链滴

垃圾回收算法原理及其应用

作者：vcjmhg

原文链接：https://ld246.com/article/1618058852226

来源网站：链滴

许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

<p><img src="https://ld246.com/images/img-loading.svg" alt="" data-src="https://b3logfile
com/bing/20180401.jpg?imageView2/1/w/960/h/540/interlace/1/q/100"></p>
<h2 id="概述">概述</h2>
<p>有 java 开发经历的小伙伴必然对 <code>垃圾回收</code> 不陌生。垃圾回收简单来说就是一
自动的内存管理机制，当一个电脑上的某一块内存不在被使用时，就应该释放，供其他应用利用，而
该内存的回收过程，就称之为 <code>垃圾回收</code>。</p>
<p>那如何进行垃圾回收那？</p>
<blockquote>
<p>简单来说只有两步：</p>
<p>第一步，找到垃圾。</p>
<p>第二步，将垃圾扔掉。</p>
</blockquote>
<p>好，既然需要找到垃圾，那肯定需要对垃圾有一个判别标准，即"什么是垃圾？"</p>
<h2 id="什么是垃圾-">什么是垃圾？</h2>
<p>就计算机而言，不在使用的内存空间就都可以称之为垃圾。在 java 中一切
为对象，因而在 java 中，那些已经“死去”不再被使用的对象就可以称之为垃圾</strong
。</p>
<h2 id="如何找到垃圾-">如何找到垃圾？</h2>
<p>我们知道 JVM 在运行程序的过程中，必然会伴随着对象的创建与消亡，并且具体创建哪些对象
创建多少对象，这部分在程序运行前是不知道的，因而针对对象的分配和回收是动态</stro
g>的。所以在做垃圾回收之前，我们必须要有算法来区分这些“活着”的对象以及“死去”的对象。
/p>
<h2 id="引用计数算法">引用计数算法</h2>
<p>该算法整个过程如下：</p>
<p>首先在对象中添加一个计数器，每当有一个地方用到该对象的时候就将该计数器数加一；当引用
效时，计数器的值减一；当一个计数器的值减为零时，则认为该对象是不可能被使用的</st
ong>，即为“垃圾”对象。</p>
<p>客观来讲，虽然引用计数算法占据了一些额外的空间，但其算法原理简单，而且判定效率很高，
此它有着广泛的应用。</p>
<p>但该算法也有着其不足的一面，比如该算法很难解决对象的相互
用问题。</p>
<p>例如下边的两个对象，对象 A 和对象 B，两者相互引用。</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/siyuan/1568182249394/assets/image-20210406180221-psnf8o2.png?imageVi
w2/2/interlace/1/format/jpg"></p>
<p>因而导致即使对象 A 和对象 B 都不再使用，它们的引用计数器也不为零，如果单纯的使用引用
数算法，那么对象 A 和对象 B 永远不可能被定义为垃圾对象而被回收。</p>
<p>基于以上原因，当前主流的商用的程序语言的内存管理系统，都是通过 <code>可达性分析</co
e>（Reachability Analysis）算法来判断对象是否存活的。</p>
<h2 id="可达性分析算法">可达性分析算法</h2>
<p><code>可达性分析算法</code> 判断一个对象是否存活的过程，就像一颗森林</str
ng>的 <code>前序遍历</code>，首先以一系列（注意根节点不止一个）称
为 <code>GC Root</code> 的根对象作为起始节点，根据对象的引用关系开
向下搜索，搜索过程中所走过的路径，称为 <code>引用链</code>，如果一个对象到 <code>GC
oot</code> 间没有任何引用链，则认为该对象是可被回收的。</p>
<p>以下图为例</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/siyuan/1568182249394/assets/image-20210406201141-9bm8w8g.png?image
iew2/2/interlace/1/format/jpg"></p>
<p>Object5、Object6、Object7 即使相互引用，但由于和 <code>GC Roots</code> 没有任何
用链相连，这三个对象也是可回收的。</p>
<p>前边我们说了 <code>GC Roots</code> 是一系列根节点，那究竟什么样的对象可以作为 <co
e>GC Roots</code>？</p>
<p>在 java 技术体系中，固定可以作为 <code>GC Roots</code> 的对象包括以下几种：</p>

```
<ul>
<li>在虚拟机栈中所引用的对象，譬如各个线程被调用的方法堆栈中所使用到的参数、局部变量、临
变量。</li>
<li>在方法区中类静态属性引用的对象，譬如 java 类的引用类型静态变量。</li>
<li>本地方法中常量引用的对象，比如字符串常量池中的引用。</li>
<li>本地方法栈中 <code>JNI</code>（Native 方法）引用的对象。</li>
<li>java 虚拟机的内部引用，比如基本数据类型对应的 Class 对象，一些常驻的异常对象等，还有类
载器。</li>
<li>所有被同步锁（synchronized 关键字）持有的对象。</li>
<li>反映 Java 虚拟机内部情况的 JMXBean、JVMTI 中注册的回调、本地代码缓存等。</li>
</ul>
<p>当然除了这些固定的 <code>GC Roots</code> 集合之外，根据用户所选用的垃圾收集器以及
前回收的内容区域，也有一些其他的对象会被放到 <code>GC Roots</code> 集合中，此处不再详
。</p>
<h3 id="-再谈引用-">"再谈引用"</h3>
<p>纵观 <code>可达性分析算法</code> 执行的整个过程，对象的引用关系是及其重要的，但是
<strong>所有的引用</strong>都能够构成引用链？</p>
<p>在 <code>JDK 1.2</code> 之前 Java 中对引用的定义十分传统，如果 reference 类型的数据
存储的数值代表另一块内存的起始地址，则该 reference 数据就代表了某块内存或者某个对象的引用
按照这个定义，一个对象的引用状态只有两种，<strong>被引用</strong>或者<strong>未被引用<
strong>。</p>
<p>但这样对引用的狭义定义在描述一些“<strong>可有可无</strong>”对象时，就会显得“力
从心”。比如描述一个对象在内存充足时可以驻留内存；当内存空间不足时，可被回收，释放空间。
里可能有些同学会有疑问，我们为什么会需要一个“可有可无”的对象的呢？</p>
<p>我们以一个例子来说明这个“可有可无对象”存在的必要性：</p>
<blockquote>
<p>当我们点击一个浏览器的回退按钮时,回退时显示的网页应该重新请求还是应该从缓存中取？</p>
<p>这和浏览器的具体实现策略有关，如果一个网页在浏览结束就被回收，那么只能<strong>重新
求</strong>。如果浏览过的网页存于缓存中，那便可以从缓存中取。但如果全部都存缓存中，可能
造成<strong>大量内存的浪费</strong>，甚至会造成 OOM。</p>
<p>但如果我们把一个网页对象定义成一个“可有可无”的对象，岂不完美的解决了这个问题？如果
间足够就缓存网页，增强用户体验，如果内存不足，回收空间，提高空间利用率 :smile: !!!</p>
</blockquote>
<p>按照引用存在的必要性，将引用分成了四种：</p>
<h4 id="强引用">强引用</h4>
<p>强引用 <code>是最传统的引用，也就是程序代码之间普遍存在的</code> 引用赋值 `，比如
边的代码:</p>
<pre><code class="language-java highlight-chroma"><span class="highlight-line"><span c
ass="highlight-cl"><span class="highlight-n">Object</span> <span class="highlight-n">ob
</span> <span class="highlight-o">=</span> <span class="highlight-k">new</span> <sp
n class="highlight-n">Object</span><span class="highlight-o">();</span>
</span></span></code></pre>
<p>当<strong>内存空间不足时</strong>，JVM 虚拟机宁可跑出 <code>OutOfMemoryError</
ode> 错误，使程序异常终止也不会随意会有具有强引用的对象来解决内存不足的问题。</p>
<p>如果 <code>强引用</code> 对象不使用时，需要弱化引用从而能够使 <code>GC</code>
收，例如为引用赋空值。</p>
<pre><code class="language-java highlight-chroma"><span class="highlight-line"><span c
ass="highlight-cl"><span class="highlight-n">obj</span> <span class="highlight-o">=</s
an> <span class="highlight-kc">null</span><span class="highlight-o">;</span>
</span></span></code></pre>
<h4 id="软引用">软引用</h4>
<p>软引用就是前边所说的"可有可无对象"，当内存空间充足时，<code>GC</code> 不会回收它
当内存空间不足时才会进行回收，并且在回收时，会按照那些<strong>长时间不用的</strong>弱
用对象，有点类似于"LRU 算法"。</p>
```

```
<p>软引用创建可以通过 <code>SoftReference</code> 类来创建：</p>
<pre><code class="language-java highlight-chroma"><span class="highlight-line"><span c
ass="highlight-cl">    <span class="highlight-c1">// 强引用
</span></span></span><span class="highlight-line"><span class="highlight-cl"><span cla
s="highlight-c1"></span>    <span class="highlight-n">String</span> <span class="highlig
t-n">strongReference</span> <span class="highlight-o">=</span> <span class="highlight
k">new</span> <span class="highlight-n">String</span><span class="highlight-o">(</sp
n><span class="highlight-s">"abc"</span><span class="highlight-o">);</span>
</span></span><span class="highlight-line"><span class="highlight-cl">    <span class="h
ghlight-c1">// 软引用
</span></span></span><span class="highlight-line"><span class="highlight-cl"><span cla
s="highlight-c1"></span>    <span class="highlight-n">String</span> <span class="highlig
t-n">str</span> <span class="highlight-o">=</span> <span class="highlight-k">new</spa
> <span class="highlight-n">String</span><span class="highlight-o">(</span><span class
"highlight-s">"abc"</span><span class="highlight-o">);</span>
</span></span><span class="highlight-line"><span class="highlight-cl">    <span class="h
ghlight-n">SoftReference</span><span class="highlight-o">&lt;</span><span class="highl
ght-n">String</span><span class="highlight-o">&gt;</span> <span class="highlight-n">s
ftReference</span> <span class="highlight-o">=</span> <span class="highlight-k">new</
pan> <span class="highlight-n">SoftReference</span><span class="highlight-o">&lt;</sp
n><span class="highlight-n">String</span><span class="highlight-o">&gt;(</span><span
lass="highlight-n">str</span><span class="highlight-o">);</span>
</span></span></code></pre>
<h4 id="弱引用">弱引用</h4>
<p><code>弱引用对象</code> 也是用来描述那些可有可无的对象，但它的强度比软引用要更弱
些，它只能生存到<strong>下一次垃圾收集发生之前，<strong>当垃圾收集器开始工作时，<stron
>无论当前内存是否足够</strong>，弱引用对象</strong>都会被回收</strong>。我们可以通过 <
ode>WeakReference</code> 来创建一个弱引用对象：</p>
<pre><code class="language-java highlight-chroma"><span class="highlight-line"><span c
ass="highlight-cl"> <span class="highlight-n">WeakReference</span><span class="highlig
t-o">&lt;</span><span class="highlight-n">String</span><span class="highlight-o">&gt;<
span> <span class="highlight-n">weakReference</span> <span class="highlight-o">=</sp
n> <span class="highlight-k">new</span> <span class="highlight-n">WeakReference</sp
n><span class="highlight-o">&lt;&gt;(</span><span class="highlight-n">str</span><span
class="highlight-o">);</span>
</span></span></code></pre>
<h4 id="虚引用">虚引用</h4>
<p><strong>虚引用</strong> 顾名思义，就是<strong>形同虚设</strong>。与其他几种引用
同，<code>虚引用</code> 完全不会影响其所指向对象的生命周期，也<strong>无法</strong>
过该引用获取对象的实例，它存在的主要作用就是用来<strong>跟踪对象</strong> 被垃圾回收器<
trong>回收</strong> 的活动，即为了能在这个对象被垃圾回收时，收到一个通知。可以通过 <cod
>PhantomReference</code> 类来创建一个虚引用。</p>
<h2 id="垃圾一定马上就会被回收吗-">垃圾一定马上就会被回收吗？</h2>
<p>那些被可达性分析算法，判定为不可达的对象，就一定会被回收吗？</p>
<p>答案明显是否定的，真正宣告一个对象<strong>死亡</strong>，需要经过两次标记过程。</p

<p>我们借助一个流程图来说明</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/siyuan/1568182249394/assets/image-20210407152742-jesmofq.png?imageVi
w2/2/interlace/1/format/jpg"></p>
<p>首先一个对象在进行可达性分析后发现没有与 <code>GC Root</code> 相连的引用链，就会
该对象就行<strong>第一次标记</strong>，随后进行一次筛选，筛选条件是否有必要执行 <code>f
nilize()</code> 方法。而判定 <code>finilize()</code> 需要被执行的情况只有一种：<code>finil
ze()</code> 方法被重写过，且<strong>从来没有被调用过</strong>  。这两条任何一条不满足
```

```
虚拟机都会认为 <code>finilize()</code> 方法不需要执行。</p>
<p>这里的 <code>finilize()</code> 方法也是逃脱死亡命运的<strong>最后机会</strong>，后
对象就会正式进入回收流程无法拯救。具体流程如下：</p>
<ol>
<li>将待回收对象放到到 <code>F-Queue</code> 队列中</li>
<li>等待 <code>收集器</code> 对 <code>F-Queue</code> 队列中的对象进行第二次标记。</l
>
<li>标记后的对象会进入到待回收集合中，等待 <code>收集器</code> 回收。</li>
<li>至此对象回收结束。</li>
</ol>
<p>前边我们说了 <code>finilize()</code> 方法是对象回收自己的<strong>最后一次机会</stro
g>，那我们该如何 “抓住” 这个机会来拯救对象呢？</p>
<p>简单来说，只要我们在重写 <code>finilize()</code> 后，将该对象<strong>重新</strong>
引用链上的任意对象建立引用链即可，比如将自己的 this 关键字赋值给类变量或者对象的成员变量即
。具体代码如下：</p>
<pre><code class="language-java highlight-chroma"><span class="highlight-line"><span c
ass="highlight-cl"><span class="highlight-n">Public</span> <span class="highlight-kd">cl
ss</span> <span class="highlight-nc">FinalizeEscapeGC</span> <span class="highlight-o"
{</span>
</span></span><span class="highlight-line"><span class="highlight-cl">    <span class="h
ghlight-kd">public</span> <span class="highlight-kd">static</span> <span class="highlig
t-n">FinalizeEscapeGC</span> <span class="highlight-n">SAVE_HOOK</span> <span class
"highlight-o">=</span> <span class="highlight-kc">null</span><span class="highlight-o">
</span>
</span></span><span class="highlight-line"><span class="highlight-cl">    <span class="h
ghlight-nd">@Override</span>
</span></span><span class="highlight-line"><span class="highlight-cl">    <span class="h
ghlight-kd">protected</span> <span class="highlight-kt">void</span> <span class="highl
ght-nf">finilize</span><span class="highlight-o">()</span> <span class="highlight-kd">th
ows</span> <span class="highlight-n">Exception</span> <span class="highlight-o">{</sp
n>
</span></span><span class="highlight-line"><span class="highlight-cl">        <span class=
highlight-kd">super</span><span class="highlight-o">.</span><span class="highlight-na
>finalize</span><span class="highlight-o">();</span>
</span></span><span class="highlight-line"><span class="highlight-cl">        <span class=
highlight-n">FinalizeEscapeGC</span><span class="highlight-o">.</span><span class="hi
hlight-na">SAVE_HOOK</span> <span class="highlight-o">=</span> <span class="highlig
t-k">this</span><span class="highlight-o">;</span>
</span></span><span class="highlight-line"><span class="highlight-cl">    <span class="h
ghlight-o">}</span>
</span></span><span class="highlight-line"><span class="highlight-cl"><span class="high
ight-o">}</span>
</span></span></code></pre>
<p>上述对象在第一次被垃圾回收时，就会触发 <code>finilize()</code> 方法完成对对象的第一
拯救。但其拯救行动只会又一次。当下一次垃圾回收被触发时，对象依然会被回收，因为在第二次垃
回收时，<code>finilize()</code> 方法已经被执行过一次，会被判定没有必要被执行，直接进入回
流程。</p>
<h2 id="如何对垃圾进行收集-">如何对垃圾进行收集？</h2>
<p>前边我们已经知道了如何判定某个对象是否是垃圾，但如果想要对垃圾进行回收还需要借助于垃
回收算法来对垃圾进行收集。</p>
<p>具体垃圾收集算法关系如下图所示：</p>
<div data-code="%7B%22name%22:%20%22%E5%9E%83%E5%9C%BE%E5%9B%9E%E6%94
B6%E7%AE%97%E6%B3%95%22,%20%22children%22:%20%5B%7B%22name%22:%20%22%
5%BC%95%E7%94%A8%E8%AE%A1%E6%95%B0%E5%BC%8F%E5%9E%83%E5%9C%BE%E6
```

94%B6%E9%9B%86%22,%20%22children%22:%20%5B%7B%22name%22:%20%22java%E8%
9%9A%E6%8B%9F%E6%9C%BA%E4%B8%AD%E6%9C%AA%E4%BD%BF%E7%94%A8%22%7
%5D%7D,%20%7B%22name%22:%20%22%E8%BF%BD%E8%B8%AA%E5%BC%8F%E5%9E%8
%E5%9C%BE%E6%94%B6%E9%9B%86%22,%20%22children%22:%20%5B%7B%22name%22
%20%22%E6%A0%87%E8%AE%B0-%E6%B8%85%E9%99%A4%E7%AE%97%E6%B3%95%22
7D,%20%7B%22name%22:%20%22%E6%A0%87%E8%AE%B0-%E5%A4%8D%E5%88%B6%E
%AE%97%E6%B3%95%22%7D,%20%7B%22name%22:%20%22%E6%A0%87%E8%AE%B0-%E
%95%B4%E7%90%86%E7%AE%97%E6%B3%95%22%7D%5D%7D%5D%7D" class="language
mindmap">- 垃圾回收算法
  - 引用计数式垃圾收集
    - java虚拟机中未使用
  - 追踪式垃圾收集
    - 标记-清除算法
    - 标记-复制算法
    - 标记-整理算法

</div>
<p>在正式垃圾具体的垃圾算法，我们需要先了解一下垃圾的 <code>分代收集理论</code>。因
当前的垃圾收集器都基于该理论进行设计。</p>
<h2 id="-分代垃圾回收理论-">"分代垃圾回收理论"</h2>
<p>分代收集理论 <code>名为</code> 理论 ` 实际上就是一套符合大多数程序运行实际情况的经
法则，它建立在两个分代假说之上：</p>

弱分代假说： 绝多数对象都是朝生夕灭的，即存活
间很短。
强分代假说：熬过越多次垃圾回收的对象，就越难以消亡。

<p>基于以上原则，java 收集器将 java 堆分成了不同的区域，然后需要回收的对象根据其<strong
年龄分配到不同的区域中进行存储。这样，如果一个区域中大多数对象都是朝生夕灭，难
熬过垃圾收集过程的话，就把它们放在一起，每次回收都重点关注那些<strong
少量存活的对象，而不需要去标注那些大量需要被回收的对象，这
就可以以较低的代价回收大量的空间。对于那些难以消亡的对象，把它们放在一起，虚拟机便可以以<
trong>较低的频率来回收这个区域，从而就兼顾了垃圾回收的时间开销和空间利用率。</
>
<p>另外由于对象进行了分代，因而免不了会有跨代引用的情况发生，为了解
该问题，就提出了第三条经验法则：</p>
<ol start="3">
跨代引用假说： 跨代引用相对于同代引用来说仅仅占了极少数</
trong>。

<p>基于这条理论，我们就没有必要为了少量的跨代引用区扫描整个老年区，也没有必要梁飞空间专
记录每一个对象是否存在以及存在哪些跨带引用，只需要在新生代上建立起一个全局的数据结构--"记
集" 来解决。</p>
<h2 id="标记-清除算法">标记-清除算法</h2>
<p>首先最早出现也是最基础的垃圾回收算法是“标记-清除”算法，它回收的整个过程如下图所示：
/p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/siyuan/1568182249394/assets/image-20210407173312-rdnwj5y.png?imageVi
w2/2/interlace/1/format/jpg"></p>
<p>该算法如其名称一样，分成“标记”和“清除”两个阶段：</p>
<p>首先标记处所有需要回收的对象，然后统一进行回收，当然反之也是可以
，标记存活的对象，然后统一回收未被标记的对象。</p>
<p>但该算法有两个明显的缺点：</p>

执行效率不稳定，标记和清除的执行效率会随着对象数量的增长而<strong
降低。
空间问题，可能造成大量的内存碎片。

<h2 id="标记-复制算法">标记-复制算法</h2>
<p>为了解决 <code>标记-回收</code> 算法存在的两个问题，有人提出了 <code>半区复制</co
e> 算法，它将内存空间分成大小相等的两块，每次只是用其中的一块，当这一块内存空间使用完后
一次性将该块空间的内容复制到另一块空间中，然后清空该块空间。</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/siyuan/1568182249394/assets/image-20210407175505-jaz05u0.png?imageVi
w2/2/interlace/1/format/jpg"></p>
<p>这个算法虽然引入了复制开销，但是有效的解决了空间碎片的问题，但这
算法的缺陷也显而易见，就是将可用的内存缩小到了原来的一半，空间利用率降低。</p>
<p>为了提高空间的利用率，大神 Appel 根据对象 “朝生夕灭” 的特点，提出了一种更加优化的分
复制分代策略，称之为 <code>Appel式</code> 回收。</p>
<p>虚拟机将 java 堆分成 <code>新生代</code> 和 <code>老年代</code>，新生代又被分成 <
ode>Eden区</code>、<code>From区</code>、<code>To区</code> 三块区域。</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="" data-src="https://b3logfile
com/file/2021/04/solo-fetchupload-1573231724530527970-0d6dcfe8.png?imageView2/2/int
rlace/1/format/jpg"></p>
<p>其中我们把 Eden : From Survivor : To Survivor 空间大小设成 8 : 1 : 1，新创建的对象总是 Ede
 区创建，From 区存放当前存活的对象。To 区为空，一次 <code>gc</code> 发生后：</p>
<p>Eden 区中存活的对象和 From 区中的对象复制到 to 区，然后清空 Eden 区和 From 区；交换 Fr
m 区和 To 区的逻辑关系，即 From 区变成 To 区，To 区变成 From 区。</p>
<p>整个过程中，可以看出只有 Eden 区快慢的时候才会触发 <code>Miror GC</code>（新生代
圾回收），而 Eden 区占整个新生代的大多数，因而 <code>Miror GC</code> 的频率大为降低。<
p>
<p>这里可能会有小伙伴问，为何要预留一个 To 空间来复制？</p>
<p>这个主要是为了保证程序运行的实时性，执行的过程中可以不停顿。因为
前使用的空间如果直接回收，强制终止当前运行的进程，影响程序执行的实时性</strong
。</p>
<p>同时能够解决在垃圾回收过程中产生的内存碎片的问题，提供空间的利用率</strong
。</p>
<p>那这个 8:1:1 这个比例又是怎么来的？</p>
<p>这个是 HotSpot 虚拟机的默认设置，也即为新生代中可被回收的内存空间为整个新生代容量的 <
trong>90%。当然由于这个比例是在仅仅是在 “普通场景” 下测试得来的，在实际运行
，谁也无法保证每次只有报超过 10% 的对象可被回收。因此 Appeal 式回收又增加了一个
生门设计。当 <code>Surivior空间</code> 不足以容纳一次 <code>Minor GC</code>
之后存活的对象时，就需要依赖其他内存区域（大多数情况下老年代）来进行担保分配</st
ong>。</p>
<p>具体分配时就是在 <code>to Surivor空间</code> 没有足够空间存放上一次新生代收集下来
存活对象，这些对象就可以通过担保分配机制直接进入老年代。</p>
<p>当然新生代对象直接进入老年代会有一些风险，因此在时间分配时，虚拟机会执行以下逻辑：</
>

判断老年代连续空间是否大于新生代所有对象总空间
如果满足该条件，则可以保证此次 <code>Minor GC</code> 是安全的。
如果不成立则查看 <code>-XX:HandlePromotionFailture</code> 参数是否允许担保失败，
允许的话，直接进行一次 <code>Minor GC</code>。
如果允许则继续检查老年代最大可用空间是否大于历次晋升到老年代对象的平均大小</
trong>，如果大于，就进行 <code>Minor GC</code>，如果小于则要进行
次 <code>Full GC</code>。

<h2 id="标记-整理算法">标记-整理算法</h2>

<p>优化后的 <code>Appel算法</code> 似乎非常好用，但它必须要要有空间</strong
来进行"担保分配"，比如新生代在使用该算法时，可以通过老年代来进行担保
但老年代使用的时候怎么办？似乎并没有额外的空间可以为老年代提供担保，因而老年代一般不能直
使用该算法。</p>
<p>针对老年代对象的存亡特征，一位大神提出了 <code>标记-整理算法</co
e>，其中标记过程和原来一样，但是后续步骤不是直接对可回收的对象进行回收，而是把所有存活的
象向内存的一端进行移动，然后直接清理掉边界以外的内存。</stro
g> 具体的过程如下图所示：</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/siyuan/1568182249394/assets/image-20210407213309-2yd0nzz.png?imageVi
w2/2/interlace/1/format/jpg"></p>
<p>当然由于这个 <code>标记-整理算法</code> 是一个移动式回收算法，
而每次回收开销极大，会出现“Stop The World”现象，会影响程序的实时性，造成延迟。</p>
<p>因而该算法会用在关注高吞吐量的收集器上，比如 <code>Parallel Scavenge</code> 收集器
而关注低延迟的收集器比如 CMS 收集器则会使用 <code>标记-清除算法</co
e>，在空间碎片过多时再通过 <code>标记-整理算法</code> 来对碎片进行整理。</p>
<h2 id="垃圾收集算法工程实现">垃圾收集算法工程实现</h2>
<p>前边讲的各种垃圾回收算法更多的只是停留的理论层面之上，那它们是如何在工程上得到<stron
>应用的呢？</p>
<p>下边我们展开来说：</p>
<div data-code="%7B%22name%22:%20%22%E5%9E%83%E5%9C%BE%E6%94%B6%E9%9B
86%E5%99%A8%22,%20%22children%22:%20%5B%7B%22name%22:%20%22Serial%E6%94
B6%E9%9B%86%E5%99%A8%22%7D,%20%7B%22name%22:%20%22ParNew%E6%94%B6%
9%9B%86%E5%99%A8%22%7D,%20%7B%22name%22:%20%22Parallel%20Scavenge%E6%9
%B6%E9%9B%86%E5%99%A8%22%7D,%20%7B%22name%22:%20%22CMS%E6%94%B6%E
%9B%86%E5%99%A8%22%7D,%20%7B%22name%22:%20%22G1%E6%94%B6%E9%9B%86
E5%99%A8%22%7D%5D%7D" class="language-mindmap">- 垃圾收集器
    - Serial收集器
    - ParNew收集器
    - Parallel Scavenge收集器
    - CMS收集器
    - G1收集器
</div>
<h2 id="Serial-收集器">Serial 收集器</h2>
<p><code>Serial收集器</code> 是最古老的垃圾收集器，从它的名字 Serial（串行）就大概能够
出它是一个单线程工作的垃圾收集器，它在工作时，必须暂停其他线程的工作
到它完成收集工作。其工作的示意图如下所示：</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/siyuan/1568182249394/assets/image-20210407221026-9rqlzln.png?imageVie
2/2/interlace/1/format/jpg"></p>
<p>很明显对于该收集器会有 <code>Stop The World</code> 情况发生，但它也不是一无是处，
于其简单而高效（与其他收集器的单线程相比） 。Serial 收集器由于没有线程
互的开销，自然可以获得很高的单线程收集效率。Serial 收集器对于运行在 Client 模式下的虚拟机来
是个不错的选择。</p>
<h2 id="ParNew收集器">ParNew 收集器</h2>
<p>ParNew 收集器其实就是 Serial 收集器的多线程版本，除了使用多线程进行垃圾收集
，其余行为（控制参数、收集算法、回收策略等等）和 Serial 收集器完全一样。</p>
<p>新生代采用标记-复制算法，老年代采用标记-整理算法。</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/siyuan/1568182249394/assets/image-20210407221807-l52es4z.png?imageVi
w2/2/interlace/1/format/jpg"></p>
<p>它是许多在 Server 模式下的虚拟机的首要选择，除了 <code>Serial收集器</code> 之外，它
能和"CMS 收集器"配合工作。</p>
<h2 id="Parallel-Scavenge收集器">Parallel Scavenge 收集器</h2>

<p><code>Parallel Scaveng收集器</code> 从表面的一些特性跟 <code>PraNew收集器</code
一样，比如新生代都是通过标记-复制算法来收集，老年代则采用标记-整理算法来收集</st
ong>。</p>
<p>但它最大的不同点在于：与 CMS 收集器尽可能缩短垃圾收集时用户线程
停顿时间，Parallel Scavenge 收集器的目标则是达到一个可控制的吞吐量（高效地利用 C
U 资源）。</p>
<p>所谓吞吐量就是 CPU 中用于运行用户代码的时间与 CPU 总消耗时间的比
。</p>
<h2 id="Serial-Old收集器">Serial Old 收集器</h2>
<p>Serial 收集器的老年代版本，它同样是一个单线程收集器。它主要有两大
途：一种用途是在 JDK1.5 以及以前的版本中与 Parallel Scavenge 收集器搭配使用，另一种用途是
为 CMS 收集器的后备方案。</p>
<h2 id="Parallel-Old收集器">Parallel Old 收集器</h2>
<p>Parallel Scavenge 收集器的老年代版本。使用多线程和“标记-整理”算
。在注重吞吐量以及 CPU 资源的场合，都可以优先考虑 Parallel Scavenge 收集器和 Parallel Old
集器。</p>
<h2 id="CMS收集器">CMS 收集器</h2>
<p>CMS（Concurrent Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的
集器。它非常符合在注重用户体验的应用上使用。</p>
<p>CMS（Concurrent Mark Sweep）收集器是 HotSpot 虚拟机第一款真正意义上的并
收集器，它第一次实现了让垃圾收集线程与用户线程（基本上）同时工作。</p>
<p>从名字中的 Mark Sweep 这两个词可以看出，CMS 收集器是一种 <stron
>“标记-清除”算法 实现的，它的运作过程相比于前面几种垃圾收集器来说更加复杂一
。整个过程分为四个步骤：</p>

初始标记： 暂停所有的其他线程，标记与 GC Roots 直接相连的对象，速
很快；
并发标记： 同时开启 GC 和用户线程，从 GC Roots 直接相连的对象出发
遍历整个对象图
重新标记： 重新标记阶段就是为了修正并发标记期间因为用户程序继续运
而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段的时间
长，远远比并发标记阶段时间短
并发清除： 开启用户线程，同时 GC 线程开始对未标记的区域做清扫。</li

<p><img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:
/b3logfile.com/siyuan/1568182249394/assets/image-20210407224727-b4nm5he.png?imageV
ew2/2/interlace/1/format/jpg"></p>
<p>从它的名字就可以看出它是一款优秀的垃圾收集器，主要优点：并发收集、低停顿</st
ong>。但是它有下面三个明显的缺点：</p>

对 CPU 资源敏感； 当 CPU 核心数少的时候，导致应用程序执行速度突然
幅度变慢。
无法处理浮动垃圾，因为它在回收的过程中，用户进程继续运行会伴随这
的垃圾产生，而这部分垃圾需要等到下次 GC 时才能被回收。
它使用的回收算法-“标记-清除”算法会导致收集结束时会有大量空间碎片产生。</st
ong>

<h2 id="G1收集器">G1 收集器</h2>
<p>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
内存的机器. 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.</p>
<p>在空间划分上，较之于之前的收集器最大不同在于，G1 收集器不再坚持固定大小以及
定数量的分代区域划分而是把连续的 Java 堆划分成<strong>多个大小相等且独立</stro
g>的区域（Region），每个区域都可以根据其需要扮演 Eden 空间，Surivor 空间或者老年代空间。

/p>
<p>同时因为它将 Region 作为单次回收目标，它可以建立可预测的时间模型，来提高回收的效率，先回收那些价值高的空间（单位时间内回收的空间大）。</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="image.png" data-src="https:/b3logfile.com/siyuan/1568182249394/assets/image-20210407232011-3791ics.png?imageView2/2/interlace/1/format/jpg"></p>
<p>具体来说 G1 收集器的运作过程分为以下四个步骤：</p>

初始标记：标记与 GC Roots 直接相连的对象，修改 TAMS 指针。
并发标记
最终标记
筛选回收：负责更新 Region 的统计数据，对各个 Region 的回收价值和成本进行排序，然后将要回收的那部分 Region 复制到空的 Region 中，然后清理掉旧的 Region

<p>CMS 回收期和 G1 回收器都十分优秀，但具体场景上，如果内存较小（小于 6 个 G）那么 CMS 收集器会更占优势，反之则 G1 收集器会更好。</p>
<h2 id="总结">总结</h2>
<p>本篇文章我们主要总结 jvm 虚拟机在进行垃圾回收时所使用的算法和原理以及其工程实现，纵这么多垃圾回收算法，我们发现并没有一个万金油式的算法，每种算法以及垃圾收集器都是为了解决一类问题而设计出来的，都有对应的 Trade Off，需要我们根据应用场景加以甄别使用。</p>
<h2 id="引用">引用</h2>

理解 Java 的强引用、软引用、弱引用和引用
VM 垃圾回收