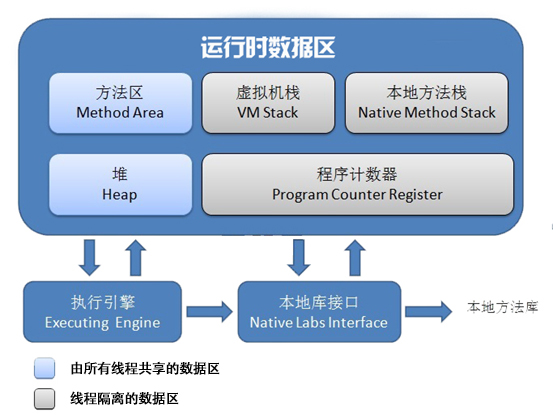
# JVM虚拟机优化

Java HotSpot JVM运行时数据区

## 简介

Java虚拟机在执行Java程序的过程中会把它所管理的内存划分为若干个不同的数据区域，这些区域都有各自的用途，以及创建和销毁时间，有的区域随着虚拟机进程的启动而存在，有些区域则依赖用户线程的启动和结束而建立和销毁。根据《Java虚拟机规范（Java SE7版）》，Java虚拟机所管理的内存将会包括以下几个运行时数据区域：



Java虚拟机运行时数据区

如上图所示，**程序计数器**、**虚拟机栈**和**本地方法栈**是线程自己私有的，而**方法区**和**堆**是所有虚拟机线程之间共享的数据区。

**程序计数器**（Program Counter Register）：是一块较小的内存空间，它可以看做是当前线程执行的字节码的行号指示器，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字节码指令。为了线程切换后能恢复到争取的执行位置，每条线程都需要有一个独立的线程计数器，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址；如果正在执行的是Native方法，这个计数器的值则为空（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情况的区域。

**Java虚拟机栈**（Java Virtual Machine Stacks）：也是线程私有的，它用来描述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 Frame）用于储存局部变量表、操作数栈和方法出口等信息。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中入栈到出栈的过程。局部变量表存放了编译期可知的各种基本数据类型（八个基本数据类型）、对象引用（reference类型）和returnAddress类型（指向了一条字节码指令的地址）。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允许的深度，将抛出StackOverflowError异常；如果虚拟机栈可以动态扩展并且扩展时无法申请到足够的内存，就会抛出OutOfMemoryError异常。

**本地方法栈**（Native Method Stack）：它与虚拟机栈所发挥的作用是非常相似的，他们之间的区别不过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。有的虚拟机（如Sun HotSpot虚拟机）直接把虚拟机栈和本地方法栈合二为一。与虚拟机栈一样，本地方法栈也会抛出StackOverflowError和OutOfMemoryError异常。

**Java堆**（Java Heap）：Java堆是Java虚拟机管所管理的内存中最大的一块，它是被所有线程共享的一块内存区域，在虚拟机启动时创建。此内存区域的唯一目的就是存放对象实例。虚拟机规范中的描述是：所有对象实例以及数组都要在堆上分配，但是随着JIT编译器的发展与逃逸分析技术的逐渐成熟，栈上分配、标量替换优化技术将会导致一些微妙的变化发生，所有的对象都在Java堆上分配渐渐变得不是那么绝对了。Java堆是垃圾回收的主要区域，因此很多时候也被称作“GC堆”。从内存回收的角度看，由于现在收集器基本都采用分代收集算法，所以Java堆中还可以细分为**新生代**和**老年代**；再细致一点的有Eden空间、From Survivor空间、To Survivor空间等。根据Java虚拟机规范的规定，Java堆额可以处于物理上不连续的内存空间中。在实现时，既可以实现成固定大小的，也可以是扩展的（通过-Xmx和-Xms控制）。当堆再也无法扩展时，将抛出OutOfMemoryError异常。

**方法区**（Method Area）：和Java堆一样，是各个线程共享的内存区域。它用于储存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。Java虚拟机规范对方法去的限制非常宽松，除了和Java堆一样不需要连续的内存和可以选择固定大小或者可扩展外，还可以选择不实现垃圾收集。当方法区无法满足内存分配需求时，将抛出OutOfMemoryError异常。运行时常量池（Runtime Constant Pool）是方法区的一部分。

经常有人把Java内存区分为堆内存（Heap）和栈内存（Stack），其实Java内存区域的划分远比这复杂。这种划分方式的流行只能说明大多数程序员最关注的的、与对象内存分配关系最密切的内存区域是这两块。其中的“堆”是指**Java堆**，而其中的“栈”指的是**Java虚拟机栈**。

**直接内存**（DirectMemory）并不是虚拟机运行时数据区的一部分，也不是Java虚拟机规范中定义的内存区域。这部分区域也会抛出OutOfMemoryError异常。在JDK1.4中加入了java.nio类，引入了一种基于通道（Channel）与缓冲区（Buffer）的I/0方式，它可以使用Native函数库直接分配堆外内存，然后存储在Java堆中的DirectByteBuffer对象作为这块内存的引用进行操作。这样能在一些场景中显著提高性能，因为避免了在Java堆和Native堆中来回复制数据。

## 垃圾收集算法

判断对象是“存活”还是“死去”，有**引用计数算法**和**可达性分析算法**。但主流Java虚拟机没有选用引用计数法来管理内存，其主要原因是它很难解决对象之间相互循环引用的问题。

引用计数算法：给对象添加一个引用计数器，每当有一个地方引用它，计数器就加1；当引用失效，计数器值就减1；任何时刻计数器为0的对象就是不可能再被使用的。

可达性分析算法：通过一些列的成为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径成为引用链（Reference China），当一个对象到GC Roots没有任何引用链相连时，则证明此对象是不可用的。

**垃圾收集算法在JVM中主要是复制算法（新生代GC）和标记/整理算法（老年代GC）。**

垃圾收集算法有如下几种：

**【标记-清除（Mark-Sweep）算法】**

算法过程：

1. 先判定对象是否可回收，对其标记。  
2. 统一回收（简单地删除对垃圾对象的内存引用）。

优点：简单直观容易实现和理解。缺点：效率不高，内存空间碎片化。

**【标记-整理（Mark-Compact）】**

算法过程：

1. 标记：标记可回收对象（垃圾对象）和存活对象。

2. 紧凑（也称“整理”）：将所有存活对象向内存开始部位移动，称为内存紧凑（相当于碎片整理）。完毕后，清理剩余内存空间。

**【复制（Copying）算法】**

将内存平均分成A、B两块，算法过程：

1. 新生对象被分配到A块中未使用的内存当中。当A块的内存用完了， 把A块的存活对象对象复制到B块。

2. 清理A块所有对象。

3. 新生对象被分配的B块中未使用的内存当中。当B块的内存用完了， 把B块的存活对象对象复制到A块。

4. 清理B块所有对象。

5. goto 1。

优点：简单高效。缺点：内存代价高，有效内存为占用内存的一半。

对复制算法进一步优化：使用Eden/S0/S1三个分区

平均分成A/B块太浪费内存，采用Eden/S0/S1三个区更合理，空间比例为Eden:S0:S1==8:1:1，有效内存（即可分配新生对象的内存）是总内存的9/10。

算法过程：

1. Eden+S0可分配新生对象；

2.对Eden+S0进行垃圾收集，存活对象复制到S1。清理Eden+S0。一次新生代GC结束。

3. Eden+S1可分配新生对象；

4.对Eden+S1进行垃圾收集，存活对象复制到S0。清理Eden+S1。二次新生代GC结束。

5.goto 1。

**【分代收集算法】**

由于不同的对象适合使用不同的垃圾收集算法，所以引入“代”这个概念。不同的代有不同的分区，一般分为新生代区和老年代区。

**新生代：**适合采用【复制】算法进行垃圾收集，对象分布在Eden/S0/S1三个区。

**老年代：**适合采用【标记-整理】算法进行垃圾收集。

**Heap（堆）分区的目的**

1. 为了分代：不同代的对象放到不同的内存分区中，实现“代提升”，也方便实现对不同分代采用不同的垃圾收集算法。

2. 垃圾收集算法需要：新生代GC使用到复制算法，该算法需要将对应的分区划分成三个分区：Eden/S0/S1。

**术语**

Generation代

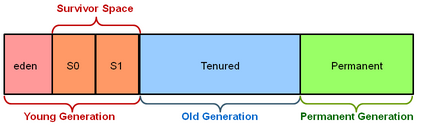
 -- YongGeneration/NewGeneration：新生代，在Eden/S0/S1的存活的对象。

 -- OldGeneration：老年代，在Tenured区存活的对象。  
 -- PermanentGeneration：永久代。  
Space 区

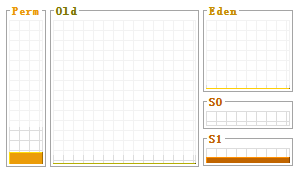
 -- Eden：伊甸园区，是新生代的一个区。  
 -- Survivor：幸存区，属于新生代，为了复制算法的需要。一般分成大小相等的两个区（S0/S1或者From/To）。

 -- Tenured：存放老年代的区域。  
 -- Permanent：终身区。

下图：Hotspot 的 Heap 分区



下图：VisualVM 中通过 VisualGC插件显示的分区



**Eden/S0/S1 新生代**

S0/S1是大小相当的两个区域，共同组成Survivor区。

空间比例：Eden:S0==8:1。设定方法：-XX:SurvivorRatio=8。

新生对象在Eden/S0或者Eden/S1中分配，Eden区的对象量达到一个阈值后，发生一次新生代GC。

**Old 老年代**

每个对象有“对象年龄计数器”。对象由Eden收集到Survivor区后，年龄+1。进行新生代GC后，年龄+1。依次，当年龄>=15后进入老年代。

最大年龄阈值设定：-XX:MaxTenuringThreshold。

动态年龄：如果在Survivor中所有相同年龄对象占用了空间的一半多，大于等于上述年龄的对象直接进入老年代。

大对象（比如大的数组）直接进入老年代。阈值设定：-XX:PretenureSizeThreshold。

**Perm 永久代（PermanentGeneration）**

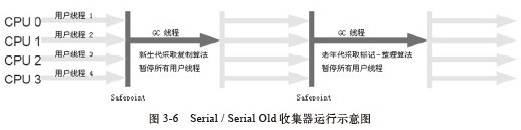
用于存放不变对象，如类、方法、字符串等。

Java7把驻留字符串（intentd string）放到了老年代区。Java8中移除了Hotspot的永久代区。

## 垃圾收集器

JVM给了三种垃圾收集器：**单线程收集器、并行收集器 和 并发收集器。**

**单线程收集器（Serial GC）：**使用复制收集算法，收集时会暂停所有工作线程(我们将这件事情称之为Stop The World)，直到收集结束，默认情况下，JDK5.0以前都是使用串行收集器，如果想使用其他收集器需要在启动时加入相应参数。JDK5.0以后，JVM会根据当前[系统配置](http://java.sun.com/j2se/1.5.0/docs/guide/vm/server-class.html)进行判断。 优点是：简单高效（与其他收集器的单线程相比），对于限定单个CPU的环境来说，Serial收集器没有现成交互的开销。在堆比较小的情况下，一般停顿时 间很短，是可以使用这种收集器的。如下图：



使用方法：

java -Xmx12m -Xms3m -Xmn1m -XX:PermSize=20m -XX:MaxPermSize=20m **-XX:+UseSerialGC** -jar c:\javademos\demo\jfc\Java2D\Java2demo.jar

## JVM参数调优

| **Switch** | **Description** |
| --- | --- |
| -Xms | Sets the initial heap size for when the JVM starts. |
| -Xmx | Sets the maximum heap size. |
| -Xmn | Sets the size of the Young Generation. |
| -XX:PermSize | Sets the starting size of the Permanent Generation. |
| -XX:MaxPermSize | Sets the maximum size of the Permanent Generation |

1. 虚拟机提供了一个 **–XX:PretenureSizeThreshold**参数，令大于这个设置值的对象直接在老年代分配。这样做的目的是避免在Eden区及两个Survivor区之间发生大量的内存复制。你应该知道，对于JVM新生代采用复制算法收集内存时，它最怕遇到用户新建一个“朝生夕灭”的短命大对象，更可怕的是用户建立了一堆这样的短命大对象。但注意，该参数值对Serial和ParNew两款收集器有效。
2. **–XX:MaxTenuringThreshold**用来设置新生代对象的年龄计数器达到多少可以晋升到老年代对象，但这并不表示只有对象的年龄达到这个值才能晋升成老年代对象。
3. 在JKD1.5中java.lang.Thread类新增了一个getAllStackTraces()方法用于获取虚拟机中所有的StackTraceElement对象，使用这个方法可以通过简单的几行代码就完成了jstack的大部分功能。在实际项目中可以用这个方法做个管理员界面。
4. JVM调优目的