黑客派

对象内存分配与回收策略

作者：dreamertn9527

原文链接：https://hacpai.com/article/1513076380617

来源网站：黑客派

<h2 id="对象内存分配与回收策略">对象内存分配与回收策略</h2>

<h2 id="JVM内存结构">JVM 内存结构</h2>

JVM 内存结构主要有三大块：堆内存、方法区和栈。堆内存是 JVM 中最大的一块由年轻代和老年代组成，而年轻代内存又被分成三分，Eden 空间、From Survivor 空间、To Survivor 空间，默认情况下年轻代按照 8:1:1 的比例来分配；
方法区存储类信息、常量、静态变量等数据，是线程共享的区域，为与 Java 堆区分，方法区还一个别名 Non-Heap(非堆)；
栈又分为 Java 虚拟机栈和本地方法栈主要用于方法的执行。
在通过一张图来了解如何通过参数来控制各区域的内存大小：

控制参数

-Xms 设置堆的最小空间大小。
-Xmx 设置堆的最大空间大小。
-XX:NewSize 设置新生代最小空间大小。
-XX:MaxNewSize 设置新生代最大空间大小。
-XX:PermSize 设置永久代最小空间大小。
-XX:MaxPermSize 设置永久代最大空间大小。
-Xss 设置每个线程的堆栈大小。

没有直接设置老年代的参数，但是可以设置堆空间大小和新生代空间大小两个参数来间接控制。
<blockquote>
老年代空间大小 = 堆空间大小-年轻代大空间大小
</blockquote>
从更高的一个维度再次来看 JVM 和系统调用之间的关系：

方法区和对是所有线程共享的内存区域；而 Java 栈、本地方法栈和程序员计数器是运行线程私有的内存区域。
下面我们详细介绍每个区域的作用
<h3 id="Java堆-Heap-">Java 堆（Heap）</h3>
对于大多数应用来说，Java 堆（Java Heap）是 Java 虚拟机所管理的内存中最大的一块。Java 堆是被所有线程共享的一块内存区域，在虚拟机启动时创建。此内存区域的唯一的就是存放对象实例，几乎所有的对象实例都在这里分配内存。
Java 堆是垃圾收集器管理的主要区域，因此很多时候也被称做“GC 堆”如果从内存回收的角度看，由于现在收集器基本都是采用的分代收集算法，所以 Java 堆中还可以细为：新生代和老年代；再细致一点的有 Eden 空间、From Survivor 空间、To Survivor 空

等。
根据 Java 虚拟机规范的规定，Java 堆可以处于物理上不连续的内存空间中，只要逻辑上是连续即可，就像我们的磁盘空间一样。在实现时，既可以实现成固定大小的，也可以是可扩展的，不过当主流的虚拟机都是按照可扩展来实现的（通过-Xmx 和-Xms 控制）。
如果在堆中没有内存完成实例分配，并且堆也无法再扩展时，将会抛出 OutOfMemoryError 异。
<h3 id="方法区-Method-Area-">方法区（Method Area）</h3>
方法区（Method Area）与 Java 堆一样，是各个线程共享的内存区域，**它用于存储已被虚拟加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。**虽然 Java 虚拟机规范把方法描述为堆的一个逻辑部分，但是它却有一个别名叫做 Non-Heap（非堆），目的应该是与 Java 堆区开来。
对于习惯在 HotSpot 虚拟机上开发和部署程序的开发者来说，很多人愿意把方法区称为“永久”（Permanent Generation），本质上两者并不等价，仅仅是因为 HotSpot 虚拟机的设计团队选把 GC 分代收集扩展至方法区，或者说使用永久代来实现方法区而已。
<script async src="https://pagead2.googlesyndication.com/pagead/js/adsbygoogle.js"></script>
<!-- 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>
Java 虚拟机规范对这个区域的限制非常宽松，除了和 Java 堆一样不需要连续的内存和可以选择定大小或者可扩展外，还可以选择不实现垃圾收集。相对而言，垃圾收集行为在这个区域是比较少出的，但并非数据进入了方法区就如永久代的名字一样“永久”存在了。这个区域的内存回收目标主要针对常量池的回收和对类型的卸载，一般来说这个区域的回收“成绩”比较难以令人满意，尤其是类的卸载，条件相当苛刻，但是这部分区域的回收确实是有必要的。
根据 Java 虚拟机规范的规定，当方法区无法满足内存分配需求时，将抛出 OutOfMemoryError 异常。
方法区有时被称为持久代（PermGen）。
<img src="https://static.hacpai.com/images/img-loading.svg" alt="JUtH_20121024_RuntmeDataAreas_2_MemoryModel-1.png" data-src="http://upload-images.jianshu.io/upload_images/2832613-fc8e7de537c31e5f.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/120">
所有的对象在实例化后的整个运行周期内，都被存放在堆内存中。堆内存又被划分成不同的部分伊甸区(Eden)，幸存者区域(Survivor Sapce)，老年代（Old Generation Space）。
方法的执行都是伴随着线程的。原始类型的本地变量以及引用都存放在线程栈中。而引用关联的象比如 String，都存在在堆中。
<h3 id="程序计数器-Program-Counter-Register-">程序计数器（Program Counter Register）<h3>
程序计数器（Program Counter Register）是一块较小的内存空间，它的作用可以看做是当前程所执行的字节码的行号指示器。在虚拟机的概念模型里（仅是概念模型，各种虚拟机可能会通过一更高效的方式去实现），字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的节码指令，分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。
由于 Java 虚拟机的多线程是通过线程轮流切换并分配处理器执行时间的方式来实现的，在任何个确定的时刻，一个处理器（对于多核处理器来说是一个内核）只会执行一条线程中的指令。因此，了线程切换后能恢复到正确的执行位置，每条线程都需要有一个独立的程序计数器，各条线程之间的数器互不影响，独立存储，我们称这类内存区域为“线程私有”的内存。
如果线程正在执行的是一个 Java 方法，这个计数器记录的是正在执行的虚拟机字节码指令的地；如果正在执行的是 Natvie 方法，这个计数器值则为空（Undefined）。
此内存区域是唯一一个在 Java 虚拟机规范中没有规定任何 OutOfMemoryError 情况区域。
<h3 id="JVM栈-JVM-Stacks-">JVM 栈（JVM Stacks）</h3>

与程序计数器一样，Java 虚拟机栈（Java Virtual Machine Stacks）也是线程私有的，**它的命周期与线程相同。虚拟机栈描述的是 Java 方法执行的内存模型：**每个方法被执行的时候都会同创建一个栈帧（Stack Frame）用于存储局部变量表、操作栈、动态链接、方法出口等信息。<stron>每一个方法被调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。</st ong>
局部变量表存放了编译期可知的各种基本数据类型（boolean、byte、char、short、int、float、ong、double）、对象引用（reference 类型，它不等同于对象本身，根据不同的虚拟机实现，它可是一个指向对象起始地址的引用指针，也可能指向一个代表对象的句柄或者其他与此对象相关的位置和 returnAddress 类型（指向了一条字节码指令的地址）。

其中 64 位长度的 long 和 double 类型的数据会占用 2 个局部变量空间（Slot），其余的数据型只占用 1 个。局部变量表所需的内存空间在编译期间完成分配，当进入一个方法时，这个方法需要帧中分配多大的局部变量空间是完全确定的，在方法运行期间不会改变局部变量表的大小。
在 Java 虚拟机规范中，对这个区域规定了两种异常状况：如果线程请求的栈深度大于虚拟机所许的深度，将抛出 StackOverflowError 异常；如果虚拟机栈可以动态扩展（当前大部分的 Java 虚机都可动态扩展，只不过 Java 虚拟机规范中也允许固定长度的虚拟机栈），当扩展时无法申请到足的内存时会抛出 OutOfMemoryError 异常。
<h3 id="本地方法栈-Native-Method-Stacks-">本地方法栈（Native Method Stacks）</h3>
本地方法栈（Native Method Stacks）与虚拟机栈所发挥的作用是非常相似的，其区别不过是拟机栈为虚拟机执行 Java 方法（也就是字节码）服务，而**本地方法栈则是为虚拟机使用到的 Native方法服务。**虚拟机规范中对本地方法栈中的方法使用的语言、使用方式与数据结构并没有强制规定因此具体的虚拟机可以自由实现它。甚至有的虚拟机（譬如 Sun HotSpot 虚拟机）直接就把本地方栈和虚拟机栈合二为一。与虚拟机栈一样，本地方法栈区域也会抛出 StackOverflowError 和 OutOf emoryError 异常。
<h2 id="JVM内存分配">JVM 内存分配</h2>
对象的内存分配往大的方向上讲，就是在堆上分配。对象主要分配在新生代的 Eden 区上，如果动了本地线程分配缓冲，将按照线程优先在 TLAB 上分配。少数情况下，也可能直接会分配在老年代，分配的规则并不是百分之百固定的，其细节取决于当前使用的是哪一种垃圾收集器组合，还有虚拟与内存参数的设置。
<h3 id="对象优先在Eden分配">对象优先在 Eden 分配</h3>
大多数情况下，新生代在 Eden 区中分配。当 Eden 区没有足够的空间进行分配时，虚拟机将发一次 Minor GC。
Minor GC 和 Full GC

新生代 GC(Minor GC): 指发生在新生代的垃圾收集动作，因为 Java 对象大多具有朝生夕灭的性，所以 Minor GC 非常频繁，一般回收速度也很快。
老年代 GC(Major GC / Full GC)：指发生在老年代的 GC，出现了 MajorGC，经常会伴随至少次的 Minor GC（但是并非绝对的，在 Parallel Scavenge 收集器的收集策略就有直接进行 Major GC的策略选择过程）。MajorGC 的速度一般会比 Minor GC 慢 10 倍以上。

<h3 id="大对象直接进入老年代">大对象直接进入老年代</h3>
所谓大对象，是指需要大量连续空间的 Java 对象，最典型的对象就是那种很长的字符串以及数。大对象对虚拟机的内存分配来说就是一个坏消息，经常出现大对象容易导致内存还有不少空间就提触发垃圾收集以获取足够的连续空间来安置他们。
<h3 id="长期存活的对象进入老年代">长期存活的对象进入老年代</h3>
既然虚拟机采用了分代收集的思想来管理内存，那么内存回收时就必须能识别哪些对象应该放在

生代，哪些对象应该放在老年代中。为了做到这点，虚拟机给每个对象对应了一个对象年龄计数器。果对象在 Eden 出生并经过第一次 Minor GC 后任然存活，并且可以被 Survivor 容纳的话，将被移到 Survivor 空间中，并且设置对象的年龄为 1,。对象在 Survivor 每经历一次 Minor GC，年龄就增加 1 岁，当他的年龄增加到一定程度(默认为 15 岁)，就将会被晋级到老年代中。对象晋升老年代年龄的值，可以通过参数-XX:MaxTenuringThreshold 进行设置。

<h3 id="动态对象年龄的判定">动态对象年龄的判定</h3>
为了更好的适应不同程序的内存状况，虚拟机并不是永远的要求对象的年龄必须达到 MaxTenurigThreshold 才能晋升到老年代。如果在 Survivor 空间中相同年龄所有对象大小的总和大于 Survivor 空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代，无须等待到 MaxTenuringThresold 中要求的年龄。
<h3 id="空间分配担保">空间分配担保</h3>
在发送 Minor GC 之前，虚拟机会先检查老年代最大可用的连续空间是否大于新生代所有对象总间。如果这条件成立，那么 Minor GC 可以确保是安全的。如果不成立，则虚拟机会查看 HandlePromotionFailure 设置值是否允许担保失败。如果允许，那么会继续检查老年代最大可用的连续空间是大于历次晋升到老年代对象的平均大小，如果大小，讲尝试进行一次 Minor GC，尽管这次 Minor gc 是有风险的；如果小于 HandlePromotionFailure 设置不允许冒险，那这时也要改为进行一次 FUll GC