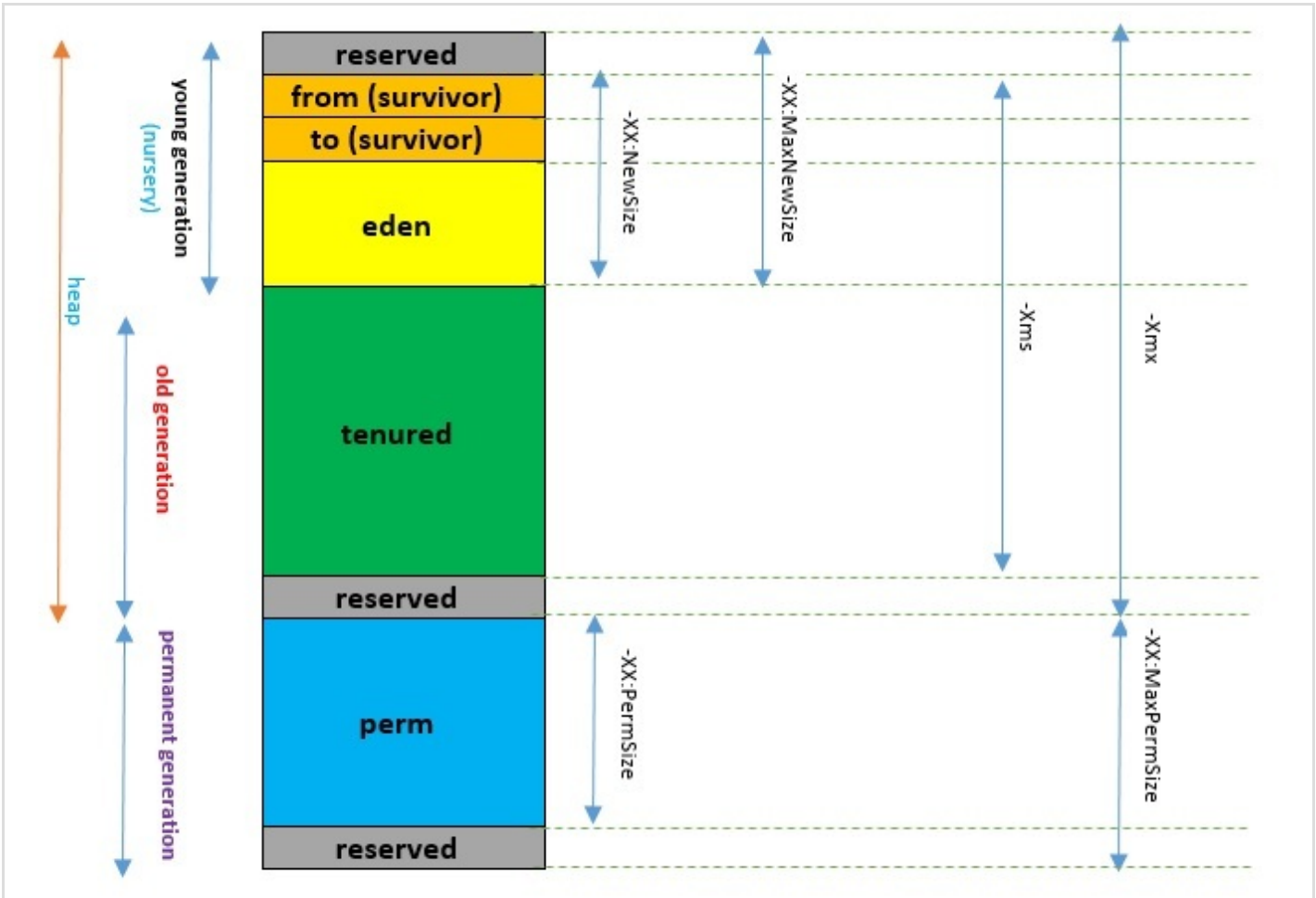
Java 内存结构备忘录

本文详细描述了 Java 堆内存模型，垃圾回收算法以及处理内存泄露的最佳方案，并辅之以图表，希望能对理解 Java 内存结构有所帮助。原文作者 Sumith Puri，本文系 OneAPM 工程师编译整理。

下图展示了 Java 堆内存模型，以及运行在 Java 虚拟机中任意 Java 应用的 PermGen (内存永久保存区域)，下面的比率展示了 JVM 各代类型允许的内存大小分配情况，所有的数据均适用于 Java 1.7 及以下版本。该图也被称为 Java 内存模型的“管理区(Managed Area)”。

Java 内存结构(Java 内存模型)



Java Memory Architecture (Java Memory Model) and Ratio

Usual Ratios (from Total Available Heap Memory)	
Young Generation	**1/3**
Survivor 1 (From)	1/8
Survivor 2 (To)	1/8
Eden	3/4
Older Generation	**2/3**

除此之外，还有一块堆栈区(Stack Area)，可通过 -Xss 选项进行配置。该区域存储了所有线程的堆引用、本地引用、程序计数器寄存器、代码缓存以及本地变量。该区域也称为内存模型的本地区(Native Area)。

Java 内存模型(结构)的管理区

[Young Generation/Nursery] 伊甸园区(Eden Space)

所有新对象都首先在 Eden Space 创建。一旦该区达到由 JVM 设定的任意阈值，新生代垃圾回收机制(Minor GC)就会启动。它会首先清除所有的非引用对象，并将引用对象从 'eden' 与 'from' 区移至 'to' 幸存者区。垃圾回收一结束，'from' 与 'to' 的角色(名字)就会对换。

[Young Generation/Nursery] 幸存者 1 区 (From)

这是幸存者区的一部分。或者视为幸存者区中的一个角色。这儿就是之前垃圾回收中的 'to' 角色。

[Young Generation/Nursery] 幸存者 2 区 (To)

这也是幸存者区的一部分。也可以视为幸存者区中的一个角色。垃圾回收过程中的所有引用对象都会从 'from' 与 'eden' 区移至此处。

[Old Generation] 年老代区(Tenured)

根据阈值限定的不同，对象们会从 'to' 幸存者区移至年老代区。你可以使用 **-XX:+PrintTenuringDistribution** 检查阈值，该指令会按照年龄显示对象(占用的字节空间)。年龄是指对象在幸存者区内移动的次数。

其他重要的标记还有 **-XX:InitialTenuringThreshold**、**-XX:MaxTenuringThreshold** 与 **-XX:TargetSurvivorRatio** ，这些标记能帮你实现最佳的年老代区与幸存者区使用方案。

通过设置 **-XX:InitialTenuringThreshold** 与 **-XX:MaxTenuringThreshold**，可以指定年龄的最初值与最大值，而幸存者区 (To) 的使用率则由 **-XX:+NeverTenure** 与 **-XX:+AlwaysTenure** 决定。前者是指永远不将对象存储到年老代区，而后者恰恰相反，总是将对象存储到年老代区。

此处进行的垃圾回收是年老代垃圾回收(Major GC)。当堆空间已满或者年老代区占满时，就会触发 Major GC。此时，通常会由一个“停止一切(Stop-the-World)”事件或线程执行垃圾回收。此外，还有另一种称为全垃圾回收(Full GC)的垃圾回收机制，会涉及诸如永久内存区域。

与整体堆内存相关的另两个重要且有趣的标记是 **-XX:SurvivorRatio** 与 **-XX:NewRatio**，前者指定伊甸园区相对幸存者区的比率，后者指定年老代区相对新生代区的比率。

[Permanent Generation] 永久代区(Permgen space)

永久代区(Permgen)用于存储以下信息：常量池 (内存池)，字段与方法数据及代码。

垃圾回收算法

串行 GC(Serial GC) (-XX:UseSerialGC): 针对年轻代与年老代的垃圾回收

该算法使用简单的“标记-清扫-压缩(mark-sweep-compact)”循环清理年轻代与年老代，适合内存占用较低、CPU 使用量较少的客户端系统。

并行 GC(Parallel GC) (-XX:UseParallelGC): 针对年轻代与年老代的垃圾回收

该算法使用 N 个线程(N 的值可以通过 **-XX:ParallelGCThreads=N** 设定，N 同时代表垃圾回收占用的 CPU 内核数)。其中，年轻代垃圾回收会使用 N 个线程，而年老代只用一个线程。

并行 Old GC (-XX:UseParallelOldGC): 针对年轻代与年老代的垃圾回收

该算法对年轻代与年老代均使用 N 个线程，其他方面与并行 GC 完全一致。

并发 Mark and Sweep GC (-XX:ConcMarkSweepGC): 针对年老代的垃圾回收

顾名思义，CMS GC 会最小化垃圾回收所需的停顿时间。该算法最适于创建高响应度的应用，且只作用于年老代。它会创建多条垃圾回收的线程，与应用线程同时工作。垃圾回收的线程数量可以使用 **-XX:ParallelCMSThreads=n** 标记指定。

G1 GC (-XX:UseG1GC): 针对年轻代与年老代的垃圾回收 (将堆内存等分为大小相同的区块)

这是一种并行、并发、不断压缩的低停顿垃圾回收器。G1 是在 Java 7 中引入以取代 CMS GC 的，它会先将堆内存分为多个大小相等的区块，继而执行垃圾回收。通常，从活动数据最少的区块开始，因此以垃圾为先。

最常见的内存溢出问题

所有 Java 程序员都应该知道的最常见的内存溢出问题：

- **Exception in thread "main": java.lang.OutOfMemoryError:** Java heap space(Java 堆内存)。这并不一定意味着内存泄露，也可能是分配的堆内存空间太小。此外，在运行时间较长的应用中，也可能是因为一个无意识的引用被指向堆对象(内存泄露)。即便是应用本身调用的 APIs，也可能保存着指向无依据对象的引用。而且，在大量使用终结器的应用中，对象们有时可能正排在终结队列中。当这样的应用创建高优先级的线程时，会导致越来越多的对象排在终结队列中，最终导致内存溢出。
- **Exception in thread "main": java.lang.OutOfMemoryError:** PermGen space(永久存储空间)。如果加载了很多类与方法，或者创建了很多字符串常量，特别是使用 intern() 方法进行创建(从 JDK 7 开始，interned 字符串就不再存储在 PermGen 中)，这类错误就会出现。当出现这类错误时，打印的堆栈跟踪附近可能会出现如下文本：ClassLoader.defineClass。

- **Exception in thread "main": java.lang.OutOfMemoryError:** Requested array size exceeds VM limit (请求的数组大小超出 VM 限制)。当请求的数组大小超过可用的堆空间时，这类报错就会出现。这类错误通常归咎于编程错误，在运行时请求了极大的数组大小。
- **Exception in thread "main": java.lang.OutOfMemoryError:** `request <s> bytes for <r>`，交换空间溢出？这是内存泄露最常见的根源。通常，当操作系统没有足够的交换空间，或另一个进程占用了系统中的所有可用内存，就会导致内存泄露。简而言之，由于空间用尽，堆内存无法提供所请求的空间大小。该信息中的 's' 代表失败的请求所需的内存大小(以字节为单位)，而 'r' 代表内存请求的原因。在大多数情况下，此处的 'r' 是报告分配失败的源模块，有时也会是具体的原因。
- **Exception in thread "main": java.lang.OutOfMemoryError:** <reason> <stack trace> (Native method)(<原因><堆栈跟踪><本地方法>)。该报错意味着一个本地方法遇到内存分配失败。问题的根源在于 Java Native Interface(Java 本地接口) 中存在的错误，而非 JVM 中运行的代码错误。若是本地代码不检查内存分配错误，应由会直接崩溃，而不会出现内存溢出。

内存泄露的定义

你可以将内存泄露看做一种疾病，而内存溢出错误为一种征兆。但不是所有的内存溢出错误都意味着内存泄露，不是所有的内存泄露都以内存溢出为征兆。

维基百科的定义：在计算机科学中，内存泄露是一种以如下方式发生的资源泄露——计算机程序错误地分配内存，导致不再需要的内存得不到释放。在面向对象的编程语言中，一个对象若存储在内存中，却无法由运行的代码获取，即为内存泄露。

Java 中常用的内存泄露定义

当不再需要的对象引用仍旧多余地予以保存，即为内存泄露。

在 Java 中，内存泄露是指对象已不再使用，但垃圾回收未能将他们视做不使用对象予以回收。

当程序不再使用某个对象，但在一些无法触及的位置该对象仍旧被引用，即为内存泄露。也因此，垃圾回收器无法删除它。该对象占用的内存空间无法释放，程序所需的总内存就会增加。久而久之，应用的性能就会下降，JVM 可能会耗尽所有内存。

在某种程度上，当无法再给年老区分配内存时，内存泄露就会发生。

内存泄露最常见的一些情况：

- ThreadLocal 变量
- 循环与复杂的双向引用
- JNI 内存泄露
- 可变的静态域(最为常见)

我建议结合使用 Visual VM 与 JDK，对内存泄露问题进行调试。

常见的内存泄露调试方法

1. NetBeans 分析器
2. 使用 jhat Utility
3. 创建 Heap Dump
4. 获取当下运行进程的堆内存柱状图
5. 获取内存溢出错误的堆内存柱状图
6. 监控在等待终结的对象数量
7. 第三方内存调试器

调试内存泄露问题的常用策略或步骤：

- 确认征兆
- 启用详细的垃圾回收机制(verbose GC)
- 启用性能分析
- 分析堆栈跟踪

原文地址：https://dzone.com/articles/java-memory-architecture-model-garbage-collection