Universe Scientific Publishing
Computer System Networking and Telecommunications
计算机系统网络和电信，2019,1(2)
ISSN:2661-3719 (Print); 2661-3727 (Online)

# Implementation Principle of Cross-platform Core Module of Java Virtual Machine

Shijian FENG　Zhengde BAO　Yawen TANG
School of Computer and Software, Jincheng College, Sichuan University, Chengdu, Sichuan, 611731

**Abstract**

JAVA has become popular in the field of programming because of its portability. Using the bytecode loader in JVM, one compilation can be realized and run everywhere. This paper mainly expounds the bytecode loading engine and its implementation principle in JVM, and briefly analyses how to transform high-level language into machine language which can be recognized by machine through a series of processing, and realize cross-platform operation.

**Key Words**

Java Virtual Machine, Bytecode, Dynamic Allocation, Stack frame

**DOI:10.18686/jsjxt.v1i2.648**

# Java 虚拟机跨平台核心模块的实现原理

冯世建　鲍正德　唐娅雯
四川大学锦城学院计算机与软件学院，四川，成都，611731

## 摘　要

**JAVA 因其可移植性在编程领域热度不减。利用 JVM 中的字节码加载器可以实现一处编译，到处运行。本文主要阐述了 JVM 中的字节码加载引擎及实现的原理，浅析了如何将高级语言经过一系列的处理转化为机器可以识别的机器语言，并实现跨平台运行。**

## 关键字

**JAVA 虚拟机；字节码；动态分派；栈帧**

## 1.引言

JAVA 可以跨平台是因为 JVM 的可移植性，JVM 起到一个连接 OS 和 JAVA 源码的作用。JAVA 虚拟机可以帮助程序员完成许多难以完成的事，所以被广大程序员使用，但却很少有人了解其内部具体的实现原理。从原理的角度对虚拟机进行剖析，确定了我们优化的方向：栈帧的复用和修改基于栈的指令集都可以达到优化虚拟机的目的。虚拟机中的栈有着独有的特殊结构，这极大的提高了静态解析和分派的效率，为后面的解释执行做了铺垫。

## 2.JVM 字节码执行引擎运行时的栈帧结构

线程中存在的栈，在虚拟机中被叫做虚拟机栈，Java 虚拟机栈一般存在于通用 RAM 中，程序指令在进行操作时，需要由出入栈配合来完成[1]，虚拟机栈是由一个个栈帧组成的。在虚拟机堆栈中，虚拟机栈被分配到每一个线程中，栈中包含一定的栈帧。每个栈帧都包含方法，动态连接，操作数栈，局部变量表和方法返回地址信息等。在活动的线程中，每次调用方法是都伴随着栈帧的入栈与出栈，栈帧是从栈顶开始出栈的，出栈栈帧被叫做有效栈帧。同时值得注意的是栈中的栈帧有数量限制，所以尽量避免过多的方法调用导致内存的溢出。

### 2.1 栈帧的数据结构

在栈帧中，变量槽是最小的单位，包含 32 个字节。在 32 位的虚拟机中，一个变量槽代表 32 位，同理在

64 位的虚拟机中两个变量槽代表 64 位。如果在 64 位的虚拟机中模拟 32 位的填充，可以通过补码。在代码编译的过程中，栈帧所需要的大小就已经被计算出来了。所以，在运行的时候，栈帧所需的内存大小不会被其他变量所影响。

### 2.2 局部变量表

栈帧中有局部变量表，局部变量表储存了已经写好的方法和这个方法所需要的参数信息。这个表的基本单位是上文提到的变量槽。虚拟机查询这个表需要用到索引。在 32 位的变量中，通过索引 k 可以查找到第 k 个变量槽；而在 64 位的变量中第 k 个索引占用了 k 和 k+1 这两个变量槽。

在 64 位的虚拟机中，两个变量槽是同时被访问的，不可拆分，如果遇到了单独访问的情况，需要抛出一个异常。在栈帧中，重用变量槽可以带来性能的提升，但这可能会导致 GC 没有及时回收垃圾。做一个实验，如果我们在一个类中执行栈帧中的对象，并在对象后面调用 GC，这时候并没有触发 GC 的回收，这是因为这个对象处于被使用的状态，生命周期还没有结束：

```
byte[] dataType = new byte[24*1024];
System.gc();
```

但是如果我们把这个对象放在普通代码块中，并在这个代码块执行完毕之后调用 GC，这个对象的生命周期已然结束，可是依旧没有引发 GC 的回收。

```
{byte[] dataType = new byte[24*1024];}
System.gc();
```

而以下代码会引发 GC 的回收：

```
{byte[] dataType = new byte[24*1024];}
int temp = 0;
System.gc();
```

通过这个实验证明：只有当使用公用变量槽的方法时置空不用的对象，才会引发 GC 的回收。这里使用到了 Java 高效编程的思想，随时置空不使用的对象。

### 2.3 操作数栈

操作数栈中的元素是后入先出的。他和上文的局部变量表一样在编译期间就确定了大小，其中可以存放 JAVA 中的所有类型的数据。当栈帧中的方法被执行时，方法中的操作栈会伴随一系列的出栈和入栈。可使用字节码指令可以将内容压入到这个操作栈中，这是 pop / push 操作。操作栈可以运行一般的计算方程式，当调用计算的方法时，算术符号被当做参数压栈。例如，执行一个简单的加法运算，通过 iadd 指令可以将操作栈中最后进来的元素依次取出，并进行相加。执行此指令时，将弹出并添加两个数值，并将添加的结构压栈。值得注意的是，指令应在适当的范围内使用，iadd 指令用于整数加法，其他类型的数据类型相加的话就会抛出异常。有序虚拟机栈中的栈帧是互不相干的独立个体，所以我们可以想到通过栈帧的重叠来优化数据结构，得到数据的重用。

### 2.4 动态链接

通过把程序分离，可以达到更好的空间利用的目的，同时栈帧中的元素也可以被动态的更改了。字节码中，存在许多符号引用，这些符号引用被当做参数被方法调用指令所调用，当含有这个符号引用的时候便可以实现动态链接动态的调用这个引用。在类开始加载的时候（解析）或首次使用时，符号引用会直接转换为直接引用，这个过程就叫做静态链接。如果一些符号引用在运行期被转换为直接引用，就被叫做动态链接。所以虚拟机可以直接将这些符号引用链接起来运行，而不用去寻找一个个文件，等程序运行起来的时候可以动态的将他们链接在一起，就达到了程序模块分离的效果。

## 3.JVM 字节码执行引擎运行时的方法调用

### 3.1 解析

#### 3.1.1 解析调用

通过一个 Class 文件我们可以得到程序中的所有类，接口和方法的信息[2]。在常量池中，符号引用还包括调用的方法所指的目标方法。也就是说，方法会在编译阶段被明确调用，这个方法的调用就是解析调用，这是一个非动态的过程。

#### 3.1.2 适合在编译期解析的方法

JAVA 中的静态方法和私有方法在运行期间是不能改变的，因此他们可以在编译期间被确定。私有方法不能在类的外部被访问，静态方法与类型直接关联，所以不可以重写，所以它们往往在类开始加载的时候被解析。方法的调用主要是为了确定在运行期调用哪个方法，并不会真正的运行，所以这个过程是静态的。这种特殊的

链接方式极大地提高了 Java 的可变性和可扩展性，但却增加了复杂度。

### 3.1.3 字节码的调用方法

在字节码中，常量池中调用非虚方法的方式主要有两种：调用静态方法 invokestatic（静态绑定，速度快），调用虚方法 invokevirtual（实例方法）[3]。下面我们将在静态解析中演示非虚方法的调用：

```
public class Calling{
public static void callingMethod(){
System.out.println(“This method has been called”);}
public static void main(String[] args){
Calling.callingMethod();}}
```

运行这段代码之后，我们在目录中找到生成的 class 文件，使用 javap 观察其中的内容：public static void callingMethod();

```
descriptor: ()V
Code:
0: getstatic #2 // Fieldjava/lang/System.out:Ljava/io/PrintStream;
3: ldc #3 // String This method has been called
5: invokevirtual #4 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
8: return
```

经过实验证明 invokestatic 确实调用了 callingMethod 方法。

### 3.2 JVM 实现分派

除了静态的解析，分派是在另外一个层次的目标方法筛选。由于 JAVA 中特殊的动态分配，提出了虚方法表的方法对字节码中的方法就行高效率的分配。JVM 分派的最终目的是确定执行时方法的执行顺序，这也体现了 JAVA 的多态性。

### 3.2.1 分派的原理

静态过程叫做解析，而分派则可能是一个动态的过程也可能是一个静态的过程。值得注意的是：静态本机编译器不支持 Java 的动态加载功能，而动态本机编译器支持 Java[4]。因此，在不影响 JVM 移植性的条件下，JAVA 在运行时才进行对本地代码的编译，而不是在运行之前进行编译。

### 3.2.2JAVA 的动态编译

java 的动态分配可以很好的处理方法的调用，考虑到使用的频繁，在这里可以想到对虚拟机性能的一些优化。如果每一次的动态分配都去搜索元数据的话会影响代码的执行效率。所以，JVM 在类的方法区域中构建虚拟方法表以提高性能。每个类都有一个用于保存每个方法的实际条目的虚方法表，这大大减少了搜索的次数，但同时也会牺牲一点点空间。考虑到复用的问题，当我们写了一个子类重写了虚方法表中的这个方法的话，得到的结果是这个方法的地址原来父类的地址是相同的。这就证明了，子类的方法指向了原来父类中的方法。这便是重写在 JAVA 中的一个体现。

### 3.2.3 动态编译的实验

用一个实验来证明：准备两个静态类型相同的类 A，B 并继承同一个父类 C，并重写 C 的同一个方法

```
class Assignment{
static abstract class C {
protected abstract void printMethod();}
static class A extends C {
protected void printMethod(){System.out.print("A");}}
static class B extends C {
protected void printMethod(){System.out.print("B");}}
public static void main(String[] args){
C a = new A();
C b = new B();
a.printMethod();
b.printMethod();
a= new B();
a.printMethod();}
}
```

得到的结果是：ABB。可以得知调用重新方法是两个类得到了不同的结果，A 在第一次调用和第二次调用中执行的方法不同。为了得到其中的原因，我们将 A，B 放入变量槽中并压到栈顶，再使用方法调用的指令调用他们，发现这两个类的指令和参数均相同，但最终执行的方法却不同。让我们用 javap 反编译看看方法在字

节码中的调用情况：

```
E:\java\develop>javap -c -s Assignment.class
Compiled from "test.java"
class com.test.Assignment {
  com.test.Assignment();
    descriptor: ()V
    Code:
      0: aload_0
      1: invokespecial #1                  // Method
java/lang/Object."<init>":()V
      4: return
  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
    Code:
      0: new           #2                  // class
com/test/Assignment$A
      3: dup
      4: invokespecial #3                  // Method
com/test/Assignment$A."<init>":()V
      7: astore_1
      8: new           #4                  // class
com/test/Assignment$B
     11: dup
     12: invokespecial #5                  // Method
com/test/Assignment$B."<init>":()V
     15: astore_2
     16: aload_1
     17: invokevirtual #6                  // Method
com/test/Assignment$C.printMethod:()V
     20: aload_2
     21: invokevirtual #6                  //
Method com/test/Assignment$C.printMethod:()V
     24:    new                           #4
// class com/test/Assignment$B
     27: dup
     28: invokespecial #5                  //
Method com/test/Assignment$B."<init>":()V
     31: astore_1
     32: aload_1
     33: invokevirtual #6                  //
Method com/test/Assignment$C.printMethod:()V
     36: return
}
```

出现这个结果的原因是由于 A 和 B 的实际类型在运行期开始的时候就被虚拟机 invokevirtual 指令接收了，导致第一次调用的符号引用和第二次调用的符号引用被解析到了两个不同的直接引用上。因此，动态分派在 main 函数运行的时候根据 A，B 的实际类型确定了 A，B 应该执行的方法。

## 4.基于栈的字节码解释执行引擎

### 4.1 解释执行和编译执行

在 JDK1.0 发布的时候，初代的虚拟机是通过解释执行的。随着虚拟机的发展与壮大，出现了解释执行和编译执行共存的高性能虚拟机。例如，HotSpot 的执行引擎采用解释器和即时编译器共存的架构，对于一般的代码采用解释器每次读取字节码指令，将指令解释成本地代码并予以执行[5]。它们的基本实现原理大致相同，通过对语言的词法分析和语法分析，生成一个抽象的语法树。

### 4.2 基于栈的指令集与基于寄存器的指令集

JVM 中解释执行引擎使用的是栈的指令集，一般寄存器很难被直接使用，但是 JVM 可以帮助我们将我们经常使用到的数据直接放进其中，极大地提升了性能。但不可避免的是执行速度会变慢。当一个数据频繁的入栈和出栈就会影响其执行的性能，因为这其中伴随着更多的字节码指令的生成。而且这个过程是内存中进行的，内存的负担加重，效率就相应的变慢了。还有另外一种方式是使用寄存器中的指令集，这种方式的效率相对较高 ，但是太过依赖平台，降低了可移植性。

### 4.3 基于栈的解释器执行的优化

JAVA 虚拟机的最终将优化基于栈的解释器实现以提高性能，例如一些主流 JAVA 虚拟机： HopSpot，VM 等，其中的解析器和编译期会对字节码做一个优化处理，比如合并，替换输入的字节码等方式，以此提高解释执行性能。相信 JAVA 在未来内存技术的快速发展中能有更好的优势，以空间换取时间未尝不是一种对的选择。

## 5.结束语

JVM 字节码解释执行引擎可以将字节码转换为机

器指令从而达到跨平台运行的目的，其中对栈帧存储的优化和基于栈的指令集的操作是我们可以考虑对虚拟机优化的一个方向。JAVA 的编译过程中的方法调用与常量池中的符号引用有着密切的关系，动态分派的实现让 JAVA 摒弃了编译过程中的连接过程，这与 C++相比无疑是提升了很强的动态扩展能力。

## 参考文献

[1]王茂钢.Java 内存模型描述及变量运用分析[J].现代信息科技,2019,3(04):98-99.
[2]李阳. Java 的实时性研究[D].江南大学,2008.
[3]游锦鑫. 域敏感的 Java 程序副作用分析研究[D].江西师范大学,2015.
[4]冀振燕,程虎.Java 编译程序技术与 Java 性能(英文)[J].软件学报,2000(02):173-178.
[5]郭书超.HotSpot 虚拟机类加载及优化的原理与实现[J].山东工业技术,2014(21):112+126.

## 作者简介

第一作者：（1997-）冯世建，男，汉，四川省成都市，本科，四川大学锦城学院，研究方向：J2EE
第二作者（通讯作者）：鲍正德（1989-），男，汉，黑龙江哈尔滨，研究生，四川大学锦城学院，研究方向：电子商务。
第三作者：唐娅雯（1999-），女，汉，四川省资阳市，本科，四川大学锦城学院，研究方向：信息管理、J2EE