链滴

HotSpot 探究——GC 的运行过程以及机制

作者：ChuanJinLiu

原文链接：https://ld246.com/article/1569576176545

来源网站：链滴

<img src="https://ld246.com/images/img-loading.svg" alt="" data-src="https://b3logfile
com/bing/20180720.jpg?imageView2/1/w/960/h/540/interlace/1/q/100">
写在前面： 先说说为什么要开始学这些东西，因为自己还只是一个本科生
最近在看各大公司面试题以及招聘要求的时候深深的感受到了自己的不足，所以趁自己还有属于自己
闲余时间，努力的充实自己吧！JVM 的底层原理是基本每个公司在招聘 JAVA 程序员的时候都会出
面试题，虽然有 “面试造火箭,入职拧螺丝!” 这种吐槽，但是不得不说一个优秀的 JAVA 程序员一
是对 JVM 有过深入研究的。正确学习 JVM 的方式应该先找到你学习的目的以及目标，然后参考大牛
书籍逐步深入才是正确的打开方式，我因为最近个人需要的原因，所以提前先看了一下 GC 相关的问
，这篇博客暂时先放在这里，以后逐步搭建了自己的知识体系之后会重新整合一下。另外如果有哪里
的不对的，欢迎指正。
<hr>
<h2 id="GC-是用来干嘛的">GC 是用来干嘛的</h2>
其实 GC 并不是 Java 独有的产物，它只是一个垃圾回收机制。但在
ava 中它的作用是在创建对象时对内存进行空间管理控制，将不需要的内存空间回
释放，避免无限制的内存增长导致的 OOM（OutOfMemoryError 异常）。
一般情况下，我们可以通过四种方式创建一个对象:

new 关键字
clone() 方法
反射
反序列化
 而每创建一个对象都需要对其分配内存，而内存是宝贵的资源，如何充分的利用好有限的空
清除无用的占用就是 GC 做的事情。

<h2 id="作用区域">作用区域</h2>
先了解一下 Java Hotspot VM 中 堆内存的组成：
一般划分为年轻代(Young Generation)，老年代</stron
>(Old Generation)以及永久代(持久代)(Perm Generation)。</
>
年轻代是存放生命周期短且体积较小的对象,在 JAVA 程序运行过程中这类对象通常占 80% ，生
灭灭，来来往往。而老年代呢则是存放体积较大且生命周期较长的对象。这里的生命周期在于他们在
存中存活的时间，而存活则是被程序所引用。除了这两个以外的而永久代则是保存定义类，类加载器
方法，字段，常量等一般不会改变的信息。在这之中年轻代又被分为 Eden 区和
urvivor 区，而 Survivor 区又分为 From Space 区和 To S
ace 区。一般情况下年轻代和老年代的内存大小比例为 <code>1:2 </code> or
<code>1:3</code> ，Eden 区和 Survivor 区的大小比例一般是&nbs
; <code>1:8</code> 。这些都是由参数去控制的，我们也可以去修改这些参数。
<h2 id="响应过程">响应过程</h2>
这里参考了知乎的回答：<a href="https://ld246.com/forward?goto=https%3A%2F%
Fmp.weixin.qq.com%2Fs%3F__biz%3DMzI3ODg2OTY1OQ%3D%3D%26mid%3D2247483973
26idx%3D1%26sn%3Daceedab6b71c209c4d1dc590623329d5%26chksm%3Deb5121b1dc26a
a710d69c2914f9fffd38325c12622147a42a8da9ecb9a9dfeacf75b6e30f6e%26scene%3D21%2
wechat_redirect" target="_blank" rel="nofollow ugc">一篇文章彻底搞定所有 GC 面试问题
/em>
那么什么时候 GC 会被调用呢？除了通过手动调用 <code>System.
c() </code> 方法外(这里建议在通过该方法调用 GC 时应当调用 Full GC
但 JVM 不保证一定调用)，就是在新生代或者老年代不够新对象存放时调用。我们需要了解这么一个
程，在 Hotspot 下，一开始我们的所有区块都是空的，我们在创建一个对象时，首先
将对象放在新生代的 Eden 区，当 Eden 的剩余空间不够新对象存放时
触发 minor GC ，每次调用 minor GC 时会发生这么几件事:
1.首先检测最近检查之前每次 Minor GC 时转移到老年代的对象平均大小是否超过了老年
剩余空间大小，如果剩余空间不足则直接触发 Full GC(Full GC 是指会
行 Stop the world 暂停掉所有正在运行的 Java 程序并回
整个堆内存的垃圾回收机制，使用的算法有标记–清除，标记–整理，分代收集算法)

2.如果小于，则去查看 HandlePromotionFailure 参数的值，如果为 false，则也直接触发 Full&bsp;GC；如果为 true（默认为 true，表示允许担保失败，虽然剩余空间大于之前晋升到老年代的平大小，但是依旧可能担保失败），则仅触发 Minor GC，如果期间发生老年代不足以容纳新生存活的对象，此时会触发 Full GC 。
3.检测老年代空间的确充足的情况下我们接着执行 minor GC ，(为什么一要先检测老年代的剩余空间我们一会再说)，这个时候 Eden 区是满的，我们先将当前创建但由于 Eden 区内存不够而没能创建的对象存放到 To Space 区，然后检测 Form Space 区和 Eden 区中所有还存活的对象，他们通过复制算法转移到 To Space 区，而后清空 Eden 区和&nbp;From Space 区(后面会讲复制算法是什么以及使用复制算法的好处)，然后另 Fom Space 区和 To Space 区互换，即现在的 from 区变成了 to 区，to 区变成了 form 区。另外，如果在复制到&nbp;To Space 区时 To Space 区满了，则溢出的对象将通过分配担机制进入老年代。
4.大对象直接进入老年代:那么为什么我们要先检测老年代的空间呢，这是因为当新建的对象如果个大对象，则直接让它进入老年代，倘若此时老年代空间不足则会触发 Full GC ，这里的大对象到底有多大才算大呢，其实是通过参数 -XX +PretenuerSizeThreshol 来控制”大对象的“的大小。
5.长期存活的对象将进入老年代:对象经历第一次 minor GC 从 Ede 区复制到 Survivo 区中的 To Space 区时设置其年龄为 1nbsp;，此后每经过一次 minor GC ，Survivo 区中存活的对象的年龄都 +1 ，当年龄增加到一定的临界值时，就会晋升到老年代中。这个临界值是由参数：-XX MaxTenuringThreshold 来设置的。
6.动态对象年龄判定：但是虚拟机并不总是要求对象的年龄必须达到 MaxTenuringThresold 所设定的值才能晋升到老年代，如果在 Survivor 区(To Spac 区)中相同年龄的所有对象的大小之和超过了 Survivor 空间的一半，年龄大于等于该年龄的对象就可以直接进入老年代，无需等到 MaxTenuringThreshold 中要求年龄。
<h2 id="处理对象">处理对象</h2>
在知道回收的过程之后，我们已经知道 GC 机制会回收生命周期结束的对象，即不再被引用的弃对象。那么怎么才能在一次 GC 中找到这些不再存活的对象呢？在年轻代的 minor GC 执行的<strog>复制算法中，是逐一对对象进行判断后将仍然存活的对象直接复制到新的内存空间去（o Space 区），然后将旧的内存全部清空。这种方法的好处是不会产生内存碎片，同时可以在一定程上加快处理速度，相当于空间换时间的形式，但注意这是建立在年轻代中对象 80% 都是用之则消亡快速更替的情况下才使用的，这种情况下每次需要复制的对象其实并不会很多，所以才快。但在老年的 Full GC 执行的时候由于需要对整个堆内存进行垃圾回收，那么此时不可能再使用复制算法，因为有那么多空间给你进行转换同时需要对太多大对象或者长生命周期对象进行复制，由于执行 Full GC 程序处于停顿状态，这样会严重影响程序的运行流畅度和用户体验。故此，执行 Full GC 时通常使用是标记-清除，标记-整理算法。
<h2 id="标记-清除-标记-整理-分代收集算法的实现">标记-清除、标记-整理、分代收集算法的实现</h2>
讲了这三个算法，那么他们到底是怎样实现的呢？
<hr>
<blockquote>
标记 -清除算法（Mark-Sweep）
算法分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象。之所以说它是最基础的收集算法，是因为后续的收集算法都是基于这种思路并对缺点进行改进而得到的。
    它的主要缺点有两个：
        （1）效率问题：标记和清除过程的效率都不高；<b>
        （2）空间问题：标记清除之后会产生量不连续的内存碎片，空间碎片太多可能会导致，碎片过多会导致大对象无法分配到足够的连续内存从而不得不提前触发 GC，甚至 Stop The World。

</blockquote>
<blockquote>
<img src="https://ld246.com/images/img-loading.svg" alt="标记清除.png" data-src="https://b3logfile.com/file/2019/07/%E6%A0%87%E8%AE%B0%E6%B8%85%E9%99%A4-0ec8899a png?imageView2/2/interlace/1/format/jpg">
</blockquote>
<blockquote>
复制算法（Copying）
    为解决效率问题，“复制”收集算法出现了。它将可用内存按容量划为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另一块上面，然后再把已使用过的内存空间一次清理掉。
</blockquote>
<blockquote>
    这样使得每次都是对其中的一块进行内存回收，内存分配时也就用考虑内存碎片等复杂情况，只要移动堆顶指针，按顺序分配内存即可，实现简单，运行高效。
   它的主要缺点有两个：
        （1）效率问题：在对象存活率较高时，复制操作次多，效率降低；
        （2）空间问题：内存缩小了一半；需额外空间做分配担保(老年代)
</blockquote>
<blockquote>
    From Survivor, To Survivor 使用的就是复制算法。老年代不使用这种算法

<img src="https://ld246.com/images/img-loading.svg" alt="复制算法.jpg" data-src="https://3logfile.com/file/2019/07/%E5%A4%8D%E5%88%B6%E7%AE%97%E6%B3%95-907a67d7.jpg?mageView2/2/interlace/1/format/jpg">
</blockquote>
<blockquote>
标记-整理(Mark-Compact)
    复制收集算法在对象存活率较高时就要执行较多的复制操作，效率将变低。更关键的是，如果不想浪费 50% 的空间，就需要有额外的空间进行分配担保，以应对被使用内存中所有对象都 100% 存活的极端情况，所以在老年代一般不能直接选用这种算法。
    根据老年代的特点，有人提出了另外一种“标记-整理”（Mark-Compact）算法，标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象进行清理，是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存。
</blockquote>
<blockquote>
<img src="https://ld246.com/images/img-loading.svg" alt="标记整理.png" data-src="https://b3logfile.com/file/2019/07/%E6%A0%87%E8%AE%B0%E6%95%B4%E7%90%86-47376cec png?imageView2/2/interlace/1/format/jpg">
</blockquote>
<hr>
上面的比较官方，我用大白话概括一下：
复制算法呢我上面讲过了，标记-清除算法呢总的来说就是将整个程序的对象通过某种方式遍历一遍将遍历到的对象标记，遍历结束后将没有被标记的对象释放掉。由于是直接释放，就会导致在内存中现一个一个的内存空洞。这中不连续的小内存空间就是内存碎片啦。我们知道一个对象的存放必须是续的足够的空间，是不能分段存放的，所以这样产生的内存碎片很容易导致下一次的 CG 提前到来。了改进这个缺点，就有了标记-整理算法的出现啦。根据名字就能知道，它在标记-清除的基础上对内进行了整理的意思，当然事实上并不是先清除了再整理，而是先整理了再清除，将所有存活的对象往侧移动，超出边界的部分删除，这样就得到了一个连续的内存空间，也就不存在内存碎片啦。结合图来看应该就会比较清晰啦。将上面三个算法分别作用在根据对象的特性划分的年轻代和老年代上就是谓的分代收集算法（Generational Collection）

那么这个遍历标记的过程是怎么进行的呢？《The Garbage Collection Handbook》详细地描述了一算法。这里简单说一下，遍历的过程涉及到了“引用计数法”以及“可达性分析算法”，由于前者一些内存泄漏的情况，所以现在主流的 Java 虚拟机的主流垃圾回收器采取的是可达性分析算法。我讲讲后者,简单来说就是从一个叫做 GC Roots 的对象开始遍历（GC Roots 是些由堆外指向堆内的引用），然后将被 GC Roots 引用的对象标记并加入一个集合，然后以这个集合标准继续寻找被这个集合有引用关系的对象标记并加入集合，然后巡返往复直到再也没有引用的对象止。这个构成的集合也称之为引用链。这是简单的理解，来看看更详细的概括：
/p>
<blockquote>
可达性分析算法的基本思路就是以一系列的称为 “GC Roots” 的象作为起点，从这些节点开始向下搜索，搜索所走过的路径称为引用链,当一个象到 GC Roots 没有任何引用链相连的时候（即该对象不可达），则证明此对象是不可用的。在 java 中，可作为 GC Roots 的对象包括以下几种：栈中引用的对象（栈帧中的本地变量表）、方法区中类态属性引用的对象、方法区中常量引用的对象。
</blockquote>
<blockquote>
在“可达性分析算法”中标记为不可达的对象，并非是“非死不可”的，还有回旋的余地要宣告一个对象死亡，至少要经过两次标记的过程：如果对象在进行可达性分析后发现没有与 GC&nbp;Roots 相连的引用链，那它将会被 第一次 标记并进行筛选，筛选的条件是此象是否有必要执行 finalize 方法。如果对象没有覆盖 finalize 方法或者该方法已经执行过了，则被视“没有必要执行”，宣告死亡。剩下的对象将被加入一个低优先级的队列中执行 finalize 方法。这里执行指的是会触发这个方法，并不保证执行完该方法（只保证虚拟机会触发该方法），否则如该方法在死循环，该队列就已经卡死了，GC 也瘫痪了，所以只保证触发该方法。Finalize 是对象逃脱死亡最后一次机会（可以在 finalize 方法中重新与引用链上的任何一个对象建立关联）。在触发 finalize 法之后，GC 将对该队列中的对象进行 第二次 标记，如果此时该对象仍不在引链上，该对象就会被回收。如果第二次标记前，该对象成功与引用链上的对象建立了连接，它会被移“即将回收的集合”，自救成功。注：任何一个对象的 finalize 方法只会被系统调用一次</trong>，即在 finalize 方法中最多能实现一次自救。另外，finalize 方法在 jdk9 中被标记为“废弃方法了，不建议使用。
</blockquote>
还想再详细了解的可以看看这篇博文：GC？垃圾回收？GCRoots?简单聊
有了前面的铺垫，这个时候再去看这篇文章可能很多问题就豁然开朗啦！
<h2 id="垃圾回收器">垃圾回收器</h2>
有了算法就必然有执行算法的程序，毕竟算法是不可能自己跑起来滴。执行这些算法的程序就垃圾回收器。在 JVM 中有多种垃圾回收器：
<blockquote>
新生代收集器有：Serial 收集器、ParNew 收集器、Parallel Scavenge 收集器；
</blockquote>
<blockquote>
老年代收集器：Cocurrent Mark Sweep(CMS)收集器、Serial Old（MSC）收集器、Parallel Ol 收集器。
G1 收集器: G1 独自管理整个内存，不再分新生代和老年代了。
</blockquote>
其中：
jdk9 及更新的版本中默认的是 G1 收集器 ；
jdk8 默认收集器：
新生代：Parallel Scanvage 收集器；
老年代：Parallel Old 收集器
Parallel Scanvage 使用的是复制算法， Parallel Old 使用的是标记-整理算法的垃圾回收器。
<hr>
好啦！暂时先到这里啦，想到再补充~