CMS 垃圾回收机制

1：CMS 是什么

CMS 全称 Concurrent Low Pause Collector，，是一款并发的、使用标记-清除算法的垃圾回收器，是 jdk1.4 后期版本开始引入的新 gc 算法，在 jdk5 和 jdk6 中得到了进一步改进，它的主要适合场景是对响应时间的重要性需求 大于对吞吐量的要求，能够承受垃圾回收线程和应用线程共享处理器资源，并且应用中存在比较多的长生命周期的对象的应用。CMS 是用于对 tenured generation 的回收，也就是年老代的回收，目标是尽量减少应用的暂停时间，减少 full gc 发生的几率，利用和应用程序线程并发的垃圾回收线程来标记清除年老代。在我们的应用中，因为有缓存的存在，并且对于响应时间也有比较高的要求，因此希 望能尝试使用 CMS 来替代默认的 server 型 JVM 使用的并行收集器，以便获得更短的垃圾回收的暂停时间，提高程序的响应性。

2：CMS 有什么用？

CMS 以获取最小停顿时间为目的。

在一些对响应时间有很高要求的应用或网站中，用户程序不能有长时间的停顿，CMS 可以用

3: CMS 执行过程

初始标记(STW initial mark)

并发标记(Concurrent marking)

并发预清理(Concurrent precleaning)

重新标记(STW remark)

并发清理(Concurrent sweeping)

并发重置(Concurrent reset)

初始标记 ：在这个阶段，需要虚拟机停顿正在执行的任务，官方的叫法 STW(Stop The Word)。这个过程从垃圾回收的"根对象"开始，只扫描到能够和"根对象"直接关联的对象，并作标记。所以这个过程虽然暂停了整个 JVM，但是很快就完成了。

并发标记 ：这个阶段紧随初始标记阶段，在初始标记的基础上继续向下追溯标记。并发标记阶段，应用程序的线程和并发标记的线程并发执行，所以用户不会感受到停顿。

并发预清理 ：并发预清理阶段仍然是并发的。在这个阶段，虚拟机查找在执行并发标记阶段新进入老年代的对象(可能会有一些对象从新生代晋升到老年代， 或者有一些对象被分配到老年代)。通过重新扫描，减少下一个阶段"重新标记"的工作，因为下一个阶段会 Stop The World。

重新标记 ：这个阶段会暂停虚拟机，收集器线程扫描在 CMS 堆中剩余的对象。扫描从"跟对象"开始向下追溯，并处理对象关联。

并发清理 ：清理垃圾对象，这个阶段收集器线程和应用程序线程并发执行。

并发重置 ：这个阶段，重置 CMS 收集器的数据结构，等待下一次垃圾回收。

4: CMS 有什么缺点？

1)CMS 回收器采用的基础算法是 Mark-Sweep。所有 CMS 不会整理、压缩堆空间。这样就会有一个问题：经过 CMS 收集的堆会产生空间碎片。 CMS 不对堆空间整理压缩节约了垃圾回收的停顿时间，但也带来的堆空间的浪费。为了解决堆空间浪费问题，CMS 回收器不再采用简单的指针指向一块可用堆空 间来为下次对象分配使用。而是把一些未分配的空间汇总成一个列表，当 JVM 分配对象空间的时候，会搜索这个列表找到足够大的空间来 hold 住这个对象

2)需要更多的 CPU 资源。从上面的图可以看到，为了让应用程序不停顿，CMS 线程和应用程序线程并发执行，这样就需要有更多的 CPU，单纯靠线程切 换是不靠谱的。并且，重新标记阶段，为空保证 STW 快速完成，也要用到更多的甚至所有的 CPU 资源。当然，多核多 CPU 也是未来的趋势

3) CMS 的另一个缺点是它需要更大的堆空间。因为 CMS 标记阶段应用程序的线程还是在执行的，那么就会有堆空间继续分配的情况，为了保证在 CMS 回 收完堆之前还有空间分配给正在运行的应用程序，必须预留一部分空间。也就是说，CMS 不会在老年代满的时候才开始收集。相反，它会尝试更早的开始收集在回收完成之前，堆没有足够空间分配;默认当老年代使用 68%的时候，CMS 就开始行动了。 –XX:CMSInitiatingOccupancyFraction

5: 其他

Cms 其他参数

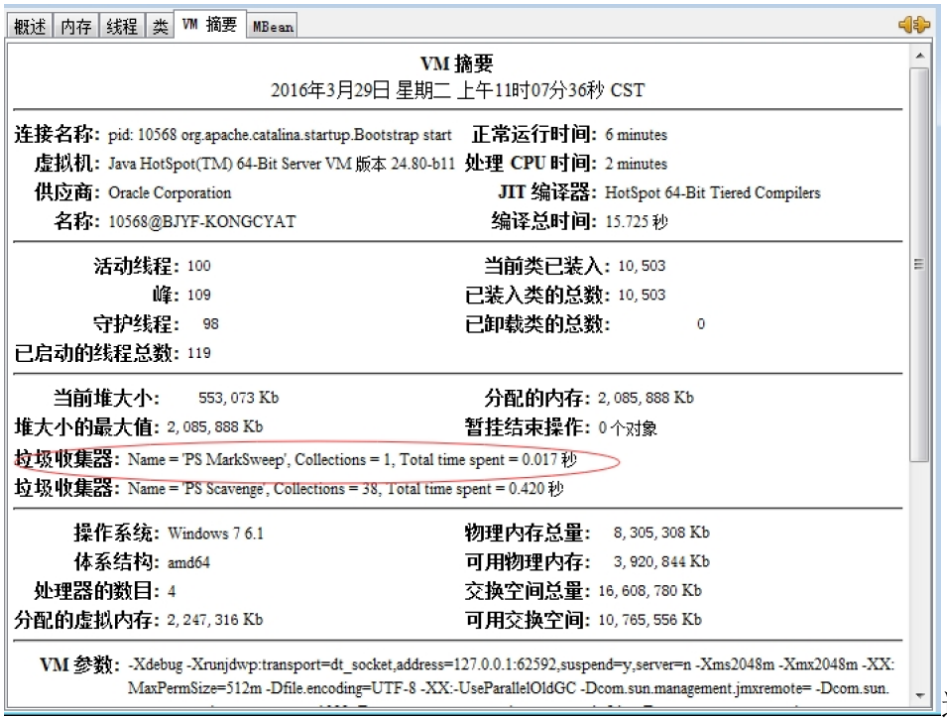
-XX:+UseConcMarkSweepGC	开启此参数使用 CMS 搜集器
-XX:+UseCMSInitiatingOccupancyOnly	使用手动定义初始化定义开始 CMS 收集;禁止 hostspot 自行触发 CMS GC 默认值为 true;与 CMSInitiatingOccupancyFraction 搭配使用
-XX:CMSInitiatingOccupancyFraction=70	设定 CMS 在对内存占用率达到 70%的时候开始 GC
XX: CMSFullGCsBeforeCompaction	在上一次 CMS 并发 GC 执行过后 ,到底还要再执行多少次 full GC 才会做压缩。默认值为 0
-XX:+CMSScavengeBeforeRemark	CMS GC 前启动一次 ygc，目的在于减少 old gen 对 ygc gen 的引用，降低 remark 时的开销

6: 实际应用

jdk1.6 默认的 gc 策略是 UseParallelGC 年轻代并行年老代串行，jdk1.7 默认的 gc 策略是 UseParallelOldGC 年轻代并行年老代也并行

jdk1.6 设置参数-XX:+UseParallelOldGC 就设置成年轻代和年老代都并行了。

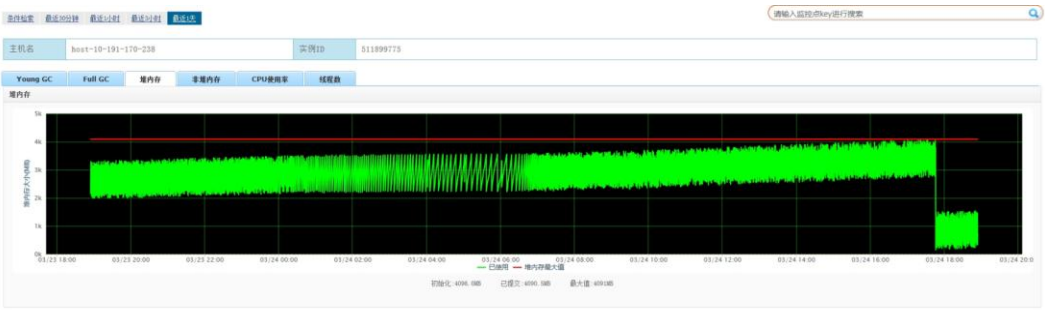
ps:在测试过程中发现加-XX:+UseParallelOldGC 参数，虽然从 JVM 参数看生效了，但是通过 jconsole 查看，加这个参数前后的老年代的收集器都是 PS MarkSweep 收集器

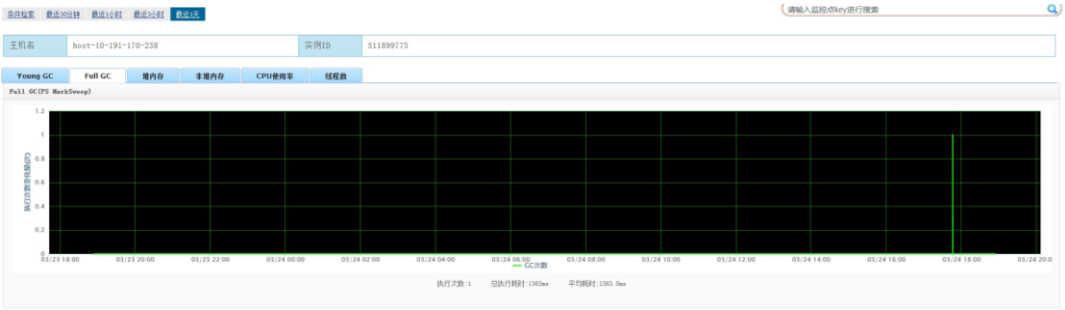


这只是那个

MXBean 的名字叫做 PS MarkSweep 而已。只用 UseParallelGC 与用了 UseParallelOldGC 背后实际的 collector 不一样，前者在 HotSpot VM 内是 PSMarkSweep，后者是 PSParallelCompact

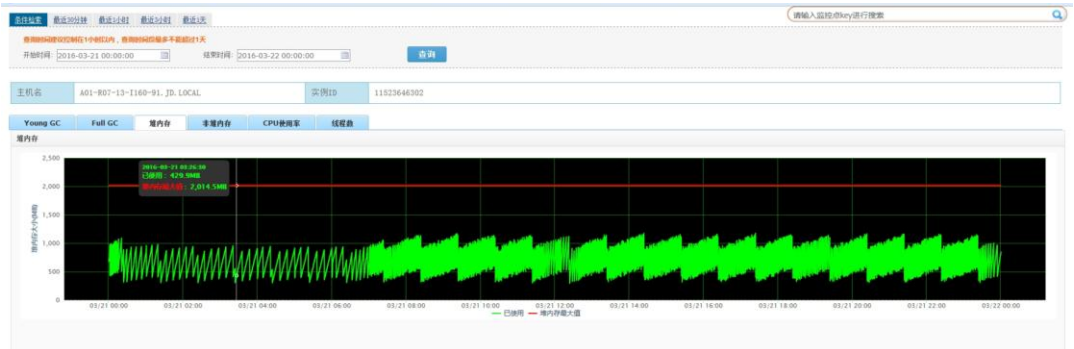
无论是 1.6 还是 1.7,年老代在**无外力**的情况下,是要达到峰值才进行 gc，如下图

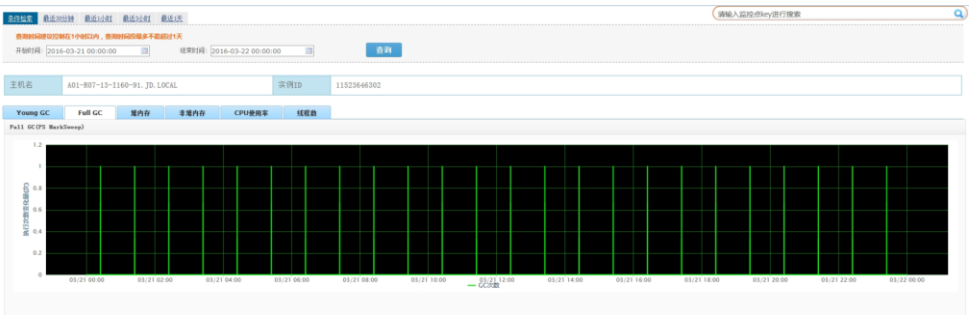




二、tomcat6 会定时执行 gc，默认间隔是一小时

为啥很多应用堆内存、非堆内存使用量都较少,但是也触发了 full gc 呢，如图





观察 full gc 的情况，是不是执行的很有规律，像是定时执行的，是的，tomcat6 **是会定时执行 gc，默认的时间是 1 小时，**这样预防了内存泄露的情况，但是也影响到了收集器的原本的性能，所以在 **tomcat 7028 和 6036 之后的版本里，把延迟时间调整了，不会再每小时调用一次了**

而且对于内存增长很快，一个小时内就把堆内存打满的情况（当然排除是堆内存大小设置的不合理，或代码有问题情况外），这个防护显然是无效的，还是应该选择合适系统的收集器，而不是单依赖这个防护

三、gc 收集器

表 3-1　垃圾收集相关的常用参数

参　数	描　述
UseSerialGC	虚拟机运行在 Client 模式下的默认值，打开此开关后，使用 Serial + Serial Old 的收集器组合进行内存回收
UseParNewGC	打开此开关后，使用 ParNew + Serial Old 的收集器组合进行内存回收
UseConcMarkSweepGC	打开此开关后，使用 ParNew + CMS + Serial Old 的收集器组合进行内存回收。Serial Old 收集器将作为 CMS 收集器出现 Concurrent Mode Failure 失败后的后备收集器使用
UseParallelGC	虚拟机运行在 Server 模式下的默认值，打开此开关后，使用 Parallel Scavenge + Serial Old（PS MarkSweep）的收集器组合进行内存回收
UseParallelOldGC	打开此开关后，使用 Parallel Scavenge + Parallel Old 的收集器组合进行内存回收

这个图片引用自周志明先生的《深入理解 Java 虚拟机：JVM 高级特性与最佳实践》中表 3-1，P65

图片中是参数对应的收集器，在实践中我们发现对于大多数的应用领域，评估一个垃圾收集(GC)算法如何根据如下两个标准：

1. 吞吐量越高算法越好
2. 暂停时间越短算法越好

串行的就不用多说了，单处理器可以使用

Parallel Scavenge 收集器是基于吞吐量的，重点考虑的是高性能。为了增加吞吐量，gc 的次数会控制，如第一点中的截图所示，快达到堆内存峰值时才执行一次 full gc ,gc 一次时间比较长,我们都知道 gc 时，服务是暂停的，对于交互性的系统来说，显然是不行的。所以 Parallel Scavenge 比较适合没有交互的系统，如 worker、mq 消费的系统

CMS 收集器是基于暂停时间的，重点考虑的是响应时间，暂停时间要小。这个算法不会等到年老代满了才进行回收，而是达到一定比例（可用参数设置）就进行回收（如下图），对

于交互型的系统比较适合此收集器