Java 内存区域

作者：DongXiaokai0819
原文链接：https://ld246.com/article/1572700519334
来源网站：链滴

<h2 id="Java内存区域">Java 内存区域</h2>
<h3 id="1-运行时数据区域">1.运行时数据区域</h3>
Java 虚拟机在执行 Java 程序的过程中会把它管理的内存划分成若干个不同的数据区域。

程序计数器
虚拟机栈
本地方法栈
Java 堆
方法区（运行时常量池）
直接内存

站在线程的角度看：
Java 堆与方法区是所有线程共享的
程序计数器、虚拟机栈、本地方法栈是线程私有的

<h4 id="1-1程序计数器">1.1 程序计数器</h4>
<blockquote>
程序计数器是一块较小的内存空间，可以看作是当前线程所执行的字节码的行号指示器。
线程私有内存区域。
是 Java 虚拟机规范中唯一一个没有规定任何 OutOfMemoryError 情况的区域。
它的生命周期随着线程的创建而创建，随着线程的结束而死亡。
</blockquote>
程序计数器的俩个作用：

（供字节码解释器使用）字节码解释器通过改变程序计数器来依次读取指令，从而实现代码的流控制，如：顺序执行、选择、循环、异常处理。
在多线程的情况下，程序计数器用于记录当前线程执行的位置，从而当线程被切换回来的时候能知道该线程上次运行到哪儿了。

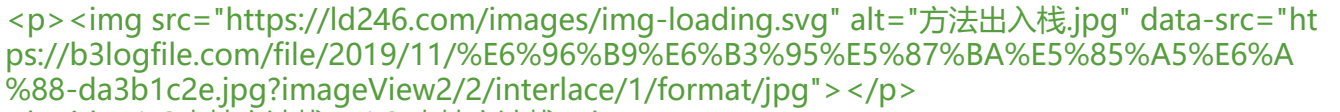
<h4 id="1-2虚拟机栈">1.2 虚拟机栈</h4>
Java 内存可以粗糙的区分为堆内存（Heap）和栈内存 (Stack)，其中的栈就是现在说的虚拟机，或者说是虚拟机栈中局部变量表部分。
<blockquote>
虚拟机栈描述的是 Java 方法执行的内存模型：每个方法在执行的同时都会创建一个栈帧（Stack rame）用于存储局部变量表、操作数栈、动态链接、方法出口等信息。（栈帧是方法运行时的基础据结构，这里先了解。）
每一个方法从调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程。
线程私有内存区域。
生命周期与线程相同。
</blockquote>
局部变量表主要存放了编译器可知的各种数据类型（boolean、byte、char、short、int、float、ong、double）、对象引用（reference 类型，它不同于对象本身，可能是一个指向对象起始地址的用指针，也可能是指向一个代表对象的句柄或其他与此对象相关的位置）。
其中 64 位长度的 long 和 double 类型的数据会占用俩个局部变量空间，其它的数据类型只会占用个。
局部变量表所需的内存空间在编译期间就完成分配了，也就是说，当进入一个方法时，这方法所需要在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。

Java 虚拟机栈可能会出现两种异常：StackOverFlowError 和 OtOfMemoryError。

StackOverFlowError： 若 Java 虚拟机栈的内存大小不允许动态扩展，那么当线程请求栈的深度过当前 Java 虚拟机栈的最大深度的时候，就抛出 StackOverFlowError 异常。
OutOfMemoryError： 若 Java 虚拟机栈的内存大小允许动态扩展，且当线程请求栈时内存用完，无法再动态扩展了，此时抛出 OutOfMemoryError 异常。

<h5 id="Java中方法是如何调用的呢-">Java 中方法是如何调用的呢？</h5>
Java 栈可用类比数据结构中栈，Java 栈中保存的主要内容是栈帧，每一次函数调用都会有一个应的栈帧被压入 Java 栈，每一个函数调用结束后，都会有一个栈帧被弹出。
Java 方法有两种返回方式：

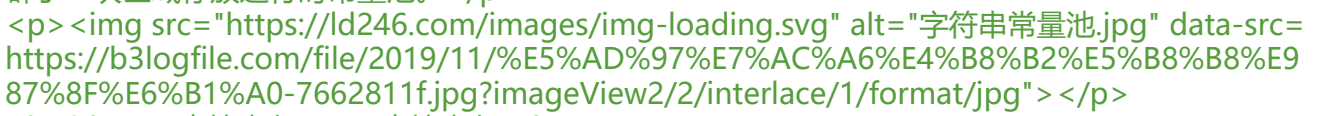
return 语句。
抛出异常。
不管哪种返回方式都会导致栈帧被弹出。

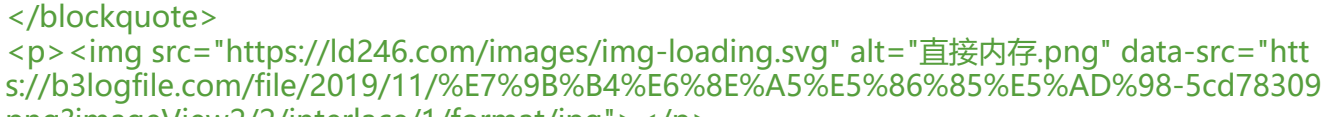

<h4 id="1-3本地方法栈">1.3 本地方法栈</h4>
<blockquote>
和虚拟机栈所发挥的作用非常相似，区别是： 虚拟机栈为虚拟机执行 Java 方法 （也就是字节码服务，而本地方法栈则为虚拟机使用到的 Native 方法服务。
在 HotSpot 虚拟机中和 Java 虚拟机栈合二为一。
</blockquote>
本地方法被执行的时候，在本地方法栈也会创建一个栈帧，用于存放该本地方法的局部变量表、作数栈、动态链接、出口信息。
方法执行完毕后相应的栈帧也会出栈并释放内存空间，也会出现 StackOverFlowError 和 OutOfemoryError 两种异常。
<h4 id="1-4Java堆-Java-Heap-">1.4Java 堆(Java Heap)</h4>
<blockquote>
Java 虚拟机所管理的内存中最大的一块，Java 堆是所有线程共享的一块内存区域。
在虚拟机启动时创建。
此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例以及数组都在这里分配内存。（随着术的演变，所有的对象都在堆上分配，也变得不是那么绝对了）
</blockquote>
Java 堆是垃圾收集器管理的主要区域，因此也被称作 GC 堆（Garbage Collected Heap）。
现在的收集器基本都采用分代垃圾收集算法，所以 Java 堆还可以细分为：新生代和老年代：再细致点有：Eden 空间、From Survivor、To Survivor 空间等。
从内存分配的角度来看，线程共享的 Java 堆中可能划分出多个线程私有的分配缓冲区（Thread ocal Allocation Buffer ,TLAB）。无论如何划分，都与存放内容无关，无论哪个区域，存放的都是对实例，进一步划分的目的是更好地回收内存，或者更快地分配内存。
Java 堆可以处于物理上不连续的内存空间中，只要逻辑上是连续的即可。
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。
可用以下参数来配置堆：

-Xmx：堆的最大值；例：- Xmx256m
-Xms：堆的最小值；
-Xmn：新生代的大小；
-XX:NewSize；新生代最小值；
-XX:MaxNewSize：新生代最大值；

<h4 id="1-5方法区--运行时常量池-">1.5 方法区（运行时常量池）</h4>
<blockquote>
方法区与 Java 堆一样，是各个线程共享的内存区域.
用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。
</blockquote>
虽然 Java 虚拟机规范把方法区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 堆区分开来。
Java 虚拟机规范中对方法区的限制非常宽松，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。
JDK 1.8 后，方法区（HotSpot 的永久代）被彻底移除了（JDK1.7 就已经把原本放在永久代中字符串常量池移出），取而代之是元空间，元空间使用的是直接内存。
当方法区无法满足内存分配需求时，将会抛出 OutOfMemoryError。
<h5 id="方法区与永久代的关系">方法区与永久代的关系</h5>
<blockquote>
《Java 虚拟机规范》只是规定了有方法区这么个概念和它的作用，并没有规定如何去实现它。么，在不同的 JVM 上方法区的实现肯定是不同的了。 方法区和永久代的关系很像 Java 中接口和类关系，类实现了接口，而永久代就是 HotSpot 虚拟机对虚拟机规范中方法区的一种实现方式。 也就说，永久代是 HotSpot 的概念，方法区是 Java 虚拟机规范中的定义，是一种规范，而永久代是一种现，一个是标准一个是实现，其他的虚拟机实现并没有永久代这一说法。
</blockquote>
<h5 id="为什么要将永久代--PermGen--替换为元空间--MetaSpace--呢-">为什么要将永久代 (PerGen) 替换为元空间 (MetaSpace) 呢?</h5>
整个永久代有一个 JVM 本身设置固定大小上限，无法进行调整，而元空间使用的是直接内存，本机可用内存的限制，并且永远不会得到 java.lang.OutOfMemoryError。

-XX：MaxMetaspaceSize 标志设置最大元空间大小，默认值为 unlimited，这意味着它只受系内存的限制。
-XX：MetaspaceSize 调整标志定义元空间的初始大小如果未指定此标志，则 Metaspace 将根运行时的应用程序需求动态地重新调整大小。

<h4 id="1-6运行时常量池">1.6 运行时常量池</h4>
运行时常量池是方法区的一部分。Class 文件中除了有类的版本、字段、方法、接口等描述信息，还有常量池信息（用于存放编译期生成的各种字面量和符号引用）
既然运行时常量池是方法区的一部分，自然受到方法区内存的限制，当常量池无法再申请到内存会抛出 OutOfMemoryError 异常。
JDK1.7 及之后版本的 JVM 已经将运行时常量池从方法区中移了出来，在 Java 堆（Heap）中辟了一块区域存放运行时常量池。


<h4 id="1-7直接内存">1.7 直接内存</h4>
<blockquote>
直接内存并不是虚拟机运行时数据区的一部分，也不是虚拟机规范中定义的内存区域，但是这部内存也被频繁地使用。
也可能导致 OutOfMemoryError 异常出现。
</blockquote>


JDK1.4 中新加入的 NIO(New Input/Output) 类，引入了一种基于通道（Channel） 与缓存区 Buffer） 的 I/O 方式，它可以直接使用 Native 函数库直接分配堆外内存，然后通过一个存储在 Java 堆中的 DirectByteBuffer 对象作为这块内存的引用进行操作。这样就能在一些场景中显著提高性能因为避免了在 Java 堆和 Native 堆之间来回复制数据。

本机直接内存的分配不会受到 Java 堆的限制，但是，既然是内存就会受本机总内存大以及处理器寻址空间的限制。
可通过：

-XX:MaxDirectMemorySize 来设置（默认与堆内存最大值一样），所以也会出现 OOM 异常。/li>

<h4 id="1-8各个版本内存区域的变化">1.8 各个版本内存区域的变化</h4>
jdk1.6：

jdk1.7:

jdk1.8

<h4 id="1-9辨析堆和栈">1.9 辨析堆和栈</h4>
功能

栈以栈帧的方式存储方法调用的过程， 并存储方法调用过程中基本数据类型的变量（ int、 shor 、 long、 byte、float、 double、 boolean、 char 等） 以及对象的引用变量， 其内存分配在栈上 变量出了作用域就会自动释放；
堆内存用来存储 Java 中的对象。 无论是成员变量， 局部变量， 还是类变量， 它们指向的对象存储在堆内存
中；

线程独享还是共享

栈内存归属于单个线程， 每个线程都会有一个栈内存， 其存储的变量只能在其所属线程中可见，即栈内存可以
理解成线程的私有内存。
堆内存中的对象对所有线程可见。 堆内存中的对象可以被所有线程访问。

空间大小

栈的内存要远远小于堆内存， 栈的深度是有限制的， 可能发生 StackOverFlowError 问题。</li

<h4 id="1-11栈上分配">1.11 栈上分配</h4>
栈上分配是虚拟机提供的一种优化技术，基本思想是：
<blockquote>
对于线程私有的对象，将它打散分配在栈上，而不分配在堆上。好处是对象跟着方法调用自行销，不需要进行垃圾回收，可以提高性能。
栈上分配需要逃逸分析作为技术基础
逃逸分析的目的是判断对象的作用域是否会逃逸出方法体。
注意，任何可以在多个线程之间共享的对象，一定都属于逃逸对象。
</blockquote>

<pre><code class="language-public highlight-chroma">String x = “”;
User u = ...
.....
}
</code></pre>
User 类型的对象 u 就没有逃逸出方法 test。
<pre><code class="language-public highlight-chroma">String x = “”;
User u = ...
.....
return u;
}
</code></pre>
User 类型的对象 u 就逃逸出方法 test。
<h5 id="如何启用栈上分配">如何启用栈上分配</h5>

-server JVM 运行的模式之一, server 模式才能进行逃逸分析， JVM 运行模式还有 mix 混合模式/client
-XX:+DoEscapeAnalysis：启用逃逸分析(默认打开)
-XX:+EliminateAllocations：标量替换(默认打开)
-XX:-UseTLAB 关闭本地线程分配缓冲(TLAB 下文有讲)

对栈上分配发生影响的参数就是三个，-server、-XX:+DoEscapeAnalysis 和-XX:+EliminateAllcations，任何一个发生变化都不会发生栈上分配，因为启用逃逸分析和标量替换默认是打开的，所，在我们的例子中，JVM 的参数只用-server 一样可以有栈上替换的效果(以 Mark 老师机器上 JDK1.8 为例，其他版本 JDK 请自行尝试)。
<h3 id="2-虚拟机中的对象">2.虚拟机中的对象</h3>
<h4 id="2-1对象的创建">2.1 对象的创建</h4>
下图为 Java 对象的创建过程，要掌握每一步干了什么。


<h5 id="1-类加载检查">1、类加载检查</h5>
虚拟机遇到一条 new 指令时，首先将去检查这个指令的参数是否能在常量池中定位到这个类的号引用，并且检查这个符号引用代表的类是否已被加载过、解析和初始化过。
如果没有，那必须先执行相应的类加载过程。
<h5 id="2-分配内存">2、分配内存</h5>
在类加载检查通过后，接下来虚拟机将为新生对象分配内存。对象所需的内存大小在类加载完成便可确定，为对象分配空间的任务等同于把一块确定大小的内存从 Java 堆中划分出来。
分配方式有** “指针碰撞” **和 “空闲列表” 两种，选择那种分配方式由 Java 堆是否规整决定，而 Java 堆是否规整又由所采用的垃圾收集器是否带有压缩整理功能决定。
<h6 id="指针碰撞---空闲列表">指针碰撞 & 空闲列表</h6>
指针碰撞：

如果 Java 堆中内存是绝对规整的，所有用过的内存都放在一边，空闲的内存放在另一边，中间着一个指针作为分界点的指示器，那所分配内存就仅仅是把那个指针向空闲空间那边挪动一段与对象小相等的距离，这种分配方式称为“指针碰撞”。

空闲列表:

如果 Java 堆中的内存并不是规整的，已使用的内存和空闲的内存相互交错，那就没有办法简单进行指针碰撞了，虚拟机就必须维护一个列表，记录上哪些内存块是可用的，在分配的时候从列表中到一块足够大的空间划分给对象实例，并更新列表上的记录，这种分配方式称为“空闲列表”。

在使用 Serial、ParNew 等带有 Compact 过程的收集器时，系统采用的分配算法是指针碰撞。<r>
而使用 CMS 这种基于 Mark-Sweep 算法的收集器时，系统采用的是空闲列表。
<h6 id="内存分配并发安全问题">内存分配并发安全问题</h6>
在创建对象的时候有一个很重要的问题，就是线程安全，因为在实际开发过程中，创建对象是很繁的事情，作为虚拟机来说，必须要保证线程是安全的，通常来讲，虚拟机采用两种方式来保证线程全：

CAS+ 失败重试： CAS 是乐观锁的一种实现方式。所谓乐观锁就是，每次不加锁而是假设没有突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。虚拟机采用 CAS 配上失败重试的式保证更新操作的原子性。
TLAB： 为每一个线程预先在 Eden 区分配一块儿内存，JVM 在给线程中的对象分配内存时，首在 TLAB 分配，当对象大于 TLAB 中的剩余内存或 TLAB 的内存已用尽时，再采用上述的 CAS 进行存分配。

<h6 id="TLAB-Thread-Local-Allocation-Buffer-">TLAB（Thread Local Allocation Buffer）</h>
把内存分配的动作按照线程划分在不同的空间之中进行，即每个线程在 Java 堆中预先分配一小私有内存，也就是本地线程分配缓冲（Thread Local Allocation Buffer,TLAB），高效率，当 Buffer 容量不够的时候，再重新从 Eden 区域申请一块继续使用。
TLAB 的目的是在为新对象分配内存空间时，让每个 Java 应用线程能在使用自己专属的分配指针来分空间，减少同步开销。
TLAB 只是让每个线程有私有的分配指针，但底下存对象的内存空间还是给所有线程访问的，只是其线程无法在这个区域分配而已。当一个 TLAB 用满（分配指针 top 撞上分配极限 end 了），就新申一个 TLAB。
<h5 id="3-初始化零值">3、初始化零值</h5>
内存分配完成后，虚拟机需要将分配到的内存空间都初始化为零值（如 int 值为 0，boolean 值为 false 等等，但不包括对象头），这一步操作保证了对象的实例字段在 Java 代码中可以不赋初始值就接使用，程序能访问到这些字段的数据类型所对应的零值。
<h5 id="4-设置对象头">4、设置对象头</h5>
初始化零值完成之后，虚拟机要对对象进行必要的设置，例如这个对象是那个类的实例、如何才找到类的元数据信息、对象的哈希码、对象的 GC 分代年龄等信息。 这些信息存放在对象头中。 另，根据虚拟机当前运行状态的不同，如是否启用偏向锁等，对象头会有不同的设置方式。
<h5 id="5-执行-init-方法">5、执行 init 方法</h5>
在上面工作都完成之后，从虚拟机的视角来看，一个新的对象已经产生了，但从 Java 程序的视来看，对象创建才刚开始， 方法还没有执行，所有的字段都还为零。所以一般来说，执行 new 指令后会接着执行 方法，把对象按照程序员的意愿进行初始化，这样一个真正可用的对象才算完全产生来。
<h4 id="2-2对象的内存布局">2.2 对象的内存布局</h4>

在 Hotspot 虚拟机中，对象在内存中的布局可以分为 3 块区域：

对象头（Header）
实例数据（Instance Data）
对齐填充(Padding)

Hotspot 虚拟机的对象头包括两部分信息，第一部分用于存储对象自身的身运行时数据（哈希码、GC 分代年龄、锁状态标志等等）
另一部分是类型指针，即对象指向它的类元数据的指针，虚拟机通过这个指针确定这个对象是那个类的实例。

实例数据部分是对象真正存储的有效信息，也是在程序中所定义的各种类的字段内容。
对齐填充部分不是必然存在的，也没有什么特别的含义，仅仅起占位作用
因为 Hotspot 虚拟机的自动内存管理系统要求对象起始地址必须是 8 字节的整数倍，换句话说就是象的大小必须是** 8 字节的整数倍**。而对象头部分正好是 8 字节的倍数（1 倍或 2 倍），因此，当象实例数据部分没有对齐时，就需要通过对齐填充来补全。
<h4 id="2-3对象的访问定位">2.3 对象的访问定位</h4>
立对象就是为了使用对象，我们的 Java 程序通过栈上的 reference 数据来操作堆上的具体对象对象的访问方式由虚拟机实现而定，目前主流的访问方式有 ① 使用句柄和 ② 直接指针两种：

句柄： 如果使用句柄的话，那么 Java 堆中将会划分出一块内存来作为句柄池，reference 中存的就是对象的句柄地址，而句柄中包含了对象实例数据与类型数据各自的具体地址信息；

<ol start="2">
直接指针： 如果使用直接指针访问，那么 Java 堆对象的布局中就必须考虑如何放置访问类型数的相关信息，而 reference 中存储的直接就是对象的地址。

这两种对象访问方式各有优势。
使用句柄来访问的最大好处是 reference 中存储的是稳定的句柄地址，在对象被移动时只会改变句柄的实例数据指针，而 reference 本身不需要修改。
使用直接指针访问方式最大的好处就是速度快，它节省了一次指针定位的时间开销。
对 Sun HotSpot 而言，它是使用直接指针访问方式进行对象访问的。
<h3 id="3-OOM-OutOfMemoryError-">3.OOM(OutOfMemoryError)</h3>
<blockquote>
OOM 如果通俗点儿说，就是 JVM 内存不够用了，javadoc 中对 OutOfMemoryError 的解释，没有空闲内存，并且垃圾收集器也无法提供更多内存。
</blockquote>
这里面隐含着一层意思是，在抛出 OutOfMemoryError 之前，通常垃圾收集器会被触发，尽其能去清理出空间。
上面所说的只是一般情况，而当我们去分配一个超大对象，类似于一个超大数组，这个数组的大小已超过了堆的最大值，JVM 可以判断出垃圾收集并不能解决这个问题，从而直接抛出 OutOfMemoryEr or。
除了程序计数器，其他区域都有可能会因为可能的空间不足发生 OutOfMemoryError，总结如：

堆内存不足是最常见的 OOM 原因之一，抛出的错误信息是“java.lang.OutOfMemoryError:Jav heap space”，原因可能千奇百怪，例如，可能存在内存泄漏问题；也很有可能就是堆的大小不合，比如我们要处理比较可观的数据量，但是没有显式指定 JVM 堆大小或者指定数值偏小；或者出现 J M 处理引用不及时，导致堆积起来，内存无法释放等。
而对于 Java 虚拟机栈和本地方法栈，这里要稍微复杂一点。如果我们写一段程序不断的进行递调用，而且没有退出条件，就会导致不断地进行压栈。类似这种情况，JVM 实际会抛出 StackOverFl wError；当然，如果 JVM 试图去扩展栈空间的的时候失败，则会抛出 OutOfMemoryError。
对于老版本的 Oracle JDK，因为永久代的大小是有限的，并且 JVM 对永久代垃圾回收（如，常池回收、卸载不再需要的类型）非常不积极，所以当我们不断添加新类型的时候，永久代出现 OutOf emoryError 也非常多见，尤其是在运行时存在大量动态类型生成的场合；类似 Intern 字符串缓存占

太多空间，也会导致 OOM 问题。对应的异常信息，会标记出来和永久代相关： “java.lang.OutOfMoryError: PermGen space” 。
随着元数据区的引入，方法区内存已经不再那么窘迫，所以相应的 OOM 有所改观，出现 OOM 异常信息则变成了： “java.lang.OutOfMemoryError: Metaspace” 。
直接内存不足，也会导致 OOM

<h3 id="参考">参考</h3>
《深入理解 Java 虚拟机》
Java Guide：https://snailclimb.top/JavaGuide/#/?id=java