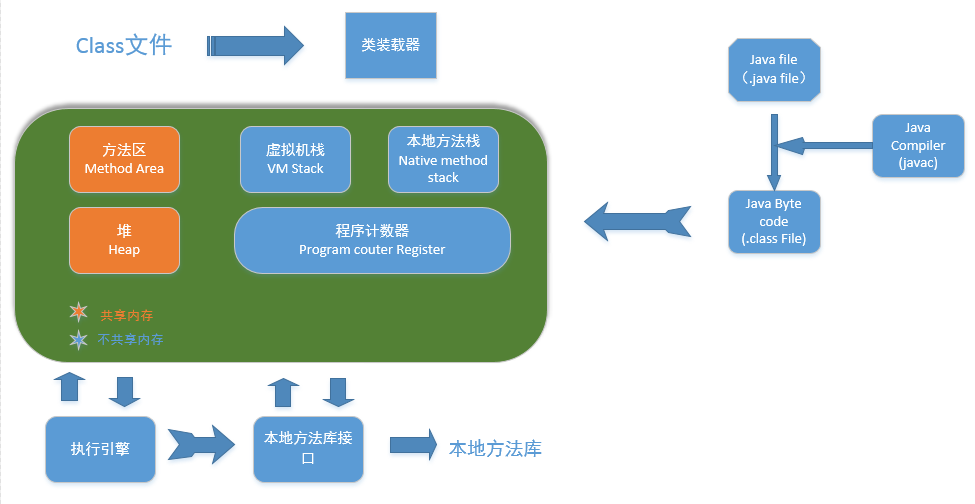
**深入理解JAVA虚拟机JVM**

<http://www.ppzedu.com/archives/201.html>

Java 虚拟机（Java virtual machine，JVM）是运行 Java 程序必不可少的机制。java之所以能实现一次编写到处执行，也就是因为jVM。  
**原理：**  
编译后的 Java 程序指令并不直接在硬件系统的 CPU 上执行，而是由 JVM 执行。JVM抹平了与具体平台相关的信息，使Java语言编译程序只需要生成在JVM上运行的目标字节码（.class）,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执行。因此实现了java平台无关性。它是 Java 程序能在多平台间进行无缝移植的可靠保证，同时也是 Java 程序的安全检验引擎（还进行安全检查）。  
JVM 是 编译后的 Java 程序（.class文件）和硬件系统之间的接口 （ 编译后：javac 是收录于 JDK 中的 Java 语言编译器。该工具可以将后缀名为. java 的源文件编译为后缀名为. class 的可以运行于 Java 虚拟机的字节码。）  
JVM的工作结构大致如下：



**JVM结构浅显了解：**

java开发人员写好的文件，都是.java文件。如果是想要被jvm加载必须要先经过编译为字节码文件。在经过调用javac编译好后生成了java字节码文件(.class)。然后才能被jvm加载。  
在生成了class文件之后，当然虚拟机并不关心Class的来源是什么语言，只要它符合Java class文件格式就可以在Java虚拟机中运行。使用Java编译器可以把Java代码编译为存储字节码的Class文件，使用JRuby等其他语言的编译器一样可以把程序代码编译成Class文件。  
**类装载器子系统**  
负责查找并装载Class 文件到内存，最终形成可以被虚拟机直接使用的Java类型，这个过程主要包括三个步骤。  
（1）装载  
装载过程负责找到二进制字节码并加载至JVM中，JVM通过类名、类所在的包名通过ClassLoader来完成类的加载，同样，**也采用以上三个元素来标识一个被加载了的类：类名+包名+ClassLoader实例ID。**  
（2）链接  
链接过程负责对二进制字节码的格式进行**校验**、初始化装载类中的**静态变量**以及解析类中**调用的接口、类。**  
在完成了校验后，JVM初始化类中的静态变量，并将其值赋为默认值。  
最后一步为对类中的所有属性、方法进行验证，以确保其需要调用的属性、方法存在，以及具备应的权限（例如public、private域权限等），会造成NoSuchMethodError、NoSuchFieldError等错误信息。  
（3）初始化  
初始化过程即为执行类中的静态初始化代码、构造器代码以及静态属性的初始化，在四种情况下初始化过程会被触发执行：  
**调用了new；反射调用了类中的方法；子类调用了初始化；JVM启动过程中指定的初始化类。**  
**方法区**  
在类装载器加载class文件到内存的过程中，虚拟机会提取其中的类型信息，并将这些信息存储到方法区。方法区用于存储已被**虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据**。由于所有线程都共享方法区，因此它们对方法区数据的访问必须被设计为是**线程安全的。**

方法区在物理上也不需要是连续的，可以选择固定大小或可扩展大小，并且方法区比堆还多了一个限制：可以选择是否执行垃圾收集。一般的，方法区上执行的垃圾收集是很少的，这也是方法区被称为永久代的原因之一（HotSpot），但这也不代表着在方法区上完全没有垃圾收集，其上的垃圾收集主要是针对**常量池的内存回收和对已加载类的卸载。**在方法区上进行垃圾收集，条件苛刻而且相当困难，效果也不令人满意，所以一般不做太多考虑，可以留作以后进一步深入研究时使用。定义了OutOfMemoryError:PermGen space异常，在内存不足时抛出。 在Java虚拟机规范中，将方法区作为堆的一个逻辑部分来对待，但事实上，方法区并不是堆（Non-Heap）；另外，不少人的博客中，将Java GC的分代收集机制分为3个代：青年代，老年代，永久代，之所以这些作者将方法区定义为“永久代”，这是因为，对于之前的HotSpot Java虚拟机的实现方式中，将分代收集的思想扩展到了方法区，并将方法区设计成了永久代。不过，除HotSpot之外的多数虚拟机，**并不将方法区当做永久代，HotSpot本身，也计划取消永久代。**本文中，由于笔者主要使用Oracle JDK6.0，因此仍将使用永久代一词。  
**内存堆：**  
存储Java程序创建的类实例。所有线程共享，因此设计程序时也要考虑到多线程访问对象(堆数据)的同步问题。这部分是运行时内存，是JVM调优以及，GC日志涉及非常多的地方。  
堆区是理解Java GC机制最重要的区域，没有之一。在JVM所管理的内存中，堆区是最大的一块，堆区也是Java GC机制所管理的**主要内存区域，堆区由所有线程共享，在虚拟机启动时创建**。堆区的存在是为了存储对象实例，原则上讲，所有的对象都在堆区上分配内存（不过现代技术里，也不是这么绝对的，也有栈上直接分配的）。  
一般的，根据Java虚拟机规范规定，堆内存需要在逻辑上是连续的（在物理上不需要），在实现时，可以是固定大小的，也可以是可扩展的，目前主流的虚拟机都是可扩展的。如果在执行垃圾回收之后，仍没有足够的内存分配，也不能再扩展，将会抛出OutOfMemoryError:Java heap space异常。  
**Java栈：**  
Java栈是线程私有的。每当启动一个新线程时，Java虚拟机都会为它分配一个Java栈。Java栈以帧为单位保存线程的运行状态。虚拟机只会直接对Java栈执行两种操作：**以帧为单位的压栈或出栈**。当线程调用java方法时，虚拟机压入一个新的栈帧到该线程的java栈中。当方法返回时，这个栈帧被从java栈中弹出并抛弃。一个栈帧包含一个java方法的调用状态，它存储有**局部变量表、操作栈、动态链接、方法出口等信息。**

虚拟机栈中定义了两种异常，如果线程调用的栈深度大于虚拟机允许的最大深度，则抛出StatckOverFlowError（栈溢出）；不过多数Java虚拟机都允许动态扩展虚拟机栈的大小(有少部分是固定长度的)，所以线程可以一直申请栈，直到内存不足，此时，会抛出OutOfMemoryError（内存溢出）。  
每个线程对应着一个虚拟机栈，因此虚拟机栈也是线程私有的。  
**程序计数器：**  
一个运行中的Java程序，每当启动一个新线程时，都会为这个新线程创建一个自己的PC(程序计数器)寄存器。程序计数器的作用可以看做是当前线程所执行的字节码的行号指示器。字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令，**分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成**。如果线程正在执行的是一个Java方法，这个计数器记录的是正在执行的虚拟机字节码指令的地址；如果正在执行的是Natvie方法，这个计数器值则为空(Undefined)。

每个程序计数器只用来记录一个线程的行号，所以它是**线程私有**（一个线程就有一个程序计数器）的。  
如果程序执行的是一个Java方法，则计数器记录的是正在执行的虚拟机字节码指令地址；如果正在执行的是一个本地（native，由C语言编写完成）方法，则计数器的值为Undefined，由于程序计数器只是记录当前指令地址，所以不存在内存溢出的情况，因此，程序计数器也是所有JVM内存区域中唯一一个没有定义OutOfMemoryError的区域。  
**本地方法栈：**  
本地方法栈与虚拟机栈所发挥的作用是非常相似的，其区别不过是虚拟机栈为虚拟机执行Java方法（也就是字节码）服务，而本地方法栈则是为虚拟机使用到的Native方法服务。任何本地方法接口都会使用某种本地方法栈。当线程调用Java方法时，虚拟机会创建一个新的栈帧并压入Java栈。然而当它调用的是本地方法时，虚拟机会保持Java栈不变，不再在线程的Java栈中压入新的帧，虚拟机只是简单地动态链接并直接调用指定的本地方法。如果某个虚拟机实现的本地方法接口是使用C连接模型的话，那么它的本地方法栈就是C栈。  
**执行引擎：**  
**负责执行字节码**。方法的字节码是由Java虚拟机的指令序列构成的。每一条指令包含一个单字节的操作码，后面跟随0个或多个操作数。执行引擎执行字节码时，首先取得一个操作码，如果操作码有操作数，取得它的操作数。它执行操作码和跟随的操作数规定的动作，然后再取得下一个操作码。这个执行字节码的过程在线程完成前将一直持续。