链滴

《深入理解 Java 虚拟机》读书笔记：虚拟机字节码执行引擎

作者：jingqueyimu

原文链接：https://ld246.com/article/1584287907478

来源网站：链滴

许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

正文

执行引擎是 Java 虚拟机最核心的组成部分之一。在不同的虚拟机实现里，执行引擎在执行 Java 代码
可能会有解释执行（通过解释器执行）和编译执行（通过即时编译器产生本地代码执行）两种选择，
可能两者兼备。但从外观上看，所有 Java 虚拟机的执行引擎都是一致的：输入的是字节码文件，处
过程是字节码解析的等效过程，输出的是执行结果。

物理机与虚拟机的执行引擎：

● 物理机的执行引擎：直接建立在处理器、硬件、指令集和操作系统层面上。

● 虚拟机的执行引擎：由自己实现，可自行制定指令集与执行引擎的体系结构，能够执行那些不被硬
直接支持的指令集格式。

一、运行时栈帧结构

栈帧（Stack Frame）是用于支持虚拟机进行方法调用和方法执行的数据结构，它是虚拟机运行时数
区中虚拟机栈的栈元素。栈帧存储了方法的局部变量表、操作数栈、动态连接、方法返回地址和一些
外的附加信息。每一个方法从调用开始至执行完成的过程，都对应着一个栈帧在虚拟机里面从入栈到
栈的过程。

对于执行引擎来说，在活动线程中，只有位于栈顶的栈帧才是有效的，称为当前栈帧，与这个栈帧相
联的方法称为当前方法。执行引擎运行的所有字节码指令都只针对当前栈帧进行操作。

1、局部变量表

局部变量表（Local Variable Table）是一组变量值存储空间，用于存放方法参数和方法内部定义的
部变量。

局部变量表的容量以变量槽（Variable Slot，下称 Slot）为最小单位，虚拟机规范中并没有明确指明
个 Slot 应占用的内存空间大小。为了尽可能节省栈帧空间，局部变量表中的 Slot 是可以重用的。

虚拟机通过索引定位的方式使用局部变量表，索引值的范围是从 0 开始至局部变量表最大的 Slot 数
。

在方法执行时，虚拟机通过局部变量表完成参数值到参数变量列表的传递。如果执行的是实例方法（非
static 方法），那局部变量表中第 0 位索引的 Slot 默认用于传递方法所属对象实例的引用，在方法
可通过关键字“this”访问到这个隐含的参数。其余参数则按照参数表顺序排列，参数表分配完毕后
再根据方法体内部定义的变量顺序和作用域分配其余的 Slot。

2、操作数栈

操作数栈（Operand Stack）也称为操作栈，它是一个后入先出的栈。操作数栈的每一个元素可以是
意的 Java 数据类型。

当一个方法刚开始执行时，这个方法的操作数栈是空的，在方法执行过程中，会有各种字节码指令往
作数栈中写入和提取内容，这就是出栈/入栈操作。

在概念模型中，两个栈帧作为虚拟机栈的元素，是完全独立的。但在大多数虚拟机的实现里会做一些
化处理，令两个栈帧出现一部分重叠。这样在进行方法调用时就可以共用一部分数据，无须进行额外
参数复制传递。

3、动态连接

每个栈帧都包含一个指向运行时常量池中，该栈帧所属方法的引用，持有这个引用是为了支持方法调过程中的动态连接（Dynamic Linking）。

- 静态解析：在类加载阶段或第一次使用时，将符号引用转化为直接引用。
- 动态连接：在每一次运行期间，将符号引用转化为直接引用。

4、方法返回地址

两种退出方法的方式：

- **正常完成出口：** 执行引擎遇到任意一个方法返回的字节码指令，此时可能会有返回值传递给上层的法调用者。
- **异常完成出口：** 方法执行过程中遇到了异常，并且这个异常没有在方法体内得到处理，这种退出方不会有返回值。。

无论何种退出方式，方法退出后都需要返回到方法被调用的位置，程序才能继续执行，方法返回时可需要在栈帧中保存一些信息，用来帮助恢复它的上层方法的执行状态。

一般来说，方法正常退出时，调用者的 PC 计数器的值可以作为返回地址，栈帧中很可能会保存这个数器值。而方法异常退出时，返回地址要通过异常处理器表来确定，栈帧中一般不会保存这部分信息。

方法退出的过程实际上等同于把当前栈帧出栈，因此退出时可能执行的操作有：恢复上层方法的局部量表和操作数栈，把返回值（有的话）压入调用者栈帧的操作数栈中，调整 PC 计数器的值以指向方调用指令后面的一条指令等。

5、附加信息

虚拟机规范允许具体的虚拟机实现增加一些规范里没有描述的信息到栈帧中，例如与调试相关的信息。

二、方法调用

方法调用并不等于方法执行，方法调用阶段唯一的任务就是确定被调用方法的版本（即调用哪一个方），暂时还不涉及方法内部的具体运行过程。

一切方法调用在 Class 文件里存储的只是符号引用，而不是直接引用，只有在类加载期间，甚至是运期间才能确定目标方法的直接引用。

方法调用字节码指令：

- invokestatic：调用静态方法。
- invokespecial：调用实例构造器 init() 方法、私有方法、父类方法。
- invokevirtual：调用所有虚方法。
- invokeinterface：调用接口方法，会在运行时再确定一个实现此接口的对象。
- invokedynamic：先在运行时动态解析出调用点限定符所引用的方法，然后再执行该方法。

1、解析

在类加载的解析阶段，将方法的符号引用转化为直接引用，这类方法调用称为解析。这种解析能成立前提是：方法在程序执行之前有一个可确定的调用版本，并且这个方法的调用版本在运行期不可改变即“编译期可知，运行期不可变”。

只要能被 invokestatic 和 invokespecial 指令调用的方法，都可以在解析阶段确定唯一的调用版本，此都能在类加载阶段被解析。这些方法称为非虚方法，与之相反，其他方法称为虚方法。

final 方法虽然是使用 invokevirtual 指令调用的，但由于它无法被覆盖，没有其他版本，所以无须对法接收者进行多态选择。因此，fanal 方法也属于非虚方法。

2、分派

（1）静态分派

依赖静态类型（又称外观类型）来定位方法执行版本的分派动作，称为静态分派。静态分派的典型应是方法重载。

静态类型是编译期可知的。

静态分派发生在编译阶段，因此确定静态分派的动作不是由虚拟机来执行的。

（2）动态分派

在运行期根据实际类型确定方法执行版本的分派过程，称为动态分派。动态分派的典型应用是方法重。

实际类型是在运行期才可确定。

动态分派是非常频繁的动作，而且运行时需要在类的方法元数据中搜索合适的目标方法。基于性能的虑，大部分的虚拟机实现都不会真正地进行如此频繁的搜索。最常用的“稳定优化”手段是为类在方区中建立一个虚方法表，使用虚方法表索引来代替元数据查找以提高性能。

虚方法表中存放着各个方法的实际入口地址。

（3）单分派与多分派

根据分派基于多少种宗量，可以将分派划分为单分派和多分派。单分派是根据一个宗量对目标方法进选择，多分派则是根据多于一个宗量对目标方法进行选择。

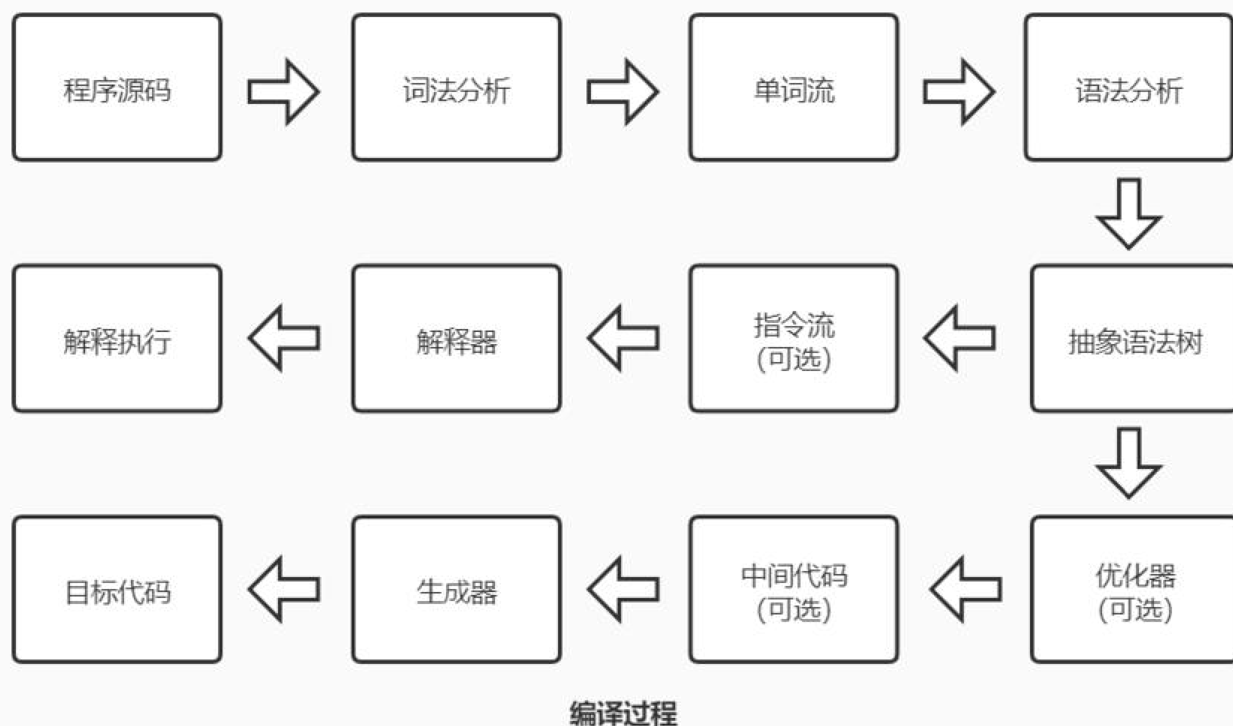
方法的接收者与方法的参数统称为方法的宗量。

静态分派是根据方法接收者的静态类型和方法参数来选择目标方法的，因此静态分派属于多分派类型。

动态分派只根据方法接收者的实例类型来选择目标方法，因此动态分派属于单分派类型。

三、基于栈的字节码解释执行引擎

1、编译过程



如今，基于物理机、虚拟机的语言，大多都会遵循基于现代经典编译原理的思路，在执行前先对程序码进行词法分析和语法分析处理，把源码转化为抽象语法树。

对于一门具体语言的实现来说，词法分析、语法分析以及后面的优化器和目标代码生成器都可以选择立于执行引擎，形成一个完整意义的编译器去实现，这类代表是 C/C++ 语言。也可以选择把其中的部分（如生成抽象语法树之前的步骤）实现为一个半独立的编译器，这类代表是 Java 语言。又或者这些步骤和执行引擎全部集中封装在一个封闭的黑匣子之中，如大多数的 JavaScript 执行器。

Java 语言中，Javac 编译器完成了程序代码经过词法分析、语法分析到抽象语法树，再遍历语法树生线性的字节码指令流的过程。因为这一部分动作是在 Java 虚拟机之外进行的，而解释器在虚拟机内，所以 Java 程序的编译就是半独立的实现。

2、解释执行

Java 语言经常被定位为“解释执行”的语言，在 Java 初生的 JDK1.0 时代，这种定义还算准确，但主流的虚拟机中包含了即时编译器后，Class 文件中的代码到底会被解释执行还是编译执行，就成了有虚拟机自己才能准确判断的事情。

3、基于栈的指令集与基于寄存器的指令集

Java 编译器输出的指令流，基本上是一种基于栈的指令集架构，指令流中的指令大部分是零地址指令它们依赖操作数栈进行工作。与之相对的另外一套常用的指令集架构是基于寄存器的指令集，最典型就是 x86 的二地址指令集，这些指令依赖寄存器进行工作。

- 基于栈的指令集：可移植，但执行速度相对较慢。
- 基于寄存器的指令集：执行速度快，但由于寄存器由硬件直接提供，程序不可避免要受到硬件的约。