Broadview
www.broadview.com.cn


# 揭秘Java虚拟机

## JVM设计原理与实现

封亚飞 著

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn



# 揭秘Java虚拟机

## JVM设计原理与实现

封亚飞 著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

## 内 容 简 介

本书从源码角度解读HotSpot的内部实现机制，本版本主要包含三大部分——JVM数据结构设计与实现、执行引擎机制及内存分配模型。

数据结构部分包括Java字节码文件格式、常量池解析、字段解析、方法解析。每一部分都给出详细的源码实现分析，例如字段解析一章，从源码层面详细分析了Java字段重排、字段继承等关键机制。再如方法解析一章，给出了Java多态特性在源码层面的实现方式。本书通过直接对源代码的分析，从根本上梳理和澄清Java领域中的关键概念和机制。

执行引擎部分包括Java方法调用机制、栈帧创建机制、指令集架构与解释器实现机制。这一话题是全书技术含量最高的部分，需要读者具备一定的汇编基础。不过千万不要被“汇编”这个词给吓着，其实在作者看来，汇编相比于高级语言而言，语法非常简单，语义也十分清晰。执行引擎部分重点描述Java源代码如何转换为字节码，又如何从字节码转换为机器指令从而能够被物理CPU所执行的技术实现。同时详细分析了Java函数堆栈的创建全过程，在源码分析的过程中，带领读者从本质上理解到底什么是Java函数堆栈和栈帧，以及栈帧内部的详细结构。

内存分配部分主要包括类型创建与加载、对象实例创建与内存分配，例如new关键字的工作机制，import关键字的作用，再如java.lang.ClassLoader.loadClass()接口的本地实现机制。

本书并不是简单地分析源码实现，而是在描述HotSpot内部实现机制的同时，分析了HotSpot如此这般实现的技术必然性。读者在阅读本书的过程中，将会在很多地方看到作者本人的这种思考。



**图书在版编目（CIP）数据**

揭秘 Java 虚拟机：JVM 设计原理与实现 / 封亚飞著. —北京：电子工业出版社，2017.7
ISBN 978-7-121-31541-1

Ⅰ. ①揭… Ⅱ. ①封… Ⅲ. ①JAVA 语言－程序设计Ⅳ. ①TP312.8

中国版本图书馆 CIP 数据核字(2017)第 101824 号

策划编辑：刘　皎
责任编辑：郑柳洁
特约编辑：梁卫红
印　　刷：北京京科印刷有限公司
装　　订：三河市良远印务有限公司
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：42.25　　字数：942 千字
版　　次：2017 年 7 月第 1 版
印　　次：2017 年 7 月第 1 次印刷
定　　价：129.00 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）88254888，88258888。

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

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

# 推荐序

从 Java 诞生至今已有二十余年，基于虚拟机的技术屏蔽了底层环境的差异，“一次编译，随处运行”的思想促进了整个 IT 上层技术应用产生了翻天覆地的变化。Java 作为服务端应用语言的首选，确实大大降低了学习和应用的门槛。现实生活中，绝大多数 Java 程序员对于虚拟机的原理和实现了解并不深入，也似乎并不那么关心。而随着互联网的极速发展，现在的 Java 服务端应用需要应对极高的并发访问和大量的数据交互，从机制和设计原理上了解虚拟机的核心原理和实现细节显然能够帮助 Java 程序员编写出更高效优质的代码。

虽然市面上从 Java 使用者角度介绍虚拟机的书也有不少佳作，但一般较为宽泛，尤其在谈及虚拟机如何运行、处理的细节时总有些浅尝辄止的遗憾。而作者凭借深厚的 C 与 Java 技术功底以及多年对于 JVM 的深入研究编写的这本书，真正从虚拟机指令执行处理层面，结合 JVM 规范的设计原理，完整和详尽地阐述了 Java 虚拟机在处理类、方法和代码时的设计和实现细节。书中大量的代码和指令细节能够让程序员更加直接地理解相关原理。

这是一本优秀的技术工具书，可以让阅读者更加深刻地理解虚拟机的原理和处理细节，值得每一位具有极客精神、追求细节的优秀程序员反复阅读和收藏。

菜鸟平台技术部　陌铭

# 前言

文明需要创造，也需要传承。JVM 作为一款虚拟机，本身便是技术之集大成者，里面包含方方面面的底层技术知识。抛开如今 Java 如日中天之态势不说，纯粹从技术层面看，JVM 也值得广大技术爱好者深入研究。可以说，从最新的硬件特性，到最新的软件技术，只要技术被证明是成熟的，都会在 JVM 里面见到其踪影。JDK 的每一次更新，从内部到核心类库，JVM 都会及时引入这些最新的技术或者算法，这便是技术传承意义之所在。随着云计算、大数据、人工智能等最新技术的发展，Java 技术生态圈也日益庞大，JVM 与底层平台以及与其他编程语言和技术的交互、交织日益深入，这些都离不开对 JVM 内部机制的深入理解。如果说以前在中间件与框架领域的大展身手，依靠的是 Java 语言层面的特性和技术，那么以后越来越多的技术红利将会因 JVM 层面之创新而得以显现。

被真相所蒙蔽，是一件痛苦的事。我们在一个被层层封装的世界里进行开发和设计，操作系统、各种中间件与框架，将底层世界隐藏得结结实实。我们一方面享受着高级编程语言所带来的高效、稳定、快速的开发体验，然而另一方面，却又如同行走于黑暗之中。我们不知道路的下面是否有坑，即使有坑，可能也不知道如何排除。Java 的很多概念和技术，很多时候由于我们对底层机制的不了解，而让我们感到十分高深莫测，无法知其全貌。这种感觉非常痛苦，尤其是技术修炼到一定阶段的时候。

纸上得来终觉浅，绝知此事要躬行。即使从 Java 语言层面下探到 JVM 层面，但是若只囿于对 JVM 机制理论和概念上的理解，很多时候仍然觉得缺乏那种大彻大悟之感。计算机作为一门科学，与其他的科学领域一样，不仅需要对其理论的理解，也需要能够去实证。例如爱因斯坦的相对论十分高深，但是通过对引力波和红移的观测，其变得形象和生动起来。Java 的部分概念经过“口口相传”，似有过于夸大其技术神秘性之嫌，让人望而生畏。例如，与 volatile 关键字相关的内存可见性、指令乱序等概念，给人无比博大深奥的印象，但是如能抛开概念，直接看底层实现机制，并辅以具体的实验论证，则会形成深刻而彻底的认知。其实，这世界本来就很简单。在可观测的实验结果与可理解的底层机制面前，一切浮夸的概念都自然会现出原形。

因此，采用自底而上的技术研究之道，相比自顶而下的办法，便多了更多窥透本质的自信和平实。同一个底层概念，在不同的高级编程语言里，在概念、叫法上很少能够保持一致。采用自底而上的探索方法，能够揭开各种深奥概念的神秘面纱，还原一个清明简洁的世界。自然理解曲线也不会有大起大落。

研究 JVM 的过程，就是与大师们进行精神沟通和心灵交流的过程，虽然过程会比较痛苦。研究诸如 Linux、JVM 这样的底层程序，你能学习到大师级的理念，更能够见识到经无数牛人反复锤炼后的技术。天长日久的耳濡目染，终有一天你也会成为大师，你也会拥有大师级的眼光，你也会拥有开阔的胸怀。如同音乐家李健，人们如此喜欢他，并不仅仅是因为他歌唱得好，更多的是因为气质。而这种气质来自于博览群书，来自于对艺术的长久修炼。计算机从某种程度上而言，也是一门艺术，工程师和程序员们要想进化，对计算机艺术的修炼必不可少。与大师进行精神沟通，不仅能够修炼到计算机的艺术，更能直接感受并养成大师身上所具备的气质。

我不知道 Java 还能走多远，未来是否会被淘汰，但你不能因此就否定研究 JVM 的意义。JVM 作为一款虚拟机，各种底层技术和理论都有涉及，若你能研究透彻，则能一通百通。例如，本人在研究过程中，也翻阅了诸如 Python、JavaScript 等高级面向对象语言虚拟机的机制，发现它们内部的整体思路都相差不大。同时，JVM 本身在运行期干了一部分 C 或 C++语言编译器所干的事，例如符号解析、链接、面向对象机制的实现等，通过对这些机制的分析，从来没有研究过 C/C++编译器原理的我，基本也能够猜出 C/C++编译器可能的实现方式，后来翻阅了相关资料，果不其然。理解编译与虚拟机的实现机制是一方面，另一方面，通过深挖 JDK 核心类库的内部实现，则能够深刻理解线程、并发、I/O 等比较高深的技术内幕。例如 Java NIO，何谓 VMA？何谓内核映射？若想真正彻底理解这些概念，不从底层入手，恐怕很难有一个具象化的认知。总之，研究 JVM，是一件非常能够提升开发者内功的事情，未来无论出现什么样的新语言、新技术、新概念，你总是能够不被表面的东西所迷惑，而是能够透过层层封装，看清事物的本质，你总是能够以极低的学习成本，迅速理解新的东西。从一个更为广阔的视角，使用发散的思维去看，不一定非要研究 JVM 才能有很大收获，研究其他技术的底层，会有异曲同工之妙。而我只不过恰好生在了这个年代，这个 Java 语言大行其道的年代，所以就恰好对其做了一个比较深入的研究而已。工具有时空疆界，而技术思想则没有，其总能穿越千万年的时空，无限延伸。

JVM 涉及的知识面十分广阔，因此限于篇幅，本书并未覆盖 JVM 的全部内容。总体而言，本书重点描述了 JVM 从启动开始到完成函数执行的详细机制，读完本书，相信你一定能够明白 JVM 执行 Java 程序的底层机制，能够明白 JVM 将 Java 语言一步步转换为 CPU 可执行的机器码的内部机制，以及为此而制定的各种规范的实现之道，例如 oop-klass 模型、堆栈分配模型、类加载模型等。

本书作为笔者本人的处女作，前后写了有两年之久。之所以写这么久，一方面是因为 JVM

本身涉及大量的知识，另一方面则是笔者本人在写作过程中，力求对每一个知识点都做实验进行验证，避免因为笔者的错误理解而误导了别人。同时，在这个过程中，笔者不仅仅满足于读懂 JVM 的源代码，也不仅仅满足于通过实验去验证各个技术点，笔者花了更多的时间在思考 JVM 各种技术选择的必然性。换言之，在具体的硬件和操作系统的约束之下，在 JVM“write once, run anywhere”这一思路设定下，JVM 内部的技术实现机制是确定的，别无他法。例如，JVM 的 GC 机制便是一种技术必然性选择。每一次对这种技术必然性之思考，就好像与 JVM 的作者大牛们进行了一次心灵交流。

我问大牛：JVM 的堆栈结构为什么要有操作数栈和局部变量表？

大牛回答：因为……。

大牛其实并没有回答我，所有的一切都需要笔者自己去想明白。等想明白了之后，才有种真正看透事物本质的快感。

希望本书的读者也能够跟随本书，一起去进行这样的思考。

正是因为对“知其所以然”之追求，所以本书的写作过程是漫长的。在此，要特别感谢我的老婆金艳和我的儿子佑佑，很多个周末我都没能抽出时间陪伴他们。当我开始打算写作本书时，我的孩子还在襁褓之中。而等我写完本书，孩子已经开始上小班了。欠缺的太多！

JVM 所涉及的知识面既广且深，而个人所知毕竟有限，书中定有错误之处，若有发现，请发送邮件至：chaomengyuexiang@126.com。

在写作本书的过程中，遇到了很多技术障碍，有很多技术障碍都是在查阅了 R 大以及阿里技术专家三红、寒泉子等人的文章后才得以攻克，在此表示感谢！向这些大牛致敬！

同时，也要感谢我的领导和同事所给予的大力支持，尤其要感谢菲青、陌铭、祝幽、兰博等人的鼓励，同时你们也是我学习的榜样！

# 目录

# 第 1 章
# Java 虚拟机概述

本章摘要

- ◎ Java 语言产生的历史背景
- ◎ 编程语言跨平台的实现
- ◎ 中间语言的实现

## 1.1 从机器语言到 Java——詹爷，你好

当年——在 60 多年前，程序员是这样干活的：

写一段程序，将其打在纸带或卡片上，1 打孔，0 不打孔，然后将纸带或卡片输入计算机。那时候的程序都是只用 0 和 1 写成，注意，是只用 0 和 1 哟！

道理大家都懂，计算机嘛，只识别 0 和 1。

当年，程序员用于编程的 IDE 就是剪刀+胶水，只这两板斧，就能闯天下。

图 1.1 所示就是传说中的穿孔卡带，这就是当年程序员们开发出来的程序。

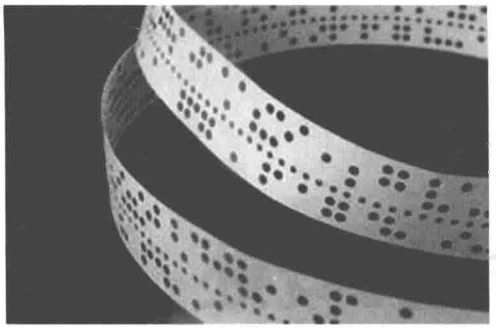

图 1.1　保存程序的卡带

请聚精会神盯着这个纸带上面的孔看 3 秒钟，计时开始：1，2，3。OK，时间到，怎么样？晕了吧！这哪里是程序，分明是个千疮百孔的纸带嘛！

什么，没晕!!! 好，那我们一起来玩个游戏吧，这个游戏就是：大家一起来找茬（相信很多经历并见证诺基亚这个手机巨人倒下以及安卓兴起这段历史的朋友们，对这款游戏一定有很深的印象）。好的，游戏开始，先看下面第一幅图（没错，下面就是使用数字 0 和 1 所绘制而成的特殊的“图画”）：

10100001000000100000000

00000011000010100000010

再看第二幅图：

10100001000000100000000

00000011000010010000010

现在，请比较这两幅图，找出其中的差异。相信眼尖的你一定很快就发现了其中的不同。其实，这段机器代码是在执行一个小学一年级的简单数学题目：1+2。

什么，还没晕！

你这分明是逼我放大招的节奏啊！那好，我就满足你一下，现在，我们稍微提升一下游戏的难度，还是两幅图，第一幅图如下：

10100001000000100000000

01001011000111010000000

01001011000011010000100

00000001110101000

00000011000111010000000l

1110001011110110

第二幅图如下：

10100001000000010000000O

01001011000111010000000O

01001011000011010000010O

0000000111010100

00000011000011010000000l

1110001011110110

现在请找出这两幅图中的差异点。

现在真的晕了吧！还是让我悄悄地告诉你差异点在哪里吧，倒数第 2 行左边开始第 12 个位置的 1 变成了 0 哦，别问我是怎么知道的，我是不会告诉你的。

其实，这段程序是在计算一个小学 2 年级的数学题目：求 1~8 的和。这段机器码中使用了循环，如果写成 Java 代码，类似这样：

```
int sum=0;
for(int i=0; i < 8; i++){
    sum += i;
}
```

游戏玩完了，怎么样？一定很无聊吧。当年的程序员们也是这么想的。设想一下，如果你恰好不幸出生在那个年代，并且恰好不幸当了一名程序员，某一天你编写了一个程序，这个程序不大，只有 2000 行，结果你恰好一不小心把其中某一行的某个 0 写成了 1，或者把某个 1 写成了 0，或者少写了一个 0，那是一件多么不幸的事！代码写错了。可这对于现代程序员而言，那都不是事，大不了断点调试一把嘛！但在那个 IDE 简陋到只有剪刀+胶水的年代，那可真是抱黄连敲门——苦到家了。你只能把眼睛瞪得跟铜锣一样大，对着这 2000 行代码，仔仔细细、一行一行地排查问题究竟出在哪里。等你“望眼欲穿”，终于发现问题后，你得重新操起家伙（剪刀和胶水），重新制作纸带，然后重新运行程序。可是，怀着万分期待的你，绝望地发现程序还是运行出错了！恭喜你，还得麻烦你老再次逐行排查这 2000 行机器码，保证不把你的眼睛看瞎。保重，不谢！

于是，大家纷纷要求改变这种反人类的工作，而改变的思路就是：既然机器码这么难以阅读、理解和排错，那就用助记符吧。于是人们开始用助记符来编写程序，编写完了，先用人脑来执行一遍（厉害吧），确保没有问题后，再将助记符手工转换成机器码，制作成孔带。