JVM面试题

# Minor GC和FULLGC的区别

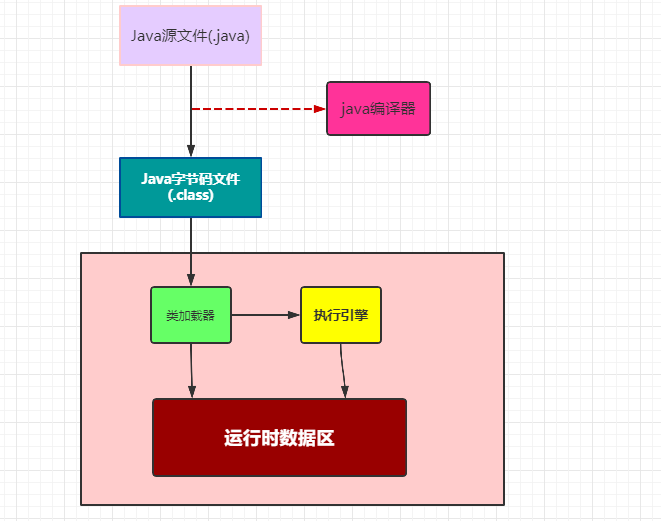
Minor GC是发生在新生代中的垃圾回收，大多数Java对象都具有朝生夕灭的特点，所以Minor GC非常频繁，回收速度也比较快。

FULL GC是发生在老年代的GC，速度一般比Minor GC慢10倍以上。

# JVM

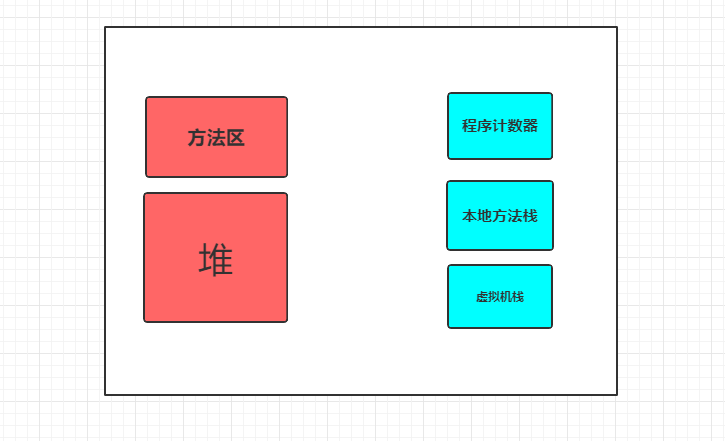
<https://juejin.im/post/5ea8b6d9f265da7be959ef5c>

## JVM的内存机制



如上图所示，首先Java源代码文件(.java后缀)会被Java编译器编译为字节码文件(.class后缀)，然后由JVM中的类加载器加载各个类的字节码文件，加载完毕之后，交由JVM执行引擎执行。在整个程序执行过程中，JVM会用一段空间来存储程序执行期间需要用到的数据和相关信息，这段空间一般被称作为Runtime Data Area（运行时数据区），也就是我们常说的JVM内存。因此，在Java中我们常常说到的内存管理就是针对这段空间进行管理（如何分配和回收内存空间）。

### 运行时数据区包括哪几部分



#### 程序计数器

程序计数器（Program Counter Register），也有称作为PC寄存器。想必学过汇编语言的朋友对程序计数器这个概念并不陌生，在汇编语言中，程序计数器是指CPU中的寄存器，它保存的是程序当前执行的指令的地址（也可以说保存下一条指令的所在存储单元的地址），当CPU需要执行指令时，需要从程序计数器中得到当前需要执行的指令所在存储单元的地址，然后根据得到的地址获取到指令，在得到指令之后，程序计数器便自动加1或者根据转移指针得到下一条指令的地址，如此循环，直至执行完所有的指令。

　　虽然JVM中的程序计数器并不像汇编语言中的程序计数器一样是物理概念上的CPU寄存器，但是JVM中的程序计数器的功能跟汇编语言中的程序计数器的功能在逻辑上是等同的，也就是说是用来指示 执行哪条指令的。

　　由于在JVM中，多线程是通过线程轮流切换来获得CPU执行时间的，因此，在任一具体时刻，一个CPU的内核只会执行一条线程中的指令，因此，为了能够使得每个线程都在线程切换后能够恢复在切换之前的程序执行位置，每个线程都需要有自己独立的程序计数器，并且不能互相被干扰，否则就会影响到程序的正常执行次序。因此，可以这么说，程序计数器是每个线程所私有的。

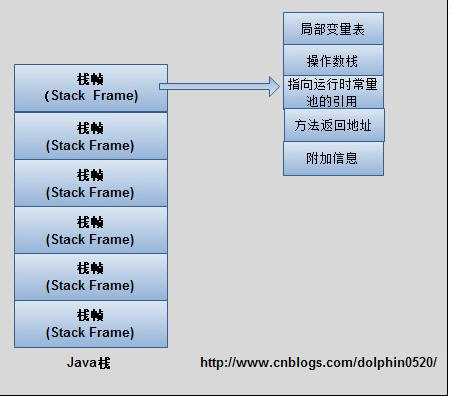
　　在JVM规范中规定，如果线程执行的是非native方法，则程序计数器中保存的是当前需要执行的指令的地址；如果线程执行的是native方法，则程序计数器中的值是undefined。

　　由于程序计数器中存储的数据所占空间的大小不会随程序的执行而发生改变，因此，对于程序计数器是不会发生内存溢出现象(OutOfMemory)的。

#### Java栈

　Java栈也称作虚拟机栈（Java Vitual Machine Stack），也就是我们常常所说的栈，跟C语言的数据段中的栈类似。事实上，Java栈是Java方法执行的内存模型。为什么这么说呢？下面就来解释一下其中的原因。

　　Java栈中存放的是一个个的栈帧，每个栈帧对应一个被调用的方法，在栈帧中包括局部变量表(Local Variables)、操作数栈(Operand Stack)、指向当前方法所属的类的运行时常量池（运行时常量池的概念在方法区部分会谈到）的引用(Reference to runtime constant pool)、方法返回地址(Return Address)和一些额外的附加信息。当线程执行一个方法时，就会随之创建一个对应的栈帧，并将建立的栈帧压栈。当方法执行完毕之后，便会将栈帧出栈。因此可知，线程当前执行的方法所对应的栈帧必定位于Java栈的顶部。讲到这里，大家就应该会明白为什么 在 使用 递归方法的时候容易导致栈内存溢出的现象了以及为什么栈区的空间不用程序员去管理了（当然在Java中，程序员基本不用关系到内存分配和释放的事情，因为Java有自己的垃圾回收机制），这部分空间的分配和释放都是由系统自动实施的。对于所有的程序设计语言来说，栈这部分空间对程序员来说是不透明的。下图表示了一个Java栈的模型：



局部变量表，顾名思义，想必不用解释大家应该明白它的作用了吧。就是用来存储方法中的局部变量（包括在方法中声明的非静态变量以及函数形参）。对于基本数据类型的变量，则直接存储它的值，对于引用类型的变量，则存的是指向对象的引用。局部变量表的大小在编译器就可以确定其大小了，因此在程序执行期间局部变量表的大小是不会改变的。

　　操作数栈，想必学过数据结构中的栈的朋友想必对表达式求值问题不会陌生，栈最典型的一个应用就是用来对表达式求值。想想一个线程执行方法的过程中，实际上就是不断执行语句的过程，而归根到底就是进行计算的过程。因此可以这么说，程序中的所有计算过程都是在借助于操作数栈来完成的。

　　指向运行时常量池的引用，因为在方法执行的过程中有可能需要用到类中的常量，所以必须要有一个引用指向运行时常量。

　　方法返回地址，当一个方法执行完毕之后，要返回之前调用它的地方，因此在栈帧中必须保存一个方法返回地址。

　　由于每个线程正在执行的方法可能不同，因此每个线程都会有一个自己的Java栈，互不干扰。

#### 本地方法栈

本地方法栈与Java栈的作用和原理非常相似。区别只不过是Java栈是为执行Java方法服务的，而本地方法栈则是为执行本地方法（Native Method）服务的。在JVM规范中，并没有对本地方发展的具体实现方法以及数据结构作强制规定，虚拟机可以自由实现它。在HotSopt虚拟机中直接就把本地方法栈和Java栈合二为一。

#### 堆

　在C语言中，堆这部分空间是唯一一个程序员可以管理的内存区域。程序员可以通过malloc函数和free函数在堆上申请和释放空间。那么在Java中是怎么样的呢？

　　Java中的堆是用来存储对象本身的以及数组（当然，数组引用是存放在Java栈中的）。只不过和C语言中的不同，在Java中，程序员基本不用去关心空间释放的问题，Java的垃圾回收机制会自动进行处理。因此这部分空间也是Java垃圾收集器管理的主要区域。另外，堆是被所有线程共享的，在JVM中只有一个堆。

#### 方法区

　方法区在JVM中也是一个非常重要的区域，它与堆一样，是被线程共享的区域。在方法区中，存储了每个类的信息（包括类的名称、方法信息、字段信息）、静态变量、常量以及编译器编译后的代码等。

　　在Class文件中除了类的字段、方法、接口等描述信息外，还有一项信息是常量池，用来存储编译期间生成的字面量和符号引用。

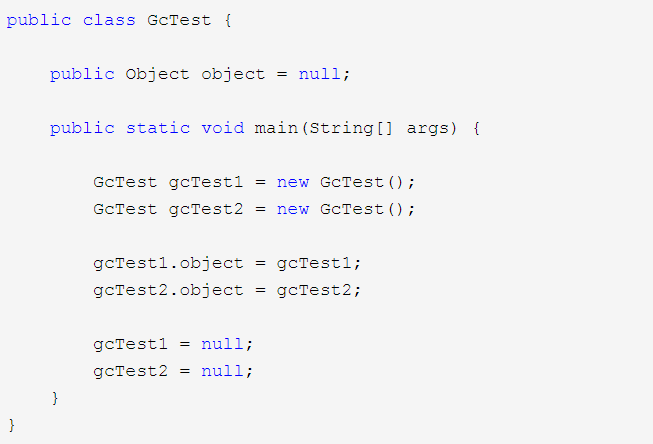
　　在方法区中有一个非常重要的部分就是运行时常量池，它是每一个类或接口的常量池的运行时表示形式，在类和接口被加载到JVM后，对应的运行时常量池就被创建出来。当然并非Class文件常量池中的内容才能进入运行时常量池，在运行期间也可将新的常量放入运行时常量池中，比如String的intern方法。

　　在JVM规范中，没有强制要求方法区必须实现垃圾回收。很多人习惯将方法区称为“永久代”，是因为HotSpot虚拟机以永久代来实现方法区，从而JVM的垃圾收集器可以像管理堆区一样管理这部分区域，从而不需要专门为这部分设计垃圾回收机制。不过自从JDK7之后，Hotspot虚拟机便将运行时常量池从永久代移除了。

## 垃圾回收机制

在java中是通过引用来和对象进行关联的，也就是说如果要操作对象，必须通过引用来进行。那么很显然一个简单的办法就是通过引用计数来判断一个对象是否可以被回收。不失一般性，如果一个对象没有任何引用与之关联，则说明该对象基本不太可能在其他地方被使用到，那么这个对象就成为可被回收的对象了。这种方式成为引用计数法。

　　这样的方法简单粗暴，而且效率很高。效率高必然会暴露一些问题，如果某些对象呗循环引用，即使你把对象赋值为null，这种算法照样不能回收。看下下面的代码



虽然gcTest1，gcTest2是null，他们指向的对象已经不会被访问到了，但是由于它们互相引用对方，导致它们的引用计数都不为0，那么垃圾收集器就永远不会回收它们。

　　上面的问题已经暴露出来了，下面看看jvm是怎么解决这个问题的。为了解决这个问题，在Java中采取了可达性分析法。该方法的基本思想是通过一系列的“GC Roots”对象作为起点进行搜索，如果在“GC Roots”和一个对象之间没有可达路径，则称该对象是不可达的，不过要注意的是被判定为不可达的对象不一定就会成为可回收对象。被判定为不可达的对象要成为可回收对象必须至少经历两次标记过程，如果在这两次标记过程中仍然没有逃脱成为可回收对象的可能性，则基本上就真的成为可回收对象了。在《深入理解jvm》讲解的很仔细，笔者就简单介绍下GC Roots的概念，想深入了解的可以去读下笔者介绍的这本书。

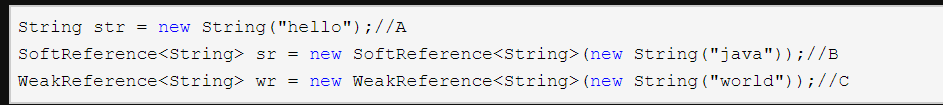
　　以下三类对象在jvm中作为GC roots，来判断一个对象是否可以被回收 (通常来说我们只要知道虚拟机栈和静态引用就够了)

 　　1、虚拟机栈(JVM stack)中引用的对象(准确的说是虚拟机栈中的栈帧(frames)) 。我们知道，每个方法执行的时候，jvm都会创建一个相应的栈帧(栈帧中包括操作数栈、局部变量表、运行时常量池的引用)，栈帧中包含这在方法内部使用的所有对象的引用(当然还有其他的基本类型数据)，当方法执行完后，该栈帧会从虚拟机栈中弹出，这样一来，临时创建的对象的引用也就不存在了，或者说没有任何gc roots指向这些临时对象，这些对象在下一次GC时便会被回收掉

　　2、方法区中类静态属性引用的对象 。静态属性是该类型(class)的属性，不单独属于任何实例，因此该属性自然会作为gc roots。只要这个class存在，该引用指向的对象也会一直存在。class 也是会被回收的，在面后说明

 　　3、本地方法栈(Native Stack)引用的对象

 　　下面介绍下关于软引用（softReference）和弱引用（weakReference）的对象垃圾回收对他们做的处理



上面的几个对象中回收情况如下，B在内存不足的情况下会将String对象判定为可回收对象，C无论什么情况下String对象都会被判定为可回收对象。也就是说软引用会在内存溢出（OOM）的时候回收，而弱引用无论什么情况都会在下一轮回收的时候回收掉。

　　一般jvm会对这些对象回收

　　1、显示地将某个引用赋值为null或者将已经指向某个对象的引用指向新的对象。

　　2、局部引用所指向的对象。

　　3、上面说的弱引用（weakReference）。

### 垃圾收集算法

#### Mark-Sweep（标记-清除）算法

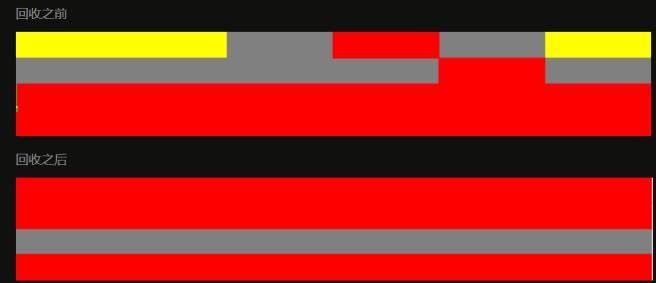
这是最基础的垃圾回收算法，之所以说它是最基础的是因为它最容易实现，思想也是最简单的。标记-清除算法分为两个阶段：标记阶段和清除阶段。标记阶段的任务是标记出所有需要被回收的对象，清除阶段就是回收被标记的对象所占用的空间。图解来自网络，很好的说明了标记-清楚算法的处理前和处理后的内存分布。



很容易看出这样的操作是有弊端的，这样讲标记的对象的清楚后，内存块就变的零零散散，如果现在有一个对象占用的内存很大，这个时候必须要在执行一遍垃圾回收，为这个大的对象腾出空间。

#### Copying（复制）算法

为了解决Mark-Sweep算法的缺陷，Copying算法就被提了出来。它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用的内存空间一次清理掉，这样一来就不容易出现内存碎片的问题。



复制算法会提前空出一般的内存，在垃圾回收的时候将存活的对象移动的另外一半内存，这样内存的移动消耗太大，虽然内存不是零散的，但是代价太高。

#### Mark-Compact（标记-整理）算法

为了解决Copying算法的缺陷，充分利用内存空间，提出了Mark-Compact算法。该算法标记阶段和Mark-Sweep一样，但是在完成标记之后，它不是直接清理可回收对象，而是将存活对象都向一端移动，然后清理掉端边界以外的内存。具体过程如下图所示：



#### Generational Collection（分代收集）算法

　分代收集算法是目前大部分JVM的垃圾收集器采用的算法。它的核心思想是根据对象存活的生命周期将内存划分为若干个不同的区域。一般情况下将堆区划分为老年代（Tenured Generation）和新生代（Young Generation），老年代的特点是每次垃圾收集时只有少量对象需要被回收，并不是回收所有，而新生代的特点是每次垃圾回收时都有大量的对象需要被回收，那么就可以根据不同代的特点采取最适合的收集算法。可以调用System.gc()方法查看回收情况。

　　目前大部分垃圾收集器对于新生代都采取Copying算法，因为新生代中每次垃圾回收都要回收大部分对象，也就是说需要复制的操作次数较少，但是实际中并不是按照1：1的比例来划分新生代的空间的，一般来说是将新生代划分为一块较大的Eden空间和两块较小的Survivor空间，每次使用Eden空间和其中的一块Survivor空间，当进行回收时，将Eden和Survivor中还存活的对象复制到另一块Survivor空间中，然后清理掉Eden和刚才使用过的Survivor空间。

　　而由于老年代的特点是每次回收都只回收少量对象，一般使用的是Mark-Compact算法。

　　注意，在堆区之外还有一个代就是永久代（Permanet Generation），它用来存储class类、常量、方法描述等。对永久代的回收主要回收两部分内容：废弃常量和无用的类。

### 典型的垃圾收集器

　　下面都是些概率性的东西，笔者看得也似懂非懂，直接搬过来分享给大家

## **1.Serial/Serial Old**

　　Serial/Serial Old收集器是最基本最古老的收集器，它是一个单线程收集器，并且在它进行垃圾收集时，必须暂停所有用户线程。Serial收集器是针对新生代的收集器，采用的是Copying算法，Serial Old收集器是针对老年代的收集器，采用的是Mark-Compact算法。它的优点是实现简单高效，但是缺点是会给用户带来停顿。

## **2.ParNew**

　　ParNew收集器是Serial收集器的多线程版本，使用多个线程进行垃圾收集。

## **3.Parallel Scavenge**

　　Parallel Scavenge收集器是一个新生代的多线程收集器（并行收集器），它在回收期间不需要暂停其他用户线程，其采用的是Copying算法，该收集器与前两个收集器有所不同，它主要是为了达到一个可控的吞吐量。

## **4.Parallel Old**

　　Parallel Old是Parallel Scavenge收集器的老年代版本（并行收集器），使用多线程和Mark-Compact算法。

## **5.CMS**

　　CMS（Current Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的收集器，它是一种并发收集器，采用的是Mark-Sweep算法。

## **6.G1**

　　G1收集器是当今收集器技术发展最前沿的成果，它是一款面向服务端应用的收集器，它能充分利用多CPU、多核环境。因此它是一款并行与并发收集器，并且它能建立可预测的停顿时间模型。

### 总结和补充

　对象的内存分配，往大方向上讲就是在堆上分配，对象主要分配在新生代的Eden Space和From Space，少数情况下会直接分配在老年代。如果新生代的Eden Space和From Space的空间不足，则会发起一次GC，如果进行了GC之后，Eden Space和From Space能够容纳该对象就放在Eden Space和From Space。在GC的过程中，会将Eden Space和From  Space中的存活对象移动到To Space，然后将Eden Space和From Space进行清理。如果在清理的过程中，To Space无法足够来存储某个对象，就会将该对象移动到老年代中。在进行了GC之后，使用的便是Eden space和To Space了，下次GC时会将存活对象复制到From Space，如此反复循环。当对象在Survivor区躲过一次GC的话，其对象年龄便会加1，默认情况下，如果对象年龄达到15岁，就会移动到老年代中。

　　一般来说，大对象会被直接分配到老年代，所谓的大对象是指需要大量连续存储空间的对象，最常见的一种大对象就是大数组，比如：

　　byte[] data = new byte[4\*1024\*1024]

　　这种一般会直接在老年代分配存储空间。

　　当然分配的规则并不是百分之百固定的，这要取决于当前使用的是哪种垃圾收集器组合和JVM的相关参数。

# 类加载

需要看一下深入理解java虚拟机

## 类加载器

虚拟机设计团队把类加载阶段中的”通过一个类的全限定名来获取描述此类的二进制字节流”这个动作放到Java虚拟机外部去实现，以便让应用程序自己决定如何去获取所需要的的类。

实现这个动作的代码模块叫做“类加载器”。

## 双亲委派模型

从Java虚拟机的角度来讲，只存在两种不同的类加载器：一种是启动类加载器，使用C++语言实现的，是虚拟机自身的一部分；另一种就是其他的类加载器，这些类加载是由Java语言实现的，独立于虚拟机外部，并且全部继承java.lang.ClassLoader。

从Java开发者的角度来讲，类加载可以分为：

* 启动类加载
* 扩展类加载器：
* 应用程序类加载：
* 用户自定义的类加载：

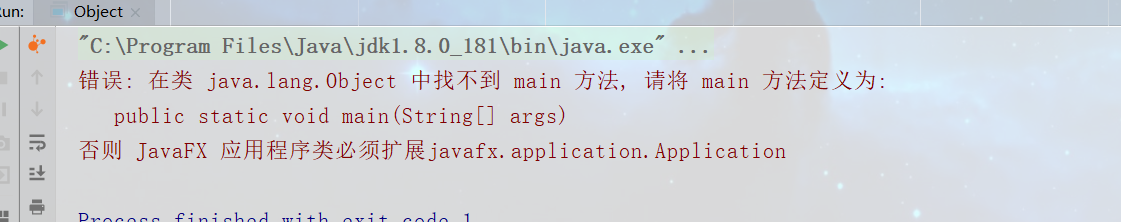
它们之间的层次关系如下：



双亲委派模型要求除了顶层的启动类加载之外，其余的类加载都需要有自己的父类加载器。这里类加载之间的父子关系一个不以继承关系来实现，而是都使用组合的方式来复用父类加载器的代码。

双亲委派模型的工作过程就是：如果一个类加载收到了类加载的请求，它首先不会自己去加载这个类，而是把请求委派给父类加载器去完成，每一个层次的类加载都是如此，因此所有的加载请求最终都会传送给启动类加载，只有当父类加载器反馈自己无法完成这个加载请求（它的搜索范围没有找到所需的类）的时候，子加载器才会尝试自己去加载。

使用双亲委派模型来组织类之间的关系，有一个显而易见的好处就是Java类随着类加载一起具备了一种带有优先级的层次关系。例如类java.lang.Object，它存放在rt.jar中，无论哪一个类加载器要加载这个类，最终都会委派给处于顶层的启动类加载器进行加载，因此Object类在程序的各种类加载器环境中都是同一个类。相反，如果没有使用双亲委派模型，由各个加载器自行加载的话，如果用户自己编写了一个称为java.lang.Object的类，并放在程序的ClassPath中，那系统中会出现多个不同的Object类，Java类型体系最基础的行为也就无法保证，应用程序也就会变得一片混乱。



# 线程安全

## 什么是线程安全

当多个线程访问某个对象的时候，不管运行时环境采用何种调度方式或者进程如何交替执行，并且没有采取其他的同步措施，该对象都能表现出正确的行为，就称这个对象时线程同步的。

## 如何保证线程安全

### 互斥同步

同步是指多个线程访问共享变量的时候，同一时刻只能有一个线程可以访问，其他的线程将处于阻塞等待中。

Synchonized关键字实现互斥同步的最基本的方式。

RetreentLock

### 非阻塞同步

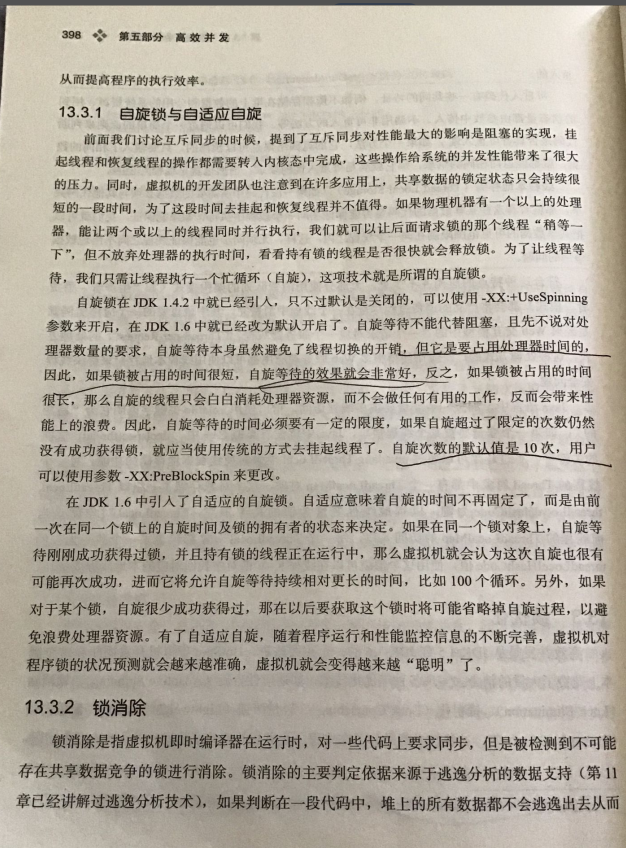
CAS

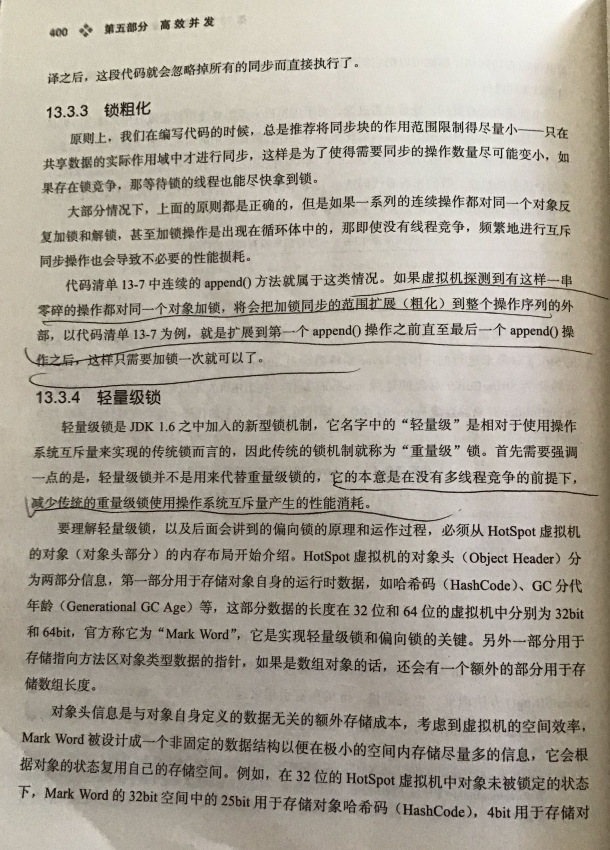
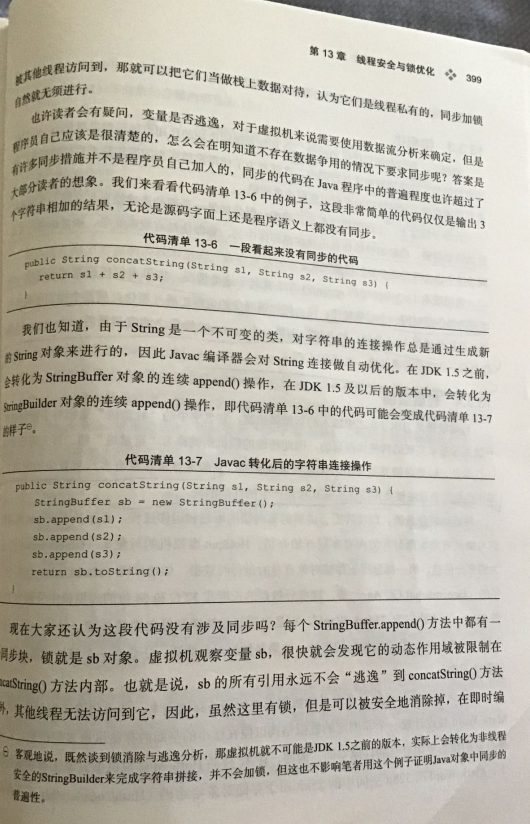
### 线程本地存储

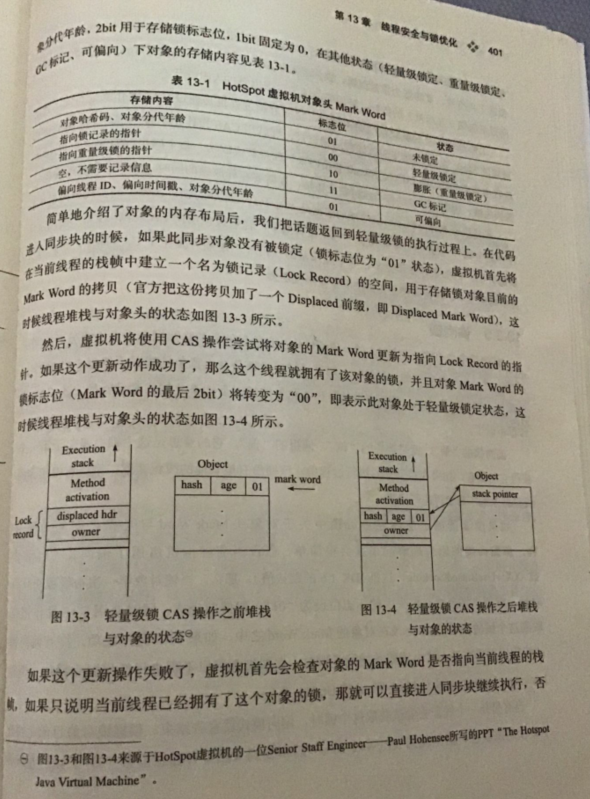
ThredLocal

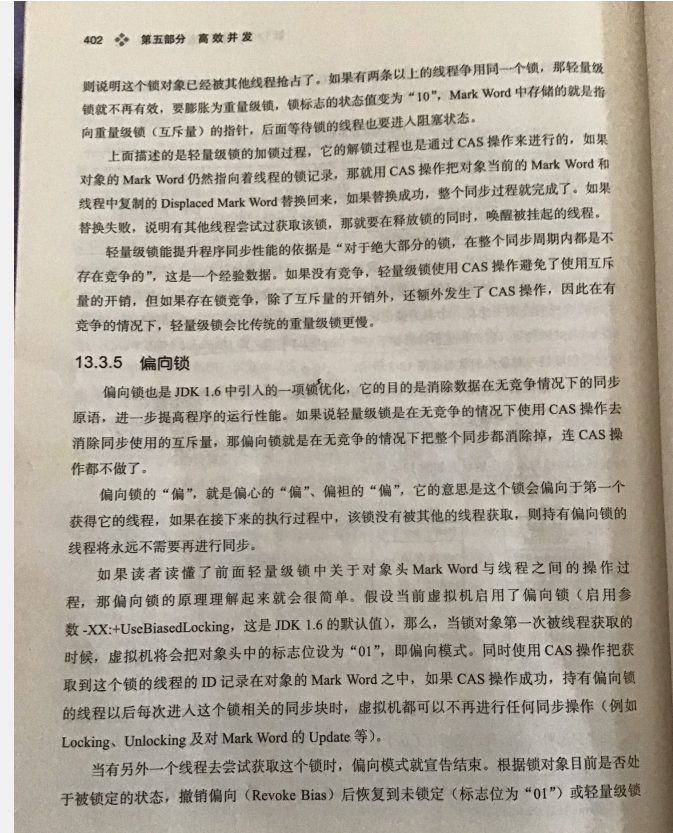
### 可以使用常量

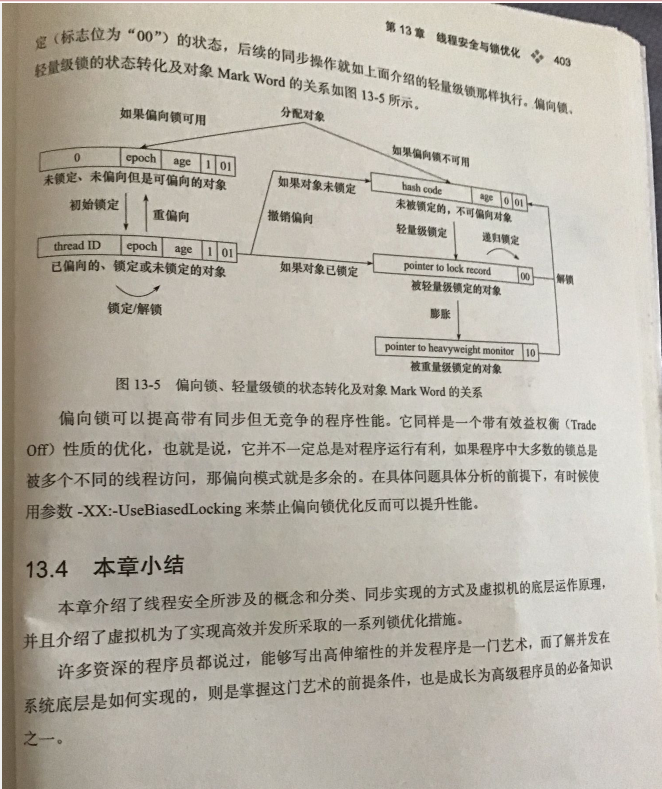
# 锁优化





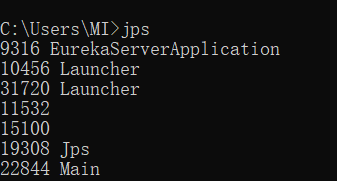




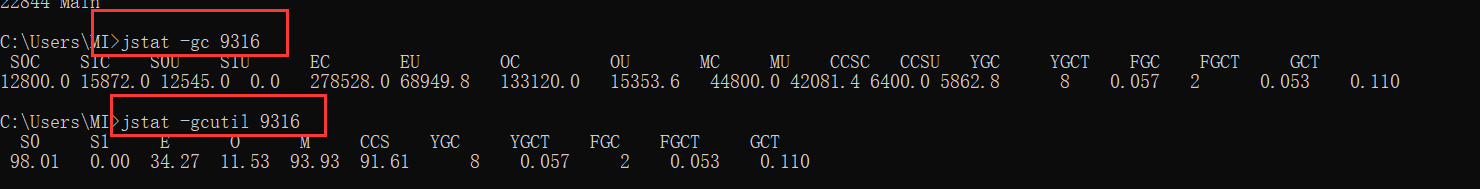


# 虚拟机性能监控及故障处理工具

## Jps



## Jstat



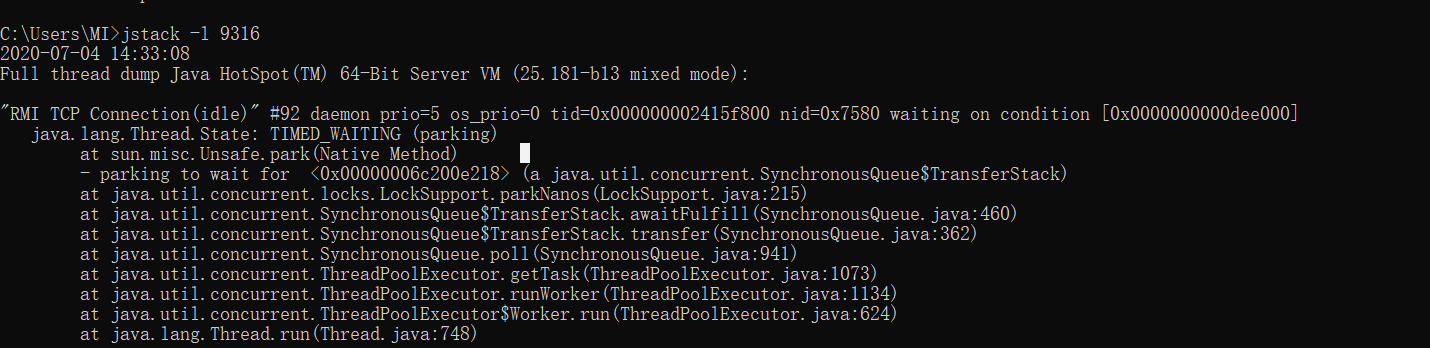
## Jinfo



## Jstack

|  |  |
| --- | --- |
| -l | 除堆栈外，显示关于锁的附加信息 |

堆栈跟踪工具



# 调优策略

两个基本原则：

* 将转移到老年代的对象数量降到最少。
* 减少Full GC的执行时间。目标是Minor GC时间在100ms以内，Full GC时间在1s以内。

主要调优参数：

· 设定堆内存大小，这是最基本的。

· -Xms：启动JVM时的堆内存空间。

· -Xmx：堆内存最大限制。

· 设定新生代大小。  
新生代不宜太小，否则会有大量对象涌入老年代。

· -XX:NewRatio：新生代和老年代的占比。

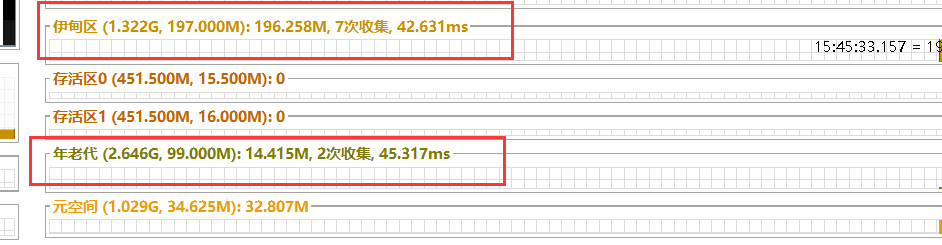
· -XX:NewSize：新生代空间。

· -XX:SurvivorRatio：伊甸园空间和幸存者空间的占比。

· -XX:MaxTenuringThreshold：对象进入老年代的年龄阈值。

· 设定垃圾回收器  
年轻代：-XX:+UseParNewGC。  
老年代：-XX:+UseConcMarkSweepGC。  
CMS可以将STW时间降到最低，但是不对内存进行压缩，有可能出现“并行模式失败”。比如老年代空间还有300MB空间，但是一些10MB的对象无法被顺序的存储。这时候会触发压缩处理，但是CMS GC模式下的压缩处理时间要比Parallel GC长很多。  
G1采用”标记-整理“算法，解决了内存碎片问题，建立了可预测的停顿时间类型，能让使用者指定在一个长度为M毫秒的时间段内，消耗在垃圾收集上的时间不得超过N毫秒。

调整参数之前：



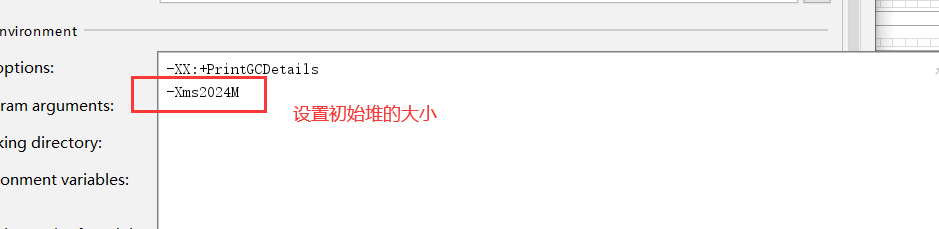
## 调整堆的初始大小

新生代的大小在不断变大。

|  |
| --- |
| [GC (Allocation Failure) [PSYoungGen: 65024K->7632K(75776K)] 65024K->7648K(249344K), 0.0080584 secs] [Times: user=0.11 sys=0.00, real=0.01 secs]  [GC (Allocation Failure) [PSYoungGen: 72656K->10292K(75776K)] 72672K->10316K(249344K), 0.0062092 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]  [GC (Metadata GC Threshold) [PSYoungGen: 23898K->6619K(75776K)] 23922K->6651K(249344K), 0.0037695 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]  [Full GC (Metadata GC Threshold) [PSYoungGen: 6619K->0K(75776K)] [ParOldGen: 32K->6352K(82944K)] 6651K->6352K(158720K), [Metaspace: 20532K->20530K(1067008K)], 0.0181962 secs] [Times: user=0.02 sys=0.00, real=0.02 secs]  [GC (Allocation Failure) [PSYoungGen: 65024K->5646K(122880K)] 71376K->12006K(205824K), 0.0031404 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]  [GC (GCLocker Initiated GC) [PSYoungGen: 122382K->10739K(136192K)] 128742K->17655K(219136K), 0.0065738 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]  [GC (Allocation Failure) [PSYoungGen: 136179K->13301K(217088K)] 143119K->22677K(300032K), 0.0089964 secs] [Times: user=0.02 sys=0.00, real=0.01 secs]  [GC (Metadata GC Threshold) [PSYoungGen: 26835K->9513K(220672K)] 36211K->18896K(303616K), 0.0056357 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]  [Full GC (Metadata GC Threshold) [PSYoungGen: 9513K->0K(220672K)] [ParOldGen: 9383K->14726K(99328K)] 18896K->14726K(320000K), [Metaspace: 33595K->33595K(1079296K)], 0.0300128 secs] [Times: user=0.17 sys=0.00, real=0.03 secs] |



给新生代设置初始值大小：



新生代的垃圾回收次数只剩了2次：时间缩短了一半。



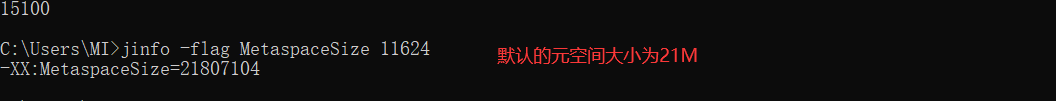
## 调整元空间的大小

****JDK8 HotSpot JVM 将移除永久区，使用本地内存来存储类元数据信息并称之为：元空间（Metaspace）****

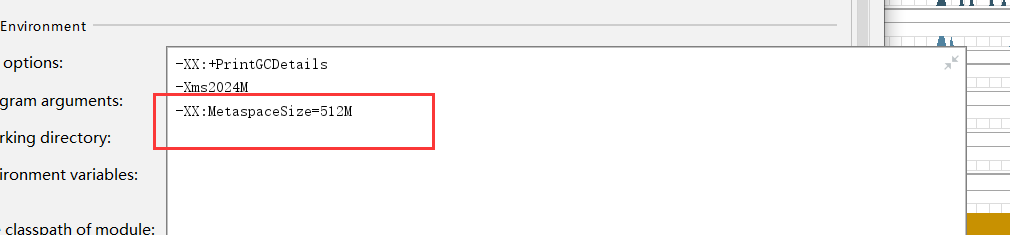
以下是Full GC的日志：



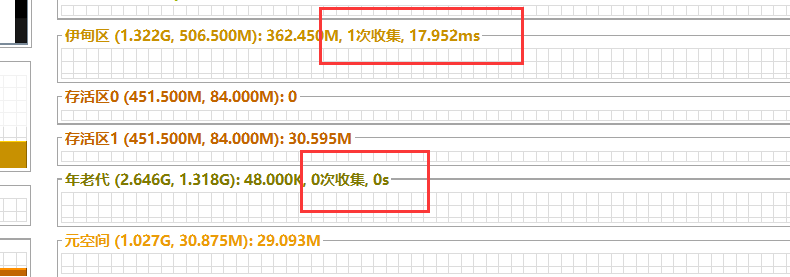
之所以发生Full GC是因为元空间初始的大小约为21M，空间较小，只要将元空间的内存调大一些，发生Full GC的次数将会变少。



将元空间设置为512M:

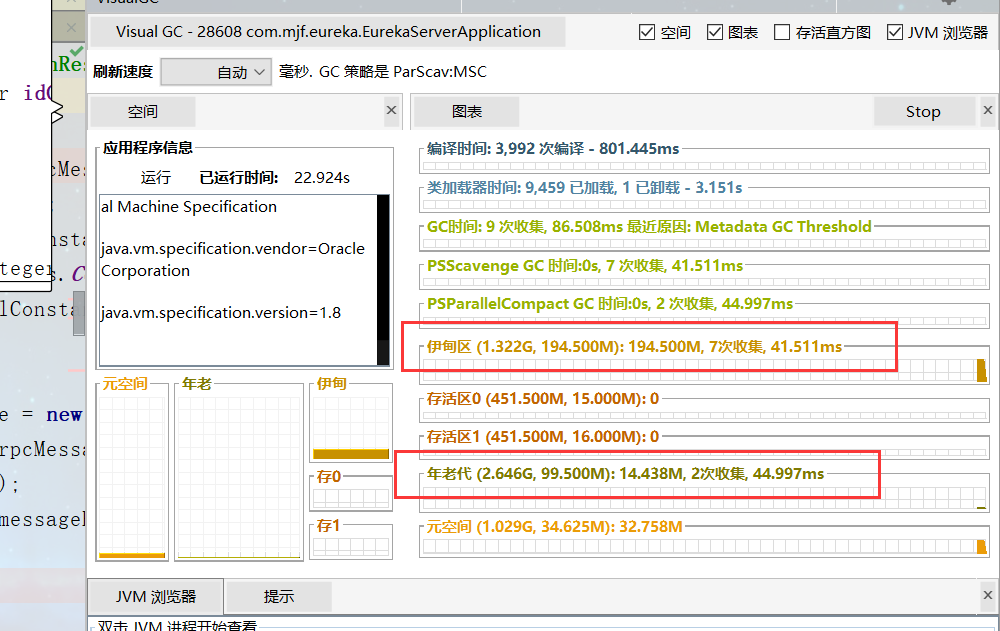


调整之后，老年代GC的次数变为了0，由当初的87ms降到了17ms，降了5倍。



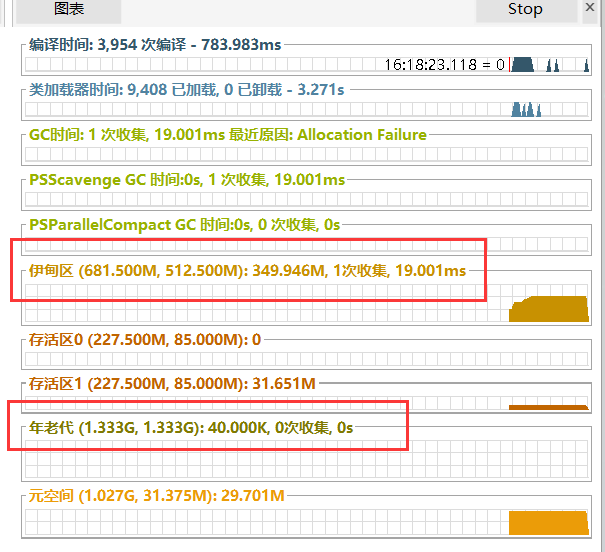
## 第二次测试

未设置参数之前：总共9次回收



设置参数之后：总共一次回收

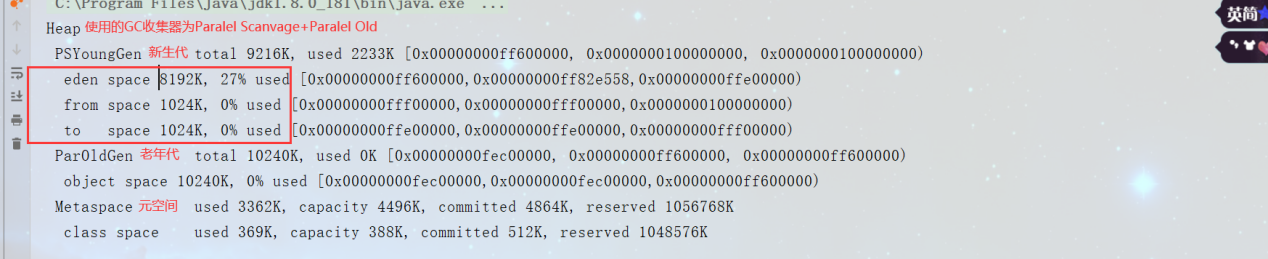
|  |
| --- |
| -XX:+PrintGCDetails  -Xms2048m  -Xmx2048m  -XX:MetaspaceSize=512M |



# 内存分配

1. 对象优先在Eden区中分配
2. 大对象直接进入老年代
3. 长期存活的对象将进入老年代（对象晋升老年代的年龄阈值可以通过参数：-XX:MaxTenuringThreshold=1）

## GC日志分析



### 案例一

|  |
| --- |
| -Xms20M  -Xmx20M  -Xmn10M  -XX:+PrintGCDetails  -XX:SurvivorRatio=8 |

|  |
| --- |
| **public class** TestGC {   **private static final int *\_1MB*** = 1024 \* 1024;   */\*\*  \* [GC (Allocation Failure) [PSYoungGen: 6165K(新生代在GC前总的使用情况)->840K(新生代在GC后的使用情况)(9216K(新生代总的容量))]  \* 6165K(堆在GC前使用的总的容量)->4944K(GC后堆的使用的总的容量)(19456K(堆的总容量，不包括to Survivor区域的)),  \* 0.0030609 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]  \* Heap  \* PSYoungGen total 9216K, used 7307K [0x00000000ff600000, 0x0000000100000000, 0x0000000100000000)  \* eden space 8192K, 78% used [0x00000000ff600000,0x00000000ffc50e58,0x00000000ffe00000)  \* from space 1024K, 82% used [0x00000000ffe00000,0x00000000ffed2020,0x00000000fff00000)  \* to space 1024K, 0% used [0x00000000fff00000,0x00000000fff00000,0x0000000100000000)  \* ParOldGen total 10240K, used 4104K [0x00000000fec00000, 0x00000000ff600000, 0x00000000ff600000)  \* object space 10240K, 40% used [0x00000000fec00000,0x00000000ff002020,0x00000000ff600000)  \* Metaspace used 3428K, capacity 4496K, committed 4864K, reserved 1056768K  \* class space used 373K, capacity 388K, committed 512K, reserved 1048576K  \** ***@param args*** *\*/* **public static void** main(String[] args) {  **byte**[] byte1 = **new byte**[2 \* ***\_1MB***];  **byte**[] byte2 = **new byte**[2 \* ***\_1MB***];  **byte**[] byte3 = **new byte**[2 \* ***\_1MB***];  **byte**[] byte4 = **new byte**[4 \* ***\_1MB***];  }  } |

### 案例二

Jvm参数的设置同案例一。

大对象直接在老年代进行分配

