# 

# 堆的核心概述

## 1.1、认识堆内存

### 堆与进程

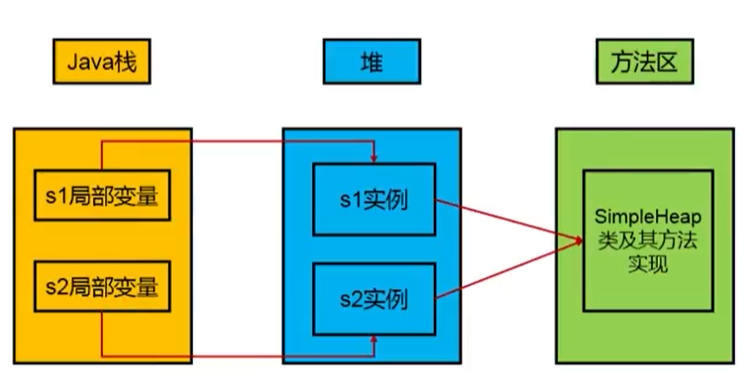
1. 堆针对一个JVM进程来说是唯一的，也就是一个进程只有一个JVM
2. 进程包含多个线程，他们是共享同一堆空间的

### 对堆的认识

1. 一个JVM实例只存在一个堆内存，堆也是Java内存管理的核心区域。
2. Java堆区在JVM启动的时候即被创建，其空间大小也就确定了，堆是JVM管理的最大一块内存空间，并且堆内存的大小是可以调节的。
3. 《Java虚拟机规范》规定，堆可以处于物理上不连续的内存空间中，但在逻辑上它应该被视为连续的。
4. 所有的线程共享Java堆，在这里还可以划分线程私有的缓冲区[Thread Local Allocation Buffer,TLAB] 面试问题:堆空间一定是所有线程共享的么？不是，TLAB线程在堆中独有的）
5. 《Java虚拟机规范》中对Java堆的描述是：所有的对象实例以及数组都应当在运行时分配在堆上。（The heap is the run-time data area from which memory for all class instances and arrays is allocated）

老师说：从实际使用角度看的，“几乎”所有的对象实例都在这里分配内存。因为还有一些对象是在栈上分配的（逃逸分析，标量替换）

1. 数组和对象可能永远不会存储在栈上，因为栈帧中保存引用，这个引用指向对象或者数组在堆中的位置。
2. 在方法结束后，堆中的对象不会马上被移除，仅仅在垃圾收集的时候才会被移除。
   1. 也就是触发了GC的时候，才会进行回收
   2. 如果堆中对象马上被回收，那么用户线程就会收到影响，因为有stop the word
3. 堆，是GC（Garbage Collection，垃圾收集器）执行垃圾回收的重点区域。



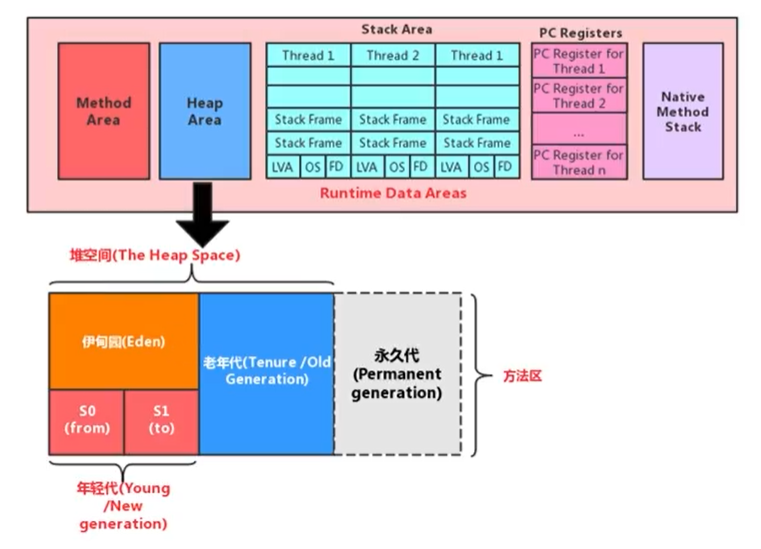
## 1.2 堆内存结构细分

* Java 7及之前堆内存逻辑上分为三部分：新生区 + 养老区 + **永久区**

1. Young/New Generation Space 新生区，又被划分为Eden区和Survivor区
2. Old/Tenure generation space 养老区
3. Permanent Space永久区 Perm

* Java 8及之后堆内存逻辑上分为三部分：新生区 + 养老区 + **元空间**

1. Young/New Generation Space 新生区，又被划分为Eden区和Survivor区
2. Old/Tenure generation space 养老区
3. Meta Space 元空间 Meta



* 约定：新生区 <-> 新生代 <-> 年轻代 、 养老区 <-> 老年区 <-> 老年代、 永久区 <-> 永久代
* 堆空间内部结构，JDK1.8之前从永久代 替换成 元空间

## 1.3 设置堆内存大小与OOM

Jvm参数的设置是在java运行环节加入进来的，java代码先编译成字节码文件，在执行java指令准备运行代码的时候，将jvm参数加入进来

### -Xms和-Xmx

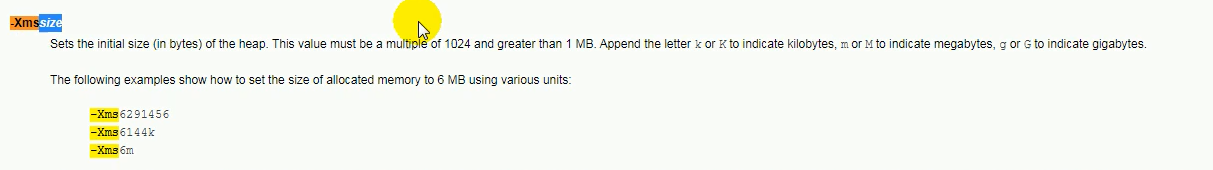
* 逻辑上分为新生区+养老区（即Xms/Xmx分配的内存物理上没有涉及方法区）
* 可以通过选项“-Xmx”和“-Xms”来设置堆**（新生代 + 老年代）**的大小：
  + **-Xms：**用于表示堆区的起始内存，等价于 -XX:InitialHeapSize
* -X：jvm 的运行参数
* ms：memory start
  + **-Xmx：用于表示堆区的最大内存，等价于 -XX:MaxHeapSize**

一旦堆区中的内存大小超过 -Xmx 所指定的最大内存时，将会抛出 OutOfMemoryError 异常。

* 通常会将 -Xms 和 -Xmx 两个参数配置相同的值，其目的是为了能够在Java垃圾回收机制清理完堆区后不需要重新分隔计算堆区的大小，从而提高性能

避免频繁扩容、释放，避免GC，让系统承担额外压力

* 默认情况下：
* 初始内存大小：物理内存大小 / 64
* 最大内存大小：物理内存大小 / 4
* 配置大小的单位是byte

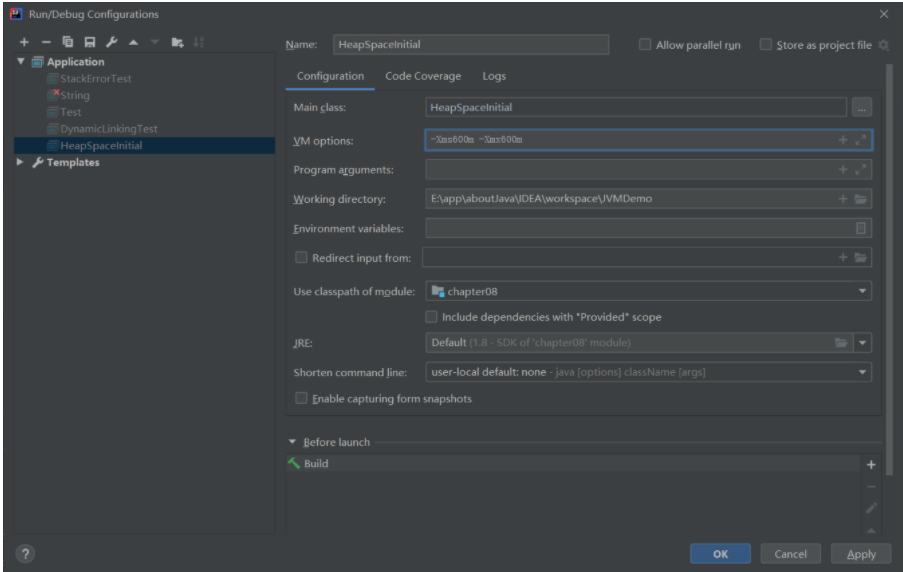


### IDEA中设置堆内存大小示例

* 查看默认堆内存大小

|  |
| --- |
| public class HeapSpaceInitial {  public static void main(String[] args) {  *// 返回 Java 虚拟机中的堆内存总量* long initialMemory = Runtime.*getRuntime*().totalMemory() / 1024 / 1024;  *// 返回 Java 虚拟机试图使用的最大内存总量* long maxMemory = Runtime.*getRuntime*().maxMemory() / 1024 / 1024;   System.*out*.println("-Xms: " + initialMemory + "M");  **//-Xms: 243M** System.*out*.println("-Xmx: " + maxMemory + "M");  **//-Xmx: 3604M** System.*out*.println("系统内存大小为：" + initialMemory \* 64.0 / 1024 + "G"); **//系统内存大小为：15.1875G** System.*out*.println("系统内存大小为：" + maxMemory \* 4.0 / 1024 + "G"); **//系统内存大小为：14.078125G** try {  Thread.*sleep*(1000000);  } catch (InterruptedException e) {  e.printStackTrace();  }  } } |

* Idea中手动设置堆大小



打印结果为：

-Xms: 575M

-Xmx: 575M

系统内存大小为：35.9375G

系统内存大小为：2.24609375G

### 查看堆内存

#### 方式1：通过CMD进入终端jps + jstat -gc 进程id

* jps Jps指令可以查看当前进行的java进程
* jstat -gc 进程id； 查看进程内存使用情况



* 各个参数说明：

这里单位是Kbyte；

S0和S1为幸存者区，EC为Eden区；

OC为老年代；

新生代总内存 = S0C+S1C+EC (25m+25m+150m = 200m)

新生代已用内存 = S0U + S1U + EU

老年代总内存 = OC (400m)

老年代已用内存 = OU

堆总内存 = S0C + S1C + OC +EC

* 设置堆大小为600m，为什么打印结果为575m？

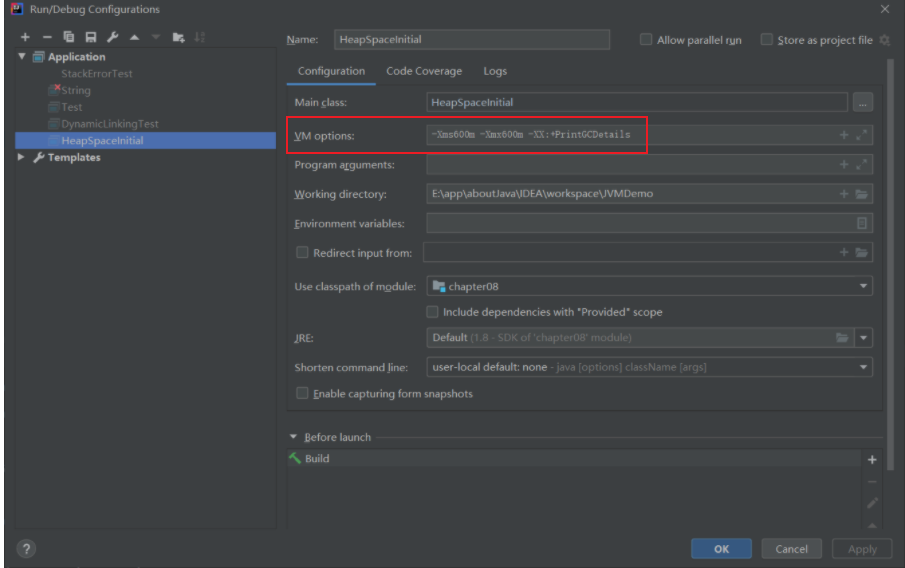
打印结果：Runtime.*getRuntime*().totalMemory() 和 Runtime.*getRuntime*().maxMemory()

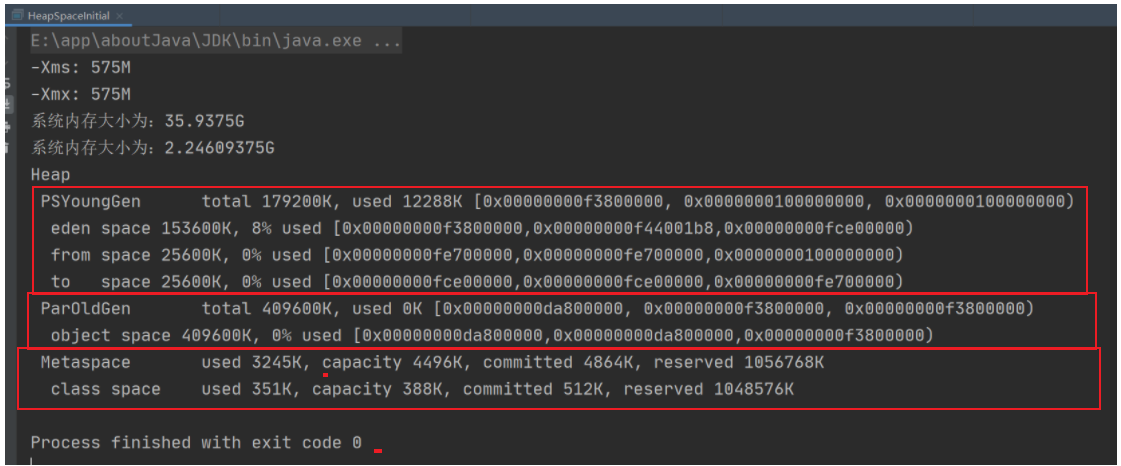
这是因为幸存者区S0和S1各占据了25m，但是他们始终有一个是空的; 存放对象的是伊甸园区和一个幸存者区。

通过Runtime获取的Memory的大小为： S0(或者S1) + Eden区 + old区;少了一个S区，所以计算结果比实际的堆内存要小。

#### 方式2：VM参数 控制台打印详细

（控制台打印）Edit Configurations->VM Options 添加 -XX:+PrintGCDetails



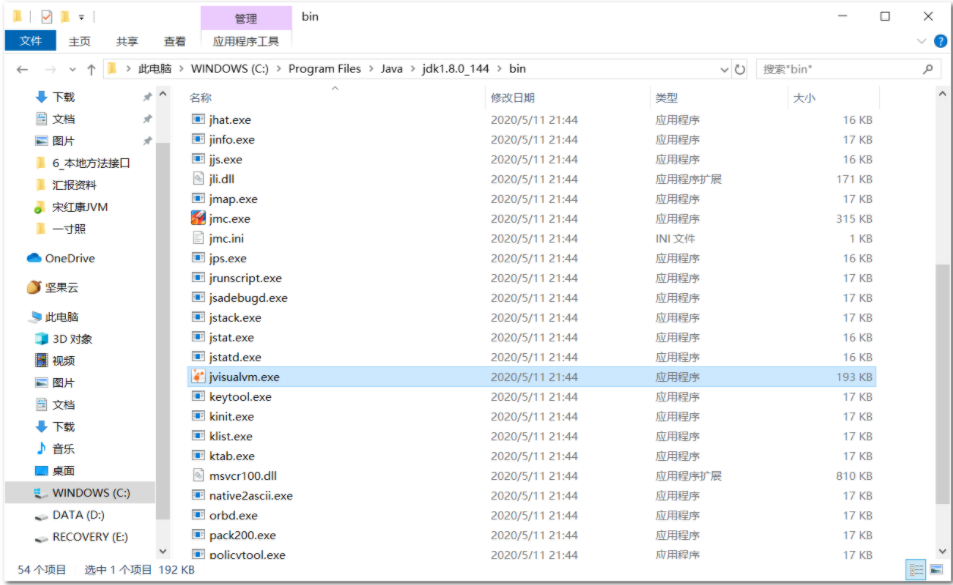


注意：这里新生代总大小179200k 仍然是只计算了eden+一个S区

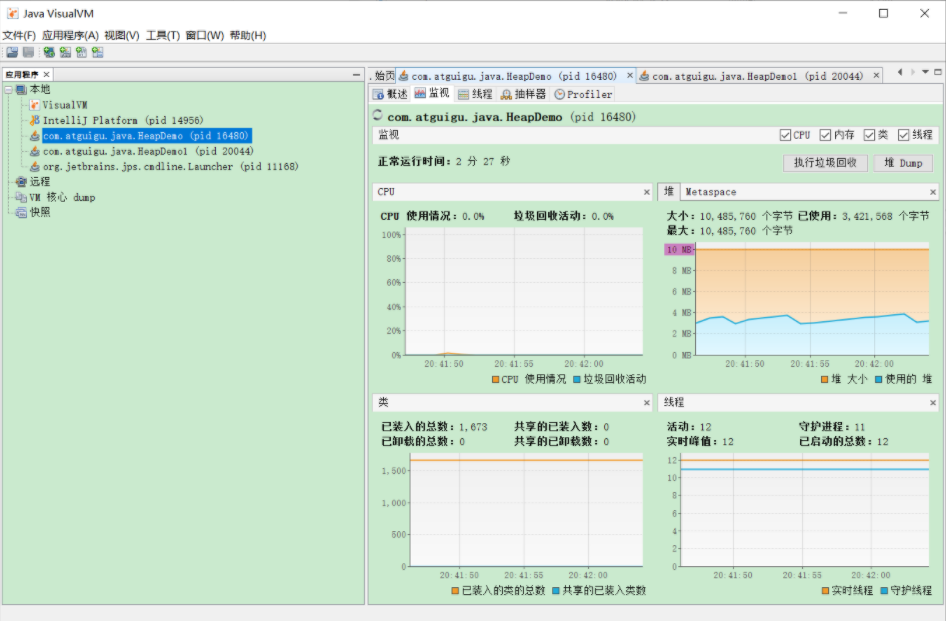
#### 方式3：Java VisualVM

使用 JDK 自带的工具：Java VisualVM ，来查看堆内存

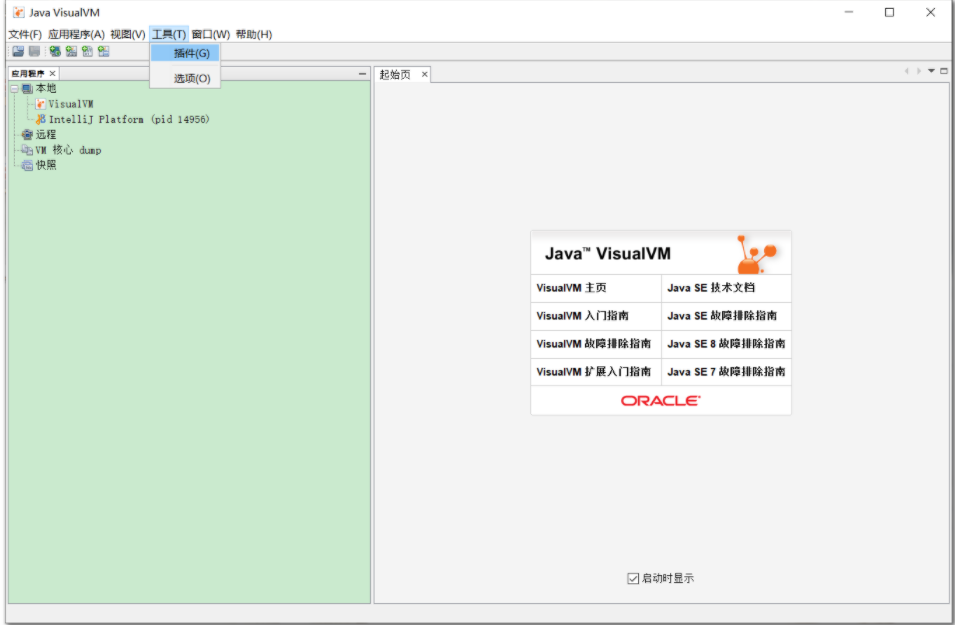
Java VisualVM 在 JDK 的 bin 目录下



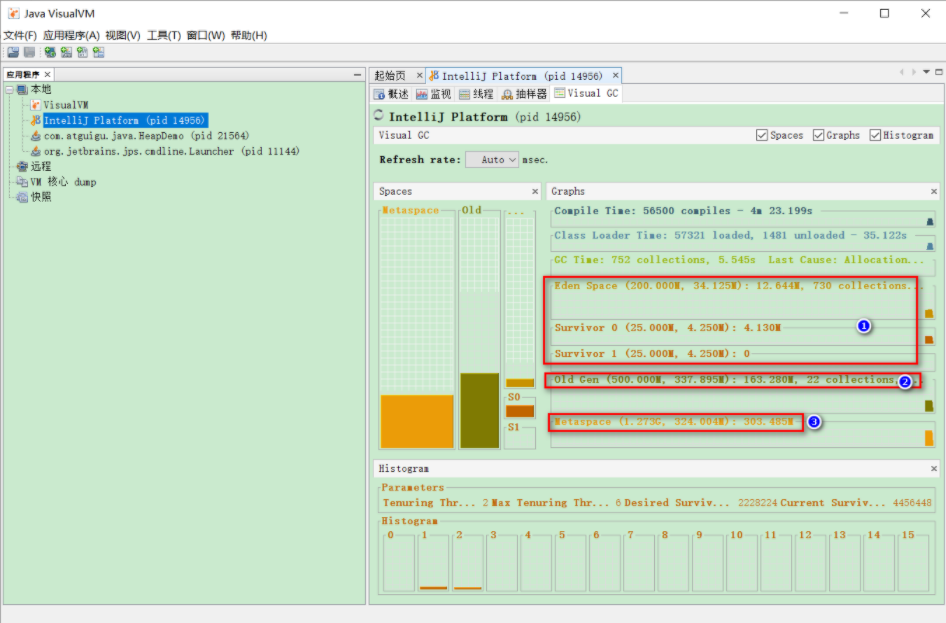
堆内存为10M：



安装Visual GC插件 可以查看堆内存细分情况







### OOM

* oom示意图：

https://imgconvert.csdnimg.cn/aHR0cDovL2hleWdvLm9zcy1jbi1zaGFuZ2hhaS5hbGl5dW5jcy5jb20vaW1hZ2VzL2ltYWdlLTIwMjAwNzI4MjI0OTQzNTUzLmdpZg

* OOM原因：

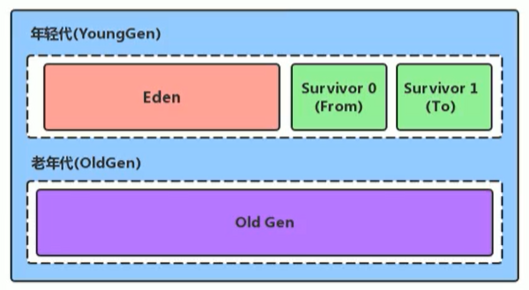
老年区一点一点在变大，直到最后一次垃圾回收器无法回收垃圾时，堆内存被撑爆，抛出 OutOfMemoryError 错误(**当Old区满了**，再添加对象，没办法进行GC了，就OOM了)

## 1.4年轻代与老年代的内存分配设置

存储在JVM中的java对象可以被分为两类：

* 生命周期短暂，这类对象的创建和消亡都非常迅速
* 另一类对象生命周期长，极端情况下还能与JVM生命周期保持一致。

Java堆区进一步细分可以分为年轻代（YoungGen）和老年代（OldGen）。其中年轻代可以分为Eden空间、Survivor0空间和Survivor1空间（有时也叫frmo区，to区）



### 配置新生代与老年代在堆结构的占比

* 一般不会调这个比例,当发现在整个项目中，生命周期长的对象偏多，那么就可以通过调整老年代的大小，来进行调优
* -XX:NewRatio=..
  + 默认-XX:NewRatio=2，表示新生代占1，老年代占2，新生代占整个堆的1/3
  + 可以修改-XX:NewRatio=4，表示新生代占1，老年代占4，新生代占整个堆的1/5



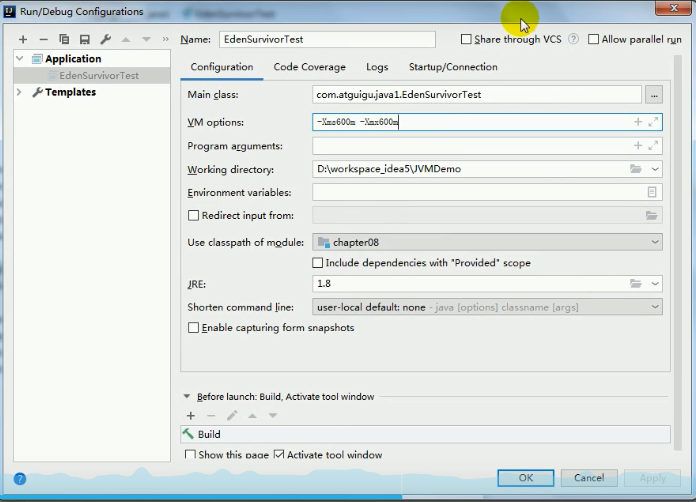
### 配置Eden和S在新生代中的占比

* 在hotSpot中，Eden空间和另外两个Survivor空间缺省所占的比例是8：1：1，开发人员可以通过选项 -XX:SurvivorRatio 调整空间比例，如-XX:SurvivorRatio=8
* -XX:SurvivorRatio

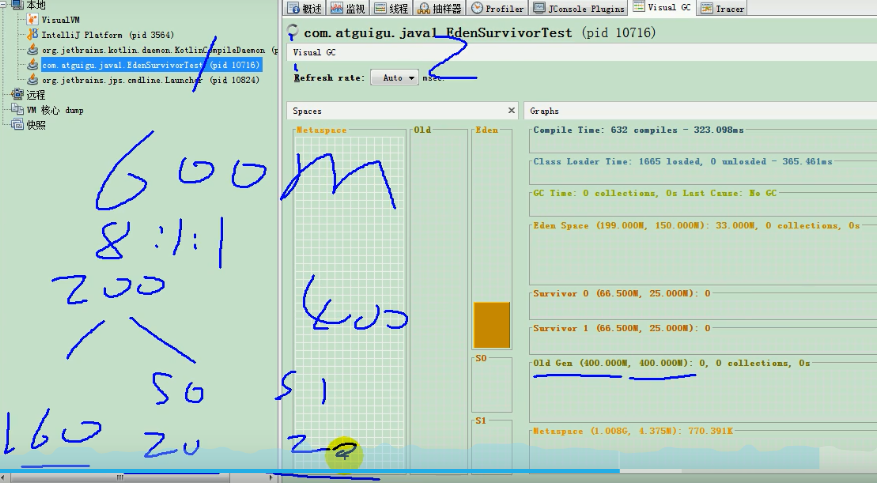
-XX:SurvivorRatio= 8 即 8：1：1

* 配置演示

1. 设置堆内存为600M:



1. 此时不设置-XX:SurvivorRatio= 8 查看堆内存情况：



会发现Eden只有150M S区25M [eden和s区不是按照8：1：1来的]

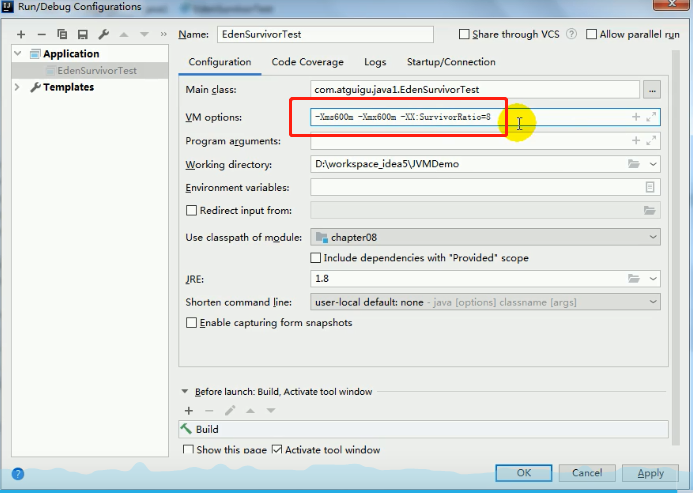
从命令行看：

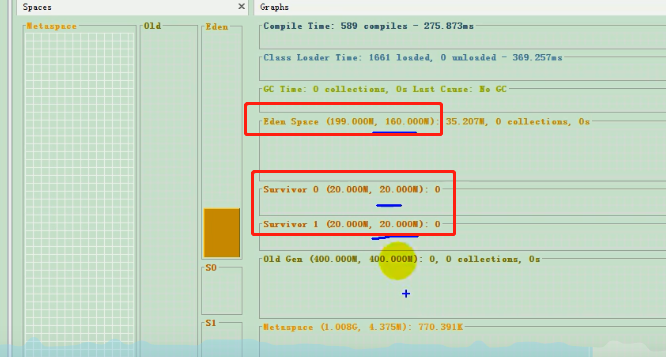


仍然是6：1：1

1. 要想实现真正的8：1：1 必须显示的设置：

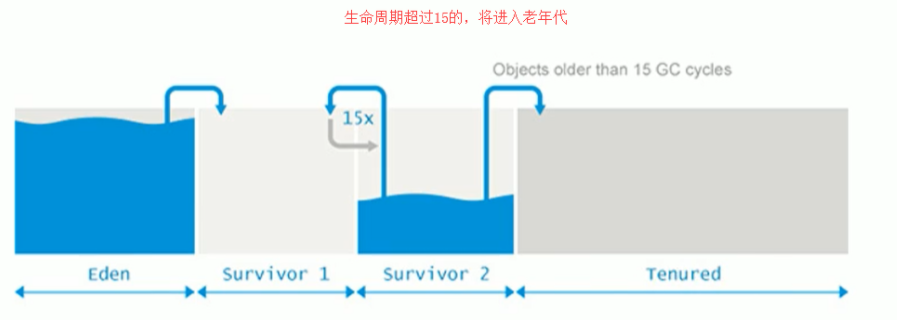
如下所示 设置虚拟机参数：



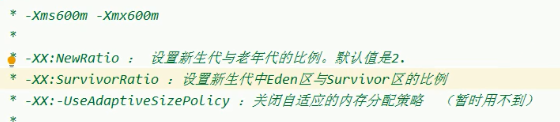


可以看出实现了8：1：1

* 几乎所有的对象都是在Eden中new出来的，绝大部分的Java对象的销毁都在新生代进行了(有些大的对象在Eden区无法存储时候，将直接进入老年代),当然在创建对象的时候其实是Eden+一个S区一起使用的。
* -Xmn：设置新生代最大内存，如果和-XX:NewRatio 冲突的时候，以-Xmn为准。(一般为默认)
* 新生区的对象默认生命周期超过 15 ，就会去养老区养老



本节中对堆空间的三种设置：



### 通过命令行查看各种比例

1)查看新生代与老年代的比例

jps

jinfo -flag NewRatios 进程id

2)查看新生区中伊甸园区与幸存者区的比例

jps

jinfo -flag SurvivorRatio 进程id

# 对象分配过程

## 2.1 对象分配过程

* 对象分配难点

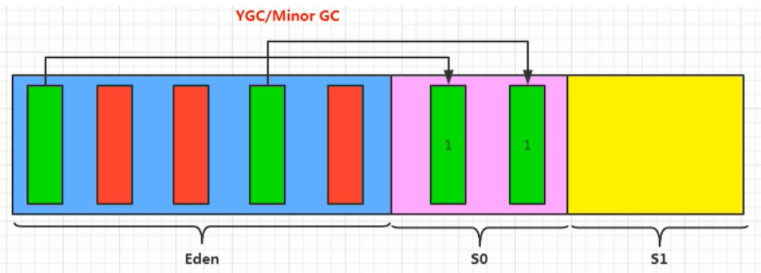
为新对象分配内存是一件非常严谨和复杂的任务，JVM的设计者们不仅需要考虑内存如何分配、在哪里分配等问题，并且由于内存分配算法与内存回收算法密切相关，所以还需要考虑GC执行完内存回收后是否会在内存空间中产生内存碎片。

* 对象分配过程

1. 起初创建对象，对象都分配在Eden区，此区有大小限制。
2. 当Eden区满了,程序又需要创建对象，JVM的垃圾回收器将对伊甸园区进行垃圾回收（Minor GC)，将伊甸园区中的不再被其他对象所引用的对象进行销毁，并且将Eden区中存活的对象移动到其中一个Survivor区，再加载新的对象放到Eden区
3. 如果再次触发垃圾回收，则将 Eden 区和 Survivor0 区中幸存下来的对象放到 Survivor1 区。Survivor0 区和 Survivor1 区依次交换。
4. 默认超过15次，就会移动到养老区
   1. 可以使用 -XX:MaxTenuringThreshold=<N> 参数进行设置。

* 图解

(1)Eden第一次满了，触发Minor GC回收

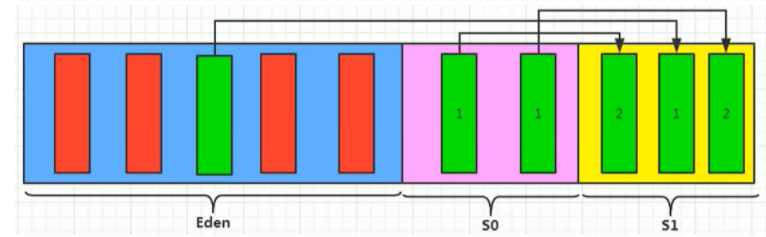


* **补充说明：**

每个对象都会有年龄计数器，每发生一次GC，存活下来的话都会加一。

图中绿色的是存活的，GC后仍然存活的，会被放进其中一个Survivor from区。

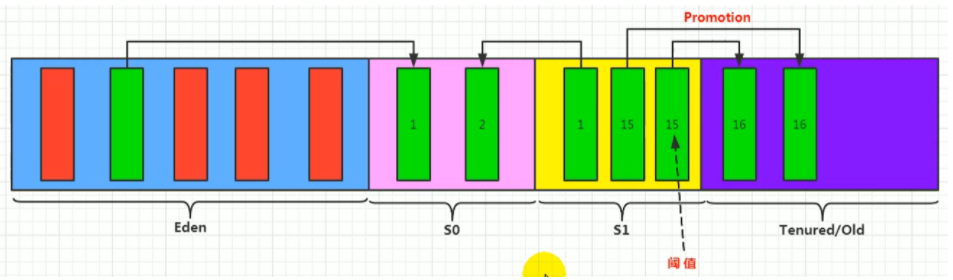
(2) Eden 第二次GC



Eden区继续存放对象，当Eden区再次存满的时候，又会触发一个MinorGC操作;**此时GC将会把 Eden和Survivor From中的对象进行一次垃圾收集**，把存活的对象放到 Survivor To区，同时让存活的对象年龄 + 1

* **注意：**针对幸存者s0,s1区：复制之后有交换，谁空谁是to。

(3) 第16次GC



我们继续不断的进行对象生成和垃圾回收，当Survivor中的对象的年龄达到15的时候，将会触发一次 Promotion 晋升的操作，也就是将年轻代中的对象晋升到老年代中

* 额外补充

1. 进入老年代的阈值

默认超过15次，就会移动到养老区。

可以使用 -XX:MaxTenuringThreshold=<N> 参数进行设置。

1. Minor GC触发机制

**注意：只有Eden满了才会触minorGC/youngGC,而幸存者区满了是绝对不会触发minorGC的**。

1. Major GC 和 OOM

在养老区，相对悠闲。**当老年区内存不足时，再次触发GC：Major GC**，进行养老区的内存清理。若养老区执行了Major GC之后发现依然无法进行对象的保存，**就会产生OOM异常。**

1. 关于垃圾回收

频繁在新生区收集，很少在养老区收集，几乎不在永久区/元空间收集。

1. 进入老年代有两种方式

一种是年纪到达阈值。

另一种是在年轻代满了，仍然添加对象的时候，会有**老年代担保**，直接进入老年代。

## 2.2 对象分配的特殊情况



来了一个新对象申请堆内存，先看看 Eden 是否放的下？

如果 Eden 放得下，则直接放到 Eden 区

如果 Eden 放不下，则触发 YGC ，执行垃圾回收，看看还能不能放下

如果 Eden 执行了 YGC 还是无法放不下该对象，那没得办法，只能说明是超大对象，只能直接怼到老年代

万一老年代都放不下，则先触发Full GC ，再看看能不能放下但如果还是放不下，那只能报 OOM 啦~~~

如果 Eden 区满了，在Minor GC的过程中，会判断S区能不能放的下存活的对象，放得下就将对象往幸存区拷贝，发现幸存区放不下啦，那只能便宜了某些新对象，让他们直接晋升至老年区

# 3、常用调优工具

常用调优工具

1. JDK命令行
2. Eclipse：Memory Analyzer Tool
3. Jconsole
4. Visual VM（实时监控 推荐~）
5. Jprofiler（推荐~）
6. Java Flight Recorder（实时监控）
7. GCViewer
8. GCEasy

# 4、Minor GC\Major GC\Full GC

我们都知道，JVM的调优的一个环节，也就是垃圾收集，我们需要尽量的避免垃圾回收，因为在垃圾回收的过程中，容易出现STW（Stop the World）的问题，而 Major GC 和 Full GC出现STW的时间，是Minor GC的10倍以上

JVM在进行GC时，并非每次都对上面三个内存区域一起回收的，大部分时候回收的都是指新生代。针对Hotspot VM的实现，它里面的GC按照回收区域又分为两大种类型：一种是部分收集(Partial GC),一种是整堆收集(Full GC)

而调优是为了让GC少一些，因为GC占用资源，影响用户线程。

## GC分类

GC方式可以根据收集范围分为两大类：

1. 部分收集(不是完整收集整个Java堆的垃圾收集)
   1. **新生代收集（Minor GC/Young GC）**：只是新生代的垃圾收集
   2. **老年代收集（Major GC/Old GC）**：只是老年代的圾收集。

目前，只有CMS GC会有单独收集老年代的行为。

注意，很多时候Major GC会和Full GC混淆使用，需要具体分辨是老年代回收还是整堆回收。

* 1. **混合收集（Mixed GC）：**收集整个新生代以及部分老年代的垃圾收集。

目前，只有G1 GC会有这种行为

1. 整堆收集（Full GC）：收集整个java堆和方法区的垃圾收集

## Minor GC触发机制

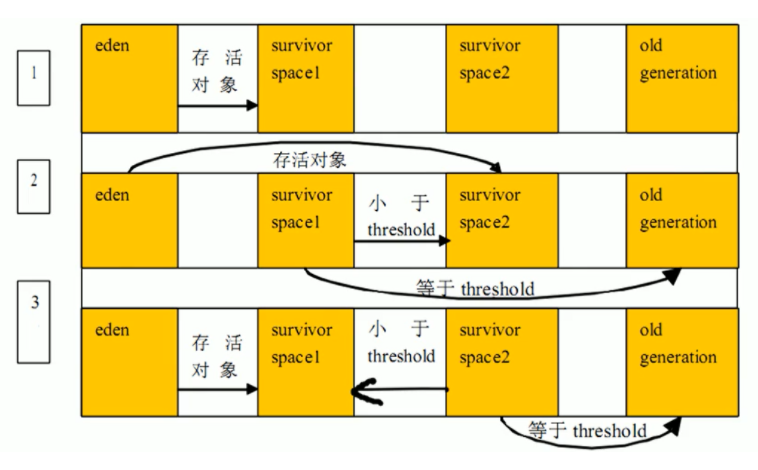
因为Java对象大多都具备朝生夕灭的特性，所以Monor GC 非常频繁，一般回收速度也比较快，这一定义既清晰又利于理解。

* **Minor GC触发时机：Eden区满了**

当年轻代空间不足时，就会触发Minor GC，这里的年轻代满指的是Eden代满，Survivor满不会引发GC.(每次Minor GC会清理整个年轻代的内存，Survivor是被动GC，不会主动GC)

* **Minor GC 会导致STW**

Minor GC会导致STW现象,暂停用户线程，等待垃圾回收结束，用户线程才恢复运行。



## 老年代GC（Major或者Full GC） 触发机制

* 二者都是指发生在老年代的GC,对象从老年代消失时，Major GC 或者 Full GC 发生了
* 出现了Major GC，经常会伴随至少一次的Minor GC(不是绝对的,在Parallel Scavenge 收集器的收集策略里就有直接进行Major GC的策略选择过程),也就是老年代空间不足时，会先尝试触发Minor GC。如果之后空间还不足，则触发Major GC
* Major GC速度一般会比Minor GC慢10倍以上，STW时间更长
* 当Major GC后，内存还不足，就报OOM了(并不是老年代满了就OOM,而是满了后触发Major GC之后内存仍然不足，才OOM)

## Full GC触发机制

触发Full GC执行的情况有以下五种

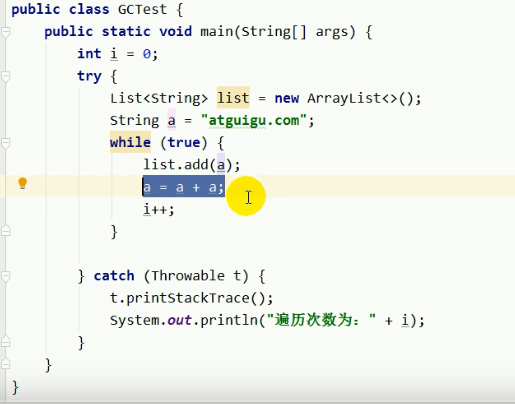
1. 调用System.gc()时，系统建议执行Full GC，但是不必然执行
2. 老年代空间不足
3. 方法区空间不足
4. **通过Minor GC后进入老年代的平均大小大于老年代的可用内存**
5. 由Eden区，Survivor S0（from）区向S1（to）区复制时，对象大小大于To Space可用内存，则把该对象转存到老年代，且老年代的可用内存小于该对象大小

**说明：Full GC 是开发或调优中尽量要避免的，这样暂停时间会短一些**

## GC日志

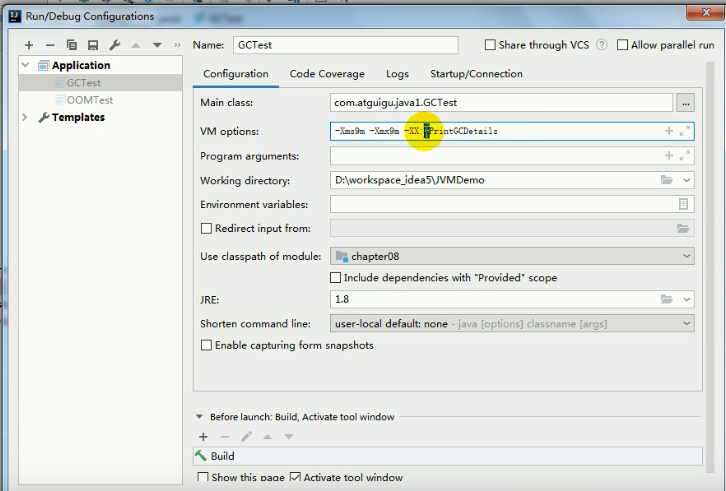
GC 日志：在 OOM 之前，一定会触发一次 Full GC ，因为只有在老年代空间不足时候，才会爆出OOM异常

代码：

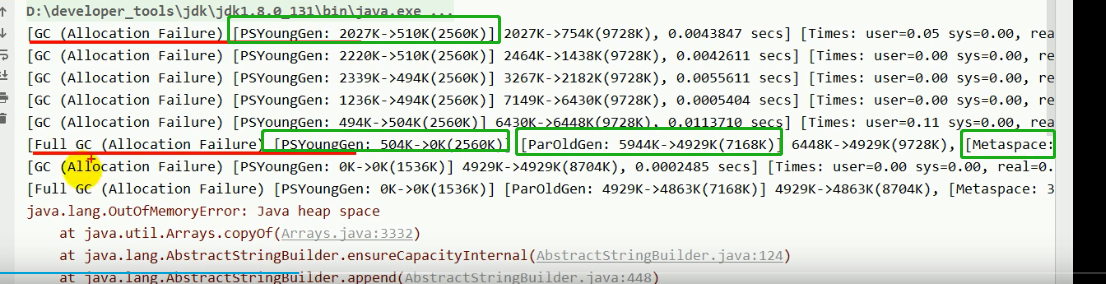


* JVM参数设置： -Xms9m -Xmx9m -XX:+PrintGCDetails

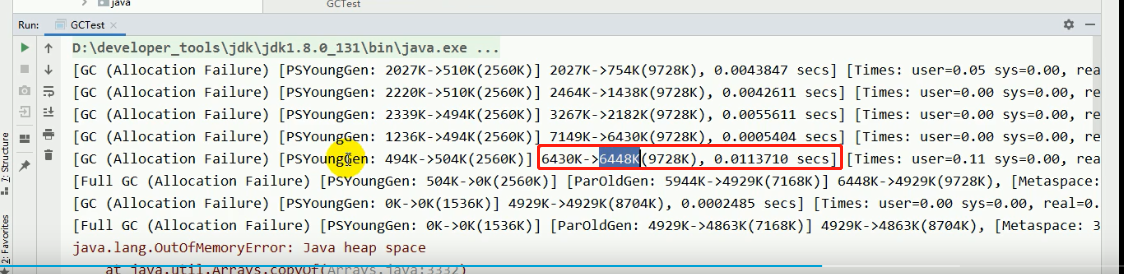
-XX:+PrintGCDetails打印GC日志



* 显示垃圾回收细节



可以看出堆空间占用空间越来越大 最后触发FULL GC



* 日志截取：

[Full GC (Ergonomics) [PSYoungGen: 1319K->0K(2560K)] [ParOldGen: 6782K->4864K(7168K)] 8102K->4864K(9728K), [Metaspace: 3452K->3452K(1056768K)], 0.0050464 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]

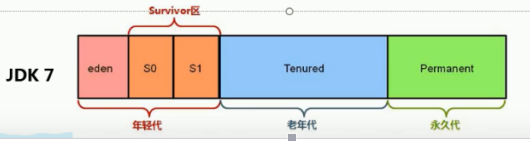
* [PSYoungGen: 1319K->0K(2560K)] ：年轻代总空间为 2560K ，当前占用 1319K ，经过垃圾回收后剩余 0K
* [ParOldGen: 6782K->4864K(7168K)] ：老年代总空间为 7168K ，当前占用 6782K ，经过垃圾回收后剩余 4864K
* 8102K->4864K(9728K)：堆内存总空间为 9728K ，当前占用 8102K ，经过垃圾回收后剩余 4864K
* [Metaspace: 3452K->3452K(1056768K)] ：元空间总空间为 1056768K ，当前占用 3452K ，经过垃圾回收后剩余 3452K
* 0.0050464 secs ：垃圾回收用时 0.0050464 secs

# 5、堆空间分代思想

**为什么要把Java堆分代？不分代就不能正常工作了吗？**

经研究，不同对象的生命周期不同。70%-99%的对象都是临时对象。

* 新生代：有Eden、Survivor构成（s0,s1 又称为from to），to总为空
* 老年代：存放新生代中经历多次依然存活的对象



其实不分代完全可以，**分代的唯一理由就是优化GC性能**。

如果没有分代，那所有的对象都在一块，就如同把一个学校的人都关在一个教室。GC的时候要找到哪些对象没用，这样就会对堆的所有区域进行扫描，而很多对象都是朝生夕死的，如果分代的话，把新创建的对象放到某一地方，当GC的时候先把这块存储“朝生夕死”对象的区域进行回收，这样就会腾出很大的空间出来。



# 6、内存分配策略

## 1.对象晋升规则

1. 对象在Eden出生并经过第一次Minor GC后，存活对象放进Survivor中，并年龄+1。
2. 对象在Survivor中每熬过一次MinorGC，年龄就会加1
3. 年龄到了一定程度，就会进入老年代。
4. 年龄的阈值可以通过：-XX:MaxTenuringThreshold来设置

## 2.不同年龄段对象分配原则：

1. 优先分配到Eden 开发中比较长的字符串或者数组，会直接存在老年代，但是因为新创建的对象都是朝生夕死的，所以这个大对象可能也很快被回收，但是因为老年代触发Major GC的次数比 Minor GC要更少，因此回收起来就会比较慢
2. 大对象直接分配到老年代 如字符串、数组 应该避免程序中出现过多的大对象
3. 长期存活的对象分配到老年代
4. **动态对象年龄判断** 如果Survivor区中相同年龄的所有对象大小的总和大于Survivor空间的一半，年龄大于或者等于该年龄的对象可以直接进入老年代，不用等到阈值年纪。



## 3.空间分配担保机制（☆☆☆）

空间分配担保： -XX:HandlePromotionFailure ，也就是经过Minor GC后，所有的对象都存活，因为Survivor比较小，所以就需要将Survivor无法容纳的对象，存放到老年代中。

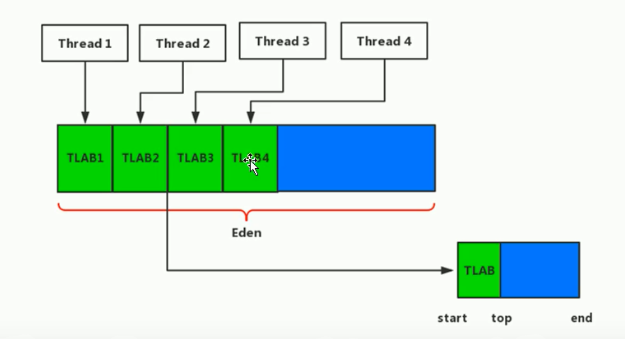
|  |
| --- |
| public class GCTest {  public static void main(String[] args) {  byte[] allocation1, allocation2,allocation3,allocation4,allocation5;  allocation1 = new byte[32000\*1024];  allocation2 = new byte[1000\*1024];  allocation3 = new byte[1000\*1024];  allocation4 = new byte[1000\*1024];  allocation5 = new byte[1000\*1024];  }  } |

给 allocation2 分配内存的时候 eden 区内存几乎已经被分配完了，我们刚刚讲了当 Eden 区没有足够空间进行分配时，虚拟机将发起一次 Minor GC.GC 期间虚拟机又发现 allocation1 无法存入 Survivor 空间，所以只好通过 分配担保机制：把新生代的对象提前转移到老年代中去，老年代上的空间足够存放 allocation1，所以不会出现 Full GC。执行 Minor GC 后，后面分配的对象如果能够存在 eden 区的话，还是会在 eden 区分配内存。

# 7、TLAB(Thread Local Allocation Buffer)

* **JVM为每个线程分配了一个私有缓存区域，该区域在Eden区域中。**

所有OpenJDK衍生出来的JVM都提供了TLAB的设计



* 为啥要有？

由于堆是线程共享区域，任何线程都可以访问到堆区中的共享数据；又因为对象实例的创建在JVM中非常频繁，因此在并发环境下从堆区中划分内存空间是线程不安全的

为避免多个线程操作同一地址，需要使用加锁等机制，进而影响分配速度。

* TLAB再说明

不是所有的实例对象都在TLAB中成功分配内存，但是JVM是将TLAB作为内存分配的首选。

一旦对象在TLAB空间分配内存失败，JVM就会尝试着通过使用加锁机制来确保数据操作的原子性，从而直接在Eden空间中分配内存。

* **TLAB配置及使用**

1. 可以使用-XX:UseTLAB 来设置是否开启TLAB空间 (默认开启)

命令行可以查看是否开启：

先启动一个Java进程，设置睡眠时间很久

然后jps看进程id, jinfo -flag UseTLAB + 进程号

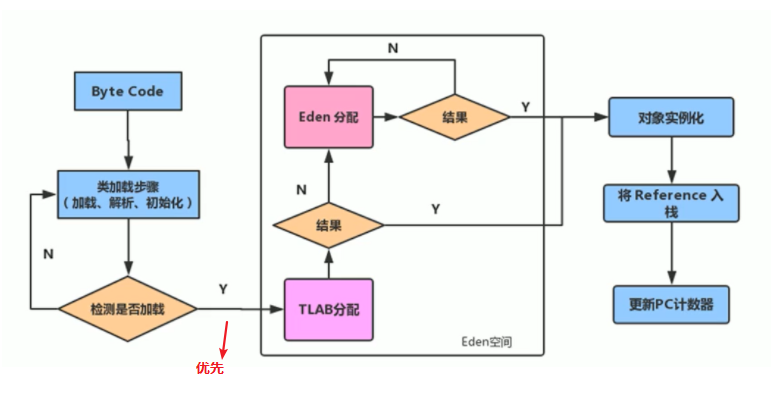
结果为： -XX:+UseTLAB 表示开启

测试代码：

|  |
| --- |
| public class TLABArgsTest {  public static void main(String[] args) {  System.out.println("我只是来打个酱油~");  try {  Thread.sleep(1000000);  } catch (InterruptedException e) {  e.printStackTrace();  }  }  } |

1. 默认情况下，TLAB空间非常小，只占Eden空间的1%，可以通过-XX:TLABWasteTargetPercent来设置TLAB占Eden的占比。

## 带TLAB的分配过程



结论：所以 堆空间未必都是共享的！！！

# 堆空间设置参数

官方文档：

https://docs.oracle.com/javase/8/docs/technotes/tools/unix/java.html

1. -XX:PrintFlagsInitial: 查看所有参数的默认初始值
2. -XX:PrintFlagsFinal：查看所有的参数的最终值（可能会存在修改，不再是初始值）
3. 具体查看某个参数的指令：
4. jps：查看当前运行中的进程
5. jinfo -flag SurvivorRatio 进程id： 查看新生代中Eden和S0/S1空间的比例
6. -Xms: 初始堆空间内存（默认为物理内存的1/64）
7. -Xmx: 最大堆空间内存（默认为物理内存的1/4）
8. -Xmn: 设置新生代大小（初始值及最大值）
9. -XX:NewRatio: 配置新生代与老年代在堆结构的占比
10. -XX:SurvivorRatio：设置新生代中Eden和S0/S1空间的比例
11. -XX:MaxTenuringThreshold：设置新生代垃圾的最大年龄(默认15)
12. -XX:+PrintGCDetails：输出详细的GC处理日志
13. 打印gc简要信息：① -XX:+PrintGC ② -verbose:gc
14. -XX:HandlePromotionFailure：是否设置空间分配担保

# 9、老年代担保机制

发生Minor GC之前，JVM会检查老年代**最大可用的连续空间**是否大于**新生代所有对象的总空间**。

如果大于，直接Minor GC，因为安全

如果小于，那么虚拟机会检查 -XX:HandlePromotionFailure设置的值是否允许担保失败。（JDK 7以后的规则HandlePromotionFailure可以认为就是true）

如果HandlePromotionFailure=false 意味着不给担保，那么进行full GC

如果HandlePromotionFailure=true 那么会继续检查老年代**最大的可用连续空间**是否大于**以往晋升到老年代的对象的平均大小。**

**如果大于，则进行一次Minor GC，但是会有风险**

**如果小于，进行Full GC**

JDK7以后，HandlePromotionFailure 该参数就失效了，默认为=true，这意味着上面的过程可以缩写为：

发生Minor GC之前，JVM会检查老年代最大可用的连续空间是否大于新生代所有对象的总空间。

如果大于，直接Minor GC，因为安全

如果小于那么会继续检查老年代最大的可用连续空间是否大于以往晋升到老年代的对象的平均大小或者新生代。

**最简洁版本为：**

**也就是说只要老年代的连续空间大于新生代对象的总大小或者以往晋升到老年代的对象的平均大小，就Minor GC 否则FullGC**

# 10、堆是对象存储的唯一选择吗？

在《深入理解Java虚拟机》中关于Java堆内存有这样一段描述：

随着JIT编译期的发展与**逃逸分析技术**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，所有的对象都分配到堆上也渐渐变得不那么“绝对”了。  
  在Java虚拟机中，对象是在Java堆中分配内存的，这是一个普遍的常识。但是，有一种特殊情况，那就是如果经过逃逸分析（Escape Analysis)后发现，一个对象并没有逃逸出方法的话，那么就可能被优化成栈上分配。这样就无需在堆上分配内存，也无须进行垃圾回收了。这也是最常见的堆外存储技术。  
  此外，前面提到的基于OpenJDK深度定制的TaoBaoVM,其中创新的GCIH(GCinvisible heap)技术实现off-heap,将生命周期较长的Java对象从heap中移至heap外，并且GC不能管理GCIH内部的Java对象，以此达到降低GC的回收频率和提升GC的回收效率的目的。

不是。栈上分配技术和标量替换优化技术。

逃逸分析：如果一个对象没有逃逸出方法的话，那么就可能被优化称为栈上分配。

## 1.逃逸分析技术（☆☆☆）

**逃逸分析的作用：**通过逃逸分析，Java Hotspot编译器能够分析出一个新的对象的引用的使用范围从而决定是否要将这个对象分配到堆上。没有发生逃逸的对象，则可以分配到栈上，随着方法执行的结束，栈空间就被移除

**逃逸分析的描述：**这是一种可以有效减少Java程序中同步负载和内存堆分配压力的跨函数全局数据流分析算法。

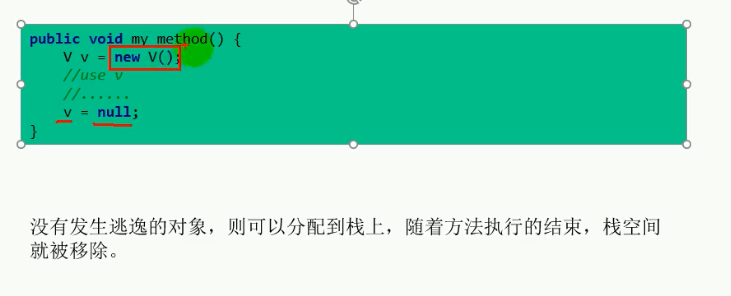
**逃逸分析的评判标准：**就是分析对象动态作用域（生命周期）

**前提：当一个对象在方法中被定义。** 如果这个对象只在方法内部使用，则认为没有发生逃逸。如果它被外部方法所引用，则认为发生逃逸。例如作为调用参数传递到其他地方中。

**逃逸分析的本质：**如何快速的判断是否发生了逃逸分析，就看new的对象实体是否有可能在方法外被调用

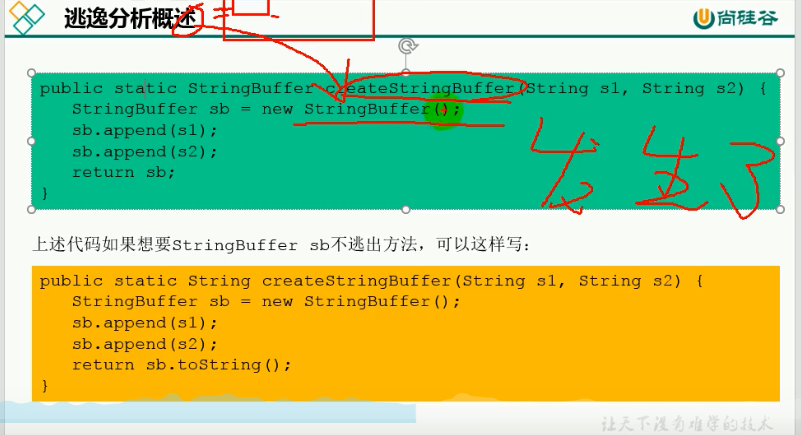
**逃逸分析的发生时间：**在JIT编译期间。

### （1）例1



new V(); 在堆空间中分配了一个V对象，但是该V对象的生命周期仅限于这个方法内。这就认为该对象没有发生逃逸，这样的话，该对象可以分配到栈上，随着方法执行结束，栈空间就被移除。

### （2）例2



第一个例子，StringBuffer对象在执行append()方法后，得到的仍然是StringBuffer对象，最后对象被返回了，很可能被别的方法调用，因此逃逸了，没有随着方法的结束而消亡。

第二个例子中，返回的不是StringBuffer对象，StringBuffer调用toString()后得到的是新的String对象，因此StringBuffer对象逃逸了。

## 2.如何判断对象是否会逃逸

最根本的判断依据就是方法体内New的对象会不会在外部被使用。

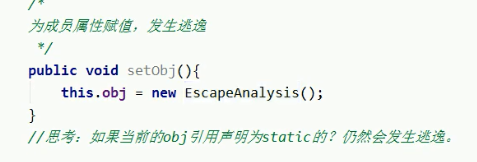
### （1）对象作为返回值 被弹出方法。



**如果obj是null，那么创建一个EscapeAnalysis()对象，然后弹出 发生逃逸。**

**如果不是null，那么就不会创建该对象。**

### （2）方法中创建的对象赋值给成员属性



EscapeAnalysis对象赋值给了实例变量或者类变量，这说明他的生命周期不随方法结束而结束，便发生了逃逸。

如果当前的obj引用声明为static的？仍然会发生逃逸。

### （3）方法中引用的是成员变量

该成员变量压根就不属于发方法的，而是实例的属性，逃逸。本质上这不是逃逸问题





## 逃逸分析参数设置

在JDK 1.7版本之后，HotSpot中默认就已经开启了逃逸分析

如果使用了较早的版本，开发人员可以通过

* -XX:DoEscapeAnalysis 显式开启逃逸分析
* -XX:+PrintEscapeAnalysis查看逃逸分析的筛选结果

**结论：开发中能使用局部变量的，就不要使用在方法外定义**

## 根据逃逸分析进行代码优化

使用逃逸分析，编译器可以对代码进行以下几种优化方式：

1. **栈上分配：** 就是说一个方法的局部变量，其生命周期仅限于该方法内部，随着方法结束而消亡，此时可以不进行堆分配，而是像局部变量表一样在栈上分配。
2. **同步省略： 如果一个对象被发现只能从一个线程被访问到，那么对于这个对象的操作可以不考虑同步。**
3. **分离对象或者标量替换。**有的对象可能不需要作为一个连续的内存结构存在也可以被访问到，那么对象的部分（或全部）可以不存储在内存，而是存储在CPU寄存器中。

### 栈上分配

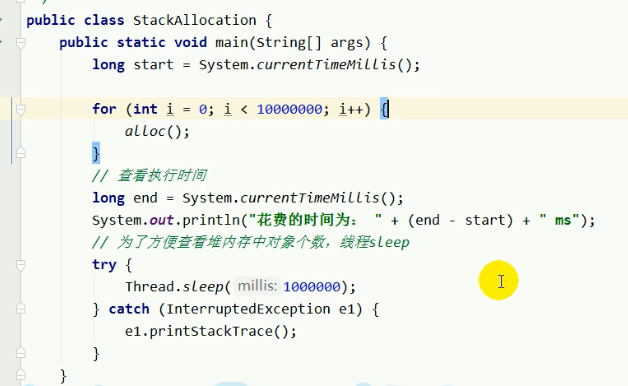
JIT编译器在编译期间根据逃逸分析的结果，发现如果一个对象并没有逃逸出方法的话，就可能被优化成栈上分配。分配完成之后，继续在调用栈内执行，最后线程结束，栈空间被回收，局部变量对象也被回收。这样就无须机型垃圾回收了

**常见的发生逃逸的场景：**给成员变量赋值、方法返回值、实例引用传递

代码：

|  |
| --- |
| /\*\*  **\* 栈上分配测试 JVM参数配置：**  **\* -Xmx256m -Xms256m -XX:-DoEscapeAnalysis -XX:+PrintGCDetails**  \*  \* @author shkstart shkstart@126.com  \* @create 2020 10:31  \*/  public class StackAllocation {  public static void main(String[] args) {  long start = System.currentTimeMillis();  for (int i = 0; i < 10000000; i++) {  alloc();  }  // 查看执行时间  long end = System.currentTimeMillis();  System.out.println("花费的时间为： " + (end - start) + " ms");  // 为了方便查看堆内存中对象个数，线程sleep  try {  Thread.sleep(1000000);  } catch (InterruptedException e1) {  e1.printStackTrace();  }  }  private static void alloc() {  User user = new User();//未发生逃逸  }  static class User {  }  } |

以下代码是创建10000000个对象。对象没有逃逸。





**不开启逃逸分析**：也就是说在堆中创建对象。



**开启逃逸分析后:只要4ms**

**减小堆空间大小在测试一次：**

不开启逃逸分析



开启逃逸分析：

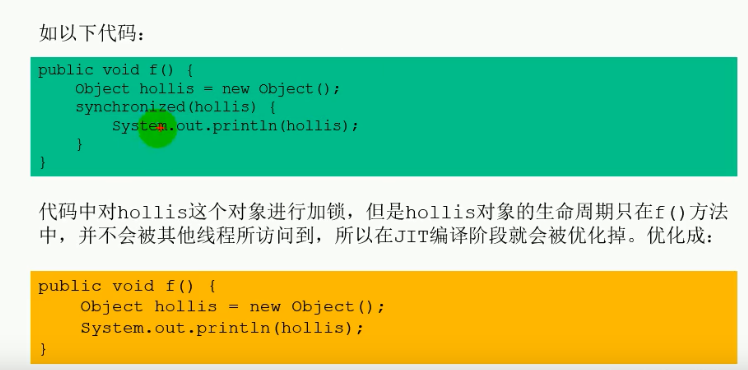


可以看出开启逃逸分析，未逃逸对象只在栈上分配，随着方法结束而消失，不需要GC，不需要在堆空间分配内存，因此速度要快很多，而且JVM不会维护10000000个对象实例。

### 同步省略

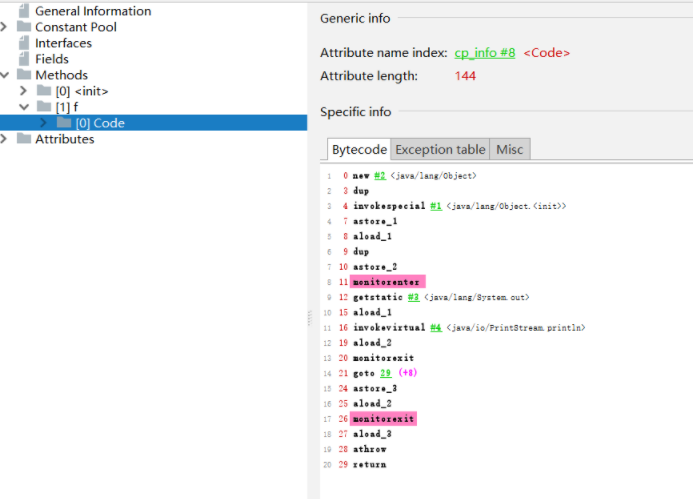
线程同步的代价是相当高的，同步的后果是降低并发性和性能。

在动态编译同步块的时候，JIT编译器可以借助逃逸分析来判断同步块所使用的锁对象是否只能够被一个线程访问而没有被发布到其他线程。如果没有，那么JIT编译器在编译这个同步块的时候就会取消对这部分代码的同步。这样就能大大提高并发性和性能。这个取消同步的过程就叫同步省略，也叫**锁消除。**



同步代码需要加锁，为不同线程的同步共享。但是要求所加的锁是同一个锁才有意义，上述同步代码块每次的锁对象hollos都是新new出来的，这并不符合加锁的同步效果。可以说是一个错误的代码。但能说明问题！

* 示例代码的字节码分析



字节码文件中并没有进行优化，可以看到加锁和释放锁的操作依然存在，同步省略操作是在解释运行时发生的

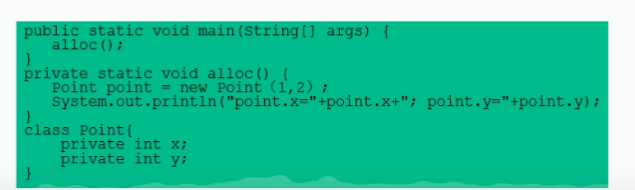
### 标量替换

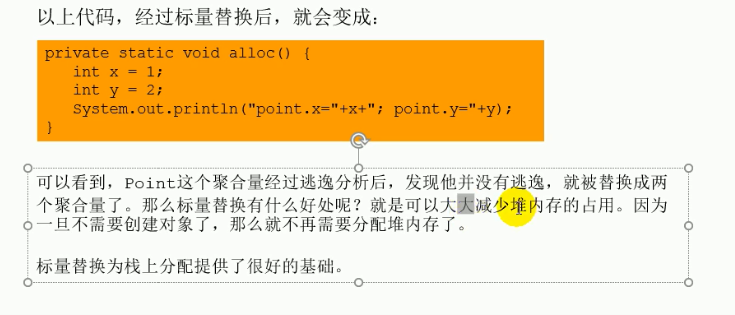
标量：无法被分解的最小的数据。Java中原始数据类型就是标量。

聚合量：可以被分解的就是聚合量。Java中对象就是聚合量。

**在JIT阶段**，经过逃逸分析发现一个对象不能逃逸的话，经过JIT优化，就会把这个对象拆解成各个标量，这个过程就是标量替换。

**如下所示：**alloc()方法中，可以将Point对象拆解成两个int标量。





可以看到，Point这个聚合量经过逃逸分析后，发现他并没有逃逸，就被替换成两个标量了。**那么标量替换有什么好处呢？**就是可以大大减少堆内存的占用。因为一旦不需要创建对象了，那么就不再需要分配堆内存了。  
 标量替换为栈上分配提供了很好的基础。

* **配置参数：-XX:+EliminateAllocations，开启标量替换。**
* **标量替换测试**

**测试代码：**

|  |
| --- |
| public class ScalarReplace {  public static class User {  public int id;//标量（无法再分解成更小的数据）  public String name;//聚合量（String还可以分解为char数组） }  public static void alloc() {  User u = new User();//未发生逃逸  u.id = 5;  u.name = "www.atguigu.com";  }  public static void main(String[] args) {  long start = System.currentTimeMillis();  for (int i = 0; i < 10000000; i++) {  alloc();  }  long end = System.currentTimeMillis();  System.out.println("花费的时间为： " + (end - start) + " ms");  }  } |

(1)关闭标量替换测试

VM设置：

-Xmx100m -Xms100m

-XX:+DoEscapeAnalysis

-XX:+PrintGC

-XX:-EliminateAllocations

代码执行结果：

|  |
| --- |
| [GC (Allocation Failure) 25600K->736K(98304K), 0.0199386 secs]  [GC (Allocation Failure) 26336K->736K(98304K), 0.0404454 secs]  [GC (Allocation Failure) 26336K->768K(98304K), 0.0254357 secs]  [GC (Allocation Failure) 26368K->720K(98304K), 0.0008551 secs]  [GC (Allocation Failure) 26320K->720K(98304K), 0.0212983 secs]  [GC (Allocation Failure) 26320K->720K(101376K), 0.0398378 secs]  [GC (Allocation Failure) 32464K->684K(101376K), 0.0382330 secs]  [GC (Allocation Failure) 32428K->684K(101376K), 0.0155815 secs]  花费的时间为： 286 ms |

(2)开启标量替换

VM设置：

-Xmx100m

-Xms100m

-XX:+DoEscapeAnalysis

-XX:+PrintGC

-XX:+EliminateAllocations

执行结果：

|  |
| --- |
| 花费的时间为： 10 ms |

可以测试对比一下，如果开启标量替换的性能提升，还是很巨大的。

### 逃逸分析总结

关于逃逸分析的论文在1999年就已经发表了，但直到JDK1.6才有实现，而且这项技术到如今也并不是十分成熟的。

其根本原因就是无法保证逃逸分析的性能消耗一定能高于他的消耗。虽然经过逃逸分析可以做标量替换、栈上分配、和锁消除。但是逃逸分析自身也是需要进行一系列复杂的分析的，这其实也是一个相对耗时的过程。

一个极端的例子，就是经过逃逸分析之后，发现没有一个对象是不逃逸的。那这个逃逸分析的过程就白白浪费掉了。

虽然这项技术并不十分成熟，但是它也是即时编译器优化技术中一个十分重要的手段。 注意到有一些观点，认为通过逃逸分析，JVM会在栈上分配那些不会逃逸的对象，这在理论上是可行的，但是取决于JVM设计者的选择。

**据我所知，Oracle HotspotJVM中并未这么做，这一点在逃逸分析相关的文档里已经说明，所以可以明确所有的对象实例都是创建在堆上。**回归本篇的提问：堆是分配对象的唯一选择么？一开始我们是否定的态度，现在我们又去肯定这一观点。这岂不是自相矛盾。。正所谓否定之否定的观点，通过这篇文章，我们还是有进步的，至少分离对象或标量替换这一点还是存在的，优化了代码运行的效率，减少了堆空间的占用，体现了栈上分配的思想。

目前很多书籍还是基于JDK7以前的版本，JDK已经发生了很大变化，intern字符串的缓存和静态变量曾经都被分配在永久代上，而永久代已经被元数据区取代。但是，intern字符串缓存和静态变量并不是被转移到元数据区，而是直接在堆上分配，所以这一点同样符合前面一点的结论：对象实例都是分配在堆上。

年轻代是对象的诞生、生长、消亡的区域，一个对象在这里产生、应用、最后被垃圾回收器收集、结束生命

老年代放置长生命周期对象，通常都是从Survivor区域筛选拷贝过来的Java对象。当然，也有特殊情况，我们知道普通的对象会被分配在TLAB上，如果对象较大，JVM会试图直接分配在Eden其他位置上；如果对象太大，完全无法在新生代找到足够长的连续空闲空间，JVM就会直接分配到老年代

当GC只发生在年轻代中，回收年轻对象的行为被称为MinorGC。当GC发生在老年代时则被称为MajorGC或者FullGC。一般的，MinorGC的发生频率要比MajorGC高很多，即老年代中垃圾回收发生的频率大大低于年轻代

逃逸分析只作用于服务器端JVM -server

**Oracle HotSpot JVM 没有应用栈上分配，而是用标量替换来提高性能，所以最终还是认为对象是分配在堆上的。**