**GC 优化的一些总结**

JUN 25TH, 2015 | [COMMENTS](http://engineering.xueqiu.com/blog/2015/06/25/jvm-gc-tuning/#disqus_thread)

**合理堆大小的设置**

基本上所有会写Java的人也都知道怎么设置堆相关的参数，会设置但并不意味着这个是一个简单事情。常见的OOM问题，大部分都是因为不恰当的堆设置引起的。

**该给Java进程配置多大的堆？**

换个方式问这个问题： **大堆、小堆对Java进程会有怎样的影响？**

* 对Java程序而言，堆大小最大的影响的在于垃圾回收STW的时间。因此，对在意吞吐量而不在意响应时间的应用，堆可以设置的很大。
* 而对于响应时间要求比较高的应用，堆大小则需要仔细考虑。设小了，会导致频繁gc，gc效率太低；设大了，则STW的时间太长，无法满足时延要求。

**响应时间优先的应用如何确定堆大小？**

为方便讨论，做以下假设：

* 假设响应时间要求99%小于100ms
* 假设应用qps为1000
* 平均响应时间为50ms
* 忽略GC外其他方面对时延造成的影响
* 假设应用是典型的无状态应用，进程内长生命周期对象主要为连接池、线程池、缓存等

由以上假设，可以做一点简单的计算:

* 99%的请求100ms以内、每秒1000个请求、平均响应时间50ms，意味着5s内可以留给gc做暂停的时间，最长为100ms。
* 100ms的暂停时间中，其中有50个请求受gc影响，暂停时间大于50ms，加上业务自身的50ms，超过了100ms；另外50ms，暂停时间小于50ms，因此没有超过100ms. 5s内5000个请求，其中50个超时，刚好99%在100ms以内

通过上面的计算，我们得出一个结论:

在以上假设的条件下，**5s内超过可以留给gc做暂停的时间，最长为100ms**

这个结论变换为以下说法: **5s内发发生一次young gc，最长时间为100ms**, 同时，要求old区gc最长不超过100ms

继续:

* 有了gc频率和gc最长停止时间，我们就有了推算堆大小的方法
* 如果young区设置为4G时，刚好能保证5s一次young gc，平均100ms，那么4G的young区是一个能够满足业务目标的young区堆大小设置。

**如何确定old区大小?**

* 如果使用并行收集器，那么full gc的时候多半会超过young gc时的最长时间，因为大家算法差不多，但是full gc的时候堆变大了。因此，在响应时间优先的业务中，并行收集器不被推荐，一次full gc就有可能影响整个业务。
* 如果使用并发收集器(CMS)，CMS在old gc时有两次短暂的暂停（initial阶段和remark阶段），能保证这两次暂停小于100ms的整个堆大小设置，是一个可以作为参考值的old区大小设置。

**young区大小小于前面4G的合理值会不会更好呢？至少显而易见的是超时率更小了**

* 堆大小会直接影响的一个指标是gc效率，即: gc时间占整个系统运行时间的比例。一般来说堆越大效率会越高，堆越小效率会越底。因此，吞吐量优先的应用，尽可把堆设置的大。
* gc效率低，说明应用很大部分的cpu时间和等待时间都在gc上，这种开销规模小的时候不明显，但是规模大了以后，需要尽可能减少。

**并发收集器(CMS)的问题**

通过前面部分可以看到，响应时间优先的应用，并发收集器（CMS）是一个好的选择，但是CMS也有自己的一些问题。这些问题会导致并发收集器长时间暂停，从而影响业务。

相对并行收集器，并发收集器设计为在进行垃圾回收时不会暂停应用，因此并发收集器存在几个根本的问题:

* 不能暂停进行垃圾回收，意味着进行并发垃圾回收时，需要预留空间，保证垃圾回收进行时，有足够的空间能够装下回收过程中的新增对象
* 不能暂停，同时CMS本身内存结构的限制，会导致内存碎片。碎片意味着当有大对象出现时，需要进行空间整理。

**Concurrent Mode Failure**

常见导致这个问题有两种情况：一种确实是内存泄露了，导致无法垃圾回收。这时会看到连续的Concurrent Mode Failure。另一种是预留空间不足，我们主要讨论后一种。

这个错误在CMS的gc日志中有可能会看到，这个问题对应开头第一点，即: 在进行old区垃圾回收的时候，old区空间没了，此时，Java进程会暂停应用，进行full gc

一般来说，增加预留空间就可以解决这个问题，可以选择：

* 预留空间比例不变的情况下，增加old区堆大小。
* 堆大小不变的情况下，增加预留比例。即: 缩小-XX:CMSInitiatingOccupancyFraction=70。比如：整个堆4G，young区2G，因此old区有2G。如果设置 -XX:CMSInitiatingOccupancyFraction=70，即表示，old区使用2Gx0.7=1.4G时，触发old区gc，预留空间为600m。如果缩小这个值到-XX:CMSInitiatingOccupancyFraction=50，表示old区使用2Gx0.5=1G时，触发old区gc，预留空间为1g

**Promotion Failed**

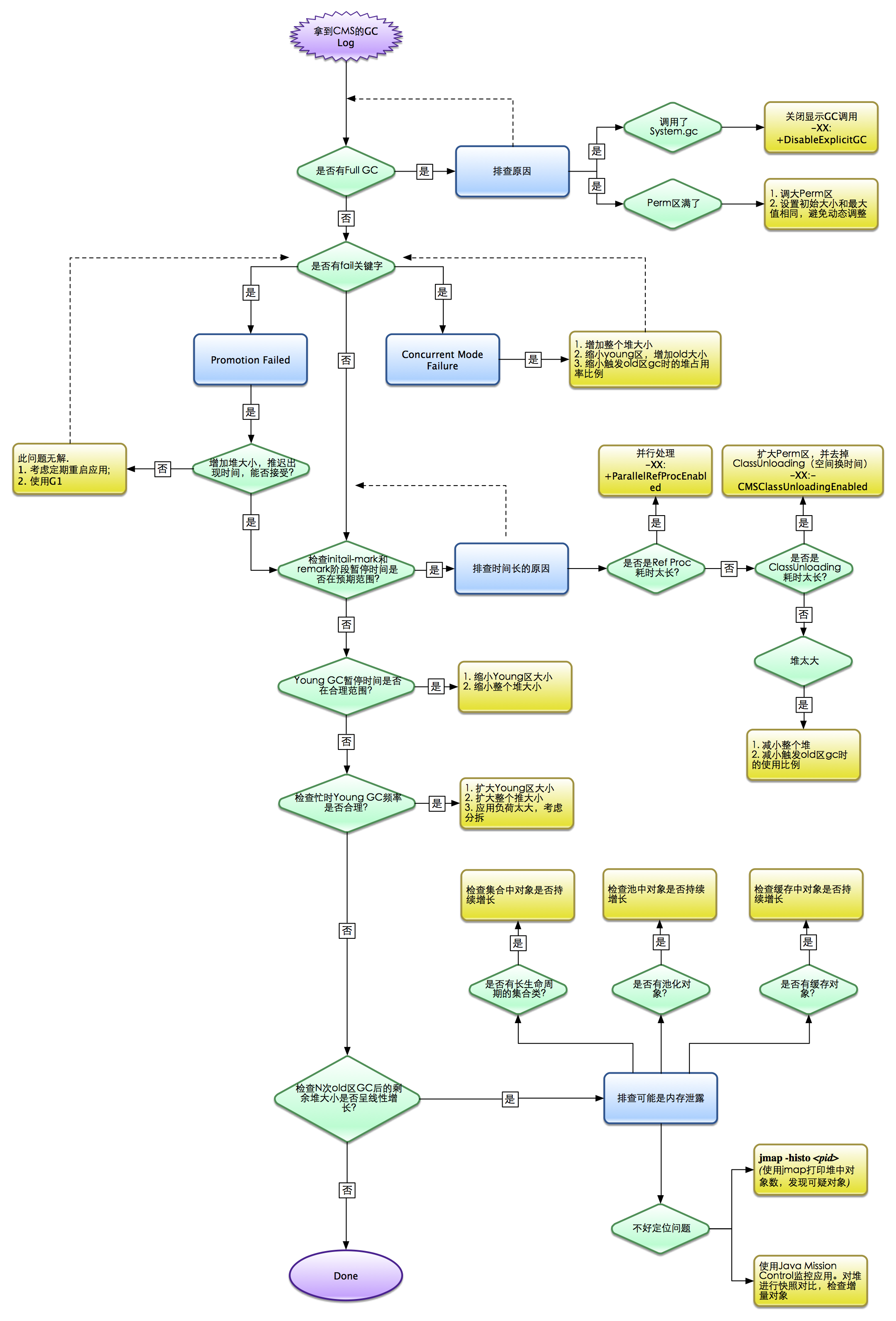
这个问题对应最开始的第二点，碎片问题。

* 这个问题在young gc时有大对象需要迁移到old区，但是old区没有连续空间，随后会暂停应用，进行空间整理。
* 因为是由于碎片造成，因此，这个问题出现会比较随机。与业务相关，与业务产生的对象相关。
* 这个问题基本无解，因为，不进行full gc或者空间整理，那么碎片只会越来越多。
* 可以考虑增加堆大小，这样能减缓碎片产生，或者在流量小的时候，主动触发一次full gc，减少影响。

碎片问题算是CMS的硬伤，也才有了G1。

以上两个问题，均与业务产生的对象有较大关系，有些业务再大的量也不见得会有上面问题，而有的业务，在很小量的时候就会出现。

**CMS参数优化的大致流程**



**关于G1**

G1也是并发收集器，G1解决了CMS碎片的硬伤，同时增加了更多响应时间优先的特性。

* G1之所以能解决CMS存在的问题，是因为对内存结构进行了彻底的改造。把堆进行了分块处理，可以按照内存块为单位计算其使用率等。
* G1也使用分代的垃圾回收方式，这点与以前是一样的.
* G1的优化很多是围绕内存块的优化

**G1的垃圾回收log**

关于G1的日志可以看Oracle的这篇[g1gc logs - basic - how to print and how to understand](https://blogs.oracle.com/g1gc/entry/g1gc_logs_how_to_print)，写的很详细了。

其中需要注意的是，G1也是并发的时候，与CMS类似分了一些阶段。其中G1收集器在remark阶段和cleanup阶段STW，而CMS在initial阶段和remark阶段STW。

**G1涉及到的一些优化参数**

* -XX:G1HeapRegionSize=4m: 设置内存分块的大小。范围是1MB~32MB。是一个很常用的参数。当系统中存在大量大对象的时候，大的Region会提升GC效率。
* XX:InitiatingHeapOccupancyPercent=50: 设置使用整个对的x%时，系统开始进行并行GC。注意是**整个堆**的百分比。这与CMS收集器的类似参数不同。
* -XX:MaxGCPauseMillis=200: 单位为毫秒。此值为建议JVM的最长暂停时间。只是建议值，G1只能尽量保证，而无法完全保证。
* -XX:ParallelGCThreads: 进行young区和old区垃圾回收时的并发线程数。默认线程数由CPU数量决定（通常小于CPU个数）。当有较大的Update RSet时间时，可以尝试调整此值。
* -XX:ConcGCThreds: 并发GC时的线程数。mixed GC情况下，较长的cycle start时间，可以尝试调整此值。
* -XX:+ParallelRefProcEnabled: 当看到有较长的Ref Proc建议配置此值。之前我们从JDK6升级到JDK8时，发现Ref Proc时间有较长的提升，CMS收集器和G1收集器均由这个问题。配置以后暂停明显缩小。

对响应时间优先的应用，G1从目前的使用看情况是比较理想的。即没有CMS存在的硬伤，在优化层面也相对简单。从G1内存结构看，G1实际使用时可以配置比较大的堆。可以预见，G1将会逐步替换目前CMS收集器。

**参考文献**

* [JVM相关参数](http://docs.oracle.com/javase/8/docs/technotes/tools/unix/java.html)
* [G1的log解读](https://blogs.oracle.com/g1gc/entry/g1gc_logs_how_to_print)
* [Garbage First Garbage Collector Tuning](http://www.oracle.com/technetwork/articles/java/g1gc-1984535.html)