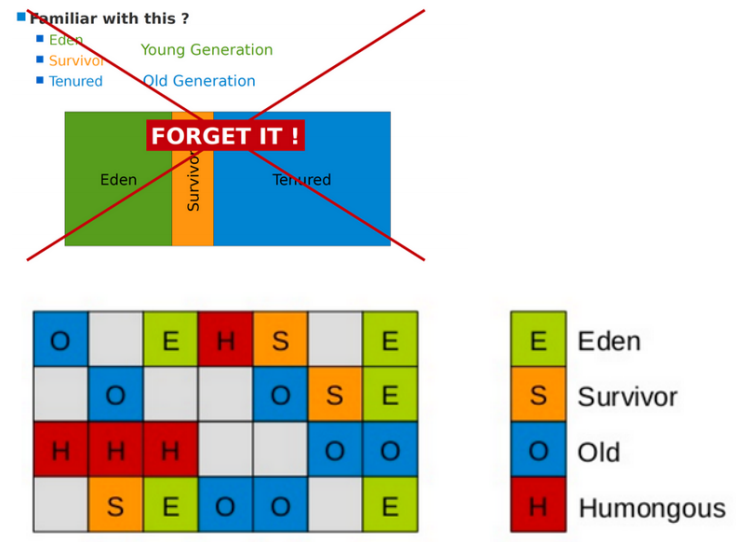
### 垃圾收集器

#### G1收集器

G1 (Garbage-First)是一款面向服务器的垃圾收集器,主要针对配备多颗处理器及大容量内存(堆内存8G以上)的机器. 以极高概率满足GC停顿时间要求的同时,还具备高吞吐量性能特征。



G1将java堆分为N个大小相等的独立区域(Region),jvm最多可以分配2048块region。

Jvm默认讲堆分配成2048块，例如4096M的堆内存，单个region就是2m。可以通过**XX:G1HeapRegionSize**手动指定Region大小，但是推荐默认的计算方式。

1. **G1的分代**

G1保留年轻代和老年代，但是不再物理隔开，他们都是region。而且region可以不停复用，垃圾收集之前是年轻代，收集之后可能会分配成老年代。

* 年轻代

默认占堆内存的5%。例如堆内存是4096m，年轻代就是200m左右，大概100块region。可以通过**“-XX:G1NewSizePercent”**设置新生代占比。其实不设置jvm也会自动给它提高，默认最高是60%。最高可以通过**“-XX:G1MaxNewSizePercent”**调整。

年轻代的edan和survivor比例默认也是8:1:1，例如100个region就有80个eden，10个s0,10个s1。

* 老年代

跟之前收集器的一样，年轻代15次gc以后加入老年代。

* Humongous

针对大对象处理跟之前收集器不一样，之前的是加入老年代，G1新增了humongous区，专门存放大对象。例如单块region是2m，大对象是6m则分配3块连续的region给此大对象。

怎么定义大对象?默认超过50%region容量的对象就是大对象。Humongous区适用于存放短期且巨型的对象，避免老年代空间不足导致full gc。

G1的Full gc会清空年轻代、老年代、humongous区的垃圾。

1. **G1的收集过程(跟cms类似)**

* **初始标记（initial mark，STW）**

暂停其他用户线程，然后记录gc root标记的所有对象，速度很快，类似STW。

* **并发标记（Concurrent Marking）**

同cms，用户线程、标记线程同时运行，由于是并发执行，会导致已标记对象但又发生了变化。

* **最终标记（Remark，STW）**

同cms的重新标记。补救2的，会stw，时间比1稍长，但是远低于2。主要用到**三色标记**里的增量更新算法做重新标记。

* **筛选回收（Cleanup，STW）**

筛选回收阶段会对各个region的回收成本进行计算、排序，然后根据用户希望的GC停顿时间来进行回收。GC停顿时间可以通过-XX:MaxGCPauseMillis指定，默认200ms。

例如有1000个region都满了，而G1需要在200ms内完成gc，则G1会根据之前整理的成本来挑选region执行gc。有可能仅仅清空了800 geregion，一切为了满足200ms而执行。

筛选回收会STW,而且是用户线程都要暂停。

1. **收集算法与回收原则**

回收算法主要是复制算法，将一个region中存活对象复制到另一个region，执行回收。跟cms对比，cms标记清理会导致内存碎片化，还要另外开销来整理内存。G1几乎不会有太多碎片内存。

(注意：CMS回收阶段是跟用户线程一起并发执行的，G1因为内部实现太复杂暂时没实现并发回收，不过到了Shenandoah就实现了并发收集，Shenandoah可以看成是G1的升级版本)

G1的回收原则是优先回收价值大的region，例如回收一个10m垃圾的region需花200ms，跟回收一个20m垃圾的region需花50ms比，G1优先回收后者。

**通过region分区、回收成本队列的机制，保证G1在优先时间内高效完成收集。**

1. **G1优点**

* **并行与并发：**G1能充分利用CPU、多核环境下的硬件优势，使用多个CPU（CPU或者CPU核心）来缩短StopThe-World停顿时间。部分其他收集器原本需要停顿Java线程来执行GC动作，G1收集器仍然可以通过并发的方式让java程序继续执行。
* **分代收集：**虽然G1可以不需要其他收集器配合就能独立管理整个GC堆，但是还是保留了分代的概念。
* **空间整合：**与CMS的“标记--清理”算法不同，G1从整体来看是基于“标记整理”算法实现的收集器；从局部上来看是基于“复制”算法实现的。
* **可预测的停顿：**这是G1相对于CMS的另一个大优势，降低停顿时间是G1 和 CMS 共同的关注点，但G1 除了追求低停顿外，还能建立可预测的停顿时间模型，能让使用者明确指定在一个长度为M毫秒的时间片段(通过参数"**-XX:MaxGCPauseMillis**"指定)内完成垃圾收集。
* **总结**

毫无疑问,可以由用户指定期望的停顿时间是G1收集器很强大的一个功能,设置不同的期望停顿时间,可使得G1在不同应用场景中取得关注吞吐量和关注延迟之间的最佳平衡。 不过,这里设置的“期望值”必须是符合实际的,不能异想天开,毕竟G1是要冻结用户线程来复制对象的,这个停顿时间再怎么低也得有个限度。

它默认的停顿目标为200ms。一般来说,回收阶段占到几十到一百甚至接近两百毫秒都很正常,但如果我们把停顿时间调得非常低,譬如设置为二十毫秒,很可能出现的结果就是由于停顿目标时间太短,导致每次选出来的回收集只占堆内存很小的一部分,收集器收集的速度逐渐跟不上分配器分配的速度,导致垃圾慢慢堆积。 很可能一开始收集器还能从空闲的堆内存中获得一些喘息的时间,但应用运行时间一长就不行了,最终占满堆引发Full GC反而降低性能。

所以通常把期望停顿时间设置为一两百毫秒或者两三百毫秒会是比较合理的。

1. **G1垃圾收集分类**

* **YoungGC**

YoungGC并不会跟之前的收集器一起等eden区满了才触发，而是根据-XX:MaxGCPauseMillis的设定的值来估算当前年轻代gc耗时是否约大于等于MaxGCPauseMillis。是的话触发YoungGC，否则增加年轻代的region，继续存放对象，直到可以YoungGC为止。

* **Mixed GC**

不是fullGC，G1独有，是根据老年代的堆占有率达到参数(-XX:InitiatingHeapOccupancyPercent)设定值则触发。回收所有的young、部分old(根据期待的gc停顿时间来确定old区垃圾收集的优先顺序)、humongous区。

一般G1先做MixedGC，MixedGC主要用复制算法，把各个区的region存活对象复制到别的region，如果不够region则触发full gc。

* **Full GC**

会触发STW,然后使用单线程标记、清理、压缩整理，以便空下region供下一次MixedGC使用，这个过程相当耗时。(Shenandoah优化成多线程收集了)。

1. **G1收集器参数设置**

-XX:+UseG1GC:使用G1收集器

-XX:ParallelGCThreads:指定GC工作的线程数量

-XX:G1HeapRegionSize:指定分区大小(1MB~32MB，且必须是2的N次幂)，默认将整堆划分为2048个分区

-XX:MaxGCPauseMillis:目标暂停时间(默认200ms)

-XX:G1NewSizePercent:新生代内存初始空间(默认整堆5%)

-XX:G1MaxNewSizePercent:新生代内存最大空间，默认最高60%

-XX:TargetSurvivorRatio:Survivor区的填充容量(默认50%)，Survivor区域里的一批对象(年龄1+年龄2+年龄n的多个年龄对象)总和超过了Survivor区域的50%，此时就会把年龄n(含)以上的对象都放入老年代

-XX:MaxTenuringThreshold:最大年龄阈值(默认15)

-XX:InitiatingHeapOccupancyPercent:老年代占用空间达到整堆内存阈值(默认45%)，则执行新生代和老年代的混合收集(MixedGC)，比如我们之前说的堆默认有2048个region，如果有接近1000个region都是老年代的region，则可能就要触发MixedGC了

-XX:G1MixedGCLiveThresholdPercent(默认85%) region中的存活对象低于这个值时才会回收该region，如果超过这个值，存活对象过多，回收的的意义不大。

-XX:G1MixedGCCountTarget:在一次回收过程中指定做几次筛选回收(默认8次)，在最后一个筛选回收阶段可以回收一会，然后暂停回收，恢复系统运行，一会再开始回收，这样可以让系统不至于单次停顿时间过长。

-XX:G1HeapWastePercent(默认5%): gc过程中空出来的region是否充足阈值，在混合回收的时候，对Region回收都是基于复制算法进行的，都是把要回收的Region里的存活对象放入其他Region，然后这个Region中的垃圾对象全部清理掉，这样的话在回收过程就会不断空出来新的Region，一旦空闲出来的Region数量达到了堆内存的5%，此时就会立即停止混合回收，意味着本次混合回收就结束了。

1. **G1垃圾收集器优化建议**

假设参数 -XX:MaxGCPauseMills 设置的值很大，导致系统运行很久，年轻代可能都占用了堆内存的60%了，此时才触发年轻代gc。

那么存活下来的对象可能就会很多，此时就会导致Survivor区域放不下那么多的对象，就会进入老年代中。

或者是你年轻代gc过后，存活下来的对象过多，导致进入Survivor区域后触发了动态年龄判定规则，达到了Survivor区域的50%，也会快速导致一些对象进入老年代中。

所以这里核心还是在于调节 -XX:MaxGCPauseMills 这个参数的值，在保证他的年轻代gc别太频繁的同时，还得考虑每次gc过后的存活对象有多少,避免存活对象太多快速进入老年代，频繁触发mixed gc。

1. **什么场景适合使用G1**

* 50%以上的堆被存活对象占用
* 对象分配和晋升的速度变化非常大
* 垃圾回收时间特别长，超过1秒
* **8GB以上的堆内存(建议值)**
* 停顿时间是500ms以内

1. **G1实例**

Kafka类似的支撑高并发消息系统大家肯定不陌生，对于kafka来说，每秒处理几万甚至几十万消息时很正常的，一般来说部署kafka需要用大内存机器(比如64G，最低32g)，也就是说可以给年轻代分配个三四十G的内存用来支撑高并发处理。

这里就涉及到一个问题了，我们以前常说的对于eden区的young gc是很快的，这种情况下它的执行还会很快吗？很显然，不可能，因为内存太大，处理还是要花不少时间的，假设三四十G内存回收可能最快也要几秒钟，按kafka这个并发量放满三四十G的eden区可能也就一两分钟吧，那么意味着整个系统每运行一两分钟就会因为young gc卡顿几秒钟没法处理新消息，显然是不行的。

那么对于这种情况如何优化了，我们可以使用G1收集器，设置 -XX:MaxGCPauseMills 为50ms，假设50ms能够回收三到四个G内存，然后50ms的卡顿其实完全能够接受，用户几乎无感知，那么整个系统就可以在卡顿几乎无感知的情况下一边处理业务一边收集垃圾。

G1天生就适合这种大内存机器的JVM运行，可以比较完美的解决大内存垃圾回收时间过长的问题。