JVM中CMS收集器

这种垃圾收集器的官方名称是“Mostly Concurrent Mark and Sweep Garbage Collector”。它在Yong代使用并行，“stop-the-world”、标记复制算法，在Old代使用并发标记清除算法。

设计这种算法目的是，在回收Old代的时候去避免长时间停顿。他的实现分为两方面。第一，它不清楚Old代，而是使用空闲列表管理回收空间。第二，它在标记清除期间的大部分工作和应用程序并发执行。这意味着执行这些阶段，收集器不会刻意停顿应用线程。应当注意的是，它依然和应用程序抢占CPU时间片。默认的，这种算法的使用的线程数等于你的机器物理核心数的¼。

你在命令行通过如下图所示的选项显式指定这个收集器：

```
java -XX:+UseConcMarkSweepGC
```

如果你有意向的话，在多核机器上使用这样的组合是个不错的选择。应用用户可以明显感知个别GC阶段停顿时长减少带来的变化，快速响应给他们带来更好用户体验。大多时候，GC消耗至少几个CPU资源，而不是执行你的应用代码。在单核应用中CMS性能通常低于Parallel。

就拿上面的GC算法来说，让我们看一下这种算法在实践应用当中Minor GC和Major GC阶段如何工作：

```
2015-05-26T16:23:07.219-0200: 64.322: [GC (Allocation Failure) 64.322: [ParNew: 613404K-
>68068K(613440K), 0.1020465 secs] 10885349K->10880154K(12514816K), 0.1021309 secs] [Times:
user=0.78 sys=0.01, real=0.11 secs]
2015-05-26T16:23:07.321-0200: 64.425: [GC (CMS Initial Mark) [1 CMS-initial-mark:
10812086K(11901376K)] 10887844K(12514816K), 0.0001997 secs] [Times: user=0.00 sys=0.00, real=0.00
secs]
2015-05-26T16:23:07.321-0200: 64.425: [CMS-concurrent-mark-start]
2015-05-26T16:23:07.357-0200: 64.460: [CMS-concurrent-mark: 0.035/0.035 secs] [Times: user=0.07
sys=0.00, real=0.03 secs]
2015-05-26T16:23:07.357-0200: 64.460: [CMS-concurrent-preclean-start]
2015-05-26T16:23:07.373-0200: 64.476: [CMS-concurrent-preclean: 0.016/0.016 secs] [Times:
user=0.02 sys=0.00, real=0.02 secs]
2015-05-26T16:23:07.373-0200: 64.476: [CMS-concurrent-abortable-preclean-start]
2015-05-26T16:23:08.446-0200: 65.550: [CMS-concurrent-abortable-preclean: 0.167/1.074 secs]
[Times: user=0.20 sys=0.00, real=1.07 secs]
2015-05-26T16:23:08.447-0200: 65.550: [GC (CMS Final Remark) [YG occupancy: 387920 K (613440
K)]65.550: [Rescan (parallel) , 0.0085125 secs]65.559: [weak refs processing, 0.0000243
secs]65.559: [class unloading, 0.0013120 secs]65.560: [scrub symbol table, 0.0008345 secs]65.561:
[scrub string table, 0.0001759 secs][1 CMS-remark: 10812086K(11901376K)] 11200006K(12514816K),
0.0110730 secs] [Times: user=0.06 sys=0.00, real=0.01 secs]
2015-05-26T16:23:08.458-0200: 65.561: [CMS-concurrent-sweep-start]
2015-05-26T16:23:08.485-0200: 65.588: [CMS-concurrent-sweep: 0.027/0.027 secs] [Times: user=0.03
sys=0.00, real=0.03 secs]
2015-05-26T16:23:08.485-0200: 65.589: [CMS-concurrent-reset-start]
2015-05-26T16:23:08.497-0200: 65.601: [CMS-concurrent-reset: 0.012/0.012 secs] [Times: user=0.01
sys=0.00, real=0.01 secs]
```

Minor GC

从日志看出，GC事件第一步操作是Minor GC清理Young代。我们分析一下面示意图中收集器是如何工作的：

```
2015-05-26T16:23:07.219-0200[1]: 64.322[2]:[GC[3](
Allocation Failure[4]) 64.322: [ParNew[5]: 613404K->68068K[6]
(613440K)[7], 0.1020465 secs[8]] 10885349K->10880154K[9]
(12514816K)[10], 0.1021309 secs[11]]
[Times: user=0.78 sys=0.01, real=0.11 secs][12]
```

1. GC事件开始的时间。
2. GC开始相对于JVM启动时间时间差。
3. 区分Minor GC和Full GC的标记。它表示是一个Minor GC。
4. 收集原因。这个原因是，一次分配请求不适合在Young代的任何区域。
5. 使用收集器的名称。它表示，在Young代中使用的是并行，标记复制，“stop-the-world”算法。被设计在Old代工作的并发标记清除筹集起的组合。
6. Young代回收前后的已使用空间。
7. Young代的大小。
8. 清除时间时长
9. 回收前后Heap区已使用大小。
10. Heap大小
11. Young代垃圾收集器标记复制存活对象消耗时长。包括，CMS收集器通讯时长，对象升迁进入Old代的时间，垃圾循环收集结束后的最终清除时间。
12. GC事件时长-记录不同的类型：
    1.user-回收期间垃圾收集器线程消耗CPU事件
    2.sys-调用操作系统活着等待系统事件消耗时间
    3.real-应用停顿的时钟时间。Parallel GC时间接近（user+sys）垃圾收集器的使用线程数运行时间。应当 注意， 因为一些活动不被允许并行执行，它通常超过合计时间。

通过上面我们可以看出，回收前heap区已使用空间10,885,349K，Young代已使用空间613,404K。这意味着Old代可用空间是10,271,945K。回收后，Young代减少545,336K但是heap区仅仅减少5,195K。这意味着540,141K是Yong代升迁到Old代的对象的大小。

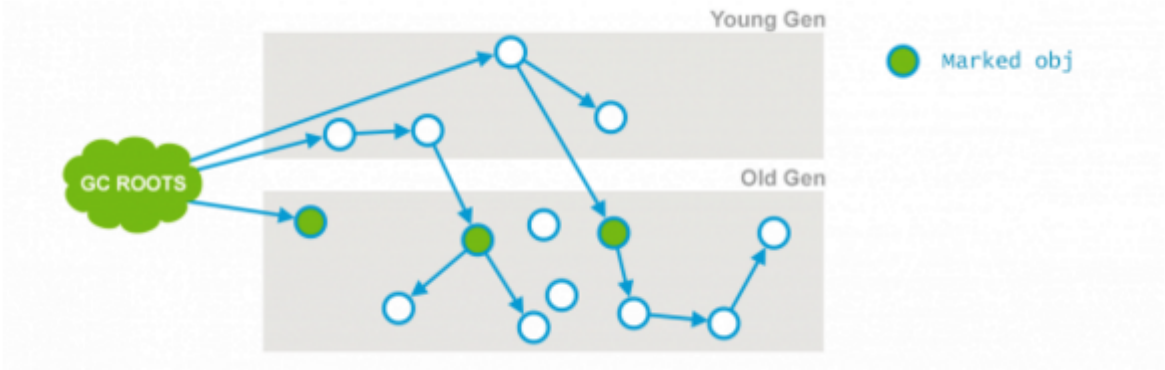
Full GC

现在，你已经习惯阅读GC日志,这章介绍日志中另外一种形式的垃圾收集器。下图中冗长的输出囊括了Old代主要垃圾收集器不同阶段。我们不一次性地简单概括日志事件，相反，我们一条一条分析日志的不同阶段。回顾一下，CMS收集器类似于下图：

```
2015-05-26T16:23:07.321-0200: 64.425: [GC (CMS Initial Mark) [1 CMS-initial-mark:
10812086K(11901376K)] 10887844K(12514816K), 0.0001997 secs] [Times: user=0.00 sys=0.00, real=0.00
secs]
2015-05-26T16:23:07.321-0200: 64.425: [CMS-concurrent-mark-start]
2015-05-26T16:23:07.357-0200: 64.460: [CMS-concurrent-mark: 0.035/0.035 secs] [Times: user=0.07
sys=0.00, real=0.03 secs]
2015-05-26T16:23:07.357-0200: 64.460: [CMS-concurrent-preclean-start]
2015-05-26T16:23:07.373-0200: 64.476: [CMS-concurrent-preclean: 0.016/0.016 secs] [Times:
user=0.02 sys=0.00, real=0.02 secs]
2015-05-26T16:23:07.373-0200: 64.476: [CMS-concurrent-abortable-preclean-start]
2015-05-26T16:23:08.446-0200: 65.550: [CMS-concurrent-abortable-preclean: 0.167/1.074 secs]
[Times: user=0.20 sys=0.00, real=1.07 secs]
2015-05-26T16:23:08.447-0200: 65.550: [GC (CMS Final Remark) [YG occupancy: 387920 K (613440
K)]65.550: [Rescan (parallel) , 0.0085125 secs]65.559: [weak refs processing, 0.0000243
secs]65.559: [class unloading, 0.0013120 secs]65.560: [scrub symbol table, 0.0008345 secs]65.561:
[scrub string table, 0.0001759 secs][1 CMS-remark: 10812086K(11901376K)] 11200006K(12514816K),
0.0110730 secs] [Times: user=0.06 sys=0.00, real=0.01 secs]
2015-05-26T16:23:08.458-0200: 65.561: [CMS-concurrent-sweep-start]
2015-05-26T16:23:08.485-0200: 65.588: [CMS-concurrent-sweep: 0.027/0.027 secs] [Times: user=0.03
sys=0.00, real=0.03 secs]
2015-05-26T16:23:08.485-0200: 65.589: [CMS-concurrent-reset-start]
2015-05-26T16:23:08.497-0200: 65.601: [CMS-concurrent-reset: 0.012/0.012 secs] [Times: user=0.01
sys=0.00, real=0.01 secs]
```

但要记住，实际上Old代并发收集期间的任何时候，Young代都会发生Minor GC。在这种情况下，可以看到上面的Major GC和Minor GC 事件交替执行。

阶段1：初始化标记。CMS期间两次“”stop-the-world“”中的一次在这个阶段发生。这个阶段的目标是标记Old代所有的GC root可达对象，Young代中被引用的对象。Old代独立回收这点很重要。

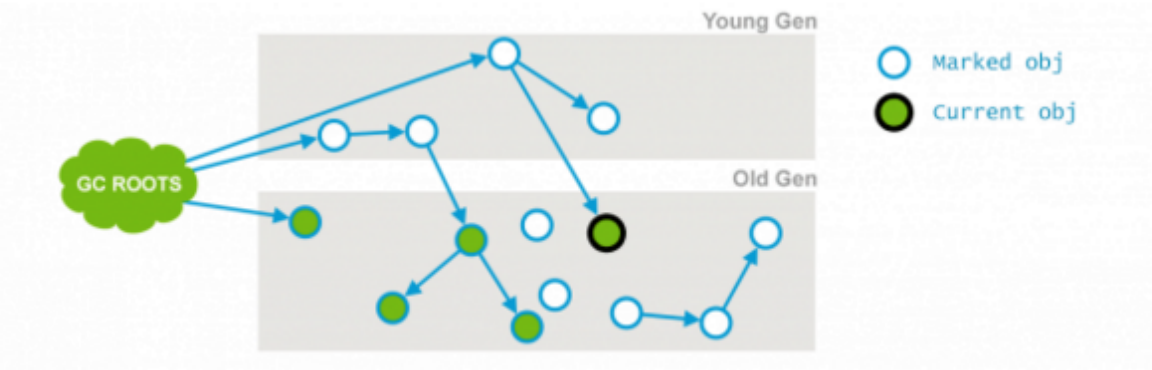


2015-05-26T16:23:07.321-0200: 64.42[1]: [GC ([CMS Initial Mark][2] [1 CMS-initial-mark:
10812086K[3] (11901376K)[4]] 10887844K[5] (12514816K)[6],
0.0001997 secs] [Times: user=0.00 sys=0.00, real=0.00 secs][7]

1. GC时间开始时间，包含时钟时间和距离相对于JVM启用的时间，简单略过贯穿以后的阶段的类似的概念。
2. 回收阶段-这时的“”Initial Mark“”收集GC Roots对象。
3. Old代当前已使用空间。
4. Old代大小。
5. heap区大小。

6. 这次过程持续时间-user,sys,real的实际时间。

阶段2: 并发标记。这个阶段期间，垃圾回收器扫描整个Old代，标记所有存活对象，从上一个“”Initial Mark“”阶段发现的所有Root对象。“”Concurrent Mark“”阶段，正如名称所示，和应用线程并发运行，不会停顿应用线程。由于标记期间应用改变对象引用，Old代不是所有存活对象也许被标记。

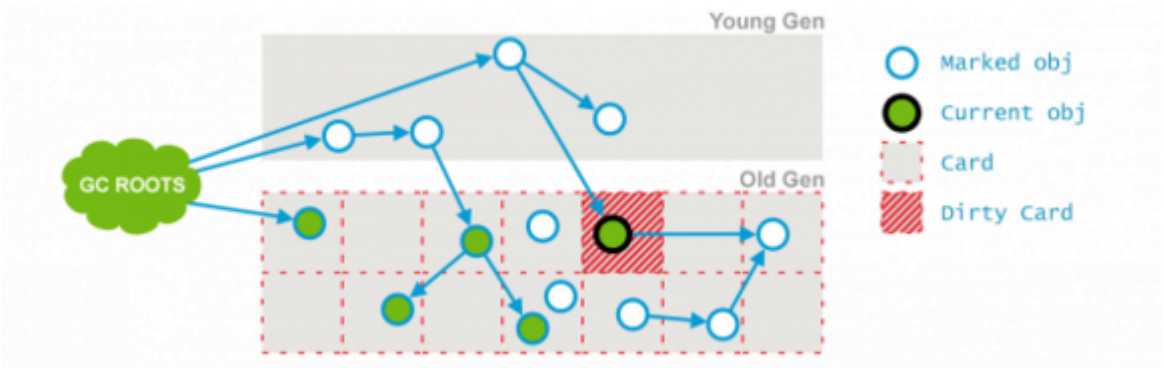


上面的示意图说明，标记线程并发地移除没有被“”Current obj“”指针引用的对象。

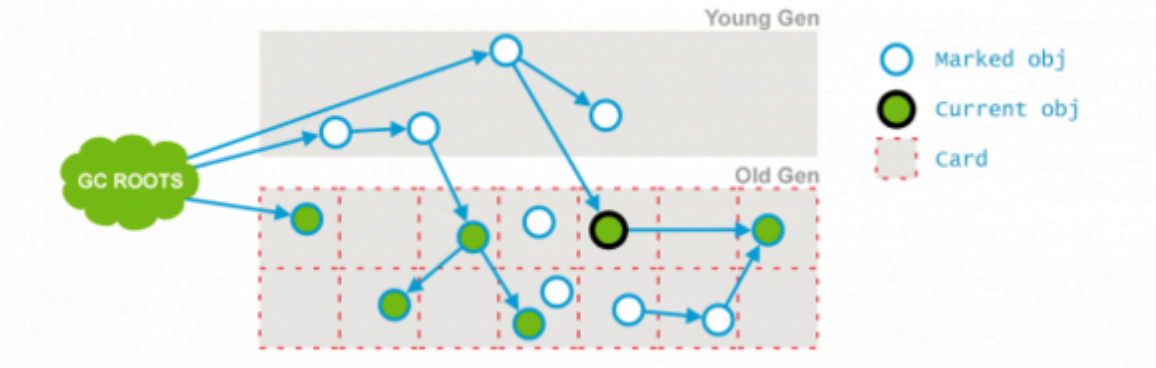
```
2015-05-26T16:23:07.321-0200: 64.425: [CMS-concurrent-mark-start]
2015-05-26T16:23:07.357-0200: 64.460: [CMS-concurrent-mark 1: 035/0.035 secs 2]
[Times: user=0.07 sys=0.00, real=0.03 secs] 3
```

1. 收集阶段，“”Concurrent Mark“”-扫描整个Old区,标记所有存活对象。
2. 阶段持续时间，展示相对钟墙时间的消耗时间。
3. “”Times“”部分，对于并发阶段中计算并发标记开始时间更有意义，不只是包含并发标记完成工作的时间。

阶段3:并发清除 。这是一个并发阶段，和应用线程并行执行,不会引起线程停顿。虽然上一个阶段和应用线程并发执行，但是一些引用被改变。无论何时发生，JVM标记heap区（“”Card“”）包含突变成“”dirty“”的对象（了解 Card Marking）。



在预清理阶段，这些“”脏“”对象被占用，被他们引用的对象依然被标记，当它释放后，Card被清除。



补充一点，一些必要的最后标记的准备工作被执行。

```
2015-05-26T16:23:07.357-0200: 64.460: [CMS-concurrent-preclean-start]
2015-05-26T16:23:07.373-0200: 64.476: [CMS-concurrent-preclean[1]: 0.016/0.016 secs[2]]
[Times: user=0.02 sys=0.00, real=0.02 secs][3]
```

1. 收集阶段-“Concurrent Preclean”阶段-前一个标记阶段期间引用发生变化。
2. 相对于钟墙时间的消耗时间。
3. “”Times“”部分，对于并发阶段中计算并发标记开始时间更有意义，不只是包含并发标记完成工作的时间。

阶段4：并发预处理。又一个并发的，不需要“”stop-the-world“”的阶段尝试去尽可能多消除最终标记由于停顿的影响。由于他做循环做着同样的事情直到遇到一个中断式条件（例如迭代次数，有用工作完成的数量，钟墙时间消耗，等等），因此这个阶段的准确好事取决于这些因素。

```
2015-05-26T16:23:07.373-0200: 64.476: [CMS-concurrent-abortable-preclean-start]
2015-05-26T16:23:08.446-0200: 65.550: [CMS-concurrent-abortable-preclean[1]: 0.167/1.074 secs[2]]
[Times: user=0.20 sys=0.00, real=1.07 secs][3]
```

1. “”并发预清理“”阶段。
2. 整个阶段消耗时间，暂时耗费时间和单独的钟墙时间。有趣的一点是，user time大大小于时钟时间。通常我们看到时钟时间小于user time，意味着一些工作并行执行，因此时钟时间小于cpu 时间。现在我们看一些一些任务：0.167s 的cpu 时间，垃圾收集器线程等待很长时间。本质上，他们尽量去避开STW阶段。默认的，这各阶段持续5s。
3. “”Times“”部分，对于并发阶段中计算并发标记开始时间更有意义，不只是包含并发标记完成工作的时间。

这个阶段会明显影响到即将带来的“”stop-the-world“”阶段，有很多的配置选项和失败模式。

阶段5：最终标记。这是第二个也是最后一次事件发生期间“”stop-the-world“”的阶段。“”stop-the-world“”的目的是最后标记Old代中所有的存活对象。由于预处理阶段是并发的，它们也许赶不上应用线程改变的速度。“”stop-the-world“”需要完成这些考验。

通常当年轻代为了尽可能多清除几次紧接着发生的几次“”stop-the-world“”而CM去执行最终标记。

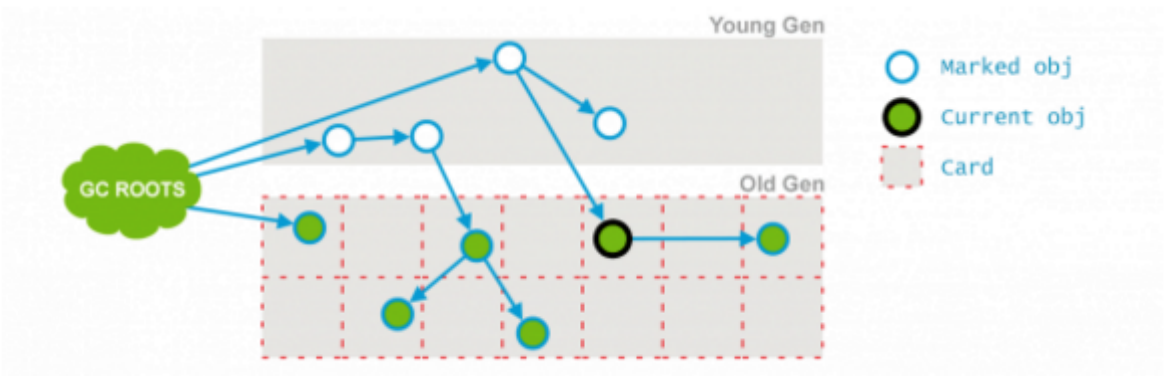
这次事件看起来比之前的阶段复杂得多：

```
2015-05-26T16:23:08.447-0200: 65.550[1]: [GC ( CMS Final Remark[2]) [
YG occupancy: 387920 K (613440 K)[3]]65.550: [Rescan (parallel) , 0.0085125 secs][4]65.559: [
weak refs processing, 0.0000243 secs]65.559[5]: [ class unloading, 0.0013120 secs]65.560[6]: [
scrub string table, 0.0001759 secs[7]][1 CMS-remark: 10812086K(11901376K)[8]]
11200006K(12514816K)[9], 0.0110730 secs[10]] [ [Times: user=0.06 sys=0.00, real=0.01 secs][11]
```

1. GC 事件开始时间，包括时钟时间和相对于JVM启动的时间
2. 收集阶段-“”“”Final Remark”，标记Old代所有的存活对象，包括并发标记阶段新建的，应用改变的对象。
3. 当前已使用的空间和Young代的空间大小
4. “扫描”-应用线程停顿期间标记所有的存活对象。这种情况下并发扫描花费0.0085125 s。
5. 第一个子阶段处理弱引用d的时间戳。
6. 第二个子阶段卸载无用class对象的时间戳。
7. 最后一个子阶段清除拥有class级别元数据的符号和字符串表，和其内部的String。这个阶段通常包括时钟时间。
8. 这个阶段过后Old代已使用和Old代空间大小。
9. 阶段过后heap已使用和heap空间大小。
10. 阶段持续时长。
11. 阶段持续时长，user, system and real 的时间。

最后标记阶段过后，Old代所有存活对象被标记，垃圾收集器去准备去回收Old代中的无用对象。

阶段6：并发清除。和应用线程并发执行，不需要“”stop-the-world“”。这个阶段的目的清除无用对象，以备以后使用。




```
2015-05-26T16:23:08.458-0200: 65.561: [CMS-concurrent-sweep-start] 2015-05-26T16:23:08.485-
0200: 65.588: [ CMS-concurrent-sweep[1]: 0.027/0.027 secs[2]] [
[Times: user=0.03 sys=0.00, real=0.03 secs][3]
```

1. 清除没有标记和无用的对象以回收空间。
2. 展示运行时间和墙钟时间。
3. time 部分对并发阶段有重大意义，正如并发标记开始时间，包含但不限于并发标记的工作。

阶段7: 并发重置。并发执行阶段，重置CMS算法内部的数据结构，为下一个周期做准备。

```
2015-05-26T16:23:08.485-0200: 65.589: [CMS-concurrent-reset-start] 2015-05-26T16:23:08.497-
0200: 65.601: [ CMS-concurrent-reset [1]: 0.012/0.012 secs [2] ] [
[Times: user=0.01 sys=0.00, real=0.01 secs] [3]
```

1. 并发阶段，重置CMS算法内部的数据结构，为下次循环做准备。2.展示运行时间和墙钟时间。
2. time 部分对并发阶段有重大意义，正如并发标记开始时间，包含但不限于并发标记的工作。

总而言之，CMS收集器通过并发线程卸载大量工作而不用去停顿应用线程在减少阶段运行时间上做了伟大的工作。然而，他也存在一些缺点，最值得的注意的是造成Old代碎片化，阶段持续时间在某些情况下缺乏可预见性，尤其在大heap区。