JDK5.0垃圾收集优化之--Don't Pause

算法多线程JVM应用服务器CMS

作者：江南白衣，最新版链接：http://blog.csdn.net/calvinxiu/archive/2007/05/18/1614473.aspx，

原本想把题目更简单的定为--《不要停》的，但还是自己YY一下就算了。

Java开发Server最大的障碍，就是JDK1.4版之前的的串行垃圾收集机制会引起长时间的服务暂停，明白原理后，想想那些用JDK1.3写Server的先辈，不得不后怕。

好在JDK1.4已开始支持多线程并行的后台垃圾收集算法，JDK5.0则优化了默认值的设置。

一、参考资料：

1. Tuning Garbage Collection with the 5.0 Java Virtual Machine 官方指南。
2. Hotspot memory management whitepaper 官方白皮书。
3. Java Tuning White Paper 官方文档。
4. FAQ about Garbage Collection in the Hotspot 官方FAQ，JVM1.4.2。
5. Java HotSpot 虚拟机中的垃圾收集 JavaOne2004上的中文ppt
6. A Collection of JVM Options JVM选项的超完整收集。

二、基本概念

1、堆(Heap)

JVM管理的内存叫堆。在32Bit操作系统上有1.5G-2G的限制，而64Bit的就没有。

JVM初始分配的内存由-Xms指定，默认是物理内存的1/64但小于1G。

JVM最大分配的内存由-Xmx指定，默认是物理内存的1/4但小于1G。

默认空余堆内存小于40%时，JVM就会增大堆直到-Xmx的最大限制，可以由-XX:MinHeapFreeRatio=指定。

默认空余堆内存大于70%时，JVM会减少堆直到-Xms的最小限制，可以由-XX:MaxHeapFreeRatio=指定。

服务器一般设置-Xms、-Xmx相等以避免在每次GC 后调整堆的大小，所以上面的两个参数没啥用。

2.基本收集算法

1. 复制：将堆内分成两个相同空间，从根(ThreadLocal的对象，静态对象）开始访问每一个关联的活跃对象，将空间A的活跃对象全部复制到空间B，然后一次性回收整个空间A。

因为只访问活跃对象，将所有活动对象复制走之后就清空整个空间，不用去访问死对象，所以遍历空间的成本较小，但需要巨大的复制成本和较多的内存。

2. 标记清除**(mark-sweep)**：收集器先从根开始访问所有活跃对象，标记为活跃对象。然后再遍历一次整个内存区域，把所有没有标记活跃的对象进行回收处理。该算法遍历整个空间的成本较大暂停时间随空间大小线性增大，而且整理后堆里的碎片很多。

3. 标记整理**(mark-sweep-compact)**：综合了上述两者的做法和优点，先标记活跃对象，然后将其合并成较大的内存块。

可见，没有免费的午餐，无论采用复制还是标记清除算法，自动的东西都要付出很大的性能代价。

3.分代

分代是**Java**垃圾收集的一大亮点，根据对象的生命周期长短，把堆分为**3**个代：**Young**，**Old**和**Permanent**，根据不同代的特点采用不同的收集算法，扬长避短也。

Young(Nursery)，年轻代。研究表明大部分对象都是朝生暮死，随生随灭的。因此所有收集器都为年轻代选择了复制算法。

复制算法优点是只访问活跃对象，缺点是复制成本高。因为年轻代只有少量的对象能熬到垃圾收集，因此只需少量的复制成本。而且复制收集器只访问活跃对象，对那些占了最大比率的死对象视而不见，充分发挥了它遍历空间成本低的优点。

Young的默认值为**4M**，随堆内存增大，约为**1/15**，**JVM**会根据情况动态管理其大小变化。

-XX:NewRatio= 参数可以设置**Young**与**Old**的大小比例，**-server**时默认为**1:2**，但实际上**young**启动时远低于这个比率？如果信不过**JVM**，也可以用**-Xmn**硬性规定其大小，有文档推荐设为**Heap**总大小的**1/4**。

Young的大小非常非常重要，见"后面暂停时间优先收集器"的论述。

Young里面又分为**3**个区域，一个**Eden**，所有新建对象都会存在于该区，两个**Survivor**区，用来实施复制算法。每次复制就是将**Eden**和第一块**Survior**的活对象复制到第**2**块，然后清空**Eden**与第一块**Survior**。**Eden**与**Survivor**的比例由**-XX:SurvivorRatio=**设置，默认为**32**。**Survivio**大了会浪费，小了的话，会使一些年轻对象潜逃到老人区，引起老人区的不安，但这个参数对性能并不重要。

Old(Tenured)，年老代。年轻代的对象如果能够挺过数次收集，就会进入老人区。老人区使用标记整理算法。因为老人区的对象都没那么容易死的，采用复制算法就要反复的复制对象，很不合算，只好采用标记清理算法，但标记清理算法

其实也不轻松，每次都要遍历区域内所有对象，所以还是没有免费的午餐啊。

-XX:MaxTenuringThreshold=设置熬过年轻代多少次收集后移入老人区，**CMS**中默认为**0**，熬过第一次**GC**就转入，可以用**-XX:+PrintTenuringDistribution**查看。

Permanent，持久代。装载**Class**信息等基础数据，默认**64M**，如果是类很多很多的服务程序，需要加大其设置**-XX:MaxPermSize=**，否则它满了之后会引起**fullgc()**或**Out of Memory**。 注意**Spring**，**Hibernate**这类喜欢**AOP**动态生成类的框架需要更多的持久代内存。

4.minor/major collection

每个代满了之后都会促发**collection**，（另外**Concurrent Low Pause Collector**默认在老人区**68%**的时候促发)。**GC**用较高的频率对**young**进行扫描和回收，这种叫做**minor collection**。

而因为成本关系对**Old**的检查回收频率要低很多，同时对**Young**和**Old**的收集称为**major collection**。

System.gc()会引发**major collection**，使用**-XX:+DisableExplicitGC**禁止它，或设为**CMS**并发**-XX:+ExplicitGCInvokesConcurrent**。

5.小结

Young -- minor collection -- 复制算法

Old(Tenured) -- major colletion -- 标记清除/标记整理算法

三、收集器

1.古老的串行收集器**(Serial Collector)**

使用 **-XX:+UseSerialGC**，策略为年轻代串行复制，年老代串行标记整理。

2.吞吐量优先的并行收集器**(Throughput Collector)**

使用 **-XX:+UseParallelGC** ，也是**JDK5 -server**的默认值。策略为：

1.年轻代暂停应用程序，多个垃圾收集线程并行的复制收集，线程数默认为**CPU**个数，**CPU**很多时，可用**–XX:ParallelGCThreads=**减少线程数。

2.年老代暂停应用程序，与串行收集器一样，单垃圾收集线程标记整理。

所以需要**2+**的**CPU**时才会优于串行收集器，适用于后台处理，科学计算。

可以使用**-XX:MaxGCPauseMillis=** 和 **-XX:GCTimeRatio** 来调整**GC**的时间。

3.暂停时间优先的并发收集器**(Concurrent Low Pause Collector-CMS)**

前面说了这么多，都是为了这节做铺垫**……**

使用**-XX:+UseConcMarkSweepGC**，策略为：

1.年轻代同样是暂停应用程序，多个垃圾收集线程并行的复制收集。

2.年老代则只有两次短暂停，其他时间应用程序与收集线程并发的清除。

3.1 年老代详述

并行**(Parallel)**与并发**(Concurrent)**仅一字之差，并行指多条垃圾收集线程并行，并发指用户线程与垃圾收集线程并发，程序在继续运行，而垃圾收集程序运行于另一个个**CPU**上。

并发收集一开始会很短暂的停止一次所有线程来开始初始标记根对象，然后标记线程与应用线程一起并发运行，最后又很短的暂停一次，多线程并行的重新标记之前可能因为并发而漏掉的对象，然后就开始与应用程序并发的清除过程。可见，最长的两个遍历过程都是与应用程序并发执行的，比以前的串行算法改进太多太多了！！！

串行标记清除是等年老代满了再开始收集的，而并发收集因为要与应用程序一起运行，如果满了才收集，应用程序就无内存可用，所以系统默认**68%**满的时候就开始收集。内存已设得较大，吃内存又没有这么快的时候，可以用**-XX:CMSInitiatingOccupancyFraction=**恰当增大该比率。

3.2 年轻代详述

可惜对年轻代的复制收集，依然必须停止所有应用程序线程，原理如此，只能靠多**CPU**，多收集线程并发来提高收集速度，但除非你的**Server**独占整台服务器，否则如果服务器上本身还有很多其他线程时，切换起来速度就**.....** 所以，搞到最后，暂停时间的瓶颈就落在了年轻代的复制算法上。

因此**Young**的大小设置挺重要的，大点就不用频繁**GC**，而且增大**GC**的间隔后，可以让多点对象自己死掉而不用复制了。但**Young**增大时，**GC**造成的停顿时间攀升得非常恐怖，比如在我的机器上，默认**8M**的**Young**，只需要几毫秒的时间，**64M**就升到**90**毫秒，而升到**256M**时，就要到**300**毫秒了，峰值还会攀到恐怖的**800ms**。谁叫复制算法，要等**Young**满了才开始收集，开始收集就要停止所有线程呢。

3.3 持久代

可设置**-XX:+CMSClassUnloadingEnabled-**

XX:+CMSPermGenSweepingEnabled，使**CMS**收集持久代的类，而不是**fullgc**，**netbeans5.5 performance**文档的推荐。

4.增量**(train**算法**)**收集器**(Incremental Collector)**

已停止维护，**–Xincgc**选项默认转为并发收集器。

四、暂停时间显示

加入下列参数 **(**请将**PrintGC**和**Details**中间的空格去掉，**CSDN**很怪的认为是禁止字句）

-verbose:gc -XX:+PrintGC Details -XX:+PrintGCTimeStamps
会程序运行过程中将显示如下输出
 9.211: [GC 9.211: [ParNew: 7994K->0K(8128K), 0.0123935 secs] 427172K->419977K(524224K), 0.0125728 secs]
 显示在程序运行的9.211秒发生了Minor的垃圾收集，前一段数据针对新生区，从7994k整理为0k，新生区总大小为8128k，程序暂停了12ms，而后一段数据针对整个堆。
对于年老代的收集，暂停发生在下面两个阶段，CMS-remark的中断是17毫秒：
[GC [1 CMS-initial-mark: 80168K(196608K)] 81144K(261184K), 0.0059036 secs]
[1 CMS-remark: 80168K(196608K)] 82493K(261184K),0.0168943 secs]
再加两个参数 -XX:+PrintGCApplicationConcurrentTime -XX:+PrintGCApplicationStoppedTime对暂停时间看得更清晰。
五、真正不停的BEA JRockit 与Sun RTS2.0
 Bea的JRockit 5.0 R27 的特色之一是动态决定的垃圾收集策略，用户可以决定自己关心的是吞吐量，暂停时间还是确定的暂停时间，再由JVM在运行时动态决定、改变改变垃圾收集策略。

 它的Deterministic GC的选项是-Xgcprio: deterministic，号称可以把暂停可以控制在10-30毫秒，非常的牛，一句Deterministic道尽了RealTime的真谛。不过细看一下文档，30ms的测试环境是1 GB heap 和 平均 30% 的活跃对象(也就是300M)活动对象，2 个 Xeon 3.6 GHz 4G内存 ，或者是4 个Xeon 2.0 GHz，8G内存。
 最可惜JRockt的license很奇怪，虽然平时使用免费，但这个30ms的选项就需要购买整个Weblogic Real Time Server的license。
 其他免费选项，有：

- -Xgcprio:pausetime -Xpausetarget=210ms

 因为免费，所以最低只能设置到200ms pause target。 200ms是Sun认为Real-Time的分界线。

- -Xgc:gencon

普通的并发做法，效率也不错。
 JavaOne2007上有Sun的 Java Real-Time System 2.0 的介绍，RTS2.0基

于JDK1.5，在Real-Time Garbage Collctor上又有改进，但还在beta版状态，只供给OEM，更怪。

六、JDK 6.0的改进

因为JDK5.0在Young较大时的表现还是不够让人满意，又继续看JDK6.0的改进，结果稍稍失望，不涉及我最头痛的年轻代复制收集改良。

1.年老代的标识-清除收集，并行执行标识

JDK5.0只开了一条收集进程与应用线程并发标识，而6.0可以开多条收集线程来做标识，缩短标识老人区所有活动对象的时间。

2.加大了Young区的默认大小

默认大小从4M加到16M，从堆内存的1/15增加到1/7

3.System.gc()可以与应用程序并发执行

使用-XX:+ExplicitGCInvokesConcurrent 设置

七、小结

1. JDK5.0/6.0

对于服务器应用，我们使用Concurrent Low Pause Collector，对年轻代，暂停时多线程并行复制收集；对年老代，收集器与应用程序并行标记--整理收集，以达到尽量短的垃圾收集时间。

本着没有深刻测试前不要胡乱优化的宗旨，命令行属性只需简单写为：

```
-server -Xms<heapsize></heapsize>M -Xmx<heapsize>
</heapsize>M -XX:+UseConcMarkSweepGC  -XX:+PrintGC Details  -
XX:+PrintGCTimeStamps
```

然后要根据应用的情况，在测试软件辅助可以下看看有没有JVM的默认值和自动管理做的不够的地方可以调整，如-xmn 设Young的大小，-XX:MaxPermSize 设持久代大小等。

2. JRockit 6.0 R27.2

但因为JDK5的测试结果实在不能满意，后来又尝试了JRockit，总体效果要好些。

JRockit的特点是动态垃圾收集器是根据用户关心的特征动态决定收集算法的，参数如下

```
 -Xms<heapsize></heapsize>M -Xmx<heapsize></heapsize>M -
Xgcprio:pausetime -Xpausetarget=200ms -XgcReport -XgcPause -
Xverbose:memory
```