判断对象是否存活的算法

1. 引用计数算法

   为每一个对象设置一个对象引用计算器，每当有地方引用到该对象的时候，该对象的引用计算器就自动的加一，如果每当引用失效的时候，该对象的引用计算器就会相应的减一。任何时刻当该对象的引用计数器为0的时候就说明了该对象不再被引用。但是不能解决循环引用问问题（A对象引用B对象，B对象又引用A对象，但是A,B对象已不被任何其他对象引用），同时每次计数器的增加和减少都带来了很多额外的开销，所以在JDK1.1之后，这个算法已经不再使用了。

2. 可达性分析算法

   这个算法的基本思想是通过一个根节点RootGC作为一个起始点，从这个节点往下搜索，搜索所走过的路径就是引用链（Reference Chain），当一个对象到RootGC都没有引用链的时候（图论说法就是当一个GCRoot节点到该对象不可达），则证明此对象是不可用的。

   2.1 java中可以用作GC Root的对象

   1.虚拟机栈中引用的对象

2.方法区中静态属性引用的对象

3.方法区中常量引用的对象

4.本地方法栈JNI（即一般说的Native方法）引用的对象

2.2 gc自我拯救

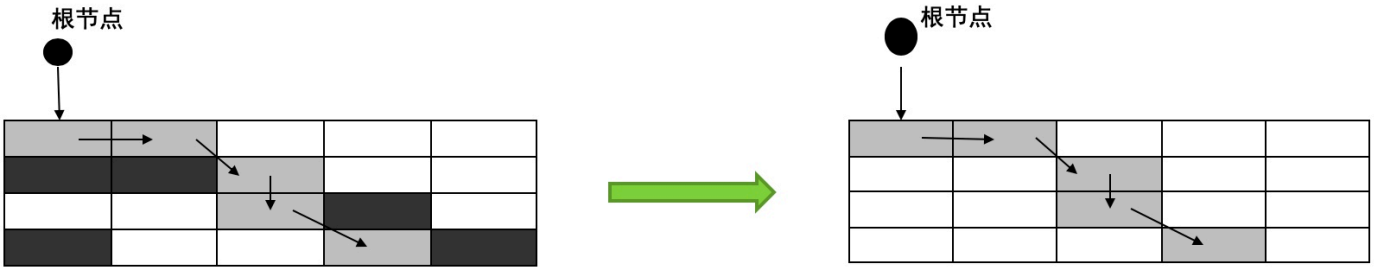
当通过系统分析发现，RootGC节点到该对象不可达的时候，是否对象就会被回收的呢，答案不是一定的，这时候他暂时时处于缓刑阶段，至少要经过两次的标记的过程，才真正宣告一个对象的死亡，第一次是当系统检测到该对象到RootGC节点不可达的时候，进行第一次的标记，然后系统就会检查该对象有没有覆盖finalize方法，如果有的话便会执行finalize方法，如果该对象在finalize方法中与任何一个对象进行关联的话便可以不会被回收。

垃圾回收算法

GC的对象是堆空间和永久区

元空间的内存管理由元空间虚拟机来完成。每一个类加载器的存储区域都称作一个元空间，所有的元空间合在一起就是我们一直说的元空间。当一个类加载器被垃圾回收器标记为不再存活，其对应的元空间会被回收。在元空间的回收过程中没有重定位和压缩等操作。但是元空间内的元数据会进行扫描来确定Java引用。
-XX:MetaspaceSize 初始元空间大小,达到该值就会触发垃圾收集进行类型卸载，同时GC会对该值进行调整,如果释放了大量空间,就适当减少,如果释放了很少的空间,在不超过MaxMetaspaceSize时可以增加.
-XX:MaxMetaspaceSize 最大的元空间大小,元空间大小不得超过这个值,如果超过会抛出OOM异常

- 标记-清除
  标记-清除算法是现代垃圾回收算法的思想基础。标记-清除算法将垃圾回收分为两个阶段：标记阶段和清除阶段。一种可行的实现是，在标记阶段，首先通过根节点，标记所有从根节点开始的可达对象。因此，未被标记的对象就是未被引用的垃圾对象。然后，在清除阶段，清除所有未被标记的对象。



缺点：

1、效率问题，标记和清除两个过程的效率都不高；

2、空间问题，标记清除之后会产生大量不连续的内存碎片，空间碎片太多可能会导致以后在程序运行过程中需要分配较大的对象时，无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作。

- 标记-压缩

  标记-压缩算法适合用于存活对象较多的场合，如老年代。它在标记-清除算法的基础上做了一些优化。和标记-清除算法一样，标记-压缩算法也首先需要从根节点开始，对所有可达对象做一次标记。但之后，它并不简单的清理未标记的对象，而是将所有的存活对象压缩到内存的一端。之后，清理边界外所有的空间。

缺点：

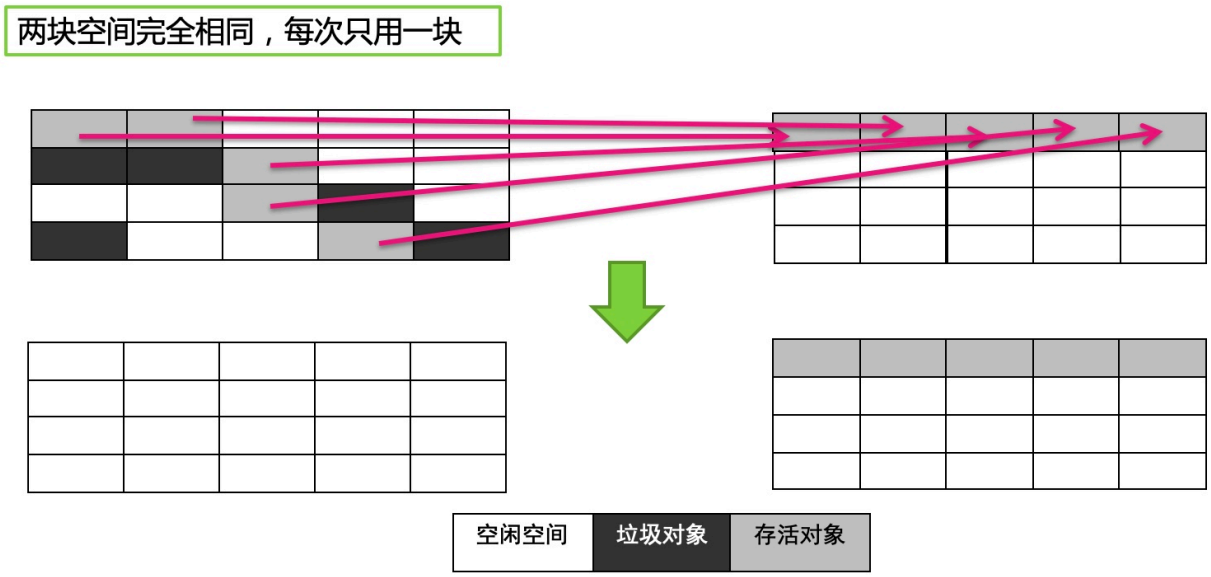
1、效率问题，（同标记清除算法）标记和整理两个过程的效率都不高；

优点：

1、相对标记清除算法，解决了内存碎片问题。

2、没有内存碎片后，对象创建内存分配也更快速了（可以使用TLAB进行分配）。

- 复制算法
  与标记-清除算法相比，复制算法是一种相对高效的回收方法
  不适用于存活对象较多的场合 如老年代
  将原有的内存空间分为两块，每次只使用其中一块，在垃圾回收时，将正在使用的内存中的存活对象复制到未使用的内存块中，之后，清除正在使用的内存块中的所有对象，交换两个内存的角色，完成垃圾回收



  优点
  效率高，没有内存碎片
  缺点：
  1、浪费一半的内存空间
  2、复制收集算法在对象存活率较高时就要进行较多的复制操作，效率将会变低。

- 分代
  当前商业虚拟机都是采用分代收集算法，它根据对象存活周期的不同将内存划分为几块，一般是把Java堆分为两个不同的区域：新生代 (Young)、老年代 (Old)。新生代 (Young) 又被划分为三个区域：Eden、From Survivor、To Survivor。
  这样划分的目的是为了使 JVM 能够更好的管理堆内存中的对象，包括内存的分配以及回收。
  堆的内存模型大致为：

从图中可以看出： 堆大小 = 新生代 + 老年代。其中，堆的大小可以通过参数 –Xms、-Xmx 来指定。
（本人使用的是 JDK1.6，以下涉及的 JVM 默认值均以该版本为准。）
默认的，新生代 (Young) 与老年代 (Old) 的比例的值为 1:2 (该值可以通过参数 –XX:NewRatio 来指定)，即：新生代 (Young) = 1/3 的堆空间大小。老年代 (Old) = 2/3 的堆空间大小。其中，新生代 (Young) 被细分为 Eden 和 两个 Survivor 区域，这两个 Survivor 区域分别被命名为 from 和 to，以示区分。
默认的，Edem : from : to = 8 : 1 : 1 (可以通过参数 –XX:SurvivorRatio 来设定)，即： Eden = 8/10 的新生代空间大小，from = to = 1/10 的新生代空间大小。
JVM 每次只会使用 Eden 和其中的一块 Survivor 区域来为对象服务，所以无论什么时候，总是有一块 Survivor 区域是空闲着的。因此，新生代实际可用的内存空间为 9/10 (即90%)的新生代空间。

Minor GC 和 Full GC的区别
新生代GC（Minor GC）：Minor GC指发生在新生代的GC，因为新生代的Java对象大多都是朝生夕死，所以Minor GC非常频繁，一般回收速度也比较快。当Eden空间不足以为对象分配内存时，会触发Minor GC。
老年代GC（Full GC/Major GC）：Full GC指发生在老年代的GC，出现了Full GC一般会伴随着至少一次的Minor GC（老年代的对象大部分是Minor GC过程中从新生代进入老年代），比如：分配担保失败。Full GC的速度一般会比Minor GC慢10倍以上。当老年代内存不足或者显式调用System.gc()方法时，会触发Full GC。

Full GC触发条件：
（1）调用System.gc时，系统建议执行Full GC，但是不必然执行
（2）老年代空间不足
（3）方法去空间不足

（4）通过Minor GC后进入老年代的平均大小大于老年代的可用内存
（5）由Eden区、From Space区向To Space区复制时，对象大小大于To Space可用内存，则把该对象转存到老年代，且老年代的可用内存小于该对象大小

垃圾收集器

收集算法是jvm内存回收过程中具体的、通用的方法，垃圾收集器是jvm内存回收过程中具体的执行者，即各种GC算法的具体实现。

- 相关概念
  - 并行和并发
    并行（Parallel）：指多条垃圾收集线程并行工作，但此时用户线程仍然处于等待状态。
    并发（Concurrent）：指用户线程与垃圾收集线程同时执行（但不一定是并行的，可能会交替执行），用户程序在继续运行。而垃圾收集程序运行在另一个CPU上。
  - 吞吐量（Throughput）
    吞吐量就是CPU用于运行用户代码的时间与CPU总消耗时间的比值，即
    吞吐量 = 运行用户代码时间 /（运行用户代码时间 + 垃圾收集时间）。
    假设虚拟机总共运行了100分钟，其中垃圾收集花掉1分钟，那吞吐量就是99%。
  - Minor GC 和 Full GC
    新生代GC（Minor GC）：指发生在新生代的垃圾收集动作，因为Java对象大多都具备朝生夕灭的特性，所以Minor GC非常频繁，一般回收速度也比较快。
    老年代GC（Major GC / Full GC）：指发生在老年代的GC，出现了Major GC，经常会伴随至少一次的Minor GC（但非绝对的，在Parallel Scavenge收集器的收集策略里就有直接进行Major GC的策略选择过程）。Major GC的速度一般会比Minor GC慢10倍以上。

  - Stop-The-World
    Java中一种全局暂停的现象，全局停顿，所有Java代码停止，native代码可以执行，但不能和JVM交互

    - 多半由于GC引起
    - Dump线程
    - 死锁检查
    - 堆Dump

    Serial、ParNew、Parallel Scavenge用于新生代；

CMS、Serial Old、Paralled Old用于老年代。
并且他们相互之间以相对固定的组合使用（具体组合关系如上图）。G1是一个独立的收集器不依赖其他6种收集器。ZGC是目前JDK 11的实验收集器。

1. Serial收集器
   Serial（串行）收集器是最基本、发展历史最悠久的收集器，它是采用复制算法的新生代收集器，曾经（JDK 1.3.1之前）是虚拟机新生代收集的唯一选择。它是一个单线程收集器，只会使用一个CPU或一条收集线程去完成垃圾收集工作，更重要的是它在进行垃圾收集时，必须暂停其他所有的工作线程，直至Serial收集器收集结束为止（“Stop The World”）。这项工作是由虚拟机在后台自动发起和自动完成的，在用户不可见的情况下把用户正常工作的线程全部停掉，这对很多应用来说是难以接收的。
   垃圾收集的过程中会Stop The World（服务暂停）
   参数控制：
   -XX:+UseSerialGC 串行收集器

2. ParNew收集器
   ParNew收集器除了使用多线程收集外，其他与Serial收集器相比并无太多创新之处，但它却是许多运行在Server模式下的虚拟机中首选的新生代收集器，其中有一个与性能无关的重要原因是，除了Serial收集器外，目前只有它能和CMS收集器（Concurrent Mark Sweep）配合工作，在多CPU环境下，随着CPU的数量增加，它对于GC时系统资源的有效利用是很有好处的。它默认开启的收集线程数与CPU的数量相同
   参数控制：
   -XX:+UseParNewGC ParNew收集器
   -XX:ParallelGCThreads 限制线程数量参数

3. Parallel Scavenge 收集器
   Parallel Scavenge与ParNew类似，都是用于年轻代回收的使用复制算法的并行收集器，与ParNew不同的是，Parallel Scavenge的目标是达到一个可控的吞吐量，吞吐量=程序运行时间/（程序运行时间+GC时间），如程序运行了99s，GC耗时1s，吞吐量=99/（99+1）=99%。Parallel Scavenge提供了两个参数用以精确控制吞吐量，分别是用以控制最大GC停顿时间的-XX:MaxGCPauseMillis及直接控制吞吐量的参数-XX:GCTimeRatio.MaxGCPauseMiilis:单位为ms，适用于高用户体验的场景，虚拟机将尽力保证每次MinorGC耗时不超过所设时长，但并不是该时间越小越好，因为GC耗时缩短是用调小年轻代获取的，回收500m的对象肯定要比回收2000m的对象耗时更短，但是回收频率也大大增大了，吞吐量也随之下去了。使用该参数的理论效果：MaxGCPauseMillis越小，单次MinorGC的时间越短，MinorGC次数增多，吞吐量降低
   GCTimeRatio:从字面意思上理解是花费在GC上的时间占比，但是实际含义并非如此，GC耗

时的计算公式为1/(1+n)，n为GCTimeRatio，因此，GCTimeRatio的实际用途是直接指定吞吐量。GCTimeRatio的默认值为99，因此，GC耗时的占比应为1/（1+99）=1%。使用参数的理论效果：GCTimeRatio越大，吞吐量越大，GC的总耗时越小。有可能导致单次MinorGC耗时变长。适用于高运算场景。
此外，还有个参数和以上两个参数息息相关，那就是-XX:+UseAdaptiveSizePolicy，默认为启用，搭配MaxGCPauseMillis或GCTimeRatio使用，打开该开关后，虚拟机将根据当前系统运行情况收集性能监控信息，动态调整SurvivorRatio，PretenureSizeThreshold等细节参数。
使用方式：该收集器是server模式下的默认收集器，也可-XX:+UseParallelGC强制使用该收集器，打开该收集器后，将使用Parallel Scavenge（年轻代）+Serial Old(老年代)的组合进行GC。

4. Serial Old收集器
   Serial Old 是 Serial收集器的老年代版本，它同样是一个单线程收集器，使用“标记-整理”（Mark-Compact）算法。
   此收集器的主要意义也是在于给Client模式下的虚拟机使用。如果在Server模式下，它还有两大用途：
   - 在JDK1.5 以及之前版本（Parallel Old诞生以前）中与Parallel Scavenge收集器搭配使用。
   - 作为CMS收集器的后备预案，在并发收集发生Concurrent Mode Failure时使用。

5. Parallel Old 收集器
   Parallel Old收集器是Parallel Scavenge收集器的老年代版本，使用多线程和“标记-整理”算法。前面已经提到过，这个收集器是在JDK 1.6中才开始提供的，在此之前，如果新生代选择了Parallel Scavenge收集器，老年代除了Serial Old以外别无选择，所以在Parallel Old诞生以后，“吞吐量优先”收集器终于有了比较名副其实的应用组合，在注重吞吐量以及CPU资源敏感的场合，都可以优先考虑Parallel Scavenge加Parallel Old收集器。
   参数控制：-XX:+UseParallelOldGC使用Parallel收集器+ 老年代并行

6. CMS收集器
   CMS（Concurrent Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的收集器，它非常符合那些集中在互联网站或者B/S系统的服务端上的Java应用，这些应用都非常重视服务的响应速度。从名字上（“Mark Sweep”）就可以看出它是基于“标记-清除”算法实现的。
   CMS收集器工作的整个流程分为以下4个步骤：
   i. 初始标记（CMS initial mark）：仅仅只是标记一下GC Roots能直接关联到的对象，速

度很快，需要“Stop The World”。

ii. 并发标记（CMS concurrent mark）：进行GC Roots Tracing的过程，在整个过程中耗时最长。

iii. 重新标记（CMS remark）：为了修正并发标记期间因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段稍长一些，但远比并发标记的时间短。此阶段也需要“Stop The World”。

iv. 并发清除（CMS concurrent sweep）
由于整个过程中耗时最长的并发标记和并发清除过程收集器线程都可以与用户线程一起工作，所以，从总体上来说，CMS收集器的内存回收过程是与用户线程一起并发执行的。

优点：
并发收集、低停顿，因此CMS收集器也被称为并发低停顿收集器（Concurrent Low Pause Collector）。
缺点：
Mark Sweep算法会导致内存碎片比较多
CMS的并发能力依赖于CPU资源，所以在CPU数少和CPU资源紧张的情况下，性能较差
并发清除阶段，用户线程依然在运行，所以依然会产生新的垃圾，此阶段的垃圾并不会再本次GC中回收，而放到下次。所以GC不能等待内存耗尽的时候才进行GC，这样的话会导致并发清除的时候，用户线程可以了利用的空间不足。所以这里会浪费一些内存空间给用户线程预留。

并发模式失败（Concurrent Mode Failure）
CMS收集器无法处理浮动垃圾（Floating Garbage），可能出现“Concurrent Mode Failure”失败而导致另一次Full GC的产生。

（1）如果对象提升到年老代的速度太快，而CMS收集器不能保持足够多的可用空间时，就会导致年老代的运行空间不足；
（2）当年老代的碎片化达到某种程度，使得没有足够空间容纳从新生代提升上来的对象时，也会发生并发模式失败。

当发生并发模式失败时，年老代将进行垃圾收集以释放可用空间，同时也会整理压缩以消除碎片，这个操作需要停止所有的java应用线程，并且需要执行相当长时间。

浮动垃圾
由于CMS并发清理阶段用户线程还在运行着，伴随程序运行自然就还会有新的垃圾不断产生，这一部分垃圾出现在标记过程之后，CMS无法在当次收集中处理掉它们，只好留待下一次GC时再清理掉。这一部分垃圾就称为“浮动垃圾”。

CMSInitiatingOccupancyFraction
也是由于在垃圾收集阶段用户线程还需要运行，那也就还需要预留有足够的内存空间给用户线程使用，因此CMS收集器不能像其他收集器那样等到老年代几乎完全被填满了再进行收集，需要预留一部分空间提供并发收集时的程序运作使用。

在JDK 1.5的默认设置下，CMS收集器当老年代使用了68%的空间后就会被激活，这是一个偏保守的设置，如果在应用中老年代增长不是太快，可以适当调高参数-XX:CMSInitiatingOccupancyFraction的值来提高触发百分比，以便降低内存回收次数从而获取更好的性能，

在JDK 1.6中，CMS收集器的启动阈值已经提升至92%。要是CMS运行期间预留的内存无法满足程序需要，就会出现一次“Concurrent Mode Failure”失败，这时虚拟机将启动后备预案：临时启用Serial Old收集器来重新进行老年代的垃圾收集，这样停顿时间就很长了。

所以说参数-XX:CMSInitiatingOccupancyFraction设置得太高很容易导致大量“Concurrent Mode Failure”失败，性能反而降低。

CMS内存整理
CMS是一款基于“标记—清除”算法实现的收集器，这意味着收集结束时会有大量空间碎片产生。

空间碎片过多时，将会给大对象分配带来很大麻烦，往往会出现老年代还有很大空间剩余，但是无法找到足够大的连续空间来分配当前对象，不得不提前触发一次Full GC。

为了解决这个问题，CMS收集器提供了一个-XX:+UseCMSCompactAtFullCollection开关参数（默认就是开启的），用于在CMS收集器顶不住要进行FullGC时开启内存碎片的合并整理过程，内存整理的过程是无法并发的，空间碎片问题没有了，但停顿时间不得不变长。

虚拟机设计者还提供了另外一个参数-XX:CMSFullGCsBeforeCompaction，这个参数是用于设置执行多少次不压缩的Full GC后，跟着来一次带压缩的（默认值为0，表示每次进入Full GC时都进行碎片整理）。

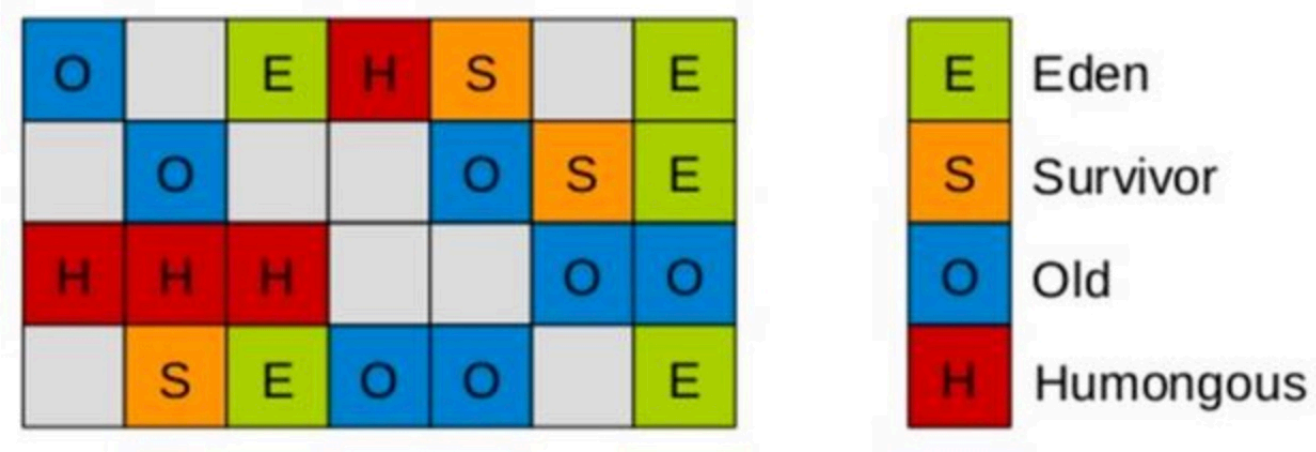
Promotion Failure
过早提升（Premature Promotion），MinorGC过程中，Survivor可能不足以容纳Eden和另外一个Survivor中存活的对象，如果Survivor中的存活对象溢出，多余的对象将被移到年老代。

在MinorGC过程中，如果年老代满了无法容纳更多的对象，则MinorGC之后，通常会进行FullGC，这将导致遍历整个java堆，这称为提升失败（Promotion Failure）

7. G1收集器
   在以下场景下G1更适合：
   服务端多核CPU、JVM内存占用较大的应用（至少大于4G）
   应用在运行过程中会产生大量内存碎片、需要经常压缩空间
   想要更可控、可预期的GC停顿周期；防止高并发下应用雪崩现象

   为了避免全堆扫描的发生，虚拟机为G1中每个Region维护了一个与之对应的Remembered Set。虚拟机发现程序在对Reference类型的数据进行写操作时，会产生一个Write Barrier暂时中断写操作，检查Reference引用的对象是否处于不同的Region之中（在分代的例子中就是检查是否老年代中的对象引用了新生代中的对象），如果是，便通过CardTable把相关引用信息记录到被引用对象所属的Region的Remembered Set之中。当进行内存回收时，在GC根节点的枚举范围中加入Remembered Set即可保证不对全堆扫描也不会有遗漏。

   内存模型



   - G1堆内存结构
     堆内存会被切分成为很多个固定大小区域（Region），每个是连续范围的虚拟内存。
     堆内存中一个区域(Region)的大小可以通过-XX:G1HeapRegionSize参数指定，大小区间最小1M、最大32M，总之是2的幂次方。
     默认把堆内存按照2048份均分。

   - G1堆内存分配
     每个Region被标记了E、S、O和H，这些区域在逻辑上被映射为Eden，Survivor和老年代。
     存活的对象从一个区域转移（即复制或移动）到另一个区域。区域被设计为并行收集垃圾，可能会暂停所有应用线程。
     如上图所示，区域可以分配到Eden，survivor和老年代。此外，还有第四种类型，被称为巨型区域（Humongous Region）。Humongous区域是为了那些存储超过50%标准region大小的对象而设计的，它用来专门存放巨型对象。如果一个H区装不下一个巨型对

象，那么G1会寻找连续的H分区来存储。为了能找到连续的H区，有时候不得不启动Full GC。

- G1提供了两种GC模式，Young GC和Mixed GC，两种都是完全Stop The World的。
  - Young GC：选定所有年轻代里的Region。通过控制年轻代的region个数，即年轻代内存大小，来控制young GC的时间开销。
  - Mixed GC：选定所有年轻代里的Region，外加根据global concurrent marking统计得出收集收益高的若干老年代Region。在用户指定的开销目标范围内尽可能选择收益高的老年代Region。
- Mixed GC不是full GC，它只能回收部分老年代的Region，如果mixed GC实在无法跟上程序分配内存的速度，导致老年代填满无法继续进行Mixed GC，就会使用serial old GC（full GC）来收集整个GC heap。所以我们可以知道，G1是不提供full GC的。
- global concurrent marking，它的执行过程类似CMS，但是不同的是，在G1 GC中，它主要是为Mixed GC提供标记服务的，并不是一次GC过程的一个必须环节。global concurrent marking的执行过程分为四个步骤：
  - 初始标记（initial mark，STW）。它标记了从GC Root开始直接可达的对象。
  - 并发标记（Concurrent Marking）。这个阶段从GC Root开始对heap中的对象标记，标记线程与应用程序线程并行执行，并且收集各个Region的存活对象信息。
  - 最终标记（Remark，STW）。标记那些在并发标记阶段发生变化的对象，将被回收。
  - 清除垃圾（Cleanup）。清除空Region（没有存活对象的），加入到free list。

    第一阶段initial mark是共用了Young GC的暂停，这是因为他们可以复用root scan操作，所以可以说global concurrent marking是伴随Young GC而发生的。第四阶段Cleanup只是回收了没有存活对象的Region，所以它并不需要STW。
- 发生Mixed GC其实是由一些参数控制着的，另外也控制着哪些老年代Region会被选入CSet。
  - G1HeapWastePercent：在global concurrent marking结束之后，我们可以知道old gen regions中有多少空间要被回收，在每次YGC之后和再次发生Mixed GC之前，会检查垃圾占比是否达到此参数，只有达到了，下次才会发生Mixed GC。
  - G1MixedGCLiveThresholdPercent：old generation region中的存活对象的占比，只有在此参数之下，才会被选入CSet。

    - G1MixedGCCountTarget：一次global concurrent marking之后，最多执行Mixed GC的次数。
    - G1OldCSetRegionThresholdPercent：一次Mixed GC中能被选入CSet的最多old generation region数量。

8. ZGC收集器

    - 目标
      支持TB级堆内存（最大4T）
      最大GC停顿10ms
      对吞吐量影响最大不超过15%

      ZGC没有分代，每次GC都会标记整个堆，大部分对象标记和对象转移都是可以和应用线程并发。只会在以下阶段会发生stop-the-world GC开始时对root set的标记时； 在标记结束的时候，由于并发的原因，需要确认所有对象已完成遍历，需要进行暂停； 在relocate root-set 中的对象时

- ZGC流程
  ZGC分为Mark（标记）、Relocate（迁移）、Remap（重映射）三个阶段
  Mark: 所有活的对象都被记录在对应Page的Livemap（活对象表，bitmap实现）中，以及对象的Reference（引用）都改成已标记（Marked0或Marked1）状态
  Relocate: 根据页面中活对象占用的大小选出的一组Page，将其中中的活对象都复制到新的Page, 并在额外的forward table（转移表）中记录对象原地址和新地址对应关系
  Remap: 所有Relocated的活对象的引用都重新指向了新的正确的地址

- ZGC为什么更快
  设计颜色指针标记对象状态，保障引用关系一致；通过读屏障检测对象状态，通过 CAS 对重映射对象进行迁移；使用 NUMA架构技术高效的分配空间和进行对象的扫描

内存分配策略

1. 对象优先在 Eden/TLAB 分配
   虚拟机将新生代内存分为一块较大的 Eden 空间和两块较小的 Survivor 空间（默认比例是 8:1:1），大多数情况下，分配对象时，使用 Eden 和其中一块 Survivor 空间，当没有足够空间进行分配时，虚拟机将会进行一次 MinorGC。
   如果虚拟机打开了 TLAB，那么对象优先在 TLAB 上分配。TLAB 全称是本地线程分配缓冲（Thread Local Allocation Buffer），它是每个线程在 Java 堆中预先分配的一小块内存。因

为 TLAB是线程私有的，没有锁开销，因此性能较好，在 JDK7 之后默认开启。

2. 大对象直接进入老年代
   虚拟机提供了一个 -XX:PretenureSizeThreshold 参数，令大于这个设置值的对象直接在老年代分配，这样做的目的是避免在 Eden 区和及两个 Survivor 区之间发生大量的内存复制。注意！该参数只对 Serial 和 ParNew 收集器有效，Parallel Scavenge 并不认识该参数。
   一般我们代码中常见的大对象是指那种很长的字符串以及数组，写程序的时候应当避免，经常出现大对象容易导致内存还有不少空间时就提前触发垃圾收集以获取足够的内存空间来“安置”它们。

3. 长期存活的对象将进入老年代
   前面我们介绍对象创建的时候，说对象头中有一个 “GC 分代年龄” ，如果对象在 Eden 出生并经过第一次 Minor GC 后仍然存活，并且能被 Survivor 容纳的话，将被移动到 Survivor 空间中，并且对象年龄设为1。对象在 Survivor 空间中每“熬过”一次 Minor GC，年龄就增加 1 岁，当它的年龄到达一定程度（最大为 15 岁），就将会被晋升到老年代。对象晋升老年代的年龄阈值，可以通过参数 -XX:MaxTenuringThreshold 设置。
   对象是否能够晋升到老年代，也不全由 -XX:MaxTenuringThreshold 参数控制，如果 Survivor 空间中相同年龄的所有对象大小总和大于 Survivor 空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代。

4. 空间分配担保
   新生代在发生 Minor GC 之前，虚拟机会先检查老年代最大可用的连续空间是否大于新生代所有对象之和（或者历次晋升老年代对象的平均大小）。如果这个条件不成立，那么虚拟机将直接进行 Full GC 动作；如果这个条件成立，那么虚拟机就会进行一次 Minor GC 操作，但是这次 Minor GC 是有风险的，因为比较的值是平均值，可能出现极端的情况 —— 大量对象在 Minor GC 后还存活，这时就只好在失败后重新发起一次 Full GC。

常用JVM参数

修改 bin/catalina.sh
2核4g

```
export JAVA_OPTS="-server -Xms3550m -Xmx3550m -Xmn1330m -Xss256k -XX:+Aggress
iveOpts -XX:+UseBiasedLocking -XX:PermSize=512M -XX:MaxPermSize=1g -XX:+Disab
leExplicitGC -XX:MaxTenuringThreshold=31 -XX:+UseConcMarkSweepGC -XX:+UseParN
ewGC -XX:+CMSParallelRemarkEnabled -XX:+UseCMSCompactAtFullCollection -XX:+Us
eFastAccessorMethods -XX:CMSInitiatingOccupancyFraction=70 -Djava.awt.headles
s=true "
```

参数解析

```
-server 必须，服务模式

-Xms -Xmx jvm初始和最大堆内存

-Xmn 年轻代大小

-Xss 栈大小

-PermSize


-MaxPermSize


-AggressiveOpts JVM都会使用最新加入的优化技术（如果有的话）

-UseBiasedLocking 优化线程锁

垃圾回收使用-响应时间优先


-DisableExplicitGC 不响应 System.gc() 代码

-MaxTenuringThreshold 设置垃圾在年轻代的存活次数

-UseConcMarkSweepGC 设置年老代为并发收集

-UseParNewGC 设置年轻代为并发收集

-CMSParallelRemarkEnabled 在使用UseParNewGC 的情况下，尽量减少 mark 的时间

-UseCMSCompactAtFullCollection 在CMS垃圾收集后，进行一次内存碎片整理
-UseFastAccessorMethods get,set 方法转成本地代码

-UseCMSInitiatingOccupancyOnly
```

对象的强、软、弱和虚引用

在JDK 1.2以前的版本中，若一个对象不被任何变量引用，那么程序就无法再使用这个对象。也就是说，只有对象处于可触及（reachable）状态，程序才能使用它。从JDK 1.2版本开始，把对象的引用分为4种级别，从而使程序能更加灵活地控制对象的生命周期。这4种级别由高到低依次为：强引用、软引用、弱引用和虚引用。

1. 强引用（StrongReference）
   强引用是使用最普遍的引用。如果一个对象具有强引用，那垃圾回收器绝不会回收它。当内存空间不足，Java虚拟机宁愿抛出OutOfMemoryError错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足的问题。 ps：强引用其实也就是我们平时A a = new A()这个意思。
2. 软引用（SoftReference）
   如果一个对象只具有软引用，则内存空间足够，垃圾回收器就不会回收它；如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用。软引用可用来实现内存敏感的高速缓存
   软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收器回收，Java虚拟机就会把这个软引用加入到与之关联的引用队列中。
   SoftReference sf = new SoftReference(obj);
3. 弱引用（WeakReference）
   弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。不过，由于垃圾回收器是一个优先级很低的线程，因此不一定会很快发现那些只具有弱引用的对象
   弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java虚拟机就会把这个弱引用加入到与之关联的引用队列中。WeakReference wf = new WeakReference(obj);
4. 虚引用（PhantomReference）
   “虚引用”顾名思义，就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收器回收。
   虚引用主要用来跟踪对象被垃圾回收器回收的活动。虚引用与软引用和弱引用的一个区别在于：虚引用必须和引用队列 （ReferenceQueue）联合使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之 关联的引用队列中。
   PhantomReference pf = new PhantomReference(obj);

常见OOM及原因

堆溢出

- 错误
  java.lang.OutOfMemoryError:Java heap space
  这是最常见的OOM原因。
  堆中主要存放各种对象实例，还有常量池等结构。当JVM发现堆中没有足够的空间分配给新对象时，抛出该异常。具体来讲，在刚发现空间不足时，会先进行一次Full GC，如果GC后还是空间不足，再抛出异常。
- 原因主要有：
  - 无法在 Java 堆中分配对象
  - 应用程序保存了无法被GC回收的对象。
  - 应用程序过度使用 finalizer。
- 解决
  - 使用内存映像分析工具（如Eclipsc Memory Analyzer或者Jprofiler）对Dump出来的堆储存快照进行分析，分析清楚是内存泄漏还是内存溢出。
  - 如果是内存泄漏，可进一步通过工具查看泄漏对象到GC Roots的引用链，修复应用程序中的内存泄漏。
  - 如果不存在泄漏，先检查代码是否有死循环，递归等，再考虑用 -Xmx 增加堆大小。

- demo
  Exception in thread "main" java.lang.OutOfMemoryError: Java heap space

```
import java.util.ArrayList;
import java.util.List;
/**
 * JVM配置参数
 * -Xms20m    JVM初始分配的内存20m
 * -Xmx20m   JVM最大可用内存为20m
 * -XX:+HeapDumpOnOutOfMemoryError 当JVM发生OOM时，自动生成DUMP文件
 * -XX:HeapDumpPath=/Users/weihuaxiao/Desktop/dump/  生成DUMP文件的路径
 */
public class HeapOOM {
    static class OOMObject {
    }
    public static void main(String[] args) {
        List<OOMObject> list = new ArrayList<OOMObject>();
        //在堆中无限创建对象
        while (true) {
```

```
                list.add(new OOMObject());
            }
        }
    }
```

栈溢出

- 错误：
  - 如果线程请求的栈深度大于虚拟机所允许的深度，将抛出StackOverflowError 异常；
  - 如果虚拟机栈可以动态扩展，当扩展时无法申请到足够的内存时会抛出OutOfMemoryError 异常。
- 栈溢出原因
  - 在单个线程下，栈帧太大，或者虚拟机栈容量太小，当内存无法分配的时候，虚拟机抛出StackOverflowError 异常。
  - 不断地建立线程的方式会导致内存溢出。
- 栈溢出排查解决思路
  - 查找关键报错信息，确定是StackOverflowError还是OutOfMemoryError
  - 如果是StackOverflowError，检查代码是否递归调用方法等
  - 如果是OutOfMemoryError(Unable to create new native thread)，检查是否有死循环创建线程等，通过-Xss降低的每个线程栈大小的容量
- demo
  Exception in thread "main" java.lang.OutOfMemoryError: unable to create new native thread

```
/**
 * -Xss2M
 */
public class JavaVMStackOOM {
    private void dontStop(){
        while(true){
        }
    }
    public void stackLeakByThread(){
```

```
            while(true){
                Thread thread = new Thread(new Runnable(){
                    public void run() {
                        dontStop();
                    }
                });
                thread.start();}
        }
        public static void main(String[] args) {
            JavaVMStackOOM oom = new JavaVMStackOOM();
            oom.stackLeakByThread();
        }
    }
    Thread thread = new Thread(new Runnable(){
                    public void run() {
                        dontStop();
                    }
                });
```

方法区溢出

方法区，（又叫永久代，JDK8后，元空间替换了永久代），用于存放Class的相关信息，如类名、访问修饰符、常量池、字段描述、方法描述等。运行时产生大量的类，会填满方法区，造成溢出。

- 方法区溢出原因
  使用CGLib生成了大量的代理类，导致方法区被撑爆
  在Java7之前，频繁的错误使用String.intern方法
  大量jsp和动态产生jsp
  应用长时间运行，没有重启

- 方法区溢出排查解决思路
  检查是否永久代空间设置得过小
  检查代码是否频繁错误得使用String.intern方法
  检查是否跟jsp有关。
  检查是否使用CGLib生成了大量的代理类
  重启大法，重启JVM

demo
Caused by: java.lang.OutOfMemoryError: Metaspace

```
import org.springframework.cglib.proxy.Enhancer;
import org.springframework.cglib.proxy.MethodInterceptor;
import org.springframework.cglib.proxy.MethodProxy;

import java.lang.reflect.Method;

/**
 *  jdk8以上的话,
 *  虚拟机参数: -XX:MetaspaceSize=10M -XX:MaxMetaspaceSize=10M
 */
public class JavaMethodAreaOOM {
    public static void main(String[] args) {
        while (true) {
            Enhancer enhancer = new Enhancer();
            enhancer.setSuperclass(OOMObject.class);
            enhancer.setUseCache(false);
            enhancer.setCallback(new MethodInterceptor() {
                public Object intercept(Object obj, Method method,
                                        Object[] args, MethodProxy proxy)
 throws Throwable {
                    return proxy.invokeSuper(obj, args);
                }
            });
            enhancer.create();
        }
    }
    static class OOMObject {
    }
}
```

java.lang.OutOfMemoryError:Permgen space
jdk7中，方法区被实现在永久代中，错误原因同上。

JVM常用监控命令

jps

jps（JVM Process Status Tool），功能和ps类似：可以列出正在运行的虚拟机进程，并显示虚拟机执行主类（Main Class，main()函数所在的类）的名称，以及这些进程的本地虚拟机的唯一ID（LVMID，Local Virtual Machine Identifier）。虽然功能比较单一，但它是使用

频率最高的JDK命令行工具，因为其他JDK工具大多需要输入它查询到的LVMID来确定要监控的是哪一个虚拟机进程。对于本地虚拟机进程来说，LVMID与操作系统的进程ID（PID，Process Identifier）是一致的，使用Windows的任务管理器或Unix的ps命令也可以查询到虚拟机进程的LVMID，但如果使用了多个虚拟机进程，无法根据进程名称定位时，那就只能依赖jps命令显示主类的功能区才能区分了。

- option参数
  - -q 只显示pid，不显示class名称,jar文件名和传递给main 方法的参数。
  - -l 输出应用程序main class的完整package名 或者 应用程序的jar文件完整路径名。
  - -m 输出传递给main方法的参数
  - -v 输出传递给JVM的参数

jstat：虚拟机统计信息监控工具

jstat（JVM Statistics Monitoring Tool）是用于监控虚拟机各种运行状态信息的命令行工具。它可以显示本地或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，在没有GUI图像界面，只提高了纯文本控制台环境的服务器上，它将是运行期定位虚拟机性能问题的首选工具。

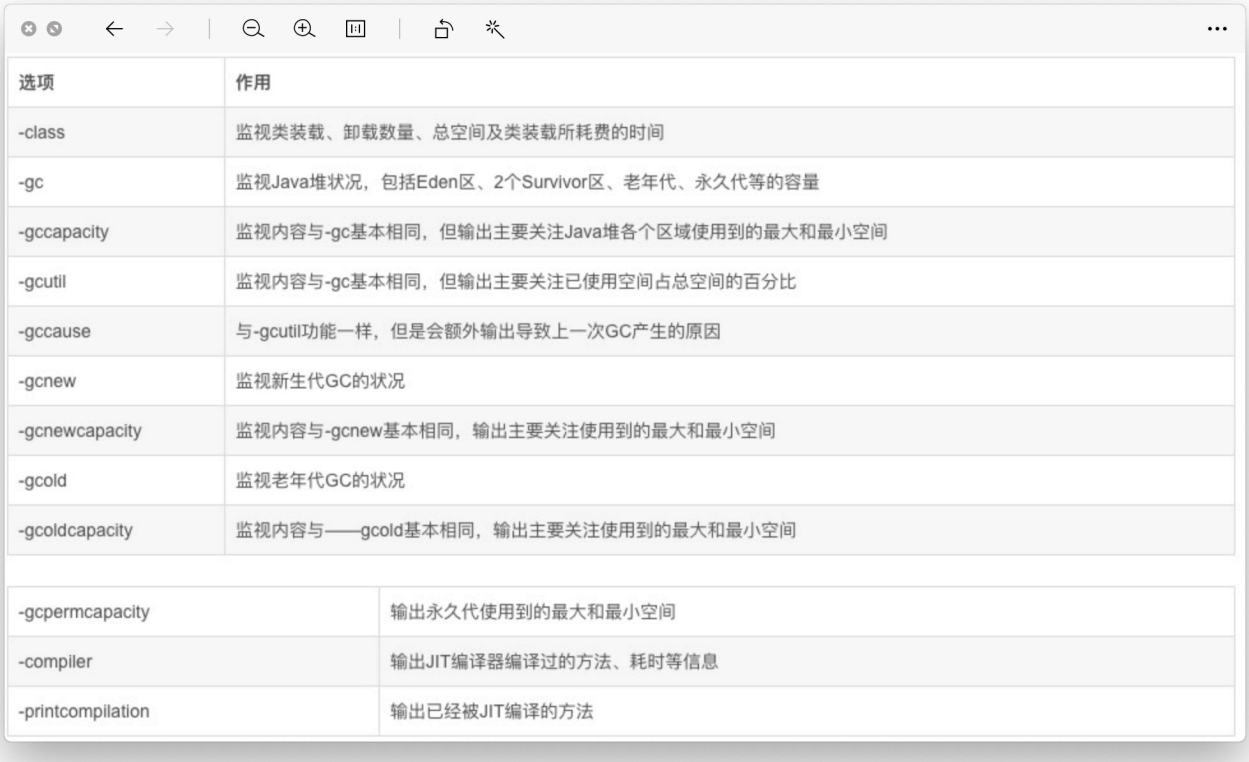
jstat命令格式：

jstat [option vmid [interval[s|ms] [count]]]

-- VMID与LVMID需要特别说明下：如果是本地虚拟机进程，VMID和LVMID是一致的，如果是远程虚拟机进程，那VMID的格式应当是：[protocol:][//] lvmid [@hostname[:port]/servername]

--参数interval和count代表查询间隔和次数，如果省略这两个参数，说明只查询一次。假设需要每250毫秒查询一次进程10524垃圾收集状况，一共查询5次

选项option代表这用户希望查询的虚拟机信息，主要分为3类：类装载、垃圾收集和运行期编译状况，具体选项及租用参见下表：



选项	作用
-class	监视类装载、卸载数量、总空间及类装载所耗费的时间
-gc	监视Java堆状况，包括Eden区、2个Survivor区、老年代、永久代等的容量
-gccapacity	监视内容与-gc基本相同，但输出主要关注Java堆各个区域使用到的最大和最小空间
-gcutil	监视内容与-gc基本相同，但输出主要关注已使用空间占总空间的百分比
-gccause	与-gcutil功能一样，但是会额外输出导致上一次GC产生的原因
-gcnew	监视新生代GC的状况
-gcnewcapacity	监视内容与-gcnew基本相同，输出主要关注使用到的最大和最小空间
-gcold	监视老年代GC的状况
-gcoldcapacity	监视内容与——gcold基本相同，输出主要关注使用到的最大和最小空间
-gcpermcapacity	输出永久代使用到的最大和最小空间
-compiler	输出JIT编译器编译过的方法、耗时等信息
-printcompilation	输出已经被JIT编译的方法

例子
-gcutil

```
jstat -gcutil 5828
  S0     S1     E      O      P     YGC     YGCT    FGC    FGCT     GCT
  0.00   0.00   1.82  52.18  99.91    329    4.894   269   80.244   85.13
9
```

查询结果表明：新生代Eden区（E，表示Eden）使用了1.82%的空间，两个Survivor区（S0、S1，表示Survivor0、Survivor1）里面都是空的，老年代（O，表示Old）和永久代（P,表示Permanent）则分别使用了52.18%和99.91%的空间。程序运行以来共发生Minor GC（YGC,Young GC）329次，总耗时（YGCT，Young GC Time）4.894秒，发生Full GC（FGC）269次，总耗时（FGCT）80.244秒，所有GC总耗时（GCT）85.139秒。

-class
监视类装载、卸载数量、总空间以及耗费的时间

```
$ jstat -class 11589
 Loaded  Bytes  Unloaded  Bytes     Time
```

```
   7035  14506.3     0     0.0       3.67
```

Loaded : 加载class的数量
Bytes : class字节大小
Unloaded : 未加载class的数量
Bytes : 未加载class的字节大小
Time : 加载时间

jmap

jmap(JVM Memory Map)命令用于生成heap dump文件，如果不使用这个命令，还可以使用-XX:+HeapDumpOnOutOfMemoryError参数来让虚拟机出现OOM的时候·自动生成dump文件。
jmap不仅能生成dump文件，还阔以查询finalize执行队列、Java堆和永久代的详细信息，如当前使用率、当前使用的是哪种收集器等。

option参数
dump : 生成堆转储快照
finalizerinfo : 显示在F-Queue队列等待Finalizer线程执行finalizer方法的对象
heap : 显示Java堆详细信息
histo : 显示堆中对象的统计信息

dump堆到文件,format指定输出格式，live指明是活着的对象,file指定文件名,dump.hprof这个后缀是为了后续可以直接用MAT(Memory Anlysis Tool)打开。

```
$ jmap -dump:live,format=b,file=dump.hprof 28920
  Dumping heap to /home/xxx/dump.hprof ...
  Heap dump file created
```

打印等待回收对象的信息

```
$ jmap -finalizerinfo 28920
  Attaching to process ID 28920, please wait...
  Debugger attached successfully.
  Server compiler detected.
  JVM version is 24.71-b01
  Number of objects pending for finalization: 0
```

-heap

打印heap的概要信息，GC使用的算法，heap的配置及wise heap的使用情况,可以用此来判断内存目前的使用情况以及垃圾回收情况

```
$ jmap -heap 28920
  Attaching to process ID 28920, please wait...
  Debugger attached successfully.
  Server compiler detected.
  JVM version is 24.71-b01

  using thread-local object allocation.
  Parallel GC with 4 thread(s)//GC 方式

  Heap Configuration: //堆内存初始化配置
     MinHeapFreeRatio = 0 //对应jvm启动参数-XX:MinHeapFreeRatio设置JVM堆最小空闲比率(default 40)
     MaxHeapFreeRatio = 100 //对应jvm启动参数 -XX:MaxHeapFreeRatio设置JVM堆最大空闲比率(default 70)
     MaxHeapSize      = 2082471936 (1986.0MB) //对应jvm启动参数-XX:MaxHeapSize=设置JVM堆的最大大小
     NewSize          = 1310720 (1.25MB)//对应jvm启动参数-XX:NewSize=设置JVM堆的‘新生代’的默认大小
     MaxNewSize       = 17592186044415 MB//对应jvm启动参数-XX:MaxNewSize=设置JVM堆的‘新生代’的最大大小
     OldSize          = 5439488 (5.1875MB)//对应jvm启动参数-XX:OldSize=<value>:设置JVM堆的‘老生代’的大小
     NewRatio         = 2 //对应jvm启动参数-XX:NewRatio=:‘新生代’和‘老生代’的大小比率
     SurvivorRatio    = 8 //对应jvm启动参数-XX:SurvivorRatio=设置年轻代中Eden区与Survivor区的大小比值
     PermSize         = 21757952 (20.75MB)  //对应jvm启动参数-XX:PermSize=<value>:设置JVM堆的‘永生代’的初始大小
     MaxPermSize      = 85983232 (82.0MB)//对应jvm启动参数-XX:MaxPermSize=<value>:设置JVM堆的‘永生代’的最大大小
     G1HeapRegionSize = 0 (0.0MB)

  Heap Usage://堆内存使用情况
  PS Young Generation
  Eden Space://Eden区内存分布
     capacity = 33030144 (31.5MB)//Eden区总容量
     used     = 1524040 (1.4534378051757812MB)  //Eden区已使用
     free     = 31506104 (30.04656219482422MB)  //Eden区剩余容量
     4.614088270399305% used //Eden区使用比率
```

```
  From Space:  //其中一个Survivor区的内存分布
     capacity = 5242880 (5.0MB)
     used     = 0 (0.0MB)
     free     = 5242880 (5.0MB)
     0.0% used
  To Space:  //另一个Survivor区的内存分布
     capacity = 5242880 (5.0MB)
     used     = 0 (0.0MB)
     free     = 5242880 (5.0MB)
     0.0% used
  PS Old Generation //当前的Old区内存分布
     capacity = 86507520 (82.5MB)
     used     = 0 (0.0MB)
     free     = 86507520 (82.5MB)
     0.0% used
  PS Perm Generation//当前的 “永生代” 内存分布
     capacity = 22020096 (21.0MB)
     used     = 2496528 (2.3808746337890625MB)
     free     = 19523568 (18.619125366210938MB)
     11.337498256138392% used

  670 interned Strings occupying 43720 bytes.
```

打印堆的对象统计，包括对象数、内存大小等等 （因为在dump:live前会进行full gc，如果带上live则只统计活对象，因此不加live的堆大小要大于加live堆的大小 ）

```
  $ jmap -histo:live 28920 | more
 num     #instances         #bytes  class name
----------------------------------------------
   1:         83613       12012248  <constMethodKlass>
   2:         23868       11450280  [B
   3:         83613       10716064  <methodKlass>
   4:         76287       10412128  [C
   5:          8227        9021176  <constantPoolKlass>
   6:          8227        5830256  <instanceKlassKlass>
   7:          7031        5156480  <constantPoolCacheKlass>
   8:         73627        1767048  java.lang.String
   9:          2260        1348848  <methodDataKlass>
  10:          8856         849296  java.lang.Class
   ....
```

jmap -dump这个命令执行，JVM会将整个heap的信息dump写入到一个文件，heap如果比

较大的话，就会导致这个过程比较耗时，并且执行的过程中为了保证dump的信息是可靠的，所以会暂停应用。
jmap -histo:live 这个命令执行，JVM会先触发gc，然后再统计信息。

jstack

jstack用于生成java虚拟机当前时刻的线程快照。线程快照是当前java虚拟机内每一条线程正在执行的方法堆栈的集合，生成线程快照的主要目的是定位线程出现长时间停顿的原因，如线程间死锁、死循环、请求外部资源导致的长时间等待等。 线程出现停顿的时候通过jstack来查看各个线程的调用堆栈，就可以知道没有响应的线程到底在后台做什么事情，或者等待什么资源。 如果java程序崩溃生成core文件，jstack工具可以用来获得core文件的java stack和native stack的信息，从而可以轻松地知道java程序是如何崩溃和在程序何处发生问题。另外，jstack工具还可以附属到正在运行的java程序中，看到当时运行的java程序的java stack和native stack的信息, 如果现在运行的java程序呈现hung的状态，jstack是非常有用的。

option参数
-F : 当正常输出请求不被响应时，强制输出线程堆栈
-l : 除堆栈外，显示关于锁的附加信息
-m : 如果调用到本地方法的话，可以显示C/C++的堆栈

示例

```
$ jstack -l 11494|more
2016-07-28 13:40:04
Full thread dump Java HotSpot(TM) 64-Bit Server VM (24.71-b01 mixed mode)
:

"Attach Listener" daemon prio=10 tid=0x00007febb0002000 nid=0x6b6f waitin
g on condition [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE

   Locked ownable synchronizers:
        - None

"http-bio-8005-exec-2" daemon prio=10 tid=0x00007feb94028000 nid=0x7b8c w
aiting on condition [0x00007fea8f56e000]
   java.lang.Thread.State: WAITING (parking)
        at sun.misc.Unsafe.park(Native Method)
        - parking to wait for  <0x00000000cae09b80> (a java.util.concurre
nt.locks.AbstractQueuedSynchronizer$ConditionObject)
```

```
        at java.util.concurrent.locks.LockSupport.park(LockSupport.java:1
86)
        at java.util.concurrent.locks.AbstractQueuedSynchronizer$Conditio
nObject.await(AbstractQueuedSynchronizer.java:2043)
        at java.util.concurrent.LinkedBlockingQueue.take(LinkedBlockingQu
eue.java:442)
        at org.apache.tomcat.util.threads.TaskQueue.take(TaskQueue.java:1
04)
        at org.apache.tomcat.util.threads.TaskQueue.take(TaskQueue.java:3
2)
        at java.util.concurrent.ThreadPoolExecutor.getTask(ThreadPoolExec
utor.java:1068)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolEx
ecutor.java:1130)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolE
xecutor.java:615)
        at org.apache.tomcat.util.threads.TaskThread$WrappingRunnable.run
(TaskThread.java:61)
        at java.lang.Thread.run(Thread.java:745)

   Locked ownable synchronizers:
        - None
       .....
```

dump 文件里，值得关注的线程状态有：
死锁，Deadlock（重点关注）
执行中，Runnable

等待资源，Waiting on condition（重点关注）
等待获取监视器，Waiting on monitor entry（重点关注）
暂停，Suspended
对象等待中，Object.wait() 或 TIMED_WAITING
阻塞，Blocked（重点关注）
停止，Parked

jinfo： Java配置信息工具

jinfo(JVM Configuration info)这个命令作用是实时查看和调整虚拟机运行参数。
之前的jps -v口令只能查看到显示指定的参数，如果想要查看未被显示指定的参数的值就要使用jinfo口令

option参数
-flag : 输出指定args参数的值
-flags : 不需要args参数，输出所有JVM参数的值
-sysprops : 输出系统属性，等同于System.getProperties()

执行样例：查询CMSInitiatingOccupancyFraction参数值。
jinfo -flag CMSInitiatingOccupancyFraction 5828
-XX:CMSInitiatingOccupancyFraction=-1

CPU占用过高问题排查

i. 查看系统状况
top 命令查看CPU、内存等使用情况

```
# top
top - 14:52:54 up 514 days,  7:00,  8 users,  load average: 2.85, 1.35
, 1.62
Tasks: 147 total,   1 running, 146 sleeping,   0 stopped,   0 zombie
Cpu(s): 57.6%us,  6.3%sy,  0.0%ni,  9.2%id, 26.2%wa,  0.0%hi,  0.0%si,
  0.7%st
Mem:   3922928k total,  3794232k used,   128696k free,   403112k buffe
rs
Swap:  4194296k total,    65388k used,  4128908k free,  1492204k cache
d
PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND

6764 root      20   0 2428m 1.1g  11m S 190.0 28.3  36:38.55 java

1161 root      20   0     0    0    0 D  0.3  0.0  32:43.06 flush-253:
0
```

从top命令的结果发现。pid为6764的java进程CPU利用持续占用过高,达到了190%。内存占用率为28.3%。

ii. 定位问题线程
使用ps -mp pid -o THREAD,tid,time命令查看该进程的线程情况，发现该进程的两个线程占用率很高

```
# ps -mp 6764 -o THREAD,tid,time
USER     %CPU PRI SCNT WCHAN  USER SYSTEM   TID     TIME
root     71.7   -    - -         -      -     - 00:36:52
root      0.0  19    - futex_    -      -  6764 00:00:00
root      0.0  19    - poll_s    -      -  6765 00:00:01
root     44.6  19    - futex_    -      -  6766 00:23:32
root     44.6  19    - futex_    -      -  6767 00:23:32
root      1.2  19    - futex_    -      -  6768 00:00:38
```

从上面可以看出6766和6767两个线程占用CPU大约有半个小时，每个线程的CPU利用率约为45%。接下来需要查看对应线程的问题堆栈

iii. 查看问题线程堆栈
将线程id转换为16进制

```
#  printf "%x\n" 6766
1a6e
```

iv. jstack查看线程堆栈信息
jstack命令打印线程堆栈信息，命令格式：jstack pid |grep tid

```
jstack 6764 | grep 1a6e
"GC task thread#0 (ParallelGC)" prio=10 tid=0x00007ffeb8016800 nid=0x1
a6e runnable
"GC task thread#0 (ParallelGC)" prio=10 tid=0x00007ffeb8016800 nid=0x1
a6e runnable
"GC task thread#1 (ParallelGC)" prio=10 tid=0x00007ffeb8016800 nid=0x1
a6e runnable
"VM Periodic Task Thread" prio=10 tid=0x00007ffeb8016800 nid=0x3700 wa
iting on condition
```

可以看出都是GC的线程。那么可以推断，很有可能就是内存不够导致GC不断执行。接下来我们就需要查看gc 内存的情况

v. jstat查看进程内存状况
命令: jstat -gcutil

```
# jstat -gcutil 6764 2000 10
S0     S1     E      O      P     YGC     YGCT    FGC    FGCT     GCT
```

```
0.00   0.00  100.00 100.00  97.74   1863   33.937   310  453.788  487.
726
0.00   0.00  100.00 100.00  97.74   1863   33.937   310  453.788  487.
726
0.00   0.00  100.00 100.00  97.74   1863   33.937   310  453.788  487.
726
0.00   0.00  100.00 100.00  97.74   1863   33.937   310  453.788  487.
726
```

vi. jstack 和 jmap 分析进程堆栈和内存状况

使用jmap命令导出heapdump文件，然后拿到本地使用mat、jprofiler等工具分析。

命令: jmap [option] vmid

jmap -dump:format=b,file=dump.bin 6764

命令: jstack [option] vmid

jstack -l 6764 >> jstack.out

从heapdump文件中定位到程序中的工作现场，和内存状况，如下：

线程: