

HZ BOOKS
华章科技

围绕内存管理、执行子系统、程序编译与优化、高效并发等核心内容对JVM进行了全面而深入的分析，深刻揭示JVM的工作原理
注重实战，以解决实践中的疑难问题为首要目的，包含大量经典案例和最佳实践

华章  精品

深入理解

Java 虚拟机

JVM高级特性与最佳实践

Understanding the JVM
Advanced Features and Best Practices

周志明 著



机械工业出版社
China Machine Press

Understanding the JVM

Advanced Features and Best Practices



Java 程序是如何运行的？Java虚拟机在其中扮演了怎样的角色？如何让Java程序具有更高的并发性？许多Java程序员都会有诸如此类的疑问。无奈，国内在很长一段时间里都没有一本从实际应用的角度讲解Java虚拟机的著作，本书的出版可谓填补了这个空白。它从Java程序员的角度出发，系统地将Java程序运行过程中涉及的各种知识整合到了一起，并配以日常工作中可能会碰到的疑难案例，引领读者轻松踏上探索Java虚拟机的旅途，是广大对Java虚拟机感兴趣的读者的福音！

——莫枢 (RednaxelaFX) 虚拟机和编程语言爱好者

在武侠的世界里，无论是至刚至强的《易筋经》，还是阴柔无比的《葵花宝典》，都离不开内功修炼。没有了内功心法，这些武术只是花拳绣腿的拙劣表演而已。软件业是武林江湖的一个翻版，也有着大量的模式、套路、规范等外功，但“外功修行，内功修神”，要想成为“扫地僧”一样的绝世高人，此书是必备的。

——秦小波 资深Java技术专家 / 著有畅销书《设计模式之禅》

对Java程序员来说，Java虚拟机可以说是既熟悉又神秘，极少有Java程序员能够抑制住自己探究它的冲动。可惜，分析JVM实现原理的书籍（特别是国内作者原创的）是少之又少。本书的出版可谓Java程序员的福音，作者将自己多年来在Java虚拟机领域的实践经验和研究心得汇集在了这本书中，不仅系统地讲解了Java虚拟机的工作机制和底层原理，而且更难能可贵的是与实践很好地结合了起来，具有非常强的实践指导意义，强烈推荐！

——计文柯 资深Java技术专家
著有畅销书《Spring技术内幕：深入解析Spring架构与设计实现原理》



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

华章网站 <http://www.hzbook.com>

网上购书: www.china-pub.com



上架指导: 计算机 / 程序设计 / Java

ISBN 978-7-111-34966-2



9 787111 349662

定价: 69.00元

深入理解 Java 虚拟机

JVM高级特性与最佳实践

Understanding the JVM
Advanced Features and Best Practices

周志明 著



机械工业出版社
China Machine Press

PDG

作为一位 Java 程序员，你是否也曾经想深入理解 Java 虚拟机，但是却被它的复杂和深奥拒之门外？没关系，本书极尽化繁为简之妙，能带领你在轻松中领略 Java 虚拟机的奥秘。本书是近年来国内出版的唯一一本与 Java 虚拟机相关的专著，也是唯一一本同时从核心理论和实际运用这两个角度去探讨 Java 虚拟机的著作，不仅理论分析得透彻，而且书中包含的典型案例和最佳实践也极具现实指导意义。

全书共分为五大部分。第一部分从宏观的角度介绍了整个 Java 技术体系的过去、现在和未来，以及如何独立地编译一个 OpenJDK7，这对理解后面的内容很有帮助。第二部分讲解了 JVM 的自动内存管理，包括虚拟机内存区域的划分原理以及各种内存溢出异常产生的原因；常见的垃圾收集算法以及垃圾收集器的特点和工作原理；常见的虚拟机的监控与调试工具的原理和使用方法。第三部分分析了虚拟机的执行子系统，包括 Class 的文件结构以及如何存储和访问 Class 中的数据；虚拟机的类创建机制以及类加载器的工作原理和它对虚拟机的意义；虚拟机字节码的执行引擎以及它在实行代码时涉及的内存结构。第四部分讲解了程序的编译与代码的优化，阐述了泛型、自动装箱拆箱、条件编译等语法糖的原理；讲解了虚拟机的热点探测方法、HotSpot 的即时编译器、编译触发条件，以及如何从虚拟机外部观察和分析 JIT 编译的数据和结果。第五部分探讨了 Java 实现高效并发的原理，包括 JVM 内存模型的结构和操作；原子性、可见性和有序性在 Java 内存模型中的体现；先行发生原则的规则和使用；线程在 Java 语言中的实现原理；虚拟机实现高效并发所做的一系列锁优化措施。

本书适合所有 Java 程序员、系统调优师和系统架构师阅读。

封底无防伪标均为盗版

版权所有，侵权必究

本书法律顾问 北京市展达律师事务所

图书在版编目 (CIP) 数据

深入理解 Java 虚拟机：JVM 高级特性与最佳实践 / 周志明著. —北京：机械工业出版社，2011.6

ISBN 978-7-111-34966-2

I. 深… II. 周… III. JAVA 语言—程序设计 IV. TP312

中国版本图书馆 CIP 数据核字 (2011) 第 103021 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：陈佳媛

北京京北印刷有限公司印刷

2011 年 9 月第 1 版第 3 次印刷

186mm×240mm·25.5 印张

标准书号：ISBN 978-7-111-34966-2

定价：69.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：(010) 88378991；88361066

购书热线：(010) 68326294；88379649；68995259

投稿热线：(010) 88379604

读者信箱：hzjsj@hzbook.com





前 言

Java 是目前用户最多、使用范围最广的软件开发技术，Java 的技术体系主要由支撑 Java 程序运行的虚拟机、为各开发领域提供接口支持的 Java API、Java 编程语言及许许多多的第三方 Java 框架（如 Spring 和 Struts 等）构成。在国内，有关 Java API、Java 语言及第三方框架的技术资料和书籍非常丰富，相比之下，有关 Java 虚拟机的资料却显得异常贫乏。

这种状况很大程度上是由 Java 开发技术本身的一个重要优点导致的：在虚拟机层面隐藏了底层技术的复杂性以及机器与操作系统的差异性。运行程序的物理机器情况千差万别，而 Java 虚拟机则在千差万别的物理机上面建立了统一的运行平台，实现了在任意一台虚拟机上编译的程序都能在任何一台虚拟机上正常运行。这一极大的优势使得 Java 应用的开发比传统 C/C++ 应用的开发更高效和快捷，程序员可以把主要精力集中在具体业务逻辑上，而不是物理硬件的兼容性上。一般情况下，一个程序员只要了解了必要的 Java API、Java 语法并学习适当的第三方开发框架，就已经基本能满足日常开发的需要了，虚拟机会在用户不知不觉中完成对硬件平台的兼容以及对内存等资源的管理工作。因此，了解虚拟机的运作并不是一般开发人员必须掌握的知识。

然而，凡事都具备两面性。随着 Java 技术的不断发展，它被应用于越来越多的领域之中。其中一些领域，如电力、金融、通信等，对程序的性能、稳定性和可扩展性方面都有极高的要求。一个程序很可能在 10 个人同时使用时完全正常，但是在 10000 个人同时使用时就会变慢、死锁甚至崩溃。毫无疑问，要满足 10000 个人同时使用需要更高

性能的物理硬件，但是在绝大多数情况下，提升硬件效能无法等比例地提升程序的性能和并发能力，有时甚至可能对程序的性能没有任何改善作用。这里面有 Java 虚拟机的原因：为了达到为所有硬件提供一致的虚拟平台的目的，牺牲了一些硬件相关的性能特性。更重要的是人为原因：开发人员如果不了解虚拟机的一些技术特性的运行原理，就无法写出最适合虚拟机运行和可自优化的代码。

其实，目前商用的高性能 Java 虚拟机都提供了相当多的优化特性和调节手段，用于满足应用程序在实际生产环境中对性能和稳定性的要求。如果只是为了入门学习，让程序在自己的机器上正常运行，那么这些特性可以说是可有可无的；如果用于生产环境，尤其是企业级应用开发中，就迫切需要开发人员中至少有一部分人对虚拟机的特性及调节方法具有很清晰的认识，所以在 Java 开发体系中，对架构师、系统调优师、高级程序员等角色的需求一直都非常大。学习虚拟机中各种自动运作的特性的原理也成为了 Java 程序员成长道路上必然会接触到的一课。通过本书，读者可以以一种相对轻松的方式学习虚拟机的运作原理，对 Java 程序员的成长也有较大的帮助。

本书读者对象

(1) 使用 Java 技术体系的中、高级开发人员

Java 虚拟机作为中、高级开发人员必须修炼的知识，有着较高的学习门槛，本书可作为学习虚拟机的优秀教材。

(2) 系统调优师

系统调优师是近几年才兴起的职业，本书中的大量案例、代码和调优实战将会对系统调优师的日常工作有直接的帮助。

(3) 系统架构师

保障系统高效、稳定和可伸缩是系统架构师的主要职责之一，而这与虚拟机的运作密不可分，本书可以作为他们设计应用系统底层框架的参考资料。

如何阅读本书

本书一共分为五个部分：走近 Java、自动内存管理机制、虚拟机执行子系统、程序编译与代码优化、高效并发。各个部分基本上是相互独立的，没有必然的前后依赖关

系，读者可以从任何一个感兴趣的专题开始阅读，但是每个部分中的各个章节间有先后顺序。

本书并不假设读者在 Java 领域具备很专业的技术水平，因此在保证逻辑准确的前提下，尽量用通俗的语言和案例讲述虚拟机中与开发关系最为密切的内容。当然学习虚拟机技术本身就需要读者有一定的技术基础，且本书的读者定位是中、高级程序员，因此本书假设读者自己了解一些常用的开发框架、Java API 和 Java 语法等基础知识。

语言约定

本书在语言和技术上有如下的约定：

- 本书中提到 HotSpot 虚拟机、JRockit 虚拟机、WebLogic 服务器等产品的所有者时，仍然使用 Sun 和 BEA 公司的名称。实际上 BEA 和 Sun 分别于 2008 年和 2010 年被 Oracle 公司收购，现在已经不存在这两个商标了，但是毫无疑问它们都是对 Java 领域做出过卓越贡献的、值得程序员纪念的公司。
- JDK 从 1.5 版本开始，在官方的正式文档与宣传资料中已经不再使用类似“JDK 1.5”的名称，只有在程序员内部使用的开发版本号（Developer Version，例如 `java -version` 的输出）中才继续沿用 1.5、1.6 和 1.7 的版本号，而公开版本号（Product Version）则改为 JDK 5、JDK 6 和 JDK 7 的命名方式。为了行文一致，本书所有场合统一采用开发版本号的命名方式。
- 由于版面关系，本书中的许多示例代码都没有遵循最优的代码编写风格，如使用的流没有关闭流等，请读者在阅读时注意这一点。
- 如果没有特殊说明，本书中所有的讨论都是以 Sun JDK 1.6 为技术平台的。不过如果有某个特性在各个版本间的变化较大，一般都会说明它在各个版本间的差异。

内容特色

第一部分 走近 Java

本书的第一部分为后文的讲解建立了良好的基础。尽管了解 Java 技术的来龙去脉，以及编译自己的 OpenJDK 对于读者理解 Java 虚拟机并不是必需的，但是这些准备过程

可以为走近 Java 技术和 Java 虚拟机提供很好的引导。第一部分只有第 1 章：

第 1 章 介绍了 Java 技术体系的过去、现在和未来的发展趋势，并介绍了如何独立编译一个 OpenJDK 7。

第二部分 自动内存管理机制

因为程序员把内存控制的权力交给了 Java 虚拟机，所以可以在编码的时候享受自动内存管理的诸多优势，不过也正因为这个原因，一旦出现内存泄漏和溢出方面的问题，如果不了解虚拟机是怎样使用内存的，那么排查错误将会成为一项异常艰难的工作。第二部分包括第 2 ~ 5 章：

第 2 章 讲解了虚拟机中的内存是如何划分的，哪部分区域、什么样的代码和操作可能导致内存溢出异常，并讲解了各个区域出现内存溢出异常的常见原因。

第 3 章 分析了垃圾收集的算法和 JDK 1.6 中提供的几款垃圾收集器的特点及运作原理，通过代码实例验证了 Java 虚拟机中的自动内存分配及回收的主要规则。

第 4 章 介绍了随 JDK 发布的 6 个命令行工具与 2 个可视化的故障处理工具的使用方法。

第 5 章 与读者分享了几个比较有代表性的实际案例，还准备了一个所有开发人员都能“亲身实战”的练习，读者可通过实践来获得故障处理和调优的经验。

第三部分 虚拟机执行子系统

执行子系统是虚拟机中必不可少的组成部分，了解了虚拟机如何执行程序，才能写出更优秀的代码。第三部分包括第 6 ~ 9 章：

第 6 章 讲解了 Class 文件结构中的各个组成部分，以及每个部分的定义、数据结构和使用方法，以实战的方式演示了 Class 的数据是如何存储和访问的。

第 7 章 介绍了在类加载过程的“加载”、“验证”、“准备”、“解析”和“初始化”这五个阶段中虚拟机分别执行了哪些动作，还介绍了类加载器的工作原理及其对虚拟机的意义。

第 8 章 分析了虚拟机在执行代码时如何找到正确的方法，如何执行方法内的字节码，以及执行代码时涉及的内存结构。

第 9 章 通过四个类加载及执行子系统的案例，分享了使用类加载器和处理字节码的一些值得欣赏和借鉴的思路，并通过一个实战练习来加深对前面理论知识的理解。

第四部分 程序编译与代码优化

Java 程序从源码编译成字节码和从字节码编译成本地机器码的这两个过程，合并起来其实就等同于一个传统编译器所执行的编译过程。第四部分包括第 10 和 11 章：

第 10 章 分析了 Java 语言中的泛型、自动装箱拆箱、条件编译等多种语法糖的前因后果，并通过实战案例演示了如何使用插入式注解处理器来实现一个检查程序命名规范的编译器插件。

第 11 章 讲解了虚拟机的热点探测方法、HotSpot 的即时编译器、编译触发条件，以及如何从虚拟机外部观察和分析 JIT 编译的数据和结果。此外，还讲解了几种常见的编译期优化技术。

第五部分 高效并发

Java 语言和虚拟机提供了原生的、完善的多线程支持，使得它天生就适合开发多线程并发的应用程序。不过我们不能期望系统来完成所有与并发相关的处理，了解并发的内幕也是一个高级程序员不可缺少的课程。第五部分包括第 12 和 13 章：

第 12 章 讲解了虚拟机的 Java 内存模型的结构和操作，以及原子性、可见性和有序性在 Java 内存模型中的体现，介绍了先行发生原则及使用，还讲解了线程在 Java 语言中是如何实现的。

第 13 章 介绍了线程安全所涉及的概念和分类、同步实现的方式以及虚拟机的底层运作原理，并且还介绍了虚拟机实现高效并发所采取的一系列锁优化措施。

参考资料

本书名为“深入理解 Java 虚拟机”，但要想真的深入理解虚拟机，仅凭一本书肯定是远远不够的，读者可以通过下面的信息找到更多关于 Java 虚拟机方面的资料。我在写作此书的时候，也从下面这些参考资料中获得了很大的帮助。

(1) 书籍

□《The Java Virtual Machine Specification, Second Edition》^①

《Java 虚拟机规范（第 2 版）》，1999 年 4 月出版。国内并没有引进这本书，

^① 官方地址：http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html。

自然也就没有中文译本，但全书的电子版是免费发布的，在 IT 书籍中它已经非常“高寿”了^①（这本书的第 3 版已处于基本完成的草稿状态，在 JDK 1.7 正式版发布后这本书应该就会推出第 3 版）。要学习虚拟机，虚拟机规范无论如何都是必须读的。这本书的概念和细节描述与 Sun 的早期虚拟机（Sun Classic VM）高度吻合，不过，随着技术的发展，高性能虚拟机真正的细节实现方式与虚拟机规范所描述的差距已经越来越大。但是，如果只能选择一本参考书来了解虚拟机的话，那仍然是这本书。

□ 《The Java Language Specification, Third Edition》^②

《Java 语言规范（第 3 版）》，2005 年 7 月由机械工业出版社出版，不过出版的是影印版，没有中文译本。虽然 Java 虚拟机并不是 Java 语言专有的，但是了解 Java 语言的各种细节规定对虚拟机的行为也是很有帮助的，它与《Java 虚拟机规范（第 2 版）》都是 Sun 官方出品的书籍，而且这本书还是由 Java 之父 James Gosling 亲自撰写的。

□ 《Oracle JRockit The Definitive Guide》

《Oracle JRockit 权威指南》，2010 年 7 月出版，国内也没有（可能是尚未）引进这本书，它是由 JRockit 的两位资深开发人员（其中一位是 JRockit Mission Control 团队的 TeamLeader）撰写的高级 JRockit 虚拟机使用指南。虽然 JRockit 的用户量可能不如 HotSpot 多，但也是最流行的三大商业虚拟机之一，并且不同虚拟机中的很多实现思路都是可以对比参照的。这本书是了解现代高性能虚拟机的很好的途径。

□ 《Inside the Java 2 Virtual Machine, Second Edition》

《深入 Java 虚拟机（第 2 版）》，2000 年 1 月出版，2003 年机械工业出版社出版了中文译本。在相当长的时间里，这本书是唯一一本关于 Java 虚拟机的中文图书。

① 在这十多年间虚拟机规范的更新可以通过 JSR-924 规范来跟踪，JSR-924 规范的地址为：<http://jcp.org/aboutJava/communityprocess/maintenance/jsr924/index3.html>。

② 官方地址：<http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf>。

(2) 网站资源

□ 高级语言虚拟机圈子：<http://hllvm.group.iteye.com/>

里面有一些国内关于虚拟机的讨论，并不只限于 JVM，而是涉及对所有的高级语言虚拟机（High-Level Language Virtual Machine）的讨论，但该网站建立在 ITEye^①上，自然还是以讨论 Java 虚拟机为主。圈主撒迦（莫枢）的博客（<http://rednaxelafx.iteye.com/>）是另外一个非常有价值的虚拟机及编译原理等资料的分享园地。

□ HotSpot Internals：<http://wikis.sun.com/display/HotSpotInternals/Home>

一个关于 OpenJDK 的 Wiki 网站，许多文章都由 JDK 的开发团队编写，更新很慢，但是仍然有很大的参考价值。

□ The HotSpot Group：<http://openjdk.java.net/groups/hotspot/>

HotSpot 组群，包含虚拟机开发、编译器、垃圾收集和运行时四个邮件组，其中有关于 HotSpot 虚拟机的最新讨论。

联系作者

在本书完稿时，我并没有像想象中那样兴奋或放松，写作时的那种“战战兢兢、如履薄冰”的感觉依然萦绕在心头。在每一章、每一节落笔之时，我都在考虑如何才能把各个知识点更有条理地讲述出来，都在担心会不会由于自己理解有偏差而误导了大家。囿于我的写作水平和写作时间，书中难免存在不妥之处，所以特地开通了一个读者邮箱（understandingjvm@gmail.com）与大家交流，大家如有任何意见或建议都欢迎与我联系。此外，大家也可以通过我的微博（<http://t.sina.com.cn/icyfenix>）与我取得联系。

勘误

写书和写代码一样，刚开始都是不完美的，需要不断地修正和重构，本书也不例外。如果大家在阅读本书的过程中发现了本书中存在的任何问题，都欢迎反馈给我们。我们会把本书的勘误集中公布在 icyfenx.iteye.com/blog/1119214。

^① 该网站原名为 JavaEye，近期更名为 ITEye。



致 谢

首先要感谢我的家人，全靠家人在本书写作期间对我的悉心照顾，才让我能够全身心地投入到写作之中，而无后顾之忧。

同时要感谢我的工作单位远光软件，公司为我提供了宝贵的工作、学习和实践的环境，本书中的许多知识点都来自于工作中的实践；也感谢与我一起工作的同事们，非常荣幸能与你们一起在这个富有激情的团队中共同奋斗。

还要感谢淘宝网的莫枢（rednaxelafx.iteye.com）在百忙之中抽空审阅了本书，提出了许多宝贵的建议和意见。

最后，感谢机械工业出版社华章公司的编辑们，本书能够顺利出版，离不开他们的敬业精神和一丝不苟的工作态度。

周志明

2011年5月



目 录

前 言
致 谢

第一部分 走近 Java

第 1 章 走近 Java / 2

- 1.1 概述 / 2
- 1.2 Java 技术体系 / 3
- 1.3 Java 发展史 / 5
- 1.4 展望 Java 技术的未来 / 9
 - 1.4.1 模块化 / 9
 - 1.4.2 混合语言 / 9
 - 1.4.3 多核并行 / 11
 - 1.4.4 进一步丰富语法 / 12
 - 1.4.5 64 位虚拟机 / 13
- 1.5 实战：自己编译 JDK / 13
 - 1.5.1 获取 JDK 源码 / 13
 - 1.5.2 系统需求 / 14



资源分享
PDF

- 1.5.3 构建编译环境 / 15
- 1.5.4 准备依赖项 / 17
- 1.5.5 进行编译 / 18
- 1.6 本章小结 / 21

第二部分 自动内存管理机制

第 2 章 Java 内存区域与内存溢出异常 / 24

- 2.1 概述 / 24
- 2.2 运行时数据区域 / 25
 - 2.2.1 程序计数器 / 25
 - 2.2.2 Java 虚拟机栈 / 26
 - 2.2.3 本地方法栈 / 27
 - 2.2.4 Java 堆 / 27
 - 2.2.5 方法区 / 28
 - 2.2.6 运行时常量池 / 29
 - 2.2.7 直接内存 / 29
- 2.3 对象访问 / 30
- 2.4 实战：OutOfMemoryError 异常 / 32
 - 2.4.1 Java 堆溢出 / 32
 - 2.4.2 虚拟机栈和本地方法栈溢出 / 35
 - 2.4.3 运行时常量池溢出 / 38
 - 2.4.4 方法区溢出 / 39
 - 2.4.5 本机直接内存溢出 / 41
- 2.5 本章小结 / 42

第 3 章 垃圾收集器与内存分配策略 / 43

- 3.1 概述 / 43
- 3.2 对象已死？ / 44
 - 3.2.1 引用计数算法 / 44



- 3.2.2 根搜索算法 / 46
 - 3.2.3 再谈引用 / 47
 - 3.2.4 生存还是死亡? / 48
 - 3.2.5 回收方法区 / 50
 - 3.3 垃圾收集算法 / 51
 - 3.3.1 标记 - 清除算法 / 52
 - 3.3.2 复制算法 / 52
 - 3.3.3 标记 - 整理算法 / 54
 - 3.3.4 分代收集算法 / 54
 - 3.4 垃圾收集器 / 55
 - 3.4.1 Serial 收集器 / 56
 - 3.4.2 ParNew 收集器 / 57
 - 3.4.3 Parallel Scavenge 收集器 / 59
 - 3.4.4 Serial Old 收集器 / 60
 - 3.4.5 Parallel Old 收集器 / 61
 - 3.4.6 CMS 收集器 / 61
 - 3.4.7 G1 收集器 / 64
 - 3.4.8 垃圾收集器参数总结 / 64
 - 3.5 内存分配与回收策略 / 65
 - 3.5.1 对象优先在 Eden 分配 / 66
 - 3.5.2 大对象直接进入老年代 / 68
 - 3.5.3 长期存活的对象将进入老年代 / 69
 - 3.5.4 动态对象年龄判定 / 71
 - 3.5.5 空间分配担保 / 73
 - 3.6 本章小结 / 75
- 第 4 章 虚拟机性能监控与故障处理工具 / 76**
- 4.1 概述 / 76



XIV

- 4.2 JDK 的命令行工具 / 76
 - 4.2.1 jps : 虚拟机进程状况工具 / 79
 - 4.2.2 jstat : 虚拟机统计信息监视工具 / 80
 - 4.2.3 jinfo : Java 配置信息工具 / 82
 - 4.2.4 jmap : Java 内存映像工具 / 82
 - 4.2.5 jhat : 虚拟机堆转储快照分析工具 / 84
 - 4.2.6 jstack : Java 堆栈跟踪工具 / 85
- 4.3 JDK 的可视化工具 / 87
 - 4.3.1 JConsole : Java 监视与管理控制台 / 88
 - 4.3.2 VisualVM : 多合一故障处理工具 / 96
- 4.4 本章小结 / 105

第 5 章 调优案例分析与实战 / 106

- 5.1 概述 / 106
- 5.2 案例分析 / 106
 - 5.2.1 高性能硬件上的程序部署策略 / 106
 - 5.2.2 集群间同步导致的内存溢出 / 109
 - 5.2.3 堆外内存导致的溢出错误 / 110
 - 5.2.4 外部命令导致系统缓慢 / 112
 - 5.2.5 服务器 JVM 进程崩溃 / 113
- 5.3 实战 : Eclipse 运行速度调优 / 114
 - 5.3.1 调优前的程序运行状态 / 114
 - 5.3.2 升级 JDK 1.6 的性能变化及兼容问题 / 117
 - 5.3.3 编译时间和类加载时间的优化 / 122
 - 5.3.4 调整内存设置控制垃圾收集频率 / 126
 - 5.3.5 选择收集器降低延迟 / 130
- 5.4 本章小结 / 133



第三部分 虚拟机执行子系统

第 6 章 类文件结构 / 136

- 6.1 概述 / 136
- 6.2 无关性的基石 / 136
- 6.3 Class 类文件结构 / 138
 - 6.3.1 魔数与 Class 文件的版本 / 139
 - 6.3.2 常量池 / 141
 - 6.3.3 访问标志 / 147
 - 6.3.4 类索引、父类索引与接口索引集合 / 148
 - 6.3.5 字段表集合 / 149
 - 6.3.6 方法表集合 / 153
 - 6.3.7 属性表集合 / 155
- 6.4 Class 文件结构的发展 / 168
- 6.5 本章小结 / 170

第 7 章 虚拟机类加载机制 / 171

- 7.1 概述 / 171
- 7.2 类加载的时机 / 172
- 7.3 类加载的过程 / 176
 - 7.3.1 加载 / 176
 - 7.3.2 验证 / 178
 - 7.3.3 准备 / 181
 - 7.3.4 解析 / 182
 - 7.3.5 初始化 / 186
- 7.4 类加载器 / 189
 - 7.4.1 类与类加载器 / 189
 - 7.4.2 双亲委派模型 / 191
 - 7.4.3 破坏双亲委派模型 / 194



7.5 本章小结 / 197

第 8 章 虚拟机字节码执行引擎 / 198

8.1 概述 / 198

8.2 运行时栈帧结构 / 199

8.2.1 局部变量表 / 199

8.2.2 操作数栈 / 204

8.2.3 动态连接 / 206

8.2.4 方法返回地址 / 206

8.2.5 附加信息 / 207

8.3 方法调用 / 207

8.3.1 解析 / 207

8.3.2 分派 / 209

8.4 基于栈的字节码解释执行引擎 / 221

8.4.1 解释执行 / 221

8.4.2 基于栈的指令集与基于寄存器的指令集 / 223

8.4.3 基于栈的解释器执行过程 / 224

8.5 本章小结 / 230

第 9 章 类加载及执行子系统的案例与实战 / 231

9.1 概述 / 231

9.2 案例分析 / 231

9.2.1 Tomcat：正統的类加载器架构 / 232

9.2.2 OSGi：灵活的类加载器架构 / 235

9.2.3 字节码生成技术与动态代理的实现 / 238

9.2.4 Retrotranslator：跨越 JDK 版本 / 242

9.3 实战：自己动手实现远程执行功能 / 246

9.3.1 目标 / 246

9.3.2 思路 / 247



- 9.3.3 实现 / 248
- 9.3.4 验证 / 255
- 9.4 本章小结 / 256

第四部分 程序编译与代码优化

第 10 章 早期（编译期）优化 / 258

- 10.1 概述 / 258
- 10.2 Javac 编译器 / 259
 - 10.2.1 Javac 的源码与调试 / 259
 - 10.2.2 解析与填充符号表 / 262
 - 10.2.3 注解处理器 / 264
 - 10.2.4 语义分析与字节码生成 / 264
- 10.3 Java 语法糖的味道 / 268
 - 10.3.1 泛型与类型擦除 / 268
 - 10.3.2 自动装箱、拆箱与遍历循环 / 273
 - 10.3.3 条件编译 / 275
- 10.4 实战：插入式注解处理器 / 276
 - 10.4.1 实战目标 / 276
 - 10.4.2 代码实现 / 277
 - 10.4.3 运行与测试 / 284
 - 10.4.4 其他应用案例 / 286
- 10.5 本章小结 / 286

第 11 章 晚期（运行期）优化 / 287

- 11.1 概述 / 287
- 11.2 HotSpot 虚拟机内的即时编译器 / 288
 - 11.2.1 解释器与编译器 / 288
 - 11.2.2 编译对象与触发条件 / 291



XVIII

- 11.2.3 编译过程 / 294
- 11.2.4 查看与分析即时编译结果 / 297
- 11.3 编译优化技术 / 301
 - 11.3.1 优化技术概览 / 301
 - 11.3.2 公共子表达式消除 / 305
 - 11.3.3 数组边界检查消除 / 307
 - 11.3.4 方法内联 / 307
 - 11.3.5 逃逸分析 / 309
- 11.4 Java 与 C/C++ 的编译器对比 / 311
- 11.5 本章小结 / 313

第五部分 高效并发

第 12 章 Java 内存模型与线程 / 316

- 12.1 概述 / 316
- 12.2 硬件的效率与一致性 / 317
- 12.3 Java 内存模型 / 318
 - 12.3.1 主内存与工作内存 / 319
 - 12.3.2 内存间交互操作 / 320
 - 12.3.3 对于 volatile 型变量的特殊规则 / 322
 - 12.3.4 对于 long 和 double 型变量的特殊规则 / 327
 - 12.3.5 原子性、可见性与有序性 / 328
 - 12.3.6 先行发生原则 / 330
- 12.4 Java 与线程 / 333
 - 12.4.1 线程的实现 / 333
 - 12.4.2 Java 线程调度 / 337
 - 12.4.3 状态转换 / 339
- 12.5 本章小结 / 341



第 13 章 线程安全与锁优化 / 342

- 13.1 概述 / 342
- 13.2 线程安全 / 343
 - 13.2.1 Java 语言中的线程安全 / 343
 - 13.2.2 线程安全的实现方法 / 348
- 13.3 锁优化 / 356
 - 13.3.1 自旋锁与自适应自旋 / 356
 - 13.3.2 锁消除 / 357
 - 13.3.3 锁粗化 / 358
 - 13.3.4 轻量级锁 / 358
 - 13.3.5 偏向锁 / 361
- 13.4 本章小结 / 362

附录 A Java 虚拟机家族 / 363

附录 B 虚拟机字节码指令表 / 366

附录 C HotSpot 虚拟机主要参数表 / 372

附录 D 对象查询语言 (OQL) 简介 / 376

附录 E JDK 历史版本轨迹 / 383





第一部分 走近 Java

第 1 章 走近 Java



资源知识
PDG

第 1 章 走近 Java

本章主要内容

- 概述
- Java 技术体系
- Java 发展史
- 展望 Java 技术的未来
- 实战：自己编译 JDK

世界上并没有完美的程序，但我们并不因此而沮丧，因为写程序本来就是一个不断追求完美的过程。

1.1 概述

Java 不仅仅是一门编程语言，它还是一个由一系列计算机软件和规范形成的技术体系，这个技术体系提供了完整的用于软件开发和跨平台部署的支持环境，并广泛应用于嵌入式系统、移动终端、企业服务器和大型机等各种场合，如图 1-1 所示。时至今日，Java 技术体系已经吸引了 600 多万软件开发者，这是全球最大的软件开发团队。使用 Java 的设备多达几十亿台，其中包括 8 亿多台个人计算机、21 亿部移动电话及其他手持设备、35 亿个智能卡，以及大量机顶盒、导航系统和其他设备^①。

Java 能获得如此广泛的认可，除了因为它拥有一门结构严谨、面向对象的编程语言之外，还有许多不可忽视的优点：它摆脱了硬件平台的束缚，实现了“一次编写，到处运行”的理想；它提供了一种相对安全的内存管理和访问机制，避免了绝大部分的内存泄漏和指针越界问题；它实现了热点代码检测和运行时编译及优化，这使得 Java 应用能

^① 这些数据是 Java 的广告词，它们来源于：http://www.java.com/zh_CN/about/。

随着运行时间的增加而获得更高的性能；它有一套完善的应用程序接口，还有无数的来自商业机构和开源社区的第三方类库来帮助实现各种各样的功能……Java 所带来的这些好处让程序的开发效率得到了很大的提升。作为一名 Java 程序员，在编写程序时除了尽情发挥 Java 的各种优势外，还应该去了解和思考一下 Java 技术体系中这些技术是如何实现的。认清这些技术的运作本质，是自己思考“程序这样写好不好”的基础和前提。当我们在使用一门技术时，如果不再依赖书本和他人就能得到这个问题的答案，那才算升华到了“不惑”的境界。

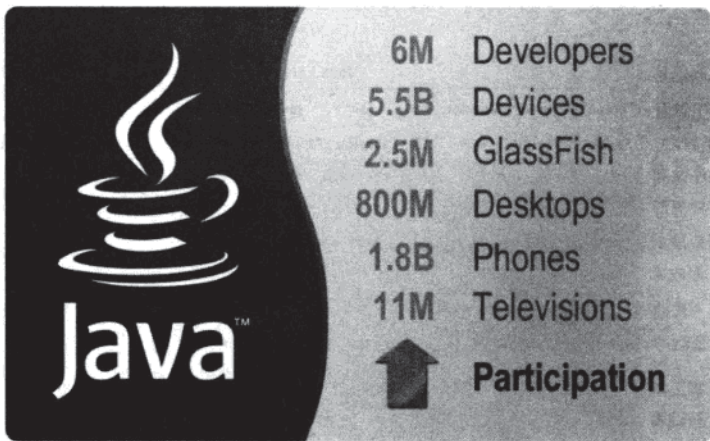


图 1-1 Java 技术的广泛应用

本书将会与读者一起分析 Java 技术中最重要的那些特性的实现原理。在本章中，我们将重点介绍 Java 技术体系所包括的内容，以及 Java 的历史、现在和未来的发展趋势。

1.2 Java 技术体系

从广义上讲，Clojure、JRuby、Groovy 等运行于 Java 虚拟机上的语言及其相关的程序都属于 Java 技术体系的一员。如果仅从传统意义上来看，Sun 官方所定义的 Java 技术体系包括了以下几个组成部分：

- Java 程序设计语言
- 各种硬件平台上的 Java 虚拟机
- Class 文件格式

4 ❖ 第一部分 走近 Java

□ Java API 类库

□ 来自商业机构和开源社区的第三方 Java 类库

我们可以把 Java 程序设计语言、Java 虚拟机、Java API 类库这三部分统称为 JDK (Java Development Kit), JDK 是用于支持 Java 程序开发的最小环境, 在后面的内容中, 为了讲解方便, 有一些地方会以 JDK 来代替整个 Java 技术体系。另外, 可以把 Java API 类库中的 Java SE API 子集^①和 Java 虚拟机这两部分统称为 JRE (Java Runtime Environment), JRE 是支持 Java 程序运行的标准环境。图 1-2 展示了 Java 技术体系所包括的内容, 以及 JDK 和 JRE 所涵盖的范围。

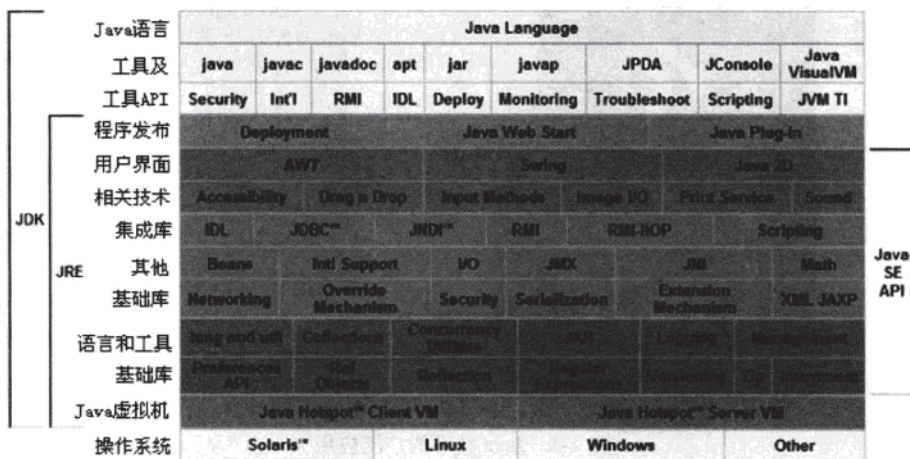


图 1-2 Java 技术体系所包括的内容^②

以上是各个组成部分的功能来进行划分的, 如果按照技术所服务的领域来划分, 或者说按照 Java 技术关注的重点业务领域来划分, Java 技术体系可以分为四个平台, 分别为:

□ Java Card: 支持一些 Java 小程序 (Applets) 运行在小内存设备 (如智能卡) 上的平台。

□ Java ME (Micro Edition): 支持 Java 程序运行在移动终端 (手机、PDA) 上的平台, 对 Java API 有所精简, 并加入了针对移动终端的支持, 这个版本以前称为 J2ME。

① JDK 1.6 的 Java SE API 范围: <http://download.oracle.com/javase/6/docs/api/>。

② 图片来源: <http://download.oracle.com/javase/6/docs/>。

- Java SE (Standard Edition): 支持面向桌面级应用 (如 Windows 下的应用程序) 的 Java 平台, 提供了完整的 Java 核心 API, 这个版本以前称为 J2SE。
- Java EE (Enterprise Edition): 支持使用多层架构的企业应用 (如 ERP、CRM 应用) 的 Java 平台, 除了提供 Java SE API 外, 还对其做了大量的扩充^①并提供了相关的部署支持, 这个版本以前称为 J2EE。

1.3 Java 发展史

从第一个 Java 版本诞生到现在已经有 16 年的时间了。沧海桑田一瞬间, 转眼 16 年过去了, 在图 1-3 所展示的时间线中, 我们看到 JDK 已经发展到了 1.7 版。在这 16 年里还诞生了无数和 Java 相关的产品、技术和标准。现在让我们进入时间隧道, 从孕育 Java 语言的时代开始, 再来回顾一下 Java 的发展轨迹和历史变迁。

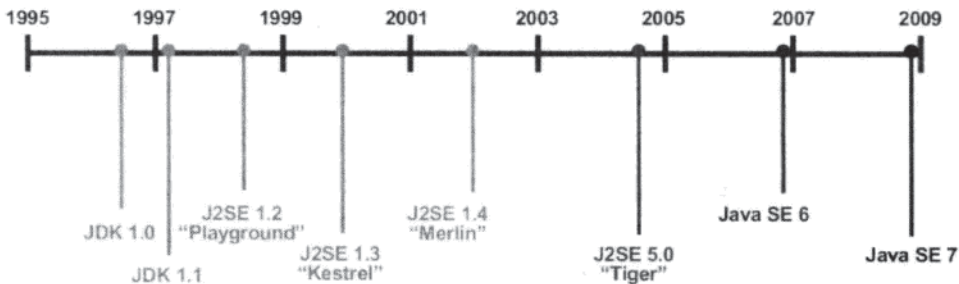


图 1-3 Java 技术发展的时间线

1991 年 4 月, 由 James Gosling 博士领导的绿色计划 (Green Project) 开始启动, 此计划的目的是开发一种能够在各种消费性电子产品 (如机顶盒、冰箱、收音机等) 上运行的程序架构。这个计划的产品就是 Java 语言的前身: Oak (橡树)。Oak 当时在消费品市场上并不算成功, 但随着 1995 年互联网潮流的兴起, Oak 迅速找到了最适合自己发展的市场定位并蜕变成为 Java 语言。

1995 年 5 月 23 日, Oak 语言改名为 Java, 并且在 SunWorld 大会上正式发布了 Java 1.0 版本。Java 语言第一次提出了 “Write Once, Run Anywhere” 的口号。

^① 这些扩展一般以 javax.* 作为包名, 而以 java.* 为包名的包都是 Java SE API 的核心包, 但由于历史原因, 一部分曾经是扩展包的 API 后来进入了核心包, 因此核心包中也包含了不少 javax.* 的包名。

6 ❖ 第一部分 走近 Java

1996年1月23日，JDK 1.0发布，Java语言有了第一个正式版本的运行环境。JDK 1.0提供了一个纯解释执行的Java虚拟机实现（Sun Classic VM）。JDK 1.0版本的技术代表包括：Java虚拟机、Applet和AWT等。

1996年4月，10个最主要的操作系统供应商声明将在其产品中嵌入Java技术。同年9月，已有大约8.3万个网页应用Java技术来制作。在1996年5月底，Sun于美国旧金山举行了首届JavaOne大会，从此JavaOne成为全世界数百万Java语言开发者每年一度的技术盛会。

1997年2月19日，Sun发布了JDK 1.1，Java技术的一些最基础的支撑点（如JDBC等）都是在JDK 1.1版本中发布的，JDK 1.1的技术代表有：JAR文件格式、JDBC、JavaBeans、RMI。Java语法也有了一定的发展，如内部类（Inner Class）和反射（Reflection）都是在这个时候出现的。

直到1999年4月8日，JDK 1.1一共发布了1.1.0至1.1.8九个版本。从1.1.4之后，每个JDK版本都有一个自己的名字（工程代号），分别为：JDK 1.1.4 - Sparkler（宝石）、JDK 1.1.5 - Pumpkin（南瓜）、JDK 1.1.6 - Abigail（阿比盖尔，女子名）、JDK 1.1.7 - Brutus（布鲁图，古罗马政治家和将军）和JDK 1.1.8 - Chelsea（切尔西，城市名）。

1998年12月4日，JDK迎来了一个里程碑式的版本JDK 1.2，工程代号为Playground（竞技场），Sun在这个版本中把Java技术体系拆分为3个方向，分别是面向桌面应用开发的J2SE（Java 2 Platform, Standard Edition）、面向企业级开发的J2EE（Java 2 Platform, Enterprise Edition）和面向手机等移动终端开发的J2ME（Java 2 Platform, Micro Edition）。在这个版本中出现的代表性技术非常多，如EJB、Java Plug-in、Java IDL、Swing等，并且在这个版本中Java虚拟机第一次内置了JIT（Just In Time）编译器（JDK 1.2中曾并存过3个虚拟机，分别是Classic VM、Hot Spot VM和Exact VM，其中Exact VM只在Solaris平台出现过；后面2个虚拟机都是内置JIT编译器的，而之前的版本所带的Classic VM只能以外挂形式使用JIT编译器）。在语言和API级别上，Java添加了strictfp关键字与现在Java编码之中极为常用的一系列Collections集合类。在1999年3月和7月，分别有JDK 1.2.1和JDK 1.2.2两个小版本发布。

1999年4月27日，HotSpot虚拟机发布，HotSpot最初由一家名为“Longview Technologies”的小公司开发，因为HotSpot的优异表现，这家公司在1997年被Sun公司收购了。HotSpot虚拟机发布时是作为JDK 1.2的附加程序提供的，后来它成为了JDK 1.3及之后所有版本的Sun JDK的默认虚拟机。

2000 年 5 月 8 日，工程代号为 Kestrel（美洲红隼）的 JDK 1.3 发布，JDK 1.3 相对于 JDK 1.2 的改进主要表现在一些类库（如数学运算和新的 Timer API 等）上，JNDI 服务从 JDK 1.3 开始被作为一项平台级服务提供（以前 JNDI 仅仅是一项扩展），使用 CORBA IIOP 来实现 RMI 的通讯协议，等等。这个版本还对 Java 2D 做了很多改进，提供了大量新的 Java 2D API，并且新添加了 JavaSound 类库。JDK 1.3 有 1 个修正版本 JDK 1.3.1，工程代号为 Ladybird（瓢虫）于 2001 年 5 月 17 日发布。

自从 JDK 1.3 开始，Sun 维持了一个习惯：大约每隔两年发布一个 JDK 的主版本，以动物名称命名，期间发布的各个修正版本则以昆虫名称作为工程名称。

2002 年 2 月 13 日，JDK 1.4 发布，工程代号为 Merlin（灰背隼）。JDK 1.4 是 Java 真正走向成熟的一个版本，Compaq、Fujitsu、SAS、Symbian、IBM 等著名公司都有参与甚至实现自己独立的 JDK 1.4。哪怕是在近 10 年后的今天，仍然有许多主流应用（Spring、Hibernate、Struts 等）能直接运行在 JDK 1.4 之上，或者继续发布能运行在 1.4 上的版本。JDK 1.4 同样发布了很多新的技术特性，如正则表达式、异常链、NIO、日志类、XML 解析器和 XSLT 转换器，等等。JDK 1.4 有两个后续修正版：2002 年 9 月 16 日发布的工程代号为 Grasshopper（蚱蜢）的 JDK 1.4.1 与 2003 年 6 月 26 日发布的工程代号为 Mantis（螳螂）的 JDK 1.4.2。

2002 年前后还发生了一件与 Java 没有直接关系，但事实上对 Java 的发展进程影响很大的事件，即微软的 .NET Framework 发布。这个无论是技术实现还是目标用户上都与 Java 有很多相近之处的技术平台给 Java 带来了许多讨论、比较和竞争，.NET 平台和 Java 平台之间声势浩大的孰优孰劣的论战到今天为止仍然在继续。

2004 年 9 月 30 日，JDK 1.5 发布^①，工程代号为 Tiger（老虎）。从 JDK 1.2 以来，Java 在语法层面上的变化一直很小，而 JDK 1.5 在 Java 语法易用性上做出了非常大的改进。自动装箱、泛型、动态注解、枚举、可变长参数、遍历循环（foreach 循环）等语法特性都是在 JDK 1.5 中加入的。在虚拟机和 API 层面上，这个版本改进了 Java 的内存模型（Java Memory Model, JMM）、提供了 `java.util.concurrent` 并发包等。另外，JDK 1.5 是官方声明可以支持 Windows 9x 平台的最后一个 JDK 版本。

① JDK 从 1.5 版本开始，官方在正式文档与宣传上已经不再使用类似 JDK 1.5 的命名，只有程序员内部使用的开发版本号（Developer Version，例如 `java-version` 的输出）中才继续沿用 1.5、1.6、1.7 的版本号，而公开版本号（Product Version）则改为 JDK 5、JDK 6、JDK 7 的命名方式，本书为了行文一致，所有场合统一采用开发版本号的命名方式。

8 ❖ 第一部分 走近 Java

2006年12月11日，JDK 1.6发布，工程代号为Mustang（野马）。这是目前为止最新的正式版JDK（截至本书完稿时，JDK 1.7仍然处于Early Access版本）。在这个版本中，Sun终结了从JDK 1.2开始已经有8年历史的J2EE、J2SE、J2ME的命名方式，启用了Java SE 6、Java EE 6、Java ME 6的命名来代替。JDK 1.6的改进包括：提供动态语言支持（通过内置Mozilla JavaScript Rhino引擎实现）、提供编译API和微型HTTP服务器API，等等。同时，这个版本对Java虚拟机的内部做了大量改进，包括锁与同步、垃圾收集、类加载等方面的算法都有相当多的改动。

在2006年11月13日的JavaOne大会上，Sun宣布最终会把Java开源，并在随后的一年多时间内，陆续地在GPL v2（GNU General Public License v2）协议下公开了JDK各个部分的源码，并建立了OpenJDK组织对这些源码进行独立管理。除了极少量的产权代码（Encumbered Code，这部分代码大多是Sun本身也无权限进行开源处理的）外，OpenJDK几乎包括了Sun JDK的全部代码。OpenJDK的质量主管曾经表示，在JDK 1.7中，Sun JDK和OpenJDK除了代码文件头的版权注释之外，代码基本上完全一样，所以OpenJDK 7与Sun JDK 1.7本质上就是同一套代码库出来的产品。

JDK 1.6发布以后，由于代码复杂性的增加、JDK开源、开发JavaFX、经济危机及Sun收购案等原因，Sun在JDK发展以外的事情上耗费了很多资源，JDK的更新没有再维持两年发布一个主版本的发展速度。JDK 1.6到今天为止一共发布了25个Update，最新的版本为Java SE 6 Update 25，于2011年4月21日发布。

2009年2月19日，工程代号为Dolphin（海豚）的JDK 1.7完成了其第一个里程碑版本。根据JDK 1.7的功能规划，一共设置了10个里程碑。最后一个里程碑版本于2010年9月9日结束。从发布的Early Access版看来目前JDK 1.7的主体功能，已经比较完善，只剩下Lambda项目（Lambda表达式）、Jigsaw（模块化支持）和Coin（语言细节进化）子项目的部分工作尚未完成，Oracle宣布JDK 1.7正式版将于2011年7月28日推出，可能会^①把不能按时完成的Lambda、Jigsaw和部分Coin放入JDK 1.8之中。JDK 1.7的主要改进包括：提供新的G1收集器、加强对非Java语言的调用、语言级的模块化支持（取决于Jigsaw项目能不能完成）、升级类加载架构，等等。

2009年4月20日，Oracle宣布正式以74亿美元的价格收购Sun公司，Java商标从

^① 在本书初稿完成之际，已收到确定消息证实Jigsaw、Lambda等项目无法在JDK 7中发布，最早将在JDK 8中提供。

此正式归 Oracle 所有（Java 语言本身并不属于哪家公司所有，它由 JCP 组织进行管理，尽管 JCP 主要是由 Sun 或者说 Oracle 所领导的）。由于此前 Oracle 已经收购了另外一家大型的中间件企业 BEA 公司，当完成对 Sun 公司的收购之后，Oracle 分别从 BEA 和 Sun 中取得了目前三大商业虚拟机的其中两个：JRockit 和 HotSpot，Oracle 宣布在未来 1 至 2 年的时间内，将把这两个优秀的虚拟机互相取长补短，最终合二为一^①。可以预见在不久的将来，Java 技术体系将会产生相当巨大的变化。

1.4 展望 Java 技术的未来

在 Java 语言诞生 10 周年（2005 年）的 SunOne 技术大会上，Java 语言之父 James Gosling 做过一个题为“Java 技术的下一个十年”的演讲。笔者不具备 James Gosling 博士那样高屋建瓴的视角，这里仅从 Java 平台中几个新生的但已经开始展现出蓬勃之势的技术发展点来看一下后续 1 至 2 个 JDK 版本内的一些很有希望的技术重点。

1.4.1 模块化

模块化是解决应用系统与技术平台越来越复杂、越来越庞大而产生的一系列问题的一个重要途径。无论是开发人员还是产品的最终用户，都不希望为了系统中的一小块功能而不得不下载、安装、部署及维护整套庞大的系统。最近几年 OSGi 技术的迅速发展正说明了通过模块化实现按需部署、降低复杂性和维护成本的需求是相当迫切的。

预计在未来的 Java 平台中，将会对模块化提供语法层面的支持。在 Java SE 7 发展初期，两个重要的 JSR 曾经试图解决依赖关系管理问题，分别是 JSR-294：Java 编程语言中的改进模块性支持（Improved Modularity Support in the Java Programming Language）和 JSR-277：Java 模块系统（Java Module System），两者分别关注 Java 模块概念的开发和部署方面。在具体实现方面，Java SE 7 中已建立了一个名为 Jigsaw（拼图）的项目来推动这两个规范在 Java 平台中转变为具体的实现。

1.4.2 混合语言

当单一的 Java 语言已经无法满足当前软件的复杂需求时，越来越多基于 Java 虚拟

^① HotRockit 项目的相关介绍：<http://hirt.se/presentations/WhatToExpect.ppt>。

除了催生出大量的新语言外，许多已经有很长历史的程序语言也出现了基于 Java 虚拟机实现的版本。这样的混合编程对许多以前使用其他语言的“老”程序员也有相当大的吸引力，软件企业投入了大量资本的现有代码资产也能被很好地保护起来。表 1-1 中列举了常见语言的 Java 虚拟机实现版本。

表 1-1 常见语言的 JVM 实现版本

语 言	基于 JVM 实现的版本
Ada	JGNAT
AWK	Jawk
C	C to Java Virtual Machine compilers
Cobol	Veryant is Cobol
ColdFusion	Adobe ColdFusion、Railo、Open BlueDragon
Common Lisp	Armed Bear Common Lisp、CLforJava、Jatha (Common LISP)
Component Pascal	Gardens Point Component Pascal
Erlang	Erjang
Forth	myForth
JavaScript	Rhino
LOGO	jLogo、XLogo
Lua	Kahlua、Luaj、Jill
Oberon-2	Canterbury Oberon-2 for JVM
Objective Caml (OCaml)	OCaml-Java
Pascal	Canterbury Pascal for JVM
PHP	IBM WebSphere sMash PHP (P8)、Caucho Quercus
Python	Jython
Rexx	IBM NetRexx
Ruby	JRuby
Scheme	Bigloo、Kawa、SISC、JScheme

1.4.3 多核并行

如今，CPU 硬件的发展方向已经从高频率转变为多核心，随着多核时代的来临，软件开发越来越关注并行编程的领域。早在 JDK 1.5 之中就已经引入 `java.util.concurrent` 包实现了一个粗粒度的并发框架，而 JDK 1.7 中将会加入的 `java.util.concurrent.forkjoin` 包则是对这个框架的一次重要扩充。Fork/Join 模式是处理并行编程的一种经典方法，如图 1-5 所示。虽然不能解决所有的问题，但是在它的适用范围之内，能够轻松地利用多个 CPU 核心提供的计算资源来协作完成一个复杂的计算任务。通过利用 Fork/Join 模式，

我们能够更加顺畅地过渡到多核的时代。

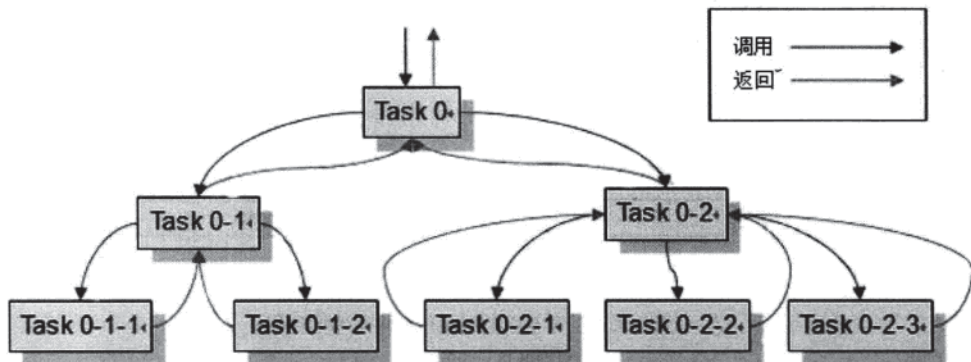


图 1-5 Fork/Join 模式示意图^①

在 JDK 外围，也出现了专为满足并行计算需求的计算框架，如 Apache 的 Hadoop Map/Reduce，这是一个简单易懂的并行框架，能够运行在由上千个商用机器组成的大型集群上，并能以一种可靠的容错方式并行处理上 TB 级别的数据集。另外，还出现了诸如 Scala、Clojure 及 Erlang 等天生就具备并行计算能力的语言。

1.4.4 进一步丰富语法

JDK 1.5 曾经对 Java 语法进行了一次扩充，这次扩充加入了自动装箱、泛型、动态注解、枚举、可变长参数、遍历循环等语法特性，使得 Java 语言的精确性和易用性有了很大的进步。在 JDK 1.7（由于进度压力，许多改进已被推迟至 JDK 1.8）中，将会对 Java 语法进行另一次大规模的扩充。Sun（Oracle）为此发起了 Coin 子项目^②来统一处理对 Java 语法的细节修改，如二进制数的原生支持、在 switch 语句中支持字符串、“<>”操作符、异常处理的改进、简化变长参数方法调用、面向资源的 try-catch-finally 语句等都是 Coin 项目之中提交的内容。另外，JSR-335（Lambda Expressions for the Java™ Programming Language）中定义的 Lambda 表达式^③也将对 Java 的语法和语言习惯产生很大的影响，函数式编程可能会成为主流。

① 图片来源：<http://www.ibm.com/developerworks/cn/java/j-lo-forkjoin/>。

② Coin 项目主页：<http://wikis.sun.com/display/ProjectCoin/Home>。

③ Lambda 项目主页：<http://openjdk.java.net/projects/lambda/>。

1.4.5 64 位虚拟机

几年之前，主流的 CPU 就开始支持 64 位架构。Java 虚拟机也在很早之前就推出了支持 64 位系统的版本。但 Java 程序运行在 64 位虚拟机上需要付出比较大的额外代价：首先是内存问题，由于指针膨胀和各种数据类型对齐补白的原因，运行于 64 位系统上的 Java 应用需要消耗更多的内存，通常要比 32 位系统额外增加 10% ~ 30% 的内存消耗；其次是多个机构的测试结果显示，64 位虚拟机的运行速度在各个测试项上几乎都全面落后于 32 位虚拟机，两者大约有 15% 左右的性能差距。

但是在 Java EE 方面，企业级应用经常需要使用超过 4G 的内存，对于 64 位虚拟机的需求是非常迫切的，由于上述的原因，许多企业应用都仍然选择使用虚拟集群等方式继续在 32 位虚拟机中进行部署。Sun 也注意到了这些问题，并做出了一些改善，在 JDK 1.6 Update 14 之后，提供了普通对象指针压缩功能（-XX:+ UseCompressedOops），在解释器解释字节码时，植入压缩指令以节省内存消耗。随着硬件的进一步发展，计算机终究会完全过渡到 64 位的时代，这是一件毫无疑问的事情，主流的虚拟机应用终究也会从 32 位发展至 64 位，而虚拟机对 64 位的支持也将会进一步完善。

1.5 实战：自己编译 JDK

想要一探 JDK 内部的实现机制，最便捷的路径之一就是自己编译一套 JDK。通过阅读和跟踪调试 JDK 源码去了解 Java 技术体系的原理，虽然门槛会高一点，但肯定会比阅读各种文章、书籍更加容易贴近本质。另外，JDK 中的很多底层方法都是 Native 的，当需要跟踪这些方法的运作或对 JDK 进行 Hack 的时候，都需要编译一套自己的 JDK。

现在网络上有不少开源的 JDK 实现可供选择，如 Apache Harmony、OpenJDK 等。考虑到 Sun 系列的 JDK 是现在使用得最广泛的 JDK 版本，本书选择了 OpenJDK 进行这次编译实战。

1.5.1 获取 JDK 源码

首先确定要使用的 JDK 版本，OpenJDK 6 和 OpenJDK 7 都是开源的，源码都可以在它们的主页（<http://openjdk.java.net/>）上找到，OpenJDK 6 的源码其实是从 OpenJDK 7

14 ❖ 第一部分 走近 Java

的某个基线中引出的，然后剥离掉 JDK 1.7 相关的代码，从而得到一份可以通过 TCK 6 的 JDK 1.6 实现，因此直接编译 OpenJDK 7 会更加“原汁原味”一些，其实这两个版本的编译过程差异并不大。

获取源码有两种方式：一种是通过 Mercurial 代码版本管理工具从 Repository 中直接取得源码（Repository 地址：<http://hg.openjdk.java.net/jdk7/jdk7/>），这是最直接的方式，从版本管理中看变更轨迹比看任何 Release Note 都来得实在，不过坏处自然是太麻烦了一些，尤其是 Mercurial 远不如 SVN、ClearCase 或 CVS 之类的版本控制工具那样普及；另外一种就是直接下载官方打包好的源码包了，可以从 Source Releases 页面（地址：<http://download.java.net/openjdk/jdk7/>）取得打包好的源码，一般来说大概一个月左右会更新一次，虽然不够及时，但的确方便了许多。笔者下载的是 OpenJDK 7 Early Access Source Build b121 版，2010 年 12 月 9 日发布的，大概 81.7MB，解压后约 308MB。

1.5.2 系统需求

如果可能，笔者建议尽量在 Linux 或 Solaris 上构建 OpenJDK，这要比在 Windows 平台上轻松许多，而且网上能找到的资料绝大部分都是在 Linux 上编译的。如果一定要在 Windows 平台上编译，建议认真阅读一下源码中的 README-builds.html 文档（无论是在 OpenJDK 网站上，还是在下载的源码包里面都有这份文档），因为编译过程中需要注意的细节非常多。虽然不至于像文档上所描述的^①那么夸张，但是如果大家是第一次编译，在上面耗费一整天乃至更多的时间都很正常。

本书在本次实战中演示的是在 32 位 Windows 7 平台下编译 x86 版的 OpenJDK（也就是 32 位的 JDK），如果需要编译 x64 版，那毫无疑问也需要一个 64 位的操作系统。另外，编译涉及的所有文件都必须存放在 NTFS 格式的文件系统中，因为 FAT32 格式无法支持大小写敏感的文件名。在官方文档上写道：编译至少需要 512MB 的内存和 600MB 的磁盘空间。如果大家耐心很好的话，512MB 的内存基本上也可以凑合使用，不过 600MB 的磁盘空间仅仅是指存放 OpenJDK 源码和相关依赖项的空间，要完成编

① Building the source code for the JDK requires a high level of technical expertise. Sun provides the source code primarily for technical experts who want to conduct research.（编译 JDK 需要很高的专业技术，Sun 提供 JDK 源码是为了技术专家进行研究之用）

译, 600MB 肯定是如何都不够的, 这次实战中所下载的工具、依赖项、源码, 全部安装和解压完成最少 (最少是指只下载 C++ 编译器, 不下载 VS 的 IDE) 需要超过 1GB 的空间。

对系统的最后一点要求就是所有的文件, 包括源码和依赖项目, 都不要放在包含中文或空格的目录里面, 这样做不是一定不可以, 只是这样会为后续建立 CYGWIN 环境带来很多额外的工作, 这是由于 Linux 和 Windows 的磁盘路径存在差别, 我们也没有必要自己给自己找麻烦。

1.5.3 构建编译环境

准备编译环境的第一步是安装一个 CYGWIN^①。这是一个在 Windows 平台下模拟 Linux 运行环境的软件, 提供了一系列的 Linux 命令支持。需要 CYGWIN 的原因是因为在编译中要使用 GNU Make 来执行 Makefile 文件 (C/C++ 程序员肯定很熟悉, 如果只使用 Java, 把它当做 C++ 版本的 ANT 看待就可以了)。安装 CYGWIN 时不能直接采用默认安装方式, 因为表 1-2 中所示的工具在默认情况下都不会安装, 但又是编译过程中需要的, 因此要在图 1-6 的安装界面中进行手工选择。

表 1-2 需要手工选择安装的 CYGWIN 工具

文件名	分类	包	描述
ar.exe	Devel	binutils	The GNU assembler, linker and binary utilities
make.exe	Devel	make	The GNU version of the 'make' utility built for CYGWIN.
m4.exe	Interpreters	m4	GNU implementation of the traditional Unix macro processor
cpio.exe	Utils	cpio	A program to manage archives of files
gawk.exe	Utils	awk	Pattern-directed scanning and processing language
file.exe	Utils	file	Determines file type using 'magic' numbers
zip.exe	Archive	zip	Package and compress (archive) files
unzip.exe	Archive	unzip	Extract compressed files in a ZIP archive
free.exe	System	procps	Display amount of free and used memory in the system

CYGWIN 安装时的定制包选择界面如图 1-6 所示。

① CYGWIN 下载地址: <http://www.cygwin.com/>。

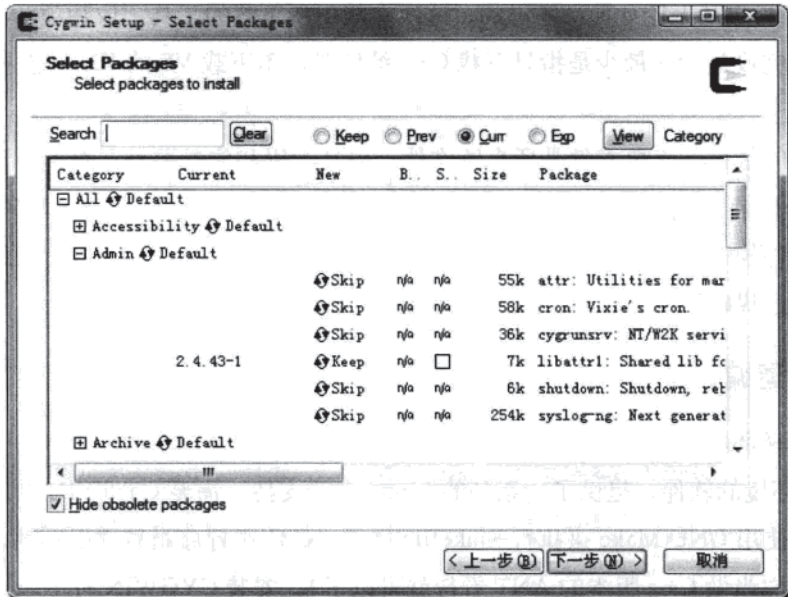


图 1-6 CYGWIN 安装界面

建立编译环境的第二步是安装编译器。JDK 中最核心的代码（Java 虚拟机及 JDK 中 Native 方法的实现等）是使用 C++ 语言及少量的 C 语言编写的，官方文档中说它们的内部开发环境是在 Microsoft Visual Studio C++ 2003（VS2003）中进行编译的，同时也在 Microsoft Visual Studio C++ 2010（VS2010）中测试过，所以最好只选择这两个编译器中的一个进行编译。如果选择 VS2010，那么在编译器之中已经包含了 Windows SDK v 7.0a，否则可能还要自己去下载这个 SDK，并且更新 PlatformSDK 目录。由于笔者没有购买 Visual Studio 2010 的 IDE，所以仅仅下载了 VS2010 Express 中提取出来的 C++ 编译器，这部分是免费的，但单独安装好编译器比较麻烦，建议读者选择使用整套 Visual Studio C++ 2010 或 Visual Studio C++ 2010 Express 版进行编译。

注意 CYGWIN 和 VS2010 安装之后都会在操作系统的 PATH 环境变量中写入自己的 bin 目录路径，必须检查并保证 VS2010 的 bin 目录一定要在 CYGWIN 的 bin 目录之前，因为这两个软件的 bin 目录之中各自都有个连接器“link.exe”，但是只有 VS2010 中的连接器可以完成 OpenJDK 的编译。

准备 JDK 编译环境的第三步是下载一个已经编译好的 JDK。这听起来也许有点滑稽——要用鸡蛋孵小鸡还真得先养一只母鸡呀？但仔细想想其实这个步骤很合理：因为 JDK 包含的各个部分（HotSpot、JDK API、JAXWS、JAXP……）有的是使用 C++ 编写的，而更多的代码则是使用 Java 自身实现的，因此编译这些 Java 代码需要用到一个可用的 JDK，官方称这个 JDK 为“Bootstrap JDK”。如果编译 OpenJDK 7，Bootstrap JDK 必须使用 JDK6 Update 14 或之后的版本，笔者选用的是 JDK6 Update 21。

最后一个步骤是下载一个 Apache ANT，JDK 中的 Java 代码部分都是使用 ANT 脚本进行编译的，ANT 版本要求在 1.6.5 以上，这部分是 Java 的基础知识，对本书的读者来说应该没有难度，笔者就不再详述。

1.5.4 准备依赖项

前面说过，OpenJDK 中开放的源码并没有达到 100%，还有极少量的无法开源的产权代码存在。OpenJDK 承诺日后将逐步使用开源实现来替换掉这部分产权代码，但至少在今天，编译 JDK 还需要这部分闭源包，官方称之为“JDK Plug”^①，它们从前面的 Source Releases 页面就可以下载到。在 Windows 平台下的 JDK Plug 是以 Jar 包的形式提供的，通过下面这条命令可以安装它：

```
java -jar jdk-7-ea-plug-b121-windows-i586-09_dec_2010.jar
```

运行后将会显示如图 1-7 所示的协议，点击 ACCEPT 接受协议，然后把 Plug 安装到指定目录即可。安装完毕后建立一个环境变量“ALT_BINARY_PLUGS_PATH”，变量值为此 JDK Plug 的安装路径，后面编译程序时需要用到它。

除了要用到 JDK Plug 外，编译时还需要引用 JDK 的运行时包，它是编译 JDK 中使用 Java 代码编写的那部分所需要的，如果仅仅是想编译一个 HotSpot 虚拟机，则可以不用。官方文档把这部分称之为“Optional Import JDK”，可以直接使用前面的 Bootstrap JDK 的运行时包，我们需要建立一个名为“ALT_JDK_IMPORT_PATH”的环境变量指向 JDK 的安装目录。

第三步是安装一个大于 2.3 版的 FreeType^②，这是一个免费的字体渲染库，JDK

^① 在 2011 年，JDK Plug 已经不在需要了，但笔者在实战时使用的是 2010 年 12 月 9 日发布的 OpenJDK b121 版，还是需要这些 JDK Plug。

^② FreeType 主页：<http://www.freetype.org/>。

18 ❖ 第一部分 走近 Java

的 Swing 部分和 JConsole 这类工具要使用到它。安装好后建立两个环境变量“ALT_FREETYPE_LIB_PATH”和“ALT_FREETYPE_HEADERS_PATH”，分别指向 FreeType 安装目录下的 bin 目录和 include 目录。另外，还有一点官方文档没有提到但必须要做的事情是把 FreeType 的 bin 目录加入到 PATH 环境变量中。

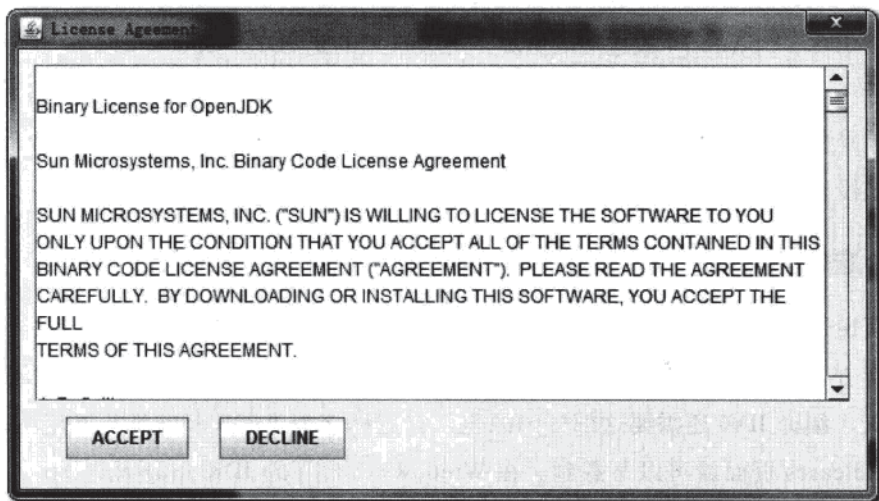


图 1-7 JDK Plug 安装协议

第四步是下载 Microsoft DirectX 9.0 SDK (Summer 2004)，安装后大约有 298MB，在微软官方网站上搜索一下就可以找到下载地址，它是免费的。安装后建立环境变量“ALT_DXSDK_PATH”指向 DirectX 9.0 SDK 的安装目录。

第五步是去寻找一个名为“MSVCR100.DLL”的动态链接库，如果读者在前面安装了全套的 Visual Studio 2010，那这个文件在本机就能找到，否则上网搜索一下也能找到单独的下载地址，大概有 744KB。建立环境变量“ALT_MSVCRRN_DLL_PATH”指向这个文件所在的目录。如果读者选择的是 VS2003，这个文件名应当为“MSVCR73.DLL”。很多软件中都包含有这个文件，如果找不到，前面下载的“Bootstrap JDK”的 bin 目录中应该也有一个，直接拿来用吧。

1.5.5 进行编译

现在，需要下载的编译环境和依赖项目都准备齐全了，最后我们还需要对系统进行

一些设置以便编译能够顺利通过。

首先执行 VS2010 中的 VCVAR32.BAT，这个批处理文件的主要目的是设置 INCLUDE、LIB 和 PATH 这几个环境变量。如果大家和笔者一样只是下载了编译器，则需要手工设置它们，各个环境变量的设置值可以参考代码清单 1-1 中的内容。批处理运行完之后建立“ALT_COMPILER_PATH”环境变量，让 Makefile 知道在哪里可以找到编译器。

再建立“ALT_BOOTDIR”和“ALT_JDK_IMPORT_PATH”两个环境变量，指向前面提到的 JDK 1.6 的安装目录，建立“ANT_HOME”指向 Apache ANT 的安装目录。建立的环境变量很多，为了避免遗漏，笔者写了一个批处理文件以供读者参考，如代码清单 1-1 所示。

代码清单 1-1 环境变量设置

```
SET ALT_BOOTDIR=D:/_DevSpace/JDK 1.6.0_21
SET ALT_BINARY_PLUGS_PATH=D:/jdkBuild/jdk7plug/openjdk-binary-plugs
SET ALT_JDK_IMPORT_PATH=D:/_DevSpace/JDK 1.6.0_21
SET ANT_HOME=D:/jdkBuild/apache-ant-1.7.0
SET ALT_MSVCRRN_DLL_PATH=D:/jdkBuild/msvcr100
SET ALT_DXSDK_PATH=D:/jdkBuild/msdxsdk
SET ALT_COMPILER_PATH=D:/jdkBuild/vcpp2010.x86/bin
SET ALT_FREETYPE_HEADERS_PATH=D:/jdkBuild/freetype-2.3.5-1-bin/include
SET ALT_FREETYPE_LIB_PATH=D:/jdkBuild/freetype-2.3.5-1-bin/bin

SET INCLUDE=D:/jdkBuild/vcpp2010.x86/include;D:/jdkBuild/vcpp2010.x86/sdk/
Include;%INCLUDE%
SET LIB=D:/jdkBuild/vcpp2010.x86/lib;D:/jdkBuild/vcpp2010.x86/sdk/Lib;%LIB%
SET LIBPATH=D:/jdkBuild/vcpp2010.x86/lib;%LIB%
SET PATH=D:/jdkBuild/vcpp2010.x86/bin;D:/jdkBuild/vcpp2010.x86/dll/x86;D:/
Software/OpenSource/cygwin/bin;%ALT_FREETYPE_LIB_PATH%;%PATH%
```

最后还需要再做两项调整，官方文档没有说明这两项，但是必须要做完才能保证编译过程顺利完成：一项是取消环境变量 JAVA_HOME，这点很简单；另外一项是尽量在英文的操作系统上编译，估计大部分读者会感到比较为难吧。如果不能在英文的系统上编译，就把系统的文字格式调整为“英语（美国）”，在控制面板中的“区域和语言”选项的第一个页签中可以设置。如果这个设置还不能更改就建立一个“BUILD_CORBA”的环境变量，将值设置为 false，取消编译 CORBA 部分。否则 Java IDL (idlj.exe) 为

*.idl 文件生成 CORBA 适配器代码的时候会产生中文注释，而这些中文注释会因为字符集的问题而导致编译失败。

完成了上述繁琐的准备工作之后，我们终于可以开始编译了。进入控制台（Cmd.exe）后运行刚才准备好的设置环境变量的批处理文件，然后输入 bash 进入 Bourne Again Shell 环境（习惯 sh 或 ksh 的读者请自便）。如果 JDK 的安装源码中存在“jdk_generic_profile.sh”这个 Shell 脚本，先执行它，笔者下载的 OpenJDK 7 B121 版没有这个文件了，所以可直接输入 make sanity 来检查我们前面所做的设置是否全部正确。如果一切顺利，几秒钟之后会有类似代码清单 1-2 所示的输出。

代码清单 1-2 make sanity 检查

```
D:\jdkBuild\openjdk7>bash
bash-3.2$ make sanity
cygwin warning:
  MS-DOS style path detected: C:/Windows/system32/wscript.exe
  Preferred POSIX equivalent is: /cygdrive/c/Windows/system32/wscript.exe
  CYGWIN environment variable option "nodosfilewarning" turns off this warning.
  Consult the user's guide for more details about POSIX paths:
    http://cygwin.com/cygwin-ug-net/using.html#using-pathnames
( cd ./jdk/make && \

.....

OpenJDK-specific settings:
  FREETYPE_HEADERS_PATH = D:/jdkBuild/freetype-2.3.5-1-bin/include
  ALT_FREETYPE_HEADERS_PATH = D:/jdkBuild/freetype-2.3.5-1-bin/include
  FREETYPE_LIB_PATH = D:/jdkBuild/freetype-2.3.5-1-bin/bin
  ALT_FREETYPE_LIB_PATH = D:/jdkBuild/freetype-2.3.5-1-bin/bin

OPENJDK Import Binary Plug Settings:
  IMPORT_BINARY_PLUGS = true
  BINARY_PLUGS_JARFILE = D:/jdkBuild/jdk7plug/openjdk-binary-plugs/jre/lib/rt-closed.jar
  ALT_BINARY_PLUGS_JARFILE =
  BINARY_PLUGS_PATH = D:/jdkBuild/jdk7plug/openjdk-binary-plugs
  ALT_BINARY_PLUGS_PATH = D:/jdkBuild/jdk7plug/openjdk-binary-plugs
  BUILD_BINARY_PLUGS_PATH = J:/re/jdk/1.7.0/promoted/latest/openjdk/binaryplugs
  ALT_BUILD_BINARY_PLUGS_PATH =
  PLUG_LIBRARY_NAMES =
```

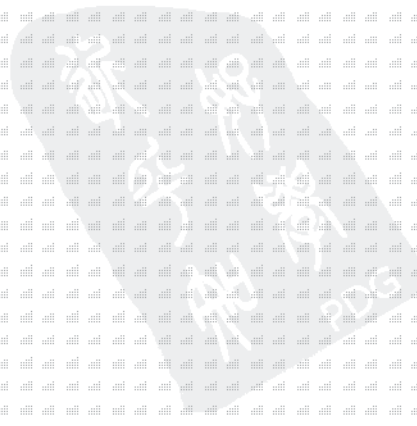
```
Previous JDK Settings:
PREVIOUS_RELEASE_PATH = USING-PREVIOUS_RELEASE_IMAGE
  ALT_PREVIOUS_RELEASE_PATH =
PREVIOUS_JDK_VERSION = 1.6.0
  ALT_PREVIOUS_JDK_VERSION =
PREVIOUS_JDK_FILE =
  ALT_PREVIOUS_JDK_FILE =
PREVIOUS_JRE_FILE =
  ALT_PREVIOUS_JRE_FILE =
PREVIOUS_RELEASE_IMAGE = D:/_DevSpace/JDK 1.6.0_21
  ALT_PREVIOUS_RELEASE_IMAGE =
Sanity check passed.
```

Makefile 的 Sanity 检查过程输出了编译所需的所有环境变量，如果看到“Sanity check passed.”说明检查过程通过了，可以输入“make”执行整个 Makefile，然后去喝杯下午茶再回来。笔者的 Core i5 / 4GB RAM 的机器编译整个 JDK 大概需要半个多小时。如果失败，则需要根据系统输出的失败原因，回头再检查一下对应的设置，并且最好在下一编译之前先执行“make clean”来清理掉上次编译遗留的文件。

编译完成之后，打开 OpenJDK 源码下的 build 目录，看看是不是已经有一个编译好的 JDK 在那里等着了？执行一下“java -version”，看到以自己的机器命名的 JDK 了吧，很有成就感吧？

1.6 本章小结

本章介绍了 Java 技术体系的过去、现在和未来的发展趋势，并通过实践的方式介绍了如何自己来独立编译一个 OpenJDK 7。作为全书的引言部分，本章建立了后文研究所必需的环境。在了解 Java 技术的来龙去脉后，后面的章节将分为四部分去讲解 Java 在内存管理、Class 文件结构与执行引擎、编译器优化和多线程并发方面的实现原理。





第二部分

自动内存管理机制

- 第 2 章 Java 内存区域与内存溢出异常
- 第 3 章 垃圾收集器与内存分配策略
- 第 4 章 虚拟机性能监控与故障处理工具
- 第 5 章 调优案例分析与实战



第 2 章 Java 内存区域与内存溢出异常

本章主要内容

- 概述
- 运行时数据区域
- 对象访问
- 实战：OutOfMemoryError 异常

Java 与 C++ 之间有一堵由内存动态分配和垃圾收集技术所围成的高墙，墙外面的人想进去，墙里面的人却想出来。

2.1 概述

对于从事 C 和 C++ 程序开发的开发人员来说，在内存管理领域，他们既是拥有最高权力的皇帝，又是从事最基础工作的劳动人民——既拥有每一个对象的“所有权”，又担负着每一个对象生命开始到终结的维护责任。

对于 Java 程序员来说，在虚拟机的自动内存管理机制的帮助下，不再需要为每一个 new 操作去写配对的 delete/free 代码，而且不容易出现内存泄漏和内存溢出问题，看起来由虚拟机管理内存一切都很美好。不过，也正是因为 Java 程序员把内存控制的权力交给了 Java 虚拟机，一旦出现内存泄漏和溢出方面的问题，如果不了解虚拟机是怎样使用内存的，那排查错误将会成为一项异常艰难的工作。

本章是第二部分的第 1 章，笔者将从概念上介绍 Java 虚拟机内存的各个区域，讲解这些区域的作用、服务对象以及其中可能产生的问题，这是翻越虚拟机内存管理这堵围墙的第一步。

2.2 运行时数据区域

Java 虚拟机在执行 Java 程序的过程中会把它所管理的内存划分为若干个不同的数据区域。这些区域都有各自的用途，以及创建和销毁的时间，有的区域随着虚拟机进程的启动而存在，有些区域则是依赖用户线程的启动和结束而建立和销毁。根据《Java 虚拟机规范（第 2 版）》的规定，Java 虚拟机所管理的内存将会包括以下几个运行时数据区域，如图 2-1 所示。

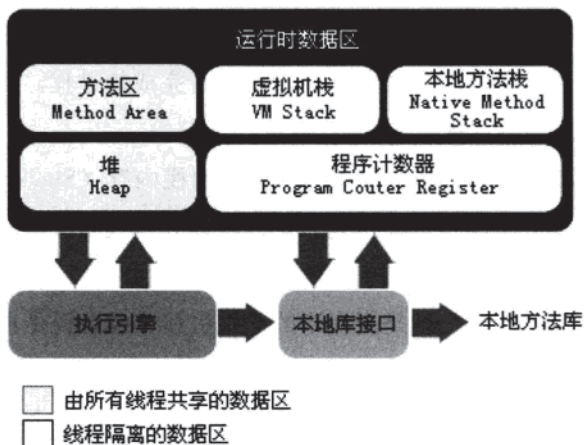


图 2-1 Java 虚拟机运行时数据区

2.2.1 程序计数器

程序计数器（Program Counter Register）是一块较小的内存空间，它的作用可以看做是当前线程所执行的字节码的行号指示器。在虚拟机的概念模型里（仅是概念模型，各种虚拟机可能会通过一些更高效的方式去实现），字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。

由于 Java 虚拟机的多线程是通过线程轮流切换并分配处理器执行时间的方式来实现的，在任何一个确定的时刻，一个处理器（对于多核处理器来说是一个内核）只会执行一条线程中的指令。因此，为了线程切换后能恢复到正确的执行位置，每条线程都需要有一个独立的程序计数器，各条线程之间的计数器互不影响，独立存储，我们称这类内存区域为“线程私有”的内存。

如果线程正在执行的是一个 Java 方法，这个计数器记录的是正在执行的虚拟机字节码指令的地址；如果正在执行的是 Native 方法，这个计数器值则为空（Undefined）。此内存区域是唯一一个在 Java 虚拟机规范中没有规定任何 `OutOfMemoryError` 情况的区域。

2.2.2 Java 虚拟机栈

与程序计数器一样，Java 虚拟机栈（Java Virtual Machine Stacks）也是线程私有的，它的生命周期与线程相同。虚拟机栈描述的是 Java 方法执行的内存模型：每个方法被执行的时候都会同时创建一个栈帧（Stack Frame^①）用于存储局部变量表、操作栈、动态链接、方法出口等信息。每一个方法被调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。

经常有人把 Java 内存区分为堆内存（Heap）和栈内存（Stack），这种分法比较粗糙，Java 内存区域的划分实际上远比这复杂。这种划分方式的流行只能说明大多数程序员最关注的、与对象内存分配关系最密切的内存区域是这两块。其中所指的“堆”在后面会专门讲述，而所指的“栈”就是现在讲的虚拟机栈，或者说是虚拟机栈中的局部变量表部分。

局部变量表存放了编译期可知的各种基本数据类型（boolean、byte、char、short、int、float、long、double）、对象引用（reference 类型，它不等同于对象本身，根据不同的虚拟机实现，它可能是一个指向对象起始地址的引用指针，也可能指向一个代表对象的句柄或者其他与此对象相关的位置）和 `returnAddress` 类型（指向了一条字节码指令的地址）。

其中 64 位长度的 long 和 double 类型的数据会占用 2 个局部变量空间（Slot），其余的数据类型只占用 1 个。局部变量表所需的内存空间在编译期间完成分配，当进入一个方法时，这个方法需要在帧中分配多大的局部变量空间是完全确定的，在方法运行期间不会改变局部变量表的大小。

在 Java 虚拟机规范中，对这个区域规定了两种异常状况：如果线程请求的栈深度大于虚拟机所允许的深度，将抛出 `StackOverflowError` 异常；如果虚拟机栈可以动态扩展（当前大部分的 Java 虚拟机都可动态扩展，只不过 Java 虚拟机规范中也允许固定长度的虚拟机栈），当扩展时无法申请到足够的内存时会抛出 `OutOfMemoryError` 异常。

① 栈帧是方法运行期的基础数据结构，在本书的第 8 章中会对帧进行详细讲解。

2.2.3 本地方法栈

本地方法栈 (Native Method Stacks) 与虚拟机栈所发挥的作用是非常相似的, 其区别不过是虚拟机栈为虚拟机执行 Java 方法 (也就是字节码) 服务, 而本地方法栈则是为虚拟机使用到的 Native 方法服务。虚拟机规范中对本地方法栈中的方法使用的语言、使用方式与数据结构并没有强制规定, 因此具体的虚拟机可以自由实现它。甚至有的虚拟机 (譬如 Sun HotSpot 虚拟机) 直接就把本地方法栈和虚拟机栈合二为一。与虚拟机栈一样, 本地方法栈区域也会抛出 `StackOverflowError` 和 `OutOfMemoryError` 异常。

2.2.4 Java 堆

对于大多数应用来说, Java 堆 (Java Heap) 是 Java 虚拟机所管理的内存中最大的一块。Java 堆是被所有线程共享的一块内存区域, 在虚拟机启动时创建。此内存区域的唯一目的就是存放对象实例, 几乎所有的对象实例都在这里分配内存。这一点在 Java 虚拟机规范中的描述是: 所有的对象实例以及数组都要在堆上分配^①, 但是随着 JIT 编译器的发展与逃逸分析技术的逐渐成熟, 栈上分配、标量替换^②优化技术将会导致一些微妙的变化发生, 所有的对象都分配在堆上也渐渐变得不是那么“绝对”了。

Java 堆是垃圾收集器管理的主要区域, 因此很多时候也被称做“GC 堆” (Garbage Collected Heap, 幸好国内没翻译成“垃圾堆”)。如果从内存回收的角度看, 由于现在收集器基本都是采用的分代收集算法, 所以 Java 堆中还可以细分为: 新生代和老年代; 再细致一点的有 Eden 空间、From Survivor 空间、To Survivor 空间等。如果从内存分配的角度看, 线程共享的 Java 堆中可能划分出多个线程私有的分配缓冲区 (Thread Local Allocation Buffer, TLAB)。不过, 无论如何划分, 都与存放内容无关, 无论哪个区域, 存储的都仍然是对象实例, 进一步划分的目的是为了更快地回收内存, 或者更快地分配内存。在本章中, 我们仅仅针对内存区域的作用进行讨论, Java 堆中的上述各个区域的分配和回收等细节将会是下一章的主题。

① Java 虚拟机规范中的原文: The heap is the runtime data area from which memory for all class instances and arrays is allocated.

② 逃逸分析与标量替换的相关内容, 请参见第 11 章的相关内容。

28 ❖ 第二部分 自动内存管理机制

根据 Java 虚拟机规范的规定，Java 堆可以处于物理上不连续的内存空间中，只要逻辑上是连续的即可，就像我们的磁盘空间一样。在实现时，既可以实现成固定大小的，也可以是可扩展的，不过当前主流的虚拟机都是按照可扩展来实现的（通过 `-Xmx` 和 `-Xms` 控制）。如果在堆中没有内存完成实例分配，并且堆也无法再扩展时，将会抛出 `OutOfMemoryError` 异常。

2.2.5 方法区

方法区（Method Area）与 Java 堆一样，是各个线程共享的内存区域，它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。虽然 Java 虚拟机规范把方法区描述为堆的一个逻辑部分，但是它却有一个别名叫做 `Non-Heap`（非堆），目的应该是与 Java 堆区分开来。

对于习惯在 HotSpot 虚拟机上开发和部署程序的开发者来说，很多人愿意把方法区称为“永久代”（Permanent Generation），本质上两者并不等价，仅仅是因为 HotSpot 虚拟机的设计团队选择把 GC 分代收集扩展至方法区，或者说使用永久代来实现方法区而已。对于其他虚拟机（如 BEA JRockit、IBM J9 等）来说是不存在永久代的概念的。即使是 HotSpot 虚拟机本身，根据官方发布的路线图信息，现在也有放弃永久代并“搬家”至 Native Memory 来实现方法区的规划了。

Java 虚拟机规范对这个区域的限制非常宽松，除了和 Java 堆一样不需要连续的内存和可以选择固定大小或者可扩展外，还可以选择不实现垃圾收集。相对而言，垃圾收集行为在这个区域是比较少出现的，但并非数据进入了方法区就如永久代的名字一样“永久”存在了。这个区域的内存回收目标主要是针对常量池的回收和对类型的卸载，一般来说这个区域的回收“成绩”比较难以令人满意，尤其是类型的卸载，条件相当苛刻，但是这部分区域的回收确实是有必要的。在 Sun 公司的 BUG 列表中，曾出现过的若干个严重的 BUG 就是由于低版本的 HotSpot 虚拟机对此区域未完全回收而导致内存泄漏。

根据 Java 虚拟机规范的规定，当方法区无法满足内存分配需求时，将抛出 `OutOfMemoryError` 异常。

2.2.6 运行时常量池

运行时常量池 (Runtime Constant Pool) 是方法区的一部分。Class 文件中除了有类的版本、字段、方法、接口等描述等信息外, 还有一项信息是常量池 (Constant Pool Table), 用于存放编译期生成的各种字面量和符号引用, 这部分内容将在类加载后存放方法区的运行时常量池中。

Java 虚拟机对 Class 文件的每一部分 (自然也包括常量池) 的格式都有严格的规定, 每一个字节用于存储哪种数据都必须符合规范上的要求, 这样才会被虚拟机认可、装载和执行。但对于运行时常量池, Java 虚拟机规范没有做任何细节的要求, 不同的提供商实现的虚拟机可以按照自己的需要来实现这个内存区域。不过, 一般来说, 除了保存 Class 文件中描述的符号引用外, 还会把翻译出来的直接引用也存储在运行时常量池中^①。

运行时常量池相对于 Class 文件常量池的另外一个重要特征是具备动态性, Java 语言并不要求常量一定只能在编译期产生, 也就是并非预置入 Class 文件中常量池的内容才能进入方法区运行时常量池, 运行期间也可能将新的常量放入池中, 这种特性被开发人员利用得比较多的便是 String 类的 intern() 方法。

既然运行时常量池是方法区的一部分, 自然会受到方法区内存的限制, 当常量池无法再申请到内存时会抛出 OutOfMemoryError 异常。

2.2.7 直接内存

直接内存 (Direct Memory) 并不是虚拟机运行时数据区的一部分, 也不是 Java 虚拟机规范中定义的内存区域, 但是这部分内存也被频繁地使用, 而且也可能导致 OutOfMemoryError 异常出现, 所以我们放到这里一起讲解。

在 JDK 1.4 中新加入了 NIO (New Input/Output) 类, 引入了一种基于通道 (Channel) 与缓冲区 (Buffer) 的 I/O 方式, 它可以使用 Native 函数库直接分配堆外内存, 然后通过一个存储在 Java 堆里面的 DirectByteBuffer 对象作为这块内存的引用进行操作。这样能在一些场景中显著提高性能, 因为避免了在 Java 堆和 Native 堆中来回复制数据。

^① 关于 Class 文件格式和符号引用等概念可参见第 6 章。

显然，本机直接内存的分配不会受到 Java 堆大小的限制，但是，既然是内存，则肯定还是会受到本机总内存（包括 RAM 及 SWAP 区或者分页文件）的大小及处理器寻址空间的限制。服务器管理员配置虚拟机参数时，一般会根据实际内存设置 `-Xmx` 等参数信息，但经常会忽略掉直接内存，使得各个内存区域的总和大于物理内存限制（包括物理上的和操作系统级的限制），从而导致动态扩展时出现 `OutOfMemoryError` 异常。

2.3 对象访问

介绍完 Java 虚拟机的运行时数据区之后，我们就可以来探讨一个问题：在 Java 语言中，对象访问是如何进行的？对象访问在 Java 语言中无处不在，是最普通的程序行为，但即使是最简单的访问，也会涉及 Java 栈、Java 堆、方法区这三个最重要内存区域之间的关联关系，如下面的这句代码：

```
Object obj = new Object();
```

假设这句代码出现在方法体中，那“`Object obj`”这部分的语义将会反映到 Java 栈的本地变量表中，作为一个 `reference` 类型数据出现。而“`new Object()`”这部分的语义将会反映到 Java 堆中，形成一块存储了 `Object` 类型所有实例数据值（Instance Data，对象中各个实例字段的数据）的结构化内存，根据具体类型以及虚拟机实现的对象内存布局（Object Memory Layout）的不同，这块内存的长度是不固定的。另外，在 Java 堆中还必须包含能查找到此对象类型数据（如对象类型、父类、实现的接口、方法等）的地址信息，这些类型数据则存储在方法区中。

由于 `reference` 类型在 Java 虚拟机规范里面只规定了一个指向对象的引用，并没有定义这个引用应该通过哪种方式去定位，以及访问到 Java 堆中的对象的具体位置，因此不同虚拟机实现的对象访问方式会有所不同，主流访问方式有两种：使用句柄和直接指针。

- 如果使用句柄访问方式，Java 堆中将会划分出一块内存来作为句柄池，`reference` 中存储的就是对象的句柄地址，而句柄中包含了对象实例数据和类型数据各自的具体地址信息，如图 2-2 所示。

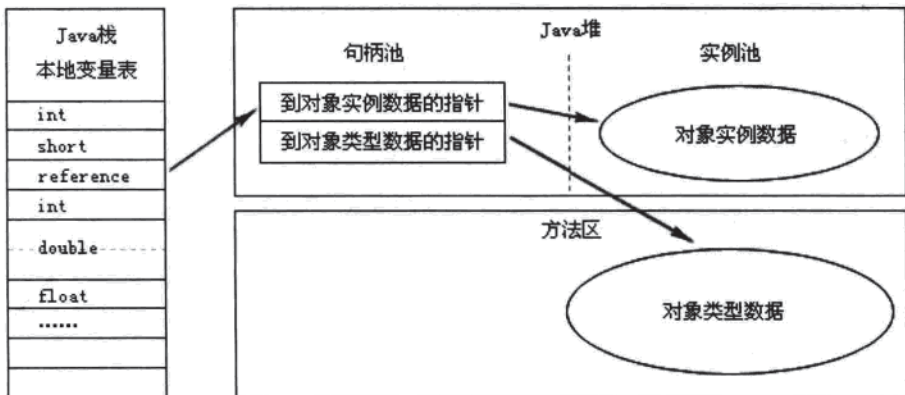


图 2-2 通过句柄访问对象

- 如果使用直接指针访问方式，Java 堆对象的布局中就必须考虑如何放置访问类型数据的相关信息，reference 中直接存储的就是对象地址，如图 2-3 所示。

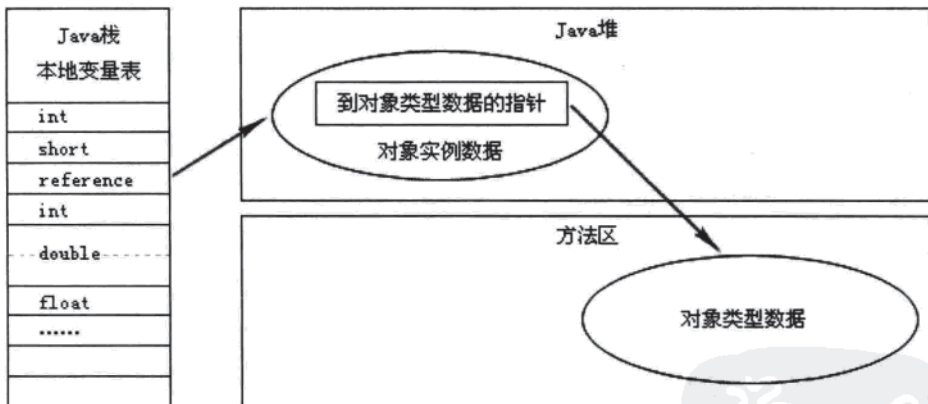


图 2-3 通过直接指针访问对象

这两种对象的访问方式各有优势，使用句柄访问方式的最大好处就是 reference 中存储的是稳定的句柄地址，在对象被移动（垃圾收集时移动对象是非常普遍的行为）时只会改变句柄中的实例数据指针，而 reference 本身不需要被修改。

使用直接指针访问方式的最大好处就是速度更快，它节省了一次指针定位的时间开销，由于对象的访问在 Java 中非常频繁，因此这类开销积少成多后也是一项非常可观的执行成本。就本书讨论的主要虚拟机 Sun HotSpot 而言，它是使用第二种方式进行对象

访问的，但从整个软件开发的范围来看，各种语言和框架使用句柄来访问的情况也十分常见。

2.4 实战：OutOfMemoryError 异常

在 Java 虚拟机规范的描述中，除了程序计数器外，虚拟机内存的其他几个运行时区域都有发生 OutOfMemoryError（下文称 OOM）异常的可能，本节将通过若干实例来验证异常发生的场景（代码清单 2-1 至代码清单 2-6 的几段简单代码），并且会初步介绍几个与内存相关的最基本的虚拟机参数。

本节内容的目的有两个：第一，通过代码验证 Java 虚拟机规范中描述的各个运行时区域储存的内容；第二，希望读者在工作中遇到实际的内存溢出异常时，能根据异常的信息快速判断是哪个区域的内存溢出，知道怎样的代码可能会导致这些区域的内存溢出，以及出现这些异常后该如何处理。

下面代码的开头都注释了执行时所需要设置的虚拟机启动参数（注释中“VM Args”后面跟着的参数），这些参数对实验的结果有直接影响，请读者调试代码的时候不要忽略掉。如果读者使用控制台命令来执行程序，那直接跟在 Java 命令之后书写就可以。如果读者使用 Eclipse IDE，则可以参考图 2-4 在 Debug/Run 页签中的设置。

下文的代码都是基于 Sun HotSpot 17.1-b03（JDK 1.6 Update 22 中带的虚拟机）运行的，对于不同公司的不同版本的虚拟机，参数和程序运行的结果可能会有所差别。

2.4.1 Java 堆溢出

Java 堆用于储存对象实例，我们只要不断地创建对象，并且保证 GC Roots 到对象之间有可达路径来避免垃圾回收机制清除这些对象，就会在对象数量到达最大堆的容量限制后产生内存溢出异常。

代码清单 2-1 中限制 Java 堆的大小为 20MB，不可扩展（将堆的最小值 -Xms 参数与最大值 -Xmx 参数设置为一样即可避免堆自动扩展），通过参数 -XX:+HeapDumpOnOutOfMemoryError 可以让虚拟机在出现内存溢出异常时 Dump 出当前的内存堆转储快照以便事后进行分析^①。

^① 关于堆转储快照文件分析方面的内容，可参见第 4 章。

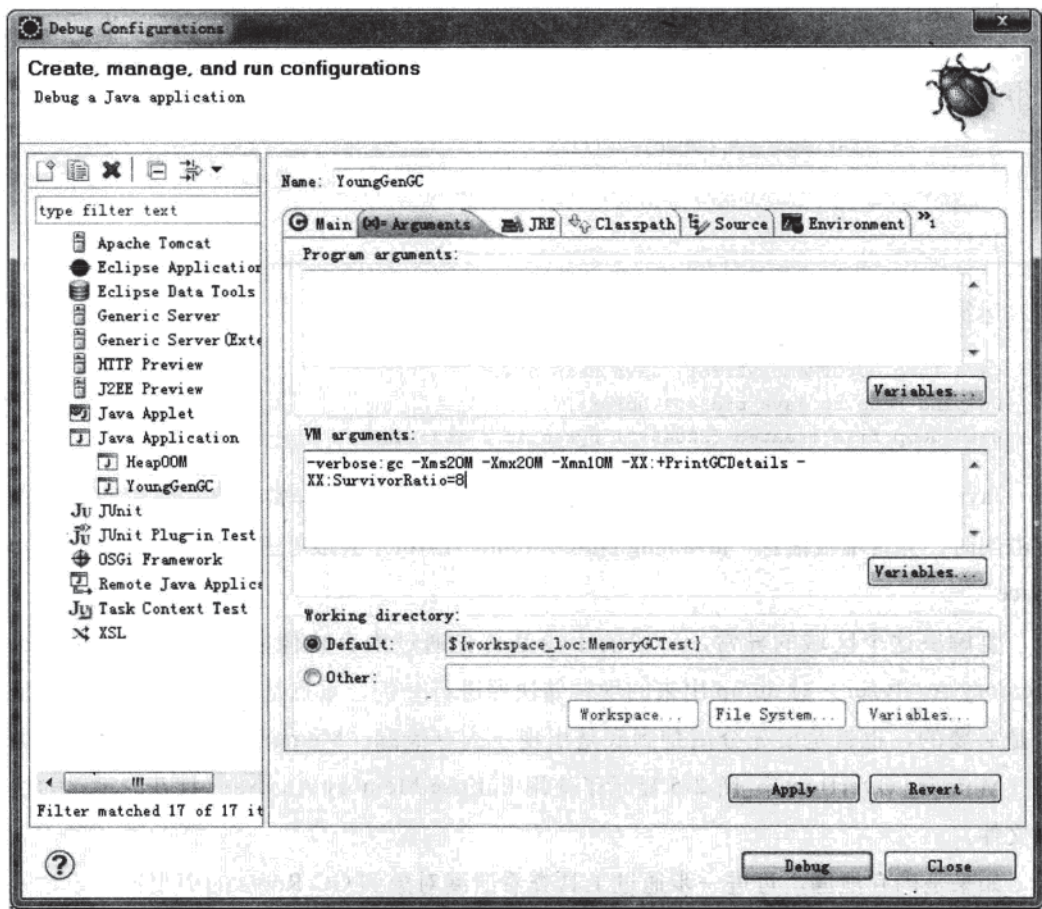


图 2-4 在 Eclipse 的 Debug 页签中设置虚拟机参数

代码清单 2-1 Java 堆内存溢出异常测试

```

/**
 * VM Args : -Xms20m -Xmx20m -XX:+HeapDumpOnOutOfMemoryError
 * @author zzm
 */
public class HeapOOM {

    static class OOMObject {
    }
}

```


34 ❖ 第二部分 自动内存管理机制

```
public static void main(String[] args) {  
    List<OOMObject> list = new ArrayList<OOMObject>();  
  
    while (true) {  
        list.add(new OOMObject());  
    }  
}
```

运行结果：

```
java.lang.OutOfMemoryError: Java heap space  
Dumping heap to java_pid3404.hprof ...  
Heap dump file created [22045981 bytes in 0.663 secs]
```

Java 堆内存的 OOM 异常是实际应用中常见的内存溢出异常情况。出现 Java 堆内存溢出时，异常堆栈信息“java.lang.OutOfMemoryError”会跟着进一步提示“Java heap space”。

要解决这个区域的异常，一般的手段是首先通过内存映像分析工具（如 Eclipse Memory Analyzer）对 dump 出来的堆转储快照进行分析，重点是确认内存中的对象是否是必要的，也就是要先分清楚到底是出现了内存泄漏（Memory Leak）还是内存溢出（Memory Overflow）。图 2-5 显示了使用 Eclipse Memory Analyzer 打开的堆转储快照文件。

如果是内存泄漏，可进一步通过工具查看泄漏对象到 GC Roots 的引用链。于是就能找到泄漏对象是通过怎样的路径与 GC Roots 相关联并导致垃圾收集器无法自动回收它们的。掌握了泄漏对象的类型信息，以及 GC Roots 引用链的信息，就可以比较准确地定位出泄漏代码的位置。

如果不存在泄漏，换句话说就是内存中的对象确实都还必须存活着，那就应当检查虚拟机的堆参数（-Xmx 与 -Xms），与机器物理内存对比看是否还可以调大，从代码上检查是否存在某些对象生命周期过长、持有状态时间过长的情况，尝试减少程序运行期的内存消耗。

以上是处理 Java 堆内存问题的简略思路，处理这些问题所需要的知识、工具与经验是后面三章的主题。

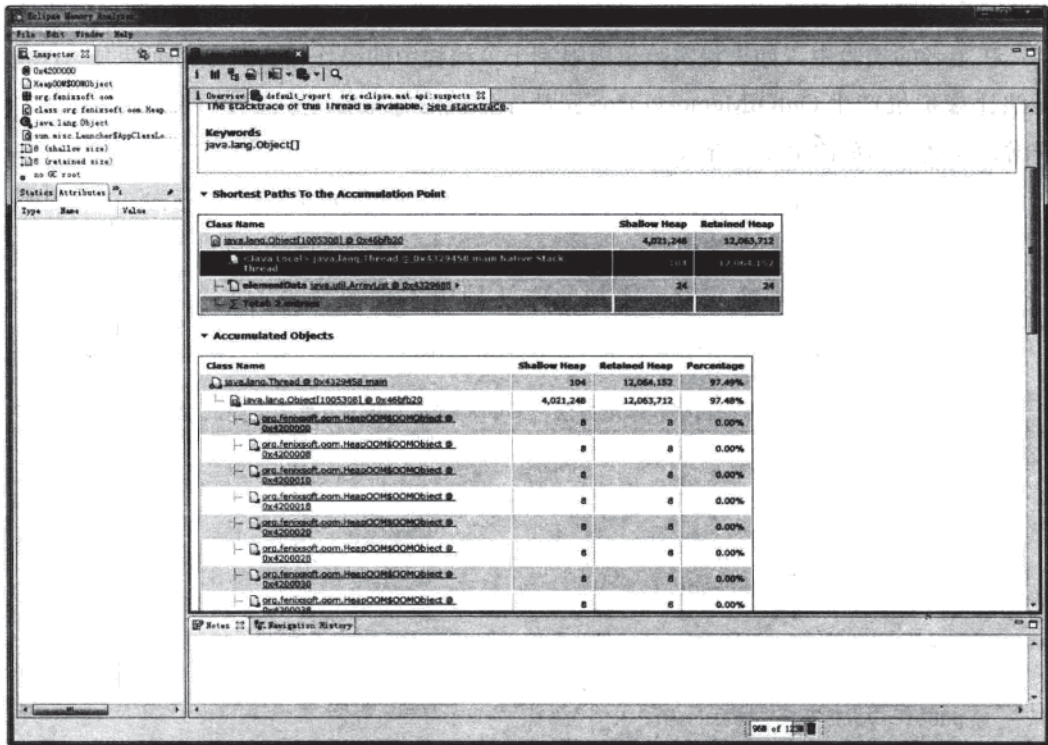


图 2-5 使用 Eclipse Memory Analyzer 打开的堆转储快照文件

2.4.2 虚拟机栈和本地方法栈溢出

由于在 HotSpot 虚拟机中并不区分虚拟机栈和本地方法栈，因此对于 HotSpot 来说，-Xoss 参数（设置本地方法栈大小）虽然存在，但实际是无效的，栈容量只由 -Xss 参数设定。关于虚拟机栈和本地方法栈，在 Java 虚拟机规范中描述了两种异常：

- 如果线程请求的栈深度大于虚拟机所允许的最大深度，将抛出 StackOverflowError 异常。
- 如果虚拟机在扩展栈时无法申请到足够的内存空间，则抛出 OutOfMemoryError 异常。

这里把异常分成两种情况看似更加严谨，但却存在着一些互相重叠的地方：当栈空间无法继续分配时，到底是内存太小，还是已使用的栈空间太大，其本质上只是对同一

36 ❖ 第二部分 自动内存管理机制

件事情的两种描述而已。

在笔者的实验中，如果将实验范围限制于单线程中的操作，尝试了下面两种方法均无法让虚拟机产生 `OutOfMemoryError` 异常，尝试的结果都是获得 `StackOverflowError` 异常，测试代码如清单 2-2 所示。

- 使用 `-Xss` 参数减少栈内存容量。结果：抛出 `StackOverflowError` 异常，异常出现时输出的栈深度相应缩小。
- 定义了大量的本地变量，增加此方法帧中本地变量表的长度。结果：抛出 `StackOverflowError` 异常时输出的栈深度相应缩小。

代码清单 2-2 虚拟机栈和本地方法栈 OOM 测试（仅作为第 1 点测试程序）

```
/**
 * VM Args: -Xss128k
 * @author zzm
 */
public class JavaVMStackSOF {

    private int stackLength = 1;

    public void stackLeak() {
        stackLength++;
        stackLeak();
    }

    public static void main(String[] args) throws Throwable {
        JavaVMStackSOF oom = new JavaVMStackSOF();
        try {
            oom.stackLeak();
        } catch (Throwable e) {
            System.out.println("stack length:" + oom.stackLength);
            throw e;
        }
    }
}
```

运行结果：

```
stack length:2402
Exception in thread "main" java.lang.StackOverflowError
    at org.fenixsoft.oom.VMStackSOF.leak(VMStackSOF.java:20)
```

```
at org.fenixsoft.oom.VMStackSOF.leak(VMStackSOF.java:21)
at org.fenixsoft.oom.VMStackSOF.leak(VMStackSOF.java:21)
……后续异常栈信息省略
```

实验结果表明：在单个线程下，无论是由于栈帧太大，还是虚拟机栈容量太小，当内存无法分配的时候，虚拟机抛出的都是 `StackOverflowError` 异常。

如果测试时不限于单线程，通过不断地建立线程的方式倒是可以产生内存溢出异常，如代码清单 2-3 所示。但是，这样产生的内存溢出异常与栈空间是否足够大并不存在任何联系，或者准确地说，在这种情况下，给每个线程的栈分配的内存越大，反而越容易产生内存溢出异常。

原因其实不难理解，操作系统分配给每个进程的内存是有限制的，譬如 32 位的 Windows 限制为 2GB。虚拟机提供了参数来控制 Java 堆和方法区的这两部分内存的最大值。剩余的内存为 2GB（操作系统限制）减去 `Xmx`（最大堆容量），再减去 `MaxPermSize`（最大方法区容量），程序计数器消耗内存很小，可以忽略掉。如果虚拟机进程本身耗费的内存不计算在内，剩下的内存就由虚拟机栈和本地方法栈“瓜分”了。每个线程分配到的栈容量越大，可以建立的线程数量自然就越少，建立线程时就越容易把剩下的内存耗尽。

这一点读者需要在开发多线程应用的时候特别注意，出现 `StackOverflowError` 异常时有错误堆栈可以阅读，相对来说，比较容易找到问题的所在。而且，如果使用虚拟机默认参数，栈深度在大多数情况下（因为每个方法压入栈的帧大小并不是一样的，所以只能说大多数情况下）达到 1000 ~ 2000 完全没有问题，对于正常的方法调用（包括递归），这个深度应该完全够用了。但是，如果是建立过多线程导致的内存溢出，在不能减少线程数或者更换 64 位虚拟机的情况下，就只能通过减少最大堆和减少栈容量来换取更多的线程。如果没有这方面的经验，这种通过“减少内存”的手段来解决内存溢出的方式会比较难以想到。

代码清单 2-3 创建线程导致内存溢出异常

```
/**
 * VM Args: -Xss2M
 * @author zzm
 */
public class JavaVMStackOOM {

    private void dontStop() {
        while (true) {
```

```
    }  
}  
  
public void stackLeakByThread() {  
    while (true) {  
        Thread thread = new Thread(new Runnable() {  
            @Override  
            public void run() {  
                dontStop();  
            }  
        });  
        thread.start();  
    }  
}  
  
public static void main(String[] args) throws Throwable {  
    JavaVMStackOOM oom = new JavaVMStackOOM();  
    oom.stackLeakByThread();  
}  
}
```

注意 特别提示一下，如果读者要尝试运行上面这段代码，记得要先保存当前的工作，由于在 Windows 平台的虚拟机中，Java 的线程是映射到操作系统的内核线程上的^①，所以上述代码执行时有较大的风险，可能会导致操作系统假死。

运行结果：

```
Exception in thread "main" java.lang.OutOfMemoryError: unable to create new native  
thread
```

2.4.3 运行时常量池溢出

如果要向运行时常量池中添加内容，最简单的做法就是使用 `String.intern()` 这个 Native 方法。该方法的作用是：如果池中已经包含一个等于此 `String` 对象的字符串，则返回代表池中这个字符串的 `String` 对象；否则，将此 `String` 对象包含的字符串添加到常量池中，并且返回此 `String` 对象的引用。由于常量池分配在方法区内，我们可以通过 `-XX:PermSize` 和 `-XX:MaxPermSize` 限制方法区的大小，从而间接限制其中常量池的容量，如代码清单 2-4 所示。

① 关于虚拟机线程实现方面的内容可以参考本书第 12 章。

代码清单 2-4 运行时常量池导致的内存溢出异常

```
/**
 * VM Args : -XX:PermSize=10M -XX:MaxPermSize=10M
 * @author zzm
 */
public class RuntimeConstantPoolOOM {

    public static void main(String[] args) {
        // 使用 List 保持着常量池引用, 避免 Full GC 回收常量池行为
        List<String> list = new ArrayList<String>();
        // 10MB 的 PermSize 在 integer 范围内足够产生 OOM 了
        int i = 0;
        while (true) {
            list.add(String.valueOf(i++).intern());
        }
    }
}
```

运行结果：

```
Exception in thread "main" java.lang.OutOfMemoryError: PermGen space
at java.lang.String.intern(Native Method)
at org.fenixsoft.oom.RuntimeConstantPoolOOM.main(RuntimeConstantPoolOOM.java:18)
```

从运行结果中可以看到，运行时常量池溢出，在 `OutOfMemoryError` 后面跟随的提示信息是“PermGen space”，说明运行时常量池属于方法区（HotSpot 虚拟机中的永久代）的一部分。

2.4.4 方法区溢出

方法区用于存放 Class 的相关信息，如类名、访问修饰符、常量池、字段描述、方法描述等。对于这个区域的测试，基本的思路是运行时产生大量的类去填满方法区，直到溢出。虽然直接使用 Java SE API 也可以动态产生类（如反射时的 `GeneratedConstructorAccessor` 和动态代理等），但在本次实验中操作起来比较麻烦。在代码清单 2-5 中，笔者借助 `CGLib`^① 直接操作字节码运行时，生成了大量的动态类。

^① CGLib 开源项目：<http://cglib.sourceforge.net/>。

40 ❖ 第二部分 自动内存管理机制

值得特别注意的是，我们在这个例子中模拟的场景并非纯粹是一个实验，这样的应用经常会出现在实际应用中：当前的很多主流框架，如 Spring 和 Hibernate 对类进行增强时，都会使用到 CGLib 这类字节码技术，增强的类越多，就需要越大的方法区来保证动态生成的 Class 可以加载入内存。

代码清单 2-5 借助 CGLib 使得方法区出现内存溢出异常

```
/**
 * VM Args: -XX:PermSize=10M -XX:MaxPermSize=10M
 * @author zzm
 */
public class JavaMethodAreaOOM {

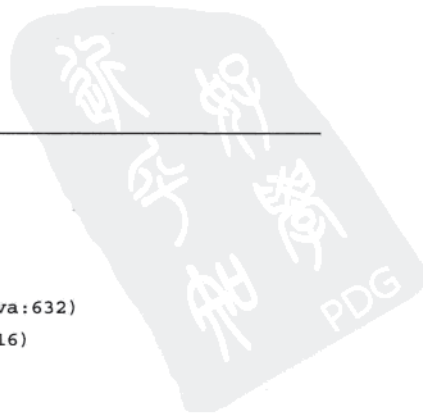
    public static void main(String[] args) {
        while (true) {
            Enhancer enhancer = new Enhancer();
            enhancer.setSuperclass(OOMObject.class);
            enhancer.setUseCache(false);
            enhancer.setCallback(new MethodInterceptor() {
                public Object intercept(Object obj, Method method, Object[]
args, MethodProxy proxy) throws Throwable {
                    return proxy.invokeSuper(obj, args);
                }
            });
            enhancer.create();
        }

        static class OOMObject {

        }
    }
}
```

运行结果：

```
Caused by: java.lang.OutOfMemoryError: PermGen space
at java.lang.ClassLoader.defineClass1(Native Method)
at java.lang.ClassLoader.defineClassCond(ClassLoader.java:632)
at java.lang.ClassLoader.defineClass(ClassLoader.java:616)
... 8 more
```



方法区溢出也是一种常见的内存溢出异常，一个类如果要被垃圾收集器回收掉，判定条件是非常苛刻的。在经常动态生成大量 Class 的应用中，需要特别注意类的回收状况。这类场景除了上面提到的程序使用了 GCLib 字节码增强外，常见的还有：大量 JSP 或动态产生 JSP 文件的应用（JSP 第一次运行时需要编译为 Java 类）、基于 OSGi 的应用（即使是同一个类文件，被不同的加载器加载也会视为不同的类）等。

2.4.5 本机直接内存溢出

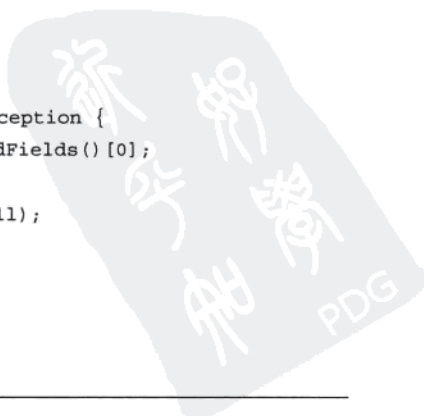
DirectMemory 容量可通过 `-XX:MaxDirectMemorySize` 指定，如果不指定，则默认与 Java 堆的最大值（`-Xmx` 指定）一样。代码清单 2-6 越过了 `DirectByteBuffer` 类，直接通过反射获取 `Unsafe` 实例并进行内存分配（`Unsafe` 类的 `getUnsafe()` 方法限制了只有引导类加载器才会返回实例，也就是设计者希望只有 `rt.jar` 中的类才能使用 `Unsafe` 的功能）。因为，虽然使用 `DirectByteBuffer` 分配内存也会抛出内存溢出异常，但它抛出异常时并没有真正向操作系统申请分配内存，而是通过计算得知内存无法分配，于是手动抛出异常，真正申请分配内存的方法是 `unsafe.allocateMemory()`。

代码清单 2-6 使用 unsafe 分配本机内存

```
/**
 * VM Args: -Xmx20M -XX:MaxDirectMemorySize=10M
 * @author zzm
 */
public class DirectMemoryOOM {

    private static final int _1MB = 1024 * 1024;

    public static void main(String[] args) throws Exception {
        Field unsafeField = Unsafe.class.getDeclaredFields()[0];
        unsafeField.setAccessible(true);
        Unsafe unsafe = (Unsafe) unsafeField.get(null);
        while (true) {
            unsafe.allocateMemory(_1MB);
        }
    }
}
```



42 ❖ 第二部分 自动内存管理机制

运行结果：

```
Exception in thread "main" java.lang.OutOfMemoryError
    at sun.misc.Unsafe.allocateMemory(Native Method)
    at org.fenixsoft.oom.DMOOM.main(DMOOM.java:20)
```

2.5 本章小结

通过本章的学习，我们明白了虚拟机里面的内存是如何划分的，哪部分区域、什么样的代码和操作可能导致内存溢出异常。虽然 Java 有垃圾收集机制，但内存溢出异常离我们并不遥远，本章只是讲解了各个区域出现内存溢出异常的原因，下一章将详细讲解 Java 垃圾收集机制为了避免内存溢出异常的出现都做了哪些努力。



第3章 垃圾收集器与内存分配策略

本章主要内容

- 概述
- 对象已死?
- 垃圾收集算法
- 垃圾收集器
- 内存分配与回收策略

Java 与 C++ 之间有一堵由内存动态分配和垃圾收集技术所围成的高墙，墙外面的人想进去，墙里面的人却想出来。

3.1 概述

说起垃圾收集 (Garbage Collection, GC)，大部分人都把这项技术当做 Java 语言的伴生产物。事实上，GC 的历史远远比 Java 久远，1960 年诞生于 MIT 的 Lisp 是第一门真正使用内存动态分配和垃圾收集技术的语言。当 Lisp 还在胚胎时期时，人们就在思考 GC 需要完成的三件事情：

- 哪些内存需要回收?
- 什么时候回收?
- 如何回收?

经过半个世纪的发展，内存的动态分配与内存回收技术已经相当成熟，一切看起来都进入了“自动化”时代，那为什么我们还要去了解 GC 和内存分配呢？答案很简单：当需要排查各种内存溢出、内存泄漏问题时，当垃圾收集成为系统达到更高并发量的瓶颈时，我们就需要对这些“自动化”的技术实施必要的监控和调节。

把时间从半个世纪以前拨回到现在，回到我们熟悉的 Java 语言。第 2 章介绍了 Java 内存运行时区域的各个部分，其中程序计数器、虚拟机栈、本地方法栈三个区域随线程而生，随线程而灭；栈中的栈帧随着方法的进入和退出而有条不紊地执行着出栈和入栈操作。每一个栈帧中分配多少内存基本上是在类结构确定下来时就已知的（尽管在运行期会由 JIT 编译器进行一些优化，但在本章基于概念模型的讨论中，大体上可以认为是编译期可知的），因此这几个区域的内存分配和回收都具备确定性，在这几个区域内不需要过多考虑回收的问题，因为方法结束或线程结束时，内存自然就跟随着回收了。而 Java 堆和方法区则不一样，一个接口中的多个实现类需要的内存可能不一样，一个方法中的多个分支需要的内存也可能不一样，我们只有在程序处于运行期间时才能知道会创建哪些对象，这部分内存的分配和回收都是动态的，垃圾收集器所关注的是这部分内存，本书后续讨论中的“内存”分配与回收也仅指这一部分内存。

3.2 对象已死？

堆中几乎存放着 Java 世界中所有的对象实例，垃圾收集器在对堆进行回收前，第一件事情就是要确定这些对象有哪些还“存活”着，哪些已经“死去”（即不可能再被任何途径使用的对象）。

3.2.1 引用计数算法

很多教科书判断对象是否存活的算法是这样的：给对象中添加一个引用计数器，每当有一个地方引用它时，计数器值就加 1；当引用失效时，计数器值就减 1；任何时刻计数器都为 0 的对象就是不可能再被使用的。笔者面试过很多的应届生和一些有多年工作经验的开发人员，他们对于这个问题给予的都是这个答案。

客观地说，引用计数算法（Reference Counting）的实现简单，判定效率也很高，在大部分情况下它都是一个不错的算法，也有一些比较著名的应用案例，例如微软的 COM（Component Object Model）技术、使用 ActionScript 3 的 FlashPlayer、Python 语言以及在游戏脚本领域中被广泛应用的 Squirrel 中都使用了引用计数算法进行内存管理。但是，Java 语言中没有选用引用计数算法来管理内存，其中最主要的原因是它很难解决对象之间的相互循环引用的问题。

举个简单的例子，请看代码清单 3-1 中的 testGC() 方法：对象 objA 和 objB 都有字

段 instance, 赋值令 objA.instance = objB 及 objB.instance = objA, 除此之外, 这两个对象再无任何引用, 实际上这两个对象已经不可能再被访问, 但是它们因为互相引用着对方, 导致它们的引用计数都不为 0, 于是引用计数算法无法通知 GC 收集器回收它们。

代码清单 3-1 引用计数算法的缺陷

```
/**
 * testGC() 方法执行后, objA 和 objB 会不会被 GC 呢?
 * @author zzm
 */
public class ReferenceCountingGC {

    public Object instance = null;

    private static final int _1MB = 1024 * 1024;

    /**
     * 这个成员属性的唯一意义就是占点内存, 以便能在 GC 日志中看清楚是否被回收过
     */
    private byte[] bigSize = new byte[2 * _1MB];

    public static void testGC() {
        ReferenceCountingGC objA = new ReferenceCountingGC();
        ReferenceCountingGC objB = new ReferenceCountingGC();
        objA.instance = objB;
        objB.instance = objA;

        objA = null;
        objB = null;

        // 假设在这行发生 GC, 那么 objA 和 objB 是否能被回收?
        System.gc();
    }
}
```

运行结果:

```
[Full GC (System) [Tenured: 0K->210K(10240K), 0.0149142 secs] 4603K->210K(19456K),
[Perm : 2999K->2999K(21248K)], 0.0150007 secs] [Times: user=0.01 sys=0.00, real=0.02
secs]
```

Heap

```
def new generation      total 9216K, used 82K [0x000000000055e0000,
```

46 ❖ 第二部分 自动内存管理机制

```

0x0000000005fe0000, 0x0000000005fe0000)
    Eden space 8192K,    1% used [0x00000000055e0000, 0x00000000055f4850,
0x0000000005de0000)
    from space 1024K,   0% used [0x0000000005de0000, 0x0000000005de0000,
0x0000000005ee0000)
    to   space 1024K,   0% used [0x0000000005ee0000, 0x0000000005ee0000,
0x0000000005fe0000)
    tenured generation    total 10240K, used 210K [0x0000000005fe0000,
0x00000000069e0000, 0x00000000069e0000)
    the space 10240K,     2% used [0x0000000005fe0000, 0x0000000006014a18,
0x0000000006014c00, 0x00000000069e0000)
    compacting perm gen   total 21248K, used 3016K [0x00000000069e0000,
0x0000000007ea0000, 0x000000000bde0000)
    the space 21248K,    14% used [0x00000000069e0000, 0x0000000006cd2398,
0x0000000006cd2400, 0x0000000007ea0000)
    No shared spaces configured.

```

从运行结果中可以清楚地看到 GC 日志中包含“4603K->210K”，意味着虚拟机并没有因为这两个对象互相引用就不回收它们，这也从侧面说明虚拟机并不是通过引用计数算法来判断对象是否存活的。

3.2.2 根搜索算法

在主流的商用程序语言中（Java 和 C#，甚至包括前面提到的古老的 Lisp），都是使用根搜索算法（GC Roots Tracing）判定对象是否存活的。这个算法的基本思路就是通过一系列的名为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链（Reference Chain），当一个对象到 GC Roots 没有任何引用链相连（用图论的话来说就是从 GC Roots 到这个对象不可达）时，则证明此对象是不可用的。如图 3-1 所示，对象 object 5、object 6、object 7 虽然互相有关联，但是它们到 GC Roots 是不可达的，所以它们将会被判定为是可回收的对象。

在 Java 语言里，可作为 GC Roots 的对象包括下面几种：

- 虚拟机栈（栈帧中的本地变量表）中的引用的对象。
- 方法区中的类静态属性引用的对象。
- 方法区中的常量引用的对象。
- 本地方法栈中 JNI（即一般说的 Native 方法）的引用的对象。

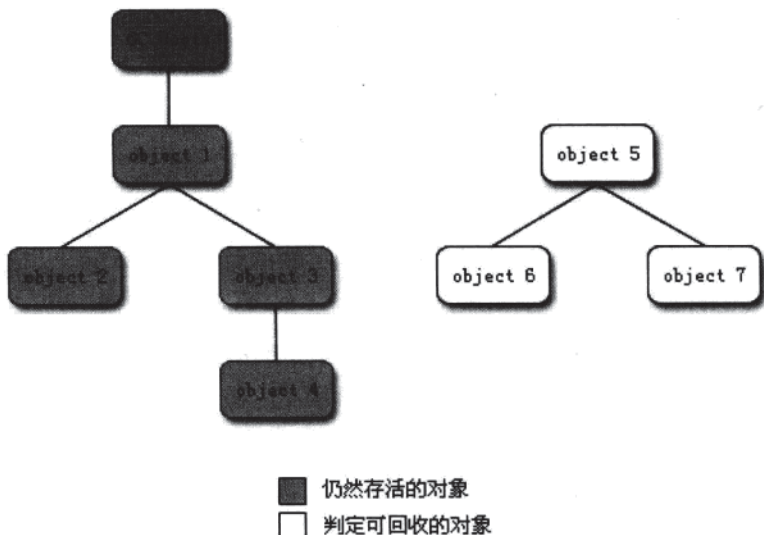


图 3-1 根搜索算法判定对象是否可回收

3.2.3 再谈引用

无论是通过引用计数算法判断对象的引用数量，还是通过根搜索算法判断对象的引用链是否可达，判定对象是否存活都与“引用”有关。在 JDK 1.2 之前，Java 中的引用的定义很传统：如果 reference 类型的数据中存储的数值代表的是另外一块内存的起始地址，就称这块内存代表着一个引用。这种定义很纯粹，但是太过狭隘，一个对象在这种定义下只有被引用或者没有被引用两种状态，对于如何描述一些“食之无味，弃之可惜”的对象就显得无能为力。我们希望能描述这样一类对象：当内存空间还足够时，则能保留在内存之中；如果内存存在进行垃圾收集后还是非常紧张，则可以抛弃这些对象。很多系统的缓存功能都符合这样的应用场景。

在 JDK 1.2 之后，Java 对引用的概念进行了扩充，将引用分为强引用（Strong Reference）、软引用（Soft Reference）、弱引用（Weak Reference）、虚引用（Phantom Reference）四种，这四种引用强度依次逐渐减弱。

- 强引用就是指在程序代码之中普遍存在的，类似“Object obj = new Object()”这类的引用，只要强引用还存在，垃圾收集器永远不会回收掉被引用的对象。
- 软引用用来描述一些还有用，但并非必需的对象。对于软引用关联着的对象，在

系统将要发生内存溢出异常之前，将会把这些对象列进回收范围之中并进行第二次回收。如果这次回收还是没有足够的内存，才会抛出内存溢出异常。在 JDK 1.2 之后，提供了 `SoftReference` 类来实现软引用。

- 弱引用也是用来描述非必需对象的，但是它的强度比软引用更弱一些，被弱引用关联的对象只能生存到下一次垃圾收集发生之前。当垃圾收集器工作时，无论当前内存是否足够，都会回收掉只被弱引用关联的对象。在 JDK 1.2 之后，提供了 `WeakReference` 类来实现弱引用。
- 虚引用也称为幽灵引用或者幻影引用，它是最弱的一种引用关系。一个对象是否有虚引用的存在，完全不会对其生存时间构成影响，也无法通过虚引用来取得一个对象实例。为一个对象设置虚引用关联的唯一目的就是希望能在该对象被垃圾收集器回收时收到一个系统通知。在 JDK 1.2 之后，提供了 `PhantomReference` 类来实现虚引用。

3.2.4 生存还是死亡？

在根搜索算法中不可达的对象，也并非是非“非死不可”的，这时候它们暂时处于“缓刑”阶段，要真正宣告一个对象死亡，至少要经历两次标记过程：如果对象在进行根搜索后发现没有与 GC Roots 相连接的引用链，那它将会被第一次标记并且进行一次筛选，筛选的条件是此对象是否有必要执行 `finalize()` 方法。当对象没有覆盖 `finalize()` 方法，或者 `finalize()` 方法已经被虚拟机调用过，虚拟机将这两种情况都视为“没有必要执行”。

如果这个对象被判定为有必要执行 `finalize()` 方法，那么这个对象将会被放置在一个名为 F-Queue 的队列之中，并在稍后由一条由虚拟机自动建立的、低优先级的 Finalizer 线程去执行。这里所谓的“执行”是指虚拟机会触发这个方法，但并不承诺会等待它运行结束。这样做的原因是，如果一个对象在 `finalize()` 方法中执行缓慢，或者发生了死循环（更极端的情况），将很可能导致 F-Queue 队列中的其他对象永久处于等待状态，甚至导致整个内存回收系统崩溃。`finalize()` 方法是对象逃脱死亡命运的最后一次机会，稍后 GC 将对 F-Queue 中的对象进行第二次小规模标记，如果对象要在 `finalize()` 中成功拯救自己——只要重新与引用链上的任何一个对象建立关联即可，譬如把自己（`this` 关键字）赋值给某个类变量或对象的成员变量，那在第二次标记时它将被移除出“即将

回收”的集合；如果对象这时候还没有逃脱，那它就真的离死不远了。从代码清单 3-2 中我们可以看到一个对象的 `finalize()` 被执行，但是它仍然可以存活。

代码清单 3-2 一次对象自我拯救的演示

```
/**
 * 此代码演示了两点：
 * 1. 对象可以在被 GC 时自我拯救。
 * 2. 这种自救的机会只有一次，因为一个对象的 finalize() 方法最多只会被系统自动调用一次
 * @author zzm
 */
public class FinalizeEscapeGC {

    public static FinalizeEscapeGC SAVE_HOOK = null;

    public void isAlive() {
        System.out.println("yes, i am still alive :)");
    }

    @Override
    protected void finalize() throws Throwable {
        super.finalize();
        System.out.println("finalize mehtod executed!");
        FinalizeEscapeGC.SAVE_HOOK = this;
    }

    public static void main(String[] args) throws Throwable {
        SAVE_HOOK = new FinalizeEscapeGC();

        // 对象第一次成功拯救自己
        SAVE_HOOK = null;
        System.gc();
        // 因为 Finalizer 方法优先级很低，暂停 0.5 秒，以等待它
        Thread.sleep(500);
        if (SAVE_HOOK != null) {
            SAVE_HOOK.isAlive();
        } else {
            System.out.println("no, i am dead :(");
        }

        // 下面这段代码与上面的完全相同，但是这次自救却失败了
        SAVE_HOOK = null;
    }
}
```



50 ❖ 第二部分 自动内存管理机制

```
System.gc();
// 因为 Finalizer 方法优先级很低, 暂停 0.5 秒, 以等待它
Thread.sleep(500);
if (SAVE_HOOK != null) {
    SAVE_HOOK.isAlive();
} else {
    System.out.println("no, i am dead :(");
}
}
}
```

运行结果：

```
finalize mehtod executed!
yes, i am still alive :)
no, i am dead :(
```

从代码清单 3-2 的运行结果可以看到，SAVE_HOOK 对象的 finalize() 方法确实被 GC 收集器触发过，并且在被收集前成功逃脱了。

另外一个值得注意的地方就是，代码中有两段完全一样的代码片段，执行结果却是一次逃脱成功，一次失败，这是因为任何一个对象的 finalize() 方法都只会被系统自动调用一次，如果对象面临下一次回收，它的 finalize() 方法不会被再次执行，因此第二段代码的自救行动失败了。

需要特别说明的是，上面关于对象死亡时 finalize() 方法的描述可能带有悲情的艺术色彩，笔者并不鼓励大家使用这种方法来拯救对象。相反，笔者建议大家尽量避免使用它，因为它不是 C/C++ 中的析构函数，而是 Java 刚诞生时为了使 C/C++ 程序员更容易接受它所做出的一个妥协。它的运行代价高昂，不确定性大，无法保证各个对象的调用顺序。有些教材中提到它适合做“关闭外部资源”之类的工作，这完全是对这种方法的用途的一种自我安慰。finalize() 能做的所有工作，使用 try-finally 或其他方式都可以做得更好、更及时，大家完全可以忘掉 Java 语言中还有这个方法的存在。

3.2.5 回收方法区

很多人认为方法区（或者 HotSpot 虚拟机中的永久代）是没有垃圾收集的，Java 虚拟机规范中确实说过可以不要虚拟机在方法区实现垃圾收集，而且在方法区进行垃圾收集的“性价比”一般比较低：在堆中，尤其是在新生代中，常规应用进行一次垃圾收

集一般可以回收 70% ~ 95% 的空间，而永久代的垃圾收集效率远低于此。

永久代的垃圾收集主要回收两部分内容：废弃常量和无用的类。回收废弃常量与回收 Java 堆中的对象非常类似。以常量池中字面量的回收为例，假如一个字符串“abc”已经进入了常量池中，但是当前系统没有任何一个 String 对象是叫做“abc”的，换句话说没有任何 String 对象引用常量池中的“abc”常量，也没有其他地方引用了这个字面量，如果在这时候发生内存回收，而且必要的话，这个“abc”常量就会被系统“请”出常量池。常量池中的其他类（接口）、方法、字段的符号引用也与此类似。

判定一个常量是否是“废弃常量”比较简单，而要判定一个类是否是“无用的类”的条件则相对苛刻许多。类需要同时满足下面 3 个条件才能算是“无用的类”：

- 该类所有的实例都已经被回收，也就是 Java 堆中不存在该类的任何实例。
- 加载该类的 ClassLoader 已经被回收。
- 该类对应的 java.lang.Class 对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。

虚拟机可以对满足上述 3 个条件的无用类进行回收，这里说的仅仅是“可以”，而不是和对象一样，不使用了就必然会回收。是否对类进行回收，HotSpot 虚拟机提供了 `-Xnoclassgc` 参数进行控制，还可以使用 `-verbose:class` 及 `-XX:+TraceClassLoading`、`-XX:+TraceClassUnLoading` 查看类的加载和卸载信息。`-verbose: class` 和 `-XX:+TraceClassLoading` 可以在 Product 版的虚拟机中使用，但是 `-XX:+TraceClassLoading` 参数需要 fastdebug 版的虚拟机支持。

在大量使用反射、动态代理、CGLib 等 bytecode 框架的场景，以及动态生成 JSP 和 OSGi 这类频繁自定义 ClassLoader 的场景都需要虚拟机具备类卸载的功能，以保证永久代不会溢出。

3.3 垃圾收集算法

由于垃圾收集算法的实现涉及大量的程序细节，而且各个平台的虚拟机操作内存的方法又各不相同，因此本节不打算过多地讨论算法的实现，只是介绍几种算法的思想及其发展过程。

3.3.1 标记 - 清除算法

最基础的收集算法是“标记 - 清除”（Mark-Sweep）算法，如它的名字一样，算法分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，在标记完成后统一回收掉所有被标记的对象，它的标记过程其实在前一节讲述对象标记判定时已经基本介绍了。之所以说它是最基础的收集算法，是因为后续的收集算法都是基于这种思路并对其缺点进行改进而得到的。它的主要缺点有两个：一个是效率问题，标记和清除过程的效率都不高；另外一个空间问题，标记清除之后会产生大量不连续的内存碎片，空间碎片太多可能会导致，当程序在以后的运行过程中需要分配较大对象时无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作。标记 - 清除算法的执行过程如图 3-2 所示。

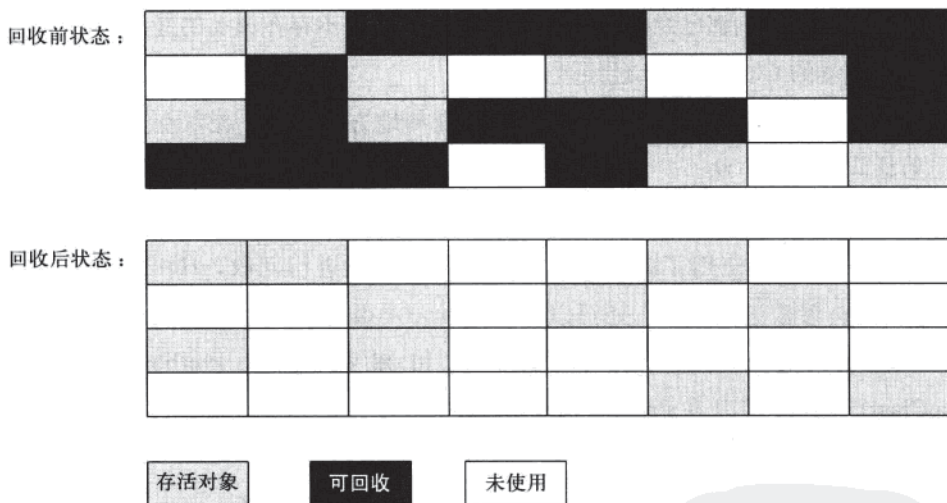


图 3-2 “标记 - 清除”算法示意图

3.3.2 复制算法

为了解决效率问题，一种称为“复制”（Copying）的收集算法出现了，它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。这样使得每次都是对其中的一块进行内存回收，内存分配时就不用考虑内存碎片等复杂情况，只要移动堆顶指针，按顺序分配内存即可，实现简单，运行高效。只是这种算法的代价是将

内存缩小为原来的一半，未免太高了一点。复制算法的执行过程如图 3-3 所示。

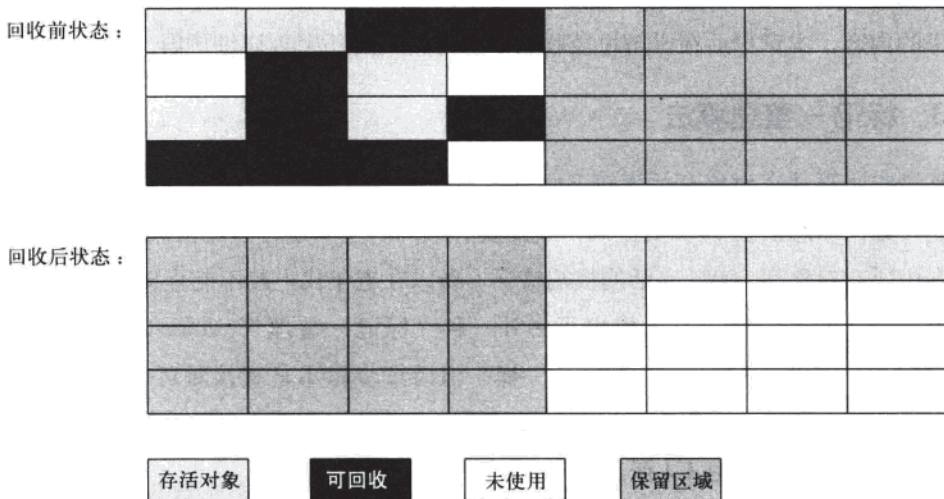


图 3-3 复制算法示意图

现在的商业虚拟机都采用这种收集算法来回收新生代，IBM 的专门研究表明，新生代中的对象 98% 是朝生夕死的，所以并不需要按照 1:1 的比例来划分内存空间，而是将内存分为一块较大的 Eden 空间和两块较小的 Survivor 空间，每次使用 Eden 和其中的一块 Survivor^①。当回收时，将 Eden 和 Survivor 中还存活着的对象一次性地拷贝到另外一块 Survivor 空间上，最后清理掉 Eden 和刚才用过的 Survivor 的空间。HotSpot 虚拟机默认 Eden 和 Survivor 的大小比例是 8:1，也就是每次新生代中可用内存空间为整个新生代容量的 90% (80%+10%)，只有 10% 的内存是会被“浪费”的。当然，98% 的对象可回收只是一般场景下的数据，我们没有办法保证每次回收都只有不多于 10% 的对象存活，当 Survivor 空间不够用时，需要依赖其他内存（这里指老年代）进行分配担保 (Handle Promotion)。

内存的分配担保就好比我们去银行借款，如果我们信誉很好，在 98% 的情况下都能按时偿还，于是银行可能会默认我们下一次也能按时按量地偿还贷款，只需要有一个担保人能保证如果不能还款时，可以从他的账户扣钱，那银行就认为没有风险了。内存

^① 这里需要说明一下，在 HotSpot 的分代收集，新生代最初就是这种布局，与“IBM 的研究”并没有实际联系。本书列举 IBM 的研究只是为了说明这种分代布局的意义所在。

54 ❖ 第二部分 自动内存管理机制

的分配担保也一样，如果另外一块 Survivor 空间没有足够的空间存放上一次新生代收集下来的存活对象，这些对象将直接通过分配担保机制进入老年代。关于对新生代进行分配担保的内容，本章稍后在讲解垃圾收集器执行规则时还会再详细讲解。

3.3.3 标记 - 整理算法

复制收集算法在对象存活率较高时就要执行较多的复制操作，效率将会变低。更关键的是，如果不想浪费 50% 的空间，就需要有额外的空间进行分配担保，以应对被使用的内存中所有对象都 100% 存活的极端情况，所以在老年代一般不能直接选用这种算法。

根据老年代的特点，有人提出了另外一种“标记 - 整理”（Mark-Compact）算法，标记过程仍然与“标记 - 清除”算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存，“标记 - 整理”算法的示意图如图 3-4 所示。

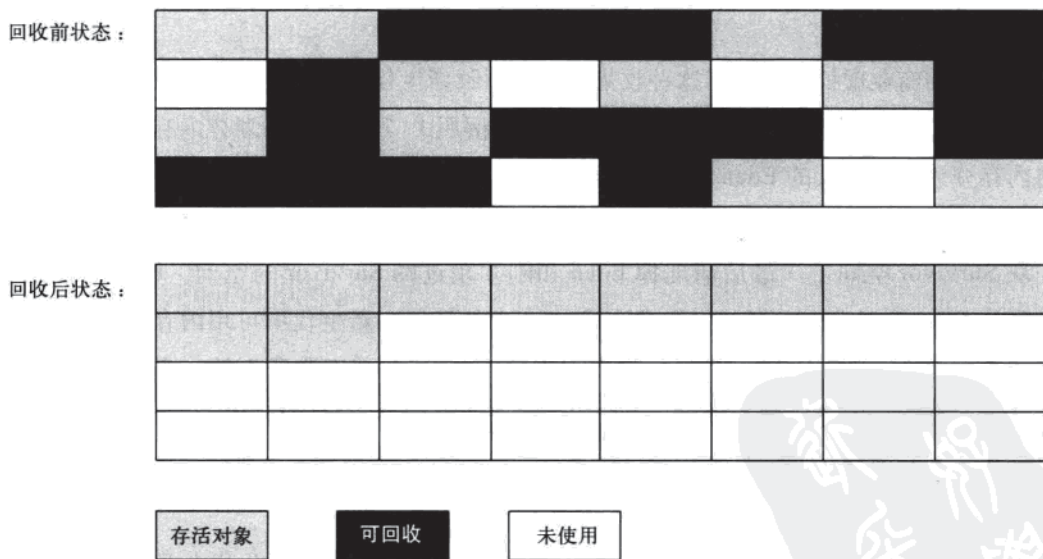


图 3-4 “标记 - 整理”算法示意图

3.3.4 分代收集算法

当前商业虚拟机的垃圾收集都采用“分代收集”（Generational Collection）算法，这种算法并没有什么新的思想，只是根据对象的存活周期的不同将内存划分为几块。一般

是把 Java 堆分为新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法。在新生代中，每次垃圾收集时都发现有大批对象死去，只有少量存活，那就选用复制算法，只需要付出少量存活对象的复制成本就可以完成收集。而老年代中因为对象存活率高、没有额外空间对它进行分配担保，就必须使用“标记-清理”或“标记-整理”算法来进行回收。

3.4 垃圾收集器

如果说收集算法是内存回收的方法论，垃圾收集器就是内存回收的具体实现。Java 虚拟机规范中对垃圾收集器应该如何实现并没有任何规定，因此不同的厂商、不同版本的虚拟机所提供的垃圾收集器都可能会有很大的差别，并且一般都会提供参数供用户根据自己的应用特点和要求组合出各个年代所使用的收集器。这里讨论的收集器基于 Sun HotSpot 虚拟机 1.6 版 Update 22，这个虚拟机包含的所有收集器如图 3-5 所示。

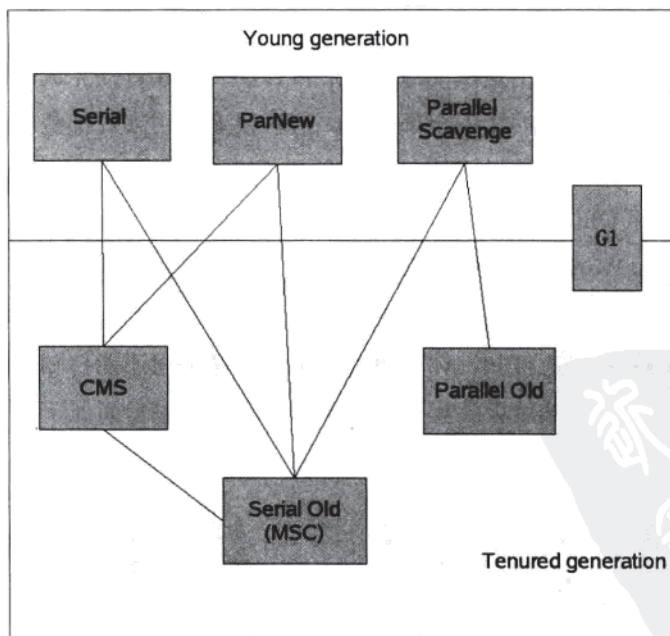


图 3-5 HotSpot JVM1.6 的垃圾收集器^①

^① 图片来源：http://blogs.sun.com/jonthecollector/entry/our_collectors。

图 3-5 展示了 7 种作用于不同分代的收集器（包括 JDK 1.6_Update14 后引入的 Early Access 版 G1 收集器），如果两个收集器之间存在连线，就说明它们可以搭配使用。

在介绍这些收集器各自的特性之前，我们先来明确一个观点：虽然我们是在对各个收集器进行比较，但并非为了挑选一个最好的收集器出来。因为直到现在为止还没有最好的收集器出现，更加没有万能的收集器，所以我们选择的只是对具体应用最合适的收集器。这点不需要多加解释就能证明：如果有一种放之四海皆准、任何场景下都适用的完美收集器存在，那 HotSpot 虚拟机就没必要实现那么多不同的收集器了。

3.4.1 Serial 收集器

Serial 收集器是最基本、历史最悠久的收集器，曾经（在 JDK 1.3.1 之前）是虚拟机新生代收集的唯一选择。大家看名字就知道，这个收集器是一个单线程的收集器，但它的“单线程”的意义并不仅仅是说明它只会使用一个 CPU 或一条收集线程去完成垃圾收集工作，更重要的是在它进行垃圾收集时，必须暂停其他所有的工作线程（Sun 将这件事情称之为“Stop The World”），直到它收集结束。“Stop The World”这个名字也许听起来很酷，但这项工作实际上是由虚拟机在后台自动发起和自动完成的，在用户不可见的情况下把用户的正常工作的线程全部停掉，这对很多应用来说都是难以接受的。你想想，要是你的电脑每运行一个小时就会暂停响应 5 分钟，你会有什么样的心情？图 3-6 示意了 Serial / Serial Old 收集器的运行过程。

对于“Stop The World”带给用户的恶劣体验，虚拟机的设计者们表示完全理解，但也表示非常委屈：“你妈妈在给你打扫房间的时候，肯定也会让你老老实实在椅子上或房间外待着，如果她一边打扫，你一边乱扔纸屑，这房间还能打扫完吗？”这确实是一个合情合理的矛盾，虽然垃圾收集这项工作听起来和打扫房间属于一个性质的，但实际上肯定还要比打扫房间复杂得多啊！

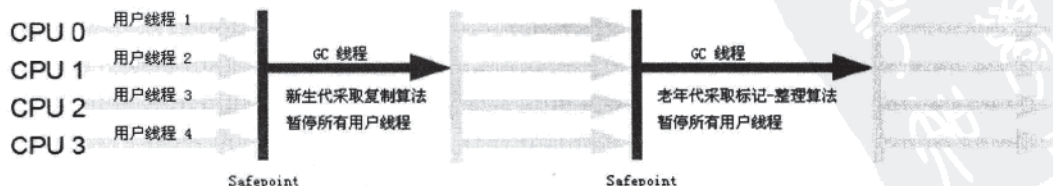


图 3-6 Serial / Serial Old 收集器运行示意图

从 JDK 1.3 开始，一直到现在还没正式发布的 JDK 1.7，HotSpot 虚拟机开发团队为消除或减少工作线程因内存回收而导致停顿的努力一直在进行着，从 Serial 收集器到 Parallel 收集器，再到 Concurrent Mark Sweep (CMS) 现在还未正式发布的 Garbage First (G1) 收集器，我们看到了一个个越来越优秀（也越来越复杂）的收集器的出现，用户线程的停顿时间在不断缩短，但是仍然没有办法完全消除（这里暂不包括 RTSJ 中的收集器）。寻找更优秀的垃圾收集器的工作仍在继续！

写到这里，笔者似乎已经把 Serial 收集器描述成一个老而无用，食之无味弃之可惜的鸡肋了，但实际上到现在为止，它依然是虚拟机运行在 Client 模式下的默认新生代收集器。它也有着优于其他收集器的地方：简单而高效（与其他收集器的单线程比），对于限定单个 CPU 的环境来说，Serial 收集器由于没有线程交互的开销，专心做垃圾收集自然可以获得最高的单线程收集效率。在用户的桌面应用场景中，分配给虚拟机管理的内存一般来说不会很大，收集几十兆甚至一两百兆的新生代（仅仅是新生代使用的内存，桌面应用基本上不会再大了），停顿时间完全可以控制在几十毫秒最多一百多毫秒以内，只要不是频繁发生，这点停顿是可以接受的。所以，Serial 收集器对于运行在 Client 模式下的虚拟机来说是一个很好的选择。

3.4.2 ParNew 收集器

ParNew 收集器其实就是 Serial 收集器的多线程版本，除了使用多条线程进行垃圾收集之外，其余行为包括 Serial 收集器可用的所有控制参数（例如：`-XX:SurvivorRatio`、`-XX:PretenureSizeThreshold`、`-XX:HandlePromotionFailure` 等）、收集算法、Stop The World、对象分配规则、回收策略等都与 Serial 收集器完全一样，实现上这两种收集器也共用了相当多的代码。ParNew 收集器的工作过程如图 3-7 所示。

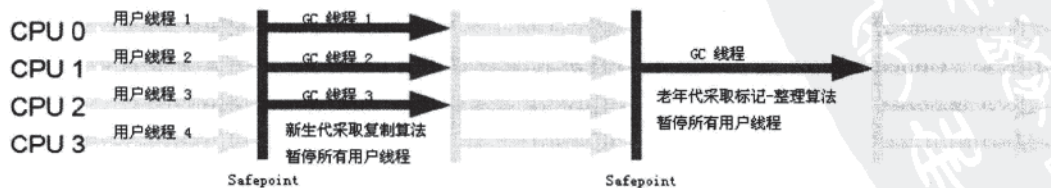


图 3-7 ParNew / Serial Old 收集器运行示意图

ParNew 收集器除了多线程收集之外，其他与 Serial 收集器相比并没有太多创新之处，但它却是许多运行在 Server 模式下的虚拟机中首选的新生代收集器，其中有一个与性能无关但很重要的原因是，除了 Serial 收集器外，目前只有它能与 CMS 收集器配合工作。在 JDK 1.5 时期，HotSpot 推出了一款在强交互应用中几乎可称为有划时代意义的垃圾收集器——CMS 收集器（Concurrent Mark Sweep，本节稍后将详细介绍这款收集器），这款收集器是 HotSpot 虚拟机中第一款真正意义上的并发（Concurrent）收集器，它第一次实现了让垃圾收集线程与用户线程（基本上）同时工作，用前面那个例子的话来说，就是做到了在你妈妈打扫房间的时候你还能同时往地上扔纸屑。

不幸的是，它作为老年代的收集器，却无法与 JDK 1.4.0 中已经存在的新生代收集器 Parallel Scavenge 配合工作^①，所以在 JDK 1.5 中使用 CMS 来收集老年代的时候，新生代只能选择 ParNew 或 Serial 收集器中的一个。ParNew 收集器也是使用 `-XX:+UseConcMarkSweepGC` 选项后的默认新生代收集器，也可以使用 `-XX:+UseParNewGC` 选项来强制指定它。

ParNew 收集器在单 CPU 的环境中绝对不会有比 Serial 收集器更好的效果，甚至由于存在线程交互的开销，该收集器在通过超线程技术实现的两个 CPU 的环境中都不能百分之百地保证能超越 Serial 收集器。当然，随着可以使用的 CPU 的数量的增加，它对于 GC 时系统资源的利用还是很有好处的。它默认开启的收集线程数与 CPU 的数量相同，在 CPU 非常多（譬如 32 个，现在 CPU 动辄就 4 核加超线程，服务器超过 32 个逻辑 CPU 的情况越来越多了）的环境下，可以使用 `-XX:ParallelGCThreads` 参数来限制垃圾收集的线程数。

注意 从 ParNew 收集器开始，后面还将会接触到几款并发和并行的收集器。在大家可能产生疑惑之前，有必要先解释两个名词：并发和并行。这两个名词都是并发编程中的概念，在谈论垃圾收集器的上下文语境中，他们可以解释为：

- ❑ 并行（Parallel）：指多条垃圾收集线程并行工作，但此时用户线程仍然处于等待状态。
 - ❑ 并发（Concurrent）：指用户线程与垃圾收集线程同时执行（但不一定是并行的，可能会交替执行），用户程序继续运行，而垃圾收集程序运行于另一个 CPU 上。
-

^① 原因 Parallel Scavenge 收集器及后面提到的 G1 收集器都没有使用传统的 GC 收集器代码框架，而另外独立实现，其余集中收集器则共用了部分的框架代码，详细可参考：http://blogs.sun.com/jonthecollector/entry/our_collectors。

3.4.3 Parallel Scavenge 收集器

Parallel Scavenge 收集器也是一个新生代收集器，它也是使用复制算法的收集器，又是并行的多线程收集器……看上去和 ParNew 都一样，那它有什么特别之处呢？

Parallel Scavenge 收集器的特点是它的关注点与其他收集器不同，CMS 等收集器的关注点尽可能地缩短垃圾收集时用户线程的停顿时间，而 Parallel Scavenge 收集器的目标则是达到一个可控制的吞吐量（Throughput）。所谓吞吐量就是 CPU 用于运行用户代码的时间与 CPU 总消耗时间的比值，即 $\text{吞吐量} = \text{运行用户代码时间} / (\text{运行用户代码时间} + \text{垃圾收集时间})$ ，虚拟机总共运行了 100 分钟，其中垃圾收集花掉 1 分钟，那吞吐量就是 99%。

停顿时间越短就越适合需要与用户交互的程序，良好的响应速度能提升用户的体验；而高吞吐量则可以最高效率地利用 CPU 时间，尽快地完成程序的运算任务，主要适合在后台运算而不需要太多交互的任务。

Parallel Scavenge 收集器提供了两个参数用于精确控制吞吐量，分别是控制最大垃圾收集停顿时间的 `-XX:MaxGCPauseMillis` 参数及直接设置吞吐量大小的 `-XX:GCTimeRatio` 参数。

`MaxGCPauseMillis` 参数允许的值是一个大于 0 的毫秒数，收集器将尽力保证内存回收花费的时间不超过设定值。不过大家不要异想天开地认为如果把这个参数的值设置得稍小一点就能使得系统的垃圾收集速度变得更快，GC 停顿时间缩短是以牺牲吞吐量和新生代空间来换取的：系统把新生代调小一些，收集 300MB 新生代肯定比收集 500MB 快吧，这也直接导致垃圾收集发生得更频繁一些，原来 10 秒收集一次、每次停顿 100 毫秒，现在变成 5 秒收集一次、每次停顿 70 毫秒。停顿时间的确在下降，但吞吐量也降下来了。

`GCTimeRatio` 参数的值应当是一个大于 0 小于 100 的整数，也就是垃圾收集时间占总时间的比率，相当于是吞吐量的倒数。如果把此参数设置为 19，那允许的最大 GC 时间就占总时间的 5%（即 $1 / (1+19)$ ），默认值为 99，就是允许最大 1%（即 $1 / (1+99)$ ）的垃圾收集时间。

由于与吞吐量关系密切，Parallel Scavenge 收集器也经常被称为“吞吐量优先”收集器。除上述两个参数之外，Parallel Scavenge 收集器还有一个参

数 `-XX:+UseAdaptiveSizePolicy` 值得关注。这是一个开关参数，当这个参数打开之后，就不需要手工指定新生代的大小（`-Xmn`）、Eden 与 Survivor 区的比例（`-XX:SurvivorRatio`）、晋升老年代对象年龄（`-XX:PretenureSizeThreshold`）等细节参数了，虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最合适的停顿时间或最大的吞吐量，这种调节方式称为 GC 自适应的调节策略（GC Ergonomics）^①。如果读者对于收集器运作原理不太了解，手工优化存在困难的时候，使用 Parallel Scavenge 收集器配合自适应调节策略，把内存管理的调优任务交给虚拟机去完成将是一个不错的选择。只需要把基本的内存数据设置好（如 `-Xmx` 设置最大堆），然后使用 `MaxGCPauseMillis` 参数（更关注最大停顿时间）或 `GCTimeRatio` 参数（更关注吞吐量）给虚拟机设立一个优化目标，那具体细节参数的调节工作就由虚拟机完成了。自适应调节策略也是 Parallel Scavenge 收集器与 ParNew 收集器的一个重要区别。

3.4.4 Serial Old 收集器

Serial Old 是 Serial 收集器的老年代版本，它同样是一个单线程收集器，使用“标记-整理”算法。这个收集器的主要意义也是被 Client 模式下的虚拟机使用。如果在 Server 模式下，它主要还有两大用途：一个是在 JDK 1.5 及之前的版本中与 Parallel Scavenge 收集器搭配使用^②，另外一个就是作为 CMS 收集器的后备预案，在并发收集发生 Concurrent Mode Failure 的时候使用。这两点都将在后面的内容中详细讲解。Serial Old 收集器的工作过程如图 3-8 所示。

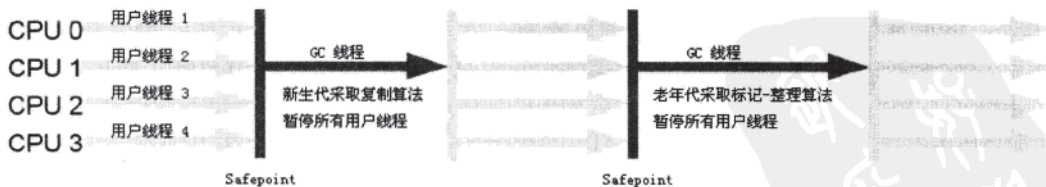


图 3-8 Serial / Serial Old 收集器运行示意图

- ① 官方介绍：<http://download.oracle.com/javase/1.5.0/docs/guide/vm/gc-ergonomics.html>。
- ② 需要说明一下，Parallel Scavenge 收集器架构中本身有 PS MarkSweep 收集器来进行老年代收集，并没有直接使用 Serial Old 收集器，但是这个 PS MarkSweep 收集器是以 Serial Old 收集器为模板设计的，与 Serial Old 的实现非常接近，所以在官方的许多资料中都是直接用 Serial Old 替代 PS MarkSweep 进行讲解，本书也采用此说法。

3.4.5 Parallel Old 收集器

Parallel Old 是 Parallel Scavenge 收集器的老年代版本，使用多线程和“标记-整理”算法。这个收集器是在 JDK 1.6 中才开始提供的，在此之前，新生代的 Parallel Scavenge 收集器一直处于比较尴尬的状态。原因是，如果新生代选择了 Parallel Scavenge 收集器，老年代除了 Serial Old (PS MarkSweep) 收集器外别无选择（还记得上面说过 Parallel Scavenge 收集器无法与 CMS 收集器配合工作吗？）。由于单线程的老年代 Serial Old 收集器在服务端应用性能上的“拖累”，即便使用了 Parallel Scavenge 收集器也未必能在整体应用上获得吞吐量最大化的效果，又因为老年代收集中无法充分利用服务器多 CPU 的处理能力，在老年代很大而且硬件比较高级的环境中，这种组合的吞吐量甚至还不一定有 ParNew 加 CMS 的组合“给力”。

直到 Parallel Old 收集器出现后，“吞吐量优先”收集器终于有了比较名副其实的应用组合，在注重吞吐量及 CPU 资源敏感的场所，都可以优先考虑 Parallel Scavenge 加 Parallel Old 收集器。Parallel Old 收集器的工作过程如图 3-9 所示。

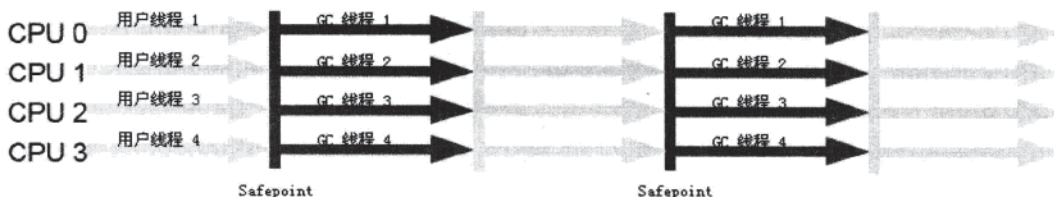


图 3-9 Parallel Scavenge / Parallel Old 收集器运行示意图

3.4.6 CMS 收集器

CMS (Concurrent Mark Sweep) 收集器是一种以获取最短回收停顿时间为目标的收集器。目前很大一部分的 Java 应用都集中在互联网站或 B/S 系统的服务端上，这类应用尤其重视服务的响应速度，希望系统停顿时间最短，以给用户带来较好的体验。CMS 收集器就非常符合这类应用的需求。

从名字（包含“Mark Sweep”）上就可以看出 CMS 收集器是基于“标记-清除”算法实现的，它的运作过程相对于前面几种收集器来说要更复杂一些，整个过程分为 4 个步骤，包括：

- 初始标记 (CMS initial mark)
- 并发标记 (CMS concurrent mark)
- 重新标记 (CMS remark)
- 并发清除 (CMS concurrent sweep)

其中初始标记、重新标记这两个步骤仍然需要“Stop The World”。初始标记仅仅只是标记一下 GC Roots 能直接关联到的对象，速度很快，并发标记阶段就是进行 GC Roots Tracing 的过程，而重新标记阶段则是为了修正并发标记期间，因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段稍长一些，但远比并发标记的时间短。

由于整个过程中耗时最长的并发标记和并发清除过程中，收集器线程都可以与用户线程一起工作，所以总体上来说，CMS 收集器的内存回收过程是与用户线程一起并发地执行的。通过图 3-10 可以比较清楚地看到 CMS 收集器的运作步骤中并发和需要停顿的时间。

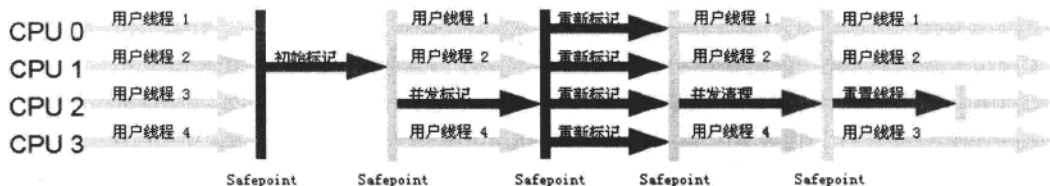


图 3-10 Concurrent Mark Sweep 收集器运行示意图

CMS 是一款优秀的收集器，它的最主要优点在名字上已经体现出来了：并发收集、低停顿，Sun 的一些官方文档里面也称之为并发低停顿收集器 (Concurrent Low Pause Collector)。但是 CMS 还远达不到完美的程度，它有以下三个显著的缺点：

- CMS 收集器对 CPU 资源非常敏感。其实，面向并发设计的程序都对 CPU 资源比较敏感。在并发阶段，它虽然不会导致用户线程停顿，但是会因为占用了一部分线程（或者说 CPU 资源）而导致应用程序变慢，总吞吐量会降低。CMS 默认启动的回收线程数是 $(\text{CPU 数量} + 3) / 4$ ，也就是当 CPU 在 4 个以上时，并发回收时垃圾收集线程最多占用不超过 25% 的 CPU 资源。但是当 CPU 不足 4 个时（譬如 2 个），那么 CMS 对用户程序的影响就可能变得很大，如果 CPU 负载本来就比较大的时候，还分出一半的运算能力去执行收集器线程，就可能导致用户程序

的执行速度忽然降低了 50%，这也让人受不了。为了解决这种情况，虚拟机提供了一种称为“增量式并发收集器”（Incremental Concurrent Mark Sweep / i-CMS）的 CMS 收集器变种，所做的事情和单 CPU 年代 PC 机操作系统使用抢占式来模拟多任务机制的思想一样，就是在并发标记和并发清理的时候让 GC 线程、用户线程交替运行，尽量减少 GC 线程的独占资源的时间，这样整个垃圾收集的过程会更长，但对用户程序的影响就会显得少一些，速度下降也就没有那么明显，但是目前版本中，i-CMS 已经被声明为“deprecated”，即不再提倡用户使用。

□ CMS 收集器无法处理浮动垃圾（Floating Garbage），可能出现“Concurrent Mode Failure”失败而导致另一次 Full GC 的产生。由于 CMS 并发清理阶段用户线程还在运行着，伴随程序的运行自然还会有新的垃圾不断产生，这一部分垃圾出现在标记过程之后，CMS 无法在本次收集处理掉它们，只好留待下一次 GC 时再将其清理掉。这一部分垃圾就称为“浮动垃圾”。也是由于在垃圾收集阶段用户线程还需要运行，即还需要预留足够的内存空间给用户线程使用，因此 CMS 收集器不能像其他收集器那样等到老年代几乎完全被填满了再进行收集，需要预留一部分空间提供并发收集时的程序运作使用。在默认设置下，CMS 收集器在老年代使用了 68% 的空间后就会被激活，这是一个偏保守的设置，如果在应用中老年代增长不是太快，可以适当调高参数 `-XX:CMSInitiatingOccupancyFraction` 的值来提高触发百分比，以便降低内存回收次数以获取更好的性能。要是 CMS 运行期间预留的内存无法满足程序需要，就会出现一次“Concurrent Mode Failure”失败，这时候虚拟机将启动后备预案：临时启用 Serial Old 收集器来重新进行老年代的垃圾收集，这样停顿时间就很长了。所以说参数 `-XX:CMSInitiatingOccupancyFraction` 设置得太高将会很容易导致大量“Concurrent Mode Failure”失败，性能反而降低。

□ 还有最后一个缺点，在本节在开头说过，CMS 是一款基于“标记 - 清除”算法实现的收集器，如果读者对前面这种算法介绍还有印象的话，就可能想到这意味着收集结束时会产生大量空间碎片。空间碎片过多时，将会给大对象分配带来很大的麻烦，往往会出现老年代还有很大的空间剩余，但是无法找到足够大的连续空间来分配当前对象，不得不提前触发一次 Full GC。为了解决这个问题，CMS 收集器提供了一个 `-XX:+UseCMSCompactAtFullCollection` 开关参数，用于在“享

受”完 Full GC 服务之后额外免费附送一个碎片整理过程，内存整理的过程是无法并发的。空间碎片问题没有了，但停顿时间不得不变长了。虚拟机设计者们还提供了另外一个参数 `-XX:CMSFullGCsBeforeCompaction`，这个参数用于设置在执行多少次不压缩的 Full GC 后，跟着来一次带压缩的。

3.4.7 G1 收集器

G1 (Garbage First) 收集器是当前收集器技术发展的最前沿成果，在 JDK 1.6_Update14 中提供了 Early Access 版本的 G1 收集器以供试用。在将来 JDK 1.7 正式发布的时候，G1 收集器很可能会有一个成熟的商用版本随之发布。这里只对 G1 收集器进行简单介绍^①。

G1 收集器是垃圾收集器理论进一步发展的产物，它与前面的 CMS 收集器相比有两个显著的改进：一是 G1 收集器是基于“标记 - 整理”算法实现的收集器，也就是说它不会产生空间碎片，这对于长时间运行的应用系统来说非常重要。二是它可以非常精确地控制停顿，既能让使用者明确指定在一个长度为 M 毫秒的时间片段内，消耗在垃圾收集上的时间不得超过 N 毫秒，这几乎已经是实时 Java (RTSJ) 的垃圾收集器的特征了。

G1 收集器可以实现在基本不牺牲吞吐量的前提下完成低停顿的内存回收，这是由于它能够极力地避免全区域的垃圾收集，之前的收集器进行收集的范围都是整个新生代或老年代，而 G1 将整个 Java 堆（包括新生代、老年代）划分为多个大小固定的独立区域 (Region)，并且跟踪这些区域里面的垃圾堆积程度，在后台维护一个优先列表，每次根据允许的收集时间，优先回收垃圾最多的区域（这就是 Garbage First 名称的来由）。区域划分及有优先级的区域回收，保证了 G1 收集器在有限的时间内可以获得最高的收集效率。

3.4.8 垃圾收集器参数总结

JDK 1.6 中的各种垃圾收集器到此已全部介绍完毕，在描述过程中提到了很多虚拟机非稳定的运行参数，表 3-1 整理了这些参数以供读者实践时参考。

^① G1 收集器最早在 2004 年的一篇论文中被提出，请参考：<http://labs.oracle.com/jtech/pubs/04-g1-paper-ismm.pdf>。

表 3-1 垃圾收集相关的常用参数

参 数	描 述
UseSerialGC	虚拟机运行在 Client 模式下的默认值，打开此开关后，使用 Serial + Serial Old 的收集器组合进行内存回收
UseParNewGC	打开此开关后，使用 ParNew + Serial Old 的收集器组合进行内存回收
UseConcMarkSweepGC	打开此开关后，使用 ParNew + CMS + Serial Old 的收集器组合进行内存回收。Serial Old 收集器将作为 CMS 收集器出现 Concurrent Mode Failure 失败后的后备收集器使用
UseParallelGC	虚拟机运行在 Server 模式下的默认值，打开此开关后，使用 Parallel Scavenge + Serial Old (PS MarkSweep) 的收集器组合进行内存回收
UseParallelOldGC	打开此开关后，使用 Parallel Scavenge + Parallel Old 的收集器组合进行内存回收
SurvivorRatio	新生代中 Eden 区域与 Survivor 区域的容量比值，默认为 8，代表 Eden : Survivor=8 : 1
PretenureSizeThreshold	直接晋升到老年代的对象大小，设置这个参数后，大于这个参数的对象将直接在老年代分配
MaxTenuringThreshold	晋升到老年代的对象年龄。每个对象在坚持过一次 Minor GC 之后，年龄就加 1，当超过这个参数值时就进入老年代
UseAdaptiveSizePolicy	动态调整 Java 堆中各个区域的大小以及进入老年代的年龄
HandlePromotionFailure	是否允许分配担保失败，即老年代的剩余空间不足以应付新生代的整个 Eden 和 Survivor 区的所有对象都存活的极端情况
ParallelGCThreads	设置并行 GC 时进行内存回收的线程数
GCTimeRatio	GC 时间占总时间的比率，默认值为 99，即允许 1% 的 GC 时间。仅在使用 Parallel Scavenge 收集器时生效
MaxGCPauseMillis	设置 GC 的最大停顿时间。仅在使用 Parallel Scavenge 收集器时生效
CMSInitiatingOccupancyFraction	设置 CMS 收集器在老年代空间被使用多少后触发垃圾收集。默认值为 68%，仅在使用 CMS 收集器时生效
UseCMSCompactAtFullCollection	设置 CMS 收集器在完成垃圾收集后是否要进行一次内存碎片整理。仅在使用 CMS 收集器时生效
CMSFullGCsBeforeCompaction	设置 CMS 收集器在进行若干次垃圾收集后再启动一次内存碎片整理。仅在使用 CMS 收集器时生效

3.5 内存分配与回收策略

Java 技术体系中所提倡的自动内存管理最终可以归结为自动化地解决了两个问题：给对象分配内存以及回收分配给对象的内存。关于回收内存这一点，我们已经使用了大

量的篇幅去介绍虚拟机中的垃圾收集器体系及其运作原理，现在我们再一起来探讨一下给对象分配内存的那点事儿。

对象的内存分配，往大方向上讲，就是在堆上分配（但也可能经过 JIT 编译后被拆散为标量类型并间接地在栈上分配^①），对象主要分配在新生代的 Eden 区上，如果启动了本地线程分配缓冲，将按线程优先在 TLAB 上分配。少数情况下也可能会直接分配在老年代中，分配的规则并不是百分之百固定的，其细节取决于当前使用的是哪一种垃圾收集器组合，还有虚拟机中与内存相关的参数的设置。

接下来我们将会讲解几条最普遍的内存分配规则，并通过代码去验证这些规则。本节中的代码在测试时使用 Client 模式虚拟机运行，没有手工指定收集器组合，换句话说，验证的是使用 Serial / Serial Old 收集器下（ParNew / Serial Old 收集器组合的规则也基本一致）的内存分配和回收的策略。读者不妨根据自己项目中使用的收集器写一些程序去验证一下使用其他几种收集器的内存分配策略。

3.5.1 对象优先在 Eden 分配

大多数情况下，对象在新生代 Eden 区中分配。当 Eden 区没有足够的空间进行分配时，虚拟机将发起一次 Minor GC。

虚拟机提供了 `-XX:+PrintGCDetails` 这个收集器日志参数，告诉虚拟机在发生垃圾收集行为时打印内存回收日志，并且在进程退出的时候输出当前内存各区域的分配情况。在实际应用中，内存回收日志一般是打印到文件后通过日志工具进行分析，不过本实验的日志并不多，直接阅读就能看得很清楚。

代码清单 3-3 的 `testAllocation()` 方法中，尝试分配 3 个 2MB 大小和 1 个 4MB 大小的对象，在运行时通过 `-Xms20M`、`-Xmx20M` 和 `-Xmn10M` 这 3 个参数限制 Java 堆大小为 20MB，且不可扩展，其中 10MB 分配给新生代，剩下的 10MB 分配给老年代。`-XX:SurvivorRatio=8` 决定了新生代中 Eden 区与一个 Survivor 区的空间比例是 8 比 1，从输出的结果也能清晰地看到“eden space 8192K、from space 1024K、to space 1024K”的信息，新生代总可用空间为 9216KB（Eden 区 + 1 个 Survivor 区的总容量）。

^① JIT 即时编译器相关优化可参见第 11 章。

执行 `testAllocation()` 中分配 `allocation4` 对象的语句时会发生一次 Minor GC，这次 GC 的结果是新生代 6651KB 变为 148KB，而总内存占用量则几乎没有减少（因为 `allocation1`、`2`、`3` 三个对象都是存活的，虚拟机几乎没有找到可回收的对象）。这次 GC 发生的原因是给 `allocation4` 分配内存的时候，发现 Eden 已经被占用了 6MB，剩余空间已不足以分配 `allocation4` 所需的 4MB 内存，因此发生 Minor GC。GC 期间虚拟机又发现已有的 3 个 2MB 大小的对象全部无法放入 Survivor 空间（Survivor 空间只有 1MB 大小），所以只好通过分配担保机制提前转移到老年代去。

这次 GC 结束后，4MB 的 `allocation4` 对象被顺利分配在 Eden 中。因此程序执行完的结果是 Eden 占用 4MB（被 `allocation4` 占用），Survivor 空闲，老年代被占用 6MB（被 `allocation1`、`2`、`3` 占用）。通过 GC 日志可以证实这一点。

注意 作者多次提到的 Minor GC 和 Full GC 有什么不一样吗？

- 新生代 GC (Minor GC)：指发生在新生代的垃圾收集动作，因为 Java 对象大多都具备朝生夕灭的特性，所以 Minor GC 非常频繁，一般回收速度也比较快。
 - 老年代 GC (Major GC / Full GC)：指发生在老年代的 GC，出现了 Major GC，经常会伴随至少一次的 Minor GC（但非绝对的，在 ParallelScavenge 收集器的收集策略里就有直接进行 Major GC 的策略选择过程）。MajorGC 的速度一般会比 Minor GC 慢 10 倍以上。
-

代码清单 3-3 新生代 Minor GC

```
private static final int _1MB = 1024 * 1024;

/**
 * VM 参数: -verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:SurvivorRatio=8
 */
public static void testAllocation() {
    byte[] allocation1, allocation2, allocation3, allocation4;
    allocation1 = new byte[2 * _1MB];
    allocation2 = new byte[2 * _1MB];
    allocation3 = new byte[2 * _1MB];
    allocation4 = new byte[4 * _1MB]; // 出现一次 Minor GC
}
```

运行结果：

```
[GC [DefNew: 6651K->148K(9216K), 0.0070106 secs] 6651K->6292K(19456K), 0.0070426
secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
Heap
 def new generation   total 9216K, used 4326K [0x029d0000, 0x033d0000, 0x033d0000)
   eden space 8192K,   51% used [0x029d0000, 0x02de4828, 0x031d0000)
   from space 1024K,   14% used [0x032d0000, 0x032f5370, 0x033d0000)
   to   space 1024K,   0% used [0x031d0000, 0x031d0000, 0x032d0000)
 tenured generation   total 10240K, used 6144K [0x033d0000, 0x03dd0000,
0x03dd0000)
   the space 10240K,   60% used [0x033d0000, 0x039d0030, 0x039d0200, 0x03dd0000)
 compacting perm gen  total 12288K, used 2114K [0x03dd0000, 0x049d0000,
0x07dd0000)
   the space 12288K,   17% used [0x03dd0000, 0x03fe0998, 0x03fe0a00, 0x049d0000)
 No shared spaces configured.
```

3.5.2 大对象直接进入老年代

所谓大对象就是指，需要大量连续内存空间的 Java 对象，最典型的大对象就是那种很长的字符串及数组（笔者例子中的 `byte[]` 数组就是典型的大对象）。大对象对虚拟机的内存分配来说就是一个坏消息（替 Java 虚拟机抱怨一句，比遇到一个大对象更加坏的消息就是遇到一群“朝生夕灭”的“短命大对象”，写程序的时候应当避免），经常出现大对象容易导致内存还有不少空间时就提前触发垃圾收集以获取足够的连续空间来“安置”它们。

虚拟机提供了一个 `-XX:PretenureSizeThreshold` 参数，令大于这个设置值的对象直接在老年代中分配。这样做的目的是避免在 Eden 区及两个 Survivor 区之间发生大量的内存拷贝（复习一下：新生代采用复制算法收集内存）。

执行代码清单 3-4 中的 `testPretenureSizeThreshold()` 方法后，我们看到 Eden 空间几乎没有被使用，而老年代 10MB 的空间被使用了 40%，也就是 4MB 的 allocation 对象直接就分配在老年代中，这是因为 `PretenureSizeThreshold` 被设置为 3MB（就是 3145728B，这个参数不能与 `-Xmx` 之类的参数一样直接写 3MB），因此超过 3MB 的对象都会直接在老年代中进行分配。

注意 `PretenureSizeThreshold` 参数只对 Serial 和 ParNew 两款收集器有效，Parallel Scavenge 收集器不认识这个参数，Parallel Scavenge 收集器一般并不需要设置。如果遇到必须使用此参数的场合，可以考虑 ParNew 加 CMS 的收集器组合。

代码清单 3-4 大对象直接进入老年代

```
private static final int _1MB = 1024 * 1024;

/**
 * VM 参数: -verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:SurvivorRatio=8
 * -XX:PretenureSizeThreshold=3145728
 */
public static void testPretenureSizeThreshold() {
    byte[] allocation;
    allocation = new byte[4 * _1MB]; // 直接分配在老年代中
}
```

运行结果：

```
Heap
def new generation total 9216K, used 671K [0x029d0000, 0x033d0000, 0x033d0000)
  eden space 8192K, 8% used [0x029d0000, 0x02a77e98, 0x031d0000)
  from space 1024K, 0% used [0x031d0000, 0x031d0000, 0x032d0000)
  to space 1024K, 0% used [0x032d0000, 0x032d0000, 0x033d0000)
tenured generation total 10240K, used 4096K [0x033d0000, 0x03dd0000,
0x03dd0000)
  the space 10240K, 40% used [0x033d0000, 0x037d0010, 0x037d0200, 0x03dd0000)
  compacting perm gen total 12288K, used 2107K [0x03dd0000, 0x049d0000,
0x07dd0000)
  the space 12288K, 17% used [0x03dd0000, 0x03fdefd0, 0x03fd0000, 0x049d0000)
No shared spaces configured.
```

3.5.3 长期存活的对象将进入老年代

虚拟机既然采用了分代收集的思想来管理内存，那内存回收时就必须能识别哪些对象应当放在新生代，哪些对象应放在老年代中。为了做到这点，虚拟机给每个对象定义了一个对象年龄（Age）计数器。如果对象在 Eden 出生并经过第一次 Minor GC 后仍然存活，并且能被 Survivor 容纳的话，将被移动到 Survivor 空间中，并将对象年龄设为 1。对象在 Survivor 区中每熬过一次 Minor GC，年龄就增加 1 岁，当它的年龄增加到一定程度（默认为 15 岁）时，就会被晋升到老年代中。对象晋升老年代的年龄阈值，可以通过参数 -XX:MaxTenuringThreshold 来设置。

读者可以试试分别以 -XX:MaxTenuringThreshold=1 和 -XX:MaxTenuringThreshold=15 两种设置来执行代码清单 3-5 中的 testTenuringThreshold() 方法，此方法中 allocation1

70 ❖ 第二部分 自动内存管理机制

对象需要 256KB 的内存空间，Survivor 空间可以容纳。当 MaxTenuringThreshold=1 时，allocation1 对象在第二次 GC 发生时进入老年代，新生代已使用的内存 GC 后会非常干净地变成 0KB。而 MaxTenuringThreshold=15 时，第二次 GC 发生后，allocation1 对象则还留在新生代 Survivor 空间，这时候新生代仍然有 404KB 的空间被占用。

代码清单 3-5 长期存活的对象进入老年代

```
private static final int _1MB = 1024 * 1024;

/**
 * VM 参 数: -verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:SurvivorRatio=8
-XX:MaxTenuringThreshold=1
 * -XX:+PrintTenuringDistribution
 */
@SuppressWarnings("unused")
public static void testTenuringThreshold() {
    byte[] allocation1, allocation2, allocation3;
    allocation1 = new byte[_1MB / 4];
    // 什么时候进入老年代取决于 XX:MaxTenuringThreshold 设置
    allocation2 = new byte[4 * _1MB];
    allocation3 = new byte[4 * _1MB];
    allocation3 = null;
    allocation3 = new byte[4 * _1MB];
}

```

以 MaxTenuringThreshold=1 的参数设置来运行的结果：

```
[GC [DefNew
Desired Survivor size 524288 bytes, new threshold 1 (max 1)
- age 1: 414664 bytes, 414664 total
: 4859K->404K(9216K), 0.0065012 secs] 4859K->4500K(19456K), 0.0065283 secs]
[Times: user=0.02 sys=0.00, real=0.02 secs]
[GC [DefNew
Desired Survivor size 524288 bytes, new threshold 1 (max 1)
: 4500K->0K(9216K), 0.0009253 secs] 8596K->4500K(19456K), 0.0009458 secs] [Times:
user=0.00 sys=0.00, real=0.00 secs]
Heap
def new generation total 9216K, used 4178K [0x029d0000, 0x033d0000, 0x033d0000)
eden space 8192K, 51% used [0x029d0000, 0x02de4828, 0x031d0000)
from space 1024K, 0% used [0x031d0000, 0x031d0000, 0x032d0000)

```

```

to space 1024K, 0% used [0x032d0000, 0x032d0000, 0x033d0000)
tenured generation total 10240K, used 4500K [0x033d0000, 0x03dd0000,
0x03dd0000)
    the space 10240K, 43% used [0x033d0000, 0x03835348, 0x03835400, 0x03dd0000)
    compacting perm gen total 12288K, used 2114K [0x03dd0000, 0x049d0000,
0x07dd0000)
        the space 12288K, 17% used [0x03dd0000, 0x03fe0998, 0x03fe0a00, 0x049d0000)
No shared spaces configured.

```

以 MaxTenuringThreshold=15 的参数设置来运行的结果：

```

[GC [DefNew
Desired Survivor size 524288 bytes, new threshold 15 (max 15)
- age 1: 414664 bytes, 414664 total
: 4859K->404K(9216K), 0.0049637 secs] 4859K->4500K(19456K), 0.0049932 secs]
[Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [DefNew
Desired Survivor size 524288 bytes, new threshold 15 (max 15)
- age 2: 414520 bytes, 414520 total
: 4500K->404K(9216K), 0.0008091 secs] 8596K->4500K(19456K), 0.0008305 secs]
[Times: user=0.00 sys=0.00, real=0.00 secs]
Heap
def new generation total 9216K, used 4582K [0x029d0000, 0x033d0000, 0x033d0000)
eden space 8192K, 51% used [0x029d0000, 0x02de4828, 0x031d0000)
    from space 1024K, 39% used [0x031d0000, 0x03235338, 0x032d0000)
to space 1024K, 0% used [0x032d0000, 0x032d0000, 0x033d0000)
tenured generation total 10240K, used 4096K [0x033d0000, 0x03dd0000,
0x03dd0000)
    the space 10240K, 40% used [0x033d0000, 0x037d0010, 0x037d0200, 0x03dd0000)
    compacting perm gen total 12288K, used 2114K [0x03dd0000, 0x049d0000,
0x07dd0000)
        the space 12288K, 17% used [0x03dd0000, 0x03fe0998, 0x03fe0a00, 0x049d0000)
No shared spaces configured.

```

3.5.4 动态对象年龄判定

为了更好地适应不同程序的内存状况，虚拟机并不总是要求对象的年龄必须达到 MaxTenuringThreshold 才能晋升老年代，如果在 Survivor 空间中相同年龄所有对象大小的总和大于 Survivor 空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代，无须等到 MaxTenuringThreshold 中要求的年龄。

72 ❖ 第二部分 自动内存管理机制

执行代码清单 3-6 中的 testTenuringThreshold2() 方法，并设置参数 -XX:MaxTenuringThreshold=15，会发现运行结果中 Survivor 的空间占用仍然为 0%，而老年代比预期增加了 6%，也就是说 allocation1、allocation2 对象都直接进入了老年代，而没有等到 15 岁的临界年龄。因为这两个对象加起来已经达到了 512KB，并且它们是同年代的，满足同年对象达到 Survivor 空间的一半规则。我们只要注释掉其中一个对象的新操作，就会发现另外一个不会晋升到老年代中去了。

代码清单 3-6 动态对象年龄判定

```
private static final int _1MB = 1024 * 1024;

/**
 * VM 参 数: -verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:SurvivorRatio=8
-XX:MaxTenuringThreshold=15
 * -XX:+PrintTenuringDistribution
 */
@SuppressWarnings("unused")
public static void testTenuringThreshold2() {
    byte[] allocation1, allocation2, allocation3, allocation4;
    allocation1 = new byte[_1MB / 4];
    // allocation1+allocation2 大于 survivor 空间的一半
    allocation2 = new byte[_1MB / 4];
    allocation3 = new byte[4 * _1MB];
    allocation4 = new byte[4 * _1MB];
    allocation4 = null;
    allocation4 = new byte[4 * _1MB];
}
}
```

运行结果：

```
[GC [DefNew
Desired Survivor size 524288 bytes, new threshold 1 (max 15)
- age 1: 676824 bytes, 676824 total
: 5115K->660K(9216K), 0.0050136 secs] 5115K->4756K(19456K), 0.0050443 secs]
[Times: user=0.00 sys=0.01, real=0.01 secs]
[GC [DefNew
Desired Survivor size 524288 bytes, new threshold 15 (max 15)
: 4756K->0K(9216K), 0.0010571 secs] 8852K->4756K(19456K), 0.0011009 secs] [Times:
user=0.00 sys=0.00, real=0.00 secs]
Heap
```

```
def new generation    total 9216K, used 4178K [0x029d0000, 0x033d0000, 0x033d0000)
  eden space 8192K,   51% used [0x029d0000, 0x02de4828, 0x031d0000)
  from space 1024K,   0% used [0x031d0000, 0x031d0000, 0x032d0000)
  to   space 1024K,   0% used [0x032d0000, 0x032d0000, 0x033d0000)
  tenured generation    total 10240K, used 4756K [0x033d0000, 0x03dd0000,
0x03dd0000)
    the space 10240K,   46% used [0x033d0000, 0x038753e8, 0x03875400, 0x03dd0000)
    compacting perm gen    total 12288K, used 2114K [0x03dd0000, 0x049d0000,
0x07dd0000)
      the space 12288K,   17% used [0x03dd0000, 0x03fe09a0, 0x03fe0a00, 0x049d0000)
No shared spaces configured.
```

3.5.5 空间分配担保

在发生 Minor GC 时，虚拟机检测之前每次晋升到老年代的平均大小是否大于老年代的剩余空间大小，如果大于，则改为直接进行一次 Full GC。如果小于，则查看 HandlePromotionFailure 设置是否允许担保失败；如果允许，那只会进行 Minor GC；如果不允许，则也要改为进行一次 Full GC。

前面提到过，新生代使用复制收集算法，但为了内存利用率，只使用其中一个 Survivor 空间来作为轮换备份，因此当出现大量对象在 Minor GC 后仍然存活的情况时（最极端就是内存回收后新生代中所有对象都存活），就需要老年代进行分配担保，让 Survivor 无法容纳的对象直接进入老年代。与生活中的贷款担保类似，老年代要进行这样的担保，前提是老年代本身还有容纳这些对象的剩余空间，一共有多少对象会活下来，在实际完成内存回收之前是无法明确知道的，所以只好取之前每一次回收晋升到老年代对象容量的平均大小值作为经验值，与老年代的剩余空间进行比较，决定是否进行 Full GC 来让老年代腾出更多空间。

取平均值进行比较其实仍然是一种动态概率的手段，也就是说如果某次 Minor GC 存活后的对象突增，远远高于平均值的话，依然会导致担保失败（HandlePromotionFailure）。如果出现了 HandlePromotionFailure 失败，那就只好在失败后重新发起一次 Full GC。虽然担保失败时绕的圈子是最大的，但大部分情况下都还是会将 HandlePromotionFailure 开关打开，避免 Full GC 过于频繁，参见代码清单 3-7。

代码清单 3-7 空间分配担保

```

private static final int _1MB = 1024 * 1024;

/**
 * VM 参数: -verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:SurvivorRatio=8 -XX:-
HandlePromotionFailure
 */
@SuppressWarnings("unused")
public static void testHandlePromotion() {
    byte[] allocation1, allocation2, allocation3, allocation4, allocation5,
allocation6, allocation7;
    allocation1 = new byte[2 * _1MB];
    allocation2 = new byte[2 * _1MB];
    allocation3 = new byte[2 * _1MB];
    allocation1 = null;
    allocation4 = new byte[2 * _1MB];
    allocation5 = new byte[2 * _1MB];
    allocation6 = new byte[2 * _1MB];
    allocation4 = null;
    allocation5 = null;
    allocation6 = null;
    allocation7 = new byte[2 * _1MB];
}

```

以 `HandlePromotionFailure = false` 的参数设置来运行的结果：

```

[GC [DefNew: 6651K->148K(9216K), 0.0078936 secs] 6651K->4244K(19456K), 0.0079192
secs] [Times: user=0.00 sys=0.02, real=0.02 secs]
[GC [DefNew: 6378K->6378K(9216K), 0.0000206 secs][Tenured: 4096K->4244K(10240K),
0.0042901 secs] 10474K->4244K(19456K), [Perm : 2104K->2104K(12288K)], 0.0043613 secs]
[Times: user=0.00 sys=0.00, real=0.00 secs]

```

以 `MaxTenuringThreshold= true` 的参数设置来运行的结果：

```

[GC [DefNew: 6651K->148K(9216K), 0.0054913 secs] 6651K->4244K(19456K), 0.0055327
secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [DefNew: 6378K->148K(9216K), 0.0006584 secs] 10474K->4244K(19456K), 0.0006857
secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

```

3.6 本章小结

本章介绍了垃圾收集的算法、几款 JDK 1.6 中提供的垃圾收集器特点及其运作原理。通过代码实例验证了 Java 虚拟机中自动内存分配及回收的主要规则。

内存回收与垃圾收集器在很多时候都是影响系统性能、并发能力的主要因素之一，虚拟机之所以提供多种不同的收集器及大量的调节参数，是因为只有根据实际应用需求、实现方式选择最优的收集方式才能获取最好的性能。没有固定收集器、参数组合，也没有最优的调优方法，虚拟机也没有什么必然的内存回收行为。因此学习虚拟机内存知识，如果要到实践调优阶段，必须了解每个具体收集器的行为、优势和劣势、调节参数。在接下来的两章中，作者将会介绍内存分析的工具和调优的一些具体案例。



第 4 章 虚拟机性能监控与故障处理工具

本章主要内容

- 概述
- JDK 的命令行工具
- JDK 的可视化工具

Java 与 C++ 之间有一堵由内存动态分配和垃圾收集技术所围成的高墙，墙外面的人想进去，墙里面的人却想出来。

4.1 概述

经过前面两章对于虚拟机内存分配与回收技术各方面的介绍，相信读者已经建立了一个比较完整的理论基础。理论总是作为指导实践的工具，能把这些知识投入到实际工作中才是我们的最终目的。接下来的两章，我们将从实践的角度去了解虚拟机内存管理的世界。

给一个系统定位问题的时候，知识、经验是关键基础，数据是依据，工具是运用知识处理数据的手段。这里说的数据包括：运行日志、异常堆栈、GC 日志、线程快照（`threaddump` / `javacore` 文件）、堆转储快照（`heapdump` / `hprof` 文件）等。经常使用适当的虚拟机监控和分析的工具可以加快我们分析数据和定位解决问题的速度，但我们在学习工具前，也应当意识到工具永远都是知识技能的一层包装，没有什么工具是“秘密武器”，学会了就能包治百病。

4.2 JDK 的命令行工具

Java 开发人员肯定都知道 JDK 的 bin 目录中有“`java.exe`”和“`javac.exe`”这两个

命令行工具，但并非所有程序员都了解过 JDK 的 bin 目录之中其他命令程序的作用。每逢 JDK 更新版本之时，bin 目录下命令行工具的数量和功能总会不知不觉地增加和增强，bin 目录的内容如图 4-1 所示。



名称	修改日期	类型	大小
xjc.exe	2010/7/19 11:56	应用程序	27 KB
wsimport.exe	2010/7/19 11:56	应用程序	27 KB
wsgen.exe	2010/7/19 11:56	应用程序	27 KB
unpack200.exe	2010/7/19 11:56	应用程序	124 KB
tnameserv.exe	2010/7/19 11:56	应用程序	27 KB
servertool.exe	2010/7/19 11:56	应用程序	27 KB
serialver.exe	2010/7/19 11:56	应用程序	27 KB
schemagen.exe	2010/7/19 11:56	应用程序	27 KB
rmiregistry.exe	2010/7/19 11:56	应用程序	27 KB
rmid.exe	2010/7/19 11:56	应用程序	27 KB
rmic.exe	2010/7/19 11:56	应用程序	27 KB
policytool.exe	2010/7/19 11:56	应用程序	27 KB
packager.exe	2010/7/19 11:56	应用程序	72 KB

图 4-1 Sun JDK 中的工具目录

在本章中，笔者将介绍这些工具的其中一部分，主要是用于监视虚拟机和故障处理的工具。这些故障处理工具被 Sun 公司作为“礼物”附赠给 JDK 的使用者，在软件的使用说明中把它们声明为“没有技术支持并且是实验性质的”（unsupported and experimental）^①的产品，但事实上这些工具都非常稳定而且功能强大，能在处理应用程序性能问题、定位故障时发挥很大的作用。

说起 JDK 的工具，读者如果比较细心的话，可能会注意到这些工具的程序体积都异常的小。假如以前没注意到，现在不妨再看看图 4-1 中的最后一列“大小”，各个工具的体积基本上都稳定在 27KB 左右。并非 JDK 开发团队刻意把它们制作得如此精炼来炫耀

^① <http://download.oracle.com/javase/6/docs/technotes/tools/index.html>.

78 ❖ 第二部分 自动内存管理机制

编程水平，而是因为这些命令行工具大多数是 `jdk\lib\tools.jar` 类库的一层薄包装而已，它们主要的功能代码是在 `tools` 类库中实现的。读者把图 4-1 和图 4-2 两张图片对比一下就可以看得很清楚。

假如读者使用的是 Linux 版本的 JDK，还会发现这些工具中很多甚至就是由 Shell 脚本直接写成的，可以用 `vim` 直接打开它们。

JDK 开发团队选择采用 Java 代码来实现这些监控工具有特别用意的：当应用程序部署到生产环境后，无论是直接接触物理服务器还是远程 Telnet 到服务器上都有可能受到限制。借助 `tools.jar` 类库里面的接口，我们可以直接在应用程序中实现功能强大的监控分析功能^①。

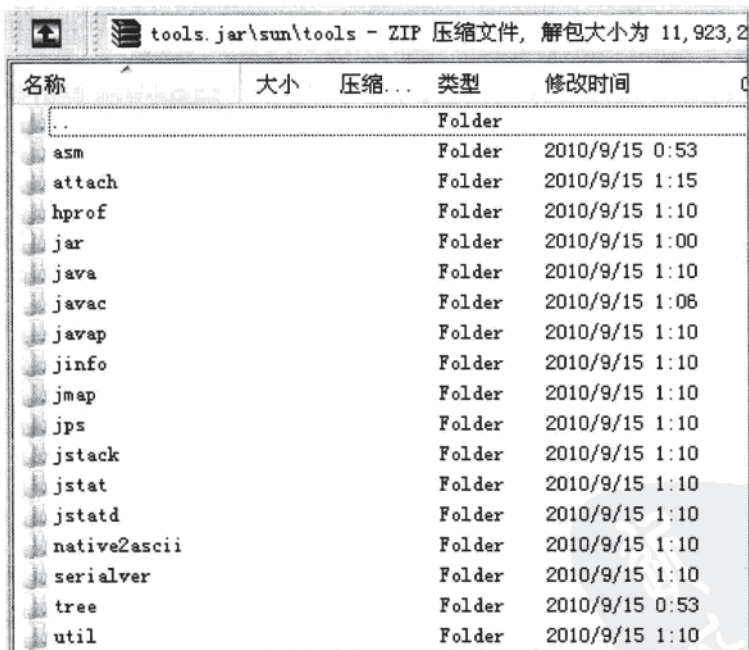


图 4-2 tools.jar 包的内部状况

^① `tools.jar` 中的类库不属于 Java 的标准 API，如果引入这个类库，就意味着你的程序只能运行于 Sun Hotspot（或一些从 Sun 买了 JDK 的源码 License 的虚拟机，如 IBM J9、BEA JRockit）上面，或者在部署程序时需要一起部署 `tools.jar`。

需要特别说明的是，本章介绍的工具全部基于 Windows 平台下的 JDK 1.6 Update 21，如果 JDK 版本、操作系统不同，工具所支持的功能可能会有较大差别。大部分工具在 JDK 1.5 中就已经提供，但为了避免运行环境带来的差异和兼容性问题，建议读者使用 JDK 1.6 来验证本章介绍的内容，因为 JDK 1.6 的工具可以正常兼容运行于 JDK 1.5 的虚拟机之上的程序，反之则不一定。表 4-1 中列举了 JDK 主要命令行监控工具的用途。

注意 如果读者在工作中需要监控运行于 JDK 1.5 的虚拟机之上的程序，在程序启动时请添加参数“-Dcom.sun.management.jmxremote”开启 JMX 管理功能，否则由于部分工具都是基于 JMX 的（包括下一节的可视化工具），因此它们都将会无法使用，如果被监控程序运行于 JDK 1.6 的虚拟机之上，那 JMX 管理默认是开启的，虚拟机启动时无须再添加任何参数。

表 4-1 Sun JDK 监控和故障处理工具

名称	主要作用
jps	JVM Process Status Tool，显示指定系统内所有的 HotSpot 虚拟机进程
jstat	JVM Statistics Monitoring Tool，用于收集 HotSpot 虚拟机各方面的运行数据
jinfo	Configuration Info for Java，显示虚拟机配置信息
jmap	Memory Map for Java，生成虚拟机的内存转储快照（heapdump 文件）
jhat	JVM Heap Dump Browser，用于分析 heapdump 文件，它会建立一个 HTTP/HTML 服务器，让用户可以在浏览器上查看分析结果
jstack	Stack Trace for Java，显示虚拟机的线程快照

4.2.1 jps：虚拟机进程状况工具

JDK 的很多小工具的名字都参考了 Unix 命令的命名方式，jps（JVM Process Status Tool）是其中的典型。除了名字像 Unix 的 ps 命令之外，它的功能也和 ps 命令类似：可以列出正在运行的虚拟机进程，并显示虚拟机执行主类（Main Class，main() 函数所在的类）的名称，以及这些进程的本地虚拟机的唯一 ID（LVMID，Local Virtual Machine Identifier）。虽然功能比较单一，但它是使用频率最高的 JDK 命令行工具，因为其他的 JDK 工具大多须要输入它查询到的 LVMID 来确定要监控的是哪一个虚拟机进程。对于本地虚拟机进程来说，LVMID 与操作系统的进程 ID（PID，Process Identifier）是一致的，使用 Windows 的任务管理器或 Unix 的 ps 命令也可以查询到虚拟机进程的 LVMID，但

80 ❖ 第二部分 自动内存管理机制

如果同时启动了多个虚拟机进程，无法根据进程名称定位时，那就只能依赖 `jps` 命令显示主类的功能才能区分了。

`jps` 命令格式：

```
jps [ options ] [ hostid ]
```

`jps` 执行样例：

```
D:\Develop\Java\jdk1.6.0_21\bin>jps -l
2388 D:\Develop\glassfish\bin\..\modules\admin-cli.jar
2764 com.sun.enterprise.glassfish.bootstrap.ASMain
3788 sun.tools.jps.Jps
```

`jps` 可以通过 RMI 协议查询开启了 RMI 服务的远程虚拟机进程状态，`hostid` 为 RMI 注册表中注册的主机名。`jps` 的其他常用选项见表 4-2。

表 4-2 `jps` 工具主要选项

选项	作用
-q	只输出 LVMIID，省略主类的名称
-m	输出虚拟机进程启动时传递给主类 <code>main()</code> 函数的参数
-l	输出主类的全名，如果进程执行的是 Jar 包，输出 Jar 路径
-v	输出虚拟机进程启动时 JVM 参数

4.2.2 `jstat`：虚拟机统计信息监视工具

`jstat` (JVM Statistics Monitoring Tool) 是用于监视虚拟机各种运行状态信息的命令行工具。它可以显示本地或远程^①虚拟机进程中的类装载、内存、垃圾收集、JIT 编译等运行数据，在没有 GUI 图形界面，只提供了纯文本控制台环境的服务器上，它将是运行期定位虚拟机性能问题的首选工具。

`jstat` 命令格式为：

```
jstat [ option vmid [interval[s|ms] [count]] ]
```

对于命令格式中的 VMID 与 LVMIID 需要特别说明一下：如果是本地虚拟机进程，VMID 与 LVMIID 是一致的，如果是远程虚拟机进程，那 VMID 的格式应当是：

① 需要远程主机提供 RMI 支持，Sun 提供了 `jstatd` 工具可以很方便地建立远程 RMI 服务器。

```
[protocol:] [//]lvmid[@hostname[:port]/servername]
```

参数 interval 和 count 代表查询间隔和次数，如果省略这两个参数，说明只查询一次。假设需要每 250 毫秒查询一次进程 2764 垃圾收集的状况，一共查询 20 次，那命令应当是：

```
jstat -gc 2764 250 20
```

选项 option 代表着用户希望查询的虚拟机信息，主要分为 3 类：类装载、垃圾收集和运行期编译状况，具体选项及作用请参考表 4-3 中的描述。

表 4-3 jstat 工具主要选项

选 项	作 用
-class	监视类装载、卸载数量、总空间及类装载所耗费的时间
-gc	监视 Java 堆状况，包括 Eden 区、2 个 survivor 区、老年代、永久代等的容量、已用空间、GC 时间合计等信息
-gccapacity	监视内容与 -gc 基本相同，但输出主要关注 Java 堆各个区域使用到的最大和最小空间
-gcutil	监视内容与 -gc 基本相同，但输出主要关注已使用空间占总空间的百分比
-gccause	与 -gcutil 功能一样，但是会额外输出导致上一次 GC 产生的原因
-gcnnew	监视新生代 GC 的状况
-gcnnewcapacity	监视内容与 -gcnnew 基本相同，输出主要关注使用到的最大和最小空间
-gcold	监视老年代 GC 的状况
-gcoldcapacity	监视内容与 -gcold 基本相同，输出主要关注使用到的最大和最小空间
-gcpermcapacity	输出永久代使用到的最大和最小空间
-compiler	输出 JIT 编译器编译过的方法、耗时等信息
-printcompilation	输出已经被 JIT 编译的方法

jstat 监视选项众多，囿于版面原因无法逐一演示，这里仅举一例监视一台刚刚启动的 GlassFish v3 服务器的内存状况的例子来演示如何查看监视结果。监视参数与输出结果如代码清单 4-1 所示。

代码清单 4-1 jstat 执行样例

```
D:\Develop\Java\jdk1.6.0_21\bin>jstat -gcutil 2764
```

S0	S1	E	O	P	YGC	YGCT	FGC	FGCT	GCT
0.00	0.00	6.20	41.42	47.20	16	0.105	3	0.472	0.577

查询结果表明：这台服务器的新生代 Eden 区（E，表示 Eden）使用了 6.2% 的空间，两个 Survivor 区（S0、S1，表示 Survivor0、Survivor1）里面都是空的，老年代（O，

82 ❖ 第二部分 自动内存管理机制

表示 Old) 和永久代 (P, 表示 Permanent) 则分别使用了 41.42% 和 47.20% 的空间。程序运行以来共发生 Minor GC (YGC, 表示 Young GC) 16 次, 总耗时 0.105 秒, 发生 Full GC (FGC, 表示 Full GC) 3 次, Full GC 总耗时 (FGCT, 表示 Full GC Time) 为 0.472 秒, 所有 GC 总耗时 (GCT, 表示 GC Time) 为 0.577 秒。

使用 jstat 工具在纯文本状态下监视虚拟机状态的变化, 确实不如后面将会提到的 VisualVM 等可视化的监视工具直接以图表展现的那样直观。但许多服务器管理员都习惯了在文本控制台中工作, 直接在控制台中使用 jstat 命令依然是一种常用的监控方式。

4.2.3 jinfo : Java 配置信息工具

jinfo (Configuration Info for Java) 的作用是实时地查看和调整虚拟机的各项参数。使用 jps 命令的 -v 参数可以查看虚拟机启动时显式指定的参数列表, 但如果想知道未被显式指定的参数的系统默认值, 除了去找资料外, 就只能使用 jinfo 的 -flag 选项进行查询了 (如果只限于 JDK 1.6 或以上版本的话, 使用 java -XX:+PrintFlagsFinal 查看参数默认值也是一个很好的选择), jinfo 还可以使用 -sysprops 选项把虚拟机进程的 System.getProperties() 的内容打印出来。这个命令在 JDK 1.5 时期已经随着 Linux 版的 JDK 发布, 当时只提供了信息查询的功能, JDK 1.6 之后, jinfo 在 Windows 和 Linux 平台都有提供, 并且加入了运行期修改参数的能力, 可以使用 -flag [+|-]name 或 -flag name=value 修改一部分运行期可写的虚拟机参数值。JDK 1.6 中, jinfo 对于 Windows 平台的功能仍然有较大的限制, 只提供了最基本的 -flag 选项。

jinfo 命令格式:

```
jinfo [ option ] pid
```

执行样例: 查询 CMSInitiatingOccupancyFraction 参数值。

```
C:\>jinfo -flag CMSInitiatingOccupancyFraction 1444  
-XX:CMSInitiatingOccupancyFraction=85
```

4.2.4 jmap : Java 内存映像工具

jmap (Memory Map for Java) 命令用于生成堆转储快照 (一般称为 heapdump 或 dump 文件)。如果不使用 jmap 命令, 要想获取 Java 堆转储快照还有一些比较“暴力”

的手段：譬如在第 2 章中用过的 `-XX:+HeapDumpOnOutOfMemoryError` 参数，可以让虚拟机在 OOM 异常出现之后自动生成 dump 文件，通过 `-XX:+HeapDumpOnCtrlBreak` 参数则可以使用 `[Ctrl]+[Break]` 键让虚拟机生成 dump 文件，又或者在 Linux 系统下通过 `Kill -3` 命令发送进程退出信号“恐吓”一下虚拟机，也能拿到 dump 文件。

`jmap` 的作用并不仅仅是为了获取 dump 文件，它还可以查询 `finalize` 执行队列，Java 堆和永久代的详细信息，如空间使用率、当前用的是哪种收集器等。

和 `jinfo` 命令一样，`jmap` 有不少功能在 Windows 平台下都是受限的，除了生成 dump 文件的 `-dump` 选项和用于查看每个类的实例、空间占用统计的 `-histo` 选项所有操作系统都提供之外，其余选项都只能在 Linux / Solaris 下使用。

`jmap` 命令格式：

```
jmap [ option ] vmid
```

option 选项的合法值与具体含义如表 4-4 所示。

表 4-4 jmap 工具主要选项

选 项	作 用
<code>-dump</code>	生成 Java 堆转储快照。格式为： <code>-dump:[live,]format=b,file=<filename></code> ，其中 <code>live</code> 子参数说明是否只 dump 出存活的对象
<code>-finalizerinfo</code>	显示在 <code>F-Queue</code> 中等待 <code>Finalizer</code> 线程执行 <code>finalize</code> 方法的对象。只在 Linux / Solaris 平台下有效
<code>-heap</code>	显示 Java 堆详细信息，如使用哪种回收器、参数配置、分代状况等。只在 Linux / Solaris 平台下有效
<code>-histo</code>	显示堆中对象统计信息，包括类、实例数量和合计容量
<code>-permstat</code>	以 <code>ClassLoader</code> 为统计口径显示永久代内存状态。只在 Linux / Solaris 平台下有效
<code>-F</code>	当虚拟机进程对 <code>-dump</code> 选项没有响应时，可使用这个选项强制生成 dump 快照。只在 Linux / Solaris 平台下有效

代码清单 4-2 是使用 `jmap` 生成一个正在运行的 Eclipse 的 dump 快照文件的例子，例子中的 3500 是通过 `jps` 命令查询到的 LVMID。

代码清单 4-2 使用 `jmap` 生成 dump 文件

```
C:\Users\IcyFenix>jmap -dump:format=b,file=eclipse.bin 3500
Dumping heap to C:\Users\IcyFenix\eclipse.bin ...
Heap dump file created
```

4.2.5 jhat : 虚拟机堆转储快照分析工具

Sun JDK 提供 jhat (JVM Heap Analysis Tool) 命令与 jmap 搭配使用, 来分析 jmap 生成的堆转储快照。jhat 内置了一个微型的 HTTP/HTML 服务器, 生成 dump 文件的分析结果后, 可以在浏览器中查看。不过实事求是地说, 在实际工作中, 除非笔者手上真的没有别的工具可用, 否则一般都不会去直接使用 jhat 命令来分析 dump 文件, 主要原因有二: 一是一般不会在部署应用程序的服务器上直接分析 dump 文件, 即使可以这样做, 也会尽量将 dump 文件拷贝到其他机器^①上进行分析, 因为分析工作是一个耗时而且消耗硬件资源的过程, 既然都要在其他机器上进行, 就没必要受到命令行工具的限制了。另外一个原因是 jhat 的分析功能相对来说比较简陋, 后文将会介绍到的 VisualVM, 以及专业用于分析 dump 文件的 Eclipse Memory Analyzer、IBM HeapAnalyzer^②等工具, 都能实现比 jhat 更强大更专业的分析功能。代码清单 4-3 演示了使用 jhat 分析上一节采用 jmap 生成的 Eclipse IDE 的内存快照文件。

代码清单 4-3 使用 jhat 分析 dump 文件

```
C:\Users\IcyFenix>jhat eclipse.bin
Reading from eclipse.bin...
Dump file created Fri Nov 19 22:07:21 CST 2010
Snapshot read, resolving...
Resolving 1225951 objects...
Chasing references, expect 245 dots....
Eliminating duplicate references...
Snapshot resolved.
Started HTTP server on port 7000
Server is ready.
```

屏幕显示“Server is ready.”的提示后, 用户在浏览器中键入 <http://localhost:7000/> 就可以看到分析结果, 如图 4-3 所示。

分析结果默认以包为单位进行分组显示, 分析内存泄漏问题主要会使用到其中的“Heap Histogram”(与 jmap-histo 功能一样)与 OQL 页签的功能, 前者可以找到内存中

-
- ① 用于分析的机器一般也是服务器, 由于加载 dump 快照文件需要比生成 dump 更大的内存, 所以一般在 64 位 JDK、大内存的服务器上进行。
 - ② IBM HeapAnalyzer 用于分析 IBM J9 虚拟机生成的映像文件, 各个虚拟机产生的映像文件格式并不一致, 所以分析工具也不能通用。

总容量最大的对象，后者是标准的对象查询语言，使用类似 SQL 的语法对内存中的对象进行查询统计，读者若对 OQL 有兴趣的话可以参考本书附录 D 的介绍。

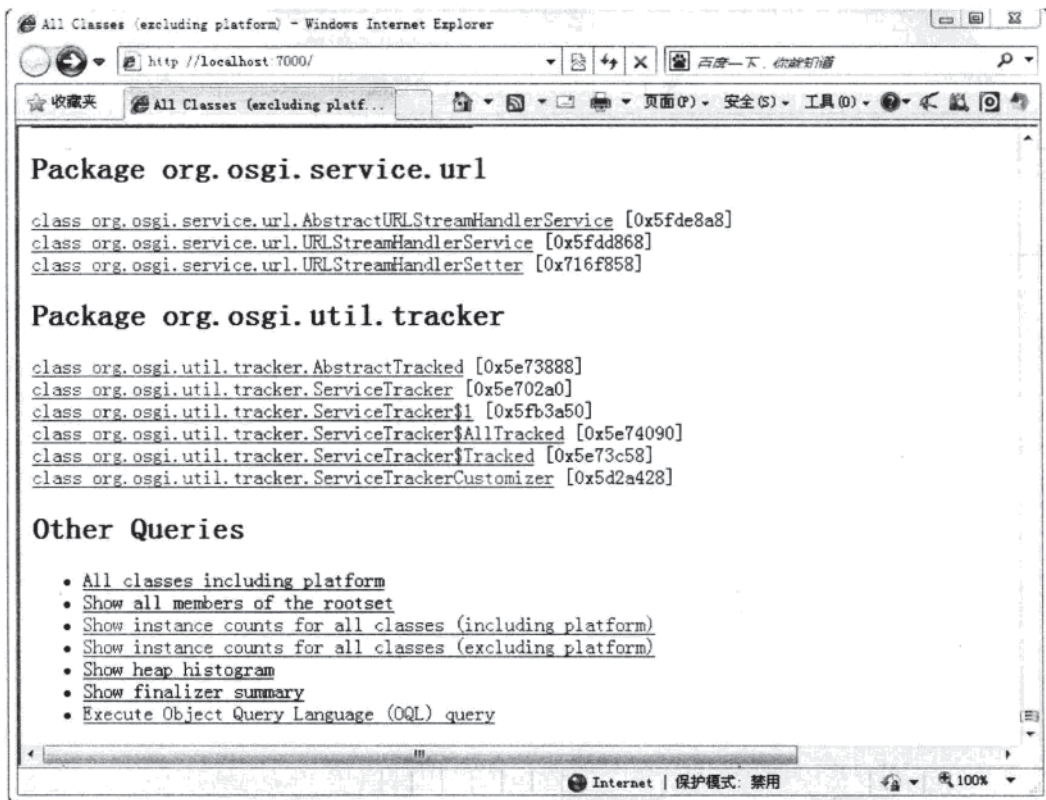


图 4-3 jhat 的分析结果

4.2.6 jstack : Java 堆栈跟踪工具

jstack (Stack Trace for Java) 命令用于生成虚拟机当前时刻的线程快照 (一般称为 threaddump 或 javacore 文件)。线程快照就是当前虚拟机内每一条线程正在执行的方法堆栈的集合，生成线程快照的主要目的是定位线程出现长时间停顿的原因，如线程间死锁、死循环、请求外部资源导致的长时间等待等都是导致线程长时间停顿的常见原因。线程出现停顿的时候通过 jstack 来查看各个线程的调用堆栈，就可以知道没有响应的线程到底在后台做些什么事情，或者等待着什么资源。

86 ❖ 第二部分 自动内存管理机制

jstack 命令格式：

```
jstack [ option ] vmid
```

option 选项的合法值与具体含义如表 4-5 所示。

表 4-5 jstack 工具的主要选项

选项	作 用
-F	当正常输出的请求不被响应时，强制输出线程堆栈
-l	除堆栈外，显示关于锁的附加信息
-m	如果调用到本地方法的话，可以显示 C/C++ 的堆栈

代码清单 4-4 是使用 jstack 查看 Eclipse 线程堆栈的例子，例子中的 3500 是通过 jps 命令查询到的 LVMID。

代码清单 4-4 使用 jstack 查看线程堆栈（部分结果）

```
C:\Users\IcyFenix>jstack -l 3500
2010-11-19 23:11:26
Full thread dump Java HotSpot(TM) 64-Bit Server VM (17.1-b03 mixed mode):
"[ThreadPool Manager] - Idle Thread" daemon prio=6 tid=0x0000000039dd4000
nid=0xf50 in Object.wait() [0x000000003c96f000]
    java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
            - waiting on <0x0000000016bdcc60> (a org.eclipse.equinox.internal.util.impl.
tpt.threadpool.Executor)
        at java.lang.Object.wait(Object.java:485)
        at org.eclipse.equinox.internal.util.impl.tpt.threadpool.Executor.run(Executor.
java:106)
            - locked <0x0000000016bdcc60> (a org.eclipse.equinox.internal.util.impl.tpt.
threadpool.Executor)

Locked ownable synchronizers:
    - None
```

在 JDK 1.5 中，java.lang.Thread 类新增了一个 getAllStackTraces() 方法用于获取虚拟机中所有线程的 StackTraceElement 对象。使用这个方法可以通过简单的几行代码就完成 jstack 的大部分功能，在实际项目中不妨调用这个方法做个管理员页面，可以随时使用浏览器来查看线程堆栈，如代码清单 4-5 所示，这是笔者的小经验。

代码清单 4-5 查看线程状况的 JSP 页面

```
<%@ page import="java.util.Map"%>

<html>
<head>
<title> 服务器线程信息 </title>
</head>
<body>
<pre>
<%
    for (Map.Entry<Thread, StackTraceElement[]> stackTrace : Thread.
getAllStackTraces().entrySet()) {
        Thread thread = (Thread) stackTrace.getKey();
        StackTraceElement[] stack = (StackTraceElement[]) stackTrace.getValue();
        if (thread.equals(Thread.currentThread())) {
            continue;
        }
        out.print("\n 线程: " + thread.getName() + "\n");
        for (StackTraceElement element : stack) {
            out.print("\t"+element+"\n");
        }
    }
%>
</pre>
</body>
</html>
```

4.3 JDK 的可视化工具

JDK 中除了提供大量的命令行工具外，还有两个功能强大的可视化工具：JConsole 和 VisualVM，这两个工具是 JDK 的正式成员，没有被贴上“unsupported and experimental”的标签。

其中 JConsole 是在 JDK 1.5 时期就已经提供的虚拟机监控工具，而 VisualVM 在 JDK 1.6 Update7 中才首次发布，现在已经成为 Sun (Oracle) 主力推动的多合一故障处理工具^①，并且已经从 JDK 中分离出来成为可以独立发展的开源项目。

^① VisualVM 官方站点：<https://visualvm.dev.java.net/>。

为了避免本节的讲解成为对软件说明文档的简单翻译，笔者准备了一些代码样例，都是笔者特意编写的反面教材。后面将会使用两款工具去监控和分析这几段代码存在的问题，算是本节简单的实战分析。读者可以把在可视化工具上观察到的数据和现象与前面两章中讲解的理论知识互相印证。

4.3.1 JConsole : Java 监视与管理控制台

JConsole (Java Monitoring and Management Console) 是一款基于 JMX 的可视化监视和管理的工具。它管理部分的功能是针对 JMX MBean 进行管理，由于 MBean 可以使用代码、中间件服务器的管理控制台或者所有符合 JMX 规范的软件进行访问，所以本节中将会着重介绍 JConsole 监视部分的功能。

1. 启动 JConsole

通过 JDK/bin 目录下的“jconsole.exe”启动 JConsole 后，将自动搜索出本机运行的所有虚拟机进程，不需要用户自己再使用 jps 来查询了，如图 4-4 所示。双击选择其中一个进程即可开始监控，也可以使用下面的“远程进程”功能来连接远程服务器，对远程虚拟机进行监控。

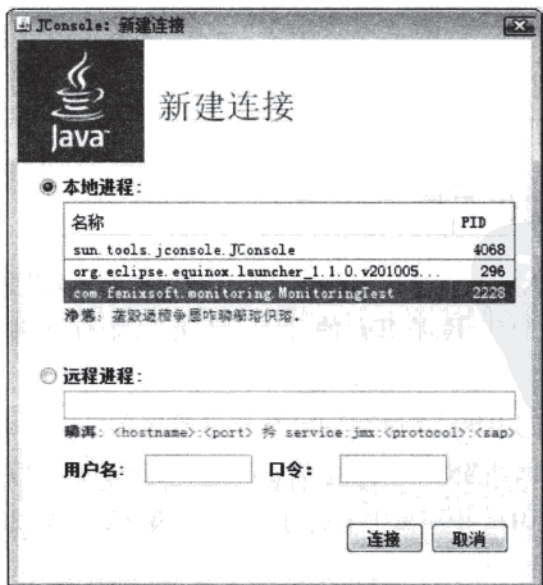


图 4-4 JConsole 连接页面

从图 4-4 中可以看到笔者的机器现在运行了 Eclipse、JConsole 和 MonitoringTest 一共三个本地虚拟机进程，其中 MonitoringTest 就是笔者准备的“反面教材”代码之一。双击它进入 JConsole 主界面，可以看到主界面里共包括“概述”、“内存”、“线程”、“类”、“VM 摘要”和“MBean”六个页签，如图 4-5 所示。

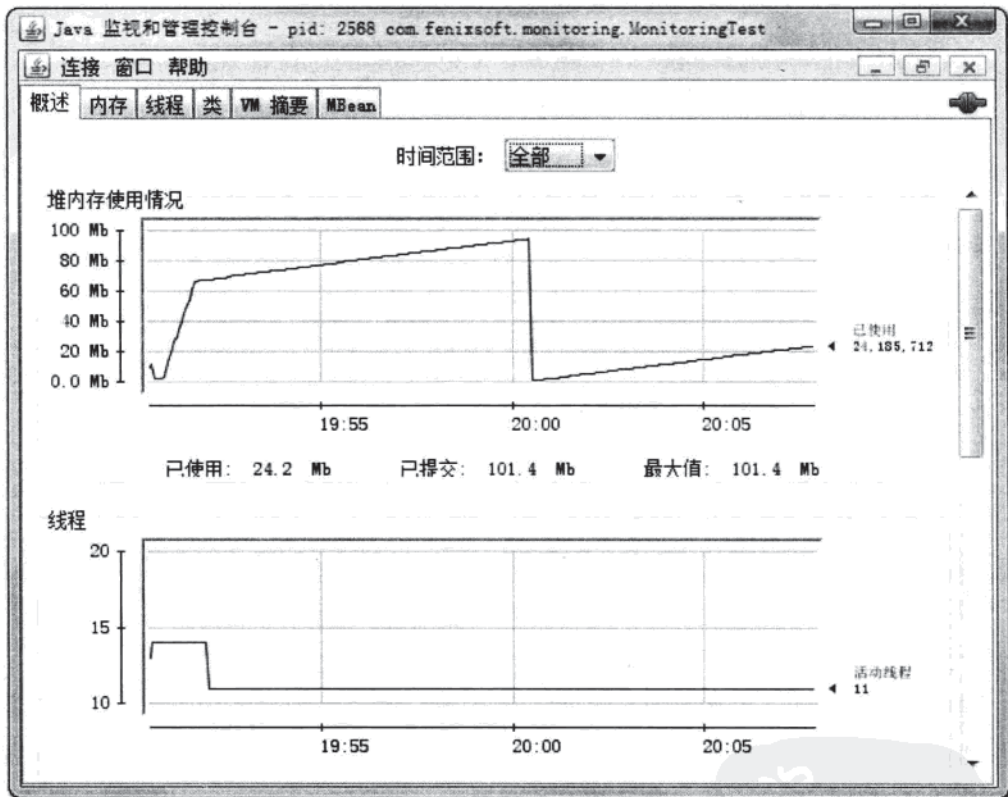


图 4-5 JConsole 主界面

“概述”页签显示的是整个虚拟机主要运行数据的概览，其中包括“堆内存使用情况”、“线程”、“类”、“CPU 使用情况”四项信息的曲线图，这些曲线图是后面“内存”、“线程”、“类”页签的信息汇总，具体内容将在后面介绍。

2. 内存监控

“内存”页签相当于可视化的 jstat 命令，用于监视受收集器管理的虚拟机内存（Java 堆和永久代）的变化趋势。我们通过运行代码清单 4-6 中的代码来体验一下它的

90 ❖ 第二部分 自动内存管理机制

监视功能。运行时设置的虚拟机参数为：-Xms100m -Xmx100m -XX:+UseSerialGC，这段代码的作用是以 64KB/50 毫秒的速度往 Java 堆中填充数据，一共填充 1000 次，使用 JConsole 的“内存”页签进行监视，观察曲线和柱状指示图的变化。

代码清单 4-6 JConsole 监视代码

```
/**
 * 内存占位符对象，一个 OOMObject 大约占 64K
 */
static class OOMObject {
    public byte[] placeholder = new byte[64 * 1024];
}

public static void fillHeap(int num) throws InterruptedException {
    List<OOMObject> list = new ArrayList<OOMObject>();
    for (int i = 0; i < num; i++) {
        // 稍作延时，令监视曲线的变化更加明显
        Thread.sleep(50);
        list.add(new OOMObject());
    }
    System.gc();
}

public static void main(String[] args) throws Exception {
    fillHeap(1000);
}
```

程序运行后，在“内存”页签中可以看到内存池 Eden 区的运行趋势呈现折线状，如图 4-6 所示。而监视范围扩大至整个堆后，会发现曲线是一条向上增长的平滑曲线。并且从柱状图可以看到，在 1000 次循环执行结束，运行了 System.gc() 后，虽然整个新生代 Eden 和 Survivor 区都基本被清空了，但是代表老年代的柱状图仍然保持峰值状态，说明被填充进堆中的数据在 System.gc() 方法执行之后仍然存活着。笔者的分析就到此为止，提两个小问题供读者思考一下，答案在下一页。

1) 虚拟机启动参数只限制了 Java 堆为 100MB，没有指定 -Xmn 参数，能否从监控图中估计出新生代有多大？

2) 为何执行了 System.gc() 之后，图 4-6 中代表老年代的柱状图仍然显示为峰值状态，代码需要如何调整才能让 System.gc() 回收掉填充到堆中的对象？

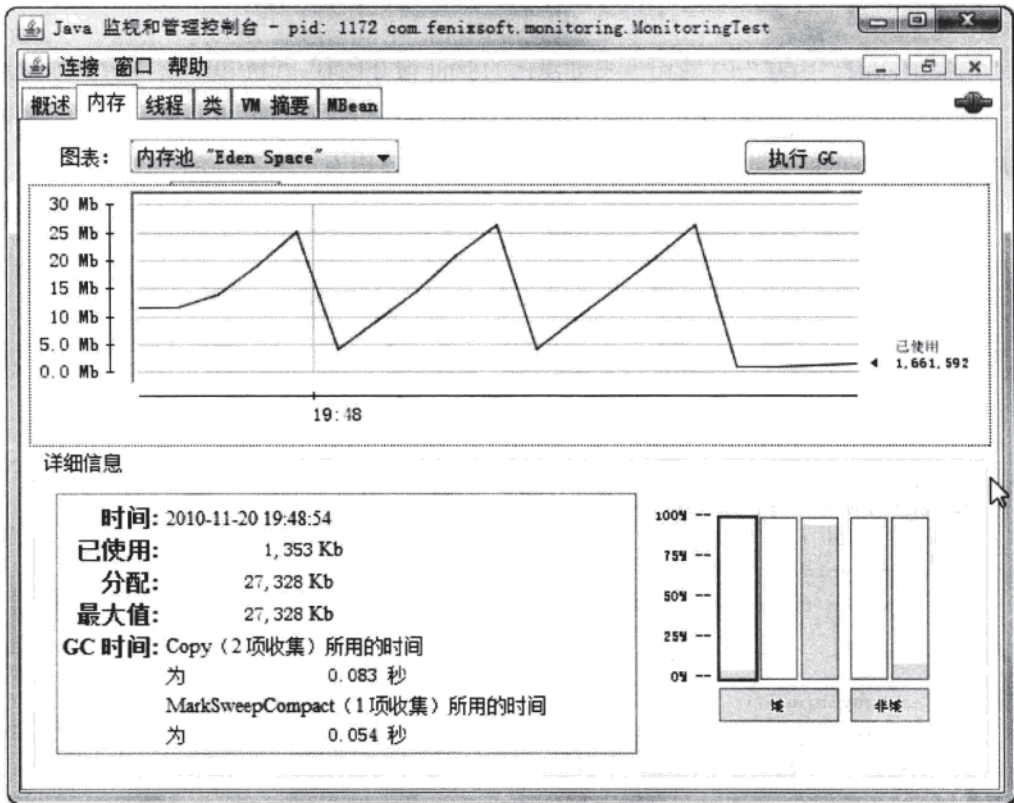


图 4-6 Eden 区内内存变化状况

问题 1 的答案：图 4-6 显示 Eden 空间为 27 328KB，因为没有设置 `-XX:SurvivorRatio` 参数，所以 Eden 与 Survivor 空间比例为默认值 8 : 1，因此整个新生代空间大约为 $27,328\text{KB} \times 125\% = 34,160\text{KB}$ 。

问题 2 的答案：System.gc() 之后，空间未能回收是因为 List<OOMObject> list 对象仍然存活着，fillHeap() 方法仍然没有退出，因此 list 对象在执行 System.gc() 时仍然处于作用域之内^①。如果把 System.gc() 移动到 fillHeap() 方法外，调用就可以回收掉全部内存。

① 准确地说，只有虚拟机使用解释器执行的时候，“在作用域之内”才能保证它不会被回收，因为这里的回收还涉及局部变量表 Slot 复用、即时编译器介入时机等问题，具体内容可参考第 8 章中关于局部变量表内存回收的例子。

3. 线程监控

如果上面的“内存”页签相当于可视化的jstat命令的话，“线程”页签的功能则相当于可视化的jstack命令，遇到线程停顿的时候可以使用这个页签进行监控分析。前面讲解jstack命令的时候提到过线程长时间停顿的主要原因有：等待外部资源（数据库连接、网络资源、设备资源等）、死循环、锁等待（活锁和死锁）。通过代码清单4-7分别演示一下这几种情况。

代码清单 4-7 线程等待演示代码

```
/**
 * 线程死循环演示
 */
public static void createBusyThread() {
    Thread thread = new Thread(new Runnable() {
        @Override
        public void run() {
            while (true) // 第41行
                ;
        }
    }, "testBusyThread");
    thread.start();
}

/**
 * 线程锁等待演示
 */
public static void createLockThread(final Object lock) {
    Thread thread = new Thread(new Runnable() {
        @Override
        public void run() {
            synchronized (lock) {
                try {
                    lock.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }, "testLockThread");
    thread.start();
}
```



```
}  
  
public static void main(String[] args) throws Exception {  
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));  
    br.readLine();  
    createBusyThread();  
    r.readLine();  
    Object obj = new Object();  
    createLockThread(obj);  
}
```

程序运行后，首先在“线程”页签中选择 main 线程，如图 4-7 所示。堆栈追踪显示 BufferedReader 在 readBytes 方法中等待 System.in 的键盘输入，这时候线程为 Runnable 状态，Runnable 状态的进程会被分配运行时间，但 readBytes 方法检查到流没有更新时会立刻归还执行令牌，这种等待只消耗很小的 CPU 资源。

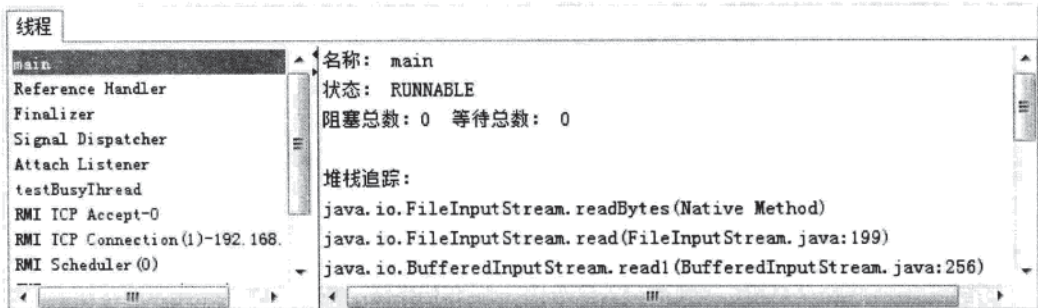


图 4-7 main 线程

接着监控 testBusyThread 线程，如图 4-8 所示，testBusyThread 线程一直在执行空循环，从堆栈追踪中可以看到一直停留在 MonitoringTest.java 代码的 41 行，41 行为 while (true)。这时候线程为 Runnable 状态，而且没有归还线程执行令牌的动作，会在空循环上用尽全部执行时间直到线程切换，这种等待会消耗较多的 CPU 资源。

图 4-9 显示 testLockThread 线程在等待着 lock 对象的 notify 或 notifyAll 方法的出现，线程这时候处在 WAITING 状态，在被唤醒前不会被分配执行时间。

testLockThread 线程正处于正常的活锁等待状态，只要 lock 对象的 notify() 或 notifyAll() 方法被调用，这个线程便能激活以继续执行。代码清单 4-8 演示了一个无法再被激活的死锁等待。

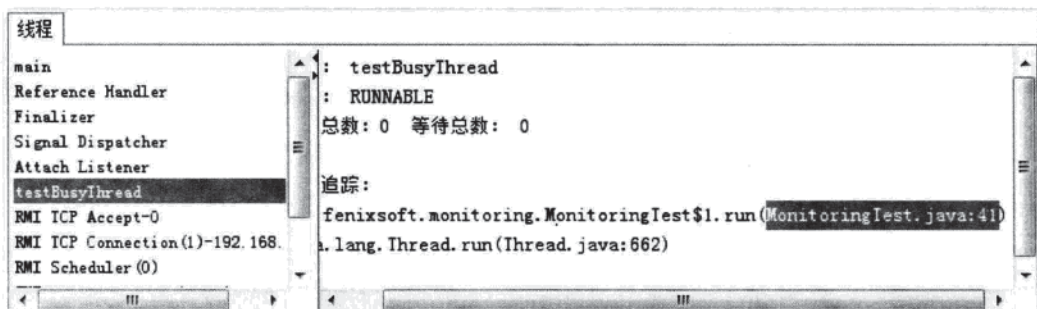


图 4-8 testBusyThread 线程

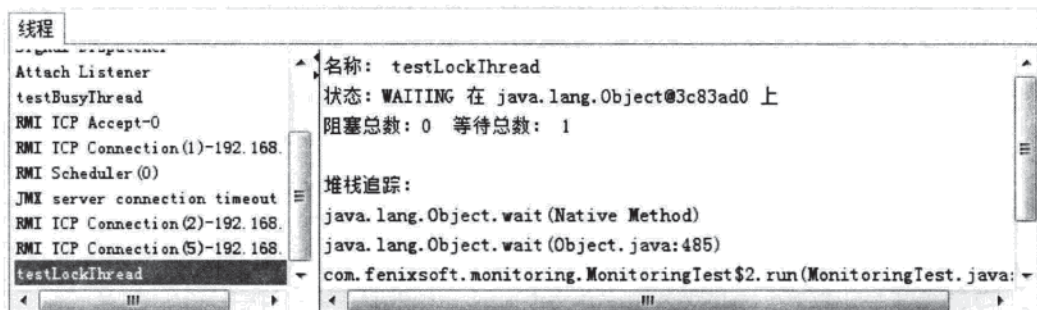


图 4-9 testLockThread 线程

代码清单 4-8 死锁代码样例

```

/**
 * 线程死锁等待演示
 */
static class SynAddRunnable implements Runnable {
    int a, b;
    public SynAddRunnable(int a, int b) {
        this.a = a;
        this.b = b;
    }

    @Override
    public void run() {
        synchronized (Integer.valueOf(a)) {
            synchronized (Integer.valueOf(b)) {
                System.out.println(a + b);
            }
        }
    }
}

```



```

    }
    }
}

public static void main(String[] args) {
    for (int i = 0; i < 100; i++) {
        new Thread(new SynAddRunalbe(1, 2)).start();
        new Thread(new SynAddRunalbe(2, 1)).start();
    }
}
}

```

这段代码开了 200 个线程分别去计算 1+2 及 2+1 的值，其实 for 循环是可省略的，两个线程也可能会导致死锁，不过那样概率太小，需要尝试运行很多次才能看到效果。如果运气不是特别差的话，带 for 循环的版本最多运行 2~3 次就会遇到线程死锁，程序无法结束。造成死锁的原因是 Integer.valueOf() 方法基于减少对象创建次数和节省内存的考虑，[-128,127] 之间的数字会被缓存^①，当 valueOf() 方法在这个范围之内传入参数，将直接返回缓存中的对象。也就是说代码中调用了 200 次 Integer.valueOf() 方法一共就只返回了两个不同的对象。假如在某个线程的两个 synchronized 块之间发生了一次线程切换，就会出现线程 A 等着被线程 B 持有的 Integer.valueOf(1)，线程 B 又等着被线程 A 持有的 Integer.valueOf(2)，结果大家都跑不下去的情景。

出现线程死锁之后，点击 JConsole 线程面板的“检测到死锁”的按钮，将出现一个新的“死锁”页签，如图 4-10 所示。



图 4-10 线程死锁

① 默认值，实际值取决于 java.lang.Integer.IntegerCache.high 参数的设置。

图中很清晰地显示线程 Thread-43 在等待一个被线程 Thread-12 持有 Integer 对象，而点击线程 Thread-12 则显示它也在等待一个 Integer 对象，被线程 Thread-43 持有，这样两个线程就互相卡住，都不存在等到锁释放的希望了。

4.3.2 VisualVM：多合一故障处理工具

VisualVM (All-in-One Java Troubleshooting Tool) 是到目前为止，随 JDK 发布的功能最强大的运行监视和故障处理程序，并且可以预见在未来一段时间内都是官方主力发展的虚拟机故障处理工具。官方在 VisualVM 的软件说明中写上了“*All-in-One*”的描述字样，预示着它除了运行监视、故障处理外，还提供了很多其他方面的功能。如性能分析 (Profiling)，VisualVM 的性能分析功能甚至比起 JProfiler、YourKit 等专业且收费的 Profiling 工具都不会逊色多少，而且 VisualVM 的还有一个很大优点：不需要被监视的程序基于特殊 Agent 运行，因此它对应用程序的实际性能的影响很小，使得它可以直接应用在生产环境中。这个优点是 JProfiler、YourKit 等工具无法与之媲美的。

1. VisualVM 兼容范围与插件安装

VisualVM 基于 NetBeans 平台开发，因此它一开始就具备了插件扩展功能的特性，通过插件扩展支持，VisualVM 可以做到：

- 显示虚拟机进程及进程的配置和环境信息 (jps、jinfo)。
- 监视应用程序的 CPU、GC、堆、方法区及线程的信息 (jstat、jstack)。
- dump 及分析堆转储快照 (jmap、jhat)。
- 方法级的程序运行性能分析，找出被调用最多、运行时间最长的方法。
- 离线程序快照：收集程序的运行时配置、线程 dump、内存 dump 等信息建立一个快照，可以将快照发送开发者处进行 Bug 反馈。
- 其他 plugins 的无限的可能性……

VisualVM 在 JDK 1.6 update 7 中才首次出现，但并不意味着它只能监控运行于 JDK 1.6 上的程序，它具备很强的向下兼容能力，甚至能向下兼容至近 10 年前发布的 JDK 1.4.2 平台^①，这对无数已经处于实施、维护状态的项目很有意义。当然，并非所有的功能都能完美地向下兼容，主要特性的兼容性如表 4-6 所示。

^① 早于 JDK1.6 的平台，需要打开 `-Dcom.sun.management.jmxremote` 参数才能被 VisualVM 管理。

表 4-6 VisualVM 主要功能兼容性列表

特 性	JDK 1.4.2	JDK 1.5	JDK 1.6 local	JDK 1.6 remote
运行环境信息	√	√	√	√
系统属性			√	
监视面板	√	√	√	√
线程面板		√	√	√
性能监控			√	
堆、线程 Dump			√	
MBean 管理		√	√	√
JConsole 插件		√	√	√

首次启动 VisualVM 后，读者先不必着急找应用程序进行监测，因为现在 VisualVM 还没有加载任何插件，虽然基本的监视、线程面板的功能主程序都以默认插件的形式提供了，但是不给 VisualVM 装任何扩展插件，就相当于放弃了它最精华的功能，和没有安装任何应用软件操作系统差不多。

插件可以手工安装，在网站^①上下载 *.nbm 包后，点击“工具”→“插件”→“已下载”菜单，然后在弹出的对话框中指定 nbm 包路径便可进行安装，插件安装后存放在 `JDK_HOME/lib/visualvm/visualvm` 中。不过手工安装并不常用，使用 VisualVM 的自动安装功能已经可以找到所需的大多数插件，在有网络连接的环境下，点击“工具”→“插件”菜单，弹出如图 4-11 所示的插件页签，在页签的“可用插件”中列举了当前版本 VisualVM 可以使用的插件，选中插件后在右边窗口将显示这个插件的基本信息，如开发者、版本、功能描述等。

大家可以根据自己的工作需求和兴趣选择合适的插件，然后点击安装按钮，弹出如图 4-12 所示的下载进度窗口，跟着提示操作稍后即可安装完成。

安装完插件后，选择一个需要监视的程序就进入程序的主界面了，如图 4-13 所示。根据读者选择安装插件数量的不同，看到的页签可能和笔者截图中的会有差异。

VisualVM 中“概述”、“监视”、“线程”、“MBeans”的功能与前面介绍的 JConsole 差别不大，读者根据上一节的内容类比使用即可，下面挑选几个特色功能和插件进行介绍。

① 插件中心地址：<http://java.net/projects/visualvm/content/pluginscenters.html>。

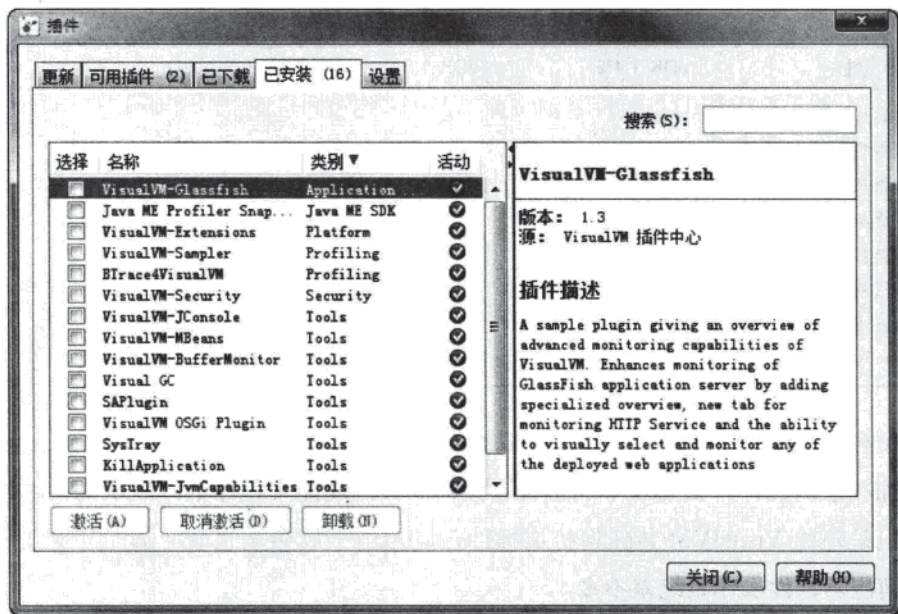


图 4-11 VisualVM 插件网页

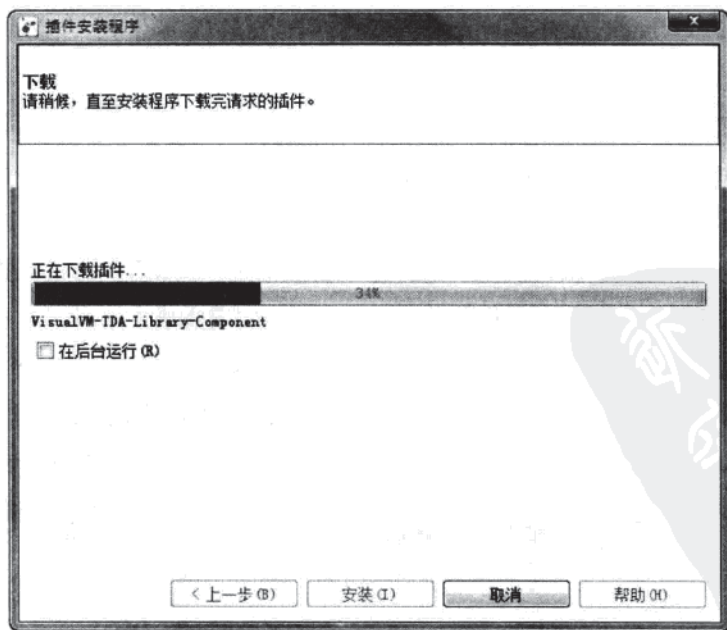


图 4-12 VisualVM 插件安装过程

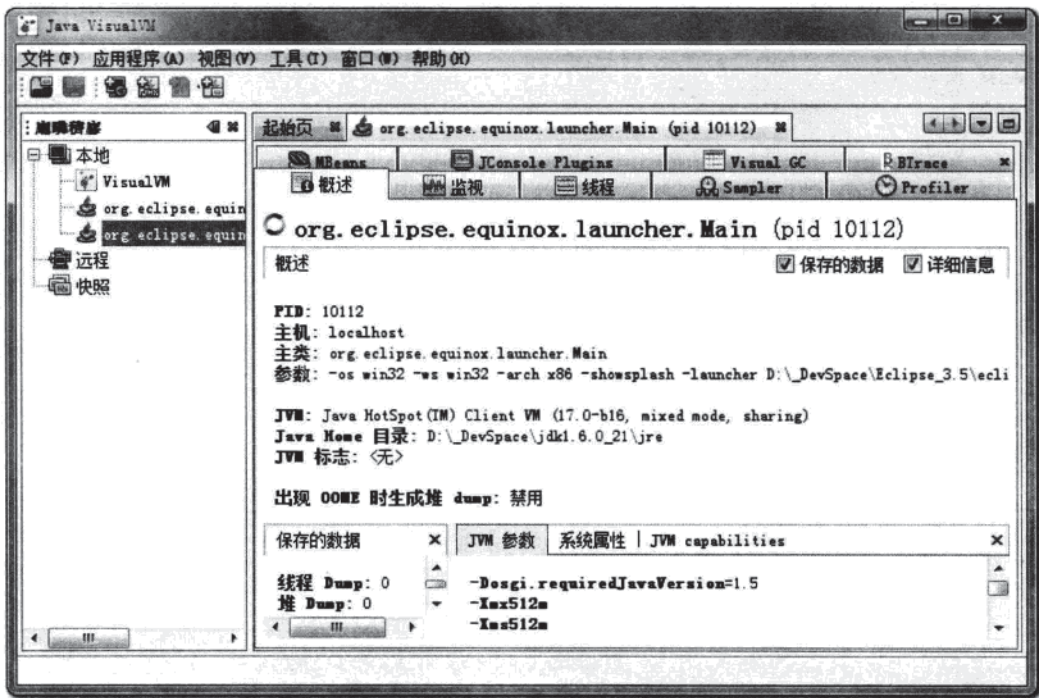


图 4-13 VisualVM 主界面

2. 生成和浏览堆转储快照

在 VisualVM 中生成 dump 文件有两种方式，可以执行下列任一操作：

- 在“应用程序”窗口中右键单击应用程序节点，然后选择“堆 Dump”。
- 在“应用程序”窗口中双击应用程序节点以打开应用程序标签，然后在“监视”标签中单击“堆 Dump”。

生成了 dump 文件之后，应用程序页签将在该堆的应用程序下增加一个以 [heapdump] 开头的子节点，并且在主页签中打开该转储快照，如图 4-14 所示。如果要把 dump 文件保存或发送出去，要在 heapdump 节点上右键选择“另存为”菜单，否则当 VisualVM 关闭时，生成的 dump 文件会被当做临时文件被删除掉。要打开一个已经存在的 dump 文件，通过文件菜单中的“装入”功能，选择硬盘上的 dump 文件即可。

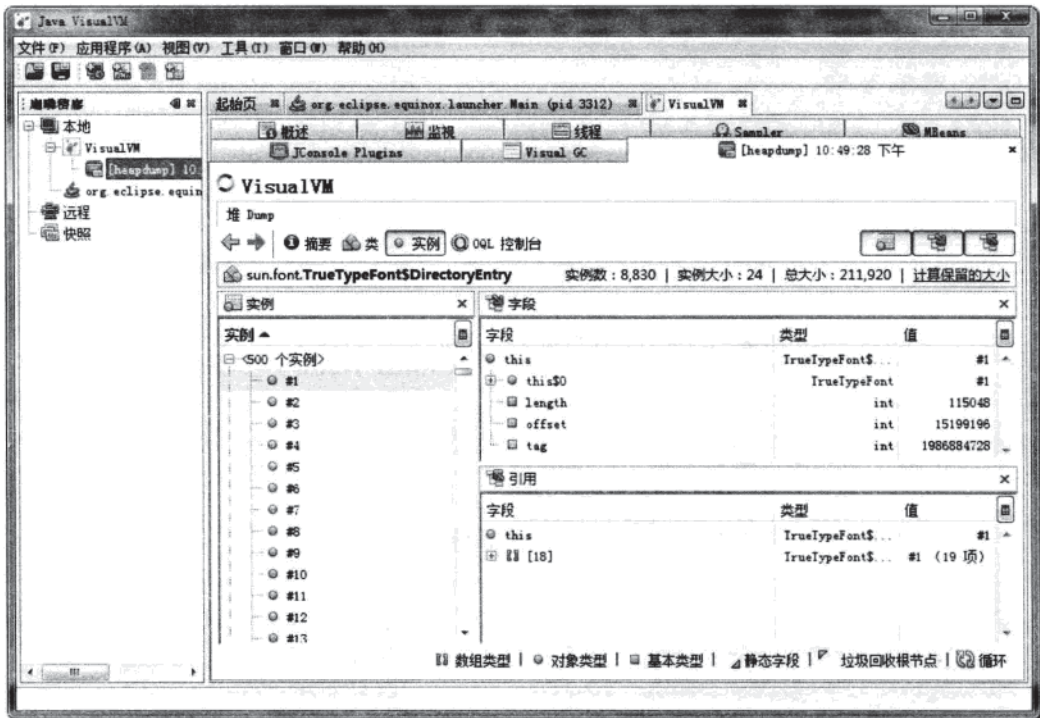


图 4-14 浏览 dump 文件

从堆页签中的“摘要”面板可以看到应用程序 dump 时的运行时参数、System.getProperties() 的内容、线程堆栈等信息，“类”面板则是以类为统计口径统计类的实例数量和容量信息，“实例”面板不能直接使用，因为不能确定用户想查看哪个类的实例，所以需要通过“类”面板进入，在“类”中选择一个关心的类后双击鼠标，即可在“实例”中看见此类中 500 个实例的具体属性信息。“OQL 控制台”面板里面就是运行 OQL 查询语句的，同 jhat 里面介绍的 OQL 功能一样。如果需要了解具体 OQL 语法和使用，可参见本书附录 D 的内容。

3. 分析程序性能

在 Profiler 页签中，VisualVM 提供了程序运行期间方法级的 CPU 执行时间分析及内存分析，进行 Profiling 分析肯定会对程序运行性能有比较大的影响，所以一般不在生产环境中使用这项功能。

要开始分析，先选择“CPU”和“内存”按钮中的一个，然后切换到应用程序中对

程序进行操作，VisualVM 会记录到这段时间中应用程序执行过的方法。如果是 CPU 分析，将会统计每个方法的执行次数、执行耗时；如果是内存分析则会统计每个方法关联的对象数及这些对象所占的空间。分析结束后，点击“停止”按钮结束监控过程，如图 4-15 所示。

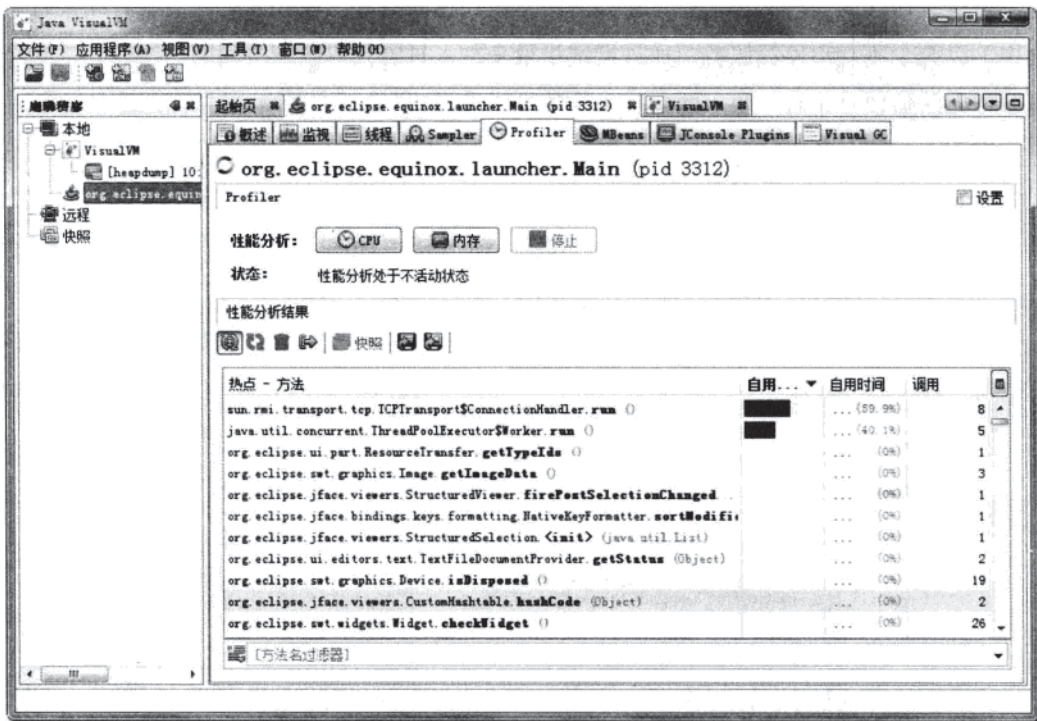


图 4-15 对应用程序进行 CPU 执行时间分析

注意 在 JDK 1.5 之后，在 Client 模式下的虚拟机加入并且自动开启了类共享——这是一个在多虚拟机进程中共享 rt.jar 中的类数据以提高加载速度和节省内存的优化，而根据相关 Bug 报告的反映，VisualVM 的 Profiler 功能可能会因为类共享而导致被监视的应用程序崩溃，所以读者进行 Profiling 前，最好在被监视的程序中使用 `-Xshare:off` 参数来关闭类共享优化。

图 4-15 中是对 Eclipse IDE 一段操作的录制和分析结果，读者分析自己的应用程序时，可以根据实际业务的复杂程度与方法的时间和调用次数做比较，找到最有优化价值的方法。

4. BTrace 动态日志跟踪

BTrace^①是一个很“有趣”的 VisualVM 插件，本身也是可以独立运行的程序。它的作用是在不停止目标程序运行的前提下，通过 HotSpot 虚拟机的 HotSwap 技术^②动态加入原本并不存在的调试代码。这项功能对实际生产中的程序很有意义：经常遇到程序出现问题，但排查错误的一些必要信息，譬如方法参数、返回值等，在开发时并没有打印到日志之中，以至于不得不停掉服务，通过调试增量来加入日志代码以解决问题。当遇到生产环境服务无法随便停止时，缺一两句日志导致排错进行不下去是一件非常郁闷的事情。

在 VisualVM 中安装了 BTrace 插件后，应用程序面板右键点击要调试的程序，会出现“Trace Application”菜单，点击将进入 BTrace 面板。这个面板里面看起来就像一个简单的 Java 程序开发环境，里面还有一小段 Java 代码，如图 4-16 所示。

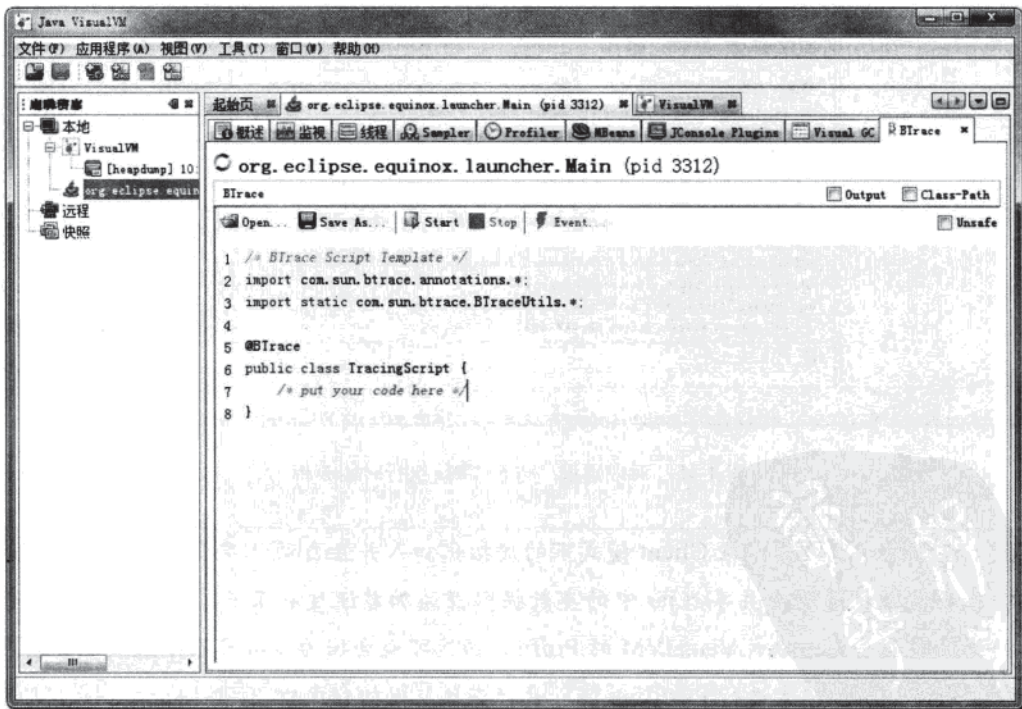


图 4-16 BTrace 动态跟踪

① 官方主页：<http://kenai.com/projects/btrace/>。

② HotSwap 技术：代码热替换技术，HotSpot 虚拟机允许在不停止运行的情况下，更新已经加载的类的代码。

笔者准备了一段很简单的 Java 代码来演示 BTrace 的功能：产生两个 1000 以内的随机整数，输出这 2 个数字相加的结果，如代码清单 4-9 所示。

代码清单 4-9 BTrace 跟踪演示

```
public class BTraceTest {

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

    public static void main(String[] args) throws IOException {
        BTraceTest test = new BTraceTest();
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        for (int i = 0; i < 10; i++) {
            reader.readLine();
            int a = (int) Math.round(Math.random() * 1000);
            int b = (int) Math.round(Math.random() * 1000);
            System.out.println(test.add(a, b));
        }
    }
}
```

程序运行后，在 VisualVM 中打开该程序的监视，在 BTrace 页签填充 TracingScript 的内容，输入调试代码如代码清单 4-10 所示。

代码清单 4-10 BTrace 调试代码

```
/* BTrace Script Template */
import com.sun.btrace.annotations.*;
import static com.sun.btrace.BTraceUtils.*;

@BTrace
public class TracingScript {
    @OnMethod(
        clazz="org.fenixsoft.monitoring.BTraceTest",
        method="add",
        location=@Location(Kind.RETURN)
    )

    public static void func(@Self org.fenixsoft.monitoring.BTraceTest
instance,int a,int b,@Return int result) {
```



```

println(" 调用堆栈 :");
jstack();
println(strcat(" 方法参数 A:",str(a)));
println(strcat(" 方法参数 B:",str(b)));
println(strcat(" 方法结果 :",str(result)));
}
}

```

点击“Start”按钮后稍等片刻，编译完成后，可见 Output 面板中出现“BTrace code successfully deployed”的字样。程序运行的时候在 Output 面板将会输出如图 4-17 所示的调试信息。

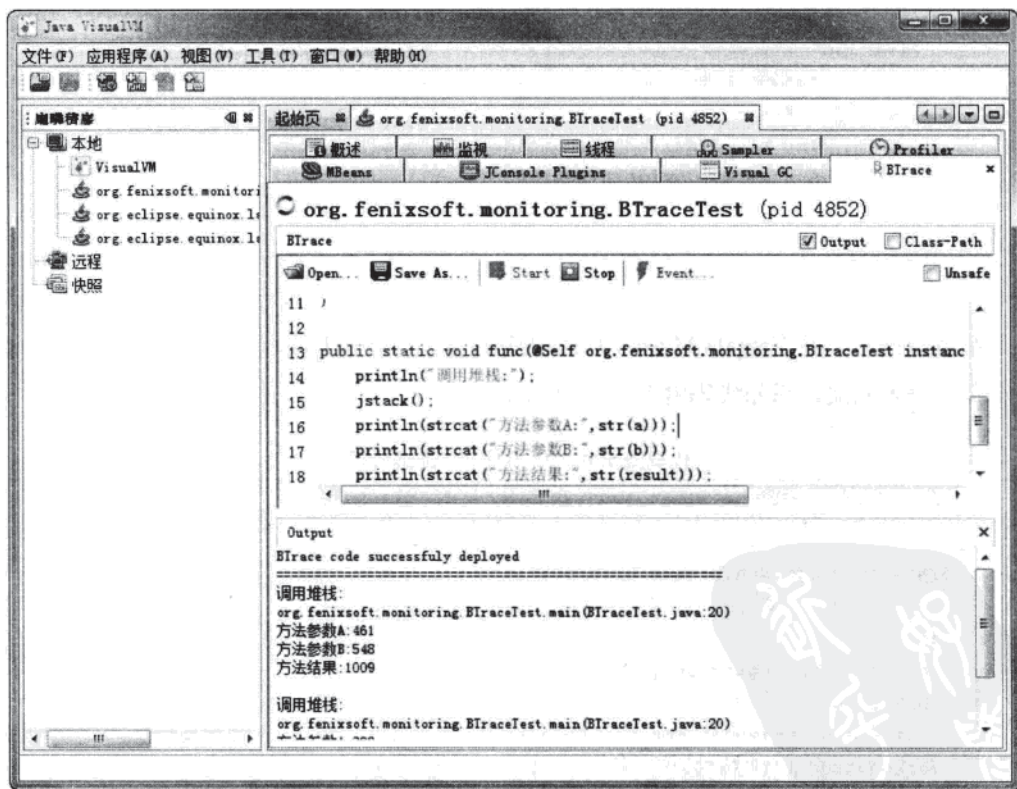


图 4-17 BTrace 跟踪结果

BTrace 的用法还有许多，打印调用堆栈、参数、返回值只是最基本的应用，在它的网站上有使用 BTrace 进行性能监视、定位连接泄漏、内存泄漏、解决多线程竞争问题等

的使用例子，有兴趣的读者可以去网上了解相关信息。

4.4 本章小结

本章介绍了随 JDK 发布的 6 个命令行工具与 2 个可视化的故障处理工具，灵活使用这些工具，可以给处理问题带来很大的便利。

除了 JDK 自带的工具之外，常用的故障处理工具还有很多，如果读者使用的是非 Sun 系列的 JDK，非 HotSpot 的虚拟机，就需要使用对应的工具进行分析，如：

- IBM 的 Support Assistant^①、Heap Analyzer^②、Javacore Analyzer^③、Garbage Collector Analyzer^④适用于 IBM J9 VM。
- HP 的 HPjmeter^⑤、HPjtune 适用于 HP-UX、SAP、HotSpot VM。
- Eclipse 的 Memory Analyzer Tool^⑥ (MAT) 适用于 HP-UX、SAP、HotSpot VM，安装 IBM DTFJ^⑦ 插件后可支持 IBM J9 VM。
- BEA 的 JRockit Mission Control^⑧，适用于 JRockit VM。

① <http://www-01.ibm.com/software/support/isa/>。

② <http://www.alphaworks.ibm.com/tech/hepanalyzer/download>。

③ <http://www.alphaworks.ibm.com/tech/jca/download>。

④ <http://www.alphaworks.ibm.com/tech/pmat/download>。

⑤ <https://h20392.www2.hp.com/portal/swdepot/displayProductInfo.do?productNumber=HPJMETER>。

⑥ <http://www.eclipse.org/mat/>。

⑦ <http://www.ibm.com/developerworks/java/jdk/tools/dtj.html>。

⑧ http://download.oracle.com/docs/cd/E13150_01/jrockit_jvm/jrockit/tools/index.html。



第 5 章 调优案例分析与实战

本章主要内容

- 概述
- 案例分析
- 实战：Eclipse 运行速度调优

Java 与 C++ 之间有一堵由内存动态分配和垃圾收集技术所围成的高墙，墙外面的人想进去，墙里面的人却想出来。

5.1 概述

前面三章介绍了处理 Java 虚拟机内存问题的知识与工具，在处理实际项目的问题时，除了知识与工具外，经验也是一个很重要的因素。因此本章将与读者分享几个比较有代表性的实际案例。考虑到虚拟机故障处理和调优主要面向各类服务端应用，而大部分 Java 程序员较少有机会直接接触生产环境的服务器，因此本章还准备了一个所有开发人员都能够进行“亲身实战”的练习，希望通过实践使读者获得故障处理和调优的经验。

5.2 案例分析

本章中的案例大部分来源于笔者处理过的一些问题，还有一小部分来源于网上有特色和代表性的案例总结。出于对客户商业信息保护的目，在不影响前后逻辑的前提下，笔者对实际环境和用户业务做了一些屏蔽和精简。

5.2.1 高性能硬件上的程序部署策略

一个 15 万 PV/ 天左右的在线文档类型网站最近更换了硬件系统，新的硬件为 4 个

CPU、16GB 物理内存，操作系统为 64 位 CentOS 5.4，Resin 作为 Web 服务器。整个服务器暂时没有部署别的应用，所有硬件资源都可以提供给访问量并不算太大的网站使用。管理员为了尽量利用硬件资源选用了 64 位的 JDK 1.5，并通过 `-Xmx` 和 `-Xms` 参数将 Java 堆固定在 12GB。使用一段时间后发现使用效果并不理想，网站经常不定期出现长时间没有响应的现象。

监控服务器运行状况后发现网站没有响应是由 GC 停顿导致的，虚拟机运行在 Server 模式，默认使用吞吐量优先收集器，回收 12GB 的堆，一次 Full GC 的停顿时间高达 14 秒。并且由于程序设计的关系，访问文档时要把文档从磁盘提取到内存中，导致内存中出现很多由文档序列化产生的大对象，这些大对象很多都进入了老年代，没有在 Minor GC 中清理掉。这种情况下即使有 12GB 的堆，内存也很快会被消耗殆尽，由此导致每隔十几分钟出现十几秒的停顿，令网站开发人员和管理员感到很沮丧。

这里先不延伸讨论程序代码问题，程序部署上的主要问题显然是过大的堆内存进行回收时带来的长时间的停顿。硬件升级前使用 32 位系统 1.5GB 的堆，用户只感到访问网站比较缓慢，但不会发生十分明显的停顿，因此才考虑升级硬件提升程序效能，如果重新缩小给 Java 堆分配的内存，那么硬件上的投资就浪费了。

在高性能硬件上部署程序，目前主要有两种方式：

- 通过 64 位 JDK 来使用大内存。
- 使用若干个 32 位虚拟机建立逻辑集群来利用硬件资源。

此案例中的管理员采用了第一种部署方式。对于用户交互性强、对停顿时间敏感的系统，可以给 Java 虚拟机分配超大堆的前提是有把握把应用程序的 Full GC 频率控制得足够低，至少要低到不会影响用户使用，譬如十几个小时乃至一天才出现一次 Full GC，这样可以通过在深夜执行定时任务的方式触发 Full GC 甚至自动重启应用服务器来将内存可用空间保持在一个稳定的水平。

控制 Full GC 频率的关键是看应用中绝大多数对象是否符合“朝生夕灭”的原则，即大多数对象的生存时间不应当太长，尤其是不能产生成批量的、长生存时间的大对象，这样才能保障老年代空间的稳定。

在大多数网站形式的应用里，主要对象的生存周期都应该是请求级或页面级的，会话级和全局级的长生命对象相对很少。只要代码写得合理，应当都能实现在超大堆中正常使用而没有 Full GC，这样的话，使用超大堆内存时，网站响应的速度才比较有

保证。除此之外，如果读者计划使用 64 位 JDK 来管理大内存，还需要考虑下面可能面临的问题：

- 内存回收导致的长时间停顿。
- 现阶段，64 位 JDK 的性能测试结果普遍低于 32 位 JDK。
- 需要保证程序足够稳定，因为这种应用要是产生堆溢出几乎就无法产生堆转储快照（因为要产生十几 GB 乃至更大的 dump 文件），哪怕产生了快照也几乎无法进行分析。
- 相同的程序在 64 位 JDK 中消耗的内存一般比 32 位 JDK 大，这是由指针膨胀及数据类型对齐补白等因素导致的。

上面的问题听起来有点吓人，所以现阶段不少管理员还是选择第二种方式：使用若干个 32 位虚拟机建立逻辑集群来利用硬件资源。具体做法是在一台物理机器上启动多个应用服务器进程，给每个服务器进程分配不同的端口，然后在前端搭建一个负载均衡器，以反向代理的方式来分配访问请求。读者不需要太在意均衡器转发所消耗的性能，即使使用 64 位 JDK，许多应用也不止有一台服务器，因此在许多应用中前端的均衡器总是要存在的。

考虑到在一台物理机器上建立逻辑集群的目的仅仅是尽可能地利用硬件资源，并不需要关心状态保留、热转移之类的高可用性需求，也不需要保证每个虚拟机进程有绝对准确的均衡负载，因此使用无 Session 复制的亲合式集群是一个相当不错的选择。我们仅仅需要保障集群具备亲和性，也就是均衡器按一定的规则算法（一般根据 SessionID 分配）将一个固定的用户请求永远分配到固定的一个集群节点进行处理即可，这样程序开发阶段就基本不用为集群环境做什么特别的考虑。

当然，很少有没有缺点的方案，如果读者计划使用逻辑集群的方式来部署程序，可能会遇到下面一些问题：

- 尽量避免节点竞争全局的资源，最典型的的就是磁盘竞争，各个节点如果同时访问某个磁盘文件的话（尤其是并发写操作容易出现问题的），很容易导致 IO 异常。
- 很难最高效率地利用某些资源池，譬如连接池，一般都是在各个节点建立自己独立的连接池，这样有可能导致一些节点池满了而另外一些节点仍有较多空余。尽管可以使用集中式的 JNDI，但这有一定的复杂性并且可能带来额外的性能代价。

- 各个节点仍然不可避免地受到 32 位的内存限制，在 32 位 Windows 平台中每个进程只能使用 2GB 的内存，考虑到堆以外的内存开销，堆一般最多只能开到 1.5GB。在某些 Linux, Unix 系统（如 Solaris）中，可以提升到 3GB 乃至接近 4GB 的内存，但 32 位中仍然受最高 4GB (2^{32}) 内存的限制。
- 大量使用本地缓存（如大量使用 HashMap 作为 K/V 缓存）的应用，在逻辑集群中会造成较大的内存浪费，因为每个逻辑节点上都有一份缓存，这时可以考虑把本地缓存改为集中式缓存。

介绍完这两种部署方式，再重新回到这个案例之中，最后的部署方案调整为建立 5 个 32 位 JDK 的逻辑集群，每个进程按 2GB 内存计算（其中堆固定为 1.5GB），占用了 10GB 的内存。另外建立一个 Apache 服务作为前端均衡代理访问门户。考虑到用户对响应速度比较关心，并且文档服务的主要压力集中在磁盘和内存访问上，CPU 资源敏感度较低，因此改为 CMS 收集器进行垃圾回收。部署方式调整后，服务再没有出现长时间停顿，速度比硬件升级前有较大提升。

5.2.2 集群间同步导致的内存溢出

一个基于 B/S 的 MIS 系统，硬件为两台 2 个 CPU、8GB 内存的 HP 小型机，服务器是 WebLogic 9.2，每台机器启动了 3 个 WebLogic 实例，构成一个 6 个节点的亲合式集群。由于是亲合式集群，节点之间没有进行 Session 同步，但是有一些需求要实现部分数据在各个节点间共享。开始这些数据存放在数据库中，但由于读写频繁竞争很激烈，对性能的影响较大，后面使用 JBossCache 构建了一个全局缓存。全局缓存启用后，服务正常使用了较长的一段时间。但最近不定期地多次出现内存溢出问题。

在不出现内存溢出异常的时候，服务内存回收状况一直正常，每次内存回收后都能恢复到一个稳定的可用空间，开始怀疑是程序的某些不常用的代码路径中存在内存泄漏，但管理员反映最近程序并未更新或升级过，也没有进行什么特别的操作。只好让服务带着 `-XX:+HeapDumpOnOutOfMemoryError` 参数运行了一段时间。在最近一次溢出之后，管理员发回了 heapdump 文件，发现里面存在着大量的 `org.jgroups.protocols.pbcast.NAKACK` 对象。

JBossCache 是基于自家的 JGroups 进行集群间的数据通信，JGroups 使用协议栈的方式来实现收发数据包的各种所需特性的自由组合，数据包接收和发送时要经过每层

协议栈的 up() 和 down() 方法, 其中的 NAKACK 栈用于保障各个包的有效顺序及重发。JBossCache 协议栈如图 5-1 所示。

```

Daemon Thread [DownHandler (VIEW_SYNC)] (Suspended (breakpoint at line 401 in NAKACK))
  NAKACK.down(Event) line: 401
  NAKACK(Protocol).receiveDownEvent(Event) line: 517
  UNICAST(Protocol).passDown(Event) line: 551
  UNICAST.down(Event) line: 355
  UNICAST(Protocol).receiveDownEvent(Event) line: 517
  STABLE(Protocol).passDown(Event) line: 551
  STABLE.down(Event) line: 283
  STABLE(Protocol).receiveDownEvent(Event) line: 517
  FRAG(Protocol).passDown(Event) line: 551
  FRAG.down(Event) line: 139
  FRAG(Protocol).receiveDownEvent(Event) line: 517
  VIEW_SYNC(Protocol).passDown(Event) line: 551
  VIEW_SYNC.down(Event) line: 166
  DownHandler.run() line: 121
  
```

图 5-1 JBossCache 协议栈

由于信息有传输失败需要重发的可能性, 在确认所有注册在 GMS (Group Membership Service) 的节点都收到正确的信息前, 发送的信息必须在内存中保留。而此 MIS 的服务端中有一个负责安全校验的全局 Filter, 每当接收到请求时, 均会更新一次最后的操作时间, 并且将这个时间同步到所有的节点中, 使得一个用户在一段时间内不能在多台机器上登录。在服务使用过程中, 往往一个页面会产生数次乃至数十次的请求, 因此这个过滤器导致集群各个节点之间的网络交互非常频繁。当网络情况不能满足传输要求时, 重发数据在内存中不断地堆积, 很快就产生了内存溢出。

这个案例中的问题, 既有 JBossCache 的缺陷, 也有 MIS 系统实现方式上的缺陷。JBossCache 官方的 maillist 中讨论过很多次类似的内存溢出异常问题, 据说后续版本有了改进。而更重要的缺陷是这一类被集群共享的数据如果要使用类似 JBossCache 这种集群缓存来同步的话, 可以允许读操作频繁, 因为数据在本地内存有一份副本, 读取的动作不会耗费多少资源, 但不应当有过于频繁的写操作, 这会带来很大的网络同步的开销。

5.2.3 堆外内存导致的溢出错误

这是一个学校的小型项目: 基于 B/S 的电子考试系统, 为了实现客户端能实时地

从服务端接收考试数据，系统使用了逆向 AJAX 技术（也称为 Comet 或 Server Side Push），选用 CometD 1.1.1 作为服务端推送框架，服务器是 Jetty 7.1.4，硬件为一台普通 PC 机，Core i5 CPU，4GB 内存，运行 32 位 Windows 操作系统。

测试期间发现服务端不定时抛出内存溢出异常，服务器不一定每次都会出现异常，但假如正式考试时崩溃一次，那估计整场电子考试都会乱套，网站管理员尝试过把堆开到最大，32 位系统最多到 1.6GB 基本无法再加大了，而且开大了也基本没效果，抛出内存溢出异常好像更加频繁了。加入 `-XX:+HeapDumpOnOutOfMemoryError`，居然也没有任何反应，抛出内存溢出异常时什么文件都没有产生。无奈之下只好挂着 jstat 使劲盯屏幕，发现 GC 并不频繁，Eden 区、Survivor 区、老年代及永久代内存全部都表示“情绪稳定，压力不大”，但照样不停地抛出内存溢出异常，管理员压力很大。最后，在内存溢出后从系统日志中找到异常堆栈，如代码清单 5-1 所示。

代码清单 5-1 异常堆栈 1

```
[org.eclipse.jetty.util.log] handle failed java.lang.OutOfMemoryError: null
  at sun.misc.Unsafe.allocateMemory(Native Method)
  at java.nio.DirectByteBuffer.<init>(DirectByteBuffer.java:99)
  at java.nio.ByteBuffer.allocateDirect(ByteBuffer.java:288)
  at org.eclipse.jetty.io.nio.DirectNIOBuffer.<init>
  .....
```

如果认真阅读过本书的第 2 章，看到异常堆栈就应该清楚这个内存溢出异常是怎么回事了。大家知道操作系统对每个进程能管理的内存是有限制的，这台服务器使用的 32 位 Windows 平台的限制是 2GB，其中给了 Java 堆 1.6GB，而 Direct Memory 并不算在 1.6GB 的堆之内，因此它只能在剩余的 0.4GB 空间中分出一部分。在此应用中导致溢出的关键是：垃圾收集进行时，虚拟机虽然会对 Direct Memory 进行回收，但是 Direct Memory 却不能像新生代和老年代那样，发现空间不足了就通知收集器进行垃圾回收，它只能等待老年代满了后 Full GC，然后“顺便地”帮它清理掉内存的废弃对象。否则，它只能等到抛出内存溢出异常时，先 catch 掉，再在 catch 块里面“大喊”一声：“System.gc()！”。要是虚拟机还是不听（譬如打开了 `-XX:+DisableExplicitGC` 开关），那就只能眼睁睁地看着堆中还有许多空闲内存，自己却不得不抛出内存溢出异常了。而本案例中使用的 CometD 1.1.1 框架，正好有大量的 NIO 操作需要用到 Direct Memory。

112 ❖ 第二部分 自动内存管理机制

从实践经验的角度出发，除了 Java 堆和永久代之外，我们注意到下面这些区域还会占用较多的内存，这里所有的内存总和会受到操作系统进程最大内存的限制：

- Direct Memory：可通过 `-XX:MaxDirectMemorySize` 调整大小，内存不足时抛出 `OutOfMemoryError` 或 `OutOfMemoryError: Direct buffer memory`。
- 线程堆栈：可通过 `-Xss` 调整大小，内存不足时抛出 `StackOverflowError`（纵向无法分配，即无法分配新的栈帧）或 `OutOfMemoryError: unable to create new native thread`（横向无法分配，即无法建立新的线程）。
- Socket 缓存区：每个 Socket 连接都 Receive 和 Send 两个缓存区，分别占大约 37KB 和 25KB 的内存，连接多的话这块内存占用也比较可观。如果无法分配，则可能会抛出 `IOException: Too many open files` 异常。
- JNI 代码：如果代码中使用 JNI 调用本地库，那本地库使用的内存也不在堆中。
- 虚拟机和 GC：虚拟机和 GC 的代码执行也要消耗一定的内存。

5.2.4 外部命令导致系统缓慢

这是一个来自网络的案例：一个数字校园应用系统，运行在一台 4 个 CPU 的 Solaris 10 操作系统上，中间件为 GlassFish 服务器。系统在进行大并发压力测试的时候，发现请求响应时间比较慢，通过操作系统的 `mpstat` 工具发现 CPU 使用率很高，并且占用绝大多数 CPU 资源的程序并不是应用系统本身。这是个不正常的现象，通常情况下用户应用的 CPU 占用率应该占主要地位，才能说明系统是正常工作的。

通过 Solaris 10 的 `Dtrace` 脚本可以查看当前情况下哪些系统调用花费了最多的 CPU 资源，`Dtrace` 运行后发现最消耗 CPU 资源的竟然是“fork”系统调用。众所周知，“fork”系统调用是 Linux 用来产生新进程的，在 Java 虚拟机中，用户编写的 Java 代码最多只有线程的概念，不应当有进程的产生。

这是个非常异常的现象。通过本系统的开发人员最终找到了答案：每个用户请求的处理都需要执行一个外部 shell 脚本来获得系统的一些信息。执行这个 shell 脚本是通过 Java 的 `Runtime.getRuntime().exec()` 方法来调用的。这种调用方式可以达到目的，但是它在 Java 虚拟机中非常消耗资源，即使外部命令本身能很快执行完毕，频繁调用时创建进程的开销也非常可观。Java 虚拟机执行这个命令的过程是：首先克隆一个和当前虚拟机拥有一样环境变量的进程，再用这个新的进程去执行外部命令，最后再退出这个进

程。如果频繁执行这个操作，系统的消耗会很大，不仅是 CPU，内存的负担也很重。

用户根据建议去掉这个 shell 脚本执行的语句，改为使用 Java 的 API 去获取这些信息后，系统很快就恢复了正常。

5.2.5 服务器 JVM 进程崩溃

一个基于 B/S 的 MIS 系统，硬件为两台 2 个 CPU、8GB 内存的 HP 系统，服务器是 WebLogic 9.2（就是第二个案例中的那套系统）。正常运行一段时间后，最近发现在运行期间频繁出现集群节点的虚拟机进程自动关闭的现象，留下了一个 `hs_err_pid###.log` 文件后，进程就消失了，两台物理机器里的每个节点都出现过进程崩溃的现象。从系统日志中注意到，每个节点的虚拟机进程在崩溃前不久，都发生过大量相同的异常，见代码清单 5-2。

代码清单 5-2 异常堆栈 2

```
java.net.SocketException: Connection reset
at java.net.SocketInputStream.read(SocketInputStream.java:168)
at java.io.BufferedInputStream.fill(BufferedInputStream.java:218)
at java.io.BufferedInputStream.read(BufferedInputStream.java:235)
at org.apache.axis.transport.http.HTTPSender.readHeadersFromSocket(HTTPSender.
java:583)
at org.apache.axis.transport.http.HTTPSender.invoke(HTTPSender.java:143)
... 99 more
```

这是一个远端断开连接的异常，通过系统管理员了解到系统最近与一个 OA 门户做了集成，在 MIS 系统工作流的待办事项变化时，要通过 Web 服务通知 OA 门户系统，把待办事项的变化同步到 OA 门户之中。通过 SoapUI 测试了一下同步待办事项的几个 Web 服务，发现调用后竟然需要长达 3 分钟才能返回，并且返回的结果都是连接中断。

由于 MIS 系统的用户多，待办事项变化很快，为了不被 OA 系统的速度拖累，使用了异步的方式调用 Web 服务，但由于两边服务的速度完全不对等，时间越长就累积了越多 Web 服务没有调用完成，导致在等待的线程和 Socket 连接越来越多，最终超过虚拟机的承受能力后使得虚拟机进程崩溃。通知 OA 门户方修复无法使用的集成接口，并将异步调用改为生产者 / 消费者模式的消息队列实现后，系统恢复正常。

5.3 实战：Eclipse 运行速度调优

很多 Java 开发人员都有这样一种观念：系统调优的工作都是针对服务端应用而言的，规模越大的系统，需要越专业的调优运维团队参与。这个观点不能说不正确，上一节中笔者所列举的案例确实都是服务端运维和调优的例子，但服务端应用需要调优，并不说明其他应用就不需要了，作为一个普通的 Java 开发人员，前面讲的各种虚拟机的原理和最佳实践的方法距离我们并不遥远，开发者身边的很多场景都可以使用上面这些知识。下面就通过一个普通程序员日常工作中可以随时接触到的开发工具开始这次实战。

5.3.1 调优前的程序运行状态

笔者使用 Eclipse 3.5 作为日常工作中的主要 IDE 工具，由于安装的插件比较大（如 Klocwork、ClearCase LT 等）、代码也很多，启动 Eclipse 直到所有项目编译完成需要四五分钟。一直对开发环境的速度感到不满意，趁着编写这本书的机会，决定对 Eclipse 进行“动刀”调优。

笔者机器的 Eclipse 运行平台是 32 位 Windows 7 系统，虚拟机为 HotSpot VM 1.5 b64，硬件为 ThinkPad X201，Intel i5 CPU，4GB 物理内存。在初始的配置文件 eclipse.ini 中，除了指定 JDK 的路径、设置最大堆为 512MB 及开启了 JMX 管理（需要在 VisualVM 中收集原始数据）外，未作任何改动，原始配置内容如代码清单 5-3 所示。

代码清单 5-3 Eclipse 3.5 初始配置

```
-vm
D:/_DevSpace/jdk1.5.0/bin/javaw.exe
-startup
plugins/org.eclipse.equinox.launcher_1.0.201.R35x_v20090715.jar
--launcher.library
plugins/org.eclipse.equinox.launcher.win32.win32.x86_1.0.200.v20090519
-product
org.eclipse.epp.package.jee.product
--launcher.XXMaxPermSize
256M
-showsplash
org.eclipse.platform
-vmargs
-Dosgi.requiredJavaVersion=1.5
```



```
-Xmx512m  
-Dcom.sun.management.jmxremote
```

为了与调优后的结果进行量化对比，调优开始前笔者先做了一次初始数据测试。测试用例很简单，就是收集从 Eclipse 启动开始，直到所有插件加载完成为止的总耗时及运行状态数据，虚拟机的运行数据通过 VisualVM 及其扩展插件 VisualGC 进行采集。测试过程中反复启动 Eclipse 数次直到测试结果稳定后，取最后一次运行的结果作为数据样本（为了避免操作系统未能及时进行磁盘缓存而产生的影响），数据样本如图 5-2 所示。

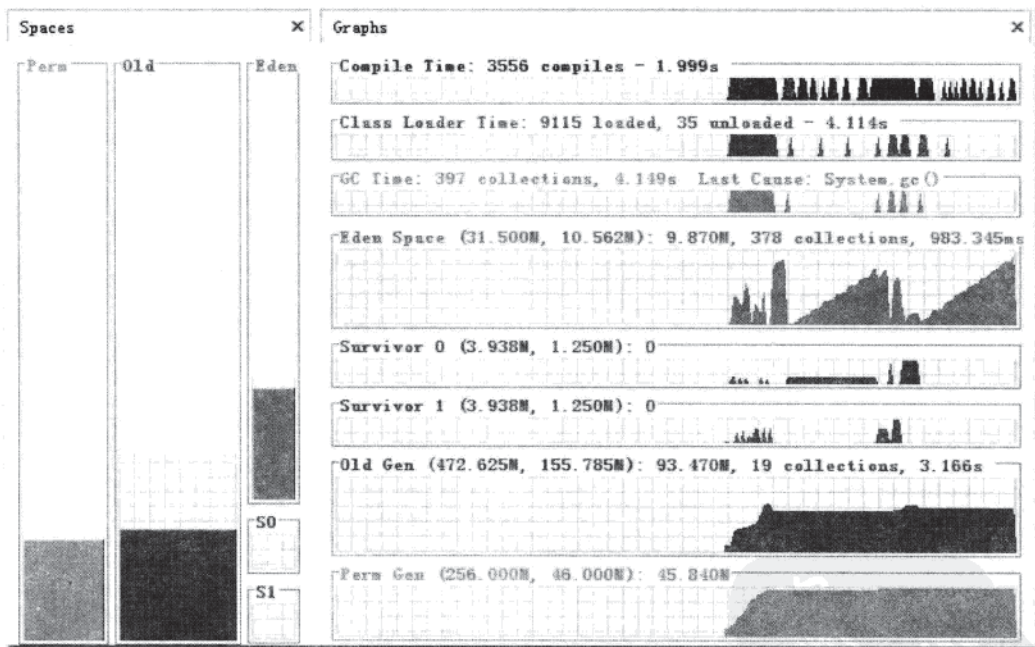


图 5-2 Eclipse 原始运行数据

Eclipse 启动的总耗时没有办法从监控工具中直接获得，因为 VisualVM 不可能知道 Eclipse 运行到什么阶段才算是启动完成。为了保证测试的准确性，笔者写了一个简单的 Eclipse 插件，用于统计 Eclipse 的启动耗时。由于代码很简单，并且本书不是 Eclipse RCP 的开发教程，所以只列出代码清单 5-4 供读者参考，不再延伸讲解。如果读者需要这个插件，可以使用下面的代码自己编译或者发 E-mail 给笔者索取。

代码清单 5-4 Eclipse 启动耗时统计插件

ShowTime.java 代码:

```
import org.eclipse.jface.dialogs.MessageDialog;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;
import org.eclipse.ui.IStartup;

/**
 * 统计 Eclipse 启动耗时
 * @author zzm
 */
public class ShowTime implements IStartup {
    public void earlyStartup() {
        Display.getDefault().syncExec(new Runnable() {
            public void run() {
                long eclipseStartTime = Long.parseLong(System.getProperty("eclipse.startTime"));
                long costTime = System.currentTimeMillis() - eclipseStartTime;
                Shell shell = Display.getDefault().getActiveShell();
                String message = "Eclipse 启动耗时: " + costTime + "ms";
                MessageDialog.openInformation(shell, "Information", message);
            }
        });
    }
}
```

plugin.xml 代码:

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.4"?>
<plugin>
    <extension
        point="org.eclipse.ui.startup">
        <startup class="eclipsestarttime.actions.ShowTime"/>
    </extension>
</plugin>
```

上述代码打包成 jar 后放到 Eclipse 的 plugins 目录中, 反复启动几次后, 插件显示的平均时间稳定在 15 秒左右, 如图 5-3 所示。

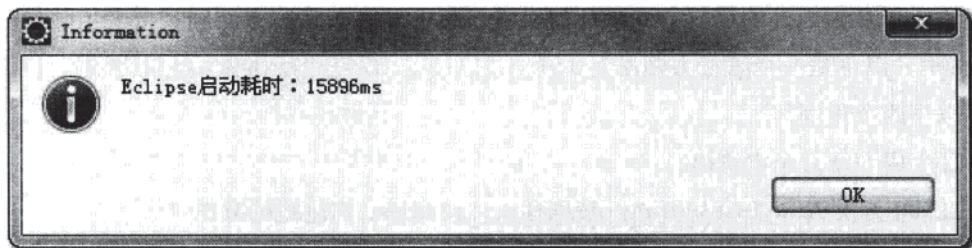


图 5-3 耗时统计插件运行效果

根据 VisualGC 和 Eclipse 插件收集到的信息，总结原始配置下的测试结果如下：

- 整个启动过程平均耗时约 15 秒。
- 最后一次启动的数据样本中，垃圾收集总耗时 4.149 秒，其中：
 - Full GC 被触发了 19 次，共耗时 3.166 秒。
 - Minor GC 被触发了 378 次，共耗时 0.983 秒。
- 加载类 9115 个，耗时 4.114 秒。
- JIT 的编译时间为 1.999 秒。
- 虚拟机 512MB 的堆内存被分配为 40MB 的新生代（31.5MB 的 Eden 空间和 2 个 4MB 的 Survivor 空间）及 472MB 的老年代。

客观地说，由于机器硬件还不错（请读者以 2010 年普通 PC 机的标准来衡量），15 秒的启动时间其实还在可接受的范围以内，但是从 VisualGC 中反映的数据来看，主要问题是非用户程序时间（图 5-2 中的 Compile Time、Class Loader Time、GC Time）非常高，占了整个启动过程耗时的一半以上（这里存在少许夸张成分，因为如果 JIT 编译等动作是在后台线程完成的，用户程序在此期间也正常执行，所以并没有占用一半以上的绝对时间）。虚拟机后台占用太多时间也直接导致 Eclipse 在启动后的使用过程中经常有停顿的感觉，所以进行调优有较大的价值。

5.3.2 升级 JDK 1.6 的性能变化及兼容问题

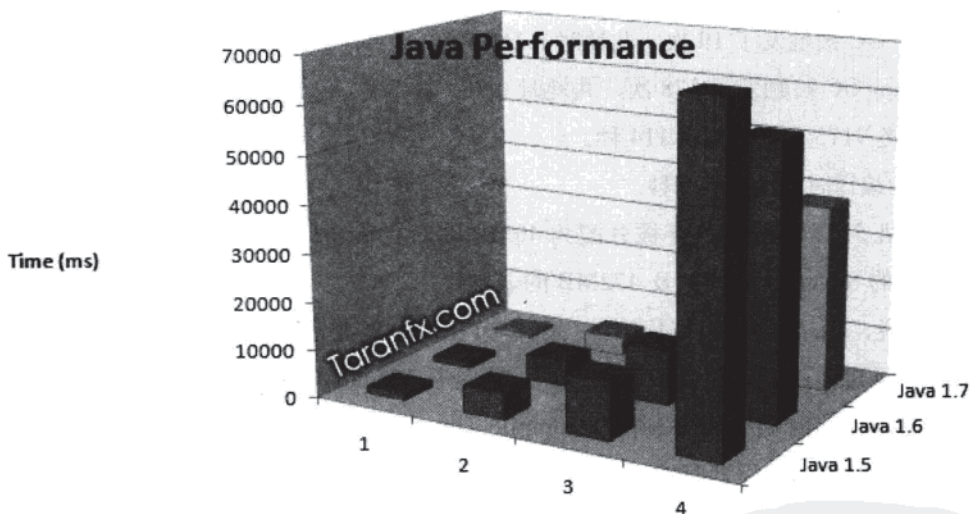
对 Eclipse 进行调优的第一步就是先把虚拟机的版本做个升级，希望能先从虚拟机版本上得到一些“免费的”性能提升。

每次 JDK 的大版本发布时，开发商肯定都会宣称虚拟机的运行速度比上一版本有了很大的提高，这虽然是个广告性质的宣言，经常被人从升级列表或技术白皮书中直接忽

略过去，但从国内外的第三方评测数据来看，版本升级在某些方面确实带来了一定的性能改善^①，以下是一个第三方网站对 JDK 1.5、1.6、1.7 三个版本做的性能评测，分别测试了以下四个用例^②：

- 生成 500 万个字符串。
- 500 万次 ArrayList <String> 数据插入，使用第一点生成的数据。
- 生成 500 万个 HashMap <String, Integer>，每个键-值对通过并发线程计算，测试并发能力。
- 打印 500 万个 ArrayList <String> 中的值到文件，并重读回内存。

三个版本的 JDK 分别运行这些用例的测试程序，测试结果如图 5-4 所示。



	1	2	3	4
Java 1.5	1453	5600	11844	68140
Java 1.6	1250	5282	11328	56156
Java 1.7	860	3895	9859	38349

图 5-4 JDK 横向性能对比

① 版本升级也有不少性能倒退的案例，受程序、第三方包兼容性及中间件限制，在企业应用中升级 JDK 版本是一件需要慎重考虑的事情。

② 测试用例、数据及图片来源于 <http://www.taranfx.com/java-7-whats-new-performance-benchmark-1-5-1-6-1-7>。

从这 4 个用例的测试结果来看, JDK 1.6 比 1.5 有大约 15% 的性能提升, 尽管对 JDK 仅测试这 4 个用例并不能说明什么问题, 需要通过测试数据来量化描述一个 JDK 比旧版提升了多少是很难做到非常科学准确的(要做稍微靠谱一点的测试, 可以使用 SPECjvm2008^①来完成, 或者把相应版本的 TCK^②中数万个测试用例的性能数据对比一下可能更有说服力), 但我还是选择相信这次“软广告”性质的测试, 把 JDK 版本升级到 1.6 Update 21, 升级的最主要理由是: 本书是基于 JDK 1.6 编写的。

与所有小说作者(嗯……我知道本书不是小说)设计的故事情节一样, 获得最后的胜利之前总是要经历各种各样的挫折, 这次升级到 JDK 1.6 之后, 性能有什么变化暂且不谈, 在使用几分钟之后, 笔者的 Eclipse 就和前面几个服务端的案例一样“不负众望”地发生了内存溢出, 如图 5-5 所示。

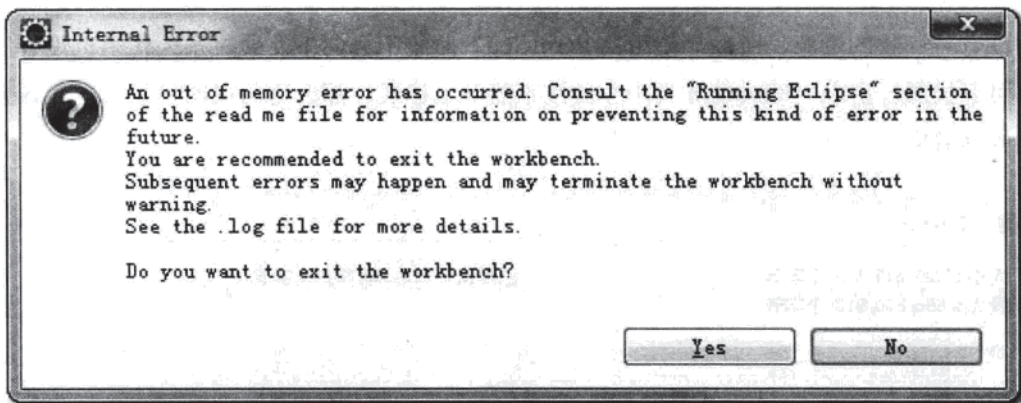


图 5-5 Eclipse OutOfMemoryError

这次内存溢出完全出乎笔者的意料: 决定对 Eclipse 做调优是因为速度慢, 但开发环境一直都很稳定, 至少没有出现过内存溢出的问题, 而这次升级除了 eclipse.ini 中的 JVM 路径改了之外, 还未进行任何运行参数的调整, 进到 Eclipse 主界面之后随便开了几个文件居然就抛出内存溢出异常了, 难道 JDK 1.6 Update21 有哪个 API 出现了严重的泄漏问题吗?

① 官方网站: <http://www.spec.org/jvm2008/docs/UserGuide.html>。

② TCK (Technology Compatibility Kit) 是一套由一组测试用例和相应的测试工具组成的工具包, 用于保证一个使用 Java 技术的实现能够完全遵守其适用的 Java 平台规范, 并且符合相应的参考实现。

事实上并不是 JDK 1.6 出现了什么问题，根据前面三章中讲解的原理和工具，我们要查明这个异常的原因并且解决它一点也不困难。打开 VisualVM，监视页签中的内存曲线部分，如图 5-6 和图 5-7 所示。

在 Java 堆的监视曲线里，“堆大小”曲线与“使用的堆”曲线一直都有很大的间隔距离，每当两条曲线开始出现互相靠近的趋势时，“最大堆”曲线就会快速向上转向，而“使用的堆”曲线会向下转向。“最大堆”曲线向上是虚拟机内部在进行堆扩容，运行参数中并没有指定最小堆（-Xms）的值与最大堆（-Xmx）相等，所以堆容量一开始并没有扩展到最大值，而是根据使用情况进行伸缩扩展。“使用的堆”曲线向下是因为虚拟机内部触发了一次垃圾收集，一些废弃对象的空间被回收后，内存用量相应减少，从图形上看，Java 堆运作是完全正常的。但永久代的监视曲线就很有问题了，“PermGen 大小”曲线与“使用的 PermGen”曲线几乎完全重合在一起，这说明永久代中没有可回收的资源，所以“使用的 PermGen”曲线不会向下发展，永久代中也没有空间可以扩展，所以“PermGen 大小”曲线不能向上扩展。那么这次很明显是永久代导致的内存溢出。

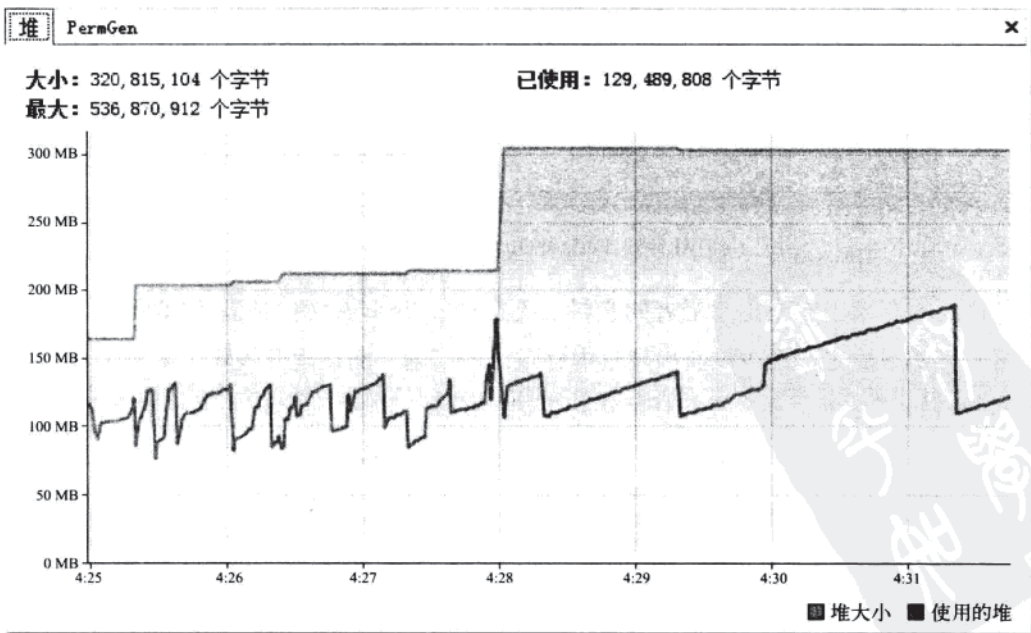


图 5-6 Java 堆监视曲线

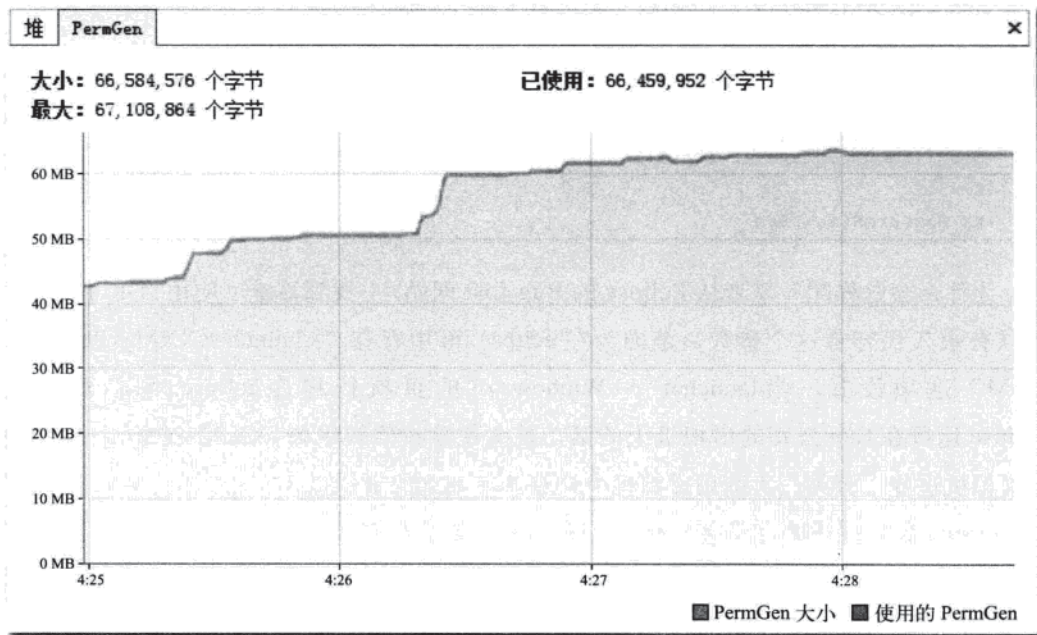


图 5-7 永久代监视曲线

再注意看图 5-7 中永久代的最大容量：“67,108,864 个字节”，也就是 64MB，这恰好是 JDK 在未使用 `-XX:MaxPermSize` 参数明确指定永久代最大容量时的默认值，无论 JDK 1.5 还是 JDK 1.6，这个默认值都是 64MB。对于 Eclipse 这种规模的 Java 程序来说，64MB 的永久代内存空间显然是不够的，溢出很正常，那为何在 JDK 1.5 中没有发生过溢出呢？

在 VisualVM 的“概述 -JVM 参数”页签中，分别检查使用 JDK 1.5 和 JDK 1.6 运行 Eclipse 时的 JVM 参数，发现使用 JDK 1.6 时只有以下 3 个 JVM 参数，如代码清单 5-5 所示。

代码清单 5-5 JDK 1.6 的 Eclipse 运行期参数

```
-Dcom.sun.management.jmxremote  
-Dorg.osgi.requiredJavaVersion=1.5  
-Xmx512m
```

而使用 JDK 1.5 运行时则有 4 个 JVM 参数，其中多出来的一个正好就是设置永久代最

大容量的 `-XX:MaxPermSize=256M`，如代码清单 5-6 所示。

代码清单 5-6 JDK 1.5 的 Eclipse 运行期参数

```
-Dcom.sun.management.jmxremote
-Dosgi.requiredJavaVersion=1.5
-Xmx512m
-XX:MaxPermSize=256M
```

为什么会这样呢？笔者从 Eclipse 的 Bug List 网站^①上找到答案：使用 JDK 1.5 时之所以有永久代容量这个参数，是因为在 `eclipse.ini` 中存在“`--launcher.XXMaxPermSize 256M`”这项设置，当 launcher——Windows 下的可执行程序 `eclipse.exe` 检测到是 Eclipse 运行在 Sun 公司的虚拟机上的话，就会把参数值转化为 `-XX:MaxPermSize` 传递给虚拟机进程，因为三大商用虚拟机中只有 Sun 系列的虚拟机才有永久代的概念，即只有 HotSpot 虚拟机需要设置这个参数，JRockit 虚拟机和 IBM J9 虚拟机都不需要设置。

2010 年 4 月 10 日，Oracle 正式完成了对 Sun 的收购，此后无论是网页还是具体的程序产品，提供商都从 Sun 变为了 Oracle，而 `eclipse.exe` 就是根据程序提供商判断是否是 Sun 的虚拟机，当 JDK 1.6 Update21 中 `java.exe`、`javaw.exe` 的“Company”属性从“Sun Microsystems Inc.”变为“Oracle Corporation”之后，Eclipse 就完全不认识这个虚拟机了，因此没有把最大永久代的参数传递过去。

了解原因之后，解决方案就简单了，launcher 不认识就只好由人来告诉它，在 `eclipse.ini` 中明确指定 `-XX:MaxPermSize=256M` 这个参数就可以了。

5.3.3 编译时间和类加载时间的优化

从 Eclipse 启动时间上看，升级到 JDK 1.6 所带来的性能提升是……嗯？基本上没有提升？多次测试的平均值与 JDK 1.5 的差距完全在实验误差范围之内。

各位读者不必失望，Sun JDK 1.6 性能白皮书^②描述的众多相对于 JDK 1.5 的提升不至于全部是广告，虽然总启动时间没有减少，但在查看运行细节的时候，却发现了一件很值得注意的事情：在 JDK 1.6 中启动完 Eclipse 所消耗类加载时间比 JDK 1.5 长了接近一倍，不要看反了，这里写的是 1.6 的类加载比 1.5 慢一倍，测试结果如代码清单 5-7

^① https://bugs.eclipse.org/bugs/show_bug.cgi?id=319514

^② http://java.sun.com/performance/reference/whitepapers/6_performance.html

所示，反复测试多次仍然是相似的结果。

代码清单 5-7 JDK 1.5 和 1.6 中的类加载时间对比

使用 JDK 1.6 的类加载时间：

```
C:\Users\IcyFenix>jps
3552
6372 org.eclipse.equinox.launcher_1.0.201.R35x_v20090715.jar
6900 Jps
```

C:\Users\IcyFenix>jstat -class 6372

Loaded	Bytes	Unloaded	Bytes	Time
7917	10190.3	0	0.0	8.18

使用 JDK 1.5 的类加载时间：

```
C:\Users\IcyFenix>jps
3552
7272 Jps
7216 org.eclipse.equinox.launcher_1.0.201.R35x_v20090715.jar
```

C:\Users\IcyFenix>jstat -class 7216

Loaded	Bytes	Unloaded	Bytes	Time
7902	9691.2	3	2.6	4.34

在本例中类加载时间上的差距并不能作为一个普遍性的测试结果去说明 JDK 1.6 的类加载必然比 1.5 慢，笔者测试了自己机器上的 Tomcat 和 ClassFish 启动过程，并没有出现类似的差距。在国内最大的 Java 社区中，笔者发起过关于此问题的讨论^①，从参与者反馈的测试结果来看，此问题只在一部分机器上存在，而且 JDK 1.6 的各个 update 版之间也存在很大差异。

多次试验后，发现在笔者机器上的两个 JDK 进行类加载时，字节码验证部分的耗时差距尤其严重。考虑到实际情况：Eclipse 使用者甚多，它的编译代码我们认为是可靠的，不需要在加载的时候再进行字节码验证，因此通过参数 `-Xverify:none` 禁止掉字节码验证过程也可作为一项优化措施。加入这个参数后，两个版本的 JDK 类加载速度都有所提高，JDK 1.6 的类加载速度仍然比 1.5 慢，但是两者的耗时已经接近了许多，测试数据

^① 关于 JDK 1.6 与 JDK 1.5 在 Eclipse 启动时类加载速度差异的讨论，参见：<http://www.javaeye.com/topic/826542>。

124 ❖ 第二部分 自动内存管理机制

如代码清单 5-8 所示。关于类与类加载的话题，譬如刚刚提到的字节码验证是怎么回事，本书专门规划了两个章节进行详细讲解，在此不再延伸。

代码清单 5-8 JDK 1.5 和 1.6 中取消字节码验证后的类加载时间对比

使用 JDK 1.6 的类加载时间：

```
C:\Users\IcyFenix>jps
5512 org.eclipse.equinox.launcher_1.0.201.R35x_v20090715.jar
5596 Jps
```

```
C:\Users\IcyFenix>jstat -class 5512
```

Loaded	Bytes	Unloaded	Bytes	Time
6749	8837.0	0	0.0	3.94

使用 JDK 1.5 的类加载时间：

```
C:\Users\IcyFenix>jps
4724 org.eclipse.equinox.launcher_1.0.201.R35x_v20090715.jar
5412 Jps
```

```
C:\Users\IcyFenix>jstat -class 4724
```

Loaded	Bytes	Unloaded	Bytes	Time
6885	9109.7	3	2.6	3.10

在取消字节码验证之后，JDK 1.5 的平均启动时间降到了 13 秒，而 JDK 1.6 的测试数据平均比 JDK 1.5 快 1 秒，平均下降到 12 秒左右，如图 5-8 所示。在类加载时间仍然落后的情况下，依然可以看到 JDK 1.6 在性能上比 JDK 1.5 略有优势，说明至少在 Eclipse 启动这个测试用例上，升级 JDK 版本确实能带来一些“免费的”性能提升。

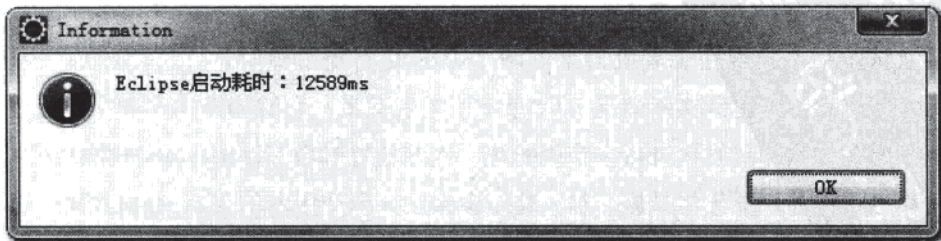


图 5-8 运行在 JDK 1.6 下取消字节码验证的启动时间

前面说过，除了类加载时间以外，在 VisualGC 的监视曲线中显示了两项很大的非用户程序耗时：编译时间（Compile Time）和垃圾收集时间（GC Time）。垃圾收集时间读者应该非常清楚了，而编译时间是什么？程序在运行之前不是已经编译了吗？虚拟机的 JIT 编译与垃圾收集一样，是本书的一个重要部分，后面有专门章节讲解，这里先简单介绍一下：编译时间是指虚拟机的 JIT 编译器（Just In Time Compiler）编译热点代码（Hot Spot Code）的耗时。我们知道 Java 语言为了实现跨平台的特性，Java 代码编译出来后形成的 Class 文件中储存的是字节码（ByteCode），虚拟机通过解释方式执行字节码命令，比起 C/C++ 编译成本地二进制代码来说，速度要慢不少。为了解决程序解释执行的速度问题，JDK 1.2 以后，虚拟机内置了两个运行时编译器^①，如果一段 Java 方法被调用的次数到达一定程度，就会被判定为热代码，从而交给 JIT 编译器即时编译为本地代码，以提高运行速度（这就是 HotSpot 虚拟机名字的由来）。甚至有可能在运行期动态编译比 C/C++ 的编译器静态编译出来的代码更优秀，因为运行期可以收集很多编译器无法知道的信息，甚至可以采用一些很激进的优化手段，在优化条件不成立的时候再逆优化退回来。所以 Java 程序只要代码没有问题（主要是泄漏问题，如内存泄漏、连接泄漏），随着代码被编译得越来越彻底，运行速度应当是越来越快。Java 运行期编译最大的缺点就是编译需要消耗程序正常的运行时间，也就是上面所说的“编译时间”。

虚拟机提供了一个参数 `-Xint` 禁止编译器运作，强制虚拟机对字节码采用纯解释方式执行。如果读者想使用这个参数省下 Eclipse 启动中那 2 秒的编译时间获得一个“更好看”的成绩的话，那恐怕要失望了，加上这个参数之后虽然编译时间确实下降到 0，但 Eclipse 启动的总时间将剧增到 27 秒。看来这个参数现在最大的作用就是让用户缅怀一下 JDK 1.2 之前那令人心酸和心碎的运行速度。

与解释执行相对应的另一方面，虚拟机还有力度更强的编译器：当虚拟机运行在 `-client` 模式的时候，使用的是一个代号为 C1 的轻量级编译器，另外还有一个代号为 C2 的相对重量级的编译器，它能提供更多的优化措施。如果使用 `-server` 模式的虚拟机启动 Eclipse 将会用到 C2 编译器，这时从 VisualGC 可以看到启动过程中虚拟机使用了超过 15 秒的时间去进行代码编译。如果读者的工作习惯是长时间不关闭 Eclipse 的话，C2 编译器所消耗的额外编译时间最终还是会在运行速度的提升之中赚回来，这样使

^① JDK 1.2 之前也可以使用外挂 JIT 编译器进行本地编译，但只能与解释器二选其一，不能同时工作。

用 `-server` 模式也是一个不错的选择。不过至少在本次实战中，我们还是继续选用 `-client` 虚拟机来运行 Eclipse。

5.3.4 调整内存设置控制垃圾收集频率

三大块非用户程序时间中，还剩下 GC 时间没有调整，而 GC 时间却是其中最重要的一块，并不只是因为它是耗时最长的一块，更因为它是一个持续稳定的过程。由于我们所做的测试是在检测程序的启动时间，类加载和编译时间在这项测试中的影响力被大幅度放大了。在绝大多数的应用中，不可能出现持续不断的类被加载和卸载。在程序运行一段时间后，热点方法不断被编译，新的热点方法数量也总会下降，但是垃圾收集是随着程序的运行而不断运作的，所以它对性能的影响才尤为重要。

在 Eclipse 启动的原始数据样本中，短短 15 秒类共发生了 19 次 Full GC 和 378 次 Minor GC，一共 397 次 GC，共造成了超过 4 秒的停顿，也就是超过 1/4 的时间都是在进行垃圾收集，这个运行数据看起来实在是太糟糕了。

首先来解决新生代中的 Minor GC，虽然 GC 的总时间只有不到 1 秒，但却发生了 378 次之多。从 VisualGC 的线程监视中看到 Eclipse 启动期间一共发起了超过 70 个线程，同时在运行的线程数超过 25 个，每当发生一次垃圾收集的动作，所有的用户线程^①都必须跑到最近的一个安全点 (SafePoint)，然后挂起线程等待垃圾回收。这样过于频繁的 GC 就会导致很多没有必要的安全点检测、线程挂起及恢复操作。

新生代 GC 频繁发生，很明显是由于虚拟机分配给新生代的空间太小而导致的，Eden 区加上一个 Survivor 区还不到 35MB，因此很有必要使用 `-Xmn` 参数调整新生代的大小。

再来看一看那 19 次 Full GC，看起来 19 次并“不多”（相对于 378 次 Minor GC 来说），但总共消耗了 3.166 秒，占了绝大部分的 GC 时间，降低 GC 时间的主要目标就是降低这部分时间。从 VisualGC 的曲线图上看得不够精确，这次直接从 GC 日志^②中分析一下这些 Full GC 是如何产生的，代码清单 5-9 中是启动最开始的 2.5 秒内发生的 10 次 Full GC 的记录。

-
- ① 严格来说，不包括正在执行 native 代码的用户线程，因为 native 代码一般不会改变 Java 对象的引用关系，所以没有必要挂起它们来等待垃圾回收。
 - ② 可以通过以下几个参数要求虚拟机生成 GC 日志：`-XX:+PrintGCTimeStamps`（打印 GC 停顿时间）、`-XX:+PrintGCDetails`（打印 GC 详细信息）、`-verbose:gc`（打印 GC 信息，输出内容已被前一个参数包括，可以不写）、`-Xloggc:gc.log`。

代码清单 5-9 Full GC 记录

```

0.278: [GC 0.278: [DefNew: 574K->33K(576K), 0.0012562 secs]0.279: [Tenured: 1467K->997K(1536K), 0.0181775 secs] 1920K->997K(2112K), 0.0195257 secs]
0.312: [GC 0.312: [DefNew: 575K->64K(576K), 0.0004974 secs]0.312: [Tenured: 1544K->1608K(1664K), 0.0191592 secs] 1980K->1608K(2240K), 0.0197396 secs]
0.590: [GC 0.590: [DefNew: 576K->64K(576K), 0.0006360 secs]0.590: [Tenured: 2675K->2219K(2684K), 0.0256020 secs] 3090K->2219K(3260K), 0.0263501 secs]
0.958: [GC 0.958: [DefNew: 551K->64K(576K), 0.0011433 secs]0.959: [Tenured: 3979K->3470K(4084K), 0.0419335 secs] 4222K->3470K(4660K), 0.0431992 secs]
1.575: [Full GC 1.575: [Tenured: 4800K->5046K(5784K), 0.0543136 secs] 5189K->5046K(6360K), [Perm : 12287K->12287K(12288K)], 0.0544163 secs]
1.703: [GC 1.703: [DefNew: 703K->63K(704K), 0.0012609 secs]1.705: [Tenured: 8441K->8505K(8540K), 0.0607638 secs] 8691K->8505K(9244K), 0.0621470 secs]
1.837: [GC 1.837: [DefNew: 1151K->64K(1152K), 0.0020698 secs]1.839: [Tenured: 14616K->14680K(14688K), 0.0708748 secs] 15035K->14680K(15840K), 0.0730947 secs]
2.144: [GC 2.144: [DefNew: 1856K->191K(1856K), 0.0026810 secs]2.147: [Tenured: 25092K->24656K(25108K), 0.1112429 secs] 26172K->24656K(26964K), 0.1141099 secs]
2.337: [GC 2.337: [DefNew: 1914K->0K(3136K), 0.0009697 secs]2.338: [Tenured: 41779K->27347K(42056K), 0.0954341 secs] 42733K->27347K(45192K), 0.0965513 secs]
2.465: [GC 2.465: [DefNew: 2490K->0K(3456K), 0.0011044 secs]2.466: [Tenured: 46379K->27635K(46828K), 0.0956937 secs] 47621K->27635K(50284K), 0.0969918 secs]

```

括号中加粗的数字代表着老年代的容量，这组 GC 日志显示了 10 次 Full GC 发生的原因全部都是老年代空间耗尽，每发生一次 Full GC 都伴随着一次老年代空间扩容：1536KB → 1664KB → 2684KB → …… → 42056KB → 46828KB，10 次 GC 以后老年代容量从起始的 1536KB 扩大到 46828KB，当 15 秒后 Eclipse 启动完成时，老年代容量扩大到了 103428KB，代码编译开始后，老年代容量达到顶峰 473MB，整个 Java 堆达到最大容量 512MB。

日志还显示有些时候内存回收状况很不理想，空间扩容成为获取可用内存的最主要手段，譬如，“Tenured: 25092K → 24656K(25108K), 0.1112429 secs”代表老年代的当前容量为 25108KB，内存使用到 25092KB 的时候发生 Full GC，花费 0.11 秒把内存使用降低到 24656KB，只回收了不到 500KB 的内存，这次 GC 基本没有什么回收效果，仅仅做了扩容，扩容过程比起回收过程可以看做是基本不需要花费时间的，所以说这 0.11 秒几乎是白白浪费了。

由上述分析可以得出结论：Eclipse 启动时 Full GC 大多数是由于老年代容量扩展而导致的，由永久代空间扩展而导致的也有一部分。为了避免这些扩展所带来的性能浪费，

128 ❖ 第二部分 自动内存管理机制

我们可以把 `-Xms` 和 `-XX:PermSize` 参数值分别设置为 `-Xmx` 和 `-XX:PermSizeMax` 参数值，强制虚拟机在启动的时候就把老年代和永久代的容量固定下来，避免运行时自动扩展^①。

根据分析，优化计划确定为：把新生代容量提升到 128MB，避免新生代频繁 GC；把 Java 堆、永久代的容量分别固定为 512MB 和 96MB^②，避免内存扩展。这几个数值都是根据机器硬件、Eclipse 插件和工程数量来决定的，读者实战的时候应根据 VisualGC 中收集到的实际数据进行设置。改动后的 `eclipse.ini` 配置如代码清单 5-10 所示。

代码清单 5-10 内存调整后的 Eclipse 配置文件

```
-vm
D:/_DevSpace/jdk1.6.0_21/bin/javaw.exe
-startup
plugins/org.eclipse.equinox.launcher_1.0.201.R35x_v20090715.jar
--launcher.library
plugins/org.eclipse.equinox.launcher.win32.win32.x86_1.0.200.v20090519
-product
org.eclipse.epp.package.jee.product
-showsplash
org.eclipse.platform
-vmargs
-Dosgi.requiredJavaVersion=1.5
-Xverify:none
-Xmx512m
-Xms512m
-Xmn128m
-XX:PermSize=96m
-XX:MaxPermSize=96m
```

现在的这个配置之下，GC 次数已经大幅度降低，图 5-9 是 Eclipse 启动后 1 分钟的监视曲线，只发生了 8 次 Minor GC 和 4 次 Full GC，总耗时为 1.928 秒。

这个结果已经算是基本正常，但是还存在一点瑕疵：从 Old Gen 曲线上看，永久代直接固定在 384MB，而内存使用量只有 66MB，并且一直很平滑，完全不应该发生 Full GC 才对，那 4 次 Full GC 是怎么来的？使用 `jstat -gccause` 查询一下最近一次 GC 的原因，见代码清单 5-11。

① 需要说明一点，虚拟机启动的时候就会把参数中所设定的内存全部划为私有，即使扩容前有一部分内存不会被用户代码用到，这部分内存也不会交给其他进程使用。这部分内存存在虚拟机中被标识为“Virtual”内存。

② 512MB 和 96MB 两个数值对于笔者的应用情况来说依然偏少，但由于笔者需要同时开 VMWare 工作，所以要预留较多内存空间，读者在实际调优时不妨再设置大一些。

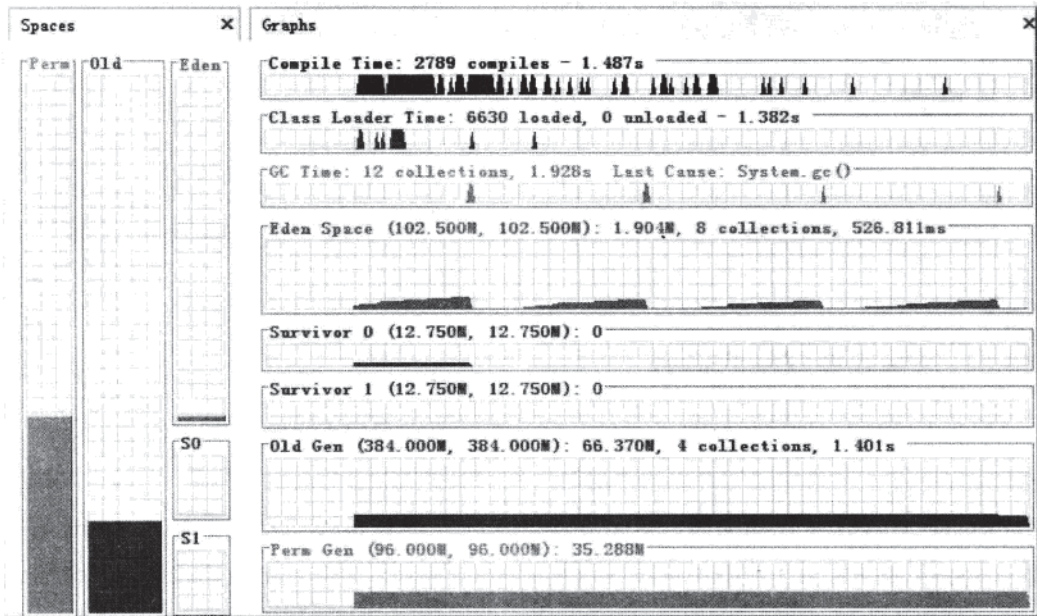


图 5-9 GC 调整后的运行数据

代码清单 5-11 查询 GC 原因

```
C:\Users\IcyFenix>jps
9772 Jps
4068 org.eclipse.equinox.launcher_1.0.201.R35x_v20090715.jar

C:\Users\IcyFenix>jstat -gccause 4068
  S0   S1   E    O    P   YGC   YGCT   FGC   FGCT   GCT   LGCC   GCC
  0.00  0.00  1.00  14.81  39.29    6   0.422   20   5.992   6.414
System.gc()                No GC
```

从 LGCC (Last GC Cause) 中看到原来是代码调用 System.gc() 显式触发的 GC, 在内存设置调整后, 这种显式 GC 不符合我们的期望, 因此在 eclipse.ini 中加入参数 -XX:+DisableExplicitGC 屏蔽掉 System.gc()。再次测试发现启动期间的 Full GC 已经完全没有了, 只有 6 次 Minor GC, 耗时 417 毫秒, 与调优前 4.149 秒的测试样本相比, 正好是十分之一。进行 GC 调优后 Eclipse 的启动时间下降非常明显, 比整个 GC 时间降低的绝对值还大, 现在启动只需要 7 秒多, 如图 5-10 所示。

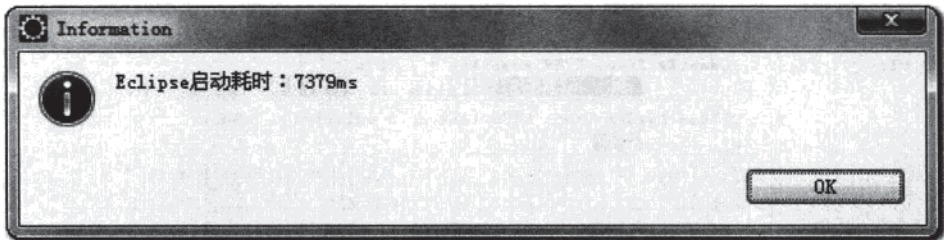


图 5-10 Eclipse 启动时间

5.3.5 选择收集器降低延迟

现在 Eclipse 启动已经比较迅速了，但我们的调优实战还没有结束，毕竟 Eclipse 是拿来写程序的，不是拿来测试启动速度的。我们不妨再在 Eclipse 中测试一个非常常用但又比较耗时的操作：代码编译。图 5-11 是当前配置下 Eclipse 进行代码编译时的运行数据，从图中可以看出，新生代每次回收耗时约 65 毫秒，老年代每次回收耗时约 725 毫秒。对于用户来说，新生代 GC 的耗时还好，65 毫秒在使用中无法察觉到，而老年代每次 GC 停顿的时间接近 1 秒钟，虽然比较长时间才会出现一次，但停顿还是太长了一些。

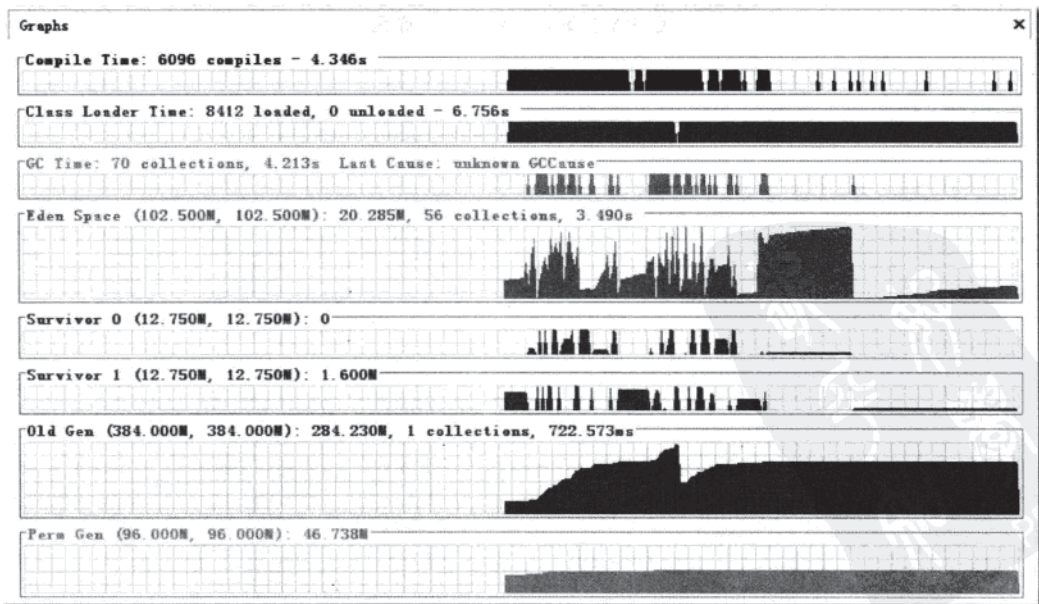


图 5-11 编译期间运行数据

再注意看一下编译期间的CPU资源使用状况，图5-12是Eclipse在编译期间的CPU使用率曲线图，整个编译过程中平均只使用了不到30%的CPU资源，垃圾收集的CPU使用率曲线更是几乎与坐标横轴紧贴在一起，这说明CPU资源还有很多可利用的余地。

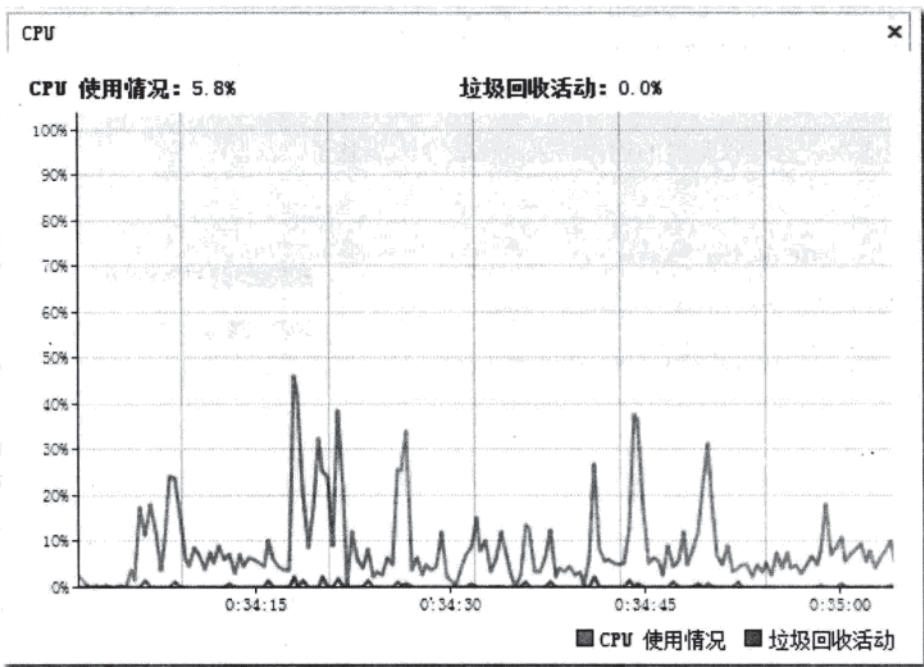


图 5-12 编译期间 CPU 的曲线

列举 GC 停顿时间、CPU 资源富余的目的，都是为接下来替换掉 Client 模式的虚拟机中默认的新生代及老年代串行收集器做铺垫。

Eclipse 应当算是与使用者交互非常频繁的应用程序，由于代码太多，笔者习惯在做全量编译或清理动作的时候，使用“Run in Backgroup”功能一边编译一边继续工作。回顾一下在第3章提到的几种收集器，很容易想到 CMS 是最符合这类场景的收集器。因此尝试在 eclipse.ini 中再加入两个参数 `-XX:+UseConcMarkSweepGC` 和 `-XX:+UseParNewGC`（ParNew 收集器是使用 CMS 收集器后的默认新生代收集器，写上仅是为了使得配置更加清晰），要求虚拟机在新生代和老年代分别使用 ParNew 和 CMS 收集器进行垃圾回收。指定收集器之后，再次测试的结果如图 5-13 所示，与原来

使用的串行收集器对比，新生代停顿从每次 65 毫秒下降到了每次 53 毫秒，而老年代的停顿时间更是从 725 毫秒大幅下降到了 36 毫秒。

当然，CMS 的停顿阶段只是收集过程中的一小部分，并不是真的把垃圾收集时间从 725 毫秒变成 36 毫秒了。在 GC 日志中可以看到 CMS 与程序并发的时间约为 400 毫秒。不过由于 CMS 默认老年代使用了 68% 就进行收集，所以 Full GC 次数上升到了 6 次，为了避免总体吞吐量下降得太厉害，使用 `-XX:CMSInitiatingOccupancyFraction = 85` 将 GC 临界值提升到 85%，修改 `-XX:CMSInitiatingOccupancyFraction` 参数后，Full GC 次数下降至 3 次，这样收集器的运作结果就比较令人满意了。

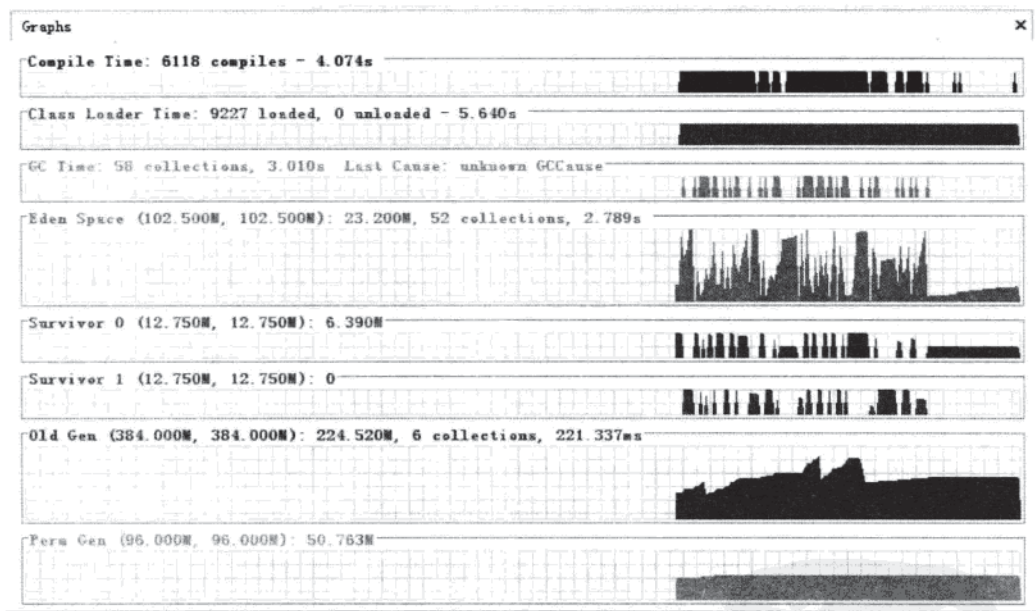


图 5-13 指定 ParNew 和 CMS 收集器后的 GC 数据

到这里为止，对于虚拟机内存的调优就基本结束了，这次实战可以看做是一次简化的服务端调优过程，服务端调优有可能还会出现在更多方面，如数据库、资源池、磁盘 I/O 等，对于虚拟机内存部分的优化，与这次实战中的思路没有太大差别。即使读者实际工作中接触不到服务器，根据自己的工作环境做一些试验，总结几个参数让自己日常工作环境的速度有较大幅度的提升也是很划算的。最终 `eclipse.ini` 的配置如代码清单 5-12 所示。

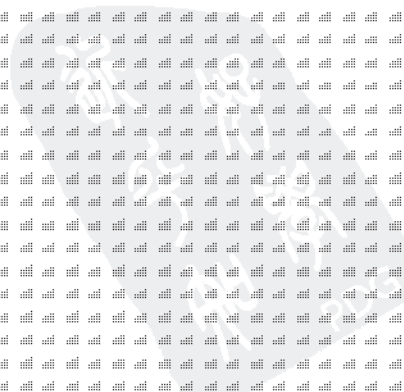
代码清单 5-12 修改收集器配置后的 Eclipse 配置

```
-vm
D:/_DevSpace/jdk1.6.0_21/bin/javaw.exe
-startup
plugins/org.eclipse.equinox.launcher_1.0.201.R35x_v20090715.jar
--launcher.library
plugins/org.eclipse.equinox.launcher.win32.win32.x86_1.0.200.v20090519
-product
org.eclipse.epp.package.jee.product
-showsplash
org.eclipse.platform
-vmargs
-Dcom.sun.management.jmxremote
-Dosgi.requiredJavaVersion=1.5
-Xverify:none
-Xmx512m
-Xms512m
-Xmn128m
-XX:PermSize=96m
-XX:MaxPermSize=96m
-XX:+DisableExplicitGC
-Xnoclassgc
-XX:+UseParNewGC
-XX:+UseConcMarkSweepGC
-XX:CMSInitiatingOccupancyFraction=85
```

5.4 本章小结

Java 虚拟机的内存管理与垃圾收集是虚拟机结构体系中最重要的一部分，对我们程序的性能和稳定性有着非常大的影响，在本书的第 2～5 章中，笔者从理论知识、异常现象、代码、工具、案例和实战等几个方面对其进行讲解，希望读者能有所收获。

本书关于虚拟机内存管理的部分到此为止就结束了，下一章我们将开始学习 Class 文件与虚拟机执行子系统方面的知识。





第三部分 虚拟机执行子系统

- 第 6 章 类文件结构
- 第 7 章 虚拟机类加载机制
- 第 8 章 虚拟机字节码执行引擎
- 第 9 章 类加载及执行子系统的案例与实战



第 6 章 类文件结构

本章主要内容

- 概述
- 无关性的基石
- Class 类文件的结构
- Class 文件结构的发展

代码编译的结果从本地机器码转变为字节码，是存储格式发展的一小步，却是编程语言发展的一大步。

6.1 概述

记得在第一节计算机程序课上老师就讲过：“计算机只认识 0 和 1，所以我们写的程序需要被编译器翻译成由 0 和 1 构成的二进制格式才能被计算机执行”。10 多年的时间过去了，今天的计算机仍然只能识别 0 和 1，但由于最近 10 年内虚拟机及建立在虚拟机之上的大量程序语言如雨后春笋般出现并蓬勃发展，将我们编写的程序编译成二进制本地机器码（Native Code）已不再是唯一的选择，越来越多的程序语言选择了与操作系统和机器指令集无关的、平台中立的格式作为程序编译后的存储格式。

6.2 无关性的基石

如果计算机的 CPU 指令集就只有 x86 一种，操作系统就只有 Windows 一种，那也许就不会有 Java 语言的出现。Java 在刚刚诞生之时曾经提出过一个非常著名的宣传口号：“一次编写，到处运行（Write Once, Run Anywhere）”，这句话充分表达了软

件开发人员对冲破平台界限的渴求。在无时无刻不充满竞争的 IT 领域，不可能只有 Wintel^①存在，我们也不希望只有 Wintel 存在，各种不同的硬件体系结构和不同的操作系统定将会长期并存发展。“与平台无关”的理想最终实现在操作系统的应用层上：Sun 公司及其他虚拟机提供商发布了许多可以运行在各种不同平台上的虚拟机，这些虚拟机都可以载入和执行同一种平台无关的字节码，从而实现程序的“一次编写，到处运行”。

各种不同平台的虚拟机与所有平台都统一使用的程序存储格式——字节码 (ByteCode) 是构成平台无关性的基石，但本节标题中刻意省略了“平台”二字，那是因为笔者注意到虚拟机的另外一种中立的特性——如今语言无关性越来越被开发者所重视。到今天为止，或许大部分程序员都还认为 Java 虚拟机执行 Java 程序是一件理所当然和天经地义的事情。但在 Java 发展之初，设计者们就曾经考虑过并实现了让其他语言运行在 Java 虚拟机之上的可能性，他们在发布规范文档的时候，也刻意把 Java 的规范拆分成了 Java 语言规范《The Java Language Specification》及 Java 虚拟机规范《The Java Virtual Machine Specification》。时至今日，商业机构和开源机构已经在 Java 语言之外发展出一大批在 Java 虚拟机之上运行的语言，如 Clojure、Groovy、JRuby、Jython、Scala，等等。使用过这些语言的开发者可能还不是非常多，但是听说过的人肯定已经不少，随着时间的推移，谁能保证日后 Java 虚拟机在语言无关性上的优势不会赶上甚至超越它在平台无关性上的优势呢？

实现语言无关性的基础仍然是虚拟机和字节码存储格式，使用 Java 编译器可以把 Java 代码编译为存储字节码的 Class 文件，使用 JRuby 等其他语言的编译器一样可以把程序代码编译成 Class 文件，虚拟机并不关心 Class 的来源是什么语言，只要它符合 Class 文件应有的结构就可以在 Java 虚拟机中运行，如图 6-1 所示。

Java 语言中的各种变量、关键字和运算符的语义最终都是由多条字节码命令组合而成的，因此字节码命令所能提供的语义描述能力肯定会比 Java 语言本身更强大。因此，有一些 Java 语言本身无法有效支持的语言特性并不代表字节码本身无法有效支持，这也为其他语言实现一些有别于 Java 的语言特性提供了基础。

① Wintel：微软的 Windows 与 Intel 的芯片相结合，曾经是业界最强大的联盟。

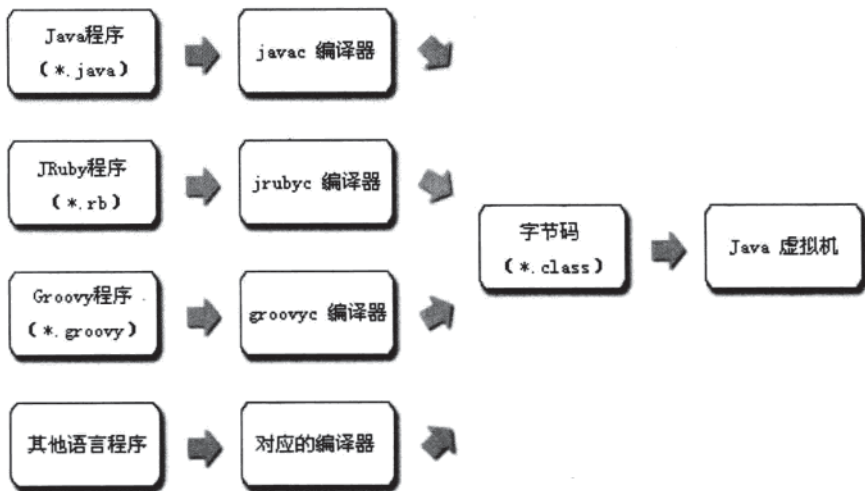


图 6-1 Java 虚拟机提供的语言无关性

6.3 Class 类文件的结构

解析 Class 文件的数据结构是本章的最主要内容。笔者曾经在前言中阐述过本书的写作风格：力求在保证逻辑准确的前提下，用尽量通俗的语言和案例去讲述虚拟机中与开发关系最为密切的内容。但是，对数据结构方面的讲解不可避免地会比较枯燥，而这部分内容又是了解虚拟机的重要基础之一。如果想比较深入地了解虚拟机，这部分是不能不接触的。

Class 文件是一组以 8 位字节为基础单位的二进制流，各个数据项目严格按照顺序紧凑地排列在 Class 文件之中，中间没有添加任何分隔符，这使得整个 Class 文件中存储的内容几乎全部都是程序运行的必要数据，没有空隙存在。当遇到需要占用 8 位字节以上空间的数据项时，则会按照高位在前的方式分割成若干个 8 位字节进行存储。

根据 Java 虚拟机规范的规定，Class 文件格式采用一种类似于 C 语言结构体的伪结构来存储，这种伪结构中只有两种数据类型：无符号数和表，后面的解析都要以这两种数据类型为基础，所以这里要先讲明白这两个概念。

无符号数属于基本的数据类型，以 u1、u2、u4、u8 来分别代表 1 个字节、2 个字节、4 个字节和 8 个字节的无符号数，无符号数可以用来描述数字、索引引用、数量值，或

者按照 UTF-8 编码构成字符串值。

表是由多个无符号数或其他表作为数据项构成的复合数据类型，所有表都习惯性地以“_info”结尾。表用于描述有层次关系的复合结构的数据，整个 Class 文件本质上就是一张表，它由表 6-1 所示的数据项构成。

表 6-1 Class 文件格式

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

无论是无符号数还是表，当需要描述同一类型但数量不定的多个数据时，经常会使用一个前置的容量计数器加若干个连续的数据项的形式，这时候称这一系列连续的某一类型的数据为某一类型的集合。

本节结束之前，笔者需要再重复一下，Class 的结构不像 XML 等描述语言，由于它没有任何分隔符号，所以在表 6-1 中的数据项，无论是顺序还是数量，都是被严格限定的，哪个字节代表什么含义，长度是多少，先后顺序如何，都不允许改变。接下来我们将一起看看这个表中各个数据项的具体含义。

6.3.1 魔数与 Class 文件的版本

每个 Class 文件的头 4 个字节称为魔数 (Magic Number)，它的唯一作用是用于确

定这个文件是否为一个能被虚拟机接受的 Class 文件。很多文件存储标准中都使用魔数来进行身份识别，譬如图片格式，如 gif 或 jpeg 等在文件头中都存有魔数。使用魔数而不是扩展名来进行识别主要是基于安全考虑，因为文件扩展名可以很随意地被改动。文件格式的制定者可以自由地选择魔数值，只要这个魔数值还没有被广泛采用过而且不会引起混淆即可。Class 文件的魔数的获得很有“浪漫气息”，值为：0xCAFEBABE（咖啡宝贝？），这个魔数值在 Java 还被称做“Oak”语言的时候（大约是 1991 年前后）就已经确定下来了。它还有一段很有趣的历史，据 Java 开发小组最初的关键成员 Patrick Naughton 所说：“我们一直在寻找一些好玩的、容易记忆的东西，选择 0xCAFEBABE 是因为它象征着著名咖啡品牌 Peet's Coffee 中深受欢迎的 Baristas 咖啡”，这个魔数似乎也预示着日后“Java”这个名称的出现。

紧接着魔数的 4 个字节存储的是 Class 文件的版本号：第 5 和第 6 个字节是次版本号（Minor Version），第 7 个和第 8 个字节是主版本号（Major Version）。Java 的版本号是从 45 开始的，JDK 1.1 之后的每个 JDK 大版本发布主版本号向上加 1（JDK 1.0 ~ 1.1 使用了 45.0 ~ 45.3 的版本号），高版本的 JDK 能向下兼容以前版本的 Class 文件，但不能运行以后版本的 Class 文件，即使文件格式并未发生变化。JDK 1.1 能支持版本号为 45.0 ~ 45.65535 的 Class，无法执行版本号为 46.0 以上的 Class，而 JDK 1.2 则能支持 45.0 ~ 46.65535 的 Class 文件。现在，最新的 JDK 版本为 1.7，可生成的 Class 文件主版本号的最大值为 51.0。为了讲解方便，笔者准备了一段最简单的 Java 代码（如代码清单 6-1 所示）本章后面的讲解都将以这段小程序使用 JDK 1.6 编译输出的 Class 文件为基础来讲解。

代码清单 6-1 简单的 Java 代码

```
package org.fenixsoft.clazz;

public class TestClass {

    private int m;

    public int inc() {
        return m + 1;
    }

}
```



图 6-2 显示的是使用十六进制编辑器 WinHex 打开这个 Class 文件的结果，可以清楚地看见开头 4 个字节的十六进制表示的是 0xCAFEBA BE，代表次版本号的第 5 个和第 6 个字节值为 0x0000，而主版本号的值为 0x0032，即十进制的 50，该版本号说明这个是可以被 JDK 1.6 或以上版本的虚拟机执行的 Class 文件。

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00000000	CA	FE	BA	BE	00	00	00	32	00	16	07	00	02	01	00	1D	漱壕... ..
00000010	6F	72	67	2F	66	65	6E	69									org/fenixsoft/cl
00000020	61	7A	7A	2F	54	65	73	74									azz/TestClass...
00000030	01	00	10	6A	61	76	61	2F									...java/lang/Obj

图 6-2 Java Class 文件的结构

表 6-2 列举了从 JDK 1.1 到 1.7 之间，主流 JDK 版本编译器输出的默认和可支持的 Class 文件版本号。

表 6-2 Class 文件版本号

编译器版本	-target 参数	十六进制版本号	十进制版本号
JDK 1.1.8	不能带 target 参数	00 03 00 2D	45.3
JDK 1.2.2	不带（默认为 -target 1.1）	00 03 00 2D	45.3
JDK 1.2.2	-target 1.2	00 00 00 2E	46.0
JDK 1.3.1_19	不带（默认为 -target 1.1）	00 03 00 2D	45.3
JDK 1.3.1_19	-target 1.3	00 00 00 2F	47.0
JDK 1.4.2_10	不带（默认为 -target 1.2）	00 00 00 2E	46.0
JDK 1.4.2_10	-target 1.4	00 00 00 30	48.0
JDK 1.5.0_11	不带（默认为 -target 1.5）	00 00 00 31	49.0
JDK 1.5.0_11	-target 1.4 -source 1.4	00 00 00 30	48.0
JDK 1.6.0_01	不带（默认为 -target 1.6）	00 00 00 32	50.0
JDK 1.6.0_01	-target 1.5	00 00 00 31	49.0
JDK 1.6.0_01	-target 1.4 -source 1.4	00 00 00 30	48.0
JDK 1.7.0	不带（默认为 -target 1.7）	00 00 00 33	51.0
JDK 1.7.0	-target 1.6	00 00 00 32	50.0
JDK 1.7.0	-target 1.4 -source 1.4	00 00 00 30	48.0

6.3.2 常量池

紧接着主次版本号之后的是常量池入口，常量池是 Class 文件结构中与其他项目关联最多的数据类型，也是占用 Class 文件空间最大的数据项目之一，同时它还是在 Class

文件中第一个出现的表类型数据项目。

由于常量池中常量的数量是不固定的，所以在常量池的入口需要放置一项 u2 类型的数据，代表常量池容量计数值 (constant_pool_count)。与 Java 中的语言习惯不一样的是，这个容量计数是从 1 而不是 0 开始的，如图 6-3 所示，常量池容量（偏移地址：0x00000008）为十六进制数 0x0016，即十进制的 22，这就代表常量池中有 21 项常量，索引值为 1~21。制定 Class 文件格式规范时，将第 0 项常量空出来是有特殊考虑的，这样做是为了满足后面某些指向常量池的索引值的数据在特定情况下需要表达“不引用任何一个常量池项目”的意思，这种情况就可以把索引值置为 0 来表示。Class 文件结构中只有常量池的容量计数是从 1 开始的，对于其他集合类型，包括接口索引集合、字段表集合、方法表集合等的容量计数都与一般习惯相同，是从 0 开始的。

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00000000	CA	FE	BA	BE	00	00	00	32	00	16	07	00	02	01	00	1D	漱壕...2.█.....
00000010	6F	72	67	2F	66	65	6E	69	78	73	6F	66	74	2F	63	6C	org/fenixsoft/cl
00000020	61	7A	7A	2F	54	65	73	74	43	6C	61	73	73	07	00	04	azz/TestClass...
00000030	01	00	10	6A	61	76	61	2F	6								...java/lang/Obj
00000040	65	63	74	01	00	01	6D	01	0								ect...m...I...<i
00000050	6E	69	74	3E	01	00	03	28	2								nit>...()V...Cod

图 6-3 常量池结构

常量池之中主要存放两大类常量：字面量 (Literal) 和符号引用 (Symbolic References)。字面量比较接近于 Java 语言层面的常量概念，如文本字符串、被声明为 final 的常量值等。而符号引用则属于编译原理方面的概念，包括了下面三类常量：

- 类和接口的全限定名 (Fully Qualified Name)
- 字段的名称和描述符 (Descriptor)
- 方法的名称和描述符

Java 代码在进行 Javac 编译的时候，并不像 C 和 C++ 那样有“连接”这一步骤，而是在虚拟机加载 Class 文件的时候进行动态连接。也就是说，在 Class 文件中不会保存各个方法和字段的最终内存布局信息，因此这些字段和方法的符号引用不经过转换的话是无法直接被虚拟机使用的。当虚拟机运行时，需要从常量池获得对应的符号引用，再在类创建时或运行时解析并翻译到具体的内存地址之中。关于类的创建和动态连接的内容，在下一章会有详细的讲解。

常量池中的每一项常量都是一个表，共有 11 种结构^①各不相同的表结构数据，这 11 种表都有一个共同的特点，就是表开始的第一位是一个 u1 类型的标志位（tag，取值为 1 至 12，缺少标志为 2 的数据类型），代表当前这个常量属于哪种常量类型，11 种常量类型所代表的具体含义如表 6-3 所示。

表 6-3 常量池的项目类型

类 型	标志	描 述
CONSTANT_Utf8_info	1	UTF-8 编码的字符串
CONSTANT_Integer_info	3	整型字面量
CONSTANT_Float_info	4	浮点型字面量
CONSTANT_Long_info	5	长整型字面量
CONSTANT_Double_info	6	双精度浮点型字面量
CONSTANT_Class_info	7	类或接口的符号引用
CONSTANT_String_info	8	字符串类型字面量
CONSTANT_Fieldref_info	9	字段的符号引用
CONSTANT_Methodref_info	10	类中方法的符号引用
CONSTANT_InterfaceMethodref_info	11	接口中方法的符号引用
CONSTANT_NameAndType_info	12	字段或部分符号引用

之所以说常量池是最繁琐的数据，是因为这 11 种常量类型各自均有自己的结构。回头看看图 6-3 中常量池的第一项常量，它的标志位（偏移地址：0x0000000A）是 0x07，查看表 6-3 的标志列会发现这个常量属于 CONSTANT_Class_info 类型，此类型的常量代表一个类或接口的符号引用。CONSTANT_Class_info 的结构比较简单，如表 6-4 所示。

表 6-4 CONSTANT_Class_info 型常量的结构

类型	名称	数量
u1	tag	1
u2	name_index	1

tag 是标志位，上面已经讲解过了；name_index 是一个索引值，它指向常量池中一个 CONSTANT_Utf8_info 类型的常量，此常量代表了这个类（或者接口）的全限定名，这里的 name_index 值（偏移地址：0x0000000B）为 0x0002，即指向了常量池中的第二

① JDK 1.7 中为了实现 JSR-292 规范《Supporting Dynamically Typed Languages on the Java Platform》，很可能添加 CONSTANT_InvokeDynamic 和 CONSTANT_InvokeDynamicTrans 这两个常量类型，这两个常量在目前的 OpenJDK 7 的代码中已存在，但由于 JDK 1.7 正式版尚未发布，本书暂不计算这两个常量。

144 ❖ 第三部分 虚拟机执行子系统

项常量。继续从图 6-3 中查找第二项常量，它的标志位（地址：0x000000D）是 0x01，查看表 6-3 可知确实是一个 CONSTANT_Utf8_info 类型的常量。CONSTANT_Utf8_info 类型的结构如表 6-5 所示。

表 6-5 CONSTANT_Utf8_info 型常量的结构

类型	名称	数量
u1	tag	1
u2	length	1
u1	bytes	length

length 值说明了这个 UTF-8 编码的字符串长度是多少字节，它后面紧跟着的长度为 length 字节的连续数据是一个使用 UTF-8 缩略编码表示的字符串。UTF-8 缩略编码与普通 UTF-8 编码的区别是：从 '\u0001' 到 '\u007f' 之间的字符（相当于 1 ~ 127 的 ASCII 码）的缩略编码使用一个字节表示，从 '\u0080' 到 '\u07ff' 之间的所有字符的缩略编码用两个字节表示，从 '\u0800' 开始到 '\uffff' 之间的所有字符的缩略编码就按照普通 UTF-8 编码规则使用三个字节表示。

顺便提一下，由于 Class 文件中方法、字段等都需要引用 CONSTANT_Utf8_info 型常量来描述名称，所以 CONSTANT_Utf8_info 型常量的最大长度也就是 Java 中方法和字段名的最大长度。而这里的最大长度就是 length 的最大值，即 u2 类型能表达的最大值 65535。所以 Java 程序中如果定义了超过 64KB 英文字符的变量或方法名，将会无法编译。

本例中这个字符串的 length 值（偏移地址：0x0000000E）为 0x001D，也就是长 29 个字节，往后 29 个字节正好都在 1 ~ 127 的 ASCII 码范围以内，内容为“org/fenixsoft/clazz/TestClass”，有兴趣的读者可以自己逐个字节换算一下，换算结果如图 6-4 选中的部分所示。

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00000000	CA	FE	BA	BE	00	00	00	32	00	16	07	00	02	01	00	1D	漱壕...2.....
00000010	6F	72	67	2F	66	65	6E	69	78	73	6F	66	74	2F	63	6C	org/fenixsoft/cl
00000020	61	7A	7A	2F	54	65	73	74	43	6C	61	73	7E	07	00	04	azz/TestClas█...
00000030	01	00	10	6A	61	76	61	2F	6C	61	6E	67	2F	4F	62	6A	...java/lang/Obj

图 6-4 常量池 UTF-8 字符串结构

到此为止，我们分析了 TestClass.class 常量池中 21 个常量中的两个，其余的 19 个常量都可以通过类似的方法计算出来。为了避免计算过程占用过多的版面，后续的 19 个常量的计算过程可以借助计算机来帮我们完成。在 JDK 的 bin 目录中，Oracle 公司已经为我们准备好一个专门用于分析 Class 文件字节码的工具：javap，代码清单 6-2 中列出了使用 javap 工具的 -verbose 参数输出 TestClass.class 文件的字节码内容（此清单中省略了常量池以外的信息）。前面我们曾经提到过 Class 文件中还有很多数据项都要引用常量池中的常量，所以代码清单 6-2 中的内容在后续的讲解之中还要经常使用到。

代码清单 6-2 使用 Javap 命令输出常量表

```
C:\>javap -verbose TestClass
Compiled from "TestClass.java"
public class org.fenixsoft.clazz.TestClass extends java.lang.Object
  SourceFile: "TestClass.java"
  minor version: 0
  major version: 50
  Constant pool:
const #1 = class      #2;    // org/fenixsoft/clazz/TestClass
const #2 = Asciz      org/fenixsoft/clazz/TestClass;
const #3 = class      #4;    // java/lang/Object
const #4 = Asciz      java/lang/Object;
const #5 = Asciz      m;
const #6 = Asciz      I;
const #7 = Asciz      <init>;
const #8 = Asciz      ()V;
const #9 = Asciz      Code;
const #10 = Method    #3.#11; // java/lang/Object."<init>":()V
const #11 = NameAndType #7:#8; // "<init>":()V
const #12 = Asciz      LineNumberTable;
const #13 = Asciz      LocalVariableTable;
const #14 = Asciz      this;
const #15 = Asciz      Lorg/fenixsoft/clazz/TestClass;;
const #16 = Asciz      inc;
const #17 = Asciz      ()I;
const #18 = Field     #1.#19; // org/fenixsoft/clazz/TestClass.m:I
const #19 = NameAndType #5:#6; // m:I
const #20 = Asciz      SourceFile;
const #21 = Asciz      TestClass.java;
```


从代码清单 6-2 中可以看到计算机已经帮我们把整个常量池的 21 项常量都计算了出来，并且第 1、2 项常量的计算结果与我们手工计算的结果一致。仔细看一下会发现，其中有一些常量似乎从来没有在代码中出现过，如“`I`”、“`V`”、“`<init>`”、“`LineNumberTable`”、“`LocalVariableTable`”等，这些看起来在代码任何一处都没有出现过的常量是哪里来的？

这部分自动生成的常量的确没有在 Java 代码里面直接出现过，但它们会被后面即将讲到的字段表（`field_info`）、方法表（`method_info`）、属性表（`attribute_info`）所引用到，它们会被用来描述一些不方便使用“固定字节”来表达的内容。譬如描述方法的返回值是什么？有几个参数？每个参数的类型是什么？因为 Java 中的“类”是无穷无尽的，无法通过简单的无符号字节来描述一个方法用到了什么类，因此在描述方法的这些信息时，需要引用常量表中的符号引用进行表达。这部分内容将在后面作进一步的详细阐述。最后，笔者将 11 种常量项的结构定义总结为表 6-6 所示。

表 6-6 常量池中的 11 种数据类型的结构总表

常量	项目	类型	描述
CONSTANT_Utf8_info	tag	u1	值为 1
	length	u2	UTF-8 编码的字符串占用了字节数
	bytes	u1	长度为 length 的 UTF-8 编码的字符串
CONSTANT_Integer_info	tag	u1	值为 3
	bytes	u4	按照高位在前存储的 int 值
CONSTANT_Float_info	tag	u1	值为 4
	bytes	u4	按照高位在前存储的 float 值
CONSTANT_Long_info	tag	u1	值为 5
	bytes	u8	按照高位在前存储的 long 值
CONSTANT_Double_info	tag	u1	值为 6
	bytes	u8	按照高位在前存储的 double 值
CONSTANT_Class_info	tag	u1	值为 7
	index	u2	指向全限定名常量项的索引
CONSTANT_String_info	tag	u1	值为 8
	index	u2	指向字符串字面量的索引
CONSTANT_Fieldref_info	tag	u1	值为 9
	index	u2	指向声明字段的类或接口描述符 CONSTANT_Class_info 的索引项
	index	u2	指向字段描述符 CONSTANT_NameAndType 的索引项

(续)

常量	项目	类型	描述
CONSTANT_Methodref_info	tag	u1	值为 10
	index	u2	指向声明方法的类描述符 CONSTANT_Class_info 的索引项
	index	u2	指向名称及类型描述符 CONSTANT_NameAndType 的索引项
CONSTANT_InterfaceMethodref_info	tag	u1	值为 11
	index	u2	指向声明方法的接口描述符 CONSTANT_Class_info 的索引项
	index	u2	指向名称及类型描述符 CONSTANT_NameAndType 的索引项
CONSTANT_NameAndType_info	tag	u1	值为 12
	index	u2	指向该字段或方法名称常量项的索引
	index	u2	指向该字段或方法描述符常量项的索引

6.3.3 访问标志

在常量池结束之后，紧接着的 2 个字节代表访问标志（access_flags），这个标志用于识别一些类或接口层次的访问信息，包括：这个 Class 是类还是接口；是否定义为 public 类型；是否定义为 abstract 类型；如果是类的话，是否被声明为 final，等等。具体的标志位及标志的含义见表 6-7。

表 6-7 访问标志

标志名称	标志值	含 义
ACC_PUBLIC	0x0001	是否为 public 类型
ACC_FINAL	0x0010	是否被声明为 final，只有类可设置
ACC_SUPER	0x0020	是否允许使用 invokespecial 字节码指令，JDK 1.2 之后编译出来的类的这个标志为真
ACC_INTERFACE	0x0200	标识这是一个接口
ACC_ABSTRACT	0x0400	是否为 abstract 类型，对于接口或抽象类来说，此标志值为真，其他类值为假
ACC_SYNTHETIC	0x1000	标识这个类并非由用户代码产生的
ACC_ANNOTATION	0x2000	标识这是一个注解
ACC_ENUM	0x4000	标识这是一个枚举

access_flags 中一共有 32 个标志位可以使用，当前只定义了其中的 8 个^①，没有使用到的标志位要求一律为 0。以代码清单 6-1 中的代码为例，TestClass 这个类被 public 关键字修饰但没有被声明为 final 和 abstract，并且它使用了 JDK 1.2 之后的编译器进行编译，因此它的 ACC_PUBLIC、ACC_SUPER 标志应当为真，而 ACC_FINAL、ACC_INTERFACE、ACC_ABSTRACT、ACC_SYNTHETIC、ACC_ANNOTATION、ACC_ENUM 这六个标志应当为假，因此它的 access_flags 的值应为：0x0001 | 0x0020 = 0x0021。从图 6-5 中可以看到，access_flags 标志（偏移地址：0x000000EF）的确为 0x0021。

000000D0	06 01 00 0A 53 6F 75 72	63 65 46 69 6C 65 01 00SourceFile..
000000E0	0E 54 65 73 74 43 6C 61	73 73 2E 6A 61 76 61 00	.TestClass.java.
000000F0	21 00 01 00 03 00 00 00	01 00 02 00 05 00 06 00	!.....
00000100	00 00 02 00 01 00 07 00	08 00 01 00 09 00 00 00

图 6-5 access_flags 标志

6.3.4 类索引、父类索引与接口索引集合

类索引（this_class）和父类索引（super_class）都是一个 u2 类型的数据，而接口索引集合（interfaces）是一组 u2 类型的数据的集合，Class 文件中由这三项数据来确定这个类的继承关系。类索引用于确定这个类的全限定名，父类索引用于确定这个类的父类的全限定名。由于 Java 语言不允许多重继承，所以父类索引只有一个，除了 java.lang.Object 之外，所有的 Java 类都有父类，因此除了 java.lang.Object 外，所有 Java 类的父类索引都不为 0。接口索引集合就用来描述这个类实现了哪些接口，这些被实现的接口将按 implements 语句（如果这个类本身是一个接口，则应当是 extends 语句）后的接口顺序从左到右排列在接口的索引集合中。

类索引、父类索引和接口索引集合都按顺序排列在访问标志之后，类索引和父类索引引用两个 u2 类型的索引值表示，它们各自指向一个类型为 CONSTANT_Class_info 的类描述符常量，通过 CONSTANT_Class_info 类型的常量中的索引值可以找到定义在 CONSTANT_Utf8_info 类型的常量中的全限定名字符串。图 6-6 演示了代码清单 6-1 的代码的类索引查找过程。

^① 在 Java 虚拟机规范中，只定义了开头 5 种标志。JDK 1.5 中增加了后面 3 种。这些标志在 JSR-202 规范中声明，是对《Java 虚拟机规范（第 2 版）》的补充。本书介绍的访问标志以 JSR-202 规范为准。

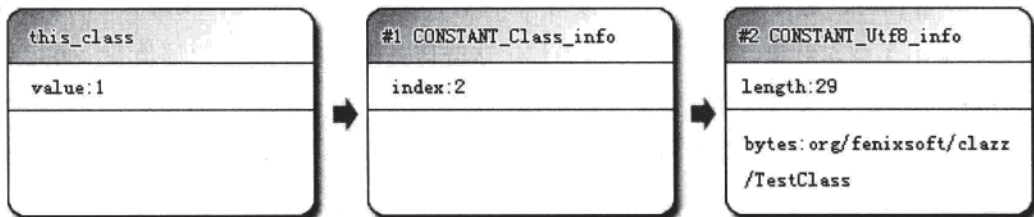


图 6-6 类索引查找全限定名的过程

对于接口索引集合，入口的第一项——u2 类型的数为接口计数器（`interfaces_count`），表示索引表的容量。如果该类没有实现任何接口，那么该计数器值为 0，后面接口的索引表不再占用任何字节。代码清单 6-1 中的代码的类索引、父类索引与接口表索引的内容如图 6-7 所示。

```

000000D0 | 06 01 00 0A 53 6F 75 72 63 65 46 69 6C 65 01 00 | ....SourceFile..
000000E0 | 0E 54 65 73 74 43 6C 61 73 73 2E 6A 61 76 61 00 | .TestClass.java.
000000F0 | 21 00 01 00 03 00 00 00 01 00 02 00 05 00 06 00 | !.....
00000100 | 00 00 02 00 01 00 07 00 08 00 01 00 09 00 00 00 | .....

```

图 6-7 类索引、父类索引、接口索引集合

从偏移地址 0x000000F1 开始的 3 个 u2 类型的值分别为 0x0001、0x0003 和 0x0000，也就是类索引为 1，父类索引为 3，接口索引集合大小为 0，查询前面代码清单 6-2 中 `javap` 命令计算出来的常量池，找出对应的类和父类的常量，结果如代码清单 6-3 所示。

代码清单 6-3 部分常量池的内容

```

const #1 = class #2; // org/fenixsoft/clazz/TestClass
const #2 = Asciz org/fenixsoft/clazz/TestClass;
const #3 = class #4; // java/lang/Object
const #4 = Asciz java/lang/Object;

```

6.3.5 字段表集合

字段表（`field_info`）用于描述接口或类中声明的变量。字段（`field`）包括了类级变量或实例级变量，但不包括在方法内部声明的变量。我们可以想一想在 Java 中描述一个字段可以包含什么信息？可以包括的信息有：字段的作用域（`public`、`private`、`protected` 修饰符）、是类级变量还是实例级变量（`static` 修饰符）、可变性（`final`）、并发可见性

150 ❖ 第三部分 虚拟机执行子系统

(volatile 修饰符, 是否强制从主内存读写)、可否序列化 (transient 修饰符)、字段数据类型 (基本类型、对象、数组)、字段名称。这些信息中, 各个修饰符都是布尔值, 要么有某个修饰符, 要么没有, 很适合使用标志位来表示。而字段叫什么名字、字段被定义为什么数据类型, 这些都是无法固定的, 只能引用常量池中的常量来描述。表 6-8 中列出了字段表的最终格式。

表 6-8 字段表结构

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

字段修饰符放在 access_flags 项目中, 它与类中的 access_flags 项目是非常类似的, 都是一个 u2 的数据类型, 其中可以设置的标志位和含义如表 6-9 所示。

表 6-9 字段访问标志

标志名称	标志值	含 义
ACC_PUBLIC	0x0001	字段是否 public
ACC_PRIVATE	0x0002	字段是否 private
ACC_PROTECTED	0x0004	字段是否 protected
ACC_STATIC	0x0008	字段是否 static
ACC_FINAL	0x0010	字段是否 final
ACC_VOLATILE	0x0040	字段是否 volatile
ACC_TRANSIENT	0x0080	字段是否 transient
ACC_SYNTHETIC	0x1000	字段是否由编译器自动产生的
ACC_ENUM	0x4000	字段是否 enum

很明显, 在实际情况中, ACC_PUBLIC、ACC_PRIVATE、ACC_PROTECTED 三个标志最多只能选择其一, ACC_FINAL、ACC_VOLATILE 不能同时选择。接口之中的字段必须有 ACC_PUBLIC、ACC_STATIC、ACC_FINAL 标志, 这些都是由 Java 本身的语言规则所决定的。

跟随 access_flags 标志的是两项索引值: name_index 和 descriptor_index。它们都是对常量池的引用, 分别代表着字段的简单名称及字段和方法的描述符。现在需要解

释一下“简单名称”、“描述符”及前面出现过多次的“全限定名”这三种特殊字符串的概念。

全限定名和简单名称很好理解，以代码清单 6-1 中的代码为例，“org/fenixsoft/clazz/TestClass”是这个类的全限定名，仅仅是把类全名中的“.”替换成了“/”而已，为了使连续的多个全限定名之间不产生混淆，在使用时最后一般会加入一个“;”号表示全限定名结束。简单名称则就是指没有类型和参数修饰的方法或字段名称，这个类中的 inc() 方法和 m 字段的简单名称分别是“inc”和“m”。

相对于全限定名和简单名称来说，方法和字段的描述符就要复杂一些。描述符的作用是用来描述字段的数据类型、方法的参数列表（包括数量、类型以及顺序）和返回值。根据描述符规则，基本数据类型（byte、char、double、float、int、long、short、boolean）及代表无返回值的 void 类型都用一个大写字母来表示，而对象类型则用字符 L 加对象的全限定名来表示，详见表 6-10。

表 6-10 描述符标识字符含义

标识字符	含义
B	基本类型 byte
C	基本类型 char
D	基本类型 double
F	基本类型 float
I	基本类型 int
J	基本类型 long
S	基本类型 short
Z	基本类型 boolean
V ^①	特殊类型 void
L	对象类型，如 Ljava/lang/Object;

对于数组类型，每一维度将使用一个前置的“[”字符来描述，如一个定义为“java.lang.String[][]”类型的二维数组，将被记录为：“[[Ljava/lang/String;”，一个整型数组“int[]”将被记录为“[I”。

用描述符来描述方法时，按照先参数列表，后返回值的顺序描述，参数列表按照

① void 类型在虚拟机规范之中单独列出为“VoidDescriptor”，笔者为了结构统一，将其列在基本数据类型中一起描述。

参数的严格顺序放在一组小括号“()”之内。如方法 `void inc()` 的描述符为“(V)”，方法 `java.lang.String toString()` 的描述符为“(Ljava/lang/String;”，方法 `int indexOf(char[] source, int sourceOffset, int sourceCount, char[] target, int targetOffset, int targetCount, int fromIndex)` 的描述符为“([CII[CHII)I”。

对于代码清单 6-1 中的 `TestClass.class` 文件来说，字段表集合从地址 `0x000000F8` 开始，第一个 `u2` 类型的数据为容量计数器 `fields_count`，如图 6-8 所示，其值为 `0x0001`，说明这个类只有一个字段表数据。接下来紧跟着容量计数器的是 `access_flags` 标志，值为 `0x0002`，代表 `private` 修饰符的 `ACC_PRIVATE` 标志位为真（`ACC_PRIVATE` 标志的值为 `0x0002`），其他修饰符为假。代表字段名称的 `name_index` 的值为 `0x0005`，从代码清单 6-2 列出的常量表中可查得第五项常量是一个 `CONSTANT_Utf8_info` 类型的字符串，其值为“m”，代表字段描述符的 `descriptor_index` 的值为 `0x0006`，指向常量池的字符串“I”，根据这些信息，我们可以推断出原代码定义的字段为“`private int m;`”。

000000D0	06 01 00 0A 53 6F 75 72 63 65 46 69 6C 65 01 00Sourcefile..
000000E0	0E 54 65 73 74 43 6C 61 73 73 2E 6A 61 76 61 00	.TestClass.java.
000000F0	21 00 01 00 03 00 00 00 01 00 02 00 00 05 00 06 00	!.....
00000100	00 00 02 00 01 00 07 00 08 00 01 00 09 00 00 00
	↑ ↑ ↑	
	fields_count access_flags name_index descriptor_index	

图 6-8 字段表结构实例

字段表都包含的固定数据项目到 `descriptor_index` 为止就结束了，不过在 `descriptor_index` 之后跟随着一个属性表集合用于存储一些额外的信息，字段都可以在属性表中描述 0 至多项额外的信息。对于本例中的字段 `m`，它的属性表计数器为 0，也就是没有需要额外描述的信息，但是，如果将字段 `m` 的声明改为“`final static int m = 123;`”，那就可能会存在一项名为 `ConstantValue` 的属性，其值指向常量 123。关于 `attribute_info` 的其他内容，将在 6.3.7 节介绍属性表的数据项目时再进一步讲解。

字段表集合中不会列出从超类或父接口中继承而来的字段，但有可能列出原本 Java 代码之中不存在的字段，譬如在内部类中为了保持对外部类的访问性，会自动添加指向外部类实例的字段。另外，在 Java 语言中字段是无法重载的，两个字段的数据类型、修饰符不管是否相同，都必须使用不一样的名称，但是对于字节码来讲，如果两个字段的描述符不一致，那字段重名就是合法的。

6.3.6 方法表集合

如果理解了上一节关于字段表的内容，那本节关于方法表的内容将会变得很简单。Class 文件存储格式中对方法的描述与对字段的描述几乎采用了完全一致的方式，方法表的结构如同字段表一样，依次包括了访问标志（access_flags）、名称索引（name_index）、描述符索引（descriptor_index）、属性表集合（attributes）几项，如表 6-11 所示。这些数据项目的含义非常类似，仅在访问标志和属性表集合的可选项中有所区别。

表 6-11 方法表结构

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

因为 volatile 关键字和 transient 关键字不能修饰方法，所以方法表的访问标志中没有了 ACC_VOLATILE 标志和 ACC_TRANSIENT 标志。与之相对的，synchronized、native、strictfp 和 abstract 关键字可以修饰方法，所以方法表的访问标志中增加了 ACC_SYNCHRONIZED、ACC_NATIVE、ACC_STRICTFP 和 ACC_ABSTRACT 标志。对于方法表，所有标志位及其取值可参见表 6-12。

表 6-12 方法访问标志

标志名称	标志值	含 义
ACC_PUBLIC	0x0001	方法是否为 public
ACC_PRIVATE	0x0002	方法是否为 private
ACC_PROTECTED	0x0004	方法是否为 protected
ACC_STATIC	0x0008	方法是否为 static
ACC_FINAL	0x0010	方法是否为 final
ACC_SYNCHRONIZED	0x0020	方法是否为 synchronized
ACC_BRIDGE	0x0040	方法是否是由编译器产生的桥接方法
ACC_VARARGS	0x0080	方法是否接受不定参数
ACC_NATIVE	0x0100	方法是否为 native
ACC_ABSTRACT	0x0400	方法是否为 abstract
ACC_STRICT	0x0800	方法是否为 strictfp
ACC_SYNTHETIC	0x1000	方法是否是由编译器自动产生的

中，因此 Java 语言里面是无法仅仅依靠返回值的不同来对一个已有方法进行重载的。但是在 Class 文件格式之中，特征签名的范围更大一些，只要描述符不是完全一致的两个方法也可以共存。也就是说，如果两个方法有相同的名称和特征签名，但返回值不同，那么也是可以合法共存于同一个 Class 文件中的。

6.3.7 属性表集合

属性表 (attribute_info) 在前面的讲解之中已经出现过数次，在 Class 文件、字段表、方法表中都可以携带自己的属性表集合，以用于描述某些场景专有的信息。

与 Class 文件中其他的数据项目要求严格的顺序、长度和内容不同，属性表集合的限制稍微宽松了一些，不再要求各个属性表具有严格的顺序，并且只要不与已有的属性名重复，任何人实现的编译器都可以向属性表中写入自己定义的属性信息，Java 虚拟机运行时会忽略掉它不认识的属性。为了能正确地解析 Class 文件，《Java 虚拟机规范（第 2 版）》中预定义了 9 项虚拟机实现应当能识别的属性，具体如表 6-13 所示。后文中将对这 9 种属性逐一进行讲解^①。

表 6-13 虚拟机规范预定义的属性

属性名称	使用位置	含 义
Code	方法表	Java 代码编译成的字节码指令
ConstantValue	字段表	final 关键字定义的常量值
Deprecated	类、方法表、字段表	被声明为 deprecated 的方法和字段
Exceptions	方法表	方法抛出的异常
InnerClasses	类文件	内部类列表
LineNumberTable	Code 属性	Java 源码的行号与字节码指令的对应关系
LocalVariableTable	Code 属性	方法的局部变量描述
SourceFile	类文件	源文件名称
Synthetic	类、方法表、字段表	标识方法或字段为编译器自动生成的

^① 随着 JDK 版本的发展，JCP 也通过发布 JSR 规范的方式保持着对《Java 虚拟机规范（第 2 版）》的更新，在尚处于草稿状态的《Java 虚拟机规范（第 3 版）》中，属性表集合的预定义内容已增加到 19 项。由于篇幅的关系，本书介绍 Class 格式中的属性表部分时仍然以《Java 虚拟机规范（第 2 版）》为基准，但会在最后一节集中描述这些新加入的信息，并且后面在讲解 JDK 1.5 和 JDK 1.6 相对于 JDK 1.4 的改进和优化时，将会重新提及这些新加入的特性。而在介绍 Class 文件格式的其他部分时，将以 JSR-202 中对虚拟机规范的最新补充为基准。

对于每个属性，它的名称需要从常量池中引用一个 `CONSTANT_Utf8_info` 类型的常量来表示，而属性值的结构则是完全自定义的，只需要说明属性值所占用的位数长度即可。一个符合规则的属性表应该满足表 6-14 中所定义的结构。

表 6-14 属性表结构

类 型	名 称	数 量
u2	attribute_name_index	1
u2	attribute_lenght	1
u1	info	attribute_lenght

1. Code 属性

Java 程序方法体里面的代码经过 `Javac` 编译器处理之后，最终变为字节码指令存储在 Code 属性内。Code 属性出现在方法表的属性集合之中，但并非所有的方法表都必须存在这个属性，譬如接口或抽象类中的方法就不存在 Code 属性，如果方法表有 Code 属性存在，那么它的结构将如表 6-15 所示。

表 6-15 Code 属性表的结构

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

`attribute_name_index` 是一项指向 `CONSTANT_Utf8_info` 型常量的索引，常量值固定为“Code”，它代表了该属性的属性名称，`attribute_length` 指示了属性值的长度，由于属性名称索引与属性长度一共是 6 个字节，所以属性值的长度固定为整个属性表的长度减去 6 个字节。

`max_stack` 代表了操作数栈 (Operand Stacks) 深度的最大值。在方法执行的任意

时刻，操作数栈都不会超过这个深度。虚拟机运行的时候需要根据这个值来分配栈帧（Frame）中的操作栈深度。

`max_locals` 代表了局部变量表所需的存储空间。在这里，`max_locals` 的单位是 Slot，Slot 是虚拟机为局部变量分配内存所使用的最小单位。对于 `byte`、`char`、`float`、`int`、`short`、`boolean`、`reference` 和 `returnAddress` 等长度不超过 32 位的数据类型，每个局部变量占用 1 个 Slot，而 `double` 和 `long` 这两种 64 位的数据类型则需要 2 个 Slot 来存放。方法参数（包括实例方法中的隐藏参数“this”）、显式异常处理器的参数（Exception Handler Parameter，即 try-catch 语句中 catch 块所定义的异常）、方法体中定义的局部变量都需要使用局部变量表来存放。另外，并不是在方法中用到了多少个局部变量，就把这些局部变量所占的 Slot 之和作为 `max_locals` 的值，原因是局部变量表中的 Slot 可以重用，当代码执行超出一个局部变量的作用域时，这个局部变量所占的 Slot 就可以被其他局部变量所使用，编译器会根据变量的作用域来分类 Slot 并分配给各个变量使用，然后计算出 `max_locals` 的大小。

`code_length` 和 `code` 用来存储 Java 源程序编译后生成的字节码指令。`code_length` 代表字节码长度，`code` 是用于存储字节码指令的一系列字节流。既然名为字节码指令，那么每个指令就是一个 `u1` 类型的单字节，当虚拟机读取到 `code` 中的一个字节码时，就可以相应地找出这个字节码代表的是什么指令，并且可以知道这条指令后面是否需要跟随参数，以及参数应当如何理解。我们知道一个 `u1` 数据类型的取值范围为 `0x00` 至 `0xFF`，对应十进制的 0~255，也就是一共可以表达 256 条指令。目前，Java 虚拟机规范已经定义了其中约 200 条编码值对应的指令含义，编码与指令之间的对应关系可查阅本书的附录 B “虚拟机字节码指令表”。

关于 `code_length`，还有一件值得注意的事情，虽然它是一个 `u4` 类型的长度值，理论上最大值可以达到 $2^{32}-1$ ，但是虚拟机规范中限制了一个方法不允许超过 65535 条字节码指令，如果超过这个限制，Javac 编译器就会拒绝编译。一般来讲，只要我们写 Java 代码时不是刻意去编写超长的方法，就不会超过这个最大值的限制。但是，在编译复杂的 JSP 文件时，可能会因为这个原因而导致编译失败。

Code 属性是 Class 文件中最重要的一個属性，如果把一个 Java 程序中的信息分为代码（Code，方法体里面的 Java 代码）和元数据（Metadata，包括类、字段、方法定义及其他信息）两部分，那么在整个 Class 文件里，Code 属性用于描述代码，所有的其他数

据项目就都用于描述元数据。了解 Code 属性是学习后面两章关于字节码执行引擎内容的必要基础，直接阅读字节码也是工作中分析 Java 代码语义问题的必要工具，因此笔者准备了一个比较详细的实例来讲解虚拟机是如何使用这个属性的。

继续以代码清单 6-1 的 TestClass.class 文件为例，如图 6-10 所示，这是上一节分析过的实例构造器“<init>”方法的 Code 属性。它的操作数栈的最大深度和本地变量表的容量都为 0x0001，字节码区域所占空间的长度为 0x0005。虚拟机读取到字节码区域的长度后，按照顺序依次读入紧随的 5 个字节，并根据字节码指令表翻译出所对应的字节码指令。“2A B7 00 0A B1”的翻译过程为：

1) 读入 2A，查表得 0x2A 所对应的指令为 aload_0，这个指令的含义是将第 0 个 Slot 中为 reference 类型的本地变量推送至栈顶。

2) 读入 B7，查表得 0xB7 所对应的指令为 invokespecial，这条指令的作用是以栈顶的 reference 类型的数据所指向的对象作为方法接收者，调用此对象的实例构造器方法、private 方法或其父类的方法。这个方法有一个 u2 类型的参数说明具体调用哪一个方法，它指向常量池中的一个 CONSTANT_Methodref_info 类型常量，即此方法的方法符号引用。

3) 读入 00 0A，这是 invokespecial 的参数，查常量池得 0x000A 对应的常量为实例构造器“<init>”方法的符号引用。

4) 读入 B1，查表得 0xB1 对应的指令为 return，含义是返回此方法，并且返回值为 void。这条指令执行后，当前方法结束。

000000F0	21	00	01	00	03	00	00	00	01	00	02	00	05	00	06	00
00000100	00	00	02	00	01	00	07	00	08	00	01	00	09	00	00	00
00000110	2F	00	01	00	01	00	00	00	05	2A	B7	00	0A	B1	00	00	/.....*?.±..
00000120	00	02	00	0C	00	00	00	06	00	01	00	00	00	03	00	0D
		↑		↑		↑		↑									
		max_stack		max_locals		code_length		code									

图 6-10 Code 属性结构实例

这段字节码虽然很短，但是可以看出它执行过程中的数据交换、方法调用等操作都是基于栈（操作栈）进行的。我们可以初步猜测：Java 虚拟机执行字节码是基于栈的体系结构。但是与一般基于堆栈的零字节指令又不太一样，某些指令（如 invokespecial）后面还会带有参数，关于虚拟机字节码执行的讲解是后面两章的重点，我们不妨把这里

的疑问放到第8章去解决。

我们再次使用 javap 命令把此 Class 文件中的另外一个方法的字节码指令也计算出来, 结果如代码清单 6-4 所示。

代码清单 6-4 用 Javap 命令计算字节码指令

```
// 原始 Java 代码
public class TestClass {
    private int m;

    public int inc() {
        return m + 1;
    }
}

C:\>javap -verbose TestClass
// 常量表部分的输出见代码清单 6-1, 因版面原因在此省略掉。
{
public org.fenixsoft.clazz.TestClass();
    Code:
        Stack=1, Locals=1, Args_size=1
        0:   aload_0
        1:   invokespecial   #10; //Method java/lang/Object.<init>:()V
        4:   return
   LineNumberTable:
        line 3: 0

    LocalVariableTable:
        Start  Length  Slot  Name   Signature
        0       5       0     this   Lorg/fenixsoft/clazz/TestClass;

public int inc();
    Code:
        Stack=2, Locals=1, Args_size=1
        0:   aload_0
        1:   getfield         #18; //Field m:I
        4:   iconst_1
        5:   iadd
        6:   ireturn
   LineNumberTable:
        line 8: 0
```

```

LocalVariableTable:
  Start  Length  Slot  Name  Signature
    0      7      0   this      Lorg/fenixsoft/clazz/TestClass;
}

```

如果大家注意到 javap 中输出的 “Args_size” 的值，可能会有疑问：这个类有两个方法——实例构造器 <init>() 和 inc()，这两个方法很明显都是没有参数的，为什么 Args_size 会为 1？而且无论是在参数列表里还是方法体内，都没有定义任何局部变量，那 Locals 又为什么会等于 1？如果有这样的疑问，大家可能是忽略了一点：在任何实例方法中，都可以通过 “this” 关键字访问到此方法所属的对象。这个访问机制对 Java 程序的编写很重要，而它的实现却非常简单，仅仅是通过 Javac 编译器在编译的时候把对 this 关键字的访问转变为对一个普通方法参数的访问，然后在虚拟机调用实例方法时自动传入此参数即可。因此在实例方法的局部变量表中至少会存在一个指向当前对象实例的局部变量，局部变量表中也会预留出第一个 Slot 位来存放对象实例的引用，方法参数值从 1 开始计算。这个处理只对实例方法有效，如果代码清单 6-1 中的 inc() 方法被声明为 static，那 Args_size 就不会等于 1 而是等于 0 了。

在字节码指令之后的是这个方法的显式异常处理表（以下简称异常表）集合，异常表对于 Code 属性来说并不是必须存在的，如代码清单 6-4 中就没有异常表生成。

异常表的格式如表 6-16 所示，它包含四个字段，这些字段的含义为：如果字节码从第 start_pc 行^①到第 end_pc 行之间（不含第 end_pc 行）出现了类型为 catch_type 或其子类的异常（catch_type 为指向一个 CONSTANT_Class_info 型常量的索引），则转到第 handler_pc 行继续处理。当 catch_type 的值为 0 时，代表任何的异常情况都需要转向到 handler_pc 处进行处理。

异常表实际上是 Java 代码的一部分，编译器使用异常表而不是简单的跳转命令来实现 Java 异常及 finally 处理机制^②，代码清单 6-5 是一段演示异常表如何运作的例子，这

① 此处字节码的“行”是一种形象的描述，指的是字节码相对于方法体开始的偏移量，而不是 Java 源码的行号，下同。

② 在 JDK1.4.2 之前的 Javac 编译器采用了 jsr 和 ret 指令实现 finally 语句，但在 1.4.2 之后就已经改为异常表实现了。

段代码同时也演示字节码层面的 try-catch-finally 是如何实现的。阅读字节码之前，大家不妨先看看下面的 Java 源码，想一下这段代码的返回值在出现异常和不出现异常的情况下分别应为多少？

表 6-16 属性表结构

类 型	名 称	数 量
u2	start_pc	1
u2	end_pc	1
u2	handler_pc	1
u2	catch_type	1

代码清单 6-5 异常表运作演示

```
// Java 源码
public int inc() {
    int x;
    try {
        x = 1;
        return x;
    } catch (Exception e) {
        x = 2;
        return x;
    } finally {
        x = 3;
    }
}

// 编译后的 ByteCode 字节码及异常表
public int inc();
Code:
Stack=1, Locals=5, Args_size=1
0:  iconst_1    // try 块中的 x=1
1:  istore_1
2:  iload_1     // 保存 x 到 returnValue 中, 此时 x=1
3:  istore  4
5:  iconst_3    // finally 块中的 x=3
6:  istore_1
7:  iload  4    // 将 returnValue 中的值放到栈顶, 准备给 ireturn 返回
9:  ireturn
10: astore_2   // 给 catch 中定义的 Exception e 赋值, 存储在 Slot 2 中
11: iconst_2   // catch 块中的 x=2
```


162 ❖ 第三部分 虚拟机执行子系统

```

12: istore_1
13: iload_1    // 保存 x 到 returnValue 中, 此时 x=2
14: istore    4
16: iconst_3   // finally 块中的 x=3
17: istore_1
18: iload    4    // 将 returnValue 中的值放到栈顶, 准备给 ireturn 返回
20: ireturn
21: astore_3   // 如果出现了不属于 java.lang.Exception 及其子类的异常才会走到这里
22: iconst_3   // finally 块中的 x=3
23: istore_1
24: aload_3    // 将异常放置到栈顶, 并抛出
25: athrow

```

Exception table:

from	to	target	type
0	5	10	Class java/lang/Exception
0	5	21	any
10	16	21	any

编译器为这段 Java 源码生成了三条异常表记录, 对应三条可能出现的代码执行路径。从 Java 代码的语义上讲, 这三条执行路径分别为:

- 如果 try 语句块中出现属于 Exception 或其子类的异常, 则转到 catch 语句块处理。
- 如果 try 语句块中出现不属于 Exception 或其子类的异常, 则转到 finally 语句块处理。
- 如果 catch 语句块中出现任何异常, 则转到 finally 语句块处理。

返回到我们上面提出的问题, 这段代码的返回值应该是多少? 熟悉 Java 语言的读者应该很容易就能说出答案: 如果没有出现异常, 则返回值是 1; 如果出现了 Exception 异常, 则返回值是 2; 如果出现了 Exception 以外的异常, 则方法非正常退出, 没有返回值。我们一起来分析一下字节码的执行过程, 从字节码的层面上来看看为何会有这样的返回结果。

字节码中第 0 至 4 行所做的操作就是将整数 1 赋值给变量 x, 并且将此时 x 的值复制一份副本到最后一个本地变量表的 Slot 中 (这个 Slot 里面的值在 ireturn 指令执行前将会被重新读到操作栈顶, 作为方法返回值来使用。为了讲解方便, 笔者给这个 Slot 取名为: returnValue)。如果这时候没有出现异常, 则会继续走到第 5 至 9 行, 将变量 x 赋值为 3, 然后将之前保存在 returnValue 中的整数 1 读入到操作栈顶, 最后 ireturn 指令会以 int 的形式返回操作栈顶中的值, 方法结束。如果出现了异常, PC 寄存器指针转到

第 10 行，第 10 至 20 行所做的事情是将 2 赋值给变量 x，然后将变量 x 此时的值赋给 returnValue，最后再将变量 x 的值改为 3。方法返回前同样是将 returnValue 中保留的整数 2 读到操作栈顶。从第 21 行开始的代码，作用是将变量 x 的值赋为 3，并将栈顶的异常抛出，方法结束。

尽管大家都知道这段代码出现异常的概率非常的小，但是并不影响它为我们演示异常表的作用。如果大家到这里仍然对字节码的运作过程比较模糊，其实也不要紧，关于虚拟机执行字节码的过程，本书第 8 章“虚拟机字节码执行引擎”中将会有更详细的讲解。

2. Exceptions 属性

这里的 Exceptions 属性是在方法表中与 Code 属性同级的一项属性，读者不要与前面刚刚讲解完的异常表产生混淆。Exceptions 属性的作用是列举出方法中可能抛出的受查异常 (Checked Exceptions)，也就是方法描述时在 throws 关键字后面列举的异常。它的结构如表 6-17 所示。

表 6-17 属性表结构

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

此属性中的 number_of_exceptions 项表示方法可能抛出 number_of_exceptions 种受查异常，每一种受查异常使用一个 exception_index_table 项表示，exception_index_table 是一个指向常量池中 CONSTANT_Class_info 型常量的索引，代表了该受查异常的类型。

3. LineNumberTable 属性

LineNumberTable 属性用于描述 Java 源码行号与字节码行号 (字节码的偏移量) 之间的对应关系。它并不是运行时必需的属性，但默认会生成到 Class 文件之中，可以在 Javac 中使用 -g:none 或 -g:lines 选项来取消或要求生成这项信息。如果选择不生成 LineNumberTable 属性，对程序运行产生的最主要的影响就是在抛出异常时，堆栈中将不会显示出错的行号，并且在调试程序的时候无法按照源码来设置断点。LineNumberTable 属性的结构如表 6-18 所示。

表 6-18 LineNumberTable 属性结构

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

line_number_table 是一个数量为 line_number_table_length、类型为 line_number_info 的集合，line_number_info 表包括了 start_pc 和 line_number 两个 u2 类型的数据项，前者是字节码行号，后者是 Java 源码行号。

4. LocalVariableTable 属性

LocalVariableTable 属性用于描述栈帧中局部变量表中的变量与 Java 源码中定义的变量之间的关系，它不是运行时必需的属性，默认也不会生成到 Class 文件之中，可以在 Javac 中使用 -g:none 或 -g:vars 选项来取消或要求生成这项信息。如果没有生成这项属性，最大的影响就是当其他人引用这个方法时，所有的参数名称都将丢失，IDE 可能会使用诸如 arg0、arg1 之类的占位符来代替原有的参数名，这对程序运行没有影响，但是会给代码编写带来较大的不便，而且在调试期间调试器无法根据参数名称从运行上下文中获得参数值。LocalVariableTable 属性的结构如表 6-19 所示。

表 6-19 LocalVariableTable 属性结构

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

其中 local_variable_info 项目代表了一个栈帧与源码中的局部变量的关联，结构如表 6-20 所示。

start_pc 和 length 属性分别代表了这个局部变量的生命周期开始的字节码偏移量及其作用范围覆盖的长度，两者结合起来就是这个局部变量在字节码之中的作用域范围。

name_index 和 descriptor_index 都是指向常量池中 CONSTANT_Utf8_info 型常量的索引，分别代表了局部变量的名称及该局部变量的描述符。

表 6-20 local_variable_info 项目结构

类 型	名 称	数 量
u2	start_pc	1
u2	length	1
u2	name_index	1
u2	descriptor_index	1
u2	index	1

index 是这个局部变量在栈帧局部变量表中 Slot 的位置。当这个变量的数据类型是 64 位类型时 (double 和 long)，它占用的 Slot 为 index 和 index+1 两个位置。

顺便提一下，在 JDK 1.5 引入泛型之后，LocalVariableTable 属性增加了一个“姐妹属性”：LocalVariableTypeTable，这个新增的属性结构与 LocalVariableTable 非常相似，仅仅是把记录的字段描述符的 descriptor_index 替换成了字段的特征签名 (Signature)，对于非泛型类型来说，描述符和特征签名能描述的信息基本是一致的，但是引入泛型之后，由于描述符中泛型的参数化类型被擦除掉^①了，描述符就不能准确地描述泛型类型了，因此出现了 LocalVariableTypeTable。

5. SourceFile 属性

SourceFile 属性用于记录生成这个 Class 文件的源码文件名称。这个属性也是可选的，可以使用 Javac 的 -g:none 或 -g:source 选项来关闭或要求生成这项信息。在 Java 中，对于大多数的类来说，类名和文件名是一致的，但是有一些特殊情况（如内部类）例外。如果不生成这项属性，当抛出异常时，堆栈中将不会显示出错误代码所属的文件名。这个属性是一个定长的属性，其结构如表 6-21 所示。

表 6-21 SourceFile 属性结构

类 型	名 称	数 量
u2	attribute_name_index	1
u4	attribute_length	1
u2	sourcefile_index	1

sourcefile_index 数据项是指向常量池中 CONSTANT_Utf8_info 型常量的索引，常量值是源码文件的文件名。

^① 详见第 10 章中关于语法糖部分的内容。

6. ConstantValue 属性

ConstantValue 属性的作用是通知虚拟机自动为静态变量赋值。只有被 static 关键字修饰的变量（类变量）才可以使用这项属性。在 Java 程序里面类似“int x = 123”和“static int x = 123”这样的变量定义是非常常见的事情，但虚拟机对这两种变量赋值的方式和时刻都有所不同。对于非 static 类型的变量（也就是实例变量）的赋值是在实例构造器 <init> 方法中进行的；而对于类变量，则有两种方式可以选择：赋值在类构造器 <clinit> 方法中进行，或者使用 ConstantValue 属性来赋值。目前 Sun Javac 编译器的选择是：如果同时使用 final 和 static 来修饰一个变量（或者说常量更贴切），并且这个变量的数据类型是基本类型或 java.lang.String 的话，就生成 ConstantValue 属性来进行初始化，如果这个变量没有被 final 修饰，或者并非基本类型及字符串，则选择在 <clinit> 方法中进行初始化。

虽然有 final 关键字才更符合“ConstantValue”的语义，但虚拟机规范中并没有强制要求字段必须设置了 ACC_FINAL 标志，只要求了有 ConstantValue 属性的字段必须设置 ACC_STATIC 标志，对 final 关键字的要求是 Javac 编译器自己加入的限制。而 ConstantValue 的属性值则只限于基本类型和 String，不过笔者不认为这是什么限制，因为此属性的属性值只是一个常量池的索引号，由于 Class 文件格式的常量类型中只有与基本属性和字符串相对应的字面量，所以就算 ConstantValue 属性想支持别的类型也无能为力，ConstantValue 属性的结构如表 6-22 所示。

表 6-22 ConstantValue 属性结构

类 型	名 称	数 量
u2	attribute_name_index	1
u4	attribute_length	1
u2	constantvalue_index	1

从数据结构中可以看出 ConstantValue 属性是一个定长属性，它的 attribute_length 数据项值必须固定为 2。constantvalue_index 数据项代表了常量池中一个字面量常量的引用，根据字段类型的不同，字面量可以是 CONSTANT_Long_info、CONSTANT_Float_info、CONSTANT_Double_info、CONSTANT_Integer_info 和 CONSTANT_String_info 常量中的一种。

7. InnerClasses 属性

InnerClasses 属性用于记录内部类与宿主类之间的关联。如果一个类中定义了内部类，那编译器将会为它及它所包含的内部类生成 InnerClasses 属性。属性的结构如表 6-23 所示。

表 6-23 InnerClasses 属性结构

类 型	名 称	数 量
u2	attribute_name_index	1
u4	attribute_length	1
u2	number_of_classes	1
inner_classes_info	inner_classes	number_of_classes

数据项 number_of_classes 代表需要记录多少个内部类信息，每一个内部类的信息都由一个 inner_classes_info 表进行描述。inner_classes_info 表的结构如表 6-24 所示。

表 6-24 inner_classes_info 表的结构

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

inner_class_info_index 和 outer_class_info_index 都是指向常量池中 CONSTANT_Class_info 型常量的索引，分别代表了内部类和宿主类的符号引用。

inner_name_index 是指向常量池中 CONSTANT_Utf8_info 型常量的索引，代表这个内部类的名称，如果是匿名内部类，则这项值为 0。

inner_class_access_flags 是内部类的访问标志，类似于类的 access_flags，它的取值范围如表 6-25 所示。

8. Deprecated 及 Synthetic 属性

Deprecated 和 Synthetic 两个属性都属于标志类型的布尔属性，只存在有和没有的区别，没有属性值的概念。

Deprecated 属性用于表示某个类、字段或方法，已经被程序作者定为不再推荐使用，它可以通过在代码中使用 @deprecated 注释进行设置。

表 6-25 inner_class_access_flags 标志

标志名称	标志值	含 义
ACC_PUBLIC	0x0001	内部类是否为 public
ACC_PRIVATE	0x0002	内部类是否为 private
ACC_PROTECTED	0x0004	内部类是否为 protected
ACC_STATIC	0x0008	内部类是否为 static
ACC_FINAL	0x0010	内部类是否为 final
ACC_INTERFACE	0x0020	内部类是否为 synchronized
ACC_ABSTRACT	0x0400	内部类是否为 abstract
ACC_SYNTHETIC	0x1000	内部类是否并非由用户代码产生的
ACC_ANNOTATION	0x2000	内部类是否是一个注解
ACC_ENUM	0x4000	内部类是否是一个枚举

Synthetic 属性代表此字段或方法并不是由 Java 源码直接产生的，而是由编译器自行添加的，在 JDK 1.5 之后，标识一个类、字段或方法是编译器自动产生的，也可以设置它们访问标志中的 ACC_SYNTHETIC 标志位，其中最典型的例子就是 Bridge Method。所有由非用户代码产生的类、方法及字段都应当至少设置 Synthetic 属性和 ACC_SYNTHETIC 标志位中的一项，唯一的例外是实例构造器 “<init>” 方法和类构造器 “<clinit>” 方法。

Deprecated 和 Synthetic 属性的结构非常简单，如表 6-26 所示。

表 6-26 Deprecated 及 Synthetic 属性的结构

类 型	名 称	数 量
u2	attribute_name_index	1
u4	attribute_length	1

其中 attribute_length 数据项的值必须为 0x00000000，因为没有任何属性值需要设置。

6.4 Class 文件结构的发展

Class 文件结构自 Java 虚拟机规范第一版订立以来，已经有十多年的历史。这十多年间，Java 技术体系有了翻天覆地的改变，JDK 的版本号已经从 1.0 提升到了 1.7。相对于语言、API 及 Java 技术体系中其他方面的变化，Class 文件结构一直处于一个相对比较稳定的状态，Class 文件的主体结构几乎没有发生过变化，对 Class 文件格式的改进都

集中在向访问标志、属性表这些在设计上本就可扩展的数据结构中添加内容。

如果以《Java 虚拟机规范（第2版）》为基准进行比较的话，在后续 JDK 的发展中，访问标志里新加入了 ACC_SYNTHETIC、ACC_ANNOTATION、ACC_ENUM、ACC_BRIDGE、ACC_VARARGS 共五个标志，这些标志的意义和作用在前面已经介绍过。

而属性表集合中，在 JDK 1.5 和 JDK 1.6 版本内一共增加了 10 项新的属性。这些属性的名称及具体作用可以参见表 6-27。

表 6-27 JDK 1.5 和 JDK 1.6 中添加的属性

属性名称	使用位置	含 义
StackMapTable	Code 属性	JDK 1.6 中添加的属性，为了加快 Class 文件的校验速度，把类型校验时需要用到的相关信息直接写入到了 Class 文件中，以前这些信息都是通过代码数据流分析得到的
EnclosingMethod	类	JDK 1.5 中添加的属性，当一个类为局部类或匿名类时，可通过这个属性来声明其访问范围
Signature	类、方法表、 字段表	JDK 1.5 中添加的属性，存储类、方法、字段的特征签名。JDK 1.5 引入泛型是 Java 语言的一个巨大进步，虽然使用了类型擦除手段以避免在字节码级别产生改变，但是元数据中的泛型信息仍然需要保留下来，而这种情况下描述符再无法精确地描述泛型信息，因此添加了这个特征签名属性
SourceDebugExtension	类	JDK 1.6 中添加的属性，SourceDebugExtension 属性用于存储额外的调试信息。譬如在进行 JSP 文件调试时，无法通过 Java 堆栈来定位到 JSP 文件的行号，JSR-45 规范为这些非 Java 语言编写，却需要编译成字节码并运行在 Java 虚拟机中的程序提供了一个进行调试的标准机制，使用 SourceDebugExtension 属性就可以用于存储这个标准所新加入的调试信息
LocalVariableTypeTable	类	JDK 1.5 中添加的属性，其作用在上文讲解 LocalVariableTable 属性的时候已经提到过，它使用特征签名代替描述符，是为了引入泛型语法之后能描述泛型参数化类型而添加的
RuntimeVisibleAnnotations	类、方法表、 字段表	JDK 1.5 中添加的属性，为动态注解提供支持。RuntimeVisibleAnnotations 属性的用于指明哪些注解是运行时（实际上运行时就是进行反射调用）可见的
RuntimeInvisibleAnnotations	类、方法表、 字段表	JDK 1.5 中添加的属性，与 RuntimeVisibleAnnotations 属性的作用刚好相反，用于指明哪些注解是运行时不可见的
RuntimeVisibleParameterAnnotations	方法表	JDK 1.5 中添加的属性，作用与 RuntimeVisibleAnnotations 属性类似，只不过作用对象为方法参数
RuntimeInvisibleParameterAnnotations	方法表	JDK 1.5 中添加的属性，作用与 RuntimeInvisibleAnnotations 属性类似，只不过作用对象为方法参数
AnnotationDefault	方法表	JDK 1.5 中添加的属性，用于记录注解类元素的默认值

上面这些属性大部分用于支持 Java 中许多新出现的语言特性，如枚举、变长参数、泛型、动态注解等。还有一些是为了支持性能改进和调试信息，譬如 JDK 1.6 的新类型校验器使用的 StackMapTable 属性和对非 Java 代码调试中用到的 SourceDebugExtension 属性。在本书后面的两章中，我们还会接触到这些新加入的高级特性。

6.5 本章小结

Class 文件是 Java 虚拟机执行引擎的数据入口，也是 Java 技术体系的基础支柱之一。了解 Class 文件的结构对后面进一步了解虚拟机执行引擎有很重要的意义。

本章详细讲解了 Class 文件结构中的各个组成部分，以及每个部分的定义、数据结构和使用方法。通过代码清单 6-1 的 Java 代码与它的 Class 文件样例，以实战的方式演示了 Class 的数据是如何存储和访问的。从下一章开始，我们将以动态的、运行时的角度去看看字节码流在虚拟机执行引擎中是怎样被解释执行的。



第 7 章 虚拟机类加载机制

本章主要内容

- 概述
- 类加载的时机
- 类加载的过程
- 类加载器

代码编译的结果从本地机器码转变为字节码，是存储格式发展的一小步，却是编程语言发展的一大步。

7.1 概述

上一章我们了解了 Class 文件存储格式的具体细节，在 Class 文件中描述的各种信息，最终都需要加载到虚拟机中之后才能被运行和使用。而虚拟机如何加载这些 Class 文件？Class 文件中的信息进入到虚拟机后会发生什么变化？这些都是本章将要讲解的内容。

虚拟机把描述类的数据从 Class 文件加载到内存，并对数据进行校验、转换解析和初始化，最终形成可以被虚拟机直接使用的 Java 类型，这就是虚拟机的类加载机制。

与那些在编译时需要进行连接工作的语言不同，在 Java 语言里面，类型的加载和连接过程都是在程序运行期间完成的，这样会在类加载时稍微增加一些性能开销，但是却能为 Java 应用程序提供高度的灵活性，Java 中天生可以动态扩展的语言特性就是依赖运行期动态加载和动态连接这个特点实现的。例如，如果编写一个使用接口的应用程序，可以等到运行时再指定其实际的实现。这种组装应用程序的方式广泛应用于 Java 程序之中。

为了避免语言表达中可能产生的偏差，在本章正式开始之前，笔者先设立两个语言上的约定：第一，在实际情况中，每个 Class 文件都有可能代表着 Java 语言中的一个类或接口，后文中直接对“类”的描述都包括了类和接口的可能性，而对于类和接口需要分开描述的场景会特别指明；第二，笔者所说的“Class 文件”并非指 Class 必须是存在于具体磁盘中的某个文件，这里说的 Class 文件指的是一串二进制的字节流，无论以何种形式存在都可以。

7.2 类加载的时机

类从被加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期包括了：加载（Loading）、验证（Verification）、准备（Preparation）、解析（Resolution）、初始化（Initialization）、使用（Using）和卸载（Unloading）七个阶段。其中验证、准备和解析三个部分统称为连接（Linking），这七个阶段的发生顺序如图 7-1 所示。

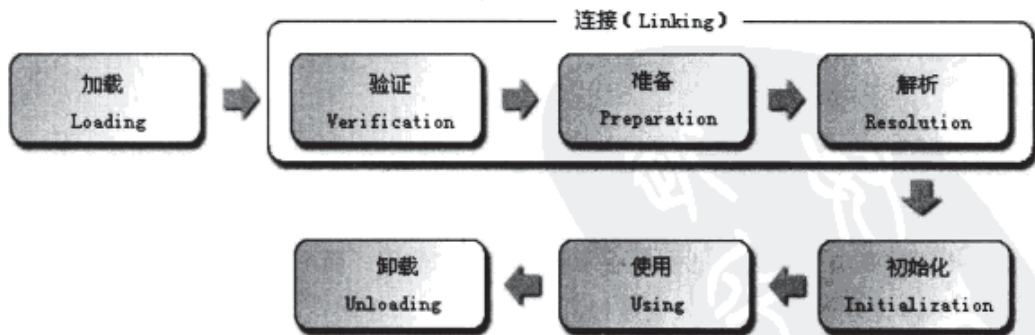


图 7-1 类的生命周期

图 7-1 中，加载、验证、准备、初始化和卸载这五个阶段的顺序是确定的，类的加载过程必须按照这种顺序按部就班地开始，而解析阶段则不一定：它在某些情况下可以在初始化阶段之后再开始，这是为了支持 Java 语言的运行时绑定（也称为动态绑定或晚期绑定）。请注意这里写的是按部就班地“开始”，而不是按部就班地“进行”或“完成”，因为这些阶段通常都是互相交叉地混合式进行的，通常会在一个阶段执行的过程中调用或激活另外一个阶段。

什么情况下需要开始类加载过程的第一个阶段：加载。虚拟机规范中并没有进行强制约束，这点可以交给虚拟机的具体实现来自由把握。但是对于初始化阶段，虚拟机规

范则是严格规定了有且只有四种情况必须立即对类进行“初始化”（而加载、验证、准备自然需要在此之前开始）：

1) 遇到 `new`、`getstatic`、`putstatic` 或 `invokestatic` 这 4 条字节码指令时，如果类没有进行过初始化，则需要先触发其初始化。生成这 4 条指令的最常见的 Java 代码场景是：使用 `new` 关键字实例化对象的时候、读取或设置一个类的静态字段（被 `final` 修饰、已在编译期把结果放入常量池的静态字段除外）的时候，以及调用一个类的静态方法的时候。

2) 使用 `java.lang.reflect` 包的方法对类进行发射调用的时候，如果类没有进行过初始化，则需要先触发其初始化。

3) 当初始化一个类的时候，如果发现其父类还没有进行过初始化，则需要先触发其父类的初始化。

4) 当虚拟机启动时，用户需要指定一个要执行的主类（包含 `main()` 方法的那个类），虚拟机会先初始化这个主类。

对于这四种会触发类进行初始化的场景，虚拟机规范中使用了一个很强烈的限定语：“有且只有”，这四种场景中的行为称为对一个类进行主动引用。除此之外所有引用类的方式，都不会触发初始化，称为被动引用。下面举三个例子来说明被动引用，分别见代码清单 7-1、代码清单 7-2 和代码清单 7-3。

代码清单 7-1 被动引用的例子之一

```
package org.fenixsoft.classloading;

/**
 * 被动使用类字段演示一：
 * 通过子类引用父类的静态字段，不会导致子类初始化
 */
public class SuperClass {

    static {
        System.out.println("SuperClass init!");
    }

    public static int value = 123;
}
```


174 ❖ 第三部分 虚拟机执行子系统

```
public class SubClass extends SuperClass {  
  
    static {  
        System.out.println("SubClass init!");  
    }  
}  
  
/**  
 * 非主动使用类字段演示  
 **/  
public class NotInitialization {  
  
    public static void main(String[] args) {  
        System.out.println(SubClass.value);  
    }  
}  
}
```

上述代码运行之后，只会输出“SuperClass init!”，而不会输出“SubClass init!”。对于静态字段，只有直接定义这个字段的类才会被初始化，因此通过其子类来引用父类中定义的静态字段，只会触发父类的初始化而不会触发子类的初始化。至于是否要触发子类的加载和验证，在虚拟机规范中并未明确规定，这点取决于虚拟机的具体实现。对于Sun HotSpot虚拟机来说，可通过-XX:+TraceClassLoading参数看到此操作会导致子类的加载。

代码清单 7-2 被动引用的例子之二

```
package org.fenixsoft.classloading;  
  
/**  
 * 被动使用类字段演示二：  
 * 通过数组定义来引用类，不会触发此类的初始化  
 **/  
public class NotInitialization {  
  
    public static void main(String[] args) {  
        SuperClass[] sca = new SuperClass[10];  
    }  
}  
}
```

这段代码为了节省版面，复用了代码清单 7-1 中的 SuperClass，运行之后发现没有输出“SuperClass init!”，说明并没有触发类 org.fenixsoft.classloading.SuperClass 的初始化阶段。但是这段代码里面触发了另外一个名为“[Lorg.fenixsoft.classloading.SuperClass”的类的初始化阶段，对于用户代码来说，这并不是一个合法的类名称，它是一个由虚拟机自动生成的、直接继承于 java.lang.Object 的子类，创建动作由字节码指令 newarray 触发。

这个类代表了一个元素类型为 org.fenixsoft.classloading.SuperClass 的一维数组，数组中应有的属性和方法（用户可直接使用的只有被修饰为 public 的 length 属性和 clone() 方法）都实现在这个类里。Java 语言中对数组的访问比 C/C++ 相对安全，因为这个类包装了数组元素的访问方法^①，而 C/C++ 直接翻译为对数组指针的移动。在 Java 语言中，当检查到发生数组越界时会抛出 java.lang.ArrayIndexOutOfBoundsException 异常。

代码清单 7-3 被动引用的例子之三

```
package org.fenixsoft.classloading;

/**
 * 被动使用类字段演示三：
 * 常量在编译阶段会存入调用类的常量池中，本质上没有直接引用到定义常量的类，因此不会触发定义常量的类的初始化。
 */
public class ConstClass {

    static {
        System.out.println("ConstClass init!");
    }

    public static final String HELLOWORLD = "hello world";
}

/**
 * 非主动使用类字段演示
 */
public class NotInitialization {
```

^① 准确地说，越界检查并不是封装在数组元素访问的类中，而是封装在数组访问的 xaload、xastore 字节码指令中。

```
public static void main(String[] args) {  
    System.out.println(ConstClass.HELLOWORLD);  
}
```

上述代码运行之后，也没有输出“ConstClass init!”，这是因为虽然在 Java 源码中引用了 ConstClass 类中的常量 HELLOWORLD，但是在编译阶段将此常量的值“hello world”存储到了 NotInitialization 类的常量池中，对常量 ConstClass. HELLOWORLD 的引用实际都被转化为 NotInitialization 类对自身常量池的引用了。也就是说实际上 NotInitialization 的 Class 文件之中并没有 ConstClass 类的符号引用入口，这两个类在编译成 Class 之后就不存在任何联系了。

接口的加载过程与类加载过程稍有一些不同，针对接口需要做一些特殊说明：接口也有初始化过程，这点与类是一致的，上面的代码都是用静态语句块“static {}”来输出初始化信息的，而接口中不能使用“static {}”语句块，但编译器仍然会为接口生成“<clinit>()”类构造器^①，用于初始化接口中所定义的成员变量。接口与类真正有所区别的是前面讲述的四种“有且仅有”需要开始初始化场景中的第三种：当一个类在初始化时，要求其父类全部都初始化过了，但是一个接口在初始化时，并不要求其父接口全部都完成了初始化，只有在真正使用到父接口的时候（如引用接口中定义的常量）才会初始化。

7.3 类加载的过程

接下来我们详细讲解一下类加载的全过程，也就是加载、验证、准备、解析和初始化这五个阶段的过程。

7.3.1 加载

“加载”（Loading）阶段是“类加载”（Class Loading）过程的一个阶段，希望您没有混淆这两个看起来相似的名词。在加载阶段，虚拟机需要完成以下三件事情：

- 1) 通过一个类的全限定名来获取定义此类的二进制字节流。
- 2) 将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构。

^① 关于类构造器 <clinit> 和方法构造器 <init> 的生成过程和作用，可参见第 10 章的相关内容。

3) 在 Java 堆中生成一个代表这个类的 `java.lang.Class` 对象，作为方法区这些数据的访问入口。

虚拟机规范的这三点要求实际上并不具体，因此虚拟机实现与具体应用的灵活度相当大。例如“通过一个类的全限定名来获取定义此类的二进制字节流”，并没有指明二进制字节流要从一个 `Class` 文件中获取，准确地说是根本没有指明要从哪里获取及怎样获取。虚拟机设计团队在加载阶段搭建了一个相当开放的、广阔的舞台，Java 发展历程中，充满创造力的开发人员们则在这个舞台上玩出了各种花样，许多举足轻重的 Java 技术都建立在这一基础之上，例如：

- 从 ZIP 包中读取，这很常见，最终成为日后 JAR、EAR、WAR 格式的基础。
- 从网络中获取，这种场景最典型的应用就是 Applet。
- 运行时计算生成，这种场景使用得最多的就是动态代理技术，在 `java.lang.reflect.Proxy` 中，就是用了 `ProxyGenerator.generateProxyClass` 来为特定接口生成 `*$Proxy` 的代理类的二进制字节流。
- 由其他文件生成，典型场景：JSP 应用。
- 从数据库中读取，这种场景相对少见些，有些中间件服务器（如 SAP Netweaver）可以选择把程序安装到数据库中来完成程序代码在集群间的分发。
- ……

相对于类加载过程的其他阶段，加载阶段（准确地说，是加载阶段中获取类的二进制字节流的动作）是开发期可控性最强的阶段，因为加载阶段既可以使用系统提供的类加载器来完成，也可以由用户自定义的类加载器去完成，开发人员们可以通过定义自己的类加载器去控制字节流的获取方式。关于类加载器的话题，本章将在 7.4 节专门讲述。

加载阶段完成后，虚拟机外部的二进制字节流就按照虚拟机所需的格式存储在方法区之中，方法区中的数据存储格式由虚拟机实现自行定义，虚拟机规范未规定此区域的具体数据结构。然后在 Java 堆中实例化一个 `java.lang.Class` 类的对象，这个对象将作为程序访问方法区中的这些类型数据的外部接口。加载阶段与连接阶段的部分内容（如一部分字节码文件格式验证动作）是交叉进行的，加载阶段尚未完成，连接阶段可能已经开始，但这些夹在加载阶段之中进行的动作，仍然属于连接阶段的内容，这两个阶段的开始时间仍然保持着固定的先后顺序。

7.3.2 验证

验证是连接阶段的第一步，这一阶段的目的是为了确保证 Class 文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。

Java 语言本身是相对安全的语言（依然是相对于 C/C++ 来说），使用纯粹的 Java 代码无法做到诸如访问数组边界以外的数据、将一个对象转型为它并未实现的类型、跳转到不存在的代码行之类的事情，如果这样做了，编译器将拒绝编译。但前面已经说过，Class 文件并不一定要求用 Java 源码编译而来，可以使用任何途径，包括用十六进制编辑器直接编写来产生 Class 文件。在字节码的语言层面上，上述 Java 代码无法做到的事情都是可以实现的，至少语义上是可以表达出来的。虚拟机如果不检查输入的字节流，对其完全信任的话，很可能会因为载入了有害的字节流而导致系统崩溃，所以验证是虚拟机对自身保护的一项重要工作。

尽管验证阶段是非常重要的，并且验证阶段的工作量在虚拟机的类加载子系统中占了很大一部分，但虚拟机规范对这个阶段的限制和指导显得非常笼统，仅仅说了一句如果验证到输入的字节流不符合 Class 文件的存储格式，就抛出一个 `java.lang.VerifyError` 异常或其子类异常，具体应当检查哪些方面，如何检查，何时检查，都没有强制要求或明确说明，所以不同的虚拟机对类验证的实现可能会有所不同，但大致上都会完成下面四个阶段的检验过程：文件格式验证、元数据验证、字节码验证和符号引用验证。

1. 文件格式验证

第一阶段要验证字节流是否符合 Class 文件格式的规范，并且能被当前版本的虚拟机处理。这一阶段可能包括下面这些验证点：

- 是否以魔数 `0xCAFEBABE` 开头。
- 主、次版本号是否在当前虚拟机处理范围之内。
- 常量池的常量中是否有不被支持的常量类型（检查常量 `tag` 标志）。
- 指向常量的各种索引值中是否有指向不存在的常量或不符合类型的常量。
- `CONSTANT_Utf8_info` 型的常量中是否有不符合 UTF8 编码的数据。
- Class 文件中各个部分及文件本身是否有被删除的或附加的其他信息。
-

实际上第一阶段的验证点还远不止这些，上面这些只是从 HotSpot 虚拟机源码^①中摘抄的一小部分，该验证阶段的主要目的是保证输入的字节流能正确地解析并存储于方法区之内，格式上符合描述一个 Java 类型信息的要求。这阶段的验证是基于字节流进行的，经过了这个阶段的验证之后，字节流才会进入内存的方法区中进行存储，所以后面的三个验证阶段全部是基于方法区的存储结构进行的。

2. 元数据验证

第二阶段是对字节码描述的信息进行语义分析，以保证其描述的信息符合 Java 语言规范的要求，这个阶段可能包括的验证点如下：

- 这个类是否有父类（除了 `java.lang.Object` 之外，所有的类都应当有父类）。
- 这个类的父类是否继承了不允许被继承的类（被 `final` 修饰的类）。
- 如果这个类不是抽象类，是否实现了其父类或接口之中要求实现的所有方法。
- 类中的字段、方法是否与父类产生了矛盾（例如覆盖了父类的 `final` 字段，或者出现不符合规则的方法重载，例如方法参数都一致，但返回值类型却不同等）。
- ……

第二阶段的主要目的是对类的元数据信息进行语义校验，保证不存在不符合 Java 语言规范的元数据信息。

3. 字节码验证

第三阶段是整个验证过程中最复杂的一个阶段，主要工作是进行数据流和控制流分析。在第二阶段对元数据信息中的数据类型做完校验后，这阶段将对类的方法体进行校验分析。这阶段的任务是保证被校验类的方法在运行时不会做出危害虚拟机安全的行为，例如：

- 保证任意时刻操作数栈的数据类型与指令代码序列都能配合工作，例如不会出现类似这样的情况：在操作栈中放置了一个 `int` 类型的数据，使用时却按 `long` 类型来加载入本地变量表中。
- 保证跳转指令不会跳转到方法体以外的字节码指令上。
- 保证方法体中的类型转换是有效的，例如可以把一个子类对象赋值给父类数据类型，这是安全的，但是把父类对象赋值给子类数据类型，甚至把对象赋值给与它

^① 源码位置：`hotspot/src/share/vm/classfile/classFileParser.cpp`。

毫无继承关系、完全不相干的一个数据类型，则是危险和不合法的。

□ ……

如果一个类方法体的字节码没有通过字节码验证，那肯定是有问题的；但如果一个方法体通过了字节码验证，也不能说明其一定就是安全的。即使字节码验证之中进行了大量的检查，也不能保证这一点。这里涉及了离散数学中一个很著名的问题“Halting Problem”^①：通俗一点的说法就是，通过程序去校验程序逻辑是无法做到绝对准确的——不能通过程序准确地检查出程序是否能在有限的时间之内结束运行。

由于数据流验证的高复杂性，虚拟机设计团队为了避免将过多的时间消耗在字节码验证阶段，在JDK 1.6之后的Javac编译器中进行了一项优化，给方法体的Code属性的属性表中增加了一项名为“StackMapTable”的属性，这项属性描述了方法体中所有的基本块（Basic Block，按照控制流拆分的代码块）开始时本地变量表和操作栈应有的状态，这可以将字节码验证的类型推导转变为类型检查从而节省一些时间。当然，理论上StackMapTable属性也存在错误或被篡改的可能，所以是否有可能在恶意篡改了Code属性的同时，也生成相应的StackMapTable属性来骗过虚拟机的类型校验则是虚拟机实现时值得思考的问题。

在JDK 1.6的HotSpot虚拟机中提供了-XX:-UseSplitVerifier选项来关闭掉这项优化，或者使用参数-XX:+FailOverToOldVerifier要求在类型校验失败的时候退回到旧的类型推导方式进行校验。而在JDK 1.7之后，对于主版本号大于50的Class文件，使用类型检查来完成数据流分析校验则是唯一的选择，不允许再退回到类型推导的校验方式。

4. 符号引用验证

最后一个阶段的校验发生在虚拟机将符号引用转化为直接引用的时候，这个转化动作将在连接的第三个阶段——解析阶段中发生。符号引用验证可以看做是对类自身以外（常量池中的各种符号引用）的信息进行匹配性的校验，通常需要校验以下内容：

□ 符号引用中通过字符串描述的全限定名是否能找到对应的类。

^① 停机问题就是判断任意一个程序是否会在有限的时间之内结束运行的问题。如果这个问题可以在有限的时间之内解决，则可以有一个程序判断其本身是否会停机并做出相反的行为。这时候显然不管停机问题的结果是什么都不会符合要求。所以这是一个不可解的问题。具体的证明过程可参考：<http://zh.wikipedia.org/zh/停机问题>。

- 在指定类中是否存在符合方法的字段描述符及简单名称所描述的方法和字段。
- 符号引用中的类、字段和方法的访问性（private、protected、public、default）是否可被当前类访问。
-

符号引用验证的目的是确保解析动作能正常执行，如果无法通过符号引用验证，将会抛出一个 `java.lang.IncompatibleClassChangeError` 异常的子类，如 `java.lang.IllegalAccessError`、`java.lang.NoSuchFieldError`、`java.lang.NoSuchMethodError` 等。

验证阶段对于虚拟机的类加载机制来说，是一个非常重要的、但不一定是必要的阶段。如果所运行的全部代码（包括自己写的、第三方包中的代码）都已经被反复使用和验证过，在实施阶段就可以考虑使用 `-Xverify:none` 参数来关闭大部分的类型验证措施，以缩短虚拟机类加载的时间。

7.3.3 准备

准备阶段是正式为类变量分配内存并设置类变量初始值的阶段，这些内存都将在方法区中进行分配。这个阶段中有两个容易产生混淆的概念需要强调一下，首先是这时候进行内存分配的仅包括类变量（被 `static` 修饰的变量），而不包括实例变量，实例变量将会在对象实例化时随着对象一起分配在 Java 堆中。其次是这里所说的初始值“通常情况”下是数据类型的零值，假设一个类变量的定义为：

```
public static int value = 123;
```

那么变量 `value` 在准备阶段过后的初始值为 0 而不是 123，因为这时候尚未开始执行任何 Java 方法，而把 `value` 赋值为 123 的 `putstatic` 指令是程序被编译后，存放于类构造器 `<clinit>()` 方法之中，所以把 `value` 赋值为 123 的动作将在初始化阶段才会被执行。表 7-1 列出了 Java 中所有基本数据类型的零值。

上面提到，在“通常情况”下初始值是零值，那相对的会有一些“特殊情况”：如果类字段的字段属性表中存在 `ConstantValue` 属性，那在准备阶段变量 `value` 就会被初始化为 `ConstantValue` 属性所指定的值，假设上面类变量 `value` 的定义变为：

```
public static final int value = 123;
```

编译时 `Javac` 将会为 `value` 生成 `ConstantValue` 属性，在准备阶段虚拟机就会根据 `ConstantValue` 的设置将 `value` 赋值为 123。

表 7-1 基本数据类型的零值

数据类型	零 值
int	0
long	0L
short	(short) 0
char	'\u0000'
byte	(byte) 0
boolean	false
float	0.0f
double	0.0d
reference	null

7.3.4 解析

解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程，符号引用在前一章讲解 Class 文件格式的时候就已经出现过多次，在 Class 文件中它以 `CONSTANT_Class_info`、`CONSTANT_Fieldref_info`、`CONSTANT_Methodref_info` 等类型的常量出现，那解析阶段中所说的直接引用与符号引用又有什么关联呢？

- 符号引用 (Symbolic References)：符号引用以一组符号来描述所引用的目标，符号可以是任何形式的字面量，只要使用时能无歧义地定位到目标即可。符号引用与虚拟机实现的内存布局无关，引用的目标并不一定已经加载到内存中。
- 直接引用 (Direct References)：直接引用可以是直接指向目标的指针、相对偏移量或是一个能间接定位到目标的句柄。直接引用是与虚拟机实现的内存布局相关的，同一个符号引用在不同虚拟机实例上翻译出来的直接引用一般不会相同。如果有了直接引用，那引用的目标必定已经在内存中存在。

虚拟机规范之中并未规定解析阶段发生的具体时间，只要求在执行 `anewarray`、`checkcast`、`getfield`、`getstatic`、`instanceof`、`invokeinterface`、`invokespecial`、`invokestatic`、`invokevirtual`、`multianewarray`、`new`、`putfield` 和 `putstatic` 这 13 个用于操作符号引用的字节码指令之前，先对它们所使用的符号引用进行解析。所以虚拟机实现会根据需要来判断，到底是在类被加载器加载时就对常量池中的符号引用进行解析，还是等到一个符号引用将要被使用前才去解析它。

对同一个符号引用进行多次解析请求是很常见的事情，虚拟机实现可能会对第一次

解析的结果进行缓存（在运行时常量池中记录直接引用，并把常量标识为已解析状态）从而避免解析动作重复进行。无论是否真正执行了多次解析动作，虚拟机需要保证的都是同一个实体中，如果一个符号引用之前已经被成功解析过，那么后续的引用解析请求就应当一直成功；同样地，如果第一次解析失败了，其他指令对这个符号的解析请求也应该收到相同的异常。

解析动作主要针对类或接口、字段、类方法、接口方法四类符号引用进行，分别对应于常量池的 `CONSTANT_Class_info`、`CONSTANT_Fieldref_info`、`CONSTANT_Methodref_info` 及 `CONSTANT_InterfaceMethodref_info` 四种常量类型^①。下面将讲解这四种引用的解析过程。

1. 类或接口的解析

假设当前代码所处的类为 D，如果要把一个从未解析过的符号引用 N 解析为一个类或接口 C 的直接引用，那虚拟机完成整个解析的过程需要包括以下 3 个步骤：

1) 如果 C 不是一个数组类型，那虚拟机将会把代表 N 的全限定名传递给 D 的类加载器去加载这个类 C。在加载过程中，由于无数据验证、字节码验证的需要，又将可能触发其他相关类的加载动作，例如加载这个类的父类或实现的接口。一旦这个加载过程出现了任何异常，解析过程就将宣告失败。

2) 如果 C 是一个数组类型，并且数组的元素类型为对象，也就是 N 的描述符会是类似 “[Ljava.lang.Integer” 的形式，那将会按照第 1 点的规则加载数组元素类型。如果 N 的描述符如前面所假设的形式，需要加载的元素类型就是 “java.lang.Integer”，接着由虚拟机生成一个代表此数组维度和元素的数组对象。

3) 如果上面的步骤没有出现任何异常，那么 C 在虚拟机中实际上已经成为一个有效的类或接口了，但在解析完成之前还要进行符号引用验证，确认 C 是否具备对 D 的访问权限。如果发现不具备访问权限，将抛出 `java.lang.IllegalAccessError` 异常。

2. 字段解析

要解析一个未被解析过的字段符号引用，首先将会对字段表内 `class_index`^② 项中索

^① 严格来说，`CONSTANT_String_info` 和 `CONSTANT_InterfaceMethodref_info` 这两种类型的常量也有解析过程。

^② 参见第 6 章中关于 `CONSTANT_Fieldref_info` 常量的相关内容。

引的 `CONSTANT_Class_info` 符号引用进行解析，也就是字段所属的类或接口的符号引用。如果在解析这个类或接口符号引用的过程中出现了任何异常，都会导致字段符号引用解析的失败。如果解析成功完成，那将把这个字段所属的类或接口用 `C` 表示，虚拟机规范要求按照如下步骤对 `C` 进行后续字段的搜索：

1) 如果 `C` 本身就包含了简单名称和字段描述符都与目标相匹配的字段，则返回这个字段的直接引用，查找结束。

2) 否则，如果在 `C` 中实现了接口，将会按照继承关系从上往下递归搜索各个接口和它的父接口，如果接口中包含了简单名称和字段描述符都与目标相匹配的字段，则返回这个字段的直接引用，查找结束。

3) 否则，如果 `C` 不是 `java.lang.Object` 的话，将会按照继承关系从上往下递归搜索其父类，如果在父类中包含了简单名称和字段描述符都与目标相匹配的字段，则返回这个字段的直接引用，查找结束。

4) 否则，查找失败，抛出 `java.lang.NoSuchFieldError` 异常。

如果查找过程成功返回了引用，将会对这个字段进行权限验证，如果发现不具备对字段的访问权限，将抛出 `java.lang.IllegalAccessError` 异常。

在实际应用中，虚拟机的编译器实现可能会比上述规范要求得更加严格一些，如果有一个同名字段同时出现在 `C` 的接口和父类中，或者同时在自己或父类的多个接口中出现，那编译器将可能拒绝编译。在代码清单 7-4 中，如果注释了 `Sub` 类中的“`public static int A = 4;`”，接口与父类同时存在字段 `A`，那编译器将提示“`The field Sub.A is ambiguous`”，并且会拒绝编译这段代码。

代码清单 7-4 字段解析

```
package org.fenixsoft.classloading;

public class FieldResolution {

    interface Interface0 {
        int A = 0;
    }

    interface Interface1 extends Interface0 {
        int A = 1;
    }
}
```

```
interface Interface2 {
    int A = 2;
}

static class Parent implements Interface1 {
    public static int A = 3;
}

static class Sub extends Parent implements Interface2 {
    public static int A = 4;
}

public static void main(String[] args) {
    System.out.println(Sub.A);
}
}
```

3. 类方法解析

类方法解析的第一个步骤与字段解析一样，也是需要先解析出类方法表的 `class_index`^①项中索引的方法所属的类或接口的符号引用，如果解析成功，我们依然用 `C` 表示这个类，接下来虚拟机将会按照如下步骤进行后续的分类方法搜索：

- 1) 类方法和接口方法符号引用的常量类型定义是分开的，如果在类方法表中发现 `class_index` 中索引的 `C` 是个接口，那就直接抛出 `java.lang. IncompatibleClassChangeError` 异常。

- 2) 如果通过了第(1)步，在类 `C` 中查找是否有简单名称和描述符都与目标相匹配的方法，如果有则返回这个方法的直接引用，查找结束。

- 3) 否则，在类 `C` 的父类中递归查找是否有简单名称和描述符都与目标相匹配的方法，如果有则返回这个方法的直接引用，查找结束。

- 4) 否则，在类 `C` 实现的接口列表及它们的父接口之中递归查找是否有简单名称和描述符都与目标相匹配的方法，如果存在匹配的方法，说明类 `C` 是一个抽象类，这时候查找结束，抛出 `java.lang. AbstractMethodError` 异常。

- 5) 否则，宣告方法查找失败，抛出 `java.lang. NoSuchMethodError`。

① 参见第6章关于 `CONSTANT_Methodref_info` 常量的相关内容。

最后，如果查找过程成功返回了直接引用，将会对这个方法进行权限验证；如果发现不具备对此方法的访问权限，将抛出 `java.lang.IllegalAccessError` 异常。

4. 接口方法解析

接口方法也是需要先解析出接口方法表的 `class_index`^①项中索引的方法所属的类或接口的符号引用，如果解析成功，依然用 `C` 表示这个接口，接下来虚拟机将会按照如下步骤进行后续的接口方法搜索：

1) 与类方法解析相反，如果在接口方法表中发现 `class_index` 中的索引 `C` 是个类而不是接口，那就直接抛出 `java.lang.IncompatibleClassChangeError` 异常。

2) 否则，在接口 `C` 中查找是否有简单名称和描述符都与目标相匹配的方法，如果有则返回这个方法的直接引用，查找结束。

3) 否则，在接口 `C` 的父接口中递归查找，直到 `java.lang.Object` 类（查找范围会包括 `Object` 类）为止，看是否有简单名称和描述符都与目标相匹配的方法，如果有则返回这个方法的直接引用，查找结束。

4) 否则，宣告方法查找失败，抛出 `java.lang.NoSuchMethodError` 异常。

由于接口中的所有方法都默认是 `public` 的，所以不存在访问权限的问题，因此接口方法的符号解析应当不会抛出 `java.lang.IllegalAccessError` 异常。

7.3.5 初始化

类初始化阶段是类加载过程的最后一步，前面的类加载过程中，除了在加载阶段用户应用程序可以通过自定义类加载器参与之外，其余动作完全由虚拟机主导和控制。到了初始化阶段，才真正开始执行类中定义的 Java 程序代码（或者说是字节码）。

在准备阶段，变量已经赋过一次系统要求的初始值，而在初始化阶段，则是根据程序员通过程序制定的主观计划去初始化类变量和其他资源，或者可以从另外一个角度来表达：初始化阶段是执行类构造器 `<clinit>()` 方法的过程。我们放到后面再讲 `<clinit>()` 方法是怎么生成的，在这里，我们先看一下 `<clinit>()` 方法执行过程中可能会影响程序运行行为的一些特点和细节，这部分相对更贴近于普通的程序开发人员^②：

① 参见第 6 章关于 `CONSTANT_InterfaceMethodref_info` 常量的相关内容。

② 这里只限于 Java 语言编译产生的 `class` 文件，并不包括其他 JVM 语言。

- <clinit>() 方法是由编译器自动收集类中的所有类变量的赋值动作和静态语句块 (static {} 块) 中的语句合并产生的, 编译器收集的顺序是由语句在源文件中出现的顺序所决定的, 静态语句块中只能访问到定义在静态语句块之前的变量, 定义在它之后的变量, 在前面的静态语句块中可以赋值, 但是不能访问。
- <clinit>() 方法与类的构造函数 (或者说实例构造器 <init>() 方法) 不同, 它不需要显式地调用父类构造器, 虚拟机会保证在子类的 <clinit>() 方法执行之前, 父类的 <clinit>() 方法已经执行完毕。因此在虚拟机中第一个被执行的 <clinit>() 方法的类肯定是 java.lang.Object。
- 由于父类的 <clinit>() 方法先执行, 也就意味着父类中定义的静态语句块要优先于子类的变量赋值操作, 如代码清单 7-5 中, 字段 B 的值将会是 2 而不是 1。

代码清单 7-5 <clinit>() 方法执行顺序

```
static class Parent {
    public static int A = 1;
    static {
        A = 2;
    }
}

static class Sub extends Parent {
    public static int B = A;
}

public static void main(String[] args) {
    System.out.println(Sub.B);
}
```

- <clinit>() 方法对于类或接口来说并不是必须的, 如果一个类中没有静态语句块, 也没有对变量的赋值操作, 那么编译器可以不为这个类生成 <clinit>() 方法。
- 接口中不能使用静态语句块, 但仍然有变量初始化的赋值操作, 因此接口与类一样都会生成 <clinit>() 方法。但接口与类不同的是, 执行接口的 <clinit>() 方法不需要先执行父接口的 <clinit>() 方法。只有当父接口中定义的变量被使用时, 父接口才会被初始化。另外, 接口的实现类在初始化时也一样不会执行接口的 <clinit>() 方法。

- 虚拟机会保证一个类的 `<clinit>()` 方法在多线程环境中被正确地加锁和同步，如果多个线程同时去初始化一个类，那么只会有一个线程去执行这个类的 `<clinit>()` 方法，其他线程都需要阻塞等待，直到活动线程执行 `<clinit>()` 方法完毕。如果在一个类的 `<clinit>()` 方法中有耗时很长的操作，那就可能造成多个进程阻塞，在实际应用中这种阻塞往往是很隐蔽的。代码清单 7-6 演示了这种场景。

代码清单 7-6 字段解析

```

static class DeadLoopClass {
    static {
        // 如果不加上这个 if 语句，编译器将提示 "Initializer does not complete normally"
        并拒绝编译
        if (true) {
            System.out.println(Thread.currentThread() + "init DeadLoopClass");
            while (true) {
            }
        }
    }
}

public static void main(String[] args) {
    Runnable script = new Runnable() {
        public void run() {
            System.out.println(Thread.currentThread() + "start");
            DeadLoopClass dlc = new DeadLoopClass();
            System.out.println(Thread.currentThread() + " run over");
        }
    };

    Thread thread1 = new Thread(script);
    Thread thread2 = new Thread(script);
    thread1.start();
    thread2.start();
}

```

运行结果如下，一条线程正在死循环以模拟长时间操作，另外一条线程在阻塞等待：

```

Thread[Thread-0,5,main]start
Thread[Thread-1,5,main]start
Thread[Thread-0,5,main]init DeadLoopClass

```

7.4 类加载器

虚拟机设计团队把类加载阶段中的“通过一个类的全限定名来获取描述此类的二进制字节流”这个动作放到 Java 虚拟机外部去实现，以便让应用程序自己决定如何去获取所需要的类。实现这个动作的代码模块被称为“类加载器”。

类加载器可以说是 Java 语言的一项创新，也是 Java 语言流行的重要原因之一，它最初是为了满足 Java Applet 的需求而被开发出来的。如今 Java Applet 技术基本上已经死掉^①，但类加载器却在类层次划分、OSGi、热部署、代码加密等领域大放异彩，成为了 Java 技术体系中一块重要的基石，真可谓是失之桑榆，收之东隅。

7.4.1 类与类加载器

类加载器虽然只用于实现类的加载动作，但它在 Java 程序中起到的作用却远远不限于类加载阶段。对于任意一个类，都需要由加载它的类加载器和这个类本身一同确立其在 Java 虚拟机中的唯一性。这句话可以表达得更通俗一些：比较两个类是否“相等”，只有在这两个类是由同一个类加载器加载的前提之下才有意义，否则，即使这两个类是来源于同一个 Class 文件，只要加载它们的类加载器不同，那这两个类就必定不相等。

这里所指的“相等”，包括代表类的 Class 对象的 equals() 方法、isAssignableFrom() 方法、isInstance() 方法的返回结果，也包括了使用 instanceof 关键字做对象所属关系判定等情况。如果没有注意到类加载器的影响，在某些情况下可能会产生具有迷惑性的结果，代码清单 7-7 中演示了不同的类加载器对 instanceof 关键字运算结果的影响。

代码清单 7-7 不同的类加载器对 instanceof 关键字运算结果的影响

```
/**
 * 类加载器与 instanceof 关键字演示
 *
 * @author zzm
 */
public class ClassLoaderTest {

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

^① 特指浏览器上的 Java Applets，在其他领域，如智能卡上，Java Applets 仍然有很广阔的市场。


```
ClassLoader myLoader = new ClassLoader() {
    @Override
    public Class<?> loadClass(String name) throws ClassNotFoundException {
        try {
            String fileName = name.substring(name.lastIndexOf(".") + 1) +
                ".class";
            InputStream is = getClass().getResourceAsStream(fileName);
            if (is == null) {
                return super.loadClass(name);
            }
            byte[] b = new byte[is.available()];
            is.read(b);
            return defineClass(name, b, 0, b.length);
        } catch (IOException e) {
            throw new ClassNotFoundException(name);
        }
    }
};

Object obj = myLoader.loadClass("org.fenixsoft.classloading.
    ClassLoaderTest").newInstance();

System.out.println(obj.getClass());
System.out.println(obj instanceof org.fenixsoft.classloading.
    ClassLoaderTest);
}
}
```

运行结果：

```
class org.fenixsoft.classloading.ClassLoaderTest
false
```

代码清单 7-7 中构造了一个简单的类加载器，尽管很简陋，但是对于这个演示来说还是够用了。它可以加载与自己在同一路径下的 Class 文件。我们使用这个类加载器去加载了一个名为“org.fenixsoft.classloading.ClassLoaderTest”的类，并实例化了这个类的对象。两行输出结果中，从第一句可以看到这个对象确实是类 org.fenixsoft.classloading.ClassLoaderTest 实例化出来的对象，但从第二句可以发现这个对象与类 org.fenixsoft.classloading.ClassLoaderTest 做所属类型检查的时候却返回了 false，这是因为虚拟机中存在着两个 ClassLoaderTest 类，一个是由系统应用程序类加载器加载的，另外

一个是由我们自定义的类加载器加载的，虽然都来自同一个 Class 文件，但依然是两个独立的类，做对象所属类型检查时结果自然为 false。

7.4.2 双亲委派模型

站在 Java 虚拟机的角度讲，只存在两种不同的类加载器：一种是启动类加载器 (Bootstrap ClassLoader)，这个类加载器使用 C++ 语言实现^①，是虚拟机自身的一部分；另外一种就是所有其他的类加载器，这些类加载器都由 Java 语言实现，独立于虚拟机外部，并且全都继承自抽象类 `java.lang.ClassLoader`。

从 Java 开发人员的角度来看，类加载器还可以划分得更细致一些，绝大部分 Java 程序都会使用到以下三种系统提供的类加载器：

- ❑ 启动类加载器 (Bootstrap ClassLoader)：前面已经介绍过，这个类加载器负责将存放在 `<JAVA_HOME>\lib` 目录中的，或者被 `-Xbootclasspath` 参数所指定的路径中的，并且是虚拟机识别的（仅按照文件名识别，如 `rt.jar`，名字不符合的类库即使放在 `lib` 目录中也不会被加载）类库加载到虚拟机内存中。启动类加载器无法被 Java 程序直接引用。
- ❑ 扩展类加载器 (Extension ClassLoader)：这个加载器由 `sun.misc.Launcher$ExtClassLoader` 实现，它负责加载 `<JAVA_HOME>\lib\ext` 目录中的，或者被 `java.ext.dirs` 系统变量所指定的路径中的所有类库，开发者可以直接使用扩展类加载器。
- ❑ 应用程序类加载器 (Application ClassLoader)：这个类加载器由 `sun.misc.Launcher$AppClassLoader` 来实现。由于这个类加载器是 `ClassLoader` 中的 `getSystemClassLoader()` 方法的返回值，所以一般也称它为系统类加载器。它负责加载用户类路径 (ClassPath) 上所指定的类库，开发者可以直接使用这个类加载器，如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器。

^① 这里只限于 Hot Spot，像 MRP、Maxine 等虚拟机，整个虚拟机本身都是由 Java 编写的，Bootstrap ClassLoader 自然也是由 Java 语言实现的，而不是由 C++ 实现的。退一步讲，除了 Hot Spot 外的其他两个高性能虚拟机 JRockit 和 J9 都有一个代表 Bootstrap ClassLoader 的 Java 类存在，但是关键方法的实现仍然是使用 JNI 回调到 C（注意，不是 C++）的实现上，这个 Bootstrap ClassLoader 的实例也无法被用户获取到。

我们的应用程序都是由这三种类加载器互相配合进行加载的，如果有必要，还可以加入自己定义的类加载器。这些类加载器之间的关系一般会如图 7-2 所示。

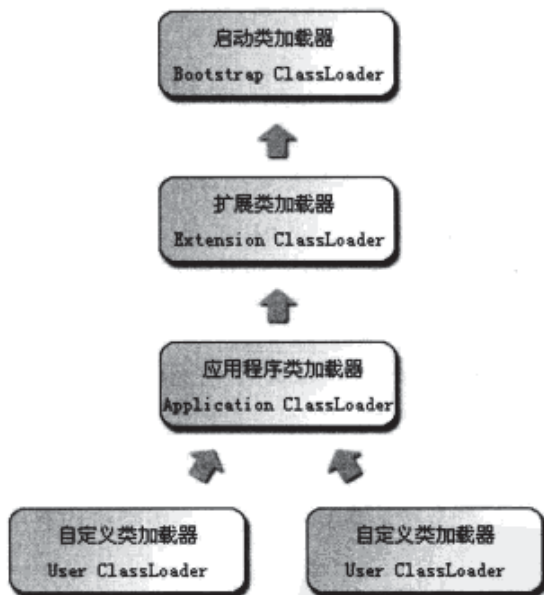


图 7-2 类加载器双亲委派模型

图 7-2 中所展示的种类加载器之间的这种层次关系，就称为类加载器的双亲委派模型（Parents Delegation Model）。双亲委派模型要求除了顶层的启动类加载器外，其余的类加载器都应当有自己的父类加载器。这里类加载器之间的父子关系一般不会以继承（Inheritance）的关系来实现，而是都使用组合（Composition）关系来复用父加载器的代码。

类加载器的双亲委派模型在 JDK 1.2 期间被引入并被广泛应用于之后几乎所有的 Java 程序中，但它并不是一个强制性的约束模型，而是 Java 设计者们推荐给开发者们的一种类加载器实现方式。

双亲委派模型的工作过程是：如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成，每一个层次的类加载器都是如此，因此所有的加载请求最终都应该传送到顶层的启动类加载器中，只有当父加载器反馈自己无法完成这个加载请求（它的搜索范围中没有找到所需的类）时，子

加载器才会尝试自己去加载。

使用双亲委派模型来组织类加载器之间的关系，有一个显而易见的好处就是 Java 类随着它的类加载器一起具备了一种带有优先级的层次关系。例如类 `java.lang.Object`，它存放在 `rt.jar` 之中，无论哪一个类加载器要加载这个类，最终都是委派给启动类加载器进行加载，因此 `Object` 类在程序的各种类加载器环境中都是同一个类。相反，如果没有使用双亲委派模型，由各个类加载器自行去加载的话，如果用户自己写了一个名为 `java.lang.Object` 的类，并放在程序的 `ClassPath` 中，那系统中将会出现多个不同的 `Object` 类，Java 类型体系中最基础的行为也就无从保证，应用程序也将会变得一片混乱。如果您有兴趣的话，可以尝试去写一个与 `rt.jar` 类库中已有类重名的 Java 类，将会发现可以正常编译，但永远无法被加载运行^①。

双亲委派模型对于保证 Java 程序的稳定运作很重要，但它的实现却非常简单，实现双亲委派的代码都集中在 `java.lang.ClassLoader` 的 `loadClass()` 方法之中，如代码清单 7-8 所示，逻辑清晰易懂：先检查是否已经被加载过，若没有加载则调用父加载器的 `loadClass()` 方法，若父加载器为空则默认使用启动类加载器作为父加载器。如果父类加载失败，则在抛出 `ClassNotFoundException` 异常后，再调用自己的 `findClass()` 方法进行加载。

代码清单 7-8 双亲委派模型的实现

```
protected synchronized Class<?> loadClass(String name, boolean resolve) throws
ClassNotFoundException
{
    // 首先，检查请求的类是否已经被加载过了
    Class c = findLoadedClass(name);
    if (c == null) {
        try {
            if (parent != null) {
                c = parent.loadClass(name, false);
            } else {
                c = findBootstrapClassOrNull(name);
            }
        }
    }
}
```

^① 即使自定义了自己的类加载器，强行用 `defineClass()` 方法去加载一个以“`java.lang`”开头的类也不会成功。如果读者尝试这样做的话，将会收到一个由虚拟机自己抛出的“`java.lang.SecurityException: Prohibited package name: java.lang`”异常。


```
    } catch (ClassNotFoundException e) {
        // 如果父类加载器抛出 ClassNotFoundException
        // 则说明父类加载器无法完成加载请求
    }
    if (c == null) {
        // 在父类加载器无法加载的时候
        // 再调用本身的 findClass 方法来进行类加载
        c = findClass(name);
    }
}
if (resolve) {
    resolveClass(c);
}
return c;
}
```

7.4.3 破坏双亲委派模型

上文提到过双亲委派模型并不是一个强制性的约束模型，而是 Java 设计者们推荐给开发者们的类加载器实现方式。在 Java 的世界里面大部分的类加载器都遵循这个模型，但也有例外的情况，到现在为止，双亲委派模型主要出现过三次较大规模的“被破坏”情况。

双亲委派模型的第一次“被破坏”其实发生在双亲委派模型出现之前——即 JDK 1.2 发布之前。由于双亲委派模型在 JDK 1.2 之后才被引入的，而类加载器和抽象类 `java.lang.ClassLoader` 则在 JDK 1.0 时代就已经存在，面对已经存在的用户自定义类加载器的实现代码，Java 设计者们引入双亲委派模型时不得不做出一些妥协。为了向前兼容，JDK 1.2 之后的 `java.lang.ClassLoader` 添加了一个新的 `protected` 方法 `findClass()`，在此之前，用户去继承 `java.lang.ClassLoader` 的唯一目的是为了重写 `loadClass()` 方法，因为虚拟机在进行类加载的时候会调用加载器的私有方法 `loadClassInternal()`，而这个方法的唯一逻辑就是去调用自己的 `loadClass()`。

上一节我们已经看过 `loadClass()` 方法的代码，双亲委派的具体逻辑就实现在这个方法之中，JDK 1.2 之后已不提倡用户再去覆盖 `loadClass()` 方法，而应当把自己的类加载逻辑写到 `findClass()` 方法中，在 `loadClass()` 方法的逻辑里如果父类加载失败，则会调用自己的 `findClass()` 方法来完成加载，这样就可以保证新写出来的类加载器是符合双亲委

派规则的。

双亲委派模型的第二次“被破坏”是由这个模型自身的缺陷所导致的，双亲委派很好地解决了各个类加载器的基础类的统一问题（越基础的类由越上层的加载器进行加载），基础类之所以被称为“基础”，是因为它们总是作为被用户代码调用的 API，但世事往往没有绝对的完美，如果基础类又要调用回用户的代码，那该怎么办了？

这并非是不可能的事情，一个典型的例子便是 JNDI 服务，JNDI 现在已经是 Java 的标准服务，它的代码由启动类加载器去加载（在 JDK 1.3 时代放进去的 `rt.jar`），但 JNDI 的目的就是对资源进行集中管理和查找，它需要调用由独立厂商实现并部署在应用程序的 `ClassPath` 下的 JNDI 接口提供者（SPI，Service Provider Interface）的代码，但启动类加载器不可能“认识”这些代码啊！那该怎么办？

为了解决这个困境，Java 设计团队只好引入了一个不太优雅的设计：线程上下文类加载器（Thread Context ClassLoader）。这个类加载器可以通过 `java.lang.Thread` 类的 `setContextClassLoader()` 方法进行设置，如果创建线程时还未设置，它将会从父线程中继承一个；如果在应用程序的全局范围内都没有设置过，那么这个类加载器默认就是应用程序类加载器。

有了线程上下文类加载器，就可以做一些“舞弊”的事情了，JNDI 服务使用这个线程上下文类加载器去加载所需要的 SPI 代码，也就是父类加载器请求子类加载器去完成类加载的动作，这种行为实际上就是打通了双亲委派模型的层次结构来逆向使用类加载器，已经违背了双亲委派模型的一般性原则，但这也是无可奈何的事情。Java 中所有涉及 SPI 的加载动作基本上都采用这种方式，例如 JNDI、JDBC、JCE、JAXB 和 JBI 等。

双亲委派模型的第三次“被破坏”是由于用户对程序动态性的追求而导致的，这里所说的“动态性”指的是当前一些非常“热”的名词：代码热替换（HotSwap）、模块热部署（Hot Deployment）等，说白了就是希望应用程序能像我们的电脑外设那样，插上鼠标或 U 盘，不用重启机器就能立即使用，鼠标有问题或要升级就换个鼠标，不用停机也不用重启。对于个人电脑来说，重启一次其实没有什么大不了的，但对于一些生产系统来说，关机重启一次可能就要被列为生产事故，这种情况下热部署就对软件开发者，尤其是企业级软件开发者具有很大的吸引力。

在 JSR-297^①、JSR-277^②规范从纸上标准变成真正可运行的程序之前，OSGi 是当前业界“事实上”的 Java 模块化标准，而 OSGi 实现模块化热部署的关键则是它自定义的类加载器机制的实现。每一个程序模块（OSGi 中称为 Bundle）都有一个自己的类加载器，当需要更换一个 Bundle 时，就把 Bundle 连同类加载器一起换掉以实现代码的热替换。

在 OSGi 环境下，类加载器不再是双亲委派模型中的树状结构，而是进一步发展为网状结构，当收到类加载请求时，OSGi 将按照下面的顺序进行类搜索：

- (1) 将以 `java.*` 开头的类，委派给父类加载器加载。
- (2) 否则，将委派列表名单内的类，委派给父类加载器加载。
- (3) 否则，将 Import 列表中的类，委派给 Export 这个类的 Bundle 的类加载器加载。
- (4) 否则，查找当前 Bundle 的 ClassPath，使用自己的类加载器加载。
- (5) 否则，查找类是否在自己的 Fragment Bundle 中，如果在，则委派给 Fragment Bundle 的类加载器加载。
- (6) 否则，查找 Dynamic Import 列表的 Bundle，委派给对应 Bundle 的类加载器加载。
- (7) 否则，类查找失败。

上面的查找顺序中只有开头两点仍然符合双亲委派规则，其余的类查找都是在平级的类加载器中进行的。

笔者虽然使用了“被破坏”这个词来形容上述不符合双亲委派模型原则的行为，但这里“被破坏”并不带有贬义的感情色彩。只要有足够意义和理由，突破已有的原则就可算作一种创新。正如 OSGi 中的类加载器并不符合传统的双亲委派的类加载器，并且业界对其为了实现热部署而带来的额外的高复杂度还存在不少争议，但在 Java 程序员中基本有一个共识：OSGi 中对类加载器的使用是很值得学习的，弄懂了 OSGi 的实现，自然就明白了类加载器的精粹。

① JSR-294 : Improved Modularity Support in the Java Programming Language (Java 编程语言中的改进模块性支持)。

② JSR-277 : Java Module System (Java 模块系统)。

7.5 本章小结

本章介绍了类加载过程的“加载”、“验证”、“准备”、“解析”和“初始化”五个阶段中虚拟机进行了哪些动作，还介绍了类加载器的工作原理及其对虚拟机的意义。

经过第 6、7 两章的讲解，相信您已经对如何在 Class 文件中定义类，如何将类加载到虚拟机中这两个问题有了一个比较系统的了解，下一章我们将一起来看看虚拟机如何执行定义在 Class 文件里的字节码。



第 8 章 虚拟机字节码执行引擎

本章主要内容

- 概述
- 运行时栈帧结构
- 方法调用
- 基于栈的字节码解释执行引擎

代码编译的结果是从本地机器码转变为字节码，是存储格式发展的一小步，却是编程语言发展的一大步。

8.1 概述

执行引擎是 Java 虚拟机最核心的组成部分之一。“虚拟机”是一个相对于“物理机”的概念，这两种机器都有代码执行能力，其区别是物理机的执行引擎是直接建立在处理器、硬件、指令集和操作系统层面上的，而虚拟机的执行引擎则是由自己实现的，因此可以自行制定指令集与执行引擎的结构体系，并且能够执行那些不被硬件直接支持的指令集格式。

在 Java 虚拟机规范中制定了虚拟机字节码执行引擎的概念模型，这个概念模型成为各种虚拟机执行引擎的统一外观（Facade）。在不同的虚拟机实现里面，执行引擎在执行 Java 代码的时候可能有解释执行（通过解释器执行）和编译执行（通过即时编译器产生本地代码执行）两种选择^①，也可能两者兼备，甚至还可能包含几个不同级别的编译器执行引擎。但从外观上看起来，所有的 Java 虚拟机的执行引擎都是一致的：输入的是字

^① 有一些虚拟机（如 Sun Classic VM）的内部只存在解释器，只能解释执行，另外一些虚拟机（如 BEA JRockit）的内部只存在即时编译器，只能编译执行。

节码文件，处理过程是字节码解析的等效过程，输出的是执行结果。本章将主要从概念模型的角度来讲解虚拟机的方法调用和字节码执行。

8.2 运行时栈帧结构

栈帧（Stack Frame）是用于支持虚拟机进行方法调用和方法执行的数据结构，它是虚拟机运行时数据区中的虚拟机栈（Virtual Machine Stack）^①的栈元素。栈帧存储了方法的局部变量表、操作数栈、动态连接和方法返回地址等信息。每一个方法从调用开始到执行完成的过程，就对应着一个栈帧在虚拟机栈里面从入栈到出栈的过程。

每一个栈帧都包括了局部变量表、操作数栈、动态连接、方法返回地址和一些额外的附加信息。在编译程序代码的时候，栈帧中需要多大的局部变量表、多深的操作数栈都已经完全确定了，并且写入到方法表的 Code 属性之中^②，因此一个栈帧需要分配多少内存，不会受到程序运行期变量数据的影响，而仅仅取决于具体的虚拟机实现。

一个线程中的方法调用链可能会很长，很多方法都同时处于执行状态。对于执行引擎来讲，活动线程中，只有栈顶的栈帧是有效的，称为当前栈帧（Current Stack Frame），这个栈帧所关联的方法称为当前方法（Current Method）。执行引擎所运行的所有字节码指令都只针对当前栈帧进行操作，栈帧的概念结构如图 8-1 所示。

接下来我们将详细了解一下栈帧中的局部变量表、操作数栈、动态连接、方法返回地址等各个部分的作用和数据结构。

8.2.1 局部变量表

局部变量表是一组变量值存储空间，用于存放方法参数和方法内部定义的局部变量。在 Java 程序被编译为 Class 文件时，就在方法的 Code 属性的 max_locals 数据项中确定了该方法所需要分配的最大局部变量表的容量。

^① 详细内容请参见 2.2 节的相关内容。

^② 详细内容请参见 6.3.7 节的相关内容。



图 8-1 栈帧的概念结构

局部变量表的容量以变量槽 (Variable Slot, 下称 Slot) 为最小单位, 虚拟机规范中并没有明确指明一个 Slot 应占用的内存空间大小, 只是很有“导向性”地说明每个 Slot 都应该能存放一个 boolean、byte、char、short、int、float、reference 或 returnAddress 类型的数据, 这种描述与明确指出“每个 Slot 占用 32 位长度的内存空间”有一些差别, 它允许 Slot 的长度随着处理器、操作系统或虚拟机的不同而发生变化。不过无论如何, 即使在 64 位虚拟机中使用了 64 位长度的内存空间来实现一个 Slot, 虚拟机仍要使用对齐和补白的手段让 Slot 在外观上看起来与 32 位虚拟机中的一致。

既然前面提到了数据类型，在此顺便说一下，一个 Slot 可以存放一个 32 位以内的数据类型，Java 中占用 32 位以内的数据类型有 boolean、byte、char、short、int、float、reference^①和 returnAddress 八种类型。前面六种不需要多加解释，大家都认识，而后面的 reference 是对象的引用。虚拟机规范既没有说明它的长度，也没有明确指出这个引用应有怎样的结构，但是一般来说，虚拟机实现至少都应当能从此引用中直接或间接地找到对象在 Java 堆中的起始地址索引和方法区中的对象类型数据。而 returnAddress 是为字节码指令 jsr、jsr_w 和 ret 服务的，它指向了一条字节码指令的地址。

对于 64 位的数据类型，虚拟机会以高位在前的方式为其分配两个连续的 Slot 空间。Java 语言中明确规定的 64 位的数据类型只有 long 和 double 两种（reference 类型则可能是 32 位也可能是 64 位）。值得一提的是，这里把 long 和 double 数据类型分割存储的做法与“long 和 double 的非原子性协定”中把一次 long 和 double 数据类型读写分割为两次 32 位读写的做法类似，读者阅读到 Java 内存模型时可以对比一下。不过，由于局部变量表建立在线程的堆栈上，是线程私有的数据，无论读写两个连续的 Slot 是否是原子操作，都不会引起数据安全问题^②。

虚拟机通过索引定位的方式使用局部变量表，索引值的范围是从 0 开始到局部变量表最大的 Slot 数量。如果是 32 位数据类型的变量，索引 n 就代表了使用第 n 个 Slot，如果是 64 位数据类型的变量，则说明要使用第 n 和第 $n+1$ 两个 Slot。

在方法执行时，虚拟机是使用局部变量表完成参数值到参数变量列表的传递过程的，如果是实例方法（非 static 的方法），那么局部变量表中第 0 位索引的 Slot 默认是用于传递方法所属对象实例的引用，在方法中可以通过关键字“this”来访问这个隐含的参数。其余参数则按照参数表的顺序来排列，占用从 1 开始的局部变量 Slot，参数表分配完毕后，再根据方法体内部定义的变量顺序和作用域分配其余的 Slot。

局部变量表中的 Slot 是可重用的，方法体中定义的变量，其作用域并不一定会覆盖整个方法体，如果当前字节码 PC 计数器的值已经超出了某个变量的作用域，那么这个

① Java 虚拟机规范中没有明确规定 reference 类型的长度，它的长度与实际使用 32 还是 64 位的虚拟机有关，如果是 64 位虚拟机，还与是否开启某些对象指针压缩的优化有关，这里我们暂且只取 32 位虚拟机的 reference 长度。

② 这是 Java 内存模型中定义的内容，关于原子操作与“long 和 double 的非原子性协定”等问题，将在本书第 12 章中再做详细讲解。

202 ❖ 第三部分 虚拟机执行子系统

变量对应的 Slot 就可以交给其他变量使用。这样的设计不仅仅是为了节省栈空间，在某些情况下 Slot 的复用会直接影响到系统的垃圾收集行为，请看代码清单 8-1 的演示。

代码清单 8-1 局部变量表 Slot 复用对垃圾收集的影响之一

```
public static void main(String[] args) {  
    byte[] placeholder = new byte[64 * 1024 * 1024];  
    System.gc();  
}
```

代码清单 8-1 中的代码很简单，向内存填充了 64MB 的数据，然后通知虚拟机进行垃圾收集。我们在虚拟机运行参数中加上“-verbose:gc”来看看垃圾收集的过程，发现在 System.gc() 运行后并没有回收掉这 64MB 的内存，下面是运行的结果：

```
[GC 66846K->65824K(125632K), 0.0032678 secs]  
[Full GC 65824K->65746K(125632K), 0.0064131 secs]
```

没有回收掉 placeholder 所占的内存能说得过去，因为在执行 System.gc() 时，变量 placeholder 还处于作用域之内，虚拟机自然不敢回收掉 placeholder 的内存。我们把代码修改一下，变成代码清单 8-2 中的样子。

代码清单 8-2 局部变量表 Slot 复用对垃圾收集的影响之二

```
public static void main(String[] args) {  
    {  
        byte[] placeholder = new byte[64 * 1024 * 1024];  
    }  
    System.gc();  
}
```

加入了花括号之后，placeholder 的作用域被限制在花括号之内，从代码逻辑上讲，在执行 System.gc() 的时候，placeholder 已经不可能再被访问了，但执行一下这段程序，会发现运行结果如下，还是有 64MB 的内存没有被回收掉，这又是为什么呢？

```
[GC 66846K->65888K(125632K), 0.0009397 secs]  
[Full GC 65888K->65746K(125632K), 0.0051574 secs]
```

在解释为什么之前，我们先对这段代码进行第二次修改，在调用 System.gc() 之前加入一行代码“int a = 0;”，变成代码清单 8-3 的样子。

代码清单 8-3 局部变量表 Slot 复用对垃圾收集的影响之三

```
public static void main(String[] args) {  
    {  
        byte[] placeholder = new byte[64 * 1024 * 1024];  
    }  
    int a = 0;  
    System.gc();  
}
```

这个修改看起来很莫名其妙，但运行一下程序，却发现这次内存真的被正确回收了：

```
[GC 66401K->65778K(125632K), 0.0035471 secs]  
[Full GC 65778K->218K(125632K), 0.0140596 secs]
```

代码清单 8-1 至 8-3 中，placeholder 能否被回收的根本原因就是：局部变量表中的 Slot 是否还存有关于 placeholder 数组对象的引用。第一次修改中，代码虽然已经离开了 placeholder 的作用域，但在此之后，没有任何对局部变量表的读写操作，placeholder 原本所占用的 Slot 还没有被其他变量所复用，所以作为 GC Roots 一部分的局部变量表仍然保持着对它的关联。这种关联没有被及时打断，在绝大部分情况下影响都很轻微。但如果遇到一个方法，其后面的代码有一些耗时很长的操作，而前面又定义了占用了大量内存、实际上已经不会再被使用的变量，手动将其设置为 null 值（用来代替“int a=0;”，把变量对应的局部变量表 Slot 清空）就不是一个毫无意义的操作，这种操作可以作为一种在极特殊情形（对象占用内存大、此方法的栈帧长时间不能被回收、方法调用次数达不到 JIT 的编译条件）下的“奇技”来使用。但不应当对赋 null 值操作有过多的依赖，也没有必要把它当做一个普遍的编码方法来推广^①。以恰当的变量作用域来控制变量回收时间才是最优雅解决方法，如代码清单 8-3 那样的场景并不多见。

另外，赋 null 值的操作在经过虚拟机 JIT 编译器优化之后会被消除掉，这时候将变量设置为 null 实际上是没有意义的。字节码被编译为本地代码后，对 GC Roots 的枚举也与解释执行时期有所差别，代码清单 8-2 在经过 JIT 编译后，System.gc() 执行时就可以正确地回收掉内存，而无须写成代码清单 8-3 的样子了。

^① 《Practical Java》一书中推荐把所有不再需要的引用手工赋 null 值，但没有明确说明原因，笔者认为这样做是没有必要的，读完本节后读者可以自行思考。

关于局部变量表，还有一点可能会对实际开发产生影响，就是局部变量不像前面介绍的类变量那样存在“准备阶段”。通过前一章的讲解，我们已经知道类变量有两次赋初始值的过程，一次在准备阶段，赋予系统初始值；另外一次在初始化阶段，赋予程序员定义的初始值。因此即使在初始化阶段程序员没有为类变量赋值也没有关系，类变量仍然具有一个确定的初始值。但局部变量就不一样了，如果一个局部变量定义了但没有赋初始值是不能使用的。所以不要认为 Java 中任何情况下都存在诸如整型变量默认为 0、布尔型变量默认为 false 之类的默认值。如代码清单 8-4 所示，这段代码其实并不能运行，所幸编译器能在编译期间检查到并提示这一点。即便编译器能通过手动生成字节码的方式制造出下面的代码效果，字节码检验的时候也会被虚拟机发现，从而导致类加载失败。

代码清单 8-4 未赋值的局部变量

```
public static void main(String[] args) {  
    int a;  
    System.out.println(a);  
}
```

8.2.2 操作数栈

操作数栈也常被称为操作栈，它是一个后入先出（Last In First Out, LIFO）栈。同局部变量表一样，操作数栈的最大深度也在编译的时候被写入到 Code 属性的 max_stacks 数据项之中。操作数栈的每一个元素可以是任意的 Java 数据类型，包括 long 和 double。32 位数据类型所占的栈容量为 1，64 位数据类型所占的栈容量为 2。在方法执行的任何时候，操作数栈的深度都不会超过在 max_stacks 数据项中设定的最大值。

当一个方法刚刚开始执行的时候，这个方法的操作数栈是空的，在方法的执行过程中，会有各种字节码指令向操作数栈中写入和提取内容，也就是入栈出栈操作。例如，在做算术运算的时候是通过操作数栈来进行的，又或者在调用其他方法的时候是通过操作数栈来进行参数传递的。

举个例子，整数加法的字节码指令 iadd 在运行的时候要求操作数栈中最接近栈顶的两个元素已经存入了两个 int 型的数值，当执行这个指令时，会将这两个 int 值出栈并相

加，然后将相加的结果入栈。

操作数栈中元素的数据类型必须与字节码指令的序列严格匹配，在编译程序代码的时候，编译器要严格保证这一点，在类校验阶段的数据流分析中还要再次验证这一点。再以上面的 `iadd` 指令为例，这个指令用于整型数加法，它在执行时，最接近栈顶的两个元素的数据类型必须为 `int` 型，不能出现一个 `long` 和一个 `float` 使用 `iadd` 命令相加的情况。

另外，在概念模型中，两个栈帧作为虚拟机栈的元素，相互之间是完全独立的。但是大多数虚拟机的实现里都会做一些优化处理，令两个栈帧出现一部分重叠。让下面栈帧的部分操作数栈与上面栈帧的部分局部变量表重叠在一起，这样在进行方法调用时就可以共用一部分数据，而无须进行额外的参数复制传递了，重叠的过程如图 8-2 所示。

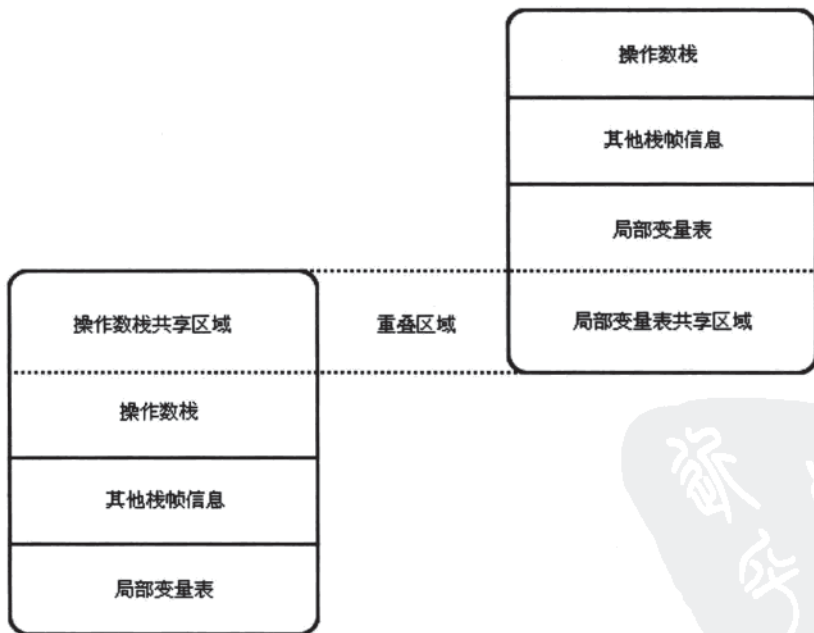


图 8-2 两个栈帧之间的数据共享

Java 虚拟机的解释执行引擎称为“基于栈的执行引擎”，其中所指的“栈”就是操作数栈。本章稍后会对基于栈的代码过程进行更详细的讲解。

8.2.3 动态连接

每个栈帧都包含一个指向运行时常量池^①中该栈帧所属方法的引用，持有这个引用是为了支持方法调用过程中的动态连接。通过第 6 章的讲解，我们知道 Class 文件的常量池中存有大量的符号引用，字节码中的方法调用指令就以常量池中指向方法的符号引用为参数。这些符号引用一部分会在类加载阶段或第一次使用的时候转化为直接引用，这种转化称为静态解析。另外一部分将在每一次的运行期间转化为直接引用，这部分称为动态连接。关于这两个转化过程的详细信息，将在 8.3 节中再详细讲解。

8.2.4 方法返回地址

当一个方法被执行后，有两种方式退出这个方法。第一种方式是执行引擎遇到任意一个方法返回的字节码指令，这时候可能会有返回值传递给上层的方法调用者（调用当前方法的方法称为调用者），是否有返回值和返回值的类型将根据遇到何种方法返回指令来决定，这种退出方法的方式称为正常完成出口（Normal Method Invocation Completion）。

另外一种退出方式是，在方法执行过程中遇到了异常，并且这个异常没有在方法体内得到处理，无论是 Java 虚拟机内部产生的异常，还是代码中使用 `athrow` 字节码指令产生的异常，只要在本方法的异常表中没有搜索到匹配的异常处理器，就会导致方法退出，这种退出方法的方式称为异常完成出口（Abrupt Method Invocation Completion）。一个方法使用异常完成出口的方式退出，是不会给它的上层调用者产生任何返回值的。

无论采用何种退出方式，在方法退出之后，都需要返回到方法被调用的位置，程序才能继续执行，方法返回时可能需要在栈帧中保存一些信息，用来帮助恢复它的上层方法的执行状态。一般来说，方法正常退出时，调用者的 PC 计数器的值就可以作为返回地址，栈帧中很可能会保存这个计数器值。而方法异常退出时，返回地址是要通过异常处理器表来确定的，栈帧中一般不会保存这部分信息。

方法退出的过程实际上等同于把当前栈帧出栈，因此退出时可能执行的操作有：恢复上层方法的局部变量表和操作数栈，把返回值（如果有的话）压入调用者栈帧的操作数栈中，调整 PC 计数器的值以指向方法调用指令后面的一条指令等。

^① 运行时常量池的相关内容详见第 2 章。

8.2.5 附加信息

虚拟机规范允许具体的虚拟机实现增加一些规范里没有描述的信息到栈帧之中，例如与调试相关的信息，这部分信息完全取决于具体的虚拟机实现，这里不再详述。在实际开发中，一般会把动态连接、方法返回地址与其他附加信息全部归为一类，称为栈帧信息。

8.3 方法调用

方法调用并不等同于方法执行，方法调用阶段唯一的任务就是确定被调用方法的版本（即调用哪一个方法），暂时还不涉及方法内部的具体运行过程。在程序运行时，进行方法调用是最普遍、最频繁的操作，但前面已经讲过，Class 文件的编译过程中不包含传统编译中的连接步骤，一切方法调用在 Class 文件里面存储的都只是符号引用，而不是方法在实际运行时内存布局中的入口地址（相当于之前所说的直接引用）。这个特性给 Java 带来了更强大的动态扩展能力，但也使得 Java 方法的调用过程变得相对复杂起来，需要在类加载期间甚至到运行期间才能确定目标方法的直接引用。

8.3.1 解析

继续前面关于方法调用的话题，所有方法调用中的目标方法在 Class 文件里面都是一个常量池中的符号引用，在类加载的解析阶段，会将其中的一部分符号引用转化为直接引用，这种解析能成立的前提是：方法在程序真正运行之前就有一个可确定的调用版本，并且这个方法的调用版本在运行期是不可改变的。换句话说，调用目标在程序代码写好、编译器进行编译时就必须确定下来。这类方法的调用称为解析（Resolution）。

在 Java 语言中，符合“编译期可知，运行期不可变”这个要求的方法主要有静态方法和私有方法两大类，前者与类型直接关联，后者在外部不可被访问，这两种方法都不可能通过继承或别的方式重写出其他版本，因此它们都适合在类加载阶段进行解析。

与之相对应，在 Java 虚拟机里面提供了四条方法调用字节码指令^①，分别是：

□ `invokestatic`：调用静态方法。

^① JSR-292 中引入了第 5 条新的字节码指令 `invokedynamic` 以及新的 `java.lang.invoke`（未稳定前叫 `java.dyn`）包实现对动态语言的支持，这些特性有望在 JDK 1.7 中正式提供，本文暂以现有的 JDK 1.6 为基础进行阐述。

- `invokespecial` : 调用实例构造器 `<init>` 方法、私有方法和父类方法。
- `invokevirtual` : 调用所有的虚方法。
- `invokeinterface` : 调用接口方法, 会在运行时再确定一个实现此接口的对象。

只要能被 `invokestatic` 和 `invokespecial` 指令调用的方法, 都可以在解析阶段确定唯一的调用版本, 符合这个条件的有静态方法、私有方法、实例构造器和父类方法四类, 它们在类加载的时候就会把符号引用解析为该方法的直接引用。这些方法可以称为非虚方法, 与之相反, 其他方法就称为虚方法 (除去 `final` 方法, 后文会提到)。代码清单 8-5 演示了一个最常见的解析调用的例子, 此样例中, 静态方法 `sayHello()` 只可能属于类型 `StaticResolution`, 没有任何手段可以覆盖或隐藏这个方法。

代码清单 8-5 方法静态解析演示

```

/**
 * 方法静态解析演示
 *
 * @author zzm
 */
public class StaticResolution {

    public static void sayHello() {
        System.out.println("hello world");
    }

    public static void main(String[] args) {
        StaticResolution.sayHello();
    }

}

```

使用 `javap` 命令查看这段程序的字节码, 会发现的确是通过 `invokestatic` 命令来调用 `sayHello()` 方法的:

```

D:\Develop>javap -verbose StaticResolution
public static void main(java.lang.String[]);
Code:
  Stack=0, Locals=1, Args_size=1
  0:   invokestatic   #31; //Method sayHello:()V
  3:   return

```

```
LineNumberTable:  
  line 15: 0  
  line 16: 3
```

Java 中的非虚方法除了使用 `invokestatic` 和 `invokespecial` 调用的方法之外还有一种，就是被 `final` 修饰的方法。虽然 `final` 方法是使用 `invokevirtual` 指令来调用的，但是由于它无法被覆盖，没有其他版本，所以也无须对方法接收者进行多态选择，又或者说多态选择的结果肯定是唯一的。在 Java 语言规范中明确说明了 `final` 方法是一种非虚方法。

解析调用一定是个静态的过程，在编译期间就完全确定，在类装载的解析阶段就会把涉及的符号引用全部转变为可确定的直接引用，不会延迟到运行期再去完成。而分派 (`Dispatch`) 调用则可能是静态的也可能是动态的，根据分派依据的宗量数^①可分为单分派和多分派。这两类分派方式两两组合就构成了静态单分派、静态多分派、动态单分派、动态多分派四种分派情况，下面我们看看虚拟机中的方法分派是如何进行的。

8.3.2 分派

众所周知，Java 是一门面向对象的程序设计语言，因为 Java 具备面向对象的三个基本特征：继承、封装和多态。本节讲解的分派调用过程将会揭示多态性特征的一些最基本的体现（如“重载”和“重写”），在 Java 中是如何实现的，这里的实现当然不是语法上该如何写，我们关心的依然是虚拟机如何确定正确的目标方法。

1. 静态分派

在开始讲解静态分派^②前，笔者准备了一段经常出现在面试题中的程序代码，读者不妨先看一遍，想一下程序的输出结果是什么。后面我们的话题将围绕这个类的方法来重载 (`Overload`) 代码，以分析虚拟机和编译器确定方法版本的过程。程序如代码清单 8-6 所示。

代码清单 8-6 方法静态分派演示

```
package org.fenixsoft.polymorphic;  
  
/**
```

① 这里涉及的分派的相关概念（如“宗量”等）在后文中会有所解释。

② 严格来说，“`Dispatch`”这个词一般不用在静态环境中，英文技术文档的称呼是“`Method Overload Resolution`”，但国内的各种资料都普遍将这种行为称为“静态分派”，特此说明。

210 ❖ 第三部分 虚拟机执行子系统

```
* 方法静态分派演示
* @author zzm
*/
public class StaticDispatch {

    static abstract class Human {
    }

    static class Man extends Human {
    }

    static class Women extends Human {
    }

    public void sayHello(Human guy) {
        System.out.println("hello,guy!");
    }

    public void sayHello(Man guy) {
        System.out.println("hello,gentleman!");
    }

    public void sayHello(Women guy) {
        System.out.println("hello,lady!");
    }

    public static void main(String[] args) {
        Human man = new Man();
        Human women = new Women();
        StaticResolution sr = new StaticResolution();
        sr.sayHello(man);
        sr.sayHello(women);
    }
}
```

运行结果：

```
hello,guy!
hello,guy!
```

代码清单 8-6 中的代码实际上是在考察读者对重载的理解程度，相信对 Java 稍有经验的程序员看完程序后都能得出正确的运行结果，但为什么会选择执行参数类型为

Human 的重载呢？在解决这个问题之前，我们先按如下代码定义两个重要的概念：

```
Human man = new Man();
```

我们把上面代码中的“Human”称为变量的静态类型（Static Type）或者外观类型（Apparent Type），后面的“Man”则称为变量的实际类型（Actual Type），静态类型和实际类型在程序中都可以发生一些变化，区别是静态类型的变化仅仅在使用时发生，变量本身的静态类型不会被改变，并且最终的静态类型是在编译期可知的；而实际类型变化的结果在运行期才可确定，编译器在编译程序的时候并不知道一个对象的实际类型是什么。如下面的代码：

```
// 实际类型变化  
Human man = new Man();  
man = new Women();  
// 静态类型变化  
sr.sayHello((Man) man)  
sr.sayHello((Women) man)
```

解释了这两个概念，再回到代码清单 8-6 的样例代码中。main() 里面的两次 sayHello() 方法调用，在方法接收者已经确定是对象“sr”的前提下，使用哪个重载版本，就完全取决于传入参数的数量和数据类型。代码中刻意地定义了两个静态类型相同、实际类型不同的变量，但虚拟机（准确地说是编译器）在重载时是通过参数的静态类型而不是实际类型作为判定依据的。并且静态类型是编译期可知的，所以在编译阶段，Javac 编译器就根据参数的静态类型决定使用哪个重载版本，所以选择了 sayHello(Human) 作为调用目标，并把这个方法的符号引用写到 main() 方法里的两条 invokevirtual 指令的参数中。

所有依赖静态类型来定位方法执行版本的分派动作，都称为静态分派。静态分派的最典型应用就是方法重载。静态分派发生在编译阶段，因此确定静态分派的动作实际上不是由虚拟机来执行的。另外，编译器虽然能确定出方法的重载版本，但在很多情况下这个重载版本并不是“唯一的”，往往只能确定一个“更加合适的”版本。这种模糊的结论在由 0 和 1 构成的计算机世界中算是个比较“稀罕”的事件，产生这种模糊结论的主要原因是字面量不需要定义，所以字面量没有显式的静态类型，它的静态类型只能通过语言上的规则去理解和推断。代码清单 8-7 演示了何为“更加合适的”版本。

代码清单 8-7 重载方法匹配优先级

```
package org.fenixsoft.polymorphic;

public class Overload {

    public static void sayHello(Object arg) {
        System.out.println("hello Object");
    }

    public static void sayHello(int arg) {
        System.out.println("hello int");
    }

    public static void sayHello(long arg) {
        System.out.println("hello long");
    }

    public static void sayHello(Character arg) {
        System.out.println("hello Character");
    }

    public static void sayHello(char arg) {
        System.out.println("hello char");
    }

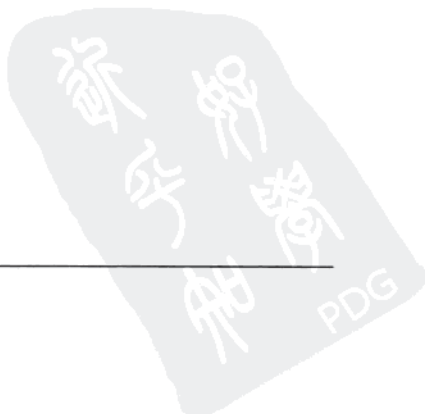
    public static void sayHello(char... arg) {
        System.out.println("hello char ...");
    }

    public static void sayHello(Serializable arg) {
        System.out.println("hello Serializable");
    }

    public static void main(String[] args) {
        sayHello('a');
    }
}
```

上面的代码运行后会输出：

hello char



这很好理解，'a' 是一个 char 类型的数据，自然会寻找参数类型为 char 的重载方法，如果注释掉 sayHello(char arg) 方法，那么输出会变为：

```
hello int
```

这时发生了一次自动类型转换，'a' 除了可以代表一个字符串外，还可以代表数字 65（字符 'a' 的 Unicode 数值为十进制数字 65），因此参数类型为 int 的重载也是合适的。我们继续注释掉 sayHello(int arg) 方法，那么输出会变为：

```
hello long
```

这时发生了两次自动类型转换，'a' 转换为整数 65 之后，进一步转换为长整数 65L，匹配了参数类型为 long 的重载。笔者在代码中没有编写其他类型（如 float、double 等）的重载，不过实际上自动转型还能继续发生多次，按照 char → int → long → float → double 的顺序转型进行匹配。但不会匹配到 byte 和 short 类型的重载，因为 char 到 byte 或 short 的转型是不安全的。我们继续注释掉 sayHello(long arg) 方法，那么输出会变为：

```
hello Character
```

这时发生了一次自动装箱，'a' 被包装为它的封装类型 java.lang.Character，所以匹配到了参数类型为 Character 的重载，继续注释掉 sayHello(Character arg) 方法，那么输出会变为：

```
hello Serializable
```

这个输出可能会让人感觉摸不着头脑，一个字符或数字与序列化有什么关系？出现 hello Serializable，是因为 java.lang.Serializable 是 java.lang.Character 类实现的一个接口，自动装箱之后发现还是找不到装箱类，但是找到了装箱类实现了的接口类型，所以紧接着又发生一次自动转型。char 可以转型成 int，但是 Character 是绝对不会转型为 Integer 的，它只能安全地转型为它实现的接口或父类。Character 还实现了另外一个接口 java.lang.Comparable<Character>，如果同时出现两个参数分别为 Serializable 和 Comparable<Character> 的重载方法，那么它们此时的优先级是一样的。编译器无法确定要自动转型为哪种类型，会提示类型模糊，拒绝编译。程序必须在调用时显式地指定字面量的静态类型，如 sayHello((Comparable<Character>) 'a')，才能通过编译。下面继续

注释掉 `sayHello(Serializable arg)` 方法，输出会变为：

```
hello Object
```

这时是 `char` 装箱后转型为父类了，如果有多个父类，那么将在继承关系中从下往上开始搜索，越接近上层的优先级越低。即使方法调用传入的参数值为 `null`，这个规则仍然适用。我们把 `sayHello(Object arg)` 也注释掉，输出将会变为：

```
hello char ...
```

七个重载方法已经被注释得只剩一个了，可见变长参数的重载优先级是最低的，这时候字符 `'a'` 被当作了一个数组元素。笔者使用的是 `char` 类型的变长参数，读者在验证时还可以选择 `int` 类型、`Character` 类型、`Object` 类型等的变长参数重载来把上面的过程重新演示一遍。但是要注意的是，有一些在单个参数中能成立的自动转型，如 `char` 转型为 `int`，在变长参数中是不成立的^①。

代码清单 8-7 演示了编译期间选择静态分派目标的过程，这个过程也是 Java 语言实现方法重载的本质。演示所用的这段程序属于很极端的例子，除了用作面试题为难求职者之外，在实际工作中几乎不可能有用。笔者拿来演示仅仅是用于讲解重载时目标方法选择的过程，大部分情况下进行这样极端的重载都可算作真正的“奇技”。无论对重载的认识有多么深刻，一个合格的程序员都不应该在实际应用中写如此极端的重载代码。

另外还有一点读者可能比较容易混淆：笔者讲述的解析与分派这两者之间的关系并不是二选一的排他关系，它们是在不同层次上去筛选和确定目标方法的过程。例如，前面说过静态方法会在类加载期就进行解析，而静态方法显然也是可以拥有重载版本的，选择重载版本的过程是通过静态分派完成的。

2. 动态分派

了解了静态分派，我们接下来看一下动态分派的过程，它和多态性的另外一个重要体现^②——重写（Override）有着很密切的关联。我们还是用前面的 `Man` 和 `Women` 一起 `sayHello` 的例子来讲解动态分派，请看代码清单 8-8 中所示的代码。

① 重载中选择最合适方法的过程，可参见 Java 语言规范的 § 15.12.2.5 Choosing the Most Specific Method 章节。

② 有一种观点认为：因为重载是静态的，重写是动态的，所以只有重写算是多态性的体现，而重载不算多态。笔者认为这种争论没有意义，概念仅仅是说明问题的一种工具。

代码清单 8-8 方法动态分派演示

```
package org.fenixsoft.polymorphic;

/**
 * 方法动态分派演示
 * @author zzm
 */
public class DynamicDispatch {

    static abstract class Human {
        protected abstract void sayHello();
    }

    static class Man extends Human {
        @Override
        protected void sayHello() {
            System.out.println("man say hello");
        }
    }

    static class Women extends Human {
        @Override
        protected void sayHello() {
            System.out.println("women say hello");
        }
    }

    public static void main(String[] args) {
        Human man = new Man();
        Human women = new Women();
        man.sayHello();
        women.sayHello();
        man = new Women();
        man.sayHello();
    }
}
```

运行结果：

```
man say hello
women say hello
women say hello
```

216 ❖ 第三部分 虚拟机执行子系统

这个运行结果相信不会出乎任何人的意料，习惯了面向对象思维的 Java 程序员会觉得这是完全理所当然的。我们现在的问题还是和前面的一样，虚拟机是如何知道要调用哪个方法的？

显然这里是不可能根据静态类型来决定的，因为静态类型都是 Human 的两个变量 man 和 women 在调用 sayHello() 方法时执行了不同的行为，并且变量 man 在两次调用中执行了不同的方法。导致这个现象的原因很明显，是这两个变量的实际类型不同，Java 虚拟机是如何根据实际类型来分派方法执行版本的呢？我们使用 javap 命令输出这段代码的字节码，结果如代码清单 8-9 所示：

代码清单 8-9 main() 方法的字节码

```
public static void main(java.lang.String[]);
Code:
  Stack=2, Locals=3, Args_size=1
  0:   new      #16; //class
org/fenixsoft/polymorphic/DynamicDispatch$Man
  3:   dup
  4:   invokespecial  #18; //Method
org/fenixsoft/polymorphic/DynamicDispatch$Man."<init>":()V
  7:   astore_1
  8:   new      #19; //class
org/fenixsoft/polymorphic/DynamicDispatch$Women
 11:  dup
 12:  invokespecial  #21; //Method
org/fenixsoft/polymorphic/DynamicDispatch$Women."<init>":()V
 15:  astore_2
 16:  aload_1
 17:  invokevirtual  #22; //Method
org/fenixsoft/polymorphic/DynamicDispatch$Human.sayHello:()V
 20:  aload_2
 21:  invokevirtual  #22; //Method
org/fenixsoft/polymorphic/DynamicDispatch$Human.sayHello:()V
 24:  new      #19; //class
org/fenixsoft/polymorphic/DynamicDispatch$Women
 27:  dup
 28:  invokespecial  #21; //Method
org/fenixsoft/polymorphic/DynamicDispatch$Women."<init>":()V
 31:  astore_1
 32:  aload_1
```



```
33: invokevirtual #22; //Method
org/fenixsoft/polymorphic/DynamicDispatch$Human.sayHello:()V
36: return
```

0~15 行的字节码是准备动作，作用是建立 man 和 women 的内存空间、调用 Man 和 Women 类型的实例构造器，将这两个实例的引用存放在第 1 和第 2 个局部变量表 Slot 之中，这个动作对应了代码中的这两句：

```
Human man = new Man();
Human women = new Women();
```

接下来的第 16 ~ 21 行是关键部分，第 16 和第 20 两行分别把刚刚创建的两个对象的引用压到栈顶，这两个对象是即将执行的 sayHello() 方法的所有者，称为接收者 (Receiver)；第 17 和第 21 两行是方法调用指令，单从字节码的角度来看，这两条调用指令无论是指令（都是 invokevirtual）还是参数（都是常量池中第 22 项的常量，注释显示了这个常量是 Human.sayHello() 的符号引用）都完全一样，但是这两条指令最终执行的目标方法并不相同，其原因需要从 invokevirtual 指令的多态查找过程开始说起，invokevirtual 指令的运行时解析过程大致分为以下步骤：

- 1) 找到操作数栈顶的第一个元素所指向的对象的实际类型，记作 C。
- 2) 如果在类型 C 中找到与常量中的描述符和简单名称都相符的方法，则进行访问权限校验，如果通过则返回这个方法的直接引用，查找过程结束；不通过则返回 java.lang.IllegalAccessError 异常。
- 3) 否则，按照继承关系从下往上依次对 C 的各个父类进行第 2 步的搜索和验证过程。
- 4) 如果始终没有找到合适的方法，则抛出 java.lang.AbstractMethodError 异常。

由于 invokevirtual 指令执行的第一步就是在运行期确定接收者的实际类型，所以两次调用中的 invokevirtual 指令把常量池中的类方法符号引用解析到了不同的直接引用上，这个过程就是 Java 语言中方法重写的本质。我们把这种在运行期根据实际类型确定方法执行版本的分派过程称为动态分派。

3. 单分派与多分派

方法的接收者与方法的参数统称为方法的宗量，这个定义最早应该来源于《Java 与模式》一书的译文。根据分派基于多少种宗量，可以将分派划分为单分派和多分派两

218 ❖ 第三部分 虚拟机执行子系统

种。单分派是根据一个宗量对目标方法进行选择，多分派则是根据多于一个的宗量对目标方法进行选择。

单分派和多分派的定义相当拗口，不过对照着实例看就不难理解了，代码清单 8-10 中列举了一个 Father 和 Son 一起来做出“一个艰难的决定”的例子。

代码清单 8-10 单分派和多分派

```
/**
 * 单分派、多分派演示
 * @author zzm
 */
public class Dispatch {

    static class QQ {}

    static class _360 {}

    public static class Father {
        public void hardChoice(QQ arg) {
            System.out.println("father choose qq");
        }

        public void hardChoice(_360 arg) {
            System.out.println("father choose 360");
        }
    }

    public static class Son extends Father {
        public void hardChoice(QQ arg) {
            System.out.println("son choose qq");
        }

        public void hardChoice(_360 arg) {
            System.out.println("son choose 360");
        }
    }

    public static void main(String[] args) {
        Father father = new Father();
        Father son = new Son();
        father.hardChoice(new _360());
    }
}
```



```
        son.hardChoice(new QQ());  
    }  
}
```

运行结果：

```
father choose 360  
son choose qq
```

在 main 函数中调用了两次 hardChoice() 方法，这两次 hardChoice() 方法的选择结果在程序输出中已经显示得很清楚了。

再来看看编译阶段编译器的选择过程，即静态分派的过程。这时候选择目标方法的依据有两点：一是静态类型是 Father 还是 Son，二是方法参数是 QQ 还是 360。这次选择结果的最终产物是产生了两条 invokevirtual 指令，两条指令的参数分别为常量池中指向 Father.hardChoice(360) 及 Father.hardChoice(QQ) 方法的符号引用。因为是根据两个宗量进行选择，所以 Java 语言的静态分派属于多分派类型。

再看看运行阶段虚拟机的选择，即动态分派的过程。在执行“son.hardChoice(new QQ())”这句代码时，更准确地说，在执行这句代码所对应的 invokevirtual 指令时，由于编译期已经决定目标方法的签名必须为 hardChoice(QQ)，虚拟机此时不会关心传递过来的参数“QQ”到底是“腾讯 QQ”还是“奇瑞 QQ”，因为这时候参数的静态类型、实际类型都不会对方法的选择构成任何影响，唯一可以影响虚拟机选择的因素只有此方法的接收者的实际类型是 Father 还是 Son。因为只有一个宗量作为选择依据，所以 Java 语言的动态分派属于单分派类型。

根据上述论证的结果，我们可以总结如下：今天（JDK 1.6 时期）的 Java 语言是一门静态多分派、动态单分派的语言。强调“今天的 Java 语言”是因为这个结论未必会恒久不变，C# 在 3.0 及之前的版本与 Java 一样是动态单分派语言，但在 C# 4.0 中引入了 dynamic 类型后，就可以很方便地实现动态多分派。Java 也已经在 JSR-292^①中开始规划对动态语言的支持了，日后将有可能提供类似的动态类型功能。

4. 虚拟机动态分派的实现

前面介绍的分派过程，作为对虚拟机概念模型的解析基本上已经足够了，它已经解

^① JSR-292 : Supporting Dynamically Typed Languages on the Java Platform (Java 平台的动态语言支持)。

决了虚拟机在分派中“会做什么”这个问题。但是虚拟机“具体是如何做到的”，可能各种虚拟机的实现都会有所差别。

由于动态分派是非常频繁的动作，而且动态分派的方法版本选择过程需要运行时在类的方法元数据中搜索合适的目标方法，因此在虚拟机的实际实现中基于性能的考虑，大部分实现都不会真的进行如此频繁地搜索。面对这种情况，最常用的“稳定优化”手段就是为类在方法区中建立一个虚方法表（Virtual Method Table，也称为 vtable，与此对应，在 `invokeinterface` 执行时也会用到接口方法表——Interface Method Table，简称 itable），使用虚方法表索引来代替元数据查找以提高性能。我们先看看代码清单 8-10 所对应的虚方法表结构示例，如图 8-3 所示。

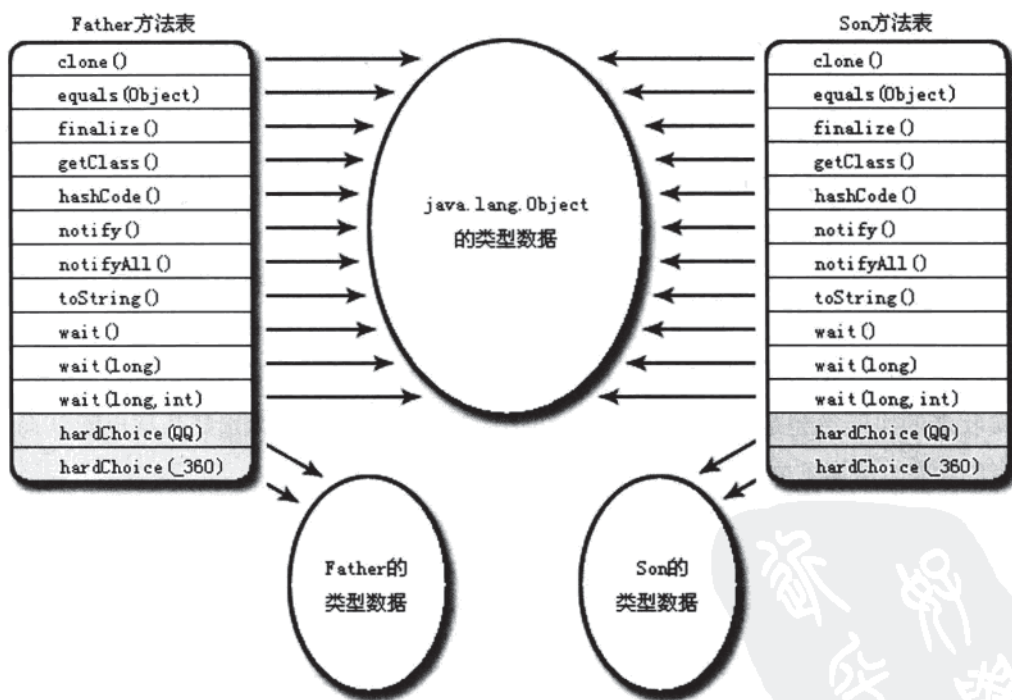


图 8-3 方法表结构

虚方法表中存放着各个方法的实际入口地址。如果某个方法在子类中没有被重写，那么子类的虚方法表里面的地址入口和父类相同方法的地址入口是一致的，都指向父类的实现入口。如果子类中重写了这个方法，子类方法表中的地址将会被替换为

指向子类实现版本的入口地址。图 8-3 中, Son 重写了来自 Father 的全部方法, 因此 Son 的方法表没有指向 Father 类型数据的箭头。但是 Son 和 Father 都没有重写来自 Object 的方法, 所以它们的方法表中所有从 Object 继承来的方法都指向了 Object 的数据类型。

为了程序实现上的方便, 具有相同签名的方法, 在父类、子类的虚方法表中都应当具有一样的索引序号, 这样当类型变换时, 仅需要变更查找的方法表, 就可以从不同的虚方法表中按索引转换出所需的入口地址。

方法表一般在类加载的连接阶段进行初始化, 准备了类的变量初始值后, 虚拟机会把该类的方法表也初始化完毕。

上文中笔者说方法表是分派调用的“稳定优化”手段, 虚拟机除了使用方法表之外, 在条件允许的情况下, 还会使用内联缓存 (Inline Cache) 和基于“类型继承关系分析” (Class Hierarchy Analysis, CHA) 技术的守护内联 (Guarded Inlining) 两种非稳定的“激进优化”手段来获得更高的性能, 关于这两种优化技术的原理和运作过程, 读者可以参考本书第 11 章中的相关内容。

8.4 基于栈的字节码解释执行引擎

关于虚拟机是如何调用方法已经讲解完毕, 从本节开始, 我们来探讨虚拟机是如何执行方法里面的字节码指令的。概述中提到过, 许多 Java 虚拟机的执行引擎在执行 Java 代码的时候都有解释执行 (通过解释器执行) 和编译执行 (通过即时编译器产生本地代码执行) 两种选择, 下面我们先来探讨一下在解释执行时虚拟机执行引擎是如何工作的。

8.4.1 解释执行

Java 语言经常被人们定位为“解释执行”的语言, 在 Java 初生的 JDK 1.0 时代, 这种定义还算是比较准确的, 但当主流的虚拟机中都包含了即时编译器后, Class 文件中的代码到底会被解释执行还是编译执行, 就成了只有虚拟机自己才能准确判断的事。再后来, Java 发展出了可以直接生成本地代码的编译器 (如 GCJ^①, GNU Compiler

^① GCJ: <http://gcc.gnu.org/java/>。

for the Java), 而 C/C++ 语言也出现了通过解释器执行的版本 (如 CINT^①), 这时候再笼统地说“解释执行”对于整个 Java 语言来说几乎就是没有意义的概念了, 只有确定了谈论对象是某种具体的 Java 实现版本和执行引擎运行模式时, 谈解释执行还是编译执行才会比较确切。

不论是解释还是编译, 也不论是物理机还是虚拟机, 对于应用程序, 机器都不可能如人那样阅读和理解, 然后就获得了执行能力。大部分的程序代码到物理机的目标代码或虚拟机能执行的指令集之前, 都需要经过图 8-4 中的各个步骤。如果读者对大学编译原理的相关课程还有印象的话, 很容易就会发现图中下面的那条分支, 就是传统编译原理中程序代码到目标机器代码的生成过程, 而中间的那条分支自然就是解释执行的过程。

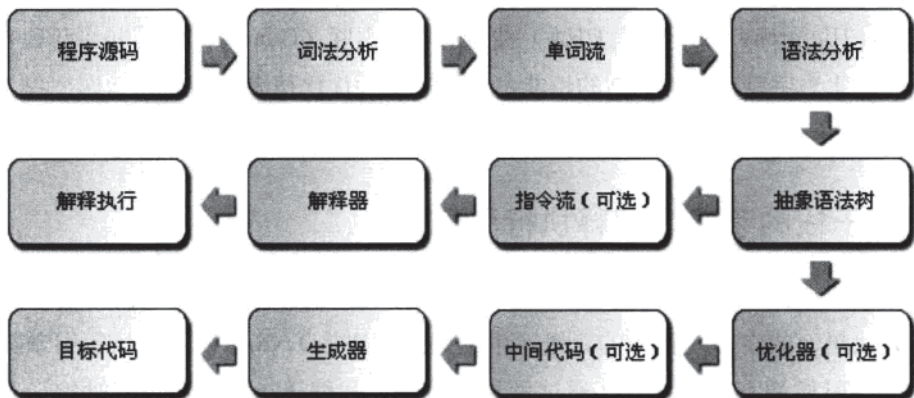


图 8-4 编译过程

如今, 基于物理机、Java 虚拟机或者是非 Java 的其他高级语言虚拟机 (HLLVM) 的语言, 大多都遵循这种基于现代经典编译原理的思路, 在执行前先对程序源码进行词法分析和语法分析处理, 把源码转化为抽象语法树 (Abstract Syntax Tree, AST)。对于一门具体语言的实现来说, 词法和语法分析乃至后面的优化器和目标代码生成器都可以选择独立于执行引擎, 形成一个完整意义的编译器去实现, 这类代表是 C/C++ 语言。也可以选择把其中一部分步骤 (如生成抽象语法树之前的步骤) 实现为一个半独立的编译

① CINT : <http://root.cern.ch/drupal/content/cint>。

器，这类代表是 Java 语言。又或者把这些步骤和执行引擎全部集中封装在一个封闭的黑匣子之中，如大多数的 JavaScript 执行器。

Java 语言中，Javac 编译器完成了程序代码经过词法分析、语法分析到抽象语法树，再遍历语法树生成线性的字节码指令流的过程。因为这一部分动作是在 Java 虚拟机之外进行的，而解释器在虚拟机的内部，所以 Java 程序的编译就是半独立的实现。

8.4.2 基于栈的指令集与基于寄存器的指令集

Java 编译器输出的指令流，基本上^①是一种基于栈的指令集架构（Instruction Set Architecture, ISA），指令流里面的指令大部分都是零地址指令，它们依赖操作数栈进行工作。与之相对的另外一套常用的指令集架构是基于寄存器的指令集，最典型的的就是 x86 的二地址指令集，更通俗一些，就是现在我们主流 PC 中直接支持的指令集架构，这些指令依赖寄存器进行工作。那么，基于栈的指令集与基于寄存器的指令集这两者之间有什么不同呢？

举个最简单的例子，分别使用这两种指令集去计算“1+1”的结果，基于栈的指令集会是这样的：

```
iconst_1
iconst_1
iadd
istore_0
```

两条 `iconst_1` 指令连续地把两个常量 1 压入栈后，`iadd` 指令把栈顶的两个值出栈并相加，然后把结果放回栈顶，最后 `istore_0` 把栈顶的值放到局部变量表的第 0 个 Slot 中。

如果是基于寄存器的指令集，那么程序可能会是这个样子的：

```
mov  eax, 1
add  eax, 1
```

`mov` 指令把 EAX 寄存器的值设为 1，然后 `add` 指令再把这个值加 1，结果就保存在 EAX 寄存器里面。

^① 使用“基本上”是因为部分字节码指令会带有参数，而纯粹基于栈的指令集架构中应当全部都是零地址指令，也就是不存在显式的参数。Java 这样实现主要是考虑了代码的可校验性。

了解了基于栈的指令集与基于寄存器的指令集的区别后，读者可能会有进一步的疑问，这两套指令集谁更好一些呢？

应该说，既然两套指令集同时并存和发展，那么肯定是各有优势的，如果有一套指令集全面优于另外一套的话，就不存在选择的问题。

基于栈的指令集最主要的优点就是可移植性，寄存器由硬件直接提供^①，程序直接依赖这些硬件寄存器则不可避免地要受到硬件的约束。例如，现在 32 位 x86 体系的处理器中提供了 8 个 32 位的寄存器，而 ARM 体系的 CPU（在当前的手机、PDA 中相当流行的一种处理器）则提供了 16 个 32 位的通用寄存器。如果使用栈架构的指令集，用户程序不会直接用到这些寄存器，那就可以由虚拟机实现来自行决定把一些访问最频繁的数据（程序计数器、栈顶缓存等）放到寄存器中以获取尽量好的性能，这样实现起来也更加简单。栈架构的指令集还有一些其他优点，如代码相对更紧凑（字节码中每个字节就对应一条指令，而多地址指令集中还需要存放参数）、编译器实现更加简单（不需要考虑空间分配的问题，所需空间都在栈上操作）等。

栈架构指令集的主要缺点是执行速度相对来说稍慢一些。所有主流物理机的指令集都是寄存器架构也从侧面印证了这一点。

栈架构指令集的代码虽然紧凑，但是完成相同功能所需的指令数量一般会比寄存器架构多，因为出栈、入栈操作本身就产生了相当多的指令。更重要的是栈实现在内存之中，频繁的栈访问也就意味着频繁的内存访问，相对于处理器来说，内存始终是执行速度的瓶颈。尽管虚拟机可以采取栈顶缓存的手段，把最常用的操作映射到寄存器中以避免直接内存访问，但这也只能是优化措施而不是解决本质问题的方法。因此，由于指令数量和内存访问的原因，导致了栈架构指令集的执行速度相对较慢。

8.4.3 基于栈的解释器执行过程

初步的理论已经讲解完了，本小节准备了一段 Java 代码，看看在虚拟机中实际上是如何执行的。前面曾经举过一个计算“1+1”的例子，下面给出的例子涉及的是四则运算加减乘除法，如代码清单 8-11 所示。

① 这里说的是物理机器上的寄存器。也有基于寄存器的虚拟机，如 Google Android 平台的 Dalvik VM。即使是基于寄存器的虚拟机，也会希望把虚拟机寄存器尽量映射到物理寄存器上以获取尽可能高的性能。

代码清单 8-11 一段简单的算术代码

```
public int calc() {  
    int a = 100;  
    int b = 200;  
    int c = 300;  
    return (a + b) * c;  
}
```

这段代码从 Java 语言的角度没有任何评论的必要，直接使用 javap 命令看看它的字节码指令，如代码清单 8-12 所示。

代码清单 8-12 一段简单的算术代码的字节码表示

```
public int calc();  
Code:  
Stack=2, Locals=4, Args_size=1  
0:  bipush  100  
2:  istore_1  
3:  sipush  200  
6:  istore_2  
7:  sipush  300  
10: istore_3  
11: iload_1  
12: iload_2  
13: iadd  
14: iload_3  
15: imul  
16: ireturn  
}
```

javap 提示这段代码需要深度为 2 的操作数栈和 4 个 Slot 的局部变量空间，笔者就根据这些信息画了图 8-5 至图 8-11 共七张图片，来描述代码清单 8-12 执行过程中的代码、操作数栈和局部变量表的变化情况。

首先，执行偏移地址为 0 的指令，bipush 指令的作用是将单字节的整型常量值（-128 ~ 127）推入操作数栈顶，后跟一个参数，指明推送的常量值，这里是 100。

执行偏移地址为 1 的指令，istore_1 指令的作用是将操作数栈顶的整型值出栈并存放到第 1 个局部变量 Slot 中。后续四条指令（直到偏移为 11 的指令为止）都是做同样的事情，也就是在对应代码中把变量 a、b、c 赋值为 100、200、300。这四条指令的图示略过。



图 8-5 执行偏移地址为 0 的指令的情况

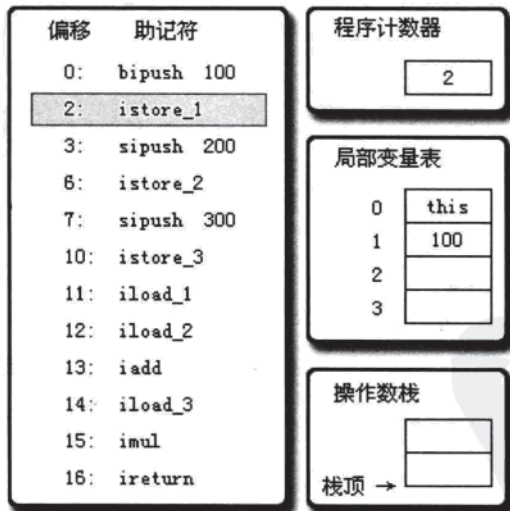


图 8-6 执行偏移地址为 1 的指令的情况

执行偏移地址为 11 的指令，`iload_1` 指令的作用是将局部变量表第 1 个 Slot 中的整型值复制到操作数栈顶。

执行偏移地址为 12 的指令，`iload_2` 指令的执行过程与 `iload_1` 类似，把第 2 个 Slot 的整型值入栈。画出这个指令的图示主要是为了显示下一条 `iadd` 指令执行前的堆栈状况。

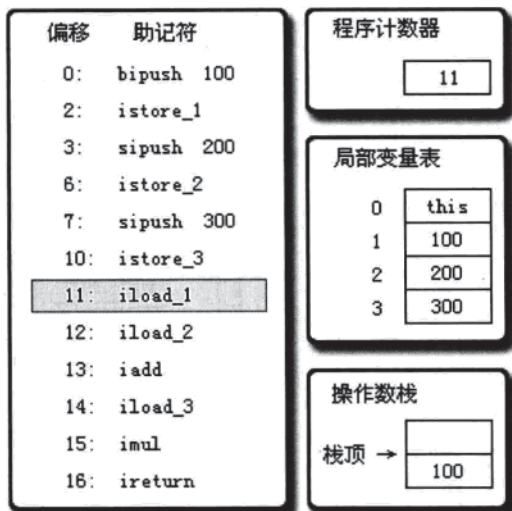


图 8-7 执行偏移地址为 11 的指令的情况

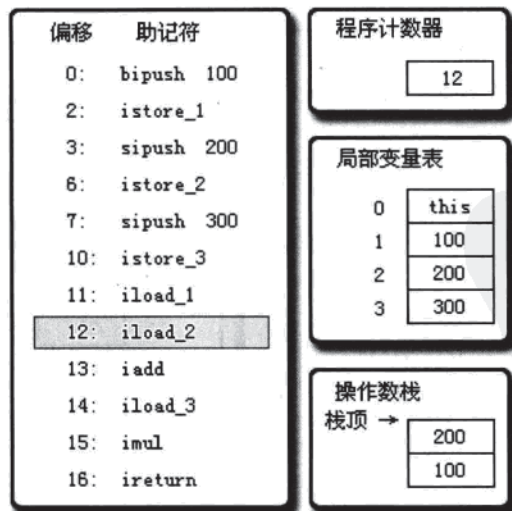


图 8-8 执行偏移地址为 12 的指令的情况

执行偏移地址为 13 的指令，iadd 指令的作用是将操作数栈中前两个栈顶元素出栈，做整型加法，然后把结果重新入栈。在 iadd 指令执行完毕后，栈中原有的 100 和 200 出栈，它们的和 300 重新入栈。

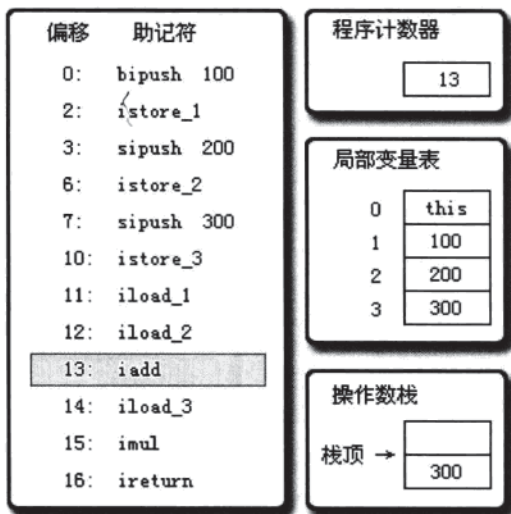


图 8-9 执行偏移地址为 13 的指令的情况



图 8-10 执行偏移地址为 14 的指令的情况

执行偏移地址为 14 的指令，`iload_3` 指令把存放在第 3 个局部变量 Slot 中的 300 入栈到操作数栈中。这时操作数栈为两个整数 300。下一条指令 `imul` 是将操作数栈中前两个栈顶元素出栈，做整型乘法，然后把结果重新入栈，这里与 `iadd` 完全类似，所以笔者省略图示。

执行偏移地址为 16 的指令，`ireturn` 指令是方法返回指令之一，它将结束方法执行并将操作数栈顶的整型值返回给此方法的调用者。到此为止，这段方法执行结束。



图 8-11 执行偏移地址为 16 的指令的情况

上面的执行过程仅仅是一种概念模型，虚拟机最终会对执行过程做出一些优化来提高性能，实际的运作过程不一定完全符合概念模型的描述，更准确地说，实际情况会和上面描述的概念模型有非常大的差距，差距产生的原因是虚拟机中解析器和即时编译器都会对输入的字节码进行优化，例如，HotSpot 虚拟机中就有许多以“fast_”开头的非标准字节码指令用于合并和替换输入的字节码以提升解释执行性能，即时编译器的优化手段更是花样繁多^①。

不过我们从这段程序的执行中也可以看出栈结构指令集的一般运行过程，整个运算过程的中间变量都以操作数栈的出栈和入栈为信息交换途径，符合我们在前面分析的特点。

^① 具体可以参考第 11 章中的相关内容。

8.5 本章小结

本章中，我们分析了虚拟机在执行代码时如何找到正确的方法，如何执行方法内的字节码，以及执行代码时涉及的内存结构。在第 6、7、8 三章中，我们针对 Java 程序是如何存储的、如何载入（创建）的以及如何执行的问题把相关知识讲解了一下，下一章我们将一起看看这些理论知识在具体开发中的经典应用。



第9章 类加载及执行子系统的案例与实战

本章主要内容

- 概述
- 案例分析
- 实战：自己动手实现远程执行功能

代码编译的结果从本地机器码转变为字节码，是存储格式发展的一小步，却是编程语言发展的一大步。

9.1 概述

在 Class 文件格式与执行引擎这部分里，用户的程序能直接影响的内容并不太多，Class 文件以何种格式存储，类型何时加载、如何连接，以及虚拟机如何执行字节码指令等都是由虚拟机直接控制的行为，用户程序无法对其进行改变。能通过程序进行操作的，主要是字节码生成与类加载器这两部分的功能，但仅仅在如何处理这两点上，就已经出现了许多值得欣赏和借鉴的思路，这些思路后来成为了许多常用功能和程序实现的基础。在本章中，我们将看一下前面所学的知识在实际开发之中是如何应用的。

9.2 案例分析

在案例分析部分，笔者准备了四个例子，关于类加载器和字节码的案例各有两个。并且这两个领域的案例中各有一个案例是大多数 Java 开发人员都使用过的工具或技术，另外一个案例虽然不一定每个人都使用过，但特别精彩地演绎出这个领域中的技术特性。希望这些案例能引起你的思考，并给你的日常工作带来灵感。

9.2.1 Tomcat：正统的类加载器架构

主流的 Java Web 服务器，如 Tomcat、Jetty、WebLogic、WebSphere 或其他笔者没有列举的服务器，都实现了自己定义的类加载器（一般都不止一个）。因为一个功能健全的 Web 服务器，都要解决如下几个问题：

- 部署在同一个服务器上的两个 Web 应用程序所使用的 Java 类库可以实现相互隔离。这是最基本的需求，两个不同的应用程序可能会依赖同一个第三方类库的不同版本，不能要求一个类库在一个服务器中只有一份，服务器应当可以保证两个应用程序的类库可以互相独立使用。
- 部署在同一个服务器上的两个 Web 应用程序所使用的 Java 类库可以互相共享。这个需求也很常见，例如用户可能有 10 个使用 Spring 组织的应用程序部署在同一台服务器上，如果把 10 份 Spring 分别存放在各个应用程序的隔离目录中，将会是很大的资源浪费——这主要倒不是浪费磁盘空间的问题，而是指类库在使用时都要被加载到服务器内存，如果类库不能共享，虚拟机的方法区很容易就会出现过度膨胀的风险。
- 服务器需要尽可能地保证自身的安全不受部署的 Web 应用程序影响。目前，有许多主流的 Java Web 服务器自身也是使用 Java 语言来实现的。因此服务器本身也有类库依赖的问题，一般来说，基于安全考虑，服务器所使用的类库应该与应用程序的类库互相独立。
- 支持 JSP 应用的 Web 服务器，十有八九都需要支持 HotSwap 功能。我们知道 JSP 文件最终要被编译成 Java Class 才能被虚拟机执行，但 JSP 文件由于其纯文本存储的特性，被运行时修改的概率远远大于第三方类库或程序自己的 Class 文件。而且 ASP、PHP 和 JSP 这些网页应用也把修改后无须重启作为一个很大的“优势”来看待，因此“主流”的 Web 服务器都会支持 JSP 生成类的热替换，当然也有“非主流”的，如运行在生产模式（Production Mode）下的 WebLogic 服务器默认就不会处理 JSP 文件的变化。

由于存在上述问题，在部署 Web 应用时，单独的一个 ClassPath 就无法满足需求了，所以各种 Web 服务器都不约而同地提供了好几个 ClassPath 路径供用户存放第三方类库，这些路径一般都以“lib”或“classes”命名。被放置到不同路径中的类库，具备不同的

访问范围和服务对象，通常，每一个目录都会有一个相应的自定义类加载器去加载放置在里面的 Java 类库。现在，笔者就以 Tomcat 服务器^①为例，看一看 Tomcat 具体是如何规划用户的类库结构和类加载器的。

在 Tomcat 目录结构中，有三组目录（“/common/*”、“/server/*”和“/shared/*”）可以存放 Java 类库，另外还可以加上 Web 应用程序自身的目录“/WEB-INF/*”，一共四组，把 Java 类库放置在这些目录中的含义分别是：

- 放置在 /common 目录中：类库可被 Tomcat 和所有的 Web 应用程序共同使用。
- 放置在 /server 目录中：类库可被 Tomcat 使用，对所有的 Web 应用程序都不可见。
- 放置在 /shared 目录中：类库可被所有的 Web 应用程序共同使用，但对 Tomcat 自己不可见。
- 放置在 /WebApp/WEB-INF 目录中：类库仅仅可以被此 Web 应用程序使用，对 Tomcat 和其他 Web 应用程序都不可见。

为了支持这套目录结构，并对目录里面的类库进行加载和隔离，Tomcat 自定义了多个类加载器，这些类加载器按照经典的双亲委派模型来实现，其关系如图 9-1 所示。

灰色背景的三个类加载器是 JDK 默认提供的类加载器，这三个加载器在第 7 章中已经详细介绍过了。而 CommonClassLoader、CatalinaClassLoader、SharedClassLoader 和 WebappClassLoader 则是 Tomcat 自己定义的类加载器，它们分别加载 /common/*、/server/*、/shared/* 和 /WebApp/WEB-INF/* 中 Java 类库的逻辑。其中 WebApp 类加载器和 Jsp 类加载器通常会存在多个实例，每一个 Web 应用程序对应一个 WebApp 类加载器，每一个 JSP 文件对应一个 Jsp 类加载器。

从图 9-1 的委派关系中可以看出，CommonClassLoader 能加载的类都可以被 CatalinaClassLoader 和 SharedClassLoader 使用，而 CatalinaClassLoader 和 SharedClassLoader 自己能加载的类则与对方相互隔离。WebAppClassLoader 可以使用 SharedClassLoader 加载到的类，但各个 WebAppClassLoader 实例之间相互隔离。而 JasperLoader 的加载范围仅仅是这个 JSP 文件所编译出来的那一个 Class，它出现的目的是为了被丢弃：当服

① Tomcat 是 Apache 基金会下一款开源的 Java Web 服务器，主页地址为：<http://tomcat.apache.org>。本案例中选用的是 Tomcat 5.x 服务器的目录和类加载器结构，在 Tomcat 6.x 的默认配置下，/common、/server 和 /shared 三个目录已经合并到一起了。

务器检测到 JSP 文件被修改时，会替换掉目前的 JasperLoader 的实例，并通过再建立一个新的 Jsp 类加载器来实现 JSP 文件的 HotSwap 功能。

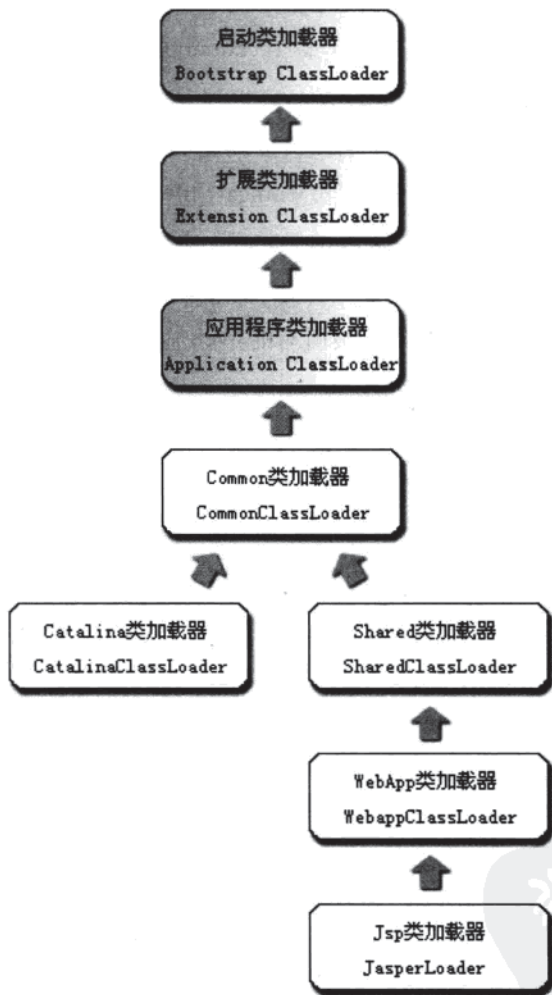


图 9-1 Tomcat 服务器的类加载架构

对于 Tomcat 的 6.x 版本，只有指定了 tomcat/conf/catalina.properties 配置文件的 server.loader 和 share.loader 项后才会真正建立 CatalinaClassLoader 和 SharedClassLoader 的实例，否则会用到这两个类加载器的地方都会用 CommonClassLoader 的实例来代替，而默认的配置文件中没有设置这两个 loader 项，所以 Tomcat 6.x 顺理成章地把

`/common`、`/server` 和 `/shared` 三个目录默认合并到一起变成一个 `/lib` 目录，这个目录里的类库相当于以前 `/common` 目录中类库的作用。这是 Tomcat 设计团队为了简化大多数的部署场景所做的一项改进，如果默认设置不能满足需要，用户可以通过修改配置文件指定 `server.loader` 和 `share.loader` 的方式重新启用 Tomcat 5.x 的加载器架构。

Tomcat 加载器的实现清晰易懂，并且采用了官方推荐的“正统”的使用类加载器的方式。如果读者阅读完上面的案例后，能毫不费力地完全理解 Tomcat 设计团队这样布置加载器架构的用意，那说明您已经大致掌握了类加载器“主流”的使用方式，那么笔者不妨再提一个问题让各位思考一下：前面曾经提到过一个场景，如果有 10 个 Web 应用程序都是用 Spring 来进行组织和管理的的话，可以把 Spring 放到 `Common` 或 `Shared` 目录下让这些程序共享。Spring 要对用户程序的类进行管理，自然要能访问到用户程序的类，而用户的程序显然是放在 `/WebApp/WEB-INF` 目录中的。那么被 `CommonClassLoader` 或 `SharedClassLoader` 加载的 Spring 如何访问并不在其加载范围内的用户程序呢？如果你读过了本书第 7 章的相关内容，相信回答这个问题一样会毫不费力。

9.2.2 OSGi：灵活类加载器架构

Java 程序社区中流传着这么一个观点：“学习 JEE 规范，去看 JBoss 源码；学习类加载器，就去看 OSGi 源码”。尽管“JEE 规范”和“类加载器的知识”并不是一个对等的概念，不过，既然这个观点能在程序员中流传开来，也从侧面说明了 OSGi 对类加载器的运用确实有其独到之处。

OSGi^①（Open Service Gateway Initiative）是 OSGi 联盟（OSGi Alliance）制订的一个基于 Java 语言的动态模块化规范，这个规范最初由 Sun、IBM、爱立信等公司联合发起，目的是使服务提供商通过住宅网关为各种家用智能设备提供各种服务，后来这个规范在 Java 的其他技术领域也有相当不错的发展，现在已经成为 Java 世界中“事实上”的模块化标准，并且已经有了 Equinox、Felix 等成熟的实现。OSGi 在 Java 程序员中最著名的应用案例就是 Eclipse IDE，另外还有许多大型的软件平台和中间件服务器都基于或声明将会基于 OSGi 规范来实现，如 IBM Jazz 平台、GlassFish 服务器、Weblogic

① 官方站点：<http://www.osgi.org/Main/HomePage>。

10.3 所使用的 mSA 架构等。

OSGi 中的每个模块（称为 Bundle）与普通的 Java 类库区别并不太大，两者一般都以 JAR 格式进行封装，并且内部存储的都是 Java Package 和 Class。但是一个 Bundle 可以声明它所依赖的 Java Package（通过 Import-Package 描述），也可以声明它允许导出发布的 Java Package（通过 Export-Package 描述）。在 OSGi 里面，Bundle 之间的依赖关系从传统的上层模块依赖底层模块转变为平级模块之间的依赖（至少外观上是如此），而且类库的可见性能得到了非常精确的控制，一个模块里只有被 Export 过的 Package 才可能被外界访问，其他的 Package 和 Class 将会被隐藏起来。除了更精确的模块划分和可见性控制外，引入 OSGi 的另外一个重要理由是，基于 OSGi 的程序很可能（只是很可能，并不是一定会）可以实现模块级的热插拔功能，当程序升级更新或调试除错时，可以只停用、重新安装然后启用程序的其中一部分，这对企业级程序开发来说是一个非常具有诱惑力的特性。

OSGi 之所以能有上述诱人的特点，要归功于它灵活的类加载器架构。OSGi 的 Bundle 类加载器之间只有规则，没有固定的委派关系。例如，某个 Bundle 声明了一个它依赖的 Package，如果有其他 Bundle 声明发布了这个 Package 后，那么对这个 Package 的所有类加载动作都会委派给发布它的 Bundle 类加载器去完成。不涉及某个具体的 Package 时，各个 Bundle 加载器都是平级的关系，只有具体使用到某个 Package 和 Class 的时候，才会根据 Package 导入导出定义来构造 Bundle 间的委派和依赖。

另外，一个 Bundle 类加载器为其他 Bundle 提供服务时，会根据 Export-Package 列表严格控制访问范围。如果一个类存在于 Bundle 的类库中但是没有被 Export，那么这个 Bundle 的类加载器能找到这个类，但不会提供给其他 Bundle 使用，而且 OSGi 平台也不会把其他 Bundle 的类加载请求分配给这个 Bundle 来处理。

我们可以举一个更具体一些的简单例子，假设存在 Bundle A、Bundle B 和 Bundle C 三个模块，并且这三个 Bundle 定义的依赖关系为：

- Bundle A：声明发布了 packageA，依赖了 java.* 的包；
- Bundle B：声明依赖了 packageA 和 packageC，同时也依赖了 java.* 的包；
- Bundle C：声明发布了 packageC，依赖了 packageA。

那么，这三个 Bundle 之间的类加载器及父类加载器之间的关系如图 9-2 所示。

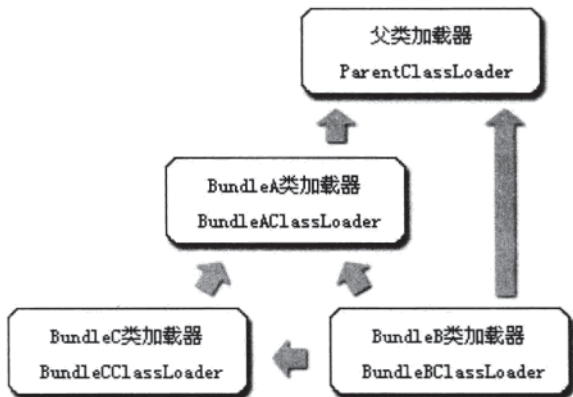


图 9-2 OSGi 的类加载器架构

由于没有牵扯到具体的 OSGi 实现，图中的类加载器都没有指明具体的加载器实现，只是一个体现了加载器间关系的概念模型，并且只是体现了 OSGi 中最简单的加载器委派关系。一般来说，在 OSGi 里，加载一个类可能发生的查找行为和委派关系会比图 9-2 中显示的复杂得多，类加载时可能进行的查找规则如下：

- ❑ 以 `java.*` 开头的类，委派给父类加载器加载。
- ❑ 否则，委派列表名单内的类，委派给父类加载器加载。
- ❑ 否则，Import 列表中的类，委派给 Export 这个类的 Bundle 的类加载器加载。
- ❑ 否则，查找当前 Bundle 的 Classpath，使用自己的类加载器加载。
- ❑ 否则，查找是否在自己的 Fragment Bundle 中，如果是则委派给 Fragment Bundle 的类加载器加载。
- ❑ 否则，查找 Dynamic Import 列表的 Bundle，委派给对应 Bundle 的类加载器加载。
- ❑ 否则，类查找失败。

从图 9-2 中还可以看出，在 OSGi 里面，加载器之间的关系不再是双亲委派模型的树形结构，而是已经进一步发展成了一种运行时才能确定的网状结构。这种网状的类加载器架构在带来更优秀的灵活性的同时，也可能会产生许多新的隐患。笔者曾经参与过将一个非 OSGi 的大型系统向 Equinox OSGi 平台迁移的项目，由于历史原因，代码模块之间的依赖关系错综复杂，勉强分离出各个模块的 Bundle 后，发现在高并发环境下经常出现死锁。我们很容易就找到了死锁的原因：如果出现了 Bundle A 依赖 Bundle B 的 Package B，而 Bundle B 又依赖了 Bundle A 的 Package A，这两个 Bundle 进行类加

载时就很容易发生死锁。具体情况是当 Bundle A 加载 Package B 的类时，首先需要锁定当前类加载器的实例对象（`java.lang.ClassLoader.loadClass()` 是一个 `synchronized` 方法），然后把请求委派给 Bundle B 的加载器处理，但如果这时候 Bundle B 也正好想加载 Package A 的类，它也先锁定自己的加载器再去请求 Bundle A 的加载器处理，这样两个加载器都在等待对方处理自己的请求，而对方处理完之前自己又一直处于同步锁定的状态，因此它们就互相死锁，永远无法完成加载请求了。Equinox 的 Bug List 中也有关于这类问题的 Bug^①，并提供了一个以牺牲性能为代价的解决方案——用户可以启用 `osgi.classloader.singleThreadLoads` 参数来按单线程串行化的方式强制进行类加载动作。在 JDK 1.7 中，将会为非树状继承关系下的类加载器架构进行一次专门的升级^②，希望从底层避免这类死锁状况，这个动作也是为将来实现“官方的”模块化规范（与前文所提的 OSGi 是“事实上”的模块化规范对应）——JSR-297、JSR-277 做准备。

总体来说，OSGi 描绘了一个很美好的模块化开发的目标，而且定义了实现这个目标所需要的各种服务，同时也有成熟框架对其提供实现支持。对于单个虚拟机下的应用，从开发初期就建立在 OSGi 上是一个很不错的选择，这样便于约束依赖。但并非所有的应用都适合采用 OSGi 作为基础架构，OSGi 在提供强大功能的同时，也引入了额外的复杂度，带来了线程死锁和内存泄漏的风险。

9.2.3 字节码生成技术与动态代理的实现

“字节码生成”并不是什么高深的技术，读者在看到“字节码生成”这个标题时也不必先去想诸如 Javassist、CGLib 和 ASM 之类的字节码类库，因为 JDK 里面的 `javac` 命令就是字节码生成技术的“老祖宗”，并且 `javac` 也是一个由 Java 语言写成的程序，它的代码存放在 OpenJDK 的 `jdk7/langtools/src/share/classes/com/sun/tools/javac` 目录中^③。要深入了解字节码生成，阅读 `javac` 的源码是个很好的途径，不过 `javac` 对于我们这个例子来说太过庞大了。在 Java 里面除了 `javac` 和字节码类库外，使用到字节码生成的例子还有很多，如 Web 服务器中的 JSP 编译器，编译时织入的 AOP 框架，还有很常用的动态代理技术，甚至在使用反射的时候虚拟机都有可能会在运行时生成字节码来提

① Bug-121737 : https://bugs.eclipse.org/bugs/show_bug.cgi?id=121737。

② JDK 1.7-Upgrade class-loader architecture : <http://openjdk.java.net/projects/jdk7/features/#f352>。

③ 如何获取 OpenJDK 源码，请参见本书第 1 章的相关内容。

高执行速度。我们选择其中相对简单的动态代理来看看字节码生成技术是如何影响程序运作的。

相信许多 Java 开发人员都使用过动态代理，即使没有直接使用过 `java.lang.reflect.Proxy` 或实现过 `java.lang.reflect.InvocationHandler` 接口，应该也用过 Spring 来做过 Bean 的组织管理。如果使用过 Spring，那大多数情况下就都用过动态代理，因为如果 Bean 是面向接口编程，那么在 Spring 内部则都是通过动态代理的方式来对 Bean 进行增强的。动态代理中所谓的“动态”，是针对使用 Java 代码实际编写了代理类的“静态”代理而言的，它的优势不在于省去了编写代理类那一点工作量，而是实现了可以在原始类和接口还未知的时候，就确定代理类的代理行为，当代理类与原始类脱离直接联系后，就可以很灵活地重用于不同的应用场景之中。

代码清单 9-1 演示了一个最简单的动态代理的用法，原始的逻辑是打印一句“hello world”，代理类的逻辑是在原始类的方法执行前打印一句“welcome”。我们先看一下代码，然后再分析 JDK 是如何做到的。

代码清单 9-1 动态代理的简单示例

```
public class DynamicProxyTest {  
  
    interface IHello {  
        void sayHello();  
    }  
  
    static class Hello implements IHello {  
        @Override  
        public void sayHello() {  
            System.out.println("hello world");  
        }  
    }  
  
    static class DynamicProxy implements InvocationHandler {  
  
        Object originalObj;  
  
        Object bind(Object originalObj) {  
            this.originalObj = originalObj;  
            return Proxy.newProxyInstance(originalObj.getClass().getClassLoader(),  
originalObj.getClass().getInterfaces(), this);  
        }  
    }  
}
```

240 ❖ 第三部分 虚拟机执行子系统

```

    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        System.out.println("welcome");
        return method.invoke(originalObj, args);
    }
}

public static void main(String[] args) {
    IHello hello = (IHello) new DynamicProxy().bind(new Hello());
    hello.sayHello();
}
}
}

```

运行结果如下：

```

welcome
hello world

```

上述代码中，唯一的“黑匣子”就是 `Proxy.newProxyInstance()` 方法，除此之外再没有任何特殊之处。这个方法返回一个实现了 `IHello` 的接口，并且代理了 `new Hello()` 实例行为的对象。跟踪这个方法的源码，可以看到程序进行了验证、优化、缓存、同步、生成字节码和显式类加载等操作，前面的步骤并不是我们关注的重点，而最后它调用了 `sun.misc.ProxyGenerator.generateProxyClass()` 方法来完成生成字节码的动作，这个方法可以在运行时产生一个描述代理类的字节码 `byte[]` 数组。如果想看一看这个方法在运行时产生的代理类中写了些什么，可以在 `main()` 方法中加入下面这句：

```
System.getProperties().put("sun.misc.ProxyGenerator.saveGeneratedFiles", "true");
```

加入这句代码后再次运行程序，磁盘中将会产生一个名为“`$Proxy0.class`”的代理类 `Class` 文件，反编译后可以看见如代码清单 9-2 所示的内容。

代码清单 9-2 反编译的动态代理类的代码

```

package org.fenixsoft.bytecode;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;

```

```
import java.lang.reflect.Proxy;
import java.lang.reflect.UndeclaredThrowableException;

public final class $Proxy0 extends Proxy
    implements DynamicProxyTest.IHello
{
    private static Method m3;
    private static Method m1;
    private static Method m0;
    private static Method m2;

    public $Proxy0(InvocationHandler paramInvocationHandler)
        throws
    {
        super(paramInvocationHandler);
    }

    public final void sayHello()
        throws
    {
        try
        {
            this.h.invoke(this, m3, null);
            return;
        }
        catch (RuntimeException localRuntimeException)
        {
            throw localRuntimeException;
        }
        catch (Throwable localThrowable)
        {
            throw new UndeclaredThrowableException(localThrowable);
        }
    }

    // 此处由于版面原因，省略了 equals()、hashCode()、toString() 三个方法的代码
    // 这 3 个方法的内容与 sayHello() 非常相似

    static
    {
        try
        {
```



242 ❖ 第三部分 虚拟机执行子系统

```

        m3 = Class.forName("org.fenixsoft.bytecode.DynamicProxyTest$IHello").
getMethod("sayHello", new Class[0]);
        m1 = Class.forName("java.lang.Object").getMethod("equals", new Class[] {
Class.forName("java.lang.Object") });
        m0 = Class.forName("java.lang.Object").getMethod("hashCode", new Class[0]);
        m2 = Class.forName("java.lang.Object").getMethod("toString", new Class[0]);
        return;
    }
    catch (NoSuchMethodException localNoSuchMethodException)
    {
        throw new NoSuchMethodError(localNoSuchMethodException.getMessage());
    }
    catch (ClassNotFoundException localClassNotFoundException)
    {
        throw new NoClassDefFoundError(localClassNotFoundException.getMessage());
    }
}
}
}

```

这个代理类的实现代码也很简单，它为传入接口中的每一个方法，以及从 `java.lang.Object` 中继承来的 `equals()`、`hashCode()`、`toString()` 方法都生成了对应的实现，并且统一调用了 `InvocationHandler` 对象的 `invoke()` 方法（代码中的“this.h”就是父类 `Proxy` 中保存的 `InvocationHandler` 实例变量）来实现这些方法的内容，各个方法的区别不过是传入的参数和 `Method` 对象有所不同而已，所以无论调用动态代理的哪一个方法，实际上都是在执行 `InvocationHandler.invoke()` 中的代理逻辑。

这个例子中并没有讲到 `generateProxyClass()` 方法具体是如何产生代理类“`$Proxy0.class`”的字节码的，大致的生成过程其实就是根据 `Class` 文件的格式规范去拼装字节码，但是在实际开发中，以 `byte` 为单位直接拼装出字节码的应用场合很少见，这种生成方式也只能产生一些高度模板化的代码。对于用户的程序代码来说，如果有要大量操作字节码的需求，还是使用封装好的字节码类库比较合适。如果你对动态代理的字节码拼装过程确实很感兴趣，可以在 `OpenJDK` 的 `jdk/src/share/classes/sun/misc` 目录下找到 `sun.misc.ProxyGenerator` 的源码。

9.2.4 Retrotranslator：跨越 JDK 版本

一般来说，以“做项目”为主的软件公司比较容易更新技术，在下一个项目中换一

个技术框架、升级到最时髦的 JDK 版本、甚至把 Java 换成 C# 来开发都是有可能的。但是当公司发展壮大，技术有所积累，逐渐成为以“做产品”为主的软件公司后，自主选择技术的权利就会丧失掉，因为之前所积累的代码和技术都是用真金白银砸出来的，一个稳健的团队也不会随意地改变底层的技术。然而在飞速发展的程序设计领域，新技术总是日新月异层出不穷，偏偏这些新技术又如鲜花之于蜜蜂、美酒之于醉汉一样，对程序员们有着难以抗拒的吸引力。

在 Java 的世界里，每一次 JDK 大版本的发布，就伴随着一场大规模的技术革新，而对 Java 程序编写习惯改变最大的，无疑是 JDK 1.5 的发布。自动装箱、泛型、动态注解、枚举、变长参数、遍历循环（foreach 循环）……事实上在没有这些语法特性的年代，Java 程序也照样能写，但是现在看来，上述每一种语法的改进几乎都是“必不可少”的。就如用习惯了 24 寸液晶显示器的程序员，很难再在 15 寸纯平显示器上编写代码。但假如“不幸”因为要保护现有投资、维持程序结构稳定等，必须使用 1.5 以前版本的 JDK 呢？幸好，我们没有办法把 15 寸纯平显示器变成 24 寸液晶显示器，但却可以跨越 JDK 版本之间的沟壑，把 JDK 1.5 中编写的代码放到 JDK 1.4 或 1.3 的环境中去部署和使用。为了解决这个问题，一种名为“Java 逆向移植”的工具（Java Backporting Tools）应运而生，Retrotranslator^①是这类工具中最出色的一个。

Retrotranslator 的作用是将 JDK 1.5 编译出来的 Class 文件转变为可以在 JDK 1.4 或 1.3 上部署的版本，它可以很好地支持自动装箱、泛型、动态注解、枚举、变长参数、遍历循环、静态导入这些语法特性，甚至还可以支持 JDK 1.5 中新增的集合改进、并发包及对泛型、注解等的反射操作……嗯，差不多了，毕竟 Retrotranslator 也没有给笔者广告赞助费。了解了 Retrotranslator 这种逆向移植工具可以做什么后，现在关心的是它是怎样做到的？

要想知道 Retrotranslator 是如何在旧版本 JDK 中模拟新版本 JDK 的功能，首先要搞清楚 JDK 升级中会提供哪些新的功能。JDK 的每次升级中新增的功能大致可以分为以下四类：

- 在编译器层面所做的改进。如自动装箱拆箱，实际上就是编译器在程序中使用到包装对象的地方自动插入了很多 `Integer.valueOf()` 和 `Float.valueOf()` 之类的代码；

^① 官方站点：<http://retrotranslator.sf.net>。

变长参数在编译之后就被自动转化成了一个数组来完成参数传递；泛型的信息则在编译阶段就已经被擦除掉了（但是在元数据中还保留着），相应的地方被编译器自动插入了类型转换代码^①。

- ❑ 对 Java API 的代码增强。譬如 JDK 1.2 时代引入的 `java.util.Collections` 等一系列集合类，在 JDK 1.5 时代引入的 `java.util.concurrent` 并发包等。
- ❑ 需要在字节码中进行支持的改动。如 JDK 1.7 里面新加入的语法特性：动态语言支持，就需要在虚拟机中新增一条 `invokedynamic` 字节码指令来实现相关的调用功能。不过字节码指令集一直处于相对比较稳定的状态，这种需要在字节码层面直接进行的改动是比较少见的。
- ❑ 虚拟机内部的改进。如 JDK 1.5 中实现的 JSR-133^② 规范重新定义的 Java 内存模型（JMM, Java Memory Model）、CMS 收集器之类的改动，这类改动对于程序员编写代码基本是透明的，但会对程序运行时产生影响。

上述四类新功能中，Retrotranslator 只能模拟前两类，对于后面两类直接在虚拟机内部实现的改进，一般所有的逆向移植工具都无能为力，至少不能完整地或者在可接受的效率上完成全部模拟，否则虚拟机设计团队也没有必要舍近求远地改动处于 JDK 底层的虚拟机。在可以模拟的两类功能中，第二类模拟相对更容易实现一些，如 JDK 1.5 引入的 `java.util.concurrent` 包，实际是由多线程大师 Doug Lea 开发的一套并发包，在 JDK 1.5 出现之前就已经存在（那时候名字叫做 `dl.util.concurrent`，引入 JDK 时由作者的 JDK 开发团队共同做出了一些改进），所以要在旧的 JDK 中支持这部分功能，以独立类库的方式便可实现。Retrotranslator 中就附带了一个名为“`backport-util-concurrent.jar`”的类库（由另一个名为“Backport of JSR 166”的项目所提供）来代替 JDK 1.5 的并发包。

至于 JDK 在编译阶段进行处理的那些改进，Retrotranslator 则是使用 ASM 框架直接对字节码进行处理。由于组成 Class 文件的字节码指令数量并没有改变，所以无论是 JDK 1.3、JDK 1.4 还是 JDK 1.5，能用字节码表达的语义范围应该是一致的。当然，肯定不可能简单地把 Class 的文件版本号从 49.0 改回 48.0 就能解决问题了，虽然字节码指令的数量没有变化，但是元数据信息和一些语法支持的内容还是要做相应的修改。以枚举为例，在 JDK 1.5 中增加了 `enum` 关键字，但是 Class 文件常量池的 `CONSTANT_Class_`

① 如果想了解编译器在这个阶段所做的各种动作的详细信息，可以参考第 10 章的“解语法糖”部分。

② JSR-133：Java Memory Model and Thread Specification Revision（Java 内存模型和线程规范修订）。

info 类型常量并没有发生任何语义变化，仍然是代表一个类或接口的符号引用，没有加入枚举，也没有增加过“CONSTANT_Enum_info”之类的“枚举符号引用”常量。所以使用 enum 关键字定义常量，虽然从 Java 语法上看起来与使用 class 关键字定义类、使用 interface 关键字定义接口是同一层次的，但实际上这是由 Javac 编译器做出来的假象，从字节码的角度来看，枚举仅仅是一个继承于 java.lang.Enum、自动生成了 values() 和 valueOf() 方法的普通 Java 类而已。

Retrotranslator 对枚举所做的主要处理就是把枚举类的父类从“java.lang.Enum”替换为它运行时类库中包含的“net.sf.retrotranslator.runtime.java.lang.Enum_”，然后再在类和字段的访问标志中抹去 ACC_ENUM 标志位。当然，这只是处理的总体思路，具体的实现要比上面说的复杂得多。可以想象既然两个父类的实现都不一样，values() 和 valueOf() 的方法自然需要重写，常量池需要引入大量新的来自父类的符号引用，这些都是实现细节。图 9-3 是一个使用 JDK 1.5 编译的枚举类与被 Retrotranslator 转换处理后的字节码的对比图。

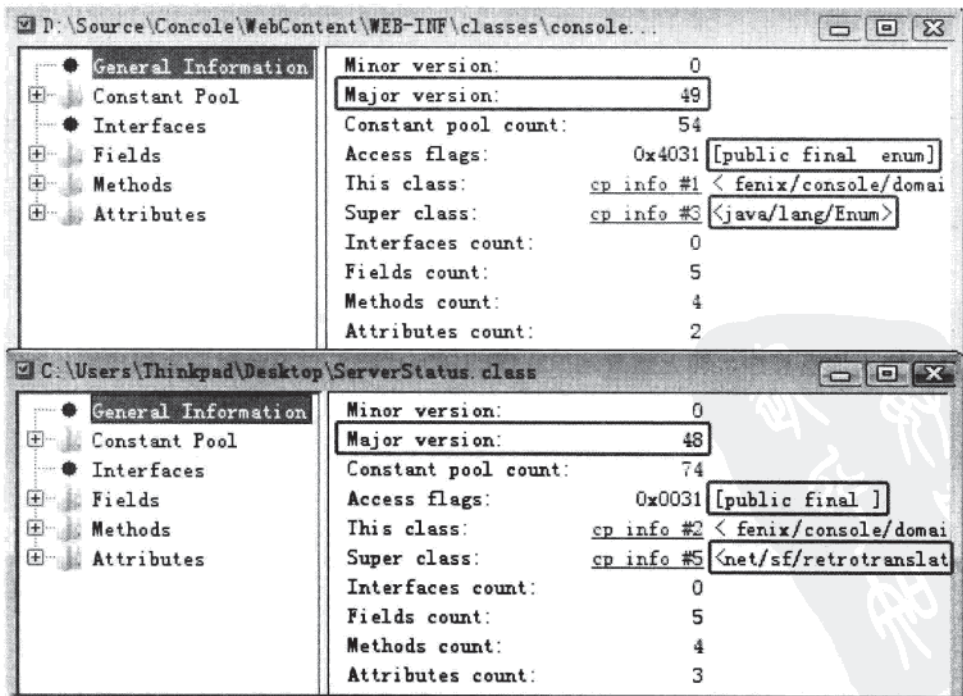


图 9-3 Retrotranslator 处理前后的枚举类字节码对比

9.3 实战：自己动手实现远程执行功能

不知道读者在做程序维护的时候是否遇到过这类情形：排查问题的过程中，想查看内存中的一些参数值，却又没有方法把这些值输出到界面或日志中。又或者定位到某个缓存数据有问题，但缺少缓存的统一管理界面，不得不重启服务才能清理掉这个缓存。类似的需求有一个共同的特点，就是只要在服务中执行一段程序代码，就可以定位或排除问题，但就是偏偏找不到可以让服务器执行临时代码的途径，这时候多希望 Java 服务器中也有提供类似 Groovy Console 的功能啊。

JDK 1.6 之后提供了 Compiler API，可以动态地编译 Java 程序，这样虽然达不到动态语言的灵活度，但让服务器执行临时代码的需求是可以得到解决了。在 JDK 1.6 之前，也可以通过“曲线救国”的方式来做到，譬如写一个 JSP 文件上传到服务器，然后在浏览器中运行它，或者在服务端的程序中加入一个 BeanShell Script、JavaScript 等的执行引擎（如 Mozilla Rhino^①）去执行动态脚本。在本章的实战部分，我们将使用前面学到的关于类加载及虚拟机执行子系统的知识去完成在服务端执行临时代码的功能。

9.3.1 目标

在实现“在服务端执行临时代码”这个需求之前，先来明确一下本次实战的具体目标，我们希望最终的产品是这样的：

- 不依赖 JDK 版本，能在目前还被普遍使用的 JDK 中部署，也就是使用 JDK 1.4 至 JDK 1.7 都可以运行。
- 不改变原有服务端程序的部署，不依赖任何第三方类库。
- 不侵入原有程序，即无须改动原程序的任何代码。也不会对原有程序的运行带来任何影响。
- 考虑到 BeanShell Script 或 JavaScript 等脚本编写起来不太方便，“临时代码”需要直接支持 Java 语言。
- “临时代码”应当具备足够的自由度，不需要依赖特定的类或实现特定的接口。这里写的是“不需要”而不是“不可以”，当“临时代码”需要引用其他类库时也没有限制，只要服务端程序能使用的，临时代码应当都能直接引用。

^① Rhino 站点：<http://www.mozilla.org/rhino/>，Rhino 已被收编入 JDK 1.6 中。

□“临时代码”的执行结果能返回到客户端，执行结果可以包括程序中输出的信息及抛出的异常等。

看完上面列出的目标，你觉得完成这个需求需要做多少工作呢？也许答案比大多数人所想的都要简单一些：5个类，250行代码（含注释），大约一个半小时左右的开发时间就可以了，现在就开始编写程序吧！

9.3.2 思路

在程序实现的过程中，我们需要解决以下三个问题：

- 如何编译提交到服务器的 Java 代码？
- 如何执行编译之后的 Java 代码？
- 如何收集 Java 代码的执行结果？

对于第一个问题，我们有两个方案可以选择，一个是使用 tools.jar 包（在 Sun JDK/lib 目录下）中的 com.sun.tools.javac.Main 类来编译 Java 文件，这其实和使用 Javac 命令来编译是一样的。这种思路的缺点是引入了额外的 JAR 包，而且把程序绑死在 Sun 的 JDK 上了，要部署到其他公司的 JDK 中还得把 tools.jar 带上（虽然 JRockit 和 J9 虚拟机也有这个 JAR 包，但它总不是标准所规定必须存在的）。另外一种思路是直接在客户端编译好，把字节码而不是 Java 代码传到服务端，这听起来好像有点投机取巧，一般来说确实不应该假定客户端一定具有编译代码的能力，但是既然程序员会写 Java 代码去给服务端排查问题，那么很难想象他的机器上会连编译 Java 程序的环境都没有。

对于第二个问题，简单地一想：要执行编译后的 Java 代码，让类加载器加载这个类生成一个 Class 对象，然后反射调用一下某个方法就可以了（因为不实现任何接口，我们可以借用一下 Java 中人人皆知的“main()”方法）。但我们还应该考虑得更周全些：一段程序往往不是编写和运行一次就能达到效果，同一个类可能要被反复地修改、提交和执行。另外，提交上去的类要能访问到服务端的其他类库才行。还有就是，既然提交的是临时代码，那提交的 Java 类在执行完后就应当能被卸载和回收掉。

最后一个问题，我们想把程序往标准输出（System.out）和标准错误输出（System.err）中打印的信息收集起来。但标准输出设备是整个虚拟机进程级别的资源，如果使用 System.setOut() / System.setErr() 方法把输出流重定向到自己定义的 PrintStream 对象上固然可以收集到输出信息，但也会对原有程序产生影响：会把其他进程向标准输

出中打印的信息也收集了。虽然这些并不是不能解决的问题，不过为了达到完全不影响原程序的目的，我们可以采用另外一种办法：直接在执行的类中把对 System.out 的符号引用替换为我们准备的 PrintStream 的符号引用，依赖前面学习到的知识，做到这一点并不困难。

9.3.3 实现

在程序实现部分，我们主要来看看代码及其中的注释。首先看看实现过程中需要用到的四个支持类。第一个类用于实现同一个类的代码可以被多次加载，即用于解决上一节列举的第 2 个问题的 HotSwapClassLoader，具体程序如下面的代码清单 9-3 所示。

HotSwapClassLoader 所做的事情仅仅是公开父类（即 java.lang.ClassLoader）中的 protected 方法 defineClass()，我们将会使用这个方法把提交执行的 Java 类的 byte[] 数组转变为 Class 对象。HotSwapClassLoader 中并没有重写 loadClass() 或 findClass() 方法，因此如果不算外部手工调用 loadByte() 方法的话，这个类加载器的类查找范围与它的父类加载器是完全一致的，在被虚拟机调用时，它会按照双亲委派模型交给父类加载。构造函数中指定为加载 HotSwapClassLoader 类的类加载器作为父类加载器，这一步是实现提交的执行代码可以访问服务端引用类库的关键，下面我们来看看代码清单 9-3。

代码清单 9-3 HotSwapClassLoader 的实现

```
/**
 * 为了多次载入执行类而加入的加载器 <br>
 * 把 defineClass 方法开放出来，只有外部显式调用的时候才会使用到 loadByte 方法
 * 由虚拟机调用时，仍然按照原有的双亲委派规则使用 loadClass 方法进行类加载
 * @author zzm
 */
public class HotSwapClassLoader extends ClassLoader {

    public HotSwapClassLoader() {
        super(HotSwapClassLoader.class.getClassLoader());
    }

    public Class loadByte(byte[] classByte) {
        return defineClass(null, classByte, 0, classByte.length);
    }
}
```

```
}  
  
}
```

第二个类是实现将 `java.lang.System` 替换为我们自己定义的 `HackSystem` 类的过程，它直接修改符合 Class 文件格式的 `byte[]` 数组中的常量池部分，将常量池中指定内容的 `CONSTANT_Utf8_info` 常量替换为新的字符串，具体代码如下面的代码清单 9-4 所示。ClassModifier 中涉及对 `byte[]` 数组操作的部分，主要是将 `byte[]` 与 `int` 和 `String` 互相转换，以及把对 `byte[]` 数据的替换操作封装在代码清单 9-5 所示的 `ByteUtils` 中。

经过 `ClassModifier` 处理后的 `byte[]` 数组才会传给 `HotSwapClassLoader.loadByte()` 方法进行类加载，`byte[]` 数组在这里替换符号引用之后，与客户端直接在 Java 代码中引用 `HackSystem` 类再编译生成的 Class 是完全一样的。这样的实现既避免了客户端编写临时执行代码时要依赖特定的类（不然无法引入 `HackSystem`），又避免了服务端修改标准输出后影响到其他程序的输出。下面我们来看看代码清单 9-4 和代码清单 9-5：

代码清单 9-4 ClassModifier 的实现

```
/**  
 * 修改 Class 文件，暂时只提供修改常量池常量的功能  
 * @author zzm  
 */  
public class ClassModifier {  
  
    /**  
     * Class 文件中常量池的起始偏移  
     */  
    private static final int CONSTANT_POOL_COUNT_INDEX = 8;  
  
    /**  
     * CONSTANT_Utf8_info 常量的 tag 标志  
     */  
    private static final int CONSTANT_Utf8_info = 1;  
  
    /**  
     * 常量池中 11 种常量所占的长度，CONSTANT_Utf8_info 型常量除外，因为它不是定长的  
     */  
    private static final int[] CONSTANT_ITEM_LENGTH = { -1, -1, 5, -1, 5, 9, 9, 3,
```


250 ❖ 第三部分 虚拟机执行子系统

```
3, 5, 5, 5, 5 };
```

```
private static final int u1 = 1;
private static final int u2 = 2;

private byte[] classByte;

public ClassModifier(byte[] classByte) {
    this.classByte = classByte;
}

/**
 * 修改常量池中 CONSTANT_Utf8_info 常量的内容
 * @param oldStr 修改前的字符串
 * @param newStr 修改后的字符串
 * @return 修改结果
 */
public byte[] modifyUTF8Constant(String oldStr, String newStr) {
    int cpc = getConstantPoolCount();
    int offset = CONSTANT_POOL_COUNT_INDEX + u2;
    for (int i = 0; i < cpc; i++) {
        int tag = ByteUtils.bytes2Int(classByte, offset, u1);
        if (tag == CONSTANT_Utf8_info) {
            int len = ByteUtils.bytes2Int(classByte, offset + u1, u2);
            offset += (u1 + u2);
            String str = ByteUtils.bytes2String(classByte, offset, len);
            if (str.equalsIgnoreCase(oldStr)) {
                byte[] strBytes = ByteUtils.string2Bytes(newStr);
                byte[] strLen = ByteUtils.int2Bytes(newStr.length(), u2);
                classByte = ByteUtils.bytesReplace(classByte, offset - u2,
                    u2, strLen);
                classByte = ByteUtils.bytesReplace(classByte, offset, len,
                    strBytes);
                return classByte;
            } else {
                offset += len;
            }
        } else {
            offset += CONSTANT_ITEM_LENGTH[tag];
        }
    }
    return classByte;
}
```

```
}

/**
 * 获取常量池中常量的数量
 * @return 常量池数量
 */
public int getConstantPoolCount() {
    return ByteUtils.bytes2Int(classByte, CONSTANT_POOL_COUNT_INDEX, u2);
}
}
```


代码清单 9-5 ByteUtils 的实现

```
/**
 * Bytes 数组处理工具
 * @author
 */
public class ByteUtils {

    public static int bytes2Int(byte[] b, int start, int len) {
        int sum = 0;
        int end = start + len;
        for (int i = start; i < end; i++) {
            int n = ((int) b[i]) & 0xff;
            n <<= (--len) * 8;
            sum = n + sum;
        }
        return sum;
    }

    public static byte[] int2Bytes(int value, int len) {
        byte[] b = new byte[len];
        for (int i = 0; i < len; i++) {
            b[len - i - 1] = (byte) ((value >> 8 * i) & 0xff);
        }
        return b;
    }

    public static String bytes2String(byte[] b, int start, int len) {
        return new String(b, start, len);
    }
}
```



252 ❖ 第三部分 虚拟机执行子系统

```

public static byte[] string2Bytes(String str) {
    return str.getBytes();
}

public static byte[] bytesReplace(byte[] originalBytes, int offset, int len,
byte[] replaceBytes) {
    byte[] newBytes = new byte[originalBytes.length + (replaceBytes.length -
len)];
    System.arraycopy(originalBytes, 0, newBytes, 0, offset);
    System.arraycopy(replaceBytes, 0, newBytes, offset, replaceBytes.length);
    System.arraycopy(originalBytes, offset + len, newBytes, offset +
replaceBytes.length, originalBytes.length - offset - len);
    return newBytes;
}
}
}

```

最后一个类就是前面提到过的用来代替 `java.lang.System` 的 `HackSystem`，这个类中的方法看起来不少，但其实除了把 `out` 和 `err` 两个静态变量改成使用 `ByteArrayOutputStream` 作为打印目标的同一个 `PrintStream` 对象，以及增加了读取、清理 `ByteArrayOutputStream` 中内容的 `getBufferString()` 和 `clearBuffer()` 方法外，就再没有其他新鲜的内容了。其余的方法全部都来自于 `System` 类的 `public` 方法，方法名字、参数和返回值都完全一样，并且实现也是直接转调了 `System` 类的对应方法而已。保留这些方法的目的，是为了在 `System` 被替换成 `HackSystem` 之后，执行代码中调用的 `System` 的其余方法仍然可以继续使用，`HackSystem` 的实现如代码清单 9-6 所示。

代码清单 9-6 `HackSystem` 的实现

```

/**
 * 为 JavaClass 劫持 java.lang.System 提供支持
 * 除了 out 和 err 外，其余的都直接转发给 System 处理
 * @author zzm
 */
public class HackSystem {

    public final static InputStream in = System.in;

    private static ByteArrayOutputStream buffer = new ByteArrayOutputStream();

```

```
public final static PrintStream out = new PrintStream(buffer);

public final static PrintStream err = out;

public static String getBufferString() {
    return buffer.toString();
}

public static void clearBuffer() {
    buffer.reset();
}

public static void setSecurityManager(final SecurityManager s) {
    System.setSecurityManager(s);
}

public static SecurityManager getSecurityManager() {
    return System.getSecurityManager();
}

public static long currentTimeMillis() {
    return System.currentTimeMillis();
}

public static void arraycopy(Object src, int srcPos, Object dest, int destPos,
int length) {
    System.arraycopy(src, srcPos, dest, destPos, length);
}

public static int identityHashCode(Object x) {
    return System.identityHashCode(x);
}

// 下面所有的方法都与 java.lang.System 的名称一样
// 实现都是字节转调 System 的对应方法
// 因版面原因, 省略了其他方法
}
```

四个支持类已经讲解完毕, 我们来看看最后一个类 `JavaClassExecuter`, 它是为外部调用提供的入口, 调用前面几个支持类组装逻辑, 完成类加载工作。 `JavaClassExecuter` 只有一个 `execute()` 方法, 用输入的符合 Class 文件格式的 `byte[]` 数组替换掉 `java.lang.`

254 ❖ 第三部分 虚拟机执行子系统

System 的符号引用后，使用 HotSwapClassLoader 加载生成一个 Class 对象，由于每次执行 execute() 方法都会生成一个新的类加载器实例，因此同一个类可以实现重复加载。然后反射调用这个 Class 对象的 main() 方法，如果期间出现任何异常，将异常信息打印到 HackSystem.out 中，最后把缓冲区中的信息作为方法的结果来返回。JavaClassExecuter 的实现代码如代码清单 9-7 所示。

代码清单 9-7 JavaClassExecuter 的实现

```

/**
 * JavaClass 执行工具
 * @author zzm
 */
public class JavaClassExecuter {

    /**
     * 执行外部传过来的代表一个 Java 类的 Byte 数组 <br>
     * 将输入类的 byte 数组中代表 java.lang.System 的 CONSTANT_utf8_info 常量修改为劫持后的
HackSystem 类
     * 执行方法为该类的 static main(String[] args) 方法，输出结果为该类向 System.out/err
输出的信息
     * @param classByte 代表一个 Java 类的 Byte 数组
     * @return 执行结果
     */
    public static String execute(byte[] classByte) {
        HackSystem.clearBuffer();
        ClassModifier cm = new ClassModifier(classByte);
        byte[] modiBytes = cm.modifyUTF8Constant("java/lang/System", "org/
fenixsoft/classloading/execute/HackSystem");
        HotSwapClassLoader loader = new HotSwapClassLoader();
        Class clazz = loader.loadByte(modiBytes);
        try {
            Method method = clazz.getMethod("main", new Class[] { String[].class
});
            method.invoke(null, new String[] { null });
        } catch (Throwable e) {
            e.printStackTrace(HackSystem.out);
        }
        return HackSystem.getBufferString();
    }
}

```

9.3.4 验证

远程执行功能的编码到此就完成了，接下来就来检验一下我们的劳动成果。只是作为测试的话，任意写一个 Java 类，内容无所谓，只要向 System.out 输出信息即可，取名为 TestClass，放到服务器 C 盘的根目录中。然后建立一个 JSP 文件写上如代码清单 9-8 所示的内容，就可以在浏览器中看到这个类的运行结果了。

代码清单 9-8 测试 JSP

```
<%@ page import="java.lang.*" %>
<%@ page import="java.io.*" %>
<%@ page import="org.fenixsoft.classloading.execute.*" %>
<%
    InputStream is = new FileInputStream("c:/TestClass.class");
    byte[] b = new byte[is.available()];
    is.read(b);
    is.close();

    out.println("<textarea style='width:1000;height=800'>");
    out.println(JavaClassExecuter.execute(b));
    out.println("</textarea>");
%>
```

当然，上面的做法只是用于测试和演示，实际使用这个 JavaExecuter 执行器的时候，如果还要手工复制一个 Class 文件到服务器上就没有什么意义了。笔者给这个执行器写了一个“外壳”，是一个 Eclipse 插件，可以把 Java 文件编译后传输到服务器中，然后把执行器的返回结果输出到 Eclipse 的 Console 窗口里，这样就可以在有灵感的时候随时写几行调试代码，放到测试环境的服务器上立即运行了。实现虽然简单，但效果很不错，对调试问题非常有用，如图 9-4 所示。

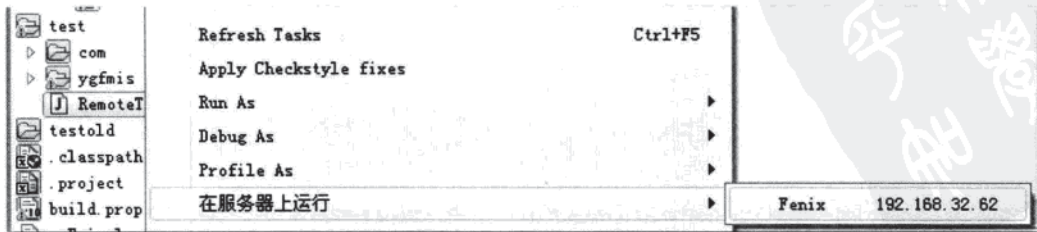


图 9-4 JavaClassExecuter 的使用

9.4 本章小结

本书第 6 ~ 9 章介绍了 Class 文件格式、类加载及虚拟机执行引擎几部分内容，这些内容是虚拟机中必不可少的组成部分，了解了虚拟机如何执行程序，才能更好地理解怎样才能写出优秀的代码。

关于虚拟机执行子系统的介绍到此为止就结束了，通过这 4 章的讲解，我们描绘了一个虚拟机应该如何运行 Class 文件的概念模型。对于具体到某个虚拟机的实现，为了使实现简单清晰，或者为了更快的运行速度，在虚拟机内部的运作跟概念模型可能会有非常大的差异，但从最终的执行结果来看应该是一致的。从下一章开始，我们将探索虚拟机在语法上和运行性能上，是如何对程序编写做出各种优化的。





第四部分 程序编译与代码优化

第 10 章 早期（编译期）优化

第 11 章 晚期（运行期）优化



第 10 章 早期（编译期）优化

本章主要内容

- 概述
- Javac 编译器
- Java 语法糖的味道
- 实战：插入式注解处理器

从计算机程序出现的第一天起，对效率的追逐就是程序天生的坚定信仰，这个过程犹如一场没有终点、永不停歇的 F1 方程式竞赛，程序员是车手，技术平台则是在赛道上飞驰的赛车。

10.1 概述

Java 语言的“编译期”是一段“不确定”的操作过程，因为它可能是指一个前端编译器（其实叫“编译器的前端”更准确一些）把 *.java 文件转变成 *.class 文件的过程；也可能是指虚拟机的后端运行期编译器（JIT 编译器，Just In Time Compiler）把字节码转变成机器码的过程；还可能是指使用静态提前编译器（AOT 编译器，Ahead Of Time Compiler）直接把 *.java 文件编译成本地机器代码的过程。下面列举了这三类编译过程中一些比较有代表性的编译器：

- 前端编译器：Sun 的 Javac、Eclipse JDT 中的增量式编译器（ECJ）^①。
- JIT 编译器：HotSpot VM 的 C1、C2 编译器。
- AOT 编译器：GNU Compiler for the Java（GCJ）^②、Excelsior JET ^③。

① JDT 官方站点：<http://www.eclipse.org/jdt/>。

② GCJ 官方站点：<http://gcc.gnu.org/java/>。

③ Excelsior JET 官方站点：<http://www.excelsior-usa.com/>。

这三类过程中最符合大家对 Java 程序编译认知的应该是第一类，在本章的后续文字里，笔者提到的“编译期”和“编译器”都仅限于第一类编译过程。限制了编译范围后，对于“优化”二字的定义就需要宽松一些，因为 Javac 这类编译器对代码的运行效率几乎没有任何优化措施（在 JDK 1.3 之后，Javac 的 `-O` 优化参数就不再有意义）。虚拟机设计团队把对性能优化集中到了后端的即时编译器中，这样可以让那些不是由 Javac 产生的 Class 文件（如 JRuby、Groovy 等语言的 Class 文件）也同样能享受到编译器优化所带来的好处。但是 Javac 做了许多针对编码过程的优化措施来改善程序员的编码风格和提高编码效率。相当多新生的 Java 语法特性，都是靠编译器的“语法糖”来实现，而不是依赖虚拟机的底层改进来支持，可以说，Java 中即时编译器在运行期的优化过程对于程序运行来说更重要，而前端编译器在编译期的优化过程对于程序编码来说关系更加密切。

10.2 Javac 编译器

分析源码是了解一项技术实现内幕的最有效的手段，Javac 编译器不像 HotSpot 虚拟机那样使用 C++ 语言（包含少量 C 语言）实现，它本身就是一个由 Java 语言编写的程序，这为纯 Java 的程序员了解它的编译过程带来了很大的便利。

10.2.1 Javac 的源码与调试

Javac 的源码存放在 `JDK_SRC_HOME/langtools/src/share/classes/com/sun/tools/javac` 中^①，除了 JDK 自身的 API 外，就只引用了 `JDK_SRC_HOME/langtools/src/share/classes/com/sun/*` 里面的代码，所以调试环境建立起来简单方便，基本上不需要处理依赖关系。

以 Eclipse IDE 环境为例，先建立一个名为“Compiler_javac”的 Java 工程，然后把 `JDK_SRC_HOME/langtools/src/share/classes/com/sun/*` 目录下的源文件全部拷贝到工程的源码目录中，如图 10-1 所示。

导入代码期间，源码文件“AnnotationProxyMaker.java”可能会提示“Access Restriction”，被 Eclipse 拒绝编译，如图 10-2 所示。

① 如何获取 OpenJDK 源码请参考本书第 1 章。

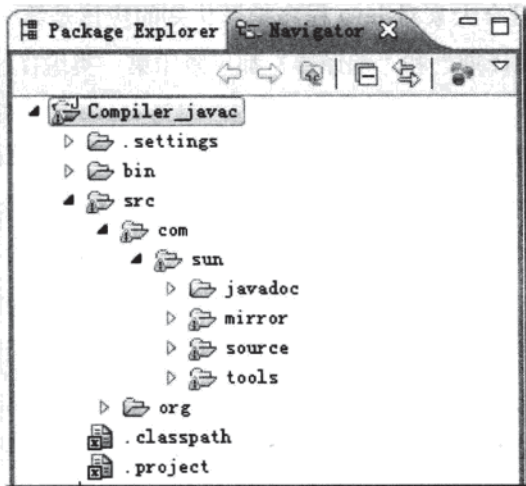


图 10-1 Eclipse 中的 Javac 工程

```

/**
 * Returns a dynamic proxy for an annotation mirror.
 */
private Annotation generateAnnotation() {
    return AnnotationParser.annotationForMap(annoType,
}

```

Access restriction: The method annotationForMap(Class, Map<String, Object>) in AnnotationParser is not accessible due to restriction on required library D:_DevSpace\jdk1.6.0_21\jre\lib\rt.jar

图 10-2 AnnotationProxyMaker 被拒绝编译

这是由于 Eclipse 的 JRE System Library 中默认包含了一系列的代码访问规则 (Access Rules)，如果代码中引用了这些访问规则所禁止引用的类，就会提示这个错误。可以通过添加一条允许访问 Jar 包中所有类的访问规则来解决这个问题，如图 10-3 所示。

导入了 Javac 的源码后，就可以运行 com.sun.tools.javac.Main 的 main() 方法来执行编译了，与命令行中使用 Javac 的命令没有什么区别，编译的文件与参数在 Eclipse 的“Debug Configurations”面板中的“Arguments”页签中指定。

虚拟机规范严格定义了 Class 文件的格式，但是对如何把 Java 源码文件转变为 Class 文件的编译过程未作任何定义，所以这部分内容是与具体 JDK 实现相关的。从

Sun Javac 的代码来看，编译过程大致可以分为三个过程，分别是：

- 解析与填充符号表过程。
- 插入式注解处理器的注解处理过程。
- 分析与字节码生成过程。

这三个步骤之间的关系与交互顺序如图 10-4 所示。

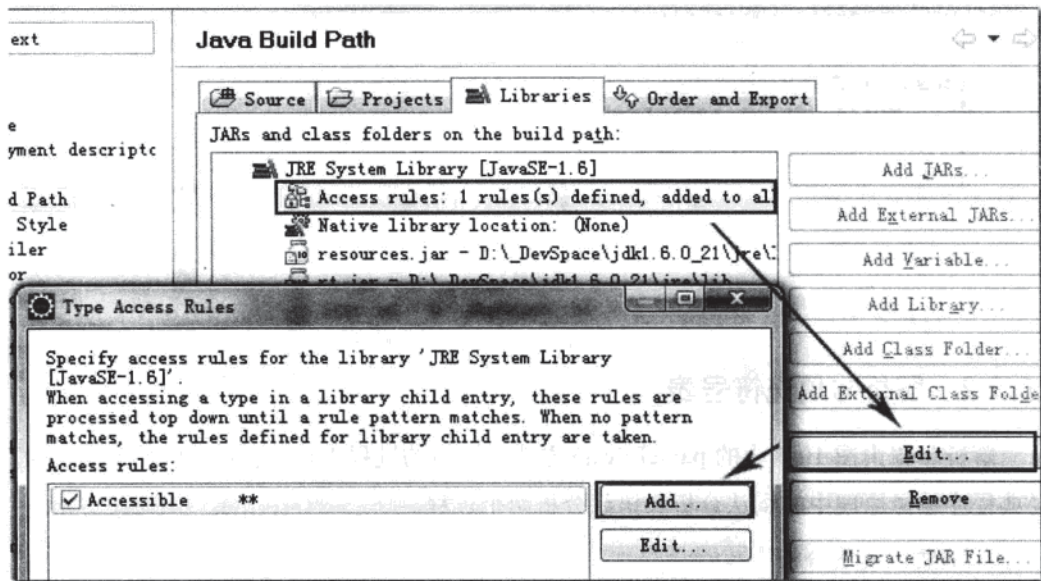


图 10-3 设置访问规则



图 10-4 Javac 的编译过程^①

Javac 编译动作的入口是 `com.sun.tools.javac.main.JavaCompiler` 类，上述三个过程的代码逻辑集中在这个类的 `compile()` 和 `compile2()` 方法里，其中主体代码如图 10-5 所示，整个编译最关键的处理就由图中标注的 8 个方法来完成，下面我们具体看一下这 8 个方法实现了什么功能。

^① 图片来源：<http://openjdk.java.net/groups/compiler/doc/compilation-overview/index.html>，本书对该图做了汉化处理。

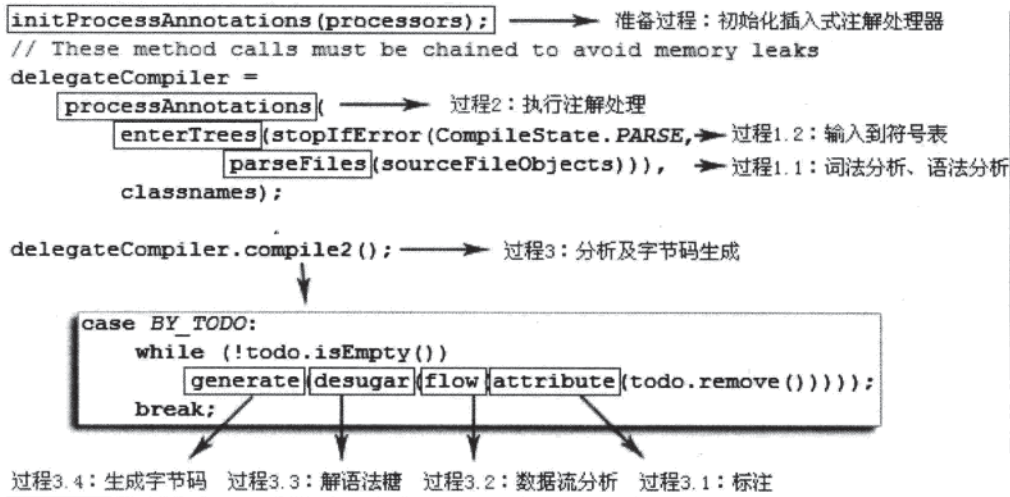


图 10-5 Javac 编译过程的主体代码

10.2.2 解析与填充符号表

解析步骤由图 10-5 中的 `parseFiles()` 方法（图中的过程 1.1）完成，解析步骤包括了经典程序编译原理中的词法分析和语法分析两个过程。

1. 词法、语法分析

词法分析是将源代码的字符流转变为标记（Token）集合，单个字符是程序编写过程的最小元素，而标记则是编译过程的最小元素，关键字、变量名、字面量和运算符都可以成为标记，如“`int a = b+2`”这句代码包含了 6 个标记，分别是 `int`、`a`、`=`、`b`、`+`、`2`，虽然关键字 `int` 由三个字符构成，但是它只是一个 Token，不可再拆分。在 Javac 的源码中，词法分析过程由 `com.sun.tools.javac.parser.Scanner` 类来实现。

语法分析是根据 Token 序列来构造抽象语法树的过程，抽象语法树（AST，Abstract Syntax Tree）是一种用来描述程序代码语法结构的树形表示方式，语法树的每一个节点都代表着程序代码中的一个语法结构（Construct），例如包、类型、修饰符、运算符、接口、返回值甚至连代码注释等都可以是一个语法结构。

图 10-6 是 Eclipse AST View 插件分析出来的某段代码的抽象语法树视图，读者可以通过这张图对抽象语法树有一个直观的认识。在 Javac 的源码中，语法分析过程由

com.sun.tools.javac.parser.Parser 类来实现，这个阶段产出的抽象语法树由 com.sun.tools.javac.tree.JCTree 类来表示，经过这个步骤之后，编译器就基本不会再对源码文件进行操作了，后续的操作都建立在抽象语法树之上。

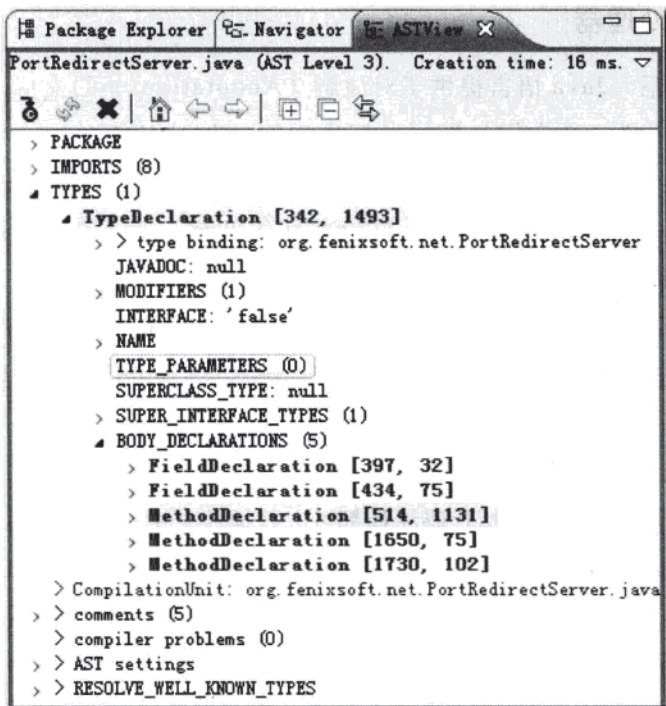


图 10-6 抽象语法树结构视图

2. 填充符号表

完成了语法分析和词法分析之后，下一步就是填充符号表的过程，也就是图 10-5 中 enterTrees() 方法（图中的过程 1.2）所做的事情。符号表（Symbol Table）是由一组符号地址和符号信息构成的表格，读者可以把它想象成哈希表中 K-V 值对的形式（实际上符号表不一定是哈希表实现，可以是有序符号表、树状符号表和栈结构符号表等）。符号表中所登记的信息在编译的不同阶段都要用到。在语义分析中，符号表所登记的内容将用于语义检查（如检查一个名字的使用和原先的说明是否一致）和产生中间代码。在目标代码生成阶段，当对符号名进行地址分配时，符号表是地址分配的依据。

在 Javac 源代码中，填充符号表的过程由 `com.sun.tools.javac.comp.Enter` 类实现，此过程的出口是一个待处理列表（To Do List），包含了每一个编译单元的抽象语法树的顶级节点，以及 `package-info.java`（如果存在的话）的顶级节点。

10.2.3 注解处理器

JDK 1.5 之后，Java 语言提供了对注解（Annotations）的支持，这些注解与普通的 Java 代码一样，是在运行期间发挥作用的。在 JDK 1.6 中实现了 JSR-269 规范^①，提供了一组插入式注解处理器的标准 API 在编译期间对注解进行处理，我们可以把它看做是一组编译器的插件，在这些插件里面，可以读取、修改、添加抽象语法树中的任意元素。如果这些插件在处理注解期间对语法树进行了修改，那么编译器将回到解析及填充符号表的过程重新处理，直到所有的插入式注解处理器都没有再对语法树进行修改为止，每一次循环称为一个 Round，也就是上文中图 10-4 的那个回环过程。

有了编译器注解处理的标准 API 后，我们的代码才有可能干涉编译器的行为，由于语法树中的任意元素，甚至包括代码注释都可以在插件之中访问到，所以通过插入式注解处理器实现的插件在功能上有很大的发挥空间。只要有足够的创意，程序员可以使用插入式注解处理器来实现许多原本只能在编码中完成的事情，本章最后有一个使用插入式注解处理器的简单实战。

在 Javac 源码中，插入式注解处理器的初始化过程是在 `initProcessAnnotations()` 方法中完成的，而它的执行过程则是在 `processAnnotations()` 方法中完成的，这个方法判断是否还有新的注解处理器需要执行，如果有的话，则通过 `com.sun.tools.javac.processing.JavacProcessingEnvironment` 类的 `doProcessing()` 方法生成一个新的 `JavaCompiler` 对象对编译的后续步骤进行处理。

10.2.4 语义分析与字节码生成

语法分析之后，编译器获得了程序代码的抽象语法树表示，语法树能表示一个结构正确的源程序的抽象，但无法保证源程序是符合逻辑的。而语义分析的主要任务是对结

^① JSR-269 : Pluggable Annotations Processing API (插入式注解处理 API)。

构上正确的源程序进行上下文有关性质的审查，如进行类型审查。举个例子，假设有如下的三个变量定义语句：

```
int a = 1;
boolean b = false;
char c = 2;
```

后续可能出现的赋值运算如下：

```
int d = a + c;
int d = b + c;
char d = a + c;
```

后续代码中如果出现了如上三种赋值运算的话，那它们都能构成结构正确的语法树，但是只有第一种写法在语义上是没有问题的，能够通过编译，其余两种在 Java 语言中是不合逻辑的，无法编译（是否合乎语义逻辑必须限定在具体的语言与具体的上下文环境之中才有意义。如在 C 语言中，a、b、c 的上下文定义不变，第二、三种写法都是可以正确编译的）。

1. 标注检查

Javac 的编译过程中，语义分析过程分为标注检查和数据及控制流分析两个步骤，分别由图 10-5 的 `attribute()` 和 `flow()` 方法（分别对应图中的过程 3.1 和过程 3.2）完成。

标注检查步骤检查的内容包括诸如变量使用前是否已被声明、变量与赋值之间的数据类型是否能够匹配，等等。在标注检查步骤中，还有一个重要的动作称为常量折叠，如果我们在代码中写了如下定义：

```
int a = 1 + 2;
```

在语法树上仍然能看到字面量“1”、“2”和操作符“+”号，但是在经过常量折叠之后，它们将会被折叠为字面量“3”，如图 10-7 所示，这个插入式表达式（Infix Expression）的值已经在语法树上标注出来了（`ConstantExpressionValue : 3`）。由于编译期间进行了常量折叠，所以在代码里面定义“`a=1+2`”比起直接定义“`a=3`”，并不会增加程序运行期哪怕仅仅一个 CPU 指令的运算量。

标注检查步骤在 Javac 源码中的实现类是 `com.sun.tools.javac.comp.Attr` 类和 `com.sun.tools.javac.comp.Check` 类。


```

    ▲ INITIALIZER
      ▲ InfixExpression [105, 5]
        ▲ > (Expression) type binding: int
          NAME: 'int'
          KEY: 'I'
          IS RECOVERED: false
          QUALIFIED NAME: 'int'
          KIND: isPrimitive
        > CREATE ARRAY TYPE (+1): int[]
          BINARY NAME: 'I'
          ANNOTATIONS (0)
          >java element: null
          Boxing: false; Unboxing: false
          ConstantExpressionValue: 3
        ▲ LEFT_OPERAND
          > NumberLiteral [105, 1]
          OPERATOR: '+'
        ▲ RIGHT_OPERAND
          > NumberLiteral [109, 1]

```

图 10-7 常量折叠

2. 数据及控制流分析

数据及控制流分析是对程序上下文逻辑更进一步的验证，它可以检查出诸如程序局部变量在使用前是否有赋值、方法的每条路径是否都有返回值、是否所有的受查异常都被正确处理了等问题。编译时期的数据及控制流分析与类加载时的数据及控制流分析的目的基本上是一致的，但校验范围有所区别，有一些校验项只有在编译期或运行期才能进行。下面举一个关于 `final` 修饰符的数据及控制流分析的例子，见代码清单 10-1 所示。

代码清单 10-1 `final` 语义校验

```

// 方法一带有 final 修饰
public void foo(final int arg) {
    final int var = 0;
    // do something
}

// 方法二没有 final 修饰
public void foo(int arg) {
    int var = 0;
    // do something
}

```

这两个 `foo()` 方法中，一个方法的参数和局部变量定义使用了 `final` 修饰符，另外一

个则没有，在代码编写时程序肯定会受到 `final` 修饰符的影响，不能再改变 `arg` 和 `var` 变量的值，但是这两段代码编译出来的 Class 文件是没有任何一点区别的，通过第 6 章的讲解我们已经知道，局部变量与字段（实例变量、类变量）是有区别的，它在常量池中并没有 `CONSTANT_Fieldref_info` 的符号引用，自然就没有访问标志（`Access_Flags`）的信息，甚至可能连名称都不会被保留下来（取决于编译时的选项），自然在 Class 文件中不可能知道一个局部变量是不是被声明为 `final` 了。因此，将局部变量声明为 `final`，对运行期是没有影响的，变量的不变性仅仅由编译器在编译期间保障。在 `Javac` 的源码中，数据及控制流分析的入口是图 10-5 中的 `flow()` 方法（图中的过程 3.2），具体操作由 `com.sun.tools.javac.comp.Flow` 类来完成。

3. 解语法糖

语法糖（Syntactic Sugar），也称糖衣语法，是由英国计算机科学家彼得·约翰·兰达（Peter J. Landin）发明的一个术语，指在计算机语言中添加的某种语法，这种语法对语言的功能并没有影响，但是更方便程序员使用。通常来说使用语法糖能够增加程序的可读性，从而减少程序代码出错的机会。

Java 在现代编程语言之中属于“低糖语言”（相对于 C# 及许多其他 JVM 语言来说），尤其是 JDK 1.5 之前的版本，“低糖”语法也是 Java 语言被怀疑已经“落后”的一个表面理由。Java 中最常用的语法糖主要是前面提到过的泛型（泛型并不一定都是语法糖实现，如 C# 的泛型就是直接由 CLR 支持的）、变长参数、自动装箱拆箱，等等，虚拟机运行时不支持这些语法，它们在编译阶段被还原回简单的基础语法结构，这个过程就称为解语法糖。Java 的这些语法糖被解除后是什么样子，将在下一节中详细讲述。

在 `Javac` 的源码中，解语法糖的过程由 `desugar()` 方法触发，在 `com.sun.tools.javac.comp.TransTypes` 类和 `com.sun.tools.javac.comp.Lower` 类中完成。

4. 字节码生成

字节码生成是 `Javac` 编译过程的最后一个阶段，在 `Javac` 源码里面由 `com.sun.tools.javac.jvm.Gen` 类来完成。字节码生成阶段不仅仅是把前面各个步骤所生成的信息（语法树、符号表）转化成字节码写到磁盘中，编译器还进行了少量的代码添加和转换工作。

例如前面章节中多次提到的实例构造器 `<init>()` 方法和类构造器 `<clinit>()` 方法就是在这个阶段被添加到语法树之中的（请注意这里的实例构造器并不是指默认构造函数，

如果用户代码中没有提供任何构造函数，那编译器将会添加一个没有参数的、访问性（`public`、`protected` 或 `private`）与当前类一致的默认构造函数，这个工作在填充符号表阶段就已经完成），这两个构造器的产生过程实际上是一个代码收敛的过程，编译器会把语句块（对于实例构造器而言是“{}”块，对于类构造器而言是“`static{}`”块）、变量初始化（实例变量和类变量）、调用父类的实例构造器（仅仅是实例构造器，`<clinit>()`方法中无须调用父类的 `<clinit>()` 方法，虚拟机会自动保证父类构造器的执行，但在 `<clinit>()` 方法中经常会生成调用 `java.lang.Object` 的 `<init>()` 方法的代码）等操作收敛到 `<init>()` 和 `<clinit>()` 方法之中，并且保证一定是按先执行父类的实例构造器，然后初始化变量，最后执行语句块的顺序进行，上面所述的动作由 `Gen.normalizeDefs()` 方法来实现。除了生成构造器以外，还有其他的一些代码替换工作作用于优化程序的实现逻辑，如把字符串的加操作替换为 `StringBuffer` 或 `StringBuilder`（取决于目标代码的版本是否大于或等于 JDK 1.5）的 `append()` 操作，等等。

完成了对语法树的遍历和调整之后，就会把填充了所有所需信息的符号表交到 `com.sun.tools.javac.jvm.ClassWriter` 类手上，由这个类的 `writeClass()` 方法输出字节码，生成最终的 Class 文件，到此为止整个编译过程宣告结束。

10.3 Java 语法糖的味道

几乎各种语言或多或少都提供过一些语法糖来方便程序员的代码开发，这些语法糖虽然不会提供实质性的功能改进，但是它们或能提高效率，或能提升语法的严谨性，或能减少编码出错的机会。不过也有一种观点认为语法糖并不一定都是有益的，大量添加和使用含糖的语法容易让程序员产生依赖，无法看清语法糖的糖衣背后程序代码的真实面目。

总而言之，语法糖可以看做是编译器实现的一些“小把戏”，这些“小把戏”可能会使得效率有一个“大提升”，但我们也应该去了解这些“小把戏”背后的真实世界，那样才能利用好它们，而不是被它们所迷惑。

10.3.1 泛型与类型擦除

泛型是 JDK 1.5 的一项新特性，它的本质是参数化类型（Parameterized Type）的应用，也就是说所操作的数据类型被指定为一个参数。这种参数类型可以用在类、接口和

方法的创建中，分别称为泛型类、泛型接口和泛型方法。

泛型思想早在 C++ 语言的模板（Templates）中就开始生根发芽，在 Java 语言还没有出现泛型时，只能通过 Object 是所有类型的父类和类型强制转换两个特点的配合来实现类型泛化。例如在哈希表的存取中，JDK 1.5 之前使用 HashMap 的 get() 方法，返回值就是一个 Object 对象，由于 Java 语言里面所有的类型都继承于 java.lang.Object，那 Object 转型成任何对象都是有可能的。但是也因为有无限的可能性，就只有程序员和运行期的虚拟机才知道这个 Object 到底是个什么类型的对象。在编译期间，编译器无法检查这个 Object 的强制转型是否成功，如果仅仅依赖程序员去保障这项操作的正确性，许多 ClassCastException 的风险就会被转嫁到程序运行期中。

泛型技术在 C# 和 Java 之中的使用方式看似相同，但实现上却有着根本性的分歧，C# 里面泛型无论在程序源码中、编译后的 IL 中（Intermediate Language，中间语言，这时候泛型是一个占位符）还是运行期的 CLR 中都是切实存在的，List<int> 与 List<String> 就是两个不同的类型，它们在系统运行期生成，有自己的虚方法表和类型数据，这种实现称为类型膨胀，基于这种方法实现的泛型被称为真实泛型。

Java 语言中的泛型则不一样，它只在程序源码中存在，在编译后的字节码文件中，就已经被替换为原来的原生类型（Raw Type，也称为裸类型）了，并且在相应的地方插入了强制转型代码，因此对于运行期的 Java 语言来说，ArrayList<int> 与 ArrayList<String> 就是同一个类。所以说泛型技术实际上是 Java 语言的一颗语法糖，Java 语言中的泛型实现方法称为类型擦除，基于这种方法实现的泛型被称为伪泛型。

代码清单 10-2 是一段简单的 Java 泛型例子，我们可以看一下它编译后的结果是怎样的？

代码清单 10-2 泛型擦除前的例子

```
public static void main(String[] args) {  
    Map<String, String> map = new HashMap<String, String>();  
    map.put("hello", "你好");  
    map.put("how are you?", "吃了没?");  
    System.out.println(map.get("hello"));  
    System.out.println(map.get("how are you?"));  
}
```


把这段 Java 代码编译成 Class 文件，然后再用字节码反编译工具进行反编译后，将会发现泛型都不见了，程序又变回了 Java 泛型出现之前的写法，泛型类型都变回了原生类型，如代码清单 10-3 所示。

代码清单 10-3 泛型擦除后的例子

```
public static void main(String[] args) {  
    Map map = new HashMap();  
    map.put("hello", "你好");  
    map.put("how are you?", "吃了没?");  
    System.out.println((String) map.get("hello"));  
    System.out.println((String) map.get("how are you?"));  
}
```

当初 JDK 设计团队为什么选择类型擦除的方式来实现 Java 语言的泛型支持呢？是因为实现简单、兼容性考虑还是别的原因？我们已不得而知，但确实有不少人对 Java 语言提供的伪泛型颇有微词，当时甚至连《Thinking In Java》一书的作者 Bruce Eckel 也发表了一篇文章《这不是泛型！》^①来批评 JDK 1.5 中的泛型实现。

当时众多的批评之中，有一些是比较表面的，还有一些从性能上说泛型会由于强制转型操作和运行期缺少针对类型的优化等原因从而导致比 C# 的泛型慢一些，则是完全偏离了方向，姑且不论 Java 泛型是不是真的会比 C# 泛型慢，选择从性能的角度上评价用于提升语义准确性的泛型思想，就犹如在讨论刘翔打斯诺克的水平与丁俊晖有多大的差距一般。但笔者也并非在为 Java 的泛型辩护，它在某些场景下确实存在不足，笔者认为通过擦除法来实现泛型丧失了泛型思想一些应有的优雅，例如下面代码清单 10-4 的例子。

代码清单 10-4 当泛型遇见重载 1

```
public class GenericTypes {  
  
    public static void method(List<String> list) {  
        System.out.println("invoke method(List<String> list)");  
    }  
  
    public static void method(List<Integer> list) {  
        System.out.println("invoke method(List<Integer> list)");  
    }  
}
```

① 原文：<http://www.anyang-window.com.cn/quotthis-is-not-a-genericquot-bruce-eckel-eyes-of-the-generic-java/>。

```
}  
}
```

请想一想，上面这段代码是否正确，能否编译执行？也许您已经有了答案，这段代码是不能被编译的，是因为参数 `List<Integer>` 和 `List<String>` 编译之后都被擦除了，变成了一样的原生类型 `List<E>`，擦除动作导致这两个方法的特征签名变得一模一样。初步看来，无法重载的原因已经找到了，但是真的就是如此吗？只能说，泛型擦除成相同的原生类型只是无法重载的一部分原因，请再接着看一看代码清单 10-5 中的内容。

代码清单 10-5 当泛型遇见重载 2

```
public class GenericTypes {  
  
    public static String method(List<String> list) {  
        System.out.println("invoke method(List<String> list)");  
        return "";  
    }  
  
    public static int method(List<Integer> list) {  
        System.out.println("invoke method(List<Integer> list)");  
        return 1;  
    }  
  
    public static void main(String[] args) {  
        method(new ArrayList<String>());  
        method(new ArrayList<Integer>());  
    }  
}
```

执行结果：

```
invoke method(List<String> list)  
invoke method(List<Integer> list)
```

代码清单 10-5 与代码清单 10-4 的差别，是两个 `method` 方法添加了不同的返回值，由于这两个返回值的加入，方法重载居然成功了，即这段代码可以被编译和执行^①了。

① 测试的时候请使用 Sun JDK 的 `Javac` 编译器进行编译，其他编译器，如 Eclipse JDT 的 `ECJ` 编译器，仍然可能会拒绝编译这段代码，`ECJ` 编译时会提示 “Method `method(List<String>)` has the same erasure `method(List<E>)` as another method in type `GenericTypes`”。

这是我们对 Java 语言中返回值不参与重载选择的基本认知的挑战吗？

代码清单 10-5 中的重载当然不是根据返回值来确定的，之所以这次能编译和执行成功，是因为两个 `method()` 方法加入了不同的返回值后才能共存在一个 Class 文件之中。第 6 章介绍 Class 文件方法表 (`method_info`) 的数据结构时曾经提到过，方法重载要求方法具备不同的特征签名，返回值并不包含在方法的特征签名之中，所以返回值不参与重载选择，但是在 Class 文件格式之中，只要描述符不是完全一致的两个方法就可以共存。也就是说两个方法如果有相同的名称和特征签名，但返回值不同，那它们也是可以合法地共存于一个 Class 文件中的。

由于 Java 泛型的引入，各种场景（虚拟机解析、反射等）下的方法调用都可能对原有的基础产生影响和新的需求，如在泛型类中如何获取传入的参数化类型等。所以 JCP 组织对虚拟机规范做出了相应的修改，引入了诸如 `Signature` 和 `LocalVariableTypeTable` 等新的属性用于解决伴随泛型而来的参数类型的识别问题，`Signature` 是其中最重要的一项属性，它的作用就是存储一个方法在字节码层面的特征签名^①，这个属性中保存的参数类型并不是原生类型，而是包括了参数化类型的信息。修改后的虚拟机规范^②要求所有能识别 49.0 以上版本的 Class 文件的虚拟机都要能正确地识别 `Signature` 参数。

从上面的例子可以看到擦除法对实际编码带来的影响，由于 `List<String>` 和 `List<Integer>` 擦除后是同一个类型，我们只能添加两个并不需要实际使用到的返回值才能完成重载，这是一种毫无优雅和美感可言的解决方案。同时，从 `Signature` 属性的出现我们还可以得出结论，擦除法所谓的擦除，仅仅是对方法的 `Code` 属性中的字节码进行擦除，实际上元数据中还是保留了泛型信息，这也是我们能通过反射手段取得参数化类型的根本依据。

① 在《Java 虚拟机规范（第 2 版）》（JDK 1.5 修改后的版本）的“§ 4.4.4 Signatures”章节及《Java 语言规范（第 3 版）》的“§ 8.4.2 Method Signature”章节中分别定义了字节码层面的方法特征签名，以及 Java 代码层面的方法特征签名，特征签名最重要的任务就是作为方法独一无二不可重复的 ID，在 Java 代码中的方法特征签名只包括了方法名称、参数顺序及参数类型，而在字节码中的特征签名还包括方法返回值及受查异常表，本书中如果指的是字节码层面的方法签名，笔者会加入限定语进行说明，也请读者根据上下文语境注意区分。

② JDK 1.5 对虚拟机规范修改：http://java.sun.com/docs/books/jvms/second_edition/jvms-clarify.html。

10.3.2 自动装箱、拆箱与遍历循环

就纯技术的角度而论，自动装箱、自动拆箱与遍历循环（Foreach 循环）这些语法糖，无论是实现上还是思想上都不能和上一节介绍的泛型相比，两者的难度和深度都有很大的差距。专门拿出一节来讲解它们只有一个理由：毫无疑问，它们是 Java 语言里面被使用得最多的语法糖。我们通过下面的代码清单 10-6 和代码清单 10-7 中所示的代码来看看这些语法糖在编译后会发生什么样的变化。

代码清单 10-6 自动装箱、拆箱与遍历循环

```
public static void main(String[] args) {
    List<Integer> list = Arrays.asList(1, 2, 3, 4);
    // 如果在 JDK 1.7 中，还有另外一颗语法糖①，
    // 能让上面这句代码进一步简写成 List<Integer> list = [1, 2, 3, 4];
    int sum = 0;
    for (int i : list) {
        sum += i;
    }
    System.out.println(sum);
}
```

代码清单 10-7 自动装箱、拆箱与遍历循环编译之后

```
public static void main(String[] args) {
    List list = Arrays.asList( new Integer[] {
        Integer.valueOf(1),
        Integer.valueOf(2),
        Integer.valueOf(3),
        Integer.valueOf(4) });

    int sum = 0;
    for (Iterator localIterator = list.iterator(); localIterator.hasNext(); ) {
        int i = ((Integer)localIterator.next()).intValue();
        sum += i;
    }
    System.out.println(sum);
}
```

^① 在本章完稿之后，此语法糖随着 Project Coin 一起被划分到 JDK 1.8 中了，在 JDK 1.7 里不会包括。

上面代码清单 10-6 中一共包含了泛型、自动装箱、自动拆箱、遍历循环与变长参数五种语法糖，代码清单 10-7 则展示了它们在编译后的变化。泛型就不必说了，自动装箱、拆箱在编译之后被转化成了对应的包装和还原方法，如本例子中的 `Integer.valueOf()` 与 `Integer.intValue()` 方法，而遍历循环则是把代码还原成了迭代器的实现，这也是为何遍历循环需要被遍历的类实现 `Iterable` 接口的原因。最后再看看变长参数，它在调用的时候变成了一个数组类型的参数，在变长参数出现之前，程序员就是使用数组来完成类似功能的。

这些语法糖虽然看起来很简单，但也不见得就没有任何值得我们注意的地方，下面代码清单 10-8 演示了自动装箱的一些错误用法。

代码清单 10-8 自动装箱的陷阱

```
public static void main(String[] args) {  
    Integer a = 1;  
    Integer b = 2;  
    Integer c = 3;  
    Integer d = 3;  
    Integer e = 321;  
    Integer f = 321;  
    Long g = 3L;  
    System.out.println(c == d);  
    System.out.println(e == f);  
    System.out.println(c == (a + b));  
    System.out.println(c.equals(a + b));  
    System.out.println(g == (a + b));  
    System.out.println(g.equals(a + b));  
}
```

看完代码清单 10-8，不妨思考两个问题：一是代码中的这六句打印语句的输出是什么？二是这六句打印语句中，解除语法糖后参数会是什么样子？这两个问题的答案很容易就都能试验出来，笔者在此暂且略去答案，希望读者自己上机实践一下。无论你的回答是否正确，鉴于包装类的“==”运算在没有遇到算术运算的情况下不会自动拆箱，而且它们的 `equals()` 方法不会处理数据转型的关系，笔者建议在实际编码中应尽量避免这样来使用自动装箱与拆箱。

10.3.3 条件编译

许多程序设计语言都提供了条件编译的途径，如 C、C++ 中使用预处理器指示符（`#ifdef`）来完成条件编译。C、C++ 的预处理器最初的任务是解决编译时的代码依赖关系（众所周知的 `#include` 预处理命令），而在 Java 语言之中并没有使用预处理器，因为 Java 语言天然的编译方式（编译器并非一个一个地编译 Java 文件，而是将所有的编译单元的语法树顶级节点输入到待处理列表后再进行编译，因此各个文件之间能够互相提供符号信息）无须使用预处理器。那 Java 语言是否有办法实现条件编译呢？

Java 语言当然也可以进行条件编译，方法就是使用条件为常量的 `if` 语句。如代码清单 10-9 所示，此代码中的 `if` 语句不同于其他 Java 代码，它在编译阶段就会被“运行”，生成的字节码之中只包括“`System.out.println("block 1");`”一条语句，并不会包含 `if` 语句及另外一个分子中的“`System.out.println("block 2");`”

代码清单 10-9 Java 语言的条件编译

```
public static void main(String[] args) {  
    if (true) {  
        System.out.println("block 1");  
    } else {  
        System.out.println("block 2");  
    }  
}
```

此代码编译后 Class 文件的反编译结果：

```
public static void main(String[] args) {  
    System.out.println("block 1");  
}
```

只能使用条件为常量的 `if` 语句才能达到上述效果，如果使用常量与其他带有条件判断能力的语句搭配，则可能在控制流分析中提示错误，被拒绝编译，如下面的代码清单 10-10 所示的代码就会被编译器拒绝编译。

代码清单 10-10 不能使用其他条件语句来完成条件编译

```
public static void main(String[] args) {  
    // 编译器将会提示“Unreachable code”  
    while (false) {
```

```
        System.out.println("");  
    }  
}
```

Java 语言中条件编译的实现，也是 Java 语言的一颗语法糖，根据布尔常量值的真假，编译器将会把分支中不成立的代码块消除掉，这一工作将在编译器解除语法糖的阶段（com.sun.tools.javac.comp.Lower 类中）完成。由于这种条件编译的实现方式使用了 if 语句，所以它必须遵循最基本的 Java 语法，只能写在方法体内部，因此它只能实现语句基本块（Block）级别的条件编译，而没有办法实现根据条件调整整个 Java 类的结构。

除了本节中介绍的泛型、自动装箱、自动拆箱、遍历循环、变长参数和条件编译之外，Java 语言还有不少其他的语法糖，如内部类、枚举类、断言语句、对枚举和字符串（在 JDK 1.7 中支持）的 switch 支持、在 try 语句中定义和关闭资源（在 JDK 1.7 中支持）等，你可以通过跟踪 Javac 源码、反编译 Class 文件等方式了解他们的本质实现，囿于篇幅，笔者就不再一一介绍了。

10.4 实战：插入式注解处理器

JDK 编译优化部分在本书中并没有设置独立的实战章节，因为我们开发程序，考虑的主要是程序会如何运行，很少会有针对程序编译的需求。也因为这个原因，在 JDK 的编译子系统里面，提供给用户直接控制的功能相对较少，除了下一章会介绍的虚拟机 JIT 编译的几个相关参数以外，我们就只有使用 JSR-296 中定义的插入式注解处理器 API 来对 JDK 编译子系统的行为产生一些影响。

但是笔者并不认为相对于前两部分介绍的内存管理子和字节码执行子系统，JDK 的编译子系统就不那么重要了。一套编程语言中编译子系统的优劣，很大程度上决定了程序运行性能的好坏和编码效率的高低，尤其在 Java 语言中，运行期即时编译与虚拟机执行子系统非常紧密地互相依赖并配合运作（下一章将主要讲解这方面的内容）。了解 JDK 如何编译和优化代码，有助于我们写出适合 JDK 自优化的程序。话题说远了，下面再回到本章的实战中来，看看插入式注解处理器 API 能为我们实现什么功能。

10.4.1 实战目标

通过阅读 Javac 编译器的源码，我们知道编译器在把 Java 程序源码编译为字节码

的时候，会对 Java 程序源码做各方面的检查校验。这些校验主要以程序“写得对不对”为出发点，虽然也有各种 WARNING 的信息，但总体来讲还是较少去校验程序“写得好不好”。有鉴于此，业界出现了许多针对程序“写得好不好”的辅助校验工具，如 CheckStyle、FindBug、Klocwork 等。这些代码校验工具有一些是基于 Java 的源码进行校验，有一些是通过扫描字节码来完成，在本节的实战中，我们将会使用注解处理器 API 来编写一款拥有自己编码风格的校验工具：NameCheckProcessor。

当然，由于我们的实战都是为了学习和演示技术原理，而不是为了做出一款能媲美 CheckStyle 等工具的产品来，所以 NameCheckProcessor 的目标也仅定为对 Java 程序命名进行检查，根据《Java 语言规范（第 3 版）》中第 6.8 节的要求，Java 程序命名应当符合下列格式的书写规范：

- 类（或接口）：符合驼式命名法，首字母大写。
- 方法：符合驼式命名法，首字母小写。
- 字段：
 - 类或实例变量：符合驼式命名法，首字母小写。
 - 常量：要求全部由大写字母或下划线构成，并且第一个字符不能是下划线。

上文提到的驼式命名法（Camel Case Names），正如它的名称所表示的那样，是指混合使用大小写字母来分割构成变量或函数的名字，犹如驼峰一般，这是当前 Java 语言中主流的命名规范，我们的实战目标就是为 Javac 编译器添加一个额外的功能，在编译程序时检查程序名是否符合上述对类（或接口）、方法、字段的命名要求^①。

10.4.2 代码实现

要通过注解处理 API 实现一个编译器插件，首先需要了解这组 API 的一些基本知识。我们实现注解处理器的代码需要继承抽象类 `javax.annotation.processing.AbstractProcessor`，这个抽象类中只有一个必须覆盖的 abstract 方法：“`process()`”，它是 Javac 编译器在执行注解处理器代码时要调用的过程，我们可以从这个方法的第一个参数“`annotations`”中获取到此注解处理器所要处理的注解集合，从第二个参数“`roundEnv`”中访问到当前这个 Round 中的语法树节点，每个语法树节点在这里表示为

^① 在 JDK 的 `sample/javac/processing` 目录中有这次实战的源码（稍微复杂一些，但大体上差距不大），读者可以阅读参考。

一个 Element。在 JDK 1.6 新增的 `javax.lang.model` 包中定义了 16 类 Element，包括了 Java 代码中最常用的元素，如：“包 (PACKAGE)、枚举 (ENUM)、类 (CLASS)、注解 (ANNOTATION_TYPE)、接口 (INTERFACE)、枚举值 (ENUM_CONSTANT)、字段 (FIELD)、参数 (PARAMETER)、本地变量 (LOCAL_VARIABLE)、异常 (EXCEPTION_PARAMETER)、方法 (METHOD)、构造函数 (CONSTRUCTOR)、静态语句块 (STATIC_INIT，即 `static{}` 块)、实例语句块 (INSTANCE_INIT，即 `{}` 块)、参数化类型 (TYPE_PARAMETER，即泛型尖括号内的类型) 和未定义的其他语法树节点 (OTHER)”。除了 `process()` 方法的传入参数之外，还有一个很常用的实例变量 “`processingEnv`”，它是 `AbstractProcessor` 中的一个 `protected` 变量，在注解处理器初始化的时候 (`init()` 方法执行的时候) 创建，继承了 `AbstractProcessor` 的注解处理器代码可以直接访问到它。它代表了注解处理器框架提供的一个上下文环境，要创建新的代码、向编译器输出信息、获取其他工具类等都需要用到这个实例变量。

注解处理器除了 `process()` 方法及其参数之外，还有两个可以配合使用的 Annotations：`@SupportedAnnotationTypes` 和 `@SupportedSourceVersion`，前者代表了在这个注解处理器对哪些注解感兴趣，可以使用星号 “*” 作为通配符代表对所有的注解都感兴趣，后者指出这个注解处理器可以处理哪些版本的 Java 代码。

每一个注解处理器在运行的时候都是单例的，如果不需要改变或生成语法树的内容，`process()` 方法就可以返回一个值为 `false` 的布尔值，通知编译器这个 Round 中的代码未发生变化，无须构造新的 `JavaCompiler` 实例，在这次实战的注解处理器中只对程序命名进行检查，不需要改变语法树的内容，因此 `process()` 方法的返回值都是 `false`。关于注解处理器的 API，笔者就简单介绍这些，对这个领域有兴趣的读者可以阅读相关的帮助文档。我们来看看注解处理器 `NameCheckProcessor` 的具体代码，如代码清单 10-11 所示。

代码清单 10-11 注解处理器 `NameCheckProcessor`

```
// 可以用 "*" 表示支持所有的 Annotations
@SupportedAnnotationTypes ("*")
// 只支持 JDK 1.6 的 Java 代码
@SupportedSourceVersion (SourceVersion.RELEASE_6)
public class NameCheckProcessor extends AbstractProcessor {
```

```
private NameChecker nameChecker;

/**
 * 初始化名称检查插件
 */
@Override
public void init(ProcessingEnvironment processingEnv) {
    super.init(processingEnv);
    nameChecker = new NameChecker(processingEnv);
}

/**
 * 对输入的语法树的各个节点进行进行名称检查
 */
@Override
public boolean process(Set<? extends TypeElement> annotations,
    RoundEnvironment roundEnv) {
    if (!roundEnv.processingOver()) {
        for (Element element : roundEnv.getRootElements())
            nameChecker.checkNames(element);
    }
    return false;
}
}
```

从上面的代码可以看到 NameCheckProcessor 能处理基于 JDK 1.6 的源码，它不限于特定的注解，对任何代码都“感兴趣”，而在 process() 方法中是把当前 Round 中的每一个 RootElement 传递到一个名为 NameChecker 的检查器中执行名称检查逻辑，NameChecker 的代码如代码清单 10-12 所示。

代码清单 10-12 命名检查器 NameChecker

```
/**
 * 程序名称规范的编译器插件：<br>
 * 如果程序命名不合规范，将会输出一个编译器的 WARNING 信息
 */
public class NameChecker {
    private final Messenger messenger;

    NameCheckScanner nameCheckScanner = new NameCheckScanner();
```

280 ❖ 第四部分 程序编译与代码优化

```
NameChecker(ProcessingEnvironment processingEnv) {
    this.messenger = processingEnv.getMessenger();
}
```

```
/**
```

* 对 Java 程序命名进行检查, 根据《Java 语言规范 (第 3 版)》6.8 节的要求, Java 程序命名应当符合下列格式:

```
*
* <ul>
* <li> 类或接口: 符合驼式命名法, 首字母大写。
* <li> 方法: 符合驼式命名法, 首字母小写。
* <li> 字段:
* <ul>
* <li> 类、实例变量: 符合驼式命名法, 首字母小写。
* <li> 常量: 要求全部大写。
* </ul>
* </ul>
*/
```

```
public void checkNames(Element element) {
    nameCheckScanner.scan(element);
}
```

```
/**
```

```
* 名称检查器实现类, 继承了 JDK 1.6 中新提供的 ElementScanner6<br>
* 将会以 Visitor 模式访问抽象语法树中的元素
*/
```

```
private class NameCheckScanner extends ElementScanner6<Void, Void> {
```

```
/**
```

```
* 此方法用于检查 Java 类
```

```
*/
```

```
@Override
```

```
public Void visitType(TypeElement e, Void p) {
    scan(e.getTypeParameters(), p);
    checkCamelCase(e, true);
    super.visitType(e, p);
    return null;
}
```

```
/**
```

```
* 检查方法命名是否合法
```



```

*/
@Override
public Void visitExecutable(ExecutableElement e, Void p) {
    if (e.getKind() == METHOD) {
        Name name = e.getSimpleName();
        if (name.contentEquals(e.getEnclosingElement().getSimpleName()))
            messenger.printMessage(WARNING, "一个普通方法 " + name + " 不应
            当与类名重复, 避免与构造函数产生混淆 ", e);
        checkCamelCase(e, false);
    }
    super.visitExecutable(e, p);
    return null;
}

/**
 * 检查变量命名是否合法
 */
@Override
public Void visitVariable(VariableElement e, Void p) {
    // 如果这个 Variable 是枚举或常量, 则按大写命名检查, 否则按照驼式命名法规则检查
    if (e.getKind() == ENUM_CONSTANT || e.getConstantValue() != null ||
    heuristicallyConstant(e))
        checkAllCaps(e);
    else
        checkCamelCase(e, false);
    return null;
}

/**
 * 判断一个变量是否是常量
 */
private boolean heuristicallyConstant(VariableElement e) {
    if (e.getEnclosingElement().getKind() == INTERFACE)
        return true;
    else if (e.getKind() == FIELD && e.getModifiers().containsAll(EnumSet.
of(PUBLIC, STATIC, FINAL)))
        return true;
    else {
        return false;
    }
}

/**

```


282 ❖ 第四部分 程序编译与代码优化

```
* 检查传入的 Element 是否符合驼式命名法, 如果不符合, 则输出警告信息
*/
private void checkCamelCase(Element e, boolean initialCaps) {
    String name = e.getSimpleName().toString();
    boolean previousUpper = false;
    boolean conventional = true;
    int firstCodePoint = name.codePointAt(0);

    if (Character.isUpperCase(firstCodePoint)) {
        previousUpper = true;
        if (!initialCaps) {
            messenger.printMessage(WARNING, "名称 " + name + " 应当以小写字
            母开头", e);
            return;
        }
    } else if (Character.isLowerCase(firstCodePoint)) {
        if (initialCaps) {
            messenger.printMessage(WARNING, "名称 " + name + " 应当以大写字
            母开头", e);
            return;
        }
    } else
        conventional = false;

    if (conventional) {
        int cp = firstCodePoint;
        for (int i = Character.charCount(cp); i < name.length(); i +=
        Character.charCount(cp)) {
            cp = name.codePointAt(i);
            if (Character.isUpperCase(cp)) {
                if (previousUpper) {
                    conventional = false;
                    break;
                }
                previousUpper = true;
            } else
                previousUpper = false;
        }
    }

    if (!conventional)
        messenger.printMessage(WARNING, "名称 " + name + " 应当符合驼式命名法
        (Camel Case Names)", e);
}
```




```
}

/**
 * 大写命名检查, 要求第一个字母必须是大写的英文字母, 其余部分可以是下划线或大写字母
 */
private void checkAllCaps(Element e) {
    String name = e.getSimpleName().toString();

    boolean conventional = true;
    int firstCodePoint = name.codePointAt(0);

    if (!Character.isUpperCase(firstCodePoint))
        conventional = false;
    else {
        boolean previousUnderscore = false;
        int cp = firstCodePoint;
        for (int i = Character.charCount(cp); i < name.length(); i +=
            Character.charCount(cp)) {
            cp = name.codePointAt(i);
            if (cp == (int) '_') {
                if (previousUnderscore) {
                    conventional = false;
                    break;
                }
                previousUnderscore = true;
            } else {
                previousUnderscore = false;
                if (!Character.isUpperCase(cp) && !Character.isDigit(cp)) {
                    conventional = false;
                    break;
                }
            }
        }
    }

    if (!conventional)
        messenger.printMessage(WARNING, "常量" + name + "应当全部以大写字母
        或下划线命名, 并且以字母开头", e);
}
}
```



284 ❖ 第四部分 程序编译与代码优化

NameChecker 的代码看起来有点长，但实际上注释占了很大一部分，而且即使算上注释也不到 190 行。它通过一个继承于 `javax.lang.model.util.ElementScanner6` 的 `NameCheckScanner` 类，以 Visitor 模式来完成对语法树的遍历，分别执行 `visitType()`、`visitVariable()` 和 `visitExecutable()` 方法来访问类、字段和方法，这 3 个 visit 方法对各自的命名规则做相应的检查，`checkCamelCase()` 与 `checkAllCaps()` 方法则用于实现驼式命名法和全大写命名规则的检查。

整个注解处理器只需 `NameCheckProcessor` 和 `NameChecker` 两个类就可以全部完成，为了验证我们的实战成果，代码清单 10-13 中提供了一段命名规范的“反面教材”代码，其中的每一个类、方法及字段的命名都存在问题，但是使用普通的 `Javac` 编译这段代码时不会提示任意一个 Warning 信息。

代码清单 10-13 包含了多处不规范命名的代码样例

```
public class BADLY_NAMED_CODE {  
  
    enum colors {  
        red, blue, green;  
    }  
  
    static final int _FORTY_TWO = 42;  
  
    public static int NOT_A_CONSTANT = _FORTY_TWO;  
  
    protected void BADLY_NAMED_CODE() {  
        return;  
    }  
  
    public void NOTcamelCASEmethodName() {  
        return;  
    }  
}
```

10.4.3 运行与测试

我们可以通过 `Javac` 命令的“-processor”参数来执行编译时需要附带的注解处理器，如果有多个注解处理器的话，用逗号分隔。还可以使用 `-XprintRounds` 和 `-XprintProcessorInfo` 参数来查看注解处理器运作的详细信息，本次实战中的

NameCheckProcessor 的编译及执行过程如代码清单 10-14 所示。

代码清单 10-14 注解处理器的运行过程

```
D:\src>javac org/fenixsoft/compile/NameChecker.java

D:\src>javac org/fenixsoft/compile/NameCheckProcessor.java

D:\src>javac -processor org.fenixsoft.compile.NameCheckProcessor org/fenixsoft/
compile/BADLY_NAMED_CODE.java

org\fenixsoft\compile\BADLY_NAMED_CODE.java:3: 警告: 名称 "BADLY_NAMED_CODE" 应当符合
驼式命名法 (Camel Case Names)
public class BADLY_NAMED_CODE {
    ^

org\fenixsoft\compile\BADLY_NAMED_CODE.java:5: 警告: 名称 "colors" 应当以大写字母开头
    enum colors {
        ^

org\fenixsoft\compile\BADLY_NAMED_CODE.java:6: 警告: 常量 "red" 应当全部以大写字母或下划
线命名, 并且以字母开头
        red, blue, green;
        ^

org\fenixsoft\compile\BADLY_NAMED_CODE.java:6: 警告: 常量 "blue" 应当全部以大写字母或下
划线命名, 并且以字母开头
        red, blue, green;
        ^

org\fenixsoft\compile\BADLY_NAMED_CODE.java:6: 警告: 常量 "green" 应当全部以大写字母或下
划线命名, 并且以字母开头
        red, blue, green;
        ^

org\fenixsoft\compile\BADLY_NAMED_CODE.java:9: 警告: 常量 "_FORTY_TWO" 应当全部以大写字
母或下划线命名, 并且以字母开头
        static final int _FORTY_TWO = 42;
        ^

org\fenixsoft\compile\BADLY_NAMED_CODE.java:11: 警告: 名称 "NOT_A_CONSTANT" 应当以小写
字母开头
        public static int NOT_A_CONSTANT = _FORTY_TWO;
        ^

org\fenixsoft\compile\BADLY_NAMED_CODE.java:13: 警告: 名称 "Test" 应当以小写字母开头
        protected void Test() {
            ^

org\fenixsoft\compile\BADLY_NAMED_CODE.java:17: 警告: 名称 "NOTcamelCASEmethodName"
应当以小写字母开头
```



```
public void NOTcamelCASEmethodName() {
```

10.4.4 其他应用案例

NameCheckProcessor 的实战例子只演示了 JSR-269 嵌入式注解处理 API 其中的一部分功能，基于这组 API 支持的项目还有用于校验 Hibernate 标签使用正确性的 Hibernate Validator Annotation Processor^①（本质上与 NameCheckProcessor 所做的事情差不多）、自动为字段生成 getter 和 setter 方法的 Project Lombok^②（根据已有元素生成新的语法树元素）等，读者有兴趣的话可以参考它们官方站点的相关内容。

10.5 本章小结

在本章中，我们从编译器源码实现的层次上了解了 Java 源代码编译为字节码的过程，分析了 Java 语言中泛型、主动装箱拆箱、条件编译等多种语法糖的前因后果，并实战练习了如何使用插入式注解处理器来完成一个检查程序命名规范的编译器插件。如本章概述中所说的，在前端编译器中，“优化”手段主要用于提升程序的编码效率，之所以把 Javac 这类将 Java 代码转变为字节码的编译器称做“前端编译器”，是因为它只完成了从程序到抽象语法树或中间字节码的生成，而在此之后，还有一组内置于虚拟机内部的“后端编译器”完成了从字节码生成本地机器码的过程，即前面多次提到的即时编译器或 JIT 编译器，这个编译器的编译速度及编译结果的优劣，是衡量虚拟机性能很重要的一个指标。下一章将会讲解即时编译器的运作和优化过程。

① 官方站点：<http://www.hibernate.org/subprojects/validator.html>。

② 官方站点：<http://projectlombok.org/>。



第 11 章 晚期（运行期）优化

本章主要内容

- 概述
- HotSpot 虚拟机内的即时编译器
- 编译优化技术
- Java 与 C/C++ 的编译器对比

从计算机程序出现的第一天起，对效率的追逐就是程序天生的坚定信仰，这个过程犹如一场没有终点、永不停歇的 F1 方程式竞赛，程序员是车手，技术平台则是在赛道上飞驰的赛车。

11.1 概述

在部分的商用虚拟机（Sun HotSpot、IBM J9）中，Java 程序最初是通过解释器（Interpreter）进行解释执行的，当虚拟机发现某个方法或代码块的运行特别频繁，就会把这些代码认定为“热点代码”（Hot Spot Code），为了提高热点代码的执行效率，在运行时，虚拟机将会把这些代码编译成与本地平台相关的机器码，并进行各种层次的优化，完成这个任务的编译器称为即时编译器（Just In Time Compiler，下文中简称 JIT 编译器）。

即时编译器并不是虚拟机必需的部分，Java 虚拟机规范并没有规定 Java 虚拟机内必须要有即时编译器，更没有限定或指导即时编译器应该如何去实现。但是，即时编译器编译性能的好坏、代码优化程度的高低却是衡量一款商用虚拟机优秀与否的最关键的指标之一，它也是虚拟机中最核心最能体现技术水平的部分。在本章中，我们将走进虚拟机的内部，探索即时编译器的运作过程。

由于 Java 虚拟机规范没有具体的约束规则去限制即时编译器应该如何实现，所以这部分功能完全是与虚拟机具体实现（Implementation Specific）相关的内容，如无特殊说明，本章中所提及的编译器、即时编译器都是指 HotSpot 虚拟机内的即时编译器，虚拟机也是特指 HotSpot 虚拟机。不过，本章中的大部分内容是描述即时编译器的行为，涉及编译器实现层面的内容较少，而主流虚拟机中即时编译器的行为又有很多相似相通之处，因此对其他虚拟机来说也具备较大的参考意义。

11.2 HotSpot 虚拟机内的即时编译器

本节中，我们将要了解 HotSpot 虚拟机内的即时编译器的运作过程，同时，我们要解决以下几个问题：

- ❑ 为何 HotSpot 虚拟机要使用解释器与编译器并存的架构？
- ❑ 为何 HotSpot 虚拟机要实现两个不同的即时编译器？
- ❑ 程序何时使用解释器执行？何时使用编译器执行？
- ❑ 哪些程序代码会被编译为本地代码？如何编译本地代码？
- ❑ 如何从外部观察即时编译器的编译过程和编译结果？

11.2.1 解释器与编译器

尽管并不是所有的 Java 虚拟机都采用解释器与编译器并存的架构，但许多主流的商用虚拟机，如 HotSpot、J9 等，都同时包含解释器与编译器^①，解释器与编译器两者各有优势：当程序需要迅速启动和执行的时候，解释器可以首先发挥作用，省去编译的时间，立即执行。当程序运行后，随着时间的推移，编译器逐渐发挥作用，把越来越多的代码编译成本地代码之后，可以获取更高的执行效率。当程序运行环境中内存资源限制较大（如部分嵌入式系统中），可以使用解释执行节约内存，反之可以使用编译执行来提升效率。同时，解释器还可以作为编译器激进优化时的一个“逃生门”，让编译器根据概率选择一些大多数时候都能提升运行速度的优化手段，当激进优化的假设不成立，如加载了新类后类型继承结构出现变化、出现“罕见陷阱”（Uncommon Trap）时可以通

^① 作为三大商用虚拟机之一的 JRockit 是个例外，它内部没有解释器，因此会存在本书中所说的“启动响应时间长”之类的缺点，但它主要是面向服务端的应用，这类应用一般不会重点关注启动时间。

过逆优化（Deoptimization）退回到解释状态继续执行（部分没有解释器的虚拟机中也会采用不进行激进优化的 C1 编译器担任“逃生门”的角色），因此在整个虚拟机执行架构中，解释器与编译器经常是相辅相成地配合工作的，如图 11-1 所示。

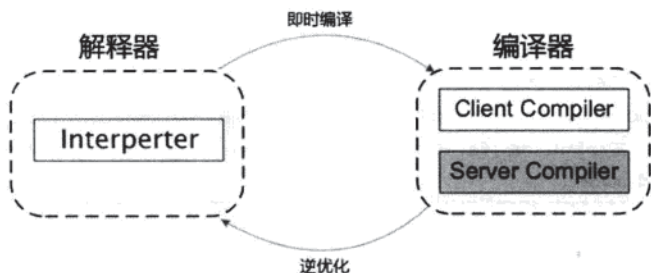


图 11-1 解释器与编译器的交互

HotSpot 虚拟机中内置了两个即时编译器，称为 Client Compiler 和 Server Compiler，或者简称为 C1 编译器和 C2 编译器（也叫 Opto 编译器）。目前主流的 HotSpot 虚拟机中（Sun 系列 JDK 1.6 及之前版本的虚拟机），默认是采用解释器与其中一个编译器直接配合的方式工作，程序使用哪个编译器，取决于虚拟机运行的模式，HotSpot 虚拟机会根据自身版本与宿主机器的硬件性能自动选择运行模式，用户也可以使用 `-client` 或 `-server` 参数去强制指定虚拟机运行在 Client 模式还是 Server 模式。

无论采用的编译器是 Client Compiler 还是 Server Compiler，解释器与编译器搭配使用的方式在虚拟机中被称为“混合模式”（Mixed Mode），用户可以使用参数 `-Xint` 强制虚拟机运行于“解释模式”（Interpreted Mode），这时候编译器完全不介入工作，全部代码都使用解释方式执行。另外，也可以使用参数 `-Xcomp` 强制虚拟机运行于“编译模式”（Compiled Mode），这时候将优先采用编译方式执行程序，但是解释器仍然要在编译无法进行的情况下介入执行过程，可以通过虚拟机的 `-version` 命令的输出结果显示出这三种模式，如代码清单 11-1 所示，请读者注意**黑体字**部分。

代码清单 11-1 虚拟机执行模式

```

C:\>java -version
java version "1.6.0_22"
Java(TM) SE Runtime Environment (build 1.6.0_22-b04)
Dynamic Code Evolution 64-Bit Server VM (build 0.2-b02-internal, 19.0-b04-internal, mixed mode)
  
```


290 ❖ 第四部分 程序编译与代码优化

```
C:\>java -Xint -version
java version "1.6.0_22"
Java(TM) SE Runtime Environment (build 1.6.0_22-b04)
Dynamic Code Evolution 64-Bit Server VM (build 0.2-b02-internal, 19.0-b04-internal, interpreted mode)
```

```
C:\>java -Xcomp -version
java version "1.6.0_22"
Java(TM) SE Runtime Environment (build 1.6.0_22-b04)
Dynamic Code Evolution 64-Bit Server VM (build 0.2-b02-internal, 19.0-b04-internal, compiled mode)
```

由于即时编译器编译本地代码需要占用程序运行时间，要编译出优化程度更高的代码，所花费的时间可能越长；而且想要编译出优化程度更高的代码，解释器可能还要替编译器收集性能监控信息，这对解释执行的速度也有所影响。为了在程序启动响应速度与运行效率之间达到最佳平衡，HotSpot 虚拟机将会逐渐启用分层编译（Tiered Compilation）的策略^①，分层编译的概念在 JDK 1.6 时期出现，后来一直处于改进阶段，最终在 JDK 1.7 的 Server 模式虚拟机中作为默认编译策略被开启。分层编译根据编译器编译、优化的规模与耗时，划分出不同的编译层次，其中包括：

- 第 0 层：程序解释执行，解释器不开启性能监控功能（Profiling），可触发第 1 层编译。
- 第 1 层：也称为 C1 编译，将字节码编译为本地代码，进行简单可靠的优化，如有必要将加入性能监控的逻辑。
- 第 2 层（或 2 层以上）：也称为 C2 编译，也是将字节码编译为本地代码，但是会启用一些编译耗时较长的优化，甚至会根据性能监控信息进行一些不可靠的激进优化。

实施分层编译后，Client Compiler 和 Server Compiler 将会同时工作，许多代码都可能被多次编译，用 Client Compiler 获取更高的编译速度，用 Server Compiler 来获取更好的编译质量，在解释执行的时候也无须再承担收集性能监控信息的任务。

^① Tiered Compilation 的概念在 JDK 1.6 时期出现，但 JDK 1.7 之前需要使用 `-XX:+TieredCompilation` 参数来手动开启，如果不开启分层编译策略，而虚拟机又运行在 Server 模式，Server Compiler 需要性能监控信息提供编译依据，则可由解释器收集性能监控信息供 Server Compiler 使用。分层编译的相关资料可参见：<http://weblogs.java.net/blog/forax/archive/2010/09/04/tiered-compilation>。

11.2.2 编译对象与触发条件

在概述中提到过在运行过程中会被即时编译器编译的“热点代码”有两类，即：

- 被多次调用的方法。
- 被多次执行的循环体。

前者很好理解，一个方法被调用得多了，方法体内代码执行的次数自然就多，它成为“热点代码”是理所当然的。而后者则是为了解决当一个方法只被调用过一次或几次，但是方法体内部存在循环次数较多的循环体，这样循环体的代码也被重复执行多次，因此这些代码也应该成为“热点代码”。

对于第一种情况，由于是由方法调用触发的编译，那编译器理所当然地会以整个方法作为编译对象，这种编译也是虚拟机中标准的编译方式。而对于后一种情况，尽管编译动作是由循环体所触发的，但编译器依然会以整个方法（而不是单独的循环体）作为编译对象。这种编译方式因为编译发生在方法执行过程之中，因此被很形象地称为栈上替换（On Stack Replacement, OSR）。

读者可能还会有疑问，在上面的文字描述里，无论是“多次执行的方法”，还是“多次执行的代码块”，所谓“多次”都不是一个具体、严谨的用语，那到底多少次才算“多次”呢？还有一个问题，就是虚拟机如何统计一个方法或一段代码被执行过多少次呢？解决了这两个问题，也就回答了即时编译被触发的条件。

要知道一段代码是不是热点代码，是不是需要触发即时编译，这个行为称为热点探测（Hot Spot Detection），其实进行热点探测并不一定要知道方法具体被调用了多少次，目前主要的热点探测判定方式有两种^①，分别是：

- 基于采样的热点探测（Sample Based Hot Spot Detection）：采用这种方法的虚拟机会周期性地检查各个线程的栈顶，如果发现某个（或某些）方法经常出现在栈顶，那这个方法就是“热点方法”。基于采样的热点探测的好处是实现简单高效，还可以很容易地获取方法调用关系（将调用堆栈展开即可），缺点是很难精确地确认一个方法的热度，容易因为受到线程阻塞或别的外界因素的影响而扰乱热点探测。

^① 除这两种方式外，还有其他热点代码的探测方式，如基于“踪迹”（Trace）的热点探测在最近相当流行，像 Firefox 里的 TraceMonkey 和 Dalvik 里新的 JIT 编译器都使用这种热点探测方式。

□ 基于计数器的热点探测 (Counter Based Hot Spot Detection)：采用这种方法的虚拟机会为每个方法（甚至是代码块）建立计数器，统计方法的执行次数，如果执行次数超过一定的阈值就认为它是“热点方法”。这种统计方法实现起来麻烦一些，需要为每个方法建立并维护计数器，而且不能直接获取到方法的调用关系。但是它的统计结果相对来说更加精确严谨。

在 HotSpot 虚拟机中使用的是第二种——基于计数器的热点探测方法，因此它为每个方法准备了两个计数器：方法调用计数器 (Invocation Counter) 和回边计数器 (Back Edge Counter)。

在确定虚拟机运行参数的前提下，这两个计数器都有一个确定的阈值，当计数器超过阈值溢出了，就会触发 JIT 编译。

我们首先来看看方法调用计数器。顾名思义，这个计数器用于统计方法被调用的次数，它的默认阈值在 Client 模式下是 1500 次，在 Server 模式下是 10000 次，这个阈值可以通过虚拟机参数 `-XX:CompileThreshold` 来人工设定。当一个方法被调用时，会先检查该方法是否存在被 JIT 编译过的版本，如果存在，则优先使用编译后的本地代码来执行。如果不存在已被编译过的版本，则将此方法的调用计数器值加 1，然后判断方法调用计数器与回边计数器值之和是否超过方法调用计数器的阈值。如果已超过阈值的话，将会向即时编译器提交一个该方法的代码编译请求。

在默认设置下，执行引擎并不会同步等待编译请求完成，而是继续进入解释器按照解释方式执行字节码，直到提交的请求被编译器编译完成。当编译工作完成之后，这个方法的调用入口地址就会被系统自动改写成新的地址，下一次调用该方法时就会使用已编译的版本，整个 JIT 编译的交互过程如图 11-2 所示。

在默认设置下，方法调用计数器统计的并不是方法被调用的绝对次数，而是一个相对的执行频率，即一段时间之内方法被调用的次数。当超过一定的时间限度，如果方法的调用次数仍然不足以让它提交给即时编译器编译，那这个方法的调用计数器就会被减少一半，这个过程称为方法调用计数器的热度衰减 (Counter Decay)，而这段时间就称为此方法统计的半衰周期 (Counter Half Life Time)，进行热度衰减的动作是在虚拟机进行垃圾收集时顺便进行的，可以使用虚拟机参数 `-XX:-UseCounterDecay` 来关闭热度衰减，让方法计数器统计方法调用的绝对次数，这样，只要系统运行时间足够长，绝大部分方法都会被编译成本地代码。另外可以使用 `-XX:CounterHalfLifeTime` 参数设置半衰

周期的时间，单位是秒。

现在我们来再看看另外一个计数器——回边计数器，它用于统计一个方法中循环体代码执行的次数^①，在字节码中遇到控制流向后跳转的指令就称为“回边（Back Edge）”，显然建立回边计数器统计的目的就是为了触发 OSR 编译。

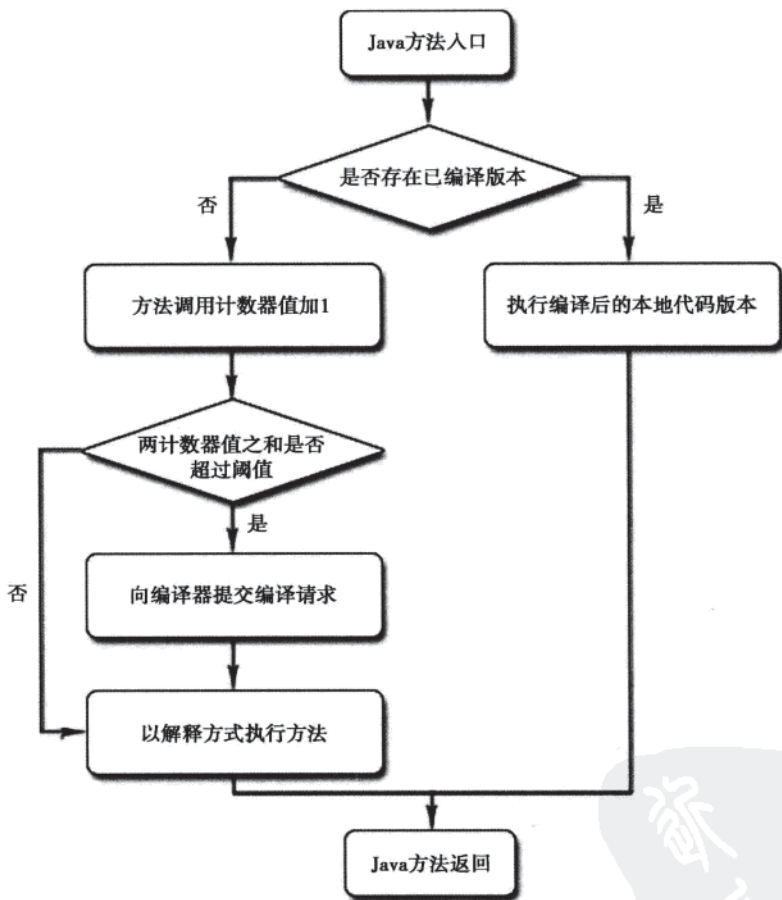


图 11-2 方法调用计数器触发即时编译

关于回边计数器的阈值，虽然 HotSpot 虚拟机也提供了一个类似于方法调用计数器阈值 `-XX:CompileThreshold` 的参数 `-XX:BackEdgeThreshold` 供用户设置，但

^① 准确地说，应当是回边的次数而不是循环次数，因为并非所有的循环都是回边，如空循环实际上就可以视为自己跳转到自己的过程，因此并不算作控制流向后跳转，也不会被回边计数器统计。

是当前的虚拟机实际上并未使用此参数，因此我们需要设置另外一个参数 `-XX:OnStackReplacePercentage` 来间接调整回边计数器的阈值，其计算公式为：

- 虚拟机运行在 Client 模式下，回边计数器阈值计算公式为：方法调用计数器阈值（`CompileThreshold`）乘以 OSR 比率（`OnStackReplacePercentage`）除以 100。其中 `OnStackReplacePercentage` 默认值为 933，如果都取默认值，那 Client 模式虚拟机的回边计数器的阈值为 13995。
- 虚拟机运行在 Server 模式下，回边计数器阈值的计算公式为：方法调用计数器阈值（`CompileThreshold`）乘以（OSR 比率（`OnStackReplacePercentage`）减去解释器监控比率（`InterpreterProfilePercentage`）的差值）除以 100。其中 `OnStackReplacePercentage` 默认值为 140，`InterpreterProfilePercentage` 默认值为 33，如果都取默认值，那 Server 模式虚拟机回边计数器的阈值为 10700。

当解释器遇到一条回边指令时，会先查找将要执行的代码片段是否有已经编译好的版本，如果有的话，它将会优先执行已编译的代码，否则就把回边计数器的值加 1，然后判断方法调用计数器的值与回边计数器的值两者之和是否超过回边计数器的阈值。当超过阈值的时候，将会提交一个 OSR 编译请求，并且把回边计数器的值降低一些，以便继续在解释器中执行循环，等待编译器输出编译结果，整个执行过程如图 11-3 所示。

与方法计数器不同，回边计数器没有计数热度衰减的过程，因此这个计数器统计的就是该方法循环执行的绝对次数。当计数器溢出的时候，它还会把方法计数器的值也调整到溢出状态，这样下次再进入该方法的时候就会执行标准编译过程。

最后需要提醒一点，图 11-2 和图 11-3 都仅仅是描述了 Client VM 的即时编译方式，对于 Server VM 来说，执行情况会比上面描述的还要复杂。

11.2.3 编译过程

在默认设置下，无论是方法调用产生的即时编译请求，还是 OSR 编译请求，虚拟机在代码编译器还未完成之前，都仍然将按照解释方式继续执行，而编译动作则在后台的编译线程中进行。用户可以通过参数 `-XX:-BackgroundCompilation` 来禁止后台编译，禁止后台编译后，当达到 JIT 的编译条件，执行线程向虚拟机提交编译请求后将会一直等待，直到编译过程完成后再开始执行编译器输出的本地代码。

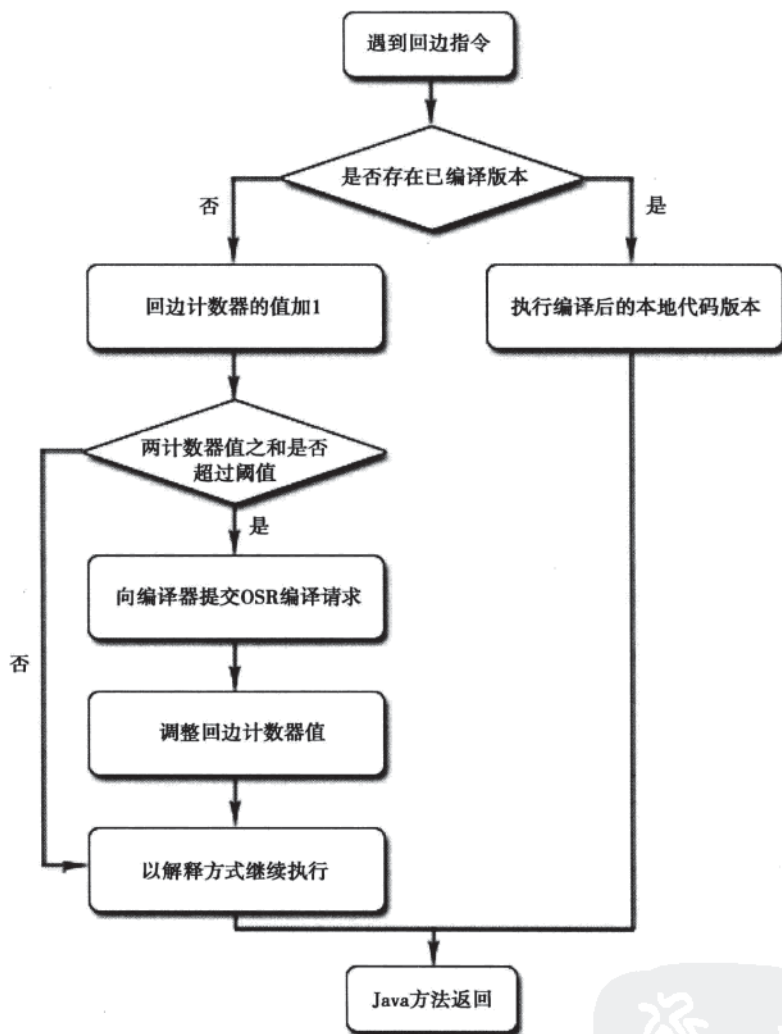


图 11-3 回边计数器触发即时编译

那在后台执行编译的过程中，编译器做了什么事情呢？Server Compiler 和 Client Compiler 两个编译器的编译过程是不一样的。对于 Client Compiler 来说，它是一个简单快速的三段式编译器，主要的关注点在于局部性的优化，而放弃了许多耗时较长的全局优化手段。

在第一个阶段，一个平台独立的前端将字节码构造成一种高级中间代码表示 (High-Level Intermediate Representation, HIR)。HIR 使用静态单分配 (Static Single

Assignment, SSA) 的形式来代表代码值, 这可以使得一些在 HIR 的构造过程之中中和之后进行的优化动作更容易实现。在此之前编译器会在字节码上完成一部分基础优化, 如方法内联、常量传播等优化将会在字节码被构造成为 HIR 之前完成。

在第二个阶段, 一个平台相关的后端从 HIR 中产生低级中间代码表示 (Low-Level Intermediate Representation, LIR), 而在此之前会在 HIR 上完成另外一些优化, 如空值检查消除、范围检查消除等, 以便让 HIR 达到更高效的代码表示形式。

最后的阶段是在平台相关的后端使用线性扫描算法 (Linear Scan Register Allocation) 在 LIR 上分配寄存器, 并在 LIR 上做窥孔 (Peephole) 优化, 然后产生机器代码。Client Compiler 的大致执行过程如图 11-4 所示。

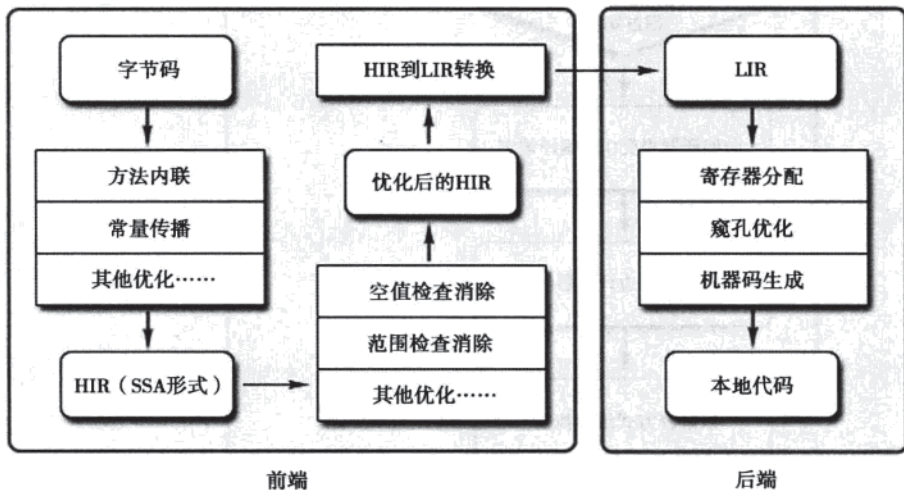


图 11-4 Client Compiler 架构

而 Server Compiler 则是专门面向服务端的典型应用并为服务端的性能配置特别调整过的编译器, 也是一个充分优化过的高级编译器, 几乎能达到 GNU C++ 编译器使用 -O2 参数时的优化强度, 它会执行所有的经典的优化动作, 如: 无用代码消除 (Dead Code Elimination)、循环展开 (Loop Unrolling)、循环表达式外提 (Loop Expression Hoisting)、公共子表达式消除 (Common Subexpression Elimination)、常量传播 (Constant Propagation)、基本块重排序 (Basic Block Reordering) 等, 还会实施一些与 Java 语言特性密切相关的优化技术, 如范围检查消除 (Range Check Elimination)、空值检查消除 (Null Check Elimination, 不过并非所有的空值检查消除都是依赖编译

器进行优化的，有一些是在代码运行过程中自动优化了）等。另外，还可能根据解释器或 Client Compiler 提供的性能监控信息，进行一些不稳定的激进优化，如守护内联（Guarded Inlining）、分支频率预测（Branch Frequency Prediction）等，本章的下半部分将会挑选上述的一部分优化手段进行分析讲解。

Server Compiler 的寄存器分配器是一个全局图着色分配器，它可以充分利用某些处理器架构（如 RISC）上的大寄存器集合。以即时编译的标准来看，Server Compiler 无疑是比较缓慢的，但它的速度仍然远远超过传统的静态优化编译器，而且它相对于 Client Compiler 编译输出的代码质量有所提高，可以减少本地代码的执行时间，从而抵消了额外的编译时间开销，所以也有很多非服务端的应用选择使用 Server 模式的虚拟机运行。

11.2.4 查看与分析即时编译结果

一般来说，虚拟机的即时编译过程对用户程序是完全透明的，虚拟机通过解释执行代码还是编译执行代码，对于用户来说并没有什么影响（执行结果没有影响，速度上会有很大差别），大多数情况下用户也没有必要知道。但是虚拟机也提供了一些参数用来输出即时编译和某些优化手段（如方法内联）的执行状况，本节将介绍如何从外部观察虚拟机的即时编译行为。

本节中提到的运行参数有一部分需要 Debug 或 FastDebug 版虚拟机的支持，Product 版的虚拟机无法使用这部分参数。如果读者使用的是根据本书第一章的教程自己编译的 JDK，请注意将 SKIP_DEBUG_BUILD 或 SKIP_FASTDEBUG_BUILD 参数设置为 false，也可以在 OpenJDK 网站上直接下载 FastDebug 版的 JDK。本节中所有的测试都基于代码清单 11-2 所示的 Java 代码。

代码清单 11-2 测试代码

```
public static final int NUM = 15000;

public static int doubleValue(int i) {
    return i * 2;
}

public static long calcSum() {
    long sum = 0;
```


298 ❖ 第四部分 程序编译与代码优化

```

    for (int i = 1; i <= 100; i++) {
        sum += doubleValue(i);
    }
    return sum;
}

public static void main(String[] args) {
    for (int i = 0; i < NUM; i++) {
        calcSum();
    }
}

```

首先运行这段代码，并且确认这段代码是否触发了即时编译，要知道某个方法是否被编译过，可以使用参数 `-XX:+PrintCompilation` 要求虚拟机在即时编译时将编译成本地代码的方法名称打印出来，如代码清单 11-3 所示（其中带有“%”的输出说明是由回边计数器触发的 OSR 编译）。

代码清单 11-3 被即时编译的代码

```

VM option '+PrintCompilation'
310 1      java.lang.String::charAt (33 bytes)
329 2      org.fenixsoft.jit.Test::calcSum (26 bytes)
329 3      org.fenixsoft.jit.Test::doubleValue (4 bytes)
332 1%     org.fenixsoft.jit.Test::main @ 5 (20 bytes)

```

从代码清单 11-3 输出的确认信息中可以确认 `main()`、`calcSum()` 和 `doubleValue()` 方法已经被编译，我们还可以加上参数 `-XX:+PrintInlining` 要求虚拟机输出方法内联信息，如代码清单 11-4 所示。

代码清单 11-4 内联信息

```

VM option '+PrintCompilation'
VM option '+PrintInlining'
273 1      java.lang.String::charAt (33 bytes)
291 2      org.fenixsoft.jit.Test::calcSum (26 bytes)
    @ 9     org.fenixsoft.jit.Test::doubleValue inline (hot)
294 3      org.fenixsoft.jit.Test::doubleValue (4 bytes)
295 1%     org.fenixsoft.jit.Test::main @ 5 (20 bytes)
    @ 5     org.fenixsoft.jit.Test::calcSum inline (hot)
    @ 9     org.fenixsoft.jit.Test::doubleValue inline (hot)

```

从代码清单 11-4 的输出中可以看到方法 `doubleValue()` 被内联编译到 `calcSum()` 中，而 `calcSum()` 又被内联编译到方法 `main()` 里面，所以虚拟机再次执行 `main()` 方法的时候（尽管 `main()` 方法并不会运行两次），`calcSum()` 和 `doubleValue()` 方法都不会再被调用，它们的代码逻辑都被直接内联到 `main()` 方法里面了。

除了查看哪些方法被编译之外，还可以进一步查看即时编译器生成的机器码内容，不过如果虚拟机输出一串 0 和 1，对于我们的阅读来说是没有意义的，机器码必须反汇编成基本的汇编语言才可能被阅读。虚拟机提供了一组通用的反汇编接口^①，可以接入各种平台下的反汇编适配器来使用，如使用 32 位 x86 平台则选用 `hstdis-i386` 适配器^②，其余平台的适配器还有如 `hstdis-amd64`、`hstdis-sparc` 和 `hstdis-sparcv9` 等，可以下载或自己编译出反汇编适配器后，将其放置在 `JRE/bin/client` 或 `/servier` 目录下，只要与 `jvm.dll` 的路径相同即可被虚拟机调用。为虚拟机安装了反汇编适配器之后，就可以使用 `-XX:+PrintAssembly` 参数要求虚拟机打印编译方法的汇编代码了如果没有 `hstdis` 支持，也可以使用 `-XX:printOptoAssembly`（用于 Server VM）或 `-XX:+print LIR`（用于 Client VM）来输出比较接近最终结果的中间代码表示，代码清单 11-2 被编译后部分反汇编（使用 `-XX: +printOptoAssembly`）的输出结果如代码清单 11-5 所示。

代码清单 11-5 本地机器码反汇编信息（部分）

```

.....
000  B1: #      N1 <- BLOCK HEAD IS JUNK   Freq: 1
000      pushq  rbp
          subq   rsp, #16          # Create frame
          nop   # nop for patch_verified_entry
006      movl   RAX, RDX          # spill
008      sall   RAX, #1
00a      addq   rsp, 16 # Destroy frame
          popq   rbp
          testl  rax, [rip + #offset_to_poll_page] # Safepoint: poll for GC
.....

```

① 相关信息：<http://wikis.sun.com/display/HotSpotInternals/PrintAssembly>。

② `hstdis` 的源码可以从这里获取：<http://hg.openjdk.java.net/jdk7/hotspot/hotspot/file/tip/src/share/tools/hstdis/>。另外，以下地址可以下载到一个已经编译好了的适合 32 位 x86 平台使用的反汇编适配器（此地址可能需要海外代理才能访问）：<http://classparser.blogspot.com/2010/03/hstdis-i386dll.html>。此适配器在 ITEye 的高级语言虚拟机圈子的共享区（<http://hllvm.group.iteye.com/group/share>）中也有备份可以下载。

300 ❖ 第四部分 程序编译与代码优化

如果除了本地代码的生成结果外，还想再进一步跟踪本地代码生成的具体过程，那还可以使用参数 `-XX:+PrintCFGToFile`（使用 Client Compiler）或 `-XX:PrintIdealGraphFile`（使用 Server Compiler）令虚拟机将编译过程中各个阶段的数据（如对 C1 编译器来说包括：字节码、HIR 生成、LIR 生成、寄存器分配过程、本地代码生成等数据）输出到文件中。然后使用 Java HotSpot Client Compiler Visualizer^①（使用 Client Compiler）或 Ideal Graph Visualizer^②（使用 Server Compiler）打开这些数据文件进行分析，如图 11-5 所示。

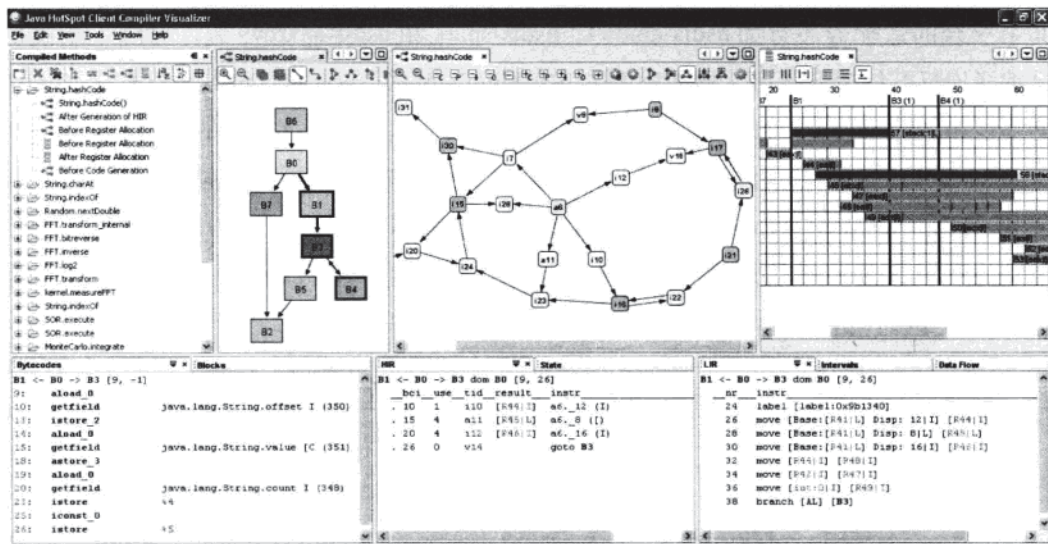


图 11-5 Java HotSpot Client Compiler Visualizer 界面

实际使用的时候请注意，要输出 CFG 或 IdealGraph 文件，需要一个 Debug 版的虚拟机支持，Product 版或 FastDebug 版的虚拟机无法输出这些文件。前面提到的使用 `-XX:+PrintAssembly` 参数输出反汇编信息也需要 FastDebug 版的虚拟机才能直接支持，如果使用 Product 版的虚拟机，则需要加入参数 `-XX:+UnlockDiagnosticVMOptions` 打开虚拟机诊断模式后才能使用。

① 官方网站：<http://java.net/projects/c1visualizer/>。

② 官方网站：<http://sww.jku.at/General/Staff/TW/igv.html>。

11.3 编译优化技术

Java 程序员都有一个共同的认知，以编译方式执行本地代码比解释方式更快，其中除去虚拟机解释执行字节码时额外消耗的时间以外，还有一个很重要的原因就是 JDK 设计团队几乎把对代码的所有优化措施都集中在了即时编译器之中，所以一般来说即时编译器产生的本地代码会比 Javac 产生的字节码更优秀^①，接下来我们就介绍一些 HotSpot 虚拟机的即时编译器在生成代码时采用的代码优化技术。

11.3.1 优化技术概览

Sun 官方的 Wiki 上，HotSpot 虚拟机设计团队列出了一个相对比较全面的、即时编译器中采用的优化技术列表^②（如表 11-1 所示），其中有不少经典编译器的优化手段，也有许多针对 Java 语言（准确地说是针对运行在 Java 虚拟机上的所有语言）本身进行的优化技术，本节将对这些技术进行一遍简单的概览，在后面的几节中，笔者将挑选若干最重要或最典型的优化，与读者一起看看优化前后的代码发生了怎样的变化。

表 11-1 即时编译器优化技术一览

类 型	优化技术
编译器策略 (compiler tactics)	延迟编译 (delayed compilation)
	分层编译 (tiered compilation)
	栈上替换 (on-stack replacement)
	延迟优化 (delayed reoptimization)
	程序依赖图表示 (program dependence graph representation)
	静态单赋值表示 (static single assignment representation)
基于性能监控的优化技术 (profile-based techniques)	乐观空值断言 (optimistic nullness assertions)
	乐观类型断言 (optimistic type assertions)
	乐观类型增强 (optimistic type strengthening)
	乐观数组长度增强 (optimistic array length strengthening)
	裁剪未被选择的分支 (untaken branch pruning)
	乐观的多态内联 (optimistic N-morphic inlining)
	分支频率预测 (branch frequency prediction)
调用频率预测 (call frequency prediction)	

① 本地代码与字节码两者是无法直接比较的，准确地说应当是指：由编译器优化得到的本地代码与由解释器解释字节码后实际执行的本地代码之间的对比。

② 地址：<http://wikis.sun.com/display/HotSpotInternals/PerformanceTacticIndex>。

(续)

类 型	优化技术
基于证据的优化技术 (proof-based techniques)	精确类型推断 (exact type inference)
	内存值推断 (memory value inference)
	内存值跟踪 (memory value tracking)
	常量折叠 (constant folding)
	重组 (reassociation)
	操作符退化 (operator strength reduction)
	空值检查消除 (null check elimination)
	类型检测退化 (type test strength reduction)
	类型检测消除 (type test elimination)
	代数化简 (algebraic simplification)
	公共子表达式消除 (common subexpression elimination)
数据流敏感重写 (flow-sensitive rewrites)	条件常量传播 (conditional constant propagation)
	基于流承载的类型缩减转换 (flow-carried type narrowing)
	无用代码消除 (dead code elimination)
语言相关的优化技术 (language-specific techniques)	类型继承关系分析 (class hierarchy analysis)
	去虚拟机化 (devirtualization)
	符号常量传播 (symbolic constant propagation)
	自动装箱消除 (autobox elimination)
	逃逸分析 (escape analysis)
	锁消除 (lock elision)
	锁膨胀 (lock coarsening)
	消除反射 (de-reflection)
内存及代码位置变换 (memory and placement transformation)	表达式提升 (expression hoisting)
	表达式下沉 (expression sinking)
	冗余存储消除 (redundant store elimination)
	相邻存储合并 (adjacent store fusion)
	交汇点分离 (merge-point splitting)
循环变换 (loop transformations)	循环展开 (loop unrolling)
	循环剥离 (loop peeling)
	安全点消除 (safepoint elimination)
	迭代范围分离 (iteration range splitting)
	范围检查消除 (range check elimination)
	循环向量化 (loop vectorization)
全局代码调整 (global code shaping)	内联 (inlining)
	全局代码外提 (global code motion)
	基于热度的代码布局 (heat-based code layout)
	Switch 调整 (switch balancing)

(续)

类 型	优化技术
控制流图变换 (control flow graph transformation)	本地代码编排 (local code scheduling)
	本地代码封包 (local code bundling)
	延迟槽填充 (delay slot filling)
	着色图寄存器分配 (graph-coloring register allocation)
	线性扫描寄存器分配 (linear scan register allocation)
	复写聚合 (copy coalescing)
	常量分裂 (constant splitting)
	复写移除 (copy removal)
	地址模式匹配 (address mode matching)
	指令窥孔优化 (instruction peepholing)
	基于确定有限状态机的代码生成 (DFA-based code generator)

上述的优化技术看起来很多，而且名字看起来都显得有点“高深莫测”，实际上实现这些优化也许确实有些难度，但大部分技术理解起来都并不困难，笔者举一个最简单的例子来展示其中几种优化技术是如何发挥作用的。首先从原始代码开始，如代码清单 11-6 所示^①。

代码清单 11-6 优化前的原始代码

```

static class B {
    int value;
    final int get() {
        return value;
    }
}

public void foo() {
    y = b.get();
    // ...do stuff...
    z = b.get();
    sum = y + z;
}

```

首先需要明确一点的是，这些代码的优化变换都是建立在代码的某种中间表示上，

^① 本例修改自：http://download.oracle.com/docs/cd/E13150_01/jrocket_jvm/jrocket/geninfo/diagnos/underst_jit.html。

绝不是建立在 Java 源码之上的，笔者为了展示方便，使用了 Java 语言的语法来表示这些优化技术所发挥的作用。

代码清单 11-6 的代码已经非常简单了，但是仍有许多优化的余地。首先进行方法内联 (Method Inlining)，内联的主要目的有两个，一是去除方法调用的成本（如建立栈帧等），二是为其他优化建立良好的基础。方法内联膨胀之后可以便于在更大范围上进行后续的优化手段，可以获得更好的优化效果，因此各种编译器一般都会把内联优化放在优化序列的靠前位置。内联后的代码如代码清单 11-7 所示。

代码清单 11-7 内联后的代码

```
public void foo() {
    y = b.value;
    // ...do stuff...
    z = b.value;
    sum = y + z;
}
```

下一步进行冗余访问消除 (Redundant Loads Elimination)，假设代码中间注释掉的“do stuff...”所代表的操作不会改变 b.value 的值，那就可以把“z = b.value”替换为“z = y”，因为上一句“y = b.value”已经保证了变量 y 与 b.value 是一致的，这样就可以不用再去访问对象 b 的局部变量了。如果把 b.value 看做是一个表达式，那也可以把这项优化看成是公共子表达式消除 (Common Subexpression Elimination)，优化后的代码如代码清单 11-8 所示。

代码清单 11-8 冗余存储消除的代码

```
public void foo() {
    y = b.value;
    // ...do stuff...
    z = y;
    sum = y + z;
}
```

第三步我们进行复写传播 (Copy Propagation)，因为这段程序的逻辑之中并没有必要使用一个额外的变量“z”，它与变量“y”是完全相等的，因此我们可以使用“y”来代替“z”。复写传播之后程序如代码清单 11-9 所示。

代码清单 11-9 复写传播的代码

```
public void foo() {  
    y = b.value;  
    // ...do stuff...  
    y = y;  
    sum = y + y;  
}
```

第四步我们再进行无用代码消除（Dead Code Elimination），无用代码可能是永远不会被执行的代码，也可能是完全没有意义的代码。它又被很形象地称为“Dead Code”，在代码清单 11-9 中，“y = y”是没有意义的，把它消除后的程序如代码清单 11-10 所示。

代码清单 11-10 进行无用代码消除的代码

```
public void foo() {  
    y = b.value;  
    // ...do stuff...  
    sum = y + y;  
}
```

经过四次优化之后，代码清单 11-10 与代码清单 11-6 所达到的效果是一致的，但是前者比后者省略了许多语句（体现在字节码和机器码指令上的差距会更大），执行效率也更高。编译器的这些优化技术实现起来也许比较复杂，但是要理解它们的行为对于一个普通的程序员来说是没有困难的，接下来我们继续查看如下的几项优化技术是如何运作的，它们分别是：

- 语言无关的经典优化技术之一：公共子表达式消除
- 语言相关的经典优化技术之一：数组范围检查消除
- 最重要的优化技术之一：方法内联
- 最前沿的优化技术之一：逃逸分析

11.3.2 公共子表达式消除

公共子表达式消除是一个普遍应用于各种编译器的经典优化技术，它的含义是：如果一个表达式 E 已经被计算过了，并且从先前的计算到现在 E 中所有变量的值都没有发生变化，那么 E 的这次出现就成为了公共子表达式。对于这种表达式，没有必要花时间再对它进行计算，只需要直接用前面计算过的表达式结果代替 E 就可以了。如

306 ❖ 第四部分 程序编译与代码优化

果这种优化仅限于程序的基本块内，便称为局部公共子表达式消除 (Local Common Subexpression Elimination)，如果这种优化的范围涵盖了多个基本块，那就称为全局公共子表达式消除 (Global Common Subexpression Elimination)。举个简单的例子来说明它的优化过程，假设存在如下代码：

```
int d = (c * b) * 12 + a + (a + b * c);
```

如果这段代码交给 Javac 编译器则不会进行任何优化，那生成的代码将如代码清单 11-11 所示，是完全遵照 Java 源码的写法直译而成的。

代码清单 11-11 未作任何优化的字节码

```

iload_2      // b
imul        // 计算 b*c
bipush 12   // 推入 12
imul        // 计算 (c * b) * 12
iload_1     // a
iadd       // 计算 (c * b) * 12 + a
iload_1     // a
iload_2     // b
iload_3     // c
imul        // 计算 b * c
iadd       // 计算 a + b * c
iadd       // 计算 (c * b) * 12 + a + a + b * c
istore 4

```

当这段代码进入到虚拟机即时编译器后，它将进行如下优化：编译器检测到“ $c * b$ ”与“ $b * c$ ”是一样的表达式，而且在计算期间 b 与 c 的值是不变的。因此这条表达式就可能被视为：

```
int d = E * 12 + a + (a + E);
```

这时候，编译器还可能（取决于哪种虚拟机的编译器及具体的上下文）进行另外一种优化：代数化简 (Algebraic Simplification)，把表达式变为：

```
int d = E * 13 + a * 2;
```

表达式进行变换之后，再计算起来就可以节省一些时间了。如果您还对其他的经典编译优化技术感兴趣，可以参考《编译原理》（俗称龙书，推荐使用 Java 的程序员看

2006 年版的紫龙书）中的相关章节。

11.3.3 数组边界检查消除

数组边界检查消除（Array Bounds Checking Elimination）是即时编译器中的一项语言相关的经典优化技术。我们知道 Java 语言是一门动态安全的语言，对数组的读写访问也不像 C、C++ 那样本质上是裸指针操作。如果有一个数组 `foo[]`，在 Java 语言中访问数组元素 `foo[i]` 的时候系统将会自动进行上下界的范围检查，即检查 `i` 必须满足 `i >= 0 && i < foo.length` 的这个条件，否则将抛出一个运行时异常：`java.lang.ArrayIndexOutOfBoundsException`。这对软件开发者来说是一件很好的事情，即使程序员没有专门编写防御代码，也可以避免大部分的溢出攻击。但是对于虚拟机的执行子系统来说，每次数组元素的读写都带有一次隐含的条件判定操作，对于拥有大量数组访问的程序代码，这无疑也是一种性能负担。

无论如何，为了安全，数组边界检查肯定是必须做的，但数组边界检查是不是必须在运行期间一次不漏地检查则是可以“商量”的事情。例如这个简单的情况：数组下标是一个常量，如 `foo[3]`，只要在编译期根据数据流分析来确定 `foo.length` 的值，并判断下标“3”没有越界，执行的时候就无须判断了。更加常见的情况是数组访问发生在循环之中，并且使用循环变量来进行的数组访问，如果编译器只要通过数据流分析就可以判定循环变量的取值范围永远在区间 $[0, \text{foo.length})$ 之内，那在整个循环中就可以把数组的上下界检查消除掉，这可以节省很多次的条件判断操作。

与语言相关的其他消除操作还有不少，如自动装箱消除（Autobox Elimination）、安全点消除（Safepoint Elimination）、消除反射（Dereflection）等，笔者就不再一一介绍了。

11.3.4 方法内联

在前面的讲解中我们提到过方法内联，它是编译器最重要的优化手段之一，除了消除方法调用的成本之外，它更重要的意义是为其他优化手段建立良好的基础，如下面代码清单 11-12 所示的简单例子就揭示了内联对其他优化手段的意义：事实上 `testInline()` 方法的内部全部都是无用的代码，如果不做内联，后续即使进行了无用代码消除的优化，也无法发现任何“Dead Code”，因为如果分开来看，`foo()` 和 `testInline()` 两个方法里

面的操作都可能是有意义的。

代码清单 11-12 未作任何优化的字节码

```
public static void foo(Object obj) {
    if (obj != null) {
        System.out.println("do something");
    }
}

public static void testInline(String[] args) {
    Object obj = null;
    foo(obj);
}
```

方法内联的优化行为看起来很简单，不过是把目标方法的代码“复制”到发起调用的方法之中，避免发生真实的方法调用而已。但实际上 Java 虚拟机中的内联过程远远没有那么简单，因为如果不是即时编译器做了一些特别的努力，按照经典编译原理的优化理论，大多数的 Java 方法都无法进行内联！

无法内联的原因其实在前面第 8 章中讲解 Java 方法解析和分派调用的时候就已经说过。只有使用 `invokespecial` 指令调用的私有方法、实例构造器、父类方法和使用 `invokestatic` 指令进行调用的静态方法才是在编译期进行解析的，除了上述四种方法之外，其他的 Java 方法调用都需要在运行时进行方法接收者的多态选择，并且都有可能存在多于一个版本的方法接收者（最多再除去被 `final` 修饰的方法这种特殊情况，尽管它使用 `invokevirtual` 指令调用，但也是非虚方法，Java 语言规范中明确说明了这点），简而言之，Java 语言中默认的实例方法就是虚方法。

对于一个虚方法，编译器做内联的时候根本就无法确定应该使用哪个方法版本，例如，前面代码清单 11-7 中的把 `b.get()` 内联为 `b.value`，就是不依赖上下文就无法确定 `b` 的实际类型是什么。假如有 `ParentB` 和 `SubB` 两个具有继承关系的类，并且子类重写了父类的 `get()` 方法，那么，是要执行父类的 `get()` 方法还是子类的 `get()` 方法，需要在运行期才能确定，编译期无法得出结论。

由于 Java 语言提倡使用面向对象的编程方式进行编程，而 Java 对象的方法默认就是虚方法，因此 Java 间接鼓励了程序员使用大量的虚方法来完成程序逻辑。根据我们上面的分析，内联与虚方法之间会产生“矛盾”，那该怎么办呢？是不是为了提高执行性

能，就要到处使用 final 关键字去修饰方法呢？

为了解决虚方法的内联问题，Java 虚拟机设计团队想了很多办法，首先是引入了一种名为“类型继承关系分析”（Class Hierarchy Analysis, CHA）的技术，这是一种基于整个应用程序的类型分析技术，它用于确定在目前已加载的类中，某个接口是否有多于一种的实现，某个类是否存在子类且子类是否为抽象类等信息。

编译器在进行内联时，如果是非虚方法，那么直接进行内联就可以了，这时候的内联是有稳定前提保障的。如果遇到虚方法，则会向 CHA 查询此方法在当前程序下是否有多个目标版本可供选择，如果查询结果只有一个版本，那也可以进行内联，不过这种内联就属于激进优化，需要预留一个“逃生门”（Guard 条件不成立时的 Slow Path），称为守护内联（Guarded Inlining）。如果程序的后续执行过程中，虚拟机一直没有加载到会令这个方法的接收者的继承关系发生变化的类，那这个内联优化的代码就可以一直使用下去。但是如果加载了导致继承关系发生变化的新类，那就需要抛弃掉已经编译的代码，退回到解释状态执行，或者重新进行编译。

如果向 CHA 查询出来的结果是有多个版本的目标方法可供选择，则编译器还将会进行最后一次努力，使用内联缓存（Inline Cache）来完成方法内联，这是一个建立在目标方法正常入口之前的缓存，它的工作原理大致是：在未发生方法调用之前，内联缓存状态为空，当第一次调用发生后，缓存记录下方法接收者的版本信息，并且，每次进行方法调用时都比较接收者版本，如果以后进来的每次调用的方法接收者版本都是一样的，那这个内联还可以一直用下去。如果发生了方法接收者不一致的情况，就说明程序真正使用到了虚方法的多态特性，这时候才会取消内联，查找虚方法表进行方法分派。

所以说，在许多情况下虚拟机进行的内联都是一种激进优化，激进优化的手段在高性能的商用虚拟机中很常见，除了内联之外，对于出现概率很小（通过经验数据或解释器收集到的性能监控信息确定概率大小）的隐式异常、使用概率很小的分支等都可以被激进优化“移除”掉，如果真的出现了小概率事件，这时才会从“逃生门”回到解释状态重新执行。

11.3.5 逃逸分析

逃逸分析（Escape Analysis）是目前 Java 虚拟机中比较前沿的优化技术，它与类型继承关系分析一样，并不是直接优化代码的手段，而是为其他优化手段提供依据的分析

技术。

逃逸分析的基本行为就是分析对象动态作用域：当一个对象在方法里面被定义后，它可能被外部方法所引用，例如作为调用参数传递到其他方法中，这种行为称为方法逃逸。甚至还有可能被外部线程访问到，譬如赋值给类变量或可以在其他线程中访问的实例变量，这种行为称为线程逃逸。

如果能证明一个对象不会逃逸到方法或线程之外，也就是别的方法或线程无法通过任何途径访问到这个对象，则可能为这个变量进行一些高效的优化，如：

- 栈上分配 (Stack Allocations)：Java 虚拟机中，在 Java 堆上分配创建对象的内存空间几乎是 Java 程序员都清楚的常识了，Java 堆中的对象对于各个线程都是共享和可见的，只要持有这个对象的引用，就可以访问堆中存储的对象数据。虚拟机的垃圾收集系统可以回收掉堆中不再使用的对象，但回收动作无论是筛选可回收对象，还是回收和整理内存都需要耗费时间。如果确定一个对象不会逃逸出方法之外，那让这个对象在栈上分配内存将会是一个很不错的主意，对象所占用的内存空间就可以随栈帧出栈而销毁。在一般应用中，不会逃逸的局部对象所占的比率很大，如果能使用栈上分配，那大量的对象就会随着方法的结束而自动销毁了，垃圾收集系统的压力将会小很多。
- 同步消除 (Synchronization Elimination)：线程同步本身就是一个相对耗时的过程，如果逃逸分析能够确定一个变量不会逃逸出线程，无法被其他线程访问，那这个变量的读写肯定就不会有竞争，对这个变量实施的同步措施也就可以消除掉。
- 标量替换 (Scalar Replacement)：标量 (Scalar) 是指一个数据已经无法再分解成更小的数据来表示了，Java 虚拟机中的原始数据类型 (int、long 等数值类型及 reference 类型等) 都不能再进一步分解，它们就可以被称为标量。相对的，如果一个数据可以继续分解，那它就被称做聚合量 (Aggregate)，Java 中的对象就是最典型的聚合量。如果把一个 Java 对象拆散，根据程序访问的情况，将其使用到的成员变量恢复原始类型来访问就叫做标量替换。如果逃逸分析证明一个对象不会被外部访问，并且这个对象可以被拆散的话，那程序真正执行的时候将可能不创建这个对象，而改为直接创建它的若干个被这个方法使用到的成员变量来代替。将对象拆分后，除了可以让对象的成员变量在栈上 (栈上存储的数据，很大

机会会被虚拟机分配至物理机器的高速寄存器中存储）分配和读写之外，还可以为后续进一步的优化手段创建条件。

逃逸分析在 Sun JDK 1.6 中实现，但是现在这项优化尚未成熟，仍有巨大的改进余地。不成熟的原因主要是不能保证逃逸分析的性能收益必定高于它的消耗。如果要百分之百准确地判断一个对象是否会逃逸，需要进行数据流敏感的复杂分析，来确定程序各个分支执行时对此对象的影响。这是一个相对高耗时的过程，如果分析完后发现没有几个不逃逸的对象，那时间就白白浪费了，所以目前虚拟机只能采用不那么准确，但时间压力相对较小的算法来完成逃逸分析。还有一点是基于逃逸分析的一些优化手段，如上面提到的“栈上分配”，由于 HotSpot 虚拟机目前的实现方式导致栈上分配实现起来比较复杂，因此在 HotSpot 中暂时还没有做这项优化。

在测试结果上，实施逃逸分析后的程序在 MicroBenchmarks 中往往能运行出不错的成绩，但是在实际的应用程序，尤其是大型程序中，反而发现实施逃逸分析可能会出现效果不稳定的情况，或因分析过程耗时却无法有效地判别出非逃逸对象而导致性能（即时编译的收益）有所下降，所以即使是 Server Compiler，也默认不开启逃逸分析^①，甚至在某些版本（如 JDK 1.6 Update 18）中还曾经短暂地完全禁止了这项优化。

如果有需要，并且确认对程序运行有益，用户可以使用参数 `-XX:+DoEscapeAnalysis` 来手动开启逃逸分析，开启之后可以通过参数 `-XX:+PrintEscapeAnalysis` 来查看分析结果。另外，用户可以使用参数 `-XX:+EliminateAllocations` 来开启标量替换，使用参数 `+XX:+EliminateLocks` 来开启同步消除，使用参数 `-XX:+PrintEliminateAllocations` 来查看标量的替换情况。

尽管目前逃逸分析的技术仍未完全成熟，它却是即时编译器优化技术的一个重要的发展方向，在日后的虚拟机中，逃逸分析技术肯定会支撑起一系列实用有效的优化技术。

11.4 Java 与 C/C++ 的编译器对比

大多数程序员都认为 C/C++ 会比 Java 语言快，甚至觉得从 Java 语言诞生以来，“执行速度缓慢”的帽子就应当被扣在头顶，这种观点的出现是由于 Java 刚出现的时候即时

^① 初稿完成之前，在最新的 JDK 1.6 Update 23 的 Server Compiler 中已默认开启了逃逸分析。

编译技术还不成熟，主要靠解释器执行的 Java 语言性能确实比较低下。但是在今天即时编译技术已经发展成熟，Java 语言有可能在速度上与 C/C++ 一争高下吗？要想知道这个问题的答案得从两者的编译器谈起^①。

Java 与 C/C++ 的编译器对比实际上代表了最经典的即时编译器与静态编译器的对比，很大程度上也决定了 Java 与 C/C++ 的性能对比的结果，因为无论是 C/C++ 还是 Java 代码，最终编译之后被机器执行的都是本地机器码，哪种语言的性能更高，除了它们自身的 API 库实现得好坏以外，其余的比较就成了一场“拼编译器”和“拼输出代码质量”的游戏。当然，这种比较也是剔除了开发效率的片面对比，语言间孰优孰劣、谁快谁慢的问题都是很难有结果的争论，下面我们就回到正题，看看这两种语言的编译器各有何种优势。

Java 虚拟机的即时编译器与 C/C++ 的静态优化编译器相比，可能会由于下列这些原因导致输出的本地代码有一些劣势（下面列举的也包括一些虚拟机执行子系统的性能劣势）：

首先，因为即时编译器运行占用的是用户程序的运行时间，具有很大的时间压力，它能提供的优化手段也严重受制于编译成本。如果编译速度不能达到要求，那用户将在启动程序或程序的某部分察觉到重大延迟，这点使得即时编译器不敢随便引入大规模的优化技术，而编译的时间成本在静态优化编译器中并不是主要的关注点。

其次，Java 语言是动态的类型安全语言，这就意味着需要由虚拟机来确保程序不会违反语言的语义或访问非结构化内存。在实现层面上看，这就意味着虚拟机必须频繁地进行动态检查，如实例方法访问时检查空指针、数组元素访问时检查上下界范围、类型转换时检查继承关系，等等。对于这类程序代码没有明确写出的检查行为，尽管编译器会努力进行优化，但是总体上仍然要消耗不少的运行时间。

第三，Java 语言中虽然没有 virtual 关键字，但是使用虚方法的频率却远远大于 C/C++ 语言，这就意味着运行时对方法接收者进行多态选择的频率要远远大于 C/C++ 语言，也意味着即时编译器在进行一些优化（如前面提到的方法内联）时的难度要远远大于 C/C++ 的静态优化编译器。

^① C/C++ 与 Java 孰优孰劣、谁快谁慢这类话题已经争论了十几年，双方的支持者们从来都没有说服过对方，有朋友好意提醒过笔者不要跳入这种语言性能争论的“火坑”，把这节移除掉。笔者在此也特别说明，本节的目的仅是从编译和执行的角来探讨两者的差异，而不是去评判孰优孰劣。

第四，Java 语言是可以动态扩展的语言，运行时加载新的类可能改变程序类型的继承关系，这使得很多全局的优化都难以进行，因为编译器无法看见程序的全貌，许多全局的优化措施都只能以激进优化的方式来完成，编译器不得不时刻注意并随着类型的变化而在运行时撤销或重新进行一些优化。

第五，Java 语言中对象的内存分配都是在堆上进行的，只有方法中的局部变量才能在栈上分配^①。而 C/C++ 的对象则有多种内存分配方式，既可能在堆上分配，又可能在栈上分配，如果可以在栈上分配线程私有的对象，将减轻内存回收的压力。另外，C/C++ 中主要由用户程序代码来回收分配的内存，这就不存在无用对象筛选的过程，因此效率上（仅指运行效率，排除了开发效率）也比垃圾收集机制要高。

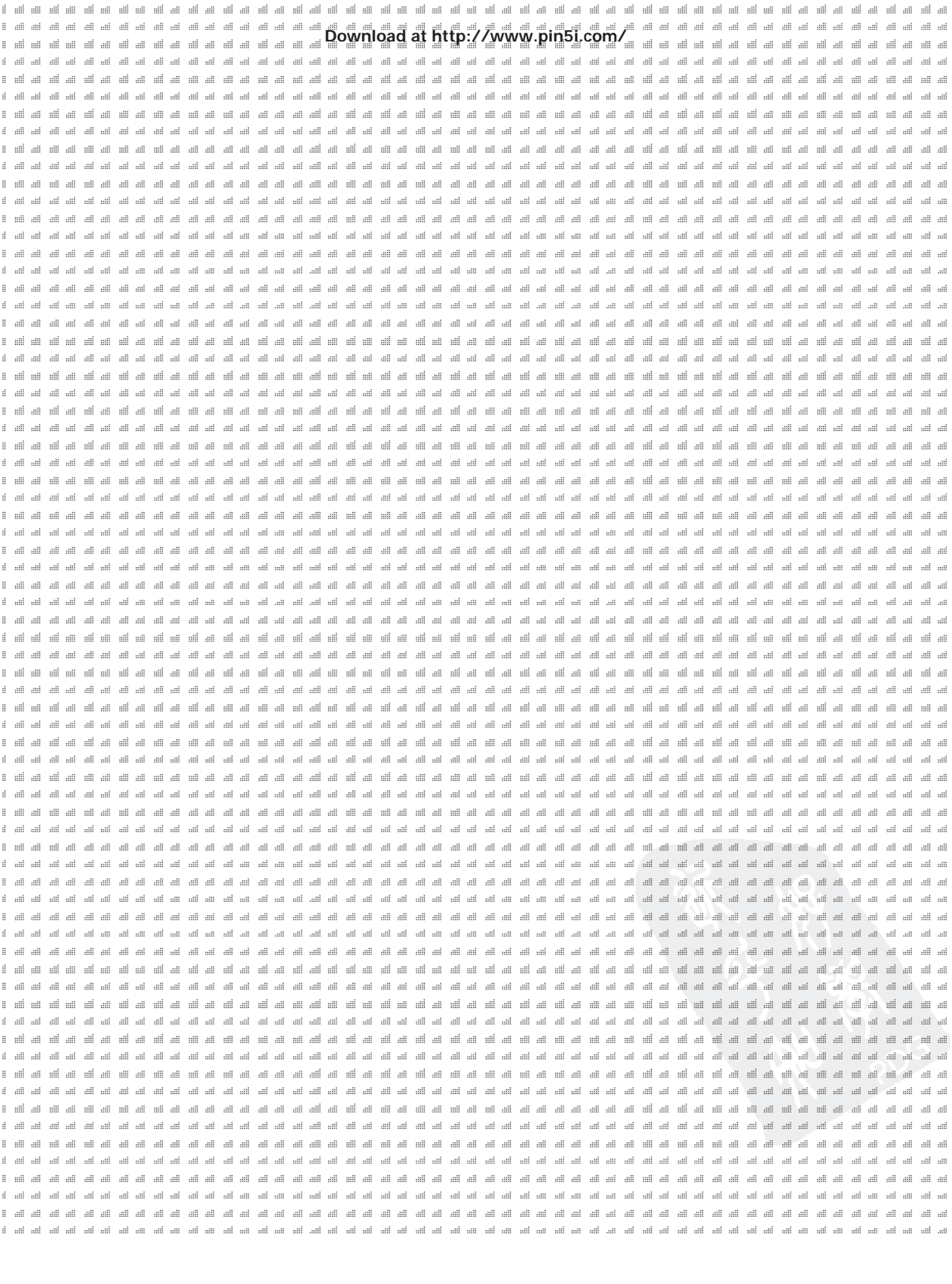
上面说了一大堆 Java 语言相对 C/C++ 的劣势，倒不是说 Java 就真的不如 C/C++ 了，相信读者也注意到了，Java 语言的这些性能上的劣势都是为了换取开发效率上的优势而付出的代价，动态安全、动态扩展、垃圾回收这些“拖后腿”的特性都为 Java 语言的开发效率做出了很大的贡献。何况，还有 Java 的即时编译器能做，而 C/C++ 的静态优化编译器不能做的优化：由于 C/C++ 编译器的静态特性，以运行期性能监控为基础的优化措施它都无法进行，如调用频率预测（Call Frequency Prediction）、分支频率预测（Branch Frequency Prediction）、裁剪未被选择的分支（Untaken Branch Pruning）等，这些都会成为 Java 语言独有的性能优势。

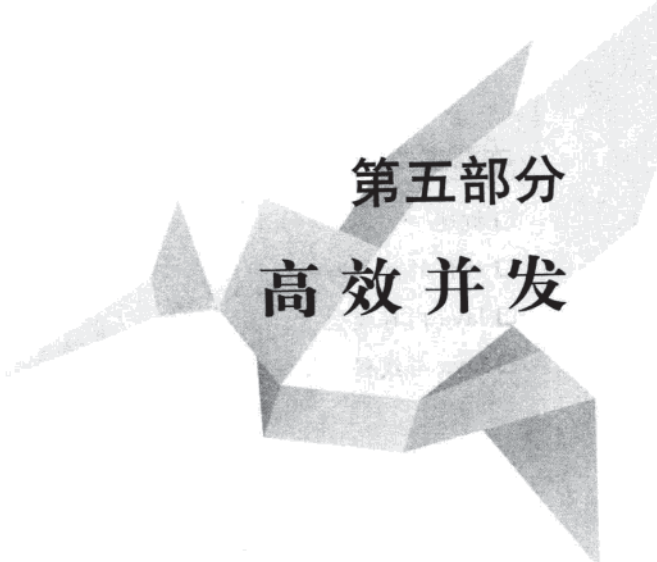
11.5 本章小结

第 10 章和第 11 章两章分别介绍了 Java 程序从源码编译成字节码和从字节码编译成本地机器码的过程，Javac 字节码编译器与虚拟机内的 JIT 编译器的执行过程合并起来其实就等同于一个传统的编译器所执行的编译过程。

本章中，我们着重了解了虚拟机的热点探测方法、HotSpot 的即时编译器、编译触发条件，以及如何从虚拟机外部观察和分析 JIT 编译的数据和结果，还选择了几种常见的编译期优化技术进行讲解，对 Java 编译器的深入了解，有助于在工作中分辨哪些代码是编译器可以帮我们处理的，哪些代码需要自己调节以便更适合编译器的优化。

^① Java 中非逃逸对象的标量替换优化可以看做是一种高度优化后的栈上分配，但它相当于把对象拆散成局部变量再进行的栈上分配，而不是 C/C++ 那种程序代码可控的栈上分配方式。





第五部分 高效并发

第 12 章 Java 内存模型与线程

第 13 章 线程安全与锁优化



资源 分享
PDG

第 12 章 Java 内存模型与线程

本章主要内容

- 概述
- 硬件的效率与一致性
- Java 内存模型
- Java 与线程

并发处理的广泛应用是使得 Amdahl 定律代替摩尔定律^①成为计算机性能发展源动力的根本原因，也是人类压榨计算机运算能力最有力的武器。

12.1 概述

多任务处理在现代计算机操作系统中几乎已是一项必备的功能了。在许多情况下，让计算机同时去做几件事情，不仅是因为计算机的运算能力强大了，还有一个很重要的原因是计算机的运算速度与它的存储和通讯子系统速度的差距太大，大部分时间都花在了磁盘 I/O、网络通讯和数据库访问上。如果不希望处理器在大部分时间里都处于等待其他资源的状态，就必须使用一些手段去把处理器的运算能力“压榨”出来，否则就会造成很大的“浪费”，而让计算机同时处理几项任务则是最容易想到、也被证明是非常有效的“压榨”手段。

除了充分利用计算机处理器的能力外，一个服务端同时对多个客户端提供服务则是另一个更具体的并发应用场景。衡量一个服务性能的高低好坏，每秒事务处理数

^① Amdahl 定律通过系统中并行化与串行化的比重来描述多处理器系统能获得的运算加速能力，摩尔定律则用于描述处理器晶体管数量与运行效率之间的发展关系。这两个定律的更替代表了近年来硬件发展从追求处理器频率到追求多核心并行处理的发展过程。

(Transactions Per Second, TPS) 是最重要的指标之一, 它代表着一秒内服务端平均能响应的请求总数, 而 TPS 值与程序的并发能力又有非常密切的关系。对于计算量相同的任务, 程序线程并发协调得越有条不紊, 效率自然就会越高; 反之, 线程之间频繁阻塞甚至死锁, 将会大大降低程序的并发能力。

服务端是 Java 语言最擅长的领域之一, 这个领域的应用占了 Java 应用中最大的一块份额^①, 不过如何写好并发应用程序却是程序开发的难点之一, 处理好并发方面的问题通常需要更多的经验。幸好 Java 语言和虚拟机提供了许多工具, 把并发编程的门槛降低了不少。另外, 各种中间件服务器、各类框架都努力地替程序员处理尽可能多的线程并发细节, 使得程序员在编码时能更关注业务逻辑, 而不是花费大部分时间去关注此服务会同时被多少人调用。但是无论语言、中间件和框架如何先进, 我们都不能期望它们能独立完成并发处理的所有事情, 了解并发的内幕也是成为一个高级程序员不可缺少的课程。

“高效并发”是本书讲解 Java 虚拟机的最后一部分, 将会向读者介绍虚拟机如何实现多线程、多线程之间由于共享和竞争数据而导致的一系列问题及解决方案。

12.2 硬件的效率与一致性

在正式讲解 Java 虚拟机并发相关的知识之前, 我们先花费一点时间去了解一下物理计算机中的并发问题, 物理机遇到的并发问题与虚拟机中的情况有不少相似之处, 物理机对并发的处理方案对虚拟机的实现也有相当大的参考意义。

“让计算机并发执行若干个运算任务”与“更充分地利用计算机处理器的效能”之间的因果关系, 看起来顺理成章, 实际上并没有想象中的那么容易实现, 因为所有的运算任务都不可能只靠处理器“计算”就能完成, 至少与内存的交互, 如读取运算数据、存储运算结果等, 就是很难消除的(不能仅仅靠寄存器来解决)。由于计算机的存储设备与处理器的运算速度之间有着几个数量级的差距, 所以现代计算机系统都不得不加入一层读写速度尽可能接近处理器运算速度的高速缓存(Cache)来作为内存与处理器之间的缓冲: 将运算需要使用到的数据复制到缓存中, 让运算能快速进行, 当运算结束后再从缓存同步回内存之中, 这样处理器就无须等待缓慢的内存读写了。

^① 必须以代码的总体规模来衡量, 服务端应用不能与 JavaCard、移动终端这些领域去比绝对数量。

基于高速缓存的存储交互很好地解决了处理器与内存的速度矛盾，但是也引入了新的问题：缓存一致性（Cache Coherence）。在多处理器系统中，每个处理器都有自己的高速缓存，而它们又共享同一主内存（Main Memory），如图 12-1 所示。当多个处理器的运算任务都涉及同一块主内存区域时，将可能导致各自的缓存数据不一致的情况，如果真的发生这种情况，那同步回到主内存时以谁的缓存数据为准呢？为了解决一致性的问题，需要各个处理器访问缓存时都遵循一些协议，在读写时要根据协议来进行操作，这类协议有 MSI、MESI（Illinois Protocol）、MOSI、Synapse、Firefly 及 Dragon Protocol，等等。Java 虚拟机内存模型中定义的内存访问操作与硬件的缓存访问操作是具有可比性的。

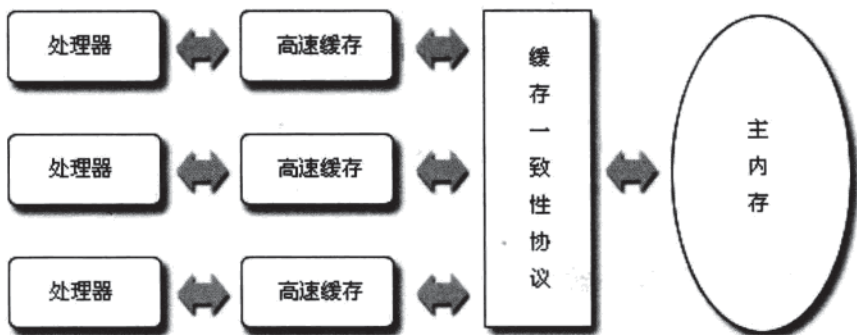


图 12-1 处理器、高速缓存、主内存间的交互关系

除此之外，为了使得处理器内部的运算单元能尽量被充分利用，处理器可能会对输入代码进行乱序执行（Out-Of-Order Execution）优化，处理器会在计算之后将乱序执行的结果重组，保证该结果与顺序执行的结果是一致的，但并不保证程序中各个语句计算的先后顺序与输入代码中的顺序一致，因此如果存在一个计算任务依赖另外一个计算任务的中间结果，那么其顺序性并不能靠代码的先后顺序来保证。与处理器的乱序执行优化类似，Java 虚拟机的即时编译器中也有类似的指令重排序（Instruction Reorder）优化。

12.3 Java 内存模型

Java 虚拟机规范中试图定义一种 Java 内存模型^①（Java Memory Model, JMM）来

^① 本书中的 Java 内存模型都特指目前正在使用的，在 JDK 1.2 之后建立起来并在 JDK 1.5 中完备过的内存模型。

屏蔽掉各种硬件和操作系统的内存访问差异，以实现让 Java 程序在各种平台下都能达到一致的并发效果。在此之前，主流程序语言（如 C/C++ 等）直接使用物理硬件（或者说操作系统的内存模型），因此，会由于不同平台上内存模型的差异，导致程序在一套平台上并发完全正常，而在另外一套平台上并发访问却经常出错，因此经常需要针对不同的平台来编写程序。

定义 Java 内存模型并非一件容易的事情，这个模型必须定义得足够严谨，才能让 Java 的并发操作不会产生歧义；但是，也必须定义得足够宽松，使得虚拟机的实现能有足够的自由空间去利用硬件的各种特性（寄存器、高速缓存等）来获取更好的执行速度。经过长时间的验证和修补，在 JDK 1.5（实现了 JSR-133^①）发布后，Java 的内存模型就已经成熟和完善起来了。

12.3.1 主内存与工作内存

Java 内存模型的主要目标是定义程序中各个变量的访问规则，即在虚拟机中将变量存储到内存和从内存中取出变量这样的底层细节。此处的变量（Variable）与 Java 编程中所说的变量略有区别，它包括了实例字段、静态字段和构成数组对象的元素，但是不包括局部变量与方法参数，因为后者是线程私有的^②，不会被共享，自然就不存在竞争问题。为了获得较好的执行效能，Java 内存模型并没有限制执行引擎使用处理器的特定寄存器或缓存来和主内存进行交互，也没有限制即时编译器调整代码执行顺序这类权利。

Java 内存模型规定了所有的变量都存储在主内存（Main Memory）中（此处的主内存与介绍物理硬件时的主内存名字一样，两者也可以互相类比，但此处仅是虚拟机内存的一部分）。每条线程还有自己的工作内存（Working Memory，可与前面所讲的处理器高速缓存类比），线程的工作内存中保存了被该线程使用到的变量的主内存副本拷贝，线程对变量的所有操作（读取、赋值等）都必须在工作内存中进行，而不能直接读写主内存中的变量^③。不同的线程之间也无法直接访问对方工作内存中的变量，线程间变量

① JSR-133：Java Memory Model and Thread Specification Revision（Java 内存模型和线程规范修订）。

② 此处请读者注意区分概念：如果局部变量是一个 reference 类型，它引用的对象在 Java 堆中可被各个线程共享，但是 reference 本身在 Java 栈的局部变量表中，它是线程私有的。

③ 根据 Java 虚拟机规范的规定，volatile 变量依然有工作内存的拷贝，但是由于它特殊的操作顺序性规定（后文会讲到），所以看起来如同直接在主内存中读写访问一般，因此这里的描述对于 volatile 也没有例外。

值的传递均需要通过主内存来完成，线程、主内存、工作内存三者的交互关系如图 12-2 所示。

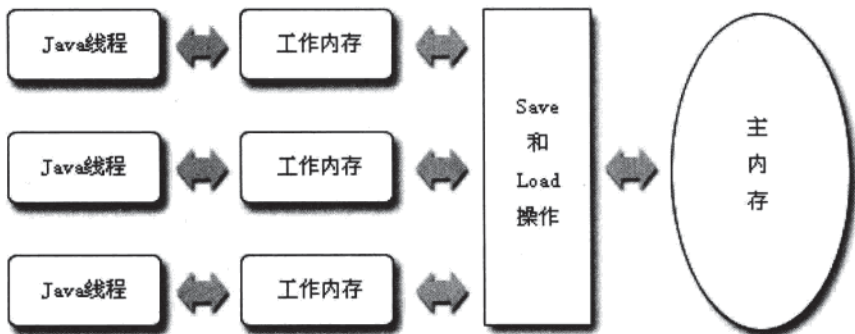


图 12-2 线程、主内存、工作内存三者的交互关系（请与图 12-1 对比）

这里所讲的主内存、工作内存与本书第 2 章所讲的 Java 内存区域中的 Java 堆、栈、方法区等并不是同一个层次的内存划分。如果两者一定要勉强对应起来，那从变量、主内存、工作内存的定义来看，主内存主要对应于 Java 堆中对象的实例数据部分^①，而工作内存则对应于虚拟机栈中的部分区域。从更低的层次来说，主内存就是硬件的内存，而为了获取更好的运行速度，虚拟机及硬件系统可能会让工作内存优先存储于寄存器和高速缓存中。

12.3.2 内存间交互操作

关于主内存与工作内存之间具体的交互协议，即一个变量如何从主内存拷贝到工作内存、如何从工作内存同步回主内存之类的实现细节，Java 内存模型中定义了以下八种操作来完成^②：

- lock（锁定）：作用于主内存的变量，它把一个变量标识为一条线程独占的状态。
- unlock（解锁）：作用于主内存的变量，它把一个处于锁定状态的变量释放出来，释放后的变量才可以被其他线程锁定。

① 除了实例数据，Java 堆还保存了对象的其他信息，对于 HotSpot 虚拟机来讲，有 Mark Word（存储对象哈希码、GC 标志、GC 年龄、同步锁等信息）、Klass Point（指向存储类型元数据的指针）及一些用于字节对齐补白的填充数据（如果实例数据刚好满足 8 字节对齐的话，则可以不存在补白）。

② 虚拟机实现时必须保证每一种操作都是原子的、不可再分的（对于 double 和 long 类型的变量来说，load、store、read 和 write 操作在某些平台上可以有例外，这个问题留待后文再讲）。

- **read** (读取): 作用于主内存的变量, 它把一个变量的值从主内存传输到线程的工作内存中, 以便随后的 **load** 动作使用。
- **load** (载入): 作用于工作内存的变量, 它把 **read** 操作从主内存中得到的变量值放入工作内存的变量副本中。
- **use** (使用): 作用于工作内存的变量, 它把工作内存中一个变量的值传递给执行引擎, 每当虚拟机遇到一个需要使用到变量的值的字节码指令时将会执行这个操作。
- **assign** (赋值): 作用于工作内存的变量, 它把一个从执行引擎接收到的值赋值给工作内存的变量, 每当虚拟机遇到一个给变量赋值的字节码指令时执行这个操作。
- **store** (存储): 作用于工作内存的变量, 它把工作内存中一个变量的值传送到主内存中, 以便随后的 **write** 操作使用。
- **write** (写入): 作用于主内存的变量, 它把 **store** 操作从工作内存中得到的变量的值放入主内存的变量中。

如果要把一个变量从主内存复制到工作内存, 那就要按顺序地执行 **read** 和 **load** 操作, 如果要把变量从工作内存同步回主内存, 就要按顺序地执行 **store** 和 **write** 操作。注意, Java 内存模型只要求上述两个操作必须按顺序执行, 而没有保证必须是连续执行。也就是说 **read** 与 **load** 之间、**store** 与 **write** 之间是可插入其他指令的, 如对主内存中的变量 **a**、**b** 进行访问时, 一种可能出现的顺序是 **read a**、**read b**、**load b**、**load a**。除此之外, Java 内存模型还规定了在执行上述八种基本操作时必须满足如下规则:

- 不允许 **read** 和 **load**、**store** 和 **write** 操作之一单独出现, 即不允许一个变量从主内存读取了但工作内存不接受, 或者从工作内存发起回写了但主内存不接受的情况出现。
- 不允许一个线程丢弃它的最近的 **assign** 操作, 即变量在工作内存中改变了之后必须把该变化同步回主内存。
- 不允许一个线程无原因地 (没有发生过任何 **assign** 操作) 把数据从线程的工作内存同步回主内存中。
- 一个新的变量只能在主内存中“诞生”, 不允许在工作内存中直接使用一个未被初始化 (**load** 或 **assign**) 的变量, 换句话说就是对一个变量实施 **use** 和 **store** 操作

之前，必须先执行过了 `assign` 和 `load` 操作。

- 一个变量在同一个时刻只允许一条线程对其进行 `lock` 操作，但 `lock` 操作可以被同一条线程重复执行多次，多次执行 `lock` 后，只有执行相同次数的 `unlock` 操作，变量才会被解锁。
- 如果对一个变量执行 `lock` 操作，将会清空工作内存中此变量的值，在执行引擎使用这个变量前，需要重新执行 `load` 或 `assign` 操作初始化变量的值。
- 如果一个变量事先没有被 `lock` 操作锁定，则不允许对它执行 `unlock` 操作；也不允许去 `unlock` 一个被其他线程锁定住的变量。
- 对一个变量执行 `unlock` 操作之前，必须先要把此变量同步回主内存中（执行 `sotre` 和 `write` 操作）。

12.3.3 对于 `volatile` 型变量的特殊规则

关键字 `volatile` 可以说是 Java 虚拟机提供的最轻量级的同步机制，但是它并不容易被正确地、完整地理解，以至于许多程序员都习惯不去使用它，遇到需要处理多线程数据竞争的问题时一律使用 `synchronized` 来进行同步。了解 `volatile` 变量的语义对后面了解多线程操作的其他特性有很重要的意义，在本节中我们将多花费一些时间去弄清楚 `volatile` 的语义到底是什么？

Java 内存模型对 `volatile` 专门定义了一些特殊的访问规则，在介绍这些比较拗口的规则定义之前，笔者先用不那么正式，但通俗易懂一些的语言来介绍一下这个关键字的作用。

当一个变量被定义成 `volatile` 之后，它将具备两种特性，第一是保证此变量对所有线程的可见性，这里的“可见性”是指当一条线程修改了这个变量的值，新值对于其他线程来说是可以立即得知的。而普通变量不能做到这一点，变量值在线程间传递均需要通过主内存来完成，如：线程 A 修改一个普通变量的值，然后向主内存进行回写，另外一条线程 B 在线程 A 回写完成了之后再从主内存进行读取操作，新变量的值才会对线程 B 可见。

关于 `volatile` 变量的可见性，经常会被开发人员误解，认为以下描述成立：“`volatile` 变量对所有线程是立即可见的，对 `volatile` 变量所有的写操作都能立刻反应到其他线程之中，换句话说，`volatile` 变量在各个线程中是一致的，所以基于 `volatile` 变量的运算在

并发下是安全的”。这句话的论据部分并没有错，但是其论据并不能得出“基于 volatile 变量的运算在并发下是安全的”这个结论。volatile 变量在各个线程的工作内存中不存在一致性问题（在各个线程的工作内存中 volatile 变量也可以存在不一致的情况，但由于每次使用之前都要先刷新，执行引擎看不到不一致的情况，因此可以认为不存在一致性问题），但是 Java 里面的运算并非原子操作，导致 volatile 变量的运算在并发下一样是不安全的，我们可以通过一段简单的演示来说明原因，请看代码清单 12-1 中演示的例子。

代码清单 12-1 volatile 的运算

```
/**
 * volatile 变量自增运算测试
 *
 * @author zzm
 */
public class VolatileTest {

    public static volatile int race = 0;

    public static void increase() {
        race++;
    }

    private static final int THREADS_COUNT = 20;

    public static void main(String[] args) {
        Thread[] threads = new Thread[THREADS_COUNT];
        for (int i = 0; i < THREADS_COUNT; i++) {
            threads[i] = new Thread(new Runnable() {
                @Override
                public void run() {
                    for (int i = 0; i < 10000; i++) {
                        increase();
                    }
                }
            });
            threads[i].start();
        }

        // 等待所有累加线程都结束
        while (Thread.activeCount() > 1)
```



```

        Thread.yield();

        System.out.println(race);
    }
}

```

这段代码发起了 20 个线程，每个线程对 race 变量进行 10000 次自增操作，如果这段代码能够正确并发的话，最后输出的结果应该是 200000。读者运行完这段代码之后，并不会获得期望的结果，而且会发现每次运行程序，输出的结果都不一样，都是一个小于 200000 的数字，这是为什么呢？

问题就出现在自增运算“race++”之中，我们用 Javap 反编译这段代码后会得到代码清单 12-2，发现只有一行代码的 increase() 方法在 Class 文件中是由 4 条字节码指令构成的（return 指令不是由 race++ 产生的，这条指令可以不算），从字节码层面上很容易就分析出并发失败的原因了：当 getstatic 指令把 race 的值取到操作栈顶时，volatile 关键字保证了 race 的值在此时是正确的，但是在执行 iconst_1、iadd 这些指令的时候，其他线程可能已经把 race 的值加大了，而在操作栈顶的值就变成了过期的数据，所以 putstatic 指令执行后就可能把较小的 race 值同步回主内存之中。

代码清单 12-2 VolatileTest 的字节码

```

public static void increase();
Code:
  Stack=2, Locals=0, Args_size=0
  0:  getstatic      #13; //Field race:I
  3:  iconst_1
  4:  iadd
  5:  putstatic     #13; //Field race:I
  8:  return
LineNumberTable:
  line 14: 0
  line 15: 8

```

实事求是地说，笔者在此使用字节码来分析并发问题，仍然是不严谨的，因为即使编译出来只有一条字节码指令，也并不意味着执行这条指令就是一个原子操作。一条字节码指令在解释执行时，解释器将要运行许多行代码才能实现它的语义，如果是编译执行，一条字节码指令也可能转化成若干条本地机器码指令，此处使用 -XX:+

PrintAssembly 参数输出反汇编来分析会更加严谨一些，但是考虑到读者阅读的方便，并且字节码已经能说明问题，所以此处使用字节码来分析。

由于 volatile 变量只能保证可见性，在不符合以下两条规则的运算场景中，我们仍然要通过加锁（使用 synchronized 或 java.util.concurrent 中的原子类）来保证原子性。

□ 运算结果并不依赖变量的当前值，或者能够确保只有单一的线程修改变量的值。

□ 变量不需要与其他的状态变量共同参与不变约束。

而在像如下的代码清单 12-3 所示的这类场景就很适合使用 volatile 变量来控制并发，当 shutdown() 方法被调用时，能保证所有线程中执行的 doWork() 方法都立即停下来。

代码清单 12-3 volatile 的使用场景

```
volatile boolean shutdownRequested;

public void shutdown() {
    shutdownRequested = true;
}

public void doWork() {
    while (!shutdownRequested) {
        // do stuff
    }
}
```

使用 volatile 变量的第二个语义是禁止指令重排序优化，普通的变量仅仅会保证在该方法的执行过程中所有依赖赋值结果的地方都能获取到正确的结果，而不能保证变量赋值操作的顺序与程序代码中的执行顺序一致。因为在一个线程的方法执行过程中无法感知到这点，这也就是 Java 内存模型中描述的所谓的“线程内表现为串行的语义”（Within-Thread As-If-Serial Semantics）。

上面的描述仍然比较拗口难明，我们还是继续通过一个例子来看看为何指令重排序会干扰程序的并发执行吧，演示程序如代码清单 12-4 所示。

代码清单 12-4 指令重排序

```
Map configOptions;
char[] configText;
// 此变量必须定义为 volatile
volatile boolean initialized = false;
```



```
// 假设以下代码在线程 A 中执行
// 模拟读取配置信息，当读取完成后
// 将 initialized 设置为 true 来通知其他线程配置可用
configOptions = new HashMap();
configText = readConfigFile(fileName);
processConfigOptions(configText, configOptions);
initialized = true;

// 假设以下代码在线程 B 中执行
// 等待 initialized 为 true，代表线程 A 已经把配置信息初始化完成
while (!initialized) {
    sleep();
}
// 使用线程 A 中初始化好的配置信息
doSomethingWithConfig();
```

代码清单 12-4 中的程序是一段伪代码，其中描述的场景十分常见，只是我们在处理配置文件时一般不会出现并发而已。如果定义 `initialized` 变量时没有使用 `volatile` 修饰，就可能会由于指令重排序的优化，导致位于线程 A 中最后一句的代码“`initialized = true`”被提前执行，这样在线程 B 中使用配置信息的代码就可能出现错误，而 `volatile` 关键字则可以避免此类情况的发生^①。

解决了 `volatile` 的语义问题，再来看看在众多保障并发安全的工具中选用 `volatile` 的意义——它能让我们的代码比使用其他的同步工具更快吗？确实在某些情况下，`volatile` 同步机制的性能要优于锁（使用 `synchronized` 关键字或 `java.util.concurrent` 包里面的锁），但是由于虚拟机对锁实行的许多消除和优化，使得我们很难量化地说 `volatile` 就会比 `synchronized` 快上多少。如果让 `volatile` 自己与自己比较，则可以确定一个原则：`volatile` 变量读操作的性能消耗与普通变量几乎没有什么差别，但是写操作则可能会慢上一些，因为它需要在本地代码中插入许多内存屏障（`Memory Barrier` 或 `Memory Fence`）指令^②来保证处理器不发生乱序执行。不过即便如此，大多数场景下 `volatile` 的总开销仍

① `volatile` 屏蔽指令重排序的语义在 JDK 1.5 中才被完全修复，此前的 JDK 中即使将变量声明为 `volatile` 也仍然不能完全避免重排序所导致的问题（主要是 `volatile` 变量前后的代码仍然存在重排序问题），这点也是在 JDK 1.5 之前的 Java 中无法安全地使用 DCL（双锁检测）来实现单例模式的原因。

② Doug Lea 列出了各种处理器架构下的内存屏障指令：<http://g.oswego.edu/dl/jmm/cookbook.html>。

然要比锁来得低，我们在 `volatile` 与锁中选择的唯一判断依据仅仅是 `volatile` 的语义能否满足使用场景的需求。

本节的最后，我们再回头来看看 Java 内存模型中对 `volatile` 变量定义的特殊规则。假定 `T` 表示一个线程，`V` 和 `W` 分别表示两个 `volatile` 型变量，那么在进行 `read`、`load`、`use`、`assign`、`store` 和 `write` 操作时需要满足如下的规则：

- 只有当线程 `T` 对变量 `V` 执行的前一个动作是 `load` 的时候，线程 `T` 才能对变量 `V` 执行 `use` 动作；并且，只有当线程 `T` 对变量 `V` 执行的后一个动作是 `use` 的时候，线程 `T` 才能对变量 `V` 执行 `load` 动作。线程 `T` 对变量 `V` 的 `use` 动作可以认为是与线程 `T` 对变量 `V` 的 `load` 和 `read` 动作相关联的，必须一起连续出现。（这条规则要求在工作内存中，每次使用 `V` 前都必须先从主内存刷新最新的值，用于保证能看见其他线程对变量 `V` 所做的修改后的值）。
- 只有当线程 `T` 对变量 `V` 执行的前一个动作是 `assign` 的时候，线程 `T` 才能对变量 `V` 执行 `store` 动作；并且，只有当线程 `T` 对变量 `V` 执行的后一个动作是 `store` 的时候，线程 `T` 才能对变量 `V` 执行 `assign` 动作。线程 `T` 对变量 `V` 的 `assign` 动作可以认为是与线程 `T` 对变量 `V` 的 `store` 和 `write` 动作相关联的，必须一起连续出现（这条规则要求在工作内存中，每次修改 `V` 后都必须立刻同步回主内存中，用于保证其他线程可以看到自己对变量 `V` 所做的修改）。
- 假定动作 `A` 是线程 `T` 对变量 `V` 实施的 `use` 或 `assign` 动作，假定动作 `F` 是与动作 `A` 相关联的 `load` 或 `store` 动作，假定动作 `P` 是与动作 `F` 相应的对变量 `V` 的 `read` 或 `write` 动作；类似地，假定动作 `B` 是线程 `T` 对变量 `W` 实施的 `use` 或 `assign` 动作，假定动作 `G` 是与动作 `B` 相关联的 `load` 或 `store` 动作，假定动作 `Q` 是与动作 `G` 相应的对变量 `W` 的 `read` 或 `write` 动作。如果 `A` 先于 `B`，那么 `P` 先于 `Q`（这条规则要求 `volatile` 修饰的变量不会被指令重排序优化，保证代码的执行顺序与程序的顺序相同）。

12.3.4 对于 `long` 和 `double` 型变量的特殊规则

Java 内存模型要求 `lock`、`unlock`、`read`、`load`、`assign`、`use`、`store` 和 `write` 这八个操作都具有原子性，但是对于 64 位的数据类型（`long` 和 `double`），在模型中特别定义了一条宽松的规定：允许虚拟机将没有被 `volatile` 修饰的 64 位数据的读写操作划分为两次

32 位的操作来进行，即允许虚拟机实现选择可以不保证 64 位数据类型的 load、store、read 和 write 这四个操作的原子性，这点就是所谓的 long 和 double 的非原子性协定 (Nonatomic Treatment of double and long Variables)。

如果有多个进程共享一个并未声明为 volatile 的 long 或 double 类型的变量，并且同时对它们进行读取和修改操作，那么某些线程可能会读取到一个既非原值，也不是其他线程修改值的代表了“半个变量”的数值。

不过这种读取到“半个变量”的情况非常罕见，因为 Java 内存模型虽然允许虚拟机不把 long 和 double 变量的读写实现成原子操作，但允许虚拟机选择把这些操作实现为具有原子性的操作，而且还“强烈建议”虚拟机这样实现。在实际开发中，目前各种平台下的商用虚拟机几乎都选择把 64 位数据的读写操作作为原子操作来对待，因此我们在编写代码时一般不需要将用到的 long 和 double 变量专门声明为 volatile。

12.3.5 原子性、可见性与有序性

介绍完 Java 内存模型的相关操作和规则，我们再整体回顾一下这个模型的特征。Java 内存模型是围绕着在并发过程中如何处理原子性、可见性和有序性这三个特征来建立的，我们逐个来看一下哪些操作实现了这三个特性。

原子性 (Atomicity)：由 Java 内存模型来直接保证的原子性变量操作包括 read、load、assign、use、store 和 write 这六个，我们大致可以认为基本数据类型的访问读写是具备原子性的 (long 和 double 的非原子性协定例外，笔者的观点是知道这件事情就可以了，无须太过在意这些几乎不会发生的例外情况)。

如果应用场景需要一个更大范围的原子性保证 (经常会遇到)，Java 内存模型还提供了 lock 和 unlock 操作来满足这种需求，尽管虚拟机未把 lock 和 unlock 操作直接开放给用户使用，但是却提供了更高层次的字节码指令 monitorenter 和 monitorexit 来隐式地使用这两个操作，这两个字节码指令反映到 Java 代码中就是同步块——synchronized 关键字，因此在 synchronized 块之间的操作也具备原子性。

可见性 (Visibility)：可见性就是指当一个线程修改了共享变量的值，其他线程能够立即得知这个修改。上文在讲解 volatile 变量的时候我们已详细讨论过这一点。Java 内存模型是通过在变量修改后将新值同步回主内存，在变量读取前从主内存刷新变量值这种依赖主内存作为传递媒介的方式来实现可见性的，无论是普通变量还是 volatile 变量

都是如此，普通变量与 `volatile` 变量的区别是 `volatile` 的特殊规则保证了新值能立即同步到主内存，以及每次使用前立即从主内存刷新。因此我们可以说 `volatile` 保证了多线程操作时变量的可见性，而普通变量则不能保证这一点。

除了 `volatile` 之外，Java 还有两个关键字能实现可见性，它们是 `synchronized` 和 `final`。同步块的可见性是由“对一个变量执行 `unlock` 操作之前，必须先把此变量同步回主内存中（执行 `sotre` 和 `write` 操作）”这条规则获得的，而 `final` 关键字的可见性是指：被 `final` 修饰的字段在构造器中一旦被初始化完成，并且构造器没有把“`this`”的引用传递出去（`this` 引用逃逸是一件很危险的事情，其他线程有可能通过这个引用访问到“初始化了一半”的对象），那么在其他线程中就能看见 `final` 字段的值。如代码清单 12-5 所示，变量 `i` 与 `j` 都具备可见性，它们无须同步就能被其他线程正确地访问。

代码清单 12-5 `final` 与可见性

```
public static final int i;

public final int j;

static {
    i = 0;
    // do something
}

{
    // 也可以选择在构造函数中初始化
    j = 0;
    // do something
}
```

有序性 (Ordering)：Java 内存模型的有序性在前面讲解 `volatile` 时也详细地讨论过了，Java 程序中天然的有序性可以总结为一句话：如果在本线程内观察，所有的操作都是有序的；如果在一个线程中观察另一个线程，所有的操作都是无序的。前半句是指“线程内表现为串行的语义”（`Within-Thread As-If-Serial Semantics`），后半句是指“指令重排序”现象和“工作内存与主内存同步延迟”现象。

Java 语言提供了 `volatile` 和 `synchronized` 两个关键字来保证线程之间操作的有序性，`volatile` 关键字本身就包含了禁止指令重排序的语义，而 `synchronized` 则是由“一个变量

在同一个时刻只允许一条线程对其进行 lock 操作”这条规则获得的，这个规则决定了持有同一个锁的两个同步块只能串行地进入。

介绍完并发的三种重要特性，读者有没有发现 synchronized 关键字在需要这三种特性的时候都可以作为其中一种的解决方案？看起来很“万能”吧？的确，大部分的并发控制操作都能使用 synchronized 来完成。synchronized 的“万能”也间接造就了它被程序员滥用的局面，越“万能”的并发控制，通常会伴随着越大的性能影响，这点我们将在下一章再谈。

12.3.6 先行发生原则

如果 Java 内存模型中所有的有序性都只靠 volatile 和 synchronized 来完成，那么有一些操作将会变得很啰嗦，但是我们在编写 Java 并发代码的时候并没有感觉到这一点，这是因为 Java 语言中有一个“先行发生”（happens-before）的原则。这个原则非常重要，它是判断数据是否存在竞争，线程是否安全的主要依据，依赖这个原则，我们可以通过几条规则一揽子解决并发环境下两个操作之间是否可能存在冲突的所有问题。

现在就来看看“先行发生”原则指的是什么。先行发生是 Java 内存模型中定义的两项操作之间的偏序关系，如果说操作 A 先行发生于操作 B，其实就是在发生操作 B 之前，操作 A 产生的影响能被操作 B 观察到，“影响”包括修改了内存中共享变量的值、发送了消息、调用了方法等。这句话不难理解，但它意味着什么呢？我们可以举个例子来说明一下，如代码清单 12-6 中所示的这三句伪代码：

代码清单 12-6 先行发生原则的示例 1

```
// 以下操作在线程 A 中执行
i = 1;

// 以下操作在线程 B 中执行
j = i;

// 以下操作在线程 C 中执行
i = 2;
```

假设线程 A 中的操作“i = 1”先行发生于线程 B 的操作“j = i”，那我们就可以确定在线程 B 的操作执行后，变量 j 的值一定是等于 1，得出这个结论的依据有两个，一是

根据先行发生原则，“ $i = 1$ ”的结果可以被观察到；二是线程 C 登场之前，线程 A 操作结束之后没有其他线程会修改变量 i 的值。现在再来考虑线程 C，我们依然保持线程 A 和 B 之间的先行发生关系，而 C 出现在线程 A 和 B 的操作之间，但是 C 与 B 没有先行发生关系，那 j 的值会是多少呢？答案是不确定！1 和 2 都有可能，因为线程 C 对变量 i 的影响可能会被线程 B 观察到，也可能不会，这时候线程 B 就存在读取到过期数据的风险，不具备多线程安全性。

下面是 Java 内存模型下一些“天然的”先行发生关系，这些先行发生关系无须任何同步器协助就已经存在，可以在编码中直接使用。如果两个操作之间的关系不在此列，并且无法从下列规则推导出来的话，它们就没有顺序性保障，虚拟机可以对它们进行随意地重排序。

- **程序次序规则 (Program Order Rule)**：在一个线程内，按照程序代码顺序，书写在前面的操作先行发生于书写在后面的操作。准确地说应该是控制流顺序而不是程序代码顺序，因为要考虑分支、循环等结构。
- **管程锁定规则 (Monitor Lock Rule)**：一个 `unlock` 操作先行发生于后面对同一个锁的 `lock` 操作。这里必须强调的是同一个锁，而“后面”是指时间上的先后顺序。
- **volatile 变量规则 (Volatile Variable Rule)**：对一个 `volatile` 变量的写操作先行发生于后面对这个变量的读操作，这里的“后面”同样是指时间上的先后顺序。
- **线程启动规则 (Thread Start Rule)**：Thread 对象的 `start()` 方法先行发生于此线程的每一个动作。
- **线程终止规则 (Thread Termination Rule)**：线程中的所有操作都先行发生于对此线程的终止检测，我们可以通过 `Thread.join()` 方法结束、`Thread.isAlive()` 的返回值等手段检测到线程已经终止执行。
- **线程中断规则 (Thread Interruption Rule)**：对线程 `interrupt()` 方法的调用先行发生于被中断线程的代码检测到中断事件的发生，可以通过 `Thread.interrupted()` 方法检测到是否有中断发生。
- **对象终结规则 (Finalizer Rule)**：一个对象的初始化完成（构造函数执行结束）先行发生于它的 `finalize()` 方法的开始。
- **传递性 (Transitivity)**：如果操作 A 先行发生于操作 B，操作 B 先行发生于操作 C，那就可以得出操作 A 先行发生于操作 C 的结论。

Java 语言无须任何同步手段保障就能成立的先行发生规则就只有上面这些了，笔者演示一下如何使用这些规则去判定操作间是否具备顺序性，对于读写共享变量的操作来说，就是线程是否安全，读者还可以从下面这个例子中感受一下“时间上的先后顺序”与“先行发生”之间有什么不同。演示例子如代码清单 12-7 所示。

代码清单 12-7 先行发生原则的示例 2

```
private int value = 0;

public void setValue(int value){
    this.value = value;
}

public int getValue(){
    return value;
}
```

代码清单 12-7 中显示的是一组再普通不过的 getter/setter 方法，假设存在线程 A 和 B，线程 A 先（时间上的先后）调用了“setValue(1)”，然后线程 B 调用了同一个对象的“getValue()”，那么线程 B 收到的返回值是什么？

我们依次分析一下先行发生原则中的各项规则，由于两个方法分别由线程 A 和 B 调用，不在一个线程中，所以程序次序规则在这里不适用；由于没有同步块，自然就不会发生 lock 和 unlock 操作，所以管程锁定规则不适用；由于 value 变量没有被 volatile 关键字修饰，所以 volatile 变量规则不适用；后面的线程启动、终止、中断规则和对象终结规则也和这里完全扯不上关系。因为没有一个是适用的先行发生规则，所以最后一条传递性也无从谈起，因此我们可以判定尽管线程 A 在操作时间上先于线程 B，但是无法确定 B 中“getValue()”方法的返回结果，换句话说，这里面的操作不是线程安全的。

那怎么修复这个问题呢？我们至少有两种比较简单的方案可以选择：要么把 getter/setter 方法都定义为 synchronized 方法，这样就可以套用管程锁定规则；要么把 value 定义为 volatile 变量，由于 setter 方法对 value 的修改不依赖 value 的原值，满足了 volatile 关键字使用场景，这样就可以套用 volatile 变量规则来实现先行发生关系。

通过上面的例子，我们可以得出结论：一个操作“时间上的先发生”不代表这个操作会是“先行发生”，那如果一个操作“先行发生”是否就能推导出这个操作必定是“时间上的先发生”呢？很遗憾，这个推论也是不成立的，一个典型的例子就是多次提

到的“指令重排序”，演示例子如代码清单 12-8 所示。

代码清单 12-8 先行发生原则示例 3

```
// 以下操作在同一个线程中执行
int i = 1;
int j = 2;
```

代码清单 12-8 的两条复制语句在同一个线程之中，根据程序次序规则，“int i = 1”的操作先行发生于“int j = 2”，但是“int j = 2”的代码完全可能先被处理器执行，这并不影响先行发生原则的正确性，因为我们在这条线程之中没有办法感知到这点。

上面两个例子综合起来证明了一个结论：时间上的先后顺序与先行发生原则之间基本没有太大的关系，所以我们衡量并发安全问题的时候不要受到时间顺序的干扰，一切必须以先行发生原则为准。

12.4 Java 与线程

并发不一定要依赖多线程（如 PHP 中很常见的多进程并发），但是在 Java 里面谈论并发，大多数都与线程脱不开关系。既然我们这本书探讨的话题是 Java 虚拟机的特性，那讲到 Java 线程，我们就从 Java 线程在虚拟机中的实现开始讲起。

12.4.1 线程的实现

我们知道，线程是比进程更轻量级的调度执行单位，线程的引入，可以把一个进程的资源分配和执行调度分开，各个线程既可以共享进程资源（内存地址、文件 I/O 等），又可以独立调度（线程是 CPU 调度的最基本单位）。

主流的操作系统都提供了线程实现，Java 语言则提供了在不同硬件和操作系统平台下对线程操作的统一处理，每个 `java.lang.Thread` 类的实例就代表了一个线程。不过 `Thread` 类与大部分的 Java API 有着显著的差别，它的所有关键方法都被声明为 `Native`。在 Java API 中一个 `Native` 方法可能就意味着这个方法没有使用或无法使用平台无关的手段来实现（当然也可能是为了执行效率而使用 `Native` 方法，不过通常最高效率的手段也就是平台相关的手段）。正因为这个原因，作者把本节的标题定为“线程的实现”而不是“Java 线程的实现”。

实现线程主要有三种方式：使用内核线程实现，使用用户线程实现，使用用户线程加轻量级进程混合实现。

1. 使用内核线程实现

内核线程（Kernel Thread, KLT）就是直接由操作系统内核（Kernel，下称内核）支持的线程，这种线程由内核来完成线程切换，内核通过操纵调度器（Scheduler）对线程进行调度，并负责将线程的任务映射到各个处理器上。每个内核线程都可以看做是内核的一个分身，这样操作系统就有能力同时处理多件事情，支持多线程的内核就叫多线程内核（Multi-Threads Kernel）。

程序一般不会直接去使用内核线程，而是去使用内核线程的一种高级接口——轻量级进程（Light Weight Process, LWP），轻量级进程就是我们通常意义上所讲的线程，由于每个轻量级进程都由一个内核线程支持，因此只有先支持内核线程，才能有轻量级进程。这种轻量级进程与内核线程之间 1:1 的关系称为一对一的线程模型，如图 12-3 所示。

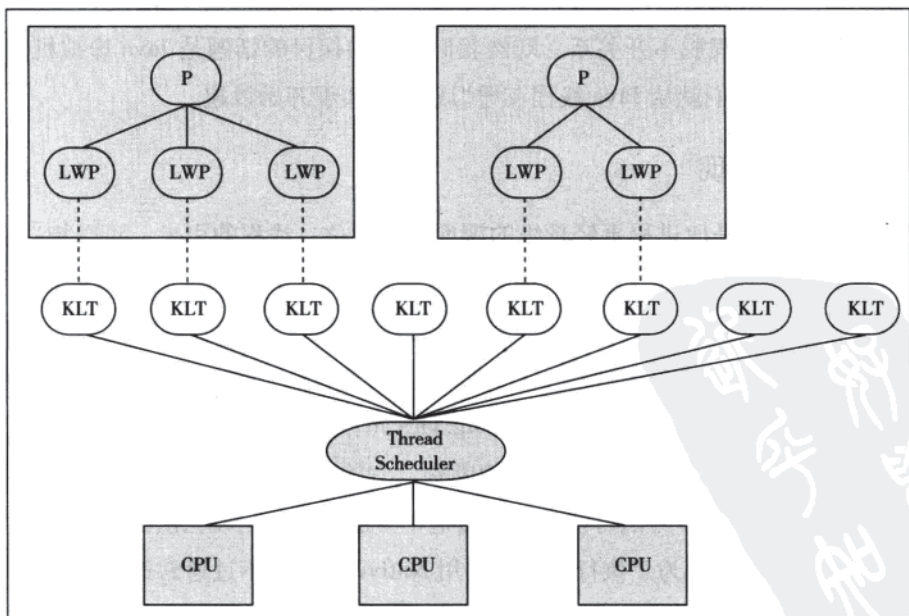


图 12-3 轻量级进程与内核线程之间 1:1 的关系

由于内核线程的支持，每个轻量级进程都成为一个独立的调度单元，即使有一个轻量级进程在系统调用中阻塞了，也不会影响整个进程继续工作，但是轻量级进程具有它的局限性：首先，由于是基于内核线程实现的，所以各种进程操作，如创建、析构及同步，都需要进行系统调用。而系统调用的代价相对较高，需要在用户态（User Mode）和内核态（Kernel Mode）中来回切换。其次，每个轻量级进程都需要有一个内核线程的支持，因此轻量级进程要消耗一定的内核资源（如内核线程的栈空间），因此一个系统支持轻量级进程的数量是有限的。

2. 使用用户线程实现

广义上来讲，一个线程只要不是内核线程，那就可以认为是用户线程（User Thread, UT），因此从这个定义上来讲轻量级进程也属于用户线程，但轻量级进程的实现始终是建立在内核之上的，许多操作都要进行系统调用，因此效率会受到限制。

而狭义上的用户线程指的是完全建立在用户空间的线程库上，系统内核不能感知到线程存在的实现。用户线程的建立、同步、销毁和调度完全在用户态中完成，不需要内核的帮助。如果程序实现得当，这种线程不需要切换到内核态，因此操作可以是非常快速且低消耗的，也可以支持规模更大的线程数量，部分高性能数据库中的多线程就是由用户线程实现的。这种进程与用户线程之间 1:N 的关系称为一对多的线程模型，如图 12-4 所示。

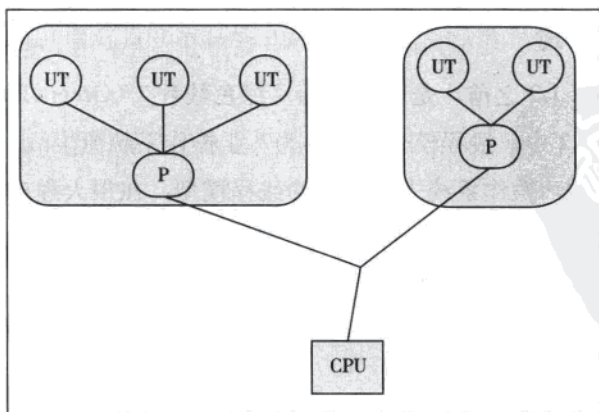


图 12-4 进程与用户线程之间 1:N 的关系

使用用户线程的优势在于不需要系统内核支援，劣势也在于没有系统内核的支援，所有的线程操作都需要用户程序自己处理。线程的创建、切换和调度都是需要考虑问题，而且由于操作系统只把处理器资源分配到进程，那诸如“阻塞如何处理”、“多处理器系统中如何将线程映射到其他处理器上”这类问题解决起来将会异常困难，甚至不可能完成。因而使用用户线程实现的程序一般都比较复杂^①，除了以前在不支持多线程的操作系统中（如 DOS）的多线程程序与少数有特殊需求的程序外，现在使用用户线程的程序越来越少了，Java、Ruby 等语言都曾经使用过用户线程，最终又都放弃了使用它。

3. 混合实现

线程除了依赖内核线程实现和完全由用户程序自己实现之外，还有一种将内核线程与用户线程一起使用的实现方式。在这种混合实现下，既存在用户线程，也存在轻量级进程。用户线程还是完全建立在用户空间中，因此用户线程的创建、切换、析构等操作依然廉价，并且可以支持大规模的用户线程并发。而操作系统提供支持的轻量级进程则作为用户线程和内核线程之间的桥梁，这样可以使用内核提供的线程调度功能及处理器映射，并且用户线程的系统调用要通过轻量级线程来完成，大大降低了进程被阻塞的风险。在这种混合模式中，用户线程与轻量级进程的数量比是不定的，是 M:N 的关系，如图 12-5 所示，这种就是多对多的线程模型。

许多 Unix 系列的操作系统，如 Solaris、HP-UX 等都提供了 M:N 的线程模型实现。

4. Java 线程的实现

Java 线程在 JDK 1.2 之前，是基于名为“绿色线程”（Green Threads）的用户线程实现的，而在 JDK 1.2 中，线程模型被替换为基于操作系统原生线程模型来实现。因此在目前的 JDK 版本中，操作系统支持怎样的线程模型，在很大程度上就决定了 Java 虚拟机的线程是怎样映射的，这点在不同的平台上没有办法达成一致，虚拟机规范中也并未限定 Java 线程需要使用哪种线程模型来实现。线程模型只对线程的并发规模和操作成本产生影响，对 Java 程序的编码和运行过程来说，这些差异都是透明的。

^① 此处所讲的“复杂”与“程序自己完成线程操作”，并没有限制程序中必须编写了复杂的用于实现用户线程的代码，使用用户线程的程序，很多都依赖特定的线程库来完成基本的线程操作，这些复杂性都封装在线程库之中。

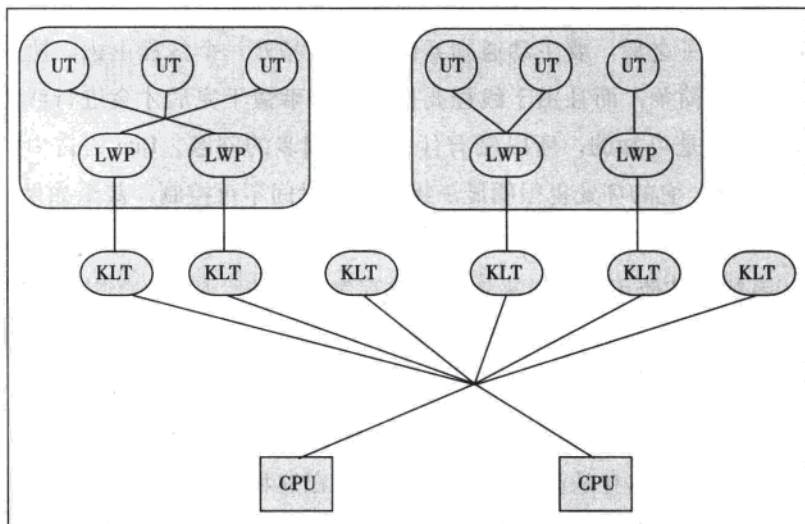


图 12-5 用户线程与轻量级进程之间 M:N 的关系

对于 Sun JDK 来说，它的 Windows 版与 Linux 版都是使用一对一的线程模型来实现的，一条 Java 线程就映射到一条轻量级进程之中，因为 Windows 和 Linux 系统提供的线程模型就是一对一的^①。

而在 Solaris 平台中，由于操作系统的线程特性可以同时支持一对一（通过 Bound Threads 或 Alternate Libthread 实现）及多对多（通过 LWP / Thread Based Synchronization 实现）的线程模型，因此在 Solaris 版的 JDK 中也对应提供了两个平台专有的虚拟机参数：`-XX:+UseLWPSynchronization`（默认值）和 `-XX:+UseBoundThreads` 来明确指定虚拟机使用的是哪种线程模型。

12.4.2 Java 线程调度

线程调度是指系统为线程分配处理器使用权的过程，主要调度方式有两种，分别是协同式（Cooperative Threads-Scheduling）线程调度和抢占式（Preemptive Threads-Scheduling）线程调度。

^① Windows 下有纤程包（Fiber Package），Linux 下也有 NGPT（在 2.4 内核的年代）来实现 M:N 模型，但是它们都没有成为主流。

如果使用协同式调度的多线程系统，线程的执行时间由线程本身来控制，线程把自己的工作执行完了之后，要主动通知系统切换到另外一个线程上去。协同式多线程的最大好处是实现简单，而且由于线程要把自己的事情干完后才会进行线程切换，切换操作对线程自己是可知的，所以没有什么线程同步的问题。Lua 语言中的“协同例程”就是这类实现。它的坏处也很明显：线程执行时间不可控制，甚至如果一个线程编写有问题，一直不告知系统进行线程切换，那么程序就会一直阻塞在那里。很久以前的 Windows 3.x 系统就是使用协同式来实现多进程多任务的，那是相当的不稳定，一个进程坚持不让出 CPU 执行时间就会导致整个系统的崩溃。

如果使用抢占式调度的多线程系统，那么每个线程将由系统来分配执行时间，线程的切换不由线程本身来决定（在 Java 中，`Thread.yield()` 可以让出执行时间，但是要获取执行时间的话，线程本身是没有什么办法的）。在这种实现线程调度的方式下，线程的执行时间是系统可控的，也不会有一个线程导致整个进程阻塞的问题，Java 使用的线程调度方式就是抢占式调度^①。与前面所说的 Windows 3.x 的例子相对，在 Windows 9x/NT 内核中就是使用抢占式来实现多进程的，当一个进程出了问题，我们还可以使用任务管理器把这个进程杀掉，而不至于导致系统崩溃。

虽然说 Java 线程调度是系统自动完成的，但是我们还是可以“建议”系统给某些线程多分配一点执行时间，另外的一些线程则可以少分配一点——这项操作可以通过设置线程优先级来完成。Java 语言一共设置了 10 个级别的线程优先级（`Thread.MIN_PRIORITY` 至 `Thread.MAX_PRIORITY`），在两个线程同时处于 Ready 状态时，优先级越高的线程越容易被系统选择执行。

不过，线程优先级并不是太靠谱，原因是 Java 的线程是被映射到系统的原生线程上来实现的，所以线程调度最终还是由操作系统说了算，虽然现在很多操作系统都提供线程优先级的概念，但是并不见得能与 Java 线程的优先级一一对应，如 Solaris 中有 2147483648（2 的 31 次方）种优先级，但 Windows 中就只有 7 种，比 Java 线程优先级多的系统还好说，中间留下一点空位就是了，但比 Java 线程优先级少的系统，就不得不出现几个优先级相同的情况了，表 12-1 显示了 Java 线程优先级与 Windows 线程优先级

^① 在 JDK 1.7 中有可能提供协程（Coroutines）方式来进行多任务处理，相关资料可参见 <http://wikis.sun.com/display/mlvm/Coroutines>。

之间的对应关系，Windows 平台的 JDK 中使用了除 `THREAD_PRIORITY_IDLE` 之外的其余 6 种线程优先级。

表 12-1 Java 线程优先级与 Windows 线程优先级之间的对应关系

Java 线程优先级	Windows 线程优先级
1 (<code>Thread.MIN_PRIORITY</code>)	<code>THREAD_PRIORITY_LOWEST</code>
2	<code>THREAD_PRIORITY_LOWEST</code>
3	<code>THREAD_PRIORITY_BELOW_NORMAL</code>
4	<code>THREAD_PRIORITY_BELOW_NORMAL</code>
5 (<code>Thread.NORM_PRIORITY</code>)	<code>THREAD_PRIORITY_NORMAL</code>
6	<code>THREAD_PRIORITY_ABOVE_NORMAL</code>
7	<code>THREAD_PRIORITY_ABOVE_NORMAL</code>
8	<code>THREAD_PRIORITY_HIGHEST</code>
9	<code>THREAD_PRIORITY_HIGHEST</code>
10 (<code>Thread.MAX_PRIORITY</code>)	<code>THREAD_PRIORITY_CRITICAL</code>

上文说到“线程优先级并不是太靠谱”，不仅仅是说在一些平台上不同的优先级实际会变得相同这一点，还有其他情况让我们不能太依赖优先级：优先级可能会被系统自行改变。例如在 Windows 系统中存在一个名为“优先级推进器”的功能（Priority Boosting，当然它可以被关闭掉），它的大致作用就是当系统发现一个线程被执行得特别“勤奋努力”的话，可能会越过线程优先级去为它分配执行时间。因此我们不能在程序中通过优先级来完全准确地判断一组状态都为 Ready 的线程将会先执行哪一个。

12.4.3 状态转换

Java 语言定义了 5 种进程状态，在任意一个时间点中，一个进程只能有且只有其中的一种状态，这 5 种状态分别是：

- 新建 (New)：创建后尚未启动的线程处于这种状态。
- 运行 (Runnable)：Runnable 包括了操作系统线程状态中的 Running 和 Ready，也就是处于此状态的线程有可能正在执行，也有可能正在等待着 CPU 为它分配执行时间。
- 无限期待 (Waiting)：处于这种状态的进程不会被分配 CPU 执行时间，它们要

等待被其他线程显式地唤醒。以下方法会让线程陷入无限期的等待状态：

- 没有设置 Timeout 参数的 Object.wait() 方法。
- 没有设置 Timeout 参数的 Thread.join() 方法。
- LockSupport.park() 方法。

□ 限期等待 (Timed Waiting)：处于这种状态的进程也不会被分配 CPU 执行时间，不过无须等待被其他线程显式地唤醒，在一定时间之后它们会由系统自动唤醒。以下方法会让线程进入限期等待状态：

- Thread.sleep() 方法。
- 设置了 Timeout 参数的 Object.wait() 方法。
- 设置了 Timeout 参数的 Thread.join() 方法。
- LockSupport.parkNanos() 方法。
- LockSupport.parkUntil() 方法。

□ 阻塞 (Blocked)：进程被阻塞了，“阻塞状态”与“等待状态”的区别是：“阻塞状态”在等待着获取到一个排它锁，这个事件将在另外一个线程放弃这个锁的时候发生；而“等待状态”则是在等待一段时间，或者唤醒动作的发生。在程序等待进入同步区域的时候，线程将进入这种状态。

□ 结束 (Terminated)：已终止线程的线程状态，线程已经结束执行。

上述 5 种状态在遇到特定事件发生的时候将会互相转换，它们的转换关系如图 12-6 所示。

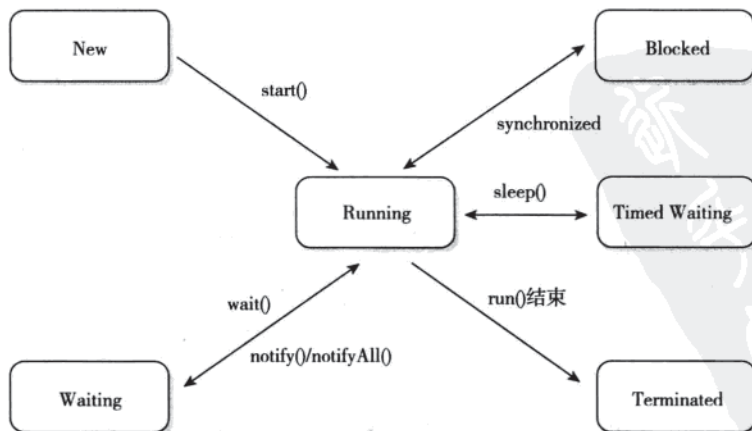


图 12-6 线程状态转换关系

12.5 本章小结

本章中，我们了解了虚拟机 Java 内存模型的结构及操作，并且讲解了原子性、可见性、有序性在 Java 内存模型中的体现，介绍了先行发生原则的规则及使用。另外，我们还了解了线程在 Java 语言之中是如何实现的。

关于“高效并发”这个话题，在本章中主要介绍了虚拟机如何实现“并发”，在下一章中，我们的主要关注点将是虚拟机如何实现“高效并发”，虚拟机对我们编写的并发代码提供了什么样的优化手段。



第 13 章 线程安全与锁优化

本章主要内容

- 概述
- 线程安全
- 锁优化

并发处理的广泛应用是使得 Amdahl 定律代替摩尔定律成为计算机性能发展源动力的根本原因，也是人类压榨计算机运算能力最有力的武器。

13.1 概述

在软件业发展的初期，程序编写都是以算法为核心的，程序员会把数据和过程分别作为独立的部分来考虑，数据代表问题空间中的客体，程序代码则用于处理这些数据，这种思维方式是直接站在计算机的角度去抽象问题和解决问题，称为面向过程的编程思想。与此相对，面向对象的编程思想则站在现实世界的角度去抽象和解决问题，它把数据和行为都看做是对象的一部分，这样可以让程序员能以符合现实世界的思维方式来编写和组织程序。

面向过程的编程思想极大地提升了现代软件开发的生产效率和软件可以达到的规模，但是现实世界与计算机世界之间不可避免地存在一些差异。例如，人们很难想象现实中的对象在一项工作进行期间，会被不停地中断和切换，对象的属性（数据）可能会在中断期间被修改和变脏，而这些事件在计算机世界中则是很正常的事情。有时候，良好的设计原则不得不向现实做出一些让步，我们必须让程序在计算机中正确无误地运行，然后再考虑如何将代码组织得更好，让程序运行得更快。对于这部分的主题“高效并发”来讲，首先需要保证并发的正确性，然后在此基础上来实现高效。本章就先从如

何保证并发的正确性，如何实现线程安全说起。

13.2 线程安全

“线程安全”这个名称，相信稍有经验的程序员都会听说过，甚至在代码编写和走查的时候可能还会经常挂在嘴边，但是如何找到一个不太拗口的概念来定义线程安全却不是一件容易的事情，笔者尝试在 Google 中搜索它的概念，找到的是类似于“如果一个对象可以安全地被多个线程同时使用，那它就是线程安全的”这样的定义——并不能说它不正确，但是令人无法从中获取到任何有用的信息。

笔者认为《Java Concurrency In Practice》的作者 Brian Goetz 对“线程安全”有一个比较恰当的定义：“当多个线程访问一个对象时，如果不用考虑这些线程在运行时环境下的调度和交替执行，也不需要进行额外的同步，或者在调用方进行任何其他的协调操作，调用这个对象的行为都可以获得正确的结果，那这个对象就是线程安全的”。

这个定义很严谨，它要求了线程安全的代码必须都具备一个特征：代码本身封装了所有必要的正确性保障手段（如互斥同步等），令调用者无须关心多线程的问题，更无须自己实现任何措施来保证多线程的正确调用。这点并不容易做到，在大多数场景中，我们都会将这个定义弱化一些，如果把“调用这个对象的行为”限定为“单次调用”，这个定义的其他描述也能够成立的话，我们就可以称它是线程安全的了，为什么要弱化这个定义，我们稍后再详细探讨。

13.2.1 Java 语言中的线程安全

我们已经有了线程安全的一个抽象定义，那接下来我们就讨论一下在 Java 语言中，线程安全具体是如何体现的？有哪些操作是线程安全的？我们这里讨论的线程安全，就限定于多个线程之间存在共享数据访问这个前提，因为如果一段代码根本不会与其他线程共享数据，那么从线程安全的角度上看，程序是串行执行还是多线程执行对它来说是完全没有区别的。

为了更深入地理解线程安全，在这里我们可以不把线程安全当作一个非真即假的二元排它选项来看待，按照线程安全的“安全程度”由强至弱来排序，我们^①可以将 Java

① 这种划分方法也是 Brian Goetz 发表在 IBM developWorkers 上的一篇文章中提出的，这里写“我们”纯粹是笔者下笔行文中的语言用法。

语言中各种操作共享的数据分为以下五类：不可变、绝对线程安全、相对线程安全、线程兼容和线程对立。

1. 不可变

在 Java 语言里面（特指 JDK 1.5 以后，即 Java 内存模型被修正之后的 Java 语言），不可变（Immutable）的对象一定是线程安全的，无论是对象的方法实现还是方法的调用者，都不需要再进行任何的线程安全保障措施，在上一章里我们谈到过 final 关键字带来的可见性时曾经提到过这一点，只要一个不可变的对象被正确地构建出来（没有发生 this 引用逃逸的情况），那其外部的可见状态永远也不会改变，永远也不会看到它在多个线程之中处于不一致的状态。“不可变”带来的安全性是最简单最纯粹的。

Java 语言中，如果共享数据是一个基本数据类型，那么只要在定义时使用 final 关键字修饰它就可以保证它是不可变的。如果共享数据是一个对象，那就需要保证对象的行为不会对其状态产生任何影响才行，如果读者还没想明白这句话，不妨想一想 java.lang.String 类的对象，它是一个典型的不可变对象，我们调用它的 substring()、replace() 和 concat() 这些方法都不会影响它原来的值，只会返回一个新构造的字符串对象。

保证对象行为不影响自己状态的途径有很多种，其中最简单的就是把对象中带有状态的变量都声明为 final，这样在构造函数结束之后，它就是不可变的，例如代码清单 13-1 中 java.lang.Integer 构造函数所示的，它通过将内部状态变量 value 定义为 final 来保障状态不变。

代码清单 13-1 JDK 中 Integer 类的构造函数

```
/**
 * The value of the Integer.
 * @serial
 */
private final int value;

/**
 * Constructs a newly allocated Integer object that
 * represents the specified int value.
 *
 * @param value the value to be represented by the
 * Integer object.
 */
```



```
public Integer(int value) {  
    this.value = value;  
}
```

在 Java API 中符合不可变要求的类型，除了上面提到的 String 之外，常用的还有枚举类型，以及 `java.lang.Number` 的部分子类，如 Long 和 Double 等数值包装类型，BigInteger 和 BigDecimal 等大数据类型；但同为 Number 的子类型的原子类 AtomicInteger 和 AtomicLong 则并非不可变的，读者不妨看看这两个原子类的源码，想一想为什么。

2. 绝对线程安全

绝对的线程安全完全满足 Brian Goetz 给出的线程安全的定义，这个定义其实是很严格的，一个类要达到“不管运行时环境如何，调用者都不需要任何额外的同步措施”通常需要付出很大的，甚至是不切实际的代价。在 Java API 中标注自己是线程安全的类，大多数都不是绝对的线程安全。我们可以通过 Java API 中一个不是“绝对线程安全”的线程安全类来看看这里的“绝对”是什么意思。

如果说 `java.util.Vector` 是一个线程安全的容器，相信所有的 Java 程序员对此都不会有异议，因为它的 `add()`、`get()` 和 `size()` 这类方法都是被 `synchronized` 修饰的，尽管这样效率很低，但确实是安全的。但是，即使它所有的方法都被修饰成同步，也不意味着调用它的时候永远都不再需要同步手段了，请看看下面代码清单 13-2 中的测试代码。

代码清单 13-2 对 Vector 线程安全的测试

```
private static Vector<Integer> vector = new Vector<Integer>();  
  
public static void main(String[] args) {  
    while (true) {  
        for (int i = 0; i < 10; i++) {  
            vector.add(i);  
        }  
  
        Thread removeThread = new Thread(new Runnable() {  
            @Override  
            public void run() {  
                for (int i = 0; i < vector.size(); i++) {  
                    vector.remove(i);  
                }  
            }  
        });  
        removeThread.start();  
    }  
}
```




```

    }
}
};

Thread printThread = new Thread(new Runnable() {
    @Override
    public void run() {
        for (int i = 0; i < vector.size(); i++) {
            System.out.println((vector.get(i)));
        }
    }
});

removeThread.start();
printThread.start();

// 不要同时产生过多的线程，否则会导致操作系统假死
while (Thread.activeCount() > 20);
}
}

```

运行结果如下：

```

Exception in thread "Thread-132"
java.lang.ArrayIndexOutOfBoundsException:
Array index out of range: 17
    at java.util.Vector.remove(Vector.java:777)
    at org.fenixsoft.mulithread.VectorTest$1.run(VectorTest.java:21)
    at java.lang.Thread.run(Thread.java:662)

```

很明显，尽管这里使用到的 `Vector` 的 `get()`、`remove()` 和 `size()` 方法都是同步的，但是在多线程的环境中，如果不在方法调用端做额外的同步措施，使用这段代码仍然是不安全的，因为如果另一个线程恰好在错误的的时间里删除了一个元素，导致序号 `i` 已经不再可用的话，`get()` 方法就会抛出一个 `ArrayIndexOutOfBoundsException`。如果要保证这段代码能正确地执行下去，我们不得不把 `removeThread` 和 `printThread` 的定义改成如下面代码清单 13-3 所示的这样。

代码清单 13-3 必须加入同步以保证 `Vector` 访问的线程安全性

```

Thread removeThread = new Thread(new Runnable() {
    @Override

```

```
public void run() {
    synchronized (vector) {
        for (int i = 0; i < vector.size(); i++) {
            vector.remove(i);
        }
    }
}

Thread printThread = new Thread(new Runnable() {
    @Override
    public void run() {
        synchronized (vector) {
            for (int i = 0; i < vector.size(); i++) {
                System.out.println((vector.get(i)));
            }
        }
    }
});
```

3. 相对线程安全

相对的线程安全就是我们通常意义上所讲的线程安全，它需要保证对这个对象单独的操作是线程安全的，我们在调用的时候不需要做额外的保障措施，但是对于一些特定顺序的连续调用，就可能需要在调用端使用额外的同步手段来保证调用的正确性。上面代码清单 13-2 和代码清单 13-3 就是相对线程安全的一个很明显的案例。

在 Java 语言中，大部分的线程安全类都属于这种类型，例如 Vector、HashTable、Collections 的 synchronizedCollection() 方法包装的集合等。

4. 线程兼容

线程兼容是指对象本身并不是线程安全的，但是可以通过在调用端正确地使用同步手段来保证对象在并发环境中安全地使用，我们平常说一个类不是线程安全的，绝大多数指的都是这种情况。Java API 中大部分的类都是线程兼容的，如与前面的 Vector 和 HashTable 相对应的集合类 ArrayList 和 HashMap 等。

5. 线程对立

线程对立是指不管调用端是否采取了同步措施，都无法在多线程环境中并发使用的代码。由于 Java 语言天生就具备多线程特性，线程对立这种排斥多线程的代码是很少出

现的，而且通常都是有害的，应当尽量避免。

一个线程对立的例子是 Thread 类的 suspend() 和 resume() 方法，如果有两个线程同时持有一个线程对象，一个尝试去中断线程，一个尝试去恢复线程，如果并发进行的话，无论调用时是否进行了同步，目标线程都是存在死锁风险的，如果 suspend() 中断的线程就是即将要执行 resume() 的那个线程，那就肯定要产生死锁了。也正是由于这个原因，suspend() 和 resume() 方法已经被 JDK 声明废弃 (@Deprecated) 了。常见的线程对立的还有 System.setIn()、System.setOut() 和 System.runFinalizersOnExit() 等。

13.2.2 线程安全的实现方法

了解了什么是线程安全之后，紧接着的一个问题就是我们应该如何实现线程安全，这听起来似乎是一件由代码如何编写来决定事情，确实，如何实现线程安全与代码的编写有很大的关系，但虚拟机提供的同步和锁机制也起到了非常重要的作用。本节将介绍代码编写如何实现线程安全和虚拟机如何实现同步与锁，相对而言更偏重后者一些，只要读者了解了虚拟机线程安全手段的运作过程，自己去思考代码如何编写并不是一件困难的事情。

1. 互斥同步

互斥同步 (Mutual Exclusion & Synchronization) 是最常见的一种并发正确性保障手段，同步是指在多个线程并发访问共享数据时，保证共享数据在同一个时刻只被一条 (或者是一些，使用信号量的时候) 线程使用。而互斥是实现同步的一种手段，临界区 (Critical Section)、互斥量 (Mutex) 和信号量 (Semaphore) 都是主要的互斥实现方式。因此在这四个字里面，互斥是因，同步是果，互斥是方法，同步是目的。

在 Java 里面，最基本的互斥同步手段就是 synchronized 关键字，synchronized 关键字经过编译之后，会在同步块的前后分别形成 monitorenter 和 monitorexit 这两个字节码指令，这两个字节码都需要一个 reference 类型的参数来指明要锁定和解锁的对象。如果 Java 程序中的 synchronized 明确指定了对象参数，那就是这个对象的 reference；如果没有明确指定，那就根据 synchronized 修饰的是实例方法还是类方法，去取对应的对象实例或 Class 对象来作为锁对象。

根据虚拟机规范的要求，在执行 monitorenter 指令时，首先要去尝试获取对象的锁。如果这个对象没被锁定，或者当前线程已经拥有了那个对象的锁，把锁的计数器加 1，相

应地，在执行 `monitorexit` 指令时会将锁计数器减 1，当计数器为 0 时，锁就被释放了。如果获取对象锁失败了，那当前线程就要阻塞等待，直到对象锁被另外一个线程释放为止。

在虚拟机规范对 `monitorenter` 和 `monitorexit` 的行为描述中，有两点是需要特别注意的。首先，`synchronized` 同步块对同一条线程来说是可重入的，不会出现自己把自己锁死的问题。其次，同步块在已进入的线程执行完之前，会阻塞后面其他线程的进入。上一章讲过，Java 的线程是映射到操作系统的原生线程之上的，如果要阻塞或唤醒一条线程，都需要操作系统来帮忙完成，这就需要从用户态转换到核心态中，因此状态转换需要耗费很多的处理器时间。对于代码简单的同步块（如被 `synchronized` 修饰的 `getter()` 或 `setter()` 方法），状态转换消耗的时间可能比用户代码执行的时间还要长。所以 `synchronized` 是 Java 语言中一个重量级（Heavyweight）的操作，有经验的程序员都会在确实必要的情况下才使用这种操作。而虚拟机本身也会进行一些优化，譬如在通知操作系统阻塞线程之前加入一段自旋等待过程，避免频繁地切入到核心态之中。

除了 `synchronized` 之外，我们还可以使用 `java.util.concurrent`（下文称 J.U.C）包中的重入锁（`ReentrantLock`）来实现同步，在基本用法上，`ReentrantLock` 与 `synchronized` 很相似，他们都具备一样的线程重入特性，只是代码写法上有点区别，一个表现为 API 层面的互斥锁（`lock()` 和 `unlock()` 方法配合 `try/finally` 语句块来完成），一个表现为原生语法层面的互斥锁。不过 `ReentrantLock` 比 `synchronized` 增加了一些高级功能，主要有以下三项：等待可中断、可实现公平锁，以及锁可以绑定多个条件。

- ❑ 等待可中断是指当持有锁的线程长期不释放锁的时候，正在等待的线程可以选择放弃等待，改为处理其他事情，可中断特性对处理执行时间非常长的同步块很有帮助。
- ❑ 公平锁是指多个线程在等待同一个锁时，必须按照申请锁的时间顺序来依次获得锁；而非公平锁则不保证这一点，在锁被释放时，任何一个等待锁的线程都有机会获得锁。`synchronized` 中的锁是非公平的，`ReentrantLock` 默认情况下也是非公平的，但可以通过带布尔值的构造函数要求使用公平锁。
- ❑ 锁绑定多个条件是指一个 `ReentrantLock` 对象可以同时绑定多个 `Condition` 对象，而在 `synchronized` 中，锁对象的 `wait()` 和 `notify()` 或 `notifyAll()` 方法可以实现一个隐含的条件，如果要和多于一个的条件关联的时候，就不得不额外地添加一个锁，而 `ReentrantLock` 则无须这样做，只需要多次调用 `newCondition()` 方法即可。

如果需要使用到上述功能的时候，选用 `ReentrantLock` 是一个很好的选择，那如果是基于性能考虑呢？关于 `synchronized` 和 `ReentrantLock` 的性能问题，Brian Goetz 对这两种锁在 JDK 1.5、单核处理器及双 Xeon 处理器环境下做了一组吞吐量对比的实验^①，实验结果如图 13-1 和图 13-2 所示。

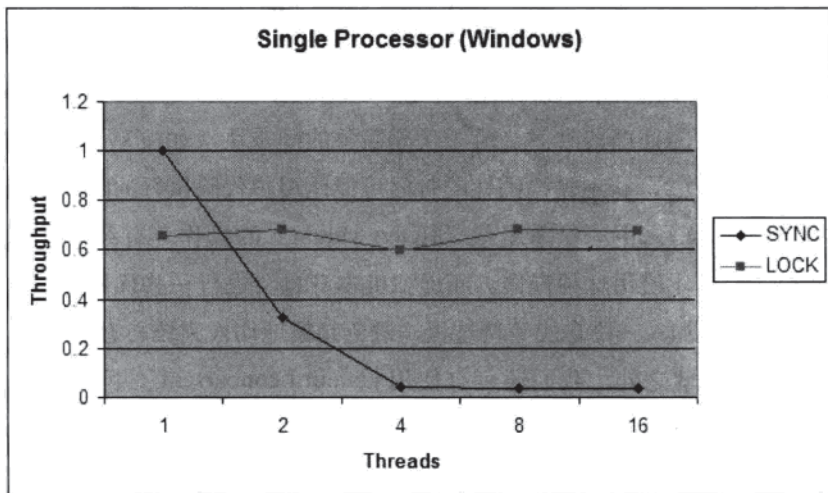


图 13-1 JDK 1.5、单核处理器下两种锁的吞吐量对比

从图 13-1 和图 13-2 可以看出，多线程环境下 `synchronized` 的吞吐量下降得非常严重，而 `ReentrantLock` 则能基本保持在同一个比较稳定的水平上。与其说 `ReentrantLock` 性能好，倒还不如说 `synchronized` 还有非常大的优化余地。后续的技术发展也证明了这一点，JDK 1.6 中加入了针对锁的优化措施（下一节我们会讲解这些优化措施），JDK 1.6 发布之后，人们就发现 `synchronized` 与 `ReentrantLock` 的性能基本上是完全持平了。因此如果读者的程序是使用 JDK 1.6 部署的话，性能因素就不再是选择 `ReentrantLock` 的理由了，虚拟机在未来的性能改进中肯定也会更加偏向于原生的 `synchronized`，所以还是提倡在 `synchronized` 能实现需求的情况下，优先考虑使用 `synchronized` 来进行同步。

^① 本例中的数据及图片来源于 Brian Goetz 为 IBM developerWorks 撰写的论文：《Java theory and practice: More flexible, scalable locking in JDK 5.0》，原文地址是：http://www.ibm.com/developerworks/java/library/j-jtp10264/?S_TACT=105AGX52&S_CMP=cn-a-j。

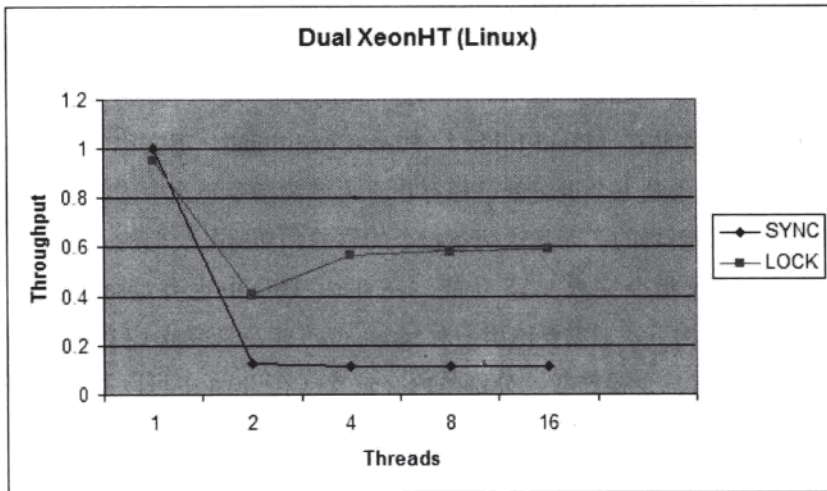


图 13-2 JDK 1.5、双 Xeon 处理器下两种锁的吞吐量对比

2. 非阻塞同步

互斥同步最主要的问题就是进行线程阻塞和唤醒所带来的性能问题，因此这种同步也被称为阻塞同步（Blocking Synchronization）。另外，它属于一种悲观的并发策略，总是认为只要不去做正确的同步措施（加锁），那就肯定会出现问题，无论共享数据是否真的会出现竞争，它都要进行加锁（这里说的是概念模型，实际上虚拟机机会优化掉很大一部分不必要的加锁）、用户态核心态转换、维护锁计数器和检查是否有被阻塞的线程需要被唤醒等操作。随着硬件指令集的发展，我们有了另外一个选择：基于冲突检测的乐观并发策略，通俗地说就是先进行操作，如果没有其他线程争用共享数据，那操作就成功了；如果共享数据有争用，产生了冲突，那就再进行其他的补偿措施（最常见的补偿措施就是不断地重试，直到试成功为止），这种乐观的并发策略的许多实现都不需要把线程挂起，因此这种同步操作被称为非阻塞同步（Non-Blocking Synchronization）。

为什么笔者说使用乐观并发策略需要“硬件指令集的发展”才能进行呢？因为我们需要操作和冲突检测这两个步骤具备原子性，靠什么来保证呢？如果这里再使用互斥同步来保证就失去意义了，所以我们只能靠硬件来完成这件事情，硬件保证一个从语义上看起来需要多次操作的行为只通过一条处理器指令就能完成，这类指令常用的有：

□ 测试并设置（Test-and-Set）；

- 获取并增加 (Fetch-and-Increment)；
- 交换 (Swap)；
- 比较并交换 (Compare-and-Swap, 下文称 CAS)；
- 加载链接 / 条件储存 (Load-Linked / Store-Conditional, 下文称 LL/SC)。

其中,前面的三条是上个世纪就已经存在于大多数指令集之中的处理器指令,后面的两条是现代处理器新增的,而且这两条指令的目的和功能是类似的。在 IA64、x86 指令集中通过 `cmpxchg` 指令完成 CAS 功能,在 `sparc-TSO` 中也有 `casa` 指令实现,而在 ARM 和 PowerPC 架构下,则需要使用一对 `ldrex/strex` 指令来完成 LL/SC 的功能。

CAS 指令需要有三个操作数,分别是内存位置(在 Java 中可以简单理解为变量的内存地址,用 V 表示)、旧的预期值(用 A 表示)和新值(用 B 表示)。CAS 指令执行时,当且仅当 V 符合旧预期值 A 时,处理器用新值 B 更新 V 的值,否则它就不执行更新,但是不管是否更新了 V 的值,都会返回 V 的旧值,上述的处理过程是一个原子操作。

在 JDK 1.5 之后,Java 程序中才可以使用 CAS 操作,该操作由 `sun.misc.Unsafe` 类里面的 `compareAndSwapInt()` 和 `compareAndSwapLong()` 等几个方法包装提供,虚拟机在内部对这些方法做了特殊处理,即时编译出来的结果就是一条平台相关的处理器 CAS 指令,没有方法调用的过程,或者可以认为是无条件内联进去了^①。

由于 `Unsafe` 类不是提供给用户程序调用的类(`Unsafe.getUnsafe()` 的代码中限制了只有启动类加载器(Bootstrap ClassLoader)加载的 Class 才能访问它),如果不采用反射手段,我们只能通过其他的 Java API 来间接使用它,如 J.U.C 包里面的整数原子类,其中的 `compareAndSet()` 和 `getAndIncrement()` 等方法都使用了 `Unsafe` 类的 CAS 操作。

我们不妨拿一段在上一章中没有解决的问题代码来看看如何使用 CAS 操作来避免阻塞同步,代码如上一章的代码清单 12-1 所示。我们曾经通过这段 20 个线程自增 10000 次的代码来证明 `volatile` 变量不具备原子性,那么如何才能让它具备原子性呢?把“`race++`”操作或 `increase()` 方法用同步块包裹起来当然是一个办法,但是如果改成如代码清单 13-4 所示的代码,那效率将会提高许多。

^① 这种被虚拟机特殊处理的方法称为固有函数 (Intrinsics), 类似的固有函数还有 `Math.sin()` 等。

代码清单 13-4 Atomic 的原子自增运算

```
/**
 * Atomic 变量自增运算测试
 *
 * @author zzm
 */
public class AtomicTest {

    public static AtomicInteger race = new AtomicInteger(0);

    public static void increase() {
        race.incrementAndGet();
    }

    private static final int THREADS_COUNT = 20;

    public static void main(String[] args) throws Exception {
        Thread[] threads = new Thread[THREADS_COUNT];
        for (int i = 0; i < THREADS_COUNT; i++) {
            threads[i] = new Thread(new Runnable() {
                @Override
                public void run() {
                    for (int i = 0; i < 10000; i++) {
                        increase();
                    }
                }
            });
            threads[i].start();
        }

        while (Thread.activeCount() > 1)
            Thread.yield();

        System.out.println(race);
    }
}
```

运行结果如下：

```
200000
```

使用 AtomicInteger 代替 int 后，程序输出了正确的结果，一切都要归功于

incrementAndGet() 方法的原子性。它的实现其实非常简单，如代码清单 13-5 所示。

代码清单 13-5 incrementAndGet() 方法的 JDK 源码

```
/**
 * Atomically increment by one the current value.
 * @return the previous value
 */
public final int getAndIncrement() {
    for (;;) {
        int current = get();
        int next = current + 1;
        if (compareAndSet(current, next))
            return current;
    }
}
```

incrementAndGet() 方法在一个无限循环中，不断尝试将一个比当前值大 1 的新值赋值给自己。如果失败了，那说明在执行“获取 - 设置”操作的时候值已经有了修改，于是再次循环进行下一次操作，直到设置成功为止。

尽管 CAS 看起来很美，但显然这种操作无法涵盖互斥同步的所有使用场景，并且 CAS 从语义上来说并不是完美的，存在这样的一个逻辑漏洞：如果一个变量 V 初次读取的时候是 A 值，并且在准备赋值的时候检查到它仍然为 A 值，那我们就能说它的值没有被其他线程改变过了吗？如果在这段期间它的值曾经被改成了 B，后来又被改回为 A，那 CAS 操作就会误认为它从来没有被改变过。这个漏洞称为 CAS 操作的“ABA”问题。J.U.C 包为了解决这个问题，提供了一个带有标记的原子引用类“AtomicStampedReference”，它可以通过控制变量值的版本来保证 CAS 的正确性。不过目前来说这个类比较鸡肋，大部分情况下 ABA 问题不会影响程序并发的正确性，如果需要解决 ABA 问题，改用传统的互斥同步可能会比原子类更高效。

3. 无同步方案

要保证线程安全，并不是一定就要进行同步，两者没有因果关系。同步只是保障共享数据争用时的正确性的手段，如果一个方法本来就不涉及共享数据，那它自然就无须任何同步措施去保证正确性，因此会有一些代码天生就是线程安全的，笔者简单介绍其中的两类。

可重入代码 (Reentrant Code): 这种代码也叫纯代码 (Pure Code), 可以在代码执行的任何时刻中断它, 转而去执行另外一段代码 (包括递归调用它本身), 而在控制权返回后, 原来的程序不会出现任何错误。相对线程安全来说, 可重入性是更基本的特性, 它可以保证线程安全, 即所有的可重入的代码都是线程安全的, 但是并非所有的线程安全的代码都是可重入的。

可重入代码有一些共同的特征: 例如不依赖存储在堆上的数据和公用的系统资源、用到的状态量都由参数中传入、不调用非可重入的方法等。我们可以通过一个简单一些的原则来判断代码是否具备可重入性: 如果一个方法, 它的返回结果是可以预测的, 只要输入了相同的数据, 就都能返回相同的结果, 那它就满足可重入性的要求, 当然也就是线程安全的。

线程本地存储 (Thread Local Storage): 如果一段代码中所需要的数据必须与其他代码共享, 那就看看这些共享数据的代码是否能保证在同一个线程中执行? 如果能保证, 我们就可以把共享数据的可见范围限制在同一个线程之内, 这样, 无须同步也能保证线程之间不出现数据争用的问题。

符合这种特点的应用并不少见, 大部分使用消费队列的架构模式 (如“生产者-消费者”模式) 都会将产品的消费过程尽量在一个线程中消费完, 其中最重要的一个应用实例就是经典 Web 交互模型中的“一个请求对应一个服务器线程” (Thread-per-Request) 的处理方式, 这种处理方式的广泛应用使得 Web 服务端的很多应用都可以使用线程本地存储来解决线程安全问题。

Java 语言中, 如果一个变量要被多线程访问, 可以使用 `volatile` 关键字声明它为“易变的”; 如果一个变量要被某个线程独享, 因为 Java 中没有类似 C++ 中 `__declspec(thread)`^① 这样的关键字, 不过可以通过 `java.lang.ThreadLocal` 类来实现线程本地存储的功能。每一个线程的 `Thread` 对象中都有一个 `ThreadLocalMap` 对象, 这个对象存储了一组以 `ThreadLocal.threadLocalHashCode` 为键, 以本地线程变量为值的 K-V 值对, `ThreadLocal` 对象就是当前线程的 `ThreadLocalMap` 的访问入口, 每一个 `ThreadLocal` 对象都包含了一个独一无二的 `threadLocalHashCode` 值, 使用这个值就可以在线程 K-V 值对中找到对应的本地线程变量。

^① 在 Visual C++ 中是“`__declspec(thread)`”关键字, 在 GCC 中是“`__thread`”。

13.3 锁优化

高效并发是 JDK 1.6 的一个重要主题，HotSpot 虚拟机开发团队在这个版本上花费了大量的精力去实现各种锁优化技术，如适应性自旋（Adaptive Spinning）、锁消除（Lock Elimination）、锁粗化（Lock Coarsening）、轻量级锁（Lightweight Locking）、偏向锁（Biased Locking）等，这些技术都是为了在线程之间更高效地共享数据，以及解决竞争问题，从而提高程序的执行效率。

13.3.1 自旋锁与自适应自旋

前面我们讨论互斥同步的时候，提到了互斥同步对性能最大的影响是阻塞的实现，挂起线程和恢复线程的操作都需要转入内核态中完成，这些操作给系统的并发性能带来了很大的压力。同时，虚拟机的开发团队也注意到在许多应用上，共享数据的锁定状态只会持续很短的一段时间，为了这段时间去挂起和恢复线程并不值得。如果物理机器有一个以上的处理器，能让两个或以上的线程同时并行执行，我们就可以让后面请求锁的那个线程“稍等一会儿”，但不放弃处理器的执行时间，看看持有锁的线程是否很快就会释放锁。为了让线程等待，我们只须让线程执行一个忙循环（自旋），这项技术就是所谓的自旋锁。

自旋锁在 JDK 1.4.2 中就已经引入，只不过默认是关闭的，可以使用 `-XX:+UseSpinning` 参数来开启，在 JDK 1.6 中就已经改为默认开启了。自旋等待不能代替阻塞，且先不说对处理器数量的要求，自旋等待本身虽然避免了线程切换的开销，但它是要占用处理器时间的，所以如果锁被占用的时间很短，自旋等待的效果就会非常好，反之如果锁被占用的时间很长，那么自旋的线程只会白白消耗处理器资源，而不会做任何有用的工作，反而会带来性能的浪费。因此自旋等待的时间必须要有一定的限度，如果自旋超过了限定的次数仍然没有成功获得锁，就应当使用传统的方式去挂起线程了。自旋次数的默认值是 10 次，用户可以使用参数 `-XX:PreBlockSpin` 来更改。

在 JDK 1.6 中引入了自适应的自旋锁。自适应意味着自旋的时间不再固定了，而是由前一次在同一个锁上的自旋时间及锁的拥有者的状态来决定。如果在同一个锁对象上，自旋等待刚刚成功获得过锁，并且持有锁的线程正在运行中，那么虚拟机就会认为这次自旋也很有可能再次成功，进而它将允许自旋等待持续相对更长的时间，比如 100

个循环。另一方面，如果对于某个锁，自旋很少成功获得过，那在以后要获取这个锁时将可能省略掉自旋过程，以避免浪费处理器资源。有了自适应自旋，随着程序运行和性能监控信息的不断完善，虚拟机对程序锁的状况预测就会越来越准确，虚拟机就会变得越来越“聪明”了。

13.3.2 锁消除

锁消除是指虚拟机即时编译器在运行时，对一些代码上要求同步，但是被检测到不可能存在共享数据竞争的锁进行消除。锁消除的主要判定依据来源于逃逸分析的数据支持（第 11 章已经讲解过逃逸分析技术），如果判断到一段代码中，在堆上的所有数据都不会逃逸出去被其他线程访问到，那就可以把它们当做栈上数据对待，认为它们是线程私有的，同步加锁自然就无须进行。

也许读者会有疑问，变量是否逃逸，对于虚拟机来说需要使用数据流分析来确定，但是程序员自己应该是很清楚的，怎么会在明知道不存在数据争用的情况下要求同步呢？答案是有许多同步措施并不是程序员自己加入的，同步的代码在 Java 程序中的普遍程度也许超过了大部分读者的想象。我们来看看下面代码清单 13-6 中的例子，这段非常简单的代码仅仅是输出三个字符串相加的结果，无论是源码字面上还是程序语义上都没有同步。

代码清单 13-6 一段看起来没有同步的代码

```
public String concatString(String s1, String s2, String s3) {  
    return s1 + s2 + s3;  
}
```

我们也知道，由于 String 是一个不可变的类，对字符串的连接操作总是通过生成新的 String 对象来进行的，因此 Javac 编译器会对 String 连接做自动优化。在 JDK 1.5 之前，会转化为 StringBuffer 对象的连续 append() 操作，在 JDK 1.5 及以后的版本中，会转化为 StringBuilder 对象的连续 append() 操作。即代码清单 13-6 中的代码可能会变成代码清单 13-7 的样子^①。

^① 实事求是地说，既然谈到锁消除与逃逸分析，那虚拟机就不可能是 JDK 1.5 之前的版本，所以实际上会转化为非线程安全的 StringBuilder 来完成字符串拼接，而不会加锁。但是这也不影响笔者用这个例子来证明 Java 对象中同步的普遍性。

代码清单 13-7 Javac 转化后的字符串连接操作

```
public String concatString(String s1, String s2, String s3) {  
    StringBuffer sb = new StringBuffer();  
    sb.append(s1);  
    sb.append(s2);  
    sb.append(s3);  
    return sb.toString();  
}
```

现在大家还认为这段代码没有涉及同步吗？每个 `StringBuffer.append()` 方法中都有一个同步块，锁就是 `sb` 对象。虚拟机观察变量 `sb`，很快就会发现它的动态作用域被限制在 `concatString()` 方法的内部。也就是 `sb` 的所有引用永远不会“逃逸”到 `concatString()` 方法之外，其他线程无法访问到它，所以这里虽然有锁，但是可以被安全地消除掉，在即时编译之后，这段代码就会忽略掉所有的同步而直接执行了。

13.3.3 锁粗化

原则上，我们在编写代码的时候，总是推荐将同步块的作用范围限制得尽量小——只在共享数据的实际作用域中才进行同步，这样是为了使得需要同步的操作数量尽可能变小，如果存在锁竞争，那等待锁的线程也能尽快地拿到锁。

大部分情况下，上面的原则都是正确的，但是如果一系列的连续操作都对同一个对象反复加锁和解锁，甚至加锁操作是出现在循环体中的，那即使没有线程竞争，频繁地进行互斥同步操作也会导致不必要的性能损耗。

代码清单 13-7 中连续的 `append()` 方法就属于这类情况。如果虚拟机探测到有这样一串零碎的操作都对同一个对象加锁，将会把加锁同步的范围扩展（粗化）到整个操作序列的外部，以代码清单 13-7 为例，就是扩展到第一个 `append()` 操作之前直至最后一个 `append()` 操作之后，这样只需要加锁一次就可以了。

13.3.4 轻量级锁

轻量级锁是 JDK 1.6 中加入的新型锁机制，它名字中的“轻量级”是相对于使用操作系统互斥量来实现的传统锁而言的，因此传统的锁机制就被称为“重量级”锁。首先需要强调一点的是，轻量级锁并不是用来代替重量级锁的，它的本意是在没有多线程竞争的前提下，减少传统的重量级锁使用操作系统互斥量产生的性能消耗。

要理解轻量级锁，以及后面会讲到的偏向锁的原理和运作过程，必须从 HotSpot 虚拟机的对象（对象头部分）的内存布局开始介绍。HotSpot 虚拟机的对象头（Object Header）分为两部分信息，第一部分用于存储对象自身的运行时数据，如哈希码（HashCode）、GC 分代年龄（Generational GC Age）等，这部分数据的长度在 32 位和 64 位的虚拟机中分别为 32 个和 64 个 Bits，官方称它为“Mark Word”，它是实现轻量级锁和偏向锁的关键。另外一部分用于存储指向方法区对象类型数据的指针，如果是数组对象的话，还会有一个额外的部分用于存储数组长度。

对象头信息是与对象自身定义的数据无关的额外存储成本，考虑到虚拟机的空间效率，Mark Word 被设计成一个非固定的数据结构以便在极小的空间内存储尽量多的信息，它会根据对象的状态复用自己的存储空间。例如在 32 位的 HotSpot 虚拟机中对象未被锁定的状态下，Mark Word 的 32 个 Bits 空间中的 25Bits 用于存储对象哈希码（HashCode），4Bits 用于存储对象分代年龄，2Bits 用于存储锁标志位，1Bit 固定为 0，在其他状态（轻量级锁定、重量级锁定、GC 标记、可偏向）下对象的存储内容如表 13-1 所示。

表 13-1 HotSpot 虚拟机对象头 Mark Word

存储内容	标志位	状 态
对象哈希码、对象分代年龄	01	未锁定
指向锁记录的指针	00	轻量级锁定
指向重量级锁的指针	10	膨胀（重量级锁定）
空，不需要记录信息	11	GC 标记
偏向线程 ID、偏向时间戳、对象分代年龄	01	可偏向

简单地介绍完了对象的内存布局，我们把话题返回到轻量级锁的执行过程上。在代码进入同步块的时候，如果此同步对象没有被锁定（锁标志位为“01”状态），虚拟机首先将在当前线程的栈帧中建立一个名为锁记录（Lock Record）的空间，用于存储锁对象目前的 Mark Word 的拷贝（官方把这份拷贝加了一个 Displaced 前缀，即 Displaced Mark Word），这时候线程堆栈与对象头的状态如图 13-3 所示。

然后，虚拟机将使用 CAS 操作尝试将对象的 Mark Word 更新为指向 Lock Record 的指针。如果这个更新动作成功了，那么这个线程就拥有了该对象的锁，并且对象 Mark Word 的锁标志位（Mark Word 的最后两个 Bits）将转变为“00”，即表示此对象处于轻量级锁定的状态，这时候线程堆栈与对象头的状态如图 13-4 所示。

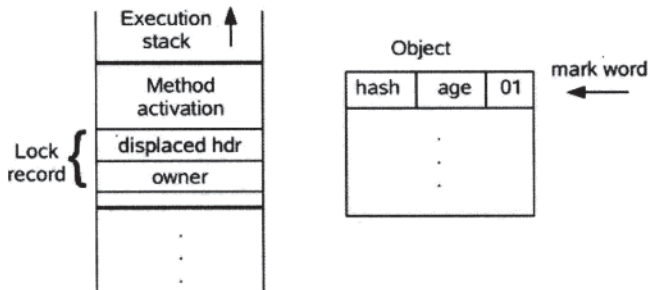
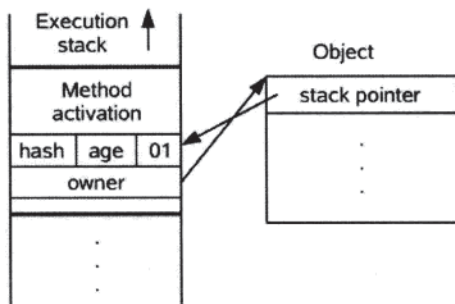
图 13-3 轻量级锁 CAS 操作之前堆栈与对象的状态^①

图 13-4 轻量级锁 CAS 操作之后堆栈与对象的状态

如果这个更新操作失败了，虚拟机首先会检查对象的 Mark Word 是否指向当前线程的栈帧，如果是就说明当前线程已经拥有了这个对象的锁，可以直接进入同步块继续执行，否则说明这个锁对象已经被其他线程抢占了。如果有两条以上的线程争用同一个锁，那轻量级锁就不再有效，要膨胀为重量级锁，锁标志的状态值变为“10”，Mark Word 中存储的就是指向重量级锁（互斥量）的指针，后面等待锁的线程也要进入阻塞状态。

上面描述的是轻量级锁的加锁过程，它的解锁过程也是通过 CAS 操作来进行的，如果对象的 Mark Word 仍然指向着线程的锁记录，那就用 CAS 操作把对象当前的 Mark Word 和线程中复制的 Displaced Mark Word 替换回来，如果替换成功，整个同步过程就完成了。如果替换失败，说明有其他线程尝试过获取该锁，那就要在释放锁的同时，唤醒被挂起的线程。

^① 图 13-3 和图 13-4 来源于 HotSpot 虚拟机的一位 Senior Staff Engineer——Paul Hohensee 所写的 PPT “The Hotspot Java Virtual Machine”。

轻量级锁能提升程序同步性能的依据是“对于绝大部分的锁，在整个同步周期内都是不存在竞争的”，这是一个经验数据。如果没有竞争，轻量级锁使用 CAS 操作避免了使用互斥量的开销，但如果存在锁竞争，除了互斥量的开销外，还额外发生了 CAS 操作，因此在有竞争的情况下，轻量级锁会比传统的重量级锁更慢。

13.3.5 偏向锁

偏向锁也是 JDK 1.6 中引入的一项锁优化，它的目的是消除数据在无竞争情况下的同步原语，进一步提高程序的运行性能。如果说轻量级锁是在无竞争的情况下使用 CAS 操作去消除同步使用的互斥量，那偏向锁就是在无竞争的情况下把整个同步都消除掉，连 CAS 操作都不做了。

偏向锁的“偏”，就是偏心的“偏”、偏袒的“偏”。它的意思是这个锁会偏向于第一个获得它的线程，如果在接下来的执行过程中，该锁没有被其他的线程获取，则持有偏向锁的线程将永远不需要再进行同步。

如果读者读懂了前面轻量级锁中关于对象头 Mark Word 与线程之间的操作过程，那偏向锁的原理理解起来就会很简单。假设当前虚拟机启用了偏向锁（启用参数 `-XX:+UseBiasedLocking`，这是 JDK 1.6 的默认值），那么，当锁对象第一次被线程获取的时候，虚拟机将会把对象头中的标志位设为“01”，即偏向模式。同时使用 CAS 操作把获取到这个锁的线程的 ID 记录在对象的 Mark Word 之中，如果 CAS 操作成功，持有偏向锁的线程以后每次进入这个锁相关的同步块时，虚拟机都可以不再进行任何同步操作（例如 Locking、Unlocking 及对 Mark Word 的 Update 等）。

当有另外一个线程去尝试获取这个锁时，偏向模式就宣告结束。根据锁对象目前是否处于被锁定的状态，撤销偏向（Revoke Bias）后恢复到未锁定（标志位为“01”）或轻量级锁定（标志位为“00”）的状态，后续的同步操作就如上面介绍的轻量级锁那样执行。偏向锁、轻量级锁的状态转化及对象 Mark Word 的关系如图 13-5 所示。

偏向锁可以提高带有同步但无竞争的程序性能。它同样是一个带有效益权衡（Trade Off）性质的优化，也就是说它并不一定总是对程序运行有利，如果程序中大多数的锁都总是被多个不同的线程访问，那偏向模式就是多余的。在具体问题具体分析的前提下，有时候使用参数 `-XX:-UseBiasedLocking` 来禁止偏向锁优化反而可以提升性能。

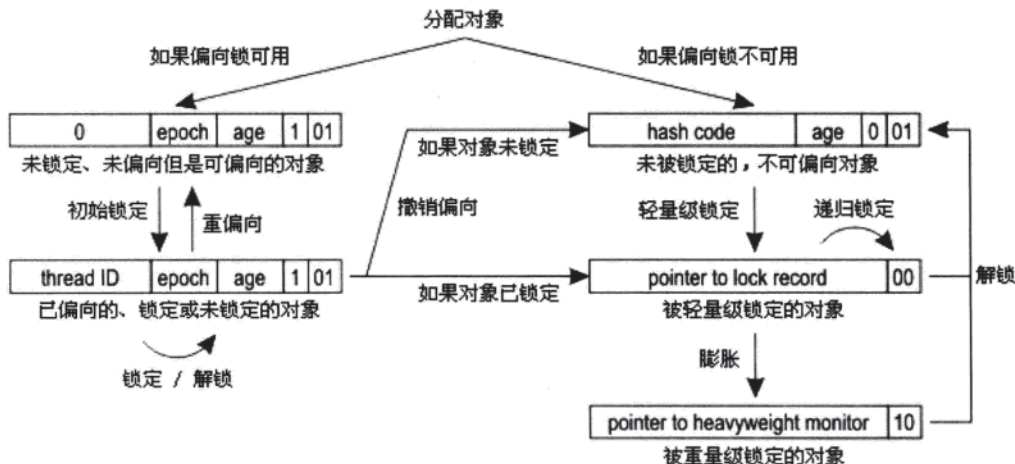


图 13-5 偏向锁、轻量级锁的状态转化及对象 Mark Word 的关系

13.4 本章小结

本章介绍了线程安全所涉及的概念和分类、同步实现的方式及虚拟机的底层运作原理，并且介绍了虚拟机实现高效并发所做的一系列锁优化措施。

许多老程序员都说过，能够写出高伸缩性的并发程序是一门艺术，而了解并发在系统底层是如何实现的，则是掌握这门艺术的前提条件，也是成长为高级程序员的必备知识之一。



附录 A Java 虚拟机家族

A.1 商用高性能虚拟机

□ Sun HotSpot

应用最为广泛的 Java 虚拟机，本书就是以这个虚拟机为平台进行讲解的。它最初由一家名为“Longview Technologies”的小公司开发，因为 HotSpot 的优异表现，这家公司在 1997 年被 Sun 公司收购了。HotSpot 虚拟机发布时是作为 Sun JDK 1.2 的附加程序提供的，后来它成为了 Sun JDK 1.3 及之后所有版本的 Sun JDK 的默认虚拟机。

□ BEA JRockit

JRockit 虚拟机是 BEA 公司于 2002 年从 Appeal Virtual Machines 收购获得的虚拟机。它是一款面向服务器硬件和服务端使用场景高度优化过的虚拟机，曾经号称是“世界上速度最快的虚拟机”。由于专注于服务端应用，它的内部不包含解析器的实现，全部代码都靠即时编译器编译后执行。

□ IBM J9

J9 虚拟机是 IBM 公司单独开发的高性能虚拟机，它并不独立出售，而是作为 IBM 公司各种产品的执行平台，IBM 把它定义为一个可以适应从嵌入式设备到大型企业级应用的、高可移植性的 Java 运行平台。

A.2 其他影响较大的虚拟机

□ Sun Classic 虚拟机

以现在的眼光来看，这个虚拟机很原始，几乎是实验室研究的作品。但它意义重大，是在 JDK 1.0 时代使用的 Java 虚拟机，是各种虚拟机的始祖。它的内部不存在即时编译器，只能使用纯解释的方式运行，如果要使用 JIT，必须通过外挂的方式插入额外的即时编译器（如 Symantec JIT 编译器），但插入后解释器就不再工作了。

□ Sun Exact 虚拟机

这是 Sun 公司在 HotSpot 之外的另一个 Java 虚拟机，在 JDK 1.2 时代曾短暂地投入过商用，它与 HotSpot 在同时期开发，但最终被 HotSpot 所取代。在它身上已经可以看见日后 Sun 虚拟机的模糊轮廓，如两级即时编译器、编译器与解释器混合工作模式、多层编译等。

□ Apache Harmony 虚拟机

Harmony 是 Apache 软件基金会主导的、开源的、独立的、实际兼容于 JDK 1.5 和 JDK 1.6 的虚拟机实现，它虽然不像前面两款虚拟机那样是 Java 混沌初生时开疆拓土的功臣，但论影响力一点也不逊色：它间接催生了 Google Android 平台的 Dalvik 虚拟机，Android 的影响力日益庞大，目前已是成功的数码设备通用平台；由于它的 TCK 授权问题，直接导致 Apache 与 Oracle 的决裂，从而退出了 JCP 组织，这是近代 Java 阵营遇到的最严重的分裂危机。

A.3 嵌入式虚拟机

□ Dalvik 虚拟机

Dalvik 虚拟机是 Google 等厂商合作开发的 Android 移动设备平台的核心组成部分之一，它的名字来源于冰岛一个名为 Dalvik 的小渔村。Dalvik 并不是一个 Java 虚拟机，它执行 dex (Dalvik Executable) 文件而不是 class 文件，使用寄存器架构而不是栈架构。但是它的开发体系与 Java 有着千丝万缕的关系：可以直接使用大部分的 Java API、dex 文件可以直接从 class 文件转化而来。目前此虚拟机随着 Android 一起处于迅猛发展的阶段，在 Android 2.2 中提供了即时编译器的实现，执行性能有了很大的提高。

□ KVM 虚拟机

KVM 中的 K 是“Kilobyte”的意思，它强调简单、轻量、高度可移植，但是运行速度比较慢。在 Android、iOS 等智能手机操作系统出现之前，曾广泛应用于手机平台。

□ CDC/CLDC HotSpot 虚拟机

也称为 phoneME Advanced/Feature 虚拟机，CDC 和 CLDC HotSpot 分别是 Sun 针对高端嵌入式设备和中低端嵌入式设备的虚拟机，用来代替 KVM。

A.4 其他虚拟机实现

- ❑ JavaInJava : <http://labs.oracle.com/kanban/JavaInJava.html>
- ❑ Maxine : <http://research.sun.com/projects/maxine/>
- ❑ JamVM : <http://jamvm.sourceforge.net/>
- ❑ cacaovm : <http://www.cacaovm.org/>
- ❑ SableVM : <http://www.sablevm.org/>
- ❑ Kaffe : <http://www.kaffe.org/>
- ❑ Jelatine JVM : <http://jelatine.sourceforge.net/>
- ❑ NanoVM : <http://www.harbaum.org/till/nanovm/index.shtml>
- ❑ MRP : <http://mrp.codehaus.org/>
- ❑ Moxie JVM : <http://moxie.sourceforge.net/>
- ❑ Jikes RVM : <http://jikesrvm.org/>



附录 B 虚拟机字节码指令表

字节码	助记符	指令含义
0x00	nop	什么都不做
0x01	aconst_null	将 null 推送至栈顶
0x02	iconst_m1	将 int 型 -1 推送至栈顶
0x03	iconst_0	将 int 型 0 推送至栈顶
0x04	iconst_1	将 int 型 1 推送至栈顶
0x05	iconst_2	将 int 型 2 推送至栈顶
0x06	iconst_3	将 int 型 3 推送至栈顶
0x07	iconst_4	将 int 型 4 推送至栈顶
0x08	iconst_5	将 int 型 5 推送至栈顶
0x09	lconst_0	将 long 型 0 推送至栈顶
0x0a	lconst_1	将 long 型 1 推送至栈顶
0x0b	fconst_0	将 float 型 0 推送至栈顶
0x0c	fconst_1	将 float 型 1 推送至栈顶
0x0d	fconst_2	将 float 型 2 推送至栈顶
0x0e	dconst_0	将 double 型 0 推送至栈顶
0x0f	dconst_1	将 double 型 1 推送至栈顶
0x10	bipush	将单字节的常量值 (128~127) 推送至栈顶
0x11	sipush	将一个短整型常量值 (32768~32767) 推送至栈顶
0x12	ldc	将 int、float 或 String 型常量值从常量池中推送至栈顶
0x13	ldc_w	将 int、float 或 String 型常量值从常量池中推送至栈顶 (宽索引)
0x14	ldc2_w	将 long 或 double 型常量值从常量池中推送至栈顶 (宽索引)
0x15	iload	将指定的 int 型本地变量推送至栈顶
0x16	lload	将指定的 long 型本地变量推送至栈顶
0x17	fload	将指定的 float 型本地变量推送至栈顶
0x18	dload	将指定的 double 型本地变量推送至栈顶
0x19	aload	将指定的引用类型本地变量推送至栈顶
0x1a	iload_0	将第一个 int 型本地变量推送至栈顶
0x1b	iload_1	将第二个 int 型本地变量推送至栈顶
0x1c	iload_2	将第三个 int 型本地变量推送至栈顶
0x1d	iload_3	将第四个 int 型本地变量推送至栈顶
0x1e	lload_0	将第一个 long 型本地变量推送至栈顶
0x1f	lload_1	将第二个 long 型本地变量推送至栈顶

(续)

字节码	助记符	指令含义
0x20	lload_2	将第三个 long 型本地变量推送至栈顶
0x21	lload_3	将第四个 long 型本地变量推送至栈顶
0x22	fload_0	将第一个 float 型本地变量推送至栈顶
0x23	fload_1	将第二个 float 型本地变量推送至栈顶
0x24	fload_2	将第三个 float 型本地变量推送至栈顶
0x25	fload_3	将第四个 float 型本地变量推送至栈顶
0x26	dload_0	将第一个 double 型本地变量推送至栈顶
0x27	dload_1	将第二个 double 型本地变量推送至栈顶
0x28	dload_2	将第三个 double 型本地变量推送至栈顶
0x29	dload_3	将第四个 double 型本地变量推送至栈顶
0x2a	aload_0	将第一个引用类型本地变量推送至栈顶
0x2b	aload_1	将第二个引用类型本地变量推送至栈顶
0x2c	aload_2	将第三个引用类型本地变量推送至栈顶
0x2d	aload_3	将第四个引用类型本地变量推送至栈顶
0x2e	iaload	将 int 型数组指定索引的值推送至栈顶
0x2f	laload	将 long 型数组指定索引的值推送至栈顶
0x30	faload	将 float 型数组指定索引的值推送至栈顶
0x31	daload	将 double 型数组指定索引的值推送至栈顶
0x32	aaload	将引用型数组指定索引的值推送至栈顶
0x33	baload	将 boolean 或 byte 型数组指定索引的值推送至栈顶
0x34	caload	将 char 型数组指定索引的值推送至栈顶
0x35	saload	将 short 型数组指定索引的值推送至栈顶
0x36	istore	将栈顶 int 型数值存入指定本地变量
0x37	lstore	将栈顶 long 型数值存入指定本地变量
0x38	fstore	将栈顶 float 型数值存入指定本地变量
0x39	dstore	将栈顶 double 型数值存入指定本地变量
0x3a	astore	将栈顶引用型数值存入指定本地变量
0x3b	istore_0	将栈顶 int 型数值存入第一个本地变量
0x3c	istore_1	将栈顶 int 型数值存入第二个本地变量
0x3d	istore_2	将栈顶 int 型数值存入第三个本地变量
0x3e	istore_3	将栈顶 int 型数值存入第四个本地变量
0x3f	lstore_0	将栈顶 long 型数值存入第一个本地变量
0x40	lstore_1	将栈顶 long 型数值存入第二个本地变量
0x41	lstore_2	将栈顶 long 型数值存入第三个本地变量
0x42	lstore_3	将栈顶 long 型数值存入第四个本地变量
0x43	fstore_0	将栈顶 float 型数值存入第一个本地变量
0x44	fstore_1	将栈顶 float 型数值存入第二个本地变量

(续)

字节码	助记符	指令含义
0x45	fstore_2	将栈顶 float 型数值存入第三个本地变量
0x46	fstore_3	将栈顶 float 型数值存入第四个本地变量
0x47	dstore_0	将栈顶 double 型数值存入第一个本地变量
0x48	dstore_1	将栈顶 double 型数值存入第二个本地变量
0x49	dstore_2	将栈顶 double 型数值存入第三个本地变量
0x4a	dstore_3	将栈顶 double 型数值存入第四个本地变量
0x4b	astore_0	将栈顶引用型数值存入第一个本地变量
0x4c	astore_1	将栈顶引用型数值存入第二个本地变量
0x4d	astore_2	将栈顶引用型数值存入第三个本地变量
0x4e	astore_3	将栈顶引用型数值存入第四个本地变量
0x4f	iastore	将栈顶 int 型数值存入指定数组的指定索引位置
0x50	lastore	将栈顶 long 型数值存入指定数组的指定索引位置
0x51	fastore	将栈顶 float 型数值存入指定数组的指定索引位置
0x52	dastore	将栈顶 double 型数值存入指定数组的指定索引位置
0x53	aastore	将栈顶引用型数值存入指定数组的指定索引位置
0x54	bastore	将栈顶 boolean 或 byte 型数值存入指定数组的指定索引位置
0x55	castore	将栈顶 char 型数值存入指定数组的指定索引位置
0x56	sastore	将栈顶 short 型数值存入指定数组的指定索引位置
0x57	pop	将栈顶数值弹出 (数值不能是 long 或 double 类型的)
0x58	pop2	将栈顶的一个 (对于 long 或 double 类型) 或两个数值 (对于非 long 或 double 的其他类型) 弹出
0x59	dup	复制栈顶数值并将复制值压入栈顶
0x5a	dup_x1	复制栈顶数值并将两个复制值压入栈顶
0x5b	dup_x2	复制栈顶数值并将三个 (或两个) 复制值压入栈顶
0x5c	dup2	复制栈顶一个 (对于 long 或 double 类型) 或两个 (对于非 long 或 double 的其他类型) 数值并将复制值压入栈顶
0x5d	dup2_x1	dup_x1 指令的双倍版本
0x5e	dup2_x2	dup_x2 指令的双倍版本
0x5f	swap	将栈最顶端的两个数值互换 (数值不能是 long 或 double 类型)
0x60	iadd	将栈顶两 int 型数值相加并将结果压入栈顶
0x61	ladd	将栈顶两 long 型数值相加并将结果压入栈顶
0x62	fadd	将栈顶两 float 型数值相加并将结果压入栈顶
0x63	dadd	将栈顶两 double 型数值相加并将结果压入栈顶
0x64	isub	将栈顶两 int 型数值相减并将结果压入栈顶
0x65	lsub	将栈顶两 long 型数值相减并将结果压入栈顶
0x66	fsub	将栈顶两 float 型数值相减并将结果压入栈顶
0x67	dsub	将栈顶两 double 型数值相减并将结果压入栈顶

(续)

字节码	助记符	指令含义
0x68	imul	将栈顶两 int 型数值相乘并将结果压入栈顶
0x69	lmul	将栈顶两 long 型数值相乘并将结果压入栈顶
0x6a	fmul	将栈顶两 float 型数值相乘并将结果压入栈顶
0x6b	dmul	将栈顶两 double 型数值相乘并将结果压入栈顶
0x6c	idiv	将栈顶两 int 型数值相除并将结果压入栈顶
0x6d	ldiv	将栈顶两 long 型数值相除并将结果压入栈顶
0x6e	fdiv	将栈顶两 float 型数值相除并将结果压入栈顶
0x6f	ddiv	将栈顶两 double 型数值相除并将结果压入栈顶
0x70	irem	将栈顶两 int 型数值作取模运算并将结果压入栈顶
0x71	lrem	将栈顶两 long 型数值作取模运算并将结果压入栈顶
0x72	frem	将栈顶两 float 型数值作取模运算并将结果压入栈顶
0x73	drem	将栈顶两 double 型数值作取模运算并将结果压入栈顶
0x74	ineg	将栈顶 int 型数值取负并将结果压入栈顶
0x75	lneg	将栈顶 long 型数值取负并将结果压入栈顶
0x76	fneg	将栈顶 float 型数值取负并将结果压入栈顶
0x77	dneg	将栈顶 double 型数值取负并将结果压入栈顶
0x78	ishl	将 int 型数值左移位指定位数并将结果压入栈顶
0x79	lshl	将 long 型数值左移位指定位数并将结果压入栈顶
0x7a	ishr	将 int 型数值右(带符号)移位指定位数并将结果压入栈顶
0x7b	lshr	将 long 型数值右(带符号)移位指定位数并将结果压入栈顶
0x7c	iushr	将 int 型数值右(无符号)移位指定位数并将结果压入栈顶
0x7d	lushr	将 long 型数值右(无符号)移位指定位数并将结果压入栈顶
0x7e	iand	将栈顶两 int 型数值作“按位与”并将结果压入栈顶
0x7f	land	将栈顶两 long 型数值作“按位与”并将结果压入栈顶
0x80	ior	将栈顶两 int 型数值作“按位或”并将结果压入栈顶
0x81	lor	将栈顶两 long 型数值作“按位或”并将结果压入栈顶
0x82	ixor	将栈顶两 int 型数值作“按位异或”并将结果压入栈顶
0x83	lxor	将栈顶两 long 型数值作“按位异或”并将结果压入栈顶
0x84	iinc	将指定 int 型变量增加指定值(如 i++, i--, i+=2 等)
0x85	i2l	将栈顶 int 型数值强制转换成 long 型数值并将结果压入栈顶
0x86	i2f	将栈顶 int 型数值强制转换成 float 型数值并将结果压入栈顶
0x87	i2d	将栈顶 int 型数值强制转换成 double 型数值并将结果压入栈顶
0x88	l2i	将栈顶 long 型数值强制转换成 int 型数值并将结果压入栈顶
0x89	l2f	将栈顶 long 型数值强制转换成 float 型数值并将结果压入栈顶
0x8a	l2d	将栈顶 long 型数值强制转换成 double 型数值并将结果压入栈顶
0x8b	f2i	将栈顶 float 型数值强制转换成 int 型数值并将结果压入栈顶
0x8c	f2l	将栈顶 float 型数值强制转换成 long 型数值并将结果压入栈顶

(续)

字节码	助记符	指令含义
0x8d	f2d	将栈顶 float 型数值强制转换成 double 型数值并将结果压入栈顶
0x8e	d2i	将栈顶 double 型数值强制转换成 int 型数值并将结果压入栈顶
0x8f	d2l	将栈顶 double 型数值强制转换成 long 型数值并将结果压入栈顶
0x90	d2f	将栈顶 double 型数值强制转换成 float 型数值并将结果压入栈顶
0x91	i2b	将栈顶 int 型数值强制转换成 byte 型数值并将结果压入栈顶
0x92	i2c	将栈顶 int 型数值强制转换成 char 型数值并将结果压入栈顶
0x93	i2s	将栈顶 int 型数值强制转换成 short 型数值并将结果压入栈顶
0x94	lcmp	比较栈顶两 long 型数值的大小, 并将结果 (1、0 或 -1) 压入栈顶
0x95	fcmpl	比较栈顶两 float 型数值的大小, 并将结果 (1、0 或 -1) 压入栈顶; 当其中一个数值为“NaN”时, 将 -1 压入栈顶
0x96	fcmpg	比较栈顶两 float 型数值的大小, 并将结果 (1、0 或 -1) 压入栈顶; 当其中一个数值为“NaN”时, 将 1 压入栈顶
0x97	dcmpl	比较栈顶两 double 型数值的大小, 并将结果 (1、0 或 -1) 压入栈顶; 当其中一个数值为“NaN”时, 将 -1 压入栈顶
0x98	dcmpg	比较栈顶两 double 型数值的大小, 并将结果 (1、0 或 -1) 压入栈顶; 当其中一个数值为“NaN”时, 将 1 压入栈顶
0x99	ifeq	当栈顶 int 型数值等于 0 时跳转
0x9a	ifne	当栈顶 int 型数值不等于 0 时跳转
0x9b	iflt	当栈顶 int 型数值小于 0 时跳转
0x9c	ifge	当栈顶 int 型数值大于或等于 0 时跳转
0x9d	ifgt	当栈顶 int 型数值大于 0 时跳转
0x9e	ifle	当栈顶 int 型数值小于或等于 0 时跳转
0x9f	if_icmpeq	比较栈顶两 int 型数值的大小, 当结果等于 0 时跳转
0xa0	if_icmpne	比较栈顶两 int 型数值的大小, 当结果不等于 0 时跳转
0xa1	if_icmplt	比较栈顶两 int 型数值的大小, 当结果小于 0 时跳转
0xa2	if_icmpge	比较栈顶两 int 型数值的大小, 当结果大于等于 0 时跳转
0xa3	if_icmpgt	比较栈顶两 int 型数值的大小, 当结果大于 0 时跳转
0xa4	if_icmple	比较栈顶两 int 型数值的大小, 当结果小于或等于 0 时跳转
0xa5	if_acmpeq	比较栈顶两引用型数值, 当结果相等时跳转
0xa6	if_acmpne	比较栈顶两引用型数值, 当结果不相等时跳转
0xa7	goto	无条件跳转
0xa8	jsr	跳转至指定的 16 位 offset 位置, 并将 jsr 的下一条指令地址压入栈顶
0xa9	ret	返回至本地变量指定的 index 的指令位置 (一般与 jsr 或 jsr_w 联合使用)
0xaa	tableswitch	用于 switch 条件跳转, case 值连续 (可变长度指令)
0xab	lookupswitch	用于 switch 条件跳转, case 值不连续 (可变长度指令)
0xac	ireturn	从当前方法返回 int
0xad	lreturn	从当前方法返回 long

(续)

字节码	助记符	指令含义
0xae	freturn	从当前方法返回 float
0xaf	dreturn	从当前方法返回 double
0xb0	areturn	从当前方法返回对象引用
0xb1	return	从当前方法返回 void
0xb2	getstatic	获取指定类的静态域, 并将其值压入栈顶
0xb3	putstatic	为指定的类的静态域赋值
0xb4	getfield	获取指定类的实例域, 并将其值压入栈顶
0xb5	putfield	为指定的类的实例域赋值
0xb6	invokevirtual	调用实例方法
0xb7	invokespecial	调用超类构造方法, 实例初始化方法, 私有方法
0xb8	invokestatic	调用静态方法
0xb9	invokeinterface	调用接口方法
0xba	--	无此指令
0xbb	new	创建一个对象, 并将其引用值压入栈顶
0xbc	newarray	创建一个指定的原始类型 (如 int、float、char 等) 的数组, 并将其引用值压入栈顶
0xbd	anewarray	创建一个引用型 (如类、接口、数组) 的数组, 并将其引用值压入栈顶
0xbe	arraylength	获得数组的长度值并压入栈顶
0xbf	athrow	将栈顶的异常抛出
0xc0	checkcast	检验类型转换, 检验未通过将抛出 ClassCastException
0xc1	instanceof	检验对象是否是指定的类的实例, 如果是将 1 压入栈顶, 否则将 0 压入栈顶
0xc2	monitorenter	获得对象的锁, 用于同步方法或同步块
0xc3	monitorexit	释放对象的锁, 用于同步方法或同步块
0xc4	wide	扩展本地变量的宽度
0xc5	multianewarray	创建指定类型和指定维度的多维数组 (执行该指令时, 操作栈中必须包含各维度的长度值), 并将其引用值压入栈顶
0xc6	ifnull	为 null 时跳转
0xc7	ifnonnull	不为 null 时跳转
0xc8	goto_w	无条件跳转 (宽索引)
0xc9	jsr_w	跳转至指定的 32 位 offset 位置, 并将 jsr_w 的下一条指令地址压入栈顶

附录 C HotSpot 虚拟机主要参数表

本参数表以 JDK 1.6 为基础编写，JDK 1.6 的 HotSpot 虚拟机有很多非稳定参数（Unstable Options，即以 -XX: 开头的参数，JDK 1.6 的虚拟机中大概有 660 多个），使用 -XX:+PrintFlagsFinal 参数可以输出所有参数的名称及默认值，下面的各个表格只包含了其中最常用的（或在本书中介绍到的）部分。参数使用的方式有如下三种：

- -XX:+<option> 开启 option 参数。
- -XX:-<option> 关闭 option 参数。
- -XX: <option>=<value> 将 option 参数的值设置为 value。

C.1 内存管理参数

参 数	默认值	使用介绍
DisableExplicitGC	默认关闭	忽略来自 System.gc() 方法触发的垃圾收集
ExplicitGCInvokesConcurrent	默认关闭	当收到 System.gc() 方法提交的垃圾收集申请时，使用 CMS 收集器进行收集
UseSerialGC	Client 模式的虚拟机默认开启，其他模式关闭	虚拟机运行在 Client 模式下的默认值，打开此开关后，使用 Serial + Serial Old 的收集器组合进行内存回收
UseParNewGC	默认关闭	打开此开关后，使用 ParNew + Serial Old 的收集器组合进行内存回收
UseConcMarkSweepGC	默认关闭	打开此开关后，使用 ParNew + CMS + Serial Old 的收集器组合进行内存回收。如果 CMS 收集器出现 Concurrent Mode Failure，则 Serial Old 收集器将作为后备收集器
UseParallelGC	Server 模式的虚拟机默认开启，其他模式关闭	虚拟机运行在 Server 模式下的默认值，打开此开关后，使用 Parallel Scavenge + Serial Old 的收集器组合进行内存回收
UseParallelOldGC	默认关闭	打开此开关后，使用 Parallel Scavenge + Parallel Old 的收集器组合进行内存回收
SurvivorRatio	默认为 8	新生代中 Eden 区域与 Survivor 区域的容量比值

(续)

参 数	默认值	使用介绍
PretenureSizeThreshold	无默认值	直接晋升到老年代的对象大小, 设置这个参数后, 大于这个参数的对象将直接在老年代分配
MaxTenuringThreshold	默认值为 15	晋升到老年代的对象年龄。每个对象在坚持过一次 Minor GC 之后, 年龄就加 1, 当超过这个参数值时就进入老年代
UseAdaptiveSizePolicy	默认开启	动态调整 Java 堆中各个区域的大小及进入老年代的年龄
HandlePromotionFailure	JDK 1.5 及以前是默认关闭, JDK 1.6 默认开启	是否允许分配担保失败, 即老年代的剩余空间不足以应付新生代的整个 Eden 和 Survivor 区的所有对象都存活的极端情况
ParallelGCThreads	少于或等于 8 个 CPU 时默认值为 CPU 数量值, 多于 8 个时比 CPU 数量值小	设置并行 GC 时进行内存回收的线程数
GCTimeRatio	默认值为 99	GC 时间占总时间的比率, 默认值为 99, 即允许 1% 的 GC 时间。仅在使用 Parallel Scavenge 收集器时生效
MaxGCPauseMillis	无默认值	设置 GC 的最大停顿时间。仅在使用 Parallel Scavenge 收集器时生效
CMSInitiatingOccupancyFraction	默认值为 68	设置 CMS 收集器在老年代空间被使用多少后触发垃圾收集, 仅在使用 CMS 收集器时生效
UseCMSCompactAtFullCollection	默认开启	设置 CMS 收集器在完成垃圾收集后是否要进行一次内存碎片整理。仅在使用 CMS 收集器时生效
CMSFullGCsBeforeCompaction	无默认值	设置 CMS 收集器在进行若干次垃圾收集后再启动一次内存碎片整理。仅在使用 CMS 收集器时生效
ScavengeBeforeFullGC	默认开启	在 Full GC 发生之前触发一次 Minor GC
UseGCOverheadLimit	默认开启	禁止 GC 过程无限制地执行, 如果过于频繁, 就直接发生 OutOfMemory 异常
UseTLAB	Server 模式默认开启	优先在本地线程缓冲区中分配对象, 避免分配内存时的锁定过程
MaxHeapFreeRatio	默认值为 70	当 Xmx 值比 Xms 值大时, 堆可以动态收缩和扩展, 这个参数控制当堆空闲大于指定比率时自动收缩
MinHeapFreeRatio	默认值为 40	当 Xmx 值比 Xms 值大时, 堆可以动态收缩和扩展, 这个参数控制当堆空闲小于指定比率时自动扩展
MaxPermSize	大部分情况下默认值是 64MB	永久代的最大值

C.2 即时编译参数

参 数	默认值	使用介绍
CompileThreshold	Client 模式下默认值是 1500, Server 模式下是 10000	触发方法即时编译的阈值
OnStackReplacePercentage	Client 模式下默认值是 933, Server 模式下是 140	OSR 比率, 它是 OSR 即时编译阈值计算公式的一个参数, 用于代替 BackEdgeThreshold 参数控制回边计数器的实际溢出阈值
ReservedCodeCacheSize	大部分情况下默认值是 32MB	即时编译器编译的代码缓存的最大值

C.3 类型加载参数

参 数	默认值	使用介绍
UseSplitVerifier	默认开启	使用依赖 StackMapTable 信息的类型检查代替数据流分析, 以加快字节码校验速度
FailOverToOldVerifier	默认开启	当类型校验失败时, 是否允许回到老的类型推导校验方式进行校验, 如果开启则允许
RelaxAccessControlCheck	默认关闭	在校验阶段放松对类型访问性的限制

C.4 多线程相关参数

参 数	默认值	使用介绍
UseSpinning	JDK 1.6 默认开启, JDK 1.5 默认关闭	开启自旋锁以避免线程频繁的挂起和唤醒
PreBlockSpin	默认值为 10	使用自旋锁时默认的自旋次数
UseThreadPriorities	默认开启	使用本地线程优先级
UseBiasedLocking	默认开启	是否使用偏向锁, 如果开启则使用
UseFastAccessorMethods	默认开启	当频繁反射执行某个方法时, 生成字节码来加快反射的执行速度

C.5 性能参数

参 数	默认值	使用介绍
AggressiveOpts	JDK 1.6 默认开启, JDK 1.5 默认关闭	使用激进的优化特性, 这些特性一般是具备正面和负面双重影响的, 需要根据具体应用特点分析才能判定是否对性能有好处
UseLargePages	默认开启	如果可能, 使用大内存分页, 这项特性需要操作系统的支持
LargePageSizeInBytes	默认为 4MB	使用指定大小的内存分页, 这项特性需要操作系统的支持
StringCache	默认开启	是否使用字符串缓存, 开启则使用

C.6 调试参数

参 数	默认值	使用介绍
HeapDumpOnOutOfMemoryError	默认关闭	在发生内存溢出异常时是否生成堆转储快照, 关闭则不生成
OnOutOfMemoryError	无默认值	当虚拟机抛出内存溢出异常时, 执行指定的命令
OnError	无默认值	当虚拟机抛出 ERROR 异常时, 执行指定的命令
PrintClassHistogram	默认关闭	使用 [ctrl]-[break] 快捷键输出类统计状态, 相当于 jmap -histo 的功能
PrintConcurrentLocks	默认关闭	打印 J.U.C 中锁的状态
PrintCommandLineFlags	默认关闭	打印启动虚拟机时输入的非稳定参数
PrintCompilation	默认关闭	打印方法即时编译信息
PrintGC	默认关闭	打印 GC 信息
PrintGCDetails	默认关闭	打印 GC 的详细信息
PrintGCTimeStamps	默认关闭	打印 GC 停顿耗时
PrintTenuringDistribution	默认关闭	打印 GC 后新生代各个年龄对象的大小
TraceClassLoading	默认关闭	打印类加载信息
TraceClassUnloading	默认关闭	打印类卸载信息
PrintInlining	默认关闭	打印方法的内联信息
PrintCFGToFile	默认关闭	将 CFG 图信息输出到文件, 只有 DEBUG 版虚拟机才支持此参数
PrintIdealGraphFile	默认关闭	将 Ideal 图信息输出到文件, 只有 DEBUG 版虚拟机才支持此参数
UnlockDiagnosticVMOptions	默认关闭	让虚拟机进入诊断模式, 一些参数 (如 PrintAssembly) 需要在诊断模式中才能使用
PrintAssembly	默认关闭	打印即时编译后的二进制信息

附录 D 对象查询语言 (OQL) 简介^①

D.1 SELECT 子句

SELECT 子句用于确定查询语句需要从堆转储快照中选择什么内容。如果需要显示堆转储快照中的对象，并且浏览这些对象的引用关系，可以使用“*”，这与传统 SQL 语句中的习惯是一致的，例如：

```
SELECT * FROM java.lang.String
```

D.1.1 选择特定的显示列

查询也可以选择特定的需要显示的字段，例如：

```
SELECT toString(s), s.count, s.value FROM java.lang.String s
```

查询可以用“@”符号来使用 Java 对象的内存属性访问器。MAT 提供了一系列的内置函数来获取与分析相关的信息，例如：

```
SELECT toString(s), s.@usedHeapSize, s.@retainedHeapSize FROM java.lang.String s
```

关于对象属性访问器的具体内容，可以参见下文的“属性访问器”。

D.1.2 使用列别名

可以使用 AS 关键字来对选择的列进行命名，例如：

```
SELECT toString(s) AS Value,  
       s.@usedHeapSize AS "Shallow Size",  
       s.@retainedHeapSize AS "Retained Size"  
FROM java.lang.String s
```

可以使用“AS RETAINED SET”关键字来获得与选择对象相关联的对象集合，例如：

^① 本附录翻译自 Eclipse Memory Analyzer Tool (MAT, Eclipse 出品的内存分析工具) 的 OQL 帮助文档。

```
SELECT AS RETAINED SET * FROM java.lang.String
```

D.1.3 拼合成为一个对象列表选择项目

可以使用“OBJECTS”关键字把 SELECT 子句中查找出来的数据项目转变为对象，例如：

```
SELECT OBJECTS dominators(s) FROM java.lang.String s
```

上面的例子中，函数“dominators()”将会返回一个对象数组，所以如果没有“OBJECTS”关键字，上面的查询将返回一组二维的对象数组列表。通过使用关键字“OBJECTS”迫使 OQL 把查询结果缩减为一维的对象列表。

D.1.4 排除重复对象

使用“DISTINCT”关键字可以排除结果集中的重复对象，例如：

```
SELECT DISTINCT classof(s) FROM java.lang.String s
```

上面的例子中，函数“classof()”的作用是返回对象所属的 Java 类，当然，所有字符串对象的所属类都是 java.lang.String，所以如果上面的查询中没有加入 DISTINCT 关键字，查询结果就会返回与快照中的字符串数量一样多的行记录，并且每行记录的内容都是 java.lang.String 类型。

D.2 FROM 子句

D.2.1 FROM 子句指定需要查询的类

OQL 查询需要在 FROM 子句定义的查询范围上进行操作。FROM 子句可以接受的查询范围描述包括如下几种方式：

1) 通过类名进行查询，例如：

```
SELECT * FROM java.lang.String
```

2) 通过正则表达式匹配一组类名进行查询，例如：

```
SELECT * FROM "java\.lang\..*"
```



378 ❖ 附录 D 对象查询语言 (OQL) 简介

3) 通过类对象在堆转储快照中的地址进行查询, 例如:

```
SELECT * FROM 0xe14a100
```

4) 通过对象在堆转储快照中的 ID 进行查询, 例如:

```
SELECT * FROM 3022
```

5) 在子查询中的结果集中进行查询, 例如:

```
SELECT * FROM (SELECT * FROM java.lang.Class c WHERE c implements org.eclipse.mat.snapshot.model.IClass)
```

上面的查询返回堆转储快照中所有实现了“org.eclipse.mat.snapshot.model.IClass”接口的类。下面的这句查询使用属性访问器达到了同样的效果, 它直接调用了 ISnapshot 对象的方法:

```
SELECT * FROM $snapshot.getClasses()
```

D.2.2 包含子类

使用“INSTANCEOF”关键字把指定类的子类列入查询结果集之中, 例如:

```
SELECT * FROM INSTANCEOF java.lang.ref.Reference
```

这个查询的结果集中将会包含 WeakReference、SoftReference 和 PhantomReference 类型的对象, 因为它们都继承自 java.lang.ref.Reference。下面这句查询也有相同的结果:

```
SELECT * FROM $snapshot.getClassesByName("java.lang.ref.Reference", true)
```

D.2.3 禁止查询类实例

在 FROM 子句中使用“OBJECTS”关键字可以禁止 OQL 把查询的范围解释为对象实例, 例如:

```
SELECT * FROM OBJECTS java.lang.String
```

这个查询的结果不是返回快照中所有的字符串, 而是只有一个对象, 也就是 java.lang.String 类对应的 Class 对象。

D.3 WHERE 子句

D.3.1 >=、<=、>、<、[NOT] LIKE、[NOT] IN (关系操作)

WHERE 子句用于指定搜索的条件，即从查询结果中删除不需要的数据，例如：

```
SELECT * FROM java.lang.String s WHERE s.count >= 100
SELECT * FROM java.lang.String s WHERE toString(s) LIKE ".*day"
SELECT * FROM java.lang.String s WHERE s.value NOT IN dominators(s)
```

D.3.2 =、!= (等于操作)

```
SELECT * FROM java.lang.String s WHERE toString(s) = "monday"
```

D.3.3 AND (条件与操作)

```
SELECT * FROM java.lang.String s WHERE s.count > 100 AND s.@retainedHeapSize > s.@usedHeapSize
```

D.3.4 OR (条件或操作)

“条件或操作”可以应用于表达式、常量文本和子查询之中，例如：

```
SELECT * FROM java.lang.String s WHERE s.count > 1000 OR s.value.@length > 1000
```

D.3.5 文字表达式

文字表达式包括了布尔值、字符串、整型、长整型和 null，例如：

```
SELECT * FROM java.lang.String s
    WHERE ( s.count > 1000 ) = true
    WHERE toString(s) = "monday"
    WHERE dominators(s).size() = 0
    WHERE s.@retainedHeapSize > 1024L
    WHERE s.@GCRootInfo != null
```

D.4 属性访问器

D.4.1 访问堆转储快照中对象的字段

对象的内存属性可以通过传统的“点表示法”进行访问，格式为：



380 ❖ 附录 D 对象查询语言 (OQL) 简介

```
[<alias>.] <field>.<field>.<field>...
```

D.4.2 访问 Java Bean 属性

格式为：

```
[<alias>.] @<attribute> ...
```

使用 @ 符号，OQL 可以访问底层 Java 对象的内存属性。下表列出了一些常用的 Java 属性。

目标	接口	属性	含义
任意堆中的对象	Iobject	objectId	快照中对象的 ID
		objectAddress	快照中对象的地址
		Class	对象所属的类
		usedHeapSize	对象的 ShallowSize
		retainedHeapSize	对象的 RetainedSize
		displayName	对象的显示名称
类对象	Iclass	classLoaderId	类加载器的 ID
任意数组	Iarray	length	数组的长度

D.4.3 调用 OQL Java 方法

格式为：

```
[<alias>.]@<方法>([<表达式>, <表达式>])...
```

加“()”将会令 MAT 解释为一个 OQL Java 方法调用。这个方法的调用是通过反射执行的。常见的 OQL Java 方法如下：

目标	接口	方法	含义
Snapshot	Isnapshot	getClasses()	获取所有类的集合
		getClassesByName(String name, boolean includeSubClasses)	获取指定类的集合
Class object	Iclass	hasSuperClass()	如果对象有父类，则返回 true
		isArrayType()	如果 Class 是数组类型，则返回 true

D.4.4 OQL 的内建函数

格式为：

<function> (<parameter>)

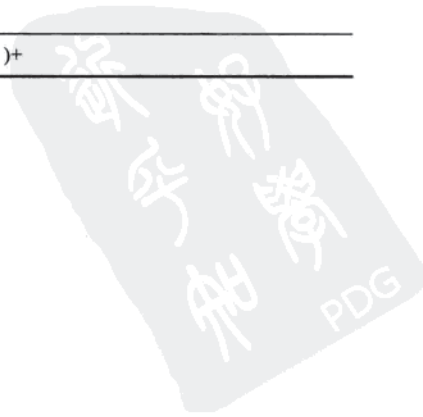
函数名称	作用
toHex(number)	以十六进制的形式打印数字
toString(object)	返回对象的值，即使用一个字符串表示对象的内容
dominators(object)	返回直接持有指定对象的对象列表
outbounds(object)	获取对象的外部引用
inbounds(object)	获取对象的内部引用
classof(object)	获取对象所属的类型对象
dominatorof(object)	返回直接持有当前对象的对象列表，如果没有则返回 -1

D.5 OQL 语言的 BNF 范式

目标	接口	方法
SelectStatement	::=	"SELECT" SelectList FromClause (WhereClause)? (UnionClause)?
SelectList	::=	(("DISTINCT" "AS RETAINED SET")? ("*" "OBJECTS" SelectItem SelectItem ("," SelectItem)*))
SelectItem	::=	(PathExpression EnvVarPathExpression) ("AS" (<STRING_LITERAL> <IDENTIFIER>))? <IDENTIFIER> <STRING_LITERAL>
PathExpression	::=	(ObjectFacet BuildInFunction) ("." ObjectFacet)*
EnvVarPathExpression	::=	("\$" <IDENTIFIER>) ("." ObjectFacet)*
ObjectFacet	::=	("@")? <IDENTIFIER> (ParameterList)?
ParameterList	::=	"(" (PrimaryExpression ("," PrimaryExpression)*))? "
FromClause	::=	"FROM" ("OBJECTS")? ("INSTANCEOF")? (FromItem "(SelectStatement)") (<IDENTIFIER>)?
FromItem	::=	(ClassName <STRING_LITERAL> ObjectAddress ("," ObjectAddress)* ObjectId ("," ObjectId)* EnvVarPathExpression)
ClassName	::=	(<IDENTIFIER> ("." <IDENTIFIER>)* ("["])?)
ObjectAddress	::=	<HEX_LITERAL>
ObjectId	::=	<INTEGER_LITERAL>

(续)

目 标	接 口	方 法
WhereClause	::=	"WHERE" ConditionalOrExpression
ConditionalOrExpression	::=	ConditionalAndExpression ("or" ConditionalAndExpression)*
ConditionalAndExpression	::=	EqualityExpression ("and" EqualityExpression)*
EqualityExpression	::=	RelationalExpression (("=" RelationalExpression "!=" RelationalExpression))*
RelationalExpression	::=	(PrimaryExpression (("<" PrimaryExpression ">" PrimaryExpression "<=" PrimaryExpression ">=" PrimaryExpression (LikeClause InClause) "implements" ClassName)))?
LikeClause	::=	("NOT")? "LIKE" <STRING_LITERAL>
InClause	::=	("NOT")? "IN" PrimaryExpression
PrimaryExpression	::=	Literal
		"(" (ConditionalOrExpression SubQuery) ")
		PathExpression
		EnvVarPathExpression
SubQuery	::=	SelectStatement
Function	::=	(("toHex" "toString" "dominators" "outbounds" "inbounds" "classof" "dominatorof") "(" ConditionalOrExpression ")")
Literal	::=	(<INTEGER_LITERAL> <LONG_LITERAL> <FLOATING_POINT_LITERAL> <CHARACTER_LITERAL> <STRING_LITERAL> BooleanLiteral NullLiteral)
BooleanLiteral	::=	"true"
		"false"
NullLiteral	::=	<NULL>
UnionClause	::=	("UNION" "(" SelectStatement ")")+



附录 E JDK 历史版本轨迹

大部分的 JDK 历史版本（JDK 1.1.6 之后的版本），以及 JDK 附带的各种工具的历史版本，都可以从 Oracle 公司的网站^①上下载到。

主版本	子版本及虚拟机版本	工程代号	发布日期
JDK 1.0	JDK 1.0		1996-01-23
	JDK 1.0.1		
	JDK 1.0.2		
JDK 1.1	JDK 1.1.0		1997-02-18
	JDK 1.1.1		
	JDK 1.1.2		
	JDK 1.1.3		
	JDK 1.1.4	Sparkler	1997-09-12
	JDK 1.1.5	Pumpkin	1997-12-03
	JDK 1.1.6	Abigail	1998-04-24
	JDK 1.1.7	Brutus	1998-09-28
	JDK 1.1.8	Chelsea	1999-04-08
JDK 1.2	JDK 1.2.0	Playground	1998-12-04
	JDK 1.2.1	(none)	1999-03-30
	JDK 1.2.2	Cricket	1999-07-08
JDK 1.3	JDK 1.3.0 (HotSpot 1.3.0-C)	Kestrel	2000-05-08
	JDK 1.3.0 Update 1 (HotSpot 1.3.0_01)		
	JDK 1.3.0 Update 2 (HotSpot 1.3.0_02)		
	JDK 1.3.0 Update 3 (HotSpot 1.3.0_03)		
	JDK 1.3.0 Update 4 (HotSpot 1.3.0_04)		
	JDK 1.3.0 Update 5 (HotSpot 1.3.0_05)		
	JDK 1.3.1 (HotSpot 1.3.1)	Ladybird	2001-05-17

① 下载页面地址：<http://www.oracle.com/technetwork/java/archive-139210.html>。

(续)

主版本	子版本及虚拟机版本	工程代号	发布日期
JDK 1.3	JDK 1.3.1 Update 1 (HotSpot 1.3.1_01)		
	JDK 1.3.1 Update 1a (HotSpot 1.3.1_01a)		
	JDK 1.3.1 Update 2 (HotSpot 1.3.1_02)		
	JDK 1.3.1 Update 3 (HotSpot 1.3.1_03)		
	JDK 1.3.1 Update 4 (HotSpot 1.3.1_04)		
	JDK 1.3.1 Update 5 (HotSpot 1.3.1_05)		
	JDK 1.3.1 Update 6 (HotSpot 1.3.1_06)		
	JDK 1.3.1 Update 7 (HotSpot 1.3.1_07)		
	JDK 1.3.1 Update 8 (HotSpot 1.3.1_08)		
	JDK 1.3.1 Update 9 (HotSpot 1.3.1_09)		
	JDK 1.3.1 Update 10 (HotSpot 1.3.1_10)		
	JDK 1.3.1 Update 11 (HotSpot 1.3.1_11)		
	JDK 1.3.1 Update 12 (HotSpot 1.3.1_12)		
JDK 1.4	JDK 1.4.0 (HotSpot 1.4.0)	Merlin	2002-02-13
	JDK 1.4.0 Update 1 (HotSpot 1.4.0_01)		
	JDK 1.4.0 Update 2 (HotSpot 1.4.0_02)		
	JDK 1.4.0 Update 3 (HotSpot 1.4.0_03)		
	JDK 1.4.0 Update 4 (HotSpot 1.4.0_04)		
	JDK 1.4.1 (HotSpot 1.4.1)	Grasshopper	2002-09-16
	JDK 1.4.1 Update 1 (HotSpot 1.4.1_01)		
	JDK 1.4.1 Update 2 (HotSpot 1.4.1_02)		
	JDK 1.4.1 Update 3 (HotSpot 1.4.1_03)		
	JDK 1.4.1 Update 4 (HotSpot 1.4.1_04)		
	JDK 1.4.1 Update 5 (HotSpot 1.4.1_05)		
	JDK 1.4.1 Update 6 (HotSpot 1.4.1_06)		
	JDK 1.4.1 Update 7 (HotSpot 1.4.1_07)		
	JDK 1.4.2 (HotSpot 1.4.2-b28)	Mantis	2003-06-26
	JDK 1.4.2 Update 1 (HotSpot 1.4.2_01)		
	JDK 1.4.2 Update 2 (HotSpot 1.4.2_02)		
JDK 1.4.2 Update 3 (HotSpot 1.4.2_03)			

(续)

主版本	子版本及虚拟机版本	工程代号	发布日期
JDK 1.4	JDK 1.4.2 Update 4 (HotSpot 1.4.2_04)		
	JDK 1.4.2 Update 5 (HotSpot 1.4.2_05)		
	JDK 1.4.2 Update 6 (HotSpot 1.4.2_06)		
	JDK 1.4.2 Update 7 (HotSpot 1.4.2_07)		
	JDK 1.4.2 Update 8 (HotSpot 1.4.2_08-b03)		
	JDK 1.4.2 Update 9 (HotSpot 1.4.2_09-b05)		
	JDK 1.4.2 Update 10 (HotSpot 1.4.2_10-b03)		
	JDK 1.4.2 Update 11 (HotSpot 1.4.2_11-b06)		
	JDK 1.4.2 Update 12 (HotSpot 1.4.2_12-b03)		
	JDK 1.4.2 Update 13 (HotSpot 1.4.2_13-b03)		
	JDK 1.4.2 Update 14 (HotSpot 1.4.2_14-b05)		
	JDK 1.4.2 Update 15 (HotSpot 1.4.2_15-b02)		
	JDK 1.4.2 Update 16 (HotSpot 1.4.2_16-b01)		
	JDK 1.4.2 Update 17 (HotSpot 1.4.2_17-b06)		
	JDK 1.4.2 Update 18 (HotSpot 1.4.2_18-b06)		
	JDK 1.4.2 Update 19 (HotSpot 1.4.2_19-b04)		
JDK 1.5	JDK 1.5.0 (HotSpot 1.5.0-b64)	Tiger	2004-09-29
	JDK 1.5.0 Update 1 (HotSpot 1.5.0_01)		
	JDK 1.5.0 Update 2 (HotSpot 1.5.0_02-b09)		
	JDK 1.5.0 Update 3 (HotSpot 1.5.0_03-b07)		
	JDK 1.5.0 Update 4 (HotSpot 1.5.0_04-b05)		
	JDK 1.5.0 Update 5 (HotSpot 1.5.0_05-b05)		
	JDK 1.5.0 Update 6 (HotSpot 1.5.0_06-b05)		
	JDK 1.5.0 Update 7 (HotSpot 1.5.0_07-b03)		
	JDK 1.5.0 Update 8 (HotSpot 1.5.0_08-b03)		
	JDK 1.5.0 Update 9 (HotSpot 1.5.0_09-b03)		
	JDK 1.5.0 Update 10 (HotSpot 1.5.0_10-b02)		
	JDK 1.5.0 Update 11 (HotSpot 1.5.0_11-b03)		
	JDK 1.5.0 Update 12 (HotSpot 1.5.0_12-b04)		
JDK 1.5.0 Update 13 (HotSpot 1.5.0_13-b01)			

(续)

主版本	子版本及虚拟机版本	工程代号	发布日期
JDK 1.5	JDK 1.5.0 Update 14 (HotSpot 1.5.0_14-b03)		
	JDK 1.5.0 Update 15 (HotSpot 1.5.0_15-b04)		
	JDK 1.5.0 Update 16 (HotSpot 1.5.0_16-b02)		
	JDK 1.5.0 Update 17 (HotSpot 1.5.0_17-b04)		
	JDK 1.5.0 Update 18 (HotSpot 1.5.0_18-b02)		
	JDK 1.5.0 Update 19 (HotSpot 1.5.0_19-b02)		
	JDK 1.5.0 Update 20 (HotSpot 1.5.0_20-b02)		
	JDK 1.5.0 Update 21 (HotSpot 1.5.0_21-b01)		
	JDK 1.5.0 Update 22 (HotSpot 1.5.0_22-b03)		
	JDK 1.5.1	Dragonfly	取消发布
JDK 1.6	JDK 1.6.0 (HotSpot 1.6.0-b105)	Mustang	2006-12-11
	JDK 1.6.0 Update 1 (HotSpot 1.6.0_01-b06)		
	JDK 1.6.0 Update 2		
	JDK 1.6.0 Update 3		
	JDK 1.6.0 Update 4 (HotSpot 10.0-b19)		
	JDK 1.6.0 Update 5 (HotSpot 10.0-b19)		
	JDK 1.6.0 Update 6		
	JDK 1.6.0 Update 7 (HotSpot 10.0-b23)		
	JDK 1.6.0 Update 10 (HotSpot 11.0-b15)		
	JDK 1.6.0 Update 11		
	JDK 1.6.0 Update 12		
	JDK 1.6.0 Update 13 (HotSpot 11.3-b02)		
	JDK 1.6.0 Update 14 (HotSpot 14.0-b16)		2009-05-28
	JDK 1.6.0 Update 15 (HotSpot 14.1-b02)		
	JDK 1.6.0 Update 16 (HotSpot 14.2-b01)		
	JDK 1.6.0 Update 17 (HotSpot 14.3-b01)		
	JDK 1.6.0 Update 18 (HotSpot 16.0-b13)		
	JDK 1.6.0 Update 19 (HotSpot 16.2-b04)		
	JDK 1.6.0 Update 20 (HotSpot 16.3-b01)		
	JDK 1.6.0 Update 21 (HotSpot 17.0-b17)		2010-07-10

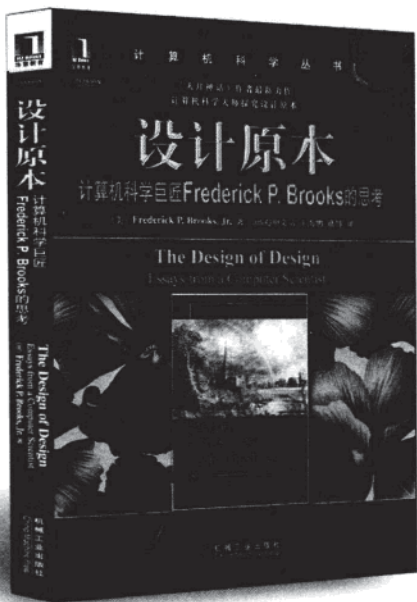
(续)

主版本	子版本及虚拟机版本	工程代号	发布日期
JDK 1.6	JDK 1.6.0 Update 22 (HotSpot17.1-b03)		2010-10-22
	JDK 1.6.0 Update 23 (HotSpot 19.0-b09)		2010-12-08
JDK 1.7	JDK 1.7.0	Dolphin	2011-07-28

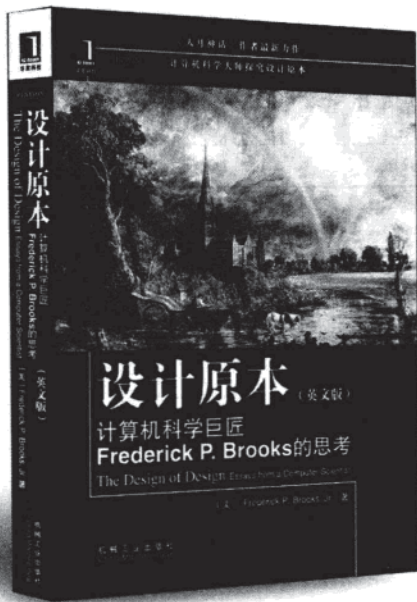




一本打开的书。
一扇开启的门。
通向科学圣殿的阶梯。
托起一流人才的基石。



ISBN: 978-7-111-32557-4
定价: 55.00



ISBN: 978-7-111-32503-1
定价: 69.00

《人月神话》作者最新力作 计算机科学大师探究设计原本

本书包含了多个行业设计者的特别领悟。Frederick P. Brooks, Jr.精确发现了所有设计项目中内在的不变因素，揭示了进行优秀设计的过程和模式。通过与几十位优秀设计者的对话，以及他自己在几个设计领域的经验，作者指出，大胆的设计决定会产生更好的结果。

本书几乎涵盖所有有关设计的议题：从设计哲学谈到设计实践，从设计过程到设计灵感，既强调了设计思想的重要性，又对沟通中的种种细节都做了细致入微的描述，并且谈到了因地制宜做出妥协的具体准则。



深入理解计算机系统 (原书第2版)
ISBN: 978-7-111-32133-0
定价: 99.00



深入理解计算机系统 (英文版·第2版)
ISBN: 978-7-111-32631-1
定价: 128.00



Linux内核设计与实现 (英文版·第3版)
ISBN: 978-7-111-32792-9
定价: 69.00



Linux内核设计与实现 (原书第3版)
ISBN: 978-7-111-33829-1
定价: 69.00元

好书推荐



作者: Bruce Eckel
书号: 978-7-111-21382-6
定价: 108.00元



作者: Joshua Bloch
书号: 978-7-111-25583-3
定价: 52.00元



作者: Gary Cornell
书号: 978-7-111-23950-5
定价: 98.00元



作者: Gary Cornell
书号: 978-7-111-25611-3
定价: 118.00元



专业成就人生
立体服务大众

www.hzbook.com

填写读者调查表 加入华章书友会
获赠精彩技术书 参与活动和抽奖

尊敬的读者：

感谢您选择华章图书。为了聆听您的意见，以便我们能够为您提供更优秀的图书产品，敬请您抽出宝贵的时间填写本表，并按底部的地址邮寄给我们（您也可通过www.hzbook.com填写本表）。您将加入我们的“华章书友会”，及时获得新书资讯，免费参加书友会活动。我们将定期选出若干名热心读者免费赠送我们出版的图书。请一定填写书名书号并留全您的联系信息，以便我们联络您，谢谢！

书名： _____ 书号： 7-111-(_____)

姓名：	性别： <input type="checkbox"/> 男 <input type="checkbox"/> 女	年龄：	职业：
通信地址：	E-mail：		
电话：	手机：	邮编：	

1. 您是如何获知本书的：

朋友推荐 书店 图书目录 杂志、报纸、网络等 其他

2. 您从哪里购买本书：

新华书店 计算机专业书店 网上书店 其他

3. 您对本书的评价是：

技术内容	<input type="checkbox"/> 很好	<input type="checkbox"/> 一般	<input type="checkbox"/> 较差	<input type="checkbox"/> 理由_____
文字质量	<input type="checkbox"/> 很好	<input type="checkbox"/> 一般	<input type="checkbox"/> 较差	<input type="checkbox"/> 理由_____
版式封面	<input type="checkbox"/> 很好	<input type="checkbox"/> 一般	<input type="checkbox"/> 较差	<input type="checkbox"/> 理由_____
印装质量	<input type="checkbox"/> 很好	<input type="checkbox"/> 一般	<input type="checkbox"/> 较差	<input type="checkbox"/> 理由_____
图书定价	<input type="checkbox"/> 太高	<input type="checkbox"/> 合适	<input type="checkbox"/> 较低	<input type="checkbox"/> 理由_____

4. 您希望我们的图书在哪些方面进行改进？

5. 您最希望我们出版哪方面的图书？如果有英文版请写出书名。

6. 您有没有写作或翻译技术图书的想法？

是，我的计划是_____ 否

7. 您希望获取图书信息的形式：

邮件 信函 短信 其他_____

请寄：北京市西城区百万庄南街1号 机械工业出版社 华章公司 计算机图书策划部收
邮编：100037 电话：(010) 88379512 传真：(010) 68311602 E-mail: hzsj@hzbook.com

[General Information]

淘宝助手、yaowohaoabaobei 荣誉制作

QQ: 1583929799

旺旺: yaowohaoabaobei

淘宝书不缺网: <http://shubuque.taobao.com>

您身边的电子书专家

简介:

千万本电子书任你选, 您只需提供书名、作者、出版日期 即可:

电子书选购网站 (校内网用户):

超星: <http://www.ssreader.com/>

中美百万: <http://www.cadal.net/>

国家图书馆: <http://res4.nlc.gov.cn/home/index.trsf?channelid=19>

以上资源库, 只要能搜到的书, 均可PDF 格式提供

指针: <http://www.zhizhen.com/>

读秀: <http://www.duxiu.com/>

书生: <http://www.souba.org/sou/>

圣才: <http://library.100xuexi.com/video/>

以上网站中所有有试读页的图书均可PDF 格式提供!

如果您在校外网, 以上部分资源库登陆不了, 请登陆:

指针: <http://www.zhizhen.com/>

读秀: <http://www.szdnnet.org.cn/>

这两个网站是面向全部网络用户的开放的

欢迎广大书友对本人制作书籍提供宝贵意见

联系QQ: 1583929799; 旺旺: yaowohaoabaobei;

网址: <http://shubuque.taobao.com>;

博客: <http://blog.163.com/hhxytyxyt@126/blog>