一、安全点：Stop the world

1、Stop the world

GC 操作的某些阶段，如可达性分析遍历 GC ROOT 节点找引用链，需要确保在一致性的快照中进行（一致性的意思指分析期间整个执行系统好像冻结在某个时间节点上，不可以出现分析过程中对象引用关系还在不断变化的情况，该点不满足分析的准确性就无法保证）。**这是导致GC进行时必须停顿所有的执行线程的其中一个重要原因。（Sun 将这件事情称之为 “Stop the world”。）**即使是在号称几乎不会发生停顿的CMS收集器中，枚举根节点时也是必须停顿的。

2、OopMap (*选学)

OopMap：在所有执行线程停顿下来后，虚拟机并不需要逐个检查每个栈的局部变量表来查找引用的位置，而应当有办法直接得知哪些地方存着对象的应用。在执行到某条指令时，栈中什么位置存放了什么变量是确定的，在编译期间虚拟机就可以计算出这些信息。在 HotSpot 的实现中，用一组称为 OopMap 的数据结构记录了某一些执行节点哪些位置是引用，这样，在GC扫描时就可以直接获取这些信息了。

3、SafePoint (*选学)

在 OopMap 的协助下，HotSpot 可以快速且准确地完成 GC Roots 枚举。但是随着程序的运行，引用关系会随之变化，虚拟机不可能为每一个执行节点都生成 OopMap 来记录引用关系，那将需要大量额外的空间，GC的成本将会变得很高。实际上，HotSpot虚拟机也的确没有为每一条指令都生成OopMap，只是在特定位置记录这些信息，这些位置称为“安全点”（Safepoint），即程序执行时并非在所有地方都停顿下来开始GC，只有在到达安全点时才能暂停。

Safepoint既不能选的太少导致GC等待太长时间，也不能过于频繁以至于过分增大运行时的负荷。安全点一般选取在方法调用、循环跳转、异常跳转等让程序长时间执行的指令。

对于Safepoint，另一个需要考虑的问题是如何在GC发生时让所有线程（这里不包括执行JNI调用的线程）都“跑”到最近的安全点上再停顿下来。主动式中断的思想是当GC需要中断线程的时候，不直接对线程操作，仅仅简单的设置一个标志，各线程执行时主动去轮询这个标志，当发现中断标志为真时就自己中断挂起。轮询标志的位置和安全点是重合的，另外再加上创建对象需要分配内存的地方。

4、SafeRegion (*选学)

Safepoint机制保证了程序执行时，在不太长的时间内就会遇到可进入GC的Safepoint。但是程序不执行的时候呢？不执行就是程序没有分配到CPU时间，典型的例子就是线程处于Sleep状态或Blocked状态，这时候线程无法响应JVM的中断请求，“走”到安全的地方再挂起。JVM显然也不会等待线程重新分配CPU时间。对于这种情况就需要安全区域（Safe Region）来解决。

安全区域是指在一段代码片段中引用关系不会发生变化，在这个区域的任意地方开始GC都是安全的。我们也把Safe Region称为扩展的Safepoint。

在线程执行到Safe Region中的代码的时候，首先标识自己进入了Safe Region，这样，当在这段时间里JVM要发起GC时，就不用考虑标识自己为Safe Region状态的线程了。在线程要离开Safe Region时，它要检查系统是否已经完成了根节点枚举（）或者整个GC过程如果完成了，那线程就继续执行，否则它就必须等待收到可以安全离开Safe Region的信号为止。

二、垃圾收集器

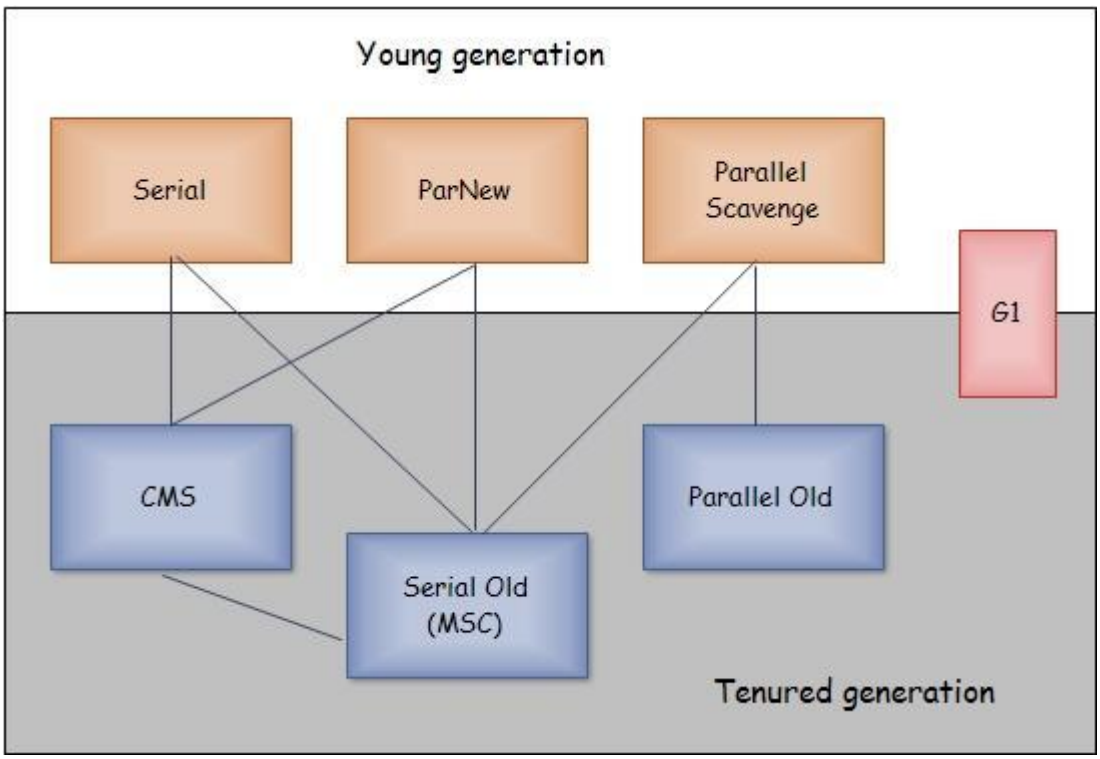
参考博客：https://blog.csdn.net/tjiyu/article/details/53983650

垃圾收集器是垃圾回收算法（标记-清除算法、复制算法、标记-整理算法）的具体实现，不同商家、不同版本的JVM所提供的垃圾收集器可能会有很在差别，下面主要介绍HotSpot虚拟机中的垃圾收集器。

本节介绍这些收集器的特性、基本原理和使用场景。没有最好的收集器，更没有万能的收集，选择的只能是适合具体应用场景的收集器。

1、垃圾收集器组合

JDK7/8后，HotSpot虚拟机所有收集器及组合（连线），如下图：



- 图中展示了**7种**不同分代的收集器：

Serial、ParNew、Parallel Scavenge、Serial Old、Parallel Old、CMS、G1；

- 它们所处区域，表明其是属于新生代收集器还是老年代收集器：

  新生代收集器：Serial、ParNew、Parallel Scavenge

  老年代收集器：Serial Old、Parallel Old、CMS

  整堆收集器：G1

- 两个收集器间有连线，表明它们**可以搭配使用**：

Serial/Serial Old、Serial/CMS、ParNew/Serial Old、ParNew/CMS、Parallel Scavenge/Serial Old、Parallel Scavenge/Parallel Old、G1；

- 其中Serial Old作为CMS出现**"Concurrent Mode Failure"**失败的后备预案。

2、 Minor GC 和 Full GC

- **Minor GC**

又称**新生代GC**，指发生在新生代的垃圾收集动作；

因为Java对象大多是朝生夕灭，所以Minor GC非常频繁，一般回收速度也比较快；

- **Full GC**

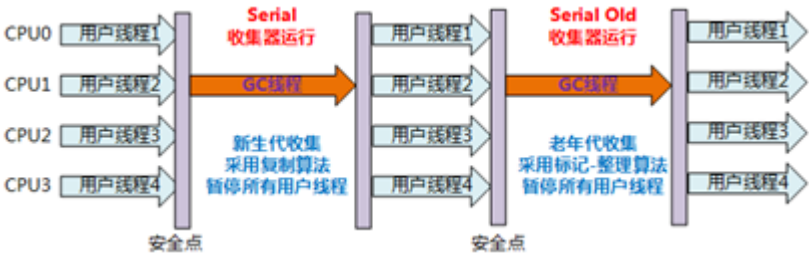
又称**Major GC或老年代GC**，指发生在老年代的GC；

出现Full GC经常会伴随至少一次的Minor GC（不是绝对，Parallel Sacvenge收集器就可以选择设置Major GC策略）；

Major GC速度一般比Minor GC慢10倍以上；

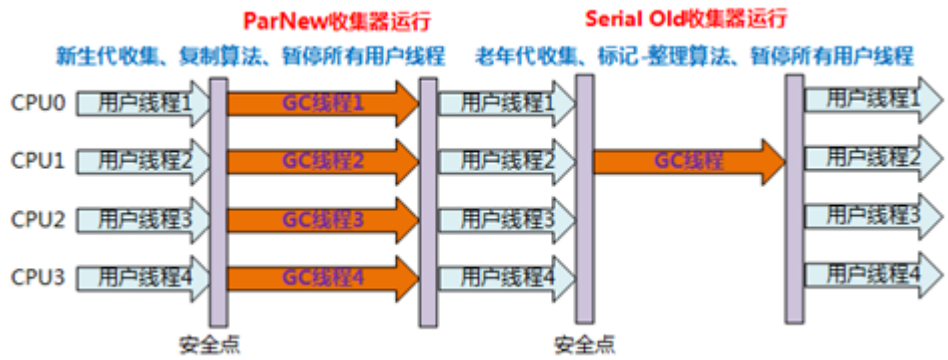
3、Serial 收集器(*选学)

- Serial（串行）垃圾收集器是最基本、发展历史最悠久的收集器；JDK1.3.1前是HotSpot新生代收集的唯一选择。
- 特点： 针对新生代；采用复制算法； 单线程收集；进行垃圾收集时，必须暂停所有工作线程，直到完成。
- 应用场景：依然是HotSpot在Client模式下默认的新生代收集器；也有优于其他收集器的地方：简单高效（与其他收集器的单线程相比）；对于限定单个CPU的环境来说，Serial收集器没有线程交互（切换）开销，可以获得最高的单线程收集效率；在用户的桌面应用场景中，可用内存一般不大（几十M至一两百M），可以在较短时间内完成垃圾收集（几十MS至一百多MS）,只要不频繁发生，这是可以接受的。
- Serial/Serial Old组合收集器运行示意图如下：



4、ParNew收集器(*选学)

- ParNew垃圾收集器是Serial收集器的多线程版本。
- 除了多线程外，其余的行为、特点和Serial收集器一样；如Serial收集器可用控制参数、收集算法、Stop The World、内存分配规则、回收策略等；两个收集器共用了不少代码；
- 应用场景：在Server模式下，ParNew收集器是一个非常重要的收集器，因为除Serial外，目前只有它能与CMS收集器配合工作；但在单个CPU环境中，不会比Serail收集器有更好的效果，因为存在线程交互开销。
- ParNew/Serial Old组合收集器运行示意图如下：



5、Parallel Scavenge收集器(*选学)

- Parallel Scavenge垃圾收集器因为与吞吐量关系密切，也称为吞吐量收集器（Throughput Collector）。
- 特点：有一些特点与 ParNew 收集器相似，新生代收集器；采用复制算法；多线程收集。主要特点是：**Parallel Scavenge 收集器的目标则是达一个可控制的吞吐量（Throughput），系统通过调节新生老生代空间比例等，来调节吞吐量。**
- Parallel Scavenge 收集器提供了两个参数用来精确控制吞吐量，分别是控制最大垃圾收集停顿时间的-XX:MaxGCPauseMillis，以及直接设置吞吐量大小 -XXGCTimeRatio。停顿时间和吞吐量存在相互制约的关系：GC停顿时间缩短是以牺牲吞吐量和新生代空间来换取的：系统把新生代调小一点，收集300MB新生代肯定比收集500MB快吧，这也直接导致垃圾收集发生的更频繁一些，原来10收集一次，现在变成5秒收集一次，每次停顿70毫秒，停顿时间的确是下降了，但吞吐量也降下来了。
- 应用场景： 高吞吐量为目标，即减少垃圾收集时间，让用户代码获得更长的运行时间；当应用程序运行在具有多个CPU上，对暂停时间没有特别高的要求时，即程序主要在后台进行计算，而不需要与用户进行太多交互；例如，那些执行批量处理、订单处理、工资支付、科学计算的应用程序；
- 吞吐量与收集器关注点说明

> > （**A**）、吞吐量（**Throughput**）
> >
> > 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）；
> >
> > 高吞吐量即减少垃圾收集时间，让用户代码获得更长的运行时间；
> >
> > （**B**）、垃圾收集器期望的目标（关注点）
> >
> > > （**1**）、停顿时间
> > >
> > > 停顿时间越短就适合需要与用户交互的程序；
> > >
> > > 良好的响应速度能提升用户体验；
> > >
> > > （**2**）、吞吐量
> > >
> > > 高吞吐量则可以高效率地利用CPU时间，尽快完成运算的任务；
> > >
> > > 主要适合在后台计算而不需要太多交互的任务；
> > >
> > > （**3**）、覆盖区（**Footprint**）
> > >
> > > 在达到前面两个目标的情况下，尽量减少堆的内存空间；
> > >
> > > 可以获得更好的空间局部性；

6、Serial Old收集器(*选学)

- Serial Old是 Serial收集器的老年代版本；
- 针对老年代；采用"标记-整理"算法（还有压缩，Mark-Sweep-Compact）；单线程收集；
- 应用场景：主要用于Client模式；而在Server模式有两大用途：（A）、在JDK1.5及之前，与Parallel Scavenge收集器搭配使用（JDK1.6有Parallel Old收集器可搭配）；（B）、作为CMS收集器的后备预案，在并发收集发生Concurrent Mode Failure时使用（后面详解）；

7、Parallel Old收集器(*选学)

- Parallel Old垃圾收集器是Parallel Scavenge收集器的老年代版本； JDK1.6中才开始提供；
- 针对老年代；采用"标记-整理"算法；多线程收集；
- JDK1.6及之后用来代替老年代的Serial Old收集器；特别是在Server模式，多CPU的情况下；这样在注重吞吐量以及CPU资源敏感的场景，就有了Parallel Scavenge加Parallel Old收集器的"给力"应用组合；

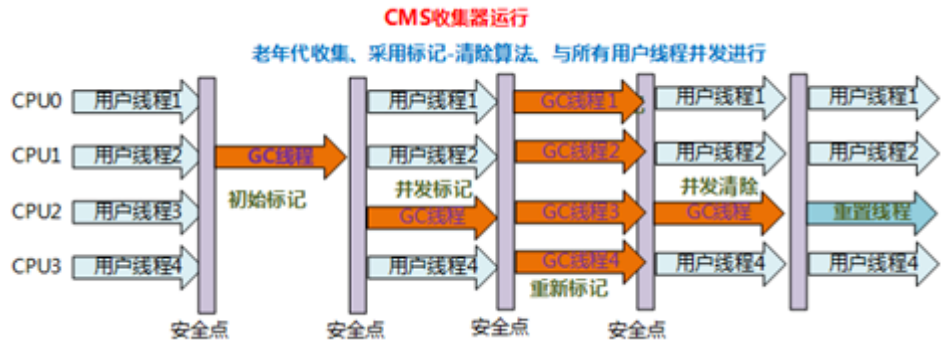
8、CMS收集器(**选学)

- 并发标记清理（Concurrent Mark Sweep，CMS）收集器也称为并发低停顿收集器（Concurrent Low Pause Collector）或低延迟（low-latency）垃圾收集器；**CMS等收集器的关注点是尽可能地缩短垃圾收集时用户线程的停顿时间；**
- 特点：针对老年代；基于"标记-清除"算法(不进行压缩操作，产生内存碎片)；以获取最短回收停顿时间为目标；并发收集、低停顿；需要更多的内存；是HotSpot在JDK1.5推出的第一款真正意义上的并发（Concurrent）收集器；第一次实现了让垃圾收集线程与用户线程（基本上）同时工作；
- 缺点：

  （A）对CPU资源非常敏感：并发收集虽然不会暂停用户线程，但因为占用一部分CPU资源，还是会导致应用程序变慢，总吞吐量降低。

  （B）无法处理浮动垃圾,可能出现"Concurrent Mode Failure"失败：在并发清除时，用户线程新产生的垃圾，称为浮动垃圾；这使得并发清除时需要预留一定的内存空间，不能像其他收集器在老年代几乎填满再进行收集；如果CMS预留内存空间无法满足程序需要，就会出现一次"Concurrent Mode Failure"失败；这时JVM启用后备预案：临时启用Serail Old收集器，而导致另一次Full GC的产生；

  （C）产生大量内存碎片：由于CMS基于"标记-清除"算法，清除后不进行压缩操作；
- 适用场景：与用户交互较多的场景，希望系统停顿时间最短，注重服务的响应速度；以给用户带来较好的体验；如**常见WEB、B/S系统的服务器上的应用；**
- CMS收集器运行示意图如下：



9、G1收集器(**选学)

- G1（Garbage-First）是JDK7-u4才推出商用的收集器；
- 特点：

  （A）并行与并发：能充分利用多CPU、多核环境下的硬件优势；可以并行来缩短"Stop The World"停顿时间；也可以并发让垃圾收集与用户程序同时进行；

  （B）分代收集，收集范围包括新生代和老年代：能独立管理整个GC堆（新生代和老年代），而不需要与其他收集器搭配；能够采用不同方式处理不同时期的对象；虽然保留分代概念，但Java堆的内存布局有很大差别；将整个堆划分为多个大小相等的独立区域（Region）；新生代和老年代不再是物理隔离，它们都是一部分Region（不需要连续）的集合；

  （C）结合多种垃圾收集算法，空间整合，不产生碎片。从整体看，是基于标记-整理算法；从局部（两个Region间）看，是基于复制算法；这是一种类似火车算法的实现；

  （D）可预测的停顿：低停顿的同时实现高吞吐量G1除了追求低停顿处，还能建立可预测的停顿时间模型；可以明确指定M毫秒时间片内，垃圾收集消耗的时间不超过N毫秒；

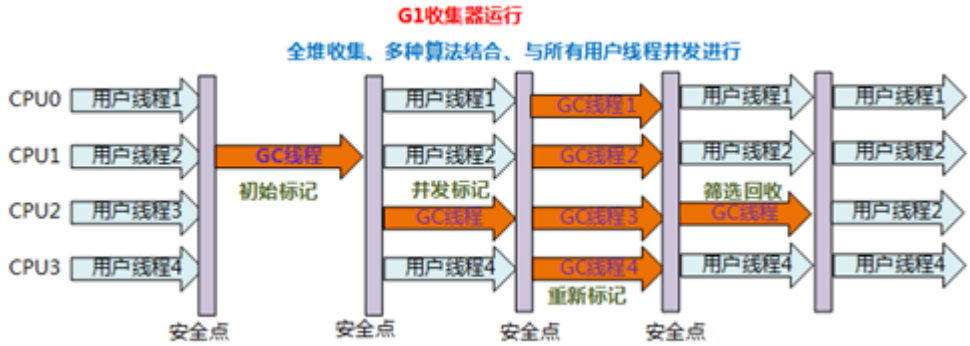
- 应用场景：面向服务端应用，针对具有大内存、多处理器的机器；最主要的应用是为需要低GC延迟，并具有大堆的应用程序提供解决方案；如：在堆大小约6GB或更大时，可预测的暂停时间可以低于0.5秒； 用来替换掉JDK1.5中的CMS收集器；在下面的情况时，使用G1可能比CMS好：

  （1）超过50%的Java堆被活动数据占用；

  （2）对象分配频率或年代提升频率变化很大；

  （3）GC停顿时间过长（长于0.5至1秒）。 是否一定采用G1呢？也未必：如果现在采用的收集器没有出现问题，不用急着去选择G1；如果应用程序追求低停顿，可以尝试选择G1；是否代替CMS需要实际场景测试才知道。

- G1 收集器运行示意图如下：



三、内存分配与回收策略

- 对象优先在 **Eden** 分配
- **大对象直接进入老年代**：所谓的大对象是指，需要大量连续内存空间的Java对象，最典型的大对象就是那种很长的字符串以及数组，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连续内存空间来存放他们。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区以及两个Survivor区之间发生大量的内存复制（新生代使用复制算法收集内存）。
- **长期存活的对象将进入老年代**：虚拟机给每个对象定义了一个对象年龄计数器。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所容纳的话，将被移动到Survivor区，并且对象年龄设置为1。对象在Survivor区中没“熬过”一次Minor GC，年龄就增加1，当它的年龄增加到一定程度（默认15岁），就将会晋升到老年代中。对象晋升到老年代的年龄阀值，可以通过参数-XX:MaxTenuringThreshold设置。
- **动态对象年龄判定**：为了能更好的适应不同程序的内存状况，虚拟机并不是永远地要求对象的年龄必须达到了MaxTenuringThreshold才能晋升到老年代，如果在Survivor空间中相同年龄所有对象大小的总和大于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，无须等到MaxTenuringThreshold中要求的年龄。
- **空间分配担保**：在发生Minor GC之前，虚拟机会首先检查老年代最大可用连续空间是否大于新生代所有对象总空间，如果这个条件成立，那么Minor GC可用确保是安全的。如果不成立，则虚拟机会查看HandlePromotionFailure设置值是否允许担保失败。如果允许，那么会继续检查老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小，如果大于，将尝试第一次Minor GC，即使这次Minor GC是有风险的；如果小于，或者HandlePromotionFailure设置为不允许冒险，那这时也要改为进行一次Full GC。如果出现HandlePromotionFailure失败，那就只好在失败之后重新发起一次Full GC。虽然担保失败时绕的圈子是最大的，但大部分情况下都还是会将HandlePromotionFailure开关打开，避免Full GC过于频繁。

四、调优案例

（一）高性能硬件上的程序部署策略

1、问题描述：

例如，一个15万PV/天左右的在线文档类型网站最近更换了硬件系统，新的硬件为4个CPU、16GB物理内存，操作系统为 64 位 CentOS5.4，Resin作为Web服务器。整个服务器暂时没有部署别的应用，所有硬件资源都可以提供给这访问量不算太大的网站使用。管理人员为了尽量利用硬件资源选用了 64 位的 JDK1.5，并通过 -Xms 和 -Xmx 参数将 Java 堆固定在 12GB。使用一段时间后发现使用效果并不理想，网站经常不定期出现长时间失去响应的情况。

2、问题分析：

监控服务器运行状况后发现网站失去响应是由 GC 停顿造成的，虚拟机运行在Server模式，默认使用吞吐量优先收集器，回收 12GB 的堆，一次 Full GC 的停顿时间高达 14 秒。并且由于程序设计的关系，访问文档时要把文档从磁盘提取到内存中，导致内存中出现很多由于文档序列化产生的大对象，这些大对象很多都进入了老年代，没有在 Minor GC 中清理掉。这种情况即使有 12GB 的堆，内存很快也会被耗尽，由此导致每隔十几分钟甚至十几秒的停顿，领网站开发人员和管理人员感到很沮丧。

这里暂不讨论代码的问题，程序部署上的问题显然是过大的堆内存进行回收时导致长时间的停顿，硬件升级前使用 32 位系统 1.5GB 的堆，用户只感觉到使用网站比较缓慢，但不会出现十分明显的停顿，因此才考虑升级硬件以提升程序效能，如果重新缩小给Java堆分配的内存，那么硬件上的资源就显得很浪费。

总结：

> （1）管理员为程序分配了超大堆（12G），超大堆 Full GC 时停顿时间长。
>
> （2）文档序列化频繁产生大对象，导致 Full GC 频繁，不定期出现长时间失去响应的情况。

3、方案分析：

在高性能硬件上部署程序，目前主要有两种方式：

- 使用 64 位 JDK 来使用大内存。
- 使用若干个32位虚拟机建立逻辑集群来利用硬件资源。

此案例中的管理员使用了第一种方式。对于用户交互性强、对停顿时间敏感的系统，可以给Java虚拟机分配超大堆的前提是有把握把应用程序的 FullGC 频率控制的足够低，至少要低到不会影响用户使用，比如十几小时乃至一天才出现一次Full GC，这样可以通过在深夜执行定时任务的方式触发 Full GC 甚至自动重启应用服务器来保持内存可用空间在一个稳定的水平。

控制 Full GC 频率的关键是看应用中绝大多数对象是否符合“朝生夕死”的原则，即大多数对象的生存时间不应太长，尤其是不能有成批量的、长生存时间的大对象产生，这样才能保证老年代空间的稳定。在大多数网站形式的应用里，主要对象的生存周期都应该是请求级或者页面级的，会话级和全局级的长生命对象相对较少。只要代码写的合理，应当都能实现在超大堆中正常使用而没有Full GC，这样的话，使用超大堆内存时，网站响应速度才会有保证。

除此之外，如果计划使用64位的JDK来管理大内存，还需要考虑下面可能遇到的问题：

- 内存回收导致的长时间停顿。
- 现阶段64位JDk的性能测试结果普遍低于32位JDK。
- 需要保证程序足够稳定，因为这种应用要是产生堆溢出几乎就无法产生堆存储快照（因为要产生十几GB乃至更大的Dump文件），哪怕产生了快照也几乎无法进行分析。

- 相同程序在64位JDK消耗的内存一般比32位JDK大，这是由于指针膨胀，以及数据类型对齐补白等因素造成的。

上面的问题听起来有点吓人，所以现阶段不少管理员还是选择第二种方式：使用若干个32位虚拟机建立逻辑集群来利用硬件资源。具体做法是在一台物理机上启动多个应用服务器进程，每个服务器进程分配不同的端口，然后在前端搭建一个负载均衡器，以反向代理的方式来分配访问请求。

考虑到在一台物理机器上建立逻辑集群的目的仅仅是为了尽可能的利用硬件资源，并不需要关系状态保留、热转移之类的高可用性需求，也不需要保证每个虚拟机有绝对准确的负载均衡，因此使用无session复制的亲合式集群是一个相当不错的选择，我们仅仅需要保障集群具备亲合性，也就是均衡器按一定的规则算法（一般根据SessionID分配）将一个固定的用户请求永远分配到固定的一个集群节点进行处理即可，这样程序开发阶段就基本不用为集群环境做什么特别的考虑了。

当然，很少有没有缺点的方案，如果计划使用逻辑集群的方式来部署程序，可能会遇到下面一些问题：

- 尽量避免节点竞争全局资源，最典型的就是磁盘竞争，各个节点如果同时访问某个磁盘文件的情况下（尤其是并发写操作容易出现问题），很容易导致IO异常。
- 很难最高效率的利用某些资源池，比如连接池，一般都是在各个节点建立独立的连接池，这样有可能导致一些节点池满了而另外一些节点仍有较多空余，尽管可以使用集中式的JNDI，但这个有一定的复杂性并可能带来额外的性能开销。
- 各个节点仍然不可避免的受到32位的内存限制，在32位Windows平台中每个进程只能使用2GB左右的内存，考虑到堆以外的内存开销，堆一般最多只能开到1.5GB。在某些Linux或Unix系统中，可以提升到3GB乃至接近4GB的内存，但32位中仍然受最高4GB内存的限制。
- 大量使用本地缓存（如大量使用HashMap作为K/V缓存）的应用，在逻辑集群中会造成较大的内存浪费，因为每个逻辑节点上都有一份缓存，这时候可以考虑把本地缓存改成集中式缓存。

总结：

> （1）受操作系统限制，32位程序可分配的最大内存为4G，64位程序可分配的最大内存为16G。
>
> （2）由于JVM除了堆内存还有堆外内存，因此32位最大可分配堆内存小于4G，根据操作系统的不同，最大可分配1.5G-3G。64位可分配更大的内存。
>
> （3）要充分利用高性能硬件的内存空间，有两种方法：（a）使用 64 位 JDK 来使用大内存。（b）使用若干个32位虚拟机建立逻辑集群来利用硬件资源。具体做法是在一台物理机上启动多个应用服务器进程，每个服务器进程分配不同的端口，然后在前端搭建一个负载均衡器，以反向代理的方式来分配访问请求。
>
> （4）64位的 JDK 存在一定问题：（a）Full GC 超大堆导致长时间停顿。（b）现阶段64位JDk的性能测试结果普遍低于32位JDK。 所以现阶段一般选用第二种方式。
>
> （5）如果选用64 位 JDK 使用大内存，则应避免超大堆GC导致长时间停顿。要么是非交互型，用户对停顿不敏感。对于和用户交互频繁，停顿敏感的程序，要保证 Full GC 发生的频率足够低，这样就可以通过深夜定时任务触发GC或者重启系统来清理内存。Full GC 频率低的条件是大多数对象的生存时间不应太长，尤其是不能有成批量的、长生存时间的大对象产生，这样才能保证老年代空间的稳定。大多数网站式应用都满足大部分对象朝生夕死的条件。

4、解决方案

介绍完两种部署方式之后，再重新回到这个案例之中，最后的部署方案调整为建立5个32位JDK的逻辑集群，每个进程按2GB内存计算（其中堆内存固定为1.5GB），占用了10GB内存。另外建立一个Apache服务作为前端均衡代理访问门户。考虑到用户对响应速度比较关心，并且文档服务的主要压力集中在磁盘和内存访问，CPU资源敏感度较低，因此改为CMS收集器进行垃圾回收。部署方式调整后，服务再没有出现长时间的停顿，速度比硬件升级前有较大提升。

（二）集群间同步导致的内存溢出

1、问题描述：

例如，有一个基于B/S的MIS系统，硬件为两台2个CPU、8GB内存的惠普小型机，服务器是WebLogic9.2，每台机器启动了3个WebLogic实例，构成6个节点的亲合式集群。由于亲合式集群，节点之间没有进行Session同步，但是有一些需求要实现部分数据在各个节点间共享。开始这些数据存放在数据库中，但由于读写频繁竞争激烈，性能影响较大，后面使用JBossCache构建了一个全局缓存。全局缓存启用后，服务器正常使用了一段较长的时间，但最近却不定期出现了多次的内存溢出。

在内存溢出不出现的时候，服务内存回收状况一直正常，每次内存回收后都能恢复到一个稳定的可用空间，开始怀疑是程序某些不常用的代码路径中存在内存泄露，但管理员反映最近程序并未更新、升级过，也没有进行特别的操作。只好让服务带着-XX:+HeapDumpOnOutOfMemoryError参数运行一段时间。在最近一次溢出之后，管理员发回了heapdump文件，发现里面存在这大量的org.jgroups.protocols.pbcast.NAKACK对象。

2、问题分析：

JBossCache是基于自家的JGroups进行集群间的数据通信，JGroups使用协议栈的方式来实现收发数据包的各种所需特性自由组合，数据包接收和发送时要经过每层协议栈的up()和down()方法，其中的NAKACK栈用于保障各个包的有效顺序和重发。

由于信息有传输失败需要重发的可能性，在确认所有注册在GMS（Group MembershipService）的节点都收到正确的信息前，发送的信息必须在内存中保留。而此MIS的服务端中有一个负责安全校验的全局Filter，每当接收到请求时，均会更新一次最后操作时间，并且将这个时间同步到所有节点去，使得一个用户在一段时间内不能在多台机器上登录。在服务使用过程中，往往一个页面会产生数次乃至数十次的请求，因此这个过滤器导致集群各个节点之间网络交互非常频繁。当网络情况不能满足传输要求时，要重发数据在内存中不断堆积，很快就产生了内存溢出。

这个案例中的问题，既有JBossCache的缺陷，也有MIS系统实现方式上的缺陷。JBossCache官方的maillist中讨论过很多此类似内存溢出的问题，据说后续版本也有所改进。而更重要的缺陷是这一类被集群共享的数据要使用类似JBossCache这种集群缓存来同步的话，可以允许操作频繁，因为数据在本地内存有一份副本，读取的动作不会耗费多少资源，但不应当有过于频繁的写操作，那样会带来很大的网络同步开销。

3、总结

> （1）本系统是6个节点的亲合式集群，节点之间没有进行Session同步，但是有一些需求要实现部分数据在各个节点间共享。
>
> （2）开始这些数据存放在数据库中，但由于读写频繁竞争激烈，性能影响较大。
>
> （3）后面使用 JBossCache 构建了一个全局缓存，出现了内存溢出的问题。这是因为JBossCache的缺陷：在网络情况不能满足传输要求时，要重发数据在内存中不断堆积，很快产生内存溢出。
>
> （4）这一类被集群共享的数据要使用类似JBossCache这种集群缓存来同步，更大的缺陷是，可以允许操作频繁，因为数据在本地内存有一份副本，读取的动作不会耗费多少资源，但不应当有过于频繁的写操作，那样会带来很大的网络同步开销。

（三）堆外内存导致的溢出错误

1、问题描述

例如，一个学校的小型项目：基于B/S的电子考试系统，为了实现客户端能实时地从服务器端接收考试数据，系统使用了逆向AJAX技术（也称为Comet或者Service Side Push），选用Comet1.1.1作为服务端推送框架，服务器是Jetty7.1.4，硬件为一个普通PC机，Corei5CPU，4GB内存，运行32位Windows操作系统。

测试期间发现服务端不定时抛出内存溢出异常，服务器不一定每次都会出现异常，但假如正式考试时崩溃一次，那估计整场电子考试都会乱套，网站管理员尝试把堆开到最大，而32位系统最多到1.6GB就基本无法再加大了，而且开大了基本没效果，抛出内存溢出异常好像更加频繁了。加入-XX:+HeapDumpOnOutOfMemoryError，居然也没有任何反应，抛出内存溢出异常时什么文件都没有产生。无奈之下只好挂着jstat并一直紧盯着屏幕，发现GC并不频繁，Eden区，Survivor区、老年代以及永久代全部都比较稳定，但就是照样不停的抛出内存溢出异常。最后，在内存溢出后从系统日志中找到异常堆栈，是DirectNIOBuffer处抛得异常。

2、问题分析

大家知道操作系统对每个进程能管理的内存是有限制的，这台服务器使用的32位Windows平台的限制是2GB，其中划了1.6GB给Java堆，而Direct Memory内存并不算入1.6GB的堆之内，因此它最大也之内在剩余的0.4GB空间中分出一部分。在此应用中导致溢出的关键是：垃圾收集进行时，虽然虚拟机会堆Direct Memory进行回收，但是DirectMemory却不能像新生代、老年代那样，发现空间不足了就通知收集器进行垃圾回收，它只能等待老年代满了之后进行Full GC，然后顺便的帮它清理掉内存的废弃对象。否则它只能一直等到抛出内存溢出异常时，先catch掉，再往catch块里进行System.gc()，要是虚拟机还是不管（比如开了-XX:+DisableExplicitGC开关），就只能看着堆中还有许多空闲内存，自己却不得不抛出内存溢出异常了。而本例中使用的Comet1.1.1框架，正好有大量的NIO操作需要使用到Direct Memory内存。

从实践经验角度出发，除了Java堆和永久代之外，我们注意到下面这些区域还会占用较多的内存，这里所有的内存总和受到操作系统进程最大内存的限制。

- DirectMemory：可通过-XX:MaxDirectMemorySize调整大小，内存不足时抛出OutOfMemoryError或者OutOfMemoryError：Direct buffer memory。
- 线程堆栈:可通过-Xss调整大小，内存不足时抛出StackOverFlowError（纵向无法分配即无法分配新的栈帧）或者OutOfMemoryError：unable tocreate new native thread（横向无法分配，即无法建立新的线程）。
- Socket缓存区：每个Socket连接都有Receive和Send缓冲区，分别占大约37KB和25KB内存，连接多的话这块内存占用也比较可观。如果无法分配，则可能抛出IOException：Too manyopenfiles异常。
- JNI代码：如果代码中使用JNI调用本地库，那本地库使用的内存也不在堆中。
- 虚拟机和GC：虚拟机和GC的代码执行也要消耗一定的内存。

3、总结

> （一）本例中使用的Comet1.1.1框架，有大量的NIO操作需要使用到Direct Memory内存。
>
> （二）服务端不定时抛出内存溢出异常，管理员将堆开到最大也无济于事。原因是堆开得越大，剩余可被直接内存区使用的空间越小，直接内存区变得更容易溢出。
>
> （三）DirectMemory 进行垃圾回收的过程：只能等待老年代满了之后进行Full GC，顺便的帮它清理掉内存的废弃对象。否则它只能一直等到抛出内存溢出异常时，先catch掉，再往catch块里进行System.gc()，要是虚拟机还是不管（比如开了-XX:+DisableExplicitGC开关），就只能看着堆中还有许多空闲内存，自己却不得不抛出内存溢出异常了。
>
> （四）除了堆和永久代，还有一些区域会占用较多内存，堆的空间分配过大会挤占这些内存区域。如：DirectMemory、线程堆栈、Socket缓存区、JNI代码、虚拟机和GC。

（四）外部命令导致系统缓慢

1、问题描述

这是一个来自网络的案例：一个数字校园应用系统，运行在一台4个CPU的Solaris 10操作系统上，中间件为GlassFish 服务器。系统在做大并发压力测试的时候，发现请求响应的时间比较缓慢，通过操作系统的mpstat工具发现CPU使用率很高。并且系统占用绝大多数的CPU资源的程序并不是应用程序本身。这是个不正常的现象，通常情况下用户应用的CPU占用率应该占主要地位，才能说明系统是正常工作的。

通过 Solaris 10 的 Dtrace 脚本可以查看当前情况下哪些系统调用花费了最多的CPU资源，Dtrace运行后发现最消耗CPU资源的竟是“fork”系统调用。“fork”系统调用是Linux用来产生新进程的，在Java虚拟机中，用户编写的Java代码最多只有线程的概念，不应当有进程的产生。

2、解决方案

这是一个非常异常的现象。通过本系统的开发人员，最终找到答案：每个用户请求的处理都需要执行一个外部shell脚本获取系统的一些信息。执行这个脚本是通过 Java 的 Runtime.getRuntime().exec() 方法来调用的。这种调用方式可以达到目的，但是它在 Java 虚拟机中是非常耗资源的操作，即使外部命令本身能很快执行，频繁调用时创建进程的开销也是非常大的。Java 虚拟机执行这个命令的过程是：首先克隆一个和当前虚拟机拥有一样环境变量的进程，再用这个新的进程去执行外部命令，最后再退出这个进程。如果频繁执行这个操作，系统的消耗会很大，不仅是 CPU，内存负担也很重。

用户根据建议去掉这个Shell脚本执行的语句，改为使用Java的API去获取这些信息后，系统很快恢复了正常。

3、总结

> （一）本例中CPU使用率很高，并且系统占用绝大多数的CPU资源的程序并不是应用程序本身。经监控发现最消耗CPU资源的竟是“fork”系统调用。
>
> （二）每个用户请求的处理都需要执行一个外部shell脚本获取系统的一些信息。Java 虚拟机执行这个命令的过程是：首先克隆一个和当前虚拟机拥有一样环境变量的进程，再用这个新的进程去执行外部命令，最后再退出这个进程，频繁执行这个操作消耗非常大。
>
> （三）用户根据建议去掉这个Shell脚本执行的语句，改为使用Java的API去获取这些信息后，系统很快恢复了正常。

（五）服务器 JVM 进程崩溃

1、问题描述

例如，一个基于B/S的MIS系统，硬件为两台2个CPU、8GB内存的HP系统，服务器是WebLogic9.2。正常运行一段时间后，最近发现在运行期间频繁出现集群节点的虚拟机进程自动关闭的现象，最后留下了一个hs_err_pid###.log文件后，进程就消失了，两台物理机器里的每个节点都出现过崩溃的现象。从系统日志可以看出，每个节点的虚拟机进程在崩溃前不久，都发生过大量相同的异常：“java.net.SocketException:Connection reset...”

这是一个远端断开连接的异常，通过系统管理员了解到系统最近与一个OA门户做了集成，在MIS系统工作流的代办事项变化时，要通过Web服务通知OA门户系统，把代办事项的变化同步到OA门户中。通过SoapUI测试了一个同步代办事项的几个Web服务，发现调用后竟然需要长达3分钟才能返回，并且返回结果都是连接中断。

2、解决方案

由于MIS系统的用户多，代办事项变化很快，为了不被OA系统速度拖累，使用了异步方式调用Web服务，但由于两边服务速度的完全不对等，时间越长就积累了越多的Web服务没有调用完成，导致在等待的线程和Socket连接越来越多，最终在超过虚拟机的承受能力后使得虚拟机进程崩溃。解决办法：通过OA门户方修复无法使用的集成接口，并将异步调用改为生产者/消费者模式的消息队列实现后，系统恢复正常。

3、总结

> （一）本例中：MIS系统需要将待办事项变化同步到OA门户中，MIS代办事项变化很快，OA处理速度比较慢，两边服务器速度不对等。
>
> （二）使用异步方式调用Web服务，由于两边服务速度的完全不对等，时间越长就积累了越多的Web服务没有调用完成，导致在等待的线程和Socket连接越来越多，最终在超过虚拟机的承受能力后使得虚拟机进程崩溃。
>
> （三）通过OA门户方修复无法使用的集成接口，并将异步调用改为生产者/消费者模式的消息队列实现后，系统恢复正常

（六）不恰当数据结构导致内存占用过大

1、问题描述

例如，有一个后台RPC服务器，使用64位虚拟机，内存配置为-Xms4g –Xmx8g –Xmn1g，使用ParNew+CMS的收集器组合。平时对外服务的Minor GC时间约在30毫秒以内，完全可以接受。但业务上需要每10分钟加载一个约80MB的数据文件到内存进行数据分析，这些数据会在内存形成超过100万个HashMapEntry，在这段时间里面Minor GC就会造成超过500毫秒的停顿，对于这个停顿时间就接受不了了。观察这个案例，发现平时的Minor GC时间很短，原因是新生代的绝大多数对象都是可清除的，在Minor GC之后Eden和Survivor基本上处于完全空闲的状态。而在分析数据文件期间，800MB的Eden空间很快被填满引发GC，但Minor GC之后，新生代中绝大多数对象依然是存活的。我们知道ParNew收集器使用的是复制算法，这个算法的高效是建立在大部分对象都是“朝生夕死”的特性上的，如果存活对象过多，把这些对象复制到Survivor并维持这些对象的引用的正确就称为了一个负担，因此导致GC暂停时间明显变长。

2、问题分析

如果不修改程序，仅从GC调优的角度去解决这个问题，可以考虑将Survivor空间去掉（加入参数-XX:SurvivorRatio=65536、-XX:MaxTenuringThreshold=0或者-XX:+AlwaysTenure），让新生代存活的对象在第一次MinorGC后立即进入老年代，等到Major GC的时候再清理它们。这种措施可以治标，但也有很大的副作用，治本的方案需要修改程序，因为这里的问题产生的根本原因是用HashMap结果来存储数据文件空间效率太低。

下面具体分析一下空间效率。在HashMap结构中，只有Key和Value所存放的两个长整型数据是有效数据，共16B（2 * 8B）。这两个长整型数据包装成java.lang.Long对象之后，就分别具有8B的MarkWord、8B的Klass指针，再加8B存储数据的long值。在这两个Long对象组成Map:Entry之后，又多了16B的对象头，然后一个8B的next字段和4B的int型的hash字段，为了对齐，还必须添加4B的空白填充，最后还有HashMap中对这个Entry的8B的引用，这样增加两个长整型数字，实际耗费的内存为（Long（24B）*2+Entry（32B）+HashMap Ref（8B）=88B），空间效率为16B/88B=18% ，实在太低了。

3、总结

> （一）本例业务上需要每10分钟加载一个约80MB的数据文件到内存进行数据分析，800M的新生代很快被填满，分析期间Minor GC会造成超过500毫秒的停顿，对于这个停顿时间接受不了。
>
> （二）新生代复制算法的高效是建立在大部分对象都是“朝生夕死”的特性上的，如果存活对象过多，把这些对象复制到Survivor并维持这些对象的引用的正确就称为了一个负担，因此导致GC暂停时间明显变长。
>
> （三）可以考虑将Survivor空间去掉，让新生代存活的对象在第一次MinorGC后立即进入老年代，等到Major GC的时候再清理它们。但是治标不治本。
>
> （四）根本解决方式是修改程序储存的数据结构。

（七）由 Windows 虚拟内存导致的长时间停顿

1、问题描述

例如，有一个带心跳检测功能的GUI桌面程序，每15秒会发送一次心跳检测信号，如果对方30秒以内没有信号返回，那就认为和对方程序的连接已断开。程序上线后发现心跳检测有误报的概率，查询日志发现误报的原因是程序会偶尔出现间隔约一分钟左右的时间完全无日志输出，处于停顿状态。

2、问题分析

因为是桌面程序，所需内存并不大（-Xmx256m），所以开始并没有想到是GC导致的程序停顿，但是加入参数-XX:+PrintGCApplicationStoppedTime –XX:+PrintGCDateStamps -Xloggc:gclog.log后，从GC日志文件中确认了停顿确实是由GC导致的，大部分GC时间都控制在100ms以内，但偶尔会出现接近一分钟的GC。

从GC日志中找到长时间停顿的具体日志信息（添加了-XX:+PrintReferenceGC参数），从日志中可以看出，真正执行GC动作的时间不是很长，但从准备开始GC，到真正开始GC之间所消耗的时间却占了绝大部分。

除GC日志之外，还观察到这个GUI程序内存变化的一个特点，当它最小化的时候，资源管理器中显示的占用内存大幅度减小，但是虚拟机内存则没有变化，因此怀疑程序在最小化时它的工作内存被自动交换到磁盘的页面文件中了，这样发生GC时就有可能因为恢复页面文件的操作而导致不正常的GC停顿。

3、解决方案

因此，在Java的GUI程序中要避免这样的情况，还可以加入参数“-Dsun.awt.KeepWorkingSetOnMinimize=ture”来解决。这个参数在许多AWT的程序上都有应用，例如JDK自带的Visual VM，用于保证程序在恢复最小化时能够立即响应。在这个案例中加入该参数之后，问题得到解决。

4、总结

> （一）本例是一个GUI桌面程序，大部分GC时间都控制在100ms以内，但偶尔会出现接近一分钟的GC。由于每15秒会发送一次心跳检测信号检测程序连接是否断开，所以一分钟的停顿无法接受。
>
> （二）从日志中看出，真正执行GC动作的时间不是很长，但从准备开始GC，到真正开始GC之间所消耗的时间却占了绝大部分。怀疑程序在最小化时它的工作内存被自动交换到磁盘的页面文件中了，这样发生GC时就有可能因为恢复页面文件的操作而导致不正常的GC停顿。
>
> （三）在Java的GUI程序中要避免这样的情况，可以加入参数“-Dsun.awt.KeepWorkingSetOnMinimize=ture”来解决，用于保证程序在恢复最小化时能够立即响应。