链滴

走进 JVM 之 GC

作者：matthewhan

原文链接：https://ld246.com/article/1600915744858

来源网站：链滴

<img src="https://ld246.com/images/img-loading.svg" alt="gc.png" data-src="https://b3logfile.com/file/2020/09/gc-45d9a734.png?imageView2/2/interlace/1/format/jpg">
<h2 id="GC知识点">GC 知识点</h2>
主要分三个部分：

如何判断该对象要回收（是垃圾）
回收的算法有哪些？
垃圾收集器

<h2 id="判断一个对象是否可以回收">判断一个对象是否可以回收</h2>
<h3 id="1--引用计数法">1. 引用计数法</h3>
<blockquote>
Python 采用该种算法，但是解决了相互引用的问题
</blockquote>
为对象添加一个引用计数器，当对象增加一个引用时计数器加 1，引用失效时计数器减 1。引用数为 0 的对象可被回收。
在两个对象出现循环引用的情况下，此时引用计数器永远不为 0，导致无法对它们进行回收。正因为循环引用的存在，因此 Java 虚拟机不使用引用计数算法。
<h3 id="2--GC-root-可达性算法-">2. GC root（可达性算法）</h3>

从 GCroot 出发遍历，标记每个可访达的对象为活动对象，遍历不到的对象（Obj4）就会被接来要讲的几种算法回收了。
GC root 可以是以下几种：

Java 方法栈桢中的局部变量；
已加载类的静态变量；
JNI handles；
已启动且未停止的 Java 线程。

多线程环境下，会产生误报和漏报。
误报：A 线程修改了访问，B 线程会导致垃圾也被标记了，不会被清除，造成内存浪费。
漏报：将仍被应用的对象标记为垃圾。
误报和漏报的理解：误报大不了不回收嘛，漏报比较严重。垃圾回收是先标记活的对象，后回收的对象，那么如果标记好后，其它线程产生了垃圾，即将活的变死了，这种内存是不会释放的。另外如果这时产生了新对象，由于没被标记为活的，所以被释放了，这就危险了。
Q：如何解决？（实际 JVM 虚拟机并不会出现这种情况）
A：Stop-the-world 以及安全点（安全词），垃圾回收线程工作时，其他非垃圾回收程一律等待！
<h2 id="垃圾回收算法">垃圾回收算法</h2>
<h3 id="1-标记---清除算法">1.标记 - 清除算法</h3>
利用 GC root 遍历到的标记（非垃圾），那么其余的就是垃圾了，直接把剩下的对象记为空闲内存。

优点：标记和清除的过程效率不高
缺点：产生大量不连续碎片，就无法给大内存的对象分配空间了

<h3 id="2-标记---整理算法-压缩算法-">2.标记 - 整理算法（压缩算法）</h3>
即把存活的对象聚集到内存区域的起始位置，从而留下一段连续的内存空间。

优点：结果是很美好的，拥有了连续的内存空间，所以一般老年代使用该种算法，因为老年代的率明显比新生代低很多。
缺点：性能拉胯，毕竟要移动。（让我想到了为么 MySQL 用 b+ 树，而不用有序数组，因为数量大的时候插入、删除、移动的效率太低下了）

<img src="https://ld246.com/images/img-loading.svg" alt="压缩" data-src="https://b3logfile.com/file/2020/09/solofetchupload9039373362148320333-ab098a53.png?imageView2/2/interlace/1/format/jpg">
<h3 id="3-复制算法">3.复制算法</h3>
将内存划分为大小相等的两块（Java8 的中 S0、S1），每次只使用其中一块，当这一块内存用了就将还存活的对象复制到另一块上面，然后再把使用过的内存空间进行一次清理。（有点像数组的ystem.arraycopy，利用 from 和 to 两个指针）。
所以所谓的 S0 和 S1 是相互复制然后全标记的，都会成为「空」的空间和复制来的对象的空间
主要的不足是每次 S0 和 S1 肯定有一块是空的，为了接下来的复制，所以也算是小浪费。
<blockquote>
HotSpot 虚拟机的 Eden 和 Survivor 大小比例默认为 8:1，保证了内存的利用率达到 90%。如每次回收有多于 10% 的对象存活，那么一块 Survivor 就不够用了，此时需要依赖于老年代进行空间配担保，也就是借用老年代的空间存储放不下的对象。
</blockquote>
<img src="https://ld246.com/images/img-loading.svg" alt="复制算法" data-src="https://3logfile.com/file/2020/09/solofetchupload7086110904909834589-1e0c19a8.png?imageView22/interlace/1/format/jpg">
<h3 id="4-分代收集算法">4.分代收集算法</h3>
现在的商业虚拟机采用分代收集算法，它根据对象存活周期将内存划分为几块，不同块采用适当收集算法。
一般将堆分为新生代和老年代。

新生代使用：复制算法
老年代使用：标记 - 清除 或者 标记 - 整理 算法

<h3 id="5-总结---内存">5.总结 <img src="https://ld246.com/images/img-loading.svg" alt="内存" data-src="https://b3logfile.com/file/2020/09/solofetchupload6962021547738270298-460bf45.png?imageView2/2/interlace/1/format/jpg"></h3>
先 Minor GC（Young GC），再 Major GC（Full GC）。S0/S1 交换 14 次后晋升至老年代（Jaa8 之后是元空间，位于本地内存），jvm 默认值是 15。
一般来说 OOM 是内存耗尽，也有超大内存（不多）的情况。
<img src="https://ld246.com/images/img-loading.svg" alt="gc-flow" data-src="https://blogfile.com/file/2020/09/solofetchupload8249039927526853800-db27e70b.png?imageView22/interlace/1/format/jpg">
以上是我画的图，其中少了一步，当新生代晋升至老年代的时候失败怎么办？其实这里是执行 Maor GC，也就是走下面的那条流程。
<h2 id="垃圾收集器">垃圾收集器</h2>
可以简单理解成新生代搞一个收集器，老年代搞一个收集器，Java8 默认的就是 Parallel + Parall Old 收集器

串行回收：Serial GC
并行回收：Parallel GC：吞吐量高（因为并行，花费 GC 的时间少，所以可以让出时间给其他非C 线程），适用于科学计算。<img src="https://ld246.com/images/img-loading.svg" alt="Parall" data-src="https://b3logfile.com/file/2019/07/jvmparold-fa222155.png?imageView2/2/w/180/format/jpg/interlace/1/q/100">
并发回收：CMS（ConcMarkSweep，并发标记清除）：其实有 STW，也有并行。后备方案是 Serial Old。而且是标记清除，会有碎片，会导致提前 Major GC；吞吐量低，但是适用是高并发、快速应的场景。<img src="https://ld246.com/images/img-loading.svg" alt="CMS" data-src="https:

/b3logfile.com/file/2019/07/jvmcms-1147f681.png?imageView2/2/interlace/1/format/jpg"></
i>
Java9 之后的：G1，Java11 之后的：ZGC（在 G1 基础上），使命就是为了替换 CMS。

<img src="https://ld246.com/images/img-loading.svg" alt="垃圾回收器" data-src="https:
/b3logfile.com/file/2019/07/jvmGC-0f70e054.png?imageView2/2/interlace/1/format/jpg"></
>
JVM 调优配了新生代，老年代会跟着配对应的垃圾收集器。
除了以上 4 种，主要还有以下 3 种：

Serial Old（已废弃）
ParNew
ParOld

<table>
<thead>
<tr>
<th>参数</th>
<th>新生代</th>
<th>新生代算法</th>
<th>老年代（元空间）</th>
<th>老年代算法</th>
</tr>
</thead>
<tbody>
<tr>
<td>-xx:+UseSerialGC</td>
<td>Serial</td>
<td>复制</td>
<td>Serial old</td>
<td>标记整理</td>
</tr>
<tr>
<td>-xx:+UseParNewGC</td>
<td>ParNew</td>
<td>复制</td>
<td>Serial old</td>
<td>标记整理</td>
</tr>
<tr>
<td>-xx:+UseParallelGC</td>
<td>Parallel</td>
<td>复制</td>
<td>Parallel old</td>
<td>标记整理</td>
</tr>
<tr>
<td>-xx:+UseConcMarkSweepGC</td>
<td>ParNew</td>
<td>复制</td>
<td>CMS / Serial old</td>
<td>标记清除</td>
</tr>
<tr>

<td>-xx:+UseG1GC</td>
<td>G1</td>
<td>不区别新老年代</td>
<td>整体是标记整理</td>
<td>局部是复制算法</td>
</tr>
</tbody>
</table>
G1 和 CMS 的区别：

G1 分块设计，会移动不同的内存对象。G1 虽然是标记清除算法，但是不会产生碎片
G1 可以设置 STW 的时间，假如设置太小，就会频繁 GC，降低吞吐量
