[译]Java8官方GC调优指南 --(八)CMS收集器 - 掘金

juejin.cn/post/6844904053554561037

2020年1月28日

> 本套文章是Java8官方GC调优指南的全文翻译，点击查看原文,原文章名称《Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide》

8 Concurrent Mark Sweep (CMS) Collector CMS收集器

使用参数`-XX:+UseConcMarkSweepGC`来启用CMS收集器。

类似于其他的收集器，CMS收集器也是分代的；minor 和 major都有。CMS收集器尝试通过使用独立的垃圾收集器线程，在应用程序线程执行的同时跟踪可到达的对象，从而减少由于major gc而导致的暂停时间。在每个major gc周期中，CMS收集器在收集开始时暂停所有应用程序线程一小段时间，然后在收集中期再次暂停。第二次停顿往往是两次停顿中较长的停顿。在这两个暂停期间，使用多个线程来执行收集工作。收集的其余部分(包括对活动对象的大部分跟踪和对不可到达对象的清扫)是使用一个或多个收集器线程完成的，此时应用程序自身的线程也在并发执行。minor gc可以与正在进行的major gc循环交叉进行，并以类似于parallel collector的方式进行(特别是，应用程序线程在minor gc期间也会停顿)。

Concurrent Mode Failure 并发模式失败

CMS收集器使用一个或多个垃圾收集器线程，这些线程与应用程序线程同时运行，其目标是在永久代满之前完成对其的收集。如前所述，在正常操作中，CMS收集器在应用程序线程仍然运行的情况下执行大部分跟踪和清除工作，因此应用程序线程只能看到短暂的暂停。然而,如果CMS收集器在tenured区填满之前回收所有不可达对象,或者tenured区的剩余空闲空间已经不足以分配一个新的对象,那么就会产生停顿——所有应用的线程全部停止，直到完成一次Full GC。没办法完成并发收集被称为Concurrent Mode Failure，这表明需要调整CMS收集器参数。如果并发收集被显式垃圾收集(System.gc())或为提供诊断工具提供所需信息，则会报告并发模式中断。

Excessive GC Time and OutOfMemoryError 过长的GC时间和OOM

和平行收集器差不多，不多说

Floating Garbage 漂浮垃圾

CMS收集器与Java HotSpot VM中的所有其他收集器一样，是一种跟踪收集器，它至少标识堆中所有可达对象。用Richard Jones和Rafael D. Lins在他们的书《垃圾收集:自动动态内存算法》中的说法，它是一个增量更新收集器。由于应用程序线程和垃圾收集器线程在major gc期间并发运行，垃圾收集器线程跟踪的对象可能在收集过程结束时就变为不可达的了。这种尚未回收的不可达对象称为Floating Garbage。Floating Garbage的数量取决于并发收集周期的持续时间和应用程序引用更新(也称为突变)的频率。此外，由于young区和

tenured区是独立收集的，一个是另一个的root。一个粗暴的方法是，考虑Floating Garbage对内存的消耗，可以尝试将tenured区的大小增加20%，防止Floating Garbage造成OOM。

一个并发收集周期结束时堆中新产生的Floating Garbage将在下一个收集周期中被清理掉。

Pauses 停顿

CMS收集器在并发收集周期中会造成**两次停顿**。

第一次停顿是将从GC root(例如，来自线程堆栈和寄存器、静态对象等的对象引用)和堆中的其他区域(例如，young区)能直接访问到的对象标记为活动的。

第一次停顿称为initial mark pause 初始化标记停顿。

第二次停顿出现在并发跟踪阶段的结束之后，寻找那些并发跟踪阶段没跟踪到的对象，这种对象一般是由于程序线程引用刚好在并发跟踪后发生了变化。

第二次停顿称为remark pause 再标记停顿。

Concurrent Phases 并发停顿

对可达对象的并发追踪阶段发生在initial mark pause和remark pause之间。在这个并发跟踪阶段，一个或多个并发垃圾收集器线程可能正在使用处理器资源(CPU)，否则应用程序就可以使用这些资源。因此，即使应用程序线程没有暂停，在这个阶段和其他并发阶段，应用程序的吞吐量依然可能会下降(因为CMS要消耗CPU时间片)。在remark pause之后，并发清除阶段CMS将收集到的对象标记为不可达。收集周期完成后，CMS收集器将等待，几乎不消耗任何计算资源，直到下一个major gc开始。

Starting a Concurrent Collection Cycle 启动一次并发收集周期

对于串行收集器，只要tenured区已满，就会触发major gc，并且在收集完成之前停止所有应用程序线程。相反，并发收集都是定时开始的，所以收集可以在tenured区满之前结束;否则，应用程序由于Concurrent Mode Failure触发Full GC产生更长的暂停。有几种方法可以启动并发收集。

根据最近的历史记录，CMS收集器维护了一个预估的tenured耗尽的剩余时间和并发收集周期所需的时间。基于这些动态的估算，就会启动一次并发收集周期，目的是在tenured区耗尽之前完成这次收集循环。这些预估是为了安全而进行的，因为并发模式失败的代价可能非常大。

如果tenured区的占用超过初始占用(一个tenured区的百分比)，也会启动并发收集。该初始占用阈值的默认值约为92%，但是该值可能随版本的不同而变化。这个值可以使用参数调整：`-XX:CMSInitiatingOccupancyFraction=<N>`，其中是tenured区大小的整数百分比(0到100)。

Scheduling Pauses 定期停顿

young区收集和tenured区收集的停顿是独立的。它们不重叠，但可以快速地连续发生，一次收集的停顿，紧接着另一次收集的暂停，看起来像是一次更长的停顿。为了避免这种情况，CMS收集器尝试将remark pause安排在两次young区停顿之间的中间。这种调度目前还不支持 initial mark pause，它通常比remark pause短得多。

Incremental Mode 增量模式

请注意，Incremental Mode在Java SE 8中被弃用，可能在将来的主要版本中被删除。

CMS收集器可以在以递增的方式完成并发阶段。回想一下，在并发阶段，垃圾收集器线程使用一个或多个处理器。Incremental Mode的目的是通过周期性地yield并发阶段线程从而将处理器交还给应用程序，来减少长并发阶段的影响。这种模式在这里称为i-cms，它将收集器并发完成的工作划分为小块时间，并且分布在多次young gc之间。当应用程序部署在CPU核数低的机器上(小于等于双核)，并且需要低停顿时间时，此特性会比较有用。

并发收集周期通常包括以下步骤：

- 停止所有的应用线程，从gc root识别可达对象，然后恢复所有应用线程。
- 并发追踪可达对象图，使用1或多个处理器，同时应用线程也在正常运行。
- 并发重新追踪那些上一阶段之后被修改的对象，使用1个处理器
- 停止所有线程，重新追踪gc root和对象图，防止上一阶段之后对象引用又出现变更，然后恢复所有应用线程。
- 并发清除不可达对象，使用单处理器。
- 并发resize整个heap，为下次收集循环准备好数据，使用单处理器。

通常，CMS收集器在整个并发跟踪阶段使用一个或多个处理器，而不会主动放弃它们。类似地，在整个并发清除阶段使用一个处理器，同样不放弃它。这些开销对于一个有响应时间限制的应用来说，可能太大了，尤其是那种运行在单核或者双核的物理机服务。Incremental Mode通过将并发阶段拆解成分布在minor gc之间的多个部分来解决这个问题。(通过Thread.yield方法交还CPU让应用程序线程获得执行机会)

i-cms模式在CMS放弃处理器资源之前使用duty cycle来控制并发阶段的工作量。duty cycle是两次minor gc的之间允许CMS回收器工作的时间百分比。i-cms模式可以根据应用程序的行为(推荐的方法，称为自动步调)自动计算duty cycle，duty cycle也可以通过参数来设置。

Command-Line Options 命令行选项

下表是ims的命令行参数：

Option	Description	Java5之前的默认值	Java6之后的默认值
-XX:+CMSIncrementalMode	开启增量模式,同时也会开启CMS	禁用	禁用

Option	Description	Java5之前的默认值	Java6之后的默认值
-XX:CMSIncrementalPacing	开启自动步长，duty cycle会自动调整	禁用	禁用
-XX:CMSIncrementalDutyCycle=	minor gc之间cms的执行时间百分比，如果设置了pacing,那么这个参数就是初始值	50	10
-XX:CMSIncrementalDutyCycleMin=	开启pacing后duty cycle的下界	10	0
-XX:CMSIncrementalSafetyFactor=	计算duty cycle时稳定增加的百分比	10	10
-XX:CMSIncrementalOffset=	duty cycle右移的百分比	0	0
-XX:CMSExpAvgFactor=	对当前样本数量加权的百分比	25	25

Recommended Options

Java 8 要使用i-cms，使用下面的命令行参数:

```
-XX:+UseConcMarkSweepGC -XX:+CMSIncrementalMode \
-XX:+PrintGCDetails -XX:+PrintGCTimeStamps
复制代码
```

后面两个参数就是打日志用的，可以后期分析GC行为。

Basic Troubleshooting 基本的问题定位

i-cms自动步长特性在程序运行期间收集各项指标来计算duty cycle，这样并发收集可以在heap被填满之前完成。但是，通过过去的行为也不能一直预测外来的行为，这个预估可能也不会那么精确。如果发生了过多的GC，可以根据下表来进行调优:

Step	Options
增加安全系数	-XX:CMSIncrementalSafetyFactor=
增加最小duty cycle	-XX:CMSIncrementalDutyCycleMin=
关闭自动步长，使用固定的duty cycle	-XX:-CMSIncrementalPacing -XX:CMSIncrementalDutyCycle=

Measurements 度量

下面的日志片段是CMS收集器增加参数`-verbose:gc`和`-XX:+PrintGCDetails`后输出的。

注意，CMS收集器的输出与minor gc的输出穿插在一起;通常，许多minor gc发生在并发收集周期中。CMS-initial-mark表示并发收集周期的开始，CMS-concurrent-mark表示并发标记阶段的结束，CMS-concurrent-sweep表示并发清除阶段的结束。之前没有讨论过CMS-concurrent-preclean代表的的预清理阶段。preclean阶段和remark阶段是并发进行的。最后一个阶段是CMS-concurrent-reset，表示正在为下一次并发收集做准备。

```
[GC [1 CMS-initial-mark: 13991K(20288K)] 14103K(22400K), 0.0023781 secs]
[GC [DefNew: 2112K->64K(2112K), 0.0837052 secs] 16103K->15476K(22400K), 0.0838519
secs]
...
[GC [DefNew: 2077K->63K(2112K), 0.0126205 secs] 17552K->15855K(22400K), 0.0127482
secs]
[CMS-concurrent-mark: 0.267/0.374 secs]
[GC [DefNew: 2111K->64K(2112K), 0.0190851 secs] 17903K->16154K(22400K), 0.0191903
secs]
[CMS-concurrent-preclean: 0.044/0.064 secs]
[GC [1 CMS-remark: 16090K(20288K)] 17242K(22400K), 0.0210460 secs]
[GC [DefNew: 2112K->63K(2112K), 0.0716116 secs] 18177K->17382K(22400K), 0.0718204
secs]
[GC [DefNew: 2111K->63K(2112K), 0.0830392 secs] 19363K->18757K(22400K), 0.0832943
secs]
...
[GC [DefNew: 2111K->0K(2112K), 0.0035190 secs] 17527K->15479K(22400K), 0.0036052
secs]
[CMS-concurrent-sweep: 0.291/0.662 secs]
[GC [DefNew: 2048K->0K(2112K), 0.0013347 secs] 17527K->15479K(27912K), 0.0014231
secs]
[CMS-concurrent-reset: 0.016/0.016 secs]
[GC [DefNew: 2048K->1K(2112K), 0.0013936 secs] 17527K->15479K(27912K), 0.0014814
secs]
```

复制代码

相对于minor gc的停顿时间，initial mark停顿时间通常较短。并发阶段(并发标记、并发预清理和并发清理)通常持续的时间明显长于minor gc的停顿时间，如上面的日志所示。但是，请注意，应用程序在这些并发阶段不会停顿，虽然持续时间长，但是注意程序不会停顿。remark造成的停顿时间通常与minor gc停顿时间相当。remark停顿受某些应用程序特征(例如，高频率修改对象引用可能会增加此停顿的持续时间)和上一次minor gc的持续时间(例如，young区的对象越多，停顿时间越长)的影响。