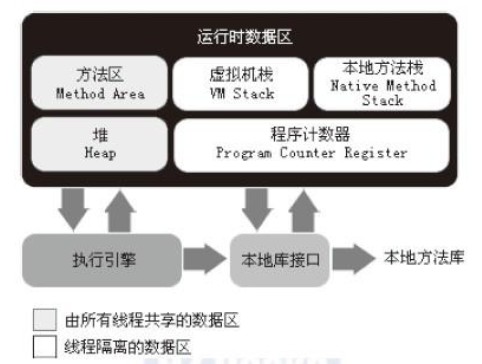
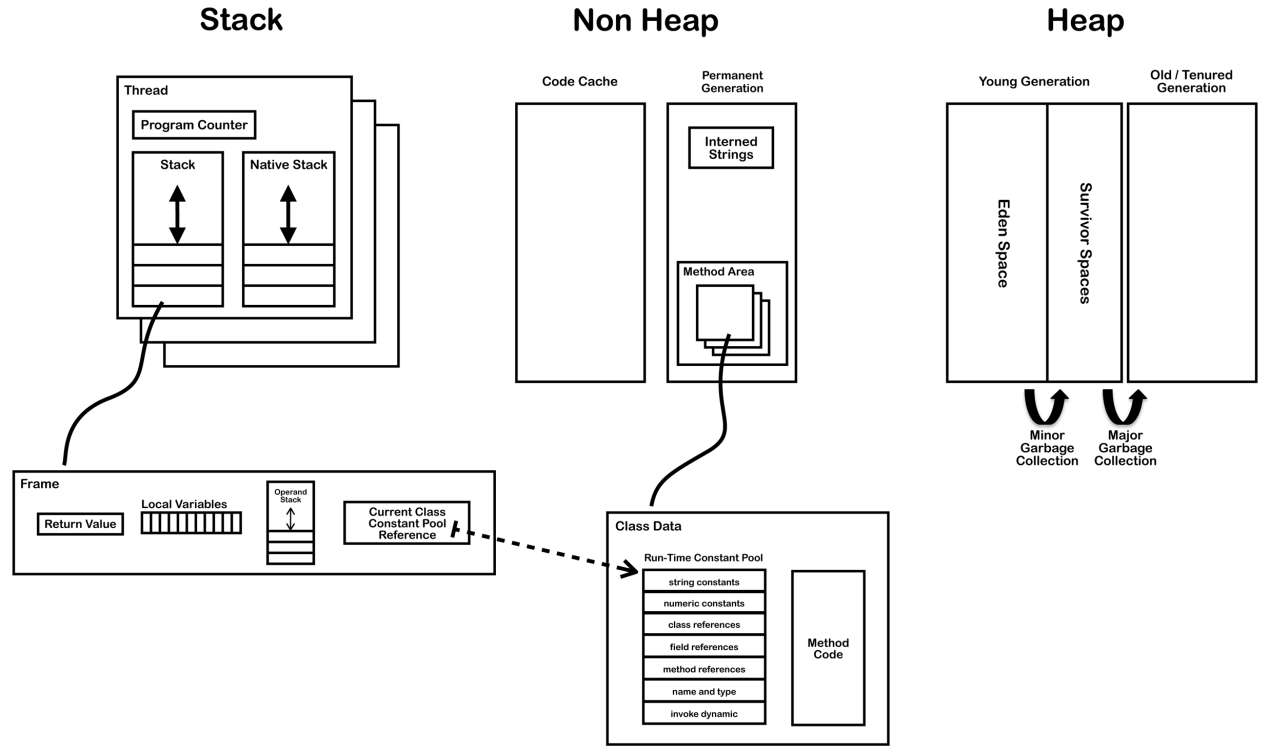
# 1 Java 内存区域管理

## 1.1 Java运行时区域





## 1.2 [从JVM角度理解线程](http://blog.csdn.net/iter_zc/article/details/41843595)

Thread

JVM允许一个应用程序被多个线程并发执行，Hotspot JVM中Java线程和操作系统本地线程之间有个直接的映射，在为线程局部存储，分配缓冲区，同步对象，堆栈和程序计数器等Java线程准备好所有状态之后，本地线程随之被创建，本地线程随着Java线程终止而结束，操作系统负责调度所有的线程被分配可用的CPU，一旦本地线程初始化，它调用Java线程中的run（）方法，当run方法返回时且没有捕获到任何异常, 那么本地线程确认JVM是否需要作为线程终止的结果而被终止（即，它是最后一个非Deamon线程）。当线程终止时，本地和Java线程的所有资源都被释放。

有两种方式可以让用户在JVM中创建线程

1. new java.lang.Thread().start()

2. 使用JNI将一个native thread attach到JVM中

针对 new java.lang.Thread().start()这种方式，只有调用start()方法的时候，才会真正的在JVM中去创建线程，主要的生命周期步骤有：

1. 创建对应的JavaThread的instance

2. 创建对应的OSThread的instance

3. 创建实际的底层操作系统的native thread

4. 准备相应的JVM状态，比如ThreadLocal存储空间分配等

5. 底层的native thread开始运行，调用java.lang.Thread生成的Object的run()方法

6. 当java.lang.Thread生成的Object的run()方法执行完毕返回后,或者抛出异常终止后，终止native thread

7. 释放JVM相关的thread的资源，清除对应的JavaThread和OSThread

JVM System Thread

通过查看线程堆栈，可以看到有许多线程在后台运行，在Hotspot Jvm主要有以下后台线程：

|  |  |
| --- | --- |
| VM Thread | 这个线程就比较牛b了，是jvm里面的线程母体，根据hotspot源码（vmThread.hpp）里面的注释，它是一个单例的对象（最原始的线程）会产生或触发所有其他的线程，这个单个的VM线程是会被其他线程所使用来做一些VM操作（如，清扫垃圾等）。在 VMThread的结构体里有一个VMOperationQueue列队，所有的VM线程操作(vm\_operation)都会被保存到这个列队当中，VMThread本身就是一个线程，它的线程负责执行一个自轮询的loop函数(具体可以参考：VMThread.cpp里面的void VMThread::loop())，该loop函数从VMOperationQueue列队中按照优先级取出当前需要执行的操作对象(VM\_Operation)，并且调用VM\_Operation->evaluate函数去执行该操作类型本身的业务逻辑。ps：VM操作类型被定义在vm\_operations.hpp文件内，列举几个：ThreadStop、ThreadDump、PrintThreads、GenCollectFull、GenCollectFullConcurrent、CMS\_Initial\_Mark、CMS\_Final\_Remark |
| VM Periodic Task Thread | 该线程是JVM周期性任务调度的线程，它由WatcherThread创建，是一个单例对象。该线程在JVM内使用得比较频繁，比如：定期的内存监控、JVM运行状况监控，还有我们经常需要去执行一些jstat这类命令查看gc的情况，如下：jstat -gcutil 23483 250 7  这个命令告诉jvm在控制台打印PID为：23483的gc情况，间隔250毫秒打印一次，一共打印7次 |
| CompilerThread0 | 用来调用JITing，实时编译装卸class。通常，jvm会启动多个线程来处理这部分工作，线程名称后面的数字也会累加，例如：CompilerThread1 |
| Attach Listener | Attach Listener线程是负责接收到外部的命令，而对该命令进行执行的并且吧结果返回给发送者。通常我们会用一些命令去要求jvm给我们一些反馈信息，如：java -version、jmap、jstack等等。如果该线程在jvm启动的时候没有初始化，那么，则会在用户第一次执行jvm命令时，得到启动。 |
| Signal Dispatcher | 前面我们提到第一个Attach Listener线程的职责是接收外部jvm命令，当命令接收成功后，会交给signal dispather线程去进行分发到各个不同的模块处理命令，并且返回处理结果。signal dispather线程也是在第一次接收外部jvm命令时，进行初始化工作。 |
| Low MemoryDetector | 这个线程是负责对可使用内存进行检测，如果发现可用内存低，分配新的内存空间。 |
| GC Daemon | GC Daemon线程是JVM为RMI提供远程分布式GC使用的，GC Daemon线程里面会主动调用System.gc()方法，对服务器进行Full GC。 其初衷是当RMI服务器返回一个对象到其客户机（远程方法的调用方）时，其跟踪远程对象在客户机中的使用。当再没有更多的对客户机上远程对象的引用时，或者如果引用的“租借”过期并且没有更新，服务器将垃圾回收远程对象。不过，我们现在jvm启动参数都加上了-XX:+DisableExplicitGC配置，所以，这个线程只有打酱油的份了 |
| Finalizer | 这个线程也是在main线程之后创建的，其优先级为10，主要用于在垃圾收集前，调用对象的finalize()方法；关于Finalizer线程的几点：1)只有当开始一轮垃圾收集时，才会开始调用finalize()方法；因此并不是所有对象的finalize()方法都会被执行；2)该线程也是daemon线程，因此如果虚拟机中没有其他非daemon线程，不管该线程有没有执行完finalize()方法，JVM也会退出；3) JVM在垃圾收集时会将失去引用的对象包装成Finalizer对象（Reference的实现），并放入ReferenceQueue，由Finalizer线程来处理；最后将该Finalizer对象的引用置为null，由垃圾收集器来回收；4) JVM为什么要单独用一个线程来执行finalize()方法呢？如果JVM的垃圾收集线程自己来做，很有可能由于在finalize()方法中误操作导致GC线程停止或不可控，这对GC线程来说是一种灾难； |
| Reference Handler | JVM在创建main线程后就创建Reference Handler线程，其优先级最高，为10，它主要用于处理引用对象本身（软引用、弱引用、虚引用）的垃圾回收问题。 |
| ConcurrentMark-SweepGCThread | 并发标记清除垃圾回收器（就是通常所说的CMS GC）线程，该线程主要针对于老年代垃圾回收。ps：启用该垃圾回收器，需要在jvm启动参数中加上：-XX:+UseConcMarkSweepGC |

## 1.3 内存区域分析

<http://www.journaldev.com/4098/java-heap-space-vs-stack-memory>

<https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/garbage_collect.html>

<http://www.cnblogs.com/editice/p/5420716.html>

<http://blog.jamesdbloom.com/JVMInternals.html>

**栈空间**：

每个线程都有自己的栈空间，为该线程执行上的每个方法都会创建一个栈帧，该空间是一个后进先出的数据结构，因此当前执行方法再栈的顶部。每个方法调用的时候会新建一个栈帧然后放到栈顶部，然后做入栈出栈操作。当方法正常返回或者在方法调用执行期间抛出了未捕获异常时栈帧会被移除。栈帧主要存储的是方法返回值、局部变量表、操作数栈、当前类常量池引用、动态链接。

栈空间存在两个限制：

1)超过栈的最大深度会抛出StackOverflowError

2)如果一个线程需要一个新的帧，没有足够的内存来分配它，那么抛出一个OutOfMemoryError

栈帧： 1) 局部变量表：this引用、方法入参、局部变量。类文件结构中方法代码有个标示max\_locals

2) 操作数栈：后进先出的数据结构

**堆内存**：每个new操作的Java对象、数组都在堆进行分配。当然也并非绝对，因为逃逸分析技术。堆内存分为新生代和老年代，再细分有Eden、From Survivor、To Survivor等空间，堆里面可以为每个线程划分一个私有空间线程缓冲区（TLAB）

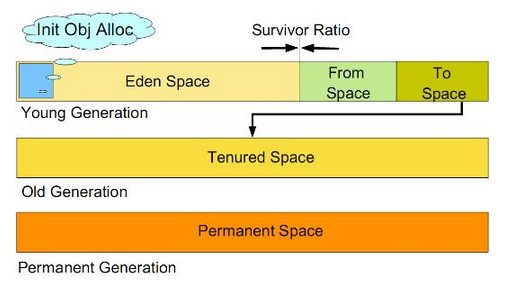
**非堆内存**：主要包括方法区、字符串区、代码缓存区（ 用于存放被JIT编译器编译为本地代码的方法）。

方法区存储运行时常量池（数字常量、字段引用、方法引用）、静态变量、类信息、即时编译后代码数据（字段数据、方法数据、方法代码数据）。

运行时常量池是类文件常量池的运行时表示，它包含从编译时期已知的数字常量、以及运行时被解析的方法和字段引用，它的内存分配是从方法区。

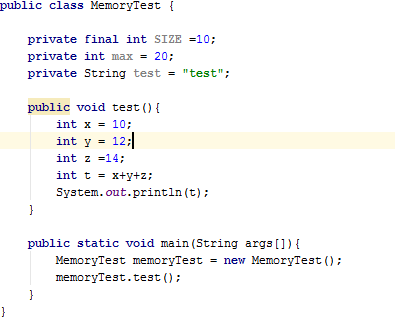
**直接内存**：

**栈VS堆，他们之间的区别不同之处**：

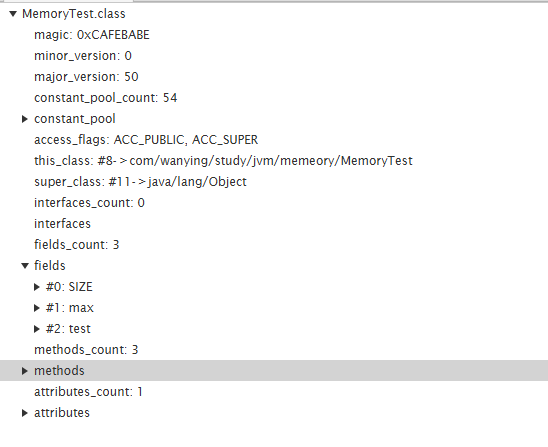


## 1.3 类文件结构

原始代码



字节码文件解析



## 1.4 OutofMemoryError

# 2 垃圾收集器与内存分配策略

# 3 JVM监控工具

# 4 JVM调优与性能分析

# 5 类文件结构

# 6 类加载

# 7 类加载案例分析

# 8 代码编译与优化

# 9 参考文献

JVM 系统线程梳理

<http://ifeve.com/jvm-thread/>

JVM内存管理

<http://blog.jamesdbloom.com/JVMInternals.html>

JVM 虚拟操作参数

<http://www.oracle.com/technetwork/articles/java/vmoptions-jsp-140102.html>

JVM日志安全点

<http://ifeve.com/logging-stop-the-world-pauses-in-jvm/>

JVM 博客

<http://lovestblog.cn/>

类文件查看工具

<https://www.codeproject.com/articles/35915/java-class-viewer>

<https://github.com/zxh0/classpy>