所以，启动类装载器的名字也因此而得。

在版本1.2中，由用户自定义类装载器来负责其他class文件的装载，例如用于应用程序运行的class文件，用于安装或下载标准扩展的class文件，在类路径中发现的类库的class文件，等等。当1.2版本的Java虚拟机开始运行时，在应用程序启动以前，它至少创建一个用户自定义类装载器，也可能创建多个。所有这些类装载器被连接在一个双亲-孩子的关系链中，在这条链的顶端是启动类装载器，在这条链的末端是一个在版本1.2中被称为“系统类装载器”的类装载器。在版本1.2以前，“系统类装载器”这个名字有时指内置的类装载器，它也被称作原始类装载器。在版本1.2中，系统类装载器这个名字有了更正式的定义，它是指由Java应用程序创建的、新的用户定义类装载器的默认委派双亲。这个默认的委派双亲通常是一个用户自定义的类装载器（译者注：被称为用户自定义的类装载器，这是和启动类装载器相对而言的，实际上它是由Java虚拟机的实现提供的），它装载应用程序的初始类，但是它也可能是任何用户自定义类装载器，这是由实现Java平台的设计者决定的。

例如，假设你写了一个Java应用程序，装载了一个类装载器，这个装载器是通过网络下载来装载class文件的。设想你在虚拟机上运行这个应用程序，在启动时实例化了两个用户自定义类装载器：一个“已安装扩展”的类装载器，一个“类路径”类装载器。这些类装载器和启动类装载器一起联入一个双亲-孩子关系链中，如图3-2所示。类路径的类装载器的双亲是已安装了扩展的类装载器，而它的双亲是启动类装载器。在图3-2中，类路径类装载器被设计成系统类装载器，新的用户自定义类装载器的默认委派双亲被应用程序实例化。假设当应用程序实例化它的网络类装载器时，它指明了系统类装载器作为它的双亲。

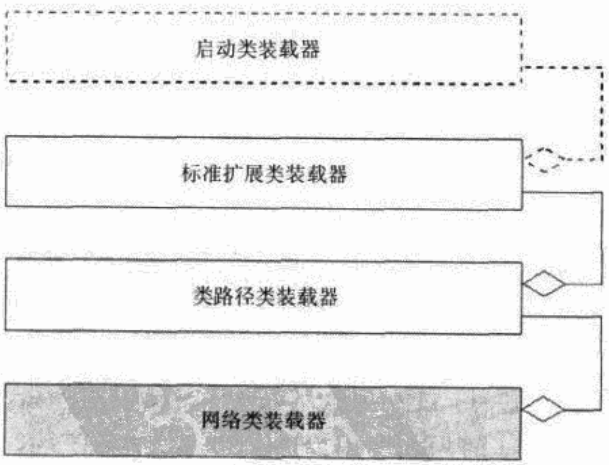


图3-2　双亲-孩子类装载器委派链

设想在运行Java应用程序的过程中，类装载器发出一个装载Volcano类的请求，类装载器必须先询问它的双亲——类路径类装载器——来查找并装载这个类。这个类路径类装载器依次将向它自己的双亲发出同样的请求，它的双亲即为已安装扩展的类装载器。这个类装载器也是首先将这个请求委派给它自己的双亲——启动类装载器。假设Volcano类不是Java API的一部分，

也不是一个已安装扩展的一部分，也不在类路径上，所有这些类装载器将返回而不会提供一个名为Volcano的已装载类。当类路径类装载器回答，它和所有它的双亲都不能装载这个类时，你的类装载器可能将试图用它自己特定的方式来装载Volcano类，它会通过网络下载Volcano。假设你的类装载器可以下载Volcano类，这样Volcano类就可以在应用程序以后的执行过程中发挥作用。我们继续讨论这个例子，假设以后的某一时刻Volcano类的一个方法首次被调用，并且那个方法引用了Java API中的类java.util.HashMap，因为这个引用是首次被运行的程序使用，所以虚拟机会请求你的类装载器（装载Volcano的类装载器）来装载java.util.HashMap。就像以前一样，你的类装载器首先将请求传递给它的双亲类装载器，然后这个请求一路委派直到委派给启动类装载器。但是这一次，启动类装载器可以将java.util.HashMap类返回给你的类装载器，因为启动类装载器可以找到这个类，所以已安装扩展的类装载器就不必在已安装扩展中查找这个类型，类路径类装载器也不必在类路径中查找这个类型，同样，你的类装载器也不必从网上下载这个类。所有这些类装载器仅需要返回由启动类装载器返回的类java.uthil.HashMap。从这一时刻开始，不管何时Volcano类引用名为java.util.HashMap的类，虚拟机就可以直接使用这个java.util.HashMap类了。

在给出类装载器怎样工作的背景后，读者可能想知道如何使用类装载器来保护可信任类库。类装载器的体系结构是通过剔除装作被信任的不可靠类，来保护可信任类库的边界的。如果某个恶意的类可以成功地欺骗Java虚拟机，使Java虚拟机相信它是一个来自Java API的可信任类，那么，这个恶意的类可能突破沙箱的阻隔。为了防止不可靠的类扮演被信任的类，类装载器的体系结构阻塞了危及Java运行时安全的潜在途径。

在有双亲委派模式的情况下，启动类装载器可以抢在标准扩展类装载器之前去装载类，而标准扩展类装载器可以抢在类路径类装载器之前去装载那个类，类路径类装载器又可以抢在网络类装载器之前去装载它。这样，在使用双亲-孩子委派链的方法中，启动类装载器会在最可信的类库——核心Java API——中首先检查每个被装载的类型，然后，才依次到标准扩展、类路径上的本地类文件中检查。所以，如果网络类装载器装载的某段移动代码发出指示，试图通过网络下载一个和Java API中某个类型同名的类型，例如java.lang.Integer，它将不能成功。如果java.lang.Integer的class文件在Java API中已存在，它将被启动类装载器抢先装载，而网络类装载器将没有机会下载并定义名为java.lang.Integer的类，它只能使用由它的双亲返回的类，这个类应该是由启动类装载器装载的。用这种方法，类装载器的体系结构就可以防止不可靠的代码用它们自己的版本来替代可信任的类。

但是如果这个移动代码不是去试图替换一个被信任的类，而是想在一个被信任的包中插入一个全新的类型，情况会怎样呢？想像一下，如果刚才那个例子中，要求网络类装载器装载一个名为java.lang.Virus（病毒）的类时，将会发生什么？像以前一样，这个请求将首先被一路向上委派给启动类装载器，虽然这个启动类装载器负责装载核心Java API的class文件，包括名为java.lang的包，但它无法在包java.lang中找到名为Virus的成员。假设这个类在已安装扩展以及本地类路径中也找不到，你的类装载器将试图从网络上下载这个类。

假设你的类装载器成功地下载并定义了这个名为java.lang.Virus的类，Java允许在同一个包中的类拥有彼此访问的特殊权限，而这个包外的类则没有这个权限。所以，因为你的类装载器

装载了一个名为java.lang.Virus的类，暗示这个类是Java API的一部分，可想而知，它将得到访问java.lang中被信任类的特殊访问权限，并且可以使用这个特殊的访问权限达到不可告人的目的。类装载器机制可以防止这个代码得到访问java.lang包中被信任类的访问权限，因为Java虚拟机只把彼此访问的特殊权限授予由同一个类装载器装载到同一个包中的类型。因为Java API的java.lang包中的被信任类是由启动类装载器装载的，而恶意的类java.lang.Virus是由网络类装载器装载的，所以这些类型不属于同一个运行时包。运行时包这个名词，是在Java虚拟机第2版规范中第一次出现的，它指由同一个类装载器装载的、属于同一个包的、多个类型的集合。在允许两个类型之间对包内可见的成员（声明为受保护的或包访问的成员）进行访问前，虚拟机不但要确定这两个类型属于同一个包，还必须确认它们属于同一个运行时包——它们必须是由同一个类装载器装载的。这样，因为java.lang.Virus和来自核心Java API的java.lang的成员不属于同一个运行时包，java.lang.Virus就不能访问Java API的java.lang包中的类型和包内可见的成员。

之所以提出运行时包的概念，动机之一是使用不同的类装载器装载不同的类。启动类装载器装载核心Java API的class文件，这些class文件是最可信的。已安装扩展的类装载器装载来自于任何已安装扩展的class文件，已安装扩展是非常可信的，但这个可信度是在一定程度上的可信度，它们不能简单地通过将新类型插入到Java API的包中来获得对包内可见成员的访问权，这是因为已安装扩展是由不同于核心API的类装载器装载的。同样，由类路径类装载器在类路径中发现的代码不能访问已安装扩展或Java API中的包内可见成员。

类装载器可以用另一种方法来保护被信任的类库的边界，它只需通过简单地拒绝装载特定的禁止类型就可以了。例如，你可能已经安装了一些包，这些包中包含了应用程序需要装载的类，这些类必须是由网络类装载器的双亲——类路径类装载器——装载的，而不是由网络类装载器装载的。假设已经创建了一个名为absolutepower的包，并且将它安装在了本地类路径中的某个地方，在这里它可以被类路径类装载器访问到。而且假设你不想让由自己的类装载器装载的类，能装载来自absolutepower包中的任何类。在这种情况下，必须编写自己的类装载器，让它做的第一件事就是确认被请求的类不是absolutepower包中的一个成员。如果这样的类被请求装载，你的类装载器将抛出一个安全异常，而不是将这个类的名字传给双亲类装载器。

类装载器要知道一个类是否来源于一个被禁止的包，例如absolutepower，只有一个方法，就是通过它的类名来检测。因为类名absolutepower.FancyClassLoader指明了它是包absolutepower的一部分，而包absolutepower被列在被禁止的包列表中，所以你的类装载器应该立即抛出一个安全异常。

除了屏蔽不同命名空间中的类以及保护被信任的类库的边界外，类装载器还起到另外的安全作用。类装载器必须将每一个被装载的类放置在一个保护域中，一个保护域定义了这个代码在运行时将得到怎样的权限。这是类装载器的一个非常重要的安全工作，关于它的更多信息将稍后在本章给出。

3.4 class文件检验器

和类装载器一起，class文件检验器保证装载的class文件内容有正确的内部结构，并且这些class文件相互间协调一致。如果class文件检验器在class文件中发现了问题，它将抛出异常。好

的Java编译器不应该产生畸形的class文件，但是Java虚拟机并不知道某个特定的class文件是如何被创建的。因为一个class文件实质上是一个字节序列，所以虚拟机无法分辨特定的class文件是由正常的Java编译器产生的，还是由黑客特制的（黑客可能威胁虚拟机的完整性）。所以，所有的Java虚拟机的实现必须有一个class文件检验器，文件检验器可以调用class文件以确保这些定义的类型可以安全地使用。

class文件检验器实现的安全目标之一就是程序的健壮性。如果某个有漏洞的编译器，或是某个聪明的黑客，产生了一个class文件，而这个class文件中包含了一个方法，这个方法的字节码中含有一条跳转到方法之外的指令，那么，一旦这个方法被调用，它将导致虚拟机的崩溃。所以，出于对健壮性的考虑，由虚拟机检验它装载的字节码的完整性是非常重要的。

Java虚拟机的class文件检验器在字节码执行之前，必须完成大部分检验工作。它只在执行前而不是在执行中对字节码进行一次分析（并检验它的完整性），每一次遇到一个跳转指令时都进行检验。作为字节码确认工作的一部分，虚拟机将确认所有的跳转指令会到达另一条合法的指令，而且这条指令是在这个方法的字节码流中的。在大多数情况下，在执行前就对所有字节码进行一次检查，对于保证健壮性来说就足够了，而不必在它运行时每次都检验每一条字节码指令。

class文件检验器要进行四趟独立的扫描来完成它的操作。第一趟扫描是在类被装载时进行的，在这次扫描中，class文件检验器检查这个class文件的内部结构，以保证它可以被安全地编译。第二和第三趟扫描是在连接过程中进行的，在这两次扫描中，class文件检验器确认类型数据遵从Java编程语言的语义，包括检验它所包含的所有字节码的完整性。第四趟扫描是在进行动态连接的过程中解析符号引用时进行的，在这次扫描中，class文件检验器确认被引用的类、字段以及方法确实存在。

3.4.1 第一趟：class文件的结构检查

在第一趟扫描中，对每一段将被当作类型导入的字节序列，class文件检验器都会确认它是否符合Java class文件的基本结构。在这次扫描中，检验器将进行许多检查，例如每个class文件必须以四个同样的字节开始：魔数0xCAFEBABE。这个魔数的用处是让class文件分析器很容易分辨出某个文件有明显问题而加以拒绝，这个文件可能是被破坏了的class文件，或者是压根儿就不是class文件。这样，class文件检验器所做的第一件事极可能就是检查导入的文件是否是以0xCAFEBABE开头的。检验器还必须确认在class文件中声明的主版本号和次版本号，这个版本号必须在这个Java虚拟机实现可以支持的范围之内。

而且，在第一趟扫描中，class文件检验器必须检验确认这个class文件没有被删节，尾部也没有附带其他的字节。虽然不同的class文件有不同的长度，但是在class文件中包含的每一个组成部分都声明了它的长度和类型。检验器可以使用组成部分的类型和长度来确定整个class文件的正确的总长度。用这种方法，它就可以检查一个装入的文件，其长度是否和它里面的内容相一致。

第一趟扫描的主要目的就是保证这个字节序列正确地定义了一个新类型，它必须遵从Java的class文件的固定格式，这样它才能被编译成在方法区中的（基于实现的）内部数据结构。第二、第三和第四趟扫描不是在符合class文件格式的二进制数据上进行的，而是在方法区中的、由实现决定的数据结构上进行的。

3.4.2 第二趟：类型数据的语义检查

在第二趟扫描中，class文件检验器进行的检查不需要查看字节码，也不需要查看和装载任何其他类型。在这趟扫描中，检验器查看每个组成部分，确认它们是否是其所属类型的实例，它们的结构是否正确。例如，方法描述符（它的返回类型，以及参数的类型和个数）在class文件中被存储为一个字符串，这个字符串必须符合特定的上下文无关文法。检验器对每个组成部分进行检查的目的之一是，为了确认每个方法描述符都是符合特定语法的、格式正确的字符串。

另外，class文件检验器检查这个类本身是否符合特定的条件，它们是由Java编程语言规定的。例如，检验器强制规定除Object类以外的所有类，都必须有一个超类。在第二趟扫描中，检验器还要检查final（最终的）类没有被子类化，而且final方法没有被覆盖。还要检查常量池中的条目是合法的，而且常量池的所有索引必须指向正确类型的常量池条目。也就是说，class文件检验器在运行时检查了一些Java语言应该在编译时遵守的强制规则。因为检验器并不能确定class文件是否是由一个善意的、没有漏洞的编译器产生的，所以它会检查每个class文件，以确保这些规则得到遵守。

3.4.3 第三趟：字节码验证

在class文件检验器成功地进行了两趟检查后，它将把注意力放在字节码上，这一趟扫描被称为“字节码验证”。在这趟扫描中，Java虚拟机对字节流进行数据流分析，这些字节流代表的是类的方法。为了理解字节码检验器，必须对字节码和栈帧有一定的了解。

字节码流代表了Java的方法，它是由被称为操作码的单字节指令组成的序列，每一个操作码后都跟着一个或多个操作数。操作数用于在Java虚拟机执行操作码指令时提供所需的额外的数据。执行字节码时，依次执行每个操作码，这就在Java虚拟机内构成了执行的线程。每一个线程被授予自己的Java栈，这个栈是由不同的栈帧构成的。每一个方法调用将获得一个自己的栈帧——栈帧其实就是一个内存片断，其中存储着局部变量和计算的中间结果。在栈帧中，用于存储方法的中间结果的部分被称为该方法的操作数栈。操作码和它的（可选的）操作数可能指存储在操作数栈中的数据，或存储在方法栈帧中的局部变量中的数据。这样，在执行一个操作码时，除了可以使用紧随其后的操作数，虚拟机还可以使用操作数栈中的数据，或局部变量中的数据，或是两者都用。

字节码检验器要进行大量的检查，以确保采用任何路径在字节码流中都得到一个确定的操作码，确保操作数栈总是包含正确的数值以及正确的类型。它必须保证局部变量在赋予合适的值以前不能被访问，而且类的字段中必须总是被赋予正确类型的值，类的方法被调用时总是传递正确数值和类型的参数。字节码检验器还必须保证每一个操作码都是合法的，即每一个操作码都有合法的操作数，以及对每一个操作码，合适类型的数值位于局部变量中或是在操作数栈中。这些仅仅是字节码检验器所做的大量检验工作中的一小部分，在整个检验过程通过后，它就能保证这个字节码流可以被Java虚拟机安全地执行。

字节码检验器并不试图检测出所有的安全问题。如果要这样做的话，它将会遇到“停机问题”。停机问题是计算机科学领域的一个著名论题：即不可能写出一个程序，用它来判断作为其输入而读入的某个程序在执行时是否停机。一个程序是否会停机被称为是程序的“不可判定”特性，因为不可能写出一个程序，让它100%地告诉你任何一个给定的程序是否含有这种特性。

停机问题的不可判定性可以扩展成计算机程序的许多特性，如一个Java字节码的集合是否能被虚拟机安全地执行。

字节码检验器处理停机问题的方法是，不去试图精确地让每个安全的程序都通过检查。虽然不能写出一个程序来判定任何给定程序是否会停机，但是可以写出一个简单的程序，让它只是识别出某些一定会停机的程序。例如，如果一个程序的第一条指令就是停机，那么，这个程序一定可以停机。如果一个程序内没有循环，它也一定可以停机，如此等等。同样，虽然不可能写出一个能扫描检查所有字节码流是否能被虚拟机安全执行的检验器，但是可以写出一个能让其中一部分安全的字节码流通过的检验器。Java的字节码检验器恰恰就是这么做的。这个检验器检查确认读入的每一个字节码集合是否符合一个特定的规则集合。如果一个字节码集合能够遵从所有这些规则，那么检验器就知道它可以被虚拟机安全地执行。如果不是，那么，这些字节码可能可以被虚拟机安全地执行，也可能不能安全地执行。这样，通过识别一些安全的字节码流，但不是全部，检验器就绕过了停机问题。由于字节码检验器强制检查的特性，只要定义好规则，任何程序只要可以用Java编程语言书写，编译器就可以确保编译出来的字节码可以被检验器通过。有些程序虽然不能用Java编程语言源代码表达出来，但仍可以通过检验器的检验。另外还有些程序（也不能用Java源代码表示），它们虽然实际上也能被Java虚拟机安全地执行，却不能通过检验器的检验。

在第一、第二、第三趟扫描中，class文件检验器可以保证导入的class文件构成合理，内在一致，符合Java编程语言的限制条件，并且包含的字节码可以被Java虚拟机安全地执行。如果class文件检验器发现其中任何一点不正确，它将会抛出一个错误，这个class文件将不会被程序使用。

3.4.4 第四趟：符号引用的验证

在动态连接的过程中，如果包含在一个class文件中的符号引用被解析时，class文件检验器将进行第四趟检查。在这趟检查中，Java虚拟机将追踪那些引用——从被验证的class文件到被引用的class文件，以确保这个引用是正确的。因为第四趟扫描必须检查被检测的class文件以外的其他类，所以这次扫描可能需要装载新的类。大多数Java虚拟机的实现采用延迟装载类的策略，直到类真正地被程序使用时才装载。即使一个实现确实预先装载了这些类，这是为了加快装载过程的速度，那它还是会表现为延迟装载。例如，如果Java虚拟机在预先装载中发现它不能找到某个特定的被引用类，它并不在当时抛出NoClassDefFoundError错误，而是直到（或者除非）这个被引用类首次被运行程序使用时才抛出。这样，如果Java虚拟机进行预先连接，第四趟扫描可以紧随第三趟扫描发生。但是如果Java虚拟机在某个符号引用第一次被使用时才进行解析，那么第四趟扫描将在第三趟扫描以后很久、当字节码被执行时才进行。

class文件检验器的第四趟扫描仅仅是动态连接过程的一部分。当一个class文件被装载时，它包含了对其他类的符号引用以及它们的字段和方法。一个符号引用是一个字符串，它给出了名字，并且可能还包含了其他关于这个被引用项的信息——这些信息必须足以惟一地识别一个类、字段或方法。这样，对于其他类的符号引用必须给出这个类的全名；对于其他类的字段的符号引用必须给出类名、字段名以及字段描述符；对于其他类中的方法的引用必须给出类名、方法名以及方法的描述符。

动态连接是一个将符号引用解析为直接引用的过程。当Java虚拟机执行字节码时，如果它遇到一个操作码，这个操作码第一次使用一个指向另一个类的符号引用，那么虚拟机就必须解析这个符号引用。在解析时，虚拟机执行两个基本任务：

1) 查找被引用的类（如果必要的话就装载它）。

2) 将符号引用替换为直接引用，例如一个指向类、字段或方法的指针或偏移量。

虚拟机必须记住这个直接引用，这样当它以后再次遇到相同的引用时，它就可以立即使用这个直接引用，而不必花时间再次解析这个符号引用了。

当Java虚拟机解析一个符号引用时，class文件检验器的第四趟扫描确保了这个引用是合法的。当这个引用是个非法引用时——例如，这个类不能被装载，或这个类的确存在，但是不包含被引用的字段或方法——class文件检验器将抛出一个错误。

再以Volcano类为例。如果Volcano类中的某个方法调用了名为Lava的类中的某个方法，这个Lava中的方法的全名和描述符将包含在Volcano的class文件的二进制数据中。当Volcano的方法在执行过程中第一次调用Lava的方法时，Java虚拟机必须确认类Lava中存在这个方法，并且这个方法的名字和描述符与Volcano类中期待的相匹配。如果这个符号引用（类名、方法名和描述符）是正确的，那么，虚拟机将把它替换为一个直接引用，例如一个指针，从那时开始将使用这个指针。但如果Volcano类中的符号引用不能匹配Lava类中的任何方法时，第四趟扫描验证失败，Java虚拟机将抛出一个NoSuchMethodError。

3.4.5 二进制兼容

正因为Java程序是动态连接的，所以class文件检验器在第四次扫描中，必须检查相互引用的类之间是否兼容。如果你修改了一个类，Java编译器常会重编译这些类，从而在编译时检测是否有任何的不兼容性。但是也有很多时候，编译器不对受影响的类进行重编译，例如，如果正在开发一个大型系统，很可能将系统分割成几个部分放入包中。如果对每个包进行独立的编译，当改动包中的一个类时，可能将导致同一个包内受影响的那些类需要重编译，但是对于其他包来说就不需要进行重编译了。此外，如果使用了其他人的一些包，尤其是程序在运行时通过网络下载了一些类，就不可能在编译时检验兼容性。这就是为什么class文件检验器在第四趟扫描时，必须在运行时检查兼容性的原因。

举一个修改不兼容的例子。假设你用一个Java编译器编译了Volcano类（见以前的例子），因为Volcano中的一个方法调用了另一个类Lava中的方法，Java编译器将查找类Lava的class文件或源文件，以确认Lava中是否有一个方法具有这个名字、返回类型以及相同数量和类型的参数。如果编译器不能找到任何名为Lava的类，或者找到一个Lava类，但是这个类中不包含想要找的方法，那么，编译器将产生一个错误，并且将不为Volcano类生成class文件。否则，Java编译器将为Volcano产生一个class文件，这个class文件和Lava的class文件相兼容。这样，Java编译器拒绝为Volcano类产生任何与Lava类不兼容的class文件。但是反过来就不一定是正确的，Java编译器可以为Lava类生成与Volcano类不兼容的class文件。如果Lava类不引用Volcano类，当改变Lava类中的被Volcano引用的方法的方法名时，这样只会对Lava类进行重编译，如果在运行你的程序时，你试图使用新版本的Lava，而继续使用老版本的Volcano类，而这个Volcano类将和新版本的Lava不兼容，那么当Volcano试图调用在Lava中不再存在的方法时，在第四趟class文件检验中将

抛出一个NoSuchMethodError。

在这种情况下，对于类Lava的修改将打破它与已有的Volcano的class文件的二进制兼容性。实际上，当更新一个已经在使用的类库，并且它的新版本与已有的代码不兼容时，这种情况就会发生。为了能方便地修改类库的代码，Java编程语言被设计成允许对一个类做多种修改，但并不要求对依赖于它的那些类进行重编译。Java语言规范中列出了用户可以做的多种改动，这些改动称为二进制兼容性规则。这些规则明确地定义了：在一个类中，哪些可以被修改、增加和删除，而并不破坏这个被修改的类与依赖于它的那些事先已存在的类之间的二进制兼容性。例如，向一个类中增加一个新的方法始终是一个影响二进制兼容性的改动，但是不能删除一个正在被其他类使用的方法。所以在这种情况下，当改变了Lava类中被Volcano调用的方法的名称时，就破坏了二进制兼容规则，因为实际上是删除了一个老的方法，并加入了一个新的方法。但是如果你加入了一个新的方法，并改写了老的方法，让它调用新的方法，那么，这个改动和所有早已使用Lava的、事先已有的class文件是二进制兼容的，包括Volcano。

3.5 Java虚拟机中内置的安全特性

Java虚拟机装载了一个类，并且对它进行了第一到第三趟的class文件检验，这些字节码就可以被运行了。除了对符号引用的检验（class文件检验的第四趟扫描），Java虚拟机在执行字节码时还进行其他一些内置的安全机制的操作。这些机制大多数是Java的类型安全的基础，它们作为Java编程语言保证Java程序健壮性的特性，已在第1章中列出了。同样，它们也是Java虚拟机的特性：

- 类型安全的引用转换。
- 结构化的内存访问（无指针算法）。
- 自动垃圾收集（不必显式地释放被分配的内存）。
- 数组边界检查。
- 空引用检查。

通过保证一个Java程序只能使用类型安全的、结构化的方法去访问内存，Java虚拟机使得Java程序更为健壮，也使得它们的运行更为安全。如果一个程序破坏内存、崩溃，或者可能导致其他程序崩溃，那么，它就是一个安全裂口。例如，如果正在运行一个任务关键的服务器进程，那么保证这个进程不能崩溃就非常重要。这种层次的健壮性在嵌入式系统中也非常重要，例如蜂窝电话，因为人们通常不希望重启机器。如果对内存的访问不加限制条件，会导致安全隐患的另一个原因是，一个老谋深算的黑客很可能暗中利用它破坏安全系统。例如，如果一个黑客知道一个类装载器在内存中的位置，他可以赋一个指针指向那块内存，从而对类装载器的数据进行操作。通过强制对内存的结构化访问，Java虚拟机可以产生健壮的程序，而且还可以阻挠那些黑客，使他们不能为了达到某些目的而破坏Java虚拟机的内在存储。

内置在Java虚拟机中的另一个安全特性——作为内存的结构化访问的一个后备——就是并未指明运行时数据空间在Java虚拟机内部是怎样分布的。运行时数据空间是指一些内存空间，Java虚拟机用这些空间来存储运行一个Java程序时所需要的数据：Java栈（每个线程一个）、一个存储字节码的方法区，以及一个垃圾收集堆（它用来存储由运行的程序创建的对象）。如果查看一

个class文件的内部，将找不到任何内存地址。当Java虚拟机装载一个class文件时，由它决定将这些字节码以及其他从class文件中解析得到的数据放置在内存的什么地方。当Java虚拟机启动一个线程时，由它决定将它为这个线程创建Java栈放到哪里。当它创建一个新的对象时，也是由它决定将这个对象放到内存中的什么地方。这样，一个黑客就不可能仅凭class文件中的内容，就知道在内存中的哪些数据代表了这个类，或从那些这个类实例化而得到的对象。（对于黑客来说）更糟糕的是，黑客不能通过阅读Java虚拟机的规范，来得到关于内存布局的任何信息。在规范中，并未说明Java虚拟机是怎样布局它的内存数据的。对于每个Java虚拟机的实现来说，由它的设计者来决定使用什么数据结构来表示运行时数据空间，并且将它们存放在内存的哪个位置。因此，即使黑客可以在一定程度上突破Java虚拟机的内存访问限制，他们也会在四处查找某些东西想进行暗中破坏时遇到困难。

禁止对内存进行非结构化访问，其实并不是Java虚拟机必须主动强制正在运行的程序这样做，这种禁止其实是字节码指令集本身的内在本质。就像在Java编程语言中，没有办法表达一个非结构化的内存访问，在字节码中也没有办法表达非结构化的内存访问——即使你自己书写字节码。因此，禁止对内存的非结构化访问是防止对内存恶意破坏的一种固有阻碍。

但是，对于由支持Java虚拟机的类型安全机制所建立的安全屏障，还是有办法可以突破的。虽然字节码指令集没有向用户提供不安全的、非结构化的内存访问方法，但是可以绕过字节码，即调用本地方法。当调用本地方法时，Java安全沙箱完全不起作用。首先，健壮性的保证对于本地方法并不适用，虽然不能通过Java方法去破坏内存，但是可以通过本地方法达到这个目的。最重要的是，本地方法没有经过Java API（这样它们就绕过了Java API），所以当一个本地方法试图做一些具有破坏性的动作时，安全管理器并未被检查。（当然，Java API本身也是经常这样做的，但Java API使用的本地方法是“被信任的”。）这样，一旦某个线程进入一个本地方法，不管Java虚拟机内置了何种安全策略，只要这个线程运行这个本地方法，安全策略将不再对这个线程适用。因此，安全管理器中包含了一个方法，该方法用来确定一个程序是否能装载动态连接库，因为在调用本地方法时动态连接库是必需的。例如，不可靠的applet就不能装载新的动态连接库，因此它们就不能安装自己的新的本地方法。但它们可以调用Java API中的方法，这些方法可能是本地的，但是这些方法是可信的。当线程调用本地方法时，这个线程就跳出了沙箱；因此，对于本地方法，这个安全模型就和以前提过的保障计算机安全的传统的安全模型完全相同了：在调用本地方法前必须确认它是可信任的。

为了保证安全而内置于Java虚拟机的最后一个机制，就是异常的结构化错误处理。因为Java虚拟机支持异常，所以当一些违反安全的行为发生时，它会做一些结构化处理，Java虚拟机将抛出一个异常或者一个错误，而不是崩溃。这个异常或错误将导致这个错误线程的死亡，而不是使整个系统陷入瘫痪。抛出一个错误（和抛出一个异常对应）总是导致抛出错误的这个线程死亡。这对一个运行的Java程序来说通常是一个不便因素，但它不会导致整个程序的中止。如果这个程序还有一些线程正在正常工作，则这些线程有可能继续正常工作，即使它的同伴已经死亡。而抛出一个异常可能导致这个线程的死亡，但是它经常作为一种手段被使用，使程序能够将控制从发生异常的地方转到处理这个异常的情况。

3.6 安全管理器和Java API

Java安全模型的前三个组成部分——类装载器体系结构、class文件检验器以及Java中内置的安全特性——一起达到一个共同的目的：保持Java虚拟机的实例和它正在运行的应用程序的内部完整性，使得它们不被下载的恶意或有漏洞的代码侵犯。相反，这个安全模型的第四个组成部分是安全管理器，它主要用于保护虚拟机的外部资源不被虚拟机内运行的恶意或有漏洞的代码侵犯。这个安全管理器是一个单独的对象，在运行的Java虚拟机中，它在访问控制——对于外部资源的访问控制——中起中枢作用。

安全管理器定义了沙箱的外部边界。因为它是可定制的，所以它允许为程序建立自定义的安全策略。当Java API进行任何可能不安全的操作时，它都会向安全管理器请求许可，从而强制执行自定义的安全策略。要向安全管理器请求许可，Java API将调用安全管理器对象的“check”方法。因为这些方法的名都以“check”开头，所以它们被称为“check”方法。例如，安全管理器的checkRead（）方法决定了线程是否可以读取一个特定的文件，checkWrite（）方法决定了线程能否对一个特定的文件进行写操作。这些方法的实现定义了应用程序的定制安全策略。

因为Java API在进行一个可能不安全的操作前，总是检查安全管理器，所以Java API不会在安全管理器建立的安全策略之下执行被禁止的操作。如果安全管理器禁止这个操作，Java API就不会执行这个操作。

当Java应用程序启动时，它还没有安全管理器，但是，应用程序通过将一个指向java.lang.SecurityManager或是其子类的实例传给setSecurityManager()，以此来安装安全管理器，这个动作是可选的。如果应用程序没有安装安全管理器，那么它将不会对请求Java API的任何动作做限制——Java API将做任何被请求的事（这就是Java应用程序在默认情况下将不会有任何安全限制的原因，例如限制不可靠applet的动作）。如果应用程序确实安装了安全管理器，那么在版本1.0或者1.1中，安全管理器将负责应用程序整个剩余的生命周期，它不能被替代、扩展或者修改。从这一点开始，Java API将只执行那些被安全管理器同意的请求。然而，在版本1.2中，当前安装的安全管理器可以被允许替换它的代码所替换，这是通过对指向另一个不同的安全管理器对象的引用调用System.setSecurityManager()实现的。

一般来说，如果一个受检查的动作被禁止，安全管理器的“check”方法将抛出一个安全异常，如果这个动作被允许，则简单地返回。因此，当一个Java API即将进行一个潜在不安全的动作时，它将遵循以下两个步骤。首先，Java API的代码检查有没有安装安全管理器，如果没有安装，则跳过第二步直接继续这个潜在不安全的动作。否则，在第二步中，它将调用安全管理器中的合适的“check”方法。如果这个操作被禁止，那么这个“check”方法会抛出一个安全异常，这将导致该Java API方法立即中止，这个潜在不安全的操作将不会被执行。相反，如果这个操作被允许，那么这个“check”方法将简单地返回。在这种情况下，这个Java API方法将继续运行，并执行这个潜在不安全的操作。

在本章前面曾提到，安全管理器负责两个方面的工作：说明一个安全策略以及执行这个安全策略。安全策略指明了哪种代码将被允许执行哪种操作，它是由安全管理器中的“check”方法的代码定义的。当它们被调用时，这个策略将被这个“check”方法所实施。

在版本1.2以前，java.lang.SecurityManager是一个抽象类。要在版本1.0或1.1中建立定制的安全策略，必须子类化SecurityManager并实现它的check方法，从而编写出自己的安全管理器。你的应用程序将实例化并安装这个安全管理器，从此，在应用程序剩余的生命周期中，它将实施你在其check方法的代码中所定义的安全策略。

虽然安全管理器的可配置性是Java安全模型的最大优点之一，但它也是一个潜在的弱点。编写一个安全管理器是一项复杂的任务，并可能导致错误。在实现安全管理器的check方法时，任何错误都将变成运行时的安全漏洞。为了帮助开发人员和终端用户方便地、尽量正确地建立一个基于签名代码的细粒度的安全策略，版本1.2中的类java.lang.SecurityManager是一个具体的类，它提供了一个默认的安全管理器的实现。（在本书的剩余部分，这个由版本1.2提供的安全管理器的默认实现将被称为“具体安全管理器”。）用户的应用程序可以显式地实例化并安装这个安全管理器，或者也可以让它自动安装。例如，在Sun的Java 2 SDK版本1.2中，可以在命令行使用-Djava.security.manager选项来指明安装具体安全管理器。

具体安全管理器类允许用户不用Java代码定义自己的定制策略，而是用一个称为策略文件的ASCII文件。在策略文件中，可以给代码来源授予权限。权限是用类定义的，它是java.security.Permission的子类。例如，java.io.FilePermission表示了对一个文件的读、写、执行或者删除权限。代码来源是由代码库的URL和一些签名组成的，从这个URL可以装载代码，而签名则为这个代码作担保。当创建安全管理器时，它对策略文件进行解析，并创建CodeSource（代码来源）和Permission（权限）对象，这些对象被封装在一个单独的Policy对象中，这个Policy对象就代表了运行时的策略。任何时刻只能有一个Policy对象被安装。

类装载器将类型放到保护域中，保护域封装了授予代码来源的所有权限，这些代码来源由装载的类型代表。在版本1.2中，每一个被装在虚拟机中的类型都属于一个且只属于一个保护域。这个保护域会被记下来，并且在决定这个代码是否被允许执行一些可能不安全的操作时使用它。

当具体安全管理器的check方法被调用时，它们中的大多数都将请求传递给一个称为AccessController的类。这个AccessController使用了包含在保护域对象中的信息，这个对象所属的类的方法在调用栈中，AccessController进行栈检查以确定这个操作能否被执行。

版本1.2中安全管理器有一些变化。在版本1.0和1.1中，每一个check方法都在它们的方法名中指出了将检查什么。为了检查能否读取一个特定的文件，Java API调用了安全管理器的checkRead()方法，并且将被读取文件的路径名作为参数传入。例如，在试图读取文件/tmp/finances.dat前，安全管理器调用了checkRead（“/tmp/finances.dat”）。

安全管理器声明了28个这种check方法，本章的后半部分将这些check方法称为“老式的check方法”。虽然在版本1.2中加入了一些新的方法，使得这些老式的check方法过时了，但是为了保持Java API的向后兼容性，这些API仍然可以调用老式的check方法，就像在以前的版本中一样。

下面列出了这28个老式的check方法，同时也列出了它们被Java API代码触发调用时的潜在不安全动作：

- checkConnect（String host，int port）——打开一个指定主机和端口号的socket连接前被调用。
- checkConnect（String host，int port，Object context）——在被传递的安全上下文中打开

一个指定主机和端口号的socket连接前被调用。

- checkAccept（String host，int port）——接收一个来自于指定主机和端口号的socket连接前被调用。
- checkCreateClassLoader()——创建一个新的类装载器前被调用。
- checkAccess（Thread t）——改变一个线程（例如改变它的优先级，中止它等等）前被调用。
- checkAccess（ThreadGroup t）——改变一个线程组（如加入一个新的线程，设置守护进程等等）前被调用。
- checkExit()——应用程序退出前被调用。
- checkLink()——装载一个包含本地方法的动态库前被调用。
- checkRead（FileDescriptor fd）——读取指定的文件前被调用。
- checkRead（String file）——读取指定的文件前被调用。
- checkRead（String file，Object context）——在被传递的安全上下文中读取指定的文件前被调用。
- checkWrite（FileDescriptor fd）——对指定的文件进行写操作前被调用。
- checkWrite（String file）——对指定的文件进行写操作前被调用。
- checkDelete（String file）——删除指定的文件前被调用。
- checkListen（int port）——在指定的本地端口号上等待连接前被调用。
- checkMulticast（InedAddress maddr）——加入、离开、发送或者接收IP组播前被调用。
- checkMulticast（InedAddress maddr，byte ttl）——加入、离开、发送或者接收IP组播前被调用。
- checkPropertiesAccess()——访问和修改一般的系统属性前被调用。
- checkPropertiesAccess（String key）——访问或修改指定的系统属性前被调用。
- checkTopLevelWindow（Object Window）——不出示任何警告地显示指定的窗口前被调用。
- checkPrintJobAccess()——初始化一个打印任务请求前被调用。
- checkSystemClipboardAccess()——访问系统剪贴板前被调用。
- checkAWTEventQueueAccess()——访问AWT事件队列前被调用。
- checkPackageAccess（String pkg）——访问指定的包（被类装载器使用）中的类型前被调用。
- checkPackageDefinition（String pkg）——在指定的包（被类装载器使用）中加入一个新类前被调用。
- checkSetFactory()——设置被ServerSocket或Socket使用的socket类或者设置被URL使用的URL流处理器前被调用。
- checkMemberAccess()——通过映像API访问类信息前被调用。

在版本1.2中，定义了一些许可类，这些类的实例代表了这样的代码：即它的操作是被允许的。在版本1.2的java.lang.SecurityManager类中，添加了一对新的check方法，方法名都是checkPermission（）：

- checkPermission（Permission perm）——进行某个操作（它需要指定的权限）前被调用。

• checkPermission（Permission perm，Object context）——在被传递的安全上下文中进行某个操作（它需要指定的权限）前被调用。

这个checkPermission()方法接受一个Permission对象的引用，它指出了被请求的操作。这样，这个方法就提供了另一种方式，询问安全管理器是否可以执行一个潜在不安全的动作。例如，要确定是否可以读文件/tmp/finances.dat，版本1.2中的Java API可以在两种方法中任选一种。Java API可以采用老式的步骤，调用老式的方法checkRead（），并将字符串“/tmp/finances.dat”作为参数传递给它。或者，Java API也可以采用新的方法，创建一个java.io.FilePermission对象，将字符串“/tmp/finances.dat”和“read”传给FilePermission构造器，然后Java API将这个Permission对象传给安全管理器的checkPermission（）方法。

不管是使用老式方式调用一个老式的check方法，还是使用新的方法创建一个Permission对象并调用checkPermission（），都将产生相同的结果。为了保持安全管理器对版本1.0和1.1的向后兼容性，版本1.2中的Java API继续使用了老的方法。在版本1.2中，Java API继续调用了28个老式的check方法。但是，在具体安全管理器类中，老的方法大部分都用新的checkPermission（）方法实现了。因此通过调用具体安全管理器的老式方法，Java API实际上间接地调用了checkPermission（）方法。例如，在具体安全管理器中，方法checkRead（）的实现只是简单地实例化了一个新的FilePermission对象，将传给它的路径名字符串和字符串“read”一起传递给FilePermission的构造器。这个checkRead（）方法然后调用checkPermission（），并将FilePermission对象的引用传递给它。

Java API可能多次直接调用checkPermission（）。对于版本1.2及其以后的版本中引入的新的潜在不安全操作的概念（译者注：不在上面列表中的新的检查点），不存在老式的check方法。所以，在这种情况下，Java API将创建一个新的Permission对象，这个对象不存在与之相关的check方法。然后，Java API将把这个Permission对象直接传给安全管理器的checkPermission（）方法。

在具体安全管理器类中，checkPermission（）方法同样负责决定，是否允许将某个操作的任务委派给另一个方法。这个checkPermission（）方法只是简单地调用了类java.security.AccessController中的静态checkPermission（）方法，并将这个Permission对象传递给它。因此，在使用具体安全管理器时，这个AccessController类是真正负责执行安全策略的实体。

版本1.2中的所有这些改变都是对版本1.0和1.1向后兼容的。如果你为版本1.1创建了一个安全管理器，它也可以在版本1.2中正常运行。也可以在版本1.2中创建一个自定义的安全管理器，这样可以创建一个不同的安全基础架构，从而满足具体安全管理器实现所不能解决的特殊的安全需要。然而，利用内置在具体安全管理器中的灵活性和可扩展性，大多数人的安全需要都应该能被满足。

3.7　代码签名和认证

Java安全模型很重要的一点就是它能支持认证，这是在Java 1.1的java.security包及其子包中引入的特性。认证功能加强了用户的能力，使用户能通过实现一个沙箱来建立多种安全策略，这个沙箱可以依赖于为这个代码提供担保的对象来改变。认证可以使用户确认，由某些团体担

保的一些class文件是值得信任的，并且这些class文件在到达用户虚拟机的途中没有被改变。这样，如果用户在一定程度上信任这个为代码作担保的团体，也就可以在一定程度上简化沙箱对这段代码实施的限制。可以对由不同团体签名的代码建立不同的安全限制。

要对一段代码作担保或者签名，必须首先生成一个公钥/私钥对。用户应该保管那把私钥，而把公钥公开。至少，应该把公钥给那些要在你的签名上建立安全策略的人。（发布公钥并不像看起来那么简单，这将稍后作解释。）一旦拥有了一个公钥/私钥对，就必须将要签名的class文件和其他文件放到一个JAR文件中，然后使用一个工具（例如版本1.2 SDK中的jarsigner）对整个JAR文件签名。这个签名工具将首先对JAR文件的内容进行单向散列计算，以产生一个散列。然后这个工具将用私钥对这个散列进行签名，并且将经过签名后的散列加到JAR文件的末尾。这个签名后的散列代表了你对这个JAR文件内容的数字签名。当你发布这个包含签名散列的JAR文件时，那些持有你的公钥的人将对JAR文件验证两件事：这个JAR文件确实是你签名的，并且在你签名后这个JAR文件没有做过任何改动。

数字签名过程的第一步是一个单向的散列计算，它输入大量的数据，但产生少量的数据，称为散列。在这个JAR文件的例子中，这个计算的大量输入就是组成这个JAR文件内容的字节流。这个单向散列计算之所以被称为“单向”，是因为在只给出散列（即那个少量的数据）的情况下，这个散列值不能包含足够的输入的信息，因此不能从散列重新生成原输入。这个计算是单向的，从大到小，从输入到散列。

散列也被称为消息文摘，它相当于一种输入“指纹”。虽然不同的输入可能产生相同的散列，但通常认为，在实际情况下，一个散列足以代表了产生它的输入。就像用指纹代表人一样，一个散列也被用于识别用单向散列算法产生这个散列的输入。在认证过程中，散列被用于验证某个输入是否和产生这个原始散列的输入相同，换句话说，这个输入在到达目的地的途中有没有被改动。

因为不可能仅仅用散列重构原输入，一个散列仅在可以得到原输入时才有用。因此，必须将输入和散列一起传输。对它们本身来说，输入和散列的组合并不安全，因为就算是一个不怎么聪明的黑客，也可以方便地将输入和散列一起替换掉。为了防止这种情况的发生，必须在发送散列前，用私钥对它进行加密。只加密散列而不是对整个JAR进行加密，这是因为用私钥进行加密是一个相当费时的过程，一般来说，从JAR文件中计算、产生一个单项散列，并对这个散列用私钥进行加密，要比对整个JAR文件用私钥进行加密来得快。只有当一个黑客拥有你的私钥时，他才能同时替换输入和加密后的散列，因此要小心保存你的私钥。这样，相对于输入和散列组合，破解输入和加密散列组合就更困难了，因为黑客不可能拥有你的私钥。

任何用你的私钥加密的东西都可以用你的公钥解密。公钥/私钥对具有这种特点，在仅给出公钥的情况下时，想要产生私钥是非常困难的。如果黑客不能得到你的私钥，对他来说最好的选择就是试图将原输入替换为另一个输入，这个输入必须和原输入产生相同的散列值。例如，如果一个黑客想要在你的JAR文件中将一个class文件替换成另一个执行恶意动作的class文件，被修改的JAR文件（包含了那个恶意的class文件）产生一个不同的散列的几率是非常高的。但是这个黑客可以往JAR文件中添加随机的数据，直到（对这个改动后的JAR文件进行单向散列计算）产生和原来一样的散列值。如果黑客可以产生这样一个可供选择的输入——既可以帮助黑客达

到他邪恶的目的，又可以产生和你原来的输入一样的散列——这个黑客就不需要你的私钥了。因为这个黑客的输入产生了和你的输入一样的散列值，而且你已经用你的私钥对这个散列进行了签名，所以这个黑客只要简单地将JAR文件中的、你签名后的散列加到他的输入后就可以了。怎样才能防止黑客采用这种方法呢？然而对于黑客来说，这样的方法会花去大量的时间，因此几乎是不可行的。

因为单向散列算法是从大量数据（输入）中产生少量数据（消息摘要或者散列），所以不同的输入可能产生相同的散列。单向散列算法倾向于充分随机地分布产生相同散列的输入，从而使产生相同散列值的概率主要依赖于散列的大小。例如，如果使用了一个长8位的散列值，散列算法最多产生256个不同的散列值。如果有一个JAR文件，它的散列值是100，然后你开始将这个8位的散列算法在其他JAR文件上运用，毫无疑问，每进行大约256次计算，将可能得到一个值为100的散列。当然，如果散列的位数越多，产生相同散列值的情况就越不可能发生。在实际情况中，普遍采用的是64位或128位的散列，通常认为这个长度已经足够了，这时要想从一个不同的输入中产生一个相同的散列的计算是不可行的。因此，防止黑客用恶意输入替换你的善意输入，并且产生相同的散列值的主要障碍在于，他必须花费大量的时间和资源才能找到这个恶意的输入。

在产生散列值并用私钥对它签名以后，随后一个步骤就是将这个加密后的散列值加到同一个JAR文件中，这个JAR文件还包含了你最初产生这个散列的文件。这样，一个经签名的JAR文件，就包含了输入——你要担保的class文件和数据文件——以及用你的私钥加密过的散列值（由输入产生）。加密的散列代表了你对在同一个JAR文件中的类和数据文件的数字签名，图3-3中图形化地画出了对一个JAR文件进行签名的过程。

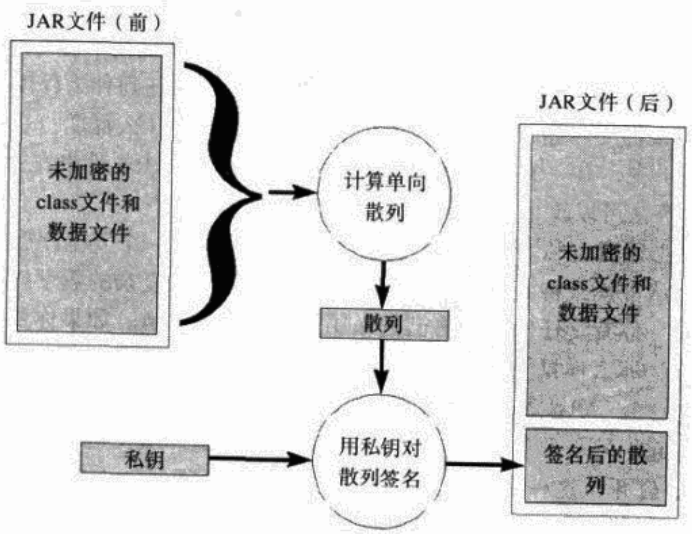


图3-3　对一个JAR文件数字签名

要认证一个已签名的JAR文件，接收者必须用公钥对签名散列进行解密，得到的结果应该和从JAR文件计算而得到的散列值相等。为了验证一个JAR文件在签名后未被改动，接收者只要对

JAR文件的内容实施单向散列算法，就像在签名过程中所做的那样。（记住，并没有对JAR文件的内容进行加密，所以任何人都可以看见它。你只是将一个数字签名加到了那个JAR文件中。）如果得到的散列值和加密的散列值匹配，那么接收者就可以推断，你确实为JAR文件进行了担保，而且这个JAR文件的内容在加上你的签名以后没有被改动过。这个JAR文件中包含的代码就可以被放在一个不严格的沙箱中，这个沙箱信任你的签名。图3-4中显示了验证一个经过数字签名后的JAR文件的过程。

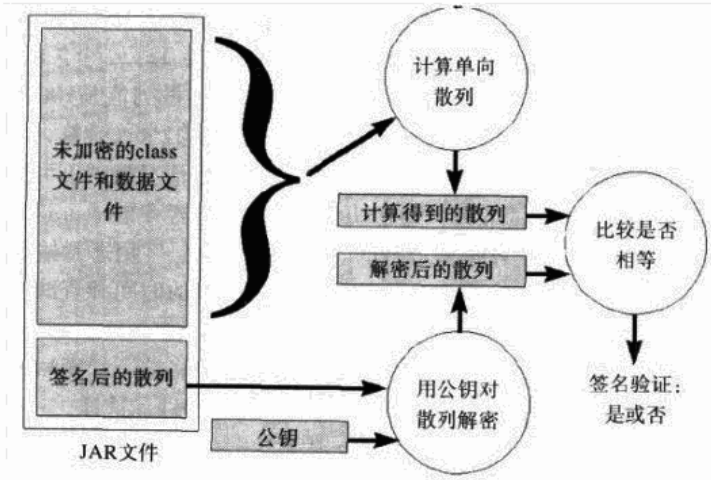


图3-4 认证一个经过数字签名后的JAR文件

尽管最初在Java 1.1版本中引入的认证技术利用了可信赖的数学原理，但是数学并不能解决所有问题，实际上，Java的认证技术引发了许多问题。例如，认证技术没有说明应该信任谁，以及对他们的信任程度等问题。你对自己从未听说过的小公司信任到什么程度？你对一个家喻户晓的公司信任到什么程度？你对自己公司的另一个部门信任到什么程度？有没有可能在一个特定的公司（或部门）中，有一个捣蛋的员工在这个公司签名的JAR文件中安置了一颗时间炸弹？没有任何密码学的算法可以解答这些问题。

在认证技术中必须假设私钥已被秘密保管，这就引发了另一个安全问题。如果私钥没有被保管好，那么整个认证模式将变成一个不但无效、而且危险的复杂的数学操作，因为它给出一个错误的安全意义。你有责任保管好自己的私钥不让其他人知道。如果你要授权一段签名的代码访问自己的系统，那么你只能指望签名的公司或者个人妥善保管了它们的私钥。对于任何一个团体来说，建立一个密钥管理模式来保证私钥不被泄露出去是一项具有挑战性的任务。

由这个技术引起了另一个和公钥的发布有关的问题。虽然最初这看起来很奇怪，但是，在认证技术中将公钥公开，这种假设本身就产生了一些安全问题。例如，假设想在沙箱中对由Evan担保的代码放宽限制，为了达到这个目的，必须得到Evan的公钥。但是怎样才能得到他的公钥呢？如果你认识Evan，可以邀请他喝咖啡并要求他将他的公钥带给你，这样他就可以亲自将公钥交给你。但是如果你不认识Evan怎么办？你可能想，可以简单地访问Evan的Web站点从网页上得到他的公钥，或者，可以打电话给Evan要求他将公钥通过Email发给你。Evan应该不会

不把他的公钥给你这样一个陌生人，因为公钥本来就是设计成公开的，Evan甚至不必担心谁得到了他的公钥。那么，问题是什么呢？问题在于虽然Evan在将他的公钥传给你时不必担心你的身份，但是你会担心他的身份。Evan很乐意把他的公钥给你，但是你怎么知道自己得到的公钥确实是Evan发送的呢？

公钥发布的困难在于，无论采取何种通信方式，消息——即公钥——可能潜在地被篡改或者偷偷替换了。当你访问Evan的网页时，网页很有可能在发送到你的浏览器的途中已经被截取并涂改了。比如有一个黑客Dastardly Doug，当你想从Evan的网页上把他的公钥拷贝下来时，实际上可能拷贝的是Dastardly Doug的公钥。Doug可能也改变了Evan的Email地址，并用他自己的恶意的公钥替换了Evan的善意的公钥。如果Doug可以成功地用自己的公钥替换Evan的公钥，Doug就可以假装成Evan，并且利用你对Evan签名的信任来侵入你的系统了。

难道公钥发布的困难不也是一个认证问题吗，而这个问题正是认证技术本身需要解决的。事实上确实如此，通过认证施加于本身，Evan可以使得Doug想要用自己的公钥替换Evan的公钥的企图变得更为困难。

为了解决公钥发布的困难，建立了许多证书机构来为这些公钥做担保。例如，Evan可以到一个证书机构去，给出他的信任状（例如出生证明、驾驶证、护照等等）以及他的公钥。一旦确认Evan是如他自己所说的那个人，证书机构将会用证书机构的私钥对Evan的公钥进行签名，最终得到的数字序列被称为证书。

这样，Evan就可以发布他的证书，而不是他的公钥了。你可以从Evan的网页上获取Evan的证书，或者通过Email及其他任何未必安全的通信方式得到。当你得到了证书以后，可以用证书机构的公钥对证书进行解密以得到Evan的公钥。这种证书模式使得Doug可以替换Evan的公钥的可能性更小了，因为要达到这个目的，Doug还必须得到证书机构的私钥。

虽然证书很大程度上改善了公钥的发布，但是仍然存在一些问题。首先，怎样得到证书机构的公钥？你要用这个公钥来认证其他所有人的公钥，那么，如果你认识证书机构中的某些工作人员，你可以邀请他过来喝咖啡并且要求他将他们的公钥亲自带来。但是如果你不认识证书机构的任何员工呢？任何人可以假扮成一个证书机构。难道证书机构就不会因为某些捣蛋的员工而和其他公司一样可疑吗？

尽管有这些问题，但总的来说，在Java 1.1中引入的代码签名功能提供了足够的安全，这种安全使用户在需要时可以放宽其沙箱限制。虽然认证技术并没有消除所有和放宽沙箱限制相关的风险，但是它可以减小这些风险。安全性是一种代价和安全之间的折衷：安全风险越小，安全的代价就越高。用户必须将计算机或网络安全策略有关的代价和被保护的信息或计算资源被窃取破坏后所承受的代价放在一起衡量。计算机或网络安全策略的特性应该由所要保护的财产的价值来决定。Java的认证技术是一种有用的工具，它和Java的沙箱一样，可以帮助管理在系统上运行网络移动代码的代价和风险。

3.8 一个代码签名示例

这是一个用Java 2 SDK 1.2中的工具jarsigner进行代码签名的例子，考虑以下类型：Doer、Friend和Stranger。第一个类型Doer定义了另外两种类型（类Friend和类Stranger）实现的接口：

```
// On CD-ROM in file
// security/ex2/com/artima/security/doer/Doer.java
package com.artima.security.doer;

public interface Doer {

    void doYourThing();
}
```

Doer仅声明了一个方法doYourThing ( )。类Friend和类Stranger用基本上相同的方式实现了这个方法。实际上，这两种方法除了名字不同以外，本质上是一样的：

```
// On CD-ROM in file
// security/ex2/com/artima/security/friend/Friend.java
package com.artima.security.friend;
import com.artima.security.doer.Doer;
import java.security.AccessController;
import java.security.PrivilegedAction;

public class Friend implements Doer {

    private Doer next;
    private boolean direct;

    public Friend(Doer next, boolean direct) {
        this.next = next;
        this.direct = direct;
    }

    public void doYourThing() {

        if (direct) {

            next.doYourThing();
        }
        else {
            AccessController.doPrivileged(
                new PrivilegedAction() {
                    public Object run() {
                        next.doYourThing();
                        return null;
                    }
                }
            );
        }
    }
}
```

```
// On CD-ROM in file
// security/ex2/com/artima/security/stranger/Stranger.java
package com.artima.security.stranger;
import com.artima.security.doer.Doer;
import java.security.AccessController;
import java.security.PrivilegedAction;

public class Stranger implements Doer {

    private Doer next;
    private boolean direct;

    public Stranger(Doer next, boolean direct) {
        this.next = next;
        this.direct = direct;
    }

    public void doYourThing() {

        if (direct) {

            next.doYourThing();
        }
        else {
            AccessController.doPrivileged(
                new PrivilegedAction() {
                    public Object run() {
                        next.doYourThing();
                        return null;
                    }
                }
            );
        }
    }
}
```

这些类型（Doer、Friend和Stranger）是为了说明访问控制的栈检查机制而设计的。在本章后面将给出一些栈检查的例子，那时读者就会理解设计它们的目的。然而在这里，通过编译Friend和Stranger而产生的class文件必须被签名，以便在以后的栈检查的例子中使用它们。从Friend.java产生的class文件将由一个比较信任的称为“friend”的团体签名，而从Stranger.java产生的class文件将由一个不太信任的称为“stranger”的团体签名。由Doer产生的class文件不用签名。

在这些文件被签名以前，你必须将它们放入JAR文件中。因为Friend和Stranger的class文件将被两个不同的团体签名，所以它们将被放置在两个不同的JAR文件中。通过编译Friend.java产生的两个class文件Friend.class和Friend$1.class，将被放置在一个名为friend.jar的JAR文件中；同

样，由编译Stranger.java产生的两个class文件Stranger.class和Stranger$1.class，将被放到一个名为stranger.jar的JAR文件中。（注意，虽然这些例子中的所有文件都在CD-ROM的security/ex2目录下，但是要实践产生这些文件的命令，必须将整个security/ex2目录拷贝到一个可写的媒体中，例如硬盘。这点你可能已经知道了？）

Friend.java的class文件被javac编译器放到了目录security/ex2/com/artima/security/friend之下，因为Friend类在包com.artima.security.friend中被声明，Friend.java的class文件必须被放置在com/artima/security/friend目录下的JAR文件中。在security/ex2目录下执行下面的命令，将会把Friend.class和Friend$1.class放到一个新建的名为friend.jar的JAR文件中，这个JAR文件就被放在了当前目录security/ex2下：

```
jar cvf friend.jar com/artima/security/friend/*.class
```

一旦上面的命令执行完后，Friend.java的class文件必须被删除，以便Java虚拟机在运行下面的访问控制的例子时无法找到它：

```
rm com/artima/security/friend/Friend.class
rm com/artima/security/friend/Friend$1.class
```

Stranger.java的class文件被javac放在了security/ex2/com/artima/security/stranger目录下，将它放入一个JAR文件的过程和上面的过程相同。在security/ex2目录下，执行命令：

```
jar cvf stranger.jar com/artima/security/stranger/*.class
rm com/artima/security/stranger/Stranger.class
rm com/artima/security/stranger/Stranger$1.class
```

为了用Java 2 SDK 1.2中的jarsigner工具对JAR文件进行签名，keystore文件中必须存储签名者的公钥/私钥对，这个文件用来存储已命名的、受密码保护的密钥。Java 2 SDK 1.2中的keytool程序可用来生成新的密钥对，将这个密钥对和一个名称（或别名）相关联，并且用密码将它们保护起来。这个别名在每一个keystore文件中都是独立的，它用于在一个特定的keystore文件中识别一个特定的密钥对。要访问或者修改包含在keystore文件中的密钥对的信息，必须要有这个密钥对的密码。

这个访问控制的例子需要在security/ex2目录下的名为ivjmkeys的keystore文件，这个文件中包含别名为“friend”和“stranger”的两个密钥对。在security/ex2目录下运行下面的命令，将为别名friend产生密码为friend4life的密钥对。在这个过程中，它将生成一个名为ijvmkeys的keystore文件：

```
keytool -genkey -alias friend -keypass friend4life
-validity 10000 -keystore ijvmkeys
```

在这个keytool命令中，-validity 10000这个命令行参数说明了这个名密钥对将在10000天之内有效，相当于27年多，这个时间应该是足够长的，超过了本书的生命周期。当命令运行时，它将产生一个keystore密码，在对这个keystore文件进行任意访问或修改时都需要这个keystore密码。赋给ijvmkeys的密码是“ijvm2ed”。

可以用一条类似的命令为stranger生成密钥对：

```
keytool -genkey -alias stranger -keypass stranger4life
-validity 10000 -keystore ijvmkeys
```

既然现在keystore文件ijvmkeys包含了friend和stranger的密钥对，而且JAR文件friend.jar和stranger.jar包含了合适的class文件，JAR文件就可以被最终签名了。在example/ex2目录下执行下面的jarsigner命令，将用friend的私钥对包含在friend.jar文件中的class文件进行签名：

```
jarsigner -keystore ijvmkeys -storepass ijvm2ed -keypass
friend4life friend.jar friend
```

同样，下面的命令将用stranger的私钥对包含在stranger.jar文件中的class文件进行签名：

```
jarsigner -keystore ijvmkeys -storepass ijvm2ed -keypass
stranger4life stranger.jar stranger
```

好了，为了对两个JAR文件进行签名，必须做上面这许多事。值得注意的是，在现实世界中，必须确保不要让那些意图不轨的人得到你的私钥，并要和他们保持距离。这就意味着你不能丢失这个keystore文件，必须记住密码，等等。而且，还必须让那些试图用你的签名来让你的代码访问他们的系统的人得到你的公钥。

3.9 策略

在以前已经提到过，Java沙箱安全模型的最大的优点之一就是沙箱可以是用户自定义的。在Java版本1.1中引入的代码签名和认证技术使正在运行的应用程序可以对代码区分不同的信任度。通过自定义沙箱，被信任的代码可以比不可靠的代码获得更多的访问系统资源的权限。这就防止了不可靠代码访问系统，但是却允许被信任的代码访问系统并进行工作。Java安全体系结构的真正好处在于，它可以对代码授予不同层次的信任度来部分地访问系统。

Microsoft提供了ActiveX控件认证技术，它和Java的认证技术相类似，但是ActiveX控件并不在沙箱中运行。这样，使用了ActiveX，一系列移动代码要么是被完全信任的，要么是完全不被信任的。如果一个ActiveX控件不被信任，它将被拒绝执行。虽然这对于没有认证来说是一个很大的提高，但是如果一些恶意的或是有漏洞的代码得到了认证，这段危险的代码将拥有对系统的完全访问权。Java的安全体系结构的优点之一就是，代码可以被授予只对它需要的资源进行访问的有限权限。即使一些恶意的或者有漏洞的代码得到了认证，它也很少有机会进行破坏。例如，一段恶意的或者有漏洞的代码可能只能删除一个固定目录下的为它设置的文件，而不是在本地硬盘上的所有文件。

版本1.2的安全体系结构的主要目标之一就是使建立（以签名代码为基础的）细粒度的访问控制策略的过程更为简单且更少出错。为了将不同的系统访问权限授予不同的代码单元，Java的访问控制机制必须能确认应该给每个代码段授予什么样的权限。为了使这个过程变得容易，载入版本1.2或其他Jva虚拟机的每一个代码段（每个class文件）将和一个代码来源关联。代码来源主要说明了代码从哪里来，如果它被某个人签名担保的话，是从谁那里来。在版本1.2的安全模型中，权限（系统访问权限）是授给代码来源的。因此，如果代码段请求访问一个特定的系统资源，只有当这个访问权限是和那段代码的代码来源相关联时，Java虚拟机才会把对那个资源的访问权限授予这段代码。

在版本1.2的安全体系结构中，对应于整个Java应用程序的一个访问控制策略是由抽象类java.security.Policy的一个子类的单个实例所表示的。在任何时候，每一个应用程序实际上都只有一个Policy对象。获得许可的代码可以用一个新的Policy对象替换当前的Policy对象，这是通过调用Policy.setPolicy（）并把一个新的Policy对象的引用传递给它来实现的。类装载器利用这个Policy对象来帮助它们决定，在把一段代码导入虚拟机时应该给它们什么样的权限。

安全策略是一个从描述运行代码的属性集合到这段代码所拥有的权限的映射。在版本1.2的安全体系结构中，描述运行代码的属性被总称为代码来源。一个代码来源是由一个java.security.CodeSource对象表示的，这个对象中包含了一个java.net.URL，它表示代码库和代表了签名者的零个或多个证书对象的数组。证书对象是抽象类java.security.cert.Certificate的子类的一个实例，一个Certificate对象抽象表示了从一个人到一个公钥的绑定，以及另一个为这个绑定作担保的人（以前提过的证书机构）。CodeSource对象包含了一个Certificate对象的数组，因为同一段代码可以被多个团体签名（担保）。这个签名通常是从JAR文件中获得的。

在版本1.2中，所有和具体安全管理器有关的工具和访问控制体系结构都只能对证书起作用，而不能对“赤裸”的公钥起作用。如果附近没有证书机构，可以用私钥对公钥签名，生成一个自签名的证书。Java 2 SDK 1.2中的keytool程序在生成密钥时，总是会产生一个自签名的证书。例如，在本章上面给出的代码签名的例子中，keytool不仅产生了公钥/私钥对，还为别名friend和stranger产生了自签名的证书。

权限是用抽象类java.security.Permission的一个子类的实例表示的。一个Permission对象有三个属性：类型、名字和可选的操作。权限的类型是由Permission类的名字指定的，例如：java.io.FilePermission，java.net.SocketPermission，以及java.awt.AWTPermission。权限的名字是封装在Permission对象内的。例如，某个FilePermission的名字可能是“/my/finances.dat”，某个SocketPermission的名字可能是“applets.artima.com:2000”，某个AWTPermission的名字可能是“showWindowWithoutBannerWarning”。Permission对象的第三个属性是它的动作。并不是所有的权限都有动作。例如，FilePermission的动作是“read，write”，SocketPermission的动作是“accept，connect”。如果一个FilePermission的名字为/my/finances.dat，并且有动作“read，write”，那么它就表示对文件/my/finances.dat可进行读写操作。名字和动作都是由字符串来表示的。

Java API有一个很大的权限层次结构，表示了所有可能潜在危险的操作。可以根据自己的目的，创建自己的Permission类来表示自定义的权限。例如，可以创建一个Permission类来表示对属性数据库的特定记录的访问权限。定义自定义的Permission类也是一种扩展版本1.2的安全机制来满足自己需要的方法。如果你创建了自己的Permission类，可以像使用Java API中的Permission类一样来使用它们。

在Policy对象中，每一个CodeSource是和一个或多个Permission对象相关联的。和一个CodeSource相关联的Permission对象被封装在java.security.PermissionCollection的一个子类实例中。类装载器可以调用Policy.getPolicy（）来获得一个当前有效的Policy对象的引用。然后它们可以调用Policy对象的getPermission（）方法，传入一个CodeSource，从而得到和那个CodeSource对应的Permission对象的PermissionCollection。类装载器然后可以使用这个从Policy对象中得到的PermissionCollection来帮助判断应该给导入的代码授予什么权限。

策略文件

java.security.Policy是一个抽象类，具体Policy子类的实现细节之一就是该子类的实例怎样知道策略应该是什么。子类可以采取多种方法，例如对一个已序列化的Policy对象进行并行化，从数据库中抽取策略，或者从文件中读取策略。由Sun提供的在Java 1.2平台下的具体Policy子类采用了最后一种方法：在一个ASCII策略文件中用上下文无关文法描述安全策略。

一个策略文件包括了一系列grant子句，每一个grant子句将一些权限授给一个代码来源。在上面已经讲过，一个代码来源包含了一个代码库和一系列签名，代码库是指出这个代码从那里下载的URL。在策略文件中，签名用别名来代表，这些签名是保存在keystore文件中的签名者的公钥。这个keystore可以在策略文件中用一个keystore语句显式说明。

策略文件的例子如CD-ROM中security/ex2目录下的policyfile.txt文件：

```
keystore "ijvmkeys";

grant signedBy "friend" {
    permission java.io.FilePermission "question.txt","read";
    permission java.io.FilePermission "answer.txt", "read";
};

grant signedBy "stranger" {
    permission java.io.FilePermission "question.txt", "read";
};

grant codeBase "file:${com.artima.ijvm.cdrom.home}/security/ex2/*" {
    permission java.io.FilePermission "question.txt","read";
    permission java.io.FilePermission "answer.txt", "read";
};
```

在这个policyfile.txt文件中，第一条语句是keystore语句：

```
keystore "ijvmkeys";
```

这个keystore语句说明，密钥别名（策略文件其余部分将提到）指向存储在名为“ijvmkeys”的文件中的证书。因为这个文件名没有包含路径，这个文件必须是存储在当前目录下——Java应用程序使用该策略文件的启动目录。

这个策略文件中的第二条语句是一条grant语句：

```
grant signedBy "friend" {
    permission java.io.FilePermission "question.txt", "read";
    permission java.io.FilePermission "answer.txt", "read";
};
```

这条语句将授予由别名为“friend”的实体签名的所有代码两个权限。被授予的权限是：读取question.txt文件的权限，以及读取answer.txt文件的权限。因为这些文件名没有路径，所以它们必须是存储在当前目录下的，也就是这个应用程序的启动目录。因为这个grant子句中没有提到代码库，所以由friend签名的代码可以来自任何代码库。任何由friend签名的代码，不管是来

自哪个代码库的，都将被授予对question.txt和answer.txt进行读操作的权限。

policyfile.txt中的第三个语句也是一条grant语句，和上一条语句的形式类似：

```
grant signedBy "stranger" {
    permission java.io.FilePermission "question.txt", "read";
};
```

这条语句将授予所有由别名为“stranger”的公司或个人签名的代码以下权限：读取名为question.txt的文件的权限。这个文件必须位于当前目录下，也就是这个应用程序的启动目录。因为在这条grant语句中没有提到代码库，所以来自所有代码库的代码，只要是由stranger签名的，都可以得到读question.txt的权限。注意，虽然stranger已经被允许读取question.txt中的问题，但是stranger不能看到answer.txt中的答案。而授予friend的权限正好相反，它可以读取所有的问题以及答案。

在策略文件中的第四条、也是最后一条语句仍然是一个grant语句：

```
grant codeBase "file:${com.artima.ijvm.cdrom.home}/security/ex2/*" {
    permission java.io.FilePermission "question.txt", "read";
    permission java.io.FilePermission "answer.txt", "read";
};
```

这条grant语句将两个权限授给所有从一个特定目录中装载的代码：对question.txt文件的读权限以及对answer.txt文件的读权限。这两个文件都必须在当前目录下，也就是这个应用程序的启动目录。注意，这个grant子句没有指明任何签名者，所以，这个代码可以是被任何人签名的，或者是未被签名的。只要它是从指定的目录下被装载的，它就可以被授予上面所列出的权限。

这条grant语句中的代码库URL采用了文件的形式：它包含了一个属性${com.artima.ijvm.cdrom.home}。如果要运行本章后面描述的AccessControl实例程序，必须将这个com.artima.ijvm.cdrom.home属性设置为本书附带的CD-ROM，或者是将这个CD-ROM移动后的目录。这个Policy对象是基于policyfile.txt的内容被实例化的，在它为这个grant子句的CodeSource构建URL时，将考虑com.artima.ijvm.cdrom.home属性。

3.10 保护域

当类装载器将类型装入Java虚拟机时，它们将为每个类型指派一个保护域。保护域定义了授予一段特定代码的所有权限。（一个保护域对应策略文件中的一个或多个grant子句。）装载入Java虚拟机的每一个类型都属于一个且仅属于一个保护域。

类装载器知道它装载的所有类或接口的代码库和签名者。它利用这些信息来创建一个CodeSource对象。它将这个CodeSource对象传递给当前Policy对象的getPermissions（）方法，得到这个抽象类java.security.PermissionCollection的子类实例。这个PermissionCollection包含了到所有Permission对象的引用（这些Permission对象由当前策略授予指定代码来源）。利用它创建的CodeSource和它从Policy对象得到的PermissionCollection，它可以实例化一个新的ProtectDomain对象。它通过将合适的ProtectionDomain对象传递给defineClass（）方法，来将这段代码放到一个保护域中。DefineClass（）方法是类ClassLoader的一个实例方法，用户自定义类装载器调用它来将类型导入到Java虚拟机中。将类型指派到保护域中是一个重要的工作，就像

在本章前面提到的一样，它是类装载器体系结构支持Java沙箱安全模型的三个方法中的一个。

虽然这个Policy对象代表了一个从代码来源到权限的全局映射，但是最终还是由类装载器负责决定代码执行时将获得什么样的权限。例如，一个类装载器可以完全忽略当前的策略，而随机地赋予权限。或者，一个类装载器可以向由Policy对象的getPermissions（）方法返回的权限中再添加一些权限。例如，如果一个类型装载器要装载一个applet代码，除了由当前策略可能授予这段代码的权限以外，它还可以添加一个权限，使得它可以建立一个到这个applet的源主机的socket连接。现在读者可以明白了，类装载器在装载类时起到了重要的安全作用。

图3-5用图形化的方式描述了保护域、代码来源以及权限。在图3-5中，来自friend.jar的代码在由policyfile.txt定义的策略下被装载以后，就看到了方法区和堆。friend.jar是CD-ROM中security/ex2/jars目录下的一个JAR文件，policyfile.txt是security/ex2目录下的一个ASCII策略文件。这个friend.jar文件包含了两个class文件：Friend.class和Friend$1.class。在本章前面提到的代码签名的例子中，这两个class文件都是由friend签名的。当这些类由类装载器定义时，它们将被放到一个保护域中，这个保护域的CodeSource对象说明了两件事。首先，这个CodeSource说明这些class文件是从一个本地JAR文件中被装载的，这个JAR文件的URL是：file:///fl/security/ex2/jars/friend.jar；其次，这个CodeSource说明这些class文件是由friend签名的，friend是一个和本地keystore中的一个证书相关联的别名。ProtectionDomain对象封装了一个到CodeSource对象的引用以及一个到java.security.Permissions对象的引用。java.security.Permissions是抽象类java.security.PermissionCollection的一个具体类，代表了一个同构权限的集合。这个Permissions对象持有指向两个java.io.FilePermission对象的引用。这两个FilePermissions指明了对当前目录下的question.txt文件和answer.txt文件的权限。

当一个类装载器将Friend和Friend$1导入方法区时，就像图3-5中表示的那样，类装载器将把一个ProtectionDomain对象的引用和这些class文件的字节传递给defineClass（）方法。这个defineClass（）方法将Friend和Friend$1所在的方法区中的类型数据和被传递的ProtectDomain对象相关联。图3-5中表示出了这个关联——包括箭头部分（该箭头在Friend和Friend$1所在的方法区中作为类型数据的一部分，表示到ProtectDomain对象的引用）。

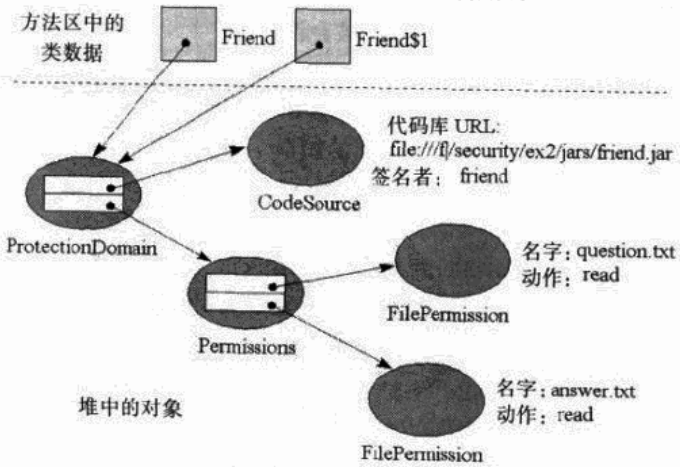


图3-5　保护域、代码来源以及权限

3.11 访问控制器

类java.security.AccessController提供了一个默认的安全策略执行机制，它使用栈检查来决定潜在不安全的操作是否被允许。这个访问控制器不能被实例化，它不是一个对象，而是集合在单个类中的多个静态方法。AccessController的最核心方法是它的静态方法checkPermission（），这个方法决定一个特定的操作能否被允许。这个方法将指向Permission对象的引用作为惟一的参数，并返回void。和安全管理器中的check方法相类似，如果AccessController确定这个操作被允许，它的checkPermission（）方法将简单地返回；但是如果AccessController确定一个操作被禁止，它的checkPermission（）方法将异常中止，并抛出一个AccessControlException，或者是它的一个子类。

在前面提到过，具体安全管理器中的老式check方法（例如checkRead（）和checkWrite（））的实现只是简单地实例化一个合适的Permission对象，并且调用具体安全管理器的checkPermission（）方法。这个具体安全管理器的checkPermission（）方法简单地调用了AccessController中的checkPermission（）方法。因此，如果你安装了具体安全管理器，其实最终是由这个AccessController来决定一个潜在不安全的方法是否被允许。

由AccessController的checkPermission（）实现的基本算法决定了调用栈中的每个帧是否有权执行潜在不安全的操作。每一个栈帧代表了由当前线程调用的某个方法，每一个方法是在某个类中定义的，每一个类又属于某个保护域，每个保护域包含一些权限。因此，每个栈帧间接地和一些权限相关。为了使传递给AccessController的checkPermission（）方法的Permission对象所代表的操作被允许，这个AccessController的基本算法要求，和调用栈上的每个帧相关联的权限必须包含或隐含传给checkPermission（）的Permission对象。

AccessController的checkPermission（）方法自顶向下检查栈，只要它遇到一个没有权限帧，它将抛出一个AccessControlException。通过抛出这个异常，AccessController指明这个操作不能被允许。相反，如果checkPermission（）方法到达栈的底部，也没有遇到这种栈帧（即无权限执行潜在不安全操作）的情况，checkPermission（）方法将简单地返回。通过简单返回而不是抛出异常，AccessController指明这个操作可以被允许。

由AccessController的checkPermission（）方法实现的真正的算法比这里描述的基本算法要稍微复杂一点。通过调用类AccessController的众多doPrivileged（）方法中的任何一个，程序就可以让AccessController中止它对栈的一帧一帧的扫描。这个doPrivileged（）方法将在本章后面详细介绍。

3.11.1 implies（）方法

为了决定由传递给AccessController的checkPermission（）方法的Permission对象所代表的操作，是否包含在（或隐含在）和调用栈中的代码相关联的权限中，AccessController利用了一个名为implies（）的重要方法。这个implies（）方法是在Permission类以及PermissionCollection类和ProtectionDomain类中声明的。implies（）将一个Permission对象作为它惟一的参数，返回一个布尔值true或false。Permission类的implies（）方法确定由Permission对象所代表的权限，是否在本质上隐含在由一个不同的Permission对象所代表的权限中。PermissionCollection和

ProtectionDomain的implies（）方法确认了一个被传递的Permission是否包含或隐含在封装在PermissionCollection或ProtectionDomain中的Permission对象集合中。

例如，读取/tmp目录下所有文件的权限本质上隐含了读取/tmp目录下特定文件/tmp/f的权限，但是反过来则不成立。如果你询问一个代表了读取/tmp目录下的所有文件的权限的FilePermission对象，它是否隐含了读取文件/tmp/f的权限，implies（）方法将返回true。但是如果你询问一个代表了读取/tmp/f 权限的FilePermission对象是否隐含了读取/tmp下任何文件的权限时，implies（）方法将返回false。

在CD-ROM中的security/ex1目录下，应用程序Example1示范了implies（）的这个意义：

```
import java.security.Permission;
import java.io.FilePermission;
import java.io.File;

// On CD-ROM in file security/ex1/Example1.java
class Example1 {

    public static void main(String[] args) {

        char sep = File.separatorChar;

        // Read permission for "/tmp/f"
        Permission file = new FilePermission(
            sep + "tmp" + sep + "f", "read");

        // Read permission for "/tmp/*", which
        // means all files in the /tmp directory
        // (but not any files in subdirectories
        // of /tmp)
        Permission star = new FilePermission(
            sep + "tmp" + sep + "*", "read");

        boolean starImpliesFile = star.implies(file);
        boolean fileImpliesStar = file.implies(star);

        // Prints "Star implies file = true"
        System.out.println("Star implies file = "
            + starImpliesFile);

        // Prints "File implies star = false"
        System.out.println("File implies star = "
            + fileImpliesStar);
    }
}
```

应用程序Example1创建了两个FilePermission对象，一个代表对特定目录的读权限，另一个代表对同一个目录下的特定文件的读权限。这个从局部变量star引用的FilePermission对象代表了读取/tmp下任何文件的权限，而从局部变量file引用的FilePermission对象代表了读取文件/tmp/f

的权限。在执行时，应用程序输出：

```
Star implies file = true
File implies star = false
```

implies（）方法被AccessController用来确定一个线程是否拥有进行某些操作的权限。例如，如果AccessController的checkPermission（）方法被调用，用以确定这个线程是否有权读取文件/tmp/f，AccessController将调用和这个线程的调用栈中的每个栈帧相关联的ProtectionDomain对象的implies（）方法。对于每个implies（）方法，AccessController将把一个FilePermission对象传递给它的checkPermission（）方法，这个FilePermission对象代表了读取文件/tmp/f的权限。每个ProtectionDomain对象的implies（）方法会调用它封装的PermissionCollection的implies（）方法，传递给它同一个FilePermission。同样，每一个PermissionCollection会调用它包含的Permission对象上的implies（）方法，再一次传递这个FilePermission对象的引用。一旦PermissionCollection的implies（）方法遇到了一个Permission对象，这个Permission对象返回了true，那么，这个PermissionCollection的implies（）方法也将返回true。只有当在PermissionCollection中包含的所有Permission对象的implies（）方法都没有返回true时，这个PermissionCollection才返回false。ProtectionDomain的implies（）方法简单地返回了PermissionCollection的implies（）方法的返回值。如果AccessController从与一个特定栈帧相关联的ProtectionDomain的implies（）方法中得到true时，这个栈帧所代表的代码就拥有了执行这个潜在不安全操作的权限。

3.11.2 栈检查示例

下面几节将给出几个示例，说明AccessController执行栈检查的方法。在下面的例子中，由friend和stranger签名的代码将在一定程度上被信任，但由friend签名的代码要比由stranger签名的代码可信度高。具体地说，由friend和stranger签名的代码都可以得到读取question.txt文件的权限，这个文件中包含了问题。由friend签名的代码可以得到读取answer.txt文件的权限，这个文件中包含了回答question.txt中的问题的答案，但是由stranger签名的代码则没有这个权限。在CD-ROM的security/ex2目录下的policyfile.txt文件中列出了这些授予friend和stranger的权限，这个文件在本章前面已经讲过。下面的每一个例子将采取policyfile.txt中描述的策略。

这些栈检查的示例都使用了实现接口Doer的类：

```
// On CD-ROM in file
// security/ex2/com/artima/security/doer/Doer.java
package com.artima.security.doer;

public interface Doer {

    void doYourThing();
}
```

为了成为Doer，类必须提供一个doYourThing（）方法的实现，实现Doer的类可以在它们的doYourThing（）方法中干任何它们喜欢的事。例如，这里有一个名为TextFileDisplayer的类，它实现了Doer，它做的“事”是显示一个文本文件的内容：

```
// On CD-ROM in file security/ex2/TextFileDisplayer.java

import com.artima.security.doer.Doer;
import java.io.FileReader;
import java.io.CharArrayWriter;
import java.io.IOException;

public class TextFileDisplayer implements Doer {

    private String fileName;

    public TextFileDisplayer(String fileName) {
        this.fileName = fileName;
    }

    public void doYourThing() {

        try {
            FileReader fr = new FileReader(fileName);

            try {
                CharArrayWriter caw = new
                CharArrayWriter();

                int c;
                while ((c = fr.read()) != -1) {
                    caw.write(c);
                }

                System.out.println(caw.toString());
            }
            catch (IOException e) {
            }
            finally {
                try {
                    fr.close();
                }
                catch (IOException e) {
                }
            }
        }
        catch (IOException e) {
        }
    }
}
```

当创建一个TextFileDisplayer对象时，必须将一个文件路径名传给它的构造器，这个TextFileDisplayer构造器将把这个路径名存储在名为filename的实例变量中。当调用这个TextFileDisplayer对象的doYourThing（）方法时，它将试图打开并读取这个文件的内容，并把它们打印到标准输出上。

doYourThing（）方法的另一个例子来自类Friend和类Stranger，它们在本章前面的代码签名的示例中已经出现过，可能读者已经忘记，在这里再提一次：

```
// On CD-ROM in file
// security/ex2/com/artima/security/friend/Friend.java
package com.artima.security.friend;
import com.artima.security.doer.Doer;
import java.security.AccessController;
import java.security.PrivilegedAction;

public class Friend implements Doer {

    private Doer next;
    private boolean direct;

    public Friend(Doer next,boolean direct) {
        this.next = next;
        this.direct = direct;
    }

    public void doYourThing() {

        if (direct) {

            next.doYourThing();
        }
        else {
            AccessController.doPrivileged(
                new PrivilegedAction() {
                    public Object run() {
                        next.doYourThing();
                        return null;
                    }
                }
            );
        }
    }
}

// On CD-ROM in file
// security/ex2/com/artima/security/stranger/Stranger.java
package com.artima.security.stranger;
```

```
import com.artima.security.doer.Doer;
import java.security.AccessController;
import java.security.PrivilegedAction;

public class Stranger implements Doer {

    private Doer next;
    private boolean direct;

    public Stranger(Doer next, boolean direct) {
        this.next = next;
        this.direct = direct;
    }

    public void doYourThing() {

        if (direct) {
            next.doYourThing();
        }
        else {
            AccessController.doPrivileged(
                new PrivilegedAction() {
                    public Object run() {
                        next.doYourThing();
                        return null;
                    }
                }
            );
        }
    }
}
```

Friend和Stranger中有很大部分是相同的，它们有一样的实例变量、构造器以及doYourThings（）方法，它们仅仅是所在的包以及名称不同。当创建一个新的Friend或Stranger对象时，必须向构造器传递一个布尔值和到另一个对象（它的类实现Doer接口）的引用。这个构造器将传递进来的Doer引用存放在实例变量next中，并将布尔值存放在实例变量direct中。当一个Friend对象或一个Stranger对象的doYourThing（）方法被调用时，这个方法直接或间接地调用next中包含的Doer引用的doYourThing（）方法。如果direct为true，Friend或Stranger仅仅直接调用next的doYourThing（）方法；否则，Friend或Stranger的doYourThing（）通过一个doPrivileged（）调用，间接调用next的doYourThing（）方法。

3.11.3 一个回答“是”的栈检查

在第一个栈检查示例中，先来看一个Example2a应用程序：

```
// On CD-ROM in file security/ex2/Example2a.java
import com.artima.security.friend.Friend;
import com.artima.security.stranger.Stranger;
```

```
// This succeeds because everyone has permission to
// read answer.txt
class Example2a {

    public static void main(String[] args) {

        TextFileDisplayer tfd = new TextFileDisplayer("question.txt");

        Friend friend = new Friend(tfd, true);

        Stranger stranger = new Stranger(friend, true);

        stranger.doYourThing();
    }
}
```

这个Example2a应用程序创建了三个Doer对象：TextFileDisplayer、Stranger和Friend。TextFileDisplayer构造器传递了字符串“question.txt”，当它的doYourThing（）方法被调用时，它将试着打开一个在当前目录中的名为question.txt的文件，以读取它的内容并打印到标准输出。给Friend对象的构造器传递了指向TextFileDisplayer对象（也是一个Doer对象）的引用以及true布尔值，因为这个被传递的布尔值为true，所以当Friend的doYourThing（）方法被调用时，它将直接调用TextFileDisplayer对象的doYourThing（）方法。给Stranger对象的构造器传递了指向Friend对象（也是一个Doer对象）的引用以及true布尔值。因为被传递的布尔值为true，所以当Stranger的doYourThing（）方法被调用时，它将直接调用Friend对象的doYourThing（）方法。创建完这三个Doer对象，并按照描述把它们挂在一起之后，Example2a的main（）方法就调用了Stranger对象的doYourThing（）方法，有趣的事开始了。

Example2a程序通过stranger引用变量调用Stranger对象的doYourThing（）方法，Stranger对象又调用了Friend对象的doYourThing（）方法，Friend对象又调用了TextFileDisplayer的doYourThing（）方法。TextFileDisplayer的doYourThing（）方法试图打开并读取当前目录下（Example2a应用程序的启动目录）的名为question.txt的文件，并将它的内容打印到标准输出。当TextFileDisplayer的doYourThing（）方法创建了一个新的FileReader对象时，FileReader的构造器创建了一个新的FileInputStream，FileInputStream的构造器检查是否已经安装了一个安全管理器。在现在这个例子中，已经安装了具体安全管理器，因此，FileInputStream的构造器调用了具体安全管理器的checkRead（）方法。这个checkRead（）方法实例化了一个新的、代表读文件question.txt权限的FilePermission对象，并将这个对象传给具体安全管理器的checkPermission（）方法，这个checkPermission（）方法又把这个对象传递给AccessController的checkPermission（）方法。AccessController的checkPermission（）方法执行了栈检查，确定这个线程是否有权打开并读取文件question.txt。

图3-6显示了当AccessController的checkPermission（）方法被调用时，调用栈的情况。在图3-6中，调用栈的每一个栈帧用由多个元素组成的一行表示。在每一个栈帧行的最左边，是一个标记为“类”的元素，它是一个类的全限定名，由栈帧表示的方法在这个类中定义。它右边的第二个元素被标记为“方法”，在这里给出了方法的名字。接下来的右边的元素被标记为“保护

域”，表示了这个栈帧所关联的保护域。行的最右边的元素是一个箭头，说明了当一个AccessController的checkPermission（）方法检查每一个栈帧是否有权执行被请求的操作时，它的行进方向。在箭头的左边是数字，每个栈帧对应一个。就像本书中每个显示栈的图一样，栈的顶显示在图的最底端。这样，在图3-6中，栈的顶是一个计数为10的栈帧。

类	方法	保护域	
Example2b	main()	CDROM	1
com.artima.security.stranger.Stranger	doYourThing()	STRANGER	2
com.artima.security.friend.Friend	doYourThing()	FRIEND	3
TextFileDisplayer	doYourThing()	CDROM	4
java.io.FileReader	<init>()	BOOTSTRAP	5
java.io.FileInputStream	<init>()	BOOTSTRAP	6
java.lang.SecurityManager	checkRead()	BOOTSTRAP	7
java.lang.SecurityManager	checkPermission()	BOOTSTRAP	8
java.security.AccessController	checkPermission()	BOOTSTRAP	9
java.security.AccessControlContext	checkPermission()	BOOTSTRAP	10

图3-6　Example2a中的栈检查：所有栈帧都有权限

图3-6中的栈图的“保护域”列表明，每一个栈帧都和下面四个保护域中的一个相关联：“FRIEND”、“STRANGER”、“CD-ROM”以及“BOOTSTRAP”。这些保护域中的三个和policyfile.txt文件中的grant子句相对应。和FRIEND保护域相关联的grant子句赋予所有由friend签名的代码读取文件question.txt和文件answer.txt的权限；和STRANGER相关联的grant子句赋予所有由stranger签名的代码读取question.txt的权限；和CD-ROM相关联的grant子句赋予由目录“S{com.artima.ijvm.cdrom.home}/security/ex2”装载的代码读取文件question.txt和文件answer.txt的权限。第四个保护域名为BOOTSTRAP，它也是最后一个保护域，它不与policyfile.txt中的任何grant子句相对应，而是代表赋予所有由启动类装载器装载的代码的权限，启动类装载器负责装载Java API的class文件。在BOOTSTRAP保护域中的代码被赋予了java.lang.AllPermission，该权限允许做任何事。

为了使Example2a应用程序示范栈检查的过程，必须用一条合适的命令启动这个应用程序。当在Java 2 SDK版本1.2中使用这个应用程序时，这条命令采用以下形式：

```
java -Djava.security.manager -Djava.security.policy=
policyfile.txt -Dcom.artima.ijvm.cdrom.home=d:\books\
InsideJVM\manuscript\cdrom -cp
.;jars/friend.jar;jars/stranger.jar Example2a
```

这条命令被包含在CD-ROM中的security/ex2目录下的ex2a.bat文件中，它是用于让例子开始工作的一种命令。通过在命令行定义java.security.manager属性，说明具体安全管理器必须被自动安装。因为Example2a应用程序没有显式安装安全管理器，所以如果在命令行没有定义java.security.manager属性，那么就不会安装任何安全管理器，而代码也就可以做任何事。-cp参数设置了类路径，使得虚拟机在当前目录、friend.jar以及jar子目录下的stranger.jar文件中查找class文件。com.artima.ijvm.cdrom.home属性指出了Doer、Example2a以及TextFileDisplayer被放置在哪个目录下。这个属性被policyfile.txt中的第三个grant子句使用，这个子句和名为“CD-ROM”的

保护域相对应。因此，类型Doer、Example2a以及TextFileDisplayer将被装载到CD-ROM保护域，并且被赋予读取question.txt和answer.txt的权限。要在系统上执行Example2a，必须将com.artima.ijvm.cdrom.home属性设置为CD-ROM上的security/ex2目录，或者是将CD-ROM上的目录security/ex2移动后的任何目录。

当AccessController执行它的栈检查时，它从栈顶栈帧10开始，逐个向下到栈帧1，栈帧1是由线程调用的第一个方法——类Example2a的main（）。在Example2a应用程序的例子中，调用栈的每一个栈帧都有权执行读取文件question.txt的操作。这是因为在调用栈上表示的四个保护域——FRIEND、STRANGER、CD-ROM和BOOTSTRAP——都包含或隐含了读取当前目录下的question.txt的FilePermission。当AccessController的checkPermission（）方法到达了栈底，而且没有遇到任何栈帧无权读取这个文件的情况，它将正常返回，而不抛出任何异常。FileInputStream将继续执行并打开文件进行读取。Example2a应用程序读取question.txt的内容并将它们打印到标准输出，看起来就像下面这样：

```
To what extent does complexity threaten security?
```

3.11.4 一个回答“不”的栈检查

下面一个栈检查的例子，是没有权限情况下的栈检查，下面看CD-ROM中security/ex2目录下的Example2b应用程序：

```
// On CD-ROM in file security/ex2/Example2b.java
import com.artima.security.friend.Friend;
import com.artima.security.stranger.Stranger;

// This fails because the Stranger code doesn't have
// permission to read file question.txt

class Example2b {

    public static void main(String[] args) {
TextFileDisplayer tfd = new TextFileDisplayer("answer.txt");

        Friend friend = new Friend(tfd, true);

        Stranger stranger = new Stranger(friend, true);

        stranger.doYourThing();
    }
}
```

Example2b和以前的例子Example2a只有一个不同，Example2a中是将名字为“question.txt”的文件传递给TextFileDisplayer构造器，而Example2b则是传递名字为“answer.txt”的文件。应用程序中的这个小小改动将导致程序最后结果的重大变化，因为在栈中，有一个方法没有访问answer.txt的权限。

当Example2b程序调用由stranger变量引用的Stranger对象上的doYourThing（）方法时，这

个Stranger对象又调用了Friend对象的doYourThing（）方法，而Friend对象又调用了TextFileDisplayer对象的doYourThing（）方法。TextFileDisplayer的doYourThing（）方法试图在当前目录（Example2b应用程序的启动目录）打开并读取一个名为answer.txt的文件，并将它的内容打印到标准输出。当TextFileDisplayer的doYourThing（）方法创建了一个新的FileReader对象，FileReader的构造器创建了一个新的FileInputStream，FileInputStream的构造器检查是否已经安装了安全管理器。在本例的情况中，已经安装了一个具体安全管理器，所以FileInputStream的构造器调用了具体安全管理器的checkRead（）方法。这个checkRead（）方法实例化了一个新的、代表读取文件answer.txt权限的FilePermission对象，并且将这个对象传递给具体安全管理器的checkPermission（）方法，checkPermission（）方法又把这个对象传递给AccessController的checkPermission（）方法。AccessController的checkPermission（）方法执行栈检查，以确定这个线程是否有权打开文件answer.txt并读取其内容。

图3-7中显示了在Example2b中被检查的调用栈，这个栈看起来和Example2a中的调用栈相似，惟一的不同在于，在Example2b中，AccessController将确认栈中的每一个栈帧是否拥有读取answer.txt的权限，而不是检查是否拥有读取question.txt的权限。在通常情况下，栈检查总是从栈顶开始，并且逐个向下直到栈帧1。但是在这个例子中，这个检查的过程并没有真正到达栈帧1，当AccessController到达栈帧2时，它发现栈帧2的doYourThing（）方法属于Stranger类的代码，而这个类没有读取answer.txt文件的权限。因为，必须要栈中所有的栈帧都拥有这个权限，所以当栈检查过程到达栈帧2时，就没有必要再进行下去了。AccessController的checkPermission（）方法抛出了一个AccessControllerException。

类	方法	保护域	
Example2b	main()	CDROM	1
com.artima.security.stranger.Stranger	doYourThing()	STRANGER	2
com.artima.security.friend.Friend	doYourThing()	FRIEND	3
TextFileDisplayer	doYourThing()	CDROM	4
java.io.FileReader	<init>()	BOOTSTRAP	5
java.io.FileInputStream	<init>()	BOOTSTRAP	6
java.lang.SecurityManager	checkRead()	BOOTSTRAP	7
java.lang.SecurityManager	checkPermission()	BOOTSTRAP	8
java.security.AccessController	checkPermission()	BOOTSTRAP	9
java.security.AccessControlContext	checkPermission()	BOOTSTRAP	10

图3-7　Example2b的栈检查：栈帧2没有权限

为了使Example2b应用程序如预期一般执行，必须用一个合适的命令启动这个应用程序。当在Java 2 SDK 1.2中使用这个Java程序时，这个命令采取以下形式：

```
java -Djava.security.manager -Djava.security.policy=policyfile.txt -
Dcom.artima.ijvm.cdrom.home=d:\books\InsideJVM\manuscript\cdrom -
cp.;jars/friend.jar;jars/stranger.jar Example2b
```

这个命令包含在CD-ROM中security/ex2目录下的ex2b.bat文件中，它是用于让例子开始工作的一种命令。就像以前一样，要在自己的系统上运行Example2b，必须将com.artima.ijvm. cdrom .home属性设置为CD-ROM的securtiy/ex2目录，或者是将CD-ROM中的security/ex2拷贝后的位置。当运行这个程序后，将看到以下输出：

```
Exception in thread "main" java.security.
AccessControlException: access denied (java.io.
FilePermission answer.txt read)
    at java.security.AccessControlContext.
    checkPermission(AccessControlContext. java:195)
    at java.security.AccessController.checkPermission
    (AccessController.java:403)
    at java.lang.SecurityManager.checkPermission
    (SecurityManager.java:549)
    at java.lang.SecurityManager.checkRead
    (SecurityManager.java:873)
    at java.io.FileInputStream.<init>(FileInputStream. java:65)
    at java.io.FileReader.<init>(FileReader.java:35)
    at TextFileDisplayer.doYourThing(TextFileDisplayer. java, Compiled Code)
    at com.artima.security.friend.Friend.doYourThing
    (Friend.java:21)
    at com.artima.security.stranger.Stranger.doYourThing
    (Stranger.java:21)
    at Example2b.main(Example2b.java:18)
```

3.11.5 doPrivileged ( ) 方法

到现在为止，在本章中给出的基本算法中，AccessController自顶向下对栈进行检查，严格地要求每一个栈帧都有执行某个操作的权限，以防一段不可靠的代码隐藏在一段可信任代码后面。因为AccessController一路向下地查看调用栈，所以，它最终会找到任何不能被允许执行被请求操作的方法。例如，虽然Example2b中不被信任的Stranger对象将可信任的Friend和TextFileDisplayer的代码放置它和在Java API方法的中间，其中，Java API的方法试图打开文件answer.txt，但不被信任的Stranger代码仍然不能藏在被信任的代码后。就像图3-7所示，虽然AccessController在到达栈帧2之前，必须检查八个栈帧是否拥有读取answer.txt文件的权限，它最终还是到达了栈帧2。一旦它到达了栈帧2，它将会发现和Stranger类的doYourThing ( ) 方法相关联的保护域没有读取answer.txt文件的权限。因此，这个AccessController将抛出一个AccessControllerException，从而禁止了读取操作。

这个基本的AccessController算法防止了任何代码执行（或导致执行）任何不可信任的代码。因此，属于一个权限较少的保护域的方法无权调用属于权限更高的保护域的方法。这个基本算法同时隐含了，如果一个属于较高权限保护域中的方法调用了属于较低权限保护域中的方法，它必须自动放弃某些权限。虽然这个基本算法提供了一些一般情况下不需要的操作，但是AccessController严格坚持在调用栈中的所有栈帧都必须含有执行被请求操作的权限，而这一点在很多情况下都太苛刻了。有的时候，调用栈较上层（更靠近栈顶）的代码可能希望执行一段代码，而这段代码在调用栈的较下层是不允许执行的。例如，假设一个不可靠的applet请求Java API用bold Helvetica字体在它的applet面板中显示一个文本字符串。为了满足这个请求，Java API可能需要打开一个本地硬盘上的字体文件，以装载applet显示文本所需要的bold Helvetica字体。一个类显式地请求打开这个字体文件，因为它属于Java API，所以它可能拥有打开这个文件

的权限。但是，这个不被信任的applet的代码是用调用栈下层的一个栈帧表示的，它很可能并不拥有打开这个文件的权限。如果用这个基本的算法，AccessController将禁止打开这个字体文件，因为这个不被信任的applet的代码在栈中的没有打开这个文件的权限。

为了使可信的代码执行较不可靠的代码操作（这段不可靠的代码位于调用栈的较下层且没有执行这个操作的权限），AccessController类重载了四个名为doPrivileged（）的静态方法。这个四个方法的每一个都接受一个实现java.security.PrivilegedAction接口或者java.security.PrivilegedExceptionAction接口的对象作为它的参数。这两个接口都声明了一个名为run（）的方法，并且没有任何参数，返回void。这两个接口的惟一不同在于，PrivilegedExceptionAction的run（）方法在它的throws子句中声明了Exception，而PrivilegedAction则没有声明throws子句。要在调用栈较下层有不可信任代码的情况下执行某个操作，必须创建一个对象，它实现了一个PrivilegedAction接口，它的run（）方法执行了这个操作，并且要将这个对象传递给doPrivileged()。

当调用doPrivileged（）方法时，就像调用其他任何方法一样，都会将一个新的栈帧压入栈。在由AccessController执行的栈检查中，一个doPrivileged（）方法调用的栈帧标识了检查过程的提前终止点。如果和调用doPrivileged（）的方法相关联的保护域拥有执行被请求操作的权限，AccessController将立即返回。这样这个操作就被允许，即使在栈下层的代码可能没有执行这个操作的权限。

如果一个不可靠的applet请求Java API在它的applet面板上显示一个测试字符串，Java API代码通过把文件打开操作包装在一个doPrivileged（）调用中，就可以打开本地字体文件。AccessController将允许这样的请求，即使这个不被信任的applet代码没有打开这个文件的权限。因为这个不被信任的applet代码的栈帧位于由Java API代码调用的doPrivileged（）栈帧的下方，AccessController甚至不必考虑不被信任的applet代码的权限。

作为doPrivileged（）方法调用的一个例子，再来看一下Friend类的doYourThing（）方法：

```
// On CD-ROM in file
// security/ex2/com/artima/security/friend/Friend.java
package com.artima.security.friend;
import com.artima.security.doer.Doer;
import java.security.AccessController;
import java.security.PrivilegedAction;

public class Friend implements Doer {

    private Doer next;
    private boolean direct;

    public Friend(Doer next, boolean direct) {
        this.next = next;
        this.direct = direct;
    }

    public void doYourThing() {
```

```
        if (direct) {

            next.doYourThing();
        }
        else {
            AccessController.doPrivileged(
                new PrivilegedAction() {
                    public Object run() {
                        next.doYourThing();
                        return null;
                    }
                }
            );
        }
    }
}
```

如果direct实例变量为true，Friend的doYourThing（）方法将简单地直接调用next引用的doYourThing（）方法。但是如果direct为false，doYourThing（）将把next引用的doYourThing（）调用包装在一个doPivileged（）调用中。为了这么做，Friend实例化了一个匿名内部类，这个类实现了PrivilegedAction，它的run（）方法调用了next的doYourThing（），并将这个对象传递给doPrivileged（）。

下面看一下CD-ROM中security/ex2目录下的Example2c应用程序，看一看Friend的doPrivileged（）的调用过程：

```
// On CD-ROM in file security/ex2/Example2c.java
import com.artima.security.friend.Friend;
import com.artima.security.stranger.Stranger;

// This succeeds because Friend code executes a
// doPrivileged() call. (Passing false as
// the second arg to Friend constructor causes
// it to do a doPrivileged().)

class Example2c {

    public static void main(String[] args) {

TextFileDisplayer tfd = new TextFileDisplayer ("answer.txt");

        Friend friend = new Friend(tfd, false);

        Stranger stranger = new Stranger(friend, true);

        stranger.doYourThing();
    }
}
```

Example2c应用程序中的main（）方法和以前的例子Example2b中的main（）方法之间只有

一个不同。应用程序Example2b实例化了一个Friend对象，它传递了true作为第二个参数，而Example2c则传递了false。如果回到本章前面去看Friend（和Stranger）的代码，就会发现这个参数用于判断，是否直接调用Doer（它作为传递给构造器的第一个参数）的doYourThing（）方法。因为Example2c传递了false，Friend类将不会直接调用doYourThing（），而会通过一个AccessController.doPrivileged（）调用来间接地调用它。

当Example2c程序调用由stranger变量引用的Stranger对象的doYourThing（）方法时，Stranger对象调用了Friend对象的doYourThing（）方法，Friend（因为direct为false）又调用了doPrivileged（），它传递了实现PriviligedAction的匿名内部类实例。这个doPrivileged（）方法调用了被传递的PrivilegedAction对象的run（）方法，而PrivilegeAction又调用了TextFileDisplayer对象的doYourThing（）方法。

就像在前面的例子中一样，TextFileDisplayer的doYourThing（）方法试图打开并读取当前目录下名为answer.txt的文件，并将它的内容打印到标准输出。当TextFileDisplayer的doYourThing（）方法创建了一个新的FileReader对象，FileReader构造器创建了一个新的FileInputStream（它的构造器检查，确定是否已经安装了安全管理器）。在这里，再一次安装了具体安全管理器，所以FileInputStream的构造器调用了具体安全管理器的checkRead（）方法。这个checkRead（）方法实例化了一个新的、代表了读取文件answer.txt权限的FilePermission对象，并把这个对象传递给具体安全管理器的checkPermission（）方法，这个checkPermission（）方法又把这个对象传递给AccessController的checkPermission（）方法。AccessController的checkPermission（）方法执行栈检查，以确定这个线程是否有权打开文件answer.txt并读取其内容。这个栈在图3-8中给出。

类	方法	保护域	
Example2b	main()	CDROM	1
com.artima.security.stranger.Stranger	doYourThing()	STRANGER	2
com.artima.security.friend.Friend	doYourThing()	FRIEND	3
java.security.AccessController	dPrivileged()	BOOTSTRAP	4
com.artima.security.friend.Friend$1	run()	FRIEND	5
TextFileDisplayer	doYourThing()	CDROM	6
java.io.FileReader	<init>()	BOOTSTRAP	7
java.io.FileInputStream	<init>()	BOOTSTRAP	8
java.lang.SecurityManager	checkRead()	BOOTSTRAP	9
java.lang.SecurityManager	checkPermission()	BOOTSTRAP	10
java.security.AccessController	checkPermission()	BOOTSTRAP	11
java.security.AccessControlContext	checkPermission()	BOOTSTRAP	12

图3-8　Example2c的栈检查：在栈帧3停止

在Example2c中要被检查的调用栈看起来很像在Example2a和Example2b中检查的调用栈，区别在于Example2c的调用栈由两个另外的栈帧：栈帧4代表了doPrivileged（）调用，栈帧5代表了PrivilegedAction对象的run（）调用。在通常情况下，栈检查从栈顶开始一路向下直到栈帧1。当AccessController到达栈帧4时，它发现了一个doPrivileged（）调用。因此，AccessController又进行了一个检查：它检查由栈帧3代表的代码，也就是调用了doPrivileged（）的代码，是否有读取文件answer.txt的权限。因为栈帧3是与FRIEND保护域相关联的，而FRIEND保护域拥有读取文件question.txt的权限，所以AccessController的checkPermission（）方法正常返回。因为

AccessController在栈帧3停止了它的检查，它就不再考虑栈帧2，即使栈帧2是和STRANGER保护域相关联的，而STRANGER保护域没有读取文件answer.txt的权限。这样，通过调用doPrivileged（），Friend代码可以读取文件anwser.txt，即使调用栈中在它下方的代码没有打开这个文件的权限。

为了使Example2c应用程序如预期一般正常工作，必须用一个合适的命令启动应用程序，就像在前面的例子中一样。当在Java 2 SDK版本1.2中使用Java程序时，这个命令采取以下形式：

```
java -Djava.security.manager -Djava.security.policy=policyfile.txt
-Dcom.artima.ijvm.cdrom.home=d:\books\InsideJVM\manuscript\cdrom -
cp.;jars/friend.jar;jars/stranger.jar Example2c
```

这个命令包含在CD-ROM中security/ex2目录下的ex2c.bat文件中，它是一个用于启动例子工作的命令的例子。就像以前一样，要在自己的系统上执行Example2c，必须将com.artima.ijvm.cdrom.home属性设置为CD-ROM的security/ex2目录，或者设置为将CD-ROM中security/ex2目录拷贝后的位置。当运行这个程序时，它将打印answer.txt的内容：

```
Complexity threatens security to a significant extent. The more
complicated a security infrastructure becomes, the more likely
parties responsible for configuring security will either make
mistakes that open up security holes or avoid using the
security infrastructure altogether.
```

3.11.6 doPrivileged（）的一个无效使用

有一点很重要，必须理解，那就是一个方法不能授予它自己比它现在已经用doPrivileged（）调用所得到的权限更多的权限。通过调用doPrivileged（），一个方法仅仅能使用它现在已经被授予的权限。它告诉AccessController它实现其权限的职责，这样AccessController就应该忽略它的调用者的权限。因此在上面的例子Example2c中，doPrivileged（）调用允许读取文件answer.txt，因为执行doPrivileged（）的类Friend已经拥有读取这个文件的权限，并且在这个栈中所有上层的栈帧都拥有了这个权限。

作为一个doPrivileged（）的无效使用的例子，下面来看一下CD-ROM中security/ex2目录下的Example2d应用程序：

```
// On CD-ROM in file security/ex2/Example2d.java
import com.artima.security.friend.Friend;
import com.artima.security.stranger.Stranger;

// This fails because even though Stranger does
// a doPrivileged() call, Stranger doesn't have
// permission to read question.txt. (Passing
// false as second arg to Stranger constructor
// causes it to do a doPrivileged().)

class Example2d {

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

```
        TextFileDisplayer tfd = new TextFileDisplayer("answer.txt");

        Stranger stranger = new Stranger(tfd, false);

        Friend friend = new Friend(stranger, true);

        friend.doYourThing();
    }
}
```

Example2d和以前的例子Example2c的不同在于，Stranger和Friend对象已经互换了位置和角色。现在对象在栈的较上层，而Friend在栈的较下层。这一次，是由Stranger调用了doPrivileged（），而不是Friend。

当Example2d程序调用了由friend变量引用的Friend对象的doYourThing（）方法时，Friend对象调用了Stranger对象的doYourThing（）方法，Stranger（因为direct为false）又调用了doPrivileged（），它传递了实现PrivilegedAction的匿名内部类实例。这个doPrivileged（）方法调用了被传递的PrivilegedAction对象的run（）方法，run（）方法又调用了TextFileDisplayer对象的doYourThing（）方法。

就像在前面的两个例子中一样，TextFileDisplayer的doYourThing（）方法试图在当前目录下打开并读取名为answer.txt的文件，并将它的内容打印到标准输出。当TextFileDisplayer的doYourThing（）方法创建了一个新的FileReader对象时，FileReader的构造器创建了一个新的FileInputStream，它的构造器检查、确定是否已经安装了安全管理器。就像在前面所有的例子中一样，这里已经安装了一个具体安全管理器，所以这个FileInputStream的构造器调用了具体安全管理器上的checkRead（）方法。这个checkRead（）方法实例化了一个新的、代表读取文件answer.txt权限的FilePermission对象，并将这个对象传递给具体安全管理器的checkPermission（）方法。checkPermission（）方法又将这个对象传递给AccessController的checkPermission（）方法。AccessController的checkPermission（）方法执行栈检查以确定这个线程是否被允许打开文件answer.txt并读取其内容。图3-9中显示了代表Example2d中AccessController的栈。

类	方法	保护域	
Example2b	main()	CDROM	1
com.artima.security.stranger.Friend	doYourThing()	FRIEND	2
com.artima.security.friend.Stranger	doYourThing()	**STRANGER**	3
java.security.AccessController	dPrivileged()	BOOTSTRAP	4
com.artima.security.friend.Stranger$1	run()	**STRANGER**	5
TextFileDisplayer	doYourThing()	CDROM	6
java.io.FileReader	<init>()	BOOTSTRAP	7
java.io.FileInputStream	<init>()	BOOTSTRAP	8
java.lang.SecurityManager	checkRead()	BOOTSTRAP	9
java.lang.SecurityManager	checkPermission()	BOOTSTRAP	10
java.security.AccessController	checkPermission()	BOOTSTRAP	11
java.security.AccessControlContext	checkPermission()	BOOTSTRAP	12

图3-9　Example2d的栈检查：栈帧5没有权限

在Example2d中要检查的调用栈看起来和Example2c中要检查的调用栈很相像，惟一的不同

在于Friend和Stranger互换了位置。通常情况下，栈检查从栈顶开始一路向下直到栈帧1。但同理，栈检查不会真的到达栈帧1。当AccessController到达栈帧5，它发现这个栈帧和STRANGER保护域相关联，而STRANGER保护域没有读取文件answer.txt的权限。在这种情况下，AccessController抛出一个AccessControlException，说明请求读取answer.txt的操作不能被执行。

如果类Stranger能得到某些实现PrivilegedAction的类的实例的帮助，执行了TextFile Displayer的doYourThing（）方法的调用，并且所属的保护域拥有读取文件answer.txt的权限，那么，在doPrivileged（）帮助下，Stranger试图打开文件answer.txt的操作仍然是无效的。例如，试想一下，由Example2d调用栈的栈帧5所代表的run（）方法的代码是和CD-ROM保护域相关联的。在这种情况下，AccessController将确定栈帧5拥有打开文件answer.txt的权限，并将继续检查栈帧4。在栈帧4中，AccessController会发现doPrivileged（）调用，因此，AccessController将进行另一个检查：它将确认调用doPrivileged（）的方法，也就是由栈帧3所代表的Stranger的doYourThing（）方法，是否拥有读取文件answer.txt的权限。因为栈帧3是和STRANGER保护域相关联的，而STRANGER保护域没有读取文件answer.txt的权限，所以AccessController将抛出一个AccessControlException。

为了使Example2d应用程序能够如预期那样正常工作，必须用另一条合适的命令启动这个应用程序。当在Java 2 SDK 1.2中使用Java程序时，命令采取以下形式：

```
java -Djava.security.manager -Djava.security.policy=policyfile.txt -
Dcom.artima.ijvm.cdrom.home=d:\books\InsideJVM\manuscript\cdrom -
cp.;jars/friend.jar;jars/stranger.jar Example2d
```

这个命令包含在CD-ROM中security/ex2目录下的ex2d.bat文件中，它是一个用于启动例子工作的命令的例子。像以前一样，要在自己的系统上执行Example2d，必须将com.artima.ijvm.cdrom.home属性设置为CD-ROM的security/ex2目录，或者设置为将CD-ROM的security/ex2目录拷贝后的所在位置。当运行这个程序时，你将看到黑客在任何地方都不想看到的输出：

```
Exception in thread "main" java.security.AccessControlException: access
denied (java.io.FilePermission answer.txt read)
    at java.security.AccessControlContext.checkPermission
    (AccessControlContext.java:195)
    at java.security.AccessController.checkPermission(AccessController.java:403)
    at java.lang.SecurityManager.checkPermission(SecurityManager.java:549)
    at java.lang.SecurityManager.checkRead(SecurityManager.java:873)
    at java.io.FileInputStream.<init>(FileInputStream.java:65)
    at java.io.FileReader.<init>(FileReader.java:35)
    at TextFileDisplayer.doYourThing(TextFileDisplayer.java, Compiled Code)
    at com.artima.security.stranger.Stranger$1.run(Stranger.java:27)
    at java.security.AccessController.doPrivileged(Native Method)
    at com.artima.security.stranger.Stranger.doYourThing(Stranger.java:24)
    at com.artima.security.friend.Friend.doYourThing(Friend.java:21)
    at Example2d.main(Example2d.java:21)
```

3.12 Java安全模型的不足和今后的发展方向

Java的安全模型虽然影响深远，但是并没有解决由移动代码所引起的所有潜在的威胁。例如，目前Java的安全模型还不能解决恶意移动代码的两个可能的活动：

- 不断分配内存直到内存耗尽。
- 不断生成线程导致每件事都慢得不可忍受。

这种类型的攻击被称为拒绝服务（DOS），因为这种攻击是拒绝终端用户使用他们自己的计算机。Java的安全模型现在还没有办法对不可靠代码使用线程或内存进行限制。试图阻止这种恶意代码的困难在于无法识别出它们，例如一个分配了许多内存的恶意applet和一个做正常工作的图像处理applet之间的区别。然而，这种类型的攻击在某些情况下是一个严重的问题，例如对于运行Java servlet的承担重要任务的服务器来说。

另一个目前没有放入安全模型的领域是关于将权限映射到系统用户，代码以这个用户的名义来运行。这种访问控制的一个较为常见的例子就是UNIX操作系统，它基于用户ID对文件的访问进行控制，这种访问只能通过一个正确的登录名和密码来获得。因为这种访问控制在分布式系统中非常重要，例如在Jini中实现的，Sun正在积极努力，争取将这种以用户为中心的安全功能加入到Java中。Java认证和授权服务（JAAS）的目的在于使得访问控制不仅仅基于授予代码库和签名者的权限，还能基于授予用户（那些执行这些代码的用户）的权限。

3.13 和体系结构无关的安全性

为了保证有效，一个计算机或网络的安全策略必须是全面的，它不能仅仅包含运行下载的Java代码的沙箱。例如，从Internet上下载并在你的计算机上运行的Java applet不能读取自己的最机密业务计划的Word处理文件，假若发生了以下事情，那么安全模型就根本不起作用：

- 从Internet上例行公事地下载本地可执行程序，不管它们是不是可信任的就轻易运行它们。
- 丢弃业务计划的打印件时，没有切碎它们。
- 离开时没有关门。
- 如果你雇用了某人帮助你，而实际上这个人是你的对手派来的间谍。

然而，在一个全面的安全策略环境中，Java的安全模型有着重要的作用。

Java安全模型的优点在于，你一旦安装了它，它将为你做大部分的工作。你不必担心一个特定的程序是否是可靠的——Java运行时将为你决定；如果它是不可靠的，Java运行时会将这段不可靠代码封闭在沙箱中，从而保护了你的财产。问题在于，虽然Java安全体系结构的设计者力图使保护工作尽量简单，但由于安全体系结构提供的高度功能性和灵活性的需要，体系结构仍然比较复杂。就像在文件answer.txt中提到的那样，本章前面给出的AccessController例子中，类Stranger非常想要读取文件，复杂本身就给安全带来了隐患。一个安全体系结构越复杂，负责安全配置的团体就越有可能犯错，从而导致打开一个安全漏洞，或者这个团体干脆完全拒绝使用这个安全体系结构。

Java软件的终端用户不能仅仅依靠内置在Java体系结构中的安全机制，他们必须采取一套全面的、适合于他们真正安全需要的安全策略。同样，Java技术的安全策略本身并不仅仅依靠本章

中所描述的体系结构安全机制。例如，Java安全策略的一个方面就是，任何人都可以签署一个许可协议，从而得到Sun的Java平台实现的源代码。Java安全体系结构的内部实现向任何想探究它的人开放，而不是将它作为一个秘密的“黑箱”。这就鼓励了安全方面的专家寻找好的技术，来试图寻找这个实现中的安全漏洞。当他们发现安全漏洞时，就可以打补丁。这样，Java内部实现的开放性就成为Java整个安全策略的一部分。除了开放性，Java的整个安全策略还有一些其他方面和它的体系结构没有直接联系。读者可以访问资源页得到更多关于Java整个安全策略的信息。

3.14 资源页

要了解更多关于Java安全性的信息，请访问资源页： http://www.artima.com/insidejvm/resources/。

第4章　网络移动性

前面两章讨论了Java是如果处理网络计算环境下软件开发人员所面临的两大挑战的。平台无关性是一个挑战，因为同一网络中通常连接了多种不同的计算机和设备。而安全模式也是一个挑战，因为网络可以方便地传输病毒和其他形式的恶意代码。本章不是描述Java体系结构如何处理这些挑战，而是Java如何把握网络所带来的巨大机遇。

为什么说Java是网络软件环境下的一个有用工具呢？一个主要原因就是Java的体系结构使软件的网络移动性成为可能。实际上，也正是这个原因，Java技术在软件行业中才被看做是移动技术的新范型。本章分析了网络上移动的软件，以及Java是如何使这种移动性成为可能的。

4.1　为什么需要网络移动性

在个人电脑流行之前，占主要地位的计算模式是服务于多个终端用户的大型计算机系统。大型主机利用分时技术分别关注从哑终端登录到主机的多个终端用户。软件应用程序存储在主机的磁盘上，使用户不仅可以共享同一个CPU，而且可以共享同样的应用程序。这种模式有个缺点，如果某个用户运行的作业占用了大量CPU资源的话，其他用户的性能就会大受影响。

微处理器的出现推动了个人计算机的蓬勃发展。硬件的这种变化引起了软件的相应变化，各用户不再共享存储在主机上的软件应用程序，而是在各自的PC机上拥有自己的软件拷贝。每个用户在自己专用的CPU上运行软件，因此，这种计算模式解决了多用户共享同一主机CPU时间所带来的问题。

最初，个人计算机像一些独立的孤岛一样分别进行计算。居统治地位的软件模式是在孤立的个人计算机上运行孤立的软件。但是很快，个人计算机开始互联成网。因为个人计算机只为自己的用户服务，它解决了大型计算机系统中CPU分时的难题。但除非这些个人计算机连接成网络，否则它们就不能像大型计算机系统那样使多个用户共享集中存储的数据了。

当个人计算机互联成网变得越来越普遍的时候，另一种软件模式日益重要起来，即客户机/服务器模式。客户机/服务器模式将任务分为两部分，分别运行在两种计算机上：客户端进程运行在终端用户的个人计算机上，而服务器端进程运行在同一网络的另一台计算机上。客户端和服务器端的进程通过网络来回发送数据进行传输。服务器端进程通常只是简单地接受网络中客户端发来的数据请求命令，从中央数据库中提取需要的数据，并将该数据发送给客户端。而客户端在接到数据后，进行处理，然后显示并允许用户操纵数据。这样的模式允许个人计算机的终端用户读取并操作放在中央存储仓库的数据，而不需强迫这些用户共享中央CPU来处理数据。终端用户的确是共享了运行服务器端进程的CPU，但在一定程度上，数据处理是由客户端完成的，因此服务器端CPU的负载大大减轻了。

很快，客户机/服务器模式中就不止包括两个处理器了。最初它被称做两层客户机/服务器模式：一层是客户端，另一层是服务器。更复杂一些的模型叫做三层（表示有三个进程）、四层

（表示有四个进程）或者N层结构——层次结构越来越多了。最后，当更多的进程加入计算时，客户端和服务器的区别模糊了，于是人们开始使用这样一个名词——分布式处理——来涵盖所有这些结构模式。

分布式处理模式综合了网络和处理器发展的优点，将进程分布在多个处理器上运行，并允许这些进程共享数据。尽管这种模式有许多大型计算机系统所无法比拟的优势，但它也有个不可忽视的缺点：分布式处理比大型计算机系统更难管理。在大型计算机系统中，软件应用程序存储在主机的磁盘上，虽然可以有多个用户使用该软件，但它只需在一个地方安装和维护。升级一个软件后，所有用户在下一次登录并启动该软件的时候都可以得到这个新的版本。但是相反，在分布式系统中，不同组件的软件往往存储在不同的磁盘上，因此，系统管理员需要在分布式系统的不同组件上安装和维护软件。要升级一个软件时，管理员不得不分别升级每台计算机上的这个软件。所以，分布式处理的系统管理比大型计算机系统要困难得多。

Java的体系结构使软件的网络移动性成为可能，同时也预示了一种新的计算模式的到来。这种新的模式建立在流行的分布式处理模式的基础上，并可以将软件通过网络自动传送到各台计算机上。这样就解决了分布式处理系统中系统管理的难题。例如，在一个客户端/服务器系统中，客户端软件可以存储在网络中的一台中央计算机上，当终端用户需要用该软件的时候，这个中央计算机会通过网络将可执行的软件传送到终端用户的计算机上运行。

因此，软件的网络移动性标志着计算模式发展历程中的重要一步，尤其是它解决了分布式处理系统中系统管理的问题，简化了将软件分布在多台CPU上的工作。它使数据可以跟相关软件（该软件知道如何操纵或显示数据）一起传送。因为代码可以跟数据一起传输，所以用户可以得到最新版本的代码。因此，有了网络移动性，分布式系统既可以像大型计算机系统那样集中管理软件，又能将计算任务分布在多个CPU上进行。

4.2 一种新的软件模式

从大型计算机模式过渡到分布式处理模式是个人计算机革命的一个产物。而个人计算机革命的到来得益于处理器性能的快速增长和价格的降低。无独有偶，软件模式向着具有网络移动性的分布式处理的方向发展，则得益于另一种硬件的发展，即网络带宽的性能提高和价格下降。网络带宽是指网络中可负载的信息总量。带宽的增长使传输新的数据成为可能；而每增加一种传输信息，网络就会呈现一种新的特性。这样，随着带宽的增长，网络上传输的简单文本可以附带上图片，网络就实现了报纸和杂志的功能。一旦带宽足以负载音频数据流，网络就能够担当起收音机、CD播放机或者电话的任务。带宽继续增长的话，传输视频数据就成为可能，网络就可以与电视机、录像机匹敌了。除此之外，网络带宽的增长还促进了另一种事务的发展，即计算机软件。因为网络是由互相连接的处理器组成的，理论上讲，只要有足够的带宽，一个处理器就可以通过网络将代码发送到另一个处理器上执行。一旦软件可以像数据一样被传输，整个网络就仿佛一台计算机一样。

一旦软件可以通过网络传输，那么不仅网络，就连软件本身，也会呈现出一种新的特征。具有网络移动性的代码很容易确保终端用户拥有必备的软件来浏览和操纵网络传输的数据，因为软件可以随数据一起传输。在旧的模式中，用户启动本地磁盘上的可执行软件来浏览网络传输过来

的数据，所以软件应用程序通常是一个与数据完全独立的实体。而新的模式中，由于软件和数据都是由网络传输的，软件和数据的区别已经不那么明显了，软件和数据被统称为“内容”。

当软件的本质发生了变化时，终端用户与软件的关系也随之变化。在有网络移动性以前，终端用户不得不考虑软件应用程序的版本问题。软件通常是通过磁带、软件或者光盘等介质来发布的。如果想使用某个应用程序，终端用户必须先得到这种安装介质，将它们插入计算机附加的驱动器，运行安装程序，将安装介质上的文件拷贝到计算机硬盘上。不仅如此，用户经常需要为同一个应用程序多次重复该安装过程，因为软件经常会有新旧版本的更替（新版本解决了旧版本中的问题，代替旧版本，但同时也会带来新的麻烦）。当新版本软件发布时，终端用户不得不决定是否要升级该软件。如果要升级，就得重复安装过程。因此，用户不得不考虑软件的版本问题，并花大量精力来更新软件。

在新的软件模式下，终端用户可以较少考虑软件的版本问题，而享受一种自动更新的“内容服务”。尽管传统软件的安装和升级对用户来说是一项需要慎重考虑的工作，但软件的网络移动性可以使软件的安装和升级自动完成。网络发布的软件不需要让用户知道断断续续的版本号，用户也不必决定是否升级，不必亲自动手去升级。网络发布的软件能够自动保持版本一致。终端用户不用再购买版本众多的软件应用程序，而只需订购一个通过网络发布的并带有相关数据的内容服务的软件，然后，这个软件和数据就可以自动更新啦。

一旦抛弃了旧有的、有多个版本的软件，而采用交互的内容自动更新的软件发布方式，终端用户就会丧失一些控制权力。在旧的软件模式下，如果软件的新版本出现了严重的bug，终端用户只要不升级就可以避免问题了。但在新的软件模式下，因为用户可能没有权力控制升级过程，所以，有可能在发现新版本的问题前就已经被安装了新版软件。

对于某些软件，尤其是那些庞大的、功能齐全的软件，终端用户更希望能保留权力，让自己决定什么时候、在哪里升级。因此，在某些情况下，软件提供商会通过网络发布内容服务软件的不同版本。至少，提供商会发布一个服务的两个版本，一个是beta版本，一个是正式发布版本。希望使用最新版本的终端用户可以订购beta版本的服务，其他用户可以订购正式版本——因为尽管正式版本不像beta版本那样有最新的特性，但是更稳定，健壮性更好。

对于某些内容服务，尤其是简单的软件，大多数终端用户不想因过多地考虑版本问题而增加软件的使用难度。终端用户不得不了解不同版本间的差别，并决定什么时候、是否要费劲儿地去升级这个软件。而没有分散成多个版本的内容服务相比之下就容易使用多了，因为它们可以自动升级。因为用户无需维护，而只要简单使用即可，所以这样的内容服务看上去就像一个盛软件的器皿。

许多自动更新的内容服务与普通的家居器皿有两个相似的重要特征：有一个主要的功能和一个简单的用户接口。比如说烤箱吧，烤箱的主要功能就是准备烘烤食物，而且有一个简单的用户接口——你想使用烤箱时，并不想去读复杂的说明书吧？你希望简单地把面包放进去，合上烤箱，看着里面亮起桔黄色的微光，片刻后，你的面包就烤好了。如果不小心烤得太焦或者不够，你就希望能有一个旋钮，以便下次烤面包时做相应的调整。这就是烤箱的功能和接口。与之相似，许多内容服务的功能也很单一，用户接口简单易用。比如想在网络上订购一部电影，你肯定不想费心去考虑是否有合适的电影订购软件版本，也不愿去安装这种软件。你只想打开

订购电影的内容服务，通过简单的用户接口来订购你的电影。然后，就可以坐下来欣赏网上传输过来的电影和享用烤面包了。

内容服务的一个绝好例子就是万维网网页。当你看一个html文件时，它可以看做某种程序的源文件；但是如果你把浏览器看做应用程序的话，那html文件就可以看做数据了。因此，源代码与数据间的区别不很清晰。同样，人们浏览万维网时，希望网页能自动地不断更新。人们不希望看到网页的多个版本，不希望费很大力气手工地升级到网页的最新版本。

今后，现在的许多媒体都将在某种程度上溶入网络，转型到内容服务上去。收音机、电视机、电话、应答机、传真机、录像带出租店、报纸、杂志、书籍，甚至计算机软件……这一切都会受到网络发展的影响。正如电视机不能完全取代收音机一样，网络传输的内容服务也不可能完全替代现有的媒体。但内容服务有可能取代现有媒体某些方面的功能，使其做出相应的调整，并创造出一些新的媒体形式。

在计算机领域，内容服务模式也不能完全取代旧的软件模式。它只能替代旧模式的某些方面（它们更适合新的软件模式），增加一些新的形式，促使旧的软件模式进行调整。

本书就是网络影响现有媒体的一个典型例子。这本书并没有完全被内容服务所代替，但书中不再给出详细的参考资料，取而代之的是相关的网页。因为参考资料会经常更新，将网络资料作为本书的一部分确实很有意义。这样，书中参考资料的部分就以内容服务的形式出现了。

综上所述，这种新的软件模式的关键就在于，软件开始呈现出一种“容器”的特性。终端用户不必再担心安装、升级以及软件版本的问题。因为代码和数据一起通过网络传输，所以软件可以自动进行发布和升级。Java通过实现代码的网络移动性，为软件的开放、发布和使用提供了一种全新的思路。

4.3 Java体系结构对网络移动性的支持

Java体系结果对网络移动性的支持是和它对平台无关性和安全性的支持密不可分的。虽然平台无关性和安全性对网络移动性而言并非是必需的，但它们对网络移动性的实际应用有很大帮助。平台无关性使得在网络上传送程序更加容易，因为不需要为每个不同的主机平台都准备一个单独的版本，因此也不需要判断每台计算机需要哪个特定的版本，一个版本就可以对付所有的计算机。Java的安全特性促进了网络移动性的推广，因为最终用户就算从不信任的来源下载class文件，也可以充满自信。因此实际上，Java体系结构通过对平台无关性和安全性的支持，更好地推广了它的class文件的网络机动性。

除了平台无关性和安全性之外，Java体系结构对网络移动性的支持主要集中在对在网络上传送程序的时间进行管理上。假若你在服务器上保存了一个程序，在需要的时候通过网络来下载它，这个过程一般都会比从本地执行该程序要慢。因此，对于在网络上传送程序来说，网络移动性的一个主要难题就是时间。Java体系结构通过把传统的单一二进制可执行文件切割成小的二进制碎片——Java class文件——来解决这个问题。class文件可以独立在网络上传播，因为Java程序是动态连接、动态扩展的，最终用户不需要等待所有的程序class文件都下载完毕，就可以开始运行程序了。第一个class文件到手，程序就开始执行。class文件本身也被设计得很紧凑，所以它们可以在网络上飞快地传送。因此，Java体系结构为网络移动性带来的直接主要好处就是

把一个单一的大二进制文件分割成小的class文件，这些class文件可以按需装载。

Java应用程序从某个类的main（）方法开始执行，其他的类在程序需要的时候才动态连接。如果某个类在一次操作中没有被用到，这个类就不会被装载。比如说，假若你在使用一个字处理程序，它有一个拼写检查器，但是在你使用的这次操作中没有使用拼写检查器，那么它就不会被装载。

除了动态连接之外，Java体系结构也允许动态扩展。动态扩展是装载class文件的另一种方式（可能是从网络上下载它们），可以延迟到Java应用程序运行时才装载。使用用户自定义的类装载器，或者Class类的forname（）方法，Java程序可以在运行时装载额外的程序，这些程序就会变成运行程序的一部分。因此，动态连接和动态扩展给了Java程序员一些设计上的灵活性，即可以决定何时装载程序的class文件——而这又决定了最终用户需要等待多少时间来从网络上装载class文件。

除了动态连接和动态扩展，Java体系结构对网络移动性的直接支持还通过class文件格式体现。为了减少在网络上传送程序的时间，class文件被设计得很紧凑。它们包含的字节码流设计得特别紧凑——之所以被称为“字节码”，是因为每条指令都只占据一个字节。除了两个例外情况，所有的操作码和它们的操作数都是按照字节对齐的，这使得字节码流更小。这两个例外是这样一些操作码，在操作码和它们的操作数之间会填上一到三个字节，以便操作数都按照字边界对齐。

class文件的紧凑性隐含着另外一个含义，那就是Java编译器不会做太多的局部优化。因为二进制兼容性规则的存在，Java编译器不能做一些全局优化，比如把一个方法调用转化为整个方法的内嵌（内嵌指把被调用方法的整个方法体都替换到发起调用的方法中去，这样在代码运行的时候，可以节省方法调用和返回的时间）。二进制兼容性要求，假若一个方法被现有的class文件依赖，那么改变这个方法的时候必须不破坏已有的调用关系。在同一个类中使用的方法可能使用内嵌，但是一般来说Java编译器不会做这种优化，部分原因是因为，这样为class文件瘦身得不偿失。优化常常是在代码大小和执行速度间进行的折中。因此，Java编译器通常会把优化工作留给Java虚拟机，后者在装载类之后，在解释执行、即时编译或者自适应编译的时候都可以优化代码。

除了动态连接、动态扩展和紧凑的class文件之外，还有一些并非体系结构必需的策略，可以帮助控制在网络上传送class文件的时间。因为HTTP协议需要单独为Java applet中用到的每一个class文件单独请求连接，那么我们会发现下载applet的很大一部分时间并不是用来实际传输class文件的时间，而是每一个class文件请求的网络协议握手的时间。一个文件需要的总时间是按照需要下载的class文件的数目倍增的。为了解决这个问题，Java 1.1包含了对JAR（Java打包归档文件）的支持，JAR文件允许在一次网络传输过程中传送多个文件，这和一次传送一个个单独class文件相比，大幅度降低了需要的总体下载时间。更好的优点是，JAR文件中的数据可以压缩，从而使下载时间更少。所以有时候通过一个大文件来传送软件。假如有些class文件是程序开始运行之前所必需的，这些文件可以很快地通过JAR文件一次性传送。

另外一个降低最终用户等待时间的策略就是不采取按需下载class文件的做法。有几种不同的技术，例如Marimba Castanet使用的订阅模式，可以在需要class文件之前就已经把它们下载下来了，这样程序就可以更快地启动。读者可以在本章的资源页找到一些有关这种方法的详细资料。

因此，除了平台无关性和安全性能够对网络移动性有利外，Java体系结构主要的着眼点就是

控制class文件在网络上传送的时间。动态连接和动态扩展允许Java程序按照小功能单元设计，在最终用户需要的时候才单独下载。class文件的紧凑性本身有助于减少Java程序在网络上传送的时间。JAR文件允许在一次网络连接中传送多个文件，还允许数据压缩。

4.4 applet：网络移动性代码的示例

Java是一种网络化的技术，在网络被逐渐证明成为下一次计算革命的时代出现。然而Java被如此迅速而广泛使用的原因，并不在于它是一种适时的技术，而是因为它有合适的市场。Java并非是20世纪90年代中期开始发展的惟一一种基于网络的语言。虽然它是一个好技术，它并非一定是最好的——但是它可能是最有市场的。Java在1995年初打开了一小块市场，结果获得了很强烈的回应，使得很多开发类似技术的公司被迫取消了他们的项目。拥有类似技术的公司，比如AT&T，他们拥有一个网络化的技术称为Inferno，不得不面对Java窃取了它们本可能获得的掌声这样一种事实。

Java从最初产生到获得巨大的市场成功，有几个很重要的事实。首先，它有一个很“酷”的名字——除了程序员以外，非程序员也能赏识它。其次，它在所有的应用中都是免费的——对潜在的客户来说这是一个魔咒。但是Java获得市场成功最重要的一点就是，Sun的工程师适时把Java技术和WWW融合了起来，那恰恰正好是Netscape试图把它们的网络浏览器从一个图形化的超文本查看器变成一个全功能的计算平台的时候。随着WWW如同不断增高的巨潮席卷整个软件产业（也是一次全球思潮），Java就站在了浪尖上。因此，Java的成功可以说是因为Java会“在网上冲浪”。它在正确的时间抓住了浪潮，并且一次次稳稳地站在浪尖上，它潜在的竞争者都无声无息地被冰冷的暗涛吞没了。Sun的工程师把Java技术和WWW融合起来的方法——也就是Java市场成功的关键所在——就是创造了一种Java程序的特殊形式，可以在Web浏览器内部运行：Java applet（Java小应用程序）。

Java applet展示了Java基于网络的所有特性：平台无关性、网络移动性和安全性。平台无关性对于WWW来说是一个主要原则，Java applet正好符合。在任何平台上，只要有支持Java（Java-capable）的浏览器，Java applet就可以运行。Java applet也展示了Java在安全上的能力，因为它们是在一个严格受限的沙箱中运行的。但是最重要的，Java applet展示了它承诺的网络移动性。如图4-1所示，Java applet可以在一个中心服务器上维护，可以通过网络传送到很多不同种类的计算机中。要升级一个applet，只需要升级服务器上的即可。用户下一次使用applet的时候，就可以得到升级过后的版本。因此，维护是本地的，运行却是分布式的。

支持Java的浏览器取代了使用Java程序来包容浏览器显示类applet的方法。要显示一个网页，Web浏览器从HTTP 服务器请求一个HTML文件。假如HTML文件中包含一个applet，浏览器会看到下面这样的HTML 标记：

```
<applet CODE="HeapOfFish.class"
CODEBASE="gcsupport/classes"
WIDTH=525
HEIGHT=360></applet>
```

这个“applet”标记给了浏览器足够的信息来显示这个applet。CODE属性标示了applet初始

class文件的名字，这里是HeapOfFish.class。CODEBASE属性指明了class文件相对于这个网页的URL路径。WIDTH和HEIGHT属性表明了applet屏幕的像素宽度和高度，也就是在网页上applet会显示出来的区域。

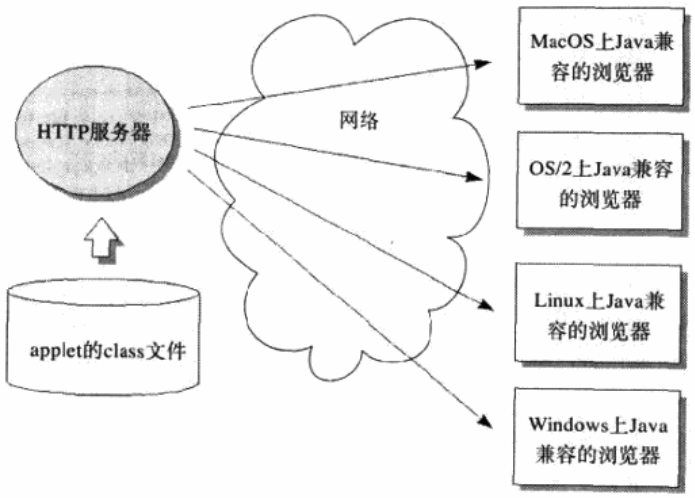


图4-1 Java applet和网络

当浏览器遇到一个包含applet标记的网页时，它把信息从标记送到一个运行中的Java程序。Java程序创建一个新的用户自定义的类装载器对象（或者重用已有的那个），来装载这个applet的初始class文件。然后它首先调用applet的初始类的init（）方法，再调用start（）方法，这样就初始化了applet。applet需要的其他class文件按照按需下载的原则在动态连接的处理过程中下载。比如说，当一个applet的初始类第一次用到某个新类的时候，必须解析新类的符号引用。在解析过程中，如果类早已被装载，Java虚拟机会要求装载同一个用户自定义的类装载器（该装载器装载该applet的初始类以装载新类）。如果用户自定义的类装载器不能本地装载类，它就会尝试从网络上下载这个class文件。装载器将试图从它获得applet的初始类相同的位置下载。一旦applet的初始化完成了，applet就如同网页的一部分一样在浏览器中显示出来。

4.5 Jini服务对象：网络移动对象的示例

虽然Java体系结构使代码的网络移动性成为可能，并由代表了计算模型的一次重要革命的Java applet表现出来，但Java结构还有另外一个承诺：对象的网络移动性。对象在网络中穿梭，携带着定义自己的类，加上表示对象状态的快照数据。如同代码的网络移动性可以简化系统管理员的工作一样，对象的网络移动性可以简化软件开发者设计和部署分布式系统的工作。通过对象序列化和远程方法调用（RMI），Java API提供了一个在本地对象模型上扩展而成的分布式对象模型，打破了Java虚拟机之间的界限。分布式对象模型使得一个虚拟机中的对象可以引用另一个虚拟机中的对象，调用那些远程对象的方法，在虚拟机之间把对象当作参数、返回值或者方法调用抛出的意外来交换。这种由Java底层基于网络的体系结构所带来的能力，可以简化设计分布式系统的任务，因为它们有效地把面向对象编程带入了网络。

通过Java的网络友好的底层体系结构，以及对象序列化和RMI技术，有一种技术利用了对象网络移动性的全部好处，那就是Sun的Jini技术。Jini是一系列协议和API的集合，可以支持对分布式系统的编写和部署，目标是把正在快速发展的、没有硬盘的嵌入设备连接到网络中来。Jini体系结构的一个特别组成部分是服务对象，它可以告诉我们对象移动性是多么有用。

Jini系统是以“查找服务”为中心的，其他的服务在“查找服务”中注册，这是通过在对象之间传递一种特殊的“服务对象”来完成的。服务对象对客户机来说就代表服务。需要访问服务的客户机从查找服务中获取一个服务对象的拷贝，通过调用服务对象的方法来和服务进行交互。服务对象负责实现服务——不管是本地实现的服务，还是和网络另一端的软件进程或者硬件对话实现的服务。

4.5.1 Jini是什么

在Jini的“思维”方式中，网络是由“服务”组成的，客户机或者其他服务可以利用这些服务。服务可能是网络上的任何形式，它们准备好实现某种功能。硬件设备、软件服务器，或者是通信信道，甚至是用户自己都可以成为服务。比如说支持Jini的磁盘驱动器，可以提供“存储”服务。支持Jini的打印机可以提供“打印”服务。

为了完成一项任务，客户机征用一些服务来帮助它。比如，客户程序可能从数码相机的图像存储服务中抓取（上传）图片，把它们下传到磁盘驱动器提供的永久存储服务中，把一整页缩小的图片发送到彩色打印机提供的打印服务。在这个例子中，客户程序建立了一个分布式系统，包括程序本身、图像存储服务、永久存储服务，还有彩色打印服务。客户机和这个分布式系统中的各个服务协同工作来完成任务：把数码相机中的照片取出来，打印出一张缩略图。

4.5.2 Jini如何工作

Jini提供了一个运行时基础结构，以允许服务提供者为客户机提供服务，也使得客户机可以找到并访问服务。运行时基础结构在网络的三个部分中存在：网络上的查找服务、服务提供者（比如是支持Jini的设备）和客户机。查找服务是基于Jini的系统的核心组织机制。新的服务在网络上出现，它们自会在查找服务中注册。当客户机试图查找某个服务来帮助它完成某个任务的时候，它们就去找查找服务。

运行时基础结构采用一种网络级协议——称为“探索”，以及两种对象级协议——分别称为“加入”和“查找”。“探索”让客户机和服务可以找到查找服务。“加入”让服务在查找服务中注册自己。“查找”让客户机询问查找服务，是否存在一种服务可以帮助自己完成工作。

探索过程 当某个服务提供者，比如提供存储服务的、支持Jini的磁盘驱动器插入到网络的时候，探索过程自动开始。一连上网，服务提供者就开始广播一个关于自己存在的通知，具体方法是向一个公开的端口发送组播包。在存在通知中，它会通报自己的IP地址和端口号，以便查找服务能够和它取得联系。

查找服务在公开的端口上监听存在通知包。一旦查找服务收到了通知，它会检查通知的详情。从包中包含的信息中，查找服务就可以做出判断，是否应该和通知的发出者取得联系。如果确认，它会使用包中包含的IP地址和端口号建立和服务提供者之间的直接TCP连接。查找服务可以使用RMI来发送一个叫做服务注册器的对象，通过网络传送到通知包的源头。服务注册器是用来和查找服务建立更进一步的沟通而设计的。发出存在通知的对象通过这个服务注册对象

就可以对查找服务发出join（加入）和lookup（查找）操作。对这个磁盘驱动器来说，查找服务会和它建立TCP连接，发送过来一个服务注册器，磁盘驱动器就可以使用join操作来注册自己的存储服务了。

加入过程　服务注册器对象是探索的结果，服务提供者一旦获得了它之后，就可以进行加入操作了——在查找服务中注册自己。服务调用服务注册器对象的register（）方法来完成注册，调用的时候传递一个服务条目（service item）作为参数，在这个服务条目中包含了一些描述这个服务的对象。register（）方法把服务条目的一份拷贝传递回查找服务，查找服务负责保存所有的服务条目。这一步完成后，服务提供者的加入过程就算完成了——在查找服务中已经注册好了。

服务条目包含几个对象，其中包括一种服务对象，客户机可以用它来和服务交互。服务条目也可以包含任意数目的属性，可以是任何对象类型的。可能包含图标对象，或者为服务提供图形界面的对象，或者是给出更加详细服务信息的对象。

服务对象通常会实现一个或多个接口，客户机可以藉此来和服务交互。比如说，查找对象本身也是一个Jini服务，它的服务对象就是服务注册器。服务提供者在加入服务的时候调用的register（）方法就是在ServiceRegistrar接口中声明的，所有的服务注册器对象都会实现它。客户机和服务提供者和查找服务通话时通过调用服务注册器的方法，这些方法都是在ServiceRegistrar接口中声明的。类似的，磁盘驱动器可以提供一个服务对象，它实现了一些常见的存储服务接口。客户机可以通过存储服务接口来和磁盘驱动器交互。

查找过程　一旦服务通过加入过程在查找服务中注册之后，客户机通过查询查找服务就可以使用它了。要想建立一个能够协同运作完成某项任务的分布式系统，客户机必须能够查找到每个服务，来寻求它们的帮助。客户机通过查找过程来完成找到服务的任务。

客户机通过调用服务注册器上的lookup（）方法进行查找。（客户机也是通过和前面描述的探索过程一样的过程，从查找服务获得一个服务注册器的。）客户机给lookup（）方法传递一个叫做服务模板的参数，这是一个表示搜索条件的对象。服务模板可以包含一个指向Class对象数组的引用。这些Class对象表明了客户机需要的服务对象的Java类型。服务模板也可以包含一个服务编号ID，它可以惟一地表示一个服务；还有属性，必须和服务提供者在服务条目中列出的属性完全相同。服务模板的任何字段都可以包含通配符。比如说服务编号ID字段如果包含通配符，就可以匹配任何服务编号ID。lookup（）方法把这个服务模板送到查找服务，查找服务进行查询，返回零个或多个满足条件的服务对象。客户机可以从lookup（）方法返回值中取得这些匹配的服务对象的引用。

一般来说，客户机通过Java类型（通常是接口）来查找服务。比如说，假若客户机需要使用打印机，它会编写一个服务模板，其中一个Class对象表示公开的打印服务接口。所有的打印服务都应该实现这个接口。查找过程就会返回所有实现这个接口的服务对象（或者对象组）。通过更改服务模板的属性，可以减小这样基于类型搜索返回结果的数目。客户机可以通过公开的打印服务接口中声明的方法来使用打印服务。

4.5.3 服务对象的优点

在Jini系统中，网络移动对象可以移动到任何地方。当客户机或者服务进行探索的时候，它

会从查找服务收到一个服务注册器。当通过服务注册器进行加入的时候，它会传送一个服务条目对象给查找服务，而服务条目本身也是一个包含了很多对象的容器，其中包括属性和服务对象。当客户机进行查找的时候，它会发送一个服务模板对象，包含一系列对象，表示查询需要的搜索条件。如果查询成功了，客户机会收到匹配查询的服务对象，或者是服务的整个服务条目。

这些在网络上客户机、服务和查找服务之间移动的对象到底能对分布式程序有什么好处呢？简单说来，Jini使用的网络移动的对象（特别是网络移动的服务对象）提高了分布式系统编成的抽象级别，有效地把网络编程转变为面向对象编程。

Jini体系通过把面向对象编程引入网络，带来了面向对象的一个基本优点：接口和实现分离。比如，服务对象允许客户机用好几种方法来获取服务，客户从查询服务下载、在本地运行的服务对象可以代表整个服务。或者说，服务对象可以作为远程服务器的一个代理。当客户机调用服务对象的方法的时候，它把请求通过网络传送回服务器，那才是真正工作的地方。本地服务对象和远程服务器也可能共同分担工作。

Jini体系的一个重要推论就是，在服务对象代理和远程服务器之间使用的网络协议，客户机是无需关心的。在图4-2中可以看出，网络协议是实现服务的一部分，协议完全是服务开发者的私人事务。客户机可以通过这种私有协议和服务器交互，因为服务把它自己的服务对象送到客户机的地址空间中——服务对象在服务和客户机的网络上来回传送。注入客户机的服务对象可以用任何协议和后端的服务通信，RMI、CORBA、DCOM，或者是自己在socket和流上面建立的协议，甚至是其他的任何方法。客户端完全不需要关心网络协议，因为它只需和服务对象实现的公开接口打交道。服务对象负责任何需要进行的网络交流。

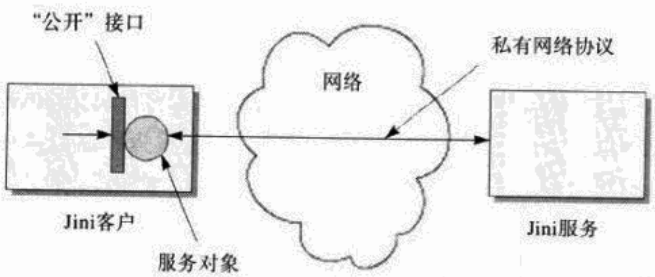


图4-2 客户机和服务通过公开的接口交互

同一服务接口的不同实现可以使用完全不同的方法和完全不同的网络协议。服务可能使用特制的硬件来满足客户机的请求，或者使用软件实现所有功能。服务不同的实现方法可以针对不同的环境优化。另外，服务采用的方法可能随时间而变化。客户机可以确信服务对象了解服务是如何实现的，因为客户机就是（通过查找服务）从服务那儿取得这个服务对象的。对于客户机来说，不管服务是如何实现的，任何服务看起来都是公开的接口。

因此，Jini试图提升分布式系统编程的抽象级别，从网络协议级别提升到对象接口级别。在越来越多嵌入式设备都要连接到网络上的情况下，分布式系统的各个部分可能来自不同的供应商。Jini让供应商不需要依附于某一种低层网络协议（这种协议让它们的设备互连）。相反，供

应商只需要在互连设备的高层Java接口层上达成一致就可以了。讨论的级别从网络协议层提升到对象接口层，可以让供应商更加集中于高层的概念而非拘泥在基层细节中。Jini讨论的高层问题可以使类似产品的供应商达成一个一致协议，来描述它们的服务如何和客户机交互。

除此之外，Jini体系允许软件开发者在开发分布式系统时享受接口和实现分离的便利。一个好处是良好设计的对象接口可以使软件开发者在大规模的分布式系统项目中更加有效地协同开发。对象接口为面向对象程序的各个部分之间签订了合同，同样，对象接口也可以用来明确大项目中团队成员之间的互相合作关系，他们每人负责一块程序。接口和实现分离的另外一个好处就是，程序员可以用它来减少变化带来的冲击，因为耦合度降低了。设计良好的对象之间结合的惟一途径就是它们的接口，实现者在对象内部做的改变不会影响到其他对象中的代码。

通过为分布式系统编程提升抽象层次和提供清晰的分离接口和实现，Jini带来了面向对象的好处。这都是因为Java对网络移动对象的支持。对象在网络上移动，是通过Java的底层结构、对象序列化、RMI来实现的，并通过Jini服务对象展示出来，这些技术给网络带来了巨大的好处。

4.6 网络移动性：Java设计的中心

网络移动的Java软件除了前面说的两种Java applet和Jini 服务对象形式之外，还可以有很多形式。虽然Java的网络移动性不仅仅局限于applet和服务对象，但是支持任何其他形式的网络移动性框架，多多少少和支持applet以及服务对象的框架有些相似。比如说如同applet和服务对象，其他形式的Java网络移动性都会在一个主机Java应用程序的上下文中运行，网络移动的Java代码需要的class文件会用用户自定义的类装载器装载。用户自定义的类装载器可以用自定义的方法从网络上下载class文件，这通常是启动类装载器所做不到的。并且因为网络移动的class文件并非总是值得信任，需要用用户自定义的类装载器提供的单独的命名空间来防止恶意的或者是有bug的代码，干涉从其他来源装载的代码。最后，因为网络移动的class文件不总是能被信任，通常会有一个安全管理器或者一个控制器来建立网络移动代码的安全政策。

所以，理解Java体系结构的关键，就在于理解代码和对象的网络移动能力是Java设计的中心。虽然Java可以提供很多有价值的优点（比如提高程序员生产力，提高程序的健壮性），甚至有时候你根本不会和远程的网络发生关系，但Java体系的主要焦点还是网络。Java虚拟机、Java class文件、Java API和Java程序设计语言一起使编写网络移动的软件成为可能且可行的。通过对代码和对象在网络上移动的支持，Java帮助软件开发者在不断发展的网络时代里，面对一切挑战和机遇。

4.7 资源页

要了解更多有关Java网络移动性的例子，参见资源页http://www.artima.com/insidejvm/resources。

第5章 Java虚拟机

本书前4章概述了整个Java技术的体系结构，同时也指出了Java虚拟机在Java技术体系中相对于其他组成部分（比如语言和API）而言所扮演的角色，本书剩余的部分将主要集中于对Java虚拟机的讨论。这一章首先对Java虚拟机的内部机制作一个概览。

Java虚拟机之所以被称之为是“虚拟”的，就是因为它仅仅是由一个规范来定义的抽象计算机。因此，要运行某个Java程序，首先需要一个符合该规范的具体实现。本章主要描述这个规范本身，但是为了更细致地描述某些具体特性，我们也将讨论它们可能以哪些方式来实现。

5.1 Java虚拟机是什么

要理解Java虚拟机，你首先必须意识到，当你说“Java虚拟机”时，可能指的是如下三种不同的东西：

- 抽象规范。
- 一个具体的实现。
- 一个运行中的虚拟机实例。

Java虚拟机抽象规范仅仅是个概念，在Tim Lindholm和Frank Yellin编著的《The Java Virtual Machine Specification》一书中详细地描述了它。而该规范的具体实现，可能来自多个提供商，并存在于多个平台上。它或者完全用软件实现，或者以硬件和软件相结合的方式来实现。当运行一个Java程序的同时，也就在运行了一个Java虚拟机实例。

每个Java程序都运行于某个具体的Java虚拟机实现的实例上。在本书中，术语“Java虚拟机”可能表示上述三种情形之一，当无法联系上下文来确定其准确意思的时候，我们会在文中指明究竟是“抽象规范”，“一个具体的实现”，还是“一个运行中的虚拟机实例”。

5.2 Java虚拟机的生命周期

一个运行时的Java虚拟机实例的天职就是：负责运行一个Java程序。当启动一个Java程序时，一个虚拟机实例也就诞生了。当该程序关闭退出，这个虚拟机实例也就随之消亡。如果在同一台计算机上同时运行三个Java程序，将得到三个Java虚拟机实例。每个Java程序都运行于它自己的Java虚拟机实例中。

Java虚拟机实例通过调用某个初始类的main（）方法来运行一个Java程序。而这个main（）方法必须是公有的（public）、静态的（static），返回值为void，并且接受一个字符串数组作为参数。任何拥有这样一个main（）方法的类都可以作为Java程序运行的起点。

比如，考虑这样一个Java程序，它打印出传给它的命令行参数：

```
// On CD-ROM in file jvm/ex1/Echo.java
class Echo {
```

```
    public static void main(String[] args) {
        int len = args.length;
        for (int i = 0; i < len; ++i) {
            System.out.print(args[i] + " ");
        }
        System.out.println();
    }
}
```

必须（以某种与实现相关的方式）告诉Java虚拟机要运行的Java程序中初始类的名字，整个程序将从它的main（）方法开始运行。现实中一个Java虚拟机实现的例子如Sun Java 2 SDK的java程序。比如，如果想要在Windows 98上使用Sun的java来运行Echo程序，得键入这样一个命令：

```
java Echo Greetings, Planet.
```

该命令中的第一个单词“java”，告诉操作系统应该运行来自Sun Java 2 SDK的Java虚拟机。第二个词“Echo”则指出初始类的名字。Echo这个初始类中必须有个公有的、静态的方法main（），它获得一个字符串数组参数并且返回void。上述命令行中剩下的单词序列“Greetings，Planet.”，作为该程序的命令行参数以字符串数组的形式传递给main（）。因此，对于上面这个例子，传递给Echo类中main（）方法的字符串数组参数的内容就是：

arg[0]为“Greetings，”

arg[1]则为“Planet.”

Java程序初始类中的main（）方法，将作为该程序初始线程的起点，任何其他的线程都是由这个初始线程启动的。

在Java虚拟机内部有两种线程：守护线程与非守护线程。守护线程通常是由虚拟机自己使用的，比如执行垃圾收集任务的线程。但是，Java程序也可以把它创建的任何线程标记为守护线程。而Java程序中的初始线程——就是开始于main（）的那个，是非守护线程。

只要还有任何非守护线程在运行，那么这个Java程序也在继续运行（虚拟机仍然存活）。当该程序中所有的非守护线程都终止时，虚拟机实例将自动退出。假若安全管理器允许，程序本身也能够通过调用Runtime类或者System类的exit（）方法来退出。

在前面的Echo程序中，它的main（）方法并未调用其他的线程。因此，当它打印完命令行参数后，main（）方法返回。这就终止了该程序中惟一的非守护线程，最终导致虚拟机实例退出。

5.3 Java虚拟机的体系结构

在Java虚拟机规范中，一个虚拟机实例的行为是分别按照子系统、内存区、数据类型以及指令这几个术语来描述的。这些组成部分一起展示了抽象的虚拟机的内部抽象体系结构。但是规范中对它们的定义并非是要强制规定Java虚拟机实现内部的体系结构，更多的是为了严格地定义这些实现的外部特征。规范本身通过定义这些抽象的组成部分以及它们之间的交互，来定义任何Java虚拟机实现都必须遵守的行为。

图5-1是Java虚拟机的结构框图，包括在规范中描述的主要子系统和内存区。前一章我们曾

提到，每个Java虚拟机都有一个类装载器子系统，它根据给定的全限定名来装入类型（类或接口）。同样，每个Java虚拟机都有一个执行引擎，它负责执行那些包含在被装载类的方法中的指令。

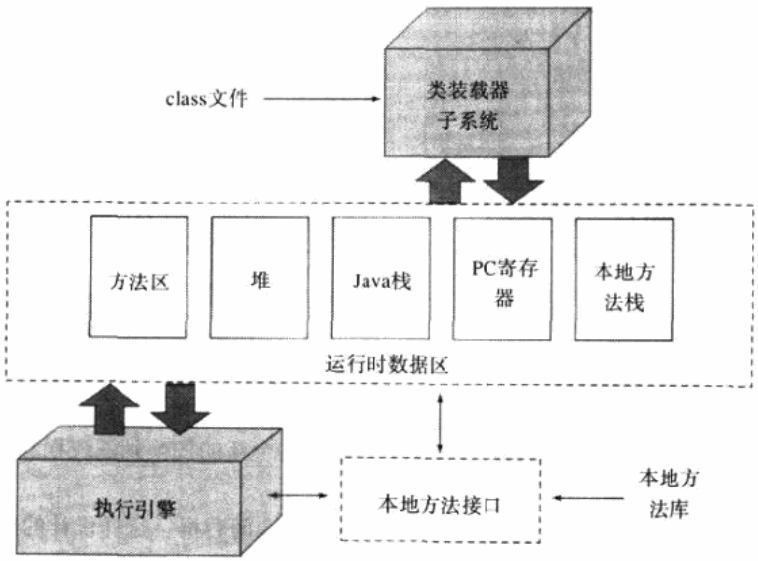


图5-1 Java虚拟机的内部体系结构

当Java虚拟机运行一个程序时，它需要内存来存储许多东西，例如，字节码，从已装载的class文件中得到的其他信息，程序创建的对象，传递给方法的参数，返回值，局部变量，以及运算的中间结果等等。Java虚拟机把这些东西都组织到几个“运行时数据区”中，以便于管理。

尽管这些“运行时数据区”都会以某种形式存在于每一个Java虚拟机实现中，但是规范对它们的描述却是相当抽象的。这些运行时数据区结构上的细节，大多数都由具体实现的设计者决定。

不同的虚拟机实现可能具有很不同的内存限制，有的实现可能有大量的内存可用，有的可能只有很少。有的实现可以利用虚拟内存，有的则不能。规范本身对“运行时数据区”只有抽象的描述，这就使得Java虚拟机可以很容易地在各种计算机和设备上实现。

某些运行时数据区是由程序中所有线程共享的，还有一些则只能由一个线程拥有。每个Java虚拟机实例都有一个方法区以及一个堆，它们是由该虚拟机实例中所有线程共享的。当虚拟机装载一个class文件时，它会从这个class文件包含的二进制数据中解析类型信息。然后，它把这些类型信息放到方法区中。当程序运行时，虚拟机会把所有该程序在运行时创建的对象都放到堆中。请看图5-2中对这些内存区域的描绘。

当每一个新线程被创建时，它都将得到它自己的PC寄存器（程序计数器）以及一个Java栈。如果线程正在执行的是一个Java方法（非本地方法），那么PC寄存器的值将总是指示下一条将被执行的指令，而它的Java栈则总是存储该线程中Java方法调用的状态——包括它的局部变量，被调用时传进来的参数，它的返回值，以及运算的中间结果等等。而本地方法调用的状态，则是以某种依赖于具体实现的方式存储在本地方法栈中，也可能是在寄存器或者其他某些与特定实现相关的内存区中。

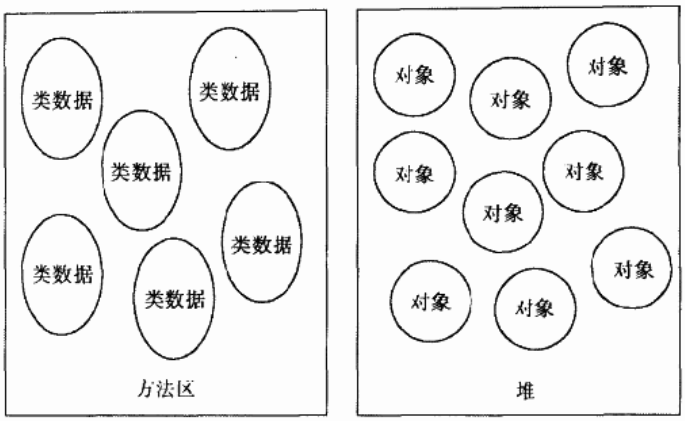


图5-2 由所有线程共享的运行时数据区

Java栈是由许多栈帧（stack frame）或者说帧（frame）组成的，一个栈帧包含一个Java方法调用的状态。当线程调用一个Java方法时，虚拟机压入一个新的栈帧到该线程的Java栈中；当该方法返回时，这个栈帧被从Java栈中弹出并抛弃。

Java虚拟机没有寄存器，其指令集使用Java栈来存储中间数据。这样设计的原因是为了保持Java虚拟机的指令集尽量紧凑、同时也便于Java虚拟机在那些只有很少通用寄存器的平台上实现。另外，Java虚拟机的这种基于栈的体系结构，也有助于运行时某些虚拟机实现的动态编译器和即时编译器的代码优化。

请看图5-3，它描绘了Java虚拟机为每一个线程创建的内存区，这些内存区域是私有的，任何线程都不能访问另一个线程的PC寄存器或者Java栈。

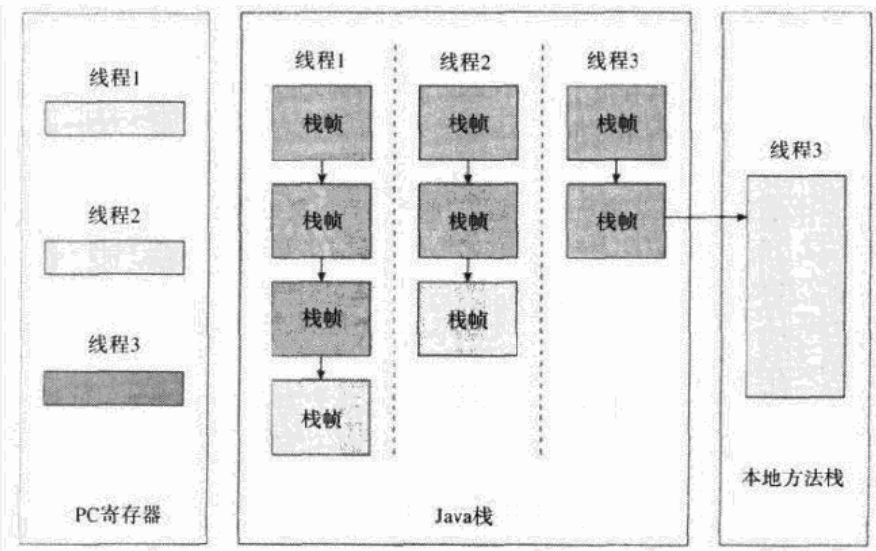


图5-3 线程专有的运行时数据区

图5-3展示了一个虚拟机实例的快照，它有三个线程正在执行。线程1和线程2都正在执行Java方法，而线程3则正在执行一个本地方法。

在图5-3中，和本书其他地方一样，Java栈都是向下生长的，而栈顶都显示在图的底部。当前正在执行的方法的栈帧则以浅色表示。对于一个正在运行Java方法的线程而言，它的PC寄存器总是指向下一条将被执行的指令。在图5-3中，像这样的PC寄存器（比如线程1和线程2的）都是以浅色显示的。由于线程3当前正在执行一个本地方法，因此，它的PC寄存器——以深色显示的那个，其值是不确定的。

5.3.1 数据类型

Java虚拟机是通过某些数据类型来执行计算的，数据类型及其运算都是由Java虚拟机规范严格定义的。数据类型可以分为两种：基本类型和引用类型。基本类型的变量持有原始值，而引用类型的变量持有引用值。术语“引用值”指的是对某个对象的引用，而不是该对象本身。与此相对，原始值则是真正的原始数据。请看图5-4中对Java虚拟机中数据类型的描绘。

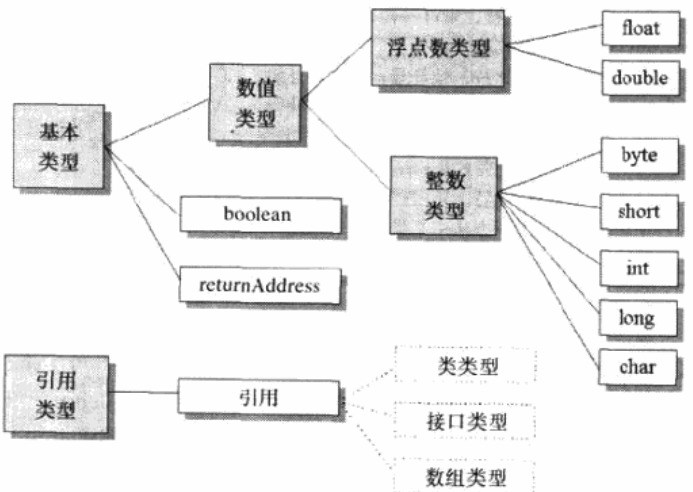


图5-4 Java虚拟机中的数据类型

Java语言中的所有基本类型同样也都是Java虚拟机中的基本类型。但是boolean有点特别，虽然Java虚拟机也把boolean看做基本类型，但是指令集对boolean只有很有限的支持。当编译器把Java源码编译为字节码时，它会用int或byte来表示boolean。在Java虚拟机中，false是由整数零来表示的，所有非零整数都表示true。涉及boolean值的操作则会使用int。另外，boolean数组是当做byte数组来访问的，但是在“堆”区，它也可以被表示为位域。

除了boolean以外，Java语言中的基本类型构成了Java虚拟机中的数值类型。虚拟机中的数值类型分为两种：整数类型（包括byte、short、int、long、char）和浮点数类型（包括float和double）。和Java语言一样，Java虚拟机的基本类型的值域在任何地方都是一致的，比如，不管底层的主机平台是什么，一个long在任何虚拟机中总是一个64位二进制补码表示的有符号整数。

Java虚拟机中还有一个只在内部使用的基本类型：returnAddress，Java程序员不能使用这个类型。这个基本类型被用来实现Java程序中的finally子句。关于returnAddress的用法我们将在第

18章详细讨论。

Java虚拟机的引用类型被统称为“引用”（reference），有三种引用类型：类类型，接口类型，以及数组类型，它们的值都是对动态创建对象的引用。类类型的值是对类实例的引用；数组类型的值是对数组对象的引用，在Java虚拟机中，数组是个真正的对象；而接口类型的值，则是对实现了该接口的某个类实例的引用。还有一种特殊的引用值是null，它表示该引用变量没有引用任何对象。

Java虚拟机规范定义了每一种数据类型的取值范围，但是却没有定义它们的位宽。存储这些类型的值所需的占位宽度，是由具体的虚拟机实现的设计者决定的。关于Java虚拟机数据类型的取值范围，请看表5-1。更多关于浮点类型的取值范围的信息，请看第14章。

表5-1 Java虚拟机数据类型的取值范围

类 型	范 围
byte	8比特，带符号，二进制补码（-2^7到2^7-1，包括两端的值在内）
short	16比特，带符号，二进制补码（-2^{15}到$2^{15}-1$，包括两端的值在内）
int	32比特，带符号，二进制补码（-2^{31}到$2^{31}-1$，包括两端的值在内）
long	64比特，带符号，二进制补码（-2^{63}到$2^{63}-1$，包括两端的值在内）
char	16比特，不带符号，Unicode字符（0到$2^{16}-1$，包括两端的值在内）
float	32比特，IEEE 754标准单精度浮点数
double	64比特，IEEE 754标准双精度浮点数
returnAddress	同一方法中某操作码的地址
reference	堆中对某对象的引用，或者是null

5.3.2 字长的考量

Java虚拟机中，最基本的数据单元就是字（word），它的大小是由每个虚拟机实现的设计者来决定的。字长必须足够大，至少是一个字单元就足以持有byte、short、int、char、float、returnAddress或者reference类型的值，而两个字单元就足以持有long或者double类型的值。因此，虚拟机实现的设计者至少得选择32位作为字长，或者选择更为高效的字长大小。通常根据底层主机平台的指针长度来选择字长。

在Java虚拟机规范中，关于运行时数据区的大部分内容，都是基于“字”这个抽象概念的。比如，关于栈帧的两个部分——局部变量和操作数栈——都是按照“字”来定义的。这些内存区域能够容纳任何虚拟机数据类型的值，当把这些值放到局部变量或者操作数栈中时，它将占用一个或两个字单元。

在运行时，Java程序无法侦测到底层虚拟机的字长大小；同样，虚拟机的字长大小也不会影响程序的行为——它仅仅是虚拟机实现的内部属性。

5.3.3 类装载器子系统

在Java虚拟机中，负责查找并装载类型的那部分被称为类装载器子系统。在第1章中，我们曾经简单地介绍过它，随后在第3章中我们也讨论过类装载器子系统在Java的安全模型中所扮演的角色。在这一章我们将会更为详细地讨论它，并展示它和虚拟机内部体系的其他组件是如何互相联系的。

Java虚拟机有两种类装载器：启动类装载器和用户自定义类装载器。前者是Java虚拟机实现的一部分，后者则是Java程序的一部分。由不同的类装载器装载的类将被放在虚拟机内部的不同命名空间中。

类装载器子系统涉及Java虚拟机的其他几个组成部分，以及几个来自java.lang库的类。比如，用户自定义的类装载器是普通的Java对象，它的类必须派生自java.lang.ClassLoader类。ClassLoader中定义的方法为程序提供了访问类装载器机制的接口。此外，对于每一个被装载的类型，Java虚拟机都会为它创建一个java.lang.Class类的实例来代表该类型。和所有其他对象一样，用户自定义的类装载器以及Class类的实例都放在内存中的堆区，而装载的类型信息则都位于方法区。

装载、连接以及初始化　类装载器子系统除了要定位和导入二进制class文件外，还必须负责验证被导入类的正确性，为类变量分配并初始化内存，以及帮助解析符号引用。这些动作必须严格按以下顺序进行：

1) 装载——查找并装载类型的二进制数据。

2) 连接——执行验证，准备，以及解析（可选）。

验证　确保被导入类型的正确性。

准备　为类变量分配内存，并将其初始化为默认值。

解析　把类型中的符号引用转换为直接引用。

3) 初始化——把类变量初始化为正确初始值。

这些处理过程的细节我们将在第7章详细讨论。

启动类装载器　只要是符合Java class文件格式的二进制文件，Java虚拟机实现都必须能够从中辨别并装载其中的类和接口。某些虚拟机实现也可以识别其他的非规范的二进制格式文件，但它必须能够辨别class文件。

每个Java虚拟机实现都必须有一个启动类装载器，它知道怎么装载受信任的类，比如Java API的class文件。Java虚拟机规范并未规定启动类装载器如何去寻找class文件，这又是一件保留给具体的实现设计者去决定的事情。

只要给定某个类型的全限定名，启动类装载器就必须能够以某种方式得到定义该类型的数据。比如，在Windows 98平台上，Sun的JDK 1.1是这样工作的：首先它逐个搜索用户在CLASSPATH环境变量中定义的目录，直到找到一个名为“该类型名 + .class”的文件为止。除非该类型属于某个未命名的包，否则启动类装载器会在CLASSPATH包含的目录下的子目录中寻找这样一个文件，这些子目录的路径名是根据类型的包名称构建的。比如，如果启动类装载器正在搜索这样一个类：java.lang.Object，那么它将在每一个CLASSPATH包含的目录下，查找java\lang这样一个子目录，以及其中的Object.class文件。

在Sun的JDK 1.2中，与1.1版本不同，启动类装载器将只在系统类（Java API的类文件）的安装路径中查找要装入的类；而搜索CLASSPATH目录的任务，现在交给了系统类装载器——它是一个自定义的类装载器，当虚拟机启动时就被自动创建。更多关于Sun的Java 2 SDK中类装载机制的信息请查阅第8章。

用户自定义类装载器　尽管“用户自定义类装载器”本身是Java程序的一部分，但类Class-

Loader中的四个方法是通往Java虚拟机的通道：

```
// Four of the methods declared in class java.lang.ClassLoader:
protected final Class defineClass(String name, byte data[],
    int offset, int length);
protected final Class defineClass(String name, byte data[],
    int offset, int length, ProtectionDomain protectionDomain);
protected final Class findSystemClass(String name);
protected final void resolveClass(Class c);
```

任何Java虚拟机实现都必须把这些方法连到内部的类装载器子系统中。

两个被重载的defineClass（）方法都要接受一个名为data[]的字节数组作为输入参数，并且在data[offset]到data[offset +length]之间的二进制数据必须符合Java class文件格式——它表示一个新的可用类型。而name参数是个字符串，它指出该类型的全限定名。当使用第一个defineClass（）时，该类型将被赋以默认的保护域。使用第二个defineClass（）时，该类型的保护域将由它的protectionDomain参数指定。每个Java虚拟机实现都必须保证ClassLoader类的defineClass（）方法能够把新类型导入到方法区中。

findSystemClass（）方法接受一个字符串作为参数，它指出将被装入类型的全限定名。在版本1.0和版本1.1中，这个方法会通过启动类装载器来装载指定类型。如果启动类装载器装载完成，它会返回对Class对象（该对象描述了该类型）的引用。如果没有找到相应的class文件，它会抛出ClassNotFoundException异常。在版本1.2中，findSystemClass（）方法使用系统类装载器来装载指定类型。任何Java虚拟机实现都必须保证findSystemClass（）方法能够以这种方式调用启动类装载器（如果运行版本1.0或版本1.1），或者系统类装载器（如果运行版本1.2或以上）。

resolveClass（）方法接受一个Class实例的引用作为参数，它将对该Class实例表示的类型执行连接动作。而前面提到的defineClass（）方法则只负责装载。当defineClass（）方法返回一个Class实例时，也就表示指定的class文件已经被找到并装载到方法区了，但是却不一定被连接和初始化了。Java虚拟机实现必须保证ClassLoader类的resolveClass（）方法能够让类装载器子系统执行连接动作。

关于Java虚拟机怎样执行装载、连接以及初始化动作，请查阅第8章。

命名空间　在第3章讲过，每个类装载器都有自己的命名空间，其中维护着由它装载的类型。所以一个Java程序可以多次装载具有同一个全限定名的多个类型。这样一个类型的全限定名就不足以确定在一个Java虚拟机中的惟一性。因此，当多个类装载器都装载了同名的类型时，为了惟一地标识该类型，还要在类型名称前加上装载该类型（指出了它所位于的命名空间）的类装载器的标识。

Java虚拟机中的命名空间，其实是解析过程的结果。对于每一个被装载的类型，Java虚拟机都会记录装载它的类装载器。当虚拟机解析一个类到另一个类的符号引用时，它需要被引用类的类装载器。关于这个过程，更详细的描述参考第8章。

5.3.4 方法区

在Java虚拟机中，关于被装载类型的信息存储在一个逻辑上被称为方法区的内存中。当虚拟机装载某个类型时，它使用类装载器定位相应的class文件，然后读入这个class文件——一个线

性二进制数据流——然后将它传输到虚拟机中。紧接着虚拟机提取其中的类型信息，并将这些信息存储到方法区。该类型中的类（静态）变量同样也是存储在方法区中。

Java虚拟机在内部如何存储类型信息，这是由具体实现的设计者来决定的。比如，在class文件中，多字节值总是以高位在前（即代表较大数的字节在前）的顺序存储。但是当这些数据被引入到方法区后，虚拟机可以以任何方式存储它。假设某个实现是运行在低位优先的处理器上，那么它很可能会把多字节值以低位优先的顺序存储到方法区。

当虚拟机运行Java程序时，它会查找使用存储在方法区中的类型信息。设计者应当为类型信息的内部表示设计适当的数据结构，以尽可能在保持虚拟机小巧紧凑的同时加快程序的运行效率。如果正在设计一个需要在少量内存的限制中操作的实现，设计者可能会决定以牺牲某些运行速度来换取紧凑性。另外一方面，如果设计一个将在虚拟内存系统中运行的实现，设计者可能会决定在方法区中保存一些冗余信息，以此来加快执行速度。（如果底层主机没有提供虚拟内存，但是提供了一个硬盘，设计者可能会在实现中创建一个虚拟内存系统。）Java虚拟机的设计者可以根据目标平台的资源限制和需求，在空间和时间上做出权衡，选择实现什么样的数据结构和数据组织。

由于所有线程都共享方法区，因此它们对方法区数据的访问必须被设计为是线程安全的。比如，假设同时有两个线程都企图访问一个名为Lava的类，而这个类还没有被装入虚拟机，那么，这时只应该有一个线程去装载它，而另一个线程则只能等待。

方法区的大小不必是固定的，虚拟机可以根据应用的需要动态调整。同样，方法区也不必是连续的，方法区可以在一个堆（甚至是虚拟机自己的堆）中自由分配。另外，虚拟机也可以允许用户或者程序员指定方法区的初始大小以及最小和最大尺寸等。

方法区也可以被垃圾收集，因为虚拟机允许通过用户定义的类装载器来动态扩展Java程序，因此一些类也会成为程序“不再引用”的类。当某个类变为不再被引用的类时，Java虚拟机可以卸载这个类（垃圾收集），从而使方法区占据的内存保持最小。类的卸装以及一个类变为“不再被引用”的必需条件，都将在第7章中描述。

类型信息　对每个装载的类型，虚拟机都会在方法区中存储以下类型信息：

- 这个类型的全限定名。
- 这个类型的直接超类的全限定名（除非这个类型是java.lang.Object，它没有超类）。
- 这个类型是类类型还是接口类型。
- 这个类型的访问修饰符（public、abstract或final的某个子集）。
- 任何直接超接口的全限定名的有序列表。

在Java class文件和虚拟机中，类型名总是以全限定名出现。在Java源代码中，全限定名由类所属包的名称加一个“.”，再加上类名组成。例如，类Object的所属包为java.lang，那它的全限定名应该是java.lang.Object，但在class文件里，所有的“.”都被斜杠“/”代替，这样就成为java/lang/Object。至于全限定名在方法区中的表示，则因不同的设计者有不同的选择而不同，可以用任何形式和数据结构来代表。

除了上面列出的基本类型信息外，虚拟机还得为每个被装载的类型存储以下信息：

- 该类型的常量池。

- 字段信息。
- 方法信息。
- 除了常量以外的所有类（静态）变量。
- 一个到类ClassLoader的引用。
- 一个到Class类的引用。

在下面的小节中会描述这些数据。

常量池　虚拟机必须为每个被装载的类型维护一个常量池。常量池就是该类型所用常量的一个有序集合，包括直接常量（string、integer和floating point常量）和对其他类型、字段和方法的符号引用。池中的数据项就像数组一样是通过索引访问的。因为常量池存储了相应类型所用到的所有类型、字段和方法的符号引用，所以它在Java程序的动态连接中起着核心的作用。常量池在本章后面和第6章中会详细讨论。

字段信息　对于类型中声明的每一个字段，方法区中必须保存下面的信息。除此之外，这些字段在类或者接口中的声明顺序也必须保存。下面是字段信息的清单：

- 字段名。
- 字段的类型。
- 字段的修饰符（public、private、protected、static、final、volatile、transient的某个子集）。

方法信息　对于类型中声明的每一个方法，方法区中必须保存下面的信息。和字段一样，这些方法在类或者接口中的声明顺序也必须保存。下面是方法信息的清单：

- 方法名。
- 方法的返回类型（或 void）。
- 方法参数的数量和类型（按声明顺序）。
- 方法的修饰符（public、private、protected、static、final、synchronized、native、abstract的某个子集）。

除上面的清单中列出的条目之外，如果某个方法不是抽象的和本地的，它还必须保存下列信息：

- 方法的字节码（bytecodes）。
- 操作数栈和该方法的栈帧中的局部变量区的大小（这些在本章的后面会详细描述）。
- 异常表（参见第17章）。

类（静态）变量　类变量是由所有类实例共享的，但是即使没有任何类实例，它也可以被访问。这些变量只与类有关——而非类的实例，因此它们总是作为类型信息的一部分而存储在方法区。除了在类中声明的编译时常量外，虚拟机在使用某个类之前，必须在方法区中为这些类变量分配空间。

而编译时常量（就是那些用final声明以及用编译时已知的值初始化的类变量）则和一般的类变量的处理方式不同，每个使用编译时常量的类型都会复制它的所有常量到自己的常量池中，或嵌入到它的字节码流中。作为常量池或字节码流的一部分，编译时常量保存在方法区中——就和一般的类变量一样。但是当一般的类变量作为声明它们的类型的一部分数据而保存的时候，编译时常量作为使用它们的类型的一部分而保存。这种特殊处理方式在第6章中更详细地讨论。

指向ClassLoader类的引用　每个类型被装载的时候，虚拟机必须跟踪它是由启动类装载器还是由用户自定义类装载器装载的。如果是用户自定义类装载器装载的，那么虚拟机必须在类型信息中存储对该装载器的引用。这是作为方法表中的类型数据的一部分保存的。

虚拟机会在动态连接期间使用这个信息。当某个类型引用另一个类型的时候，虚拟机会请求装载发起引用类型的类装载器来装载被引用的类型。这个动态连接的过程，对于虚拟机分离命名空间的方式也是至关重要的。为了能够正确地执行动态连接以及维护多个命名空间，虚拟机需要在方法表中得知每个类都是由哪个类装载器装载的。关于动态连接和命名空间的细节请参见第8章。

指向Class类的引用　对于每一个被装载的类型（不管是类还是接口），虚拟机都会相应地为它创建一个java.lang.Class类的实例，而且虚拟机还必须以某种方式把这个实例和存储在方法区中的类型数据关联起来。

在你的Java程序中，你可以得到并使用指向Class对象的引用。Class类中的一个静态方法可以让用户得到任何已装载的类的Class实例的引用。

```
// A method declared in class java.lang.Class:
public static Class forName(String className);
```

比如，如果调用forName（"java.lang.Object"），那么将得到一个代表java.lang.Object的Class对象的引用。如果调用forName（"java.util.Enumeration"），那么得到的是代表java.util包中java.util.Enumeration接口的Class对象的引用。可以使用forName（）来得到代表任何包中任何类型的Class对象的引用，只要这个类型可以被（或者已经被）装载到当前命名空间中。如果虚拟机无法把请求的类型装载到当前命名空间，那么forName（）会抛出ClassNotFoundException异常。

另一个得到Class对象引用的方法是，可以调用任何对象引用的getClass（）方法。这个方法被来自Object类本身的所有对象继承：

```
// A method declared in class java.lang.Object:
public final Class getClass();
```

比如，如果你有一个到java.lang.Integer类的对象的引用，那么你只需简单地调用Integer对象引用的getClass（）方法，就可以得到表示java.lang.Integer类的Class对象。

给出一个指向Class对象的引用，就可以通过Class类中定义的方法来找出这个类型的相关信息。如果查看这些方法，会很快意识到，Class类使得运行程序可以访问方法区中保存的信息。下面是Class类中声明的方法：

```
// Some of the methods declared in class java.lang.Class:
public String getName();
public Class getSuperClass();
public boolean isInterface();
public Class[] getInterfaces();
public ClassLoader getClassLoader();
```

这些方法仅能返回已装载类型的信息。getName（）返回类型的全限定名，getSuperClass（）返回类型的直接超类的Class实例。如果类型是java.lang.Object类或者是一个接口，它们都没有

超类，getSuperClass（）返回null。isInterface（）判断该类型是否是接口，如果Class对象描述一个接口就返回true；如果它描述一个类则返回false。getInterfaces（）返回一个Class对象数组，其中每个Class对象对应一个直接超接口，超接口在数组中以类型声明超接口的顺序出现。如果该类型没有直接超接口，getInterfaces（）则返回一个长度为零的数组。getClassLoader（）返回装载该类型的ClassLoader对象的引用，如果类型是由启动类装载器装载的，则返回null。所有这些信息都直接从方法区中获得。

方法表　为了尽可能提高访问效率，设计者必须仔细设计存储在方法区中的类型信息的数据结构，因此，除了以上讨论的原始类型信息，实现中还可能包括其他数据结构以加快访问原始数据的速度，比如方法表。虚拟机对每个装载的非抽象类，都生成一个方法表，把它作为类信息的一部分保存在方法区。方法表是一个数组，它的元素是所有它的实例可能被调用的实例方法的直接引用，包括那些从超类继承过来的实例方法。（对于抽象类和接口，方法表没有什么帮助，因为程序决不会生成它们的实例。）运行时可以通过方法表快速搜寻在对象中调用的实例方法。方法表在第8章将深入探讨。

方法区使用示例　为了展示虚拟机如何使用方法区中的信息，我们举个例子，看下面这个类：

```
// On CD-ROM in file jvm/ex2/Lava.java
class Lava {

    private int speed = 5; // 5 kilometers per hour

    void flow() {
    }
}

// On CD-ROM in file jvm/ex2/Volcano.java
class Volcano {

    public static void main(String[] args) {
        Lava lava = new Lava();
        lava.flow();
    }
}
```

下面的段落描述了某个实现中是如何执行Volcano程序中main（）方法的字节码中第一条指令的。不同的虚拟机实现可能会用完全不同的方法来操作，下面描述的只是其中一种可能——但并不是仅有的一种，下面看一下Java虚拟机是如何执行Volcano程序中main（）方法的第一条指令的。

要运行Volcano程序，首先得以某种“依赖于实现的”方式告诉虚拟机“Volcano”这个名字。之后，虚拟机将找到并读入相应的class文件“Volcano.class”，然后它会从导入的class文件里的二进制数据中提取类型信息并放到方法区中。通过执行保存在方法区中的字节码，虚拟机开始执行main（）方法，在执行时，它会一直持有指向当前类（Volcano类）的常量池（方法区中的一个数据结构）的指针。

注意，虚拟机开始执行Volcano类中main（）方法的字节码的时候，尽管Lava类还没被装载，但是和大多数（也许所有）虚拟机实现一样，它不会等到把程序中用到的所有类都装载后才开始运行程序。恰好相反，它只在需要时才装载相应的类。

main（）的第一条指令告知虚拟机为列在常量池第一项的类分配足够的内存。所以虚拟机使用指向Volcano常量池的指针找到第一项，发现它是一个对Lava类的符号引用，然后它就检查方法区，看Lava类是否已经被装载了。

这个符号引用仅仅是一个给出了类Lava的全限定名“Lava”的字符串。为了能让虚拟机尽可能快地从一个名称找到类，设计者应当选择最佳的数据结构和算法。这里可以采用各种方法，如散列表，搜索树等等。同样的算法也可以用于实现Class类的forName（）方法，这个方法根据给定的全限定名返回Class引用。

当虚拟机发现还没有装载过名为“Lava”的类时，它就开始查找并装载文件“Lava.class”，并把从读入的二进制数据中提取的类型信息放在方法区中。

紧接着，虚拟机以一个直接指向方法区Lava类数据的指针来替换常量池第一项（就是那个字符串“Lava”）——以后就可以用这个指针来快速地访问Lava类了。这个替换过程称为常量池解析，即把常量池中的符号引用替换为直接引用。这是通过在方法区中搜索被引用的元素实现的，在这期间可能又需要装载其他类。在这里，我们替换掉符号引用的“直接引用”是一个本地指针。

终于，虚拟机准备为一个新的Lava对象分配内存。此时，它又需要方法区中的信息。还记得刚刚放到Volcano类常量池第一项的指针吗？现在虚拟机用它来访问Lava类型信息（此前刚放到方法区中的），找出其中记录的这样一个信息：一个Lava对象需要分配多少堆空间。

Java虚拟机总能够通过存储于方法区的类型信息来确定一个对象需要多少内存，但是，某个特定对象事实上需要多少内存，是跟特定实现相关的。对象在虚拟机内部的表示是由实现的设计者来决定的，本章稍后将详细讨论这个问题。

当Java虚拟机确定了一个Lava对象的大小后，它就在堆上分配这么大的空间，并把这个对象实例的变量speed初始化为默认初始值0。假如Lava类的超类Object也有实例变量，则也会在此时被初始化为相应的默认值。更多信息请参考第7章。

当把新生成的Lava对象的引用压到栈中，main（）方法的第一条指令也完成了。接下来的指令通过这个引用调用Java代码（该代码把speed变量初始化为正确初始值5）。另外一条指令将用这个引用调用Lava对象引用的flow（）方法。

5.3.5 堆

Java程序在运行时创建的所有类实例或数组都放在同一个堆中。而一个Java虚拟机实例中只存在一个堆空间，因此所有线程都将共享这个堆。又由于一个Java程序独占一个Java虚拟机实例，因而每个Java程序都有它自己的堆空间——它们不会彼此干扰。但是同一个Java程序的多个线程却共享着同一个堆空间，在这种情况下，就得考虑多线程访问对象（堆数据）的同步问题了。

Java虚拟机有一条在堆中分配新对象的指令，却没有释放内存的指令。正如你无法用Java代码去明确释放一个对象一样，字节码指令也没有对应的功能。虚拟机自己负责决定如何以及何时释放不再被运行的程序引用的对象所占据的内存。程序本身不用去考虑何时需回收对象所占

用的内存，通常，虚拟机把这个任务交给垃圾收集器。

垃圾收集　垃圾收集器的主要工作就是自动回收不再被运行的程序引用的对象所占用的内存。此外，它也可能去移动那些还在使用的对象，以此减少堆碎片。

Java虚拟机规范并没有强制规定垃圾收集器，它只要求虚拟机实现必须“以某种方式”管理自己的堆空间。举个例子，某个实现可能只有固定大小的堆空间可用，当空间填满，它就简单地抛出OutOfMemory异常，根本不去考虑回收垃圾对象的问题。这样的一个实现虽然简陋，但却是符合规范的。总之，Java虚拟机规范并没有规定具体的实现必须为Java程序准备多少内存，也没有说它必须怎么管理自己的堆空间，它仅仅告诉实现的设计者：Java程序需要从堆中为对象分配空间，并且程序本身不会主动释放它。因此堆空间的管理（包括垃圾收集）问题得由设计者自行去考虑处理方式。

Java虚拟机规范没有指定垃圾收集应该采用什么技术。这些都由虚拟机的设计者根据他们的目标、考虑所受的限制、用自己的能力去决定什么才是最好的技术。因为到对象的引用可能很多地方都存在，如Java栈、堆、方法区、本地方法栈，所以垃圾收集技术的使用在很大程度上会影响到运行时数据区的设计。在第9章中列举了几种不同的的垃圾收集技术。

和方法区一样，堆空间也不必是连续的内存区。在程序运行时，它可以动态扩展或收缩。事实上，一个实现的方法区可以在堆顶实现。换句话说，就是当虚拟机需要为一个新装载的类分配内存时，类型信息和实际对象可以都在同一个堆上。因此，负责回收无用对象的垃圾收集器可能也要负责无用类的释放（卸载）。另外，某些实现可能也允许用户或程序员指定堆的初始大小、最大最小值等等。

对象的内部表示　Java虚拟机规范并没有规定Java对象在堆中是如何表示的。对象的内部表示也影响着整个堆以及垃圾收集器的设计，它由虚拟机的实现者决定。

Java对象中包含的基本数据由它所属的类及其所有超类声明的实例变量组成。只要有一个对象引用，虚拟机就必须能够快速地定位对象实例的数据。另外，它也必须能通过该对象引用访问相应的类数据（存储于方法区的类型信息）。因此在对象中通常会有一个指向方法区的指针。

一种可能的堆空间设计就是，把堆分为两部分：一个句柄池，一个对象池，如图5-5所示。而一个对象引用就是一个指向句柄池的本地指针。句柄池的每个条目有两部分：一个指向对象实例变量的指针，一个指向方法区类型数据的指针。这种设计的好处是有利于堆碎片的整理，当移动对象池中的对象时，句柄部分只需要更改一下指针指向对象的新地址就可以了——就是在句柄池中的那个指针。缺点是每次访问对象的实例变量都要经过两次指针传递。这种对象表示的方法在图5-5中绘出。第9章有一个这种堆的交互演示——HeapOfFish applet。

另一种设计方式是使对象指针直接指向一组数据，而该数据包括对象实例数据以及指向方法区中类数据的指针。这样设计的优缺点正好与前面的方法相反，它只需要一个指针就可以访问对象的实例数据，但是移动对象就变得更加复杂。当使用这种堆的虚拟机为了减少内存碎片而移动对象的时候，它必须在整个运行时数据区中更新指向被移动对象的引用。图5-6描绘了这种表示对象的方法。

有如下几个理由要求虚拟机必须能够通过对象引用得到类（类型）数据：当程序在运行时需要转换某个对象引用为另一种类型时，虚拟机必须要检查这种转换是否被允许，被转换的对

象是否的确是被引用的对象或者它的超类型。当程序在执行instanceof操作时，虚拟机也进行了同样的检查。在这两种情况下，虚拟机都需要查看被引用的对象的类数据。最后，当程序中调用某个实例方法时，虚拟机必须进行动态绑定，换句话说，它不能按照引用的类型来决定将要调用的方法，而必须根据对象的实际类。为此，虚拟机必须再次通过对象的引用去访问类数据。

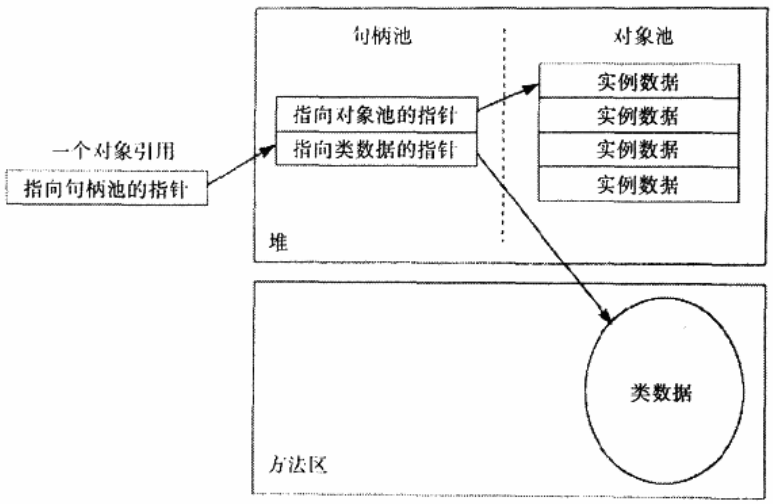


图5-5 划分为对象池和方法池的对象

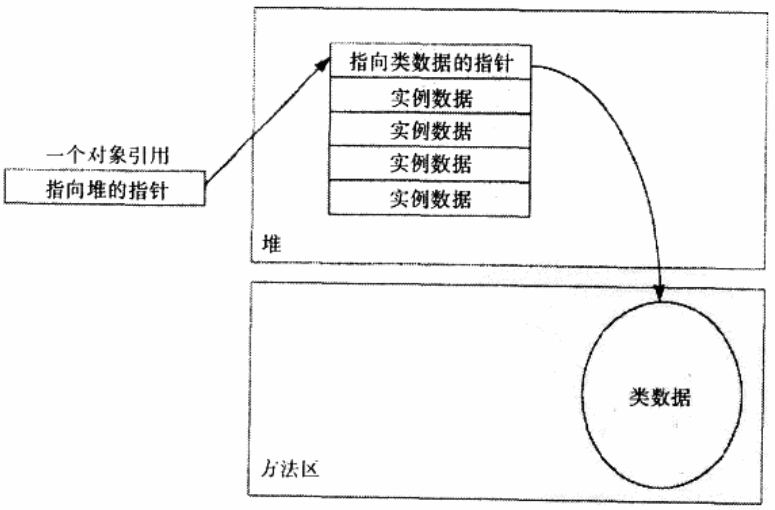


图5-6 保持对象数据在一起

不管虚拟机的实现使用什么样的对象表示法，很可能每个对象都有一个方法表，因为方法表加快了调用实例方法时的效率，从而对Java虚拟机实现的整体性能起着非常重要的正面作用。但是Java虚拟机规范并未要求必须使用方法表，所以并不是所有实现中都会使用它。比如那些具有严格内存资源限制的实现，或许它们根本不可能有足够的额外内存资源来存储方法表。如果

一个实现使用方法表，那么仅仅使用一个指向对象的引用，就可以很快地访问到对象的方法表。

图5-7展示了一种把方法表和对象引用联系起来的实现方式。每个对象的数据都包含一个指向特殊数据结构的指针，这个数据结构位于方法区，它包括两部分：

- 一个指向方法区对应类数据的指针。
- 此对象的方法表。

方法表是个指针数组，其中的每一项都是一个指向“实例方法数据”的指针，实例方法可以被那类的对象调用。方法表指向的实例方法数据包括以下信息：

- 此方法的操作数栈和局部变量区的大小。
- 此方法的字节码。
- 异常表。

这些信息足够虚拟机去调用一个方法了。方法表中包含有方法指针——指向类或其超类声明的方法的数据。也就是说，方法表所指向的方法可能是此类声明的，也可能是它继承下来的。更多关于方法表的内容可以在第8章找到。

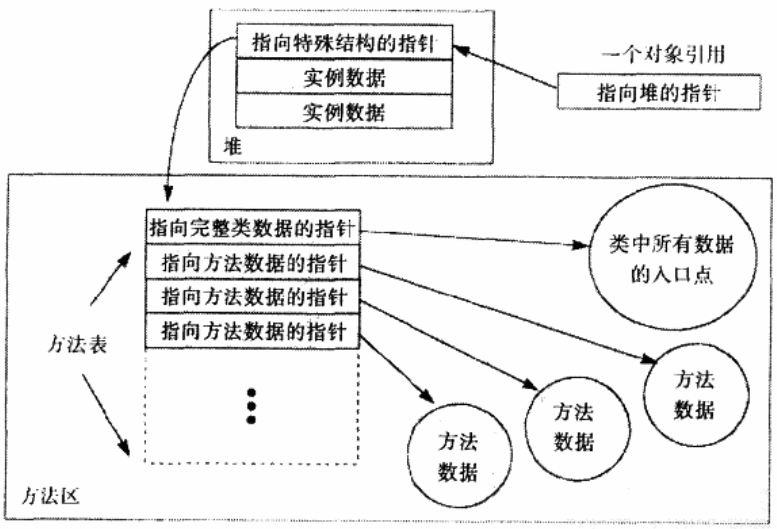


图5-7 保持方法区数据随手可用

如果读者熟悉C++的内部机制，就会发现这和C++的VTBL（C++对象的虚拟表）非常相似。在C++中，对象由实例数据和一组指向对象可以调用的虚拟函数的指针组成。Java虚拟机也可以采用这种方法，虚拟机可以在堆中为每个对象都附加一个方法表，这样较之图5-7的方法会占用更多的内存，但可轻微提高一些效率。一般来说，该方案只适用于内存足够充裕的系统。

图5-5和图5-6中显示的还有另一种数据，堆上的对象数据中还有一个逻辑部分，那就是对象锁，这是一个互斥对象。虚拟机中的每个对象都有一个对象锁，它被用于协调多个线程访问同一个对象时的同步。在任何时刻，只能有一个线程“拥有”这个对象锁，因此只有这个线程才能访问该对象的数据。此时其他希望访问这个对象的线程只能等待，直到拥有对象锁的线程释放锁。当某个线程拥有一个对象锁后，可以继续对这个锁追加请求。但请求几次，必须对应地

释放几次，之后才能轮到其他线程。比如一个线程请求了三次锁，在它释放三次锁之前，它一直保持“拥有”这个锁。

很多对象在其整个生命周期内都没有被任何线程加锁。在线程实际请求某个对象的锁之前，实现对象锁所需要的数据是不必要的。这样正如图5-5和图5-6所示，很多实现不在对象自身内部保存一个指向锁数据的指针。而只有当第一次需要加锁的时候才分配对应的锁数据，但这时虚拟机需要用某种间接方法来联系对象数据和对应的锁数据，例如把锁数据放在一个以对象地址为索引的搜索树中。

除了实现锁所需要的数据外，每个Java对象逻辑上还与实现等待集合（wait set）的数据相关联。锁是用来实现多个线程对共享数据的互斥访问的，而等待集合是用来让多个线程为完成一个共同目标而协调工作的。

等待集合由等待方法和通知方法联合使用。每个类都从Object那里继承了三个等待方法（三个名为wait（）的重载方法）和两个通知方法（notify()及notifyAll()）。当某个线程在一个对象上调用等待方法时，虚拟机就阻塞这个线程，并把它放在了这个对象的等待集合中。直到另一个线程在同一个对象上调用通知方法，虚拟机才会在之后的某个时刻唤醒一个或多个在等待集合中被阻塞的线程。正像锁数据一样，在实际调用对象的等待方法或通知方法之前，实现对象的等待集合的数据并不是必需的。因此，许多虚拟机实现都把等待集合数据与实际对象数据分开，只有在需要时才为此对象创建同步数据（通常是在第一次调用等待方法或通知方法时）。关于锁和等待集合的更多内容，请参见第20章。

最后一种数据类型——可以作为堆中某个对象映像的一部分，是与垃圾收集器有关的数据。垃圾收集器必须（以某种方式）跟踪程序引用的每个对象，这个任务不可避免地要附加一些数据给这些对象，数据的类型要视垃圾收集使用的算法而定。例如，假如垃圾收集器使用“标记并清除”算法，这就需要能够标记对象能否被引用。此外，对于不再被引用的对象，还需要指明它的终结方法（finalizer）是否已经运行过了。像线程锁一样，这些数据也可以放在对象数据外。有一些垃圾收集技术只在垃圾收集器运行时需要额外数据。例如“标记并清除”算法就使用一个独立的位图来标记对象的引用情况。几种不同的垃圾收集器技术，以及它们每一种所需要的数据，请参见第9章。

除了标记对象的引用情况外，垃圾收集器还要区分对象是否调用了终结方法。对于在其类中声明了终结方法的对象，在回收它之前，垃圾收集器必须调用它的终结方法。Java语言规范指出，垃圾收集器对每个对象只能调用一次终结方法，但是允许终结方法复活（resurrect）这个对象，即允许该对象被再次引用。这样当这个对象再次被回收时，就不用再调用终结方法了。需要终结方法的对象不多，而需要复活的更少，所以对一个对象回收两次的情况很少见。这种用来标志终结方法的数据虽然逻辑上是对象的一部分，但通常实现上不随对象保存在堆中。大部分情况下，垃圾收集器会在一个单独的空间保存这个信息。第9章有关于终结过程的详细内容。

数组的内部表示　在Java中，数组是真正的对象。和其他对象一样，数组总是存储在堆中。同样，和普通对象一样，实现的设计者将决定数组在堆中的表示形式。

和其他所有对象一样，数组也拥有一个与它们的类相关联的Class实例，所有具有相同维度和类型的数组都是同一个类的实例，而不管数组的长度（多维数组每一维的长度）是多少。例

如一个包含3个int整数的数组和一个包含300个int整数的数组拥有同一个类。数组的长度只与实例数据有关。

数组类的名称由两部分组成：每一维用一个方括号“[”表示，用字符或字符串表示元素类型。比如，元素类型为int整数的、一维数组的类名为“[I”，元素类型为byte的三维数组为“[[[B”，元素类型为Object的二维数组为“[[Ljava/lang/Object”。这些数组类的命名约定将在第6章详细讨论。

多维数组被表示为数组的数组。比如，int类型的二维数组，将表示为一个一维数组，其中的每个元素是一个一维int数组的引用，如图5-8所示。

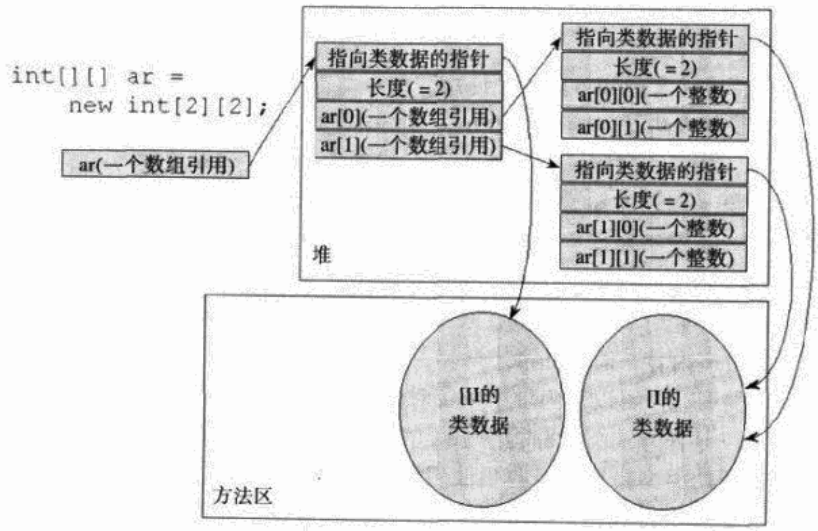


图5-8 用堆表示数组的一种可能

在堆中的每个数组对象还必须保存的数据是数组的长度、数组数据，以及某些指向数组的类数据的引用。虚拟机必须能够通过一个数组对象的引用得到此数组的长度，通过索引访问其元素（其间要检查数组边界是否越界），调用所有数组的直接超类Object声明的方法等等。

5.3.6 程序计数器

对于一个运行中的Java程序而言，其中的每一个线程都有它自己的PC（程序计数器）寄存器，它是在该线程启动时创建的。PC寄存器的大小是一个字长，因此它既能够持有一个本地指针，也能够持有一个returnAddress。当线程执行某个Java方法时，PC寄存器的内容总是下一条将被执行指令的“地址”，这里的“地址”可以是一个本地指针，也可以是在方法字节码中相对于该方法起始指令的偏移量。如果该线程正在执行一个本地方法，那么此时PC寄存器的值是“undefined”。

5.3.7 Java栈

每当启动一个新线程时，Java虚拟机都会为它分配一个Java栈。前面我们曾经提到，Java栈以帧为单位保存线程的运行状态。虚拟机只会直接对Java栈执行两种操作：以帧为单位的压栈或出栈。

某个线程正在执行的方法被称为该线程的当前方法，当前方法使用的栈帧称为当前帧，当前方法所属的类称为当前类，当前类的常量池称为当前常量池。在线程执行一个方法时，它会跟踪当前类和当前常量池。此外，当虚拟机遇到栈内操作指令时，它对当前帧内数据执行操作。

每当线程调用一个Java方法时，虚拟机都会在该线程的Java栈中压入一个新帧。而这个新帧自然就成为了当前帧。在执行这个方法时，它使用这个帧来存储参数、局部变量、中间运算结果等等数据。

Java方法可以以两种方式完成。一种通过return返回的，称为正常返回；一种是通过抛出异常而异常中止的。不管以哪种方式返回，虚拟机都会将当前帧弹出Java栈然后释放掉，这样上一个方法的帧就成为当前帧了。

Java栈上的所有数据都是此线程私有的。任何线程都不能访问另一个线程的栈数据，因此我们不需要考虑多线程情况下栈数据的访问同步问题。当一个线程调用一个方法时，方法的局部变量保存在调用线程Java栈的帧中。只有一个线程能总是访问那些局部变量，即调用方法的线程。

像方法区和堆一样，Java栈和帧在内存中也不必是连续的。帧可以分布在连续的栈里，也可以分布在堆里，或者二者兼而有之。表示Java栈和栈帧的实际数据结构由虚拟机的实现者决定，某些实现允许用户指定Java栈的初始大小和最大最小值。

5.3.8 栈帧

栈帧由三部分组成：局部变量区，操作数栈和帧数据区。局部变量区和操作数栈的大小要视对应的方法而定，它们是按字长计算的。编译器在编译时就确定了这些值并放在class文件中。而帧数据区的大小依赖于具体的实现。

当虚拟机调用一个Java方法时，它从对应类的类型信息中得到此方法的局部变量区和操作数栈的大小，并据此分配栈帧内存，然后压入Java栈中。

局部变量区　Java栈帧的局部变量区被组织为一个以字长为单位、从0开始计数的数组。字节码指令通过从0开始的索引来使用其中的数据。类型为int、float、reference和returnAddress的值在数组中只占据一项，而类型为byte、short和char的值在存入数组前都将被转换为int值，因而同样占据一项。但是类型为long和double的值在数组中却占据连续的两项。

在访问局部变量中的long和double值的时候，指令只需指出连续两项中第一项的索引值。例如某个long值占据第3、4项，那么指令会取索引为3的long值。局部变量区的所有值都是字对齐的，long和double这样占据两项数组元素的值同样可以起始于任何索引。

局部变量区包含对应方法的参数和局部变量。编译器首先按声明的顺序把这些参数放入局部变量数组。图5-9描绘了下面两个方法的局部变量区。

```
// On CD-ROM in file jvm/ex3/Example3a.java
class Example3a {

    public static int runClassMethod(int i, long l, float f,
        double d, Object o, byte b) {

        return 0;
    }
```

```
    public int runInstanceMethod(char c, double d, short s,
        boolean b) {

        return 0;
    }
}
```

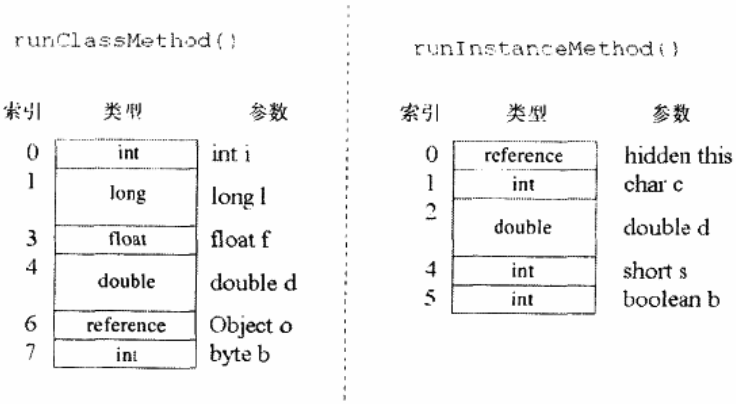


图5-9 在Java栈中本地变量区中方法的参数

注意在图5-9中显示的方法runInstanceMethod（）中，局部变量中第一个参数是一个reference（引用）类型。尽管在方法源代码中没有显式声明这个参数，但这个参数this对于任何一个实例方法都是隐含加入的，它用来表示调用该方法的对象本身。与此相反，方法runClassMethod（）中就没有这个隐含的this变量，因为它是一个类方法。类方法只与类相关，而与具体的对象无关。不能直接通过类方法访问类实例的变量，因为在方法调用的时候没有关联到一个具体实例。

我们注意到，在源代码中的byte、short、char和boolean在局部变量区都被转换成了int，在操作数栈中也一样。前面我们曾经说过，虚拟机并不直接支持boolean类型，因此Java编译器总是用int来表示boolean。但Java虚拟机对byte、short和char是直接支持的，这些类型的值可以作为实例变量或者数组元素存储在局部变量区，也可以作为类变量存储在方法区中。但在局部变量区和操作数栈中都会被转换成int类型的值。它们在栈帧中的时候都是当做int来进行处理的，只有当它被存回堆或方法区时，才会转换回原来的类型。

同样需要注意的是作为runClassMethod（）的引用被传递的参数Object o。在Java中，所有的对象都按引用传递，并且都存储在堆中，永远都不会在局部变量区或操作数栈中发现对象的拷贝，只会有对象引用。

除了Java方法的参数（编译器首先严格按照它们的声明顺序放到局部变量数组中，而对于真正的局部变量，它可以任意决定放置顺序，甚至可以用一个索引指代两个局部变量——比如当两个局部变量的作用域不重叠时，像下面Example3b中的局部变量i和j就是这种情形。在方法的前半段，在j开始生效之前，0号索引的入口可以被用来代表i。在方法的后半段，i已经超过了有效作用域，0号入口就可以用来表示j了。

```
// On CD-ROM in file jvm/ex3/Example3b.java
class Example3b {

    public static void runtwoLoops() {

        for (int i = 0; i < 10; ++i) {
            System.out.println(i);
        }

        for (int j = 9; j >= 0; --j) {
            System.out.println(j);
        }
    }
}
```

和其他运行时内存区一样，虚拟机的实现者可以为局部变量区设计任意的数据结构。比如对于怎样把long和double类型的值存储到两个数组项中，Java虚拟机规范没有指定。假如某个虚拟机实现的字长为64位，这时就可以把整个long或double数据放在数组中相邻两数组项的低项内，而使高项保持为空。

操作数栈　和局部变量区一样，操作数栈也是被组织成一个以字长为单位的数组。但是和前者不同的是，它不是通过索引来访问，而是通过标准的栈操作——压栈和出栈——来访问的。比如，如果某个指令把一个值压入到操作数栈中，稍后另一个指令就可以弹出这个值来使用。

虚拟机在操作数栈中存储数据的方式和在局部变量区中是一样的，如int、long、float、double、reference和returnType的存储。对于byte、short以及char类型的值在压入到操作数栈之前，也会被转换为int。

不同于程序计数器，Java虚拟机没有寄存器，程序计数器也无法被程序指令直接访问。Java虚拟机的指令是从操作数栈中而不是从寄存器中取得操作数的，因此它的运行方式是基于栈的而不是基于寄存器的。虽然指令也可以从其他地方取得操作数，比如从字节码流中跟随在操作码（代表指令的字节）之后的字节中或从常量池中，但是主要还是从操作数栈中获得操作数。

虚拟机把操作数栈作为它的工作区——大多数指令都要从这里弹出数据，执行运算，然后把结果压回操作数栈。比如，iadd指令就要从操作数栈中弹出两个整数，执行加法运算，其结果又压回到操作数栈中。看看下面的示例，它演示了虚拟机是如何把两个int类型的局部变量相加，再把结果保存到第三个局部变量的：

```
iload_0    // push the int in local variable 0
iload_1    // push the int in local variable 1
iadd       // pop two ints, add them, push result
istore_2   // pop int, store into local variable 2
```

在这个字节码序列里，前两个指令iload_0和iload_1将存储在局部变量区中索引为0和1的整数压入操作数栈中，其后iadd指令从操作数栈中弹出那两个整数相加，再将结果压入操作数栈。第四条指令istore_2则从操作数栈中弹出结果，并把它存储到局部变量区索引为2的位置。图5-10详细表述了这个过程中局部变量和操作数栈的状态变化，图中没有使用的局部变量区和操作数

栈区域以空白表示。

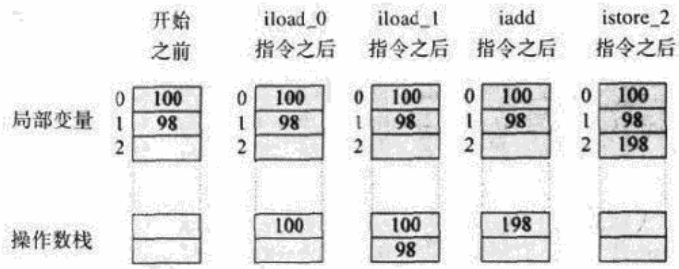


图5-10 两个局部变量的相加过程

帧数据区 除了局部变量区和操作数栈外，Java栈帧还需要一些数据来支持常量池解析、正常方法返回以及异常派发机制。这些信息都保存在Java栈帧的帧数据区中。

Java虚拟机中的大多数指令都涉及到常量池入口。有些指令仅仅是从常量池中取出数据然后压入Java栈（这些数据的类型包括int、long、float、double和String）；还有些指令使用常量池的数据来指示要实例化的类或数组、要访问的字段，或要调用的方法；还有些指令需要常量池中的数据才能确定某个对象是否属于某个类或实现了某个接口。

每当虚拟机要执行某个需要用到常量池数据的指令时，它都会通过帧数据区中指向常量池的指针来访问它。以前讲过，常量池中对类型、字段和方法的引用在开始时都是符号。当虚拟机在常量池中搜索的时候，如果遇到指向类、接口、字段或者方法的入口，假若它们仍然是符号，虚拟机那时候才会（也必须）进行解析。

除了用于常量池的解析外，帧数据区还要帮助虚拟机处理Java方法的正常结束或异常中止。如果是通过return正常结束，虚拟机必须恢复发起调用的方法的栈帧，包括设置PC寄存器指向发起调用的方法中的指令——即紧跟着调用了完成方法的指令的下一个指令。假如方法有返回值，虚拟机必须将它压入到发起调用的方法的操作数栈。

为了处理Java方法执行期间的异常退出情况，帧数据区还必须保存一个对此方法异常表的引用。异常表会在第17章深入描述，它定义了在这个方法的字节码中受catch子句保护的范围，异常表中的每一项都有一个被catch子句保护的代码的起始和结束位置（译者注：即try子句内部的代码），可能被catch的异常类在常量池中的索引值，以及catch子句内的代码开始的位置。

当某个方法抛出异常时，虚拟机根据帧数据区对应的异常表来决定如何处理。如果在异常表中找到了匹配的catch子句，就会把控制权转交给catch子句内的代码。如果没有发现，方法会立即异常中止。然后虚拟机使用帧数据区的信息恢复发起调用的方法的帧，然后在发起调用的方法的上下文中重新抛出同样的异常。

除了上述信息（支持常量池解析、正常方法返回和异常派发的数据）外，虚拟机的实现者也可以将其他信息放入帧数据区，如用于调试的数据等。

Java栈的可能实现方式 实现的设计者可以任意按自己的想法设计Java栈，正如前面提到的，一个可能的方式就是从堆中分配每一个帧。例如，考虑下面的类：

```
// On CD-ROM in file jvm/ex3/Example3c.java
```

```
class Example3c {

    public static void addAndPrint() {
        double result = addTwoTypes(1, 88.88);
        System.out.println(result);
    }

    public static double addTwoTypes(int i, double d) {
        return i + d;
    }
}
```

图5-11显示了一个线程执行addAndPrint（）方法的三次快照。在这个Java虚拟机的实现中，每个帧都单独从堆中分配。为了调用方法addTwoTypes（），方法addAndPrint（）首先把int 1和double 88.88压入到它的操作数栈中，然后调用addTwoTypes（）方法。

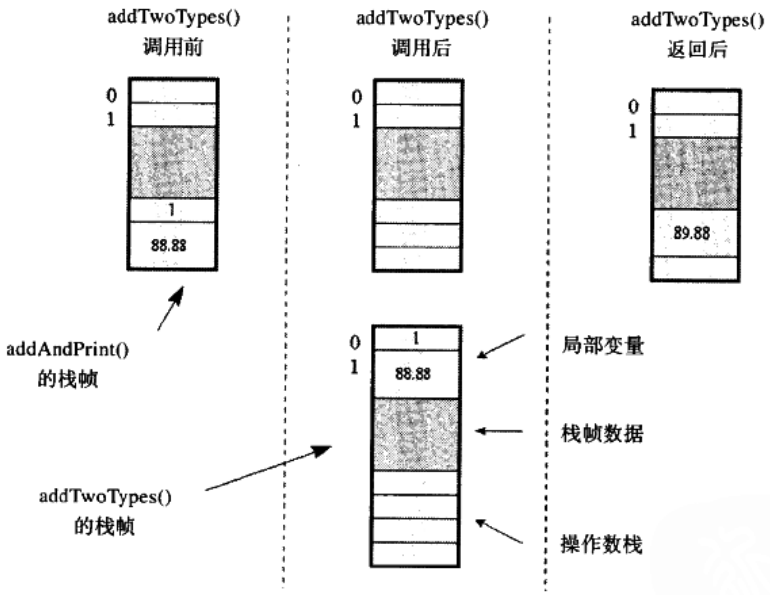


图5-11 帧的分配

调用addTwoTypes（）的指令指向一项常量池的数据，因此虚拟机在常量池中查找这些数据，这期间如有必要还需要进行解析。

注意addAndPrint（）方法也要使用常量池才能确定addTwoTypes（）方法——尽管这两个方法是属于一个类的。由此可见，和引用其他类的字段或方法一样，对同一个类的方法和字段的引用初始时也是符号，因此在使用之前同样需要解析。

解析后的常量池数据项将指向方法区中对应方法addTwoTypes（）的信息。虚拟机需要使用这些信息来决定addTwoTypes（）的局部变量区和操作数栈的大小。如果使用Sun的javac编译器

（JDK 1.1）的话，addTwoTypes（）的局部变量区需要3个字长，操作数栈需要4个字长（帧数据区的大小依赖于具体的实现）。虚拟机紧接着从堆中为这个方法分配足够大的栈帧。然后从方法addAndPrint（）的操作数栈中弹出double参数和int参数（88.88和1），并把它们分别放在addTwoTypes（）的局部变量区中索引为1和0的位置。

当addTwoTypes（）返回时，它首先把类型为double的返回值（这里是89.88）压入自己的操作数栈里。紧接着虚拟机使用帧数据区中的信息找到调用者（即addAndPrint（））的栈帧，然后将返回值压入addAndPrint（）的操作数栈中并释放方法addTwoTypes（）的栈帧所占用的内存。最后虚拟机把addTwoTypes（）的栈帧作为当前帧，从调用执行的下一条指令开始继续执行方法addAndPrint（）。

图5-12显示了另一种虚拟机实现执行同一方法的Java栈快照。它的栈帧不是从堆中单独分配，而是从一个连续的栈中分配，因而这种方式允许相邻方法的栈帧可以相互重叠。这里调用者的操作数栈部分（它包含要传给被调用者的参数）就成了被调者的局部变量区的底层。在这个例子中，addAndPrint（）的整个操作数栈刚好成为addTwoType（）的整个局部变量区。

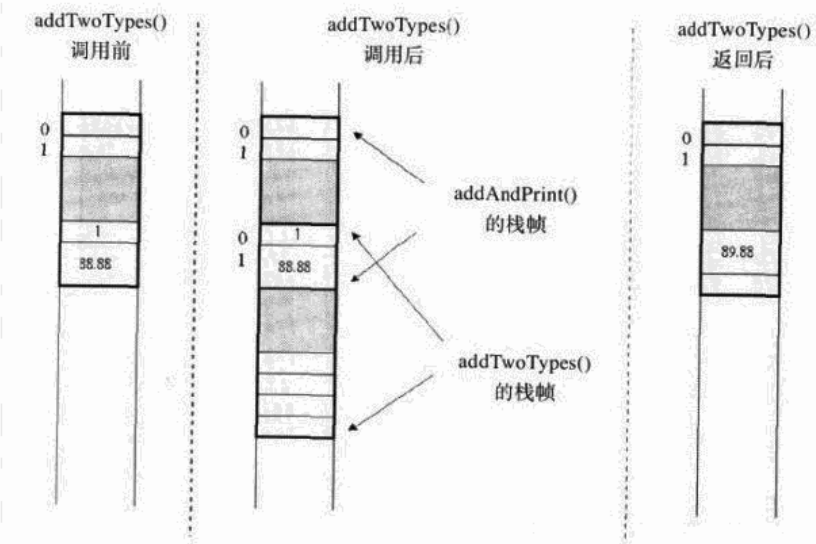


图5-12 从一个连续的栈中分配帧

这种方式不仅节省了内存空间，因为发起调用的方法和被调用的方法用相同的内存保存参数，而且也节省了时间，因为虚拟机不再需要费时地把参数从一个栈帧拷贝到另一个栈帧中了。

注意当前帧的操作数栈总是在Java栈的“顶”部。尽管这样可能可以更好地说明图5-12中连续内存的实现，但不管Java栈是如何实现的（前面说过，在本书中所有涉及栈的图形中，栈是从上向下生长的。栈的“顶部”一直在图形中处于底部），对操作数栈的压入（或者从栈中弹出）总是在当前帧执行的。这样，在当前帧的操作数栈中压入一个值也可以看做是往整个Java栈压入一个值。在本书的剩余部分，我们说“把一个值压入栈”都是指把值压入当前帧的操作数栈。

Java栈还有一些其他的实现方式，但基本上都是图5-11和图5-12两种情形的混合。比如虚拟

机可以在线程启动时就从栈中分出一大段空间，之后只要还在这段连续的空间里，虚拟机都可以采用如图5-12所示的重叠方法。如果栈生长超过了这段连续空间，虚拟机可以再从堆中分配另一段空间。如果发起调用的方法的栈帧位于旧的那段空间中，而被调用的方法的栈帧位于新的那段空间中，就使用如图5-11所示的方法把它们连接起来。在新的空间段中，虚拟机又可以继续使用连续内存方法。

5.3.9 本地方法栈

前面提到的所有运行时数据区都是在Java虚拟机规范中明确定义的，除此之外，对于一个运行中的Java程序而言，它还可能会用到一些跟本地方法相关的数据区。当某个线程调用一个本地方法时，它就进入了一个全新的并且不再受虚拟机限制的世界。本地方法可以通过本地方法接口来访问虚拟机的运行时数据区，但不止于此，它还可以做任何它想做的事情。比如，它甚至可以直接使用本地处理器中的寄存器，或者直接从本地内存的堆中分配任意数量的内存等等。总之，它和虚拟机拥有同样的权限（或者说能力）。

本地方法本质上是依赖于实现的，虚拟机实现的设计者们可以自由地决定使用怎样的机制来让Java程序调用本地方法。

任何本地方法接口都会使用某种本地方法栈。当线程调用Java方法时，虚拟机会创建一个新的栈帧并压入Java栈。然而当它调用的是本地方法时，虚拟机会保持Java栈不变，不再在线程的Java栈中压入新的帧，虚拟机只是简单地动态连接并直接调用指定的本地方法。可以把这看做是虚拟机利用本地方法来动态扩展自己。就如同Java虚拟机的实现在按照其中运行的Java程序的吩咐，调用属于虚拟机内部的另一个（动态连接的）方法。

如果某个虚拟机实现的本地方法接口是使用C连接模型的话，那么它的本地方法栈就是C栈。我们知道，当C程序调用一个C函数时，其栈操作都是确定的。传递给该函数的参数以某个确定的顺序压入栈，它的返回值也以确定的方式传回调用者。同样，这就是该虚拟机实现中本地方法栈的行为。

很可能本地方法接口需要回调Java虚拟机中的Java方法（这也是由设计者决定的），在这种情形下，该线程会保存本地方法栈的状态并进入到另一个Java栈。

图5-13描绘了这种情况，就是当一个线程调用一个本地方法时，本地方法又回调虚拟机中的另一个Java方法。这幅图展示了Java虚拟机内部线程运行的全景图。一个线程可能在整个生命周期中都执行Java方法，操作它的Java栈；或者它可能毫无障碍地在Java栈和本地方法栈之间跳转。

如图5-13所示，该线程首先调用了两个Java方法，而第二个Java方法又调用了一个本地方法，这样导致虚拟机使用了一个本地方法栈。图中的本地方法栈显示为一个连续的内存空间。假设这是一个C语言栈，其间有两个C函数，它们都以包围在虚线中的灰色块表示。第一个C函数被第二个Java方法当做本地方法调用，而这个C函数又调用了第二个C函数。之后第二个C函数又通过本地方法接口回调了一个Java方法（第三个Java方法），最终这个Java方法又调用了一个Java方法（它成为图中的当前方法）。

就像其他运行时内存区一样，本地方法栈占用的内存区也不必是固定大小的，它可以根据需要动态扩展或者收缩。某些实现也允许用户或者程序员指定该内存区的初始大小以及最大、最小值。

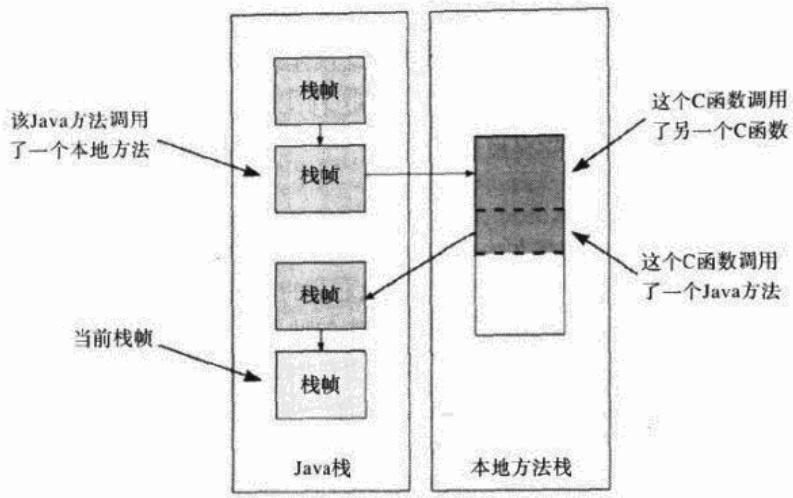


图5-13 一个线程调用Java方法和本地方法时的栈

5.3.10 执行引擎

任何Java虚拟机实现的核心都是它的执行引擎。在Java虚拟机规范中，执行引擎的行为使用指令集来定义。对于每条指令，规范都详细规定了当实现执行到该指令时应该处理什么，但是却对如何处理言之甚少。在前面的章节中提到过，实现的设计者有权决定如何执行字节码。实现可以采取解释、即时编译或直接用芯片上的指令执行，还可以是它们的混合，或任何你能想到的新技术。

和本章开头提到的对“Java虚拟机”这个术语有三种不同的理解一样，“执行引擎”这个术语也可以有三种理解：一个是抽象的规范，一个是具体的实现，另一个是正在运行的实例。抽象规范使用指令集规定了执行引擎的行为。具体实现可能使用多种不同的技术——包括软件方面、硬件方面或数种技术的集合。作为运行时实例的执行引擎就是一个线程。

运行中Java程序的每一个线程都是一个独立的虚拟机执行引擎的实例。从线程生命周期的开始到结束，它要么在执行字节码，要么在执行本地方法。一个线程可能通过解释或者使用芯片级指令直接执行字节码，或者间接通过即时编译器执行编译过的本地代码。Java虚拟机的实现可能用一些对用户程序不可见的线程，比如垃圾收集器。这样的线程不需要是实现的执行引擎的实例。所有属于用户运行程序的线程，都是在实际工作的执行引擎。

指令集　方法的字节码流是由Java虚拟机的指令序列构成的。每一条指令包含一个单字节的操作码，后面跟随0个或多个操作数。操作码表明需要执行的操作；操作数向Java虚拟机提供执行操作码需要的额外信息。操作码本身就已经规定了它是否需要跟随操作数，以及如果有操作数的话，它是什么形式的。很多Java虚拟机的指令不包含操作数，仅仅是由一个操作码字节构成的。根据操作码的需要，虚拟机可能除了跟随操作码的操作数之外，还需要从另外一些存储区域得到操作数。当虚拟机执行一条指令的时候，可能使用当前常量池中的项、当前帧的局部变量中的值，或者位于当前帧操作数栈顶端的值。

抽象的执行引擎每次执行一条字节码指令。Java虚拟机中运行的程序的每个线程（执行引擎实例）都执行这个操作。执行引擎取得操作码，如果操作码有操作数，取得它的操作数。它执行操作码和跟随的操作数规定的动作，然后再取得下一个操作码。这个执行字节码的过程在线程完成前将一直持续，通过从它的初始方法返回，或者没有捕获抛出的异常都可以标志着线程的完成。

执行引擎会不时遇到请求本地方法调用的指令。在这个时候，虚拟机负责试着发起这个本地方法调用。如果本地方法返回了（假设是正常返回，而不是抛出了一个异常），执行引擎会继续执行字节码流中的下一条指令。

可以这样来看，本地方法是Java虚拟机指令集的一种可编程扩展。如果一条指令请求一个对本地方法的调用，执行引擎就会调用这个本地方法。运行这个本地方法就是Java虚拟机对这条指令的执行。当本地方法返回了，虚拟机继续执行下一条指令。如果本地方法异常中止了（抛出了一个异常），虚拟机就按照好比是这条指令抛出这个异常一样的步骤来处理这个异常。

执行一条指令包含的任务之一就是决定下一条要执行的是什么指令。执行引擎决定下一个操作码时有三种方法。很多指令的下一个操作码就是在当前操作码和操作数之后紧跟的那个字节（如果字节码流里面还有的话）。另外一些指令，比如goto或者return，执行引擎决定下一个操作码时把它当做当前执行指令的一部分。假若一条指令抛出了一个异常，那么执行引擎将搜索合适的catch子句，以决定下一个将执行的操作码是什么。

有些指令可以抛出异常。比如，athrow指令，就明确地抛出一个异常。这条指令就是Java源代码中的throw语句的编译后形式。每当执行一条athrow指令的时候，它都将抛出一个异常。其他抛出异常的指令都只有在满足某些特定条件的时候才抛出异常。比如，假若Java虚拟机发现程序试图用0除一个整数，它就会抛出一个ArithmeticException异常。这只有在执行四条特定的除法指令（idev、ldiv、irem和lrem）的时候，或者计算int或者long的余数的时候，才可能发生。

Java虚拟机指令集的每种操作码都有助记符。使用典型的汇编语言风格，Java字节码流可以用助记符跟着（可选的）操作数值来表示。

方法的字节码流和助记符的例子如下所示，考虑这个类里的doMathForever（）方法：

```
// On CD-ROM in file jvm/ex4/Act.java
class Act {

    public static void doMathForever() {
        int i = 0;
        for (;;) {
            i += 1;
            i *= 2;
        }
    }
}
```

doMathForever（）方法的字节码流可以被反汇编成下面的助记符。Java虚拟机规范中没有定义正式的表示方法字节码的助记符的语法。下面显示的代码说明了本书所采用的用助记符表示字节码的方式。左边的列表示每条指令开始时从字节码的开头开始算起的每条指令的字节偏移量；

中间的列表示指令和它的操作数；右边的列包含注释，用双斜杠隔开，如同Java源代码的格式。

```
// Bytecode stream: 03 3b 84 00 01 1a 05 68 3b a7 ff f9
// Disassembly:
// Method void doMathForever()
// Left column: offset of instruction from beginning of method
// |   Center column: instruction mnemonic and any operands
// |   |                   Right column: comment
   0   iconst_0            // 03
   1   istore_0            // 3b
   2   iinc 0, 1           // 84 00 01
   5   iload_0             // 1a
   6   iconst_2            // 05
   7   imul                // 68
   8   istore_0            // 3b
   9   goto 2              // a7 ff f9
```

这种表示助记符的方式和Sun的Java 2 SDK里的javap程序的输出很相似。使用javap可以查看任何class文件中方法的字节码助记符。请注意跳转地址是按照从方法起始开始算起的偏移量来给出的。goto指令导致虚拟机跳转到从方法起始计算的位于偏移量2的指令。实际上操作数是负7。要执行这条指令，虚拟机在当前PC寄存器的内容上加上这个操作数。结果就是iinc指令的地址：偏移量2。为了让助记符更加易读，所看到的这条跳转指令后面的操作数已经是经过计算后的结果了。助记符显示的是“goto 2”，而不是“goto –7”。

Java虚拟机指令集关注的中心是操作数栈。一般是把将要使用的值会压入栈中。虽然Java虚拟机没有保存任意值的寄存器，但每个方法都有一个局部变量集合。指令集实际的工作方式就是把局部变量当做寄存器，用索引来访问。不过，不同于iinc指令——它可以直接增加一个局部变量的值，要使用保存在局部变量中的值之前，必须先将它从压入操作数栈。

举例来说，用一个局部变量除另外一个，虚拟机必须把它们都压入栈，执行除法，然后把结果重新保存到局部变量。要把数组元素或对象的字段保存到局部变量中，虚拟机必须先把值压入栈，然后保存到局部变量中去。要把保存在局部变量中的值赋于数组元素或者对象字段，虚拟机必须按照相反的步骤操作。它首先必须把局部变量的值压入栈，然后从栈中弹出，再放入位于堆上的数组元素或对象字段中。

Java虚拟机指令集的设计遵循几个不同的目标，但它们之间是有冲突的。这几个目标就是本书的前面所描述的整个Java体系结构的目的所在：平台无关性、网络移动性以及安全性。

平台无关性是影响指令集设计的最大因素。指令集的这种以栈为中心、而非以寄存器为中心的设计方法，使得在那些只有很少的寄存器，或者寄存器很没有规律的机器上实现Java更便利，Intel 80X86就是一个例子。由于指令集具有这种以栈为中心的特征，所以在很多平台体系结构上都很容易实现Java虚拟机。

Java以栈为中心设计指令集的另一个动机是，编译器一般采用以栈为基础的结构向连接器或优化器传递编译的中间结果。Java class文件在很多方面都和C编译器产生的.o文件（UNIX）或者.obj文件（Windows）很相似，实际上它表示了一种Java程序的中间编译结果形式。对于Java

的情况来，虚拟机是作为（动态）连接器使用的，也可以作为优化器。在Java虚拟机指令集设计中，以栈为中心的体系结构可以将运行时进行的优化工作与执行即时编译或者自适应优化的执行引擎结合起来。

在第4章中讲过，设计中一个主要考虑因素是class文件的紧凑性。紧凑性对于提高在网络上传递class文件的速度是很重要的。在class文件中保存的字节码，除了两个处理表跳转的指令之外，都是按照字节对齐的。操作码的总数很小，所以操作码可以只占据一个字节。这种设计策略有助于class文件的紧凑，但却是以可能影响程序运行的性能为代价的。某些Java虚拟机实现，特别是那些在芯片上执行字节码的实现，单字节的操作码可能使得一些可以提高性能的优化无法实现。同样，假若字节码流是以字对齐而非字节对齐的话，某些实现可能会得到更好的性能。（实现可以重新对齐字节码流，或者在装载类的时候把操作码转换成更加有效的形式。字节码在class文件中是按字节对齐的，在抽象方法区和执行引擎的规范中也是这么规定的。不同的具体实现可以用它们喜欢的任何形式保存装载后的字节码流。）

指导指令集设计的另一个目标就是进行字节码验证的能力，特别是使用数据流分析器进行的一次性验证。Java的安全框架需要这种验证能力。在装载字节码的时候使用数据流分析器进行一次性验证，而非在执行每条指令的时候进行验证，这样做有助于提高执行速度。在指令集中体现这个目标的表现之一，就是绝大部分操作码都指明了它们需要操作的类型。

比如说，不是简单地采用一条指令（该指令从操作数栈中取出一个字并保存到局部变量中），Java虚拟机的指令集而是采用两条指令。一条指令是istore——弹出并保存int类型；另一条指令是fstore——弹出并保存float类型。在执行的时候这两条指令所完成的功能是完全一致的：弹出一个字并保存。要区分弹出并保存的到底是int类型还是float类型，只对验证过程有重要作用。

对于某些指令，虚拟机需要知道被操作的类型，以决定如何执行操作。比如，Java虚拟机支持两种把两个字加起来并得到一个结果字的操作。一种是把字当做int处理，另一种是当做float处理。这两条指令的区别在于方便验证，同时也告诉虚拟机需要的是整数操作还是浮点数操作。

有一些指令可以操作任何类型。比如说dup指令不管栈顶的字是什么类型都可以复制它。还有一些指令不操作有类型的值，比如goto。但是大部分指令都操作特定的类型。这种“有类型”的指令，可以使用助记符通过一个字符前缀来表明它们操作的类型。表5-2列举了不同类型的前缀。有一些指令不包含前缀，比如arraylength或instanceof，因为它们的类型是再明显不过的。arraylength操作码需要一个数组引用。instanceof操作码需要一个对象引用。

表5-2 字节码助记符的前缀

类 型	代 码	示 例	描 述
byte	b	baload	从数组装载byte类型
short	s	saload	从数组装载short类型
int	i	iaload	从数组装载int类型
long	l	laload	从数组装载long类型
char	c	caload	从数组装载char类型
float	f	faload	从数组装载float类型
double	d	daload	从数组装载double类型
reference	a	aaload	从数组装载引用类型

操作数栈中的数值必须按照适合它们类型的方式使用。比如说压入4个int，但却把它们当作两个long来做加法，这是非法的。把一个float值从局部变量压入操作数栈，然后把它作为int保存到堆中的数组中去，这也是非法的。从一个位于堆中的对象字段压入一个double值，然后把栈中最顶端的两个字作为类型引用保存到局部变量，这也是非法的。Java编译器所坚持的强类型规则对Java虚拟机实现同样也是适用的。

当执行那些与类型无关的一般性栈操作指令时，实现也必须遵守一些规则。前面讲过，不管是什么类型，dup指令压入栈中顶端那个字的拷贝。这条指令可以用在任何占据一个字的值类型上，如int、float、引用或returnAddress。但是，如果栈顶包含的是long或者double类型，它们占据了两个连续的栈空间，这时使用dup就是非法的。位于栈顶的long或者double需要用dup2指令复制两个字，在操作数栈中压入栈顶的两个字的拷贝。一般性指令不能用来切割双字值。

为了使指令集足够小，用单字节表示每一个操作码，但并不是在所有类型上都支持所有的操作。很多操作对byte、short和char都不支持。这些类型在从堆或者方法区转移到栈帧的时候被转换成int，它们被当作int来进行操作，然后在操作完成后重新保存到堆或方法去的时候再转换为byte、short或者char。

表5-3展示了Java虚拟机中保存的每个类型所对应的计算类型。这里，保存类型是堆中类型值所体现的形式。保存类型对应Java源代码中变量的类型。计算类型是这些类型在Java栈帧中体现的形式。

表5-3 Java虚拟机中的保存类型和计算类型

保存类型	堆或者方法区中的最小比特数	计算类型	Java栈帧中的字长
byte	8	int	1
short	16	int	1
int	32	int	1
long	64	long	2
char	16	int	1
float	32	float	1
double	64	double	2
reference	32	reference	1

Java虚拟机实现必须以某种方法确保数值是被对应其类型的指令所操作。实现可以在类验证过程中就预先验证字节码，或者在执行的时候验证，或者采用前两种验证方式的混合方式。字节码验证在第7章详细描述。第10章到第20章详细描述了整个指令集。

执行技术　实现可以使用多种执行技术：解释、即时编译、自适应优化、芯片级直接执行，这些在第1章中介绍过。关于执行技术要记住的最主要的一点就是，实现可以自由选择任何技术来执行字节码，只要它遵守Java虚拟机指令集的定义。

最有意义也是最迅速的执行技术之一是自适应优化。自适应优化已经在几种现有的Java虚拟机实现中使用了，如Sun的Hotspot虚拟机。它们都从早期虚拟机实现所使用的技术中得到了很多借鉴。最初的虚拟机每次解释一条字节码；第二代虚拟机加入了即时编译器，在第一次执行方法的时候先编译成本地代码，然后执行这段本地代码。也就是说，不管什么时候调用方法，总

是执行本地代码。自适应优化器搜集那些只在运行时才有效的信息，试图以某种方式把字节码解释和编译成本地代码结合起来，以得到最优化的性能。

自适应优化的虚拟机开始的时候对所有的代码都是解释运行，但是它会监视代码的执行情况。大多数程序花费80%～90%的时间用来执行10%～20%的代码。因为可以监视程序的执行情况，所以虚拟机可以意识到哪些方法是程序的“热区”——就是那10%～20%的代码，它们占整个执行时间的80%～90%。

当自适应优化的虚拟机判断出某个特定的方法是瓶颈的时候，它启动一个后台线程，把字节码编译成本地代码，非常仔细地优化这些本地代码。同时，程序仍然通过解释来执行字节码。因为程序没有中途挂起，并且只编译和优化那些“热区”（大约10%～20%的代码），虚拟机可以比传统的即时编译更注重优化性能。

自适应优化技术使程序最终能把原来占80%～90%运行时间的代码变为极度优化的、静态连接的C++本地代码，而使用的总内存数并不比全部解释Java程序大多少。换句话说，就是更快了。自适应优化的虚拟机可以保留原来的字节码，等待方法从热区移出（程序的热区在执行的过程中可能会转移）。当方法变得不再是热区的时候，取消那些编译过的代码，重新开始解释执行那些字节码。

读者可能会注意到，自适应优化方法令Java程序运行得更快，它采取的办法和程序员用来提高程序性能的方法是很相似的。不同于通常的即时编译虚拟机，自适应优化的虚拟机并不进行“过早的优化”。自适应优化的虚拟机通过解释执行字节码开始，当程序运行的时候，虚拟机“统计”程序，找到程序的“热区”——就是那10%～20%的代码，它们花费了80%～90%的运行时间。就如同一个优秀的程序员那样，自适应优化的虚拟机只对那些对性能产生重大影响的代码进行仔细优化。

但是自适应优化的情况不止这些，它还有另外的着眼点。自适应优化器可以在运行时根据Java程序的特征进行微调——特别是对“设计良好”的Java程序。根据JavaSoft Hotspot的经理David Griswold的说法，“Java比C++更加面向对象。你可以测量它，可以发现方法调用的频度，动态派发的频度，等等。这些频度（对于Java）要比C++中高得多。”现在，在一个设计良好的Java程序中，这种方法调用和动态派发的频度更加高了，因为Java程序良好设计的尺度之一就是高效率、高产出的设计——换句话说就是，使方法和对象更紧凑及内聚性更高。

这些Java程序的运行时特征，就是方法调用和动态派发的高频度发生，它们从两个方面影响性能。首先，每次动态派发都会产生相关的管理费用；其次，更重要的是方法调用降低了编译器优化的有效性。

方法调用会使优化器的有效性降低，因为优化器在不同的方法调用间不能够有效地工作，因此优化器在方法调用的时候就无法专注于代码了。方法调用频度越高，方法调用之间可以用来优化的代码就越少，优化器就变得越低效。

这个问题的标准解决方案就是内嵌——把被调用方法的方法体直接拷贝到发起调用的方法中。内嵌消除了方法调用，因此可以让优化器处理更多的代码。这可能令优化器工作更有效，代价就需要更多的运行时内存。

麻烦之处在于，在面向对象的语言（比如Java和C++）中实现内嵌，要比非面向对象的语言

（比如C）更加困难，因为面向对象语言使用了动态派发。在Java中比在C++中更加严重，因为Java的方法调用和动态派发的频度要比C++高得多。

一个C程序的标准优化静态编译器可以直接使用内嵌，因为每一个函数调用都有一个函数实现。对于面向对象语言来说，内嵌就变得复杂了，因为动态方法派发意味着一个函数调用可能有多个函数实现（方法）。换句话说，虚拟机运行时根据方法调用的对象类，可能会有很多不同的方法实现可供选择。

内嵌一个动态派发的方法调用，一种解决办法就是把所有可能在运行时被选择的方法实现都内嵌进去。这种思路的问题在于，如果有很多方法实现，就会让优化后的代码变得非常大。

自适应编译比静态编译的优点就在于，因为它是在运行时工作的，它可以使用静态编译器所无法得到的信息。比如说，对于一个特定的方法调用，就算有30个可能的方法实现，运行时可能只会有其中的两个被调用。自适应方法就可以只把这两个方法内嵌，有效地减少了优化后的代码大小。

线程　Java虚拟机规范定义了线程模型，这个模型的目标是要有助于在很多体系结构上都实现它。Java线程模型的一个目标就是使实现的设计者，在可能的情况下使用本地线程。否则，设计者可以在它们的虚拟机实现内部实现线程机制。在一台多处理器的主机上使用本地线程的好处就是，Java程序不同的线程可以在不同的处理器上并行工作。

Java线程模型的折中之一就是优先级的规范考虑最小公分母问题。Java线程可以运行于10个优先级中的任何一个。级别1是优先级最低的，而级别10是最高的。如果设计者使用本地线程，他可以用合适的方法把10个Java优先级映射到机器本地的优先级上。Java虚拟机规范对于不同优先级别的线程行为，只规定了所有最高级别的线程会得到大多数的CPU时间，较低优先级别的线程只有在所有比它优先级更高的线程全都阻塞的情况下才能保证得到CPU时间。级别低的线程在级别高的线程没有被阻塞的时候也可能得到CPU时间，但是这没有任何保证。

规范没有假设不同优先级的线程采用时间分片方式。因为并不是所有的体系结构都采用时间片（在这里，时间分片的含义是：就算没有线程被阻塞，所有优先级的所有线程都会保证得到一些CPU时间）。就算在那些采用时间片的体系结构上，用来分配时间片给不同优先级线程的算法也存在非常大的差异。

第2章中讲到，程序的正确运行不能依靠时间分片。只有在向Java虚拟机给出提示，某个线程应该比其他线程使用更多的时间，这时候才使用线程优先级。要协调多线程之间的活动，应该使用同步。

任何Java虚拟机的线程实现都必须支持同步的两个方面：对象锁定，线程等待和通知。对象锁定使独立运行的线程访问共享数据的时候互斥。线程等待和通知使得线程为了达到同一个目标而互相协同工作。运行中的程序通过Java虚拟机指令集来访问上锁机制，还通过Object类的wait（）方法、notify（）方法和notifyAll（）方法来访问线程等待和通知机制。更多的细节请参阅第20章。

在Java虚拟机规范中，Java线程的行为是通过术语——变量、主存和工作内存——来定义的。每一个Java虚拟机实例都有一个主存，用于保存所有的程序变量（对象的实例变量、数组的元素以及类变量）。每一个线程都有一个工作内存，线程用它保存所使用和赋值的变量的“工作拷

贝”。局部变量和参数，因为它们是每个线程私有的，可以从逻辑上看成是工作内存或者主存的一部分。

Java虚拟机规范定义了许多规则，用来管理线程和主存之间的低层交互行为。比如，一条规则声明：所有对基本类型的操作，除了某些对long类型和double类型的操作之外，都必须是原子级的。再比如，如果两个线程竞争，对一个int变量写了不同的两个值，就算不存在同步，变量最终会采用二者之一。变量不会包含一个不正确的值。或者说，如果一个线程赢得了竞争，把它要写的值先写入到了变量。但失败的那个线程也可以重写那个变量，覆盖那个以为自己“胜利”的线程所写入的值。

这条规则也有例外情况，即任何没有声明为volatile的long或者double变量。某些实现可能把它们作为两个原子性的32位值对待，而非一个原子性的64位值。比如说，把一个非volatile的long保存到内存，可能是两次32位的写操作。这种对于long和double的非原子操作可能导致两个竞争性的线程在试图写入不同的值到一个long或者double变量时，最终得到的是一个不正确的结果。

虽然实现的设计者不是必须对非volatile的long和double进行原子处理，但Java虚拟机规范鼓励他们这么做。这种对long和double的非原子操作，对那条“对所有基本类型的操作都必须是原子级的”的规则而言，就是一个例外。这个例外的目的是，如果处理器不有效的支持和内存交换64位的值，线程模型也能经济地实现。将来，这个例外可能被终止。然而现在，Java程序员必须确保通过同步来操作共享的long和double。

基本上，管理低层线程行为的规则，规定了一个线程何时可以做及何时必须做以下的事情：

1) 把变量的值从主存拷贝到它的工作内存。

2) 把值从它的工作内存写回到主存。

在特定条件下，规则指定了精确的和可预言的读写内存的顺序。然而另外一些条件下，规则没有规定任何顺序。规则，是设计来让Java程序员利用可以预期的行为建立多线程程序，而给实现的设计者更多的灵活性的。这种灵活性使Java虚拟机实现的设计者从标准硬件和软件技术中得到好处，它们可以提高多线程程序的性能。

所有管理线程行为的低层规则的高层含义是：如果访问某个没有被同步的变量，允许线程用任何顺序来更新主存。不使用同步，多线程程序可能在某些Java虚拟机实现上表现出令人惊讶的行为。但是通过正确地使用同步，可以创建多线程的Java程序，它们按照可以预期的方式，可以在任何Java虚拟机上工作。

5.3.11 本地方法接口

并不强求Java虚拟机实现支持任何特定的本地方法接口。有些实现可以根本不支持本地方法接口，还有一些可能支持少数几个，每一个对应一种不同的需求。

Sun的Java本地接口，或者称作JNI，是为可移植性准备的。JNI设计的可以被任何Java虚拟机实现支持，而不管它们使用何种垃圾收集或者对象表示技术。这样它能使开发者在一个特定的主机平台上，把同样的（与JNI兼容的）本地方法二进制形式连接到任何支持JNI的虚拟机实现上。

实现设计者可以选择创建一些私有的本地方法接口，扩展或者取代JNI。为了实现可移植性，JNI在指针和指针之间、指针和方法之间使用了很多间接方法。为了得到最好的性能，实现设

计者可以提供他们自己的低层本地方法接口，以便和他们所使用的特定实现结构能更加紧密地结合。设计者也可以提供比JNI更高层的本地方法接口，比如把Java对象加入到一种组件软件模型中。

为了做好工作，本地方法必须能够和Java虚拟机实例的某些内部状态有某种程度的交互。比如，本地方法接口允许本地方法完成下列部分或全部工作：

- 传递或返回数据。
- 操作实例变量或者调用使用垃圾收集的堆中的对象的方法。
- 操作类变量或者调用类方法。
- 操作数组。
- 对堆中的对象加锁，以便被当前线程独占使用。
- 在使用垃圾收集的堆中创建新的对象。
- 装载新的类。
- 抛出新的异常。
- 捕获本地方法调用的Java方法抛出的异常。
- 捕获虚拟机抛出的异步异常。
- 指示垃圾收集器某个对象不再需要。

设计一个提供这些服务的本地方法接口是非常复杂的，需要确认垃圾收集器没有释放那些正在被本地方法使用的对象。如果实现的垃圾收集器为了减少堆碎片移动了一个对象，本地方法设计必须保证下面二者之一：

1) 当对象的引用被传递给了一个本地方法之后，它可以移动。

2) 任何其引用传递给了本地方法的对象都被钉住，直到本地方法返回，或者它表明自己已经完成了对象的操作。

由此可见，本地方法接口和Java虚拟机内部工作纠缠在了一起。

5.4 真实机器

在本章一开始就说过，所有的子系统、运行时数据区和Java虚拟机规范定义的内部行为都是抽象的。设计者没有必要把实际实现按照和规范中定义的抽象元素完全对应的方式来组织。抽象的内部组件和行为只不过是一个词汇表，根据它可以定义任何Java虚拟机实现所必需的外部行为。

换句话说，一个实现内部可能是任何样子的，只要它外部看起来如同Java虚拟机那样工作即可。实现必须能够辨别Java class文件，必须遵守class文件包含的Java代码的语义。除此以外什么都可以。字节码是如何执行的，运行时数据区是如何组织的，垃圾收集器是如何运作的，线程是如何实现的，启动类装载器怎样发现类，如何支持本地方法接口，这些内容都是由设计实现者来决定的。

规范的灵活性给了设计者自由，他们可以自由裁减设计，以满足特定的需求。某些实现必须严格地限制使用很少的资源；还有一些实现，因为资源非常充裕，所以追求最高的性能可能是惟一的目标。

通过清楚地区分Java虚拟机外部行为和内部实现之间的界限，规范保持了所有实现之间的兼

容，同时也促进了创新。鼓励设计者发挥自己的天才和创意来构造更好的Java虚拟机。

5.5 一个模拟："Eternal Math"

随书光盘包含几个模拟applet，作为本书的补充材料。图5-14展示的applet模拟了一个Java虚拟机正在执行一小段字节码。读者可以通过任何使用Java技术的浏览器，或者任何支持JDK 1.0的applet浏览器来装载随书光盘的applets/EternalMath.html，这样就可以运行该applet。

模拟中的指令描述了Act类中doMathForever（）方法的方法体，它在本章前面的"指令集"部分用到了。这个模拟展示了当前帧的局部变量和操作数栈，PC寄存器，以及方法区中的字节码。它还展示了一个optop寄存器，读者可以认为这是帧数据的一部分，是这个Java虚拟机实现特有的。optop寄存器一直指向操作数栈顶之上的一个字。

这个applet有四个按钮：Step（步进），Reset（重启），Run（运行），Stop（停止）。每次按下"Step"按钮，Java虚拟机模拟会执行当前PC寄存器所指向的指令。最初，PC寄存器指向iconst_0指令，所以第一次按下"Step"按钮的时候，虚拟机会执行iconst_0并把一个0压入栈中，而且把PC寄存器指向要执行的下一条指令。不断地按下"Step"按钮，会依次执行下面的指令，PC寄存器起指路的作用。如果按下"Run"按钮，模拟会一直运行，直到按下"Stop"按钮。要重新开始模拟，请按"Reset"按钮。

每个寄存器（PC和optop）都用两种方法显示。每个寄存器的内容都是一个从方法字节码或者操作数栈开始算起的偏移量，在一个编辑框中显示。同时，用一个小箭头（"pc>"或者"optop>"）指示当前寄存器的位置。

在模拟中操作数栈当有字压入的时候是从上往下生长的（内存偏移量是向上递增的）。当有字弹出的时候，栈就向上退。

doMathForever（）方法只有一个局部变量i，它位于数组元素0的位置，开始的两条指令，iconst_0和istore_0初始化局部变量为0。下一条指令iinc，把i加1。这条指令实现了doMathForever（）中的i += 1语句。下一条指令iload_0，把局部电量的值压入操作数栈。iconst_2把2压入操作数栈。imul把操作数栈顶部的两个int弹出，执行乘法计算，再把结果压入。istore_0指令把乘法结果弹出，放入到局部变量中。这四条指令实现了doMathForever（）中的i *= 2语句。最后一条指令goto，把程序计数器送回到iinc指令。这个goto实现了doMathForever（）的for（;;）循环。

只要有足够的耐心，点击"Step"按钮（或者"Run"按钮按下后等待一段时间），会得到一次算术溢出。当Java虚拟机遇到这样的情况时，它仅仅是截断（如同在模拟中显示的那样），它不会抛出任何异常。

每按下一次"Step"，applet底部都会显示下一条要执行的指令，如图5-14所示。

5.6 随书光盘

随书光盘的jvm目录下包含了本章中用到的示例源代码，Eternal Math applet保存在applets/EternalMath.html文件中。也可以在applets/JVMSimulators目录和applets/JVMSimulators/ COM/artima/jvmsim目录下找到它的源代码和class文件。

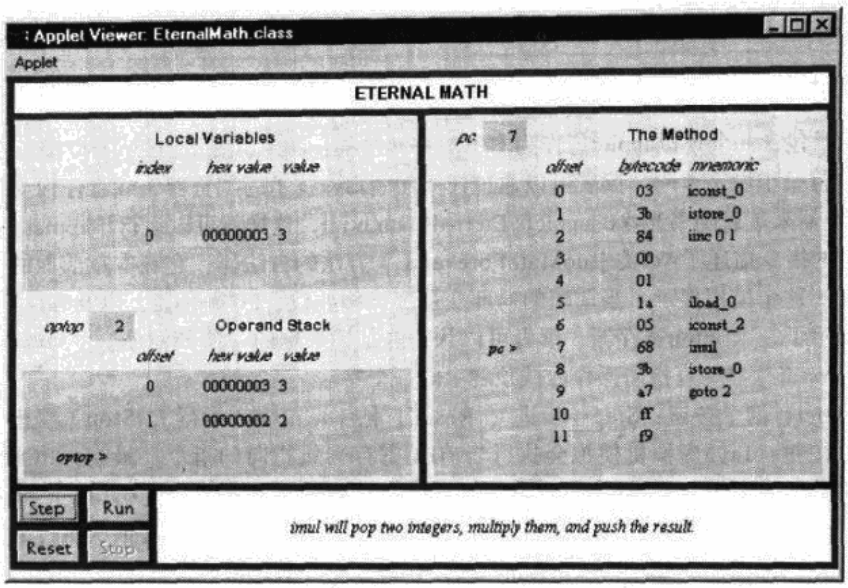


图5-14 Eternal Math applet

5.7 资源页

要了解更多关于Java虚拟机的信息，请访问资源页面：http://www.artima.com/insidejvm/resources。

第6章　Java class文件

前一章对Java虚拟机作了概述。下面四章将分别专门讨论Java虚拟机的不同方面。本章主要讨论Java class文件[⊖]，并描述了class文件的内容，包括常量池的结构及其格式等。本章可以作为Java class文件格式的全面参考。

随书光盘上有一个applet使用交互方式阐述了本章内容，名为“Getting Loaded”。“Getting Loaded”模拟了Java虚拟机读取一个Java class文件的全过程。本章最后介绍了此applet及其使用方式。

6.1　Java class文件是什么

Java class 文件是对Java程序二进制文件格式的精确定义。每一个Java class 文件都对一个Java类或者Java接口作出了全面描述。一个class文件中只能包含一个类或者接口。无论Java class文件在何种系统上产生，无论虚拟机在何种系统上运行，对Java class文件的精确定义使得所有Java虚拟机都能够正确地读取和解释所有Java class文件。

尽管class文件与Java语言结构相关，但它并不一定必须与Java语言相关。如图6-1所示，可以使用其他语言来编写程序，然后将其编译为class文件，或者把Java程序编译为另一种不同的二进制文件格式。实际上，Java class文件的形式能够表示Java源代码中无法表达的有效程序。然而，绝大多数Java开发者几乎都会选择使用class文件作为将程序传给虚拟机的首要方式。

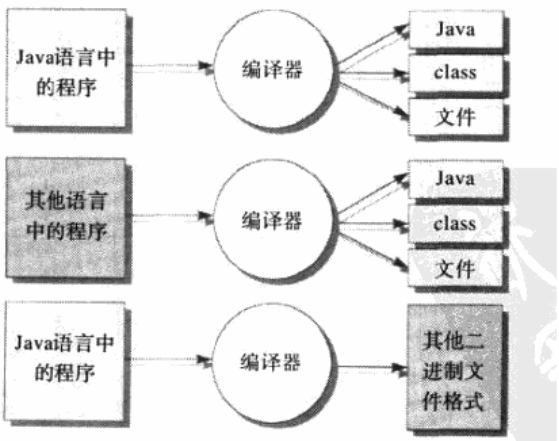


图6-1　Java语言与class文件之间非排他性的关系

⊖ 也有些译者把class file直译为类文件。因为本书是一本阐述Java虚拟机底层的书，有必要明确区分“通常以.class为后缀名的Java虚拟机可装载文件”，“.class后缀文件中包含的二进制流”，以及“.class文件装载后在内存中形成的类或者接口映像”，所以，“class文件”一词特指“通常以.class为后缀名的Java虚拟机可装载文件”，其中可能包含类，也可能包含接口。

如前所述，Java class文件是8位字节的二进制流。数据项按顺序存储在class文件中，相邻的项之间没有任何间隔，这样可以使class文件紧凑。占据多个字节空间的项按照高位在前的顺序分为几个连续的字节存放。

和Java的类可以包含多个不同的字段、方法、方法参数、局部变量等一样，Java class文件也能够包含许多不同大小的项。在class文件中，可变长度项的大小和长度位于其实际数据之前。这个特性使得class文件流可以从头到尾被顺序解析，首先读出项的大小，然后读出项的数据。

6.2 class文件的内容

Java class文件中包含了Java虚拟机所需知道的、关于类或接口的所有信息。本章剩余部分将用表格形式描述class文件格式。每个表格都有一个名字，每个表格都显示了在class文件中出现的项的有序列表。这些项按照出现在class文件中的顺序在表中列出。每一项都包括类型、名称及该项的数量。类型或者为表名，或者为如表6-1所示的“基本类型”。所有存储在类型u2、u4和u8项中的值，在class文件中以高位在前的形式出现。

表6-1 class文件“基本类型”

u1	1个字节，无符号类型
u2	2个字节，无符号类型
u4	4个字节，无符号类型
u8	8个字节，无符号类型

可变长度的ClassFile表中的项，如表6-2所示，按照它们在class文件中出现的顺序列出了主要部分。

表6-2 ClassFile表的格式

类 型	名 称	数 量
u4	magic	1
u2	minor_version	1
u2	major_version	1
u2	constant_pool_count	1
cp_info	constant_pool	constant_pool_count- 1
u2	access_flags	1
u2	this_class	1
u2	super_class	1
u2	interfaces_count	1
u2	interfaces	interfaces_count
u2	fields_count	1
field_info	fields	fields_count
u2	methods_count	1
method_info	methods	methods_count
u2	attributes_count	1
attribute_info	attributes	attributes_count

ClassFile表中各项简介如下：

(1) magic（魔数）

每个Java class文件的前4个字节被称为它的魔数（magic number）：0xCAFEBABE。魔数的作用在于，可以轻松地分辨出Java class文件和非Java class文件。如果一个文件不是以0xCAFEBABE开头，那它就肯定不是Java class文件。文件格式定义者能够自由选择魔数，前提是这个选定的魔数值没有被广泛应用。当Java还被称为“Oak”的时候，这个魔数就已经定下来了。依照Patrick Naughton（最初Java开发小组的关键成员）的说法：“早在Java第一次作为该语言的名字发布以前，我们就在寻找一些好玩的、惟一的、容易记忆的东西。选择0xCAFEBABE只不过是一个巧合，它象征着著名咖啡品牌Peet's Coffee中深受欢迎的baristas(一种咖啡的名称)，它预示了Java这个名字的出现。”

(2) minor_version和major_version

class文件的下面4个字节包含了主、次版本号。随着Java技术的发展，Java class文件格式可能会加入新特性。class文件格式一旦发生变化，版本号也会随之变化。对于Java虚拟机来说，版本号确定了特定的class文件格式，通常只有给定主版本号和一系列次版本号后，Java虚拟机才能够读取class文件。如果class文件的版本号超出了Java虚拟机所能处理的有效范围，Java虚拟机将不会处理该class文件。

在Sun的JDK 1.0.2发布版中，Java虚拟机实现支持从45.0（主版本号为45，次版本号为0）到45.3的class文件格式。在所有JDK 1.1发布版中的虚拟机都能够支持版本从45.0到45.65535的class文件格式。在Sun的1.2版本的SDK中，虚拟机能够支持从版本45.0到46.0的class文件格式。

1.0或1.2版本的编译器能够产生版本号为45.3的class文件。在Sun的1.2版本SDK中，javac编译器默认产生版本号为45.3的class文件。但如果在javac命令行中指定了-target 1.2标志，1.2版本的编译器将产生版本号为46.0的class文件。1.0或1.1版本的虚拟机上不能运行使用-target 1.2标志所产生的class文件。

Java虚拟机实现的第二版中修改了对class文件主版本号和次版本号的解释。对于第二版而言，class文件的主版本号与Java平台主发布版的版本号保持一致（例如，在Java 2平台发布版上，主版本号从45升至46），次版本号与特定主平台发布版的各个发布版相关。因此，尽管不同的class文件格式可以由不同的版本号表示，但版本号不一样并不代表class文件格式不同。版本号不同的原因可能只是因为class文件由不同发布版本的Java平台产生，可能class文件的格式并没有改变。

(3) constant_pool_count和constant_pool

在class文件中，魔数和版本号后面的是常量池。正如第5章中所述，常量池包含了与文件中类和接口相关的常量。常量池中存储了诸如文字字符串、final变量值、类名和方法名的常量。Java虚拟机把常量池组织为入口列表的形式。在实际列表constant_pool之前，是入口在列表中的计数constant_pool_count。

常量池中的许多入口都指向其他的常量池入口，而且class文件中紧随着常量池的许多条目也会指向常量池中的入口。在整个class文件中，指示常量池入口在常量池列表中位置的整数索引都指向这些常量池入口。列表中的第一项索引值为1，第二项索引值为2，以此类推。尽管constant_pool列表中没有索引值为0的入口，但缺失的这一入口也被constant_pool_count计数在

内。例如，当constant_pool中有14项（索引值从1到14）时，constant_pool_count的值为15。

每个常量池入口都从一个长度为一个字节的标志开始，这个标志指出了列表中该位置的常量类型。一旦Java虚拟机获取并解析这个标志，Java虚拟机就会知道在标志后的常量类型是什么。表6-3列出了所有常量池标志的名字和值。

表6-3中的每一个标志都有一个相对应的表，表名通过在标志名后加上“_info”后缀来产生。例如，对应于CONSTANT_Class标志的表名为CONSTANT_Class_info，表名为CONSTANT_Utf8_into的表中存储着Unicode字符串的压缩形式。对应于各种不同常量池入口的表将在本章后面详细描述。

表6-3 常量池标志

入口类型	标志值	描述
CONSTANT_Utf8	1	UTF-8编码的Unicode字符串
CONSTANT_Integer	3	int类型字面值
CONSTANT_Float	4	float类型字面值
CONSTANT_Long	5	long类型字面值
CONSTANT_Double	6	double类型字面值
CONSTANT_Class	7	对一个类或接口的符号引用
CONSTANT_String	8	String类型字面值
CONSTANT_Fieldref	9	对一个字段的符号引用
CONSTANT_Methodref	10	对一个类中声明的方法的符号引用
CONSTANT_InterfaceMethodref	11	对一个接口中声明的方法的符号引用
CONSTANT_NameAndType	12	对一个字段或方法的部分符号引用

在动态连接的Java程序中，常量池充当了十分重要的角色。除了字面常量（或者说直接量）值以外，常量池还可以容纳下面几种符号引用：

- 类和接口的全限定名。
- 字段的名称和描述符。
- 方法的名称和描述符。

字段是类或接口的实例变量或者类变量。字段的描述符是一个指示字段的类型的字符串。方法的描述符也是一个字符串，该字符串指示方法的返回值和参数的数量、顺序和类型。在运行时，Java虚拟机使用常量池的全限定名、方法和字段的描述符，把当前类或接口中的代码与其他类或接口中的代码连接起来。由于class文件并不包含其内部组件最终内存布局的信息，因此类、字段和方法并不能被class文件中的字节码直接引用。Java虚拟机从常量池获得符号引用，然后在运行时解析引用项的实际地址。例如，用来调用方法的字节码指令把一个符号引用的常量池索引传给所调用的方法。在常量池中使用符号引用的过程在第8章有更详尽的阐述。

(4) access_flags

紧接常量池后的两个字节称为access_flags，它展示了文件中定义的类或接口的几段信息。例如，访问标志指明文件中定义的是类还是接口；访问标志还定义了在类或接口的声明中，使用了哪种修饰符；类和接口是抽象的，还是公共的；类的类型可以为final，而final类不可能是抽象的；接口不能为final类型。这些标志位的定义如表6-4所示。

表6-4 ClassFile表内access_flags项的标志位

标 志 名	值	设置后的含义	设 置 者
ACC_PUBLIC	0x0001	public类型	类和接口
ACC_FINAL	0x0010	类为final类型	只有类
ACC_SUPER	0x0020	使用新型的invokespecial语义	类和接口
ACC_INTERFACE	0x0200	接口类型，不是类类型	所有的接口，没有类
ACC_ABSTRACT	0x0400	abstract类型	所有的接口，部分类

ACC_SUPER标志与Sun的老版本Java编译器向后兼容。Sun当前版本的Java虚拟机中，invokespecial指令的语义比老版本中的更为严格。所有新版本的编译器都必须设置ACC_SUPER标志。所有新的Java虚拟机实现都必须实现更新的、更严格的invokespecial语义（关于这些语法，请参见附录A中的invokespecial指令）。Sun的老版本编译器产生class文件时，将ACC_SUPER标志设为0。即使设定了这个标志，Sun的老版本Java虚拟机也将忽略它。

在access_flags中所有未使用的位都必须由编译器置0，而且Java虚拟机必须忽略它。

(5) this_class

接下来的两个字节为this_class项，它是一个对常量池的索引。在this_class位置的常量池入口必须为CONSTANT_Class_info表。该表由两个部分组成——标签和name_index。标签部分是一个具有CONSTANT_Class值的常量，在name_index位置的常量池入口为一个包含了类或接口全限定名的CONSTANT_Utf8_info表。

this_class项提供了一个如何使用常量池的范例。对于它自身来说，this_class项只是一个指向常量池的索引。当Java虚拟机在this_class位置查阅常量池入口的时候，它会发现一个通过把自己的标签设为CONSTANT_Class来识别自身的项。Java虚拟机知道，在CONSTANT_Class_info入口中，标签的后面总会有一个名为name_index的、指向常量池的索引。于是虚拟机在name_index位置查找常量池入口，在这个位置，Java虚拟机应该能找到一个容纳了类或者接口全限定名的CONSTANT_Utf8_info入口。对于这个过程，图6-2有一个图形描述。

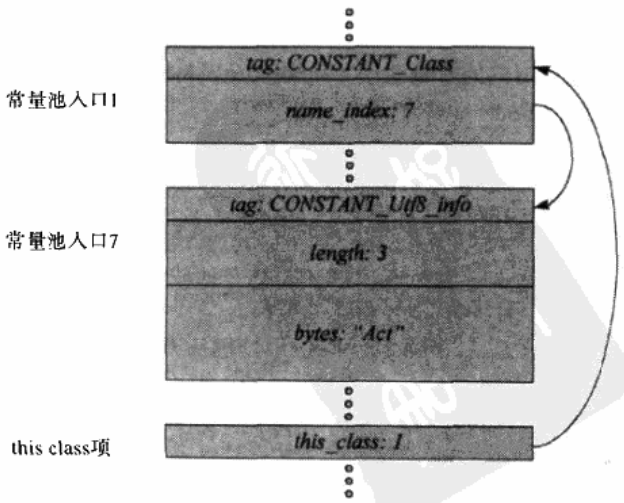


图6-2 常量池的使用示例

(6) super_class

在class文件中，紧接在this_class之后的是super_class项，它是一个两个字节的常量池索引。在super_class位置的常量池入口是一个指向该类超类全限定名的CONSTANT_Class_info入口。因为Java程序中所有对象的基类都是java.lang.Object类，除了Object类以外，常量池索引super_class对于所有的类均有效。对于Object类，super_class的值为0。对于接口，在常量池入口super_class位置的项为java.lang.Object。

(7) interfaces_count和interfaces

紧接着super_class的是interfaces_count。此项的含义为：在文件中由该类直接实现或者由接口所扩展的父接口的数量。在这个计数的后面，是名为interfaces的数组，它包含了对每个由该类或者接口直接实现的父接口的常量池索引。每个父接口都使用一个常量池中的CONSTANT_Class_info入口来描述，该CONSTANT_Class_info入口指向接口的全限定名。这个数组只容纳那些直接出现在类声明的implements子句或者接口声明的extends子句中的父接口。超类按照在implements子句和extends子句中出现的顺序（从左到右）在这个数组中显现。

(8) fields_count和fields

在class文件中，紧接在interfaces后面的是对在该类或者接口中所声明的字段的描述。首先是名为fields_count的计数，它是类变量和实例变量的字段的数量总和。在这个计数后面的是不同长度的field_info表的序列（fields_count指出了序列中有多少个field_info表）。只有在文件中由类或者接口声明了的字段才能在fields列表中列出。在fields列表中，不列出从超类或者父接口继承而来的字段。另一方面，fields列表可能会包含在对应的Java源文件中没有叙述的字段，这是因为Java编译器可能会在编译时向类或者接口添加字段。例如，对于一个内部类的fields列表来说，为了保持对外围类实例的引用，Java编译器会为每个外围类实例添加实例变量。源代码中并没有叙述任何在fields列表中的字段，它们是被Java编译器在编译时添加进去的，这些字段使用Synthetic属性标识。

每个field_info表都展示了一个字段的信息。此表包含了字段的名字、描述符和修饰符。如果该字段被声明为final，field_info表还会展示其常量值。这样的信息有些放在field_info表中，有些则放在由field_info表所指向的常量池中。field_info表将在本章后面进一步阐述。

(9) methods_count和methods

在class文件中，紧接着fields后面的是对在该类或者接口中所声明的方法的描述。首先是名为methods_count的计数，它是一个双字节长度的对于该类或者接口中声明的所有方法的总计数。这个总计数只包括在该类或者接口中显式定义的方法（从超类或者父接口中继承来的方法不被计入）。在methods_count后面的是方法本身，它在一个method_info表的列表中进行了阐述（methods_count指出了列表中有多少个method_info表）。

method_info表中包含了与方法相关的一些信息，包括方法名和描述符（方法的返回值类型和参数类型）。如果方法既不是抽象的，又不是本地的，那么method_info表就包含方法局部变量所需的栈空间长度、为方法所捕获的异常表、字节码序列以及可选的行数和局部变量表。如果方法能够抛出任何已验证的异常，那么method_info表就会包括一个关于这些已验证异常的列表。method_info将在本章后面进一步阐述。

(10) attributes_count和attributes

class文件中最后的部分是属性（attribute），它给出了在该文件中类或者接口所定义的属性的基本信息。属性部分由attributes_count开始，attributes_count是指出现在后续attributes列表中的attribute_info表的数量总和。每个attribute_info的第一项是指向常量池中CONSTANT_Utf8_info表的索引，该表给出了属性的名称。

属性有许多种。Java虚拟机规范定义了几种属性，但任何人都可以创建他们自己的属性种类（通过特定的规则），并且把它们置于class文件中。Java虚拟机实现必须忽略任何不能识别的属性。创建新属性种类的规则将在本章后面阐述。

属性出现在class文件中的多处，而不仅仅在顶层ClassFile表的attributes项中出现。出现在ClassFile表中的属性主要给出了与文件中所定义的类和接口相关的信息；出现在field_info表中的属性主要给出了与字段相关的信息；出现在method_info表中的属性主要给出了与方法相关的信息。

Java虚拟机实现定义了两种属性——SourceCode和InnerClasses，它们出现在ClassFile表中属性列表中。这两种属性将在本章后面进一步阐述。

6.3 特殊字符串

常量池中容纳的符号引用包括三种特殊的字符串：全限定名、简单名称和描述符。所有的符号引用都包括类或者接口的全限定名。字段的符号引用除了全限定类型名之外，还包括简单字段名和字段描述符。方法的符号引用除了全限定类型名之外，还包括简单方法名和方法描述符。

在符号引用中使用的特殊字符串也同样用来描述被class文件定义的类或者接口。例如，定义过的类或者接口会有一个全限定名。对于每一个在类或者接口中声明的字段，常量池中都会有一个简单名称和字段描述符。对于每一个在类或者接口中声明的方法，常量池中都会有一个简单名称和方法描述符。

6.3.1 全限定名

当常量池入口指向类或者接口时，它们给出该类或者接口的全限定名。在class文件中，全限定名中的点用斜线取代了。例如，在class文件中，java.lang.Object的全限定名表示为java/lang/Object；在class文件中，java.util.Hashtable的全限定名表示为java/util/Hashtable。

6.3.2 简单名称

字段名和方法名以简单名称（非全限定名）形式出现在常量池入口中。例如，一个指向类java.lang.Object所属方法String toString（）的常量池入口有一个形如“toString”的方法名。一个指向类java.lang.System所属字段java.io.PrintStream out的常量池入口有一个形如“out”的字段名。

6.3.3 描述符

除了类（或接口）的全限定名和简单字段（或方法）名，指向字段和方法的符号引用还包含描述符字符串。字段的描述符给出了字段的类型；方法描述符给出了方法的返回值和方法参数的数量、类型以及顺序。

字段和方法的描述符由如下所示的上下文无关语法定义。该语法中非终结符号用斜体字标

出，如*FieldType*；终结符号使用等宽度字体标出，如B或V；星号代表紧接在它前面的符号（中间没有空格）将会出现0次或者多次。

FieldDescriptor:
    FieldType
ComponentType:
    FieldType
FieldType:
    BaseType
    ObjectType
    ArrayType
BaseType:
    B
    C
    D
    F
    I
    J
    S
    Z
ObjectType:
    L<classname>;
ArrayType:
    [*ComponentType*
MethodDescriptor:
    (*ParameterDescriptor** **)** *ReturnDescriptor*
ParameterDescriptor:
    FieldType
ReturnDescriptor:
    FieldType
    V

表6-5中列出了每个基本类型终结符的含义。V终结符表示方法返回值为void类型。八种基本类型终结符中的每一个、返回值描述符终结符V、对象类型终结符L和;、数组类型终结符[，以及方法描述符终结符（和），都是ASCII字符（除了空字符null外，能够对应于ASCII字符的每一个Unicode字符在UTF-8格式中，都可以使用相对应的ASCII字符值来描述）。对象类型中的Classname部分为全限定名。这里的全限定名与class文件中的全限定名一样，都用斜线取代了点。

表6-5 基本类型终结符

终结符	类型
B	byte
C	char
D	double
F	float
I	int
J	long
S	short
Z	boolean

表6-6列出了一些字段描述符的例子，表6-7列出了一些方法描述符的例子。需要注意的是：

实例方法的方法描述符并没有包含作为第一个参数被传给所有实例方法的隐藏this参数。但所有调用实例方法的Java虚拟机指令都会隐式传递this参数（this引用永远不会传给类方法，因为类方法不会被对象调用）。

方法描述符只能包含255个字长以内的参数。传给实例方法的隐藏this参数引用占用一个字长，属于基本类型的long或者double占用两个字长，其他的参数占用一个字长。

表6-6 字段描述符示例

描述符	字段声明
I	int I;
[[J	long[][] windingRoad;
[Ljava/lang/Object;	java.lang.Object[] stuff;
Ljava/util/Hashtable;	java.util.Hashtable ht;
[[[Z	boolean[][][] isReady;

表6-7 方法描述符示例

描述符	方法声明
()I	int getSize ( ) ;
( ) Ljava/lang/String;	String toString ( ) ;
([Ljava/lang/String;)V	void main (String[] args) ;
( ) V	void wait ( )
(JI) V	void wait (long timeout， int nanos)
(ZILjava/lang/String;II) Z	boolean regionMatches (boolean ignoreCase， int toOffset， String other， int ooffset， int len) ;
([BII)I	int read (byte[] b， int off，int len) ;

6.4 常量池

常量池是一个可变长度cp_info表的有序序列。这些cp_info表的通常形式如表6-8所示。cp_info表中的tag（标志）项是一个无符号的byte类型值，它表明了表的类型和格式。cp_info表一共有11种类型，这些类型将在下面的小节中一一进行阐述。

表6-8 cp_info表的通常形式

类型	名称	数量
u1	tag	1
u1	info	根据tag值决定

6.4.1 CONSTANT_Utf8_info表

可变长度的CONSTANT_Utf8_info表使用一种UTF-8格式的变体来存储一个常量字符串。这种类型的表可以存储多种字符串，包括：

- 文字字符串，如String对象。

- 被定义的类和接口的全限定名。
- 被定义的类的超类（如果有的话）的全限定名。
- 被定义的类和接口的父接口的全限定名。
- 由类或者接口声明的任意字段的简单名称和描述符。
- 由类或者接口声明的任意方法的简单名称和描述符。
- 任何引用的类和接口的全限定名。
- 任何引用的字段的简单名称和描述符。
- 任何引用的方法的简单名称和描述符。
- 与属性相关的字符串。

如下文所示：CONSTANT_Utf8_info表中存储了四种基本信息类型：文字字符串、被定义的类和接口描述、对其他类或接口的符号引用以及与属性相关的字符串。一些与属性相关的字符串如：属性名称、产生class文件的源文件名称、局部变量的名称以及描述符。

UTF-8编码模式允许字符串中的所有Unicode字符以2个字节的形式表示，而ASCII字符（空字符null除外）以一个字节的形式表示。表6-9列出了CONSTANT_Utf8_info的格式。

表6-9 CONSTANT_Utf8_info表的格式

类 型	名 称	数 量
u1	tag	1
u2	length	1
u1	bytes	length

CONSTANT_Utf8_info表中各项如下：

tag

tag项的值为CONSTANT_Utf8（1）。

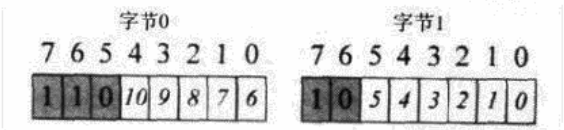
length

length项给出了后续bytes项的长度（字节数）。

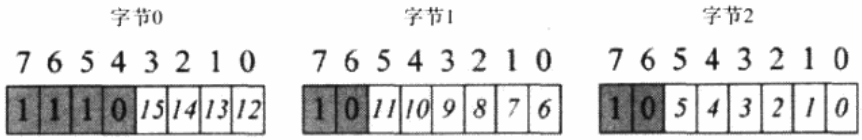
bytes

bytes项中包含按照变体UTF-8格式存储的字符串中的字符。从‘\u0001’到‘\u007f’的所有字符（除空字符null外所有的ASCII字符）都使用一个字节表示。

空字符null（‘\u0000’）和从‘\u0080’到‘\u07ff’的所有字符使用两个字节表示。



从‘\u0800’到‘\uffff’的所有字符使用三个字节表示。



在CONSTANT_Utf8_info表内，bytes项中的UTF-8字符串编码与标准UTF-8格式的区别在于：第一，在标准UTF-8编码模式中，空字符null使用一个字节表示；在CONSTANT_Utf8_info表中，空字符使用两个字节表示。这种对于空字符null的双字节编码，意味着bytes项的值永远不会为0。第二，bytes项中只使用了标准UTF-8编码中的单字节、双字节和三字节编码，而标准UTF-8编码还包括未在CONSTANT_Utf8_info表中使用的较长的格式。

6.4.2 CONSTANT_Integer_info表

固定长度的CONSTANT_Integer_info表用来存储常量int类型值。该表只存储int值，不存储符号引用。表6-10列出了CONSTANT_Integer_info表的格式。

表6-10 CONSTANT_Integer_info表的格式

类 型	名 称	数 量
u1	tag	1
u4	bytes	1

CONSTANT_Integer_info表中各项如下：

tag

tag项的值为CONSTANT_Integer（3）。

bytes

bytes项中按照高位在前的格式存储int类型值。

6.4.3 CONSTANT_Float_info表

固定长度的CONSTANT_Float_info表用来存储常量float类型值。该表只存储float类型值，不存储符号引用。表6-11列出了CONSTANT_Float_info表的格式。

表6-11 CONSTANT_Float_info表的格式

类 型	名 称	数 量
u1	tag	1
u4	bytes	1

CONSTANT_Float_info表中各项如下：

tag

tag项的值为CONSTANT_Float（4）。

bytes

bytes项中按照高位在前的格式存储float类型值。有关Java class文件中float类型的详细内容请参阅第14章。

6.4.4 CONSTANT_Long_info表

固定长度的CONSTANT_Long_info表用于存储long类型常量。该表只存储long类型值，不存储符号引用。表6-12列出了CONSTANT_Long_info表的格式。

表6-12 CONSTANT_Long_info表的格式

类 型	名 称	数 量
u1	tag	1
u8	bytes	1

如前所述，一个long类型值在常量池中占据常量池中的两个位置。在class文件中，一个long类型入口紧接着下一个入口，但下一个入口的索引值却比紧挨着的上一个入口的值多2。

CONSTANT_Long_info表中各项如下：

tag

tag项的值为CONSTANT_Long（5）。

bytes

bytes项中按照高位在前的格式存储long类型值。

6.4.5 CONSTANT_Double_info表

固定长度的CONSTANT_Double_info表用来存储double类型常量。该表只用来存储double值，不存储符号引用。表6-13列出了CONSTANT_Double_info表的格式。

表6-13 CONSTANT_Double_info表的格式

类 型	名 称	数 量
u1	tag	1
u8	bytes	1

如前所述，一个double类型值在常量池中占据常量池中的两个位置。在class文件中，double入口的下一入口紧随其后，但下一入口的索引值却要比上一个入口的索引值大2。

CONSTANT_ Double_info表中各项如下：

tag

tag项的值为CONSTANT_Double（6）。

bytes

bytes项中按照高位在前的格式存储double类型值。有关Java class文件中double类型的详细内容请参阅第14章。

6.4.6 CONSTANT_Class_info表

固定长度的CONSTANT_Class_info表使用符号引用来表述类或者接口。无论指向类、接口、字段，还是方法，所有的符号引用都包含一个CONSTANT_Class_info表。表6-14列出了CONSTANT_Class_info表的格式。

表6-14 CONSTANT_Class_info表的格式

类 型	名 称	数 量
u1	tag	1
u2	name_index	1

CONSTANT_Class_info表中各项如下：

tag

tag项的值为CONSTANT_Class（7）。

name_index

name_index项给出了包含类或者接口全限定名的CONSTANT_Utf8_info表的索引。

由于Java中的数组是完善的对象，CONSTANT_Class_info表也能够用来描述数组类。CONSTANT_Class_info表中的name_index项指向CONSTANT_Utf8_info表，该表中包含了数组的描述符，描述符可作为数组类的名称。例如，一个double[][]数组类型的类名为它的描述符[[D；net.jini.core.lookup.ServiceItem[][][]数组类型的类名为它的描述符[[[Lnet/jini/core/lookup/ServiceItem;。由于Java数组最多只能有255维，数组描述符最多只能有255个引导符“[”。

6.4.7 CONSTANT_String_info表

固定长度的CONSTANT_String_info表用来存储文字字符串值，该值亦可表示为类java.lang.String的实例。该表只存储文字字符串值，不存储符号引用。表6-15列出了CONSTANT_String_info表的格式。

表6-15 CONSTANT_String_info表的格式

类 型	名 称	数 量
u1	tag	1
u2	string_index	1

CONSTANT_String_info表中各项如下：

tag

tag项的值为CONSTANT_String（8）。

string_index

string_index项给出了包含文字字符串值的CONSTANT_Utf8_info表的索引。

6.4.8 CONSTANT_Fieldref_info表

固定长度的CONSTANT_Fieldref_info表描述了指向字段的符号引用。表6-16列出了CONSTANT_Fieldref_info表的格式。

表6-16 CONSTANT_Fieldref_info表的格式

类 型	名 称	数 量
u1	tag	1
u2	class_index	1
u2	name_and_type_index	1

CONSTANT_Fieldref_info表中各项如下：

tag

tag项的值为CONSTANT_Fieldref（9）。

class_index

class_index项给出了声明被引用字段的类或者接口的CONSTANT_Class_info入口的索引。

需要注意的是，由class_index指定的CONSTANT_Class_info不只是代表类，还可能代表接口。尽管接口中能够声明字段，而且可以分别声明为公开、静态和final类型。但如前所述，如果其他类的静态final字段使用编译时的常量进行初始化操作，那么class文件不包含对这些字段的符号引用。但是，class文件可以包含它使用的任何这些静态final字段的常量值的复本。例如，如果类使用在接口中声明的float类型的静态 final字段，而且它被初始化为编译时的常量，该类将会在它自己的存储float值的常量池中拥有一个CONSTANT_Float_info表。但是如果该接口使用只有在运行时才能计算出的表达式来初始化它的静态final字段，那么在使用该字段的类的常量池中，将会有一个对该接口中的字段进行符号引用的CONSTANT_Fieldref_info表。有关对静态final字段特殊处理的详细内容请参阅第8章。

name_and_type_index

name_and_type_index提供了CONSTANT_NameAndType_info入口的索引，该入口提供了字段的简单名称以及描述符。

6.4.9 CONSTANT_Methodref_info表

固定长度的CONSTANT_Methodref_info表使用符号引用来表述类中声明的方法（不包括接口中的方法）。表6-17列出了CONSTANT_Methodref_info表的格式。

表6-17 CONSTANT_Methodref_info表的格式

类 型	名 称	数 量
u1	tag	1
u2	class_index	1
u2	name_and_type_index	1

CONSTANT_Methodref_info表中各项如下：

tag

tag项的值为CONSTANT_Methodref（10）。

class_index

class_index项给出了声明了被引用方法的类的CONSTANT_Class_info入口的索引。由class_index所指定的CONSTANT_Class_info表必须为类，不能为接口。指向接口中声明的方法的符号引用使用CONSTANT_InterfaceMethodref表。

name_and_type_index

name_and_type_index提供了CONSTANT_NameAndType_info入口的索引，该入口提供了方法的简单名称以及描述符。如果方法的简单名称开始于“<”（'\u003c'）符号，该方法必须为一个实例初始化方法。它的简单名称必须为<init>，它的返回值必须为void类型。否则，该方法

的名称必须为一个有效的Java程序设计语言的标识符。

6.4.10 CONSTANT_InterfaceMethodref_info表

固定长度的CONSTANT_InterfaceMethodref_info表使用符号引用来描述接口中声明的方法（不包括类中的方法）。表6-18列出了CONSTANT_InterfaceMethodref_info表的格式。

表6-18 CONSTANT_InterfaceMethodref_info表的格式

类 型	名 称	数 量
u1	tag	1
u2	class_index	1
u2	name_and_type_index	1

CONSTANT_InterfaceMethodref_info表中各项如下：

tag

tag项的值为CONSTANT_InterfaceMethodref（11）。

class_index

class_index项给出了声明了被引用方法的接口的CONSTANT_Class_info入口的索引。由class_index所指定的CONSTANT_Class_info表必须为接口，不能为类。在类中声明的方法的符号引用使用CONSTANT_Methodref表。

name_and_type_index

name_and_type_index提供了CONSTANT_NameAndType_info入口的索引，该入口提供了方法的简单名称以及描述符。

6.4.11 CONSTANT_NameAndType_info表

固定长度的CONSTANT_NameAndType_info表构成指向字段或者方法的符号引用的一部分。该表提供了所引用字段或者方法的简单名称和描述符的常量池入口。表6-19列出了CONSTANT_NameAndType_info表的格式。

表6-19 CONSTANT_NameAndType_info表的格式

类 型	名 称	数 量
u1	tag	1
u2	name_index	1
u2	descriptor_index	1

CONSTANT_NameAndType_info表中各项如下：

tag

tag项的值为CONSTANT_NameAndType（12）。

name_index

name_index项给出了CONSTANT_Utf8_info入口的索引，该入口给出了字段或者方法的名称。该名称或者为一个有效的Java程序设计语言的标识符，或者为<init>。

descriptor_index

descriptor_index提供了CONSTANT_Utf8_info入口的索引，该入口提供了字段或者方法的描述符。该描述符必须为一个有效的字段或者方法的描述符。

6.5 字段

在类或者接口中声明的每一个字段（类变量或者实例变量）都由class文件中的一个名为field_info的可变长度的表进行描述。在一个class文件中，不会存在两个具有相同名字和描述的字段。（需要注意的是，尽管在Java程序设计语言中不会有两个具有相同名字的字段存在于同一个类或者接口中，但一个class文件中的两个字段可以拥有同一个名字——只要它们的描述符不同。换句话说，尽管在Java程序设计语言中无法在同一个类或者接口中定义两个具有同样名字和不同类别的字段，但是两个这样的字段却可以同时合法地出现在一个Java class文件中。）表6-20中列出了field_info表的格式。

表6-20 field_info表的格式

类 型	名 称	数 量
u2	access_flags	1
u2	name_index	1
u2	descriptor_index	1
u2	attributes_count	1
attribute_info	attributes	attributes_count

field_info表中各项如下所示：

access_flags

声明字段时使用的修饰符存放在字段的access_flags项中。表6-21列出了各个标志所使用的位。

表6-21 field_info表中access_flags项的标志

标志名称	值	设定含义	设定者
ACC_PUBLIC	0x0001	字段设为public	类和接口
ACC_PRIVATE	0x0002	字段设为private	只有类
ACC_PROTECTED	0x0004	字段设为protected	只有类
ACC_STATIC	0x0008	字段设为static	类和接口
ACC_FINAL	0x0010	字段设为final	类和接口
ACC_VOLATILE	0x0040	字段设为volatile	只有类
ACC_TRANSIENT	0x0080	字段设为transient	只有类

类（不包括接口）中声明的字段，只能拥有ACC_PUBLIC、ACC_PRIVATE、ACC_PROTECTED这三个标志中的一个。ACC_FINAL和ACC_VOLATILE不能同时设置。所有接口中声明的字段必须有且只能有ACC_PUBLIC、ACC_STATIC和ACC_FINAL这三种标志。

access_flags中没有用到的位都被设为0，Java虚拟机实现将忽略它们。

name_index

name_index项提供了给出字段简单名称（不是全限定名）的CONSTANT_Utf8_info入口的索

引。在class文件中的每一个字段的名称都必须符合Java程序设计语言中对名称的有效规定。

descriptor_index

descriptor_index提供了给出字段描述符的CONSTANT_Utf8_info入口的索引。

attributes_count和attributes

attributes项是由多个attribute_info表组成的列表。attributes_count指出列表中attribute_info表的数量。一个字段在其列表中可以有任意数量的属性。由Java虚拟机规范定义的三种可能会出现在此项中的属性是：ConstantValue、Deprecated和Synthetic。这三种属性将在本章后面进一步阐述。Java虚拟机惟一需要识别的属性是ConstantValue属性。虚拟机实现必须忽略任何无法识别的属性。

6.6 方法

在class文件中，每个在类和接口中声明的方法，或者由编译器产生的方法，都由一个可变长度的method_info表来描述。同一个类中不能存在两个名字及描述符完全相同的方法。需要注意的是，在Java程序设计语言中，尽管在同一个类或者接口中声明的两个方法不能有同样的特征签名（除返回类型之外的描述符），但在同一个class文件中，两个方法可以拥有同样的特征签名，前提是它们的返回值不能相同。换句话说：在Java源文件的同一个类里，如果声明了两个具有相同名字和相同参数类型、但返回值不同的方法，这个程序将无法编译通过。在Java程序设计语言中，不能仅仅通过返回值的不同来重载方法。但是这样的两个方法可以和谐地在一个class文件中共存。

有可能在class文件中出现的两种编译器产生的方法是：实例初始化方法（名为<init>）和类与接口初始化方法（名为<clinit>）。需要了解更多有关编译器产生的方法，请参照第7章。method_info表的格式如表6-22所示。

表6-22 method_info表的格式

类　型	名　称	数　量
u2	access_flags	1
u2	name_index	1
u2	descriptor_index	1
u2	attributes_count	1
attribute_info	attributes	attributes_count

method_info表中各项如下：

access_flags

在声明方法时使用的修饰符存放在方法的access_flags项中。表6-23列出了各个标志所使用的位。1.2版本中加进了ACC_STRICT标志，它指明方法中的所有表达式都必须使用FP-strict模式进行计算。第14章详细阐述了FP-strict模式。

类（不包括接口）中声明的方法只能拥有ACC_PUBLIC、ACC_PRIVATE、ACC_PROTECTED这三个标志中的一个。如果设定了一个方法的ACC_ABSTRACT标志，那么

它的ACC_PRIVATE、ACC_STATIC、ACC_FINAL、ACC_SYNCHRONIZED、ACC_NATIVE以及ACC_STRICT标志都必须清除。接口中声明的所有方法必须有ACC_PUBLIC和ACC_ABSTRACT标志，除此以外接口方法不能使用其他标志，但接口初始化方法（<clinit>）可以使用ACC_STRICT标志。

表6-23 method_info表中access_flags项的标志

标志名称	值	设定含义	设定者
ACC_PUBLIC	0x0001	方法设为public	类和所有的接口方法
ACC_PRIVATE	0x0002	方法设为private	只有类
ACC_PROTECTED	0x0004	方法设为protected	只有类
ACC_STATIC	0x0008	方法设为static	只有类
ACC_FINAL	0x0010	方法设为final	只有类
ACC_SYNCHRONIZED	0x0020	方法设为synchronized	只有类
ACC_NATIVE	0x0100	方法设为native	只有类
ACC_ABSTRACT	0x0400	方法设为abstract	类和所有的接口方法
ACC_STRICT	0x0800	方法设为strictFP	类和接口的<clinit>方法

实例初始化方法（<init>）可以只使用ACC_PUBLIC、ACC_PRIVATE和ACC_PROTECTED标志。因为类与接口初始化方法（<clinit>）只由Java虚拟机直接调用，永远不会被Java字节码直接调用，这样，<clinit>方法的access_flags中的标志位，除去ACC_STRICT之外的所有位都应该被忽略。

在access_flags中未用到的位都被设为0，Java虚拟机实现也将忽略它们。

name_index

name_index项提供了CONSTANT_Utf8_info入口的索引，该入口给出了方法的简单名称（不是全限定名）。在class文件中的每一个方法的名称，都必须或者为<init>，或者为<clinit>，或者是Java程序设计语言中有效的方法名称（简单名称，不是全限定名）。

descriptor_index

descriptor_index提供了CONSTANT_Utf8_info入口的索引，该入口给出了方法的描述符。

attributes_count和attributes

attributes项是由多个attribute_info表组成的列表。attributes_count给出列表中attribute_info表的数量。一个字段在其列表中可以有任意数量的属性。在此项中可能会出现的由Java虚拟机规范定义的四种属性是：Code、Deprecated、Exceptions和Synthetic。这四种属性将在本章后面进一步阐述。Java虚拟机只需要识别Code和Exception属性。虚拟机实现必须忽略任何无法识别的属性。

6.7 属性

如前所述，属性在Java class文件中多处出现。它们可以出现在ClassFile、field_info、method_info和Code_attribute表中。Code_attribute表本身即为一个属性，本节将对它进行阐述。

如表6-24所示，Java虚拟机规范定义了9种属性。为了正确地解释Java class文件，所有Java虚拟机实现都必须能够识别下列三种属性：Code，ConstantValue和Exception。为了正确地实现

Java和Java 2平台类库，虚拟机实现必须能够识别InnerClasses和Synthetic属性，但可以自主选择究竟是识别还是忽略其他一些预定义的属性（在Java 1.1中，加入了Deprecated、InnerClasses和Synthetic属性（attribute））。所有这些预定义的属性将在本章详细阐述。

表6-24 由规范定义的attribute_info表的类型

名 称	使 用 者	描 述
Code	method_info	方法的字节码和其他数据
ConstantValue	field_info	final变量的值
Deprecated	field_info、method_info	字段或者方法被禁用的指示符
Exceptions	method_info	方法可能抛出的可被检测的异常
InnerClasses	ClassFile	内部、外部类的列表
LineNumberTable	Code_attribute	方法的行号与字节码的映射
LocalVariableTable	Code_attribute	方法的局部变量的描述
SourceFile	ClassFile	源文件名
Synthetic	field_info、method_info	编译器产生的字段或者方法的指示符

表6-25 attribute_info表的格式

类型	名 称	数 量
u2	attribute_name_index	1
u4	attribute_length	1
u1	info	attribute_length

如果需要向Java class文件中加入新的属性，除Sun公司以外的任何人都必须遵循下列两个步骤：

1）任何不是由规范进行预定义的属性都不能影响类或者接口类型的语义。新的属性只能向class文件添加新的信息，如在调试过程中用到的信息等。

2）属性必须使用与Internet域名方案颠倒的命名方式，Internet域名方案是针对Java语言规范中包的命名所定义的。例如，如果拥有Internet域名artima.com，而需要创建的新属性为CompilerVersion，那么属性则应该命名为com.artima.CompilerVersion。

6.7.1 属性格式

如表6-25所示每一种属性都遵循同样的可变长度attribute_info表的一般格式。attribute_name_index属性的前两个字节构成了到CONSTANT_Utf8_info表的常量池的索引，CONSTANT_Uft8-info表中包含了属性的字符串名称。因此，每一个attribute_info表，由其表中的第1项指出了它的“类型”，这种方式就像cp_info表中由初始tag字节指出它们的类型一样。不同的是，cp_info表的类型由一个无符号字节值指定，如3（CONSTANT_Integer_info），而attribute_info表的类型由一个字符串指定。

紧随attribute_name_index其后的是4字节长的attribute_length项，它给出除去起始6个字节后整个attribute_info表的长度（attribute_length项可以为0）。因为只要遵循一定规则（如下所列），任何人都能够向Java class文件中加入属性，所以长度是不可缺少的。Java虚拟机实现能够识别

新属性，实现必须忽略任何无法识别的属性。当解析class文件时，attribute_length允许虚拟机跳过无法识别的属性。

attribute_info表中各项如下所示：

attribute_name_index

attribute_name_index项给出了包含属性名称的CONSTANT_Utf8入口的常量池中的索引。

attribute_length

attribute_length项给出了属性数据的长度（以字节计），但是包括attribute_name_index和attribute_length在内的起始6个字节不包括在内。

info

info项包含属性数据。

6.7.2 Code属性

可变长度的Code_attribute表定义了方法的字节码序列和其他信息。在所有不是抽象或者本地方法的method_info信息中，都存在一个Code_attribute表。Code_attribute表的格式如表6-26所示。

表6-26 Code_attribute表的格式

类 型	名 称	数 量
u2	attribute_name_index	1
u4	attribute_length	1
u2	max_stack	1
u2	max_locals	1
u4	code_length	1
u1	code	code_length
u2	exception_table_length	1
exception_info	exception_table	exception_table_length
u2	attributes_count	1
attribute_info	attributes	attributes_count

Code_attribute表中各项如下所示：

attribute_name_index

attribute_name_index项给出包含字符串“Code”的CONSTANT_Utf8_info入口的常量池索引。

attribute_length

attribute_length项给出除去起始6个字节（包含attribute_name_index和attribute_length项）后，Code属性以字节为单位的长度。

max_stack

在方法执行中的任意时刻，max_stack项给出该方法的操作数栈的最大长度（以字为单位）。

max_locals

max_locals项给出方法的局部变量所需存储空间的长度（以字为单位）。无论虚拟机什么时候调用被Code属性所描述的方法，它都必须分配一个长度为max_locals的局部变量数组。这个数组用来存储传递给方法的参数以及为方法所使用的局部变量。long或者double类型值的最大有效的局部变量索引是max_locals - 2。任何其他类型值的最大有效局部变量索引为max_locals - 1。

code_length和code

code_length项给出该方法字节码流的长度（按字节计）。字节码本身将会出现在code项中。code_length的值必须大于0。

exception_table_length和exception_table

exception_table项是一个exception_info表的列表。每个exception_info表都描述了一个异常表项。exception_table_length项给出了exception_table中exception_info表的数目。exception_table表在列表中按照方法执行抛出异常时Java虚拟机检查匹配异常处理器（catch子句）的顺序进行排列。表exccption_table的格式如表6-27所述，并在下面“exception_info表”部分进行阐述。如果需要了解更多异常表的相关信息，请参照第17章。

表6-27 exception_info表的格式

类 型	名 称	数 量
u2	start_pc	1
u2	end_pc	1
u2	handler_pc	1
u2	catch_type	1

attributes_count和attributes

attributes项是一个attribute_info表的列表。attributes_count项给出了列表中attribute_info表的数目。该项中可以出现Java虚拟机规范所定义的两种属性：LineNumberTable和LocalVariableTable。这两种属性将在本章后面详细描述。Java虚拟机实现允许忽略Code属性内attributes项中的任何属性，如果Java虚拟机无法识别这些属性，Java虚拟机必须忽略它们。

exception_info表 固定长度的exception_info表描述了一个异常表项。该表在Code属性中的exception_info项中出现，它是exception_info表序列的组成部分。exception_info表的格式如表6-27所示。如果需要获取更多有关异常表的信息，请参照第17章。

exception_info表中各项如下所示：

start_pc

start_pc项给出从代码数组起始处到异常处理器起始处的偏移量。

end_pc

end_pc项给出从代码数组的起始处到异常处理器结束后一个字节的偏移量。

handler_pc

handler_pc项给出一条指令从代码数组起始处跳转到异常处理器的第1条指令的偏移量——如果抛出的异常被该项捕获的话。

catch_type

catch_type项给出被该异常处理器所捕获的异常类型的CONSTANT_Class_info入口的常量池索引。CONSTANT_Class_info入口必须描述了java.lang.Throwable类或其子类。

如果catch_type的值为0（不是一个有效的常量池索引，因为常量池从索引1开始），那么异常处理器将处理所有的异常。一个值为0的catch_type用于实现finally子句。如果需要了解finally

子句的实现，请参照第18章。

6.7.3 ConstantValue属性

固定长度的ConstantValue属性出现在值为常量的字段的field_info表中。给定field_info表的属性项中最多只可能出现一个ConstantValue属性。在包含ConstantValue属性的field_info表内的access_flag中必须设定ACC_STATIC标志。尽管可能并不需要，但也可以设定ACC_FINAL标志。当虚拟机初始化一个具有ConstantValue属性的字段时，它将一个常量值赋给这个字段。赋值操作在虚拟机调用声明此字段的类或者接口的初始化方法之前进行。ConstantValue_attribute表的格式如表6-28所示。

表6-28 ConstantValue_attribute表的格式

类 型	名 称	数 量
u2	attribute_name_index	1
u4	attribute_length	1
u2	constantvalue_index	1

ConstantValue_attribute表中各项如下所示：

attribute_name_index

attribute_name_index项给出包含字符串“ConstantValue”的CONSTANT_Utf8_info入口的常量池索引。

attribute_length

ConstantValue_attribute中的attribute_length项的值永远为2。

constantvalue_index

constantvalue_index项给出提供常量值的入口的常量池索引。表6-29给出了对应于每一种字段类型的入口的类型。

表6-29 常量值属性的常量池入口类型

常量值类型	常量池入口类型
byte，short，char，int，boolean	CONSTANT_Integer_info
long	CONSTANT_Long_info
float	CONSTANT_Float_info
double	CONSTANT_Double_info
java.lang.String	CONSTANT_String_info

6.7.4 Deprecated属性

固定长度的Deprecated属性存在于field_info、method_info和ClassFile表内的attributes项中，它是一个可选的项，它指出了所禁用的字段、方法或者类型（这里“禁用”的意思是：尽管一个字段、方法或者类型仍然存在于执行的方法中，但程序员永远再不会用到它。更确切地说，程序员使用其他相近的字段、方法和类型，而不会使用所禁用的项）。编译器、虚拟机或者用来读取class文件的任何工具都可以使用Deprecated属性来通知程序员，程序使用了禁用的字段、方

法或者类型。Deprecated属性在Java 1.1版本中加入，用来支持javadoc工具使用的文档注释中的@deprecated标志。Deprecated_attribute表的格式如表6-30所示。

表6-30 Deprecated_attribute表的格式

类 型	名 称	数 量
u2	attribute_name_index	1
u4	attribute_length	1

Deprecated_attribute表中各项如下所示：

attribute_name_index

attribute_name_index给出包含字符串“Deprecated”的CONSTANT_Utf8_info入口的常量池索引。

attribute_length

attribute_length必须为0。

6.7.5 Exceptions属性

可变长度的Exceptions属性列出了方法可能抛出的异常。Exceptions_attribute表会出现在每一个可能抛出已检出异常的方法的method_info表中。Exceptions_attribute表的格式如表6-31所示。

表6-31 Exceptions_attribute表的格式

类 型	名 称	数 量
u2	attribute_name_index	1
u4	attribute_length	1
u2	number_of_exceptions	1
u2	exception_index_table	number_of_exceptions

如果一个方法是RuntimeException、Error或者是在方法的Exceptions属性中所列出的异常的实例或子类，那么它应该只抛出一个异常。尽管这条规则应该被Java编译器强制执行，但它没有被Java虚拟机强制执行。因此，为了Java编译器，Exceptions属性存在于Java class文件中。

Exceptions_attribute表中各项如下所示：

attribute_name_index

attribute_name_index给出包含字符串“Exceptions”的CONSTANT_Utf8_info入口的常量池索引。

attribute_length

attribute_length给出了除去起始6个字节（其中包含attribute_name_index和attribute_length项）后，Exceptions_attribute的长度（按字节计算）。

number_of_exceptions和exception_index_table

exception_index_table是一个该方法内throws子句中所声明异常的常量池中CONSTANT_Class_info入口的索引的数组。换句话说，exception_index_table列出了该方法可能抛出的所有已检出的异常。number_of_exceptions项指出了数组中索引的数目。

6.7.6 InnerClasses 属性

可变长度的InnerClasses属性对名字、访问标志以及被声明为成员的任何嵌入类型的外围类型，或者用别的方法由类或者接口陈述的类型（嵌入类型不是包中的成员，而是类或者接口的成员）。如果类或者接口的代码指向一个嵌入类型，该类或者接口的常量池将会包含一个用于此嵌入类型的CONSTANT_Class_info入口。常量池还必须为每一个嵌入类型（如果存在的话）包含一个CONSTANT_Class_info入口，这些嵌入类型被定义为类或者数组的直接成员——尽管类或者接口并没有另外描述内嵌类型。如果类或者接口的常量池包含任何内嵌类型的CONSTANT_Class_info入口，那么此类或者接口的class文件必须包含一个InnerClasses_attribute表，此表存在于它自身的ClassFile表的attributes项中。InnerClasses_attribute表的格式如表6-32所示。

表6-32 InnerClasses_attribute表的格式

类 型	名 称	数 量
u2	attribute_name_index	1
u4	attribute_length	1
u2	number_of_classes	1
inner_classes_info	classes	number_of_classes

Java虚拟机没有立即验证由InnerClasses_attribute表所陈述的class文件表示类型是否与InnerClasses_attribute相一致。

InnerClasses _attribute表中各项如下所示：

attribute_name_index

attribute_name_index给出包含字符串“InnerClasses”的CONSTANT_Utf8_info入口的常量池索引。

attribute_length

attribute_length给出除去起始6个字节（其中包含attribute_name_index和attribute_length项）后，InnerClasses_attribute的长度（按字节计算）。

number_of_classes和classes

classes项是一个成员为inner_class_info表的数组。number_of_classes项给出了class数组中inner_class_info表的数目。inner_class_info表的格式如表6-33所示。

表6-33 inner_class_info表的格式

类 型	名 称	数 量
u2	inner_class_info_index	1
u2	outer_class_info_index	1
u2	inner_name_index	1
u2	inner_class_access_flags	1

InnerClass属性的classes项将会为每一个在常量池中CONSTANT_Class_info入口陈述的内嵌类包含一个inner_class_info表。因为每个作为其外围类型的直接成员声明的内嵌类型的

CONSTANT_Class_info入口都必须出现在该外围类型的常量池中，外围类型的InnerClasses属性的classes项将会为每一个作为外围类型的直接成员声明的内嵌类型明确包含一个inner_class_info表。

例如，如果类Rain、类Snow和接口Wet都声明为类Weather的成员，Weather类的InnerClasses属性将会明确地包含一个Rain、Snow和Wet的inner_class_info表。同样，如果类Thunder被声明为类Rain的成员（Rain类在Weather中声明），那么类Rain的InnerClasses属性将会为类Thunder明确包含inner_class_info表。Thunder的inner_class_info表也可能出现于类Weather的InnerClass属性中，但这并不是必须的。因为类Thunder并没有声明为Weather的成员，只有当类Weather的代码明确引用Thunder类时，Thunder才会在Weather类的InnerClasses属性中出现。

除了陈述所有作为成员声明的内嵌类型之外，InnerClasses属性还将陈述所有内嵌类型的外围类。在ClassFile表内this_class项所指向的CONSTANT_Class_info入口中，所有的类型通常都在它们自己的常量池中陈述自身。因此，如果class文件定义的类型是内嵌类型（不是包的成员，而是其他类或者接口的成员），被定义的类型将会出现在它自身的InnerClasses属性中。因为给定内嵌类型的inner_class_info表的outer_class_info_index项指向该内嵌类型的外围类型，定义了内嵌类型的class文件中的InnerClasses属性将会为它所有的外围类型包含一个inner_class_info表。

例如，如果类Thunder被声明为类Rain的成员，类Rain被声明为类Weather的成员，类Weather是一个包的成员，那么类Thunder的InnerClasses属性将会为它的所有外围类型（Rain和Weather）明确地包含一个inner_class_info表。同样，类Rain的InnerClasses属性将会为它的外围类型Weather明确包含一个inner_class_info表。

inner_class_info表　固定长度的inner_class_info表位于InnerClasses属性的classes项中，它提供类型的相关信息，该类型或者本身为内嵌类型，或者至少有一个其他类型将声明为其成员（换句话说，每一个inner_class_info表描述了一种类型，该类型或者为内嵌类型，或者为外围类型，或者两者都是）。inner_class_info表的格式如表6-33所示。

inner_class_info表中各项如下所示：

inner_class_info_index

inner_class_info_index给出了指向常量池中CONSTANT_Class_info入口的索引，该CONSTANT_Class_info入口说明了被inner_class_info表所描述的内嵌类。

outer_class_info_index

outer_class_info_index给出了指向常量池中CONSTANT_Class_info入口的索引（指被inner_class_info表所描述的内嵌类型中的类型，它被声明为成员）。如果inner_class_info表没有对内嵌类型进行描述，那么outer_class_info_index的值必须为0。inner_class_info表可以描述非内嵌类型（换句话说，声明为包成员的类型），因为外围类型也在InnerClasses属性中陈述过。任何内嵌类型的最远外围类型总是包中的成员。

例如，如果将类Rain声明为类Weather的成员，将类Weather声明为包成员，类Rain的InnerClass属性将会包含一个关于Weather的inner_class_info表。因为类Weather声明为包的成员，它的outer_class_info_index将为0。

inner_name_index

如果inner_class_info表描述的不是一个匿名inner类，那么inner_name_index就会给出常量池

中指向一个CONSTANT_Utf8_info入口的索引，该CONSTANT_Uft8_info入口给出了被此inner_class_info表所描述的类型的简单名称。如果inner_class_info表描述的是一个匿名inner类，那么inner_name_index的值为0。

需要注意的是，被inner_class_info表所描述的任何类型，都可以通过一个两步查询过程获取类型名称：首先，通过inner_class_info_index获取该类型的CONSTANT_Class_info入口；然后，从CONSTANT_Class_info入口的name_index项中获取一个给出该类型简单名称的CONSTANT_Utf8_info入口。对于匿名inner类，通过这个两步查询过程可以得到由编译器给出的名称。对于所有由inner_class_info表描述的（非匿名）类型，通过这个两步查询过程将会得到被inner_name_index所指向的同样的名称。因此，获取由inner_class_info表所描述的类型名称时，并不一定需要inner_name_index。更恰当地说，inner_name_index用来区分源代码中匿名inner类（名称由编译器产生）和非匿名类型（其名称由程序员在源代码中规定）。

inner_class_access_flags

inner_class_access_flags项给出了对inner类的访问标志。当源代码无法使用的时候，编译器使用这些标志来恢复关于内嵌类声明的信息。在此项中用到的标志如表6-34所示。inner_class_access_flags中所有未用到的位都被编译器置为0，并且被Java虚拟机实现所忽略。

表6-34 inner_class_info表内inner_class_access_flags项中的标志位

标志名称	值	设定时的含义
ACC_PUBLIC	0x0001	源代码中显式或者隐式指定的public 限定
ACC_PRIVATE	0x0002	源代码中显式指定的private限定
ACC_PROTECTED	0x0004	源代码中显式指定的protected限定
ACC_STATIC	0x0008	源代码中显式或者隐式指定的static限定
ACC_FINAL	0x0010	源代码中显式指定的final限定
ACC_INTERFACE	0x0200	源代码中的接口
ACC_ABSTRACT	0x0400	源代码中显式或者隐式指定的abstract限定

6.7.7 LineNumberTable属性

可变长度的LineNumberTable属性建立了方法字节码流偏移量和源代码行号之间的映射关系。LineNumberTable_attribute表可能会出现（可选）在Code_attribute表中的属性部分中。LineNumberTable_attribute表的格式如表6-35所示。

表6-35 LineNumberTable_attribute表的格式

类型	名称	数量
u2	attribute_name_index	1
u4	attribute_length	1
u2	line_number_table_length	1
line_number_info	line_number_table	line_number_table_length

LineNumberTable _attribute表中各项如下所示：

attribute_name_index

attribute_name_index给出包含字符串“Line_Number_Table”的CONSTANT_Utf8_info入口的常量池索引。

attribute_length

attribute_length给出除去起始6个字节（其中包含attribute_name_index和attribute_length项）后，LineNumberTable_attribute的长度（按字节计算）。

line_number_table_length和line_number_table

line_number_table项是一个成员为line_number_info表的数组。line_number_table_length给出line_number_table数组中line_number_info表的数目。该数组中的表可以按照任何顺序排列，该数组中也可能会有多个表对应一个行号的情况发生。line_number_info表的格式如表6-36所述。

line_number_info表　固定长度的line_number_info表位于LineNumberTable_attribute表内的line_number_table项中，它建立了源代码行号与字节码数组中的指令之间的联系，这里的字节码数组对应于该行源代码的编译形式的开始位置。line_number_info表的格式如表6-36所示。

表6-36　line_number_info表的格式

类　型	名　称	数　量
u2	start_pc	1
u2	line_number	1

line_number_info表中各项如下所示：

start_pc

start_pc给出了新行开始时代码数组的偏移量，该偏移量从代码数组的起始位置开始计算。start_pc的值必须小于该LineNumberTable属性所属的Code属性中code_length项的值。

line_number

line_number项给出从start_pc开始的行号。

6.7.8　LocalVariableTable属性

可变长度的LocalVariable属性建立了方法的栈帧中局部变量部分内容与源代码中局部变量的名称和描述符之间的映射关系。LocalVariableTable_attribute表可以（但不是一定）存在于Code_attribute表的属性部分中。LocalVariableTable_attribute表的格式如表6-37所示。

表6-37　LocalVariableTable_attribute表的格式

类　型	名　称	数　量
u2	attribute_name_index	1
u4	attribute_length	1
u2	local_variable_table_length	1
local_variable_info	local_variable_table	local_variable_table_length

LocalVariableTable _attribute表中各项如下所示：

attribute_name_index

attribute_name_index给出包含字符串“LocalVariableTable”的CONSTANT_Utf8_info入口

的常量池索引。

attribute_length

attribute_length给出除去起始6个字节（其中包含attribute_name_index和attribute_length项）后，LocalVariableTable_attribute的长度（按字节计算）。

local_variable_table_length和local_variable_table

local_variable_table项是一个成员为local_variable_info表的数组。local_variable_table_length给出了local_variable_table数组中local_variable_info表的数目。local_variable_info表的格式如表6-38所示。

local_variable_info表　固定长度的表local_variable_info位于LocalVariableTable_attribute表内的local_variable_table项中，它建立起源代码中局部变量的名称、类型与局部变量在字节码数组中的作用域、栈帧内局部变量中的索引之间的联系。local_variable_info表的格式如表6-38所示。

表6-38　local_variable_info表的格式

类　型	名　称	数　量
u2	start_pc	1
u2	length	1
u2	name_index	1
u2	descriptor_index	1
u2	index	1

local_variable_info表中各项如下所示：

start_pc和length

start_pc项给出代码数组中指令开始位置的偏移量。length项给出从start_pc开始的、所有局部变量有效的、代码的长度。位于从代码数组开始偏移量start_pc + length位置处的字节只能有下列两种情况：或者为一个指令的首字节、或者为代码数组结束后的首字节。

name_index

name_index项给出包含局部变量名称的CONSTANT_Utf8_info入口的常量池索引。

descriptor_index

descriptor_index提供了CONSTANT_Utf8_info入口的索引，该入口提供了局部变量的描述符（局部变量描述符符合与字段描述符相一致的语法）。

index

index项给出了在此方法栈帧中局部变量部分的索引，这是当方法执行时该局部变量数据所保存的位置。如果局部变量的数据类型为long或者double，那么数据占据index和index + 1位置的两个字节；其他类型的变量数据占据index位置的一个字节。

6.7.9 SourceFile属性

固定长度的SourceFile属性可能存在于ClassFile表内的属性项中，它是一个可选的项，它提供了产生class文件的源文件的名称。在ClassFile表内的attributes表中只能有一个SourceFile_attribute表存在。SourceFile _attribute表的格式如表6-39所示。

表6-39 SourceFile_attribute表的格式

类 型	名 称	数 量
u2	attribute_name_index	1
u4	attribute_length	1
u2	sourcefile_index	1

SourceFile _attribute表中各项如下所示：

attribute_name_index

attribute_name_index给出包含字符串“SourceFile”的CONSTANT_Utf8_info入口的常量池索引。

attribute_length

SourceFile _attribute表中attribute_length项的值永远为2。

sourcefile_index

Sourcefile_index项给出了包含源文件名称的CONSTANT_Utf8_info入口的常量池索引。源文件名永远不会包括目录路径。

6.7.10 Synthetic属性

固定长度的Synthetic属性可能存在于field_info、method_info和ClassFile表内的attributes项中，它是一个可选的项，它指明了为编译器所产生的字段、方法或者类型。未出现在源代码中的类成员必须使用Synthetic属性标注。Synthetic属性在Java 1.1版本中加入，用来提供对内嵌类的支持。Synthetic_attribute表的格式如表6-40所示。

表6-40 Synthetic属性表的格式

类 型	名 称	数 量
u2	attribute_name_index	1
u4	attribute_length	1

Synthetic_attribute表中各项如下所示：

attribute_name_index

attribute_name_index给出包含字符串“Synthetic”的CONSTANT_Utf8_info入口的常量池索引。

attribute_length

attribute_length必须为0。

6.8 一个模拟：“Getting Loaded”

如图6-3所示的“Getting Loaded” applet模拟了一个Java虚拟机装载class文件的过程。这里class文件是由1.1版本的javac编译器从下面的Java源代码中产生。尽管模拟使用的代码片断在现实世界中并不一定有用，但它还是被编译为一个真正的class文件，并作为一个合理的class文件格式的简单示例提供给读者。该类就是第5章中描述的“External Math” applet所用的同一个类。

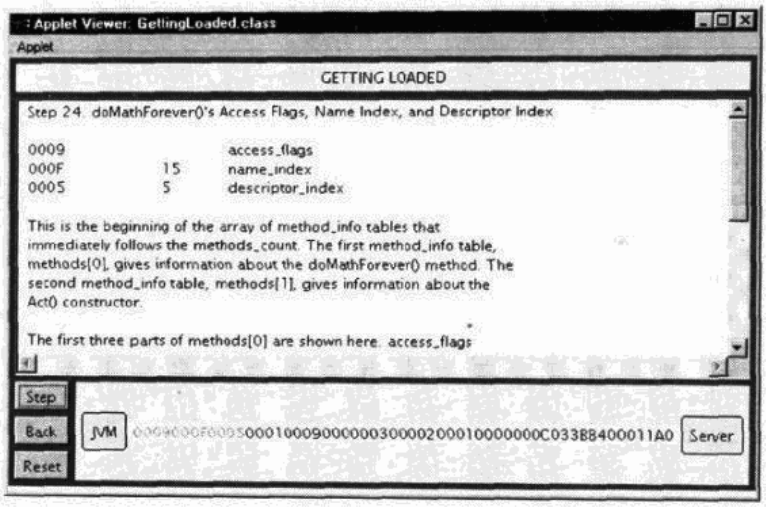


图6-3 “Getting Loaded” applet

“Getting Loaded” applet实现单步执行类装载的模拟功能。在这个过程中的每一步中，都可以看到下一段将要被Java虚拟机读取并解析的代码。按下“Step”按钮，可以使Java虚拟机读取下一段代码，按下“Back”按钮，可以撤销上一步骤，按下“Reset”，整个模拟将返回初始状态，这样就可以重新开始。

Java虚拟机被显示在窗口下方的左边，它读取组成class文件Act.class的字节流。字节在屏幕下方的右边显示，它们使用十六进制流的方式从服务器出来。字节在服务器和Java虚拟机之间，按照从右至左的顺序，一次一段代码进行传送。按下左边的“Step”按钮后，被Java虚拟机读取的代码使用红色显示。这些着重显示的代码的详细描述显示在Java虚拟机上面的文本区域中。下一段代码中所有剩余的字节以黑色显示。

如前所述，class文件中的许多项都指向常量池入口。为了在单步执行模拟时能够方便地查阅常量池入口，Act类的常量池内容列表如表6-41所示。

每一段代码都在文本区域中有详细解释。由于文本区域中的细节太多，读者可能希望快速执行一遍整个过程，然后再回头了解更多的细节。

表6-41 类Act的常量池

索 引	类 型	值
1	CONSTANT_Class_info	7
2	CONSTANT_Class_info	16
3	CONSTANT_Methodref_info	2，4
4	CONSTANT_NameAndType_info	6，5
5	CONSTANT_Utf8_info	“( ) V”
6	CONSTANT_Utf8_info	“<init>”
7	CONSTANT_Utf8_info	“Act”

（续）

索　引	类　型	值
8	CONSTANT_Utf8_info	"Act.java"
9	CONSTANT_Utf8_info	"Code"
10	CONSTANT_Utf8_info	"ConstantValue"
11	CONSTANT_Utf8_info	"Exceptions"
12	CONSTANT_Utf8_info	"LineNumberTable"
13	CONSTANT_Utf8_info	"LocalVariables"
14	CONSTANT_Utf8_info	"SourceFile"
15	CONSTANT_Utf8_info	"doMathForever"
16	CONSTANT_Utf8_info	"java/lang/Object"

6.9 随书光盘

光盘的classfile目录中有本章中用到的示例源代码。光盘上的applets/GettingLoaded.html文件里包含了“Getting Loaded” applet。这个applet的源代码可以与它的class文件放在一起，其具体位置是applets/GettingLoaded目录。

6.10 资源页

如果需要了解更多有关class文件的信息，请访问资源页：http://www.artima.com/insidejvm/resources。

第7章　类型的生命周期

前一章详细地描述了Java class文件的格式，它以标准的二进制形式来表现Java类型。这一章我们来看看当二进制的类型数据被导入到Java虚拟机中时，到底会发生什么。我们以一个Java类型（类或接口）的生命周期（从它进入虚拟机开始一直到最终退出）为例来讨论开始阶段的装载、连接和初始化，以及占Java类型生命周期绝大部分时间的对象实例化、垃圾收集和对象终结，然后是Java类型生命周期的结束，也就是从虚拟机中卸载。

7.1　类型装载、连接与初始化

Java虚拟机通过装载、连接和初始化一个Java类型，使该类型可以被正在运行的Java程序所使用。其中，装载就是把二进制形式的Java类型读入Java虚拟机中；而连接就是把这种已经读入虚拟机的二进制形式的类型数据合并到虚拟机的运行时状态中去。连接阶段分为三个子步骤——验证、准备和解析。“验证”步骤确保了Java类型数据格式正确并且适于Java虚拟机使用。而“准备”步骤则负责为该类型分配它所需的内存，比如为它的类变量分配内存。“解析”步骤则负责把常量池中的符号引用转换为直接引用。虚拟机的实现可以推迟解析这一步，它可以在当运行中的程序真正使用某个符号引用时再去解析它（把该符号引用转换为直接引用）。当验证、准备和（可选的）解析步骤都完成了时，该类型就已经为初始化做好了准备。在初始化期间，都将给类变量赋以适当的初始值。整个过程如图7-1所示。

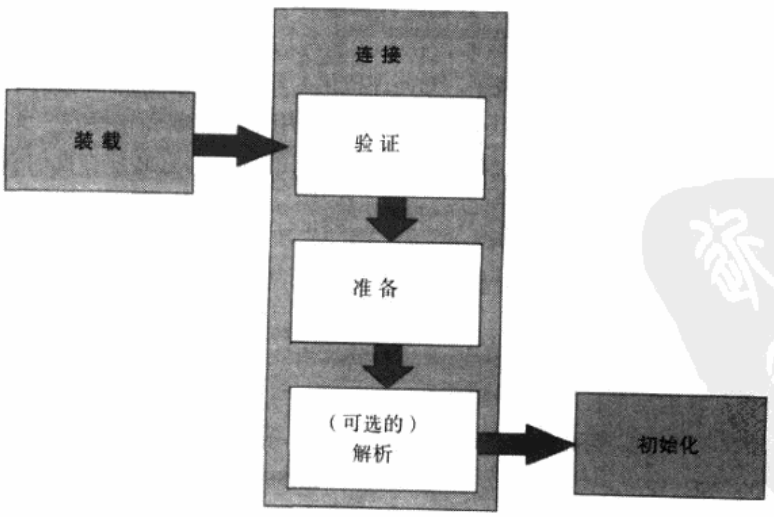


图7-1　类型生命周期的开始

就像在图7-1中所看到的那样，装载、连接和初始化这三个阶段必须按顺序进行。惟一的例

外就是连接阶段的第三步——解析，它可以在初始化之后再进行。

在类和接口被装载和连接的时机上，Java虚拟机规范给实现提供了一定的灵活性。但是它严格地定义了初始化的时机。所有的Java虚拟机实现必须在每个类或接口首次主动使用时初始化。下面这六种情形符合主动使用的要求。

- 当创建某个类的新实例时（或者通过在字节码中执行new指令；或者通过不明确的创建、反射、克隆或者反序列化）。
- 当调用某个类的静态方法时（即在字节码中执行invokestatic指令时）。
- 当使用某个类或接口的静态字段，或者对该字段赋值时（即在字节码中，执行getstatic或putstatic指令时），用final修饰的静态字段除外，它被初始化为一个编译时的常量表达式。
- 当调用Java API中的某些反射方法时，比如类Class中的方法或者java.lang.reflect包中的类的方法。
- 当初始化某个类的子类时（某个类初始化时，要求它的超类已经被初始化了）。
- 当虚拟机启动时某个被标明为启动类的类（即含有main（）方法的那个类）。

除上述这六种情形外，所有其他使用Java类型的方式都是被动使用，它们都不会导致Java类型的初始化。本章稍后会给出几个例子以说明主动使用和被动使用的区别。

在上面我们曾提到，任何一个类的初始化都要求它的超类在此之前已经初始化了。以此类推，该规则就意味着某个类的所有祖先类必须在该类之前被初始化。然而，对于接口来说，这条规则并不适用。只有在某个接口所声明的非常量字段被使用时，该接口才会被初始化，而不会因为实现这个接口的子接口或类要初始化而被初始化。因而，任何一个类的初始化都要求它的所有祖先类（而不是祖先接口）预先被初始化。而一个接口的初始化，并不要求它的祖先接口预先被初始化。

“在首次主动使用时初始化”这个规则直接影响着装载、连接和初始化类的机制。在首次主动使用时，其类型必须被初始化。然而，在类型能被初始化之前，它必须已经被连接了，而在它能被连接之前，它必须已经被装载了。Java虚拟机的实现可以根据需要在更早的时候装载以及连接类型，没有必要一直要等到该类型的首次主动使用才去装载和连接它。无论如何，如果一个类型在它的首次主动使用之前还没有被装载和连接的话，那它必须在此时被装载和连接，这样它才能被初始化。

7.1.1 装载

装载阶段由三个基本动作组成，要装载一个类型，Java虚拟机必须：

- 通过该类型的完全限定名，产生一个代表该类型的二进制数据流。
- 解析这个二进制数据流为方法区内的内部数据结构。
- 创建一个表示该类型的java.lang.Class类的实例。

这个二进制数据流可能遵守Java class文件格式，但是也可能遵守其他的格式。就像前一章提到的那样，所有的Java虚拟机实现必须能识别Java class文件格式，但是个别的实现也可以识别其他的二进制格式。

Java虚拟机规范并没有说Java类型的二进制数据应该怎样产生。下面是一些可能的产生“类型的二进制数据”的方式：

- 从本地文件系统装载一个Java class文件。
- 通过网络下载一个Java class文件。
- 从一个ZIP、JAR、CAB或者其他某种归档文件中提取Java class文件。
- 从一个专有数据库中提取Java class文件。
- 把一个Java源文件动态编译为class文件格式。
- 动态为某个类型计算其class文件数据。
- 使用上述任何方法，但是使用不同于Java class文件的其他二进制文件格式。

有了类型的二进制数据之后，Java虚拟机必须对这些数据进行足够的处理，然后它才能创建类java.lang.Class的实例对象。虚拟机必须把这些二进制数据解析为与实现相关的内部数据结构（请参考第5章中关于存储类数据的可能数据结构的讨论）。装载步骤的最终产品就是这个Class类的实例对象，它成为Java程序与内部数据结构之间的接口。要访问关于该类型的信息（它们是存储在内部数据结构中的），程序就要调用该类型对应的Class实例对象的方法。这样一个过程，就是把一个类型的二进制数据解析为方法区中的内部数据结构、并在堆上建立一个Class对象的过程，这被称为“创建”类型。

就像在前一章曾提到的，Java类型要么由启动类装载器装载，要么通过用户自定义的类装载器装载。启动类装载器是虚拟机实现的一部分，它以与实现无关的方式装载类型（包括Java API的类和接口），用户自定义的类装载器是类java.lang.ClassLoader的子类实例，它以定制的方式装入类。用户自定义的类装载器的内部工作机制将在第8章中更详细地说明。

类装载器（启动型或者用户自定义的）并不需要一直等到某个类型“首次主动使用”时再去装入它。Java虚拟机规范允许类装载器缓存Java类型的二进制表现形式，在预料某个类型将要被使用时就装载它，或者把这些类型装载到一些相关的分组里面。如果一个类装载器在预先装载时遇到问题，无论如何，它应该在该类型被首次主动使用时报告该问题（通过抛出一个LinkageError异常的子类）。换句话说，如果一个类装载器在预先装载时遇到缺失或者错误的class文件，它必须等到程序首次主动使用该类时才报告错误。如果这个类一直没有被程序主动使用，那么该类装载器将不会报告错误。

7.1.2 验证

当类型被装载后，就准备进行连接了。连接过程的第一步是验证——确认类型符合Java语言的语义，并且它不会危及虚拟机的完整性。

在验证上，不同的虚拟机实现拥有一些灵活性。虚拟机实现的设计者可以决定如何以及何时验证类型。Java虚拟机规范列出了虚拟机可以抛出的异常以及在何种条件下必须抛出它们。不管Java虚拟机可能遇到了什么样的麻烦，都应该有一个异常或者错误可以抛出。规范表明了在每种情形下应该抛出何种异常或者错误。某些情况下，规范明确地说明何时这种异常或者错误应该被抛出，但是通常没有严格地规定应该如何或者在何时检查错误条件。

不管怎样，在大多数Java虚拟机实现中特定类型的检查一般都在特定的时间发生。比如，在装载过程中，虚拟机必须解析代表类型的二进制数据流，并且构造内部数据结构。在这个时候，必须做一些特定的检查，以保证解析二进制数据的初始工作不会导致虚拟机崩溃。在这个解析期间，虚拟机大多会检查二进制数据以确保数据全部是预期的格式。Java class文件格式的解析

器可能检查魔数，确保每一个部分都在正确的位置，拥有正确的长度，验证文件不是太长或者太短，等等。虽然这些检查在装载期间完成，是在正式的连接验证阶段之前进行，但它们仍然在逻辑上属于验证阶段。检查被装载的类型是否有任何问题的整个过程都属于验证。

另外一个很可能在装载时进行的检查是，确保除了Object之外的每一个类都有一个超类。在装载时检查的原因是当虚拟机装载一个类时，它必须确保该类的所有超类都已经被装载了。对于给定的类，得到其超类名字的惟一方法就是观察类的二进制数据。因为Java虚拟机无论如何都要在装载的时候检查每个类的超类数据，所以在装载阶段做这个检查是顺理成章的。

在大部分虚拟机实现中，还有一种检查往往发生在正式的验证阶段之后，那就是符号引用的验证。在前面的章节中描述过，动态连接的过程包括通过保存在常量池中的符号引用查找被引用的类、接口、字段以及方法，把符号引用替换成直接引用。当虚拟机搜寻一个被符号引用的元素（类型、字段或者方法）时，它必须首先确认该元素存在。如果虚拟机发现元素存在，它必须进一步检查引用类型有访问该元素的权限。这些对存在性和访问权限的检查逻辑上是验证的一部分，属于连接的第一阶段，但是往往在解析的时候发生，那是连接的第三阶段。解析自身也可能延迟到符号引用第一次被程序所使用时，所以这些检查甚至有可能在初始化之后才进行。

那么在正式的验证阶段做哪些检查呢？任何在此之前还没有进行的检查以及在此之后不会被检查的项目都包含在内。在正式的验证阶段需要完成的候选检查在下面列出。首先列出确保各个类之间二进制兼容的检查：

- 检查final的类不能拥有子类。
- 检查final的方法不能被覆盖。
- 确保在类型和超类型之间没有不兼容的方法声明（比如两个方法拥有同样的名字，参数在数量、顺序、类型上都相同，但是返回类型不同）。

请注意，当这些检查需要查看其他类型的时候，它只需要查看超类型。超类需要在子类初始化前被初始化，所以这些类应该已经被装载了。当实现了父接口的类被初始化的时候，不需要初始化父接口。然而，当实现了父接口的子类（或者是扩展了父接口的子接口）被装载时，父接口也必须被装载。（它们不会被初始化，只是被装载了，可能被某些虚拟机实现可选地连接了。）装载一个类的时候，它所有的超类都会被装载。在验证期间，这个类和它所有的超类型都需要确保互相之间仍然二进制兼容。

- 检查所有的常量池入口相互之间一致（比如，一个CONSTANT_String_info入口的string_index项目必须是一个CONSTANT_Utf8_info入口的索引）。
- 检查常量池中的所有的特殊字符串（类名、字段名和方法名、字段描述符和方法描述符）是否符合格式。
- 检查字节码的完整性。

上面所列出的最复杂的任务就是字节码验证。所有的Java虚拟机都必须设法为它们执行的每个方法检验字节码的完整性。比如，不能因为一个超出了方法末尾的跳转指令就导致虚拟机实现崩溃。它们必须在字节码验证的时候检查出这样的跳转指令是非法的，从而抛出一个错误。

虚拟机的实现没有强求在正式的连接验证阶段进行字节码验证。比如，实现可以自由地选择在执行每条语句的时候单独进行验证。然而，Java虚拟机指令集设计的一个目标就是使得字节

码流可以通过一次性使用一个数据流分析器进行验证。在连接过程中一次性验证字节码流，而非在程序执行的时候动态验证，使得Java程序的运行速度得到很大的提高。

当通过一个数据流分析器进行字节码验证的时候，虚拟机可能不得不为了确保符合Java语言的语义而装载其他的类。比如，设想一个类包含一个方法，其中把一个java.lang.的实例的引用赋值给了一个java.lang.Number类型的字段。在这个情况下，虚拟机将在字节码验证的时候装载类Float，确保这是一个Number类的子类。它也不得不装载Number来确保它没有被声明为final。虚拟机此时不需要初始化Float，只需要装载它。Float会在首次主动使用时被初始化。

更多关于类验证过程的信息，请参见第3章。

7.1.3 准备

随着Java虚拟机装载了一个类，并执行了一些它选择进行的验证之后，类就可以进入准备阶段了。在准备阶段，Java虚拟机为类变量分配内存，设置默认初始值。但在到达初始化阶段之前，类变量都没有被初始化为真正的初始值。（在准备阶段是不会执行Java代码的。）在准备阶段，虚拟机把给类变量新分配的内存根据类型设置为默认值。不同类型的默认值在表7-1中列出。

表7-1 主要类型和引用类型的默认初始值

类　型	默认初始值
int	0
long	0L
short	(short) 0
char	'\u0000'
byte	(byte) 0
boolean	false
reference	null
float	0.0f
double	0.0d

虽然在表7-1中出现了boolean类型，Java虚拟机不太支持boolean。在内部，boolean常常被实现为一个int，会被默认地置为0（就是boolean取false值）。因此，boolean类变量，就算它们在内部是被作为int实现的，也总是被初始化成false。

在准备阶段，Java虚拟机实现可能也为一些数据结构分配内存，目的是提高运行程序的性能。这种数据结构的例子如方法表，它包含指向类中每一个方法（包括从超类继承的方法）的指针。方法表可以使得继承的方法执行时不需要搜索超类。第8章中描述了方法表的更多细节。

7.1.4 解析

类型经过了连接的前两个阶段——验证和准备——之后，它就可以进入第三个（也就是最后一个）连接阶段了——解析。解析过程就是在类型的常量池中寻找类、接口、字段和方法的符号引用，把这些符号引用替换成直接引用的过程。在前面提到过，在符号引用被程序首次使用之前，连接的这个步骤都是可选的。第8章中描述了解析的更多细节。

7.1.5 初始化

为了准备让一个类或者接口被首次主动使用，最后一个步骤就是初始化，也就是为类变量

赋予正确的初始值。这里的“正确”初始值指的是程序员希望这个类变量所具备的起始值。正确的初始值是和在准备阶段赋予的默认初始值对比而言的。前面说过，根据类型的不同，类变量已经被赋予了默认初始值。而正确的初始值是根据程序员制定的主观计划而生成的。

在Java代码中，一个正确的初始值是通过类变量初始化语句或者静态初始化语句给出的。一个类变量初始化语句是变量声明后面的等号和表达式：

```
// On CD-ROM in file classlife/ex1/Example1a.java
class Example1a {

    // "= 3 * (int) (Math.random() * 5.0)" is the class variable
    // initializer
    static int size = 3 * (int) (Math.random() * 5.0);
}
```

静态初始化语句是一个以static关键字开头的程序块：

```
// On CD-ROM in file classlife/ex1/Example1b.java
class Example1b {

    static int size;

    // This is the static initializer
    static {

        size = 3 * (int) (Math.random() * 5.0);
    }
}
```

所有的类变量初始化语句和类型的静态初始化器都被Java编译器收集在一起，放到一个特殊的方法中。对于类来说，这个方法被称作类初始化方法；对于接口来说，它被称为接口初始化方法。在类和接口的Java class文件中，这个方法被称为“<clinit>”。通常的Java程序方法是无法调用这个<clinit>方法的。这种方法只能被Java虚拟机调用，专门用把类型的静态变量设置为它们的正确初始值。

初始化一个类包含两个步骤：

1) 如果类存在直接超类的话，且直接超类还没有被初始化，就先初始化直接超类。

2) 如果类存在一个类初始化方法，就执行此方法。

当初始化一个类的直接超类的时候，也是需要包含这两个步骤。因此，第一个被初始化的类永远是Object，然后是被主动使用的类的继承树上所有的类。超类总是在子类之前被初始化。

初始化接口并不需要初始化它的父接口，因此初始化一个接口只需一步：如果接口存在一个接口初始化方法的话，就执行此方法。

<clinit>（）方法的代码并不显式地调用超类的<clinit>（）方法。在Java虚拟机调用类的<clinit>（）方法之前，它必须确认超类的<clinit>（）方法已经被执行了。

Java虚拟机必须确保初始化过程被正确地同步。如果多个线程需要初始化一个类，仅仅允许

一个线程来执行初始化，其他的线程序需要等待。当活动的线程完成了初始化过程之后，它必须通知其他等待的线程。第20章介绍了有关同步、等待和通知的内容。

1. <clinit>（）方法

前面说过，Java编译器把类变量初始化语句和静态初始化语句的代码都放到class文件的<clinit>（）方法中，顺序就按照它们在类或者接口声明中出现的顺序。思考下面的类的例子：

```
// On CD-ROM in file classlife/ex1/Example1c.java
class Example1c {

    static int width;
    static int height = (int) (Math.random() * 2.0);

    // This is the static initializer
    static {

        width = 3 * (int) (Math.random() * 5.0);
    }
}
```

Java编译器生成了下面的<clinit>（）方法：

```
// The code for height's class variable initializer begins here
                          // Invoke Math.random(), which will push
                          // a double return value
 0 invokestatic #6 <Method double random()>
 3 ldc2_w #8 <Double 2.0>  // Push double constant 2.0
 6 dmul                    // Pop two doubles, multiply, push result
 7 d2i                     // Pop double, convert to int, push int
                           // Pop int, store into class variable
                           // height
 8 putstatic #5 <Field int height>

// The code for the static initializer begins here
11 iconst_3                // Push int constant 3
                           // Invoke Math.random(), which will push
                           // a double return value
12 invokestatic #6 <Method double random()>
15 ldc2_w #10 <Double 5.0> // Push double constant 5.0
18 dmul                    // Pop two doubles, multiply, push result
19 d2i                     // Pop double, convert to int, push int
20 imul                    // Pop two ints, multiply, push int result
                           // Pop int, store into class variable
                           // width
21 putstatic #7 <Field int width>
24 return                  // Return void from <clinit> method
```

<clinit>（）方法首先执行Example1c惟一的一个类变量初始化代码，初始化了height变量，

然后执行静态初始化语句，初始化了width变量。初始化按照这个顺序进行是因为在Example1c类的源代码中，类变量初始化语句出现在静态初始化语句之前。

并非所有的类都需要在它们的class文件中拥有一个<clinit>（）方法。如果类没有声明任何类变量，也没有静态初始化语句，那么它就不会有<clinit>（）方法。如果类声明了类变量，但是没有明确使用类变量初始化语句或者静态初始化语句初始化它们，那么类不会有<clinit>（）方法。如果类仅包含静态final变量的类变量初始化语句，而且这些类变量初始化语句采用编译时常量表达式，类也不会有<clinit>（）方法。只有那些的确需要执行Java代码来赋予类变量正确初始值的类才会有类初始化方法。

下面是一个类的例子，Java编译器不会为它产生<clinit>（）方法。

```
// On CD-ROM in file classlife/ex1/Example1d.java
class Example1d {

    static final int angle = 35;
    static final int length = angle * 2;
}
```

类Example1d声明了两个常量——angle和length，并且通过表达式给它们赋予了初始值，这些表达式是编译时常量。编译器知道angle表示35，length表示70。但Example1d类被Java虚拟机装载的时候，angle和length并没有作为类变量保存在方法区中。因此，不需要<clinit>（）方法来初始化它们。angle和length字段并非类变量，它们是常量，被Java编译器特殊处理了。

Example1d的angle和length字段没有被当做类变量，Java虚拟机在使用它们的任何类的常量池或者字节码流中直接存放的是它们表示的常量的int值。比如，如果一个类使用了Example1d的angle字段，该类不会在常量池中保存一个指向Example1d类的angle字段的符号引用，而是直接在它们的字节码流中嵌入一个值35。如果angle的常量值超过了short值的限制（-32 768 ~ 32 767），比如是35000，那么类会在它的常量池中保存一个CONSTANT_Integer_info入口，值为35000。

下面是一个类的例子，它同时使用一个常量和一个其他类的类变量。

```
// On CD-ROM in file classlife/ex1/Example1e.java
class Example1e {

    // The class variable initializer for symbolicRef uses a symbolic
    // reference to the size class variable of class Example1a
    static int symbolicRef = Example1a.size;

    // The class variable initializer for localConst doesn't use a
    // symbolic reference to the length field of class Example1d.
    // Instead, it just uses a copy of the constant value 70.
    static int localConst = Example1d.length * (int) (Math.random()
        * 3.0);
}
```

Java编译器为类Example1e产生如下的<clinit>（）方法：

```
// The code for symbolicRef's class variable initializer begins here:
                          // Push int value from Example1a.size.
                          // This getstatic instruction refers to a
                          // symbolic reference to Example1a.size.
 0 getstatic #9 <Field int size>
                          // Pop int, store into class variable
                          // symbolicRef
 3 putstatic #10 <Field int symbolicRef>

// The code for localConst's class variable intializer begins here:
                          // Expand byte operand to int, push int
                          // result. This is the local copy of
 6 bipush 70              // Example1d's length constant, 70.
                          // Invoke Math.random(), which will push
                          // a double return value
 8 invokestatic #8 <Method double random()>
11 ldc2_w #11 <Double 3.0> // Push double constant 3.0
14 dmul                   // Pop two doubles, multiply, push result
15 d2i                    // Pop double, convert to int, push int
16 imul                   // Pop two ints, multiply, push int result
                          // Pop int, store into class variable
                          // localConst
17 putstatic #7 <Field int localConst>
20 return                 // Return void from <clinit> method
```

偏移量为0的getstatic指令使用一个符号引用（位于常量池入口9）指向类Example1a的size字段。在偏移量6的bipush指令后面跟随了一个字节，保存了Example1d.length表示的常量值。Example1e的常量池没有包含指向Example1d的任何符号引用。

接口也可能在class文件中包含一个<clinit>（）方法。所有在接口中声明的隐式公开（public）、静态（static）和最终（final）字段都必须在字段初始化语句中初始化。如果接口包含任何不能在编译时被解析成为一个常量的字段初始化语句，接口就会拥有一个<clinit>（）方法。下面是一个例子：

```
// On CD-ROM in file classlife/ex1/Example1f.java
interface Example1f {

    int ketchup = 5;
    int mustard = (int) (Math.random() * 5.0);
}
```

Java编译器为接口Example1f生成了下列<clinit>（）方法：

```
// The code for mustard's class variable initializer begins here
                          // Invoke Math.random(), which will push
                          // a double return value
 0 invokestatic #6 <Method double random()>
 3 ldc2_w #7 <Double 5.0>  // Push double constant 2.0
```

```
 6 dmul                    // Pop two doubles, multiply, push result
 7 d2i                     // Pop double, convert to int, push int
                           // Pop int, store into class variable
                           // mustard
 8 putstatic #5 <Field int mustard>
11 return                  // Return void from <clinit> method
```

请注意，只有mustard字段被这个<clinit>（）方法初始化了。因为ketchup字段被初始化为一个编译时常量，它被编译器特殊处理了。虽然使用Example1f.mustard的类型保存指向这个字段的符号引用，但是使用Example1f.ketchup的类型将会保存ketchup的常量值5的一份本地拷贝。

2. 主动使用和被动使用

在前面讲过，Java虚拟机在首次主动使用类型时初始化它们。只有6种活动被认为是主动使用：创建类的新实例，调用类中声明的静态方法，操作类或者接口中声明的非常量静态字段，调用Java API中特定的反射方法，初始化一个类的子类，以及指定一个类作为Java虚拟机启动时的初始化类。

使用一个非常量的静态字段只有当类或者接口的确声明了这个字段时才是主动使用。比如，类中声明的字段可能会被子类引用；接口中声明的字段可能会被子接口或者实现了这个接口的类引用。对于子类、子接口和实现了接口的类来说，这就是被动使用——使用它们并不会触发它们的初始化。只有当字段的确是被类或者接口声明的时候才是主动使用。下面的例子说明了这个原理：

```
// On CD-ROM in file classlife/ex2/NewParent.java
class NewParent {

    static int hoursOfSleep = (int) (Math.random() * 3.0);

    static {
        System.out.println("NewParent was initialized.");
    }
}

// On CD-ROM in file classlife/ex2/NewbornBaby.java
class NewbornBaby extends NewParent {

    static int hoursOfCrying = 6 + (int) (Math.random() * 2.0);

    static {
        System.out.println("NewbornBaby was initialized.");
    }
}

// On CD-ROM in file classlife/ex2/Example2.java
class Example2 {

    // Invoking main() is an active use of Example2
```

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

        // Using hoursOfSleep is an active use of NewParent,
        // but a passive use of NewbornBaby
        int hours = NewbornBaby.hoursOfSleep;
        System.out.println(hours);
    }

    static {
        System.out.println("Example2 was initialized.");
    }
}
```

在上面的例子中，执行Example2的main（）方法只会导致Example2和NewParent被初始化。NewbornBaby没有被初始化，也不需要被装载。标准输出打印出如下文字：

```
Example2 was initialized.
NewParent was initialized.
2
```

如果一个字段既是静态（static）的又是最终（final）的，并且使用一个编译时常量表达式初始化，使用这样的字段，就不是对声明该字段的类的主动使用。在前面说过，Java编译器把这样的字段解析成对常量的本地拷贝（该常量存在于引用者类的常量池中或者字节码流中，或者二者都有）。下面是一个说明这种对静态final字段特殊处理的例子：

```
// On CD-ROM in file classlife/ex3/Angry.java
interface Angry {

    String greeting = "Grrrr!";

    int angerLevel = Dog.getAngerLevel();
}

// On CD-ROM in file classlife/ex3/Dog.java
class Dog {

    static final String greeting = "Woof, woof, world!";

    static {
        System.out.println("Dog was initialized.");
    }

    static int getAngerLevel() {

        System.out.println("Angry was initialized");
        return 1;
    }
```

```
}

// On CD-ROM in file classlife/ex3/Example3.java
class Example3 {

    // Invoking main() is an active use of Example3
    public static void main(String[] args) {

        // Using Angry.greeting is a passive use of Angry
        System.out.println(Angry.greeting);

        // Using Dog.greeting is a passive use of Dog
        System.out.println(Dog.greeting);
    }

    static {
        System.out.println("Example3 was initialized.");
    }
}
```

运行Example3程序，得到如下输出：

```
Example3 was initialized.
Grrrr!
Woof, woof, world!
```

如果Angry被初始化了，字符串“Angry was initialized.”就应该被输出到标准输出。同理，如果Dog被初始化了，字符串“Dog was initialized.”应该被输出到标准输出。读者可以从上面的输出中看到，接口Angry和类Dog都没有在Example3程序的执行中被初始化。

要了解更多的关于静态final变量的特殊处理的知识，参阅第8章。

7.2 对象的生命周期

一旦一个类被装载、连接和初始化，它就随时可以使用了。程序可以访问它的静态字段，调用它的静态方法，或者创建它的实例。本节会讨论类的实例化和初始化，即对象生命起始阶段的活动；还要讨论垃圾收集和终结，即对象生命尽头的活动。

7.2.1 类实例化

在Java程序中，类可以被明确或者隐含地实例化。实例化一个类有四种途径：明确地使用new操作符；调用Class或者java.lang.reflect.Constructor对象的newInstance（）方法；调用任何现有对象的clone（）方法；或者通过java.io.ObjectInputStream类的getObject（）方法反序列化。下面的例子中演示了其中三种创建新的类实例的方法：

```
// On CD-ROM in file classlife/ex4/Example4.java
class Example4 implements Cloneable {

    Example4() {
```

```
        System.out.println("Created by invoking newInstance()");
    }

    Example4(String msg) {
        System.out.println(msg);
    }

    public static void main(String[] args)
        throws ClassNotFoundException, InstantiationException,
        IllegalAccessException, CloneNotSupportedException {

        // Create a new Example4 object with the new operator
        Example4 obj1 = new Example4("Created with new.");

        // Get a reference to the Class instance for Example4, then
        // invoke newInstance() on it to create a new Example4 object
        Class myClass = Class.forName("Example4");
        Example4 obj2 = (Example4) myClass.newInstance();

        // Make an identical copy of the the second Example4 object
        Example4 obj3 = (Example4) obj2.clone();
    }
}
```

运行后，Example4程序打印出了如下的输出：

```
Created with new.
Created by invoking newInstance()
```

除了这四种在Java源代码中明确地实例化对象的方法之外，还有几种情况下对象会被隐含地实例化，即在源代码中看不见明确的new、newInstance ( )、clone ( ) 或者ObjectInputStream.readObject ( )。

在任何Java程序中第一个隐含实例化对象可能就是保存命令行参数的String对象。每一个命令行参数都会有一个String对象的引用，把它们组织成一个String数组并作为一个参数传递到每一个程序的main ( ) 方法中。

另外两种隐含实例化类的方法和类装载的过程有关。首先，对于Java虚拟机装载的每一个类型，它会暗中实例化一个Class对象来代表这个类型。其次，当Java虚拟机装载了在常量池中包含CONSTANT_String_info入口的类的时候，它会创建新的String对象的实例来表示这些常量字符串。把方法区中的CONSTANT_String_info入口转换成一个堆中的String实例的过程是常量池解析过程的一部分，这个过程将在第8章详细描述。

还有一条隐含创建对象的途径是通过执行包含字符串连接操作符的表达式产生对象。如果这样的字符串不是一个编译时常量，用于中间处理的String和StringBuffer对象会在计算表达式的过程中创建。下面是一个例子：

```
// On CD-ROM in file classlife/ex5/Example5.java
```

```
class Example5 {

    public static void main(String[] args) {

        if (args.length < 2) {
            System.out.println("Must enter any two args.");
            return;
        }

        System.out.println(args[0] + args[1]);
    }
}
```

javac 为Example5的main ( ) 方法产生了下面的字节码：

```
 0 aload_0       // Push the objref from loc var 0 (args)
 1 arraylength   // Pop arrayref, calc array length, push int length
 2 iconst_2      // Push int constant 2
                 // Pop 2 ints, compare, branch if (length >= 2) to
 3 if_icmpge 15 // offset 15.
                 // Push objref from System.out
 6 getstatic #11 <Field java.io.PrintStream out>
                 // Push objref of string literal
 9 ldc #1 <String "Must enter any two args.">
                 // Pop objref to String param, objref to System.out,
                 // invoke println()
11 invokevirtual #12 <Method void println(java.lang.String)>
14 return        // Return void from main()
                 // Push objref from System.out
15 getstatic #11 <Field java.io.PrintStream out>

// The string concatenation operation begins here
                 // Allocate mem for new StringBuffer object, and
                 // initialize mem to default initial values, push
                 // objref to new object
18 new #6 <Class java.lang.StringBuffer>
21 dup           // Duplicate objref to StringBuffer object
22 aload_0       // Push ref from loc var 0 (args)
23 iconst_0      // Push int constant 0
                 // Pop int, arrayref, push String at arrayref[int],
24 aaload        // which is args[0]
                 // Pop objref, invoke String's class method
                 // valueOf(), passing it the objref to the args[0]
                 // String object. valueOf() calls toString() on the
                 // ref, and returns (and pushes) the result, which
                 // happens to be the original args[0] String. In this
                 // case, the stack will look precisely the same
```

```
                // before and after this instruction is executed.
                // Thus here, the 1.1 javac compiler has
                // over-enthusiastically generated an unnecessary
                // instruction.
25 invokestatic #14 <Method java.lang.String valueOf(
        java.lang.Object)>
                // Pop objref to args[0] String, objref of the
                // StringBuffer object, invoke <init>() method on the
                // StringBuffer object passing the args[0] objref as
                // the only parameter.
28 invokespecial #9 <Method java.lang.StringBuffer(java.lang.String)>
31 aload_0      // Push objref from loc var 0 (args)
32 iconst_1     // Push int constant 1
                // Pop int, arrayref, push String at arrayref[int],
33 aaload       // which is args[1]
                // Pop objref to args[1] String, objref of the
                // StringBuffer object (there's still another objref
                // to this same object on the stack because of the
                // dup instruction above), invoke append() method on
                // StringBuffer object, passing args[1] as the only
                // parameter. append() will return an objref to this
                // StringBuffer object, which will be pushed back
                // onto the stack.
34 invokevirtual #10 <Method java.lang.StringBuffer append(java.lang.String)>
                // Pop objref to StringBuffer (pushed by append()),
                // invoke toString() on it, which returns the value
                // of the StringBuffer as a String object. Push
                // objref of String object.
37 invokevirtual #13 <Method java.lang.String toString()>
// The string concatenation operation is now complete

                // Pop objref of concatenated String, objref of
                // System.out that was pushed by the getstatic
                // instruction at offset 15. Invoke println() on
                // System.out, passing the concatenated String as
                // the only parameter.
40 invokevirtual #12 <Method void println(java.lang.String)>
43 return       // Return void from main()
```

Example5的main（）方法的字节码包含了三个隐含创建的String对象和一个StringBuffer对象。其中两个String对象的引用作为传递到main（）方法的args数组的一部分，是通过位于偏移量为24和33的aaload指令压入栈的。StringBuffer是在偏移量为18的new指令创建的，被偏移量为28的invokespecial指令初始化。最后一个String对象代表args[0]和args[1]的连接，是通过调用StringBuffer对象的toString（）方法建立的，这是由位于偏移量为37的invokevirtual指令完成的。

当Java虚拟机创建一个类的新实例时，不管是明确的还是隐含的，首先都需要在堆中为保存

对象的实例变量分配内存。所有在对象的类中和它的超类中声明的变量（包括隐藏的实例变量）都要分配内存。在第5章中描述过，堆中对象的映像中其他一些与实现相关的元素，比如指向方法区中类数据的指针，大致也是在这个时间分配的。一旦虚拟机为新的对象准备好了堆内存，它立即把实例变量初始化为默认的初始值。这和在表7-1中为类变量赋予的默认初始值是一样的。

一旦虚拟机完成了为新对象分配内存和为实例变量赋默认初始值后，它随后就会为实例变量赋正确的初始值。根据创建对象的方法不同，Java虚拟机使用三种技术之一来完成这个工作。如果对象是通过clone（）调用来创建的，虚拟机把原来被克隆的实例变量中的值拷贝到新对象中。如果对象是通过一个ObjectInputStream的readObject（）调用反序列化的，虚拟机通过从输入流中读入的值来初始化那些非暂时性的实例变量。否则，虚拟机调用对象的实例初始化方法。实例初始化方法把对象的实例变量初始化为正确的初始值。

Java编译器为它编译的每一个类都至少生成一个实例初始化方法。在Java 的class文件中，这个实例初始化方法被称为“<init>”。针对源代码中每一个类的构造方法，Java编译器都产生一个<init>（）方法。如果类没有明确地声明任何构造方法，编译器默认产生一个无参数的构造方法，它仅仅调用超类的无参数构造方法。和其他的构造方法一样，编译器在class文件中创建一个<init>（）方法，对应它的默认构造方法。

一个<init>（）方法中可能包含三种代码：调用另一个<init>（）方法，实现对任何实例变量的初始化，构造方法体的代码。如果构造方法通过明确地调用同一个类中的另一个构造方法（一个this（）调用）开始，它对应的<init>（）方法由两部分组成：

- 一个同类的<init>（）方法的调用。
- 实现了对应构造方法的方法体的字节码。

如果构造方法不是通过一个this（）调用开始的，而且这个对象不是Object，<init>（）方法则由三部分组成：

- 一个超类的<init>（）方法的调用。
- 任意实例变量初始化方法的字节码。
- 实现了对应构造方法的方法体的字节码。

如果构造方法没有使用this（）调用开始，并且这个类是Object，上面列表中的第一个元素就不存在。因为Object没有超类，它的<init>（）方法就不能通过调用超类的<init>（）方法开始。

如果构造方法通过明确地调用超类的构造方法（一个super（）调用）开始，它的<init>（）方法会调用对应的超类的<init>（）方法。比如，如果一个构造方法通过明确地调用“super（int，String）构造方法”开始，对应的<init>（）方法会从调用超类的“<init>（int，String）”方法开始。如果构造方法没有明确地从this（）或者super（）调用开始，对应的<init>（）方法默认会调用超类的无参数<init>（）方法。

下面的例子包含了三个构造方法，编号从1到3：

```
// On CD-ROM in file classlife/ex6/Example6.java
class Example6 {

    private int width = 3;

    // Constructor one:
```

```
    // This constructor begins with a this() constructor invocation,
    // which gets compiled to a same-class <init>() method
    // invocation.
    Example6() {
        this(1);
        System.out.println("Example6(), width = " + width);
    }

    // Constructor two:
    // This constructor begins with no explicit invocation of another
    // constructor, so it will get compiled to an <init>() method
    // that begins with an invocation of the superclass's no-arg
    // <init>() method.
    Example6(int width) {
        this.width = width;
        System.out.println("Example6(int), width = " + width);
    }

    // Constructor three:
    // This constructor begins with super(), an explicit invocation
    // of the superclass's no-arg constructor. Its <init>() method
    // will begin with an invocation of the superclass's no-arg
    // <init>() method.
    Example6(String msg) {
        super();
        System.out.println("Example6(String), width = " + width);
        System.out.println(msg);
    }

    public static void main(String[] args) {
        String msg
            = "The Agapanthus is also known as Lily of the Nile.";
        Example6 one = new Example6();
        Example6 two = new Example6(2);
        Example6 three = new Example6(msg);
    }
}
```

执行后，Example6程序打印出如下输出：

```
Example6(int), width = 1
Example6(), width = 1
Example6(int), width = 2
Example6(String), width = 3
The Agapanthus is also known as Lily of the Nile.
```

Example6的无参数<init>（）方法（对应第一个构造方法的<init>（）方法）的字节码如下：

```
// The first component, the same-class <init>() invocation, begins
// here:
 0 aload_0        // Push the objref from loc var 0 (this)
 1 iconst_1       // Push int constant 1
                  // Pop int and objref, invoke <init>() method on
                  // objref (this), passing the int (a 1) as the
                  // only parameter.
 2 invokespecial #12 <Method Example6(int)>

// The second component, the body of the constructor, begins
// here:
                  // Push objref from System.out
 5 getstatic #16 <Field java.io.PrintStream out>
                  // Allocate mem for new StringBuffer object, and
                  // initialize mem to default initial values, push
                  // objref to new object
 8 new #8 <Class java.lang.StringBuffer>
11 dup            // Duplicate objref to StringBuffer object
                  // Push objref to String literal from constant pool
12 ldc #1 <String "Example6(), width = ">
                  // Pop objref to literal String, pop objref of the
                  // StringBuffer object, invoke <init>() method on the
                  // StringBuffer object passing the args[0] objref as
                  // the only parameter.
14 invokespecial #14 <Method java.lang.StringBuffer(
       java.lang.String)>
17 aload_0        // Push objref from loc var 0 (this)
                  // Pop this reference, Push int value of width field
18 getfield #19 <Field int width>
                  // Pop int (width), pop objref (StringBuffer object),
                  // invoke append() on StringBuffer object passing the
                  // width int as the only parameter. append() will add
                  // the string representation of the int to the end of
                  // the buffer, and return an objref to the same
                  // StringBuffer object.
21 invokevirtual #15 <Method java.lang.StringBuffer append(int)>
                  // Pop objref to StringBuffer (pushed by append()),
                  // invoke toString() on it, which returns the value
                  // of the StringBuffer as a String object. Push
                  // objref of String object.
24 invokevirtual #18 <Method java.lang.String toString()>
                  // Pop objref of String, pop objref of System.out
                  // that was pushed by the getstatic instruction at
                  // offset 5. Invoke println() on System.out,
                  // passing the String as the only parameter:
                  // System.out.println("Example6(), width = "
```

```
                //      + width);
 27 invokevirtual #17 <Method void println(java.lang.String)>
 30 return        // Return void from <init>()
```

注意这个对应第一个构造方法的<init>（）方法从一个同类的<init>（）方法的调用开始，然后执行对应的构造方法体。因为构造方法从一个this（）调用开始，它对应的<init>（）方法不包含任何实例变量初始化方法的字节码。

Example6的使用一个int参数的<init>（）方法（对应第二个构造方法的<init>（）方法）的字节码如下：

```
// The first component, the superclass <init>() invocation, begins
// here:
 0 aload_0       // Push the objref from loc var 0 (this)
                 // Pop objref (this), invoke the superclass's
                 // no-arg<init>() method on objref.
 1 invokespecial #11 <Method java.lang.Object()>

// The second component, the instance variable initializers, begins
// here:
 4 aload_0       // Push the objref from loc var 0 (this)
 5 iconst_3      // Push int constant 3
                 // Pop int (3), pop objref (this), store 3 into
                 // width instance variable of this object
 6 putfield #19 <Field int width>

// The third component, the body of the constructor, begins
// here:
 9 aload_0       // Push the objref from loc var 0 (this)
10 iload_1       // Push int from loc var 1 (int param width)
                 // Pop int (param width), pop objref (this), store
                 // int param value into width field of this object:
                 // this.width = width
11 putfield #19 <Field int width>
                 // Push objref from System.out
14 getstatic #16 <Field java.io.PrintStream out>
                 // Allocate mem for new StringBuffer object, and
                 // initialize mem to default initial values, push
                 // objref to new object
17 new #8 <Class java.lang.StringBuffer>
20 dup           // Duplicate objref to StringBuffer object
                 // Push objref to String literal from constant pool
21 ldc #3 <String "Example6(int), width = ">
                // Pop objref to literal String, pop objref of the
                // StringBuffer object, invoke <init>() method on the
                // StringBuffer object passing the args[0] objref as
                // the only parameter.
```

```
23 invokespecial #14 <Method java.lang.StringBuffer(
        java.lang.String)>
26 iload_1          // Push int from loc var 1 (int param width)
                    // Pop int (width), pop objref (StringBuffer object),
                    // invoke append() on StringBuffer object passing the
                    // width int as the only parameter. append() will add
                    // the string representation of the int to the end of
                    // the buffer, and return an objref to the same
                    // StringBuffer object.
27 invokevirtual #15 <Method java.lang.StringBuffer append(int)>
                    // Pop objref to StringBuffer (pushed by append()),
                    // invoke toString() on it, which returns the value
                    // of the StringBuffer as a String object. Push
                    // objref of String object.
30 invokevirtual #18 <Method java.lang.String toString()>
                    // Pop objref of String, pop objref of System.out
                    // that was pushed by the getstatic instruction at
                    // offset 14. Invoke println() on System.out,
                    // passing the String as the only parameter:
                    // System.out.println("Example6(int), width = "
                    //     + width);
33 invokevirtual #17 <Method void println(java.lang.String)>
36 return           // Return void from <init>()
```

对应第二个构造方法的<init>（）方法包含三部分。第一部分是一个对超类（Object）的无参数<init>（）方法的调用。编译器默认生成这个调用，因为在第二个构造方法的方法体中第一条语句并不是一个明确的super（）调用。随后是第二个部分——width实例变量初始化方法的字节码。<init>（）方法包含的第三部分是第二个构造方法的方法体的字节码。

Example6的使用一个String参数的<init>（）方法（对应第三个构造方法的<init>（）方法）的字节码如下：

```
// The first component, the superclass <init>() invocation, begins
// here:
 0 aload_0          // Push the objref from loc var 0 (this)
                    // Pop objref (this), invoke the superclass's
                    // no-arg<init>() method on objref.
 1 invokespecial #11 <Method java.lang.Object()>

// The second component, the instance variable initializers, begins
// here:
 4 aload_0          // Push the objref from loc var 0 (this)
 5 iconst_3         // Push int constant 3
                    // Pop int (3), pop objref (this), store 3 into
                    // width instance variable of this object
 6 putfield #19 <Field int width>
```

```
// The third component, the body of the constructor, begins
// here:
                  // Push objref from System.out
 9 getstatic #16 <Field java.io.PrintStream out>
                  // Allocate mem for new StringBuffer object, and
                  // initialize mem to default initial values, push
                  // objref to new object
12 new #8 <Class java.lang.StringBuffer>
15 dup            // Duplicate objref to StringBuffer object
                  // Push objref to String literal from constant pool
16 ldc #2 <String "Example6(String), width = ">
                  // Pop objref to literal String, pop objref of the
                  // StringBuffer object, invoke <init>() method on the
                  // StringBuffer object passing the args[0] objref as
                  // the only parameter.
18 invokespecial #14 <Method java.lang.StringBuffer(
       java.lang.String)>
21 aload_0        // Push objref from loc var 0 (this)
                  // Pop this reference, Push int value of width field
22 getfield #19 <Field int width>
                  // Pop int (width), pop objref (StringBuffer object),
                  // invoke append() on StringBuffer object passing the
                  // width int as the only parameter. append() will add
                  // the string representation of the int to the end of
                  // the buffer, and return an objref to the same
                  // StringBuffer object.
25 invokevirtual #15 <Method java.lang.StringBuffer append(int)>
                  // Pop objref to StringBuffer (pushed by append()),
                  // invoke toString() on it, which returns the value
                  // of the StringBuffer as a String object. Push
                  // objref of String object.
28 invokevirtual #18 <Method java.lang.String toString()>
                  // Pop objref of String, pop objref of System.out
                  // that was pushed by the getstatic instruction at
                  // offset 9. Invoke println() on System.out,
                  // passing the String as the only parameter:
                  // System.out.println("Example6(String), width = "
                  //     + width);
31 invokevirtual #17 <Method void println(java.lang.String)>
                  // Push objref from System.out
34 getstatic #16 <Field java.io.PrintStream out>
37 aload_1        // Push objref from loc var 1 (param msg)
                  // Pop objref of String, pop objref of System.out
                  // that was pushed by the getstatic instruction at
                  // offset 37. Invoke println() on System.out,
                  // passing the String as the only parameter:
```

```
                // System.out.println(msg);
38 invokevirtual #17 <Method void println(java.lang.String)>
41 return       // Return void from <init>()
```

第三个构造方法的<init>（）方法和第二个构造方法的<init>（）方法一样，都有三个部分：一个对超类的<init>（）的调用，width初始化方法的字节码，构造方法体的字节码。第二个构造方法和第三个构造方法的区别是，第二个构造方法没有用一个明确的this（）或者super（）调用开始。因此，编译器在第二个构造方法的<init>（）方法中默认放置了一个超类的无参数<init>（）方法的调用。而第三个构造方法使用一个明确的super（）调用开始，编译器就在对应的<init>（）方法中把它转换成对应的、超类的<init>（）方法的调用。

对于除Object以外的每一个类，不管是同类的还是直接超类的，<init>（）方法都必须从另一个<init>（）方法调用开始。<init>（）方法不允许捕捉由它们所调用的<init>（）方法抛出的任何异常。比如，如果子类的<init>（）方法调用一个被意外中止的超类的<init>（）方法，那么子类的<init>（）方法也必须同样被意外中止。

7.2.2 垃圾收集和对象的终结

前面的章节中曾讲过，Java虚拟机实现必须具有某种自动堆存储管理策略——大部分是采用垃圾收集器。本章前面也讲过，程序可以明确或者隐含地为对象分配内存，但是不能明确地释放内存。但一个对象不再为程序所引用了，虚拟机必须回收（垃圾收集）那部分内存。实现可以决定何时应垃圾收集不再被引用的对象——或者决定是否根本不收集它们。并没有要求Java虚拟机实现一定要释放不再被引用的对象所占据的内存。

如果类声明了一个名为finalize（）的返回void的方法，垃圾收集器会在释放这个实例所占据的内存空间之前执行这个方法（被称为终结方法）一次。下面是一个声明了终结方法的类的例子：

```
// On CD-ROM in file classlife/ex7/Finale.java
class Finale {
    protected void finalize() {
        System.out.println("A Finale object was finalized.");
        //...
    }
    //...
}
```

因为一个终结方法是一个普通的Java方法，它可以直接被程序所调用。这样的直接调用不会影响垃圾收集器的自动调用过程。垃圾收集器（最多）只会调用一个对象的终结方法一次——在对象变成不再被引用的之后的某个时候，在占据的对象被重用之前。如果终结方法代码执行后，对象重新被引用了（复活了），随后再次变得不被引用，垃圾收集器不会第二次调用终结方法。

垃圾收集器自动调用的finalize（）方法抛出的任何异常都将被忽略。垃圾收集器可以用任意顺序调用finalize（）方法，使用任意线程，甚至并行使用多线程。第9章将描述终结过程的细节。

7.3 卸载类型

在很多方面，Java虚拟机中类的生命周期和对象的生命周期很相似。虚拟机创建并初始化对

象，使程序能使用对象，然后在对象变得不再被引用后可选地执行垃圾收集。同样，虚拟机装载、连接并初始化类，使程序能使用类，当程序不在引用它们的时候可选地卸载它们。

类的垃圾收集和卸载之所以在Java虚拟机中很重要，是因为Java程序可以在运行时通过用户自定义的类装载器装载类型来动态地扩展程序。所有被装载的类型都在方法区占据内存空间。如果Java程序持续通过用户自定义的类装载器装载类型，方法区的内存足迹就会不断增长。如果某些动态装载的类型只是临时需要，当它们不再被引用之后，占据的内存空间可以通过卸载类型而释放。

Java虚拟机通过何种方法来确定一个动态装载的类型是否仍然被程序需要呢，其判断方式与判断对象是否仍然被程序需要的方式很类似。如果程序不再引用某类型，那么这个类型就无法再对未来的计算过程产生影响。类型变成不可触及的，而且可以被垃圾收集。

使用启动类装载器装载的类型永远是可触及的，所以永远不会被卸载。只有使用用户定义的类装载器装载的类型才会变成不可触及的，从而被虚拟机回收。如果某个类型的Class实例被发现无法通过正常的垃圾收集堆触及，那么这个类型就是不可触及的。

判断动态装载的类型的Class实例在正常的垃圾收集过程中是否可以触及有两种方式。第一种，也是最明显的，如果程序保持对Class实例的明确引用，它就是可触及的。其次，如果在堆中还存在一个可触及的对象，在方法区中它的类型数据指向一个Class实例，那么这个Class实例就是可触及的。在第5章讲过，仅仅给出一个对象的引用，实现必须能够在方法区中找到对象的类的类型数据。因此，从类型数据，Java虚拟机必须能够确定对象的类、它的所有超类以及所有超接口的Class实例。图7-2是关于这种“触及”Class实例的图形化描述。

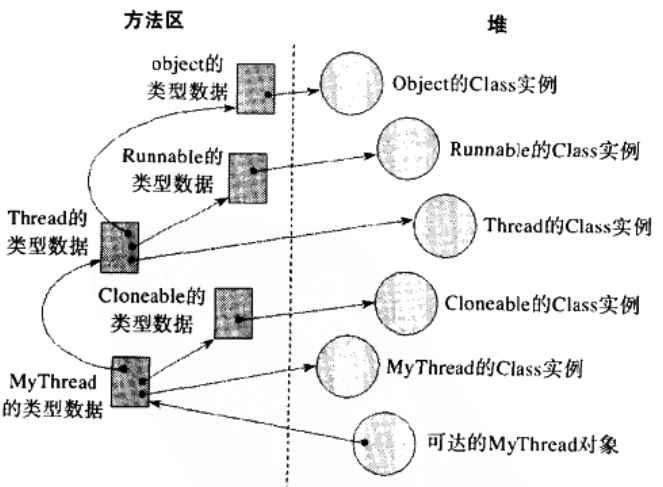


图7-2 通过可触及的对象触及Class实例

图7-2显示了垃圾收集器必须从可触及的MyThread类的对象，通过它在方法区中的类型数据找到可触及的Class实例。在该图中，堆中的对象显示为圆圈；在方法区中的类型数据显示为灰色的四边形。MyThread类有如下的声明：

```
// On CD-ROM in file classlife/ex8/MyThread.java
class MyThread extends Thread implements Cloneable {
}
```

从可触及的MyThread对象（显示在图7-2中的右下角）开始，垃圾收集器跟随一个指向MyThread的类型数据的指针，它找到了：

- 一个指向堆中的MyThread的Class实例的引用。
- 一个指向MyThread的直接超接口Cloneable的类型数据的指针。
- 一个指向MyThread的直接超类Thread的类型数据的指针。

从Cloneable的类型数据开始，垃圾收集器找到了：

- 一个指向堆中Cloneable的Class实例的引用。

从Thread的类型数据开始，垃圾收集器找到了：

- 一个指向堆中Thread的Class实例的引用。
- 一个指向Thread的直接超接口Runnable的类型数据的指针。
- 一个指向Thread的直接超类Object的类型数据的指针。

从Runnable的类型数据开始，垃圾收集器找到了：

- 一个指向堆中Runnable的Class实例的引用。

从Object的类型数据开始，垃圾收集器找到了：

- 一个指向堆中Object的Class实例的引用。

也就是说，仅仅给出一个可触及的类MyThread的实例，垃圾收集器可以“触及”MyThread和它所有超类型（包括Cloneable、Thread、Runnable和Object）的Class实例。

第8章的最后给出了一个动态装载的类变得无法触及从而被卸载的例子。

7.4 随书光盘

随书光盘的classfile目录中有本章中用到的示例源代码。

7.5 资源页

要了解更多关于Java虚拟机的信息，请访问资源页面：http://www.artima.com/insidejvm/resources。

第8章 连 接 模 型

从程序员的角度来看，理解Java体系结构最重要的方面之一就是连接模型。前几章曾讲过，Java的连接模型允许用户自行设计类装载器，这样以来就可以在运行时定制地扩展用户的程序。通过用户自定义的类装载器，你的程序可以装载在编译时并不知道或许尚未存在的类或者接口，并动态连接它们。

驱动Java连接模型的引擎是解析过程。前一章描述了类生命周期中的各个阶段，但是没有深究装载和解析的细节。这一章深入研究装载和解析的细节，并展示解析过程是如何和动态扩展相得益彰的。本章包括连接模型的概览，常量池解析，方法表，并展示了如何编写和使用类装载器，而且还给出了几个例子。

8.1 动态连接和解析

当编译一个Java程序的时候，会得到程序中每个类或者接口的独立的class文件。虽然独立的文件看上去毫无关联，实际上它们之间通过接口（harbor）符号互相联系，或者与Java API的class文件相联系。当运行程序的时候，Java虚拟机装载程序的类和接口，并且在动态连接的过程中把它们互相勾连起来。在程序运行中，Java虚拟机内部组织了一张互相连接的类和接口的网。

class文件把它所有的引用符号保存在一个地方——常量池。每一个class文件有一个常量池，每一个被Java虚拟机装载的类或者接口都有一份内部版本的常量池，被称作运行时常量池。运行时常量池是一个特定于实现的数据结构，数据结构映射到class文件中的常量池。因此，当一个类型被首次装载时，所有来自于类型的符号引用都装载到了类型的运行时常量池。

在程序运行的某些时刻，如果某个特定的符号引用将要被使用，它首先要被解析。解析过程就是根据符号引用查找到实体，再把符号引用替换成一个直接引用的过程。因为所有的符号引用都保存在常量池中，所以这个过程常被称作常量池解析。

在第6章中描述过，常量池按照一系列项组织。每一项拥有一个惟一的索引，很像数组元素。符号引用是可以出现在常量池中的一种项目。使用符号引用的Java虚拟机指令指定位于常量池中的符号引用的索引。比如，getstatic操作码（它把一个静态字段的值压入栈中）在字节码流中会跟有一个常量池索引。常量池中指定索引指向的入口，是一个CONSTANT_Fieldref_info入口，它显示出这个字段所在类的全限定名以及字段的名字和类型。

记住，Java虚拟机为每一个装载的类和接口保存一份独立的常量池。当一条指令引用常量池中的第5个元素的时候，它指向的是当前类的常量池中的第5个元素，即定义Java虚拟机当前正执行的方法的类。

来自相同或不同方法中的几条指令，可能指向同一个常量池入口，但是每一个常量池入口都只被解析一次。当符号引用被一条指令解析过后，来自其他指令的访问该符号引用的后续尝试会认为这项工作已经完成，都使用第一次解析出的直接引用结果。

连接不仅仅包括把符号引用替换成直接引用，还包括检查正确性和权限。在第7章曾介绍过，检查符号引用的存在性和访问权限（全面验证阶段的一个方面）就是在解析的时候完成的。比如，当Java虚拟机把getstatic指令解析为其他类中的字段时，Java虚拟机会检查确认是否符合下列条件：

- 那个其他类存在。
- 该类有权访问那个其他类。
- 那个其他类中存在名字相符的字段。
- 那个字段的类型和期望的类型相符（对一个字段的符号引用包含了字段类型）。
- 本类有权访问那个字段。
- 那个字段的确是一个静态类变量，而不是一个实例变量。

如果这些检查中的任何一项失败了，都会抛出一个错误，解析也就失败。否则，这个符号引用被直接引用替换，解析则成功了。

在第7章曾描述过，不同的Java虚拟机实现允许在程序执行的不同时间进行解析。实现可以选择预先解析所有的符号引用，从初始类开始，到后续的各个类，直到所有的符号引用都被解析了。在这种情形中，程序在它的main（）方法尚未被调用时就已经完全连接了，这种方法被称为早解析。另外一种方式是，实现可以在访问每一个符号引用的最后一刻才去解析它。在这种情形中，Java虚拟机只会在执行程序第一次用到这个符号引用的时候才去解析它，这种方法被称为迟解析。实现时也可以选择两种极端情况之间的折衷解析策略。

虽然Java虚拟机的实现有选择何时解析符号引用的自由，但不管怎样，都应该给外界一个迟解析的印象。对于特定的Java虚拟机来说，不管何时执行解析，都在程序执行过程中第一次实际试图访问一个符号引用的时候才抛出错误。用这种方式，对用户来说看上去都是迟解析。如果Java虚拟机使用早解析，在早解析的过程中发现某个class文件无法找到，它不会抛出对应的错误，直到后来程序实际访问这个class文件中的某些东西时才抛出错误。如果程序不使用这个类，错误永远不会被抛出。

8.1.1 解析和动态扩展

除了简单地在运行时连接类型之外，Java程序还可以在运行时决定连接哪一个类型。Java的体系结构允许动态扩展Java程序，这个过程包括运行时决定所使用的类型，装载它们，使用它们。通过传递类型的名字到java.lang.Class的forName（）方法，或者用户自定义的类装载器的loadClass（）方法，可以动态扩展Java程序。用户自定义的类装载器可以从java.lang.ClassLoader的任何子类创建。两种方法都可以使运行中的程序去调用在源代码中未曾提及的、而是在程序运行中决定的类型。动态扩展的例子如支持Java的Web浏览器，它跨网络装载applet的class文件。当浏览器启动的时候，它不知道将要从网络上装载什么class文件，当它遇到包含这些applet的网页的时候才知道每个applet所需的类和接口的名字。

动态扩展Java程序最直接的方式就是使用java.lang.Class的forName（）方法，它有两种重载的形式：

```
// A method declared in class java.lang.Class:
public static Class forName(String className)
```

```
    throws ClassNotFoundException;
public static Class forName(String className, boolean initialize,
    ClassLoader loader)
    throws ClassNotFoundException;
```

forName ( ) 的三参数形式是在1.2版中加入的，将类型的全限定名装入String类型的className参数。如果boolean类型的initialize参数为true，类型会在forName ( ) 方法返回之前连接并初始化；如果initialize参数是false，类型会被装载，可能会被连接但是不会被forName ( ) 方法明确地初始化。然而，如果该类型在调用forName ( ) 之前已经被初始化了，即使将false作为第二个参数传递到forName ( )，返回的类型也已经被初始化了。第三个参数为ClassLoader loader，传递一个用户定制的类装载器的引用给forName ( )，让其使用这个类装载器来请求类型。也可以指定forName ( ) 用默认的启动类装载器来请求类型，只需传递null作为ClassLoader loader参数。forName ( ) 还有一个只采用一个参数的版本（该参数即为要装载的类型的全限定名），它总是使用当前的类装载器（就是装载执行forName ( ) 请求的类的类装载器），并且总是初始化该类型。两个版本的forName ( ) 方法都返回Class实例的引用，它代表被装载的类型。如果类型无法被装载，会抛出ClassNotFoundException异常。

动态扩展Java程序的另外一种方式就是使用用户自定义类装载器的loadClass ( ) 方法。如果需要用自定义的类装载器请求类型，只需调用那个类装载器的loadClass ( ) 方法。类ClassLoader包含两个名为loadClass ( ) 的重载方法，其形式如下：

```
// A method declared in class java.lang.ClassLoader:
protected Class loadClass(String name)
    throws ClassNotFoundException;
protected Class loadClass(String name, boolean resolve)
    throws ClassNotFoundException;
```

两个loadClass ( ) 方法都接受装载类型的全限定名装入String类型的name参数。loadClass ( ) 的语义和forName ( ) 是一样的。如果loadClass ( ) 方法已经用String类型的name参数传递的全限定名装载了类型，它会返回这个已经被装载的类型的Class实例。否则，该方法会试图用某种用户定制的方式来装载请求的类型（用户定制的方式取决于用户自定义类装载器的作者）。如果类装载器用定制的方式成功地装载了类型，loadClass ( ) 应该返回一个Class的实例，表示新近装载的类型。否则，方法将抛出ClassNotFoundException异常。如何编写用户自定义的类装载器后面会详细介绍。

双参数版本的loadClass ( ) 中，boolean类型的resolve参数表示是否在装载时执行该类型的连接。前几章曾提到过，连接过程包含三个步骤：校验被装载的类型，准备（包括为类型分配内存），解析类型中的符号引用，其中第三步是可选的。如果resolve参数为true，loadClass ( ) 方法会确保在方法返回某个类型的Class实例之前已经装载并连接了该类型。如果resolve参数是false，loadClass ( ) 方法仅仅去试图装载请求的类型，而不关心类型是否被连接了。因为Java虚拟机的规范中对实现何时进行连接给了一定的自由，当传递false给resolve参数时，从loadClass ( ) 方法得到的类型可能被连接了，也可能没有。双参数版本的loadClass ( ) 是一个过时的方法，实际上从Java 1.1开始，resolve参数就没有作用了。通常，应该调用单参数版本的loadClass ( )

方法，它和使用resolve参数取false值的双参数版本是等价的。当调用单参数版本的loadClass（）时，它会试图装载类型并返回，而把连接和初始化类型的时机留给虚拟机去掌握。

使用forName（）还是调用用户自定义类装载器的loadClass（）方法取决于用户的需要。如果没有特别要使用类装载器的要求，或许应该用forName（），因为forName（）是动态扩展最直接的方法。另外，如果需要请求的类型在装载时就初始化（并且连接）的话，则不得不使用forName（）。当loadClass（）方法返回类型的时候，类型有可能没有被连接。当调用单参数版本的forName（）方法或者调用它的三参数版本并且传递true作为initialize参数的值时，返回的类型一定已经被连接、初始化过了。

初始化是很重要的。比如JDBC驱动程序通常用forName（）调用装载的。因为每一个JDBC驱动程序类的静态初始化方法都用DriverManager注册驱动程序，这样才能被应用程序所使用，驱动程序类必须被初始化，而不是仅仅被加载。假若一个驱动程序被装载了，但是没有初始化，那么类的静态初始化方法就无法被执行，驱动程序就没有在DriverManager中被注册，驱动程序就无法被应用程序使用。使用forName（）来装载驱动程序可以确保类被初始化，就可以确保forName（）返回后应用程序就可以使用这个驱动程序了。

然而，类装载器可以满足一些forName（）无法满足的需求。如果需要一些特定的装载类型的方法，比如从网络上下载，从数据库中取出，从加密文件中提取，甚至动态地创建它们，这时就需要一个类装载器。创建用户自定义的类装载器，其中一个重要原因就是能够以定制方式把类型的全限定名转换成一个Java class文件格式（它定义了命名的类型）的字节数组。使用类装载器而非forName（）的其他理由和安全性相关。第3章提到过，每一个类装载器拥有一个独立的命名空间，这就为在不同的命名空间中装载的类型提供了一层安全防护。可以编写一个Java程序，类型无法看见不在同一命名空间装载的其他类型。第3章还讲过，类装载器负责把装载的代码放到保护域中，也就是说，如果安全上需要包含一种定制方式把类型装载到保护域中，就需要使用类装载器而非forName（）。

不管是动态扩展的整体过程，还是各个类装载器使用的单独命名空间，都是解析支持的一个方面：在虚拟机解析符号引用时，它可以选择类装载器。当解析常量池中的入口需要装载类型的时候，虚拟机使用装载引用类型的同一个类装载器来装载所需的类型。比如，想像一个Cat类通过常量池符号引用一个Mouse类型。假设Cat是被一个用户自定义的类装载器装载的，当虚拟机解析指向Mouse的引用时，先检查是否Mouse已经被装载到Cat所属的命名空间中（检查装载Cat的装载器在装载Cat之前已经装载了名为Mouse的类型）。如果没有，虚拟机使用装载Cat的同一个类装载器来请求Mouse。就算类名Mouse已经被装载到另外一个命名空间，这一点仍然成立。使用启动类装载器装载的类型，当它的符号引用被解析时，Java虚拟机也使用启动类装载器来装载被引用的类型。使用用户自定义的类装载器装载的类型，当它的符号引用被解析时，Java虚拟机也使用同一个用户自定义的类装载器来装载被引用的类型。

8.1.2 类装载器与双亲委派模型

在第3章曾讲过，1.2版本中引入了类装载器的形式化双亲委派模型。虽然老式版本（即1.2版本之前）编写的类装载器无法享受双亲委派模型的好处，但仍然可以在1.2版本中使用，1.2版本及更高版本中推荐使用双亲委派模型创建类装载器。1.2版本中每一个用户自定义的类装载器

在创建时被分配一个“双亲”类装载器。如果没有显式地传递一个双亲类装载器给用户自定义的类装载器的构造方法，系统类装载器就默认被指定为双亲。或者，在调用用户自定义的新类装载器的时候，双亲装载器可以被显式地传递过去。如果传递到构造方法的是一个已有的用户自定义类装载器的引用，该用户自定义装载器就作为双亲。如果向构造方法传递了null，启动类装载器就是双亲。

为了更好地理解双亲委派模型，假设一个Java程序创建了一个名为“Grandma”的自定义类装载器。因为程序传递了null到Grandma的构造方法，Grandma的双亲就是启动类装载器。过了一段时间，程序创建了另一个名为“Mom”的类装载器。因为程序传递了Grandma的引用到Mom的构造方法，Mom的双亲被设成是一个自定义的类装载器，指向Grandma。又过了一段时间，程序创建了一个新的类装载器“Cindy”。因为应用程序传递了指向Mom的引用到Cindy的构造方法，Cindy的双亲就被设定为用户自定义的Mom类装载器。

现在假设程序要求Cindy去装载一个名为java.io.FileReader的类型。当符合双亲委派模型的类装载一个类型的时候，它首先委派它的双亲——请求它的双亲试着装载这个类型，它的双亲再依次调用各自的双亲。这个委派的过程一直进行到委派链的末端，一般来说应该是启动类装载器。因此，Cindy第一件事就是去找Mom来装载那个类型；Mom所做的第一件事就是去找Grandma来装载那个类型；而Grandma首先是找启动类装载器去装载。在这个例子中，启动类装载器可以装载（或者已经装载了）那个类型，它就返回代表java.io.FileReader的Class实例给Grandma。Grandma传递该Class的引用回Mom，Mom再回传给Cindy，Cindy返回给程序。

请注意，在类装载器之间具有了委派关系，首先发起装载要求的类装载器不必是定义该类型的装载器。在上面的例子中，程序首先要求Cindy去装载类型，但最终却是启动类装载器定义了那个类型。在Java术语中，要求某个类装载器去装载一个类型，但是却返回了其他类装载器装载的类型，这种装载器被称为是那个类型的初始类装载器；而实际定义那个类型的类装载器被称为该类型的定义类装载器。因此在上一个例子中，java.io.FileReader定义类装载器是系统类装载器。Cindy是初始类装载器，但是Mom、Grandma甚至系统类装载器也是初始类装载器。任何被要求装载类型，并且能够返回Class实例的引用代表这个类型的类装载器，都是这个类型的初始类装载器。

再比如，假设程序要求Cindy装载一个名为com.artima.knitting.QuiltPattern的类型。Cindy委派给Mom，Mom委派给Grandma，Grandma委派给系统类装载器。然而在这个例子中，系统类装载无法装载这个类型，所以控制权回到Grandma，Grandma试图去用自定义的方法去装载类型。因为Grandma负责装载标准扩展，而com.artima.knitting包正确地以JAR文件格式被安装到了标准扩展目录，Grandma能够装载这个类型。Grandma定义了该类型，并返回了代表com.artima.knitting.QuiltPattern的Class实例给Mom。Mom传递Class的引用给Cindy，Cindy返回给程序。在这个例子中，Mom是com.artima.knitting.QuiltPattern类型的定义类装载器，Cindy、Mom和Grandma——但是不包括启动类装载器——是该类型的初始类装载器。

8.1.3 常量池解析

本节描述解析每一种常量池入口类型的细节，包括可能在解析过程中抛出的错误。如果在解析过程中抛出了错误，错误被看成是由指向执行解析的常量池入口的引用者抛出的。除了这

里描述的错误，触发常量池入口解析的不同的指令，可能导致抛出其他的错误。比如，getstatic导致CONSTANT_Fieldref_info入口被解析。如果这个入口被成功解析，虚拟机执行一个附加的检查：确认字段是否的确是静态的（即，是类变量而非实例变量）。如果字段不是静态的，虚拟机抛出一个错误。本节所描述的抛出错误，以及其他可能在解析过程中抛出的错误，都可以在附录A中按照每个指令找到。

在以下的部分，术语“当前类装载器”指的是一个定义类装载器，不管它是一个用户自定义的类装载器，还是一个启动类装载器，它的常量池包含正被解析的符号引用。术语“当前命名空间”表示当前类装载器的命名空间，是由所有认为当前类转装载器是自己的初始类装载器的类型名字组成的。

8.1.4 解析CONSTANT_Class_info入口

在所有的常量池入口类型中，解析起来最复杂的就是CONSTANT_Class_info了。这种入口类型用来表示指向类（包括数组类）和接口的符号引用。有几个指令，比如new 和anewarray，直接使用CONSTANT_Class_info入口。其他的指令，比如putfield或者invokevirtual，从其他类型的入口间接指向CONSTANT_Class_info。举个例子来说，putfield指令使用CONSTANT_Fieldref_info入口。CONSTANT_Fieldref_info的class_index项目则包含指向CONSTANT_Class_info入口的常量池索引。

根据类型是否是数组，或者引用的类型（包含正在被解析的CONSTANT_Class_info入口的常量池的类型）是由启动类装载器还是由用户自定义的类装载器装载的，解析CONSTANT_Class_info入口的细节会有所不同。

1. 数组类

如果一个CONSTANT_Class_info入口的name_index项指向的CONSTANT_Utf8_info字符串是由一个左方括号开始的，比如“[I”，那么它指向的是一个数组类。在第6章中描述过，内部使用的数组名字的每一维使用一个左方括号，然后是元素的类型。如果元素类型由一个“L”开头，比如“Ljava.lang.Integer;”，那么数组是一个关于引用的数组；否则，元素类型是一个基本类型，比如“I”表示int，“D”表示double，数组就是一个基本类型组成的数组。

指向数组类的符号引用的最终解析结果是一个Class实例，表示该数组类。如果当前类装载器已经被记录为被解析的数组类的初始装载器，就使用同样的类。否则，虚拟机执行下列步骤：如果数组的元素类型是一个引用类型（数组是一个关于引用的数组），虚拟机用当前类装载器解析元素类型。举例来说，如果解析名为“[[Ljava.lang.Integer”的数组类，虚拟机会确认java.lang.Integer被装载到当前类装载器的命名空间中。如果数组是关于基本类型的数组，那么虚拟机立即就会创建关于那个元素类型的新数组类，维数也在此时确定，然后创建一个Class的实例来代表这个类型。如果数组是关于引用的数组，那么这一步骤发生在解析了元素类型之后。如果是关于引用的数组，数组会被标记为是由定义它的元素类型的类装载器定义的。如果是关于基本类型的数组，数组类会被标记为是由启动类装载器定义的。

2. 非数组类和接口

如果CONSTANT_Class_info入口的name_index项指向一个并非由左方括号开始的CONSTANT_Utf8_info字符串，那么这是一个指向非数组类或者接口的符号引用。解析这种类

型的符号引用分为多步。

要解析任何指向非数组类或者接口的符号引用（任何CONSTANT_Class_info入口），Java虚拟机都要执行相同的基本步骤。下面我们用步骤1a和步骤1b来说明。在步骤1a，类型被装载。在步骤1b，检查访问类型的权限。虚拟机执行步骤1a的精确方式取决于该引用类型是被启动类装载器装载，还是被用户自定义的类装载器装载。

这一部分还要说明步骤2a到步骤2d，它们描述了如何连接和初始化新解析的类型。对于将要被连接和初始化的类型来说，这些步骤并不是解析的一部分。解析非数组类或者接口只包含步骤1a和步骤1b，（潜在的）装载类并检查它的访问权限。然而如果解析符号引用到类型是由第一次使用这个类型触发的，在解析类型的符号引用之后，立即开始连接并初始化这个类型。因为Java虚拟机的实现允许进行早期解析，无论如何，解析类型的引用可能远早于连接和初始化这些类型。第7章曾提到，初始化（这是步骤2d）在第一次使用这个类型时激活。在类型可以被初始化之前，它首先要被连接（步骤2a到步骤2c），而在它能够被连接之前，它必须被装载（步骤1a）。

步骤1a：装载类型或者任何超类型

解析非数组类或者接口的基本要求是确认类型被装载到了当前命名空间。作为第一步，虚拟机必须确定是否被引用的类型已经被装载进了当前命名空间。为了做出决定，虚拟机必须查明是否当前类装载器被标记为该类型的初始装载器（该类型包含所需的全限定名，是由被解析的符号引用给出）。对于每一个类装载器，Java虚拟机维护一张列表，其中记录了所有其类装载器是一个初始类装载器的类型的名字。每一张这样的列表就组成了Java虚拟机内部的命名空间。在解析过程中，虚拟机使用这些列表来决定是否一个类已经被一个特定的类装载器装载过了。如果虚拟机发现希望装载的全限定名已经在当前命名空间被列出了，它将只使用已经被装载的类型，该类型由方法区的类型数据块所定义，并由堆中相关的Class实例所表示。首先检查当前命名空间是否已经包含了希望装载的全限定名，虚拟机保证每一个类装载器都只装载一个给定名字的类型。

如果希望装载的类型还没有被装载进当前命名空间，虚拟机把类型的全限定名传递给当前类装载器。Java虚拟机总是要求当前类装载器，就是发起引用的类型的定义类装载器，也就是运行时常量池包含正在被解析的CONSTANT_Class_info入口的类装载器，来试图装载被引用的类型。如果发起引用的类型是被启动类装载器定义的，虚拟机要求启动类装载器调用被引用的类型。否则，发起引用的类型是被一个用户自定义的类装载器定义的，虚拟机就要求同一个类装载器来装载被引用的类型。

如果当前类装载器就是启动类装载器，虚拟机根据不同的实现使用不同的方式装载类型。如果当前类装载器是一个用户自定义的类装载器，Java虚拟机通过调用用户自定义类装载器的loadClass（）方法来完成装载请求，把需要装载的类型的全限定名作为参数传递进去。

装载类型时，不管是请求启动类装载器还是用户自定义的类装载器，都有两个选择：类装载器可以选择自行装载这个类型，或者委派其他的类装载器完成这个工作。用户自定义的类装载器可能要求另一个用户自定义的类装载器或者启动类装载器来试着装载这个类型；而启动类装载器可能要求一个用户自定义的类装载器来试着装载这个类型。

若要委派给一个用户自定义的类装载器，类装载器（不管是启动类装载器或者是用户自定义的类装载器）调用被委派的类装载器的loadClass（）方法，把需要装载的类型的全限定名作为参数来传递。若要委派给启动类装载器，一个用户自定义类装载器调用java.lang.ClassLoader的一个静态方法findSystemClass（），把需要装载的类型的全限定名作为参数来传递。被委派的类装载器可以决定是否自行装载该类型，或者委派给另一个类装载器。最终，某个类装载器决定到此为止了，不需要再委派了，然后自己去装载类型。如果装载成功，那么这个类装载器就被标记为该类型的定义类装载器。在这个过程中涉及的所有类装载器——定义类装载器和所有产生委派的类装载器——都被标记为该类型的初始装载器。

考虑到本章前面提到的双亲委派模型的存在，如果一个用户自定义的类装载器产生委派，它委派的往往是在双亲委派模型中的双亲。双亲再委派给它的双亲，以此类推。委派的过程一直进行到委派的末端，有一个类装载器不再委派，而是决定装载这个类型。大多数情况下，末端的类装载器就是启动类装载器。如果一个双亲类装载器试图装载这个类型但是却失败了，控制权重新回到子装载器。在双亲委派模型中，子装载器在得知它的双亲（包括祖父，曾祖父……）无法装载此类型时，它会试图自行装载。如果委派链中的某个类装载器第一个成功地装载了类型，那么这个类装载器就会被标记为定义类装载器。这个类装载器以及所有在委派链中排在它前面的类装载器会被标记为初始类装载器。然而，它的双亲、祖父、曾祖父，以及更上一代，他们没有一个成功装载了这个类型，所以不会被标记为类型的初始类装载器。

如果用户自定义类装载器的loadClass（）方法能够找到或者产生一个字节数组，用Java class文件格式描述了该类型，loadClass（）必须调用defineClass（），把类型全限定名和指向那个字节数组的引用传递进去。调用defineClass（）方法会使得虚拟机试图解析二进制数据，变为方法区中的内部数据结构。这时候虚拟机会进行一项在第3章描述过的检查，要保证传递来的字节数组按照Java class文件格式的基本结构组织。Java虚拟机用传递进来的全限定名来校验，需要装载的类型名字是否就是传递进来的字节数组定义的类型名字。

一旦被引用的类型被装载了，虚拟机仔细检查它的二进制数据。如果类型是一个类，并且不是java.lang.Object，虚拟机根据类的数据得到它的直接超类的全限定名。虚拟机接着会察看超类是否已经被装载进当前命名空间了。如果没有，先装载超类。一旦超类被装载了，虚拟机再次仔细检查它的二进制数据来找到它的超类。一直重复到超类为Object为止。

当虚拟机调用超类的时候，它实际上只是解析另外一个符号引用。为了确定一个类的超类的全限定名，虚拟机察看class文件的super_class域。这个域给出了一个CONSTANT_Class_info常量池入口的索引，作为指向类的超类的符号引用。但虚拟机装载超类的时候，它对超类的符号引用执行解析步骤1a。从而，作为解析CONSTANT_Class_info入口过程步骤1a的一部分，虚拟机递归地在每一个超类上应用解析CONSTANT_Class_info入口的过程，直到最终遇到Object。

在从Object返回的路上，虚拟机再次仔细检查每个类型的数据，看它们是否直接实现了任何接口。如果是这样，它会先确保那些接口也被装载了。对于每一个虚拟机装载的接口，虚拟机检查它们的类型数据，看它们是否直接扩展了任何其他接口。如果是这样，虚拟机会确认那些超接口也被装载了。

当虚拟机装载超接口时，它再次解析更多的CONSTANT_Class_info入口。正在被装载的类

型直接实现或者扩展的接口保存在class文件的interfaces元素中，它实际保存的是作为符号引用的一些常量池入口。当虚拟机装载超接口时，它解析interfaces元素中指定的CONSTANT_Class_info入口，递归地应用CONSTANT_Class_info入口的解析过程。

当虚拟机递归地在超类和超接口上应用解析过程时，它使用发起引用的子类型的定义类装载器。虚拟机来用通常的方式做出请求，这是通过调用发起引用的子类型的定义类装载器的loadClass（）方法实现的，把需要装载的直接超类或者直接超接口的全限定名作为参数传递进去。

一旦一个类型被装载进入了当前命名空间，而且通过递归，所有该类型的超类和超接口也都被成功装载了，虚拟机就会创建新的Class实例来代表这个类型。如果定义类型的字节是由用户自定义的类装载器确定或者生成，然后传递到defineClass（）方法，defineClass（）会在这个时候返回这个新的Class实例。或者，如果用户自定义的类装载器通过findSystemClass（）调用委派启动类装载器来装载，findSystemClass（）会在这个时候返回Class实例。直到从defineClass（）方法或者findSystemClass（）方法接收到了Class实例，loadClass（）方法才会返回这个Class实例给它的调用者。如果用户自定义的类装载器委派给了另一个用户自定义的类装载器，那么，当被委派的用户自定义类装载器的loadClass（）方法返回时，它会收到Class实例。直到从被委派的类装载器收到Class实例，发起委派的类装载器才会从它自己的loadClass（）方法返回这个实例。

通过步骤1a，Java虚拟机确认某个类型是否被装载了，如果这个类型是一个类，确保它的所有超类都被装载了。不管这个类型是类还是接口，Java虚拟机确保它的所有超接口也都被装载了。在这个步骤中，这些类型没有被连接或者初始化——仅仅是装载。

在步骤1a，虚拟机可能抛出如下错误：

- 如果虚拟机直接调用启动类装载器（而不是通过一次findSystemClass（）调用），而它无法确定或者生成所请求类型的二进制数据，虚拟机抛出NoClassDefFoundError。
- 如果用户自定义的类装载器通过一次findSystemClass（）调用委派给启动类装载器，而启动类装载器无法确定或者生成所请求的二进制数据，findSystemclass（）方法产生一个ClassNotFoundError中断。同样，如果用户自定义的类装载器通过loadClass（）调用委派给另一个用户自定义的类装载器，并且用户自定义的类装载器无法确定或者生成所请求类型的二进制数据，它的loadClass（）方法产生ClassNotFoundError中断。
- 如果二进制数据被启动类装载器确定或者生成了，但是它的结构不正确，虚拟机抛出ClassFormatError异常。同样，如果用户自定义的类装载器能够确定或者生成二进制数据，并且调用了defineClass（）方法，但是defineClass（）方法发现二进制数据并非合适的结构，defineClass（）会产生一个ClassFormatError中断。
- 如果二进制数据被生成了，但是版本号无法识别（比如Java class文件的主版本号或者次版本号太高），虚拟机抛出UnsupportedClassVersionError异常。
- 如果二进制数据被生成了而且组织良好，但是在类或者接口之后跟着的并非是所需的名字（比如文件CuteKitty.class被发现包含名为HungryTiger的类而非CuteKitty），虚拟机抛出NoClassDefFoundError异常。
- 如果组织良好的二进制数据被传递给defineClass（），但是包含的类或者接口的名字已经

在当前类装载器的命名空间中存在，defineClass（）方法产生一个LinkageError中断。

- 如果类不包含一个超类，并且自己也不是Object类本身，虚拟机抛出ClassFormatError异常（注意这个检查在装载这一步完成，因为虚拟机在这一步需要一部分信息——指向超类的符号索引。在步骤1，虚拟机必须递归地装载所有的超类）。
- 如果一个类看上去是它自己的超类，或者一个接口作为自己的接口，虚拟机抛出ClassCircularityError异常。
- 如果类型引用的超类其实是个接口，或者引用的超接口其实是个类，虚拟机抛出IncompatibleClassChangeError异常。

步骤1b：检查访问权限

随着装载结束，虚拟机检查访问权限。如果发起引用的类型没有访问被引用的类型的权限，虚拟机抛出IllegalAccessError异常。逻辑上说，步骤1b是校验的一部分，但是并非在正式校验阶段完成。检查访问权限总是在步骤1a之后，以确保符号引用指向的类型被装载进正确的命名空间，这是解析符号引用的一部分。一旦检查结束，步骤1b以及整个解析CONSTANT_Class_info入口的过程就结束了。

如果步骤1a或者1b发生了错误，符号引用解析就失败了。但是如果在步骤1b权限检查之前一切正常，这个类总体上来说还是可以使用的，只不过不能被发起引用的类型使用。如果错误在检查权限之前抛出，类型是不可使用的，必须被标记为不可使用或者被取消。

步骤2：连接并初始化类型和任何超类

在这个时候，被解析的、被CONSTANT_Class_info入口引用的类型已经被装载了，但是还没有进行必要的连接和初始化。类型所有超类和超接口也被装载了，但是也没有进行必要的连接和初始化。某些超类型可能已经被初始化了，因为它们可能是在早期解析过程中被初始化的。

在第7章中讲过，超类必须在子类之前被初始化。如果虚拟机因为主动使用一个类而正在解析该类（不是接口）的引用，它必须确认它的所有超类都被初始化了，从Object开始沿着继承的结构向下处理，直到被引用的类（请注意这和步骤1a装载的顺序是相反的）。如果一个类型还没有被连接，在初始化之前必须被连接。请注意只有超类必须被初始化，超接口是不必的。

步骤2a：校验类型

步骤2从第7章描述的正式连接校验阶段开始。第7章曾讲过，校验过程可能要求虚拟机装载新的类型来确认字节码符合Java语言的语义。比如，如果一个指向特定类的实例的引用被赋给一个变量，而该变量被声明为不同的类类型，虚拟机可能不得不装载这两种类型，以确认其中一个是另一个的子类。其他的类可能被装载，甚至被连接了，但是肯定不会被初始化。

如果在校验阶段Java虚拟机遇到了麻烦，它会抛出VerifyError异常。

步骤2b：准备类型

随着正式校验阶段的结束，类型必须被准备好。在第7章描述过，在准备阶段虚拟机为类变量以及随实现不同而有差别的数据结构（比如方法表）分配内存。

步骤2c：可选的步骤，解析类型

在这时候，类型已经被装载、校验了，也准备好了。按照第7章所介绍的，虚拟机的实现可以可选地在这时候解析类型。记住，我们现在是在解析过程中的一个阶段，关于一个被引用的

类型，步骤1a、2a、2b已经解析了发起引用的类型的常量池的CONSTANT_Class_info入口。步骤2c是关于被引用类型（而非发起引用的类型）中所包含的符号引用的解析。（顺便说一句，步骤2b之前没有被提到被引用类型是因为步骤2b与被引用的类型的装载、连接和初始化过程毫无关系。步骤2b实际上是发起引用的类型的连接阶段的4次校验中的一部分，发起引用的类型是指包含指向被引用类型的符号引用的类型。）

举个例子，假若虚拟机正在解析一个从Cat类指向Mouse类的符号引用，虚拟机为Mouse类执行了步骤1a、2a和2b。在从Cat类的常量池中解析指向Mouse的符号引用时，虚拟机可能可选地（作为步骤2c）解析Mouse类的常量池中的所有符号引用。比如Mouse的常量池中包含一个指向Cheese类的符号引用，虚拟机这时候可能装载并可选地连接（但并非初始化）Cheese类。虚拟机不能在这儿试图去初始化Cheese，因为Cheese没有被主动使用。（当然，Cheese可能实际上早就被装载了，因为在别的地方被主动使用过了。所以可能已经在这个命名空间中被装载、连接并且初始化过了。）

在本章的前面部分提到过，如果一个实现在解析过程的这个时刻执行步骤2c（早期解析），它必须在这个符号引用被运行的程序实际使用之前不报告任何错误。比如，如果在解析Mouse类的常量池的过程中，虚拟机无法找到Cheese类，那么它也不会抛出NoClassDefFound错误，除非Cheese类被程序实际使用。

步骤2d：初始化类型

在这个时刻，类型已经被装载、校验、准备好了，可能也可选地被解析了。经过漫长的过程，类型终于准备好进行初始化了。按照第7章中定义的，初始化包括两个步骤。如果类型拥有任何超类，初始化类型的超类是按照自顶向下的顺序进行的。如果类型有一个类初始化方法，那也在此时执行。如果类拥有类初始化方法，恰好是在步骤2d执行它。因为步骤2d在所有被引用的类型的超类上是按照自顶向下的顺序执行的，所以步骤2d会先在超类上执行，而后在子类上执行。

如果类初始化方法随着抛出某个非Error子类的异常而终止，虚拟机抛出ExceptionInInitializerError异常，把抛出的异常作为构造方法的参数。否则，如果抛出的异常是Error的子类，虚拟机就抛出那个错误。如果虚拟机因为内存不足而无法创建新的ExceptionInInitializerError，它抛出OutOfMemoryError异常。

8.1.5 解析CONSTANT_Fieldref_info入口

要解析类型是CONSTANT_Fieldref_info的常量池入口，虚拟机必须首先解析class_index项中指明的CONSTANT_Class_info入口。因此，解析CONSTANT_Fieldref_info时可能抛出任何因解析CONSTANT_Class_info而抛出的错误。如果CONSTANT_Class_info解析成功，虚拟机在此类型和它的超类型上搜索所需要的字段。如果找到了需要的字段，虚拟机要检查当前类是否拥有访问这个字段的权限。

如果解析CONSTANT_Class_info成功完成，虚拟机按照如下步骤执行字段搜索过程：

1）虚拟机在被引用的类型中查找具有指定的名字和类型的字段。如果虚拟机找到了这样一个字段，这个字段就是成功的字段搜索结果。

2）否则，虚拟机检查类型直接实现或扩展的接口，以及递归地检查它们的超接口。如果找

到了名字和类型都符合的字段，这个字段就是成功的字段搜索结果。

3) 否则，如果类型拥有一个直接的超类，虚拟机检查类型的直接超类，并且递归地检查类型的所有超类。如果找到了名字和类型都符合的字段，这个字段就是成功的字段搜索结果。

4) 否则，字段搜索失败。

如果虚拟机在被引用的类或者任何它的超类型中没有找到名字和类型都符合的字段（字段搜索失败），虚拟机就抛出NoSuchFieldError异常。否则，如果字段搜索成功了，但是当前的类没有权限去访问该字段，虚拟机就抛出IllegalAccessError异常。

否则，虚拟机把这个入口标记为已解析，并在这个常量池入口的数据中放上指向这个字段的直接引用。

8.1.6 解析CONSTANT_Methodref_info入口

要解析CONSTANT_Methodref_info类型的常量池入口，虚拟机必须先解析class_index项中指定的CONSTANT_Class_info入口。也就是说，所有CONSTANT_Class_info解析过程中可能抛出的错误在解析CONSTANT_Methodref_info的过程中都可能被抛出。如果解析CONSTANT_Class_info成功，虚拟机在类型和它的超类型中搜索指定的方法。如果找到了指定的方法，虚拟机检查当前类是否有权限去访问这个方法。

如果解析CONSTANT_Class_info成功完成，虚拟机使用如下步骤来执行方法解析：

1) 如果被解析的类型是一个接口，而非类，虚拟机抛出IncompatibleClassChangeError异常。

2) 否则，被解析的类型是一个类。虚拟机检查被引用的类是否有一个方法符合指定的名字以及描述符。如果虚拟机找到了这样的一个方法，该方法就是成功的方法搜索结果。

3) 否则，如果类有一个直接超类，虚拟机检查类的直接超类，并且递归地检查类的所有超类，查看是否有方法符合指定的名字和描述符。如果虚拟机找到了这样的一个方法，该方法就是成功的方法搜索结果。

4) 否则，虚拟机检查是否这个类直接实现了任何接口，并且递归地检查由类型直接实现的接口的超接口，察看是否有方法符合指定的名字和描述符。如果虚拟机找到了这样的一个方法，该方法就是成功的方法搜索结果。

5) 否则，方法搜索失败。

如果虚拟机没有在被引用的类和它的任何超类型中找到名字、返回类型、参数数量和类型都符合的方法（方法搜索失败），虚拟机抛出NoSuchMethodError异常。否则，如果方法存在，但是方法是一个抽象方法，虚拟机抛出AbstractMethodError异常。否则，如果方法存在，但是当前类没有访问方法的权限，虚拟机抛出IllegalAccessError异常。

否则，虚拟机把这个入口标记为已解析。并在这个常量池入口的数据中放上指向该方法的直接引用。

8.1.7 解析CONSTANT_InterfaceMethodref_info入口

要解析类型为CONSTANT_InterfaceMethodref_info的常量池入口，虚拟机首先要解析class_index项中指定的CONSTANT_Class_info入口。所以，任何在解析CONSTANT_Class_info过程中可能抛出的错误，在解析CONSTANT_InterfaceMethodref_info过程中都有可能抛出。如果解析CONSTANT_Class_info成功，虚拟机在接口和它的超类型中搜索指定的方法。（虚拟机不

需要确认当前类型由访问方法的权限，因为接口中定义的所有方法都是隐含公开的。）

如果解析CONSTANT_Class_info成功完成了，虚拟机按照如下步骤来执行接口方法解析：

1) 如果被解析的类型是一个类，而非接口，虚拟机抛出IncompatibleClassChangeError异常。

2) 否则，被解析的类型是一个接口。虚拟机检查被引用的接口是否有方法符合指定的名字和描述符。如果发现了这样的一个方法，该方法就是成功的接口方法搜索结果。

3) 否则，虚拟机检查类（此处怀疑为原书错误？应该是接口。——译者注）的直接超接口，并且递归地检查接口的所有超接口以及java.lang.Object类来查找符合指定名字和描述符的方法。如果发现了这样的一个方法，该方法就是成功的接口方法搜索结果。

4) 如果虚拟机没有在被引用的接口和它的任何超类型中找到名字、返回类型、参数的数量和类型都符合的方法，虚拟机抛出NoSuchMethodError异常。

否则，虚拟机把这个入口标记为已解析，并且在这个常量池入口的数据中放上指向这个方法的直接引用。

8.1.8 解析CONSTANT_String_info入口

要解析类型是CONSTANT_String_info的入口，Java虚拟机必须把一个指向内部字符串对象的引用放置到要被解析的常量池入口数据中去。该字符串对象（java.lang.String类的实例）必须按照string_index项在CONSTANT_String_info中指明的CONSTANT_Utf8_info入口所指定的字符顺序组织。

每一个Java虚拟机必须维护一张内部列表，它列出了所有在运行程序的过程中已被“拘留（intern）”的字符串对象的引用。基本上，如果一个字符串在虚拟机的拘留列表上出现，就说它是被拘留的。维护这个列表的关键是任何特定的字符序列在这个列表上只出现一次。

要拘留CONSTANT_String_info入口所代表的字符序列，虚拟机要检查内部拘留名单上这个字符序列是否已经在编了。如果已经在编，虚拟机使用指向以前拘留的字符串对象的引用。否则，虚拟机按照这个字符序列创建一个新的字符串对象，并把这个对象的引用编入列表。要完成CONSTANT_String_info入口的解析过程，虚拟机应把指向被拘留的字符串对象的引用放置到被解析的常量表入口数据中去。

在Java程序中，可以调用String类的intern（）方法来拘留一个字符串。所有字面上表达的字符串都在解析CONSTANT_String_info入口的过程中被拘留了。如果具有相同序列的Unicode字符串已经被拘留过，intern（）方法返回一个指向相符的已经被拘留的字符串对象的应用。如果字符串对象的intern（）方法被调用（该字符串对象包含的字符序列还没有被拘留过），那么这个对象本身就被拘留。对象的intern（）在被调用的时候将返回指向同一个字符串对象的引用。

这里有个例子：

```
// On CD-ROM in file linking/ex1/Example1.java
class Example1 {

    // Assume this application is invoked with one command-line
    // argument,  the string "Hi!".
    public static void main(String[] args) {
```

```
        // argZero, because it is assigned a String from the command
        // line, does not reference a string literal. This string
        // is not interned.
        String argZero = args[0];

        // literalString, however, does reference a string literal.
        // It will be assigned a reference to a String with the value
        // "Hi!" by an instruction that references a
        // CONSTANT_String_info entry in the constant pool. The
        // "Hi!" string will be interned by this process.
        String literalString = "Hi!";

        // At this point, there are two String objects on the heap
        // that have the value "Hi!". The one from arg[0], which
        // isn't interned, and the one from the literal, which
        // is interned.
        System.out.print("Before interning argZero: ");
        if (argZero == literalString) {
        System.out.println("they're the same string object!");
    }
    else {
        System.out.println("they're different string objects.");
    }
        // argZero.intern() returns the reference to the literal
        // string "Hi!" that is already interned. Now both argZero
        // and literalString have the same value. The non-interned
        // version of "Hi!" is now available for garbage collection.
        argZero = argZero.intern();
        System.out.print("After interning argZero: ");
        if (argZero == literalString) {
          System.out.println("they're the same string object!");
    }
        else {
            System.out.println("they're different string objects.");
        }
    }
}
```

把字符串“Hi!”作为第一个命令行参数执行时，示例程序Example1输出下列结果：

```
Before interning argZero: they're different string objects.
After interning argZero: they're the same string object!
```

8.1.9 解析其他类型的入口

CONSTANT_Integer_info、CONSTANT_Long_info、CONSTANT_Float_info和CONSTANT_Double_info入口本身包含它们所表示的常量值，它们可以直接被解析。要解析这类入口，很多虚拟机的实现什么都不需要做，直接使用那些值就行了。然而还有一些实现可能

需要做一些处理。比如，在“小数在前”的计算机上运行的虚拟机可能选择在解析时交换值的字节顺序。

CONSTANT_Utf8_info和CONSTANT_NameAndType_info类型的入口永远不会被指令直接引用。它们只有通过其他入口类型才能被引用，并且在那些引用入口被解析时才被解析。

8.1.10 装载约束

Java类型可以符号化地引用常量池中的其他类型，解析时需要特别关照，当存在多个类装载器的时候，要保证类型安全。当一个类型包含指向另一个类型中的字段的符号引用时，符号引用包含一个描述符——它指明了该字段的类型。当一个类型包含指向另外一个类型的方法的符号引用时，符号引用也包含一个描述符——它指明了返回值的类型和参数（如果有参数的话）。如果引用的类型和被引用类型并非由同一个初始装载器装载，虚拟机必须确保在字段或者方法描述符中提及的类型在不同的命名空间中保持一致。比如说，假设类Cat包含指向在类Mouse中声明的字段和方法的符号引用，两个不同的类装载器分别初始装载了Cat和Mouse。为了保证存在多个类装载器时类型的安全，虚拟机必须保证，Cat中包含的字段和方法描述符所提及的类型的全限定名，必须和类Mouse中同样的名字指向同一类型数据（在方法区中）。

为了确保Java虚拟机实现能够保证类型在不同命名空间保持一致性，Java虚拟机规范第2版定义了几种装载约束。每一个Java虚拟机都必须维护一个有关这些约束的内部列表，每一个约束基本上都表明了一个命名空间中的某个名字必须和另一个命名空间中的同一个名字指向同一类型数据。无论何时Java虚拟机遇到某些指向被引用类型的字段和方法的符号引用，且被引用类型的初始装载并非是初始装载引用类型的同一个类装载器，虚拟机就会在列表中加上一个约束。虚拟机在解析符号引用的时候必须检查当前已经装载的所有约束。

为了描述装载约束，符号<C，Ld>^{Li}用来表示类型。C表示类型的全限定名，Ld表示类型的定义类装载器，Li表示类型的初始类装载器。如果不讨论定义类装载器，简化用C^{Li}来表示类型和它的初始类装载器。如果不讨论初始类装载器，简化用<C，Ld>表示类型和它的定义类装载器。在两个类型之间的等于符号，表示两个类型实际上是同样的类型，表示方法区的同一段类型数据。

有了这些符号，产生装载约束的规则表示如下：

- 当解析一个包含在<C，L1>中的符号引用（它指向的是类<D，L2>中声明的类型T的字段）时，虚拟机必须产生下列装载约束：

  $T^{L1} = T^{L2}$

- 当解析一个包含在<C，L1>中的符号引用（它指向的是一个方法，其返回值是T_0类型，其参数是（T_1，...，T_n）类型，在类<D，L2>中声明）时，虚拟机必须产生下列装载约束：

  $T_0^{L1} = T_0^{L2}$，...，$T_n^{L1} = T_n^{L2}$

- 当<C，L1>重载方法（该方法返回值是T_0类型，其参数是（T_1，...，T_n）类型，在类<D，L2>中声明），虚拟机必须产生下列装载约束：

  $T_0^{L1} = T_0^{L2}$，...，$T_n^{L1} = T_n^{L2}$

如果虚拟机关于约束的内部列表包含两条约束：$T^{L1} = T^{L2}$和$T^{L2} = T^{L3}$，这意味着$T^{L1} = T^{L3}$。就算在虚拟机实例执行中类型T从没有被L2装载过，L1和L3装载的名为T的类型是严格一致的。

了解装载约束时如果不想看这么多数学用语，请参考本章8.12节的示例，该例子显示了，如果没有装载约束，一个黑客攻击者是如何突破Java虚拟机的类型安全保障的。

8.1.11 编译时常量解析

在第7章讲过，被初始化为编译时常量的静态final变量的引用，在编译时被解析为常量值的一个本地拷贝，这对于所有的基本类型和java.lang.String都是正确的。

这种对于常量的特别处理使Java语言具有了两个特性。首先，常量值的本地拷贝使得静态final变量可以用于switch语句中的case表达式。在字节码中实现switch语句的两条虚拟机指令是tableswitch和lookupswitch，需要case值内嵌在字节码流中。这些指令不支持运行时解析case值。要了解这两条指令的更多信息，请参阅第16章。

隐藏在常量的特殊处理后面的另一个动机是条件编译。通过if语句（其表达式解析成编译时常量），Java支持条件编译。下面是一个例子：

```
// On CD-ROM in file linking/ex2/AntHill.java
class AntHill {

    static final boolean debug = true;
}

// On CD-ROM in file linking/ex2/Example2.java
class Example2 {

    public static void main(String[] args) {
        if (AntHill.debug) {
            System.out.println("Debug is true!");
        }
    }
}
```

因为基本类型常量的特别处理，Java编译器可以通过AntHill.debug的值，决定是否包括Example2. main（）中的if语句体。因为在这个情形下AntHill.debug是true，javac为Example2的main（）方法生成的字节码中就包含if语句的语句体，但是并不包括检查AntHill.debug的值。在Example2的常量池中并没有指向类AntHill的符号引用。下面是main（）方法的字节码：

```
                // Push objref from System.out
0 getstatic #8 <Field java.io.PrintStream out>
                // Push objref to literal string "Debug is true!"
3 ldc #1 <String "Debug is true!">
                // Pop objref (to a String), pop objref(to
                // System.out), invoke println() on System.out
                // passing the string as the only parameter:
                // System.out.println("Debug is true!");
5 invokevirtual #9 <Method void println(java.lang.String)>
8 return        // return void
```

如果指向AntHill.debug的引用是在运行时解析的，编译器就需要检查AntHill.debug的值和if

语句的语句体，以防AntHill.debug的值改变了。实际上AntHill.debug的值编译之后就不可能改变，因为它声明为final。然而，仍然可以改变AntHill的源代码并重新编译AntHill，但是不重新编译Example2。

因为指向AntHill.debug的引用是在编译时解析的，编译器如果发现AntHill.debug是false，编译器就会有条件地编译if语句的语句体。请注意，这意味着如果只是把AntHill设置为false，只重新编译AntHill，而无法改变Example2程序的行为。必须也重新编译Example2。

下面的Example3，是把Example2的名字换成了Example3，并且把AntHill的debug值设置为false时的编译结果。

```
// On CD-ROM in file linking/ex3/AntHill.java
class AntHill {

    static final boolean debug = false;
}

// On CD-ROM in file linking/ex3/Example3.java
class Example3 {

    public static void main(String[] args) {
        if (AntHill.debug) {
            System.out.println("Debug is true!");
        }
    }
}
```

下面是javac生成的Example3的main（）方法的字节码。

```
0 return      // return void
```

可以看到，Java编译器把整个if语句都从Example3.main（）方法中去处了。在这个短短的字节码序列中，甚至没有任何println（）调用的提示。

8.1.12 直接引用

常量池解析的最终目标是把符号引用替换为直接引用。符号引用的格式在第6章中详细定义了，但是直接引用应该是什么格式呢？你可能认为，直接引用的格式也是由不同的Java虚拟机实现的设计者决定的。然而，在大多数实现中，总会有一些通用的特征。

指向类型、类变量和类方法的直接引用可能是指向方法区的本地指针。类型的直接引用可能简单地指向保存类型数据的方法区中的与实现相关的数据结构。类变量的直接引用可以指向方法区中保存的类变量的值。类方法的直接引用可以指向方法区中的一段数据结构方法区中包含调用方法的必要数据）。比如，类方法的数据结构可能包含方法是否为本地方法的标志信息。如果方法是本地的，数据结构可能包含一个指向动态连接的本地方法实现的函数指针。如果方法不是本地的，数据结构可能包含方法的字节码、max_stack、max_local等信息。如果有一个该方法的即时编译版本，数据结构可能包含指向即时编译的本地代码的指针。

指向实例变量和实例方法的直接引用都是偏移量。实例变量的直接引用可能是从对象的映

像开始算起到这个实例变量位置的偏移量。实例方法的直接引用可能是到方法表的偏移量。

使用偏移量来表示实例变量和实例方法的直接引用，取决于类的对象映像中字段的顺序和类方法表中方法的顺序的预先决定。尽管不同实现的设计者可以选择在对象映像中存放实例变量的方式，以及在方法表中存放方法的方式，但几乎可以肯定的是，他们对所有的类型都使用同样的方式。所以，在任何实现中，对象中字段的顺序和方法表中方法的顺序是被定义好的，也是可以预测的。

例如，考虑下面的由三个类和一个接口组成的层次关系。

```
// On CD-ROM in file linking/ex4/Friendly.java
interface Friendly {

    void sayHello();
    void sayGoodbye();
}

// On CD-ROM in file linking/ex4/Dog.java
class Dog {

    // How many times this dog wags its tail when
    // saying hello.
    private int wagCount = ((int) (Math.random() * 5.0)) + 1;

    void sayHello() {

        System.out.print("Wag");
        for (int i = 0; i < wagCount; ++i) {
            System.out.print(", wag");
        }
        System.out.println(".");
    }

    public String toString() {

        return "Woof!";
    }
}

// On CD-ROM in file linking/ex4/CockerSpaniel.java
class CockerSpaniel extends Dog implements Friendly {

    // How many times this Cocker Spaniel woofs when saying hello.
    private int woofCount = ((int) (Math.random() * 4.0)) + 1;

    // How many times this Cocker Spaniel wimpers when saying
    // goodbye.
```

```
    private int wimperCount = ((int) (Math.random() * 3.0)) + 1;

    public void sayHello() {

        // Wag that tail a few times.
        super.sayHello();

        System.out.print("Woof");
        for (int i = 0; i < woofCount; ++i) {
            System.out.print(", woof");
        }
        System.out.println("!");
    }

    public void sayGoodbye() {

        System.out.print("Wimper");
        for (int i = 0; i < wimperCount; ++i) {
            System.out.print(", wimper");
        }
        System.out.println(".");
    }
}

// On CD-ROM in file linking/ex4/Cat.java
class Cat implements Friendly {

    public void eat() {

        System.out.println("Chomp, chomp, chomp.");
    }

    public void sayHello() {

        System.out.println("Rub, rub, rub.");
    }

    public void sayGoodbye() {

        System.out.println("Scamper.");
    }

    protected void finalize() {

        System.out.println("Meow!");
    }
}
```

假设装载这些类型的Java虚拟机采用的组织对象的方式是，实例变量在子类中声明之前，就把在超类中声明的该实例变量放到了对象映像中；并且每一个类的实例变量出现的顺序和它们在class文件中出现的顺序是一致的。假设类Object没有实例变量，Dog、CockerSpaniel和Cat的对象映像如图8-1所示。

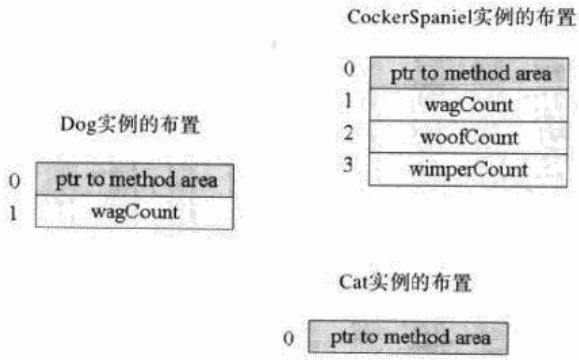


图8-1 对象映像

在图8-1中，CockerSpaniel的对象映像很好地说明了这个特定的虚拟机排列对象的方式。来自超类Dog的实例变量出现在来自子类CockerSpaniel的实例变量之前。CockerSpaniel的实例变量按照它们声明的顺序出现：先是woofCount，然后是wimperCount。

注意实例变量wagCount在Dog和CockerSpaniel中都作为偏移量1出现。在这个Java虚拟机实现中，指向类Dog的wagCount字段的符号引用会被解析成为一个偏移量为1的直接引用。不管实际的对象是Dog、CockerSpaniel，或任何Dog的子类，实例变量wagCount总是在对象映像中作为偏移量1出现。

在方法表中也呈现出同样的情形。方法表中的一个入口以某种方式关联到方法区中的一段数据结构（方法区包含让虚拟机调用此方法的足够信息）。假设在我们现在描述的Java虚拟机实现中，方法表是关于指向方法区的指针的数组。方法表入口指向的数据结构和我们前面提到的类方法的数据结构类似。假设这种特定的Java虚拟机实现装载方法表的方法是，来自超类的方法出现在来自子类的方法之前；并且每个类排列指针的顺序和方法在class文件中出现的顺序相同。这种排列顺序的例外情况是，被子类的方法覆盖的方法出现在超类中该方法第一次出现的地方。

这个虚拟机组织Dog类方法表的情况如图8-2所示。在该图中，指向在类Object中定义的方法的方法表入口，在图中显示为深灰色；指向在Dog中定义的方法的入口显示为浅灰色。

注意在这个方法表中只有非私有的实例方法才会出现。用invokestatic指令调用的类方法不在这里出现，因为它们是静态绑定的，不需要在方法表的间接指向。私有的方法和实例的初始化方法不需要在这里出现，因为它们是被invokespecial指令调用的，所以也是静态绑定的。只有被invokevirtual或者invokeinterface调用的方法才需要出现在的这个方法表中。参见第19章讨论的调用指令。

通过源代码，可以看到Dog覆盖了Object类中定义的toString（）方法。在Dog的方法表中，toString（）方法只出现了一次，在Object的方法表中出现的同样的位置出现（偏移量7）。在Dog

的方法表中，这个指针位于偏移量7，并且指向Dog的toString（）实现的数据。在这个Java虚拟机实现中，指向toString（）方法数据的指针会在每个类的方法表中都处于偏移量7。（实际上，可以编写一个定制版本的java.lang.Object，并且用一个自定义的类装载器来装载。用这种方法，可以创建一个命名空间，使用同样的虚拟机，在这个命名空间中指向toString（）方法的指针就可以不处于方法表的偏移量7的位置。）

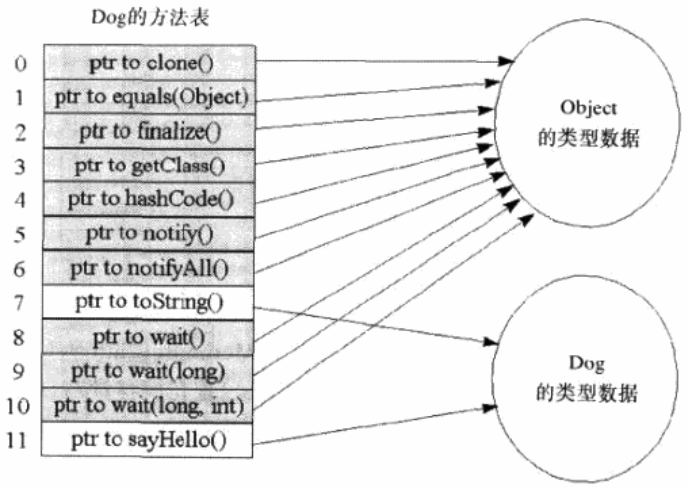


图8-2 Dog类的方法表

位于Object中声明的方法下方的第一个方法，是Dog中声明的方法，这些方法没有重载Object的方法。只有一个sayHello（）方法，位于方法表偏移量11。所有Dog的子类都会继承或者覆盖这个sayHello（）方法的实现，并且所有Dog的子类的sayHello（）会一直出现在偏移量11。

图8-3显示了CockerSpaniel的方法表。注意，因为CockerSpaniel声明了sayHello（）和sayGoodbye（），这些方法的指针指向这些方法的CockerSpaniel实现的数据。因为CockerSpaniel继承了Dog的toString（）实现，该方法的指针（仍然在偏移量7）指向Dog的该方法的实现的数据。CockerSpaniel继承了Object的所有其他方法，所以这些方法的指针直接指向Object的类型数据。注意sayHello（）仍然位于偏移量11，和它在Dog的方法表中的偏移量一致。

当虚拟机解析一个符号引用（CONSTANT_Methodref_info入口）到任何类的toString（）方法的时候，它指向方法表偏移量7。当虚拟机解析指向Dog或者任何子类的sayHello（）方法的符号引用的时候，直接引用是方法表偏移量11。当虚拟机解析指向CockerSpaniel或者任何子类的sayGoodbye（）方法的符号引用的时候，直接引用就是方法表偏移量12。

一旦一个指向实例方法的符号引用被解析为一个方法表偏移量后，虚拟机仍然需要实际调用此方法。要调用一个实例方法，虚拟机在对象中搜寻对象的类的方法表。在第5章中讲过，给定一个指向对象的引用，每一个虚拟机实现必须有办法找到这个对象的类的类型数据。除此之外，给定一个指向对象的引用，方法表（对象的类的类型数据的一部分）一般需要非常快地访问。（图5-7显示了一种可能的情形。）一旦虚拟机有了对象的类的方法表，它用偏移量来找到正确的需要调用的方法。

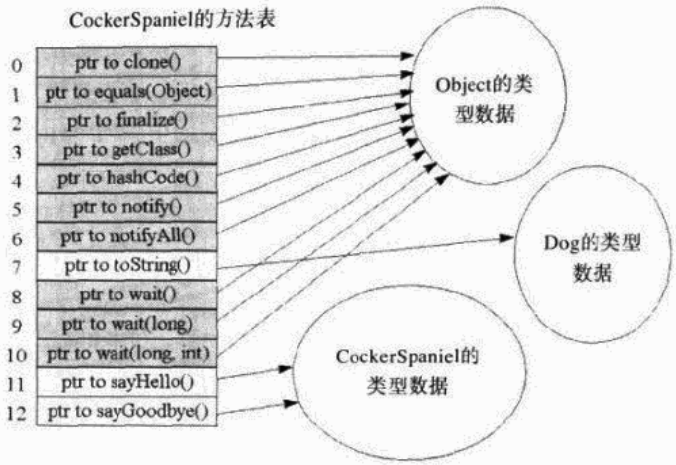


图8-3 CockerSpaniel类的方法表

当虚拟机有一个指向类类型的引用（CONSTANT_Methodref_info入口）的时候，它总是可以依靠方法表偏移量。如果在Dog类中sayHello（）方法出现在偏移量7，那么在Dog的所有子类中它都出现在偏移量7。不过当引用是指向接口类型（CONSTANT_InterfaceMethodref_info入口）的时候，这就不成立了。当通过接口引用来访问实例方法的时候，直接引用不能保证得到方法表偏移量。考虑一下图8-4中的Cat类的方法表。

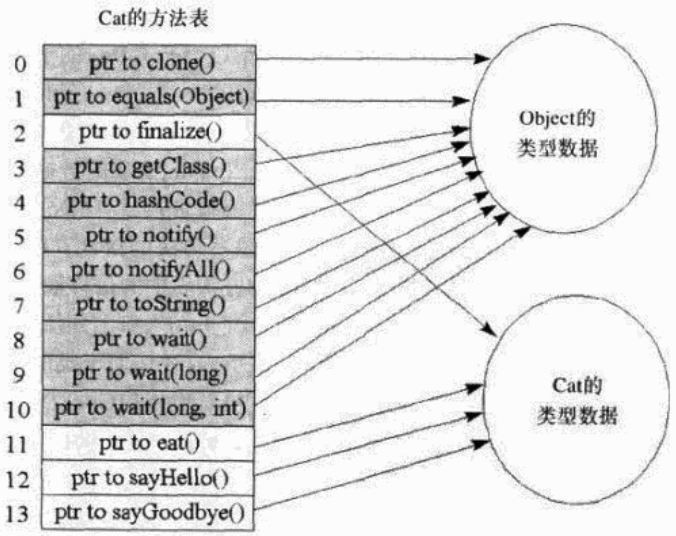


图8-4 Cat类的方法表

注意Cat和CockerSpaniel都实现了Friendly接口。一个类型为Friendly的变量可能保存的是指向Cat对象的引用，也可能是指向CockerSpaniel对象的引用。用这个引用，你的程序可以调用

Cat或者CockerSpaniel或者任何其他实现了Friendly接口的对象的sayHello（）和sayGoodbye（）方法。Example4程序演示了这一点：

```
// On CD-ROM in file linking/ex4/Example4.java
class Example4 {

    public static void main(String[] args) {

        Dog dog = new CockerSpaniel();

        dog.sayHello();

        Friendly fr = (Friendly) dog;

        // Invoke sayGoodbye() on a CockerSpaniel object through a
        // reference of type Friendly.
        fr.sayGoodbye();

        fr = new Cat();

        // Invoke sayGoodbye() on a Cat object through a reference
        // of type Friendly.
        fr.sayGoodbye();
    }
}
```

在Example4中，本地变量fr调用了CockerSpaniel对象和Cat对象的sayGoodbye（）方法，调用二者的该方法使用的是常量池中的同一个CONSTANT_InterfaceMethodref_info入口。但是当虚拟机解析指向sayHello（）的符号引用的时候，它不能只保存一个方法表偏移量，然后指望将来对常量池入口操作时依赖这个偏移量。

麻烦之处在于，实现Friendly接口的类并不能保证都是从同一个超类继承的，这个超类也同样实现Friendly接口。这样，Friendly中声明的方法并不能保证处于方法表的同一位置。比如，如果比较一下CockerSpaniel和Cat的方法表的话，会发现在CockerSpaniel中，sayHello（）位于偏移量11；但是在Cat中，sayHello（）出现在偏移量12。同样，CockerSpaniel的sayGoodbye（）方法的指针位于偏移量12，而Cat的sayGoodbye（）方法的指针位于偏移量13。

因此，不管何时Java虚拟机从接口引用调用一个方法，它必须搜索对象的类的方法表来找到一个合适的方法。这种调用接口引用的实例方法的途径会比在类引用上调用实例方法慢很多。当然，关于如何搜索类的方法表，虚拟机实现可以灵活一些。例如，实现可以保存最后找到方法时的索引，然后在下次搜索时首先查找该位置。或者某些实现可能在准备的时候就建立一些数据结构，这有助于在给定一个接口引用的时候搜索方法表。不管怎样，给定接口引用时调用方法总是比给定类引用时调用方法慢得多。

8.1.13 _quick 指令

Java虚拟机的第一版规范中描述了一种用来加速字节码解释的技术，Sun早期的一种Java虚

拟机实现中曾经使用过它。这种方案是，如果常量池入口已经被解析过，使用常量池入口的操作码会被一个“_quick”操作码取代。当虚拟机遇到一个_quick指令时，它就知道常量池入口已经被解析过了，所以这条指令可以执行得更快。

Java虚拟机的核心指令集包含200条单字节操作码，它们都可以在附录A中找到。在class文件中只会见到这200条操作码。使用“_quick”技术的虚拟机实现内部还另外使用25条“_quick”单字节操作码。

举例来说，当使用_quick技术的虚拟机解析一个ldc指令（操作码0x12）引用的常量池入口时，它在字节码流中用ldc_quick指令（操作码Oxcb）替换ldc操作码字节。在Sun的早期虚拟机中，这项技术是属于直接引用替换符号引用这个操作的一部分。

除了用_quick操作码覆盖原来的普通操作码之外，使用_quick技术的虚拟机对某些指令还会用表示直接引用的数据覆盖指令的操作数。比如，除了把invokevirtual操作码替换成invokevirtual_quick之外，虚拟机还把方法表偏移量和参数的个数放入每个invokevirtual指令后面的两个操作数字节中。在invokevirtual_quick操作码后面的字节码流中直接用方法表偏移量，节约了在解析过的常量池入口中查找偏移量的时间。

8.1.14 示例：Salutation程序的连接

Java连接模型的例子如下面的Salutation程序：

```
// On CD-ROM in file linking/ex5/Salutation.java
class Salutation {

    private static final String hello = "Hello, world!";
    private static final String greeting = "Greetings, planet!";
    private static final String salutation = "Salutations, orb!";

    private static int choice = (int) (Math.random() * 2.99);

    public static void main(String[] args) {

        String s = hello;
        if (choice == 1) {
            s = greeting;
        }
        else if (choice == 2) {
            s = salutation;
        }

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

假设想让Java虚拟机运行Salutation。当虚拟机启动时，它试图调用Salutation的main（）方法。但虚拟机很快意识到，不管用什么办法，也无法调用main（）。调用类中声明的方法是对类的一次主动使用，在类被初始化之前，这是不能允许的。所以，在虚拟机可以调用main（）之

前，它必须初始化Salutation。在能够初始化Salutation之前，它必须首先装载并连接它。所以，虚拟机把Salutation的全限定名交给了启动类装载器，后者取得类的二进制形式，把二进制数据解析成内部的数据结构，并且创建一个java.lang.Class的实例。Salutation的常量池如表8-1所示。

表8-1 类Salutation的常量池

索 引	类 型	值
1	CONSTANT_String_info	30
2	CONSTANT_String_info	31
3	CONSTANT_String_info	39
4	CONSTANT_Class_info	37
5	CONSTANT_Class_info	44
6	CONSTANT_Class_info	45
7	CONSTANT_Class_info	46
8	CONSTANT_Class_info	47
9	CONSTANT_Methodref_info	7，16
10	CONSTANT_Fieldref_info	4，17
11	CONSTANT_Fieldref_info	8，18
12	CONSTANT_Methodref_info	5，19
13	CONSTANT_Methodref_info	6，20
14	CONSTANT_Double_info	2.99
16	CONSTANT_NameAndType_info	26，22
17	CONSTANT_NameAndType_info	41，32
18	CONSTANT_NameAndType_info	49，34
19	CONSTANT_NameAndType_info	50，23
20	CONSTANT_NameAndType_info	51，21
21	CONSTANT_Utf8_info	"（）D"
22	CONSTANT_Utf8_info	"（）V"
23	CONSTANT_Utf8_info	"（Ljava/lang/String;）V"
24	CONSTANT_Utf8_info	"（[Ljava/lang/String;）V"
25	CONSTANT_Utf8_info	"<clinit>"
26	CONSTANT_Utf8_info	"<init>"
27	CONSTANT_Utf8_info	"Code"
28	CONSTANT_Utf8_info	"ConstantValue"
29	CONSTANT_Utf8_info	"Exceptions"
30	CONSTANT_Utf8_info	"Greetings， planet!"
31	CONSTANT_Utf8_info	"Hello， world!"
32	CONSTANT_Utf8_info	"I"
33	CONSTANT_Utf8_info	"LineNumberTable"
34	CONSTANT_Utf8_info	"Ljava/io/PrintStream;"
35	CONSTANT_Utf8_info	"Ljava/lang/String;"
36	CONSTANT_Utf8_info	"LocalVariables"
37	CONSTANT_Utf8_info	"Salutation"
38	CONSTANT_Utf8_info	"Salutation.java"
39	CONSTANT_Utf8_info	"Salutations， orb!"
40	CONSTANT_Utf8_info	"SourceFile"

（续）

索　引	类　型	值
41	CONSTANT_Utf8_info	"choice"
42	CONSTANT_Utf8_info	"greeting"
43	CONSTANT_Utf8_info	"hello"
44	CONSTANT_Utf8_info	"java/io/PrintStream"
45	CONSTANT_Utf8_info	"java/lang/Math"
46	CONSTANT_Utf8_info	"java/lang/Object"
47	CONSTANT_Utf8_info	"java/lang/System"
48	CONSTANT_Utf8_info	"main"
49	CONSTANT_Utf8_info	"out"
50	CONSTANT_Utf8_info	"println"
51	CONSTANT_Utf8_info	"random"
52	CONSTANT_Utf8_info	"salutation"

Salutation装载过程中，Java虚拟机必须首先确认所有Salutation的超类都被装载了。在开始这个过程前，虚拟机察看super_class项所指定的Salutation的类型数据，它的值是7。虚拟机查询常量池中的第7个入口，发现这是一个CONSTANT_Class_info入口，它的内容是一个指向java.lang.Object的符号引用，图8-5是这个符号引用的一个图形化描绘。虚拟机解析这个符号引用，这导致装载类Object。因为Object是Salutation继承树的顶端，所以虚拟机就连接并初始化它。

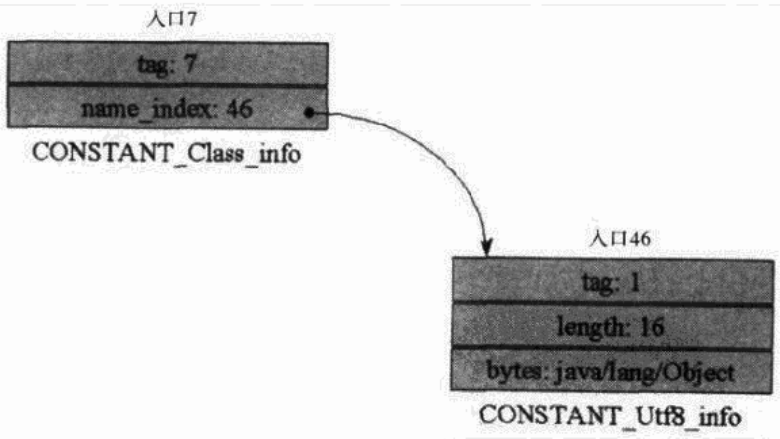


图8-5　Salutation到Object的符号引用

现在Java虚拟机已经装载了Salutation类，并且也已经装载、连接并且初始化了它的所有超类，现在虚拟机准备来连接Salutation了。连接过程的第一个步骤，就是虚拟机校验类Salutation的二进制形式的完整性。假设这种Java虚拟机的实现已经在前面做过了所有其他的校验了，只剩下校验符号引用了。所以，在正式的连接校验步骤完成之前，虚拟机会校验：

1) Salutation的二进制数据结构上是正确的。

2) Salutation正确地实现了Java语言的语义。

3) Salutation的字节码不会导致虚拟机崩溃。

当Java虚拟机校验完Salutation后，它必须为Salutation准备需要的内存空间。在这个阶段，虚拟机为Salutation的类变量choice分配内存，并且给它一个默认初始值。因为choice类变量是一个int型数据，所以它接受默认初始值0。

三个文字字符串（hello、greeting和salutation）是常量，而非类变量。它们不在方法区中作为类变量占据内存空间，它们也不需要接受默认初始值。因为它们被定义为静态的和最终的，在Salutation的常量池中作为CONSTANT_String_info入口出现。javac产生的Salutation的常量池如表8-1所示。表示Salutation的常量字符串的入口如下：第一个入口表示greeting；第二个表示hello；第三个表示salutation。

当校验和准备过程成功结束后，类已经准备好被解析了。前面提到过，不同的虚拟机实现在不同的时期执行连接的解析步骤。Salutation在其生命周期中的这个时间点进行解析是可选的，在符号引用实际被程序使用前，Java虚拟机并不需要解析符号引用。如果一个符号引用并不会被程序所调用，虚拟机并不需要去解析它。

如同前面说到的，某种Java虚拟机实现可能在程序生命的这个时间点执行递归的解析过程。如果是这样，程序在main（）被调用前就已经被完全连接了。有的Java虚拟机实现可能在这个时间点根本就不执行任何解析过程，而是直到运行的程序第一次真正使用某个符号引用的时候才去解析它。还有的虚拟机实现可能采取介于这两个极端方式之间的解析策略。虽然不同的虚拟机实现可能在不同的时间执行解析，但是所有的虚拟机实现都必须确保类型在使用前已经装载、校验、准备并且初始化了。

假设Java虚拟机的实现使用迟解析。每个符号引用都在程序第一次使用时被解析，会检查它的正确性并转换成一个直接引用。还假设这个虚拟机也使用把引用常量池的操作码替换成_quick等价形式的技术。

一旦这个Java虚拟机装载、校验、准备了Salutation，就可以初始化了。在前面提到过，Java虚拟机必须在初始化一个类之前初始化它所有的超类。在这个例子中，虚拟机已经初始化了Salutation的超类——Object。

当虚拟机确认所有Salutation的超类已经被初始化了（在这个例子中就是类Object），它就准备执行Salutation的<clinit>（）方法。因为Salutation包含一个类变量width，它有一个无法在编译时解析为常量的初始化方法，所以编译器就在Salutation的class文件中放了一个<clinit>（）方法。

下面就是Salutation的<clinit>（）方法：

```
            // Invoke class method Math.random(), passing no
            // parameters. Push double result.
 0 invokestatic #13 <Method double random()>
           // Push double constant 2.99 from constant pool.
 3 ldc2_w #14 <Double 2.99>
 6 dmul     // Pop two doubles, multiple, push double result.
 7 d2i      // Pop double, convert to int, push int result.
           // Pop int, store int Salutation.choice
 8 putstatic #10 <Field int choice>
11 return   // Return void from <clinit>()
```

Java虚拟机执行Salutation的<clinit>（）方法，把choice属性设置成正确的初始值。在执行

<clinit>（）之前，choice的默认初始值为0；在执行<clinit>（）之后，Choice的值被伪随机地置为三者之一：0，1，或者2。

<clinit>（）方法的第一条指令——invokestatic #13，引用常量池入口13，它是一个CONSTANT_Methodref_info入口，代表一个对java.lang.Math的random（）方法的符号引用。可以在图8-6中看到这个符号引用的图形化描绘。Java虚拟机解析这个符号引用，这导致对类java.lang.Math的装载、连接和初始化。然后在常量池入口13放一个指向random（）方法的直接引用，把这个入口标记为已解析的，并且把invokestatic操作码替换成invokestatic_quick。

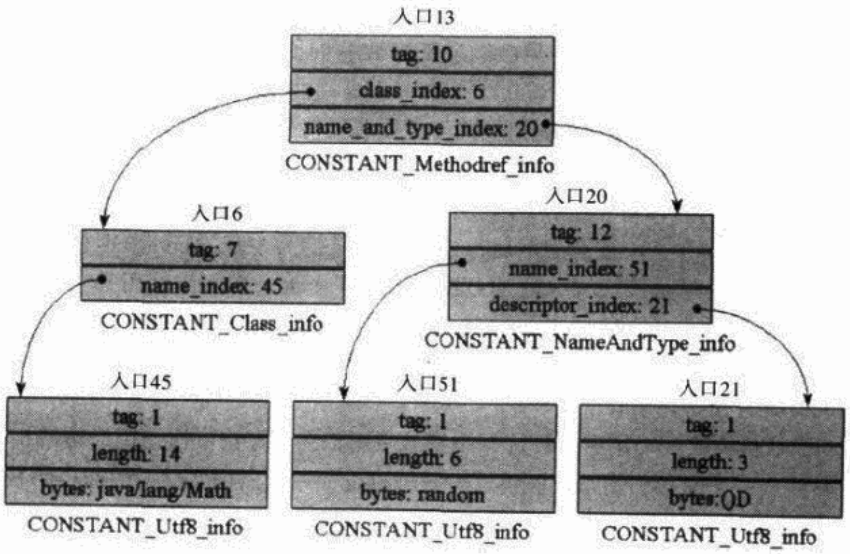


图8-6 从Salutation到Math.random（）的符号引用

完成解析常量池入口13的过程后，Java虚拟机已经可以调用这个方法了。当虚拟机实际调用random（）方法的时候，它会装载、连接并初始化任何从Math的常量池和random（）的代码中符号引用的类型。当这个方法返回的时候，虚拟机会把返回的double值压入main（）方法的操作数栈中。

为了执行下一条指令——ldc2_w #14，虚拟机查找常量池入口14，发现一个未解析的CONSTANT_Double_info入口。虚拟机把这个入口解析成double值2.99，把这个入口标记为已解析，并且把ldc2_w操作码替换成ldc2_w_quick。一旦虚拟机解析了常量池入口14后，它马上把常量double值2.99压入操作数栈。

注意入口CONSTANT_Double_info，它没有指向任何其他常量池入口，也没有引用本类以外的内容，8个字节的double值2.99是在这个入口内部指定的。

还要注意，这个常量池中没有索引为15的入口。在第6章中讲过，CONSTANT_Double_info和CONSTANT_Long_info类型的入口在常量池中占据两个位置，所以，可以认为CONSTANT_Double_info同时占据了索引14和15。

执行下一条指令——dmul，虚拟机弹出两个double数，把它们相乘，再把double结果压入栈。

在下一条指令中，虚拟机弹出这个double值，把它转换成int，再把int结果压入栈。假设这次执行Salutation，这个操作的结果是整数2。

下一条指令——putstatic #10，引用了常量池中的另一个符号引用，Salutation自己的变量choice。一个类的字节码除了可以符号引用其他类型的字段和方法外，也可以引用自己的字段和方法，这条指令就是一个例子。当虚拟机执行这条指令的时候，它查看常量池入口10，发现这是一个未解析的CONSTANT_Fieldref_info入口，图8-7就是这个符号引用的图形化描绘。虚拟机解析这个引用，在方法区中，找到Salutation的类型数据中的choice类变量，把一个指向这个实际变量数据的指针放入常量池入口10。然后标记这个入口已经被解析过了，并且把putstatic操作码替换成putstatic_quick。

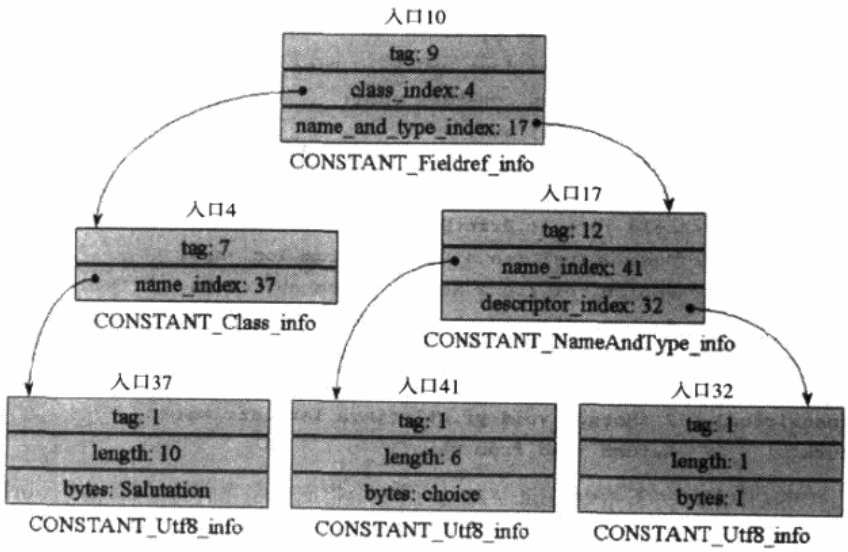


图8-7 从Salutation指向它自己的choice字段的符号引用

一旦解析了choice的CONSTANT_Fieldref_info入口，虚拟机从操作数栈弹出int值（这里是2），把它放入choice变量。这条putstatic指令现在就执行完了。

最后，虚拟机执行return指令，它通知虚拟机，<clinit>（）方法完成了（因此类Salutation的初始化也结束了）。

现在类Salutation已经被初始化了，终于可以使用它了。Java虚拟机调用main（），程序开始执行。下面是Salutation的main（）方法的字节码序列：

```
                // Push objref to literal string from constant pool
                // entry 2
0 ldc #2 <String "Hello, world!">
2 astore_1      // Pop objref into loc var 1: String s = hello;
                // Push int from static field Salutation.choice. Note
                // that by this time, choice has definitely been
                // given its proper initial value.
3 getstatic #10 <Field int choice>
```

```
 6 iconst_1      // Push int constant 1
                 // Pop two ints, compare, if not equal branch to 16:
 7 if_icmpne 16 // if (choice == 1) {
                 // Here, choice does equal 1. Push objref to string
                 // literal from constant pool:
10 ldc #1 <String "Greetings, planet!">
12 astore_1     // Pop objref into loc var 1: s = greeting;
13 goto 26      // Branch unconditionally to offset 26
                 // Push int from static field Salutation.choice
16 getstatic #10 <Field int choice>
19 iconst_2     // Push int constant 2
                 // Pop two ints, compare, if not equal branch to 26:
20 if_icmpne 26 // if (choice == 2) {
                 // Here, choice does equal 2. Push objref to string
                 // literal from constant pool:
23 ldc #3 <String "Salutations, orb!">
25 astore_1     // Pop objref into loc var 1: String s = salutation;
                 // Push objref from System.out
26 getstatic #11 <Field java.io.PrintStream out>
29 aload_1      // Push objref (to a String) from loc var 1
                 // Pop objref (to a String), pop objref(to
                 // System.out), invoke println() on System.out
                 // passing the string as the only parameter:
                 // System.out.println(s);
30 invokevirtual #12 <Method void println(java.lang.String)>
33 return       // Return void from main()
```

main（）方法中的第一条指令是ldc #2，使用一个指向文字字符串“Hello，world!”的符号引用。当虚拟机执行这条指令的时候，它查看常量池入口2，发现这是一个还没有被解析的CONSTANT_String_info入口。图8-8是这个指向文字字符串的符号引用的图形化描述。

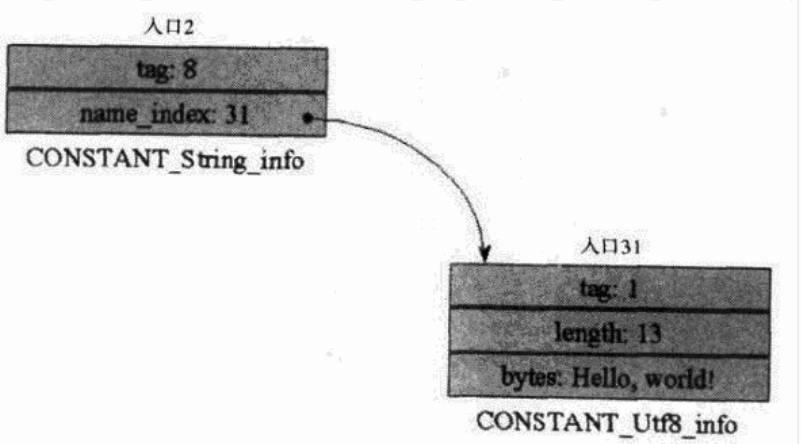


图8-8 从Salutation指向“Hello，world!”的符号引用

虚拟机解析常量池入口是ldc指令执行中的一部分。它创建一个新的值为“Hello，world!”的字符串对象，并且拘留它，在这个常量池入口中放入对这个字符串对象的引用，并把该入口标记为已解析，并且把ldc操作码替换成一个ldc_quick操作码。

现在虚拟机已经解析了“Hello，world”字符串，它把指向字符串对象的引用压入栈。下一条指令astore_1，把引用弹出栈并把它保存到第一个局部变量s。

要执行下一条语句getstatic #10，虚拟机查看常量池入口10，并且发现这是个已经被解析过的CONSTANT_Fieldref_info入口，这个入口指向Salutation自身的choice字段的符号引用，在<clinit>（）方法中被putstatic #10指令解析过了。虚拟机只需简单地把getstatic操作码替换成getstatic_quick，然后把choice的int值压入栈。

要执行main（）方法的下一条指令iconst_1，虚拟机简单地在操作数栈中压入int 1。下一条指令ificmpne 16，虚拟机把最顶端的两个int值弹出来，用一个减去另一个。在这个例子中，因为choice在<clinit>（）方法中被设置为2，所以相减的结果不是0。结果是，虚拟机进入分支并更新PC寄存器，以便下一条要执行的指令变成偏移量为16的getstatic指令。

偏移量16的getstatic指令引用了和偏移量3的getstatic指令同样的常量池入口10。当虚拟机执行位于偏移量16的getstatic指令时，发现常量池10是一个已经解析过的CONSTANT_Fieldref_info入口。它把getstatic指令替换为getstatic_quick，并且把Salutation的choice类变量的int值（2）压入操作数栈。

要执行下一条指令iconst_2，虚拟机把一个int 2压入栈。下一条指令又是一个ificmpne 26，虚拟机把两个整数从栈中弹出来相减。这一次，两个int都是2，所以相减的结果是0。结果是，虚拟机没有执行分支，继续执行字节码数组中的下一条指令——又是一个ldc。

位于偏移量23的ldc指令引用了第3个常量池入口，那是一个CONSTANT_String_info入口，表示指向文字字符串“Salutations，orb!”的符号引用。虚拟机在常量池中查找，发现这个入口还没有被解析过。为了解析它，虚拟机创建了一个值为“Salutations，orb!”的新字符串对象，并且在内部拘留它。然后把这个新对象的引用放到常量池入口3，把ldc指令替换成ldc_quick。解析这个字符串之后，虚拟机把字符串对象的引用压入栈。

要执行的下一条指令是astore_1，虚拟机把字符串“Salutations，orb!”的对象引用从栈中弹出来，放入第一个局部变量，就把位于偏移量2的那一条astore_1指令设置的指向“Hello，world!”的引用覆盖了。

要执行的下一条指令是getstatic #11，使用了一个指向java.lang.System的公开静态类变量的符号引用，这个类变量的名字是out，类型是java.io.PrintStream。这个符号引用是位于常量池索引号11的CONSTANT_Fieldref_info入口。图8-9是这个符号引用的图形化描述。

要解析System.out的引用，Java虚拟机必须装载、连接还要初始化java.lang.System，这样才能确保有一个公开的静态字段，名字叫做out，类型是java.io.PrintStream。然后，虚拟机会把符号引用替换成直接引用，比如是一个本地指针，这样以后Salutation使用System.out的时候就不再需要解析，速度会变得更快。最后虚拟机会把getstatic操作码替换成getstatic_quick。

一旦虚拟机成功地解析了这个符号引用，它会把一个指向System.out的引用压入栈。执行下一条指令aload_1，虚拟机简单地把本地变量1中的对象引用压入栈，那是一个指向“Salutations，

orb!”文字字符串的引用。

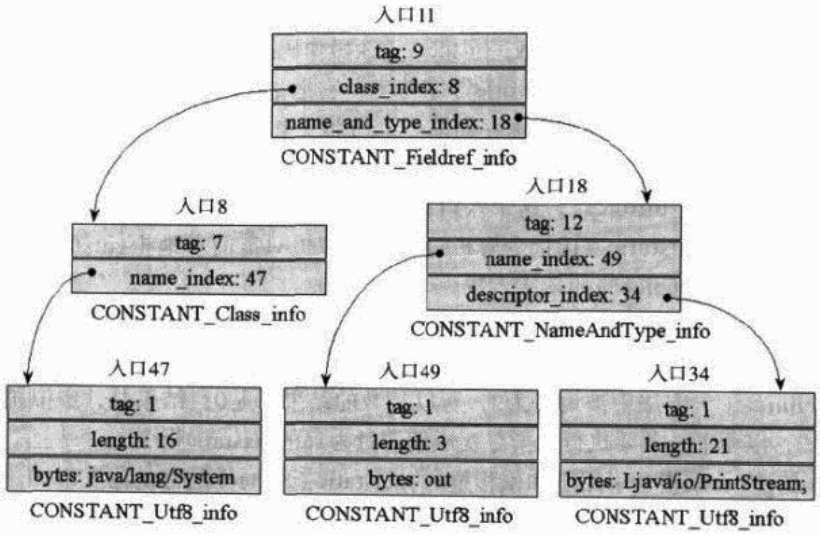


图8-9 从Salutation指向System.out的符号引用

要执行的下一条指令是invokevirtual #12，Java虚拟机查找常量池入口12，发现这是个未解析的CONSTANT_Methodref_info入口，一个指向java.io.PrintStream的println（）方法的符号引用。图8-10是这个符号引用的图形化描述。虚拟机装载、连接、初始化java.io.PrintStream，并且确保它有一个公开方法println（），返回void，使用一个字符串作为参数。它标记这个入口是解析过了，把直接引用（就是在PrintStream的方法表中的索引）放到解析过的常量池入口的数据中。然后，虚拟机把invokevirtual操作码替换成invokevirtual_quick，把方法表索引和方法所接受的参数个数作为invokevirtual_quick操作码的操作数。

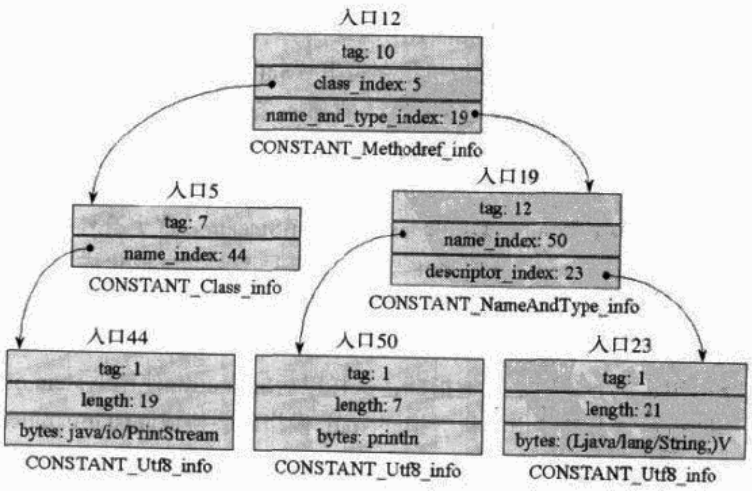


图8-10 从Salutation指向PrintStream.println（）的符号引用

当虚拟机实际调用println（）方法的时候，它会装载、连接并初始化在PrintStream的常量池中和println（）的代码中出现过的任何符号引用。

下一条指令是main（）方法的最后一条指令：return。因为main（）作为Salutation程序的惟一一个非守护线程执行，执行这条return指令能让虚拟机退出。请注意常量池入口1，它保存了指向"Greetings，planet!"文字字符串的符号引用，在Salutation程序执行的过程中还没有被解析过。因为choice恰好被初始化为2，所以位于偏移量10的ldc #1指令没有被执行，ldc #1本来是引用常量池入口1的。结果是，虚拟机没有创建一个值为"Greetings， planet!"的字符串对象。

8.1.15 示例：Greet程序的动态扩展

通过用户自定义的类装载器执行动态扩展的例子，如下面的类所示：

```
// On CD-ROM in file linking/ex6/Greet.java
import com.artima.greeter.*;

public class Greet {

    // Arguments to this application:
    //     args[0] - path name of directory in which class files
    //               for greeters are stored
    //     args[1], args[2], ... - class names of greeters to load
    //               and invoke the greet() method on.
    //
    // All greeters must implement the com.artima.greeter.Greeter
    // interface.
    //
    static public void main(String[] args) {

        if (args.length <= 1) {
            System.out.println(
                "Enter base path and greeter class names as args.");
            return;
        }

        GreeterClassLoader gcl = new GreeterClassLoader(args[0]);

        for (int i = 1; i < args.length; ++i) {
            try {

                // Load the greeter specified on the command line
                Class c = gcl.loadClass(args[i]);

                // Instantiate it into a greeter object
                Object o = c.newInstance();

                // Cast the Object ref to the Greeter interface type
```

```
                // so greet() can be invoked on it
                Greeter greeter = (Greeter) o;

                // Greet the world in this greeter's special way
                greeter.greet();
            }
            catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

Greet程序是一个典型的“Hello，world!”程序。Greet使用一个用户自定义的类装载器使用类（称为“greeters”）来扩展它自身。这些类实际上承担了对公众说“Hello”的实际工作。

一个greeter是任何实现了com.artima.greeter.Greeter接口的类（如下所示）：

```
// On CD-ROM in file linking/ex6/com/artima/greeter/Greeter.java
package com.artima.greeter;

public interface Greeter {

    void greet();
}
```

可以从上面的代码中看到，Greeter接口仅声明了一个方法：greet（）。当一个greeter对象的greet（）方法被调用的时候，这个对象应该用它自己独特的方法对公众说Hello。下面是几个greeter对象的例子：

```
// On CD-ROM in file linking/ex6/greeters/Hello.java
import com.artima.greeter.Greeter;

public class Hello implements Greeter {

    public void greet() {
        System.out.println("Hello, world!");
    }
}

// On CD-ROM in file linking/ex6/greeters/Greetings.java
import com.artima.greeter.Greeter;

public class Greetings implements Greeter {

    public void greet() {
        System.out.println("Greetings, planet!");
    }
```

```
}

// On CD-ROM in file linking/ex6/greeters/Salutations.java
import com.artima.greeter.Greeter;

public class Salutations implements Greeter {

    public void greet() {
        System.out.println("Salutations, orb!");
    }
}

// On CD-ROM in file linking/ex6/greeters/HowDoYouDo.java
import com.artima.greeter.Greeter;

public class HowDoYouDo implements Greeter {

    public void greet() {
        System.out.println("How do you do, globe!");
    }
}
```

greeter对象可以比上面的4个例子更复杂。下面这个greeter对象的例子根据一天中的时间变化来选择欢迎词：

```
// On CD-ROM in file linking/ex6/greeters/HiTime.java
import com.artima.greeter.Greeter;
import java.util.Date;

public class HiTime implements Greeter {

    public void greet() {

        // Date's no-arg constructor initializes itself to the
        // current date and time
        Date date = new Date();
        int hours = date.getHours();

        // Some hours: midnight, 0; noon, 12; 11PM, 23;
        if (hours >= 4 && hours <= 11) {
            System.out.println("Good morning, world!");
        }
        else if (hours >= 12 && hours <= 16) {
            System.out.println("Good afternoon, world!");
        }
        else if (hours >= 17 && hours <= 21) {
            System.out.println("Good evening, world!");
```

```
        }
        else {
            System.out.println("Good night, world!");
        }
    }
}
```

Greet程序在编译时并不知道哪些greeter类会被装载，也不知道那些类在哪儿保存。在运行时，它会从第一个命令行参数得到目录路径，从第二个参数得到greeter类的名字。它试图用这个路径作为基础目录来装载这些greeter类。

比如，假设我们用下面的命令行来调用Greet程序。

```
java Greet greeters Hello
```

在这个命令行里，java是Java虚拟机可执行程序的名字。Greet是Greet程序的类名。greeters是相对于当前目录的路径名，Greet程序会在那儿搜索greeter对象。Hello是greeter对象的名字。

当使用上面的命令行调用Greet程序时，它试图装载greeters/Hello.class并且调用Hello的greet（）方法。如果Hello.class文件的确位于目录greeters中，那么程序会打印出：

```
Hello, world!
```

Greet程序可以处理多个greeter对象。使用下面的命令行调用程序：

```
java Greet greeters Hello Greetings Salutations HowDoYouDo
```

Greet程序会装载列出的四个greeter对象，并调用各自的greet（）方法，结果是：

```
Hello, world!
Greetings, planet!
Salutations, orb!
How do you do, globe!
```

Greet程序首先检查是否至少有两个命令行参数：一个是目录路径，至少还有一个greeter类的名字。然后它创建一个新的GreeterClassLoader对象的实例，后者会负责装载这些greeter对象。（类GreeterClassLoader是java.lang.ClassLoader的一个子类，它的内部工作细节会在后面描述。）GreeterClassLoader的构造方法接受一个字符串，它会用这个字符串作为查找greeter对象的目录路径。

当它创建了GreeterClassLoader对象之后，Greet程序针对每个出现在命令行中的greeter对象名字来调用它的loadClass（）方法。当它调用loadClass（）的时候，把greeter类名args[i]传递过去，每一个类名都单独传递。

```
// Load the greeter specified on the command line
Class c = gcl.loadClass(args[i]);
```

如果loadClass（）方法没有成功，它会抛出一个异常或者一个错误。如果loadClass（）方法成功了，它会返回新装载的类型的Class实例。

请注意，除了被装载以外，请求的loadClass（）的类型也可能在loadClass（）返回之前已经被连接以及初始化了。如果在loadClass（）调用（用于请求类型）之前这个类型已经被主动

使用过了——主动使用会触发类型的装载、连接和初始化。无论如何，在下一条指令（它调用类引用的newInstance（））被执行之前，这个类型一定会被初始化。如果类型还没有初始化，调用newInstance（）会触发它的初始化（这必须是一个类），因为在类被实例化为对象之前它必须被初始化。所以，如果类型在前面的loadClass（）过程中没有被初始化，调用newInstance（）一定会触发初始化过程。

一旦loadClass（）返回了一个Class实例，Greet程序的main（）方法，就通过调用Class实例的newInstance（）方法创建了一个新的greeter实例。

```
// Instantiate it into a greeter object
Object o = c.newInstance();
```

当一个Class对象的newInstance（）方法被调用后，虚拟机创建并且初始化该Class对象代表的类的实例。为了初始化这个新实例，虚拟机调用它的不包含参数的构造方法（注意，如果不想让这条语句出错，新装载的类型必须是一个类，而不是一个接口；必须是可以操作的，而不是抽象的；必须拥有一个无参数的构造方法可以接受操作）。Greet程序接着把对象引用转换成一个Greeter类型。

```
// Cast the Object ref to the Greeter interface type
// so greet() can be invoked on it
Greeter greeter = (Greeter) o;
```

最后，对Greeter引用，main（）方法调用greeter对象的greet（）方法。

```
// Greet the world in this greeter's special way
greeter.greet();
```

Greet程序演示了Java连接模型内在的灵活性。在编译时，Greet程序并不知道它在运行时会装载、动态连接什么。在上面的例子中，类Greet调用了类Hello、类Greetings、类Salutations和类HowDoYouDo的greet（）方法。但是如果你观察Greet的常量池，那里没有指向任何这些类的符号引用。只有指向它们的公用超接口 com.artima.greeter.Greeter的符号引用。greeters对象本身，只要实现了com.artima.greeter.Greeter 接口，可以是任何类，也可以在任何时间被编译，甚至在Greet程序自身被编译之后也可以。

8.1.16 使用1.1版本的用户自定义类装载器

在1.2版本之前，java.lang.ClassLoader的loadClass（）方法是抽象的。要创建自定义的类装载器，必须创建ClassLoader的子类并且实现loadClass（）方法。在版本1.2中，ClassLoader包含了一个具体的loadClass（）实现，这个具体的loadClass（）支持版本1.2提出的双亲委托模型，并且使编写自定义的类装载器变得更加容易，以及减少了出错的机会。在版本1.2中创建一个自定义类装载器时，可以创建ClassLoader的子类，并且除了覆盖loadClass（），还可以选择覆盖findClass（）——这是一个比loadClass（）简单得多的方法。创建用户自定义的类装载器的方法将在本章后面部分讲到。

为了让读者对类装载器在版本1和版本2之间的变化有一个纵向的了解，我们研究一下GreeterClassLoader的实现，这是在本书的第一版中包含的。

```
// On CD-ROM in file
```

```
// linking/ex6/COM/artima/greeter/GreeterClassLoader.java
package COM.artima.greeter;

import java.io.*;
import java.util.Hashtable;

public class GreeterClassLoader extends ClassLoader {

    // basePath gives the path to which this class
    // loader appends "/<typename>.class" to get the
    // full path name of the class file to load
    private String basePath;

    public GreeterClassLoader(String basePath) {

        this.basePath = basePath;
    }

    public synchronized Class loadClass(String className,
        boolean resolveIt) throws ClassNotFoundException {

        Class result;
        byte classData[];

        // Check the loaded class cache
        result = findLoadedClass(className);
        if (result != null) {
            // Return a cached class
            return result;
        }

        // Check with the primordial class loader
        try {
            result = super.findSystemClass(className);
            // Return a system class
            return result;
        }
        catch (ClassNotFoundException e) {
        }

        // Don't attempt to load a system file except through
        // the primordial class loader
        if (className.startsWith("java.")) {
            throw new ClassNotFoundException();
        }
```

```
        // Try to load it from the basePath directory.
        classData = getTypeFromBasePath(className);
        if (classData == null) {
            System.out.println("GCL - Can't load class: "
                + className);
            throw new ClassNotFoundException();
        }

        // Parse it
        result = defineClass(className, classData, 0,
            classData.length);
        if (result == null) {
            System.out.println("GCL - Class format error: "
                + className);
            throw new ClassFormatError();
        }

        if (resolveIt) {
            resolveClass(result);
        }

        // Return class from basePath directory
        return result;
    }

    private byte[] getTypeFromBasePath(String typeName) {

        FileInputStream fis;
        String fileName = basePath + File.separatorChar
            + typeName.replace('.', File.separatorChar)
            + ".class";

        try {
            fis = new FileInputStream(fileName);
        }
        catch (FileNotFoundException e) {
            return null;
        }

        BufferedInputStream bis = new BufferedInputStream(fis);

        ByteArrayOutputStream out = new ByteArrayOutputStream();

        try {
            int c = bis.read();
            while (c != -1) {
```

```
                out.write(c);
                c = bis.read();
            }
        }
        catch (IOException e) {
            return null;
        }

        return out.toByteArray();
    }
}
```

版本1.1的GreeterClassLoader声明了一个实例变量——basePath。这个变量是一个字符串变量，用来保存目录路径（从GreetingClassLoader的构造方法传来），loadClass（）方法应该在这个路径中搜索任何被请求调用的类型。

loadClass（）方法从检查被请求调用的类型是否已经被自己装载过了开始。这是通过调用findLoadedClass（）实现的，后者是ClassLoader的一个方法，传递被请求的类型的全限定名作为参数。如果这个类装载已经被标记为是这个具有该全限定名的类型的初始类装载器，findLoadedClass（）就会返回表示这个类型的Class实例。

```
// Check the loaded class cache
result = findLoadedClass(className);
if (result != null) {
    // Return a cached class
    return result;
}
```

本章前面曾提到过，虚拟机会为每一个类装载器维护一张列表，列表中是已经被请求过的类型的名字。这些列表包含了每一个类装载器被标记为初始类装载器的类型，它们代表了每一个类装载器的命名空间里当前填充的惟一名字的集合。当在解析CONSTANT_Class_info入口的步骤1a装载类的时候（在本章前面讲到的），虚拟机总是会在调用loadClass（）之前检查这个内部列表。这样，虚拟机永远不会自动在同一个用户自定义类装载器上调用同一个名字的类型两次。然而，GreeterClassLoader调用了findLoadedClass（）用内部已装载的类型名字列表检查被请求的类型，这是为什么呢？因为，虽然虚拟机永远不会要求用户自定义的类装载器装载两次同样的类，程序却可能这样做。

举例来说，假设Greet程序是这样被命令行调用的：

```
java Greet greeters Hello Hello Hello Hello Hello
```

在这个命令行中，Greet程序会使用同样的名字Hello在同一个GreeterClassLoader类装载器对象上调用loadClass（）五次。第一次，GreeterClassLoader会装载这个类；但是以后的4次，GreeterClassLoader会通过调用findLoadedClass（）简单地把Hello类的Class实例返回。它只会装载Hello类一次。

如果loadClass（）方法认定被请求的类型没有被装载进它的命名空间，它下一步会传递被

请求的类型给findSystemClass ( ):

```
// Check with the primordial class loader
try {
    result = super.findSystemClass(className);
    // Return a system class
    return result;
}
catch (ClassNotFoundException e) {
}
```

当findSystemClass ( ) 方法在1.1版虚拟机中被调用的时候，原始的类装载器试图装载这个类型。在版本1.2中，系统类装载器试图装载这个类型。如果装载成功了，findSystemClass ( ) 会返回代表这个类型的Class实例，loadClass ( ) 返回同样的Class实例。

如果原始的类装载器（在版本1.1中）或者系统的类装载器（在版本1.2中）不能够装载这个类型，findSystemClass ( ) 就抛出ClassNotFoundError异常。在这个例子中，loadClass ( ) 方法检查确认被请求的类不是java包的一部分：

```
// Don't attempt to load a system file except through
// the primordial class loader
if (className.startsWith("java.")) {
    throw new ClassNotFoundException();
}
```

这个检查可以防止标准的java包（java.lang，java.io，等等）被除了启动类装载器之外的其他类装载器装载。在第3章中讲过，在同样的命名包中声明的两个类型只能拥有访问各自包内可视的成员，假如它们属于同一个运行时包（如果它们被同一个类装载器装载）。但是“运行时包”这个概念及其在可访问性上的作用是在Java虚拟机规范第2版中才引入的。也就是说，早期版本的类装载器不得不自行显式地防止用户自定义的类装载器试图装载某些类型，它们声明自己是Java API（或者任何其他“受限的”包）的一部分，但是又不能被启动类装载器装载。

如果类名字不是由java.开始的，loadClass ( ) 方法下面会调用getTypeFromBasePath ( )，getTypeFromBasePath试图用用户自定义的类装载器的定制方式装载二进制数据。

```
// Try to load it from the basePath directory.
classData = getTypeFromBasePath(className);
if (classData == null) {
    throw new ClassNotFoundException();
}
```

getTypeFromBasePath ( ) 方法在基础路径中查找类型名带有“.class”后缀的文件。（这个基础路径是传递到GreeterClassLoader的构造方法中的。）如果getTypeFromBasePath ( ) 方法无法找到这个文件，它返回一个null结果，并且loadClass ( ) 方法抛出ClassNotFoundException异常。否则，loadClass ( ) 使用getTypeFromBasePath ( ) 返回的字节数组作为参数调用defineClass ( ):

```
// Parse it
```

```
result = defineClass(className, classData, 0,
    classData.length);
if (result == null) {
    System.out.println("GCL - Class format error: "
        + className);
    throw new ClassFormatError();
}
```

defineClass（）方法通过以下步骤完成装载过程：即解析二进制数据到内部数据格式，并且创建一个Class实例。defineClass（）方法并不连接和初始化类型。（在本章前面提到过，defineClass（）方法也会确认所有该类型的超类型也被装载。这是通过在每一个直接超类和超接口上调用这个用户自定义的类装载器的loadClass（）方法实现的，层次结构中的所有超类型的解析过程是一个递归过程。）

如果defineClass（）成功了，loadClass（）方法检查resolve参数是否是true。如果是，它调用resolveClass（），把defineClass（）返回的Class实例作为参数。resolveClass（）方法连接该类。最终，loadClass（）返回新创建的Class实例：

```
if (resolveIt) {
    resolveClass(result);
}

// Return class from basePath directory
return result;
```

8.1.17 使用1.2版本的用户自定义类装载器

前一节描述的类装载器最初是为1.1版虚拟机设计的，但也可以在1.2版本中工作。虽然1.2版本为java.lang.ClassLoader加入了默认的loadClass（）具体实现，这个具体的方法仍然可以被子类所覆盖。因为loadClass（）的工作方式从版本1.1到版本1.2没有变化，所以老式的采取覆盖loadClass（）方式的用户自定义类装载器仍然可以在版本1.2中正常工作。

loadClass（）的基本工作方式是：给定需要查找的类型的全限定名，loadClass（）方法会用某种方式找到或者生成字节数组，里面的数据采用Java class文件的格式（用该格式定义类型）。如果loadClass（）无法找到或者生成这些字节，它会抛出ClassNotFoundException异常。否则，loadClass（）会传递这个字节数组到类ClassLoader声明的某一个defineClass（）方法。通过把字节数组传递给defineClass（），loadClass（）会要求虚拟机把传入的字节数组导入到这个用户自定义的类装载器的命名空间中去。在版本1.2中，当loadClass（）调用defineClass（）的时候，它可以指定这个类型数据所属的保护域。当类装载器的loadClass（）方法成功地装载了类型时，它返回代表这个新装载的类型的java.lang.Class对象。

java.lang.ClassLoader类中的loadClass（）的具体实现通过如下步骤来实现loadClass（）方法的工作方式。

1）查看是否请求的类型已经被这个类装载器装载进命名空间了（通过findLoadedClass（）方法）。如果的确如此，返回这个已经装载的类型的Class实例。

2）否则，委派到这个类装载器的双亲装载器。如果双亲返回了一个Class实例，就把这个

Class实例返回。

3) 否则，调用findClass（），findClass（）会试图寻找或者生成一个字节数组，内容采用Java class文件格式（它定义了所需的类型）。如果成功，findClass（）把这个字节传递给defineClass（），后者试着导入这个类型，返回一个Class实例。如果findClass（）返回了一个Class实例，loadClass（）就把这个Class实例返回。

4) 否则，findClass（）抛出某些异常来中止处理，而且loadClass（）也抛出同样的异常并中止。

虽然在版本1.2中仍然可以生成ClassLoader的子类并且覆盖loadClass（）方法，但是在版本1.2中创建自定义的类装载器时，推荐采用生成ClassLoader的子类并实现findClass（）方法的方式。findClass（）方法看上去像这样：

```
// A method declared in class java.lang.ClassLoader:
protected Class findClass(String name)
    throws ClassNotFoundException;
```

findClass（）方法的基本工作方式是：findClass（）接受需要装载的类型的全限定名作为惟一的参数。findClass（）首先试图查找或者生成一个字节数组，内容是Java class文件格式（格式定义了所需要装载的类型）。如果findClass（）无法确定或者生成字节数组，它抛出ClassNotFoundException异常并中止。否则，findClass（）调用defineClass（），把所需的类型名字、字节数组和一个可选的指定了这个类型所属的受保护域的ProtectionDomain对象作为参数。如果defineClass（）返回了一个代表这个类型的Class实例，findClass（）简单地把同一个Class实例返回给它的调用者。否则，defineClass（）抛出某些异常并中止，findClass（）也抛出同样的异常并中止。

下面是一个采取覆盖findClass（）方法（而不是覆盖loadClass（）方法）的GreeterClassLoader版本：

```
// On CD-ROM in file
// linking/ex7/com/artima/greeter/GreeterClassLoader.java
package com.artima.greeter;

import java.io.*;

public class GreeterClassLoader extends ClassLoader {

    // basePath gives the path to which this class
    // loader appends "/<typename>.class" to get the
    // full path name of the class file to load
    private String basePath;

    public GreeterClassLoader(String basePath) {

        this.basePath = basePath;
    }
```

```
public GreeterClassLoader(ClassLoader parent, String basePath) {

    super(parent);
    this.basePath = basePath;
}

protected Class findClass(String className)
    throws ClassNotFoundException {

    byte classData[];

    // Try to load it from the basePath directory.
    classData = getTypeFromBasePath(className);
    if (classData == null) {
        throw new ClassNotFoundException();
    }

    // Parse it
    return defineClass(className, classData, 0,
        classData.length);
}

private byte[] getTypeFromBasePath(String typeName) {

    FileInputStream fis;
    String fileName = basePath + File.separatorChar
        + typeName.replace('.', File.separatorChar)
        + ".class";

    try {
        fis = new FileInputStream(fileName);
    }
    catch (FileNotFoundException e) {
        return null;
    }

    BufferedInputStream bis = new BufferedInputStream(fis);

    ByteArrayOutputStream out = new ByteArrayOutputStream();

    try {
        int c = bis.read();
        while (c != -1) {
            out.write(c);
            c = bis.read();
        }
```

```
            }
            catch (IOException e) {
                return null;
            }

            return out.toByteArray();
        }
    }
```

该GreeterClassLoader版本位于随书CD-ROM的linking/ex7目录中。所有的源代码文件都在linking/ex6目录，在前一节我们已经详细讨论过它们了。除了GreeterClassLoader.java，其他文件都没有变动。前一节描述的linking/ex6下的GreeterClassLoader类采取的是覆盖loadClass（）方法，而本节描述的linking/ex7下的GreeterClassLoader采取的是覆盖findClass（）方法。

这个GreeterClassLoader的第2版本声明了一个实例变量basePath，这是一个字符串变量，用来保存一个目录路径，findClass（）会在这个路径中搜索要求装载的类型的class文件。basePath字符串变量是GreeterClassLoader的单参数构造方法所需的惟一参数。因为这个单参数的构造方法没有接受指向调用者定义的双亲类装载器的引用，所以这个类装载器无法调用使用用户自定义类装载器引用的超类构造方法。也就是说，它简单地默认调用超类的无参数版本的构造方法，这会使这个类装载器的双亲设置为系统类装载器。另外一个构造方法（两参数构造方法）除了basePath字符串变量之外还接受一个指向用户自定义的类装载器实例的引用。这个构造方法可以明确地调用超类的单参数构造方法，传递这个引用。超类设置这个类装载器的双亲为传递来的这个用户自定义类装载器的实例。

比较一下GreeterClassLoader这个版本的findClass（）实现和GreeterClassLoader上一个版本的loadClass（）实现，很容易发现编写findClass（）比编写loadClass（）容易得多。编写findClass（）的时候，需要担心的可能出错的地方要少得多。findClass（）只不过是调用getTypeFromBasePath（）来试图用这个自定义类装载器的方式装载所需的类型。如果getTypeFromBasePath无法在basePath目录找到所需的类型，它返回null，findClass（）方法就抛出ClassNotFoundException异常。否则，getTypeFromBasePath（）返回字节数组，findClass（）简单地把字节数组传递到defineClass（）。如果defineClass（）返回指向Class实例的引用来代表成功装载的类型，findClass（）返回同一个引用。否则，defineClass（）抛出某个异常并中止，并导致findClass（）抛出同样的异常并中止。

findClass（）方法的工作方式是loadClass（）方法的工作方式的子集。findClass（）把loadClass（）的两个部分分离开来（这两个部分通过创建java.lang.ClassLoader的子类一般可以定制）：

1) 给定一个类型的全限定名，用定制化的方法查找或者生成一个字节数组。

2) 可选的，用定制化的方式决定一个类型的保护域。

findClass（）的实现需要执行这两个任务，结果是一个字节数组和一个ProtectionDomain对象的引用。findClass（）把字节数组和ProtectionDomain的引用都传递给defineClass（）。

8.1.18 示例：使用forName（）的动态扩展

使用forName（）完成动态扩展的Java程序例子，如下面的EasyGreet类：

```
// On CD-ROM in file linking/ex7/EasyGreet.java
import com.artima.greeter.*;

public class EasyGreet {

    // Arguments to this application:
    //     args[0], args[1], ... - class names of greeters to load
    //                and invoke the greet() method on.
    //
    // All greeters must implement the com.artima.greeter.Greeter
    // interface.
    //
    static public void main(String[] args) {

        if (args.length == 0) {
            System.out.println(
                "Enter greeter class names as args.");
            return;
        }

        for (int i = 0; i < args.length; ++i) {
            try {

                // Load the greeter specified on the command line
                Class c = Class.forName(args[i]);

                // Instantiate it into a greeter object
                Object o = c.newInstance();

                // Cast the Object ref to the Greeter interface type
                // so greet() can be invoked on it
                Greeter greeter = (Greeter) o;

                // Greet the world in this greeter's special way
                greeter.greet();
            }
            catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

EasyGreet程序和上一个例子中的Greet程序非常类似。像Greet那样，EasyGreet试图使用命令行参数中指定的名字动态装载并执行greeter。但是和Greet不同，EasyGreet并不使用第一个命令行参数作为greeter的class文件存放的目录。所有的EasyGreet的命令行参数都是greeter类的名

字。另一个区别是因为EasyGreeter将使用forName（）动态装载greeter，它不需要创建一个GreeterClassLoader实例。还有，Greet调用它的GreeterClassLoader实例的loadClass（）方法，而EasyGreet调用forName（）——这是类Class的一个静态方法。

EasyGreet的forName（）调用和Greet的loadClass（）调用看上去非常相似。与loadClass（）类似，forName（）用它的字符串参数接受所需的类型的全限定名。如果成功地装载了类型（或者类型早先已经被装载过了），forName和loadClass（）一样返回代表类型的Class实例。如果不成功，forName（）和loadClass（）一样抛出ClassNotFoundException异常。两种方式最大的不同之处在于，loadClass（）试图保证被装载的类型是被装载到用户自定义的类装载器的命名空间里，而forName（）试图确认所需的类型被装载到当前命名空间中，这个当前命名空间就是类型（该类型的方法包括forName（）调用）所属的定义类装载器的命名空间。

因为forName（）从类EasyGreet的main（）方法中被调用，forName（）请求装载所需类型的类装载器就是EasyGreet的定义类装载器。当在Sun的Java2 SDK 1.2版本下运行的时候，装载EasyGreet的就是系统类装载器，它会在类路径中寻找类。要使用类路径环境变量，可以通过一个如下的命令行执行随书光盘中linking/ex7目录下的EasyGreet程序：

```
java EasyGreet Hello
```

如果没有明确地在命令行或者环境变量中指定一个类路径，系统类装载器会在当前目录中寻找所需的类型。因为当前目录（随书光盘的linking/ex7目录）中没有包含Hello.class，系统类装载器无法找到Hello.class。forName（）方法抛出ClassNotFoundException异常并中止，随后，EasyGreet的main（）方法也同样抛出该异常并中止：

```
java.lang.ClassNotFoundException: Hello
        at java.net.URLClassLoader$1.run(URLClassLoader.java: 202)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(URLClassLoader.java: 191)
        at java.lang.ClassLoader.loadClass(ClassLoader.java: 290)
        at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java: 275)
        at java.lang.ClassLoader.loadClass(ClassLoader.java: 247)
        at java.lang.Class.forName0(Native Method)
        at java.lang.Class.forName(Class.java: 124)
        at EasyGreet.main(EasyGreet.java, Compiled Code)
```

为了能让EasyGreet找到Hello.class，只需要在类路径中包含greeters目录，这可以通过在命令行使用“-cp”选项指定，如下所示：

```
java -cp .;greeters; EasyGreet Hello
```

用这个命令启动，EasyGreet程序打印出：

```
Hello, world!
```

如同Greet方法一样，EasyGreet可以在命令行接受多个greeter的名字：

```
java -cp .;greeters; EasyGreet Hello Greetings Salutations HowDoYouDo
```

使用这个命令调用，EasyGreet程序会装载列出的四个欢迎者并且调用它们的greet（）方法，

输出如下：

```
Hello, world!
Greetings, planet!
Salutations, orb!
How do you do, globe!
```

使用Greet中GreeterClassLoader的loadClass（）方法与EasyGreet中使用forName（）方法的最重要的区别就是，被装载的greeter类所被装载入的命名空间不同。在Greet里，greeter类被装载到GreeterClassLoader的命名空间；在EasyGreet里面，greeter类被装载到系统类装载器的命名空间。

8.1.19 示例：卸载无法触及的greeter类

动态装载的类型变为无法触及而要被虚拟机卸载时的例子如下面的程序：

```
// On CD-ROM in file linking/ex7/GreetAndForget.java
import com.artima.greeter.*;

public class GreetAndForget {

    // Arguments to this application:
    //     args[0] - path name of directory in which class files
    //               for greeters are stored
    //     args[1], args[2], ... - class names of greeters to load
    //               and invoke the greet() method on.
    //
    // All greeters must implement the com.artima.greeter.Greeter
    // interface.
    //
    static public void main(String[] args) {

        if (args.length <= 1) {
            System.out.println(
                "Enter base path and greeter class names as args.");
            return;
        }

        for (int i = 1; i < args.length; ++i) {
            try {

                GreeterClassLoader gcl =
                    new GreeterClassLoader(args[0]);

                // Load the greeter specified on the command line
                Class c = gcl.loadClass(args[i]);

                // Instantiate it into a greeter object
```

```
                Object o = c.newInstance();

                // Cast the Object ref to the Greeter interface type
                // so greet() can be invoked on it
                Greeter greeter = (Greeter) o;

                // Greet the world in this greeter's special way
                greeter.greet();

                // Forget the class loader object, Class
                // instance, and greeter object
                gcl = null;
                c = null;
                o = null;
                greeter = null;

                // At this point, the types loaded through the
                // GreeterClassLoader object created at the top of
                // this for loop are unreferenced and can be unloaded
                // by the virtual machine.
            }
            catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

GreetAndForget程序接受和前面的Greet示例程序同样的命令行参数。第一个参数是一个基础目录路径名，GreetAndForget程序会在其中寻找greeter对象。后面的参数是greeter对象的名字。要理解这个例子，必须熟悉本章前面所描述的Greet程序。

假设使用如下命令行调用GreetAndForget程序：

```
java GreetAndForget greeters Surprise HiTime Surprise
```

HiTime会根据一天中的不同时间选择不同的欢迎词，在本章前一节中演示过。Surprise伪随机地选择四个greeter对象之一——Hello， Greetings，Salutations，HowDoYouDo——并且调用它的greet（）方法，它的源代码如下所示：

```
// On CD-ROM in file linking/ex7/greeters/Surprise.java
import com.artima.greeter.Greeter;

public class Surprise implements Greeter {

    public void greet() {

        // Choose one of four greeters pseudo-randomly and
```

```
        // invoke its greet() method.
        int choice = (int) (Math.random() * 3.99);

        Greeter g;

        switch(choice) {

        case 0:
            g = new Hello();
            g.greet();
            break;

        case 1:
            g = new Greetings();
            g.greet();
            break;

        case 2:
            g = new Salutations();
            g.greet();
            break;

        case 3:
            g = new HowDoYouDo();
            g.greet();
            break;
        }
    }
}
```

假定使用上面指定的命令行，GreetAndForget程序首先调用Surprise的greet（）方法，然后是HiTime，再然后又是Surprise。GreetAndForget实际输出会根据一天当中的时间和Surprise的伪随机结果而变化。为了演示这个示例，假设你敲入了上面的命令行，按下“回车”键，会得到下面的输出：

```
How do you do, globe!
Good afternoon, world!
Greetings, planet!
```

这个输出表示Surprise首先选择了执行HowDoYouDo的greet（）方法，第二次则选择了执行Greetings的greet（）方法。

GreetAndForget的第一次for循环，虚拟机装载了Surprise类并且调用它的greet（）方法。Surprise的常量池包含指向它可能选择的4个greeter助手的符号引用：Hello，Greetings，Salutations以及HowDoYouDo。假设你用来运行GreetAndForget的Java虚拟机使用迟解析方式，在GreetAndForget的for循环第一次执行的时候只有一个符号引用被解析了，即指向

HowDoYouDo的符号引用。虚拟机在执行字节码遇到了Surprise的greet（）方法的这个语句时解析这个符号引用：

```
g = new HowDoYouDo();
```

要解析从Surprise的常量池指向HowDoYouDo的这个符号引用，虚拟机调用GreeterClassLoader对象的loadClass（）方法，在name参数中传递“HowDoYouDo”这个字符串。虚拟机使用GreeterClassLoader对象来装载HowDoYouDo，因为Surprise也是通过GreeterClassLoader对象来装载的。在本章前面提到过，当Java虚拟机解析一个符号引用的时候，它使用和定义引用类型（这里是Surprise）的同一个类装载器来初始装载被引用的类型（这里是HowDoYouDo）。

当Surprise的greet（）方法创建了一个新的HowDoYouDo实例的时候，它调用后者的greet（）方法：

```
g.greet();
```

当虚拟机执行HowDoYouDo的greet（）方法的时候，它必须解析HowDoYouDo的常量池中的两个符号引用：一个是指向类java.lang.System的，另一个是指向类java.io.PrintStream的。要解析这些符号引用，虚拟机调用GreeterClassLoader的loadClass（），分别使用java.lang.System和java.io.PrintStream作为name参数。和前面一样，虚拟机使用GreeterClassLoader来装载这些类，因为引用的类（这里是HowDoYouDo）是被GreeterClassLoader对象装载的。这两个类都是Java API的一部分，不管怎样都会最终被启动类装载器装载，因为loadClass（）首先会把请求委派给它的双亲。

记住在GreeterClassLoader的loadClass（）方法试图自行在基础目录（这里是greeters目录）中搜索所请求的类型之前，它会调用它的双亲——系统类装载器。系统类装载器也会先委派给它的双亲，它的双亲也会再委派给它自己的双亲，这样不断重复。最终findSystemClass（）被调用，委派给了启动类装载器，委派链到了终点。因为启动类装载器（通过findSystemClass（））能够装载java.lang.System和java.io.PrintStream，loadClass（）方法简单地返回从findSystemClass（）返回的Class实例。这些类不会被标记为已由GreeterClassLoader对象定义，而是被标记为已由启动类装载器定义。如果要解析任何从java.lang.System或者java.io.PrintStream发起的引用，虚拟机不会调用GreeterClassLoader对象的loadClass（）方法，也不是系统类装载器，它会直接使用启动类装载器。

在Surprise的greet（）方法返回后，有两个类型被标记为已由GreeterClassLoader对象定义：类Surprise和类HowDoYouDo。这两个类型在虚拟机中位于由GreeterClassLoader对象定义的类型列表中。

Surprise的greet（）方法一返回，Surprise和HowDoYouDo的Class实例就可以被程序所触及了。垃圾收集器不会回收这些Class实例占据的空间，因为程序代码能够访问并且使用它们。图8-11是关于这两个Class实例的可触及性的图形化描述。

Surprise的Class实例可以通过两条途径触及。首先，GreetAndForget的main（）方法中的局部变量c可以直接触及它。其次，它可以通过局部变量o和greeter触及，它们指向的是同一个Surprise对象。通过Surprise对象，虚拟机可以访问Surprise对象的类型数据，类型数据中包含指

向Surprise类对象的引用。第三种方法是通过GreetAndForget的main（）方法的gcl局部变量访问。这个局部变量指向GreeterClassLoader对象，而后者包含了一个HashTable对象，其中保存了一个指向Surprise的类实例的引用。

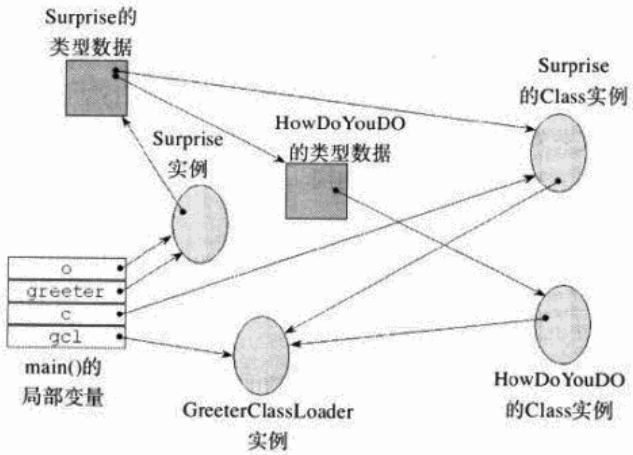


图8-11 Surprise和HowDoYouDo的类实例的可触及性

HowDoYouDo的类实例可以通过两条途径触及。它的第一条途径也是触及Surprise的途径中的一条：GreetAndForget的main（）方法中的gcl局部变量指向GreeterClassLoader对象，后者包含了一个HashTable对象的引用。而HashTable对象包含了一个指向HowDoYouDo的Class实例的引用。触及HowDoYouDo的类实例的第二条途径是通过Surprise的常量池。当虚拟机解析完从Surprise的常量池指向HowDoYouDo的符号引用后，它用一个直接引用替换符号引用。这个直接引用指向HowDoYouDo的类型数据，类型数据中包含指向HowDoYouDo的Class实例的引用。

因此，从Surprise的常量池开始，HowDoYouDo的Class实例是可以触及的。但是为什么垃圾收集器要首先注意从Surprise的常量池发出的直接引用呢？因为Surprise的Class实例是可以引用的。当垃圾收集器发现它可以触及Surprise的Class实例时，它必须确信从Surprise的常量池中发出的任何直接引用指向的类型都是可触及的。只要Surprise仍然是活动的，虚拟机就不能卸载任何Surprise可能使用的类型。

请注意，在上面指出的可以触及Surprise的途径中，没有任何一个涉及到其他类型的常量池。Surprise在GreetAndForget的常量池中没有作为符号引用出现。GreetAndForget在编译的时候不知道有Surprise的存在。GreetAndForget程序是在运行时决定装载并连接类Surprise的。因此，类Surprise的Class实例只能够从GreetAndForget的main（）方法的局部变量触及。对Surprise来说不好的是（最终对HowDoYouDo也是一样），这根维系生命的“稻草”并不是很牢固。

GreetAndForget的main（）方法中的以下4条语句，将会完全改变可触及状况：

```
// Forget the user-defined class loader, Class
// instance, and greeter object
gcl = null;
c = null;
```

```
o = null;
greeter = null;
```

这四条语句把可以触及Surprise的Class实例的四个起点全都变成了null。结果就是，这四条语句执行后，Surprise的Class实例再也不能够被触及了。这些语句也使得HowDoYouDo的Class实例无法触及。原来使o和greeter变量指向Surprise实例，gcl变量指向GreeterClassLoader实例，GreeterClassLoader对象的类变量指向HashTable实例。现在这4个对象都可以被当作垃圾收集了。

当垃圾收集器处理并且释放不再被引用的Surprise和HowDoYouDo的Class实例的时候，它可以同时释放Surprise和HowDoYouDo在方法区中相关联的类型数据。因为这些类的Class实例是不可触及的，这些类型自己也是不可触及的了，可以被虚拟机卸载。

请注意，for循环语句再经过两次循环（假设使用上面的命令行），GreetAndForget程序会再一次装载类Surprise。请记住虚拟机并不会重用第一次循环中装载的Surprise的类型数据。在第一遍循环的终点，那个类型数据已经被授权卸载了。但是就算Surprise的Class实例在第一遍循环的终点还没有变成不再被引用，第三遍循环也并不会重用第一遍循环使用的类型数据。

每一次循环，GreetAndForget的main（）方法都创建一个新的GreeterClassLoader对象。因此，每一个被GreetAndForget装载的greeter都是被一个不同的用户自定义类装载器装载的。比如，如果使用那个列举了5次Hello欢迎者的命令行来调用GreetAndForget程序，程序会创建类GreeterClassLoader的5个实例。Hello欢迎者会被5个不同的用户自定义类装载器装载5次。方法区会保存Hello的类型数据的5份拷贝。栈中会保存代表Hello类的5个Class实例——每一个装载了Hello的命名空间都有一个。当其中的一个Hello的Class实例变成不再被引用的时候，只有与这个特定的Class实例相关联的Hello类型数据会被准许卸载。

8.1.20 示例：类型安全性与装载约束

在Java虚拟机的早期实现中，搞乱Java的类型系统是有可能的。一个Java程序可能欺骗Java虚拟机，用一种类型的对象冒充另一种类型的对象。这种能力让破坏者非常高兴，因为他们可以欺骗受信任的类非法访问非公开的数据，或者通过把类替换成新的版本改变方法的行为。比如，如果一个破坏者编写了一个类，并且成功地愚弄Java虚拟机，让后者认为这个类是SecurityManager类，这样破坏者就可能突破整个沙箱。本节给出的示例用来帮助读者理解委派类装载器可能带来的类型安全性问题，并说明Java虚拟机规范第2版中提出的装载约束是如何解决这个问题的。

类型安全性问题的出现是因为在一个Java虚拟机中的多个命名空间可能共享类型。如果某个类装载器委派另外一个类装载器，而后者定义了这个类型，这两个类装载器都会被标记为这个类型的初始类装载器。被委派的类装载器装载的这个类型，在所有被标记为该类型的初始类装载器的命名空间中共享。

在编译时，类型被它的全限定名所惟一确定。比如说，只有一个名为Spoofed的类能够在编译时存在。然而在运行时，仅仅一个全限定名已经不足以惟一地确定被Java虚拟机装载的类型了。因为一个Java程序可能拥有多个类装载器，每个类装载器独有自己的命名空间，具有相同全限定名的多个类型可能被装载进同一个Java虚拟机。因此，要惟一地确定一个类型，Java虚拟机需要知道类型的全限定名以及这个类型的定义类装载器。

正是因为Java虚拟机早期对编译时一个全限定名可以惟一地确认一个类型的信任，才带来了这种类装载器结构上可能存在的安全性问题。可以在同一个Java虚拟机里面装载两个类型，它们的名字都叫做Spoofed，每一个Spoofed类是用一个不同的类装载器定义的。但是要一些小手腕，就可以愚弄一个早期的Java虚拟机实现，让它认为其中一个Spoofed的实例是另一个Spoofed的实例。

为了解决这个问题，Java虚拟机规范第2版中引入了装载约束的概念。从根本上来说，装载约束可以让Java虚拟机加强类型安全性，它不仅仅基于全限定名，也基于定义类装载器——而并非强迫更多的类装载。当虚拟机在常量池解析时发现潜在的类型混淆时，它会在一个内部约束列表上加上一个约束。以后所有的解析必须满足这个新约束，如同必须满足这个列表上其他的所有约束一样。

类型混淆问题及其装载约束解决方案的例子如下所示，考虑下面的greeter实现，这是由一个恶意破坏者编写的：

```
// On CD-ROM in file linking/ex8/greeters/Cracker.java
import com.artima.greeter.Greeter;

public class Cracker implements Greeter {

    public void greet() {

        Spoofed spoofed = new Spoofed();

        System.out.println("secret val = "
            + spoofed.giveMeFive());

        spoofed = Delegated.getSpoofed();

        System.out.println("secret val = "
            + spoofed.giveMeFive());
    }
}
```

如同以前例子中的Hello或者Salutations一样，类Cracker是一个greeter，因为它实现了com.artima.greeter.Greeter接口。读者可以在随书光盘的linking/ex8目录中找到类Cracker。

除了GreeterClassLoader之外，其他所有linking/ex7目录中的类和在linking/ex8中都没有变化，GreeterClassLoader被稍稍改写了。（后面还会做更多的改写。）可以像其他greeter类一样调用Cracker的greet（）方法。在linking/ex8目录下输入：

```
java Greet greeters Cracker
```

Greet的main（）方法如同在前面的几个例子中那样，创建一个GreeterClassLoader，调用它的loadClass（）方法，把Cracker作为名字传递进取。GreeterClassLoader的loadClass（）方法会在greeters目录中查找、装载Cracker.class，初始化一个新的Cracker对象，调用它的greet（）方法。Cracker的greet（）方法从创建一个新的Spoofed对象开始。这也是阴谋的开始。

最后，有两个名为Spoofed的类的实现。在linking/ex8目录下的是“受信任”的实现，系统

类装载器会发现它：

```
// On CD-ROM in file linking/ex8/Spoofed.java
// Trusted version - when asked to give five, gives 5

public class Spoofed {

    private int secretValue = 42;

    public int giveMeFive() {
        return 5;
    }

    static {
        System.out.println(
            "linking/ex8/Spoofed initialized.");
    }
}
```

受信任的Spoofed声明了一个名为secretValue的私有变量，初始值为42。这个私有变量代表任何可能需要保密的数据：信用卡号码，私有密钥，电子现金，指向当前Policy对象的引用，等等。因为这个类的设计者不希望整个世界都知道这个秘密值，他把这个secretValue声明为私有的，只有Spoofed的方法可以操作secretValue。如果检查受信任的Spoofed类的代码，可以看到Spoofed的设计者没有提供任何方法来暴露secretValue的值。Spoofed中惟一的方法是giveMeFive( )，返回固定值5。

但是，如果一个狡猾的破坏者能够欺骗虚拟机，声称这个受信任的Spoofed其实是另一个类，名字也是Spoofed的一个实例，事情会怎么样呢？另外一个由破坏者编写的也称为Spoofed的代码如下：

```
// On CD-ROM in file linking/ex8/greeters/Spoofed.java
// Malicious version - when asked to give five, this
//     version of Spoofed reveals secret_value

public class Spoofed {

    private int secretValue = 100;

    public int giveMeFive() {
        return secretValue;
    }

    static {
        System.out.println(
            "linking/ex8/greeters/Spoofed initialized.");
    }
}
```

当这个Spoofed类的giveMeFive（）方法被调用的时候，它返回secretValue，这样就让这个私有的变量尽人皆知了。

那么Cracker会使用哪个Spoofed版本呢？Cracker迂回地试图两个都使用。首先，Cracker的greet（）方法装载恶意的Spoofed，并且调用它的greet（）方法，如下所示：

```
Spoofed spoofed = new Spoofed();

System.out.println("secret val = "
    + spoofed.giveMeFive());
```

Java编译器把这个new Spoofed（）表达式翻译成一段字节码指令，它给出了CONSTANT_Class_info常量池入口的索引，常量池入口代表了指向Spoofed的符号引用。当虚拟机解析这个引用的时候，它会要求Craker的定义装载器装载Spoofed。Crakcer的定义装载器是修改过的GreeterClassLoader版本，破坏者有机会这样修改：

```
// On CD-ROM in file
// linking/ex8/COM/artima/greeter/GreeterClassLoader.java
package com.artima.greeter;

import java.io.*;
import java.util.Hashtable;

public class GreeterClassLoader extends ClassLoader {

    // basePath gives the path to which this class
    // loader appends "/<typename>.class" to get the
    // full path name of the class file to load
    private String basePath;

    public GreeterClassLoader(String basePath) {

        this.basePath = basePath;
    }

    public synchronized Class loadClass(String className,
        boolean resolveIt) throws ClassNotFoundException {

        Class result;
        byte classData[];

        // Check the loaded class cache
        result = findLoadedClass(className);
        if (result != null) {
            // Return a cached class
            return result;
        }
```

```
        // If Spoofed, don't delegate
        if (className.compareTo("Spoofed") != 0) {

            // Check with the system class loader
            try {
                result = super.findSystemClass(className);
                // Return a system class
                return result;
            }
            catch (ClassNotFoundException e) {
            }
        }

        // Don't attempt to load a system file except through
        // the primordial class loader
        if (className.startsWith("java.")) {
            throw new ClassNotFoundException();
        }

        // Try to load it from the basePath directory.
        classData = getTypeFromBasePath(className);
        if (classData == null) {
            System.out.println("GCL - Can't load class: "
                + className);
            throw new ClassNotFoundException();
        }

        // Parse it
        result = defineClass(className, classData, 0,
            classData.length);
        if (result == null) {
            System.out.println("GCL - Class format error: "
                + className);
            throw new ClassFormatError();
        }

        if (resolveIt) {
            resolveClass(result);
        }

        // Return class from basePath directory
        return result;
    }

    private byte[] getTypeFromBasePath(String typeName) {
```

```
        FileInputStream fis;
        String fileName = basePath + File.separatorChar
            + typeName.replace('.', File.separatorChar)
            + ".class";

        try {
            fis = new FileInputStream(fileName);
        }
        catch (FileNotFoundException e) {
            return null;
        }

        BufferedInputStream bis = new BufferedInputStream(fis);

        ByteArrayOutputStream out = new ByteArrayOutputStream();

        try {
            int c = bis.read();
            while (c != -1) {
                out.write(c);
                c = bis.read();
            }
        }
        catch (IOException e) {
            return null;
        }

        return out.toByteArray();
    }
}
```

要创建这个用户自定义的类装载器，破坏者只需从CD-ROM的linking/ex6目录中得到GreeterClassLoader（覆盖了loadClass（）的那个），然后加上几行代码：

```
// If Spoofed, don't delegate
if (className.compareTo("Spoofed") != 0) {

    // Check with the system class loader
    try {
        result = super.findSystemClass(className);
        // Return a system class
        return result;
    }
    catch (ClassNotFoundException e) {
    }
}
```

如果传递给loadClass（）的类型名字是“Spoofed”，loadClass（）方法在试图以定制方式（通过查找basePath目录）装载类之前，并不首先委派系统类装载器装载。因此，当虚拟机要求类装载器（破坏者定义的类装载器）装载Spoofed的时候，它的loadClass（）并没有委派，而是直接从basePath目录查找Spoofed.class。当它找到了，它就装载这个恶意的Spoofed的定义。程序打印出：

```
linking/ex8/greeters/Spoofed initialized.
```

Cracker的greet（）方法的下一条语句调用新的Spoofed实例的giveMeFive（）并打印出返回值：

```
secret val = 100
```

破坏者试过了giveMeFive（）方法并感到满意后，就让Cracker的greet（）方法调用名为Delegated类的静态方法，它返回Spoofed类型的引用：

```
spoofed = Delegated.getSpoofed();
```

Java编译器把源代码中的Delegated.getSpoofed（）表达式转换成一个invokestatic字节码指令（它给出一个指向常量池中CONSTANT_Methodref_info入口的索引）。要执行这个指令，虚拟机必须解析这个常量池入口。作为解析指向getSpoofed（）符号引用的第一步，虚拟机需要解析一个CONSTANT_Class_info引用，它的索引在刚才的CONSTANT_Methodref_info入口的class_index域中给出。这个CONSTANT_Class_info入口是一个指向类Delegated的符号引用。

要解析从Cracker到Delegated的符号引用，虚拟机请求Cracker的定义类装载器装载Delegated。虚拟机又一次调用了GreeterClassLoader的loadClass（）方法，这一次传递了Delegated这个名字。然而，因为这一次请求的名字不是"Spoofed"，loadClass（）方法继续向下执行并把这个装载请求委派给了系统类装载器。因为Delegated.class在linking/ex8目录下，系统类装载器能够装载这个类。系统类装载器被标记为Deligated的定义类装载器，而系统类装载器和GreeterClassLoader二者都被标记为初始类装载器。

一旦Delegated被装载了，虚拟机完成了对CONSTANT_Methodref_info的解析，并调用getSpoofed（）方法。Delegated的getSpoofed（）方法如下：

```
// On CD-ROM in file linking/ex8/Delegated.java

public class Delegated {

    public static Spoofed getSpoofed() {

        return new Spoofed();
    }
}
```

这段Java源程序看上去是无害的。getSpoofed（）方法只不过是创建了另一个Spoofed对象的实例并且返回指向它的引用。然而在Java虚拟机内部，却对Java的保证安全类型提出了严峻考验。

当Java编译器在类Delegated中遇到new Spoofed（）表达式的时候，它产生了一个new字节码，给出了CONSTANT_Class_info入口的索引（这个入口是一个指向Spoofed的符号引用）。这

和虚拟机在类Cracker中遇到new Spoofed（）表达式时发生的事情一样。当Java虚拟机执行这句new指令的时候，如同刚才在Cracker的greet（）方法中执行new指令一样，它从解析指向Spoofed的符号引用开始。虚拟机要求Delegated的定义装载器来装载Spoofed，这个装载器就是系统类装载器。

虽然这个过程和虚拟机前面解析Cracker的指向Spoofed的符号引用是完全一样的，但是虚拟机用来装载请求的类装载器却不一样。因为Cracker的定义类装载器是GreeterClassLoader，所以虚拟机要求GreeterClassLoader来装载Spoofed。但是因为Delegated的定义类装载器是系统类装载器，所以现在虚拟机要求系统类装载器来装载Spoofed。

因为受信任的Spoofed版本位于随书光盘的linking/ex8目录下，系统类装载器可以读取Spoofed.class的字节并且传递给defineClass（）。下一步会发生什么就取决于程序是否运行在遵守装载约束的Java虚拟机中（装载约束在Java虚拟机规范第2版中规定）。

假设程序运行在老版本的Java虚拟机实现中，它没有使用装载约束。在这种情形下，defineClass（）可以用从linking/ex2/Spoofed.class读入的字节来定义类型。虚拟机创建了这个受信任的Spoofed类型。很快，Delegated的getSpoofed（）方法返回了一个对受信任的Spoofed对象的引用给它的调用者（即Cracker的greet（）方法）。Cracker把这个引用保存在局部变量spoofed中，进一步打印出调用Spoofed的giveMeFive（）方法的返回值。

在Cracker.java被编译的时候，Java编译器把第二个giveMeFive（）调用转换为另一个invokevirtual指令，这个指令引用了常量池中的CONSTANT_Methodref_info入口：指向Spoofed中giveMeFive（）方法的符号引用。然而当虚拟机解析这个符号引用的时候，它发现这个引用已经被解析过了。第二次giveMeFive（）调用指定的CONSTANT_Methodref_info入口和第一次调用指定的是同一个，都解析为恶意的giveMeFive（）实现。虚拟机在受信任的Spoofed对象上调用了恶意的Spoofed的方法，程序打印出：

```
secret val = 42
```

虽然这种混淆类型进行攻击的可能性在很多1.2版本之前的Java虚拟机中都存在，但是实际上一般不会发生，因为它需要类装载器的配合。在这个例子中，破坏者在GreeterClassLoader的loadClass（）方法中加入了一条if语句来对Spoofed特别处理。但是如果破坏者试图通过一个不受信任的applet来进行这种类型混淆攻击，他或者她就会遇到麻烦。不受信任的applet不能够创建类装载器。因此，在把applet装载进浏览器的应用程序中，假设它的类装载器的设计人员做了正确的处理，破坏者就没有办法利用这个Java类型安全性的弱点。

在进行检查装载约束的虚拟机实现中，这已经是Java虚拟机规范的一部分了，类型混淆根本不可能发生。所有的虚拟机现在都必须保留一张关于已经装载的类型约束的内部列表。比如，当这样的一个虚拟机解析Cracker的常量池中的CONSTANT_Methodref_info入口的时候，如果这个入口是一个指向类Delegated的getSpoofed（）方法的符号引用，虚拟机记下了一个装载约束。因为Delegated和Cracker相比是被另外一个类装载器装载的，而Delegated的getSpoofed（）方法返回了一个指向Spoofed的引用，虚拟机记下了这样一个约束：

系统类装载器（Delegated的定义类装载器）被标记为Spoofed类型的初始类装载器，
GreeterClassLoader（Cracker的定义类装载器）也被标记为Spoofed类型的初始类装载器，

这两个类型必须是同一个类型。

这个约束以后就被用到了，当虚拟机解析Delegated的CONSTANT_Class_info入口，后者指向一个类Spoofed的符号引用时，这一次，虚拟机发现约束被违反了。这个即将要被系统类装载器装载的名为Spoofed的类型并不是那个以前被GreeterClassLoader装载的同名类型。因此，Java虚拟机抛出了一个LinkageError异常：

```
Exception in thread "main" java.lang.LinkageError: Class Spoofed violates loader
constraints
        at java.lang.ClassLoader.defineClass0(Native Method)
        at java.lang.ClassLoader.defineClass(ClassLoader.java:422)
        at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:10)
        at java.net.URLClassLoader.defineClass(URLClassLoader.java:248)
        at java.net.URLClassLoader.access$1(URLClassLoader.java:216)
        at java.net.URLClassLoader$1.run (URLClassLoader.java:197)
        at java.security.AccessController.doPrivileged (Native Method)
        at java.net.URLClassLoader.findClass(URLClassLoader.java:191)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:290)
        at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:275)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:247)
        at Delegated.getSpoofed(Delegated.java, Compiled Code)
        at Cracker.greet(Cracker.java:13)
        at Greet.main(Greet.java, Compiled Code)
```

Java对于类型安全性的保证是其安全模型的基础。类型安全性意味着程序只允许按照类所设计的那样处理在栈中的对象实例占据的内存。同样，类型安全性意味着程序只允许按照类所设计的那样处理在方法区中类的静态变量所占据的内存。如果虚拟机可能混淆类型，如同这个例子所演示的，恶意的代码就可能看到或者修改非公开的变量。除此之外，如果恶意的代码可能使用某个类型版本中返回一个int变量的方法，然后在另一个版本的这个类型中解释这个int返回值是一个数组，恶意的代码就可以把一个int转换成一个数组引用。通过这种伪造的指针，恶意的代码可以制造一场浩劫。因此，给Java的类型安全性提供保障是很重要的。装载约束就可以保证，就算在存在多个命名空间的情况下，Java的类型安全性在运行时也要坚持。

8.2 随书光盘

随书光盘的linking目录下包含了本章用到的例子的源代码。

8.3 资源页

要了解更多关于Java虚拟机的信息，请访问资源页面：http://www.artima.com/insidejvm/resources。

第9章 垃圾收集

在Java虚拟机的堆里存放着正在运行的Java程序所创建的所有对象。使用new、newarray、anewarray和multianewarray指令来创建对象，但是没有明确的代码来释放它们。垃圾收集就是自动释放不再被程序所使用的对象的过程。

这一章并不是描述正式的Java垃圾收集堆，因为根本不存在这样一个正式描述。前几章讲过，Java虚拟机规范不要求任何特定的垃圾收集技术，这根本就不是必需的。但是在发明可以无限使用的内存之前，大部分的Java虚拟机都会附带垃圾收集堆。本章描述几种不同的垃圾收集技术，并且解释垃圾收集在Java虚拟机里是如何工作的。

本书的附带光盘中为本章准备了一个applet程序，它交互地演示了本章所介绍的内容。该applet被称为“Heap of Fish”，它模拟了Java虚拟机中的一个垃圾收集堆。该模拟演示一个简洁的“标记并清除”收集器，它允许你扮演一个Java程序来和这个堆交互：比如，你可以分配对象，创建到变量的应用。它也允许你扮演Java虚拟机：比如，你可以驱动垃圾处理和压缩堆的过程。在本章的最后可以找到关于这个applet的描述及使用它的指令。

9.1 为什么要使用垃圾收集

“垃圾收集”这个名字暗示着程序不再需要的对象就是“垃圾”，可以被丢弃。更精确、更新的说法是“内存回收”。当一个对象不再被程序所引用时，它所使用的堆空间可以被回收，以便被后续的新对象所使用。垃圾收集器必须能断定哪些对象是不再被引用的，并且能够把它们所占据的堆空间释放出来。在释放不再被引用的对象的过程中，垃圾收集器运行将要被释放的对象的终结方法（finalizer）。

除了释放不再被引用的对象，垃圾收集器还要处理堆碎块。堆碎块是在正常的程序运行过程中产生的。新的对象分配了空间，不再被引用的对象被释放，所以堆内存的空闲位置介于活动的对象之间。请求分配新对象时可能不得不增大堆空间的大小，虽然可以使用的总空闲空间是足够的。这是因为，堆中没有连续的空闲空间放得下新的对象。在一个虚拟内存系统中，增长的堆所需要的额外分页（或交换）空间会影响运行程序的性能。在内存较小的嵌入式系统中，碎块导致虚拟机产生不必要的“内存不足”错误。

垃圾收集把用户从释放占用内存的重担中解救出来。知道何时明确地释放内存是非常需要技巧的。把这项工作交给Java虚拟机有几个好处。首先，可以提高生产率。在一个不具有垃圾收集机制的语言下编程时，你可能需要花费好多时间来加班解决难以捉摸的内存问题。当使用Java编程时，你就可以更有效地利用这些时间。

垃圾收集的第二个好处是能帮助程序保持完整性。垃圾收集是Java安全策略的一个重要部分，Java程序员不可能因失误（或者故意）错误地释放内存而导致Java虚拟机崩溃。

使用垃圾收集堆，有一个潜在缺陷是它加大了程序负担，可能影响程序性能。Java虚拟机必

须追踪哪些对象被正在执行的程序所引用，并且动态地终结并释放不再被使用的对象。和明确释放不再被使用的内存比起来，这个活动会需要更多的CPU时间。并且，在垃圾收集环境下，程序员对安排CPU时间来释放无用对象缺乏控制。

9.2 垃圾收集算法

任何垃圾收集算法都必须做两件事情。首先，它必须检测出垃圾对象。其次，它必须回收垃圾对象所使用的堆空间并还给程序。

垃圾检测通常通过建立一个根对象的集合并且检查从这些根对象开始的可触及性来实现。如果正在执行的程序可以访问到的根对象和某个对象之间存在引用路径，这个对象就是可触及的。对于程序来说，根对象总是可以访问的。从这些根对象开始，任何可以被触及的对象都被认为是“活动”的对象。无法被触及的对象被认为是垃圾，因为它们不再影响程序的未来执行。

Java虚拟机的根对象集合根据实现不同而不同，但是总会包含局部变量中的对象引用和栈帧的操作数栈（以及类变量中的对象引用）。另外一个根对象的来源是被加载的类的常量池中的对象引用，比如字符串。被加载的类的常量池可能指向保存在堆中的字符串，比如类名字，超类的名字，超接口的名字，字段名，字段特征签名，方法名或者方法特征签名。还有一个来源是传递到本地方法中的、没有被本地方法“释放”的对象引用。（根据本地方法接口，本地方法可以通过简单地返回来释放引用，或者显式地调用一个回调函数来释放传递来的引用，或者是这两者的结合。）再一个潜在的根对象的来源就是，Java虚拟机运行时数据区中从垃圾收集器的堆中分配的部分。举例来说，在某些实现中，方法区中的类数据本身可能被存放在使用垃圾收集器的堆中，以便使用和释放对象同样的垃圾收集算法来检测和卸载不再被引用的类。

任何被根对象引用的对象都是可触及的，从而是活动的。另外，任何被活动的对象引用的对象都是可触及的。程序可以访问任何可触及的对象，所以这些对象必须保留在堆里面。任何不可触及的对象都可以被收集，因为程序没有办法来访问它们。

在Java虚拟机实现中，有些垃圾收集器可以区别真正的对象引用和看上去像合法对象引用的基本类型（比如一个int）之间的差别。（例如一个int整数，如果被解释是一个本地指针，可能指向堆中的一个对象。）可是某些垃圾收集器仍然选择不区分真正的对象引用和“伪装品”，这种垃圾收集器被称为保守的（conservative），因为它们可能不能总是释放每一个不再被引用的对象。对保守的收集器，有时候垃圾对象也被错误地判断为活动的，因为有一个看上去像是对象引用的基本类型“引用”了对象。保守的收集器使垃圾收集速度提高了，因为有一些垃圾被遗忘了。

区分活动对象和垃圾的两个基本方法是引用计数和跟踪。引用计数垃圾收集器通过为堆中的每一个对象保存一个计数来区分活动对象和垃圾对象。这个计数记录下了对那个对象的引用次数。跟踪垃圾收集器实际上追踪从根结点开始的引用图。在追踪中遇上的对象以某种方式打上标记，当追踪结束时，没有被打上标记的对象就被判定是不可触及的，可以被当作垃圾收集。

9.3 引用计数收集器

引用计数是垃圾收集的早期策略。在这种方法中，堆中每一个对象都有一个引用计数。当一个对象被创建了，并且指向该对象的引用被分配给一个变量，这个对象的引用计数被置为1。

当任何其他变量被赋值为对这个对象的引用时，计数加1。当一个对象的引用超过了生存期或者被设置一个新的值时，对象的引用计数减1。任何引用计数为0的对象可以被当作垃圾收集。当一个对象被垃圾收集的时候，它引用的任何对象计数值减1。在这种方法中，一个对象被垃圾收集后可能导致后续其他对象的垃圾收集行动。

这种方法的好处是，引用计数收集器可以很快地执行，交织在程序的运行之中。这个特性对于程序不能被长时间打断的实时环境很有利。坏处就是，引用计数无法检测出循环（即两个或者更多的对象互相引用）。循环的例子如，父对象有一个对子对象的引用，子对象又反过来引用父对象。这些对象永远都不可能计数为0，就算它们已经无法被执行程序的根对象可触及。还有一个坏处就是，每次引用计数的增加或者减少都带来额外开销。

因为引用计数方法固有的缺陷，这种技术现在已经不为人所接受。现实生活中所遇到的Java虚拟机更有可能在垃圾收集堆中使用追踪算法。

9.4 跟踪收集器

跟踪收集器追踪从根结点开始的对象引用图。在追踪过程中遇到的对象以某种方式打上标记。总的来说，要么在对象本身设置标记，要么用一个独立的位图来设置标记。当追踪结束时，未被标记的对象就知道是无法触及的，从而可以被收集。

基本的追踪算法被称作“标记并清除”。这个名字指出垃圾收集过程的两个阶段。在标记阶段，垃圾收集器遍历引用树，标记每一个遇到的对象。在清除阶段，未被标记的对象被释放了，使用的内存被返回到正在执行的程序。在Java虚拟机中，清除步骤必须包括对象的终结。

9.5 压缩收集器

Java虚拟机的垃圾收集器可能有对付堆碎块的策略。标记并清除收集器通常使用的两种策略是压缩和拷贝。这两种方法都是快速地移动对象来减少堆碎块。压缩收集器把活动的对象越过空闲区滑动到堆的一端，在这个过程中，堆的另一端出现一个大的连续空闲区。所有被移动的对象的引用也被更新，指向新的位置。

更新被移动的对象的引用有时候通过一个间接对象引用层可以变得更简单。不直接引用堆中的对象，对象的引用实际上指向一个对象句柄表。对象句柄才指向堆中对象的实际位置。当对象被移动了，只有这个句柄需要被更新为新位置。所有的程序中对这个对象的引用仍然指向这个具有新值的句柄，而句柄本身没有移动。这种方法简化了消除堆碎块的工作，但是每一次对象访问都带来了性能损失。

9.6 拷贝收集器

拷贝垃圾收集器把所有的活动对象移动到一个新的区域。在拷贝的过程中，它们被紧挨着布置，所以可以消除原本它们在旧区域的空隙。原有的区域被认为都是空闲区。这种方法的好处是对象可以在从根对象开始的遍历过程中随着发现而被拷贝，不再有标记和清除的区分。对象被快速拷贝到新区域，同时转向指针仍然留在原来的位置。转向指针可以让垃圾收集器发现已经被转移的对象的引用。然后垃圾收集器可以把这些引用设置为转向指针的值，所以它们现

在指向对象的新位置。

一般的拷贝收集器算法被称为“停止并拷贝”。在这个方案中，堆被分为两个区域，任何时候都只使用其中的一个区域。对象在同一个区域中分配，直到这个区域被耗尽。此时，程序执行被中止，堆被遍历，遍历时遇到的活动对象被拷贝到另外一个区域。当停止和拷贝过程结束时，程序恢复执行。内存将从新的堆区域中分配，直到它也被用尽。那时程序将再次中止，遍历堆，活动对象又被拷贝回原来的区域。这种方法带来的代价就是，对于指定大小的堆来说需要两倍大小的内存，因为任何时候都只能使用其中的一半。

在图9-1中可以看到使用“停止并拷贝”算法的垃圾收集堆的图形描述，这幅图显示了随着时间推移堆的9次快照。在第一张快照中，堆的下半部分没有被使用，上半部分零散地被对象填充。堆中包含对象的那部分用灰色斜线表示。快照2显示堆的上半部分逐渐被对象填充，直到如同快照3所示被填满了。

在这时，垃圾收集器中止程序执行，从根结点开始追踪活动对象图。当遇到活动对象时就拷贝到堆的下半部分，每一个都紧挨着上一个被拷贝的对象。这个过程如同快照4所示。

快照5显示了垃圾收集结束后的堆。现在堆的上半部分成了空闲的，下半部分部分填充了活动对象。快照6显示下半部分逐渐被对象填充，直到它被填满了，如同快照7所示。

再一次，垃圾收集器停止了程序，追踪活动对象图。这一次它把遇到的每个活动对象拷贝到堆的上半部分，如快照8所示。快照9显示了垃圾收集行动的结果：下半部分再次变成空闲区，上半部分填充了对象。在程序执行中，这个过程一次又一次地重复。

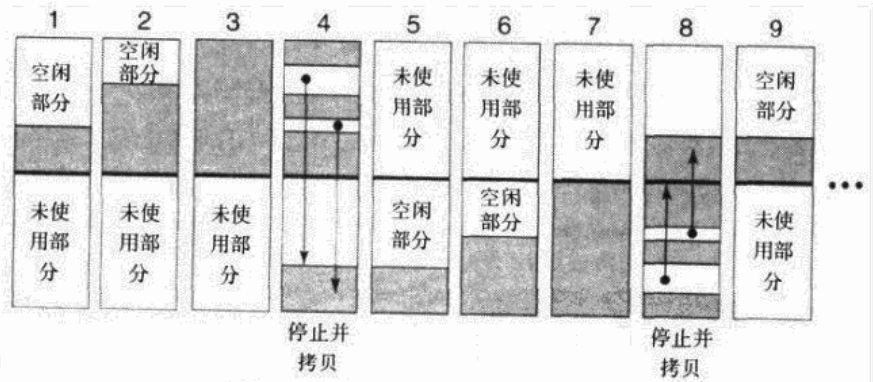


图9-1 一个“停止并拷贝”垃圾收集堆

9.7 按代收集的收集器

简单的停止并拷贝收集器的缺点是，每一次收集时，所有的活动对象都必须被拷贝。大部分语言的大多数程序都有以下特点，如果我们全面考虑这些，拷贝算法的这个缺点是可以被改进的。

1) 大多数程序创建的大部分对象都具有很短的生命期。

2) 大多数程序都创建一些具有非常长生命周期的对象。

简单的拷贝收集器浪费效率的一个主要原因就是，它们每次都把这些生命周期很长的对象

来回拷贝，消耗大量的时间。

按代收集的收集器通过把对象按照寿命来分组解决这个效率低下的问题，更多地收集那些短暂出现的年幼对象，而非寿命较长的对象。在这种方法里，堆被划分成两个或者更多的子堆，每一个子堆为一“代”对象服务。最年幼的那一代进行最频繁的垃圾收集。因为大多数对象都是短促出现的，只有很小部分的年幼对象可以在它们经历第一次收集后还存活。如果一个最年幼的对象经历了好几次垃圾收集后仍然存活，那么这个对象就成长为寿命更高的一代：它被转移到另外一个子堆中去。年龄更高的每一代的收集都没有年轻的那一代来得频繁。每当对象在它所属的年龄层（代）中变得成熟（逃过了多次垃圾收集）之后，它们就被转移到更高的年龄层中去。

按代进行的收集技术除了可以应用于拷贝算法，也可以应用于标记并清除算法。不管在哪种情况下，把堆按照对象年龄层分解都可以提高最基本的垃圾收集算法的性能。

9.8 自适应收集器

自适应收集器算法利用了如下事实：在某种情况下某些垃圾收集算法工作得更好，而另外一些收集算法在另外的情况下工作得更好。自适应算法监视堆中的情形，并且对应地调整为合适的垃圾收集技术。在程序调整的时候可能会调整某种简单的垃圾收集算法的参数，也可能快速转换到另一种不同的算法。或者把堆划分为子堆，同时在不同的子堆中使用不同的算法。

使用自适应方法，Java虚拟机实现的设计者不需要只选择一种特定的垃圾收集算法。可以使用多种技术，以便在每种技术最擅长的场合使用它们。

9.9 火车算法

垃圾收集算法和明确释放对象比起来有一个潜在的缺点，即垃圾收集算法中程序员对安排CPU时间进行内存回收缺乏控制。要精确地预测出何时（甚至是否）进行垃圾收集、收集需要多长时间，基本上都是不可能的。因为垃圾收集一般都会停止整个程序的运行来查找和收集垃圾对象，它们可能在程序执行的任意时刻暂停，并且暂停的时间也无法确定。这种垃圾收集暂停有时候长得让用户都注意到了。垃圾收集也可能使得程序对事件响应迟钝，无法满足实时系统的要求。如果一种垃圾收集算法可能导致用户可察觉得到的停顿或者使得程序无法适合实时系统的要求，这种算法被称作破坏性的。为了减少垃圾收集和明确释放对象之间的潜在差距，设计垃圾收集算法的一个基本目标就是使本质上的破坏性尽可能少，如果可能的话，尽可能消除这种破坏性。

达到（或者试图达到）非破坏性垃圾收集的方法是使用渐进式收集算法。渐进式垃圾收集器就是不试图一次性发现并回收所有不可触及的对象，而是每次发现并回收一部分。因为每次都只有堆的一部分执行垃圾收集，因此理论上说每一次收集会持续更短的时间。如果有一个这样的支持渐进收集方法的垃圾收集器，每次可以保证（或者至少非常接近）不超过一个最大时间长度，就可以让Java虚拟机适合实时环境。限时渐进垃圾收集器在用户环境中也令人满意，因为这样的收集器可以消除用户可察觉得到的垃圾收集停顿。

通常渐进式收集器都是按代收集的收集器，大部分调用中，都是收集堆的一部分。在本章

的前面部分曾经提到，按代收集的收集器把堆划分为两个或多个年龄层，每一个都拥有自己的子堆。凭经验可知，大部分对象都很快消亡，利用这一点，按代收集的垃圾收集器在年幼的子堆中比在年长的子堆中活动更频繁。因为除了最高寿的那个年龄层（成熟对象空间）之外，每一个子堆中都可以给定一个最大尺寸，按代收集的收集器可以大体上保证在一个最大时间值内渐进地收集所有的对象（最高寿的除外）。成熟对象空间无法给定最大尺寸，因为，任何在其他年龄层中不再适合的对象总要有个去处，它们没有其他地方可去。

火车算法最早是由理查德·哈德森（Richard Hudson）和埃里特·莫斯（Eliot Moss）提出的，目前正用于Sun公司的Hotspot虚拟机，该算法详细说明了按代收集的垃圾收集器的成熟对象空间的组织。火车算法的目的是为了在成熟对象空间提供限定时间的渐进收集。

9.9.1 车厢、火车和火车站

火车算法把成熟对象空间划分为固定长度的内存块，算法每次在一个块中单独执行。“火车算法”这个名字来源于算法组织这些块的方式。每一个块属于一个集合。在一个集合内的块排了序，这些集合本身也排了序。在哈德森和莫斯的原始论文中，为了更好地解释算法，把块叫做“车厢”，把集合叫做“火车”。使用这个比喻，成熟对象空间扮演火车站的角色。同一个集合中的块被排序，就如同一列火车中的车厢是有顺序的。成熟对象空间中的集合被排序，很像在火车站中火车按照轨道1、轨道2、轨道3等排列。这个组织结构在图9-2中可以看到。

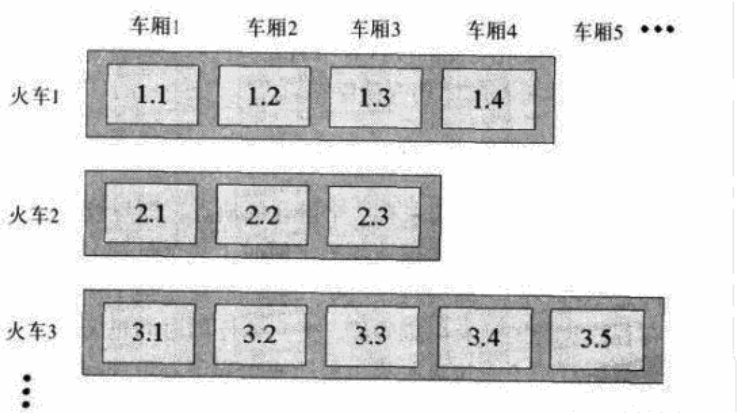


图9-2 火车算法的堆组织

火车按照它们创建时的顺序分配号码。因此，在火车站，第一列火车会被拉进轨道1，称为火车1。到达的第二列火车被拉到轨道2，称为火车2。下一列到达的火车被拉到轨道3，称为火车3，依此类推。按照这样的计划，号码较小的火车总是更早出现的火车。在火车内部，车厢（块）总是被附加到火车的尾部。附加的第一节车厢被称为车厢1，这列车附加的下一节车厢被称为车厢2。因此，在一列车内部，较小的数字表示更早出现的车厢。这个命名计划给出了成熟对象空间中块的总体顺序。

图9-2中显示了三列车，标记为1，2，3。火车1拥有四节车厢，标记为1.1到1.4。火车2拥有三节车厢，标记为2.1到2.3。火车3拥有5节车厢，标记为3.1到3.5。这种标记车厢的方式是：火车号码加上一个点再加上车厢号码，用这种方式表示出了成熟对象空间中所有的块的总体顺序。

车厢1.1在车厢1.2的前面，车厢1.2在车厢1.3的前面，以此类推。火车1的最后一节车厢总是在火车2的第一节车厢的前面，所以车厢1.4在车厢2.1之前。同理，车厢2.3在车厢3.1之前。火车算法每一次执行的时候，只会对一个块（号码最低的块）执行垃圾收集。也就是说，第一次火车算法执行时，堆如同图9-2所示，它会收集块1.1。下一次执行时会收集块1.2。当它收集了火车1的最后一个块，算法在下一次执行时收集火车2的第一块。

对象从更年轻的年龄层的子堆中提出来进入成熟对象空间。不管何时从年轻年龄层中提出，它们被附加到任何已经存在的火车中（最小号码的火车除外），或者专为容纳它们而创建的一列或多列火车中。也就是说，你可以想像新的对象可能有两种方法到达火车站，要么被打包成车厢，挂接到除了号码最小之外的火车的尾部，要么作为一列新的火车开进火车站。

9.9.2 车厢收集

每一次火车算法被执行的时候，它要么收集最小数字火车中的最小数字车厢，要么收集整列最小数字火车。算法首先检查指向最小数字火车中任何车厢的引用，如果不存在任何来自最小数字火车以外的引用指向它内部包含的对象，那么整列最小数字火车包含的都是垃圾，可以被抛弃。这第一步使得火车算法可以一次收集大型的、无法在一个块中容纳的循环数据结构。在下面将要描述的火车算法步骤中，这种大型的循环数据结构，保证可以在同一火车中被销毁。

假如最小数字火车被认为都是垃圾，火车算法归还火车中所有车厢中的对象并且返回。（此时，这次火车算法执行结束。）如果最小数字火车并不都是垃圾，那么算法把它的注意力放到火车的最小数字车厢上。在这个处理过程中，算法或者转移或者释放车厢中的任何对象。算法首先把所有被最小数字车厢外部的车厢引用的对象转移到其他车厢去。当进行这个移动后，任何保留在车厢内的对象都是没有引用的，可以被垃圾收集。火车算法归还整节最小数字车厢占据的空间（因此释放了所有仍然保留在车厢内的不再被引用的对象）并且返回。

保证整列火车中没有循环的数据结构的关键是算法如何移动对象。如果正被收集的车厢中有一个对象存在来自成熟对象空间以外的引用，这个对象被转移到正在被收集的火车之外的其他车厢去。如果对象被成熟对象空间的其他火车引用，对象就被转移到引用它的那列火车中去。然后转移过后的对象被扫描，查找对原车厢的引用。发现的任何被引用的对象都被转移到引用它的火车中去。新被转移的对象也被扫描，这个过程不断重复，直到没有任何来自其他火车的引用指向正被收集的那节车厢。如果接收对象的火车没有空间了，那么算法会创建新的车厢（一节空车厢），并附加到那列火车的尾部。

一旦没有从成熟对象空间外部来的引用，也没有从成熟对象空间内其他火车来的引用，那么这节正在被收集的车厢剩余的外部引用都是来自于同一列火车的其他车厢。算法把这样的对象转移到这节最小数字火车的最后一个车厢去。然后这些对象被扫描，查找对原被收集车厢的引用。任何新发现的被引用对象也被转移到同一列火车的尾部，也被扫描。这个过程不断重复，直到没有任何形式的引用指向被收集的车厢。然后算法归还整个最小数字车厢占据的空间，释放所有仍然在车厢内的对象，并且返回。

因此，在每一次执行时，火车算法或者收集最小数字的火车中最小数字的车厢，或者收集整列最小数字火车。火车算法最重要的方面之一，就是它保证大型的循环数据会完全被收集，即使它们不能被放置在一个车厢中。因为对象被转移到引用它们的火车，相关的对象会变得集

中。最后，成为垃圾的循环数据结构中的所有对象，不管有多大，会被放置到同一列火车中去。增大循环数据结构的大小只会增大最终组成同一列火车的车厢数。因为火车算法在检查最小数字车厢之前，首先检查最小数字火车是否完全就是垃圾，它可以收集任何大小的循环数据结构。

9.9.3 记忆集合和流行对象

前面提到过，火车算法的目标是为了提供限定时间内的按代收集的收集器中成熟对象空间的渐进式收集。因为块（车厢）可以指定一个最大尺寸限度，并且每一次执行只收集一个块，大部分情况下，火车算法可以保证每次的执行时间在某个最长时间限度以内。不过，火车算法不能确保每一次执行都在最长时间限度之内，因为算法不仅仅是拷贝对象。

为了促进收集过程，火车算法使用了记忆集合。一个记忆集合是一个数据结构，它包含了所有对一节车厢或者一列火车的外部引用。算法为成熟对象空间内每节车厢和每列火车都维护一个记忆集合。所以，一节特定车厢的记忆集合包含了指向车厢内对象的所有引用（或“记忆”）的集合。一个空的记忆集合显示车厢或者火车中的对象都不再被车厢或者火车外的任何变量引用（已经被“遗忘”）。被遗忘的对象就是不可触及的，可以被垃圾收集。

记忆集合是一种可以帮助火车算法更有效地完成工作的技术。当火车算法发现一节车厢的记忆集合是空的时，它就知道车厢里面都是垃圾，可以立即归还这节车厢占用的所有内存。同样，当火车算法发现一列火车的记忆集合是空的时，它可以立即归还整列火车占用的内存。当火车算法把一个对象转移到另外一节车厢或者另外一列火车中时，记忆集合中的信息有助于它高效地更新所有指向被移动对象的引用，它们就可以指向新的位置。

虽然火车算法每次执行时需要拷贝的字节总数受限于块的大小，但是移动一个很受欢迎的对象（具有很多外部连接的对象）所需要的工作几乎是不可能限制的。每次算法移动一个对象时，它必须遍历对象的记忆集合，更新每一个连接，以便使连接指向新的位置。因为指向一个对象的连接数是无法限定的，更新一个被移动对象的所有连接需要的总时间长度也是无法限定的。也就是说，在特定条件下，火车算法仍然可能是破坏性的。不过，除了流行对象情况下不太适用外，火车算法在大部分情况下工作得非常好，为成熟对象空间内进行渐进的、非破坏性的垃圾收集提供了很好的方法。

9.10 终结

在Java语言里，一个对象可以拥有终结方法：这个方法是垃圾收集器在释放对象前必须运行。这个可能存在的终结方法使得任何Java虚拟机的垃圾收集器要完成的工作更加复杂。

给一个类加上终结方法，只需简单地在类中声明一个方法：

```
// On CD-ROM in file gc/ex2/Example2.java
class Example2 {

    protected void finalize() throws Throwable {
        //...
        super.finalize();
    }
    //...
}
```

垃圾收集器必须检查它所发现的不再被引用的对象是否存在finalize（）方法。

因为，存在终结方法时，Java虚拟机的垃圾收集器必须每次在收集时执行一些额外的步骤。首先，垃圾收集器必须使用某种方法检测出不再被引用的对象（称作第一遍扫描）。然后，它必须检查它检测出的不再被引用的对象是否声明了终结方法。如果时间允许的话，可能在这个时候垃圾收集过程就着手处理这些存在的终结方法。

当执行了所有的终结方法之后，垃圾收集器必须从根结点开始再次检测不再被引用的对象（称作第二遍扫描）。这个步骤是必要的，因为终结方法可能“复活”了某些不再被引用的对象，使它们再次被引用了。最后，垃圾收集器才能释放那些在第一次和第二次扫描中发现的都没有被引用的对象。

为了减少释放内存的时间，在扫描到某些对象拥有终结方法和运行这些终结方法之间，垃圾收集器可以有选择地插入一个步骤。一旦垃圾收集器执行了第一遍扫描，并且找到一些不再被引用的对象需要执行终结，它可以运行一次小型的追踪，从需要执行终结的对象开始（而非从根结点开始）。任何满足如下条件的对象——从根结点开始不可触及（在第一遍扫描中检测出），以及从将要被终结的对象开始不可触及——这些对象不可能在执行终结方法时复活，它们可以立即被释放。

如果一个带有终结方法的对象不再被引用，并且它的终结方法运行过了，垃圾收集器必须使用某种方法记住这一点，而不能再次执行这个对象的终结方法。如果这个对象被它自己的终结方法或者其他对象的终结方法复活了，稍后再次不再被引用，垃圾收集器必须像对待一个没有终结方法的对象一样对待它。

使用Java编程时，必须记住一点:是垃圾收集器运行对象的终结方法。因为总是无法预测何时对象会被垃圾收集，所以也无法预测对象的终结方法何时运行。在第2章曾讲过，应该避免编写这样的程序，即程序的正确性依赖于对象的终结方法所运行的时机。比如，如果不再被引用的对象的终结方法释放了一个以后程序将会用到的资源，这个资源将直到垃圾执行器运行了这个对象的终结方法后才能够使用。如果程序在垃圾收集器有机会终结这个不在被引用的对象之前需要这个资源，这个程序将无法得到该资源。

9.11 对象可触及性的生命周期

在版本1.2之前，在垃圾收集器看来，堆中的每一个对象都有三种状态之一：可触及的，可复活的，以及不可触及的。如果垃圾收集器可以从根节点开始通过追踪“触及”到这个对象，它就是可触及的。每一个对象都是从可触及状态开始它的生命周期的，只要程序还保留至少一个可以触及的引用到该对象，它就一直保持可触及状态。一旦程序释放了所有到该对象的引用，然后这个对象就变成可复活状态。

如果一个对象满足如下条件，它就处于可复活状态：它在从根节点开始的追踪图中不可触及，但是有可能在垃圾收集器执行某些终结方法时触及。不仅仅是那些声明了finalize（）方法的对象，而是所有的对象都要经过可复活状态。前面的部分讲到了，通过再次触及对象，对象的终结方法可能“复活”对象本身或者其他对象。因为通过对象自己定义的finalize（），或者其他对象的该方法，任何处于可复活状态的对象都可能再次复活，所以垃圾收集器就不能够归还

可复活的对象所占据的内存，直到它确信不再有任何终结方法有机会把这个对象复活。在执行所有可复活对象可能声明的finalize（）方法之后，垃圾收集器会把那些处于可复活状态的对象或者转化为可触及的状态（那些被复活的对象），或者前进到不可触及状态。

不可触及状态标志着不但对象不再被触及，而且也不可能通过任何终结方法复活。不可触及的对象不再对程序的执行产生影响，可以自由地回收它们所占据的内存。

在版本1.2中，对原来的三个可触及性状态——可触及的，可复活的，不可触及的——扩充了三个新状态：软可触及，弱可触及，以及影子可触及。因为这三个新状态表示了三种新的可触及性（逐渐减弱），原来在版本1.2之前简单地被称作“可触及”的状态现在从版本1.2开始被称作“强可触及”。从根节点开始的任何直接引用，比如一个局部变量，是强可触及的。同理，任何从强可触及对象的实例变量引用的对象也是强可触及的。

9.11.1 引用对象

可触及性的三个比较弱的形式涉及到从版本1.2开始新引入的实体——引用对象。引用对象封装了指向其他对象的连接。被指向的对象称为引用目标。所有的引用对象都是抽象的java.lang.ref.Reference类的子类的实例。Reference类家族如图9-3所示，包含了三个直接的子类：SoftReference，WeakReference，PhantomReference。SoftReference对象封装了对引用目标的“软引用”；WeakReference对象封装了对引用目标的“弱引用”；而PhantomReference对象封装了对引用目标的“影子引用”。强引用和较弱形式的引用——软引用、弱引用和影子引用——之间最基本的差别是，强引用禁止引用目标被垃圾收集，而软引用、弱引用和影子引用不禁止。

```
package java.lang.ref;
```

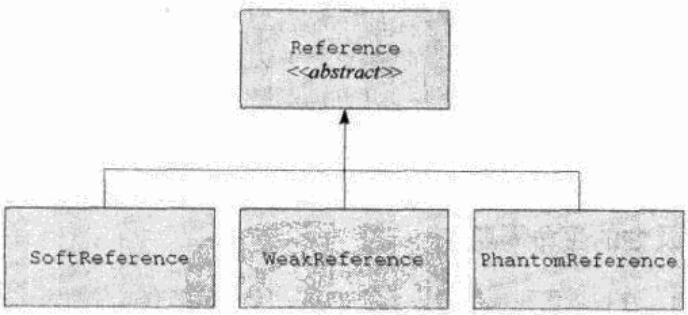


图9-3 引用家族

要创建一个软引用、弱引用或者影子引用，简单地把强引用传递到对应的引用对象的构造方法中去。比如，要创建一个对某个Cow对象的软引用，就把一个指向Cow对象的强引用传递到一个新的SoftReference对象的构造方法中。通过维护对这个SoftReference对象的强引用，也维护了对这个Cow对象的软引用。

图9-4表示了这样一个SoftReference对象，它封装了对一个Cow对象的软引用。SoftReference对象被一个局部变量所强引用，和所有的局部变量一样，对于垃圾收集器来说这是一个根节点。在前面说过，垃圾收集器的根节点包含的引用和强可触及对象的实例变量包含的引用都是强引用。因为图9-4中所示的SoftReference对象被一个强引用所引用，这个SoftReference

对象是强可触及的。假设只有这个SoftReference对象才拥有对此Cow对象的引用，那么这个Cow对象是软可触及的。因为垃圾收集器从根节点开始只有通过一个软引用才能触及到此Cow对象。

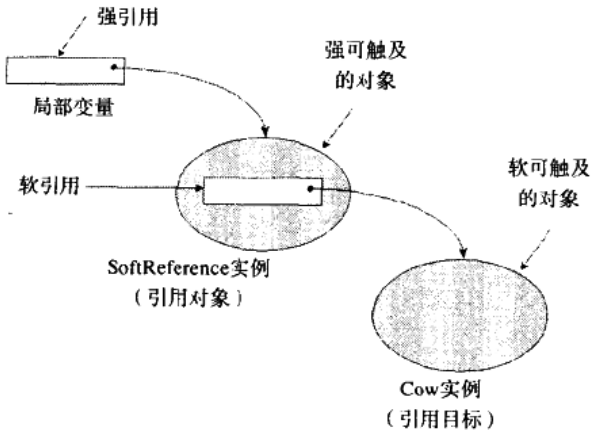


图9-4 一个引用对象和它的引用目标

一旦一个引用对象创建后，它将一直维持到它的引用目标的软引用、弱引用或者影子引用，直到它被程序或者垃圾收集器清除。要清除一个引用对象，程序或者垃圾收集器只需要调用引用对象的clear（）方法，这个方法是Reference类定义的。通过清除引用对象，就切断了引用对象的软引用、弱引用或者影子引用。比如，如果程序或者垃圾收集器试图运行图9-4中的SoftReference对象的clear（）方法，通往此Cow对象的软连接就被切断了，这个Cow对象就不再是软可触及的了。

9.11.2 可触及性状态的变化

前面讲到过，引用对象的目的是为了能够指向某些对象，这些对象仍然随时可以被垃圾收集器收集。换个说法就是，垃圾收集器可以随意更改不是强可触及的任何对象的可触及性状态。当使用软引用、弱引用或者影子引用的时候，因为跟踪垃圾收集器改变对象的可触及性状态常常很重要，因此可以安排在这种变化发生时得到一个通知。如果对可触及性状态的改变有兴趣，可以把引用对象和引用队列关联起来。引用队列是java.lang.ref.ReferenceQueue类的一个实例，垃圾收集器在改变可触及性状态时会添加（编入队列）所涉及的引用对象。设置并且观察引用队列，当垃圾收集器是对你感兴趣的对象改变可触及性状态时，你就可以异步得到通知了。

要把一个引用对象和一个引用队列关联起来，可以简单地在创建引用对象时把引用作为构造方法的参数传递到引用队列。如此创建的引用对象除了保持对引用目标的引用外，还保持对引用队列的引用。当垃圾收集器对引用目标的可触及性状态做了改变时，它就会把引用对象加入到与它有关联的引用队列中去。比如，图9-5中所示的WeakReference对象创建的时候，有两个引用被传递到构造方法：一个指向Fox对象的引用和一个指向ReferenceQueue对象的引用。当垃圾收集器决定收集弱可触及的Fox对象的时候，它会清除WeakReference对象（执行WeakReference的clear（）方法），可能立即就把这个WeakReference对象加入到它的引用队列中，也可能在稍后的某个时间加入。

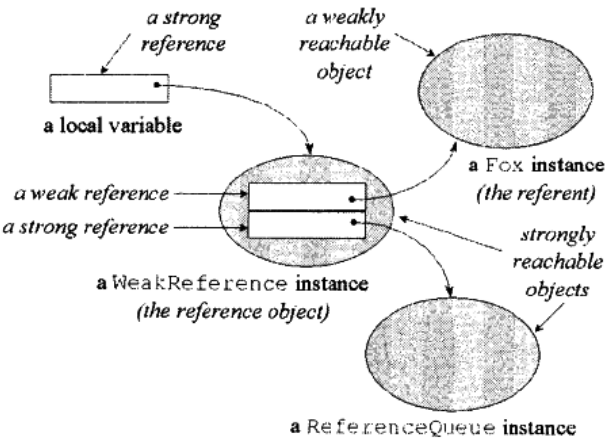


图9-5 一个和引用队列相关联的引用对象

为了把引用对象加入到它所关联的队列中，垃圾收集器执行它的enqucue( )方法。enqueue( )方法是在超类Reference中定义的，只有在创建引用对象时关联了一个队列、并且仅当该对象的enqueue ( ) 方法第一次执行时，才把引用对象加入到这个队列中。程序可以有两种方法来监控引用队列：用poll ( ) 方法拉出来，或者用remove ( ) 方法阻塞它。如果在队列中有一个引用对象在等待，不管在队列上执行poll ( ) 还是remove ( ) 方法，都会取走这个对象，返回给调用者。如果没有引用对象在等待，那么poll ( ) 会立即返回null，而remove ( ) 会阻塞，直到有一个引用对象加入队列。一旦有对象到达队列，remove ( ) 会取走并且返回它。

在不同的情况下，垃圾收集器把软引用、弱引用和影子引用对象加入队列表示三种不同的可触及性状态的转换。这表示六种可触及状态，状态变化的详情如下所列：

- 强可触及　对象可以从根节点不通过任何引用对象搜索到。对象生命周期从强可触及状态开始，并且只要有根节点或者另外一个强可触及对象引用它，就保持强可触及状态。垃圾收集器不会试图回收强可触及对象占据的内存空间。
- 软可触及　对象不是强可触及的，但是可以从根节点开始通过一个或多个（未被清除的）软引用对象触及。垃圾收集器可能回收软可触及的对象所占据的内存。如果这发生了，它会清除所有到此软可触及对象的软引用。当垃圾收集器清除一个和引用队列有关联的软引用对象时，它把该软引用对象加入队列。
- 弱可触及　对象既不是强可触及的也不是软可触及的，但是从根节点开始可以通过一个或多个（未被清除的）弱引用对象触及。垃圾收集器必须归还弱可触及对象所占据的内存。这发生的时候，它会清除所有到此弱可触及对象的弱引用。当垃圾收集器清除一个和引用队列有关联的弱引用对象时，它把该弱引用对象加入队列。
- 可复活的　对象既不是强可触及、软可触及，也不是弱可触及，但是仍然可能通过执行某些终结方法复活到这几种状态之一。
- 影子可触及　对象不是强可触及、软可触及，也不是弱可触及，并已经被断定不会被任何

终结方法复活（如果它自己定义了终结方法，它的终结方法已经被运行过了），并且它可以从根节点开始通过一个或多个（未被清除的）影子引用对象触及。一旦某个被影子引用的对象变成影子可触及状态，垃圾收集器立即把该引用对象加入队列。垃圾收集器从不会清除一个影子引用，所有的影子引用都必须由程序明确地清除。

- 不可触及　一个对象不是强可触及、软可触及、弱可触及，也不是影子可触及，并且它不可复活。不可触及的对象已经准备好被回收了。

请注意，垃圾收集器在把软引用对象和弱引用对象加入队列的时候，是在它们的引用目标离开相应的可触及状态时；而把影子引用对象加入队列是在引用目标进入相应状态时。在垃圾收集器清除引用对象时也有差别，软引用对象和弱引用对象在加入队列之前得到了清除，而影子引用对象却不会。这就是说，垃圾收集器把软引用对象加入队列标志着它的引用对象刚刚离开软可触及状态；同样，垃圾收集器把弱引用对象加入队列标志着它的引用对象刚刚离开弱可触及状态；但是垃圾收集器把影子引用对象加入队列标志着引用目标已经进入了影子可触及状态。影子可触及对象会保持影子可触及状态，直到程序显式地清除了引用对象。

9.11.3 缓存、规范映射和临终清理

垃圾收集器对待软、弱和影子对象的方法不同，因为每一种都是被设计为，为程序提供不同的服务。软引用使你可以创建内存中的缓存，它与程序的整体内存需求有关。弱引用使你可以创建规范映射，比如哈希表，它的关键字和值在没有其他程序部分的引用时可以从映射中清除。影子引用使你可以实现除终结方法以外的更加复杂的临终清理政策。

要使用一个软引用或者弱引用的引用目标，可以调用引用对象的get（）方法。如果引用目标还没有被清除，则会得到对引用目标的一个强引用，就可以用通常的方法去使用它了。如果引用目标已经被清除了，则会得到null。如果调用影子引用对象的get（）方法，那么无论如何只能得到null，即使引用对象还没有被清除。因为影子可触及状态只有经过可复活状态之后才能获得，一个影子引用对象没有提供任何方法来访问它的引用目标。调用影子引用对象的get（）方法只能得到null，即使影子引用还没有清除。因为，如果它返回一个影子可触及对象的强引用，实际上它就复活了这个对象。这就是说，如果一个对象到达了影子可触及状态，它不能再被复活。

虚拟机的实现需要在抛出OutOfMemoryError之前清除软引用，但在其他情况下可以自行选择清理的时间或是否清除它们。实现最好是只在内存不敷所需时才去清除软连接，清除的时候先清除老的而不是新的，清除长期未用的而不是最近刚刚用过的。

软引用可以让你在内存中缓存那些需要从外部数据源费时取回的数据，比如文件中、数据库里或者网络上的数据。所以只要虚拟机有足够的内存，可以在堆中保存所有的强引用的数据以及软引用的数据，软引用大体上对于在堆中保存软引用的数据已经足够强了。如果内存变得紧张，垃圾收集器会决定清除软引用，回收被软引用的数据所占用的空间。下一次程序需要使用这个数据的时候，可能不得不再次从外部数据源装入。同时，虚拟机就有更多的空间用来调整程序强引用（或者其他软引用）需要的内存。

弱引用类似于软引用，不同的是：垃圾收集器可以自行决定是否清除指向软可触及的对象的软连接，而它必须在判断出对象处于弱连接状态时就立即清除弱引用。弱引用使得你可以用关键字和值来创建规范映射。java.util.WeakHashMap类就是用弱引用提供这样的规范映射。可

以通过put（）方法加入键-值对到WeakHashMap的实例，如同可以对实现了java.util.Map的任何类的实例所做的那样。但是在WeakHashMap中，关键字对象是通过一个关联到引用队列的弱引用实现的。如果垃圾收集器断定某个关键字对象是弱可触及的，它会清除引用并且把任何弱引用到该对象的引用加入各自的队列。下一次当WeakHashMap被访问的时候，它从引用队列里面拉出所有垃圾收集器放在那儿的弱引用对象。WeakHashMap就会清除它的映射中任何关键字属于队列中弱引用的键-值对。这就是说，如果把一个键-值对加入到WeakHashMap，在程序显式地使用remove（）方法移出它或者垃圾收集器发现关键字对象是弱可触及之前，它会一直保留在WeakHashMap里面。

影子可触及性表示对象即将被回收。当垃圾收集器断定影子引用对象的引用目标处于影子可触及状态时，它把该影子引用加入到所关联的引用队列。（和软引用对象、弱引用对象不同，软引用对象和弱引用对象可以在创建时选择不和一个引用队列关联，而影子引用对象没有一个关联的引用队列就无法创建实例。）可以利用引用队列中影子引用的到达来触发一些你希望在对象生命周期的最后时刻需要完成的动作。因为无法获得对影子可触及对象的强引用（get（）方法总是返回null），所以无法完成那些需要访问影子目标的实例变量的动作。在完成了影子可触及对象的临终清理之后，必须调用指向它的影子引用对象的clear（）方法。调用一个影子引用对象的clear（）方法是对它的引用对象的致命一击，把引用目标从影子可触及状态导向它的终点：不可触及状态。

9.12 一个模拟："Heap of Fish"

图9-6到9-9所示的"Heap of Fish"（鱼堆）applet，演示了一个压缩的、标记并清除的垃圾收集堆。为了便于压缩，堆使用对象的间接句柄而非直接引用，这个程序被称作"Heap of Fish"，这是因为在演示的堆里面，所有对象的类型都是鱼，如下所定义：

```
// On CD-ROM in file gc/ex1/YellowFish.java
class YellowFish {

    YellowFish myFriend;
}

// On CD-ROM in file gc/ex1/BlueFish.java
class BlueFish {

    BlueFish myFriend;
    YellowFish myLunch;
}

// On CD-ROM in file gc/ex1/RedFish.java
class RedFish {

    RedFish myFriend;
    BlueFish myLunch;
```

```
    YellowFish mySnack;
}
```

可以看到，有三种鱼类：红色的、黄色的和蓝色的。红色的鱼体形最大，拥有三个实例变量。黄色的鱼体形最小，只有一个实例变量。蓝色的体形中等，拥有两个实例变量。

鱼对象的实例变量是指向其他鱼对象的引用，比如BlueFish.myLunch是一个指向YellowFish对象的引用。在这个垃圾收集堆的实现里面，一个对象的引用占用4个字节。也就是说，一个RedFish对象的实例数据大小是12个字节，一个BlueFish对象是8个字节，而一个YellowFish对象的大小是4个字节。

“Heap of Fish”有5种模式，可以通过applet左下方的单选按钮来选择。applet一开始是“游泳”模式，游泳模式只是个普通的模拟，让你回忆起大鱼吃中鱼，中鱼吃小鱼的熟悉场景。其他四个模式——分配鱼、设置引用、垃圾收集和压缩堆——让你和堆可以互动。在“分配鱼”模式，可以创建新的鱼对象。在“设置引用”模式，可以创建局部变量网，描绘出鱼之间的引用。在“垃圾收集”模式，一个“标记并清除”操作会释放任何未被引用的鱼。“压缩堆”模式允许你滑动堆里的对象，让它们在堆的一侧紧紧地贴在一起，把连续的空闲空间块集中在堆的另外一端。

9.12.1 分配鱼

在分配鱼模式（如图9-6所示），可以在堆中创建新的鱼对象。在此模式中，可以看到堆由两部分组成：对象池和句柄池。对象池是一大块连续的空间，新对象在此创建。对象被构建成一系列的内存块，每一个内存块有四个字节长的头部来指出内存块的大小和它是否空闲。applet中这些头部表示成黑色的横线。

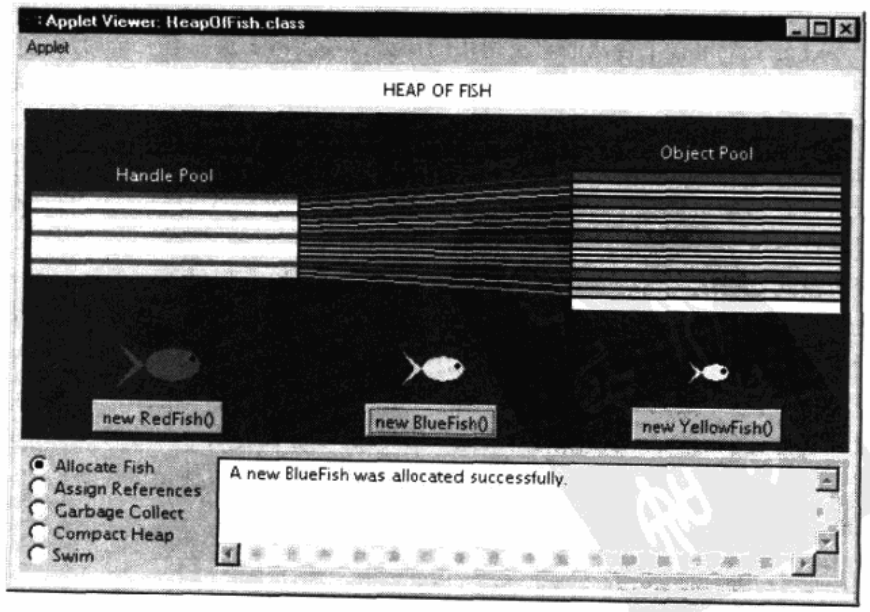


图9-6 “Heap of Fish”applet的分配鱼模式

“Heap of Fish”的对象池是用一个int类型的数组实现的。第一个头部总在objectPool[0]。对象池的一系列内存块可以通过在头部和头部之间跳跃来遍历。每一个头部给出了它的内存块的长度，还显示了下一个内存块的头部的位置。下一个内存块就是这个内存块之后紧跟着的整数。

当一个新对象被分配的时候，遍历内存池来找到一个足够大小的内存块来容纳这个新对象。对象池中分配的对象显示为彩色条。YellowFish对象显示为黄色，BlueFish对象显示为蓝色，RedFish对象显示为红色。空闲的内存块——那些现在还没有容纳鱼的地方，显示为白色。

“Heap of Fish”的句柄池使用ObjectHandle类的对象数组来实现。ObjectHandle中包含了对象的信息，包括指向对象池数组的那个至关重要的索引。对象池的索引用作指向对象池中的对象的实例数据的引用。ObjectHandle也包含了关于鱼对象类的信息。第5章中讲过：堆中每一个对象必须和它在方法区的类信息相关联。在“Heap of Fish”中，ObjectHandle把分配的每一对象和它的类（不管这是一条RedFish、BlueFish还是YellowFish）等信息相关联，还与applet用户界面中显示这条鱼时需要的一些数据相关联。

句柄池的存在是为了在压缩的时候更方便于清理对象池碎片。指向对象的引用，可能是栈中的局部变量，也可能是其他对象的实例变量，但并不是对象池数组中的直接索引，而是句柄池数组的索引。在压缩的时候，对象池中的对象可能会转移，只有对应的ObjectHandle需要更新为对象在对象池数组中的新索引。

句柄池中的每一个指向鱼对象的句柄都显示成一个横条，具有和它指向的鱼相同的颜色。在每一个句柄和它对应的对象池中的鱼实例变量之间，用一条线连接起来。暂时没有使用的句柄显示成白色。

9.12.2 设置引用

在设置引用模式（如图9-7所示），允许你建立一个本地变量和已创建的鱼对象之间的引用网。一个引用只不过是一个包含了合法的对象引用的局部或者实例变量。有三个局部变量，它们充当垃圾收集的根节点，和鱼的种类一一对应。如果没有把任何鱼连接到局部变量，那么所有的鱼都被认为是不可触及的，并且会被垃圾收集器释放。

设置引用模式有三个子模式：移动鱼、连接鱼和断开连接。可以通过画面下方的单选按钮来选择子模式。在移动鱼模式，可以点击一条鱼，把它拉动到一个新的位置。可能在该模式下让网络变得好看一些（或者仅仅是因为想让鱼在海里重新组队）。

在连接鱼模式，可以点击一条鱼或者局部变量，拖一条连接到另外一条鱼上面。拖拽的起点鱼或者局部对象会设置一个引用到拖放的目的鱼，二者之间会显示一条连线。两条鱼之间的连线从使用引用的鱼嘴上连接到被引用的鱼尾上。

YellowFish只有一个实例变量——myFriend，指向一个YellowFish对象。也就是说，黄色的鱼只能连接到其他黄色的鱼。当连接两条黄色鱼的时候，“拖拽起始”的鱼的myFriend变量会被设置为指向“拖放目的”的鱼。这个动作如果使用Java代码实现，可能是这样的：

```
// Fish are allocated somewhere
YellowFish draggedFromFish = new YellowFish();
YellowFish droppedUponFish = new YellowFish();

// Sometime later the assignment takes place
```

```
draggedFromFish.myFriend = droppedUponFish;
```

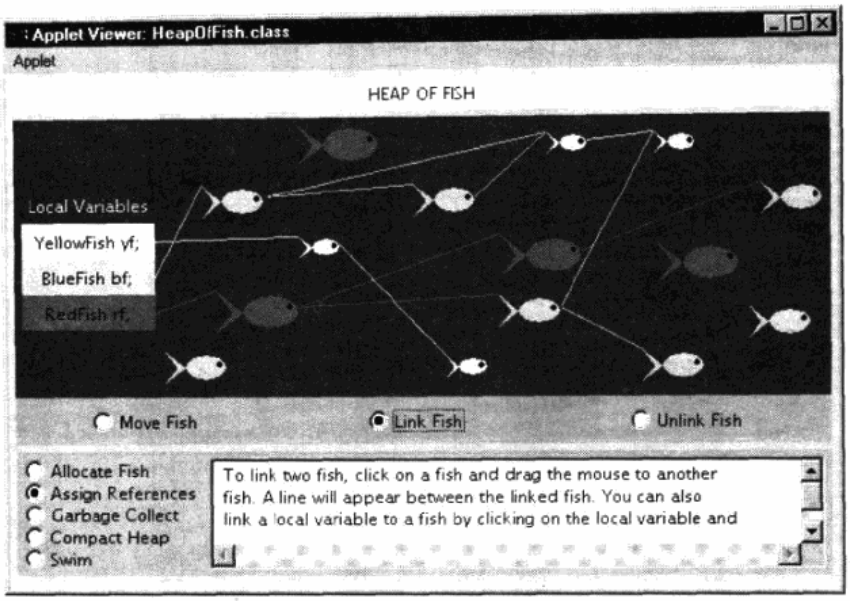


图9-7 “Heap of Fish” applet的设置引用模式

类BlueFish有两个实例变量——BlueFish myFriend和YellowFish myLunch，也就是说，一条蓝色的鱼可以连接到一条蓝色的鱼和一条黄色的鱼。类RedFish由三个实例变量——RedFish myFriend、BlueFish myLunch和YellowFish mySnack，也就是说红色的鱼可以连接到每种鱼的一个实例。

在断开连接模式，可以通过鼠标来解除两条鱼之间的连线。当鼠标移动到线上的时候，线会变成黑色，如果点击变成黑色的线，连接就会置为null，线也消失了。

9.12.3 垃圾收集

在垃圾收集模式（如图9-8所示），可以使用标记并清除算法。画面下方的“Step”按钮带领你每次一步地完成垃圾收集过程。点击“Reset”按钮可以随时重置垃圾收集器。不管怎样，只要被垃圾收集器清除了，被释放的鱼就永久消失了，不管怎样按“Reset”按钮也找不回它们了。

垃圾收集过程分为标记步骤和清除步骤。在标记步骤，堆中的鱼对象从局部变量开始被按照深度优先的方式遍历。在清除步骤，所有没有被标记的鱼都被释放了。

在标记步骤的开始，所有的局部变量、鱼和连接都是白色的。每按一次“Step”按钮，按照深度优先的方式遍历一个节点。当前被遍历的节点（不管是局部变量还是鱼），显示成紫红色。当垃圾收集器遍历一个分支时，这条分支上的鱼从白色变为灰色。灰色意味着遍历触及了这条鱼，但是这条鱼所属的分支还没有完全被触及。一旦整个分支的终点被触及，终点的颜色变为黑色，遍历沿着分支返回。一旦鱼节点后方所有的鱼都标记成黑色，这条鱼也标记为黑色，遍历从来路返回。

在标记阶段的最后，所有被触及到的鱼都被标记为黑色，而不可触及的鱼为白色。清除步

骤释放所有白色的鱼所占据的内存。

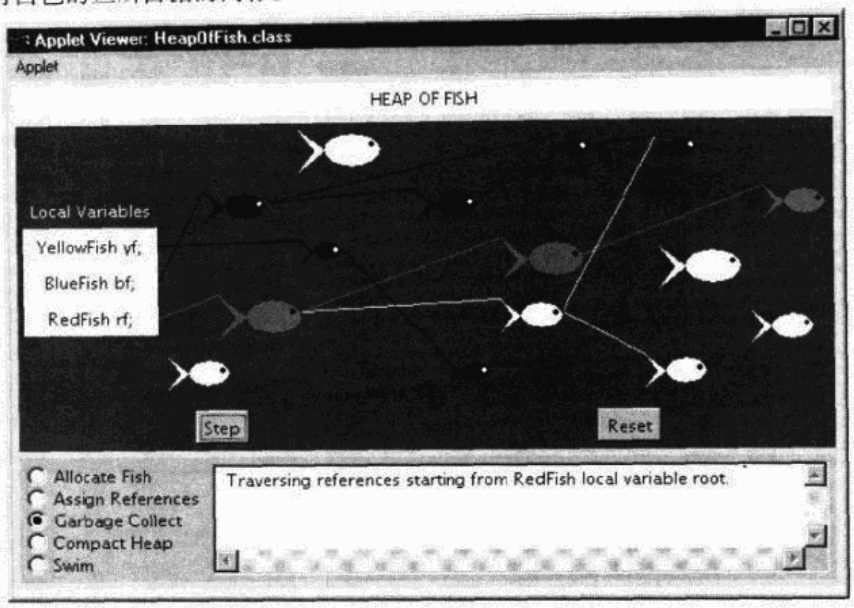


图9-8 “Heap of Fish”applet的垃圾收集模式

9.12.4 压缩堆

在压缩堆模式（如图9-9所示），可以每次移动一个对象到对象池的一端。每按一次“Slide”按钮，移动一个对象。可以看到，只有对象池里对象的实例数据移动了，句柄池里的句柄没有移动。

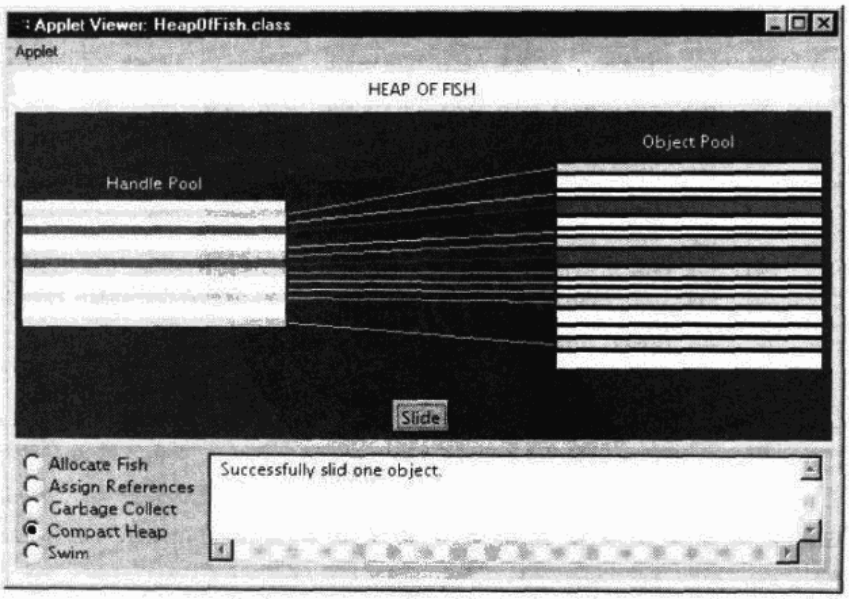


图9-9 “Heap of Fish”applet的压缩堆模式

“Heap of Fish” applet允许你分配新的鱼对象、连接鱼、垃圾收集及压缩堆。这些动作的顺序可以随意安排。试玩一下这个applet，可以领会到“标记并清除”垃圾收集堆是如何运作的。在applet下方，还有一些帮助文字。

9.13 随书光盘

随书光盘在gc目录下存放了本章的示例源代码。“Heap of Fish” applet包含在CD-ROM中的一个网页上，文件名为applets/HeapOfFish.html。这个applet的源代码和它的class文件一起，存放在applets/HeapOfFish目录下。

9.14 资源页

要了解与本章内容相关的更多的信息，请访问资源页面http://www.artima.com/insidejvm/resources。

第10章　栈和局部变量操作

如第5章所述，执行引擎的抽象规范是根据指令集来定义的。本书余下各章（第10章到第20章）是此指令集的说明手册。这些章节对功能组中的指令进行了描述，并为每个功能组提供了相关的背景信息。

本章包括处理操作数栈和局部变量的指令。由于Java虚拟机是基于栈的机器，几乎所有Java虚拟机的指令都与操作数栈相关。绝大多数指令都会在执行自己功能的时候进行入栈、出栈操作。本章对专门用于对操作数栈进行操作的一些指令进行了阐述，这些指令执行下列操作：把常量压入操作数栈，执行通用的栈操作，并在操作数栈和局部变量之间往返传输值。

随书光盘上有一个使用交互方式来阐述本章内容的applet，名为“Fibonacci Forever”。“Fibonacci Forever”模拟了Java虚拟机执行一个产生斐波那契序列的方法的全过程。在这个处理过程中，它说明了虚拟机是如何将常量压入栈，如何将值弹出栈并存入局部变量，以及如何将局部变量中的值压入栈的。本章的末尾描述了此applet及其所执行的字节码。

10.1　常量入栈操作

许多操作码执行常量入栈操作。操作码在执行常量入栈操作之前，使用如下三种方式指明常量的值：常量值隐式包含在操作码内部，常量值在字节码流中如同操作数一样紧随在操作码之后，或者从常量池中取出常量。

一些操作码自行指明入栈的常量的类型和值，例如，iconst_1操作码告知Java虚拟机向栈压入一个值为1的int类型数。Java虚拟机为经常压入栈的各种不同类型的数据定义了一些这样的操作码。相对于从字节码流中取出操作数或者指向常量池的指令来说，上述这些指令都是冗余指令，但它们更有效率。因为这些指令在字节码流中仅仅占据一个字节的空间，它们提高了字节码的执行效率，并减少了字节码的尺寸。向栈中压入int和float类型值的操作码如表10-1所示。

表10-1中所示的操作码把int和float类型压入栈，int和float都是一个字长的值。Java栈中的每一个位置的长度都是一个字长（至少32位宽）。因此，每当一个int或者float类型被压入栈时，它都将占据一个位置。

表10-1　将一个字长的常量压入栈

操作码	操作数	说明
iconst_m1	（无）	将int类型值-1压入栈
iconst_0	（无）	将int类型值0压入栈
iconst_1	（无）	将int类型值1压入栈
iconst_2	（无）	将int类型值2压入栈
iconst_3	（无）	将int类型值3压入栈
iconst_4	（无）	将int类型值4压入栈

（续）

操作码	操作数	说明
iconst_5	（无）	将int类型值5压入栈
fconst_0	（无）	将float类型值0压入栈
fconst_1	（无）	将float类型值1压入栈
fconst_2	（无）	将float类型值2压入栈

表10-2中所示的操作码将long和double类型值压入栈。long和double类型值是64位长度的值。每当一个long或者double类型的值被压入栈时，它都将占据2个位置。

表10-2 将两个字长的常量压入栈

操作码	操作数	说明
lconst_0	（无）	将long类型值0压入栈
lconst_1	（无）	将long类型值1压入栈
dconst_0	（无）	将double类型值0压入栈
dconst_1	（无）	将double类型值1压入栈

还有一个操作码能够将一个隐式声明的常量压入栈。如表10-3所示，aconst_null 操作码将一个空的对象引用类型压入栈。

表10-3 将空（null）对象引用压入栈

操作码	操作数	说明
aconst_null	（无）	将空（null）对象引用压入栈

如前所述，对象引用的格式取决于具体的Java虚拟机实现。一个对象引用可能会莫名其妙地指向垃圾收集堆中的Java对象。一个空的对象引用表明一个当前还没有指向任何有效对象的对象引用变量。给一个对象引用变量赋空值的过程中，将会使用aconst_null 操作码。

如表10-4所示的两个操作码用来将整数常量压入栈，该整数常量的值在byte和short数据类型的有效范围之内，这两个操作码通过使用一个紧随在操作码之后的操作数明确指定将要压入栈的常量。紧随操作码的byte或者short类型在压入栈之前被扩展成int类型值。将int类型值压入栈的操作实际上取代了将byte和short类型值压入栈的操作。

表10-4 将byte和short类型常量压入栈

操作码	操作数	说明
bipush	byte1	将byte1（数据类型为byte）转换为int类型，然后将其压入栈
sipush	byte1，byte2	将byte1和byte2（数据类型为short）转换为int类型，然后将其压入栈

如表10-5所示的三个操作码从常量池中取出常量，然后将其压入栈。这些操作码使用表示常量池索引的操作数。Java虚拟机通过给定的索引查找相应的常量池入口，决定这些常量的类型和值，并把它们压入栈。

常量池索引是一个无符号值，它在字节码流中直接跟随在操作码后面。ldc和ldc_w这两种操作码把一个字长的项压入栈，该项或者是一个int类型、float类型的值，或者是一个String类型的对象引用。ldc和ldc_w之间的区别在于：由于ldc的索引只有一个字长，它只能指向常量池中1～255（常量池位置0没有使用）范围内的位置。而ldc_w有两个字节长度的索引，因此，它能指向任何包含long类型或者double类型（占据两个字节长度）的常量池位置。把从常量池取出的常量压入栈的操作码如表10-5所示。

表10-5 将常量池入口压入栈

操作码	操作数	说明
ldc	indexbyte1	从由indexbyte1指向的常量池入口中取出一个字长的值，然后将其压入栈
ldc_w	indexbyte1，indexbyte2	从由indexbyte1和indexbyte2指向的常量池入口中取出一个字长的值，然后将其压入栈
ldc2_w	indexbyte1，indexbyte2	从由indexbyte1和indexbyte2指向的常量池入口中取出两个字长的值，然后将其压入栈

Java源代码中所有的字符串文字最终都作为入口存储于常量池中。如果同一个应用程序的多个类都使用同样的字符串文字，那么此字符串文字将在使用它的所有类的class文件中出现。例如，如果有三个类使用了字符串文字“Harumph!”，那么这个字符串将分别在这三个class文件的常量池中出现。这些类的方法可以使用ldc或者ldc_w指令来把指向一个有着“Harumph!”值的String对象的引用压入栈。

如第8章所述，Java虚拟机把所有具有相同字符顺序的字符串文字处理为同一个String对象。换句话说，如果多个类使用了同一个字符串文字，比如“Harumph!”，Java虚拟机将只会创建一个具有“Harumph!”值的String对象来表示所有的字符串文字。

当虚拟机解析一个字符串文字的常量池入口时，它“拘留”这个字符串。首先，虚拟机检查这个字符串中字符的顺序是否已经被拘留了。如果是这样，那么虚拟机就会使用与已拘留的字符串同样的引用。否则，它将会创建一个新的String对象，把对这个新String对象的引用加入到已拘留的字符串集合中去，然后，再把这个引用赋给新的已拘留的字符串。

10.2 通用栈操作

尽管Java虚拟机指令集中的大多数指令都只处理一种特定的类型，但还是有一些指令可以进行类型无关的栈操作。如第5章所述，这些通用（无类型）指令不能用于分解两个字长的值。这些指令如表10-6所示。

表10-6 栈操作

操作码	操作数	说明
nop	（无）	不做任何操作
pop	（无）	从操作数栈中弹出栈顶部的一个字
pop2	（无）	从操作数栈中弹出最顶端的两个字

（续）

操作码	操作数	说明
swap	（无）	交换栈顶部的两个字
dup	（无）	复制栈顶部的一个字
dup2	（无）	复制栈顶部的两个字
dup_x1	（无）	复制栈顶部的一个字，并将复制内容及原来弹出的两个字长的内容压入栈
dup_x2	（无）	复制栈顶部的一个字，并将复制内容及原来弹出的三个字长的内容压入栈
dup2_x1	（无）	复制操作数栈顶部的两个字，并将复制内容及原来弹出的三个字长的内容压入栈
dup2_x2	（无）	复制操作数栈顶部的两个字，并将复制内容及原来弹出的四个字长的内容压入栈

表10-6中的最后四种指令可能会有一点难以理解。可以参考附录A中对这些指令的描述，那里会对指令执行的前后栈状态进行描述。

10.3 把局部变量压入栈

有几个操作码用于把int类型和float类型局部变量压入栈。一些操作码隐式地指向一个通常使用的局部变量位置。例如，iload_0把int类型局部变量读入位置0。而其他局部变量则被一个从紧随操作码后第一个字节位置读取局部变量索引的操作码压入栈。

把int类型和float类型局部变量压入栈的操作码如表10-7所示。

表10-7 将一个字长的局部变量压入栈

操作码	操作数	说明
iload	vindex	将位置为vindex的int类型局部变量压入栈
iload_0	（无）	将位置为0的int类型局部变量压入栈
iload_1	（无）	将位置为1的int类型局部变量压入栈
iload_2	（无）	将位置为2的int类型局部变量压入栈
iload_3	（无）	将位置为3的int类型局部变量压入栈
fload	vindex	将位置为vindex的float类型局部变量压入栈
fload_0	（无）	将位置为0的float类型局部变量压入栈
fload_1	（无）	将位置为1的float类型局部变量压入栈
fload_2	（无）	将位置为2的float类型局部变量压入栈
fload_3	（无）	将位置为3的float类型局部变量压入栈

表10-8列出了将long类型和double类型的局部变量压入栈的指令。这些指令从栈帧的局部变量段向操作数栈段移动了两个字长的数据。

表10-8 将两个字长的局部变量压入栈

操作码	操作数	说明
lload	vindex	将位置为vindex和（vindex + 1）的long类型局部变量压入栈
lload_0	（无）	将位置为0和1的long类型局部变量压入栈
lload_1	（无）	将位置为1和2的long类型局部变量压入栈
lload_2	（无）	将位置为2和3的long类型局部变量压入栈
lload_3	（无）	将位置为3和4的long类型局部变量压入栈

（续）

操作码	操作数	说明
dload	vindex	将位置为vindex和（vindex + 1）的double类型局部变量压入栈
dload_0	（无）	将位置为0和1的double类型局部变量压入栈
dload_1	（无）	将位置为1和2的double类型局部变量压入栈
dload_2	（无）	将位置为2和3的double类型局部变量压入栈
dload_3	（无）	将位置为3和4的double类型局部变量压入栈

把局部变量压入栈的最后的操作码组从栈帧的局部变量段向操作数段移动对象引用（占据一个字长的空间）。这些操作码如表10-9所示。

表10-9 将对象引用局部变量压入栈

操作码	操作数	说明
aload	vindex	将位置为vindex的对象引用局部变量压入栈
aload_0	（无）	将位置为0的对象引用局部变量压入栈
aload_1	（无）	将位置为1的对象引用局部变量压入栈
aload_2	（无）	将位置为2的对象引用局部变量压入栈
aload_3	（无）	将位置为3的对象引用局部变量压入栈

10.4 弹出栈顶部元素，将其赋给局部变量

对于每个将局部变量压入栈的操作码而言，都存在相对应的弹出栈顶部元素并将其存储到局部变量中的操作码。执行弹出操作的操作码助记符可以通过把执行压入栈操作的操作码助记符中的“save”改为“load”的方式来表示。表10-10列出了从操作数栈顶部弹出int类型和float类型值并将其存储到局部变量中的操作码。这些操作码从栈顶部向局部变量移动一个字长的值。

表10-10 弹出一个字长的值，将其赋给局部变量

操作码	操作数	说明
istore	vindex	从栈中弹出int类型值，然后将其存到位置为vindex的局部变量中
istore_0	（无）	从栈中弹出int类型值，然后将其存到位置为0的局部变量中
istore_1	（无）	从栈中弹出int类型值，然后将其存到位置为1的局部变量中
istore_2	（无）	从栈中弹出int类型值，然后将其存到位置为2的局部变量中
istore_3	（无）	从栈中弹出int类型值，然后将其存到位置为3的局部变量中
fstore	vindex	从栈中弹出float类型值，然后将其存到位置为vindex的局部变量中
fstore_0	（无）	从栈中弹出float类型值，然后将其存到位置为0的局部变量中
fstore_1	（无）	从栈中弹出float类型值，然后将其存到位置为1的局部变量中
fstore_2	（无）	从栈中弹出float类型值，然后将其存到位置为2的局部变量中
fstore_3	（无）	从栈中弹出float类型值，然后将其存到位置为3的局部变量中

表10-11列出了弹出long类型和double类型值并将其存储到局部变量中的指令。这些指令从操作数栈顶部向局部变量移动两个字长的值。

表10-11 弹出两个字长的值，并将其赋给局部变量

操作码	操作数	说明
lstore	vindex	从栈中弹出long类型值，然后将其存到位置为vindex和（vindex + 1）的局部变量中
lstore_0	（无）	从栈中弹出long类型值，然后将其存到位置为0和1的局部变量中
lstore_1	（无）	从栈中弹出long类型值，然后将其存到位置为1和2的局部变量中
lstore_2	（无）	从栈中弹出long类型值，然后将其存到位置为2和3的局部变量中
lstore_3	（无）	从栈中弹出long类型值，然后将其存到位置为3和4的局部变量中
dstore	vindex	从栈中弹出double类型值，然后将其存到位置为vindex和（vindex + 1）的局部变量中
dstore_0	（无）	从栈中弹出double类型值，然后将其存到位置为0和1的局部变量中
dstore_1	（无）	从栈中弹出double类型值，然后将其存到位置为1和2的局部变量中
dstore_2	（无）	从栈中弹出double类型值，然后将其存到位置为2和3的局部变量中
dstore_3	（无）	从栈中弹出double类型值，然后将其存到位置为3和4的局部变量中

最后一组从栈弹出值并将其存储到局部变量中的操作码如表10-12所示。这些操作码从操作数栈顶弹出一个对象引用，并将其存储到局部变量中。

表10-12 弹出对象引用，并将其值赋给局部变量

操作码	操作数	说明
astore	vindex	从栈中弹出对象引用，然后将其存到位置为vindex的局部变量中
astore_0	（无）	从栈中弹出对象引用，然后将其存到位置为0的局部变量中
astore_1	（无）	从栈中弹出对象引用，然后将其存到位置为1的局部变量中
astore_2	（无）	从栈中弹出对象引用，然后将其存到位置为2的局部变量中
astore_3	（无）	从栈中弹出对象引用，然后将其存到位置为3的局部变量中

10.5 wide指令

无符号8位局部变量索引（比如iload指令后面的那个索引），把方法中局部变量数限制在256以下。一条单独的wide的指令可以将8位的索引再扩展8位，这样就可以把对局部变量数的限制扩展到65 536。wide操作码修改了其他的操作码。wide指令能够在诸如iload之类使用8位无符号局部变量索引的指令的前面执行。跟随在wide 操作码和修改过的操作码之后的两个字节组成指向局部变量的16位无符号索引。

表10-13列出了所有能够被wide指令修改的操作码（有两条例外） 例外的两条指令（iinc和ret）将在后面的章节中讨论。iinc指令和它的wide变量将在第12章讨论。ret指令和它的wide变量将在第18章讨论。

当验证包含wide指令的字节码序列时，被wide指令修改的操作码看上去像一个传递wide的操作数。跳转指令并不允许直接跳转到被wide指令修改过的操作码。例如，如果一个字节流序列包含了下面的指令

```
wide iload 257
```

那么在这个方法的字节流序列中，不会允许其他任何操作码直接跳转到iload 操作码的位置。在这种情况下，iload 操作码必须一直作为wide操作码的一个操作数来执行。

表10-13 弹出对象引用的值，并将其存储到局部变量中

操作码	操作数	说明
wide	iload，indexbyte1，indexbyte2	从局部变量位置为index的地方取出int类型值，并将其压入栈
wide	lload，indexbyte1，indexbyte2	从局部变量位置为index的地方取出long类型值，并将其压入栈
wide	fload，indexbyte1，indexbyte2	从局部变量位置为index的地方取出float类型值，并将其压入栈
wide	dload，indexbyte1，indexbyte2	从局部变量位置为index的地方取出double类型值，并将其压入栈
wide	aload，indexbyte1，indexbyte2	从局部变量位置为index的地方取出对象引用，并将其压入栈
wide	istore，indexbyte1，indexbyte2	从栈中弹出int类型值，然后将其存入位置为index的局部变量中
wide	lstore，indexbyte1，indexbyte2	从栈中弹出long类型值，然后将其存入位置为index的局部变量中
wide	fstore，indexbyte1，indexbyte2	从栈中弹出float类型值，然后将其存入位置为index的局部变量中
wide	dstore，indexbyte1，indexbyte2	从栈中弹出double类型值，然后将其存入位置为index的局部变量中
wide	astore，indexbyte1，indexbyte2	从栈中弹出对象引用，然后将其存入位置为index的局部变量中

10.6 一个模拟："Fibonacci Forever"

如图10-1所示的applet演示了Java虚拟机执行一个产生斐波那契序列的字节码序列的过程。此applet嵌在光盘上的applets/FibonacciForever.html网页文件中。下面类中calcSequence（）方法的字节码序列是由javac编译器产生的。

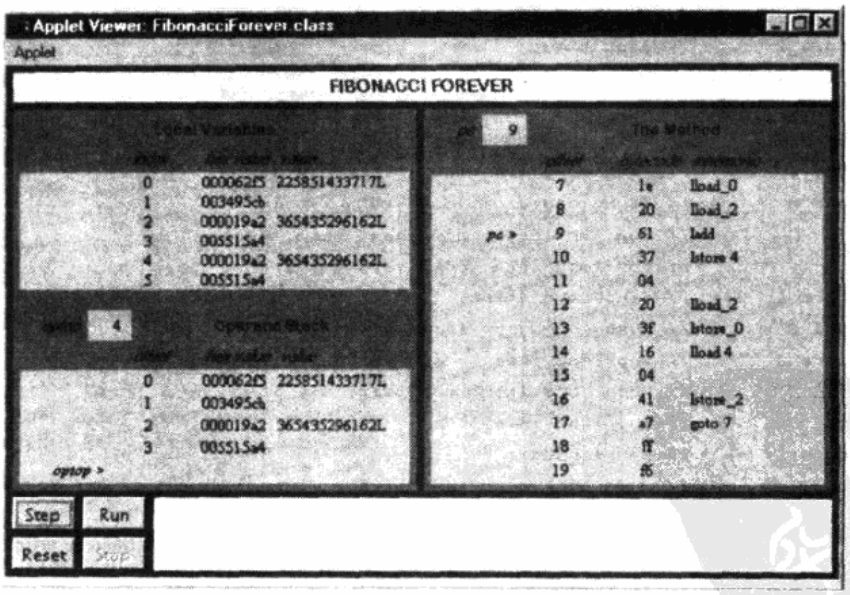


图10-1 "Fibonacci Forever" applet

```
// On CD-ROM in file stackops/ex1/Fibonacci.java
class Fibonacci {

    static void calcSequence() {
```

```
        long fiboNum = 1;
        long a = 1;
        long b = 1;

        for (;;) {
            fiboNum = a + b;
            a = b;
            b = fiboNum;
        }
    }
}
```

calcSequence ( ) 方法产生了斐波那契序列，并将每一个产生的斐波那契数都成功地存储在fiboNum变量中。序列中开头的两个斐波那契数都只是1，后续每一个斐波那契数都是前面两个数相加的结果，例如：1，1，2，3，5，8，13，21，34，55，……

javac产生的calcSequence ( ) 方法的字节码如下所示：

```
 0 lconst_1        // Push long constant 1
                   // Pop long into local vars 0 & 1:
 1 lstore_0        // long a = 1;
 2 lconst_1        // Push long constant 1
                   // Pop long into local vars 2 & 3:
 3 lstore_2        // long b = 1;
 4 lconst_1        // Push long constant 1
                   // Pop long into local vars 4 & 5:
 5 lstore 4        // long fiboNum = 1;
 7 lload_0         // Push long from local vars 0 & 1
 8 lload_2         // Push long from local vars 2 & 3
 9 ladd            // Pop two longs, add them, push result
                   // Pop long into local vars 4 & 5:
10 lstore 4        // fiboNum = a + b;
12 lload_2         // Push long from local vars 2 & 3
13 lstore_0        // Pop long into local vars 0 & 1: a = b;
14 lload 4         // Push long from local vars 4 & 5
                   // Pop long into local vars 2 & 3:
16 lstore_2        // b = fiboNum;
17 goto 7          // Jump back to offset 7: for (;;) {}
```

javac编译器把局部变量a从源文件中放到了栈帧中0和1位置的局部变量中，把b放到了栈帧中2和3的位置，把fiboNum放到了4和5的位置。每当计算出一个连续的斐波那契数，javac编译器就会把这个数字存储到fiboNum变量中。因此，运行的时候会发现斐波那契序列以long类型值的形式存放在4和5位置的局部变量中。

也许会注意到，long类型值分为两段存放，低位（0～31位）存放在第一个位置，高位（32～63位）存放在第二个位置。例如，fiboNum变量的低位存放在位置4的局部变量中，高位存放在位置5的局部变量中。操作数栈中也一样，当一个long类型值被压入栈，低位将会首先被压入

栈，接下来再轮到高位。

需要记住的是：这种在局部变量和操作数栈中表示long类型值的方式只是对Java虚拟机的特定实现的一个模拟。如第5章所述，规范并没有规定任何特定的方法来确定如何在栈帧中放置long类型和double类型的两个字。

尽管已经在数学上证明，斐波那契序列是无穷的，但calcSequence（）方法只能在一段有限的时间内产生斐波那契数，而且long类型也有范围。这个模拟所能运算出来的最大的斐波那契数，也就是long类型能够表示的最大的斐波那契数是7540113804746346429L。当模拟计算到这个值时，下一步计算将会溢出。

使用“Step”、“Reset”、“Run”和“Stop”按钮即可启动“Fibonacci Forever”模拟。当点击“Step”按钮时，模拟就会执行PC寄存器指向的指令。当点击“Run”按钮时，模拟会继续进行下一步模拟，直到点击“Stop”按钮才会停止。点击“Reset”按钮，会重新开始模拟。对于每一步模拟，applet底部的面板都将显示出对下一步指令所做工作的解释。

10.7 随书光盘

光盘上的stackops目录中有本章源代码示例。光盘上的applets/FibonacciForever.html网页文件中包含了这个“Fibonacci Forever”applet。这个applet的源代码和它的class文件一起放在applets/JVMSimulators和applets/JVMSimulators/COM/artima/jvmsim目录。

10.8 资源页

如果需要了解更多与本章相关的信息，请访问资源页面http://www.artima.com/insidejvm/resources。

第11章 类 型 转 换

本章介绍了Java虚拟机中实现不同基本数据类型值之间转换功能的指令，还介绍了由于Java虚拟机对byte、short和char等类型的有限支持所产生的一些类型转换的特征。

随书光盘上有一个使用交互方式来阐述本章内容的applet，名为“Conversion Diversion”。该applet模拟了Java虚拟机执行类型转换的过程，本章结尾处介绍了此applet和它所执行的字节码。

11.1 转换操作码

Java虚拟机包括许多进行基本类型转换工作的操作码，这些执行转换工作的操作码后面没有操作数，转换的值从栈顶端获得。Java虚拟机从栈顶端弹出一个值，对它进行转换，然后再把转换结果压入栈。进行int、long、float和double类型之间转换的操作码如表11-1所示，针对这四种类型之间的每一种可能的类型转换，都存在相应的操作码。

表11-1 int、long、float和double类型之间的相互转换

操 作 码	操 作 数	说 明
i2l	（无）	将int类型值转换为long类型值
i2f	（无）	将int类型值转换为float类型值
i2d	（无）	将int类型值转换为double类型值
l2i	（无）	将long类型值转换为int类型值
l2f	（无）	将long类型值转换为float类型值
l2d	（无）	将long类型值转换为double类型值
f2i	（无）	将float类型值转换为int类型值
f2l	（无）	将float类型值转换为long类型值
f2d	（无）	将float类型值转换为double类型值
d2i	（无）	将double类型值转换为int类型值
d2l	（无）	将double类型值转换为long类型值
d2f	（无）	将double类型值转换为float类型值

如表11-2所示的操作码是把int类型转换为比int类型占据更小空间的数据类型。这些操作码从操作数栈中弹出一个int类型值，将它转换为能用byte、short或char类型描述的int类型值，然后再把这个转换后的int类型值压入栈。i2b指令将弹出的int类型值截短为byte类型，然后再对其进行带符号扩展，恢复成int类型。i2s指令将弹出的int类型值截短为short类型，然后再对其进行带符号扩展，恢复成int类型。i2c指令将弹出的int类型值截短为char类型，然后再对其进行零扩展，恢复成int类型。

表11-2 int数据类型向byte、char、short类型的转换

操作码	操作数	说明
i2b	（无）	将int类型值转换为byte类型值
i2c	（无）	将int类型值转换为char类型值
i2s	（无）	将int类型值转换为short类型值

Java虚拟机中没有把long、float、double类型值直接转换为比int类型占据更小空间的数据类型的操作码。因此，把float类型值转换为byte类型需要两个步骤：首先，float类型值必须通过f2i指令转换为int类型值，然后，所得的int类型值再通过i2b指令转换成byte类型值。

尽管有操作码可以把int类型值转换为比int类型值占据更小空间的数据类型（byte、short和char），但并不存在执行相反方向转换操作的操作码。因为任何byte、short和char类型值在压入栈的时候，就已经有效地被转换成int类型值了。从数组或者堆中的对象中接受byte、short和char类型值的指令和把这些值压入栈的指令都会把它们转换为int类型值。这些指令将在第15章叙述。

涉及byte、short和char类型的运算操作首先会把这些值转换为int类型，然后对int类型值进行运算，最后得到int类型的结果。因此，如果把两个byte类型值相加，最后会得到一个int类型的结果。如果需要得到byte类型结果，必须将这个int类型的结果显式转换为byte类型。例如，下面的代码会导致编译失败：

```
// On the CD-ROM in file opcodes/ex1/BadArithmetic.java
class BadArithmetic {

    static byte addOneAndOne() {
        byte a = 1;
        byte b = 1;
        byte c = a + b;
        return c;
    }
}
```

当遇到上述代码时，javac会给出如下提示：

```
BadArithmetic.java(7): Incompatible type for declaration.
Explicit cast needed to convert int to byte.
        byte c = a + b;
                 ^
```

为了对这种情况进行补救，必须把a + b所获得的int类型结果显式转换为byte类型。代码如下：

```
// On the CD-ROM in file opcodes/ex1/GoodArithmetic.java
class GoodArithmetic {

    static byte addOneAndOne() {
        byte a = 1;
        byte b = 1;
```

```
        byte c = (byte) (a + b);
        return c;
    }
}
```

该操作能够通过javac的编译，并产生GoodArithmetic.class文件，此文件包含了如下的addOneAndOne（）方法的字节码序列：

```
 0 iconst_1        // Push int constant 1.
 1 istore_0        // Pop into local variable 0, which is a:
                   // byte a = 1;
 2 iconst_1        // Push int constant 1 again.
 3 istore_1        // Pop into local variable 1, which is b:
                   // byte b = 1;
 4 iload_0         // Push a (a is already stored as an int in
                   // local
                   // variable 0).
 5 iload_1         // Push b (b is already stored as an int in
                   // local
                   // variable 1).
 6 iadd            // Perform addition. Top of stack is
                   // now (a + b), an int.
 7 i2b             // Convert int result to byte (result still
                   // occupies 32 bits).
 8 istore_2        // Pop into local variable 3, which is
                   // byte c: byte c = (byte) (a + b);
 9 iload_2         // Push the value of c so it can be
                   // returned.
10 ireturn         // Proudly return the result of the
                   // addition: return c;
```

11.2 一个模拟："Conversion Diversion"

如图11-1所示的"Conversion Diversion" applet演示了Java虚拟机执行字节码序列的过程。此applet嵌在随书光盘上applets/ConversionDriversion.html的网页文件里。模拟中javac产生的下列类的Convert（）方法的字节码序列如下所示：

```
// On CD-ROM in file

 opcodes/ex1/Diversion.java
class Diversion {

    static void Convert() {

        byte imByte = 0;
        int imInt = 125;
```

```
        for (;;) {
            ++imInt;
            imByte = (byte) imInt;

            imInt *= -1;
            imByte = (byte) imInt;
            imInt *= -1;
        }
    }
}
```

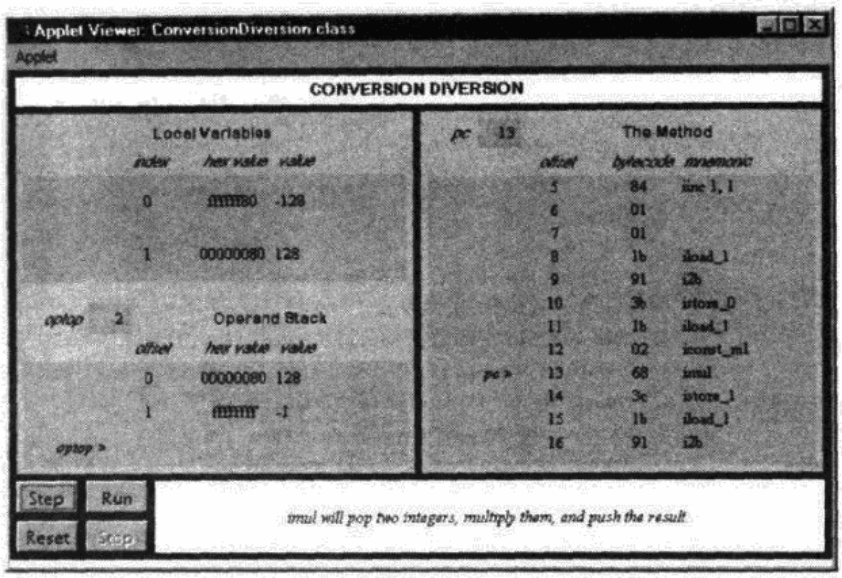


图11-1 “Conversion Diversion”applet

由javac产生的Convert（）的字节码如下所示：

```
 0 iconst_0       // Push int constant 0.
                  // Pop to local variable 0, which is
 1 istore_0       // imByte: byte imByte = 0;
 2 bipush 125     // Expand byte constant 125 to int and push.
                  // Pop to local variable 1, which
4 istore_1        // is imInt: int imInt = 125;
                  // Increment local variable 1 (imInt) by 1:
 5 iinc 1 1       // ++imInt;
 8 iload_1        // Push local variable 1 (imInt).
                  // Truncate and sign extend top of stack so
 9 i2b            // it has a valid byte value.
                  // Pop to local variable 0 (imByte):
10 istore_0       // imByte = (byte) imInt;
```

```
11 iload_1          // Push local variable 1 (imInt) again.
12 iconst_m1        // Push integer -1.
13 imul             // Pop top two ints, multiply, push result.
                    // Pop result of multiply to local variable
14 istore_1         // 1 (imInt): imInt *= -1;
15 iload_1          // Push local variable 1 (imInt).
                    // Truncate and sign extend top of stack
16 i2b              // so it has a valid byte value.
                    // Pop to local variable 0 (imByte):
17 istore_0         // imByte = (byte) imInt;
18 iload_1          // Push local variable 1 (imInt) again.
19 iconst_m1        // Push integer -1.
20 imul             // Pop top two ints, multiply, push result.
                    // Pop result of multiply to local variable
21 istore_1         // 1 (imInt): imInt *= -1;
                    // Jump back to the iinc instruction:
22 goto 5           // for (;;) {}
```

该Convert（）方法演示了Java虚拟机中将int类型值转换为byte类型值的方式。imInt变量开始的值为125，每经过一次循环，它都会自增1，并转换为byte类型。然后，它的值变为原值与-1的乘积，最后再次转换为byte类型。这个模拟过程可以快速地显示出在byte类型有效范围的边界上发生的事情。

byte类型的最大值为127，最小值为-128。在这个范围内的int类型值被直接转换为byte类型值，而当int类型值超出这个有效范围时，事情就变得有趣起来。

Java虚拟机通过截短和带符号扩展的方式将int类型值转换成为byte类型值。long、int、short和byte类型的最高位（“符号位”）指出此int类型值是正还是为负。如果符号位为0，值为正；如果符号位为1，值为负。byte类型的第7位为符号位。从int类型值转换到byte类型值的时候，第7位的值将会被拷贝到第8位到第31位。这样就产生了一个int类型值，这个值与原来int类型值被转换为byte类型值后所获得的结果具有相同的数值。在执行完截短和带符号扩展操作后，这个int类型变量中将容纳一个有效的byte类型的值。

applet模拟列出了一个超出byte类型有效范围的int类型值转换为byte类型时所发生的事情。例如，imInt变量的值为128（0x00000080），它被转换为byte类型后，所得到的byte类型值为-128（0xffffff80）。然后，当imInt变量的值为-129（0xffffff7f）时，它被转换为byte类型后所得到的值为127（0x0000007f）。

使用“Step”、“Reset”、“Run”和“Stop”按钮即可启动“Conversion Diversion”模拟。当点击“Step”按钮时，模拟就会执行PC寄存器指向的指令。当点击“Run”按钮时，模拟会继续进行下一步模拟，直到点击“Stop”按钮才会停止。点击“Reset”按钮，会重新开始模拟。对于每一步模拟，applet底部的面板都将显示出对下一步指令所做工作的解释。

11.3 随书光盘

光盘上opcodes子目录中有本章源代码示例，applets/Conversion Diversion.html网页文件里包

含了该“Conversion Diversion” applet。这个applet的源代码和它的class文件一起放在applets/JVMSimulators和applets/JVMSimulators/COM/artima/jvmsim目录。

11.4 资源页

如果需要了解更多与本章相关的信息，请访问资源页面http://www.artima.com/insidejvm/resources。

第12章 整数运算

本章介绍了Java虚拟机中的整数运算，包括二进制补码运算（Java虚拟机用来进行整数运算的机制）和执行整数运算的指令。

随书光盘上有两个使用交互方式来阐述本章内容的applet：名为“Inner Int”的applet能够观察和操作补足二进制补码数部分；另一个名为“Prime Time”的applet模拟了Java虚拟机产生素数的过程，本章结尾处介绍了这个applet和它所执行的字节码。

12.1 二进制补码运算

Java虚拟机所支持的所有的整数类型——byte、short、int和long，它们都是带符号的二进制补码数。二进制补码方案既能够描述正整数，也能够描述负整数。在一个二进制补码数中，最重要的位就是它的符号位。符号位为1，表示负整数；符号位为0，表示正整数和数字0。

能够被二进制补码方案表示的数的范围为：2的总位数次幂。例如，在Java中，short类型是16位带符号的二进制补码整数。能够惟一表示的整数数为：2^{16}，或者65 536。short类型值范围的一半被用来表示0和正整数，另一半被用来表示负数。16位二进制补码负数的范围是-32 768（0x8000）~-1（0xffff），零用0x0000来表示，整数的范围是1（0x0001）~32 767（0x7fff）。

正整数直觉上只不过是数的两种表示法之一。负数可以通过负数和2的某次方幂相加而得出。例如，short类型的长度为16位，因此二进制补码表示法可以通过一个负数和2的16次幂（或者65 536）的相加来得到一个有效范围内（-32 768~-1）的负数。-1的二进制补码表示为65 536+（-1）或者65 535（0xffff）。-2的二进制补码表示为65 536+（-2）或者65 534（0xfffe）。

在带符号二进制补码数上进行的加法运算与在无符号二进制数上进行的加法运算一样。两个数相加（忽略溢出），结果被解释为一个带符号二进制补码数。这个过程将在运算结果是在该类型的有效范围内的情况下进行。例如，要获得4+（-2）的结果，只要把0x00000004和0xfffffffe相加即可。结果是0x100000002，但是因为int类型只有32位，于是溢出部分被忽略，结果为0x00000002。

Java虚拟机中出现的整数运算的溢出并不会导致抛出异常，其结果只被简单地截短以符合数据类型（或者为int类型，或者为long类型）。例如，把int类型值0x7fffffff和1相加，将会得到0x80000000。因此，如果相加值的类型为int而非long，Java虚拟机中2 147 483 647加上1的结果将会是-2 147 483 648。在Java中编程时，必须随时注意可能发生的溢出。必须在每一种情况下确认所选择的数据类型（int或者long）是否正确。整数被0除时会抛出一个ArithmeticException异常，所以应该时刻牢记此类异常将会被抛出，必须在必要的时候捕获异常。

如果long类型的长度仍然不能满足需要，可以使用java.math包中的BigInteger类，这个类的实例可以描述任意长度的整数。BigInteger类支持在任意长度整数上进行的所有数学运算，前提是这些运算是基于Java虚拟机和java.lang.Math包所支持的基本类型的。

12.2 Inner Int：揭示Java int类型内部性质的applet

如图12-1所示的“Inner Int”applet让你有机会试一下Java虚拟机中二进制补码格式的整数运算。“Max”和“Min”按钮能够给出int类型的最大值与最小值。在点击“Max”按钮后，继续点击“++”按钮，就可以得到最大值加1的结果。点击“Min”按钮后，继续点击“--”按钮，就能够得到最小值减1的结果。这些操作都会导致溢出的产生，但是Java虚拟机并没有抛出异常。此applet嵌在光盘上的applets/InnerInt.html网页文件中。

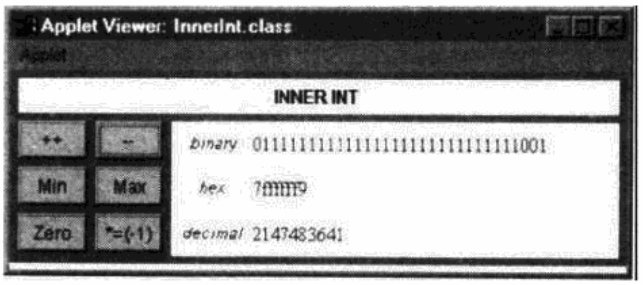


图12-1 “Inner Int”applet

12.3 运算操作码

Java虚拟机提供几种进行整数算术运算的操作码，它们执行基于int和long类型的运算。如前所述，当byte、short和char类型值参与算术运算时，首先会将它们转换为int类型。对于每一个执行int类型算术运算的操作码，在long类型的相同运算中有对应的操作码。

整数加法可以在int和long类型上进行。表12-1描述了完成下列任务的操作码：弹出栈顶部的两个值，相加，把结果压入栈。必须有指令先把两个相加的值压入栈。值的类型由操作码自己指定，最后得到的结果总是与相加的成员具有同样的类型。这些操作码都不会导致任何异常抛出，溢出在这里通常被忽略。

前面我们得到了这样一条规则：运算操作码从栈中取出它们的操作数，但在表12-2中显示了一个例外情况，即iinc操作码对int类型局部变量的加法操作。用于加法运算的局部变量位于字节码流中紧接于iinc指令之后的第一个字节，将要加到局部变量上的值从iinc指令之后的第二个字节取出。第二个字节被解释为一个8位的带符号二进制补码数。局部变量和8位带符号值相加，相加的结果被写回局部变量。这条操作码可以用来给局部变量赋-128～127之间的值。这条操作码与用于控制循环（for或者while）执行的变量的加减相比，效率更高。与加法指令一样，这里没有任何异常抛出，溢出通常被忽略。

表12-2的第二行说明了iinc指令的wide变量。如第10章中所述，通过使用wide指令，可以把无符号局部变量索引从8位扩展到16位。使用16位索引的指令可以对多达65 536个位置的变量寻址。在iinc指令的处理过程中，wide指令也是用来把带符号的增量值从8位扩展到16位。因此，iinc操作码的wide变量可以在-32 768～32 767范围内改变一个局部变量的值。

表12-1 整数加法

操作码	操作数	说明
iadd	（无）	从栈中弹出两个int类型数，相加，然后将所得int类型结果压回栈
ladd	（无）	从栈中弹出两个long类型数，相加，然后将所得long类型结果压回栈

表12-2 将一个常量与局部变量相加

操作码	操作数	说明
iinc	vindex，const	把常量与一个位于vindex位置的int类型局部变量相加
wide	iinc，indexbyte1，indexbyte2，constbyte1，constbyte2	把常量与一个位于index位置的int类型局部变量相加

如表12-3所示的操作码执行int和long类型的整数减法运算。每个操作码都会从栈中弹出两个相同类型的值，顶端的值充当减数，底端的值充当被减数，进行减法运算，结果被压回栈。此类操作不会导致抛出异常。

表12-3 整数减法

操作码	操作数	说明
isub	（无）	从栈中弹出两个int类型数，相减，然后将所得int类型结果压回栈
lsub	（无）	从栈中弹出两个long类型数，相减，然后将所得long类型结果压回栈

如表12-4所示的操作码执行int和long类型的整数乘法运算。每一个操作码都会从栈中弹出两个相同类型的值并相乘，结果与相乘的两个值具有同样类型，并被压回栈。此类操作不会导致异常抛出。

表12-4 整数乘法

操作码	操作数	说明
imul	（无）	从栈中弹出两个int类型数，相乘，然后将所得int类型结果压回栈
lmul	（无）	从栈中弹出两个long类型数，相乘，然后将所得long类型结果压回栈

如表12-5所示的操作码执行int和long类型的整数除法运算。除法的操作码从栈中弹出两个相同类型的值，底端的数除以栈顶端的数（换句话说，首先被压入栈的数作为被除数或者分子，其次被压入的数——栈顶部的数——作为除数或者分母）。结果被压回栈。对于整数除法所产生的结果将进行取整操作。如果整数被0除，则会抛出ArithmeticException异常。

表12-5 整数除法

操作码	操作数	说明
idiv	（无）	从栈中弹出两个int类型数，相除，然后将所得int类型结果压回栈
ldiv	（无）	从栈中弹出两个long类型数，相除，然后将所得long类型结果压回栈

如表12-6所示的操作码执行int和long类型的取余运算。这两种操作码从栈中弹出两个值，底端的数作为被除数或者分子，栈顶端的数作为除数或者分母，除法的余数被压回栈。整数取余操作时，如果除数为0，将会抛出ArithmeticException异常。

表12-6 整数取余

操作码	操作数	说明
irem	（无）	从栈中弹出两个int类型数，相除，然后将所得int类型余数压回栈
lrem	（无）	从栈中弹出两个long类型数，相除，然后将所得long类型余数压回栈

如表12-7所示的操作码进行int和long类型的取反运算。取反操作码把栈顶部的值从栈弹出，取反，把结果压回栈。

表12-7 整数取反

操作码	操作数	说明
ineg	（无）	从栈中弹出一个int类型数，取反，然后将所得int类型结果压回栈
lneg	（无）	从栈中弹出一个long类型数，取反，然后将所得long类型结果压回栈

12.4 一个模拟："Prime Time"

如图12-2所示的"Prime Time" applet演示了Java虚拟机执行一个产生素数的字节码序列的过程。这个applet嵌在随书光盘上的applets/PrimeTime.html网页文件里。由javac编译器产生的下列类的findPrimes（）方法的字节码序列如下所示：

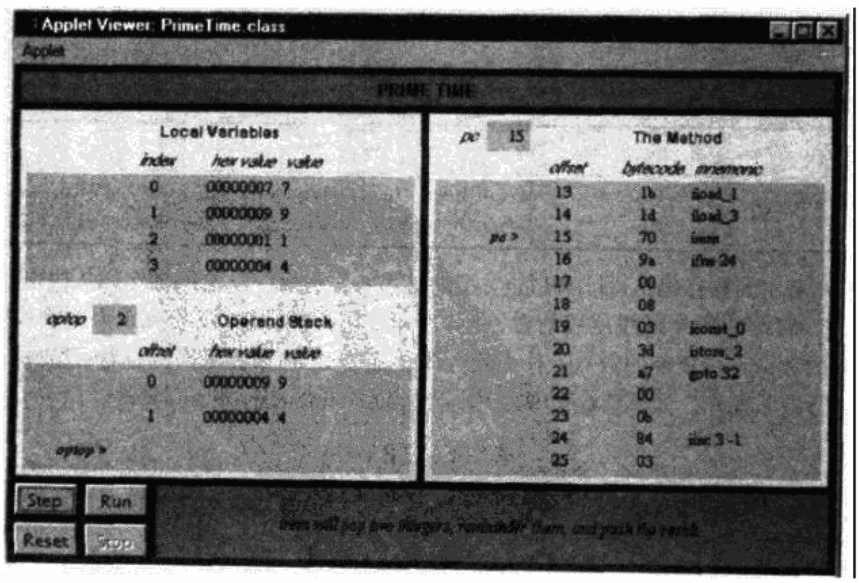


图12-2 "Prime Time" applet

```
// On CD-ROM in file integer/ex1/PrimeFinder.java
class PrimeFinder {

    static void findPrimes() {

        int primeNum = 1;
        int numToCheck = 2;

        for (;;) {

            boolean foundPrime = true;

            for (int divisor = numToCheck / 2;
                divisor > 1; --divisor) {

                if (numToCheck % divisor == 0) {
                    foundPrime = false;
                    break;
                }
            }

            if (foundPrime) {
                primeNum = numToCheck;
            }

            ++numToCheck;
        }
    }
}
```

findPrimes（）方法每次产生一个素数，按照递增的顺序，把产生的素数值赋给primeNum变量。为了查找素数，该方法从2开始，按照递增的顺序检查每个正整数。findPrimes（）方法把当前正在检查的数放在numToCheck变量中，外层循环是一个死循环，这样就可以让计算素数的过程无限延续下去。这里使用如下方法来判定一个数是否为素数：依次用比该数小的数除该数，如果余数为0，那么这个数就有除了1和它自身以外的其他因子，那么这个数就肯定不是素数。

为了对每一个数执行检查，判断其是否为素数，findPrimes（）方法首先用2除当前数，这个整数除法所得的结果作为此数的可能存在的整数因子第一个被检查。

在内层循环中，findPrimes（）方法使用从该数的1/2到2之间的每一个正整数作为因子进行试验，如果取余操作的结果为0，那么就结束内层循环，进行下一次外层循环，检查下一个数。如果试验到2还没有找到该数的因子，那么就说明发现了一个素数，于是内层循环结束，这个素数值被赋给primeNum，然后进行下一个数的检查。

例如，当numToCheck为10的时候，findPrimes（）首先使用2除10来获得第一个可能的因子

5。对10和5进行取余操作，产生了一个值为0的余数，之后会中断内层循环，把numToCheck的值设为11（不会把primeNum设为10）。接着使用2除11来获得第一个用于检查的因子，这一次得到的又是5。然后分别对11和整数5，4，3，2进行取余操作，没有一次操作能够产生值为0的余数，于是内层循环结束，primeNum的值被设为11，接下来进行对12的检查。

由javac产生的findPrimes（）方法的字节码如下所示：

```
 0 iconst_1      // Push int constant 1
                 // Pop into local var 0:
 1 istore_0      // int primeNum = 1;
 2 iconst_2      // Push int constant 2
                 // Pop into local var 1:
 3 istore_1      // int numToCheck = 2;

// The outer for loop (the "forever" loop) begins here:
 4 iconst_1      // Push int constant 1
                 // Pop into local var 2:
 5 istore_2      // boolean foundPrime = true;

// The inner for loop begins here. First, initialize
// divisor.
 6 iload_1       // Push int in local var 1 (numToCheck)
 7 iconst_2      // Push int constant 2
                 // Pop two ints, divide them, push int
 8 idiv          // result
                 // Pop int into local var 3:
 9 istore_3      // int divisor = numToCheck / 2;

// Next, test the inner for loop's termination condition
10 goto 27       // Jump to for loop condition check

// The body of the inner for loop begins here.
                 // Push the int in local var 1
13 iload_1       // (numToCheck)
14 iload_3       // Push the int in local var 3 (divisor)
                 // Pop two ints, remainder them, push
15 irem          // result
                 // Pop int, jump if equal to zero:
16 ifne 24       // if (numToCheck % divisor == 0)
19 iconst_0      // Push int constant 0
                 // Pop into local var 2:
20 istore_2      // foundPrime = false;
21 goto 32       // Jump out of inner for loop
```

```
// At this point, the body of the inner for loop is done.
// Now just perform the third statement of the for
// expression: decrement divisor.
24 iinc 3 -1      // Increment local var 3 by -1: -divisor

// The test for the inner for loop's termination condition
// begins here. This loop will keep on looping while
// (divisor > 1).
27 iload_3        // Push int from local var 3 (divisor)
28 iconst_1       // Push int constant 1
29 if_icmpgt 13   // Pop top two ints, jump if greater than

// At this point, the inner for loop has completed. Next
// check to see if a prime number was found.
32 iload_2        // Push int from local var 2 (foundPrime)
                  // Pop top int, jump if zero:
33 ifeq 38        // if (foundPrime) {
36 iload_1        // Push int from local var 1 (numToCheck)
                  // Pop into local var 0:
37 istore_0       // primeNum = numToCheck;
                  // Increment local var 1 by 1:
38 iinc 1 1       // ++numToCheck;
41 goto 4         // Jump back to top of outer for loop.
```

javac编译器把源代码中的primeNum局部变量放在栈结构中的位置0。它把numToCheck放在位置1，foundPrime放在位置2，divisor放在位置3。如前所述，当此方法成功发现一个素数时，它会把这个数放在primeNum变量中。因此，运行这个模拟时，素数就会以int类型值的形式一个接一个地出现在存储在0位置的局部变量中。

值得注意的是，该字节码序列描述了在Java字节码的栈结构内处理Java源代码中boolean类型的方式。存储在位置2的局部变量中的值表示源代码中的boolean foundPrime变量，它是一个int类型。它通过压入一个int常量为1或者0的指令设置为true或者false；通过执行int类型值与0进行比较的指令来检查boolean类型的值。

关于这个模拟需要注意的另外一点是，最后numToCheck变量将会溢出。当溢出发生时，虚拟机不会抛出任何异常，findPrimes（）方法将会继续执行，并产生和素数不再有任何关系的int类型值。

使用“Step”、“Reset”、“Run”和“Stop”按钮即可启动“Prime Time”模拟。当点击“Step”按钮时，模拟就会执行PC寄存器指向的指令。当点击“Run”按钮时，模拟会继续进行下一步模拟运算，直到点击“Stop”按钮才会停止。点击“Reset”按钮，会重新开始模拟运算。对于每一步模拟，applet底部的面板都将显示出对下一步指令所做工作的解释。

12.5 随书光盘

光盘上integer子目录中有本章源代码示例。光盘上的applets/PrimeTime.html网页文件里包含了“Prime Time” applet。这个applet的源代码和它的class文件一起放在applets/ JVMSimulators目录。

12.6 资源页

如果需要了解更多与本章相关的信息，请访问资源页面http://www.artima.com/insidejvm/resources。

第13章 逻辑运算

本章介绍了Java虚拟机中执行位逻辑运算的指令，包括对整数进行移位和布尔运算的操作码。布尔运算针对整数值的每一位进行操作。

随书光盘上有一个使用交互方式来阐述本章内容的applet，名为“Logical Result”。这个applet模拟了Java虚拟机执行使用一些逻辑操作码的方法的过程，本章结尾处介绍了这个applet和它所执行的字节码。

13.1 逻辑操作码

Java虚拟机的逻辑操作主要针对int和long类型。这些处理int和long类型的操作与处理带符号二进制补码数的方式不同，而是按照通用的位模式进行处理。整数移位通过ishl、ishr、iushr操作码进行。Java的“<<”操作符由ishl实现，“>>”操作符由ishr实现，“>>>”操作符由iushr实现。ishr和iushr之间的不同在于：ishr进行符号扩展。表13-1列出了对int类型值进行左右移位操作的指令。

表13-1 对int类型值进行移位操作

操作码	操作数	说明
ishl	（无）	向左对int类型值进行移位操作
ishr	（无）	向右对int类型值进行算术移位操作
iushr	（无）	向右对int类型值进行逻辑移位操作

表13-2列出了对long类型值进行左右移位操作的指令。

表13-2 对long类型值进行移位操作

操作码	操作数	说明
lshl	（无）	向左对long类型值进行移位操作
lshr	（无）	向右对long类型值进行算术移位操作
lushr	（无）	向右对long类型值进行逻辑移位操作

如表13-3所示的操作码在int类型上执行位逻辑运算，这些操作码实现了Java的“&”，“|”和“^”操作。

表13-3 对int类型值进行位逻辑操作

操作码	操作数	说明
iand	（无）	对两个int类型值进行逻辑“与”操作
ior	（无）	对两个int类型值进行逻辑“或”操作
ixor	（无）	对两个int类型值进行逻辑“异或”操作

如表13-4所示的操作码在long类型上执行位逻辑运算。

表13-4 对long类型值进行位逻辑操作

操作码	操作数	说明
land	（无）	对两个long类型值进行逻辑“与”操作
lor	（无）	对两个long类型值进行逻辑“或”操作
lxor	（无）	对两个long类型值进行逻辑“异或”操作

如前所述，在Java虚拟机中，并不存在本地的boolean类型。Java虚拟机使用int类型表示boolean类型。指令集中有许多将int类型转换为boolean类型，再根据其值决定是否进行跳转的指令，第16章中介绍了这些指令。

13.2 一个模拟：“Logical Results”

如图13-1所示的“Logical Results”applet演示了Java虚拟机执行一段字节码序列的过程。这个applet嵌在随书光盘上的applets/LogicalResults.html网页文件里。模拟中由javac产生的VulcanCounter类的incrementLogically（）方法的字节码序列如下所示：

```
// On CD-ROM in file opcodes/ex1/VulcanCounter.java
class VulcanCounter {

    static void incrementLogically() {
        int spock = 0;
        for (;;) {
            int tempSpock = spock;
            for (int i = 0; i < 32; ++i) {
                int mask = 0x1 << i;
                if ((tempSpock & mask) == 0) {
                    tempSpock |= mask;  // Change 0 to 1
                    break;
                }
                else {
                    tempSpock &= ~mask; // Change 1 to 0
                }
            }
            spock = tempSpock;
        }
    }
}
```

由javac为incrementLogically（）方法所产生的字节码如下所示：

```
0 iconst_0      // Push int constant 0.
1 istore_0      // Pop to local variable 0: int spock = 0;
2 iload_0       // Push local variable 0 (spock).
                // Pop to local variable 1:
```

```
 3 istore_1       // int tempSpock = spock;
 4 iconst_0       // Push int constant 0.
 5 istore_2       // Pop to local variable 2: int i = 0;
 6 goto 35        // Jump unconditionally ()
 9 iconst_1       // Push int constant 1.
10 iload_2        // Push local variable 2 (i).
                  // Arithmetic shift left top int (i) by
11 ishl           // next to top int (1).
                  // Pop to local variable 3:
12 istore_3       // int mask = i << 0x1;
13 iload_1        // Push local variable 1 (tempSpock).
14 iload_3        // Push local variable 3 (mask).
                  // Bitwise AND top two ints:
15 iand           // (spock & mask)
                  // Jump if top of stack is not equal to
16 ifne 26        // zero: if ((spock & mask) == 0) {
19 iload_1        // Push local variable 1 (tempSpock).
20 iload_3        // Push local variable 3 (mask).
                  // Bitwise OR top two ints
21 ior            // (tempSpock | mask)
                  // Pop to local variable 1:
22 istore_1       // tempSpock |= mask;
                  // Jump unconditionally (to just after
23 goto 41        // inner for): break;
26 iload_1        // Push local variable 1 (tempSpock).
27 iload_3        // Push local variable 3 (mask).
28 iconst_m1      // Push -1.
                  // Bitwise EXCLUSIVE-OR top two ints:
29 ixor           // ~mask
                  // Bitwise AND top two ints:
30 iand           // tempSpock & (~mask)
31 istore_1       // Pop to local variable 1: tempSpock &= ~mask;
32 iinc 2 1       // Increment local variable 2 by 1: ++i
35 iload_2        // Push local variable 2 (i).
36 bipush 32      // Push integer constant 32.
                  // Jump (to top of inner for) if "next to
                  // top" integer is less than "top"
38 if_icmplt 9    // integer: i < 32
41 iload_1        // Push local variable 1 (tempSpock)
                  // Pop to local variable 0:
42 istore_0       // spock = tempSpock;
                  // Jump unconditionally (to top of outer
43 goto 2         // for).
```

incrementLogically ( ) 方法反复对一个int类型变量进行自增操作，但没有使用“+”或者“++”操作符，它只使用了逻辑操作符&、|和~。方法如下：搜索当前int类型值的所有位，从最

低位开始，把1换成0；当在这个int类型值中遇到一个0时，这个0会被换成1，然后搜索结束，这样，自增操作就成功地完成了。所得到的int类型值表示为原来的int类型值自增1。然后这个过程又从这个新的int类型值开始。每一个被自增的数都被存在spock变量中。spock在编译过的字节码中，是一个位于位置0的局部变量，因此，就可以看到位置为0的局部变量值为0，1，2，3，等等。

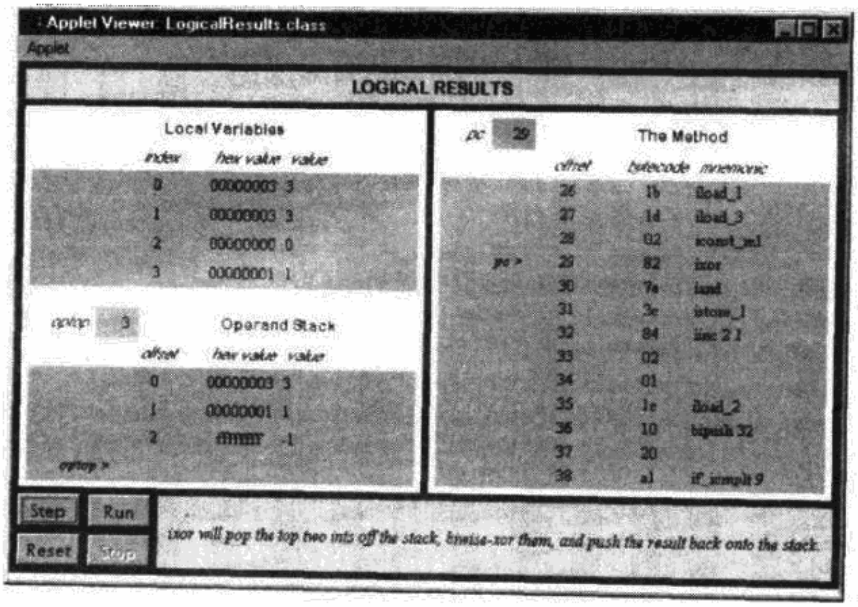


图13-1 “Logical Results” applet

使用“Step”、“Reset”、“Run”和“Stop”按钮即可启动“Logical Results”模拟。当点击“Step”按钮时，模拟就会执行PC寄存器指向的指令。当点击“Run”按钮时，模拟会继续进行下一步模拟运算，直到点击“Stop”按钮才会停止。点击“Reset”按钮，会重新开始模拟运算。对于模拟中的每一步，applet底部的面板都将显示出对下一步指令所做工作的解释。

13.3 随书光盘

光盘上opcodes子目录中有本章源代码示例。光盘上的applets/LogicalResults.html网页文件里包含了该“Logical Result” applet。这个applet的源代码和它的class文件一起放在applets/JVMSimulators和applets/JVMSimulators/COM/artima/jvmsim目录。

13.4 资源页

如果需要了解更多与本章相关的信息，请访问资源页面http://www.artima.com/insidejvm/resources。

第14章 浮点运算

本章介绍了浮点数和Java虚拟机中执行浮点运算的指令。本章描述的浮点数符合IEEE 754浮点数标准（所有Java虚拟机实现都必须遵循的标准）。

随书光盘上有两个使用交互方式来阐述本章内容的applet：名为“Inner Float”的applet能够对组成浮点数的各个部分进行观察和操作；名为“Circle of Squares”的applet对Java虚拟机进行模拟，并执行一个使用一些浮点操作码的方法，本章结尾处介绍了这个applet和它所执行的字节码。

14.1 浮点数

Java虚拟机的浮点支持符合IEEE-754 1985浮点标准，该标准定义了32位和64位浮点数的格式以及这些浮点数的运算。在Java虚拟机中，浮点运算基于32位float类型和64位double类型进行。每一个执行float类型运算的操作码，都会有一个与之对应的操作码，该操作码在double类型上实现同样功能。

浮点数由符号、尾数、基数和指数四部分组成。符号位的值要么是1，要么是-1。尾数永远是一个正数，它确定浮点数的有效位数。指数指与尾数、符号相乘的基数的幂的值，幂值可以为正，也可以为负。如下面公式所示：符号位与尾数相乘，然后再乘以基数的指数次幂，即得到所指的浮点数。

符号 × 尾数 × 基数的指数次幂

因为同一个浮点数可以表示为多个不同的尾数、基数和指数的组合，所以浮点数可以有多种表示形式。例如，数字-5可以被表示为表14-1中所列的几种形式，这几种形式都是以10为基数。

表14-1 -5的表示形式

符号	尾数	基数^指数
-1	50	10^{-1}
-1	5	10^{0}
-1	0.5	10^{1}
-1	0.05	10^{2}

对于每一个浮点数来说，都会有一种被称为“规范化”的表示形式。如果一个浮点数的尾数满足下面所列的关系式，则称这个浮点数为规范化的浮点数。

1/基数 <= 尾数 < 1

在以10为基数的浮点数中，尾数的小数点位置在第一个不为0的数字的左边。因此，-5的规范化浮点数表示为：$-1 \times 0.5 \times 10^{1}$。换句话说，一个规范化浮点数的尾数，它的小数点左边的数字一定为0，紧接小数点右边的数字一定不为0。不符合这条规则的浮点数被称为非规范化的浮点数。需要注意的是，因为0在小数点右边没有不为0的数字，因此数字0没有规范化的表示。

“为什么要规范化”是在处理数字0的时候经常发出的感叹。

Java虚拟机中的浮点数使用2为基数，因此，它们可以表示为如下形式：

符号 × 尾数 × 2的指数次幂

Java虚拟机中浮点数的尾数使用二进制来表示。规范化尾数的二进制小数点（基于二进制的，与十进制中作用相同）在最高有效非0数字的左边。因为二进制系统只有两个数字——0和1，所以对于规范化尾数来说，最高有效的数字就是1。

对于float类型或者double类型来说，符号位是最高有效位。尾数在float类型中占最低有效位的23位，在double类型中占最低有效位的52位。指数位于符号和尾数之间，它在float类型中占8位，在double类型中占11位。float类型的格式如下所示，符号位表示为s，指数位表示为e，尾数位表示为m：

```
s eeeeeeee mmmmmmmmmmmmmmmmmmmmmmm
```

符号位为0，表示正数；符号位为1，表示负数。尾数通常表示为二进制正数。它并不是二进制补码数。如果符号位为1，浮点数的值为负，但是尾数仍然表示为正数，但如果要表示浮点数的值，就需要乘上-1。

指数位的解释有三种方式。指数位全为1，表示该数为乘法或者减法所产生的特殊值之一——无穷大或者非数字（NaN），NaN是某种特殊操作，诸如0除以0的结果。指数位均为0，表示该数是一个非规范化浮点数。其他类型的指数位均表示该数为一个规范化的浮点数。

尾数区域包含一位附加精度位，它不同于尾数中的其他位的作用。float类型的尾数只占据23位，但它有24位精度。double类型的尾数占据52位，但它有53位精度。由于Java虚拟机中的浮点数的指数指明该数是否是规范化的，尾数中的最高有效位是可以预知的，因此没有被划入位数的范围。如果指数位全为0，那么该数为非规范化的浮点数，可知尾数的最高有效位为0。否则，浮点数是规范化的，那么尾数的最高有效位为1。

Java虚拟机在任何浮点操作中均不抛出异常。特殊值（例如正无穷大、负无穷大或者NaN）作为可疑操作（比如除0）的结果返回。指数位全为1，表示特殊的浮点值。指数位全为1而且尾数位全为0，表示无穷大。无穷大的符号由符号位表示出来。指数位全为1，尾数位不全为0，表示该数为“NaN”。Java虚拟机总是为NaN产生同样的尾数：除了尾数中最高有效位为1外，其余各位均为0。float类型的这些特殊值如表14-2所示。

表14-2 float类型的特殊值

值	浮点位（符号 指数 尾数）
+Infinity	0 11111111 00000000000000000000000
-Infinity	1 11111111 00000000000000000000000
NaN	1 11111111 10000000000000000000000

如果指数位既不全为0，又不全为1，该数即为规范化的浮点数。可以通过把指数位看作是一个正数，然后从这个正数中减去一个偏移量的方法来确定2的幂指数。对于float类型，偏移量为127。对于double类型，偏移量为1023。例如，一个浮点数的指数区域为00000001，那么它的幂指数可以通过如下步骤得到：把指数区域看作是一个正整数（1），然后减去偏移量（127），

然后就可以得到2的幂指数为1-127=-126。这个值是float类型的2的幂指数的最小值。另外，指数区域为11111110的2的幂指数为（254-127），即为127。127是float类型的最大的2的幂指数。表14-3举出了几个规范化浮点数的例子。

表14-3 float类型的规范化值

操作码	浮点位（符号 指数 尾数）	无偏差指数
最大的正（有限）float值	0 11111110 11111111111111111111111	127
最大的负（有限）float值	1 11111110 11111111111111111111111	127
最小的规范化浮点数	1 00000001 00000000000000000000000	-126
π	0 10000000 10010010000111111011011	-1

指数位全为0表示尾数非规范化，这意味着未指明的最重要位的值为0，而不是1。此时，2的幂指数与规范化尾数的2的幂指数最小值相等。对于float类型，该值为-126。与2的-126次幂相乘的规范化尾数，其指数区域为00000001；而与2的-126次幂相乘的非规范化尾数，其指数区域为00000000。

指数范围底端的非规范化浮点数允许渐进的下溢。如果这个最小的指数用来描述规范化数，那么一些较大的数将会下溢至0。换句话说，指定了非规范化数最小指数使得描述一些更小的数成为可能。这些更小的数与那些规范化数相比较，它们的精度较低，但是这样可以有效地解决指数一旦达到规范化最小值，就会下溢到0的问题。表14-4举出了几个非规范化浮点数值的例子。

表14-4 Float类型的非规范化值

值	浮点位（符号 指数 尾数）
最小的正浮点数（非0）	0 00000000 00000000000000000000001
最小的负浮点数（非0）	1 00000000 00000000000000000000001
最大非规范化浮点数	1 00000000 11111111111111111111111
正0	0 00000000 00000000000000000000000
负0	1 00000000 00000000000000000000000
正无穷	0 11111111 00000000000000000000000
负无穷	1 11111111 00000000000000000000000
NaN	1 11111111 10000000000000000000000

14.2 Inner Float：揭示Java float类型内部性质的applet

如图14-1所示的applet提供了了解浮点数格式的机会。浮点数的值可以使用多种格式显示。二进制科学计数法格式指出了尾数和以10为基的指数。在显示以前，当前的尾数被乘上2^{24}（得出一个整数），无偏移量的指数被减去24。整数形式的尾数和指数比较容易转换为十进制数，而且也比较容易显示。该applet嵌在随书光盘上的applets/InnerFloat.html网页文件里。网页文件内容包括几个按键序列，它们用来演示浮点数的各种属性。

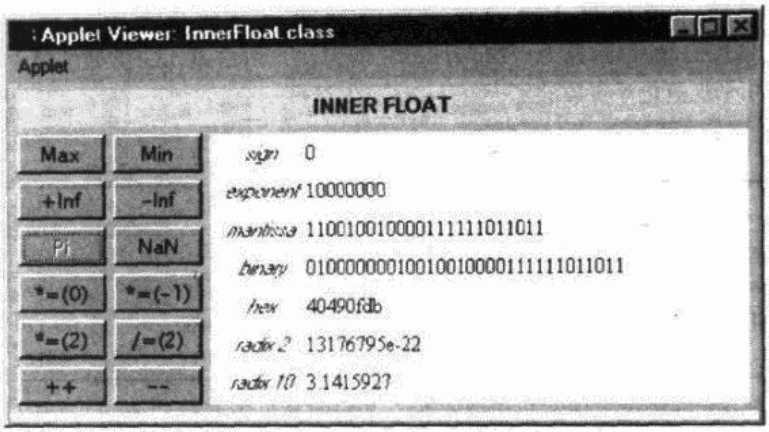


图14-1 "Inner Float" applet

14.3 浮点模式

Java虚拟机规范第2版略微放松了对于Java虚拟机中浮点操作的控制。最初的规范要求：所有单精度和双精度浮点运算都必须符合IEEE 754所规定的单精度和双精度格式。这个要求使得虚拟机无法最大限度地利用支持扩展精度格式及操作的硬件性能（特别是Intel x86系列处理器）。为了使虚拟机实现充分利用已有的硬件能力，Java虚拟机规范第2版在某些特定条件下允许在浮点运算中使用扩展精度格式。

新的浮点规则规定：每个方法都要有一个浮点模式，该方法的ACC_STRICT标志的值决定其浮点数模式（一个方法的ACC_STRICT标志是method_info表中access_flags项的一位，method_info表用来描述class文件中的方法）。如果设定了一个方法的ACC_STRICT标志，它的浮点模式就会是FP-strict。反之，它的浮点模式以Java官方术语而言就会是not FP-strict（或FP-default，这是一个非官方术语，在本章余下的部分，将使用这个术语）。一个模式为FP-strict的方法将会在最初对于浮点运算控制比较严格的规则下运行。一个模式为FP-default的方法将会在新的对于浮点运算控制比较不严格的规则下运行。如果设置了ACC_STRICT标志，在Java虚拟机第2版出台之前发布的编译器将无法产生class文件，因此，那些较老的Java编译器所产生的class文件中，方法的模式默认值是FP-default。

除了在class文件的method_info表中的access_flags项增加了新标志（ACC_STRICT标志）之外，新的浮点规则还为Java语言引入了新的关键字——strictfp（原书中此处为fpstrict，可能原书有误。——编者注）。1.2版本之前的Java编译器不会使用ACC_STRICT标志。从1.2版本开始，当strictfp关键字对方法声明以及类或接口（该方法作为其成员）的声明进行修改时，Java编译器将设置该方法的ACC_STRICT标志。类或接口中所有声明了strictfp的方法（包括编译器产生的方法，诸如<init>和<clinit>）都将会被设定ACC_STRICT标志位（对于接口来说，这个方法只可能是<clinit>方法。抽象方法的ACC_STRICT标志将不会被设定）。另外，如果任何封装的方法、类或者接口声明了strictfp的话，嵌套类中的方法将会被设定ACC_STRICT标志。

14.3.1 浮点值集合

最初的Java虚拟机规范要求所有的虚拟机实现都支持两种浮点值的标准集：float类型值集和double类型值集。如前所述，float类型值集是所有能够被Java的float类型变量或表达式所表述的值的集合。同样，如前所述，double类型值集是所有能够被Java的double类型变量或表达式所表述的值的集合。Java虚拟机规范第2版延续了这个传统，要求所有的虚拟机实现都必须支持最初的float类型和double类型值集。第2版规范还允许虚拟机实现支持两种扩展的浮点值集，它们分别称为float-extended-exponent值集和double-extended-exponent值集。在某些情况下，第2版规范允许虚拟机实现使用这些扩展的浮点值集的值来描述float或者double类型表达式的值。

相比相应的标准值集而言，扩展浮点值集允许使用更大的指数来表示一个值，但符号位和尾数的长度仍然不变。例如，标准的float值有1位符号位，24位尾数位和8位指数位。而扩展指数float值有1位符号位，24位尾数位，这些都与相应的标准值相同，但是它有11位指数位。标准的double值有1位符号位，53位尾数位和11位指数位。同样，扩展指数double值有1位符号位，53位尾数位，但它有15位或者更长的指数位。

14.3.2 浮点值集的转换

新的浮点规则（由第2版Java虚拟机规范定义）允许虚拟机实现使用扩展值集来表述操作数栈上的float或double类型的特定表达式。特别是，如果表达式设定为FP-strict模式，当写回操作数栈时，虚拟机必须使用标准值集来表述结果。反之，虚拟机可以使用一个扩展值集来表述结果。例如，当写回操作数栈时，设定为FP-strict模式的float类型表达式必须使用float值集表述。但如果是设定为FP-default模式的float类型表达式，则在写回操作数栈时可以根据虚拟机实现的判断，使用float-extended-exponent值集进行表述。

无论虚拟机什么时候把一个值从扩展指数值集转换为相应的标准指数值集，它都必须选择与标准值集中与之最接近的元素。转换并不改变值的类型。一个float类型值仍然是一个float类型值，一个double类型值仍然是一个double类型值。从扩展指数值集向标准值集转换只不过是尝试使用更小的指数来表述同样的浮点值。由于标准值集形式比相应的扩展指数形式的值位数少，从扩展指数值集向标准值集转换可能会引起上溢或者下溢。如果发生上溢，结果值将会是与原值具有同样符号的无穷大。如果发生下溢，结果值可能是比原值精度更差的非规范化浮点数，也可能是与原值符号相同的0值。

14.3.3 相关规则的本质

由于略微放松了对浮点操作的控制，Java虚拟机规范第2版也不再严格地承诺“Java程序将在任何虚拟机上有同样的表现”。因为某些虚拟机实现可能使用扩展指数值集，另外一些虚拟机实现可能没有使用它们，不同虚拟机实现上同样的浮点运算结果很可能不同。因此，新的浮点规则在性能与平台无关性方面必须有一个折中。

但在实践中，由相关浮点规则所导致的Java平台无关性方面的妥协看上去很可能是没有意义的。FP-default模式在绝大多数场合都会得出和FP-strict模式同样的浮点结果。在FP-default模式中，只有指数被扩展了，尾数并没有被扩展。因此，运算时只有当在FP-default中会上溢或下溢，而在FP-strict中不会发生的时候，FP-default模式中浮点操作的结果才会与FP-strict模式中的结果

不同。在实践中，由于绝大多数浮点运算不会在上溢或者下溢附近进行，FP-default模式将在大多数情况下与FP-strict模式获得同样的结果。

14.4 浮点操作码

如表14-5所示，操作码把两个浮点值从栈顶弹出，相加，然后把结果压入栈。值的类型由操作码自身决定，结果类型总是与相加的值类型相同。这些操作码执行时不会导致异常的抛出。上溢会得到正无穷或者负无穷，下溢会得到正0或者负0。

表14-5 浮点加法

操作码	操作数	说明
fadd	（无）	将两个float类型值弹出栈，相加，再将float类型的结果压入栈
dadd	（无）	将两个double类型值弹出栈，相加，再将double类型的结果压入栈

如表14-6所示的操作码执行float类型和double类型的减法。每一个操作码都会从栈中弹出两个适当类型的值，下面的那个值减去栈顶端的值，并将结果压入栈。这些操作码执行时不会导致异常抛出。

表14-6 浮点减法

操作码	操作数	说明
fsub	（无）	将两个float类型值弹出栈，相减，再将float类型的结果压入栈
dsub	（无）	将两个double类型值弹出栈，相减，再将double类型的结果压入栈

如表14-7所示的操作码执行float类型和double类型的乘法。每一个操作码都会从栈中弹出两个相同类型的值，然后相乘，与相乘的值具有同样类型的结果被压入栈。没有异常抛出。

表14-7 浮点乘法

操作码	操作数	说明
fmul	（无）	将两个float类型值弹出栈，相乘，再将float类型的结果压入栈
dmul	（无）	将两个double类型值弹出栈，相乘，再将double类型的结果压入栈

如表14-8所示的操作码执行float类型和double类型的除法。每一个操作码都会从栈中弹出两个适当类型的值，用下面的值除以栈顶端的值（换句话说，首先压入栈的值是被除数或分子，其次被压入栈的值，即栈顶端的值为分母或除数）。除法的结果被压入栈。

表14-8 浮点除法

操作码	操作数	说明
fdiv	（无）	将两个float类型值弹出栈，相除，再将float类型的结果压入栈
ddiv	（无）	将两个double类型值弹出栈，相除，再将double类型的结果压入栈

任何浮点除法都不会导致异常抛出。无穷大值除以0的浮点除法得到一个正无穷大或者负无

穷大。0除以0的浮点除法得到NaN。关于无穷大、0、NaN和有限值之间的多种组合的除法结果如表14-10所示。

float和double类型的取余操作通过如表14-9所示的操作码进行。下述的操作码会从栈中弹出两个值，用下面的值除以栈顶端的值，除法的余数被压入栈。

表14-9 浮点取余

操 作 码	操 作 数	说 明
frem	（无）	将两个float类型值弹出栈，取余，再将float类型的结果压入栈
drem	（无）	将两个double类型值弹出栈，取余，再将double类型的结果压入栈

所有的浮点取余操作都不会导致异常抛出。任何值除以0的取余操作都会得出NaN的结果。关于无穷大、0、NaN和有限值之间的多种组合的取余结果如表14-10所示。

表14-10 各种浮点除法的结果

a	b	a/b	a%b
Finite	+–0.0	+–Infinity	NaN
Finite	+–Infinity	+–0.0	a
+–0.0	+–0.0	NaN	NaN
+–Infinity	Finite	+–Infinity	NaN
+–Infinity	+Infinity	NaN	NaN

由frem和drem所提供的取余操作与irem和lrem所提供的整数取余操作遵循同样的规则：

```
(a/b)*b + a%b == a
```

如果对浮点数进行操作，那么除法（a/b）必须被转换为int或者long类型，小数部分将会被去除，如下式所示：

```
((long)(a/b))*b + a%b == a
```

这里的取余操作没有遵循IEEE 754标准。如果需要使用遵循IEEE 754的取余操作，请使用java.lang.Math包中的IEEEremainder（）方法。

如表14-11所示的操作码对float类型和double类型进行取反运算。取反操作码把栈顶端值弹出栈，取反，再压入栈。

表14-11 浮点数取反

操 作 码	操 作 数	说 明
fneg	（无）	将一个float类型值弹出栈，取反，再将float类型的结果压入栈
dneg	（无）	将一个double类型值弹出栈，取反，再将double类型的结果压入栈

14.5 一个模拟："Circle of Squares"

如图14-2所示的"Circle of Squares" applet演示了Java虚拟机执行浮点运算的字节码序列。

这个applet是随书光盘上applets/CircleOfSquares.html网页文件的一部分。在该模拟中，由javac为类中的squareItForever（）方法产生的字节码序列如下所示：

```
// On CD-ROM in file opcodes/ex1/SquareCircle.java
class SquareCircle {

    static void squareItForever() {
        float f = 2;
        for (;;) {
            f *= f;
            f = 0 - f;
        ;
    ;
;
```

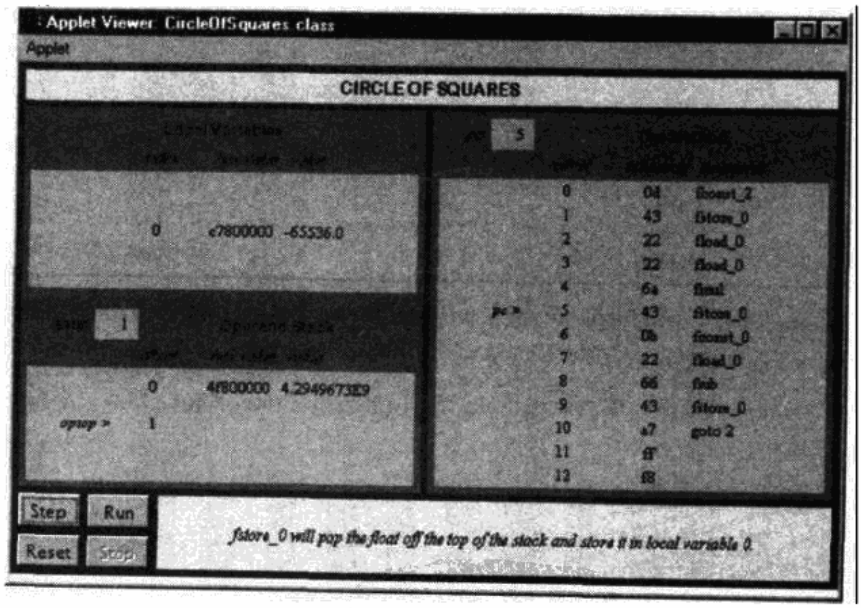


图14-2 “Circle of Squares” applet

由javac为squareItForever（）方法产生的字节码如下所示：

```
0 fconst_2          // Push float constant 2.
1 fstore_0          // Pop to local variable 0 (float f): float f = 2;
2 fload_0           // Push local variable 0 (float f).
3 fload_0           // Push local variable 0 (float f).
4 fmul              // Pop top two floats, multiply, push float result.
5 fstore_0          // Pop to local variable 0 (float f): f *= f;
6 fconst_0          // Push float constant 0.
7 fload_0           // Push local variable 0 (float f).
8 fsub              // Subtract top float from next to top float:
```

```
                    // imByte = (byte) imInt;
 9 fstore_0         // Pop result to local variable 0 (float f): f = 0 - f;
10 goto 2           // Jump back to the first fload_0 instruction:
                    // for (;;) {}
```

squareItForever ( ) 方法不断地对一个float值进行平方操作，直至无穷。每一次对这个float值进行平方操作后，还将对它进行取反操作。这个float值从2开始。在达到无穷之前，只需要反复7次，与现实生活中的情况并不相同。在applet中，“hex value”列显示的是组成这个浮点数的各位的十六进制表现形式，“value”列使用合乎一般习惯的方式来显示值。这个用户界面非常友好的值由Float.toString ( ) 方法产生。

使用“Step”、“Reset”、“Run”和“Stop”按钮即可启动“Circle of Squares”模拟。当点击“Step”按钮时，模拟就会执行PC寄存器指向的指令。当点击“Run”按钮时，模拟会继续进行下一步模拟，直到点击“Stop”按钮才会停止。点击“Reset”按钮，会重新开始模拟。对于每一步模拟，applet底部的面板都将显示出对下一步指令所做工作的解释。

14.6 随书光盘

随书光盘上opcodes子目录中有本章源代码示例。

“Inner Float”applet嵌在光盘上的applets/InnerFloat.html网页文件里。网页文件中包括几个按键序列，它们用来演示浮点数的多种属性。这个applet的源代码和它的class文件一起放在applets/InnerFloat目录。

光盘上的applets/CircleOfSquares.html网页文件里包含了“Circle of Squares”applet。这个applet的源代码和它的class文件一起放在applets/JVMSimulators和applets/JVMSimulators/COM/artima/jvmsim目录。

14.7 资源页

如果需要了解更多与本章相关的信息，请访问资源页面http://www.artima.com/insidejvm/resources。

第15章　对象和数组

本章介绍了Java虚拟机中创建、操作对象和数组的指令。这里所描述的所有指令都涉及到堆，大多数指令都涉及到常量池中的入口。

随书光盘上有一个使用交互方式来阐述本章内容的applet，名为“Three-Dimensional Array”。这个applet模拟了Java虚拟机执行一个分配和初始化三维数组的方法的过程，本章结尾处介绍了这个applet和它所执行的字节码。

15.1　关于对象和数组的回顾

如前所述，在Java虚拟机中，内存只能以对象形式在垃圾收集堆中分配。除非作为对象的一部分，否则不能为基本类型在堆中分配内存。如果需要在必须使用对象引用的场合使用基本类型，可以从java.lang包中为类型分配一个包装器对象。例如，Integer类把int类型以对象的形式封装起来。只有对象引用和基本类型可以在Java的栈中以局部变量形式存在。Java栈不能容纳对象。

Java虚拟机中的对象和基本类型的结构分离在Java编程语言中体现为：对象不能作为局部变量声明，只有对象引用和基本类型可以。对象引用在声明后并不指向任何有意义的东西，只有在引用被显式初始化后（无论是使引用指向一个已存在的对象还是新建一个对象）对象引用才会指向一个真实的对象。

在Java虚拟机指令集中，除了数组以外，所有的对象都使用同样的操作码来实例化和存取。如前所述，Java中的数组是完善（full-fledged）的对象。和Java中其他对象一样，数组是动态创建的，数组引用可以在任何需要用到引用来标识对象的地方使用，数组中对象的任何方法都可以被调用。但Java虚拟机中仍然使用特殊的字节码来处理数组。

如同其他的对象，数组不能作为局部变量来使用，只有数组引用才可以。数组对象本身通常包括基本类型数组或者对象引用数组。如果声明了对象数组，获得的将是对象引用的数组。对象本身必须通过new操作显式创建，并且赋给数组成员。

15.2　针对对象的操作码

实例化一个新对象需要通过new操作码来实现，如表15-1所示。new操作码后面紧随着两个字长的操作数，这两个字长的操作数合起来表示常量池中的一个不带符号的16位长度的索引。在特定偏移量位置处的常量池入口给出了新对象所属类的信息。如果还没有这些信息，那么虚拟机会解析这个常量池入口。它会为这个堆中的对象建立一个新的实例，用默认初始值初始化对象实例变量，然后把新对象的引用压入栈。

把对象的字段弹出、压入栈的操作码如表15-2所示。putfield和getfield这两个操作码只在字段是实例变量的情况下才执行。putstatic和getstatic对静态变量进行存取操作（这将在后面讨论）。putfield和getfield指令都有两个操作数，这两个操作数合起来表示常量池中的不带符号的16位长

度的索引。这个索引所指向的常量池入口包含了该字段的所属类、名字和类型等信息。如果还没有这些信息，虚拟机会解析这个常量池入口。putfield和getfield对对象引用进行栈操作。putfield指令从栈中取出实例变量值，getfield指令把获得的实例变量值压入栈。

表15-1 对象的创建

操作码	操作数	说明
new	indexbyte1，indexbyte2	在堆中创建一个新的对象，将其引用压入栈

表15-2 存取实例变量

操作码	操作数	说明
putfield	indexbyte1，indexbyte2	设置对象字段（由index指定）的值，值value和对象引用objectref均从栈中获得
getfield	indexbyte1，indexbyte2	将对象字段（由index指定）压入栈，对象引用objectref从栈中获得

如表15-3所示，类变量通过getstatic和putstatic操作码进行存取操作。getstatic和putstatic都有两个长度为一个字长的操作数，在Java虚拟机中，这两个操作数合起来表示常量池中的16位不带符号的偏移量。该位置处的常量池项给出了一个类的静态字段的相关信息。如果还没有这些信息，虚拟机会解析这个常量池入口。因为没有任何特定的对象与静态字段相关联，所以getstatic和putstatic不会使用对象引用。putstatic指令把需要使用的值从栈中取出，getstatic指令把获得的值压入栈。

表15-3 存取类变量

操作码	操作数	说明
putstatic	indexbyte1，indexbyte2	设置静态字段（由index指定）的值（值从栈中获得）
getstatic	indexbyte1，indexbyte2	将静态字段（由index指定）压入栈

下面所要介绍的操作码用来检查栈顶的对象引用是指向一个类的实例，还是指向以紧随操作码的操作数为索引的接口。在这两种情况下，虚拟机都会产生指向常量池入口的无符号16位长度索引，然后把它的值赋给操作码后的两个字节。如果还没有这些内容，虚拟机会解析常量池入口。

如果所给的对象不是指定类或者接口的实例，checkcast指令则会抛出一个CheckCastException异常。否则，任何事情都不会发生。对象引用仍在栈中，下一条指令会接着执行。这条指令确保运行时类型转换的安全，并且是Java虚拟机安全框架的组成部分。

如表15-4所示，instanceof指令从栈顶端弹出对象引用，然后压入1或者0。如果对象确实是指定的类或者接口的实例，就向栈中压入1；反之，就向栈中压入0。instanceof指令用来实现Java语言中的instanceof关键字，这个关键字用来测试一个对象是否为一个指定类或者接口的实例。

表15-4 类型检验

操作码	操作数	说明
checkcast	indexbyte1，indexbyte2	如果栈中的对象引用不能转换为位于index位置的类，则抛出ClassCastException异常
instanceof	indexbyte1，indexbyte2	如果栈中的对象引用是位于index位置的类的实例，则向栈中压入true (1)，否则压入false (0)

15.3 针对数组的操作码

如表15-6所示，实例化新数组的工作可以通过newarray、anewarray和multianewarray操作码来完成。newarray操作码用来创建基本类型的数组，而不是对象引用的数组。基本类型由紧随newarray操作码的单字节操作数“atype”指定。newarray指令能够创建byte、short、char、int、long、float、double或者boolean类型的数组。

表15-5列出了atype的有效值和相应的数组类型。

表15-5 atype的值

数组类型	atype
T_BOOLEAN	4
T_CHAR	5
T_FLOAT	6
T_DOUBLE	7
T_BYTE	8
T_SHORT	9
T_INT	10
T_LONG	11

表15-6 创建新数组

操作码	操作数	说明
newarray	atype	从栈中弹出数组长度，使用atype所指定的基本数据类型分配新数组，将新数组的对象引用压入栈
anewarray	indexbyte1，indexbyte2	从栈中弹出数组长度，使用由indexbyte1和indexbyte2所指定的类分配新对象数组，将新数组的对象引用压入栈
multianewarray	indexbyte1，indexbyte2，dimensions	从栈中弹出数组的维数，使用由indexbyte1和indexbyte2所指定的类分配新多维数组，将新数组的对象引用压入栈

需要注意的是，当数组类型显式声明为boolean时，Java虚拟机中创建数组的指令会以位为单位进行操作。此功能使得实现，特别是需要控制内存需求的时候，可以采用位映像模式来压缩boolean数组。在这种表示方法中，一个数组的每个boolean元素都可以使用1位来表示。当内存不是很紧张的时候，boolean数组可以使用byte数组的方式实现，尽管这样会消耗掉更多的内存。无论虚拟机对于boolean数组使用哪一种内部实现，都会使用存取byte数组元素的操作码访

问boolean数组的元素。这些操作码将在本章后面讨论。

anewarray指令用来建立一个对象引用的数组。anewarray指令有两个单字节长的操作数，它们紧随anewarray操作码之后，这两个操作数合起来表示常量池中的一个不带符号的16位长度的索引。创建的数组所针对的对象，其所属类的描述可以通过这个索引在常量池中找到。如果还没有相关描述，虚拟机将会解析常量池入口。这条指令为对象引用数组分配空间，并把引用值初始化为null。

multianewarray指令用来分配多维数组，所谓多维数组，也就是数组的数组。而多维数组也可以通过重复使用anewarray和newarray来进行分配。multianewarray指令只不过把创建多维数组所需要的字节码压缩到一条指令中。multianewarray指令有两个紧随multianewarray操作码之后的单字节长的操作数，这两个操作数合起来表示常量池中的不带符号的16位长度的索引。创建的数组所针对的对象，其所属类的描述可以通过这个索引在常量池中找到。如果还没有相关描述，虚拟机将会解析这个符号引用。紧随这两个操作数之后，是一个无符号的单字节操作数，这个操作数用来表示多维数组的维数。每一维的长度都会从栈中弹出。这条指令为所有组成多维数组的数组分配空间。

multianewarray指令使用的常量池入口包含带有数组类名的CONSTANT_Class的项。例如，一个四维float类型数组的常量池入口将会有一个形如“[[[[F”的名字。常量池入口中的类名可以拥有比维数字节指定的更多（但不会更少）的左括号。虚拟机通常根据维数字节创建数组维数。

arraylength指令如表15-7所示。arraylength从栈顶端弹出一个数组引用，然后把这个数组的长度压入栈。

表15-7 获取数组长度

操作码	操作数	说明
arraylength	（无）	从栈中弹出一个数组的对象引用，将数组长度压入栈

如表15-8所示的操作码从数组中获取一个元素。虚拟机从栈中弹出数组的索引和数组引用，再将位于给定数组的指定索引位置的值压入栈。baload操作码把byte或者boolean类型的值通过符号扩展转换为int类型，然后把这个int类型的值压入栈。同样，saload操作码把short类型的值通过符号扩展转换为int类型，然后再把这个int类型的值压入栈。caload把char类型的值通过零扩展转换为int类型的值，然后再把这个int类型的值压入栈。

表15-8 获取数组元素

操作码	操作数	说明
baload	（无）	将byte类型或者boolean类型的数组的索引index和数组引用arrayref弹出栈，将arrayref[index]压入栈
caload	（无）	将char类型的数组的索引index和数组引用arrayref弹出栈，将arrayref[index]压入栈
saload	（无）	将short类型的数组的索引index和数组引用arrayref弹出栈，将arrayref[index]压入栈
iaload	（无）	将int类型的数组的索引index和数组引用arrayref弹出栈，将arrayref[index]压入栈
laload	（无）	将long类型的索引index和数组引用arrayref弹出栈，将arrayref[index]压入栈
faload	（无）	将float类型的数组的索引index和数组引用arrayref弹出栈，将arrayref[index]压入栈

（续）

操作码	操作数	说明
daload	（无）	将double类型的数组的索引index和数组引用arrayref弹出栈，将arrayref[index]压入栈
aaload	（无）	将对象引用类型的数组的索引index和数组引用arrayref弹出栈，将arrayref[index]压入栈

表15-9列出了将一个值保存到数组元素的操作码。值、索引和数组引用都从栈顶端弹出。bastore指令只存储所弹出int类型值的低8位。sastore和castore指令只存储所弹出int类型值的低16位。

表15-9 设定数组元素的值

操作码	操作数	说明
bastore	（无）	将byte类型或者boolean类型的数组的值value、索引index和数组引用arrayref弹出栈，赋值为arrayref[index]=value
castore	（无）	将char类型的数组的值value、索引index和数组引用arrayref弹出栈，赋值为arrayref[index] =value
sastore	（无）	将short类型的数组的值value、索引index和数组引用arrayref弹出栈，赋值为arrayref[index] =value
iastore	（无）	将int类型的数组的值value、索引index和数组引用arrayref弹出栈，赋值为arrayref[index] = value
lastore	（无）	将long类型的数组的值value、索引index和数组引用arrayref弹出栈，赋值为arrayref[index]=value
fastore	（无）	将float类型的数组的值value、索引index和数组引用arrayref弹出栈，赋值为arrayref[index] =value
dastore	（无）	将double类型的数组的值value、索引index和数组引用arrayref弹出栈，赋值为arrayref[index] =value
aastore	（无）	将对象引用类型的数组的值value、索引index和数组引用arrayref弹出栈，赋值为arrayref[index] =value

15.4 一个模拟："Three-Dimensional Array"

如图15-1所示的"Three-Dimensional Array" applet演示了Java虚拟机执行一段字节码序列的过程。该applet嵌在随书光盘的applets/ThreeDarray.html网页文件里。

在这个模拟中，由javac为下列类的initAnArray（）方法产生的字节码序列如下所示：

```
// On CD-ROM in file opcodes/ex1/ThreeDTree.java
class ThreeDTree {

    static void initAnArray() {

        int[][][] threeD = new int[5][4][3];

        for (int i = 0; i < 5; ++i) {
            for (int j = 0; j < 4; ++j) {
                for (int k = 0; k < 3; ++k) {
```

```
                    threeD[i][j][k] = i + j + k;
                }
            }
        }
    }
}
```

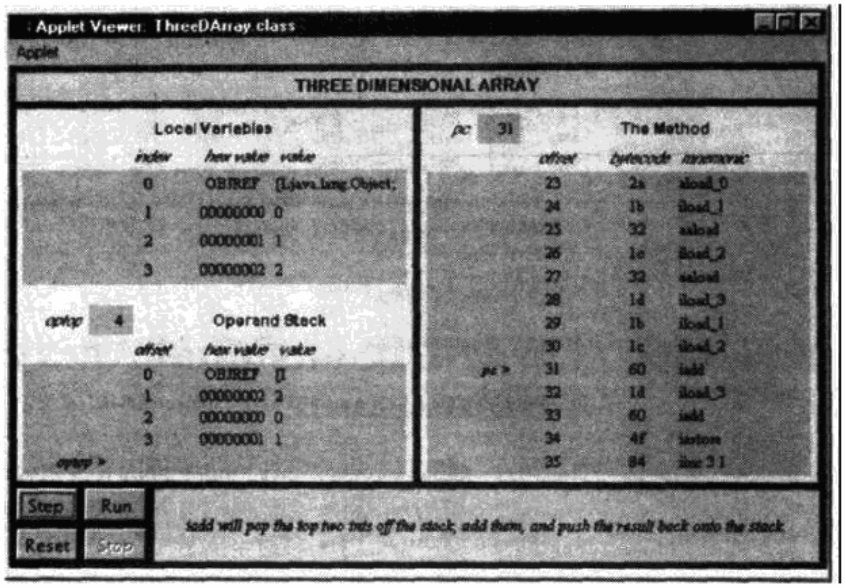


图15-1 “Three-Dimensional Array”applet

由javac为initAnArray（）方法产生的字节码如下所示：

```
 0 iconst_5        // Push constant int 5.
 1 iconst_4        // Push constant int 4.
 2 iconst_3        // Push constant int 3.
                   // Create a new multi-dimensional array
                   // using constant pool entry #2 as the
                   // class (which is [[[I, an 3D array of
                   // ints) with a dimension of 3.
 3 multianewarray #2 dim #3 <Class [[[I>
                   // Pop object ref into local variable 0:
 7 astore_0        // int threeD[][][] = new int[5][4][3];
 8 iconst_0        // Push constant int 0.
                   // Pop int into local variable 1:
 9 istore_1        // int i = 0;
                   // Go to section of code that tests
10 goto 54         // outer loop.
13 iconst_0        // Push constant int 0.
                   // Pop int into local variable 2:
```

```
14 istore_2       // int j = 0;
                  // Go to section of code that tests
15 goto 46        // middle loop.
18 iconst_0       // Push constant int 0.
                  // Pop int into local variable 3:
19 istore_3       // int k = 0;
                  // Go to section of code that tests
20 goto 38        // inner loop.
23 aload_0        // Push object ref from local variable 0.
24 iload_1        // Push int from local variable 1 (i).
                  // Pop index and arrayref, push object
                  // ref at arrayref[index] (gets
25 aaload         // threeD[i]).
26 iload_2        // Push int from local variable 2 (j).
                  // Pop index and arrayref, push object
                  // ref at arrayref[index] (gets
27 aaload         // threeD[i][j]).
28 iload_3        // Push int from local variable 3 (k).
                  // Now calculate the int that will be
                  // assigned to threeD[i][j][k]
29 iload_1        // Push int from local variable 1 (i).
30 iload_2        // Push int from local variable 2 (j).
                  // Pop two ints, add them, push int
31 iadd           // result (i + j).
32 iload_3        // Push int from local variable 3 (k).
                  // Pop two ints, add them, push int
33 iadd           // result (i + j + k).
                  // Pop value, index, and arrayref; assign
                  // arrayref[index] = value:
34 iastore        // threeD[i][j][k] = i + j + k;
                  // Increment by 1 the int in local
35 iinc 3 1       // variable 3: ++k;
38 iload_3        // Push int from local variable 3 (k).
39 iconst_3       // Push constant int 3.
                  // Pop right and left ints, jump if
40 if_icmplt 23   // left < right: for (...; k < 3;...)
                  // Increment by 1 the int in local
43 iinc 2 1       // variable 2: ++j;
46 iload_2        // Push int from local variable 2 (j).
47 iconst_4       // Push constant int 4.
                  // Pop right and left ints, jump if
48 if_icmplt 18   // left < right: for (...; j < 4;...)
                  // Increment by 1 the int in local
51 iinc 1 1       // variable 1: ++i;
54 iload_1        // Push int from local variable 1 (i).
55 iconst_5       // Push constant int 5.
```

```
56 if_icmplt 13   // Pop right and left ints, jump if
                  // left < right: for (...; i < 5;...)
59 return
```

initAnArray（）方法只对一个三维数组进行分配和初始化操作。该模拟演示了Java虚拟机处理三维数组的过程。为了对请求分配三维数组的指令multianewarray做出响应，Java虚拟机创建了一个树状的一维数组。由multianewarray指令返回的引用指向基础一维数组在initAnArray（）方法中，基础数组有5个元素：threeD[0]到threeD[4]。基础数组的每个元素自身又是另外一个包含4个元素的一维数组的引用，这些二级数组可以通过threeD[0][0]到threeD[4][3]来访问。而这20个数组元素也都是数组引用，每一个都包含3个元素。这些元素都是int类型，同时，它们也都是这个三维数组的元素，它们都可以通过threeD[0][0][0]到threeD[4][3][2]来访问。

为了对initAnArray（）方法中的multianewarray指令做出响应，Java虚拟机创建了一个拥有5个数组元素的数组，5个拥有4个数组元素的数组，20个拥有3个int类型元素的数组。Java虚拟机为这26个数组在堆中分配空间，初始化它们的元素（让所有的元素形成了一棵树），然后返回基础数组的引用。

为了给此三维数组的元素赋一个int类型的值，Java虚拟机使用aaload来获取基础数组的一个元素，然后Java虚拟机再对此元素（它本身是一个数组的数组）使用aaload指令来获取分支数组的一个子元素，这个子元素是一个指向第3级int类型数组的引用。最后，Java虚拟机使用iastore指令把一个int类型的值赋给这个第3级数组的元素。Java虚拟机通过对多个一维数组的访问来完成对多维数组的操作。

使用“Step”、“Reset”、“Run”和“Stop”按钮即可启动“Three-Dimensional Array”模拟。当点击“Step”按钮时，模拟就会执行PC寄存器指向的指令。当点击“Run”按钮时，模拟会继续进行下一步模拟，直到点击“Stop”按钮才会停止。点击“Reset”按钮，会重新开始模拟。对于每一步模拟，applet底部的面板都将显示出对下一步指令所做工作的解释。

15.5 随书光盘

随书光盘上opcodes子目录中有本章源代码示例。光盘上的applets/ThreeDArray.html网页文件里包含了该“Three-Dimensional Array”applet。这个applet的源代码和它的class文件一起放在applets/JVMSimulators和applets/JVMSimulators/COM/artima/jvmsim目录。

15.6 资源页

如果需要了解更多与本章相关的信息，请访问资源页面http://www.artima.com/insidejvm/resources。

第16章 控 制 流

本章介绍了控制Java虚拟机在同一个方法中进行条件或者无条件分支操作的指令。本章内容包含了Java源代码中if、if-else、while、do-while、for和switch语句的实现。

随书光盘上有一个使用交互方式阐述本章内容的applet，名为“Saying Tomato”。这个applet模拟了Java虚拟机执行一个方法的过程，其中包含完成表跳转（Java源代码中switch语句编译后的版本）的字节码，本章结尾处介绍了这个applet和它所执行的字节码。

16.1 条件分支

在Java源代码中，可以在一个方法中使用if、if-else、while、do-while、for和switch语句来指定基本的控制流。当把所有源代码转换成为字节码的时候，除了switch语句外，Java编译器使用同样的操作码集。例如，Java提供的最简单的控制流是if语句。当编译一个Java程序时，针对if语句表达式的不同行为，if语句能够被转换成任意一组操作码。每一种操作码都会从栈顶端弹出一个或者两个值，然后进行比较。从栈中弹出一个值的操作码把该值与0进行比较；从栈中弹出两个值的操作码对这两个值进行比较。如果比较成功（成功在不同的操作码中定义不同），Java虚拟机将按照由比较操作码的操作数所提供的偏移量执行分支或者跳转操作。

对于所有的条件分支操作码，Java虚拟机都通过同样的过程来决定下一条将要执行的指令。虚拟机首先执行由操作码所决定的比较。如果比较失败，虚拟机将继续执行条件分支语句后面的代码；如果比较成功，虚拟机将会使用紧随操作码后的两个操作数字节来产生一个带符号的16位的偏移量。虚拟机给当前PC寄存器加上这个偏移量（条件分支操作码的地址）来获取目标地址。目标地址必须指向同一个方法中的一条指令的操作码。程序会继续从目标地址开始运行。

如表16-1所示的一组if操作码执行对0的整数比较操作。当Java虚拟机遇到上述这些操作码时，虚拟机将会从栈中弹出一个int类型值，并将其于0进行比较。

表16-1 条件分支：整数与0的比较

操作码	操作数	说明
ifeq	branchbyte1，branchbyte2	从栈中弹出int类型值，如果该值为0，则跳转到偏移量指定位置执行分支操作
ifne	branchbyte1，branchbyte2	从栈中弹出int类型值，如果该值不为0，则跳转到偏移量指定位置执行分支操作
iflt	branchbyte1，branchbyte2	从栈中弹出int类型值，如果该值小于0，则跳转到偏移量指定位置执行分支操作
ifle	branchbyte1，branchbyte2	从栈中弹出int类型值，如果该值小于等于0，则跳转到偏移量指定位置执行分支操作
ifgt	branchbyte1，branchbyte2	从栈中弹出int类型值，如果该值大于0，则跳转到偏移量指定位置执行分支操作
ifge	branchbyte1，branchbyte2	从栈中弹出int类型值，如果该值大于等于0，则跳转到偏移量指定位置执行分支操作

如表16-2所示的另一组if操作码从栈顶端弹出两个整数，将它们进行比较。如果比较成功，虚拟机就执行分支操作。在这些操作码执行前，值2是栈顶端的值，值1是紧跟在值2下面的值。

如表16-2所示的操作码对int类型进行操作。这些操作码用于short类型、byte类型和char类型的比较，Java虚拟机在处理比int类型小的数据类型时，通常首先将它们转换为int类型，然后再对int类型进行操作。

表16-2 条件分支：两个整数的比较

操作码	操作数	说明
if_icmpeq	branchbyte1，branchbyte2	从栈中弹出int类型值value1和value2，如果value1 == value2，则跳转到偏移量指定位置执行分支操作
if_icmpne	branchbyte1，branchbyte2	从栈中弹出int类型值value1和value2，如果value1 != value2，则跳转到偏移量指定位置执行分支操作
if_icmplt	branchbyte1，branchbyte2	从栈中弹出int类型值value1和value2，如果value1 < value2，则跳转到偏移量指定位置执行分支操作
if_icmple	branchbyte1，branchbyte2	从栈中弹出int类型值value1和value2，如果value1 <= value2，则跳转到偏移量指定位置执行分支操作
if_icmpgt	branchbyte1，branchbyte2	从栈中弹出int类型值value1和value2，如果value1 > value2，则跳转到偏移量指定位置执行分支操作
if_icmpge	branchbyte1，branchbyte2	从栈中弹出int类型值value1和value2，如果value1 >= value2，则跳转到偏移量指定位置执行分支操作

如表16-3所示的第三组操作码对其他基本类型（long、float、double）进行比较操作。这些操作码本身并不会执行分支操作，而是把代表比较结果的int类型值（0表示相等，1表示大于，-1表示小于）压入栈，然后使用一种前面已经介绍过的对int类型进行比较的操作码进行实际分支跳转。

表16-3 long类型、float类型和double类型的比较

操作码	操作数	说明
lcmp	（无）	从栈中弹出long类型值value1和value2，进行比较，将所得int类型结果压入栈
fcmpg	（无）	从栈中弹出float类型值value1和value2，进行比较，将所得int类型结果压入栈
fcmpl	（无）	从栈中弹出float类型值value1和value2，进行比较，将所得int类型结果压入栈
dcmpg	（无）	从栈中弹出double类型值value1和value2，进行比较，将所得int类型结果压入栈
dcmpl	（无）	从栈中弹出double类型值value1和value2，进行比较，将所得int类型结果压入栈

用于比较float类型的两个操作码（fcmpg和fcmpl），其不同之处在于处理NaN的方式。在Java虚拟机中，如果进行比较的值之一是NaN，浮点值比较通常会失败。当两个值都不是NaN时，如果两个值相等，fcmpg和fcmpl指令都会将0压入栈，如果第一个值大于第二个值，那么都会将1压入栈，如果第二个值大于第一个值，那么都会将-1压入栈。但当至少有一个值为NaN时，fcmpg指令会将1压入栈，而fcmpl值领会将-1压入栈。由于这两个操作码都可以使用，在两个浮点数之间进行的任何比较操作都会得到相同的结果，其结果也将被压入栈，这与是否因为NaN的出现而导致失败没有关系。对于比较两个double类型值的操作码——dcmpg和dcmpl，上述结

论依然成立。

如表16-4所示的第4组操作码从栈顶端弹出对象引用，将其与null进行比较。如果比较成功，Java虚拟机将会执行分支操作。

表16-4 条件分支：对象引用与null的比较

操 作 码	操 作 数	说 明
ifnull	branchbyte1，branchbyte2	从栈中弹出引用值value，如果value == null，则跳转至偏移量指定位置执行分支操作
ifnonnull	branchbyte1，branchbyte2	从栈中弹出引用值value，如果value ！= null，则跳转至偏移量指定位置执行分支操作

如表16-5所示的最后一组操作码从栈中弹出两个对象引用，对它们进行比较。在这种情况下，只存在两种比较结果："相等"或者"不相等"。如果引用相等，说明它们指向堆中同一对象；如果不相等，说明它们指向两个不同对象。与其他所有的if操作码一样，如果比较成功，Java虚拟机执行分支操作。

表16-5 条件分支：两个对象引用的比较

操 作 码	操 作 数	说 明
if_acmpeq	branchbyte1，branchbyte2	从栈中弹出引用值value1和value2，如果value1 == value2，则跳转至偏移量指定位置执行分支操作
if_acmpne	branchbyte1，branchbyte2	从栈中弹出引用值value1和value2，如果value1 != value2，则跳转至偏移量指定位置执行分支操作

16.2 无条件分支

上一节描述了所有使Java虚拟机执行条件分支操作的操作码。下面描述另一组使Java虚拟机进行无条件分支操作的操作码。如表16-6所示的这些操作码被称为goto指令。为了执行goto指令，虚拟机首先根据两个紧随goto指令的操作数字节，得出一个带符号的16位偏移量（为了执行goto_w指令，虚拟机需要首先根据紧随goto_w指令的4个操作数字节得出一个带符号的32位偏移量），接着，虚拟机再把所得到的偏移量加到当前PC寄存器上。最后得到的地址必须指向当前方法中一条指令的操作码的位置。虚拟机将会在这条指令处继续执行。

如表16-6所示的操作码足以表述字节码中任何控制流，这些控制流在Java源文件中以if、if-else、while、do-while或者for语句表示。前述的操作码也能够用来表述switch语句，但Java虚拟机的指令集为switch语句专门设计了两个操作码：tableswitch和lookupswitch。

表16-6 无条件分支

操 作 码	操 作 数	说 明
goto	branchbyte1，branchbyte2	跳转至偏移量指定位置执行分支
goto_w	branchbyte1，branchbyte2，branchbyte3，branchbyte4	跳转至偏移量指定位置执行分支

16.3 使用表的条件分支

如表16-7所示，tableswitch和lookupswitch指令都包含一个默认的分支偏移量和一组可变长度的“case值/分支偏移量”对。这两条指令都会将键值（紧随switch关键字后的括号中表达式的值）从栈中弹出。它们会把键值与所有case值进行比较。如果发现匹配项，则取与该case值相关的程序分支偏移量，若未发现匹配项，则取默认程序分支偏移量。

表16-7 表分支操作

操作码	操作数	说明
lookupswitch	<0-3 byte pad>defaultbyte1，defaultbyte2，defaultbyte3，defaultbyte4，npairs1，npairs2，npairs3，npairs4，case value/branch offset pairs…	弹出键值，与case值相匹配，如果匹配成功，则跳转至相关程序分支偏移量处，否则跳转至默认程序分支偏移量处
tableswitch	<0-3 byte pad>defaultbyte1，defaultbyte2，defaultbyte3，defaultbyte4，lowbyte1，lowbyte2，lowbyte3，lowbyte4，highbyte1，highbyte2，highbyte3，highbyte4，branch offsets…	弹出键值，如果不在低/高值范围内，则跳转至默认程序分支偏移量处，否则获取（键值-低值）程序分支偏移量，并跳转至该偏移量处

指令tableswitch与lookupswitch之间的不同之处在于，它们采用不同的方法指定case值。指令lookupswitch比tableswitch适用的范围更广，但tableswitch具有更高的效率。这两条指令后面都有0到3个填充字节，这是为了使紧随在填充字节后面的字节能以4字节的整数倍（从方法的开头算起）位置处开始（这两条指令是整个Java虚拟机指令集中仅有的考虑了边界对齐的多字节指令）。对于这两条指令来说，填充字节后面的4个字节容纳了默认的分支偏移量。

操作码lookupswitch之后是0到3个填充字节和4个字节的默认分支偏移量，接着就是一个4字节长的值（npairs），它指明了指令后附带的“case值/分支偏移量”对的数量。case值是一个int类型值，这充分说明了Java语言中的switch语句需要的是一个类型为int、short、char或者byte的键值表达式。如果将long、float或者double类型的值作为switch语句的键值，那么程序就无法通过编译。与每一个case值相关的程序分支偏移量都是一个4字节的偏移量。“值/分支偏移量”对必须按照case值递增的顺序依次出现。

在tableswitch指令中，紧随在0到3个填充字节和4个字节的默认分支偏移量后面的是低、高int类型值。低、高值指明了包含在本tableswitch指令中case值的范围。在低、高值后面的是程序分支偏移量列表。列表中项数为（高值 - 低值 + 1），列表内容为：高值程序分支偏移量、低值程序分支偏移量以及介于高值和低值之间每一个整数case值的程序分支偏移量。低值程序分支偏移量紧随高值程序分支偏移量之后。

因此，当Java虚拟机遇到一条lookupswitch指令时，它必须把键值与每个case值相比较，直到发生下列情况之一时才结束查找：发现相匹配的值；检索到case值大于键值（“case值/分支偏移量”对按照case值递增的顺序排列）；所有case值均检索完毕。如果虚拟机没有检索到匹配的值，它将使用默认的程序分支偏移量。而当Java虚拟机遇到tableswitch指令时，它会简单地检查键值是否位于高值和低值之间。如果不在此范围内，那么就使用默认的程序分支偏移量；如果在此范围之内，虚拟机将用键值减去低值，得出一个偏移量，该偏移量列在了分支偏移量列表

中。通过这样的方法，虚拟机能够确定适当的程序分支，而无需对每一个case值进行检查。

除了前面的表中所叙述的操作码之外，Java虚拟机中能够影响控制流的只有处理异常抛出与捕获、finally子句以及调用方法和从方法中返回的指令。这些操作码将在后续章节中讨论。

16.4 一个模拟："Saying Tomato"

如图16-1所示的"Saying Tomato" applet演示了Java虚拟机执行字节码序列的过程。该applet是随书光盘上applets/SayingTomato.html网页文件的一部分。在该模拟中，由javac为下列类中的argue（）方法产生的字节码序列如下所示：

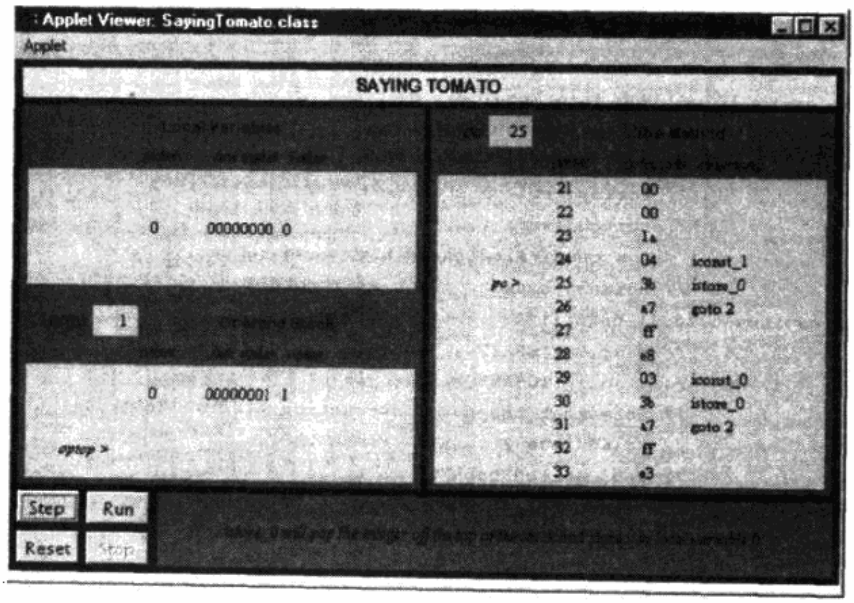


图16-1 "Saying Tomato" applet

```
// On CD-ROM in file opcodes/ex1/Struggle.java
class Struggle {

    public final static int TOMAYTO = 0;
    public final static int TOMAHTO = 1;

    static void argue() {

        int say = TOMAYTO;

        for (;;) {

            switch (say) {

            case TOMAYTO:

                say = TOMAHTO;
```

```
                break;

            case TOMAHTO:

                say = TOMAYTO;
                break;
            }
        }
    }
}
```

由javac为argue（）方法产生的字节码如下所示：

```
 0 iconst_0        // Push constant 0 (TOMAYTO)
                   // Pop into local var 0:
 1 istore_0        // int say = TOMAYTO;
 2 iload_0         // Push key for switch from local var 0
                   // Perform switch statement:
                   // switch (say) {...
                   // Low case value is 0, high case value 1
                   // Default branch offset will goto 2
 3 tableswitch 0 to 1: default=2
          0: 24    // case 0 (TOMAYTO): goto 24
          1: 29    // case 1 (TOMAHTO): goto 29
                   // Note that the next instruction starts
                   // at address 24, which means the
                   // tableswitch took up 21 bytes
24 iconst_1        // Push constant 1 (TOMAHTO)
25 istore_0        // Pop into local var 0: say = TOMAHTO
                   // Branch unconditionally to 2, top of
26 goto 2          // while loop
29 iconst_0        // Push constant 1 (TOMAYTO)
30 istore_0        // Pop into local var 0: say = TOMAYTO
                   // Branch unconditionally to 2, top of
31 goto 2          // while loop
```

方法argue（）的功能只不过是将say的值在TOMAYTO和TOMAHTO之间来回切换。由于TOMAYTO和TOMAHTO的值是连续的（TOMAYTO的值为0，TOMAHTO的值为1），javac编译器使用tableswitch指令。指令tableswitch比指令lookupswitch效率更高，而同等性质的lookupswitch指令的长度为28字节——比tableswitch长4个字节。

由此可知，当TOMAYTO的值为0，而TOMAHTO的值为2时，javac编译器仍将使用tableswitch。因为，尽管这里需要为默认值1额外分配程序分支偏移量，但tableswitch指令的长度仍然只有28字节——与功能相同的lookupswitch指令的长度相同，而tableswitch比lookupswitch效率更高，所以这里使用tableswitch。但当TOMAHTO的值为3时，javac开始使用lookupswitch指令，因为此时tableswitch的列表中将需要两个额外的分支偏移量（1和2），这样，该指令长度就将达

到32字节。指令lookupswitch的长度就会小于tableswitch，所以javac选择lookupswitch指令。

对应于case值的程序分支偏移量使Java虚拟机跳转到改变局部变量say值的代码分支处。say的值将会在TOMAYTO和TOMAHTO两者间不断变换，直到用户终止程序，从而终止所有操作。

使用“Step”、“Reset”、“Run”和“Stop”按钮即可启动“Saying Tomato”模拟。当点击“Step”按钮时，模拟就会执行PC寄存器指向的指令。当点击“Run”按钮时，模拟会继续进行下一步模拟，直到点击“Stop”按钮才会停止。点击“Reset”按钮，会重新开始模拟。对于每一步模拟，applet底部的面板都将显示出对下一步指令所做工作的解释。

16.5 随书光盘

光盘上opcodes子目录中有本章源代码示例。光盘上的applets/ SayingTomato.html网页文件里包含了该“Saying Tomato”applet。该applet的源代码和它的class文件一起放在applets/JVMSimulators和applets/JVMSimulators/COM/artima/jvmsim目录。

16.6 资源页

如果需要了解更多与本章相关的信息，请访问资源页面http://www.artima.com/insidejvm/resources。

第17章 异 常

本章阐述了字节码实现异常的过程。主要内容包括：显式抛出异常的指令、异常表，并且示范了catch子句的工作过程。

随书光盘上有一个使用交互方式来阐述本章内容的applet，名为“Play Ball!”。这个applet模拟了Java虚拟机执行抛出和捕获异常的过程，本章结尾处介绍了这个applet和它所执行的字节码。

17.1 异常的抛出与捕获

有了异常处理，就能够在程序运行时平稳地处理意外情况。为了描述Java虚拟机处理异常的方式，让我们看一个名为NitPickyMath的类。这个类提供了一些方法，这些方法完成整数的加、减、乘、除和取余操作。除了发生上溢、下溢和被0除时能够抛出指定的异常外，NitPickyMath与Java所提供的+、-、*、/和%运算符具有同样的功能。当发生整数被0除的情况时，Java虚拟机将会抛出ArithmeticException异常，但当上溢、下溢发生时，Java虚拟机不会抛出任何异常。被NitpickyMath的方法所抛出的异常定义如下：

```
// On CD-ROM in file except/ex1/OverflowException.java
class OverflowException extends Exception {
}

// On CD-ROM in file except/ex1/UnderflowException.java
class UnderflowException extends Exception {
}

// On CD-ROM in file except/ex1/DivideByZeroException.java
class DivideByZeroException extends Exception {
}
```

NitPickyMath类的remainder（）方法是一个简单的捕获和抛出异常的方法。

```
// On CD-ROM in file except/ex1/NitpickyMath.java
class NitpickyMath {

    static int add(int a, int b)
        throws OverflowException, UnderflowException {

        long longA = (long) a;
        long longB = (long) b;
        long result = a + b;
        if (result > Integer.MAX_VALUE) {
            throw new OverflowException();
```

```
        }
        if (result < Integer.MIN_VALUE) {
            throw new UnderflowException();
        }
        return (int) result;
    }

    static int subtract(int minuend, int subtrahend)
        throws OverflowException, UnderflowException {

        long longMinuend = (long) minuend;
        long longSubtrahend = (long) subtrahend;
        long result = longMinuend - longSubtrahend;
        if (result > Integer.MAX_VALUE) {
            throw new OverflowException();
        }
        if (result < Integer.MIN_VALUE) {
            throw new UnderflowException();
        }
        return (int) result;
    }

    static int multiply(int a, int b)
        throws OverflowException, UnderflowException {

        long longA = (long) a;
        long longB = (long) b;
        long result = a * b;
        if (result > Integer.MAX_VALUE) {
            throw new OverflowException();
        }
        if (result < Integer.MIN_VALUE) {
            throw new UnderflowException();
        }
        return (int) result;
    }

    static int divide(int dividend, int divisor)
        throws OverflowException, DivideByZeroException {

        // Overflow can occur in division when dividing
        // the negative integer of the largest possible
        // magnitude (Integer.MIN_VALUE) by -1, because
        // this would just flip the sign, but there is no
        // way to represent that number in an int.
        if ((dividend == Integer.MIN_VALUE) &&
```

```
            (divisor == -1)) {
            throw new OverflowException();
        }
        try {
            return dividend / divisor;
        }
        catch (ArithmeticException e) {
            throw new DivideByZeroException();
        }
    }

    static int remainder(int dividend, int divisor)
        throws OverflowException, DivideByZeroException {

        // Overflow can occur in division when dividing
        // the negative integer of the largest possible
        // magnitude (Integer.MIN_VALUE) by -1, because
        // this would just flip the sign, but there is no
        // way to represent that number in an int.
        if ((dividend == Integer.MIN_VALUE) &&
            (divisor == -1)) {
            throw new OverflowException();
        }
        try {
            return dividend % divisor;
        }
        catch (ArithmeticException e) {
            throw new DivideByZeroException();
        }
    }
}
```

remainder（）方法只是简单地在作为参数传入的两个int类型值之间执行取余操作。如果取余操作的除数为0，取余操作将会抛出ArithmeticException异常。remainder（）方法捕获该异常，并抛出DivideByZeroException异常。

两个异常之间的区别在于：DivideByZeroException异常是已被检验的，而ArithmeticException异常是未被检验的。由于ArithmeticException异常是未检验的，即使可能抛出此异常，方法中也可以不在throws子句中声明对此异常的处理。所有作为Error和RuntimeException的子类的异常都是未检验的（ArithmeticException异常是RuntimeException的子类）。通过捕获ArithmeticException异常和抛出DivideByZeroException异常，remainder（）方法强迫调用它的程序对可能发生的被0除异常进行处理，处理方式可以是捕获该异常，也可以是在它们自己的throws子句中声明DivideByZeroException。

javac为remainder（）方法产生下列字节码序列：

```
// The main bytecode sequence for remainder():
```

```
            // Push local variable 0 (arg passed as
 0 iload_0  // dividend)
            // Push the minimum integer value
 1 ldc #1 <Integer -2147483648>
            // If the dividend isn't equal to the minimum
            // integer, jump to the remainder calculation
 3 if_icmpne 19

            // Push local variable 1 (arg passed as
 6 iload_1  // divisor)
            // Push -1
 7 iconst_m1
            // If the divisor isn't equal to -1, jump
            // to the remainder calculation
 8 if_icmpne 19
            // This is an overflow case, so throw an
            // exception. Create a new OverflowException,
            // push reference to it onto the stack
11 new #4 <Class OverflowException>
14 dup      // Make a copy of the reference
            // Pop one copy of the reference and invoke
            // the <init> method of new OverflowException
            // object
15 invokespecial #10 <Method OverflowException()>
            // Pop the other reference to the
            // OverflowException and throw it
18 athrow

// Calculate the remainder

            // Push local variable 0 (arg passed as
19 iload_0  // dividend)
            // Push local variable 1 (arg passed as
20 iload_1  // divisor)
            // Pop divisor; pop dividend; calculate, push
21 irem     // remainder
22 ireturn  // Return int on top of stack (the remainder)


// The bytecode sequence for the
// catch (ArithmeticException) clause:

           // Pop the reference to the
           // ArithmeticException because it isn't used
23 pop     // by this catch clause.
```

```
24 new #2 <Class DivideByZeroException>
            // Create and push reference to new object of
            // class DivideByZeroException.
            // Duplicate the reference to the new object
            // on the top of the stack because it must be
            // both initialized and thrown. The
            // initialization will consume the copy of the
27 dup      // reference created by the dup.
            // Call the no-arg <init> method for the
            // DivideByZeroException to initialize it.
            // This instruction will pop the top reference
            // to the object.
28 invokespecial #9 <Method DivideByZeroException()>
            // Pop the reference to a Throwable object, in
            // this case the DivideByZeroException,
31 athrow   // and throw the exception.
```

方法remainder（）的字节码序列包含两个部分：第一部分是方法的正常执行路径，该部分指的是pc指针偏移量0到22；第二部分是catch子句，该部分指的是pc指针偏移量23到31。

17.2 异常表

主字节码序列中的指令irem可能会抛出ArithmeticException异常。如果发生这种情况，Java虚拟机将在表中查找异常，然后跳转到实现catch子句的字节码序列。每个捕获异常的方法都与异常表相关联，该异常表与方法的字节码序列一起送到class文件中。每一个被try语句块捕获的异常都与异常表中的一个入口（项）相对应。异常表中的每个入口都包括四部分信息：

- 起点。
- 终点。
- 将要跳转到的字节码序列中的pc指针偏移量。
- 被捕获的异常类的常量池索引。

下面是NitPickyMath类的remainder（）方法的异常表：

```
Exception table:
   from   to  target type
    19    23    23   <Class java.lang.ArithmeticException>
```

上述异常表说明，ArithmeticException异常在pc指针偏移量19到22（含）中被捕获。try语句块的终点值在“to”栏中列出。这个终点值总是比捕获异常位置的pc指针偏移量的最大值还要大1。在这个例子中，终点值为23，但捕获异常位置的pc偏移量最大值为22。偏移量19到22（含）之间的语句对应于实现remainder（）内try语句块代码的字节码序列。上述表格中列出的target栏指出了如果ArithmeticException异常在pc指针偏移量19到22（含）之间抛出，pc指针偏移量将要跳转到的位置。

如果异常在方法执行时抛出，Java虚拟机将会在整个异常表中搜寻相匹配的项。如果当前程序计数器在异常表入口所指定的范围内，而且所抛出的异常类是该入口所指向的类（或为指定

类的子类），那么该入口即为所搜寻的入口。Java虚拟机按照每个入口在表中出现的顺序进行检索。当遇到第一个匹配项时，虚拟机将程序计数器设为新的pc指针偏移量位置，然后从该位置继续执行。如果没有发现相匹配的项，虚拟机将当前栈帧从栈中弹出，再次抛出同样的异常。当Java虚拟机弹出当前栈帧时，虚拟机马上终止当前方法的执行，并且返回至调用本方法的方法中，但是并非继续正常执行该方法，而是在该方法中抛出同样的异常，这就使得虚拟机在该方法中再次执行同样的搜寻异常表的操作。

Java程序员可以使用throw语句抛出异常，正如remainder（）方法中的catch（Arithmetic Exception）子句，这里创建了异常DivideByZeroException，并将其抛出。执行抛出异常操作的字节码如表17-1所示。

表17-1 抛出异常

操作码	操作数	说明
athrow	（无）	弹出Throwable对象引用，抛出异常

指令athrow从栈顶端弹出字，并假设它是一个对象的引用，该对象为Throwable类的子类的实例（或者就是Throwable类的实例）。所抛出异常的类型由弹出的对象引用类型决定。

17.3 一个模拟："Play Ball!"

如图17-1所示的"Play Ball!" applet演示了Java虚拟机执行字节码序列的过程。该applet是随书光盘上applets/PlayBall.html网页文件的一部分。在该模拟中，由javac为下列类的playBall（）方法产生的字节码序列如下所示。

```
// On CD-ROM in file except/ex2/Ball.java
class Ball extends Exception {
}

// On CD-ROM in file except/ex2/Pitcher.java
class Pitcher {

    private static Ball ball = new Ball();

    static void playBall() {
        int i = 0;
        for (;;) {
            try {
                if (i % 4 == 3) {
                    throw ball;
                }
                ++i;
            }
            catch (Ball b) {
                i = 0;
```

```
            }
        }
    }
}
```

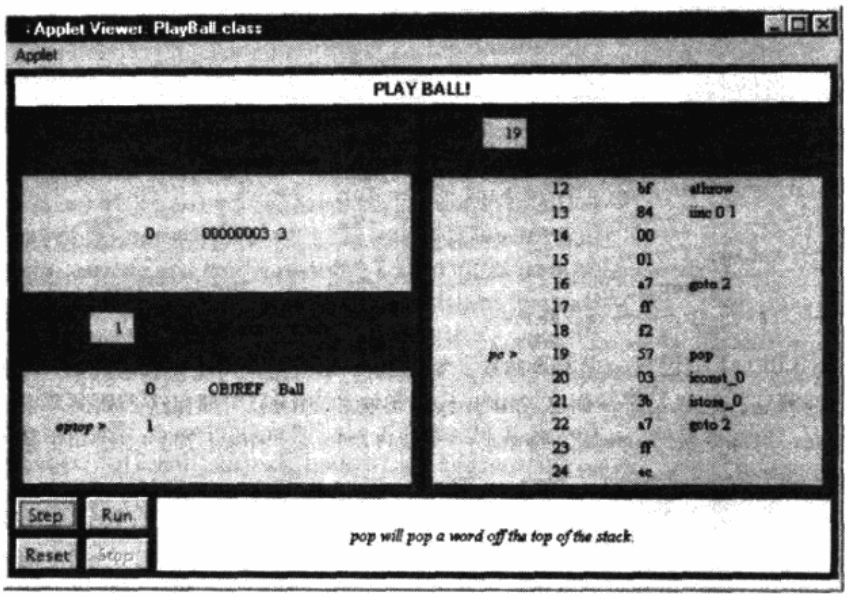


图17-1 “Play Ball!” applet

由javac为playBall（）方法产生的字节码如下所示：

```
// The main bytecode sequence for playBall():

 0 iconst_0       // Push constant 0
 1 istore_0       // Pop into local var 0: int i = 0;
                  // The try block starts here (see the
                  // exception table, below).
 2 iload_0        // Push local var 0
 3 iconst_4       // Push constant 4
 4 irem           // Calc remainder of top two operands
 5 iconst_3       // Push constant 3
                  // Jump if remainder not equal to 3:
 6 if_icmpne 13   // if (i % 4 == 3) {
                  // Push the static field at constant pool
                  // location #6, which is the Ball
                  // exception eager to be thrown
 9 getstatic #6 <Field Ball ball>
12 athrow         // Heave it home: throw ball;
                  // Increment the int at local var 0 by 1:
13 iinc 0 1       // ++i;
```

```
                    // The try block ends here (see the
                    // exception table, below).
16 goto 2           // jump always back to 2: for (;;) {}

// The bytecode sequence for the catch (Ball) clause:

                    // Pop the exception reference because it
19 pop              // is unused
20 iconst_0         // Push constant 0
21 istore_0         // Pop into local var 0: i = 0;
22 goto 2           // Jump always back to 2: for (;;) {}

Exception table:
   from   to  target type
     2    16    19   <Class Ball>
```

方法playBall（）中包含一个无穷循环。每循环4次，playBall（）就会抛出一个Ball类型的异常，然后捕获它，这只是游戏而已。由于try语句块和catch子句都在这个死循环中，这个游戏永远不会结束。局部变量i从0开始，每循环一次就自增1。每当i等于3时，if语句中条件为真，于是就会抛出Ball异常。

Java虚拟机检查异常表，然后发现可应用的入口确实存在。该入口的有效范围为2到15（含），而异常在pc指针偏移量12位置处抛出。被此入口所捕获的异常属于Ball类，所抛出的异常也属于Ball类。一旦匹配，虚拟机就将被抛出的异常对象压入栈，然后从pc指针偏移量19位置处继续执行。从偏移量19位置处开始的catch子句只是把int类型变量i置为0，然后重新开始循环。

使用“Step”、“Reset”、“Run”和“Stop”按钮即可启动“Play Ball!”模拟。当点击“Step”按钮时，模拟就会执行PC寄存器指向的指令。当点击“Run”按钮时，模拟会继续进行下一步模拟，直到点击“Stop”按钮才会停止。点击“Reset”按钮，会重新开始模拟。在模拟的每一步，applet底部的面板都将显示出对下一步指令所做工作的解释。

17.4 随书光盘

光盘上except子目录中有本章源代码示例。光盘上的applets/ PlayBall.html网页文件里包含了该“Play Ball!”applet，该applet的源代码和它的class文件一起放在applets/JVMSimulators和applets/JVMSimulators/COM/artima/jvmsim目录。

17.5 资源页

如果需要了解更多与本章相关的信息，请访问资源页面http://www.artima.com/insidejvm/resources。

第18章 finally子句

本章主要介绍字节码实现的finally子句，包括相关指令以及这些指令的使用方式。此外，本章还介绍了Java源代码中finally子句所展示的一些令人惊讶的特性，并从字节码角度对这些特性进行了解释。

随书光盘上有一个使用交互方式来阐述本章内容的applet，名为 “Hop Around”。这个applet模拟了Java虚拟机执行一个包含finally子句的方法的过程，本章结尾处介绍了这个applet和它所执行的字节码。

18.1 微型子例程

字节码中的finally子句在方法内部的表现很像“微型子例程”。Java虚拟机在每个try语句块和与其相关的catch子句的结尾处都会“调用”finally子句的子例程。finally子句结束后（这里的结束指的是finally子句中最后一条语句正常执行完毕，不包括抛出异常，或执行return、continue、break等情况），隶属于这个finally子句的微型子例程执行“返回”操作。程序在第一次调用微型子例程的地方继续执行后面的语句。

jsr指令是使Java虚拟机跳转到微型子例程的操作码。jsr指令使用一个双字节长度的操作数，这个操作数指出从jsr指令处到微型子例程开始处的16位带符号的偏移量。另外一条指令是jsr_w，它与jsr完成同样的功能，但是它支持更长的操作数（4个字节长）。当Java虚拟机遇到jsr或者jsr_w指令，它会把返回地址压入栈，然后从微型子例程的开始处继续执行。返回地址是紧接在jsr或jsr_w操作码和操作数后字节码的地址（偏移量或者本地指针）。该地址的类型为returnAddress。

微型子例程执行完毕后，将调用ret指令，ret指令的功能是执行从子例程中返回的操作。ret指令只有一个操作数，这个操作数是一个存储返回地址的局部变量的索引。表18-1中总结了处理finally子句的操作码。

不要混淆微型子例程与Java方法。Java方法与微型子例程使用不同的指令集。例如，调用Java方法可以使用invokevirtual和invokespecial等指令，使Java方法返回可以使用return，areturn，ireturn等指令。

表18-1 finally子句

操作码	操作数	说明
jsr	branchbyte1，branchbyte2	把返回地址压入栈，跳转至偏移量指定位置处执行分支操作
jsr_w	branchbyte1，branchbyte2，branchbyte3，branchbyte4	把返回地址压入栈，跳转至扩展宽度后的偏移量指定位置处执行分支操作
ret	index	返回存储在局部变量index中的地址
wide	ret，indexbyte1，indexbyte2	返回存储在局部变量index中的地址

jsr指令并不会调用Java方法，它只能跳转到相同方法中不同的操作码处。同样，ret指令也不能令Java方法返回，它只能使虚拟机跳回相同方法中调用jsr操作码和它的操作数之后的位置。本书中，实现finally子句的字节码被称为“微型子例程”，因为它们在一个方法的字节码流中的表现如同一个小子例程一样。

18.2 不对称的调用和返回

你也许会认为，ret指令应当从栈中弹出返回地址，因为返回地址也已被jsr指令压入栈。不是这样的，ret指令并不会这样做。在每一个子例程的开始处，返回地址都从栈顶端弹出，并且存储在局部变量中，稍后，ret指令将会从这个局部变量中取出返回地址。这种对返回地址的不对称的工作方式是必要的，因为finally子句（微型子例程）本身会抛出异常或者含有return、break、continue等语句。由于这些可能性的存在，这个被jsr指令压入栈的额外返回地址必须立即从栈中移除，因此，当finally子句通过break、continue、return或者抛出异常退出时，这个问题就不必再考虑了。

例如，下面的代码包含了一个通过break语句退出的finally子句。执行这段代码的结果是，无论给方法surpriseTheProgrammer（）的参数bVal传入什么，该方法都将返回false。

```
// On CD-ROM in file opcodes/ex3/Surprise.java
class Surprise {

    static boolean surpriseTheProgrammer(boolean bVal) {
        while (bVal) {
            try {
                return true;
            }
            finally {
                break;
            }
        }
        return false;
    }
}
```

上面的例子指出了返回地址必须在finally子句开始之处存入局部变量中的原因。因为finally子句在break语句处退出，它绝不会执行ret指令。因此，Java虚拟机不会执行返回true的语句。当它执行完break语句后，会退出至while语句的终结处，即闭括号处。下一条语句将会返回false，这个推断与Java虚拟机的执行过程完全相同。

无论使用break、return、continue，还是通过抛出异常退出finally子句，所显示出的特性都是相同的。在这些例子中，finally子句结尾处的ret指令永远不会执行到。正因为它永远不会被执行到，就无法指望它从栈中除去返回地址。因此，虚拟机在finally子句开始的地方就将返回地址存储到局部变量中。

下面所列出的方法是finally子句的一个完全范例，它包含了一个try语句块，这个try语句块

中有两个退出点。在这个例子里，两个退出点都是return语句。

```
// On CD-ROM in file opcodes/ex1/Nostalgia.java
class Nostalgia {

    static int giveMeThatOldFashionedBoolean(
        boolean bVal) {
        try {
            if (bVal) {
                return 1;
            }
            return 0;
        }
        finally {
            System.out.println("Got old fashioned.");
        }
    }
}
```

方法giveMeThatOldFashionedBoolean（）被编译为如下的字节码：

```
// The bytecode sequence for the try block:
 0 iload_0   // Push local variable 0 (bval parameter)
 1 ifeq 11   // Pop int, if equal to 0, jump to 11 (just
             // past the if statement): if (bval) {}
 4 iconst_1  // Push int 1
             // Pop an int (the 1), store into local
 5 istore_1  // variable 1
             // Jump to the mini-subroutine for the
 6 jsr 24    // finally clause
 9 iload_1   // Push local variable 1 (the 1)
             // Return int on top of the stack (the 1):
10 ireturn   // return 1;
11 iconst_0  // Push int 0
             // Pop an int (the 0), store into local
12 istore_1  // variable 1
             // Jump to the mini-subroutine for the
13 jsr 24    // finally clause
16 iload_1   // Push local variable 1 (the 0)
             // Return int on top of the stack (the 0):
17 ireturn   // return 0;

// The bytecode sequence for a catch clause that catches
// any kind of exception thrown from within the try block.
             // Pop the reference to the thrown exception,
18 astore_2  // store into local variable 2
             // Jump to the mini-subroutine for the
```

```
19 jsr 24     // finally clause
              // Push the reference (to the thrown
22 aload_2    // exception) from local variable 2
23 athrow     // Rethrow the same exception

// The miniature subroutine that implements the finally
// block.
              // Pop the return address, store it in local
24 astore_3   // variable 3
              // Get a reference to java.lang.System.out
25 getstatic #7 <Field java.io.PrintStream out>
              // Push reference to "Got old fashioned."
              // String from the constant pool
28 ldc #1 <String "Got old fashioned.">
              // Invoke System.out.println()
30 invokevirtual #8
    <Method void println(java.lang.String)>
              // Return to return address stored in local
33 ret 3      // variable 3
```

try语句块的字节码中包含了两条jsr指令，此外，catch子句里还有一条jsr指令。如果在执行try语句块过程中抛出了一个异常，最后的语句块必须接着执行。因此，编译器在字节码中加入了catch子句。所以这里的catch子句只调用了描述finally子句的微型子例程，然后再抛出一个同样的异常。如下所列的giveMeThatOldFashionedBoolean（）方法的异常表指出：凡是在地址0到17（含）之间（即所有实现try语句块的字节码中）抛出的异常，都被从地址18开始的catch子句处理。

```
Exception table:
   from    to   target type
     0     18     18   any
```

开始执行finally子句时，返回地址被弹出栈并被保存在局部变量3中。在finally子句结束的时候，ret指令再从局部变量3中取得返回地址。

18.3 一个模拟："Hop Around"

如图18-1所示的"Hop Around" applet演示了Java虚拟机执行一个字节码序列的过程。这个applet嵌在随书光盘上的applets/HopAround.html网页文件里。在这个模拟中，由javac为下列类的hopAround（）方法产生字节码序列：

```
// On CD-ROM in file opcodes/ex1/Clown.java
class Clown {

    static int hopAround() {
        int i = 0;
        while (true) {
```

```
            try {
                try {
                    i = 1;
                }
                finally {  // The first finally clause
                    i = 2;
                }
                i = 3;
                // This return never completes, because of
                // the continue in the second finally
                // clause
                return i;
            }
            finally {      // The second finally clause
                if (i == 3) {
                    // This continue overrides the return
                    // statement
                    continue;
                }
            }
        }
    }
}
```

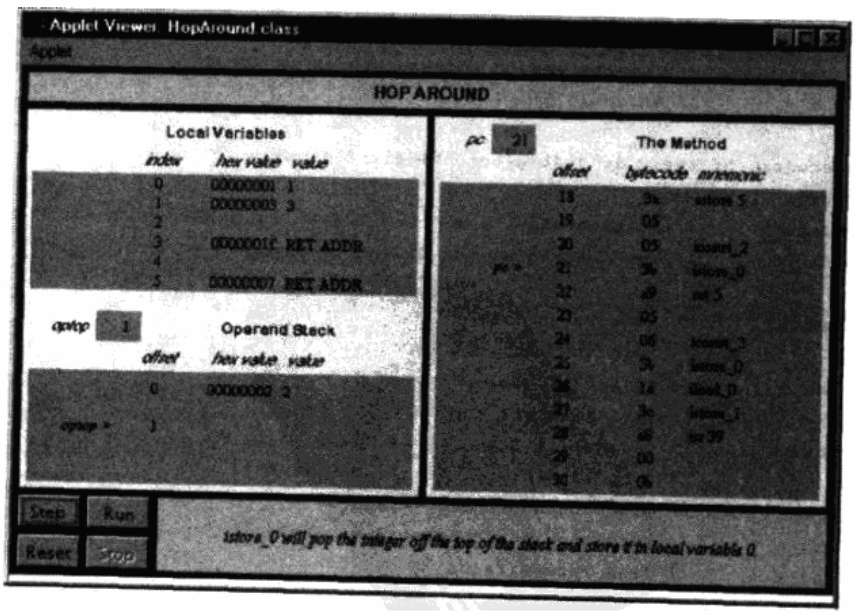


图18-1 “Hop Around” applet

由javac产生的hopAround（）方法的字节码如下所示：

```
 0 iconst_0      // Push constant 0
 1 istore_0      // Pop into local var 0: int i = 0;

// Both try blocks start here (see exception table, below):
 2 iconst_1      // Push constant 1
 3 istore_0      // Pop into local var 0: i = 1;
                 // Jump to mini-subroutine at offset 18
 4 jsr 18        // (the first finally clause)
                 // Jump to offset 24 (to just below first
 7 goto 24       // finally clause)

// Catch clause for the first finally clause:
                 // Pop the reference to thrown exception,
10 astore 4      // store in local variable 4
                 // Jump to mini-subroutine at offset 18
12 jsr 18        // (the first finally clause)
                 // Push the reference (to thrown
15 aload 4       // exception) from local variable 4
17 athrow        // Rethrow the same exception

// The first finally clause:
                 // Store the return address in local
18 astore 5      // variable 5
20 iconst_2      // Push constant 2
21 istore_0      // Pop into local var 0: i = 2;
                 // Jump to return address stored in local
22 ret 5         // variable 5

// Bytecodes for the code just after the first finally
// clause:
24 iconst_3      // Push constant 3
25 istore_0      // Pop into local var 0: int i = 3;

// Bytecodes for the return statement:
                 // Push the int from local
26 iload_0       // variable 0 (i, which is 3)
                 // Pop and store the int into local
27 istore_1      // variable 1 (the return value, i)
                 // Jump to mini-subroutine at offset 39
28 jsr 39        // (the second finally clause)
                 // Push the int from local variable 1
31 iload_1       // (the return value)
32 ireturn       // Return the int on the top of the stack

// Catch clause for the second finally clause:
                 // Pop the reference to thrown exception,
```

```
33 astore_2       // store in local variable 2
                  // Jump to mini-subroutine at offset 39
34 jsr 39         // (the second finally clause)
                  // Push the reference (to thrown
37 aload_2        // exception) from local variable 2
38 athrow         // Rethrow the same exception

// The second finally clause:
                  // Store the return address in local
39 astore_3       // variable 3
40 iload_0        // Push the int from local variable 0 (i)
41 iconst_3       // Push constant 3
                  // If the top two ints on the stack are
                  // equal, jump to offset 47:
42 if_icmpeq 47   // if (i == 3) {
                  // Jump to return address stored in local
45 ret 3          // variable 3
                  // Jump to offset 2 (the top of the while
47 goto 2         // block): continue;

Exception table:
   from   to  target type
     2     4    10   any
     2    31    31   any
```

hopAround（）方法在第一条finally子句中通过执行到闭括号返回，而它在第二条finally子句中是通过执行一条continue语句返回。第一条finally子句通过它的ret指令退出。而由于第二条finally子句使用continue退出，这样就不会执行它的ret指令。continue语句使Java虚拟机跳转至while循环的开始处。尽管方法中有return语句，但执行这条return语句前，将会首先执行第二条finally子句，于是得出结论：这个循环是一个死循环。在finally子句中的continue语句取代了return语句，这个方法永远无法返回。

需要注意的是，实现return语句的字节码在跳转到实现第二条finally子句的微型子例程的时，已经把返回值存入到局部变量1中。在微型子例程返回后（在上述情况下，它永远无法返回，因为总是在返回之前执行continue语句），再从局部变量1中取出返回值，并且返回。

这个过程强调了Java虚拟机在finally子句执行完毕前返回一个值的方式。尽管i的值是在执行完finally子句后返回的，但虚拟机仍将返回执行finally子句前i的值。所以就算finally子句会改变i的值，该方法仍将返回finally子句未执行前i的值。如果需要使用finally子句来改变方法返回值，就不得不在finally子句中再加入一条return语句，用来返回被finally子句更新过的返回值。

使用“Step”、“Reset”、“Run”和“Stop”按钮即可启动“Hop Around”模拟。当点击Step按钮时，模拟就会执行PC寄存器指向的指令。当点击“Run”按钮时，模拟会继续进行下一步模拟，直到点击“Stop”按钮才会停止。点击“Reset”按钮，会重新开始模拟。在模拟的每一步，applet底部的面板都将显示出对下一步指令所做工作的解释。

18.4 随书光盘

光盘上opcodes子目录中有本章源代码示例。光盘上的applets/HopAround.html网页文件里包含了该“Hop Around” applet，这个applet的源代码和它的class文件一起放在applets/JVMSimulators和applets/JVMSimulators/COM/artima/jvmsim目录。

18.5 资源页

如果需要了解更多与本章相关的信息，请访问资源页面http://www.artima.com/insidejvm/resources。

第19章　方法的调用与返回

Java虚拟机的指令集包括四种调用方法的指令，本章对这四种指令和这些指令执行的环境进行阐述。

19.1　方法调用

Java程序设计语言提供了两种基本的方法：实例方法和类（或者静态）方法。这两种方法的区别在于：

1) 实例方法在被调用之前，需要一个实例，而类方法不需要。

2) 实例方法使用动态（迟）绑定，而类方法使用静态（早）绑定。

当Java虚拟机调用一个类方法时，它会基于对象引用的类型（通常在编译时可知）来选择所调用的方法。相反，当虚拟机调用一个实例方法时，它会基于对象实际的类（只能在运行时得知）来选择所调用的方法。

Java虚拟机使用两种不同的指令分别调用这两种方法。对于实例方法，使用invokevirtual指令；对于类方法，使用invokestatic指令。这两种指令如表19-1所示。

表19-1 方法调用

操作码	操作数	说明
invokevirtual	indexbyte1，indexbyte2	把objectref（对象引用）和args（参数）从栈中弹出，调用常量池索引指向的实例方法
invokestatic	indexbyte1，indexbyte2	把args从栈中弹出，调用常量池索引指向的类方法

如前所述，到方法的引用最初是符号化的。所有的调用指令（例如invokevirtual和invokestatic）都指向一个最初包含符号引用的常量池入口。当Java虚拟机遇到一条调用指令时，如果还没有解析符号引用，那么虚拟机把解析符号引用作为执行指令调用执行过程中的一部分。

要解析一个符号引用，Java虚拟机要确定被符号化引用的方法，然后再用一个直接引用来代替符号引用。直接引用就如同偏移量指针一样，如果将来再次使用该引用，它可以使虚拟机更快地调用这个方法。

例如，在执行invokevirtual指令之前，Java虚拟机使用紧随invokevirtual操作码之后的indexbyte1和indexbyte2这两个操作数，产生一个指向当前类常量池的无符号16位索引。这个常量池入口包含一个指向将要调用的方法的符号引用。

在解析过程中，Java虚拟机还将执行几次确认检验，这些检验能够确保遵循了Java语言规范，能够确保调用invoke指令的安全。比方说，虚拟机首先确认被符号化引用的方法是否存在，如果存在，虚拟机再检验当前类是否能够合法地访问这个方法。比方说，如果这个方法为私有方法，那么它必须为当前类的成员。上述任何一个检验失败，Java虚拟机都会抛出异常。第5章对这个

解析过程进行了概述。第8章对这个过程进行了详细描述。

一旦解析了一个方法后，Java虚拟机就准备调用它。如果这个方法是一个实例方法，它必须在一个对象中被调用。对每一次实例方法的调用，虚拟机需要在栈里存在一个对象引用（objectref）。如果该方法需要参数，那么除了objectref，虚拟机还需要在栈中存在该方法所需要的参数（args）。如果这个方法是一个类方法，虚拟机不再需要objectref，因为虚拟机不会在对象上调用一个类方法，栈中存在的将只有args。

objectref和args（或者在类方法的情况下只有args）必须在调用指令执行前，被其他指令压入所调用方法的操作数栈。

19.1.1 Java方法的调用

如第5章所述，虚拟机为每一个调用的Java（非本地）方法建立一个新的栈帧。栈帧包括：为方法的局部变量所预留的空间，该方法的操作数栈，以及特定虚拟机实现需要的其他所有的信息。局部变量和操作数栈的大小在编译时计算出来，并放置到class文件中去，然后虚拟机就能够了解到方法的栈帧需要多少内存。当虚拟机调用一个方法的时候，它为该方法创建恰当大小的栈帧，再将新的栈帧压入Java栈。

处理实例方法时，虚拟机从所调用方法栈帧内的操作数栈中弹出objectref和args。虚拟机把objectref作为局部变量0放到新的栈帧中，把所有的args作为局部变量1，2，……等处理。objectref是隐式传给所有实例方法的this指针。

对于类方法，虚拟机只从所调用方法栈帧中的操作数栈中弹出参数，并将它们放到新的栈帧中去作为局部变量0，1，2，……

当objectref和args（对于类方法则只有args）被赋给新栈帧中的局部变量之后，虚拟机把新的栈帧作为当前栈帧，然后将程序计数器指向新方法的第一条指令。

19.1.2 本地方法的调用

如第5章所述，虚拟机使用一种“与实现相关”的风格调用本地方法。当调用本地方法时，虚拟机不会将一个新的栈帧压入Java栈。当线程进入到本地方法的那一刻，它就将Java栈抛在身后。直到本地方法返回以后，Java栈才被重新使用。

19.2 方法调用的其他形式

尽管通常使用invokevirtual指令调用实例方法，但在某些特定的情况中，也会使用另外两种操作码——invokespecial和invokeinterface，如表19-2所示。

表19-2 方法调用

操作码	操作数	说明
invokespecial	indexbyte1，indexbyte2	把objectref和args从栈中弹出，调用常量池索引指向的实例方法
invokeinterface	indexbyte1，indexbyte2	把objectref和args从栈中弹出，调用常量池索引指向的实例方法

当根据引用的类型来调用实例方法，而不是根据对象的类来调用的时候，通常使用invokespecial指令。这又分三种情况：

1）实例初始化（<init>（））方法。

2) 私有方法。

3) 使用super关键字所调用的方法。

当给出一个接口的引用时，使用invokeinterface来调用一个实例方法。

如第7章所述，Java虚拟机总是直接调用类初始化（或者<clinit>（））方法，类的初始化方法永远不会被任何字节码调用。在Java虚拟机指令集中，没有任何调用<clinit>（）方法的指令。如果class文件尝试使用本章所介绍的四种指令来调用<clinit>（）方法，会导致虚拟机抛出异常。

19.3 指令invokespecial

invokespecial和invokevirtual的主要区别在于：invokespecial通常（只有一个例外）根据引用的类型选择方法，而不是根据对象的类来选择。换句话说，它使用静态绑定而不是动态绑定。在下列使用invokespecial的三种情况中，动态绑定并不会产生所期望的结果。

19.3.1 指令invokespecial 和 <init>（）方法

如第7章所述，<init>（）方法（或者实例初始化方法）是编译器为构造方法和实例变量初始化方法放置代码的地方。类会为源文件中每个构造方法提供一个<init>（）方法。如果没有在源文件中显式声明一个构造方法，编译器就会为这个类产生一个默认的无参数的构造方法。这个默认构造方法通常以class文件中的一个<init>（）方法结束。因此，就像每个类都至少会有一个构造方法一样，每个类都至少会有一个<init>（）方法，这些方法通常使用invokespecial调用。

只有创建一个新的实例的时候，才调用<init>（）方法。新创建对象的继承路径中，每个类都至少会调用一个<init>（）方法，而且继承路径中的任何一个类都有可能调用多个<init>（）方法。

使用invokespecial来调用<init>（）的原因在于：子类的<init>（）方法需要拥有调用超类的<init>（）方法的能力。这就揭示了为什么一个对象实例化时，有多个<init>（）方法被调用的原因。虚拟机调用对象类中声明过的<init>（）方法，这个<init>（）方法首先调用同一个类中的其他<init>（）方法或者超类中的<init>（）方法，这个过程贯穿于对象的整个生命周期。

例如，考虑下列代码：

```
// On CD-ROM in file invoke/ex1/Dog.java
class Dog {
}
// On CD-ROM in file invoke/ex1/CockerSpaniel.java
class CockerSpaniel extends Dog {

    public static void main(String args[]) {
        CockerSpaniel bootsie = new CockerSpaniel();
    }
}
```

当调用main（）方法的时候，虚拟机会为新的CockerSpaniel对象分配空间，然后调用CockerSpaniel的默认无参的<init>（）方法来初始化那段空间。那个方法将会首先调用Dog的<init>（）方法，然后再依次调用Object的<init>（）方法。类CockerSpaniel的main（）方法的字节码如下：

```
            // Create a new CockerSpaniel object, push ref
0 new #1 <Class CockerSpaniel>
            // Invoke <init>() method on object ref
            // new CockerSpaniel();
3 invokespecial #3 <Method <init>() of class CockerSpaniel>
            // Note compiler didn't store resulting ref in a var
            // representing bootsie, because it was never used
6 return    // return void from main()
```

CockerSpaniel类的main（）方法为新的CockerSpaniel对象分配内存，并使用new指令将分配的内存初始化为默认的初始化值（“#1”指出了需要实例化的类的常量池入口，这里特指CockerSpaniel类）。new指令把一个指向CockerSpaniel对象的引用压入栈。然后main（）方法使用invokespecial指令通过该对象引用调用类CockerSpaniel的<init>（）方法（“#3”指出了常量池入口，其中包含了对CockerSpaniel的<init>（）方法的引用）。Java虚拟机把一个新的栈帧压入Java栈，然后把对象引用赋给新栈帧中的局部变量0。类CockerSpaniel的<init>（）方法的字节码如下所示：

```
0 aload_0    // Push object ref from local var 1
             // Invoke Dog's <init>() on object ref
1 invokespecial $4   <Method <init>() of class Dog>
4 return     // return void from CockerSpaniel's <init>()
```

如前所述，这里的<init>（）方法相当于编译器为类CockerSpaniel自动产生的默认无参数构造方法。这个方法首先把从局部变量0中取出的已被初始化的对象引用压入栈，然后通过这个引用调用Dog的<init>（）方法（“#4”指出了常量池入口，其中包含了指向Dog的<init>（）方法的引用）。Dog类的<init>（）方法的字节码如下所示：

```
0 aload_0    // Push obj ref from local var 0
             // Invoke Object's <init>() method on obj ref
1 invokespecial #3   <Method <init>() of class java.lang.Object>
4 return     // return void from Dog's <init>()
```

这里的<init>（）方法相当于编译器为类Dog自动产生的默认无参数构造方法。这个方法首先把从局部变量0中取出的已被初始化的对象的引用压入栈，然后通过这个引用调用Object的<init>（）方法（这里的“#3”指出了常量池入口，其中包含指向Object的<init>（）方法的引用。这个常量池并不是指向类CockerSpaniel的方法的那个常量池，每个类都有自己的常量池）。当这三个<init>方法都返回后，由main（）新建的CockerSpaniel对象才完成了初始化工作。

由于每个类至少有一个<init>（）方法，类的<init>（）方法拥有相同的特征签名是很普遍的现象（方法的特征签名是指它的名字、参数的数量和类型）。例如，在CockerSpaniel类继承路径中的下列三个类的<init>（）方法，它们的特征签名相同。CockerSpaniel、Dog和Object都有一个名为<init>（）的无参数方法。

invokevirtual指令会执行动态绑定和调用CockerSpaniel的<init>（）方法，所以从CockerSpaniel的<init>（）方法使用invokevirtual指令调用Dog的<init>（）方法是不可能的。然而通过使用invokespecial指令，CockerSpaniel的<init>（）方法能够调用Dog类的<init>方法，因

为放在CockerSpaniel的class文件（常量池入口#4）中的引用的类型为Dog。

19.3.2 指令invokespecial和私有方法

当处理私有实例方法时，必须允许子类使用与超类中实例方法同样的特征签名来声明实例方法（invokespecial只用来调用私有实例方法，不能调用私有类方法，私有类方法由invokestatic指令调用）。例如，在下列代码中，interestingMethod（）是超类中的私有方法，子类对其具有包访问权限。

```
// On CD-ROM in file invoke/ex2/Superclass.java
class Superclass {

    private void interestingMethod() {
       System.out.println("Superclass's interesting method.");
    }

    void exampleMethod() {
       interestingMethod();
    }
}

// On CD-ROM in file invoke/ex2/Subclass.java
class Subclass extends Superclass {

    void interestingMethod() {
       System.out.println("Subclass's interesting method.");
    }

    public static void main(String args[]) {
       Subclass me = new Subclass();
       me.exampleMethod();
    }
}
```

如前所述，当调用像前面所定义的Subclass中的main（）方法时，虚拟机会实例化一个新的Subclass对象，然后调用exampleMethod（）方法。类Subclass的main（）方法的字节码如下所示：

```
                // Create a new instance of class Subclass, push ref
 0 new #2 <Class Subclass>
 3 dup          // Duplicate ref, push duplicate
                // Invoke <init>() method on new object:
                // Subclass me = new Subclass();
 4 invokespecial #6 <Method <init>() of class Subclass>
 7 astore_1     // Pop object ref into local var 1
 8 aload_1      // Push ref from local var 1
                // Invoke exampleMethod() on object ref:
                // me.exampleMethod();
 9 invokevirtual #8 <Method void exampleMethod() of class Superclass>
12 return       // return void from main()
```

Subclass从Superclass处继承了exampleMethod（）方法。当方法main（）调用Subclass对象中的方法exampleMethod（）时，它使用invokevirtual指令。正如类Superclass中所定义的，Java虚拟机将会创建一个新的栈帧并将其压入栈，然后开始执行exampleMethod（）的字节码。方法exampleMethod（）的字节码如下所示：

```
0 aload_0     // Push obj ref from local var 1
              // Invoke interestingMethod() on obj ref:
              // interestingMethod();
1 invokespecial #7 <Method void interestingMethod() of Superclass>
4 return      // return void from exampleMethod()
```

需要注意的是，方法exampleMethod（）首先将赋给局部变量0的引用压入栈（隐含参数this被传给所有的实例方法），然后使用invokespecial指令通过这个引用调用interestingMethod（）方法。尽管这里的对象是Subclass类的实例，而且Subclass类中的interestingMethod（）方法也是能够访问的，但Java虚拟机最终还是调用了Superclass类中的interestingMethod（）方法。

程序的正确输出为："Superclass's interesting method"。如果使用指令invokevirtual而不是指令invokespecial来调用方法interestingMethod（），那么程序的输出结果会是："SubClass's interesting method"。这是因为，虚拟机将会根据对象实际所属的类来选择调用哪一个interestingMethod（），这里对象实际所属的类是Subclass，因此，虚拟机将会使用Subclass的interestingMethod（）。相反，当使用invokespecial指令时，虚拟机会按照引用的类型来选择调用的方法，因此就会调用Superclass的*interestingMethod*（）版本。

19.3.3 指令invokespecial和super关键字

当使用super关键字来调用方法时（例如super.someMethod（）），尽管当前类重载了该方法，但使用者真正希望调用的是超类的方法。再说一次，指令invokevirtual只能调用当前类的方法，无法使用超类的方法。例如，考虑如下代码：

```
// On CD-ROM in file invoke/ex3/Cat.java
class Cat {

    void someMethod() {
    }
}

//  On CD-ROM in file invoke/ex3/TabbyCat.java
class TabbyCat extends Cat{

    void someMethod() {
        super.someMethod();
    }
}
```

类TabbyCat的someMethod（）方法的字节码如下：

```
0 aload_0     // Push obj ref from local var 0
              // Invoke Cat's someMethod() on obj ref
```

```
1 invokespecial #4 <Method void someMethod() of class Cat>
4 return     // return void from TabbyCat's someMethod()
```

如果这里使用invokevirtual指令，那么将会调用类TabbyCat的someMethod（）方法。但因为这里使用了invokespecial指令，并且常量池入口（这里是#4）指明调用的是类Cat中声明的someMethod（）方法，因此Java虚拟机将会准确无误地调用超类（类Cat）的someMethod（）方法。

Java虚拟机是否使用静态绑定来执行invokespecial指令（或者使用一种特殊的动态绑定）取决于所指向的类是否设定了ACC_SUPER标志。JDK 1.0.2版本以前的各个版本中，invokespecial指令的名称为invokenonvirtual，而且总会导致静态绑定的使用，结果却是invokenonvirtual所坚持的静态绑定无法保证所有情况下Java语言语义的正确实现（换句话说，这是指令集中的一个“Bug”）。在JDK 1.0.2版本中，invokenonvirtual指令更名为invokespecial，它的语义也改变了。此外，Java class文件中的access_flags项中还加入了一个新的标志：ACC_SUPER。

class文件的ACC_SUPER标志项指明，Java虚拟机应该使用哪一种语义来执行在class文件字节码中所遇到的invokespecial指令。如果没有设置ACC_SUPER标志，虚拟机将会使用旧语义（invokenonvirtual）；如果设置了ACC_SUPER标志，虚拟机将会使用新语义。所有新版本的Java编译器都会在生成的class文件中默认设置ACC_SUPER标志。

根据旧语义，当执行invokespecial指令时，虚拟机将会在任何情况下都使用静态绑定。而新语义除了调用超类方法之外，其他情况下一律使用静态绑定。

根据新语义，当Java虚拟机解析一个invokespecial指令中指向超类方法的符号引用时，它会动态搜寻当前类的超类，找到离得最近的超类中的该方法的实现（换句话说，就是在继承树中与当前类最接近的超类中所声明的方法的实现）。在大多数情况下，虚拟机很可能发现最近方法实现存在于符号引用中列出的超类中。但是虚拟机也可能在另外一个不同的超类中发现最近的方法实现。

例如，如果创建了一棵包含3个类的继承树（这三个类是Animal、Dog和CockerSpaniel）。假设类Dog是Animal类的子类，CockerSpaniel类是Dog类的子类，在CockerSpaniel类中定义了一个方法，它使用invokespecial来调用一个名为walk（）的非私有的超类方法。再假设当编译CockerSpaniel时，编译器设定了ACC_SUPER标志。此外，假设当编译CockerSpaniel类时，Animal类定义了walk（）方法，但Dog类没有定义该方法。此时，CockerSpaniel类中指向walk（）方法的符号引用将会把Animal类作为它的类。当执行CockerSpaniel类中的invokespecial指令时，虚拟机会进行动态选择，并调用Animal类的walk（）方法。

如果后来在Dog类中加入了walk（）方法，并且重新编译Dog类，但是没有重新编译CockerSpaniel类，它的指向超类的walk（）方法的符号引用仍然会将Animal作为自己的类，尽管在Dog类中已经有了walk（）方法的实现。不过，当执行CockerSpaniel类中的invokespecial指令时，虚拟机将会动态选择并调用Dog类中的walk（）方法的实现。

19.4 指令invokeinterface

操作码invokeinterface与invokevirtual的功能相同：它调用实例方法并使用动态绑定。这两

条指令的区别在于：当引用的类型为类的时候，使用invokevirtual；当引用类型为接口时，使用invokeinterface。

Java虚拟机使用不同于类引用的操作码来调用接口引用的方法，这是因为Java不能像使用类引用那样，使用许多与方法表偏移量相关的假设（方法表在第8章中提及）。对于类引用来说，无论对象实际的类是什么，方法在方法表中始终占据相同的位置。但对于接口引用来说，情况就不是这样了。位于不同类中的同一方法所占据的位置是不同的，尽管这些类实现同一个接口。

19.5节列出了在字节码中使用invokeinterface的例子。

19.5 指令的调用和速度

可想而知，调用接口引用方法可能要比调用类引用方法慢。因为，当Java虚拟机遇到invokevirtual指令时，它把实例方法的符号引用解析为直接引用，所生成的直接引用很可能是方法表中的一个偏移量，而且从此往后都可以使用同样的偏移量。但对于invokeinterface指令来说，虚拟机每一次遇到invokeinterface指令，都不得不重新搜寻一遍方法表，因为虚拟机不能够假设这一次的偏移量与上一次的偏移量相同。

最快的指令是invokespecial和invokestatic，因为这些指令调用的都是静态方法。当Java虚拟机为这些指令解析符号引用时，将符号引用转换成为直接引用，所生成的直接引用将包含一个指向实际操作码的指针。

19.6 方法调用的实例

下面的代码举例说明了Java虚拟机调用方法的多种途径，并对各种情况下适合使用什么样的操作码做出了演示。

```
// On CD-ROM in file invoke/ex4/InYourFace.java
interface InYourFace {
    void interfaceMethod ();
}

// On CD-ROM in file invoke/ex4/ItsABirdItsAPlaneItsSuperclass.java
class ItsABirdItsAPlaneItsSuperclass implements InYourFace {

    ItsABirdItsAPlaneItsSuperclass(int I) {
        super();                    // invokespecial (of an <init>)
    }

    static void classMethod() {
    }

    void instanceMethod() {
    }

    final void finalInstanceMethod() {
```

```
    }

    public void interfaceMethod() {
    }
}

// On CD-ROM in file invoke/ex4/Subclass.java
class Subclass extends ItsABirdItsAPlaneItsSuperclass {

    Subclass() {
        this(0);                // invokespecial (of an <init>)
    }

Subclass(int I) {
    super(i);                   // invokespecial (of an <init>)
}

private void privateMethod() {
}

void instanceMethod() {
}

final void anotherFinalInstanceMethod() {
}

void exampleInstanceMethod() {

    instanceMethod();               // invokevirtual
    super.instanceMethod();         // invokespecial

    privateMethod();                // invokespecial

    finalInstanceMethod();          // invokevirtual
    anotherFinalInstanceMethod();   // invokevirtual

    interfaceMethod();              // invokevirtual

    classMethod();                  // invokestatic
    }
}
// On CD-ROM in file invoke/ex4/UnrelatedClass.java
class UnrelatedClass {

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

```
        Subclass sc = new Subclass();  // invokespecial (on an <init>)
        Subclass.classMethod();        // invokestatic
        sc.classMethod();              // invokestatic
        sc.instanceMethod();           // invokevirtual
        sc.finalInstanceMethod();      // invoke virtual
        sc.interfaceMethod();          // invokevirtual

        InYourFace iyf = sc;
        iyf.interfaceMethod();         // invokeinterface
    }
}
```

下面是由javac为每一个类所产生的字节码（编译器没有为InYourFace接口产生字节码）：

```
// Methods of class ItsABirdItsAPlaneItsSuperclass

// Method <init>(int)
 0 aload_0    // Push obj ref from local var 0
              // Invoke Object's <init>() on obj ref: super();
1 invokespecial #4 <Method <init>() of class java.lang.Object>
4 return     // return void from <init>(int)

// Method void classMethod()
 0 return     // return void from classMethod()
// Method void instanceMethod()
 0 return     // return void from instanceMethod()

// Method void finalInstanceMethod()
 0 return     // return void from finalInstanceMethod()

// Method void interfaceMethod()
 0 return     // return void from interfaceMethod()

// ————————————————
// Methods of class Subclass

// Method <init.()
 0 aload_0    // Push obj ref from local var 0
 1 iconst_0   // Push int constant 0
              // Invoke Subclass's <init>(int): this(0);
2 invokespecial #4 <Method <init>(int) of class Subclass>
5 return     // return void from Subclass's <init>()

// Method <init>(int)
 0 aload_0    // Push obj ref from local var 0
 1 iload_1    // Push int from local var 1
              // Invoke ItsABirdItsAPlaneItsSuperclass's <init>(int):
```

```
            // super(i);
 2 invokespecial #3 <Method <init>(int) of class
            ItsABirdItsAPlaneItsSuperclass>
 5 return    // return void from Subclass's <init>(int)

// Method void privateMethod()
 0 return    // return void from privateMethod()

// Method void instanceMethod()
 0 return    // return void from instanceMethod()

// Method void anotherFinalInstanceMethod()
 0 return    // return void from anotherFinalInstanceMethod()

// Method void exampleInstanceMethod()
 0 aload_0   // Push obj ref from local var 0
             // Invoke instanceMethod() on obj ref: instanceMethod();
1 invokevirtual #9 <Method void instanceMethod() of class Subclass>
4 aload_0    // Push obj ref from local var 0 again
             // Invoke ItsABirdItsAPlaneItsSuperclass's
             // instanceMethod() on obj ref: super.instanceMethod();
5 invokespecial #8 <Method void instanceMethod() of class
      ItsABirdItsAPlaneItsSuperclass>
8 aload_0    // Push obj ref from local var 0 yet again
             // Invoke Subclass's privateMethod() on obj ref:
             // privateMethod();
9 invokespecial #11 <Method void privateMethod() of class Subclass>
12 aload_0   // Push, you guessed it, obj ref from local var 0
             // Invoke finalInstanceMethod() on obj ref:
             // finalInstanceMethod();
13 invokevirtual #7 <Method void finalInstanceMethod() of class
      ItsABirdItsAPlaneItsSuperclass>
16 aload_0   // Push obj ref from local var 0
             // Invoke anotherFinalInstanceMethod() on obj ref:
             // anotherFinalInstanceMethod();
17 invokevirtual #5 <Method void anotherFinalInstanceMethod() of
      class Subclass>
20 aload_0   // Push obj ref from local var 0
             // Invoke interfaceMethod() on obj ref:
             // interfaceMethod();
21 invokevirtual #10 <Method void interfaceMethod() of class
      ItsABirdItsAPlaneItsSuperclass>
            // Invoke ItsABirdItsAPlaneItsSuperclass's static
            // classMethod(): classMethod();
24 invokestatic #6 <Method void classMethod() of class
      ItsABirdItsAPlaneItsSuperclass>
```

```
27 return     // return void from Subclass's exampleInstanceMethod()

// ----------------------
// Methods of class UnrelatedClass

// Method <init>()
 0 aload_0    // Push obj ref from local var 0
              // Invoke Object's <init>() on obj ref
1 invokespecial #7 <Method java.lang.Object()>
4 return      // return void from UnrelatedClass's <init>()

// Method void main(java.lang.String[])
              // Create new Subclass object, push ref
 0 new #3 <Class Subclass>
3 dup         // Duplicate obj ref, push duplicate
              // Invoke Subclass's <init>() on obj ref:
              // new Subclass();
4 invokespecial #6 <Method <init>() of class Subclass>
              // Pop obj ref, store into local var 1:
7 astore_1    // Subclass sc = new Subclass();
              // Invoke ItsABirdItsAPlaneItsSuperclass's static
              // classMethod(): Subclass.classMethod();
8 invokestatic #8 <Method void classMethod() of class
      ItsABirdItsAPlaneItsSuperclass>
              // Invoke ItsABirdItsAPlaneItsSuperclass's static
              // classMethod(): sc.classMethod();
11 invokestatic #8 <Method void classMethod() of class
       ItsABirdItsAPlaneItsSuperclass>
14 aload_1    // Push obj ref from local var 1
              // Invoke instanceMethod() on obj ref:
              // sc.instanceMethod()
15 invokevirtual #10 <Method void instanceMethod() of class Subclass>
18 aload_1    // Push obj ref from local var 1
              // Invoke finalInstanceMethod() on obj ref:
              // sc.finalInstanceMethod()
19 invokevirtual #9 <Method void finalInstanceMethod() of class
       ItsABirdItsAPlaneItsSuperclass>
22 aload_1    // Push obj ref from local var 1
              // Invoke interfaceMethod() on obj ref:
              // sc.interfaceMethod()
23 invokevirtual #12 <Method void interfaceMethod() of class
       ItsABirdItsAPlaneItsSuperclass>
26 aload_1    // Push obj ref from local var 1
              // Pop obj ref, store in local var 2:
27 astore_2   // InYourFace iyf = sc;
28 aload_2    // Push obj ref from local var 2
```

```
            // Invoke InYourFace's interfaceMethod() on obj ref:
            // iyf.interfaceMethod();
29 invokeinterface (args 1) #11 <Method void interfaceMethod() of
       interface InYourFace>
34 return    // Return void from UnrelatedClass's main()
```

19.7 从方法中返回

下列几种操作码具有使方法返回的功能——每一种操作码对应一种返回的数据类型，如表19-3所示，这些操作码都没有操作数。如果有返回值，它必须被放置在操作数栈中。返回值从操作数栈中弹出，然后又被压入调用方法栈帧的操作数栈中。弹出当前栈帧，调用方法的栈帧成为当前栈帧。程序计数器被重置，其值为调用方法中紧随调用返回方法那条指令的下一条指令。

指令ireturn用于返回int、char、byte或者short数据类型的方法。

表19-3 抛出异常

操作码	操作数	说明
ireturn	（无）	弹出int类型值，压入调用方法的栈并返回
lreturn	（无）	弹出long类型值，压入调用方法的栈并返回
freturn	（无）	弹出float类型值，压入调用方法的栈并返回
dreturn	（无）	弹出double类型值，压入调用方法的栈并返回
areturn	（无）	弹出对象引用，压入调用方法的栈并返回
return	（无）	返回void

19.8 随书光盘

光盘的invoke子目录中有本章源代码示例。

19.9 资源页

如果需要了解更多与本章相关的信息，请访问资源页面http://www.artima.com/insidejvm/resources。

第20章 线 程 同 步

可以在语言级支持多线程是Java语言的一大优势，这种支持主要集中在同步上，或调节多个线程的活动和共享数据。Java所使用的同步机制是监视器。本章讲述了监视器，展示了Java虚拟机使用它们的方式，还描述了指令集在数据的锁定和解锁方面是如何支持监视器的。

20.1 监视器

Java中的监视器支持两种线程：互斥和协作。Java虚拟机通过对象锁来实现互斥，允许多个线程在同一个共享数据上独立而互不干扰地工作。协作则是通过Object类的wait方法和notify方法来实现，允许多个线程为了同一个目标而共同工作。

我们可以将监视器比作一个建筑，它有一个很特别的房间，房间里有一些数据，而且在同一时间只能被一个线程占据。一个线程从进入这个房间到它离开前，它可以独占地访问房间中的全部数据。我们用一些术语来定义这一系列动作，进入这个建筑叫做“进入监视器”，进行建筑中的那个特别的房间叫作“获得监视器”，占据房间叫做“持有监视器”，离开房间叫做“释放监视器”，离开建筑叫做“退出监视器”。

除了与一些数据关联之外，监视器还会关联到一些或更多的代码——在本书中，这样的代码被称作监视区域。对于一个监视器来说，监视区域是最小的、不可分割的代码块。换句话说，在同一个监视器中，监视区域只会同时被一个线程执行，即使同时有多个并发的线程，监视器会保证在监视区域上同一时间只会执行一个线程。一个线程想要进入监视器的惟一途径就是到达该监视器所关联的一个监视区域的开始处，而线程想要继续执行线程区域的惟一途径就是获得该监视器。

当一个线程到达了一个监视区域的开始处，它就会被放置到该监视器的入口区。入口区就好像是监视器前面的走廊。如果没有其他线程在入口区中等待，也没有线程正持有该监视器，则这个线程就可以获得监视器，并继续执行监视区域中的代码。当这个线程执行完监视区域后，它就会退出（并释放）该监视器。

如果一个线程到达了一个监视区域的开始处，但这个监视区域已经有线程持有该监视器而被该监视器保护起来，则这个刚刚到达的线程必须在入口区等待。当监视器的持有者退出监视器后，新到达的线程必须与其他已经在入口区等待的线程进行一次比赛，最终只会有一个线程赢得比赛并获得监视器。

前面提到的第一种同步——互斥，可以在多线程环境中互斥地执行一段被称作监视区域的代码，在任何时候，特定监视器上只会有一个线程执行监视区域。通常，互斥只在多个线程需要共享数据或其他资源时显得重要，如果两个线程并不使用任何公有数据或资源，它们通常会互不干扰，也就不需要互斥执行。可是如果Java虚拟机实现不是基于时间片的，即使没有线程共享数据，一个不被阻塞的高优先级的线程也将妨碍任何优先级比它低的线程。高优先级的线程

会独占CPU，从而让低优先级的线程永远得不到CPU时间和执行的机会。在这种情况下，使用一个并不保护任何数据的监视器来协调这些线程，以确保所有的线程都可以得到一些CPU时间。不过，大多数情况下，监视器保护那些通过监视区域代码来访问的数据，在这种情况下——即要求数据仅可以由监视区域访问，监视器可以确保线程会互斥地访问这些数据。

另一种我们提到的被监视器所支持的同步是协作。互斥帮助线程在访问共享数据时不被其他线程干扰，而协作帮助线程与其他线程共同工作。

当一个线程需要一些特别状态的数据，而由另一个线程负责改变这些数据的状态时，同步就显得特别重要。举个例子，一个“读线程”会从缓冲区中读数据，而另一个“写线程”会向缓冲区中填充数据。“读线程”需要缓冲区处于一个“非空”的状态，这样它才可以从中读数据，如果“读线程”发现缓冲区是空的，它就必须等待。“写线程”就负责向缓冲区中写数据，只有“写线程”完成了一些数据的写入，“读线程”才能做相应的读取动作。

Java虚拟机所使用的这种监视器被称作“等待并唤醒”监视器（有时也被称作“发信号并继续”监视器）。在这种监视器中，一个已经持有监视器的线程，可以通过执行一个等待命令，暂停自身的执行。当线程执行了等待命令后，它就会释放监视器，并进入一个等待区，这个线程会在那里一直持续暂停状态，直到一段时间后，这个监视器中的其他线程执行了唤醒命令。当一个线程执行了唤醒命令后，它会继续持有监视器，直到它主动释放监视器，如执行了一个等待命令或者执行完监视区域。当执行唤醒的线程释放了监视器后，等待线程会苏醒，并重新获得监视器。

Java虚拟机中的这种监视器有时也被称作“发信号并继续”监视器的原因是，在一个线程做了唤醒操作（发信号）后，它还会继续持有监视器并继续执行监视区域（继续），过了一些时间之后，唤醒线程释放监视器，等待线程才会苏醒。推断起来，等待线程将自身挂起是因为监视器保护数据并不处于它想要继续执行正确操作的状态。同样，唤醒线程在它将监视器保护数据置成等待线程想要的状态后执行唤醒命令。但是因为唤醒线程会继续执行，它可能会在执行唤醒后又修改了数据的状态，让等待线程不能够继续工作。另一种情况是，第三个线程可能在唤醒线程释放了监视器，而等待线程还没有获得监视器之前抢先获得了监视器，而且这个线程可能会修改监视器保护数据的状态。因为以上事实，一次唤醒往往会被等待线程看作是一次提醒，告诉它“数据已经是你想要的状态了”。每次等待线程苏醒的时候，它都需要再次检查状态，以确定是否可以继续完成工作。如果数据不是它所需要的状态，这个线程可能会再次执行等待命令或者放弃等待退出监视器。

例如，再考虑我们上面提到过的场景：一个缓冲区，一个读线程，一个写线程。假定缓冲区是由某个监视器所保护的，当读线程进入这个监视器时，它会检查缓冲区是否是空的，如果缓冲区不是空的，读线程会从中读取（并且删除）一些数据，然后退出了监视器。而如果这个缓冲区是空的，读线程会执行一个等待命令，同时它会暂停执行并被放入等待区。这样，读线程释放了监视器，让它变得对于其他线程来说是“可用”的。稍后，写线程进入了监视器，向缓冲区中写了一些数据，然后执行唤醒，并退出监视器。当写线程执行了唤醒后，读线程被标志为“可能苏醒”，当写线程退出监视器后，读线程被唤醒并成为监视器的持有者。如果存在其他线程先进入了监视器并“消耗”完写线程留下的数据的情况，这个读线程必须做明确检查，

确保缓冲区不是空的。如果不存在这种情况，读线程可以假定数据已经存在，这样读线程会从缓冲区中读取（并删除）一些数据，然后退出监视器。

Java虚拟机中的这种监视器模型如图20-1所示，将监视器分成了三个区域。中间的大方框包括一个单独的线程，是监视器的持有者；左边小的方框中是入口区；右边另一个小方框是等待区。活动线程用深灰色圆画出，暂停的线程用浅灰色圆画出。

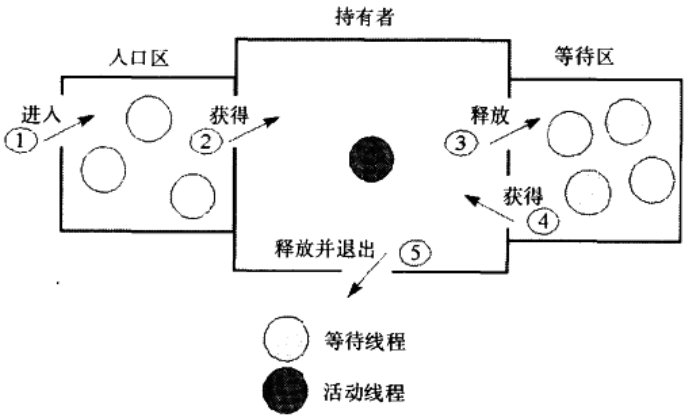


图20-1 Java监视器

图20-1也显示了线程与监视器交互所必须“通过”的几道门。当一个线程到达监视区域的开始处时，它会通过最左边的1号门进入监视器，发现自己身处在那个叫入口区的方框中。如果没有任何线程正持有监视器，也没有其他线程正在入口区中等待，这个线程就会立刻通过下一道门——2号门，并持有监视器。作为这个监视器的持有者，它将继续执行监视区域中的代码。或者也可能出现另一种情况，已经有另一个线程正持有监视器，这个新到达的线程就必须在入口区中等待，很可能那里已经有一些线程在等待了，这个线程会被阻塞，所以不能执行监视区域中的代码。

图20-1中有3个线程在入口区中暂停，有4个线程在等待区中暂停，这些线程会一直在那儿，直到监视器当前的持有者（即活动线程）释放监视器。活动线程会通过两条途径释放监视器：完成它正在执行的监视区域或者执行一个等待命令。如果它执行完监视区域，它会通过中间方框下方的5号门退出监视器。如果它执行了等待命令，它会通过3号门进入等待区，从而释放监视器。

如果上一个监视器的持有者在它释放监视器前没有执行唤醒命令（同时在此之前也没有任何等待线程被唤醒并等待苏醒），那么位于入口区中的那些线程将会竞争获得监视器。如果上一个持有者执行了唤醒命令，入口区中的线程就不得不与一个或多个等待区中的线程来竞争。如果入口区中的一个线程在竞争中获胜，它就会通过2号门，从而成为监视器的新的持有者。而如果是等待区中的某个线程赢了，它会通过4号门退出等待区并重新获得监视器。注意，一个线程只有通过3号和4号门才能进入或退出等待区。一个线程只有在它正持有监视器时才能执行等待命令，而且它只能通过再次成为监视器的持有者才能离开等待区。

在Java虚拟机中，线程在执行等待命令时可以随意指定一个暂停时间。如果一个线程指定了暂停时间，而且在暂停时间截止之前没有其他线程执行唤醒命令，这个等待线程会从虚拟机得到一个自动唤醒的命令。也就是说，在暂停时间到了之后，即使没有来自其他线程的明确的唤醒命令，它也会自动苏醒。

Java虚拟机提供了两种唤醒命令："notify"和"notify all"。notify命令随意从等待区中选择一个线程并将其标志为"可能苏醒"，而notify all命令会将等待区中的所有线程都标志成"可能苏醒"。

Java虚拟机如何从等待区以及入口区选择下一个线程来执行，在很大程度上取决于Java虚拟机的设计者。比如说，设计者可以在以下方面做出选择：

- 哪一个等待区中的线程将获得notify命令。
- 使用notify all命令后，位于等待区的线程依次苏醒的顺序。
- 允许入口区的线程获得监视器的顺序。
- 在得到一个notify命令后，在入口区（与等待区对比）中如何选择线程。

可以设想使用先入先出（FIFO）队列来管理入口区和等待区比较合理，那么在等待区排队时间最长的那个线程将首先获得监视器。或者，也可以选择实现10个先入先出队列，分别为Java虚拟机中每一种线程优先级服务。虚拟机会选其中有等待线程的、优先级最高的队列中等待时间最长的那一个。实现可能采取这些方法，但是你不能假设它采取某种方法。实现可以自由选择，它可能是按照后进先出（LIFO）来管理队列，或者选择更低优先级的线程，而不是选择更高优先级的线程，也可能做出任何其他没有意义的决定。总之，实现可以任意自由选择哪一个线程。

程序员必须不依赖任何特定的有关优先级的算法或者安排，至少在编写平台无关的Java程序时应该这样。比如说，因为不知道notify命令将会导致等待区中的哪一个线程苏醒，只有当绝对确认只会有一个线程在等待区中挂起的时候，才应该使用notify（相对notify all而言）。只要存在同时有多个线程在等待区中被挂起的可能性，就应该使用notify all。否则，在某些Java虚拟机实现中可能导致某个特定的线程在等待区中等待非常长的时间。如果虚拟机采取最后到达等待区的线程总是被notify优先选择的话，早就开始在等待区等待的那些线程可能永远不会有机会苏醒。

20.2 对象锁

在前面的章节中提到过，Java虚拟机的一些运行时数据区会被所有的线程共享，其他的数据是各个线程私有的。因为堆和方法区是被所有线程共享的，Java程序需要为两种多线程访问数据进行协调：

- 保存在堆中的实例变量。
- 保存在方法区中的类变量。

程序不需要协调保存在Java栈中的局部变量，因为Java栈中的数据是属于拥有该栈的线程私有的。

在Java虚拟机中，每个对象和类在逻辑上都是和一个监视器相关联的。对于对象来说，相关联的监视器保护对象的实例变量。对于类来说，监视器保护类的类变量。如果一个对象没有实例变量，或者一个类没有类变量，相关联的监视器就什么都不监视。

为了实现监视器的排他性监视能力，Java虚拟机为每一个对象和类都关联一个锁（有时候被称为互斥体（mutex））。一个锁就像一种任何时候只允许一个线程“拥有”的特权。线程访问实例变量或者类变量不需要获取锁。但是如果线程获取了锁，那么在它释放这个锁之前，就没有其他线程可以获取同样数据的锁了。（“锁住一个对象”就是获取对象相关联的监视器。）

类锁实际上用对象锁实现。前几章提到过，当Java虚拟机装载一个class文件的时候，它会创建一个java.lang.Class类的实例。当锁住一个类的时候，实际上锁住的是那个类的Class对象。

一个线程可以允许多次对同一个对象上锁。对于每一个对象来说，Java虚拟机维护一个计数器，记录对象被加了多少次锁。没有被锁的对象的计数器是0。但一个线程第一次获得锁的时候，计数器跳到1。线程每加锁一次，计数器就加1。（只有已经拥有了这个对象的锁的线程才能对该对象再次加锁。在它释放锁之前，其他的线程不能对这个对象加锁。）每当线程释放锁一次，计数器就减1。当计数器跳到0的时候，锁就被完全释放了，其他的线程才可以使用它。

Java虚拟机中的一个线程在它到达监视区域开始处的时候请求一个锁。在Java中，有两种监视区域：同步语句和同步方法（这些在本章后面会详细描述）。Java程序中每一个监视区域都和一个对象引用相关联。当一个线程到达监视区域的第一条指令的时候，线程必须对该引用对象加锁，否则线程不允许执行其中的代码。一旦它获得了锁，线程就进入了被保护的代码。当线程离开这块代码的时候，不管它是如何离开的，它都会释放相关对象上的锁。

注意，Java编程人员不需要自己动手加锁，对象锁是在Java虚拟机内部使用的。在Java程序中，你只需要编写同步语句或者同步方法就可以标志一个监视区域。当Java虚拟机运行你的程序的时候，每一次进入一个监视区域的时候，它每次都会自动锁上对象或者类。

20.3 指令集中对同步的支持

如前所述，语言提供了两种内置方式来标志监视区域：同步语句和同步方法。这两个机制实现了同步的互斥，是被Java虚拟机的指令集支持的。

20.3.1 同步语句

要建立一个同步语句，在一个计算对象引用的表达式中加上synchronized关键字就可以了，就像下面的reverseOrder（）方法这样：

```
// On CD-ROM in file threads/ex1/KitchenSync.java
class KitchenSync {

    private int[] intArray = new int[10];

    void reverseOrder() {
       synchronized (this) {
           int halfWay = intArray.length / 2;
           for (int i = 0; i < halfWay; ++i) {
               int upperIndex = intArray.length - 1 - i;
               int save = intArray[upperIndex];
               intArray[upperIndex] = intArray[i];
               intArray[i] = save;
```

```
            }
        }
    }
    // ...
}
```

在上面的例子中，如果没有获得对当前对象（this）的锁，在同步语句块内的语句是不会被执行的。如果使用的不是this引用，而是用一个表达式获得对另一个对象的引用，在线程执行语句体之前，需要获得那个对象的锁。如果用表达式获得对Class类实例的引用，就需要锁住那个类。

方法内的同步语句块会使用monitorenter和monitorexit这两个操作码。这些操作码在表20-1中。

表20-1 监视器

操作码	操作数	描述
monitorenter	无	弹出objectref（对象引用），获得和objectref相关联的锁
monitorexit	无	弹出objectref，释放和objectref相关联的锁

当Java虚拟机遇到monitorenter的时候，它获得栈中objectref所引用的对象的锁。如果线程已经拥有了那个对象的锁，锁的计数器会加1。线程中每条monitorexit指令都会引起计数器减1。当计数器变成0的时候，监视器就被释放了。

下面是KitchenSync类的reverseOrder（）方法生成的字节码序列：

```
// First place the reference to the object to lock into local
// variable 1. This local variable will be used by both the
// monitorenter and monitorexit instructions.
 0 aload_0          // Push local var 0 (the this reference)
 1 astore_1         // Store into local var 1

// Now acquire the lock on the referenced object
                    // Push local var 1 (the this reference; the
 2 aload_1          // object to lock)
                    // Pop reference, acquire the lock
 3 monitorenter     // on referenced object

// The code of the synchronized block begins here. A thread will not
// execute the next instruction, aload_0, until a lock has been
// successfully acquired on the this reference above.
 4 aload_0          // Push the object ref at loc var 0 (the this ref)
                    // Pop object ref, push ref to instance variable
                    // intArray
 5 getfield #4 <Field int intArray[]>
 8 arraylength      // Pop array ref, push int array length
 9 iconst_2         // Push constant int 2
10 idiv             // Pop two ints, divide, push int result
                    // Pop int into local var 3:
11 istore_3         // int halfway = intArray.length/2;
```

```
// This is the start of the code for the for loop
12 iconst_0        // Push constant int 0
13 istore 4        // Pop into local var 2: int i = 0;
15 goto 65         // Jump to for loop condition check

// This is the start of the body of the for loop
18 aload_0         // Push the object ref at loc var 0 (the this ref)
                   // Pop object ref, push ref to instance variable
                   // intArray
19 getfield #4 <Field int intArray[]>
22 arraylength     // Pop array ref, push int array length
23 iconst_1        // Push constant int 1
24 isub            // Pop two ints, subtract, push int result
25 iload 4         // Push int at local var 4 (i)
27 isub            // Pop two ints, subtract, push int result
                   // Pop int into local var 5:
28 istore 5        // int upperindex = intArray.length - 1 - i;
30 aload_0         // Push the object ref at loc var 0 (the this ref)
                   // Pop object ref, push ref to instance variable
                   // intArray
31 getfield #4 <Field int intArray[]>
34 iload 5         // Push int at local var 5 (upperIndex)
36 iaload          // Pop index, arrayref, push int at arrayref[index]
                   // Pop into local var 6:
37 istore 6        // int save = intArray[upperIndex];
39 aload_0         // Push the object ref at loc var 0 (the this ref)
                   // Pop object ref, push ref to instance variable
                   // intArray
40 getfield #4 <Field int intArray[]>
43 iload 5         // Push int at local var 5 (upperIndex)
45 aload_0         // Push the object ref at loc var 0 (the this ref)
                   // Pop object ref, push ref to instance variable
                   // intArray
46 getfield #4 <Field int intArray[]>
49 iload 4         // Push int at local var 4 (i)
51 iaload          // Pop index, arrayref, push int at arrayref[index]
                   // Set arrayref[index] = value:
52 iastore         // intArray[upperIndex] = intArray[i];
53 aload_0         // Push the object ref at loc var 0 (the this ref)
                   // Pop object ref, push ref to instance variable
                   // intArray
54 getfield #4 <Field int intArray[]>
57 iload 4         // Push int at local var 4 (i)
59 iload 6         // Push int at local var 6 (save)
                   // Set arrayref[index] = value:
```

```
61 iastore        // intArray[i] = save;

// The body of the for loop is now done, this instruction does
// the incrementing of the loop variable I
62 iinc 4 1       // Increment by 1 int at local var 4: ++i;

// This is the for loop condition check:
65 iload 4        // Push int at local var 4 (i)
67 iload_3        // Push int at local var 3 (halfway)
                  // Pop two ints, compare, jump if less than to
68 if_icmplt 18   // top of for loop body: for (; i < halfway;)
// The code of the synchronized block ends here
// The next two instructions unlock the object, making it available
// for other threads. The reference to the locked object was stored
// in local variable 1 above.
71 aload_1        // Push local var 1 (the this reference)
72 monitorexit    // Pop ref, unlock object
73 return         // return normally from method

// This is a catch clause for any exception thrown (and not caught
// from within the synchronized block. If an exception is thrown,
// the locked object is unlocked, making it available for other
// threads.
74 aload_1        // Push local var 1 (the this reference)
75 monitorexit    // Pop ref, unlock object
76 athrow         // rethrow the same exception

// The exception table shows the "catch all" clause covers the
// entire synchronized block, from just after the lock is acquired
// to just before the lock is released.
Exception table:
from to target type
4 71 74 any
```

注意，catch子句用来确保被加锁的对象将被释放（解锁），即使从同步语句块中抛出异常。不管被同步的语句块是如何退出的，线程进入这个块时获得的锁总是一定会被释放的。

20.3.2 同步方法

要同步整个方法，只需要在方法修饰符中加上synchronized关键字，如同下例所示：

```
// On CD-ROM in file threads/ex1/HeatSync.java
class HeatSync {

    private int[] intArray = new int[10];

    synchronized void reverseOrder() {
        int halfWay = intArray.length / 2;
```

```
        for (int i = 0; i < halfWay; ++i) {
            int upperIndex = intArray.length - 1 - i;
            int save = intArray[upperIndex];
            intArray[upperIndex] = intArray[i];
            intArray[i] = save;
        }
    }
    // ...
}
```

Java虚拟机调用同步方法或者从同步方法中返回没有使用任何特别的操作码。当虚拟机解析对方法的符号引用时，它判断这个方法是否是同步的。如果是同步的，虚拟机就在调用方法之前获取一个锁。对于实例方法来说，虚拟机在方法将要被调用的时候获取对象相关联的锁。对于类方法来说，它获取方法所属的类的锁（其实是对Class对象上锁）。当同步方法执行完毕的时候，不管是正常结束还是抛出异常，虚拟机都会释放这个锁。

下面是javac生成的HeatSync的reverseOrder（）方法的字节码：

```
0 aload_0       // Push the object ref at loc var 0 (the this ref)
                // Pop object ref, push ref to instance variable
                // intArray
1 getfield #4 <Field int intArray[]>
4 arraylength   // Pop array ref, push int array length
5 iconst_2      // Push constant int 2
6 idiv          // Pop two ints, divide, push int result
                // Pop int into local var 1:
7 istore_1      // int halfway = intArray.length/2;

// This is the start of the code for the for loop
8 iconst_0      // Push int constant 0
9 istore_2      // Pop into local var 2: int i = 0
10 goto 54      // Jump to for loop condition check

// This is the start of the body of the for loop
13 aload_0      // Push the object ref at loc var 0 (the this ref)
                // Pop object ref, push ref to instance variable
                // intArray
14 getfield #4 <Field int intArray[]>
17 arraylength  // Pop array ref, push int array length
18 iconst_1     // Push constant int 1
19 isub         // Pop two ints, subtract, push int result
20 iload_2      // Push int at local var 2 (i)
21 isub         // Pop two ints, subtract, push int result
                // Pop int into local var 3:
22 istore_3     // int upperindex = intArray.length - 1 - i;
23 aload_0      // Push the object ref at loc var 0 (the this ref)
                // Pop object ref, push ref to instance variable
```

```
                // intArray
24 getfield #4 <Field int intArray[]>
27 iload_3      // Push int at local var 3 (upperIndex)
28 iaload       // Pop index, arrayref, push int at arrayref[index]
                // Pop into local var 4:
29 istore 4     // int save = intArray[upperIndex];
31 aload_0      // Push the object ref at loc var 0 (the this ref)
                // Pop object ref, push ref to instance variable
                // intArray
32 getfield #4 <Field int intArray[]>
35 iload_3      // Push int at local var 3 (upperIndex)
36 aload_0      // Push the object ref at loc var 0 (the this ref)
                // Pop object ref, push ref to instance variable
                // intArray
37 getfield #4 <Field int intArray[]>
40 iload_2      // Push int at local var 2 (i)
41 iaload       // Pop index, arrayref, push int at arrayref[index]
                // Pop value, index, and arrayref,
                // Set arrayref[index] = value:
42 iastore      // intArray[upperIndex] = intArray[i];
43 aload_0      // Push the object ref at loc var 0 (the this ref)
                // Pop object ref, push ref to instance variable
                // intArray
44 getfield #4 <Field int intArray[]>
47 iload_2      // Push int at local var 2 (i)
48 iload 4      // Push int at local var 4 (save)
                // Pop value, index, and arrayref,
                // Set arrayref[index] = value:
50 iastore      // intArray[i] = save;

// The body of the for loop is now done, this instruction does
// the incrementing of the loop variable i
51 iinc 2 1     // Increment by 1 int at local var 2: ++;
// This is the for loop condition check:
54 iload_2      // Push int at local var 2 (i)
55 iload_1      // Push int at local var 1 (halfway)
                  // Pop two ints, compare, jump if less than to
56 if_icmplt 13 // top of for loop body: for (; i < halfway;)

59 return       // return (void) from method
```

如果把这段字节码和上面KitchenSync的reverseOrder（）方法的相比较，会发现这些字节码更加高效，因为这段代码没有进入和离开监视器的代码。HeatSync字节码中偏移量0到56对应KitchenSync字节码中偏移量4到68的指令。因为HeatSync的reverseOrder（）方法不包含用于保存加锁对象的局部变量，方法每一次使用的局部变量的位置都是不固定的。但是这两段这些指

令的功能是完全一致的。

两个reverseOrder（）方法的另外一个区别是编译器没有为HeatSync的reverseOrder（）方法创建异常表。在HeatSync的例子中，异常表不是必须的。当方法被调用的时候，Java虚拟机自动获取这个对象的锁。如果这个方法异常退出，如同正常退出一样，虚拟机会自动释放这个对象的锁。

同步类方法（静态方法）和上面例子中的同步实例方法以同样的方式操作。不同之处是类方法不使用this（类方法没有this），线程必须获得对应的Class实例的锁。

20.4 Object类中的协调支持

Object类声明了5个方法，程序员可以用来访问Java虚拟机同步的协调支持。这些方法都是被声明成公开的（public）和最终的（final），所以它们被所有的类继承。只有在同步方法或者同步语句中才能调用这些方法。换句话说，在这些方法被调用的时候，相关联的对象必须已经被加锁了。这些方法在表20-2中列出。

表20-2 Object类的wait和notify方法

方 法	描 述
void wait（）；	进入监视器的等待区，直到被其他线程唤醒
void wait（long timeout）；	进入监视器的等待区，直到被其他线程唤醒，或者经过timeout所指定的毫秒数后自动苏醒
void wait（long timeout，int nanos）；	进入监视器的等待区，直到被其他线程唤醒，或者经过timeout所指定的毫秒数加上nanos所指定的纳秒数后自动苏醒
void notify（）；	唤醒监视器的等待区中的一个等待线程（如果没有线程在等待，就什么都不干）
void notifyAll（）；	唤醒监视器的等待区中的所有线程（如果没有线程在等待，就什么都不干）

20.5 随书光盘

在随书光盘的threads目录下可以找到本章示例的源代码。

20.6 资源页

要了解更多关于本章内容的相关信息，请访问资源页面http://www.artima.com/insidejvm/resources。

附录A　按操作码助记符排列的指令集

本附录按照字母顺序列出了Java虚拟机指令。本附录对所有200条可以合法地出现在字节码流中的指令（存储在Java class文件中）作了详细阐述。

除了可能会出现在class文件中的这200条指令的操作码，Java虚拟机规范还定义了另外两类操作码：保留的操作码和“quick”操作码，这两种操作码不会合法地出现在Java class文件中。

Java虚拟机规范列出了三种保留操作码，如表A-1所示。这些操作码是为Java虚拟机实现及其工具所保留的。虚拟机规范保证：这三种操作码将来永远不会成为Java虚拟机指令集的一部分。可想而知，操作码breakpoint的功能是为调试程序提供断点实现。保留操作码impdep1和impdep2用来作为与实现相关的软件功能的“后门”或者与实现相关的硬件功能的“陷阱”。

Java虚拟机的第1版规范还列出了25条“_quick”操作码，这是Sun的早期虚拟机实现用来加速字节码的解释执行的。此项优化技术在第8章中有介绍，但是本附录中并没有对单独的“_quick”指令进行详细介绍（附录C中列出了“_quick”操作码的简要介绍和相应操作码的字节值）。如同保留操作码一样，“_quick”操作码可能不会合法地出现在Java class文件中。但是与保留操作码不同的是：Java虚拟机规范保留了这些执行码在将来的扩展指令集中使用新的含义的可能性。

表A-1　保留操作码

助 记 符	字 节 值
breakpoint	202(0xca)
impdep1	254(0xfe)
impdep2	255(0xff)

本附录中列出的每一条指令都会包含如下信息：

- 助记符。
- 简介。
- 使用十进制和十六进制表示的操作码的字节值。
- 指令格式。
- 指令执行前后操作数栈的状态。
- 所有相关约束的描述。
- 指令执行过程的描述。
- 指令可能抛出的所有异常和错误的描述。

“指令格式”项中列出了指令的格式，格式中项与项之间以逗号隔开，每一项都代表字节码流中的一个字节。首先出现的是操作码助记符，在助记符后列出所有的操作数。

附录对每条指令都给出两段操作数栈的快照：第一段快照的标题是“前”，显示指令执行前操作数栈的状态，第二段快照的标题是“后”，显示指令执行后操作数栈的状态。在这两段快照中，操作数栈使用以逗

号隔开的列表的方式来表示，每一项代表栈中一个字长的数据。尽管本书中别处可能使用了栈的向下递增的表现形式（栈顶端表示为图的底部），但本附录中操作数栈使用从左到右的表现形式。在每个快照中，最右边的项位于栈顶端。不受影响的操作数栈部分以省略号表示。

对每一条指令而言，“描述”段落包括三部分信息：约束、执行过程、异常和错误。如第3章所述，Java虚拟机实现需要在运行时对于它所产生的方法的字节码强制执行某种约束。出现在本附录中指令描述中的“必须”这个词，指的是执行该指令时所有虚拟机实现都必须执行的动作。例如，实现无条件跳转功能的指令goto，可以只用来跳转至同一方法的其他操作码。本附录关于goto的描述中的相关约束是这样开始的：“目标地址必须为与goto语句位于同一方法内的操作码地址。”

每一个Java虚拟机实现的设计者都能够决定如何以及何时检测字节码约束的违规。如果虚拟机实现检测到了约束违规，当（假如）运行中的程序仍然尝试执行该指令时，虚拟机必须通过抛出VerifyError异常的方法来报告这个违规。

除了描述每条指令的约束之外，附录还对每条指令的执行过程进行了描述，列出了在执行指令过程中可能会抛出的异常和错误。除开在指令描述中明确列出的异常和错误，另外还存在一组错误，它们可以在任何时候、作为任何指令的执行结果被抛出。这些错误是VirtualMachineError的子类。下面列出了VirtualMachineError的四种子类，以及产生它们的相关环境：

- OutOfMemoryError：虚拟机耗尽了所有实际或虚拟的内存，而且垃圾收集器无法获取到足以使线程继续执行的空间。
- StackOverflowError：线程耗尽了栈空间所能提供的所有内存（通常是因为应用程序中无节制地使用递归而引起）。
- InternalError：虚拟机遇到了实现自身的bug，使其无法完全实现Java语言的语义。
- UnknownError：虚拟机遇到了一些错误，但无法通过抛出相应的异常或错误来报告实际情况。

aaload

从数组中装载引用<reference>数据类型。

操作码字节值：50(0x32)

指令格式：aaload

栈：

前：……，*arrayref*，*index*

后：……，*value*

描述：要执行aaload指令，Java虚拟机首先从操作数栈中弹出两个字长的值。值*arrayref*必须是一个指向引用数组的引用。*index*的数据类型必须为int类型。虚拟机从*arrayref*这个引用所指向的数组中取出序号为*index*的引用型*value*，并将其压入操作数栈。

如果*arrayref*引用的数组为null，Java虚拟机将会抛出NullPointerException异常。如果*arrayref*引用的数组不为null，但*index*并不是*arrayref*所指向数组的有效索引，虚拟机将抛出ArrayIndexOutOfBounds-Exception异常。

如果需要了解有关aaload指令的更多信息，请参考第15章。

aastore

将引用类型存入数组。

操作码字节值：83(0x53)

指令格式：aastore

栈：

前：……，*arrayref*，*index*，*value*

后：……

描述：要执行aastore指令，Java虚拟机首先从操作数栈中弹出三个字长度的值。*arrayref*的数据类型必须是一个指向浮点数数组的引用类型。*index*的数据类型必须为int类型。*value*必须为引用类型。赋给*value*的数据类型必须与*arrayref*所指向数组的元素数据类型相兼容。虚拟机将引用型*value*存入*arrayref*所指向数组的*index*位置。

如果*arrayref*引用的数组为null，Java虚拟机将会抛出NullPointerException异常。如果*arrayref*引用的数组不为null，但*index*并不是*arrayref*所指向数组的有效索引，虚拟机将抛出ArrayIndexOutOfBoundsException异常。如果*arrayref*和*index*均有效，但*value*所指向值的实际数据类型与*arrayref*所指向数组元素的实际数据类型不兼容，那么虚拟机将抛出ArrayStoreException异常。

如果需要了解有关aastore指令的更多信息，请参考第15章。

aconst_null

将null对象引用压入栈。

操作码字节值：1(0x1)

指令格式：aconst_null

栈：

前：……

后：……，*null*

描述：要执行aconst_null指令，Java虚拟机将一个null对象引用压入操作数栈（需要注意的是，Java虚拟机规范并没有为null指定任何实际的值，而是将其留给每一种实现的设计者自行决定）。

如果需要获取有关aconst_null指令的更多信息，请参考第10章。

aload

从局部变量中装载引用类型。

操作码字节值：25(0x19)

指令格式：aload，*index*

栈：

前：……

后：……，*value*

描述：作为一个对当前栈帧内局部变量的8位无符号索引，操作数*index*必须指向一个包含了引用的局

部变量字。要执行aload指令，Java虚拟机将*index*指向的局部变量中的引用压入操作数栈。

需要注意的是，aload指令前面可以使用wide指令，以允许16位无符号偏移量访问局部变量。

还需要注意的是，尽管aload指令可以用于从操作数栈中弹出一个returnAddress类型值，然后存入一个局部变量，但是aload指令不能用于将一个returnAddress类型值压入操作数栈。如果想更多地了解returnAddress类型值的使用，请参考第18章。

如果需要获取有关aload指令的更多信息，请参考第10章。

aload_0

从局部变量0中装载引用类型。

操作码字节值：42(0x2a)

指令格式：aload_0

栈：

前：……

后：……，*value*

描述：位于索引0处的局部变量字必须包含一个引用。要执行aload_0指令，Java虚拟机将局部变量字0中的引用型*value*压入操作数栈。

需要注意的是，尽管aload_0指令可以用于从操作数栈中弹出一个returnAddress类型值，然后存入一个局部变量，但是aload_0指令不能用于将一个returnAddress类型值压入操作数栈。如果需要更多地了解returnAddress类型值的使用，请参考第18章。

如果需要获取更多有关aload_0指令的信息，请参考第10章。

aload_1

从局部变量1中装载引用类型。

操作码字节值：43(0x2b)

指令格式：aload_1

栈：

前：……

后：……，*value*

描述：位于索引1处的局部变量字必须包含一个引用。要执行aload_1指令，Java虚拟机将局部变量字1中的引用型*value*压入操作数栈。

需要注意的是，尽管aload_1指令可以用于从操作数栈中弹出一个returnAddress类型值，然后存入一个局部变量，但是aload_1指令不能用于将一个returnAddress类型值压入操作数栈。如果需要获取更多有关returnAddress类型值用法的信息，请参考第18章。

如果需要获取更多有关aload_1指令的信息，请参考第10章。

aload_2

从局部变量2中装载引用类型。

操作码字节值：44(0x2c)

指令格式：aload_2

栈：

前：……

后：……，*value*

描述：位于索引2处的局部变量字必须包含一个引用。要执行aload_2指令，Java虚拟机将局部变量字2中的引用型*value*压入操作数栈。

需要注意的是，尽管aload_2指令可以用于从操作数栈中弹出一个returnAddress类型值，然后存入一个局部变量，但是aload_2指令不能用于将一个returnAddress类型值压入操作数栈。如果需要获取更多有关returnAddress类型值用法的信息，请参考第18章。

如果需要获取更多有关aload_2指令的信息，请参考第10章。

aload_3

从局部变量3中装载引用类型。

操作码字节值：45(0x2d)

指令格式：aload_3

栈：

前：……

后：……，*value*

描述：位于索引3处的局部变量字必须包含一个引用。要执行aload_3指令，Java虚拟机将局部变量字3中容纳的引用型*value*压入操作数栈。

需要注意的是，尽管aload_3指令可以用于从操作数栈中弹出一个returnAddress类型值，然后存入一个局部变量，但是aload_3指令不能用于将一个returnAddress类型值压入操作数栈。如果需要获取更多有关returnAddress类型值用法的信息，请参考第18章。

如果需要获取更多有关aload_3指令的信息，请参考第10章。

anewarray

分配数据成员为引用类型的新数组。

操作码字节值：189(0xbd)

指令格式：anewarray，*indexbyte1*，*indexbyte2*

栈：

前：……，*count*

后：*arrayref*

描述：*count*（操作数栈顶端的一个字长的数据）必须为int类型。要执行anewarray指令，Java虚拟机首先通过计算(*indexbyte1* << *8*)| *indexbyte2*，得到一个指向常量池的16位无符号索引。然后，虚拟机根据这个索引查找相应的常量池入口。该索引所指向的常量池入口必须为CONSTANT_Class_info入口。如果该入口不存在，那么虚拟机就解析入口。该入口的类型可以是类、接口或者数组。

如果解析操作成功，Java虚拟机将*count*弹出栈，并且在堆中创建一个长度为*count*、数据类型为CONSTANT_Class_info入口所指明的引用类型的数组。虚拟机将每一个数组元素初始化为默认初始值（null），然后将*arrayref*（新数组的引用）压入操作数栈。

执行这条指令的结果是，虚拟机在解析CONSTANT_Class_info入口的过程中，有可能会抛出第8章中所列出的任何连接错误。如果解析成功，但*count*小于0，虚拟机将会抛出NegativeArraySizeException异常。

如果需要获取更多有关anewarray指令的信息，请参考第15章。

areturn

从方法中返回引用类型。

操作码字节值：176(0xb0)

指令格式：areturn

栈：

前：……，*objectref*

后：[空]

描述：方法的返回类型必须为引用类型。*objectref*（操作数栈顶的字）的数据类型必须为引用，而且引用类型必须与返回方法的描述符所代表的类型相兼容。要执行areturn指令，Java虚拟机从当前栈帧内的操作数栈中弹出*objectref*，然后将其压入调用方法栈帧内的操作数栈。虚拟机将抛弃所有返回方法栈帧中仍然可能存在的值。如果返回方法是同步的，那么该方法被调用时所获得的监视器将被释放，调用方法的栈帧被设为当前值，虚拟机继续执行调用方法中的语句。

当方法被调用的时候，如果当前方法是同步的，而且当前线程不持有已被获取和重入的监视器，虚拟机将会抛出IllegalMonitorStateException异常。否则，如果虚拟机实现强制执行monitorexit指令中所描述的结构化锁定的规则，虚拟机将会抛出IllegalMonitorStateException异常。

如果需要获取更多有关监视器的信息，请参考第20章。如果需要获取更多有关areturn指令的信息，请参考第19章。

arraylength

获取数组长度。

操作码字节值：190(0xbe)

指令格式：arraylength

栈：

前：……，*arrayref*

后：……，*length*

描述：*arrayref*（操作数栈顶的字）必须为指向数组的引用类型。要执行arraylength指令，Java虚拟机从栈中弹出*arrayref*，然后将*arrayref*所指向数组的长度压入栈。

如果*arrayref*为null，虚拟机将会抛出NullPointerException异常。

如果需要了解更多有关arraylength指令的信息，请参考第15章。

astore

将引用类型和returnAddress类型值存入局部变量。

操作码字节值：58(0x3a)

指令格式：astore，*index*

栈：

前：……，*value*

后：……

描述：操作数*index*必须指明一个指向当前栈帧内局部变量的有效的8位无符号索引。操作数栈顶部的值*value*必须为引用类型或者returnAddress类型。要执行astore指令，Java虚拟机从操作数栈顶部弹出一个引用类型或者returnAddress类型值（*value*），然后将该值存入由*index*指定的局部变量。

需要注意的是，astore指令前面可以使用wide指令，以允许将*value*存入16位无符号偏移量指定的局部变量中。

如果需要了解更多有关astore指令的信息，请参考第10章。

astore_0

将引用类型或者returnAddress类型值存入局部变量0。

操作码字节值：75(0x4b)

指令格式：astore_0

栈：

前：……，*value*

后：……

描述：索引0必须是一个指向当前栈帧内局部变量的有效索引。操作数栈顶部的字*value*必须为引用类型或者returnAddress类型。要执行astore_0指令，Java虚拟机从操作数栈顶部弹出一个引用类型或者returnAddress类型值（*value*），然后将该值存入由索引0指定的局部变量字中。

如果需要了解更多有关astore_0指令的信息，请参考第10章。

astore_1

将引用类型或者returnAddress类型值存入局部变量1。

操作码字节值：76(0x4c)

指令格式：astore_1

栈：

前：……，*value*

后：……

描述：索引1必须是一个指向当前栈帧内局部变量的有效索引。操作数栈顶部的字*value*必须为引用类型或者returnAddress类型。要执行astore_1指令，Java虚拟机从操作数栈顶部弹出一个引用类型或者returnAddress类型值（*vlaue*），然后将该值存入由索引1指定的局部变量字中。

如果需要了解更多有关astore_1指令的信息，请参考第10章。

astore_2

将引用类型或者returnAddress类型值存入局部变量2。

操作码字节值：77(0x4d)

指令格式：astore_2

栈：

前：……，*value*

后：……

描述：索引2必须是一个指向当前栈帧内局部变量的有效索引。操作数栈顶部的字*value*必须为引用类型或者returnAddress类型。要执行astore_2指令，Java虚拟机从操作数栈顶部弹出一个引用类型或者returnAddress类型值（*vlaue*），然后将该值存入由索引2指定的局部变量字中。

如果需要了解更多有关astore_2指令的信息，请参考第10章。

astore_3

将引用类型或者returnAddress类型值存入局部变量3。

操作码字节值：78(0x4e)

指令格式：astore_3

栈：

前：……，*value*

后：……

描述：索引3必须是一个指向当前栈帧内局部变量的有效索引。操作数栈顶部的字*value*必须为引用类型或者returnAddress类型。要执行astore_3指令，Java虚拟机从操作数栈顶部弹出一个引用类型或者returnAddress类型值（*vlaue*），然后将该值存入由索引3指定的局部变量字中。

如果需要了解更多有关astore_3指令的信息，请参考第10章。

athrow

抛出异常或者错误。

操作码字节值：191(0xbf)

指令格式：athrow

栈：

前：……，*objectref*

后：*objectref*

需要注意的是，“前”栈快照部分显示了属于方法（方法中包含将要执行的athrow指令）的栈帧内操作数栈的状态。如果catch子句存在的话，“后”栈快照部分显示了属于方法（方法中包含catch子句）的栈帧内操作数栈的状态。如果catch子句不存在，线程将退出，对于此线程，不会再有任何操作数栈存在。

描述：*objectref*（操作数栈顶端的字）必须为指向类java.lang.Throwable的实例的引用类型，或者是指

向类java.lang.Throwable子类实例的引用类型。要执行athrow指令，Java虚拟机从操作数栈中弹出*objectref*。虚拟机通过搜索当前方法的异常表，找到最近的catch子句（该子句要么捕获由*objectref*指向的可抛出对象的类，要么捕获可抛出对象类的子类），以此来“抛出”异常。如果当前方法的异常表中包含匹配项，虚拟机取出处理程序的地址，从匹配异常表项处跳转到处理程序的地址。虚拟机将操作数栈中所有的内容弹出，再压入*objectref*，将程序计数器设为处理程序的地址，然后继续执行。如果当前方法的异常表中没有一个相匹配的catch子句，虚拟机就将当前方法的整个栈帧弹出栈，在上层的方法中重新抛出异常。如果执行完毕的方法是同步的，当调用方法时，会释放（或者退出）所获取（或者重入）的监视器。这个过程一直重复，直到找到匹配的catch子句或者当前线程的调用栈以及所有方法的栈帧都被弹出。如果在这个过程中没有找到catch子句，当前线程将会退出。

如果*objectref*的值为null，虚拟机将会抛出NullPointerException异常。

如果当前方法为同步的，而且当调用该方法时，当前线程不持有已被获取或者重入的监视器，虚拟机将会抛出IllegalMonitorStateException异常。否则，如果虚拟机实现强制执行结构化锁定规则（在monitorexit指令中有相关描述），而且执行该指令时违反了规则1，虚拟机将抛出IllegalMonitorStateException异常。

如果需要了解更多有关athrow指令的信息，请参考第17章。

baload

从数组中读取byte或者boolean类型的数据。

操作码字节值：51(0x33)

指令格式：baload

栈：

前：……，*arrayref*，*index*

后：……，*value*

描述：要执行baload指令，Java虚拟机首先从操作数栈中弹出两个字长的数据（*arrayref*和*index*）。*arrayref*必须为指向byte或者boolean数组的引用类型。*index*必须为int类型。虚拟机从*arrayref*所指向的数组获取由*index*指定的byte或者boolean类型的值（*value*），并将它带符号扩展为int类型的值，然后将其压入操作数栈。

如果*arrayref*为null，虚拟机将会抛出NullPointerException异常。如果*arrayref*不为null，但*index*不是*arrayref*数组的有效索引，虚拟机将会抛出ArrayIndexOutOfBoundsException异常。

如果需要了解更多有关baload指令的信息，请参考第15章。

bastore

将byte或者boolean类型的数据存入数组。

操作码字节值：84(0x54)

指令格式：bastore

栈：

前：……，*arrayref*，*index*，*value*

后：……

描述：要执行bastore指令，Java虚拟机首先从操作数栈中弹出三个字长的数据（*arrayref*，*index*，*value*）。*arrayref*的数据类型必须为指向byte或者boolean数组的引用类型。*index*和*value*必须为int类型。虚拟机将int类型值*value*截短为byte类型值，然后将其存入*index*所指定位置的*arrayref*指向的数组。

如果*arrayref*为null，虚拟机将会抛出NullPointerException异常。如果*arrayref*不为null，但*index*不是*arrayref*数组的有效索引，虚拟机将会抛出ArrayIndexOutOfBoundsException异常。

如果需要了解更多有关bastore指令的信息，请参考第15章。

bipush

将一个8位带符号整数压入栈。

操作码字节值：16(0x10)

指令格式：bipush，*byte*

栈：

前：……

后：……，*value*

描述：要执行bipush指令，Java虚拟机首先对操作数*byte*进行符号扩展，从8位带符号整数扩展为int类型的数。然后，虚拟机将得到的int类型值（*value*）压入操作数栈。

如果需要了解更多有关bipush指令的信息，请参考第10章。

caload

从栈中装载char类型的数据。

操作码字节值：52(0x34)

指令格式：caload

栈：

前：……，*arrayref*，*index*

后：……，*value*

描述：要执行caload指令，Java虚拟机首先从操作数栈中弹出两个字长的数据（*arrayref*，*index*）。*arrayref*的数据类型必须为指向char数组的引用类型。*index*必须为int类型。虚拟机从*arrayref*所指向的数组获取*index*所指定的char类型的值（*value*），并将其0扩展为int类型的值，然后将其压入操作数栈。

如果*arrayref*引用的数组为null，虚拟机将会抛出NullPointerException异常。如果*arrayref*引用的数组不为null，但*index*不是*arrayref*数组的有效索引，虚拟机将会抛出ArrayIndexOutOf BoundsException异常。

如果需要了解更多有关caload指令的信息，请参考第15章。

castore

将char类型的数据存入数组。

操作码字节值：85(0x55)

指令格式：castore

栈：

前：……，*arrayref*，*index*，*value*

后：……

描述：要执行castore指令，Java虚拟机首先从操作数栈中弹出三个字长的数据（*arrayref*，*index*，*value*）。*arrayref*的数据类型必须为指向char数组的引用类型。*index*和*value*必须为int类型。虚拟机将int类型值*value*截短为char类型值，然后将其存入*index*所指定位置的*arrayref*指向的数组。

如果*arrayref*引用的数组为null，虚拟机将会抛出NullPointerException异常。如果*arrayref*引用的数组不为null，但*index*不是*arrayref*数组的有效索引，虚拟机将会抛出ArrayIndexOutOf BoundsException异常。

如果需要了解更多有关castore指令的信息，请参考第15章。

checkcast

确定对象为所给定的类型。

操作码字节值：192(0xc0)

指令格式：checkcast，*indexbyte1*，*indexbyte2*

栈：

前：……，*objectref*

后：……，*objectref*

描述：位于栈顶部的*objectref*必须为引用类型。要执行checkcast指令，Java虚拟机首先通过计算(*indexbyte1* << 8)| *indexbyte2*，得到一个指向常量池的16位无符号索引。然后，虚拟机根据这个索引查找相应的常量池入口。这个索引所指向的常量池入口必须为CONSTANT_Class_info入口。如果该入口不存在，那么虚拟机就解析入口。该入口的类型可以是类、接口或者数组。如果*objectref*的值为null，或者*objectref*能够转换为被解析的类型，栈将会维持原状。否则，虚拟机将会抛出ClassCastException异常。

为了弄清*objectref*所指的对象是否能够转换为被解析的类型，虚拟机首先确定对象是类的实例还是数组（此对象不可能为接口实例，因为接口不能被实例化）。如果对象为类实例，且被解析的类型是类而不是接口，如果对象所属的类是被解析的类或其子类，此对象就能够转换成为被解析的类。另外，如果该对象为类实例，被解析的类型为接口而不是类，如果此对象的类实现了被解析的接口，那么此对象就能够转换为被解析的接口。否则，对象为数组。如果被解析的类型是类，那么该类必须为java.lang.Object。如果被解析的类型是基本数据类型的数组，那么此对象也必须是基本数据类型的数组。如果被解析的类型是一个数据成员为引用类型的数组，那么此对象必须是一个数据成员为同样的引用类型的数组，这样，此对象才能够被转换为被解析的数组的类型（需要注意的是，checkcast并不会检查数组的维数，它只会检查数组数据成员的类型）。

执行这条指令的结果是，虚拟机在解析CONSTANT_Class_info入口的过程中，有可能会抛出第8章中所列出的任何连接错误。如果解析成功，但被解析的数据类型无法相互转换，虚拟机将会抛出ClassCastException异常。

如果需要获取更多有关checkcast指令的信息，请参考第15章。

d2f

把double类型的数据转换为float类型。

操作码字节值：144(0x90)

指令格式：d2f

栈：

前：……，*value.word1*，*value.word2*

后：……，*result*

描述：位于栈顶部的两个字长的数据必须为double类型。要执行d2f指令，Java虚拟机从操作数栈弹出double类型值（*value*），将double类型转换为float类型，然后把float结果类型压入栈。

要将double类型值（*value*）转换成float，Java虚拟机首先检查弹出值（*value*）是否为NaN（不为数字）。如果值为NaN，那么float结果值（*value*）也必须为NaN。如果值不为NaN，那么当double类型的输入值太小，无法用float类型表示时，float结果值为与弹出值同符号的0；当double类型的输入值太大，无法用float类型表示时，float结果值为与弹出值同符号的无穷大。此外，虚拟机将按照IEEE 754所规定的四舍五入（round-to-nearest）模式将double类型的值（*value*）转换为float类型的零。

需要注意的是，这条指令执行的是缩窄的基本类型转换（narrowing primitive conversion）。因为并不是所有double类型值都能准确无误地转换为float类型，转换有可能导致数值和精度的丢失。如果按照FP-strict原则，指令的结果必须在float值集中。如果不按照FP-strict原则，那么结果可以在float-extended-exponent集中。

如果需要获取更多有关d2f指令的信息，请参考第11章。

d2i

把double类型的数据转换为int类型。

操作码字节值：142(0x8e)

指令格式：d2i

栈：

前：……，*value.word1*，*value.word2*

后：……，*result*

描述：位于栈顶部的两个字长的数据必须为double类型。要执行d2i指令，Java虚拟机首先将double类型值（*value*）弹出栈，将double类型值转换为int类型值，再将转换所得int类型结果值（*result*）压入栈。

为了将double类型值转换为int类型，Java虚拟机检查弹出值（*value*）是否为NaN（不为数字）。如果弹出值为NaN，那么int类型结果值必须为0。如果弹出值不为NaN，那么当double类型值不为正无穷或者负无穷时，虚拟机按照IEEE 754规定的“取整”（round-toward-zero）模式将double类型值取整，如果所得整数能够使用int类型表示，结果即为此值。如果结果超出int数据类型的表示范围，则分两种情况：如果弹出值为正，结果值即为int类型所能表示的最大正整数；如果弹出值为负，结果值即为int类型所能表示的最小负整数。

需要注意的是，这条指令执行的是缩窄的基本类型转换。因为并不是所有double类型值都能准确无误地转换为int类型，转换有可能导致数值和精度的丢失。

如果需要了解更多有关d2i指令的信息，请参考第11章。

d2l

把double类型的数据转换为long类型。

操作码字节值：143(0x8f)

指令格式：d2l

栈：

前：……，*value.word1*，*value.word2*

后：……，*result.word1*，*result.word2*

描述：位于栈顶部的两个字长的数据必须为double类型。要执行d2l指令，Java虚拟机首先将double类型值（*value*）弹出栈，将double类型值转换为long类型值，再将转换所得long类型结果值（*result*）压入栈。

为了将double类型值转换为long类型，Java虚拟机首先检查弹出值是否为NaN（不为数字）。如果弹出值为NaN，那么long类型结果值必须为0。如果弹出值不为NaN，那么当double类型值不为正无穷或者负无穷，虚拟机按照IEEE 754规定的“取整”模式将double类型值取整，如果所得整数能够使用long类型表示，结果即为此值。如果结果超出long数据类型的表示范围，则分两种情况：如果弹出值为正，结果值即为long类型所能表示的最大正整数；如果弹出值为负，结果值即为long类型所能表示的最小负整数。

需要注意的是，这条指令执行的是缩窄的基本类型转换。因为并不是所有double类型值都能准确无误地转换为long类型，转换有可能导致数值和精度的丢失。

如果需要了解更多有关d2l指令的信息，请参考第11章。

dadd

执行double类型的加法。

操作码字节值：99(0x63)

指令格式：dadd

栈：

前：……，*value1.word1*，*value1.word2*，*value2.word1*，*value2.word2*

后：……，*result.word1*，*result.word2*

描述：*value1*和*value2*（位于栈顶部四个字长的数据）必须为double类型。要执行dadd指令，Java虚拟机首先将*value1*和*value2*弹出栈，相加，再将所得double类型结果（*result*）压入栈。执行dadd指令所得到的结果由IEEE 754中的浮点算法规则决定。

如果需要获取更多有关dadd指令的信息，请参考第14章。

daload

从数组中装载double类型的数据。

操作码字节值：49(0x31)

指令格式：daload

栈：

前：……，*arrayref*，*index*

后：……，*value.word1*，*value.word2*

描述：要执行daload指令，Java虚拟机首先从操作数栈中弹出两个字长的数据（*arrayref*和*index*）。*arrayref*必须为指向double类型数组的引用。*index*必须为int类型。虚拟机从*arrayref*所指向的数组获取由

*index*所指定的double类型的值（*value*），然后将值压入操作数栈。

如果*arrayref*引用的数组为null，虚拟机将会抛出NullPointerException异常。如果*arrayref*引用的数组不为null，但*index*不是*arrayref*数组的有效索引，虚拟机将会抛出ArrayIndexOutOfBoundsException异常。

如果需要了解更多有关daload指令的信息，请参考第15章。

dastore

将double类型的数据存入数组中。

操作码字节值：82(0x52)

指令格式：dastore

栈：

前：……，*arrayref*，*index*，*value.word1*，*value.word2*

后：……

描述：要执行dastore指令，Java虚拟机首先从操作数栈中弹出四个字长的数据（*arrayref*，*index*，*value.word1*，*value.word2*）。*arrayref*必须为指向double数组的引用。*index*的数据类型必须为int，*value*必须为double类型。虚拟机将double类型的值（*value*）存入*index*所指定位置的*arrayref*数组。

如果*arrayref*引用的数组为null，虚拟机将会抛出NullPointerException异常。如果*arrayref*引用的数组不为null，但*index*不是*arrayref*数组的有效索引，虚拟机将会抛出ArrayIndexOutOfBoundsException异常。

如果需要了解更多有关dastore指令的信息，请参考第15章。

dcmpg

比较double类型值（当遇到NaN时，返回1）。

操作码字节值：152(0x98)

指令格式：dcmpg

栈：

前：……，*value1.word1*，*value1.word2*，*value2.word1*，*value2.word2*

后：……，*result*

描述：*value1*和*value2*为两个double类型值，它们在栈顶部占据四个字长的空间。要执行dcmpg指令，Java虚拟机首先从操作数栈中弹出*value1*和*value2*，然后对它们进行比较。如果*value1*与*value2*相等，虚拟机向操作数栈中压入int结果（*result*）值0；如果*value1*大于*value2*，虚拟机向操作数栈中压入int结果（*result*）值1；如果*value1*小于*value2*，虚拟机向操作数栈中压入int结果（*result*）值-1。如果*value1*和*value2*中至少有一个为NaN（不是数字），虚拟机将向操作数栈中压入int结果（*result*）值1。

执行dcmpg指令所得到的结果由IEEE 754中的浮点算法规则决定。需要注意的是，指令dcmpg与dcmpl的惟一区别在于对NaN的处理。

如果需要了解更多有关dcmpg指令的信息，请参考第16章。

dcmpl

比较double类型值（当遇到NaN时，返回-1）。

操作码字节值：151(0x97)

指令格式：dcmpl

栈：

前：……，*value1.word1*，*value1.word2*，*value2.word1*，*value2.word2*

后：……，*result*

描述：*value1*和*value2*为两个double类型值，它们在栈顶部占据四个字长的空间。要执行dcmpg指令，Java虚拟机首先从操作数栈中弹出*value1*和*value2*，然后对它们进行比较。如果*value1*与*value2*相等，虚拟机向操作数栈中压入int结果（*result*）值0；如果*value1*大于*value2*，虚拟机向操作数栈中压入int结果（*result*）值1；如果*value1*小于*value2*，虚拟机向操作数栈中压入int结果（*result*）值-1。如果*value1*和*value2*中至少有一个为NaN（不是数字），虚拟机将向操作数栈中压入int结果（*result*）值-1。

执行dcmpl指令所得到的结果由IEEE 754中的浮点算法规则决定。需要注意的是，指令dcmpg与dcmpl的惟一区别在于对NaN的处理。如果需要获取更多有关dcmpl指令的信息，请参考第16章。

dconst_0

将double类型常量0.0压入栈。

操作码字节值：14(0xe)

指令格式：dconst_0

栈：

前：……

后：……，*<0.0>-word1*，*<0.0>-word2*

描述：要执行dconst_0指令，Java虚拟机将double类型常量0.0压入操作数栈。

如果需要获取更多有关dconst_0指令的信息，请参考第10章。

dconst_1

将double类型常量1.0压入栈。

操作码字节值：15(0xf)

指令格式：dconst_1

栈：

前：……

后：……，*<1.0>-word1*，*<1.0>-word2*

描述：要执行dconst_1指令，Java虚拟机将double类型常量1.0压入操作数栈。

如果需要获取更多有关dconst_1指令的信息，请参考第10章。

ddiv

执行double类型的除法。

操作码字节值：111(0x6f)

指令格式：ddiv

栈：

前：……，*value1.word1*，*value1.word2*，*value2.word1*，*value2.word2*

后：……，*result.word1*，*result.word2*

描述：*value1*和*value2*（位于操作数栈顶部四个字长的数据）必须为double类型。要执行ddiv指令，Java虚拟机首先将*value1*和*value2*弹出栈，相除（*value1*/*value2*），再将所得double类型结果（*result*）压入栈。执行ddiv所得到的结果由IEEE 754中的浮点算法规则决定。

如果需要获取更多有关ddiv指令的信息，请参考第14章。

dload

从局部变量中装载double类型值。

操作码字节值：24(0x18)

指令格式：dload，*index*

栈：

前：……

后：……，*value.word1*，*value.word2*

描述：作为一个对当前栈帧内局部变量的8位无符号索引，操作数*index*必须指向包含double类型数据的两个连续局部变量字的第一个。要执行dload指令，Java虚拟机将包含在*index*和*index+1*所指向的两个连续局部变量字中的double类型值压入操作数栈。

需要注意的是，dload指令前面可以使用wide指令，以允许访问一个由16位无符号偏移量所表示的局部变量。

如果需要获取有关dload指令的更多信息，请参考第10章。

dload_0

从局部变量0中装载double类型值。

操作码字节值：38(0x26)

指令格式：dload_0

栈：

前：……

后：……，*value.word1*，*value.word2*

描述：在索引0和1位置的两个连续的局部变量中，必须包含一个double类型值。要执行dload_0指令，Java虚拟机将局部变量0和局部变量1中存储的double类型值（*value*）压入操作数栈。

如果需要获取有关dload_0指令的更多信息，请参考第10章。

dload_1

从局部变量1中装载double类型值。

操作码字节值：39(0x27)

指令格式：dload_1

栈：

前：……

后：……，*value.word1*，*value.word2*

描述：在索引1和2位置的两个连续的局部变量中，必须包含一个double类型值。要执行dload_1指令，Java虚拟机将局部变量1和局部变量2中存储的double类型值（*value*）压入操作数栈。

如果需要获取有关dload_1指令的更多信息，请参考第10章。

dload_2

从局部变量2中装载double类型值。

操作码字节值：40(0x28)

指令格式：dload_2

栈：

前：……

后：……，*value.word1*，*value.word2*

描述：在索引2和3位置的两个连续的局部变量中，必须包含一个double类型值。要执行dload_2指令，Java虚拟机将局部变量2和局部变量3中存储的double类型值（*value*）压入操作数栈。

如果需要获取有关dload_2指令的更多信息，请参考第10章。

dload_3

从局部变量3中装载double类型值。

操作码字节值：41(0x29)

指令格式：dload_3

栈：

前：……

后：……，*value.word1*，*value.word2*

描述：在索引3和4位置的两个连续的局部变量中，必须包含一个double类型值。要执行dload_3指令，Java虚拟机将局部变量3和局部变量4中存储的double类型值（*value*）压入操作数栈。

如果需要获取有关dload_3指令的更多信息，请参考第10章。

dmul

执行double类型的乘法。

操作码字节值：107(0x6b)

指令格式：dmul

栈：

前：……，*value1.word1*，*value1.word2*，*value2.word1*，*value2.word2*

后：……，*result.word1*，*result.word2*

描述：*value1*和*value2*（位于操作数栈顶部四个字长的数据）必须为double类型。要执行dmul指令，Java虚拟机首先将*value1*和*value2*弹出栈，相乘，再将所得double类型结果（*result*）压入栈。执行dmul所

得到的结果由IEEE 754中的浮点算法规则决定。

如果需要获取更多有关dmul指令的信息，请参考第14章。

dneg

将一个double类型的数据取反。

操作码字节值：119(0x77)

指令格式：dneg

栈：

前：……，*value.word1*，*value.word2*

后：……，*result.word1*，*result.word2*

描述：操作数栈顶部两个字长的数据必须为double类型。要执行dneg指令，Java虚拟机首先将*value*弹出栈，取反，再将所得double类型结果（*result*）压入栈。执行dneg所得到的结果由IEEE 754中的浮点算法规则决定。

需要注意的是，使用dneg指令所得到的结果值并不总是与使用dsub指令来进行“0减*value*”操作所得的结果相等。IEEE 754浮点数的范围中包括两个0——一个正0和一个负0。当*value*为+0.0时，执行dneg指令的结果为：-0.0；而当进行“+0.0减去+0.0”操作时，所得到的结果还是+0.0。

如果需要获取更多有关dneg指令的信息，请参考第14章。

drem

计算double类型除法的余数。

操作码字节值：115(0x73)

指令格式：drem

栈：

前：……，*value1.word1*，*value1.word2*，*value2.word1*，*value2.word2*

后：……，*result.word1*，*result.word2*

描述：*value1*和*value2*（位于操作数栈顶部四个字长的数据）必须为double类型。要执行drem指令，Java虚拟机首先将*value1*和*value2*弹出栈，计算余数，再将所得double类型结果（*result*）压入栈。取余操作等同于*value1* - (*value1*/*value2*) * *value2*，这里的*value1* / *value2*用的是截短除法，而不是IEEE 754所要求的取整除法。

drem的行为可以与C语言库中的fmod()函数相比较。依照IEEE 754浮点标准进行两个double类型之间的取余操作，可以使用java.lang.Math类中的IEEERemainder()方法进行。

如果需要获取更多有关drem指令的信息，请参考第14章。

dreturn

从方法中返回double类型的数据。

操作码字节值：175(0xaf)

指令格式：dreturn

栈：

前：……，*value.word1*，*value.word2*

后：[空]

描述：方法返回值必须为double类型。操作数栈顶部的两个字长的数据类型必须为double。要执行dreturn指令，Java虚拟机从当前栈帧内的操作数栈中弹出double类型值（*value*），然后将其压入调用方法栈帧内的操作数栈。虚拟机将抛弃所有返回方法栈帧中仍然可能存在的值。如果返回方法是同步的，那么该方法被调用时所获得的监视器将被释放，调用方法栈帧被设为当前值，虚拟机继续执行调用方法中的语句。

如果当方法被调用的时候，当前方法是同步的，而且当前线程不持有已被获取和重入的监视器，虚拟机将会抛出IllegalMonitorStateException异常。否则，如果虚拟机实现强制执行在monitorexit指令中结构化锁定的规则，虚拟机将会抛出IllegalMonitorStateException异常。

如果需要获取更多有关监视器的信息，请参考第20章。如果需要获取更多有关dreturn指令的信息，请参考第19章。

dstore

将double类型值存入局部变量中。

操作码字节值：57(0x39)

指令格式：dstore，*index*

栈：

前：……，*value.word1*，*value.word2*

后：……

描述：操作数*index*必须指明一个指向当前栈帧内局部变量的有效的8位无符号索引。*value*（位于操作数栈顶部的两个字长的数据）必须为double类型。要执行dstore指令，Java虚拟机从操作数栈顶部弹出double类型值（*value*），然后将该值存入由索引*index*和*index+1*所指定的两个连续局部变量字中。

需要注意的是，dstore指令前面可以使用wide指令，以使弹出的值（*value*）存到16位无符号偏移量所指定的局部变量中。

如果需要获取更多有关dstore指令的信息，请参考第10章。

dstore_0

将double类型值存入局部变量0中。

操作码字节值：71(0x47)

指令格式：dstore_0

栈：

前：……，*value.word1*，*value.word2*

后：……

描述：索引0和1必须是一个指向当前栈帧内局部变量的有效索引。操作数栈顶部的值必须为double类型。要执行dstore_0指令，Java虚拟机从操作数栈顶部弹出一个double类型值（*value*），然后将该值存入由索引0和1所指定的两个连续局部变量字中。

如果需要获取更多有关dstore_0指令的信息，请参考第10章。

dstore_1

将double类型值存入局部变量1中。

操作码字节值：72(0x48)

指令格式：dstore_1

栈：

前：……，*value.word1*，*value.word2*

后：……

描述：索引1和2必须是一个指向当前栈帧内局部变量的有效索引。操作数栈顶部的值必须为double类型。要执行dstore_1指令，Java虚拟机从操作数栈顶部弹出一个double类型值（*value*），然后将该值存入由索引1和2所指定的两个连续局部变量字中。

如果需要获取更多有关dstore_1指令的信息，请参考第10章。

dstore_2

将double类型值存入局部变量2中。

操作码字节值：73(0x49)

指令格式：dstore_2

栈：

前：……，*value.word1*，*value.word2*

后：……

描述：索引2和3必须是一个指向当前栈帧内局部变量的有效索引。操作数栈顶部的值必须为double类型。要执行dstore_2指令，Java虚拟机从操作数栈顶部弹出一个double类型值（*value*），然后将该值存入由索引2和3所指定的两个连续局部变量字中。

如果需要获取更多有关dstore_2指令的信息，请参考第10章。

dstore_3

将double类型值存入局部变量3中。

操作码字节值：74(0x4a)

指令格式：dstore_3

栈：

前：……，*value.word1*，*value.word2*

后：……

描述：索引3和4必须是一个指向当前栈帧内局部变量的有效索引。操作数栈顶部的值必须为double类型。要执行dstore_3指令，Java虚拟机从操作数栈顶部弹出一个double类型值（*value*），然后将该值存入由索引3和4所指定的两个连续局部变量字中。

如果需要获取更多有关dstore_3指令的信息，请参考第10章。

dsub

执行double数据类型的减法。

操作码字节值：103(0x67)

指令格式：dsub

栈：

前：……，*value1.word1*，*value1.word2*，*value2.word1*，*value2.word2*

后：……，*result.word1*，*result.word2*

描述：*value1*和*value2*（位于操作数栈顶部四个字长的数据）必须为double类型。要执行dsub指令，Java虚拟机首先将*value1*和*value2*弹出栈，用value1减去*value2*（*value1* –*value2*），再将所得double类型结果（*result*）压入栈。执行dsub所得到的结果由IEEE 754中的浮点算法规则决定。

如果需要获取更多有关dsub指令的信息，请参考第14章。

dup

复制栈顶部一个字长的内容。

操作码字节值：89(0x59)

指令格式：dup

栈：

前：……，*word*

后：……，*word*，*word*

描述：要执行dup指令，Java虚拟机复制了操作数栈顶部一个字长的内容，然后再将复制内容压入栈。本条指令能够从操作数栈顶部复制任何单字长的值。但绝对不要使用它来复制操作数栈顶部任何两个字长值（long类型或者double类型）中的一个字长。

如果需要获取更多有关dup指令的信息，请参考第10章。

dup_x1

复制栈顶部一个字长的内容，然后将复制内容及原来弹出的两个字长的内容压入栈。

操作码字节值：90(0x5a)

指令格式：dup_x1

栈：

前：……，*word2*，*word1*

后：……，*word1*，*word2*，*word1*

描述：要执行dup_x1指令，Java虚拟机复制了操作数栈顶部一个字长的内容，然后再将复制内容及原来弹出的两个字长的内容压入栈。这里的*word1*和*word2*都必须为单字长的值。

如果需要获取更多有关dup_x1指令的信息，请参考第10章。

dup_x2

复制栈顶部一个字长的内容，然后将复制内容及原来弹出的三个字长的内容压入栈。

操作码字节值：91(0x5b)

指令格式：dup_x2

栈：

前：……，*word3*，*word2*，*word1*

后：……，*word1*，*word3*，*word2*，*word1*

描述：要执行dup_x2指令，Java虚拟机复制了操作数栈顶部一个字长的内容，然后再将复制内容及原来弹出的三个字长的内容压入栈。这里的*word1*必须为一个字长的值；*word2*和*word3*必须为一个字长的值或者共同组成一个长度为两个字长的值（long类型或者double类型）。

如果需要获取更多有关dup_x2指令的信息，请参考第10章。

dup2

复制栈顶部长度为两个字长的内容。

操作码字节值：92(0x5c)

指令格式：dup2

栈：

前：……，*word2*，*word1*

后：……，*word2*，*word1*，*word2*，*word1*

描述：要执行dup2指令，Java虚拟机复制了操作数栈顶部两个字长的内容，然后再将复制内容压入栈。本条指令能够从操作数栈顶部复制任何长度为两个字长的值或者任何两个单字长的值。但绝对不要使用它来同时复制操作数栈顶部长度为一个字长的值和一个长度为两个字长值（long类型或者double类型）中的一个字长。

如果需要获取更多有关dup2指令的信息，请参考第10章。

dup2_x1

复制栈顶部长度为两个字长的内容，然后将复制内容及原来弹出的三个字长的内容压入栈。

操作码字节值：93(0x5d)

指令格式：dup2_x1

栈：

前：……，*word3*，*word2*，*word1*

后：……，*word2*，*word1*，*word3*，*word2*，*word1*

描述：要执行dup2_x1指令，Java虚拟机复制了操作数栈顶部两个字长的内容，然后再将复制内容及原来弹出的三个字长的内容压入栈。这里的*word3*必须为一个字长的值；*word1*和*word2*必须为一个字长的值或者同时组成一个长度为两个字长的值（long类型或者double类型）。

如果需要获取更多有关dup2_x1指令的信息，请参考第10章。

dup2_x2

复制栈顶部长度为两个字长内容，然后再将复制内容及原来弹出的四个字长的内容压入栈。

操作码字节值：94(0x5e)

指令格式：dup2_x2

栈：

前：……，*word4*，*word3*，*word2*，*word1*

后：……，*word2*，*word1*，*word4*，*word3*，*word2*，*word1*

描述：要执行dup2_x2指令，Java虚拟机复制了操作数栈顶部两个字长的内容，然后再将复制内容及原来弹出的四个字长的内容压入栈。这里的*word3*必须为一个字长的值；*word1*和*word2*必须为一个字长的值或者同时组成一个长度为两个字长的值（long类型或者double类型）。同样，*word3*和*word4*必须为一个字长的值或者同时组成一个长度为两个字长的值。

如果需要获取更多有关dup2_x2指令的信息，请参考第10章。

f2d

把float类型的数据转换为double类型。

操作码字节值：141(0x8d)

指令格式：f2d

栈：

前：……，*value*

后：……，*result.word1*，*result.word2*

描述：*value*（位于操作数栈顶部的一个字长的数据）的数据类型必须为double。要执行f2d指令，Java虚拟机首先从操作数栈中弹出一个float类型值（*value*），然后将float类型转换成为double类型，最后将得到的double类型结果值（*result*）压入栈。

如果按照FP-strict原则，本条指令实际上执行的是放宽的基本类型转换（widening primitive conversion）。因为所有的float类型数值都可以由double类型描述。

不按照FP-strict原则的话，指令执行结果可以是double-extended-exponent集中的值。如果这样，结构就不需要在double值集中进行四舍五入匹配。但不幸的是，如果float值属于float-extended-exponent集而double类型的结果属于double集的话，四舍五入就是必要的。

如果需要获取更多有关f2d指令的信息，请参考第11章。

f2i

把float类型的数据转换为int类型。

操作码字节值：139(0x8b)

指令格式：f2i

栈：

前：……，*value*

后：……，*result*

描述：*value*（位于操作数栈顶部的一个字长的数据）必须为float类型。要执行f2i指令，Java虚拟机首先将float类型值（*value*）弹出栈，将float类型值转换为int类型值，再将转换所得int类型结果（*result*）压入栈。

为了将float类型值转换为int类型，Java虚拟机首先检查弹出值（*value*）是否为NaN（不为数字）。如果值为NaN，那么int类型结果值必须为0。如果值不为NaN，那么当float类型值不为正无穷或者负无穷时，虚拟机按照IEEE 754“取整”模式将float类型值取整，如果所得整数能够使用int类型准确描述，那么结果即为此值。如果结果超出int数据类型的表示范围，则分两种情况：如果弹出值为正，结果值即为int类型所能表示的最大正整数；如果弹出值为负，结果值即为int类型所能表示的最小负整数。

需要注意的是，这条指令执行的是缩窄的基本类型转换。因为并不是所有float类型值都能准确无误地转换为int类型，转换有可能导致数值和精度的丢失。

如果需要获取更多有关f2i指令的信息，请参考第11章。

f2l

把float类型的数据转换为long类型。

操作码字节值：140(0x8c)

指令格式：f2l

栈：

前：……，*value*

后：……，*result.word1*，*result.word2*

描述：*value*（位于操作数栈顶部的一个字长的数据）必须为float类型。要执行f2l指令，Java虚拟机首先将float类型值（*value*）弹出栈，将float类型值转换为long类型值，再将转换所得long类型结果值（*result*）压入栈。

为了将float类型值转换为long类型，Java虚拟机检查弹出值是否为NaN（不为数字）。如果弹出值为NaN，那么long类型结果值必须为0。如果弹出值不为NaN，那么当float类型值不为正无穷或者负无穷，虚拟机按照IEEE 754“取整”模式将float类型值取整，如果所得整数能够使用long类型准确描述，结果即为此值。如果结果超出long数据类型的表示范围，则分两种情况：如果弹出值为正，结果值即为long类型所能表示的最大正整数；如果弹出值为负，结果值即为long类型所能表示的最小负整数。

需要注意的是，这条指令执行的是缩窄的基本类型转换。因为并不是所有float类型值都能准确无误地转换为long类型，转换有可能导致数值和精度的丢失。

如果需要获取更多有关f2l指令的信息，请参考第11章。

fadd

执行float类型的加法。

操作码字节值：98(0x62)

指令格式：fadd

栈：

前：……，*value1*，*value2*

后：……，*result*

描述：*value1*和*value2*（位于操作数栈顶部两个字长的数据）必须为float类型。要执行fadd指令，Java虚拟机首先将*value1*和*value2*弹出栈，相加，再将所得float类型结果（*result*）压入栈。执行fadd所得到的结果由IEEE 754中的浮点算法规则决定。

如果需要获取更多有关fadd指令的信息，请参考第14章。

faload

从数组中装载float类型的数据。

操作码字节值：48(0x30)

指令格式：faload

栈：

前：……，*arrayref*，*index*

后：……，*value*

描述：要执行faload指令，Java虚拟机首先从操作数栈中弹出两个字（*arrayref*和*index*）。*arrayref*必须为指向float类型数组的引用。*index*必须为int类型。虚拟机从*arrayref*所指向的数组获取*index*所指定的float类型的值（*value*），然后将其压入操作数栈。

如果*arrayref*引用的数组为null，虚拟机将会抛出NullPointerException异常。另外，如果*index*不是*arrayref*数组的有效索引，虚拟机将会抛出ArrayIndexOutOfBoundsException异常。

如果需要获取更多有关faload指令的信息，请参考第15章。

fastore

将float类型的数据存入数组中。

操作码字节值：81(0x51)

指令格式：fastore

栈：

前：……，*arrayref*，*index*，*value*

后：……

描述：要执行fastore指令，Java虚拟机首先从操作数栈中弹出三个字（*arrayref*，*index*，*value*）。*arrayref*必须为指向float类型数组的引用。*index*的数据类型必须为int，*value*必须为float类型。虚拟机将float类型值*value*存入*index*所指定位置的*arrayref*指向的数组。

如果*arrayref*引用的数组为null，虚拟机将会抛出NullPointerException异常。如果*arrayref*引用的数组不为null，但*index*不是*arrayref*数组的有效索引，虚拟机将会抛出ArrayIndexOutOf BoundsException异常。

如果需要获取更多有关fastore指令的信息，请参考第15章。

fcmpg

对float类型值进行比较（当遇到NaN时，返回1）。

操作码字节值：150(0x96)

指令格式：fcmpg

栈：

前：……，*value1*，*value2*

后：……，*result*

描述： *value1*和*value2*为两个float类型值，它们在栈顶部占据两个字长的空间。要执行fcmpg指令，Java虚拟机首先从操作数栈中弹出*value1*和*value2*，然后对它们进行比较。如果*value1*与*value2*相等，虚拟机向操作数栈中压入一个int类型结果值（*result*）0；如果*value1*大于*value2*，虚拟机向操作数栈中压入一个int类型结果值1；如果*value1*小于*value2*，虚拟机向操作数栈中压入一个int类型结果值-1。如果*value1*和*value2*中至少有一个为NaN（不是数字），虚拟机将向操作数栈中压入int类型结果值1。

执行fcmpg指令所得到的结果由IEEE 754中的浮点算法规则决定。需要注意的是，指令fcmpg与fcmpl的惟一区别在于对NaN的处理。如果需要获取更多有关fcmpg指令的信息，请参考第16章。

fcmpl

对float类型值进行比较（当遇到NaN时，返回-1）。

操作码字节值：149(0x95)

指令格式：fcmpl

栈：

前：……，*value1*，*value2*

后：……，*result*

描述：*value1*和*value2*为两个float类型值，它们在栈顶部占据两个字长的空间。要执行fcmpl指令，Java虚拟机首先从操作数栈中弹出*value1*和*value2*，然后对它们进行比较。如果*value1*与*value2*相等，虚拟机向操作数栈中压入一个int类型结果值（*result*）0；如果*value1*大于*value2*，虚拟机向操作数栈中压入一个int类型结果值1；如果*value1*小于*value2*，虚拟机向操作数栈中压入一个int类型结果值-1。如果*value1*和*value2*中至少有一个为NaN（不是数字），虚拟机将向操作数栈中压入int类型结果值-1。

执行fcmpl指令所得到的结果由IEEE 754中的浮点算法规则决定。需要注意的是，指令fcmpl与fcmpg的惟一区别在于对NaN的处理。如果需要获取更多有关fcmpl指令的信息，请参考第16章。

fconst_0

将float类型常量0.0压入栈。

操作码字节值：11(0xb)

指令格式：fconst_0

栈：

前：……

后：……，*<0.0>*

描述：要执行fconst_0指令，Java虚拟机将float类型常量0.0压入操作数栈。

如果需要获取更多有关fconst_0指令的信息，请参考第10章。

fconst_1

将float类型常量1.0压入栈。

操作码字节值：12(0xc)

指令格式：fconst_1

栈：

前：……

后：……，*<1.0>*

描述：要执行fconst_1指令，Java虚拟机将float类型常量1.0压入操作数栈。

如果需要获取更多有关fconst_1指令的信息，请参考第10章。

fconst_2

将float类型常量2.0压入栈。

操作码字节值：13(0xd)

指令格式：fconst_2

栈：

前：……

后：……，*<2.0>*

描述：要执行fconst_2指令，Java虚拟机将float类型常量2.0压入操作数栈。

如果需要获取更多有关fconst_2指令的信息，请参考第10章。

fdiv

执行float类型的除法。

操作码字节值：110(0x6e)

指令格式：fdiv

栈：

前：……，*value1*，*value2*

后：……，*result*

描述：*value1*和*value2*（位于操作数栈顶部两个字长的数据）必须为float类型。要执行fdiv指令，Java虚拟机首先将*value1*和*value2*弹出栈，相除（*value1*/*value2*），再将所得float类型结果值（result）压入栈。执行fdiv所得到的结果由IEEE 754中的浮点算法规则决定。

如果需要获取更多有关fdiv指令的信息，请参考第14章。

fload

从局部变量中装载float类型值。

操作码字节值：23(0x17)

指令格式：fload，*index*

栈：

前：……

后：……，*value*

描述：作为一个对当前栈帧中局部变量的8位无符号索引，操作数*index*必须指向一个包含float类型数据的局部变量字。要执行fload指令，Java虚拟机将包含在*index*所指定的局部变量字中的float类型值压入操

作数栈。

需要注意的是，fload指令前面可以使用wide指令，这样就能够访问由16位无符号偏移量所指向的局部变量。

如果需要获取有关fload指令的更多信息，请参考第10章。

fload_0

从局部变量0中读出float类型值。

操作码字节值：34(0x22)

指令格式：fload_0

栈：

前：……

后：……，*value*

描述：在索引0位置的局部变量字中，必须包含一个float类型值。要执行fload_0指令，Java虚拟机将局部变量字0中包含的float类型值（*value*）压入操作数栈。

如果需要获取有关fload_0指令的更多信息，请参考第10章。

fload_1

从局部变量1中装载float类型值。

操作码字节值：35(0x23)

指令格式：fload_1

栈：

前：……

后：……，*value*

描述：在索引1位置的局部变量字中，必须包含一个float类型值。要执行fload_1指令，Java虚拟机将局部变量字1中包含的float类型（*value*）值压入操作数栈。

如果需要获取有关fload_1指令的更多信息，请参考第10章。

fload_2

从局部变量2中装载float类型值。

操作码字节值：36(0x24)

指令格式：fload_2

栈：

前：……

后：……，*value*

描述：在索引2位置的局部变量中，必须包含一个float类型值。要执行fload_2指令，Java虚拟机将局部变量字2中包含的float类型值（*value*）压入操作数栈。

如果需要获取有关fload_2指令的更多信息，请参考第10章。

fload_3

从局部变量3中装载float类型值。

操作码字节值：37(0x25)

指令格式：fload_3

栈：

前：……

后：……，*value*

描述：在索引3位置的局部变量字中，必须包含一个float类型值。要执行fload_3指令，Java虚拟机将局部变量3中包含的float类型值*value*压入操作数栈。

如果需要获取有关fload_3指令的更多信息，请参考第10章。

fmul

执行float类型的乘法。

操作码字节值：106(0x6a)

指令格式：fmul

栈：

前：……，*value1*，*value2*

后：……，*result*

描述：*value1*和*value2*（位于操作数栈顶部两个字长的数据）必须为float类型。要执行fmul指令，Java虚拟机首先将*value1*和*value2*弹出栈，相乘，再将所得float类型结果值（*result*）压入栈。执行fmul所得到的结果由IEEE 754中的浮点算法规则决定。

如果需要获取更多有关fmul指令的信息，请参考第14章。

fneg

将一个float类型值取反。

操作码字节值：118(0x76)

指令格式：fneg

栈：

前：……，*value*

后：……，*result*

描述：*value*（位于栈顶部一个字长的数据）必须为float类型。要执行fneg指令，Java虚拟机首先将*value*弹出栈，取反，再将所得float类型结果值（*result*）压入栈。执行fneg所得到的结果由IEEE 754中的浮点算法规则决定。

需要注意的是，使用fneg指令所得到的结果并不总是与使用fsub指令来进行“0减*value*”操作所得的结果相等。IEEE 754浮点数的范围中包括两个0——一个正0和一个负0。当*value*为+0.0时，执行fneg指令的结果为-0.0；而当进行+0.0减去+0.0操作时，所得到的结果还是+0.0。

如果需要获取更多有关fneg指令的信息，请参考第14章。

frem

计算float类型除法的余数。

操作码字节值：114(0x72)

指令格式：frem

栈：

前：……，*value1*，*value2*

后：……，*result*

描述：*value1*和*value2*（位于操作数栈顶部两个字长的数据）必须为float类型。要执行frem指令，Java虚拟机首先将*value1*和*value2*弹出栈，计算余数，再将所得float类型结果（*result*）压入栈。取余操作等同于*value1* - (*value1* / *value2*) * *value2*，这里的*value1* / *value2*用的是截短除法，而不是IEEE 754所要求的取整除法。

frem的行为可以与C语言库中的fmod()相比较。依照IEEE 754浮点标准进行两个float类型之间的取余操作可以使用java.lang.Math类中的IEEERemainder()方法进行。

如果需要获取更多有关frem指令的信息，请参考第14章。

freturn

从方法中返回float类型的数据。

操作码字节值：174(0xae)

指令格式：freturn

栈：

前：……，*value*

后：[空]

描述：方法返回值必须为float类型。*value*（位于操作数栈顶部的一个字长的数据）必须为float类型。要执行freturn指令，Java虚拟机从当前栈帧内的操作数栈中弹出float类型值，然后将其压入调用方法栈帧内的操作数栈中。虚拟机将抛弃所有返回方法栈帧中仍然可能存在的值。如果返回方法是同步的，那么该方法被调用时所获得的监视器将被释放，调用方法栈帧被设为当前值，虚拟机继续执行调用方法中的语句。

如果当方法被调用时，当前方法是同步的，而且当前线程不持有已被获取和重入的监视器，虚拟机将会抛出IllegalMonitorStateException异常。否则，如果虚拟机实现强制执行monitorexit指令中描述的结构化锁定的规则，虚拟机将会抛出IllegalMonitorStateException异常。

如果需要获取更多有关监视器的信息，请参考第20章。如果需要获取更多有关freturn指令的信息，请参考第19章。

fstore

将float类型值存入局部变量中。

操作码字节值：56(0x38)

指令格式：fstore，*index*

栈：

前：……，*value*

后：……

描述：操作数*index*必须是一个指向当前栈帧内局部变量的有效的8位无符号索引。*value*（位于操作数栈顶部一个字长的数据）必须为int类型。要执行fstore指令，Java虚拟机从操作数栈顶部弹出float类型值（*value*），然后将该值存入由索引*index*所指定的局部变量字中。

需要注意的是，fstore指令前面可以使用wide指令，这样就能够将弹出的值*value*存到由16位无符号偏移量所指定的局部变量中。

如果需要获取更多有关fstore指令的信息，请参考第10章。

fstore_0

将float类型值存入局部变量0中。

操作码字节值：67(0x43)

指令格式：fstore_0

栈：

前：……，*value*

后：……

描述：索引0必须是一个指向当前栈帧内局部变量的有效索引。*value*（位于操作数栈顶部一个字长的数据）必须为int类型。要执行fstore_0指令，Java虚拟机从操作数栈顶部弹出一个float类型值（*value*），然后将该值存入由索引0所指定的局部变量字中。

如果需要获取更多有关fstore_0指令的信息，请参考第10章。

fstore_1

将float类型值存入局部变量1中。

操作码字节值：68(0x44)

指令格式：fstore_1

栈：

前：……，*value*

后：……

描述：索引1必须是一个指向当前栈帧内局部变量的有效索引。*value*（位于操作数栈顶部一个字长的数据）必须为int类型。要执行fstore_1指令，Java虚拟机从操作数栈顶部弹出一个float类型值（*value*），然后将该值存入由索引1所指定的局部变量字中。

如果需要获取更多有关fstore_1指令的信息，请参考第10章。

fstore_2

将float类型值存入局部变量2中。

操作码字节值：69(0x45)

指令格式：fstore_2

栈：

前：……，*value*

后：……

描述：索引2必须是一个指向当前栈帧内局部变量的有效索引。*value*（位于操作数栈顶部一个字长的数据）必须为int类型。要执行fstore_2指令，Java虚拟机从操作数栈顶部弹出一个float类型值（*value*），然后将该值存入由索引2所指定的局部变量字中。

如果需要获取更多有关fstore_2指令的信息，请参考第10章。

fstore_3

将float类型值存入局部变量3中。

操作码字节值：70 (0x46)

指令格式：fstore_3

栈：

前：……，*value*

后：……

描述：索引3必须是一个指向当前栈帧内局部变量的有效索引。*value*（位于操作数栈顶部一个字长的数据）必须为int类型。要执行fstore_3指令，Java虚拟机从操作数栈顶部弹出一个float类型值（*value*），然后将该值存入由索引3所指定的局部变量字中。

如果需要获取更多有关fstore_3指令的信息，请参考第10章。

fsub

执行float类型的减法。

操作码字节值：102(0x66)

指令格式：fsub

栈：

前：……，*value1*，*value2*

后：……，*result*

描述：*value1*和*value2*（位于操作数栈顶部两个字长的数据）必须为float类型。要执行fsub指令，Java虚拟机首先将*value1*和*value2*弹出栈，用*value1*减去*value2*（*value1* – *value2*），再将所得float类型结果（*result*）压入栈。执行fsub所得到的结果由IEEE 754中的浮点算法规则决定。

如果需要获取更多有关fsub指令的信息，请参考第14章。

getfield

从对象中获取字段。

操作码字节值：180(0xb4)

指令格式：getfield，*indexbyte1*，*indexbyte2*

栈：

前：……，*objectref*

后：……，*value*

或者

前：……，*objectref*

后：……，*value.word1*，*value.word2*

描述：*objectref*（栈顶部的一个字长的数据）必须为引用类型。要执行getfield指令，Java虚拟机首先通过计算(*indexbyte1* << 8)| *indexbyte2*，得到一个指向常量池的16位无符号索引。然后，虚拟机根据这个索引查找相应的常量池入口。这个索引所指向的常量池入口必须为CONSTANT_Fieldref_info入口。如果该入口尚未存在，那么虚拟机将解析这个入口，这将产生字段的宽度，以及从对象映象的起始点开始的字段的偏移量。如果解析操作成功，Java虚拟机将*objectref*弹出栈，并从由*objectref*所指向的对象中取出字段。如果该字段的类型为byte、short或者char，虚拟机把该字段的值带符号扩展为一个int类型值，然后再将一个字长的int类型值（*value*）压入操作数栈。如果字段的类型为int、boolean、float或者reference，虚拟机将此一个字长的值（*value*）压入操作数栈。如果字段的类型为long或者double，虚拟机将把两个字长的值（*value*）压入栈。

执行这条指令的结果是，虚拟机在解析CONSTANT_Fieldref_info入口的过程中，有可能会抛出第8章中所列出的任何连接错误。在解析CONSTANT_Fieldref_info的过程中，虚拟机检查字段的访问权限，决定是否允许当前类访问该字段。如果该字段是被保护的（protect），虚拟机需要确认该字段是否为当前类或者当前类的超类，以及由*objectref*所指向对象的类是否为当前类或者当前类的子类。如果答案是否定的（或者有任何访问权限问题），虚拟机将会抛出IllegalAccessError异常。另外，如果字段存在并且能够被当前类访问，但该字段为静态字段，虚拟机将会抛出IncompatibleClassChangeError异常。如果*objectref*的值为null，虚拟机将会抛出NullPointerException异常。

如果需要获取更多有关getfield指令的信息，请参考第15章。

getstatic

从类中获取静态字段。

操作码字节值：178(0xb2)

指令格式：getstatic，*indexbyte1*，*indexbyte2*

栈：

前：……

后：……，*value*

或者

前：……

后：……，*value.word1*，*value.word2*

描述：要执行getstatic指令，Java虚拟机首先通过计算(*indexbyte1* << 8)| *indexbyte2*，得到一个指向常量池的16位无符号索引。然后，虚拟机根据这个索引查找相应的常量池入口。这个索引所指向的常量池入

口必须为CONSTANT_Fieldref_info入口。如果该入口不存在，那么虚拟机就解析该入口。如果解析操作成功，Java虚拟机就取出静态字段的值。如果该字段的类型为byte、short或者char，虚拟机将会把该字段的值带符号扩展为一个int类型值，然后再将一个字长的int类型值（*value*）压入操作数栈。如果字段的类型为int、boolean、float或者reference，虚拟机将此一个字长的值（*value*）压入操作数栈。如果字段的类型为long或者double，虚拟机将把两个字长的值（*value*）压入栈。

执行这条指令的结果是，虚拟机在解析CONSTANT_Fieldref_info入口的过程中，有可能会抛出第8章中所列出的任何连接错误。在解析CONSTANT_Fieldref_info的过程中，虚拟机检查字段的访问权限，决定是否允许当前类访问该字段。如果该字段是被保护的（protect），虚拟机需要确认该字段是否为当前类或者当前类的超类，以及由*objectref*所指向对象的类是否为当前类或者当前类的子类。如果答案是否定的（或者有任何访问权限问题），虚拟机将会抛出IllegalAccessError异常。另外，如果字段存在并且能够被当前类访问，但该字段不是静态字段，虚拟机将会抛出IncompatibleClassChangeError异常。

如果需要获取更多有关getstatic指令的信息，请参考第15章。

goto

无条件跳转。

操作码字节值：167(0xa7)

指令格式：goto，*branchbyte1*，*branchbyte2*

栈：无变化

描述：要执行goto指令，Java虚拟机首先通过计算(*branchbyte1* << *8*)| *branchbyte2*，得到一个16位带符号偏移量。然后，虚拟机通过把算出的偏移量与goto操作码的地址相加，计算出目标（程序计数器）地址。目标地址必须为与goto操作码位于同一方法内的操作码地址。虚拟机跳转至目标地址，并在新位置继续执行。

如果需要获取更多有关goto指令的信息，请参考第16章。

goto_w

无条件跳转（宽索引）。

操作码字节值：200(0xc8)

指令格式：goto_w = 200，*branchbyte1*，*branchbyte2*，*branchbyte3*，*branchbyte4*

栈：无变化

描述：要执行goto_w指令，Java虚拟机首先通过计算(*branchbyte1* << *24*)| (*branchbyte2* << *16*) | (*branchbyte3* << *8*) |*branchbyte4*，得到一个32位带符号偏移量。然后，虚拟机通过把算出的偏移量与goto_w操作码的地址相加，计算出目标（程序计数器）地址。目标地址必须为与goto_w操作码位于同一方法内的操作码地址。虚拟机跳转至目标地址，并在新位置继续执行。

需要注意的是，尽管goto_w指令允许32位偏移量，但当前Java（1.0和1.1版本）方法在Java class文件格式中因为以下三项有65 535字节的限制：LineNumberTable属性、LocalVariableTable属性和Code属性的exception_table项中的索引长度。依据Java虚拟机规范，对Java方法的65 535字节限制可能会在将来的版本中得到改善。如果需要获取更多有关goto_w指令的信息，请参考第16章。

i2b

把int类型的数据转换为byte类型。

操作码字节值：145(0x91)

指令格式：i2b

栈：

前：……，*value*

后：……，*result*

描述：*value*（位于操作数栈顶部的一个字长的数据）必须为int类型。要执行i2b指令，Java虚拟机首先从操作数栈中弹出一个int类型值（*value*），然后将int类型值截短为byte类型值，将其符号扩展为int类型，最后将得到的int结果（*result*）压入栈。

需要注意的是，这条指令执行的是缩窄的基本类型转换。转换可能导致数量信息的丢失和符号位的改变。

参考第11章可以获取更多有关i2b指令的信息。

i2c

把int类型的数据转换为char类型。

操作码字节值：146(0x92)

指令格式：i2c

栈：

前：……，*value*

后：……，*result*

描述：*value*（位于操作数栈顶部的一个字长的数据）必须为int类型。要执行i2c指令，Java虚拟机首先从操作数栈中弹出一个int类型值（*value*），然后将int类型值截短为char类型值，将其零扩展为int类型，最后将得到的int结果（*result*）压入栈。

需要注意的是，这条指令执行的是缩窄的基本类型转换。转换可能导致数量信息的丢失，因为结果永远为正，符号位可能会改变。

参考第11章可以获取更多有关i2c指令的信息。

i2d

把int类型的数据转换为double类型。

操作码字节值：135(0x87)

指令格式：i2d

栈：

前：……，*value*

后：……，*result.word1*，*result.word2*

描述：*value*（位于操作数栈顶部的一个字长的数据）必须为int类型。要执行i2d指令，Java虚拟机首先从操作数栈中弹出一个int类型值（*value*），然后将其符号扩展为double类型，最后将得到的double类型结

果（*result*）压入栈。

需要注意的是，这条指令执行的是放宽的基本类型转换。因为所有的int类型值都能使用double类型来表示，因此转换是准确的。

参考第11章可以获取更多有关i2d指令的信息。

i2f

把int类型的数据转换为float类型。

操作码字节值：134(0x86)

指令格式：i2f

栈：

前：……，*value*

后：……，*result*

描述：*value*（位于操作数栈顶部的一个字长的数据）必须为int类型。要执行i2f指令，Java虚拟机首先从操作数栈中弹出一个int类型值（*value*），然后使用IEEE四舍五入（round-to-nearest）模式将其转换为float类型，最后将得到的float类型结果（*result*）压入栈。

需要注意的是，这条指令执行的是放宽的基本类型转换。因为并不是所有int类型值都能准确无误地转换为float类型，转换有可能导致数值和精度的丢失。

参考第11章可以获取更多有关i2f指令的信息。

i2l

把int类型的数据转换为long类型。

操作码字节值：133(0x85)

指令格式：i2l

栈：

前：……，*value*

后：……，*result.word1*，*result.word2*

描述：*value*（位于操作数栈顶部的一个字长的数据）必须为int类型。要执行i2l指令，Java虚拟机首先从操作数栈中弹出一个int类型值（*value*），然后将其符号扩展为long类型，最后将得到的long类型结果（*result*）压入栈。

需要注意的是，这条指令执行的是放宽的基本类型转换。因为所有的int类型值都能使用long类型来表示，因此转换是准确的。

参考第11章可以获取更多有关i2l指令的信息。

i2s

把int类型的数据转换为short类型。

操作码字节值：147(0x93)

指令格式：i2s

栈：

前：……，*value*

后：……，*result*

描述：*value*（位于操作数栈顶部的一个字长的数据）必须为int类型。要执行i2s指令，Java虚拟机首先从操作数栈中弹出一个int类型值（*value*），然后将int类型值截短为short类型值，将其符号扩展为int类型，最后将得到的int类型结果（*result*）压入栈。

需要注意的是，这条指令执行的是缩窄的基本类型转换，转换可能导致数量信息的丢失和符号位的改变。

参考第11章可以获取更多有关i2s指令的信息。

iadd

执行int类型的加法。

操作码字节值：96(0x60)

指令格式：iadd

栈：

前：……，*value1*，*value2*

后：……，*result*

描述：*value1*和*value2*（位于操作数栈顶部两个字长的数据）必须为int类型。要执行iadd指令，Java虚拟机首先将*value1*和*value2*弹出栈，相加，再将所得int类型结果（*result*）压入栈。如果发生溢出，取使用足够长度的二进制补码形式表示的实际数学运算结果的低32位，因此结果的符号可能与实际的数学运算结果不一致。

如果需要获取更多有关iadd指令的信息，请参考第12章。

iaload

从数组中装载int类型的数据。

操作码字节值：46(0x2e)

指令格式：iaload

栈：

前：……，*arrayref*，*index*

后：……，*value*

描述：要执行iaload指令，Java虚拟机首先从操作数栈中弹出两个字长的数据（*arrayref*和*index*）。*arrayref*必须为指向int类型数组的引用。*index*必须为int类型。虚拟机从*arrayref*所指向的数组获取*index*所指定的int类型的值（*value*），然后将其压入操作数栈。

如果*arrayref*引用的数组为null，虚拟机将会抛出NullPointerException异常。另外，如果*index*不是*arrayref*数组的有效索引，虚拟机将会抛出ArrayIndexOutOfBoundsException异常。

如果需要获取更多有关iaload指令的信息，请参考第15章。

iand

对int类型值进行“逻辑与”操作。

操作码字节值：126(0x7e)

指令格式：iand

栈：

前：……，*value1*，*value2*

后：……，*result*

描述：*value1*和*value2*（位于操作数栈顶部两个字长的数据）必须为int类型。要执行iand指令，Java虚拟机首先将*value1*和*value2*弹出栈，进行“按位逻辑与”操作，再将所得int类型结果（*result*）压入栈。

如果需要获取更多有关iand指令的信息，请参考第13章。

iastore

将int类型的数据存入数组中。

操作码字节值：79(0x4f)

指令格式：iastore

栈：

前：……，*arrayref*，*index*，*value*

后：……

描述：要执行iastore指令，Java虚拟机首先从操作数栈中弹出三个字长的数据（*arrayref*，*index*，*value*）。*arrayref*必须为指向int数组的引用。*index*和*value*的数据类型必须为int。虚拟机将int类型值*value*存入*index*所指定的*arrayref*指向的数组。

如果*arrayref*引用的数组为null，虚拟机将会抛出NullPointerException异常。如果*arrayref*引用的数组不为null，但*index*不是*arrayref*数组的有效索引，虚拟机将会抛出ArrayIndexOutOfBoundsException异常。

如果需要获取更多有关iastore指令的信息，请参考第15章。

iconst_0

将int类型常量0压入栈。

操作码字节值：3(0x3)

指令格式：iconst_0

栈：

前：……

后：……，*0*

描述：要执行iconst_0指令，Java虚拟机将int类型常量0压入操作数栈。

如果需要获取更多有关iconst _0指令的信息，请参考第10章。

iconst_1

将int类型常量1压入栈。

操作码字节值：4(0x4)

指令格式：iconst_1

栈：

前：……

后：……，*1*

描述：要执行iconst_1指令，Java虚拟机将int类型常量1压入操作数栈。

如果需要获取更多有关iconst_1指令的信息，请参考第10章。

iconst_2

将int类型常量2压入栈。

操作码字节值：5(0x5)

指令格式：iconst_2

栈：

前：……

后：……，*2*

描述：要执行iconst_2指令，Java虚拟机将int类型常量2压入操作数栈。

如果需要获取更多有关iconst_2指令的信息，请参考第10章。

iconst_3

将int类型常量3压入栈。

操作码字节值：6(0x6)

指令格式：iconst_3

栈：

前：……

后：……，*3*

描述：要执行iconst_3指令，Java虚拟机将int类型常量3压入操作数栈。

如果需要获取更多有关iconst_3指令的信息，请参考第10章。

iconst_4

将int类型常量4压入栈。

操作码字节值：7(0x7)

指令格式：iconst_4

栈：

前：……

后：……，*4*

描述：要执行iconst_4指令，Java虚拟机将int类型常量4压入操作数栈。

如果需要获取更多有关iconst_4指令的信息，请参考第10章。

iconst_5

将int类型常量5压入栈。

操作码字节值：8(0x8)

指令格式：iconst_5

栈：

前：……

后：……，*5*

描述：要执行iconst_5指令，Java虚拟机将int类型常量5压入操作数栈。

如果需要获取更多有关iconst_5指令的信息，请参考第10章。

iconst_m1

将int类型常量-1压入栈。

操作码字节值：2(0x2)

指令格式：iconst_m1

栈：

前：……

后：……，*-1*

描述：要执行iconst_m1指令，Java虚拟机将int类型常量-1压入操作数栈。

如果需要获取更多有关iconst_m1指令的信息，请参考第10章。

idiv

执行int类型的除法。

操作码字节值：108(0x6c)

指令格式：idiv

栈：

前：……，*value1*，*value2*

后：……，*result*

描述：*value1*和*value2*（位于操作数栈顶部两个字长的数据）必须为int类型。要执行idiv指令，Java虚拟机首先将*value1*和*value2*弹出栈，执行*value1*和*value2*的整数除法操作（*value1*/*value2*），再将所得int类型结果（*result*）压入栈。

整数除法对所得的实际数学运算结果（商）取整。如果除数大于被除数，那么int类型结果必然为0。还有一种特殊情况：当除数和被除数符号相同时，所得结果的符号为正，当除数和被除数符号相异时，所得结果的符号为负。下面是该规则的一种例外情况：当除数是能够使用int数据类型表述的最小的负整数，被除数为-1时，该除法的实际数学运算结果是一个能被int数据类型所表示的最大的正数还要大1的正整数。因此，除法发生溢出，结果等于被除数。

如果*value2*（除数）为0，Java虚拟机将抛出ArithmeticException异常。

如果需要获取有关idiv指令的更多信息，请参考第12章。

ifeq

如果等于0，则跳转。

操作码字节值：153(0x99)

指令格式：ifeq，*branchbyte1*，*branchbyte2*

栈：

前：……，*value*

后：……

描述：*value*（位于操作数栈顶部的一个字长的数据）必须为int类型。要执行ifeq指令，Java虚拟机首先将*value*弹出栈，然后执行*value*和0的比较。如果*value*等于0，虚拟机通过计算表达式(*branchbyte1* << 8) | *branchbyte2*给出一个带符号16位偏移量。然后，通过将计算得出的偏移量加到ifeq操作码的地址上，虚拟机计算出目标（程序计数器）地址。目标地址必须为与ifeq操作码处于同一个方法内的操作码地址。虚拟机跳转到目标地址并且在那里继续执行。如果*value*不等于0，虚拟机不会执行跳转操作，它只是继续执行ifeq指令后面的指令。

如果需要获取有关ifeq指令的更多信息，请参考第16章。

ifge

如果大于等于0，则跳转。

操作码字节值：156(0x9c)

指令格式：ifge，*branchbyte1*，*branchbyte2*

栈：

前：……，*value*

后：……

描述：*value*（位于操作数栈顶部的一个字长的数据）必须为int类型。要执行ifge指令，Java虚拟机首先将*value*弹出栈，然后执行*value*和0的比较。如果*value*大于或者等于0，虚拟机通过计算表达式（*branchbyte1* << 8）| *branchbyte2*给出一个带符号16位偏移量。然后，通过将计算得出的偏移量加到ifge操作码的地址上，虚拟机计算出目标（程序计数器）地址。目标地址必须为与ifge操作码处于同一个方法内的操作码地址。虚拟机跳转到目标地址并且在那里继续执行。如果*value*小于0，虚拟机不会执行跳转操作，它只是继续执行ifge指令后面的指令。

如果需要获取有关ifge指令的更多信息，请参考第16章。

ifgt

如果大于0，则跳转。

操作码字节值：157(0x9d)

指令格式：ifgt，*branchbyte1*，*branchbyte2*

栈：

前：……，*value*

后：……

描述：*value*（位于操作数栈顶部的一个字长的数据）必须为int类型。要执行ifgt指令，Java虚拟机首先将*value*弹出栈，然后执行*value*和0的比较。如果*value*大于0，虚拟机通过计算表达式(*branchbyte1* <<8)

*branchbyte2*给出一个带符号16位偏移量。然后，通过将计算得出的偏移量加到ifgt操作码的地址上，虚拟机计算出目标（程序计数器）地址。目标地址必须为与ifgt操作码处于同一个方法内的操作码地址。虚拟机跳转到目标地址并且在那里继续执行。如果*value*小于或等于0，虚拟机不会执行跳转操作，它只是继续执行ifgt指令后面的指令。

如果需要获取有关ifgt指令的更多信息，请参考第16章。

ifle

如果小于等于0，则跳转。

操作码字节值：158(0x9e)

指令格式：ifle，*branchbyte1*，*branchbyte2*

栈：

前：……，*value*

后：……

描述：*value*（位于操作数栈顶部的一个字长的数据）必须为int类型。要执行ifle指令，Java虚拟机首先将*value*弹出栈，然后执行*value*和0的比较。如果*value*小于或者等于0，虚拟机通过计算表达式（*branchbyte1* << 8）| *branchbyte2*给出一个带符号16位偏移量。然后，通过将计算得出的偏移量加到ifle操作码的地址上，虚拟机计算出目标（程序计数器）地址。目标地址必须为与ifle操作码处于同一个方法内的操作码地址。虚拟机跳转到目标地址并且在那里继续执行。如果*value*大于0，虚拟机不会执行跳转操作，它只是继续执行ifle指令后面的指令。

如果需要获取有关ifle指令的更多信息，请参考第16章。

iflt

如果小于0，则跳转。

操作码字节值：155(0x9b)

指令格式：iflt，*branchbyte1*，*branchbyte2*

栈：

前：……，*value*

后：……

描述：*value*（位于操作数栈顶部的一个字长的数据）必须为int类型。要执行iflt指令，Java虚拟机首先将*value*弹出栈，然后执行*value*和0的比较。如果*value*小于0，虚拟机通过计算表达式（*branchbyte1* << 8）| *branchbyte2*给出一个带符号16位偏移量。然后，通过将计算得出的偏移量加到iflt操作码的地址上，虚拟机计算出目标（程序计数器）地址。目标地址必须为与iflt操作码处于同一个方法内的操作码地址。虚拟机跳转到目标地址并且在那里继续执行。如果*value*大于或者等于0，虚拟机不会执行跳转操作，它只是继续执行iflt指令后面的指令。

如果需要获取有关iflt指令的更多信息，请参考第16章。

ifne

如果不等于0，则跳转。

操作码字节值：154(0x9a)

指令格式：ifne，*branchbyte1*，*branchbyte2*

栈：

前：……，*value*

后：……

描述：*value*（位于操作数栈顶部的一个字长的数据）必须为int类型。要执行ifne指令，Java虚拟机首先将*value*弹出栈，然后执行*value*和0的比较。如果*value*不等于0，虚拟机通过计算表达式（*branchbyte1* << 8）| *branchbyte2*给出一个带符号16位偏移量。然后，通过将计算得出的偏移量加到ifne操作码的地址上，虚拟机计算出目标（程序计数器）地址。目标地址必须为与ifne操作码处于同一个方法内的操作码地址。虚拟机跳转到目标地址并且在那里继续执行。如果*value*等于0，虚拟机不会执行跳转操作，它只是继续执行ifne指令后面的指令。

如果需要获取有关ifne指令的更多信息，请参考第16章。

ifnonnull

如果不等于null，则跳转。

操作码字节值：199(0xc7)

指令格式：ifnonnull，*branchbyte1*，*branchbyte2*

栈：

前：……，*value*

后：……

描述：*value*（位于操作数栈顶部的一个字长的数据）必须为引用类型。要执行ifnonnull指令，Java虚拟机首先将*value*弹出栈，然后执行*value*和null的比较。如果*value*不等于null，虚拟机通过计算表达式（*branchbyte1* << 8）| *branchbyte2*给出一个带符号16位偏移量。然后，通过将计算得出的偏移量加到ifnonnull操作码的地址上，虚拟机计算出目标（程序计数器）地址。目标地址必须为与ifnonnull操作码处于同一个方法内的操作码地址。虚拟机跳转到目标地址并且在那里继续执行。如果*value*等于null，虚拟机不会执行跳转操作，它只是继续执行ifnonnull指令后面的指令。

如果需要获取有关ifnonnull指令的更多信息，请参考第16章。

ifnull

如果等于null，则跳转。

操作码字节值：198(0xc6)

指令格式：ifnull，*branchbyte1*，*branchbyte2*

栈：

前：……，*value*

后：……

描述：*value*（位于操作数栈顶部的一个字长的数据）必须为引用类型。要执行ifnull指令，Java虚拟机首先将*value*弹出栈，然后执行*value*和null的比较。如果*value*等于null，虚拟机通过计算表达式

(*branchbyte1* << 8) | *branchbyte2*给出一个带符号16位偏移量。然后，通过将计算得出的偏移量加到ifnull操作码的地址上，虚拟机计算出目标（程序计数器）地址。目标地址必须为与ifnull操作码处于同一个方法内的操作码地址。虚拟机跳转到目标地址并且在那里继续执行。如果*value*不等于null，虚拟机不会执行跳转操作，它只是继续执行ifnull指令后面的指令。

如果需要获取有关ifnull指令的更多信息，请参考第16章。

if_acmpeq

如果两个对象引用相等，则跳转。

操作码字节值：165(0xa5)

指令格式：if_acmpeq，*branchbyte1*，*branchbyte2*

栈：

前：……，*value1*，*value2*

后：……

描述：*value1*与*value2*（位于操作数栈顶部的两个字长的数据）必须为引用类型。要执行if_acmpeq指令，Java虚拟机首先将*value1*与*value2*弹出栈，然后执行*value1*和*value2*的比较。如果*value1*与*value2*相等（换句话说，如果它们都指向同一对象或者同时为null），虚拟机通过计算表达式(*branchbyte1* << 8) | *branchbyte2*给出一个带符号16位偏移量。然后，通过将计算得出的偏移量加到if_acmpeq操作码的地址上，虚拟机计算出目标（程序计数器）地址。目标地址必须为与if_acmpeq操作码处于同一个方法内的操作码地址。虚拟机跳转到目标地址并且在那里继续执行。如果*value1*不等于*value2*，虚拟机不会执行跳转操作，它只是继续执行if_acmpeq指令后面的指令。

如果需要获取有关if_acmpeq指令的更多信息，请参考第16章。

if_acmpne

如果两个对象引用不相等，则跳转。

操作码字节值：166(0xa6)

指令格式：if_acmpne，*branchbyte1*，*branchbyte2*

栈：

前：……，*value1*，*value2*

后：……

描述：*value1*与*value2*（位于操作数栈顶部的两个字长的数据）必须为引用类型。要执行if_acmpne指令，Java虚拟机首先将*value1*和*value2*弹出栈，然后执行*value1*和*value2*的比较。如果*value1*与*value2*不等（换句话说，如果它们并未指向同一对象，而且不都为null），虚拟机通过计算表达式(*branchbyte1* << 8) | *branchbyte2*给出一个带符号16位偏移量。然后，通过将计算得出的偏移量加到if_acmpne操作码的地址上，虚拟机计算出目标（程序计数器）地址。目标地址必须为与if_acmpne操作码处于同一个方法内的操作码地址。虚拟机跳转到目标地址并且在那里继续执行。如果*value1*与*value2*相等（换句话说，如果它们都指向同一对象或者同时为null），虚拟机不会执行跳转操作，它只是继续执行if_acmpne指令后面的指令。

如果需要获取有关if_acmpne指令的更多信息，请参考第16章。

if_icmpeq

如果两个int类型值相等，则跳转。

操作码字节值：159 (0x9f)

指令格式：if_icmpeq，*branchbyte1*，*branchbyte2*

栈：

前：……，*value1*，*value2*

后：……

描述：*value1*与*value2*（位于操作数栈顶部的两个字长的数据）必须为int类型。要执行if_icmpeq指令，Java虚拟机首先将*value1*和*value2*弹出栈，然后执行*value1*和*value2*的比较。如果*value1*与*value2*相等，虚拟机通过计算表达式(*branchbyte1* << *8*) | *branchbyte2*给出一个带符号16位偏移量。然后，通过将计算得出的偏移量加到if_icmpeq操作码的地址上，虚拟机计算出目标（程序计数器）地址。目标地址必须为与if_icmpeq操作码处于同一个方法内的操作码地址。虚拟机跳转到目标地址并且在那里继续执行。如果*value1*不等于*value2*，虚拟机不会执行跳转操作，它只是继续执行if_icmpeq指令后面的指令。

如果需要获取有关if_icmpeq指令的更多信息，请参考第16章。

if_icmpge

如果一个int类型值大于或者等于另外一个int类型值，则跳转。

操作码字节值：162(0xa2)

指令格式：if_icmpge，*branchbyte1*，*branchbyte2*

栈：

前：……，*value1*，*value2*

后：……

描述：*value1*与*value2*（位于操作数栈顶部的两个字长的数据）必须为int类型。要执行if_icmpge指令，Java虚拟机首先将*value1*和*value2*弹出栈，然后执行*value1*和*value2*的比较。如果*value1*大于或者等于*value2*，虚拟机通过计算表达式(*branchbyte1* << *8*) | *branchbyte2*给出一个带符号16位偏移量。然后，通过将计算得出的偏移量加到if_icmpge操作码的地址上，虚拟机计算出目标（程序计数器）地址。目标地址必须为与if_icmpge操作码处于同一个方法内的操作码地址。虚拟机跳转到目标地址并且在那里继续执行。如果*value1*小于*value2*，虚拟机不会执行跳转操作，它只是继续执行if_icmpge指令后面的指令。

如果需要获取有关if_icmpge指令的更多信息，请参考第16章。

if_icmpgt

如果一个int类型值大于另外一个int类型值，则跳转。

操作码字节值：163(0xa3)

指令格式：if_icmpgt，*branchbyte1*，*branchbyte2*

栈：

前：……，*value1*，*value2*

后：……

描述：*value1*与*value2*（位于操作数栈顶部的两个字长的数据）必须为int类型。要执行if_icmpgt指令，Java虚拟机首先将*value1*和*value2*弹出栈，然后执行*value1*和*value2*的比较。如果*value1*大于*value2*，虚拟机通过计算表达式(*branchbyte1* << *8*) | *branchbyte2*给出一个带符号16位偏移量。然后，虚拟机通过将计算得出的偏移量加到if_icmpgt操作码的地址上的方法，计算出目标（程序计数器）地址。目标地址必须为与if_icmpgt操作码处于同一个方法内的操作码地址。虚拟机跳转到目标地址并且在那里继续执行。如果*value1*小于或者等于*value2*，虚拟机不会执行跳转操作，它只是继续执行if_icmpgt指令后面的指令。

如果需要获取有关if_icmpgt指令的更多信息，请参考第16章。

if_icmple

如果一个int类型值小于或者等于另外一个int类型值，则跳转。

操作码字节值：164(0xa4)

指令格式：if_icmple，*branchbyte1*，*branchbyte2*

栈：

前：……，*value1*，*value2*

后：……

描述：*value1*与*value2*（位于操作数栈顶部的两个字长的数据）必须为int类型。要执行if_icmple指令，Java虚拟机首先将*value1*和*value2*弹出栈，然后执行*value1*和*value2*的比较。如果*value1*小于或者等于*value2*，虚拟机通过计算表达式(*branchbyte1* << *8*) | *branchbyte2*给出一个带符号16位偏移量。然后，通过将计算得出的偏移量加到if_icmple操作码的地址上，虚拟机计算出目标（程序计数器）地址。目标地址必须为与if_icmple操作码处于同一个方法内的操作码地址。虚拟机跳转到目标地址并且在那里继续执行。如果*value1*大于*value2*，虚拟机不会执行跳转操作，它只是继续执行if_icmple指令后面的指令。

如果需要获取有关if_icmple指令的更多信息，请参考第16章。

if_icmplt

如果一个int类型值小于或者等于另外一个int类型值，则跳转。

操作码字节值：161(0xa1)

指令格式：if_icmplt，*branchbyte1*，*branchbyte2*

栈：

前：……，*value1*，*value2*

后：……

描述：*value1*与*value2*（位于操作数栈顶部的两个字长的数据）必须为int类型。要执行if_icmplt指令，Java虚拟机首先将*value1*和*value2*弹出栈，然后执行*value1*和*value2*的比较。如果*value1*小于或者等于*value2*，虚拟机通过计算表达式(*branchbyte1* << *8*) | *branchbyte2*给出一个带符号16位偏移量。然后，通过将计算得出的偏移量加到if_icmplt操作码的地址上，虚拟机计算出目标（程序计数器）地址。目标地址必须为与if_icmplt操作码处于同一个方法内的操作码地址。虚拟机跳转到目标地址并且在那里继续执行。如果*value1*大于*value2*，虚拟机不会执行跳转操作，它只是继续执行if_icmplt指令后面的指令。

如果需要获取有关if_icmplt指令的更多信息，请参考第16章。

if_icmpne

如果两个int类型值不相等，则跳转。

操作码字节值：160(0xa0)

指令格式：if_icmpne，*branchbyte1*，*branchbyte2*

栈：

前：……，*value1*，*value2*

后：……

描述：*value1*与*value2*（位于栈顶部的两个字长的数据）必须为int类型。要执行if_icmpne指令，Java虚拟机首先将*value1*和*value2*弹出栈，然后执行*value1*和*value2*的比较。如果*value1*不等于*value2*，虚拟机通过计算表达式(*branchbyte1* << *8*) | *branchbyte2*给出一个带符号16位偏移量。然后，通过将计算得出的偏移量加到if_icmpne操作码的地址上的方法，虚拟机计算出目标（程序计数器）地址。目标地址必须为与if_icmpne操作码处于同一个方法内的操作码地址。虚拟机跳转到目标地址并且在那里继续执行。如果*value1*与*value2*相等，虚拟机不会执行跳转操作，它只是继续执行紧随if_icmpne指令后面的指令。

如果需要获取有关if_icmpne指令的更多信息，请参考第16章。

iinc

把一个常量值加到一个int类型的局部变量上。

操作码字节值：132(0x84)

指令格式：iinc，*index*，*const*

栈：无变化

描述：操作数*index*必须为一个指向当前栈帧中局部变量的有效的8位无符号索引。要执行iinc指令，Java虚拟机将一个8位带符号的常量增量*const*加到由*index*所指定的局部变量字中。

需要注意的是，iinc指令前面可以使用wide指令，这样就能给一个由16位无符号偏移量所指向的局部变量加上一个带符号的16位常量。

如果需要获取有关iinc指令的更多信息，请参考第12章。

iload

从局部变量中装载int类型值。

操作码字节值：21(0x15)

指令格式：iload，*index*

栈：

前：……

后：……，*value*

描述：作为一个对当前栈帧中局部变量的8位无符号索引，操作数*index*必须指向一个包含int类型数据的局部变量字。要执行iload指令，Java虚拟机将包含在*index*所指向的局部变量字中int类型值（*value*）压入

操作数栈。

需要注意的是，iload指令前面可以使用wide指令，这样就能够访问一个由16位无符号偏移量所指向的局部变量。

如果需要获取有关iload指令的更多信息，请参考第10章。

iload_0

从局部变量0中装载int类型值。

操作码字节值：26(0x1a)

指令格式：iload_0

栈：

前：……

后：……，*value*

描述：在索引0位置的局部变量字必须包含一个int类型值。要执行iload_0指令，Java虚拟机将局部变量字0中存储的int类型值（*value*）压入操作数栈。

如果需要获取有关iload_0指令的更多信息，请参考第10章。

iload_1

从局部变量1中装载int类型值。

操作码字节值：27(0x1b)

指令格式：iload_1

栈：

前：……

后：……，*value*

描述：在索引1位置的局部变量字必须包含一个int类型值。要执行iload_1指令，Java虚拟机将局部变量字1中存储的int类型值（*value*）压入操作数栈。

如果需要获取有关iload_1指令的更多信息，请参考第10章。

iload_2

从局部变量2中装载int类型值。

操作码字节值：28(0x1c)

指令格式：iload_2

栈：

前：……

后：……，*value*

描述：在索引2位置的局部变量字必须包含一个int类型值。要执行iload_2指令，Java虚拟机将局部变量字2中存储的int类型值（*value*）压入操作数栈。

如果需要获取有关iload_2指令的更多信息，请参考第10章。

iload_3

从局部变量3中装载int类型值。

操作码字节值：29(0x1d)

指令格式：iload_3

栈：

前：……

后：……，*value*

描述：在索引3位置的局部变量字必须包含一个int类型值。要执行iload_3指令，Java虚拟机将局部变量字3中存储的int类型值（*value*）压入操作数栈。

如果需要获取有关iload_3指令的更多信息，请参考第10章。

imul

执行int类型的乘法。

操作码字节值：104(0x68)

指令格式：imul

栈：

前：……，*value1*，*value2*

后：……，*result*

描述：*value1*和*value2*（位于操作数栈顶部两个字长的数据）必须为int类型。要执行imul指令，Java虚拟机首先将*value1*和*value2*弹出栈，相乘，再将所得int类型结果（*result*）压入栈。如果发生溢出，取使用足够长度的二进制补码形式表示的实际数学运算结果的低32位，因此结果的符号可能与实际的数学运算结果不一致。

如果需要获取更多有关imul指令的信息，请参考第12章。

ineg

对一个int类型值进行取反操作。

操作码字节值：116(0x74)

指令格式：ineg

栈：

前：……，*value*

后：……，*result*

描述：*value*（位于操作数栈顶部一个字长的数据）必须为int类型。要执行ineg指令，Java虚拟机首先将*value*弹出栈，执行取反操作，再将所得int类型结果（*result*）压入栈。

执行ineg指令所得到的结果与使用isub指令用0减去*value*所得的结果相同。因而，当*value*为可以用int数据类型表示的最小负整数时，对其进行取反操作会发生溢出。对于这样的取反操作，实际数学运算结果是一个比用int数据类型表示的最大正整数还要大1的数值，而所得到的实际结果是符号并未改变的与原来

*value*值相等的值。

如果需要获取更多有关ineg指令的信息，请参考第12章。

instanceof

判断对象是否为给定的类型。

操作码字节值：193(0xc1)

指令格式：instanceof，*indexbyte1*，*indexbyte2*

栈：

前：……，*objectref*

后：……，*result*

描述：*objectref*（位于栈顶部一个字长的数据）的数据类型必须为引用类型。要执行instanceof指令，Java虚拟机首先通过计算(*indexbyte1*<<*8*) | *indexbyte2*来给出一个指向常量池的无符号16位索引。接下来虚拟机会通过计算得出的索引来查找常量池入口。该索引指向的常量池入口必须为CONSTANT_Class_info入口。如果该入口尚未存在，那么虚拟机将解析这个常量池入口。该入口可以为类、接口和数组类型。如果*objectref*不为null，而且由*objectref*所指向的对象是所解析类型的“实例”，虚拟机就会把int类型结果（*result*）值1压入操作数栈。否则，虚拟机把int类型结果值0压入操作数栈。

为了判断由*objectref*所指向的对象是否为所解析类型的“实例”，虚拟机首先判断对象是类实例还是数组（该对象不可能为接口实例，因为接口不能实例化）。如果该对象是一个类实例，而且所解析类型为类的话，如果对象所属的类与所解析类相同或是所解析类的子类，那么该对象就是所解析类的“一个实例”。还有，如果该对象是一个类实例，但是所解析类型为接口（而不是类）的话，如果该对象的类实现了所解析接口，那么该对象就是所解析接口的“一个实例”。另外，如果该对象是一个数组，那么如果所解析类型是类的话，这个类必须为java.lang.Object。如果所解析类型是基本数据类型，那么该对象必须是一个数据成员为同样基本数据类型的数组。如果所解析类型是一个数据成员为某种引用类型的数组，那么对象必须是数据成员也是该类型引用的数组（需要注意的是，指令instanceof并没有检查数组的维数，它只对数组成员的类型进行检查）。

执行这条指令的结果是，虚拟机可能会在解析CONSTANT_Class_info入口的过程中，抛出任意第8章中列出的连接错误。

如果需要获取更多有关instanceof指令的信息，请参考第15章。

invokeinterface

调用接口方法。

操作码字节值：185(0xb9)

指令格式：invokeinterface，*indexbyte1*，*indexbyte2*，*nargs*，*0*

栈：

前：……，*objectref*，*[arg1*，*[arg2* ……*]]*

后：……

描述：要执行invokeinterface指令，Java虚拟机首先通过计算(*indexbyte1*<<*8*) | *indexbyte2*来给出一个

指向常量池的无符号16位索引。接下来虚拟机会通过计算得出的索引来查找常量池入口。该索引指向的常量池入口必须为CONSTANT_InterfaceMethodref_info入口。如果该入口尚未存在，那么虚拟机将解析这个常量池入口。所解析方法的描述符必须与所解析接口中的一个方法的描述符完全匹配。这里的方法不能是实例初始化方法“<init>”或者类初始化方法“<clinit>”。

操作数*nargs*是一个无符号byte类型值，它用来指明被调用的方法所需要参数的字数（包括隐藏的this引用）。操作数栈必须包含*nargs-1*个参数字以及*objectref*字。参数字的类型和出现次序必须与所解析方法的参数的类型与出现次序保持一致。*objectref*字是用来调用实例方法的对象的引用，它的类型必须是引用类型。如果解析过程成功，虚拟机会将*nargs-1*个参数字和*objectref*字从栈弹出。

为了调用该方法，虚拟机首先获取对此实例方法的直接引用，然后从方法表中调用这个方法。它为*objectref*所指的对象的类定位方法表，并在其中仔细搜寻是否有与所解析方法的名称和描述符相匹配的方法。（如果对象的类是数组类型的话，虚拟机将会使用类java.lang.Object的方法表。当然，只有当Object自身实现了这个接口的时候，数组类型才能实现它。）

如果该方法是同步的，Java虚拟机代表当前线程获取与*objectref*相关的监视器。

如果调用的方法不是本地方法，虚拟机将会为该方法建立新的栈帧，并且将新栈帧压入当前线程的Java栈。虚拟机把从调用函数栈帧的操作数栈中弹出的*objectref*字和*nargs-1*个参数字赋给新栈帧中的局部变量。虚拟机把*objectref*赋给位置为0的局部变量，*arg1*赋给位置为1的局部变量，等等（*objectref*就是传给所有实例方法的隐藏的this引用）。虚拟机把新栈帧设为当前栈帧，把程序计数器设为新方法第一条指令的地址，然后在新位置继续执行。

如果调用的方法是本地方法，虚拟机将会使用与实现相关的方式调用本地方法。

执行这条指令的结果是，虚拟机可能会在解析CONSTANT_Methodref_info入口的过程中，抛出任意第8章中列出的连接错误。在解析CONSTANT_Methodref_info入口的过程中，虚拟机会检查该方法的访问权限，判断当前类是否能够访问该方法。如果为受保护的方法，虚拟机将会确认该方法是否为当前类或者当前类的超类的方法，由*objectref*指向的对象的类是否为当前类或当前类的子类。如果不是的话（或者有任何访问权限问题），虚拟机将会抛出IllegalAccessError异常。如果*objectref*的值为null，那么虚拟机将会抛出NullPointerException异常。如果被*objectref*所指向对象的类没有实现此接口，虚拟机将会抛出IncompatibleClassChangeError异常。如果该方法为抽象方法，虚拟机将会抛出AbstractMethodError异常。如果该方法为本地方法，而且方法的本地实现无法完成装载或者连接，虚拟机将会抛出UnsatisfiedLinkError异常。

如果需要获取更多有关invokeinterface指令的信息，请参考第19章。

invokespecial

使用对私有方法、超类方法和实例初始化方法的特殊处理来调用实例方法。

操作码字节值：183(0xb7)

指令格式：invokespecial，*indexbyte1*，*indexbyte2*

栈：

前：……，*objectref*，*[arg1*，*[arg2 ……]]*

后：……

描述：要执行invokespecial指令，Java虚拟机首先通过计算*(indexbyte1<<8) | indexbyte2*来给出一个指向常量池的无符号16位索引。接下来虚拟机会通过计算得出的索引来查找常量池入口。该索引指向的常量池入口必须为CONSTANT_Methodref_info入口。如果该入口尚未存在，那么虚拟机将会解析这个常量池入口，这将产生对方法数据（包括方法所需参数字的数目*nargs*）的直接引用。所解析方法的描述符必须与所解析类中声明的一个方法的描述符完全匹配。这里的方法不能是类初始化方法“<clinit>”。

指令invokespecial被用来进行三种特殊实例方法的调用：超类方法、私有方法和实例初始化方法。指令invokespecial与指令invokevirtual的不同之处在于调用实例方法的方式，指令invokevirtual通常在运行时选择所调用的方法，它的选择是基于对象类型（动态绑定）的；而指令invokespecial通常（只有一个例外）在编译时选择所调用的方法，它的选择是基于引用类型（静态绑定）的。

当下列条件满足时，将会出现指令invokespecial的纯静态绑定行为的惟一例外：

- 所解析的方法不是私有的，并且不是一个实例初始化方法。
- 所解析的方法的类是当前方法的类的超类。
- 当前方法的类设定了ACC_SUPER标志。

在这种情况下，Java虚拟机通过运行时在最近的超类中查找方法的方式，来动态选择所调用的方法，而不考虑该方法所属的类（这里提到的超类必须至少有一个与所解析方法的描述符完全匹配，不考虑所解析方法的类）。大多数情况下，所解析方法的类很可能就是所解析方法的类，但也有可能是其他的一些类。

例如，如果创建了一棵包含3个类的继承树：Animal、Dog和CockerSpaniel。假设类Dog是Animal类的子类，CockerSpaniel类是Dog类的子类，在CockerSpaniel类中定义了一个方法，它使用invokespecial来调用一个名为walk()的非私有的超类方法。再假设当编译CockerSpaniel时，编译器设定了ACC_SUPER标志。此外，假设当编译CockerSpaniel类时，Animal类定义了walk()方法，但Dog类没有定义该方法。此时，CockerSpaniel类中指向walk()方法的符号引用将会把Animal类作为它的类。当执行CockerSpaniel类的方法中的invokespecial指令时，虚拟机会进行动态选择，并调用Animal类的walk()方法。

可想而知，如果后来在Dog类中加入了walk()方法，并且重新编译Dog类，但是没有重新编译CockerSpaniel类，它的指向超类的walk()方法的符号引用仍然会将Animal作为自己的类，尽管在Dog类中已经有了walk()方法的实现。不过，当执行CockerSpaniel类中的invokespecial指令时，虚拟机将会动态选择并调用Dog类中的walk()方法的实现。

给class文件中access_flag项加上ACC_SUPER标志，以及把该操作码的名称从最初的invokenonvirtual改为现在的invokespecial的原因在于对超类调用的特殊处理（不使用静态绑定）。如果使用没有设置ACC_SUPER标志的老版本编译器来编译前面例子中的CockerSpaniel类，虚拟机将无视Dog类中的walk()方法，而调用Animal的walk()方法。如第6章所述，所有新版本的Java编译器都应该在它们所产生的每一个Java class文件中设置ACC_SUPER标志。

如果所解析方法是一个实例初始化方法“<init>”，那么每个未被初始化的对象只能调用一次该方法。此外，每个未被初始化的对象只能在第一次执行向后分支跳转操作之前调用实例初始化方法。换句话说，方法的字节码并不需要在执行完new指令之后，马上使用invokespecial指令调用<init>方法，其他指令能够在new指令和调用<init>的invokespecial指令之间存在。但是这些指令都不能执行向后分支跳转操作（诸如跳转到方法的起始位置）。

操作数栈必须包含*nargs-1*个字长的参数字以及*objectref*字。参数字的类型和出现次序必须与所解析方

法的参数的类型与出现次序保持一致。objectref字是用来调用实例方法的对象的引用，它的类型必须是引用类型。如果解析过程成功，虚拟机会将*nargs-1*个参数字和objectref从栈弹出。

如果该方法是同步的，Java虚拟机代表当前线程获取与*objectref*相关的监视器。

如果调用的方法不是本地方法，虚拟机将会为该方法建立新的栈帧，并且将新栈帧压入当前线程的Java栈。虚拟机把从调用函数栈帧的操作数栈中弹出的*objectref*字和*nargs-1*个参数字赋给新栈帧中的局部变量。虚拟机把*objectref*赋给位置为0的局部变量，*arg1*赋给位置为1的局部变量，等等（*objectref*就是传给所有实例方法的隐藏的this引用）。虚拟机把新栈帧设为当前栈帧，把程序计数器设为新方法第一条指令的地址，然后在新位置继续执行。

如果调用的方法是本地方法，虚拟机将会使用与实现相关的方式调用本地方法。

执行这条指令的结果是，虚拟机可能会在解析CONSTANT_Methodref_info入口的过程中，抛出任意第8章中列出的连接错误。在解析CONSTANT_Methodref_info入口的过程中，虚拟机会检查该方法的访问权限，判断当前类是否能够访问该方法。如果为受保护的方法，虚拟机将会确认该方法是否为当前类或者当前类的超类的方法，由*objectref*指向的对象的类是否为当前类或当前类的子类。如果不是的话（或者有任何访问权限问题），虚拟机将会抛出IllegalAccessError异常。如果所解析的方法是<init>()方法，而且声明该方法的类与符号引用所指向的类不同，那么虚拟机将抛出NoSuchMethodError异常。如果该方法存在，并且可以从当前类访问，但是该方法为静态方法，虚拟机将会抛出IncompatibleClassChangeError异常。如果没有方法与给定方法的名称与描述符相匹配，虚拟机将抛出AbstractMethodError异常。如果该方法为抽象方法，虚拟机将抛出AbstractMethodError异常。如果*objectref*的值为null，虚拟机将会抛出NullPointerException异常。如果该方法为本地方法，而且该方法的本地实现无法装载和连接，虚拟机将会抛出UnsatisfiedLinkError异常。

如果需要获取更多有关invokespecial指令的信息，请参考第19章。

invokestatic

调用类的（静态）方法。

操作码字节值：184(0xb8)

指令格式：invokestatic，*indexbyte1*，*indexbyte2*

栈：

前：……，*[arg1*，*[arg2 ……]]*

后：……

描述：要执行invokestatic指令，Java虚拟机首先通过计算(*indexbyte1*<<*8*) | *indexbyte2*来给出一个指向常量池的无符号16位索引。接下来虚拟机会通过计算得出的索引来查找常量池入口。该索引指向的常量池入口必须为CONSTANT_Methodref_info入口。如果该入口尚未存在，那么虚拟机将会解析这个常量池入口，这将产生对方法数据（包括方法所需要参数字的数目*nargs*）的直接引用。所解析方法的描述符必须与所解析类中声明的一个方法的描述符完全匹配。这里的方法不能是实例初始化方法“<init>”或者类初始化方法“<clinit>”。

操作数栈必须包含nargs个参数字。参数字的类型和出现次序必须与所解析方法的参数的类型与出现次序保持一致。如果解析过程成功，虚拟机会将nargs个参数字从栈弹出。

如果该方法是同步的，Java虚拟机代表当前线程获取与所解析方法类的class实例相关的监视器。

如果调用的方法不是本地方法，虚拟机将会为该方法建立新的栈帧，并且将新栈帧压入当前线程的Java栈。虚拟机把从调用函数栈帧的操作数栈中弹出的*nargs*个参数字赋给新栈帧中的局部变量。虚拟机把*arg1*赋给位置为0的局部变量，把*arg2*赋给位置为1的局部变量，等等。虚拟机把新栈帧设为当前栈帧，把程序计数器设为新方法第一条指令的地址，然后在新位置继续执行。

如果调用的方法是本地方法，虚拟机将会使用与实现相关的方式调用本地方法。

执行这条指令的结果是，虚拟机可能会在解析CONSTANT_Methodref_info入口的过程中，抛出任意第8章中列出的连接错误。在解析CONSTANT_Methodref_info入口的过程中，虚拟机会检查该方法的访问权限，判断当前类是否能够访问该方法。如果为受保护的方法，虚拟机将会确认该方法是否为当前类或者当前类的超类的方法。如果不是的话（或者有任何访问权限问题），虚拟机将会抛出IllegalAccessError异常。如果该方法存在，并且可以被访问，但该方法不是静态方法，虚拟机将会抛出IncompatibleClassChangeError异常。如果该方法为本地方法，而且该方法的本地实现无法完成装载或者连接，虚拟机将会抛出UnsatisfiedLinkError异常。

如果需要获取更多有关invokestatic指令的信息，请参考第19章。

invokevirtual

运行时按照对象的类来调用实例方法。

操作码字节值：182(0xb6)

指令格式：invokevirtual，*indexbyte1*，*indexbyte2*

栈：

前：……，*objectref*，*[arg1*，*[arg2 ……]]*

后：……

描述：要执行invokevirtual指令，Java虚拟机首先通过计算(*indexbyte1*<<*8*) | *indexbyte2*来给出一个指向常量池的无符号16位索引。接下来虚拟机会通过计算得出的索引来查找常量池入口。该索引指向的常量池入口必须为CONSTANT_Methodref_info入口。如果该入口尚未存在，那么虚拟机将会解析这个常量池入口，这将会产生方法的方法表索引*index*，该方法参数字的数目*nargs*。所解析方法的描述符必须与所解析类中声明的一个方法的描述符完全匹配。这里的方法不能是实例初始化方法“<init>”或者类初始化方法“<clinit>”。

操作数栈必须包含*nargs-1*个参数字以及*objectref*字。参数字的类型和出现次序必须与所解析方法的参数的类型与出现次序保持一致。*objectref*字是用来调用实例方法的对象的引用，它的类型必须是引用类型。如果解析过程成功，虚拟机会将*nargs-1*个参数字和*objectref*从栈弹出。

为了调用该方法，虚拟机首先获取对此实例方法（从方法表中调用这个方法）的直接引用。它为*objectref*所指的对象的类定位方法表，并查找处于方法表索引*index*位置的直接引用（如果对象是数组类型的话，虚拟机将使用类java.lang.Object的方法表）。

如果该方法是同步的，Java虚拟机代表当前线程获取与*objectref*相关的监视器。

如果调用的方法不是本地方法，虚拟机将会为该方法建立新的栈帧，并且将新栈帧压入当前线程的Java栈。虚拟机把从调用函数栈帧的操作数栈中弹出的*objectref*字和*nargs-1*个参数字赋给新栈帧中的局部变量。虚拟机把*objectref*赋给位置为0的局部变量，*arg1*赋给位置为1的局部变量，等等（*objectref*就是传给

所有实例方法的隐藏的this引用）。虚拟机把新栈帧设为当前栈帧，把程序计数器设为新方法第一条指令的地址，然后在新位置继续执行。

如果调用的方法是本地方法，虚拟机将会使用与实现相关的方式调用本地方法。

执行这条指令的结果是，虚拟机可能会在解析CONSTANT_Methodref_info入口的过程中，抛出任意第8章中列出的连接错误。在解析CONSTANT_Methodref_info入口的过程中，虚拟机会检查该方法的访问权限，判断当前类是否能够访问该方法。如果为受保护的方法，虚拟机将会确认该方法是否为当前类或者当前类的超类的方法，由*objectref*指向的对象的类是否为当前类或当前类的子类。如果不是的话（或者有任何访问权限问题），虚拟机将会抛出IllegalAccessError异常。如果该方法存在，并且可以被访问，但是该方法为静态方法，虚拟机将会抛出IncompatibleClassChangeError异常。如果*objectref*的值为null，那么虚拟机将会抛出NullPointerException异常。如果没有方法与给定方法的名称与描述符相匹配，虚拟机将抛出AbstractMethodError异常。如果该方法为抽象方法，虚拟机将会抛出AbstractMethodError异常。如果该方法为本地方法，而且方法的本地实现无法完成装载或者连接，虚拟机将会抛出UnsatisfiedLinkError异常。

如果需要获取更多有关invokevirtual指令的信息，请参考第19章。

ior

对int类型值进行“逻辑或”操作。

操作码字节值：128(0x80)

指令格式：ior

栈：

前：……，*value1*，*value2*

后：……，*result*

描述：*value1*和*value2*（位于操作数栈顶部两个字长的数据）必须为int类型。要执行ior指令，Java虚拟机首先将*value1*和*value2*弹出栈，进行“按位逻辑或”操作，再将所得int类型结果（*result*）压入栈。

如果需要获取更多有关ior指令的信息，请参考第13章。

irem

计算int类型除法的余数。

操作码字节值：112(0x70)

指令格式：irem

栈：

前：……，*value1*，*value2*

后：……，*result*

描述：*value1*和*value2*（位于操作数栈顶部两个字长的数据）必须为int类型。要执行irem指令，Java虚拟机首先将*value1*和*value2*弹出栈，计算出整数余数，再将所得int类型结果（*result*）压入栈。取余操作等同于*value1* -（*value1* / *value2*）* *value2*。

irem指令实现了Java的取余操作符（%）。指令irem的特性使得下列Java表达式的值永远为真（n和d为任意两个int类型值）：

```
(n/d)*d + (n%d) == n
```

该特性意味着，指令irem的结果总是与被除数（从操作数栈中弹出的*value1*）的符号保持一致。

如果*value2*（除数）为0，Java虚拟机将抛出ArithmeticException异常。

如果需要获取更多有关irem指令的信息，请参考第12章。

ireturn

从方法中返回int类型的数据。

操作码字节值：172(0xac)

指令格式：ireturn

栈：

前：……，*value*

后：[空]

描述：方法返回值的数据类型必须为下列类型之一：byte、short、int或者char。*value*（位于操作数栈顶部的一个字长的数据）的数据类型必须为int。要执行ireturn指令，Java虚拟机从当前栈帧内的操作数栈中弹出int类型值，然后将其压入调用方法栈帧内的操作数栈中。虚拟机将抛弃所有返回方法栈帧中仍然可能存在的值。如果返回方法是同步的，那么该方法被调用时所获得的监视器将被释放，调用方法栈帧被设为当前值，虚拟机继续执行调用方法中的语句。

如果当方法被调用时，当前方法是同步的，而且当前线程不持有已被获取和重入的监视器，虚拟机将会抛出IllegalMonitorStateException异常。否则，如果虚拟机实现强制执行在monitorexit指令中所描述的结构化锁定的规则，虚拟机将会抛出IllegalMonitorStateException异常。

如果需要获取更多有关ireturn指令的信息，请参考第19章。如果需要获取更多有关监视器的信息，请参考第20章。

ishl

执行int类型的向左移位操作。

操作码字节值：120(0x78)

指令格式：ishl

栈：

前：……，*value1*，*value2*

后：……，*result*

描述：*value1*和*value2*（位于操作数栈顶部两个字长的数据）必须为int类型。要执行ishl指令，Java虚拟机首先将*value1*和*value2*弹出栈，再对*value1*执行向左移位操作，所移位数由*value2*的最低5位（按照从0到31位的顺序）决定，最后将所得int类型结果（*result*）压入栈。

如果需要获取更多有关ishl指令的信息，请参考第13章。

ishr

执行int类型的向右移位操作。

操作码字节值：122(0x7a)

指令格式：ishr

栈：

前：……，*value1*，*value2*

后：……，*result*

描述：*value1*和*value2*（位于操作数栈顶部两个字长的数据）必须为int类型。要执行ishr指令，Java虚拟机首先将*value1*和*value2*弹出栈，再对*value1*执行向右符号扩展移位操作，所移位数由*value2*的最低5位（按照从0到31位的顺序）决定，最后将所得int类型结果（*result*）压入栈。

如果需要获取更多有关ishr指令的信息，请参考第13章。

istore

将int类型值存入局部变量中。

操作码字节值：54(0x36)

指令格式：istore，*index*

栈：

前：……，*value*

后：……

描述：操作数*index*必须指明一个指向当前栈帧内局部变量的有效的8位无符号索引。*value*（位于操作数栈顶部一个字长的数据）必须为int类型。要执行istore指令，Java虚拟机从操作数栈顶部弹出int类型值，然后将该值存入由索引*index*所指定的局部变量字中。

需要注意的是，istore指令前面可以使用wide指令，这样就能够将弹出的值*value*存到16位无符号偏移量所指定的局部变量中。

如果需要获取更多有关istore指令的信息，请参考第10章。

istore_0

将int类型值存入局部变量0中。

操作码字节值：59(0x3b)

指令格式：istore_0

栈：

前：……，*value*

后：……

描述：索引0必须是一个指向当前栈帧内局部变量的有效索引。*value*（位于操作数栈顶部一个字长的数据）必须为int类型。要执行istore_0指令，Java虚拟机从操作数栈顶部弹出一个int类型值，然后将该值存入由索引0所指定的局部变量字中。

如果需要获取更多有关istore_0指令的信息，请参考第10章。

istore_1

将int类型值存入局部变量1中。

操作码字节值：60(0x3c)

指令格式：istore_1

栈：

前：……，*value*

后：……

描述：索引1必须是一个指向当前栈帧内局部变量的有效索引。*value*（位于操作数栈顶部一个字长的数据）必须为int类型。要执行istore_1指令，Java虚拟机从操作数栈顶部弹出一个int类型值，然后将该值存入由索引1所指定的局部变量字中。

如果需要获取更多有关istore_1指令的信息，请参考第10章。

istore_2

将int类型值存入局部变量2中。

操作码字节值：61(0x3d)

指令格式：istore_2

栈：

前：……，*value*

后：……

描述：索引2必须是一个指向当前栈帧内局部变量的有效索引。*value*（位于操作数栈顶部一个字长的数据）必须为int类型。要执行istore_2指令，Java虚拟机从操作数栈顶部弹出一个int类型值，然后将该值存入由索引2所指定的局部变量字中。

如果需要获取更多有关istore_2指令的信息，请参考第10章。

istore_3

将int类型值存入局部变量3中。

操作码字节值：62(0x3e)

指令格式：istore_3

栈：

前：……，*value*

后：……

描述：索引3必须是一个指向当前栈帧内局部变量的有效索引。*value*（位于操作数栈顶部一个字长的数据）必须为int类型。要执行istore_3指令，Java虚拟机从操作数栈顶部弹出一个int类型值，然后将该值存入由索引3所指定的局部变量字中。

如果需要获取更多有关istore_3指令的信息，请参考第10章。

isub

执行int类型的减法。

操作码字节值：100(0x64)

指令格式：isub

栈：

前：……，*value1*，*value2*

后：……，*result*

描述：*value1*和*value2*（位于操作数栈顶部两个字长的数据）必须为int类型。要执行isub指令，Java虚拟机首先将*value1*和*value2*弹出栈，用*value1*减去*value2*（*value1* –*value2*），再将所得int类型结果（*result*）压入栈。如果发生溢出，取使用足够长度的二进制补码形式表示的实际数学运算结果的低32位，因此结果的符号可能与实际的数学运算结果不一致。

如果需要获取更多有关isub指令的信息，请参考第12章。

iushr

执行int类型的向右逻辑移位操作。

操作码字节值：124(0x7c)

指令格式：iushr

栈：

前：……，*value1*，*value2*

后：……，*result*

描述：*value1*和*value2*（位于操作数栈顶部两个字长的数据）必须为int类型。要执行iushr指令，Java虚拟机首先将*value1*和*value2*弹出栈，再对*value1*执行向右零扩展移位操作，所移位数由*value2*的最低5位（按照从0到31位的顺序）决定，最后将所得int类型结果（*result*）压入栈。

如果需要获取更多有关iushr指令的信息，请参考第13章。

ixor

对int类型值进行“逻辑异或”操作。

操作码字节值：130(0x82)

指令格式：ixor

栈：

前：……，*value1*，*value2*

后：……，*result*

描述：*value1*和*value2*（位于操作数栈顶部两个字长的数据）必须为int类型。要执行ixor指令，Java虚拟机首先将*value1*和*value2*弹出栈，进行“按位逻辑异或”操作，再将所得int类型结果（*result*）压入栈。

如果需要获取更多有关ixor指令的信息，请参考第13章。

jsr

跳转到子例程。

操作码字节值：168(0xa8)

指令格式：jsr，*branchbyte1*，*branchbyte2*

栈：

前：……

后：……，*address*

描述：要执行jsr指令，Java虚拟机首先将紧随jsr指令后的一个字长的操作码地址（程序计数器）*address*压入操作数栈。接着虚拟机通过计算(*branchbyte1*<<*8*) | *branchbyte2*来给出一个带符号16位偏移量。通过把计算得出的偏移量加到jsr操作码地址上，虚拟机计算出目标（程序计数器）地址。目标地址必须是与jsr指令处于同一个方法内的操作码地址。虚拟机将会跳转到目标地址，并在新位置继续执行。

如果需要获取更多有关jsr指令的信息，请参考第18章。

jsr_w

跳转到子例程（宽索引）。

操作码字节值：201(0xc9)

指令格式：jsr_w，*branchbyte1*，*branchbyte2*，*branchbyte3*，*branchbyte4*

栈：

前：……

后：……，*address*

描述：要执行jsr_w指令，Java虚拟机首先将紧随jsr_w指令后的一个字长的操作码地址（程序计数器）*address*压入操作数栈。接着虚拟机通过计算(*branchbyte1*<<*24*) | (*branchbyte2*<<*16*) | (*brachbyte3*<<*8*) | *branchbype4*来给出一个带符号32位偏移量。通过把计算得出的偏移量加到jsr_w操作码地址上，虚拟机计算出目标（程序计数器）地址。目标地址必须是与jsr_w指令处于同一个方法内的操作码地址。虚拟机将会跳转到目标地址，并在新位置继续执行。

需要注意的是，尽管jsr_w指令可以使用32位偏移量，但由于当前（1.0和1.1版本）Java class文件格式中以下三项的原因，把Java方法限制在65 535字节内：LineNumberTable属性、LocalVariableTable属性以及Code属性的exception_table项中索引的大小。根据Java虚拟机规范，对Java方法的65 535字节限制可能会在将来的版本中解决。如果需要获取更多有关jsr_w指令的信息，请参考第18章。

l2d

把long类型的数据转换为double类型。

操作码字节值：138(0x8a)

指令格式：l2d

栈：

前：……，*value.word1*，*value.word2*

后：……，*result.word1*，*result.word2*

描述：位于操作数栈顶部的两个字长的数据必须为long类型。要执行l2d指令，Java虚拟机首先从操作数栈中弹出一个long类型值（*value*），然后使用IEEE的“四舍五入”模式将其转换成为double类型，最后将得到的double类型结果（*result*）压入栈。

需要注意的是，这条指令执行的是放宽的基本类型转换。因为并不是所有long类型值都能准确无误地

转换为double类型，转换有可能导致数值和精度的丢失。

参考第11章可以获取更多有关l2d指令的信息。

l2f

把long类型的数据转换为float类型。

操作码字节值：137(0x89)

指令格式：l2f

栈：

前：……，*value.word1*，*value.word2*

后：……，*result*

描述：位于操作数栈顶部的两个字长的数据必须为long类型。要执行l2f指令，Java虚拟机首先从操作数栈中弹出一个*long*类型值（*value*），然后使用IEEE的“四舍五入”模式将其转换成为float类型，最后将得到的float类型结果（*result*）压入栈。

需要注意的是，这条指令执行的是放宽的基本类型转换。因为并不是所有long类型值都能准确无误地转换为float类型，转换有可能导致数值和精度的丢失。

参考第11章可以获取更多有关l2f指令的信息。

l2i

把long类型的数据转换为int类型。

操作码字节值：136(0x88)

指令格式：l2i

栈：

前：……，*value.word1*，*value.word2*

后：……，*result*

描述：位于操作数栈顶部的两个字长的数据必须为long类型。要执行l2i指令，Java虚拟机首先从操作数栈中弹出一个long类型值（*value*），然后将long类型值截短为int类型值，最后将得到的int类型结果（*result*）压入栈。

需要注意的是，这条指令执行的是放宽的基本类型转换。转换有可能导致数量信息的丢失以及符号的改变。

参考第11章可以获取更多有关l2i指令的信息。

ladd

执行long类型的加法。

操作码字节值：97(0x61)

指令格式：ladd

栈：

前：……，*value1.word1*，*value1.word2*，*value2.word1*，*value2.word2*

后：……，*result.word1*，*result.word2*

描述：*value1*和*value2*（位于操作数栈顶部四个字长的数据）必须为long类型。要执行ladd指令，Java虚拟机首先将*value1*和*value2*弹出栈，相加，再将所得long类型结果（*result*）压入栈。如果发生溢出，取使用足够长度的二进制补码形式表示的实际数学运算结果的低64位，因此结果的符号可能与实际的数学运算结果不一致。

如果需要获取更多有关ladd指令的信息，请参考第12章。

laload

从数组中读取long类型的数据。

操作码字节值：47(0x2f)

指令格式：laload

栈：

前：……，*arrayref*，*index*

后：……，*value.word1*，*value.word2*

描述：要执行laload指令，Java虚拟机首先从操作数栈中弹出两个字长的数据（*arrayref*和*index*）。*arrayref* 必须为指向long类型数组的引用类型。*index*必须为int类型。虚拟机从*arrayref* 所指向的数组获取由索引*index*指定的long类型值（*value*），然后将其压入操作数栈。

如果*arrayref* 引用的数组为null，虚拟机将会抛出NullPointerException异常。另外，如果*index*不是*arrayref* 数组的有效索引，虚拟机将会抛出ArrayIndexOutOfBoundsException异常。

如果需要获取更多有关laload指令的信息，请参考第15章。

land

对long类型值进行“逻辑与”操作。

操作码字节值：127(0x7f)

指令格式：land

栈：

前：……，*value1.word1*，*value1.word2*，*value2.word1*，*value2.word2*

后：……，*result.word1*，*result.word2*

描述：*value1*和*value2*（位于操作数栈顶部四个字长的数据）必须为long类型。要执行land指令，Java虚拟机首先将*value1*和*value2*弹出栈，进行“按位逻辑与”操作，再将所得long类型结果（*result*）压入栈。

如果需要获取更多有关land指令的信息，请参考第13章。

lastore

将long类型的数据存入数组中。

操作码字节值：80(0x50)

指令格式：lastore

栈：

前：……，*arrayref*，*index*，*value.word1*，*value.word2*

后：……

描述：要执行lastore指令，Java虚拟机首先从操作数栈中弹出四个字长的数据（*arrayref*，*index*，*value.word1*，*value.word2*）。*arrayref*必须为指向long数组的引用类型。*index*的数据类型必须为int，*value*必须为long类型。虚拟机将long类型值*value*存入由*index*所指定的*arrayref*指向的数组。

如果*arrayref*引用的数组为null，虚拟机将会抛出NullPointerException异常。如果*arrayref*引用的数组不为null，但*index*不是*arrayref*数组的有效索引，虚拟机将会抛出Array*Index*OutOf BoundsException异常。

如果需要获取更多有关lastore指令的信息，请参考第15章。

lcmp

比较long类型值。

操作码字节值：148 (0x94)

指令格式：lcmpl

栈：

前：……，*value1.word1*，*value1.word2*，*value2.word1*，*value2.word2*

后：……，*result*

描述：*value1*和*value2*（位于操作数栈顶部四个字长的数据）必须为long类型。要执行lcmpl指令，Java虚拟机首先从操作数栈中弹出*value1*和*value2*，然后对它们进行比较。如果*value1*与*value2*相等，虚拟机向操作数栈中压入int类型结果（*result*）0；如果*value1*大于*value2*，虚拟机向操作数栈中压入int类型结果1；如果*value1*小于*value2*，虚拟机向操作数栈中压入int类型结果-1。

如果需要获取更多有关lcmpl指令的信息，请参考第16章。

lconst_0

将long类型常量0压入栈。

操作码字节值：9(0x9)

指令格式：lconst_0

栈：

前：……

后：……，*<0>-word1*，*<0>-word2*

描述：要执行lconst_0指令，Java虚拟机将long类型常量0压入操作数栈。

如果需要获取更多有关lconst_0指令的信息，请参考第10章。

lconst_1

将long类型常量1压入栈。

操作码字节值：10(0xa)

指令格式：lconst_1

栈：

前：……

后：……，*<1>-word1*，*<1>-word2*

描述：要执行lconst_1指令，Java虚拟机将long类型常量1压入操作数栈。

如果需要获取更多有关lconst_1指令的信息，请参考第10章。

ldc

把常量池中的项压入栈。

操作码字节值：18(0x12)

指令格式：ldc，*index*

栈：

前：……

后：……，*item*

描述：操作数*index*必须为一个指向当前常量池的有效的无符号8位索引。要执行ldc指令，Java虚拟机首先查找*index*所指定的常量池入口。在*index*指向的常量池入口，虚拟机将会查找CONSTANT_Integer_info、CONSTANT_Float_info或者CONSTANT_String_info入口。如果还没有这些入口的话，虚拟机会解析它们。如果该入口为CONSTANT_Integer_info，虚拟机将把由该入口表示的int类型值压入操作数栈。如果该入口为CONSTANT_Float_info，虚拟机将把由该入口表示的float类型值压入操作数栈。如果该入口为CONSTANT_String_info，虚拟机将把指向被拘留String对象（由解析该入口的进程产生）的引用压入操作数栈。

需要注意的是，ldc_w执行的是与ldc相同的功能，不同之处在于ldc_w提供了常量池宽（16位）索引。

如果需要获取更多有关ldc指令的信息，请参考第10章。

ldc_w

把常量池中的项压入栈（使用宽索引）。

操作码字节值：19(0x13)

指令格式：ldc_w，*indexbyte1*，*indexbyte2*

栈：

前：……

后：……，*item*

描述：要执行ldc_w指令，Java虚拟机首先通过计算(*indexbyte1*<<*8*) | *indexbyte2*来生成一个指向当前常量池的无符号16位索引。然后虚拟机查找被该计算出的索引指定的常量池入口。在该索引指向的常量池入口，虚拟机将会查找CONSTANT_Integer_info、CONSTANT_Float_info或者CONSTANT_String_info入口。如果还没有这些入口的话，虚拟机会解析它们。如果该入口为CONSTANT_Integer_info，虚拟机将把由该入口表示的int类型值压入操作数栈。如果该入口为CONSTANT_Float_info，虚拟机将会把由该入口表示的float类型值压入操作数栈。如果该入口为CONSTANT_String_info，虚拟机将会把指向被拘留String对象（由解析该入口的进程产生）的引用压入操作数栈。

需要注意的是，ldc_w执行的是与ldc相同的功能，不同之处在于ldc_w提供了常量池宽（16位）索引，

而ldc只提供8位的常量池索引。

如果需要获取更多有关ldc_w指令的信息，请参考第10章。

ldc2_w

把常量池中long类型或者double类型的项压入栈（使用宽索引）。

操作码字节值：20(0x14)

指令格式：ldc2_w，*indexbyte1*，*indexbyte2*

栈：

前：……

后：……，*item.word1*，*item.word2*

描述：要执行ldc2_w指令，Java虚拟机首先通过计算(*indexbyte1*<<*8*) | *indexbyte2*来生成一个指向当前常量池的无符号16位索引。然后虚拟机查找被该计算出的索引指定的常量池入口。在该索引指向的常量池入口，虚拟机将会查找CONSTANT_Long_info或者CONSTANT_Double_info入口。如果还没有入口的话，虚拟机会解析它们。如果该入口为CONSTANT_Long_info，虚拟机将把由该入口表示的long类型值压入操作数栈。如果该入口为CONSTANT_Double_info，虚拟机将把由该入口表示的double类型值压入操作数栈。

需要注意的是，没有与ldc2_w执行相同功能，但只提供8位常量池索引的“ldc2”指令。所有从常量池中取出的两个字长的常量的操作，都必须使用拥有常量池宽（16位）索引的ldc2_w指令。

如果需要获取更多有关ldc2_w指令的信息，请参考第10章。

ldiv

执行long类型的除法。

操作码字节值：109(0x6d)

指令格式：ldiv

栈：

前：……，*value1.word1*，*value1.word2*，*value2.word1*，*value2.word2*

后：……，*result.word1*，*result.word2*

描述：*value1*和*value2*（位于操作数栈顶部四个字长的数据）必须为long类型。要执行ldiv指令，Java虚拟机首先将*value1*和*value2*弹出栈，执行整数除法操作（*value1*/*value2*），再将所得long类型结果（*result*）压入栈。

整数除法对所得的实际数学运算结果（商）取整。如果除数大于被除数，那么结果必然为0。当除数和被除数符号相同时，所得结果的符号为正，当除数和被除数符号相异时，所得结果的符号为负。下面是该规则的一种例外情况：当除数是能够使用long数据类型表示的最小负整数且被除数为-1时，该除法的实际数学运算结果是一个能被long数据类型所表示的最大正整数还要大1的数。因此，除法发生溢出，结果等于被除数。

如果*value2*（除数）为0，Java虚拟机将抛出ArithmeticException异常。

如果需要获取有关ldiv指令的更多信息，请参考第12章。

lload

从局部变量中装载long类型值。

操作码字节值：22(0x16)

指令格式：lload，*index*

栈：

前：……

后：……，*value1.word1*，*value1.word2*

描述：作为一个对当前栈帧中局部变量的8位无符号索引，操作数*index*必须指向一个包含long类型数据的两个连续局部变量字的第一个。要执行lload指令，Java虚拟机将*index*和*index+1*所指向的两个连续局部变量字中的long类型值（*value*）压入操作数栈。

需要注意的是，lload指令前面可以使用wide指令，这样就能够访问一个由16位无符号偏移量所指向的局部变量。

如果需要获取有关lload指令的更多信息，请参考第10章。

lload_0

从局部变量0中装载long类型值。

操作码字节值：30(0x1e)

指令格式：lload_0

栈：

前：……

后：……，*value1.word1*，*value1.word2*

描述：在索引0和1位置的两个连续的局部变量字必须包含一个long类型值。要执行lload_0指令，Java虚拟机将局部变量字0和1中存储的long类型值（*value*）压入操作数栈。

如果需要获取有关lload_0指令的更多信息，请参考第10章。

lload_1

从局部变量1中装载long类型值。

操作码字节值：31(0x1f)

指令格式：lload_1

栈：

前：……

后：……，*value1.word1*，*value1.word2*

描述：在索引1和2位置的两个连续的局部变量字必须包含一个long类型值。要执行lload_1指令，Java虚拟机将局部变量字1和2中存储的long类型值（*value*）压入操作数栈。

如果需要获取有关lload_1指令的更多信息，请参考第10章。

lload_2

从局部变量2中装载long类型值。

操作码字节值：32(0x20)

指令格式：lload_2

栈：

前：……

后：……，*value1.word1*，*value1.word2*

描述：在索引2和3位置的两个连续的局部变量字必须包含一个long类型值。要执行lload_2指令，Java虚拟机将局部变量字2和3中存储的long类型值（*value*）压入操作数栈。

如果需要获取有关lload_2指令的更多信息，请参考第10章。

lload_3

从局部变量3中装载long类型值。

操作码字节值：33(0x21)

指令格式：lload_3

栈：

前：……

后：……，*value1.word1*，*value1.word2*

描述：在索引3和4位置的两个连续的局部变量字必须包含一个long类型值。要执行lload_3指令，Java虚拟机将局部变量字3和4中存储的long类型值（*value*）压入操作数栈。

如果需要获取有关lload_3指令的更多信息，请参考第10章。

lmul

执行long类型的乘法。

操作码字节值：105(0x69)

指令格式：lmul

栈：

前：……，*value1.word1*，*value1.word2*，*value2.word1*，*value2.word2*

后：……，*result.word1*，*result.word2*

描述：*value1*和*value2*（位于操作数栈顶部四个字长的数据）必须为long类型。要执行lmul指令，Java虚拟机首先将*value1*和*value2*弹出栈，相乘，再将所得long类型结果（*result*）压入栈。如果发生溢出，取使用足够长度的二进制补码形式表示的实际数学运算结果的低64位，因此结果的符号可能与实际的数学运算结果不一致。

如果需要获取更多有关lmul指令的信息，请参考第12章。

lneg

对一个long类型值进行取反操作。

操作码字节值：117(0x75)

指令格式：lneg

栈：

前：……，*value.word1*，*value.word2*

后：……，*result.word1*，*result.word2*

描述：位于操作数栈顶部两个字长的数据必须为long类型。要执行lneg指令，Java虚拟机首先将*value*弹出栈，执行取反操作，再将所得long类型结果（*result*）压入栈。

执行lneg指令所得到的结果与使用lsub指令用0减去*value*所得结果相同。因而，当*value*为可以用long数据类型表示的最小负整数时，对其进行的取反操作会发生溢出。对于这样的取反操作，实际数学运算结果是一个比用long数据类型表示的最大正整数还要大1的数值，而所得到的实际结果是原来*value*的值，它的符号并未改变。

如果需要获取更多有关lneg指令的信息，请参考第12章。

lookupswitch

通过键值匹配访问跳转表，并执行跳转操作。

操作码字节值：171(0xab)

指令格式：lookupswtich，… 0～3个填充字节…，*defaultbyte1*，*defaultbyte2*，*defaultbyte3*，*defaultbyte4*，*npair1*，*npair2*，*npair3*，*npair4*，…*match-offsetpairs*…

栈：

前：……，*key*

后：……

描述：紧随在操作码lookupswitch后面的是3个填充字节——这样就使得紧随在填充字节后面的内容能够从一个以4字节倍数对齐的地址开始（从方法起始位置开始计算）。每个填充字节的内容都是0。紧随在填充字节后面的*default*是一个带符号32位默认分支偏移量。在*default*后面的是名为*npairs*的带符号计数器，该计数器对嵌在该lookupswitch指令中的“case值/分支偏移量对”进行计数。*npairs*的值必须大于等于0。在npairs后面的是“case值/分支偏移量对”。对于每一对，前面的*match*是带符号32位case值，后面的offset是带符号32位分支偏移量。虚拟机通过下述4个独立字节计算出所有这些带符号的32位值：(*byte1*<<*24*) | (*byte2*<<*16*) | (*byte3*<<*8*) | *byte4*。

key（位于操作数栈顶部一个字长的数据）必须为int类型。要执行lookupswitch指令，Java虚拟机首先将*key*弹出操作数栈，将其与*match*值进行比较，如果*key*与*match*值中的一个相等，通过把对应于匹配的*match*值的带符号的*offset*加到lookupswitch操作码地址上，虚拟机计算出目标（程序计数器）地址。目标地址必须是与lookupswitch指令处于同一个方法内的操作码地址。虚拟机将会跳转到目标地址，然后从那里继续执行。

如果需要获取更多有关lookupswitch指令的信息，请参考第16章。

lor

对long类型值进行“逻辑或”操作。

操作码字节值：129(0x81)

指令格式：lor

栈：

前：……，*value1.word1*，*value1.word2*，*value2.word1*，*value2.word2*

后：……，*result.word1*，*result.word2*

描述：*value1*和*value2*（位于操作数栈顶部四个字长的数据）必须为long类型。要执行lor指令，Java虚拟机首先将*value1*和*value2*弹出栈，进行“按位逻辑或”操作，再将所得long类型结果（*result*）压入栈。

如果需要获取更多有关lor指令的信息，请参考第13章。

lrem

计算long类型除法的余数。

操作码字节值：113(0x71)

指令格式：lrem

栈：

前：……，*value1.word1*，*value1.word2*，*value2.word1*，*value2.word2*

后：……，*result.word1*，*result.word2*

描述：*value1*和*value2*（位于操作数栈顶部四个字长的数据）必须为long类型。要执行lrem指令，Java虚拟机首先将*value1*和*value2*弹出栈，计算整数余数，再将所得long类型结果（*result*）压入栈。取余操作等同于*value1* -（*value1* /*value2*）* *value2*。

lrem指令实现了Java的取余操作符（%）。指令lrem的特性使得下列Java表达式的值永远为真（n和d为任意两个long类型值）：

```
(n/d)*d + (n%d) == n
```

该特性意味着，指令lrem的结果总是与被除数（从栈中弹出的*value1*）的符号保持一致。

如果*value2*（除数）为0，Java虚拟机将抛出ArithmeticException异常。

如果需要获取更多有关lrem指令的信息，请参考第12章。

lreturn

从方法中返回long类型的数据。

操作码字节值：173(0xad)

指令格式：lreturn

栈：

前：……，*value.word1*，*value.word2*

后：[空]

描述：方法返回值的数据类型必须为long。位于栈顶部的两个字长的数据必须为long类型。要执行lreturn指令，Java虚拟机从当前栈帧内的操作数栈中弹出long类型值（*value*），然后将其压入调用方法栈帧内的操作数栈中。虚拟机将抛弃所有返回方法栈帧中仍然可能存在的值。如果返回方法是同步的，那么该方法被调用时所获得的监视器将被释放，调用方法栈帧被设为当前值，虚拟机继续执行调用方法中的语句。

如果当方法被调用时，当前方法是同步的，而且当前线程不持有已被获取和重入的监视器，虚拟机将会抛出IllegalMonitorStateException异常。否则，如果虚拟机实现强制执行在monitorexit指令中所描述的结构化锁定规则，虚拟机将会抛出IllegalMonitorStateException异常。

如果需要获取更多有关lreturn指令的信息，请参考第19章。如果需要获取更多有关监视器的信息，请参考第20章。

lshl

执行long类型的向左移位操作。

操作码字节值：121(0x79)

指令格式：lshl

栈：

前：……，*value1.word1*，*value1.word2*，*value2*

后：……，*result.word1*，*result.word2*

描述：*value2*（位于操作数栈顶部一个字长的数据）必须为int类型，*value1*（接下来两个字长的数据）必须为long类型。要执行lshl指令，Java虚拟机首先将*value1*和*value2*弹出栈，再对*value1*执行向左移位操作，所移位数由*value2*的最低6位（按照从0到63位的顺序）决定，最后将所得long类型结果（*result*）压入栈。

如果需要获取更多有关lshl指令的信息，请参考第13章。

lshr

执行long类型的算术向右移位操作。

操作码字节值：123(0x7b)

指令格式：lshr

栈：

前：……，*value1.word1*，*value1.word2*，*value2*

后：……，*result.word1*，*result.word2*

描述：*value2*（位于操作数栈顶部一个字长的数据）必须为int类型，*value1*（接下来两个字长的数据）必须为long类型。要执行lshr指令，Java虚拟机首先将*value1*和*value2*弹出栈，再对*value1*执行向右符号扩展移位操作，所移位数由*value2*的最低6位（按照从0到63位的顺序）决定，最后将所得long类型结果（*result*）压入栈。

如果需要获取更多有关lshr指令的信息，请参考第13章。

lstore

将long类型值存入局部变量中。

操作码字节值：55(0x37)

指令格式：lstore，*index*

栈：

前：……，*value.word1*，*value.word2*

后：……

描述：操作数*index*必须指明一个指向当前栈帧内局部变量的有效的8位无符号索引。位于操作数栈顶部的两个字长的数据必须为long类型。要执行lstore指令，Java虚拟机从操作数栈顶部弹出long类型值（*value*），然后将该值存入由索引*index*和*index+1*所指定的两个连续局部变量字中。

需要注意的是，lstore指令前面可以使用wide指令，这样就能够将弹出的值（*value*）存到为16位无符号偏移量所指定的局部变量中。

如果需要获取更多有关lstore指令的信息，请参考第10章。

lstore_0

将long类型值存入局部变量0中。

操作码字节值：63(0x3f)

指令格式：lstore_0

栈：

前：……，*value.word1*，*value.word2*

后：……

描述：索引0和1必须是一个指向当前栈帧内局部变量的有效索引。位于操作数栈顶部的两个字长的数据必须为long类型。要执行lstore_0指令，Java虚拟机从操作数栈顶部弹出一个long类型值（*value*），然后将该值存入由索引0和1所指定的两个连续局部变量字中。

如果需要获取更多有关lstore_0指令的信息，请参考第10章。

lstore_1

将long类型值存入局部变量1中。

操作码字节值：64(0x40)

指令格式：lstore_1

栈：

前：……，*value.word1*，*value.word2*

后：……

描述：索引1和2必须是一个指向当前栈帧内局部变量的有效索引。位于操作数栈顶部的两个字长的数据必须为long类型。要执行lstore_1指令，Java虚拟机从操作数栈顶部弹出一个long类型值（*value*），然后将该值存入由索引1和2所指定的两个连续局部变量字中。

如果需要获取更多有关lstore_1指令的信息，请参考第10章。

lstore_2

将long类型值存入局部变量2中。

操作码字节值：65(0x41)

指令格式：lstore_2

栈：

前：……，*value.word1*，*value.word2*

后：……

描述：索引2和3必须是一个指向当前栈帧内局部变量的有效索引。位于操作数栈顶部的两个字长的数据必须为long类型。要执行lstore_2指令，Java虚拟机从操作数栈顶部弹出一个long类型值（*value*），然后将该值存入由索引2和3所指定的两个连续局部变量字中。

如果需要获取更多有关lstore_2指令的信息，请参考第10章。

lstore_3

将long类型值存入局部变量3中。

操作码字节值：66(0x42)

指令格式：lstore_3

栈：

前：……，*value.word1*，*value.word2*

后：……

描述：索引3和4必须是一个指向当前栈帧内局部变量的有效索引。位于操作数栈顶部的两个字长的数据必须为long类型。要执行lstore_3指令，Java虚拟机从操作数栈顶部弹出一个long类型值（*value*），然后将该值存入由索引3和4所指定的两个连续局部变量字中。

如果需要获取更多有关lstore_3指令的信息，请参考第10章。

lsub

执行long类型的减法。

操作码字节值：101(0x65)

指令格式：lsub

栈：

前：……，*value1.word1*，*value1.word2*，*value2.word1*，*value2.word2*

后：……，*result.word1*，*result.word2*

描述：*value1*和*value2*（位于操作数栈顶部四个字长的数据）必须为long类型。要执行lsub指令，Java虚拟机首先将*value1*和*value2*弹出栈，用*value1*减去*value2*（*value1* –*value2*），再将所得long类型结果（*result*）压入栈。如果发生溢出，取使用足够长度的二进制补码形式表示的实际数学运算结果的低64位，因此结果的符号可能与实际的数学运算结果不一致。

如果需要获取更多有关lsub指令的信息，请参考第12章。

lushr

执行long类型的向右逻辑移位操作。

操作码字节值：125(0x7d)

指令格式：lushr

栈：

前：……，*value1.word1*，*value1.word2*，*value2*（此处原书中有印刷错误。——译者注）

后：……，*result.word1*，*result.word2*

描述：*value2*（位于操作数栈顶部一个字长的数据）必须为int类型，*value1*（接下来两个字长的数据）必须为long类型。要执行lushr指令，Java虚拟机首先将*value1*和*value2*弹出栈，再对*value1*执行向右零扩展移位操作，所移位数由*value2*的最低6位（按照从0到63位的顺序）决定，最后将所得long类型结果（*result*）压入栈。

如果需要获取更多有关lushr指令的信息，请参考第13章。

lxor

对long类型值进行“逻辑异或”操作。

操作码字节值：131(0x83)

指令格式：lxor

栈：

前：……，*value1.word1*，*value1.word2*，*value2.word1*，*value2.word2*

后：……，*result.word1*，*result.word2*

描述：*value1*和*value2*（位于操作数栈顶部四个字长的数据）必须为long类型。要执行lxor指令，Java虚拟机首先将*value1*和*value2*弹出栈，进行“按位逻辑异或”操作，再将所得long类型结果（*result*）压入栈。

如果需要获取更多有关lxor指令的信息，请参考第13章。

monitorenter

进入并获得对象监视器。

操作码字节值：194(0xc2)

指令格式：monitorenter

栈：

前：……，*objectref*

后：……

描述：*objectref*（位于操作数栈顶部一个字长的数据）必须为引用类型。要执行monitorenter指令，Java虚拟机将代表当前线程的*objectref*弹出栈，获取与*objectref*相关联的监视器。

如果*objectref*字为null，虚拟机将抛出NullPointerException异常。

如果需要获取更多有关monitorenter指令的信息，请参考第20章。

monitorexit

释放并退出对象监视器。

操作码字节值：195(0xc3)

指令格式：monitorexit

栈：

前：……，*objectref*

后：……

描述：*objectref*（位于操作数栈顶部一个字长的数据）必须为引用类型。要执行monitorexit指令，Java虚拟机首先弹出代表当前线程的*objectref*，然后释放并退出与*objectref*相关联的监视器。

尽管Java编译器通常应该为线程执行的对象锁定操作产生相对应的解锁代码，但并不保证Java虚拟机所装载的代码将一直拥有这个性质。因此，Java虚拟机实现允许（尽管不需要）在锁定的结构化使用上强制实行这两条规则：

1) 在特定线程调用方法的过程中，由在特定对象的监视器上的线程执行的锁定操作的数量必须等于解锁操作的数量，无论方法是否正常结束。

2) 在特定线程调用方法的过程中，由在特定对象的监视器上的线程执行的解锁操作的数量决不会超过锁定操作的数量。

需要注意的是，当调用并完成同步方法时，锁定和解锁操作可能会在调用执行调用操作的方法（不是被调用方法）的过程中执行。

如果*objectref*字为null，虚拟机将会抛出NullPointerException异常。如果当前线程并不拥有监视器，虚拟机将抛出IllegalMonitorStateException异常。如果虚拟机实现强制执行前述结构化锁定规则，并且指令执行时违背了这两条规则，虚拟机将抛出IllegalMonitorStateException异常。

如果需要获取更多有关monitorexit指令的信息，请参考第20章。

multianewarray

创建新的多维数组。

操作码字节值：197(0Xc5)

指令格式：multianewarray，*indexbyte1*，*indexbyte2*，*dimensions*

栈：

前：……，*count1*，*[count2*，……*]*

后：*arrayref*

描述：无符号byte类型操作数*dimensions*指出了所要创建的数组的维数，它必须大于等于1。*count1*，*[count2*，……*]*指出数组各维的长度，它们必须是正整数。例如，对于如下定义的数组：

```
int[][][] example = new int[3][4][5];
```

对此数组而言：*dimensions*的值应该为3，操作数栈将会包含三个字长的“count”值。*count1*的值为3，*count2*的值为4，*count3*的值为5。

要执行multianewarray指令，Java虚拟机首先通过计算(*indexbyte1*<<*8*) | *indexbyte2*生成一个指向常量池的无符号16位索引。然后虚拟机根据计算出的索引来查找常量池入口。该索引所指向的常量池入口必须为CONSTANT_Class_info。如果该入口尚未存在，那么虚拟机将会解析这个常量池入口，该入口必须为维数大于或者等于*dimensions*的数组类型。

如果解析成功，Java虚拟机将在堆中创建有*dimensions*维数的多维数组。虚拟机使用引用第二维数组的形式初始化第一维数组的成员，使用引用第三维数组的形式初始化第二维数组的成员，等等。虚拟机使用默认值初始化最后一维数组的成员。最后，Java虚拟机将新创建的*dimensions*维数组的引用压入操作数栈。

执行这条指令的结果是，虚拟机可能会在解析CONSTANT_Class_info入口的过程中抛出任意第8章中

列出的连接错误。如果解析成功，而且数组成员的类型为引用类型，但是当前类并没有访问这些引用的权限，虚拟机将会抛出IllegalAccessError异常。如果从操作数栈中弹出的count值存在负数，Java虚拟机将会抛出NegativeArraySizeException异常。

需要注意的是，所解析的CONSTANT_Class_info入口中所指定的数组类型维数并不一定与操作数*dimensions*相等，它也可以大于*dimensions*。例如，一个3维int类型数组的数组类名为“[[[I”，但由multianewarray指令创建的CONSTANT_Class_info入口中指定的数组维数至少为3，有可能比3还大。例如，“[[[I”、“[[[[I”和“[[[[[[[[I”名称都可以指向3维int类型数组，只要创建数组时，参数*dimensions*为3即可。对multianewarray指令而言，多维数组类型的规定有一定的灵活性，这种灵活性可以减少某些类所需要的常量池入口的数量。

如果需要获取更多有关multianewarray指令的信息，请参考第15章。

new

创建一个新对象。

操作码字节值：187(0xbb)

指令格式：new，*indexbyte1*，*indexbyte2*

栈：

前：……

后：*objectref*

描述：要执行new指令，Java虚拟机首先通过计算(*indexbyte1*<<*8*) | *indexbyte2*生成一个指向常量池的无符号16位索引。然后虚拟机根据计算出的索引查找常量池入口。该索引所指向的常量池入口必须为CONSTANT_Class_info。如果该入口尚未存在，那么虚拟机将解析这个常量池入口，该入口的类型必须是类，不能是接口或者数组。虚拟机从堆中为新对象映象分配足够大的空间，并将对象的实例变量设为默认值。最后，虚拟机将指向新对象的引用*objectref*压入操作数栈。

执行这条指令的结果是，虚拟机可能会在解析CONSTANT_Class_info入口的过程中抛出任意第8章中列出的连接错误。如果解析成功，但所解析的类型为接口、抽象类或者数组，虚拟机将会抛出InstantiationError异常。

如果需要获取更多有关new指令的信息，请参考第15章。

newarray

创建一个数组，其成员类型为基本数据类型。

操作码字节值：188(0xbc)

指令格式：newarray，*atype*

栈：

前：……，*count*

后：*arrayref*

描述：*count*（位于操作数栈顶部一个字长的数据）必须为int类型。用来指示数组类型的操作数*atype*必须在表A-2中所列的范围内取值。要执行newarray指令，Java虚拟机首先将*count*弹出栈，然后在堆中创

建一个长度为*count*、类型为*atype*所指定的基本类型数组。虚拟机使用默认值初始化数组内的每一个元素，并将指向新数组的引用*arrayref*压入操作数栈。

如果*count*小于0，Java虚拟机将会抛出NegativeArraySizeException异常。

如果需要获取更多有关newarray指令的信息，请参考第15章。

表A-2 atype的值

数组类型	atype
T_BOOLEAN	4
T_CHAR	5
T_FLOAT	6
T_DOUBLE	7
T_BYTE	8
T_SHORT	9
T_INT	10
T_LONG	11

nop

不做任何操作。

操作码字节值：0(0x0)

指令格式：nop

栈：无变化

描述：要执行nop指令，Java虚拟机将“休息”一会儿。

如果需要获取更多有关nop指令的信息，请参考第10章。

pop

弹出栈顶端一个字长的内容。

操作码字节值：87(0x57)

指令格式：pop

栈：

前：……，*word*

后：……

描述：要执行pop指令，Java虚拟机从操作数栈中弹出一个字长的内容。所有从栈顶端弹出一个字长内容的操作都可以使用这条指令。从栈顶端去除双字值（long或者double类型）的一半不能使用这条指令。

如果需要获取更多有关pop指令的信息，请参考第10章。

pop2

弹出栈顶端两个字长的内容。

操作码字节值：88(0x58)

指令格式：pop2

栈：

前：……，*word2*，*word1*

后：……

描述：要执行pop2指令，Java虚拟机从操作数栈中弹出两个字长的内容。所有从栈顶端弹出两个字长内容的操作都可以使用这条指令。从栈顶端去除一个字长或者半个双字（long或者double类型）长度的内容不能使用这条指令。

如果需要获取更多有关pop2指令的信息，请参考第10章。

putfield

设置对象中字段的值。

操作码字节值：181(0xb5)

指令格式：pubfield，*indexbyte1*，*indexbyte2*

栈：

前：……，*objectref*，*value*

后：……

或者

前：……，*objectref*，*value.word1*，*value.word2*

后：……

描述：要执行putfield指令，Java虚拟机首先通过计算(*indexbyte1*<<8) | *indexbyte2*生成一个指向常量池的无符号16位索引。然后虚拟机根据计算出的索引来查找常量池入口。该索引所指向的常量池入口必须为CONSTANT_Fieldref_info。如果该入口尚未存在，那么虚拟机将会解析这个常量池入口，这个过程中将会产生字段的宽度和字段的偏移量（从对象映射的起始处开始计算）等信息。

占据栈顶部一个字长或者两个字长空间的*value*必须和所解析字段的描述符相兼容。如果所解析字段的描述符为byte、short、char、boolean或者int类型，那么*value*也必须为int类型。如果所解析字段的描述符为long类型，那么*value*也必须为long类型。如果所解析字段的描述符为float类型，那么*value*也必须为float类型。如果所解析字段的描述符为double类型，那么*value*也必须为double类型。如果所解析字段的描述符为引用类型，那么*value*也必须为引用类型，而且必须与所解析的描述符类型的赋值相兼容。单字长度的*objectref*必须为一个引用类型。

虚拟机把*value*和*objectref*从栈中弹出，并将值*value*赋给由*objectref*所指向对象中的恰当的字段。

执行这条指令的结果是，虚拟机可能会在解析CONSTANT_Fieldref_info入口的过程中抛出任意第8章中列出的连接错误。在解析CONSTANT_Fieldref_info入口的过程中，虚拟机会检查此字段的访问权限，判断当前类是否能够访问该字段。如果为受保护的字段，虚拟机将会确认该字段是否为当前类或者当前类的超类的字段，以及由objectref所指向的对象的类是否为当前类或当前类的子类。如果不是的话（或者有任何访问权限问题），虚拟机将会抛出IllegalAccessError异常。如果存在该字段，并且能够为当前类所访问，但是该字段是静态字段，虚拟机将会抛出IncompatibleClassChangeError异常。如果该字段为final字段，那么它必须是在当前类中声明的字段，否则虚拟机将会抛出IllegalAccessError异常。如果*objectref*的值为null，

那么虚拟机将会抛出NullPointerException异常。

如果需要获取更多有关putfield指令的信息，请参考第15章。

putstatic

设置类中静态字段的值。

操作码字节值：179(0xb3)

指令格式： putstatic，*indexbyte1*，*indexbyte2*

栈：

前：……，*value*

后：……

或者

前：……，*value.word1*，*value.word2*

后：……

描述：要执行putstatic指令，Java虚拟机首先通过计算(*indexbyte1*<<*8*) | *indexbyte2*生成一个指向常量池的无符号16位索引。然后虚拟机根据计算出的索引来查找常量池入口。该索引所指向的常量池入口必须为CONSTANT_Fieldref_info。如果该入口尚未存在，那么虚拟机将会解析这个常量池入口。

占据栈顶部一个字长或者两个字长空间的*value*必须和所解析字段的描述符相兼容。如果所解析字段的描述符为byte、short、char、boolean或者int类型，那么*value*也必须为int类型。如果所解析字段的描述符为long类型，那么*value*也必须为long类型。如果所解析字段的描述符为float类型，那么*value*也必须为float类型。如果所解析字段的描述符为double类型，那么*value*也必须为double类型。如果所解析字段的描述符为引用类型，那么*value*也必须为引用类型，而且必须与所解析字段的描述符类型的赋值相兼容。

虚拟机把*value*从栈中弹出，并将其赋给适当的静态字段。

执行这条指令的结果是，虚拟机可能会在解析CONSTANT_Fieldref_info入口的过程中抛出任意第8章中列出的连接错误。在解析CONSTANT_Fieldref_info入口的过程中，虚拟机会检查此字段的访问权限，判断当前类是否能够访问该字段。如果为受保护的字段，虚拟机将会确认该字段是否为当前类或者当前类的超类的字段。如果不是的话（或者有任何访问权限问题），虚拟机将会抛出IllegalAccessError异常。此外，如果存在该字段，并且能够为当前类所访问，但是该字段不是静态字段，虚拟机将会抛出IncompatibleClassChangeError异常。

如果需要获取更多有关putstatic指令的信息，请参考第15章。

ret

从子例程中返回。

操作码字节值：169(0xa9)

指令格式： ret，*index*

栈：无变化

描述：操作数*index*是一个指向局部变量的8位无符号偏移量。由*index*指定的局部变量字必须是一个returnAddress。要执行ret指令，Java虚拟机首先把程序计数器设为存储在*index*指定的局部变量中的

returnAddress值，然后从那里继续执行（换言之，虚拟机跳转到returnAddress）。

需要注意的是，ret指令前面可以使用wide指令，这样就能够通过16位无符号偏移量访问局部变量。

如果需要获取更多有关ret指令的信息，请参考第18章。

return

从方法中返回，返回值为void。

操作码字节值：177(0xb1)

指令格式：return

栈：

前：……

后：……[空]

描述：使用这条指令时，返回方法的返回类型必须为void。要执行return指令，Java虚拟机忽略所有返回方法栈帧中可能存在的值。如果返回方法是同步的，那么该方法被调用时所获得的监视器将被释放，调用方法栈帧被设为当前值，虚拟机继续执行调用方法中的语句。

如果当方法被调用的时候，当前方法是同步的，而且当前线程不持有已被获取和重入的监视器，虚拟机将会抛出IllegalMonitorStateException异常。否则，如果虚拟机实现强制执行在monitorexit指令中所描述的结构化锁定的规则，虚拟机将会抛出IllegalMonitorStateException异常。

如果需要获取更多有关监视器的信息，请参考第20章。如果需要获取更多有关return指令的信息，请参考第19章。

saload

从数组中装载short类型的数据。

操作码字节值：53 (0x35)

指令格式：saload

栈：

前：……，*arrayref*，*index*

后：……，*value*

描述：要执行saload指令，Java虚拟机首先从操作数栈中弹出两个字长的数据（*arrayref*和*index*）。*arrayref*必须为指向short类型数组的引用。*index*必须为int类型。虚拟机从*arrayref*所指向的数组获取*index*所指定的short类型值（*value*），将其符号扩展为int类型，然后压入操作数栈。

如果*arrayref*的值为null，虚拟机将会抛出NullPointerException异常。另外，如果*index*不是*arrayref*数组的有效索引，虚拟机将会抛出Array*Index*OutOfBoundsException异常。

如果需要获取更多有关saload指令的信息，请参考第15章。

sastore

将short类型的数据存入数组中。

操作码字节值：86(0x56)

指令格式：sastore

栈：

前：……，*arrayref*，*index*，*value*

后：……

描述：要执行sastore指令，Java虚拟机首先从操作数栈中弹出三个字长的数据（*arrayref*、*index*、*value*）。*arrayref*必须为指向short数组的引用类型。*index*和*value*的数据类型必须为int。虚拟机先将int类型值*value*截短为short类型值，最后再将它存入*index*所指定的*arrayref*指向的数组。

如果*arrayref*的值为null，虚拟机将会抛出NullPointerException异常。如果*arrayref*的值不为null，但*index*不是*arrayref*数组的有效索引，虚拟机将会抛出ArrayIndexOutOfBoundsException异常。

如果需要获取更多有关sastore指令的信息，请参考第15章。

sipush

将16位带符号整数压入栈。

操作码字节值：17(0x11)

指令格式：sipush，*byte1*，*byte2*

栈：

前：……

后：……，*value*

描述：要执行sipush指令，Java虚拟机首先从*byte1*和*byte2*通过计算(*byte1*<<*8*) | *byte2*产生一个16位带符号整数作为中间变量。接下来虚拟机将这个中间变量符号扩展为一个int类型值（*value*），然后将此值压入操作数栈。

如果需要获取更多有关sipush指令的信息，请参考第10章。

swap

交换栈顶两个字的内容。

操作码字节值：95(0x5f)

指令格式：swap

栈：

前：……，*word2*，*word1*

后：……，*word1*，*word2*

描述：要执行swap指令，Java虚拟机交换操作数栈顶部两个字的内容。*word1*和*word2*都必须为单字长的值。

如果需要获取更多有关swap指令的信息，请参考第10章。

tableswitch

通过索引访问跳转表，并跳转。

操作码字节值：170(0xaa)

指令格式：

tableswtich，… 0～3个填充字节…，*defaultbyte1*，*defaultbyte2*，*defaultbyte3*，*defaultbyte4*，*lowbyte1*，*lowbyte2*，*lowbyte3*，*lowbyte4*，*highbyte1*，*highbyte2*，*highbyte3*，*highbyte4*，…*jump offsets* …

栈：

前：……，*index*

后：……

描述：紧随在操作码tableswitch后面的是3个填充字节——这样就使得紧随在填充字节后面的内容能够从一个以4字节倍数对齐的地址开始（从方法起始位置开始计算）。每个填充字节的内容都是0。紧随在填充字节后面的*default*是一个带符号32位默认分支偏移量。在*default*后面的是两个32位带符号的、内嵌在tableswitch指令中的case值端点值。*low*（低端点）的值必须小于或者等于*high*（高端点）的值。在*low*和*high*之后，是带符号32位分支偏移量,其数目为*high* − *low* + *1*。在这些偏移量中，有一个偏移量对应*low*，有一个偏移量对应*high*，其余的偏移量对应着low和high之间的每个值。这些偏移量用于从0开始的跳转表。跳转表中的第一项（对应与*low*的分支偏移量）紧接在端点值*high*之后。虚拟机通过下述公式从4个单独字节中计算出所有这些带符号的32位值：(*byte1*<<*24*) | (*byte2*<<*16*) | (*byte3*<<*8*) | *byte4*。

index（操作数栈顶端一个字长的内容）必须为int类型。要执行tableswitch指令，Java虚拟机首先将*index*弹出操作数栈，将其分别与*low*和*high*进行比较，然后选择分支偏移量。如果*index*小于*low*或者大于*high*的话，虚拟机选择*default*作为分支偏移量。如果*index*在*low*和*high*范围之间，虚拟机将会选择跳转表中处于*index*−*low*位置的分支偏移量。选择分支偏移量之后，虚拟机通过把带符号的偏移量加到tableswitch操作码上，计算出目标（程序计数器）地址。目标地址必须是与tableswitch指令处于同一个方法内的操作码地址。虚拟机将会跳转到目标地址，然后从那里继续执行。

如果需要获取更多有关tableswitch指令的信息，请参考第16章。

wide

使用附加字节扩展局部变量索引。

操作码字节值：196(0xc4)

指令格式：wide，<*opcode*>，*indexbyte1*，*indexbyte2*

或者

wide，iinc，*indexbyte1*，*indexbyte2*，*constbyte1*，*constbyte2*

栈：要了解指令被wide修改后栈是如何变化的，需要查看未修改指令时的栈内容。被wide修改过的指令与未被wide修改过的指令相对比，栈的变化情况相同。

描述：wide操作码用来修改引用局部变量的指令，它把指令中无符号8位局部变量索引扩展成无符号16位索引。在“指令格式”中可以看到，wide指令有两种格式。当wide指令修改iload、fload、aload、lload、dload、istore、fstore、astore、lstore、dstore或者ret等指令时，使用第一种格式。当wide指令修改iinc指令时，使用第二种格式。

要执行wide指令，Java虚拟机首先通过计算（*indexbyte1* << 8）| *indexbyte2*，生成一个指向局部变量的无符号16位索引。不考虑wide修改的操作码的话，计算出的索引一定是指向当前栈帧内局部变量的有效索引。如果wide操作码修改lload、dload、lstore或者dstore等指令，那么比计算出的索引大1的值也一定是

有效的指向局部变量的索引。给出一个有效的指向局部变量的无符号16位（宽）索引后，虚拟机将使用该16位宽索引来执行修改后的指令。

wide操作码修改iinc指令后，iinc指令的格式有了变化。未修改的iinc指令有两个操作数：一个8位无符号局部变量索引*index*，一个8位有符号增量*const*。当iinc指令被wide修改后，iinc指令的局部变量索引和它的增量都被加以扩展。要执行该指令，Java虚拟机通过计算(*constbyte1* << 8) | *constbyte2*生成了一个带符号16位增量。

需要注意的是，从字节码验证器的观点看，被wide修改的操作码就像是wide指令的操作数。字节码不会允许wide单独对操作码做出修改。例如，goto指令直接跳转到为wide所修改的指令，这样的操作是非法的。

如果需要获取更多有关wide指令的信息，请参考第10章。

附录B　按功能排列的操作码助记符

栈和局部变量操作

将常量压入栈的指令

aconst_null	将null对象引用压入栈
iconst_m1	将int类型常量-1压入栈
iconst_0	将int类型常量0压入栈
iconst_1	将int类型常量1压入栈
iconst_2	将int类型常量2压入栈
iconst_3	将int类型常量3压入栈
iconst_4	将int类型常量4压入栈
iconst_5	将int类型常量5压入栈
lconst_0	将long类型常量0压入栈
lconst_1	将long类型常量1压入栈
fconst_0	将float类型常量0.0压入栈
fconst_1	将float类型常量1.0压入栈
fconst_2	将float类型常量2.0压入栈
dconst_0	将double类型常量0.0压入栈
dconst_1	将double类型常量1.0压入栈
bipush	将一个8位带符号整数压入栈
sipush	将16位带符号整数压入栈
ldc	把常量池中的项压入栈
ldc_w	把常量池中的项压入栈（使用宽索引）
ldc2_w	把常量池中long类型或者double类型的项压入栈（使用宽索引）

从栈中的局部变量中装载值的指令

iload	从局部变量中装载int类型值
lload	从局部变量中装载long类型值
fload	从局部变量中装载float类型值
dload	从局部变量中装载double类型值
aload	从局部变量中装载引用类型值（reference）
iload_0	从局部变量0中装载int类型值
iload_1	从局部变量1中装载int类型值
iload_2	从局部变量2中装载int类型值

iload_3	从局部变量3中装载int类型值
lload_0	从局部变量0中装载long类型值
lload_1	从局部变量1中装载long类型值
lload_2	从局部变量2中装载long类型值
lload_3	从局部变量3中装载long类型值
fload_0	从局部变量0中装载float类型值
fload_1	从局部变量1中装载float类型值
fload_2	从局部变量2中装载float类型值
fload_3	从局部变量3中装载float类型值
dload_0	从局部变量0中装载double类型值
dload_1	从局部变量1中装载double类型值
dload_2	从局部变量2中装载double类型值
dload_3	从局部变量3中装载double类型值
aload_0	从局部变量0中装载引用类型值
aload_1	从局部变量1中装载引用类型值
aload_2	从局部变量2中装载引用类型值
aload_3	从局部变量3中装载引用类型值
iaload	从数组中装载int类型值
laload	从数组中装载long类型值
faload	从数组中装载float类型值
daload	从数组中装载double类型值
aaload	从数组中装载引用类型值
baload	从数组中装载byte类型或boolean类型值
caload	从数组中装载char类型值
saload	从数组中装载short类型值

将栈中的值存入局部变量的指令

istore	将int类型值存入局部变量
lstore	将long类型值存入局部变量
fstore	将float类型值存入局部变量
dstore	将double类型值存入局部变量
astore	将引用类型或returnAddress类型值存入局部变量
istore_0	将int类型值存入局部变量0
istore_1	将int类型值存入局部变量1
istore_2	将int类型值存入局部变量2
istore_3	将int类型值存入局部变量3
lstore_0	将long类型值存入局部变量0
lstore_1	将long类型值存入局部变量1
lstore_2	将long类型值存入局部变量2

lstore_3	将long类型值存入局部变量3
fstore_0	将float类型值存入局部变量0
fstore_1	将float类型值存入局部变量1
fstore_2	将float类型值存入局部变量2
fstore_3	将float类型值存入局部变量3
dstore_0	将double类型值存入局部变量0
dstore_1	将double类型值存入局部变量1
dstore_2	将double类型值存入局部变量2
dstore_3	将double类型值存入局部变量3
astore_0	将引用类型或者returnAddress类型值存入局部变量0
astore_1	将引用类型或者returnAddress类型值存入局部变量1
astore_2	将引用类型或者returnAddress类型值存入局部变量2
astore_3	将引用类型或者returnAddress类型值存入局部变量3
iastore	将int类型值存入数组中
lastore	将long类型值存入数组中
fastore	将float类型值存入数组中
dastore	将double类型值存入数组中
aastore	将引用类型值存入数组
bastore	将byte类型或者boolean类型值存入数组
castore	将char类型值存入数组
sastore	将short类型值存入数组中

wide指令

wide	使用附加字节扩展局部变量索引

通用（无类型）栈操作

nop	不做任何操作
pop	弹出栈顶端一个字长的内容
pop2	弹出栈顶端两个字长的内容
dup	复制栈顶部一个字长内容
dup_x1	复制栈顶部一个字长的内容，然后将复制内容及原来弹出的两个字长的内容压入栈
dup_x2	复制栈顶部一个字长的内容，然后复制内容及原来弹出的三个字长的内容压入栈
dup2	复制栈顶部两个字长的内容
dup2_x1	复制栈顶部长度为两个字长的内容，然后将复制内容及原来弹出的三个字长的复制内容压入栈
dup2_x2	复制栈顶部长度为两个字长内容，然后再将复制内容及原来弹出的四个字长的内容压入栈
swap	交换栈顶部两个字长的内容

类型转换（第11章）

i2l	把int类型的数据转换为long类型
i2f	把int类型的数据转换为float类型
i2d	把int类型的数据转换为double类型
l2i	把long类型的数据转换为int类型
l2f	把long类型的数据转换为float类型
l2d	把long类型的数据转换为double类型
f2i	把float类型的数据转换为int类型
f2l	把float类型的数据转换为long类型
f2d	把float类型的数据转换为double类型
d2i	把double类型的数据转换为int类型
d2l	把double类型的数据转换为long类型
d2f	把double类型的数据转换为float类型
i2b	把int类型的数据转换为byte类型
i2c	把int类型的数据转换为char类型
i2s	把int类型的数据转换为short类型

整数运算（第12章）

iadd	执行int类型的加法
ladd	执行long类型的加法
isub	执行int类型的减法
lsub	执行long类型的减法
imul	执行int类型的乘法
lmul	执行long类型的乘法
idiv	执行int类型的除法
ldiv	执行long类型的除法
irem	计算int类型除法的余数
lrem	计算long类型除法的余数
ineg	对一个int类型值进行取反操作
lneg	对一个long类型值进行取反操作
iinc	把一个常量值加到一个int类型的局部变量上。

逻辑运算（第13章）

移位操作

ishl	执行int类型的向左移位操作
lshl	执行long类型的向左移位操作
ishr	执行int类型的向右移位操作

lshr	执行long类型的向右移位操作
iushr	执行int类型的向右逻辑移位操作
lushr	执行long类型的向右逻辑移位操作

按位布尔运算

iand	对int类型值进行“逻辑与”操作
land	对long类型值进行“逻辑与”操作
ior	对int类型值进行“逻辑或”操作
lor	对long类型值进行“逻辑或”操作
ixor	对int类型值进行“逻辑异或”操作
lxor	对long类型值进行“逻辑异或”操作

浮点运算（第14章）

fadd	执行float类型的加法
dadd	执行double类型的加法
fsub	执行float类型的减法
dsub	执行double类型的减法
fmul	执行float类型的乘法
dmul	执行double类型的乘法
fdiv	执行float类型的除法
ddiv	执行double类型的除法
frem	计算float类型除法的余数
drem	计算double类型除法的余数
fneg	将一个float类型的数值取反
dneg	将一个double类型的数值取反

对象和数组（第15章）

对象操作指令

new	创建一个新对象
checkcast	确定对象为所给定的类型
getfield	从对象中获取字段
putfield	设置对象中字段的值
getstatic	从类中获取静态字段
putstatic	设置类中静态字段的值
instanceof	判断对象是否为给定的类型

数组操作指令

newarray	分配数据成员类型为基本数据类型的新数组
anewarray	分配数据成员类型为引用类型的新数组
arraylength	获取数组长度

multianewarray	分配新的多维数组

控制流（第16章）

条件分支指令

ifeq	如果等于0，则跳转
ifne	如果不等于0，则跳转
iflt	如果小于0，则跳转
ifge	如果大于等于0，则跳转
ifgt	如果大于0，则跳转
ifle	如果小于等于0，则跳转
if_icmpeq	如果两个int值相等，则跳转
if_icmpne	如果两个int类型值不相等，则跳转
if_icmplt	如果一个int类型值小于另外一个int类型值，则跳转
if_icmpge	如果一个int类型值大于或者等于另外一个int类型值，则跳转
if_icmpgt	如果一个int类型值大于另外一个int类型值，则跳转
if_icmple	如果一个int类型值小于或者等于另外一个int类型值，则跳转
ifnull	如果等于null，则跳转
ifnonnull	如果不等于null，则跳转
if_acmpeq	如果两个对象引用相等，则跳转
if_acmpne	如果两个对象引用不相等，则跳转

比较指令

lcmp	比较long类型值
fcmpl	比较float类型值（当遇到NaN时，返回-1）
fcmpg	比较float类型值（当遇到NaN时，返回1）
dcmpl	比较double类型值（当遇到NaN时，返回-1）
dcmpg	比较double类型值（当遇到NaN时，返回1）

无条件转移指令

goto	无条件跳转
goto_w	无条件跳转（宽索引）

表跳转指令

tableswitch	通过索引访问跳转表，并跳转
lookupswitch	通过键值匹配访问跳转表，并执行跳转操作

异常（第17章）

athrow	抛出异常或者错误

finally子句（第18章）

jsr	跳转到子例程

jsr_w	跳转到子例程（宽索引）
ret	从子例程返回

方法调用与返回（第19章）

方法调用指令

invokevirtual	运行时按照对象的类来调用实例方法
invokespecial	根据编译时类型来调用实例方法
invokestatic	调用类（静态）方法
invokeinterface	调用接口方法

方法返回指令

ireturn	从方法中返回int类型的数据
lreturn	从方法中返回long类型的数据
freturn	从方法中返回float类型的数据
dreturn	从方法中返回double类型的数据
areturn	从方法中返回引用类型的数据
return	从方法中返回，返回值为void

线程同步（第20章）

monitorenter	进入并获得对象监视器
monitorexit	释放并退出对象监视器

附录C　按操作码字节值排列的操作码助记符

标准操作码

0	0x00	nop
1	0x01	aconst_null
2	0x02	iconst_m1
3	0x03	iconst_0
4	0x04	iconst_1
5	0x05	iconst_2
6	0x06	iconst_3
7	0x07	iconst_4
8	0x08	iconst_5
9	0x09	lconst_0
10	0x0a	lconst_1
11	0x0b	fconst_0
12	0x0c	fconst_1
13	0x0d	fconst_2
14	0x0e	dconst_0
15	0x0f	dconst_1
16	0x10	bipush
17	0x11	sipush
18	0x12	ldc
19	0x13	ldc_w
20	0x14	ldc2_w
21	0x15	iload
22	0x16	lload
23	0x17	fload
24	0x18	dload
25	0x19	aload
26	0x1a	iload_0
27	0x1b	iload_1
28	0x1c	iload_2
29	0x1d	iload_3
30	0x1e	lload_0

31	0x1f	lload_1
32	0x20	lload_2
33	0x21	lload_3
34	0x22	fload_0
35	0x23	fload_1
36	0x24	fload_2
37	0x25	fload_3
38	0x26	dload_0
39	0x27	dload_1
40	0x28	dload_2
41	0x29	dload_3
42	0x2a	aload_0
43	0x2b	aload_1
44	0x2c	aload_2
45	0x2d	aload_3
46	0x2e	iaload
47	0x2f	laload
48	0x30	faload
49	0x31	daload
50	0x32	aaload
51	0x33	baload
52	0x34	caload
53	0x35	saload
54	0x36	istore
55	0x37	lstore
56	0x38	fstore
57	0x39	dstore
58	0x3a	astore
59	0x3b	istore_0
60	0x3c	istore_1
61	0x3d	istore_2
62	0x3e	istore_3
63	0x3f	lstore_0
64	0x40	lstore_1
65	0x41	lstore_2
66	0x42	lstore_3
67	0x43	fstore_0
68	0x44	fstore_1

69	0x45	fstore_2
70	0x46	fstore_3
71	0x47	dstore_0
72	0x48	dstore_1
73	0x49	dstore_2
74	0x4a	dstore_3
75	0x4b	astore_0
76	0x4c	astore_1
77	0x4d	astore_2
78	0x4e	astore_3
79	0x4f	iastore
80	0x50	lastore
81	0x51	fastore
82	0x52	dastore
83	0x53	aastore
84	0x54	bastore
85	0x55	castore
86	0x56	sastore
87	0x57	pop
88	0x58	pop2
89	0x59	dup
90	0x5a	dup_x1
91	0x5b	dup_x2
92	0x5c	dup2
93	0x5d	dup2_x1
94	0x5e	dup2_x2
95	0x5f	swap
96	0x60	iadd
97	0x61	ladd
98	0x62	fadd
99	0x63	dadd
100	0x64	isub
101	0x65	lsub
102	0x66	fsub
103	0x67	dsub
104	0x68	imul
105	0x69	lmul
106	0x6a	fmul

107	0x6b	dmul
108	0x6c	idiv
109	0x6d	ldiv
110	0x6e	fdiv
111	0x6f	ddiv
112	0x70	irem
113	0x71	lrem
114	0x72	frem
115	0x73	drem
116	0x74	ineg
117	0x75	lneg
118	0x76	fneg
119	0x77	dneg
120	0x78	ishl
121	0x79	lshl
122	0x7a	ishr
123	0x7b	lshr
124	0x7c	iushr
125	0x7d	lushr
126	0x7e	iand
127	0x7f	land
128	0x80	ior
129	0x81	lor
130	0x82	ixor
131	0x83	lxor
132	0x84	iinc
133	0x85	i2l
134	0x86	i2f
135	0x87	i2d
136	0x88	l2i
137	0x89	l2f
138	0x8a	l2d
139	0x8b	f2i
140	0x8c	f2l
141	0x8d	f2d
142	0x8e	d2i
143	0x8f	d2l
144	0x90	d2f

145	0x91	i2b
146	0x92	i2c
147	0x93	i2s
148	0x94	lcmp
149	0x95	fcmpl
150	0x96	fcmpg
151	0x97	dcmpl
152	0x98	dcmpg
153	0x99	ifeq
154	0x9a	ifne
155	0x9b	iflt
156	0x9c	ifge
157	0x9d	ifgt
158	0x9e	ifle
159	0x9f	if_icmpeq
160	0xa0	if_icmpne
161	0xa1	if_icmplt
162	0xa2	if_icmpge
163	0xa3	if_icmpgt
164	0xa4	if_icmple
165	0xa5	if_acmpeq
166	0xa6	if_acmpne
167	0xa7	goto
168	0xa8	jsr
169	0xa9	ret
170	0xaa	tableswitch
171	0xab	lookupswitch
172	0xac	ireturn
173	0xad	lreturn
174	0xae	freturn
175	0xaf	dreturn
176	0xb0	areturn
177	0xb1	return
178	0xb2	getstatic
179	0xb3	putstatic
180	0xb4	getfield
181	0xb5	putfield
182	0xb6	invokevirtual

183	0xb7	invokespecial
184	0xb8	invokestatic
185	0xb9	invokeinterface
187	0xbb	new
188	0xbc	newarray
189	0xbd	anewarray
190	0xbe	arraylength
191	0xbf	athrow
192	0xc0	checkcast
193	0xc1	instanceof
194	0xc2	monitorenter
195	0xc3	monitorexit
196	0xc4	wide
197	0xc5	multianewarray
198	0xc6	ifnull
199	0xc7	ifnonnull
200	0xc8	goto_w
201	0xc9	jsr_w

快速操作码

203	0xcb	ldc_quick
204	0xcc	ldc_w_quick
205	0xcd	ldc2_w_quick
206	0xce	getfield_quick
207	0xcf	putfield_quick
208	0xd0	getfield2_quick
209	0xd1	putfield2_quick
210	0xd2	getstatic_quick
211	0xd3	putstatic_quick
212	0xd4	getstatic2_quick
213	0xd5	putstatic2_quick
214	0xd6	invokevirtual_quick
215	0xd7	invokenonvirtual_quick
216	0xd8	invokesuper_quick
217	0xd9	invokestatic_quick
218	0xda	invokeinterface_quick
219	0xdb	invokevirtualobject_quick
221	0xdd	new_quick

222	0xde	anewarray_quick
223	0xdf	multianewarray_quick
224	0xe0	checkcast_quick
225	0xe1	instanceof_quick
226	0xe2	invokevirtual_quick_w
227	0xe3	getfield_quick_w
228	0xe4	putfield_quick_w

保留操作码

202	0xca	breakpoint
254	0xfe	impdep1
255	0xff	impdep2

附录D　Java虚拟机的一个模拟："Slices of Pi"

本附录描述了随书光盘上最后一个交互式例子，即"Slices of Pi" applet，如图D-4所示。该applet演示了Java虚拟机执行计算Pi功能的字节码序列。光盘上applets/SlicesOfPi.html网页文件中包含了这个applet。

D.1　类PiCalculator

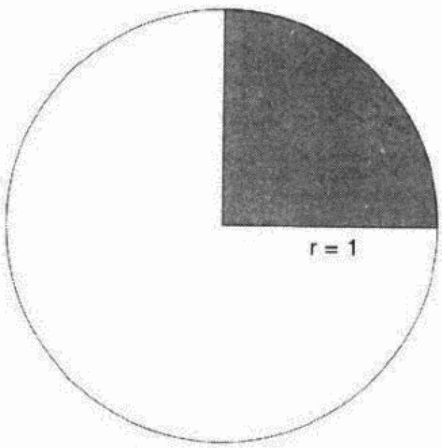
在该模拟中，javac为PiCalculator类中calculatePi()方法产生的字节码序列如下所示：

```
// On CD-ROM in file pi/ex1/PiCalculator.java
class PiCalculator {

    static void calculatePi() {

        double pi = 4.0;
        double sliceWidth = 0.5;
        double y;

        int iterations = 1;

        for (;;) {

            double x = 0.0;
            while (x < 1.0) {

                y = Math.sqrt(1 - (x * x));
                pi -= 4 * (sliceWidth * y);
                x += sliceWidth;

                y = Math.sqrt(1 - (x * x));
                pi += 4 * (sliceWidth * y);
                x += sliceWidth;
            }

            ++iterations;

            sliceWidth /= 2;
        }
    }
}
```

D.2 算法

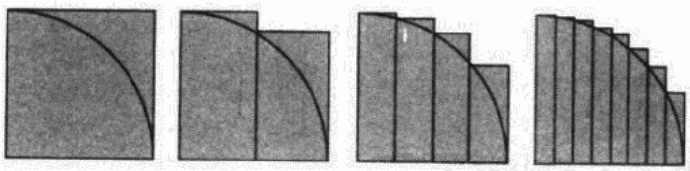
方法calculatePi()为获取π的值而一直循环运行，π即为难以捉摸的圆周率，是圆的周长与半径的比值。为了计算π的值，方法calculatePi()尝试获取半径为1的圆的周长。由于该圆的半径为1，它的周长就是π。

为了测定半径为1的圆的周长，方法calculatePi()首先尝试计算1/4圆周的长度，在得出结果后，将其乘以4即可以得到π的值。图D-1中显示了一个圆，圆中灰色部分就是方法calculatePi()所关注的区域。



图D-1 半径为1的圆的1/4区域

为了计算图D-1中灰色区域的弧长，方法calculatePi()把这一区域分割成许多连续的微小的矩形切片，如图D-2所示。



图D-2 将所关注的区域分割成微小的矩形切片

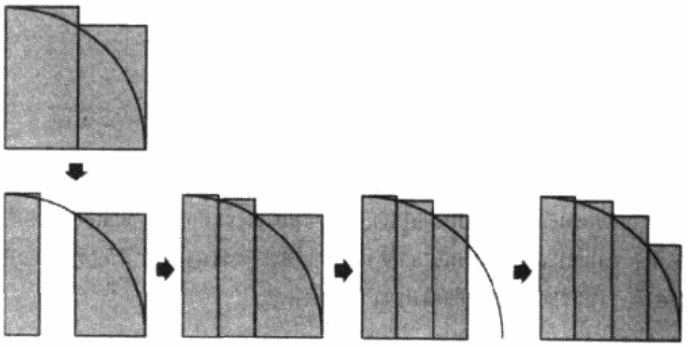
由于计算矩形区域宽度的工作是一件很简单的事情，因此方法calculatePi()能够通过简单的计算并累加，得出圆弧的长度（即1/4π的值），然后乘以4，就可以得出π的值。如图所示，这个用来计算π的办法将会产生一个与π相当接近的近似值。随着所取矩形尺寸的减小和数目的增加，近似值会越来越接近实际π的值。

方法calculatePi()构造了一个计算1/4圆周长度的循环，每一次循环使用的切片宽度都要比上一次更小。方法calculatePi()的for语句的每一次循环都是为了计算1/4圆弧长度。对于for语句的任何一次特定循环，slicewidth变量给出切片的x（水平）宽度，在每次循环过程中，slicewidth保持不变，但每一次loop循环结束处，切片的宽度slicewidth都将减半。

变量Pi的值记录了持续变化的当前π近似值。每一次calculatePi()的for循环开始时，都会把x的值设置为0，然后，每次向x加上一个大小为slicewidth的增量，直到x 等于1.0（圆的半径）为止。

在for循环中的while循环语句，每隔两个切片宽度循环一次。实际上，每一次while语句循环都会从上

次的循环中得到一个大的矩形，截取这个大矩形的一半，然后为截取后的区域计算一个新值。因为在每次for循环的结尾处，切片宽度都会减半，前次计算出的矩形宽度永远是这一次矩形宽度的两倍。图D-3显示出了calculatePi()方法把两个矩形分解成为4个矩形的步骤。



图D-3 将两个矩形分成四个

算法的结果

为了取得比calculatePi()更好的输出结果，让我们来看看这个与PiCalculator类有着密切关系的类：

```
// On CD-ROM in file pi/ex1/PiCalculatorPrinter.java
class PiCalculatorPrinter {

    static void calculateAndPrintPi() {

        double pi = 4.0;
        double sliceWidth = 0.5;
        double y;

        int iterations = 1;

        for (;;) {

            double x = 0.0;
            while (x < 1.0) {

                y = Math.sqrt(1 - (x * x));
                pi -= 4 * (sliceWidth * y);
                x += sliceWidth;

                y = Math.sqrt(1 - (x * x));
                pi += 4 * (sliceWidth * y);
                x += sliceWidth;
            }

            System.out.println(iterations + ": " + pi);
```

```
            ++iterations;

            sliceWidth /= 2;
        }
    }

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

方法PiCalculatorPrinter.calculateAndPrintPi()使用与PiCalculator.calculatePi()相同的算法来计算π，这两个类的不同之处在于：calculateAndPrintPi()使用标准输出报告它的计算过程。当PiCalculatorPrinter以Java应用程序的方式运行一段时间后，它的输出如下所示：

```
1: 3.732050807568877
2: 3.4957090681024408
3: 3.339819144357174
4: 3.248253037827741
5: 3.1976024228771323
6: 3.170546912779685
7: 3.156405792396616
8: 3.1491180829572345
9: 3.145397402719659
10: 3.143509891539023
11: 3.1425565279114656
12: 3.1420764488577837
13: 3.1418352081747196
14: 3.1417141631514287
15: 3.141653490490536
16: 3.141623101073998
17: 3.1416078875969236
18: 3.1416002742226636
19: 3.141596465189377
20: 3.1415945598432646
21: 3.141593606876806
22: 3.1415931302901634
23: 3.1415928919598097
24: 3.1415927727818502
25: 3.1415927131888464
26: 3.1415926833899084
27: 3.141592668491482
28: 3.1415926610412166
29: 3.1415926573154693
```

类java.lang.Math的静态final字段PI的值是3.14159265358979323846，该值为double类型所能够表示的最接近π的值。从上述输出可以看到：第29次循环后，算法第一次生成9位十进制有效数字的π：

3.14159265。所有π的近似值都比π的实际值要大，该算法从大到小逐步接近π的真实值。

字节码

由javac产生的calculatePi()方法的字节码如下所示：

```
// Push dual-byte value from constant pool entry
                    // In this case, a double 4.0
 0 ldc2_w #10 <Double 4.0>
                    // Pop double, store into local
 3 dstore_0         // variables 0 and 1: double pi = 4;
                    // Push dual-byte value from constant
                    // pool entry. In this case, a
                    // double 0.5
 4 ldc2_w #6 <Double 0.5>
                    // Pop double, store into local
                    // variables 2 and 3:
 7 dstore_2         // double slicewidth = 0.5;
 8 iconst_1         // Push int constant 1
                    // Pop int, store into local
 9 istore 6         // variable 6: int iterations = 1;
11 dconst_0         // Push double constant 0.0
                    // Pop double, store into local
12 dstore 7         // variables 7 and 8: double x = 0.0;
14 goto 75          // Jump to offset 75
17 dconst_1         // Push double constant 1.0
                    // Push double from local variables 7
18 dload 7          // and 8 (x)
                    // Push double from local variables 7
20 dload 7          // and 8 (x)
                    // Pop two doubles, multiply them, push
22 dmul             // double result
                    // Pop two doubles, subtract them, push
23 dsub             // double result
                    // Invoke the static method indicated
                    // by the constant entry, in this case
                    // Math.sqrt(double), which pops a
                    // double parameter and pushes a double
                    // return value
24 invokestatic #5 <Method double sqrt(double)>
                    // Pop double, store into local
                    // variables 4 and 5:
27 dstore 4         // y = Math.sqrt(1 - (x * x));
                    // Push double from local variables 0
29 dload_0          // and 1 (pi)
                    // Push dual-byte value from constant
                    // pool entry. In this case, a
                    // double 4.0
```

```
30 ldc2_w #10 <Double 4.0>
                    // Push double from local variables 2
33 dload_2          // and 3 (slicewidth)
                    // Push double from local variables 4
34 dload 4          // and 5 (y)
                    // Pop two doubles, multiply them, push
36 dmul             // double result
                    // Pop two doubles, multiply them, push
37 dmul             // double result
                    // Pop two doubles, subtract them, push
38 dsub             // result double
                    // Pop double, store into local
                    // variables 0 and 1:
39 dstore_0         // pi -= 4 * (slicewidth * y);
                    // Push double from local variables 7
40 dload 7          // and 8 (x)
                    // Push double from local variables 2
42 dload_2          // and 3 (slicewidth)
                    // Pop two doubles, add them, push
43 dadd             // double result
                    // Pop double, store into local
44 dstore 7         // variables 7 and 8: x += slicewidth;
46 dconst_1         // Push double constant 1.0
                    // Push double from local variables 7
47 dload 7          // and 8 (x)
                    // Push double from local variables 7
49 dload 7          // and 8 (x)
                    // Pop two doubles, multiply them, push
51 dmul             // double result
                    // Pop two doubles, subtract them, push
52 dsub             // double result
                    // Invoke the static method indicated
                    // by the constant entry, in this case
                    // Math.sqrt(double), which pops a
                    // double parameter and pushes a double
                    // return value
53 invokestatic #5 <Method double sqrt(double)>
                    // Pop double, store into local
                    // variables 4 and 5:
56 dstore 4         // y = Math.sqrt(1 - (x * x));
                    // Push double from local variables 0
58 dload_0          // and 1 (pi)
                    // Push dual-byte value from constant
                    // pool entry. In this case, a
                    // double 4.0
59 ldc2_w #10 <Double 4.0>
```

```
                    // Push double from local variables 2
 62 dload_2         // and 3 (slicewidth)
                    // Push double from local variables 4
 63 dload 4         // and 5 (y)
                    // Pop two doubles, multiply them, push
 65 dmul            // double result
                    // Pop two doubles, multiply them, push
 66 dmul            // double result
                    // Pop two doubles, add them, push
 67 dadd            // double result
                    // Pop double, store into local
                    // variables 0 and 1:
 68 dstore_0        // pi += 4 * (slicewidth * y);
                    // Push double from local variables 7
 69 dload 7         // and 8 (x)
                    // Push double from local variables 2
 71 dload_2         // and 3 (slicewidth)
                    // Pop two doubles, add them, push
 72 dadd            // double result
                    // Pop double, store into local
 73 dstore 7        // variables 7 and 8: x += slicewidth;
                    // Push double from local variables 7
 75 dload 7         // and 8 (x)
 77 dconst_1        // Push double constant 1.0
                    // Pop two doubles, compare them, push
 78 dcmpg           // int result
                    // Pop int, branch if less than zero to
 79 iflt 17         // offset 17: while (x < 1.0) {}
                    // Increment local variable 6 by 1:
 82 iinc 6 1        // ++iterations;
                    // Push double from local variables 2
 85 dload_2         // and 3 (slicewidth)
                    // Push dual-byte value from constant
                    // pool entry. In this case, a
                    // double 2.0
 86 ldc2_w #8 <Double 2.0>
                    // Pop two doubles, divide them, push
 89 ddiv            // double result
                    // Pop double, store into local
 90 dstore_2        // variables 2 and 3: slicewidth /= 2;
                    // Jump unconditionally to offset 11:
 91 goto 11         // for(;;) {}
```

需要注意的是，方法calculatePi（）中包括一些很好的例子，说明了在Java虚拟机中如何使用操作数栈来计算较大的表达式。例如，如果要计算“pi -= 4 * (slicewidth * y)”，Java虚拟机首先把所有4个值（pi，4.0，slicewidth和y）全部压入操作数栈，然后执行两条乘法指令和一条减法指令，最后把计算出的结果存

储到局部变量π中。此外，为了计算“(1 – (x * x)”，Java虚拟机首先把所有的3个值（1，x和x）全部压入操作数栈，然后执行一次乘法操作和一次减法操作。

处理double类型时遇到的问题

方法calculatePi（）有两个主要的问题。首先，相对于所获取π值的位数而言，它执行的时间太长，它的速度太慢是因为它运行在一个模拟的Java虚拟机上。在这个模拟的虚拟机上，一秒钟内只能执行两条指令。

其次，方法calculatePi（）的另外一个主要问题在于：它没有对取整错误做出处理。尽管方法calculatePi（）确实可以一直运行下去，但slicewidth /= 2这条指令执行到最后，会使得slicewidth值下溢至0。下溢发生以后，变量pi的值就不会再改变，但它的值并不等于Math.PI，因为在pi变量达到最终值以前，取整错误就已经发生了。

不考虑这些问题的话，方法calculatePi（）确实是Java虚拟机处理double类型值和计算较大的表达式方面的一个很好的例子。

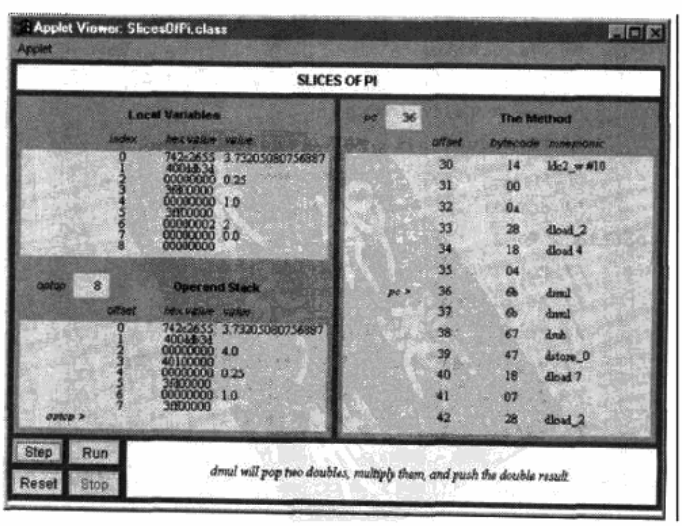
运行applet

运行该模拟的时候，可以通过观察栈帧中的局部变量来跟踪calculatePi（）方法的进展。方法claculatePi（）的局部变量被javac编译器安排到栈帧中的局部变量位置中，如表D-1所示。

表D-1 calculatePi（）的局部变量

局部变量	位置
Pi	0和1
slicewidth	2和3
y	4和5
iterations	6
x	7和8

如图D-4所示，使用“Step”、“Reset”、“Run”和“Stop”按钮即可启动“Slices of Pi”模拟。当点击



图D-4 “Slices of Pi”applet

"Step按钮时，模拟就会执行PC寄存器指向的指令。当点击Run按钮时，模拟会继续进行下一步模拟，直到点击Stop按钮才会停止。点击Reset按钮，会重新开始模拟运算。在模拟运算的每一步，applet底部的面板都将显示出对下一步指令所做工作的解释。

随书光盘

光盘上pi子目录中有该附录的源代码示例。光盘上的applets/SlicesOfPi.html网页文件里包含了该"Slices of Pi" applet。这个applet的源代码与它的class文件一起放在applets/JVMSimulators和applets/JVMSimulators/COM/artima/jvmsim目录。