



# 揭秘Java虚拟机

## JVM设计原理与实现

封亚飞 著

电子工业出版社  
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。

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

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](mailto:zlts@phei.com.cn)，盗版侵权举报请发邮件至 [dbqq@phei.com.cn](mailto:dbqq@phei.com.cn)。

本书咨询联系方式：010-51260888-819, [faq@phei.com.cn](mailto: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](mailto: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 .....	1
1.1 从机器语言到 Java——詹爷，你好 .....	1
1.2 兼容的选择：一场生产力的革命 .....	6
1.3 中间语言翻译 .....	10
1.3.1 从中间语言翻译到机器码 .....	11
1.3.2 通过 C 程序翻译 .....	11
1.3.3 直接翻译为机器码 .....	13
1.3.4 本地编译 .....	16
1.4 神奇的指令 .....	18
1.4.1 常见汇编指令 .....	20
1.4.2 JVM 指令 .....	21
1.5 本章总结 .....	24
第 2 章 Java 执行引擎工作原理：方法调用 .....	25
2.1 方法调用 .....	26
2.1.1 真实的机器调用 .....	26
2.1.2 C 语言函数调用 .....	41
2.2 JVM 的函数调用机制 .....	47
2.3 函数指针 .....	53
2.4 CallStub 函数指针定义 .....	60
2.5 _call_stub_entry 例程 .....	72

2.6 本章总结 .....	115
<b>第 3 章 Java 数据结构与面向对象 .....</b>	<b>117</b>
3.1 从 Java 算法到数据结构 .....	118
3.2 数据类型简史 .....	122
3.3 Java 数据结构之偶然性 .....	129
3.4 Java 类型识别 .....	132
3.4.1 class 字节码概述 .....	133
3.4.2 魔数与 JVM 内部的 int 类型 .....	136
3.4.3 常量池与 JVM 内部对象模型 .....	137
3.5 大端与小端 .....	143
3.5.1 大端和小端的概念 .....	146
3.5.2 大小端产生的本质原因 .....	148
3.5.3 大小端验证 .....	149
3.5.4 大端和小端产生的场景 .....	151
3.5.5 如何解决字节序反转 .....	154
3.5.6 大小端问题的避免 .....	156
3.5.7 JVM 对字节码文件的大小端处理 .....	156
3.6 本章总结 .....	159
<b>第 4 章 Java 字节码实战 .....</b>	<b>161</b>
4.1 字节码格式初探 .....	161
4.1.1 准备测试用例 .....	162
4.1.2 使用 javap 命令分析字节码文件 .....	162
4.1.3 查看字节码二进制 .....	165
4.2 魔数与版本 .....	166
4.2.1 魔数 .....	168
4.2.2 版本号 .....	168
4.3 常量池 .....	169
4.3.1 常量池的基本结构 .....	169
4.3.2 JVM 所定义的 11 种常量 .....	170
4.3.3 常量池元素的复合结构 .....	170
4.3.4 常量池的结束位置 .....	172

4.3.5	常量池元素总数量	172
4.3.6	第一个常量池元素	173
4.3.7	第二个常量池元素	174
4.3.8	父类常量	174
4.3.9	变量型常量池元素	175
4.4	访问标识与继承信息	177
4.4.1	access_flags	177
4.4.2	this_class	178
4.4.3	super_class	179
4.4.4	interface	179
4.5	字段信息	180
4.5.1	fields_count	180
4.5.2	field_info fields[fields_count]	181
4.6	方法信息	185
4.6.1	methods_count	185
4.6.2	method_info methods[methods_count]	185
4.7	本章回顾	205
第 5 章	常量池解析	206
5.1	常量池内存分配	208
5.1.1	常量池内存分配总体链路	209
5.1.2	内存分配	215
5.1.3	初始化内存	223
5.2	oop-klass 模型	224
5.2.1	两模型三维度	225
5.2.2	体系总览	227
5.2.3	oop 体系	229
5.2.4	klass 体系	231
5.2.5	handle 体系	234
5.2.6	oop、klass、handle 的相互转换	239
5.3	常量池 klass 模型（1）	244
5.3.1	klassKlass 实例构建总链路	246

5.3.2 为 klassOop 申请内存 .....	249
5.3.3 klassOop 内存清零 .....	253
5.3.4 初始化 mark .....	253
5.3.5 初始化 klassOop._metadata .....	258
5.3.6 初始化 klass .....	259
5.3.7 自指 .....	260
5.4 常量池 klass 模型（2） .....	261
5.4.1 constantPoolKlass 模型构建 .....	261
5.4.2 constantPoolOop 与 klass .....	264
5.4.3 klassKlass 终结符 .....	267
5.5 常量池解析 .....	267
5.5.1 constantPoolOop 域初始化 .....	268
5.5.2 初始化 tag .....	269
5.5.3 解析常量池元素 .....	271
5.6 本章总结 .....	279
<b>第 6 章 类变量解析 .....</b>	<b>280</b>
6.1 类变量解析 .....	281
6.2 偏移量 .....	285
6.2.1 静态变量偏移量 .....	285
6.2.2 非静态变量偏移量 .....	287
6.2.3 Java 字段内存分配总结 .....	312
6.3 从源码看字段继承 .....	319
6.3.1 字段重排与补白 .....	319
6.3.2 private 字段可被继承吗 .....	325
6.3.3 使用 HSDB 验证字段分配与继承 .....	329
6.3.4 引用类型变量内存分配 .....	338
6.4 本章总结 .....	342
<b>第 7 章 Java 栈帧 .....</b>	<b>344</b>
7.1 entry_point 例程生成 .....	345
7.2 局部变量表创建 .....	352
7.2.1 constMethod 的内存布局 .....	352

7.2.2 局部变量表空间计算 .....	356
7.2.3 初始化局部变量区 .....	359
7.3 堆栈与栈帧 .....	368
7.3.1 栈帧是什么 .....	368
7.3.2 硬件对堆栈的支持 .....	387
7.3.3 栈帧开辟与回收 .....	390
7.3.4 堆栈大小与多线程 .....	391
7.4 JVM 的栈帧 .....	396
7.4.1 JVM 栈帧与大小确定 .....	396
7.4.2 栈帧创建 .....	399
7.4.3 局部变量表 .....	421
7.5 栈帧深度与 slot 复用 .....	433
7.6 最大操作数栈与操作栈复用 .....	436
7.7 本章总结 .....	439
<b>第 8 章 类方法解析 .....</b>	<b>440</b>
8.1 方法签名解析与校验 .....	445
8.2 方法属性解析 .....	447
8.2.1 code 属性解析 .....	447
8.2.2 LVT&LVTT .....	449
8.3 创建 methodOop .....	455
8.4 Java 方法属性复制 .....	459
8.5 <clinit>与<init> .....	461
8.6 查看运行时字节码指令 .....	482
8.7 vtable .....	489
8.7.1 多态 .....	489
8.7.2 C++中的多态与 vtable .....	491
8.7.3 Java 中的多态实现机制 .....	493
8.7.4 vtable 与 invokevirtual 指令 .....	500
8.7.5 HSDB 查看运行时 vtable .....	502
8.7.6 miranda 方法 .....	505
8.7.7 vtable 特点总结 .....	508

8.7.8 vtable 机制逻辑验证 .....	509
8.8 本章总结 .....	511
<b>第 9 章 执行引擎 .....</b>	<b>513</b>
9.1 执行引擎概述 .....	514
9.2 取指 .....	516
9.2.1 指令长度 .....	519
9.2.2 JVM 的两级取指机制 .....	527
9.2.3 取指指令放在哪 .....	532
9.2.4 程序计数器在哪里 .....	534
9.3 译码 .....	535
9.3.1 模板表 .....	535
9.3.2 汇编器 .....	540
9.3.3 汇编 .....	549
9.4 栈顶缓存 .....	558
9.5 栈式指令集 .....	565
9.6 操作数栈在哪里 .....	576
9.7 栈帧重叠 .....	581
9.8 entry_point 例程机器指令 .....	586
9.9 执行引擎实战 .....	588
9.9.1 一个简单的例子 .....	588
9.9.2 字节码运行过程分析 .....	590
9.10 字节码指令实现 .....	597
9.10.1 iconst_3 .....	598
9.10.2 istore_0 .....	599
9.10.3 iadd .....	600
9.11 本章总结 .....	601
<b>第 10 章 类的生命周期 .....</b>	<b>602</b>
10.1 类的生命周期概述 .....	602
10.2 类加载 .....	605
10.2.1 类加载——镜像类与静态字段 .....	611
10.2.2 Java 主类加载机制 .....	617

10.2.3	类加载器的加载机制 .....	622
10.2.4	反射加载机制 .....	623
10.2.5	import 与 new 指令 .....	624
10.3	类的初始化 .....	625
10.4	类加载器 .....	628
10.4.1	类加载器的定义 .....	628
10.4.2	系统类加载器与扩展类加载器创建 .....	634
10.4.3	双亲委派机制与破坏 .....	636
10.4.4	预加载 .....	638
10.4.5	引导类加载 .....	640
10.4.6	加载、链接与延迟加载 .....	641
10.4.7	父加载器 .....	645
10.4.8	加载器与类型转换 .....	648
10.5	类实例分配 .....	649
10.5.1	栈上分配与逃逸分析 .....	652
10.5.2	TLAB .....	655
10.5.3	指针碰撞与 eden 区分配 .....	657
10.5.4	清零 .....	658
10.5.5	偏向锁 .....	658
10.5.6	压栈与取指 .....	659
10.6	本章总结 .....	661

# 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。

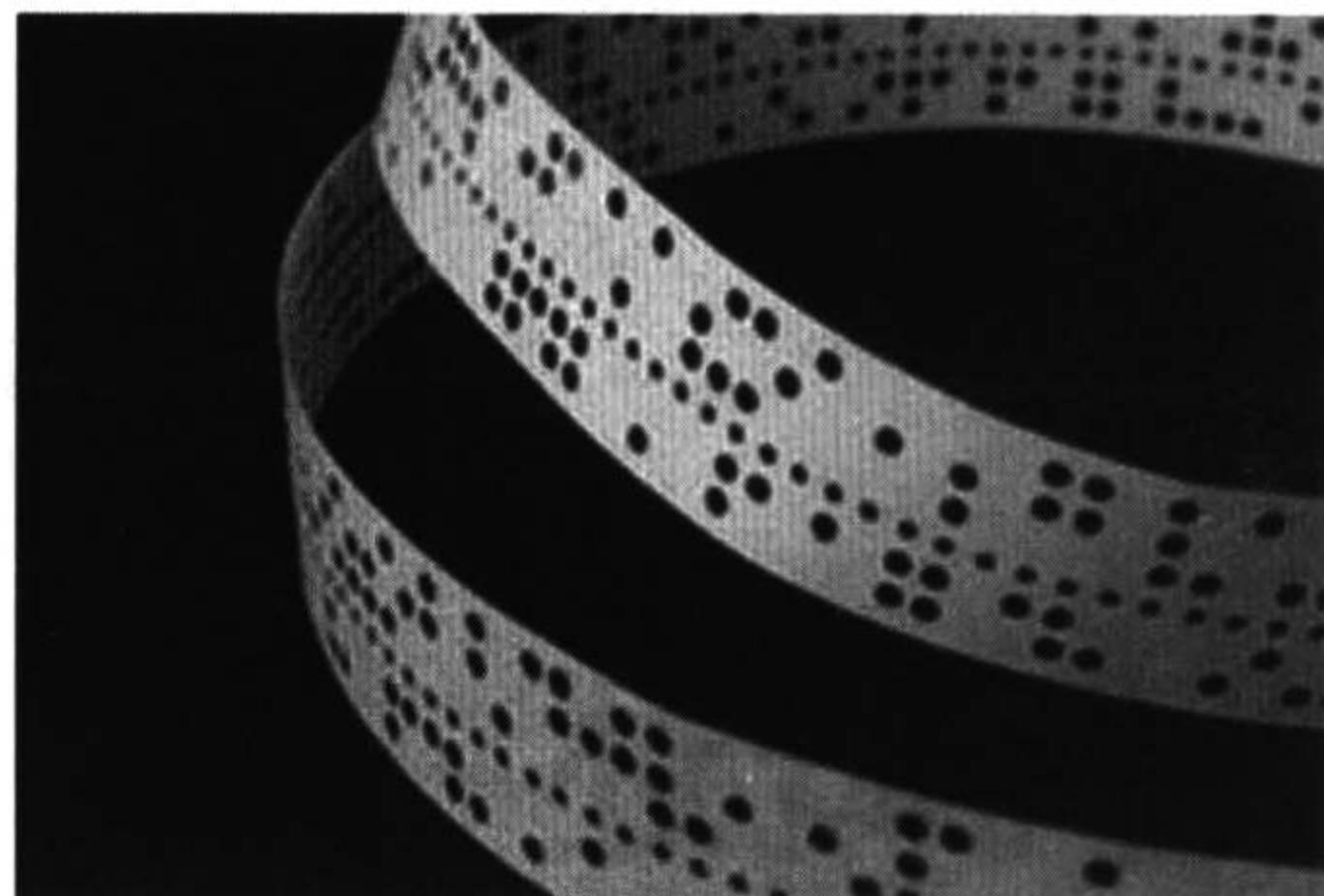


图 1.1 保存程序的卡带

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”）：

101000010000000100000000

000000110000010100000010

再看第二幅图：

101000010000000100000000

000000110000100100000010

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：

101000010000000100000000

010010110001110100000000

010010110000110100000100

0000000111010100

000000110001110100000001

1110001011110110

第二幅图如下：

101000010000000100000000

010010110001110100000000

010010110000110100000100

0000000111010100

0000000110000110100000001

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，先用大脑来执行一遍（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。

虽然那是一个不幸的年代，二战还没完全结束，全世界很多地方还有战争、饥饿和疾病，但那也是一个乱世出英雄的年代，时事造人。Grace Hopper 先生就是其中的一位牛人，他觉得这样仍然很麻烦，就开发了 A-0 system，其能够自动将助记符转换成机器码。

当时的助记符，就是所谓的汇编。

后来，人们纷纷在主流硬件平台上基于汇编进行程序开发。汇编这东西，小巧精悍，一看就懂，容易理解，上手还快，节能环保，无公害无辐射，不会引起雾霾，用过的人都说好，大家一直用它。

后来，再后来呢？再后来的事大家都知道了，聪明的前辈们先后开发出了 Fortran、B、C、C++、Delphi、VB、PHP 等各种高级语言。高级语言的出现，再一次解放了程序员的生产力，原本用汇编需要写 20 行的程序，高级语言一行代码就搞定，极大地提高了生产效率。同时，人们还开发出配套的 IDE ( Integrated Development Environment，集成开发环境)，使得调试程序和试错成为易如反掌的事，编程逐渐变得简单、有趣，从苦力活一跃成为一门艺术活。并且，高级语言的出现，为人类编写大型程序提供了坚实的基础，如果没有这些高级语言，很难想象现在会不会出现操作系统，以及基于操作系统的各种游戏、网络、电商等应用程序。

对于这段历史，如果真要去逐一追根溯源，写成的书用“汗牛充栋”来形容一点都不过分，不过所幸早有聪明的前辈把这段历史总结成了一句话：机器生汇编，汇编生 B，B 生 C，C 生万物。

这句话看着很熟悉有没有？一生二，二生三，三生万物，嗯，记起来了？你是不是发现了什么？

时间一晃四五十年过去了，我们穿越到了 20 世纪 90 年代初。那一年，整个程序界依然呈现百家争鸣、百花齐放的一派欣欣向荣的景象，各种编程语言各霸一方。那一年，地球东半球的人们普遍处于“通信基本靠吼，取暖基本靠抖”的生活水平（哈哈，有点夸张，不过那时候连手电筒都算家用电器，更别提空调和电话啦），而西半球的人们已经开始大量使用电视机、电话、闹钟、烤面包机等现代化生活电器。这些家用电器五花八门，由不同的公司制造和生产。由于大家技术各不相同，因此使用的电子芯片也不相同，大家需要针对不同的硬件进行程序开发。那一年，有个年轻的博士具有非常前瞻的商业眼光，看准了家电智能化的发展方向，便暗自下定决心，一定要运用自己所学，来实现家电智能化，开发出一种能够运行于各种不同硬件平台的编程语言，这样大家就不需要关注不同硬件平台的细节差异，只需将精力完全用在应用程序开发上面，这就能再次解放生产力，促进社会飞速发展。

这个人，就是詹爷，江湖人称“Java 之父”——詹姆斯·高斯林。

可是，要开发出这样一种能够横跨各种异构平台的编程语言，谈何容易？好比两个语言不

通的原始部落的人半途相遇，愣是急得大眼瞪小眼，王八盯绿豆，却一句话也听不懂。解决办法很多，但是关键要有第一个敢于“吃螃蟹”的人。詹爷，主意定了么？要不试试？

试试就试试，谁怕谁！

于是，在一个月不黑、风不高的夜晚，詹爷操起了那把刀，不，是打开了电脑，开启了一段创造不朽传奇的辉煌之路！

如今，20多年过去了，伊人仍在，而江湖，早已不是那个江湖。当初詹爷一心想捣鼓出个“write once, run anywhere”的编程语言来一统江湖，可是江湖局势突变，家居智能化的发展并不明朗，反而是互联网异军突起。然而，Java语言却阴差阳错地在互联网领域大展身手，乘着“互联网”这朵青云扶摇直上。到如今，Java语言已经稳坐互联网开发第一宝座很多年。近几年兴起的大数据、分布式开发，很多中间件框架和平台也都直接基于Java开发，Java语言在新的领域日渐焕发出绚烂夺目的光彩，这恐怕连当年詹爷自己都没想到。

真是世事无常！

然而，戏剧性的事情还不仅于此。在詹爷开发出Java语言的20年之后，一群志同道合者捣鼓出了一个手机操作系统——Android，其使用Dalvik虚拟机（虽然现在虚拟机被谷歌公司升级了，不再叫Dalvik了），但是语法规范完全遵循Java语言。经过谷歌公司的大力推广，Android如今占据手机市场的半壁江山，并且当下随着手机与智能家居的融合愈加深化，从某种意义上说，这反而使得詹爷当年的理想得以实现。转了一大圈，终于又转了回来。此路不可谓不曲折。

詹爷功力深厚，使用独家武功秘籍打败江湖无敌手，坐拥半壁江山（Java所占据的应用领域一直位列前茅）。我等后辈小子无才，无天赋，虽不想闻达于诸侯，却爱好追踪觅仙、寻根溯源，解密詹爷当年雄霸武林的成名功夫，其乐无穷也。

### 备注

(1) 第一个找茬游戏中，两行机器指令对应下面这段汇编（基于x86平台）：

```
mov 0x1, %eax  
add 0x2, %eax
```

(2) 第二个找茬游戏中，那段机器指令对应下面这段汇编（基于x86平台）：

```
mov 1, %eax  
mov 0, %ebx  
mov 8, %ecx  
s:  
    add %ebx, %eax  
    add 1, %ebx  
loop s
```

## 1.2 兼容的选择：一场生产力的革命

詹爷想一统江湖，希望开发出一款编程语言，能够兼容所有的硬件平台和操作系统。但是怎样实现呢？这是一个问题。

在当时，要实现兼容性，并不难。很多编程语言都具备这种能力，例如 C 语言。问题的关键在于，怎样才能在开发层面实现真正的平台无关性？

我们先看看 C 语言是如何实现兼容性的。C 语言实现系统兼容性的思路很简单，那就是通过在不同的硬件平台和操作系统上开发各自特定的编译器，从而将相同的 C 语言源代码翻译为底层平台相关的硬件指令。虽然这种思路很棒，但是仍然有明显的缺点，当涉及系统调用时，开发者仍然要关注具体底层系统的 API。例如，在 C 语言中创建一个线程，如果你是在 Linux 平台上开发，那么 C 语言程序必须要这样写：

清单：示例程序

作用：在 Linux 平台上使用 C 语言创建线程

```
#include <stdio.h>
#include <pthread.h>
//子线程函数
void threadCallBack(void)
{
    int i;
    printf("This is a pthread.\n");
}

int main(void)
{
    pthread_t id;
    int i,ret;
    ret=pthread_create(&id, NULL, (void *)threadCallBack, NULL);
    if(ret!=0){
        printf ("Create pthread error!\n");
        exit (1);
    }

    printf("This is the main process.\n");
    pthread_join(id,NULL);
    return (0);
}
```

如果你是在 Windows 平台上开发，那么 C 程序必须要这么写：

**清单：示例程序**

**作用：**在 Windows 平台上使用 C 语言创建线程

```
#include <stdio.h>
#include <windows.h>
//子线程函数
DWORD WINAPI ThreadFun(LPVOID pM)
{
    printf("This is a pthread\n");
    return 0;
}

int main()
{
    printf("This is the main process.\n");
    HANDLE handle = CreateThread(NULL, 0, ThreadFun, NULL, 0, NULL);
    WaitForSingleObject(handle, INFINITE);
    return 0;
}
```

可以看到，在 Linux 平台上，开发者需要知道 Linux 平台所提供的创建线程的接口是 `pthread_create()`；而在 Windows 平台上，开发者需要知道 Windows 平台所提供的创建线程的接口是 `CreateThread()`。另外，在 Linux 和 Windows 平台上，C 程序需要引用不同的头文件，并且所调用的创建线程的两种 API 的入参和返回值也不相同。

这就是本节一开始所说的，开发层面的平台相关性。即，开发者需要在开发时就考虑和了解各种平台的差异性。

由此可见，如果开发者想要使用 C 语言来开发一款既兼容 Linux 平台又兼容 Windows 平台的程序，不仅需要熟悉 C 语言本身的 API，也要熟练掌握 Linux 和 Windows 上的相关 API。这带来的直接后果就是，开发者为了开发一个特定平台上的功能，不得不先花费巨大的精力去熟悉该平台的特性和 API。

可能很多人对这种额外付出的成本没有概念，这里通过一组数据来对比一下，使大家对此有一种感性认识：

平台/编程语言	API 数量
C 编程语言	200 多个
Linux	300 多个
Windows	2000 多个

我们拿 C 语言举例，200 多个 API，要达到入门级水平，对于聪明的你，一定很快，你可以在分分钟内编写出第一个 demo，并在屏幕上成功打印出一行“Hello world!”。但是要精通，

事情就不简单了。你不仅要熟知这些 API 的一般用法，还需要通过大量实践，熟知里面可能存在的各种坑。更需要通过参与实际的商业项目开发，切实体会怎样才能安全、高效地使用这些 API，使你的商业程序既能拥有高性能，又能一直稳定可靠地运行下去。如果你是一个像詹爷那样拥有远大理想和崇高追求的程序员，那么你一定会追求做一个合格的架构师，你一定会基于 C 语言的 API 另外封装出自己的一套 API，使你的程序拥有良好的灵活性和扩展性。除了 C 语言的 API，还得另外熟悉 C 语言的各种语法、编译规则，利用这些规则你才能构建出一种合理的软件架构。

这么一整套东西整下来，再聪明的人，最快也需要三四年的时间，才能达到高级程序员的水平。由此可见，要想精通某一个平台或者编程语言，是一件很费时费力的事，并没有速成法，江湖上所谓的“一周精通×××”的宣传语，我们一看就知道是假的。

詹爷当年也是这么想的。人的一生是有限的，我们应该把有限的生命，投入到无限的应用开发上去，而不是浪费在无限的、层出不穷的底层细节上。

詹爷觉得开车只要学会怎样启动，怎样刹车，怎样踩油门和离合器就可以了，没必要让驾驶者关注是啥发动机，不管是涡轮增压还是自然吸气，开车的人都不需要关注，更不需要知道怎样才能操控好，那是专业车手的事。

詹爷认为，简单的，才是最美的。一个应用开发者，只需要关注功能实现本身，不应该去关注底层的细节，这样才能将生产效率提上去。

否定别人总是一件很容易的事，但要成就自己，却很难。既然通过编译器来实现兼容性，是如此地低效和费电，那应该怎样做，才能既不费电，又能实现兼容性呢？

这真是一个让人头疼的问题！

既然铁了心要解决这个问题，那就让我们再次好好分析分析问题，看看问题的焦点在哪里。现在，让我们看看所面临的问题，一共有两个问题：

- ◎ 现有的技术能够实现兼容性，但是成本太高，效率太低，太费电。
- ◎ 无法在开发层面做到平台无关性。

我们所要达成的目标也有两个：

- ◎ 实现兼容性。
- ◎ 开发者不需要关注底层平台的异构性，就能实现兼容性。

请注意我们的目标，目前两个目标里，其实兼容性已经实现了，问题的焦点就在于第 2 点，我们要做到让开发者对底层细节差异无感，能够写出可以兼容所有底层平台的程序。

还是举上面创建线程的例子。假设有这么一种编程语言，开发者编写了一条指令，说我要

创建一个线程。当编写好的程序运行在 Linux 平台上时，这条指令就自动被转换成调用 `pthread_create()` 接口；当程序运行在 Windows 平台上时，这条指令就自动被转换成调用 `CreateThread()` 接口。这样开发者就不需要关注不同底层平台上的 API 了，只需要知道有这么一条指令就可以了。

于是，中间语言（IL）就产生了。

虽然我从来没有和詹爷面对面交谈过，但我想，当年，詹爷也一定这么想过。而事实上，詹爷就是这么干的。詹爷定义了字节码规范，字节码就是中间语言指令。同时，詹爷开发了虚拟机，由虚拟机负责将字节码转换成不同平台上的特定 API 调用。由此，我们的目标终于得以达成，开发者终于不需要关注底层硬件和操作系统层面的细节，一切都由虚拟机解决，开发者只需要熟知 Java 语言规范和 API 即可实现特定功能开发，这极大地提高了生产效率，也大大降低了编程的难度和门槛，为如今全世界几百万 Java 开发者提供了饭碗。就冲这一点，我们亲切地称呼詹姆斯·高斯林为“爷”，我想大家应该都能接受吧！

OK，上面我们简单地回顾了一下当时 Java 语言产生的背景、面临的问题以及解决办法，我们知道，Java 语言从一开始就与其他语言的定位不同。现在我们知道，一款编程语言要实现兼容性，至少有两种办法：一种是通过编译器实现兼容，一种是通过中间语言实现兼容。关于这两者的区别，詹爷自己用 4 个单词做了精辟的总结，那就是：

`write once, run anywhere!`

这后一句，`run anywhere`，自然就是指兼容性，无论是编译器还是中间语言，都能实现 `run anywhere`。区别就在前一句：`write once`。使用编译器实现兼容性时，不能实现 `write once`。例如上文所举的使用 C 语言创建线程的例子，如果你的程序一开始是为 Linux 编写的，那么当你想把程序迁移到 Windows 平台上时，你必须得修改程序，把里面涉及线程创建的代码全部改成 Windows 的接口。当你的程序很大，涉及大量的系统调用时，程序中必将有很多地方都需要改写，这种改写的工作量不可谓不大，伴随而来的是高风险。你不仅要精通 Linux API，也要精通 Windows API，否则很容易一不小心中招、踩坑，为程序埋下隐患。

而使用中间语言，就用不着考虑这些想想都令人头大的问题啦！你只需要写好程序，实现程序的逻辑。程序写好后，无论你想部署到 Linux 平台上，还是想部署到 Windows 平台上，都不用修改哪怕一句代码！所有兼容性的工作都由虚拟机帮你做好了。

这种生产效率的提高绝对是革命性的！从此，Java 语言虽然没有直接走上一条康庄大道，但却在很多领域都大放异彩。“条条道路通罗马”，Java 的出现，大大缩短了商业软件的开发周期，极大地提高了 IT 业的生产效率，极大地解放了程序员的创造力。

在本节末尾，让我们一起来做个总结。

一款编程语言兼容底层系统的方式大抵上分为两种。

### 1. 通过编译器实现兼容

例如 C、C++ 等编程语言，既能运行于 Linux 操作系统，也能运行于 Windows 操作系统；既能运行于 x86 平台，也能运行于 AMD 平台。这种能力并不是编程语言本身所具备的，而是由编译器所赋予。针对不同的硬件平台和操作系统，开发特定的编译器，编译器能够将同样一段 C/C++ 程序翻译成与目标平台匹配的机器指令，从而实现编程语言的兼容性。

但是通过编译器实现兼容性时，如果涉及系统调用，往往都需要修改程序，调用特定系统的特定 API，否则程序迁移到新的平台上之后，无法运行。

### 2. 通过中间语言实现兼容

Java、C# 等语言，都属于这种兼容方式。

Java/C# 程序被编译后，生成中间语言（ML），中间语言指令由虚拟机负责解释和运行。虚拟机在运行期将中间语言实时翻译成与特定底层平台匹配的机器指令并运行。无论程序最终运行在哪种底层平台上，源代码被编译生成的中间语言指令都是相同的，中间语言的兼容性由虚拟机负责完成。

通过编译器实现兼容性，由于源代码被直接编译成了本地机器指令，因此其执行效率非常高。而这正是中间语言的软肋。Java 语言刚问世那几年，就一直因为其性能低下而被嗤之以鼻。但是随着 Java 语言版本的不断更新，随着大家对改善其性能所做出的持之以恒的努力，如今 Java 性能已经相当高，甚至比 C/C++ 程序性能还要高。这是因为 Java 虚拟机内部对寄存器进行了大量手工优化，在某些场景下，人工优化自然会比 C/C++ 编译器所做的机器优化效果要好很多。

## 1.3 中间语言翻译

上一节，我们讨论了既能实现兼容性，又能自动处理底层系统调用的又快又好的办法，那就是使用中间语言。可是 CPU 是不认中间语言的，它无法直接执行中间语言。为了使中间语言能够被 CPU 执行，虚拟机必须将其翻译成对应机器上的机器指令。

于是接下来的一个课题就是，怎样将中间语言翻译成对应的机器指令并得以执行。注意，虽然这是一个课题，但却包含两个问题哟！一个问题是怎样把中间语言翻译成对应的机器指令，另一个问题是翻译完了还要能够执行。

这里先进行总述。将中间语言翻译成对应的本地机器指令，可以使用 C 语言为每一个 Java

字节码指令写一个对应的实现函数，也可以直接为中间语言生成对应的本地机器码并通过 JMP 方式跳转到机器码来执行字节码指令。

下面我们针对这两个问题分别讲述。

### 1.3.1 从中间语言翻译到机器码

还记得本文开篇中所提到的一句话——“机器生汇编，汇编生 B，B 生 C，C 生万物”吗？

虽然现代的程序员们，手里拿着 Java、C# 等这些高端武器，沐浴在高级语言所带来的满满的幸福感中，分分钟就能编写出一个强大的程序，不需要拥有太多专业知识，仅仅依靠先进的武器，就能发出强大的威力，制敌获胜。

可是，这些高级语言的背后，却是虚拟机在辛辛苦苦、勤勤恳恳地干活，实现“万物到 C，C 到汇编，汇编到机器”的逆转换，这里的“万物”，自然也包括 Java 的中间语言——字节码指令。

将字节码指令翻译到对应的机器码，一种可行的办法是，使用 C 程序，将字节码的每一条指令，都逐行逐行地解释成 C 程序。当执行字节码的程序——JVM（Java 虚拟机）程序本身被编译后，字节码指令所对应的 C 程序被一起编译成本地机器码，于是虚拟机在解释字节码指令时，自然就会执行对应的 C 程序所对应的本地机器码。

使用语言解释起来，懂的人自然一看就明白，但是对于缺乏相关专业基础的同学，未必一下子就能看明白。所以这里举个例子进行说明。

### 1.3.2 通过 C 程序翻译

例子很简单，计算两个正整数之和。首先，假设使用 C 程序编码，程序可以写成这样：

清单：main.c

作用：C 语言求和

```
#include <stdio.h>
int run(int a, int b);

int main(){
    int a=5;
    int b=3;
    int r = run(a,b);
    printf("r=%d\n", r);
    return 0;
}
```

```

    }

    int run(int a, int b) {
        return a+b;
    }
}

```

假设我们发明了某种中间语言，该中间语言定义了一条指令来实现两个正整数相加，这条指令的助记符是 iadd，其对应的数字唯一编号是 0x01，同时假定我们开发了一款虚拟机来解释这条指令，那么解释程序将变成如下这样：

**清单： main.c**

**作用： 虚拟解释器**

```

#include <stdio.h>
int run(int a, int b, int code);

int main(){
    int a=5;
    int b=3;
    int code=0x01;//0x01
    int r = run(a,b,code);
    printf("r=%d\n", r);
    return 0;
}

int run(int a, int b, int code){
    if(code==0x01){
        return a+b;
    }
    return -1;
}

```

这段示例程序中的 run() 函数，可以看做是执行引擎（哈哈，这可能是世界上最精简小巧的虚拟机执行引擎啦）。执行引擎接收操作数和指令编码，判断指令编码是否是 iadd，如果是 iadd 指令，便对入栈的两个操作参数执行加法运算，并返回结果。

是不是很简单？

第一代 JVM 的执行引擎就是这么简单。

虽然通过 C 程序对中间语言进行解释，程序简单明了，逻辑清晰易懂，然而这种方式却有一个比较大的缺陷——效率低下。所以第一代 Java 虚拟机被广为诟病和吐槽，因为效率实在是太低了。对此，即使身怀绝技的詹爷也没有什么好的办法去优化，也不得不另外想辙。大象踩死蚂蚁是件轻松得不能再轻松的事，但不管你多么努力，一只蚂蚁也绝对不可能踩死一只大象。此路不通，那就换条道。

这辙，还真有，那就是借助于老祖宗的东西，返璞归真。既然使用 C 这种高级语言还是效率低，那就直接翻译成机器码吧，这样一定可以很快了吧！

### 1.3.3 直接翻译为机器码

将中间语言直接翻译为机器码，办法有很多。追根到底，我们还是利用了 CPU 执行代码的原理。要让 CPU 执行一段代码，只需将 CS:IP 段寄存器指向到代码段入口处即可。各位道友注意了，本节将开始接触到汇编语言了，不过不用担心，懂汇编的自然一看就明白，不懂汇编的，多看几遍也就懂了。全书很多地方都引用了汇编指令，但是加起来也不超过 5 个，不信你可以看完全书再来统计，看我有没有骗你（哈哈）。对于聪明的你，区区 5 个汇编指令还在话下吗？

这里首先解释一下啥是 CS 与 IP。这是物理 CPU 内部的两个寄存器。对于一台物理机器而言，这两个寄存器是最重要的寄存器，因为 CPU 在取指令时便完全依靠这两个寄存器。CS 寄存器保存段地址，IP 保存偏移地址。CS 和 IP 这两个寄存器的值能够唯一确定内存中的一个地址，CPU 在执行机器指令之前，便通过这两个寄存器定位到目标内存位置，并将该位置处的机器指令取出来进行运算。函数跳转的本质其实便是修改 CS 和 IP 这两个寄存器的内容，使其指向到目标函数所在内存的首地址，这样 CPU 便能执行目标函数了。Java 虚拟机要想让物理 CPU 直接执行 Java 程序所对应的目标机器码，也得修改这两个寄存器才能实现。

修改 CS:IP 段寄存器的办法有很多，既可以使用汇编直接修改，也可以在高级语言中通过语法糖的形式修改，C 语言中就有这样的语法糖（也许叫语法规则更加恰当一些，但是某种特定的写法最终都由编译器来进行转换，到了机器码层面，已经没有这些规则了，因此将其称为语法糖也未尝不可）。

C 语言中提供了一种办法，可以将 CS:IP 段寄存器直接指向到某个人口地址，这种办法就是定义函数指针（注：不是指针型函数，这两个概念相差很大，完全不同，读者千万不要搞混）。下面提供一个示例：

**清单：示例程序**

**作用：使用 C 程序提供的语法糖修改 CS:IP 段寄存器的指向**

```
#include <stdio.h>
int run(int a, int b);

int main(){
    int a=5;
    int b=3;

    int (*fun)(int,int); // 定义函数指针
    fun=(void*)run; // 初始化函数指针，使其指向 int run(int,int) 函数入口
```

```

    int r=fun(a,b); // 执行函数
    printf("r=%d\n", r);
    return 0;
}

int run(int a, int b){
    return a+b;
}

```

在本示例中，我们先定义了一个函数指针 fun，接着将其指向 int run(int,int) 函数入口，当 run() 函数被操作系统加载后，其机器码指令将被保存到代码段中，fun 指针就指向该代码段的首地址。最后，通过 int r=fun(a,b) 执行 fun，由于 fun 指向 run() 函数入口，因此系统最终执行的实际上是 run() 函数。

在 Linux 上运行这段程序，最终屏幕上将正常输出“r=8”这行文字。

其实，`fun=(void*)run` 这句代码之所以得以正常运行，完全是因为编译器提供了这种语法支持，因此我们可以将其视为 C 语言的语法糖。这句代码被编译后，实际上相当于修改了 CS:IP 段寄存器的指向，因此当 CPU 运行到这里后，由于 CS:IP 指向到了 run() 函数首地址，因此程序就跳转到 run() 函数。

上面讲了一堆，可是这与我们本节的主旨有何干系呢？我们本节要解决的问题是，如何将中间语言直接翻译成机器码。虽然本示例与本节主旨似乎没有什么关系，但是通过本示例，我们知道如何使用 C 语言所提供的语法糖来修改 CS:IP 段寄存器，有了这个法宝，我们就能实现我们的目标。因为 C 语言程序最终也会被编译成本地机器指令，因此我们可以预先定义一串本地机器指令，然后直接通过 C 语言所提供的语法糖，将 CS:IP 段寄存器指向这串机器指令，这样 C 程序就能直接动态执行机器码。

好了，废话少说，下面直接上菜：

### 清单：示例程序

作用：使用 C 程序提供的语法糖，让 CS:IP 直接指向一串机器码

```

#include <stdio.h>

/**
 * code 数组中保存的一串数据，正好是机器码指令编码，CPU 可以直接执行
 * 注：
 * (1) 对于计算机，一个数据既可以被当作指令执行，也可以被当作一个数据
 * (2) 当这个数据被 CS:IP 段寄存器指向时，就可以当作代码执行
 */
const unsigned char code[] = "\x55\x89\xe5\x8b\x45\x0c\x8b\x55\x08\x01\xd0\
\x5d\xc3";

```

```

int main() {
    int a=5;
    int b=3;

    int (*fun)(int,int); // 定义函数指针
    fun=(void*)code; // 初始化函数指针, 将其指向 code 机器码的入口

    int r=fun(a,b);
    printf("r=%d\n", r);
    return 0;
}

```

本例所要实现的功能与上例一样，都是实现对两个正整数求和。

本例也同样使用了 C 语言所提供的语法糖，通过 `fun=(void*)code` 这样的方式达到间接修改 CS:IP 段寄存器的指向的目的。但是本例与上例的不同之处在于，在上例中，`fun` 指针是直接指向了 `int run(int,int)` 函数，而本例中，`fun` 指针却是指向了一个 `char` 数组首地址。

`char` 数组中保存了一串数字，这些数字其实是 x86 平台上的一串机器指令，因此 CPU 可以直接将其当做指令来执行。

这是一个特别神奇的神器！很多大牛都用它实现了很多匪夷所思的功能。集中起来而言，就是大家用它实现了各种动态功能，各种在运行期动态修改程序走向的功能。有了这件神器，要实现一个高可扩展性的架构，再也不是难事了！

詹爷早就看好了这件神器，并且是此中高手。在 JVM 的后续版本中，正是这件神器，挽救了 Java 语言，使其性能得到突飞猛进的提高，在某种程度上快赶上甚至超越 C 和 C++ 语言的速度了，其助推 Java 语言大步向前进！

咦？貌似还少了啥事？

刚才只顾夸夸其谈，忘了本节主旨了，我们还没有实现由中间语言直接翻译为机器码的伟大目标呢！但是我相信，聪明的你一定早就想到了办法。对，果然与你想的一样，我们既然都能在 C 语言中直接动态执行机器码了，注意，是动态哟！我们只要将中间语言指令直接翻译为机器码，然后让 CS:IP 直接指向这段机器码，问题不就解决了吗？

是的，事情的确就是这么简单！现代 JVM 的确就是这么干的。

不过，在 JVM 里面又不完全是这么干的，但这不是很重要，重要的是，我们已经有办法、有能力完成将中间语言直接翻译成机器码并动态执行的宏伟目标，这是一个具有战略意义的里程碑，决定生死存亡！

---

注：本示例中，`code` 数组中所保存的一串数据，其实对应下面一组机器指令：

```

55          push    %ebp
89 e5        mov     %esp, %ebp
8b 45 0c      mov     0xc(%ebp), %eax
8b 55 08      mov     0x8(%ebp), %edx
01 d0        add     %edx, %eax
5d          pop     %ebp
c3          ret

```

### 1.3.4 本地编译

虽然将中间语言直接翻译为机器码并直接运行，其效率相比使用 C 语言来解释执行，已经提高了很多，但是，由于中间语言有自己的一套内存管理和代码执行方式，因此，实现同样的功能，虽然使用中间语言只需写几行代码，但是翻译后的机器码，比直接编写机器码，还要多出很多指令。指令数量增多，意味着在同样的硬件平台上，执行时间成本必然增加，因此其运行效率仍然不够高。

即使与同样属于高级语言的 C 语言相比，它们实现相同的功能，C 语言编译后所生成的机器码，也比中间语言直接翻译成的机器码，在数量上要精简很多。

例如，使用 C 语言对两个正整数求和。示例程序如下：

**清单：示例程序**

**作用：**使用 C 语言求和

```

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

```

本地编译该程序，得到的机器码如下（使用汇编助记）：

**清单：示例程序**

**作用：**C 语言求和程序编译后的机器码

```

pushl  %ebp
movl %esp, %ebp
movl 12(%ebp), %eax
movl 8(%ebp), %edx
addl %edx, %eax
popl %ebp
ret

```

可以看到，add()函数总共仅包含 7 条机器指令，即可完成求和运算，如果去掉 pushl %ebp 等入栈、出栈的辅助性指令，只需要下面 3 条机器指令即可完成：

**清单：示例程序**

**作用：C语言求和程序编译后的机器码**

```
movl $12(%ebp), %eax
movl $8(%ebp), %edx
addl %edx, %eax
```

而如果使用中间语言来执行求和运算，翻译后得到的机器码指令将会多出几个数量级。这里以 Java 为例。首先，编写 Java 源程序：

**清单：示例程序**

**作用：Java语言求和程序**

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

这段 Java 程序编译后生成的字节码如下：

```
0: iload_1
1: iload_2
2: iadd
3: ireturn
```

而每一条字节码指令最终都会对应一大堆机器指令，机器指令的数量远超 C 语言编译后的机器指令数量。

由此可见，中间语言由于其本身不能直接被 CPU 执行，为了能够被 CPU 执行，中间语言在完成同样一个功能时，需要准备更多便于自我管理的上下文环境，最后才能执行目标机器指令。准备上下文环境最终也是依靠机器码去实现，因此中间语言最终便生成了更多机器码，当然执行效率就降低了。

为了能够进一步提升性能，JVM 提供了一种机制，能够将中间语言（字节码）直接编译为本地机器指令。

可能聪明的你马上会想到这样一个问题，既然中间语言在运行期能够被逐个直接翻译成机器码，那么在编译期不也能吗？例如对于 Java 源代码，可以先在本地编译成字节码，再将字节码指令逐个替换为机器指令，这样最终不就生成了可直接被 CPU 执行的、由机器码组成的程序了吗？

这个思路的确可以一试，例如安卓和部分 JVM 所实现的 AOT (ahead of time) 特性便是这方面的尝试，但是这种方式并没有减少机器指令的数量级问题。

事实上，JVM 的大牛们在 JIT (即时编译)、内存分配等方面倾注了大量心血，想出了很多

天马行空而又切实可行的好主意，能够对热点代码进行大幅度指令优化，将 Java 程序的执行效率大幅提升。正是由于 JVM 可以在运行期基于上下文链路进行各种优化，因此优化后的指令质量比 C/C++ 编译出的指令质量更高，因此才会有部分 Java 程序性能反而超过 C/C++ 程序的现象。如果离开了这些动态优化，Java 程序的执行效率是无论如何也提不上去的。

## 1.4 神奇的指令

Java 语言想要一统江湖，兼容各种平台，又要实现“write once, run anywhere”的伟大梦想，只有依靠“中间语言”这一条通道了。思路是有了，但是具体如何执行，或者说如何才能实现呢？中间语言究竟长啥样，谁也不知道。

前面讲过，在使用 C 语言或汇编或 C++ 进行底层开发时，必须熟悉硬件平台本身所提供的指令，或者熟悉底层软件平台所提供的 API。在不同的平台上实现同样一种功能，需要调用不同的底层接口或指令。同时，编译器也直接依赖于平台，大家需要为不同的平台开发不同的编译器。

詹爷看着这样一种现状，口中轻轻说出一个字：NO！

詹爷觉得这种方式简直太 low 了，他觉得一款好的编程语言应该是这样的，不管程序员在哪种软硬件平台上开发，如果要实现同样一个功能，他只需要调用同一个接口，他不需要感知这个平台与其他平台之间有什么差异。至于把统一接口翻译成对应的机器的指令这事，就交给虚拟机来做吧，程序员不要再关注这事了。接口统一了，就相当于两个不同的机器说的是同一种语言，这样来实现跨平台岂不是轻而易举？如果詹爷是上帝，要他出访世界各国，他肯定不愿意学习各个国家的语言，他一定会带着精通世界各国语言的随从，他只讲国际语言，具体翻译的工作，就交给随从干吧。这个随从就类似于 Java 虚拟机。

从“统一接口”这个角度看，中间语言应该是你知、我知、所有的开发者都知道，但是唯独底层的物理 CPU “不知道”的一种语言。这样看来，Java 语言本身岂不就符合“中间语言”的标准？可是别忘了，中间语言还得有一个必须什么都懂的人，这个“人”就是 Java 虚拟机啦。虚拟机就是全世界广大程序员免费而又无比专业的贴身翻译，它必须负责将这种“中间语言”精确地翻译成对应机器平台的机器指令。可是很遗憾，虚拟机这位“贴身翻译”读不懂 Java 程序。伤心啊！

所以，Java 语言不是中间语言。

为啥虚拟机读不懂 Java 程序呢？这还得从语言的人性化谈起。高级编程语言之所以高级，就在于其语法和表达规范遵循人类的思维习惯，但是这不符合机器的思维，即使虚拟机也不行。

尤其是 Java 语言，由于比其他语言更加“面向对象”，其字面含义带有更加彻底的人类主观色彩，但是机器就不能理解了，机器只认得内存、堆栈，其他一概不认，所以 Java 虚拟机读不懂 Java 语言倒也情有可原。不过也并非所有的虚拟机都不懂得面向对象的语言，JavaScript 执行引擎就是个例外——JS 脚本不需要编译就能被 JS 引擎直接执行，虽然这么玩也行，但是 JS 的执行引擎不跨平台啊，这与 Java 语言的远大理想相去甚远，道不同不相为谋。

既然 Java 语言本身不能作为中间语言，可是中间语言又是理想的实现跨平台的技术方向，这可怎么办呢？这个问题让我们的大脑很费电啊！不过詹爷觉得 so easy，解决办法很简单，为虚拟机这位“贴身翻译”再配一个翻译，负责将 Java 程序翻译成理想中的“中间语言”不就行了？好，那就这么定了，咱就开发个编译器吧，通过编译器将 Java 语言翻译成中间语言，然后再交给虚拟机，其再将中间语言翻译成对应机器平台上的指令。

解决了中间语言怎么实现的问题之后，詹爷要做的一件重要的事就是，决定中间语言长啥样。最终的结果大家都知道了，这个所谓的中间语言就是 Java 字节码指令集，没错，这就是中间语言。虽然 Java 语言是面向对象的，符合人类思维习惯的，但是 Java 指令却是刻板的，不知啥是对象，只知道压栈，读写局部变量表，调用目标方法，等等。

詹爷定义的通用指令集如下：

```
iconst_0, 将自然数 0 压入操作数栈。  
iconst_1, 将自然数 1 压入操作数栈。  
iload_0, 将索引为 0 的局部变量表的 int 型数据压入操作数栈。  
istore_1, 将操作数栈栈顶的 int 型数据写入索引为 1 的局部变量表中。  
// ...
```

在 JVM 源代码中，定义了 Java 语言的全部指令集，如果你平时稍微接触过字节码，那么你对上面贴出来的这部分指令一定不陌生，例如，`iconst_0`、`istore_1`，等等。

很多人一定还知道，Java 的所有指令都使用 8 位二进制描述，因此，Java 的指令总数不超过 255 个。对上面这套指令集，你看不懂也没关系，但是多少瞄上一眼，并没有任何坏处，好歹先混个脸熟，艺多不压身。

有了这套通用的指令集，一统江湖就有希望了。不过这看似简单，实则不简单。表面看来，一共才区区 200 多个指令，但这正是作者深厚功力的体现。大家都知道，指令集一般是计算机硬件才有的东西，而作者却在软件层面定义了一套同样的东西。但是，软件本身不具备执行程序的能力，软件最终还得依靠硬件指令才能完成逻辑计算。因此，一套好的软件指令必须不能超出硬件指令所能表达的计算能力，同时又要对硬件指令进行高度抽象与概括。换言之，如果你定义了一套与硬件指令集完全一模一样的软件指令集，那大家还用你干嘛呀，不如直接用硬件指令得了。

所以，詹爷所设计的指令集必定有其独到之处。那么独特在哪里呢？不急，本书后续很多内容都会涉及 Java 指令集，有的是时间，慢慢品味。在品味之前，我们得先普及一下必备的专业知识。如同古玩收藏家，就算给你一幅《滕王阁序》的真迹，但是如果你不识货，你也会当成赝品，糟蹋了天物。古玩家玩金，玩玉，玩石，玩各种器具，靠的是眼观、耳听、鼻闻、舌舔，咱们玩詹爷的指令集，不用那么讲究，只要用心即可，并且你还不用担心假冒伪劣问题，詹爷的指令集绝对货真价实，绝对正品，假一赔十。

### 1.4.1 常见汇编指令

由于本书主要讲解 Java 虚拟机执行引擎的内部实现机制，而 Java 虚拟机的执行引擎有太多地方直接使用了机器码实现，因此本书很多地方都不可避免地会接触到汇编指令。不过汇编指令并没有想象中的那么可怕，至少不太聪明的我在看了几十遍之后也能看懂不是（哈哈）？下面简单介绍 5 个常见的汇编指令，也就 5 个，熟悉了这几个指令，虽说不能让你在 JVM 的世界里“横着走”，但也能让你在 JVM 的世界里“行走江湖、独步天下”了。

大部分机器指令集都支持以下 5 类计算。

#### 1. 数据传送指令

这些指令主要在寄存器与内存、寄存器与输入/输出端口之间传送数据，例如：

```
// 将自然数 1 传送到 eax 寄存器  
movl 1, %eax  
  
// 将栈顶数据弹出至 eax 寄存器  
pop %eax
```

---

注：由于机器指令是二进制，写出来谁也看不懂，因此这里使用汇编指令代替，下面也全部这样表达。汇编本来就是机器指令的助记符。

---

可以这么说，数据传送指令几乎是任何硬件系统都必须支持的指令。

#### 2. 算术运算指令

包括算术基本四则运算、浮点运算、数学运算（正弦、反弦等）等。例如：

```
// 将自然数 3 与 eax 寄存器中的数累加，并将结果存储进 eax 中  
add 3, %eax  
  
// 对 ebx 寄存器中的数增 1  
inc %ebx
```

### 3. 逻辑运算指令

与、或、非、左移、右移等指令，都属于逻辑运算指令。例如：

```
// 将 eax 中的数左移 1 个二进位  
shl %eax, 1  
  
// 对 al 寄存器中的数和操作数进行与操作  
and al, 00111011B
```

### 4. 串指令

连续空间分配，连续空间取值，传送等，都要使用串指令。很多高级编程语言都支持字符串运算，如果硬件没有串指令，不敢想象计算机的世界会变成什么样。

### 5. 程序转移指令

If…else 判断、for 循环、while 循环、函数调用等，都需要依靠程序转移指令，否则程序无法跳转。没有这些指令，程序不能模块化，无法被分隔成一个一个方法，更无法通过循环来解决很多重要的问题。常见的程序转移指令包括 jmp 跳转、loop 循环、ret 等。

好，基本专业知识的扫盲到这里先告一段落，接下来我们就可以先简单品味一下詹爷的虚拟指令集了。作为一个有良心的作者，有必要在这里先善意提醒一下大家，在研究 JVM 源码的过程中，肯定不会绕过汇编这一道坎，所以，本书会时不时地讲点汇编知识，但不会一下子讲很多。你呢，如果看不懂，也不要担心，我会尽量使用通俗易懂的方式让你懂。如果你实在看不懂，就权当是看小说罢了，无论什么语言，只要理解了其中的道理，其实都一样一样的。当然，如果能看懂，那是最好不过的，你会感受到更多的乐趣，你会感叹，啊，原来程序是可以这么写的，原来内存是可以这么划分的，原来函数是可以这么调用的！

## 1.4.2 JVM 指令

---

詹爷的指令集比上述硬件指令集更加丰富，这是因为 Java 是面向对象的编程语言，自然要有一套支持类型操作的特殊指令。总体而言，詹爷设计的指令集分为以下几部分（可能不同的人有各自不同的一套分类标准，但笔者认为基本大同小异，看你从哪个角度来分类）。

### 1. 数据交换指令

对 JVM 稍有了解的人都知道，JVM 内存分为操作数栈、局部变量表、Java 堆、常量池、方法区。既然有这些内存区域，那么必须要有指令支持数据在这些内存区域之间的传送和交换。例如，当你在 Java 方法中访问一个静态变量时，那么其运算过程必然伴随 JVM 将数据从常量

池传送到操作数栈的指令调用。这与硬件指令不同。以 x86 为例，一个在硬件上直接执行的程序，其内存一般分为寄存器、数据段、堆栈、常量区、代码段，CPU 为了完成运算，必然要涉及将数据从这些内存区域传送到寄存器的指令调用。

JVM 执行逻辑运算的主战场是操作数栈（`iinc` 指令除外，该指令可以直接对局部变量进行运算），注意，是操作数栈哦，很重要哦！不管你把数据放在堆栈中，还是放在常量池中，你要执行运算，最终 JVM 都会将数据传送到操作数栈中。而硬件执行逻辑运算的主战场是寄存器，不管你把数据放在数据段中，还是代码段，最终 CPU 都会将数据传送到寄存器中。逻辑运算完成后，再把结果转移出去。

这段文字看着有些抽象，若是配上例子，那是极好的。既然提到了战场，就拿打仗来说事。假设主战场定在 X 地区，为了取得战争胜利，作战方需要源源不断地将军队、火药、枪支、后勤物资、医疗队等都运到 X 地区。X 地区就好比是 JVM 的操作数栈，而军队、火药、物资等，都可以认为是数据。作战司令通过一道道军令来调动军队、火药、物资等，军令就类似于计算机指令。只要作战命令一下达，整个系统立马就运作起来了，一道道军令将军队、火药、物资等“数据”从各个地区“传送”到了战场上，然后又有一道道军令，将战场上的“数据”转移出去。所以，指挥作战就像编程序，或者反过来，你把编程想象成打仗，倒也是一件很有趣的事，化枯燥为神奇。其实，历史上那些伟大的军事家最后都会总结出这样一句话：战争就是一种艺术。同样，虽然计算机只有短短几十年的历史，但无数牛人也都总结出一句话：编程，就是一门艺术活儿。看来，天下牛人一般牛，无论玩什么，最后都能玩出艺术之美，这才是玩家的最高境界。

JVM 标准提供了丰富的数据交换指令，例如 `iload`、`istore`、`lload`、`lstore`、`fload`、`fstore`、`dload`、`dstore`、`ldc`、`bipush` 等指令，詹爷用这些指令来实现操作数栈和局部变量表之间的数据交换（这些在后文都会讲到）。JVM 规范还提供了像 `getfeild` 和 `putfeild` 这样的指令，詹爷用这些指令来实现 Java 堆中的对象的字段和操作数栈之间的数据交换。JVM 规范还提供了像 `getstatic` 和 `putstatic` 这样的指令，詹爷用这些指令来实现类中的字段和操作数栈之间的数据交换。JVM 规范还提供了像 `baload`、`bastore`、`caload` 和 `castore` 这样的指令，詹爷用这些指令来实现 JVM 堆中的数组和操作数栈之间的数据交换。

至此，聪明的你应该能够想到，JVM 并不是随随便便就将内存分成了操作数栈、局部变量表、Java 堆、常量池、方法区这几个区域的，人家都是有专门的指令在后面默默支撑的。或者说，既然你把这些内存区域给硬生生地“生”出来了，那么你就必须要对人家负责，你必须设计指令对这些区域进行管理。

## 2. 函数调用指令

函数调用指令集可以归入到“程序转移指令集”中，也可以单独拿出来说事。由于 Java 中

的函数类型比较丰富，因此必然要支持更多的函数调用方式。詹爷设计了多个函数调用指令，例如，`invokevirtual`、`invokeinterface`、`invokespecial`、`invokestatic` 和 `return` 等。这比硬件所支持的函数调用指令集要丰富一些。以 x86 为例，x86 中主要使用 `call` 指令和 `ret` 指令来保存现场和恢复现场，这往往会伴随 CPU 物理寄存器入栈和出栈。

JVM 没有物理寄存器，所以用操作数栈和 PC 寄存器来替代。注意，这里又提到了操作数栈哦，重要的事说三遍，何况这么重要的概念！后面会反反复复提到这个概念哟。JVM 保存现场和恢复现场的解决方案是向 Java 堆栈中压入一个栈帧，函数返回的时候从 Java 堆栈中弹出一个栈帧。

JVM 调用函数的时候，不能像 CPU 硬件那样，直接跳转就能找到对应的代码段。这是因为 Java 函数的代码并没有被存放到代码段中，而是被放在了一个 `code` 缓存中，每一个 Java 函数的代码块在这个 `code` 缓存中都会有一个索引位置，最终 JVM 会跳转到这个索引位置处执行 Java 函数调用。同时，Java 的函数一定是被封装在类中的，因此 JVM 在执行函数调用时，还需要通过类寻址等等一系列运算，最终才能定位这个入口。

如果你没有汇编基础，并且对 JVM 也从来没有了解过，那么你在看这段话的时候，一定会晕。如果看不懂，没关系，后面会对照 JVM 源码进行详细讲解，所以这里看不懂很正常哦，千万不要灰心哟！而能够看得懂的童鞋们，自然会“会心一笑”，我知道你此刻心里挺美。

### 3. 运算指令集

JVM 和运算相关的指令集主要有算术运算、位运算、比较运算、逻辑运算等，JVM 还为各种基本类型的运算提供不同的操作码；x86 也有算术运算、逻辑运算、位运算、比较运算，但是所有的操作都是直接针对寄存器中的二进制数进行的，不区分数据类型。

JVM 规范中常见的运算指令包括 `iadd`（对两个 int 型数据求和）、`isub`（对两个 int 型数据做减法）、`fadd`（对两个 float 浮点数进行求和）、`ddiv`（两个 double 双精度型数据相除）等。

### 4. 控制转移指令

与硬件 CPU 一样，JVM 规范也提供了常见的控制转移指令，例如 `switch` 分支选择指令、`if...else` 条件判断、`do...while` 循环、`for` 循环、`foreach` 循环、`return` 返回、`break` 中断循环、`continue` 继续循环。不多讲，大家都懂的。

### 5. 对象创建与类型转换指令

作为一门面向对象的语言，JVM 规范自然要提供一套创建对象的指令。在 Java 语法层面使用关键字 `new` 可以实例化一个对象，而对应的字节码指令也是 `new`。

JVM 规范还提供了“窄化类型转换”指令，与“窄化类型转换”指令相对的是“宽化类型转换”指令，只不过后者是 JVM 内部天生支持的，不需要另外使用指令。

除了以上这些指令，JVM 规范还提供了很多其他物理 CPU 所没有的指令，例如，抛出异常的指令，用于线程同步的指令，等等。

由此可见，Java 字节码指令真的很神奇，足够简单，但又足够强大。字节码指令是中间语言的一种实现手段，虽然肯定还有其他技术路径。字节码指令能够完成 Java 语言的各种功能，能够压栈和出栈，能够读写局部变量表，能够调用方法，也能够创建对象实例。而最关键的一点是，它是跨平台的。这就是它很神奇的根本原因。

## 1.5 本章总结

如果你已经认认真真地看到了这里，并且你基本能理解所讲内容，那么恭喜你，你不知不觉中已经弄明白了 JVM 最核心的部分——run engine（执行引擎）。

如果你本身就拥有汇编基础、C 语言编程基础，本章所举的例子你都能看明白，或者你在自己的机器上亲手实践了一部分示例程序，那么更要恭喜你，你基本可以毫无障碍地看明白整个 JVM 核心的执行引擎模块了。JVM 核心的执行引擎虽然纷繁复杂，但是那只不过是同类功能的简单堆砌，将同类项合并后，它的基本核心技术其实都是本章所述内容。

本章简单介绍了 Java 语言产生的历史背景。Java 语言所要解决的是如何能够不关注底层技术细节就能实现兼容性，詹爷给出的答案是使用中间语言，通过中间语言来实现跨平台兼容的目标。由于中间语言并不是本地机器指令，机器 CPU 无法直接识别，因此中间语言并不能直接由物理 CPU 运行，那怎么办呢？很简单，使用虚拟机来解释中间语言，将中间语言翻译成对应的本地机器指令。

将中间语言翻译成本地机器码的方式有很多种，例如使用 C/C++ 语言为每一个 Java 字节码指令写一个对应的实现函数。但是这种方式太低效了。而解决低效的一种机制就是直接将 Java 字节码指令翻译成本地机器指令，运行期直接由 Java 虚拟机调用对应的机器指令来执行，这种调用的机制主要就是依靠 CPU 所提供的 call 和 jmp 指令。

中间语言长啥样？外表长得确实不够“漂亮”，很多人看了一眼就不愿意继续看第二眼，但是它很有内涵，能量足够大，能够跨平台。它就是 Java 字节码指令集。该指令集就是中间语言。这个指令集是对硬件 CPU 指令集的抽象与再加工，能够满足 Java 开发的一切需要。

# 第 2 章

## Java 执行引擎工作原理：方法调用

---

### 本章摘要

- ◎ JVM 如何进行方法调用
- ◎ JVM 如何分配方法栈
- ◎ JVM 如何取指
- ◎ JVM 如何执行逻辑运算

JVM 作为一款虚拟机，也必然要涉及计算机核心的 3 大功能。

#### 1. 方法调用

方法作为程序组成的基本单元，作为原子指令的初步封装，计算机必须能够支持方法的调用。同样，Java 语言的原子指令是字节码，Java 方法是对字节码的封装，因此 JVM 必须支持对 Java 方法的调用。

#### 2. 取指

这里的“取指”，是指取出指令。

还是那句话，方法是对原子指令的封装，计算机进入方法后，最终需要逐条取出这些指令并逐条执行。Java 方法也不例外，因此 JVM 进入 Java 方法后，也要能够模拟硬件 CPU，能够从 Java 方法中逐条取出字节码指令。

### 3. 运算

计算机取出指令后，就要根据指令进行相应的逻辑运算，实现指令的功能。JVM 作为虚拟机，也需要具备对 Java 字节码的运算能力。

本章主要分析 JVM 如何从内部调用 Java 方法。

## 2.1 方法调用

到目前为止，人类发明出了若干种编程语言，有的编程语言没有类概念，有的编程语言面向过程，但不管是哪种编程语言，至少都会包含函数的概念。通过函数将一个大的程序拆分成体积小、功能明确的一个个简短的函数，从而将一个复杂的大型问题分解成若干个简单的小问题，由繁到简。虽然函数并不总是大型软件模块化的手段，但一定是模块化得以实现的基础，否则随便开发个稍微难一点的功能，一写就是几千、几万行代码，估计没几个人能看懂，更没几个人有耐心看。

同理，Java 程序最基本的组成单位是类，而 Java 类也是由一个个的函数所组成，在这一点上，Java 也玩不出什么花样。

有的编程语言由真实的物理机器运行，有的程序运行于虚拟机上。既然所有的编程语言都由函数组成，那么运行由这些编程语言所开发出来的程序的机器就必须能够执行函数调用，不管是物理机器还是虚拟机器。JVM 作为一款虚拟机，要想具备执行一个完整的 Java 程序的能力，就必定得具备执行单个 Java 函数的能力。而要具备执行 Java 函数的能力，首先必须得能执行函数调用。

经过前面的讨论我们知道，詹爷当年为了能够让 Java 这门编程语言兼容各种平台，最终使用了一个大招——在运行时将 Java 字节码指令动态翻译成本地机器指令，从而既能获取兼容性，又能获取很高的运行效率。因此，JVM 实际上最后调用的并不是真正的 Java 函数，而是其对应的一堆机器指令。那么 JVM 究竟是怎么做到直接调用机器指令的呢？要研究清楚这个问题，必定要先弄清真实的物理机器是如何调用机器指令的。下面让我们简单了解一下真实的物理机器执行函数调用的机制。

### 2.1.1 真实的机器调用

本节主要通过一个汇编程序讲解一些真实的机器调用原理，若有道友看不懂也没有关系，多看几遍就懂了（哈哈）。想研究 JVM 的执行引擎原理，汇编这道坎必须得过，别无他法。

真实的机器指令调用机制涉及的知识比较多，例如，现场保存、堆栈分配、参数传递，等等。我们不需要知道那么专业的基础知识，只需要了解大体的原理，了解了这些原理，再理解JVM的函数调用就会变得简单了。废话少说，翠花，上菜！

下面这段程序是使用汇编编写的，程序功能很简单，对2个整数进行求和（注：由于直接写机器指令，相信没人能够看得懂，因此这里以机器指令的助记符——汇编语言进行演示）。例子如下：

**清单：示例程序**

**作用：使用汇编进行求和**

```

main:
    pushl %ebp
    movl %esp, %ebp
    subl $32, %esp
    movl $5, 20(%esp)
    movl $3, 24(%esp)
    movl 24(%esp), %eax
    movl %eax, 4(%esp)
    movl 20(%esp), %eax
    movl %eax, (%esp)
    call add
    movl %eax, 28(%esp)

    movl $0, %eax
    leave
    ret

add:
    subl $16, %esp
    movl 12(%ebp), %eax
    movl 8(%ebp), %edx
    addl %edx, %eax
    movl %eax, -4(%ebp)
    movl -4(%ebp), %eax
    leave
    ret

```

如果你不具备汇编基础知识，看不懂汇编程序，那么很难理解上面这段汇编程序的逻辑。不过看不懂没关系，只要能够明白其大意即可，不管哪种编程语言，其操作的无非是内存、数据，因此能够明白这段程序对内存做了什么，就可以了。

这段汇编程序想要实现的功能很简单，就是对两个整数进行求和。

分析一段程序，一般首先看该程序由哪些模块组成，分析汇编程序也是一样。这段汇编程

序在代码段中定义了两个标号，一个是 main 标号，一个是 add 标号。汇编语言中的标号类似于 C 语言中函数的概念，这里我们就当成函数好了。那么这段汇编程序中就定义了两个函数，一个是 main() 函数，一个是 add() 函数。看到这里，也许聪明的你很快就猜到，这段程序是不是在 main() 函数中定义了两个变量，然后传递给 add() 函数，由 add() 函数执行加法运算呢？恭喜你，猜对了。

既然明白了程序的大体意思，下面来一起分析下具体的算法。在这个过程中，如果你对汇编语法不怎么了解，也没关系，我们只需要关注最终内存是怎样变化的就可以了。

## 1. main() 函数详解

首先看 main() 函数。我们先整体解释下 main() 函数。

**清单：示例程序**

**作用：**使用汇编编写一段求和程序

main:

```
// 保存调用者栈基址，并为 main() 函数分配新栈空间
pushl %ebp
movl %esp, %ebp
subl $32, %esp           // 这里就是分配新栈，一共 32 个字节

// 初始化两个数据，一个是 5，一个是 3
movl $5, 20(%esp)
movl $3, 24(%esp)

// 压栈：将 5 和 3 压入栈中
movl 24(%esp), %eax
movl %eax, 4(%esp)
movl 20(%esp), %eax
movl %eax, (%esp)

// 调用 add() 函数
call add
movl %eax, 28(%esp) // 得到 add() 函数的返回结果

// 返回
movl $0, %eax
leave
ret
```

看过 main() 函数的代码注释后，我们知道，main() 函数一共包含 5 步：保存调用者栈基地址，初始化数据，压栈，函数调用和返回。下面分别分析这 5 步过程。

### 1) 保存栈基并分配新栈

首先看第一步，main()函数从下面两条指令开始执行：

```
pushl %ebp  
movl %esp, %ebp
```

pushl %ebp 就是保存调用者的栈基地址。调用者是谁？谁能调用一个程序的主函数？当然是操作系统啦。movl %esp,%ebp 将调用者的栈基地址指向其栈顶。这两句是所有函数调用时都必定会执行的指令。add()函数的开头也是这两条指令。待会儿讲 add()函数调用时会再讲到它们，如果你不懂，可以先跳过。

执行完上面两条指令后，main()函数接下来执行下面这条指令：

```
subl $32, %esp
```

这条指令是干嘛用的？大家整天在讲分配栈空间，分配栈空间，这条指令就是干这事的。所以对于物理机器而言，分配堆栈空间非常容易，就一句话的事。这条指令中的 subl 表示减，指令中的%esp 表示当前栈顶。整条指令的含义是：将当前栈顶减去 32 字节的长度。为什么是减，而不是加呢？这是因为在 Linux 平台上，栈是向下增长的，从内存的高地址往低地址方向增长，因此每次调用一个新的函数时，需要为新的函数分配栈空间，新函数的栈顶相对于调用者函数的栈顶，内存地址一定是低位方向，因此新函数的栈顶总是通过对调用者函数的栈顶做减法而计算出来。

执行了这条指令后，main()函数就有了自己的方法栈啦，栈空间大小是 32 字节，一个字节包含 8 个二进制位，如果一个 int 类型的整数包含 4 字节的话，那么 main()函数的方法栈就一共可以容下 8 个 int 类型的数据（如图 2.1 所示）。

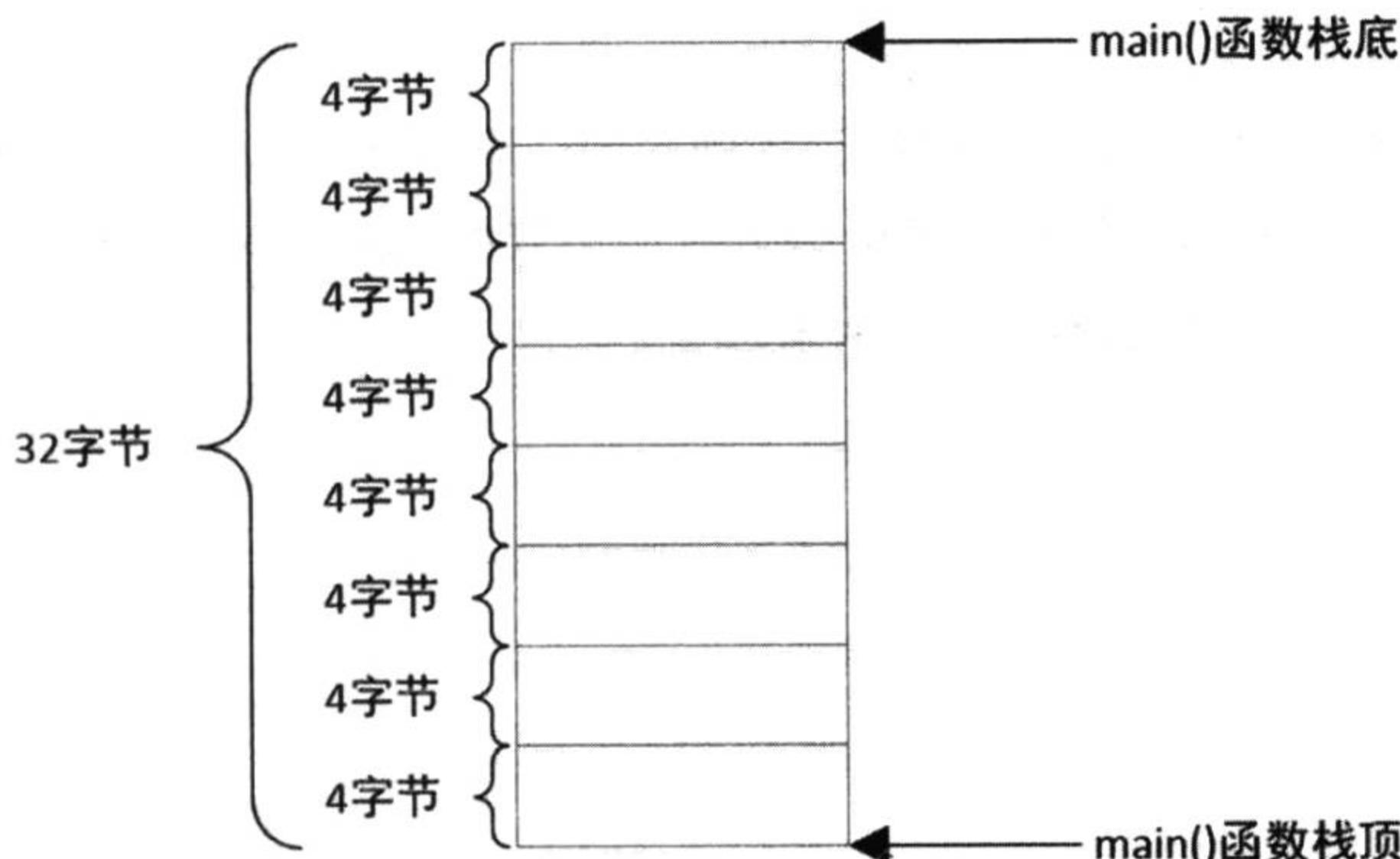


图 2.1 main()函数初始化时的堆栈

---

注：由于 main()的方法栈空间刚刚分配，还没有往里面加任何东西，因此此时的栈是空的，什么都没有。同时，系统为 main()函数一共分配了 32 个字节，在 64 位平台上，一个 int 型数据占 4 个字节，因此这里将 main()方法栈以 4 字节为单位进行划分，一共划分成 8 块，每一块代表 4 个字节大小的空间。

---

main()函数执行完上面这 3 条指令，便完成了调用者栈基地址的保存和自身栈空间的分配。

## 2 ) 初始化数据

main()函数接下来执行下面 2 条指令：

```
movl $5, 20(%esp)  
      movl $3, 24(%esp)
```

这两条指令的含义是：分别将 5 和 3 这两个整数保存到 main()栈中，其中 20(%esp)表示当前栈顶(即 esp 寄存器当前所指向的内存地址)往上移动 20 字节位置，数据 5 就被保存在这里。由于 main()函数的栈空间一共有 32 字节那么大，因此从 main()方法栈的栈顶往上移动 20 字节后的位置，依然在 main()的方法栈内。同理，整数 3 被保存到了 main()函数栈顶往上偏移 24 字节处的位置。由于一个整数占用 4 字节(在 64 位平台上，本书中所有示例均在 64 位平台上测试，因此本文所有的 int 型数据默认都占 4 字节，下文不再赘述)，因此 5 和 3 被分别保存到 main()方法栈顶往上偏移 5 个整数和 6 个整数的位置。

语言描述让人理解起来总是很费电，还是图来得直观些。下面我们就通过图示来描述偏移位置。

在使用图示来描述内存位置之前，我们先研究一下真实的物理机器上是如何进行内存定位的。

如果我们将 main()函数的栈顶位置标记为(%esp)，整个 main()方法栈空间的 32 字节，按照每 4 字节为单元进行划分，同时按照其相对于栈顶位置的偏移量来标记 main()的方法栈，那么 main()函数的方法栈内存可以如图 2.2 所示来标记。

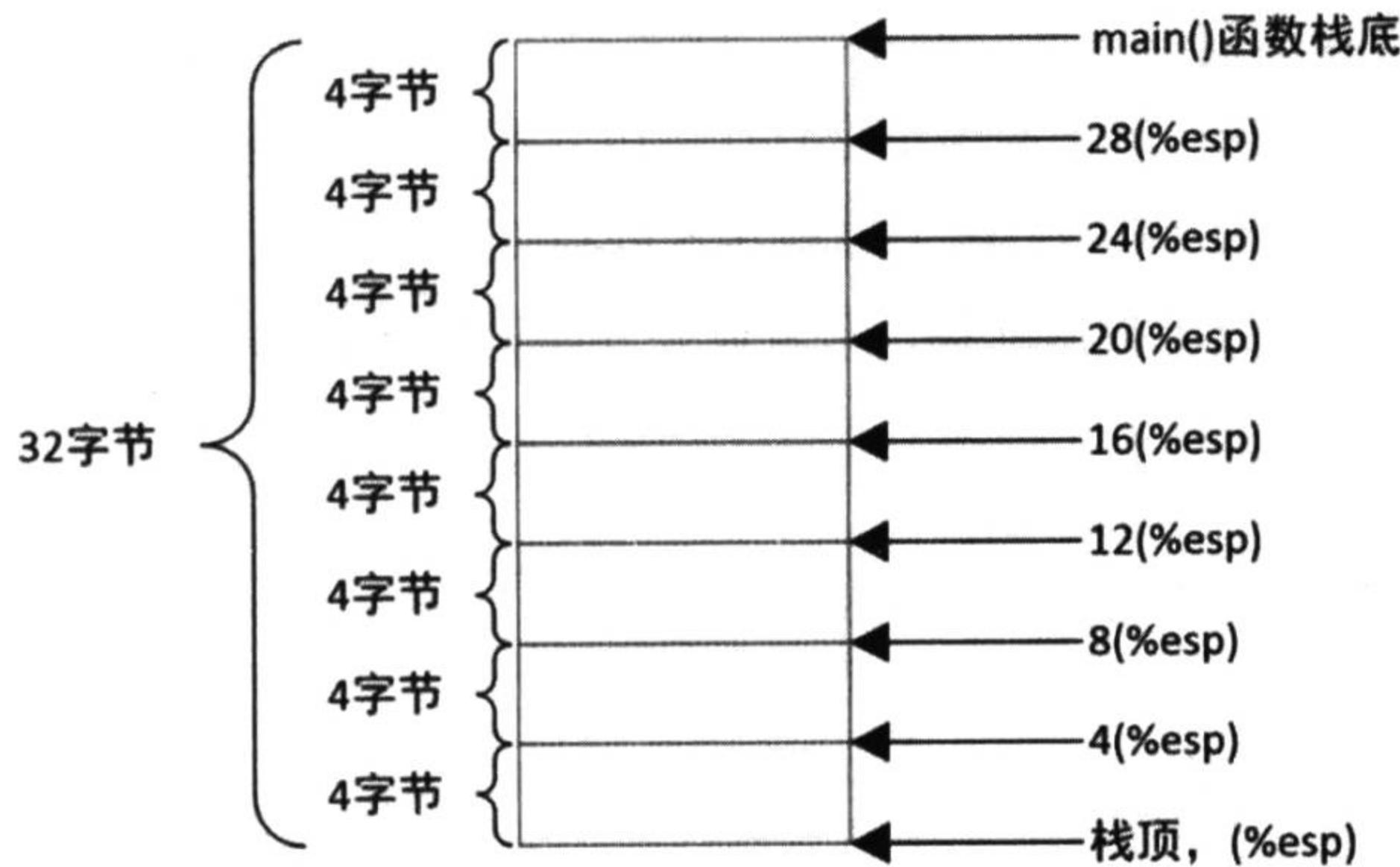


图 2.2 main()函数堆栈存储单元的相对位置

这种将方法栈内存位置按照栈顶偏移量进行标记的方法，是整个计算机的理论基础，直接影响了编程语言的模块划分方法。如果没有这种方法，编译器很难在编译期就确定各个变量的位置，更不用谈各种基于编译原理的高级应用了。

其实这种标记方法在现实世界中也是很常见的。对于地球上任意一个点，人类可以使用两种方式标记其位置，一种是绝对定位，一种是相对定位。例如以某个学校为例，我们可以说这个学校在经度  $30^{\circ}$  和纬度  $60^{\circ}$ ，也可以说这个学校在前方红绿灯右转 300 米。物理机器对方法内部的局部变量的定位就类似于对学校的“红绿灯右转 300 米”的相对定位法。

理解了相对定位方法后，我们再来看看 5 和 3 这两个数据在 main() 方法栈中的位置，如图 2.3 所示。

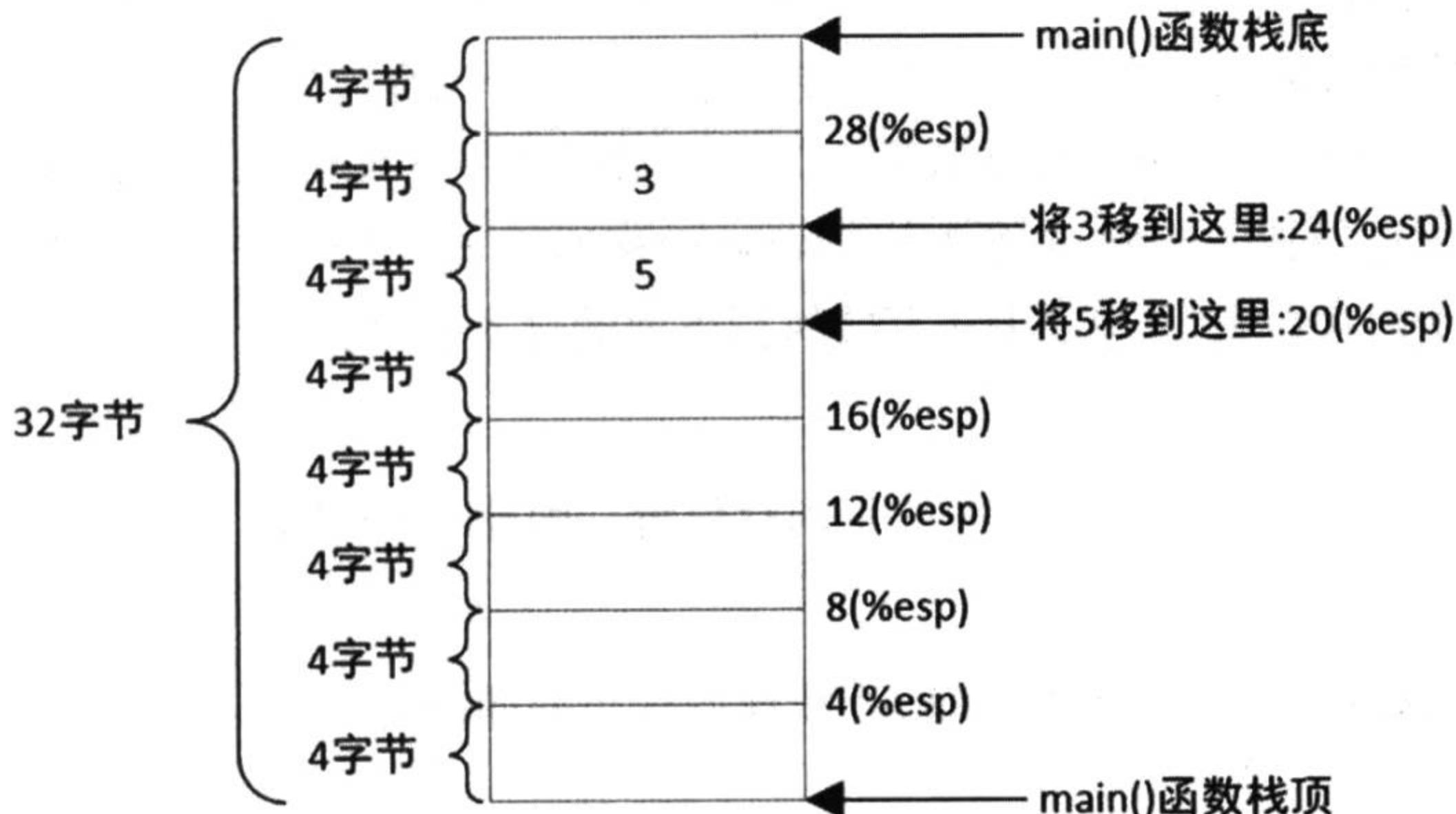


图 2.3 完成局部变量初始化的 main()方法堆栈布局

也许有些完美主义者看出点儿问题来了，即 5 和 3 为什么被分配在中间的位置，上边和下边哪都不挨，这是为什么呢？这里面是不是有什么讲究？讲究还真有，栈底那个位置即 28(%esp)是留给调用 add() 函数的返回值的，这个待会儿会讲到。

现在，main() 函数完成了数据准备，接着便要开始调用 add() 函数进行求和了。

### 3 ) 压栈

接着，main() 函数开始执行下面 4 条指令：

```
movl 24(%esp), %eax
movl %eax, 4(%esp)
movl 20(%esp), %eax
movl %eax, (%esp)
```

这 4 条指令主要作用是“压栈”。前两条指令是把数据 3 压栈，后两条指令是把数据 5 压栈。

先看 movl 24(%esp), %eax 这条指令，该指令将 24(%esp) 处的内存值传送到 eax 寄存器中。24(%esp) 处的内存值是什么呢？就是刚才第 2 步中保存的整数 3。接着 movl %eax, 4(%esp) 这条指令又将 eax 寄存器中的值传送到了 4(%esp) 这个地方。如果将这两条指令连着看，你会发现这里进行的数据传送的路径是：x->y, y->z，使用小学数学进行推理，可以推出：x->z，即 x 处的内存值被传送到了 z 处。因此，最终的效果是：整数 3 被从 24(%esp) 这个相对于栈顶的偏移位置，转移到了 4(%esp) 这个偏移位置。

同样的道理，后面两条指令的最终效果是：整数 5 被从 20(%esp) 这个相对于栈顶的偏移位置，转移到了(%esp) 这个偏移位置。（%esp）是哪个位置？请相信你的第一直觉，就是栈顶。

也许好奇心重的同学会想，既然 CPU 可以将一个数据从 x 点移到 y 点，再从 y 点移到 z 点，那么为什么不直接从 x 点移到 z 点呢？这真是个好问题。梦想是美好的，但是现实是残酷的，因为 CPU 不支持将数据从一个内存位置直接传送到另一个内存位置，若要想实现这个效果，必须使用寄存器进行中转。这里以移动数据 3 为例，数据 3 最终被从 24(%esp) 这个内存位置移到了 4(%esp) 这个内存位置，CPU 先将 3 从 24(%esp) 移到了 eax 寄存器，再将 3 从 eax 寄存器移到了 4(%esp) 这个内存位置。CPU 无法直接执行下面的这条指令：

```
movl 24(%esp), 4(%esp)
```

只因为 24(%esp) 和 4(%esp) 都代表的是内存位置，因此 CPU 无法直接完成内存之间的数据传送。

这 4 条指令执行下来，main() 函数的方法栈内存布局如图 2.4 所示。

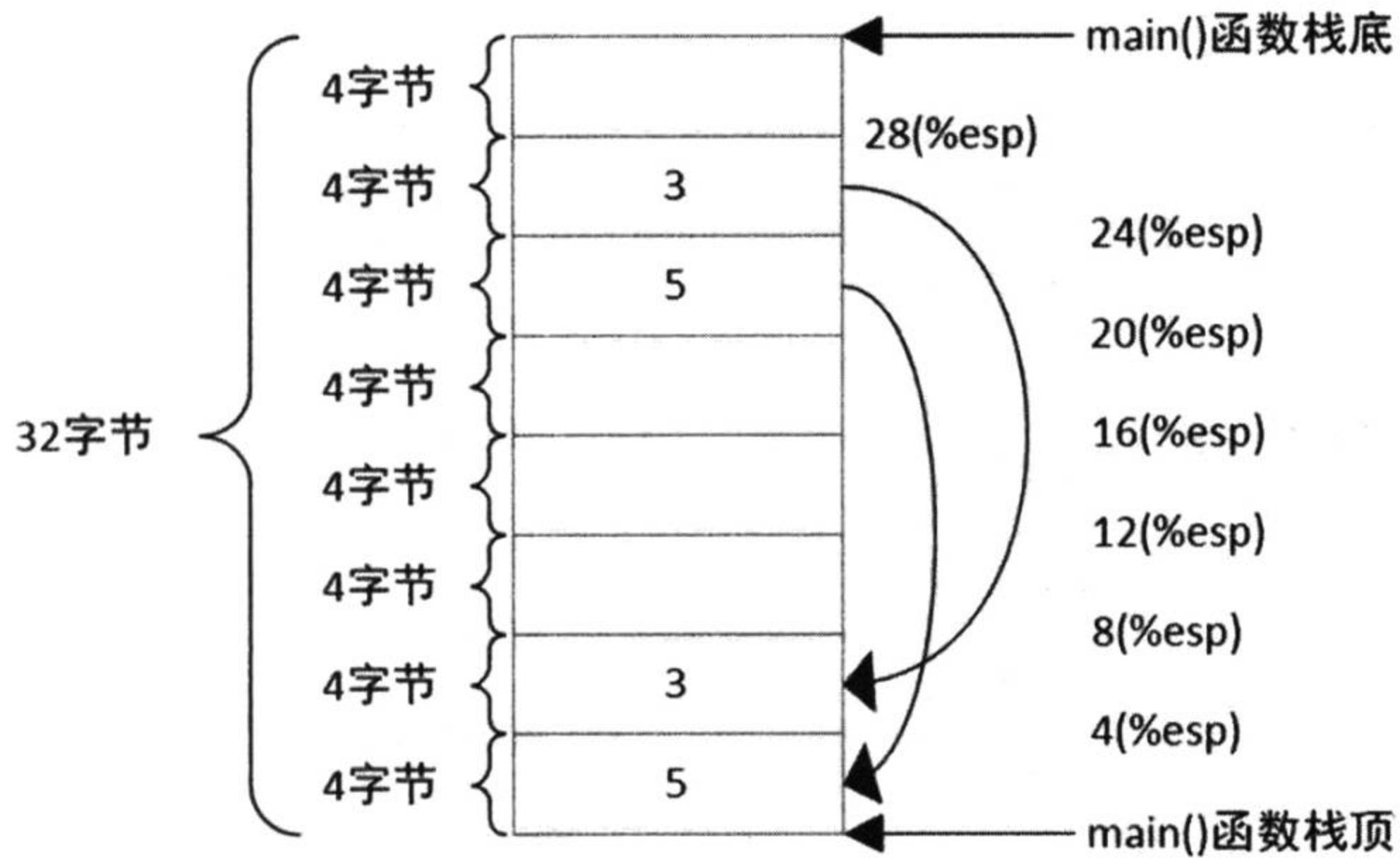


图 2.4 完成 add()参数复制后的 main()函数堆栈布局

一般而言，往栈顶传送数据的行为叫做“压栈”，这里先后将 3 和 5 都放到了栈顶处，因此这都是压栈。

main()函数为什么要压栈呢？那是因为 main()函数即将要进行函数调用了。

真实的物理机器，在发起函数调用之前，必定要进行压栈。压栈的目的是为了传参。

main()函数在这里压栈的两个数据，将会被 add()函数读取到。

#### 4 ) 函数调用

压完了栈，main()函数终于开始进行函数调用了。对于物理机器而言，函数调用特别简单，就一条指令：

```
call    add
```

但是，看着简单，背后却有一套机制在支撑。这个下面会讲。

add()函数执行完，会将计算的结果保存到 eax 寄存器中。main()函数要取得 add()函数返回值，便直接从 eax 寄存器中拿即可。因此，执行完 call add 函数调用指令后，main()函数接着调用下面的指令：

```
movl %eax, 28(%esp)
```

通过这条指令，main()函数终于完成了求和计算，并拿到了计算结果。此时，main()函数的方法栈内存布局如图 2.5 所示。

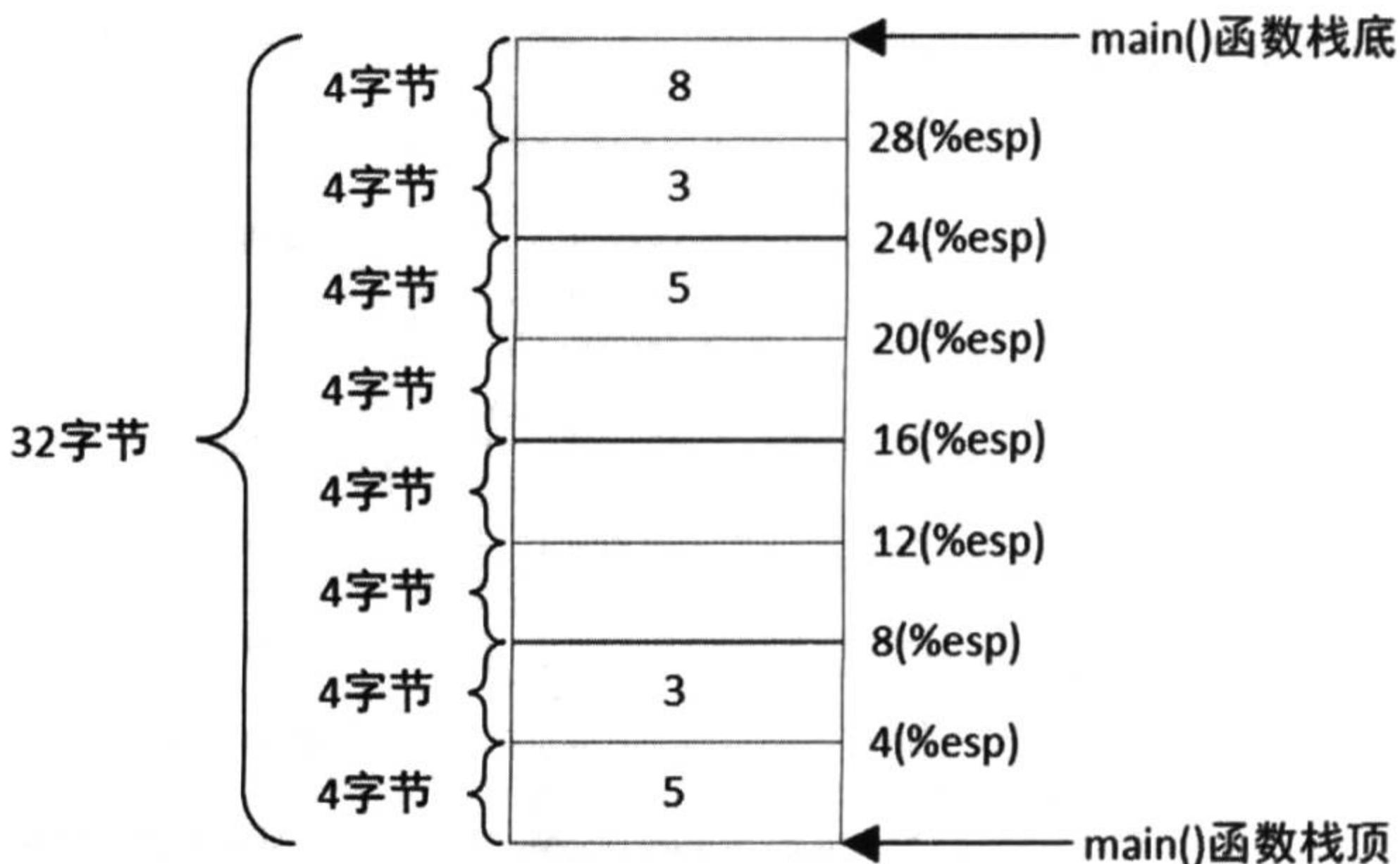


图 2.5 完成 add()调用后的 main()函数堆栈布局

到了这里，你会发现其实这样的内存布局还是挺美的，什么叫美？历史上无论是科学大牛还是艺术巨匠都告诉我们：对称的，才是美的。你看现在的 main()方法栈的内部布局图，就挺对称，上面的数据都挨着栈底，下面的数据都挨着栈顶。之所以会这样，全是编译器的功劳。编译器会将一个方法内的局部变量分配在靠近栈底的位置，而将传递的参数分配在靠近栈顶的位置。

### 5) 返回

函数返回很简单，将返回值保存到 eax 寄存器中，然后执行两条例行返回指令便大功告成。main()函数的返回指令就是下面这 3 条：

```
movl $0, %eax
leave
ret
```

好了，到此为止，我们完整地分析了 main()函数的执行过程以及栈内存的布局演变。在分析的过程中，顺带着了解了 main()函数为了调用 add()函数而做的准备。在本示例中，main()函数为了调用 add()函数，主要是将入参进行了“压栈”操作，这样在 add()函数内部才能取到参数并进行求和运算。

但是，除了压栈外，其实系统还要做一部分工作，才能完成最终的方法调用。下面我们通过分析 add()函数的执行过程，来了解系统工作的机制。

## 2. add()函数详解

如果你对上面 main()函数的执行原理已经比较熟悉，那么理解 add()函数的运行机制就容易

了。在分析 add() 函数之前，我们先看下 add() 函数的整体逻辑：

**清单：示例程序**

**作用：**使用 C 程序提供的语法糖修改 CS:IP 段寄存器的指向

```
add:
    // 保存调用者栈基地址
    pushl %ebp
    movl %esp, %ebp
    subl $16, %esp

    // 获取入参
    movl 12(%ebp), %eax
    movl 8(%ebp), %edx

    // 执行运算
    addl %edx, %eax
    movl %eax, -4(%ebp)

    // 返回
    movl -4(%ebp), %eax
    leave
    ret
```

可以看到，add() 函数总体上分为 4 步：保存调用者栈基地址，读取入参，执行运算，返回。下面我们逐一分析过程。

### 1) 保存调用者栈基地址

add() 函数也是以下面两条指令开始：

```
pushl %ebp
movl %esp, %ebp
```

这一步大家应该很熟悉，刚才在分析 main() 函数时就已经讲过。这两条指令主要是保存调用者栈基地址。这里再啰嗦一次，物理机器在执行函数调用时，被调用者总是要保存调用者栈基地址。这是因为 esp 和 ebp 这两个寄存器接下来要指向被调用者的栈基地址和栈顶，这两个寄存器原本保存的是调用者的栈基地址和栈顶地址，现在即将被修改，如果不保存起来，那么当被调用者函数执行完成，程序流返回到调用者流程中时，物理机器将无法恢复调用者的栈基和栈顶，从而导致程序无法继续执行下去。

add() 函数第 3 条指令是：

```
subl $16, %esp
```

这条指令大家应该很熟悉，就是分配栈空间。为谁分配？当然是为当前函数 add()。分配多大空间？这条指令中的 16 来指定。不过要注意，这里指 16 字节，而不是 16 个二进制位。