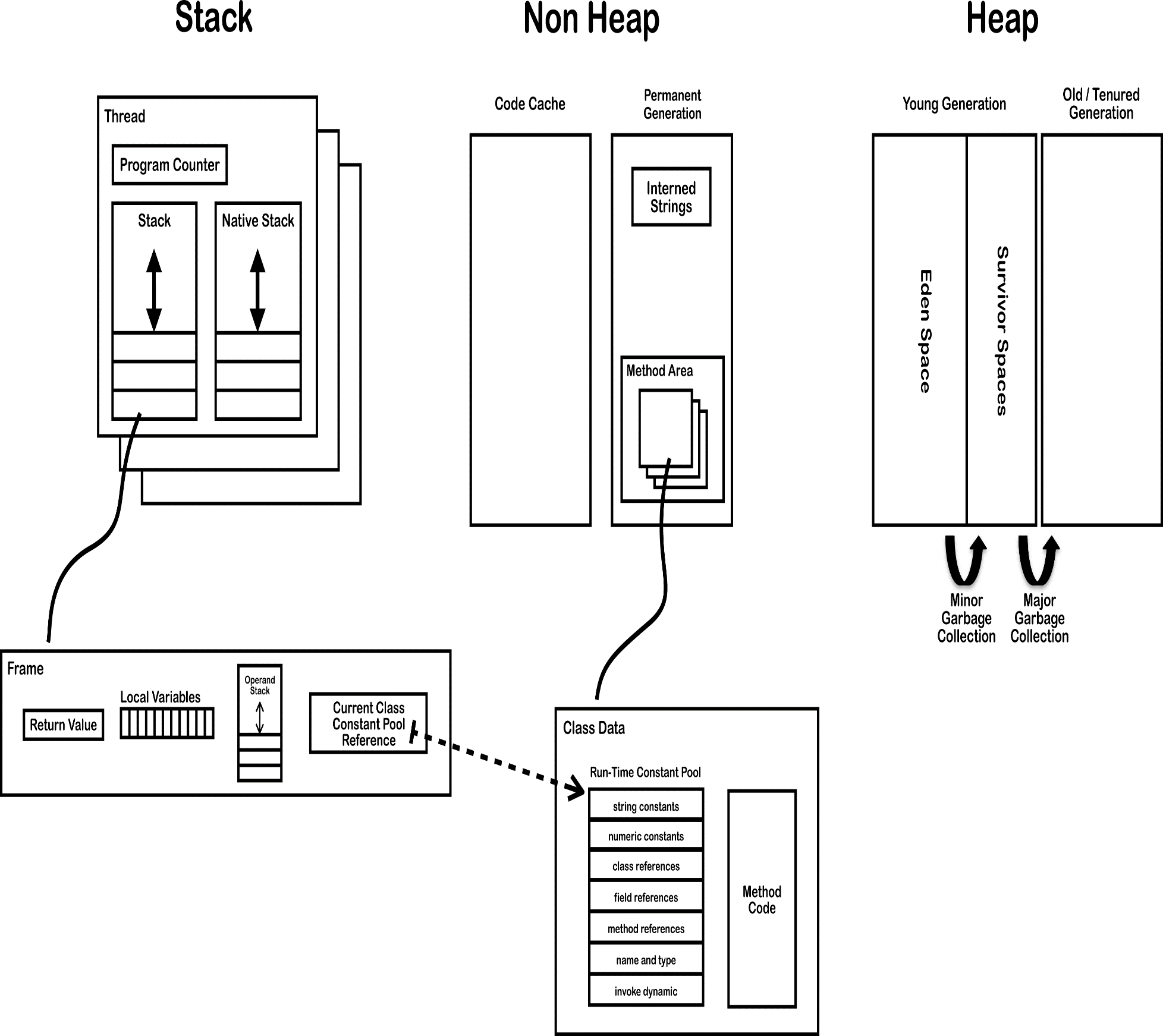
# 1 Java 内存区域管理

## 1.1 Java运行时区域



**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

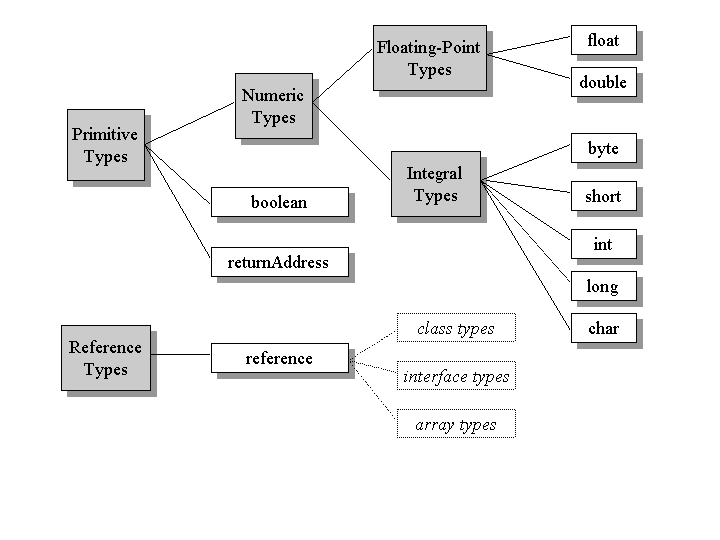
2) 操作数栈：后进先出的数据结构

**堆内存**：每个new操作的Java对象（实例数据）、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）

**方法区**：JVM所有线程共享区域，该区域主要存储运行时常量池、类变量、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）

方法代码数据：方法名称、返回值、数字和类型参数、方法的修饰符

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



## 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 类文件结构

javap -v -p -s -sysinfo –constants MemoryAllocateDemo.class > MemoryAllocateClassFile.txt

ClassFile {

u4 magic; //魔数

u2 minor\_version; //次版本号

u2 major\_version; //主版本号

u2 constant\_pool\_count; //常量池数量

cp\_info constant\_pool[constant\_pool\_count-1]; //常量池

u2 access\_flags; //

u2 this\_class;

u2 super\_class;

u2 interfaces\_count;

u2 interfaces[interfaces\_count];

u2 fields\_count;

field\_info fields[fields\_count];

u2 methods\_count;

method\_info methods[methods\_count];

u2 attributes\_count;

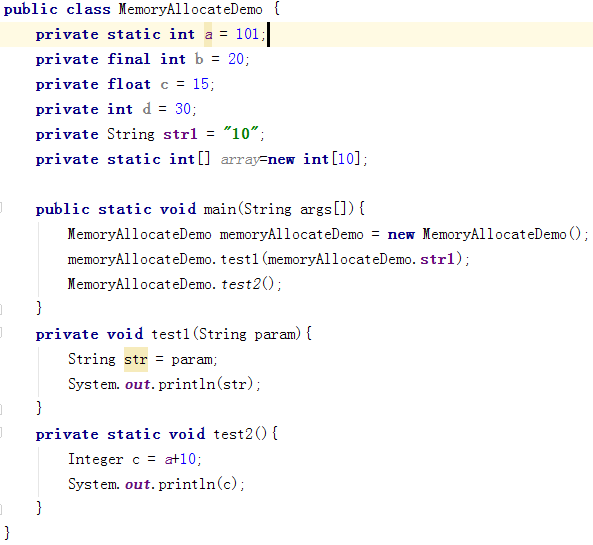
attribute\_info attributes[attributes\_count];

}

Class文件结构中的常量池主要包括字面量和符号引用，字面量：字符常量、final常量或者属性的初始值,符号引用：类和接口的全限定名、字段的名称和描述符、方法的名称和描述符

Cp\_info



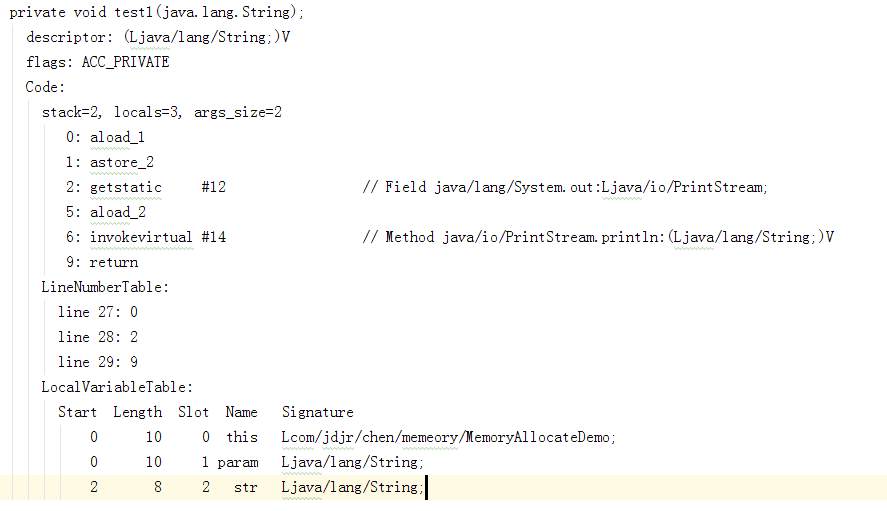




1. 为什么普通的变量值也会出现在常量池？
2. 为什么没有int类型的成员变量值？

<http://stackoverflow.com/questions/42265265/can-not-see-integer-value-in-constant-pool/42265442#42265442>

方法代码1的字节码指令



aload\_0加载一个对象引用到操作数栈，<n>指向局部变量集合中被访问的地址，但是值只能是0,1,2或者3。其他类似的操作符用于加载非对象引用，如iload\_ <n>，lload\_ <n>，fload\_<n> 和dload\_<n>，其中i是int类型，l是long类型，f是float类型，d是double类型。

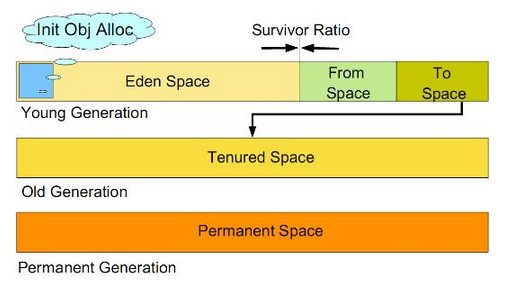
getstatic用于将一个在运行时常量池里的静态值从静态字段列表推入到操作数栈

invokespecial, 用于调用实例初始化方法，以及private方法，父类的方法。

Invokevirtual用于调用一个基于对象的类方法

## 1.4 OutofMemoryError

# 2 垃圾收集

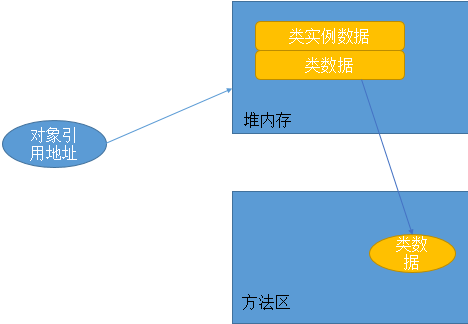


虚拟机在堆内存中进行分配的时候，有两种方式：

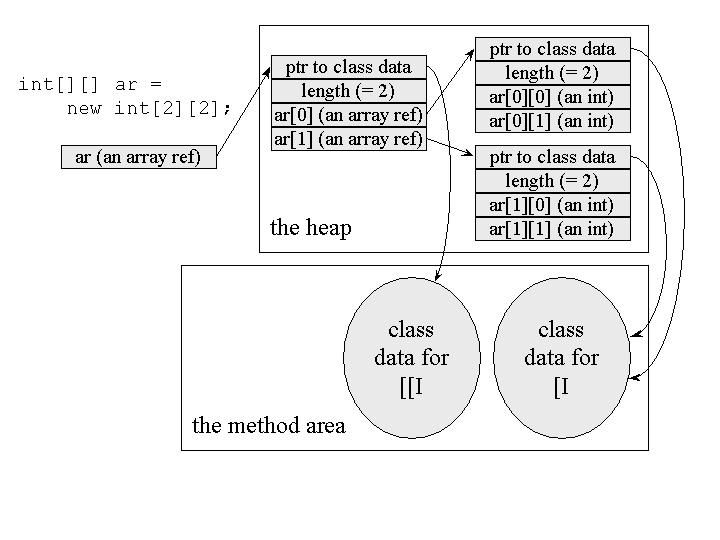
* **指针碰撞** ：用了的内存放在一边，那么用了的内存和没用的内存中间会有一个指针，然后分配内存的时候直接移动该指针即可。
* **空闲列表** ：用的内存和没用的内存交错着，这个时候需要虚拟机维护一个空闲列表，然后分配内存的时候就从空闲列表中选取一个内存区域进行分配，最后更新列表。

针对指针碰撞线程分配带来的安全问题：1）利用CAS 2）tlab

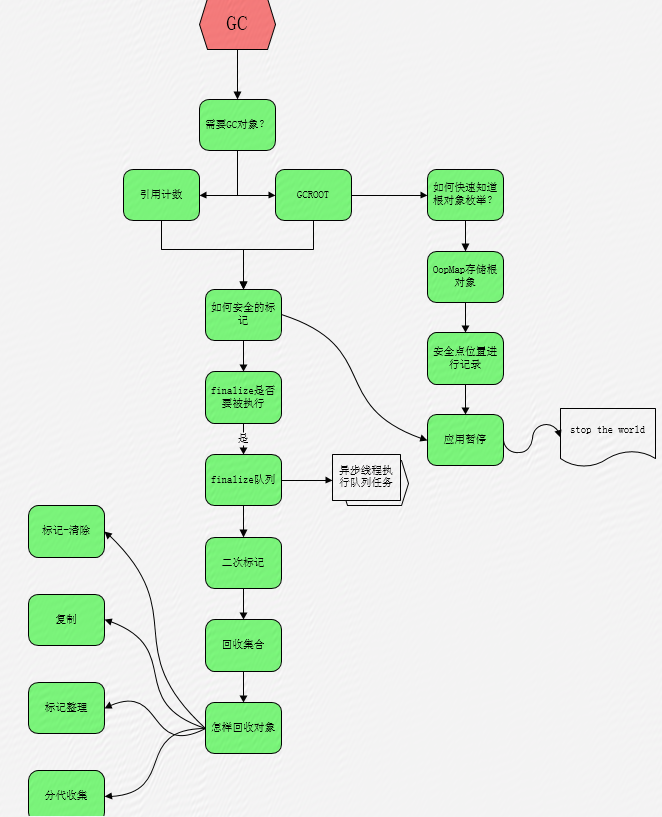
## 2.1 对象内存分配



## 2.2 数组分配

****

## 2.3 GC流程



1、对象的引用关系是实时变化的，在采用算法进行判断引用是否失效增加难度，何时进行什么位置进行？

2、哪些可以成为根对象？根对象怎样进行枚举？

枚举OopMap,在安全点进行记录不是所有指令地方都记录。

可达性分析：" GC Roots" 的对象作为起始点, 从这些节点开始向下搜索走过的路径为引用链，当一个对象到GCROOT没有任何引用链就称这个对象不可达

引用类型：强引用、软引用、弱引用、虚引用。软引用描述有用非必须的对象，在内存溢出之前进行标记，第二次GC的时候进行回收。弱引用描述的是非必须的对象，在进行GC的时候不管内存是否足够都会进行回收掉弱引用关联的对象

## 2.4 GC算法

**标记-清除**：标记需要清理的对象，然后进行内存清理。缺点是多次GC之后内存不连续，需要进行碎片化整理，若一次分配的连续内存空间不足则会触发GC。

**复制算法**：将内存分配为2块同等大小的内存。当进行GC的时候把存活对象放到另外的空间，然后对标记的对象做一次清理。缺点是使内存减半，可以做相关改进就是调整这两块内存比例,hotspot默认是 8:1（eden:Survivor），因为新生代对象生存周期短大多数对象都需要回收，但是若存活对象太多Survivor空间不足时则需要有担保机制保证存活对象的分配。

**标记整理**：标记阶段和标记清除阶段一样，只是在清除不是再是对对象直接清理而是将让存活的对象向一端移动让存活对象尽量在一起，然后就可以清理端界以外的对象，避免碎片整理。

**分代收集：**根据对象生命周期将内存划分不同的区域块，新生代和老年代，新生代每次垃圾收集都有大量对象死去则采用复制垃圾收集算法，而老年代存活率高而且没有担保则采用标记整理或者标记清除算法。

## 2.5 安全点

1、什么是安全点？安全点位置选择？安全点的应用场景？

线程的一些状态可以被确定，比如记录OopMap的状态，从而确定GC Root的信息，使JVM可以安全的进行一些操作，比如开始GC

* + 循环的末尾 (防止大循环的时候一直不进入safepoint，而其他线程在等待它进入safepoint)
  + 方法返回前
  + 调用方法的call之后
  + 抛出异常的位置

2、GC时如何让线程跑到最近的安全点？

主动式中断:直接中断所有运行的线程

被动式中断:以通知的形式，设置一个标志位每个线程运行时主动轮询进行判断。前提条件是该该线程运行

1. 所有线程都没有达到安全点则应用无法进行GC
2. 安全点如何工作？

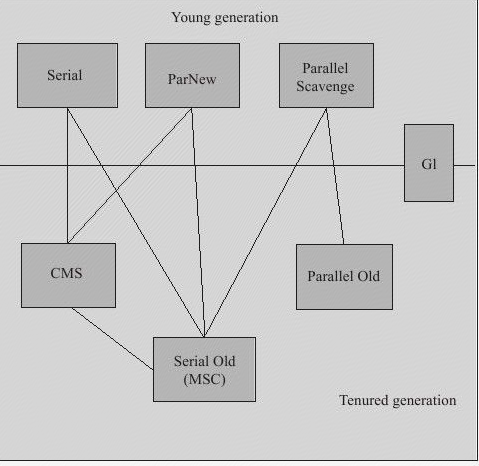
<https://www.kernel.org/doc/Documentation/memory-barriers.txt>

<http://hg.openjdk.java.net/jdk7u/jdk7u/hotspot/file/tip/src/share/vm/runtime/safepoint.cpp>

5、参数

*-XX:+PrintSafepointStatistics -XX:PrintSafepointStatisticsCount=1*

## 2.6 垃圾收集器



Serial 单线程收集器，可以与CMS收集器配合使用，客户端模式运行下采用的收集器

ParNew 多线程收集器，基于Serial来实现

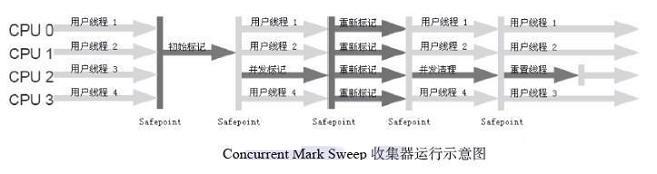
Parallel Scavenge 新生代并行收集器，采用复制算法，该收集器设计的目标是追求吞吐量，这个吞吐量指CPU运行用户代码的时间/(运行用户代码时间+垃圾收集时间)的比值。

Serial old 单线程收集器，老年代版本，采用标记-整理算法

Parallel old 多线程收集器，老年代版本只能配合Parallel Scavenge收集器，采用标记整理算法。1.6才开始提供。

CMS垃圾收集器：是以获取最短停顿时间为目标的收集器，采用标记清除算法。主要有初始标记、并发标记、重新标记、并发清楚四个阶段，初始标记、重新标记还是需要应用停顿，重新标记是修正并发标记期间因

用户线程程序运行而导致标记发生变动。



缺点：1、虽然并发标记和用户线程并发运行但是占用了系统资源导致用户线程运行响应时间增加

2、并发清理阶段由于用户线程在运行可能会产生新的对象这部分是不会被立即回收的，称为浮动垃圾，同时不能等老年代满了再运行GC必须提前预留新产生对象所需内存

3、标记清除算法本身带来的空间碎片问题，会提前触发GC

**并行收集器相关参数**

|  |  |  |  |
| --- | --- | --- | --- |
| -XX:+UseParallelGC | Full GC采用parallel MSC (此项待验证) |  | 选择垃圾收集器为并行收集器.此配置仅对年轻代有效.即上述配置下,年轻代使用并发收集,而年老代仍旧使用串行收集.(此项待验证) |
| -XX:+UseParNewGC | 设置年轻代为并行收集 |  | 可与CMS收集同时使用 JDK5.0以上,JVM会根据系统配置自行设置,所以无需再设置此值 |
| -XX:ParallelGCThreads | 并行收集器的线程数 |  | 此值最好配置与处理器数目相等 同样适用于CMS |
| -XX:+UseParallelOldGC | 年老代垃圾收集方式为并行收集(Parallel Compacting) |  | 这个是JAVA 6出现的参数选项 |
| -XX:MaxGCPauseMillis | 每次年轻代垃圾回收的最长时间(最大暂停时间) |  | 如果无法满足此时间,JVM会自动调整年轻代大小,以满足此值. |
| -XX:+UseAdaptiveSizePolicy | 自动选择年轻代区大小和相应的Survivor区比例 |  | 设置此选项后,并行收集器会自动选择年轻代区大小和相应的Survivor区比例,以达到目标系统规定的最低相应时间或者收集频率等,此值建议使用并行收集器时,一直打开. |
| -XX:GCTimeRatio | 设置垃圾回收时间占程序运行时间的百分比 |  | 公式为1/(1+n) |
| -XX:+*ScavengeBeforeFullGC* | Full GC前调用YGC | true | Do young generation GC prior to a full GC. (Introduced in 1.4.1.) |

## 2.7 案例演示

### 2.7.1 GC等待时间过长

查看停顿时间JVM参数

*-XX:+PrintGCApplicationStoppedTime*

1、进入安全点耗时，通过JVM参数查看达到安全点的停顿时间

Total time for which application threads were stopped: 0.0044595 seconds

Total time for which application threads were stopped: 0.0046200 seconds

Total time for which application threads were stopped: 0.0045507 seconds

Total time for which application threads were stopped: 0.0043442 seconds

Total time for which application threads were stopped: 19.9720769 seconds

Total time for which application threads were stopped: 26.8125418 seconds

Total time for which application threads were stopped: 26.6647086 seconds

vmop [threads: total initially\_running wait\_to\_block] [time: spin block sync cleanup vmop] page\_trap\_count

42.949: ParallelGCFailedAllocation [ 10 0 1 ] [ 0 0 0 0 0 ] 0

Total time for which application threads were stopped: 0.0003026 seconds

vmop [threads: total initially\_running wait\_to\_block] [time: spin block sync cleanup vmop] page\_trap\_count

42.949: ParallelGCFailedAllocation [ 10 0 1 ] [ 0 0 0 0 6 ] 0

11.212: ParallelGCFailedAllocation [ 10 1 1 ] [ 4926 0 4926 0 5 ] 1

total: 安全点里的总线程数

initially\_running: 安全点时开始时正在运行状态的线程数

wait\_to\_block: 在VM Operation开始前需要等待其暂停的线程数

spin: 等待线程响应safepoint号召的时间

block: 暂停所有线程所用的时间

sync: 等于 spin+block，这是从开始到进入安全点所耗的时间，可用于判断进入安全点耗时

cleanup: 清理所用时间

vmop: 真正执行VM Operation的时间

# 3 JVM调优与性能分析

# 4 类加载

# 5 参考文献

JVM 类文件结构

<http://wiki.jikexueyuan.com/project/java-vm/class.html>

<http://coolshell.cn/articles/9229.html>

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>

其他资料

<https://yq.aliyun.com/articles/49216>

<http://www.infoq.com/cn/articles/jvm-memory-collection>

<http://www.infoq.com/cn/articles/jvm-hotspot>

<https://github.com/gvsmirnov/java-perv/tree/master/labs-8/src/main/java/ru/gvsmirnov/perv/labs/gc>

性能优化

<http://calvin1978.blogcn.com/articles/jvmoption-2.html>