黑客派

JVM 探秘 4：四种引用、对象的生存与死亡

作者：Cellei

原文链接：https://hacpai.com/article/1526436293575

来源网站：黑客派

Java 虚拟机的内存区域中，程序计数器、Java 栈和本地方法栈是线程私有的，随线程而生随线而灭，因此这几个区域的内存回收和分配都有确定性，所以主要探究的是 Java 堆和方法区的内存分及回收。

<h2 id="Java堆">Java 堆</h2>
在 Java 堆中存放着所有的对象实例，垃圾收集器在对堆进行回收前，第一件事就是判断这些对中哪些还存活，哪些已经死去（即不会再被使用到的对象）。
<h3 id="Java中的引用">Java 中的引用</h3>
在 JDK1.2 及之前，关于引用的定义是这样的：如果一块内存中存储的数代表的是另外一块内存的起始地址，就称这块内存代表一个引用（reference）。但是这种定义比较隘，一个对象就只有被引用和没有被引用两种状态。还有这样一种“食之无味，弃之可惜”的对象：内存空间充足时，则能继续保留在内存中，如果内存空间在垃圾收集后非常紧张，则可以抛弃这些对。很多缓存功能都符合这样的应用场景。
在 JDK1.2 之后，对引用的概念进行了扩充，将引用分为强引用（Strong Reference）、软引用（Soft Reference）、弱引用（Weak Reference）、虚引用（Phantom Reference）4 种，这 4 种引用强度依次递减：

  强引用（Strong Reference）就是在代码中普遍存在的，类似“Object obj = new Object()”这类的引用，只要强引用还存在，垃圾收集器永远不会回收被引用的对象。
  软引用（Soft Reference）是用来描述有用非必需的对象。软引用关的对象，在系统将要发生内存溢出之前，将会对这些对象进行二次回收。如果这次回收后还没有足够内存，才会抛出内存溢出异常。上面所说的“食之无味，弃之可惜”的对象就是属于软引用。
  弱引用（Weak Reference）是用来描述非必需的对象，但是比软引更弱一些，弱引用关联的对象只能生存到下一次垃圾收集发生之前。当下一次圾收集时，无论内存是否足够，都会回收掉被弱引用关联的对象。
  虚引用（Phantom Reference）也称为幽灵引用或者幻影引用，它是最弱的一种引用。一对象是否有虚引用存在，完全不会对其生存时间造成任何影响，也无法通过虚引用获得一个对象实例为对象设置虚引用的目的，就是能在这个对象被收集器回收时收到一个系统通知。

<h3 id="引用计数算法">引用计数算法</h3>
很多书中判断对象是否存活的算法是这样的：给对象中添加一个引用计数器，每当一个地方引用，计数器值就加 1；当引用失效时，计数器值就减 1；任何时刻计数器为 0 的对象就是不再被使用的
引用记数算法虽然实现简单，判定效率也高，但是有一个弊端，就是它很难解决对象之间相互循引用的问题。下面的代码中，objA 和 objB 互相引用，如果使用引用计数法，这两个对象的引用计数值都为 1，会导致垃圾收集器无法回收它们。
<pre><code class="highlight-chroma">/**
 * 引用记数算法测试
 * VM Args: -XX:+PrintGCDetails
 * Run With JDK 1.8
 * */
public class ReferenceCountingGC {

```
}
</code></pre>

<script async src="https://pagead2.googlesyndication.com/pagead/js/adsbygoogle.js"></scr
pt>

<!-- 黑客派PC帖子内嵌-展示 -->

<ins class="adsbygoogle" style="display:block" data-ad-client="ca-pub-5357405790190342"
data-ad-slot="8316640078" data-ad-format="auto" data-full-width-responsive="true"></in
>
<script>
    (adsbygoogle = window.adsbygoogle || []).push({});
</script>
<p>运行结果：</p>
<pre><code class="highlight-chroma">[GC (System.gc()) [PSYoungGen: 7432K-&gt;728K(38
00K)] 7432K-&gt;736K(125952K), 0.0012008 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[Full GC (System.gc()) [PSYoungGen: 728K-&gt;0K(38400K)] [ParOldGen: 8K-&gt;667K(87552K)
 736K-&gt;667K(125952K), [Metaspace: 3491K-&gt;3491K(1056768K)], 0.0044445 secs] [Times:
user=0.00 sys=0.00, real=0.00 secs]
Heap
 PSYoungGen      total 38400K, used 333K [0x00000000d5c00000, 0x00000000d8680000, 0x00
0000100000000)
  eden space 33280K, 1% used [0x00000000d5c00000,0x00000000d5c534a8,0x00000000d7c8
000)
  from space 5120K, 0% used [0x00000000d7c80000,0x00000000d7c80000,0x00000000d8180
00)
  to   space 5120K, 0% used [0x00000000d8180000,0x00000000d8180000,0x00000000d86800
0)
 ParOldGen       total 87552K, used 667K [0x0000000081400000, 0x0000000086980000, 0x000
0000d5c00000)
  object space 87552K, 0% used [0x0000000081400000,0x00000000814a6cf0,0x00000000869
0000)
 Metaspace       used 3497K, capacity 4498K, committed 4864K, reserved 1056768K
  class space    used 387K, capacity 390K, committed 512K, reserved 1048576K
</code></pre>
<p>从运行结果看，GC 日志中包含“7432K-&gt;736K”，意味着虚拟机并没有因为两个对象互相
用就不回收它们，而说明虚拟机并不是通过引用计数算法来判断对象是否存活的。</p>
<h3 id="可达性分析算法">可达性分析算法</h3>
<p>在很多程序语言的主流实现中，都是通过可达性分析（Reachability Analysis）来判定对象是否
活的。这个算法的基本思想是：通过一系列的称为“GC Roots”的对象作为起始点，从这些节点开
向下搜索，搜索所走过的路径称为引用链（Reference Chain），当一个对象到 GC Roots 没有任何
用链相连（用图论的话说就是从 GC Roots 到这个对象不可达）时，则证明此对象是不可用的。</p>
<script async src="https://pagead2.googlesyndication.com/pagead/js/adsbygoogle.js"></scr
pt>
<!-- 黑客派PC帖子内嵌-展示 -->
<ins class="adsbygoogle" style="display:block" data-ad-client="ca-pub-5357405790190342"
data-ad-slot="8316640078" data-ad-format="auto" data-full-width-responsive="true"></in
>
<script>
    (adsbygoogle = window.adsbygoogle || []).push({});
</script>
<p>如下图所示，对象 Object 5、Object 6、Object 7 虽然互相关联，但是它们到 GC Roots 是不
```

达的，所以它们将被判定为可回收的对象：
<img src="https://static.hacpai.com/images/img-loading.svg" alt="image" data-src="htt
://static.cellei.com/20180416/Object-Alive.jpg">
在 Java 中，可作为 GC Roots 的对象有以下几种：

虚拟机栈（栈帧中的本地变量表）中引用的对象。
方法区中类静态属性引用的对象。
方法区中常量引用的对象。
本地方法栈中 JNI（即一般说的 Native 方法）引用的对象。

<h3 id="对象的自我救赎">对象的自我救赎</h3>
在可达性分析算法中不可达的对象也不是“必死无疑”的，这时它们会暂时处于“缓刑”阶段，
真正宣告死亡，至少要经历两次标记过程：第一次标记是当进行可达性分析后发现没有与 GC Roots
连的引用链，就标记一次；然后如果对象覆盖了 <code>finalize()</code> 方法并且还未执行过，
象就会被放入一个叫 <code>F-Queue</code> 的队列中，会有一个单独的线程依次执行队列中对
的 <code>finalize()</code> 方法，<code>finalize()</code> 方法是对象最后一次自我救赎的机
，只要跟 GC Roots 引用链上的任意对象建立关联，就可逃脱死亡，<code>F-Queue</code> 的
列中的对象会被第二次标记。两次标记过后如果对象还没有逃脱，那基本上它就真的被回收了。
以下代码是对象一次自我救赎的演示：
<pre><code class="highlight-chroma">/**
 * 对象的一次自我救赎
 * 1. 对象可以在GC时自我救赎
 * 2. 这种机会只有一次，因为一个对象的finalize()方法至多会被调用一次
 * */
public class FinalizeEscapeGC {

}
</code></pre>

<script async src="https://pagead2.googlesyndication.com/pagead/js/adsbygoogle.js"></scr
pt>

<!-- 黑客派PC帖子内嵌-展示 -->

<ins class="adsbygoogle" style="display:block" data-ad-client="ca-pub-5357405790190342"
data-ad-slot="8316640078" data-ad-format="auto" data-full-width-responsive="true"></in
>
<script>
    (adsbygoogle = window.adsbygoogle || []).push({});
</script>
运行结果：
<pre><code class="highlight-chroma">finalize method executed
yes, i am still alive
no, i am dead
</code></pre>
从运行结果可知，<code>SAVE_HOOK</code> 对象的 <code>finalize()</code> 方法确实
垃圾收集器触发过，并且在被回收之前成功逃脱了。代码中两段相同的代码，第二次没有成功逃脱，
因为一个对象的 <code>finalize()</code> 方法只会被系统自动调用一次。另外，<code>finalize()
/code> 方法运行代价高昂，不确定性大，无法保证对象的调用顺序，所以不建议使用此方法，可以用
<code>try-finally</code> 替代。
<h2 id="方法区">方法区</h2>
方法区也存在垃圾收集，只不过这块内存区域的垃圾收集效率比较低。在 JDK1.6</str

ng> 及之前，方法区的垃圾收集主要回收两部分内容：废弃常量和无用的类。但在 JDK1.7 /strong> 的时候运行时常量池挪到了 Java 堆中，所以现在方法区主要是回收无用的类。运行时常量回收跟堆内存中其他对象的回收方法基本一致。
同时满足以下三个条件，才会被判定为无用的类：

该类所有的实例都已经被回收，也就是 Java 堆中不存在该类的任何实例。
加载该类的 ClassLoader 已经被回收。
该类对应的 java.lang.Class 对象没有在任何地方被引用，无法在任何地方通过反射访问该类的法。

虚拟机可以对满足以上 3 个条件的无用类进行回收，也仅仅是“可以”，并不是一定会回收。是对类进行回收，HotSpot 虚拟机提供了 <code>-Xnoclassgc</code> 参数进行控制。在大量使用射、动态代理、CGLib 等 ByteCode 框架、动态生成 JSP 以及 OSGi 这类频繁自定义 ClassLoader 场景，都需要虚拟机具备类卸载的功能，以保证方法区不会溢出。
本文代码的 GitHub Repo 地址：https://github.com/cellei/JVM-Practice
<blockquote>
发表于 2018-04-16，最后编辑于 2018-04-19
 本文作者： Cellei
 本文链接： http://www.cellei.com/blog/2018/04161 br> 版权声明： 本博客所有文章除特别声明外，均采用 CC BY-NC-SA 4.0 许可协议。转载请注明出！
</blockquote>