（1）[Java内存区域与内存溢出](http://blog.csdn.net/ns_code/article/details/17565503)



**程序计数器（Program Counter Register）**

一块较小的内存空间，它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通过改变该计数器的值来选择下一条需要执行的字节码指令，分支、跳转、循环等基础功能都要依赖它来实现。每条线程都有一个独立的的程序计数器

## 

## Java虚拟机栈

虚拟机栈描述的是Java方法执行的内存模型：每个方法被执行的时候都会同时创建一个栈帧，栈它是用于支持续虚拟机进行方法调用和方法执行的数据结构。栈帧用于存储局部变量表、操作数栈、动态链接、方法返回地址和一些额外的附加信息。在编译程序代码时，栈帧中需要多大的局部变量表、多深的操作数栈都已经完全确定了，并且写入了方法表的Code属性之中

1. 局部变量包

   局部变量表是一组变量值存储空间，用于存放方法参数和方法内部定义的局部变量，其中存放的数据的类型是编译期可知的各种基本数据类型、对象引用（reference）和returnAddress类型（它指向了一条字节码指令的地址）。局部变量表所需的内存空间在编译期间完成分配，即在Java程序被编译成Class文件时，就确定了所需分配的最大局部变量表的容量。当进入一个方法时，这个方法需要在栈中分配多大的局部变量空间是完全确定的，在方法运行期间不会改变局部变量表的大小。

在方法执行时，虚拟机是使用局部变量表来完成参数值到参数变量列表的传递过程的

则局部变量表中的第0位索引的Slot默认是用于传递方法所属对象实例的引用，在方法中可以通过关键字“this”来访问这个隐含的参数。

Java虚拟机的解释执行引擎称为“基于栈的执行引擎”，基于栈的指令集最主要的优点是可移植性强（体会一下）

Java的动态链接是基于引用的，Class文件的常量池中存在有大量的符号引用，字节码中的方法调用指令就以常量池中指向方法的符号引用为参数（通过这个应该有点说明了栈中的内容是什么，内存就可以当做是一个硬盘）

## ****Java堆（Java Heap）****

    Java Heap是Java虚拟机所管理的内存中最大的一块，几乎所有的对象实例和数组都在这类分配内存。

## ****方法区（Method Area）****

 方法区也是各个线程共享的内存区域，它用于存储已经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。

避免了在Java堆和Native堆中来回复制数据（就是通过数据的赋值进行交互的）

（2）[Class类文件结构](http://blog.csdn.net/ns_code/article/details/17675609)

  Code属性是Class文件中最重要的一个属性，如果把一个Java程序中的信息分为代码和元数据两部分，那么在整个Class文件里，Code属性用于描述代码，所有的其他数据项目都用于描述元数据。

(3) [类初始化](http://blog.csdn.net/ns_code/article/details/17845821)

建议通过双亲机制

（4）[类加载机制](http://blog.csdn.net/ns_code/article/details/17881581)

## 加载

  1、通过一个类的全限定名来获取其定义的二进制字节流。

    2、将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构。

3、在Java堆中生成一个代表这个类的java.lang.Class对象，作为对方法区中这些数据的访问入口。

如果虚拟机所有的类加载器都是由java代码完成，这需要在？

JVM自带的ClassLoader只是懂得从本地文件系统加载标准的java class文件

（5）[Javac编译与JIT编译](http://blog.csdn.net/ns_code/article/details/18009455)

Javac编译器称为前端编译器，相对应的还有后端编译器，它在程序运行期间将字节码转变成机器码（现在的Java程序在运行时基本都是解释执行加编译执行），如HotSpot虚拟机自带的JIT（Just In Time Compiler）编译器（分Client端和Server端）。

重点javac的实现原理（之前一直没有搞懂）：

**词法、语法分析**

    词法分析是将源代码的字符流转变为标记（Token）集合。单个字符是程序编写过程中的的最小元素，而标记则是编译过程的最小元素，关键字、变量名、字面量、运算符等都可以成为标记，比如整型标志int由三个字符构成，但是它只是一个标记，不可拆分。

    语法分析是根据Token序列来构造抽象语法树的过程。抽象语法树是一种用来描述程序代码语法结构的树形表示方式，语法树的每一个节点都代表着程序代码中的一个语法结构，如bao、类型、修饰符、运算符等。经过这个步骤后，编译器就基本不会再对源码文件进行操作了，后续的操作都建立在抽象语法树之上。

**填充符号表**

    完成了语法分析和词法分析之后，下一步就是填充符号表的过程。符号表是由一组符号地址和符号信息构成的表格。符号表中所登记的信息在编译的不同阶段都要用到，在语义分析（后面的步骤）中，符号表所登记的内容将用于语义检查和产生中间代码，在目标代码生成阶段，党对符号名进行地址分配时，符号表是地址分配的依据。

**语义分析**

     语法树能表示一个结构正确的源程序的抽象，但无法保证源程序是符合逻辑的。而语义分析的主要任务是读结构上正确的源程序进行上下文有关性质的审查。语义分析过程分为标注检查和数据及控制流分析两个步骤：

* 标注检查步骤检查的内容包括诸如变量使用前是否已被声明、变量和赋值之间的数据类型是否匹配等。
* 数据及控制流分析是对程序上下文逻辑更进一步的验证，它可以检查出诸如程序局部变量在使用前是否有赋值、方法的每条路径是否都有返回值、是否所有的受查异常都被正确处理了等问题。

**字节码生成**

字节码生成是Javac编译过程的最后一个阶段。字节码生成阶段不仅仅是把前面各个步骤所生成的信息转化成字节码写到磁盘中，编译器还进行了少量的代码添加和转换工作。 实例构造器<init>（）方法和类构造器<clinit>（）方法就是在这个阶段添加到语法树之中的。（编译器是可以修改源代码的，classloader【类加载】也可以就行修改，这就是进行aop的基础）。虚拟机（如Sun HotSpot、IBM J9）中几乎都同时包含解释器和编译器