<https://baike.baidu.com/item/JVM/2902369?fr=aladdin>

JVM是Java Virtual Machine（Java[虚拟机](https://baike.baidu.com/item/%E8%99%9A%E6%8B%9F%E6%9C%BA)）的缩写，JVM是一种用于计算设备的规范，它是一个虚构出来的计算机，是通过在实际的计算机上仿真模拟各种计算机功能来实现的。

[Java语言](https://baike.baidu.com/item/Java%E8%AF%AD%E8%A8%80)的一个非常重要的特点就是与平台的无关性。而使用Java虚拟机是实现这一特点的关键。一般的高级语言如果要在不同的平台上运行，至少需要编译成不同的[目标代码](https://baike.baidu.com/item/%E7%9B%AE%E6%A0%87%E4%BB%A3%E7%A0%81" \t "_blank)。而引入Java语言虚拟机后，Java语言在不同平台上运行时不需要重新编译。Java语言使用Java虚拟机屏蔽了与具体平台相关的信息，使得Java语言[编译程序](https://baike.baidu.com/item/%E7%BC%96%E8%AF%91%E7%A8%8B%E5%BA%8F" \t "_blank)只需生成在Java虚拟机上运行的目标代码（[字节码](https://baike.baidu.com/item/%E5%AD%97%E8%8A%82%E7%A0%81" \t "_blank)），就可以在多种平台上不加修改地运行。Java虚拟机在执行字节码时，把字节码解释成具体平台上的[机器指令](https://baike.baidu.com/item/%E6%9C%BA%E5%99%A8%E6%8C%87%E4%BB%A4" \t "_blank)执行。这就是Java的能够“一次编译，到处运行”的原因。

## 支持类型

[编辑](javascript:;)

### 基本数据类型

byte://1[字节](https://baike.baidu.com/item/%E5%AD%97%E8%8A%82)有符号整数的补码

[](https://baike.baidu.com/pic/JVM/2902369/0/d35a10f44e2fc606dcc474c7?fr=lemma&ct=single)JVM

short://2字节有符号整数的补码

int://4字节有符号整数的补码

long://8字节有符号整数的补码

float://4字节IEEE754[单精度浮点数](https://baike.baidu.com/item/%E5%8D%95%E7%B2%BE%E5%BA%A6%E6%B5%AE%E7%82%B9%E6%95%B0)

double://8字节IEEE754[双精度浮点数](https://baike.baidu.com/item/%E5%8F%8C%E7%B2%BE%E5%BA%A6%E6%B5%AE%E7%82%B9%E6%95%B0)

char://2[字节](https://baike.baidu.com/item/%E5%AD%97%E8%8A%82)无符号Unicode[字符](https://baike.baidu.com/item/%E5%AD%97%E7%AC%A6)

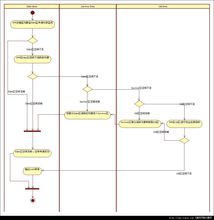
几乎所有的Java类型检查都是在编译时完成的。上面列出的原始数据类型的数据在Java执行时不需要用硬件标记。操作这些原始数据类型数据的[字节码](https://baike.baidu.com/item/%E5%AD%97%E8%8A%82%E7%A0%81" \t "_blank)（指令）本身就已经指出了[操作数](https://baike.baidu.com/item/%E6%93%8D%E4%BD%9C%E6%95%B0" \t "_blank)的数据类型，例如iadd、ladd、fadd和dadd指令都是把两个数相加，其操作数类型分别是int、long、float和double。[虚拟机](https://baike.baidu.com/item/%E8%99%9A%E6%8B%9F%E6%9C%BA" \t "_blank)没有给boolean(布尔）类型设置单独的指令。boolean型的数据是由integer指令，包括integer返回来处理的。boolean型的[数组](https://baike.baidu.com/item/%E6%95%B0%E7%BB%84" \t "_blank)则是用byte数组来处理的。虚拟机使用IEEE754格式的浮点数。不支持IEEE格式的较旧的计算机，在运行Java数值计算程序时，可能会非常慢。

### 其它数据类型

object//对一个Javaobject（对象）的4字节引用

returnAddress//4字节，用于jsr/ret/jsr-w/ret-w指令

注：Java[数组](https://baike.baidu.com/item/%E6%95%B0%E7%BB%84)被当做object处理。

[](https://baike.baidu.com/pic/JVM/2902369/0/bbe0d31134702c78b8127bc0?fr=lemma&ct=single)JVM

[虚拟机](https://baike.baidu.com/item/%E8%99%9A%E6%8B%9F%E6%9C%BA)的规范对于object内部的结构没有任何特殊的要求。在Sun公司的实现中，对object的引用是一个句柄，其中包含一对[指针](https://baike.baidu.com/item/%E6%8C%87%E9%92%88" \t "_blank)：一个指针指向该object的方法表，另一个指向该object的数据。用Java虚拟机的[字节码](https://baike.baidu.com/item/%E5%AD%97%E8%8A%82%E7%A0%81" \t "_blank)表示的程序应该遵守类型规定。Java虚拟机的实现应拒绝执行违反了类型规定的字节码程序。Java虚拟机由于字节码定义的限制似乎只能运行于32位[地址空间](https://baike.baidu.com/item/%E5%9C%B0%E5%9D%80%E7%A9%BA%E9%97%B4" \t "_blank)的机器上。但是可以创建一个Java[虚拟机](https://baike.baidu.com/item/%E8%99%9A%E6%8B%9F%E6%9C%BA)，它自动地把字节码转换成[64位](https://baike.baidu.com/item/64%E4%BD%8D)的形式。从Java虚拟机支持的数据类型可以看出，Java对数据类型的内部格式进行了严格规定，这样使得各种Java虚拟机的实现对数据的解释是相同的，从而保证了Java的与平台无关性和可移植性。

## 规格

[编辑](javascript:;)

JVM的设计目标是提供一个基于抽象规格描述的计算机模型，为解释程序开发人员提供很好的灵活性，同时也确保Java代码可在符合该规范的任何系统上运行。JVM对其实现的某些方面给出了具体的定义，特别是对Java[可执行代码](https://baike.baidu.com/item/%E5%8F%AF%E6%89%A7%E8%A1%8C%E4%BB%A3%E7%A0%81)，即[字节码](https://baike.baidu.com/item/%E5%AD%97%E8%8A%82%E7%A0%81)（Bytecode）的格式给出了明确的规格。这一规格包括[操作码](https://baike.baidu.com/item/%E6%93%8D%E4%BD%9C%E7%A0%81" \t "_blank)和[操作数](https://baike.baidu.com/item/%E6%93%8D%E4%BD%9C%E6%95%B0)的语法和数值、[标识符](https://baike.baidu.com/item/%E6%A0%87%E8%AF%86%E7%AC%A6)的数值表示方式、以及Java类文件中的Java对象、[常量缓冲](https://baike.baidu.com/item/%E5%B8%B8%E9%87%8F%E7%BC%93%E5%86%B2" \t "_blank)池在JVM的存储映象。这些定义为JVM[解释器](https://baike.baidu.com/item/%E8%A7%A3%E9%87%8A%E5%99%A8)开发人员提供了所需的信息和开发环境。Java的设计者希望给开发人员以随心所欲使用Java的自由。

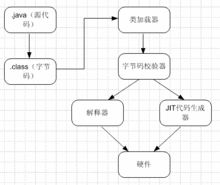
JVM定义了控制Java代码[解释执行](https://baike.baidu.com/item/%E8%A7%A3%E9%87%8A%E6%89%A7%E8%A1%8C" \t "_blank)和具体实现的五种规格，它们是：

* JVM[指令系统](https://baike.baidu.com/item/%E6%8C%87%E4%BB%A4%E7%B3%BB%E7%BB%9F)
* JVM[寄存器](https://baike.baidu.com/item/%E5%AF%84%E5%AD%98%E5%99%A8)
* JVM 栈结构
* JVM 碎片回收堆
* JVM 存储区

## 原理

[编辑](javascript:;)

JVM是java的核心和基础，在java编译器和os平台之间的虚拟处理器。它是一种基于下层的操作系统和硬件平台并利用软件方法来实现的抽象的计算机，可以在上面执行java的字节码程序。

[](https://baike.baidu.com/pic/JVM/2902369/0/0b7b02087bf40ad1bb99b82a562c11dfa8eccefe?fr=lemma&ct=single)JVM运行原理 [1]

java编译器只需面向JVM，生成JVM能理解的代码或字节码文件。Java源文件经编译器，编译成字节码程序，通过JVM将每一条指令翻译成不同平台机器码，通过特定平台运行。

**JVM执行程序的过程 ：**

I.加载.class文件

II.管理并分配内存

III.执行垃圾收集

JRE（java运行时环境）包含JVM的java程序的运行环境 [1]

JVM是Java程序运行的容器,但是他同时也是操作系统的一个进程,因此他也有他自己的运行的生命周期,也有自己的代码和数据空间。

JVM在整个jdk中处于最底层,负责与操作系统的交互,用来屏蔽操作系统环境,提供一个完整的Java运行环境,因此也叫虚拟计算机.操作系统装入JVM是通过jdk中Java.exe来完成,通过下面4步来完成JVM环境。

1.创建JVM装载环境和配置

2.装载JVM.dll

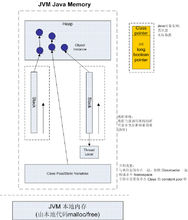
3.初始化JVM.dll并挂接到JNIENV(JNI调用接口)实例

4.调用JNIEnv实例装载并处理[class类](https://baike.baidu.com/item/class%E7%B1%BB" \t "_blank)。[2]

## 指令系统

[编辑](javascript:;)

JVM指令系统同其他计算机的指令系统极其相似。Java指令也是由[操作码](https://baike.baidu.com/item/%E6%93%8D%E4%BD%9C%E7%A0%81" \t "_blank)和操作数两部分组

[](https://baike.baidu.com/pic/JVM/2902369/0/0865b518f7d9bf6f34fa41ff?fr=lemma&ct=single)JVM

成。操作码为8位二进制数，[操作数](https://baike.baidu.com/item/%E6%93%8D%E4%BD%9C%E6%95%B0" \t "_blank)紧随在操作码的后面，其长度根据需要而不同。操作码用于指定一条指令操作的性质（在这里我们采用汇编符号的形式进行说明），如iload表示从[存储器](https://baike.baidu.com/item/%E5%AD%98%E5%82%A8%E5%99%A8" \t "_blank)中装入一个整数，anewarray表示为一个新[数组](https://baike.baidu.com/item/%E6%95%B0%E7%BB%84" \t "_blank)分配空间，iand表示两个整数的"与"，ret用于[流程控制](https://baike.baidu.com/item/%E6%B5%81%E7%A8%8B%E6%8E%A7%E5%88%B6" \t "_blank)，表示从对某一方法的调用中返回。当长度大于8位时，操作数被分为两个以上[字节](https://baike.baidu.com/item/%E5%AD%97%E8%8A%82" \t "_blank)存放。JVM采用了"big endian[3]  "的编码方式来处理这种情况，即高位bits存放在低字节中。这同 Motorola及其他的RISC CPU采用的[编码方式](https://baike.baidu.com/item/%E7%BC%96%E7%A0%81%E6%96%B9%E5%BC%8F" \t "_blank)是一致的，而与Intel采用的"little endian "的编码方式即低位bits存放在低位字节的方法不同。Java指令系统是以Java语言的实现为目的设计的，其中包含了用于调用方法和监视多线程系统的指令。Java的8[位操作](https://baike.baidu.com/item/%E4%BD%8D%E6%93%8D%E4%BD%9C)码的长度使得JVM最多有256种指令，已使用了160多种操作码。

## 寄存器

[编辑](javascript:;)

所有的CPU均包含用于保存系统状态和处理器所需信息的[寄存器组](https://baike.baidu.com/item/%E5%AF%84%E5%AD%98%E5%99%A8%E7%BB%84" \t "_blank)。如果[虚拟机](https://baike.baidu.com/item/%E8%99%9A%E6%8B%9F%E6%9C%BA)定义较多的寄存器，便可以从中得到更多的信息而不必对栈或内存进行访问，这有利于提高运行速度。然而，如果虚拟机中的寄存器比实际CPU的寄存器多，在实现虚拟机时就会占用处理器大量的时间来用常规[存储器](https://baike.baidu.com/item/%E5%AD%98%E5%82%A8%E5%99%A8" \t "_blank)模拟寄存器，这反而会降低虚拟机的效率。针对这种情况，JVM只设置了4个最为常用的寄存器。它们是：

pc程序计数器

optop操作数栈顶指针

frame当前执行环境指针

vars指向当前执行环境中第一个[局部变量](https://baike.baidu.com/item/%E5%B1%80%E9%83%A8%E5%8F%98%E9%87%8F" \t "_blank)的[指针](https://baike.baidu.com/item/%E6%8C%87%E9%92%88)

所有寄存器均为[32位](https://baike.baidu.com/item/32%E4%BD%8D" \t "_blank)。pc用于记录程序的执行。optop,frame和vars用于记录指向Java栈区的指针。

## 栈结构

[编辑](javascript:;)

作为基于栈结构的计算机，Java栈是JVM存储信息的主要方法。当JVM得到一个Java[字节码](https://baike.baidu.com/item/%E5%AD%97%E8%8A%82%E7%A0%81)应用程序后，便为该代码中一个类的每一个方法创建一个栈框架，以保存该方法的状态信息。每个栈框架包括以下三类信息：

[局部变量](https://baike.baidu.com/item/%E5%B1%80%E9%83%A8%E5%8F%98%E9%87%8F)

执行环境

[操作数](https://baike.baidu.com/item/%E6%93%8D%E4%BD%9C%E6%95%B0)栈

局部变量用于存储一个类的方法中所用到的局部变量。vars寄存器指向该变量表中的第一个局部变量。

执行环境用于保存[解释器](https://baike.baidu.com/item/%E8%A7%A3%E9%87%8A%E5%99%A8)对Java字节码进行解释过程中所需的信息。它们是：上次调用的方法、局部变量[指针](https://baike.baidu.com/item/%E6%8C%87%E9%92%88" \t "_blank)和操作数栈的栈顶和栈底指针。执行环境是一个执行一个方法的控制中心。例如：如果解释器要执行iadd（整数加法），首先要从frame寄存器中找到当前执行环境，而后便从执行环境中找到操作数栈，从栈顶弹出两个整数进行加法运算，最后将结果压入栈顶。

操作数栈用于存储运算所需操作数及运算的结果。

## 碎片回收

[编辑](javascript:;)

Java类的实例所需的存储空间是在堆上分配的。[解释器](https://baike.baidu.com/item/%E8%A7%A3%E9%87%8A%E5%99%A8" \t "_blank)具体承担为类实例分配空间的工作。解释器在为一个实例分配完存储空间后，便开始记录对该实例所占用的内存区域的使用。一旦对象使用完毕，便将其回收到堆中。在Java语言中，除了new语句外没有其他方法为一对象申请和释放内存。对内存进行释放和回收的工作是由Java运行系统承担的。这允许Java运行系统的设计者自己决定碎片回收的方法。在SUN公司开发的Java解释器和Hot Java环境中，碎片回收用后台线程的方式来执行。这不但为运行系统提供了良好的性能，而且使程序设计人员摆脱了自己控制内存使用的风险。

## 存储区

[编辑](javascript:;)

JVM有两类存储区：[常量缓冲](https://baike.baidu.com/item/%E5%B8%B8%E9%87%8F%E7%BC%93%E5%86%B2" \t "_blank)池和方法区。常量缓冲池用于存储类名称、方法和字段名称以及串常量。方法区则用于存储Java方法的[字节码](https://baike.baidu.com/item/%E5%AD%97%E8%8A%82%E7%A0%81" \t "_blank)。对于这两种[存储区域](https://baike.baidu.com/item/%E5%AD%98%E5%82%A8%E5%8C%BA%E5%9F%9F)具体实现方式在JVM规格中没有明确规定。这使得Java应用程序的存储布局必须在运行过程中确定，依赖于具体平台的实现方式。JVM是为Java字节码定义的一种独立于具体平台的规格描述，是[Java平台](https://baike.baidu.com/item/Java%E5%B9%B3%E5%8F%B0" \t "_blank)独立性的基础。JVM还存在一些限制和不足，有待于进一步的完善，但无论如何，JVM的思想是成功的。

对比分析：如果把Java原程序想象成我们的C++原程序，Java原程序编译后生成的字节码就相当于C++原程序编译后的80x86的[机器码](https://baike.baidu.com/item/%E6%9C%BA%E5%99%A8%E7%A0%81" \t "_blank)（二进制[程序文件](https://baike.baidu.com/item/%E7%A8%8B%E5%BA%8F%E6%96%87%E4%BB%B6" \t "_blank)），JVM[虚拟机](https://baike.baidu.com/item/%E8%99%9A%E6%8B%9F%E6%9C%BA)相当于80x86计算机系统，Java[解释器](https://baike.baidu.com/item/%E8%A7%A3%E9%87%8A%E5%99%A8)相当于80x86CPU。在80x86CPU上运行的是机器码，在Java解释器上运行的是Java字节码。Java解释器相当于运行Java[字节码](https://baike.baidu.com/item/%E5%AD%97%E8%8A%82%E7%A0%81)的“CPU”，但该“CPU”不是通过硬件实现的，而是用软件实现的。Java解释器实际上就是特定的平台下的一个应用程序。只要实现了特定平台下的解释器程序，Java字节码就能通过解释器程序在该平台下运行，这是Java跨平台的根本。当前，并不是在所有的平台下都有相应Java解释器程序，这也是Java并不能在所有的平台下都能运行的原因，它只能在已实现了Java解释器程序的平台下运行。

## 运行数据

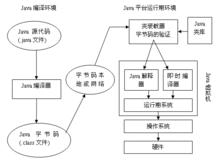
[编辑](javascript:;)

JVM定义了若干个程序执行期间使用的数据区域。这个区域里的一些数据在JVM启动的时候创建，在JVM退出的时候销毁。而其他的数据依赖于每一个线程，在线程创建时创建，在线程退出时销毁。分别有**程序计数器，堆，栈，方法区，运行时常量池。[3]**

## 体系结构

[编辑](javascript:;)

JVM可以由不同的厂商来实现。由于厂商的不同必然导致JVM在实现上的一些不同，然而JVM还是可以实现跨平台的特性，这就要归功于设计JVM时的体系结构了。我们知道，一个JVM实例的行为不光是它自己的事，还涉及到它的子系统、[存储区域](https://baike.baidu.com/item/%E5%AD%98%E5%82%A8%E5%8C%BA%E5%9F%9F" \t "_blank)、数据类型和指令这些部分，它们描述了JVM的一个抽象的内部体系结构，其目的不光规定实现JVM时它内部的体系结构，更重要的是提供了一种方式，用于严格定义实现时的外部行为。每个JVM都有两种机制，一个是装载具有合适名称的类(类或是接口），叫做类装载子系统；另外的一个负责执行包含在已装载的类或接口中的指令，叫做运行引擎。每个JVM又包括方法区、堆、Java栈、[程序计数器](https://baike.baidu.com/item/%E7%A8%8B%E5%BA%8F%E8%AE%A1%E6%95%B0%E5%99%A8" \t "_blank)和本地方法栈这五个部分，这几个部分和类装载机制与运行引擎机制一起组成的体系结构图为：

[](https://baike.baidu.com/pic/JVM/2902369/0/ac75478285acf3b90cf4d213?fr=lemma&ct=single)JVM体系结构

JVM的每个实例都有一个它自己的方法域和一个堆，运行于JVM内的所有的线程都共享这些区域；当[虚拟机](https://baike.baidu.com/item/%E8%99%9A%E6%8B%9F%E6%9C%BA" \t "_blank)装载类文件的时候，它解析其中的二进制数据所包含的类信息，并把它们放到方法域中；当程序运行的时候，JVM把程序初始化的所有对象置于堆上；而每个线程创建的时候，都会拥有自己的[程序计数器](https://baike.baidu.com/item/%E7%A8%8B%E5%BA%8F%E8%AE%A1%E6%95%B0%E5%99%A8" \t "_blank)和Java栈，其中程序计数器中的值指向下一条即将被执行的指令，线程的Java栈则存储为该线程调用Java方法的状态；本地方法调用的状态被存储在本地方法栈，该方法栈依赖于具体的实现。

下面分别对这几个部分进行说明。

执行引擎处于JVM的核心位置，在Java[虚拟机](https://baike.baidu.com/item/%E8%99%9A%E6%8B%9F%E6%9C%BA)规范中，它的行为是由[指令集](https://baike.baidu.com/item/%E6%8C%87%E4%BB%A4%E9%9B%86)所决定的。尽管对于每条指令，规范很详细地说明了当JVM执行[字节码](https://baike.baidu.com/item/%E5%AD%97%E8%8A%82%E7%A0%81" \t "_blank)遇到指令时，它的实现应该做什么，但对于怎么做却言之甚少。Java虚拟机支持大约248个字节码。每个字节码执行一种基本的CPU运算，例如，把一个整数加到寄存器，[子程序](https://baike.baidu.com/item/%E5%AD%90%E7%A8%8B%E5%BA%8F" \t "_blank)转移等。Java指令集相当于Java程序的汇编语言。Java指令集中的指令包含一个单字节的操作符，用于指定要执行的操作，还有0个或多个[操作数](https://baike.baidu.com/item/%E6%93%8D%E4%BD%9C%E6%95%B0" \t "_blank)，提供操作所需的参数或数据。许多指令没有操作数，仅由一个单字节的操作符构成。

虚拟机的内层循环的执行过程如下：

do{

取一个操作符字节；

根据操作符的值执行一个动作；

}while（程序未结束）

由于[指令系统](https://baike.baidu.com/item/%E6%8C%87%E4%BB%A4%E7%B3%BB%E7%BB%9F)的简单性，使得[虚拟机](https://baike.baidu.com/item/%E8%99%9A%E6%8B%9F%E6%9C%BA)执行的过程十分简单，从而有利于提高执行的效率。指令中操作数的数量和大小是由操作符决定的。如果操作数比一个[字节](https://baike.baidu.com/item/%E5%AD%97%E8%8A%82)大，那么它存储的顺序是高位字节优先。例如，一个16位的参数存放时占用两个字节，其值为：

第一个字节\*256+第二个字节[字节码](https://baike.baidu.com/item/%E5%AD%97%E8%8A%82%E7%A0%81" \t "_blank)。

[指令流](https://baike.baidu.com/item/%E6%8C%87%E4%BB%A4%E6%B5%81)一般只是[字节对齐](https://baike.baidu.com/item/%E5%AD%97%E8%8A%82%E5%AF%B9%E9%BD%90)的。指令tableswitch和lookup是例外，在这两条指令内部要求强制的4字节[边界对齐](https://baike.baidu.com/item/%E8%BE%B9%E7%95%8C%E5%AF%B9%E9%BD%90" \t "_blank)。对于本地方法接口，实现JVM并不要求一定要有它的支持，甚至可以完全没有。Sun公司实现Java本地接口（JNI[3]  ）是出于可移植性的考虑，当然我们也可以设计出其它的本地接口来代替Sun公司的JNI[4]  。但是这些设计与实现是比较复杂的事情，需要确保垃圾回收器不会将那些正在被本地方法调用的对象释放掉。

Java的堆是一个运行时数据区，类的实例（对象）从中分配空间，它的管理是由垃圾回收来负责的：不给程序员显式释放对象的能力。Java不规定具体使用的垃圾回收算法，可以根据系统的需求使用各种各样的算法。

Java方法区与传统语言中的编译后代码或是Unix进程中的正文段类似。它保存方法代码（编译后的java代码）和[符号表](https://baike.baidu.com/item/%E7%AC%A6%E5%8F%B7%E8%A1%A8" \t "_blank)。在当前的Java实现中，方法代码不包括在垃圾回收堆中，但计划在将来的版本中实现。每个类文件包含了一个Java类或一个Java界面的编译后的代码。可以说类文件是Java语言的执行代码文件。为了保证类文件的平台无关性，Java[虚拟机](https://baike.baidu.com/item/%E8%99%9A%E6%8B%9F%E6%9C%BA)规范中对类文件的格式也作了详细的说明。其具体细节请参考Sun公司的Java虚拟机规范。

Java虚拟机的寄存器用于保存机器的运行状态，与微处理器中的某些专用寄存器类似。Java虚拟机的寄存器有四种：

pc: Java[程序计数器](https://baike.baidu.com/item/%E7%A8%8B%E5%BA%8F%E8%AE%A1%E6%95%B0%E5%99%A8)；

optop: 指向[操作数](https://baike.baidu.com/item/%E6%93%8D%E4%BD%9C%E6%95%B0" \t "_blank)栈顶端的[指针](https://baike.baidu.com/item/%E6%8C%87%E9%92%88)；

frame: 指向当前执行方法的执行环境的指针；。

vars: 指向当前执行方法的[局部变量](https://baike.baidu.com/item/%E5%B1%80%E9%83%A8%E5%8F%98%E9%87%8F" \t "_blank)区第一个变量的指针。

在上述[体系结构](https://baike.baidu.com/item/%E4%BD%93%E7%B3%BB%E7%BB%93%E6%9E%84" \t "_blank)图中，我们所说的是第一种，即程序计数器，每个线程一旦被创建就拥有了自己的程序计数器。当线程执行Java方法的时候，它包含该线程正在被执行的指令的地址。但是若线程执行的是一个本地的方法，那么[程序计数器](https://baike.baidu.com/item/%E7%A8%8B%E5%BA%8F%E8%AE%A1%E6%95%B0%E5%99%A8" \t "_blank)的值就不会被定义。

Java[虚拟机](https://baike.baidu.com/item/%E8%99%9A%E6%8B%9F%E6%9C%BA)的栈有三个区域：局部变量区、运行环境区、[操作数](https://baike.baidu.com/item/%E6%93%8D%E4%BD%9C%E6%95%B0)区。

局部变量区

每个Java方法使用一个固定大小的局部变量集。它们按照与vars[寄存器](https://baike.baidu.com/item/%E5%AF%84%E5%AD%98%E5%99%A8" \t "_blank)的字[偏移量](https://baike.baidu.com/item/%E5%81%8F%E7%A7%BB%E9%87%8F)来寻址。[局部变量](https://baike.baidu.com/item/%E5%B1%80%E9%83%A8%E5%8F%98%E9%87%8F)都是32位的。长整数和[双精度浮点数](https://baike.baidu.com/item/%E5%8F%8C%E7%B2%BE%E5%BA%A6%E6%B5%AE%E7%82%B9%E6%95%B0" \t "_blank)占据了两个局部变量的空间，却按照第一个局部变量的索引来寻址。（例如，一个具有索引n的局部变量，如果是一个双精度浮点数，那么它实际占据了[索引](https://baike.baidu.com/item/%E7%B4%A2%E5%BC%95" \t "_blank)n和n+1所代表的存储空间）虚拟机规范并不要求在局部变量中的[64位](https://baike.baidu.com/item/64%E4%BD%8D" \t "_blank)的值是64位对齐的。虚拟机提供了把局部变量中的值装载到[操作数](https://baike.baidu.com/item/%E6%93%8D%E4%BD%9C%E6%95%B0)栈的指令，也提供了把操作数栈中的值写入局部变量的指令。

JRE和JVM的区别

JRE(JavaRuntimeEnvironment，Java运行环境)，也就是Java平台。所有的Java程序都要在JRE下才能运行。JDK的工具也是Java程序，也需要JRE才能运行。为了保持JDK的独立性和完整性，在JDK的安装过程中，JRE也是安装的一部分。所以，在JDK的安装目录下有一个名为jre的目录，用于存放JRE文件。

JVM(JavaVirtualMachine，Java虚拟机)是JRE的一部分。它是一个虚构出来的计算机，是通过在实际的计算机上仿真模拟各种计算机功能来实现的。JVM有自己完善的硬件架构，如处理器、堆栈、寄存器等，还具有相应的指令系统。Java语言最重要的特点就是跨平台运行。使用JVM就是为了支持与操作系统无关，实现跨平台。[3]

运行环境区

在运行环境中包含的信息用于[动态链接](https://baike.baidu.com/item/%E5%8A%A8%E6%80%81%E9%93%BE%E6%8E%A5)，正常的方法返回以及异常捕捉。

动态链接

运行环境包括对指向当前类和当前方法的[解释器](https://baike.baidu.com/item/%E8%A7%A3%E9%87%8A%E5%99%A8)[符号表](https://baike.baidu.com/item/%E7%AC%A6%E5%8F%B7%E8%A1%A8)的[指针](https://baike.baidu.com/item/%E6%8C%87%E9%92%88)，用于支持方法代码的动态链接。方法的[class文件](https://baike.baidu.com/item/class%E6%96%87%E4%BB%B6)代码在引用要调用的方法和要访问的变量时使用符号。动态链接把符号形式的方法调用翻译成实际方法调用，装载必要的类以解释还没有定义的符号，并把变量访问翻译成与这些变量运行时的[存储结构](https://baike.baidu.com/item/%E5%AD%98%E5%82%A8%E7%BB%93%E6%9E%84)相应的[偏移地址](https://baike.baidu.com/item/%E5%81%8F%E7%A7%BB%E5%9C%B0%E5%9D%80)。动态链接方法和变量使得方法中使用的其它类的变化不会影响到本程序的代码。

正常的方法返回

如果当前方法正常地结束了，在执行了一条具有正确类型的返回指令时，调用的方法会得到一个返回值。执行环境在正常返回的情况下用于恢复调用者的[寄存器](https://baike.baidu.com/item/%E5%AF%84%E5%AD%98%E5%99%A8" \t "_blank),并把调用者的[程序计数器](https://baike.baidu.com/item/%E7%A8%8B%E5%BA%8F%E8%AE%A1%E6%95%B0%E5%99%A8" \t "_blank)增加一个恰当的数值，以跳过已执行过的方法调用指令，然后在调用者的执行环境中继续执行下去。

异常捕捉

异常情况在Java中被称作Error（错误）或Exception（异常），是Throwable类的子类，在程序中的原因是：①[动态链接](https://baike.baidu.com/item/%E5%8A%A8%E6%80%81%E9%93%BE%E6%8E%A5)错，如无法找到所需的[class文件](https://baike.baidu.com/item/class%E6%96%87%E4%BB%B6)。②运行时错，如对一个空指针的引用。程序使用了throw语句。

当异常发生时，Java[虚拟机](https://baike.baidu.com/item/%E8%99%9A%E6%8B%9F%E6%9C%BA)采取如下措施：

§ 检查与当前方法相联系的catch子句表。每个catch子句包含其有效指令范围，能够处理的异常类型，以及处理异常的代码块地址。

§ 与异常相匹配的catch子句应该符合下面的条件：造成异常的指令在其指令范围之内，发生的异常类型是其能处理的异常类型的子类型。如果找到了匹配的catch子句，那么系统转移到指定的[异常处理](https://baike.baidu.com/item/%E5%BC%82%E5%B8%B8%E5%A4%84%E7%90%86" \t "_blank)块处执行；如果没有找到异常处理块，重复寻找匹配的catch子句的过程，直到当前方法的所有嵌套的catch子句都被检查过。

§ 由于[虚拟机](https://baike.baidu.com/item/%E8%99%9A%E6%8B%9F%E6%9C%BA" \t "_blank)从第一个匹配的catch子句处继续执行，所以catch子句表中的顺序是很重要的。因为Java代码是结构化的，因此总可以把某个方法的所有的异常处理器都按序排列到一个表中，对任意可能的[程序计数器](https://baike.baidu.com/item/%E7%A8%8B%E5%BA%8F%E8%AE%A1%E6%95%B0%E5%99%A8" \t "_blank)的值，都可以用线性的顺序找到合适的异常处理块，以处理在该程序计数器值下发生的异常情况。

§ 如果找不到匹配的catch子句，那么当前方法得到一个"未截获异常"的结果并返回到当前方法的调用者，好像异常刚刚在其调用者中发生一样。如果在调用者中仍然没有找到相应的[异常处理](https://baike.baidu.com/item/%E5%BC%82%E5%B8%B8%E5%A4%84%E7%90%86)块，那么这种错误将被传播下去。如果错误被传播到最顶层，那么系统将调用一个缺省的异常处理块。

[操作数](https://baike.baidu.com/item/%E6%93%8D%E4%BD%9C%E6%95%B0)栈区

[机器指令](https://baike.baidu.com/item/%E6%9C%BA%E5%99%A8%E6%8C%87%E4%BB%A4)只从操作数栈中取操作数，对它们进行操作，并把结果返回到栈中。选择栈结构的原因是：在只有少量寄存器或非[通用寄存器](https://baike.baidu.com/item/%E9%80%9A%E7%94%A8%E5%AF%84%E5%AD%98%E5%99%A8)的机器（如Intel486）上，也能够高效地模拟[虚拟机](https://baike.baidu.com/item/%E8%99%9A%E6%8B%9F%E6%9C%BA" \t "_blank)的行为。操作数栈是32位的。它用于给方法传递参数，并从方法接收结果，也用于支持操作的参数，并保存操作的结果。例如，iadd指令将两个整数相加。相加的两个整数应该是操作数栈顶的两个字。这两个字是由先前的指令压进[堆栈](https://baike.baidu.com/item/%E5%A0%86%E6%A0%88" \t "_blank)的。这两个整数将从堆栈弹出、相加，并把结果压回到操作数栈中。

每个原始数据类型都有专门的指令对它们进行必须的操作。每个[操作数](https://baike.baidu.com/item/%E6%93%8D%E4%BD%9C%E6%95%B0" \t "_blank)在栈中需要一个存储位置，除了long和double型,它们需要两个位置。操作数只能被适用于其类型的操作符所操作。例如，压入两个int类型的数，如果把它们当作是一个long类型的数则是非法的。在Sun的[虚拟机](https://baike.baidu.com/item/%E8%99%9A%E6%8B%9F%E6%9C%BA" \t "_blank)实现中，这个限制由[字节码](https://baike.baidu.com/item/%E5%AD%97%E8%8A%82%E7%A0%81)验证器强制实行。但是，有少数操作（操作符dupe和swap），用于对运行时数据区进行操作时是不考虑类型的。

本地方法栈，当一个线程调用本地方法时，它就不再受到虚拟机关于结构和安全限制方面的约束，它既可以访问虚拟机的运行期数据区，也可以使用本地处理器以及任何类型的栈。例如，本地栈是一个C语言的栈，那么当C程序调用C函数时，函数的参数以某种顺序被压入栈，结果则返回给调用函数。在实现Java[虚拟机](https://baike.baidu.com/item/%E8%99%9A%E6%8B%9F%E6%9C%BA)时，本地方法接口使用的是C语言的模型栈，那么它的本地方法栈的调度与使用则完全与C语言的栈相同。

## 运行过程

[编辑](javascript:;)

上面对[虚拟机](https://baike.baidu.com/item/%E8%99%9A%E6%8B%9F%E6%9C%BA)的各个部分进行了比较详细的说明，下面通过一个具体的例子来分析它的运行过程。

虚拟机通过调用某个指定类的方法main启动，传递给main一个字符串[数组](https://baike.baidu.com/item/%E6%95%B0%E7%BB%84" \t "_blank)参数，使指定的类被装载，同时链接该类所使用的其它的类型，并且初始化它们。新建一java源文件并取名HelloApp.java，内容如下：

class HelloApp {

public static void main(String[] args) {

System.out.println("Hello World!");

for (int i = 0; i < args.length; i++ ) {

System.out.println(args);

}

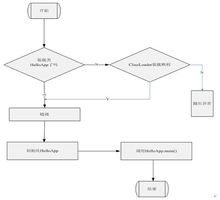
}

}

在命令模式下输入：javac HelloApp.java 进行编译，这时同目录下会产生一个编译后的文件：HelloApp.class

然后在命令行模式下键入：java HelloApp run virtual machine

将通过调用HelloApp的方法main来启动[java虚拟机](https://baike.baidu.com/item/java%E8%99%9A%E6%8B%9F%E6%9C%BA" \t "_blank)，传递给main一个包含三个字符串"run"、"virtual"、"machine"的数组。我们略述虚拟机在执行HelloApp时可能采取的步骤。

[](https://baike.baidu.com/pic/JVM/2902369/0/e865a6994c1c6f136e068c1e?fr=lemma&ct=single)JVM虚拟机运行过程

开始试图执行类HelloApp的main方法，发现该类并没有被装载，也就是说[虚拟机](https://baike.baidu.com/item/%E8%99%9A%E6%8B%9F%E6%9C%BA" \t "_blank)当前不包含该类的二进制代表，于是虚拟机使用[ClassLoader](https://baike.baidu.com/item/ClassLoader)试图寻找这样的二进制代表。如果这个进程失败，则抛出一个异常。类被装载后同时在main方法被调用之前，必须对类HelloApp与其它类型进行链接然后初始化。链接包含三个阶段：检验，准备和解析。检验检查被装载的主类的符号和语义，准备则创建类或接口的静态域以及把这些域初始化为标准的默认值，解析负责检查主类对其它类或接口的符号引用，在这一步它是可选的。类的初始化是对类中声明的静态初始化函数和静态域的初始化[构造方法](https://baike.baidu.com/item/%E6%9E%84%E9%80%A0%E6%96%B9%E6%B3%95)的执行。一个类在初始化之前它的父类必须被初始化。