



实战Java虚拟机

JVM故障诊断与性能优化

葛一鸣 | 著

通过200余示例详解Java虚拟机的各种参数配置、故障排查、性能监控及优化
技术全面，通俗易懂

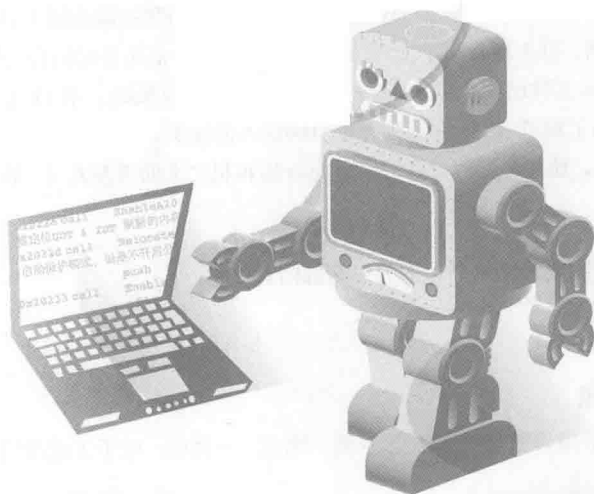


中国工信出版集团



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

51CTO学院系列丛书



实战Java虚拟机

JVM故障诊断与性能优化

葛一鸣 | 著

電子工業出版社

Publishing House of Electronics Industry

内 容 简 介

随着越来越多的第三方语言（Groovy、Scala、JRuby 等）在 Java 虚拟机上运行，Java 也俨然成为了一个充满活力的生态圈。本书将通过 200 余示例详细介绍 Java 虚拟机中的各种参数配置、故障排查、性能监控以及性能优化。

本书共 11 章。第 1~3 章介绍了 Java 虚拟机的定义、总体架构、常用配置参数。第 4~5 章介绍了垃圾回收的算法和各种垃圾回收器。第 6 章介绍了 Java 虚拟机的性能监控和故障诊断工具。第 7 章详细介绍了对 Java 堆的分析方法和案例。第 8 章介绍了 Java 虚拟机对多线程，尤其是对锁的支持。第 9~10 章介绍了 Java 虚拟机的核心——Class 文件结构，以及 Java 虚拟机中类的装载系统。第 11 章介绍了 Java 虚拟机的执行系统和字节码，并给出了通过 ASM 框架进行字节码注入的案例。

本书不仅适合 Java 程序员，还适合任何一名工作于 Java 虚拟机之上的研发人员、软件设计师、架构师。

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

图书在版编目（CIP）数据

实战 Java 虚拟机：JVM 故障诊断与性能优化 / 葛一鸣著. —北京：电子工业出版社，2015.3
（51CTO 学院系列丛书）
ISBN 978-7-121-25612-7

I. ①实… II. ①葛… III. ①JAVA 语言—程序设计 IV. ①TP312

中国版本图书馆 CIP 数据核字（2015）第 040500 号

责任编辑：董 英

印 刷：北京中新伟业印刷有限公司

装 订：三河市华成印务有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱

邮编：100036

开 本：787×980 1/16

印张：28.25

字数：633 千字

版 次：2015 年 3 月第 1 版

印 次：2015 年 3 月第 1 次印刷

印 数：3000 册 定价：79.00 元

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

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

服务热线：（010）88258888。

前 言

关于 Java 生态圈

Java 是目前应用最为广泛的软件开发平台之一。随着 Java 以及 Java 社区的不断壮大, Java 也早已不再是简简单单的一门计算机语言了, 它更是一个平台、一种文化、一个社区。

作为一个平台, Java 虚拟机扮演着举足轻重的作用。除了 Java 语言, 任何一种能够被编译成字节码的计算机语言都属于 Java 这个平台。Groovy、Scala、JRuby 等都是 Java 平台的一个部分, 它们依赖于 Java 虚拟机, 同时, Java 平台也因为它们变得更加丰富多彩。

作为一种文化, Java 几乎成为了“开源”的代名词。在 Java 程序中, 有着数不清的开源软件和框架, 如 Tomcat、Struts、Hibernate、Spring 等。就连 JDK 和 JVM 自身也有不少开源的实现, 如 OpenJDK、Harmony。可以说, “共享”的精神在 Java 世界里体现得淋漓尽致。

作为一个社区, Java 拥有无数的开发人员, 有数不清的论坛和资料。从桌面应用软件、嵌入式开发到企业级应用、后台服务器、中间件, 都可以看到 Java 的身影。其应用形式之复杂、参与人数之众多也令人咋舌。可以说, Java 社区已经俨然成为了一个良好而庞大的生态系统。

而本书, 将主要介绍这个生态系统的核心——Java 虚拟机。

本书的体系结构

本书立足于实际开发, 又不缺乏理论介绍, 力求通俗易懂、循序渐进。本书共分为 11 章:

第 1 章主要为综述，介绍了 Java 虚拟机的概念、定义，讲解了 Java 语言规范和 Java 虚拟机规范，最后，还介绍了 OpenJDK 的调试方法。

第 2 章介绍了 Java 虚拟机的总体架构，说明了堆、栈、方法区等内存空间的作用和彼此之间的联系。

第 3 章介绍了 Java 虚拟机的常用配置参数，重点对垃圾回收跟踪参数、内存配置参数做了详细的介绍，并给出了案例说明。

第 4 章从理论层面介绍了垃圾回收的算法，如引用计数、标记清除、标记压缩、复制算法等。本章是第 5 章的理论基础。

第 5 章讲解了基于垃圾回收的理论知识，进一步详细介绍了 Java 虚拟机中实际使用的各种垃圾回收器，包括串行回收器、并行回收器、CMS、G1 等。

第 6 章介绍了 Java 虚拟机的性能监控和故障诊断工具，考虑到实用性，也介绍了系统级性能监控工具的使用，两者结合，可以更好地帮助读者处理实际问题。

第 7 章详细介绍了对 Java 堆的分析方法和案例，主要讲解了 MAT 和 Visual VM 两款工具的使用，以及各自 OQL 的编写方式。

第 8 章介绍了 Java 虚拟机对多线程，尤其是对锁的支持，本章不仅介绍了虚拟机内部锁的实现、优化机制，也给出了一些 Java 语言层面的锁优化思路，最后，还介绍了无锁的并行控制方法。

第 9 章介绍了 Java 虚拟机的核心——Class 文件结构，Class 文件作为 Java 虚拟机的基石，有着举足轻重的作用，对深入理解 Java 虚拟机有着不可忽视的作用。

第 10 章介绍了 Java 虚拟机中类的装载系统，其中，着重介绍了 Java 虚拟机中 ClassLoader 的实现以及设计模式。

第 11 章介绍了 Java 虚拟机的执行系统和字节码，为了帮助读者更快更好地理解 Java 字节码，本章对字节码进行了分类讲解，并且理论联系实际，给出了通过 ASM 框架进行字节码注入的案例。

本书特色

本书的主要特点有：

1. **结构清晰。**本书采用从整体到局部的视角，首先第 1、2 章介绍了 Java 虚拟机的整体概况和结构。接着步步为营，每一章节对应一个单独的知识点，力求展示虚拟机的全貌。

2. **理论结合实战。**本书不甘心于简单地枚举理论知识，在每一个理论背后，都给出了演示示例供读者参考，帮助读者更好地消化这些理论。比如，在对 Class 文件结构和字节码的介绍中，不仅仅简单地给出了理论说明，更是使用 ASM 框架将这些理论应用于实践，尽可能地做到理论和实践结合。
3. **专注专业。**本书着眼于 Java 虚拟机，对 Java 虚拟机的原理和实践做了丰富的介绍，包括但不限于体系结构、虚拟机的调试方式、常用参数、垃圾回收系统、Class 文件结构、执行系统等，力求从多角度更专业地对 Java 虚拟机进行探讨。
4. **通俗易懂。**本书依然服务于广大虚拟机初学者，尽量避免采用过于理论的描述方式，简单的白话文风格贯穿全书，尽量做到读者在阅读过程中少盲点、无盲点。
5. **技术全面。**纵横 Windows 和 Linux 双系统下的性能诊断、涉及 32 位系统和 64 位系统的优化比较、贯穿从 JDK 1.5 到 JDK 1.8 的优化演进。

适合阅读人群

虽然本书力求通俗，但要通读本书并取得良好的学习效果，要求读者需要具备基本的 Java 知识或者一定的编程经验。因此，本书适合以下读者：

- 拥有一定开发经验的 Java 平台开发人员（Java、Scala、JRuby 等）
- 软件设计师、架构师
- 系统调优人员
- 有一定的 Java 编程基础并希望进一步理解 Java 的程序员
- 虚拟机爱好者，JVM 实践者

本书的约定

本书在叙述过程中，有如下约定：

- 本书中所述的 JDK 1.5、JDK 1.6、JDK 1.7、JDK 1.8 等同于 JDK 5、JDK 6、JDK 7、JDK 8。
- 如无特殊说明，Java 虚拟机均指 HotSpot 虚拟机。
- 如无特殊说明，本书的程序、示例均在 JDK 1.7 环境中运行。
- 本书赠送的课程优惠券，可以观看笔者在 51CTO 学院的 JVM 课程，地址是 http://edu.51cto.com/course/course_id-1952.html。

联系作者

本书的写作过程远比我想象的更艰辛，为了让全书能够更清楚、更正确地表达和论述，我经历了好多个不眠之夜，即使现在回想起来，也忍不住让我打个寒战。由于写作水平的限制，书中难免会有不妥之处，望读者谅解。

为此，如果读者有任何疑问或者建议，非常欢迎大家加入 QQ 群 397196583，一起探讨学习中的困难、分享学习的经验，我期待与大家一起交流、共同进步。同时，也希望大家可以关注我的博客 <http://www.uucode.net/>。

感谢

这本书能够面世，是因为得到了众人的支持。首先，要感谢我的妻子，她始终不辞辛劳，毫无怨言地对我照顾有加，才让我得以腾出大量时间，并可以安心工作。其次，要感谢小编为我一次又一次地审稿改错，批评指正，才能让本书逐步完善。最后，感谢我的母亲 30 年如一日对我的体贴和关心。

参与本书编写的还有宋玉红、关硕、安继宏、白慧、薛淑英、蒋玺、曹静、马玉杰、陈明、张丽萍、任娜娜、李清艺、荆海霞、赵全利、孙迪，特此感谢！

葛一鸣

目 录

第 1 章 初探 Java 虚拟机.....	1
1.1 知根知底：追溯 Java 的发展历程.....	2
1.1.1 那些依托 Java 虚拟机的语言大咖们	2
1.1.2 Java 发展史上的里程碑	2
1.2 跨平台的真相：Java 虚拟机来做中介.....	4
1.2.1 理解 Java 虚拟机的原理	4
1.2.2 看清 Java 虚拟机的种类	5
1.3 一切看我的：Java 语言规范	6
1.3.1 词法的定义	6
1.3.2 语法的定义	7
1.3.3 数据类型的定义	8
1.3.4 Java 语言规范总结	9
1.4 一切听我的：Java 虚拟机规范	9
1.5 数字编码就是计算机世界的水和电.....	10
1.5.1 整数在 Java 虚拟机中的表示	10
1.5.2 浮点数在 Java 虚拟机中的表示	12
1.6 抛砖引玉：编译和调试虚拟机.....	14
1.7 小结	19
第 2 章 认识 Java 虚拟机的基本结构.....	20
2.1 谋全局者才能成大器：看穿 Java 虚拟机的架构.....	20

2.2	小参数能解决大问题：学会设置 Java 虚拟机的参数.....	22
2.3	对象去哪儿：辨清 Java 堆	23
2.4	函数如何调用：出入 Java 栈	25
2.4.1	局部变量表.....	27
2.4.2	操作数栈.....	32
2.4.3	帧数据区.....	32
2.4.4	栈上分配.....	33
2.5	类去哪儿了：识别方法区.....	35
2.6	小结	37
第 3 章	常用 Java 虚拟机参数.....	38
3.1	一切运行都有迹可循：掌握跟踪调试参数.....	38
3.1.1	跟踪垃圾回收——读懂虚拟机日志.....	39
3.1.2	类加载/卸载的跟踪.....	42
3.1.3	系统参数查看	44
3.2	让性能飞起来：学习堆的配置参数.....	45
3.2.1	最大堆和初始堆的设置.....	45
3.2.2	新生代的配置.....	49
3.2.3	堆溢出处理.....	52
3.3	别让性能有缺口：了解非堆内存的参数配置.....	54
3.3.1	方法区配置.....	55
3.3.2	栈配置.....	55
3.3.3	直接内存配置.....	55
3.4	Client 和 Server 二选一：虚拟机的工作模式.....	58
3.5	小结	59
第 4 章	垃圾回收概念与算法	60
4.1	内存管理清洁工：认识垃圾回收.....	60
4.2	清洁工具大 PK：讨论常用的垃圾回收算法.....	61
4.2.1	引用计数法（Reference Counting）	62
4.2.2	标记清除法（Mark-Sweep）	63
4.2.3	复制算法（Copying）	64

4.2.4	标记压缩法 (Mark-Compact)	66
4.2.5	分代算法 (Generational Collecting)	67
4.2.6	分区算法 (Region)	68
4.3	谁才是真正的垃圾: 判断可触及性	69
4.3.1	对象的复活	69
4.3.2	引用和可触及性的强度	71
4.3.3	软引用——可被回收的引用	72
4.3.4	弱引用——发现即回收	76
4.3.5	虚引用——对象回收跟踪	77
4.4	垃圾回收时的停顿现象: Stop-The-World 案例实战	79
4.5	小结	83
第 5 章	垃圾收集器和内存分配	84
5.1	一心一意一件事: 串行回收器	85
5.1.1	新生代串行回收器	85
5.1.2	老年代串行回收器	86
5.2	人多力量大: 并行回收器	86
5.2.1	新生代 ParNew 回收器	87
5.2.2	新生代 ParallelGC 回收器	88
5.2.3	老年代 ParallelOldGC 回收器	89
5.3	一心多用都不落下: CMS 回收器	90
5.3.1	CMS 主要工作步骤	90
5.3.2	CMS 主要的设置参数	91
5.3.3	CMS 的日志分析	92
5.3.4	有关 Class 的回收	94
5.4	未来我做主: G1 回收器	95
5.4.1	G1 的内存划分和主要收集过程	95
5.4.2	G1 的新生代 GC	96
5.4.3	G1 的并发标记周期	97
5.4.4	混合回收	100
5.4.5	必要时的 Full GC	102
5.4.6	G1 日志	102

5.4.7	G1 相关的参数.....	106
5.5	回眸：有关对象内存分配和回收的一些细节问题.....	107
5.5.1	禁用 System.gc().....	107
5.5.2	System.gc()使用并发回收.....	107
5.5.3	并行 GC 前额外触发的新生代 GC.....	109
5.5.4	对象何时进入老年代.....	110
5.5.5	在 TLAB 上分配对象.....	117
5.5.6	方法 finalize()对垃圾回收的影响.....	120
5.6	温故又知新：常用的 GC 参数.....	125
5.7	动手才是真英雄：垃圾回收器对 Tomcat 性能影响的实验.....	127
5.7.1	配置实验环境.....	127
5.7.2	配置进行性能测试的工具 JMeter.....	128
5.7.3	配置 Web 应用服务器 Tomcat.....	131
5.7.4	实战案例 1——初试串行回收器.....	133
5.7.5	实战案例 2——扩大堆以提升系统性能.....	133
5.7.6	实战案例 3——调整初始堆大小.....	134
5.7.7	实战案例 4——使用 ParrellOldGC 回收器.....	135
5.7.8	实战案例 5——使用较小堆提高 GC 压力.....	135
5.7.9	实战案例 6——测试 ParallelOldGC 的表现.....	135
5.7.10	实战案例 7——测试 ParNew 回收器的表现.....	136
5.7.11	实战案例 8——测试 JDK 1.6 的表现.....	136
5.7.12	实战案例 9——使用高版本虚拟机提升性能.....	137
5.8	小结.....	137
第 6 章	性能监控工具.....	138
6.1	有我更高效：Linux 下的性能监控工具.....	139
6.1.1	显示系统整体资源使用情况——top 命令.....	139
6.1.2	监控内存和 CPU——vmstat 命令.....	140
6.1.3	监控 IO 使用——iostat 命令.....	142
6.1.4	多功能诊断器——pidstat 工具.....	143
6.2	用我更高效：Windows 下的性能监控工具.....	148
6.2.1	任务管理器.....	148

6.2.2	perfinon 性能监控工具	150
6.2.3	Process Explorer 进程管理工具	153
6.2.4	pslist 命令——Windows 下也有命令行工具	155
6.3	外科手术刀: JDK 性能监控工具	157
6.3.1	查看 Java 进程——jps 命令	158
6.3.2	查看虚拟机运行时信息——jstat 命令	159
6.3.3	查看虚拟机参数——jinfo 命令	162
6.3.4	导出堆到文件——jmap 命令	163
6.3.5	JDK 自带的堆分析工具——jhat 命令	165
6.3.6	查看线程堆栈——jstack 命令	167
6.3.7	远程主机信息收集——jstatd 命令	170
6.3.8	多功能命令行——jcmd 命令	172
6.3.9	性能统计工具——hprof	175
6.3.10	扩展 jps 命令	177
6.4	我是你的眼: 图形化虚拟机监控工具 JConsole	178
6.4.1	JConsole 连接 Java 程序	178
6.4.2	Java 程序概况	179
6.4.3	内存监控	180
6.4.4	线程监控	180
6.4.5	类加载情况	182
6.4.6	虚拟机信息	182
6.5	一目了然: 可视化性能监控工具 Visual VM	183
6.5.1	Visual VM 连接应用程序	184
6.5.2	监控应用程序概况	185
6.5.3	Thread Dump 和分析	186
6.5.4	性能分析	187
6.5.5	内存快照分析	189
6.5.6	BTrace 介绍	190
6.6	来自 JRockit 的礼物: 虚拟机诊断工具 Mission Control	198
6.6.1	MBean 服务器	198
6.6.2	飞机记录器 (Flight Recorder)	200
6.7	小结	203

第 7 章 分析 Java 堆.....	204
7.1 对症下药：找到内存溢出的原因.....	205
7.1.1 堆溢出.....	205
7.1.2 直接内存溢出.....	205
7.1.3 过多线程导致 OOM.....	207
7.1.4 永久区溢出.....	209
7.1.5 GC 效率低下引起的 OOM.....	210
7.2 无处不在的字符串：String 在虚拟机中的实现.....	210
7.2.1 String 对象的特点.....	210
7.2.2 有关 String 的内存泄漏.....	212
7.2.3 有关 String 常量池的位置.....	215
7.3 虚拟机也有内窥镜：使用 MAT 分析 Java 堆.....	217
7.3.1 初识 MAT.....	217
7.3.2 浅堆和深堆.....	220
7.3.3 例解 MAT 堆分析.....	221
7.3.4 支配树（Dominator Tree）.....	225
7.3.5 Tomcat 堆溢出分析.....	226
7.4 筛选堆对象：MAT 对 OQL 的支持.....	230
7.4.1 Select 子句.....	230
7.4.2 From 子句.....	232
7.4.3 Where 子句.....	234
7.4.4 内置对象与方法.....	234
7.5 更精彩的查找：Visual VM 对 OQL 的支持.....	239
7.5.1 Visual VM 的 OQL 基本语法.....	239
7.5.2 内置 heap 对象.....	240
7.5.3 对象函数.....	242
7.5.4 集合/统计函数.....	247
7.5.5 程序化 OQL 分析 Tomcat 堆.....	252
7.6 小结.....	255
第 8 章 锁与并发.....	256
8.1 安全就是锁存在的理由：锁的基本概念和实现.....	257

8.1.1	理解线程安全	257
8.1.2	对象头和锁	259
8.2	避免残酷的竞争：锁在 Java 虚拟机中的实现和优化	260
8.2.1	偏向锁	260
8.2.2	轻量级锁	262
8.2.3	锁膨胀	263
8.2.4	自旋锁	264
8.2.5	锁消除	264
8.3	应对残酷的竞争：锁在应用层的优化思路	266
8.3.1	减少锁持有时间	266
8.3.2	减小锁粒度	267
8.3.3	锁分离	269
8.3.4	锁粗化	271
8.4	无招胜有招：无锁	273
8.4.1	理解 CAS	273
8.4.2	原子操作	274
8.4.3	新宠儿 LongAddr	277
8.5	将随机变为可控：理解 Java 内存模型	280
8.5.1	原子性	280
8.5.2	有序性	282
8.5.3	可见性	284
8.5.4	Happens-Before 原则	286
8.6	小结	286
第 9 章	Class 文件结构	287
9.1	不仅跨平台，还能跨语言：语言无关性	287
9.2	虚拟机的基石：Class 文件	289
9.2.1	Class 文件的标志——魔数	290
9.2.2	Class 文件的版本	292
9.2.3	存放所有常数——常量池	293
9.2.4	Class 的访问标记（Access Flag）	300
9.2.5	当前类、父类和接口	301

9.2.6	Class 文件的字段	302
9.2.7	Class 文件的方法基本结构	304
9.2.8	方法的执行主体——Code 属性	306
9.2.9	记录行号——LineNumberTable 属性	307
9.2.10	保存局部变量和参数——LocalVariableTable 属性	308
9.2.11	加快字节码校验——StackMapTable 属性	308
9.2.12	Code 属性总结	313
9.2.13	抛出异常——Exceptions 属性	314
9.2.14	用实例分析 Class 的方法结构	315
9.2.15	我来自哪里——SourceFile 属性	318
9.2.16	强大的动态调用——BootstrapMethods 属性	319
9.2.17	内部类——InnerClasses 属性	320
9.2.18	将要废弃的通知——Deprecated 属性	321
9.2.19	Class 文件总结	322
9.3	操作字节码：走进 ASM	322
9.3.1	ASM 体系结构	322
9.3.2	ASM 之 Hello World	324
9.4	小结	325
第 10 章	Class 装载系统	326
10.1	来去都有序：看懂 Class 文件的装载流程	326
10.1.1	类装载的条件	327
10.1.2	加载类	330
10.1.3	验证类	332
10.1.4	准备	333
10.1.5	解析类	334
10.1.6	初始化	336
10.2	一切 Class 从这里开始：掌握 ClassLoader	340
10.2.1	认识 ClassLoader，看懂类加载	341
10.2.2	ClassLoader 的分类	341
10.2.3	ClassLoader 的双亲委托模式	343
10.2.4	双亲委托模式的弊端	347

10.2.5	双亲委托模式的补充	348
10.2.6	突破双亲模式	350
10.2.7	热替换的实现	353
10.3	小结	357
第 11 章	字节码执行	358
11.1	代码如何执行：字节码执行案例	359
11.2	执行的基础：Java 虚拟机常用指令介绍	369
11.2.1	常量入栈指令	369
11.2.2	局部变量压栈指令	370
11.2.3	出栈装入局部变量表指令	371
11.2.4	通用型操作	372
11.2.5	类型转换指令	373
11.2.6	运算指令	375
11.2.7	对象/数组操作指令	377
11.2.8	比较控制指令	379
11.2.9	函数调用与返回指令	386
11.2.10	同步控制	389
11.2.11	再看 Class 的方法结构	391
11.3	更上一层楼：再看 ASM	393
11.3.1	为类增加安全控制	393
11.3.2	统计函数执行时间	396
11.4	谁说 Java 太刻板：Java Agent 运行时修改类	399
11.4.1	使用 -javaagent 参数启动 Java 虚拟机	400
11.4.2	使用 Java Agent 为函数增加计时功能	402
11.4.3	动态重转换类	404
11.4.4	有关 Java Agent 的总结	407
11.5	与时俱进：动态函数调用	407
11.5.1	方法句柄使用实例	407
11.5.2	调用点使用实例	411
11.5.3	反射和方法句柄	412
11.5.4	指令 invokedynamic 使用实例	414

11.6	跑得再快点：静态编译优化.....	418
11.6.1	编译时计算.....	419
11.6.2	变量字符串的连接.....	421
11.6.3	基于常量的条件语句裁剪.....	422
11.6.4	switch 语句的优化.....	423
11.7	提高虚拟机的执行效率：JIT 及其相关参数.....	424
11.7.1	开启 JIT 编译.....	425
11.7.2	JIT 编译阈值.....	426
11.7.3	多级编译器.....	427
11.7.4	OSR 栈上替换.....	430
11.7.5	方法内联.....	431
11.7.6	设置代码缓存大小.....	432
11.8	小结.....	436

1

第 1 章

初探 Java 虚拟机

什么是 Java 虚拟机？什么是 Java 语言？两者又有何关系？作为本书开篇之章，本章将主要介绍有关 Java 虚拟机的基本概念、发展历史和实现概要。其中，将重点介绍支撑 Java 世界的两份重要规范——Java 语言规范和 Java 虚拟机规范，帮助读者更好地理解 Java 生态圈。

本章涉及的主要知识点有：

- 读懂 Java 的发展历史。
- 学习 Java 虚拟机的概念和种类。
- 接触 Java 语言规范。
- 了解 Java 虚拟机规范。
- 掌握单步调试 Java 虚拟机的方法。

PDF电子书说明：

本人可以提供各种PDF电子书资料，计算机类，文学，艺术，设计，医学，理学，经济，金融，等等。质量都很清晰，而且每本100%都带书签和目录，方便读者阅读观看，只要您提供给我书的相关信息，一般我都能找到，如果您有需求，请联系我 **QQ: 461573687**, 或者 **QQ: 2404062482**。

本人已经帮助了上万人找到了他们需要的PDF，其实网上有很多PDF,大家如果在网上不到的话，可以联系我QQ。因PDF电子书都有版权，请不要随意传播，最近pdf也越来越难做了，希望大家尊重下个人劳动，谢谢！

1.1 知根知底：追溯 Java 的发展历程

目前，Java 语言可以说是最常用的编程语言之一，在应用软件领域，它唯一的竞争对手似乎只有微软的 .NET。C/C++ 作为曾经的霸主，目前依然占据着系统软件和嵌入式系统绝对的市场份额，但正在逐步退出应用软件领域。和 C/C++ 相比，Java 在设计上有着绝对的优势，开发人员可以尽快从语言本身的复杂性中解脱出来，将更多的精力投向软件自身的业务功能。由于 Java 语言的这份简单性，也可以认为 Java 是一门极好的初学者入门语言。

但是，人无完人，Java 在不少地方依然受到了广大开发人员的诟病，它烦琐的语法经常受到 Python 等开发人员的耻笑。在语言的动态性上，甚至也远远不如和它年龄相仿的 PHP 语言。但为了支持动态语言，Java 虚拟机推出了新的函数调用指令 `invokedynamic`（本书将在第 11 章中具体介绍该指令），试图弥补 Java 在动态调用上的不足。

因此，值得欣慰，到目前为止，Java 仍然处于快速发展期，不断壮大，不断完善。

1.1.1 那些依托 Java 虚拟机的语言大咖们

无论受到多少非议，Java 的崛起已经是不争的事实。想起《康熙王朝》中的对白，哪一位千古帝王、功臣名将不是“褒满天下，谤满天下”。而且万幸的是，Java 生态系统极具活力，在 Java 8 中，已经推出了函数式编程语法，试图简化 Java 语言在语法上的诟病。如果你不喜欢这种新的语法也没关系，Clojure 语言作为 Lisp 的方言，可以很好地在 Java 虚拟机上执行。如果你受不了 Lisp 形式的怪异语法，Jython 已经可以将 Python 运行在 Java 虚拟机上。如果你只需要一个简单的脚本，Groovy 也可以成为你的选择。哦，对了，还有 Scala，专注于高并发的解决方案。在这里，你可以找到你需要的一切。

而所有这一切，仍然正在不断地蓬勃发展，它们和那个看似呆板的 Java 语言渐行渐远，但却都深深地扎根于 Java 虚拟机平台上。

1.1.2 Java 发展史上的里程碑

下面，将简要介绍一下 Java 发展史上的重大事项。

1990 年，在 Sun 计算机公司中，由 Patrick Naughton、Mike Sheridan 及 James Gosling 领导的小组 Green Team，开始研发一种可控制家用电子产品的新型计算机软件技术，并希望能够研究出一种可以跨平台的系统。起先他们试着在 C++ 的功能基础上做修改，但一直无法克服编译器的问題，所以决定自行开发新的程序语言——Oak。这里的 Oak 已经具备安全性、网络通信、

面向对象、垃圾回收、多线程等特性。后来他们发现 Oak 已经被其他公司注册，于是改名为 Java。

1995 年，Sun 正式发布 Java 和 HotJava 产品，Java 首次公开亮相。

1996 年 1 月 23 日 Sun Microsystems 发布了 JDK 1.0。这个版本包括了两部分：运行环境（即 JRE）和开发环境（即 JDK）。在运行环境中包括了核心 API、用户界面 API、发布技术、Java 虚拟机（JVM）几个部分。开发环境包括了编译 Java 程序的编译器（即 javac）。在 JDK 1.0 时代，Java 使用一款叫作 Classic 的虚拟机解释执行 Java 字节码。

1997 年 2 月 18 日 Sun 发布了 JDK 1.1，在该版本中，已经支持 AWT、内部类、JDBC、RMI、反射等特性。同年，Sun 收购了一家叫作 Longview Technologies 的公司，从而获得了 Hotspot 虚拟机。

同在 1997 年，Jim Hugunin 创造了 Jython，但由于各种原因，Jython 的进展相当缓慢，但到现在为止，Jython 已经取得了长足的进步，甚至已经可以运行 Django 框架。

1998 年，JDK 1.2 版本发布（从这个版本开始的 Java 技术都称为 Java 2）。Java 2 不仅能兼容智能卡和小型消费类设备，还能兼容大型的服务器系统，它使软件开发商、服务提供商和设备提供商更加容易地抢占市场机遇。这一开发工具极大地简化了编程人员编制企业级 Web 应用的工作。同时 Sun 发布了 JSP/Servlet、EJB 规范，以及将 Java 分成了 J2EE、J2SE 和 J2ME。这表明了 Java 开始向企业、桌面应用和移动设备应用 3 大领域挺进。此时的 Java 已经做到解释执行和编译执行混合运行。

2000 年，JDK 1.3 发布，Hotspot 虚拟机成为 Java 的默认虚拟机。

2002 年，JDK 1.4 发布，古老的 Classic 虚拟机退出历史舞台。

2003 年年底，Java 平台的 Scala 正式发布，同年 Groovy 也加入了 Java 阵营。

2004 年，JDK 1.5 发布。同时 JDK 1.5 改名为 J2SE 5.0。在这个版本中，Java 语言做了大量的改进，比如支持了泛型、注解、自动装箱拆箱、枚举类型、可变长参数、增强的 foreach 循环等。语法上的简化和改进是这一版本的一大特色。

2006 年，JDK 1.6 发布。同年，Java 开源并建立了 OpenJDK。顺理成章，Hotspot 虚拟机也成为了 OpenJDK 中的默认虚拟机。

2007 年，Java 平台迎来了新伙伴 Clojure。

2008 年，Oracle 收购了 BEA，得到了 JRockit 虚拟机。

2009 年, Twitter 宣布把后台大部分程序从 Ruby 迁移到 Scala, 这是 Java 平台的又一次大规模应用。

2010 年, Oracle 收购了 Sun, 获得最具价值的 Hotspot 虚拟机。此时, Oracle 拥有市场占有率最高的两款虚拟机 Hotspot 和 JRockit, 并计划在未来对它们进行整合。

2011 年, JDK 1.7 发布。在 JDK 1.7 中, 正式启用了新的垃圾回收器 G1, 支持了 64 位系统的压缩指针, 以及 NIO 2.0, 同时新增的 invokedynamic 指令也是该版本的一大特色。

2014 年, JDK 1.8 发布。在 JDK 1.8 中, 全新的 Lambda 表达式是一大亮点, 它彻底改变了 Java 的编程风格和习惯。

Oracle 计划在 2016 年, 发布 JDK 1.9。届时, 最令人期待的功能应该是 Java 的模块化。

注意: 在本书中, JDK 1.6 等同于 JDK 6, JDK 1.7 等同于 JDK 7, JDK 1.8 等同于 JDK 8。

1.2 跨平台的真相: Java 虚拟机来做中介

在简单了解了 Java 的发展历程之后, 本节将着重介绍 Java 虚拟机的概念, 最后了解一下目前最流行的 Java 虚拟机。

1.2.1 理解 Java 虚拟机的原理

所谓虚拟机, 就是一台虚拟的计算机。它是一款软件, 用来执行一系列虚拟计算机指令。大体上, 虚拟机可以分为系统虚拟机和程序虚拟机。大名鼎鼎的 Visual Box、VMware 就属于系统虚拟机, 它们完全是对物理计算机的仿真, 提供了一个可运行完整操作系统的软件平台。程序虚拟机的典型代表就是 Java 虚拟机, 它专门为执行单个计算机程序而设计, 在 Java 虚拟机中执行的指令我们称为 Java 字节码指令。无论是系统虚拟机还是程序虚拟机, 在上面运行的软件都被限制于虚拟机提供的资源中。

图 1.1 显示了同一个 Java 程序 (Java 字节码的集合), 通过 Java 虚拟机运行于各大主流系统平台, 该程序以虚拟机为中介, 实现了跨平台的特性。

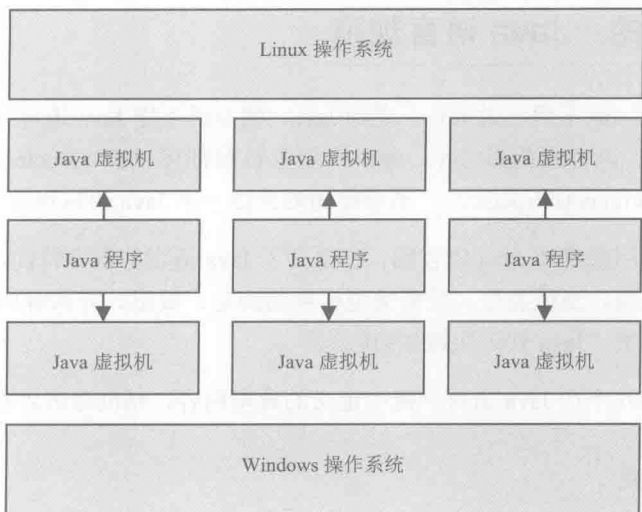


图 1.1 在操作系统之上执行的虚拟机程序

1.2.2 看清 Java 虚拟机的种类

Java 发展至今，先后出现了不少 Java 虚拟机。在 Java 发展最初，Sun 使用的是一款叫作 Classic 的 Java 虚拟机，之后，在 Solaris 平台上还曾短暂地使用过 Exact VM 虚拟机，到现在，最终被大规模部署和应用的是 Hotspot 虚拟机。

除了 Sun 公司以外，各大公司以及组织都曾积极研发过 Java 虚拟机，比如 BEA 的 JRockit，目前，JRockit 和 Hotspot 都被收入 Oracle 旗下，大有整合的趋势。在 IBM 内部，使用着一款名为 J9 的虚拟机，广泛用于 IBM 的各大产品（如果当年 IBM 成功收购了 Sun，那么很可能是 J9 和 Hotspot 进行整合了）。此外，Apache 也曾经推出过与 JDK 1.5 和 JDK 1.6 兼容的 Java 运行平台 Apache Harmony，它是开源软件，但受到同样开源的 OpenJDK 的压制，最终于 2011 年退役，虽然目前并没有 Apache Harmony 被大规模商用的案例，但是它的出现对 Android 的发展起到了极为重要的作用。在嵌入式领域，KVM 和 CDC/CLDC Hotspot 两款虚拟机也扮演着重要的角色，在 iOS 和 Android 盛行之前，这两款虚拟机也广泛运用于手机平台。

注意：由于目前 Hotspot 占有绝对的市场地位，若无特别说明，本书的示例以及参数都是针对 Hotspot 虚拟机的。

1.3 一切看我的：Java 语言规范

讲到 Java 虚拟机，就不得不说 Java，说到 Java，就不得不提 Java 语言规范（Java Language Specification）。Java 语言规范和 Java 虚拟机规范目前都可以在 Oracle 的官方网站上找到（<http://docs.oracle.com/javase/specs/>）。本节将简要介绍一下 Java 语言规范。

Java 语言规范是用来描述 Java 语言的，它定义了 Java 语言的语言特性，比如 Java 的语法、词法、支持的数据类型、变量类型、数据类型转换的约定、数组、异常等内容。Java 语言规范的目的是告诉开发人员“Java 代码应该如何编写”。

本节将简单介绍几个在 Java 语言规范中定义的典型内容，帮助读者理解这份规范的意图。

1.3.1 词法的定义

词法规定了 Java 语法中每一个单词如何书写才是合乎规范的。比如，词法定义中规定了 Java 的关键字，如果开发人员使用 Java 的关键字作为变量名，显然无法正常通过编译。

下面简单地看一个有关标示符的定义：

```
Identifier:
    IdentifierChars but not a Keyword or BooleanLiteral or NullLiteral

IdentifierChars:
    JavaLetter
    IdentifierChars JavaLetterOrDigit

JavaLetter:
    any Unicode character that is a Java letter

JavaLetterOrDigit:
    any Unicode character that is a Java letter-or-digit
```

这里定义了标示符由一个标示字符串组成，但是不能是关键字、布尔字面量或者是 Null 字面量。而标示字符串是一个由字母开头的、由字母或数字构成的串。这里的字母或数字并非简单的 ABC，而是指任意的 Unicode 字符。

根据这个定义便可以知道，下面的方法是不符合规范的：

```
public static void new () {
}
```


因为 new 为 Java 关键字, 不属于标示符, 因此不能用作方法名, 而下述代码则是符合规范的:

```
public static void 打印() {
    System.out.println("中文方法");
}
```

虽然上述代码使用了中文作为方法名, 但是根据定义, “打”、“印”属于 Unicode 字符, 因此是符合规范的名字, 也构成了符合规范的标示符。

有关词法另一个典型的案例就是数字的表示。在 JDK 1.7 中, 以下都属于符合规范的数字:

- 整数
 - 0、2、0372、0xDada_Cafe、1996、0x00_FF_00_FF
- 长整形
 - 0l、0777L、0x100000000L、2_147_483_648L、0xC0B0L
- 单精度浮点
 - 1e1f、2.f、.3f、0f、3.14f、6.022137e+23f
- 双精度浮点
 - 1e1、2.、.3、0.0、3.14、1e-9d、1e137

就直观感觉而言, 上述有些数字真是长得太奇怪了, 比如 0xDada_Café, 它不更像一个英语单词吗? 而事实上, 这是一个符合规范的 16 进制整数。Java 语言规范规定, 0x 字符开始的表示 16 进制, 同时, 为了可读性, 也允许在数字中间增加下画线进行分割 (这是 JDK 1.7 中引入的)。

得益于 Java 语言规范对于词法的定义, 现在 Java 程序可以使用更加丰富的方法来定义数字, 无论是在可读性或是在表达能力上, 都非常强劲。

1.3.2 语法的定义

词法定义规定了什么样的单词是合理的, 语法定义规定了什么样的语句是合乎规范的。以 if 语句为例, 在类似于 Basic 的语言中, 可能会用以下形式定义 if 语句:

```
if Expression then
Statement
end if
```

但是在 Java 中给出了这样的定义:

```
IfThenStatement:
    if ( Expression ) Statement
```

即在一个 if 语句中，表示条件的表达式必须用小括号标示，同时在右小括号后，书写语句块，表示执行内容。而对于 Expression 和 Statement 的具体定义，在语言规范中也有十分详细的描述，这里就不一一展开了，有兴趣的读者可以参考 Java 语言规范，JDK 1.7 版第 14 章的内容“Blocks and Statements”。

1.3.3 数据类型的定义

Java 语言规范中还定义了 Java 的数据类型。根据 Java 1.7 的规范，Java 的数据类型分为原始数据类型和引用数据类型。原始数据类型又分为数字型和布尔型。数字型又有 byte、short、int、long、char、float、double。注意，在这里 char 被定义为整数型，并且在规范中明确定义：byte、short、int、long 分别是 8 位、16 位、32 位、64 位有符号整数，而 char 为 16 位无符号整数，表示 UTF-16 的字符。布尔型只有两种取值：true 和 false。而对于 float 和 double，规范中规定，它们是满足 IEEE 754 的 32 位浮点数和 64 位浮点数。

注意：在 Java 语言中，char 占 2 字节，而不是 C 语言中的 1 字节。从这点上看，Java 的国际化是在语言底层就提供了强有力的支持。

此外，规范还定义了各类数字的取值范围、初始值，以及能够支持的各种操作。以整数为例，比较运算、数值运算、位运算、自增自减运算等都在规范中有描述。

除了基本数据类型外，引用数据类型也是 Java 重要的组成部分，引用数据类型分为 3 种：类或接口、泛型类型以及数组类型。

提醒：引用类型和原始类型在 Java 的处理中是截然不同的，尤其对于它们的“相等”操作。

【示例 1-1】在 Java 语言规范中，有一个简短的示例，说明了引用类型和原始类型的区别：

```
class Value { int val; }
class Test {
    public static void main(String[] args) {
        int i1 = 3;
        int i2 = i1;
        i2 = 4;
        System.out.print("i1==" + i1);
        System.out.println(" but i2==" + i2);
        Value v1 = new Value();
        v1.val = 5;
        Value v2 = v1;
```

```

        v2.val = 6;
        System.out.print("v1.val==" + v1.val);
        System.out.println(" and v2.val==" + v2.val);
    }
}

```

上述程序将输出：

```

i1==3 but i2==4
v1.val==6 and v2.val==6

```

从上述输出可以看出，对于原始数据类型 `int`，`i1` 和 `i2` 表示不同的变量，两者毫无关系，但是对于 `v1` 和 `v2`，它们都指向唯一一个由 `new` 关键字创建的 `Value` 对象。

由于本书并非讲解 Java 语言，因此对于这部分内容点到即止，有兴趣的读者可以参考 Java 语言规范的第 4 章 “Types, Values, and Variables”。

1.3.4 Java 语言规范总结

除上述基本内容外，Java 语言规范还定义了各种不同类型间的转换规则、方法的可见性定义、有关接口的使用、注释等。

总之，Java 语言规范完整定义和描述了 Java 语言的所有特性，因为 Java 语言本身不属于本书的讨论重点，故在此只做简要介绍。

1.4 一切听我的：Java 虚拟机规范

虽然 Java 语言和 Java 虚拟机有着密切的联系，但两者是完全不同的内容。Java 虚拟机是一台执行 Java 字节码的虚拟计算机，它拥有独立的运行机制，其运行的 Java 字节码也未必由 Java 语言编译而成，像 Groovy、Scala 等语言生成的 Java 字节码也可以由 Java 虚拟机执行。立足于 Java 虚拟机，可以产生各种各样的跨平台语言。除了语言特性各不相同外，它们可以共享 Java 虚拟机带来的跨平台性、优秀的垃圾回收器，以及可靠的即时编译器。

因此，与 Java 语言不同，Java 虚拟机是一个高效的、性能优异的、商用级别的软件运行和开发平台，而这也是本书讨论的重点。

Java 虚拟机规范的主要内容大概有以下几个部分：

- 定义了虚拟机的内部结构（将在第 2 章中详细介绍）。

- 定义了虚拟机执行的字节码类型和功能（将在第 11 章中详细介绍）。
- 定义了 Class 文件的结构（将在第 9 章中详细介绍）。
- 定义了类的装载、连接和初始化（将在第 10 章中详细介绍）。

以 Java 1.7 为例，读者可以在 <http://docs.oracle.com/javase/specs/jvms/se7/html/> 浏览虚拟机规范全文。这份规范可以说是开发 Java 虚拟机的指导性文件，如果要想实现自定义的 Java 虚拟机，则需要参考和熟悉这份规范，同时这份规范对于了解现存的流行 Java 虚拟机（如 Hotspot、IBM J9 等），也有十分重要的意义。

1.5 数字编码就是计算机世界的水和电

数字是计算机内最直接、最基础的表现类型。了解数字在计算机内的表示，对于了解整个计算机系统具有相当重要的作用，数字也是专业计算机从业人员的基本功之一。本节将主要介绍整数以及浮点数在 Java 虚拟机中的支持情况。

1.5.1 整数在 Java 虚拟机中的表示

在 Java 虚拟机中，整数有 byte、short、int、long 四种，分别表示 8 位、16 位、32 位、64 位有符号整数。整数在计算机中使用补码表示，在 Java 虚拟机中也不例外。在学习补码之前，必须先理解原码和反码。

所谓原码，就是符号位加上数字的二进制表示。以 int 为例，第 1 位表示符号位（正数或者负数），其余 31 位表示该数字的二进制值。

10 的原码为：00000000 00000000 00000000 00001010

-10 的原码为：10000000 00000000 00000000 00001010

对于原码来说，绝对值相同的正数和负数只有符号位不同。

反码就是在原码的基础上，符号位不变，其余位取反，以 -10 为例，其反码为：

11111111 11111111 11111111 11110101

负数的补码就是反码加 1，整数的补码就是原码本身。

因此，10 的补码为：

00000000 00000000 00000000 00001010

而-10的补码为:

```
11111111 11111111 11111111 11110110
```

在Java中,可以使用位运算查看整数中每一位的实际值,方法如下:

```
01 int a=-10;
02 for(int i=0;i<32;i++){
03     int t=(a & 0x80000000>>>i)>>>(31-i);
04     System.out.print(t);
05 }
```

以上代码将打印-10在虚拟机内的实际表示,程序的执行结果如下:

```
111111111111111111111111111111110110
```

可以看到,这个结果和之前补码的计算结果是完全匹配的。

这段程序的基本思想是:进行32次循环(因为int有32位),每次循环取出int值中的一位,第3行的0x80000000是一个首位为1、其余位为0的整数,通过右移i位,定位到要获取的第i位,并将除该位外的其他位统一设置为0,而该位不变,最后将该位移至最右,并进行输出。

相对于原码,使用补码作为计算机内的实际存储方式至少有以下两个好处。

(1)可以统一数字0的表示。由于0既非正数,又非负数,使用原码表示时符号位难以确定,把0归入正数或者负数得到的原码编码是不同的。但是使用补码表示时,无论把0归入正数或者负数都会得到相同的结果。计算过程如下:

如果0为正数,则补码为原码本身为:00000000 00000000 00000000 00000000。

如果0为负数,则补码为反码加1,负数0的原码为:

```
10000000 00000000 00000000 00000000
```

反码为:

```
11111111 11111111 11111111 11111111
```

补码在反码的基础上加1,结果为:

```
00000000 00000000 00000000 00000000
```

可以看到,使用补码作为整数编码,可以解决数字0的存储问题。

(2) 使用补码可以简化整数的加减法计算，将减法计算视为加法计算，实现减法和加法的完全统一，实现正数和负数加法的统一。现使用 8 位 (byte) 整数说明这个问题。

计算 $-6+5$ 的过程如下。

-6 补码: 11111010

5 补码: 00000101

直接相加得: 11111111

通过计算可知, 11111111 表示 -1。

计算 $4+6$ 的过程如下。

4 的补码: 00000100

6 的补码: 00000110

直接相加得: 00001010

通过计算可知, 00001010 表示 10 (十进制)。

可以看到, 使用补码表示时, 只需要将补码简单地相加, 即可得到算术加法的正确结果, 而无须区别正数或者负数。

1.5.2 浮点数在 Java 虚拟机中的表示

在 Java 虚拟机中, 浮点数有 float 和 double 两种, 分别是 32 位和 64 位浮点数。浮点数在虚拟机中的表示比整数略显复杂。目前, 使用最为广泛的是由 IEEE 754 定义的浮点数格式。目前 Java 虚拟机中对于浮点数的处理参考 IEEE 754 的规范。本节将以 float 为例, 简要介绍浮点数的表示方法。

在 IEEE 754 的定义中, 一个浮点数由 3 部分组成, 分别是: 符号位、指数位和尾数位。以 32 位 float 为例, 符号位占 1 位, 表示正负数, 指数位占 8 位, 尾数位占剩余的 23 位, 如图 1.2 所示。

其中, sflag 表示符号, 当 s 为 0 时, sflag 为 1, 当 s 为 1 时, sflag 为 -1。m 为尾数值, 实际占用空间为 23 位, 但是根据 e 的取值, 有 24 位精度。当 e 全为 0 时, 尾数位附加为 0, 否则, 尾数位附加为 1。e 为指数位, 用 8 位表示。

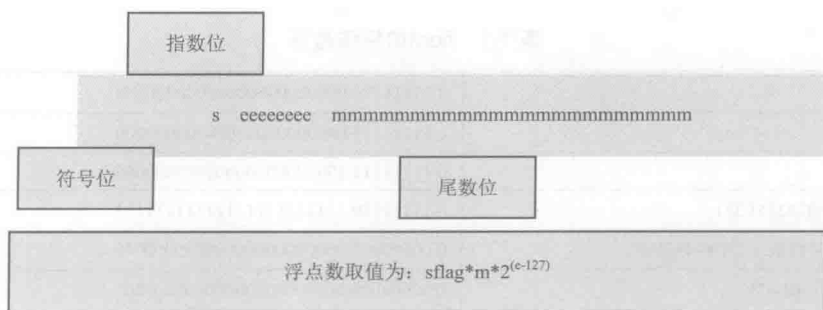


图 1.2 浮点数格式

以浮点数-5 为例，其内部表示为：

1 10000001 010000000000000000000000

符号位为 1 表示负数，指数位为 10000001，表示 129。

尾数位为: 010000000000000000000000。因为 e 不全为 0, 故实际的尾数位为:

10100000000000000000000000000000

尾数位表示 2 的指数次幂的和，每一位表示求和数列中的对应项是否为 0，这里表示：

$1*2^0+0*2^{-1}+1*2^{-2}+0*2^{-3}+0*2^{-4}+0*2^{-5}...$, 对应关系如图 1.3 所示。

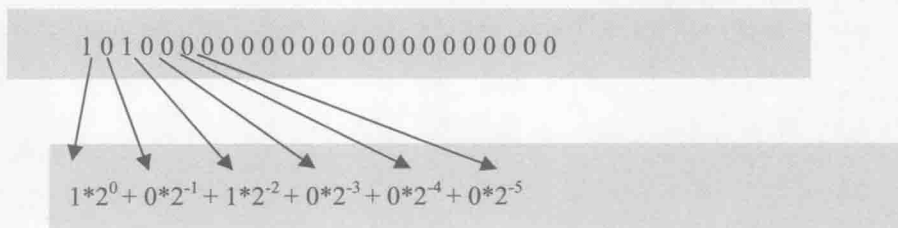


图 1.3 尾数的计算

故 1 10000001 010000000000000000000000 的值为:

$$-1*2^{(129-127)}*(1*2^0+0*2^{-1}+1*2^{-2}+0*2^{-3}+0*2^{-4}+0*2^{-5}) = -1*4*1.25 = -5$$

浮点数 float 还可以表示一些特殊的数字, 如表 1.1 所示。

表 1.1 float 的特殊数字

正无穷	0 11111111 000000000000000000000000
负无穷	1 11111111 000000000000000000000000
NaN	0 11111111 100000000000000000000000
最大浮点数(3.4028235E38)	0 11111110 111111111111111111111111
最小规范化正浮点数(1.17549435E-38)	0 00000001 000000000000000000000000
最小正浮点数(1.4E-45)	0 00000000 000000000000000000000001
0	0 00000000 000000000000000000000000

其中，指数位全为 1 的表示无穷大和 NaN 等特殊数字。指数位全为 0 的为非规范化的浮点数。

【示例 1-2】在 Java 中，使用 Float.floatToIntBits() 函数可以获得一个单精度浮点数的 IEEE 754 表示。以下代码打印了 -5 的内部表示：

```
float a=-5;
System.out.println(Integer.toBinaryString(Float.floatToIntBits(a)));

程序运行结果为：

11000000101000000000000000000000
```

其中，Float.floatToIntBits() 函数最终由 native 方法实现，具体实现如下：

```
JNIEXPORT jint JNICALL
Java_java_lang_Float_floatToIntBits(JNIEnv *env, jclass unused, jfloat v)
{
    union {
        int i;
        float f;
    } u;
    u.f = (float)v;
    return (jint)u.i;
}
```

从上述代码可以看出，为了获取 float 的内部表示，使用了 C 语言中的 union 自然实现这个转换。

1.6 抛砖引玉：编译和调试虚拟机

如果要对虚拟机进行深入的研究，那么编译和调试 Java 虚拟机是必不可少的。为何要编译

自己的虚拟机呢？主要原因有两点。

第一，通过自己编译可以得到一个 debug 或者 fastdebug 版本的调试用虚拟机，调试用虚拟机可以支持更多的虚拟机参数，这些开发专用的虚拟机参数可以帮助开发人员获得更多的虚拟机内部信息，而这些参数在正常发行版本中是无法使用的。因此考虑到本书的实用性，本书并不会过多介绍那些只在调试版本中使用的参数，但如果读者有兴趣，可以编译自己的虚拟机，进行尝试。

第二，使用自己编译的调试版的虚拟机可以用于虚拟机代码的单步调试，有利于实现对虚拟机代码的理解。由于虚拟机代码比较复杂，仅通过代码阅读很难深入理解其实现机制，有时不得不依靠单步调试进行辅助，而编译后调试版本可以帮助实现这一功能。

为编译虚拟机，首先必须获得虚拟机源码，读者可以在 OpenJDK 的官方网站上下载最新源码，下载地址为 <https://jdk7.java.net/source.html>。笔者目前使用的是 openjdk-7u40-fcs-src-b43-26_aug_2013.zip。推荐读者使用较新的版本，因为老版本的编译脚本可能在某些平台上存在问题。

笔者的编译环境为 Ubuntu 系统，读者可以选择自己喜欢的 Linux 发行版进行编译。编译虚拟机之前，还必须做一些准备工作。

1. 编译前的准备工作

(1) 安装好依赖库。在 Ubuntu 平台上可以通过 apt-get 命令安装，在 CentOS 平台上可以通过 yum 命令安装。比如，笔者在编译前至少安装了以下库：

```
apt-get install build-essential
apt-get install libxrender-devsudo
apt-get install xorg-devsudo
apt-get install libasound2-devsudo
apt-get install libcups2-dev
apt-get install libasound2-dev
apt-get install libfreetype6-dev
apt-get install libcups2-dev
apt-get install zip
apt-get install libX11-dev
apt-get install libxext-dev
apt-get install libxrender-dev
apt-get install libxtst-dev
apt-get install libxt-dev
```

如果依赖库安装不全，在编译过程中就会提示错误，从错误提示中，读者应该可以得知缺少了哪些库或者头文件，相应地安装即可。当然了，gcc 和 g++ 作为基本的编译工具也是必须要

安装的。

(2) 准备一个 Boot JDK 和 ANT。Boot JDK 用于 OpenJDK 编译的执行。笔者在这里使用的是 jdk1.6.0u45，也推荐读者使用 JDK 1.6 的版本作为 JDK 1.7 的 Boot JDK。此外，还需要准备 ANT 1.7.1 以上版本，作为编译的基础执行工具。

2. 准备编译

准备就绪之后，就可以开始准备编译了。

(1) 进入解压后的 openjdk 目录：

```
geym@hzlab001:~/openjdk/openjdk$ ls
ASSEMBLY_EXCEPTION  build.sh  get_source.sh  hotspot.log  jaxws  langtools
make README test
build corba hotspot jaxp jdk LICENSE Makefile README-builds.html THIRD_
PARTY_README
```

(2) 新建文件 build.sh，作为编译启动执行脚本。建立此脚本的目的是方便编译执行，因为编译前可能会预设一些环境变量，写成脚本后更加容易维护和执行。读者可以根据自己的执行环境，自行修改，笔者的脚本如下：

```
#!/bin/bash
export ANT_HOME=/home/geym/tools/apache-ant-1.9.4
export PATH=$ANT_HOME/bin:$PATH
export ALT_BOOTDIR=/home/geym/tools/jdk1.6.0_45
unset JAVA_HOME
export ALT_JDK_IMPORT_PATH=/home/geym/tools/jdk1.6.0_45
unset CLASSPATH

export SKIP_DEBUG_BUILD=false
export SKIP_FASTDEBUG_BUILD=false
export DEBUG_NAME=debug
export LANG=C

cd /geym/home/openjdk/openjdk
make sanity && make BUILD_JAXWS=false BUILD_JAXP=false WARNINGS_ARE_ERRORS=
```

该脚本中，首先设置了 ANT_HOME，并将 ANT 执行目录加入 PATH，接着设置了 Boot JDK 的路径。这里注意，需要 unset JAVA_HOME 以及 CLASSPATH 变量。

(3) 通过 make 命令执行整个编译，根据这个 build 脚本，将会同时生成 debug 版本、

fastdebug 和常规发行版本三个虚拟机的编译结果。编译的过程可能会花费比较长的时间，一般来说，编译一个版本可能需要 15 到 45 分钟，视计算机性能而定。当编译成功后，会有以下输出，显示了编译耗时。

```
#-- Build times -----
Target debug_build
Start 2014-11-10 18:04:19
End   2014-11-10 18:21:14
00:01:27 corba
00:07:18 hotspot
00:07:43 jdk
00:00:26 langtools
00:16:55 TOTAL
-----
```

进入 build 目录，可以看到编译的结果，下面显示了 3 个版本的编译结果：

```
geym@:~/openjdk/openjdk/build$ du -h --max-depth=1
2.5G    ./linux-amd64-fastdebug
2.4G    ./linux-amd64-debug
2.4G    ./linux-amd64
```

有了 debug 版本的虚拟机，就可以使用 gdb 对虚拟机进行调试了。接下来将简单地演示 Java 虚拟机的调试方法。

3. Java 虚拟机的调试

笔者选用 linux-amd64-debug 下的虚拟机进行调试。

(1) 首先进入 linux-amd64-debug 目录，查找名为 gamma 的文件。

```
geym@:~/openjdk/openjdk/build/linux-amd64-debug$ find -name gamma
./hotspot/outputdir/linux_amd64_compiler2/jvmg/gamma
```

该文件就是目标文件，用于调试，实际上，它就是 java 程序的 debug 版本。这里记下文件的路径。

(2) 接着，进入含有 Java Class 执行文件的目录下并启动 gdb。启动 gdb 后，先为 gamma 程序的运行设定环境变量：

```
(gdb) set environment JAVA_HOME=/home/geym/tools/jdk1.6.0_45
(gdb) set environment CLASSPATH=.:${JAVA_HOME}/jre/lib/rt.jar:${JAVA_HOME}/jre/lib/i18n.jar
(gdb) set environment HOTSPOT_BUILD_USER="geym in hotspot"
```

```
(gdb) set environment LD_LIBRARY_PATH=/home/geym/openjdk/openjdk/build/linux-
amd64-debug/hotspot/outputdir/linux_amd64_compiler2/jvms:$JAVA_HOME/jre/lib/
amd64:$JAVA_HOME/jre/lib/amd64/native_threads
```

(3) 使用 file 指令将 gamma 程序设为目标文件，此时应该会显示如下信息：

```
(gdb) file /home/geym/openjdk/openjdk/build/linux-amd64-debug/hotspot/outputdir/
linux_amd64_compiler2/jvms/gamma
Reading symbols from /home/geym/openjdk/openjdk/build/linux-amd64-debug/
hotspot/outputdir/linux_amd64_compiler2/jvms/gamma...done.
```

这表示，读取调试符号信息成功。

(4) 设置传递给 gamma 程序的参数，也就是传递给 Java 程序的参数，一般为需要执行的 Main Class 的名称，这里运行 SimpleGc 类，并为 SimpleGc 程序传入参数 ggg，这相当于执行 java SimpleGc ggg。

```
(gdb) set args SimpleGc ggg
```

(5) 设置断点，这里在 Java 进程的 main() 函数中设置断点。

```
(gdb) break main
Breakpoint 1 at 0x403a02: file /home/geym/openjdk/openjdk/hotspot/src/share/
tools/launcher/java.c, line 228.
```

(6) 执行 run 命令启动 Java 程序，开始调试。

```
(gdb) run
Starting program: /home/geym/openjdk/openjdk/build/linux-amd64-debug/hotspot/
outputdir/linux_amd64_compiler2/jvms/gamma SimpleGc ggg
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main (argc=3, argv=0x7fffffff3d8) at /home/geym/openjdk/
openjdk/hotspot/src/share/tools/launcher/java.c:228
228     {
(gdb) n
省略部分输出
(gdb)
329     if (!ParseArguments(&argc, &argv, &jarfile, &classname, &ret, jvmspath)) {
(gdb) p argv
$4 = (char **) 0x7fffffff3e0
(gdb) p *argv
$7 = 0x7fffffff695 "SimpleGc"
```

```
(gdb) p *(argv+1)
$8 = 0x7fffffff69e "ggg"
```

可以看到，当前 Java 程序从 java.c 文件的第 228 行开始执行。使用命令 `next`（缩写 `n`），进行单步调试，这里省略了中间部分输出，在运行到 `ParseArguments()` 函数时，通过 `print`（缩写 `p`）命令显示了传递给 Java 进程的参数，这里可以看到作为 Main Class 的 `SimpleGc` 及其参数“`ggg`”。

至此，读者就可以使用这套环境作为辅助，更好地深入 Java 虚拟机内部了。

1.7 小结

本章主要介绍了 Java 语言和 Java 虚拟机的发展历史，并介绍了 Java 生态环境中两份非常重要的规范——Java 语言规范和 Java 虚拟机规范。其中 Java 虚拟机规范将成为本书后续讨论的重点内容。同时，作为了解 Java 虚拟机的第一步，简要介绍了整数和浮点数在 Java 虚拟机中的表示方式。最后，为了方便读者更加深入地了解虚拟机，还给出了单步调试 Java 虚拟机的方法。

2

第 2 章

认识 Java 虚拟机的基本结构

Java 虚拟机那么复杂，它的基本结构是什么？各个组成部分有何作用？又是如何相互协调工作的呢？本章将试图解答这些问题，而要解答这些问题就必须先了解 Java 堆、Java 栈、永久区和元数据区的基本概念。

本章涉及的主要知识点有：

- 认识 Java 虚拟机中的堆。
- 了解有关栈的概念和使用。
- 了解存放类型描述的永久区和元数据区。

2.1 谋全局者才能成大器：看穿 Java 虚拟机的架构

Java 虚拟机的基本结构如图 2.1 所示。

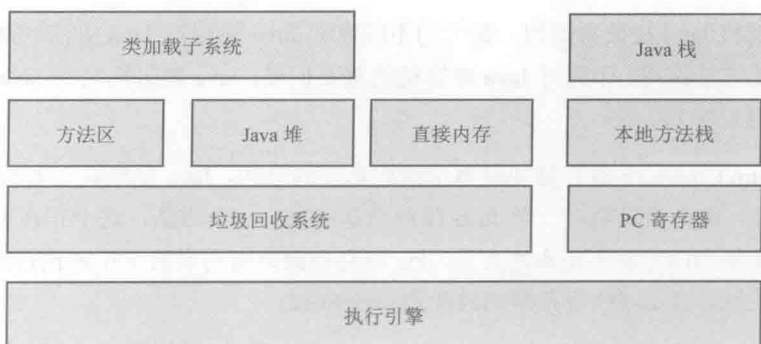


图 2.1 Java 虚拟机的基本结构

类加载子系统负责从文件系统或者网络中加载 Class 信息，加载的类信息存放于一块称为方法区的内存空间。除了类的信息外，方法区中可能还会存放运行时常量池信息，包括字符串字面量和数字常量（这部分常量信息是 Class 文件中常量池部分的内存映射）。

Java 堆在虚拟机启动的时候建立，它是 Java 程序最主要的内存工作区域。几乎所有的 Java 对象实例都存放于 Java 堆中。堆空间是所有线程共享的，这是一块与 Java 应用密切相关的内存区间。

Java 的 NIO 库允许 Java 程序使用直接内存。直接内存是在 Java 堆外的、直接向系统申请的内存区间。通常，访问直接内存的速度会优于 Java 堆。因此出于性能考虑，读写频繁的场合可能会考虑使用直接内存。由于直接内存存在 Java 堆外，因此它的大小不会直接受限于 Xmx 指定的最大堆大小，但是系统内存是有限的，Java 堆和直接内存的总和依然受限于操作系统能给出的最大内存。

垃圾回收系统是 Java 虚拟机的重要组成部分，垃圾回收器可以对方法区、Java 堆和直接内存进行回收。其中，Java 堆是垃圾收集器的工作重点。和 C/C++ 不同，Java 中所有的对象空间释放都是隐式的。也就是说，Java 中没有类似 free() 或者 delete() 这样的函数释放指定的内存区域。对于不再使用的垃圾对象，垃圾回收系统会在后台默默工作，默默查找、标识并释放垃圾对象，完成包括 Java 堆、方法区和直接内存中的全自动化管理。有关垃圾回收系统的更多信息，可以参阅第 4 章和第 5 章。

每一个 Java 虚拟机线程都有一个私有的 Java 栈。一个线程的 Java 栈在线程创建的时候被创建。Java 栈中保存着帧信息（参阅本章 2.4 节），Java 栈中保存着局部变量、方法参数，同时和 Java 方法的调用、返回密切相关。

本地方法栈和 Java 栈非常类似，最大的不同在于 Java 栈用于 Java 方法的调用，而本地方法栈则用于本地方法调用。作为对 Java 虚拟机的重要扩展，Java 虚拟机允许 Java 直接调用本地方法（通常使用 C 编写）。

PC (Program Counter) 寄存器也是每个线程私有的空间，Java 虚拟机会为每一个 Java 线程创建 PC 寄存器。在任意时刻，一个 Java 线程总是在执行一个方法，这个正在被执行的方法称为当前方法。如果当前方法不是本地方法，PC 寄存器就会指向当前正在被执行的指令。如果当前方法是本地方法，那么 PC 寄存器的值就是 `undefined`。

执行引擎是 Java 虚拟机的最核心组件之一，它负责执行虚拟机的字节码。现代虚拟机为了提高执行效率，会使用即时编译技术将方法编译成机器码后再执行。执行引擎的进一步细节描述可以参阅第 11 章。

2.2 小参数能解决大问题：学会设置 Java 虚拟机的参数

Java 虚拟机可以使用 `JAVA_HOME/bin/java` 程序启动(`JAVA_HOME` 为 JDK 的安装目录)，一般来说，Java 进程的命令行使用方法如下：

```
java [-options] class [args...]
```

其中，`-options` 表示 Java 虚拟机的启动参数，`class` 为带有 `main()` 函数的 Java 类，`args` 表示传递给主函数 `main()` 的参数。

如果需要设定特定的 Java 虚拟机参数，在 `options` 处指定即可。目前，Hotspot 虚拟机支持大量的虚拟机参数，可以帮助开发人员进行系统调优和故障排查。相关的一些参数将在本书的后续章节中逐步展开介绍，本节则主要介绍参数的设置方法。

【示例 2-1】以如下代码为例，读者先来了解一下如何设置参数。

```
package geym.zbase.ch2;

public class SimpleArgs {
    public static void main(String[] args) {
        for(int i=0;i<args.length;i++){
            System.out.println("参数" + (i+1) + ":" + args[i]);
        }
        System.out.println("-Xmx" + Runtime.getRuntime().maxMemory() / 1000 / 1000 + "M");
    }
}
```


上述代码打印了传递给 `main()` 函数的参数，同时打印了系统的最大可用堆内存。使用如下命令行运行这段代码：

```
java -Xmx32m geym.zbase.ch2.SimpleArgs a
参数 1:a
-Xmx32M
```

从结果可以看到，第一个参数 `-Xmx32m` 传递给 Java 虚拟机，生效后，使得系统最大可用堆空间为 32MB，参数 `a` 则传递给主函数 `main()`，作为应用程序的参数。

有关 `-Xmx` 会在本书后续章节中展开讨论，除了 `-Xmx` 外，虚拟机还支持大量的调优诊断参数，其设置方式都是类似的，在本书后续章节中会逐步介绍这些参数。

如果读者使用 Eclipse 等开发工具运行程序，在运行对话框的参数选项卡上，也可以设置这两个参数，如图 2.2 所示，显示了“程序参数”和“虚拟机参数”两个文本框，将所需的参数填入即可。



图 2.2 通过 Eclipse 为虚拟机设置启动参数

2.3 对象去哪儿：分清 Java 堆

Java 堆是和 Java 应用程序关系最为密切的内存空间，几乎所有的对象都存放在堆中。并且 Java 堆是完全自动化管理的，通过垃圾回收机制，垃圾对象会被自动清理，而不需要显式地释放。

根据垃圾回收机制的不同，Java 堆有可能拥有不同的结构。最为常见的一种构成是将整个 Java 堆分为新生代和老年代。其中，新生代存放新生对象或者年龄不大的对象，老年代则存放老年对象。新生代有可能分为 eden 区、s0 区、s1 区，s0 和 s1 也被称为 from 和 to 区域，它们是一块大小相等、可以互换角色的内存空间。详细信息可以参阅第 4 章。

图 2.3 显示了一个堆空间的一般结构。

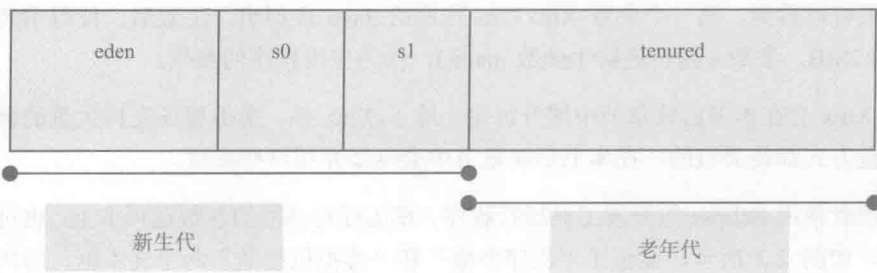


图 2.3 堆空间的一般结构

在绝大多数情况下，对象首先分配在 eden 区，在一次新生代回收后，如果对象还存活，则会进入 s0 或者 s1，之后，每经过一次新生代回收，对象如果存活，它的年龄就会加 1。当对象的年龄达到一定条件后，就会被认为是老年对象，从而进入老年代。

【示例 2-2】下面通过一个简单的示例，来展示 Java 堆、方法区和 Java 栈之间的关系。

```
public class SimpleHeap {
    private int id;
    public SimpleHeap(int id){
        this.id=id;
    }
    public void show(){
        System.out.println("My ID is "+id);
    }
    public static void main(String[] args) {
        SimpleHeap s1=new SimpleHeap(1);
        SimpleHeap s2=new SimpleHeap(2);
        s1.show();
        s2.show();
    }
}
```

上述代码声明了一个 SimpleHeap 类，并在 main() 函数中创建了两个 SimpleHeap 实例。此时，各对象和局部变量的存放如图 2.4 所示。SimpleHeap 实例本身分配在堆中，描述 SimpleHeap

类的信息存放在方法区，main()函数中 s1 和 s2 局部变量存放在 Java 栈中，并指向堆中的两个实例。

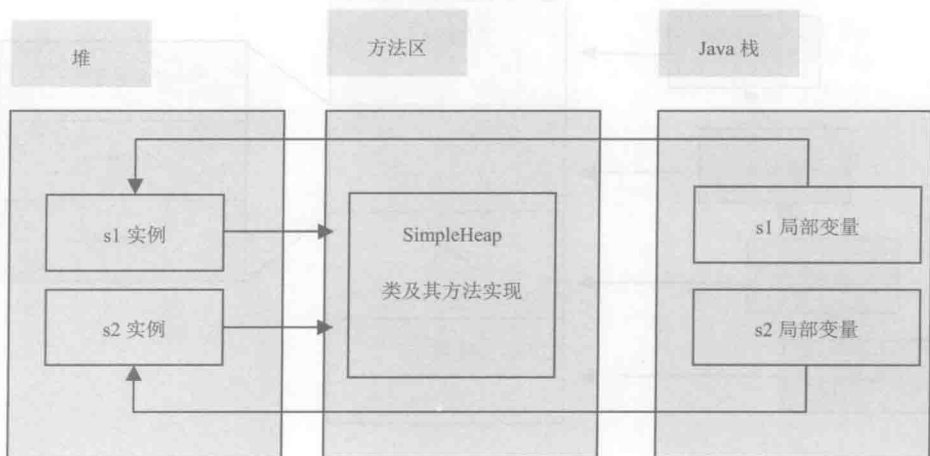


图 2.4 堆、方法区、栈的关系

2.4 函数如何调用：出入 Java 栈

Java 栈是一块线程私有的内存空间。如果说，Java 堆和程序数据密切相关，那么 Java 栈就是和线程执行密切相关的。线程执行的基本行为是函数调用，每次函数调用的数据都是通过 Java 栈传递的。

Java 栈与数据结构上的栈有着类似的含义，它是一块先进后出的数据结构，只支持出栈和入栈两种操作。在 Java 栈中保存的主要内容为栈帧。每一次函数调用，都会有一个对应的栈帧被压入 Java 栈，每一个函数调用结束，都会有一个栈帧被弹出 Java 栈。如图 2.5 所示，函数 1 对应栈帧 1，函数 2 对应栈帧 2，依此类推。函数 1 中调用函数 2，函数 2 中调用函数 3，函数 3 中调用函数 4。当函数 1 被调用时，栈帧 1 入栈；当函数 2 被调用时，栈帧 2 入栈；当函数 3 被调用时，栈帧 3 入栈；当函数 4 被调用时，栈帧 4 入栈。当前正在执行的函数所对应的帧就是当前的帧（位于栈顶），它保存着当前函数的局部变量、中间运算结果等数据。

当函数返回时，栈帧从 Java 栈中被弹出。Java 方法有两种返回函数的方式，一种是正常的函数返回，使用 `return` 指令；另外一种抛出异常。不管使用哪种方式，都会导致栈帧被弹出。

在一个栈帧中，至少要包含局部变量表、操作数栈和帧数据区几个部分。

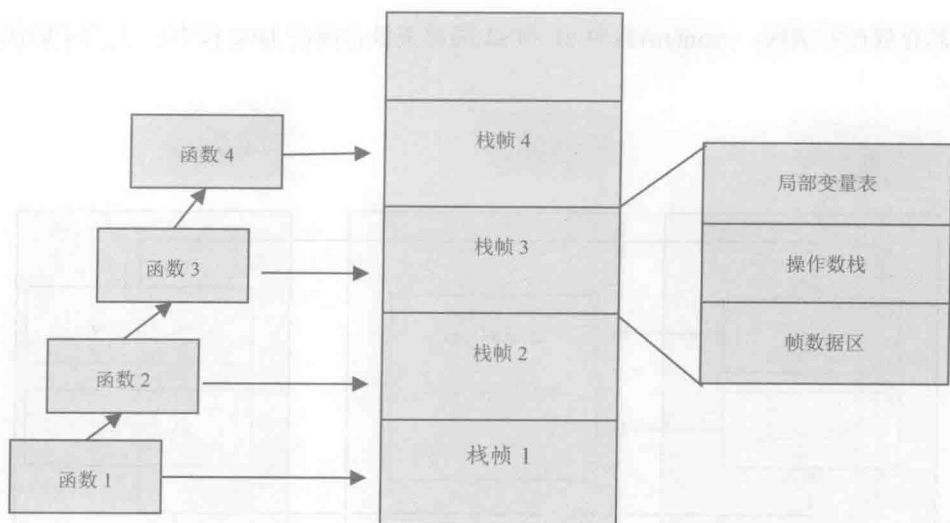


图 2.5 栈帧和函数调用

提示：由于每次函数调用都会生成对应的栈帧，从而占用一定的栈空间，因此，如果栈空间不足，那么函数调用自然无法继续进行下去。当请求的栈深度大于最大可用栈深度时，系统就会抛出 `StackOverflowError` 栈溢出错误。

Java 虚拟机提供了参数 `-Xss` 来指定线程的最大栈空间，这个参数也直接决定了函数调用的最大深度。

【示例 2-3】下面的代码是一个递归调用，由于递归没有出口，这段代码可能会出现栈溢出错误，在抛出错误后，程序打印了最大的调用深度。

```
public class TestStackDeep {
    private static int count=0;
    public static void recursion(){
        count++;
        recursion();
    }
    public static void main(String args[]){
        try{
            recursion();
        }catch(Throwable e){
            System.out.println("deep of calling = "+count);
            e.printStackTrace();
        }
    }
}
```

```

    }
}

```

使用参数-Xss128K 执行以上代码，部分结果如下所示：

```

deep of calling = 2505
java.lang.StackOverflowError
    at geym.zbase.ch2.xss.TestStackDeep.recursion(TestStackDeep.java:16)
    at geym.zbase.ch2.xss.TestStackDeep.recursion(TestStackDeep.java:17)
    ...

```

可以看到，在进行大约 2500 次调用后，发生了栈溢出错误，通过增大-Xss 的值，可以获得更深的调用层次，尝试使用参数-Xss256K 执行上述代码，可能产生如下输出，很明显，调用层次有明显的增加。

```

deep of calling = 5809
java.lang.StackOverflowError
    at geym.zbase.ch2.xss.TestStackDeep.recursion(TestStackDeep.java:16)
    at geym.zbase.ch2.xss.TestStackDeep.recursion(TestStackDeep.java:17)

```

注意：函数嵌套调用的层次在很大程度上由栈的大小决定，栈越大，函数可以支持的嵌套调用次数就越多。

2.4.1 局部变量表

局部变量表是栈帧的重要组成部分之一。它用于保存函数的参数以及局部变量。局部变量表中的变量只在当前函数调用中有效，当函数调用结束后，随着函数栈帧的销毁，局部变量表也会随之销毁。

由于局部变量表在栈帧之中，因此，如果函数的参数和局部变量较多，会使得局部变量表膨胀，从而每一次函数调用就会占用更多的栈空间，最终导致函数的嵌套调用次数减少。

【示例 2-4】下面的代码演示了这种情况，第 1 个 recursion() 函数含有 3 个参数和 10 个局部变量，因此，其局部变量表含有 13 个变量。而第 2 个 recursion() 函数不含有任何参数和局部变量。当这两个函数被嵌套调用时，第 2 个 recursion() 函数可以拥有更深的调用层次。

```

public class TestStackDeep {
    private static int count=0;
    public static void recursion(long a,long b,long c){
        long e=1,f=2,g=3,h=4,i=5,k=6,q=7,x=8,y=9,z=10;
        count++;
        recursion(a,b,c);
    }
}

```

```

    }
    public static void recursion(){
        count++;
        recursion();
    }
    public static void main(String args[]){
        try{
            // recursion(0L,0L,0L);
            recursion();
        }catch(Throwable e){
            System.out.println("deep of calling = "+count);
            e.printStackTrace();
        }
    }
}

```

使用参数-Xss128K 执行上述代码中的第 1 个 recursion()函数，输出结果如下：

```

deep of calling = 692
java.lang.StackOverflowError
    at geym.zbase.ch2.xss.TestStackDeep.recursion(TestStackDeep.java:11)

```

使用参数-Xss128K 执行上述代码中的第 2 个 recursion()函数，输出结果如下：

```

deep of calling = 2496
java.lang.StackOverflowError
    at geym.zbase.ch2.xss.TestStackDeep.recursion(TestStackDeep.java:16)

```

可以看到，在相同的栈容量下，局部变量少的函数可以支持更深的函数调用。

使用 jclasslib 工具可以更进一步查看函数的局部变量信息。图 2.6 显示了第一个 recursion()函数的最大局部变量表的大小为 26 个字。因为该函数包含总共 13 个参数和局部变量，且都为 long 型，long 和 double 在局部变量表中需要占用 2 个字，其他如 int、short、byte、对象引用等占用 1 个字。

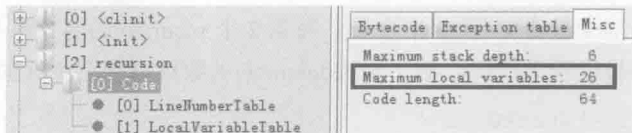
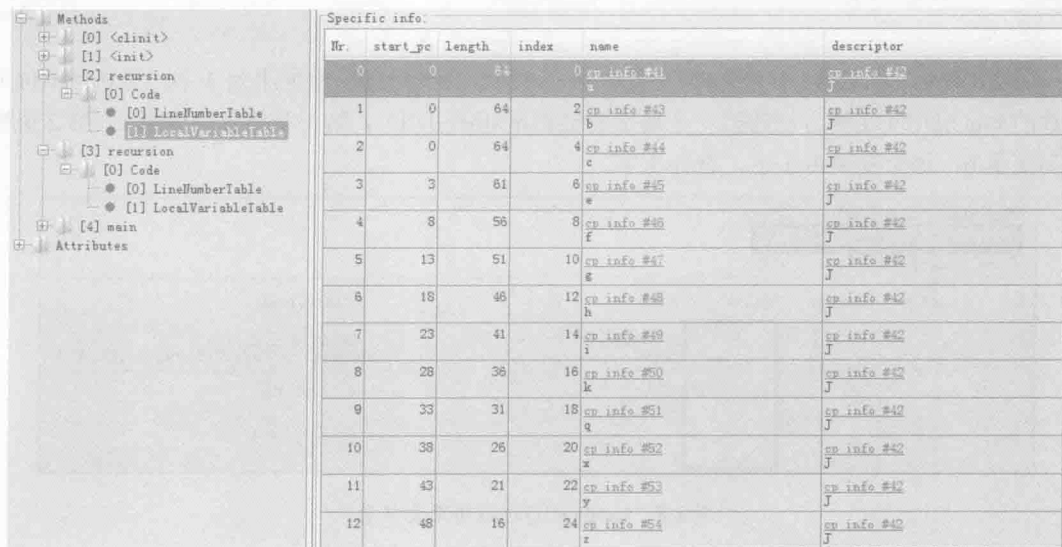


图 2.6 最大局部变量表大小

说明：字（Word）指的是计算机内存中占据一个单独的内存单元编号的一组二进制串。一般 32 位计算机上一个字为 4 个字节长度。

图 2.7 显示了在 Class 文件中的局部变量表的内容（这里说的局部变量表和上述的局部变量表不同，这里指 Class 文件的一个属性，而上述的局部变量表指 Java 栈空间的一部分）。



Mr.	start_pc	length	index	name	descriptor
0	0	64	0	cp info #41 a	cp info #42 J
1	0	64	2	cp info #43 b	cp info #42 J
2	0	64	4	cp info #44 c	cp info #42 J
3	3	61	6	cp info #45 e	cp info #42 J
4	8	56	8	cp info #46 f	cp info #42 J
5	13	51	10	cp info #47 g	cp info #42 J
6	18	46	12	cp info #48 h	cp info #42 J
7	23	41	14	cp info #49 i	cp info #42 J
8	28	36	16	cp info #50 k	cp info #42 J
9	33	31	18	cp info #51 q	cp info #42 J
10	38	26	20	cp info #52 x	cp info #42 J
11	43	21	22	cp info #53 y	cp info #42 J
12	48	16	24	cp info #54 z	cp info #42 J

图 2.7 Class 文件中的局部变量表

可以看到，在 Class 文件的局部变量表中，显示了每个局部变量的作用域范围、所在槽位的索引（index 列）、变量名（name 列）和数据类型（J 表示 long 型）。

栈帧中的局部变量表中的槽位是可以重用的，如果一个局部变量过了其作用域，那么在其作用域之后声明的新的局部变量就很有可能会复用过期局部变量的槽位，从而达到节省资源的目的。

【示例 2-5】下面的代码显示了局部变量表槽位的复用。在 `localvar1()` 函数中，局部变量 `a` 和 `b` 都作用到了函数末尾，故 `b` 无法复用 `a` 所在的位置。而在 `localvar2()` 函数中，局部变量 `a` 在第 10 行时不再有效，故局部变量 `b` 可以复用 `a` 的槽位（1 个字）。

```

01 public void localvar1(){
02     int a=0;
03     System.out.println(a);
04     int b=0;
05 }
06 public void localvar2(){
07     {
08     int a=0;

```

```

09      System.out.println(a);
10      }
11      int b=0;
12  }
```

图 2.8 显示了 `localvar1()` 的局部变量信息，该函数最大局部变量大小为 3 字，第 0 个槽位为函数的 `this` 引用（实例方法的第一个局部变量都是 `this` 引用），第 1 个槽位为变量 `a`，第 2 个槽位为变量 `b`，每个变量占 1 字，合计 3 字。

Maximum stack depth:	1
Maximum local variables:	3
Code length:	5

Mr.	start_pc	length	index	name	descriptor
0	0	5	0	cp info #12 this	cp info #13 Lgeym/zbase/ch2/localvar/Loca...
1	2	3	1	cp info #15 a	cp info #16 I
2	4	1	2	cp info #17 b	cp info #16 I

图 2.8 `localvar1()` 的局部变量信息

图 2.9 显示了 `localvar2()` 的局部变量信息，该函数的最大局部变量表为 2 字，虽然和 `localvar1()` 一样，拥有 `this`、`a`、`b` 等 3 个局部变量，但 `b` 复用了 `a` 的槽位（从它们都占用了第 1 个槽位可以知道这点），因此在整个函数执行中，同时存在的最大局部变量为 2 字。

Maximum stack depth:		2
Maximum local variables:		2
Code length:		12

Hr.	start_pc	length	index	name	descriptor
0	0	12	0	cp info #12 this	cp info #13 lgeym/zbase/ch2/localvar/Loca...
1	2	7	1	cp info #15 a	cp info #16 I
2	11	1	1	cp info #17 b	cp info #16 I

图 2.9 `localvar2()` 的局部变量信息

局部变量表中的变量也是重要的垃圾回收根节点，只要被局部变量表中直接或间接引用的对象都是不会被回收的。因此，理解局部变量表对理解垃圾回收也有一定帮助。

【示例 2-6】下面通过一个简单的示例，展示局部变量对垃圾回收的影响。

```

public void localvarGc1(){
    byte[] a=new byte[6*1024*1024];
```



```

        System.gc();
    }
    public void localVarGc2(){
        byte[] a=new byte[6*1024*1024];
        a=null;
        System.gc();
    }
    public void localVarGc3(){
        {
            byte[] a=new byte[6*1024*1024];
        }
        System.gc();
    }
    public void localVarGc4(){
        {
            byte[] a=new byte[6*1024*1024];
        }
        int c=10;
        System.gc();
    }
    public void localVarGc5(){
        localVarGc1();
        System.gc();
    }
    public static void main(String[] args) {
        LocalVarGC ins=new LocalVarGC();
        ins.localVarGc1();
    }
}

```

上述代码中，每一个 localVarGcN() 函数都分配了一块 6MB 的堆空间，并使用局部变量引用这块空间。

在 localVarGc1() 中，在申请空间后，立即进行垃圾回收，很明显，由于 byte 数组被变量 a 引用，因此无法回收这块空间。

在 localVarGc2() 中，在垃圾回收前，先将变量 a 置为 null，使 byte 数组失去强引用，故垃圾回收可以顺利回收 byte 数组。

对于 localVarGc3()，在进行垃圾回收前，先使局部变量 a 失效，虽然变量 a 已经离开了作用域，但是变量 a 依然存在于局部变量表中，并且也指向这块 byte 数组，故 byte 数组依然无法被回收。

对于 `localvarGc4()`，在垃圾回收之前，不仅使变量 `a` 失效，更是申明了变量 `c`，使变量 `c` 复用了变量 `a` 的字，由于变量 `a` 此时被销毁，故垃圾回收器可以顺利回收 `byte` 数组。

对于 `localvarGc5()`，它首先调用了 `localvarGc1()`，很明显，在 `localvarGc1()` 中并没有释放 `byte` 数组，但在 `localvarGc1()` 返回后，它的栈帧被销毁，自然也包含了栈帧中的所有局部变量，故 `byte` 数组失去引用，在 `localvarGc5()` 的垃圾回收中被回收。

读者可以使用参数 `-XX:+PrintGC` 执行上述几个函数，在输出的日志中，可以看到垃圾回收前后堆的大小，进而推断 `byte` 数组是否被回收。下面的输出是函数 `localvarGc4()` 的运行结果：

```
[Full GC 6746K->376K(15872K), 0.0034589 secs]
```

从日志中可以看到，堆空间从回收前的 6746KB 变为回收后的 376KB，释放了约 6MB 空间。进而可以推断，`byte` 数组已被回收释放。

2.4.2 操作数栈

操作数栈也是栈帧中的重要内容之一，它主要用于保存计算过程的中间结果，同时作为计算过程中变量临时的存储空间。

操作数栈也是一个先进后出的数据结构，只支持入栈和出栈两种操作。许多 Java 字节码指令都需要通过操作数栈进行参数传递。比如 `iadd` 指令，它就会在操作数栈中弹出两个整数并进行加法计算，计算结果会被入栈，如图 2.10 所示，显示了 `iadd` 前后操作数栈的变化。



图 2.10 `iadd` 指令与操作数栈的变化

有关操作数栈和局部变量表使用的详细案例，读者可以参考本书第 11.1 节。

2.4.3 帧数据区

除了局部变量表和操作数栈外，Java 栈帧还需要一些数据来支持常量池解析、正常方法返回和异常处理等。大部分 Java 字节码指令需要进行常量池访问，在帧数据区中保存着访问常量池的指针，方便程序访问常量池。

此外，当函数返回或者出现异常时，虚拟机必须恢复调用者函数的栈帧，并让调用者函数继续执行下去。对于异常处理，虚拟机必须有一个异常处理表，方便在发生异常的时候找到处理异常的代码，因此异常处理表也是帧数据区中重要的一部分。一个典型的异常处理表如下所示：

Exception table:

from	to	target type
4	16	19 any
19	21	19 any

它表示在字节码偏移量 4~16 字节可能抛出任意异常，如果遇到异常，则跳转到字节码偏移 19 处执行。当方法抛出异常时，虚拟机就会查找类似的异常表来进行处理，如果无法在异常表中找到合适的处理方法，则会结束当前函数调用，返回调用函数，并在调用函数中抛出相同的异常，并查找调用函数的异常表进行处理。

2.4.4 栈上分配

栈上分配是 Java 虚拟机提供的一项优化技术，它的基本思想是，对于那些线程私有的对象（这里指不可能被其他线程访问的对象），可以将它们打散分配在栈上，而不是分配在堆上。分配在栈上的好处是可以在函数调用结束后自行销毁，而不需要垃圾回收器的介入，从而提高系统的性能。

栈上分配的一个技术基础是进行逃逸分析。逃逸分析的目的是判断对象的作用域是否有可能逃逸出函数体。如下代码显示了一个逃逸的对象：

```
private static User u;
public static void alloc(){
    u=new User();
    u.id=5;
    u.name="geym";
}
```

对象 User u 是类的成员变量，该字段有可能被任何线程访问，因此属于逃逸对象。而以下代码片段显示了一个非逃逸的对象：

```
public static void alloc(){
    User u=new User();
    u.id=5;
    u.name="geym";
}
```

在上述代码中，对象 `User` 以局部变量的形式存在，并且该对象并没有被 `alloc()` 函数返回，或者出现了任何形式的公开，因此，它并未发生逃逸，所以对于这种情况，虚拟机就有可能将 `User` 分配在栈上，而不在堆上。

【示例 2-7】下面这个简单的示例显示了对非逃逸对象的栈上分配。

```

01 public class OnStackTest {
02     public static class User{
03         public int id=0;
04         public String name="";
05     }
06
07     public static void alloc(){
08         User u=new User();
09         u.id=5;
10         u.name="geym";
11     }
12     public static void main(String[] args) throws InterruptedException {
13         long b=System.currentTimeMillis();
14         for(int i=0;i<100000000;i++){
15             alloc();
16         }
17         long e=System.currentTimeMillis();
18         System.out.println(e-b);
19     }
20 }

```

上述代码在主函数中进行了 1 亿次 `alloc()` 调用进行对象创建，由于 `User` 对象实例需要占据约 16 字节的空间，因此累计分配空间达到将近 1.5GB。如果堆空间小于这个值，就必然会发生 GC。使用如下参数运行上述代码：

```

-server -Xmx10m -Xms10m -XX:+DoEscapeAnalysis -XX:+PrintGC -XX:-UseTLAB
-XX:+EliminateAllocations

```

这里使用参数 `-server` 执行程序，因为在 `Server` 模式下，才可以启用逃逸分析。参数 `-XX:+DoEscapeAnalysis` 启用逃逸分析，`-Xmx10m` 指定了堆空间最大为 10MB，显然，如果对象在堆上分配，必然会引起大量的 GC。如果 GC 真的发生了，参数 `-XX:+PrintGC` 将打印 GC 日志。参数 `-XX:+EliminateAllocations` 开启了标量替换（默认打开），允许将对象打散分配在栈上，比如对象拥有 `id` 和 `name` 两个字段，那么这两个字段将会被视为两个独立的局部变量进行分配。参数 `-XX:-UseTLAB` 关闭了 TLAB。

程序执行后，完整的输出打印如下：

6

可以看到，没有任何形式的 GC 输出，程序就执行完毕了。说明在执行过程中，User 对象的分配过程被优化。

如果关闭逃逸分析或者标量替换中任何一个，再次执行程序，就会看到大量的 GC 日志，说明栈上分配依赖逃逸分析和标量替换的实现。

对于大量的零散小对象，栈上分配提供了一种很好的对象分配优化策略，栈上分配速度快，并且可以有效避免垃圾回收带来的负面影响，但由于和堆空间相比，栈空间较小，因此对于大对象无法也不适合在栈上分配。

2.5 类去哪儿了：识别方法区

和 Java 堆一样，方法区是一块所有线程共享的内存区域。它用于保存系统的类信息，比如类的字段、方法、常量池等。方法区的大小决定了系统可以保存多少个类，如果系统定义了太多的类，导致方法区溢出，虚拟机同样会抛出内存溢出错误。

在 JDK 1.6、JDK 1.7 中，方法区可以理解为永久区（Perm）。永久区可以使用参数 `-XX:PermSize` 和 `-XX:MaxPermSize` 指定，默认情况下，`-XX:MaxPermSize` 为 64MB。一个大的永久区可以保存更多的类信息。如果系统使用了一些动态代理，那么有可能会在运行时生成大量的类，如果这样，就需要设置一个合理的永久区大小，确保不发生永久区内存溢出。

【示例 2-8】下面这段代码使用 CGLIB 库生成大量的动态类。

```
public class PermTest {
    public static void main(String[] args) {
        int i = 0;
        try {
            for (i = 0; i < 100000; i++) {
                CglibBean bean = new CglibBean("geym.zbase.ch2.perm" + i, new
HashMap());
            }
        } catch (Exception e) {
            System.out.println("total create count:" + i);
        }
    }
}
```

PDF电子书说明：

本人可以提供各种PDF电子书资料，计算机类，文学，艺术，设计，医学，理学，经济，金融，等等。质量都很清晰，而且每本100%都带书签和目录，方便读者阅读观看，只要您提供给我书的相关信息，一般我都能找到，如果您有需求，请联系我 **QQ: 461573687**, 或者 **QQ: 2404062482**。

本人已经帮助了上万人找到了他们需要的PDF，其实网上有很多PDF,大家如果在网上不到的话，可以联系我QQ。因PDF电子书都有版权，请不要随意传播，最近pdf也越来越难做了，希望大家尊重下个人劳动，谢谢！