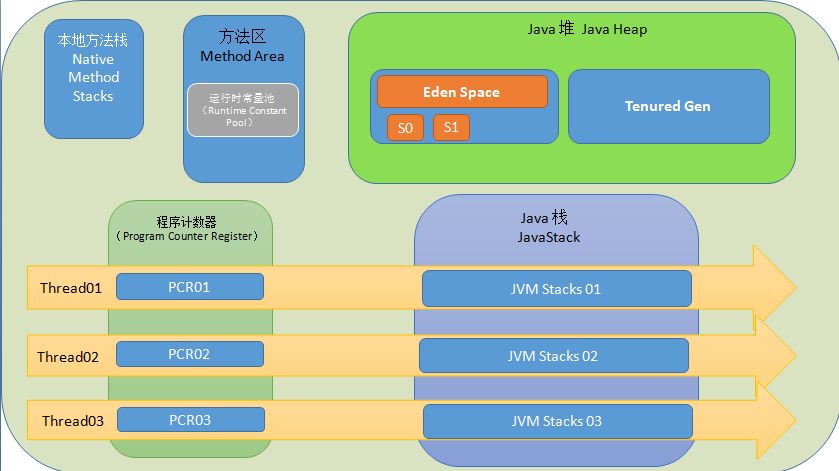
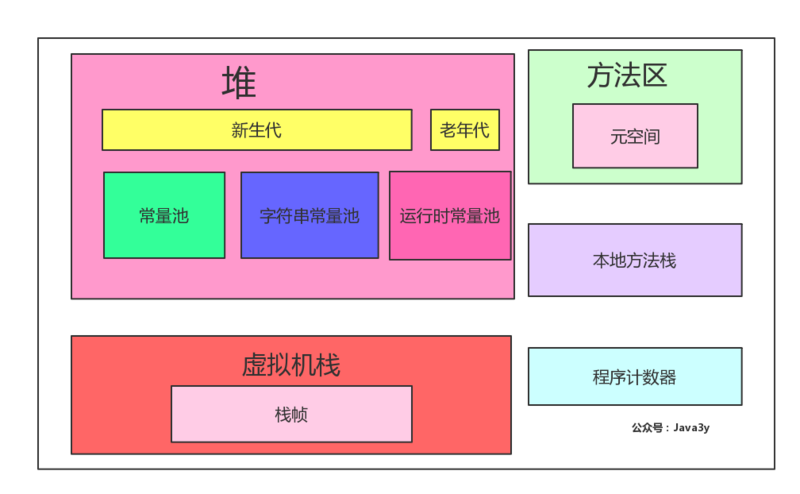
## JVM内存模型

首先我们来了解一下JVM的内存模型的怎么样的：





**1.堆：存放对象实例，几乎所有的对象实例都在这里分配内存**

* 堆得内存由-Xms指定，默认是物理内存的1/64；最大的内存由-Xmx指定，默认是物理内存的1/4。
* 默认空余的堆内存小于40%时，就会增大，直到-Xmx设置的内存。具体的比例可以由-XX:MinHeapFreeRatio指定
* 空余的内存大于70%时，就会减少内存，直到-Xms设置的大小。具体由-XX:MaxHeapFreeRatio指定。

**2.虚拟机栈**

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时创建一个栈帧（Stack Frame）用于存储局部变量表、操作栈、动态链接、方法出口等信息本地方法栈：本地方法栈则是为虚拟机使用到的Native方法服务。

**3.方法区：存储已被虚拟机加载的类元数据信息(元空间)**

1）有时候也成为永久代，在该区内很少发生垃圾回收，但是并不代表不发生GC，在这里进行的GC主要是对方法区里的常量池和对类型的卸载  
2）方法区主要用来存储已被虚拟机加载的类的信息、常量、静态变量和即时编译器编译后的代码等数据。  
该区域是被线程共享的。  
3）方法区里有一个运行时常量池，用于存放静态编译产生的字面量和符号引用。该常量池具有动态性，也就是说常量并不一定是编译时确定，运行时生成的常量也会存在这个常量池中。

**4.程序计数器**：当前线程所执行的字节码的行号指示器

**总结：**



## JVM垃圾回收算法

**1.标记-清除:** 这是垃圾收集算法中最基础的，根据名字就可以知道，它的思想就是标记哪些要被回收的对象，然后统一回收。这种方法很简单，但是会有两个主要问题：1.效率不高，标记和清除的效率都很低；2.会产生大量不连续的内存碎片，导致以后程序在分配较大的对象时，由于没有充足的连续内存而提前触发一次GC动作。

**2.复制算法**: 为了解决效率问题，复制算法将可用内存按容量划分为相等的两部分，然后每次只使用其中的一块，当一块内存用完时，就将还存活的对象复制到第二块内存上，然后一次性清楚完第一块内存，再将第二块上的对象复制到第一块。但是这种方式，内存的代价太高，每次基本上都要浪费一般的内存。 于是将该算法进行了改进，内存区域不再是按照1：1去划分，而是将内存划分为8:1:1三部分，较大那份内存交Eden区，其余是两块较小的内存区叫Survior区。每次都会优先使用Eden区，若Eden区满，就将对象复制到第二块内存区上，然后清除Eden区，如果此时存活的对象太多，以至于Survivor不够时，会将这些对象通过分配担保机制复制到老年代中。(java堆又分为新生代和老年代)

**3. 标记-整理** 该算法主要是为了解决标记-清除，产生大量内存碎片的问题；当对象存活率较高时，也解决了复制算法的效率问题。它的不同之处就是在清除对象的时候现将可回收对象移动到一端，然后清除掉端边界以外的对象，这样就不会产生内存碎片了。

**4.分代收集** 现在的虚拟机垃圾收集大多采用这种方式，它根据对象的生存周期，将堆分为新生代和老年代。在新生代中，由于对象生存期短，每次回收都会有大量对象死去，那么这时就采用复制算法。老年代里的对象存活率较高，没有额外的空间进行分配担保，所以可以使用标记-整理 或者 标记-清除。

## JVM垃圾收集器有哪些？以及优劣势比较？

**1.串行收集器**

串行收集器是最简单的，它设计为在单核的环境下工作（32位或者windows），你几乎不会使用到它。它在工作的时候会暂停整个应用的运行，因此在所有服务器环境下都不可能被使用。

使用方法：-XX:+UseSerialGC

**2.并行收集器**

这是JVM默认的收集器，跟它名字显示的一样，它最大的优点是使用多个线程来扫描和压缩堆。缺点是在minor和full GC的时候都会暂停应用的运行。并行收集器最适合用在可以容忍程序停滞的环境使用，它占用较低的CPU因而能提高应用的吞吐（throughput）。

使用方法：-XX:+UseParallelGC

**3.CMS收集器**

CMS是**Concurrent-Mark-Sweep**的缩写，并发的标记与清除。

这个算法使用多个线程并发地（concurrent）扫描堆，标记不使用的对象，然后清除它们回收内存。在两种情况下会使应用暂停（Stop the World, STW）：

1. 当初次开始标记根对象时initial mark。

2. 当在并行收集时应用又改变了堆的状态时，需要它从头再确认一次标记了正确的对象final remark。

这个收集器最大的问题是在年轻代与老年代收集时会出现的一种竞争情况（race condition），称为提升失败promotion failure。对象从年轻代复制到老年代称为提升promotion，但有时侯老年代需要清理出足够空间来放这些对象，这需要一定的时间，它收集的速度可能赶不上不断产生的要提升的年轻代对象的速度，这时就需要做STW的收集。STW正是CMS想避免的问题。为了避免这个问题，需要增加老年代的空间大小或者增加更多的线程来做老年代的收集以赶上从年轻代复制对象的速度。

除了上文所说的内容之外，CMS最大的问题就是内存空间碎片化的问题。CMS只有在触发FullGC的情况下才会对堆空间进行compact。如果线上应用长时间运行，碎片化会非常严重，会很容易造成promotion failed。为了解决这个问题线上很多应用通过定期重启或者手工触发FullGC来触发碎片整理。

对比并行收集器它的一个坏处是需要占用比较多的CPU。对于大多数长期运行的服务器应用来说，这通常是值得的，因为它不会导致应用长时间的停滞。但是它不是JVM的默认的收集器。

**4.G1收集器**

如果你的堆内存大于4G的话，那么G1会是要考虑使用的收集器。它是为了更好支持大于4G堆内存在JDK 7 u4引入的。G1收集器把堆分成多个区域，大小从1MB到32MB，并使用多个后台线程来扫描这些区域，优先会扫描最多垃圾的区域，这就是它名称的由来，垃圾优先Garbage First。

如果在后台线程完成扫描之前堆空间耗光的话，才会进行STW收集。它另外一个优点是它在处理的同时会整理压缩堆空间，相比CMS只会在完全STW收集的时候才会这么做。

使用过大的堆内存在过去几年是存在争议的，很多开发者从单个JVM分解成使用多个JVM的微服务（micro-service）和基于组件的架构。其他一些因素像分离程序组件、简化部署和避免重新加载类到内存的考虑也促进了这样的分离。

除了这些因素，最大的因素当然是避免在STW收集时JVM用户线程停滞时间过长，如果你使用了很大的堆内存的话就可能出现这种情况。另外，像Docker那样的容器技术让你可以在一台物理机器上轻松部署多个应用也加速了这种趋势。

使用方法：－XX:+UseG1GC