自动内存管理机制

作者：skyesx
原文链接：https://ld246.com/article/1469872864927
来源网站：链滴

<h2 id="toc_h2_0">第二部分 自动内存管理机制</h2>
<h2 id="toc_h2_1">第2章 JAVA内存区域与内存溢出异常</h2>
<h3 id="toc_h3_2">运行时数据区</h3>
根据JAVA虚拟机规范 SE 7 运行时数据区包括以下内容

所有线程共享的

方法区
堆

线程隔离的数据区

虚拟机栈
本地方法栈
程序计数器

<h4 id="toc_h4_3">程序计数器</h4>

存储当前线程执行到哪一个字节码。
JAVA虚拟机可以在一个系统线程内，控制切换多个 JAVA线程，程序计数器 提供了这个实现的能
执行NATIVE方法时这个计数器值为空（JIT后应该也为空）
程序计数器应该跟 栈帧 配合使用

<h4 id="toc_h4_4">JAVA虚拟机栈</h4>
虚拟机栈

虚拟机栈是一个存储当前线程执行的方法层次的一个FILO容器
虚拟机每开始执行一个JAVA方法都会创建一个 栈帧，并将其放到虚拟机栈中
虚拟机方法执行完后就会从虚拟机栈中移除这个栈帧

栈帧

栈帧保存的内容包括：局部变量表，返回值，操作数栈（？），动态链接（？）
局部变量表 存放了编译器可知的 各种基本数据类型，boolean,int,char,short,int,long,float,double，对象引用 及 returnAddress(指向字节码指令的地址)

可能抛出的异常

StackOverflowError 超出虚拟机所允许的最大栈长度
OutOfMemoryError 栈扩展时无法申请所需的空间

<h4 id="toc_h4_5">本地方法栈</h4>

与JAVA虚拟机栈相似，不过本地方法栈是为了虚拟机调用 本地方法而设定。
有些虚拟机的本地方法栈和JAVA虚拟机栈是合二为一的，如HOTSPOT

<h4 id="toc_h4_6">JAVA堆</h4>

JAVA堆设计的唯一目的是存放实例对象
几乎所有的实例对象和数组都是放在堆里

随着JIT编译器的发展 以及 逃逸分析技术逐渐成熟 栈上分配 以及 标量替换优化技术 使得对象不定在堆上分配
JAVA堆是GC的主要区域，而GC算法大多都采用分代算法因此，JAVA堆根据GC算法的不同还可分为不同的区域：

粗略的划分：新生代，老年代
细致的划分：Eden空间（刚创建，未经历过GC的对象所存放的空间），From Survivor,To Survivr
功能的划分：线程私有分配缓冲区

当没有空间完成对象的分配时，会抛出OutOfMemoryError异常

<h4 id="toc_h4_7">方法区</h4>

用于存储 已经加载的类的信息，常量，静态变量，即时编译器编译后的代码数据等
方法区 跟 JAVA堆一样，都在堆中。为了区分Java堆和方法区，方法区的一个别名是 Non-Heap/li>
方法区的具体设计在JAVA虚拟机规范中并没有强制规定，因此在一些JVM的实现中，方法区会到了JAVA堆里。
当方法区无法满足新的内存分配需求时，会抛出 OutOfMemoryError异常

常量池

常量池存储在方法区中
分为编译时常量池与运行时常量池
运行时可以把常量加到常量池中，如String.intern()
虚拟机规范对编译时常量的存储有严格规范，但是对运行时常量没有规范的要求

<h4 id="toc_h4_8">直接内存</h4>

直接内存并不是JAVA虚拟机管控的内存
JDK1.4加入了NIO类，引入了基于Channel以及Buffer的IO形式，可使用Native方法直接分配JAA堆外内存，在一些场景下能提高性能，避免在JAVA堆和Native堆中来回复制数据
直接内存不受Xmx虚拟机参数管控，因此，有可能导致软件使用的内存大于指定的空间

<h3 id="toc_h3_9">HOTSPOT虚拟机对象解密</h3>
<h4 id="toc_h4_10">对象的创建</h4>

遇到NEW时，检查常量池中有没有对应的类的符号引用
检查类有没有被初始化，若无，则初始化
在JAVA堆中分配出对象所需的空间

GC方法带有压缩过程的话，如Serial、ParNew，会使用指正碰撞的方法分配内存。（指针记录边已分配，右边未分配）
GC方法不带压缩的话，如CMS，会使用空闲列表来分配内存。
因为多线程会竞争分配，这里会降低效率，因此设计了Thread Local Allocate Buffer，预先为程分配了内存，因此线程内无需考虑并发情况
分配对象内存时，虚拟机会给分配的空间的值都值为0
然后设置对象的对象头，包括 MarkWord,对象对应的类的指针，数组长度等。
然后虚拟机负责的任务就完成了，开始调用用户自定义的初始化方法

<h4 id="toc_h4_11">对象的内存布局</h4>
包含三项内容

对象头

MarkWord
对象所属类的定义的指针
数组长度（可选，若是数组，则有这个东西）

实例数据（真正的用户数据）
Padding（对齐填白，HOTSPOT管理内存时，要求对象的起始地址要为8字节的整数倍，因此当小不符时，要填白）

<h4 id="toc_h4_12">对象访问定位</h4>
通过栈中的Referance访问对象的实现有两种类型，使用句柄 以及 直接指针访问

使用句柄

JAVA栈中的Referance指向句柄池（存储在JAVA堆）
句柄存储着对象的类型的指针以及对象自身的指针

直接指针访问

栈中的Referance直接指向对象的地址
对象自身存储着对象所属类型的指针

使用句柄的好处是，在GC的时候，只需修改句柄中对象的地址
使用直接指针访问的好处是访问速度快，不像句柄还需一次中间的转换，但是缺点是GC的时候要修改Referance中的地址
Hotspot使用的是直接指针访问
<h3 id="toc_h3_13">OutOfMemoryError实战</h3>
以下测试基于HOTSPOT
<h4 id="toc_h4_14">JAVA堆溢出</h4>
可以通过JVM参数 -XX:+HeapDumpOnOutOfMemoryError在内存出现溢出异常时，Dump出前内存堆的快照以便分析。
OutOfMemoryError异常会具体提示是哪一块出现问题了，如提示 JavaHeapSpace
分析工具Eclipse Memory Analyzer
确认分析内存中的对象是否是必要的——跟踪泄露对象到GC Roots的引用链
若没有泄露的对象，就要考虑增加最大的堆大小。
<h4 id="toc_h4_15">虚拟机栈 和 本地方法栈溢出</h4>
HOTSPOT 的虚拟机栈 以及 本地方法栈 都是同一个，因此不必区分。
经测试 无论是 减少栈容量大小 还是 增加调用方法的本地变量大小 造成的异常都是 StackOverFlwError。异常时输出的栈深度 会相对的减少
通过不断创建线程的方法，可以产生 OutOfMemoryError。
在32位WINDOWS中，系统给单个进程的内存限制为2G，减去Xmx最大堆容，再减去MaxPermSize最大方法区容量，再减去程序计数器的容量，再减去虚拟机自身的消耗，那么剩下的内存就由 虚拟机栈 和 本地方法栈 瓜分，当编写多线程程序的时候，发现报 OOM异常，且提为 JAVA栈 内存 不够时，如果 不能减少线程数 或者 更换64位虚拟机时 需要减少最大堆大小
<h4 id="toc_h4_16">方法区和运行时常量池溢出</h4>

因运行时常量池就在方法区中，因此，这两个的测试可以一并进行
JDK 1.7 开始去除永久代，因此 在 方法区 溢出时的表现 1.6与1.7 存在差异
String.intern()方法用于检索指定的字符串在 常量池中 是否存在，若存在则返回对应引用，若不在则将其放到 运行时常量池中。
一个不断往运行时常量池里通过 String.intern()塞东西的测试案例，在1.6中会报PermGen Space OOM,而在1.7中并不会报这个错误，因为1.7没有PermGem Space，且会对那些运行时常量进行回操作。
1.6的String.intern会将字符串复制一份到PermGem中，而1.7则会在运行时常量池里添加一个Heap中的Referance，而不会复制一份
造成方法区溢出的主要原因是 动态创建了太多类 或者 使用了OSGI 而类无法及时回收。
<h4 id="toc_h4_17">本机直接内存溢出</h4>
Direct Memory 容量可以通过 -XX:MaxDirectMemory来指定，若不指定，则跟Xmx一样。 Jav虚拟机会记录一共申请了多少Direct Memory，若超过了指定值，会直接报错（不会向系统申请）</>
这个Direct Memory的OOM并不会指明是哪个空间内存不够，因此需要自行判断下，是否直接存不足。
<h2 id="toc_h2_18">第3章 垃圾收集器 与 内存分配策略</h2>
<h3 id="toc_h3_19">对象已死？</h3>
<h4 id="toc_h4_20">引用计数算法</h4>
新增一个引用，则计数器+1，引用消失 则计数器 -1，计数器为0，则表示对象不可能再被使用
本方法实现简单，但无法解决循环引用的问题。即 A->B 且 B->A。这种情况，将一直无进行GC
<h4 id="toc_h4_21">可达性分析</h4>
通过GC ROOTS对象能访问得到的对象就是正在使用的对象，访问不到的，则可回收
GC Roots对象包括以下几种：

虚拟机栈（机栈中的本地变量表）
方法区中静态属性引用的对象
方法区中常量引用的对象
本地方法栈中JNI引用的对象

<h4 id="toc_h4_22">引用的分类</h4>
引用实际上在JDK1.2后，分为了几个类型

强引用，正常的引用
软引用（SoftReferance）,当内存不够，OOM发生之前，会将软引用的数据回收，若还是不够抛出OOM
弱引用（WeakReferance）,无论内存是否足够，弱引用对象都会在下一次GC的时候被回收
虚引用（PhantomReferane）,比弱引用更弱，无法通过虚引用获得一个对象实例。设置虚引用唯一目的是在对象被GC时受到一个系统通知

<h4 id="toc_h4_23">拯救死亡对象</h4>
对象若达到了GC条件，那么很快就会被回收掉，回收前夕，如果对象重写了 finalize方法，那么对象有可能通过finalize方法避免GC。只要在finalize中重新为对象建立一个到GC ROOTS的强连接即。finalize执行完后，虚拟机会再次判断该对象是否要GC，发现已经跟GC ROOTS 建立了强连接的话那么就会将其移出要回收的对象的组合。
但虚拟机并不承诺等待finalize的执行完毕，拯救行动有可能失败。 而且finalize只会被虚拟机执一遍，第二次要被GC的时候，就无法被拯救了。
拯救死亡对象并不是一个被推荐的方法，可能导致奇怪的情况出现。
<h4 id="toc_h4_24">回收方法区</h4>
方法区回收的主要内容是 废弃常量 以及 无用的类。
废弃常量的判断方法跟 可达判断类似，除了自己之外，通过其他GC Roots不可达，那么该常量被回收。

废弃类的判断比较复杂：

要求该类的所有实例都已被回收
加载该类的ClassLoader已经被回收
该类对应的 java.lang.Class对象没有在任何地方被引用（即无法在任何地方通过反射，访问该类方法）

满足以上条件的类，可以被回收。具体是否回收在HotSpot中提供了 -Xnoclassgc参数进行控制使用-verbose:class以及-XX:+TraceClassLoading（Product版本可用） -XX:+TraceClassUnLoadin（FastDebug版本可用）查看类的加载和卸载信息
大量使用反射、动态代理、CGLib等ByteCode框架需要具备类卸载的功能，否则有可能导致方区溢出（Hotspot 1.6 及之前的永久代溢出）
<h3 id="toc_h3_25">垃圾回收算法</h3>
<h4 id="toc_h4_26">标记-清楚算法</h4>
分为 标志 及 清除 两个阶段，是大多数回收算法的基础，存在以下缺点

效率，标记以及清除的效率都不高
空间，清除后会产生大量空间碎片，导致可能无法继续分配大对象

<h4 id="toc_h4_27">复制算法</h4>
将内存分为对等的两半，每次只使用其中的一半，GC时，将存活的对象直接复制到另外一半中原有的一半一次性清理掉。这样就解决了碎片的问题 以及 清理效率的问题。
但是代价也很明显，就是只能使用一半的内存。因此实际应用中，是有改进的。
据IBM的调查发现 新生代的对象 98% 都会在第一次GC时被回收掉，因此HOTSPOT新生代的默设计如下

80%新生代容量分配给Eden(一次GC都没经历过的对象所存放的空间)
有两个大小一样的survivor区，各占新生代容量的10%。

GC时，将Eden以及当前在使用的survivor存活的对象复制到另外一个survivor中。 按照如上设，新生代可用内存为新生代总内存的90%。
当然，有可能存在 存活对象的大小大于 survivor区大小的情况，这时候，survivor可以跟老年代内存。当多次GC都无法将借的内存偿还时，就将部分对象升级为 老年代，这样survivor又可以恢复原来的大小了。
<h4 id="toc_h4_28">标记-整理算法</h4>
复制算法只适用于 新生代，因为其存活率较低，但是老年代的存活率较高，因此需要一个新的法。
标记整理算法就是，标记处哪些是需要保留的对象，然后将存活的对象都往一边移动，直接清理边界以外的内存。就跟磁盘整理程序一样一样的。
<h4 id="toc_h4_29">分代收集算法</h4>
就是说 根据 对象存活的年代选用 不同的回收算法。 像上面的，新生代使用 复制算法，老年代用 标记-整理 算法。
<h3 id="toc_h3_30">HOTSPOT的算法实现</h3>
<h4 id="toc_h4_31">枚举根节点</h4>
枚举根节点进行GC分析的时候，所有线程都必须停止下来，否则不能得到一个准确的结果。 因枚举根节点的过程必须快。
根节点包括 方法区里的静态引用，常量，以及 本地方法 JAVA方法栈 里的局部变量。
在上述的位置的变量里，只有少数部分才是Referance，这些会影响GC。其余的GC并无影响。此快速准确识别出那些是Referance,才能提高GC的时间。
对于静态引用及常量，可维护一个 OOPMAP(Ordinary Object Map)，若是Referance，则将加到Map中，若静态引用及常量被移除
对于 JAVA方法 及 JIT后的方法，也会使用OOP来快速识别这些Referance。
对于解析的JAVA方法，OOP MAP的生成是动态的，在GC开始后，分析字节码生成 对于JIT后方法，OOP MAP会在编译时生成，并放在编译后的代码的一些特定位置，如：方法调用前，循环跳

，异常跳转等。这些特定的位置叫安全点
对于JNI方法 对于JNI方法中受虚拟机管控的GC ROOTS的枚举无法使用OOP MAP，因为 代码是JIT生成的，也非JVM解析执行的。 对于传入JNI的JAVA对象，必须通过句柄的包装。这样虚拟机可以通过句柄 枚举JNI调用部分的 GC ROOTS
<h4 id="toc_h4_32">准确GC</h4>
OOP MAP另外一个作用就是方便JVM实现 准确GC。就是说能识别出所有位置的Referance，当eferance指向的Object位置发生变化时，也可以对Referance中存储的地址进行变更。因此准确GC以移动Object在内存中的位置。
对应的保守GC 不能准确地判断出某些变量是否Referance，因此无法移动Object的位置
<h4 id="toc_h4_33">安全点</h4>
上面一节已经提到，存储了OopMap的位置称为安全点。
当需要GC时，所有的线程都必须处于安全点中，这个做法通常是，当GC线程发现需要进行GC时则设置某一个标志位，其余线程在安全点（OOP MAP）检查这个标志是否置上，若置上了，会进入定的等待处理逻辑。
等到所有的线程都停止了，就可以进行 GC 了。
<h4 id="toc_h4_34">安全区域</h4>
但有时 线程处于Sleep 或者 Blocked的时候，不太可能继续继续往下走，这样GC就等待不到所的线程进入安全点。因此，引入一个新的概念，安全区域。
安全区域指代 在安全区域执行代码的线程的引用关系不会发生变化。当一个线程将自己标志位进入Safe Region,那么GC线程就不管这个线程，直接进行GC了
当线程离开Safe Region的时候，要检查系统是否完成了 根节点枚举（或者是整个GC过程），果完成了，则可以安全离开，若否，则需要等待可以离开的信号。
<h3 id="toc_h3_35">垃圾收集器</h3>
以下基于 JDK 1.7 UPDATE14 后的HOTSPOT虚拟机

新生代收收集器

Serial
ParNew
Parallel Scavenge
G1

老年代收集器

CMS
Serial Old(MSC)
Parallel Old
G1

没有最好的垃圾收集器，只有根据场景最适用的垃圾收集器
<h4 id="toc_h4_36">Serial收集器</h4>
新生代区域内存收集器，使用一个线程进行GC，使用复制算法。 优点：

单线程效率最高的垃圾收集器
Client模式下默认的收集器

缺点：

GC时会STOP THE WORLD

Client模式下的默认新生代收集器，因为其适用于 回收小内存，几百M的内存回收在一百多毫秒

就能完成，用户基本察觉不到
<h4 id="toc_h4_37">ParNew收集器</h4>
Serial收集器的多线程版本，它们公用了很多的代码
他是很多服务器的首选收集器，因为只有 它以及Serial 可以跟 CMS配合使用
ParNew开启的GC线程默认跟CPU核数一样，可以通过 JVM参数 ParallelGCThreads进行限制<
p>
<h4 id="toc_h4_38">Parallel Scavenge 收集器</h4>
与PARNEW一样，是多线程复制新生代收集算法。但其可以根据运行的业务调整吞吐量，侧向
高吞吐量。吞吐量=用户运行代码时间/（运行用户代码时间+垃圾收集时间），通过以下两个参数设
：

MaxGCPauseMills 控制最大GC时间
GCTimeRation 直接控制吞吐量

可以使用UseAdaptiveSizePolicy参数开启自适应GC策略，会自动调整 EDEN,Suvivor的比例，
升老年代年龄等等。
这个自适应策略是区别于ParNew的一个重要特性。
<h4 id="toc_h4_39">Serial OLD收集器</h4>
Serial收集器的老年代版本，单线程收集器，使用 标记-整理（Mark-Compact） 算法
一般情况下也是用于客户端
若用于服务器，一般用于CMS的后备方案（后文会讲）
<h4 id="toc_h4_40">Parallel Old收集器</h4>
Parallel Scavenge的老年代版本。使用多线程 Mark-Compact 算法。只能与Parallel Scaveng
配合使用。
<h4 id="toc_h4_41">CMS（Concurrent Mark Sweep）收集器</h4>
以获取最短停顿为目标的老年代收集器。使用多线程 Mark-Sweep 算法。
本算法的主要核心算法

第一次进入SafePoint的时候，Stop the World,获取所有的 GC ROOTS进行初始标志完成后，
复大部分用户线程
剩余的一部分未恢复的线程会 继续进行 并发标志（并发 指代 用户线程 与 GC线程一起工作）</l
>
并发标志结束后，由于之前用户线程继续运行，会新增新的GC ROOT 以及 一些对象会新增一些
用
因此需要在下次Safe Point的时候再次STOP THE WORLD，所有的计算 资源都投入到这些增量
引用的计算中
经过这次STOP THE WORLD就能标志出所有可能在使用的对象，据此可以反推出 一定是无用的
象，此时可以恢复大部分用户的线程，留下一部分GC线程回收这些标志为垃圾的对象即可
当回收完毕，GC线程全部释放，计算能力全部归还用户

优点：低停顿
缺点：

当主机核数为4核以上时，会占用不少于25%的用户计算资源，如果核数为2核时，会占用50%的
算支援，1核就是全部了，实际上还是会影响服务器的性能的
CMS无法处理浮动垃圾。在并发清理或者并发标记的过程中，用户处理线程不断产生对象，导致
存不够用的时候，CMS不再适用，会报 Concurrent Model failure，改成使用 Serial Old算法来清
（所谓的FULL GC）。
CMS使用的是Sweep，因此有可能没有连续的空间用于创建大对象，此时，要进行FULL GC(Seria
OLD),Serial Old会进行COMPACT的操作。JVM也支持设置成 进行N次FULL GC后才COMPACT的
置

FULL GC的Serial Old算法可改造成 Parallel,社区已经有体验版
<h4 id="toc_h4_42">G1收集器</h4>

<h5 id="toc_h5_43">思想简介</h5>
是当前收集器技术发展最前沿的成果之一，HotSpot赋予G1的使命是最终能替换掉CMS收集器 G可用于新生代及老年代，其适用于 多核，大内存的服务器，它STOP THE WORLD的时间短，且有较的吞吐量，且G1允许用户设定 最大停顿时间。
G1的主要设计思想是化整为零，逐个击破。
G1会将JAVA堆分成很多个大小相等的REGION(具体的大小会在运行时视情况实时调整),这些REGON可以是 eden，也可以是 survivor，也可以是 ，每个Region都可以单独的被回收，回收时将存活对象都拷贝到另外一个REGION，拷贝完成后全部回收之前的REGION ，回收的个数也会视情 用户设的 最大停顿时间 及 历史经验 动态决定，但GC会优先回收 回收价值最高的REGION（根据经验值 推测回收获得的空间大小及所用的时间），这也是G1(Garbage first)的名字的由来。
<h5 id="toc_h5_44">RememberSet</h5>
在年轻代回收的时候，如果每次都需要在年轻代 及 老年代 全体对象间计算引用关系的话，那么耗费巨大的时间。为了减少 老年代 引用的计算，Serial,ParNew等GC算法 会维护一个 老年代的RemmberSet.RememberSet代表着整个老年代引用年轻代对象的记录（在为一个对象内的Referance赋时，JVM会检查对象是否为老年代 若是 且 referance指向 年轻代，那么就会为rememberSet增加一记录。删除RS记录的原理也类似）。有了RS后，Serial等Minor GC时，将老年代的RS也作为 GC ROTS，一起计算 young gen 的可回收对象。这样就大大节约了年轻代回收 标志的时间。
在G1中，为了避免全堆栈扫描，只回收部分REGION也做了类似优化。G1为每个REGION都维了RS，那么要回收特定的几个REGION时，就将其余REGION的RS也加入到GC ROOTS的计算即可。/p>
<h5 id="toc_h5_45">具体执行过程</h5>

初始标志（方法区，栈，RS作为GC ROOTS标志对应的对象，STOP THE WORLD）
并发标志（根据之前的标志，继续往下进行标志，时间较长，与用户应用一起运行）
重新标志（STOP THE WORLD，根据 RS的变化日志，栈引用变化，方法区引用变化增量标志有的对象）
并行回收（STOP THE WORLD,根据标志复制有用的对象到新的REGION，无用的清空)
恢复用户线程

<h5 id="toc_h5_46">是否选择G1?</h5>
当之前的CMS,ParNew组合无法满足 延时 需求的时候，可以考虑使用G1，否则无需更换
<h5 id="toc_h5_47">参考</h5>
G1收集器ORACLE简介
<h3 id="toc_h3_48">内存分配回收策略</h3>

内存优先在EDEN中分配
大对象直接进入老年代，避免在 复制算法中 来回复制
长期存活的对象将进入老年代（每经过一次MINOR GC都存活，默认超过15次则会进入老年代）/li>
动态年龄判断（如果某一代的对象的大小总和大于survivor的一半，那么这一代及比它们老的一会直接进入老年代。个人猜测本条可能不适用于G1...）
空间分配担保（如果survivor的大小不足于保存存活的eden对象，那么只能让一部分对象直接进老年代），参数HandlePromotionFailure

为FALSE,不允许担保失败。MinorGC发生时，检测老年代的连续空间是否大于新生代的总空间若是，则只进行MinorGC,若否 则进行FullGC.
为true,则允许担保失败。MinorGC发生时，检测老年代的连续空间是否大于新生代存活的平均小（经验值），若小于，则直接FULL GC,若小于则执行MinorGC，若由于没有足够多的连续空间而生担保失败时，将会浪费掉之前MinorGC执行的时间，然后触发一次FULL GC。

<h2 id="toc_h2_49">虚拟机性能监控与故障处理</h2>
<h3 id="toc_h3_50">JDK命令行工具</h3>
<h4 id="toc_h4_51">jps</h4>

-m 传入main的参数
-l 输出执行类的全名，如果是JAR包，则输出JAR包的地址
-v 输出虚拟机启动的参数

<h4 id="toc_h4_52">jstat 虚拟机统计信息监视工具</h4>
显示 本地或者远程 虚拟机进程中的 类装载、内存、垃圾收集、JIT编译等运行数据
jstat -<option> [-t] [-h<lines>] <vmid> [<interval> [<count>]]
/p>
options: -class -compiler -gc -gccapacity -gccause -gcnew -gcnewcapacity -gcold -gcold
apacity -gcpermcapacity -gcutil -printcompilation
jstatd可创建远程RMI服务器，然后可通过远程UI排查问题
<h4 id="toc_h4_53">jinfo(Configuration info for java)</h4>
实时的查看和调整虚拟机各项参数，不过调整能否马上生效就不清楚了</p

<h4 id="toc_h4_54">jmap(生成JAVA堆转储快照)</h4>

-dump 生成堆转储快照
-finalizerinfo 查看Finalizer等待执行的对象
-heap 查看GC详细信息
-histo 显示堆中对象统计信息，包括类，实例数量，合计容量

<h4 id="toc_h4_55">jhat</h4>
在命令行环境下分析dump的工具，并提供http形式的访问，分析功能较弱...看不出啥东西
<h4 id="toc_h4_56">jstatck</h4>
在线分析性能问题，定位无响应，死锁，死循环 最有力的工具。用于生成当前线程快照。
<h4 id="toc_h4_57">HSDIS:JIT生成代码反编译</h4>
暂时还用不着
<h3 id="toc_h3_58">可视化工具</h3>
<h4 id="toc_h4_59">jconsole</h4>
<h4 id="toc_h4_60">jvisualvm</h4>

显示虚拟机进程及进程信息(jps,jinfo)
监控应用程序的CPU,GC,堆，方法区以及线程的信息（jstat,jstack）
DUMP及分析转储快照(jmap,jhat)
性能调优，找出方法级别被调用的次数及时间
其他的PLUGIN...

<h4 id="toc_h4_61">jmc</h4>
<h2 id="toc_h2_62">调优案例与实战</h2>
<h3 id="toc_h3_63">案例分析</h3>
<h4 id="toc_h4_64">高性能硬件上的程序部署策略</h4>
若由于64位虚拟机且配上大内存导致的FULL GC时间过长，那么可以考虑将一个虚拟机改造成多
32位虚拟机的集群形式。 又或者尝试使用G1 垃圾收集器。
<h4 id="toc_h4_65">堆外内存导致的移除错误</h4>
当发生OOME的时候，且没有指明溢出位置，那很有可能就是 堆外内存发生了溢出。
堆外内存泄漏在32位JVM中特别容易发生，因为单进程的理论最大内存为4G，windows中为2G
如果给堆内存就分配了1.6G,那么堆外内存可用就特别的少了
堆外内存的清理，要在FULL GC的时候才会执行