链滴

JVM GC 算法和垃圾收集器

作者：helly

原文链接：https://ld246.com/article/1499506308788

来源网站：链滴

<h2 id="一-对象存活判断">一、对象存活判断</h2>
<h3 id="1--引用计数法">1. 引用计数法</h3>
给每个对象加一个引用计数器，记录指向这个对象的引用数量。当计数器值为 0 时回收该对象。
/p>
缺点：无法解决循环引用问题。
<h3 id="2--可达性分析">2. 可达性分析</h3>
直接或间接被 GC Roots 引用着的对象是存活对象。GC Roots 是指活跃的引用，当然，它指向
就是活跃的对象了。
JVM 中 GC Roots 指向的对象
①. 虚拟机栈中的对象；
②. 方法区中类静态属性的对象；
③. 方法区中常量的对象；
④. 本地方法栈中 Native 方法（JNI）中的对象。
⑤. 分代式 GC 是一种部分收集（partial collection）的做法。在执行部分收集时，从 GC 堆的非收
部分指向收集部分的引用，也必须作为 GC roots 的一部分。
<h2 id="二-垃圾收集算法">二、垃圾收集算法</h2>
<h3 id="1--标记-清除算法">1. 标记-清除算法</h3>
标记阶段：从 GC Roots 开始遍历，可以访问到的对象就标记为可达对象

清除阶段：对堆中的对象从头到尾遍历以便，没有被标记为可达的对象就被回
。
缺点：产生太多内存碎片。
<h3 id="2--复制算法">2. 复制算法</h3>
每次将一个半区中存活的对象拷贝到另一个半区中。
缺点：可用内存减少一半。
<h3 id="3--标记-压缩算法-又叫标记-整理算法-">3. 标记-压缩算法（又叫标记-整理算法）</h3>
标记阶段：从 GC Roots 开始遍历，可以访问到的对象就标记为可达对象

压缩阶段：将标记为可达的对象拷贝到一端连续的内存区域中。对象拷贝之后
顺序有 3 种情况：
①. 任意顺序排列；
②. 按原来顺序排列；
③. 线性排序，即存在引用关系的对象尽可能放到一起。
<h3 id="4--分代收集算法">4. 分代收集算法</h3>
新生代 - 复制算法
依据：经验告诉我们，大部分新创建的对象的生命周期都非常短。
老年代 - 标记-清除算法/标记-压缩算法
<h2 id="三-垃圾收集器">三、垃圾收集器</h2>
<h3 id="G1-GC-Garbage-First-Garbage-Collector-">G1 GC（Garbage-First Garbage Collecto
）</h3>
G1 垃圾收集器的设计目标是取代 CMS 垃圾收集器，它在 JDK6 中作为体验版面世，JDK7 正式
出，JDK9 设置为默认垃圾回收器。
G1 GC 优点：高吞吐、没有内存碎片、收集时间可控。很适合多处理器和
容量内存的服务端环境中。
G1 在堆上分配一系列相同大小的连续区域，然后在回收时先扫描所有的区域，按照每块区域内
活对象的大小进行排序，优先处理存活对象小的区域，即垃圾对象最多的区域，因为这样需要复制的
跃对象就少了，这也是 Garbage First 这个名称的由来。
G1 将内存区域分为新生代、老年代、humongous 区，其中新生代包括 Eden 区和 Survivor 区

Region：G1 的各代存储不是连续的，而是用大小相同的 Region 块来组
。Region 块的大小可以通过参数设定。
Humongous Object（H-obj）：巨型对象，是指大小超过一个 Region 块
50% 的 Java 对象，G1 用一个或者多个 Region 块来存储巨型对象，这些块组成了 Humongous 区

Mutator：指应用线程。
Collector：垃圾收集线程。
Stop-The-Wrold（STW）：指垃圾收集线程运行时其他所有应用线程都能执行，就好像世界突然停止了。
G1 GC 提供的两种垃圾回收模式
1. Young GC
触发时机：当所有 Eden 区无法再容纳新对象时，触发一次 Young GC。
过程：将整个新生代的可达对象都复制到 Survivor 区，如果一些可达对象为 Survivor 区已满无法进入或者对象的分代年龄达到设定的值（默认是 15），那么这部分对象会进老年代。
回收之后空闲的 region 会被放入空闲列表 Free List 中。
2. Mixed GC
触发时机：当老年代大小占整个堆大小百分比达到设定的阈值（InitiatingHapOccupancyPercent）时，会触发一次 Mixed GC。
Mixed GC 选定所有年轻代里的 Region，外加根据 global concurrent marking 统计得出收集益高的若干老年代 Region 进行回收。
marking bitmap： 记录所有可达对象的表（可以理解为就是快照）。</p

全局并发标记（global concurrent marking）
1. Initial Mark（STW）：扫描 GC Roots，将 GC Roots 指向的对象标记可达，并记录到 marking bitmap 中，然后将这些对象的引用字段压入扫描栈（Marking Stack）。
在分代式 G1 模式中，初始标记阶段借用 young GC 的暂停，因而没有额外的、单独的暂停阶。
2. Concurrent Mark：从扫描栈中取出引用，然后将该引用指向的对象进标记，并将该对象的引用字段压入扫描栈，这样不断递归地标记、压栈，直到扫描栈为空，说明 GC Rots 直接或间接引用的对象都被标记了。
三色标记算法：
①. 灰色：表示垃圾收集器已经访问过该对象，但是还没有访问过它的所有孩子点。
②. 黑色：表示该对象以及它的所有孩子节点都已经被垃圾收集器访问过了；<b>
③. 白色：表示该对象从来没有被垃圾收集器访问过，这就是非可达对象。
因为并发标记阶段是 GC 线程和应用线程并发执行的所以应用线程可能修改了对象的引用关系，造成对象的误标、漏标。
误标：应用线程修改了一个引用，比如为 null，造成了这个引用之前指向对象变成不可达的，但这个对象之前已经被标记为可达对象，从而造成了误标。误标没什么大危害，多造成浮动垃圾，等待下次垃圾回收就好了。
漏标的两种情况：

应用线程创建了一个白对象，然后让黑色对象的引用指向该白色对象；

一个白色对象本来有灰色对象和黑色对象的引用指向，但应用线程删除所有灰色对象到该白色对的引用。

如何解决漏标问题？SATB write barrier。
SATB（Snapshot At The Beginning）：简单地说就是 initial mark 阶段 concurrent mark 阶段标记为活的的对象就是活的。然后 concurrent mark 阶段新增或者引用重新行的对象也认为是活的。其他的就是死的。

SATB 算法具体如何实现？
TAMS（Top At Mark Start）
每个 Region 有 5 个指针：
bottom、previous TAMS、next TAMS、top、end。
previous TAMS、next TAMS 是前后两次发生并发标记时的位置。
并发标记开始，将该 Region 当前的 top 指针赋值给 next TAMS，在并发标记标记期间，分配对象都在[next TAMS, top]之间，SATB 能够确保这部分的对象都会被标记，默认都是存活的.
当并发标记结束时，将 next TAMS 所在的地址赋值给 previous TAMS，SATB 给 [bottom, previous TAMS] 之间的对象创建一个快照，所有垃圾对象能通过快照被识别出来。
两种 SATB write barrier:
post-write-barrier：用 post-write-barrier 记录新增的引用关系，然后在 emark 阶段根据这些新增引用关系为根重新扫描一遍。
pre-write-barrier：用 pre-write-barrier 将所有即将被删除的引用关系的引用记录下来，最后以这些旧引用为根重新扫描一遍（把原来有黑色对象和灰色对象引用着，但后来删除灰色对象引用的白色对象扫描加进 marking bitmap）。
logging-write-barrier：
为了尽量减少 write barrier 对 mutator 性能的影响，G1 将一部分原本要在 barrier 里做的事情挪到的线程上并发执行。
实现这种分离的方式就是通过 logging 形式的 write barrier：mutator 只在 barrier 里把要做的事情信息记（log）到一个队列里，然后另外的线程从队列里取出信息批量完成剩余的动作。
每个 Java 线程有一个独立的、定长的 SATBMarkQueue，mutator 在 barrier 里只把 old_value 压该队列中。一个队列满了之后，它就会被加到全局的 SATB 队列集合 SATBMarkQueueSet 里等待处，然后给对应的 Java 线程换一个新的、干净的队列继续执行下去。
并发标记（concurrent marker）会定期检查全局 SATB 队列集合的大小。当全局集合中队列数量超一定阈值后，concurrent marker 就会处理集合里的所有队列：把队列里记录的每个 oop 都标记上并将其引用字段压到标记栈（marking stack）上等后面做进一步标记。
Card：每个 Region 默认按照 512Kb 划分成多个 Card。
Card Table：G1 GC 的 heap 有一个覆盖整个 heap 的 card table。
RSet（Remembered Set）：每个 Region 一份。记录的是从别的 region 指向该 region 的 card。
RSet 好处：进行垃圾回收时，如果 Region1 有根对象 A 引用了 Region2 的对象 B，显然对象 B 是活的，如果没有 Rset，就需要扫描整个 Region1 或者其它 Region，才能确定对象 B 是活跃的，了 Rset 可以避免对整个堆进行扫描。
Points-into（谁引用了我）：
RSet 是 points-into 的。
Points-out（我引用了谁）：
card table 是 points-out 的，也就是说 card table 要记录的是从它覆盖的范围出发指向别的范围的针。
以分代式 GC 的 card table 为例，要记录 old -> young 的跨代指针，被标记的 card 是 old gen 范围内的。
G1 GC 则是在 points-out 的 card table 之上再加了一层结构来构成 points-into RSet：每个 r gion 会记录下到底哪些别的 region 有指向自己的指针，而这些指针分别在哪些 card 的范围内。<br

这个 RSet 其实是一个 hash table，key 是别的 region 的起始地址，value 是一个集合，里面的元是 card table 的 index。
举例来说，如果 region A 的 RSet 里有一项的 key 是 region B，value 里有 index 为 1, 2, 3, 4 的 ca d，它的意思就是 region B 的一个 card 里有引用指向 region A。所以对 region A 来说，该 RSet 录的是 points-into 的关系；而 card table 仍然记录了 points-out 的关系。
Collection Set（CSet）
在 GC 的时候，对于 old->young 和 old->old 的跨代对象引用，只要扫描对应的 CSet 的 RSet 即可。
RSet 究竟是怎么辅助 GC 的呢？在做 YGC 的时候，只需要选定 young generation region 的 R et 作为根集，这些 RSet 记录了 old->young 的跨代引用，避免了扫描整个 old generation。<b >

而 mixed gc 的时候，old generation 中记录了 old->old 的 RSet，young->old 的引用由描全部 young generation region 得到，这样也不用扫描全部 old generation region。所以 RSet 引入大大减少了 GC 的工作量。
3. Remark（STW）：根据扫描栈里的引用递归扫描可达对象，并将它们入 making bitmap。
4. cleanup：
在 marking bitmap 里统计每个 region 被标记为活的对象有多少。这个阶段如果发现完全没有活对的 region 就会将其整体回收到可分配 region 列表（free list）中。
拷贝存活对象（evacuation）（STW）
负责把一部分 region 里的活对象拷贝到空 region 里去，然后回收原本的 region 的空间。
Full GC：
收集整个堆。
触发时机：对象内存分配速度过快，Mixed GC 来不及回收，老年代被填满。
这时候会切换到 G1 之外的 Serial Old GC 来收集整个 GC heap（注意，包括 young、old、perm，这才是真正的 Full GC。
Full GC 会导致异常长时间的暂停时间，需要进行不断的调优，尽可能的避免 Full GC。
G1 不提供 Full GC。
G1 下 System.gc()是 Full GC。只有加上 -XX:+ExplicitGCInvokesConcurrent 时 G1 才会用自的并发 GC 来执行 System.gc()。
停顿预测模型 Pause Prediction Model
用户可以设定整个 GC 过程的期望停顿时间，参数-XX:MaxGCPauseMillis 指定一个 G1 收集过程目停顿时间，默认值 200ms，不过它不是硬性条件，只是期望值。那么 G1 怎么满足用户的期望呢？需要这个停顿预测模型了。G1 根据这个模型统计计算出来的历史数据来预测本次收集需要选择的 Regon 数量，从而尽量满足用户设定的目标停顿时间。
对象分配策略：
Eden 区的 TLAB(Thread Local Allocation Buffer)线程本地分配缓冲区：
每个线程在 Eden 区都有一个 TLAB，存储线程私有的对象（也就是没有发生逃逸的），是为了加快取对象的速度，因为共享变量的话，多个线程同时去拿这个共享变量会对指针进行同步，而线程私有象就只有一个线程去哪，不用同步。TLAB 的大小通过通过 start 指针和 end 指针指定，top 指针执已分配和未分配的临界点。当一个线程申请 TLAB，但 Eden 区没有那么多剩余空间是时，会触发 Yog GC。
Eden 区中分配：
Humongous 区分配：
<h2 id="GC性能指标">GC 性能指标</h2>
Throughput 吞吐量
Java 虚拟机没有花在 GC 的时间和 Java 虚拟机运行时间的百分比。
Pauses 中断
因 GC 而停顿的时间。
Footprint
overall memory a process takes to execute，影响可伸缩性（scalability）。
Promptness
time between object death, and time when memory becomes available。
参考：
《深入理解 Java 虚拟机》
java 的 gc 为什么分代？-RednaxelaFX 的回答
Java Hotspot G1 GC 的一些关键技术
请教 G1 算法的原
JVM（3）：Java GC 算法 垃圾收集器