http://hot66hot.iteye.com/blog/2075819

**一:理解GC日志格式,读GC日志的方法**

**1:开启日志**

-verbose:gc

-XX:+PrintGCDetails

-XX:+PrintGCDateStamps

-Xloggc:/path/gc.log

-XX:+UseGCLogFileRotation  启用GC日志文件的自动转储 (Since Java)

-XX:NumberOfGClogFiles=1  GC日志文件的循环数目 (Since Java)

-XX:GCLogFileSize=1M  控制GC日志文件的大小 (Since Java)

-XX:+PrintGC包含-verbose:gc

-XX:+PrintGCDetails //包含-XX:+PrintGC

只要设置-XX:+PrintGCDetails 就会自动带上-verbose:gc和-XX:+PrintGC

-XX:+PrintGCDateStamps/-XX:+PrintGCTimeStamps 输出gc的触发时间

**2:新生代(Young GC)gc日志分析**

**Java代码  [收藏代码](javascript:void())**

1. 2014-02-28T11:59:00.638+0800: 766.537:[GC2014-02-28T11:59:00.638+0800: 766.537:
2. [ParNew: 1770882K->212916K(1835008K), 0.0834220 secs]
3. 5240418K->3814487K(24903680K), 0.0837310 secs] [Times: user=1.12 sys=0.02, real=0.08 secs]

  2014-02-28T11:59:00 ...（时间戳）:[GC（Young GC）（时间戳）:[ParNew（使用ParNew作为年轻代的垃圾回收期）:

1770882K（年轻代垃圾回收前的大小）->212916K（年轻代垃圾回收以后的大小）(1835008K)（年轻代的capacity）, 0.0834220 secs（回收时间）]  
5240418K（整个heap垃圾回收前的大小）->3814487K（整个heap垃圾回收后的大小）(24903680K)（heap的capacity）, 0.0837310secs（回收时间）]  
[Times: user=1.12（Young GC用户耗时） sys=0.02（Young GC系统耗时）, real=0.08 secs（Young GC实际耗时）]

   其中 Young GC回收了1770882-212916=1557966K内存  
Heap通过这次回收总共减少了 5240418-3814487=1425931 K的内存。1557966-1425931=132035K说明这次Young GC有约128M的内存被移动到了Old Gen，  
 提示:进代量(Young->Old)需要重点观察，预防promotion failed.

**3:老年代(CMS old gc)分析**

**Java代码  [收藏代码](javascript:void())**

1. 2014-02-28T23:58:42.314+0800: 25789.661: [GC [1 CMS-initial-mark: 17303356K(23068672K)] 18642315K(24903680K), 1.0400410 secs] [Times: user=1.04 sys=0.00, real=1.04 secs]
2. 2014-02-28T23:58:43.354+0800: 25790.701: [CMS-concurrent-mark-start]
3. 2014-02-28T23:58:43.717+0800: 25791.064: [CMS-concurrent-mark: 0.315/0.363 secs] [Times: user=1.64 sys=0.02, real=0.37 secs]
4. 2014-02-28T23:58:43.717+0800: 25791.064: [CMS-concurrent-preclean-start]
5. 2014-02-28T23:58:43.907+0800: 25791.254: [CMS-concurrent-preclean: 0.181/0.190 secs] [Times: user=0.20 sys=0.01, real=0.19 secs]
6. 2014-02-28T23:58:43.907+0800: 25791.254: [CMS-concurrent-abortable-preclean-start]
7. CMS: abort preclean due to time 2014-02-28T23:58:49.082+0800: 25796.429: [CMS-concurrent-abortable-preclean: 5.165/5.174 secs] [Times: user=5.40 sys=0.04, real=5.17 secs]
8. 2014-02-28T23:58:49.083+0800: 25796.430: [GC[YG occupancy: 1365142 K (1835008 K)]2014-02-28T23:58:49.083+0800: 25796.430: [Rescan (parallel) , 0.9690640 secs]2014-02-28T23:58:50.052+0800: 25797.399: [weak refs processing, 0.0006190 secs]2014-02-28T23:58:50.053+0800: 25797.400: [scrub string table, 0.0006290 secs] [1 CMS-remark: 17355150K(23068672K)] 18720292K(24903680K), 0.9706650 secs] [Times: user=16.49 sys=0.06, real=0.97 secs]
9. 2014-02-28T23:58:50.054+0800: 25797.401: [CMS-concurrent-sweep-start]
10. 2014-02-28T23:58:51.940+0800: 25799.287: [CMS-concurrent-sweep: 1.875/1.887 secs] [Times: user=2.03 sys=0.03, real=1.89 secs]
11. 2014-02-28T23:58:51.941+0800: 25799.288: [CMS-concurrent-reset-start]
12. 2014-02-28T23:58:52.067+0800: 25799.414: [CMS-concurrent-reset: 0.127/0.127 secs] [Times: user=0.13 sys=0.00, real=0.13 secs]
13. 2014-03-01T00:00:36.293+0800: 25903.640: [GC2014-03-01T00:00:36.293+0800: 25903.640: [ParNew: 1805234K->226801K(1835008K), 0.1020510 secs] 10902912K->9434796K(24903680K), 0.1023150 secs] [Times: user=1.35 sys=0.02, real=0.10 secs]
14. 2014-03-01T00:07:13.559+0800: 26300.906: [GC2014-03-01T00:07:13.559+0800: 26300.906: [ParNew: 1799665K->248991K(1835008K), 0.0876870 secs] 14086673K->12612462K(24903680K), 0.0879620 secs] [Times: user=1.24 sys=0.01, real=0.09 secs]

 CMS的gc日志分为一下几个步骤，重点关注initial-mark和remark这两个阶段，因为这两个阶段会stop进程。

初始标记（init mark）：收集根引用，这是一个stop-the-world阶段。

并发标记（concurrent mark）：这个阶段可以和用户应用并发进行。遍历老年代的对象图，标记出活着的对象。

并发预清理（concurrent preclean）：这同样是一个并发的阶段。主要的用途也是用来标记，用来标记那些在前面标记之后，发生变化的引用。主要是为了缩短remark阶段的stop-the-world的时间。

重新标记（remark）：这是一个stop-the-world的操作。暂停各个应用，统计那些在发生变化的标记。

并发清理（concurrent sweep）：并发扫描整个老年代，回收一些在对象图中不可达对象所占用的空间。

并发重置（concurrent reset）：重置某些数据结果，以备下一个回收周期

提示:红色为全部暂停阶段,重点关注.

**Java代码  [收藏代码](javascript:void())**

1. [GC [1 CMS-initial-mark: 17303356K(23068672K)] 18642315K(24903680K), 1.0400410 secs] [Times: user=1.04 sys=0.00, real=1.04 secs]

 其中数字依表示标记前后old区的所有对象占内存大小和old的capacity，整个JavaHeap（不包括perm）所有对象占内存总的大小和JavaHeap的capacity。

**Java代码  [收藏代码](javascript:void())**

1. [GC[YG occupancy: 1365142 K (1835008 K)]2014-02-28T23:58:49.083+0800: 25796.430:
2. [Rescan (parallel) , 0.9690640 secs]2014-02-28T23:58:50.052+0800: 25797.399:
3. [weak refs processing, 0.0006190 secs]2014-02-28T23:58:50.053+0800: 25797.400: [scrub string table, 0.0006290 secs]
4. [1 CMS-remark: 17355150K(23068672K)] 18720292K(24903680K), 0.9706650 secs] [Times: user=16.49 sys=0.06, real=0.97 secs]

 Rescan (parallel)表示的是多线程处理young区和多线程扫描old+perm的总时间， parallel 表示多GC线程并行。

weak refs processing 处理old区的弱引用的总时间，用于回收native memory.等等

参考资料:

<https://blogs.oracle.com/jonthecollector/entry/the_unspoken_cms_and_printgcdetails>

<https://blogs.oracle.com/poonam/entry/understanding_cms_gc_logs>

**4:老年代(CMS old GC ) concurrent mode failure日志**

**Java代码  [收藏代码](javascript:void())**

1. 2014-03-03T09:38:26.457+0800: 233373.804: [GC [1 CMS-initial-mark: 17319615K(23068672K)] 17351070K(24903680K), 0.0419440 secs]
2. [Times: user=0.04 sys=0.00, real=0.04 secs]
3. 2014-03-03T09:38:26.499+0800: 233373.846: [CMS-concurrent-mark-start]
4. 2014-03-03T09:38:28.175+0800: 233375.522: [GC2014-03-03T09:38:28.175+0800: 233375.522: [CMS2014-03-03T09:38:28.887+0800: 233376.234:
5. [CMS-concurrent-mark: 1.989/2.388 secs] [Times: user=14.37 sys=0.24, real=2.39 secs]
6. (concurrent mode failure): 17473174K->8394653K(23068672K), 19.3309170 secs] 18319691K->8394653K(24903680K),
7. [CMS Perm : 23157K->23154K(98304K)], 19.3311700 secs] [Times: user=22.18 sys=0.00, real=19.33 secs]

 concurrent mode failure一般发生在CMS GC 运行过程中,老年代空间不足，引发MSC(Full GC)

上面的这条发日志说明CMS运行到CMS-concurrent-mark过程中就出现空间不足,产生并发失败(17319615K(23068672K)占77%),

**解决思路：降低YGC频率，降低CMS GC触发时机,适当降低CMSInitiatingOccupancyFraction.**

**5:新生代(ParNew YGC)promotion failed日志**

**Java代码  [收藏代码](javascript:void())**

1. 2014-02-27T21:19:42.460+0800: 210095.040: [GC 210095.040: [ParNew (promotion failed): 1887487K->1887488K(1887488K), 0.4818790 secs]210095.522: [CMS: 13706434K->7942818K(23068672K), 9.7152990 secs] 15358303K->7942818K(24956160K), [CMS Perm : 27424K->27373K(98304K)], 10.1974110 secs] [Times: user=12.06 sys=0.01, real=10.20 secs]

 promotion failed一般发生在新生代晋升老年代时,老年代空间不够或连续空间不够却还没达到old区的触发值,引发Full Gc.

**解决思路:由于heap碎片,YGC晋升对象过大,过长.(mid/long Time Object),调整-XX:PretenureSizeThreshold=65535,-XX:MaxTenuringThreshold=6**

**6:system.gc()产生的Full GC日志**

**Java代码  [收藏代码](javascript:void())**

1. <strong>2014-01-21T17:44:01.554+0800: 50212.568: [Full GC (System) 50212.568:
2. [CMS: 943772K220K(2596864K), 2.3424070 secs] 1477000K->220K(4061184K), [CMS Perm : 3361K->3361K(98304K)], 2.3425410 secs] [Times: user=2.33 sys=0.01, real=2.34 secs]</strong>

Full GC (System)意味着这是个system.gc调用产生的MSC。  
“943772K->220K(2596864K), 2.3424070 secs”表示：这次MSC前后old区内总对象大小，old的capacity及这次MSC耗时。  
“1477000K->220K(4061184K)”表示：这次MSC前后JavaHeap内总对象大小，JavaHeap的capacity。  
“3361K->3361K(98304K)], 2.3425410 secs”表示：这次MSC前后Perm区内总对象大小，Perm区的capacity。

**解决:**  
**使用-XX:+DisableExplicitGC参数，System.gc()会变成空调用.**  
**如果应用有地方大量使用direct memory 或 rmi，那么使用-XX:+DisableExplicitGC要小心。**  
**可以使用-XX:+ExplicitGCInvokesConcurrent替换把 System.gc()从Full GC换成CMS GC.**

原因:  
DirectByteBuffer没有finalizer，native memory的清理工作是通过sun.misc.Cleaner自动完成  
sun.misc.Cleaner是基于PhantomReference的清理工具,Full GC/Old GC会对old gen做reference processing，同时触发Cleaner对已死的DirectByteBuffer对象做清理。  
如果长时间没有GC或者只做了young GC的话,不会触发old区Cleaner的工作，容易产生DirectMemory OOM.  
参考:<https://gist.github.com/rednaxelafx/1614952>

RMI会做的是分布式GC。Sun JDK的分布式GC是用纯Java实现的,为RMI服务。   
参考:<http://docs.oracle.com/javase/6/docs/technotes/guides/rmi/sunrmiproperties.html>

**7:特殊的Full GC日志,根据动态计算直接进行MSC**

**Java代码  [收藏代码](javascript:void())**

1. 2014-02-13T13:48:06.349+0800: 7.092: [GC 7.092: [ParNew: 471872K->471872K(471872K), 0.0000420 secs]7.092: [CMS: 366666K->524287K(524288K), 27.0023450 secs] 838538K->829914K(996160K), [CMS Perm : 3196K->3195K(131072K)], 27.0025170 secs]

 ParNew的时间特别短，jvm在minor gc前会首先确认old是不是足够大，如果不够大，这次young gc直接返回，进行MSC(Full GC)。

**二:参数配置和理解**

**1:参数分类和说明**

jvm参数分固定参数和非固定参数

**1):固定参数**

如：-Xmx,-Xms,-Xmn,-Xss.

**2):非固定参数**

如:

-XX:+<option> 启用选项

-XX:-<option> 不启用选项

-XX:<option>=<number> 给选项设置一个数字类型值，可跟单位，例如 128k, 2g

-XX:<option>=<string> 给选项设置一个字符串值，例如-XX:HeapDumpPath=./dump.log

**2:JVM可设置参数和默认值**

**1):-XX:+PrintCommandLineFlags**

打印出JVM初始化完毕后所有跟最初的默认值不同的参数及它们的值,jdk1.5后支持.

线上建议打开，可以看到自己改了哪些值.

**2):-XX:+PrintFlagsFinal**

显示所有可设置的参数及"参数处理"后的默认值。参数本身只从JDK6 U21后支持

可是查看不同版本默认值,以及是否设置成功.输出的信息中"="表示使用的是初始默认值，而":="表示使用的不是初始默认值

**如:jdk6/7 -XX:+MaxTenuringThreshold 默认值都是15,但是在使用CMS收集器后,jdk6默认4 , jdk7默认6.**

**Java代码  [收藏代码](javascript:void())**

1. [hbase96 logs]# java -version
2. java version "1.6.0\_27-ea"
3. [hbase96 logs]# java -XX:+PrintFlagsInitial | grep MaxTenuringThreshold
4. intx MaxTenuringThreshold = 15 {product}
5. [hbase96 logs]# java -XX:+PrintFlagsFinal -XX:+UseConcMarkSweepGC | grep MaxTenuringThreshold
6. intx MaxTenuringThreshold := 4 {product}
8. [zw-34-71 logs]# java -version
9. java version "1.7.0\_45"
10. [zw-34-71 logs]# java -XX:+PrintFlagsInitial | grep MaxTenuringThreshold
11. intx MaxTenuringThreshold = 15 {product}
12. [zw-34-71 logs]# java -XX:+PrintFlagsFinal -XX:+UseConcMarkSweepGC | grep MaxTenuringThreshold
13. intx MaxTenuringThreshold := 6 {product}

**3):-XX:+PrintFlagsInitial**

 在"参数处理"之前所有可设置的参数及它们的值,然后直接退出程序.

这里的"参数处理"指: 检查参数之间是否有冲突,通过ergonomics调整某些参数的值等.

**Java代码  [收藏代码](javascript:void())**

1. [hbase96 logs]# java -version
2. java version "1.6.0\_27-ea"
3. [hbase96 logs]# java -XX:+PrintFlagsInitial | grep UseCompressedOops
4. bool UseCompressedOops = **false** {lp64\_product}
5. [hbase96 logs]# java -XX:+PrintFlagsFinal | grep UseCompressedOops
6. bool UseCompressedOops := **true** {lp64\_product}

**4)CMSInitiatingOccupancyFraction 默认值是多少**

jdk6/7:

**Java代码  [收藏代码](javascript:void())**

1. #java -server -XX:+UseConcMarkSweepGC -XX:+PrintFlagsFinal | grep -P "CMSInitiatingOccupancyFraction|CMSTriggerRatio|MinHeapFreeRatio"
2. intx CMSInitiatingOccupancyFraction            = -1              {product}
3. intx CMSTriggerRatio                           = 80              {product}
4. uintx MinHeapFreeRatio                          = 40              {product}

计算公式:

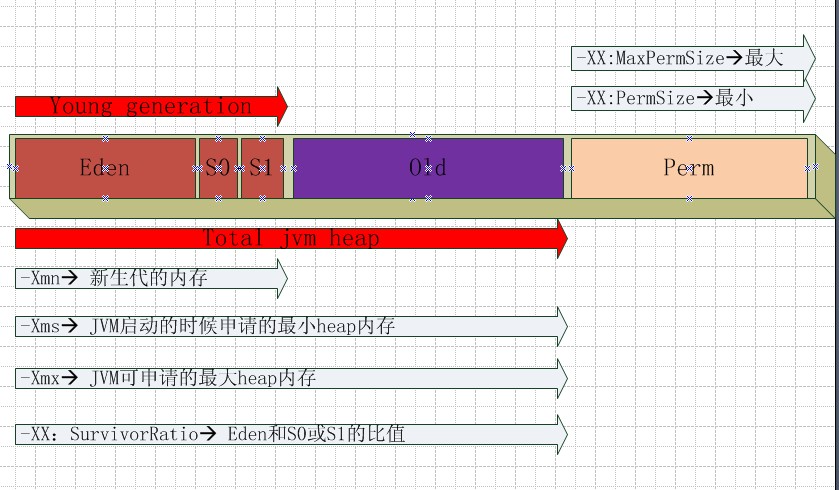
CMSInitiatingOccupancyFraction = (100 - MinHeapFreeRatio) + (CMSTriggerRatio \* MinHeapFreeRatio / 100)

最终结果: **在jdk6/7中 CMSInitiatingOccupancyFraction默认值是92% .不是网上传的68%;** 都这么传,是因为 "深入理解Java虚拟机"一书中是68%,但它用的是jdk5 , jdk5的CMSTriggerRatio默认值是20,坑爹...

**三:JVM内存区域理解和相关参数**

一图胜千言,直接上图

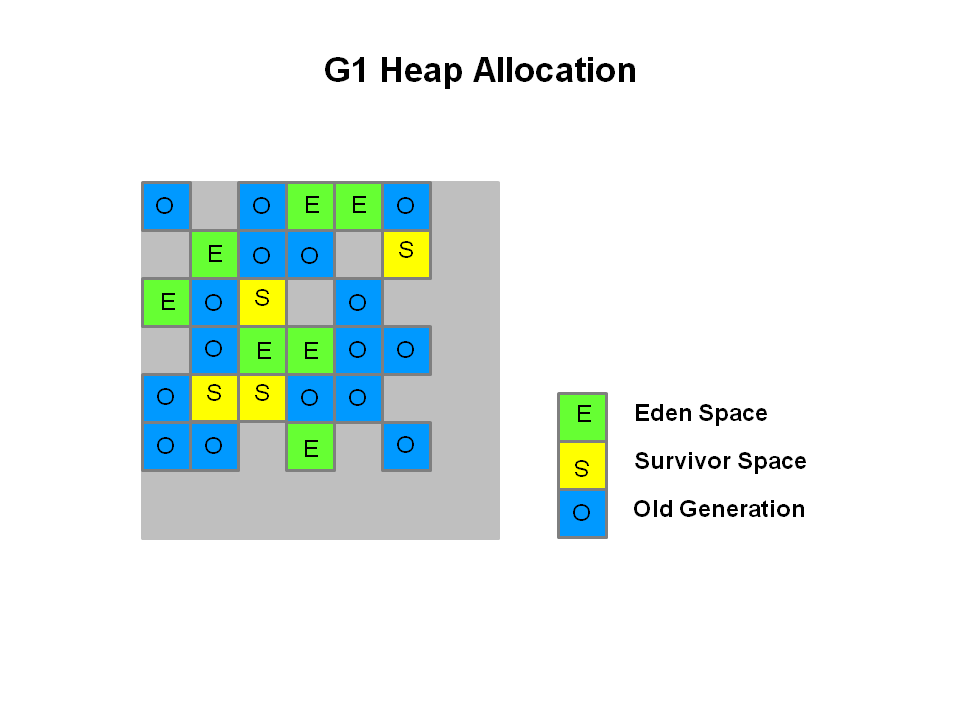
**1):物理分代图.**



物理分代是除G1之外的JVM 内存分配方式,jvm 通过-Xmx,-Xmn/newRatio等参数将jvm heap划分成物理固定大小，对于不同场景比例应该设置成多少很考验经验.

一篇JVM CMS优化讲解的非常好的文章: [how-tame-java-gc-pauses](http://java.dzone.com/articles/how-tame-java-gc-pauses)

**2) 逻辑分代图(G1)**



**逻辑分代是以后的趋势(PS:jdk8连perm都不区分了。)， 不需要使用者在纠结Xms/Xmn,SurvivorRatio等比例问题,采用动态算法调整分代大小。**