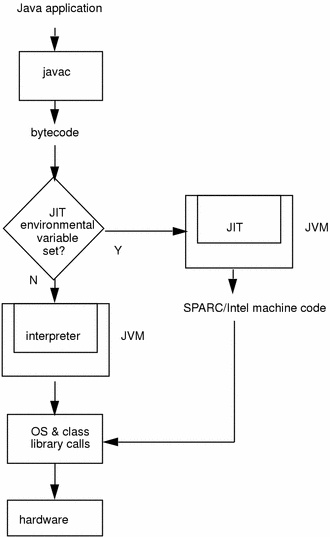
Heap : 先进先出

Heap是用来存放对象信息的，和Stack不同，Stack代表着一种运行时的状态。换句话说，栈是运行时单位，解决程序该如何执行的问题，而堆是存储的单位，解决数据存储的问题。Heap是伴随着JVM的启动而创建，负责存储所有对象实例和数组的。堆的存储空间和栈一样是不需要连续的，它分为Young Generation和Old Generation（也叫Tenured Generation）两大部分。Young Generation分为Eden和Survivor，Survivor又分为From Space和 ToSpace。

和Heap经常一起提及的概念是PermanentSpace，它是用来加载类对象的专门的内存区，是非堆内存，和Heap一起组成JAVA内存，它包含MethodArea区（在没有CodeCache的HotSpotJVM实现里，则MethodArea就相当于GenerationSpace）。在JVM初始化的时候，我们可以通过参数来分别指定，PermanentSpace的大小、堆的大小、以及Young Generation和Old Generation的比值、Eden区和From Space的比值，从而来细粒度的适应不同JAVA应用的内存需求。

2.4.3.PC Register是程序计数寄存器，每个JAVA线程都有一个单独的PC Register，他是一个指针，由Execution Engine读取下一条指令。如果该线程正在执行java方法，则PC Register存储的是 正在被执行的指令的地址，如果是本地方法，PC Register的值没有定义。PC寄存器非常小，只占用一个字宽，可以持有一个returnAdress或者特定平台的一个指针。

2.4.4.Method Area在HotSpot JVM的实现中属于非堆区，非堆区包括两部分：Permanet Generation和Code Cache，而Method Area属于Permanert Generation的一部分。Permanent Generation用来存储类信息，比如说：class definitions，structures，methods， field， method (data and code) 和 constants。Code Cache用来存储Compiled Code，即编译好的本地代码，在HotSpot JVM中通过JIT(Just In Time) Compiler生成，JIT是即时编译器，他是为了提高指令的执行效率，把字节码文件编译成本地机器代码，如下图：



引用一个经典的案例来理解Stack，Heap和Method Area的划分，就是Sring a=”xx”；Stirng b=”xx”，问是否a==b? 首先==符号是用来判断两个对象的引用地址是否相同，而在上面的题目中，a和b按理来说申请的是Stack中不同的地址，但是他们指向Method Area中Runtime Constant Pool的同一个地址，按照网上的解释，在a赋值为“xx”时，会在Runtime Contant Pool中生成一个String Constant，当b也赋值为“xx”时，那么会在常量池中查看是否存在值为“xx”的常量，存在的话，则把b的指针也指向“xx”的地址，而不是新生成一个String Constant。我查阅了网络上大家关于String Constant的存储的说说法，存在略微差别的是，它存储在哪里，有人说Heap中会分配出一个常量池，用来存储常量，所有线程共享它。而有人说常量池是Method Area的一部分，而Method Area属于非堆内存，那怎么能说常量池存在于堆中？

我认为，其实两种理解都没错。Method Area的确从逻辑上讲可以是Heap的一部分，在某些JVM实现里从堆上开辟一块存储空间来记录常量是符合JVM常量池设计目的的，所以前一种说法没问题。对于后一种说法，HotSpot JVM的实现中的确是把方法区划分为了非堆内存，意思就是它不在堆上。我在HotSpot JVM做了个简单的实验，定义多个常量之后，程序抛出OOM：PermGen Space异常，印证了JVM实现中常量池是在Permanent Space中的说法。但是，我的JDK版本是1.6的。查阅资料，JDK1.7中InternedStrings已经不再存储在PermanentSpace中，而是放到了Heap中；JDK8中PermanentSpace已经被完全移除，InternedStrings也被放到了MetaSpace中（如果出现内存溢出，会报OOM:MetaSpace，这里有个关于两者性能对比的文章：<http://blog.csdn.net/zhyhang/article/details/17246223> ）。 所以，仁者见仁，智者见智，一个馒头足以引发血案，就算是同一个商家的JVM，毕竟JDK版本在更新，或许正如StackOverFlow上大神们所说，对于理解JVM Runtime Data Area这一部分的划分逻辑，还是去看对应版本的JDK源码比较靠谱，或者是参考不同的版本JVM Specification（ <http://docs.oracle.com/javase/specs/> ）。

2.4.5.Native Method Stack是供本地方法（非java）使用的栈。每个线程持有一个Native Method Stack。

分配堆内存：

堆的大小通过-Xms和-Xmx来指定最小值和最大值，通过-Xmn来指定Young Generation的大小（一些老版本也用-XX:NewSize指定）， 即上图中的Eden加FromSpace和ToSpace的总大小。然后通过-XX:NewRatio来指定Eden区的大小，在Xms和Xmx相等的情况下，该参数不需要设置。通过-XX：SurvivorRatio来设置Eden和一个Survivor区的比值

堆的错误:

堆异常分为两种，一种是Out of Memory(OOM)，一种是Memory Leak(ML)。Memory Leak最终将导致OOM。实际应用中表现为：从Console看，内存监控曲线一直在顶部，程序响应慢，从线程看，大部分的线程在进行GC，占用比较多的CPU，最终程序异常终止，报OOM。OOM发生的时间不定，有短的一个小时，有长的10天一个月的。关于异常的处理，确定OOM/ML异常后，一定要注意保护现场，可以dump heap，如果没有现场则开启GCFlag收集垃圾回收日志，然后进行分析，确定问题所在。如果问题不是ML的话，一般通过增加Heap，增加物理内存来解决问题，是的话，就修改程序逻辑。