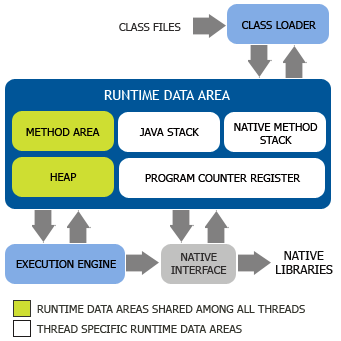
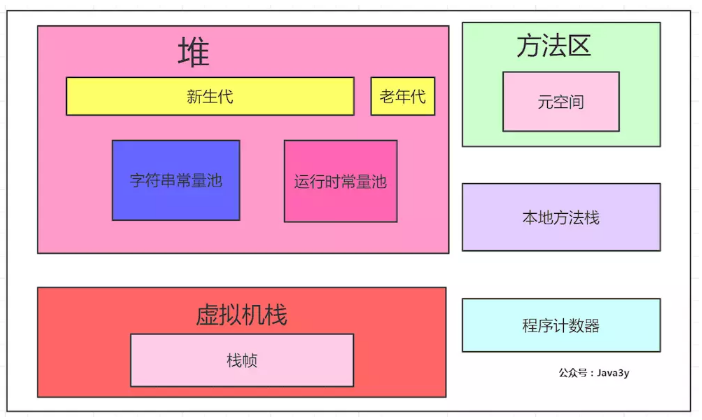
### JVM内存布局图

**以下所说的虚拟机指的都是hotspot虚拟机**





# 堆与方法区

### 1.Java堆的描述

（1）Java堆（Java heap）是Java虚拟机所管理的内存中最大的一块

（2）Java堆被所有线程共享的一块内存区域

（3）虚拟机启动时创建Java堆

（4）Java堆的**唯一目的就是存放对象实例**

（5）Java堆是垃圾收集器管理的主要区域

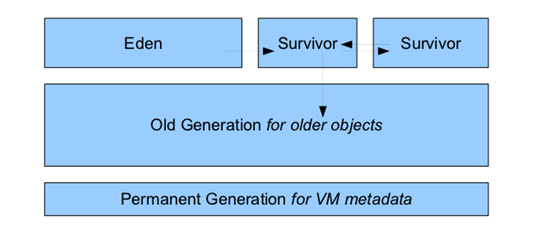
（6）从内存回收的角度看，由于现在收集器都采用分代收集算法，所以Java堆可以细分为新生代和老年代。新生代又被划分为三个区域Eden、from survivor、to survivor。但存储的都是实例对象，进一步划分的目的是为了更好的内存回收，或者更快的内存分配

（7）Java堆的大小是可扩展的，通过-Xms（初始堆大小）和-Xmx（最大堆大小）控制

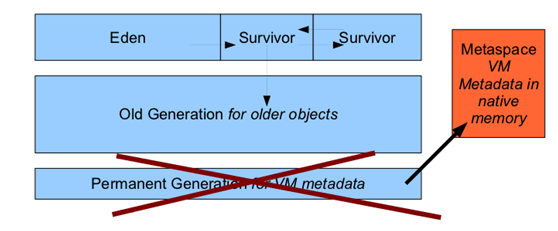
（8）如果堆内存不够分配实例对象，并且也无法扩展时，将会抛出outOfMemoryError异常

（9）堆大小=新生代+老年代。**默认新生代占堆空间的1/3，老年代占堆空间的2/3**。

### 2. JDK7以及前期的版本中，堆内存通常划分为新生代、老年代、永生代



### 2. JDK8废除了永生代并用Metaspace（元空间）替代



### 3.新生代

新生代几乎是所有Java对象出生的地方，即Java对象申请的内存以及存放都是在这个地方。Java 大部分的对象通常不会长久的存活，既有**朝生夕死**的特点。**新生代是收集垃圾的频繁区域。**

**Minor GC是发生在新生代中的垃圾收集动作，采用的是复制算法。**

新生代被细分为Eden 和两个survivor区域，两个survivor区域分别被命名为from 和to ，默认空间比例为8：1：1

JVM每次指挥使用Eden和其中的一块survivor区域来为对象服务，所以无论什么是哈韩，总有一块survivor区域是空闲的

新生代实际可用的内存空间为9/10的新生代空间

### 4.老年代

当对象在Eden（包括一个survivor区域，假设是from区域）出生后，在经过一次minor GC后，如果对象还存活，并且能够被另外一块survivor区域所容纳（这里应为to区域，即to区域有足够的空间来存储Eden和from区域中存活的对象），则使用复制算法将这些仍存活的对象复制到另一块survivor（即to区域），然后清理所使用过的Eden以及survivor（即from）区域，并且将复制到另一块survivor区域的对象年龄设为1。

此后对象在survivor区每熬过一次minor FC，就将对象的年龄+1，当对象的年龄达到某个值时，默认是 15 岁，可以通过参数 -XX:MaxTenuringThreshold 来设定 )，这些对象就会成为老年代。

### 5.Full GC

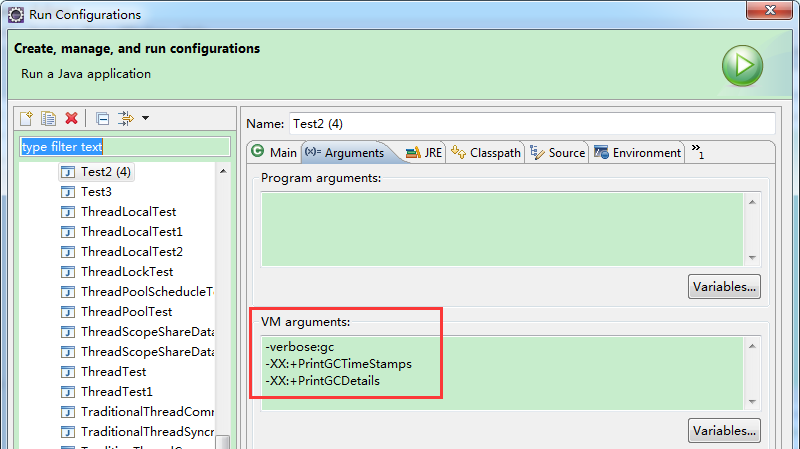
Full GC基本是整个堆空间发生了垃圾回收，采用的是**标记-清除**算法

老年代里面的对象几乎个个都是在 Survivor 区域中熬过来的，它们是不会那么容易就 “死掉” 了的。因此，**Full GC 发生的次数不会有 Minor GC 那么频繁**，并且做一次 Full GC 要比进行一次 Minor GC 的时间更长，一般是Minor GC的 10倍以上。

另外，标记-清除算法收集垃圾的时候会产生许多的内存碎片 ( 即不连续的内存空间 )，此后需要为较大的对象分配内存空间时，若无法找到足够的连续的内存空间，就会提前触发一次 GC 的收集动作

### 6.分析GC日志

在Run as - Run Configuration中配置参数,使得控制台能够显示 GC 相关的日志信息



-verbose:gc

-XX:+PrintGCTimeStamps

-XX:+PrintGCDetails

### 7.从永久代到元空间

**包括JDK7及以前的版本，永久代就相当于方法区（非堆），原本是在堆中。**

**JDK8及以后，用元空间替代了永久代，并且不再存在于堆中，而是移动到叫做MetaSpace的本地内存中，具体大小取决于系统的可用内存，需要配置参数。同样也相当于方法区。**

**元空间的本质和永久代类似，都是对JVM规范中方法区的实现。不过元空间与永久代之间最大的区别在于：元空间并不在虚拟机中，而是使用本地内存。**

**具体过程：**

**1.JDK7把字符串常量池移到了堆中**

**2.永久代被永久移除，取而代之以元空间**

### 8.虚拟机栈

每个线程有一个私有的栈，随着线程的创建而创建。栈里面存着的是一种叫栈帧的东西，每个方法会创建一个栈帧。栈帧中存放了局部变量表（基本数据类型和对象引用）、操作数栈、方法出口等信息。当栈调用深度大于JVM所允许的范围，会抛出StackOverflowError的错误，不过这个深度范围不是一个恒定的值

### 9.本地方法栈

这部分主要与虚拟机用到的native方法相关

### 10.程序计数器寄存器

JVM支持多个线程同时运行，每个线程都有自己的程序计数器。倘若当前执行的是 JVM 的方法，则该寄存器中保存当前执行指令的地址；倘若执行的是native 方法，则PC寄存器中为空

**原因：**

**1.为了和JRockit进行融合，Jrockit用户不需要配置永久代（因为Jrockit没有永久代）**

**2.简化Full GC，永久代会为GC带来不必要的复杂度，并且回收效率偏低**

**3.类与方法的信息等比较难确定大小，所以对于永久代的大小指定比较困难，太小容易出现永久代溢出**

### JVM参数选项（部分不可用）



可以用**-XX:MetaspaceSize=8m -XX:MaxMetaspaceSize=128m** 设置。如果你设置的元空间过小，你的应用程序可能得到下面错误：

java.lang.OutOfMemoryError: Metadata space