链滴

JVM 结构与垃圾回收算法原理

作者：Tooi6
原文链接：https://ld246.com/article/1606056804181
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/20180930.jpg?imageView2/1/w/960/h/540/interlace/1/q/100"></p>
<h3 id="JVM-概述">JVM 概述</h3>
<h4 id="JVM结构">JVM 结构</h4>
<p>JVM（Java Virtual Machine）是用于运行 Java 字节码的虚拟机，由 类加载器子系统（Class Lo
der Subsystem）、运行时数据区（Runtime Data Area）、执行引擎和本地接口库（Navite Interfa
e Library）。本地接口库调用本地方法库（Navicat Method Library）与操作系统进行交互。</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="JVM 结构.png" data-src="ht
ps://b3logfile.com/file/2020/11/image-f7a42de2.png?imageView2/2/interlace/1/format/jpg"
</p>

类加载器子系统：用于将编译好的 .Class 文件加载到 JVM 中
运行时数据区：用于存储在 JVM 运行过程中产生的数据，包括程序计数器、方法区、本地方法
、虚拟机栈和虚拟机堆
执行引擎：即时编译器用于将 Java 字节码编译成具体的机器码，垃圾回收用于回收在运行过程
不再使用的对象。
本地接口库：用于调用操作系统的本地方法库完成具体的指令操作。

<h4 id="JVM-运行机制">JVM 运行机制</h4>
<blockquote>
<p>Java 源文件被编译器编译成字节码文件（.Class 文件），JVM 将字节码文件编译成相应操作系
的机器码，机器码调用相应操作系统的本地方法库执行相应的方法。</p>
</blockquote>
<h3 id="JVM内存区域">JVM 内存区域</h3>
<p>JVM 内存区域分为线程私有区域（程序计数器、虚拟机栈、本地方法区）、线程共享区域（堆
方法区）和直接内存。</p>
<h4 id="直接内存">直接内存</h4>
<p>也叫做堆外内存，在并发编程中经常使用，可以避免在 Java 堆和 Native 堆中来回复制数据带
的资源浪费</p>
<blockquote>
<p>如：JDK 的 NIO 模块通过调用本地方法 Native 函数库直接在操作系统上分配堆外内存，然后
接使用 DirectByteBuffer 对象作为这块内存的引用对内存进行操作。在一些高并发框架（Netty、Flin
、HBase、Hadoop）都有用到堆外内存。</p>
</blockquote>
<h4 id="程序计数器-线程私有-无内存溢出问题">程序计数器：线程私有，无内存溢出问题</h4>
<p>一块很小的内存区域，用于存储当前运行的线程和所执行的字节码的行号指示器。</p>
<h4 id="虚拟机栈-线程私有-描述Java方法的执行过程">虚拟机栈：线程私有，描述 Java 方法的执
过程</h4>
<p>描述 Java 方法的执行过程的内存模型，它在当前栈帧中存储了局部变量表、操作数栈、动态链
、方法出口、部分运行时数据、处理动态链接（Dynamic Linking）方法的返回值和异常分派（Dispa
ch Exception）。</p>
<blockquote>
<p>栈帧用来记录方法的执行过程，在方法被执行时虚拟机会为其创建一个与之对应的栈帧，方法的
行和返回对应栈帧在虚拟机中的入栈和出栈。</p>
</blockquote>
<h4 id="本地方法栈-线程私有">本地方法栈：线程私有</h4>
<p>与虚拟机栈类似，本地方法区栈为 Native 方法服务。</p>
<h4 id="堆-线程共享">堆：线程共享</h4>
<p>也叫运行时数据区，JVM 创建对象和产生数据都被存储在堆中，堆是被线程共享的内存区域。
垃圾回收的主要区域，从 GC 角度还可以细分为：新生代、老年代和永久代。</p>
<h4 id="方法区-线程共享">方法区：线程共享</h4>
<p>也叫永久代，用于存储常量、静态变量、类信息、即时编译其编译后的机器码、运行时常量池。<
p>
<blockquote>

<p>在类信息中包含了类的版本、字段、方法、接口、常量信息等。</p>
<p>注意：方法区是 JVM 规范的一个概念定义，并不是具体的实现。JDK7 以前方法区的实现是堆内存中的永久代，JDK8 及以后移除了永久代，方法区的实现使用的是元空间。</p>
</blockquote>
<h3 id="JVM的运行时内存-JVM堆-">JVM 的运行时内存（JVM 堆）</h3>
<p>从 GC 角度可以将 JVM 堆分为新生代、老年代和永久代。其中新生代默认占 1/3，老年代默认占 2/3，永久代占非常少的堆内存空间。</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="堆内存结构.png" data-src="https://b3logfile.com/file/2020/11/image-59c214ed.png?imageView2/2/interlace/1/format/jpg"></p>
<h4 id="新生代">新生代</h4>
<p>JVM 新创建的对象都会先保存在新生代中，因此新生代会频繁的触发 MinorGC，新生代又分为 Eden 区、SurvivorFrom 区和 SurvivorTo 区。</p>

Eden：保存新创建的对象，如果对象属于大对象，则直接将其分配到老年代，可以通过 XX:PretenureSizeThreshold 设置其大小。在 Eden 区的内存不足时会触发 MinorGC。
SurvivorTo：保留上一次 MinorGC 时的幸存者
SurivivorFrom：将上一次 MInorGC 时的幸存者作为这一次 MinorGC 的被扫描者。

<p>MinorGC 采用复制算法实现，具体过程如下：</p>

将 Eden 区和 SurivivorFrom 区中存活的对象复制到 SurvivorTo，将符合条件的对象复制到老年区（条件：年龄达到 XX:MaxTenuringThreshold 设置，或者属于大对象）
清空 Eden 区和 SurivivorFrom 区中的对象。
将 SurvivorTo 区和 SurivivorFrom 互换，原来的 SurvivorTo 成为下一次 GC 时的 SurivivorFrom。

<h4 id="老年代">老年代</h4>
<p>老年代存放长生命周期的对象和大对象。老年代的 GC 过程叫做 MajorGC。在 MinorGC 后出现老年代对象且老年代空间不足或没有连续的空间分配给大对象，会触发 MajorGC。</p>
<p>MajorGC 采用标记清除算法，该算法首先会标记所有存活的对象，然后回收未被标记的对象。</p>
<p>在没有内存空间可以分配给老年代时会抛出 Out Of Memory 异常。</p>
<h4 id="永久代">永久代</h4>
<p>永久代指内存永久保存的区域，主要存放 Class 和 Meta（元数据）的信息。GC 不会对永久代进行垃圾回收，当加载的类过多时会抛出 Out of Memory 异常。</p>
<blockquote>
<p>注意：Java8 以后永久代已经被数据区（元空间）取代，元空间与永久代的区别在于元空间没有使用虚拟机的内存，而是直接使用操作系统的本地内存。因此元空间的大小不受 JVM 内存限制。</p>
</blockquote>
<h3 id="垃圾回收与算法">垃圾回收与算法</h3>
<h4 id="确定可回收对象">确定可回收对象</h4>

引用计数法：在为一个对象添加引用时，引用计数加 1；删除引用时，引用计数减 1；如果一个对象的引用计数为 0 则说明该对象可被回收。存在循环引用问题（两个对象互相引用）。
可达性分析：首先定义几个 GC Roots，然后以这些引用作为起点向下搜索，如果在 GC Roots 和一个对象之间没有可达路径，则标记该对象为不可达的。不可达对象要经过两次标记才可以判断为可回收对象。

<blockquote>
<p>GC Roots 是指一组必须活跃的引用，例如：当前所有正在被调用的方法的引用类型的参数/局部变量/临时值。</p>

```
</blockquote>
<h4 id="常用的垃圾回收算法">常用的垃圾回收算法</h4>
<ul>
<li>标记清除算法（Mark-Sweep）</li>
</ul>
<p>分为标记和清除两个步骤，在标记阶段标记所有需要回收的对象；在清除阶段清除所有可回收的
象并释放其所占用的内存空间。<strong>存在内存碎片化问题</strong></p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="标记清除算法.png" data-src=
https://b3logfile.com/file/2020/11/image-2c0b6c69.png?imageView2/2/interlace/1/format/jp
"></p>
<ul>
<li>复制（Coping）</li>
</ul>
<p>为了解决内存碎片化问题而设计的，首先将内存区域划分为两块大小相等的内存区域，新生成的
象都被存放在区域 1，当区域 1 存储满后对区域 1 进行一次标记，并将区域 1 仍存活的对象全部复
到区域 2，最后将区域一全部清除即可。<strong>存在内存浪费问题</strong>，对象在两个区域
回复制还会影响 <strong>系统运行效率</strong></p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="复制算法.png" data-src="htt
s://b3logfile.com/file/2020/11/image-88eee9cc.png?imageView2/2/interlace/1/format/jpg">
/p>
<ul>
<li>标记整理算法（Mark-Compact）</li>
</ul>
<p>结合前两种算法的优点，先标记对象，标记完成后将存活对象移动到内存的另一端，然后清除该
对象并释放内存</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="标记整理.png" data-src="htt
s://b3logfile.com/file/2020/11/image-1982af2b.png?imageView2/2/interlace/1/format/jpg">
/p>
<ul>
<li>分代收集算法（Generational Collecting）</li>
</ul>
<p>前三种算法都堆所有类型的对象都进行回收，因此针对不同类型的对象 JVM 采用了不同的垃圾
收算法，该算法被称为分代收集算法。该算法根据对象的不同类型将内存划分为不同区域，堆内存分
新生代和老年代。</p>
<p>目前，大部分 JVM 在新生代采用复制算法，老年代采用标记整理算法</p>
<h3 id="引用类型与垃圾回收">引用类型与垃圾回收</h3>
<ul>
<li>强引用：把一个对象赋给一个引用变量时，这个引用变量就是一个强引用。强引用的对象一定为
达性状态，所以不会被垃圾回收强制回收。因此，强引用是造成 Java 内存泄漏的主要原因。</li>
<li>软引用：通过 SoftReference 类实现。如果一个对象只有软引用，则在系统内存空间不足时会将
对象回收。</li>
<li>弱引用：通过 WeakReference 类实现。如果一个对象只有若引用，则在垃圾回收过程中一定会
回收。</li>
<li>虚引用：通过 PhantomReference 类实现。和引用队列联合，主要用于列联合使用，主要用于
踪对象的垃圾回收状态。</li>
</ul>
<h3 id="垃圾收集器">垃圾收集器</h3>
<p>JVM 针对新生代和老年代分别提供了多种不同的垃圾收集器。针对新生代的有：Serial、ParNe
、Parallel Scavenge；针对老年代有：Serial Old、Parallel Old、CMS；还有针对不同区域的 G1。
/p>
<table>
<thead>
<tr>
<th>垃圾回收器</th>
```

<th>分区</th>
<th>线程数</th>
<th>算法</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>新生代</td>
<td>单线程</td>
<td>复制算法</td>
</tr>
<tr>
<td>ParNew</td>
<td>新生代</td>
<td>多线程</td>
<td>复制算法</td>
</tr>
<tr>
<td>Parallel Scavenge</td>
<td>新生代</td>
<td>多线程</td>
<td>复制算法</td>
</tr>
<tr>
<td>Serial Old</td>
<td>老年代</td>
<td>单线程</td>
<td>标记整理算法</td>
</tr>
<tr>
<td>Parallel Old</td>
<td>老年代</td>
<td>多线程</td>
<td>标记整理算法</td>
</tr>
<tr>
<td>CMS</td>
<td>老年代</td>
<td>多线程</td>
<td>标记清除算法</td>
</tr>
<tr>
<td>G1</td>
<td>针对不同区域</td>
<td>多线程</td>
<td>标记整理算法</td>
</tr>
</tbody>
</table>
<h3 id="扩展">扩展</h3>
<h4 id="Q-为什么GC要分代-">Q:为什么 GC 要分代？</h4>
<p>java 的 gc 为

么要分代？ - 知乎 (zhihu.com)</p>

原文链接：JVM 结构与垃圾回收算法原理