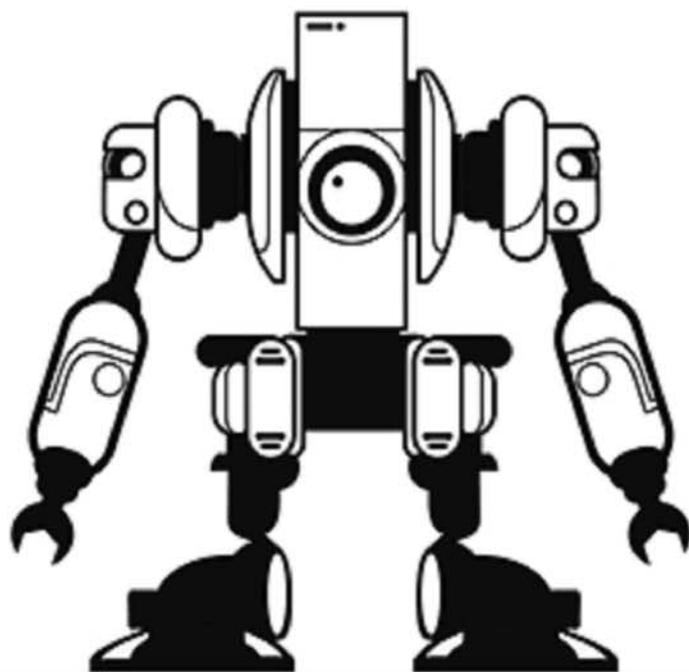


知名技术专家 蔡学镛 力荐

本书以精练的语句结合源码剖析的方式诠释了JVM的许多关键原理
阅读本书，你将有知其然并知其所以然的淋漓畅快感

Broadview[®]
www.broadview.com.cn



Java虚拟机精讲

如果你对JVM感兴趣，并且从未接触过JVM，
那么本书将会是你探索JVM世界的入门必备工具

高翔龙 编著

 中国工信出版集团

 电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

内 容 简 介

HotSpot VM 是目前市面上高性能 JVM 的代表作之一，它采用解释器+JIT 编译器的混合执行引擎，使得 Java 程序的执行性能从此有了质的飞跃。本书以极其精练的语句诠释了 HotSpot VM 的方方面面，比如：字节码的编译原理、字节码的内部组成结构、通过源码的方式剖析 HotSpot VM 的启动过程和初始化过程、Java 虚拟机的运行时内存、垃圾收集算法、垃圾收集器（重点讲解了 Serial 收集器、ParNew 收集器、Parallel 收集器、CMS（Concurrent-Mark-Sweep）收集器和 G1（Garbage-First）收集器）、类加载机制，以及 HotSpot VM 基于栈的架构模型和执行引擎（解释器的工作流程、JIT 编译器的工作流程、分层编译策略、热点探测功能）等技术。

如果你对 JVM 感兴趣，并且从未接触过 JVM，那么本书将会是你探索 JVM 世界的必备入门工具。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

Java 虚拟机精讲 / 高翔龙编著. —北京：电子工业出版社，2015.5
ISBN 978-7-121-25705-6

I. ①J… II. ①高… III. ①JAVA 语言—程序设计 IV. ①TP312

中国版本图书馆 CIP 数据核字（2015）第 050785 号

责任编辑：孙学瑛

印 刷：北京中新伟业印刷有限公司

装 订：北京中新伟业印刷有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本：787×980 1/16 印张：17.5 字数：448 千字

版 次：2015 年 5 月第 1 版

印 次：2015 年 5 月第 1 次印刷

定 价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zltz@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

第 1 章

Java 体系结构

1.1 认识 Java

经历了多年的发展，Java 早已由一门单纯的计算机编程语言，演变为一套强大的技术体系平台。根据不同的技术规范，Java 设计者们将 Java 划分为 3 种结构独立但却又彼此依赖的技术体系分支，分别是 Java SE（标准版）、Java EE（企业版）和 Java ME（精简版）。在此大家需要注意，本书所提及的这 3 种技术体系分支，分别对应着不同的规范集合和组件。Java SE 活跃在桌面领域，主要包含了 Java API 组件。而 Java EE 则活跃在企业级领域，除了包含 Java API 组件外，还扩充有 Web 组件、事务组件、分布式组件、EJB 组件、消息组件等；综合这些技术，开发人员完全可以构建出一个具备高性能、结构严谨的企业级应用，并且 Java EE 也是用于构建 SOA^①架构的首选平台。至于 Java ME 则活跃在嵌入式领域，之所以将其称之为精简版，那是因为该平台仅保留了 Java API 中的部分组件，以及适应设备的一些特有组件。

Java 在奠定了企业级领域的霸主地位后，目前正一步步朝着移动领域的方向大展拳脚，这不仅要感谢移动互联网的迅速崛起，还得多亏 Google 选择 Java 作为 Android 操作系统的应用层编程语言。就目前而言，Java 已经成为了全球开发人员使用最为广泛的一种编程语言。从随处可见的手持移动设备、嵌入式设备、个人电脑、高性能的集群服务器或大型机中，我们几乎随处都可以看见 Java 程序的身影。或许当你还在犹豫和怀疑 Java 能做什么的时候，Java 早已在企业级领域、互联网领域、移动领域、中间件领域，甚至是游戏领域都发展得

① SOA（Service-Oriented-Architecture，面向服务架构）作用于分布式的系统集成环境中，它将程序的内部功能通过定义良好的契约对外发布成体系结构中立的接口，以此满足不同系统之间的交互操作。

如火如荼。比如著名的开源 3D 游戏引擎 jME (j-Monkey-Engine)^②就是完全采用 Java 语言编写的,该引擎可以算是目前 Java 平台上最流行,同样也是应用最广泛的 3D 游戏引擎。当然这所有的一切都离不开 Java 的运行支撑系统,那就是 Java 虚拟机,Java 与生俱来的通用性、安全性和高效性都建立在 Java 虚拟机之上。

从早期版本到每一个新版本的迭代,Java 都会不断完善自身缺陷,并进行语法增强,这无疑是带给开发人员最好的礼物。本书不仅会重点讲解与 Java 虚拟机相关的一些知识点,在本书的附录中笔者还为大家讲解了有关 Java7 在语法层面上的一些改变和扩充,让大家更全面地了解和掌握 Java 技术。

1.1.1 与生俱来的优点

面向对象的思想如今已经渗透到软件开发的各个领域,例如 OOA (Object Oriented Analysis, 面向对象的分析)、OOD (Object Oriented Design, 面向对象的设计),以及开发人员时常挂在嘴边的 OOP (Object Oriented Programming, 面向对象的编程)。除了在急需注重性能与效率的应用场景下,开发人员大多数时候都是在使用面向对象等高级语言,比如 C#、C++、Ruby、PHP 等。这些高级语言无论是从设计原理或者是从实现细节上来看都是非常精妙的,那么与这些同样优秀的语言相比,Java 的优势主要体现在哪里呢?本书归纳了 Java 的 5 项重要优势:

- 体系结构中立;
- 安全性优越;
- 多线程;
- 分布式;
- 丰富的第三方开源组件。

Java 之所以能够实现“一次编译,处处运行”(Write Once, Run Anywhere),功不可没的首先当属字节码。和 C/C++等传统的编译性语言不同,Java 源代码的默认编译结果并非是可执行代码(本地机器指令),而是具有平台通用性的字节码。尽管不同平台 Java 虚拟机的内部实现机制不尽相同,但是它们共同解释出的字节码却是一样的,所以说字节码才是 Java 实现跨平台的关键要素,如图 1-1 所示。体系结构中立不仅使得 Java 天生具备跨平台的优势,同时还延伸了程序的安全性,因为 Java 程序始终只能够运行在 Java 虚拟机中,这与实际的物理宿主环境之间是相互“隔离”的,换句话说 Java 的安全模型可以禁止很多不安全的因素,有助于防止错误的发生,增强程序的可靠性。当然 Java 的部分语法限制也在某种意义上保障了程序的安全,比如废弃指针操作、自动内存管理、数组边界检查、类型

^② jME (j-Monkey-Engine) 官方地址: <http://www.jmonkeyengine.org/>。

转换检查、线程安全机制和物理环境访问限制等。

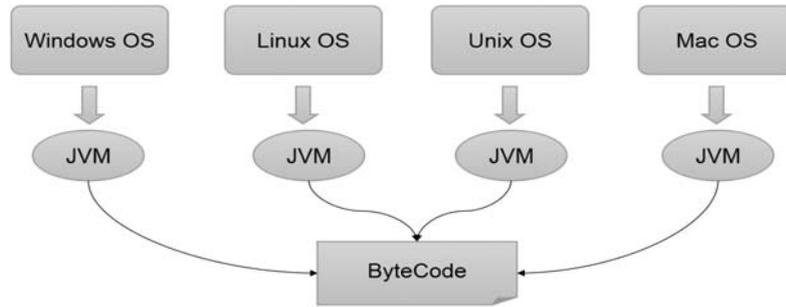


图 1-1 Java 跨平台特性

开发人员往往希望由自己编写的程序能够在性能上满足高效执行的要求，那么这就需要正确运用多线程技术去有效地并发执行操作任务。单线程是按照顺序结构进行串行执行的，我们暂且先不讨论程序的执行效率，一旦程序因执行某一个任务而发生异常，往往有可能导致程序终止而无法继续往下执行。当遇到这种情况的时候，就需要一种机制能够将这种任务从主线程中剥离出来，哪怕是发生异常也不会影响系统的整体运行。从另一个角度来说，我们可以使用多线程的并发机制将任务进行分散，而不是全部集中在主线程内，采用异步的方式去并发执行多项任务，这样的程序设计架构必然会有极高的执行效率。

由于单台服务器的处理能力很有限，甚至在某些情况下单台服务器的处理能力还存在性能瓶颈，因此在生产环境中架构师往往会考虑采用分布式架构的方式来部署应用。只有充分利用分布式环境中的每一个节点去协同处理任务，才能够换来较高的执行效率，并且还能有效降低单机负载以及提升稳定性和可用性。Java 与生俱来对分布式技术的支持就比较完善，比如 Java EE 规范中的 RMI (Java Remote Method Invocation, Java 远程方法调用)、JMS (Java Message Service, Java 消息服务) 等技术。

框架这个名词相信对于绝大多数开发人员而言并不会感觉到陌生，它是对编码规范的一种抽象。在企业内部开发人员往往会为了提高生产效率和易于扩展从而选择使用框架，因为遵守框架定义的内部契约，不仅可以更高效地解决技术难题，同时还能够缩短项目的开发周期，总之如果能够在项目中正确选择和使用框架，那么所带来的好处将会是不言而喻的。

衍生在 Java 平台上的各类第三方开源框架几乎随时都保持着更新，可以毫不客气地说，在如今的整个领域模型中，随处可见成熟的第三方开源框架已经开始“泛滥”，当然本书仅以贯穿整个领域模型的 Spring 框架为例进行介绍。开发人员不仅可以选择使用 Spring 提供的全部功能，甚至还可以根据业务需求选择使用 Spring 的部分功能子集。如果想要使用

MVC^③，Spring 提供支持 RESTful 风格的 Spring MVC，甚至允许与其他第三方 MVC 框架进行无缝集成；如果想要使用 Spring 的核心功能，那么 Spring 最新发布的 4.x 版本将会有全新体验；如果想对权限进行控制管理，Spring 提供 Spring Security；如果不想手动管理持久层事务，Spring 提供丰富的 ORM 集成策略，或者直接选用 Spring JDBC（Spring JDBC 仅仅只是针对传统的 JDBC 做了轻量级封装，熟悉 JDBC 的开发人员都能够迅速掌握并使用）作为持久层框架；如果想降低使用 Hadoop 进行海量数据处理所带来的复杂度和学习成本，Spring 提供 Spring Hadoop；甚至如果你是 Android 的开发人员，Spring 照样提供基于 Mobile 领域的各类规范和实现。

当然笔者并不是在为 Spring 打广告和做宣传，只是想告诉大家，Java 真正强大的地方是因为拥有全世界最多的技术拥护者和开源社区支持，他们无时无刻都保持着最充沛的体力与思维，一步一步地驱动着 Java 技术的走向，其实这才是 Java 最大的优势和财富。

1.1.2 语法结构和对象模型

Java 继承了 C 语言的语法结构，并改编了 C++ 语言的对象模型，所以注定了 Java 天生就适合书写优美的代码。我们都知道，类是最基本的封装单元，所有的操作都将发生在类中，那么定义一个类，其内部的组成结构又是怎样的呢？简单来说，属性和方法构成了一个简单的类，属性用于定义对象的各种“器官”，而方法则用于定义对象的一系列“行为”。虽然这样的描述很直观，却显得比较粗糙，如果将其细化分类后，大家或许会发现，Java 语法结构的设计是非常精妙的，同样又不失灵活性和简洁性。在类内部，开发人员可以定义许多元素特征，这些元素都统称为类成员，本书归纳了 Java 的一些基本类元素信息：

- 关键字；
- 标示符；
- 操作符（空白分隔符、普通分隔符）；
- 注解（@Annotation 类型、描述类型）；
- 数据类型（原始数据类型、引用类型）；
- 属性（常量、变量）；
- 运算符和表达式；
- 控制语句（流程控制语句、循环控制语句）；
- 异常处理；

③ MVC（Model-View-Controller，模型层-视图层-控制层）是一种编程模型，它将一个系统拆分为 3 大类，视图层仅只负责页面显示和数据显示工作，控制层则负责客户端的请求/响应和业务调用工作，而模型层则负责实际的业务操作。

Java 虚拟机精讲

□ 方法体。

对象模型与面向对象（Object Oriented）的特性之间保持着一种密不可分的关系。面向对象之所以目前大行其道，其中最关键的因素在于在系统构建复杂化的当下，允许开发人员以面向对象式思维设计出更具复用性、维护性、扩展性和伸缩性的应用程序。然而在语法层面上，开发人员可以在程序中直接使用 new 关键字创建一个对象，并返回当前对象的一个引用（reference）。在此大家需要注意，Java 中的引用操作绝不等价于 C++ 中的指针，因为引用类型的变量持有的仅仅只是一个引用而已而非实际值，也就是说开发人员并不能在程序中直接与对象实例打交道，而必须通过引用进行“牵引”，如图 1-2 所示。在程序中即便不存在对象实例，引用也允许独立存在，也就是说可以声明一个引用，不一定非要有一个对象实例与之关联，但务必确保在真正使用一个对象时，它已经完成了初始化操作，也就是执行了 <init>() 方法。



图 1-2 引用关系

在谈及如何在语法层面上创建一个对象之后，笔者不得不提及的还有构造方法，构造方法出现的目的是为了初始化对象以及成员变量。在 Java 中对象的初始化和创建其实是同一个操作，虽然从字义上来理解，初始化和创建并不是同一个概念，但它们却恰巧被 Java 设计者们绑定在一起，谁也离不开谁。并且在继承环境下，派生类与超类之间构造方法的加载顺序同样也是按照派生顺序进行加载的，这样做的就是为了确保每一个对象在使用前都已经被成功初始化过。

1.1.3 历史版本追溯

Java 发展至今差不多已经有二十多个年头了，从诞生到如今的茁壮，Java 可谓是具备了天时地利人和。1991 年 Sun 公司的 James Gosling（Java 语言的主要创始人）等人为嵌入式设备开发了一种叫作 Oak（一种橡树的名称）的编程语言，其实 Oak 就是如今 Java 语言的前身，只是在当时的特定环境下，Oak 语言并没有引起大多数人的注意。直到 1994 年，随着互联网和 3W（World Wide Web）的迅猛发展，他们使用 Oak 编写了一个叫作 HotJava 的浏览器，这才得到了 Sun 公司的首席执行官 Scott McNealy 的支持，最终 Java 才得以继续研发和发展。

为了促销和法律等原因，1995 年 Oak 语言正式更名为 Java，同年正式在 Sun World 大会上发布了 Java 1.0 版本，并且首次提出了“Write Once, Run Anywhere”的口号。其实 Java 的得名还有段小插曲，有一天 Oak 小组成员正在喝咖啡时，议论给新语言起个什么名字比

较好，于是有人提议用 Java（Java 是印度尼西亚盛产咖啡的一个岛屿）作为新语言的名字，这个提议得到了其他小组成员的赞同，就采用 Java 来命名此新语言。很快 Java 就被工业界所认可，许多大公司，如 IBM Microsoft.DEC 等购买了 Java 的使用权，并被美国杂志 PC Magazine 评为 1995 年十大优秀科技产品，从此开启了 Java 的新篇章。直至 2010 年 IT 巨头 Oracle 出面收购 Sun 公司，Java 则更是达到了巅峰。或许有部分开发人员经常在论坛“忧愁”Java 的未来，但相信大多数开发人员和笔者一样，都认为 Oracle 收购 Sun 是一件好事，毕竟 Java 的未来还有很长的路要走，Oracle 这样的企业必然具备优秀的技术实力可以为 Java 注入更多、更丰富的特性和变化。

Java1.0 正式版本所包含的功能并不算多，除了在 JDK 中配套一个纯解释器实现的 Java 虚拟机外（Sun Classic VM），仅支持 Applet、AWT 等技术。直到 1997 年 Java1.1 在 1.0 的基础之上添加了 JDBC、JAR 格式支持、JavaBeans、RMI 等技术，并在语法层面上开始支持反射和内部类等操作后，Java 的功能才开始逐渐变得丰富。

1998 年 Sun 公司发布了 Java1.2 版本，并在此版本中将 Java 划分为 3 种结构独立却彼此依赖的技术体系分支，分别是 J2SE（标准版）、J2EE（企业版）和 J2ME（精简版）。在这个版本中也添加了 EJB、Java Plug-in、Java IDL、Swing 等技术，并且 Sun 公司首次在 JDK 中内置了 JIT 编译器。

2000 年 Sun 公司发布了 Java1.3 版本，但这个版本并没有引进太多的技术和改变，只是在基础类库上做了一些改进。直到 2002 年 Java1.4 版本的发布才是重头戏，这意味着 Java 已经开始逐渐走向成熟。Java1.4 发布了相当多的特性，如正则表达式、异常链、NIO、日志类、XML 解析器和 XSLT 转换器等。时至今日国内某些大型企业仍然还在沿用 Java1.4 的版本，并且很多优秀的第三方开源产品同样也针对 Java1.4 版本做了向下兼容。

2004 年 Sun 公司发布了 Java1.5 版本，同时也是在 1.5 版本发布后，后续的 Java 版本都改为以 Java5、Java6、Java7 等规则进行命名。在早期版本中，Sun 公司对 Java 各个版本的语法层面的改变并不大，但 1.5 版本针对语法层面的改进却相当多，几乎导致整个 API 都发生了变化。比如自动装箱/拆箱、泛型、枚举、@Annotation、可变长参数、foreach、粗粒度的并行模型等。

2006 年 Sun 公司发布了被 Oracle 收购之前的最后一个版本 Java6。在该版本中，Sun 公司改变了从 Java1.2 开始的惯用的 J2SE（标准版）、J2EE（企业版）和 J2ME（精简版）命名方式，更名为 Java SE（标准版）、Java EE（企业版）和 Java ME（精简版）。Java6 的改变更多是体现在虚拟机内部，主要以同步对象锁、垃圾回收、类型装载等方面的算法更新为主。同年 Sun 公司正式宣布 Java 以 GPL（General Public License）v2 的开源协议进行源代码公开，并建立了 OpenJDK 对 Java 的源代码库进行独立管理。

Java 虚拟机精讲

由于 Sun 公司无力推动 Java7 的研发工作，2010 年正式被 Oracle 公司收购，并由 Oracle 正式接替 Java7 的后续研发工作。由于 Java7 预期的功能非常多，这不得不导致 Oracle 将部分功能进行裁剪，延迟到 Java8 的版本中再进行发布。Java7 的正式版本不仅在语法层面上做了较大改变，还引入了许多新的技术，比如更新了 Java 的文件系统、细粒度的 Fork/Join 并行编程、混合语言等。同时 Java 虚拟机内部也做了许多改进和调整，比如 Java7 提供了 G1 垃圾收集器、类装载器的并行装载增强实现等。

1.2 Java 重要概念

Java 的体系结构主要由 Java 编程语言、字节码、Java API 和 Java 虚拟机等 4 部分独立却相关的技术组成。或许很多时候我们并没有刻意去关注它们，但确实确实当我们在编写 Java 程序的时候，就同时用到了这 4 种技术。首先我们使用 Java 编程语言编写好 Java 程序的源代码，然后 Java 前端编译器负责将 Java 源代码编译为字节码，接着 Java 虚拟机负责将这些编译好的字节码装载进内部，最后解释/编译为对应平台上的机器指令运行。这就是一个完整的 Java 程序从编写到最终执行的结构链路。

1.2.1 Java 编程语言

Java 编程语言是一种语法结构严谨、体系结构中立、面向对象、支持多核并行的程序设计语言，它继承了 C 语言的语法结构，并改编了 C++ 的对象模型，所以 Java 天生就适合书写优美的代码。并且 Java 舍弃了 C 和 C++ 中许多不安全的语法特性，比如：废弃指针操作、自动内存管理、数组边界检查、类型转换检查、线程安全机制和物理环境访问限制等。

1.2.2 字节码

Java 最初诞生的目的就是为了在不依赖于特定的物理硬件和操作系统环境下运行，那么也就是说 Java 程序实现跨平台特性的基石其实就是字节码。Java 之所以能够解决程序的安全性问题、跨平台移植性等问题，最主要的原因就是 Java 源代码的编译结果并非是本地机器指令^④，而是字节码。当 Java 源代码成功编译成字节码后，如果想在不同的平台上面运行，则无须再次编译，也就是说 Java 源码只需一次编译就可处处运行，这就是“Write Once, Run Anywhere”的思想。所以注定了 Java 程序在任何物理硬件和操作系统环境下

④ 本地机器指令：可以被计算机 CPU 直接进行识别并执行，其表现形式为二进制。机器指令通常由操作码和操作数两部分组成，操作码负责指令需要完成的操作，即指令的功能。而操作数负责数据运算，以及运算结果所存放的位置等。

都能够顺利运行，只要对应的平台装有特定的 Java 运行环境，Java 程序都可以运行，虽然各个平台的 Java 虚拟机内部实现细节不尽相同，但是它们共同执行的字节码内容却是一样的。

那么什么是字节码（ByteCode）呢？参考《Java 虚拟机规范 Java SE7 版》的描述来看，任何编程语言的编译结果满足并包含 Java 虚拟机的内部指令集、符号表以及一些其他辅助信息，它就是一个有效的字节码文件，就能够被虚拟机所识别并装载运行。在大部分情况下，字节码更多是存储在本地磁盘文件中，比如后缀名为“.class”的文件。每一个字节码文件都对应着全局唯一的一个类或者接口的定义信息，但这也并非绝对，类和接口并不一定都只能存储在文件里，它还可以通过类装载机直接在运行时生成。

字节码结构组成比较特殊，其内部并不包含任何的分隔符区分段落，所以无论是字节顺序、数量都是有严格规定的，所有 16 位、32 位、64 位长度的数据都将构造成 2 个、4 个和 8 个 8 位字节单位来表示，多字节数据项总是按照 big-endian 顺序（高位字节在地址最低位，低位字节在地址最高位）来进行存储。也就是说，一组 8 位字节单位的字节流组成了一个完整的字节码文件。

1.2.3 Java API

API（Application Programming Interface，应用程序编程接口）是一些预先定义的接口，目的是提供应用程序与开发人员基于某软件或硬件的以访问一组例程的能力，而又无需访问源码或理解内部工作机制的细节。Java API 通过支持与平台无关性和安全性，使得 Java 程序适应任何应用场景。那么 Java API 中其实包含的就是 Java 的基础类库集合，它提供一套访问主机系统资源的标准方法。

1.2.4 Java 虚拟机

Java 技术的核心就是 Java 虚拟机（JVM，Java Virtual Machine），因为所有的 Java 程序都运行在 Java 虚拟机内部。JVM 之所以被称之为 VM，是因为它是由一组规范所定义出的抽象计算机。JVM 的主要任务就是负责将字节码装载到其内部，解释/编译为对应平台上的机器指令执行，如图 1-3 所示。

Sun 公司的 HotSpot VM 应该是大多数开发人员最熟悉的一款高性能 Java 虚拟机，它是 JDK 和 OpenJDK 中缺省自带的一款虚拟机，同样也是目前市面上应用最广的一款 Java 虚拟机。但这款虚拟机最早是由一家名不见经传的小公司“Longview Technologies”研发设计出来的，后来这家公司被 Sun 公司收购后，HotSpot 虚拟机也同样被纳入麾下。

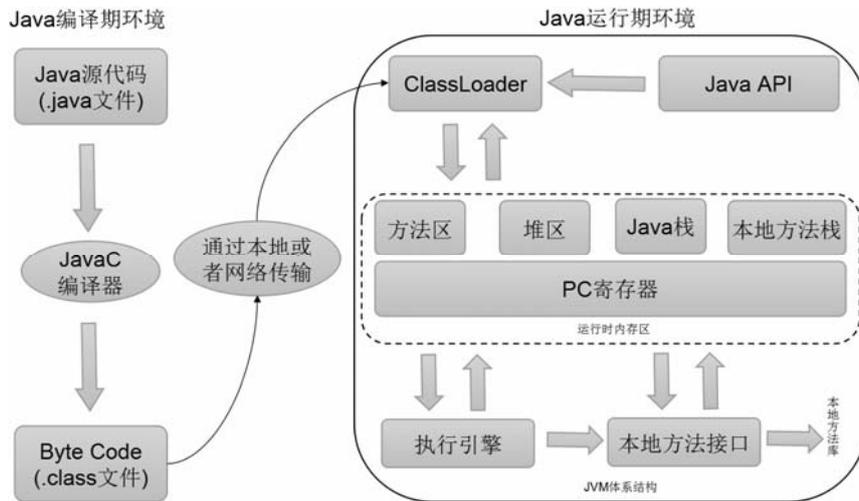


图 1-3 字节码编译期与运行期

HotSpot VM 是目前市面上高性能虚拟机的代表作之一。它具备热点探测功能，可以通过此功能将一个被频繁调用的方法或者方法体中有效循环次数较多的代码块标记为“热点代码”，然后通过内嵌的双重 JIT（Just In Time Compiler）编译器将字节码直接编译为本地机器指令。在 HotSpot VM 内部，即时编译器与解释器是并存的，通过编译器与解释器的协同工作，既可以保证程序的响应时间，同时还能够提高程序的执行性能，并且对编译器的工作压力也降低了一定程度的负载。换句话说 HotSpot 是一款解释器与编译器并存的虚拟机，缺省情况下一个程序中到底有多少字节码指令是通过解释运行的，还是通过编译运行的，这就需要依赖热点探测功能。

虽然 Java 虚拟机规范并没有强制要求虚拟机内部实现一定要采用解释器和编译器并存的架构方案，但目前市面上大多数主流虚拟机都采用此架构。这是因为当虚拟机启动的时候，解释器可以首先发挥作用，而不必等待编译器全部编译完成再执行，这样可以省去许多不必要的编译时间。并且随着程序运行时间的推移，编译器逐渐发挥作用，根据热点探测功能，将有价值的字节码编译为本地机器指令，以换取更高的程序执行效率。HotSpot VM 中内嵌有两个 JIT 编译器，分别为 Client Compiler 和 Server Compiler，但大多数情况下我们简称为 C1 编译器和 C2 编译器。开发人员可以通过命令显式指定到底使用哪一种编译器策略，缺省情况下 HotSpot 会根据操作系统版本与物理机器的硬件性能进行自动选择。当然开发人员还可以通过命令显式指定 HotSpot VM 到底是使用完全编译策略，还是完全解释策略，如果我们将虚拟机选定为完全解释策略，那么编译器将停止所有的工作，字节码将完全依靠解释器逐行解释执行。反之也可以选用完全编译策略，但解释器仍然会在编译器无法进行的特殊情况下介入执行，这主要是确保程序能够最终顺利执行，如图 1-4 所示。

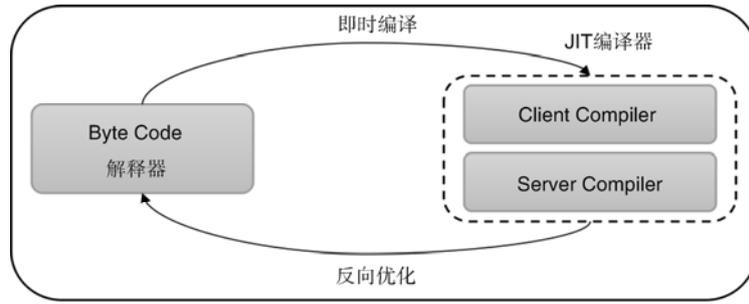


图 1-4 解释器与 JIT 编译器协作

Java7 缺省开启分层编译（Tiered Compilation）策略，由 C1 编译器和 C2 编译器相互协作共同来执行编译任务。C1 编译器会对字节码进行简单和可靠的优化，以达到更快的编译速度；而 C2 编译器会启动一些编译耗时更长的优化，以获取更好的编译质量。

其实除了 HotSpot 之外，目前市面上也不乏一些同样优秀的 Java 虚拟机产品，比如 IBM 公司研发的 IBM J9 VM，以及 Oracle 收购的 BEA JRockit VM 等。Oracle 公司将会在后续版本中，将 HotSpot VM 和 JRockit VM 合二为一，在 HotSpot VM 的基础之上整合 JRockit VM 的诸多优点。

1.3 安装与配置 Java 运行环境

所谓工欲善其事，必先利其器。在开始正式讲解后续章节之前，我们首先需要搭建好 Java 平台所需的一系列开发环境和运行环境。Java 的运行环境工具我们称之为 JDK（Java Development Kit，Java 软件开发工具包）。JDK 是 Oracle 公司提供的用于支持 Java 程序运行的开发工具包。该包中包含 Java 运行环境、Java 工具和 Java API。大家可以从 Oracle 的官方网站下载 JDK 工具包，本书所有程序示例均使用 jdk7-u15-x86 版本，建议大家还是下载和本书保持一致的版本，尽可能避免因版本问题而导致出现一些不一致的情况。

1.3.1 Windows 环境下的安装与配置

当成功下载 JDK 后，大家可以选择默认与自定义风格两种模式来进行 JDK 的安装。当成功安装好后，我们可以发现在安装目录下会出现一个“java”文件夹。该文件夹内部还包含一个“jdk”文件夹与“jre”文件夹。“jre”文件夹中仅仅只是包含 Java API 组件与 Java 运行环境，而“jdk”文件夹中除了包含除 Java API 组件与运行环境外，还一些 Java 工具及 Java API 源码。

在 jdk 或 jre 文件夹的 bin 目录下，包含 JDK 最主要的两个工具，分别是 Java 虚拟机和

Java 前端编译器。Java 虚拟机叫做“Java.exe”，而 Java 前端编译器则叫做“Javac.exe”。当然还包含一些其他工具，本书会在后续章节中陆续为大家进行讲解其使用过程。

当成功安装好 JDK 后，我们还需要为其配置环境变量。配置环境变量的目的其实是在于引导运行环境正确执行 Java 应用程序。当然如果你电脑里安装有 IDE（集成开发环境）工具，理论上不需要配置 JDK，但如果需要将 Java 程序打包在本地运行时，Java 运行环境就显得至关重要了。并且对于 Eclipse 等开源的 IDE 工具而言，本身并无内置 JDK，所以笔者还是建议大家下载好 JDK 后，按照本书的步骤配置好 Java 环境变量，这样便能一劳永逸。

在 Windows 环境下配置 Java 环境变量比较简单，选择“系统属性->环境变量”，新增如下 3 个环境变量参数即可：

- JAVA_HOME=JDK 安装路径
- PATH=JDK 安装路径\bin 路径
- CLASS_PATH=.

当配置好环境变量后，我们便可以通过打开 Windows 控制台输入命令“javac”来查看 Java 环境变量是否成功配置，如果输入该命令后返回的是用法信息，则意味着 Java 的环境变量已经配置成功。大家还可以通过命令“java -version”查看当前安装的 JDK 版本。

1.3.2 Linux 环境下的安装与配置

1.3.1 节讲解了关于如何在 Windows 环境下安装与配置 Java 运行环境。但在实际的程序开发过程中，我们往往需要将编写好的 Java 程序部署在 Linux 平台上运行。那么这个时候必然需要在 Linux 操作系统上安装与配置 Java 的一系列运行环境。其实和 Windows 平台相比起来，Linux 环境下的 Java 安装与配置则显得稍微繁琐一些。

在开始往 Linux 环境下安装与配置 Java 运行环境之前，大家首先需要确保自己的电脑里已经成功安装 Linux 操作系统。当然如果你没有多余的电脑进行 Linux 环境的搭建，那么可以在本机安装一个虚拟机工具来搭建 Linux 环境。由于目前市面上拥有较多开源的 Linux 操作系统版本，可能会导致某些刚接触 Linux 的开发人员无从选择，所以在此笔者推荐其操作系统版本为 ubuntu-12.04.1-x86/x64，因为该版本界面较为友好和人性化，适合刚接触 Linux 操作系统的开发人员。

当一切的基础环境都准备好以后，你可以登录 Oracle 的官方站点下载免安装版本的 JDK 工具包，或者通过打开 Linux 平台下的 shell 命名解析器输入命令“sudo apt-get install 名称”进行 JDK 的下载。建议选择的 Linux 免安装版本为 jdk7-u15-linux-x86。当成功下载好 JDK 后，我们需要将其解压，并存放于所指定的目录即可完成在 Linux 操作系统下的 Java 运行

环境安装。

当成功安装 Java 运行环境后，我们仍然需要配置 Java 的环境变量。打开 Linux 平台下的 shell 命名解析器输入命令“vi /etc/profile”对 profile 文件进行编辑，在最后一行新增如下 3 个环境变量参数：

- JAVA_HOME=JDK 安装路径
- PATH=JDK 安装路径\bin 路径
- CLASS_PATH=.

当成功配置好环境变量参数并保存退出后，大家还需要执行命令“source /etc/profile”使配置生效。这时我们便可以通过打开 Linux 平台下的 shell 命名解析器输入命令“javac”查看 Java 环境变量是否成功配置，如果输入该命令后返回的是用法信息，则意味着 Java 的环境变量已经配置成功。大家还可以通过命令“java -version”查看当前安装的 JDK 版本。

几乎所有的 Linux 操作系统预先就安装 Java 运行环境（OpenJDK），如果程序并没有特定依赖指定的 JDK 版本，还是建议大家不必重复安装。当然如果你仅仅是感兴趣而想尝试着在 Linux 下安装 Java 环境变量倒也无妨。

1.3.3 编写 Java 程序

当一切环境与准备工作都做好后，接下来本书将带领大家编写第一个 Java 应用程序。这个程序非常简单，只需要在控制台输出一段字符即可，并不包含过多的 Java 语言语法特性。

打开你的记事本或者 Editplus 工具，把下述代码拷贝后粘贴进去，保存后更改文件的后缀名为“.java”。在此需要提醒大家，文件名称务必保持和 Java 类型名称一致，否则 Javac 编译器无法将你所指定的 Java 源代码成功编译为有效的字节码文件。如下所示：

代码 1-1 Java 程序示例

```

/**
 * 一个简单的 Java 程序
 *
 * @author JohnGao
 */
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}

```

上述程序示例中，`main` 函数作为 Java 程序的入口点。不管其处于代码的任意位置，Java 程序都会以 `main` 函数作为调用函数。函数的意思为某一特定块代码块所要执行的具体任务，只不过在 Java 语言中其实应该准确的称呼为“方法”。

`System` 是 Java API 的预定义类，主要是给开发人员提供对底层操作系统 API 的只读访问操作，比如开发人员可以使用 `System` 类型的 `getProperty("os.name")` 函数获取当前操作系统的系统名称。当然上述程序示例中字符串的输出其实是由 `PrintStream` 类型的 `println` 函数完成的，而 `System.out` 实际上就是调用的系统内置输出流 `PrintStream` 类型。

1.3.4 编译与运行

当大家成功将 1.3.3 节的程序示例拷贝粘贴至记事本，并成功保存更改后缀名为“`.java`”后，则可以使用 `JavaC` 编译器对其进行字节码编译。当然在编译前务必要确定本地环境变量是否成功配置，如果没有配置或者配置错误 `Java` 环境变量后则无法正确编译字节码。大家打开 `Windows` 的控制台，切换到 `Java` 源代码所在的路径后，输入如下编译语法即可成功将 `Java` 源代码编译成字节码文件：

```
“路径：\>javac 包名\类名.java”
```

当成功将 `Java` 源代码编译成字节码后，`JavaC` 编译器会在与 `Java` 源代码的同级目录生成一个与源代码名称一致但后缀为“`.class`”的字节码文件。大家打开 `Windows` 控制台，切换到 `Java` 字节码所在的路径后，输入如下语法即可成功执行 `Java` 程序：

```
“路径：\>javac 包名\类名.java”
```

当成功编译并运行上述 `Java` 程序后，控制台将会输出如下字符信息：

```
Hello World
```

1.3.5 关键字与标示符

`Java` 目前大约定义了 51 个关键字。其实所谓关键字指的就是 `Java API` 内部预定义的一些字符集合（如 `public`、`private`、`class` 等），这些关键字与 `Java` 语法结构息息相关，同样开发人员也可以称其为保留字。至于标示符其实就是用于给类、方法、变量命名，只要在程序中可以自己命名的地方，统统都可以称之为标示符。在此需要提醒大家，`Java` 语法结构规定开发人员不允许在程序中使用关键字作为标示符命名，这一点请大家一定要牢记，否则将无法通过编译。

其实不仅仅是 `Java`，任何程序设计语言对于标示符的命名都遵循了业界统一的命名规范。虽然命名规范并不强制要求开发人员一定要遵循，但这对于程序开发风格的统一将会有

很大好处。如果你希望你编写的代码拥有更好的可读性和维护性，那么就应该遵循这些习惯。

对于类型标示符的命名，开发人员应该按照首字母大写的方式进行标示，比如“Xxxx”。而变量、方法则应该按照“驼形命名规则”的方式进行标示，也就是首字母需要小写，比如“xXxxX”。而对于包的命名，开发人员则需要将所有字母小写，比如“com.xxx.xxx”。常量的命名则最为特殊，所有的字母均全部按照大写的方式进行标示，比如“XXXX”。当然如果常量由多个单词组成，这时为了确保可读性，开发人员可以使用“_”符号进行分割。

其实掌握命名规范并不困难，因为这仅仅只是一种习惯，但还是需要提醒刚接触编程的开发人员，请务必遵循如下守则：

- 不要采用中文编码的方式对标示符进行命名，务必只使用英文字符；
- 不要使用特殊符号的方式对标示符进行命名，比如“@#¥%.....&*（）-+=”，但允许使用符号“_”，但建议尽量不要标示在首字母第一位；
- 如果需要以数字的方式对标示符进行命名，不能标注在首字母第一位；
- 标示符不能以关键字进行命名。

一般来说程序开发所选用的字符语言都是英文字母，这是从编程语言诞生至今就养成的一种根深蒂固的习惯。但不是说不能使用其他字符进行程序开发，前提是你需要确认程序的开发工作是一个团队还仅有你一个人，可读性才是需要考虑的首要问题。

目前国内的确已经出现一种采用中文字符进行编程的程序设计语言（API 与关键字都设定为中文字符）。该设计者的设计初衷确实能够有效解决某一部分对英文“过敏”的开发人员的读写问题，但是对于大部分本土开发人员而言，它始终受众面过于狭隘，导致该语言的推广力度较为艰难。

1.4 Java 技术的新特性

2005 年 6 月 28 日，由全球一万多名开发人员出席的 Java One 开发者大会上，Sun 公司老板 Scott McNealy 先生向 Java 之父 James Gosling 博士颁发了“终身成就奖”，这不仅是 Sun 公司对 James Gosling 博士十多年来创新性工作的肯定和感谢，同时也意味着 Java 已经成为世界上应用最为广泛的技术之一。作为 Java One 开发者大会上的压轴戏，James Gosling 博士做了一场命题为“Java 技术下一个十年”的演讲。

1.4.1 Java 模块化与 OSGi 技术

近几年来，Java 模块化一直是一个比较活跃的话题。那么究竟什么是模块化呢？其实

所谓模块化指的就是开发人员在构建大型系统时,能够将系统中的每一个功能模块进行独立的开发和物理部署,这样做的优点不仅能够有效降低各个业务模块之间的耦合,同时还能够保证当单一模块发生故障的时候不会影响系统整体的运行。当然模块化本身只是一种概念,其目的就是为了将系统中原本耦合的逻辑进行分解,以此满足各个模块之间的独立,并定义一种标准化的接口契约来进行相互之间的通信。

尽管 Java 目前并没有在 JDK 中内置模块化编程技术,但这似乎并不能阻挡开发人员选用 OSGi 技术作为模块化编程的首选。早在 2007 年的时候,由 Sun 公司主导并提交的 JSR-277 (Java 模块化系统)规范并没有通过 JCP 组织的审核,这主要是由于 JCP 专家组织通过投票将 IBM 公司提交的 JSR-291(OSGi R4.1)纳入了 Java 模块化规范标准。直到 Sun 公司在 Java7 早期时,再次提交 JSR-294 (Java 模块化系统的改进支持)规范,可惜最终还是未能如愿。不得已 Sun 公司只能避开 JCP 组织,在 OpenJDK 中创建了一个叫作 Jigsaw 的子项目来实现 Java 模块化编程技术,但该项目却被迫延期到 Java 9 中进行发布。我们先不论 Jigsaw 能否顺利在 Java 9 中出现,就目前而言 OSGi 技术早已是 Java 模块化规范标准,不得不承认 Sun 公司在这一场模块化规范争夺战中是以失败告终的。或许在后续的 Java 版本中,我们能够看到 Jigsaw 与 OSGi 技术的整合,共同为 Java 的模块化编程带来新的体验。

1.4.2 语言无关性

在很多年以前,想要能够在 Java 虚拟机平台上运行非 Java 语言编写的程序,这简直就是天方夜谭,但随着 Java7 的正式发布,这已经不再是一个奢侈而遥远的梦想,Java 虚拟机的设计者们通过 JSR-292 规范基本兑现了这个承诺。随着软件开发领域的复杂度日渐增加,使得开发人员使用单一的 Java 语言进行项目开发已经显得有些力不从心,如果 Java 语言能够与其他编程语言进行混合编程,不但能够弥补自身不足,同时又能够利用其他编程语言的优点很好地解决技术问题,这无疑是一件美好的事情。也就是说某些 Java 语法层面并不支持的特性,并不代表 Java 虚拟机也无法支持,只要其他编程语言能够有效支持,且编译结果是有效的字节码文件,Java 虚拟机就能够将其装载进内部并顺利运行,这就是为什么需要使用混合语言编程的目的和好处。

本书在此所提及的混合编程概念,指的是在同一个虚拟机的宿主环境下,能够同时运行使用 Java 语言以及其他编程语言编写的应用程序。这一切听起来似乎很神奇,Java 虚拟机为什么能够解释出非 Java 语言编写的程序呢?其实 Java 虚拟机根本不关心运行在其内部的程序到底是使用何种编程语言编写的,它只关心“字节码”文件。也就是说 Java 虚拟机拥有语言无关性,并不会单纯地与 Java 语言“终身绑定”,只要其他编程语言的编译结果满足并包含 Java 虚拟机的内部指令集、符号表以及其他的辅助信息,它就是一个有效的字节码文件,就能够被虚拟机所识别并装载运行,如图 1-5 所示。

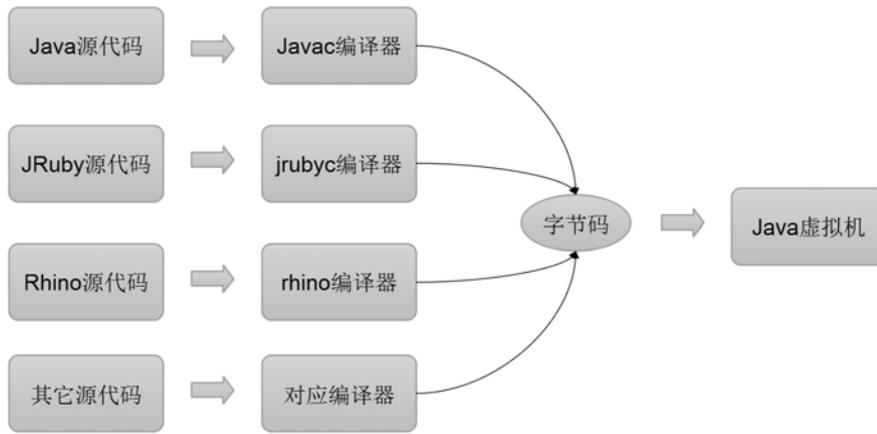


图 1-5 Java 虚拟机的与语言无关特性

如果任何编程语言的编译结果都可以满足字节码的组成结构和存储形式,那么从理论上讲都可以运行在 Java 虚拟机内部。在此大家需要注意,本书所提及的任何编程语言如果希望编译结果是字节码,那么必然需要对应 Java 虚拟机实现的语言版本。比如大家熟知的: Ruby (JRuby)、JavaScript (Rhino)、PHP (IBM WebSphere sMash PHP (P8)、Caucho Quercus) 等优秀的编程语言都具有其对应的虚拟机实现版本。

1.4.3 使用 Fork/Join 框架实现多核并行

随着如今硬件水平的高速发展和提升,就连手机等智能移动设备的 CPU 核心都采用多核超线程技术,而传统的 PC 设备、企业级服务器自然也由早期关注的高频率转换为多核心去处理日常任务。在今天如果一门编程语言不能高效地支持多核并行计算,无论其曾经多么优秀和辉煌,都将注定被开发人员所淘汰,毕竟这是谁也无法阻止的优胜劣汰原则。Java 早在多核时代还没来临的时候,就已经开始支持单核并发计算,但随着多核设备的日渐普及,开发人员更希望 Java 能够充分利用所有可用物理核心一起高效地并行处理任务。所以在 Java5 版本的时候,Java 设计者们通过 JSR-166 的规范制定,在 `java.util.concurrent` 包下为开发人员提供了基于粗粒度的多核并行计算框架,只不过这种基于粗粒度的并行计算模式,并不能在处理效率上达到令人满意的程度。因为这种基于粗粒度的并行计算,根本无法高效组合所有可用物理核心一起进行并行任务处理,甚至在某些情况下,还有可能导致部分物理核心处于空闲等待状态。

由于粗粒度的并行计算并不能够充分挖掘多核处理器的性能,所以 Java 设计者们又对 JSR-166 规范进行了一系列的整改修订,并在 Java7 版本中在 `java.util.concurrent.forkjoin` 包下新增了基于细粒度的多核并行计算 Fork/Join 框架。该框架的设计初衷是将一个任务量化到最小,并提供高计算密度的并行处理性能。简单来说,我们可以将一个任务拆分成若干个

子任务，直到这个任务足够小，然后每一个子任务被独立并行计算，直到任务执行完成，再将其逐个合并为一个完整的任务，这就是 Fork/Join 框架提供的细粒度并行计算模式，如图 1-6 所示。

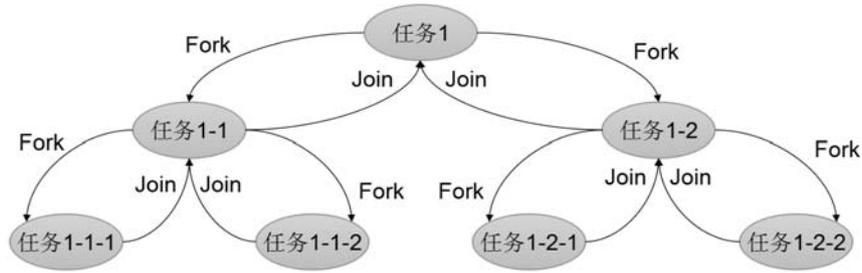


图 1-6 Fork/Join 框架计算模型

除了 Java7 能够高效地利用 Fork/Join 框架实现多核并行计算外，开源基金会 Apache 提供的 Hadoop^⑤ Map/Reduce 框架也是一个高效的海量数据计算框架，它允许开发人员将其部署在廉价的集群服务器上处理 TB 级别的数据。当然还有一些编程语言自诞生那天起，就是为了解决并行计算而来，比如 Scala、Clojure 和 Erlang 等。这类编程语言，不仅继承了面向对象（Object Oriented）的特性，同时还结合了函数式编程等特性。虽然目前 Java 同样也能够使用函数式编程，但代码将会显得非常冗余，不过 Java 设计者们在 Java8 中提供对 Lambda 的支持，这会极大改善 Java 对于函数式编程的不足。

1.4.4 丰富的语法特性

Java7 的新特性更多的是体现在语法层面上的扩充，这对于 Java 语法的易用性和准确性来说将会有非常大的提升。比如 switch 表达式将提供对 String 类型的支持、直接二进制字面值的定义、try-with-resources 实现自动资源管理、cache 语句表达式的改变、泛型的“<>”类型推断运算符支持、全新的文件系统 NIO2.0、Fork/Join 模型等。

Java7 最激动人心的地方无疑是全新的文件系统 NIO2.0 的到来。利用 NIO2.0，开发人员则无需关注 I/O 细节，因为新文件系统中封装有大量的通用操作，便于开发人员更好地关

⑤ Hadoop 是一个分布式系统基础架构，由 Apache 基金会开发。用户可以在不了解分布式底层细节的情况下开发分布式程序，充分利用集群的威力高速运算和存储。Hadoop 实现了一个分布式文件系统（Hadoop Distributed File System，简称 HDFS）。HDFS 有着高容错性的特点，并且设计用来部署在低廉的（low-cost）硬件上。而且它提供高传输率（high throughput）来访问应用程序的数据，适合那些有着超大数据集（large data set）的应用程序。Hadoop 的组成结构比较复杂，它由许多元素构成。其最底部是 HDFS，它存储 Hadoop 集群中所有存储节点上的文件。HDFS（对于本文）的上一层是 MapReduce 引擎，该引擎则由 JobTrackers 和 TaskTrackers 组成。

注自身业务。并且在 I/O 模型方面，将支持调用操作系统的 IOCP（Input/Output Completion Port，输入/输出完成端口）接口实现真正的异步 I/O，尽可能避免因 I/O 问题导致的系统瓶颈出现。

唯一遗憾的就是原本归纳在 Java7 中的 Lambda 表达式，却被迫到 Java8 中进行发布。不过当 Lambda 表达式真正到来时，也意味着 Java 将会走向函数式编程的道路，这对 Java 语法和开发人员的编码习惯都会带来巨大的影响，毕竟函数式编程在未来的几年内很可能成为主流。

1.4.5 过渡到 64 位虚拟机

早在市面上第一款 64 位处理器诞生不久后，Sun 公司随即推出了用于支持 64 位系统的 JDK。相对于传统的 32 位虚拟机，64 位虚拟机所具备的最大优势就是可以访问大内存，相信大家知道 32 位虚拟机的最大可用内存空间被限定在了 4GB，并且 Java 堆区的大小如果是在 Windows 平台下最大只能设置到 1.5GB，而在 Linux 平台下最大也只能设置到 2GB~3GB 的上限，也就是说，Java 堆区的内存大小设置还需要依赖于具体的操作平台。既然 32 位虚拟机无法满足大内存消耗的应用场景，那么 64 位虚拟机的出现则是顺理成章，64 位虚拟机之所以能够访问大内存，是因为其采用了 64 位的指针架构，这也是寻址访问大内存的关键要素。

在 JDK1.6 Update14 版本之前，64 位虚拟机的综合性能表现实际上是不如 32 位虚拟机的，这主要是因为 OOPS（Ordinary Object Pointers，普通对象指针）从 32 位膨胀到 64 位后，CPU Cache Line 中的可用 OOPS 变少，这样一来将会直接影响并降低 CPU 的缓存使用率，这就是 64 位虚拟机在性能上之所以落后于 32 位虚拟机的主要原因。其次由于部署在 64 位虚拟机中的程序大部分都需要用到大内存，尤其是互联网项目，经常需要使用多达几十乃至上百 GB 的内存，这对于传统的 32 位虚拟机将无法承载（部分企业采用 32 位虚拟机集群的方式使用大内存），只能依靠 64 位虚拟机去支撑。但是管理这么大的内存开销对于 GC 来说将会是一场非常严峻的考验，甚至很有可能会导致 GC 在执行内存回收期间消耗更长的时间，同时也就意味着工作线程的等待时间将会延长。随着如今 64 位虚拟机的逐渐成熟，JVM 的设计者们在 JDK1.6 Update14 版本开始提供了指针压缩功能，也就是说，指针压缩将会通过对齐补白等操作将 64 位指针压缩为 32 位，以此改善 CPU 缓存使用率达到提升 64 位虚拟机运行性能的目的。

在此需要提醒大家，如果所使用的 64 位虚拟机的版本在 Update14-Update22 之间，可以通过选项“-XX:+UseCompressedOops”显式开启指针压缩功能，而在 Update23 版本之后，指针压缩功能将会被缺省开启。

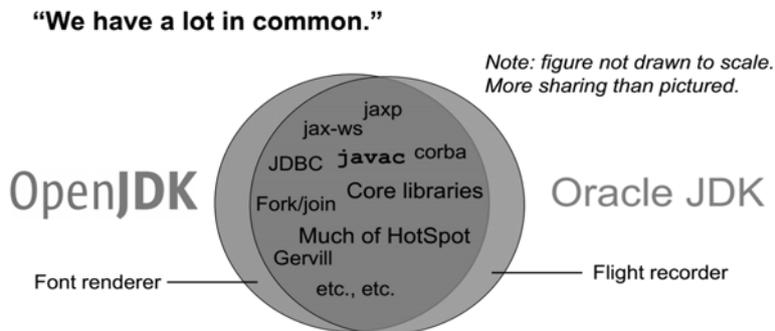
1.5 实战：玩转 OpenJDK

谈到 OpenJDK 相信大家并不会感觉到陌生，在 CentOS、Ubuntu 等常用的 Linux 发行版操作系统上，几乎都预装 OpenJDK 作为缺省的 Java 运行环境。一般来说，使用 OpenJDK 已经完全可以满足大多数的应用场景，笔者公司的项目就是直接部署在 OpenJDK 中运行。当然如果你想要彻底弄清楚 JDK 的内部实现原理，那么亲自动手编译一套 JDK 以及 Debug HotSpot 则是最直接的一种方式。千万不要认为编译 OpenJDK 需要多么深厚的技术功底，实际上这是非常简单的，只需要开发人员掌握最基本的 shell 命令，就可以成功编译出一套自己的 JDK。

在 OpenJDK 中，除了 HotSpot 是使用 C++ 以及混合了少量的 C 和汇编语言编写外，大部分内容其实都是使用 Java 语言编写的，比如 Java 的基础类库等。目前市面上开源的众多 JDK 中，开发人可以选择 Apache Harmony（该项目已于 2011 年 10 月宣布停止开发）、OpenJDK 等进行编译。考虑到 OpenJDK 的应用最为广泛，所以本书就选择使用 OpenJDK 作为这次编译实战的开源 JDK 版本。

1.5.1 JDK 与 OpenJDK 的关系

OpenJDK 简单来说就是 Sun/Oracle JDK 的一种开源版本，但 OpenJDK 却并不属于 Sun 和 Oracle 等商业公司，它属于开源社区，任何组织或个人都可以为推动 Java 未来的技术发展做出贡献。Sun 公司早在 2006 年的时候就宣布会将 JDK 以 GPL v2 的开源协议进行源代码公开，但直到 2009 年 Sun 公司才正式发布第一个开源的 JDK 版本，那就是如今的 OpenJDK。目前开发人员使用的 OpenJDK7 与 Oracle JDK7 中的代码几乎是一模一样的，唯一的区别就是 Oracle JDK7 中的部分代码因为版权问题在 OpenJDK7 中只能使用其他的技术进行替代，除此之外均没有任何区别，如图 1-7 所示^⑥。



⑥ 图片来源于：https://blogs.oracle.com/darcy/resource/OSCON/oscon2011_OpenJDKState.pdf 文档中。

和 OpenJDK7 不同，OpenJDK6 和 Sun JDK6 的源代码则存在较大差异，因为 OpenJDK6 仅仅只是 OpenJDK7 的一个分支，为了尽量符合 Java6 的标准，OpenJDK6 中删除了所有关于 Java7 的新特性。并且 OpenJDK 和 Sun/Oracle JDK 的源代码开源协议也不同，OpenJDK 采用的是 GPL V2 协议，而 Sun/Oracle JDK 采用的则是 JRL（JavaResearch License）协议。采用 GPL V2 协议发布的 OpenJDK 源代码允许商业用途，而采用 JRL 协议发布的 Sun/Oracle JDK 源代码则仅限于个人研究使用。除此之外 OpenJDK 中并不会包含 Deployment 功能，比如 Browser Plugin、Java WebStart 和 Java 控制面板等，因此开发人员无法在 OpenJDK 中使用这些功能。除了无法使用 Deployment 功能外，OpenJDK 中也不会包含 Rhino 和 Java DB 等工具，也就是说，OpenJDK 其实只是一个精简版的 JDK 而已，由于 OpenJDK 的不完整性，因此被认定为是非标准版的 JDK 版本，所以也就无法正常使用 Java 商标。但如果使用 Icedtea 补丁的 OpenJDK 版本，当输入命令“java -version”后，则会显示 Java 版本号，而非 OpenJDK 版本号。

尽管 OpenJDK 在某些方面和 Sun/Oracle JDK 还存在一定差异，但已经完全可以满足大多数的应用场景，并且 OpenJDK 还为开发人员研究 JDK 的内部实现原理提供了便捷。

1.5.2 基于 OpenJDK 深度定制的淘宝 JVM (TaobaoVM)

使用 Java 技术编写的系统，无疑在生产环境中需要对 Java 虚拟机进行正常的调优工作，既然谈到 Java 虚拟机的调优技术，笔者相信大部分的开发人员至今仍然仅停留在参数调制上。由于淘宝目前无疑是中国最大的 Java 技术应用方，那么淘宝究竟是采用什么样的技术对 Java 虚拟机进行优化的呢？淘宝的技术团队对 Java 虚拟机的优化工作其实早已不是停留在简单的参数调制上面，而是充分结合了企业自身的业务特点以及实际的应用场景，在 OpenJDK 的基础之上通过修改大量的 HotSpot 源代码，深度定制了淘宝专属的高性能 Java 虚拟机——TaobaoVM。

既然是结合业务特点深度定制的一款 Java 虚拟机，那么性能必然在某一些特定的应用场景上会比 Oracle 官方的 HotSpot 更强，如图 1-8 所示。但其弊端同样也非常明显，那就是无法实现通用。所以如果只是想对 TaobaoVM 进行研究的话，可以参考 jvm.taobao.org 中的描述编译一个 TaobaoVM，但如果需要应用在实际的项目中，笔者还是建议三思而后行，否则将会得不偿失。

淘宝的技术团队通过修改大量的 HotSpot 源代码深度定制的 TaobaoVM^⑦，其实从严格意义上来说，在提升 Java 虚拟机性能的同时，却严重依赖物理 CPU 类型。也就是说，部署有 TaobaoVM 的服务器中，CPU 全都是清一色的 Intel CPU，且编译手段采用的是 Intel C/CPP

⑦ 文章完整地址：<http://os.51cto.com/exp/velocity2012/ppt/wangcheng.pdf>。

Compiler 进行编译，以此对 GC 性能进行提升。除了优化编译效果外，TaobaoVM 还使用了 crc32 指令实现 JVM intrinsic 降低 JNI 的调用开销，如图 1-9 所示。

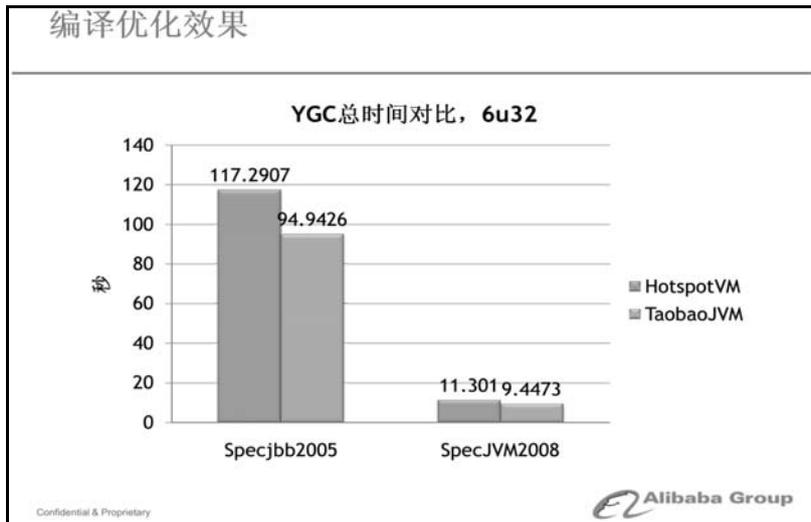


图 1-8 TaobaoVM 的优化编译效果

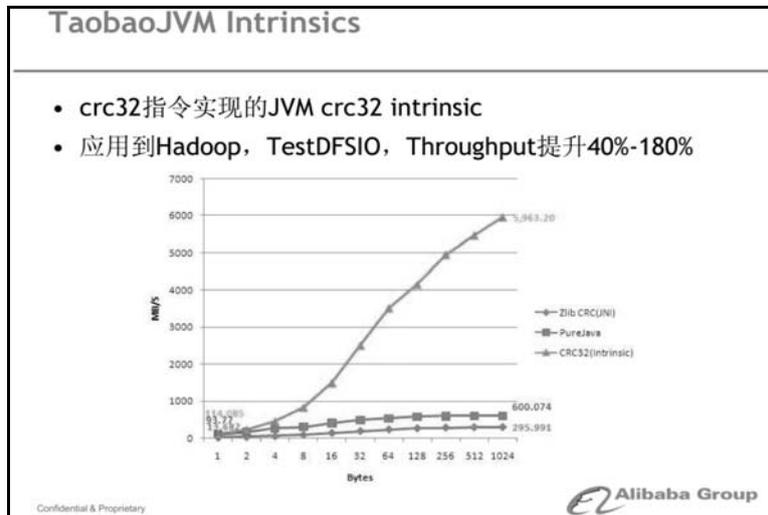


图 1-9 JVM intrinsic 降低 JNI 的调用开销

除了在性能优化方面下足了功夫，TaobaoVM 还在 HotSpot 的基础之上大幅度扩充了一些特定的增强实现。比如创新的 GCIH (GC invisible heap) 技术实现 off-heap，这样一来就可以将生命周期较长的 Java 对象从 heap 中移至 heap 之外，并且 GC 不能管理 GCIH 内部的 Java 对象，这样做最大的好处就是降低了 GC 的回收平率以及提升了 GC 的回收效率，并且

GCIH 中的对象还能够多个 Java 虚拟机进程中实现共享。其他扩充技术还有利用 PMU hardware 的 Java profiling tool 和诊断协助功能等。

1.5.3 下载 OpenJDK 源代码

大家可以登录 <http://openjdk.java.net/> 下载 OpenJDK 的源代码，本书编译实战所演示的 JDK 版本为 openjdk7-u40。当然你也可以选择下载其他版本的 OpenJDK，但是为了避免在编译过程中发生一些不必要的错误，所以建议大家尽量保持和本书一致的 JDK 版本。

想要下载 OpenJDK 的源代码一共有两种方式。一种是直接使用 Mercurial 版本管理工具从 Repository 中 check out 出源代码（地址为：<http://hg.openjdk.java.net/jdk7u/jdk7u40>）。但由于绝大多数的 Java 开发人员或许并不熟悉 Mercurial 工具的使用，以及从 Repository 中 check out 出源代码耗时太久，所以除了采用这种方式外，大家还可以采用手动下载的方式从 <http://download.java.net/openjdk/jdk7u40> 下载 OpenJDK 的源代码。

当成功下载好 OpenJDK 的 zip 包并使用命令“unzip”将其进行解压后，便可以在解压后的 OpenJDK 根目录中找到 README-builds.html。该文档中详细地描述了 OpenJDK 的完整编译步骤，所以如果你是第一次编译 OpenJDK，那么阅读该文档则会事半功倍。

1.5.4 构建编译环境

前面提到过，如果想要彻底弄清楚 JDK 的内部实现原理，亲自动手编译一套 JDK 及 Debug HotSpot 则是最直接的一种方式。但是在执行编译之前，构建编译环境则显得至关重要，因为编译环境直接决定了编译结果，如果没有构建出完整的编译环境则无法成功编译 JDK。

大家几乎可以在任何的系统平台上搭建 OpenJDK 的编译环境，比如 Linux 平台、Mac OS 平台甚至是 Window 平台上都可以构建编译环境。只不过想要在 Window 平台上构建编译环境则显得非常麻烦，而且还容易出错。因为 Window 平台不仅需要依赖 Cygwin 等虚拟机环境，更重要的是无法确保构建环境期间的正确性，所以本书选择在 Linux 平台上构建编译环境，相对 Windows 来说不仅构建成本低廉，还更简单和直接。

本书选用的 Linux 发行版为 ubuntu-12.04.1-x86，其 Linux 内核版本为 3.2。当成功准备好系统平台后，接下来的任务就是构建编译环境。想要成功编译出 JDK，还需要下载和安装 Bootstrap JDK、GCC、Make 和 Ant 等一系列的项目依赖。在 OpenJDK 中除了 HotSpot 是使用 C++ 以及混合了少量的 C 和汇编语言编写外，大部分内容其实都是使用 Java 语言编写的，比如 Java 的基础类库等，所以这时就需要用到用于编译 Java 代码的 JDK，我们可以

将其称之为 Bootstrap JDK。而 C/C++ 编写的代码则需要使用 GCC 编译器进行编译，在 Ubuntu 中 GCC 缺省就已经安装了。当 OpenJDK 中的所有源代码都已经成功编译后，剩下的任务就是 Build 操作。使用 Bootstrap JDK 编译的 Java 代码需要使用 Ant 工具进行 Build，而使用 GCC 编译的 C/C++ 代码则需要使用 Make 工具进行 Build。

本书编译实战所使用的项目依赖版本如下：

- Bootstrap JDK 版本：openjdk6-b31；
- GCC 编译器版本为：4.6.3；
- Make 工具版本为：3.8.1；
- Ant 工具版本为：1.8.2。

如果不想手动下载编译 JDK 所需的一系列项目依赖，则可以使用如下命令一次性下载和安装完成，如下所示：

```
sudo apt-get install build-essential gawk m4 openjdk-6-jdk
libasound2-dev libcups2-dev libxrender-dev xorg-dev xutils-dev
X11proto-print-dev binutils libmotif3 libmotif-dev ant
```

当成功执行上述命令后，则会自动下载和安装编译 OpenJDK 所需的一系列项目依赖，这样就可以将大家从繁琐的手动操作中解放出来，全身心地投入到编译任务中去。

1.5.5 执行整个 OpenJDK 的编译

当成功构建好 OpenJDK 所需的编译环境后，最后还需要设置一些环境变量参数，因为在编译 OpenJDK 的时候，需要用到这些环境变量做引导。参考 README-builds.html 文档中有关环境变量的设置，其中只有 LANG 和 ALT_BOOTDIR 这两个环境变量必须手动设置外，其他环境变量参数在编译 OpenJDK 时均可以使用默认设置，如下所示：

代码 1-2 设置编译 OpenJDK 所需的环境变量参数

```
#设定语言选项
export LANG=C

#定义 Bootstrap JDK 的路径
export ALT_BOOTDIR=/usr/lib/jvm/java-6-openjdk

#允许在编译过程中自动下载相关依赖

export ALLOW_DOWNLOADS=true

#并行编译策略时开启的线程数量
export HOST_BUILD_JOBS=4
```

```

#使用预编译头文件，加速编译
export USE_PRECOMPILED_HEADER=true

#编译内容，如果不设置，则编译 OpenJDK 中的所有内容，但耗时较长
export BUILD_LANGTOOLS=true
export BUILD_JAXP=true
export BUILD_JAXWS=true
export BUILD_CORBA=true
export BUILD_HOTSPOT=true
export BUILD_JDK=true

#禁止 build 出安装包
BUILD_INSTALL=false

#编译后的存储路径
export ALT_OUTPUTDIR=/home/johngao/openjdk7/build

#禁止编译器把一些警告当错误
WARNINGS_ARE_ERRORS=

#执行编译，并将编译过程中产生的日记进行存储
make 2>&1 | tee /home/johngao/openjdk7/build-log/openjdk.log

```

当成功设置好编译 OpenJDK 所需的环境变量后，为了避免每次都手动去执行，笔者建议在解压后的 OpenJDK 根目录中创建一个 shell 脚本，将上述内容粘贴进去。为了验证上述环境变量配置的正确性，可以使用命令“make sanity”进行检测，如下所示：

代码 1-3 使用 make sanity 检测

```

root@johngao-virtual-machine:/home/johngao/openjdk7/src/openjdk# ./build.sh
Build Machine Information:
  build machine = johngao-virtual-machine

Build Directory Structure:
  CWD = /home/johngao/openjdk7/src/openjdk
  TOPDIR = .
  LANGTOOLS_TOPDIR = ./langtools
  JAXP_TOPDIR = ./jaxp
  JAXWS_TOPDIR = ./jaxws
  CORBA_TOPDIR = ./corba
  HOTSPOT_TOPDIR = ./hotspot
  JDK_TOPDIR = ./jdk

Build Directives:
  BUILD_LANGTOOLS = true
  BUILD_JAXP = true

```

```
BUILD_JAXWS = true
BUILD_CORBA = true
BUILD_HOTSPOT = true
BUILD_JDK = true
DEBUG_CLASSFILES =
DEBUG_BINARIES =

Hotspot Settings:
  HOTSPOT_BUILD_JOBS =
  HOTSPOT_OUTPUTDIR = /home/johngao/openjdk7/build/hotspot/outputdir
  HOTSPOT_EXPORT_PATH = /home/johngao/openjdk7/build/hotspot/import

Bootstrap Settings:
  BOOTDIR = /usr/lib/jvm/java-6-openjdk
  ALT_BOOTDIR = /usr/lib/jvm/java-6-openjdk
  BOOT_VER = 1.6.0 [requires at least 1.6]
  OUTPUTDIR = /home/johngao/openjdk7/build
  ALT_OUTPUTDIR = /home/johngao/openjdk7/build
  ABS_OUTPUTDIR = /home/johngao/openjdk7/build

—省去部分日志内容—

OpenJDK-specific settings:
  FREETYPE_HEADERS_PATH = /usr/include
  ALT_FREETYPE_HEADERS_PATH =
  FREETYPE_LIB_PATH = /usr/lib
  ALT_FREETYPE_LIB_PATH =

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 = /usr/lib/jvm/java-6-openjdk
  ALT_PREVIOUS_RELEASE_IMAGE =

Sanity check passed.
```

输入命令“make sanity”后，如果输出日志最后提示“Sanity check passed.”，且没有任何警告信息时，则意味着环境变量配置正常可以执行 make 操作。在此需要提醒大家，全量编译整个 OpenJDK 时，编译过程会相当耗时，笔者全量编译大概耗费了一个小时左右的时间，所以大家只需在编译过程中耐心等待即可。当然如果只是增量编译，编译过程则会比较迅速。

当成功编译完 OpenJDK 后，如果输出日志最后提示如下信息则意味着你已经成功编译好了一个完整的 OpenJDK，如下所示：

```
#-- Build times -----
Target all_product_build
Start 2014-06-28 15:51:42
End   2014-06-28 16:59:30
00:01:57 corba
00:37:57 hotspot
00:00:35 jaxp
00:00:38 jaxws
00:25:36 jdk
00:01:03 langtools
01:07:48 TOTAL
-----
```

从上述的输出日志中可以发现，笔者编译整个 OpenJDK 一共耗时 1 小时 07 分 48 秒，其中 HotSpot 和 JDK 的编译过程耗时最久，大概占了整个编译周期的 95%。当成功编译好 OpenJDK 后，大家可以通过环境变量 ALT_OUTPUTDIR 中配置的存储路径找到编译后的 OpenJDK 根目录，在根目录下的 j2sdk-image 目录中，所存放的内容就是编译后的 OpenJDK。

当我们输入命令“java -version”时，便可以查看由自己亲手编译的 OpenJDK 的版本信息，如下所示：

```
root@johngao-virtual-machine:/home/johngao/openjdk7/build/j2sdk-image/
bin# ./java -version
openjdk version "1.7.0-internal"
OpenJDK Runtime Environment (build 1.7.0-internal-root_2014_06_28_15_51-b00)
OpenJDK Client VM (build 24.0-b56, mixed mode)
root@johngao-virtual-machine:/home/johngao/openjdk7/build/j2sdk-image/bin#
```

1.5.6 执行单独 HotSpot 的编译

其实编译 OpenJDK 真正有吸引力的地方是在 HotSpot 的编译部分，而非整个 OpenJDK，所以如果你只是想在成功编译好 HotSpot 后进行 Debug，则可以在 OpenJDK 源代码根目录下的/hotspot/make 目录中使用 Make 命令执行 Makefile 脚本即可。当然编译 HotSpot 和编译 OpenJDK 类似，都需要在编译时设置 LANG 和 ALT_BOOTDIR 这 2 个环境变量，如下所示：

代码 1-4 设置编译 HotSpot 所需的环境变量参数

```
#设定语言选项
export LANG=C
```

```
#定义 Bootstrap JDK 的路径
export ALT_BOOTDIR=/usr/lib/jvm/java-6-openjdk

#编译后的存储路径
export ALT_OUTPUTDIR=/home/johngao/openjdk7/build-jvm

make jvmg jvmg1 2>&1 | tee /home/johngao/openjdk7/build-log/hotspot.log
```

无论是编译整个 OpenJDK 还是只编译 HotSpot，需要设置的环境变量始终只有两个是必需的，所以笔者在编译 HotSpot 时，只设置了 LANG 和 ALT_BOOTDIR 以及指定编译后存储路径的 ALT_OUTPUTDIR 环境变量。

在/hotspot/make 目录下的 Makefile 脚本中，定义了 HotSpot 的编译目标类型，其中主要包括了 product、optimized、fastdebug 和 debug 等 4 种级别，如下所示：

代码 1-5 Makefile 脚本中 HotSpot 的编译目标类型

```
# Typical C1/C2 targets made available with this Makefile
C1_VM_TARGETS=product1 fastdebug1 optimized1 jvmg1
C2_VM_TARGETS=product fastdebug optimized jvmg
ZERO_VM_TARGETS=productzero fastdebugzero optimizedzero jvmgzero
SHARK_VM_TARGETS=productshark fastdebugshark optimizedshark jvmgshark

COMMON_VM_PRODUCT_TARGETS=product product1 docs export_product
COMMON_VM_FASTDEBUG_TARGETS=fastdebug fastdebug1 docs export_fastdebug
COMMON_VM_DEBUG_TARGETS=jvmg jvmg1 docs export_debug

# JDK directory list
JDK_DIRS=bin include jre lib demo

all:          all_product all_fastdebug

ifdef BUILD_CLIENT_ONLY
all_product:  product1 docs export_product
all_fastdebug: fastdebug1 docs export_fastdebug
all_debug:    jvmg1 docs export_debug
else
ifeq ($(MACOSX_UNIVERSAL),true)
all_product:  universal_product
all_fastdebug: universal_fastdebug
all_debug:    universal_debug
else
all_product:  $(COMMON_VM_PRODUCT_TARGETS)
all_fastdebug: $(COMMON_VM_FASTDEBUG_TARGETS)
all_debug:    $(COMMON_VM_DEBUG_TARGETS)
endif
```

```

endif

all_optimized: optimized optimized1 docs export_optimized

allzero:          all_productzero all_fastdebugzero
all_productzero:  productzero docs export_product
all_fastdebugzero: fastdebugzero docs export_fastdebug
all_debugzero:    jvmgzero docs export_debug
all_optimizedzero: optimizedzero docs export_optimized

allshark:         all_productshark all_fastdebugshark
all_productshark: productshark docs export_product
all_fastdebugshark: fastdebugshark docs export_fastdebug
all_debugshark:   jvmgshark docs export_debug
all_optimizedshark: optimizedshark docs export_optimized

```

由于编译后的 HotSpot 需要进行调试，所以笔者使用的 Make 命令为“make jvmg jvmg1”，这样一来就确定了 HotSpot 的编译目标类型是 debug 级别。

当成功编译好 HotSpot 后，大家可以通过环境变量 ALT_OUTPUTDIR 中配置的存储路径找到编译后的 HotSpot 根目录，在根目录中包含 linux_i486_compiler1 和 linux_i486_compiler2。其中 linux_i486_compiler1 目录下的内容是编译后的 Client VM，而 linux_i486_compiler2 目录下的内容则是编译后的 Server VM。

在 jvmg 目录中执行命令“./test_gamma”则可验证 HotSpot 的编译结果，如下所示：

```

root@johngao-virtual-machine:/home/johngao/openjdk7/build-jvm/
linux_i486_compiler2/jvmg# ./ test_gamma
Using java runtime at: /home/johngao/openjdk7/build/j2sdk-image/jre
openjdk version "1.7.0-internal"
OpenJDK Runtime Environment (build 1.7.0-internal-root_2014_06_28_15_51-b00)
OpenJDK Server VM (build 24.0-b56-internal-jvmg, mixed mode)

1. A1 B5 C8 D6 E3 F7 G2 H4
2. A1 B6 C8 D3 E7 F4 G2 H5
3. A1 B7 C4 D6 E8 F2 G5 H3
4. A1 B7 C5 D8 E2 F4 G6 H3
5. A2 B4 C6 D8 E3 F1 G7 H5
6. A2 B5 C7 D1 E3 F8 G6 H4
88. A7 B5 C3 D1 E6 F8 G2 H4

—省去部分日志内容—

90. A8 B2 C5 D3 E1 F7 G4 H6
91. A8 B3 C1 D6 E2 F5 G7 H4
92. A8 B4 C1 D3 E6 F2 G7 H5

```

尽管已经成功编译好 HotSpot，但是在运行之前，我们还需要修改 `jvmg` 目录下的 `env.sh` 脚本。该脚本中已经包含了 `JAVA_HOME`、`CLASSPATH` 和 `HOTSPOT_BUILD_USER` 这 3 个环境变量，但还需新增一个 `LD_LIBRARY_PATH` 环境变量，如下所示：

代码 1-6 设置 `LD_LIBRARY_PATH` 环境变量

```
: ${JAVA_HOME:=/home/johngao/openjdk7/build/j2sdk-image}
CLASSPATH=.:${JAVA_HOME}/jre/lib/rt.jar:${JAVA_HOME}/jre/lib/i18n.jar
HOTSPOT_BUILD_USER="root in hotspot"
LD_LIBRARY_PATH=.:${JAVA_HOME}/jre/lib/i386/native_threads:
    ${JAVA_HOME}/jre/lib/i386
export JAVA_HOME CLASSPATH HOTSPOT_BUILD_USER LD_LIBRARY_PATH
```

当成功配置好 `LD_LIBRARY_PATH` 环境变量后，则可以执行 `env.sh` 脚本启动 HotSpot，并使用命令 “`gamma -version`” 查看由自己亲手编译的 HotSpot 的版本信息，如下所示：

```
root@johngao-virtual-machine:/home/johngao/openjdk7/build-jvm/
linux_i486_compiler2/jvmg# ./env.sh
root@johngao-virtual-machine:/home/johngao/openjdk7/build-jvm/
linux_i486_compiler2/jvmg# ./ gamma -version
Using java runtime at: /home/johngao/openjdk7/build/j2sdk-image/jre
openjdk version "1.7.0-internal"
OpenJDK Runtime Environment (build 1.7.0-internal-root_2014_06_28_15_51-b00)
OpenJDK Server VM (build 24.0-b56-internal-jvmg,mixed mode)
```

1.5.7 导致编译失败的一些疑难杂症

如果你是第一次尝试编译整个 OpenJDK 或者单独编译 HotSpot，几乎很难做到一次性顺利通过编译。编译失败倒也在情理之中，因为构建编译环境时项目依赖较多（比如：Bootstrap JDK、GCC、Make 和 Ant 等一系列的项目依赖），缺少依赖或者版本问题都会影响编译结果。本书列举了 3 种比较常见的编译错误以及对应的解决方案，供大家参考。

1. 项目依赖不完整

在执行编译前，大家尽量确保所有的项目依赖都已经成功下载并安装。如果需要在编译过程中自动下载项目依赖，则可以添加环境变量 `ALLOW_DOWNLOADS`，如下所示：

```
export ALLOW_DOWNLOADS=true
```

2. 项目依赖版本问题

在编译 OpenJDK7 的时候，大家一定要注意 Bootstrap JDK、GCC、Make 和 Ant 等依赖的版本。其中 Bootstrap JDK 的版本必须使用 `jdk-update-14` 或更高版本，GCC 编译器的版本必须使用 4.3 或更高版本，而 Make 工具的版本必须使用 3.8.1 或更高版本，最后 Ant 工具

的版本必须使用 1.7.1 或更高版本。

3. Linux 内核版本过高

我们可以打开 `/hotspot/make/linux` 目录中的 `Makefile` 脚本，查看 `SUPPORTED_OS_VERSION` 所支持的所有 Linux 内核版本号。如果你当前的 Linux 内核版本高于 `Makefile` 脚本中所能支持的上限，则可以在末尾添加实际使用的 Linux 版本号，如下所示：

```
SUPPORTED_OS_VERSION = 2.4% 2.5% 2.6% 3.2%
```

除了可以使用上述方式外，还可以在 `Makefile` 脚本中，通过注释代码的方式，在编译时绕开 Linux 内核版本的检查步骤，如下所示：

```
check_os_version:
#ifeq ($(DISABLE_HOTSPOT_OS_VERSION_CHECK)$(EMPTY_IF_NOT_SUPPORTED),)
#    $(QUIETLY) >&2 echo "*** This OS is not supported:" `uname -a`; exit 1;
#endif
```

尽管本书列举了 3 种比较常见的编译错误以及对应的解决方案，但编译过程中仍然有太多无法预料的问题，所以当出现了其他问题而无法通过上述解决方案进行解决时，则需要大家寻求其他帮助。

1.5.8 使用 GDB 工具 Debug HotSpot

GDB 是 GUN 开源组织发布的一款用于在 Unix/Linux 下调试 C/C++ 代码的脚本调试工具，GDB 不仅使用简单且功能强大，只需牢记 GDB 的常用调试命令便可以熟练地使用 GDB 工具。当然除了可以使用 GDB 工具 Debug HotSpot 外，我们还可以使用装有插件的 IDE 工具，比如：Eclipse、NetBeans 等。尽管使用拥有图形化界面的 IDE 工具会更加方便，但使用 GDB 工具却更加灵活，所以本书就以 GDB 为例，为大家演示如何在 Linux 环境下 Debug HotSpot。

既然是 Debug HotSpot，那么调试脚本则是必不可少的。在 `jvms` 目录中的 `hotspot` 就是调试脚本，脚本中设定有大量的调试信息方便大家进行调试，如下所示：

代码 1-7 hotspot 脚本中的调试信息

```
# This is the name of the gdb binary to use
if [ ! "$GDB" ]
then
    GDB=gdb
fi

# This is the name of the gdb binary to use
```

```
if [ ! "$DBX" ]
then
    DBX=dbx
fi
# This is the name of the Valgrind binary to use
if [ ! "$VALGRIND" ]
then
    VALGRIND=valgrind
fi

# This is the name of Emacs for running GUD
EMACS=emacs

# Make sure the paths are fully specified, i.e. they must begin with /.
REL_MYDIR=`dirname $0`
MYDIR=`cd $REL_MYDIR && pwd`

# Look whether the user wants to run inside gdb
case "$1" in
    -gdb)
        MODE=gdb
        shift
        ;;
    -gud)
        MODE=gud
        shift
        ;;
    -dbx)
        MODE=dbx
        shift
        ;;
    -valgrind)
        MODE=valgrind
        shift
        ;;
    *)
        MODE=run
        ;;
esac

# HotSpot 依赖的 JDK 路径
JDK=
if [ "${ALT_JAVA_HOME}" = "" ]; then
    . ${MYDIR}/jdkpath.sh
else
    JDK=${ALT_JAVA_HOME%%/jre};
fi
if [ "${JDK}" = "" ]; then
    echo Failed to find JDK. ALT_JAVA_HOME is not set or ./
```

```
    jdkpath.sh is empty or not found.
  exit 1
fi

# We will set the LD_LIBRARY_PATH as follows:
#   o   $JVMPATH (directory portion only)
#   o   $JRE/lib/$ARCH
# followed by the user's previous effective LD_LIBRARY_PATH, if
# any.
JRE=$JDK/jre
JAVA_HOME=$JDK
export JAVA_HOME

ARCH=i386
SBP=${MYDIR}:${JRE}/lib/${ARCH}

# Set up a suitable LD_LIBRARY_PATH or DYLD_LIBRARY_PATH
OS=`uname -s`
if [ "${OS}" = "Darwin" ]
then
  if [ -z "$DYLD_LIBRARY_PATH" ]
  then
    DYLD_LIBRARY_PATH="$SBP"
  else
    DYLD_LIBRARY_PATH="$SBP:$DYLD_LIBRARY_PATH"
  fi
  export DYLD_LIBRARY_PATH
else
  # not 'Darwin'
  if [ -z "$LD_LIBRARY_PATH" ]
  then
    LD_LIBRARY_PATH="$SBP"
  else
    LD_LIBRARY_PATH="$SBP:$LD_LIBRARY_PATH"
  fi
  export LD_LIBRARY_PATH
fi

JPARMS="$@ $JAVA_ARGS";

# Locate the gamma development launcher
LAUNCHER=${MYDIR}/gamma
if [ ! -x $LAUNCHER ] ; then
  echo Error: Cannot find the gamma development launcher \"$LAUNCHER\"
  exit 1
fi

GDBSRCDIR=$MYDIR
BASEDIR=`cd $MYDIR/../../.. && pwd`
```

```
init_gdb() {
# Create a gdb script in case we should run inside gdb
  GDBSCR=/tmp/hsl.$$
  rm -f $GDBSCR
  cat >>$GDBSCR <<EOF
cd `pwd`
handle SIGUSR1 nostop noprint
handle SIGUSR2 nostop noprint
set args $JPARMS
file $LAUNCHER
directory $GDBSRC DIR

# 当执行到 InitializeJVM 函数的时候执行断点
break InitializeJVM
run
# Stop in InitializeJVM
delete 1
# We can now set breakpoints wherever we like
EOF
}

case "$MODE" in
  gdb)
    init_gdb
      $GDB -x $GDBSCR
    rm -f $GDBSCR
      ;;
  gud)
    init_gdb

# First find out what emacs version we're using, so that we can
# use the new pretty GDB mode if emacs -version >= 22.1
case ` $EMACS -version 2> /dev/null ` in
  *GNU\ Emacs\ 2[23]*)
    emacs_gud_cmd="gdba"
    emacs_gud_args="--annotate=3"
      ;;

  *)
    emacs_gud_cmd="gdb"
    emacs_gud_args=
      ;;
esac
  $EMACS --eval "($emacs_gud_cmd \"$GDB $emacs_gud_args -x $GDBSCR\")";
  rm -f $GDBSCR
      ;;
  dbx)
    $DBX -s $MYDIR/.dbxrc $LAUNCHER $JPARAMS
```

```

;;
valgrind)
    echo Warning: Defaulting to 16Mb heap to make Valgrind run faster,
    use -Xmx for larger
    heap
    echo
    $VALGRIND --tool=memcheck --leak-check=yes --num-callers=
    50 $LAUNCHER -Xmx16m $JPARMS
    ;;
run)
    LD_PRELOAD=$PRELOADING exec $LAUNCHER $JPARMS
    ;;
*)
    echo Error: Internal error, unknown launch mode \"$MODE\"
    exit 1
    ;;
esac
RETVAL=$?
exit $RETVAL

```

在 Debug HotSpot 之前，我们还需要检查 hotspot 脚本中所依赖的 JDK 版本。打开 `jdkpath.sh` 脚本即可查看到依赖的 JDK 版本信息，如下所示：

```
JDK=/home/johngao/openjdk7/build/j2sdk-image
```

当确认好 JDK 的版本后，便可使用命令“`./hotspot -gdb TestDemo`”进行 Debug HotSpot，如下所示：

```

root@johngao-virtual-machine:/home/johngao/openjdk7/build-jvm/
linux_i486_compiler2/jvmg# ./hotspot -gdb TestDemo
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>.
Breakpoint 1 at 0x804bd5d: file /home/johngao/openjdk7/src/openjdk/
hotspot/src/share/tools/launcher/java.c, line 1270.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".
Using java runtime at: /home/johngao/openjdk7/build/j2sdk-image/jre
[New Thread 0xb6a15b40 (LWP 6321)]
[Switching to Thread 0xb6a15b40 (LWP 6321)]

Breakpoint 1, InitializeJVM (pvm=0xb6a152c8, penv=0xb6a152cc, ifn=0xb6a152c0)

```

Java 虚拟机精讲

```
at /home/johngao/openjdk7/src/openjdk/hotspot/src/share/tools/launcher
/java.c:1270
1270     memset(&args, 0, sizeof(args));
(gdb) list
1265 InitializeJVM(JavaVM **pvm, JNIEnv **penv, InvocationFunctions *ifn)
1266 {
1267     JavaVMInitArgs args;
1268     jint r;
1269
1270     memset(&args, 0, sizeof(args));
1271     args.version = JNI_VERSION_1_2;
1272     args.nOptions = numOptions;
1273     args.options = options;
1274     args.ignoreUnrecognized = JNI_FALSE;
(gdb)
```

由于在 hotspot 脚本中设置了断点操作，所以当 HotSpot 执行到 InitializeJVM()函数时则会停止下来。如果希望 HotSpot 继续执行下一行代码，则可以使用命令“step”或者“next”。其中命令“step”类似于 Eclipse Debug 的按键 F5，也就是说该命令会进入函数。而命令“next”则类似于按键 F6，但是不会进入函数。如下所示：

```
(gdb) next
1271     args.version = JNI_VERSION_1_2;
(gdb) next 5
1288     r = ifn->CreateJavaVM(pvm, (void **)penv, &args);
(gdb) step
[New Thread 0x9f7d4b40 (LWP 6296)]
[New Thread 0x9f753b40 (LWP 6297)]
[New Thread 0x9f5ffb40 (LWP 6298)]
[New Thread 0x9f5aeb40 (LWP 6299)]
[New Thread 0x9edffb40 (LWP 6300)]
[New Thread 0x9ebffb40 (LWP 6301)]
[New Thread 0x9f55db40 (LWP 6302)]
[New Thread 0x9e9ffb40 (LWP 6303)]
1289     JLI_MemFree(options);
(gdb)
```

如果希望程序直接运行，则可以使用命令“continue”，如下所示：

```
(gdb) continue
Continuing.
Hello HotSpot...
[Thread 0x9e9ffb40 (LWP 6303) exited]
[Thread 0x9f5aeb40 (LWP 6299) exited]
[Thread 0x9f7d4b40 (LWP 6296) exited]
[Thread 0xb6a15b40 (LWP 6295) exited]
[Thread 0x9ebffb40 (LWP 6301) exited]
[Thread 0x9edffb40 (LWP 6300) exited]
```

```
[Thread 0x9f5ffb40 (LWP 6298) exited]
[Thread 0x9f753b40 (LWP 6297) exited]
[Thread 0xb6a16700 (LWP 6292) exited]
[Inferior 1 (process 6292) exited normally]
```

当 HotSpot 的代码全部执行完后，便会输出 Java 对象 TestDemo 中的字符串信息。如果大家熟练掌握 GDB 工具的使用，就可以结合 HotSpot 的源代码尽情 Debug HotSpot，相对于枯燥乏味地直接阅读源代码，采用 Debug 的方式则更容易加深大家对 HotSpot 底层实现原理的理解。

1.6 本章小结

本章笔者详细地介绍了 Java 的体系结构、Java 运行环境的安装和配置，以及 Java 技术未来的发展趋势，让大家对 Java 技术有了一个整体的认识和定位。而在本章的实战小节中，笔者更是重点讲解了如何编译出一套自己的 OpenJDK，并使用 GDB 工具 Debug HotSpot。作为全书的开篇，笔者在本章中并未安插一些较为复杂的概念或技术，而是会在后续章节中循序渐进，为大家讲解字节码的编译原理、HotSpot 的初始化过程、内存分配与垃圾回收、执行引擎等于虚拟机相关的方方面面，让大家更为全面地了解 Java 虚拟机。

第 8 章

剖析HotSpot的架构 模型与执行引擎

对于计算机而言，它能够做到的仅仅只是识别本地机器指令，对于那些使用编程语言编写的源代码则首先需将其编译为对应平台的本地机器指令之后计算机才能够进行正常识别和执行。随着时间的逐步推移，编程语言正在一步一步地朝着人类的思维方式日新月异的进化着，在面向对象等高级语言层出不穷的今天，诸如 Java 之类的编程语言的源代码编译结果并非还是本地机器指令，而是中间代码。这意味着编译结果一旦不依赖于特定的平台后，完全可以做到体系结构中立，一次编译处处运行（Write Once, Run Anywhere）。

既然 Java 代码的编译结果是字节码，那么这就需要一种运行介质能够让其高效运行起来，毕竟计算机并不能够直接识别这些中间代码。HotSpot VM 是目前市面上高性能虚拟机的代表作之一。它采用解释器与即时编译器并存的架构，当虚拟机启动的时候，解释器可以首先发挥作用，而不必等待即时编译器全部编译完成再执行，这样可以省去许多不必要的编译时间。并且随着程序运行时间的推移，即时编译器逐渐发挥作用，根据热点探测功能，将有价值的字节码编译为本地机器指令，以换取更高的程序执行效率。在今天，Java 程序的运行性能早已脱胎换骨，已经达到了可以和 C/C++ 程序一较高下的地步，并且 Java 技术自身的诸多优势同样也是 C/C++ 无法比拟的。

8.1 栈帧的组成结构

在 Java 虚拟机规范中，Java 栈（Java Stack）也可以被称之为 Java 虚拟机栈（Java Virtual Machine Stack），它同 PC 寄存器一样都是线程私有的，并且生命周期与线程的生命周期保持一致。Java 栈主要用于存储栈帧（Stack Frame），而栈帧中则负责存储局部变量表、操作

数栈、动态链接和方法返回值等信息。

在面向对象（Object Oriented）的世界中，类（Class）与对象（Object）是最基本的概念，字段和方法是一个类的主要构成元素。当实例化对象后，字段便可以理解为一个对象的各种“器官”，而方法则可以被理解为一个对象的一系列“行为”，因此方法就是程序用于执行命令的关键，那么方法和栈帧之间又存在什么样的关系呢？简单来说，栈帧是一种用于支持 JVM 调用/执行程序方法的数据结构，它是方法的执行环境，每一个方法被调用时都会创建一个独立的栈帧以便维系所需的各种数据信息，栈帧伴随着方法的调用而创建，伴随着方法的执行结束而销毁，那么每一个方法从调用到执行结束的过程，就对应着 Java 栈中一个栈帧从入栈到出栈的过程，并且无论方法的调用状态是否正常都算作方法结束。在此大家需要注意，不同线程中所包含的栈帧是不允许存在相互引用的。

在栈帧中，局部变量表和操作数栈所需的容量大小在编译期就可以完全被确定下来，并保存在方法的 Code 属性中，也就是说，栈帧究竟需要分配多大的内存空间完全取决于具体的 JVM 实现和方法调用时分配的实际内存。在一条活动线程中，只有当前正在执行的方法的栈帧（栈顶栈帧）是有效的，这个栈帧也被称之为当前栈帧（Current Frame），与当前栈帧相对应的方法就是当前方法（Current Method），定义这个方法类就是当前类（Current Class）。如图 8-1 所示。



图 8-1 栈帧的组成结构

大家思考一下，既然一个线程中只有当前正在执行的方法的栈帧才是当前栈帧，那么如果当前方法在执行过程中调用了另外一个新的方法时，当前栈帧会发生变化吗？如下所示：

代码 8-1 当前栈帧示例

```

/**
 * 谁是当前栈帧?
 *
 * @author JohnGao
 */
public class CurrentFrameTest {
    public void methodA() {
        System.out.println("当前栈帧对应的方法->methodA");
        methodB();
        System.out.println("当前栈帧对应的方法->methodA");
    }
    public void methodB() {
        System.out.println("当前栈帧对应的方法->methodB");
    }
}

```

在上述程序示例中, 如果与 `methodA()` 方法相对应的栈帧是当前栈帧, 那么当 `methodA()` 方法内部调用了 `methodB()` 方法时, 则会有一个与 `methodB()` 方法相对应的新栈帧作为当前栈帧被创建, 也就是说, 程序的控制权将会移交给 `methodB()` 方法。不过当 `methodB()` 方法执行完成并返回后, 当前栈帧随之被丢弃, 前一个栈帧又重新变为当前栈帧。

8.1.1 局部变量表

局部变量表 (Local Variables Table) 也可以称之为本地变量表, 它包含在一个独立的栈帧中。顾名思义, 局部变量表主要用于存储方法参数和定义在方法体内的局部变量, 这些数据类型包括各类原始数据类型、对象引用 (reference), 以及 `returnAddress` 类型。局部变量表所需的容量大小在编译期就可以被完全确定下来, 并保存在方法的 `Code` 属性中。大家思考一下, 既然方法体内定义的局部变量是存储在栈帧中的局部变量表里的, 那么原始数据类型的成员变量的值是否也存储在局部变量表中呢? 其实如果是定义在方法体外的成员变量, 不止是作用域发生了变化, 更重要的是, 其值也并非还是存储在局部变量表里, 而是存储在对象内存空间的实例数据中, 整体来看即存储在 Java 堆区内。简单来说, 与线程上下文相关的数据存储在 Java 栈中, 反之则存储在 Java 堆区内。

局部变量表可以看做是专门用于存储局部变量值的一种类似于线性表 (Linear List) 的数据结构。参考《Java 虚拟机规范 Java SE7 版》的描述来看, 局部变量表中最小的存储单元是 Slot (变量槽), 一个 Slot 可以存储一个类型为 `boolean`、`byte`、`char`、`short`、`float`、`reference` 以及 `returnAddress` 小于或等于 32bit 的数值, 2 个 Slot 可以存储一个类型为 `long` 或 `double` 的 64bit 数值。JVM 会为局部变量表中的每一个 Slot 都分配一个访问索引, 通过这个索引即可成功访问到局部变量表中指定的局部变量值, 访问索引从 0 开始到小于局部变

量表最大的 Slot 长度，如图 8-2 所示。在此大家需要注意，由于 long 和 double 类型的二进制位数是 64bit，那么当使用这 2 个类型存储数据时，理论上占用的是 2 个连续的 Slot，如果需要访问局部变量表中一个 64bit 的局部变量值时，只需要使用前一个索引即可。这就好比一个 double 类型的值存储在局部变量表中其 Slot 的访问索引为 n ，当我们需要取出这个局部变量值时，只需要根据索引 n 便可以成功取出 n 和 $n+1$ 的值，也就是一个完整的 64bit 的数据值。当然关于是否一定需要使用 2 个连续的 Slot 来存储一个 64bit 的值，Java 虚拟机规范其实并没有明确要求，这主要还需要根据 JVM 的具体实现而定。除此之外，一个 Slot 究竟应该占用多大的内存空间 Java 虚拟机规范同样也没有明确的要求，但最好使用 32bit 的内存空间用于存储 boolean、byte、char、short、float、reference 及 returnAddress 等类型的值，当然这并不会意味着 Slot 的内存大小就一定会固定为 32bit，因为 Slot 的内存大小允许根据处理器、操作系统或 JVM 实现的不同而产生相应的变化。

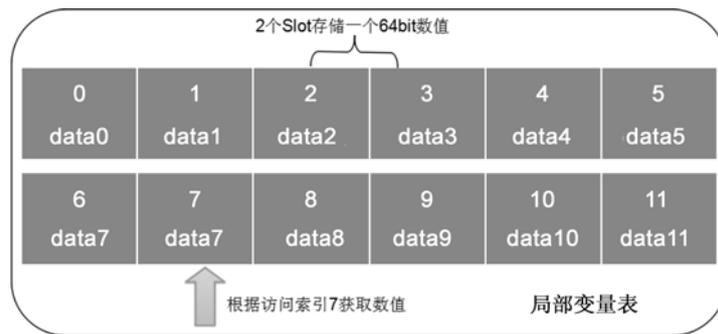


图 8-2 根据索引访问局部变量表中的数据

JVM 使用局部变量表来完成方法调用时的参数传递，当一个实例方法被调用的时候，它的方法参数和方法体内部定义的局部变量将会按照顺序被复制到局部变量表中的每一个 Slot 上。访问索引为 0 的 Slot 一定存储的是与被调用实例方法相对应的对象引用（通过 Java 语法层面的“this”关键字便可访问到这个参数），而后续的其他方法参数和方法体内定义的成员变量则会按照顺序从局部变量表中索引为 1 的 Slot 位置处展开复制。

8.1.2 操作数栈

每一个独立的栈帧中除了包含局部变量表以外，还包含一个后进先出 (Last-In-First-Out) 的操作数栈，也可以称之为表达式栈 (Expression Stack)。操作数栈和局部变量表在访问方式上存在着较大差异，操作数栈并非采用访问索引的方式来进行数据访问的，而是通过标准的入栈和出栈操作来完成一次数据访问。每一个操作数栈都会拥有一个明确的栈深度用于存储数值，一个 32bit 的数值可以用一个单位的栈深度来存储，而 2 个单位的栈深度则可以保存一个 64bit 的数值，当然操作数栈所需的容量大小在编译期就可以被完全确定下来，并保

存在方法的 Code 属性中。

在 HotSpot 中，除了 PC 寄存器之外，再也没有包含其他任何的寄存器，并且之前曾经提及过，HotSpot 中任何的操作都需要经过入栈和出栈来完成，那么由此可见，HotSpot 的执行引擎架构必然就是基于栈式架构，而非传统的寄存器架构。简单来说，操作数栈就是 JVM 执行引擎的一个工作区，当一个方法被调用的时候，一个新的栈帧也会随之被创建出来，但这个时候栈帧中的操作数栈却是空的，只有方法在执行的过程中，才会有各种各样的字节码指令往操作数栈中执行入栈和出栈操作。比如在一个方法内部需要执行一个简单的加法运算时，首先需要从操作数栈中将需要执行运算的两个数值出栈，待运算执行完成后，再将运算结果入栈。如下所示：

代码 8-2 执行加法运算的字节码指令

```
public void testAddOperation();
  Code:
    0: bipush      15
    2: istore_1
    3: bipush      8
    5: istore_2
    6: iload_1
    7: iload_2
    8: iadd
    9: istore_3
   10: return
```

在上述字节码指令示例中，首先会由“bipush”指令将数值 15 从 byte 类型转换为 int 类型后压入操作数栈的栈顶（对于 byte、short 和 char 类型的值在入栈之前，会被转换为 int 类型），当成功入栈之后，“istore_1”指令便会负责将栈顶元素出栈并存储在局部变量表中访问索引为 1 的 Slot 上。接下来再次执行“bipush”指令将数值 8 压入栈顶后，通过“istore_2”指令将栈顶元素出栈并存储在局部变量表中访问索引为 2 的 Slot 上。“iload_1”和“iload_2”指令会负责将局部变量表中访问索引为 1 和 2 的 Slot 上的数值 15 和 8 重新压入操作数栈的栈顶，紧接着“iadd”指令便会将这 2 个数值出栈执行加法运算后再将运算结果重新压入栈顶，“istore_3”指令会将运算结果出栈并存储在局部变量表中访问索引为 3 的 Slot 上。最后“return”指令的作用就是方法执行完成之后的返回操作。在操作数栈中，一项运算通常由多个子运算（subcomputation）嵌套进行，一个子运算过程的结果可以被其他外围运算所使用。

在此大家需要注意，在操作数栈中的数据必须进行正确的操作。比如不能在入栈 2 个 int 类型的数值后，却把它们当做 long 类型的数值去操作，或者入栈 2 个 double 类型的数值后，使用 iadd 指令对它们执行加法运算等情况出现。

8.1.3 动态链接

每一个栈帧内部除了包含局部变量表和操作数栈之外,还包含一个指向运行时常量池中该栈帧所属方法的引用,包含这个引用的目的就是为了支持当前方法的代码能够实现动态链接 (Dynamic Linking)。在 6.2.3 节中,笔者曾经提及过运行时常量池,一个有效的字节码文件中除了包含类的版本信息、字段、方法以及接口等描述信息外,还包含一项信息,那就是常量池表 (Constant Pool Table),那么运行时常量池就是字节码文件中常量池表的运行时表示形式。在一个字节码文件中,描述一个方法调用了另外的其他方法时,就是通过常量池中指向方法的符号引用 (Symbolic Reference) 来表示的,那么动态链接的作用就是为了将这些符号引用转换为调用方法的直接引用。

在 JVM 中,将符号引用转换为调用方法的直接引用与方法的绑定机制相关。当一个字节码文件被装载进 JVM 内部时,如果被调用的目标方法在编译期可知,且运行期保持不变时,那么在这种情况下将调用方法的符号引用转换为直接引用的过程称之为静态链接。相反如果被调用的方法在编译期无法被确定下来,也就是说,只能够在程序运行期将调用方法的符号引用转换为直接引用,由于这种引用转换过程具备动态性,因此也就被称之为动态链接。既然引用的转换方式与方法的绑定机制相关,那么究竟什么是方法绑定呢?在 Java 中一共拥有两种方法绑定方式,分别是早期绑定 (Early Binding) 和晚期绑定 (Late Binding)。顾名思义,早期绑定就是指被调用的目标方法如果在编译器可知,且运行期保持不变时,即可将这个方法与所属的类型进行绑定,这样一来,由于明确了被调用的目标方法究竟是哪一个,因此也就可以使用静态链接的方式将符号引用转换为直接引用。相反如果被调用的方法在编译期无法被确定下来,只能够在程序运行期根据实际的类型绑定相关的方法,这种绑定方式也就被称之为晚期绑定。在早期基于过程式的编程语言中,由于本身并不具备面向对象的一些特性,自然也就无法在语法层面上使用多态 (Polymorphism) 特性来实现方法的重写 (Override) 操作,也就是说,基于过程式的编程语言从严格意义上来说仅仅只存在一种绑定方式,那就是早期绑定。但随着高级语言的横空出世,类似于 Java 一样的基于面向对象的编程语言如今越来越多,尽管这类编程语言在语法风格上存在一定的差别,但是它们彼此之间始终保持着—个共性,那就是都支持封装、继承和多态等面向对象特性,既然这一类的编程语言具备多态特性,那么自然也就具备早期绑定和晚期绑定两种绑定方式。

在 Java 中,开发人员并不需要在程序中显式指定某一个方法需要在运行期支持晚期绑定,因为除了 final 方法外,几乎所有的方法都是默认基于晚期绑定的。如下所示:

代码 8-3 晚期绑定示例

```
/**
 * 晚期绑定(Late Binding)示例
```

```
*
* @author JohnGao
*/
public class LateBinding {
    public static void getName(Animal animal) {
        animal.name();
    }

    public static void main(String[] args) {
        getName(new Tiger());
        getName(new Pig());
    }
}

interface Animal {
    public void name();
}

class Tiger implements Animal {
    @Override
    public void name() {
        System.out.println("我是 Tiger, 我派生于 Animal");
    }
}

class Pig implements Animal {
    @Override
    public void name() {
        System.out.println("我是 Pig, 我派生于 Animal");
    }
}
```

在上述程序示例中，接口 `Animal` 包含 `Tiger` 和 `Pig` 两个派生类，并且这两个派生类还重写了它的 `name()` 方法。由于在编译期并不明确 `LateBinding` 类中的 `getName()` 方法究竟需要调用哪一个 `name()` 方法，也就无法使用静态链接的方式将符号引用转换为直接引用，因此这一类的方法就是基于晚期绑定的虚函数，其实虚函数的存在就是为了支持多态特性。与动态链接相反，如果程序中能够使用静态链接的方式将符号引用转换为直接引用的话，这一类的方法就是基于早期绑定的非虚函数。在此大家需要注意，从严格意义上来说，在 `Java` 中其实并不存在虚函数的概念，因为开发人员并不需要显式使用任何关键字去标示 `Java` 中的一个虚函数。简单来说，`Java` 中任何一个普通的方法其实都具备虚函数的特征，它们相当于 `C++` 语言中的虚函数（`C++` 中则需要使用关键字 `virtual` 来显式定义）。如果在程序中不希望某个方法拥有虚函数的特征时，则可以使用关键字 `final` 来标记这个方法。

8.1.4 方法返回值

一个方法在执行的过程中将会产生两种调用结果：一种是方法正常调用完成，而另外一种则是方法异常调用完成。如果是方法正常调用完成，那么这就意味着，被调用的当前方法在执行的过程中将不会有任何的异常被抛出，并且方法在执行的过程中一旦遇见字节码返回指令时，将会把方法的返回值返回给它的调用者，不过一个方法在正常调用完成之后究竟需要使用哪一个返回指令还需要根据方法返回值的实际数据类型而定。在字节码指令中，返回指令包含 `ireturn`（当返回值是 `boolean`、`byte`、`char`、`short` 和 `int` 类型时使用）、`lreturn`、`freturn`、`dreturn` 以及 `areturn`，另外还有一个 `return` 指令供声明为 `void` 的方法、实例初始化方法、类和接口的初始化方法使用。

与方法正常调用完成相反的就是方法异常调用完成。方法异常调用完成意味着当前方法在执行的过程中可能会因为某些错误的指令导致 JVM 抛出了异常，并且这些异常在当前方法中没有办法进行处理，或者方法在执行的过程中遇见了 `athrow` 指令显式抛出的异常，并且在当前方法内部没有捕获这个异常。总之，如果一个方法在执行的过程中抛出了异常，那么这个方法在调用完成之后将不会再有任何的返回值返回给它的调用者。

无论当前方法的调用结果是正常还是异常，都需要在执行完成之后返回到之前被调用的位置上，那么这个时候当前栈帧就承担着恢复调用者状态的责任。之前曾经提及过，在方法内部调用了另外一个方法时，将会有有一个与当前方法相对应的新栈帧被创建出来，当方法调用完成之后，当前栈帧随之被丢弃，前一个栈帧又重新变为了当前栈帧，而被调用的方法如果带有返回值的话，其返回值将会被压入当前栈帧的操作数栈中，并更新 PC 寄存器中下一条需要执行的字节码指令。

8.2 HotSpot 中执行引擎的架构模型

时至今日，似乎可以毫不夸张地说，Java 虚拟机实际上是一个比 Java 语言本身更成功、更优秀甚至更伟大的产品。笔者在 1.4.2 节中曾经提及过与语言无关特性，像 JRuby、Rhino、PHP（IBM WebSphere sMash PHP(P8)、Caucho Quercus）和 Scala 之类的编程语言本身其实并不依赖于 Java 语言，而是依赖于 Java 虚拟机作为程序的宿主运行环境。那么由此可见，Java 技术的核心就是 Java 虚拟机（JVM，Java Virtual Machine），因为所有的 Java 程序都运行在 Java 虚拟机内部。Java 虚拟机之所以被称之为虚拟机，是因为它是由规范所定义出的一种抽象计算机，它的主要任务就是负责装载字节码到其内部，解释/编译为对应平台上的机器指令执行，而执行引擎就是负责执行这项任务的关键，它是整个 Java 虚拟机中最重要的组成结构之一。

当然 Java 虚拟机规范并没有明确要求执行引擎究竟应该采用解释器还是 JIT 编译器的实现方式来执行字节码指令，不过早在 Java1.0 版本的时候 Sun 公司提供了一款基于纯解释器实现的 Java 虚拟机 (Sun Classic VM)，但由于解释器的执行效率非常低下，所以到了 1998 年 Sun 公司发布 Java1.2 版本的时候，基于即时编译的 JIT 编译器出现了，带给了 Java 虚拟机在运行效率上质的飞跃。如今的 HotSpot VM 内部采用的是解释器与 JIT 编译器并存方案共同执行字节码指令，当虚拟机启动的时候，解释器首先发挥作用，而不必等待编译器全部编译完成再执行，这样可以省去许多不必要的编译时间。并且随着程序运行时间的推移，JIT 编译器将会逐渐发挥它的作用，根据热点探测功能，将有价值的字节码指令直接编译为本地机器指令，以便换取更高的程序执行效率。

既然谈到了执行引擎，那就不得不提到 Java 虚拟机的架构模型。简单来说，基于寄存器架构的虚拟机，在性能上会占有优势，像 Google 公司专门为 Android 平台研发的 Dalvik 虚拟机就是基于寄存器架构的。相反如果是基于栈式架构的虚拟机在设计和实现上相对来说会更加简单，而且更适用于一些资源受限的系统（关于寄存器架构和栈式架构虚拟机的更多区别，请阅读 8.2.2 节）。大家回想一下，Java 语言的设计初衷是否就是为了能够在嵌入式设备中运行呢？由于嵌入式设备大都是一些资源受限的系统，因此当时 James Gosling 博士等人为 HotSpot VM 选择了基于栈的架构模型。大家思考一下，尽管嵌入式平台如今早就不是 Java 程序的主流运行平台了（准确来说应该是 HotSpot VM 的宿主环境已经不局限于嵌入式平台了），那么为什么不将架构更换为基于寄存器的架构呢？之前也说过了，由于栈式架构在设计和实现上会更加简单，所以在非资源受限的系统下选择栈式架构也未尝不可。从 Java 语法层面上来说，方法定义了对象的一系列“行为”，所以执行引擎本质上就是通过调用一个个的方法来执行命令。每当调用一个新方法的时候，一个与当前方法相对应的当前栈帧也就随之创建出来，栈帧伴随着方法的调用而创建，伴随着方法的执行结束而销毁，那么每一个方法从调用到执行结束的过程，就对应着 Java 栈中一个栈帧从入栈到出栈的过程。

8.2.1 本地机器指令

在计算机世界中，只有机器指令能够在其内部执行，那么在正式开始讲解 HotSpot 中的执行引擎究竟如何执行字节码指令之前，我们首先来了解下什么是机器指令。不过大家不必担心，笔者并不打算列举一些枯燥乏味的操作系统原理等相关知识来阐述本节内容。

作为一门优秀的面向对象编程语言，Java 编译结果屏蔽了与底层操作系统和物理硬件相关的一些特性，使得开发人员尽可能地只需关注于自身业务。当然字节码仅仅只是一个实现跨平台的通用契约而已，它并不能够直接运行在操作系统之上，因为字节码指令并非等价于本地机器指令，它内部包含的仅仅只是一些能够被 JVM 所识别的字节码指令、符号表，以及其他辅助信息。那么如果想要让一个 Java 程序运行起来，执行引擎就需要将字节码指

令解释/编译为对应平台上的本地机器指令才可以，简单来说，JVM 中的执行引擎充当了将高级语言翻译为机器语言的译者。

那么究竟什么是机器指令呢？机器指令其实就是一种能够被 CPU 直接识别并执行的指令，它以二进制编码作为表示形式。机器指令通常由操作码和操作数两部分构成，操作码决定了指令需要执行什么样的功能，而操作数则指定了需要参与运算的操作数，以及从哪里获取操作数、将运算结果存储在哪个位置（寄存器还是栈中）等。由于机器指令与 CPU 紧密相关，所以不同种类的 CPU 所对应的机器指令也就不同，并且它们的指令系统往往差别很大。在此大家需要注意，由于在实际的开发过程中，大部分开发人员都是使用的基于面向对象的高级语言（比如 Java、Python、JRuby 等），并且从高级语言翻译为机器语言有专门的编译器会负责完成，开发人员并不需要在意由于不同的物理架构所带来的差别。

8.2.2 寄存器架构与栈式架构之间的区别

在 8.2 节中，笔者只是简单地介绍了关于寄存器架构和栈式架构之间的区别。当然仅凭栈式架构在设计和实现上更加简单这一个理由还不足以让 JVM 的设计者们动心，那么笔者接下来将会从方方面面来阐述这两种架构之间的区别，让大家更加深刻地理解基于栈式架构所带来的好处和栈式架构的优点。

指令集不同

其实寄存器架构和栈式架构之间最本质的区别还是二者之间使用的指令集不同，在 8.2.1 节中笔者曾为大家讲解过什么是本地机器指令，指令通常由操作码和操作数两部分构成，操作码决定了指令需要执行什么样的功能，而操作数则指定了需要参与运算的操作数，以及从哪里获取操作数、将运算结果存储在哪个位置（寄存器还是栈中）等。那么根据指令操作方式的不同，我们便可以将指令划分为零地址指令、一地址指令、二地址指令和三地址指令等 n 地址指令，在此大家需要注意，由于 n 是一个自然数，因此也就意味着指令集可以是任意的 n 地址。在大部分情况下，基于寄存器架构的指令集往往都以一地址指令、二地址指令和三地址指令为主，而基于栈式架构的指令集却是以零地址指令为主。

那么不同的地址指令之间到底存在什么区别呢？以三地址指令为例，在一个简单的二元运算操作中，三地址指令正好可以指定 2 个数据源和 1 个存储目标，这样必然能够非常灵活地将二元运算和赋值操作组合在一起。如下所示：

代码 8-4 三地址指令的表示形式

```
op dest, src1, src2
```

很明显，如果是使用三地址指令去执行一项二元运算操作的确非常灵活。相反如果使用

基于栈式架构的零地址指令去执行一项二元运算时，又会如何呢？就以代码 8-2 为例，其中的“iadd”指令并没有任何参数，甚至连数据源都没有办法进行指定，那么零指令地址究竟有什么用呢？其实零地址指令意味着数据源和存储目标都是隐含参数，其实现就是依赖于一种被称之为栈的数据结构。首先会由“bipush”指令将数值 15 从 byte 类型转换为 int 类型后压入操作数栈的栈顶（对于 byte、short 和 char 类型的值在入栈之前，会被转换为 int 类型），当成功入栈之后，“istore_1”指令便会负责将栈顶元素出栈并存储在局部变量表中访问索引为 1 的 Slot 上。接下来再次执行“bipush”指令将数值 8 压入栈顶后，通过“istore_2”指令将栈顶元素出栈并存储在局部变量表中访问索引为 2 的 Slot 上。“iload_1”和“iload_2”指令会负责将局部变量表中访问索引为 1 和 2 的 Slot 上的数值 15 和 8 重新压入操作数栈的栈顶，紧接着“iadd”指令便会将这 2 个数值出栈执行加法运算后再将运算结果重新压入栈顶，“istore_3”指令会将运算结果出栈并存储在局部变量表中访问索引为 3 的 Slot 上。

既然大家都已经知道字节码文件中的指令集设计就是基于零地址指令的，那么使用零地址指令会有什么好处呢？其实零地址指令相比其他形式的指令会显得更加紧凑，因为在一个字节码文件中，除了处理 2 个表跳转的指令外，其他都是按照 8 位字节进行对齐的，操作码可以只占一个字节大小，这就意味着将会有更多的空间用于存储其他指令。因此在空间紧缺的环境中，使用零地址指令的设计将会是不错的选择。当然有利就必然会有弊，零地址指令尽管拥有良好的紧凑性，但是它完成一个操作却往往需要比二地址指令或三地址指令花费更多的出栈和入栈指令。比如在 x86 平台中的 CPU 指令集就是基于二地址指令的，如果完成一项类似的运算操作，二地址指令只需花费 2 条指令即可。

基于栈式架构的优点

- 设计和实现更简单，适用于资源受限的系统；
- 避开了寄存器的分配难题；
- 指令集更加紧凑。

如果虚拟机选用了基于栈的架构，不仅在架构的设计和实现上会更加简单，而且更适用于资源受限的系统。所谓资源受限，通常情况下大都是指一些 CPU 运算效率低下、内存较小的嵌入式设备（比如机顶盒、打印机等）。如果大家细心的话，在阅读本章的前面几个小节中的内容后应该会对基于栈式架构的指令执行方式非常熟悉了。简单来说，每当调用一个新方法的时候，一个与当前方法相对应的当前栈帧也就随之被创建出来，栈帧伴随着方法的调用而创建，伴随着方法的执行结束而销毁，那么每一个方法从调用到执行结束的过程，就对应着 Java 栈中一个栈帧从入栈到出栈的过程。并且如果是方法内部需要执行运算时，无非就是对操作数栈中的栈顶元素频繁地执行入栈和出栈操作而已。

由于基于栈式架构的零地址指令的执行方式仅仅只是对栈顶元素操作，所以在设计上根

本就不需要考虑寄存器的分配问题，因此大幅度简化了虚拟机在架构设计上的复杂度。我们都知道字节码是 Java 程序实现跨平台运行的基石，但最终字节码指令仍然需要被装载进 JVM 内部由执行引擎负责将其解释/编译为对应平台上的机器指令执行，因此基于寄存器架构的 JVM 自然会丧失掉 Java 程序与生俱来的跨平台优势，这是因为在一些寄存器较少或是寄存器不规律的平台中（典型的 CISC 处理器的通用寄存器数量很少，例如 32 位的 x86 就只有 8 个 32 位通用寄存器（如果不算 EBP 和 ESP 那就是 6 个，现在一般都算上）；典型的 RISC 处理器的各种寄存器数量多一些，例如 ARM 有 16 个 32 位通用寄存器，Sun 的 SPARC 在一个寄存器窗口里则有 24 个通用寄存器（8 in, 8 local, 8 out）），仍然需要保证 Java 程序能够正常顺利地运行下去，这几乎是不现实的。在此大家需要注意，尽管 Google 公司研发的 Dalvik 是一款根据 ARM 平台而设计的基于寄存器架构实现的虚拟机，但是从严格意义上来说，Dalvik 却并没有完全按照 Java 虚拟机规范来进行设计和实现，并且它运行的也不是传统意义上的字节码文件，而是 Android 平台专属的 dex 文件。

除了设计和实现更加简单，以及避开了寄存器的分配难题等优点外，基于栈式架构的虚拟机还有一个优点之前也已经提到过了，那就是前端编译器所生成的字节码指令相对来说更加紧凑。这是因为在基于零地址指令的字节码文件中，除了处理 2 个表跳转的指令外，其他都是按照 8 位字节进行对齐的，操作码可以只占一个字节大小，这就意味着将会有更多的空间用于存储其他指令。而基于寄存器架构的 Dalvik 虚拟机所执行的字节码指令内部却是采用 16 位双字节的方式进行设计的。

基于寄存器架构的优点

- 性能优秀和执行高效；
- 花费更少的指令去完成一项操作。

综合来看，基于栈式架构的零地址指令设计更适用于通用的虚拟机（比如 HotSpot），但实际上基于寄存器架构的虚拟机性能却显得更加高效。引用 RednaxelaFX 在博文《虚拟机随谈一》^①中的一段描述，尽管基于寄存器架构的虚拟机所使用的零地址指令更加紧凑，但是完成一项操作的时候必然需要花费更多的入栈和出栈指令，这同时也意味着将需要更多的指令分派（instruction dispatch）次数和内存读/写次数。由于访问内存是执行速度的一个重要瓶颈，二地址指令或三地址指令虽然每条指令占的空间较多，但总体来说可以用更少的指令去完成一项操作，指令分派与内存读/写次数相对来说也都更少。

① 博文地址：<http://rednaxelafx.iteye.com/blog/492667>。

8.2.3 基于栈式架构的设计

当大家都已经知道 HotSpot 虚拟机基于栈式架构后，笔者接下来为大家汇总叙述一下执行引擎在运行的过程中究竟是如何与其相关的一些组件协同运作的，当大家理解本节所阐述的内容后，相信会对执行引擎的架构设计有更清楚的认识和了解。

如图 8-3 所示，在 JVM 中，PC 寄存器和 Java 栈都同属线程私有的运行时内存区，并且它们的生命周期与线程的生命周期也是保持一致的。每当启动一个新线程的时候，JVM 就会为当前线程分配一个独立的 PC 寄存器和 Java 栈空间。那么当线程调用一个方法的时候，一个与当前方法相对应的当前栈帧也就随之被创建出来，并存储在当前线程的独立栈空间中，栈帧伴随着方法的调用而创建，伴随着方法的执行结束而销毁，那么每一个方法从调用到执行结束的过程，就对应着 Java 栈中一个栈帧从入栈到出栈的过程。栈帧主要由 4 部分构成，分别是局部变量表、操作数栈、动态链接和方法返回值，其中局部变量表主要用于存储方法参数和定义在方法体内部的局部变量，这些数据类型包括各类原始数据类型、对象引用（reference）以及 returnAddress 类型；而操作数栈则作为执行引擎的一个运行时工作区，用于存储方法在执行过程中的各种临时数据；当前方法在执行的过程中不排除会调用其它方法，因此栈帧中动态链接的任务就是负责将调用方法的符号引用转换为直接引用；一旦方法在执行的过程中遇见字节码返回指令时，便会将方法的返回值返回给它的调用者。

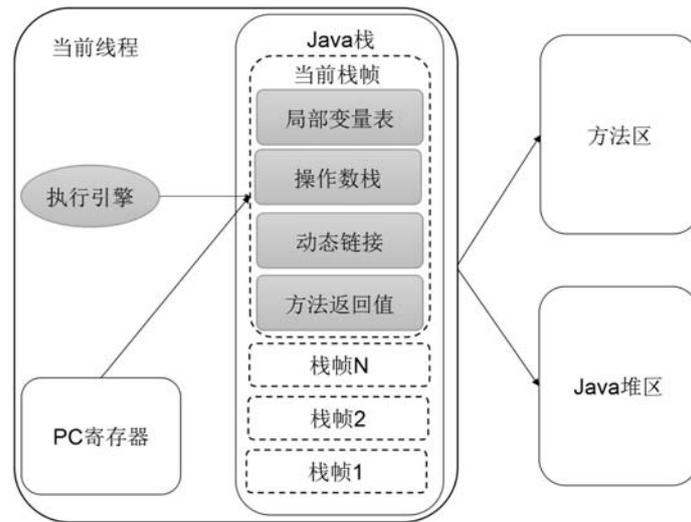


图 8-3 执行引擎的架构设计

在此大家需要注意，执行引擎在运行的过程中究竟需要执行什么样的字节码指令完全依

赖于 PC 寄存器，每当执行完一项指令操作后，PC 寄存器就会更新下一条需要被执行的指令地址。当然方法在执行的过程中，执行引擎有可能会通过存储在局部变量表中的对象引用准确定位到存储在 Java 堆区中的对象实例信息，以及通过对象头中的元数据指针定位到目标对象的类型信息。

8.2.4 调用 call_stub()函数执行 Java 方法

在 2.3.7 节中，笔者曾经讲解过 Java 的启动函数 main()是如何被执行的。首先会由 Launcher 调用 CallStaticVoidMethod()函数，CallStaticVoidMethod()函数属于结构体 JNIEnv 类型的函数指针，该类型对应的目标函数为 jni_CallStaticVoidMethod()。在 jni_CallStaticVoidMethod()函数内部又调用了包含在 /hotspot/src/share/vm/prims/jni.cpp 中的 jni_invoke_static() 函数，而在 jni_invoke_static() 函数内部，JavaCalls 首先会将 GetStaticMethodId()函数中获取出一个全局唯一的方法 ID，然后将其转换为方法句柄后，再通过 call()函数最终执行 Java 程序的 main()方法。当然除了可以执行 Java 程序的 main()方法外，程序中所有的 Java 方法执行都需要依赖 JavaCalls 模块来完成调用，并由 JavaCalls 模块负责创建与当前方法相对应的当前栈帧。

在 JavaCalls 模块的实现中包含一个 call_helper()函数，该函数的作用就是调用 CallStub()函数执行 Java 方法，并返回调用方法的返回值给其调用者。在此大家需要注意，CallStub()函数只是一个函数指针，其对应的目标函数为 StubRoutines 模块中的 call_stub()函数。call_stub()函数的源代码包含在 /hotspot/src/share/vm/runtime/javaCalls.cpp 中。

为了使大家阅读更方便，本书示例了 call_helper()函数的完整代码，如下所示：

代码 8-5 call_helper()函数的完整代码

```

void JavaCalls::call_helper(JavaValue* result, methodHandle* m,
JavaCallArguments* args, TRAPS) {
    methodHandle method = *m;
    JavaThread* thread = (JavaThread*)THREAD;
    assert(thread->is_Java_thread(), "must be called by a java thread");
    assert(method.not_null(), "must have a method to call");
    assert(!SafepointSynchronize::is_at_safepoint(), "call to Java code
during VM operation");
    assert(!thread->handle_area()->no_handle_mark_active(), "cannot
call out to Java here");

    CHECK_UNHANDLED_OOPS_ONLY(thread->clear_unhandled_oops());

    // Verify the arguments

```

```
    if (CheckJNICalls) {
        args->verify(method, result->get_type(), thread);
    }
    else debug_only(args->verify(method, result->get_type(), thread));

    // Ignore call if method is empty
    if (method->is_empty_method()) {
        assert(result->get_type() == T_VOID, "an empty method must return
a void value");
        return;
    }

    #ifndef ASSERT
    { klassOop holder = method->method_holder();
      // A klass might not be initialized since JavaCall's might be used
      during the executing of
      // the <clinit>. For example, a Thread.start might start executing
      on an object that is
      // not fully initialized! (bad Java programming style)
      assert(instanceKlass::cast(holder)->is_linked(), "rewritting must
have taken place");
    }
    #endif

    assert(!thread->is_Compiler_thread(), "cannot compile from the
compiler");
    if (CompilationPolicy::must_be_compiled(method)) {
        CompileBroker::compile_method(method, InvocationEntryBci,
CompilationPolicy::policy()->initial_compile_level(),
        methodHandle(), 0, "must_be_compiled", CHECK);
    }

    // Since the call stub sets up like the interpreter we call the
    from_interpreted_entry
    // so we can go compiled via a i2c. Otherwise initial entry method
    will always
    // run interpreted.
    address entry_point = method->from_interpreted_entry();
    if (JvmtiExport::can_post_interpreter_events()
        && thread->is_interp_only_mode()) {
        entry_point = method->interpreter_entry();
    }

    // Figure out if the result value is an oop or not (Note: This is a
    different value
    // than result_type. result_type will be T_INT of oops. (it is about
    size)
```

```

    BasicType result_type = runtime_type_from(result);

    bool oop_result_flag = (result->get_type() == T_OBJECT ||
result->get_type() == T_ARRAY);

    // NOTE: if we move the computation of the result_val_address inside
    // the call to call_stub, the optimizer produces wrong code.
    intptr_t* result_val_address =
(intptr_t*)(result->get_value_addr());

    // Find receiver
    Handle receiver = (!method->is_static()) ? args->receiver() :
Handle();

    // When we reenter Java, we need to reenale the yellow zone which
    // might already be disabled when we are in VM.
    if (thread->stack_yellow_zone_disabled()) {
        thread->reguard_stack();
    }

    // Check that there are shadow pages available before changing thread
state
    // to Java
    if (!os::stack_shadow_pages_available(THREAD, method)) {
        // Throw stack overflow exception with preinitialized exception.
        Exceptions::throw_stack_overflow_exception(THREAD, __FILE__,
__LINE__, method);
        return;
    } else {
        // Touch pages checked if the OS needs them to be touched to be mapped.
        os::bang_stack_shadow_pages();
    }

    // do call
    { JavaCallWrapper link(method, receiver, result, CHECK);
      { HandleMark hm(thread); // HandleMark used by HandleMarkCleaner
        StubRoutines::call_stub(
            (address)&link,
            // (intptr_t*)&(result->_value), // see NOTE above (compiler problem)
            result_val_address, // see NOTE above (compiler problem)
            result_type,
            method(),
            entry_point,
            args->parameters(),
            args->size_of_parameters(),
            CHECK
        );

        result = link.result(); // circumvent MS C++ 5.0 compiler bug

```

```
(result is clobbered across call)

    // Preserve oop return value across possible gc points
    if (oop_result_flag) {
        thread->set_vm_result((oop) result->get_jobject());
    }
} // Exit JavaCallWrapper (can block - potential return oop must be
preserved)

    // Check if a thread stop or suspend should be executed
    // The following assert was not realistic. Thread.stop can set that
    bit at any moment.
    //assert(!thread->has_special_runtime_exit_condition(), "no async.
    exceptions should be installed");

    // Restore possible oop return
    if (oop_result_flag) {
        result->set_jobject((jobject)thread->vm_result());
        thread->set_vm_result(NULL);
    }
}
```

8.2.5 栈顶缓存 (Top-of-Stack Cashing) 技术

尽管我们都已经知道 HotSpot 的执行引擎采用的并非是基于寄存器的架构，但这并不代表 HotSpot VM 的实现并没有间接利用到寄存器资源。寄存器是物理 CPU 中的组成部分之一，它同时也是 CPU 中非常重要的高速存储资源。一般来说，寄存器的读/写速度非常迅速，甚至可以比内存的读/写速度快上几十倍不止，不过寄存器资源却非常有限，不同平台下的 CPU 寄存器数量是不同和不规律的。寄存器主要用于缓存本地机器指令、数值和下一条需要被执行的指令地址等数据。

笔者在 8.2.2 节中曾详细分析过寄存器架构和栈式架构之间的区别，尽管基于栈式架构的虚拟机所使用的零地址指令更加紧凑，但完成一项操作的时候必然需要使用更多的入栈和出栈指令，这同时也意味着将需要更多的指令分派 (instruction dispatch) 次数和内存读/写次数。由于操作数是存储在内存中的，因此频繁地执行内存读/写操作必然会影响执行速度。为了解决这个问题，JVM 的设计者们提出了栈顶缓存 (ToS, Top-of-Stack Cashing) 技术，将栈顶元素全部缓存在物理 CPU 的寄存器中，以此降低对内存的读/写次数，提升执行引擎的执行效率。

在 HotSpot VM 中，TosState 用于描述指令执行前后 Top-of-Stack 的状态，Top-of-Stack 可以被缓存在一个或多个寄存器中。缓存值 TosState 对应数据的机器级描述。HotSpot VM

一共包含 9 种 TosState，如下所示：

代码 8-6 TosState

```

// TosState describes the top-of-stack state before and after the
// execution of a bytecode or method. The top-of-stack value may be cached
// in one or more CPU registers. The TosState corresponds to the 'machine
// representation' of this cached value. There's 4 states corresponding
// to the JAVA types int, long, float & double as well as a 5th state
// in case the top-of-stack value is actually on the top of stack
// (in memory) and thus not cached. The atos state corresponds to the
// itos state when it comes to machine representation but is used
// separately for (oop) type specific operations
// (e.g. verification code).
enum TosState {          // describes the tos cache contents
    btos = 0,           // 栈顶缓存 byte/bool 类型数据
    ctos = 1,           // 栈顶缓存 char 类型数据
    stos = 2,           // 栈顶缓存 short 类型数据
    itos = 3,           // 栈顶缓存 int 类型数据
    ltos = 4,           // 栈顶缓存 long 类型数据
    ftos = 5,           // 栈顶缓存 float 类型数据
    dtos = 6,           // 栈顶缓存 double 类型数据
    atos = 7,           // 栈顶缓存 object 类型数据
    vtos = 8,           // 栈顶缓存 tos 类型数据
    number_of_states,
    illegl               // illegal state: should not occur
};

```

从虚拟机的概念模型上来说，当执行一项 int 类型的加法运算时，操作数栈中必然会伴随着频繁的入栈和出栈操作。其中“iadd”指令用于将操作数栈中的 2 个 int 类型的栈顶元素出栈，执行运算后再将 int 类型的运算结果重新压入栈顶。在此大家需要注意，在操作数栈中的数据必须进行正确的操作，比如不能够在入栈 2 个 int 类型的数值后，却把它们当做 long 类型的数值去操作，或者入栈 2 个 double 类型的数值后，使用“iadd”指令对它们执行加法运算等情况出现。之前也说过，栈顶缓存技术就是用于降低对内存的读/写次数，提升执行引擎的执行效率，那么在 HotSpot VM 的源码实现中，栈顶元素究竟是如何执行存/取操作的呢？本书示例了包含在/openjdk/hotspot/src/cpu/x86/vm/templateTable_x86_32.cpp 中的 iop2 指令模板，如下所示：

代码 8-7 iop2 指令模板

```

void TemplateTable::iop2(Operation op) {
    transition(itos, itos);
    switch (op) {
        case add :      __ pop_i(rdx); __ addl (rax, rdx); break;
        case sub :      __ mov(rdx, rax); __ pop_i(rax); __ subl (rax, rdx); break;

```

```

case mul :          __ pop_i(rdx); __ imull(rax, rdx); break;

case _and :        __ pop_i(rdx); __ andl (rax, rdx); break;
case _or  :        __ pop_i(rdx); __ orl  (rax, rdx); break;
case _xor :        __ pop_i(rdx); __ xorl (rax, rdx); break;
case shl  : __ mov(rcx, rax); __ pop_i(rax); __ shll (rax);      break;
case shr  : __ mov(rcx, rax); __ pop_i(rax); __ sarl (rax);      break;
case ushr : __ mov(rcx, rax); __ pop_i(rax); __ shrl (rax);      break;
default  : ShouldNotReachHere();
}
}

```

在上述代码示例中，`transition()`函数的作用就是校验执行前后的 `TosState` 是否符合要求。也就是说，在执行“`iadd`”指令之前，`TosState` 必须确保是 `itos`，而执行运算之后的运算结果也同样必须是 `itos`。当通过校验后，如果匹配是加法运算指令时，便会将 `rdx` 寄存器和 `rax` 寄存器中的缓存值相加，最后将运算结果缓存在 `rax` 寄存器中后返回。由于栈顶元素并非缓存在内存中，因此也就无需对内存频繁地执行读/写操作了。

8.2.6 实战：跟踪字节码解释器的执行步骤

在 8.1.2 节中，笔者曾详细地描述过执行引擎是如何根据字节码指令执行的，当然这仅仅只是一种概念模型，不同 JVM 的实现细节不尽相同。比如在 HotSpot 虚拟机中，尽管操作数栈式作为执行引擎的一个运行时工作区，临时数据虽然缓存在操作数栈中，但为了性能考虑，栈顶元素均全部缓存在物理 CPU 的寄存器中。因此概念模型所描述的执行引擎的执行步骤与实际的执行步骤之间多少是存在一定的差别，当然有一点是可以肯定的，就是无论采用任何优化技术，从本质上来说，HotSpot 虚拟机自始至终都是基于栈式架构的。所以为了让大家更好地理解执行引擎的执行步骤，笔者将会通过本章的实战小节把代码 8-2 以图文并茂的形式为大家叙述基于概念模型的执行引擎的执行步骤。如图 8-4 至图 8-9 所示。

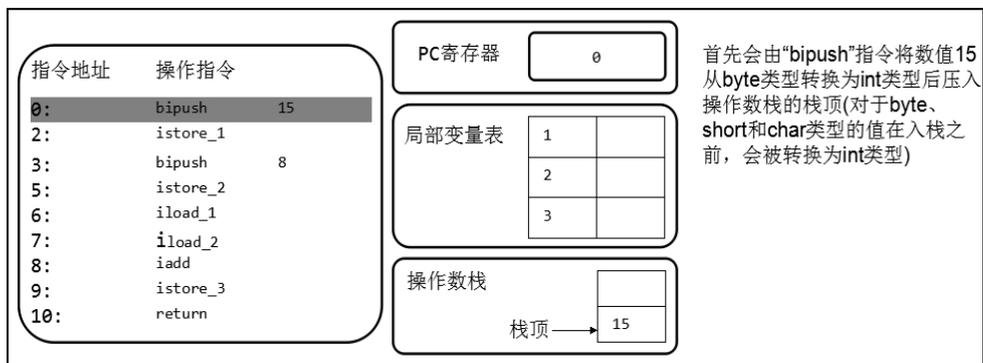


图 8-4 执行指令地址为 0 的操作指令

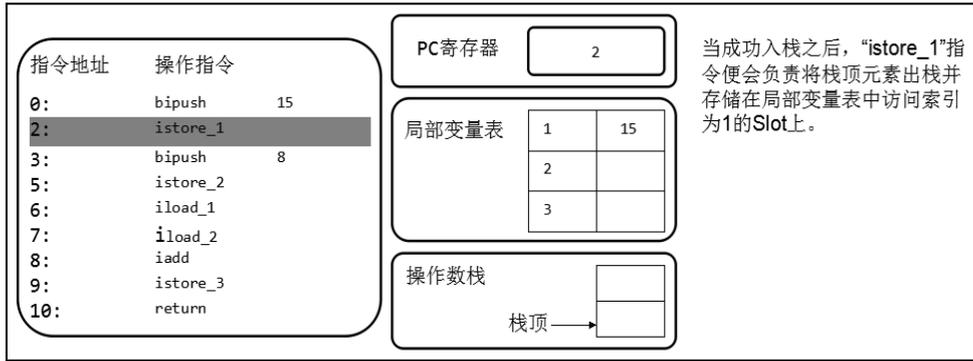


图 8-5 执行指令地址为 2 的操作指令

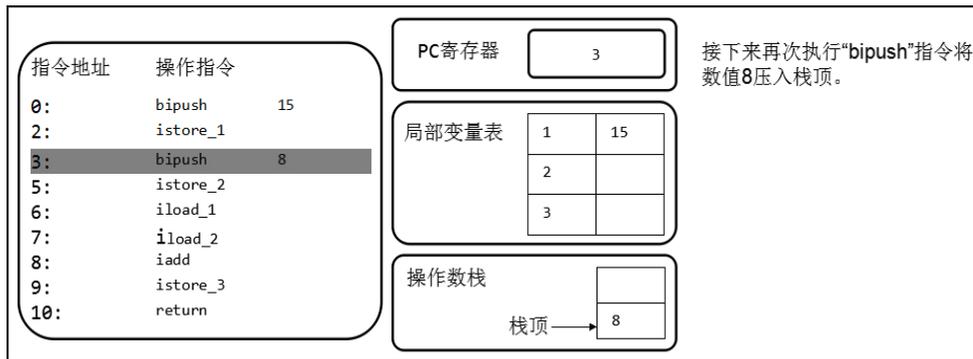


图 8-6 执行指令地址为 3 的操作指令

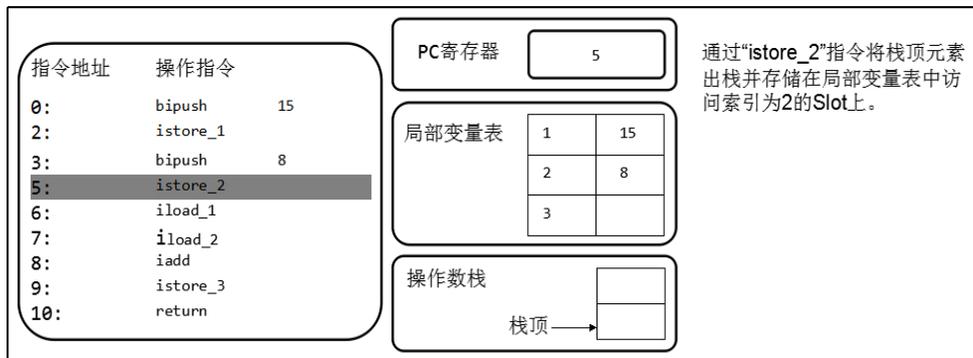


图 8-7 执行指令地址为 5 的操作指令

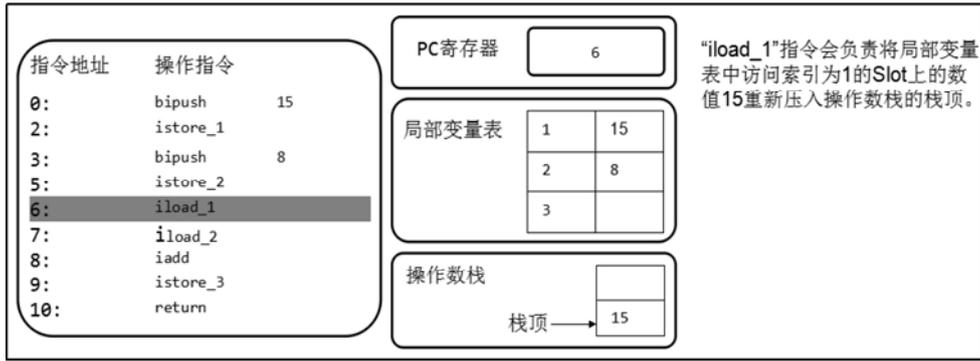


图 8-8 执行指令地址为 6 的操作指令

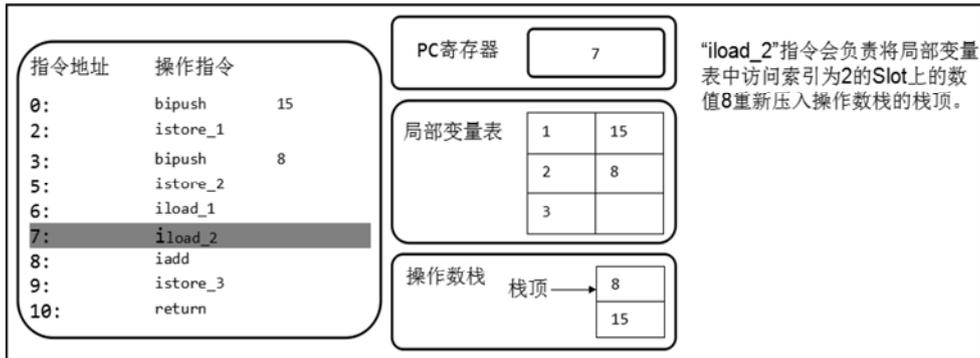


图 8-9 执行指令地址为 7 的操作指令

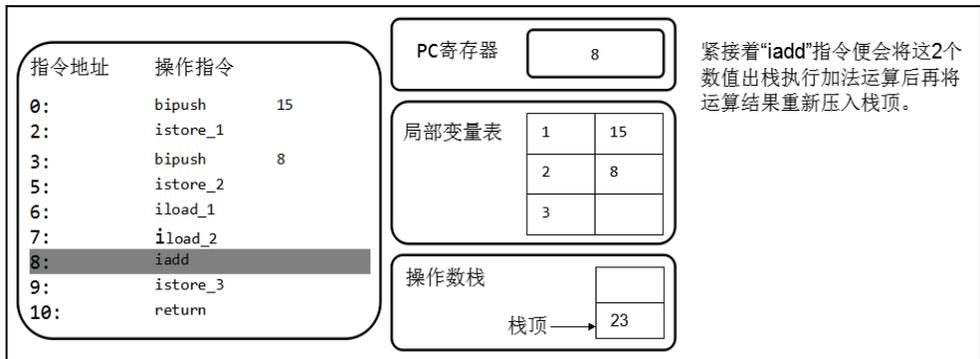


图 8-10 执行指令地址为 8 的操作指令

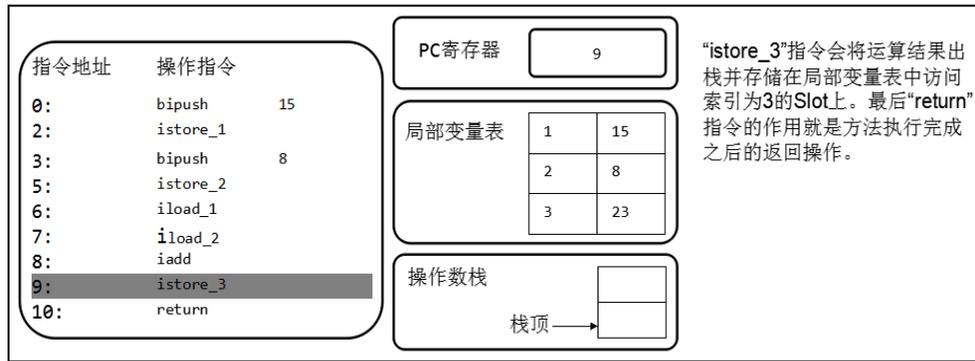


图 8-11 执行指令地址为 9 的操作指令

8.3 解释器与 JIT 编译器

早在 Java1.0 版本的时候，Sun 公司发布了一款名为 Sun Classic VM 的 Java 虚拟机，它同时也是世界上第一款商用 Java 虚拟机，在当时这款虚拟机内部只提供解释器，用今天的眼光来看待必然是效率低下的，因为如果 Java 虚拟机只能够在运行时对代码采用逐行解释执行，程序的运行性能可想而知。但是如今的 HotSpot VM 中不仅内置有解释器，还内置有先进的 JIT（Just In Time Compiler）编译器，在 Java 虚拟机运行时，解释器和即时编译器能够相互协作，各自取长补短。在此大家需要注意，无论是采用解释器进行解释执行，还是采用即时编译器进行编译执行，最终字节码都需要被转换为对应平台的本地机器指令。或许有些开发人员会感觉到诧异，既然 HotSpot VM 中已经内置 JIT 编译器了，那么为什么还需要再使用解释器来“拖累”程序的执行性能呢？比如 JRockit VM 内部就不包含解释器，字节码全部都依靠即时编译器编译后执行，尽管程序的执行性能会非常高效，但程序在启动时必然需要花费更长的时间来进行编译。对于服务端应用来说，启动时间并非是关键重点，但对于那些看中启动时间的应用场景而言，或许就需要采用解释器与即时编译器并存的架构来换取一个平衡点。

既然 HotSpot VM 中采用了即时编译器，那么这就意味着将字节码编译为本地机器指令是一件运行时任务。在 HotSpot VM 中内嵌有两个 JIT 编译器，分别为 Client Compiler 和 Server Compiler，但大多数情况下我们简称为 C1 编译器和 C2 编译器。开发人员可以通过如下命令显式指定 Java 虚拟机在运行时到底使用哪一种即时编译器，如下所示：

- client: 指定 Java 虚拟机运行在 Client 模式下，并使用 C1 编译器；
- server: 指定 Java 虚拟机运行在 Server 模式下，并使用 C2 编译器。

除了可以显式指定 Java 虚拟机在运行时到底使用哪一种即时编译器外，默认情况下

HotSpot VM 则会根据操作系统版本与物理机器的硬件性能自动选择运行在哪一种模式下，以及采用哪一种即时编译器。简单来说，C1 编译器会对字节码进行简单和可靠的优化，以达到更快的编译速度；而 C2 编译器会启动一些编译耗时更长的优化，以获取更好的编译质量。不过在 Java7 版本之后，一旦开发人员在程序中显式指定命令“-server”时，缺省将会开启分层编译（Tiered Compilation）策略，由 C1 编译器和 C2 编译器相互协作共同来执行编译任务。不过在早期版本中，开发人员则只能通过命令“-XX:+TieredCompilation”手动开启分层编译策略。

之前笔者曾经提及过，缺省情况下 HotSpot VM 是采用解释器与即时编译器并存的架构，当然开发人员可以根据具体的应用场景，通过命令显式地为 Java 虚拟机指定在运行时到底是完全采用解释器执行，还是完全采用即时编译器执行。如下所示：

- Xint: 完全采用解释器模式执行程序；
- Xcomp: 完全采用即时编译器模式执行程序；
- Xmixed: 采用解释器+即时编译器的混合模式共同执行程序。

在此大家需要注意，如果 Java 虚拟机在运行时完全采用解释器执行，那么即时编译器将会停止所有的工作，字节码将完全依靠解释器逐行解释执行。反之如果 Java 虚拟机在运行时完全采用即时编译器执行，但解释器仍然会在即时编译器无法进行的特殊情况下介入执行，以确保程序能够最终顺利执行。

由于即时编译器将本地机器指令的编译推迟到了运行时，自此 Java 程序的运行性能已经达到了可以和 C/C++ 程序一较高下的地步。这主要是因为 JIT 编译器可以针对那些频繁被调用的“热点代码”做出深度优化，而静态编译器的代码优化则无法完全推断出运行时热点，因此通过 JIT 编译器编译的本地机器指令比直接生成的本地机器指令拥有更高的执行效率也就理所当然了。比如使用 Python 实现的 PyPy 执行器，比使用 C 实现的 CPython 解释器更加灵活，更重要的是，在程序的运行性能上进行比较，PyPy 将近是 CPython 解释器执行效率的 1 至 5 倍，这就是对 JIT 技术魅力的一个有力证明。并且 Java 技术自身的诸多优势同样也是 C/C++ 无法比拟的，所谓各有所长就是这个道理。在此大家需要注意，世界上永远没有最好的编程语言，只有最适用于具体应用场景的编程语言。

8.3.1 查阅 HotSpot 的运行时执行模式

尽管从 Java5 版本开始，缺省情况下 HotSpot VM 会根据操作系统版本与物理机器的硬件性能自动选择究竟是运行在 Client 模式下，还是运行在 Server 模式下，不过物理机器的性能究竟要达到一个什么样的标准才算是符合“服务器级别”的运行环境标准呢？简单来说，如果走双核或者以上，并且内存在 2GB 或者以上的物理机器，HotSpot VM 就会认定符合“服

务器级别”的标准，因此缺省也就会以 Server 模式运行。在此大家需要注意，如果是 32 位的 Windows 平台，HotSpot VM 在任何情况下都以 Client 模式运行。

如果大家希望查阅当前 HotSpot VM 在运行时缺省是以什么样的模式运行，可以通过命令“java -version”进行查阅，如下所示：

代码 8-8 执行命令“java -version”

```
java version "1.7.0_15"
Java(TM) SE Runtime Environment (build 1.7.0_15-b03)
Java HotSpot(TM) Client VM (build 23.7-b01, mixed mode, sharing)
```

由于笔者的物理机器上所安装的操作系统为 32 位的 Windows，因此 HotSpot VM 将会选择以 Client 模式运行。当然如果是希望以 Server 模式运行，则可以显式地通过命令“java -server”强制 HotSpot VM 以 Server 模式运行。如下所示：

代码 8-9 执行命令“java -server -version”

```
java version "1.7.0_15"
Java(TM) SE Runtime Environment (build 1.7.0_15-b03)
Java HotSpot(TM) Server VM (build 23.7-b01, mixed mode, sharing)
```

如代码 8-8 和代码 8-9 所示，HotSpot VM 在默认情况下都是采用解释器+即时编译器的混合模式共同执行程序。在 8.3 节中，开发人员可以通过命令显式指定 Java 虚拟机究竟是采用解释器还是即时编译器的方式执行程序。如下所示：

代码 8-10 执行命令“-Xint”和“-Xcomp”

```
>java -Xint -version
java version "1.7.0_15"
Java(TM) SE Runtime Environment (build 1.7.0_15-b03)
Java HotSpot(TM) Client VM (build 23.7-b01, interpreted mode, sharing)

>java -Xcomp -version
java version "1.7.0_15"
Java(TM) SE Runtime Environment (build 1.7.0_15-b03)
Java HotSpot(TM) Client VM (build 23.7-b01, compiled mode, sharing)
```

如果开发人员并没有显式指定有命令“-Xint”或“-Xcomp”的话，HotSpot VM 默认采用“-Xmixed”混合模式。

8.3.2 解释器的工作机制与构成模块

在 1.2.2 节中，笔者曾经分析过 Java 程序实现跨平台的基石就是字节码。在那个年代，JVM 设计者们的初衷仅仅只是单纯地为了满足 Java 程序实现跨平台特性，因此避免采用静态编译的方式直接生成本地机器指令，从而诞生了实现解释器在运行时采用逐行解释字节码执行程序的想法。由于解释器在设计和实现上非常简单，因此除了 Java 语言之外，还有许多高级语言同样也是基于解释器执行的，比如 Python、Perl、Ruby 等。但是在今天，基于解释器执行已经沦为低效的代名词，并且时常被一些 C/C++ 程序员所调侃，即便现在 HotSpot VM 中已经内嵌先进的 JIT 编译器，但 Java 程序的执行性能或多或少地还是被一些并不真正了解 Java 的开发人员所误解，不过无论如何，基于解释器的执行模式仍然为中间语言的发展做出了不可磨灭的贡献。

对于大多数 Java 开发人员而言，对解释器的执行机制或许并不会感觉到陌生。简单来说，当 Java 虚拟机启动时会根据预定义的规范对字节码采用逐行解释的方式执行，解释器真正意义上所承担的角色就是一个运行时“翻译者”，将字节码文件中的内容“翻译”为对应平台的本地机器指令执行。当一条字节码指令被解释执行完成后，接着再根据 PC 寄存器中记录的下一条需要被执行的字节码指令执行解释操作，这就是解释器的工作机制，同样也是解释器的工作任务。

解释器的任务就是负责将字节码指令解释为对应平台的本地机器指令执行。在 HotSpot VM 中，解释器主要由 Interpreter 模块和 Code 模块构成。其中 Interpreter 模块实现了解释器的核心功能，该模块中主要包括了两种类型的解释器，分别为模板解释器和 C++ 解释器。而 Code 模块主要用于管理 HotSpot VM 在运行时生成的本地机器指令。那么本书接下来将会围绕这两个模块展开讲解。

Interpreter 模块

Interpreter 模块实现了解释器的核心功能，该模块中主要包括了两种类型的解释器，分别为模板解释器和 C++ 解释器。其中模板解释器官方版本正在使用，而 C++ 解释器官方版本却并没有在使用。在 HotSpot VM 中，模板解释器和 C++ 解释器分别由 Interpreter 模块中的 TemplateInterpreter 子模块和 CppInterpreter 子模块实现。除此之外，Interpreter 模块中还包含一个非常重要的子模块，那就是 AbstractInterpreter 模块，该模块中定义了基于汇编模型的解释器和解释器生成器的抽象行为，以及还包含了一些与平台无关的公共操作。

Code 模块

Code 模块主要用于管理 HotSpot VM 在运行时生成的本地机器指令。在 Code 模块中，不得不提及的就是 CodeCache 子模块，该模块也被称之为代码缓存模块，主要用于缓存由

HotSpot VM 在运行时生成的本地机器指令。当 Java 虚拟机启动时，会在内存空间中为其分配一块内存区域，专门用于缓存本地机器指令，该空间与方法区一起被合称为非堆内存。当然开发人员可以通过如下选项设置代码缓存区的内存大小，如下所示：

表 8-1 代码缓存区的内存选项配置

选项	缺省值	描述	备注
-XX:CodeCacheExpansionSize	与平台相关	设置代码缓存区扩展大小的参数	暂无
-XX:InitialCodeCacheSize	与平台相关	设置代码缓存区的初始内存	暂无
-XX:ReservedCodeCacheSize	与平台相关	设置代码缓存区的最大内存	暂无

8.3.3 JIT 编译器的工作机制与构成模块

HotSpot VM 的执行引擎主要由解释器与 JIT 编译器构成，并且在运行时如果开发人员并没有通过选项“-Xint”或“-Xcomp”显式指定执行模式，那么 HotSpot VM 缺省将会以采用解释器+即时编译器的混合模式共同执行程序。当然是否需要启动 JIT 编译器将字节码直接编译为对应平台的本地机器指令，则需要根据代码被调用执行的频率而定。关于那些需要被编译为本地代码的字节码，也被称之为“热点代码”，JIT 编译器在运行时会针对那些频繁被调用的“热点代码”做出深度优化，将其直接编译为对应平台的本地机器指令，以此提升 Java 程序的执行性能，这就是为什么 HotSpot VM 会被称之为 HotSpot 的原因。解释器与 JIT 编译器混合执行流程，如图 8-12 所示^②：

^② 图片来源于 RednaxelaFX 的《JVM 分享：Java Program in Action》PPT。

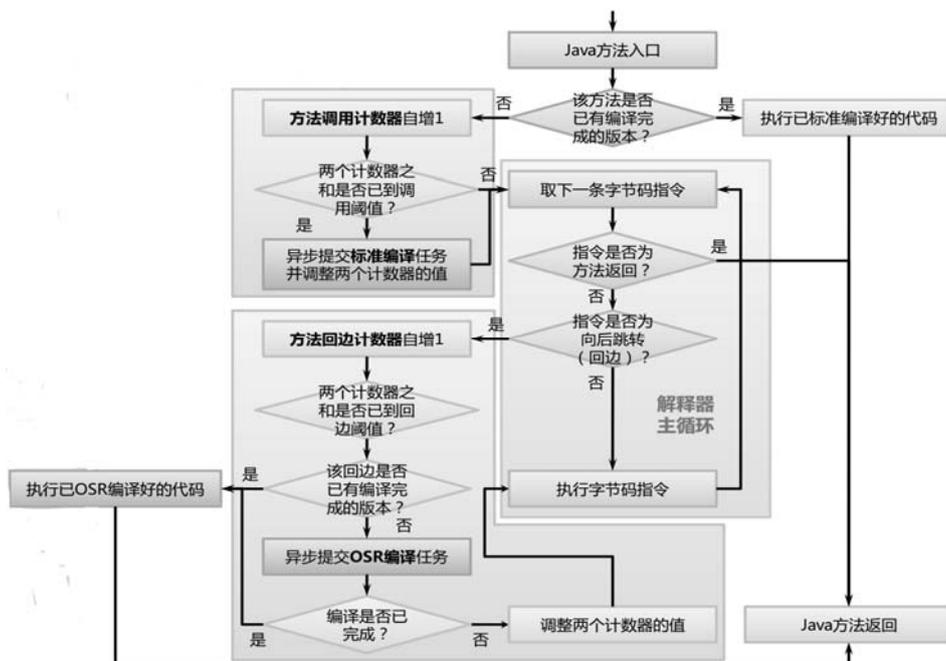


图 8-12 Client VM 模式下的解释执行与编译执行

那么“热点代码”的衡量标准究竟是什么？也就是说，一段代码究竟需要被执行多少次，才能被标记为“热点代码”然后经由 JIT 编译器执行编译呢？关于“热点代码”的具体探测方式，请阅读 8.3.5 节。

在 HotSpot VM 中，JIT 编译器主要由 C1 模块、Opto 模块和 Shark 模块构成。其中 C1 模块实现了 Client Compiler 编译器；而 Opto 模块实现了 Server Compiler 编译器，因此 C2 编译器也可以被称之为 Opto 编译器；Shark 模块中则实现了一个基于 LLVM 的编译器。在此大家需要注意，本章所有关于 JIT 编译器的内容都仅针对 C1 编译器，而关于 C2 编译器与 Shark 模块中的内容，大家则可以参考其他 JVM 文档或者直接阅读 HotSpot VM 的源码。

8.3.4 分层编译策略

之前笔者曾经提及过，缺省情况下 HotSpot VM 会根据操作系统版本与物理机器的硬件性能自动选择运行在哪一种模式下，以及采用哪一种即时编译器。如果开发人员通过命令“-client”显式指定 Java 虚拟机运行在 Client 模式下，那么就将会使用 C1 编译器，反之则运行在 Server 模式下，并使用 C2 编译器。不过在 Java7 版本之后，一旦开发人员在程序中显式指定有命令“-server”时，缺省将会开启分层编译（Tiered Compilation）策略，由 C1 编译器和 C2 编译器相互协作共同来执行编译任务。分层编译策略将会根据编译器执行编译、

优化的规模与耗时，划分出不同的编译层次，如下所示：

- 第 0 层：程序由解释器解释执行，但解释器并不开启性能监控功能，可触发第一层编译；
- 第 1 层：程序由 C1 编译器编译为本地机器指令执行，C1 编译器会对字节码进行简单和可靠的优化，以达到更快的编译速度；
- 第 2 层：程序由 C2 编译器编译为本地机器指令执行，但 C2 编译器会启动一些编译耗时更长的优化（代码将有可能被重复编译多次），甚至有可能根据性能监控信息进行一些不可靠的激进优化；
- 第 3 层：程序由 C1 编译器编译为本地机器指令执行，采集性能数据进行优化措施；
- 第 4 层：程序由 C2 编译器编译为本地机器指令执行，进行完全优化。

包含在 `hotspot/src/share/vm/utilities/globalDefinitions.hpp` 中的枚举类型 `CompLevel` 定义了 HotSpot VM 的编译层次，如下所示：

代码 8-11 `globalDefinitions.hpp` 中定义的编译层次

```
enum CompLevel {
    CompLevel_any          = -1,
    CompLevel_all          = -1,
    CompLevel_none         = 0,          // Interpreter
    CompLevel_simple       = 1,        // C1
    CompLevel_limited_profile = 2,    // C1, invocation & backedge counters
    CompLevel_full_profile  = 3,    // C1, invocation & backedge counters
+ mdo
    CompLevel_full_optimization = 4, // C2 or Shark
    ... ..
};
```

当 Java 虚拟机启动时，解释器可以首先发挥作用，而不必等待即时编译器全部编译完成后再执行，这样可以省去许多不必要的编译时间，即工作在第 0 层。接下来将会由编译策略模块决定到底是启用第 2 层还是第 3 层编译，当在第 3 层完成性能采集后，将会逐步过渡到第 4 层编译。在此大家需要注意，编译过程中，并不是所有的“热点代码”都需要经由第 4 层执行编译，因为如果是一些并不重要的代码，将会直接在第 1 层执行编译。

8.3.5 热点探测功能

究竟什么是“热点代码”？简单来说，一个被多次调用的方法，或者是一个方法体内部循环次数较多的循环体都可以被称之为“热点代码”，因此都可以通过 JIT 编译器编译为本地机器指令。对于那些被频繁调用的方法，JIT 编译器会将整个方法都作为编译目标并将其

编译为本地机器指令。但是对于方法体内部循环次数较多的循环体，JIT 编译器的编译目标同样也是针对整个方法而非循环体，由于这种编译方式发生在方法的执行过程中，因此也被称之为栈上替换，或简称为 OSR（On Stack Replacement）编译。

尽管频繁可以作为衡量“热点代码”的标准，但是一个方法究竟要被调用多少次，或者一个循环体究竟需要执行多少次循环才可以达到这个标准？也就是说，必然需要一个明确的阈值，JIT 编译器才会将这些“热点代码”编译为本地机器指令执行。衡量一段代码是否是“热点代码”主要依靠热点探测功能，目前 HotSpot VM 所采用的热点探测方式是基于计数器的热点探测。

采用基于计数器的热点探测，HotSpot VM 将会为每一个方法都建立 2 个不同类型的计数器，分别为方法调用计数器（Invocation Counter）和回边计数器（Back Edge Counter）。其中方法调用计数器用于统计方法的调用次数，而回边计数器则用于统计循环体执行的循环次数。一旦某个方法或者循环体的执行次数超出缺省所指定的阈值时，就可以被认定为“热点代码”，因此就可以被 JIT 编译器执行编译操作。基于计数器的热点探测在实现上稍显麻烦，因为需要为每一个方法都建立和维护两个私有的计数器，而且还不能直接获取方法的调用关系，不过采用这种方式相对来说却更加精准，大家权衡好利与弊，只不过是一种非常明显判定“热点代码”的探测方式。

笔者先从方法调用计数器开始讲起。在缺省情况下，Client VM 模式下的方法调用次数超过 1500 次时，将会触发 JIT 编译；而在 Server VM 模式下，方法的调用次数超过 10000 次时，才会触发 JIT 编译。当然开发人员可以通过选项“-XX:CompileThreshold”来设置方法调用计数器的阈值。如图 8-12 所示，当一个方法被调用时，第一步就是检查当前方法是否已经被 JIT 编译器执行过编译，是否存在编译后的版本，如果存在的话，则执行已经编译好的本地机器指令。如果不存在 JIT 编译后的版本，那么则将方法调用计数器的值加 1，然后判断方法调用计数器的值与回边计数器的值相加之和是否超过方法调用计数器的阈值，如果超过的话，将会向 JIT 编译器提交一个对当前方法执行编译的请求。

当然方法调用计数器中所统计的方法调用次数并非是一个绝对次数，而是一个瞬时次数。也就是说，方法调用计数器的值仅仅只会在某一段时间之内有效，一旦超出所规定的时间阈值时，如果当前方法被调用的次数仍然达不到编译标准，那么这个方法计数器的值就会消退 50%，这个消退过程也就被称之为方法调用计数器的热度消退，而这段时间就被称之为方法调用次数统计的消退周期。进行热度消退的动作是在进行 GC 时顺带一起执行的，当然开发人员可以通过选项“-XX:-UseCounterDecay”来关闭缺省的热度消退，这样一来方法调用计数器所统计的就会是方法被调用的绝对次数。除了可以使用选项“-XX:-UseCounterDecay”来关闭热度消退外，还可以使用选项“-XX:CounterHalfLifeTime”设置消退周期的时间阈值。

在此大家需要注意，如果程序中并没有显式设置“-XX:-BackgroundCompilation”或“-Xbatch”选项时，HotSpot VM 的执行引擎缺省将会按照异步模式在后台执行编译任务，程序的工作线程将不会出现暂停，而是按照解释执行的方式继续执行程序，直到提交的编译请求全部都被 JIT 编译器编译完成后，再异步安装编译结果执行已经编译好的本地机器指令。反之如果程序中显式设置这 2 个选项时，则意味着将会采用同步模式执行编译任务，这样一来一旦计数器提交有编译任务时，程序的工作线程将会出现暂停，直到 JIT 编译器完成编译之后，程序才会恢复执行。如图 8-13 所示。

当大家明白方法调用计数器之后，接下来笔者再来为大家讲解另外一个计数器：回边计数器。在缺省情况下，Client VM 模式下的循环次数超过 993 次时，将会触发 JIT 编译；而在 Server VM 模式下，循环次数超过 10700 次时，才会触发 JIT 编译。尽管 HotSpot VM 为回边计数器提供了一个类似于设置方法调用计数器阈值的选项“-XX:BackEdgeThreshold”，但是当前 HotSpot VM 中实际上并未使用该选项，因此开发人员只能够通过另外一个选项“-XX:OnStackReplacePercentage”来间接地调整回边计数器的阈值，其计算公式如下：

- Client VM 模式下，回边计数器的阈值计算公式：

$$(\text{CompileThreshold} * \text{OnStackReplacePercentage}) / 100$$

- Server VM 模式下，回边计数器的阈值计算公式：

$$(\text{CompileThreshold} * (\text{OnStackReplacePercentage} - \text{InterpreterProfilePercentage})) / 100;$$

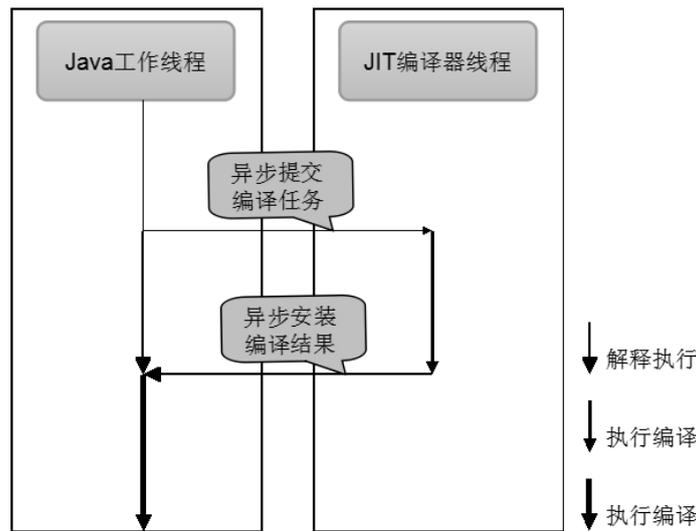


图 8-13(a) 异步提交编译任务与安装编译结果

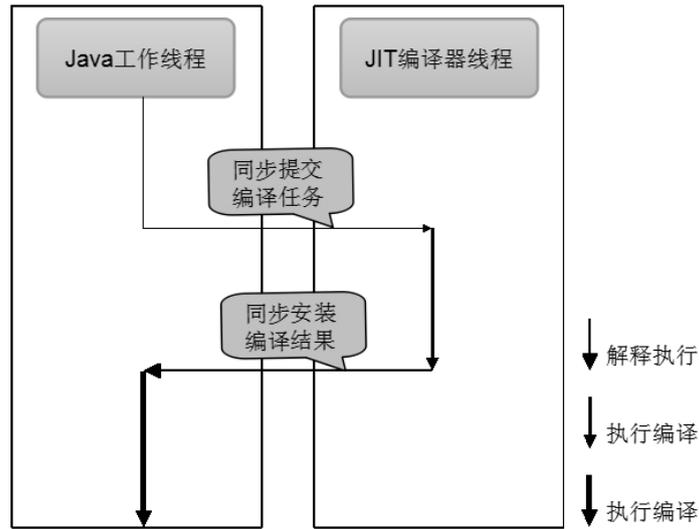


图 8-13(b) 同步提交编译任务与安装编译结果

为了使大家阅读更方便，本书示例了根据计数器触发编译条件的代码片段，如下所示：

代码 8-12 触发编译的条件

```

class InvocationCounter {
    private:
        unsigned int _counter; // bit no: |31 3| 2 | 1 0 |
                               // format: [count|carry|state]
        enum PrivateConstants {
            number_of_state_bits = 2,
            number_of_carry_bits = 1,
            number_of_noncount_bits = number_of_state_bits +
            number_of_carry_bits,
            number_of_count_bits = BitsPerInt -
            number_of_noncount_bits,
            // ...
        };
        // ...
};

void InvocationCounter::reinitialize(bool delay_overflow) {
    InterpreterInvocationLimit =CompileThreshold
    << number_of_noncount_bits;
    InterpreterProfileLimit =
    ((CompileThreshold * InterpreterProfilePercentage) / 100)
    << number_of_noncount_bits;
    // ...
    if(ProfileInterpreter) {
        InterpreterBackwardBranchLimit =
        (CompileThreshold * (OnStackReplacePercentage-
        InterpreterProfilePercentage)) / 100;
    }
}

```

```

    } else {
        InterpreterBackwardBranchLimit =
            ((CompileThreshold * OnStackReplacePercentage) / 100)
            << number_of_noncount_bits;
    }
}

```

如图 8-12 所示，当解释器遇到一条回边指令时，首先会和方法调用计数器一样检查即将执行的代码片段是否存在编译后的版本，如果存在的话，则执行已经编译好的本地机器指令。如果不存在 JIT 编译后的版本，那么则将回边计数器的值加 1，然后判断方法调用计数器的值与回边计数器的值相加之和是否超过回边计数器的阈值，如果超过的话，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 JIT 编译器的编译结果。

在此大家需要注意，由于回边计数器并没有和方法调用计数器一样存在热度消退的过程，因此回边计数器所统计的就是当前方法体内部循环体执行循环的绝对次数。

8.4 本章小结

想要了解执行引擎的工作原理，必然首先需要对栈帧非常熟悉。本章笔者详细地讲解了局部变量表、操作数栈、动态链接以及方法返回值等栈帧的几个组成结构后，又为大家详细地比较了基于寄存器架构和栈式架构的虚拟机在设计和实现上的区别。考虑到频繁访问操作数栈将会造成运行性能的下降，因此笔者又从源码的角度出发，为大家介绍了 HotSpot 中的栈顶缓存（ToS, Top-of-Stack Caching）技术是如何将栈顶元素全部缓存在物理 CPU 的寄存器中。除了为大家介绍 HotSpot VM 的架构模型外，笔者还对解释器和 JIT 编译器的工作流程进行了系统的分析，并重点讲解了 JIT 编译器的分层编译策略和“热点代码”的探测方式。

本章是本书的结尾，当大家阅读完本书的所有内容后，仍然还需要结合 HotSpot VM 的源码进行阅读和分析，这样才能加深对底层原理的认识和理解。