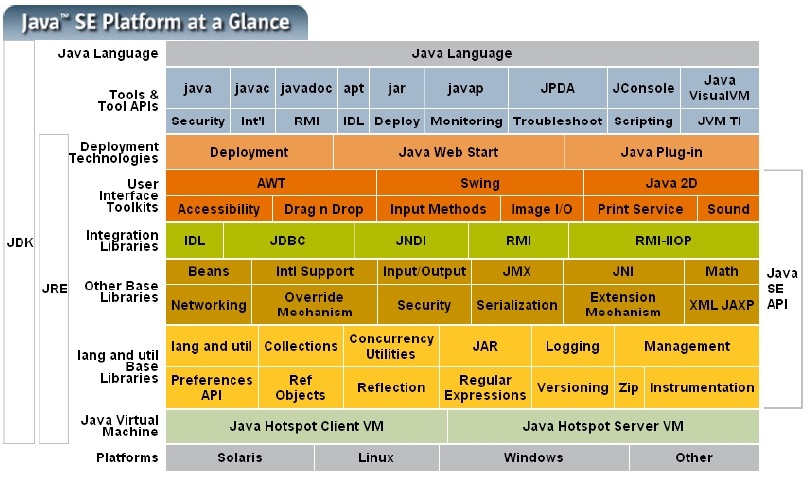
# JVM原理

## 一. 什么是JVM

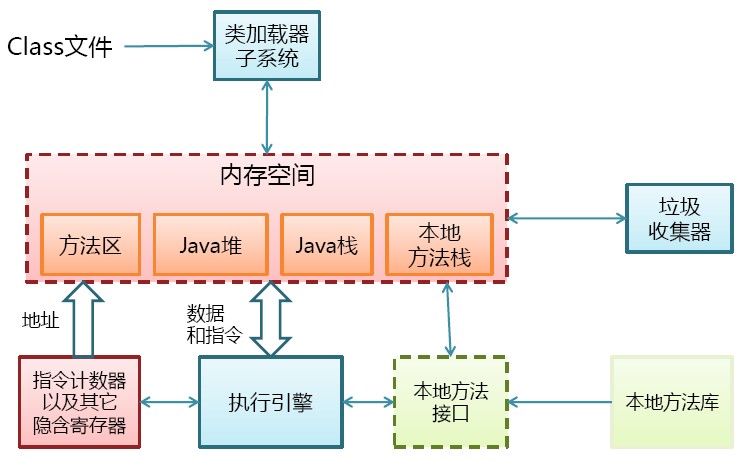
JVM是Java Virtual Machine (Java虚拟机)的缩写, JVM是一种用于计算设备的规范, 它是一个虚构出来的计算机,是通过在实际的计算机上仿真模拟各种计算机功能来实现的.

Java语言的一个非常重要的特点就是与平台的无关性. 而是用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. 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语言编译程序只需生成在Java虚拟机上运行的目标代码(字节码), 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. 这就是Java的能够"一编译,到处运行"的原因.

从Java平台的逻辑结构上来看，我们可以从下图来了解JVM：

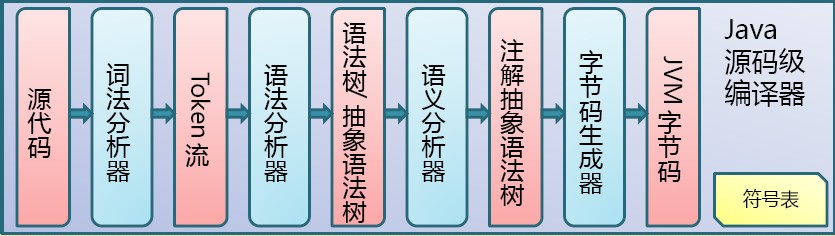


从上图能清晰看到Java平台包含的各个逻辑模块，也能了解到JDK与JRE的区别，对于JVM自身的物理结构，我们可以从下图鸟瞰一下：

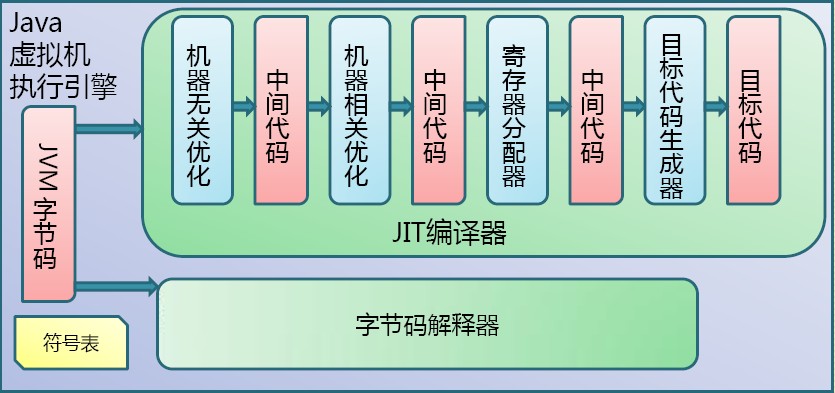


## **二、JAVA代码编译和执行过程**

Java代码编译是由Java源码编译器来完成，流程图如下所示：



Java字节码的执行是由JVM执行引擎来完成，流程图如下所示：



Java代码编译和执行的整个过程包含了以下三个重要的机制：

### Java源码编译机制

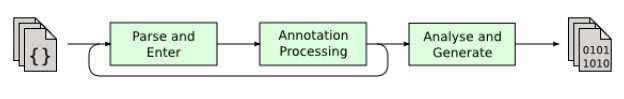
Java源码编译有以下三个过程组成:

1. 分析和输入到符号表

2. 注解处理

3. 语义分析和生成class文件

流程图如下所示：



最后生成的class文件由以下部分组成:

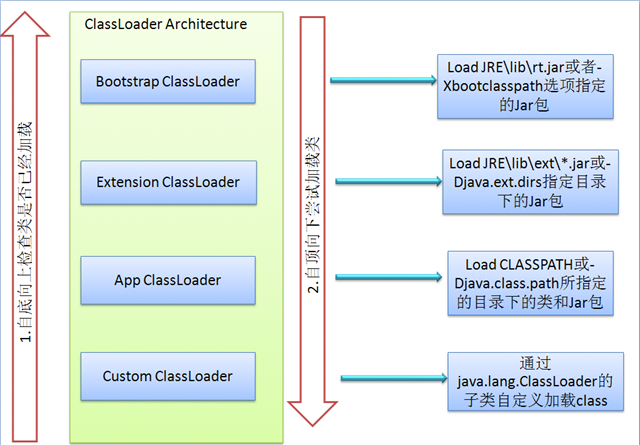
a. 结构信息: 包括class文件格式 版本号 及各部分的数量与大小信息.

b. 元数据: 对应于java源码中声明与常量的信息. 包含类/继承的超类/实现的接口的声明信息,域与方法声明信息和常量池.

c. 方法信息: 对应java源码中语句和表达式对应的信息. 包含字节码,异常处理器表,求值栈与局部变量区大小,求值栈的类型记录,调试符号信息.

### 类加载机制

JVM类加载是通过ClassLoader及其子类来完成的, 类的层次关系和加载顺序可以由下图来描述：



1. Bootstrap ClassLoader

负责加载$JAVA\_HOME中jre/lib/rt.jar里所有的class，由C++实现，不是ClassLoader子类

1. Extension ClassLoader

负责加载java平台中扩展功能的一些jar包，包括$JAVA\_HOME中jre/lib/\*.jar或-Djava.ext.dirs指定目录下的jar包

1. App ClassLoader

负责记载classpath中指定的jar包及目录中class

1. Custom ClassLoader

属于应用程序根据自身需要自定义的ClassLoader，如tomcat、jboss都会根据j2ee规范自行实现ClassLoader加载过程中会先检查类是否被已加载，

检查顺序是自底向上，从Custom ClassLoader到BootStrap ClassLoader逐层检查，只要某个classloader已加载就视为已加载此类，保证此类只

所有ClassLoader加载一次。而加载的顺序是自顶向下，也就是由上层来逐层尝试加载此类。

### 3. 类执行机制

JVM是基于栈的体系结构来执行class字节码的. 线程创建后, 都会产生程序计数器(PC) 和栈 (Stack), 程序计数器存放下一条要执行的指令在方法内的偏移量, 栈中存放一个个

栈帧 ,每个 栈帧对应着每个方法的每次调用, 而栈帧又是由 局部变量区和操作数栈 两部分组成, 局部变量区 用于存放方法中的局部变量和参数, 操作数栈中 用于存放方法执行过程中产生的中间结果. 栈的结构如下图所示：



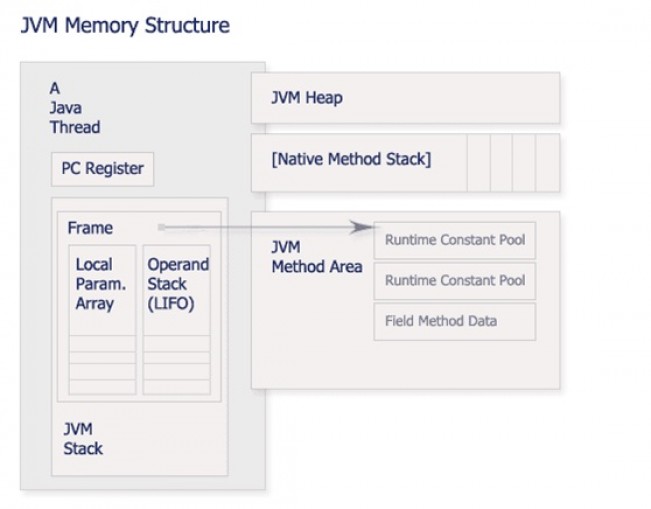
## 三. JVM内存管理和垃圾回收

### 1. 计算机内存

其实对于我们一般理解的计算机内存，它算是CPU与计算机打交道最频繁的区域，所有数据都是先经过硬盘至内存，然后由CPU再从内存中获取数据进行处理，又将数据保存到内存，通过分页或分片技术将内存中的数据再flush至硬盘。那JVM的内存结构到底是如何呢？JVM做为一个运行在操作系统上，但又独立于os运行的平台，它的内存至少应该包括象寄存器、堆栈等区域。

### JVM内存分配

JVM在运行时将数据划分为了6个区域来存储，而不仅仅是大家熟知的Heap区域，这6个区域图示如下：



下面将逐一介绍下各个区域所做的工作及其充当的功能。

#### PC Register(PC寄存器)

PC寄存器是一块很小的内存区域,主要作用是记录当前线程所执行的字节码的行号. 字节码解释器工作时就是通过改变当前线程的程序计数器选取下一条字节码指令来工作的.任何分支,循环,方法调用,判断,异常处理,线程等待以及恢复线程,递归等等都是通过这个计数器来完成的.

由于Java多线程是通过交替线程轮流切换并分配处理器时间的方式来实现的,在任何一个确定的时间里,在处理器的一个内核只会执行一条线程中的指令. 因此为了线程等待结束需要恢复到正确的位置执行, 每条线程都会有一个独立的程序计数器来记录当前指令的行号. 计数器之间相互独立互不影响, 我们称这块内存为 "线程私有" 的内存.

如果所调用的方法为native的, 则PC寄存器中不存储任何信息.

#### JVM Stack 栈

JVM栈是线程私有的, 每个线程创建的同时都会创建JVM栈, JVM栈中存放的为当前线程中局部基本类型的变量(java中定义的八种基本类型: boolean, char, byte, shot, int, long, float, double), 部分的返回结果以及 Stack Frame(栈帧), 非基本类型的对象在JVM栈上仅存放一个指向堆上的地址, 因此Java中基本类型的变量是值传递, 而非基本类型的变量是引用传递, Sun JDK的实现中JVM栈的空间是在物理内存上分配的, 而不是从堆上分配.

由于JVM栈是线程私有的, 因此其在内存分配上非常高效, 并且当线程运行完毕后, 这些内存也就被自动回收.

当JVM栈的空间不足时, 会抛出StackOverflowError的错误, 在Sun JDK中可以通过 -Xss来指定栈的大小, 例如如下代码:



当JVM参数设置为 –Xss1K, 运行后会报出类似下面的错误:

Exception in thread "Thread-0"java.lang.StackOverflowError

#### Heap 堆

Heap是大家最为熟悉的区域, 它是JVM用来存储对象实例以及数组值得区域, 可以认为Java中所有通过new创建的对象的内存都在此分配, Heap中的对象的内存需要等待GC进行回收, Heap在32位操作系统上最大为2G, 在64位操作系统上则没有限制, 其大小可以通过-Xms和-Xmx来控制, -Xms为JVM启动时申请的最小Heap内存, 默认为物理内存的1/64 但是小于1G, -Xmx为JVM可申请的最大Heap内存,默认为物理内存的1/4, 默认当空余堆内存小于40%时, JVM会增大Heap的大小到-Xmx指定的大小, 可通过-XX:MinHeapFreeRatio=来指定这个比例, 当空余内存大于70%时, JVM会将Heap的大小往-Xms指定的大小调整, 可通过 –XX:MaxHeapFreeRatio=来指定这个比例,但对于运行系统而言,为了避免频繁的 Heap Size的大小, 通常都会将-Xms和-Xmx的值设成一样, 因此这两个用于调整比例的参数通常是没用的. 其实JVM中对于 堆内存的 分配、使用、管理、收集 等有更为精巧的设计，具体可以在JVM堆内存分析中进行详细介绍。

当堆中需要使用的内存超过其允许的大小时，会抛出OutOfMemory的错误信息。

#### MethodArea 方法区域

方法区域存放了所加载的类的信息（名称、修饰符等）、类中的静态变量、类中定义为final类型的常量、类中的Field信息、类中的方法信息，当开发人员在程序中通过Class对象中的getName、isInterface等方法来获取信息时，这些数据都来源于方法区域，可见方法区域的重要性。同样，方法区域也是全局共享的，它在虚拟机启动时在一定的条件下它也会被GC，当方法区域需要使用的内存超过其允许的大小时，会抛出OutOfMemory的错误信息。

在Sun JDK中这块区域对应的为PermanetGeneration，又称为持久代，默认为64M，可通过-XX:PermSize以及-XX:MaxPermSize来指定其大小。

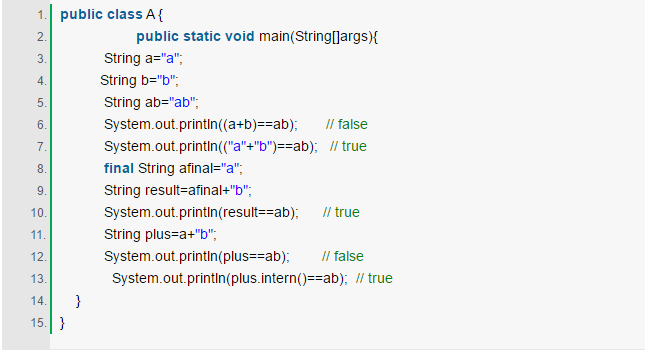
#### RuntimeConstant Pool运行时常量池

类似C中的符号表， 存放的为类中的固定的常量信息，方法和Field的引用信息等， 其空间从方法区域中分配。 类或接口的常量池在该类的class文件被java虚拟机成功装在时分配。

#### NativeMethod Stacks 本地方法栈

JVM采用本地方法堆栈来支持native方法的执行，此区域用于存储每个native方法调用的状态。

例如有这么一段代码：



分析下上面代码执行的结果，可通过javap –verbose A来辅助理解分析。

l  (a+b)==ab

a+b是两个变量相加，需要到运行时才能确定其值，到运行时后JVM会为两者相加后产生一个新的对象，因此a+b==ab的结果为false。

l  (“a”+”b”)==ab

“a”+”b”是常量，在编译时JVM已经将其变为”ab”字符串了，而ab=”ab”也是常量，这两者在常量池即为同一地址，因此(“a”+”b”)==ab为true。

l  result==ab

result=afinal+”b”，afinal是个final的变量， result在编译时也已经被转变为了”ab”，和”ab”在常量池中同样为同一地址，因此result==ab为true。

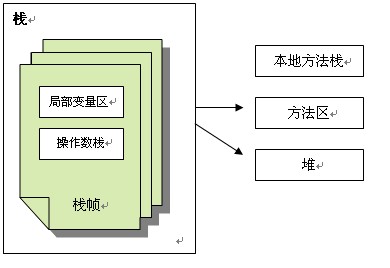
l  plus=ab

plus和a+b的情况是相同的，因此plus==ab为false。

l  plus.intern()==ab

这里的不同点在于调用了plus.intern()方法，这个方法的作用是获取plus指向的常量池地址，因此plus.intern()==ab为true。

在掌握了JVM对象内存分配的机制后，接下来看看JVM是如何做到自动的对象内存回收的，这里指的的是Heap以及Method Area的回收，其他几个区域的回收都由JVM简单的按生命周期来进行管理



### 堆heap

所有通过new创建的对象的内存都在堆中分配, 堆得大小可以通过-Xmx和-Xms来控制. 堆被划分为新生代和旧生代,

新生代又被进一步划分为 Eden和Survivor区, 最后Survivor由 From Space 和 To Space组成.

#### a. 新生代

新建的对象都是用新生代分配内存, Eden空间不足的时候, 会把存活的对象转移到Survivor中,新生代

大小可以由-Xmn来控制, 也可以用-XX:SurvivorRatio来控制 Eden和Survivor的比例

#### b. 旧生代

用于存放新生代中记过多次垃圾回收仍然存活的对象.

c. 持久带(Permanent Space) 实现方法区: 主要存放所有已加载的类信息,方法信息,常量池 等等. 可以通过

-XX:PermSize和-XX:MaxPermSize 来指定持久带初始化值 和 最大值. Permanent Space并不等同于方法区,

只不过是 Hotspot JVM 用Permanent Space来实现方法区而已, 有些虚拟机没有 Permanent Space而用其他

机制来实现方法区.

#### c. 内存配置

-Xmx: 最大堆内存, 如: -Xmx512m

-Xms: 初始时堆内存, 如: -Xms256m

-XX:MaxNewSize :最大年轻区内存

-XX:NewSize :初始时年轻区内存. 通常为 Xmx 的 1/3 或 1/4.

新生代 = Eden + 2个 Survivor 空间. 实际可用空间 为 = Eden + 1个 Survivor, 即 90%

-XX:MaxPermSize : 最大持久带内存

-XX:PermSize :持久带初始内存

-XX:+PrintGCDetails .打印GC信息

-XX:NewRatio 新生代与老年代 的比例 , 如 -XX:NewRatio=2 , 则新生代占整个堆空间的 1/3, 老年代占2/3

-XX:SurvivorRatio 新生代中 Eden 与 Survivor的比值 , 默认值 是 8, 即 Eden 占新生代 空间的 8/10,

另外两个 Survivor 各站1/10.

http://www.mamicode.com/info-detail-1028149.html