

In Doing We Learn.

- ★ 他山之石，剖析HotSpot源代码**最佳实践**
- ★ 授之以渔，展示解决JVM**疑难杂症**的实用**技巧**
- ★ 面向实战，精心设计大量**练习**

HotSpot**实战**

● 陈涛 著



人民邮电出版社
POSTS & TELECOM PRESS

HotSpot实战

陈涛 著



人民邮电出版社
北京

图书在版编目（C I P）数据

HotSpot实战 / 陈涛著. -- 北京 : 人民邮电出版社,
2014.3
ISBN 978-7-115-34363-5

I. ①H… II. ①陈… III. ①虚拟处理机 IV.
①TP338

中国版本图书馆CIP数据核字(2014)第001815号

内 容 提 要

本书深入浅出地讲解了 HotSpot 虚拟机的工作原理，将隐藏在它内部的本质内容逐一呈现在读者面前，包括 OpenJDK 与 HotSpot 项目、编译和调试 HotSpot 的方法、HotSpot 内核结构、Launcher、OOP-Klass 对象表示系统、链接、运行时数据区、方法区、常量池和常量池 Cache、Perf Data、Crash 分析方法、转储分析方法、垃圾收集器的设计演进、CMS 和 G1 收集器、栈、JVM 对硬件寄存器的利用、栈顶缓存技术、解释器、字节码表、转发表、Stubs、Code Cache、Code 生成器、JIT 编译器、C1 编译器、编译原理、JVM 指令集实现、函数的分发机制、VTABLE 和 ITABLE、异常表、虚拟机监控工具（如 jinfo、jstack、jhat、jmap 等）的实现原理和开发方法、Attach 机制、基于 GUI 的 JVM 分析工具（如 MAT、VisualVM）等内容。

除了 HotSpot 技术，本书还对方法论进行了探讨。在各个章节的讲解中，都会有一些与系统运行机制相关的实战或练习，供读者练习。通过这些实战练习，不仅有助于读者加深对知识或原理的理解，更为重要的是，它还可以培养读者独立探索的思维方式，这有助于读者把知识融会贯通并灵活应用到实际项目中。

本书适合于已具有一定 Java 编程基础的读者，以及在 Java 或基于 JVM 的编程语言平台下进行各类软件开发的开发人员、测试人员和运维人员。对于 JVM 和编程语言爱好者来说，本书也具有一定的学习参考价值。

-
- ◆ 著 陈 涛
 - 责任编辑 杜 洁
 - 责任印制 程彦红 杨林杰
 - ◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路 11 号
 - 邮编 100164 电子邮件 315@ptpress.com.cn
 - 网址 <http://www.ptpress.com.cn>
 - 北京天宇星印刷厂印刷
 - ◆ 开本：800×1000 1/16
 - 印张：22.5
 - 字数：491 千字 2014 年 3 月第 1 版
 - 印数：1—3 000 册 2014 年 3 月北京第 1 次印刷
-

定价：69.00 元

读者服务热线：(010) 81055410 印装质量热线：(010) 81055316

反盗版热线：(010) 83155315

广告经营许可证：京崇工商广字第 0021 号

序

每个 Java 开发者都知道 Java 字节码是在 JRE 上执行的。JRE 中最重要的部分就是 Java 虚拟机 (JVM)，JVM 负责分析和执行 Java 字节码。通常情况下，Java 开发人员并不需要去关心 JVM 是如何运行的。即使不理解 JVM 的工作原理，也不会给开发人员带来过多困惑。但是，如果你了解 JVM 的话，就会更加了解 Java，并且能够解决很多看似棘手的问题。

很多开发工程师不愿意花时间去了解 JVM 的底层，因为了解的过程很辛苦，也很枯燥。陈涛喜欢专研技术，他不仅对 Java、C、C++熟悉，而且对操作系统底层也很熟悉。他的知识面也比较广，能够将理论很好地应用于实践中。《HotSpot 实战》便是他潜心研究和实践的成果。

本书第一次系统全面地剖析了具体的虚拟机产品（即 HotSpot，Oracle 官方虚拟机）的实现，填补了市场上这类图书的空白。作者不仅透彻地讲解了那些看似深奥的原理，还提供了很多容易上手的实践案例。该书的一个突出特色是：读者通过自己动手实践便可掌握原本难以理解的原理。这为读者学习 JVM 提供了一条轻松的途径。此外，书中还深入浅出地介绍了很多实战应用的方法和技巧，具有较强的现实意义。

陈涛是网易宝的核心开发人员之一，同时维护了网易宝的多个系统。网易宝是网易官方的在线支付系统，对开发工程师的技术要求极高。尤其是在逻辑上，不能有半点疏忽，因为任何错误都有可能导致几百万甚至上千万的损失。他在不耽误正常工作的同时能够完成一本高质量的技术书籍，是非常不容易的。

赵刚
网易宝系统负责人，资深技术专家

前言

在聚贤庄一役中，金庸先生描写了一场颇有意思而又寓意深刻的较量。玄难舍弃自己的成名绝技不用，使出习武之人众所周知的入门功夫太祖长拳与乔峰决战。仅区区几个回合下来，便引得群雄对玄难由衷地赞叹：“同样的一招入门拳法，在他手底竟有这么强大的威力”。而当乔峰也使出太祖长拳还以颜色时，众人更是情不自禁地喝彩：“武林高手毕生所盼望达到的拳术完美之境，竟在这一招中表露无疑”。这个场面给我深刻的启示：扎实的根基与持续的打磨，才是技术人员的修行之道。

技术能力的培养与武功的修行，同样遵循着循序渐进的发展规律。在达到高手境界之前，每个人都需要从零起步并坚持不懈地学习。这听起来似乎很难，毕竟我们不知道会遇到多少困难。但有一个天大的困难却是显而易见的，那就是技术人员自身的浮躁。

近些年来，互联网技术犹如开足马力的高速列车一样，在飞速地前进着。似乎“快”一词已经成了时下互联网领域最贴切的写照。为了维持市场竞争力，我们必须持续地、快速地更新自身的产品和服务。而技术人员很容易在紧迫的 `deadline` 面前忘却了自身的技术追求。我们，正在变得浮躁起来。

为什么会写这本书

“蚓无爪牙之利，筋骨之强，上食埃土，下饮黄泉，用心一也。蟹六跪而二螯，非蛇蟮之穴无可寄托者，用心躁也”。对于技术人员来说，如果长期忽略自身技术的根基而去一味地追求高层框架技术，这无疑是舍本逐末的做法。

相较于 C 或 C++ 程序员，我发现 Java 程序员更容易忽视基础技术。JVM 的出现，为程序员屏蔽了操作系统与硬件的细节，使得程序员从诸如内存管理这样的繁琐任务中解放出来。但这并不等同于允许 Java 程序员放弃对基础的重视。我们是否有过这样的经历，在遇到内存故障、丢包、网络协议设计、资源瓶颈、证书、二进制等问题时，往往会觉得比较棘手，在寻求解决思路时更是显得力不从心。这实质上是自身技术遇到了瓶颈难以突破所致。可怕的是，想

去深究的时候却无从下手。

我写这本书的初衷是为了唤起 Java 程序员对基础技术的重视。事实上，任何平台的程序员都应当了解平台的基本特性、实现机制以及接口，这是提高自身修养的必经之路。对于 Java 程序员来说，我们需要了解的平台就是 JVM。了解 JVM 的基本实现机制，不仅对于解决实际应用中诸如 GC 等虚拟机问题有直接帮助，还有利于我们更好地理解语言本身。

所幸的是，Oracle 官方已经将虚拟机项目的源码开放出来，这对于我们来说简直就是福音。本书将以 OpenJDK 和 HotSpot 为素材，深入浅出地讲解我们最为熟悉的一款虚拟机产品的实现。除了 Java 程序员，从事与 Java 或 JVM 相关的开发、测试、运维等技术人员也可以在本书中获益。

本书内容

本书深入浅出地讲解了 HotSpot 虚拟机的工作原理，将隐藏在它内部的本质内容逐一呈现在读者面前，包括 OpenJDK 与 HotSpot 项目、编译和调试 HotSpot 的方法、HotSpot 内核结构、Launcher、OOP-Klass 对象表示系统、链接、运行时数据区、方法区、常量池和常量池 Cache、Perf Data、Crash 分析方法、转储分析方法、垃圾收集器的设计演进、CMS 和 G1 收集器、栈、JVM 对硬件寄存器的利用、栈顶缓存技术、解释器、字节码表、转发表、Stubs、Code Cache、Code 生成器、JIT 编译器、C1 编译器、编译原理、JVM 指令集实现、函数的分发机制、VTABLE 和 ITABLE、异常表、虚拟机监控工具（如 jinfo、jstack、jhat、jmap 等）的实现原理和开发方法、Attach 机制、基于 GUI 的 JVM 分析工具（如 MAT、VisualVM）等内容。

除了 HotSpot 技术，本书强调了对方法论的探讨。在各个章节的讲解中，都会有一些与系统运行机制相关的实战或练习，供读者练手。通过这些实战练习，不仅有助于读者加深对知识或原理的理解，更为重要的是，它还可以培养读者独立探索的思维方式，这有助于读者把知识融会贯通并灵活应用到实际项目中。

古人云“授人以鱼，不如授之以渔”，本书并不是简单地列举那些高深莫测的知识点，而是力求将理论与实践有机地结合起来，培养读者独立分析 JVM 底层机制的能力。读者在今后的实践中，通过自己动手实践就能揭开 HotSpot 内部机制的神秘面纱，汲取到书本上没有介绍但是实际项目中又急需的“营养”。

本书适用读者

本书适合于已具有一定 Java 编程基础的读者，以及在 Java 或基于 JVM 的编程语言平台下进行各类软件开发的开发人员、测试人员以及运维人员。对于 JVM 和编程语言爱好者来说，本书也具有一定的学习参考价值。

感谢

最想感谢的人莫过于我的父母，感谢你们对我的养育之恩。直到如今，我初为人父，才深刻体会到你们为家庭付出的艰辛。感谢哥哥和姐姐，我为我们之间的手足之情感到骄傲。特别感谢我的妻子，在我写书期间女儿出生了，我因工作和写书没能抽出太多时间照顾你们，而你从不抱怨，相反，你却很好地照顾了家庭和孩子，令我深感幸福。感谢刚出生的女儿，你的乖巧和可爱为家庭注入了活力和感动，也是我前进的动力。感谢我的家人们，你们的关爱和支持是我终身最大的财富。

本书得以出版，需要感谢人民邮电出版社的编辑杜洁，她多次对稿件进行了审读并提出许多宝贵意见，催使本书顺利完成。感谢网易宝赵刚先生的垂青，让本书有了出版的机会。感谢通策集团陈双辉先生，您对技术和互联网的见解令我深感钦佩。感谢网易宝朱晓明女士的支持，并感谢网易宝技术团队为本书提出的许多宝贵意见和建议，能与你们共事是我莫大的荣幸。

参与本书编写的还有陈为松、崔正明、陶云、陈育键、陈元元、夏崔莹、徐飞、袁超、吴倩倩、郭超、王秋子、陈晓明、赵国强、伍和海、在此一并表示感谢。

陈 涛
2014年1月

目录

1 第1章 初识 HotSpot	1
1.1 JDK 概述	2
1.1.1 JCP 与 JSR	3
1.1.2 JDK 的发展历程	4
1.1.3 Java 7 的语法变化	7
1.2 动手编译虚拟机	13
1.2.1 源代码下载	13
1.2.2 HotSpot 源代码结构	13
1.2.3 搭建编译环境	15
1.2.4 编译目标	16
1.2.5 编译过程	17
1.2.6 编译常见问题	19
1.3 实战：在 HotSpot 内调试	
HelloWorld	20
1.3.1 认识 GDB	21
1.3.2 准备调试脚本	22
1.4 小结	26
2 第2章 启动	28
2.1 HotSpot 内核	28
2.1.1 如何阅读源代码	28
2.1.2 HotSpot 内核框架	36
2.1.3 Prims	37
2.1.4 Services	39
2.1.5 Runtime	43
2.2 启动	46
2.2.1 Launcher	46
2.2.2 虚拟机生命周期	48

2.2.3 入口：main 函数	50
2.2.4 主线程	51
2.2.5 InitializeJVM 函数	53
2.2.6 JNI_CreateJavaVM	
函数	55
2.2.7 调用 Java 主方法	56
2.2.8 JVM 退出路径	56
2.3 系统初始化	57
2.3.1 配置 OS 模块	58
2.3.2 配置系统属性	60
2.3.3 加载系统库	61
2.3.4 启动线程	62
2.3.5 vm_init_globals 函数：	
初始化全局数据结构	65
2.3.6 init_globals 函数：初始化	
全局模块	65
2.4 小结	69

3 第3章 类与对象	70
3.1 对象表示机制	71
3.1.1 OOP-Klass 二分模型	71
3.1.2 Oops 模块	71
3.1.3 OOP 框架与对象访问	
机制	73
3.1.4 Klass 与 instanceKlass	79
3.1.5 实战：用 HSDB 调试	
HotSpot	82
3.2 类的状态转换	87
3.2.1 入口：Class 文件	87
3.2.2 类的状态	92
3.2.3 加载	96

3.2.4 链接 101 3.2.5 初始化 104 3.2.6 实战：类的“族谱” 107 3.2.7 实战：系统字典 111 3.3 创建对象 113 3.3.1 实例对象的创建流程 114 3.3.2 实战：探测 JVM 内部 对象 116 3.4 小结 119	5.1.1 垃圾收集 160 5.1.2 分代收集 162 5.1.3 快速分配 165 5.1.4 栈上分配和逸出分析 167 5.1.5 GC 公共模块 167 5.2 垃圾收集器 170 5.2.1 设计演进 170 5.2.2 CMS 收集器 175 5.2.3 G1 收集器 180 5.3 实战：性能分析方法 184 5.3.1 获取 GC 日志 184 5.3.2 GC 监控信息 187 5.3.3 内存分析工具 189 5.3.4 选择合适的收集器与 GC 性 能评估 190 5.3.5 不要忽略 JVM Crash 日志 195 5.4 小结 196
4 第 4 章 运行时数据区 120 4.1 堆 121 4.1.1 Java 的自动内存管理 121 4.1.2 堆的管理 122 4.2 线程私有区域 125 4.2.1 PC 125 4.2.2 JVM 栈 126 4.3 方法区 126 4.3.1 纽带作用 127 4.3.2 常量池 130 4.3.3 常量池缓存： ConstantPoolCache 133 4.3.4 方法的表示： methodOop 134 4.3.5 方法的解析：将符号引用 转换成直接引用 138 4.3.6 代码放在哪里： ConstMethodOop 141 4.3.7 实战：探测运行时常 量池 142	6 第 6 章 栈 197 6.1 硬件背景：了解真实 机器 198 6.1.1 程序是如何运行的 198 6.1.2 x86 与栈帧 199 6.1.3 ARM 对 Java 硬件级加速： Jazelle 技术 202 6.2 Java 栈 203 6.2.1 寄存器式指令集与栈式指 令集 203 6.2.2 HotSpot 中的栈 204 6.2.3 栈帧 207 6.2.4 充分利用寄存器资源 210 6.2.5 虚拟机如何调用 Java 函数 212 6.2.6 优化：栈顶缓存 221 6.2.7 实战：操作数栈 223 6.3 小结 228
5 第 5 章 垃圾收集 159 5.1 堆与 GC 160	7 第 7 章 解释器和即时 编译器 229 7.1 概述 230 7.2 解释器如何工作 231 7.2.1 Interpreter 模块 232 7.2.2 Code 模块 234

8

第 8 章 指令集 268

- 8.1 再说栈式指令集 268
- 8.2 数据传送 270
 - 8.2.1 局部变量、常量池和操作数栈之间的数据传送 270
 - 8.2.2 数据传送指令 272
 - 8.2.3 实战：数组的越界检查 277
- 8.3 类型转换 279
- 8.4 对象的创建和操作 281
- 8.5 程序流程控制 282
 - 8.5.1 控制转移指令 282
 - 8.5.2 条件转移 283
 - 8.5.3 无条件转移 284
 - 8.5.4 复合条件转移 285
 - 8.5.5 实战：switch 语句如何使用 String 287
- 8.6 运算 290
 - 8.6.1 加法：iadd 290
 - 8.6.2 取负：ineg 291
- 8.7 函数的调用和返回 292
 - 8.7.1 Java 函数分发机制：VTABLE 与 ITABLE 293

9

第 9 章 虚拟机监控工具 313

- 9.1 Attach 机制 314
 - 9.1.1 AttachProvider 与 VirtualMachine 314
 - 9.1.2 命令的下发：execute() 317
 - 9.1.3 命令的执行：Attach Listener 守护线程 319
- 9.2 查看 JVM 进程 320
 - 9.2.1 用 jps 查看 Java 进程 320
 - 9.2.2 实战：定制 jps，允许查看库路径 323
- 9.3 查看和配置 JVM 326
 - 9.3.1 用 jinfo 查看 JVM 参数配置 326
 - 9.3.2 实战：扩展 flags 选项，允许查看命令行参数 330
- 9.4 堆内存转储工具 332
 - 9.4.1 Heap Dump 332
 - 9.4.2 原理 333
- 9.5 堆转储分析 337
 - 9.5.1 Heap Dump 分析工具：jhat 337
 - 9.5.2 实战：MAT 分析过程 340
- 9.6 线程转储分析 343
 - 9.6.1 jstack 343
 - 9.6.2 实战：如何分析资源等待 344
- 9.7 小结 347

- 7.2.3 字节码表 235
- 7.2.4 Code Cache 236
- 7.2.5 InterpreterCodelet 与 Stub 队列 239
- 7.2.6 Code 生成器 241
- 7.2.7 模板表与转发表 244
- 7.2.8 实战：
InterpreterCodelet 247
- 7.3 即时编译器 250
 - 7.3.1 概述 250
 - 7.3.2 编译器模块 251
 - 7.3.3 编译器的基本结构 252
 - 7.3.4 实战：编译原理实践，了解编译中间环节 255
- 7.4 小结 267

- 8.7.2 invoke 系列指令 297
- 8.7.3 动态分发：覆盖 299
- 8.7.4 静态分发：重载 302
- 8.8 异常 305
 - 8.8.1 异常表 305
 - 8.8.2 创建异常 306
 - 8.8.3 try-catch 309
 - 8.8.4 finally 311
- 8.9 小结 312

第 1 章 初识 HotSpot



“知止而后有定，定而后能静，静而后能安，安而后能虑，虑而后能得。”

——《大学》

本章内容

- JVM 与 HotSpot VM
- 开源项目 OpenJDK 与 HotSpot 项目
- Java 语言特性的发展，以及 JCP 和 JSR 的推动作用
- Coin 项目为 Java 7 贡献的新特性
- GDB 调试工具的基本使用方式
- HotSpot 工程的编译与调试方法

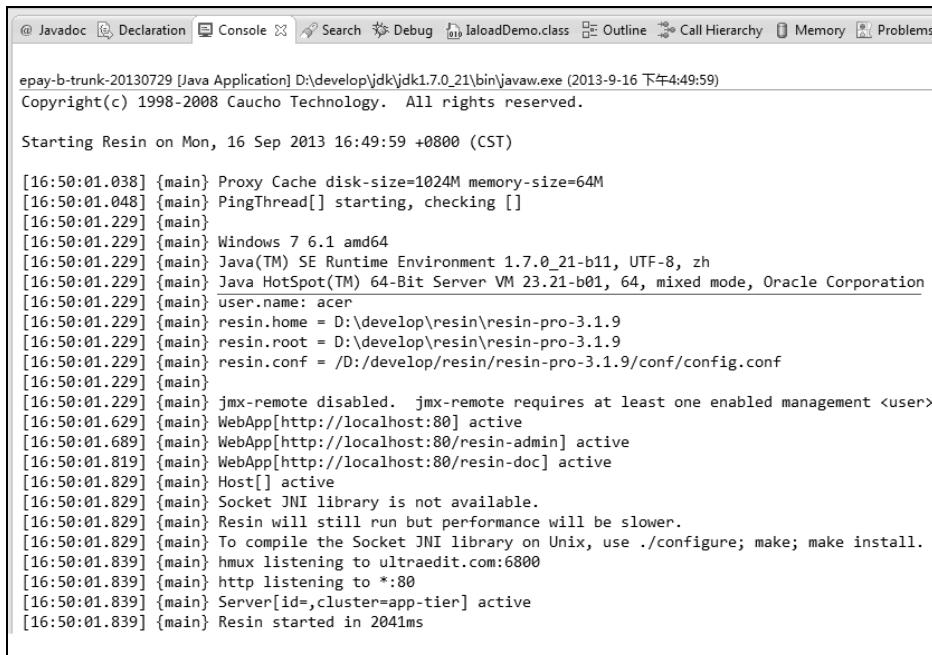
对于 Java 程序员来说，启动一个应用服务器是再平常不过的工作了。不知读者是否留意过，在启动应用服务器时，控制台可能会有关于 HotSpot 的信息输出，如图 1-1 所示。

在图 1-1 中，划线部分的字符串描述的是关于 Java 虚拟机（Java Virtual Machine，缩写为 JVM）产品的基本信息。应用服务器启动了一款名为“HotSpot”的 JVM。我们也可以直接在命令行中敲入“java -version”命令查看虚拟机版本信息，如图 1-2 所示。

HotSpot 是 Oracle JDK 官方的默认虚拟机，因此它也顺理成章地成为了 JVM 家族¹里最为家

¹ 目前市场份额较高的几款 Java 虚拟机分别是 Oracle HotSpot、BEA JRockit 和 IBM VM。BEA 后来被 Oracle 收购，这样一来，Oracle 就拥有了两款优秀的 JVM 产品。Oracle 官方宣布，未来会将 HotSpot 与 JRockit 合并。

喻户晓的产品。对于大多数Java程序员来说，HotSpot是与我们打交道最为频繁的一款虚拟机。



The screenshot shows the Eclipse IDE interface with the 'Console' tab selected. The output window displays the following log entries:

```

@ Javadoc Declaration Console Search Debug IloadDemo.class Outline Call Hierarchy Memory Problems

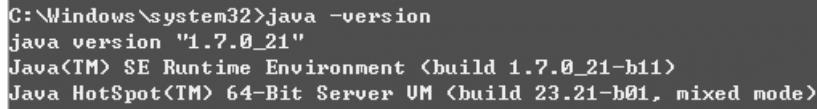
epay-b-trunk-20130729 [Java Application] D:\develop\jdk\jdk1.7.0_21\bin\javaw.exe (2013-9-16 下午4:49:59)
Copyright(c) 1998-2008 Caucheo Technology. All rights reserved.

Starting Resin on Mon, 16 Sep 2013 16:49:59 +0800 (CST)

[16:50:01.038] {main} Proxy Cache disk-size=1024M memory-size=64M
[16:50:01.048] {main} PingThread[] starting, checking []
[16:50:01.229] {main}
[16:50:01.229] {main} Windows 7 6.1 amd64
[16:50:01.229] {main} Java(TM) SE Runtime Environment 1.7.0_21-b11, UTF-8, zh
[16:50:01.229] {main} Java HotSpot(TM) 64-Bit Server VM 23.21-b01, 64, mixed mode, Oracle Corporation
[16:50:01.229] {main} user.name: acer
[16:50:01.229] {main} resin.home = D:\develop\resin\resin-pro-3.1.9
[16:50:01.229] {main} resin.root = D:\develop\resin\resin-pro-3.1.9
[16:50:01.229] {main} resin.conf = /D:/develop/resin/resin-pro-3.1.9/conf/config.conf
[16:50:01.229] {main}
[16:50:01.229] {main} jmx-remote disabled. jmx-remote requires at least one enabled management <user>
[16:50:01.629] {main} WebApp[http://localhost:80] active
[16:50:01.689] {main} WebApp[http://localhost:80/resin-admin] active
[16:50:01.819] {main} WebApp[http://localhost:80/resin-doc] active
[16:50:01.829] {main} Host[] active
[16:50:01.829] {main} Socket JNI library is not available.
[16:50:01.829] {main} Resin will still run but performance will be slower.
[16:50:01.829] {main} To compile the Socket JNI library on Unix, use ./configure; make; make install.
[16:50:01.839] {main} hmx listening to ultraedit.com:6800
[16:50:01.839] {main} http listening to :80
[16:50:01.839] {main} Server[id=,cluster=app-tier] active
[16:50:01.839] {main} Resin started in 2041ms

```

图1-1 启动HotSpot VM



```
C:\Windows\system32>java -version
java version "1.7.0_21"
Java(TM) SE Runtime Environment (build 1.7.0_21-b11)
Java HotSpot(TM) 64-Bit Server VM (build 23.21-b01, mixed mode)
```

图1-2 java -version

对于这个我们赖以生存的系统平台，我们又是否真的了解它是如何工作的呢？在实际应用中，我们是否曾屡次被它的“顽皮”折磨得筋疲力尽，又因不了解它的“脾气”而束手无策？我们能否在实际应用中驾驭好它呢？

事实上，我们之所以会遇到这些困扰，是因为对虚拟机的了解还不够。只要我们积累了足够的知识，是完全可以在实践中处理好虚拟机问题的。接下来，就让我们正式开启HotSpot的学习之旅吧。

1.1 JDK概述

Java是一门不断发展和壮大的语言，随着理论和应用的飞速发展，它不断地吸收有益营养，许多优良特性也在JDK的各个版本中逐渐添加进来。

JDK 在一开始并不是开源的。随着开源运动的蓬勃发展，Sun 公司在 2006 年的 JavaOne 大会上声称将对 Java 开放源代码，开源的 Java 平台开发主要集中在 OpenJDK 项目上。Sun 公司于 2009 年 4 月 15 日正式发布 OpenJDK。Java 7 则是 Java 开源后发布的第一个正式版本。

任何组织和个人都可以为 Java 的发展做出贡献，如果你愿意为 OpenJDK 项目添砖加瓦，可以从 AdoptOpenJDK 起步，AdoptOpenJDK 是一个旨在帮助开发人员更好地加入 OpenJDK 项目而提供指导的社区²。

在 Java 和 OpenJDK 的发展中，JCP 起到了重要的推动作用。

1.1.1 JCP 与 JSR

JCP(Java Community Process)是一套制定 Java 技术规范的机制，通过制定和审查 JSR(Java Specification Requests) 推动 Java 技术规范的发展。

一个已提交的 JSR 要想成为最终状态，需要经过正式的公开审查，并由 JCP 执行委员会投票决定。最终的 JSR 会提供一个参考实现，它是免费而且公开源代码的；除此之外，还需要提供一个用来验证是否符合 API 规范的技术兼容性工具包（Technology Compatibility Kit）。

JSR 并非只由 Oracle 管理，任何个人都可以注册并参与审查 JSR。只要你有足够的兴趣和热情，都可以注册成为 JCP 成员，并参加 JSR 的专家组，甚至提交自己的 JSR 建议。

在 JCP 官网 (<http://www.jcp.org/>) 中可以查看所有的 JSR。对 Java 语言发展动态感兴趣的读者来说，跟踪 JSR 的发展动态是一条不错的学习途径。目前提交的 JSR 大概有 300 多个，包括我们熟知的一些特性都是通过 JSR 提交的，如 Java 模块化、动态语言支持等，如表 1-1 所示。

表 1-1 部分 JSR

编号	主要内容	名 称
JSR 5	XML	XML Parsing Specification
JSR 14	范型	Add Generic Types To The Java Programming Language
JSR 47	日志 API	Logging API Specification
JSR 51	NIO	New I/O APIs for the Java Platform
JSR 56	网络协议	Java Network Launching Protocol and API
JSR 59	Merlin	J2SE Merlin Release Contents
JSR 63	XML	Java API for XML Processing 1.1
JSR 166	并发 API	Concurrency Utilities
JSR 175	注解	A Metadata Facility for the Java Programming Language
JSR 176	J2SE 5.0 (Tiger)	J2SE 5.0 (Tiger) Release Contents
JSR 199	Java 编译器 API	Java Compiler API

² AdoptOpenJDK 社区官网为 <https://java.net/projects/adoptopenjdk/pages/AdoptOpenJDK>。

续表

编号	主要内容	名 称
JSR 201	枚举、自动装箱等	Extending the Java Programming Language with Enumerations, Autoboxing, Enhanced for loops and Static Import
JSR 221	JDBC 4.0 API	JDBC 4.0 API Specification
JSR 223	脚本引擎	Scripting for the Java Platform
JSR 224	JAX-WS	Java API for XML-Based Web Services (JAX-WS) 2.0
JSR 270	Java SE 6	Java SE 6 Release Contents
JSR 292	动态语言支持	Supporting Dynamically Typed Languages on the Java Platform
JSR 294	模块化	Improved Modularity Support in the Java Programming Language
JSR 308	注解	Annotations on Java Types
JSR 310	日期和时间 API	Date and Time API
JSR 334	Coin 项目增强	Small Enhancements to the Java Programming Language
JSR 354	货币 API	Money and Currency API

1.1.2 JDK 的发展历程

接下来，就让我们翻开历史的画卷，看一看在 JDK 的发展历程中各个优良的特性是如何添加到 Java 中的。

1. 混沌初开

1996年1月23日，第一个稳定的正式版本JDK发布。该版本编号为JDK 1.0.2，Sun公司官方称为Java 1。

2. JDK 1.1

1997年2月19日，Sun公司发布JDK 1.1。在此版本中为Java增加了诸如AWT、内部类、Java Beans、JDBC、RMI和反射等特性。

3. J2SE 的开始

JDK 1.2也称为J2SE 1.2。1998年12月8日，Sun公司发布JDK 1.2，工程代号Playground(运动场)。

自该版本开始一直到后来的5.0版本，统称为Java 2平台。Java 2平台根据应用领域的不同，提供了3个版本：

- J2SE (Java 2 Standard Edition, Java 2 标准版);
- J2EE (Java 2 Enterprise Edition, Java 2 企业版);
- J2ME (Java 2 Micro Edition, Java 2 嵌入式版)。

也正是从该版本开始，官方开始使用新的术语“J2SE”来描述 Java 2 标准版的变化。JDK 1.2 为 Java 带来了许多的变化，包括 strictfp 关键字、Swing、Java Plug-in、Java IDL、Collections 集合框架等技术，并首次在虚拟机中引入了 JIT（即时）编译器。

4. J2SE 1.3

JDK 1.3 也称为 J2SE 1.3。2000 年 5 月 8 日，Sun 正式发布 JDK 1.3，其工程代号为 Kestrel（红隼）。该版本引入的技术包括：

- HotSpot 虚拟机；
- RMI 兼容 CORBA；
- JDDI；
- JPDA；
- JavaSound 等。

5. J2SE 1.4

JDK 1.4 也称为 J2SE 1.4。2002 年 2 月 6 日，Sun 正式发布 JDK 1.4，其工程代号为 Merlin（灰背隼）。该版本是经 JCP 的努力发布的第一个版本，变化很大，JSR 59 定义了改进的相关技术，包括：

- assert 关键字；
- 正则表达式；
- 异常链；
- NIO (JSR 51)；
- IPv6 支持；
- 日志 API (JSR 47)；
- 图形图像 API；
- 内置 XML 和 XSTL 解析器(即 JAXP, Java API for XML Processing, 见 JSR 5 和 JSR 63)；
- 内置安全和加密类库 JCE/JSSE/JAAS、Java Web Satrt (JSR 56) 等。

JDK 1.4 有两个修订版本：2002 年 9 月 16 日发布的工程代号为 Grasshopper（草蜢）的 JDK 1.4.1，以及在 2003 年 6 月 26 日发布的工程代号为 Mantis（螳螂）的 JDK 1.4.2。

6. Java 5

JDK 1.5 也称为 JDK 5、J2SE 5 或 Java 5。从该版本起，官方开始在公开版本(Product Version) 中使用 5.0、6.0 或 7.0 的版本号命名方式，仅在开发版本中沿用 JDK 1.5、JDK 1.6 或 JDK 1.7 的命名，在 Java 开发人员中，也广泛使用后者的命名方式。2004 年 9 月 30 日，Sun 官方正式

发布 JDK 1.5，工程代号为 Tiger（虎），对 Java 语法做了很多改进（JSR 176），包括：

- 范型（JSR 14）；
- 注解（JSR 175）；
- 装箱（JSR 201）；
- 枚举（JSR 201）；
- 可变长参数；
- foreach 循环（JSR 201）；
- 改进了 JMM（Java Memory Model，即 Java 内存模型）等。

7. Java 6

JDK 1.6 也称为 JDK 6、J2SE 6 或 Java 6。2006 年 12 月 11 日，Sun 官方发布 JDK 6。该版本的变化来自 JSR 270 的定义，工程代号为 Mustang（野马）。主要变化为：

- 脚本语言支持（JSR 223）；
- 改进了 Web Service 支持，JAX-WS（JSR 224）；
- JDBC 4.0（JSR 221）；
- Java 编译器 API（JSR 199），允许 Java 程序调用 Java 编译器；
- JAXB 2.0；
- 在虚拟机方面，提供了同步机制优化、编译器性能优化、垃圾收集增加新的算法、应用程序启动性能优化等功能。

8. Java 7

JDK 1.7 也常称为 JDK 7、J2SE 7 或 Java 7。工程代号为 Dolphin（海豚）。Oracle 收购 Sun 后，为了保证让进度一再延后的 JDK 7 正式版能够在 2011 年 7 月 28 日如期发布，不得不削减了原先的发布计划，将许多招致延误的项目迁移到了 JDK 8 的开发分支中。这样，在正式版最终发布后，JDK 7 的主要变化在于以下几个方面。

- 在虚拟机方面，提供一款性能优秀的 G1 收集器；
- 增强对动态语言的支持（JSR 292）；
- 在 64 位系统中，提供可压缩的对象指针（-XX:+UseCompressedOops）；
- Coin 项目贡献的 Java “语法糖”；
- 并发工具 API（JSR 166）；
- NIO.2；
- 网络协议库对新的协议支持，如 SCTP 协议和 SDP 协议等。

JDK 7 中曾计划但未按时完成的一些特性³，如 Lambda 表达式和部分来自 Jigsaw 项目和 Coin 项目的特性，将加入 JDK 8 的开发计划，预计随 JDK 8 在 2014 年春季发布。

9. Java 8

Oracle 计划在 2014 年 3 月正式发布 JDK 8⁴。该版本将包含原本在 JDK 7 的计划中但最终未能及时发布的特性，包括：

- Lambda 表达式，由 Lambda 项目⁵开发；
- 部分 Coin 项目提供的特性；
- JavaScript 引擎（JSR 223），允许 Java 程序嵌入 JavaScript 代码。由 Nashorn 项目⁶开发；
- Java 类型注解（JSR 308）；
- 日期和时间 API（JSR 310）。

10. Java 9

JDK 9 也称为 Java SE 9。在 2011 年的 JavaOne 大会上，Oracle 表示希望在 2016 年发布 JDK 9。JDK 9 可以很好地管理数 G 的 Java 堆，能够更好地与本地代码集成，并做到虚拟机自我调节。这些开发计划还包括：

- 模块化（JSR 294）；
- 货币 API（JSR 354）；
- 对 JavaFX 更好地集成。

1.1.3 Java 7 的语法变化

现在，我们将介绍一些 Java 7 带来的变化。说到 Java 7 的改进，Coin 项目为其做出了重要贡献。Coin 项目⁷是 OpenJDK 的子项目，它成立的主要目的是为 Java 语言贡献语法增强特性。本节将介绍 Coin 项目为 Java 7 贡献的几个语法新特性。这些新特性通过 JSR 334 提交到 JCP。

这些新特性在并未对 Java 底层做很大改动的基础上，丰富了 Java 语法的表现形式，为 Java 程序员提供了更便捷的方式表达业务逻辑。这些特性主要包括：

- 允许 switch 语句中使用 String 表达式；

³ JDK 7 的 Java 特性开发计划，可以参考 <http://openjdk.java.net/projects/jdk7/features/#f700>。

⁴ 参见官方邮件列表 (<http://mail.openjdk.java.net/pipermail/jdk8-dev/2013-April/002336.html>)，即《Proposed new schedule for JDK 8》。

⁵ Lambda 项目官网为 <http://openjdk.java.net/projects/lambda/>。

⁶ Nashorn 项目官网为 <http://openjdk.java.net/projects/nashorn/>。

⁷ Coin 项目官网为 <http://openjdk.java.net/projects/coin/>。

- 允许数值以下划线分隔;
- 允许数值以二进制表示;
- 异常处理增强;
- TWR;
- 简化范型定义、简化变长参数的方法调用等。

下面我们选取一些有代表性的特性举例说明。本书在后续章节也安排了对一些特性的实现机制的探讨，感兴趣的读者敬请留意。

1. switch语句中使用String

在Java 7之前，使用switch语句的条件表达式类型只能是枚举类型，或者byte、char、short和int类型以及它们的包装类Byte、Character、Short和Integer。若要根据String类型进行条件选择，则需要做额外的转换。换句话说，尚需跨过Java语法层面的“障碍”才能进行业务逻辑处理，从某种程度上来说，这为程序员带来了额外的负担。

在Java 7中，对条件表达式的类型有所放宽，允许在条件表达式中出现实际应用中最常用的String类型。

举例来说，在金融应用中，常常需要根据银行名获得对应的银行机构代码。在清单1-1中，通过改进的switch语句，我们可以根据bankName轻松获取bankId，中间无需转换。

清单1-1

来源：com.hotspotinaction.demo.chap1.Switch

描述：switch语句中使用String

```

1  public String getBankIdByName(String bankName) {
2      String bankId = "";
3      switch (bankName) {
4          case "ICBC" :
5              bankId = "B00001";
6              break;
7          case "ABC" :
8              bankId = "B00002";
9              break;
10         case "CCB" :
11             bankId = "B00003";
12             break;
13         case "BOC" :
14             bankId = "B00004";
15             break;
16         default :
17             bankId = "UNVALID";
18     }
19     return bankId;
20 }
```

可以看到，这一处改进，让Java语法更加灵活，这样更利于程序员对业务逻辑进行处理。

练习 1

想一想，在 Java 6 及以前的版本中，如何实现 getBankIdByName()？

2. 数值字面量增强

(1) 二进制。

在 Java 7 之前，Java 支持的整数字面量包括 3 种进制：十进制（默认）、八进制（前缀“0”）和十六进制（前缀“0X”或“0x”），但是不支持二进制的直接表示，为此，我们需要进行内部的进制间转换。

例如，为了创建二进制数“0100”，我们可能需要使用十六进制“0x04”来间接表示。或者调用包装类 Integer 的 parseInt()方法得到它，如 Integer.parseInt("0100", 2)。但这些方式，都并不直观。

在 Java 7 中，可以直接使用二进制表示：“0b0100”或“0B0010”，如清单 1-2 所示，读者要体会新特性带来的微妙变化。

清单 1-2

来源：com.hotspotinaction.demo.chap1.Literals

描述：数值字面量增强

```
1  public int getBinaryInt(String number) { /* before Java 7 */  
2      int a = -1;  
3      try {  
4          a = Integer.parseInt(number, 2);  
5      } catch (NumberFormatException e) {  
6          // TODO 异常处理  
7      }  
8      return a;  
9  }  
  
10 public int getBinaryIntLiterals() { /* Java 7: 二进制字面量 */  
11     int a = 0b0100;  
12     return a;  
13 }
```

在 Java 7 以前，为了获得以二进制表示的数据，我们可能会使用类似 getBinaryInt()函数这样的实现方式。如第 1~9 行代码所示，getBinaryInt()函数根据传入的字符串参数，经过一番数据转换，方能得到期望的数据。而在这些转换过程中，为了程序的健壮性，还不得不编写更多的异常处理代码。而这一切，在 Java 7 中却可以用一种直观优雅的方式表现出来。如第 11 行代码所示，我们可以直接声明一个二进制整数，而无需做任何转换。

(2) 利用下划线增强可读性。

假设我们要处理一个长串数据，如人民币“10000000”，想象一下在电子商务系统中，如果多敲入或者少敲入一个零将会造成多大的麻烦。

在 Java 7 中，允许对这样的数据使用下划线进行字符分隔表示，如清单 1-3 所示。

清单 1-3

来源: com.hotspotinaction.demo.chap1.Literals

描述: 用下划线表示数值字面量

```
1 long a = 10_000_000L;
2 int b = 0b1100_1000_0011_0000;
```

这样，数据便能一目了然地置于代码中，程序的可读性也大大增强了。

3. 异常处理增强

在 Java 6 及之前的版本，我们要捕获多个异常，可以通过多个 catch 语句实现，如清单 1-4 所示。

清单 1-4

来源: com.hotspotinaction.demo.chap1.MutilCatch

描述: Java 7 之前捕获多个异常的方式

```
1 try {
2     a = a / i;
3     a = Integer.parseInt(number, 2);
4 } catch (NumberFormatException e) {
5     // TODO 异常处理
6 } catch (ArithmetricException e) {
7     // TODO 异常处理
8 }
```

在 Java 7 中，可以在一个 catch 表达式中对多种类型异常进行合并捕获，如清单 1-5 所示。

清单 1-5

来源: com.hotspotinaction.demo.chap1.MutilCatch

描述: Java 7 在一个 catch 中捕获多个异常

```
1 try {
2     a = a / i;
3     a = Integer.parseInt(number, 2);
4 } catch (NumberFormatException | ArithmetricException e) {
5     // TODO 异常处理
6 }
```

4. TWR (try-with-resources)

Java 7 中新增了 Try-With-Resources⁸（缩写为 TWR）语句，允许声明一个或多个资源。资源（resource）对象在使用它的程序结束时必须进行关闭。使用 TWR 语句，能够确保在语句运行结束前关闭资源。

任何实现了 java.lang.AutoCloseable 接口的对象，都可以成为 TWR 的资源。Java 7 中对 JDK 中大部分资源对象都重新定义了一番，让它们都实现了 java.lang.AutoCloseable 接口。如果需要在应用程序中自定义新的资源，请确保它已实现了 java.lang.AutoCloseable 接口，这样就可以充分应用 TWR 特性了。

下面我们实现一个打开 Http 接口并读取返回数据的工具方法。在 Java 6 以前的版本中，可

⁸ 更多内容可以参考 <http://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html>。

以这样实现，如清单 1-6 所示。

清单 1-6

来源：com.hotspotinaction.demo.chap1.HttpUtil
描述：TWR

```

1  public static String openUrl(String strUrl, String postParams) {
2      InputStream is = null;
3      OutputStream os = null;
4      String message = "";
5      try {
6          URL webURL = new URL(strUrl);
7          HttpURLConnection conn = (HttpURLConnection) webURL.openConnection();
8          conn.setDoOutput(true); // 打开写入属性
9          conn.setDoInput(true); // 打开读取属性
10         conn.setRequestMethod("POST"); // 设置提交方法
11         conn.connect();
12         os = conn.getOutputStream();
13         os.write(postParams.toString().getBytes());
14         os.flush();
15         if (conn.getResponseCode() != HttpURLConnection.HTTP_OK) {
16             System.err.println("response error=" + conn.getResponseCode());
17         }
18         is = conn.getInputStream();
19         message = getReturnValueFromInputStream(is);
20     } catch (Exception e) {
21         // TODO 异常处理
22     } finally {
23         if (is != null) {
24             try {
25                 is.close();
26             } catch (IOException e) {
27                 // TODO 异常处理
28             }
29         }
30         if (os != null) {
31             try {
32                 os.close();
33             } catch (IOException e) {
34                 // TODO 异常处理
35             }
36         }
37     }
38     return message;
39 }
```

显然，在 finally 代码块（第 22~37 行）中关闭 InputStream 和 OutputStream 需要写上不少代码，并且这种关闭的语法很不友好，一旦遗漏 finally 语句块那就很有可能造成应用程序的资源泄露，遗漏某个非空判断则也会导致程序在运行时出现异常。

Coin 项目提供了一个很贴心的改进——TWR，它让我们能以一种更为便捷的方式实现资源关闭，如清单 1-7 所示。

清单 1-7

来源：com.hotspotinaction.demo.chap1.HttpUtil
描述：TWR

```

1  public static String openUrlTWR(String strUrl, String postParams) {
2      URL webURL = null;
```

```

3     HttpURLConnection con = null;
4     String message = "";
5     try {
6         webURL = new URL(strUrl);
7         con = (HttpURLConnection) webURL.openConnection();
8         con.setDoOutput(true);
9         con.setDoInput(true);
10        con.setRequestMethod("POST");
11        con.connect();
12    try (OutputStream os = con.getOutputStream(); InputStream is=con.getInputStream()) {
13        os.write(postParams.toString().getBytes());
14        os.flush();
15        if (con.getResponseCode() != HttpURLConnection.HTTP_OK) {
16            System.err.println("response error=" + con.getResponseCode());
17        }
18        message = getReturnValueFromInputStream(is);
19    }
20    } catch (Exception e) {
21        // TODO 异常处理
22    }
23    return message;
24 }

```

只需要在 `try` 语句中声明程序块需使用的资源即可，省去了繁琐的资源泄露处理。将资源关闭任务交给系统去做，程序员只需要关注业务逻辑的实现就行了。

5. 简化泛型定义

泛型是一种强大而又安全的设计工具，它能够让程序员在编译期就能避免把错误类型的对象添加到容器中。

清单 1-8

来源: com.hotspotinaction.demo.chap1.SimplifiedGeneric

描述: 简化泛型定义

```

1  HashMap<String, HashMap<Long, String>> map1 = new HashMap<String, HashMap<Long,
String>>();
2  HashMap<String, HashMap<Long, String>> map2 = new HashMap<>();

```

如清单 1-8 所示，第 1 行代码首先向我们展示了在 Java 6 之前是如何创建一个类型明确的容器的。我们看到，在等号左边声明了一个类型为容器 `HashMap` 的对象 `map1`，这个容器以 `String` 类型作为 `key`，但它的 `value` 类型同样是一个 `HashMap` 容器，这个内嵌的容器以 `Long` 类型作为 `key` 并以 `String` 类型作为 `value`。这种容器内嵌入容器的设计方式，在日常工作中是很常见的。麻烦的是在等号右边，还要敲入与等号左边相同的类型声明代码。显然，这是多余的，毕竟，等号左边已经明确了 `map1` 的具体类型。

这种繁琐的语法在 Java 7 中得到了简化。第 2 行代码演示了应用 Java 7 改进语法后的效果。程序员在等式右边只需要输入少量代码，剩下的类型推断工作交给编译器去做就行了。本来就应该这样，不是吗？

1.2 动手编译虚拟机

源码面前，了无秘密。对于 OpenJDK 和 HotSpot 项目来说也是如此。因此，研究虚拟机实现机制的最佳途径就是阅读和调试源代码。我们希望能够动手编译一个完整的 OpenJDK（含 HotSpot）项目，或者仅编译 HotSpot，这样就可以对虚拟机展开调试了。

虽然官方也支持在 Windows 操作系统下构建编译环境。但是经验表明，选择在 Linux 环境下搭建编译环境，可以避免不少弯路。理由有以下两点：

- Windows 上为了得到完整的编译环境，需要借助 Cygwin 等虚拟环境，而在 Linux 环境下环境下则可以省去大量的环境准备工作，成本较低；
- 深入研究时，若遇到涉及操作系统内核上的困惑，开源的 Linux 内核较易获得。

即便如此，编译一个完整的 Open JDK 对开发者的要求还是较高的。我们选择的 Linux 发行版、内核版本、编译器（GCC/G++等）以及项目依赖的第三方库的差异，都有可能导致编译过程出错。因此，需要读者具备基本的 Linux 使用技能，能够在出现问题时找到解决方案。

由于本文关注的只是 HotSpot，所以编译完整的 OpenJDK 也不是必须完成的任务，若无法编译完整的 OpenJDK，我们可以仅编译 HotSpot，这就能满足大部分的学习需求。相较于编译完整的 OpenJDK，仅编译 HotSpot 项目就相对简单很多。简单说来，可以按照如图 1-3 所示的步骤进行操作。

- (1) 下载一套含有 HotSpot 项目的 JDK 源代码；
- (2) 搭建编译环境；
- (3) 配置编译目标；
- (4) 编译；
- (5) 运行测试程序，检测编译是否成功。

接下来，我们开始动手编译 HotSpot 吧。

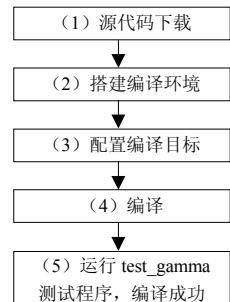


图 1-3 编译 HotSpot 工程
操作步骤

1.2.1 源代码下载

可以在 OpenJDK 官网 (<http://download.java.net/openjdk/jdk7/>) 下载源代码。将下载的源代码包解压后，可以找到一个名为 hotspot 的目录，该路径下就是 Hotspot 项目完整的源代码。

1.2.2 HotSpot 源代码结构

从 JVM 为语言的运行时提供支撑功能来看，虚拟机是 Java 语言的“系统程序”，但从本质上来说，它只是一个运行在操作系统上的普通应用程序而已。因此，对于我们来说，过分担心

它有多么的庞大和神秘，是完全没有必要的。

HotSpot项目主体是由C++实现的，并伴有少量C代码和汇编代码。此外，作为HotSpot的重要组成部分之一，Serviceability Agent⁹（可维护性代理，简称SA）及其他Agent则由Java代码实现。

HotSpot工程目录结构如图1-4所示。

在根目录Hotspot下有Agent、Make、Src和Test目录。其中Make目录包含了编译HotSpot的Makefile文件，Agent目录中的源代码主要实现SA，而Test目录下包含了一些Java实现的测试用例。

在Src目录下就是HotSpot项目的主体源代码，由Cpu、Os、Os_cpu和Share这4个子目录组成。在Cpu目录下是一些依赖具体处理器架构的代码，主要按照Sparc、x86和Zero三种计算机体系结构划分子模块；Os目录下则是一些依赖操作系统的代码，主要按照Linux、Windows、Solaris和Posix¹⁰进行模块划分；而Os_cpu目录下则是一些同时依赖于操作系统和处理器类型的代码，如Linux+Sparc、Linux+x86、Linux+Zero、Solaris+Sparc、Solaris+x86和Windows+x86等模块。

在Share目录下是独立于操作系统和处理器类型的代码，这部分代码是HotSpot工程的核心业务，实现了HotSpot的主要功能。Share由两部分组成，一部分是实现虚拟机各项功能的Vm目录。另一部分是位于Tools目录下的几个独立的虚拟机工具程序，如Hsdis、IdealGraphVisualizer、Launcher、LogCompilation和ProjectCreator。

在Vm目录下，按照虚拟机的功能划分了一些模块。这些模块构成了虚拟机的内核，它们是HotSpot内核的顶层模块，每个顶层模块封装了在功能上相对独立的业务逻辑。目前，HotSpot中主要包括了下列顶层模块。

- Adlc：平台描述文件。
- Libadt：抽象数据结构。
- Asm：汇编器。
- Code：机器码生成。

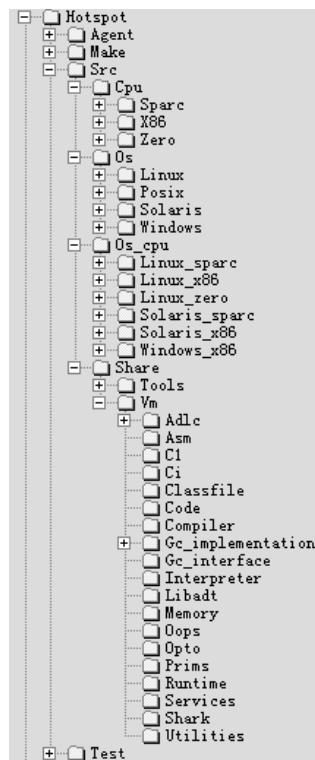


图1-4 HotSpot工程结构

⁹ 更多内容可以参考<http://openjdk.java.net/groups/hotspot/docs/Serviceability.html>。

¹⁰ 可移植操作系统接口（Portable Operating System Interface，缩写为Posix），是IEEE为在跨UNIX操作系统的应用程序的可移植保，而定义的一系列的API和标准的总称。Linux虽然没有参加正式的Posix认证，但事实上基本与Posix保持兼容。更多信息，可以参考<http://en.wikipedia.org/wiki/POSIX>。

- C1: client 编译器, 即 C1 编译器。
- Ci: 动态编译器。
- Compiler: 调用动态编译器的接口。
- Opto: Server 编译器, 即 C2 编译器。
- Shark: 基于 LLVM 实现的即时编译器。
- Interpreter: 解释器。
- Classfile: Class 文件解析和类的链接等。
- Gc_interface: GC 接口。
- Gc_implementation: 垃圾收集器的具体实现。
- Memory: 内存管理。
- Oops: JVM 内部对象表示。
- Prims: HotSpot 对外接口。
- Runtime: 运行时。
- Services: JMX 接口。
- Utilizes: 内部工具类和公共函数。

在下文中, 我们将分别对这些模块展开探讨。第 2 章不仅介绍了 Launcher 作为 JVM 的启动器是如何启动虚拟机并进行系统初始化的, 还介绍了 Prims、Services 和 Runtime 等公共模块为虚拟机提供的重要基础作用: Prims 为外界与 JVM 搭建了通信的桥梁, Services 和 Runtime 为其他模块提供了公共服务。第 3 章将会讨论 Oops 和 Classfile 模块, 前者构成了 HotSpot 内部的面向对象表示系统, 而后者则提供了类的解析功能; 第 4 章会介绍 Memory 模块提供内存管理功能; 第 5 章深入介绍了与垃圾收集器相关的 GC 模块; 第 7 章详细讨论了与解释器和编译器的实现息息相关的 Interpreter、C1、Code 等模块。

1.2.3 搭建编译环境

在各种 Linux 发行版中, Ubuntu 算是比较普及的一款产品。它具有功能丰富、更新速度快和容易上手的优良特性, 鉴于此, 我们选择 Ubuntu 12 作为开发环境。当然, 也可以选择官方推荐的其他 Linux 发行版, 如 Fedora、Debian 等, 这完全没有任何限制。

我们在这里搭建的环境如下。

- 源代码版本: OpenJDK7, 分支代号 b147。
- 操作系统: 基于 Linux Kernel 3.5 内核的 Ubuntu 12.10 (Vmware Workstation 9.0¹¹)

¹¹ 在 PC 上搭建 Ubuntu 环境有两种常见方式:一种是在 Windows 上基于 Vmware Workstation 虚拟机启动 Ubuntu; 另一种是安装双系统, 如果选择这种方式, 推荐使用 WUBI 工具实现一键式安装 Ubuntu。这两种启动 Ubuntu 的方式是等价的, 读者可以自由选择, 笔者这里使用的是前者。

发行版。

- 编译环境: GCC 4.7、G++ 4.6 和 GDB 7.5。

为方便搭建编译环境,读者可以在 Hotspot 目录下创建一个编译脚本,来节省许多手工配置工作。脚本内容如清单 1-9 所示(读者请将路径替换为本地路径)。

清单 1-9

描述: HotSpot 工程编译脚本

```
#!/bin/bash
export LANG=C
```

导入 JDK 路径:

```
export ALT_BOOTDIR="/home/chentao/mywork/soft/jdk1.6.0_35"
export ALT_JDK_IMPORT_PATH="/home/chentao/mywork/soft/jdk1.6.0_35"
```

导入 ANT 路径:

```
export ANT_HOME="/home/chentao/mywork/soft/apache-ant-1.8.4"
```

导入 PATH:

```
export
PATH="/usr/lib:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/home/chentao/mywork/soft/apache-ant-1.8.4:/usr/lib/i386-linux-gnu:/usr/lib/gcc/i686-linux-gnu/4.6"
```

其他配置:

```
export HOTSPOT_BUILD_JOBS=5
#输出目录
export ALT_OUTPUTDIR=../build/hotspot_debug
```

选择目标版本为 jvmg, 启动编译 HotSpot 命令:

```
cd make
make jvmg jvmg1 2>&1 | tee ~/hotspot-in-action/hotspot/build/hotspot_debug.log
```

编译完成后,在日志文件 hotspot_debug.log 中可以查看编译过程。阅读这份日志,有助于加深对 HotSpot 项目整体架构的理解。

1.2.4 编译目标

如清单 1-10 所示,在 Makefile 中定义了 HotSpot 项目的编译目标和级别。其中,主要包括以下 4 种基本级别。

- product: 产品级别。优化编译,但无断言。
- fastdebug: 快速调试级别。优化编译,但开启断言。
- optimized: 优化级别。优化编译,但无断言。
- debug: 调试级别。编译后的 libjvm 链接库中含有较丰富的调试信息。

清单 1-10

来源: hotspot/make/Makefile

描述: 编译目标

```

1 C1_VM_TARGETS=product1 fastdebug1 optimized1 jvmg1
2 C2_VM_TARGETS=product fastdebug optimized jvmg
3 KERNEL_VM_TARGETS=productkernel fastdebugkernel optimizedkernel jvmgkernel
4 ZERO_VM_TARGETS=productzero fastdebugzero optimizedzero jvmgzero
5 SHARK_VM_TARGETS=productshark fastdebugshark optimizedshark jvmgshark

6 all:           all_product all_fastdebug
7 ifndef BUILD_CLIENT_ONLY
8 all_product:   product product1 productkernel docs export_product
9 all_fastdebug: fastdebug fastdebug1 fastdebugkernel docs export_fastdebug
10 all_debug:    jvmg jvmg1 jvmgkernel docs export_debug
11 else
12 all_product:  product1 docs export_product
13 all_fastdebug: fastdebug1 docs export_fastdebug
14 all_debug:    jvmg1 docs export_debug
15 endif
16 all_optimized: optimized optimized1 optimizedkernel docs export_optimized

17 allzero:        all_productzero all_fastdebugzero
18 all_productzero: productzero docs export_product
19 all_fastdebugzero: fastdebugzero docs export_fastdebug
20 all_debugzero:   jvmgzero docs export_debug
21 all_optimizedzero: optimizedzero docs export_optimized

22 allshark:       all_productshark all_fastdebugshark
23 all_productshark: productshark docs export_product
24 all_fastdebugshark: fastdebugshark docs export_fastdebug
25 all_debugshark:  jvmgshark docs export_debug
26 all_optimizedshark: optimizedshark docs export_optimized

```

在清单 1-9 中, 我们在 make 命令后传递参数 “jvmg jvmg1”, 表示选择编译 debug 级别的目标。这样待编译成功后, 生成的 libjvm 库 (HotSpot VM 运行时库) 中会包含丰富的调试信息, 通过这些信息, 调试器可以建立虚拟机运行时与源代码间的关联, 为单步调试 HotSpot 做好准备。

1.2.5 编译过程

执行清单 1-9 所示的编译脚本后, 就可以启动 HotSpot 编译过程。如果一切顺利, 待编译过程结束后, 将在 Hotspot 目录下创建一个 Build 目录。Build 目录是整个编译过程的工作空间, 该目录下包含了最终的编译目标 (参见清单 1-10)。打开 Build 目录, 可以见到一些新创建的目录, 如清单 1-11 所示。

清单 1-11

```

unix> cd build/
unix> ls
hotspot_debug linux
unix> cd hotspot_debug/
unix> ls
linux_i486_compiler1 linux_i486_compiler2
unix> cd linux_i486_compiler1

```

```
unix> ls
debug fastdebug generated jvmg optimized product profiled shared_dirs.lst
```

编译结束后，执行 jvmg 目录下可执行文件 test_gamma，便可以检验整个编译过程是否成功。执行 test_gamma 后，如果能够在控制台看到类似图 1-5 所示的输出信息，就表示编译成功了。

```
java full version "1.7.0_09-b05"
Using java runtime at: /usr/lib/jvm/java-7-sun/jre
java version "1.7.0_09"
Java(TM) SE Runtime Environment (build 1.7.0_09-b05)
Java HotSpot(TM) Client VM (build 23.5-b02, 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
7. A2 B5 C7 D4 E1 F8 G6 H3
8. A2 B6 C1 D7 E4 F8 G3 H5
9. A2 B6 C8 D3 E1 F4 G7 H5
10. A2 B7 C3 D6 E8 F5 G1 H4
11. A2 B7 C5 D8 E1 F4 G6 H3
12. A2 B8 C6 D1 E3 F5 G7 H4
13. A3 B1 C7 D5 E8 F2 G4 H6
14. A3 B5 C2 D8 E1 F7 G4 H6
15. A3 B5 C2 D8 E6 F4 G7 H1
16. A3 B5 C7 D1 E4 F2 G8 H6
17. A3 B5 C8 D4 E1 F7 G2 H6
18. A3 B6 C2 D5 E8 F1 G7 H4
```

图 1-5 test_gamma 执行成功

实际上，test_gamma 也是一个脚本，其内容如清单 1-12 所示。

```
清单 1-12
来源: test_gamma
描述: 测试脚本

1 #!/bin/sh
2 # Generated by /home/chentao/hotspot-in-action/hotspot/make/linux/makefiles/ buildtree.
make
3 ./env.sh
4 if [ "" != "" ]; then { echo Cross compiling for ARCH , skipping gamma run.; exit 0; }
fi
5 if [ -z $JAVA_HOME ]; then { echo JAVA_HOME must be set to run this test.; exit 0; }
fi
6 if ! ${JAVA_HOME}/bin/java -d32 -fullversion 2>&1 > /dev/null
7 then
8   echo JAVA_HOME must point to 32bit JDK.; exit 0;
9 fi
10 rm -f Queens.class
11 ${JAVA_HOME}/bin/javac -d . /home/chentao/hotspot-in-action/hotspot/make/test/ Queens.java
12 [ -f gamma_g ] && { gamma=gamma_g; }
13 ./${gamma:-gamma} -Xbatch -showversion Queens < /dev/null
```

在第 3 行中，执行 env.sh 准备执行环境。在第 10 行中，编译测试程序 Queens.java。在第 12 和第 13 行中，最终是利用调试版启动器 gamma，来启动测试程序 Queens。Queens 是一个

求解 N 皇后问题的 Java 程序，图 1-5 便是运行 Queens 程序输出的结果。

1.2.6 编译常见问题

编译过程中可能会遇到一些问题，下面列出几个常见错误及其解决办法，供读者参考。

1. 内核版本支持

在我们下载的 HotSpot 源代码中，默认支持的 Linux 内核最高版本为 2.6，而我们所用的发行版很有可能采用了高于此版本的 Linux 内核。例如，笔者所用的 Ubuntu12 的内核是 3.5（可通过 `uname -r` 命令查看自己内核版本）。如果不进行一些调整的话，编译 HotSpot 时可能会遇到如下报错：

```
**** This OS is not supported:" 'uname -a'; exit 1;
```

如果遇到这个问题，可以在这个文件中找到解决办法：`hotspot/make/linux/Makefile`。在 `Makefile` 文件中，定位到包含字符串“`SUPPORTED_OS_VERSION`”的代码，并在该行末尾增加“`3.5%`”，这样就可以使 HotSpot 支持我们实际使用的内核版本，调整后的代码如下：

```
SUPPORTED_OS_VERSION = 2.4% 2.5% 2.6% 2.7% 3.5%
```

另一种调整方法是绕过验证操作系统版本的步骤。如清单 1-13 所示的定位到包含字符串“`check_os_version`”的代码，将其删除或者注释掉便可。

清单 1-13

来源：`hotspot/make/linux/Makefile`

描述：验证 OS 版本

```
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
```

2. 头文件的宏定义冲突的问题

`cdefs.h` 中定义的宏“`_LEAF`”与 `interfaceSupport.hpp` 冲突。可以在 `interfaceSupport.hpp` 中增加一个“`#undef _LEAF`”语句来解决冲突，具体代码如清单 1-14 所示。

清单 1-14

来源：`hotspot/src/share/vm/runtime/interfaceSupport.hpp`

描述：预定义宏`_LEAF`

```
// LEAF routines do not lock, GC or throw exceptions  
#ifdef __LEAF  
#undef __LEAF  
#define __LEAF(result_type, header)  
    TRACE_CALL(result_type, header)  
    debug_only(NoHandleMark __hm;)  
    /* begin of body */  
#endif
```

3. GCC 版本过高导致的问题

有时，编译器的版本也可能引起编译失败。例如，清单 1-15 描述了一个 GCC 版本过高引起的问题。

清单 1-15

```
Linking vm...
/usr/bin/ld: cannot find -lstdc++
collect2: error: ld returned 1 exit status
ln: failed to create symbolic link 'libjvm_g.so': File exists
ln: failed to create symbolic link 'libjvm_g.so.1': File exists
```

GCC 链接工具 `ld` 返回的错误信息显示：无法找到“`-lstdc++`”这个链接选项。这是由于 GCC 版本过高不支持“`lstdc++`”选项导致的错误。解决办法是把 `Makefile` 中的“`lstdc++`”选项去掉并重新尝试编译。

由于开发环境各不相同，每个人遇到的问题可能都不尽相同；即使遇到相同的问题，在不同的平台上解决的方式可能也有所不同。当然，对于相同的问题，也会有多种办法解决。限于篇幅，在这里不能对所有错误信息和解决办法都列举出来。具体问题应当具体对待，但绝大多数问题的解决思路是一样的：首先根据错误码或报错信息确认错误来自于哪个组件（工具程序或库），根据自身经验判断错误的原因，或者查找错误来源组件的官方资料定位错误的原因。只要原因得到确认，距离解决办法也就不远了。当然，最重要的是遇到问题时能够保持从容和耐心。

1.3 实战：在 HotSpot 内调试 HelloWorld

本节讲解的是 Java 入门程序 `HelloWorld` 在 HotSpot 上的执行过程。我们通过一个普通 Java 程序的运行过程，能够以点带面地讲解到涉及 HotSpot 内部实现的基础概念。

虽然是调试简单的 `HelloWorld` 程序，但在这个过程中会涉及 HotSpot 的基本数据结构以及环境准备等内容。理解这些，一方面使读者对 HotSpot 项目有个感性认识，其实调试 HotSpot 没有想象的那么困难，这利于我们增强驾驭 HotSpot 的自信心；另一方面，让我们正式接触到 HotSpot 的基本代码，并掌握 HotSpot 项目的基本调试方法。

调试准备过程如图 1-6 所示，具体步骤如下。

- (1) 选择调试器。
- (2) 配置 GDB 工作目录的绝对路径。
- (3) 配置 JDK 和动态链接库路径。
- (4) 定位 Launcher。
- (5) 运行 GDB 初始化脚本，准备 GDB 运行环境。
- (6) 设置 HotSpot 项目断点。

- (7) 启动调试脚本。
- (8) 虚拟机运行 HelloWorld 程序，在断点处暂停。
- (9) 利用 GDB 命令调试 HotSpot 虚拟机程序的运行。

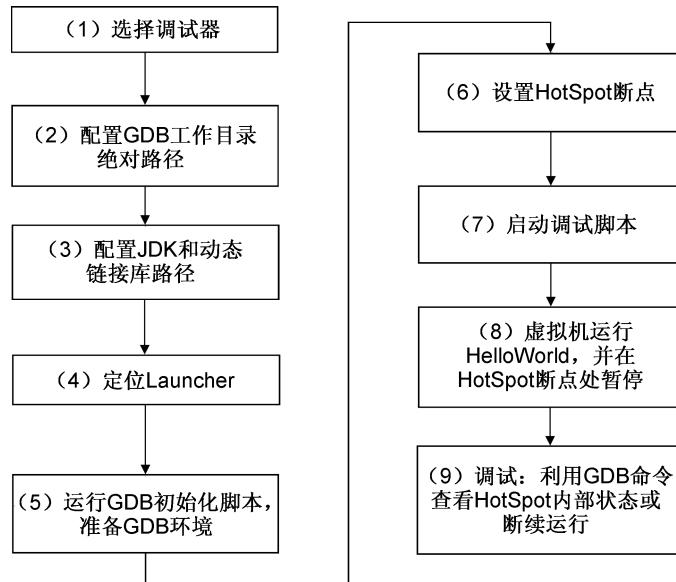


图 1-6 调试准备过程

接下来，我们先了解一下如何使用 GDB 调试程序，然后开启我们的调试之旅。

1.3.1 认识 GDB

本地程序（C/C++）的调试，一般使用 GDB 命令。对于 Java 程序员来说，GDB 有些陌生，其实我们只需要掌握一些基本的调试命令，便足够应付 HotSpot 的调试任务了。

下面附上一些常用的 GDB 命令，包括断点、执行、查看代码、查看栈帧、查看数据等，如清单 1-16 所示。

清单 1-16

```

断点:
break InitializeJVM: 在 InitializeJVM 函数入口处设置断点
break java.c:JavaMain: 在源文件 java.c 的 InitializeJVM 函数入口处设置断点
break os_linux.cpp:4380: 在源文件 os_linux.cpp 的第 4380 行处设置断点
break *0x8048000: 在地址为 0x8048000 的地址处设置断点
delete 1: 删除断点 1
delete: 删除所有断点

执行:
step: 执行 1 条语句，会进入函数
step n: 执行 n 条语句，会进入函数
next: 与 step 类似，但是不进入函数

```

```

next n: 与 step n 类似，但是不进入函数
continue: 继续运行
finish: 运行至当前函数返回后退出
查看代码:
list n: 查看当前源文件中第 n 行的代码
list InitializeJVM: 查看 InitializeJVM 函数开始位置的代码
list: 查看更多的行
list -: 查看上次查看的代码行数之前的代码
默认, GDB 打印 10 行。若需要调整, 可使用:
set listsize n: 调整打印行数为 n 行
查看栈帧:
frame n: 从当前栈帧移动到 #n 栈帧
up n: 从当前栈帧向上移动 n 个栈帧
down n: 从当前栈帧向下移动 n 个栈帧
select-frame: 查看更多的行
backtrace: 查看整个调用栈
backtrace n: 与 backtrace 类似, 只不过只查看 4 个栈帧
backtrace full: 查看整个调用栈, 另外还打印出局部变量和参数
info args: 查看函数参数
info locals: 查看局部变量
查看数据:
print expr: 查看 expr 的值, 其中 expr 是源文件中的表达式
print /f expr n: 以 f 指定的格式查看 expr 的值。其中 f 表示的格式可以为:
x: 十六进制整数
d: 有符号整数
u: 无符号整数
o: 八进制整数
t: 二进制整数
c: 字符常量
f: 浮点数
s: 字符串
r: 原始格式
a: 地址
x 0xbffffd034: 查看内存地址为 0xbffffd034 的值
disassemble: 查看汇编代码, 反汇编当前函数
info registers: 查看所有寄存器的值
print $eax: 以十进制形式查看寄存器 %eax 的值
print /x $eax: 以十六进制形式查看寄存器 %eax 的值

```

更多 GDB 的信息, 可以参考 GDB 的官方教程¹²。

1.3.2 准备调试脚本

在 HotSpot 编译完成后, 会在 Jvmg 目录下生成一个名为 hotspot 的脚本文件, 如清单 1-17 所示。使用脚本可以替代大量重复性的输入, 并且可以帮助我们准备好调试环境, 为我们轻松调试系统创造了良好的环境。我们可以在此脚本文件的基础上调试 HotSpot 项目。

在启动调试之前, 了解调试脚本究竟做了哪些工作是十分有益的, 这有助于我们掌握独立分析和解决问题的能力, 在出现问题时不至于手忙脚乱, 可以利用自身所学知识解决问题。

清单 1-17

来源: hotspot/src/os/posix/launcher/launcher.script

描述: 调试脚本

```
#!/bin/bash
```

¹² 更多内容可以参考 <http://www.gnu.org/software/gdb/documentation/>。

首先是对传入的调试器名称参数进行转换，以便于定位到指定的调试器，支持的调试器包括 GDB、GUD、DBX 和 VALGRIND 等。

```
# 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
```

用户可以通过调用该脚本时传入参数选择熟悉的调试器，这些参数可以是“-gdb”、“-gud”、“-dbx”或“-valgrind”。

```
# Make sure the paths are fully specified, i.e. they must begin with /.
SCRIPT=$(cd ${dirname $0} && pwd)/${basename $0}
RUNDIR=$(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
```

`\${MYDIR}`是配置脚本的绝对路径：

```
# Find out the absolute path to this script
```

```
MYDIR=$(cd $(dirname $SCRIPT) && pwd)
```

`${JDK}`用来配置 JDK 路径，此外，还有一些链接库路径需要配置：

```
JDK=
if [ "${ALT_JAVA_HOME}" = "" ]; then
    source ${MYDIR}/jdkpath.sh
else
    JDK=${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
ARCH=i386

# Find out the absolute path to this script
MYDIR=$(cd $(dirname $SCRIPT) && pwd)

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

# Set up a suitable LD_LIBRARY_PATH

if [ -z "$LD_LIBRARY_PATH" ]
then
    LD_LIBRARY_PATH="$SBP"
else
    LD_LIBRARY_PATH="$SBP:$LD_LIBRARY_PATH"
fi

export LD_LIBRARY_PATH
export JAVA_HOME

JPARAMS="$@ $JAVA_ARGS";
```

`${LAUNCHER}`用作定位 Launcher。关于 Launcher，我们会在下一章中展开探讨。这里只需要知道它是虚拟机启动器程序便可：

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

接下来是进行 GDB 自身初始化工作，包括配置工作路径以及信号等工作：

```
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 $GDBSRCDIR

```

在这里，可以设置断点。选择你感兴趣的 HotSpot 项目源代码位置，如 JVM 初始化模块“InitializeJVM”函数入口。接下来，便可以利用 GDB 的 break 命令设置断点，如：

```

# Get us to a point where we can set breakpoints in libjvm.so
break InitializeJVM
run
# Stop in InitializeJVM
delete 1
# We can now set breakpoints wherever we like
EOF
}

```

剩余配置代码我们可以不做调整：

```

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
$JPARAMS
;;

```

```

run)
LD_PRELOAD=$PRELOADING exec $LAUNCHER $JPARMS
;;
*)
echo Error: Internal error, unknown launch mode \"$MODE\"
exit 1
;;
esac
RETVAL=$?
exit $RETVAL

```

至此，调试脚本已经准备就绪，接下来，让我们开始 HotSpot 的调试吧。输入命令：

```
sh hotspot -gdb HelloWorld
```

启动调试，将出现如图 1-7 所示的界面。

```

chentao@ubuntu: ~/hotspot-in-action/hotspot/build/hotspot_debug/linux_i486_gdb
Breakpoint 1 at 0x804bb50: file /home/chentao/hotspot-in-action/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".
----_JAVA_LAUNCHER_DEBUG----
JRE path is /usr/lib/jvm/java-7-sun/jre
Does '/home/chentao/hotspot-in-action/hotspot/build/hotspot_debug/linux_i486_compiler1/jvmg/libjvm.so' exist ... yes.
Using java runtime at: /usr/lib/jvm/java-7-sun/jre
1 micro seconds to LoadJavaVM
[New Thread 0xb6f35b40 (LWP 5632)]
[Switching to Thread 0xb6f35b40 (LWP 5632)]

Breakpoint 1, InitializeJVM (pvm=0xb6f352b4, penv=0xb6f352b8,
    ifn=0xb6f35318)
    at /home/chentao/hotspot-in-action/hotspot/src/share/tools/launcher/java.c:1270
1270      memset(&args, 0, sizeof(args));
(gdb) 

```

图 1-7 调试 HelloWorld

HotSpot 运行在断点 1 (InitializeJVM) 上停止下来，这时就可以利用前面提到的 GDB 命令尽情地控制 HotSpot 的运行了！

如果想让程序继续执行，输入 `continue` 命令使虚拟机正常运行下去，可以看到程序输出“Hello hotspot”并正常退出。感兴趣的读者可以亲自动手尝试一下。

建议读者结合源代码，利用 GDB 命令来跟踪调试 HotSpot，查看系统运行时的内部数据和状态。这有两个好处：一方面，这能帮助我们将枯燥的阅读源码任务转换成有趣的虚拟机调试工作；另一方面，也能促进我们加深对 HotSpot 的理解。

1.4 小结

在本章开头，回顾了 JDK 的发展历程。接着，我们看到了 Java 7 带来的一些语法变化。

这些变化并没有让 JVM 在底层做出较大改动，而是通过“语法糖”的包装形式实现。OpenJDK 项目成为了 Java 官方主打的开源项目。JCP 为推动 Java 特性的发展发挥着重要的作用。

OpenJDK 的开源对于促进 Java 爱好者与从业者深入研究 JVM，具有难以估量的价值。要想系统了解 JVM 底层实现原理，亲自动手编译和调试 JVM 无疑是最为有效的一条途径。为帮助读者更快接触 HotSpot，本章通过一个实战案例讲解了编译和调试 HotSpot 的基本技术。调试 HotSpot 虽然是一项较为精细的任务，但好在还是有法可循的，只要读者掌握这些基本技能，摒除浮躁，是完全可以深入掌握 HotSpot 的。

现在，我们已经打好了继续研究 JVM 的技术基础，这有利于将枯燥的阅读源码任务转换成一项具有操作性和趣味性的实践工作。从下一章开始，我们将深入 HotSpot 内核，在阐述底层运作原理的同时也会引入一些对实际工作有益的实用技巧。

第 2 章 启动

“物有本末，事有终始。知其先后，则近道矣。”

——《大学》

本章内容

- HotSpot 内核模块
- 启动器 Launcher 和启动过程
- JVM 初始化过程
- 全局模块初始化

本章是 HotSpot 内核的入门导读。首先介绍阅读源代码的方式，接下来讲解了 HotSpot 内核模块组成和功能框架，最后重点讲解了 JVM 的启动和初始化过程。

2.1 HotSpot 内核

在引入 HotSpot 内核模块之前，我们有必要掌握一些阅读源代码的技巧。

2.1.1 如何阅读源代码

我们知道，HotSpot 项目主要是由 C++语言开发的，对于 Java 程序员来说，直接阅读这部分源代码可能会有些吃力。因此，我们有必要先阐释一些语言上的差异，扫清这些学习障碍。

1. 宏

实际上，Java 语言在语法上与 C 和 C++ 是颇为相似的。除了一些在 Java 中没有提供的语法和特性，大多数 C/C++ 代码还是很容易被 Java 程序员理解的。在这里，我们首先对在 C 和 C++ 中大量使用的“宏”做一个简单介绍。

宏是一个较为简单的概念，在编译 C/C++ 代码前，预处理器将宏代码展开，这样编译出来的代码在运行时就省去了一些额外的空间和时间开销。因此，使用宏可以在不影响程序运行性能的前提下提高代码的可读性。HotSpot 项目中，在许多情形下都使用了宏。

(1) 功能相似的数据结构。

在 HotSpot 中出现的所有内部对象，都需要一个类型进行定义。HotSpot 为了能够更好地管理内部的对象表示，设计了一套面向对象表示系统，即 OOP-Klass 二分模型。在第 3 章中，我们将继续深入认识这一系统。简单来说，在这个模型中需要定义许多内部类型。考虑到很多类型具有相似的定义方式，出于模块化和可扩展性的设计思维，HotSpot 的设计者运用宏巧妙地实现了对各种相似类型的通用定义。如清单 2-1 所示，在定义 OOP 结构类型时，设计了 DEF_OOP 宏，它能根据传入的 type 参数定义出不同的 oopDesc 类型。

清单 2-1

来源：hotspot/src/share/vm/oops/oopsHierarchy.hpp
描述：定义<type>oopDesc 类型

```

1 #define DEF_OOP(type)
2     class type##OopDesc;
3     class type##Oop : public oop {
4         public:
5             type##Oop() : oop() {}
6             type##Oop(const volatile oop& o) : oop(o) {}
7             type##Oop(const void* p) : oop(p) {}
8             operator type##OopDesc* () const { return (type##OopDesc*)obj(); }
9             type##OopDesc* operator->() const {
10                 return (type##OopDesc*)obj();
11             }
12     };
13 }
```

在 DEF_OOP 宏的定义中，预处理器根据外部传入的参数 type，将宏分别展开为不同的代码块。符号“##”表示将实际传入的“type”值与“##”右边的字符串连接在一起。显然，在调用 DEF_OOP 宏时，如果传递给它不同的“type”值，那么在展开的代码块中，最终定义的类型将肯定不同。当定义好 DEF_OOP 后，我们每次调用 DEF_OOP 宏，就完成了一个 oopDesc 类型的定义，如清单 2-2 所示。

清单 2-2

来源：hotspot/src/share/vm/oops/oopsHierarchy.hpp
描述：定义<type>oopDesc 类型

```

1 DEF_OOP(instance);
2 DEF_OOP(method);
3 DEF_OOP(methodData);
4 DEF_OOP(array);
5 DEF_OOP(constMethod);
```

```

6 DEF_OOP(constantPool);
7 DEF_OOP(constantPoolCache);
8 DEF_OOP(objArray);
9 DEF_OOP(typeArray);
10 DEF_OOP(klass);
11 DEF_OOP(compiledICHolder);

```

在清单 2-2 中，定义了 11 种相似的 oopDesc 子类型，如 instanceOopDesc、methodOopDesc 和 methodDataOopDesc 等。在定义句柄时也使用了相同的方式，如清单 2-3 所示。

清单 2-3

来源：hotspot/src/share/vm/runtime/handles.hpp

描述：定义句柄

```

1 #define DEF_HANDLE(type, is_a) \
2 class type##Handle; \
3 class type##Handle: public Handle { \
4 protected: \
5     type##Oop    obj() const { return (type##Handle::obj()); } \
6     type##Oop    non_null_obj() const { return (type##Handle::non_null_ \
7 obj()); } \
8 public: \
9     /* 构造函数 */ \
10    type##Handle () : Handle() {} \
11    type##Handle (type##Oop obj) : Handle((oop)obj) { \
12        assert(SharedSkipVerify || is_null() || ((oop)obj)->is_a(), \
13               "illegal type"); \
14    } \
15    type##Handle (Thread* thread, type##Oop obj) : Handle(thread, (oop)obj) { \
16        assert(SharedSkipVerify || is_null() || ((oop)obj)->is_a(), "illegal type"); \
17    } \
18 \
19    type##Handle (type##Oop *handle, bool dummy) : Handle((oop*)handle, dummy) {} \
20 \
21    /* 操作符重载 */ \
22    type##Oop    operator () () const { return obj(); } \
23    type##Oop    operator -> () const { return non_null_obj(); } \
24 };

```

在 DEF_HANDLE 宏的定义中，根据外部传入的 type 和 is_a 等参数，可以定义出不同的 handle 类型，如清单 2-4 所示。

清单 2-4

来源：hotspot/src/share/vm/oops/oopsHierarchy.hpp

描述：定义<type>Handle 类型局部

```

1 DEF_HANDLE(instance      , is_instance      )
2 DEF_HANDLE(method       , is_method       )
3 DEF_HANDLE(constMethod  , is_constMethod  )
4 DEF_HANDLE(methodData   , is_methodData   )
5 DEF_HANDLE(array        , is_array        )
6 DEF_HANDLE(constantPool , is_constantPool)
7 DEF_HANDLE(constantPoolCache, is_constantPoolCache)
8 DEF_HANDLE(objArray     , is_objArray     )
9 DEF_HANDLE(typeArray    , is_typeArray    )

```

(2) 函数定义。

在定义一组具有相似结构的函数时，如以“jmm_”或“JVM_”为前缀命名的虚拟机接口

函数。在这些函数中，都需要做一些相似的工作。如果为每个函数都重复编写相同的代码去做同一件事，这显然不利于维护和扩展。因此，HotSpot 中将那些事务性的工作，如处理接口返回值类型、JNI 声明、获取当前线程和调试开关的代码提取出来，由 JVM_ENTRY 宏来代替。这样依赖，我们在 HotSpot 项目中就看到了很多利用 JVM_ENTRY 宏定义的接口函数，如 jmm_GetMemoryUsage() 函数，如清单 2-5 所示。

清单 2-5

来源: hotspot/src/share/vm/services/management.cpp
描述: GetMemoryUsage 的底层实现

```
1 // Returns a java/lang/management/MemoryUsage object representing
2 // the memory usage for the heap or non-heap memory.
3 JVM_ENTRY(jobject, jmm_GetMemoryUsage(JNIEnv* env, jboolean heap))
4     ResourceMark rm(THREAD);

5     // Calculate the memory usage
6     size_t total_init = 0;
7     size_t total_used = 0;
8     size_t total_committed = 0;
9     size_t total_max = 0;
10    bool has_undefined_init_size = false;
11    bool has_undefined_max_size = false;

12   for (int i = 0; i < MemoryService::num_memory_pools(); i++) {
13       MemoryPool* pool = MemoryService::get_memory_pool(i);
14       if ((heap && pool->is_heap()) || (!heap && pool->is_non_heap())) {
15           MemoryUsage u = pool->get_memory_usage();
16           total_used += u.used();
17           total_committed += u.committed();

18           // if any one of the memory pool has undefined init_size or max_size,
19           // set it to -1
20           if (u.init_size() == (size_t)-1) {
21               has_undefined_init_size = true;
22           }
23           if (!has_undefined_init_size) {
24               total_init += u.init_size();
25           }

26           if (u.max_size() == (size_t)-1) {
27               has_undefined_max_size = true;
28           }
29           if (!has_undefined_max_size) {
30               total_max += u.max_size();
31           }
32       }
33   }

34   // In our current implementation, we make sure that all non-heap
35   // pools have defined init and max sizes. Heap pools do not matter,
36   // as we never use total_init and total_max for them.
37   assert(heap || !has_undefined_init_size, "Undefined init size");
38   assert(heap || !has_undefined_max_size, "Undefined max size");

39   MemoryUsage usage((heap ? InitialHeapSize : total_init),
40                     total_used,
41                     total_committed,
```

```

42             (heap ? Universe::heap()->max_capacity() : total_max));
43     Handle obj = MemoryService::create_MemoryUsage_obj(usage, CHECK_NULL);
44     return JNIHandles::make_local(env, obj());
45 JVM_END

```

与 `JVM_ENTRY` 相伴生的宏是 `JVM_END`，相当于语句“`}}`”的作用，表示函数定义的结束。

(3) 共同基类。

如果有大量的类都继承自同一类型，可以将继承基类的语句抽象成一个宏。

在 HotSpot 内部定义一个继承自 `_ValueObj`（表示对象类型为值）的类时，就使用了宏 `VALUE_OBJ_CLASS_SPEC`，如清单 2-6 所示。

清单 2-6

来源：hotspot/src/share/vm/runtime/thread.cpp - Threads::threads_do()

描述：迭代线程列表

```
#define VALUE_OBJ_CLASS_SPEC : public _ValueObj
```

当定义一个本身是值类型的新类型时，可以这样做，如清单 2-7 所示。

清单 2-7

来源：hotspot/src/share/vm/runtime/thread.cpp - Threads::threads_do()

描述：迭代线程列表

```
class frame VALUE_OBJ_CLASS_SPEC { ... }
```

(4) 循环条件。

在线程模块中，常常需要迭代活跃线程列表，那么可以将遍历 `Threads` 静态成员 `_thread_list` 的操作提取出来，定义成一个名为 `ALL_JAVA_THREADS` 的宏，如清单 2-8 所示。

清单 2-8

来源：hotspot/src/share/vm/runtime/thread.cpp

描述：遍历线程列表

```
#define ALL_JAVA_THREADS(X) for (JavaThread* X = _thread_list; X; X = X->next())
```

这样，在使用时，只需要调用 `ALL_JAVA_THREADS(X)`，接下来的循环体就可以直接使用 `X` 作为线程元素进行业务处理了，而不必关心循环的创建和终止，如清单 2-9 所示。

清单 2-9

来源：hotspot/src/share/vm/runtime/thread.cpp - Threads::threads_do()

描述：迭代线程列表

```

1  ALL_JAVA_THREADS(p) {
2      tc->do_thread(p);
3  }
```

(5) 调试技术。

这种类型的宏常用于内部调试逻辑，如在关键代码路径处判断程序状态是否出错，决定是否输出一些错误信息或退出程序，如清单 2-10 所示。

清单 2-10

来源：hotspot/src/share/vm/utilities/debug.hpp

描述：调试宏

```
1 #define ShouldNotReachHere() \
```

```

2   do {
3     report_should_not_reach_here(FILE__, LINE__);
4     BREAKPOINT;
5   } while (0)

```

在 JVM 内部流程中，当流程走到本不该走的地方时，调用 ShouldNotReachHere() 进行错误处理。

2. 内联函数

内联函数用于消除函数调用和返回时的寄存器存储和恢复开销，它通常应用于频繁执行的函数中。由于函数调用时，需要将程序执行流程转移到被调用函数中，并要求函数返回时回到原先的执行流程继续执行。这就要求在调用时保存现场并记住执行的地址，以待函数返回后恢复并按原程序流程继续执行。因此，函数调用会带来一定的时间和空间方面的开销，影响效率。而使用内联函数，编译器可以实现在函数调用时将内联函数展开，这样就取消了函数的入栈和出栈工作，减少了程序的开销。当函数被频繁调用的时候，节省下来的开销就十分可观了。

通常，将那些时间要求比较高，而本身长度比较短的函数定义成内联函数。例如，在定义 OOP 结构类型的成员函数时，可以采用如清单 2-11 所示的做法。

清单 2-11

来源：hotspot/src/share/vm/oops/oop.inline.hpp

描述：对象头内联函数——设置 mark word

```

1 inline markOop oopDesc::cas_set_mark(markOop new_mark, markOop old_mark) {
2   return (markOop) Atomic::cmpxchg_ptr(new_mark, &_mark, old_mark);
3 }

```

在第 1 行代码中，关键字 `inline` 表明 `oopDesc::cas_set_mark()` 是一个内联函数。

3. 内部锁

在 JVM 内部，有些操作在同一时刻，只允许一个线程执行，因此需要互斥锁来保证系统的绝对安全。例如，对诸如 SystemDictionary（系统字典）、Safepoint（安全点）或 Heap（堆）等功能实体进行操作时，需要先取得相应的锁。表 2-1 定义了这些内部锁。

表 2-1

HotSpot 内部使用的锁

锁	锁
SystemDictionary_lock	iCMS_lock
InlineCacheBuffer_lock	FullGCCount_lock
VMStatistic_lock	SATB_Q_FL_lock
JmethodIdCreation_lock	SATB_Q_CBL_mon
JfieldIdCreation_lock	Compile_lock
Heap_lock	MethodCompileQueue_lock
VtableStubs_lock	CompileThread_lock
SymbolTable_lock	BeforeExit_lock

续表

锁	锁
StringTable_lock	Notify_lock
CodeCache_lock	ProfileVM_lock
MethodData_lock	ExceptionCache_lock
VMOperationQueue_lock	OsrList_lock
VMOperationRequest_lock	PerfDataManager_lock
Safepoint_lock	OopMapCacheAlloc_lock
Threads_lock	Service_lock

4. 可移植性

Java 自一问世，便具有了“一次编译，到处运行”的跨平台的特点。因此，JVM 应当能够在各种系统平台上运行，这得益于 JVM 能够在不同 CPU 和操作系统类型上保持着良好的可移植性。我们知道，HotSpot 大部分的业务逻辑位于各个平台共享的公共代码（在 hotspot/src/share 路径下），而对于平台依赖的逻辑，往往使用一种简单有效的方法能够在不削弱系统可维护性前提下保证可移植性。

清单 2-12

来源：hotspot/src/share/vm/oops/oop.inline.hpp

描述：oop.hpp 中的内联函数

```

1 #ifdef TARGET_OS_ARCH_linux_x86
2 # include "orderAccess_linux_x86.inline.hpp"
3 #endif
4 #ifdef TARGET_OS_ARCH_linux_sparc
5 # include "orderAccess_linux_sparc.inline.hpp"
6 #endif
7 #ifdef TARGET_OS_ARCH_linux_zero
8 # include "orderAccess_linux_zero.inline.hpp"
9 #endif
10 #ifdef TARGET_OS_ARCH_solaris_x86
11 # include "orderAccess_solaris_x86.inline.hpp"
12 #endif
13 #ifdef TARGET_OS_ARCH_solaris_sparc
14 # include "orderAccess_solaris_sparc.inline.hpp"
15 #endif
16 #ifdef TARGET_OS_ARCH_windows_x86
17 # include "orderAccess_windows_x86.inline.hpp"
18 #endif
19 #ifdef TARGET_OS_ARCH_linux_arm
20 # include "orderAccess_linux_arm.inline.hpp"
21 #endif
22 #ifdef TARGET_OS_ARCH_linux_ppc
23 # include "orderAccess_linux_ppc.inline.hpp"
24 #endif

```

这种办法就是根据目标系统的类型选择性的包含不同的头文件。例如，在清单 2-12 中，就是利用这种方式保证了在目标运行环境为 Linux、Windows 和 Solaris 操作系统以及 Sparc、ARM、x86 和 PowerPC 等体系结构时能够引入正确的头文件。

5. VM 选项

对于经常与虚拟机调优打交道的人员来说，有必要了解 VM 选项（即 flag）。为方便用户使用，虚拟机选项可以分为如下几类。

- 基本配置类：对虚拟机主要组件或策略的配置或选择，如 Java 堆的大小、垃圾收集器的选择、编译模式的选择等。举例来说，`-agentlib` 选项用来加载本机代理库；`-Xint` 配置虚拟机以纯解释模式运行，`-Xmixed` 以解释器+即时编译混合模式运行；`-Xmx` 配置最大 Java 堆大小；`-XX:UseG1GC` 配置 G1 收集器等。
- 调优美类：对虚拟机组件或策略进行较为细致的配置，往往是尝试对虚拟机调优的重要参数。主要集中在`-XX` 选项。如`-XX:ThreadStackSize` 选项允许自定义线程栈大小。
- 性能监控类：开启监控选项，当虚拟机运行状态符合预定条件时，能够将相关信息输出以便定位问题。如使用`-Xloggc` 选项运行系统记录 GC 事件，即我们常说的 GC 日志；`-XX:ErrorFile` 选项可以配置当虚拟机遇到内部错误（JVM Crash）时，可以将错误日志写入文件，文件名格式默认为`hs_err_pid<pid>.log`。
- 内部校验类：增强虚拟机内部过程校验。如开启`-Xcheck:jni` 选项，虚拟机内部将对 JNI 函数进行额外的校验；`-XX:ImplicitNullChecks` 和`-XX:ImplicitDiv0Checks` 选项使虚拟机内部加强对空值和除零行为的校验。额外的内部校验在牺牲少量系统性的前提下增强了系统的健壮性。
- 调试和跟踪类：对虚拟机内部过程进行跟踪调试并输出跟踪日志，主要由一些`-XX` 选项组成。这类选项一般在 product 版中是受限的，仅在 debug 版或 fastdebug 版中彩允许使用，一般用作了解虚拟机的重要工具。

在系统初始化阶段，Arguments 模块会对传入的 VM 选项进行解析。

如果想提高自己的虚拟机调优技术，那么熟悉虚拟机选项应当成为必修课。虚拟机提供的配置选项至少有数百个，我们不能通过死记硬背了解每一个选项的含义和用法。在调优时，需要通过一定的实践、对比和分析，才能确定某个选项或参数对实际应用的确具有积极作用。但需要指出的是，我们应避免在实际应用中盲目尝试虚拟机调优，只有在对虚拟机的基本原理和运作机制理解的基础上才能够驾驭它。

如果你想系统了解虚拟机支持的所有选项，可以在非 product 版本虚拟机中打开选项`-XX:PrintFlagsWithComments`，这个选项允许我们查看虚拟机支持的所有选项及简要的功能描述。此外，还有一些辅助选项，如`-XX:PrintVMOptions`、`-XX:PrintFlagsInitial` 等也有可以派上用场。

本书将在各个章节中附上相关的虚拟机选项和基本功能描述，以供读者参考。

练习 1

阅读源代码，了解宏 `JVM_ENTRY` 和 `JVM_END` 的定义，并了解使用它们定义的函数。

2.1.2 HotSpot 内核框架

在扫清阅读源码的障碍后，现在是时候介绍 HotSpot 内核框架了。图 2-1 展示了 HotSpot 内核模块的基本结构。内核由一些顶层模块构成系统的主要功能组件，我们在上一章（如图 1-4 所示）也接触过这些顶层模块，如 Oops、Classfile 等功能模块。

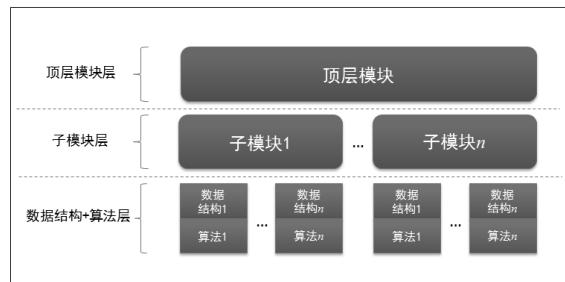


图 2-1 HotSpot 内核框架结构

HotSpot 内核主要由 C/C++ 实现，可能很多 Java 程序员会觉得阅读源码会觉得有些吃力。但事实上，只要掌握了正确的阅读源码的方法，我们完全可以打消这个疑虑。

当我们阅读任何一个开源项目源代码时，核心目标都是去理解系统的运作原理，了解功能组件如何协作和发挥作用。那么我们的着眼点应当在于抓住数据结构这一核心，去了解功能的实现算法，而不是陷入编程语言的细节。

数据结构的设计反映了功能组件的本质，从数据结构出发，可以了解组件在实现一个功能时需要考虑哪些因素：是否依赖其他组件、需要设置哪些状态、是否提供优化措施等。数据结构包括结构体、枚举、类和接口，它定义了数据成员，用以支撑算法（含功能性操作函数）的实现。而算法往往反映了功能的实现逻辑。因此从了解数据结构出发，结合算法的实现，便可以了解一个模块的具体作用，进而理解系统功能组件的实现原理。

我们知道，HotSpot 由多个顶层模块组成，主要包括 Service、Prims、Runtime、Classfile、Interpreter、Code、Memory、Compiler、Oops、C1/Opto/Shark 和 GC。

其中每个顶层模块又是由若干子模块组成。在每个子模块中，定义了一些数据结构和算法，它们相互协作实现子模块的逻辑功能。

以顶层模块 Classfile 为例，它包含了许多子模块，其中一个子模块叫做类文件解析器，位于 ClassFileParser 子模块中。在 ClassFileParser 模块中，定义了数据结构 ClassFileParser 类，用做解析*.class 格式文件（也称为 Class 文件或类文件，下同）。ClassFileParser 类用 _major_version 和 _minor_version 字段记录 Class 文件的主版本号和次版本号，并用 _class_name 字段记录类名。在算法层面，则提供了一些列的解析函数，这些解析函数实现了类文件解析器的功能主体，如 parse_constant_pool() 解析常量池、parse_interfaces() 解析接口、parse_fields() 解析字段、

parse_method()解析 Java 方法、parse_localvariable_table()解析局部变量表等。

在本书的后续章节中，将会陆续介绍 HotSpot 的内核模块。接下来，我们来了解一下对外接口模块。

2.1.3 Prims

JVM 应当具有外部访问的通道，允许外部程序访问内部状态信息。如图 2-2 所示，HotSpot 在这方面提供了丰富的通信接口，可以供 JDK 或其他应用程序调用。

在 HotSpot 内核中，由 Prims 模块定义外部接口。图 2-3 列举了 Prims 模块的部分子模块，主要包括 4 个模块。

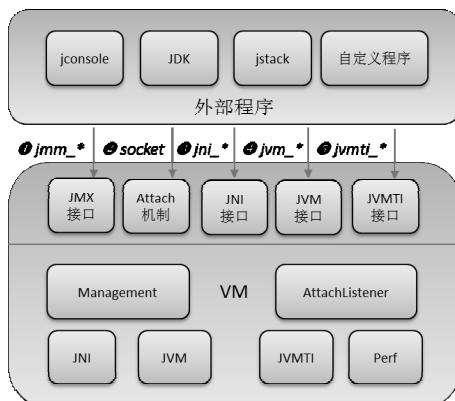


图 2-2 VM 与外界通信方式

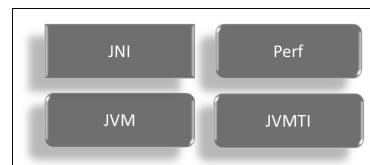


图 2-3 Prims 模块组成

1. JNI 模块

Java 本地接口（Java Native Interface，缩写为 JNI）是 Java 标准的重要组成部分。它允许 Java 代码与本地代码进行交互，如与 C/C++ 代码实现相互调用。虽然 Java 在平台移植性方面具有天然的优势，但是有的时候 Java 程序需要使用一些与底层操作系统或硬件相关的功能，这时就需要一种机制，允许调用本地库或 JVM 库。在 JDK 中定义了很多函数都依赖这些由本地语言实现的库。JNI 模块提供 Java 运行时接口，定义了许多以“jni_”为前缀命名的函数，允许 JDK 或者外部程序调用由 C/C++ 实现的库函数。

2. JVM 模块

在 JVM 模块中，虚拟机向外提供了一些函数，以“JVM_”为前缀命名，作为标准 JNI 接口的补充。这些函数可以归纳为三个部分。

首先，是一些与 JVM 相关的函数，用来支持一些需要访问本地库的 Java API。

java.lang.Object 需要这些函数实现 wait 和 notify 监视器 (monitor)，如清单 2-13 所示。

清单 2-13

来源: hotspot/src/share/vm/prims/jvm.h

描述: JVM 模块导出函数举例

```

1  JNIEXPORT void JNICALL
2  JVM_MonitorWait(JNIEnv *env, jobject obj, jlong ms);

3  JNIEXPORT void JNICALL
4  JVM_MonitorNotify(JNIEnv *env, jobject obj);

```

其次，是一些函数和常量定义，用来支持字节码验证和 Class 文件格式校验。清单 2-14 列举了部分由 “JVM_” 作为前缀命名的函数和常量，用作 Class 文件解析。

清单 2-14

来源: hotspot/src/share/vm/prims/jvm.h

描述: JVM 模块导出函数和常量举例

```

1  /*
2   * Returns the constant pool types in the buffer provided by "types."
3   */
4  JNIEXPORT void JNICALL
5  JVM_GetClassCPTypes(JNIEnv *env, jclass cb, unsigned char *types);

6  /*
7   * Returns the number of *declared* fields or methods.
8   */
9  JNIEXPORT jint JNICALL
10 JVM_GetClassFieldsCount(JNIEnv *env, jclass cb);

11 #define JVM_ACC_PUBLIC      0x0001 /* visible to everyone */
12 #define JVM_ACC_PRIVATE     0x0002 /* visible only to the defining class */
13 #define JVM_ACC_PROTECTED   0x0004 /* visible to subclasses */

```

最后，是各种 I/O 和网络操作，用来支持 Java I/O 和网络 API。清单 2-15 列举了部分由 JVM 模块导出的 I/O 函数和网络操作函数。

清单 2-15

来源: hotspot/src/share/vm/prims/jvm.h

描述: JVM 模块导出 I/O 和 network 函数

```

1  JNIEXPORT jint JNICALL
2  JVM_Open(const char *fname, jint flags, jint mode);

3  JNIEXPORT jint JNICALL
4  JVM_Read(jint fd, char *buf, jint nbytes);

5  JNIEXPORT jint JNICALL
6  JVM_Socket(jint domain, jint type, jint protocol);

7  JNIEXPORT jint JNICALL
8  JVM_Recv(jint fd, char *buf, jint nBytes, jint flags);

9  JNIEXPORT jint JNICALL
20 JVM_Send(jint fd, char *buf, jint nBytes, jint flags);

```

JVM 模块的导出函数均在头文件 jvm.h 中声明。HotSpot 项目中的 jvm.h 文件与 JDK 使用的头文件是一致的。而函数的具体实现则是在源文件 jvm.cpp 中使用 JVM_ENTRY 宏方式

定义。

3. JVMTI 模块

Java 虚拟机工具接口（Java Virtual Machine Tool Interface，缩写为 JVMTI）提供了一种编程接口，允许程序员创建代理以监视和控制 Java 应用程序。JVMTI 代理常用于对应用程序进行监控、调试或调优。例如监控内存实际使用情况、CPU 利用率以及锁信息等。该模块为外部程序提供 JVMTI 接口。

4. Perf 模块

JDK 中 sun.misc.Perf 类的底层实现，定义了一些以“Perf_”为前缀命名的函数，由外部程序调用，以监控虚拟机内部的 Perf Data 计数器。

2.1.4 Services

Services 模块为 JVM 提供了 JMX 等功能。JMX（即 Java Management Extensions）是为支持对 Java 应用程序进行管理和监控而定义的一套体系结构、设计模式、API 以及服务。通常使用 JMX 来监控系统的运行状态或者对系统进行灵活的配置，比如清空缓存、重新加载配置文件、更改配置等。

JMX 可以跨越一系列异构操作系统平台、系统体系结构和网络传输协议，为程序员开发无缝集成的系统、网络和服务管理应用提供一定程度的灵活性。

Services 模块包含以下 9 个主要子模块，如图 2-4 所示。

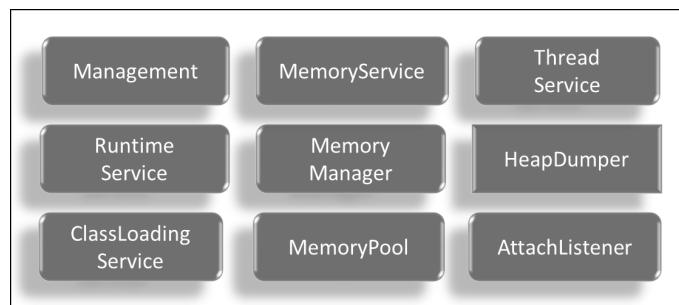


图 2-4 Services 模块组成

1. Management 模块

Management 模块提供 JMX 底层实现的基础。例如，在 Java 层开发 JMX 程序时，会遇到如下几个函数：

- Java_sun_management_MemoryManagerImpl_getMemoryPools0;
- Java_sun_management_MemoryImpl_getMemoryManagers0;
- Java_sun_management_MemoryImpl_getMemoryUsage0;
- Java_sun_management_ThreadImpl_dumpThreads0 等。

JMX 示例程序会输出内存池及垃圾收集器等信息，如清单 2-16 所示。

清单 2-16

来源: com.hotspotinaction.demo.chap2.MemoryPoolInfo

描述: 利用 JMX 获取 GC 信息

```

1  List<MemoryPoolMXBean> pools = ManagementFactory.getMemoryPoolMXBeans();
2  int poolsFound = 0;
3  int poolsWithStats = 0;
4  for (int i = 0; i < pools.size(); i++) {
5      MemoryPoolMXBean pool = pools.get(i);
6      String name = pool.getName();
7      System.out.println("found pool: " + name);

8      if (name.contains(poolName)) {
9          long usage = pool.getCollectionUsage().getUsed();
10         System.out.println(name + ": usage after GC = " + usage);
11         poolsFound++;
12         if (usage > 0) {
13             poolsWithStats++;
14         }
15     }
16 }
17 if (poolsFound == 0) {
18     throw new RuntimeException("无匹配的内存池: 请打开-XX:+UseConcMarkSweepGC");
19 }

20 List<GarbageCollectorMXBean> collectors = ManagementFactory.getGarbageCollectorMX
Beans();
21 int collectorsFound = 0;
22 int collectorsWithTime = 0;
23 for (int i = 0; i < collectors.size(); i++) {
24     GarbageCollectorMXBean collector = collectors.get(i);
25     String name = collector.getName();
26     System.out.println("found collector: " + name);
27     if (name.contains(collectorName)) {
28         collectorsFound++;
29         System.out.println(name + ": collection count = " + collector.getCollectionCount
());
30         System.out.println(name + ": collection time = " + collector.getCollectionTime
());
31         if (collector.getCollectionCount() <= 0) {
32             throw new RuntimeException("collection count <= 0");
33         }
34         if (collector.getCollectionTime() > 0) {
35             collectorsWithTime++;
36         }
37     }
38 }
```

运行得到如清单 2-17 所示的日志。

清单 2-17

```
found pool: Code Cache
found pool: Par Eden Space
found pool: Par Survivor Space
found pool: CMS Old Gen
CMS Old Gen: usage after GC = 208123288
found pool: CMS Perm Gen
CMS Perm Gen: usage after GC = 2626528
found collector: ParNew
found collector: ConcurrentMarkSweep
ConcurrentMarkSweep: collection count = 7
ConcurrentMarkSweep: collection time = 17
```

JVM 以 management 动态链接库的形式，向 JDK 提供一套监控和管理虚拟机的 jmm 接口，如清单 2-18 所示。动态链接库被安装在 JRE/bin 目录下。例如，在 Windows 平台上，JRE/bin 目录下名为 management.dll 的文件即为该库。

清单 2-18

来源: hotspot/src/share/vm/services/management.cpp

描述: jmm_interface

```
1 const struct jmmInterface_1_ jmm_interface = {
2     NULL,
3     NULL,
4     jmm_GetVersion,
5     jmm_GetOptionalSupport,
6     jmm_GetInputArguments,
7     jmm_GetThreadInfo,
8     jmm_GetInputArgumentArray,
9     jmm_GetMemoryPools,
10    jmm_GetMemoryManagers,
11    jmm_GetMemoryPoolUsage,
12    jmm_GetPeakMemoryPoolUsage,
13    jmm_GetThreadAllocatedMemory,
14    jmm_GetMemoryUsage,  
15    jmm_GetLongAttribute,
16    jmm_GetBoolAttribute,
17    jmm_SetBoolAttribute,
18    jmm_GetLongAttributes,
19    jmm_FindMonitorDeadlockedThreads,
20    jmm_GetThreadCpuTime,
21    jmm_GetVMGlobalNames,
22    jmm_GetVMGlobals,
23    jmm_GetInternalThreadTimes,
24    jmm_ResetStatistic,
25    jmm_SetPoolSensor,
26    jmm_SetPoolThreshold,
27    jmm_GetPoolCollectionUsage,
28    jmm_GetGCExtAttributeInfo,
29    jmm_GetLastGCStat,
30    jmm_GetThreadCpuTimeWithKind,
31    jmm_GetThreadCpuTimesWithKind,
32    jmm_DumpHeap0,
33    jmm_FindDeadlockedThreads,
34    jmm_SetVMGlobal,
35    NULL,
36    jmm_DumpThreads,
37    jmm_SetGCNotificationEnabled
38 };
```

如果读者想了解 JMX 在虚拟机中是如何实现的，可以在 management 模块中查看上述 jmm 接口函数（使用 JVM_ENTRY 和 JVM_END 宏定义）。

例如，清单 2-5 中由 JVM_ENTRY 宏定义的函数 getMemoryUsage0，实现的便是 JMX 接口 Java_sun_management_MemoryImpl_getMemoryUsage0。

虚拟机启动过程中，在初始化 management 模块时，将调用顶层模块 Runtime 中的 ServiceThread 模块，启动一个名为“Service Thread”的守护线程（见清单 2-33，在较早版本中也称“Low Memory Detector”），该守护线程负责向 JVM 上报内存不足报警。

若系统开启了选项-XX:ManagementServer，则加载并创建 sun.management.Agent 类，执行其 startAgent()方法启动 JMX Server。

除了 JXM 功能模块以外，还有其他几个模块，下面依次介绍。

2. MemoryService 模块

提供 JVM 内存管理服务。如堆的分配和内存池的管理等。

3. MemoryPool 模块

内存池管理模块。内存池表示由 JVM 管理的内存区域，是内存管理的基本单元。JVM 拥有至少一块内存池，它可以在 JVM 运行期间创建或删除。一块内存池可以由堆或非堆拥有，也可以由它们同时拥有。MemoryPool 分为两类：CollectedMemoryPool 和 CodeHeapPool。其中，CollectedMemoryPool 又可分为如下几种子类型：

- ContiguousSpacePool;
- SurvivorContiguousSpacePool;
- CompactibleFreeListSpacePool;
- GenerationPool。

4. MemoryManager 模块

内存管理器。一个内存管理负责管理一个或多个内存池。垃圾收集器也是一种内存管理器，它负责回收不可达对象的内存空间。在 JVM 中，允许有一个或多个内存管理器，允许在系统运行期间根据情况添加或删除内存管理器。一个内存池可以由一个或多内存管理器管理。已定义的内存管理器包括 CodeCacheMemoryManager、GCMemoryManager。其中 GCMemoryManager 又可分为如下几种子类型：

- CopyMemoryManager;
- MSCMemoryManager;
- ParNewMemoryManager;
- CMSMemoryManager;

- PSScavengeMemoryManager;
- PSMarkSweepMemoryManager;
- G1YoungGenMemoryManager;
- G1OldGenMemoryManager 等。

5. RuntimeService 模块

提供 Java 运行时的性能监控和管理服务，如 applicationTime、jvmCapabilities 等。

6. ThreadService 模块

提供线程和内部同步系统的性能监控和管理服务，包括维护线程列表、线程相关的性能统计、线程快照、线程堆栈跟踪和线程转储等功能。

7. ClassLoadingService 模块

提供类加载模块的性能监控和管理服务。

8. AttachListener 模块

JVM 系统初始化时启动名为“Attach Listener”的守护线程。它是为客户端的 JVM 监控工具提供连接（attach）服务，它维护一个操作列表用来接受已连接的客户端进程发送的操作请求，当执行完毕这些操作后，将数据返回给客户端进程。关于 Attach 机制的详细内容，请参考本书第 9 章。

9. HeapDumper 模块

提供堆转储功能，将堆转储信息写入 HPROF 格式二进制文件中。关于对转储机制和 HPROF 格式的更多内容，可以参考本书的第 9 章。

练习 2

在安装的 JRE 下面寻找 management 动态链接库，查看库中分别包含了哪些函数符号。查阅资料，了解这些函数的作用？通过已学过的知识，试着在 openjdk 源代码中独立找到这些函数的定义。

2.1.5 Runtime

Runtime 是运行时模块，它为其他系统组件提供运行时支持。JVM 在运行时需要的很多功能，如线程、安全点、PerfData、Stub 例程、反射、VMOperation 以及互斥锁等组件，均由 Runtime 模块定义。在本书后续章节对相关专题展开讲解时，还将看到 Runtime 模块发挥着重要作用。

如图 2-5 所示，顶层模块 Runtime 中定义了大量公共模块，限于篇幅，这里不能一一详解，

仅对几个主要模块进行介绍，读者如需要进一步了解，可以参阅源代码。

1. Thread 模块

定义了各种线程类型，包含 JVM 内部工作线程以及 Java 业务线程。此外，还定义了 Threads 子模块，它维护着系统有效线程队列。

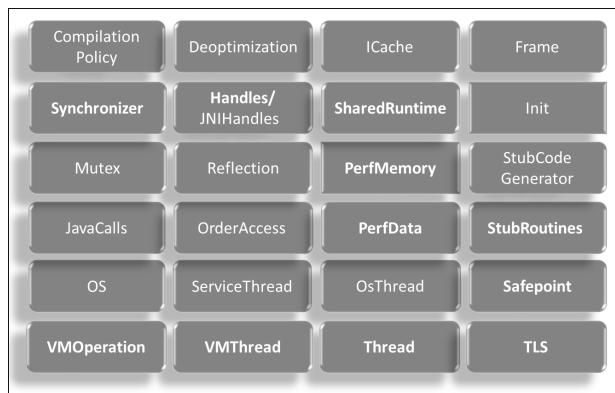


图 2-5 Runtime 模块的主要组成

2. Arguments 模块

记录和传递 VM 参数和选项，详见第 9 章。

3. StubRoutines 和 StubCodeGenerator 模块

生成 stub，关于 stub 的更多信息，可以参考第 7 章。

4. Frame 模块

Frame 表示一个物理栈帧（又称活动记录，Activation Record，详见第 7 章），Frame 模块定义了表示物理栈帧的数据结构 frame，如清单 2-19 所示。frame 是与 CPU 类型相关的，既可以表示 C 帧，也可以表示 Java 帧，对于 Java 帧既可以是解释帧，也可以是编译帧。

清单 2-19

来源: hotspot/src/share/vm/runtime/frame.hpp

描述: frame 的成员变量

```

1 class frame {
2 private:
3     // 成员变量
4     intptr_t* _sp; // stack pointer (from Thread::last_Java_sp)
5     address _pc; // program counter (the next instruction after the call)
6     CodeBlob* _cb; // CodeBlob that "owns" pc
7     enum deopt_state {

```

```
8     not_deoptimized,
9     is_deoptimized,
10    unknown
11  };
12
13  deopt_state _deopt_state;
14 }
```

在 frame 类型的定义中，会根据具体的 CPU 类型定义不同的函数实现。一个 frame 可由栈指针、PC 指针、CodeBlob 指针和状态位描述。其中，这三个指针的作用为：栈帧_sp 指向栈顶元素；PC 指针_pc 指向下一条要执行的指令地址；CodeBlob 指针指向持有相应的指令机器码的 CodeBlob。在第 6 章中，我们将会看到 frame 的设计在结构上十分类似于真实机器的栈帧结构。

5. CompilationPolicy 模块

CompilationPolicy 模块用来配置编译策略，即选择什么样的方法或循环体来编译。Runtime 模块中定义了两种编译策略类型——SimpleThresholdPolicy 和 AdvancedThresholdPolicy。CompilationPolicy 模块初始化时，将根据 VM 选项 CompilationPolicyChoice 来配置编译策略，选项为 0 表示 SimpleCompPolicy，为 1 则表示 StackWalkCompPolicy。

6. Init 模块

用于系统初始化，如启动 init_globals() 函数初始化全局模块等。

7. VmThread 模块

在虚拟机创建时，将会在全局范围内创建一个单例原生线程 VMThread（虚拟机线程），该线程名为“VM Thread”，能够派生出其他的线程。该线程的一个重要职责是：维护一个虚拟机操作队列（VMOperationQueue），接受其他线程请求虚拟机级别的操作（VMOperation），如执行 GC 等任务。事实上，VMOperation 是 JVM 对外及对内提供的核心服务。甚至是在一些外部虚拟机监控工具（详见第 9 章）中，也享受到了这些 VMOperation 所提供的服务。

这些操作根据阻塞类型以及是否进入安全点，可分为 4 种操作模式（Mode）。

- safepoint: 阻塞，进入安全点。
- no_safepoint: 阻塞，非进入安全点。
- concurrent: 非阻塞，非进入安全点。
- async_safepoint: 非阻塞，进入安全点。

8. VMOperation 模块

虚拟机内部定义的 VM 操作有 ThreadStop、ThreadDump、PrintThreads、FindDeadlocks、

ForceSafepoint、ForceAsyncSafepoint、Deoptimize、HandleFullCodeCache、Verify、HeapDumper、GenCollectFull、ParallelGCSysytemGC、CMS_Initial_Mark、CMS_Final_Remark、G1CollectFull、G1CollectForAllocation、G1IncCollectionPause、GetStackTrace、HeapWalkOperation、HeapIterateOperation 等（详见 `vm_operations.hpp`）。

这些 VM 操作均继承自共同父类 `VM_Operation`。了解哪些 VM 操作需要进入安全点，对我们在实践中排查引起程序停顿过久这类问题时大有裨益。可以通过函数 `evaluation_mode()` 判断具体的操作模式，如清单 2-20 所示。

清单 2-20
来源: `hotspot/src/share/vm/runtime/vm_operations.hpp`
描述: `VM_Operation::evaluation_mode()`

```
virtual Mode evaluation_mode() const { return _safepoint; }
```

若具体实现子类有特殊模式要求，将会覆盖该函数，否则默认模式为 `safepoint`。

2.2 启动

`Launcher`（启动器），是用来启动 JVM 和应用程序的工具。在这一节中，我们将看到 HotSpot 中提供了两种 `Launcher` 类型，分别是通用启动器和调试版启动器。

2.2.1 Launcher

通用启动器（Generic Launcher）是指我们比较熟悉的 JDK 命令程序：`java`（含 `javaw`）。`java` 是由 JDK 自带的启动 Java 应用程序的工具。为启动一个 Java 应用程序，`java` 将准备一个 Java 运行时环境（即 JRE）、加载指定的类并调用它的 `main` 方法。类加载的前提条件是由 JRE 在指定路径下找到类加载器和应用程序类。一般来说，JRE 将在以下 3 种路径下搜索类加载器和其他类：

- 引导类路径（bootstrap class path）；
- 已安装的扩展（installed extensions）；
- 用户类路径（user class path）。

类被加载进来之后，`java` 会将全限定类名或 JAR 文件名之后的非选项类参数作为参数传递给 `main` 方法。

`javaw` 命令等同于 `java`，只是 `javaw` 没有控制台窗口。当你不想显示一个命令提示符窗口时，可以使用 `javaw`。但是如果由于某些原因启动失败，`javaw` 仍将显示一个对话框提供错误信息。

1. 基本用法

`java` 和 `javaw` 的命令格式如下所示：

```
java [ option ] class [ argument ... ]
java [ option ] -jar file.jar [ argument ... ]
javaw [ option ] class [ argument ... ]
javaw [ option ] -jar file.jar [ argument ... ]
```

其中 class 是要调用的类名，而 file.jar 是要调用的 JAR 文件名。

值得注意的是，我们需要区分选项和参数的不同用途。

- 选项 (option) 是传递给 VM 的参数。目前，有两类 VM 选项，包括标准 VM 选项和非标准 VM 选项。其中，非标准选择在使用时以 “-X” 或 “-XX” 指定。
- 参数 (argument) 是传递给 main 方法的参数。

注意 对于启动器，有一套标准选项 (standard options)，在当前和将来的版本中都将支持。此外，HotSpot 虚拟机默认提供一套非标选项 (non-standard options)，这些非标选项有可能在将来版本中更改。另外，32 位 JDK 和 64 位 JDK 命令选项也会有所不同。

2. 标准 VM 选项

标准 VM 选项主要包括以下几项。

- -client、-server: 指定 HotSpot 以 client 或 server 模式运行虚拟机。对于 64 位 JDK，将忽略此选项，默认以 server 模式运行虚拟机。
- -agentlib:libname[=options]: 按照库名 libname 载入本地代理库 (agent library)。如 -agentlib:hprof、-agentlib:jdwp=help、-agentlib:hprof=help。
- -agentpath:pathname[=options]: 按照完整路径名 pathname 载入本地代理库。
- -classpath、-cp: 指定类文件搜索路径。
- -Dproperty=value: 设置系统属性值
- -jar: 执行封装在 jar 文件中的应用程序。
- -javaagent:jarpath[=options]: 加载 Java 编程语言代理库，可参阅 `java.lang.instrument`。
- -verbose、-verbose:class: 显示每个被加载的类信息。
- -verbose:gc: 报告每个垃圾回收事件。
- -verbose:jni: 报告关于调用本地方法和其他本地接口的信息。
- -X: 显示非标准选项信息，然后退出。

3. 非标准 VM 选项

以 “-X” 指定的非标准 VM 选项主要包括以下几项¹。

- -Xint: 以解释模式运行虚拟机。禁用编译本机代码，并由解释器 (interpreter) 执行所有字节码。

¹ 可以在 JDK 安装路径下找到这些选项的帮助文件 `Xusage.txt`。

- **-Xbatch:** 禁用后台编译。一般来说，虚拟机将编译方法作为后台任务，虚拟机在解释器模式下运行某方法时，需要等到后台编译完成该方法的编译任务。该参数将禁用后台编译，使方法的编译作为前台任务直到完成为止。
- **-Xbootclasspath:** 指定引导类和资源文件的搜索路径。
- **-Xcheck:jni:** 对于 Java 本地接口 JNI 函数执行额外的检查。JVM 验证传递给 JNI 函数的参数。在本机代码中遇到任何无效的数据将导致 JVM 终止。使用此选项时，会带来一些性能损失。
- **-Xfuture:** 执行严格的类文件格式检查。
- **-Xnoclassgc:** 禁用类垃圾回收。
- **-Xincgc:** 启用增量垃圾回收器。
- **-Xloggc:<file>:** 报告垃圾回收事件，并记录到<file>指定的文件中。
- **-Xms<size>:** 设置 Java 堆的初始化大小。
- **-Xmxn:** 设置 Java 堆的最大值。
- **-Xssn:** 设置 Java 线程的栈大小。
- **-Xprof:** 输出 CPU 性能数据。

4. 隐藏的非标 VM 选项

这一类选项以“-XX”指定。该类 VM 选项数量十分可观，可以说有成百上千个也不为过。本书将在各章节中，附上一些相关的虚拟机选项和功能描述，以供参考。

5. gamma: 调试版启动器

HotSpot 提供了一个精简调试 Launcher，称为 gamma。相对于通用 Launcher，gamma 就安装在与 JVM 库相同的目录下，或者与 JVM 库静态链接为一个库文件，因此可以把 gamma 看作是精简了虚拟机选项解析等逻辑的 java 命令。

事实上，为便于维护，OpenJDK 就是基于同一套 Launcher 代码维护了 gamma launcher 和通用 launcher 的，对于差异代码则使用#ifndef GAMMA 进行注释区分。gamma 启动器入口位于 hotspot/src/share/tools/luncher/java.c；通用 Launcher 的入口并不在 hotspot 工程下，感兴趣的读者可以在与 hotspot 同级目录 jdk 下找到 hotspot/../jdk/src/share/bin/main.c。

从本节开始，我们将以 Launcher 作为切入点，对 HotSpot 进行实战调试和分析。为方便调试，我们将在 Linux 平台上基于 gamma 启动器来讲解 HotSpot 启动过程。

2.2.2 虚拟机生命周期

图 2-6 描述了一个完整的虚拟机生命周期，具体过程如下。

(1) Launcher 启动后，首先进入 Launcher 的入口，即 main 函数。正像稍后看到的那样，

main 工作的重点是：创建一个运行环境，为接下来启动一个新的线程创建 JVM 并跳到 Java 主方法做好一切准备工作。

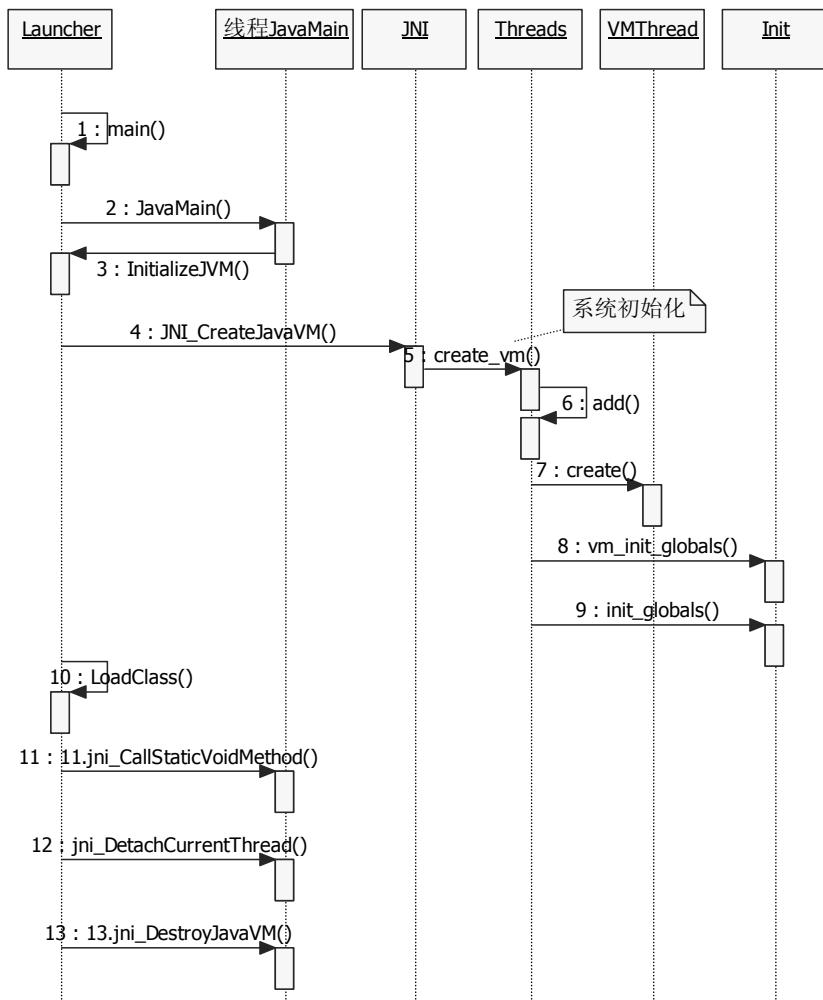


图 2-6 虚拟机生命周期

(2) 环境就绪后，Launcher 启动 JavaMain 线程，将程序参数传递给它。如清单 2-21 所示，Launcher 调用 ContinueInNewThread() 函数启动新的线程并继续执行任务。新的线程将要执行的任务由该函数的第一个参数指定，即 JavaMain() 函数。这时，新线程将要阻塞当前线程，并在新线程中开启一段相对独立的历程，去完成 Launcher 赋予它的使命。

清单 2-21

来源：hotspot/src/share/tools/luncher/java.c

描述: Launcher 启动 JavaMain 线程

```
return ContinueInNewThread(JavaMain, threadStackSize, (void*)&args);
```

(3) 一般来说, JavaMain 线程将伴随应用程序的整个生命周期。首先, 它要做的便是在 Launcher 模块内调用 InitializeJVM() 函数, 初始化 JVM。值得一提的是, 在理解虚拟机生命周期复杂的模块调用过程时, 我们不能对 Launcher 模块本身抱有过高的期待。毕竟, Launcher 模块本身无力实现这些核心功能, 它必须借助其他专门模块来提供相应功能。因此, 在阅读源代码时, 我们应当培养这样的意识, 在遇到某个核心功能或重要组件时, 首先问自己几个问题: 核心功能是由哪个模块提供的? 它最终是为系统哪个组件提供服务的? 它是以什么形式向调用者提供服务的? 养成这种意识, 对于独立分析和思考系统运作具有重要的意义。

Launcher 模块本身并不具有创建虚拟机的能力。下面我们将看到, 有哪些模块参与了这个过程。由于 Launcher 模块需要借助自身以外的力量完成任务, 理所当然地, 它需要拥有访问外部接口的能力。稍后将提到一些数据结构, 它们持有外部接口的函数指针, Launcher 通过它们可以达到调用外部接口的目的。

(4) 虚拟机在 Prims 模块中定义了一些以 “JNI_” 为前缀而命名的函数, 并向外部提供这些 jni 接口。JNI_CreateJavaVM() 函数就是其中一个, 它为外部程序提供创建 JVM 的服务。前面提到的创建 JVM 的任务, 实际上就是调用了 JNI_CreateJavaVM() 函数。JNI 模块是连接虚拟机内部与外部程序的桥梁, JVM 系统内部的命名空间对 JNI 模块都是可见的, 因此它可以调用内部模块并通过接口向外提供查看和操纵 JVM 的能力。JNI_CreateJavaVM() 函数调用 Threads 模块 create_vm() 函数完成最终的虚拟机的创建和初始化工作。

(5) 可以说, create_vm() 函数是 JVM 启动过程的精华部分, 它初始化了 JVM 系统中绝大多数的模块。

- (6) 调用 add() 函数, 将线程加入线程队列;
- (7) 调用 create() 函数, 创建虚拟机线程 “VMThread”;
- (8) 调用 vm_init_globals() 函数, 初始化全局数据结构;
- (9) 调用 init_globals() 函数, 初始化全局模块;
- (10) 调用 LoadClass() 函数, 加载应用程序主类;
- (11) 调用 jni_CallStaticVoidMethod() 函数, 实现对 Java 应用程序的主方法的调用;
- (12) 调用 jni_DetachCurrentThread() 函数;
- (13) 调用 jni_DestroyJavaVM() 函数, 销毁 JVM 后退出。

接下来, 我们将选取一些重要过程展开详解。

2.2.3 入口: main 函数

与其他应用程序一样, Launcher 的入口是一个 main 函数。在不同操作系统中, main 函数的原型看起来会有些差异。例如在 UNIX 或 Linux 系统中, 按照 POSIX 规范的函数原型如清单 2-22 所示。

清单 2-22

来源: hotspot/src/share/tools/luncher/java.c
描述: launcher 入口

```
int
main(int argc, char ** argv)
```

而在 Windows 平台上, 其原型如清单 2-23 所示。

清单 2-23

来源: jdk/src/share/bin/main.c
描述: launcher 入口

```
int WINAPI
WinMain(HINSTANCE inst, HINSTANCE previnst, LPSTR cmdline, int cmdshow)
```

main 函数的程序流程如图 2-7 所示。

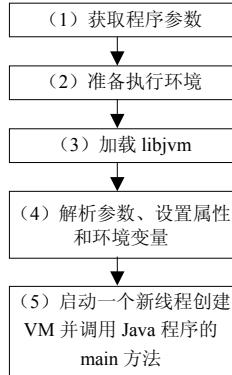


图 2-7 main 函数流程

在 main 函数执行的最后一步, 程序将启动一个新的线程并将 Java 程序参数传递给它, 接下来阻塞自己, 并在新线程中继续执行。新线程也可称为为主线程, 它的执行入口是 JavaMain。

2.2.4 主线程

一般来说, 主线程将伴随应用程序的整个生命周期。打个形象的比喻: JavaMain 好比一个外壳, 应用程序便是在这个外壳的包裹下完成执行的。它的函数原型如清单 2-24 (a) 所示。

清单 2-24(a)

来源: hotspot/src/share/tools/luncher/java.c
描述: 启动新线程执行该方法

```
int JNICALL
JavaMain(void * _args)
```

在介绍 JavaMain 的主要流程前, 我们先了解几个重要的基础数据结构, 它们在调用主方法、断开主线程和销毁 JVM 的过程中发挥重要作用的数据结构。它们分别是 JavaVM、JNIEnv 和 InvocationFunctions。

JavaVM 类型是一个结构体，它拥有一组少而精的函数指针²。顾名思义，这几个函数为 JVM 提供了诸如连接线程、断开线程和销毁虚拟机等重要功能。在 JavaMain 的流程中，我们也可以看到这些功能的执行。HotSpot 定义了大量的运行时接口，上述功能实际上是由这些接口提供的。如图 2-8 所示，在 JavaMain 运行时由 InitializeJVM 模块将 JavaVM 的这些成员赋上正确的值，指向相应的 JNI 接口函数上。

同 JavaVM 类型类似，JNIEnv 也是拥有一组函数指针的结构体。不过，相对于 JavaVM 来说，它是一个重量级类型，JNIEnv 容纳了大量的函数指针成员。同样地，在 JavaMain 运行时由 InitializeJVM 模块将 JNIEnv 的这些成员赋上正确的值，指向相应的 JNI 接口函数上。

InvocationFunctions 中定义了 2 个函数指针，CreateJavaVM 和 GetDefaultJavaVMInitArgs，如图 2-9 所示，这 2 个函数在加载 libjvm 时已经指派好了。

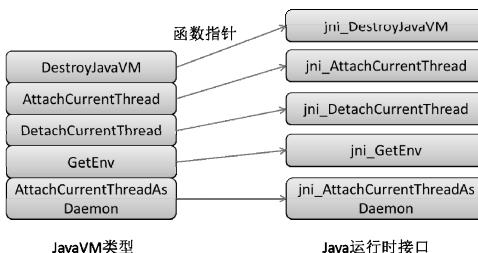


图 2-8 JavaVM 类型

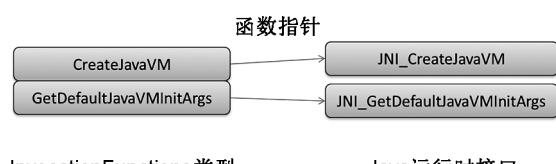


图 2-9 InvocationFunctions 类型

在 JavaMain 中，拥有 3 个局部变量：vm、env 和 ifn，分别对应着上述 JavaVM、JNIEnv 和 InvocationFunctions 这三种类型。JavaMain 的主要流程如图 2-10 所示。

(1) 初始化虚拟机：调用 InitializeJVM 模块，将 JavaVM 和 JNIEnv 类型的成员指向正确的 jni 函数上。

(2) 获取应用程序主类（main class），如清单 2-24（b）所示。

清单 2-24 (b)

```
jclass mainClass = LoadClass(env, classname);
```

(3) 获取应用程序主方法（main method），如清单 2-24（c）所示。

清单 2-24 (c)

```
jmethodID mainID = (*env)->GetStaticMethodID(env, mainClass, "main",
                                                "[Ljava/lang/String;]V");
```

(4) 传递应用程序参数并执行主方法，如清单 2-24（d）所示。

清单 2-24 (d)

```
(*env)->CallStaticVoidMethod(env, mainClass, mainID, mainArgs);
```

² 对于 Java 程序员，可能不太熟悉结构体或指针，这里可以将结构体 JavaVM 理解成拥有一组方法的类，将其中的每个函数指针成员理解成该类的方法成员。



图 2-10 JavaMain() 程序流程

(5) 与主线程断开连接，如清单 2-24 (e) 所示。

清单 2-24 (e)

```
(*vm) ->DetachCurrentThread(vm);
```

(6) 主方法执行完毕，等待非守护线程结束，然后创建一个名为“DestroyJavaVM”的 Java 线程执行销毁 JVM 任务，如清单 2-24 (f) 所示。

清单 2-24 (f)

```
(*vm) ->DestroyJavaVM(vm);
```

练习 3

阅读源代码，认真分析 JavaMain 函数，体会它在 JVM 中的作用和地位。如果通过前面的学习，你已经掌握了调试的基本方法，请仔细调试这部分程序。

2.2.5 InitializeJVM 函数

在第 1 章中，我们知道，在编译 HotSpot 项目后，启动脚本 hotspot 中会默认设置一个断点，即 InitializeJVM。启动 GDB 调试 HotSpot，JVM 开始运行 HelloWorld 程序，但是程序并不急于打印“Hello hotspot!”，而是先停在了断点“Breakpoint 1”上（如图 1-6 所示），即 InitializeJVM 函数。

现在，我们想将断点往前挪一点，以便于我们详细了解 JavaMain 的运行细节。利用 GDB，我们将断点设置在 JavaMain 函数的入口处，GDB 界面中输入如下命令：

```
(gdb) break java.c:JavaMain
```

或者直接使用代码行数：

```
(gdb) break java.c:396
```

断点设置完毕后，我们再次启动调试，如图 2-11 所示。

```

(x) chentao@ubuntu: ~/hotspot-in-action/hotspot/build/hotspot_debug/linux_i486_c
1 micro seconds to LoadJavaVM
[New Thread 0xb6f35b40 (LWP 5089)]
[Switching to Thread 0xb6f35b40 (LWP 5089)]

Breakpoint 1, JavaMain (_args=0xbffffcfe4)
    at /home/chentao/hotspot-in-action/hotspot/src/share/tools/launcher/java.c:96
96
---Type <return> to continue, or q <return> to quit---return
396      struct JavaMainArgs *args = (struct JavaMainArgs *)_args;
(gdb) 

```

图 2-11 断点设置：JavaMain

HotSpot 运行至第 396 行，停了下来。实际上，我们在这里共设置了 2 个断点（JavaMain 和 InitializeJVM）。输入 continue 命令让程序继续运行至第 1270 行，即 InitializeJVM。

InitializeJVM 的原型如清单 2-25 所示。

清单 2-25

来源: hotspot/src/share/tools/launcher/java.c & jdk/src/share/bin/java.c

描述: InitializeJVM

```

static jboolean
InitializeJVM(JavaVM **pvm, JNIEnv **penv, InvocationFunctions *ifn)

```

数字 1270 的含义是下一条将要运行的语句行数，如图 2-12 所示，这行代码是一条 memset 语句，用来对 main() 函数的参数 args 进行初始化填零。

```

(x) chentao@ubuntu: ~/hotspot-in-action/hotspot/build/hotspot_debug/linux_i486_c
Breakpoint 2, InitializeJVM (pvm=0xb6f352b4, penv=0xb6f352b8, ifn=0xb6f3531
8)
    at /home/chentao/hotspot-in-action/hotspot/src/share/tools/launcher/java.c:1270
1270      memset(&args, 0, sizeof(args));
(gdb) continue
Continuing.
JavaVM args:
    version 0x00010002, ignoreUnrecognized is JNI_FALSE, nOptions is 7
option[ 0] = '-Djava.class.path=.:~/usr/lib/jvm/java-7-sun/lib:/usr/lib/jvm/java-7-sun/jre/lib'
option[ 1] = '-XX:+TraceClassInitialization'
option[ 2] = '-XX:+UseSplitVerifier'
option[ 3] = '-XX:+Verbose'
option[ 4] = '-XX:+WizardMode'
option[ 5] = '-Dsun.java.launcher=gamma'
option[ 6] = '-Dsun.java.launcher.pid=5087'

```

图 2-12 打印 VM 版本和选项信息

利用 GDB 调试工具，我们还可以深入到 InitializeJVM 内部，看看 InitializeJVM 是如何初始化 JVM 的。由前文可以，InitializeJVM 的任务之一就是需要完成对 vm 和 env 指派接口函数的重任。在调用 InitializeJVM 返回后，通过 GDB 查看命令，我们可以看到 vm 的函数指针成员得到了赋值，如图 2-13 所示。

此外，InitializeJVM 中还会打印一些额外信息，如图 2-12 所示，可以看到 InitializeJVM 打印了一些与 JVM 的版本和选项相关的信息。

```
(gdb) p vm
$4 = (JavaVM *) 0xb7b675c4 <main_vm>
(gdb) p *vm
$5 = (JavaVM) 0xb7b59fa0 <jni_InvokeInterface>
(gdb) p **vm
$6 = {reserved0 = 0x0, reserved1 = 0x0, reserved2 = 0x0,
DestroyJavaVM = 0xb7584b83 <jni_DestroyJavaVM>,
AttachCurrentThread = 0xb7584fbd <jni_AttachCurrentThread>,
DetachCurrentThread = 0xb75850b0 <jni_DetachCurrentThread>,
GetEnv = 0xb75851e6 <jni_GetEnv>,
AttachCurrentThreadAsDaemon = 0xb7585397 <jni_AttachCurrentThreadAsDaemon>}
(gdb)
```

图 2-13 InitializeJVM 对 vm 赋值

通过这些调试过程，相信读者对使用 GDB 调试 HotSpot 又有了新的认识。可是，到现在为止，我们仍然没有接触到与 JVM 的创建或初始化相关的实质内容，只是知道在调用 CreateJavaVM 之后，得到了大量的 JNI 函数。显然，这一过程向我们屏蔽了很多细节。但是换句话说，现在我们距离 JVM 初始化的核心内容仅一步之遥了。

在继续深入了解 JVM 的创建和初始化过程之前，我们希望你能够做些小的练习，以便巩固刚才学过的知识，同时为接下来的深入学习打下良好的实践基础。

练习 4

将断点设置在 JavaMain 跟踪调试，在 InitializeJVM 返回之后，确认 env 成员已指派到了正确的 jni 接口函数上。

练习 5

试一试在你安装的正式版 JDK 中，找到下面这些函数符号：

JNI_CreateJavaVM

JNI_GetDefaultJavaVMInitArgs

提示 在 Windows 上，可以使用 DLL export Viewer 等 dll 查看工具列出其中包含的符号；在 UNIX 上，可以通过 nm 等工具查看。

2.2.6 JNI_CreateJavaVM 函数

创建 JVM 的程序模块是 JNI_CreateJavaVM。JNI_CreateJavaVM 主要任务是调用 Threads 模块的 create_vm() 函数，以完成最终的虚拟机创建和初始化工作。

在 Threads 模块中，实现了对虚拟机各个模块的初始化，以及创建虚拟机线程。这些被初始化的模块，在本书后续章节中均有大量涉及，因此理解这一过程对于其余章节的理解十分重要。为了保证知识的连贯性，避免打断对启动过程的叙述，我们将具体的初始过程安排在 2.3 小节中继续探讨。

注意 vm 和 env 是在 JNI_CreateJavaVM 接口中实现赋值的。

此外，JNI_CreateJavaVM 还将为 vm 和 env 分配 JNI 接口函数。

练习 6:

设置断点并调试 HotSpot，跟踪 vm 和 env 的赋值。

2.2.7 调用 Java 主方法

在 JavaMain 中，虚拟机得到初始化之后，接下来就将执行应用程序的主方法。通过 env 引用 jni_CallStaticVoidMethod 函数（原型如清单 2-26 所示），可以执行一个由“static”和“void”修饰的方法，即 Java 应用程序主类的 main 方法。

清单 2-26

来源: hotspot/src/share/vm/prims/jni.h

描述: JNI 函数: 调用静态 void 方法

```
void
CallStaticVoidMethod(jclass cls, jmethodID methodID, ...)
```

读到这里，细心的读者可能会想弄明白：由清单 2-24 (c) 和清单 2-26 可知，主方法是根据 JVM 内部一个唯一的方法 ID（即 methodID）定位到的。那么，我们不禁想问，JVM 是如何根据 methodID 定位到要执行的方法的？方法在 JVM 内部又是什么样的呢？如果你还没想过这个问题，那么请闭上眼睛，花上几分钟思考一下这个问题。

这里我们暂时不急着回答这个问题，通过本书后续章节对类的解析以及方法区等知识点的学习，这些疑惑就可以迎刃而解了。

为了执行主类的 main 方法，将在 jni_invoke_static 中通过调用 JavaCalls 模块完成最终的执行 Java 方法。在 HotSpot 中，所有对 Java 方法的调用都需要通过类 JavaCalls 来完成。

清单 2-27

来源: hotspot/src/share/vm/prims/jni.cpp

描述: JNI 函数: jni_invoke_static

```
methodHandle method(THREAD, JNIEnv::resolve_jmethod_id(method_id));
JavaCalls::call(result, method, &java_args, CHECK);
```

清单 2-27 中是这部分逻辑的实现：首先根据 method_id 转换成方法句柄，然后调用 JavaCalls 模块方法实现从 JVM 对 Java 方法的调用。

2.2.8 JVM 退出路径

前面讲述了 JVM 启动的过程，这里介绍 JVM 退出的过程。一般来说，JVM 有两条退出路径。其中一条路径称为虚拟机销毁（destroy vm）：当程序运行到主方法的结尾处，系统将调用 jni_DestroyJavaVM() 函数销毁虚拟机。而另外一条路径则是虚拟机退出（vm exit）：当程序调用 System.exit() 函数，或当 JVM 遇到错误时，将通过这条路径直接退出。

这两条退出途径并不完全相同，但它们在 Java 层共享 Shutdown.shutdown() 和 before_exit() 函数，并在 JVM 层共享 VM_Exit 函数。

这里，介绍一下 `destroy_vm` 的退出流程。

- 当前线程等待直到成为最后一条非守护线程。此时，所有工作仍在继续。
- Java 层调用 `java.lang.Shutdown.shutdown()` 函数。
- 调用 `before_exit()` 函数，为 JVM 退出做一些准备工作：首先，运行 JVM 层的关闭钩子函数（shutdown hooks）。这些钩子函数是通过 `JVM_OnExit` 进行注册的。目前唯一使用了这套机制的钩子函数是 `File.deleteOnExit()` 函数；其次，停止一些系统线程，如“StatSampler”，“watcher thread” 和“CMS threads” 等，并向 JVMTI 发送“thread end” 和“vm death” 事件；最后，停止信号线程。
- 调用 `JavaThread::exit()` 函数，这将释放 JNI 句柄块，并从线程列表中移除本线程。
- 停止虚拟机线程，使虚拟机进入安全点（safepoint）并停止编译器线程。
- 禁用 JNI/JVM 跟踪。
- 为那些仍在运行本地代码的线程设置“`_vm_exited`”标记。
- 删除当前线程。
- 调用 `exit_globals()` 函数，删除 tty 和 PerfMemory 等资源。
- 返回到上层调用者。

到目前为止，我们对启动过程已经有了较为整体的认识。接下来，在 2.3 小节中，我们将深入了解系统初始化过程。

2.3 系统初始化

前面提到，系统初始化过程是 JVM 启动过程中的重要组成部分。初始化过程涉及到绝大多数的 HotSpot 内核模块，因此，了解这个过程对于理解 HotSpot 整体架构具有重要意义。图 2-14 描述了系统初始化的完整过程。

系统初始化的具体步骤如下所示。

- (1) 初始化输出流模块；
- (2) 配置 Launcher 属性
- (3) 初始化 OS 模块；
- (4) 配置系统属性；
- (5) 程序参数和虚拟机选项解析；
- (6) 根据传入的参数继续初始化操作系统模块；
- (7) 配置 GC 日志输出流模块；
- (8) 加载代理（agent）库；
- (9) 初始化线程队列；
- (10) 初始化 TLS 模块；

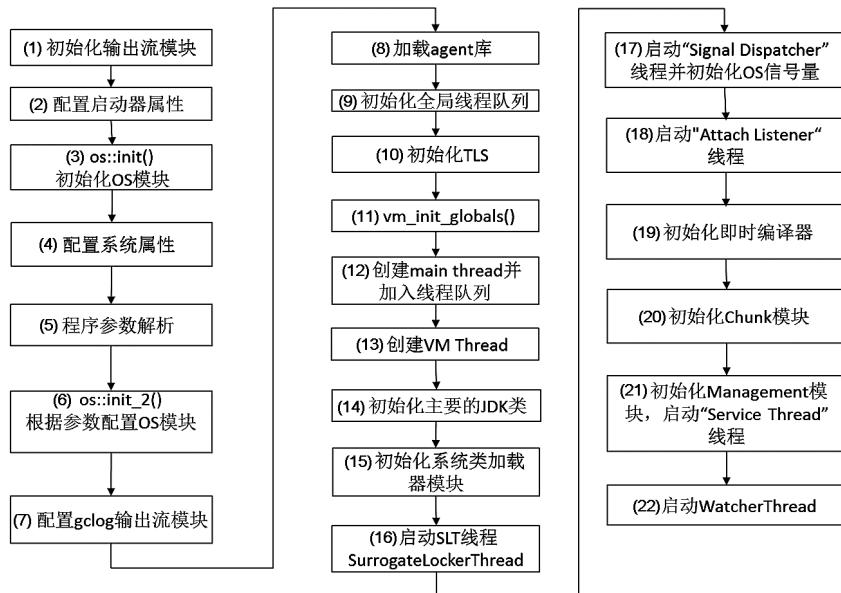


图 2-14 系统初始化流程

- (11) 调用 `vm_init_globals()` 函数初始化全局数据结构;
- (12) 创建主线程并加入线程队列;
- (13) 创建虚拟机线程;
- (14) 初始化 JDK 核心类, 如 `java.lang.String`、`java.util.HashMap`、`java.lang.System`、`java.lang.ThreadGroup`、`java.lang.Thread`、`java.lang.OutOfMemoryError` 等;
- (15) 初始化系统类加载器模块, 并初始化系统字典;
- (16) 启动 SLT 线程, 即 “`SurrogateLockerThread`” 线程;
- (17) 启动 “`Signal Dispatcher`” 线程;
- (18) 启动 “`Attach Listener`” 线程;
- (19) 初始化即时编译器;
- (20) 初始化 Chunk 模块;
- (21) 初始化 Management 模块, 启动 “`Service Thread`” 线程;
- (22) 启动 “`Watcher Thread`” 线程。

接下来, 我们将对其中几个重要的步骤做详细的讲解。

2.3.1 配置 OS 模块

OS 模块的初始化包括两个环节。

- `init()` 函数: 第一次初始化的时机是在 TLS 前, 全局参数传入之前。

- init_2()函数：第二次初始化的实际是在 args 解析后，全局参数传入之后。

1. init()函数

第一次初始化能够完成一些固定的配置，主要包括以下几项内容。

- 设置页大小。
- 设置处理器数量。
- 初始化 proc，打开 “/proc/\$pid”。
- 设置获得物理内存大小，保存在全局变量 os::Linux::_physical_memory 中；
- 获得原生主线程的句柄：获得指向原生线程的指针，并将其保存在全局变量 os::Linux::_main_thread 中。
- 系统时钟初始化：选用 CLOCK_MONOTONIC 类型时钟。从动态链接库 “librt.so.1” 或 “librt.so” 中将时钟函数 clock_gettime 装载进来，并保存在全局变量 os::Linux::_clock_gettime 中。

注意 Linux 的时钟与计时器。CLOCK_MONOTONIC 提供相对时间，它的时间值是通过 jiffies 值来计算的，jiffies 取决于系统的频率，单位是 Hz，是周期的倒数，一般表示为一秒中断产生的次数。该时钟不受系统时钟源的影响，较为稳定，只受 jiffies 值的影响。按照 POSIX 规范，与 CLOCK_MONOTONIC 相对的另外一种时钟类型是 CLOCK_REALTIME，这是系统实时时钟（RTC）。这是一个硬件时钟，用来持久存放系统时间，系统关闭后靠主板上的微型电池保持计时。系统启动时，内核通过读取 RTC 来初始化 Wall Time，并存放在 xtime 变量中，即 xtime 是从 cmos 电路中取得的时间，一般是从某一历史时刻开始到现在的时间，也就是为了取得我们操作系统上显示的日期，它的精度是微秒。

2. init_2()函数

OS 模块还有一部分配置是允许外部参数进行控制的。当解析完全局参数后，就可以根据配置参数进行配置，具体包括以下几项内容。

- 快速线程时钟初始化。
- 使用 mmap 分配共享内存，配置大页内存。
- 初始化内核信号，安装信号处理函数 SR_handler，用作线程执行过程中的 Suspended/Resumed 处理。操作系统信号（signal），作为进程间通信的一种手段，用来通知进程发生了某种类型的系统事件。
- 配置线程栈：设置栈大小、分配线程初始栈等。
- 设置文件描述符数量。
- 初始化时钟，用来串行化线程创建。

- 若开启 VM 选项 “PerfAllowAtExitRegistration”，则向系统注册 atexit 函数。
- 初始化线程优先级策略。

OS 模块初始化相关的 VM 配置、调试选项如表 2-2 所示，其中“Build”表示 VM 选项作用的 Build 版本，具体版本含义可参考 globals.hpp 中的相关定义。

表 2-2 OS 模块初始化相关的 VM 配置、调试选项

选 项	Build	默认值	描 述
-XX:+UseNUMA	product	false	使用 NUMA
-XX:+UseLargePages	pd product	true	使用大页内存
-XX:+UseSHM	product	false	使用 SYSV 共享内存
-XX:+MaxFDLimit	product	true	最大文件描述符数量
-XX:+PerfAllowAtExitRegistration	product	false	允许向系统注册 atexit 函数
-XX:+PrintMiscellaneous		false	输出未分类的调试信息（需要开启 Verbose）

练习 7

阅读源代码并调试跟踪，了解信号初始化过程，并思考 HotSpot 中初始化的信号对系统的意义。

（提示：见 SR_initialize()）

练习 8

阅读源代码并调试跟踪，了解线程优先级策略的初始化过程。

（提示：见 prio_init()）。

2.3.2 配置系统属性

配置虚拟机运行时的系统属性。首先介绍的是关于 Launcher 的属性，包括：“-Dsun.java.launcher” 和 “-Dsun.java.launcher.pid”。

接下来，要介绍的是一些与操作系统相关的系统属性，如表 2-3 所示。

表 2-3 初始化系统属性

属 性	值 示 例	读 写 属性
java.vm.specification.name	"Java Virtual Machine Specification"	只读
java.vm.version	"21.0-b17-internal-jvmg"	只读
java.vm.name	"OpenJDK Client VM"	只读
java.vm.info	"mixed mode, sharing"	读写
java.ext.dirs	NULL	读写

续表

属性	值示例	读写属性
java.endorsed.dirs	NULL	读写
sun.boot.library.path	NULL	读写
java.library.path	NULL	读写
java.home	NULL	读写
sun.boot.class.path	NULL	读写
java.class.path	""	读写
java.vm.specification.vendor	"Sun Microsystems Inc." "Oracle Corporation" (JDK 版本在 1.7x 之后)	只读
java.vm.specification.version	"1.0"	只读
java.vm.vendor	"Sun Microsystems Inc."	只读

2.3.3 加载系统库

在对虚拟机配置选项进行解析的阶段，Arguments 模块根据虚拟机选项 -agentlib 或 -agentpath，将需要加载的本地代理库逐一加入到代理库列表（AgentLibraryList）中。在加载代理库阶段，虚拟机将按照代理库列表中的库名，根据操作系统的库搜索规则，利用 OS 模块查找库并加载到虚拟机进程地址空间中。例如，若按照命令 “java -agentlib:hprof” 来启动应用程序的话，将加载 JDK 中代理库 hprof，它的库文件名为 libhprof.so 或 hprof.dll。

加载库操作需要在 Java 线程创建前完成，这样才能保证在 Java 线程需要调用时能够正确地找到本地库函数。

通过 JVMTI 接口，允许程序员开发自定义代理库，并通过选项 -agentlib 或 -agentpath 加载到虚拟机中。必须注意的是，由于 agent 代码将在虚拟机进程空间中运行，因此你的 agent 代码需要保证以下几点：多线程安全、可重入性、避免内存泄露或空指针、符合 JVMTI 和 JNI 规则等。如果你不小心触犯了这些准则，那么很有可能导致 “out of memory” 错误或虚拟机崩溃（JVM Crash，见第 4 章），这就是为什么我们在做 JVM Crash 分析时，需要考虑系统库或自定义库 bug 因素的原因。

除了 JDK 中代理库和自定义代理库，虚拟机还将加载本地库，如 libc 或 ld 库。为应用程序定位本地库可以通过两种方式：将库复制到应用程序的共享库路径下；或按照特定操作系统平台指定规则加载，如 Solaris/Linux 平台上根据环境变量 LD_LIBRARY_PATH，而在 Windows 平台上根据环境变量 PATH 来定位本地库。

在系统初始化过程中，当代理库被加载进虚拟机进程后，虚拟机将在库中查找函数符号 JVM_OnLoad 或 Agent_Onload 并调用该函数，实现代理库与虚拟机的连接。

2.3.4 启动线程

1. 线程状态和类型

在 JDK 中定义了 6 种线程状态。

- NEW：新创建但尚未启动的线程处于这种状态。通过 new 关键字创建了 java.lang.Thread 类（或其子类）的对象。
- BLOCKED：线程受阻塞并等待某个监视器对象锁。当线程执行 synchronized 方法或代码块，但未获得相对对象锁时处于这种状态。
- RUNNABLE：正在 Java 虚拟机中执行的线程处于这种状态。有三种情形，一种情形是 Thread 类的对象调用了 start() 函数，这时的线程就等待时间片轮转到自己，以便获得 CPU；另一种情形是线程在处于 RUNNABLE 状态时并没有运行完自己的 run() 函数，时间片用完之后回到 RUNNABLE 状态；还有一种情形就是处于 BLOCKED 状态的线程结束了当前的 BLOCKED 状态之后重新回到 RUNNABLE 状态。
- TERMINATED：已退出的线程处于这种状态。
- TIMED_WAITING：等待另一个线程来执行取决于指定等待时间的操作。
- WAITING：无限期地等待另一个线程来执行某一特定操作。

在 JVM 层面，HotSpot 内部定义了线程的 5 种基本状态。

- _thread_new，表示刚启动，正处在初始化过程中。
- _thread_in_native，表示运行本地代码。
- _thread_in_vm，表示在 VM 中运行。
- _thread_in_Java，表示运行 Java 代码。
- _thread_blocked，表示阻塞。

为了支持内部状态转换，还补充定义了其他几种过渡状态：`_<thread_state_type>_trans`，其中 `thread_state_type` 分别表示上述 5 种基本状态类型。

在 HotSpot 中，定义了如清单 2-28 所示的几种线程类型，其类图如 2-15 所示。

清单 2-28

来源：hotspot/src/share/vm/runtime/os.hpp

描述：线程类型

```

1 enum ThreadType {
2     vm_thread,           // VM 线程
3     cgc_thread,          // 并发 GC 线程
4     pgc_thread,          // 并行 GC 线程
5     java_thread,         // Java 线程
6     compiler_thread,     // 编译器线程
7     watcher_thread,      // watcher 线程

```

```

8     os_thread          // OS 线程
9 };

```

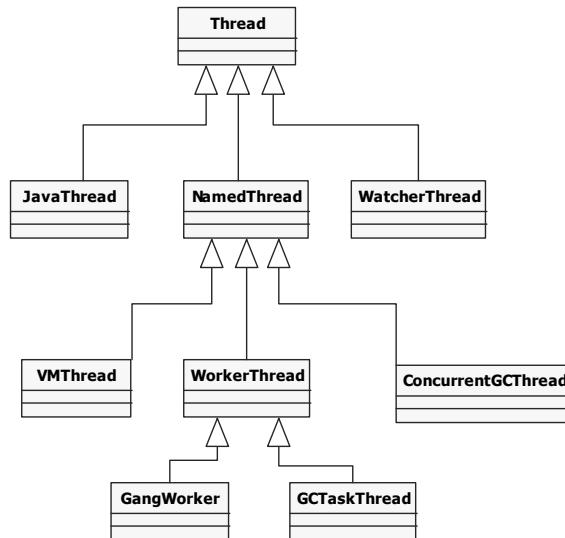


图 2-15 线程类型

2. 创建主线程

主线程（main thread）是执行应用程序的“public static void main (String[] args)”方法的线程。对应 OS 线程 ID 为 1 的即为名为“main”为主线程，如图 2-16 所示。

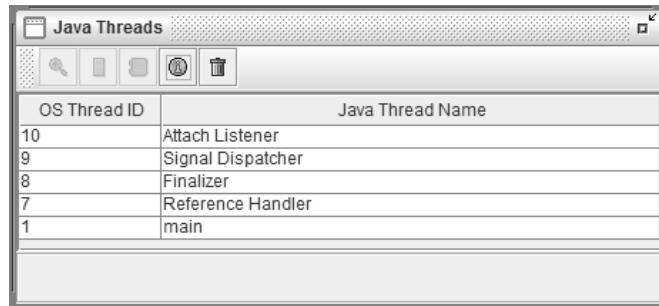


图 2-16 主线程和守护线程

系统初始化时，虚拟机首先创建的线程就是主线程。具体的创建过程如清单 2-29 所示。

清单 2-29

来源: hotspot/src/share/vm/runtime/thread.cpp - Threads::creat_vm()

描述: 创建 main thread

```

1 JavaThread* main_thread = new JavaThread();
2 main_thread->set_thread_state(_thread_in_vm);
3 main_thread->record_stack_base_and_size();

```

```

4 main_thread->initialize_thread_local_storage();
5 main_thread->set_active_handles(JNIHandleBlock::allocate_block());
6 if (!main_thread->set_as_starting_thread()) {
7     vm_shutdown_during_initialization("Failed necessary internal allocation. Out of swap
space");
8     delete main_thread;
9     *canTryAgain = false; // don't let caller call JNI_CreateJavaVM again
10    return JNI_ENOMEM;
11 }
12 main_thread->create_stack_guard_pages();

```

首先，第 1 行代码中，JVM 创建一个 JavaThread 类型的线程变量（刚创建时状态为 _thread_new）。紧接着，第 2 行将线程状态设置为 _thread_in_vm，表明该线程正处于在 JVM 中执行的状态。接下来，第 3 行记录线程栈的基址和大小；第 4 行，初始化线程本地存储区 (TLS)；第 5 行为线程设置 JNI 句柄；在第 6~11 行中，将通过 OS 模块创建原始线程，即 OS 主线程，并设置为可运行状态。接下来，在第 12 行中初始化主线程栈。

现在，main_thread 实际上是一个 JVM 内部线程，其状态为 JVM 内部定义的线程状态 _thread_in_vm。接下来需要创建 java.lang.Thread 线程：

```

13 initialize_class(vmSymbols::java_lang_System(), CHECK_0);
14 initialize_class(vmSymbols::java_lang_ThreadGroup(), CHECK_0);
15 Handle thread_group = create_initial_thread_group(CHECK_0);
16 Universe::set_main_thread_group(thread_group());
17 initialize_class(vmSymbols::java_lang_Thread(), CHECK_0);
18 oop thread_object = create_initial_thread(thread_group, main_thread, CHECK_0);
19 main_thread->set_threadObj(thread_object);
20 java_lang_Thread::set_thread_status(thread_object, java_lang_Thread::RUNNABLE);

```

当 Java 层 main 线程创建完成后，就将其状态设置为 RUNNABLE，开始运行，这样，一个 Java 主线程就开始运行了。

3. 创建 VMThread

VMThread 是在 JVM 内部执行 VMOperation 的线程。VMOperation 实现了 JVM 内部的核心操作，为其他运行时模块以及外部程序接口服务，在 HotSpot 中占有重要地位。

清单 2-30 描述了 VMThread 线程的创建过程。当 VMThread 线程创建成功后，在整个运行期间不断等待、接受并执行指定的 VMOperation。

清单 2-30

来源：hotspot/src/share/vm/runtime/thread.cpp - Threads::creat_vm()

描述：创建 VMThread

```

1 // Create the VMThread
2 { TraceTime timer("Start VMThread", TraceStartupTime);
3   VMThread::create();
4   Thread* vmthread = VMThread::vm_thread();
5
6   if (!os::create_thread(vmthread, os::vm_thread))
7     vm_exit_during_initialization("Cannot create VM thread. Out of system resources.");
8
9   // Wait for the VM thread to become ready, and VMThread::run to initialize
10  // Monitors can have spurious returns, must always check another state flag

```

```

9   {
10    MutexLocker ml(Notify_lock);
11    os::start_thread(vmthread);
12    while (vmthread->active_handles() == NULL) {
13      Notify_lock->wait();
14    }
15  }
16 }

```

4. 创建守护线程

守护线程包括“Signal Dispatcher”（该线程需要在“VMInit”事件发生前启动，详见os::signal_init()函数）、“Attach Listener”、“Watcher Thread”等。

2.3.5 vm_init_globals 函数：初始化全局数据结构

在清单 2-31 中，vm_init_globals()函数实现了对全局性数据结构的初始化。

清单 2-31

来源：hotspot/src/share/vm/runtime/intr.cpp
描述：初始化全局数据结构

```

1 void vm_init_globals() {
2   check_ThreadShadow();
3   basic_types_init();
4   eventlog_init();
5   mutex_init();
6   chunkpool_init();
7   perfMemory_init();
8 }

```

初始化的过程包括以下几个环节。

- 初始化 Java 基本类型系统。
- 分配全局事件缓存区，初始化事件队列。
- 初始化全局锁，如 iCMS_lock、FullGCCCount_lock、CMark_lock、SystemDictionary_lock、SymbolTable_lock 等，表 2-1 列举了在一些主要模块中会涉及的锁，其中有部分所可以由 VM 选项 UseConcMarkSweepGC 和 UseG1GC 控制是否开启。
- 初始化 ChunkPool，ChunkPool 包括 3 个静态 pool 链表：_large_pool、_medium_pool 和_small_pool。其实，这是 HotSpot 实现的内存池：系统全局中不会执行 malloc/free 操作，这样就能够有效避免 malloc/free 的抖动影响。内存池是系统设计的常用手段。
- 初始化 JVM 性能统计数据（Perf Data）区，可由 VM 选项 UsePerfData 控制是否开启。若开启 VM 选项 PerfTraceMemOps，可在初始化时打印该空间的分配信息。

2.3.6 init_globals 函数：初始化全局模块

如清单 2-32 所示，init_globals()函数实现了对全局模块的初始化。

清单 2-32

来源: hotspot/src/share/vm/runtime/init.cpp

描述: 初始化全局数据结构

```

1  jint init_globals() {
2      HandleMark hm;
3      management_init();
4      bytecodes_init();
5      classLoader_init();
6      codeCache_init();
7      VM_Version_init();
8      stubRoutines_init1();
9      jint status = universe_init(); // dependent on codeCache_init and stubRoutines_init
10     if (status != JNI_OK)
11         return status;
12
13     interpreter_init(); // before any methods loaded
14     invocationCounter_init(); // before any methods loaded
15     marksweep_init();
16     accessFlags_init();
17     templateTable_init();
18     InterfaceSupport_init();
19     SharedRuntime::generate_stubs();
20     universe2_init(); // dependent on codeCache_init and stubRoutines_init
21     referenceProcessor_init();
22     jni_handles_init();
23     vtableStubs_init();
24     InlineCacheBuffer_init();
25     compilerOracle_init();
26     compilationPolicy_init();
27     VMRegImpl::set_regName();
28
29     if (!universe_post_init())
30         return JNI_ERR;
31     javaClasses_init(); // must happen after vtable initialization
32     stubRoutines_init2(); // note: StubRoutines need 2-phase init
33
34     if (VerifyBeforeGC && !UseTLAB &&
35         Universe::heap()->total_collections() >= VerifyGCStartAt) {
36         Universe::heap()->prepare_for_verify();
37         Universe::verify(); // make sure we're starting with a clean slate
38     }
39
40     return JNI_OK;
41 }
```

这些模块构成了 HotSpot 整体功能的基础，也是本书后续章节所要探讨的核心内容。接下来，我们将对其中几个较为关键的内核模块做一个概念性的了解。

1. JMX: Management 模块

在 HotSpot 工程结构中，我们了解到 Services 模块为 JVM 提供 JMX 等功能，JMX 功能又可划分为如下 4 个主要模块，如图 2-17 所示。

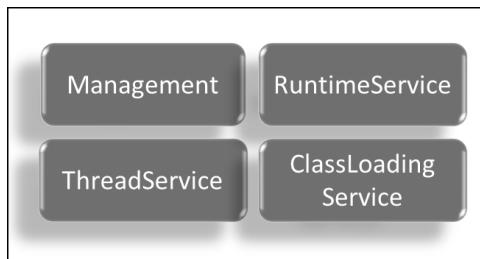


图 2-17 Services 模块主要组成

- Management 模块：启动名为“Service Thread”的守护线程（如清单 2-33 所示），注意在较早的版本中该守护线程名为“Low Memory Detector”。若系统开启了选项 -XX:ManagementServer，则加载并创建 sun.management.Agent 类，执行其 startAgent() 方法启动 JMX Server。
- RuntimeService 模块：提供运行时模块的性能监控和管理服务，如 applicationTime、jvmCapabilities 等。
- ThreadService 模块：提供线程和内部同步系统的性能监控和管理服务，包括维护线程列表、线程相关的性能统计、线程快照、线程堆栈跟踪和线程转储等功能。
- ClassLoadingService：提供类加载模块的性能监控和管理服务。

清单 2-33

```

"Service      Thread"    daemon    prio=6      tid=0x000000000b062000    nid=0x7274    runnable
[0x0000000000000000]
          java.lang.Thread.State: RUNNABLE

```

在 JVM 初始化时，会相继对这 4 个模块进行初始化，如清单 2-34 所示。

清单 2-34

来源: hotspot/src/share/vm/runtime/init.cpp
描述: 初始化 Management 模块

```

1 void management_init() {
2     Management::init();
3     ThreadService::init();
4     RuntimeService::init();
5     ClassLoadingService::init();
6 }

```

2. Code Cache

Code Cache 是指代码高速缓存，主要用来生成和存储本地代码。这些代码片段包括已编译好的 Java 方法和 RuntimeStubs 等。

通过 VM 选项 CodeCacheExpansionSize、InitialCodeCacheSize 和 ReservedCodeCacheSize 可以配置该空间大小。

此外，若在 Windows 64 位平台上开启 SHE 机制³（即通过 VM 选项 UseVectoredExceptions 关闭 Vectored Exceptions 机制⁴，默认关闭），则需要向 OS 模块注册 SHE。

3. StubRoutines

StubRoutines 位于运行时模块。该模块的初始化分为两个阶段，第一阶段初始化（stubRoutines_init1），将创建一个名为“StubRoutines (1)”的 BufferBlob，并未其分配 CodeBuffer 存储空间，并初始化 StubRoutines。在第二阶段（stubRoutines_init2）中，创建名为“StubRoutines (2)”的 BufferBlob，并为其分配 CodeBuffer 存储空间。并生成所有 stubs 并初始化 entry points。

4. Universe

Universe 模块将按照两个阶段进行初始化。第一阶段，根据 VM 选项配置的 GC 策略及算法，选择垃圾收集器和堆的种类，初始化堆。根据 VM 选项 UseCompressedOops 进行相关配置。若 VM 选项 UseTLAB 开启 TLAB，则初始化 TLAB 缓存区。第二阶段，将对共享空间进行配置以及初始化 vmSymbols 和 SystemDictionary 等全局数据结构。

5. 解释器

位于解释器模块。初始化解释器（interpreter），并注册 StubQueue。可开启 VM 选项 TraceBytecodes 跟踪。

6. 模板表

同样位于解释器模块。初始化模板表模块，将创建模版解释器使用的模板表，更多解释器内容请参考本书第 7 章。

7. stubs

位于运行时模块。在系统启动时，创建供各个运行时组件共享的 stubs 模块，诸如“wrong_method_stub”、“ic_miss_stub”、“resolve_opt_virtual_call”、“resolve_virtual_call”、“resolve_static_call”等。

除了上述模块，init_globals 还将对下面这些模块进行初始化：字节码模块 Bytecodes、类加载器模块 ClassLoader、虚拟机版本模块，以及 ReferenceProcessor、JNIHandles、VtableStubs、InlineCacheBuffer、VMRegImpl、JavaClasses 等模块。这些模块的作用和实现，在本书后续章节将会陆续看到。

³ 更多内容，可以参考 [http://msdn.microsoft.com/en-us/library/windows/desktop/ms680657\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ms680657(v=vs.85).aspx)。

⁴ 关于 Structured Exception Handlers，详见 [http://msdn.microsoft.com/en-us/library/ms681420\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms681420(VS.85).aspx)。

练习 9

阅读源代码并调试跟踪 HotSpot 的“初始化全局模块”，建议整理一份分析报告。

2.4 小结

本章首先介绍了 HotSpot 内核的结构，并引导读者掌握一些阅读源代码的技巧。在内核模块中，介绍了 Prims、Service 和 Runtime 模块，它们为 HotSpot 提供外部接口，并为内核其他模块提供部分公共功能。

启动过程是了解 HoSpot 内部实现的入口。HotSpot 提供了两种启动器，一个是产品级的，另一个则是调试级的。后者对于我们调试和学习 HotSpot 起到重要的作用。在整个启动过程中，`create_vm()` 函数是其精华部分，它完成了 JVM 系统绝大多数模块的初始化工作。

为了帮助读者打好独立阅读、分析源代码的基础，我们还需要讲解更多的知识。在下一章中，我们将接触到 HotSpot 内部的面向对象表示系统，它是贯穿于整个 HotSpot 内核的脉络。可以说，这部分的知识已渗透到 HotSpot 中方方面面的业务逻辑中。因此，对于我们来说，掌握好这部分知识是十分必要的。

HotSpot实战

HotSpot是一款高性能的Java虚拟机，可以大大提高Java运行的性能。Java原先是把源代码编译为字节码在虚拟机执行，这样整体执行效率不高。而HotSpot关注的是对部分热点（hot spot）代码的动态优化，将那些频繁执行的热点代码编译为本地原生代码，这样就显著地提高了性能。

陈涛是网易宝的核心开发人员之一，同时维护了网易宝的多个系统。网易宝是网易官方的在线支付系统，对开发工程师的技术要求极高。他喜欢专研技术，知识面宽，不仅对Java、C、C++熟悉，对操作系统底层也很熟悉，能够将理论很好地应用于实践中。本书便是他潜心研究和实践的成果。

赵刚 网易宝系统负责人 资深技术专家

陈涛在Java开发领域知识的深度和广度给我留下了深刻的印象。本书深入浅出地介绍了JVM技术，强调实践应用。对于想深入研究JVM并希望快速取得进展的开发人员来说，本书非常具有实用价值，是Java开发水平更上一层楼的阶石。

陈双辉 现任通策集团信息事业部CTO
曾在摩托罗拉移动担任Senior PM

为了环保，本书不提供光盘，如果想下载本书的相关源代码，可以访问
<http://www.ptpress.com.cn/Resources.aspx>。

更多关于HotSpot的信息可以访问 [OpenJDK](#)。



分类建议：计算机／程序设计／Java

人民邮电出版社网址：www.ptpress.com.cn

封面设计：任文杰

本书内容

- 初识HotSpot
- 启动
- 类与对象
- 运行时数据区
- 垃圾收集
- 栈
- 解释器和即时编译器
- 指令集
- 虚拟机监控工具

ISBN 978-7-115-34363-5

9 787115 343635 >