这是我公司同事的GC学习笔记，写得蛮详细的，由浅入深，循序渐进，让人一看就懂，特转到这里。

一、GC特性以及各种GC的选择

1、垃圾回收器的特性

2、对垃圾回收器的选择

2.1 连续 VS. 并行

2.2 并发 VS. stop-the-world

2.3 压缩 VS. 不压缩 VS. 复制

二、GC性能指标

三、分代回收

四、J2SE 5.0的HotSpot JVM上的GC学习 - 分代、GC类型、快速分配

五、J2SE 5.0的HotSpot JVM上的GC学习 - SerialGC

六、J2SE 5.0的HotSpot JVM上的GC学习 - ParallelGC

七、J2SE 5.0的HotSpot JVM上的GC学习 - ParallelCompactingGC

八、J2SE 5.0的HotSpot JVM上的GC学习 - CMS GC

九、启动参数学习示例

1. GC特性以及各种GC的选择

1.1 垃圾回收器的特性

该回收的对象一定要回收，不该回收的对象一定不能回收

一定要有效，并且要快！尽可能少的暂停应用的运行

需要在时间，空间，回收频率这三个要素中平衡

内存碎片的问题（一种解决内存碎片的方法，就是压缩）

可扩展性和可伸缩性（内存的分配和回收，不应该成为跑在多核多线程应用上的瓶颈）

对垃圾回收器的选择

1.2 连续 VS. 并行

连续垃圾回收器，即使在多核的应用中，在回收时，也只有一个核被利用。

但并行GC会使用多核，GC任务会被分离成多个子任务，然后这些子任务在各个CPU上并行执行。

并行GC的好处是让GC的时间减少，但缺点是增加了复杂度，并且存在产生内存碎片的可能。

1.3 并发 VS. stop-the-world

当使用stop-the-world 方式的GC在执行时，整个应用会暂停住的。

而并发是指GC可以和应用一起执行，不用stop the world。

一般的说，并发GC可以做到大部分的运行时间，是可以和应用并发的，但还是有一些小任务，不得不短暂的stop the world。

stop the world 的GC相对简单，因为heap被冻结，对象的活动也已经停止。但缺点是可能不太满足对实时性要求很高的应用。

相应的，并发GC的stop the world时间非常短，并且需要做一些额外的事情，因为并发的时候，对象的引用状态有可能发生改变的。

所以，并发GC需要花费更多的时间，并且需要较大的heap。

1.4 压缩 VS. 不压缩 VS. 复制

在GC确定内存中哪些是有用的对象，哪些是可回收的对象之后，他就可以压缩内存，把拥有的对象放到一起，并把剩下的内存进行清理。

在压缩之后，分配对象就会快很多，并且内存指针可以很快的指向下一个要分配的内存地址。

一个不压缩的GC，就原地把不被引用的对象回收，他并没有对内存进行压缩。好处就是回收的速度变快了；缺点呢，就是产生了碎片。

一般来说，在有碎片的内存上分配一个对象的代价要远远大于在没有碎片的内存上分配。

另外的选择是使用一个复制算法的GC，他是把所有被引用的对象复制到另外一个内存区域中。

使用复制GC的好处就是，原来的内存区域，就可以被毫无顾忌的清空了。但缺点也很明显，需要更多的内存，以及额外的时间来复制。

2. GC性能指标

几个评估GC性能的指标

吞吐量      应用花在非GC上的时间百分比

GC负荷    与吞吐量相反，指应用花在GC上的时间百分比

暂停时间   应用花在GC stop-the-world 的时间

GC频率     顾名思义

Footprint   一些资源大小的测量，比如堆的大小

反应速度   从一个对象变成垃圾道这个对象被回收的时间

一个交互式的应用要求暂停时间越少越好，然而，一个非交互性的应用，当然是希望GC负荷越低越好。

一个实时系统对暂停时间和GC负荷的要求，都是越低越好。

一个嵌入式系统当然希望Footprint越小越好。

3. 分代回收

3.1 什么是分代

当使用分代回收技术，内存会被分为几个代（generation）。也就是说，按照对象存活的年龄，把对象放到不同的代中。

使用最广泛的代，应属年轻代和年老代了。

根据各种GC算法的特征，可以相应的被应用到不同的代中。

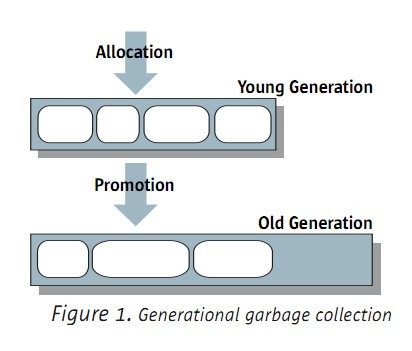
研究发现：

大部分的对象在分配后不久，就不被引用了。也就是，他们在很早就挂了。

只有很少的对象熬过来了。

年轻代的GC相当的频繁，高效率并且快。因为年轻代通常比较小，并且很多对象都是不被引用的。

如果年轻代的对象熬过来了，那么就晋级到年老代中了。如图：



通常年老代要比年轻代大，而且增长也比较慢。所以GC在年老代发生的频率非常低，不过一旦发生，就会占据较长的时间。

3.2 总结

年轻代通常使用时间占优的GC，因为年轻代的GC非常频繁

年老代通常使用善于处理大空间的GC，因为年老代的空间大，GC频率低

4. J2SE 5.0的HotSpot JVM上的GC学习 - 分代、GC类型、快速分配

J2SE5.0 update 6 的HotSpot上有4个GC。

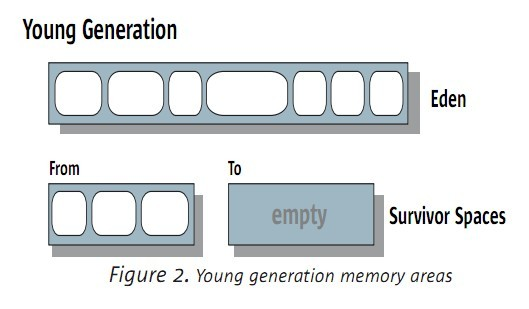
4.1 HotSpot上的分代

分成三部分：年轻代、年老代、永久代

很多的对象一开始是分配在年轻代的，这些对象在熬过了一定次数的young gc之后，就进入了年老代。同时，一些比较大的对象，一开始就可能被直接分配到年老代中（因为年轻代比较小嘛）。

4.2 年轻代

年轻代也进行划分，划分成：一个Eden和两个survivor。如下图：



大部分的对象被直接分配到年轻代的eden区（之前已经提到了是，很大的对象会被直接分配到年老代中），

survivor区里面放至少熬过一个YGC的对象，在survivor里面的对象，才有机会被考虑提升到年老代中。

同一时刻，两个survivor只被使用一个，另外一个是用来进行复制GC时使用的。

4.3 GC类型

年轻代的GC叫young GC，有时候也叫 minor GC。年老代或者永久代的GC，叫 full GC，也叫major GC。

也就是说，所有的代都会进行GC。

一般的，首先是进行年轻代的GC，（使用针对年轻代的GC），然后是年老代和永久代使用相同的GC。如果要压缩（解决内存碎片问题），每个代需要分别压缩。

有时候，如果年老区本身就已经很满了，满到无法放下从survivor熬出来的对象，那么，YGC就不会再次触发，而是会使用FullGC对整个堆进行GC（除了CMS这种GC，因为CMS不能对年轻代进行GC）

4.4 快速分配内存

多线程进行对象建立的时候，在为对象分配内存的时候，就应该保证线程安全，为此，就应该进入全局锁。但全局锁是非常消耗性能的。

为此，HotSpot引入了Thread Local Allocation Buffers （TLAB）技术，这种技术的原理就是为每个线程分配一个缓冲，用来分配线程自己的对象。

每个线程只使用自己的TLAB，这样，就保证了不用使用全局锁。当TLAB不够用的时候，才需要使用全局锁。但这时候对锁的时候，频率已经相当的低了。

为了减少TLAB对空间的消耗，分配器也想了很多方法，平均来说，TLAB占用Eden区的不到1%。

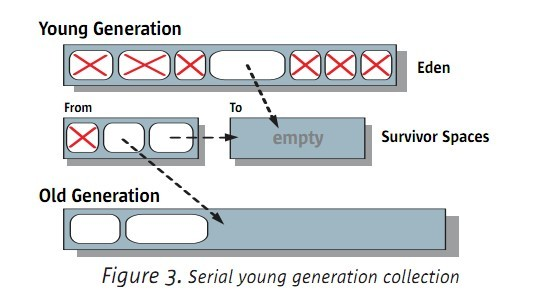
5. J2SE 5.0的HotSpot JVM上的GC学习 - SerialGC

5.1 串行GC

 串行GC，只使用单个CPU，并且会stop the world。

5.1.1 young 的串行GC

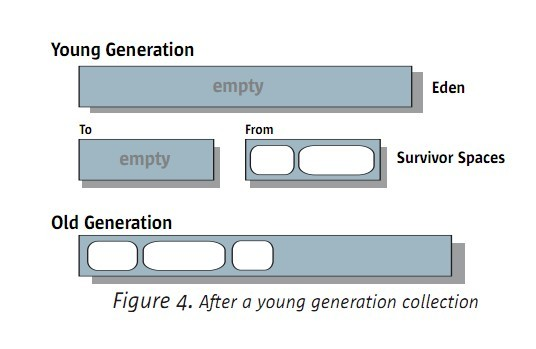
如下图：



当发生ygc的时候，Eden和From的survivor区会将被引用的对象复制到To这个survivor种。

如果有些对象在To survivor放不下，则直接升级到年老区。

当YGC完成后，内存情况如下图：



5.1.2 old区的串行GC

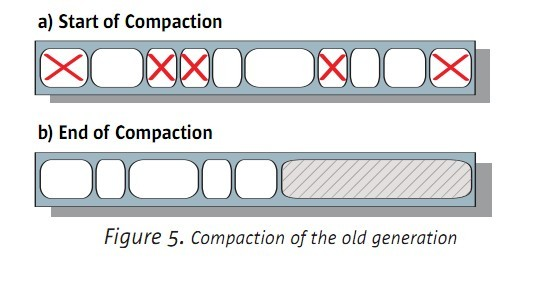
年老区和永久区使用的是Mark-Sweep-Compact的算法。

mark阶段是对有引用的对象进行标识

sweep是对垃圾进行清理

compact对把活着的对象进行迁移，解决内存碎片的问题

如下图：



5.2 何时使用串行收集器

串行GC适用于对暂停时间要求不严，在客户端下使用。

5.3 串行收集器的选择

在J2SE5.0上，在非 server 模式下，JVM自动选择串行收集器。

也可以显示进行选择，在java启动参数中增加： -XX:+UseSerialGC 。

6. J2SE 5.0的HotSpot JVM上的GC学习 - ParallelGC

6.1 并行GC

现在已经有很多java应用跑在多核的机器上了。

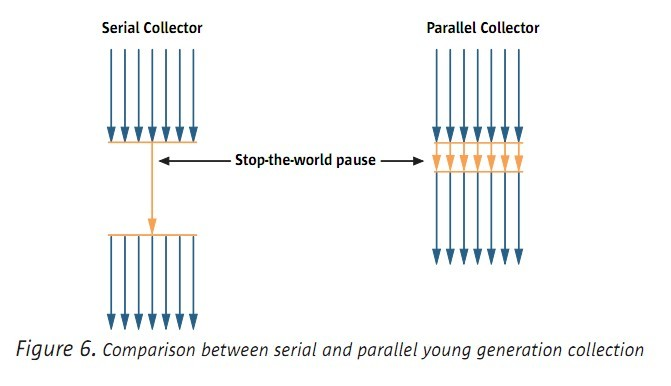
并行的GC，也称作吞吐量GC，这种GC把多个CPU都用上了，不让CPU再空转。

6.2 YGC的并行GC

YGC的情况，还是使用stop-the-world + 复制算法的GC。

只不过是不再串行，而是充分利用多个CPU，减少GC负荷，增加吞吐量。

如下图，串行YGC和并行YGC的比较：



6.3 年老区的并行GC

也是和串行GC一样，在年老区和永久区使用Mark-Sweep-Compact，利用多核增加了吞吐量和减少GC负荷。

6.4 何时使用并行GC

对跑在多核的机器上，并且对暂停时间要求不严格的应用。因为频率较低，但是暂停时间较长的Full GC还是会发生的。

6.5 选择并行GC

在server模式下，并行GC会被自动选择。

或者可以显式选择并行GC，加启动JVM时加上参数： -XX:UseParallelGC

7. J2SE 5.0的HotSpot JVM上的GC学习 - ParallelCompactingGC

7.1 Parallel Compacting GC

parallelCompactingGC是在J2SE5.0 update6 引入的。

parallel compacting GC 与 parallel GC的不同地方，是在年老区的收集使用了一个新的算法。并且以后，parallel compacting GC 会取代 parallem GC的。

7.2 YGC的并行压缩GC

与并行GC使用的算法一样：stop-the-world 和 复制。

7.3 年老区的并行压缩GC

他将把年老区和永久区从逻辑上划分成等大的区域。

分为三个阶段：

标记阶段，使用多线程对存在引用的对象进行并行标记。

分析阶段，GC对各个区域进行分析，GC认为，在经过上次GC后，越左边的区域，有引用的对象密度要远远大于右边的区域。所以就从左往右分析，当某个区域的密度达到一个值的时候，就认为这是一个临界区域，所以这个临界区域左边的区域，将不会进行压缩，而右边的区域，则会进行压缩。

压缩阶段，多各个需要压缩的区域进行并行压缩。

7.4 什么时候使用并行压缩GC

同样的，适合在多核的机器上；并且此GC在FGC的时候，暂停时间会更短。

可以使用参数 -XX:ParallelGCThreads=n 来指定并行的线程数。

7.5 开启并行压缩GC

使用参数 -XX:+UseParallelOldGC

8. J2SE 5.0的HotSpot JVM上的GC学习 - CMS GC

8.1 Concurrent mark sweep GC

很多应用对响应时间的要求要大于吞吐量。

YGC并不暂停多少时间，但FGC对时间的暂用还是很长的。特别是在年老区使用的空间较多时。

因此， HotSpot引入了一个叫做CMS的收集器，也叫低延时收集器。

8.2 CMS的YGC

与并行GC同样的方式： stop-the-world 加上 copy。

8.3 CMS的FGC

CMS的FGC在大部分是和应用程序一起并发的！

CMS在FGC的时候，一开始需要做一个短暂的暂停，这个阶段称为最初标记：识别所有被引用的对象。

在并发标记时候，会和应用程序一起运行。

因为并发标记是和程序一起运行的，所以在并发标记结束的时候，不能保证所有被引用的对象都被标记，

为了解决这个问题，GC又进行了一次暂停，这个阶段称为：重标识（remark）。

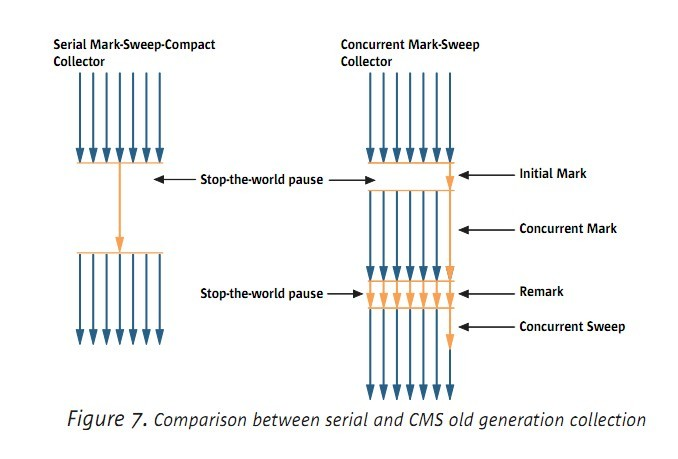
在这个过程中，GC会重新对在并发标阶段时候有修改的对象做标记。

因为remark的暂停要大于最初标记，所以在这时候，需要使用多线程来并行标记。

在上述动作完成之后，就可以保证所有被引用的对象都被标记了。

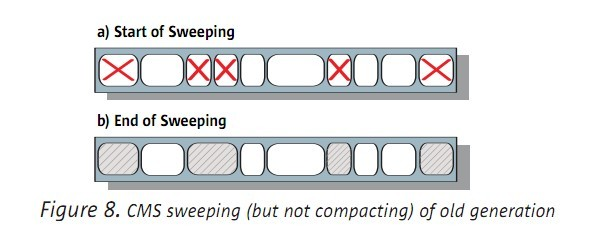
因此，并发清理阶段就可以并发的收集垃圾了。

下图是serial gc 和 CMS gc 的对比：



因为要增加很多额外的动作，比如对被引用的对象重新标记，增加了CMS的工作量，所以他的GC负荷也相应的增加。

CMS是唯一没有进行压缩的GC。如下图：



没有压缩，对于GC的过程，是节约了时间。但因此产生了内存碎片，所以对于新对象在年老区的分配，就产生了速度上的影响，

当然，也就包括了对YGC时间的影响了。

CMS的另一个缺点，就是他需要的堆比较大，因为在并发标记的时候和并发清除的时候，应用程序很有可能在不断产生新的对象，而垃圾又还没有被删除。

另外，在最初标记之后的并发标记时，原先被引用的对象，有可能变成垃圾。但在这一次的GC中，这是没有被删除的。这种垃圾叫做：漂流垃圾。

最后，由于没有进行压缩，由此而带来了内存碎片。

为了解决这个问题，CMS对热点object大小进行了统计，并且估算之后的需求，然后把空闲的内存进行拆分或者合并来满足后续的需求。

与其他的GC不同，CMS并不在年老区满了之后才开始GC，他需要提前进行GC，用以满足在GC同时需要额外的内存。

如果在GC的同时，内存不能满足要求了，则GC就变成了并行GC或者串行GC。

为了防止这种情况，会根据上一次GC的统计来确定启动时间。

或者是当年老区超过初始容量的话，CMS GC就会启动。

初始容量的设置可以在JVM启动时增加参数： -XX:CMSInitiatingOccupancyFraction=n

n是一个百分比，默认值为68。

总之，CMS比并行GC花费了更少的暂停时间，但是牺牲了吞吐量，以及需要更大的堆区。

8.4 额外模式

为了防止在并发标记的时候，GC线程长期占用CPU，CMS可以把并发标记的时候停下来，把cpu让给应用程序。

收集器会想并发标记分解成很小的时间串任务，在YGC之间来执行。

这个功能对于机器的CPU个数少，但又想降低暂停时间的应用来说，非常有用。

8.5 何时使用CMS

当CPU资源较空闲，并且需要很低的暂停时间时，可以选择CMS。比如 web servers。

8.6 选择CMS

选择CMS GC： 增加参数 -XX:UseConcMarkSweepGC

开启额外模式： 增加参数 -XX:+CMSIncreamentalMode

9. 结合线上启动参数学习

线上的启动参数

-Dprogram.name=run.sh -Xmx2g -Xms2g -Xmn256m -XX:PermSize=128m -Xss256k -XX:+DisableExplicitGC -XX:+UseConcMarkSweepGC -XX:+CMSParallelRemarkEnabled -XX:+UseCMSCompactAtFullCollection -XX:LargePageSizeInBytes=128m -XX:+UseFastAccessorMethods -XX:+UseCMSInitiatingOccupancyOnly -XX:CMSInitiatingOccupancyFraction=70 -Djava.awt.headless=true -Djava.net.preferIPv4Stack=true -Dcom.sun.management.config.file=/home/admin/web-deploy/conf/jmx/jmx\_monitor\_management.properties -Djboss.server.home.dir=/home/admin/web-deploy/jboss\_server -Djboss.server.home.url=file\:/home/admin/web-deploy/jboss\_server -Dapplication.codeset=GBK -Ddatabase.codeset=ISO-8859-1 -Ddatabase.logging=false -Djava.endorsed.dirs=/usr/alibaba/jboss/lib/endorsed

其中：

-Xmx2g -Xms2g 表示堆为2G

-Xmn256m 表示新生代为 256m

-Xss256k 设置每个线程的堆栈大小。JDK5.0以后每个线程堆栈大小为1M，以前每个线程堆栈大小为256K。更具应用的线程所需内存大小进行调整。在相同物理内存下，减小这个值能生成更多的线程。但是操作系统对一个进程内的线程数还是有限制的，不能无限生成，经验值在3000~5000左右

-XX:PermSize=128m 表示永久区为128m

-XX:+DisableExplicitGC 禁用显示的gc，程序程序中使用System.gc()中进行垃圾回收，使用这个参数后系统自动将 System.gc() 调用转换成一个空操作

-XX:+UseConcMarkSweepGC 表示使用CMS

-XX:+CMSParallelRemarkEnabled 表示并行remark

-XX:+UseCMSCompactAtFullCollection 表示在FGC之后进行压缩，因为CMS默认不压缩空间的。

-XX:+UseCMSInitiatingOccupancyOnly 表示只在到达阀值的时候，才进行CMS GC

-XX:CMSInitiatingOccupancyFraction=70 设置阀值为70%，默认为68%。

-XX:+UseCompressedOops JVM优化之压缩普通对象指针（CompressedOops），通常64位JVM消耗的内存会比32位的大1.5倍，这是因为对象指针在64位架构下，长度会翻倍（更宽的寻址）。对于那些将要从32位平台移植到64位的应用来说，平白无辜多了1/2的内存占用，这是开发者不愿意看到的。幸运的是，从JDK 1.6 update14开始，64 bit JVM正式支持了 -XX:+UseCompressedOops 这个可以压缩指针，起到节约内存占用的新参数.

关于-XX:+UseCMSInitiatingOccupancyOnly 和 -XX:CMSInitiatingOccupancyFraction ，详细解释见下：

The concurrent collection generally cannot be sped up but it can be started earlier.

A concurrent collection starts running when the percentage of allocated space in the old generation crosses a threshold. This threshold is calculated based on general experience with the concurrent collector. If full collections are occurring, the concurrent collections may need to be started earlier. The command line flag CMSInitiatingOccupancyFraction can be used to set the level at which the collection is started. Its default value is approximately 68%. The command line to adjust the value is

-XX:CMSInitiatingOccupancyFraction=<percent>

The concurrent collector also keeps statistics on the promotion rate into the old generation for the application and makes a prediction on when to start a concurrent collection based on that promotion rate and the available free space in the old generation. Whereas the use of CMSInitiatingOccupancyFraction must be conservative to avoid full collections over the life of the application, the start of a concurrent collection based on the anticipated promotion adapts to the changing requirements of the application. The statistics that are used to calculate the promotion rate are based on the recent concurrent collections. The promotion rate is not calculated until at least one concurrent collection has completed so at least the first concurrent collection has to be initiated because the occupancy has reached CMSInitiatingOccupancyFraction . Setting CMSInitiatingOccupancyFraction to 100 would not cause only the anticipated promotion to be used to start a concurrent collection but would rather cause only non-concurrent collections to occur since a concurrent collection would not start until it was already too late. To eliminate the use of the anticipated promotions to start a concurrent collection set UseCMSInitiatingOccupancyOnly to true.

-XX:+UseCMSInitiatingOccupancyOnly

关于内存管理完整详细信息，请查看这份文档：<http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf>