**很有意思~我希望大家可以好好看一下，虽然有点乱，但是好好花点时间我觉得你们可以学懂~并且可以学到很多东西~然后回味无穷。**

**字节码执行引擎：**

[JVM字节码执行引擎](https://www.cnblogs.com/zhouyuqin/p/5220728.html#wiz_toc_0)  
[**运行时栈帧结构**](https://www.cnblogs.com/zhouyuqin/p/5220728.html#wiz_toc_1)  
[局部变量表](https://www.cnblogs.com/zhouyuqin/p/5220728.html#wiz_toc_2)  
[操作数栈](https://www.cnblogs.com/zhouyuqin/p/5220728.html#wiz_toc_3)  
[动态连接](https://www.cnblogs.com/zhouyuqin/p/5220728.html#wiz_toc_4)  
[方法返回地址](https://www.cnblogs.com/zhouyuqin/p/5220728.html#wiz_toc_5)  
[**方法调用**](https://www.cnblogs.com/zhouyuqin/p/5220728.html#wiz_toc_7)  
[解析](https://www.cnblogs.com/zhouyuqin/p/5220728.html#wiz_toc_8)  
[分派 –“重载”和“重写”的实现](https://www.cnblogs.com/zhouyuqin/p/5220728.html#wiz_toc_9)  
[静态分派](https://www.cnblogs.com/zhouyuqin/p/5220728.html#wiz_toc_10)  
[动态分派](https://www.cnblogs.com/zhouyuqin/p/5220728.html#wiz_toc_11)  
[单分派和多分派](https://www.cnblogs.com/zhouyuqin/p/5220728.html#wiz_toc_12)  
[JVM动态分派的实现](https://www.cnblogs.com/zhouyuqin/p/5220728.html#wiz_toc_13)  
[**基于栈的字节码解释执行引擎**](https://www.cnblogs.com/zhouyuqin/p/5220728.html#wiz_toc_14)  
[基于栈的指令集与基于寄存器的指令集](https://www.cnblogs.com/zhouyuqin/p/5220728.html#wiz_toc_15)

我们都知道，在当前的Java中（1.0）之后，编译器讲源代码转成字节码，那么字节码如何被执行的呢？这就涉及到了JVM的字节码执行引擎，执行引擎负责具体的代码调用及执行过程。就目前而言，所有的执行引擎的基本一致：

1. 输入：字节码文件。
2. 处理：字节码解析。
3. 输出：执行结果。

物理机的执行引擎是由硬件实现的，和物理机的执行过程不同的是**虚拟机的执行引擎由于自己实现的。**

　在Java虚拟机规范中制定了虚拟机字节码执行引擎**(基于栈的指令集)**的概念模型，这个概念模型称为各种虚拟机执行引擎的统一外观。虚拟机实现中，可能会有两种的执行方式：**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）**。有些虚拟机值采用一种执行方式，但是有的采用了两种，甚至有可能包含几个不同级别的编译器执行引擎。

主要的执行技术有:解释，即时编译，自适应优化、芯片级直接执行其中解释属于第一代JVM，即时编译JIT属于第二代JVM，自适应优化（目前Sun的HotspotJVM采用这种技术）则吸取第一代JVM和第二代JVM的经验，采用两者结合的方式 。

    自适应优化：**开始对所有的代码都采取解释执行的方式，并监视代码执行情况，然后对那些经常调用的方法启动一个后台线程，将其编译为本地代码，并进行仔细优化。若方法不再频繁使用，则取消编译过的代码，仍对其进行解释执行。**

1、**java编译生成的字节码，在所有[操作系统](https://www.baidu.com/s?wd=%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F&from=1012015a&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1dBP1fvPWmvPj-hPjc4rHKW0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPHnYn1TLrjRk" \t "_blank)都是一样，故其有这样的特点**：  
 write once, run anywhere.其意思：只需要一次编码，就可以在任何环境下运行。  
2、不同的[操作系统](https://www.baidu.com/s?wd=%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F&from=1012015a&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1dBP1fvPWmvPj-hPjc4rHKW0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPHnYn1TLrjRk" \t "_blank)，其java 虚拟机是不一样的。**虚拟机将java字节代码转换对应[操作系统](https://www.baidu.com/s?wd=%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F&from=1012015a&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1dBP1fvPWmvPj-hPjc4rHKW0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPHnYn1TLrjRk" \t "_blank)的  
 相关指令，保证其正常运行。**3、java 系统支持所有的硬件的平台，不存在你提及的问题，你可以放心使用。  
**4、解释器在java虚拟机中，编译器在JDK或JRE 中。**  
5、java虚拟机就是常说的java 运行环境，其缩写是 JRE，安装在操作系统下的一个目录中，  
 这个目录在安装时可以由你自行指定，就像你安装其它应用软件一样。JDK中包含了JRE，  
 还有开发环境，如编译器，帮助文档生成器，以及系统API的jar库文件等。

**所有的Java虚拟机的执行引擎都是一致的：输入的是字节码文件、处理过程是等效字节码解析过程，输出的是执行结果。（其实就是通过编译器将JAVA源代码编译成Class字节码文件，然后该字节码文件在不同的操作系统中都可以使用，通过虚拟机根据不同系统根据不同指令进行解释执行过程，最后在不同的系统可以输出相同的结果）所以重点就是在虚拟机的字节码执行引擎是如何进行字节码解析操作的。所以：**

接下来，就让我们看一下，字节码执行引擎是如何一步一步**调用方法，如何调用指令进行解析操作**。其也是**基于操作数栈的字节码执行引擎。执行引擎运行的所有字节码指令只针对当前栈帧进行操作。（所以首先我们要跟着文章从123开始学下去）**

# 1.运行时栈帧结构

栈帧（Stack Frame） 是用于虚拟机执行时**方法调用和方法执行**时的数据结构，它是虚拟栈数据区的组成元素。每一个方法从调用到方法返回都对应着一个栈帧入栈出栈的过程。

每一个栈帧在编译程序代码的时候所需要多大的局部变量表，多深的操作数栈都已经决定了，并且写入到方发表的 Code 属性之中，一次一个栈帧需要多少内存，不会受到程序运行期变量数据的影响，仅仅取决于具体的虚拟机实现。

一个线程中方法调用可能很长，很多方法都处于执行状态。对于**执行引擎来说，只有处于栈顶的栈帧才是有效的，称为当前栈帧（Current Stack Frame），与之相关联的方法称为当前方法（Current Method）。**

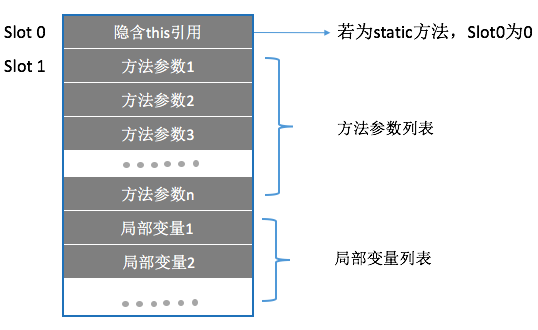
在概念模型上，典型的栈帧主要由 局部变量表（Local Stack Frame）、操作数栈（Operand Stack）、动态链接（Dynamic Linking）、返回地址（Return Address）组成，如下图所示：



接下来我们分别讲解栈帧中这四部分的具体结构。

1、局部变量表

* 局部标量表 是一组变量值的存储空间，用于存放 **方法参数** 和 **局部变量**。局部变量表以Slot（变量槽）为基本单位，int,float,reference，boolean, byte都占1 Slot；long和double数据被切割成连续2 Slots。
* 局部变量中Slot可重用，当方法体内定义的变量超出其作用域时，会被重用。



虚拟机通过**索引定位**的方式使用局部变量表。之前我们知道，局部变量表存放的是方法参数（每个方法传进来的实参）和局部变量（方法内定义的变量）。当调用方法是非static 方法时，局部变量表中第0位索引的 Slot 默认是用于传递方法所属对象实例的引用，即 “this” 关键字指向的对象（比如X.test。则保存这个X对象）。分配完方法参数后，便会依次分配方法内部定义的局部变量。

为了节省栈帧空间，局部变量表中的 Slot 是可以重用的。当离开了某些变量的作用域之后，这些变量对应的 Slot 就可以交给其他变量使用。这种机制有时候会影响垃圾回收行为。

考虑下面两段代码（需加上 -verbose:gc参数）：

代码一：

**[java]** [view plain](https://blog.csdn.net/zq602316498/article/details/38926607) [copy](https://blog.csdn.net/zq602316498/article/details/38926607)

1. **public** **static** **void** main(String[] args){
2. {
3. **byte**[] placeholder = **new** **byte**[64\*1024\*1024];
4. }
5. System.gc();
6. }

运行结果：

**[plain]** [view plain](https://blog.csdn.net/zq602316498/article/details/38926607) [copy](https://blog.csdn.net/zq602316498/article/details/38926607)

1. [GC 602K->378K(15872K), 0.0603803 secs]
2. [Full GC 378K->378K(15872K), 0.0323107 secs]
3. [Full GC 66093K->65914K(81476K), 0.0074124 secs]

代码二：

**[java]** [view plain](https://blog.csdn.net/zq602316498/article/details/38926607) [copy](https://blog.csdn.net/zq602316498/article/details/38926607)

1. **public** **static** **void** main(String[] args){
2. {
3. **byte**[] placeholder = **new** **byte**[64\*1024\*1024];
4. }
5. **int** a = 0;  //a为**局部变量**
6. System.gc();
7. }

运行结果：

**[plain]** [view plain](https://blog.csdn.net/zq602316498/article/details/38926607) [copy](https://blog.csdn.net/zq602316498/article/details/38926607)

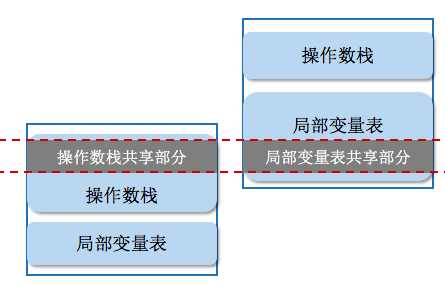
1. [GC 602K->378K(15872K), 0.0018270 secs]
2. [Full GC 378K->378K(15872K), 0.0057871 secs]
3. [Full GC 66093K->378K(81476K), 0.0054067 secs]

分析：通过结果可以知道，代码一和代码二内的 placeholder 变量在 System.gc() 执行后理应被回收了，可是结果却是只有代码二被回收了，这是为什么呢？

        这是因为代码一中 placeholder 虽然离开了作用域，但之后没有任何局部变量对其进行读写，也就是说其占用的 Slot 没有被复用，也就是说 placeholder 占用的内存仍然有引用指向它，因而它没有被回收。而代码二中的变量a由于复用了 placeholder 的 Slot ，导致 placeholder 引用被删除，因此占用的内存空间被回收。

2、操作数栈

* JVM对栈帧做了优化处理，令下面的栈帧的操作数栈和上面的栈帧局部变量表部分重叠，重叠部分就是**调用方法所需的参数部分（即不同方法的相同实参）。**



操作数栈（Operand Stack）也常称为操作栈，是一个后入先出栈。在Class 文件的Code 属性的 max\_stacks 指定了执行过程中最大的栈深度。Java 虚拟机的解释执行引擎称为”**基于栈的执行引擎**“，这里的栈就是指操作数栈。

方法执行中进行算术运算或者是调用其他的方法进行参数传递的时候是通过操作数栈进行的。

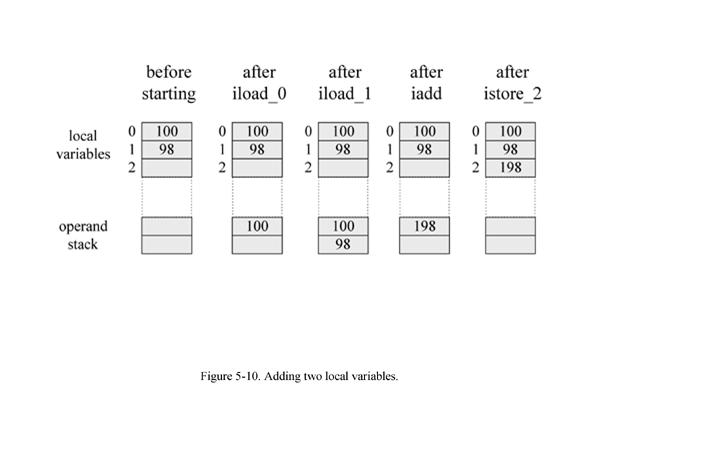
在概念模型中，两个栈帧是相互独立的。但是大多数虚拟机的实现都会进行优化，令两个栈帧出现一**部分重叠**。令下面的部分操作数栈与上面的局部变量表重叠在一块，这样在方法调用的时候可以**共用一部分数据，无需进行额外的参数复制传递。**

      和局部变量区一样，操作数栈也是被组织成一个以字长为单位的数组。但是和前者不同的是，它不是通过索引来访问，而是通过标准的栈操作—**压栈和出栈—来访问的**。比如，如果某个指令把一个值压入到操作数栈中，稍后另一个指令就可以弹出这个值来使用。  
      虚拟机在操作数栈中**存储数据的方式和在局部变量表是一样的**：如int、long、float、double、reference和returnType的存储。对于byte、short以及char类型的值在压入到操作数栈之前，也会被转换为int。  
      **虚拟机把操作数栈作为它的工作区**——大多数指令都要从这里**弹出数据，执行运算，然后把结果压回操作数栈。**比如，iadd指令就要从操作数栈中弹出两个整数，执行加法运算，其结果又压回到操作数栈中，看看下面的示例，它演示了虚拟机是如何把两个int类型的局部变量相加，再把结果保存到第三个局部变量的：

**Java代码  [收藏代码](javascript:void())**

1. begin
2. iload\_0    // push the int in local variable 0 onto the stack
3. iload\_1    // push the int in local variable 1 onto the stack
4. iadd       // pop two ints, add them, push result
5. istore\_2   // pop int, store into local variable 2
6. end

      在这个字节码序列里，前两个指令iload\_0和iload\_1将存储在局部变量中**索引为0和1的整数压入操作数栈中**，其后iadd指令从操作数栈中**弹出那两个整数相加，再将结果压入操作数栈**。第四条指令**istore\_2则从操作数栈中弹出结果，并把它存储到局部变量区索引为2的位置**。下图详细表述了这个过程中局部变量和操作数栈的状态变化，图中没有使用的局部变量区和操作数栈区域以空白表示。



3、动态连接

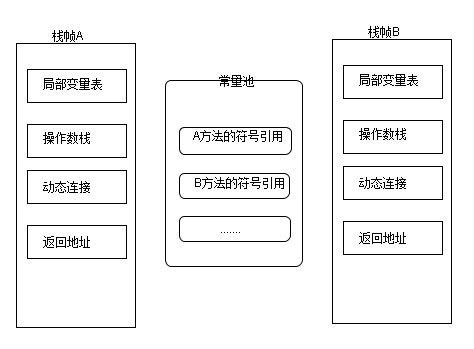
 每个栈帧**指向**运行时常量池中**该栈帧所属的方法的引用**，也就是**字节码的发放调用的引用**。动态链接就是将符号引用所表示的方法，**转换成方法的直接引用**。加载阶段或第一次使用时转化为直接引用的（将变量的访问转化为访问这些变量的存储结构所在的运行时内存位置）就叫做静态解析。JVM的动态链接还支持运行期转化为直接引用。也可以叫做Late Binding,晚期绑定。

在说明什么是动态连接之前先看看方法的大概调用过程,首先在虚拟机运行的时候,运行时常量池会保存大量的符号引用,**这些符号引用可以看成是每个方法的间接引用,**如果代表栈帧A的方法想调用代表栈帧B的方法,那么这个虚拟机的方法调用指令就**会以B方法的符号引用作为参数**,但是因为符号引用并不是直接指向代表B方法的内存位置,所以在调用之前还必须要将符号引用**转换为直接引用（直接引用则是指向B方法的内存位置）**。然后通过直接引用才可以访问到真正的方法,这时候就有一点需要注意, 如果符号引用是在**类加载阶段或者第一次使用的时候转化为直接应用**,那么这种转换成为**静态解析**,如果是在**运行期间转换为直接引用**,那么这种转换就成为**动态连接。**

在解析时，虚拟机执行两个基本任务

1.查找被引用的类，（如果必要的话就装载它）

2.将符号引用替换为直接引用，这样当它以后再次遇到相同的引用时，它就可以立即使用这个直接引用，而不必花时间再次解析这个符号引用了。



Class 文件中**存放了大量的符号引用**，字节码中的方法调用指令就是**以常量池中指向方法的符号引用作为参数**。这些符号引用一部分会在类加载阶段或第一次使用时转化为直接引用，这种转化称为**静态解析（如静态方法，构造方法）**。另一部分将在第一次运行期间转化为直接引用，这部分称为**动态连接（因为此时常量池的**内容会变为前面解析好的地址，所以来到这儿不用重新解析了直接可以用**）**。

什么是符号引用？通过常量池B的符号引用形式如下：类如一颗树的形状。**（带有类型（tag） / 结构（符号间引用层次））、**

**举个例子：**

这在Class文件中的实际编码为（以十六进制表示，Class文件里使用高位在前字节序（big-endian））：

[0A] [00 03] [00 11]

其中0x0A是CONSTANT\_Methodref\_info的tag，后面的0x0003和0x0011是该常量池项的两个部分：class\_index和name\_and\_type\_index。这两部分分别都是常量池下标，引用着另外两个常量池项。



4、方法返回地址

**当一个方法开始执行以后，只有两种方法可以退出当前方法：**

* 当执行遇到返回指令，会将**返回值（返回地址，里面存放返回值）传递给上层的方法调用者，这种退出的方式称为正常完成出口**（Normal Method Invocation Completion），一般来说，**调用者的PC计数器可以作为返回地址**。
* 当执行遇到异常，并且当前方法体内没有得到处理，**就会导致方法退出，此时是没有返回值的，称为异常完成出口**（Abrupt Method Invocation Completion），返回地址要通过异常处理器表来确定。

当方法返回时，可能进行3个操作：

* **恢复上层方法的局部变量表和操作数栈**
* **把返回值压入调用者调用者栈帧的操作数栈**
* **调整 PC 计数器的值以指向方法调用指令后面的一条指令**

# 2.方法调用

**方法调用**的主要任务就是确定被调用方法的版本（即调用哪一个方法（在动态连接里有引用的）），该过程不涉及方法具体的运行过程。按照调用方式共分为两类：

1. **解析调用：**是静态的过程，在**编译期间就完全确定目标方法**。
2. **分派调用：根据分派的形态**即可能是静态，也可能是动态的，根据**分派标准**可以分为单分派和多分派。两两组合有形成了静态单分派、静态多分派、动态单分派、动态多分派

**解析调用：**

**所有方法调用中的目标方法在Class文件里面都是一个常量池中的符号引用**。

在类加载的解析阶段，一部分符号引用会被转化为直接引用，这种解析成立的前提是：**方法在程序真正运行之前就有一个可确定的调用版本，且这个方法的调用版本在运行时是不可改变的**。符合这个条件的有**静态方法**和**私有方法**两大类。

JVM提供了5条方法调用的字节码指令：

 1.invokestatic:调用静态方法

2.invokespecial:调用构造器，私有方法和父类方法

3.invokevirtual:调用虚方法

4.invokeinterface:调用接口方法

5.invokedynamic：现在运行时动态解析出该方法，然后执行。

invokestatic & invokespecial 对应的方法，都是在加载解析后，可以直接确定的。所以这些方法为非虚方法。除此以外（除**静态方法，实例构造器，私有方法，父类方法**以外）其他方法称为虚方法。

**JAVA非虚方法除了invokestatic和invokespecial以外，还有一种就是final修饰的方法，因为该方法无法被覆盖**，java规定 final修饰的是一种非虚方法。这种被final修饰的方法是用**invokevirtual**指令调用的。

**分派调用**

**静态分派**

Parent father = new Son();

这句中，Parent被称为静态类型，Son称为实际类型。

**虚拟机（编译器）重载时通过参数的静态类型作为判断依据**。**所有依赖静态类型来定位方法执行版本的分派动作，都称为静态分派**。静态分派的最典型应用就是方法重载。

在重载的情况下，很多时候重载版本并不是唯一的，而是**寻找一个最合适**的版本。比如存在多个重载方法的情况中，调用的目标顺序是一定的。

**动态分派**

它与重写（override）有着密切的关系。

相对于前面的重载中，**引用类型对于具体调用哪个方法起决定性作用**，在重写中，**引用指向的对象的具体类型决定了调用的具体目标方法**。

**Java代码  [收藏代码](javascript:void())**

1. Human man = **new** Man();
2. Human woman = **new** Woman();
3. man.sayHello();
4. woman.sayHello();
5. man = **new** Woman();
6. man.sayHello();

这样一段代码生成的字节码为：

**Java代码  [收藏代码](javascript:void())**

1. 0: **new** #16;
2. 3; dup
3. 4: invokespecial #18;
4. 7; astore\_1
5. 8: **new** #19;
6. 11:dup
7. 12:invokespecial #21;
8. 15:astore\_2
9. 16:aload\_1
10. 17:invokevirtual #22;
11. 20:aload\_2
12. 21:invokevirtual #22;
13. 24:**new** #19;
14. 27:dup
15. 28:invokespecial #21;
16. 31:astore\_1
17. 32:aload\_1
18. 33:invokevirtual #22
19. 36:**return**

 0-15行是准备工作，用于生成对象，初始化对象，并将两个实例存放在第一和第二个局部变量表slot中。

16和20行分别将刚创建的两个对象引用压到栈顶，这两个对象是将要执行的sayHello()方法的所有者，成为接收者。

17和21行是方法调用指令，可见指令和参数都是一样的，都是invokevirtual常量池中第22项的常量---Human.sayHello()的符号引用，而结果是这两次调用的结果不同，原因是**invokevirtual**指令的运行时解析过程：

1. **找到操作数栈顶的第一个元素所指向的对象的实际类型，记作C**
2. 如果在类型C中**找到**与常量池中的描述符和简单名称都相符的方法，则进行访问权限校验，通过则返回**该方法直接引用(若未加载则通过动态连接将其方法由符号引用转变为直接引用)**，不通过抛出java.lang.IllegalAccessError
3. 否则，按照继承关系从下往上依次对C的各个父类进行第二步的搜索和验证
4. 没找到合适方法，抛出java.lang.AbstractMethodError异常

由于**第一步是解析成对象的实际类型，因此两次调用的结果不一样。**

这个顺序实际是：

**找到实际类型---在该类型中搜索--在该类型的继承结构中自下而上搜索---抛出异常**

**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。**

**单分派和多分派**

**方法的接收者与方法的参数统称为方法的宗量**。

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两种

单分派是根据一个宗量对目标方法进行选择，

多分派是根据多个宗量对目标方法进行选择。

根据之前方法调用可能生成的4种字节码，找到对应方法可能生成的字节码，再根据字节码解析过程进行判断。

**首先进行静态分派，生成相应的字节码，在常量池中生成对应的方法符号引用，这个过程根据了两个宗量进行选择（接收者（可以理解为方法名）和参数（可以理解为方法的参数）），因此静态分派是多分派类型。**

**再进行动态分派，将符号引用变成直接引用时，只对方法的接收者进行选择，（可以理解为方法名，因为每个重写方法参数都一样，）因此只有一个宗量，动态分派是单分派。**

### JVM实现动态分派

动态分派在Java中被大量使用，使用频率及其高，如果在每次动态分派的过程中都要重新在类的方法元数据中搜索合适的目标的话就可能影响到执行效率，因此JVM在类的**方法区中建立虚方法表**（virtual method table）来提高性能。**每个类中都有一个虚方法表，表中存放着各个方法的实际入口**。如果某个方法在子类中没有被重写，那子类的虚方法表中该方法的地址入口和父类该方法的地址入口一样，即子类的方法入口指向父类的方法入口。如果子类重写父类的方法，那么子类的虚方法表中该方法的实际入口将会被替换为指向子类实现版本的入口地址。 **（即子方法有重写就是子方法，没有就是父方法的入口地址。）**  
那么虚方法表什么时候被创建？**虚方法表会在类加载的连接阶段被创建并开始初始化，类的变量初始值准备完成之后，JVM会把该类的方法表也初始化完毕。**

# 3.基于栈的字节码解释执行引擎

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**，与之对应的是寄存器指令集架构。

比较起来，基于栈的指令集主要的优点是可移植性。而寄存器架构有更好的性能。

### 6.基于栈的字节码执行引擎

基于栈的指令集 和基于寄存器的指令集。

先看一个加法过程：

iconst\_1

iconst\_1

iadd

istore\_0

**这是基于栈的，也就是上面说的操作数栈。**

**先把2个元素要入栈，然后相加，放回栈顶，然后把栈顶的值存在slot 0里面。**

基于寄存器的就不解释了。

基于寄存器 和基于栈的指令集现在都存在。所以很难说孰优孰劣。

基于栈的指令集 是和硬件无关的，而基于寄存器则依赖于硬件基础。基于寄存器在效率上优势。

但是虚拟机的出现，就是为了提供跨平台的支持，**所以jvm的执行引擎是基于栈的指令集**。

[复制代码](javascript:void(0);)

public int calc()

{

int a = 100;

int b = 200;

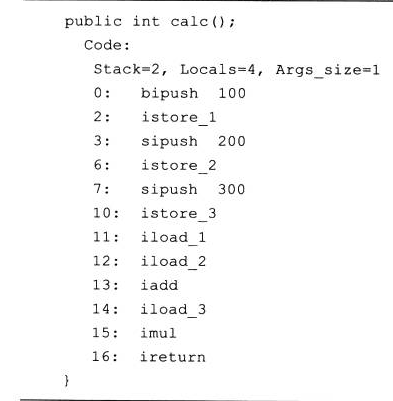
int c = 300;

return (a+b)\*c;

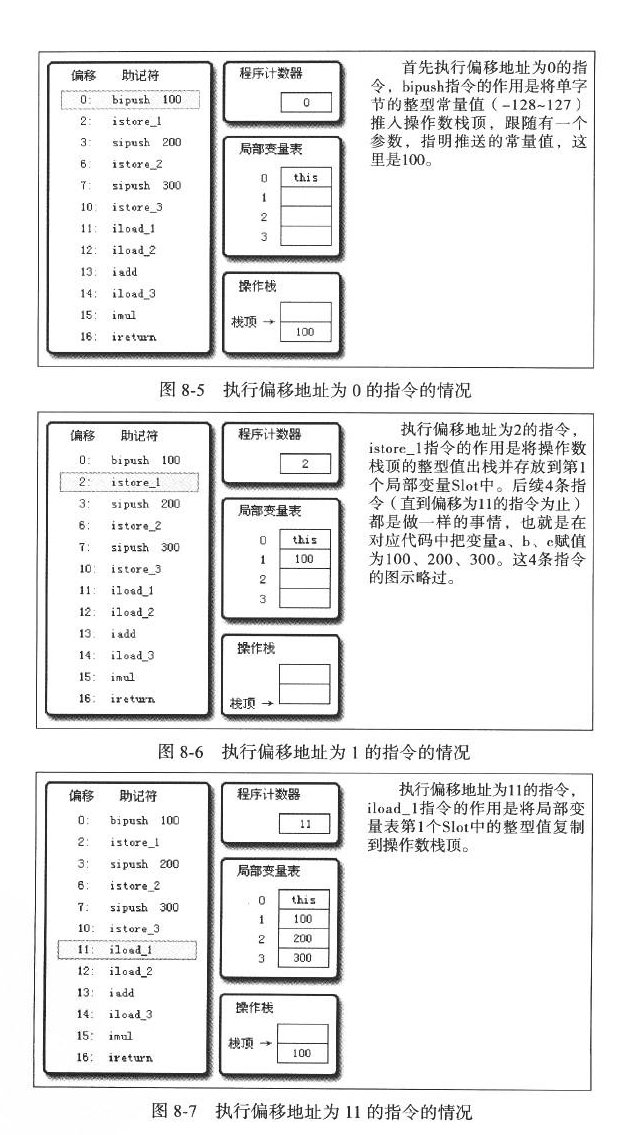
}

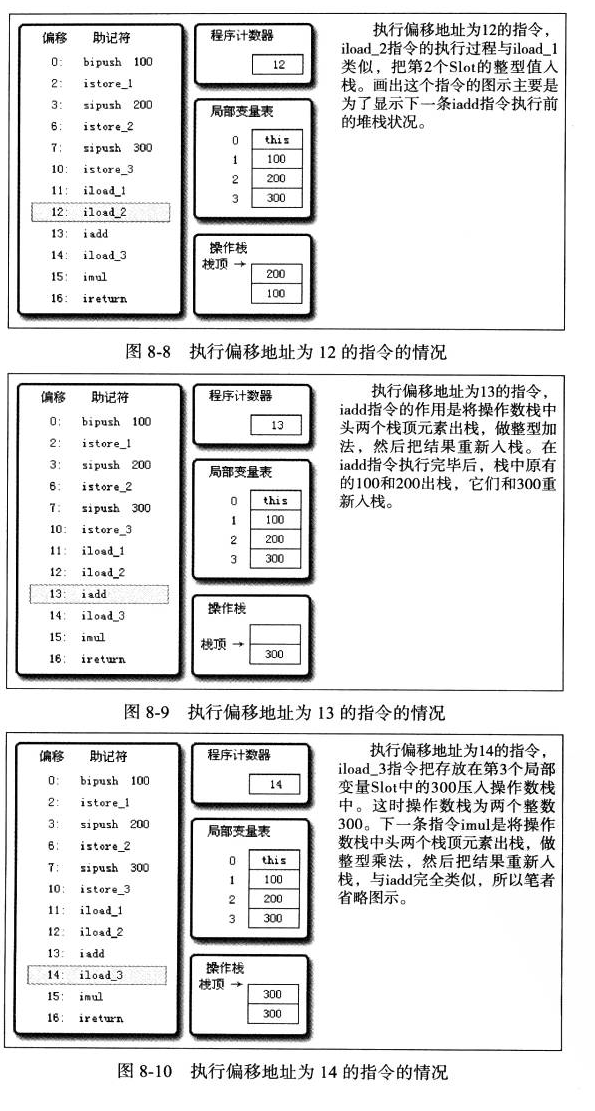
[复制代码](javascript:void(0);)

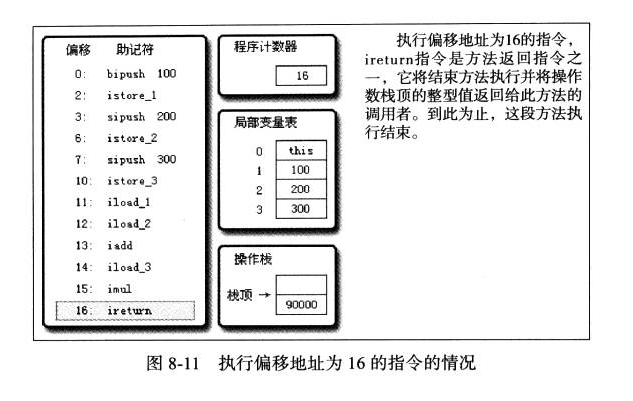
以下是javap的分析结果：



以下图片描述了整个执行过程中代码，操作数栈，& 局部变量表的变化。







  这些100，200，300都是由栈中的reference指的堆区里的对象，堆区里的对象又指向方法区的数据。方法区中的常量池取得这些值，然后推入操作栈，再推进局部变量表里的。

这些过程只是一个概念模型，实际虚拟机会有很多优化的情况。