JVM调优：选择合适的GC collector （三）

标签： jvm cms parallel user application processing

2011-03-12 23:28 12165人阅读 评论(3) 收藏 举报

分类： JAVA（15）

CMS Collector

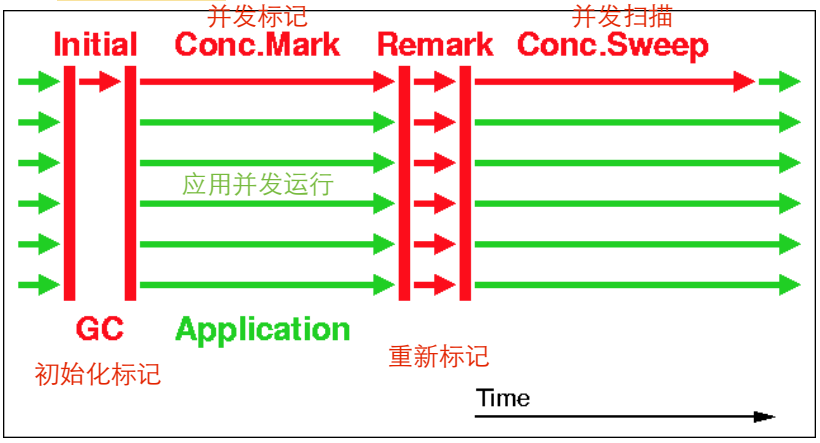
在很多地方，CMS Collector常被翻译成“并发”，而ParallelGC被称为“并行”，但中文里，这两词的区分度并不明显。事实上，所谓的Parallel是指，在执行GC的时候将会有多个GC线程共同工作，但是，在执行GC的过程中仍然是“stop-the-world”。CMS的区别在于，在执行GC的时候，GC线程是不需要暂停application的线程，而是和它们“并发”一起工作。

最关键的部分

所以，采用CMS的原因就在于它可以提供最低的pause time。

最低的停顿时间

回到CMS的示意图：



这张图表示的是CMS在执行Full GC的过程，这个过程包括了6个步骤：

STW initial mark

Concurrent marking

Concurrent precleaning

STW remark

Concurrent sweeping

Concurrent reset

STW表示的意思就是“stop-the-world”。

所以，CMS也并不是完全不会暂停application的，在这六个步骤中，有两个步骤需要STW，分别是：initial mark和remark（如图所示）。而其它的四个步骤是可以和application“并发”执行。initial mark是由一个GC thread来执行，它的暂停时间相对比较短。而remark过程的暂停时间要比initial mark更长，且通常由多个thread执行。

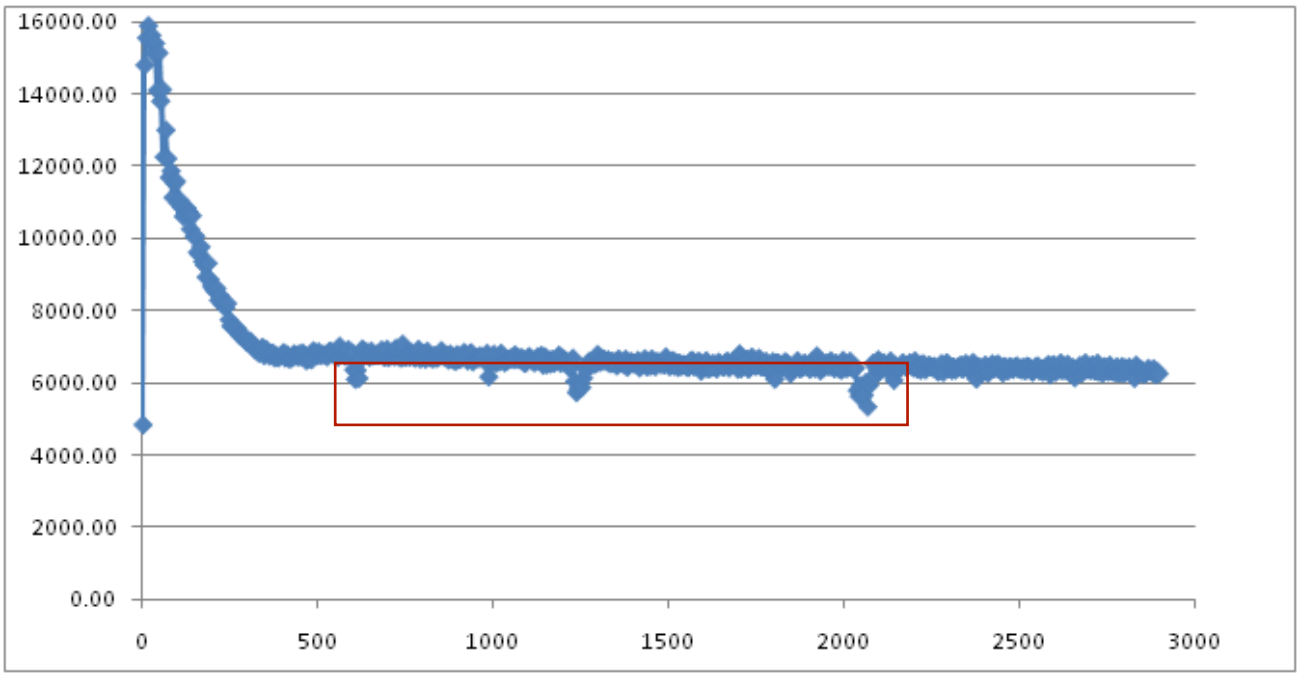
这六个步骤的具体内容我就不写了（其实俺也似懂非懂），有兴趣的可以参考【1】，【2】。

接下来看看实验结果。

实验结果

JVM参数如下：

```
Java -jar -Xms10g -Xmx15g -XX:+UseConcMarkSweepGC -XX:NewSize=6g -XX:MaxNewSize=6g
-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps  -Xloggc:./log/gc.log Slaver.jar
```



从图中可以看出，采用CMS Collector的最大的不同在于它已经没有了那个“大峡谷”，意味着发生的Full GC并没有导致系统的throughput降低到0。虽然图中也有几次曲线的下降（事实上，这就是发生Full GC的地方），

但是曲线的下降很微弱，并且，持续时间也不太长。

整体而言，系统的平均throughput大概在7000 – 6000 request/sec之间，要比Serial GC好，但略低于Parallel GC。

看一个Minor GC的log：

519.514: [GC 519.514: [ParNew: 5149852K->83183K(5662336K), 0.0831770 secs] 6955196K->1905793K(9856640K), 0.0833560 secs] [Times: **user=0.57** sys=0.03, **real=0.08 secs**]

采用CMS GC在发生Minor GC的时候采用的collector类似于Parallel GC，log也和Parallel GC的log类似。不多解释。 并发收集

重点在于Full GC的log：

2051.800: [GC [1 **CMS-initial-mark** : 6040466K(6555784K)] 6161554K(12218120K), 0.1028810 secs] [Times: user=0.10 sys=0.00, **real=0.11 secs**]

2051.903: [**CMS-concurrent-mark-start**]

2059.492: [GC 2059.492: [ParNew: 5153779K->129958K(5662336K), 0.1145560 secs] 11194245K->6189004K(12218120K), 0.1147330 secs] [Times: user=0.82 sys=0.04, real=0.11 secs]

2067.229: [GC 2067.229: [ParNew: 5163174K->92868K(5662336K), 0.1136260 secs] 11222220K->6170498K(12218120K), 0.1137820 secs] [Times: user=0.82 sys=0.00, real=0.12 secs]

2075.005: [GC 2075.005: [ParNew: 5126084K->126301K(5662336K), 0.1205450 secs] 11203714K->6222479K(12218120K), 0.1207120 secs] [Times: user=0.84 sys=0.01, real=0.12 secs]

2077.487: [CMS-concurrent-mark: 25.231/25.584 secs] [Times: user=158.91 sys=22.71, **real=25.58 secs**]

2077.487: [**CMS-concurrent-preclean-start**]

2078.512: [CMS-concurrent-preclean: 0.961/1.025 secs] [Times: user=5.97 sys=1.20, real=1.03 secs]

2078.513: [CMS-concurrent-abortable-preclean-start]

2082.466: [GC 2082.467: [ParNew: 5159517K->89444K(5662336K), 0.1162740 secs] 11255695K->6204092K(12218120K), 0.1164340 secs] [Times: user=0.82 sys=0.01, real=0.12 secs]

CMS: abort preclean due to time 2083.642: [CMS-concurrent-abortable-preclean: 4.933/5.129 secs] [Times: user=31.10 sys=4.89, real=5.12 secs]

2083.644: [GC[YG occupancy: 877128 K (5662336 K)]2083.644: [**Rescan** (parallel) , 0.5058390 secs]2084.150: [weak refs processing, 0.0000630 secs] [1 CMS-remark: 6114647K(6555784K)] 6991776K(12218120K), 0.5060260 secs] [Times: **user=3.35** sys=0.01, **real=0.50 secs**]

2084.150: [**CMS-concurrent-sweep-start**]

2090.416: [GC 2090.416: [ParNew: 5122660K->124614K(5662336K), 0.1247190 secs] 11237258K->6257803K(12218120K), 0.1248800 secs] [Times: user=0.88 sys=0.00, real=0.12 secs]

2095.868: [CMS-concurrent-sweep: 11.593/11.718 secs] [Times: user=70.11 sys=11.53, real=11.72 secs]

2095.896: [**CMS-concurrent-reset-start**]

2096.124: [CMS-concurrent-reset: 0.227/0.227 secs] [Times: user=1.33 sys=0.19, real=0.23 secs]

Full GC的log和其它的collector完全不同，简单解释一下：

log的第一行CMS-initial-mark 表示CMS执行它的第一步：initial mark。它花费的时间是real=0.11 secs，由于这一步骤是STW的，所以整个application被暂停了0.11秒。并且，user time和real time相差不大，所以确实是只有一个thread执行这一步；

CMS-concurrent-mark-start 表示开始执行第二步骤：concurrent marking。它执行时间是real=25.58 secs，但因为这一步是可以并发执行的，所以系统在这段时间内并没有暂停；

CMS-concurrent-preclean-start 表示执行第三步骤：concurrent precleaning。同样的，这一步也是并发执行；

最重要的是这一条语句：

2083.644: [GC[YG occupancy: 877128 K (5662336 K)]2083.644: [Rescan (parallel) , 0.5058390 secs]2084.150: [weak refs processing, 0.0000630 secs] [1 CMS-remark: 6114647K(6555784K)] 6991776K(12218120K), 0.5060260 secs] [Times: user=3.35 sys=0.01, real=0.50 secs]

这一步是步骤：remark的执行结果，它的执行时间是real=0.50 secs。因为这是STW的步骤，并且它的pause time一般是最长的，所以这一步的执行时间会直接决定这次Full GC对系统的影响。这次只执行了0.5秒，系统的throughput未受太大影响。

后面的log分别记录了concurrent-sweep和concurrent-reset。就不多说了，更详细的log分析可见【3】。

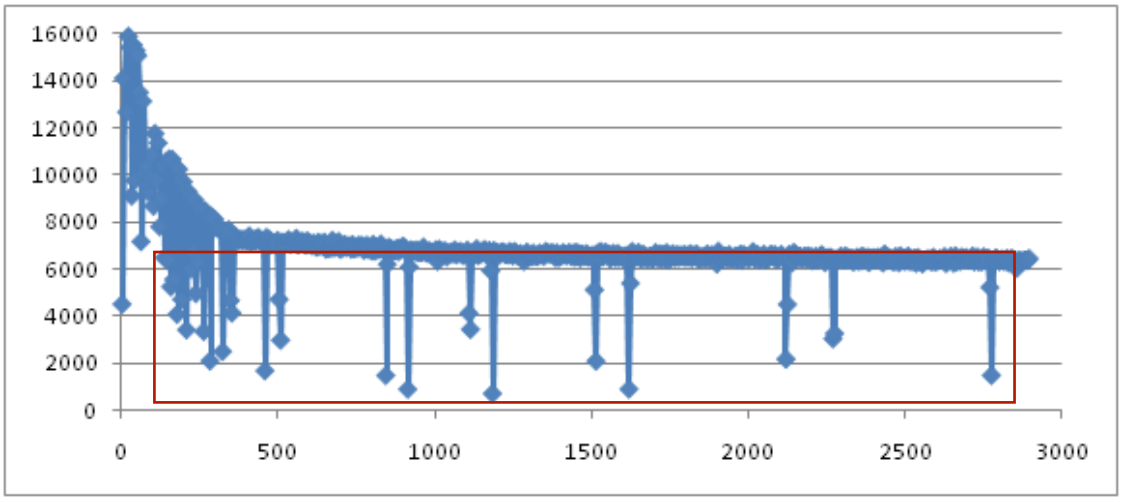
结论

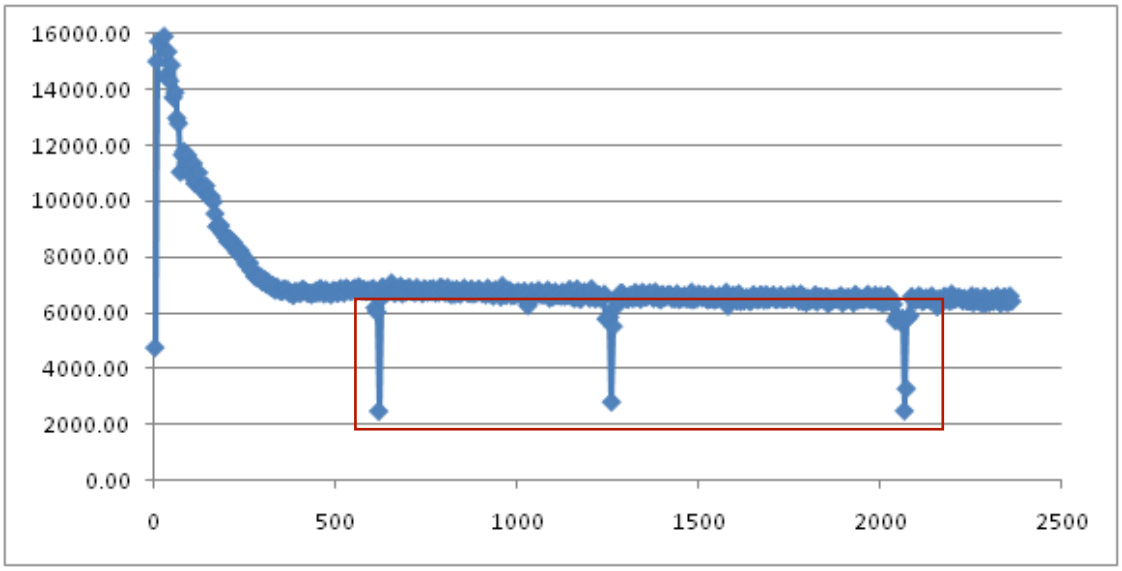
关键原因：

比较了这几种Collector，发现CMS应该是最适合我的系统的。因为它并不会因为Full GC而在未来的某一时刻突然停滞工作。这一点其实在很多系统中都是非常重要的，比如Web Server ....

多说一点

贴另外两张实验结果图：





这两张图也是采用CMS Collector实验得到的。区别在于使用了不同的参数。第一张图是CMS Collector采用了incremental model的方式：

java -jar -Xms10g -Xmx15g -XX:+UseConcMarkSweepGC -XX:+CMSIncrementalMode -XX:NewSize=6g -XX:MaxNewSize=6g -verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintHeapAtGC -Xloggc:./log/gc.log Slaver.jar

而第二张图则调整了AbortablePrecleanTime的值：

java -jar -Xms10g -Xmx15g -XX:+UseConcMarkSweepGC -XX:CMSMaxAbortablePrecleanTime=15 -XX:NewSize=6g -XX:MaxNewSize=6g -verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -Xloggc:./log/gc.log Slaver.jar

这两个参数有什么用不多解释，只是想说明，即便选择了合适的Collector，也可能由于其它参数的设置而产生巨大差异。

JVM的调优确实不简单。这个并没有绝对的准则或者公式，唯一的好办法就是实验。

(The End)

Reference：

【1】Did You Know ...

【2】Java SE 6 HotSpot[tm] Virtual Machine Garbage Collection Tuning

【3】Understanding CMS GC Logs